

TURING

图灵程序  
设计丛书

# SQL基础教程

【日】MICK 著

孙淼 罗勇 译



SQL菜鸟晋级必备

资深数据库工程师总结的实用宝典

72张图表 + 186段代码，明示各RDBMS的异同

**亚马逊五星畅销书!**



人民邮电出版社  
POSTS & TELECOM PRESS



# SQL基础教程

## 亚马逊读者评论:

- 资深数据库工程师MICK给初学者的完美教科书，期待已久。现在市面上针对初学者的SQL书不少，但它们或偏重于讲解某一数据库，或偏学术，讲解晦涩，对实际工作帮助不大。而本书提供了大量的示例程序和通俗易懂的操作步骤说明，可谓是最实用、最现代的SQL指南。
- 果真最适合零基础的初学者。字里行间都能感受到作者的谆谆苦心。各类RDBMS都有涉及，讲解通俗易懂、实例丰富、涵盖面广。可谓是“不偏科的SQL教材”。
- 涵盖作者个人主页上介绍的SQL实用技巧，我边学边按照书中的介绍，自己安装PostgreSQL，获益匪浅。我相信读完本书后，个人能力会更上一层楼。



自在  
书籍设计  
8372037@163.com

图灵社区: [www.ituring.com.cn](http://www.ituring.com.cn)  
新浪微博: @图灵教育 @图灵社区  
反馈/投稿/推荐信箱: [contact@turingbook.com](mailto:contact@turingbook.com)  
热线: (010)51095186转604

分类建议 计算机/数据库/SQL

人民邮电出版社网址: [www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-32269-2



9 787115 322692 >

ISBN 978-7-115-32269-2

定价: 69.00元(附光盘)



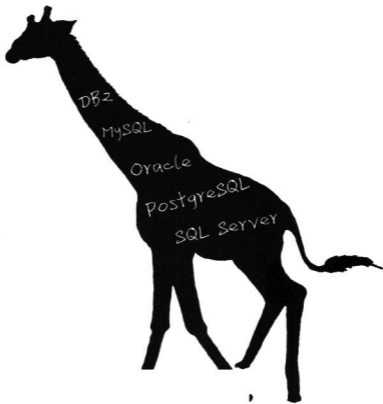
TURING

图灵程序  
设计丛书

# SQL基础教程

【日】MICK 著

孙淼 罗勇 译



人民邮电出版社  
北京

## 图书在版编目(CIP)数据

SQL基础教程 / (日) MICK 著; 孙森, 罗勇译. —

北京: 人民邮电出版社, 2013.8

(图灵程序设计丛书)

ISBN 978-7-115-32269-2

I. ①S… II. ①M… ②孙… ③罗… III. ①关系数据库系统 IV. ①TP311.138

中国版本图书馆CIP数据核字(2013)第133316号

SQLゼロからはじめるデータベース操作 (ISBN 978-4-7981-1881-9)

Copyright©2010 by Mick.

Original Japanese edition published by SHOEISHA Co.,Ltd

Simplified Chinese Character translation rights arranged with SHOEISHA Co.,Ltd

through CREEK & RIVER Co.,Ltd

Simplified Chinese Character translation copyright©2013 by Posts & Telecom Press

本书中文简体字版由SHOEISHA Co.,Ltd授权人民邮电出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

## 内 容 提 要

本书介绍了关系数据库以及用来操作关系数据库的SQL语言的使用方法,提供了大量的示例程序和详实的操作步骤说明,读者可以亲自动手解决具体问题,循序渐进地掌握SQL的基础知识和技巧,切实提高自身的编程能力。在每章结尾备有习题,用来检验读者对该章内容的理解程度。另外本书还将重要知识点总结为“法则”,方便大家随时查阅。

本书适合完全没有或者具备较少编程和系统开发经验的初学者,也可以作为大中专院校的教材及企业新人的培训用书。

- 
- ◆ 著 [日] MICK
  - 译 孙 森 罗 勇
  - 责任编辑 乐 馨
  - 执行编辑 徐 骞
  - 责任印制 焦志炜
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
  - 邮编 100061 电子邮件 315@ptpress.com.cn
  - 网址 <http://www.ptpress.com.cn>
  - 北京天宇星印刷厂印刷
  - ◆ 开本: 800×1000 1/16
  - 印张: 19.25
  - 字数: 455千字 2013年8月第1版
  - 印数: 1-3500册 2013年8月北京第1次印刷
  - 著作权合同登记号 图字: 01-2012-7099号

---

定价: 69.00元(附光盘)

读者服务热线: (010)51095186 转 604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

广告经营许可证: 京崇工商广字第0021号

---

## 前 言

本书面向完全没有编程和系统开发经验的初学者，介绍了关系数据库以及用来操作关系数据库的 SQL 语言的使用方法。各个章节结合具体示例进行解说，并在每章的结尾安排了习题，用来检验读者对该章内容的理解程度。大家可以从第 1 章开始，亲自验证示例程序，循序渐进地掌握 SQL 的基础知识和技巧。另外，本书还将重要知识点总结为法则，方便读者在学习完本书之后随时查阅。

近年来，和其他系统领域一样，数据库领域也实现了飞速发展，应用范围不断扩大。不但出现了具有新功能的数据库，而且操作的数据量也大幅增长。

本书将要介绍的关系数据库是时下最流行的数据库，也是理解其他数据库的基础。在系统领域，通常所讲的数据库指的就是关系数据库，其重要性可见一斑。

估计很多读者今后都会慢慢积累各个领域、各种规模的系统开发经验（或者可能已经开始从事开发方面的工作了）。到那时，所有的系统必定都需要使用数据库。它们使用的数据库，即便不是关系数据库，也一定是以关系数据库为基础的数据库。从这个意义上看，如果掌握了关系数据库和 SQL，就能成为任何系统开发都需要的数据库专家了。

本书旨在把数据库领域的精彩展示给大家，衷心希望本书能为大家的进步提供一些帮助。

---

## 关于本书

本书是编程学习系列的 SQL 和关系数据库篇。该系列注重对初学者编程能力的培养，本书秉承了这一宗旨。本书不仅可以用于自学，也可以作为大学、专科学校和企业新人的培训用书。这里提供了大量的示例程序和详实的操作步骤说明，读者可以亲自动手解决具体的问题，切实提高自身的编程能力。

另外，在各章的结尾处还安排了习题来帮助大家复习该章的知识要点。习题的答案和讲解收录在附录 C 中，大家可以验证自己的学习成果。

---

## 读者对象

- 不了解数据库和 SQL 知识的人
- 虽然自学了一些 SQL 知识，但仍希望进行系统学习的人
- 需要使用数据库，但不知道从何入手的人
- 在大学、专科学校和企业的教育部门等从事数据库和 SQL 教学的人
- 希望了解信息处理考试中 SQL 部分应试策略的人

---

## 学习本书前的预备知识

- 了解 Windows 的基本操作方法
- 能够使用 Windows 的资源管理器创建文件夹并进行文件复制
- 能够使用 Windows 的记事本（或者其他文本编辑器）创建文本文件

---

## 本书涉及的关系数据库

本书中使用的 SQL 语句全部都在下列关系数据库系统（RDBMS）中进行了验证。

- Oracle Database 11g
- SQL Server 2008
- DB2 9.7
- PostgreSQL 8.4
- MySQL 5.5

在这 5 种数据库之间存在差异的 SQL 语句，或者只能在某种特定的 RDBMS 中使用的 SQL 语句，本书都用下列图标进行标识，用来提示 SQL 语句执行所使用的 RDBMS。



反之，在所有 RDBMS 中都能正常执行的 SQL 语句则不用图标标识。

---

## 本书的学习安排

首先，在第 1 章前半部分学习关系数据库和 SQL 的基础知识，然后结合具体的 SQL 示例程序再进行循序渐进的学习。

在 SQL 学习方面，最重要的就是以下两点：

- 亲自编写 SQL 语句
- 通过执行 SQL 语句来学习和理解数据库操作

要提高学习效率，则需尽量亲自执行并验证本书中的示例程序，逐步深入学习。

随书光盘中收录了 SQL 的学习环境——PostgreSQL。在开始学习之前，请在自己的电脑上安装该数据库，为执行 SQL 语句做好准备。在附录 A 和附录 B 中总结了该数据库的安

装和 SQL 语句的执行方法。

如果你已经安装了前述“本书涉及的关系数据库”中记载的数据库，也可以直接使用。

另外，如无特殊说明，本书中记述的 SQL 语句的执行结果，都是在 PostgreSQL 8.4 中执行的结果。

●编程学习系列《SQL基础教程》的教辅网站

[http://www.geocities.jp/mickindex/database/db\\_support\\_prg\\_sql.html](http://www.geocities.jp/mickindex/database/db_support_prg_sql.html)

---

## 随书光盘简介

附赠的随书光盘中收录了本书使用的示例程序和 SQL 的学习环境 (PostgreSQL)。

随书光盘的文件结构如下所示：



### ReadMe.txt 文件

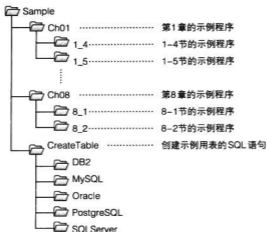
介绍了随书光盘的内容和要点，使用前请务必阅读该文件。

### PostgreSQL\_Installer 文件夹

你可以很容易地将 PostgreSQL 安装在 Windows 系统的电脑上。PostgreSQL 的安装方法以及在 PostgreSQL 中录入和执行 SQL 语句的方法，请参考附录 A 和附录 B。

### Sample 文件夹

本书中所使用的示例程序分别保存在以章节为单位的文件夹中。在 Sample\CreateTable 文件夹中，按照 RDBMS 的不同，分别保存了用来创建示例用表的 SQL 语句。



### answer 文件夹

各章末习题的答案（示例程序），分别保存在以章为单位的目录中。

## 关于示例程序

随书光盘中收录的示例程序的文件名，与书中记述的列表号码相对应。例如，1-5节记述的代码清单 1-3 的示例程序，保存的位置和文件名如下所示。



另外，像如下代码清单这样，在不同的 RDBMS 中存在差异的 SQL 语句，会在其文件名的末尾加上 RDBMS 的名称。

代码清单 1-4 增加一个可以保存 100 位可变长度字符串的列 shohin\_mei\_kana

**DB2 PostgreSQL MySQL**

```
ALTER TABLE Shohin ADD COLUMN shohin_mei_kana VARCHAR(100);
```

**Oracle**

```
ALTER TABLE Shohin ADD (shohin_mei_kana VARCHAR(100));
```

**SQL Server**

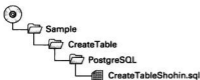
```
ALTER TABLE Shohin ADD shohin_mei_kana VARCHAR(100);
```

这种情况下，示例程序的文件名如下所示。

- List1\_4\_DB2\_PostgreSQL\_MySQL.sql
- List1\_4\_Oracle.sql
- List1\_4\_SQL Server.sql

### 创建示例用表的 SQL 语句

创建示例用表的 SQL 文件保存在 Sample/CreateTable 文件夹中。文件名为 CreateTable 表名.sql。例如，PostgreSQL 用到的表 Shohin 保存在下述目录中。



保存在 Sample 文件夹中的示例程序文件，可以使用 Windows 的记事本（或者其他文本编辑器）打开。

---

## 声明

翔泳有限公司

随书光盘中的文件已经过编辑部确认，在正常使用时不会出现任何问题。对于文件执行结果所造成的任何损失，本书作者、软件开发人员和翔泳公司概不承担相关责任。

随书光盘 Sample 文件夹中所收录文件的著作权归本书作者所有。读者可以出于个人目的，根据需要自行使用和修改其中的程序。

对于个别环境相关的问题，以及由本书对应范围外的环境设置所造成的执行错误，本公司概不负责解答。



## 目 录

前 言	III
关于本书	IV
读者对象	IV
学习本书前的预备知识	IV
本书涉及的关系数据库	V
本书的学习安排	V
随书光盘简介	VI
<b>第 1 章 数据库和 SQL</b>	<b>1</b>
1-1 数据库是什么	3
我们身边的数据库	3
为什么 DBMS 那么重要	4
DBMS 种类	6
1-2 数据库的结构	8
RDBMS 的常见系统结构	8
表的结构	10
1-3 SQL 概要	13
标准 SQL	13
SQL 语句及其种类	14
SQL 的基本书写规则	15
1-4 表的创建	18
表的内容的创建	18
数据库的创建 (CREATE DATABASE 语句)	19
表的创建 (CREATE TABLE 语句)	19
命名规则	21
数据类型的指定	22
约束的设置	24
1-5 表的删除和更新	25
表的删除 (DROP TABLE 语句)	25
表定义的更新 (ALTER TABLE 语句)	26
向 Shohin 表中插入数据	27
练习题	30

第2章 查询基础	31
2-1 SELECT 语句基础	33
列的查询	33
查询出表中所有的列	35
为列设定别名	36
常数的查询	38
从结果中删除重复行	38
根据 WHERE 语句来选择记录	41
注释的书写方法	43
2-2 算术运算符和比较运算符	45
算术运算符	45
需要注意 NULL	46
比较运算符	48
对字符串使用不等号时的注意事项	50
不能对 NULL 使用比较运算符	53
2-3 逻辑运算符	56
NOT 运算符	56
AND 运算符和 OR 运算符	58
通过括号进行强化	60
逻辑运算符和真值	62
含有 NULL 时的真值	64
练习题	66
第3章 聚合与排序	67
3-1 对表进行聚合查询	69
聚合函数	69
计算表中数据的行数	70
计算 NULL 以外数据的行数	71
计算合计值	72
计算平均值	74
计算最大值和最小值	75

	使用聚合函数删除重复值 (关键字 DISTINCT) .....	77
3-2	对表进行分组 .....	79
	GROUP BY 子句 .....	79
	聚合键中包含 NULL 的情况 .....	81
	使用 WHERE 子句时 GROUP BY 的执行结果 .....	82
	与聚合函数和 GROUP BY 子句有关的常见错误 .....	84
3-3	为聚合结果指定条件 .....	89
	HAVING 子句 .....	89
	HAVING 子句的构成要素 .....	92
	相对于 HAVING 子句, 更适合写在 WHERE 子句中的条件 .....	93
3-4	对查询结果进行排序 .....	96
	ORDER BY 子句 .....	96
	指定升序或降序 .....	98
	指定多个排序键 .....	99
	NULL 的顺序 .....	99
	在排序键中使用显示用别名 .....	100
	ORDER BY 子句中可以使用的列 .....	102
	不要使用列编号 .....	102
	练习题 .....	104
第 4 章 数据更新 .....		105
4-1	数据的插入 (INSERT 语句的使用方法) .....	107
	什么是 INSERT .....	107
	INSERT 语句的基本语法 .....	108
	清单的省略 .....	111
	插入 NULL .....	111
	插入默认值 .....	112
	从其他表中复制数据 .....	114
4-2	数据的删除 (DELETE 语句的使用方法) .....	117
	DROP TABLE 语句和 DELETE 语句 .....	117
	DELETE 语句的基本语法 .....	117

	指定删除对象的DELETE语句(搜索型DELETE).....	118
4-3	数据的更新(UPDATE语句的使用方法).....	121
	UPDATE语句的基本语法.....	121
	指定条件的UPDATE语句(搜索型UPDATE).....	122
	使用NULL进行更新.....	123
	多列更新.....	124
4-4	事务.....	126
	什么是事务.....	126
	创建事务.....	127
	ACID特性.....	132
	练习题.....	133
第5章	复杂查询.....	135
5-1	视图.....	137
	视图和表.....	137
	创建视图的方法.....	139
	视图的限制①——定义视图时不能使用ORDER BY子句.....	142
	视图的限制②——对视图进行更新.....	143
	删除视图.....	147
5-2	子查询.....	148
	子查询和视图.....	148
	子查询的名称.....	151
	标量子查询.....	151
	标量子查询的书写位置.....	154
	使用标量子查询时的注意事项.....	155
5-3	关联子查询.....	156
	普通的子查询和关联子查询的区别.....	156
	关联子查询也是用来对集合进行切分的.....	159
	结合条件一定要写在子查询中.....	160
	练习题.....	161

第6章 函数、谓词、CASE表达式	163
6-1 各种各样的函数	165
函数的种类	165
算术函数	166
字符串函数	170
日期函数	178
转换函数	182
6-2 谓词	186
什么是谓词	186
LIKE谓词——字符串的部分一致查询	186
BETWEEN谓词——范围查询	190
IS NULL、IS NOT NULL——判断是否为NULL	191
IN谓词——OR的简便用法	192
使用子查询作为IN谓词的参数	193
EXIST谓词	198
6-3 CASE表达式	202
什么是CASE表达式	202
CASE表达式的语法	202
CASE表达式的使用方法	203
练习题	209
第7章 集合运算	211
7-1 表的加减法	213
什么是集合运算	213
表的加法——UNION	213
集合运算的注意事项	216
包含重复行的集合运算——ALL选项	217
选取表中公共部分——INTERSECT	218
记录的减法——EXCEPT	219
7-2 联结(以列为单位对表进行联结)	222
什么是联结	222

---

内联结——INNER JOIN	223
外联结——OUTER JOIN	228
3张以上表的联结	231
交叉联结——CROSS JOIN	234
特定的联结语句和过时的语法	237
练习题	242
<b>第8章 SQL高级处理</b>	<b>243</b>
<b>8-1 窗口函数</b>	<b>245</b>
什么是窗口函数	245
窗口函数的语法	246
语法的基本使用方法——使用RANK函数	246
无须指定PARTITION BY	249
专用窗口函数的种类	250
窗口函数的适用范围	251
作为窗口函数使用的聚合函数	252
计算移动平均	254
两个ORDER BY	257
<b>8-2 GROUPING运算符</b>	<b>259</b>
同时计算出合计值	259
ROLLUP——同时计算出合计值和小计值	260
GROUPING函数——让NULL更加容易分辨	265
CUBE——用数据来搭积木	267
GROUPING SETS——取得期望的积木	269
练习题	270
附录A 安装PostgreSQL	271
附录B 在PostgreSQL中执行SQL的方法	278
附录C 练习题答案	284

# 第1章 数据库和SQL

数据库是什么

数据库的结构

SQL概要

表的创建

表的删除和更新

# 1



SQL

## 本章重点

---

本章介绍了数据库的构成和基本理论，以及数据库的实际应用。大家可以学习到如何对关系数据库中用来存储数据的表进行创建、更新和删除操作，同时还能掌握关系数据库专用的 SQL 语句的书写方法和规则。

### 1-1 数据库是什么

- 我们身边的数据库
- 为什么 DBMS 那么重要
- DBMS 种类

### 1-2 数据库的结构

- RDBMS 的常见系统结构
- 表的结构

### 1-3 SQL 概要

- 标准 SQL
- SQL 语句及其种类
- SQL 的基本书写规则

### 1-4 表的创建

- 表的内容的创建
- 数据库的创建 (CREATE DATABASE 语句)
- 表的创建 (CREATE TABLE 语句)
- 命名规则
- 数据类型的指定
- 约束的设置

### 1-5 表的删除和更新

- 表的删除 (DROP TABLE 语句)
- 表定义的更新 (ALTER TABLE 语句)
- 向 Shohin 表中插入数据



# 1-1

## 第1章 数据库和SQL

# 数据库是什么

### 学习重点

- 数据库是将大量数据保存起来，通过计算机加工而成的可以进行高效访问的数据集合。
- 用来管理数据库的计算机系统称为数据库管理系统 (DBMS)。
- 通过使用DBMS，多个用户便可安全、简单地操作大量数据。
- 数据库有很多种类，本书将介绍如何使用专门的SQL语言来操作关系型数据库。
- 关系型数据库通过关系型数据库管理系统 (RDBMS) 进行管理。

## 我们身边的数据库

大家都有过下面这样的经历吧？

- 收到曾经为自己诊治过的牙医寄来的明信片，上面写着“距上次检查已有半年，请您再来做个牙齿健康检查”。
- 在生日的前一个月，收到曾入住过的旅店或宾馆发来的“生日当月入住优惠”的邮件或者明信片。
- 在网上商城购物之后，收到内附“推荐商品列表”的邮件。

这可能是由于牙医、旅店或商城的经营者掌握了顾客上一次的就诊日期、生日和购买历史等信息，并且拥有能够从大量汇总信息中快速获取所需信息（比如你的住址或爱好）的设备（计算机系统）。如果利用人工完成同样的工作，真不知道要多长时间呢。

另外，现在所有地区的图书馆都配备了计算机，实现了书籍的自动查询。使用该系统，可以通过检索书名或出版年份，快速查找出希望借阅的图书的所在位置，以及是否已经借出等信息。正是因为拥有了可以保存书籍名称、出版年份以及保管位置和外借状况等信息，并且可以按需查询的设备，才使这一切成为可能。

像这样将大量数据保存起来，通过计算机加工而成的可以进行高效

### KEYWORD

- 数据
- 数据库 (DB)
- 数据库管理系统 (DBMS)

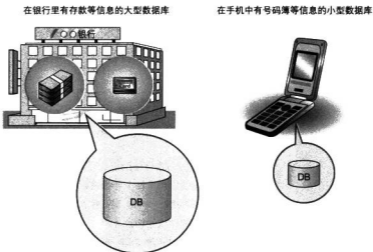
访问的数据集合称为数据库 (Database, DB)。将姓名、住址、电话号码、邮件地址、爱好和家庭构成等数据保存到数据库中, 就可以随时迅速获取想要的信息了。用来管理数据库的计算机系统称为数据库管理系统 (Database Management System, DBMS)<sup>①</sup>。

## ①

数据库 (DB) 和 DBMS 经常被混淆。为了加以区别, 本书将数据库管理系统统称为 DBMS。

系统的使用者通常无法直接接触到数据库。因此, 在使用系统的时候往往意识不到数据库的存在。其实大到银行账户的管理, 小到手机的号码簿, 可以说社会的所有系统中都有数据库的身影 (图 1-1)。

图 1-1 数据库无处不在



## 为什么 DBMS 那么重要

那么, 为什么要使用专用系统 (DBMS) 来管理数据呢。我们通过计算机管理数据的时候, 通常都是使用文本文件<sup>②</sup>或者 Excel 那样的电子制表软件就可以完成了, 非常简单。

## ②

保存只通过文字记录的数据的文件。

确实, 通过文本文件或者电子制表软件来管理数据的方法非常简便, 但也有不足。下面就举几个有代表性的例子。

### ●无法多人共享数据

保存在已连接网络的计算机中的文件, 可以通过共享设定实现多个用

户在线阅读或编辑。但是，当某个用户打开该文件的时候，其他用户就无法进行编辑了。如果是网上商城的话，当某个用户购买商品的时候，其他用户就无法购买了。

#### ●无法提供操作大量数据所需的格式

瞬间从几十万或者上百万的数据中获取想要的的数据，必须把数据保存为适当的格式。但是文本文件和 Excel 工作表等无法提供相应的数据格式。

#### ●实现读写自动化需要编程技术能力

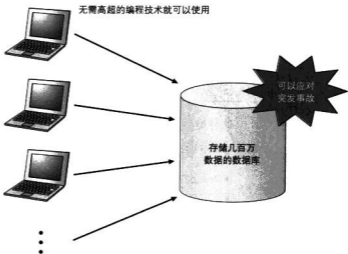
通过编写计算机程序（以下简称程序）可以实现数据读取和编辑自动化，但这必须以掌握数据结构为前提，还需具备一定的计算机编程技术水平。

#### ●无法应对突发事件

当文件被误删、硬盘故障无法读取的时候，可能会造成重要数据的丢失。同时数据还可能被他人简单地读取或窃用。

DBMS 可以克服这些不足，能够实现多个用户同时安全简单地操作大量数据（图 1-2）。这也是我们一定要使用 DBMS 的原因。

图 1-2 DBMS 能够实现多个用户同时安全简单地操作大量数据



## DBMS 种类

虽然都称为 DBMS，但它们属于不同类型。DBMS 主要是通过数据的保存格式（数据库种类）来进行分类的，现阶段主要有以下 5 种。

### KEYWORD

- 层次型数据库
- 关系型数据库 (RDB)
- 关系数据库
- SQL

### ● 层次型数据库 ( Hierarchical Database, HDB )

最古老的数据库之一，它把数据通过层次结构（树结构）的方式表现出来。层次型数据库曾经是数据库的主流，但随着关系型数据库的出现和普及，现在已经很少使用了。

### ● 关系型数据库 ( Relational Database, RDB )

也称为关系数据库，是现在应用最广泛的数据库。关系型数据库 1969 年诞生，可谓历史悠久。和 Excel 工作表一样，它也采用行列二维表结构来管理数据，所以简单易懂（表 1-1）。同时，它还使用专门的 SQL（Structured Query Language，结构化查询语言）语言对数据进行操作。

表 1-1 关系数据库中的数据

商品编号	商品名称	商品分类	销售单价	进货单价	登记日期
0001	T 恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动 T 恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	煎菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

这种类型的 DBMS 称为关系数据库管理系统（Relational Database Management System, RDBMS）。比较具有代表性的 RDBMS 有如下五种：

- Oracle Database：甲骨文公司的 RDBMS
- SQL Server：微软公司的 RDBMS
- DB2：IBM 公司的 RDBMS
- PostgreSQL：开源的 RDBMS
- MySQL：开源的 RDBMS

### KEYWORD

- RDBMS：
- 开源

将软件的内容（代码）无偿地公开在互联网上，任何人都可以进行修改并再次发布。开发项目可以将志同道合的有志之士集中起来运营。

另外，Oracle Database 通常简称为 Oracle，因此，本书在接下来的章节中也使用这一简称。

#### KEYWORD

- 面向对象数据库 (OODB)

#### ①

主要的面向对象语言包括 Java 和 C++ 等。

#### KEYWORD

- XML 数据库 (XMLDB)

#### ②

extensible Markup Language 的缩写，使用 HTML 那样的标签来表现数据结构的一种语言。以 <name> 符号 </name> 这样的形式来保存数据。

#### KEYWORD

- 键值存储系统 (KVS)

#### ● 面向对象数据库 (Object Oriented Database, OODB)

编程语言当中有一种被称为面向对象的语言<sup>①</sup>。该类数据库把数据以及对数据的操作集合起来以对象为单位进行管理，因此得名。面向对象数据库就是用来保存这些对象的数据库。

#### ● XML 数据库 (XML Database, XMLDB)

最近几年，XML<sup>②</sup> 作为在网络上进行数据交互传输的形式逐渐普及起来。XML 数据库可以对 XML 形式的大量数据进行高速处理。

#### ● 键值存储系统 (Key-Value Store, KVS)

这是一种单纯用来保存查询所使用的主键 (Key) 和值 (Value) 的组合的数据库。具有编程语言经验的读者可以把它想象成关联数组或者是散列 (hash)。近年来，随着键值存储系统被应用到 Google 等需要对大量数据进行超高速查询的 Web 服务当中，它正逐渐为人们所关注。

本书将向大家介绍使用 SQL 语言的数据库管理系统，也就是关系数据库管理系统 (RDBMS) 的操作方法。接下来还会深入讲解 RDBMS。如无特殊说明，本书所提到的数据库以及 DBMS 都是指 RDBMS。

另外，RDBMS 也可以像 XML 数据库那样操作 XML 形式的数据，并且具有面向对象数据库的功能。本书并不会介绍使用这些扩展功能的 SQL，如果要了解这些内容，请参考 RDBMS 附带的 SQL 手册或者其他介绍 SQL 的 RDBMS 书籍。

## 1-2

## 数据库的结构

## 学习重点

- RDBMS 通常使用客户端/服务器这样的系统结构。
- 通过从客户端向服务器端发送 SQL 语句来实现数据库的读写操作。
- 关系数据库采用被称为数据库表的二维表来管理数据。
- 数据库表由表示数据项目的列(字段)和表示一条数据的行(记录)所组成,以记录为单位进行数据读写。
- 本书将行和列交汇的方格称为单元格,每个单元格只能输入一个数据。

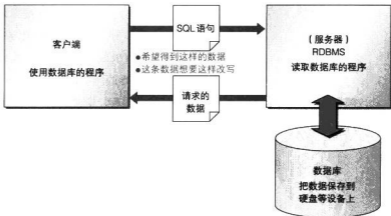
## RDBMS 的常见系统结构

使用 RDBMS 时,最常见的系统结构就是客户端/服务器(C/S 类型)这种类型(图 1-3)。

## KEYWORD

- 客户端/服务器类型(C/S 类型)

图 1-3 使用 RDBMS 时的系统结构



## KEYWORD

- 服务器
- 数据库
- 客户端

服务器指的是用来接收其他程序发出的请求,并对该请求进行相应处理的程序(软件),或者是安装了此类程序的设备(计算机)。计算机将继续执行后续处理,等待接收下一条请求。RDBMS 也是一种服务器,它能

**KEYWORD**

## ● SQL 语句

够从保存在硬盘上的数据库中读取数据并返回，还可以把数据变更为指定内容。

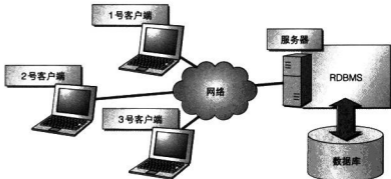
与之相对，向服务器发出请求的程序（软件），或者是安装了该程序的设备（计算机）称为客户端。访问由 RDBMS 管理的数据库，进行数据读写的程序称为 RDBMS 客户端。RDBMS 客户端将想要获取什么样的数据，或者想对哪些数据进行何种变更等信息通过 SQL 语句发送给 RDBMS 服务器。RDBMS 根据该语句的内容返回所请求的数据，或者对存储在数据库中的数据进行更新。

客户端就如同委托人，而服务器就像是服务员。由于两者关系类似服务员执行委托人发出的指令，故而得名。

这样就可以使用 SQL 语句来实现关系数据库的读写操作了。本书为了给大家讲解 SQL，使用了可以显示如何将 SQL 语句发送到 RDBMS，以及接收返回信息（数据）的客户端。具体内容请参考附录 B。

另外，RDBMS 既可以和其客户端同时安装在同一台计算机上，也可以分别安装在不同的计算机上。这样一来，通过网络不仅可以使二者相互关联，还可以实现多个客户端访问同一个 RDBMS（图 1-4）。

图 1-4 通过网络可以实现多个客户端访问同一个数据库



客户端没有必要使用同样的程序，只要能将 SQL 发送给 RDBMS，就可以操作数据库了。并且，多个客户端还可以同时对同一个数据库进行读写操作。

另外，RDBMS 除了需要同时接收多个客户端的请求之外，还需要操

作存有大量数据的数据库，因此通常都会安装在比客户端性能更优越的计算机上。操作数据量特别巨大的数据库时，还可以将多台计算机组合使用。

虽然 RDBMS 的系统结构多种多样，但是从客户端发来的 SQL 语句基本上都是一样的。

## 表的结构

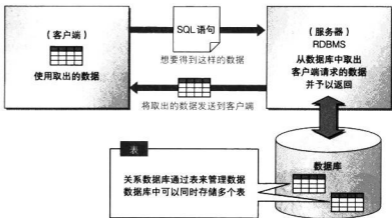
让我们再具体了解一下 RDBMS 的结构。上一节我们讲到了关系数据库通过类似 Excel 工作表那样的、由行和列组成的二维表来管理数据。用来管理数据的二维表在关系数据库中简称为表。

表存储在由 RDBMS 管理的数据库中，如图 1-5 所示。一个数据库中

### KEYWORD

- table
- 表

图 1-5 数据库和表的关系



根据 SQL 语句的内容返回的数据，同样必须是二维表的形式，这也是关系数据库的特征之一。执行结果如果不是二维表的 SQL 语句则无法执行。

另外，图 1-5 中只有一个数据库，我们还可以创建多个数据库分别用于不同用途。

图 1-6 展示了将会在第 3 节 SQL 学习中实际用到的商品表的内容。



图1-6 表的示例(商品表)

商品编号	商品名称	商品分类	销售单价	进货单价	登记日期
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	磨菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

## KEYWORD

- 列
- 字段
- 行
- 记录

表的列(垂直的方向)称为字段,它代表了保存在表中的数据项目。在表1-2的商品表中,从商品编号到登记日期一共有6列。对于列的约束比Excel更加严格,定义为数字的列只能输入数字,定义为日期的列只能输入日期(将在第4节进行详细介绍)。

与之相对,表的行(水平的方向)称为记录,它相当于一条数据。商品表中总共有8行数据。关系数据库必须以行为单位进行数据读写,请大家牢记。



## 法则1-1

关系数据库以行为单位读写数据。

## KEYWORD

- 单元格

单元格是本书特有的记述方式。实际上关系数据库对于行和列交汇的方格并没有专门的称谓。但就像图1-6显示的那样,这个方格通过类似Excel单元格的方式管理数据,因此把它称为单元格似乎也很恰当。

本书将图1-6所展示的行和列交汇的方格称为单元格。在一个单元格中只能输入一个数据。像图1-7那样,在一个单元格中输入2个或2个以上的数据是不允许的,请大家牢记。

图1-7 一个单元格中只能输入一个数据

商品编号	商品名称	商品分类	销售单价	进货单价	登记日期
0001	T恤衫 牛仔褲	衣服	1000	500	2009-09-20

如本表所示,一个单元格中无法输入两个数据



### 法则 1-2

一个单元格中只能输入一个数据。

## 专 栏

### RDBMS 的用户管理

为了防止重要数据被窃读或篡改，RDBMS 只允许注册用户接触数据库。这里的用户并不是指 Windows 等操作系统的注册用户，而是只能用于 RDBMS 的用户。RDBMS 允许注册多个用户。

注册用户的时候除了设定用户名（账号），还需要设定密码。虽然密码并不是必需的，但为了防止重要情报的泄露，还是希望大家能够设定密码。

## 1-3

## 第1章 数据库和SQL

## SQL概要

## 学习要点

- SQL是为操作数据库而开发的语言。
- 虽然SQL也有标准,但实际上根据RDBMS的不同SQL也不尽相同。
- SQL通过一条语句来描述想要进行的操作,发送给RDBMS。
- 原则上SQL语句都会使用分号结尾。
- SQL根据操作目的可以分为DDL、DML和DCL。

## KEYWORD

- SQL

## KEYWORD

- 标准SQL

## ❶

本书将介绍以[SQL:2003]为基准的标准SQL的书写方式。

## 标准SQL

如前所述,本书所要学习的SQL是用来操作关系数据库的语言。它原本是为了提高数据库查询效率而开发的语言,但是现在不仅可以进行数据查询,就连数据的插入和删除等操作也基本上都可以通过SQL来完成了。

国际标准化组织(ISO)为SQL制定了相应的标准,以此为基准的SQL称为标准SQL(相关信息请参考专栏——标准的SQL和特定的SQL)。以前,完全基于标准SQL的RDBMS很少,通常需要根据不同的RDBMS来编写特定的SQL语句。这样一来,就会造成能够在Oracle中使用的SQL语句却无法在SQL Server中使用,反之亦然。近来,对标准SQL的支持取得了一些进展,因此希望准备学习SQL的读者们能够从现在开始就牢记标准SQL的书写方式。

原则上,本书介绍的都是标准SQL<sup>❶</sup>的书写方式,但是根据RDBMS的不同也会存在一些特殊的SQL语句。如果遇到这种情况,将会通过其他途径对其进行说明。



## 法则 1-3

学会标准的SQL就可以在各种RDBMS中书写SQL语句了。

---

## SQL 语句及其种类

### KEYWORD

- 关键字

SQL 用关键字、表名、列名等组合而成的一条语句 (SQL 语句) 来描述操作的内容。关键字是指那些含义或使用方法事先已定义好的英语单词, 例如“对表进行查询”或者“参考这个表”等包含各种意义的关键词。

根据对 RDBMS 赋予的指令种类的不同, SQL 语句可以分为以下三类。

### ● DDL

### KEYWORD

- DDL (数据定义语言)

DDL (Data Definition Language, 数据定义语言) 用来创建或者删除存储数据用的数据库以及数据库中的表等对象。DDL 包含以下几种指令。

CREATE: 创建数据库和表等对象  
DROP: 删除数据库和表等对象  
ALTER: 修改数据库和表等对象的结构

### ● DML

### KEYWORD

- DML (数据操纵语言)

DML (Data Manipulation Language, 数据操作语言) 用来查询或者变更表中的记录。DML 包含以下几种指令。

SELECT: 查询表中的数据  
INSERT: 向表中插入新数据  
UPDATE: 变更表中的数据  
DELETE: 删除表中的数据

### ● DCL

### KEYWORD

- DCL (数据控制语言)

DCL (Data Control Language, 数据控制语言) 用来确认或者取消对数据库中的数据进行的变更。除此之外, 还可以对 RDBMS 的用户是否有权限操作数据库中的对象 (数据库表等) 进行设定。DCL 包含以下几种指令。

COMMIT: 确认对数据库中的数据进行的变更  
ROLLBACK: 取消对数据库中的数据进行的变更  
GRANT: 赋予用户操作权限  
REVOKE: 取消用户的操作权限

实际使用的 SQL 语句当中有 90% 属于 DML, 本书同样会以 DML 为中心进行讲解。



## 法则 1-4

SQL 根据功能不同可以分为三类，其中使用最多的是 DML。

## SQL 的基本书写规则

书写 SQL 语句时必须遵守一些规则。这些规则都非常简单，接下来就让我们逐一认识一下吧。

### ■ SQL 语句要以分号 (;) 结尾

一条 SQL 语句可以描述一个数据库操作。在 RDBMS 当中，SQL 语句也是逐一执行的。

众所周知，我们在句子的句尾加注标点表示结束，汉语句子以句号 (。) 结尾，英语以点号 (.) 结尾，而 SQL 语句则使用分号 (;) 结尾。

#### KEYWORD

●分号 (;)



## 法则 1-5

SQL 语句以分号 (;) 结尾。

### ■ SQL 语句不区分大小写

SQL 不区分关键字的大小写。例如，不管写成 `SELECT` 还是 `select`，解释都是一样的。表名和列名也是如此。

虽然可以根据个人喜好选择大写还是小写（或大小写混杂），但为了理解起来更加容易，本书使用以下规则来书写 SQL 语句。

- 关键字大写
- 表名的首个字大写
- 其余（列名等）小写



## 法则 1-6

关键字不区分大小写。

但是插入到表中的数据是区分大小写的。例如，在操作过程中，数据 `Computer`、`COMPUTER` 或 `computer`，三者是不一样的。

## 0

一个以上的连续字符。

## KEYWORD

- 常数
- 单引号(')

### ■常数的书写方式是固定的

SQL 语句常常需要直接书写字符串<sup>①</sup>、日期或者数字。例如，书写向表中插入字符串、日期或者数字等数据的 SQL 语句。

在 SQL 语句中直接书写的字符串、日期或者数字等称为常数。常数的书写方式如下所示。

SQL 语句中含有字符串的时候，需要像 'abc' 这样，使用单引号（'）将字符串括起来，用来标识这是一个字符串。

SQL 语句中含有日期的时候，同样需要使用单引号将其括起来。日期的格式有很多种（'26 Jan 2010' 或者 '10/01/26' 等），本书统一使用 '2010-01-26' 这种 '年-月-日' 的格式。

在 SQL 语句中书写数字的时候，不需要使用任何记号标识，直接写成 1000 这样的数字即可。



#### 法则 1-7

字符串和日期常数需要使用单引号（'）括起来。  
数字常数无需加单引号（直接书写数字即可）。

### ■单词需要用半角空格或者换行来分隔

SQL 语句的单词之间需使用半角空格或换行符来进行分隔。如下这种未加分隔的语句会发生错误，无法正常执行。

- CREATE TABLE Shohin
- × CREATETABLE Shohin
- × CREATE TABLEShohin

但是不能使用全角空格作为单词的分隔符，否则会发生错误，出现无法预期的结果。



#### 法则 1-8

单词之间需要使用半角空格或者换行进行分隔。

## KEYWORD

## ● 错误

由程序不匹配、故障、输入错误等多种原因造成的系统或者程序未按预定处理执行或者无法执行的情况。通常出错时，处理会被强制终止，并显示错误信息。

## 专 栏

### 标准的SQL和特定的SQL

每隔几年, ANSI (美国国家标准协会) 或 ISO (国际标准化组织) 等便会修订 SQL 的标准, 进行语法的修订并追加新功能。

1986年, ANSI 首先制定了 SQL 的标准, 之后又进行了数次修订。本书编写时使用的是 2008 年修订的最新版本 (SQL:2008)。修订后的标准以修订年份来命名, 例如 SQL:1999、SQL:2003、SQL:2008 等。以这些标准为基准的 SQL 就是标准 SQL。

但是, SQL 的标准并不强制“每种 RDBMS 都必须使用”。虽然支持标准 SQL 的 RDBMS 越来越多, 但还是存在标准 SQL 无法运行的情况。这时就需要使用只能在特定 RDBMS 中使用的特殊 SQL 语句。

其实, 这也是没有办法的事情, 起初 (大约 1980 年到 1990 年这段时间), 标准 SQL 能够实现的功能非常有限, 无法完全满足实际需要。RDBMS 的供应商为了弥补这些不足, 不得不再单独追加所需要的功能。

尽管如此, 这些特定的 SQL 所带来的并不都是负面的影响。标准 SQL 将一些独特的功能收录其中, 对其自身的发展起到了积极的推进作用。过去, 各个供应商为了展现本公司的优势和独特性, 也曾不遗余力地开发各自特定的 SQL。

目前的标准 SQL 经过多次修订, 功能已经十分完善。准备学习 SQL 的读者们, 就让我们先从牢记标准 SQL 的书写方法开始吧。

## 1-4

## 表的创建

## 学习要点

- 表通过 CREATE TABLE 语句创建而成。
- 表和列的命名要使用有意义的文字。
- 指定列的数据类型( 整数型、字符型和日期型等)。
- 可以在表中设置约束( 主键约束和 NOT NULL 约束等)。

## 表的内容的创建

我们将从第2章开始学习针对表的查询, 以及数据变更等 SQL 语句。本节将会创建学习这些 SQL 语句所必须的数据库和表。

1-2 节举例时使用的商品表如表 1-2 所示。

表 1-2 商品表

商品编号	商品名称	商品分类	销售单价	进货单价	登记日期
0001	T 恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T 恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

该表是某家小商店销售商品的一览表。商品的数量不多, 不过我们可以把它想象成大量数据中的一部分( 毕竟这只是为了学习 SQL 而创建的表)。像 0003 号商品的登记日期, 以及 0006 号商品的进货单价这样的空白内容, 我们可以认为是由于店主疏忽而忘记输入了。



大家可以看到表 1-2 由 6 列 8 行所组成。最上面一行是数据的项目名，真正的数据是从第 2 行开始的。

### 备忘

接下来，我们会逐步学习创建数据库和表所使用的 SQL 语句的书写方式。大家可以按照附录 A·B 的内容安装随书光盘中附带的 SQL 学习环境（PostgreSQL）来执行 SQL 语句。

## 数据库的创建 (CREATE DATABASE 语句)

### KEYWORD

● CREATE DATABASE 语句

#### ①

这里我们仅指定了使用该语法所需的最少项目，实际开发数据库时还需要指定各种其他项目。

#### ②

附录 B 中介绍了在 PostgreSQL 中运行 SQL 语句的方法。执行了附录 B 内容的读者应该已经创建好了名为 shop 的数据库。接下来请继续完成创建表的工作。

### KEYWORD

● CREATE TABLE 语句

#### ③

这里我们仅指定了使用该语法所需的最少项目，实际开发数据库时还需要指定各种其他项目。

前面提到，在创建表之前，一定要先创建用来存储表的数据库。运行 CREATE DATABASE 语句就可以在 RDBMS 上创建数据库了。CREATE DATABASE 语句的语法如下所示<sup>①</sup>。

#### 语法 1-1 创建数据库的 CREATE TABLE 语句

```
CREATE DATABASE <数据库名称>
```

这里我们将数据库命名为 shop，然后执行代码清单 1-1 中的 SQL 语句<sup>②</sup>。

#### 代码清单 1-1 创建数据库 shop 的 CREATE DATABASE 语句

```
CREATE DATABASE shop;
```

此外，数据库名称、表名以及列名都要使用半角字符（英文字母、数字、符号），具体内容随后会进行介绍。

## 表的创建 (CREATE TABLE 语句)

创建好数据库之后，接下来我们使用 CREATE TABLE 语句在其中创建表。CREATE TABLE 语句的语法如下所示<sup>③</sup>。

## 语法1-2 创建表的CREATE TABLE语句

```
CREATE TABLE <表名>
(<列名1> <数据类型> <该列所需约束>,
<列名2> <数据类型> <该列所需约束>,
<列名3> <数据类型> <该列所需约束>,
<列名4> <数据类型> <该列所需约束>,
    ⋮
<该表的约束1>, <该表的约束2>, ……);
```

该语法清楚地描述了我们要创建一个包含<列名1>、<列名2>、……，名称为<表名>的表，非常容易理解。每一列的数据类型（后述）是必须要指定的，还要为需要的列设置约束（后述）。约束可以在定义列的时候进行设置，也可以在语句的末尾进行设置<sup>①</sup>。

在数据库中创建表1-2中的商品表（Shohin表）的CREATE TABLE语句，请参见代码清单1-2。

## 代码清单1-2 创建Shohin表的CREATE TABLE语句

```
CREATE TABLE Shohin
(shohin_id      CHAR(4)      NOT NULL,
shohin_mei     VARCHAR(100) NOT NULL,
shohin_bunrui VARCHAR(32)   NOT NULL,
hanbai_tanka   INTEGER      ,
shiire_tanka   INTEGER      ,
torokubi       DATE         ,
PRIMARY KEY (shohin_id));
```

## 备忘

本书将持续创建出Shohin表等学习中用到一些示例表。创建这些表的SQL语句保存在随书光盘的\Sample\CreateTable\<RDBMS名>目录下，文件名为CreateTable<表名>.sql的文件当中。例如在PostgreSQL中创建Shohin表所使用的SQL语句，保存在随书光盘的\Sample\CreateTable\PostgreSQL目录下的CreateTableShohin.sql文件当中。

CreateTableShohin.sql文件包含了创建Shohin表时用到的SQL语句（代码清单1-2），以及向Shohin表中插入数据的SQL语句（代码清单1-6）。这样就可以在创建表的同时向表中预先插入数据了。

## ①

但是NOT NULL约束只能以列为单位进行设置

## 命名规则

代码清单 1-2 中的表和列的名称都使用了日语。但实际上，在数据库中创建表时是不能使用日语（全角字符）作为名称的。当然，表中存储的数据是可以使用日语的。否则，我们的数据库也就失去了存在的意义。

我们只能使用半角英文字母、数字、下划线（\_）作为数据库、表和列的名称。例如，不能将 `shohin_id` 改写成 `shohin-id`。标准 SQL 并不承认使用连字符作为列名等名称使用。\$、#、? 这样的符号同样不能作为名称使用。

尽管有些 RDBMS 允许使用上述符号或者日语（全角字符）作为列的名称来使用，但这也仅仅限于在该 RDBMS 中使用，并不能保证在其他 RDBMS 中也能使用。虽然大家可能会觉得限制有点太多了，但还是请遵守规则使用半角英文字母、数字和下划线（\_）吧。



### 法则 1-9

数据库名称、表名和列名等可以使用以下三种字符。

- 半角英文字母
- 半角数字
- 下划线（\_）

此外，名称必须以半角英文字母开头。以记号开头的名称并不多见，但有时会碰到类似 `1shohin` 或者 `2009_uriage` 这样以数字开头的名称。虽然想法可以理解，但这在标准 SQL 中是被禁止的。请大家使用 `shohin1` 或者 `uriage_2009` 这样符合规则的名称。



### 法则 1-10

名称必须以半角英文字母作为开头。

最后还有一点，在同一个数据库中不能创建两个相同名称的表，在同一个表中也不能创建两个名称相同的列。如果出现这样的情况，RDBMS 会返回错误信息。



### 法则 1-11

名称不能重复。

接下来我们根据上述规则，使用代码清单 1-2 中的 CREATE TABLE 语句来创建表 1-2 中的商品表。表名为 Shohin，表中的列名如表 1-3 所示。

表 1-3 商品表和 Shohin 表列名的对应关系

商品表中的列名	Shohin 表定义的列名
商品编号	shohin_id
商品名称	shohin_mei
商品分类	shohin_bunrui
销售单价	hanbai_tanka
进货单价	shiire_tanka
登记日期	torokubi

## 数据类型的指定

Shohin 表所包含的列，定义在 CREATE TABLE Shohin( ) 的括号中。列名右边的 INTEGER 或者 CHAR 等关键字，是用来声明该列的数据类型的，所有的列都必须指定数据类型。

数据类型表示数据的种类，包括数字型、字符型和日期型等。每一列都不能存储与该列数据类型不符的数据。声明为整数型的列中不能存储 'abc' 这样的字符串，声明为字符型的列中也不能存储 1234 这样的数字。

数据类型的种类很多，各个 RDBMS 之间也存在很大差异。根据业务需要创建真实的数据库时，一定要根据不同的 RDBMS 选用最恰当的数据类型。在学习 SQL 的时候，使用最基本的数据类型就足够了。下面我们就来介绍四种基本的数据类型。

### KEYWORD

- 数据类型
- 数字型
- 字符型
- 日期型

### KEYWORD

- INTEGER 型

#### ● INTEGER 型

用来指定存储整数的列的数据类型（数字型），不能存储小数。

### KEYWORD

- CHAR 型

#### ● CHAR 型

CHAR 是 CHARACTER（字符）的简称，是用来指定存储字符串列的数据类型（字符型）。可以像 CHAR(10) 或者 CHAR(200) 这样，在括号中指定该列可以存储的字符串的长度（最大长度）。字符串超出最大

## ⑧

字节是计算机内部的数据单位。一个字符通常需要1到3个字节来表示(根据字符的种类和表现方式有所不同)。

**KEYWORD**

- 定长字符串

**KEYWORD**

- VARCHAR 型
- 可变长字符串

## ⑨

VARCHAR 中的 VAR 是 VARYING (可变的) 的缩写。

**KEYWORD**

- VARCHAR2 型

**KEYWORD**

- DATE 型

长度的部分是无法输入到该列中的。RDBMS 不同,长度单位也不一样,既存在使用字符个数的情况,也存在使用字节长度<sup>⑧</sup>的情况。

字符串以定长字符串的形式存储在被指定为 CHAR 型的列中。所谓定长字符串,就是当列中存储的字符串长度达不到最大长度的时候,使用半角空格进行补足。例如,我们向 CHAR(8) 类型的列中输入 'abc' 的时候,会以 'abc ' (abc 后面有 5 个半角空格) 的形式保存起来。

另外,虽然之前我们说过 SQL 不区分英文字母的大小写,但是表中存储的字符串却是区分大小写的。也就是说, 'ABC' 和 'abc' 代表了两个不同意义的字符串。

## ● VARCHAR 型

同 CHAR 类型一样, VARCHAR 型也是用来指定存储字符串的列的数据类型(字符串类型)。也可以通过括号内的数字来指定字符串的长度(最大长度)。但该类型的列是以可变长字符串的形式来保存字符串<sup>⑨</sup>。定长字符串在字符数未达到最大长度时会用半角空格补足,但可变长字符串不同,即使字符数未达到最大长度,也不会用半角空格补足。例如,我们向 VARCHAR(8) 类型的列中输入字符串 'abc' 的时候,保存的就是字符串 'abc'。

该类型的列中存储的字符串也和 CHAR 类型一样,是区分大小写的。

**特定的 SQL**

Oracle 中使用 VARCHAR2 型 (Oracle 中也有 VARCHAR 这种数据类型,但并不推荐使用)。

## ● DATE 型

用来指定存储日期(年月日)的列的数据类型(日期型)。

**特定的 SQL**

除了年月日之外, Oracle 中使用的 DATE 型还包含时分秒,但在本书中我们只学习日期部分。

## 约束的设置

约束是除了数据类型之外，对列中存储的数据进行限制或者追加条件的功能。Shohin 表中设置了两种约束。

Shohin 表的 shohin\_id 列、shohin\_mei 列和 shohin\_bunrui 列的定义如下所示。

```
shohin_id      CHAR(4)      NOT NULL,
shohin_mei     VARCHAR(100) NOT NULL,
shohin_bunrui VARCHAR(32)  NOT NULL,
```

数据类型的右侧设定了 NOT NULL 的约束。NULL 是代表空白(无记录)的关键字<sup>①</sup>。在 NULL 之前加上了表示否定的 NOT，就是给该列设定了不能输入空白，也就是必须输入的约束(如果什么都不输入就会出错)。

这样一来，Shohin 表的 shohin\_id (商品编号)列、shohin\_mei (商品名称)列和 shohin\_bunrui (商品分类)列就都成了必须输入的项目。

另外，在创建 Shohin 表的 CREATE TABLE 语句的后面，还有下面这样的记述。

```
PRIMARY KEY (shohin_id)
```

这是用来给 shohin\_id 列设定主键约束的。所谓键值，就是在指定特定数据时使用的列的组合。键值种类多样，主键(primary key)就是可以特定一行数据的列<sup>②</sup>。也就是说，如果把 shohin\_id 列指定为主键，就可以通过该列取出特定的商品数据了。

反之，如果向 shohin\_id 列中输入了重复数据，就无法取出唯一的特定数据了(因为无法确定唯一的一行数据)。这样就可以为某一列设定主键约束了。

### KEYWORD

#### ●约束

### KEYWORD

- NOT NULL约束
- NULL

#### ①

NULL这个词是无或空的意思，NULL是使用SQL时的常见关键字，请大家牢记。

### KEYWORD

- 主键约束
- 键值
- 主键(primary key)

#### ②

特定一行数据，也可以说是唯一确定一行数据。

## 1-5

## 第1章 数据库和SQL

## 表的删除和更新

## 学习要点

- 使用 DROP TABLE 语句来删除表。
- 使用 ALTER TABLE 语句向表中添加列或者从表中删除列。

## KEYWORD

- DROP TABLE 语句

## 表的删除 ( DROP TABLE 语句 )

此前介绍的都是关于 Shohin 表的内容的创建，下面我们就来介绍一下删除表的方法。删除表的 SQL 语句非常简单，只需要一行 DROP TABLE 语句即可。

语法 1-3 删除表时使用的 DROP TABLE 语句

```
DROP TABLE <表名>;
```

如果想要删除 Shohin 表，只需要像代码清单 1-3 那样书写 SQL 语句即可<sup>❶</sup>。

代码清单 1-3 删除 Shohin 表

```
DROP TABLE Shohin;
```

DROP 在英语中是丢掉、舍弃的意思。需要特别注意的是，删除的表是无法恢复的<sup>❷</sup>。即使是被误删的表，也无法恢复。只能重新创建，然后重新插入数据。

如果不小心删除了重要的业务表，那就太悲剧了。特别是存储了大量数据的表，恢复起来费时费力，请大家务必注意！

## ❶

随后还需使用 Shohin 表来学习相关知识，请不要删除 Shohin 表。如果已经删除，请重新创建 Shohin 表。

## ❷

其实很多 RDBMS 都预留了恢复的功能。但还是请大家认为是无法恢复的。



## 法则 1-12

删除了的表是无法恢复的。  
在执行 DROP TABLE 语句之前请务必仔细确认。

**KEYWORD**

● ALTER TABLE 语句

**表定义的更新 (ALTER TABLE 语句)**

有时好不容易把表创建出来之后才发现少了几列，其实这时无需把表删除再重新创建，只需使用变更表定义的 ALTER TABLE 语句就可以了。ALTER 在英语中就是改变的意思。下面就给大家介绍该语句通常的使用方法。

首先是添加列时使用的语法。

**语法 1-4 添加列的 ALTER TABLE 语句**

```
ALTER TABLE <表名> ADD COLUMN <列的定义>;
```

**特定的 SQL**

Oracle 和 SQL Server 中不用写 COLUMN。

```
ALTER TABLE <表名> ADD <列名>;
```

另外，在 Oracle 中同时添加多列的时候，可以像下面这样使用括号。

```
ALTER TABLE <表名> ADD (<列名>, <列名>, ……);
```

例如，我们可以使用代码清单 1-4 中的语句在 Shohin 表中添加这样一列，shohin\_mei\_kana（商品名称（假名）），该列可以存储 100 位可变长的字符串。

**代码清单 1-4 添加一列可以存储 100 位可变长度字符串的 shohin\_mei\_kana 列**

```
DB2 PostgreSQL MySQL  
ALTER TABLE Shohin ADD COLUMN shohin_mei_kana VARCHAR(100);
```

```
Oracle  
ALTER TABLE Shohin ADD (shohin_mei_kana VARCHAR2(100));
```

```
SQL Server  
ALTER TABLE Shohin ADD shohin_mei_kana VARCHAR(100);
```

反之，删除表中某列使用的语法如下所示。

**语法 1-5 删除列的 ALTER TABLE 语句**

```
ALTER TABLE <表名> DROP COLUMN <列名>;
```



**特定的 SQL**

Oracle 和 SQL Server 中不用写 COLUMN。

```
ALTER TABLE <表名> DROP <列名>;
```

另外，在 Oracle 中同时删除多列的时候，可以像下面这样使用括号来实现。

```
ALTER TABLE <表名> DROP (<列名>, <列名>, .....);
```

例如，我们可以使用代码清单 1-5 中的语句来删除之前添加的 shohin\_mei\_kana 列。

代码清单 1-5 删除 shohin\_mei\_kana 列

```
SQL Server DB2 PostgreSQL MySQL
ALTER TABLE Shohin DROP COLUMN shohin_mei_kana;
```

```
Oracle
ALTER TABLE Shohin DROP (shohin_mei_kana);
```

ALTER TABLE 语句和 DROP TABLE 语句一样，执行之后无法恢复。误添的列可以通过 ALTER TABLE 语句删除，但是从表中删除之后就只能重新再创建了。

**法则 1-13**

表定义变更之后无法恢复。

在执行 ALTER TABLE 语句之前请务必仔细确认。

## 向 Shohin 表中插入数据

最后让我们来尝试一下向表中插入数据。从下一章开始，大家将会使用插入到 Shohin 表中的数据，来学习如何编写操作数据的 SQL 语句。

向 Shohin 表中插入数据的 SQL 语句请参见代码清单 1-6。

代码清单 1-6 向 Shohin 表中插入数据的 SQL 语句

**SQL Server PostgreSQL**

-- DML: 插入数据

```

BEGIN TRANSACTION; ①

INSERT INTO Shohin VALUES ('0001', 'T恤衫', '衣服', 1000, 500, '2009-09-20');
INSERT INTO Shohin VALUES ('0002', '打孔器', '办公用品', 500, 320, '2009-09-11');
INSERT INTO Shohin VALUES ('0003', '运动T恤', '衣服', 4000, 2800, NULL);
INSERT INTO Shohin VALUES ('0004', '菜刀', '厨房用具', 3000, 2800, '2009-09-20');
INSERT INTO Shohin VALUES ('0005', '高压锅', '厨房用具', 6800, 5000, '2009-01-15');
INSERT INTO Shohin VALUES ('0006', '叉子', '厨房用具', 500, NULL, '2009-09-20');
INSERT INTO Shohin VALUES ('0007', '擦菜板', '厨房用具', 880, 790, '2008-04-28');
INSERT INTO Shohin VALUES ('0008', '圆珠笔', '办公用品', 100, NULL, '2009-11-11');

COMMIT;

```

⇒表示下一行接续本行,只是由于版面所限而换行。

**特定的SQL**

DBMS 不同,代码清单 1-6 中的 DML 语句也略有不同。

在 MySQL 中运行时,需要把①中的 BEGIN TRANSACTION; 改写成

**START TRANSACTION;**

在 Oracle 和 DB2 中运行时,无需使用①中的 BEGIN TRANSACTION; (请予以删除)。

这些在不同的 DBMS 中使用的 DML 语句,都保存在随书光盘的 \Sample\CreateTable\&lt;RDBMS名&gt; 文件夹下的 CreateTableShohin.sql 文件中。

使用插入行的指令语句 INSERT, 就可以把表 1-2 中的数据都插入到表中了。开头的 BEGIN TRANSACTION 语句是开始插入行的指令语句, 结尾的 COMMIT 语句是确定插入行的指令语句。这些指令语句将会在第 4 章详细介绍, 大家不必急于记住这些语句。

## 专栏

## 表的修改

本节将名为 Shohin 的表作为例子进行了讲解,估计会有些读者在匆忙中把表名误写成了 Sohlin,创建出了名称错误的表,这可怎么办呢?

如果还没有向表中插入数据,那么只需要把表 DROP,再重新创建 CREATE 一个名称正确的表就可以了。可是如果在发现表名错误之前就已经向表中插入了大量数据,再这样做就麻烦了。毕竟插入大量的数据既费时又费力。抑或起初决定好的表名,之后又觉得不好想换掉,这种情况也很麻烦。

其实很多数据库都提供了可以修改表名的指令 (RENAME) 来解决这样的问题。例如,如果想把 Sohlin 表的名称变为 Shohin,可以使用代码清单 1-A 中的指令。

## 代码清单 1-A 变更表名

Oracle PostgreSQL

```
ALTER TABLE Sohlin RENAME TO Shohin;
```

DB2

```
RENAME TABLE Sohlin TO Shohin;
```

SQL Server

```
sp_rename 'Sohlin', 'Shohin';
```

MySQL

```
RENAME TABLE Sohlin to Shohin;
```

通常在 RENAME 之后按照 <变更前的名称>、<变更后的名称>的顺序来指定表的名称。

各个数据库的语法都不尽相同,是因为标准 SQL 并没有 RENAME,于是各个数据库便使用了各自惯用的语法。如上所述创建了错误的表名,或者想要保存表的备份时,使用这些语句非常方便。但美中不足的是,由于各个数据库的语法不同,很难一下子想出恰当的指令。这时大家就可以来参考本专栏。

## KEYWORD

## ● RENAME

## 练习题

- 1.1 编写一条 CREATE TABLE 语句, 用来创建一个包含表 1-A 中所列各项的表 Jyushoroku (地址簿), 并为 toroku\_bango (注册编号) 列设定主键约束。

表 1-A 表 Jyushoroku (地址簿) 中的列

列的含义	列的名称	数据类型	约束
注册编号	toroku_bango	整数类型	不能为 NULL; 主键
姓名	name	可变长字符串类型 (长度为 128)	不能为 NULL
住址	jyusho	可变长字符串类型 (长度为 256)	不能为 NULL
电话号码	tel_no	定长字符串类型 (长度为 10)	
邮件地址	mail_address	定长字符串类型 (长度为 20)	

- 1.2 假设在创建练习 1.1 中的 Jyushoroku 表时忘记添加如下列 yubin\_bango (邮政编码)。请把此列添加到 Jyushoroku 表中。

列名 : yubin\_bango  
 数据类型: 定长字符串类型 (长度为 8)  
 约束 : 不能为 NULL

- 1.3 编写 SQL 语句来删除 Jyushoroku 表。
- 1.4 编写 SQL 语句来恢复删除掉的 Jyushoroku 表。

## 第2章 查询基础

SELECT 语句基础  
算术运算符和比较运算符  
逻辑运算符



## 本章重点

---

本章将会和大家一起学习查询前一章创建的 Shohin 表中数据的 SQL 语句。这里使用的 SELECT 语句是 SQL 最基本也是最重要的语句。请大家在实际运行书中的 SELECT 语句时，亲身体验一下其书写方法和执行结果。

执行查询操作时可以指定想要查询数据的条件（查询条件）。查询时可以指定一个或多个查询条件，例如“某一列等于这个值”、“某一列计算之后的值大于这个值”等。

### 2-1 SELECT 语句基础

- 列的查询
- 查询出表中所有的列
- 为列设定别名
- 常数的查询
- 从结果中删除重复行
- 根据 WHERE 语句来选择记录
- 注释的书写方法

### 2-2 算术运算符和比较运算符

- 算术运算符
- 需要注意 NULL
- 比较运算符
- 对字符串使用不等号时的注意事项
- 不能对 NULL 使用比较运算符

### 2-3 逻辑运算符

- NOT 运算符
- AND 运算符和 OR 运算符
- 使用括号强化处理
- 逻辑运算符和真值
- 含有 NULL 时的真值

## 2-1

## 第2章 查询基础

## SELECT 语句基础

## 学习重点

- 使用 SELECT 语句从表中选取数据。
- 为列设定显示用的别名。
- SELECT 语句中可以使用常数或者表达式。
- 通过指定 DISTINCT 可以删除重复的行。
- SQL 语句中可以使用注释。
- 可以通过 WHERE 语句从表中选取符合查询条件的数据。

## KEYWORD

- SELECT 语句
- 匹配查询
- query

## 列的查询

从表中选取数据时需要使用 SELECT 语句，也就是只从表中选出 (SELECT) 必要数据的意思。通过 SELECT 语句查询并选取必要数据的过程称为匹配查询或查询 (query)。

SELECT 语句是 SQL 语句中使用最多，也是最基本的 SQL 语句。掌握了 SELECT 语句，距离掌握 SQL 语句就不远了。

SELECT 语句的基本语法如下所示。

## 语法 2-1 基本的 SELECT 语句

```
SELECT <列名>, .....
FROM <表名>;
```

## KEYWORD

- 子句

## 0

clause 在日语中也可以翻译成章节，但本书统一使用子句这一翻译。

该 SELECT 语句包含了 SELECT 和 FROM 两个子句 (clause)<sup>0</sup>。子句是 SQL 语句的组成要素，是以 SELECT 或者 FROM 等作为起始的短语。

SELECT 子句中列举了希望从表中查询出的列的名称，而 FROM 子句则指定了选取数据的表的名称。

接下来，我们尝试从第1章创建出的 Shohin（商品）表中，查询出图2-1所示的 shohin\_id（商品编号）列、shohin\_mei（商品名称）列和 shire\_tanka（进货单价）列。

图2-1 查询出Shohin表中的列

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shire_tanka (进货单价)	torokubi (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

输出这三列

对应的 SELECT 语句请参见代码清单 2-1，该语句正常执行的结果如执行结果所示<sup>①</sup>。

①

结果的显示方式根据DBMS的客户端不同略有不同（数据的内容都是相同的），如无特殊说明，本书记录的都是 PostgreSQL 8.4 的运行结果。

代码清单 2-1 从 Shohin 表中输出 3 列

```
SELECT shohin_id, shohin_mei, shire_tanka
FROM Shohin;
```

执行结果

shohin_id	shohin_mei	shire_tanka
0001	T恤衫	500
0002	打孔器	320
0003	运动T恤	2800
0004	菜刀	2800
0005	高压锅	5000
0006	叉子	
0007	擦菜板	790
0008	圆珠笔	

SELECT 语句第一行的 SELECT shohin\_id, shohin\_mei, shire\_tanka 就是 SELECT 子句。查询出的列的顺序可以任意指定。



②

行的顺序也可能存在与上述运行结果不同的情况。如果用户不设定 SELECT 语句执行结果中行的顺序, 就可能发生上述情况。行的排序方法将在第 3 章进行学习。

**KEYWORD**

● 星号 (\*)

查询多列时, 需要使用逗号进行分隔排列。查询结果中列的顺序和 SELECT 子句中的顺序相同<sup>②</sup>。

**查询出表中所有的列**

想要查询出全部列时, 可以使用代表所有列的星号 (\*)。

**语法 2-2 查询全部的列**

```
SELECT *
FROM <表名>;
```

例如, 查询 Shohin 表中全部列的语句请参见代码清单 2-2。

**代码清单 2-2 输出 Shohin 表中全部的列**

```
SELECT *
FROM Shohin;
```

得到的结果和代码清单 2-3 中的 SELECT 语句的结果相同。

**代码清单 2-3 与代码清单 2-2 具有相同含义的 SELECT 语句**

```
SELECT shohin_id, shohin_mei, shohin_bunrui, hanbai_tanka,
       shiire_tanka, torokubi
FROM Shohin;
```

执行结果如下所示。

**执行结果**

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11



## 法则 2-1

星号 (\*) 代表全部列的意思。

但是,如果使用星号的话,就无法设定列的显示顺序了。这时就会按照 CREATE TABLE 语句的定义对列进行排序。

## 专 栏

## 随意使用换行符

SQL 语句使用换行符或者半角空格来分隔单词,在任何位置进行分隔都可以。即使像下面这样通篇都是换行符也不会影响 SELECT 语句的执行。但是这样可能会由于看不清楚而出错。原则上希望大家能够以子句为单位进行换行(子句过长时,为方便起见可以换行)。

```
SELECT
*
FROM
Shohin
;
```

另外,像下面这样插入空行(无任何字符的行)会造成执行错误,请特别注意。

```
SELECT *
FROM Shohin;
```

## 为列设定别名

SQL 语句可以使用 AS 关键字为列设定别名。请参见代码清单 2-4。

代码清单 2-4 为列设定别名

```
SELECT shohin_id AS id,
shohin_mei AS namae,
shiire_tanka AS tanka
FROM Shohin;
```

## KEYWORD

- AS 关键字
- 别名

## 执行结果

id	naame	tanka
0001	T恤衫	500
0002	打孔器	320
0003	运动T恤	2800
0004	菜刀	2800
0005	高压锅	5000
0006	叉子	
0007	擦菜板	790
0008	圆珠笔	

## KEYWORD

- 双引号 (")

## ❶

使用双引号可以设定包含空格(空白)的别名。但是如果忘记使用双引号就可能出错,因此并不推荐。大家可以像shohin\_ichiran这样使用下划线(\_)来代替空白。

别名可以使用汉语,使用汉语时需要用双引号(")括起来❶。请注意不是单引号(')。设定汉语别名的SELECT语句请参见代码清单2-5。

## 代码清单2-5 设定汉语别名

```
SELECT shohin_id AS "商品编号",
       shohin_mei AS "商品名称",
       shiire_tanka AS "进货单价"
FROM Shohin;
```

## 执行结果

商品编号	商品名称	进货单价
0001	T恤衫	500
0002	打孔器	320
0003	运动T恤	2800
0004	菜刀	2800
0005	高压锅	5000
0006	叉子	
0007	擦菜板	790
0008	圆珠笔	

通过执行结果来理解就更加容易了。像这样使用别名可以让SELECT语句的执行结果更加容易理解和操作。



## 法则 2-2

设定汉语别名时需要使用双引号(")括起来。

**KEYWORD**

- 字符串常数
- 数字常数
- 日期常数

**0**

在 SQL 语句中使用字符串或者日期常数时，必须使用单引号(')将其括起来。

## 常数的查询

SELECT 子句中不仅可以书写列名，还可以书写常数。代码清单 2-6 中的 SELECT 子句中的第一列“商品”是字符串常数，第 2 列 38 是数字常数，第 3 列“2009-02-24”是日期常数，它们将与 shohin\_id 列和 shohin\_mei 列一起被查询出来。<sup>①</sup>

### 代码清单 2-6 查询常数

```
SELECT '商品' AS mojiretsu, 38 AS kazu, '2009-02-24' AS hizuke,
       shohin_id, shohin_mei
FROM Shohin;
```

### 执行结果

mojiretsu	kazu	hizuke	shohin_id	shohin_mei
商品	38	2009-02-24	0001	T恤衫
商品	38	2009-02-24	0002	打孔器
商品	38	2009-02-24	0003	运动T恤
商品	38	2009-02-24	0004	菜刀
商品	38	2009-02-24	0005	高压锅
商品	38	2009-02-24	0006	叉子
商品	38	2009-02-24	0007	擦菜板
商品	38	2009-02-24	0008	圆珠笔

如上述执行结果所示，所有的行中都显示出了 SELECT 子句中的常数。

此外，SELECT 子句中除了书写常数，还可以书写计算式。将在下一节中学习如何书写计算式。

## 从结果中删除重复行

想知道 Shohin 表中保存了哪些商品种类 (shohin\_bunrui) 时，如果能像图 2-2 那样删除重复的数据该有多好啊。

图2-2 除去重复数据后的商品种类

shohin_id {商品编号}	shohin_mei {商品名称}	shohin_bunrui {商品分类}	hanbai_tanka {销售单价}	shihre_tanka {进货单价}	torokubi {登记日期}
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

除去重复数据

shohin_bunrui {商品分类}
衣服
办公用品
厨房用具

**KEYWORD**

● DISTINCT 关键字

如上所示想要删除重复行时，可以通过在 SELECT 子句中使用 DISTINCT 来实现（代码清单 2-7）。

代码清单 2-7 使用 DISTINCT 删除 shohin\_bunrui 列中重复的数据

```
SELECT DISTINCT shohin_bunrui
FROM Shohin;
```

执行结果

```
shohin_bunrui
-----
厨房用具
衣服
办公用品
```

 法则 2-3

在 SELECT 语句中使用 DISTINCT 可以删除重复行。

在使用 DISTINCT 时，NULL 也被视为一类数据。存在多条 NULL 数

据行时，也会结合为一条 NULL 数据。对含有 NULL 数据的 shiire\_tanka（进货单价）列使用 DISTINCT 的 SELECT 语句请参见代码清单 2-8。除了两条 2800 的数据外，两条 NULL 的数据也被结合为一条。

代码清单 2-8 对含有 NULL 数据的列使用 DISTINCT 关键字

```
SELECT DISTINCT shiire_tanka
FROM Shohin;
```

执行结果

shiire_tanka
5000
790
500
2800
320

NULL 数据被保留了下来

DISTINCT 也可以像代码清单 2-9 所示的那样在多列之前使用。此时，会将多个列的数据进行组合，将重复的数据结合为一条。代码清单 2-9 中的 SELECT 语句，对 shohin\_bunrui（商品种类）列和 torokubi（登记日期）列的数据进行组合，将重复的数据结合为一条。

代码清单 2-9 在多列之前使用 DISTINCT

```
SELECT DISTINCT shohin_bunrui, torokubi
FROM Shohin;
```

执行结果

shohin_bunrui	torokubi
衣服	2009-09-20
办公用品	2009-09-11
办公用品	2009-11-11
衣服	
厨房用具	2009-09-20
厨房用具	2009-01-15
厨房用具	2008-04-28

如上述执行结果所示，shohin\_bunrui 列为“厨房用具”，同时 torokubi 列为 2009-09-20 的两条数据被结合成了一条。

DISTINCT 关键字只能用在第一个列名之前。因此，请大家注意不能写成 torokubi, DISTINCT shohin\_bunrui。

**KEYWORD**

● WHERE子句

**0**

这和Excel中根据过滤条件对行进行过滤的功能是相同的。

## 根据WHERE语句来选择记录

前面的例子都是将表中存储的数据全都选取出来，但实际的上并不是每次都需要选取全部数据，大部分情况都是要选取满足“商品种类为衣服”、“销售单价在1000日元以上”等某些条件的数据。

SELECT语句通过WHERE子句来指定查询数据的条件。在WHERE子句中可以指定“某一列的值和这个字符串相等”或者“某一列的值大于这个数字”等条件。执行含有这些条件的SELECT语句，就可以查询出只符合该条件的记录了。<sup>①</sup>

在SELECT语句中使用WHERE子句的语法如下所示。

语法2-3 SELECT语句中的WHERE子句

```
SELECT <列名>, .....
FROM <表名>
WHERE <条件表达式>;
```

图2-3显示了准备从Shohin表中选取的商品种类(shohin\_bunrui)为“衣服”的记录。

图2-3 准备选取的商品种类为“衣服”的记录

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	煎菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

选取shohin\_bunrui列为衣服的记录

从被选取的记录中还可以查询出想要的列。为了更加容易理解，我们在查询shohin\_bunrui列的同时，把shohin\_mei列也读取出来。SELECT语句请参见代码清单2-10。

代码清单2-10 用来选取shohin\_bunrui列为“衣服”的记录(SELECT语句)

```
SELECT shohin_mei, shohin_bunrui
FROM Shohin
WHERE shohin_bunrui = '衣服';
```

执行结果

```
shohin_mei |shohin_bunrui
-----+-----
T恤衫      | 衣服
运动T恤    | 衣服
```

WHERE子句中的shohin\_bunrui = '衣服'就是用来反映查询条件的表达式(条件表达式)。等号是比较两边的内容是否相等的符号,上述条件就是将shohin\_bunrui列的值和'衣服'进行比较,判断是否相等。Shohin表的所有记录都会进行比较。

接下来会从查询出的记录中选取SELECT语句指定的shohin\_mei列和shohin\_bunrui列,如执行结果所示。也就是首先通过WHERE子句查询出符合指定条件的记录,然后再选取SELECT语句指定的列(图2-4)。

图2-4 选取行之后,再输出列

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

① 选取行

② 输出列

代码清单2-10中的语句为了确认选取出的数据是否正确,通过SELECT子句把作为查询条件的shohin\_bunrui列也选取出来了,其实这并不是必须的。如果只想知道商品名称的话,可以像代码清单2-11那

## KEYWORD

● 条件表达式



样只选取出 shohin\_mei 列。

代码清单 2-11 也可以不选取出作为查询条件的列

```
SELECT shohin_mei
FROM Shohin
WHERE shohin_bunrui = '衣服';
```

执行结果

```
shohin_mei
-----
T恤衫
运动T恤
```

SQL 中子句的书写顺序是固定的，不能随意更改。**WHERE** 子句必须紧跟在 **FROM** 子句之后。书写顺序发生改变的话会造成执行错误（代码清单 2-12）。

代码清单 2-12 随意改变子句的书写顺序会造成错误

```
SELECT shohin_mei, shohin_bunrui
WHERE shohin_bunrui = '衣服'
FROM Shohin;
```

执行结果 (PostgreSQL)

```
ERROR:  "FROM"或者其前后有语法错误
第3行: FROM Shohin;
      ^
```



#### 法则 2-4

**WHERE** 子句要紧跟在 **FROM** 子句之后。

## KEYWORD

● 注释

## 注释的书写方法

最后给大家介绍一下注释的书写方法。注释是 SQL 语句中用来标识说明或者注意事项的部分。

注释对 SQL 的执行没有任何影响。因此，无论是英文字母还是汉字都可以随意使用。

注释的书写方法有如下两种。

**KEYWORD**

- 1行注释
- --

**0**

MySQL中需要在“--”之后加入半角空格(如果不加的话就不会被认为是注释)。

**KEYWORD**

- 多行注释
- /\*
- \*/

## ● 1行注释

书写在“--”之后，只能写在同一行。●

## ● 多行注释

书写在“/\*”和“\*/”之间，可以跨多行。

实际的示例请参见代码清单 2-13 和代码清单 2-14。

## 代码清单 2-13 1行注释的例子

```
-- 本SELECT语句用来除去结果中的重复数据。
SELECT DISTINCT shohin_id, shiire_tanka
FROM Shohin;
```

## 代码清单 2-14 多行注释的例子

```
/* 本SELECT语句，
   用来删除结果中的重复数据。*/
SELECT DISTINCT shohin_id, shiire_tanka
FROM Shohin;
```

任何注释都可以插在 SQL 语句中(代码清单 2-15、代码清单 2-16)。

## 代码清单 2-15 在 SQL 语句中插入 1 行注释

```
SELECT DISTINCT shohin_id, shiire_tanka
-- 本SELECT语句用来删除结果中的重复数据。
FROM Shohin;
```

## 代码清单 2-16 在 SQL 语句中插入多行注释

```
SELECT DISTINCT shohin_id, shiire_tanka
/* 本SELECT语句，
   用来删除结果中的重复数据。*/
FROM Shohin;
```

这些 SELECT 语句的执行结果与没有使用注释时完全一样。注释能够帮助阅读者更好地理解 SQL 语句，特别是在书写复杂的 SQL 语句时，希望大家能够尽量多加注简明易懂的注释。注释不仅可以写在 SELECT 语句中，而且可以写在任何 SQL 语句当中，写多少都可以。

**法则 2-5**

注释是 SQL 语句中用来标识说明或者注意事项的部分。  
分为 1 行注释和多行注释两种。

# 2-2

## 第2章 查询基础

# 算术运算符和比较运算符

### 学习重点

- 运算符就是对其两边的列或者值进行运算(计算或者大小比较等)的符号。
- 使用算术运算符可以进行四则运算。
- 括号可以提升运算的优先顺序(优先进行运算)。
- 包含NULL的运算,其结果也是NULL。
- 比较运算符可以用来判断列或者值是否相等,还可以用来比较大小。
- 判断是否为NULL,需要使用 IS NULL 或者 IS NOT NULL 运算符。

## 算术运算符

SQL 语句中可以使用计算表达式。代码清单 2-17 中的 SELECT 语句,把各个商品单价的两倍(hanbai\_tanka 的 2 倍)以 hanbai\_tanka\_x2 列的形式读取出来。

代码清单 2-17 SQL 语句中也可以使用运算表达式

```
SELECT shohin_mei, hanbai_tanka,  
       hanbai_tanka * 2 AS "hanbai_tanka_x2"  
FROM Shohin;
```

执行结果

shohin_mei	hanbai_tanka	hanbai_tanka_x2
T 恤衫	1000	2000
打孔器	500	1000
运动T恤	4000	8000
菜刀	3000	6000
高压锅	6800	13600
叉子	500	1000
擦菜板	880	1760
圆珠笔	100	200

hanbai\_tanka\_x2 列中的 hanbai\_tanka \* 2 就是计算销售单价 2 倍的表达式。以 shohin\_mei 列的值为“T 恤衫”的记录行为例,

hanbai\_tanka 列的值 1000 的 2 倍是 2000，它以 hanbai\_tanka\_x2 列的形式被查询出。同样，“打孔器”记录行的值 500 的 2 倍 1000，“运动 T 恤”记录行的值 2000 的 2 倍 8000，都被查询出了。运算就是这样以行为单位执行的。

SQL 语句中可以使用的四则运算的主要运算符如表 2-1 所示。

表 2-1 SQL 语句中可以使用的四则运算的主要运算符

含义	运算符
加法运算	+
减法运算	-
乘法运算	*
除法运算	/

#### KEYWORD

- + 运算符
- - 运算符
- \* 运算符
- / 运算符

#### KEYWORD

- 算术运算符
- 运算符

四则运算所使用的运算符 (+、-、\*、/) 称为算术运算符。运算符就是使用其两边的值进行四则运算或者字符串拼接、数值大小比较等运算，并返回结果的符号。加法运算符 (+) 前后如果是数字或者数字类型的列名的话，就会返回加算后的结果。SQL 中除了算术运算符之外还有各种各样的运算符。



#### 法则 2-6

SELECT 子句中可以使用常数或者表达式。

#### KEYWORD

- ()

当然，SQL 中也可以像平常的运算表达式那样使用括号。括号中运算表达式的优先级会得到提升，优先进行运算。例如在运算表达式 (1+2) \* 3 中，会先计算 1+2 的值，然后再对其结果进行 \* 3 运算。

括号的使用并不仅仅局限于四则运算，还可以用在 SQL 语句的任何表达式当中。具体的使用方法今后会慢慢介绍给大家。

## 需要注意 NULL

像前述代码清单 2-17 那样，SQL 语句中进行运算时，需要特别注意含有 NULL 的运算。请大家考虑一下如果 SQL 语句中进行如下运算时，

结果会是什么呢？

- Ⓐ 5 + NULL
- Ⓑ 10 - NULL
- Ⓒ 1 \* NULL
- Ⓓ 4 / NULL
- Ⓔ NULL / 9
- Ⓕ NULL / 0

正确答案全部都是 NULL。大家可能会觉得奇怪，为什么会这样呢？实际上所有包含 NULL 的计算，结果肯定是 NULL。即使像 F 那样用 NULL 除以 0 时这一原则也适用。通常情况下，类似 5/0 这样除数为 0 的话会发生错误，只有 NULL 除以 0 时不会发生错误，并且结果还是 NULL。

尽管如此，还是有很多时候我们希望 NULL 能像 0 那样，得到 5 + NULL = 5 这样的结果。不过也不要紧，SQL 中也为我们准备了可以解决这类情况的方法（将会在 6-1 节中进行介绍）。

## 专 栏

### FROM 子句真的有必要吗？

在第 1 节中我们介绍过 SELECT 语句是由 SELECT 子句和 FROM 子句组成的。可实际上 FROM 子句在 SELECT 语句中并不是必不可少的，只使用 SELECT 子句进行计算也是可以的。

代码清单 2-A 只包含 SELECT 子句的 SELECT 语句

```
SQL Server | PostgreSQL | MySQL
SELECT (100 + 200) * 3 AS keisan;
```

#### 执行结果

```
keisan
-----
900
```

实际上，通过执行 SELECT 语句来代替计算器的情况基本上是不存在的。不过在极少数情况下，还是可以通过使用没有 FROM 子句的 SELECT 语句来实现某种业务的。例如，希望得到内容为空，只包含一行临时数据的情况。

但是也存在像 Oracle 这样不允许省略 SELECT 语句中的 FROM 子句的 RDBMS，请大家注意<sup>①</sup>。

#### ①

在 Oracle 中，FROM 子句是必须的，这种情况下可以使用 DUAL 这个临时表。另外，DB2 中可以使用 SYSIBM.SYSDUMMY1 这个临时表。

## 比较运算符

在 2-1 节学习 WHERE 子句时，我们使用符号 = 从 Shohin 表中选出了商品种类 (shohin\_bunrui) 为字符串 '衣服' 的记录。下面让我们再使用符号 = 选取出销售单价 (hanbai\_tanka) 为 500 日元 (数字 500) 的记录 (代码清单 2-18)。

代码清单 2-18 选取出 hanbai\_tanka 列为 500 的记录

```
SELECT shohin_mei, shohin_bunrui
FROM Shohin
WHERE hanbai_tanka = 500;
```

执行结果

shohin_mei	shohin_bunrui
打孔器	办公用品
叉子	厨房用具

像符号 = 这样用来比较其两边的列或者值的符号称为比较运算符。符号 = 就是比较运算符。在 WHERE 子句中通过使用比较运算符可以组合出各种各样的条件表达式。

接下来，我们使用“不等于”这样代表否定含义的比较运算符 <><sup>❶</sup>，选取出 hanbai\_tanka 列的值不为 500 的记录 (代码清单 2-19)。

代码清单 2-19 选取出 hanbai\_tanka 列的值不是 500 的记录

```
SELECT shohin_mei, shohin_bunrui
FROM Shohin
WHERE hanbai_tanka <> 500;
```

执行结果

shohin_mei	shohin_bunrui
T恤衫	衣服
运动T恤	衣服
菜刀	厨房用具
高压锅	厨房用具
擦菜板	厨房用具
圆珠笔	办公用品

SQL 中主要的比较运算符如表 2-2 所示，除了等于和不等之外，还有进行大小比较的运算符。

### KEYWORD

- 比较运算符
- = 运算符
- <> 运算符

### ❶

有很多 RDBMS 可以使用比较运算符 “!=” 来实现不等于功能。但这是限于不被标准 SQL 所承认的特定 SQL。出于安全的考虑，最好不要使用。

## KEYWORD

- = 运算符
- <> 运算符
- >= 运算符
- > 运算符
- <= 运算符
- < 运算符

表 2-2 比较运算符

运算符	含义
=	和 - 相等
<>	和 - 不相等
>=	大于等于 -
>	大于 -
<=	小于等于 -
<	小于 -

这些比较运算符可以对字符、数字和日期等几乎所有数据类型的列和值进行比较。例如，从 Shohin 表中选取出销售单价 (hanbai\_tanka) 大于等于 1000 日元的记录，或者登记日期 (torokubi) 在 2009 年 9 月 27 日之前的记录。可以使用比较运算符 <= 和 <，在 WHERE 子句中生成如下条件表达式。

代码清单 2-20 选取出销售单价大于等于 1000 日元的记录

```
SELECT shohin_mei, shohin_bunrui, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka >= 1000;
```

执行结果

shohin_mei	shohin_bunrui	hanbai_tanka
T恤衫	衣服	1000
运动T恤	衣服	4000
菜刀	厨房用具	3000
高压锅	厨房用具	6800

代码清单 2-21 选取出登记日期在 2009 年 9 月 27 日之前的记录

```
SELECT shohin_mei, shohin_bunrui, torokubi
FROM Shohin
WHERE torokubi < '2009-09-27';
```

执行结果

shohin_mei	shohin_bunrui	torokubi
T恤衫	衣服	2009-09-20
打孔器	办公用品	2009-09-11
菜刀	厨房用具	2009-09-20
高压锅	厨房用具	2009-01-15
叉子	厨房用具	2009-09-20
煎菜板	厨房用具	2008-04-28

小于某个日期就是在该日期之前的意思。想要实现在某个特定日期(包含该日期)之后的查询条件时,可以使用代表大于等于的  $\geq$  运算符。

另外,在使用大于等于 ( $\geq$ ) 或者小于等于 ( $\leq$ ) 作为查询条件时,一定要注意不等号 ( $<$ 、 $>$ ) 和等号 ( $=$ ) 的位置不能颠倒。一定要让不等号在左,等号在右。如果写成 ( $=<$ ) 或者 ( $=>$ ) 就会出错。当然,代表不等于的比较运算符也不能写成 ( $><$ )。



#### 法则 2-7

使用比较运算符时一定要注意不等号和等号的位置。

除此之外,还可以使用比较运算符对计算结果进行比较。代码清单 2-22 在 WHERE 子句中指定了销售单价 (hanbai\_tanka) 比进货单价 (shiire\_tanka) 高出 500 日元以上的条件表达式。为了判断是否高出 500 日元,需要用 hanbai\_tanka 列的值减去 shiire\_tanka 列的值。

代码清单 2-22 WHERE 子句的条件表达式中也可以使用计算表达式

```
SELECT shohin_mei, hanbai_tanka, shiire_tanka
   FROM Shohin
  WHERE hanbai_tanka - shiire_tanka >= 500;
```

执行结果

shohin_mei	hanbai_tanka	shiire_tanka
T恤衫	1000	500
运动T恤	4000	2800
高压锅	6800	5000

## 对字符串使用不等号时的注意事项

对字符串使用大于等于或者小于等于不等号时会得到什么样的结果呢?接下来我们使用表 2-3 中的 Chars 表来进行确认。虽然该表中存储的都是数字,但 chr 是字符串类型 (CHAR 类型) 的列。



表2-3 Chars表

chr(字符串类型)
1
2
3
10
11
222

可以使用代码清单 2-23 中的 SQL 语句来创建 Chars 表。

代码清单 2-23 创建 Chars 表并插入数据

```
-- DDL: 创建表
CREATE TABLE Chars
(chr CHAR(3) NOT NULL,
PRIMARY KEY (chr));

-- DML: 插入数据
BEGIN TRANSACTION;

INSERT INTO Chars VALUES ('1');
INSERT INTO Chars VALUES ('2');
INSERT INTO Chars VALUES ('3');
INSERT INTO Chars VALUES ('10');
INSERT INTO Chars VALUES ('11');
INSERT INTO Chars VALUES ('222');

COMMIT;
```

**特定的 SQL**

代码清单 2-23 中的 DML 语句根据 DBMS 的不同而略有差异。在 MySQL 中执行该语句时，请大家把①的部分改成 "START TRANSACTION;"。在 Oracle 和 DB2 中执行时不需用到①的部分，请删除。

那么，对 Chars 表执行代码清单 2-24 中的 SELECT 语句（查询条件是 chr 列大于 '2'）会得到什么样的结果呢？

代码清单 2-24 选取大于 '2' 的数据的 SELECT 语句

```
SELECT chr
FROM Chars
WHERE chr > '2';
```

大家是不是觉得应该选取出比 2 大的 3、10、11 和 222 这 4 条记录呢？下面就让我们来看看该 SELECT 语句的执行结果吧。

#### 执行结果

```
chr
----
3
222
```

没想到吧？是不是觉得 10 和 11 比 2 大，所以也应该选取出来呢。大家之所以这样想是因为混淆了数字和字符串的缘故。也就是说 2 和 '2' 并不一样。

现在，chr 列被定为字符串类型，并且在对字符串类型的数据进行大小比较时，使用的是和数字比较不同的规则。典型的规则就是按照字典顺序进行比较，也就是像姓名那样，按照条目在字典中出现的顺序来进行排序。该规则最重要的一点就是，以相同字符开头的单词比不同字符开头的单词更相近。

Chars 表 chr 列中的数据按照字典顺序进行排序的结果如下所示。

```
1
10
11
2
222
3
```

'10' 和 '11'，同样都是以 '1' 开头的字符串，首先判定为比 '2' 小。这就像在字典中“提问”、“提议”和“问题”按照如下顺序排列一样。

```
提问
提议
问题
```

或者我们以书籍的章节为例也可以。1-1 节包含在第 1 章当中，所以肯定比第 2 章更靠前。

```

1
1-1
1-2
1-3
2
2-1
2-2
3

```

进行大小比较时，得到的结果是 '1-3' 比 '2' 小 ('1-3' < '2')，'3' 大于 '2-2' ('3' > '2')。

比较字符串类型大小的规则，今后还会经常使用，所以请大家牢记<sup>①</sup>。

**①**  
该规则对定长字符串和可变字符串都适用。



#### 法则 2-8

字符串类型的数据原则上按照字典顺序进行排序。不能与数字的大小顺序混淆。

## 不能对 NULL 使用比较运算符

对比较运算符来说还有一点十分重要。那就是，作为查询条件的列中含有 NULL 的情况。例如，我们把进货单价 (shiire\_tanka) 作为查询条件。请注意，商品“叉子”和“圆珠笔”的进货单价是 NULL。

我们先来选取进货单价为 2800 日元 (shiire\_tanka = 2800) 的记录 (代码清单 2-25)。

代码清单 2-25 选取进货单价为 2800 日元的记录

```

SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka = 2800;

```

## 执行结果

shohin_mei	shiire_tanka
运动T恤	2800
菜刀	2800

大家对这个结果应该都没有疑问吧？接下来我们再尝试选取进出货单价不是2800日元（`shiire_tanka <> 2800`）的记录（代码清单2-26）。

## 代码清单2-26 选取进出货单价不是2800日元的记录

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka <> 2800;
```

## 执行结果

shohin_mei	shiire_tanka
T恤衫	500
打孔器	320
高压锅	5000
煎菜板	790

执行结果中并没有“叉子”和“圆珠笔”。这两条记录由于进货单价不明（NULL），无法判定为不是2800日元。

那如果想选取进货单价为NULL的记录的话，条件表达式该怎么写呢？历经一番苦思冥想后，用“`shire_tanka = NULL`”试了试，还是一条记录也取不出来。

## 代码清单2-27 错误的SELECT语句（一条记录也取不出来）

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka = NULL;
```

## 执行结果

shohin_mei	shiire_tanka
← 一条记录也没取到（0行）	

即使使用 `<>` 运算符也还是无法选取 NULL 的记录<sup>❶</sup>。因此，SQL 提供了专门用来判断是否为 NULL 的运算符 `IS NULL`。想要选取 NULL 的记录时，可以像代码清单 2-28 那样来书写条件表达式。

❶

SQL 不识别 “= NULL” 和 “<> NULL” 的理由将会在下一节（包含 NULL 情况下的真值）中进行说明。

代码清单2-28 选取NULL的记录

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka IS NULL;
```

执行结果

shohin_mei	shiire_tanka
叉子	
圆珠笔	

**KEYWORD**

● IS NOT NULL运算符

反之，希望选取不是 NULL 的记录时，需要使用 IS NOT NULL 运算符（代码清单 2-29）。

代码清单2-29 选取不为NULL的记录

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka IS NOT NULL;
```

执行结果

shohin_mei	shiire_tanka
T恤衫	500
打孔器	320
运动T恤	2800
菜刀	2800
高压锅	5000
煎菜板	790

**法则 2-9**

希望选取NULL记录时，需要在条件表达式中使用IS NULL运算符。希望选取不是NULL的记录时，需要在条件表达式中使用IS NOT NULL运算符。

除此之外，对 NULL 使用比较运算符的方法还有很多，详细内容将会在接下来的第 6 章中进行介绍。

## 2-3

## 逻辑运算符

## 学习重点

- 通过使用逻辑运算符，可以将多个查询条件进行组合。
- 通过NOT运算符可以作成“不是~”这样的查询条件。
- 两边条件都成立时，使用AND运算符的查询条件才成立。
- 只要两边的条件中有一个成立，使用OR运算符的查询条件就可以成立。
- 值可以归结为真(TRUE)和假(FALSE)其中之一，称为真值。比较运算符在比较成立时返回真，不成立时返回假。但是，在SQL中还存在另外一个特定的真值——不确定(UNKNOWN)。
- 根据逻辑运算符对真值进行的操作和其结果归结成的表称为真值表。
- SQL中的逻辑运算是包含对真、假和不确定进行运算的三值逻辑。

## NOT 运算符

在2-2节中我们介绍过，想要指定“不是~”这样的否定条件时，需要使用<>运算符。除此之外还存在另外一个表示否定，并且使用范围更广的运算符NOT。

NOT不能单独使用，必须和其他查询条件组合起来使用。例如，选取销售单价(hanbai\_tanka)大于等于1000日元的记录的SELECT语句如下所示(代码清单2-30)。

代码清单2-30 选取销售单价大于等于1000日元的记录

```
SELECT shohin_mei, shohin_bunrui, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka >= 1000;
```

执行结果

shohin_mei	shohin_bunrui	hanbai_tanka
T恤衫	衣服	1000
运动T恤	衣服	4000
菜刀	厨房用具	3000
高压锅	厨房用具	6800

## KEYWORD

- NOT运算符

向上述 SELECT 语句的查询条件中添加 NOT 运算符之后的结果如下所示（代码清单 2-31）。

代码清单 2-31 向代码清单 2-30 的查询条件中添加 NOT 运算符

```
SELECT shohin_mei, shohin_bunrui, hanbai_tanka
FROM Shohin
WHERE NOT hanbai_tanka >= 1000;
```

执行结果

shohin_mei	shohin_bunrui	hanbai_tanka
打孔器	办公用品	500
叉子	厨房用具	500
擦黑板	厨房用具	880
圆珠笔	办公用品	100

明白了吗？通过否定销售单价大于等于 1000 日元（hanbai\_tanka >= 1000）这个查询条件，就可以选取出销售单价小于 1000 日元的商品。也就是说，代码清单 2-31 中 WHERE 子句指定的查询条件，与代码清单 2-32 中 WHERE 子句指定的查询条件（hanbai\_tanka < 1000）是等价的<sup>①</sup>（图 2-5）。

①

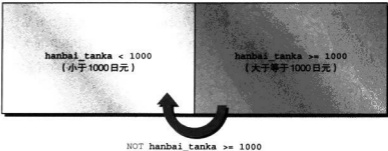
判定的结果相等。

代码清单 2-32 WHERE 子句的查询条件和代码清单 2-31 中的查询条件是等价的

```
SELECT shohin_mei, shohin_bunrui
FROM Shohin
WHERE hanbai_tanka < 1000;
```

图 2-5 使用 NOT 运算符时查询条件的变化

hanbai\_tanka(销售单价)



通过以上的例子大家可以发现，不使用 NOT 运算符也可以编写出效

果相同的查询条件。不仅如此，不使用 NOT 运算符的查询条件更容易让人理解。使用 NOT 运算符时，我们不得不每次都在脑海中进行“大于等于 1000 日元以上这个条件的否定结果就是小于 1000 日元”这样的转换。

虽然如此，但是也不能完全否定 NOT 运算符的作用。在编写复杂的 SQL 语句时，经常会看到 NOT 的身影。这里只是希望大家了解 NOT 运算符的书写方法和工作原理，同时提醒大家不要滥用该运算符。



#### 法则 2-10

NOT 运算符用来否定某一条件，但是不能滥用。

## AND 运算符和 OR 运算符

到目前为止，我们看到的每条 SQL 语句中都只有一个查询条件。但在实际使用当中，往往都是同时指定多个查询条件对数据进行查询的。例如，想要查询“商品种类为厨房用具、进货单价大于等于 5000 日元或小于 1000 日元”的商品等情况。

在 WHERE 子句中使用 AND 运算符或者 OR 运算符，可以对多个查询条件进行组合。

AND 运算符在其两侧的查询条件都成立时整个查询条件才成立。其意思相当于“并且”。

OR 运算符在其两侧的查询条件有一个成立时整个查询条件都成立。其意思相当于“或者”<sup>①</sup>。

例如，从 Shohin 表中选取“商品分类为厨房用具 (shohin\_bunrui = '厨房用具')，并且销售单价大于等于 3000 日元 (hanbai\_tanka >= 3000 日元) 的商品”的查询条件中就使用了 AND 运算符（代码清单 2-33）。

代码清单 2-33 在 WHERE 子句的查询条件中使用 AND 运算符

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shohin_bunrui = '厨房用具'
AND hanbai_tanka >= 3000;
```

### KEYWORD

- AND 运算符
- OR 运算符

### ①

需要注意的是，并不是只有一个条件成立时整个查询条件才成立，两个条件都成立时整个查询条件也同样成立。这与“到场的客人可以选择钥匙链或者迷你包作为礼品(任选其一)”中的“或者”有所不同。



## 执行结果

shohin_mei	shiire_tanka
菜刀	2800
高压锅	5000

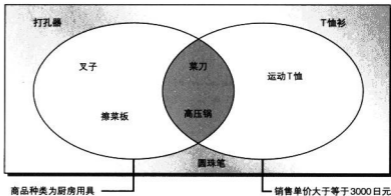
## KEYWORD

## ●文氏图

将集合(事物的聚集)的关系通过更加容易理解的图形进行可视化展示

该查询条件的文氏图如图 2-6 所示。左侧的圆圈代表符合查询条件“商品种类是厨房用具”的商品，右侧的圆圈代表符合查询条件“销售单价大于等于 3000 日元”的商品。两个圆重合的部分(同时满足两个查询条件的商品)就是通过 AND 运算符能够选取出的记录。

图 2-6 AND 运算符的工作效果图



选取“商品种类为厨房用具 (shohin\_bunrui = '厨房用具', 或者销售单价大于等于 3000 日元 (hanbai\_tanka >= 3000) 的商品)”的查询条件中使用了 OR 运算符 (代码清单 2-34)。

代码清单 2-34 在 WHERE 子句的查询条件中使用 OR 运算符

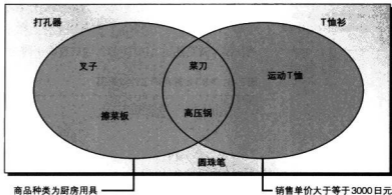
```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shohin_bunrui = '厨房用具'
OR hanbai_tanka >= 3000;
```

## 执行结果

shohin_mei	shiire_tanka
运动T恤	2800
菜刀	2800
高压锅	5000
叉子	
擦菜板	790

还是让我们来看看查询条件的文氏图吧（图2-7）。包含在左侧的圆圈（商品种类为厨房用具的商品）或者右侧的圆圈（销售单价大于等于3000日元的商品）中的部分（两个查询条件中满足任何一个的商品）就是通过OR运算符能够取出的记录。

图2-7 OR运算符的工作效果图



通过文氏图可以方便地确认由多个条件组合而成的复杂的SQL语句的查询条件，大家可以多多加以利用。



#### 法则2-11

多个查询条件进行组合时，需要使用AND运算符或者OR运算符。



#### 法则2-12

文氏图很方便。

## 通过括号进行强化

接下来我们尝试书写稍微复杂一些的查询条件。例如，使用下面的查询条件对Shohin表进行查询的SELECT语句，其WHERE子句的条件表达式该怎么写呢？

“商品种类是办公用品”

并且

“登记日期是 2009 年 9 月 11 日或者 2009 年 9 月 20 日”

满足上述查询条件的商品 (shohin\_mei) 只有“打孔器”。

把上述查询条件原封不动地写入 WHERE 子句中，得到的 SELECT 语句似乎就可以满足需求了（代码清单 2-35）。

代码清单 2-35 将查询条件原封不动地写入条件表达式

```
SELECT shohin_mei, shohin_bunrui, torokubi
FROM Shohin
WHERE shohin_bunrui = '办公用品'
AND torokubi = '2009-09-11'
OR torokubi = '2009-09-20';
```

让我们马上执行上述 SELECT 语句试试看……，却得到了下面这样的错误结果。

执行结果

shohin_mei	shohin_bunrui	torokubi
T 恤衫	衣服	2009-09-20
打孔器	办公用品	2009-09-11
菜刀	厨房用具	2009-09-20
叉子	厨房用具	2009-09-20

不想要的 T 恤衫和菜刀也被选出来了，真是头疼呀。到底为什么会得到这样的结果呢？

这是由于 AND 运算优先于 OR 运算执行所造成的。代码清单 2-35 中的条件表达式会被解释成下面这样。

```
「shohin_bunrui = '办公用品' AND torokubi = '2009-09-11」
OR
「torokubi = '2009-09-20」
```

也就是，

“商品种类是办公用品，并且登记日期是 2009 年 9 月 11 日”

或者

“登记日期是 2009 年 9 月 20 日”

这和想要指定的查询条件并不相符。想要优先执行 OR 运算时，可以像代码

**KEYWORD**

● (1)

清单 2-36 那样使用半角括号将 OR 运算符及其两侧的查询条件括起来。

代码清单 2-36 通过使用括号让 OR 运算先于 AND 运算执行

```
SELECT shohin_mei, shohin_bunrui, torokubi
FROM Shohin
WHERE shohin_bunrui = '办公用品'
      AND ( torokubi = '2009-09-11'
          OR torokubi = '2009-09-20');
```

执行结果

shohin_mei	shohin_bunrui	torokubi
打孔器	办公用品	2009-09-11

这样就选取出了想要得到的“打孔器”。

**法则 2-13**

AND 运算符的优先级高于 OR 运算符。想要优先执行 OR 运算时可以使用括号。

**KEYWORD**

- 逻辑运算符
- 真值
- 真 (TRUE)
- 假 (FALSE)

**①**

但是在 SQL 中还 存在“不确定 (UNKNOWN)”这样的值。接下来会进行详细说明。

**②**

算术运算符返回的结果是数字。除了返回结果的类型不同之外，和比较运算符一样都会返回运算结果。

**逻辑运算符和真值**

本节中介绍的三个运算符 NOT、AND 和 OR 称为逻辑运算符。这里所说的逻辑就是对真值进行操作的意思。真值就是值为真 (TRUE) 或假 (FALSE) 其中之一值<sup>①</sup>。

上一节介绍的比较运算符，会把运算的结果以真值的形式进行返回。比较结果成立时返回真 (TRUE)，比较结果不成立时返回假 (FALSE)<sup>②</sup>。例如，对于 hanbai\_tanka >= 3000 这个查询条件来说，由于 shohin\_mei 列为“运动 T 恤”的记录 hanbai\_tanka 列的值是 2800，因此会返回假 (FALSE)，而 shohin\_mei 列为“高压锅”的记录 hanbai\_tanka 列的值是 5000，所以返回真 (TRUE)。

逻辑运算符对比较运算符等返回的真值进行操作。AND 运算符两侧的真值都为真时返回真，除此之外都返回假。OR 运算符两侧的真值只要有一个不为假就返回真，只有当其两侧的真值都为假时才返回假。NOT 运算符只是单纯的将真转换为假，将假转换为真。真值表 (truth table) 就是对这类操作及其结果进行的总结 (表 2-4)。

表2-4 真值表

AND			OR			NOT	
P	Q	P AND Q	P	Q	P OR Q	P	NOT P
真	真	真	真	真	真	真	假
真	假	假	真	假	真	假	真
假	真	假	假	真	真		
假	假	假	假	假	假		

请将表 2-4 中的 P 和 Q 想象为“销售单价为 500 日元”这样的条件。逻辑运算的结果只有真和假两种，对其进行排列组合将会得到  $2 \times 2 = 4$  种结果。

在 SELECT 语句的 WHERE 子句中，通过 AND 运算符将两个查询条件连接起来时，会查询出这两个查询条件都为真的记录。通过 OR 运算符将两个查询条件连接起来时，会查询出某一个查询条件为真或者两个查询条件都为真的记录。在条件表达式中使用 NOT 运算符时，会选取出查询条件为假的记录（反过来为真）。

虽然表 2-4 中的真值表只是使用一个逻辑运算符时得到的结果，但即使使用两个以上逻辑运算符连接三个以上查询条件时，通过对逻辑运算返回的真值进行组合，不论多复杂的条件也可以得到相应的结果。

表 2-5 就是根据之前例子中的查询条件“商品种类为办公用品”，并且“登记日期为 2009 年 9 月 11 日或者 2009 年 9 月 20 日”（shohin\_bunrui = '办公用品' AND (torokubi = '2009-09-11' OR torokubi = '2009-09-20'））做成的真值表。

表2-5 查询条件为P AND (Q OR R)的真值表

P AND (Q OR R)				
P	Q	R	Q OR R	P AND (Q OR R)
真	真	真	真	真
真	真	假	真	真
真	假	真	真	真
真	假	假	假	假
假	真	真	真	假
假	真	假	真	假
假	假	真	真	假
假	假	假	假	假

P: 商品种类为办公用品  
 Q: 登记日期为 2009 年 9 月 11 日  
 R: 登记日期为 2009 年 9 月 20 日  
 Q OR R: 登记日期为 2009 年 9 月 11 日或者 2009 年 9 月 20 日  
 P AND (Q OR R): 商品种类为办公用品，并且，登记日期为 2009 年 9 月 11 日或者 2009 年 9 月 20 日

代码清单 2-36 中的 SELECT 语句,查询出了唯一满足 P AND(Q OR R) 为真的记录“打孔器”。



#### 法则 2-14

通过创建真值表,无论多复杂的条件,都会更容易理解。

## 专 栏

### 逻辑积与逻辑和

将表 2-4 的真值表中的真变为 1、假变为 0,意外地得到了下述规则。

表 2-A 真为 1、假为 0 的真值表

AND (逻辑积)				OR (逻辑和)				NOT		
P	Q	积	P AND Q	P	Q	和	P OR Q	P	反转	NOT P
1	1	1×1	1	1	1	1+1	1	1	1→0	0
1	0	1×0	0	1	0	1+0	1	0	0→1	1
0	1	0×1	0	0	1	0+1	1			
0	0	0×0	0	0	0	0+0	0			

NOT 运算符并没有什么特别的改变,但是 AND 运算的结果与乘法运算(积),OR 运算的结果与加法运算(和)的结果却是一样的<sup>①</sup>。因此,使用 AND 运算符进行的逻辑运算称为逻辑积,使用 OR 运算符进行的逻辑运算称为逻辑和。

#### ①

严格来说,此处的 1+1=1 与通常的整数运算并不相同。只是因为真值中只存在 0 和 1 两种情况,所以才有了这样的结果。

#### KEYWORD

- 逻辑积
- 逻辑和

## 含有 NULL 时的真值

上一节我们介绍了查询 NULL 时不能使用比较运算符 (= 或者 <>),需要使用 IS NULL 运算符或者 IS NOT NULL 运算符。实际上,使用逻辑运算符时也需要特别对待 NULL。

我们来看一下 Shohin (商品) 表,商品“叉子”和“圆珠笔”的进货单价 (shiire\_tanka) 为 NULL。那么,对这两条记录使用查询条件 shiire\_tanka = 2800 (进货单价为 2800 日元) 会得到什么样的真值呢? 如果结果为真,则通过该条件表达式就可以选取出“叉子”和“圆珠笔”这两条记录。但是在之前介绍“不能对 NULL 使用比较运算符”时,

我们就知道结果并不是这样的，也就是说结果不为真。

那结果会为假吗？实际上结果也不是假。如果结果为假，那么对于否定后的条件 `NOT shire_tanka = 2800`（进货单价不是 2800 日元）来说，结果应该为真，就能选取这两条记录了（因为假的对立面为真）。实际结果却并不是这样。

既不是真也不是假，那结果到底是什么呢？其实这是 SQL 中特有的情况。这时真值是除真假之外的第三种值——不确定（UNKNOWN）。一般的逻辑运算并不存在这第三种值。SQL 之外的语言也基本上只使用真和假这两种真值。与通常的逻辑运算被称为二值逻辑相对，只有 SQL 中的逻辑运算被称为三值逻辑。

因此，表 2-4 中的真值表并不完整。完整的真值表应该像表 2-6 这样包含“不确定”这个值。

表 2-6 三值逻辑中的 AND 和 OR 真值表

AND			OR		
P	Q	P AND Q	P	Q	P OR Q
真	真	真	真	真	真
真	假	假	真	假	真
真	不确定	不确定	真	不确定	真
假	真	假	假	真	真
假	假	假	假	假	假
假	不确定	假	假	不确定	不确定
不确定	真	不确定	不确定	真	真
不确定	假	假	不确定	假	不确定
不确定	不确定	不确定	不确定	不确定	不确定

#### KEYWORD

- 不确定
- 二值逻辑
- 三值逻辑

## 专栏

### Shohin 表中设置 NOT NULL 约束的原因

原本只有 4 行的真值表，如果要考虑 NULL 的话就会像表 2-6 那样增加为  $3^3=9$  行，看起来也变得更为繁琐。考虑 NULL 时的条件判断也会变得异常复杂，这与我们希望的结果大相径庭。因此，数据库领域的有识之士们达成了“尽量不使用 NULL”的共识。

这就是为什么在创建 Shohin 表时，要给某些列设定 NOT NULL 约束（禁止录入 NULL）的原因。

## 练习题

2.1 编写一条 SQL 语句,从 Shohin(商品)表中选取出“登记日期(torokubi)在 2009 年 4 月 28 日之后”的商品。查询结果要包含 shohin\_mei 和 torokubi 两列。

2.2 请说出对 Shohin 表执行如下 3 条 SELECT 语句时的返回结果。

```
① SELECT *
   FROM Shohin
   WHERE shiire_tanka = NULL;
```

```
② SELECT *
   FROM Shohin
   WHERE shiire_tanka <> NULL;
```

```
③ SELECT *
   FROM Shohin
   WHERE shohin_mei > NULL;
```

2.3 代码清单 2-22 中的 SELECT 语句能够从 Shohin 表中取出“销售单价(hanbai\_tanka)比进货单价(shiire\_tanka)高出 500 日元以上”的商品。请写出两条可以得到相同结果的 SELECT 语句。执行结果如下所示。

执行结果

shohin_mei	hanbai_tanka	shiire_tanka
T恤衫	1000	500
运动T恤	4000	2800
高压锅	6800	5000

2.4 请写出一条 SELECT 语句,从 Shohin 表中选取出满足“销售单价打九折之后利润高于 100 日元的办公用品和厨房用具”条件的记录。查询结果要包括 shohin\_mei 列、shohin\_bunrui 列以及销售单价打九折之后的利润(别名设定为 rieki)。

提示:销售单价打九折,可以通过 hanbai\_tanka 列的值乘以 0.9 获得。利润可以通过该值减去 shiire\_tanka 列的值获得。



## 第3章 聚合与排序

对表进行聚合查询

对表进行分组

为聚合结果指定条件

对查询结果进行排序



SQL

## 本章重点

---

随着表中记录（数据行）的不断积累，存储数据逐渐增加，有时我们可能希望使用某些合计操作计算出这些数据的合计值或者平均值等。我们将在本章中学习使用 SQL 语句进行合计操作的方法。此外，我们还会学习在合计操作时指定条件，以及对合计结果进行升序、降序的排序方法。

### 3-1 对表进行聚合查询

#### ■ 聚合函数

■ 计算表中数据的行数

■ 计算 NULL 以外数据的行数

■ 计算合计值

■ 计算平均值

■ 计算最大值和最小值

■ 使用聚合函数删除重复值（关键字 DISTINCT）

### 3-2 对表进行分组

■ GROUP BY 子句

■ 聚合键中包含 NULL 的情况

■ 使用 WHERE 子句时 GROUP BY 的执行结果

■ 与聚合函数和 GROUP BY 子句有关的常见错误

### 3-3 为聚合结果指定条件

■ HAVING 子句

■ HAVING 子句的构成要素

■ 相对于 HAVING 子句，更适合写在 WHERE 子句中的条件

### 3-4 对查询结果进行排序

■ ORDER BY 子句

■ 指定升序或者降序

■ 指定多个排序键

■ NULL 的顺序

■ 在排序键中使用显示用别名

■ ORDER BY 子句中可以使用列

■ 不要使用列编号

## 3-1

该列的最小值

## 对表进行聚合查询

## 学习重点

- 使用聚合函数对表中的列进行合计值或者平均值等合计操作。
- 通常，聚合函数会对 NULL 以外的对象进行合计。但是只有 COUNT 函数例外，使用 COUNT(\*) 可以查出包含 NULL 在内的全部数据行数。
- 使用 DISTINCT 关键字删除重复值

## KEYWORD

- 函数
- COUNT 函数

## 聚合函数

通过 SQL 对数据进行某种操作或计算时需要使用函数。例如，计算表中全部数据行数时，可以使用 COUNT 函数。该函数就是使用 COUNT(计数) 来命名的。除此之外，SQL 中还有很多其他用于合计的函数，请大家先记住以下 5 个常用的函数。

COUNT: 计算表中的记录数(行数)。

SUM: 计算表中数值列的数据合计值。

AVG: 计算表中数值列的数据平均值。

MAX: 求出表中任意列中数据的最大值。

MIN: 求出表中任意列中数据的最小值。

## KEYWORD

- 聚合函数
- 集合函数
- 聚合

如上所示，用于合计的函数称为聚合函数或者集合函数。本书中统称为聚合函数。所谓聚合，就是将多行汇总为一行。实际上，所有的聚合函数都是这样，输入多行输出一行。

接下来，本章将继续使用在第 1 章中创建的 Shohin 表(图 3-1)来学习函数的使用方法。

图 3-1 Shohin 表的内容

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T 恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

该列的最小值

(续表)

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0003	运动T恤	衣服	4000	2800	NULL
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500	NULL	2009-09-20
0007	煎菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	NULL	2009-11-11

该列的最小值      该列的最大值

## 计算表中数据的行数

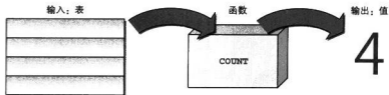
首先，我们以 COUNT 函数为例让大家对函数形成一个初步印象。函数这个词，与我们在学校数学课上学到的意思是一样的，就像是输入某个值就能输出相应结果的盒子一样<sup>①</sup>。

①

函数中的函就是盒子的意思。

使用 COUNT 函数时，输入表的列，就能够输出数据行数。如图 3-2 所示，将表中的列放入名称为 COUNT 的盒子中，咔嚓咔嚓地进行计算，咕咚一下行数就出来了……就像自动售货机那样，很容易理解吧。

图 3-2 COUNT 函数的操作演示图



接下来让我们看一下 SQL 中的具体书写方法。COUNT 函数的语法本身非常简单，像代码清单 3-1 那样写在 SELECT 子句中就可以得到表中全部的数据行数了。

代码清单 3-1 计算全部数据行数

```
SELECT COUNT(*) 参数 (parameter)
FROM Shohin;
```

## 执行结果

```
count
-----
      8 ← 返回值
```

COUNT ( ) 中的星号, 我们在 2-1 节中已经学过, 代表全部列的意思。COUNT 函数的输入值就记述在其后的括号中。

此处的输入值称为参数或者 parameter, 输出值称为返回值。这些称谓不仅本书会使用, 在多数编程语言使用函数时都会频繁出现, 请大家牢记。

## KEYWORD

- 参数 (parameter)
- 返回值

## 计算 NULL 以外数据的行数

想要计算表中全部数据行数时, 可以像 SELECT COUNT (\*) 这样使用星号。如果想得到 shiire\_tanka 列 (进货单价) 中非空行数的话, 可以像代码清单 3-2 那样, 通过将对象列设定为参数来实现。

## 代码清单 3-2 计算 NULL 之外的数据行数

```
SELECT COUNT(shiire_tanka)
FROM Shohin;
```

## 执行结果

```
count
-----
      6
```

此时, 如图 3-1 所示, shiire\_tanka 列中有两行数据是 NULL, 因此并不应该计算这两行。对于 COUNT 函数来说, 参数列不同计算的结果也会发生变化, 这一点请大家特别注意。为了有助于大家理解, 请看一下如下这个只包含 NULL 的表的极端例子。

图 3-3 只包含 NULL 的表

NullTbl 表	
列1 (col_1)	
	} NULL

我们来看一下针对上述表，将星号(\*)和列名作为参数传递给COUNT函数所得到的结果(代码清单3-3)。

**代码清单3-3** 将包含NULL的列作为参数时，COUNT(\*)和COUNT(<列名>)的结果并不相同

```
SELECT COUNT(*), COUNT(col_1)
FROM NullTbl;
```

执行结果

count	count
3	0

count (\*) 的结果

count (col\_1) 的结果

如上所示，即使对同一个表使用COUNT函数，输入的参数不同得到的结果也会不同。由于将列名作为参数时会得到NULL之外的数据行数，所以得到的结果是0行。

该特性是COUNT函数所特有的，其他函数并不能将星号作为参数(如果使用星号会出错)。



#### 法则 3-1

COUNT函数的结果根据参数的不同而不同。COUNT(\*)会得到包含NULL的数据行数，而COUNT(<列名>)会得到NULL之外的数据行数。

## 计算合计值

接下来我们学习其他4个聚合函数的使用方法。这些函数的语法基本上与COUNT函数相同。但就像我们此前所说的那样，在这些函数中不能使用星号作为参数。

首先，我们使用计算合计值的SUM函数，求出销售单价的合计值(代码清单3-4)。

**代码清单3-4** 计算销售单价的合计值

```
SELECT SUM(hanbai_tanka)
FROM Shohin;
```

### KEYWORD

● SUM函数

## 执行结果

```
sum
-----
16780
```

得到的结果 16780 日元，是所有销售单价 (hanbai\_tanka 列) 的合计，与下述计算式的结果相同。

```

    1,000
      500
    4,000
    3,000
    6,800
      500
      880
+     100
-----
   16,780
```

接下来，我们将销售单价和进货单价 (shire\_tanka 列) 的合计值一起计算出来 (代码清单 3-5)。

## 代码清单 3-5 计算销售单价和进货单价的合计值

```
SELECT SUM(hanbai_tanka), SUM(shire_tanka)
FROM Shohin;
```

## 执行结果

```
sum | sum
-----|-----
16780 | 12210
```

SUM(shire\_tanka) 的结果

SUM(hanbai\_tanka) 的结果

这次我们通过 SUM (shire\_tanka) 将进货单价的合计值也一起计算出来了，但有一点需要大家注意。具体的计算过程如下所示。

```

    500
    320
   2,800
   2,800
   5,000
    790
   NULL
+   NULL
-----
   12,210
```

大家都已经注意到了吧，与销售单价不同，进货单价中有两条不明数据 NULL。对于 SUM 函数来说，即使包含 NULL 也可以计算出合计值。还记得前一章内容的读者可能会产生如下疑问。

“四则运算中如果存在 NULL 时，结果一定是 NULL。那此时进货单价的合计值会不会也是 NULL 呢？”

有这样疑问的读者思维很敏锐，但实际上这两者并不矛盾。从结果上说，所有的聚合函数，如果以列名为参数，那么在计算之前就已经把 NULL 排除在外了。因此，无论有多少个 NULL 都会被无视。这与“等价为 0”并不相同<sup>①</sup>。

因此，上述进货单价的计算表达式，实际上应该如下所示。

500	
320	
2,800	
2,800	
5,000	
+ 790	
12,210	← NULL并不在计算表达式之中

### ①

虽然使用 SUM 函数时，“将 NULL 除外”和“等同于 0”的结果相同，但使用 AVG 函数时，这两种情况的结果就完全不同了。接下来我们会详细介绍在 AVG 函数中使用包含 NULL 的列作为参数的例子。



### 法则 3-2

聚合函数会将 NULL 排除在外。但 COUNT(\*) 例外，并不会排除 NULL。

## 计算平均值

接下来，我们练习一下计算多行数据的平均值。为此，我们需要使用 AVG 函数。其语法和 SUM 函数完全相同（代码清单 3-6）。

### KEYWORD

● AVG 函数

#### 代码清单 3-6 计算销售单价的平均值

```
SELECT AVG(hanbai_tanka)
FROM Shohin;
```

#### 执行结果

```
-----
avg
2097.5000000000000000
```



平均值的计算表达式如下所示。

$$\frac{1,000+500+4,000+3,000+6,800+500+880+100}{8}$$

(值的合计) / (值的个数) 就是平均值的计算公式了。下面我们也像使用 SUM 函数那样, 计算一下包含 NULL 的进货单价的平均值 (代码清单 3-7)。

代码清单 3-7 计算销售单价和进货单价的平均值

```
SELECT AVG(hanbai_tanka), AVG(shiire_tanka)
FROM Shohin;
```

执行结果

avg	avg
2097.5000000000000000	2035.0000000000000000

↑ AVG(hanbai\_tanka)的结果      ↑ AVG(shiire\_tanka)的结果

计算进货单价平均值的情况与 SUM 函数相同, 会事先删除 NULL 再进行计算。因此计算表达式如下所示。

$$\frac{500+320+2,800+2,800+5,000+790}{6} = 2035$$

需要注意的是分母是 6 而不是 8。减少的两个也就是那两条 NULL 的数据。

但是有时也将 NULL 作为 0 进行计算, 具体的实现方式请参考第 6 章。

$$\frac{500+320+2,800+2,800+5,000+790(+0+0)}{8} = 1526.25$$

将 NULL 改变为 0

## 计算最大值和最小值

想要计算出多条记录中的最大值或最小值, 可以分别使用 MAX 和 MIN 函数。它们是英语 maximum (最大值) 和 minimum (最小值) 的简称, 很容易记住吧。

### KEYWORD

- MAX 函数
- MIN 函数

这两个函数的语法与 SUM 的语法相同，使用时需要将列作为参数（代码清单 3-8）。

代码清单 3-8 计算销售单价的最大值和进货单价的最小值

```
SELECT MAX(hanbai_tanka), MIN(shiire_tanka)
FROM Shohin;
```

执行结果

max	min
6800	320

MIN(shiire\_tanka)的结果

MAX(hanbai\_tanka)的结果

如图 3-1 所示，我们取得了相应的最大值和最小值。

但是，MAX/MIN 函数和 SUM/AVG 函数有一点不同，那就是 SUM/AVG 函数只能对数值类型的列使用，而 MAX/MIN 函数原则上可以适用于任何数据类型的列。例如，对图 3-1 中日期类型的列 torokubi 使用 MAX/MIN 函数的结果如下所示（代码清单 3-9）。

代码清单 3-9 计算登记日期的最大值和最小值

```
SELECT MAX(torokubi), MIN(torokubi)
FROM Shohin;
```

执行结果

max	min
2009-11-11	2008-04-28

MAX(torokubi)的结果

MIN(torokubi)的结果

刚刚我们说过 MAX/MIN 函数适用于任何数据类型的列，也就是说，如果是能够排序的数据，就肯定有最大值和最小值，也就能够使用这两个函数。对日期来说，平均值和合计值并没有什么实际意义，因此不能使用 SUM/AVG 函数。这点对于字符串类型的数据也适用，字符串类型的数据能够使用 MAX/MIN 函数，但不能使用 SUM/AVG 函数。



### 法则 3-3

MAX/MIN 函数几乎适用于所有数据类型的列。SUM/AVG 函数只适用于数值类型的列。

**KEYWORD**

●DISTINCT 关键字

## 使用聚合函数删除重复值(关键字DISTINCT)

接下来我们考虑一下下面这种情况。

在图 3-1 中我们可以看到,商品种类(shohin\_bunrui 列)和销售单价(hanbai\_tanka 列)的数据中,存在多行数据相同的情况。

例如,拿商品种类来说,表中总共有 8 行 3 种商品数据,其中衣服 2 行,办公用品 2 行,厨房用具 4 行。如果想要计算出商品种类的个数,怎么办呢比较好呢?删除重复数据然后再计算数据行数似乎是个不错的办法。实际上,在使用 COUNT 函数时,将我们在 2-1 节中介绍过的 DISTINCT 关键字作为参数,就能得到我们想要的结果了(代码清单 3-10)。

代码清单 3-10 计算去除重复数据后的数据行数

```
SELECT COUNT(DISTINCT shohin_bunrui)
FROM Shohin;
```

执行结果

```
count
-----
3
```

请注意,这时 DISTINCT 必须写在括号中。这是因为必须要在计算行数之前删除 shohin\_bunrui 列中的重复数据。如果像代码清单 3-11 那样写在括号外的话,就会先计算出数据行数,然后再删除重复数据,结果就得到了 shohin\_bunrui 列的所有行数(也就是 8)。

代码清单 3-11 先计算数据行数再删除重复数据的结果

```
SELECT COUNT(shohin_bunrui)
FROM Shohin;
```

执行结果

```
count
-----
8
```



### 法则 3-4

想要计算值的种类时,可以在 COUNT 函数的参数中使用 DISTINCT。

DISTINCT 不仅限于 COUNT 函数，所有的聚合函数都可以使用。下面我来看一下使用 DISTINCT 和不使用 DISTINCT 时 SUM 函数的执行结果（代码清单 3-12）。

代码清单 3-12 使不使用 DISTINCT 时的动作差异 (SUM 函数)

```
SELECT SUM(hanbai_tanka), SUM(DISTINCT hanbai_tanka)
FROM Shohin;
```

执行结果

sum	sum
16780	16280

SUM(DISTINCT hanbai\_tanka) 的结果

SUM(hanbai\_tanka) 的结果

左侧是未使用 DISTINCT 时的合计值，和我们之前计算的结果相同，都是 16780 日元。右侧是使用 DISTINCT 后的合计值，比之前的结果少了 500 日元。这是因为表中销售单价为 500 日元的商品有两种——“打孔器”和“叉子”，在删除重复数据之后，计算对象就只剩下一条记录了。



#### 法则 3-5

在聚合函数的参数中使用 DISTINCT，可以删除重复数据。

## 3-2

该列的最大值

## 对表进行分组

## 学习重点

- 使用 GROUP BY 子句可以像切蛋糕那样将表进行分割。通过使用聚合函数和 GROUP BY 子句, 可以根据“商品种类”或者“登记日期”等将表分割后再进行聚合。
- 聚合键中包含 NULL 时, 在结果中会以“不确定”行(空行)的形式表现出来。
- 使用聚合函数和 GROUP BY 子句时需要注意以下 4 点。
  - ① 只能写在 SELECT 子句之中
  - ② GROUP BY 子句中不能使用 SELECT 子句中列的别名
  - ③ GROUP BY 子句的聚合结果是无序的
  - ④ WHERE 子句中不能使用聚合函数

## GROUP BY 子句

目前为止, 我们看到的聚合函数的使用方法, 无论是否包含 NULL, 无论是否删除了重复数据, 都是针对表中的所有数据进行的聚合处理。下面, 我们先把表分成几组, 然后再进行聚合处理。也就是按照“商品种类”、“登记日期”等进行聚合。

这里我们将要第一次接触到 GROUP BY 子句。语法结构如下所示。

## KEYWORD

● GROUP BY 子句

## 语法 3-1 使用 GROUP BY 子句进行聚合

```
SELECT <列名1>, <列名2>, <列名3>, .....
FROM <表名>
GROUP BY <列名1>, <列名2>, <列名3>, .....;
```

下面我们就按照商品种类来统计一下数据行数 (= 商品数量) (代码清单 3-13)。

代码清单 3-13 按照商品种类统计数据行数

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui;
```

执行结果

shohin_bunrui	count
衣服	2
办公用品	2
厨房用具	4

如上所示，未使用 GROUP BY 子句时，结果只有 1 行，而这次的结果却是多行。这是因为不使用 GROUP BY 子句时，是将表中的所有数据作为一组来对待的。而使用 GROUP BY 子句时，会将表中的数据分为多个组进行处理。如图 3-4 所示，GROUP BY 子句对表进行了切分。

图 3-4 按照商品种类对表进行切分



这样，GROUP BY 子句就像切蛋糕那样将表进行了分组。在 GROUP BY 子句中指定的列称为聚合键或者分组列。由于能够决定表的切分方式，所以是非常重要的列。当然，GROUP BY 子句也和 SELECT 子句一样，可以通过逗号分隔指定多列。

如果用画线的方式来切分表中数据的话，就会得到图 3-5 那样以商品种类为界线的三组数据。然后再计算每种商品的数据行数，就能得到相应的结果了。

**KEYWORD**

- 聚合键
- 分组列

图3-5 使用商品种类对表进行切分

shohin_bunrui (商品分类)	shohin_mei (商品名称)	shohin_id (商品编号)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
衣服	T恤衫	0001	1000	500	2009-09-20
	运动T恤	0003	4000	2800	
办公用品	打孔器	0002	500	320	2009-09-11
	圆珠笔	0008	100		2009-11-11
厨房用具	菜刀	0004	3000	2800	2009-09-20
	高压锅	0005	6800	5000	2009-01-15
	叉子	0006	500		2009-09-20
	擦菜板	0007	880	790	2008-04-28



## 法则 3-6

GROUP BY 就像是切分表的一把刀。

此外，GROUP BY 子句的书写位置也有严格要求，一定要写在 FROM 语句之后（如果有 WHERE 子句的话需要写在 WHERE 子句之后）。如果无视子句的书写顺序，SQL 就一定会无法正常执行，而出错。目前 SQL 的子句还没有全部登场，已经出现的各子句的暂定顺序如下所示。

## ► 子句的书写顺序（暂定）

1. SELECT → 2. FROM → 3. WHERE → 4. GROUP BY



## 法则 3-7

SQL 子句的顺序不能改变，也不能互相替换。

## 聚合键中包含 NULL 的情况

接下来我们将进货单价(shiire\_tanka)作为聚合键对表进行切分。在 GROUP BY 子句中指定进货单价的结果请参见代码清单 3-14。

### 代码清单 3-14 按照进货单价统计数据行数

```
SELECT shiire_tanka, COUNT(*)
FROM Shohin
GROUP BY shiire_tanka;
```

上述 SELECT 语句的结果如下所示。

执行结果

shire_tanka	count
	2
320	1
500	1
5000	1
2800	2
790	1

聚合键为NULL的结果

对于像 790 日元或者 500 日元这样，进货单价很清楚的数据行来说不会有什么问题，结果与之前的情况相同。问题是结果中的第一行，也就是进货单价为 NULL 的组。从结果我们可以看出，当聚合键中包含 NULL 时，也会将 NULL 作为一组特定的数据。如图 3-6 所示。

图 3-6 使用进货单价对表进行切分



这里的 NULL，大家可以理解为“不确定”。



### 法则 3-8

聚合键中包含 NULL 时，在结果中会以“不确定”行（空行）的形式表现出来。

## 使用 WHERE 子句时 GROUP BY 的执行结果

在使用了 GROUP BY 子句的 SELECT 语句中，也可以正常使用 WHERE 子句。子句的排列顺序如前所述，语法结果如下所示。



## 语法 3-2 使用 WHERE 子句和 GROUP BY 子句进行聚合处理

```
SELECT <列名 1>, <列名 2>, <列名 3>, .....
FROM <表名>
WHERE
GROUP BY <列名 1>, <列名 2>, <列名 3>, .....;
```

像这样使用 WHERE 子句进行聚合处理时，会先根据 WHERE 子句指定的条件进行过滤，然后再进行聚合处理。请大家看一下代码清单 3-15 中的例文。

## 代码清单 3-15 同时使用 WHERE 子句和 GROUP BY 子句

```
SELECT shiire_tanka, COUNT(*)
FROM Shohin
WHERE shohin_bunrui = '衣服'
GROUP BY shiire_tanka;
```

上述的 SELECT 语句，由于首先使用了 WHERE 子句对记录进行过滤，所以实际上作为聚合对象的记录如表 3-1 所示，只有 2 行。

表 3-1 WHERE 子句过滤的结果

shohin_bunrui (商品分类)	shohin_mei (商品名称)	shohin_id (商品编号)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
衣服	T 恤衫	0001	1000	500	2009-09-20
衣服	运动T恤	0003	4000	2800	

使用进货单价对这两条记录进行分组，就得到了如下的执行结果。

## 执行结果

```
shiire_tanka | count
-----+-----
          500 |      1
          2800 |      1
```

GROUP BY 和 WHERE 并用时，SELECT 语句的执行顺序如下所示。

## ► GROUP BY 和 WHERE 并用时 SELECT 语句的执行顺序

FROM → WHERE → GROUP BY → SELECT

这之前语法 3-2 中的说明顺序有些不同。这是由于在 SQL 语句中，

书写顺序和 DBMS 内部的执行顺序并不相同造成的。这也是 SQL 难以理解的原因之一。对习惯了英语思维方式的欧美人来说，可能会觉得这样的执行顺序是很自然的，但对于习惯了从前往后阅读习惯的日本人来说，一定要多加注意。

## 与聚合函数和 GROUP BY 子句有关的常见错误

截至目前，我们已经学习了聚合函数和 GROUP BY 子句的基本使用方法。虽然由于使用方便而经常被使用，但是书写 SQL 时却很容易出错，希望大家特别小心。

### ■ 常见错误①——在 SELECT 子句中书写了多余的列

在使用 COUNT 这样的聚合函数时，SELECT 子句中的元素有严格的限制。实际上，使用聚合函数时，SELECT 子句中只能存在以下三种元素。

- 常数
- 聚合函数
- GROUP BY 子句中指定的列名（也就是聚合键）

第 1 章中我们介绍过，常数就是像数字 123，或者字符串‘测试’这样写在 SQL 语句中的固定值。将常数直接写在 SELECT 子句中没有任何问题。此外还可以书写聚合函数或者聚合键，这些在之前的示例代码中都已经出现过了。

这里经常会出现的错误就是把聚合键之外的列名书写在 SELECT 子句之中。例如代码清单 3-16 中的 SELECT 语句就会发生错误，无法正常执行。

#### 代码清单 3-16 在 SELECT 子句中书写聚合键之外的列名会发生错误

```
SELECT shohin_mei, shiire_tanka, COUNT(*)
FROM Shohin
GROUP BY shiire_tanka;
```

#### 执行结果（使用 PostgreSQL 的情况）

```
ERROR: 列"shohin,shohin_mei"必须包含在GROUP BY子句之中，或者必须在聚合函数内使用
行 1: SELECT shohin_mei, shiire_tanka, COUNT(*)
```

⇒ 表示下一行接续本行，只是由于版面所限而换行。

## ❶

由于在MySQL中这样的语法也得到了认同，所以能够执行，不会发生错误！在多列统计中只要有一列满足要求就可以了。但是MySQL以外的DBMS都不支持这样的语法，因此请不要使用这样的写法。

列名 `shohin_mei` 并没有包含在 `GROUP BY` 子句当中。因此，该列名也不能书写在 `SELECT` 子句之中<sup>❶</sup>。

不支持这种语法的原因，大家仔细想一想应该就明白了。通过某个聚合键将表分组之后，结果中的一行数据就代表一组。例如，使用进货单价将表进行分组之后，一行就代表了一个进货单价。问题就出在这里，聚合键和商品名并不一定是一对一的。

例如，进货单价是2800日元的商品有“运动T恤”和“菜刀”两种，但是2800日元这一行应该对应哪个商品名呢（图3-7）。如果规定了哪种商品优先表示的话则另当别论，但其实并没有这样的规则。

图3-7 聚合键和商品名不是一对一的情况

shohin_mei	shiire_tanka	count
...	2800	2

↑ 这里应该显示什么好呢？

像这样与聚合键相对应的、同时存在多个值的列出现在 `SELECT` 子句中的情况，理论上是不可能的。



### 法则3-9

使用 `GROUP BY` 子句时，`SELECT` 子句中不能出现聚合键之外的列名。

## ❷

大家需要注意的是，虽然这样的写法在PostgreSQL和MySQL都不会发生执行错误，但是这并不是通常的使用方法

### ■ 常见错误②——在 `GROUP BY` 子句中写了列的别名

这也是一个非常常见的错误。在2-2节中我们学过，`SELECT` 子句中的项目可以通过 `AS` 关键字来指定别名。但是，在 `GROUP BY` 子句中是不能使用别名的。代码清单3-17中的 `SELECT` 语句会发生错误<sup>❷</sup>。

代码清单3-17 `GROUP BY` 子句中使用列的别名会引发错误

```
SELECT shohin_bunrui AS 别名, COUNT(*)
FROM Shohin
GROUP BY 别名;
```

↑ 在 `GROUP BY` 子句中使用在 `SELECT` 子句中定义的别名

上述语句发生错误的原因之前已经介绍过了，是由于SQL语句在DBMS内部的执行顺序造成的——`SELECT` 子句在 `GROUP BY` 子句之后执行。在执行 `GROUP BY` 子句时，`SELECT` 子句中定义的别名，DBMS

还并不知道。

使用本书提供的 PostgreSQL 执行上述 SQL 语句并不会发生错误，而会得到如下结果。但是这样的写法在其他 DBMS 中并不是通用的，因此请大家不要使用。

执行结果（使用 PostgreSQL 的情况）

sb	count
衣服	2
办公用品	2
厨房用具	4



#### 法则 3-10

在 GROUP BY 子句中不能使用 SELECT 子句中定义的别名。

#### ■ 常见错误③ —— GROUP BY 子句的结果能排序吗？

GROUP BY 子句的结果通常都包含多行，有时可能还会是成百上千行。那么，这些结果究竟是按照什么顺序排列的呢？

答案是：“随机的。”

我们完全不知道结果记录是按照什么规则进行排序的。可能乍一看是按照行数的降序或者聚合键的升序进行排列的，但其实这些全都是偶然的。当你再次执行同样的 SELECT 语句时，得到的结果的顺序可能会按照完全不同的顺序进行排列。

通常 SELECT 语句执行结果的显示顺序都是随机的。因此想要按照某种特定顺序进行排序的话，需要在 SELECT 语句中进行指定。具体的方法将在第 4 节中学习。

#### KEYWORD

● 排序



#### 法则 3-11

GROUP BY 子句结果的显示是无序的。

#### ■ 常见错误④ —— 在 WHERE 子句中使用聚合函数

最后要介绍的是初学者非常容易犯的一个错误。我们还是先来看一下之前提到的使用商品种类 (shohin\_bunrui 列) 对表进行分组，计算

每种商品数据行数的例子吧。SELECT 语句如代码清单 3-18 所示。

代码清单 3-18 按照商品种类统计数据行数

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui;
```

执行结果

shohin_bunrui	count
衣服	2
办公用品	2
厨房用具	4

如果我们想要取出恰好包含 2 行数据的组该怎么办呢？满足要求的是办公用品和衣服。

想要指定选择条件时就要用到 WHERE 子句，初学者通常会想到使用代码清单 3-19 中的 SELECT 语句吧。

代码清单 3-19 在 WHERE 子句中使用聚合函数会引发错误

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
WHERE COUNT(*) = 2
GROUP BY shohin_bunrui;
```

遗憾的是，这样的 SELECT 语句在执行时会发生错误。

执行结果 (使用 PostgreSQL 的情况)

```
ERROR: 不能在WHERE子句中使用聚合
行 3: WHERE COUNT(*) = 2
          ^
```

实际上，只有 SELECT 子句和 HAVING 子句（以及之后将要学到的 ORDER BY 子句）中能够使用 COUNT 等聚合函数。并且，HAVING 子句可以非常方便的实现上述要求。下一节我们将会学习 HAVING 子句。



#### 法则 3-12

只有 SELECT 子句和 HAVING 子句（以及 ORDER BY 子句）中能够使用聚合函数。

## 专 栏

**DISTINCT和GROUP BY**

细心的读者可能会发现，3-1 节中介绍的 DISTINCT 和 3-2 节介绍的 GROUP BY 子句，都能够删除后续列中的重复数据。例如，代码清单 3-A 中的 2 条 SELECT 语句会返回相同的结果。

**代码清单 3-A DISTINCT和GROUP BY能够实现相同的功能**

```
SELECT DISTINCT shohin_bunrui
FROM Shohin;

SELECT shohin_bunrui
FROM Shohin
GROUP BY shohin_bunrui;
```

**执行结果**

```
shohin_bunrui
-----
```

```
衣服
办公用品
厨房用具
```

除此之外，它们还都会把 NULL 作为一个独立的结果返回，对多列使用时也会得到完全相同的结果。其实不仅处理结果相同，执行速度也基本上差不多<sup>①</sup>，到底应该使用哪一个呢？

但其实这个问题本身就是本末倒置的，我们应该考虑的是该 SELECT 语句是否满足需求。选择的标准其实非常简单，在“想要删除选择结果中的重复记录”时使用 DISTINCT，在“想要计算聚合结果”时使用 GROUP BY。

不使用 COUNT 等聚合函数，而只使用 GROUP BY 子句的 SELECT 语句，会让人觉得非常奇怪。难免使人产生“到底为什么要对表进行分组呢？这样做有必要吗？”等疑问。

SQL 语句的语法与英语十分相似，理解起来非常容易，希望大家不要浪费这一优势，编写出一些难以理解的 SQL 语句。

①

它们都是数据的内部处理，都是通过排序处理来实现的。

## 3-3

该列的最大值

## 为聚合结果指定条件

## 学习重点

- 使用 COUNT 函数等对表中数据进行聚合操作时，为其指定条件的不是 WHERE 子句，而需要使用 HAVING 子句。
- 聚合函数可以在 SELECT 子句、HAVING 子句和 ORDER BY 子句中使用。
- HAVING 子句要写在 GROUP BY 子句之后。
- WHERE 子句用来指定数据行的条件，HAVING 子句用来指定分组的条件。

## HAVING 子句

使用前一节学过的 GROUP BY 子句，可以得到将表分组后的结果。在此，我们来思考一下通过指定条件来选取特定组的方法。例如，如何才能取出“聚合结果正好为 2 行的组”呢（图 3-8）。

图 3-8 取出符合指定条件的组



说到指定条件，估计大家都会首先想到 WHERE 子句。但是，WHERE 子句只能指定记录（行）的条件，而不能用来指定组的条件（例如，“数据行数为 2 行”或者“平均值为 500”等）。

## KEYWORD

## ● HAVING 子句

## ①

HAVING 就是 HAVE (拥有) 的现在分词, 并不是通常使用的英语单词。

因此, 对集合指定条件就需要使用其他的子句了。此时便可以用 HAVING 子句<sup>①</sup>。

HAVING 子句的语法如下所示。

## 语法 3-3 HAVING 子句

```
SELECT <列名 1>, <列名 2>, <列名 3>, .....
FROM <表名>
GROUP BY <列名 1>, <列名 2>, <列名 3>, .....
HAVING <分组结果对应的条件>
```

HAVING 子句必须写在 GROUP BY 子句之后。其在 DBMS 内部的执行顺序也排在 GROUP BY 子句之后。

▶ 使用 HAVING 子句时 SELECT 语句的顺序

SELECT → FROM → WHERE → GROUP BY → HAVING



## 法则 3-13

HAVING 子句要写在 GROUP BY 子句之后。

接下来就让我们练习一下 HAVING 子句吧。例如, 针对通过商品种类进行聚合分组后的结果, 指定“包含数据的行数为 2 行”这一条件的 SELECT 语句, 请参见代码清单 3-20。

代码清单 3-20 从通过商品种类进行聚合分组后的结果中, 取出“包含数据的行数为 2 行”的组

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui
HAVING COUNT(*) = 2;
```

执行结果

shohin_bunrui	count
衣服	2
办公用品	2

我们可以看到执行结果中并没有包含 4 行“厨房用具”数据。未使用 HAVING 子句时的执行结果中包含“厨房用具”, 但是通过设置 HAVING



子句的条件，我们就可以做到只选取出包含 2 行数据的组了（代码清单 3-21）。

代码清单 3-21 不使用 HAVING 子句的情况

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui;
```

执行结果

shohin_bunrui	count
衣服	2
办公用品	2
厨房用具	4

← 行数不是2的组也显示出来了

下面我们再看一个使用 HAVING 子句的例子。这次我们还是通过商品种类对表进行分组，但是条件变成了“销售单价的平均值大于等于 2500 日元”。

首先来看一下不使用 HAVING 子句的情况，请参见代码清单 3-22。

代码清单 3-22 不使用 HAVING 子句的情况

```
SELECT shohin_bunrui, AVG(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

执行结果

shohin_bunrui	avg
衣服	2500.0000000000000000
办公用品	300.0000000000000000
厨房用具	2795.0000000000000000

按照商品种类进行切分的 3 组数据都显示出来了。下面我们使用 HAVING 子句来设定条件，请参见代码清单 3-23。

代码清单 3-23 使用 HAVING 子句设定条件的情况

```
SELECT shohin_bunrui, AVG(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui
HAVING AVG(hanbai_tanka) >= 2500;
```

**执行结果**

shohin_bunrui	avg
衣服	2500.0000000000000000
厨房用具	2795.0000000000000000

销售单价的平均值为 300 日元的办公用品在结果中消失了。

## .....

### HAVING 子句的构成要素

HAVING 子句和包含 GROUP BY 子句时的 SELECT 子句一样，能够使用的要素有一定的限制。限制内容也是完全相同的，HAVING 子句中能够使用的 3 种要素如下所示。

- 常数
- 聚合函数
- GROUP BY 子句中指定的列名（即聚合键）

代码清单 3-20 中的例文指定了 HAVING COUNT(\*) = 2 这样的条件。其中 COUNT(\*) 是聚合函数，2 是常数，全都满足上述要求。反之，如果写成了下面这个样子就会发生错误（代码清单 3-24）。

#### 代码清单 3-24 HAVING 子句的不正确使用方法

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui
HAVING shohin_mei = '圆珠笔';
```

**执行结果**

```
ERROR: 列"shohin,shohin_mei" 必须包含在GROUP BY子句中, 或者必须在聚合函数中使用
行 4: HAVING shohin_mei = '圆珠笔';
```

⇒表示下一行接续本行，只是由于版面所限而换行。

shohin\_mei 列并不包含在 GROUP BY 子句之中，因此不允许写在 HAVING 子句里。在思考 HAVING 子句的使用方法时，把一次聚合后的结果（类似表 3-2 的表）作为 HAVING 子句起始点的话更容易理解。

表 3-2 通过商品种类聚合分组后的结果

shohin_bunrui	COUNT(*)
厨房用具	4
衣服	2
办公用品	2

可以把这种情况想象为使用 GROUP BY 子句时的 SELECT 子句。聚合之后得到的表中并不存在 shohin\_mei 这个列，SQL 当然也无法为表中不存在的列设定条件啦。

## 相对于 HAVING 子句， 更适合写在 WHERE 子句中的条件

也许有的读者已经发现了，有些条件既可以写在 HAVING 子句中，又可以写在 WHERE 子句中。这些条件就是聚合键所对应的条件。原表中作为聚合键的列也可以在 HAVING 子句中使用。因此，代码清单 3-25 中的 SELECT 语句也是正确的。

代码清单 3-25 将条件书写在 HAVING 子句的情况

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui
HAVING shohin_bunrui = '衣服';
```

执行结果

shohin_bunrui	count
衣服	2

上述 SELECT 语句的返回结果与代码清单 3-26 中 SELECT 语句的返回结果是相同的。

代码清单 3-26 将条件书写在 WHERE 子句中的情况

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
WHERE shohin_bunrui = '衣服'
GROUP BY shohin_bunrui;
```

## 执行结果

```
shohin_bunrui|count
-----+-----
衣服          |      2
```

虽然条件分别写在 WHERE 子句和 HAVING 子句当中，但是条件的内容，以及返回的结果都完全相同。因此，大家可能会觉得两种书写方式都没问题。

如果仅从结果来看的话，确实如此。但笔者却认为，聚合键所对应的条件还是应该书写在 WHERE 子句之中。

理由有两个。

首先，根本原因是 WHERE 子句和 HAVING 子句的作用不同。如前所述，HAVING 子句是用来指定“组”的条件的。因此，“行”所对应的条件还是应该写在 WHERE 子句当中。这样一来，书写出的 SELECT 语句不但可以分清两者各自的功能，理解起来也更加容易。

WHERE 子句 = 指定行所对应的条件

HAVING 子句 = 指定组所对应的条件

其次，对初学者来说，研究 DBMS 的内部实现这一话题有些深奥，这里就不做介绍了。感兴趣的读者可以参考随后的专栏——WHERE 子句和 HAVING 子句的执行速度。



## 法则 3-14

聚合键所对应的条件不应该书写在 HAVING 子句当中，而应该书写在 WHERE 子句当中。

## 专 栏

### WHERE 子句和 HAVING 子句的执行速度

在 WHERE 子句和 HAVING 子句中都可以使用的条件，最好写在 WHERE 子句中的另一个理由与性能即执行速度有关系。由于性能不在本书介绍的范围之内，所以暂不进行说明。通常情况下，为了得到相同的结果，将条件写在 WHERE 子句中要比写在 HAVING 子句中的处理速度更快，返回结果所需时间更短。

为了理解其中原因，就要从 DBMS 的内部运行机制来考虑。使用 COUNT 函数等对表中的数据进行聚合操作时，DBMS 内部就会进行排序处理。排序处理会大大增加机器的负担，此即所谓高负荷的处理。因此，只有尽可能减少排序的行数，才能增加处理速度。

通过 WHERE 子句指定条件时，由于排序之前就对数据进行了过滤，所以能够减少排序的数据量。但 HAVING 子句是在排序之后才对数据进行分组的，因此与在 WHERE 子句中指定条件比起来，需要排序的数据量就会多得多。虽然 DBMS 的内部处理不尽相同，但是对于排序处理来说，基本上都是一样的。

此外，WHERE 子句更具速度优势的另一个理由是，可以对 WHERE 子句指定条件所对应的列创建索引，这样也可以大幅提高处理速度。创建索引是一种非常普遍的提高 DBMS 性能的方法，效果也十分明显，这对 WHERE 子句来说也十分有利。

#### KEYWORD

●索引 (index)

## 3-4

该列的最大值

## 对查询结果进行排序

## 学习重点

- 使用ORDER BY子句对查询结果进行排序。
- 在ORDER BY子句中列名的后面使用关键字ASC可以进行升序排序，使用DESC关键字可以进行降序排序。
- ORDER BY子句中指定多个排序键。
- 排序键中包含NULL时，会在开头或末尾进行汇总。
- ORDER BY子句中可以使用SELECT子句中定义的列的别名。
- ORDER BY子句中可以使用SELECT子句中未出现的列或者聚合函数。
- ORDER BY子句中不能使用列的编号。

## ORDER BY子句

截至目前，我们使用了各种各样的条件对表中的数据进行查询。本节让我们再来回顾一下简单的SELECT语句（代码清单3-27）。

代码清单3-27 显示商品编号、商品名称、销售单价和进货单价的SELECT语句

```
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin;
```

执行结果

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0001	T恤衫	1000	500
0002	打孔器	500	320
0003	运动T恤	4000	2800
0004	菜刀	3000	2800
0005	高压锅	6800	5000
0006	叉子	500	
0007	擦菜板	880	790
0008	圆珠笔	100	

对于上述结果，在此无需特别说明。本节要为大家介绍的不是查询结果，而是查询结果的排列顺序。

那么，结果中的8行记录到底是按照什么顺序排列的呢？乍一看，貌似是按照商品编号从小到大的顺序（升序）排列的。其实，排列顺序是随

## KEYWORD

● 升序

机的,这只是个偶然。因此,再次执行同一条 SELECT 语句时,顺序可能大为不同。

通常,从表中抽取数据时,如果没有特别指定顺序,最终排列顺序便无从得知。即使是同一条 SELECT 语句,每次执行时排列顺序很可能发生改变。

但是不进行排序,很可能出现结果混乱的情况。这时,便需要通过在 SELECT 语句末尾添加 ORDER BY 子句来明确指定排列顺序。

ORDER BY 子句的语法如下所示。

#### KEYWORD

● ORDER BY 子句

#### 语法 3-4 ORDER BY 子句

```
SELECT <列名 1>, <列名 2>, <列名 3>, .....
FROM <表名>;
ORDER BY <排序基准列 1>, <排序基准列 2>, .....
```

例如,按照销售单价由低到高,也就是升序排列时,请参见代码清单 3-28。

#### 代码清单 3-28 按照销售单价由低到高(升序)进行排列

```
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY hanbai_tanka;
```

#### 执行结果

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0008	圆珠笔	100	
0006	叉子	500	
0002	打孔器	500	320
0007	擦菜板	880	790
0001	T恤衫	1000	500
0004	菜刀	3000	2800
0003	运动T恤	4000	2800
0005	高压锅	6800	5000

销售单价的升序

不论何种情况,ORDER BY 子句都需要写在 SELECT 语句的末尾。这是因为对数据进行排序的操作必须在结果即将返回时执行。ORDER BY 子句中书写的列名称为排序键。该子句与其他子句执行的顺序关系如下所示。

#### KEYWORD

● 排序键

#### ► 使用 HAVING 子句时 SELECT 语句的顺序

1. SELECT 子句 → 2. FROM 子句 → 3. WHERE 子句 → 4. GROUP BY 子句 →
5. HAVING 子句 → 6. ORDER BY 子句



## 法则 3-15

ORDER BY 子句通常写在 SELECT 语句的末尾。

不想指定数据行的排列顺序时，SELECT 语句中不写 ORDER BY 子句也没关系。

## KEYWORD

- 降序
- DESC 关键字

## 指定升序或降序

与上述示例相对，想要按照销售单价由高到低，也就是降序排列时，可以参见代码清单 3-29，在列名后面使用 DESC 关键字。

代码清单 3-29 按照销售单价由高到低（降序）进行排列

```
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY hanbai_tanka DESC;
```

执行结果

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0005	高压锅	6800	5000
0003	运动T恤	4000	2800
0004	菜刀	3000	2800
0001	T恤衫	1000	500
0007	擦菜板	880	790
0002	打孔器	500	320
0006	叉子	500	
0008	圆珠笔	100	

如上所示，这次销售单价最高（6800 日元）的高压锅排在了第一位。其实，使用升序进行排列时，正式的书写方式应该是使用关键字 ASC，但是省略该关键字时会默认使用升序进行排序。这可能是因为实际应用中按照升序排序的情况更多吧。ASC 和 DESC 是 ascendent（上升的）和 descendent（下降的）这两个单词的缩写。

## KEYWORD

- ASC 关键字



## 法则 3-16

未指定 ORDER BY 子句中排列顺序时会默认使用升序进行排列。



由于 ASC 和 DESC 这两个关键字是以列为单位指定的，所以可以同时指定一个列为升序，指定其他列为降序。

## 指定多个排序键

本节开头曾提到过对销售单价进行升序排列的 SELECT 语句（代码清单 3-27）的执行结果，我们再来回顾一下。可以发现销售单价为 500 日元的商品有 2 件。相同价格的商品的顺序并没有特地指定，或者可以说是随机排列的。

如果想要对该顺序的商品进行更细致的排序的话，就需要再添加一个排序键。在此，我们以添加商品编号的升序为例，请参见代码清单 3-30。

代码清单 3-30 按照销售单价和商品编号的升序进行排序

```
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY hanbai_tanka, shohin_id;
```

执行结果

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0008	圆珠笔	100	
0002	打孔器	500	320
0006	叉子	500	
0007	擦菜板	880	790
0001	T恤衫	1000	500
0004	菜刀	3000	2800
0003	运动T恤	4000	2800
0005	高压锅	6800	5000

价格相同时按照商品编号的升序排列

这样一来，就可以在 ORDER BY 子句中同时指定多个排序键了。会优先使用左侧的键，如果该列存在相同值的话，会接着参考右侧的键。当然，也可以同时使用 3 个以上的排序键。

## NULL 的顺序

在此前的示例中，我们已经使用过销售单价（hanbai\_tanka 列）作为排序键了，这次让我们尝试使用销售单价（shiire\_tanka 列）作

为排序键吧。此时，问题就来了，圆珠笔和叉子对应的值是 NULL。究竟 NULL 会按照什么顺序进行排列呢。NULL 是大于 100 还是小于 100 呢？或者说 5000 和 NULL 哪个更大呢？

请大家回忆一下我们在第 2 章中学过的内容。没错，不能对 NULL 使用比较运算符，也就是说，不能对 NULL 和数字进行排序。也不能与字符串和日期比较大小。因此，使用含有 NULL 的列作为排序键时，NULL 会在结果的开头或末尾汇总显示（代码清单 3-31）。

代码清单 3-31 按照进货单价的升序进行排列

```
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY shiire_tanka;
```

执行结果

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0002	打孔器	500	320
0001	T恤衫	1000	500
0007	擦菜板	880	790
0003	运动T恤	4000	2800
0004	菜刀	3000	2800
0005	高压锅	6800	5000
0006	叉子	500	
0008	圆珠笔	100	

NULL 会汇总在开头或者末尾

究竟是在开头显示还是在末尾显示，并没有特殊规定。某些 DBMS 中可以指定 NULL 在开头或末尾显示，希望大家对自己使用的 DBMS 的功能研究一下。



#### 法则 3-17

排序键中包含 NULL 时，会在开头或末尾进行汇总。

## 在排序键中使用显示用别名

在 3-2 节“常见错误②”中曾介绍过，在 GROUP BY 子句中不能使用 SELECT 子句中定义的别名。但是在 ORDER BY 子句中却是允许使用别名的。因此，代码清单 3-32 中的 SELECT 语句并不会出错，可正确执行。

代码清单 3-32 ORDER BY 子句中可以使用列的别名

```
SELECT shohin_id AS id, shohin_mei, hanbai_tanka AS ht, shiire_
_tanka
FROM Shohin
ORDER BY ht, id;
```

⇒表示下一行接续本行，只是由于版面所限而换行。

上述 SELECT 语句与之前按照“销售单价和商品编号的升序进行排列”的 SELECT 语句（代码清单 3-31）意思完全相同。

#### 执行结果

id	shohin_mei	ht	shiire_tanka
0008	圆珠笔	100	
0002	打孔器	500	320
0006	叉子	500	
0007	擦黑板	880	790
0001	T恤衫	1000	500
0004	菜刀	3000	2800
0003	运动T恤	4000	2800
0005	高压锅	6800	5000

不能在 GROUP BY 子句中使用的别名，为什么可以在 ORDER BY 子句中使用呢？这是因为 SQL 语句在 DBMS 内部的执行顺序被掩盖起来了。SELECT 语句按照子句为单位的执行顺序如下所示。

#### ►使用 HAVING 子句时 SELECT 语句的顺序

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY

这只是一个粗略的总结，虽然具体的执行顺序根据 DBMS 的不同而不同，但是大家有这样一个大致的印象就可以了。一定要记住 SELECT 子句的执行顺序在 GROUP BY 子句之后，ORDER BY 子句之前。因此，在执行 GROUP BY 子句时，SELECT 语句中定义的别名无法被识别<sup>①</sup>。对于在 SELECT 子句之后执行的 ORDER BY 子句来说，就没有这样的问题了。

#### ①

也是因为这一原因，HAVING 子句也不能使用别名。



#### 法则 3-18

在 ORDER BY 子句中可以使用 SELECT 子句中定义的别名。

## ORDER BY子句中可以使用的列

ORDER BY子句中可以使用存在于表中，但并不包含在SELECT子句之中的列（代码清单 3-33）。

代码清单 3-33 SELECT子句中未包含的列也可以在ORDER BY子句中使用

```
SELECT shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY shohin_id;
```

执行结果

shohin_mei	hanbai_tanka	shiire_tanka
T恤衫	1000	500
打孔器	500	320
运动T恤	4000	2800
菜刀	3000	2800
高压锅	6800	5000
叉子	500	
擦菜板	880	790
圆珠笔	100	

除此之外，还可以使用聚合函数（代码清单 3-34）。

代码清单 3-34 ORDER BY子句中也可以使用聚合函数

```
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui
ORDER BY COUNT(*);
```

也可以使用聚合函数

执行结果

shohin_bunrui	count
衣服	2
办公用品	2
厨房用具	4



### 法则 3-19

在ORDER BY子句中可以使用SELECT子句中未使用的列和聚合函数。

## 不要使用列编号

在ORDER BY子句中，还可以使用在SELECT子句中出现的列所对

**KEYWORD**

●列编号

应的编号，是不是没想到？列编号是指 SELECT 子句中的列按照从左到右的顺序进行排列时所对应的编号（1, 2, 3, …）。因此，代码清单 3-35 中的两条 SELECT 语句的含义是相同的。

代码清单 3-35 ORDER BY 子句中可以使用列的编号

```
-- 使用列明
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY hanbai_tanka DESC, shohin_id;

-- 使用列编号
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
ORDER BY 3 DESC, 1;
```

上述第 2 条 SELECT 语句中的 ORDER BY 子句所代表的含义，就是“按照 SELECT 子句中第 3 列的降序和第 1 列的升序进行排列”。这和第 1 条 SELECT 语句的含义完全相同。

**执行结果**

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka
0005	高压锅	6800	5000
0003	运动T恤	4000	2800
0004	菜刀	3000	2800
0001	T恤衫	1000	500
0007	擦黑板	880	790
0002	打孔器	500	320
0006	叉子	500	
0008	圆珠笔	100	

虽然列编号使用起来非常方便，但我们并不推荐使用，原因有以下两点。

第一，代码阅读起来比较难。使用列编号时，如果只看 ORDER BY 子句是无法知道当前是按照那一列进行排序的，只能去 SELECT 子句的列表中按照列编号进行确认。上述示例中 SELECT 子句的列数比较少，因此可能并没有什么明显的感觉。但是在实际应用中往往会出现列数很多的情况，而且 SELECT 子句和 ORDER BY 子句之间，还可能包含很复杂的 WHERE 子句和 HAVING 子句，直接人工确认实在太麻烦了。

第二，这也是最根本的问题。实际上，在 SQL-92<sup>①</sup> 中已经明确指出该排序功能将来会被删除。因此，虽然现在使用起来没有问题，但是将来

①

1992年制定的SQL标准。

随着 DBMS 的版本升级，可能原本能够正常执行的 SQL 突然就会出错。不光是这种单独使用的 SQL 语句，对于那些在系统中混合使用的 SQL 来说，更要极力避免。



### 法则 3-20

在 ORDER BY 不要使用列编号。

### 练习题

3.1 请指出下述 SELECT 语句中所有的语法错误。

```
SELECT shohin_id, SUM(shohin_mei)
-- 本SELECT语句中存在错误。
FROM Shohin
GROUP BY shohin_bunrui
WHERE torokubi > '2009-09-01';
```

3.2 请编写一条 SELECT 语句，求出销售单价 (hanbai\_tanka 列) 合计值是进货单价 (shiire\_tanka 列) 合计值 1.5 倍的商品种类。执行结果如下所示。

shohin_bunrui	sum	sum
衣服	5000	3300
办公用品	600	320

← SUM(shiire\_tanka) 的结果

↑  
SUM(hanbai\_tanka) 的结果

3.3 此前我们曾经使用 SELECT 语句选取出了 Shohin (商品) 表中的全部记录。当时我们使用了 ORDER BY 子句来指定排列顺序，但现在已经无法记起当时如何指定的了。请根据下列执行结果，思考 ORDER BY 子句的内容。

#### 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0003	运动T恤	衣服	4000	2800	
0008	圆珠笔	办公用品	100		2009-11-11
0006	叉子	厨房用具	500		2009-09-20
0001	T恤衫	衣服	1000	500	2009-09-20
0004	菜刀	厨房用具	3000	2800	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0005	高压锅	厨房用具	6800	5000	2809-01-15
0007	煎蛋器	厨房用具	880	790	2008-04-28

## 第4章 数据更新

数据的插入 ( INSERT 语句的使用方法 )

数据的删除 ( DELETE 语句的使用方法 )

数据的更新 ( UPDATE 语句的使用方法 )

事务

# 4



# SQL

## 本章重点

---

此前几章和大家一起学习了查询表中数据的几种方法，所使用的 SQL 语句都是 SELECT 语句。SELECT 语句并不会更改表中数据，也就是说，SELECT 语句是读取专用的指令。

本章将会给大家介绍 DBMS 中用来更新表中数据的方法。数据的更新处理大体可以分为插入 (INSERT)、删除 (DELETE) 和更新 (UPDATE) 三类。本章将会对这三类更新方法进行详细介绍。此外，还会给大家介绍数据库中用来管理数据更新的重要概念——事务。

### 4-1 数据的插入 (INSERT 语句的使用方法)

- 什么是 INSERT
- INSERT 语句的基本语法
- 列清单的省略
- 插入 NULL
- 插入默认值
- 从其他表中复制数据

### 4-2 数据的删除 (DELETE 语句的使用方法)

- DROP TABLE 语句和 DELETE 语句
- DELETE 语句的基本语法
- 指定删除对象的 DELETE 语句 (搜索型 DELETE)

### 4-3 数据的更新 (UPDATE 语句的使用方法)

- UPDATE 语句的基本语法
- 指定条件的 UPDATE 语句 (搜索型 UPDATE)
- 使用 NULL 进行更新
- 多列的更新

### 4-4 事务

- 什么是事务
- 创建事务
- ACID 特性



## 4-1

## 第4章 数据更新

## 数据的插入 (INSERT 语句的使用方法)

## 学习重点

- 使用 INSERT 语句可以向表中插入数据 (行)。原则上, INSERT 语句每次执行一行数据的插入。
- 列名和值用逗号隔开, 分别括在 ( ) 内, 这种形式称为清单。
- 对表中所有列进行 INSERT 操作时可以省略表名后的列清单。
- 插入 NULL 时需要在 VALUES 子句的值清单中写入 NULL。
- 可以为表中的列设定默认值 (初始值)。默认值可以通过在 CREATE TABLE 语句中, 为列设置 DEFAULT 约束来设定。
- 插入默认值可以通过两种方式实现。即在 INSERT 语句的 VALUES 子句中指定 DEFAULT 关键字 (显示方法), 或省略列清单 (隐式方法)。
- 使用 INSERT...SELECT 可以从其他表中复制数据。

## 什么是 INSERT

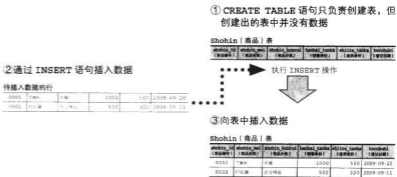
1-4 节给大家介绍了用来创建表的 CREATE TABLE 语句。通过 CREATE TABLE 语句创建出来的表, 可以将其比作一个空空如也的箱子。只有把数据装入到这个箱子后, 它才能称为数据库。用来装入数据的 SQL 就是 INSERT (插入) (图 4-1)。

本节将会和大家一起学习 INSERT 语句。

图 4-1 INSERT (插入) 的流程

## KEYWORD

- INSERT 语句



要学习 INSERT 语句，我们得首先创建一个名为 ShohinIns 的表。请大家执行代码清单 4-1 中的 CREATE TABLE 语句。该表的内容，除了为 hanbai\_tanka 列（销售单价）设置了 DEFAULT 0 的约束之外，与之前使用的 Shohin（商品）表完全相同。DEFAULT 0 的含义将会在随后进行介绍，大家暂时可以忽略。

代码清单 4-1 创建 ShohinIns 表的 CREATE TABLE 语句

```
CREATE TABLE ShohinIns
(shohin_id CHAR(4) NOT NULL,
 shohin_mei VARCHAR(100) NOT NULL,
 shohin_bunrui VARCHAR(32) NOT NULL,
 hanbai_tanka INTEGER DEFAULT 0,
 shiire_tanka INTEGER ,
 torokubi DATE ,
 PRIMARY KEY (shohin_id));
```

如前所述，这里仅仅是创建出了一个表，并没有插入数据。接下来，我们就向 ShohinIns 表中插入数据。

## INSERT 语句的基本语法

在 1-5 节，向 CREATE TABLE 语句创建出的 Shohin 表中，插入数据的 SQL 语句时，曾介绍过 INSERT 语句的使用示例。但那时只是为了准备学习 SELECT 语句所需的数据，并没有详细介绍其语法。下面就让我们来介绍一下 INSERT 语句的语法结构。

INSERT 语句的基本语法如下所示。

语法 4-1 INSERT 语句

```
INSERT INTO <表名> (列1, 列2, 列3, ……) VALUES (值1, 值2, 值3, ……);
```

例如，我们要向 ShohinIns 表中插入一行数据，各列的值如下所示。

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20

所需的 INSERT 语句可参见代码清单 4-2。

代码清单 4-2 向表中插入一行数据

```
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui, ⇨
hanbai_tanka, shiire_tanka, torokubi) VALUES ('0001', 'T恤衫', ⇨
'衣服', 1000, 500, '2009-09-20');
```

⇨表示下一行接续本行，只是由于版面所限而换行。

由于 shohin\_id 列 (商品编号) 和 shohin\_mei 列 (商品名称) 是字符型，所以插入的数据需要像 '0001' 这样用单引号括起来。日期型的 torokubi (登记日期) 列也是如此<sup>❶</sup>。

列名和值用逗号隔开，分别括在 () 内，这种形式称为清单。代码清单 4-2 中的 INSERT 语句包含如下两个清单。

- ❶ 列清单 → (shohin\_id, shohin\_mei, shohin\_bunrui, hanbai\_tanka, shiire\_tanka, torokubi)
- ❷ 值清单 → ('0001', 'T恤衫', '衣服', 1000, 500, '2009-09-20')

当然，表名后面的列清单和 VALUES 子句中值中列的数量必须保持一致。如下所示，列数不一致时会出错，无法插入数据<sup>❸</sup>。

```
-- VALUES子句中的值清单缺少一列
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui, ⇨
hanbai_tanka, shiire_tanka, torokubi) VALUES ('0001', 'T恤衫', ⇨
'衣服', 1000, 500);
```

⇨表示下一行接续本行，只是由于版面所限而换行。

此外，原则上，执行一次 INSERT 语句会插入一行数据<sup>❹</sup>。因此，插入多行时，通常需要循环执行所需行次数的 INSERT 语句。

### ❶

有关日期型的介绍，请参考第 1 章中的“指定数据类型”。

### KEYWORD

- 清单
- 列清单
- 值清单

### ❷

但是使用默认值时列数无需完全一致。相关内容将会在随后的“插入默认值”中进行介绍。

### ❸

插入多行的情况，请参考专栏“多行 INSERT”。



### 法则 4-1

原则上，执行一次 INSERT 语句会插入一行数据。

## 专栏

## 多行 INSERT

法则 4-1 中介绍了“执行一次 INSERT 语句会插入一行数据”的原则。虽然在大多数情况下，该原则都是正确的，但它也仅仅是原则而已。其实很多 RMDBS 都支持一次 INSERT 多行数据。这样的功能称为“多行 INSERT (multi row INSERT)”。

其语法请参见代码清单 4-A，将多条 VALUES 子句通过逗号进行分隔排列。

## 代码清单 4-A 通常的 INSERT 和多行 INSERT

```
-- 通常的 INSERT
INSERT INTO ShohinIns VALUES ('0002', '打孔器', →
'办公用品', 500, 320, '2009-09-11');
INSERT INTO ShohinIns VALUES ('0003', '运动T恤', →
'衣服', 4000, 2800, NULL);
INSERT INTO ShohinIns VALUES ('0004', '菜刀', →
'厨房用具', 3000, 2800, '2009-09-20');

-- 多行 INSERT (Oracle以外)
INSERT INTO ShohinIns VALUES ('0002', '打孔器', →
'办公用品', 500, 320, '2009-09-11'),
('0003', '运动T恤', →
'衣服', 4000, 2800, NULL),
('0004', '菜刀', →
'厨房用具', 3000, 2800, '2009-09-20');
```

⇒表示下一行接续本行，只是由于版面所限而换行。

该语法很容易理解，并且减少了书写语句的数量，非常方便。但是，使用该语法时请注意以下几点。

首先，INSERT 语句书写内容及插入的数据是否正确。当然，此时都会发生 INSERT 错误，但是由于是多行插入，和特定的单行 INSERT 相比，想要找出到底是哪行哪个地方出错了，就变得十分困难。

其次，该多行 INSERT 的语法并不适用于所有的 RDBMS。该多语法适用于 DB2、SQL、SQL Server、PostgreSQL 和 MySQL，但不适用于 Oracle。

## 特定的 SQL

Oracle 巧妙地使用如下语法来完成多行 INSERT 操作。

```
-- Oracle中的多行 INSERT
INSERT ALL INTO ShohinIns VALUES ('0002', '打孔器', →
'办公用品', 500, 320, '2009-09-11')
INTO ShohinIns VALUES ('0003', '运动T恤', →
'衣服', 4000, 2800, NULL)
INTO ShohinIns VALUES ('0004', '菜刀', →
'厨房用具', 3000, 2800, '2009-09-20')
SELECT * FROM DUAL;
```

⇒表示下一行接续本行，只是由于版面所限而换行。

DUAL 是 Oracle 特有（安装时的必选项）的一种临时表<sup>①</sup>。因此“SELECT \* FROM DUAL”部分也只是临时性的，并没有实际意义。

## KEYWORD

●多行 INSERT

①

在书写没有参照表的 SELECT 语句时，写在 FROM 子句中的表。它并没有实际意义，也不保存任何数据，同时也不能作为 INSERT 和 UPDATE 的对象。

## 列清单的省略

对表进行全列 INSERT 时，可以省略表名后的列清单。这时 VALUES 子句的值会默认按照从左到右的顺序赋给每一列。因此，代码清单 4-3 中的两个 INSERT 语句会插入同样的数据。

代码清单 4-3 省略列清单

```
-- 包含列清单
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui,
hanbai_tanka, shiire_tanka, torokubi) VALUES ('0005', '高压锅',
'厨房用具', 6800, 5000, '2009-01-15');

-- 省略列清单
INSERT INTO ShohinIns VALUES ('0005', '高压锅', '厨房用具',
6800, 5000, '2009-01-15');
```

⇒表示下一行接续本行，只是由于版面所限而换行。

## 插入 NULL

INSERT 语句中想给某一列赋予 NULL 值时，可以直接在 VALUES 子句的值清单中写入 NULL。例如，要向 shiire\_tanka 列（进货单价）中插入 NULL，就可以使用代码清单 4-4 中的 INSERT 语句。

代码清单 4-4 向 shiire\_tanka 列中插入 NULL

```
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui,
hanbai_tanka, shiire_tanka, torokubi) VALUES ('0006', '叉子',
'厨房用具', 500, NULL, '2009-09-20');
```

⇒表示下一行接续本行，只是由于版面所限而换行。

但是，想要插入 NULL 的列一定不能设置 NOT NULL 约束。向设置了 NOT NULL 约束的列中插入 NULL 时，INSERT 语句会出错，数据插入失败。

插入失败指的是希望通过 INSERT 语句插入的数据无法正常插入到表中，但之前已经插入的数据并不会被破坏<sup>❶</sup>。

### ❶

不仅 INSERT、DELETE 和 UPDATE 等更新语句也一样，SQL 语句执行失败时都不会对表中数据造成影响。

**KEYWORD**

- 默认值
- DEFAULT 约束

## 插入默认值

我们还可以向表中插入默认值（初始值）。默认值的设定，可以通过在创建表的 CREATE TABLE 语句中设置 DEFAULT 约束来实现。

本章开头创建的 ShohinIns 表的定义片段请参见代码清单 4-5。其中 DEFAULT 0 就是设置 DEFAULT 约束的部分。我们可以像这样通过“DEFAULT <默认值>”的形式来设定默认值。

代码清单 4-5 创建 ShohinIns 表的 CREATE TABLE 语句（节选）

```
CREATE TABLE ShohinIns
(shohin_id CHAR(4) NOT NULL,
 (略)
hanbai_tanka INTEGER DEFAULT 0, -- 销售单价的默认值设定为0;
 (略)
PRIMARY KEY (shohin_id));
```

如果在创建表的同时设定了默认值，就可以在 INSERT 语句中自动为列赋值了。默认值的使用方法通常有显示和隐示两种。

### ■ 通过显示方法插入默认值

在 VALUES 子句中指定 DEFAULT 关键字（代码清单 4-6）。

代码清单 4-6 通过显示方法设定默认值

```
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui, →
hanbai_tanka, shiire_tanka, torokubi) VALUES ('0007', →
'煎菜板', '厨房用具', DEFAULT, 790, '2009-04-28');
```

⇒表示下一行接续本行，只是由于版面所限而换行。

这样一来，RDBMS 就会在插入记录时，自动把默认值赋给对应的列。我们可以使用 SELECT 语句来确认通过 INSERT 语句插入的数据行。

```
-- 确认插入的数据行：
SELECT * FROM ShohinIns WHERE shohin_id = '0007';
```

因为 hanbai\_tanka 列（销售单价）的默认值是 0，所以 hanbai\_tanka 列被赋予了值 0。

### 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0007	煎菜板	厨房用具	0	790	2009-04-28

**KEYWORD**

- DEFAULT 关键字

### ■通过隐示方法插入默认值

插入默认值时也可以不使用 DEFAULT 关键字。只要在列清单和 VALUES 中省略设定了默认值的列就可以了。我们可以像代码清单 4-7 那样，从 INSERT 语句中删除 hanbai\_tanka 列（销售单价）。

代码清单 4-7 通过隐示方法设定默认值

```
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui,
shiiire_tanka, torokubi) VALUES ('0007', '擦菜板', '厨房用具',
790, '2009-04-28');
```

省略hanbai\_tanka列  
值也省略

⇒表示下一行接续本行，只是由于版面所限而快行。

这样也可以给 hanbai\_tanka 赋上默认值 0。

那么，在实际使用中哪种方法更好呢？笔者建议大家使用 @ 的明示书写方法。因为这样可以一目了然地知道 hanbai\_tanka 列使用了默认值，SQL 语句的含义也更加容易理解。

说到省略列名，还有一点要说明一下。如果省略了没有设定默认值的列的话，该列的值就会被设定为 NULL。因此，如果省略的是设置了 NOT NULL 约束的列的话，INSERT 语句就会出错（代码清单 4-8）。请大家一定要注意。

代码清单 4-8 未设定默认值的情况

```
-- 省略shiiire_tanka列(无约束):会赋予[NULL]
INSERT INTO ShohinIns (shohin_id, shohin_mei, shohin_bunrui,
hanbai_tanka, torokubi) VALUES ('0008', '圆珠笔', '办公用品',
100, '2009-11-11');
```

```
-- 省略shohin_mei列(设置了NOT NULL约束):错误!
INSERT INTO ShohinIns (shohin_id, shohin_bunrui, hanbai_tanka,
shiiire_tanka, torokubi) VALUES ('0009', '办公用品', 1000, 500,
'2009-12-12');
```

⇒表示下一行接续本行，只是由于版面所限而快行。



#### 法则 4-2

省略 INSERT 语句中的列名，就会自动设定为该列的默认值（没有默认值时会设定为 NULL）。

## 从其他表中复制数据

插入数据的方法，除了使用 VALUES 子句指定具体的数据之外，还可以从其他表中复制数据。下面我们就来学习如何从一张表中选取数据，复制到另外一张表中。

要学习该方法，我们首先得创建一张表（代码清单 4-9）。

代码清单 4-9 创建 ShohinCopy 表的 CREATE TABLE 语句

```
-- 用来插入数据的商品复制表
CREATE TABLE ShohinCopy
(shohin_id    CHAR(4)      NOT NULL,
shohin_mei   VARCHAR(100) NOT NULL,
shohin_bunrui VARCHAR(32) NOT NULL,
hanbai_tanka INTEGER      ,
shiire_tanka INTEGER      ,
torokubi     DATE         ,
PRIMARY KEY (shohin_id));
```

ShohinCopy（商品复制）表的结构与之前使用的 Shohin（商品）表完全一样，只是更改了一下表名而已。

接下来，就让我们赶快尝试一下将 Shohin 表中的数据插入到 ShohinCopy 表中吧。代码清单 4-10 中的语句可以将 SELECT 的结果直接 INSERT 到表中。

代码清单 4-10 INSERT ... SELECT 语句

```
-- 将商品表中的数据复制到商品复制表中（复制）
INSERT INTO ShohinCopy (shohin_id, shohin_mei, shohin_bunrui, ⇨
hanbai_tanka, shiire_tanka, torokubi)
SELECT shohin_id, shohin_mei, shohin_bunrui, hanbai_tanka, ⇨
shiire_tanka, torokubi
FROM Shohin;
```

⇨表示下一行接续本行，只是由于版面所限而换行。

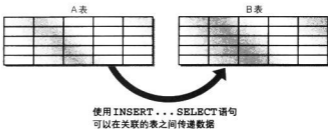
执行该 INSERT ... SELECT 语句时，如果原来 Shohin 表中有 8 行数据，那么 ShohinCopy 表中也会插入完全相同的 8 条数据。当然，Shohin 表中的原有数据不会发生改变。因此，INSERT ... SELECT 语句可以在需要进行数据备份时使用（图 4-2）。

### KEYWORD

- INSERT ... SELECT 语句



图4-2 INSERT ... SELECT 语句



### ■ 多种多样的 SELECT 语句

该 INSERT 语句中的 SELECT 语句，也可以使用 WHERE 子句或者 GROUP BY 子句等。目前为止学到的各种 SELECT 语句也都可以使用<sup>①</sup>。对关联表之间存取数据来说，这是非常方便的功能。

接下来我们尝试一下使用包含 GROUP BY 子句的 SELECT 语句进行 INSERT。代码清单 4-11 中的语句创建了一个用来插入数据的表。

代码清单 4-11 创建 ShohinBunrui 表的 CREATE TABLE 语句

```
-- 用来汇总商品种类的表；
CREATE TABLE ShohinBunrui
(shohin_bunrui VARCHAR(32) NOT NULL,
sum_hanbai_tanka INTEGER ;
sum_shiire_tanka INTEGER ;
PRIMARY KEY (shohin_bunrui));
```

该表是用来存储根据商品种类 (shohin\_bunrui) 计算出的销售单价合计数以及进货单价合计值的表。下面就让我们使用代码清单 4-12 中的 INSERT ... SELECT 语句，从 Shohin 表中选取数据插入到这张表中吧。

代码清单 4-12 插入其他表中数据合计值的 INSERT ... SELECT 语句

```
INSERT INTO ShohinBunrui (shohin_bunrui, sum_hanbai_tanka,
sum_shiire_tanka)
SELECT shohin_bunrui, SUM(hanbai_tanka), SUM(shiire_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

⇨表示下一行接续本行，只是由于版面所限而换行。

#### ①

但即便指定了 ORDER BY 子句也没有任何意义，这是因为无法保证表内部记录的排列顺序。

通过 SELECT 语句对插入结果进行确认，我们发现 ShohinBunrui 表中插入了以下 3 行数据。

```
-- 确认插入的数据行
SELECT * FROM ShohinBunrui;
```

执行结果

shohin_bunrui	sum_hanbai_tanka	sum_shiire_tanka
衣服	5000	3300
办公用品	600	320
厨房用具	11180	8590



#### 法则 4-3

INSERT 语句的 SELECT 语句中，可以使用 WHERE 子句或者 GROUP BY 子句等任何 SQL 语法。（但使用 ORDER BY 子句并不会产生任何效果。）

## 4-2

## 数据的删除(DELETE语句的使用方法)

## 学习重点

- 如果想将整个表全部删除,可以使用DROP TABLE语句,如果只想删除表中全部数据,需使用DELETE语句。
- 如果想删除部分数据行,只需在WHERE子句中书写对象数据的条件即可。通过WHERE子句指定删除对象的DELETE语句称为搜索型DELETE语句。

## DROP TABLE 语句和DELETE 语句

上一节我们学习了插入数据的方法,本节我们来学习如何删除数据。删除数据的方法,大体可以分为以下两种。

## KEYWORD

- DROP TABLE 语句
- DELETE 语句

- ① DROP TABLE 语句可以将表完全删除。
- ② DELETE 语句会留下表(容器),而删除表中的全部数据。

①中的DROP TABLE语句我们已经在1-5节中学过了,在此再简单回顾一下。DROP TABLE语句会完全删除整张表,因此删除之后再想插入数据,就必须使用CREATE TABLE语句重新创建一张表。

反之,②中的DELETE语句在删除数据(行)的同时会保留数据表,因此只需要通过INSERT语句就可以再次向表中插入数据。

本节所要介绍的数据删除,指的就是只删除数据的DELETE语句。

此外,我们在第1章中也提到过,不管使用哪种方法,删除数据时都要慎重,一旦发生误删,想要恢复数据就会变得十分困难。

## DELETE 语句的基本语法

DELETE语句的基本语法如下所示,十分简单。

语法4-2 保留数据表,仅删除全部数据行的DELETE语句

```
DELETE FROM <表名>;
```

执行使用该基本语法的 DELETE 语句，就可以删除指定表中的全部数据行了。因此，想要删除 Shohin 表中全部数据行，就可以参见代码清单 4-13 来书写 DELETE 语句。

代码清单 4-13 清空 Shohin 表

```
DELETE FROM Shohin;
```

如果语句中忘了写 FROM，而是写成了 DELETE <表名>，或者写了多余的列名，都会出错，无法正常执行，请大家特别注意。

前者无法正常执行的原因是删除对象不是表，而是表中的数据行（记录）。这样想的话就很容易理解了吧<sup>①</sup>。

后者错误的原因也是如此。由于 DELETE 语句的对象是行而不是列，所以 DELETE 语句无法只删除部分列的数据。因此，在 DELETE 语句中指定列名是错误的。当然，使用星号的写法也是不对的（DELETE \* FROM Shohin;），同样会出错。

### ①

与 INSERT 语句相同，数据的更新也是以记录为基本单位进行的。下一节将要学习的 UPDATE 语句也是如此。



### 法则 4-4

DELETE 语句的删除对象并不是表或者列，而是记录（行）。

## 指定删除对象的 DELETE 语句（搜索型 DELETE）

想要删除部分数据行时，可以像 SELECT 语句那样使用 WHERE 子句指定删除条件。这种指定了删除对象的 DELETE 语句称为搜索型 DELETE<sup>①</sup>。

搜索型 DELETE 的语法如下所示。

语法 4-3 删除部分数据行的搜索型 DELETE

```
DELETE FROM <表名>  
WHERE <条件>;
```

下面让我们以 Shohin（商品）表为例，来具体研究一下如何进行数据删除（表 4-1）。

### KEYWORD

● 搜索型 DELETE

### ①

虽然“搜索型 DELETE”是正式用语，但实际上这种说法并不常用，而是简单地称作 DELETE 语句。

表 4-1 Shohin 表

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

假设我们要删除销售单价 (hanbai\_tanka) 大于等于 4000 日元的数据 (代码清单 4-14)。上述表中满足该条件的是“运动 T 恤”和“高压锅”。

代码清单 4-14 删除销售单价 (hanbai\_tanka) 大于等于 4000 日元的数据

```
DELETE FROM Shohin
WHERE hanbai_tanka >= 4000;
```

WHERE 子句的书写方式与此前介绍的 SELECT 语句完全一样。

经过 SELECT 语句确认, 表中的数据被删除了 2 行, 只剩下 6 行。

```
-- 确认删除结果
SELECT * FROM Shohin;
```

#### 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0004	菜刀	厨房用具	3000	2800	2009-09-20
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11



#### 法则 4-5

可以通过 WHERE 子句指定对象条件来删除部分数据。

与 SELECT 语句不同的是，DELETE 语句中不能使用 GROUP BY、HAVING 和 ORDER BY 三类子句，而只能使用 WHERE 子句。原因很简单，GROUP BY 和 HAVING 是从表中选取数据时用来改变抽取数据形式的，而 ORDER BY 是用来指定取得结果显示顺序的。因此，在删除表中数据时它们都起不到什么作用。

## 专 栏

### 删除和舍弃

标准 SQL 中用来从表中删除数据的只有 DELETE 语句。但是，很多数据库产品中还存在另外一种被称为 TRUNCATE 的语句。这些产品主要包括 Oracle、SQL Server、PostgreSQL 和 MySQL。DB2 中并没有 TRUNCATE 语句。

TRUNCATE 是舍弃的意思，具体的使用方法如下所示。

#### 语法 4-A 只能删除表中全部数据的 TRUNCATE 语句

```
TRUNCATE <表名>;
```

与 DELETE 不同的是，TRUNCATE 只能删除表中的全部数据，而不能通过 WHERE 子句指定条件来删除部分数据。也正是因为它不能具体地控制删除对象，所以其处理速度比 DELETE 要快得多。实际上，DELETE 语句在 DML 语句中也属于处理时间比较长的，因此需要删除全部数据行时，使用 TRUNCATE 可以缩短执行时间。

但是，产品不同需要注意的地方也不尽相同。例如在 Oracle 中，把 TRUNCATE 定义为 DDL，而不是 DML<sup>①</sup>。使用 TRUNCATE 时，请大家仔细阅读使用手册，多加注意。便利的工具往往还是会存在一些不足之处的。

### KEYWORD

- TRUNCATE 语句

### ①

因此，Oracle 中的 TRUNCATE 不能使用 ROLLBACK。TRUNCATE 执行的同时会默认执行 COMMIT 操作。

## 4-3

## 第4章 数据更新

## 数据的更新 (UPDATE 语句的使用方法)

## 学习重点

- 使用 UPDATE 语句可以更改 (更新) 表中的数据。
- 更新部分数据行时可以使用 WHERE 来指定更新对象的条件。通过 WHERE 子句指定更新对象的 UPDATE 语句称为搜索型 UPDATE 语句。
- UPDATE 语句可以将列的值更新为 NULL。
- 同时更新多列时, 可以在 UPDATE 语句的 SET 子句中, 使用逗号分隔更新对象的多个列。

## KEYWORD

- UPDATE 语句

## KEYWORD

- SET 子句

## UPDATE 语句的基本语法

使用 INSERT 语句向表中插入数据之后, 有时却想要再更改数据。例如, “将商品销售单价登记错了”等。这时并不需要把数据删除之后再重新插入, 只需要使用 UPDATE 语句就可以改变表中的数据了。

和 INSERT 语句、DELETE 语句一样, UPDATE 语句也属于 DML 语句。通过执行该语句, 我们就可以改变表中的数据了。其基本语法如下所示。

## 语法 4-4 改变表中数据的 UPDATE 语句

```
UPDATE <表名>
SET <列名> = <表达式>;
```

将更新对象的列和更新后的值都记述在 SET 子句中。我们还是以 Shohin (商品) 表为例, 由于之前我们删除了“销售单价大于等于 4000 日元”的 2 行数据, 现在该表中只剩下了 6 行数据 (表 4-2)。

表 4-2 Shohin 表

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T 恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0004	菜刀	厨房用具	3000	2800	2009-09-20
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

接下来，让我们尝试把 torokubi 列（登记日期）的所有数据，统一更新为“2009-10-10”。具体的 SQL 语句请参见代码清单 4-15。

代码清单 4-15 将登记日期全部更新为“2009-10-10”

```
UPDATE Shohin
SET torokubi = '2009-10-10';
```

表中的数据有何变化呢，我们通过 SELECT 语句来确认一下吧。

```
-- 确认更新内容
SELECT * FROM Shohin ORDER BY shohin_id;
```

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-10-10
0002	打孔器	办公用品	500	320	2009-10-10
0004	菜刀	厨房用具	3000	2800	2009-10-10
0006	叉子	厨房用具	500		2009-10-10
0007	擦菜板	厨房用具	880	790	2009-10-10
0008	圆珠笔	办公用品	100		2009-10-10

所有行的数据都被更新为“2009-10-10”

此时，连登记日期原本为 NULL 的数据行（运动 T 恤）的值也更新为 2009-10-10 了。

## 指定条件的 UPDATE 语句（搜索型 UPDATE）

接下来，让我们看一看指定更新对象的情况。更新数据时也可以像 DELETE 语句那样使用 WHERE 子句。这种指定更新对象的 UPDATE 语句称为搜索型 UPDATE 语句。该语句的语法如下所示（与 DELETE 语句十分相似）。

### KEYWORD

● 搜索型 UPDATE

语法 4-5 更新部分数据行的搜索型 UPDATE

```
UPDATE <表名>
SET <列名> = <表达式>
WHERE <条件>;
```

例如，如果将商品种类（shohin\_bunrui）为厨房用具的记录的销售



单价(hanbai\_tanka), 更新为原来 10 倍, 请参见代码清单 4-16。

代码清单 4-16 将商品种类为厨房用具的记录的销售单价更新为原来 10 倍

```
UPDATE Shohin
  SET hanbai_tanka = hanbai_tanka * 10
  WHERE shohin_bunrui = '厨房用具';
```

我们可以使用如下 SELECT 语句来确认更新后的内容。

```
-- 确认更新内容
SELECT * FROM Shohin ORDER BY shohin_id;
```

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-10-10
0002	打孔器	办公用品	500	320	2009-10-10
0004	菜刀	厨房用具	30000	2800	2009-10-10
0006	叉子	厨房用具	5000		2009-10-10
0007	煎菜板	厨房用具	8800	790	2009-10-10
0008	圆珠笔	办公用品	100		2009-10-10

仅仅厨房用具的价格更新为原来的10倍

该语句通过 WHERE 子句中的 shohin\_bunrui = '厨房用具' 条件, 将更新对象限定为 3 行。然后通过 SET 子句中的表达式 hanbai\_tanka \* 10, 将原来的单价扩大了 10 倍。SET 子句中赋值表达式的右边不仅可以是单纯的值, 还可以是包含列的表达式。

## 使用 NULL 进行更新

使用 UPDATE 也可以将列更新为 NULL (该更新俗称为 NULL 清空)。此时只需要将赋值表达式右边的值直接写为 NULL 即可。例如, 我们可以将商品编号(shohin\_id)为 0008 的数据(圆珠笔)的登记日期(torokubi)更新为 NULL (代码清单 4-17)。

代码清单 4-17 将商品编号为 0008 的数据(圆珠笔)的登记日期更新为 NULL

```
UPDATE Shohin
  SET torokubi = NULL
  WHERE shohin_id = '0008';
```

```
-- 确认更新内容
SELECT * FROM Shohin ORDER BY shohin_id;
```

### KEYWORD

● NULL 清空

## 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-10-10
0002	打孔器	办公用品	500	320	2009-10-10
0004	菜刀	厨房用具	30000	2800	2009-10-10
0006	叉子	厨房用具	5000		2009-10-10
0007	擦桌板	厨房用具	8800	790	2009-10-10
0008	圆珠笔	办公用品	100		

登记日期被更新为NULL

和 INSERT 语句一样, UPDATE 语句也可以将 NULL 作为一个值来使用。

但是, 只有未设置 NOT NULL 约束和主键约束的列才可以清空为 NULL。如果将设置了上述约束的列更新为 NULL, 就会出错, 这点与 INSERT 语句相同。



## 法则 4-6

使用 UPDATE 语句可以将值清空为 NULL (但只限于未设置 NOT NULL 约束的列)。

## 多列更新

UPDATE 语句的 SET 子句支持同时将多个列作为更新对象。例如我们刚刚将销售单价 (hanbai\_btanka) 更新为原来 10 倍, 如果想同时将进货单价 (shiire\_tanka) 更新为原来的一半, 该怎么做呢? 最容易想到的解决办法可能就是像代码清单 4-18 那样, 执行两条 UPDATE 语句。

代码清单 4-18 能够正确执行的繁琐的 UPDATE 语句

```
-- 一条UPDATE语句只更新一列
UPDATE Shohin
SET hanbai_tanka = hanbai_tanka * 10
WHERE shohin_bunrui = '厨房用具';

UPDATE Shohin
SET shiire_tanka = shiire_tanka / 2
WHERE shohin_bunrui = '厨房用具';
```

虽然这样也能够正确地更新数据, 但执行两次 UPDATE 语句不但不浪费, 而且增加了 SQL 语句的记述量。其实, 我们可以将其合并为一

条 UPDATE 语句来处理。合并的方法有两种, 请参见代码清单 4-19 和代码清单 4-20。

方法①: 代码清单 4-19 将代码清单 4-18 的处理合并为一条 UPDATE 语句

```
-- 使用逗号将列分隔排列
UPDATE Shohin
  SET hanbai_tanka = hanbai_tanka * 10,
      shiire_tanka = shiire_tanka / 2
  WHERE shohin_bunrui = '厨房用具';
```

方法②: 代码清单 4-20 将代码清单 4-18 的处理合并为一条 UPDATE 语句的

```
-- 将列用 ( ) 括起来的列表形式
UPDATE Shohin
  SET (hanbai_tanka, shiire_tanka) = (hanbai_tanka * 10, ←
  shiire_tanka / 2)
  WHERE shohin_bunrui = '厨房用具';
```

⇨表示下一行接续本行, 只是由于版面所限而换行。

执行上述两种 UPDATE 语句, 都可以得到相同的结果: 只有厨房用具的销售单价 (hanbai\_tannka) 和进货单价 (shiire\_tanka) 被更新了。

```
-- 确认更新内容
SELECT * FROM Shohin ORDER BY shohin_id;
```

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-10-10
0002	打孔器	办公用品	500	320	2009-10-10
0004	菜刀	厨房用具	100000	1400	2009-10-10
0006	叉子	厨房用具	50000	1000	2009-10-10
0007	煎菜板	厨房用具	80000	395	2009-10-10
0008	圆珠笔	办公用品	100		

厨房用具的销售单价更新为原来的10倍

厨房用具的进货单价更新为原来的一半

当然, SET 子句中的列不仅可以是两列, 还可以是三列或者更多。

需要注意的是第一种方法——使用逗号将列进行分隔排列 (代码清单 4-19), 这一方法在所有的 DBMS 中都可以使用。但是第二种方法——将列清单化 (代码清单 4-20), 这一方法在某些 DBMS 中是无法使用的<sup>①</sup>。因此, 实际应用中通常都会使用第一种方法。

①

可以在 PostgreSQL 和 DB2 中使用。

## 4-4

## 事务

## 学习重点

- 事务是需要在同一个处理单元中执行的一系列更新处理的集合。通过使用事务，可以对数据库中的数据更新处理的提交和取消进行管理。
- 事务处理的终止指令包括 COMMIT (提交处理) 和 ROLLBACK (取消处理) 两种。
- DBMS 的事务具有原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation) 和持久性 (Durability) 四种特性。通常将这四种特性的首字母结合起来，统称为 ACID 特性。

## 什么是事务

## KEYWORD

## ● 事务

估计有些读者对事务 (transaction) 这个词并不熟悉。它通常都带有一些商务贸易或者经济活动的意味。但是在 RDBMS 中，事务代表了对表中数据进行更新的单位。简单来讲，事务就是需要在同一个处理单元中执行的一系列更新处理的集合。

如前几节所述，对表进行更新需要使用 INSERT、DELETE 或者 UPDATE 三种语句。但通常情况下，更新处理并不是执行一次就结束了，而是需要执行一系列连续的操作。这时，事务就能体现出它的价值了。

说到事务的例子，请大家思考一下下述情况。

现在，请大家把自己想象为管理 Shohin (商品) 表的程序员或者软件工程师。销售部门的领导对你提出了如下要求。

“某某，经会议讨论，我们决定把运动 T 恤的销售单价下调 1000 日元，同时把 T 恤的销售单价上浮 1000 日元，麻烦你去更新一下数据库。”

由于大家已经学习了更新数据的方法——只需要使用 UPDATE 进行更新就可以了，所以肯定会直接回答“知道了，请您放心吧”。

此时的事务就是由如下两条更新处理所组成。

### ●更新商品信息的事务

①将运动T恤的销售单价降低1000日元。

```
UPDATE Shohin
   SET hanbai_tanka = hanbai_tanka - 1000
   WHERE shohin_mei = '运动T恤';
```

②将T恤的销售单价上浮1000日元。

```
UPDATE Shohin
   SET hanbai_tanka = hanbai_tanka + 1000
   WHERE shohin_mei = 'T恤衫';
```

上述①和②的操作一定要作为同一个处理单元执行。如果只执行了①的操作而忘记了执行②的操作，或者反过来只执行了②的操作而忘记了执行①的操作，一定会受到领导的严厉批评。遇到这种需要在同一个处理单元中执行一系列更新操作的情况，一定要使用事务来进行处理。



#### 法则4-7

事务是需要要在同一个处理单元中执行的一系列更新处理的集合。

一个事务中包含多少个更新处理或者包含哪些处理，在DBMS中并没有固定的标准。而是根据用户的要求决定的（例如，运动T恤和T恤的销售单价需要同时更新这样的要求，DBMS是无法了解的）。

## 创建事务

如果想在DBMS中创建事务，可以按照如下语法结构编写SQL语句。

### 语法4-6 事务的语法

事务开始语句；

DML语句①；

DML语句②；

DML语句③；

⋮

事务结束语句（COMMIT或者ROLLBACK）；

使用事务开始语句和事务结束语句，将一系列DML语句（INSERT/UPDATE/DELETE语句）括起来，就实现了一个事务处理。

## ①

与之相对,事务结束语句只有 COMMIT 和 ROLLBACK 两种,在所有的 DBMS 中都是通用的。

## KEYWORD

- BEGIN TRANSACTION
- START TRANSACTION

这时需要特别注意的是事务的开始语句<sup>①</sup>。实际上,在标准 SQL 中并没有定义事务的开始语句,而是由各个 DBMS 自己来定义的。比较有代表性的语法如下所示。

## ● SQL Server、PostgreSQL

```
BEGIN TRANSACTION
```

## ● MySQL

```
START TRANSACTION
```

## ● Oracle、DB2

无

例如使用之前的那两个 UPDATE (①和②) 创建出的事务如代码清单 4-21 所示。

## 代码清单 4-21 更新商品信息的事务

```
SQL Server PostgreSQL  
BEGIN TRANSACTION;
```

```
-- 将运动T恤的销售单价降低1000日元  
UPDATE Shohin  
SET hanbai_tanka = hanbai_tanka - 1000  
WHERE shohin_mei = '运动T恤';  
  
-- 将T恤的销售单价上浮1000日元  
UPDATE Shohin  
SET hanbai_tanka = hanbai_tanka + 1000  
WHERE shohin_mei = 'T恤衫';
```

```
COMMIT;
```

```
MySQL  
START TRANSACTION;
```

```
-- 将运动T恤的销售单价降低1000日元  
UPDATE Shohin  
SET hanbai_tanka = hanbai_tanka - 1000  
WHERE shohin_mei = '运动T恤';  
  
-- 将T恤的销售单价上浮1000日元  
UPDATE Shohin  
SET hanbai_tanka = hanbai_tanka + 1000  
WHERE shohin_mei = 'T恤衫';
```

```
COMMIT;
```

```
Oracle DB2  
-- 将运动T恤的销售单价降低1000日元  
UPDATE Shohin  
SET hanbai_tanka = hanbai_tanka - 1000  
WHERE shohin_mei = '运动T恤';
```

```
-- 将T恤的销售单价上浮1000日元
UPDATE Shohin
   SET hanbai_tanka = hanbai_tanka + 1000
   WHERE shohin_mei = 'T恤衫';
COMMIT;
```

如上所示，各个 DBMS 事务的开始语句都不尽相同。其中 Oracle 和 DB2 并没有定义特定的开始语句。可能大家觉得这样的设计很巧妙，其实是因为标准 SQL 中规定了一种悄悄开始事务处理<sup>①</sup>的方法。因此，即使是经验丰富的工程师也经常忽略事务处理开始的时点。大家可以试着通过“是否知道某个 DBMS 中事务是什么时候开始的”这样的询问，来测试学校或者公司前辈的数据库知识。

反之，事务的结束需要用户明确地给出指示。事务结束的指令有如下两种。

#### ■ COMMIT——提交处理

COMMIT 是提交事务包含的全部更新处理的结束指令（图 4-3）。相当于文件处理中的覆盖保存。一旦提交，就无法恢复到事务开始前的状态了。因此，在提交之前一定要确认是否真的需要进行这些更新。

图 4-3 COMMIT 的流程 = 直线进行



结束后的状态：②中的所有更新都被反映到了数据库中

万一由于误操作提交了包含错误更新的事务，就只能重新回到重新建表、重新插入数据这样繁琐的老路上了。由于可能会造成数据无法恢复的后果，请大家一定要注意（特别是在执行 DELETE 语句的 COMMIT 时尤其要小心）。



#### 法则 4-8

虽然我们可以不清楚事务开始的时点，但是在事务结束时一定要仔细进行确认。

#### ①

（标准 SQL 手册 修订第 4 版）中的记述：希望大家注意事务默认开始的时点。没有“BEGIN TRANSACTION”这样明确的开始标志。

#### KEYWORD

- COMMIT
- 提交

**KEYWORD**

- ROLLBACK
- 回滚

**■ ROLLBACK——取消处理**

ROLLBACK 是取消事务包含的全部更新处理的结束指令（图 4-4）。相当于文件处理中的放弃保存。一旦回滚，数据库就会回复到事务开始之前的状态（代码清单 4-22）。通常回滚并不会像提交那样造成大规模的数据损失。

图 4-4 ROLLBACK 的流程 == 掉头回到起点



结束后的状态：和①执行前相同

代码清单 4-22 事务回滚的例子

```

SQL Server PostgreSQL
BEGIN TRANSACTION; ①
-- 将运动T恤的销售单价降低1000日元
UPDATE Shohin
   SET hanbai_tanka = hanbai_tanka - 1000
   WHERE shohin_mei = '运动T恤';
-- 将T恤的销售单价上浮1000日元
UPDATE Shohin
   SET hanbai_tanka = hanbai_tanka + 1000
   WHERE shohin_mei = 'T恤衫';
ROLLBACK;
  
```

**特定的 SQL**

通过此前所学，我们已经知道各个 DBMS 中关于事务的语法不尽相同。代码清单 4-22 中的语句在 MySQL 中执行时需要将①语句改写为“START TRANSACTION”。而在 Oracle 和 DB2 中执行时无须①语句（请将其删除）。具体请参考“创建事务”。

上述事务处理执行之后，表中的数据不会发生任何改变。这是因为执行最后一行的 ROLLBACK 之后，所有的处理都被取消了。因此，回滚执行起来就无需像提交时那样小心翼翼了（即使是想要提交的情况，也只需要重新执行事务处理就可以了）。



## 专 栏

## 事务处理何时开始

之前我们说过，事务并没有标准的开始指令存在，而是根据 DBMS 的不同而不同。

实际上，几乎所有的数据库产品的事务都无需开始指令。这是因为大部分情况下，事务在数据库连接建立时就已经悄悄开始了，并不需要用户再明确发出开始指令。例如，使用 Oracle 时，数据库连接建立之后，第一条 SQL 语句执行的同时，事务就已经悄悄开始了。

像这样不使用指令而悄悄开始事务的情况下，应该如何区分各个事务呢？通常会有如下两种情况。

- ④ 每条 SQL 语句就是一个事务（自动提交模式）。
- ⑤ 直到用户执行 COMMIT 或者 ROLLBACK 为止算作一个事务。

通常的 DBMS 都可以选择其中任意一种模式。默认使用自动提交模式的 DBMS 有 SQL Server、PostgreSQL 和 MySQL 等<sup>④</sup>。该模式下的 DML 语句如下所示，每一条语句都括在事务的开始语句和结束语句之中。

```
BEGIN TRANSACTION;
-- 将运动T恤的销售单价降低1000日元
UPDATE Shohin
SET hanbai_tanka = hanbai_tanka - 1000
WHERE shohin_mei = '运动T恤';
COMMIT;

BEGIN TRANSACTION;
-- 将T恤的销售单价上浮1000日元
UPDATE Shohin
SET hanbai_tanka = hanbai_tanka + 1000
WHERE shohin_mei = 'T恤衫';
COMMIT;
```

在默认使用 B 模式的 Oracle 中，事务都是直到用户自己执行提交或者回滚指令才会结束。

自动提交的情况需要特别注意的是 DELETE 语句。如果不是自动提交，即使使用 DELETE 语句删除了数据表，也可以通过 ROLLBACK 命令取消该事务的处理，恢复表中的数据。但这仅限于明示开始事务，或者关闭自动提交的情况。如果不小心在自动提交模式下执行了 DELETE 操作，即使再回滚也无济于事了。这是一个很严重的问题，初学者难免会碰到这样的麻烦。一旦误删了数据，如果无法重新插入，是不是想哭的心都有了？所以一定要特别小心。

## KEYWORD

- 自动提交模式

## ④

例如，PostgreSQL 的用户手册中有如下记述。“PostgreSQL 中所有的 SQL 指令语句都在事务内执行。即使不执行 BEGIN，这些命令语句也会在运行时悄悄被括在 BEGIN 和 COMMIT（如果成功的话）之间。”（《PostgreSQL 8.3.4 文档》“3-4 节 事务”）

**KEYWORD**

- ACID 特性

**KEYWORD**

- 原子性 (Atomicity)

**KEYWORD**

- 一致性 (Consistency)
- 完整性

## ACID 特性

DBMS 的事务都遵循四种标准规格的约定。将这四种特性的首字母组合起来统称为 ACID 特性。这些约定是所有 DBMS 都必须遵守的规则。

### ■ 原子性 (Atomicity)

原子性是指在事务结束时，其中所包含的更新处理要么全部执行，要么完全不执行的特性。也就是要么占有一切要么一无所有。例如，在之前的例子中，事务结束时，是绝对不可能出现运动 T 恤的价格下降了，而 T 恤的价格却没有上涨的情况。该事务的结束状态，要么是两者都执行了 (COMMIT)，要么是两者都未执行 (ROLLBACK)。

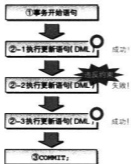
从事务中途停止的角度去考虑，就能比较容易理解原子性的重要性了。由于用户在一个事务中定义了两条 UPDATE 语句，DBMS 肯定不会只执行其中一条，否则就会对业务处理造成影响。

### ■ 一致性 (Consistency)

一致性指的是事务中包含的处理，要满足数据库提前设置的约束，如主键约束或者 NOT NULL 约束等。例如，设置了 NOT NULL 约束的列是不能更新为 NULL 的，试图插入违反主键约束的记录就会出错，无法执行。对事务来说，这些不合法的 SQL 会被回滚。也就是说这些 SQL 处理会被取消，不会执行。

一致性也称为完整性 (图 4-5)。

图 4-5 保持完整性的流程



结束后的状态：只有②-2的更新没有被反映到数据库中

**KEYWORD**

- 隔离性 ( Isolation )

**■ 隔离性 ( Isolation )**

隔离性指的是保证不同事务之间互不干扰的特性。该特性保证了事务之间不会互相嵌套。此外，在某个事务中进行的更改，在该事务结束之前，对其他事务而言是不可见的。因此，即使某个事务向表中添加了记录，在没有提交之前，其他事务是看不到新添加的记录的。

**KEYWORD**

- 持久性 ( Durability )
- 日志

**■ 持久性 ( Durability )**

持久性也可以称为耐久性，指的是事务（不论是提交还是回滚）一旦结束，DBMS 会保证该时点的数据状态得以保存的特性。即使由于系统故障导致数据丢失，数据库也一定能通过某种手段进行恢复。

如果不能保证持久性，即使是正常提交结束的事务，一旦发生了系统故障，就会导致数据丢失，一切都需要从头再来的后果。

保证持久性的方法根据实现的不同而不同，其中最常见的就是将事务的执行记录保存到硬盘等存储介质中（该执行记录称为日志）。当发生故障时，可以通过日志恢复到故障发生前的状态。

**练习题**

4.1 A 先生在自己的计算机（电脑）上，使用 CREATE TABLE 语句创建出了一张空的 Shohin（商品）表，并执行了如下的 SQL 语句向其中插入数据。

```
BEGIN TRANSACTION;
  INSERT INTO Shohin VALUES ('0001', 'T恤衫', →
'衣服', 1000, 500, '2008-09-20');
  INSERT INTO Shohin VALUES ('0002', '打孔器', →
'办公用品', 500, 320, '2008-09-11');
  INSERT INTO Shohin VALUES ('0003', '运动T恤', →
'衣服', 4000, 2800, NULL);
```

→表示下一行接续本行，只是由于版面所限而换行。

紧接着，B 先生使用其他的计算机连接上该数据库，执行了如下 SELECT 语句。这时 B 先生能得到怎样的查询结果呢？

```
SELECT * FROM Shohin;
```

提示：如果可以使用 DELETE 语句，就可以对通过 CREATE TABLE 语句创建出的空表执行该操作了。

4.2 如下所示，有包含3条记录的 Shohin 表。

商品编号	商品名称	商品分类	销售单价	进货单价	登记日期
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	

使用如下的 INSERT 语句复制这3行数据，应该就能够将表中的数据增加为6行。请说出该语句的执行结果。

```
INSERT INTO Shohin SELECT * FROM Shohin;
```

4.3 以练习 4.2 中的 Shohin 表为基础，再创建另外一张包含利润列的新表 ShohinSaeki (商品利润)。

```
-- 商品利润表
CREATE TABLE ShohinSaeki
(shohin_id CHAR(4) NOT NULL,
shohin_mei VARCHAR(100) NOT NULL,
hanbai_tanka INTEGER,
shiire_tanka INTEGER,
saeki INTEGER,
PRIMARY KEY(shohin_id));
```

请写出向上述表中插入如下数据的 SQL 语句，其中的利润可以简单地通过对 Shohin 表中的数据进行计算 (销售单价 - 进货单价) 得出。

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka	saeki
0001	T恤衫	1000	500	500
0002	打孔器	500	320	180
0003	运动T恤	4000	2800	1200

4.4 对练习 4.3 中的 ShohinSaeki 表的数据进行如下更改。

1. 将运动 T 恤的销售单价从 4000 日元下调至 3000 日元。
2. 根据上述结果再次计算运动 T 恤的利润。

更改后的 ShohinSaeki 表如下所示。请写出能够实现该变更的 SQL 语句。

shohin_id	shohin_mei	hanbai_tanka	shiire_tanka	saeki
0001	T恤衫	1000	500	500
0002	打孔器	500	320	180
0003	运动T恤	3000	2800	200

销售单价和  
利润都发生  
← 了改变

## 第5章 复杂查询

视图

子查询

关联子查询

# 5



# SQL

## 本章重点

---

前几章和大家一起学习了表的创建、查询和更新这样一整套数据库的操作方法。从本章开始，大家将会在这些基本方法的基础上，学习一些实际应用中的方法。

本章将以此前学过的 SELECT 语句，以及嵌套在 SELECT 语句中的视图和子查询等技术为中心进行学习。由于视图和子查询可以像表一样进行使用，所以如果能恰当的使用这些技术，就可以写出更加灵活的 SQL 了。

### 5-1 视图

- 视图和表

- 创建的视图方法

- 视图的限制①——定义视图时不能使用 ORDER BY 子句

- 视图的限制②——对视图进行更新

- 删除视图

### 5-2 子查询

- 子查询和视图

- 子查询的名称

- 标量子查询

- 标量子查询的书写位置

- 使用标量子查询时的注意事项

### 5-3 关联子查询

- 普通的子查询和关联子查询的区别

- 关联子查询也是用来对集合进行切分的

- 结合条件一定要写在子查询中

## 5-1

## 第5章 复杂查询

## 视图

## 学习重点

- 从SQL的角度来看，视图和表是相同的。两者的区别在于表中保存的是实际的数据，而视图中保存的是SELECT语句（视图本身并不存储数据）。
- 使用视图，可以轻松完成跨多表查询数据等复杂操作。
- 可以将常用的SELECT语句做成视图来使用。
- 创建视图需要使用CREATE VIEW语句。
- 视图包含“不能使用ORDER BY”和“可对其进行有限制的更新”两项限制。
- 删除视图需要使用DROP VIEW语句。

## KEYWORD

- 视图

## 视图和表

我们首先要学习的是一个新的工具——视图。

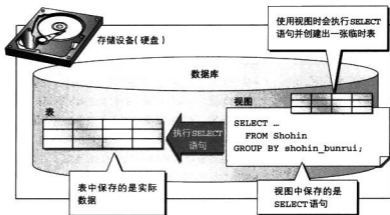
究竟视图是什么呢？如果用一句话概述的话，就是“从SQL的角度来看视图就是一张表”。实际上，在SQL语句中并不需要区分哪些是表，哪些是视图。只需要知道在更新时它们之间存在一些不同就可以了，之后会为大家进行介绍。至少在编写SELECT语句时并不需要特别在意表和视图有什么不同。

那么视图和表到底有什么不同呢？区别只有一个，那就是“是否保存了实际的数据”。

通常，我们在创建表时，会通过INSERT语句将数据保存到数据库之中。而数据库中的数据实际上会被保存到计算机的存储设备（通常是硬盘）中。因此，我们通过SELECT语句查询数据时，实际上就是从存储设备（硬盘）中读取数据，进行各种计算之后，再将结果返回给用户这样一个过程。

但是使用视图时并不会将数据保存到存储设备之中，而且也不会将数据保存到其他任何地方。实际上视图保存的是SELECT语句（图5-1）。我们从视图中读取数据时，视图会在内部执行该SELECT语句并创建出一张临时表。

图5-1 视图和表



### ■视图的优点

视图的优点大体有两点。

第一点是由于视图无需保存数据，因此可以节省存储设备的容量。例如，我们在4-1节中创建了用来汇总商品种类(shohin\_bunrui)的表。由于该表中的数据最终都会保存到存储设备之中，因此会占用存储设备的数据领域。但是，如果把同样的数据作为视图保存起来的话，就只需要代码清单5-1那样的SELECT语句就可以了，这样就节省了存储设备的数据领域。

#### 代码清单5-1 通过视图等SELECT语句保存数据

```
SELECT shohin_bunrui, SUM(hanbai_tanka), SUM(shiire_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

由于本示例中表的数据量充其量只有几行，所以使用视图并不会大幅缩小数据的大小。但是在实际的业务中数据量往往非常大，这时使用视图所节省的容量就会非常可观了。



#### 法则5-1

表中存储的是实际数据，而视图中保存的是从表中取出数据所使用的SELECT语句。

第二个优点就是可以将频繁使用的SELECT语句保存成视图，这样



就不用每次都重新书写了。创建好视图之后，只需在 `SELECT` 语句中进行调用，就可以方便地得到想要的结果了。特别是在计算合计，以及由于包含复杂的查询条件导致 `SELECT` 语句非常庞大时，使用视图可以大大提高效率。

而且，视图中的数据会随着原表的变化自动更新。视图归根到底就是 `SELECT` 语句，所谓“参照视图”也就是“执行 `SELECT` 语句”的意思。因此可以保证数据的最新状态。这也是将数据保存在表中所不具备的优势<sup>①</sup>。

#### ①

数据保存在表中时，必须要执行明示的 SQL 更新语句才能对数据进行更新。



#### 法则 5-2

应该将经常使用的 `SELECT` 语句做成视图。

#### KEYWORD

● `CREATE VIEW` 语句

## 创建视图的方法

创建视图需要使用 `CREATE VIEW` 语句。语法如下所示。

#### 语法 5-1 创建视图的 `CREATE VIEW` 语句

```
CREATE VIEW 视图名称 (<视图列名 1>, <视图列名 2>, ……)
AS
<SELECT 语句>
```

`SELECT` 语句需要书写在 `AS` 关键字之后。`SELECT` 语句中列的排列顺序和视图中列的排列顺序相同，`SELECT` 语句中的第 1 列就是视图中的第 1 列，`SELECT` 语句中的第 2 列就是视图中的第 2 列，以此类推。视图的列名在视图名称之后的列表中定义。

#### 备忘

接下来，我们将会以此前使用的 `Shohin`（商品）表为基础来创建视图。如果大家已经根据之前章节的内容更新了 `Shohin` 表中的数据，请在创建视图之前将数据恢复到初始状态。操作步骤如下所示。

① 删除 `Shohin` 表中的数据，将表清空。

```
DELETE FROM Shohin;
```

②执行代码清单 1-6 中的 SQL 语句，将数据插入到空表 Shohin 中。

②中的 SQL 语句 (CreateTableShohin.sql) 收录在随书光盘的 CD-ROM\Sample\CreateTable\PostgreSQL 目录中。

下面就让我们试着来创建视图吧。和此前一样，这次我们还是将 Shohin 表（代码清单 5-2）作为基本表。

代码清单 5-2 ShohinSum 视图

```
CREATE VIEW ShohinSum (shohin_bunrui, cnt_shohin)
AS
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui;
```

视图的列名

视图定义中的主体（内容只是一条 SELECT 语句）

这样我们就在数据库中创建出了一幅名为 ShohinSum（商品合计）的视图。请大家一定不要省略第 2 行的关键字 AS。这里的 AS 与定义别名时使用的 AS 并不相同，如果省略就会发生错误。虽然很容易混淆，但是语法就是这么决定的，所以还是请大家将其当作一条规定加以牢记。

接下来，我们来学习视图的使用方法。视图和表一样，可以书写在 SELECT 语句的 FROM 子句之中（代码清单 5-3）。

代码清单 5-3 使用视图

```
SELECT shohin_bunrui, cnt_shohin
FROM ShohinSum;
```

在 FROM 子句中使用视图来代替表

执行结果

shohin_bunrui	cnt_shohin
衣服	2
办公用品	2
厨房用具	4

通过上述视图 ShohinSum 定义的主体（SELECT 语句）我们可以看出，该视图将根据商品种类 (shohin\_bunrui) 统计出的商品数量 (cnt\_shohin) 作为结果保存了起来。这样如果大家在工作中需要频繁进行统计时，就不用每次都书写使用 GROUP BY 和 COUNT 函数的 SELECT 语句，从 Shohin 表中取得数据了。创建出视图之后，就可以通过非常简单的

SELECT 语句，随时得到想要的合计结果了。并且如前所述，Shohin 表中的数据更新之后，视图也会自动更新，非常灵活方便。

之所以能够实现上述功能，是因为视图就是保存好的 SELECT 语句。定义视图时可以使用任何 SELECT 语句。既可以使用 WHERE、GROUP BY、HAVING，也可以通过 SELECT \* 来指定全部列。

### ■使用视图的查询

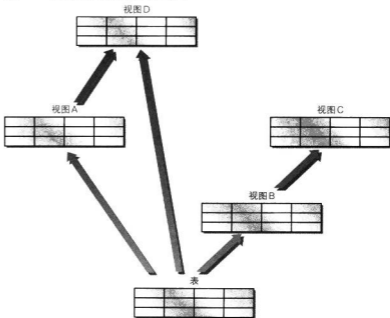
在 FROM 子句中使用视图的查询，通常有如下两个步骤：

- ① 首先执行定义视图的 SELECT 语句，
- ② 根据得到的结果，再执行在 FROM 子句中使用视图的 SELECT 语句

也就是说，使用视图的查询通常需要执行 2 条以上的 SELECT 语句<sup>①</sup>。

这里没有使用“2 条”而使用了“2 条以上”，是因为还可能出现以视图为基础创建出的类似楼中楼那样的多重视图（图 5-2）。例如，我们可以像代码清单 5-4 那样以 ShohinSum 为基础创建出视图 ShohinSumJim。

图 5-2 可以在视图的基础上创建视图



#### ①

但是根据实现方式的不同，也存在内部使用视图的 SELECT 语句本身进行重组的 DBMS。

#### KEYWORD

●多重视图

代码清单 5-4 视图 ShohinSumJim

```
CREATE VIEW ShohinSumJim (shohin_bunrui, cnt_shohin)
AS
SELECT shohin_bunrui, cnt_shohin
FROM ShohinSum ← 以视图为基础创建视图
WHERE shohin_bunrui = '办公用品';

-- 确认是否创建出了视图
SELECT shohin_bunrui, cnt_shohin
FROM ShohinSumJim;
```

## 执行结果

shohin_bunrui	cnt_shohin
办公用品	2

虽然语法上没有错误，但是我们还是应该尽量避免在视图的基础上创建视图。这是因为对多数 DBMS 来说，多重视图会降低 SQL 的性能。因此，希望大家（特别是刚刚接触视图的读者）能够使用单一视图。



## 法则 5-3

应该避免在视图的基础上创建视图。

除此之外，在使用时还要注意视图有两个限制，接下来会给大家详细介绍。

## 视图的限制①——定义视图时不能使用 ORDER BY 子句

虽然之前我们说过在定义视图时可以使用任何 SELECT 语句，但其实有一种情况例外，那就是不能使用 ORDER BY 子句。因此下述的视图定义语句是错误的。

```
-- 不能像这样定义视图
CREATE VIEW ShohinSum (shohin_bunrui, cnt_shohin)
AS
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui
ORDER BY shohin_bunrui; ← 定义视图时不能使用 ORDER BY 子句
```

**0**

例如，在 PostgreSQL 中上述 SQL 语句就没有问题，可以执行。

为什么不能使用 ORDER BY 子句呢？这是因为视图和表一样，数据行都是没有顺序的。实际上，有些 DBMS 在定义视图的语句中是可以使用 ORDER BY 子句的<sup>①</sup>，但是这并不是通用的语法。因此，在定义视图时请不要使用 ORDER BY 子句。



**法则 5-4**

定义视图时不要使用 ORDER BY 子句。

## 视图的限制②——对视图进行更新

之前我们说过，在 SELECT 语句中视图可以和表一样进行使用。那么，对于 INSERT、DELETE、UPDATE 这类更新语句（更新数据的 SQL）来说会怎么样呢？

实际上，虽然这其中有很严格的限制，但是某些时候也可以对视图进行更新。标准 SQL 中有这样的规定：如果定义视图的 SELECT 语句能够满足某些条件，那么这个视图就可以被更新。下面就给大家列举一些比较具有代表性的条件。

- ① SELECT 子句中未使用 DISTINCT
- ② FROM 子句中只有一张表
- ③ 未使用 GROUP BY 子句
- ④ 未使用 HAVING 子句

此前几章的例子中，FROM 子句里通常只有一张表。因此，大家可能会觉得②中的条件有些奇怪，但其实 FROM 子句中也可以并列使用多张表。大家在学习完下一章“表结合”的操作之后就明白了。

其他的条件大多数都与聚合有关。简单来说，像这次的例子中使用的 ShohinSum 那样，使用视图来保存原表聚合结果时，是无法判断如何将视图的更改反映到原表中的。

例如，对 ShohinSum 视图执行如下 INSERT 语句。

```
INSERT INTO ShohinSum VALUES ('电器制品', 5);
```





## 法则 5-5

视图和表需要同时进行更新，因此通过聚合得到的视图无法进行更新。

## ■能够更新视图的情况

像代码清单 5-5 这样，不是通过聚合得到的视图就可以进行更新。

## 代码清单 5-5 可以更新的视图

```
CREATE VIEW ShohinJim (shohin_id, shohin_mei, shohin_bunrui,
hanbai_tanka, shiire_tanka, torokubi)
AS
SELECT *
FROM Shohin
WHERE shohin_bunrui = '办公用品';
```

← 既没有聚合又没有结合的SELECT语句

→表示下一行接续本行，只是由于版面所限而换行。

对于上述只包含办公用品类商品的视图 ShohinJim 来说，就可以执行类似代码清单 5-6 这样的 INSERT 语句。

## 代码清单 5-6 向视图中添加数据行

```
INSERT INTO ShohinJim VALUES ('0009', '印章', '办公用品', 95, 10,
'2009-11-30');
```

↑  
向视图中添加一行

→表示下一行接续本行，只是由于版面所限而换行。

## 注意事项

由于 PostgreSQL 中的视图会被初始设定为只读，所以执行代码清单 5-6 中的 INSERT 语句时，会发生下面这样的错误。

## 执行结果 (使用 PostgreSQL)

```
ERROR: 不能向视图中插入数据
HINT: 需要一个无条件的ON INSERT DO INSTEAD规则。
```

因此，在执行 INSERT 语句执行之前，需要使用代码清单 5-A 中的指令来允许更新操作。在 DB2 和 MySQL 等其他 DBMS 中，并不需要执行这样的指令。

## 代码清单 5-A 允许 PostgreSQL 对视图进行更新

```
PostgreSQL
CREATE OR REPLACE RULE insert_rule
AS ON INSERT
TO ShohinJim DO INSTEAD
INSERT INTO Shohin VALUES (
new.shohin_id,
```

```
new.shohin_mei,
new.shohin_bunrui,
new.hanbai_tanka,
new.shiire_tanka,
new.torokubi);
```

下面让我们使用 SELECT 语句来确认数据行是否添加成功吧。

### ● 视图

```
-- 确认数据是否已经添加到视图中了
SELECT * FROM ShohinJim;
```

#### 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0002	打孔器	办公用品	500	320	2009-09-11
0008	圆珠笔	办公用品	100		2009-11-11
0009	印章	办公用品	95	10	2009-11-30

数据已经被添加进来了

### ● 原表

```
-- 确认数据是否已经添加到原表中了
SELECT * FROM Shohin;
```

#### 执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11
0009	印章	办公用品	95	10	2008-11-30

数据已经被添加进来了

UPDATE 语句和 DELETE 语句当然也可以像操作表时那样正常执行，但是对于原表来说却需要设置各种各样的约束（主键和 NOT NULL 等），需要特别注意。



**KEYWORD**

● DROP VIEW 语句

**删除视图**

删除视图需要使用 DROP VIEW 语句。语法如下所示。

语法 5-2 删除视图的 DROP VIEW 语句

```
DROP VIEW 视图名称 (<视图列名 1>, <视图列名 2>, ……)
```

例如，想要删除视图 ShohinSum 时，就可以使用代码清单 5-7 中的 SQL 语句。

代码清单 5-7 删除视图

```
DROP VIEW ShohinSum;
```

**特定的 SQL**

在 PostgreSQL 中，如果想要删除以视图为基础创建出来的多重视图的话，由于存在关联的视图，所以会发生如下错误。

**执行结果 (使用 Postgre)**

```
ERROR: 由于存在关联视图，所以无法删除视图shohinsum
DETAIL: 视图shohinsumjim与视图shohinsum相关联
HINT: 删除关联对象请使用DROP ..CASCADE ➡
```

➡表示下一行接续本行，只是由于版面所限而换行。

这时可以像下面这样，使用 CASCADE 选项来删除关联视图。

**PostgreSQL**

```
DROP VIEW ShohinSum CASCADE;
```

**备忘**

下面我们再次将 Shohin 表恢复到初始状态 (8 行)。请执行如下 DELETE 语句，删除之前添加的 1 行数据。

代码清单 5-B

```
-- 删除商品编号为0009 (印章) 的数据
DELETE FROM Shohin WHERE shohin_id = '0009';
```

## 5-2

## 第5章 复杂查询

## 子查询

## 学习重点

- 一言以蔽之，子查询就是一次性的视图（SELECT 语句）。与视图不同，子查询在 SELECT 语句执行完毕之后就会消失。
- 由于子查询需要命名，因此需要根据处理内容来指定恰当的名称。
- 标量子查询就是只能返回一行一列的子查询。

## 子查询和视图

前一节我们学习了视图这个非常方便的工具，本节将学习以视图为基础的子查询。子查询的特点概括起来就是一张一次性视图。

我们先来复习一下视图的概念，视图并不是用来保存数据的，而是通过保存读取数据的 SELECT 语句的方法来为用户提供便利的工具。反之，子查询就是将来定义视图的 SELECT 语句直接用于 FROM 子句当中。接下来，就让我们拿前一节使用的视图 ShohinSum（商品合计）来与子查询进行一番比较吧。

首先，我们再来看一下视图 ShohinSum 的定义和视图所对应的 SELECT 语句（代码清单 5-8）。

代码清单 5-8 视图 ShohinSum 和确认用的 SELECT 语句

```
-- 根据商品种类统计商品数量的视图
CREATE VIEW ShohinSum (shohin_bunrui, cnt_shohin)
AS
SELECT shohin_bunrui, COUNT(*)
FROM Shohin
GROUP BY shohin_bunrui;

-- 确认视图是否已经创建成功
SELECT shohin_bunrui, cnt_shohin
FROM ShohinSum;
```

能够实现同样功能的子查询如代码清单 5-9 所示。

## KEYWORD

- 子查询

## 代码清单 5-9 子查询

```
SQL Server 087 PostgreSQL MySQL
-- 直接在FROM子句中使用定义视图的SELECT语句
SELECT shohin_bunrui, cnt_shohin
FROM (SELECT shohin_bunrui, COUNT(*) AS cnt_shohin
      FROM Shohin
      GROUP BY shohin_bunrui) AS ShohinSum; ①
```

直接使用定义视图的SELECT语句

## 特定的SQL

在 Oracle 的 FROM 子句中，不能使用 AS [ 会发生错误 ]。因此，在 Oracle 中执行代码清单 5-9 时，需要将 ① 中的 “) AS ShohinSum;” 变为 “) ShohinSum;”。

两种方法得到结果完全相同。

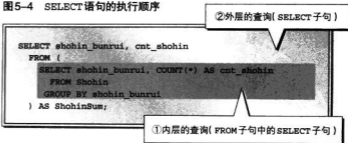
## 执行结果

shohin_bunrui	cnt_shohin
衣服	2
办公用品	2
厨房用具	4

如上所示，子查询就是将用来定义视图的 SELECT 语句直接用于 FROM 子句当中。虽然 “AS ShohinSum” 就是子查询的名称，但由于该名称是一次性的，因此不会像视图那样保存在存储介质（硬盘）之中，而是在 SELECT 语句执行之后就消失了。子查询(subquery)就是“次级(sub)”的“查询(query)”。

实际上，该 SELECT 语句包含嵌套的结构，首先会执行 FROM 子句中的 SELECT 语句，然后才会执行外层的 SELECT 语句（图 5-4）。

图 5-4 SELECT 语句的执行顺序



- ① 首先执行 FROM 子句中的 SELECT 语句（子查询）。

```
SELECT shohin_bunrui, COUNT(*) AS cnt_shohin
FROM Shohin
GROUP BY shohin_bunrui;
```

- ② 根据①的结果执行外层的 SELECT 语句。

```
SELECT shohin_bunrui, cnt_shohin
FROM ShohinSum;
```



### 法则 5-6

子查询作为内层查询会首先执行。

### ■增加子查询的层数

由于子查询的层数原则上没有限制，因此可以像“子查询的 FROM 子句中还可以继续使用子查询，该子查询的 FROM 子句中还可以再使用子查询……”这样无限嵌套下去（代码清单 5-10）。

代码清单 5-10 尝试增加子查询的嵌套层数

```
SQL Server DB2 PostgreSQL MySQL
SELECT shohin_bunrui, cnt_shohin
FROM (SELECT *
      FROM (SELECT shohin_bunrui, COUNT(*) AS cnt_shohin
            FROM Shohin
            GROUP BY shohin_bunrui) AS ShohinSum ——①
      WHERE cnt_shohin = 4) AS ShohinSum2; ——②
```

#### 特定的 SQL

在 Oracle 的 FROM 子句中不能使用 AS（会发生错误）。因此，在 Oracle 中执行代码清单 5-10 时，需要将①中的“] AS ShohinSum;”变为“ShohinSum:”，将②中的“] AS ShohinSum2;”变为“ShohinSum2;”。

#### 执行结果

```
shohin_bunrui|cnt_shohin
-----|-----
厨房用具 | 4
```

最内层的子查询(ShohinSum)与之前一样，根据商品种类(shohin\_bunrui)对数据进行聚合，其外层的子查询将商品数量(cnt\_shohin)

限定为 4，结果就得到了 1 行厨房用具的数据。

但是，随着子查询嵌套层数的增加，SQL 语句会变得越来越难读懂，性能也会越来越差。因此，请大家尽量避免使用多层嵌套的子查询。

## 子查询的名称

之前的例子中我们给子查询设定了“ShohinSum”等名称。原则上子查询必须设定名称，因此请大家尽量从处理内容的角度出发为子查询设定恰当的名称。在上述例子中，子查询用来对 Shohin 表的数据进行汇集，因此我们使用了后缀 Sum 作为其名称。

为子查询设定名称时需要使用 AS 关键字，该关键字有时也可以省略<sup>①</sup>。

### ①

其中也有像 Oracle 这样，在名称之前使用 AS 关键字就会发生错误的数据库，大家可以将其视为例外的情况。

### KEYWORD

- 标量子查询
- 标量

## 标量子查询

接下来我们学习子查询中的标量子查询（scalar subquery）。

### ■什么是标量

标量就是单一的意思，在数据库之外的领域也经常使用。

上一节我们学习的子查询基本上都会返回多行结果（虽然偶尔也会只返回 1 行数据）。由于结构和表相同，所以也会有查询不到结果的情况。

而标量子查询则有一个特殊的限制，那就是必须而且只能返回 1 行 1 列的结果。也就是返回表中某一行的某一列的值，例如“10”或者“东京都”这样的值。

### KEYWORD

- 返回值

返回值就是函数或者 SQL 语句等处理执行之后作为结果返回的值。



### 法则 5-7

标量子查询就是返回单一值的子查询。

细心的读者可能会发现，由于返回的是单一的值，因此标量子查询的返回值可以用在 = 或者 <> 这样需要单一值的比较运算符之中。这也正是标量子查询的优势所在。下面就让我们赶快来试试看吧。

### ■在WHERE子句中使用标量子查询

在 4-2 节中，我们练习了通过各种各样的条件从 Shohin（商品）表中读取数据。大家有没有想过通过下面这样的条件查询数据呢？

“查询出销售单价高于平均销售单价的商品。”

或者说想知道价格处于上游的商品时，也可以通过上述条件进行查询。

然而这并不是用普通方法就能解决的。如果我们像下面这样使用 AVG 函数的话，就会发生错误。

```
-- 在WHERE子句中不能使用聚合函数
SELECT shohin_id, shohinmei, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka > AVG(hanbai_tanka);
```

← “大于销售平均单价”  
这样的条件

虽然这样的 SELECT 语句看上去能够满足我们的要求，但是由于在 WHERE 子句中不能使用聚合函数，因此这样的 SELECT 语句是错误的。

那么究竟什么样的 SELECT 语句才能满足上述条件呢？

这时标量子查询就可以发挥它的功效了。首先，如果想要要求出 Shohin 表中商品的平均销售单价（hanbai\_tanka），可以使用代码清单 5-11 中的 SELECT 语句。

代码清单 5-11 计算平均销售单价的标量子查询

```
SELECT AVG(hanbai_tanka)
FROM Shohin;
```

执行结果

```
avg
-----
2097.5000000000000000
```

AVG 函数的使用方法和 COUNT 函数相同。其计算表达式如下所示。

$$(1000+500+4000+3000+6800+500+880+100) / 8=2097.5$$

这样计算出的平均单价大约就是 2100 日元。不难发现，代码清单 5-11 中的 SELECT 语句的查询结果是单一的值（2097.5）。因此，我们可以直接将这个结果用到之前失败的查询之中。正确的 SQL 如代码清单 5-12 所示。

代码清单 5-12 选出销售单价(hanbai\_tanka)高于全部商品的平均单价的商品

```
SELECT shohin_id, shohin_mei, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka > (SELECT AVG(hanbai_tanka)
FROM Shohin);
```

计算平均销售单价的标量子查询

执行结果

shohin_id	shohin_mei	hanbai_tanka
0003	运动T恤	4000
0004	菜刀	3000
0005	高压锅	6800

前一节我们已经介绍过，使用子查询的 SQL 会从子查询开始执行。因此，这种情况下也会先执行下述计算平均单价的子查询（图 5-5）。

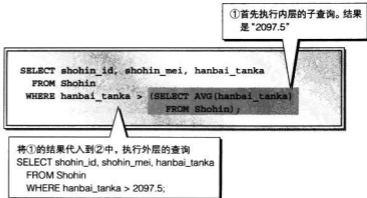
```
-- ①内层的子查询
SELECT AVG(hanbai_tanka)
FROM Shohin;
```

子查询的结果是“2097.5”，因此会用该值替换子查询的部分，生成如下 SELECT 语句。

```
-- ②外层的查询
SELECT shohin_id, shohin_mei, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka > 2097.5
```

大家都能看出该 SQL 没有任何问题可以正常执行。结果如上所述。

图 5-5 SELECT 语句的执行顺序(标量子查询)



## 标量子查询的书写位置

标量子查询的书写位置并不仅仅局限于 WHERE 子句中，通常任何可以使用单一值的位置都可以使用。也就是说，能够使用常数或者列名的地方，无论是 SELECT 子句、GROUP BY 子句、HAVING 子句，还是 ORDER BY 子句，几乎所有的地方都可以使用。

例如，在 SELECT 子句当中使用之前计算平均值的标量子查询的 SQL 语句，如代码清单 5-13 所示。

代码清单 5-13 在 SELECT 子句中使用标量子查询

```
SELECT shohin_id,
       shohin_mei,
       hanbai_tanka,
       (SELECT AVG(hanbai_tanka)
        FROM Shohin) AS avg_tanka
FROM Shohin;
```

← 标量子查询

执行结果

shohin_id	shohin_mei	hanbai_tanka	avg_tanka
0001	T恤衫	1000	2097.5000000000000000
0002	打孔器	500	2097.5000000000000000
0003	运动T恤	4000	2097.5000000000000000
0004	菜刀	3000	2097.5000000000000000
0005	高压锅	6800	2097.5000000000000000
0006	叉子	500	2097.5000000000000000
0007	擦菜板	880	2097.5000000000000000
0008	圆珠笔	100	2097.5000000000000000

从上述结果可以看出，在商品一览表中加入了全部商品的平均单价。有时我们会需要这样的单据。

此外，我们还可以像代码清单 5-14 中的 SELECT 语句那样，在 HAVING 子句中使用标量子查询。

代码清单 5-14 在 HAVING 子句中使用标量子查询

```
SELECT shohin_bunrui, AVG(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui
HAVING AVG(hanbai_tanka) > (SELECT AVG(hanbai_tanka)
                             FROM Shohin);
```

← 标量子查询



## 执行结果

shohin_bunrui	avg
衣服	2500.0000000000000000
厨房用具	2795.0000000000000000

该查询的含义是想要选取出按照商品种类计算出的销售单价高于全部商品的平均销售单价的商品种类。如果在 SELECT 语句中不使用 HAVING 子句的话,那么平均销售单价为 300 日元的办公用品也会被选取出来。但是,由于全部商品的平均销售单价是 2097.5 日元,因此低于该平均值的办公用品会被 HAVING 子句中的条件排除在外。

## 使用标量子查询时的注意事项

最后我们来介绍一下使用标量子查询时的注意事项。那就是该子查询绝对不能返回多行结果。也就是说如果子查询返回了多行结果,那么它就不再是标量子查询,而仅仅是一个普通的子查询了。因此不能被用在 = 或者 <> 等需要单一输入值的运算符当中,也不能用在 SELECT 等子句当中。

例如,如下的 SELECT 子查询会发生错误。

```
-- 由于不是标量子查询,因此不能在SELECT子句中使用
```

```
SELECT shohin_id,
       shohin_mei,
       hanbai_tanka,
       (SELECT AVG(hanbai_tanka)
        FROM Shohin
        GROUP BY shohin_bunrui) AS avg_tanka
FROM Shohin;
```

子查询

发生错误的原因很简单,就是因为会返回如下多行结果。

avg
2500.0000000000000000
300.0000000000000000
2795.0000000000000000

在 1 行 SELECT 子句之中当然不可能使用 3 行数据。因此,上述 SELECT 语句会返回“因为子查询返回了多行数据所以不能执行”这样的错误信息<sup>❶</sup>。

## ❶

例如,使用 PostgreSQL 时会返回如下错误。  
“ERROR: 副查询中使用了返回多行结果的表达式”

## 5-3

## 关联子查询

## 学习重点

- 关联子查询会在细分的组内进行比较时使用。
- 关联子查询和 GROUP BY 子句一样，也可以对表中的数据进行切分。
- 关联子查询的结合条件如果未出现在子查询之中就会发生错误。

## 普通的子查询和关联子查询的区别

按此前所学，使用子查询就能选取出销售单价 (hanbai\_tanka) 高于全部商品平均销售单价的商品。这次我们稍稍改变一下条件，选取出各商品分类中高于该分类平均销售单价的商品。

## ■ 按照商品分类与平均销售单价进行比较

只通过语言描述可能难以理解，还是让我们来看看具体示例吧。我们以厨房用具中的商品为例，该分组中包含了表 5-1 所示的 4 种商品。

表 5-1 厨房用具中的商品

商品名称	销售单价
菜刀	3000
高压锅	6800
叉子	500
擦菜板	880

因此，计算上述 4 种商品的平均价格的算术表达式如下所示。

$$(3000+6800+500+880) / 4=2795 \text{ (日元)}$$

这样我们就能得知该分组内高于平均价格的商品是菜刀和高压锅了，这两种商品就是我们要选取的对象。

我们可以对余下的分组继续使用同样的方法。衣服分组的平均销售单价是：

$(1000+4000) / 2=2500$  (日元)

因此运动 T 恤就是要选取的对象。办公用品分组的平均销售单价是：

$(500+100) / 2=300$  (日元)

因此打孔器就是我们要选取的对象。

这样大家就能明白该进行什么样的操作了吧。我们并不是要以全部商品为基础，而是要以细分的组为基础，对组内商品的平均价格和各商品的销售单价进行比较。

按照商品种类计算平均价格并不是什么难事。计算份额方法我们已经学习过了，只需按照代码清单 5-15 那样，使用 GROUP BY 子句就可以了。

代码清单 5-15 按照商品分类计算平均价格

```
SELECT AVG(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

但是，如果我们使用前一节（标量子查询）的方法，直接把上述 SELECT 语句使用到 WHERE 子句当中的话，就会发生错误。

```
-- 发生错误的子查询
SELECT shohin_id, shohinmei, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka > (SELECT AVG(hanbai_tanka)
                      FROM Shohin
                      GROUP BY shohin_bunrui);
```

出错原因前一节已经讲过了，该子查询会返回 3 行结果（2795、2500、300），并不是标量子查询。在 WHERE 子句中使用子查询时，该子查询的结果必须是单一的。

但是，如果以商品种类分组为单位，对销售单价和平均单价进行比较，除此之外似乎也没有什么办法了。到底应该怎么办才好呢？

#### ■使用关联子查询的解决方案

这时就轮到我们的好帮手——关联子查询登场了。

只需要在刚才的 SELECT 语句中追加一行，就能得到我们想要的结果了。事实胜于雄辩，还是让我们先来看看修改之后的 SELECT 语句吧（代

#### KEYWORD

● 关联子查询

码清单 5-16)。

#### 代码清单 5-16 通过关联子查询按照商品种类对平均销售单价进行比较

```

SQL Server DB2 PostgreSQL MySQL
SELECT shohin_id, shohin_mei, hanbai_tanka
FROM Shohin AS S1
WHERE hanbai_tanka > (SELECT AVG(hanbai_tanka)
FROM Shohin AS S2
WHERE S1.shohin_bunrui = S2.shohin_bunrui
GROUP BY shohin_bunrui);

```

该条件就是成功的关键!

#### 特定的SQL

Oracle中不能使用AS(会发生错误)。因此,在Oracle中执行代码清单5-16时,请大家把①中的FROM Shohin AS S1变为FROM Shohin S1,把②中的FROM Shohin AS S2变为FROM Shohin S2。

#### 执行结果

shohin_id	shohin_mei	hanbai_tanka
办公用品	打孔器	500
衣服	运动T恤	4000
厨房用具	菜刀	3000
厨房用具	高压锅	6800

这样我们就能选取出办公用品、衣服和厨房用具三类商品中高于该类商品平均销售单价的商品了。

这里起到关键作用的就是在子查询中添加的WHERE子句的条件。该条件的意思就是,在同一商品种类中对各商品的销售单价和平均单价进行比较。

这次由于作为比较对象的都是同一张Shohin表,因此为了进行区别,分别使用了S1和S2两个别名。在使用关联子查询时,需要在表所对应的列名之前加上表的别名,以“<表名>.<列名>”的形式记述。

在对表中某一部分记录的集合进行比较时,就可以使用关联子查询。因此,使用关联子查询时,通常会使用“限定(绑定)”或者“限制”这样的语言。例如本次示例就是对“限定商品种类”的平均单价进行比较。



#### 法则 5-8

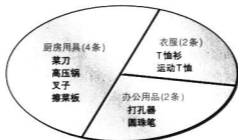
在细分的组内进行比较时,需要使用关联子查询。

## 关联子查询也是用来对集合进行切分的

换个角度来看，其实关联子查询也和 GROUP BY 子句一样，可以对集合进行进行切分。

大家还记得我们用来说明 GROUP BY 子句的图（图 5-6）吗？

图 5-6 根据商品种类对表进行切分的图示



上图显示了作为记录集合的表是如何按照商品种类被切分的。使用关联子查询进行切分的图示也基本相同（图 5-7）。

图 5-7 根据关联子查询进行切分的图示



我们首先需要计算各个商品分类中商品的平均销售单价，由于该单价会用来和商品表中的各条记录进行比较，因此关联子查询实际只能返回 1 行结果。这也是关联子查询不出错的关键。关联子查询执行时，DBMS 内部的执行结果图示请参见图 5-8。

图5-8 关联子查询执行时DBMS的内部动作图

SELECT 衣服,	T恤衫,	1000	FROM Shohin	WHERE 1000 > 2500;
SELECT 衣服,	运动T恤,	4000	FROM Shohin	WHERE 4000 > 2500;
SELECT 厨房用具,	菜刀,	3000	FROM Shohin	WHERE 3000 > 2795;
SELECT 厨房用具,	高压锅,	6800	FROM Shohin	WHERE 6800 > 2795;
SELECT 厨房用具,	叉子,	500	FROM Shohin	WHERE 500 > 2795;
SELECT 厨房用具,	擦菜板,	880	FROM Shohin	WHERE 880 > 2795;
SELECT 办公用品,	圆珠笔,	100	FROM Shohin	WHERE 100 > 300;
SELECT 办公用品,	打孔器,	500	FROM Shohin	WHERE 500 > 300;

满足条件

如果商品种类发生了变化,那么用来进行比较的平均单价也会发生变化。这样就可以将各种商品的销售单价和平均单价进行比较了。关联子查询的内部执行结果对于初学者来说是比较难以理解的,但是像上图这样将其内部执行结果可视化之后,理解起来就变得非常容易了吧。

## 结合条件一定要写在子查询中

下面给大家介绍一下 SQL 初学者在使用关联子查询时经常犯的一个错误,那就是将关联条件写在子查询之外的外层查询之中。请大家看一下下面这条 SELECT 语句。

```
-- 错误的关联子查询书写方法
SELECT shohin_bunrui, shohin_mei, hanbai_tanka
FROM Shohin AS S1
WHERE S1.shohin_bunrui = S2.shohin_bunrui
AND hanbai_tanka > (SELECT AVG(hanbai_tanka)
FROM Shohin AS S2
GROUP BY shohin_bunrui);
```

将关联条件移到子查询之外

上述 SELECT 语句只是将子查询中的关联条件移到了外层查询之中,其他并未加任何更改。但是,该 SELECT 语句会发生错误,不能正确执行。允许存在这样的书写方法可能并不奇怪,但是 SQL 的规则禁止这样的书写方法。

该书写方法究竟违反了什么规则呢?那就是关联名称的作用域。虽然这一术语有些晦涩难懂,但是一解释大家就明白了。关联名称就是像 S1、S2 这样作为表别名的名称,作用域(scope)就是生存作用域(有

### KEYWORD

- 关联名称
- 作用域

效作用域)。也就是说，关联名称存在一个有效作用域的限制。

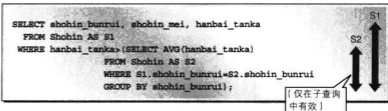
具体来讲，子查询内部设定的关联名称，只能在该子查询内部使用（图 5-9）。换句话说，就是“内部可以看到外部，而外部看不到内部”。

请大家一定不要忘记关联名称具有一定的有效作用域。如前所述，SQL 是按照先内层子查询后外层查询的顺序来执行的。这样，子查询执行结束时只会留下执行结果，作为抽出源的 S2 表其实已经不存在了<sup>❶</sup>。因此，在执行外层查询时，由于 S2 表已经不存在了，就会返回“不存在使用该名称的表”这样的错误。

❶

当然，消失的其实只是 S2 这个名称而已，Shohin 表以及其中的数据还是存在的。

图 5-9 子查询内的关联名称的有效作用域



## 练习题

5.1 创建出满足下述三个条件的视图（视图名称为 ViewRenshu5\_1）。使用 Shohin（商品）表作为参照表，假设表中包含初始状态的 8 行数据。

条件 1：销售单价大于等于 1000 日元。

条件 2：登记日期是 2009 年 9 月 20 日。

条件 3：包含商品名称、销售单价和登记日期三列。

对该视图执行 SELECT 语句的结果如下所示。

```
SELECT * FROM ViewRenshu5_1;
```

执行结果

shohin_mei	hanbai_tanka	torokubi
T恤衫	1000	2009-09-20
菜刀	3000	2009-09-20

5.2 向习题 5.1 中创建的视图 ViewRenshu5\_1 中插入如下数据, 会得到什么样的结果呢?

```
INSERT INTO ViewRenshu5_1 VALUES ('刀子', 300, '2009-11-02');
```

提示: 使用 PostgreSQL 时, 由于视图会被初始设定为只读, 所以在执行 INSERT 语句之前一定要将视图设置为可以更新。

5.3 请根据如下结果编写 SELECT 语句。其中 hanbai\_tanka\_all 列为全部商品的平均销售单价。

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	hanbai_tanka_all
0001	T恤衫	衣服	1000	2097.5000000000000000
0002	打孔器	办公用品	500	2097.5000000000000000
0003	运动T恤	衣服	4000	2097.5000000000000000
0004	菜刀	厨房用具	3000	2097.5000000000000000
0005	高压锅	厨房用具	6800	2097.5000000000000000
0006	叉子	厨房用具	500	2097.5000000000000000
0007	磨菜板	厨房用具	880	2097.5000000000000000
0008	圆珠笔	办公用品	100	2097.5000000000000000

5.4 请根据习题 5.1 中的条件编写一条 SQL 语句, 创建一幅包含如下数据的视图 (名称为 AvgTankaByBunrui)。

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	avg_hanbai_tanka
0001	T恤衫	衣服	1000	2500.0000000000000000
0002	打孔器	办公用品	500	300.0000000000000000
0003	运动T恤	衣服	4000	2500.0000000000000000
0004	菜刀	厨房用具	3000	2795.0000000000000000
0005	高压锅	厨房用具	6800	2795.0000000000000000
0006	叉子	厨房用具	500	2795.0000000000000000
0007	磨菜板	厨房用具	880	2795.0000000000000000
0008	圆珠笔	办公用品	100	300.0000000000000000

提示: 其中的关键是 avg\_hanbai\_tanka 列。与习题 5.3 不同, 这里需要计算出的各商品种类的平均销售单价。这与 5-3 节中使用关联子查询所得到的结果相同。也就是说, 该列可以使用关联子查询进行创建。问题就是应该在什么地方使用这个关联子查询。



# 第6章 函数、谓词、CASE表达式

各种各样的函数

谓词

CASE表达式



SQL

## 本章重点

---

不仅 SQL，对所有的编程语言来说，函数都起着至关重要的作用。函数就像是编程语言的“工具箱”，每种编程语言都准备了非常多的函数。使用函数，我们可以实现计算、字符串操作，日期计算等各种各样的运算功能。

本章将会和大家一起学习具有代表性的函数以及特殊版本的函数（和 CASE 表达式）的使用方法。

### 6-1 各种各样的函数

- 函数的种类
- 算术函数
- 字符串函数
- 日期函数
- 转换函数

### 6-2 谓词

- 什么是谓词
- LIKE 谓词——字符串的部分一致查询
- BETWEEN 谓词——范围查询
- IS NULL、IS NOT NULL——判断是否为 NULL
- IN 谓词——OR 的简使用法
- 使用子查询作为 IN 谓词的参数
- EXIST 谓词

### 6-3 CASE 表达式

- 什么是 CASE 表达式
- CASE 表达式的语法
- CASE 表达式的使用方法

## 6-1

## 各种各样的函数

## 学习重点

- 根据用途，函数可以大致分为算术函数、字符串函数、日期函数、转换函数和聚合函数。
- 函数的种类很多，无需全都记住。只需要记住具有代表性的函数就可以了，其他的可以在使用时再进行查询。

## KEYWORD

- 函数
- 参数 (parameter)
- 返回值

## KEYWORD

- 算术函数
- 字符串函数
- 日期函数
- 转换函数
- 聚合函数

## 函数的种类

前几章和大家一起学习了 SQL 的语法结构等必须要遵守的规则。本章将会进行一点改变，来学习一些 SQL 自带的便利工具——函数。

我们在 3-1 节中已经学习了函数的概念，这里再回顾一下。所谓函数，就是输入某一值得到相应输出结果的功能。输入值称为参数 (parameter)，输出值称为返回值。

函数大致可以分为以下几种。

- 算术函数 (用来进行数值计算的函数)
- 字符串函数 (用来进行字符串操作的函数)
- 日期函数 (用来进行日期操作的函数)
- 转换函数 (用来转换数据类型和值的函数)
- 聚合函数 (用来进行数据聚合的函数)

我们已经在第 3 章中学习了聚合函数的相关内容，大家应该对函数有初步的了解了吧。聚合函数基本上只包含 COUNT、SUM、AVG、MAX、MIN 这 5 种，而其他种类的函数总数可以轻松超过 200 种。可能大家会觉得怎么会有那么多函数啊，但其实并不需要担心，虽然数量众多，但常用函数只有 30-50 个。不熟悉的函数大家可以通过参考文档 (词典) 来进行查询<sup>①</sup>。

本节将会和大家一起学习一些具有代表性的函数。大家并不需要一次全部记住，只需要知道有这样的函数就可以了。实际应用时可以通过参考

## ①

参考文献是 DBMS 手册的一部分。大家也可以从包含函数汇总的书籍以及 Web 网站上获取相关信息。

文档进行查询。

接下来，将会按照英文字母的顺序，分类向大家介绍这些函数。

## 算术函数

算术函数是最基本的函数，其实之前我们已经学习过了，可能有些读者已经想起来了。没错，就是 2-2 节介绍的加减乘除四则运算。

### KEYWORD

- 算术函数

### KEYWORD

- + 运算符
- - 运算符
- \* 运算符
- / 运算符

- +( 加法)
- -( 减法)
- \*( 乘法)
- /( 除法)

由于这些算术运算符具有“根据输入值返回相应输出结果”的功能，所以它们是出色的算术函数。在此我们将会给大家介绍除此之外的具有代表性的函数。

为了学习算术函数，我们首先根据代码清单 6-1 创建一张示例用表 (SampleMath)。

数据类型 NUMERIC 是大多数 DBMS 都支持的一种数据类型，通过 NUMERIC (全体位数，小数位数) 的形式来指定数值的大小。接下来，将会给大家介绍常用的算术函数——ROUND 函数，由于 PostgreSQL 中的 ROUND 函数只能使用 NUMERIC 类型的数据，因此我们在示例中也使用了该数据类型。

### 代码清单 6-1 创建 SampleMath 表

```
-- DDL: 创建表
CREATE TABLE SampleMath
(m NUMERIC (10,3),
 n INTEGER,
 p INTEGER);

-- DML: 插入数据
BEGIN TRANSACTION;

INSERT INTO SampleMath(m, n, p) VALUES (500, 0, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (-180, 0, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (NULL, NULL, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (NULL, 7, -3);
```

```

INSERT INTO SampleMath(m, n, p) VALUES (NULL, 5, 2);
INSERT INTO SampleMath(m, n, p) VALUES (NULL, 4, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (8, NULL, 3);
INSERT INTO SampleMath(m, n, p) VALUES (2.27, 1, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (5.555, 2, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (NULL, 1, NULL);
INSERT INTO SampleMath(m, n, p) VALUES (8.76, NULL, NULL);

COMMIT;

```

#### 特定的SQL

不同的DBMS事务处理的语法也不尽相同。代码清单6-1中的DML语句在MySQL中执行时，需要将①部分更改为“START TRANSACTION;”。在Oracle和DB2中执行时，无需用到①的部分（请删除）。

详细内容请大家参考第4章中的“创建事务”。

下面让我们来确认一下创建好的表中的内容。其中应该包含了m、n、p三列。

```
SELECT * FROM SampleMath;
```

执行结果

m	n	p
500.000	0	
-180.000	0	
	7	3
	5	2
	4	
8.000		3
2.270	1	
5.555	2	
	1	
8.760		

## ■ ABS——绝对值

### 语法6-1 ABS函数

```
ABS(数值)
```

ABS是计算绝对值的函数。绝对值（absolute value）就是不考虑数值的符号，表示一个数到原点距离的数值。简单来讲，绝对值的计算方法就是：0和正数的绝对值就是其本身，负数的绝对值就是去掉符号后的结果。

#### KEYWORD

- ABS函数
- 绝对值

代码清单6-2 计算数值的绝对值

```
SELECT m,
       ABS(m) AS abs_col
FROM SampleMath;
```

执行结果

m	abs_col
500.000	500.000
-180.000	180.000
8.000	8.000
2.270	2.270
5.555	5.555
8.760	8.760

右侧的 abs\_col 列就是通过 ABS 函数计算出的 m 列的绝对值。请大家注意，-180 的绝对值就是去掉符号后的结果 180。

通过上述结果我们可以发现，ABS 函数的参数为 NULL 时，结果也是 NULL。并非只有 ABS 函数如此，其实绝大多数函数对于 NULL 的结果都是 NULL。

## 0

但是转换函数中的 COALESCE 函数除外。

## ■ MOD——求余

语法6-2 MOD函数

```
MOD(被除数, 除数)
```

MOD 是计算除法余数（剩余）的函数，是 modulo 的简称。例如，7/3 的余数是 1，因此 MOD(7, 3) 的结果也是 1（代码清单 6-3）。由于小数计算中并没有余数的概念，所以只能对整数类型的列使用 MOD 函数。

代码清单6-3 计算除法(n÷p)的余数

```
Oracle DB2 PostgreSQL MySQL
SELECT n, p,
       MOD(n, p) AS mod_col
FROM SampleMath;
```

## KEYWORD

## ● MOD 函数

## 执行结果

n	p	mod_col
0		
0		
7	3	1
5	2	1
4		
	3	
1		
2		
1		

mod\_col = MOD(n, p) 的返回值 (n ÷ p 的余数)

这里有一点需要大家注意。主流的 DBMS 都支持 MOD 函数，只有 SQL Server 不支持该函数。

**KEYWORD**

● % 运算符 (SQL Server)

**特定的 SQL**

SQL Server 使用特殊的运算符 (函数) “%” 来计算余数。使用如下的专用语法可以得到与代码清单 6-3 相同的结果。需要使用 SQL Server 的读者需要特别注意。

**SQL Server**

```
SELECT n, p,
       n % p AS mod_col
FROM SampleMath;
```

**ROUND——四舍五入**

## 语法 6-3 ROUND 函数

ROUND (对象数值, 保留小数的位数)

**KEYWORD**

● ROUND 函数

ROUND 函数用来进行四舍五入操作。四舍五入在英语中称为 round。如果指定四舍五入的位数为 1，那么就会对小数点第 2 位进行四舍五入处理。如果指定位数为 2，那么就会对第 3 位进行四舍五入处理 (代码清单 6-4)。

代码清单 6-4 对 m 列的数值进行 n 列位数的四舍五入处理

```
SELECT m, n,
       ROUND(m, n) AS round_col
FROM SampleMath;
```

## 执行结果

m	n	round_col
500.000	0	500
-180.000	0	-180
	7	
	5	
	4	
8.000		
2.270	1	2.3
5.555	2	5.56
	1	
8.760		

**说明**

m: 对象数值  
n: 指定舍入位数  
round\_col: ROUND(m, n) 的返回值 (包含舍入的结果)

## 字符串函数

截至目前，我们介绍的函数都是主要针对数值的算术函数。但其实算术函数只是 SQL（其他编程语言通常也是如此）自带的函数中的一部分。虽然算术函数是我们经常使用的函数，但是字符串函数也经常被使用。

在日常生活中，我们经常会像使用数字那样，对字符串进行替换、截取、简化等操作。因此 SQL 也为我们提供了很多操作字符串的功能。

为了学习字符串函数，我们再来创建一张表（SampleStr），参见代码清单 6-5。

### 代码清单 6-5 创建 SampleStr 表

```
-- DDL: 创建表
CREATE TABLE SampleStr
(str1 VARCHAR(40),
 str2 VARCHAR(40),
 str3 VARCHAR(40));
```

**SQL Server | PostgreSQL**

```
-- DML: 插入数据
BEGIN TRANSACTION;

INSERT INTO SampleStr (str1, str2, str3) VALUES ('cpqx' , =>
'rt', NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('abc' , =>
'def' , NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('山田' , =>
```

#### KEYWORD

##### ● 字符串函数



```

'太郎', '是我');
INSERT INTO SampleStr (str1, str2, str3) VALUES ('aaa' ,⇒
NULL ,NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES (NULL ,⇒
'xyz',NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('@!#$%' ,⇒
NULL ,NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('ABC' ,⇒
NULL ,NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('abc' ,⇒
NULL ,NULL);
INSERT INTO SampleStr (str1, str2, str3) VALUES ('abc太郎', ⇒
'abc' , 'ABC');
INSERT INTO SampleStr (str1, str2, str3) VALUES ('abcdefabc' , ⇒
'abc' , 'ABC');
INSERT INTO SampleStr (str1, str2, str3) VALUES ('micmic' , ⇒
'i' , 'I');

COMMIT;

```

⇒表示下一行接续本行, 只是由于版面所限而换行。

#### 特定的SQL

不同的DBMS事务处理的语法也不尽相同。代码清单6-5中的DML语句在MySQL中执行时, 需要将①部分更改为“START TRANSACTION;”。在Oracle和DB2中执行时, 无需用到①的部分(请删除)。

详细内容请大家参考第4章中的“创建事务”。

下面让我们来确认一下创建好的表中的内容。其中应该包含了str1、str2、str3三列。

```
SELECT * FROM SampleStr;
```

#### 执行结果

str1	str2	str3
opx	rt	
abc	def	
山田	太郎	是我
aaa	xyz	
@!#\$%		
ABC		
aBC		
abc太郎	abc	ABC
abcdefabc	abc	ABC
micmic	i	I

## ■ ||——拼接

## 语法 6-4 || 函数

```
字符串1||字符串2
```

在实际业务中，我们经常会碰到  $abc + de = abcde$  这样希望将字符串进行拼接的情况。在 SQL 中，可以通过由两条并列的竖线变换而成的“||”函数来实现（代码清单 6-6）。

## 代码清单 6-6 拼接两个字符串(str1+str2)

```
Oracle      DB2      PostgreSQL
SELECT str1, str2,
       str1 || str2 AS str_concat
FROM SampleStr;
```

## 执行结果

str1	str2	str_concat
opx	rt	opxrt
abc	def	abcdef
山田	太郎	山田太郎
aaa		
	xyz	
@!#\$%		
ABC		
abc		
abc太郎	abc	abc太郎abc
abcdefabc	abc	abcdefabcabc
micmic	i	micmaci

SQL  
str\_concat(str1||str2) 函数  
(拼接结果)

进行字符串拼接时，如果其中包含 NULL，那么得到的结果也是 NULL。这是因为“||”也是变了形的函数。当然，三个以上的字符串也可以进行拼接（代码清单 6-7）。

## 代码清单 6-7 拼接三个字符串(str1+str2+str3)

```
Oracle      DB2      PostgreSQL
SELECT str1, str2, str3,
       str1 || str2 || str3 AS str_concat
FROM SampleStr
WHERE str1 = '山田';
```

## 执行结果

str1	str2	str3	str_concat
山田	太郎	是我	山田太郎是我

SQL  
str\_concat(str1||str2||str3) 函数  
(拼接结果)

## KEYWORD

- || 函数

这里也有一点需要大家注意，|| 函数在 SQL Server 和 MySQL 中无法使用。

#### KEYWORD

- + 运算符 (SQL Server)
- CONCAT 函数 (MySQL)

#### ①

由于这和Java中连接字符串的方法相同，估计有些读者已经比较熟悉了。

#### 特定的SQL

SQL Server使用“+”运算符(函数)来连接字符串<sup>①</sup>。MySQL使用CONCAT函数来完成字符串的拼接。使用如下SQL Server/MySQL的专用语法能够得到与代码清单6-7相同的结果。

#### SQL Server

```
SELECT str1, str2, str3,
       str1 + str2 + str3 AS str_concat
FROM SampleStr;
```

#### MySQL

```
SELECT str1, str2, str3,
       CONCAT(str1, str2, str3) AS str_concat
FROM SampleStr;
```

## LENGTH——字符串长度

### 语法6-5 LENGTH函数

```
LENGTH(字符串)
```

想要知道字符串中包含多少个字符时，可以使用LENGTH(长度)函数(代码清单6-8)。

### 代码清单6-8 计算字符串长度

```
Oracle DB2 PostgreSQL MySQL
SELECT str1,
       LENGTH(str1) AS len_str
FROM SampleStr;
```

### 执行结果

str1	len_str
opx	3
abc	3
山田	2
aaa	3
@!#\$%	5
ABC	3
aBC	3
abc太郎	5
abcdefabc	9
micmic	6

```
len_str LENGTH(str1)
micmic 6
```

#### KEYWORD

- LENGTH 函数

需要注意的是，该函数也无法在 SQL Server 中使用。

## KEYWORD

- LEN 函数 (SQL Server)

### 特定的 SQL

SQL Server 使用 LEN 函数来计算字符串的长度。使用如下 SQL Server 的专用语法能够得到与代码清单 6-8 相同的结果。

#### SQL Server

```
SELECT str1,
       LEN(str1) AS len_str
FROM SampleStr;
```

我想大家应该逐渐明白“SQL 中有很多特定的用法”这句话的含义了吧。

## KEYWORD

- 字节
- 多字节字符

字节 (byte) 是计算机中用来表述数据大小的基本单位。如本书所述，通常情况下“1 字符 = 1 字节”。单位字节 (KB) 是字节的 1024 倍，单位兆字节 (MB) 是千字节的 1024 倍，单位千兆字节 (GB) 是兆字节的 1024 倍。表示硬盘容量时经常会使用的“100GB”、“250GB”，其中 100GB 指的是可以存储  $1024 \times 1024 \times 1024 \times 100 = 107,374,182,400$  个半角英文字母。

## 0

MySQL 中还存在计算字符串长度的自有函数 CHAR\_LENGTH。

## KEYWORD

- LENGTH 函数 (MySQL)
- CHAR\_LENGTH 函数 (MySQL)

## KEYWORD

- LOWER 函数

## 专栏

### 对 1 个字符使用 LENGTH 函数有可能得到 2 字节以上的结果

LENGTH 函数中，还有一点需要大家特别注意的地方，那就是该函数究竟以什么为单位来计算字符串的长度。这部分是初级以上阶段才会学习到的内容，在此先简单介绍一下。

可能有些读者已经有所了解，与半角英文字母占用 1 字节不同，日语汉字这样的全角字符会占用 2 个以上的字节（称为多字节字符）。因此，使用 MySQL 中的 LENGTH 这样以字节为单位的函数进行计算时，“LENGTH(山田)”的返回结果是 4。同样是 LENGTH 函数，不同 DBMS 的执行结果也不尽相同。

虽然有些混乱，但这正是我希望大家能够牢记的。

## ■ LOWER——小写转换

### 语法 6-6 LOWER 函数

LOWER (字符串)

LOWER 函数只能针对英文字母使用，它会将参数中的字符串全都转换为小写（代码清单 6-9）。因此，该函数并不适用于英文字母以外的场合。此外，该函数并不影响原本就是小写的字符。

#### 代码清单 6-9 大写转换为小写

```
SELECT str1,
       LOWER(str1) AS low_str
```

```
FROM SampleStr
WHERE str1 IN ('ABC', 'abc', '山田');
```

#### 执行结果

```
str1 | low_str
-----+-----
abc  | abc
山田 | 山田
ABC  | abc
aBC  | abc
```

low\_str: LOWER(str1)的结果值

既然存在小写转换函数，那么肯定也有大写转换函数。UPPER就是大写转换函数。

### ■ REPLACE——字符串的替换

#### 语法6-7 REPLACE函数

```
REPLACE(对象字符串, 替换前的字符串, 替换后的字符串)
```

#### KEYWORD

##### ● REPLACE函数

使用 REPLACE 函数，可以将字符串的一部分替换为其他的字符串（代码清单 6-10）。

#### 代码清单 6-10 替换字符串的一部分

```
SELECT str1, str2, str3,
       REPLACE(str1, str2, str3) AS rep_str
FROM SampleStr;
```

#### 执行结果

```
str1 | str2 | str3 | rep_str
-----+-----+-----+-----
opx  | rt   |      |
abc  | def  |      |
山田 | 太郎 | 是我 | 山田
aaa  | xyz  |      |
@!#$ |      |      |
ABC  |      |      |
aBC  |      |      |
abc太郎 | abc | ABC | ABC太郎
abcdefabc | abc | ABC | ABCdefABC
micmic | i   | I   | micmic
```

str1 对象字符串  
str2 替换前的字符串  
str3 替换后的字符串  
rep\_str REPLACE(str1, str2, str3)的结果值

**KEYWORD**

● SUBSTRING 函数

**0**

需要大家注意的是，该函数也存在和LENGTH函数同样的多字节字符的问题。详细内容请大家参考专栏“对1个字符使用LENGTH函数有可能得到2字节以上的结果”。

**■ SUBSTRING——字符串的截取**

语法6-8a SUBSTRING函数(PostgreSQL/MySQL专用语法)

SUBSTRING(对象字符串 FROM 截取的起始位置 FOR 截取的字符数)

使用SUBSTRING函数可以截取出字符串中的一部分字符串(代码清单6-11)。截取的起始位置从字符串最左侧开始计算<sup>①</sup>。

代码清单6-11 截取出字符串中第3位和第4位的字符

```
PostgreSQL MySQL
SELECT str1,
       SUBSTRING(str1 FROM 3 FOR 2) AS sub_str
FROM SampleStr;
```

执行结果

str1	sub_str
opx	x
abc	bc
山田	田
aaa	aa
@!#\$%	!#\$%
ABC	BC
aBC	BC
abc太郎	c太
abcdefabc	cd
micmic	cm

sub\_str SUBSTRING(str1 FROM 3 FOR 2)的结果

虽然上述SUBSTRING函数的语法是标准SQL承认的正式语法，但是现在只有PostgreSQL和MySQL支持该语法。

**特定的SQL**

SQL Server将语法6-8a中的内容进行了简化(语法6-8b)。

语法6-8b SUBSTRING函数(SQL Server专用语法)

SUBSTRING(对象字符串, 截取的起始位置, 截取的字符数)

Oracle和DB2将该语法进一步简化, 得到了如下结果。

语法6-8c SUBSTR函数(Oracle/DB2专用语法)

SUBSTR(对象字符串, 截取的起始位置, 截取的字符数)

SQL 有这么多特定的语法，真是有些人头疼啊。能够得到与代码清单 6-11 相同结果的各 DBMS 专用语法如下所示。

```
SQL Server
SELECT str1,
       SUBSTRING(str1, 3, 2) AS sub_str
FROM SampleStr;
```

```
Oracle DB2
SELECT str1,
       SUBSTR(str1, 3, 2) AS sub_str
FROM SampleStr;
```

## ■ UPPER——大写转换

### 语法 6-9 UPPER 函数

```
UPPER(字符串)
```

UPPER 函数只能针对英文字母使用，它会将参数中的字符串全都转换为大写（代码清单 6-12）。因此，该函数并不适用于英文字母以外的情况。此外，该函数并不影响原本就是大写的字符。

### 代码清单 6-12 将小写转换为大写

```
SELECT str1,
       UPPER(str1) AS up_str
FROM SampleStr
WHERE str1 IN ('ABC', 'aBC', 'abc', '山田');
```

### 执行结果

```
str1 | up_str
-----
abc  | ABC
山田 | 山田
ABC  | ABC
aBC  | ABC
```

与之相对，进行小写转换的是 LOWER 函数。

## KEYWORD

### ● UPPER 函数

**KEYWORD**

● 日期函数

**0**

如果想要了解日期函数的详细内容，目前只能通过各个DBMS的手册进行查询。

**KEYWORD**

● CURRENT\_DATE 函数

## 日期函数

虽然 SQL 中有很多日期函数，但是其中大部分都依存于各自的 DBMS，因此无法进行统一的说明<sup>❶</sup>。本节将会介绍那些被标准 SQL 承认，可以应用于绝大多数 DBMS 的函数。

### ■ CURRENT\_DATE——当前日期

语法 6-10 CURRENT\_DATE 函数

CURRENT\_DATE

CURRENT\_DATE 函数能够返回 SQL 执行的日期，也就是该函数执行时的日期。由于没有参数，所以无需使用括号。

执行日期不同，CURRENT\_DATE 函数的返回值也不同。如果在 2009 年 12 月 13 日执行该函数，会得到返回值“2009-12-13”。如果在 2010 年 1 月 1 日执行，就会得到返回值“2010-01-01”（代码清单 6-13）。

代码清单 6-13 获得当前日期

```
PostgreSQL MySQL
SELECT CURRENT_DATE;
```

执行结果

```
date
-----
2010-05-25
```

该函数无法在 SQL Server 中执行。此外，Oracle 和 DB2 中的语法略有不同。

**特定的 SQL**

SQL Server 使用如下的 CURRENT\_TIMESTAMP（后述）函数来获得当前日期。

```
SQL Server
-- 使用CAST（后述）函数将CURRENT_TIMESTAMP转换为日期类型
SELECT CAST(CURRENT_TIMESTAMP AS DATE) AS CUR_DATE;
```



## 执行结果

```

CUR_DATE
-----
2010-05-25

```

在 Oracle 中使用该函数时，需要在 FROM 子句中指定临时表（DUAL）。而在 DB2 中使用时，需要在 CURRENT 和 DATE 之间添加半角空格，并且还需要指定临时表 SYSIBM.SYSDUMMY1（相当于 Oracle 中的 DUAL）。这些容易混淆的地方请大家多加注意。

```

Oracle
SELECT CURRENT_DATE
FROM dual;

DB2
SELECT CURRENT DATE
FROM SYSIBM.SYSDUMMY1;

```

## ■ CURRENT\_TIME——当前时间

## 语法 6-11 CURRENT\_TIME 函数

```
CURRENT_TIME
```

## KEYWORD

## ● CURRENT\_TIME 函数

CURRENT\_TIME 函数能够取得 SQL 执行的时间，也就是该函数执行时的时间（代码清单 6-14）。由于该函数也没有参数，所以同样无需使用括号。

## 代码清单 6-14 取得当前时间

```

PostgreSQL MySQL
SELECT CURRENT_TIME;

```

## 执行结果

```

timetz
-----
17:26:50.995+09

```

该函数同样无法在 SQL Server 中执行，在 Oracle 和 DB2 中的语法同样略有不同。

**特定的SQL**

SQL Server使用如下的CURRENT\_TIMESTAMP函数（后述）来获得当前日期。

```
-- 使用CAST函数（后述）将CURRENT_TIMESTAMP转换为时间类型
SELECT CAST(CURRENT_TIMESTAMP AS TIME) AS CUR_TIME;
```

**执行结果**

```
CUR_TIME
-----
21:33:59.3400000
```

在Oracle和DB2中使用时的语法如下所示。需要注意的地方和CURRENT\_DATE函数相同。在Oracle中使用时所得到的结果还包含日期。

```
Oracle
-- 指定临时表（DUAL）
SELECT CURRENT_TIMESTAMP
FROM dual;
```

```
DB2
/* CURRENT和TIME之间使用了半角空格，指定临时表SYSIBM.SYSDUMMY1 */
SELECT CURRENT TIME
FROM SYSIBM.SYSDUMMY1;
```

**■ CURRENT\_TIMESTAMP——当前日期和时间****语法6-12 CURRENT\_TIMESTAMP函数**

```
CURRENT_TIMESTAMP
```

**KEYWORD**

- CURRENT\_TIMESTAMP  
函数

CURRENT\_TIMESTAMP函数同时具有CURRENT\_DATE + CURRENT\_TIME的功能。使用该函数可以同时得到当前的日期和时间，当然也可以从结果中截取日期或者时间。

**代码清单6-15 取得当前日期和时间**

```
SQL Server PostgreSQL MySQL
SELECT CURRENT_TIMESTAMP;
```

**执行结果**

```
NOW
-----
2010-04-25 18:31:03.704+09
```

## ❶

之前我们已经介绍过，在 SQL Server 中无法使用 CURRENT\_DATE 和 CURRENT\_TIME 函数。可能是因为 SQL Server 中，CURRENT\_TIMESTAMP 已经涵盖了这两者功能的原因吧。

该函数可以在 SQL Server 等各个主要 DBMS 中使用<sup>❶</sup>。但是，与之前的 CURRENT\_DATE 和 CURRENT\_TIME 一样，在 Oracle 和 DB2 中该函数的语法略有不同。

## 特定的 SQL

Oracle 和 DB2 使用如下写法可以得到与代码清单 6-15 相同的结果。其中的注意点与 CURRENT\_DATE 时完全相同。

```

Oracle
-- 指定临时表 (DUAL)
SELECT CURRENT_TIMESTAMP
FROM dual;

DB2
/* CURRENT和TIME之间使用了半角空格，指定临时表SYSIBM.SYSDUMMY1 */
SELECT CURRENT TIME STAMP
FROM SYSIBM.SYSDUMMY1;

```

## ■ EXTRACT——截取日期元素

语法 6-13 EXTRACT 函数

```
EXTRACT(日期元素 FROM 日期)
```

使用 EXTRACT 函数可以截取日期数据中的一部分，例如“年”、“月”，或者“小时”、“秒”等等（代码清单 6-16）。该函数的返回值并不是日期类型而是数值类型。

代码清单 6-16 截取日期元素

```

PostgreSQL MySQL
SELECT CURRENT_TIMESTAMP,
       EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS year,
       EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS month,
       EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS day,
       EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS hour,
       EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS minute,
       EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second;

```

## 执行结果

now	year	month	day	hour	minute	second
2010-04-25 19:07:33.987+09	2010	5	25	19	7	33.987

## KEYWORD

● EXTRACT 函数

需要注意的是 SQL Server 也无法使用该函数。

## KEYWORD

- DATEPART 函数  
(SQL Server)

### 特定的 SQL

SQL Server 使用如下的 DATEPART 函数会得到与 6-16 相同的结果。

#### SQL Server

```
SELECT CURRENT_TIMESTAMP,
       DATEPART(YEAR, CURRENT_TIMESTAMP) AS year,
       DATEPART(MONTH, CURRENT_TIMESTAMP) AS month,
       DATEPART(DAY, CURRENT_TIMESTAMP) AS day,
       DATEPART(HOUR, CURRENT_TIMESTAMP) AS hour,
       DATEPART(MINUTE, CURRENT_TIMESTAMP) AS minute,
       DATEPART(SECOND, CURRENT_TIMESTAMP) AS second;
```

Oracle 和 DB2 想要得到相同结果的话，需要进行如下改变。注意事项与 CURRENT\_DATE 时完全相同。

#### Oracle

```
-- 在FROM子句中指定临时表 (DUAL)
SELECT CURRENT_TIMESTAMP,
       EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS year,
       EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS month,
       EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS day,
       EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS hour,
       EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS minute,
       EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second
FROM DUAL;
```

#### DB2

```
/* CURRENT和TIME之间使用了半角空格，指定临时表SYSIBM.SYSDUMMY1 */
SELECT CURRENT_TIMESTAMP,
       EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS year,
       EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS month,
       EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS day,
       EXTRACT(HOUR FROM CURRENT_TIMESTAMP) AS hour,
       EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS minute,
       EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second
FROM SYSIBM.SYSDUMMY1;
```

## KEYWORD

- 转换函数
- 类型转换
- cast

## 转换函数

最后将要给大家介绍一类比较特殊的函数——转换函数。虽说有些特殊，但是由于这些函数的语法和之前介绍的函数类似，数量也比较少，因此很容易记忆。

## 0

类型变换在一般的编程语言中也会使用，因此并不是SQL特有的功能。

## KEYWORD

## ● CAST 函数

“转换”这个词的含义非常广泛，在SQL中主要有两层意思。一是数据类型转换，简称为“类型转换”，在英语中称为 cast<sup>①</sup>。另一层意思是值的转换。

## ■ CAST——类型转换

## 语法 6-14 CAST 函数

```
CAST( 转换前的值 AS 想要转换的数据类型)
```

进行类型转换需要使用 CAST 函数。

之所以需要进行类型转换，是因为可能会插入与表中数据类型不匹配的数据，或者在进行运算时由于数据类型不一致发生了错误，又或者是进行自动类型转换造成了处理速度的低下。这些时候都需要事前进行数据类型转换（代码清单 6-17、代码清单 6-18）。

## 代码清单 6-17 将字符串类型转换为数值类型

```
SQL Server PostgreSQL
```

```
SELECT CAST('0001' AS INTEGER) AS int_col;
```

```
MySQL
```

```
SELECT CAST('0001' AS SIGNED INTEGER) AS int_col;
```

```
Oracle
```

```
SELECT CAST('0001' AS INTEGER) AS int_col  
FROM DUAL;
```

```
DB2
```

```
SELECT CAST('0001' AS INTEGER) AS int_col  
FROM SYSIBM.SYSDUMMY1;
```

## 执行结果

```
int_col  
-----  
1
```

## 代码清单 6-18 将字符串类型转换为日期类型

```
SQL Server PostgreSQL MySQL
```

```
SELECT CAST('2009-12-14' AS DATE) AS date_col;
```

```
Oracle
```

```
SELECT CAST('2009-12-14' AS DATE) AS date_col  
FROM DUAL;
```

```

DB2
SELECT CAST('2009-12-14' AS DATE) AS date_col
FROM SYSIBM.SYSDUMMY1;

```

#### 执行结果

```

date_col
-----
2009-12-14

```

从上述结果可以看到，将字符串类型转换为整数类型时，去掉前面的“000”之后，就转换为整数了。但是，将字符串转换为日期类型时，从结果上并不能看出数据发生了什么变化，理解起来也比较困难。从中我们也可以看出，类型转换其实并不是为了方便用户使用而开发的功能，而是为了方便 DBMS 内部处理而开发的功能。

### ■ COALESCE——将 NULL 转换为其他值

#### 语法 6-15 COALESCE 函数

```
COALESCE(数据1, 数据2, 数据3……)
```

#### KEYWORD

##### ● COALESCE 函数

#### ①

参数的个数并不固定，可以自由设定个数的参数。

COALESCE 是 SQL 特有的函数。该函数会返回可变参数<sup>①</sup>中左侧开始第 1 个不是 NULL 的值。参数个数是可变的，因此可以根据需要无限增加。

其实转换函数的使用还是非常频繁的。在 SQL 语句中将 NULL 转换为其他值时就会用到转换函数（代码清单 6-19、代码清单 6-20）。就像之前我们学习的那样，运算或者函数中含有 NULL 时，结果全都会变为 NULL。能够避免这种结果的函数就是 COALESCE。

#### 代码清单 6-19 将 NULL 转换为其他值

```

SQL Server PostgreSQL MySQL
SELECT COALESCE(NULL, 1) AS col_1,
       COALESCE(NULL, 'test', NULL) AS col_2,
       COALESCE(NULL, NULL, '2009-11-01') AS col_3;

Oracle
SELECT COALESCE(NULL, 1) AS col_1,
       COALESCE(NULL, 'test', NULL) AS col_2,
       COALESCE(NULL, NULL, '2009-11-01') AS col_3
FROM DUAL;

```

```

DB2
SELECT COALESCE(NULL, 1) AS col_1,
       COALESCE(NULL, 'test', NULL) AS col_2,
       COALESCE(NULL, NULL, '2009-11-01') AS col_3
FROM SYSIBM.SYSDUMMY1;

```

执行结果

```

col_1 | col_2 | col_3
-----+-----+-----
1 | test | 2009-11-01

```

代码清单 6-20 使用 SampleStr 表中的列作例子

```

SELECT COALESCE(str2, 'NULL')
FROM SampleStr;

```

执行结果

```

coalesce
-----
iv v
def
太郎
'NULL'
aaa
'NULL'
'NULL'
'NULL'
abc
abc
a

```

这样，即使包含 NULL 的列，也可以通过 COALESCE 函数转换为其他值之后再应用到函数或者运算当中，这样结果就不再是 NULL 了。

此外，多数 DBMS 中都提供了特有的 COALESCE 的简化版函数（如 Oracle 中的 NVL 等）。但由于这些函数都依存于各自的 DBMS，因此还是推荐大家使用通用的 COALESCE 函数。

# 6-2

## 谓词

### 学习重点

- 谓词就是返回值为真值的函数。
- 掌握 LIKE 的三种使用方法(前方一致、中间一致、后方一致)。
- 需要注意 BETWEEN 包含三个参数。
- 想要取得 NULL 数据时必须使用 IS NULL。
- 可以将子查询作为 IN 和 EXISTS 的参数。

### KEYWORD

- 谓词

### 什么是谓词

本节将会和大家一起学习 SQL 的抽出条件中不可或缺的工具——谓词 (predicate)。虽然之前我们没有提及谓词这个概念,但其实大家已经使用过了。例如, =、<、>、<> 等比较运算符,其正式的名称就是比较谓词。

谓词,通俗来讲就是 6-1 节中介绍的函数中的一种,是需要满足特定条件的函数。该条件就是“返回值是真值”。对通常的函数来说,返回值有可能是数字字符串或者日期等等,但是谓词的返回值全都是真值(TRUE/FALSE/UNKNOWN)。这也是谓词和函数的最大区别。

本节将会介绍以下谓词。

- LIKE
- BETWEEN
- IS NULL、IS NOT NULL
- IN
- EXISTS

### LIKE 谓词——字符串的部分一致查询

截至目前,我们使用字符串作为查询条件的例子中,使用的都是 =。这里的 = 只有在字符串完全一致时才为真。与之相反,LIKE 谓词更加模



**KEYWORD**

- LIKE 谓词
- 部分一致查询

糊一些，当需要进行字符串的部分一致查询时需要使用该谓词。

部分一致大体可以分为前方一致、中间一致和后方一致三种类型。接下来就让我们来看一看具体示例吧。

首先我们来创建一张表 6-1 那样的只有 1 列的表。

**表 6-1 SampleLike 表**

strcol(字符串)
abcddd
dddabc
abdddc
abcdd
ddabc
abddc

创建上表以及向其中插入数据的 SQL 语句请参考代码清单 6-21。

**代码清单 6-21 创建 SampleLike 表**

```
-- DDL, 创建表
CREATE TABLE SampleLike
( strcol VARCHAR(6) NOT NULL,
  PRIMARY KEY (strcol));

SQL Server, PostgreSQL
-- DML, 插入数据
BEGIN TRANSACTION; ①

INSERT INTO SampleLike (strcol) VALUES ('abcddd');
INSERT INTO SampleLike (strcol) VALUES ('dddabc');
INSERT INTO SampleLike (strcol) VALUES ('abdddc');
INSERT INTO SampleLike (strcol) VALUES ('abcdd');
INSERT INTO SampleLike (strcol) VALUES ('ddabc');
INSERT INTO SampleLike (strcol) VALUES ('abddc');

COMMIT;
```

**特定的 SQL**

不同的 DBMS 事务处理的语法也不尽相同。代码清单 6-21 中的 DML 语句在 MySQL 中执行时，需要将①部分更改为“START TRANSACTION”。在 Oracle 和 DB2 中执行时，无需用到①的部分（请删除）。

详细内容请大家参考第 4 章中的“创建事务”。

想要从该表中读取包含字符串“ddd”的记录时，可能会得到前方

一致、中间一致和后方一致等不同的结果。

#### ●前方一致: 选取“dddabc”

所谓前方一致, 就是选取出作为查询条件的字符串(这里是“ddd”)与查询对象字符串起始部分相同的记录的查询方法。

#### ●中间一致: 选取“abcdddd”、“dddabc”、“abdddc”

所谓中间一致, 就是选取出查询对象字符串中含有作为查询条件的字符串(这里是“ddd”)的记录查询方法。无论该字符串出现在对象字符串的最后还是中间都没有关系。

#### ●后方一致: 选取“abcdddd”

后方一致与前方一致相反, 也就是选取出作为查询条件的字符串(这里是“ddd”)与查询对象字符串的末尾部分相同的记录的查询方法。

从本例中我们可以看出, 查询条件最宽松, 也就是能够取得更多记录的是中间一致。这是因为它同时包含前方一致和后方一致的查询结果。

像这样不使用“=”来指定条件字符串, 而以字符串中是否包含该条件(本例中是“包含ddd”这样的规则)的规则为基础的查询称为“模式匹配”。其中的模式也就是前面提到的“规则”。

### ■进行前方一致查询

下面让我们来实际操作一下, 对 SampleLike 表进行前方一致查询吧(代码清单 6-22)。

代码清单 6-22 使用 LIKE 进行前方一致查询

```
SELECT *
FROM SampleLike
WHERE strcol LIKE 'ddd%';
```

#### 执行结果

```
strcol
-----
dddabc
```

其中的“%”是代表“0 字符以上的任意字符串”的特殊符号。本例中代表“以 ddd 开头的所有字符串”。

#### KEYWORD

- 前方一致
- 中间一致
- 后方一致

#### KEYWORD

- 模式匹配
- 模式

#### KEYWORD

- %

这样我们就可以使用 LIKE 和模式匹配来进行查询了。

### ■中间一致查询

接下来让我们看一个中间一致的例子，查询出包含字符串“ddd”的记录（代码清单 6-23）。

代码清单 6-23 使用 LIKE 进行中间一致查询

```
SELECT *
  FROM SampleLike
 WHERE strcol LIKE '%ddd%';
```

#### 执行结果

```
strcol
-----
abcddd
dddabc
abdddc
```

在字符串的起始和结束位置加上 %，就能取出“包含 ddd 的字符串”了。

### ■后方一致查询

最后我们来看一下后方一致查询，选取出以字符串“ddd”结尾的记录（代码清单 6-24）。

代码清单 6-24 使用 LIKE 进行后方一致查询

```
SELECT *
  FROM SampleLike
 WHERE strcol LIKE '%ddd';
```

#### 执行结果

```
strcol
-----
abcddd
```

大家可以看到上述结果与前方一致正好相反。

此外，我们还可以使用 \_（下划线）来代替 %，与 % 不同的是其代表了“任意 1 个字符”。下面就让我们来尝试一下吧。

使用代码清单 6-25 选取出 strcol 列的值为“abc+任意 2 个字符”的记录。

代码清单6-25 使用LIKE和\_(下划线)进行后方一致查询

```
SELECT *
  FROM SampleLike
 WHERE strcol LIKE 'abc__';
```

执行结果

```
strcol
-----
abcd
```

“abcd”也是以“abc”开头的字符串，但是其中“d”是3个字符，所以不满足\_\_所指定的2个字符的条件。因此该字符串并不在查询结果之中。相反，代码清单6-26中的SQL语句就只能取出“abcd”这个结果。

代码清单6-26 查询“abc+任意3个字符”的字符串

```
SELECT *
  FROM SampleLike
 WHERE strcol LIKE 'abc___';
```

执行结果

```
strcol
-----
abcd
```

## BETWEEN 谓词——范围查询

使用 BETWEEN 可以进行范围查询。该谓词与其他谓词或者函数的不同之处在于它使用了3个参数。例如，从 shohin（商品）表中读取出销售单价（hanbai\_tanka）为100日元到1000日元之间的商品时，可以使用代码清单6-27中的SQL语句。

代码清单6-27 选取销售单价为100-1000日元的商品

```
SELECT shohin_mei, hanbai_tanka
  FROM Shohin
 WHERE hanbai_tanka BETWEEN 100 AND 1000;
```

### KEYWORD

- BETWEEN 谓词
- 范围查询

执行结果

shohin_mei	hanbai_tanka
T恤衫	1000
打孔器	500
叉子	500
擦菜板	880
圆珠笔	100

#### KEYWORD

- <
- >

BETWEEN 的特点就是结果中会包含 100 和 1000 这两个临界值。如果不想让结果中包含临界值，那就必须使用 < 和 > (代码清单 6-28)。

代码清单 6-28 选取出销售单价为 101~999 日元的商品

```
SELECT shohin_mei, hanbai_tanka
FROM Shohin
WHERE hanbai_tanka > 100
AND hanbai_tanka < 1000;
```

执行结果

shohin_mei	hanbai_tanka
打孔器	500
叉子	500
擦菜板	880

执行结果中不再包含 1000 日元和 100 日元的记录。

## IS NULL、IS NOT NULL——判断是否为 NULL

为了选取某些值为 NULL 的列的数据，不能使用 =，而只能使用特定的谓词 IS NULL (代码清单 6-29)。

代码清单 6-29 选取出进货单价(shiire\_tanka)为 NULL 的商品

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka IS NULL;
```

执行结果

shohin_mei	shiire_tanka
叉子	
圆珠笔	

#### KEYWORD

- IS NULL 谓词

**KEYWORD**

● IS NOT NULL谓词

与此相反,想要选取 NULL 以外的数据时,需要使用 IS NOT NULL(代码清单 6-30)。

代码清单 6-30 选取进货单价(shiire\_tanka)不为NULL的商品

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka IS NOT NULL;
```

执行结果

shohin_mei	shiire_tanka
T恤衫	500
打孔器	320
运动T恤	2800
菜刀	2800
高压锅	5000
擦菜板	790

## IN谓词——OR的简使用法

接下来让我们思考一下如何选取出货单价(shiire\_tanka)为320日元、500日元、5000日元的商品吧。我们使用之前学过的 OR 的 SQL 语句,请参考代码清单 6-31。

代码清单 6-31 通过OR指定多个进货单价进行查询

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka = 320
OR shiire_tanka = 500
OR shiire_tanka = 5000;
```

执行结果

shohin_mei	shiire_tanka
T恤衫	500
打孔器	320
高压锅	5000

虽然上述方法没有问题,但还是存在一点不足之处,那就是随着希望选取的对象越来越多,SQL语句也会越来越长,阅读起来也会越来越困难。这时,我们就可以使用代码清单 6-32 中的 IN 谓词“IN(值,……)”来替换上述 SQL 语句。

**KEYWORD**

● IN谓词

**KEYWORD**

● NOT IN 谓词

代码清单 6-32 通过 IN 来指定多个进货单价进行查询

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka IN (320, 500, 5000);
```

反之，希望选取出“进货单价不是 320 日元、500 日元、5000 日元”的商品时，可以使用否定形式 NOT IN 来实现（代码清单 6-33）。

代码清单 6-33 使用 NOT IN 进行查询时指定多个除外的进货单价进行查询

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka NOT IN (320, 500, 5000);
```

执行结果

shohin_mei	shiire_tanka
运动T恤	2800
菜刀	2800
擦菜板	790

但需要注意的是，在使用 IN 和 NOT IN 时是无法选取出 NULL 数据的。实际结果也是如此，上述两组结果中都不包含进货单价为 NULL 的叉子和圆珠笔。NULL 终究还是需要使用 IS NULL 和 IS NOT NULL 来进行判断。

## 使用子查询作为 IN 谓词的参数

### ■ IN 和子查询

IN 谓词（NOT IN 谓词）具有其他谓词所没有的使用方法。那就是可以使用子查询作为其参数来使用。我们已经在 5-2 节中学习过了，子查询就是 SQL 内部生成的表。因此也可以说“能够将表作为 IN 的参数”。同理，我们还可以说“能够将视图作为 IN 的参数”。

为了掌握详细的使用方法，让我们再添加一张新表吧。之前我们使用的全都是显示商品在库一览信息的 Shohin（商品）表，但现实中这些商品可能会在个别的商店中进行销售。下面我们来创建表 6-2，显示出哪些商店销售哪些商品的表 TenpoShohin（商店商品）。

表6-2 TenpoShohin(商店商品)表

tenpo_id (商店)	tenpomei (商店名称)	shohin_id (商品编号)	suryo (数量)
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50
000C	大阪	0006	90
000C	大阪	0007	70
000D	福冈	0001	100

商店和商品组合成为一条记录。例如，该表显示出东京店销售的商品有001(T恤)、002(打孔器)、003(运动T恤)三种。

创建该表的SQL语句请参考代码清单6-34。

代码清单6-34 创建TenpoShohin(商店商品)表的CREATE TABLE语句

```
CREATE TABLE TenpoShohin
(tenpo_id CHAR(4) NOT NULL,
tenpo_mei VARCHAR(200) NOT NULL,
shohin_id CHAR(4) NOT NULL,
suryo INTEGER NOT NULL,
PRIMARY KEY (tenpo_id, shohin_id));
```

该CREATE TABLE语句的特点是指定了2列作为主键(primary key)。这样做的目的当然还是为了区分表中每一行数据，由于单独使用商店编号(tenpo\_mei)或者商品编号(shohin\_id)不能满足要求，所以需要商店和商品进行组合。

实际上如果只使用商店编号进行区分，那么指定“000A”作为条件能够查询出3行数据。而单独使用商品编号进行区分的话，“0001”也会查询出2行数据。都无法恰当区分每行数据。



下面让我们来看一下向 TenpoShohin 表中插入数据的 INSERT 语句（代码清单 6-35）。

代码清单 6-35 向 TenpoShohin 表中插入数据的 INSERT 语句

```

SQL Server PostgreSQL
BEGIN TRANSACTION; ①

INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000A', '东京', '0001', 30);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000A', '东京', '0002', 50);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000A', '东京', '0003', 15);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000B', '名古屋', '0002', 30);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000B', '名古屋', '0003', 120);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000B', '名古屋', '0004', 20);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000B', '名古屋', '0006', 10);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000B', '名古屋', '0007', 40);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000C', '大阪', '0003', 20);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000C', '大阪', '0004', 50);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000C', '大阪', '0006', 90);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000C', '大阪', '0007', 70);
INSERT INTO TenpoShohin (tempo_id, tempo_mei, shohin_id, suryo) VALUES ('000D', '福冈', '0001', 100);

COMMIT;

```

#### 特定的 SQL

不同的 DBMS 事务处理的语法也不尽相同。代码清单 6-35 在 MySQL 中执行时，需要将①部分更改为“START TRANSACTION;”。在 Oracle 和 DB2 中执行时，无需用到①的部分（请删除）。

详细内容请大家参考第 4 章中“创建事务”部分的内容。

这样我们就完成了全部准备工作，下面就让我们来看一看在 IN 谓词中使用子查询的 SQL 的写法吧。

首先读取出“大阪店（000C）在售商品（shohin\_id）的销售单价（hanbai\_tanka）”。

TenpoShohin（商店商品）表中大阪店的在售商品很容易就能找出，有如下 4 种。

- 运动T恤（商品编号：0003）
- 菜刀（商品编号：0004）
- 叉子（商品编号：0006）
- 煎菜板（商品编号：0007）

结果自然也应该是这样。

shohin_mei	hanbai_tanka
运动T恤	4000
菜刀	3000
叉子	500
擦菜板	880

得到上述结果时，我们应该已经完了成如下两个步骤。

1. 从 TenpoShohin 表中选取出在大阪店 (tempo\_id = '000C') 中销售的商品 (shohin\_id)。
2. 从 Shohin 表中选取出通过 1 得到的商品 (shohin\_id) 的销售单价 (hanbai\_tanka)。

SQL 也是如此，同样的要分两步来完成。首先，第一步如下所示。

```
SELECT shohin_id
FROM TenpoShohin
WHERE tempo_id = '000C';
```

因为大阪店的商店编号 (tempo\_id) 是 “000C”，所以我们可以将其作为条件写在 WHERE 子句中<sup>❶</sup>。接下来，我们就可以把上述 SELECT 语句作为第二步中的条件来使用了。最终得到的 SELECT 语句请参考代码清单 6-36。

#### ❶

虽然使用 'tempo\_mei='大阪'' 作为条件可以得到同样的结果，但是通常情况下，指定数据库中的商店或者商品时，并不会直接使用日语名称。这是因为与编号比起来，名称更有可能发生改变。

#### 代码清单 6-36 使用子查询作为 IN 的参数

```
-- 读取出“大阪店(000C)在售商品(shohin_id)的销售单价(hanbai_tanka)
SELECT shohin_mei, hanbai_tanka
FROM Shohin
WHERE shohin_id IN (SELECT shohin_id
                     FROM TenpoShohin
                     WHERE tempo_id = '000C');
```

#### 执行结果

shohin_mei	hanbai_tanka
叉子	500
运动T恤	4000
菜刀	3000
擦菜板	880

如第5章“法则5-6”所述，子查询是从内层开始执行的。因此，该SELECT语句也是从内层的子查询开始执行，然后像下面这样展开。

```
-- 子查询展开后的结果
SELECT shohin_mei, hanbai_tanka
FROM Shohin
WHERE shohin_id IN ('0003', '0004', '0006', '0007');
```

这样就转换成了之前我们学习过的IN的使用方法了吧。可能有些读者会产生这样的疑问：“既然子查询展开后得到的结果同样是('0003', '0004', '0006', '0007')，为什么一定要使用子查询呢？”

这是因为TenpoShohin(商店商品)表并不是一成不变的。实际上由于各个商店销售的商品都在不断发生变化，所以TenpoShohin表内大饭店销售的商品也会发生变化。如果SELECT语句中没有使用子查询的话，一旦商品发生了改变，那么SELECT语句也不得不进行修改。而且这样的修改工作会变得没完没了。

反之，如果在SELECT语句中使用了子查询，那么即使数据发生了变更，还可以继续使用同样的SELECT语句。这样也就减少了我们的常规作业(单纯的重复操作)。

像这样可以完美应对数据变更的程序称为“易维护程序”，或者“免维护程序”<sup>①</sup>。这也是系统开发中需要重点考虑的部分。希望大家在开始学习编程时，就能够有意识地编写易于维护的代码。

#### ①

这里的“free”和“tax free”中的“free”同样都是“不要”或者“免除”的意思。

### ■ NOT IN 和子查询

IN的否定形式NOT IN同样可以使用子查询作为参数，其语法也和IN完全一样。请大家参考代码清单6-37中的例文。

代码清单6-37 使用子查询作为NOT IN的参数

```
SELECT shohin_mei, hanbai_tanka
FROM Shohin
WHERE shohin_id NOT IN (SELECT shohin_id
                        FROM TenpoShohin
                        WHERE tempo_id = '000A');
```

本例中的SQL语句是要选取“在东京店(000A)以外销售的商品(shohin\_id)的销售单价(hanbai\_tanka)”。 “NOT IN”代表了“以

外”这样的否定含义。

我们也像之前那样来看一下该 SQL 的执行步骤。因为还是首先执行子查询，所以会得到如下结果。

```
-- 执行子查询
SELECT shohin_mei, hanbai_tanka
FROM Shohin
WHERE shohin_id NOT IN ('0001', '0002', '0003');
```

之后就很简单了，上述语句应该会返回 0001-0003 “以外”的结果。

执行结果

shohin_mei	hanbai_tanka
菜刀	3000
高压锅	6800
叉子	500
擦菜板	880
圆珠笔	100

## EXIST 谓词

### KEYWORD

#### ● EXIST 谓词

本节最后将要给大家介绍的是 EXIST 谓词。将它放到最后进行学习的原因有以下 3 点。

- ① EXIST 的使用方法与之前的都不相同。
- ② 语法理解起来比较困难。
- ③ 实际上即使不使用 EXIST，基本上也都可以使用 IN（或者 NOT IN）来代替。

理由①和②都说明 EXIST 是使用方法特殊而难以理解的谓词。特别是使用否定形式 NOT EXIST 的 SELECT 语句，即使是 DB 工程师也常常无法迅速理解。此外，如理由③所述，由于使用 IN 作为替代的情况非常多（尽管不能完全替代让人有些伤脑筋），很多读者虽然记住了使用方法但还是不能实际运用。

但是如果能够熟练使用 EXIST 谓词的话，就能体会到它的极大的便利性。因此，非常希望大家能够在达到 SQL 中级水平时掌握此工具，本

## ①

希望了解EXIST谓词详细内容的读者，可以参考拙著《深入浅出SQL数据库》（翔泳公司出版）中（1-8 EXIST谓词的使用方法）的内容。

书只简单介绍其基本使用方法<sup>①</sup>。

接下来就让我们赶快看一看 EXIST 吧。

### ■ EXIST谓词的使用方法

一言以蔽之，谓词的作用就是“判断是否存在满足某种条件的记录”。如果存在这样的记录就返回真（TRUE），如果不存在就返回假（FALSE）。EXIST（存在）谓词的主语是“记录”。

我们继续使用前一节“IN和子查询”中的示例，使用EXIST选取“大饭店（000C）在售商品（shohin\_id）的销售单价（hanbai\_tanka）”。

SELECT语句请参考代码清单6-38。

代码清单6-38 使用EXIST选取“大饭店（000C）在售商品（shohin\_id）的销售单价（hanbai\_tanka）”

```

SQL Server  DB2  PostgreSQL  MySQL
SELECT shohin_mei, hanbai_tanka
FROM Shohin AS S -----①
WHERE EXISTS (SELECT *
FROM TenpoShohin AS TS -----②
WHERE TS.tempo_id = '000C'
AND TS.shohin_id = S.shohin_id);

```

#### 特定的SQL

Oracle的FROM子句中不能使用AS（会发生错误）。因此，在Oracle中执行代码清单6-38时，请将①的部分修改为“FROM Shohin S”，将②的部分修改为“FROM TenpoShohin TS”（删除FROM子句中的AS）。

#### 执行结果

shohin_mei	hanbai_tanka
叉子	500
运动T恤	4000
菜刀	3000
煎菜板	880

### ● EXIST的参数

之前我们学过的谓词，基本上都是“列LIKE字符串”或者“列BETWEEN值1AND值2”这样需要指定2个以上的参数。而EXIST的左侧并没有任何参数。很奇妙吧？这是因为EXIST是只有1个参数的谓词。EXIST只需要在右侧书写一个参数。该参数通常都会是一个子查询。这种情况下，

```
(SELECT *
FROM TenpoShohin AS TS
WHERE TS.tenpo_id = '000C'
AND TS.shohin_id = S.shohin_id)
```

这样的子查询就是唯一的参数。确切地说，由于通过条件“TS.shohin\_id = S.shohin\_id”将 Shohin 表和 TenpoShohin 表进行了联接，所以作为参数的是关联子查询。EXIST 通常都会使用关联子查询作为参数<sup>①</sup>。

## ①

虽然严格来说语法上也可以使用非关联子查询作为参数，但实际应用中几乎没有这样的情况。



## 法则 6-1

通常指定关联子查询作为 EXIST 的参数。

## ● 子查询中的 SELECT \*

可能大家会觉得子查询中的 SELECT \* 稍微有些不同，就像我们之前学到的那样，由于 EXIST 只关心记录是否存在，所以返回哪些列都没有关系。EXIST 是用来判断是否存在满足子查询中 WHERE 子句指定条件“商店编号 (tenpo\_id) 为 '000C'，商品 (Shohin) 表和商店商品 (TenpoShohin) 表中商品编号 (shohin\_id) 相同”的记录，只有存在这样的记录时才返回真 (TRUE) 的谓词。

因此，即使写成代码清单 6-39 那样，结果也不会发生改变。

代码清单 6-39 这样的写法也能得到 6-38 相同的结果

```
SQL Server DB2 PostgreSQL MySQL
SELECT shohin_mei, hanbai_tanka
FROM Shohin AS S ①
WHERE EXISTS (SELECT 1 -- 此处可以书写适当的常数
FROM TenpoShohin AS TS ②
WHERE TS.tenpo_id = '000C'
AND TS.shohin_id = S.shohin_id);
```

## 特定的 SQL

在 Oracle 中执行代码清单 6-39 时，请将<sup>①</sup>的部分修改为“FROM Shohin S”，将<sup>②</sup>的部分修改为“FROM TenpoShohin TS”（删除 FROM 子句中的 AS）。

大家可以把在 EXIST 的子查询中书写 SELECT \* 当作 SQL 的一种习惯。



## 法则 6-2

作为 EXIST 参数的子查询中经常会使用 SELECT \*。

## KEYWORD

● NOT EXIST 谓词

## ● 使用 NOT EXIST 替换 NOT IN

就像 EXIST 可以用来替换 IN 一样，NOT IN 也可以用 NOT EXIST 来替换。下面就让我们使用 NOT EXIST 来编写一条 SELECT 语句，读取“东京店(000A)在售之外的商品(shohin\_id)的销售单价(hanbai\_tanka)”(代码清单 6-40)。

代码清单 6-40 使用 NOT EXIST 读取“东京店在售之外的商品的销售单价”

```

SQL Server  DB2  PostgreSQL  MySQL
SELECT shohin_mei, hanbai_tanka
FROM Shohin AS S ①
WHERE NOT EXISTS (SELECT *
                   FROM TenpoShohin AS TS ②
                   WHERE TS.tenpo_id = '000A'
                   AND TS.shohin_id = S.shohin_id);

```

## 特定的 SQL

在 Oracle 中执行代码清单 6-40 时，请将①的部分修改为“FROM Shohin S”，将②的部分修改为“FROM TenpoShohin TS”(删除 FROM 子句中的 AS)。

## 执行结果

shohin_mei	hanbai_tanka
菜刀	3000
高压锅	6800
叉子	500
煎菜板	880
圆珠笔	100

NOT EXIST 与 EXIST 相反，当“不存在”满足子查询中指定条件的记录时返回真 (TRUE)。

将 IN (代码清单 6-36) 和 EXIST (代码清单 6-38) 的 SELECT 语句进行比较，会得到怎样的结果呢。可能大多数读者会觉得 IN 理解起来要容易一些。笔者也认为没有必要勉强使用 EXIST。因为 EXIST 拥有 IN 所不具有的便利性，严格来说两者并不相同，所以希望大家能够在中级篇中掌握这两种谓词的使用方法。

## 6-3

## CASE 表达式

## 学习重点

- CASE表达式分为简单CASE表达式和搜索CASE表达式两种。搜索CASE表达式包含简单CASE表达式的全部功能。
- 虽然CASE表达式中的ELSE子句可以省略,但为了让SQL语句更加容易理解,还是希望大家不要省略。
- CASE表达式中的END不能省略。
- 使用CASE表达式能够将SELECT语句的结果进行组合。
- 虽然有些DBMS提供了各自特有的CASE表达式的简化函数,例如Oracle中的DECODE和MySQL中的IF,等等,但由于它们并非通用的函数,功能上也有限制,因此有些场合无法使用。

## 什么是CASE表达式

本节将要学习的CASE表达式,和“1+1”或者“120/4”这样的表达式一样,是一种进行运算的功能。这就意味着CASE表达式也是函数的一种。它是SQL中数一数二的重要功能,希望大家能够在这里好好学习掌握。

CASE表达式,正如CASE(情况)这个词的含义所示,是在区分情况时使用的。这种情况的区分在程序中通常称为“(条件)分歧”<sup>①</sup>。

## CASE表达式的语法

CASE表达式的语法分为简单CASE表达式和搜索CASE表达式两种。但是,由于搜索CASE表达式包含了简单CASE表达式的全部功能,因此本节只会介绍搜索CASE表达式。想要了解简单CASE表达式语法的读者,可以参考本节末尾的“简单CASE表达式”专栏。

下面就让我们赶快来学习一下搜索CASE表达式的语法吧。

## KEYWORD

- CASE表达式
- 分歧(条件分歧)

## ①

在C语言和Java这样流行的编程语言中,通常都会使用IF语句或者CASE语句。CASE表达式就是这些语句的SQL版本。

## KEYWORD

- 简单CASE表达式
- 搜索CASE表达式



语法 6-16 搜索 CASE 表达式

```

CASE WHEN <判断表达式> THEN <表达式>
      WHEN <判断表达式> THEN <表达式>
      WHEN <判断表达式> THEN <表达式>
      :
      ELSE <表达式>
END

```

**KEYWORD**

- WHEN 子句
- 判断
- THEN 子句
- ELSE

WHEN 子句中的 <判断表达式> 就是类似“列 = 值”这样，返回值为真值 (TRUE/FALSE/UNKNOWN) 的表达式。我们也可以将其看作使用 =、!= 或者 LIKE、BETWEEN 等谓词编写出来的表达式。

CASE 表达式会从对最初的 WHEN 子句中的 <判断表达式> 进行判断开始执行。所谓“判断”，就是要调查该表达式的真值是什么。如果结果为真 (TRUE)，那么就返回 THEN 子句中的表达式，CASE 表达式的执行到此为止。如果结果不为真，那么就跳转到下一条 WHEN 子句的判断之中。如果直到最后的 WHEN 子句为止返回结果都不为真，那么就会返回 ELSE 中的表达式，执行终止。

从 CASE 表达式名称中的“表达式”我们也能看出来，上述这些整体构成了一个表达式。并且由于表达式最终会返回一个值，因此 CASE 表达式在 SQL 语句执行时，也会转化为一个值。虽然使用分支众多的 CASE 表达式编写数十行代码的情况也并不少见，但是无论多么庞大的 CASE 表达式，最后也只会返回类似“1”或者“渡边先生”这样简单的值。

## CASE 表达式的使用方法

那么，就让我们来学习一下 CASE 表达式的具体使用方法吧。例如我们来考虑这样一种情况，现在 Shohin (商品) 表中包含衣服、办公用品和厨房用具 3 种商品类型。请大家考虑一下怎样才能够得到如下结果呢。

```

A: 衣服
B: 办公用品
C: 厨房用具

```

因为表中的记录并不包含“A:”或者“B:”这样的字符串，所以需

要在 SQL 中进行添加。我们可以使用 6-1 节中学过的字符串连接函数“||”来完成这项工作。

剩下的问题就是怎样正确地将“A:”“B:”“C:”与记录结合起来。这时就可以使用 CASE 表达式来实现这样的要求了（代码清单 6-41）。

代码清单 6-41 通过 CASE 表达式将 A-C 的字符串加入到商品分类当中

```
SELECT shohin_mei,
       CASE WHEN shohin_bunrui = '衣服'
            THEN 'A:' || shohin_bunrui
            WHEN shohin_bunrui = '办公用品'
            THEN 'B:' || shohin_bunrui
            WHEN shohin_bunrui = '厨房用具'
            THEN 'C:' || shohin_bunrui
            ELSE NULL
       END AS abc_shohin_bunrui
FROM Shohin;
```

执行结果

shohin_mei	abc_shohin_bunrui
T恤衫	A. 衣服
打孔器	B. 办公用品
运动T恤	A. 衣服
菜刀	C. 厨房用具
高压锅	C. 厨房用具
叉子	C. 厨房用具
煎菜板	C. 厨房用具
圆珠笔	B. 办公用品

6行 CASE 表达式代码最后只相当于一列(abc\_shohin\_bunrui)而已，大家也许有点吃惊吧！与商品种类(shohin\_bunrui)的名称相对应，CASE 表达式中包含了三条 WHEN 子句分支。最后的“ELSE NULL”代表了“上述情况之外时返回 NULL”的意思。ELSE 子句指定了不满足 WHEN 子句中条件的记录应该执行何种操作。NULL 之外的其他值或者表达式也都可以写在 ELSE 子句之中。但由于现在表中包含的商品种类只有 3 种，因此实际上有没有 ELSE 子句都是一样的。

ELSE 子句也可以省略不写，这时会自动默认为“ELSE NULL”。但为了防止有人漏读，所以还是希望大家能够写明 ELSE 子句。

#### KEYWORD

- ELSE NULL

**法则 6-3**

虽然CASE表达式中的ELSE子句可以省略，但还是希望大家不要省略。

此外，CASE表达式最后的“END”是不能省略的，请大家特别注意不要遗漏。忘记书写END会发生语法错误，这也是初学时最容易犯的错误。

**法则 6-4**

CASE表达式中的END不能省略。

**■ CASE表达式的书写位置**

CASE表达式的便利之处就在于它是一个表达式。之所以这么说，是因为表达式可以书写在任意位置。也就是像“1+1”这样写在什么位置都可以的意思。例如，我们可以利用CASE表达式将下述SELECT语句结果中的行和列进行互换。

**执行结果**

```
sum_tanka_ihuku|sum_tanka_kitchen|sum_tanka_jimu
-----
5000 | 11180 | 600
```

上述结果是根据商品种类计算出的销售单价的合计值，通常我们将商品种类列作为GROUP BY子句的聚合键来使用，但是这样得到的结果会以“行”的形式输出，而无法以列的形式进行排列（代码清单6-42）。

**代码清单6-42 通常使用GROUP BY也无法实现行列转换**

```
SELECT shohin_bunrui,
       SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY shohin_bunrui;
```

**执行结果**

```
shohin_bunrui|sum_tanka
-----+-----
衣服 | 5000
办公用品 | 600
厨房用具 | 11180
```

我们可以像代码清单 6-43 那样在 SUM 函数中使用 CASE 表达式来获得一个三列的结果。

代码清单 6-43 使用 CASE 表达式进行行列转换

```
-- 对按照商品种类计算出的销售单价合计值进行行列转换
SELECT SUM(CASE WHEN shohin_bunrui = '衣服'
                THEN hanbai_tanka ELSE 0 END) AS sum_tanka_ihuku,
       SUM(CASE WHEN shohin_bunrui = '厨房用具'
                THEN hanbai_tanka ELSE 0 END) AS sum_tanka_kitchen,
       SUM(CASE WHEN shohin_bunrui = '办公用品'
                THEN hanbai_tanka ELSE 0 END) AS sum_tanka_jimu
FROM Shohin;
```

上述 CASE 表达式，在满足商品种类 (shohin\_bunrui) 为“衣服”或者“办公用品”等特定值时，输出该商品的销售单价 (hanbai\_tanka)，不满足时输出 0。对该结果进行合计处理，就能够得到特定商品种类的销售单价合计值了。

CASE 表达式在对 SELECT 语句的结果进行编辑时，能够发挥较大作用。

## 专 栏

### 简单 CASE 表达式

CASE 表达式分为两种，一种是本节学习的“搜索 CASE 表达式”，另一种就是其简化形式——“简单 CASE 表达式”。

简单 CASE 表达式比搜索 CASE 表达式简单，但是会受到条件的约束。因此，通常情况下都会使用搜索 CASE 表达式。在此我们简单介绍一下其语法结构。

简单 CASE 表达式的语法如下所示。

#### 语法 6-A 简单 CASE 表达式

```
CASE <表达式>
  WHEN <表达式> THEN <表达式>
  WHEN <表达式> THEN <表达式>
  WHEN <表达式> THEN <表达式>
  :
  ELSE <表达式>
END
```

与搜索CASE表达式一样,简单CASE表达式也是从最初的WHEN子句开始进行,逐一判断每个WHEN子句直到返回真值为止。此外,没有能够返回真值的WHEN子句时,也会返回ELSE子句指定的表达式。两者的不同之处在于,简单CASE表达式最初的“CASE<表达式>”也会作为判断的对象。

下面就让我们来看看搜索CASE表达式和简单CASE表达式是如何实现相同含义的SQL语句的吧。将代码清单6-41中的搜索CASE表达式的SQL改写为简单CASE表达式的结果如下所示(代码清单6-A)。

#### 代码清单6-A 使用CASE表达式将字符串A~C添加到商品种类中

```
-- 使用搜索CASE表达式(代码清单6-41)
SELECT shohin_mei,
       CASEWHEN shohin_bunrui = '衣服'
         THEN 'A:' ||shohin_bunrui
         WHEN shohin_bunrui = '办公用品'
         THEN 'B:' ||shohin_bunrui
         WHEN shohin_bunrui = '厨房用具'
         THEN 'C:' ||shohin_bunrui
         ELSE NULL
       END AS abc_shohin_bunrui
FROM Shohin;

-- 使用简单CASE表达式
SELECT shohin_mei,
       CASE shohin_bunrui
         WHEN '衣服' THEN 'A:' || shohin_bunrui
         WHEN '办公用品' THEN 'B:' || shohin_bunrui
         WHEN '厨房用具' THEN 'C:' || shohin_bunrui
         ELSE NULL
       END AS abc_shohin_bunrui
FROM Shohin;
```

像“CASE shohin\_mei”这样,简单CASE表达式在将想要判断的表达式(这里是列)书写过一次之后,就无需在之后的WHEN子句中重复书写“shohin\_bunrui”了。虽然看上去简化了书写,但是想要在WHEN子句中指定不同列时,简单CASE表达式就无能为力了。

**KEYWORD**

- DECODE 函数 (Oracle)
- IF 函数 (MySQL)

**专 栏****特定的 CASE 表达式**

由于 CASE 表达式是标准 SQL 所承认的功能, 因此在任何 DBMS 中都可以执行。但是, 有些 DBMS 还提供了一些特有的 CASE 表达式的简化函数。例如, Oracle 中的 DECODE, MySQL 中的 IF 等等。

使用 Oracle 中的 DECODE 和 MySQL 中的 IF 将字符串 A ~ C 添加到商品种类 (shohin\_bunrui) 中的 SQL 请参考代码清单 6-B。

**代码清单 6-B 使用 CASE 的特定语句将字符串 A ~ C 添加到商品种类**

```

Oracle
-- 在Oracle中使用DECODE来代替CASE表达式
SELECT shohin_mei,
       DECODE(shohin_bunrui,
              '衣服',      'A.' || shohin_bunrui,
              '办公用品', 'B.' || shohin_bunrui,
              '厨房用具', 'C.' || shohin_bunrui,
              NULL) AS abc_shohin_bunrui
FROM Shohin;

MySQL
-- 在MySQL中使用IF来代替CASE表达式
SELECT shohin_mei,
       IF( IF( IF(shohin_bunrui = '衣服',
                  CONCAT('A.', shohin_bunrui), NULL)
            IS NULL AND shohin_bunrui = '办公用品',
            CONCAT('B.', shohin_bunrui),
            IF(shohin_bunrui = '衣服',
               CONCAT('A.', shohin_bunrui), NULL))
        IS NULL AND shohin_bunrui = '厨房用具',
        CONCAT('C.', shohin_bunrui),
        IF( IF(shohin_bunrui = '衣服',
               CONCAT('A.', shohin_bunrui), NULL)
          IS NULL AND shohin_bunrui = '办公用品',
          CONCAT('B.', shohin_bunrui),
          IF(shohin_bunrui = '衣服',
             CONCAT('A.', shohin_bunrui),
             NULL))) AS abc_shohin_bunrui
FROM Shohin;

```

但上述函数只能在特定的 DBMS 中使用, 并且能够使用的条件也没有 CASE 表达式那么丰富, 因此并没有什么优势。希望大家尽量不要使用这些特定的 SQL 语句。

### 练习题

6.1 对本章中使用的 Shohin (商品) 表执行如下 2 条 SELECT 语句, 能够得到什么样的结果呢?

①

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka NOT IN (500, 2800, 5000);
```

②

```
SELECT shohin_mei, shiire_tanka
FROM Shohin
WHERE shiire_tanka NOT IN (500, 2800, 5000, NULL);
```

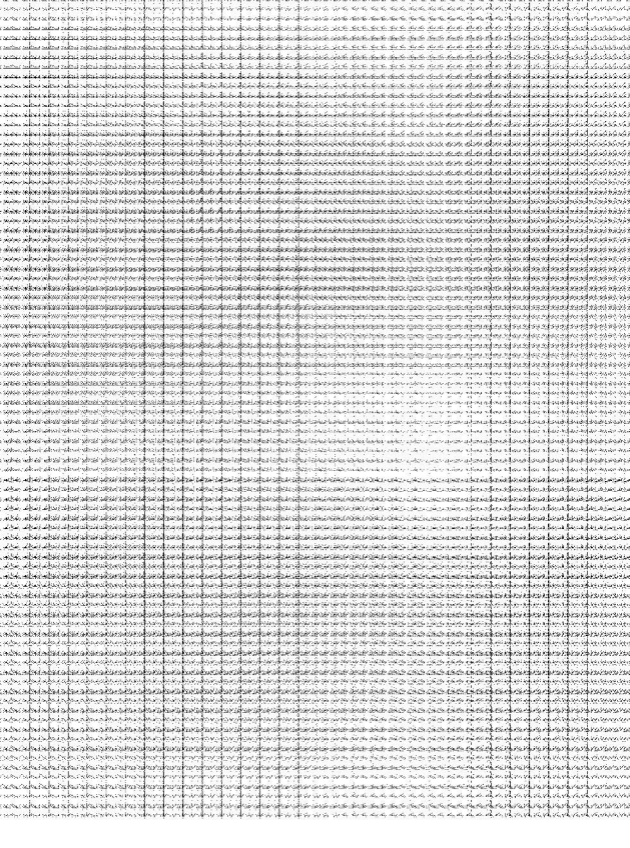
6.2 按照销售单价 (hanbai\_tanka), 对练习 6.1 中的 Shohin (商品) 表中的商品进行如下分类。

- 低档商品: 销售单价在 1000 日元以下 (T 恤、办公用品、叉子、擦菜板、圆珠笔)
- 中档商品: 销售单价在 1001 日元以上 3000 日元以下 (菜刀)
- 高档商品: 销售单价在 3001 日元以上 (运动 T 恤、高压锅)

请编写出统计上述商品种类所包含的商品数量的 SELECT 语句。结果如下所示。

执行结果

```
low_price | mid_price | mid_price
-----+-----+-----
          5 |          1 |          2
```





## 第7章 集合函数

张松海 编著

联结(以列为单位)对数据进行操作

# 7



# SQL

## 本章重点

---

前面几章我们学习了使用一张表的 SQL 语句的书写方法。本章将会和大家一起学习使用 2 张以上表的 SQL 语句。通过对表进行行方向（竖）的集合运算符和进行列方向（横）的联结，就可以将分散在多张表中的数据组合成为期望的结果。

### 7-1 表的加减法

- 什么是集合运算

- 表的加法——UNION

- 集合运算的注意事项

- 包含重复行的集合运算——ALL 选项

- 选取表中公共部分——INTERSECT

- 记录的减法——EXCEPT

### 7-2 联结（以列为单位对表进行联结）

- 什么是联结

- 内联结——INNER JOIN

- 外联结——OUTER JOIN

- 3 张以上表的联结

- 交叉联结——CROSS JOIN

- 特定的联结语句和过时的语法

## 7-1

## 表的加减法

## 学习重点

- 集合运算就是对满足同一规则的记录进行的加减等四则运算。
- 使用 UNION ( 并集 )、INTERSECT ( 交集 )、EXCEPT ( 差集 ) 等集合运算符来进行集合运算。
- 集合运算符可以去除重复行。
- 如果希望集合运算符保留重复行,就需要使用 ALL 选项。

## KEYWORD

- 集合运算
- 集合
- 记录的集合
- 集合运算符

## 什么是集合运算

本章将会和大家一起学习“集合运算”操作。在数学领域,“集合”表示“(各种各样的)事物的总和”;在数据库领域,表示“记录的集合”。具体来说,表、视图和查询的执行结果都是“记录的集合”。

截至目前,我们已经学习了从表中读取数据以及插入数据的方法。所谓集合运算,就是对满足同一规则的记录进行的加减等“四则运算”。通过集合运算,可以得到两张表中记录的集合,或者是公共记录的集合,又或者是其中某张表记录的集合。像这样用来进行集合运算的运算符称为“集合运算符”。

本节将会为大家介绍表的加减法,下一节将会和大家一起学习进行“表联结”的集合运算符以及它们的使用方法。

## 表的加法——UNION

## KEYWORD

- UNION ( 并集 )

首先为大家介绍的集合运算符是进行记录加法运算的 UNION ( 并集 )。

在学习具体的使用方法之前,我们首先添加一张表。该表的结构与之前我们使用的 Shohin ( 商品 ) 表相同,只是表名变为 Shohin2 ( 商品 2 ) ( 代码清单 7-1 )。

代码清单 7-1 创建表 Shohin2(商品 2)

```
CREATE TABLE Shohin2
(shohin_id      CHAR(4)      NOT NULL,
shohin_mei     VARCHAR(100) NOT NULL,
shohin_bunrui VARCHAR(32)  NOT NULL,
hanbai_tanka   INTEGER      ,
shiire_tanka   INTEGER      ,
torokubi       DATE         ,
PRIMARY KEY (shohin_id));
```

接下来，我们将代码清单 7-2 中的 5 条记录插入到 Shohin2 表中。商品编号 (shohin\_id) 为“0001”~“0003”的商品与之前 Shohin 表中的商品相同，而编号为“0009”的“手套”和“0010”的“水壶”是 Shohin 表中没有的商品。

代码清单 7-2 将数据插入到表 Shohin2(商品 2)中

```
SQL Server | PostgreSQL
BEGIN TRANSACTION;
INSERT INTO Shohin2 VALUES ('0001', 'T恤衫', '衣服', 1000, 500, →
'2008-09-20');
INSERT INTO Shohin2 VALUES ('0002', '打孔器', '办公用品', 500, →
320, '2009-09-11');
INSERT INTO Shohin2 VALUES ('0003', '运动T恤', '衣服', 4000, →
2800, NULL);
INSERT INTO Shohin2 VALUES ('0009', '手套', '衣服', 800, 500, NULL);
INSERT INTO Shohin2 VALUES ('0010', '水壶', '厨房用具', 2000, →
1700, '2009-09-20');
COMMIT;
```

➡表示下一行接续本行，只是由于版面所限而换行。

#### 特定的 SQL

不同的 DBMS，事务处理的语法也不尽相同。代码清单 7-2 中的 DML 语句在 MySQL 中执行时，需要将①部分更改为“START TRANSACTION;”。在 Oracle 和 DB2 中执行时，无需用到①的部分（请删除）。

详细内容请大家参考第 4 章中的“创建事务”。

这样，我们的准备工作就完成了。接下来，就让我们对上述两张表进行“Shohin 表+Shohin2 表”这样的加法计算吧。语法请参考代码清单 7-3。

代码清单 7-3 使用 UNION 对表进行加法运算

```

SELECT shohin_id, shohin_mei
FROM Shohin
UNION
SELECT shohin_id, shohin_mei
FROM Shohin2;

```

执行结果

shohin_id	shohin_mei
0001	T恤衫
0002	打孔器
0003	运动T恤
0004	菜刀
0005	高压锅
0006	叉子
0007	擦菜板
0008	圆珠笔
0009	手套
0010	水壶

上述结果包含了两张表中的所有商品。可能有些读者会发现，这就是我们在学校学过的集合中的并集运算。通过文氏图会看得更清晰（图 7-1）。

图 7-1 使用 UNION 对表进行加法（并集）运算的图示



\* 括号内的数字代表了商品的编号

商品编号为“0001”~“0003”的 3 条记录在两个表中都存在，因此大家可能会认为结果中会出现重复的记录。但是 UNION 等集合运算符通常会除去重复的记录。



## 法则 7-1

集合运算符会除去重复的记录。

## 集合运算的注意事项

其实结果中也可以包含重复的记录，在介绍该方法之前，还是让我们先来学习一下使用集合运算符时的注意事项吧。不仅限于 UNION，之后将要学习的所有运算符都要遵守这些注意事项。

### ■ 注意事项①——作为运算对象的记录的列数必须相同

例如，像下面这样，一部分记录包含 2 列，另一部分记录包含 3 列时，会发生错误，无法进行加法运算。

```
-- 列数不一致时会发生错误
SELECT shohin_id, shohin_mei
FROM Shohin
UNION
SELECT shohin_id, shohin_mei, hanbai_tanka
FROM Shohin2;
```

### ■ 注意事项②——作为运算对象的记录中列的类型必须一致

从左侧开始，相同位置上的列必须是同一数据类型。例如，下面的 SQL 语句，虽然列数相同，但是第 2 列的数据类型并不一致（一个是数值类型，一个是日期类型），因此会发生错误<sup>①</sup>。

```
-- 数据类型不一致时会发生错误
SELECT shohin_id, hanbai_tanka
FROM Shohin
UNION
SELECT shohin_id, torokubi
FROM Shohin2;
```

一定要使用不同数据类型的列时，可以使用 6-1 节中的类型转换函数 CAST。

### ■ 注意事项③——可以使用任何 SELECT 语句，但 ORDER BY 子句只能在最后使用一次

通过 UNION 进行并集运算时可以使用任何形式的 SELECT 语句。之前学过的 WHERE、GROUP BY、HAVING 等子句都可以使用。但是 ORDER BY 只能在最后使用一次（代码清单 7-4）。

代码清单 7-4 ORDER BY 子句只在最后使用一次

```
SELECT shohin_id, shohin_mei
FROM Shohin
```

#### ①

实际上，在有些 DBMS 中，即使数据类型不同，也可以通过隐式类型转换来完成操作。但由于并非所有的 DBMS 都支持这样的用法，所以还是希望大家能够使用恰当的数据类型来进行运算。

```

WHERE shohin_bunrui = '厨房用具'
UNION
SELECT shohin_id, shohin_mei
FROM Shohin2
WHERE shohin_bunrui = '厨房用具'
ORDER BY shohin_id;

```

执行结果

shohin_id	shohin_mei
0004	菜刀
0005	高压锅
0006	叉子
0007	煎菜板
0010	水壶

## KEYWORD

● ALL 选项

## 包含重复行的集合运算——ALL 选项

接下来给大家介绍在 UNION 的结果中保留重复行的语法。其实非常简单，只需要在 UNION 后面添加 ALL 关键字就可以了。这里的 ALL 选项，在 UNION 之外的集合运算符中同样可以使用（代码清单 7-5）。

代码清单 7-5 保留重复行

```

SELECT shohin_id, shohin_mei
FROM Shohin
UNION ALL
SELECT shohin_id, shohin_mei
FROM Shohin2;

```

执行结果

shohin_id	shohin_mei
0001	T 恤衫
0002	打孔器
0003	运动 T 恤
0004	菜刀
0005	高压锅
0006	叉子
0007	煎菜板
0008	圆珠笔
0001	T 恤衫
0002	打孔器
0003	运动 T 恤
0009	手袋
0010	水壶

这3行记录是重复的



## 法则 7-2

在集合运算符中使用 ALL 选项，可以保留重复行。

## 选取表中公共部分——INTERSECT

下面将要介绍的集合运算符在数的四则运算中并不存在。其实也不是很难理解，那就是选取 2 个记录集中公共部分的 INTERSECT（交集）<sup>①</sup>。

让我们赶快来看一下吧。其语法和 UNION 完全一样（代码清单 7-6）。

### KEYWORD

● INTERSECT(交集)

①

由于 MySQL 尚不支持 INTERSECT，所以无法使用。

代码清单 7-6 使用 INTERSECT 选取出表中公共部分

```

Oracle | SQL Server | DB2 | PostgreSQL
SELECT shohin_id, shohin_mei
  FROM Shohin
INTERSECT
SELECT shohin_id, shohin_mei
  FROM Shohin2
ORDER BY shohin_id;

```

执行结果

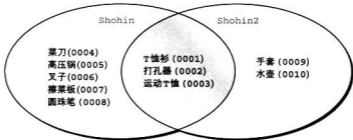
```

shohin_id | shohin_mei
-----+-----
0001      | T恤衫
0002      | 打孔器
0003      | 运动T恤

```

大家可以看到，结果中只包含两张表记录中公共部分。该运算的文氏图如下所示（图 7-2）。

图 7-2 使用 INTERSECT 选取出表中公共部分的图示



① 括号内的数字代表了商品的编号



与使用 AND 可以选出一张表中满足多个条件的公共部分不同，INTERSECT 应用于两张表，选取出它们当中的公共记录。

其注意事项与 UNION 相同，我们在“集合运算的注意事项”和“保留重复行的集合运算”中已经介绍过了。希望保留重复行时同样需要使用 INTERSECT ALL。

## 记录的减法——EXCEPT

本节最后将要给大家介绍的集合运算符就是进行减法运算的 EXCEPT (差集)<sup>①</sup>。其语法也与 UNION 相同(代码清单 7-7)。

代码清单 7-7 使用 EXCEPT 对记录进行减法运算

```
SQL Server  DB2  PostgreSQL
SELECT shohin_id, shohin_mei
FROM Shohin
EXCEPT
SELECT shohin_id, shohin_mei
FROM Shohin2
ORDER BY shohin_id;
```

### 特定的 SQL

在 Oracle 中执行代码清单 7-7 或者代码清单 7-8 种的 SQL 时，请将 EXCEPT 改为 MINUS。

```
-- Oracle中使用MINUS而不是EXCEPT
SELECT ...
FROM ...
MINUS
SELECT ...
FROM ...;
```

### 执行结果

```
shohin_id | shohin_mei
-----|-----
0004      | 菜刀
0005      | 高压锅
0006      | 叉子
0007      | 擦桌板
0008      | 圆珠笔
```

### KEYWORD

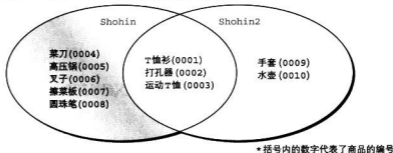
- EXCEPT(差集)

### ①

只有 Oracle 不使用 EXCEPT，而是使用其特有的 MINUS 运算符。使用 Oracle 的用户，请用 MINUS 代替 EXCEPT。此外，MySQL 还不支持 EXCEPT，因此也无法使用。

大家可以看到，结果中只包含 Shohin 表的记录中除去 Shohin2 表中记录的剩余部分。运算图示如图 7-3 所示。

图7-3 使用 EXCEPT 对记录进行减法运算的图示



EXCEPT 有一点与 UNION 和 INTERSECT 不同，需要注意一下。那就是在减法运算中减数和被减数的位置不同，所得到的结果也不相同。4+2 和 2+4 的结果相同，但是 4-2 和 2-4 的结果却不一样。因此，我们将之前 SQL 中的 Shohin 和 Shohin2 互换，就能得到代码清单 7-8 中的结果。

代码清单 7-8 被减数和减数位置不同，得到的结果也不同

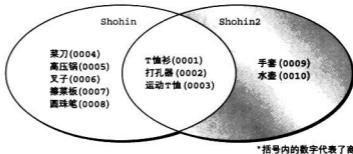
```
SQL Server DB2 PostgreSQL
-- 从Shohin2的记录中除去Shohin中存在的记录
SELECT shohin_id, shohin_mei
FROM Shohin2
EXCEPT
SELECT shohin_id, shohin_mei
FROM Shohin
ORDER BY shohin_id;
```

执行结果

```
shohin_id|shohin_mei
-----
0009    | 手套
0010    | 水壶
```

上述运算的文氏图如图 7-4 所示。

图7-4 使用EXCEPT对记录进行减法运算的图示(从Shohin2中除去Shohin中的记录)



到此，对 SQL 提供的集合运算符的学习已经结束了。可能有些读者会有“唉，怎么没有乘法和除法呢？”这样的疑问。关于乘法的相关内容，我们将在下一节进行详细介绍。此外，SQL 中虽然也存在除法，但由于除法是比较难理解的运算，属于中级内容，所以我们会在本章末尾的专栏中进行一些简单的介绍。感兴趣的读者请参考专栏“关系除法”。

## 7-2

## 联结(以列为单位对表进行联结)

## 学习重点

- 联结 (JOIN) 就是将其其他表中的列添加过来, 进行“添加列”的集合运算。UNION 是表中将满足相同规则的记录以行(纵向)为单位进行联结, 而联结是以列(横向)为单位进行联结。
- 联结大体上分为内联结和外联结两种。首先请大家清晰地掌握这两种联结的使用方法。
- 请大家一定要使用标准的 SQL 语法格式, 而不要使用过时或者特定联结运算的书写方式。但还是希望大家能够了解一下这些过时或者特定的书写方式。

## 什么是联结

前一节我们学习了 UNION 和 INTERSECT 等集合运算。这些集合运算的特征就是以行方向为单位进行操作。通俗来说, 就是进行这些集合运算时, 会导致记录行数的增减。使用 UNION 会增加记录行数, 而使用 INTERSECT 或者 EXCEPT 会减少记录行数<sup>①</sup>。

但是这些运算不会导致列数的改变。作为集合运算对象表的前提就是列数要一致。因此, 运算结果不会导致列的增减。

本节将要学习的联结 (JOIN) 运算, 简单来说, 就是将其其他表中的列添加过来, 进行“添加列”的运算(图 7-5)。该操作通常用于无法从一张表中获取期望数据(列)的情况。截至目前, 本书中出现的示例基本上都是从一张表中选取数据, 但实际上, 期望得到的数据往往会分散在不同的表之中。使用联结就可以从多张表(3张以上的表也没关系)中选取数据了。

图 7-5 联结的图示



<sup>①</sup> 根据表中数据的不同, 也存在行数不发生变化的情况。

## KEYWORD

- 联结 (JOIN)

SQL 的联结根据其用途可以分为很多种类。这里希望大家掌握的有两种,内联结和外联结。接下来,我们就以这两种联结为中心进行学习。

## 内联结——INNER JOIN

### KEYWORD

●内联结(INNER JOIN)

首先我们来学习内联结,它是应用最广泛的联结运算。大家现在可以暂时忽略“内”这个字,之后会给大家进行详细说明。

本例中我们会继续使用 Shohin 表和第 6 章创建的 TenpoShohin 表。下面我们再来回顾一下这两张表的内容。

表 7-1 Shohin(商品)表

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shiire_tanka (进货单价)	torokubi (登记日期)
0001	T 恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

表 7-2 TenpoShohin(商店商品)表

tenpo_id (商店编号)	tenpomei (商店名称)	shohin_id (商品编号)	suryo (数量)
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50

[续表]

tenpo_id (商店编号)	tenpomei (商店名称)	shohin_id (商品编号)	suryo (数量)
000C	大阪	0006	90
000C	大阪	0007	70
000D	福冈	0001	100

对这两张表包含的列进行整理后的结果如表 7-3 所示。

表 7-3 两张表及其包含的列

	Shohin	TenpoShohin
商品编号	○	○
商品名称	○	
商品分类	○	
销售单价	○	
进货单价	○	
登记日期	○	
商店编号		○
商店名称		○
数量		○

如上表所示，两张表中的列可以分为如下两类。

- Ⓐ 两张表中都包含的列 → 商品编号
- Ⓑ 只存在于一张表内的列 → 商品编号之外的列

所谓联结运算，一言以蔽之，就是“以Ⓐ中的列作为桥梁，将Ⓑ中满足同样条件的列汇集到同一结果之中”。具体过程如下所述。

从 TenpoShohin 表中的数据我们能够知道，东京店（000A）销售商品的编号为 0001、0002 和 0003。但这些商品具体的名称（shohin\_mei）和销售单价（hanbai\_tanka）在 TenpoShohin 表中并不存在。这些信息都保存在 Shohin 表中。大阪店和名古屋店的情况也是如此。

下面我们就试着从 Shohin 表中取出商品名称（shohin\_mei）和销售单价（hanbai\_tanka），与 TenpoShohin 表中的内容进行结合，所得到的结果如下所示。

## 执行结果

tenpo_id	tenpo_mei	shohin_id	shohin_mei	hanbai_tanka
000A	东京	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000A	东京	0001	T恤衫	1000
000B	名古屋	0007	擦菜板	880
000B	名古屋	0002	打孔器	500
000B	名古屋	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000B	名古屋	0006	叉子	500
000C	大阪	0007	擦菜板	880
000C	大阪	0006	叉子	500
000C	大阪	0003	运动T恤	4000
000C	大阪	0004	菜刀	3000
000D	福冈	0001	T恤衫	1000

能够得到上述结果的 SELECT 语句如代码清单 7-9 所示。

## 代码清单 7-9 将两张表进行内联结

```

SQL Server  DB2  PostgreSQL  MySQL
SELECT TS.tenpo_id, TS.tenpo_mei, TS.shohin_id, S.shohin_mei, ⇨
S.hanbai_tanka
FROM TenpoShohin AS TS INNER JOIN Shohin AS S ———①
ON TS.shohin_id = S.shohin_id;

```

⇨表示下一行接续本行，只是由于版面所限而换行。

## 特定的SQL

在 Oracle 的 FROM 子句中不能使用 AS (会发生错误)。因此，在 Oracle 中执行代码清单 7-9 时，请将①的部分变为“FROM TenpoShohin TS INNER JOIN Shohin S”。

关于内联结，请大家注意以下 3 点。

## ●内联结要点①——FROM子句

第 1 点要注意的是，之前的 FROM 子句中只有一张表，而这次我们同时使用了 TenpoShohin 和 Shohin 两张表。

```
FROM TenpoShohin AS TS INNER JOIN Shohin AS S
```

使用关键字 INNER JOIN 就可以将两张表联结在一起了。TS 和 S 分别是这两张表的别名，但别名并不是必须的。在 SELECT 子句中直接使用 TenpoShohin 和 shohin\_id 这样表的原名也没有关系。但

## ①

在FROM子句中使用表的别名时,像Shohin AS S这样使用AS是标准SQL正式的语法。但是在Oracle中使用AS会发生错误。因此,在Oracle中使用时,需要注意不要在FROM子句中使用AS。

## KEYWORD

● ON子句

## KEYWORD

● 联结键

由于表明太长会影响SQL语句的可读性,所以还是希望大家能够习惯使用别名<sup>①</sup>。



## 法则7-3

进行联结时需要在FROM子句中使用多张表。

## ● 内联结要点②——ON子句

第2点要注意的是ON后面所记载的联结条件。

```
ON TS.shohin_id = S.shohin_id
```

我们可以在ON之后指定两张表联结所使用的列(联结键)。本例中使用的是商品编号(shohin\_id)。也就是说ON是专门用来指定联结条件的,它能起到与WHERE相同的作用。需要指定多个键时,同样可以使用AND、OR。ON子句在进行内联结时是必不可少的(如果没有ON会发生错误)。并且,ON必须书写在FROM和WHERE之间。



## 法则7-4

进行内联结时必须使用ON子句,并且要书写在FROM和WHERE之间。

举个比较直观的例子,ON就像是连接河流两岸城镇的桥梁一样(图7-6)。

图7-6 使用ON进行两表加法运算(和集)的图示



联结条件也可以使用“=”来记述。在语法上,还可以使用<=和BETWEEN等谓词。但由于实际应用中九成以上都可以用“=”进行联结,所以开始时大家只要记住使用“=”就可以了。使用“=”将联结键进行关联,就能够将两张表中满足相同条件的记录进行“联结”了。



### ●内联结要点③——SELECT子句

第3点要注意的是,在SELECT子句中指定的列。

```
SELECT TS.tempo_id, TS.tempo_mei, TS.shohin_id, S.shohin_mei, →  
S.hanbai_tanka
```

→表示下一行接续本行,只是由于版面所限而换行。

在SELECT子句中,使用像TS.tempo\_id和S.hanbai\_tanka这样<表的别名>.<列名>的形式来指定列。这和使用一张表时的情况不同,由于多表联结时,某个列到底属于哪张表比较容易混乱,所以采用了这样的防范措施。语法上必须使用这样书写方式的只是那些同时存在于两张表中的列(这里是shohin\_id),其他的列可以像tempo\_id这样直接书写列名而不会发生错误。但是就像前面说的那样,为了避免混乱,还是希望大家能够在使用联结时按照<表的别名>.<列名>的格式来书写SELECT子句中全部的列。



#### 法则 7-5

使用联结时SELECT子句中的列需要按照<表的别名>.<列名>的格式进行书写。

### ■内联结和WHERE子句结合使用

如果并不想了解所有商店的情况,例如只想知道东京店(000A)的信息时,可以像之前学习的那样在WHERE子句中添加条件。这样我们就可以根据代码清单7-9中得到的全部商店的信息中选取东京店的记录了。

#### 代码清单7-10 内联结和WHERE子句结合使用

```
SQL Server DB2 PostgreSQL MySQL  
SELECT TS.tempo_id, TS.tempo_mei, TS.shohin_id, S.shohin_mei, →  
S.hanbai_tanka  
FROM TenpoShohin AS TS INNER JOIN Shohin AS S ①  
ON TS.shohin_id = S.shohin_id  
WHERE TS.tempo_id = '000A';
```

→表示下一行接续本行,只是由于版面所限而换行。

#### 特定的SQL

在Oracle中执行代码清单7-10时,请将①的部分变为“FROM TenpoShohin TS INNER JOIN Shohin S”(删掉FROM子句中的AS)。

执行结果

```

tenpo_id | tenpo_mei | shohin_id | shohin_mei | hanbai_tanka
-----+-----+-----+-----+-----
000A    | 东京      | 0001      | T恤衫      | 1000
000A    | 东京      | 0002      | 打孔器      | 500
000A    | 东京      | 0003      | 运动T恤     | 4000

```

像这样使用联结运算将满足相同规定的表联结起来时，WHERE、GROUP BY、HAVING、ORDER BY 等工具都可以正常使用。我们可以将联结之后的结果想象为新创建出来的一张表（表 7-4），对这张表使用 WHERE 子句等工具，这样理解起来就容易多了吧。

当然，这张“表”只在 SELECT 语句执行期间存在，SELECT 语句执行之后就会消失。如果希望继续使用这张“表”，还是将它创建成视图吧。

表 7-4 通过联结创建出的表 (ShohinJoinTenpoShohin) 的图示

tenpo_id (编号)	tenpo_mei (商品名称)	shohin_id (商品编号)	shohin_mei (商品名称)	hanbai_tanka (销售单价)
000A	东京	0001	T恤衫	1000
000A	东京	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000B	名古屋	0002	打孔器	500
000B	名古屋	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000B	名古屋	0006	叉子	500
000B	名古屋	0007	擦菜板	880
000C	大阪	0003	运动T恤	4000
000C	大阪	0004	菜刀	3000
000C	大阪	0006	叉子	500
000C	大阪	0007	擦菜板	880
000D	福冈	0001	T恤衫	1000

## 外联结——OUTER JOIN

内联结之外比较重要的就是外联结了。我们再来回顾一下前面的例子。在前例中，我们将 Shohin 表和 TenpoShohin 表进行内联结，从两张

### KEYWORD

●外部链接 (OUTER JOIN)

表中取出各个商店销售的商品信息。其中,实现“读取两张表信息”的就是联结功能。

外联结也是通过 ON 子句使用联结键将两张表进行联结,同时从两张表中选取相应的列。基本的使用方法并没有发生改变,只是结果却有所不同。事实胜于雄辩,还是让我们先把之前内联结的 SELECT 语句(代码清单 7-9)转换为外联结试试看吧。转换的结果请参考代码清单 7-11。

代码清单 7-11 将两张表进行外联结

```

SQL Server  DB2  PostgreSQL  MySQL
SELECT TS.tempo_id, TS.tempo_mei, S.shohin_id, S.shohin_mei, →
S.hanbai_tanka
FROM TenpoShohin AS TS RIGHT OUTER JOIN Shohin AS S ①
ON TS.shohin_id = S.shohin_id;

```

→表示下一行接续本行,只是由于版面所限而换行。

#### 特定的SQL

在 Oracle 中执行代码清单 7-11 时,请将①的部分变为“FROM TenpoShohin TS RIGHT OUTER JOIN Shohin S”(删除掉 FROM 子句中的 AS)。

#### 执行结果

tempo_id	tempo_mei	shohin_id	shohin_mei	hanbai_tanka
000A	东京	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000A	东京	0001	T恤衫	1000
000B	名古屋	0006	叉子	500
000B	名古屋	0002	打孔器	500
000B	名古屋	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000B	名古屋	0007	擦菜板	880
000C	大阪	0006	叉子	500
000C	大阪	0007	擦菜板	880
000C	大阪	0003	运动T恤	4000
000C	大阪	0004	菜刀	3000
000D	福冈	0001	T恤衫	1000
		0005	高压锅	6800
		0008	圆珠笔	100

内联结时并不存在!

#### ●外联结要点①——选取出单张表中全部的信息

与内联结的结果相比,不同点显而易见,那就是结果的行数不一样。内联结的结果中有 13 条记录,而外联结的结果中有 15 条记录。增加的 2 条记录到底是什么呢?

这正是外联结的关键点。多出的2条记录是高压锅和圆珠笔。这2条记录在 TenpoShohin 表中并不存在。也就是说这2种商品在任何商店中都没有销售。由于内联结只能选取出同时存在于两张表中的数据，所以只在 Shohin 表中存在的2种商品并没有出现在结果之中。

相反，对于外联结来说，只要数据存在于某一张表中，就能够读取出来。在实际的业务中，例如想要生成固定行数的定型单据时，就需要使用外联结。如果使用内联结的话，根据 SELECT 语句执行时商店库存状况的不同，结果的行数也会发生改变。生成单据的外观也会受到影响。而使用外联结能够得到固定行数的结果。

虽说如此，那些表中不存在的信息我们还是无法得到，结果中高压锅和圆珠笔的商店编号和商店名称都是 NULL（具体信息大家都不知道，真是无可奈何）。外联结名称的由来也跟 NULL 有关。所谓“外部”，也就是“包含元表中不存在（在元表之外）的信息”的意思。相反，只包含表内信息的联结也就被称为“内”联结了。

### ●外联结要点②——每张表都是主表吗？

外联结还有一点非常重要，那就是要把哪张表作为主表。最终的结果中会包含主表内所有的数据。指定主表的关键词是 LEFT 和 RIGHT。如其名称所示，使用 LEFT 时 FROM 子句中写在左侧的表是主表，使用 RIGHT 时右侧的表是主表。代码清单 7-11 中使用了 RIGHT，因此，右侧的表，也就是 Shohin 表是主表。

我们还可以像代码清单 7-12 这样进行调换，意思完全相同。

代码清单 7-12 调换后外联结的结果完全相同

```

SQL Server DB2 PostgreSQL MySQL
SELECT TS.tenpo_id, TS.tenpo_mei, S.shohin_id, S.shohin_mei,
S.hanbai tanka
FROM Shohin AS S LEFT OUTER JOIN TenpoShohin AS TS
ON TS.shohin_id = S.shohin_id;

```

⇒表示下一行接续本行，只是由于版面所限而换行。

#### 特定的 SQL

在 Oracle 中执行代码清单 7-12 时，请将①的部分变为“FROM TenpoShohin TS LEFT OUTER JOIN Shohin S”（删除掉 FROM 子句中的 AS ）。

#### KEYWORD

- LEFT 关键字
- RIGHT 关键字

大家可能会犹豫到底应该使用 LEFT 还是 RIGHT 呢, 其实它们的功能没有任何区别, 使用哪一个都可以。通常, 使用 LEFT 的情况会多一些, 但并没有非使用这个不可的理由, 使用 RIGHT 也没有问题。



#### 法则 7-6

外联结中使用 LEFT、RIGHT 来指定主表。使用二者所得到的结果完全相同。

## 3 张以上表的联结

通常联结只涉及两张表, 但有时也会出现必须同时联结 3 张以上表的情况。原则上联结表的数量并没有限制。下面就让我们来看一下 3 张表的联结吧。

首先我们创建一张用来管理库存商品的表(表 7-5)。假设商品都保存在 S001 和 S002 这 2 个仓库之中。

表 7-5 ZaikoShohin(库存商品)表

souko_id { 仓库编号 }	shohin_id { 商品编号 }	zaiko_suryo { 库存数量 }
S001	0001	0
S001	0002	120
S001	0003	200
S001	0004	3
S001	0005	0
S001	0006	99
S001	0007	999
S001	0008	200
S002	0001	10
S002	0002	25
S002	0003	34
S002	0004	19
S002	0005	99
S002	0006	0
S002	0007	0
S002	0008	18

创建该表及插入数据的 SQL 语句请参考代码清单 7-13。

代码清单 7-13 创建 ZaikoShohin 表并向其中插入数据

```
-- DDL: 创建表
CREATE TABLE ZaikoShohin
( souko_id CHAR(4) NOT NULL,
  shohin_id CHAR(4) NOT NULL,
  zaiko_suryo INTEGER NOT NULL,
  PRIMARY KEY (souko_id, shohin_id));

SQL Server PostgreSQL
-- DML: 插入数据
BEGIN TRANSACTION;

INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0001', 0);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0002', 120);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0003', 200);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0004', 3);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0005', 0);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0006', 99);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0007', 999);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S001', '0008', 200);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0001', 10);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0002', 25);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0003', 34);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0004', 19);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0005', 99);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0006', 0);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0007', 0);
INSERT INTO ZaikoShohin (souko_id, shohin_id, zaiko_suryo) ➡
VALUES ('S002', '0008', 18);

COMMIT;
```

➡表示下一行接续本行，只是由于版面所限而换行。

**特定的SQL**

不同的DBMS, 事务处理的语法也不尽相同。代码清单7-13中的DML语句在MySQL中执行时, 需要将①部分更改为“START TRANSACTION;”。在Oracle和DB2中执行时, 无需用到①的部分(请删除)。

详细内容请大家参考第4章中的“创建事务”。

下面我们从上表中取出保存在S001仓库中的商品数量, 并将该列添加到代码清单7-11所得到的结果中。联结方式为内联结(外联结的使用方法完全相同), 联结键为商品编号(shohin\_id)(代码清单7-14)。

**代码清单7-14 对3张表进行内联结**

```

SQL Server  DB2  PostgreSQL  MySQL
SELECT TS.tempo_id, TS.tempo_mei, TS.shohin_id, S.shohin_mei, →
S.hanbai_tanka, ZS.zaiko_suryo
FROM TempoShohin AS TS INNER JOIN Shohin AS S ①
ON TS.shohin_id = S.shohin_id
INNER JOIN ZaikoShohin AS ZS ②
ON TS.shohin_id = ZS.shohin_id
WHERE ZS.souko_id = 'S001';

```

⇒表示下一行接续本行, 只是由于版面所限而换行。

**特定的SQL**

在Oracle中执行代码清单7-14时, 请将①的部分变为“FROM TempoShohin TS INNER JOIN Shohin S”, 将②的部分变为“INNER JOIN ZaikoShohin ZS”(删除掉FROM子句中的AS)。

**执行结果**

tempo_id	tempo_mei	shohin_id	shohin_mei	hanbai_tanka	zaiko_suryo
000A	东京	0002	打孔器	500	120
000A	东京	0003	运动T恤	4000	200
000A	东京	0001	T恤衫	1000	0
000B	名古屋	0007	擦菜板	880	999
000B	名古屋	0002	打孔器	500	120
000B	名古屋	0003	运动T恤	4000	200
000B	名古屋	0004	菜刀	3000	3
000B	名古屋	0006	叉子	500	99
000C	大阪	0007	擦菜板	880	999
000C	大阪	0006	叉子	500	99
000C	大阪	0003	运动T恤	4000	200
000C	大阪	0004	菜刀	3000	3
000D	福岡	0001	T恤衫	1000	0

在代码清单7-11内联结的FROM子句中, 再次使用INNER JOIN将

ZaikoShohin 表也添加了过来。

```
FROM TenpoShohin AS TS INNER JOIN Shohin AS S
  ON TS.shohin_id = S.shohin_id
   INNER JOIN ZaikoShohin AS ZS
     ON TS.shohin_id = ZS.shohin_id
```

通过 ON 子句指定联结条件的方式也没有发生改变。使用等号将作为联结条件的 Shohin 表和 TenpoShohin 表中的商品编号 (shohin\_id) 联结起来。由于 Shohin 表和 TenpoShohin 表已经进行了联结, 所以这里无需再对 Shohin 表和 ZaikoShohin 表进行联结了 (虽然也可以进行联结, 但结果并不会发生改变)。

即使想要把联结的表增加到 4 张、5 张……, 使用 INNER JOIN 进行添加的方式也是完全相同的。

## 交叉联结——CROSS JOIN

### KEYWORD

●交叉联结 (CROSS JOIN)

接下来, 和大家一起学习第 3 种联结方式——交叉联结。其实这种联结在实际业务中并没有使用过 (笔者使用这种联结的次数也屈指可数)。那为什么还要在这里进行介绍呢? 这是因为交叉联结是所有联结运算的基础。

交叉联结本身非常简单, 但是其结果有点麻烦。下面我们就试着将 Shohin 表和 TenpoShohin 表进行交叉联结 (代码清单 7-15)。

代码清单 7-15 将两张表进行交叉联结

```
SQL Server DB2 PostgreSQL MySQL
SELECT TS.tenpo_id, TS.tenpo_mei, TS.shohin_id, S.shohin_mei
FROM TenpoShohin AS TS CROSS JOIN Shohin AS S; ①
```

### 特定的 SQL

在 Oracle 中执行代码清单 7-15 时, 请将①的部分变为 "FROM TenpoShohin TS CROSS JOIN Shohin S;" (删除掉 FROM 子句中的 AS S)。



## 执行结果

tenpo_id	tenpo_mei	shohin_id	shohin_mei
000A	东京	0001	T恤衫
000A	东京	0002	T恤衫
000A	东京	0003	T恤衫
000B	名古屋	0002	T恤衫
000B	名古屋	0003	T恤衫
000B	名古屋	0004	T恤衫
000B	名古屋	0006	T恤衫
000B	名古屋	0007	T恤衫
000C	大阪	0003	T恤衫
000C	大阪	0004	T恤衫
000C	大阪	0006	T恤衫
000C	大阪	0007	T恤衫
000D	福冈	0001	T恤衫
000A	东京	0001	打孔器
000A	东京	0002	打孔器
000A	东京	0003	打孔器
000B	名古屋	0002	打孔器
000B	名古屋	0003	打孔器
000B	名古屋	0004	打孔器
000B	名古屋	0006	打孔器
000B	名古屋	0007	打孔器
000C	大阪	0003	打孔器
000C	大阪	0004	打孔器
000C	大阪	0006	打孔器
000C	大阪	0007	打孔器
000D	福冈	0001	打孔器
000A	东京	0001	运动T恤
000A	东京	0002	运动T恤
000A	东京	0003	运动T恤
000B	名古屋	0002	运动T恤
000B	名古屋	0003	运动T恤
000B	名古屋	0004	运动T恤
000B	名古屋	0006	运动T恤
000B	名古屋	0007	运动T恤
000C	大阪	0003	运动T恤
000C	大阪	0004	运动T恤
000C	大阪	0006	运动T恤
000C	大阪	0007	运动T恤
000D	福冈	0001	运动T恤
000A	东京	0001	菜刀
000A	东京	0002	菜刀
000A	东京	0003	菜刀
000B	名古屋	0002	菜刀
000B	名古屋	0003	菜刀
000B	名古屋	0004	菜刀
000B	名古屋	0006	菜刀
000B	名古屋	0007	菜刀
000C	大阪	0003	菜刀
000C	大阪	0004	菜刀
000C	大阪	0006	菜刀
000C	大阪	0007	菜刀
000D	福冈	0001	菜刀

000A	东京	0001	高压锅
000A	东京	0002	高压锅
000A	东京	0003	高压锅
000B	名古屋	0002	高压锅
000B	名古屋	0003	高压锅
000B	名古屋	0004	高压锅
000B	名古屋	0006	高压锅
000B	名古屋	0007	高压锅
000C	大阪	0003	高压锅
000C	大阪	0004	高压锅
000C	大阪	0006	高压锅
000C	大阪	0007	高压锅
000D	福冈	0001	高压锅
000A	东京	0001	叉子
000A	东京	0002	叉子
000A	东京	0003	叉子
000B	名古屋	0002	叉子
000B	名古屋	0003	叉子
000B	名古屋	0004	叉子
000B	名古屋	0006	叉子
000B	名古屋	0007	叉子
000C	大阪	0003	叉子
000C	大阪	0004	叉子
000C	大阪	0006	叉子
000C	大阪	0007	叉子
000D	福冈	0001	叉子
000A	东京	0001	擦菜板
000A	东京	0002	擦菜板
000A	东京	0003	擦菜板
000B	名古屋	0002	擦菜板
000B	名古屋	0003	擦菜板
000B	名古屋	0004	擦菜板
000B	名古屋	0006	擦菜板
000B	名古屋	0007	擦菜板
000C	大阪	0003	擦菜板
000C	大阪	0004	擦菜板
000C	大阪	0006	擦菜板
000C	大阪	0007	擦菜板
000D	福冈	0001	擦菜板
000A	东京	0001	圆珠笔
000A	东京	0002	圆珠笔
000A	东京	0003	圆珠笔
000B	名古屋	0002	圆珠笔
000B	名古屋	0003	圆珠笔
000B	名古屋	0004	圆珠笔
000B	名古屋	0006	圆珠笔
000B	名古屋	0007	圆珠笔
000C	大阪	0003	圆珠笔
000C	大阪	0004	圆珠笔
000C	大阪	0006	圆珠笔
000C	大阪	0007	圆珠笔
000D	福冈	0001	圆珠笔

**KEYWORD**

● CROSS JOIN(笛卡儿积)

可能大家会惊讶于结果的行数,但我们还是先来介绍一下语法结构吧。对满足相同规定的表进行交叉联结的集合运算符是 CROSS JOIN(笛卡儿积)。进行交叉联结时无法使用内联结和外联结中所使用的 ON 子句,这是因为交叉联结是对两张表中的全部记录进行交叉组合,因此结果中的记录数通常是两张表中行数的乘积。前例中,由于 TenpoShohin 表存在 13 条记录,Shohin 表存在 8 条记录,所以结果中就包含了  $13 \times 8 = 104$  条记录。

可能这时会有读者想起在前一节的最后,我们提到了集合运算中的乘法会在本节中进行详细学习,这就是上面介绍的交叉联结。

内联结是交叉联结的一部分。“内部”也可以理解为“包含在交叉联结结果中的部分”。相反,外联结可以理解为“交叉联结结果之外的部分”。

交叉联结没有应用到实际业务之中的原因有两个。一是其结果没有实用价值,二是由于其结果行数太多,需要花费大量的运算时间和高性能设备的支持。

## 特定的联结语句和过时的语法

之前我们学习的内联结和外联结的语法都符合标准 SQL 的规定,可以在所有 DBMS 中执行。因此,大家可以放心使用。但是如果大家之后从事系统开发工作的话,一定会碰到需要阅读他人写的代码并进行维护的情况。而那些使用特定和过时语法的程序就会成为我们的麻烦。

SQL 是一门特定及过时语法非常多的语言,虽然之前本书中也多次提及,但联结是其中特定语法的部分,现在还有不少年长的程序员和系统工程师仍在使用这些特定的语法。

例如,将本节最初介绍的内联结的 SELECT 语句(代码清单 7-9)替换为过时语法的结果如下所示(代码清单 7-16)。

代码清单 7-16 使用过是语法的内联结 (结果与代码清单 7-9 相同)

```
SELECT TS.tempo_id, TS.tempo_mei, TS.shohin_id, S.shohin_mei, →
S.hanbai_tanka
FROM TenpoShohin TS, Shohin S
WHERE TS.shohin_id = S.shohin_id
AND TS.tempo_id = '000A';
```

→表示下一行接续本行, 只是由于版面所限而换行。

这样的书写方式所得到的结果与标准语法完全相同。并且, 这样的语法可以在所有的 DBMS 中执行, 并不能算是特定的语法, 只是过时了而已。

但是, 由于这样的语法不仅过时, 而且还存在很多其他的问题, 所以不推荐大家使用。理由主要有以下 3 点。

第一, 使用这样的语法无法马上判断出到底是内联结还是外联结 (又或者 是其他种类的联结)。

第二, 由于联结条件都写在 WHERE 子句之中, 所以无法在短时间内分辨出哪部分是联结条件, 哪部分是用来选取记录的限制条件。

第三, 就是我们不知道这样的语法到底还能使用多久。每个 DBMS 的开发者都会考虑放弃过时的语法, 转而支持新的语法。虽然并不是马上就不能使用了, 但那一天总会到来的。

虽然这么说, 但是现在使用这些过时语法编写的程序还有很多, 到目前为止还都能正常执行。我想大家很可能会碰到这样的代码, 因此还是希望大家能够了解这些知识。



#### 法则 7-7

那些过时和特定的联结书写方式, 虽然我们自己不会使用, 但还是希望大家能够读懂。

## 专 栏

## 关系除法

本章中我们学习了以下4个集合运算符。

- UNION (并集)
- EXCEPT (差集)
- INTERSECT (交集)
- CROSS JOIN (笛卡儿积)

虽然交集是独立的一种集合运算,但实际上它是“只包含公共部分的特殊UNION”。剩下的3个在四则运算中也有对应的运算。但是,除法运算还没有介绍。

难道集合运算中没有除法吗?当然不是,除法运算是存在的。集合运算中的除法通常称为“关系除法”。关系是数学领域中对表或者试图的称谓。但是并没有定义像UNION或者EXCEPT这样专用的运算符。如果要定义,估计应该是DIVIDE(除)吧。但截至目前并没有DBMS使用这样的运算符。

为什么只有除法运算不使用运算符(只有除法)对被除数进行运算呢?其中的理由有点复杂,还是让我们先来介绍一下“表的除法”具体是一种什么样的运算吧。

我们使用表7-A和表7-B两张表作为示例用表。

表7-A Skills(技术)表:关系除法中的除数

skill
Oracle
UNIX
Java

表7-B EmpSkills(员工技术)表:关系除法中的被除数

emp	skill
相田	Oracle
相田	UNIX
相田	Java
相田	C#
神崎	Oracle
神崎	UNIX
神崎	Java
平井	UNIX
平井	Oracle
平井	PHP
平井	Perl
平井	C++
若田部	Perl
渡来	Oracle

## KEYWORD

- 关系除法

创建上述两张表并向其中插入数据的 SQL 语句请参考代码清单 7-A。

代码清单 7-A 创建 Skills/EmpSkills 表并插入数据

```
-- DDL: 创建表
CREATE TABLE Skills
(skill VARCHAR(32),
 PRIMARY KEY(skill));

CREATE TABLE EmpSkills
(emp VARCHAR(32),
 skill VARCHAR(32),
 PRIMARY KEY(emp, skill));

SQL Server, PostgreSQL
-- DML: 插入数据
BEGIN TRANSACTION; ①

INSERT INTO Skills VALUES('Oracle');
INSERT INTO Skills VALUES('UNIX');
INSERT INTO Skills VALUES('Java');

INSERT INTO EmpSkills VALUES('相田', 'Oracle');
INSERT INTO EmpSkills VALUES('相田', 'UNIX');
INSERT INTO EmpSkills VALUES('相田', 'Java');
INSERT INTO EmpSkills VALUES('相田', 'C#');
INSERT INTO EmpSkills VALUES('神崎', 'Oracle');
INSERT INTO EmpSkills VALUES('神崎', 'UNIX');
INSERT INTO EmpSkills VALUES('神崎', 'Java');
INSERT INTO EmpSkills VALUES('平井', 'UNIX');
INSERT INTO EmpSkills VALUES('平井', 'Oracle');
INSERT INTO EmpSkills VALUES('平井', 'PHP');
INSERT INTO EmpSkills VALUES('平井', 'Perl');
INSERT INTO EmpSkills VALUES('平井', 'C++');
INSERT INTO EmpSkills VALUES('若田部', 'Perl');
INSERT INTO EmpSkills VALUES('渡来', 'Oracle');

COMMIT;
```

#### 特定的 SQL

不同的 DBMS, 事务处理的语法也不尽相同。代码清单 7-A 中的 DML 语句在 MySQL 中执行时, 需要将①部分更改为“START TRANSACTION;”。在 Oracle 和 DB2 中执行时, 无需用到①的部分(请删除)。

详细内容请大家参考第 4 章中的“创建事务”。

EmpSkills 表中保存了某个系统公司员工所掌握的技术信息。例如, 从该表中我们可以了解到相田掌握了 Oracle、UNIX、Java、C# 这 4 种技术。

下面我们来思考一下如何从该表中选取出了掌握了 Skills 表中所有 3 个领域技术的员工吧(代码清单 7-B)。

## 代码清单 7-8 选出掌握所有 3 个领域技术的员工

```

SELECT DISTINCT emp
FROM EmpSkills ES1
WHERE NOT EXISTS
  (SELECT skill
   FROM Skills
  EXCEPT
  SELECT skill
   FROM EmpSkills ES2
  WHERE ES1.emp = ES2.emp);

```

这样我们就得到了包含相田和神崎 2 人的结果。虽然平井也掌握了 Oracle 和 UNIX, 但很可惜他不会使用 Java, 因此没有选取出。

## 执行结果(关系除法中的商)

```

emp
-----
神崎
相田

```

这样的结果满足了除法运算的基本规则。肯定有读者会产生这样的疑问:“到底上述运算中什么地方是除法运算呢?”实际上这和数值的除法既相似又有所不同。大家从与乘法相对的乘法运算的角度去思考就能得到答案了。

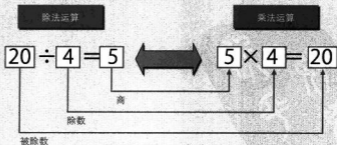
除法和乘法是相辅相成的关系, 除法运算的结果(商)乘以除数就能得到除法运算前的被除数了。例如对于  $20 \div 4 = 5$  来说, 就是  $5(\text{商}) \times 4(\text{除数}) = 20(\text{被除数})$  (图 7-A)。

关系除法中, 这样的规则也是成立的。商和除数相乘, 也就是交叉联结, 就能够得到作为被除数的集合了<sup>①</sup>。

①

虽然不能恢复成完整的被除数, 但是这里我们也不再追究了。

图 7-A 除法运算和乘法运算相辅相成的关系图



如上所述, 除法运算是集合运算中最复杂的运算。但是其在实际业务中的应用十分广泛, 因此希望大家能在达到中级以上水平时掌握其使用方法。此外, 想要详细了解 SQL 中除法运算实现方法的读者, 可以参考拙著《深入浅出 SQL 徹底指南書》(翔泳社出版)中的“1-4 HAVING 子句的力量”和“1-7 SQL 中的集合运算”。

## 练习题

7.1 请说出下述 SELECT 语句的结果。

```
-- 使用本章中的Shohin表
SELECT *
  FROM Shohin
UNION
SELECT *
  FROM Shohin
INTERSECT
SELECT *
  FROM Shohin
ORDER BY shohin_id;
```

7.2 7-2 节代码清单 7-11 中列举的外部链接的结果中，高压锅和圆珠笔 2 条记录的商店编号 (tenpo\_id) 和商店名称 (tenpo\_mei) 都是 NULL。请使用字符串“不明”替换其中的 NULL。期望结果如下所示。

执行结果

tenpo_id	tenpo_mei	shohin_id	shohin_mei	hanbai_tanka
000A	东京	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000A	东京	0001	T恤衫	1000
000B	名古屋	0006	叉子	500
000B	名古屋	0002	打孔器	500
000B	名古屋	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000B	名古屋	0007	擦菜板	880
000C	大阪	0006	叉子	500
000C	大阪	0007	擦菜板	880
000C	大阪	0003	运动T恤	4000
000C	大阪	0004	菜刀	3000
000D	福冈	0001	T恤衫	1000
不明	不明	0005	高压锅	6800
不明	不明	0008	圆珠笔	100

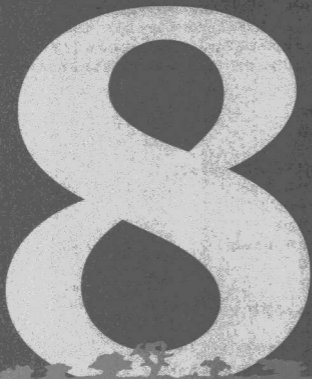
将商店编号和商店名称输出为“不确定”



# 第8章 SQL高级处理

窗口函数

GROUPING 运算符



SQL

## 本章重点

---

终于到了最后一章，我们要学习的是 SQL 中的高级聚合处理。即使是“高级处理”，说到底也还是在 SQL 中能够执行的处理。从用户的角度来说，就是那些对数值进行排序，计算销售总额等我们熟悉的处理。

和自然语言一样，SQL 语言也会随着时间而不断变化，现在每隔几年就会对标准 SQL 进行功能追加和语法修正。本章将要介绍的是最近才添加的功能。掌握了这些方便的新功能，使用 SQL 能够完成的工作范围也会不断扩展。

### 8-1 窗口函数

- 什么是窗口函数
- 窗口函数的语法
- 语法的基本使用方法——使用 RANK 函数
- 无须指定 PARTITION BY
- 专用窗口函数的种类
- 窗口函数的适用范围
- 作为窗口函数使用的聚合函数
- 计算移动平均数
- 两个 ORDER BY

### 8-2 GROUPING 运算符

- 同时计算出合计值
- ROLLUP——同时计算出合计值和小计值
- GROUPING 函数——让 NULL 更加容易分辨
- CUBE——用数据来搭积木
- GROUPING SETS——取得期望的积木

## 8-1

## 窗口函数

## 学习重点

- 窗口函数可以进行排序、生成序列号等一般的聚合函数无法实现的高级操作。
- 理解PARTITION BY和ORDER BY这两个关键字的含义十分重要。

## KEYWORD

- 窗口函数
- OLAP函数

## 注①

在Oracle中称为分析函数。

## KEYWORD

- OLAP

## 注②

目前MySQL还不支持窗口函数。详细信息请参考专栏“窗口函数的支持情况”。

## 注③

随着时间推移，标准SQL终将能够在所有的DBMS中进行使用。

## 什么是窗口函数

窗口函数也称为OLAP函数<sup>①</sup>。为了让大家快速形成直观印象，才起了这样一个容易理解的名称（“窗口”的含义我们将在随后进行说明）。

OLAP是OnLine Analytical Processing的简称，意为对数据库数据进行实时分析处理。例如，市场分析、创建财务报表、创建计划等日常性商务工作。

窗口函数就是为了实现OLAP而添加的标准SQL功能<sup>②</sup>。

## 专 栏

## 窗口函数的支持情况

很多数据库相关工作过去都会有这样的想法：“好不容易将业务数据插入了数据库，如果能够使用SQL对其进行实时分析的话，一定会很方便吧。”但是关系数据库提供支持OLAP用途的功能仅仅只有10年左右的时间。

其中的理由有很多，这里我们就不一一介绍了。大家需要注意的是，还有一部分DBMS并不支持这样的新功能。

本节将要介绍的窗口函数也是其中之一，截至2010年5月，Oracle 11g、SQL Server 2008、DB2 9.7、PostgreSQL 8.4都已经支持了该功能，但是MySQL 5.5还是不支持该功能。

通过前面的学习，我们已经知道各个DBMS都有自己支持的特定语法和不支持的语法。标准SQL添加新功能的时候也会遇到同样的问题<sup>③</sup>。

## 窗口函数的语法

接下来，就让我们通过示例来学习窗口函数吧。窗口函数的语法有些复杂。

### 语法 8-1 窗口函数

```
<窗口函数> OVER ([PARTITION BY <列清单>]
                 ORDER BY <排序用列清单>)
```

\*[] 中的内容可以省略

其中重要的关键字是 PARTITION BY 和 ORDER BY。首先，理解这两个关键字的作用是帮助我们理解窗口函数的关键。

### ■能够作为窗口函数使用的函数

在学习 PARTITION BY 和 ORDER BY 之前，我们先来列举一下能够作为窗口函数使用的函数。窗口函数大体可以分为以下 2 种。

- ①能够作为窗口函数的聚合函数 (SUM、AVG、COUNT、MAX、MIN)
- ②RANK、DENSE\_RANK、ROW\_NUMBER 等专用窗口函数

②中的函数是标准 SQL 定义的 OLAP 专用函数。本书将其统称为“专用窗口函数”。从这些函数的名称可以很容易看出其 OLAP 的用途。

其中①的部分是我们在第 3 章中学过的聚合函数。通过将聚合函数书写在“语法 8-1”的 <窗口函数> 中，就能够作为窗口函数来使用了。总之，集约函数根据使用语法的不同，可以在集约函数和窗口函数之间进行转换。

## 语法的基本使用方法——使用 RANK 函数

首先让我们通过专用窗口函数 RANK 来理解一下窗口函数的语法吧。正如其名称所示，RANK 是用来计算记录排序的函数。

例如，我们将各种类的商品 (shohin\_bunrui)，按照销售单价 (hanbai\_tanka) 从低到高的顺序，制作出之前曾使用过的 Shohin 表中的 8 件商品的排序表。结果如下所示。

### KEYWORD

●专用窗口函数

### KEYWORD

●RANK 函数

## 执行结果

shohin_mei	shohin_bunrui	hanbai_tanka	ranking
叉子	厨房用具	500	1
煎菜板	厨房用具	880	2
菜刀	厨房用具	3000	3
高压锅	厨房用具	6800	4
T恤衫	衣服	1000	1
运动T恤	衣服	4000	2
圆珠笔	办公用品	100	1
打孔器	办公用品	500	2

以厨房用具为例，销售单价最便宜的“叉子”排在第1位，最贵的“高压锅”排在第4位，确实按照我们的要求进行了排序。

能够得到上述结果的 SELECT 语句请参考代码清单 8-1。

代码清单 8-1 将各种类的商品，按照销售单价从低到高的顺序创建排序表

```

Oracle SQL Server DB2 PostgreSQL
SELECT shohin_mei, shohin_bunrui, hanbai_tanka,
       RANK () OVER (PARTITION BY shohin_bunrui
                    ORDER BY hanbai_tanka) AS ranking
FROM Shohin;

```

## KEYWORD

- PARTITION BY 子句
- ORDER BY 子句

## 0

其所遵循的规则与 SELECT 语句末尾的 ORDER BY 子句完全相同。

PARTITION BY 能够设定排序的对象范围。本例中，为了按照商品种类进行排序，我们指定了 shohin\_bunrui。

ORDER BY 能够指定按照哪一列、何种顺序进行排序。为了按照销售单价的升序进行排列，我们指定了 hanbai\_tanka。此外，窗口函数中的 ORDER BY 与 SELECT 语句末尾的 ORDER BY 一样，可以通过关键字 ASC/DESC 来指定升序和降序。省略该关键字时会默认按照 ASC，也就是升序进行排序。本例中就省略了上述关键字<sup>①</sup>。

通过图 8-1，我们就很容易理解 PARTITION BY 和 ORDER BY 的作用了。如图所示，PARTITION BY 对表的横向进行分组，而 ORDER BY 决定了纵向排序的规则。

图8-1 PARTITION BY和ORDER BY的作用

通过PARTITION BY可以将分组后记录的集合以窗口形式呼出

ORDER BY的顺序 (销售单价由低到高的顺序)

shohin_id (商品编号)	shohin_mei (商品名称)	shohin_bunrui (商品分类)	hanbai_tanka (销售单价)	shire_tanka (进货单价)	torokubi (登记日期)
0006	叉子	厨房用具	500		2009-09-20
0007	煎菜板	厨房用具	880	790	2008-04-28
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0001	T恤衫	衣服	1000	500	2009-09-20
0003	运动T恤	衣服	4000	2800	
0008	圆珠笔	办公用品	100		2009-11-11
0002	打孔器	办公用品	500	320	2009-09-11

PARTITION BY的分组 (根据商品种类)

窗口函数兼具之前我们学过的 GROUP BY 子句的分组功能，以及 ORDER BY 子句的排序功能。但是，PARTITION BY 子句并不具备 GROUP BY 子句的聚合功能。因此，使用 RANK 函数并不会减少元表中记录的行数，结果中仍然包含 8 行数据。



#### 法则 8-1

窗口函数兼具分组和排序两种功能。

#### KEYWORD

● 窗口

#### ①

从词语意思的角度考虑，可能“组”比“窗口”更合适一些，但是在 SQL 中，“组”更多的情况是用来特指使用 GROUP BY 分割后的记录集合。因此，为了避免混淆，使用 PARTITION BY 是称为窗口。



#### 法则 8-2

通过 PARTITION BY 分组后的记录集合称为“窗口”。

此外，各个窗口在定义上绝对不会包含共通的部分。就像刀切蛋糕一

样，干净利落。这与通过 GROUP BY 子句分割后的集合具有相同的特征。

## 无须指定 PARTITION BY

使用窗口函数时起到关键作用的是 PARTITION BY 和 GROUP BY。其中，PARTITION BY 并不是必须的，即使不指定也可以正常使用窗口函数。

那么就让我们来确认一下不指定 PARTITION BY 会得到什么样的结果吧。这和使用没有 GROUP BY 的聚合函数时的效果一样，也就是将整个表作为一个大的窗口来使用。

事实胜于雄辩，下面就让我们删除代码清单 8-1 中 SELECT 语句的 PARTITION BY 试试吧（代码清单）。

代码清单 8-2 不指定 PARTITION BY

```

Oracle | SQL Server | DB2 | PostgreSQL
SELECT shohin_mei, shohin_bunrui, hanbai_tanka,
       RANK () OVER (ORDER BY hanbai_tanka) AS ranking
FROM Shohin;

```

上述 SELECT 语句的结果如下所示。

执行结果

shohin_mei	shohin_bunrui	hanbai_tanka	ranking
圆珠笔	办公用品	100	1
叉子	厨房用具	500	2
打孔器	办公用品	500	2
擦菜板	厨房用具	880	4
T恤衫	衣服	1000	5
菜刀	厨房用具	3000	6
运动T恤	衣服	4000	7
高压锅	厨房用具	6800	8

之前我们得到的是按照商品种类分组后的排序，而这次变成了全部商品的排序。PARTITION BY 可以将表中的数据分为多个部分（窗口），是希望使用窗口函数时的一个选项。

## 专用窗口函数的种类

从上述结果中我们可以看到，“打孔器”和“叉子”都排在第2位，而之后的“擦菜板”跳过了第3位，直接排到了第4位。这也是通常的排序方法。但某些情况下可能并不希望跳过某个位次来进行排序。

这时可以使用 RANK 函数之外的函数来实现。下面就让我们来总结一下具有代表性的专用窗口函数吧。

### KEYWORD

- RANK 函数
- DENSE\_RANK 函数
- ROW\_NUMBER 函数

#### ● RANK 函数

计算排序时，如果存在相同位次的记录，则会跳过之后的位次。

例) 有3条记录排在第1位时: 1位、1位、1位、4位……

#### ● DENSE\_RANK 函数

同样是计算排序，即使存在相同位次的记录，也不会跳过之后的位次。

例) 有3条记录排在第1位时: 1位、1位、1位、2位……

#### ● ROW\_NUMBER 函数

赋予唯一的连续位次。

例) 有3条记录排在第1位时: 1位、2位、3位、4位……

除此之外，各 DBMS 还提供了各自特有的窗口函数。上述3个函数(对于支持窗口函数的 DBMS 来说)在所有的 DBMS 中都能够使用。下面就让我们来比较一下使用这3个函数所得到的结果吧(代码清单 8-3)。

代码清单 8-3 比较 RANK、DENSE\_RANK、ROW\_NUMBER 的结果

Oracl	SQL Server	DB2	PostgreSQL
<pre> SELECT shohin_mei, shohin_bunrui, hanbai_tanka,        RANK () OVER (ORDER BY hanbai_tanka) AS ranking,        DENSE_RANK () OVER (ORDER BY hanbai_tanka) AS dense_ranking,        ROW_NUMBER () OVER (ORDER BY hanbai_tanka) AS row_num FROM Shohin; </pre>			



执行结果

			RANK	DENSE_RANK	ROW_NUMBER
shohin_mei	shohin_bunrui	hanbai_tanka	ranking	dense_ranking	row_num
圆珠笔	办公用品	100	1	1	1
叉子	厨房用具	500	2	2	2
打孔器	办公用品	500	2	2	3
擦菜板	厨房用具	880	4	3	4
T恤衫	衣服	1000	5	4	5
菜刀	厨房用具	3000	6	5	6
运动T恤	衣服	4000	7	6	7
高压锅	厨房用具	6800	8	7	8

将结果中的 `ranking` 列和 `dense_ranking` 列进行比较可以发现, `dense_ranking` 列中有连续 2 个第 2 位, 这和 `ranking` 列的情况相同。但是接下来的“擦菜板”的位次并不是第 4 而是第 3。这就是使用 `DENSE_RANK` 函数的效果了。

此外, 我们可以看到, 在 `row_num` 列中, 不管销售单价 (`hanbai_tanka`) 是否相同, 每件商品都会按照销售单价从低到高的顺序得到一个连续的位次。销售单价相同时, DBMS 会根据适当的顺序对记录进行排列。想为记录赋予唯一的连续位次时, 就可以像这样使用 `ROW_NUMBER` 来实现。

使用 `RANK` 或 `ROW_NUMBER` 时无需任何参数, 只需要像 `RANK ()` 或者 `ROW_NUMBER ()` 这样保持括号中为空就可以了。这也是专用窗口函数通常的使用方式, 请大家牢记。这一点与作为窗口函数使用的聚合函数有很大的不同, 之后我们将会详细介绍。



### 法则 8-3

由于专用窗口函数无需参数, 所以通常括号中都是空的。

## 窗口函数的适用范围

目前为止我们学过的函数大部分都没有使用位置的限制。最多也就是在 `WHERE` 子句中使用聚合函数时会有些注意事项。但是, 使用窗口函数的位置却有非常大的限制。更确切地说, 窗口函数只能书写在一个特定的位置。

这个位置就是 `SELECT` 子句之中。反过来说, 就是这类函数不能在

## ①

语法上,除了 SELECT 子句, ORDER BY 子句或者 UPDATE 语句的 SET 子句中也可以使用。但由于几乎没有实际的业务示例,所以开始的时候大家只要记得“只能在 SELECT 子句中使用”就可以了。

WHERE 子句或者 GROUP BY 子句中使用。<sup>①</sup>



## 法则 8-4

原则上窗口函数只能在 SELECT 子句中使用。

## ②

反之,之所以在 ORDER BY 子句中能够使用窗口函数,是因为 ORDER BY 子句会在 SELECT 子句之后执行,并且记录保证不会减少。

虽然我们可以把它当作一种规则死记硬背下来,但是为什么窗口函数只能在 SELECT 子句中使用呢?(也就是不能在 WHERE 子句或者 GROUP BY 子句中使用。)下面我们就来简单说明一下其中的理由。

其理由就是,在 DBMS 内部,窗口函数是对 WHERE 子句或者 GROUP BY 子句处理后的“结果”进行的操作。大家仔细想一想就会明白,在得到用户想要的结果之前,即使进行了排序处理,结果也是错误的。在得到排序结果之后,如果通过 WHERE 子句中的条件去除了某些记录,或者使用 GROUP BY 子句进行了聚合处理,那好不容易得到的排序结果也无法使用的了。<sup>②</sup>

正是由于这样的原因,在 SELECT 子句之外“使用窗口函数是没有意义的”,所以在语法上才会有这样的限制。

## 作为窗口函数使用的聚合函数

前面给大家介绍了使用专用窗口函数的示例。下面我们再来看一下如何把之前学过的 SUM 或者 AVG 等聚合函数作为窗口函数的使用方法。

所有的聚合函数都能用作窗口函数,其语法和专用窗口函数完全相同。但由于大家可能对于所能得到的结果还没有一个直观的印象,所以我们还是通过具体的示例来进行学习。下面我们先来看一个将 SUM 函数作为窗口函数使用的例子(代码清单 8-4)。

代码清单 8-4 将 SUM 函数作为窗口函数使用<sup>③</sup>

Oracle	DB2	PostgreSQL
SELECT shohin_id, shohin_mei, hanbai_tanka,		
SUM (hanbai_tanka) OVER (ORDER BY shohin_id) AS current_sum		
FROM Shohin;		

在 SQL Server 中,聚合函数和 OVER 子句组合使用时,无法使用 ORDER BY 子句。由于存在这样的限制,所以本例中的代码无法执行。Oracle、DB2、PostgreSQL 中并没有这样的限制。

《SQL Server 2008 在线手册》(2009 年 7 月) OVER 子句(Transact-SQL)  
<http://msdn.microsoft.com/ja-jp/library/ms189461.aspx>

## 执行结果

shohin_id	shohin_mei	hanbai_tanka	current_sum	
0001	丁恤衫	1000	1000	--1000
0002	打孔器	500	1500	--1000+500
0003	运动T恤	4000	5500	--1000+500+5500
0004	菜刀	3000	8500	--1000+500+5500+3000
0005	高压锅	6800	15300	.
0006	叉子	500	15800	.
0007	磨菜板	880	16680	.
0008	圆珠笔	100	16780	.

使用 SUM 函数时，并不像 RANK 或者 ROW\_NUMBER 那样括号中的内容为空。而是和之前我们学过的一样，需要在括号内指定作为聚合对象的列。本例中我们计算出了销售单价(hanbai\_tanka)的合计值(current\_sum)。

但是我们得到的并不仅仅是合计值。而是按照 ORDER BY 子句指定的 shohin\_id 的升序进行排列，计算出商品编号“小于自己”的商品的销售单价的合计值。因此，计算该合计值的逻辑就像金字塔堆积那样，一行一行逐渐添加计算对象。在计算随时增加的销售总额，特别是需要按照时间顺序计算时，通常都会使用这种称为“累计”的统计方法。

使用其他聚合函数时的操作逻辑也和本例相同。例如，使用 AVG 来代替 SELECT 语句中的 SUM（代码清单 8-5）。

代码清单 8-5 将 AVG 函数作为窗口函数使用<sup>①</sup>

```

Oracle DB2 PostgreSQL
SELECT shohin_id, shohin_mei, hanbai_tanka,
       AVG (hanbai_tanka) OVER (ORDER BY shohin_id) AS current_avg
FROM Shohin;

```

## 执行结果

shohin_id	shohin_mei	hanbai_tanka	current_avg	
0001	丁恤衫	1000	1000.0000000000000000	--(1000)/1
0002	打孔器	500	750.0000000000000000	--(1000+500)/2
0003	运动T恤	4000	2833.3333333333333333	--(1000+500+4000)/3
0004	菜刀	3000	2125.0000000000000000	--(1000+500+4000+3000)/4
0005	高压锅	6800	3060.0000000000000000	--(1000+500+4000+3000+6800)/5
0006	叉子	500	2633.3333333333333333	.
0007	磨菜板	880	2382.8571428571428571	.
0008	圆珠笔	100	2097.5000000000000000	.

## KEYWORD

● 累计

## ①

本例中的代码同样无法在 SQL Server 中执行。理由请参考前一注释。

从结果中我们可以看到，`current_avg`的计算方法，确实是计算平均值的方法。但作为统计对象的却只是“排在自己之上”的记录。像这样以“自身记录（当前记录）”作为基准进行统计，就是将聚合函数用作窗口函数使用时的最大的特征。

**KEYWORD**

● 当前记录

## 计算移动平均

窗口函数就是将表以窗口为单位进行分割，并在其中进行排序的函数。其实其中还包含在窗口中指定更加详细的统计范围的备选功能。该备选功能中的统计范围称为“框架”。

其语法如代码清单 8-6 所示，需要在 `ORDER BY` 子句之后使用指定范围的关键字<sup>①</sup>。

代码清单 8-6 指定“最靠近的3行”作为统计对象

```

Oracle DB2
SELECT shohin_id, shohin_mei, hanbai_tanka,
       AVG (hanbai_tanka) OVER (ORDER BY shohin_id
                               ROWS 2 PRECEDING) AS moving_avg
FROM Shohin;

```

执行结果（在DB2中执行）

shohin_id	shohin_mei	hanbai_tanka	moving_avg
0001	T恤衫	1000	1000 --(1000)/1
0002	打孔器	500	750 --(1000+500)/2
0003	运动T恤	4000	1833 --(1000+500+4000)/3
0004	菜刀	3000	2500 --(500+4000+3000)/3
0005	高压锅	6800	4600 --(4000+3000+6800)/3
0006	叉子	500	3433
0007	擦菜板	880	2726
0008	圆珠笔	100	493

### ● 指定框架（统计范围）

我们将上述结果与之前的结果进行比较，可以发现商品编号为“0004”的“菜刀”以下的记录和窗口函数的计算结果并不相同。这是因为我们指定了框架，将统计对象限定为“最靠近的3行”的缘故。

这里我们使用了 `ROWS`（行）和 `PRECEDING`（之前）两个关键字，将框架指定为“截止到之前~行”。因此“`ROWS 2 PRECEDING`”也就是

**KEYWORD**

- `ROWS` 关键字
- `PRECEDING` 关键字

**0**

目前只有 Oracle 和 DB2 支持此功能。

将框架指定为“截止到之前 2 行”，作为统计对象的记录就限定为如下的“最靠近的 3 行”。也就是说，由于框架是根据当前记录来决定的，所以和固定的窗口不同，其范围会随着当前记录的变化而变化。

- 自身 (当前记录)
- 之前 1 行的记录
- 之前 2 行的记录

图 8-2 将框架指定为截止到当前记录之前 2 行 (最靠近的 3 行)

ROWS 2 PRECEDING

shohin_id (商品编号)	shohin_mei (商品名称)	hanbai_tanka (销售单价)
0001	T 恤衫	1000
0002	打孔器	500
0003	运动T恤	4000
0004	菜刀	3000
0005	高压锅	6800
0006	叉子	500
0007	擦桌板	880
0008	圆珠笔	100

← 框架  
当前记录  
{ 自身=当前行 }

如果将条件中的数字变为“ROWS 5 PRECEDING”，就是“截止到之前 5 行”（最靠近的 6 行）的意思。

这样的统计方法称为移动平均 (moving average)。由于这种方法在希望实时把握“最近状态”时非常方便，所以常常会应用在对股市趋势的实时跟踪当中。

使用关键字 FOLLOWING (之后) 替换 PRECEDING，就可以指定“截止到之后 ~ 行”作为框架了 (图 8-3)。

#### KEYWORD

- 移动平均
- FOLLOWING 关键字

图8-3 将框架指定为截止到当前记录之后2行(最最近的3行)

ROWS 2 FOLLOWING

shohin_id (商品编号)	shohin_mei (商品名称)	hanbai_tanka (销售单价)
0001	T恤衫	1000
0002	打孔器	500
0003	运动T恤	4000
0004	菜刀	3000
0005	高压锅	6800
0006	叉子	500
0007	擦菜板	880
0008	圆珠笔	100

当前记录  
(自身=当前行)

← 框架

### ● 将当前记录的前后行作为统计对象

如果希望将当前记录的前后行作为统计对象时,就可以像代码清单8-7那样,同时使用 PRECEDING(之前)和 FOLLOWING(之后)关键字来实现。

#### 代码清单8-7 将当前记录的前后行作为统计对象

```

Oracle DB2
SELECT shohin_id, shohin_mei, hanbai_tanka,
       AVG(hanbai_tanka) OVER (ORDER BY shohin_id
                               ROWS BETWEEN 1 PRECEDING AND
                               1 FOLLOWING) AS moving_avg
FROM Shohin;

```

⇒表示下一行接续本行,只是由于版面所限而换行。

#### 执行结果(在DB2中执行)

shohin_id	shohin_mei	hanbai_tanka	moving_avg
0001	T恤衫	1000	750
0002	打孔器	500	1833
0003	运动T恤	4000	2500
0004	菜刀	3000	4600
0005	高压锅	6800	3433
0006	叉子	500	2726
0007	擦菜板	880	493
0008	圆珠笔	100	490

在上述代码中,我们通过指定框架,将“1 PRECEDING”(之前1行)和“1 FOLLOWING”(之后1行)的区间作为统计对象。具体来说就是将

如下 3 行作为统计对象来进行计算（图 8-4）。

- 之前 1 行的记录
- 自身（当前记录）
- 之后 1 行的记录

如果能够熟练掌握框架功能的话，就成为窗口函数高手了。

图 8-4 将框架指定为当前记录及其前后 1 行

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

shohin_id (商品编号)	shohin_mei (商品名称)	hanbai_tanka (销售单价)
0001	T 恤衫	1000
0002	打孔器	500
0003	运动T恤	4000
0004	菜刀	3000
0005	高压锅	6800
0006	叉子	500
0007	擦桌板	880
0008	圆珠笔	100

← 框架  
当前记录  
(自身=当前行)

## 两个 ORDER BY

最后我们来介绍一下使用窗口函数时关于结果形式的注意点，那就是记录的排列顺序。由于使用窗口函数时必须要在 OVER 子句中使用 ORDER BY，所以乍一看，可能有读者会觉得结果中的记录不会按照该 ORDER BY 所指定的顺序进行排序。

但其实这只是一种错觉。OVER 子句中的 ORDER BY 只是用来决定窗口函数按照什么样的顺序进行计算的，对结果的排列顺序并没有影响。因此也有可能像代码清单 8-8 那样，得到一个记录的排列顺序比较混乱的结果。有些 DBMS 也可以按照窗口函数的 ORDER BY 子句所指定的顺序对结果进行排序，但那也仅仅是个例而已。

代码清单 8-8 无法保证如下 SELECT 语句结果的排列顺序

```

Oracle | SQL Server | DB2 | PostgreSQL
SELECT shohin_mei, shohin_bunrui, hanbai_tanka,
       RANK () OVER (ORDER BY hanbai_tanka) AS ranking
FROM Shohin;

```

有可能会得到下面这样的结果

shohin_mei	shohin_bunrui	hanbai_tanka	ranking
菜刀	厨房用具	3000	6
打孔器	办公用品	500	2
运动T恤	衣服	4000	7
T恤衫	衣服	1000	5
高压锅	厨房用具	6800	8
叉子	厨房用具	500	2
煎菜板	厨房用具	880	4
圆珠笔	办公用品	100	1

那么，如何才能让记录切实按照 ranking 列的升序进行排列呢？

答案非常简单。那就是在 SELECT 语句的最后，使用 ORDER BY 子句进行指定（代码清单 8-9）。这样就能保证 SELECT 语句结果中记录的排列顺序了，除此之外也没有其他办法了。

代码清单 8-9 在语句末尾使用 ORDER BY 子句对结果进行排序

```

Oracle | SQL Server | DB2 | PostgreSQL
SELECT shohin_mei, shohin_bunrui, hanbai_tanka,
       RANK () OVER (ORDER BY hanbai_tanka) AS ranking
FROM Shohin
ORDER BY ranking;

```

也许大家会觉得在一条 SELECT 语句中使用两次 ORDER BY 会有点别扭，但是尽管这两个 ORDER BY 看上去是相同的，但其实它们的功能却完全不同。



#### 法则 8-5

将聚合函数作为窗口函数使用时，会以当前记录为基准来决定统计对象的记录。



## 8-2

第8章 SQL 高级处理

## GROUPING 运算符

## 学习重点

- 只使用 GROUP BY 子句和聚合函数是无法同时计算出小计、合计值的。如果想要一次得到这两个值可以使用 GROUPING 运算符。
- 理解 GROUPING 运算符中 CUBE 的关键在于形成“积木搭建出的立方体”的印象。
- 虽然 GROUPING 运算符是标准 SQL 的功能，但还是有些 DBMS 尚未支持这一功能。

## 同时计算出合计值

我们在 3-2 节中学习过 GROUP BY 子句和聚合函数的使用方法，当时可能有些读者会想，是否有办法能够通过 GROUP BY 子句得到表 8-1 那样的结果呢？

表 8-1 添加合计行

合计	16780	— 存在合计行
厨房用具	11180	
衣服	5000	
办公用品	600	

虽然这是按照商品种类计算销售单价的总额时得到的结果，但问题在于最上面多出了 1 行合计行。使用代码清单 8-10 中 GROUP BY 子句的语法无法得到这一行。

代码清单 8-10 使用 GROUP BY 无法得到合计行

```
SELECT shohin_bunrui, SUM(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

执行结果

```
shohin_bunrui | sum
-----|-----
衣服          | 5000
办公用品      | 600
厨房用具      | 11180
```

**KEYWORD**

● UNION ALL

**0**

虽然也可以使用UNION来代替UNION ALL,但由于两条SELECT语句的聚合键不同,一定不会出现重复行,所以可以使用UNION ALL。UNION ALL和UNION的不同之处在于它不会对结果进行排序,因此比UNION的性能更好。

由于GROUP BY子句是用来指定聚合键的场所,所以这里指定的键只能用来进行分割数据。而合计行是不指定聚合键时得到的聚合结果,因此与之下3行通过聚合键得到的结果并不相同。按照通常的思路,想一次得到这两种结果是不可能的。

如果想要获得那样的结果,过去通常会分别计算出合计行和按照商品种类进行聚合的结果,然后通过UNION ALL<sup>①</sup>将它们连接在一起(代码清单8-11)。

代码清单8-11 分别计算出合计行和聚合结果再通过UNION ALL进行连接

```
SELECT '合计' AS shohin_bunrui, SUM(hanbai_tanka)
FROM Shohin
UNION ALL
SELECT shohin_bunrui, SUM(hanbai_tanka)
FROM Shohin
GROUP BY shohin_bunrui;
```

**执行结果**

shohin_bunrui	sum
合计	16780
衣服	5000
办公用品	600
厨房用具	11180

这样一来,为了得到想要的结果,需要执行2次几乎相同的SELECT语句,再将其结果进行连接。不但看上去十分繁琐,DBMS内部的处理成本也非常高。难道没有更合适的实现方法了吗?

**ROLLUP——同时计算出合计值和小计值**

为了满足用户的需求,标准SQL引入了GROUPING运算符。我们将在本节中着重介绍。使用该运算符就能通过非常简单的SQL,得到之前那样聚合单位不同的聚合结果了。

GROUPING运算符包含以下3种<sup>①</sup>。

- ROLLUP
- CUBE
- GROUPING SETS

**KEYWORD**

● GROUPING运算符

**0**

目前IngreSQL和MySQL并不支持GROUPING运算符。MySQL仅支持ROLLUP。具体内容请参考专栏“GROUPING运算符的支持状况”。

**KEYWORD**

● ROLLUP 运算符

**■ ROLLUP 的使用方法**

我们先从 ROLLUP 开始学习吧。使用 ROLLUP 就可以通过非常简单的 SELECT 语句同时计算出合计行了（代码清单 8-12）。

代码清单 8-12 使用 ROLLUP 同时计算出合计行和小计

```

Oracle  SQL Server  DB2
SELECT shohin_bunrui, SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY ROLLUP(shohin_bunrui); ——①

```

**特定的 SQL**

在 MySQL 中执行代码清单 8-12 时，请将①中的 GROUP BY 子句改写为“GROUP BY shohin\_bunrui WITH ROLLUP;”。

**执行结果（在 DB2 中执行）**

shohin_bunrui	sum_tanka
-----	
	16780
厨房用具	11180
办公用品	600
衣服	5000

在语法上，就是将 GROUP BY 子句中的聚合键清单像 ROLLUP (< 列 1>, < 列 2>, ...) 这样进行使用。该运算符的作用，一言以蔽之，就是“一次计算出不同聚合键的组合结果”。例如，在本例中就是一次计算出了如下 2 组组合的聚合结果。

- ① GROUP BY ()
- ② GROUP BY (shohin\_bunrui)

①中的 GROUP BY () 并没有聚合键，也就相当于没有 GROUP BY 子句（这时会得到全部数据的合计行的记录）。该合计行记录称为超级分组记录（super group row）。虽然名字听上去很炫，但还是希望大家把它当作未使用 GROUP BY 的合计行来理解。超级分组记录的 shohin\_bunrui 列的键值（对 DBMS 来说）并不明确，因此会默认使用 NULL。之后会为大家讲解在此处插入恰当的字符串的方法。

**KEYWORD**

● 超级分组记录



## 法则 8-6

超级分组记录默认使用NULL作为聚合键。

### ■将“登记日期”添加到聚合键当中

仅仅通过刚才一个例子大家的印象可能不够深刻，下面让我们再添加一个聚合键“登记日期 (torokubi)”试试看吧。首先从不使用 ROLLUP 开始 (代码清单 8-13)。

代码清单 8-13 在 GROUP BY 中添加“登记日期”(不使用 ROLLUP)

```
SELECT shohin_bunrui, torokubi, SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY shohin_bunrui, torokubi;
```

执行结果 (在 DB2 中执行)

shohin_bunrui	torokubi	sum_tanka
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	2009-09-20	1000
衣服		4000

在上述 GROUP BY 子句中使用 ROLLUP 之后，结果会发生什么变化吗 (代码清单 8-14) ?

代码清单 8-14 代码清单 8-13 在 GROUP BY 中添加“登记日期”(使用 ROLLUP)

```
Oracle SQL Server DB2
SELECT shohin_bunrui, torokubi, SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY ROLLUP(shohin_bunrui, torokubi); ——①
```

#### 特定的 SQL

在 MySQL 中执行代码清单 8-14 时，请将①中的 GROUP BY 子句改写为“GROUP BY shohin\_bunrui, torokubi WITH ROLLUP;”。

执行结果 (在 DB2 中执行)

shohin_bunrui	torokubi	sum_tanka	
		16780	+合计
厨房用具		11180	--小计(厨房用具)
厨房用具	2008-04-28	880	
厨房用具	2009-01-15	6800	
厨房用具	2009-09-20	3500	
办公用品		600	--小计(办公用品)
办公用品	2009-09-11	500	
办公用品	2009-11-11	100	
衣服		5000	--小计(衣服)
衣服	2009-09-20	1000	
衣服		4000	

将上述两个结果进行比较后我们发现,使用 ROLLUP 时多出了最上方的合计行以及 3 条不同商品种类的小计行(也就是未使用登记日期作为聚合键的记录)。这 4 行就是我们所说的超级分组记录。也就是说该 SELECT 语句的结果相当于使用 UNION 对如下 3 种模式聚合级的不同结果进行连接(图 8-5)。

- ① GROUP BY ()
- ② GROUP BY (shohin\_bunrui)
- ③ GROUP BY (shohin\_bunrui, torokubi)

图 8-5 3 种模式聚合级的结果

shohin_bunrui	torokubi	sum_tanka	
		16780	模块 1
厨房用具		11180	
办公用品		600	模块 2
衣服		5000	
办公用品	2009-09-11	500	
办公用品	2009-11-11	100	
厨房用具	2008-04-28	880	
厨房用具	2009-01-15	6800	模块 3
厨房用具	2009-09-20	3500	
衣服	2009-09-20	1000	
衣服		4000	

如果大家觉得上述结果不容易理解的话,可以参考表 8-2 中按照聚合级添加缩进和说明后的内容,理解起来就很容易了。

表 8-2 添加了缩进后的聚合级结果图示

合计		16780
厨房用具	小计	11180
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	小计	600
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	小计	5000
衣服	2009-09-20	1000
衣服		4000

ROLLUP 是“卷起”的意思，比如卷起百叶窗、窗帘卷，等等。其名称也形象地说明了该操作能够得到像从小计到合计这样，从最小的聚合级开始，聚合单位逐渐扩大的结果。



#### 法则 8-7

ROLLUP 可以同时计算出合计和小计，是非常方便的工具。

## 专 栏

### GROUPING 运算符的支持状况

本节介绍的 GROUPING 运算符与 8-1 节介绍的窗口函数都是为了实现 OLAP 用途而添加的功能，是比较新的功能（是 SQL : 1999 版本的标准 SQL 添加的新功能）。因此，还有一些 DBMS 尚未支持这些功能。截止到 2010 年 5 月，Oracle 11g、SQL Server 2008、DB2 9.7 都已经支持这些功能了，但 PostgreSQL 8.4 和 MySQL 5.5 还是不支持这些功能。

想要在不支持 GROUPING 运算符的 DBMS 中获得包含合计或者小计的结果时，只能像本章一开始介绍的那样，使用 UNION 将多条 SELECT 语句连接起来才能实现。

此外，使用 MySQL 时的情况更加复杂一些，只有一个不合规则的 ROLLUP 能够使用。这里所说的“不合规则”指的是需要使用特定的语法。

```
-- MySQL 专用
SELECT shohin_bunrui, torokubi, SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY shohin_bunrui, torokubi WITH ROLLUP;
```

遗憾的是，MySQL 5.5 并不支持 CUBE 和 GROUPING SETS。希望之后的版本能够提供对它们的支持。

## GROUPING 函数——让 NULL 更加容易分辨

可能有些读者会注意到，之前使用 ROLLUP 所得到的结果（代码清单 8-14 的执行结果）有些蹊跷。问题就出在“衣服”的分组之中。有两条记录的 torokubi 列为 NULL，但其原因却并不相同。

sum\_tanka 为 4000 日元的记录，由于商品表中运动 T 恤的注册日期为 NULL，所以就把 NULL 作为聚合键了。这在之前的示例中我们也曾见到过。

相反，sum\_tanka 为 5000 日元的记录，毫无疑问就是超级分组记录的 NULL 了（细目为 1000 日元 + 4000 日元 = 5000 日元）。但由于两者看上去都是“NULL”，实在是难以分辨。

shohin_bunrui	torokubi	sum_tanka	
衣服		5000	由于是超级分组记录的缘故， 所以登记日期为 NULL ——仅仅因为“运动 T 恤”的登记 日期为 NULL
衣服	2009-09-20	1000	
衣服		4000	

为了避免混淆，SQL 提供了一个用来判断超级分组记录的 NULL 的特定函数——GROUPING 函数。该函数在其参数列的值为超级分组记录所产生的 NULL 时返回 1，其他情况返回 0（代码清单 8-15）。

### KEYWORD

● GROUPING 函数

代码清单 8-15 使用 GROUPING 函数来判断 NULL

```
Oracle SQL Server DB2
SELECT GROUPING(shohin_bunrui) AS shohin_bunrui,
       GROUPING(torokubi) AS torokubi, SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY ROLLUP(shohin_bunrui, torokubi);
```

执行结果（在 DB2 中执行）

shohin_bunrui	torokubi	sum_tanka	
1	1	16780	
0	1	11180	
0	0	880	
0	0	6800	
0	0	3500	
0	1	600	
0	0	500	
0	0	100	
0	1	5000	← 碰到超级分组记录中的 NULL 时返回 1
0	0	1000	
0	0	4000	← 原始数据为 NULL 时返回 0

这样就能判断出是超级分组记录中的NULL还是原始数据本身为NULL了。使用GROUPING函数还能在超级分组记录的键值中插入字符串。也就是说,当GROUPING函数的返回值为1时,指定“合计”或者“小计”等字符串,其他情况返回通常的列的值(代码清单8-16)。

代码清单 8-16 在超级分组记录的键值中插入恰当的字符串

```

Oracle      SQL Server  DB2
SELECT CASE WHEN GROUPING(shohin_bunrui) = 1
           THEN '商品种类 合计'
           ELSE shohin_bunrui END AS shohin_bunrui,
       CASE WHEN GROUPING(torokubi) = 1
           THEN '登记日期 合计'
           ELSE CAST(torokubi AS VARCHAR(16)) END AS torokubi,
       SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY ROLLUP(shohin_bunrui, torokubi);

```

执行结果(在DB2中执行)

shohin_bunrui	torokubi	sum_tanka
商品种类 合计	登记日期 合计	16780
厨房用具	登记日期 合计	11180
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	登记日期 合计	600
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	登记日期 合计	5000
衣服	2009-09-20	1000
衣服		4000

将超级分组记录中的NULL  
替换为“登记日期 合计”  
原始数据中的NULL保持  
不变

在实际业务中,需要获取包含合计或者小计的统计结果(这种情况是最多的)时,就可以使用ROLLUP和GROUPING函数来实现了。

```
CAST(torokubi AS VARCHAR(16))
```

那为什么还要将SELECT子句中的torokubi列转换为CAST(torokubi AS VARCHAR(16))形式的字符串呢?这是为了满足CASE表达式所有分支的返回值必须一致的规定。如果不这样的话,那么各个分支会分别返回日期类型和字符串类型的值,执行时就会发生语法错误。



#### 法则 8-8

使用GROUPING函数能够简单地分辨出原始数据中的NULL和超级分组记录中的NULL。



## KEYWORD

● CUBE 运算符

## CUBE——用数据来搭积木

ROLLUP 之后我们来介绍另一个常用的 GROUPING 运算符——CUBE。CUBE 是“立方体”的意思，这个奇怪的名字和 ROLLUP 一样，都能形象地说明该函数的动作。究竟是什么样的动作呢，还是让我们通过一个例子来看一看吧。

CUBE 的语法和 ROLLUP 相同，只需要将 ROLLUP 替换为 CUBE 就可以了。下面我们就把代码清单 8-16 中的 SELECT 语句替换为 CUBE 试试看吧（代码清单 8-17）。

代码清单 8-17 使用 CUBE 取得全部组合的结果

```

Oracle  SQL Server  DB2
SELECT CASE WHEN GROUPING(shohin_bunrui) = 1
           THEN '商品种类 合计'
           ELSE shohin_bunrui END AS shohin_bunrui,
       CASE WHEN GROUPING(torokubi) = 1
           THEN '登记日期 合计'
           ELSE CAST(torokubi AS VARCHAR(16)) END AS torokubi,
       SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY CUBE(shohin_bunrui, torokubi);

```

执行结果（在 DB2 中执行）

shohin_bunrui	torokubi	sum_tanka
商品种类 合计	登记日期 合计	16780
商品种类 合计	2008-04-28	880 ←追加
商品种类 合计	2009-01-15	6800 ←追加
商品种类 合计	2009-09-11	500 ←追加
商品种类 合计	2009-09-20	4500 ←追加
商品种类 合计	2009-11-11	100 ←追加
商品种类 合计		4000 ←追加
厨房用具	登记日期 合计	11180
厨房用具	2008-04-28	880
厨房用具	2009-01-15	6800
厨房用具	2009-09-20	3500
办公用品	登记日期 合计	600
办公用品	2009-09-11	500
办公用品	2009-11-11	100
衣服	登记日期 合计	5000
衣服	2009-09-20	1000
衣服		4000

与 ROLLUP 的结果相比，CUBE 的结果中多出了几行记录。大家看一下应该就明白了，多出来的记录就是只把 torokubi 作为聚合键所得到的

聚合结果。

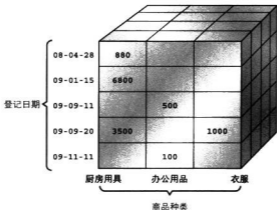
- ① GROUP BY ()
- ② GROUP BY (shohin\_bunrui)
- ③ GROUP BY (torokubi) ←添加的组合
- ④ GROUP BY (shohin\_bunrui, torokubi)

所谓 CUBE，就是将 GROUP BY 子句中聚合键的“所有可能组合”的聚合结果集中到一个结果中的功能。因此，组合的个数就是  $2^n$  ( $n$  是聚合键的个数)。本例中聚合键有 2 个，所以“ $2^2=4$ ”。如果再添加 1 个变为 3 个聚合键的话，就是“ $2^3=8$ ”<sup>①</sup>。

读到这里，可能很多读者都会觉得奇怪，究竟 CUBE 运算符和立方体有什么关系呢？

众所周知，立方体由长、宽、高 3 个轴构成。对于 CUBE 来说，一个聚合键就相当于其中的一个轴，而结果就是将数据像积木那样堆积起来（图 8-6）。

图 8-6 CUBE 的执行图示



由于本例中只有商品种类 (shohin\_bunrui) 和登记日期 (torokubi) 两个轴，所以我们看到的其实是一个正方形，请大家把它看作缺了一个轴的立方体。通过 CUBE 当然也可以指定 4 个以上的轴，但由于那已经属于 4 维空间的范畴了，是无法用图形来表示的。

①

使用 ROLLUP 时组合的个数是  $n+1$ 。随着组合个数的增加，结果的行数也会增加，因此如果使用 CUBE 时不加以注意的话，往往会得到意想不到的巨大结果。顺便说一下，ROLLUP 的结果一定包含在 CUBE 的结果之中。



## 法则 8-9

可以把 CUBE 理解为将使用聚合键进行切割的模块堆积成一个立方体。

## KEYWORD

● GROUPING SETS 运算符

## GROUPING SETS——取得期望的积木

最后要介绍给大家的 GROUPING 运算符是 GROUPING SETS。该运算符可以用于从 ROLLUP 或者 CUBE 的结果中取出部分记录。

例如，之前的 CUBE 的结果就是根据聚合键的所有可能组合计算而来的。如果希望从中选取出“商品种类”和“登记日期”各自作为聚合键的结果，反之，不想得到“合计记录和使用 2 个聚合键的记录”时，可以使用 GROUPING SETS（代码清单 8-18）。

代码清单 8-18 使用 GROUPING SETS 取得部分组合结果

```

Oracle SQL Server DB2
SELECT CASE WHEN GROUPING(shohin_bunrui) = 1
            THEN '商品种类 合计'
            ELSE shohin_bunrui END AS shohin_bunrui,
       CASE WHEN GROUPING(torokubi) = 1
            THEN '登记日期 合计'
            ELSE CAST(torokubi AS VARCHAR(16)) END AS torokubi,
       SUM(hanbai_tanka) AS sum_tanka
FROM Shohin
GROUP BY GROUPING SETS (shohin_bunrui, torokubi);

```

执行结果（在 DB2 中执行）

shohin_bunrui	torokubi	sum_tanka
商品种类 合计	2008-04-28	880
商品种类 合计	2009-01-15	6800
商品种类 合计	2009-09-11	500
商品种类 合计	2009-09-20	4500
商品种类 合计	2009-11-11	100
商品种类 合计		4000
厨房用具	登记日期 合计	11180
办公用品	登记日期 合计	600
衣服	登记日期 合计	5000

上述结果中也没有全体的合计行（16780 日元）。与 ROLLUP 或者 CUBE 能够得到规定的（业务上成为“固定的”）结果相对，GROUPING SETS 用于从中取出个别条件对应的不固定的结果。然而，由于期望获得

不固定结果的情况少之又少，所以与 ROLLUP 或者 CUBE 比起来使用 GROUPING SETS 的机会也就很少了。

### 练习题

- 8.1 请说出针对本章中使用的 Shohin (商品) 表执行如下 SELECT 语句所能得到的结果。

```
SELECT shohin_id, shohin_mei, hanbai_tanka,  
       MAX (hanbai_tanka) OVER (ORDER BY shohin_id) AS ↗  
       current_max_tanka  
FROM Shohin;
```

↗表示下一行接续本行，只是由于版面所限而换行。

- 8.2 继续使用 Shohin 表，计算出按照登记日期 (torokubi) 升序进行排列的各日期的销售单价 (hanbai\_tanka) 的总额。排序是需要将登记日期为 NULL 的“运动 T 恤”记录排在第一位(也就是将其看作比其他日期都早)。

# 附录 A

## 安装 PostgreSQL

本书附带的光盘中收录了作为 SQL 学习环境的 PostgreSQL。PostgreSQL 是 1980 年以加利福尼亚大学为主开发的 DBMS，与 MySQL 一样，都是世界上广泛应用的开源 DB。由于其支持标准 SQL 的意识非常强，所以最适合初学者学习。本附录将会引导大家将 PostgreSQL 安装到自己的电脑中。

此外，光盘中收录的 PostgreSQL 必须安装在以下的 OS（操作系统）中。

### 安装环境（32 位 OS）

Windows XP、Windows Vista、Windows 7

#### 注意事项

- 本书使用 PostgreSQL 作为 SQL 的学习环境，当然也可以使用其他的关系数据库。
- 本书前言（随书光盘简介）中介绍了光盘收录的内容。请在执行附录内容之前阅读其中的内容。
- 虽然这里介绍的是 Windows 7 中的安装方法，但是 Windows XP 或者 Windows Vista 中同样适用。

## 安装步骤

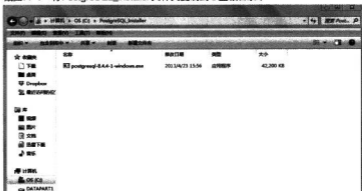
下面就让我们按照如下步骤来安装 PostgreSQL 吧。

### 步骤 1：复制 PostgreSQL\_installer 目录

首先使用电脑打开光盘，将其中的 PostgreSQL\_Installer 文件夹，通过鼠标拖拽复制到 C 盘根目录下（截图 A-1）。

PostgreSQL\_Installer 目录中收录了 PostgreSQL 的安装文件。

截图A-1 将 PostgreSQL\_Installer 文件夹复制到 C 盘根目录下



### 步骤 2: 执行安装文件

鼠标右键单击文件夹中 PostgreSQL 的安装文件“postgresql-8.4.4-1-windows.exe”，选择“以管理员身份运行”。

启动安装文件后，会显示出 Setup 步骤（截图 A-2），点击“Next >”按钮。

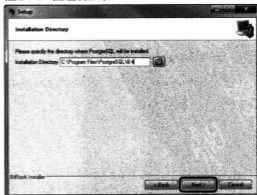
截图A-2 Setup 步骤



### 步骤 3: 设置安装目录

接下来，我们设置 PostgreSQL 的安装目录（截图 A-3）。如果没有特殊需要，请保持默认的安装目录“C:\Program Files\PostgreSQL\8.4”，点击“Next >”按钮。

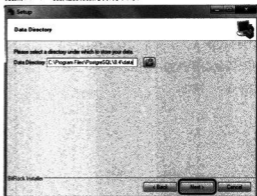
截图A-3 指定安装目录



**步骤4：设置数据的保存目录**

接下来，我们设置 PostgreSQL 数据的保存目录（截图 A-4）。指定“C:\Program Files\PostgreSQL\8.4\data”，然后点击“Next>”按钮。

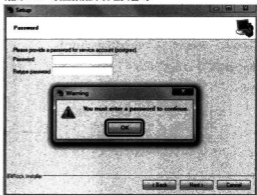
截图 A-4 指定数据的保存目录

**步骤5：设置登录密码**

设置数据库管理员（用户名：postgres）密码（截图 A-5）。

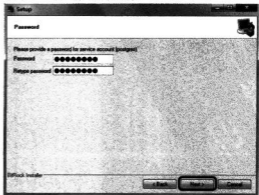
点击“OK”按钮关闭弹出的警告窗口，输入任意密码，然后点击“Next>”按钮（截图 A-6）。请在密码输入框“Password”和确认密码框“Retype password”中输入相同的密码。

截图 A-5 设置数据库管理员密码



在这里设置的密码将会之后登录 PostgreSQL 时使用，因此请大家牢记。

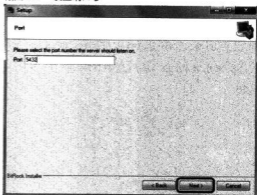
截图 A-6 设置密码后的对话框



**步骤6: 设置端口号**

设置连接 PostgreSQL 时使用的端口号(截图 A-7)。如果没有其他程序使用, 请使用默认的“5432”端口, 点击“Next >”按钮。

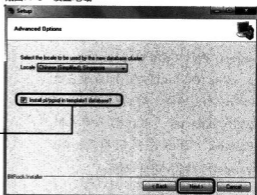
截图 A-7 设置端口号

**步骤7: 设置地域**

设置地域 (Locale)。在下拉列表“Locale”中选择“Chinese[Simplified], Singapore”, 点击“Next >”按钮(截图 A-8)。

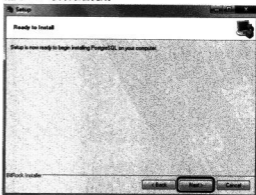
pl/pgsql 是为 SQL 添加控制结构(循环、分支等)的语言。不选择该复选框则不会安装该语言。虽然本书中并未使用该语言, 但选择默认方式(选中复选框)进行安装也没有关系。

截图 A-8 设置地域

**步骤8: 开始安装**

在安装准备完毕画面(截图 A-9)中, 点击“Next >”按钮。

截图 A-9 安装准备完成

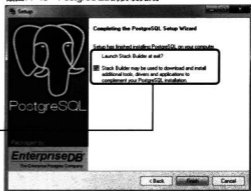




接下来开始安装，请耐心等待几分钟。安装完成之后会弹出对话框 A-10。取消“Launch Stack Builder at exit?”复选框，点击“Finish”按钮。

Stack Builder 是安装附加工具的功能，选中该复选框会继续安装附加工具。但是，由于本书中并未使用这些附加工具，所以取消该复选框也没有关系。


截图 A-10 PostgreSQL 的安装完成



## PostgreSQL 的启动确认

下面我们来确认一下 PostgreSQL 是否正常启动了。

### 步骤 1：打开控制面板

点击桌面左下角的“开始”按钮 ，打开菜单，选择“控制面板”（截图 A-11），会弹出控制面板。如截图 A-12 所示，通过右上角的“显示方式”切换到更加容易操作的“小图标”模式（Vista 以前的系统请切换到“经典模式”）。

截图 A-11 选择“控制面板”



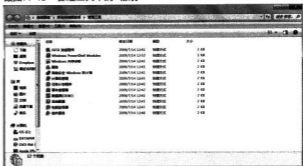
截图 A-12 控制面板



**步骤2: 打开管理工具中的“服务”**

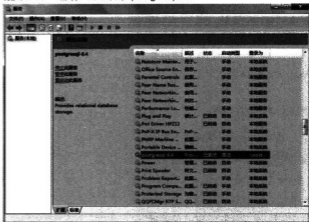
在控制面板中,按照“管理工具”→“服务”的顺序点击(截图 A-13),打开管理工具中的“服务”。

截图 A-13 管理工具中的“服务”



如果“服务”对话框(截图 A-14)中“postgres-8.4”的“状态”是“开始”,则表示 PostgreSQL 已经正常启动了。

截图 A-14 “服务”画面中的“postgres-8.4”



## 更改 PostgreSQL 的启动设置

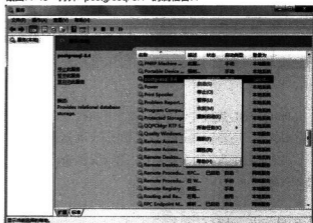
安装 PostgreSQL 之后,管理工具中“服务”对话框的“启动类型”会默认设置为“自动”。由于该设置会在电脑启动时自动启动 PostgreSQL,所以会增加电脑负荷。

如果想要将设置改为仅在需要使用数据库时才启动 PostgreSQL,可以通过以下步骤来实现。

### 步骤 1: 打开“postgresql-8.4”的属性窗口

在管理工具的“服务”画面中右键点击“postgresql-8.4”所在行，选择“属性”（截图 A-15）。

截图 A-15 打开“postgresql-8.4”的属性窗口



### 步骤 2: 改变启动类型

将“启动类型”改为“手动”（截图 A-16）。

这样 PostgreSQL 就不会（在电脑启动 / 重新启动时）自动启动了。

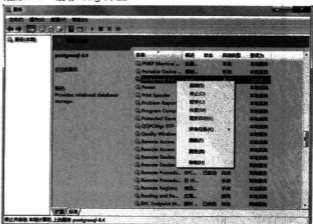
截图 A-16

在属性对话框将“启动类型”选择为“手动”



在 PostgreSQL 不能自动启动时，可以在管理工具的“服务”中，右键点击“postgresql-8.4”，选择“开始”（截图 A-17）。这样就可以启动 PostgreSQL 进行使用了。

截图 A-17 启动 PostgreSQL



# 附录 B

## 在 PostgreSQL 中执行 SQL 的方法

PostgreSQL 提供了可以通过命令行执行 SQL 的工具 `psql`。`psql` 可以将 SQL 语句发送给 PostgreSQL，并接收其返回结果进行显示。本附录将会介绍使用 `psql` 执行 SQL 的方法。

此外，这里执行的 SQL 的语法和含义在第 1 章和第 2 章中都已经学过了，因此大家无需特别注意。


---

### 登录示例数据库

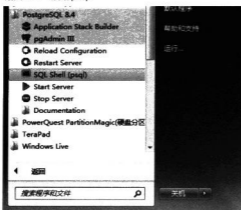
安装 PostgreSQL 之后，会自动创建名为 `postgres` 的示例数据库。

首先让我们启动 `psql`，登录该示例数据库。

#### 步骤 1：启动 `psql`

点击桌面左下角的“开始”按钮  打开开始菜单。然后右键点击“全部程序”→“PostgreSQL8.4”→“SQL Shell (psql)”，选择“以管理员身份运行”（截图 B-1）。

截图 B-1 启动 `psql`



## 步骤 2：登录

如截图 B-2 所示启动 psql。

截图 B-2 psql



由于服务器、数据库、端口、用户名和客户端字符集都可以使用初期设置值，所以只需按 5 次回车键就可以了（截图 B-3）。括号中的就是初期设置的值。

截图 B-3 按下 5 次回车键后的状态



### 注意事项

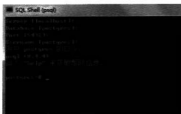
如果出现“无法连接服务器”的错误，则说明 PostgreSQL 并未启动。请按照附录 A 中“改变 PostgreSQL 的启动设置”介绍的方法，将 PostgreSQL 的状态设置为“开始”。

此时需要用到用户 `postgres` 的密码，请输入安装 PostgreSQL 时设置的密码，然后按下回车键。

```
postgres=#
```

如果显示“`postgres=#`”则代表登录成功（截图 B-4）。

截图B-4 登录示例数据库 (postgres) 后的状态



**注意**

为了保证安全, 输入的密码不会显示出来。由于闪烁的光标位置不会发生改变, 大家可能会觉得什么都没输入进去, 但其实密码已经正常输入了, 因此请在输入结束时按下回车键。

## 执行 SQL

登录数据库之后就可以执行 SQL 了。下面就让我们尝试一下执行简单的 SQL 语句。

### 步骤 1: 输入 SQL 语句

在通过 psql 登录到示例数据库 (postgres) 之后, 输入如下 1 行代码。输入后的结果应该如截图 B-5 所示。

```
SELECT 1;  
└─(半角空格 (输入空格键))
```

截图B-5 输入“SELECT 1;”



### 步骤 2: 按下回车键

输入结束之后, 请按下车键。这样就会执行该 SQL 语句了。如果显示出截图 B-6 中



那样的结果，就说明执行成功了。

截图 B-6 “SELECT 1;” 的执行结果



#### 注意事项

“;”是 SQL 语句结束的符号，不输入分号直接按下回车键是不会执行 SQL 语句的。因此，执行 SQL 语句时，请注意不要忘记写“;”。

## 创建学习用的数据库

本书从第 1 章后半部分开始将会和大家一起学习各种各样的 SQL 语句的书写方法。让我们来创建一个学习用的数据库，提前准备一下吧。

按照如下步骤，创建数据库。

### 步骤 1：执行创建数据库的 SQL 语句

输入如下 1 行代码，按下回车键。请注意数据库的名称只能使用小写字母。

```
CREATE DATABASE shop;
```

执行成功的话会显示出截图 B-7 那样的结果。

```
CREATE DATABASE
```

截图 B-7 数据库创建成功



### 步骤 2: 结束 psql

数据库创建之后，结束 psql。为了结束 psql，

```
\q
```

请输入“\q”，然后按下回车键。这样就会像截图 B-8 那样，显示出“按下任意键继续”的信息，按下任意键关闭 psql 的窗口。

截图 B-8 “\q”的执行结果




### 备注

由于当前登录的是示例数据库“postgres”，所以为了登录我们创建好的数据库，需要暂时结束 psql（注销）。psql 在其窗口关闭时也会结束，因此也可以通过点击 psql 窗口右上角的“x”按钮来结束 psql。

## 登录学习用的数据库

下面就让我们来登录之前创建好的数据库“shop”吧。

### 步骤 1: 启动 psql

与之前我们启动 psql 的方式相同，点击桌面左下角的“开始”按钮  打开开始菜单。然后按照“全部程序”→“PostgreSQL8.4”→“SQL Shell (psql)”的顺序点击，启动 psql。

### 步骤 2: 登录设置

由于这次我们要登录数据库 shop，所以需要指定数据库名称。因此在按下 1 次回车键之后，需要在“Database [postgres]”后面输入“shop”（截图 B-9）。



```
shop
```

截图B-9 输入“shop”



其他的设置和之前启动 psql 时完全相同。按下 4 次回车键之后，会要求输入用户 postgres 的密码，输入安装 PostgreSQL 时设置的密码，按下回车键。

如果登录数据库 shop 成功的话，就会像截图 B-10 那样，显示出“shop=#”。

```
shop=#
```

截图B-10 登录成功!



这样一来，执行针对数据库 shop 的 SQL 语句的环境就准备完毕了。之后就可以按照本书内容输入 SQL，然后按下回车键来执行了。

本书会使用数据库 shop，通过执行各种各样的 SQL，来学习 SQL 的书写方法。

#### 备 忘

随书光盘收录的示例程序文件（\*.sql）可以使用 psql 直接读取并执行。通过 psql 读取文件时，需要使用“\i <SQL 文件的路径>”这样的格式输入命令。此时请使用 Windows 中分隔目录用的“/”替换 <SQL 文件的路径> 中的“\”。例如，执行“C:\PostgreSQL”文件夹中的“CreateTableShohin.sql”文件时，需要在 psql 中输入如下命令。然后按下回车键，就可以执行文件中的 SQL 了。

```
shop=# \i C:/PostgreSQL/CreateTableShohin.sql
```

# 附录 C

## 练习题答案

\* 本附录中程序（SQL 语句）的答案并非唯一，还存在其他满足条件的解答方法。

\* 代码清单中的“↵”是为了配合页面显示所进行的换行操作。

### 1.1

```
CREATE TABLE Jyushoroku
(
    toroku_bango INTEGER NOT NULL,
    nameae VARCHAR(128) NOT NULL,
    jyusho VARCHAR(256) NOT NULL,
    tel_no CHAR(10) ,
    mail_address CHAR(20) ,
    PRIMARY KEY (toroku_bango));
```

### 1.2

**PostgreSQL MySQL**

```
ALTER TABLE Jyushoroku ADD COLUMN yubin_bango CHAR(8) NOT NULL;
```

**Oracle**

```
ALTER TABLE Jyushoroku ADD (yubin_bango CHAR(8)) NOT NULL;
```

**SQL Server**

```
ALTER TABLE Jyushoroku ADD yubin_bango CHAR(8) NOT NULL;
```

**DB2**

无法添加。

在 DB2 中，如果要为添加的列设置 NOT NULL 约束，需要像下面这样指定默认值，或者删除 NOT NULL 约束，否则就无法添加新列。

**DB2 特殊**

```
ALTER TABLE Jyushoroku ADD COLUMN yubin_bango CHAR(8) NOT NULL DEFAULT ↵
'0000-000';
```

### 1.3

```
DROP TABLE Jyushoroku;
```

1.4 删除后的表无法使用命令进行恢复。请使用习题 1.1 答案中的 CREATE TABLE 语句再

次创建所需的表。

## 2.1

```
SELECT shohin_mei, torokubi
FROM Shohin
WHERE torokubi >= '2009-04-28';
```

执行结果

shohin_mei	torokubi
手电形	2009-09-20
打孔器	2009-09-11
菜刀	2009-09-20
叉子	2009-09-20
圆珠笔	2009-11-11

2.2 ①~③中的 SQL 语句都无法选取出任何 1 条记录。

## 2.3 SELECT 语句①

```
SELECT shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
WHERE hanbai_tanka >= shiire_tanka + 500;
```

SELECT 语句②

```
SELECT shohin_mei, hanbai_tanka, shiire_tanka
FROM Shohin
WHERE hanbai_tanka - 500 >= shiire_tanka;
```

## 2.4

```
SELECT shohin_mei, shohin_bunrui,
       hanbai_tanka * 0.9 - shiire_tanka AS rieki
FROM Shohin
WHERE hanbai_tanka * 0.9 - shiire_tanka > 100
      AND ( shohin_bunrui = '办公用品'
           OR shohin_bunrui = '厨房用具');
```

执行结果

shohin_mei	shohin_bunrui	rieki
打孔器	办公用品	130.0
高压锅	厨房用具	1120.0

3.1 存在以下 3 个错误。

1. 使用了字符类型的列 (shohin\_mei) 作为 SUM 函数的参数。

>> 解答

SUM 函数只能使用数值类型的列作为参数。

2. WHERE 子句写在了 GROUP BY 子句之后。

>> 解答

WHERE 子句必须写在 GROUP BY 子句之前。

3. SELECT 子句中存在 GROUP BY 子句中未指定的列 (shohin\_id)。

>> 解答

使用 GROUP BY 子句时，书写在 SELECT 子句中的列有很多限制。GROUP BY 子句中未指定的列不能书写在 SELECT 子句之中。

此外，虽然在 SELECT 子句和 FROM 子句之间添加注释在语法上没有问题，但由于这样会使 SQL 语句难以阅读，所以请不要这样书写。

在 WHERE 子句中指定 torokubi 的大小关系作为条件并没有什么问题。

### 3.2

```
SELECT shohin_bunrui, SUM(hanbai_tanka), SUM(shiire_tanka)
FROM Shohin
GROUP BY shohin_bunrui
HAVING SUM(hanbai_tanka) > SUM(shiire_tanka) * 1.5;
```

>> 解答

由于该 SELECT 语句是在按照商品种类进行分组之后，指定各组所对应的条件，所以使用了 HAVING 子句。条件为“大于 1.5 倍”，而不是“大于等于 1.5 倍”，因此条件表达式为“>”而不是“>=”。

### 3.3

```
SELECT *
FROM Shohin
ORDER BY torokubi DESC, hanbai_tanka;
```

>> 解答

使用 ORDER BY 子句指定排列顺序之后，肯定有一列会按照升序或者降序进行排列。本习题中是登记日期（NULL 排在开头还是末尾会根据 DBMS 不同而不同，无需考虑）。因此我们能够推断出首先是按照登记日期的降序进行排序的。

接下来，对于日期相同的记录，例如同为“2009-09-20”的 3 条记录，可以看出是按照销售单价的升序进行排序的。

#### 4.1 1行也选取不出来。

>> 解答

A先生使用 `BEGIN TRANSACTION` 启动了事务处理，然后开始执行 `INSERT` 语句。因此，在A先生使用 `COMMIT` 确定该更新之前，B先生等其他用户都无法看到A先生进行更新的结果。这就是基于 `ACID` 特性中的 `I`，也就是独立性（`Isolation`）现象。当然，由于A先生在 `COMMIT` 之前能看到自己进行过的更新，所以如果A先生执行 `SELECT * FROM Shohin;` 的话，会得到3条记录。

顺便提一下，如果想要确认该现象，并不需要两个人。只需使用电脑打开两个窗口连接同一个数据库，一个人就能完成两个人的工作了。

#### 4.2 由于商品编号列违反了主键约束，所以会发生错误，1行也 `INSERT` 不了。

>> 解答

如果该 `INSERT` 能够正常执行的话，`Shohin`（商品）表的状态应该会像下面这样增加双倍6行数据。

Shohin(商品)表

商品编号	商品名称	商品种类	销售单价	进货单价	登记日期
0001	T恤衫	衣服	1000	500	2008-09-20
0002	打孔器	办公用品	500	320	2008-09-11
0003	运动T恤	衣服	4000	2800	
0001	T恤衫	衣服	1000	500	2008-09-20
0002	打孔器	办公用品	500	320	2008-09-11
0003	运动T恤	衣服	4000	2800	

但是，显然，上述记录违反了商品编号列的主键约束（不能存在主键重复的记录）。违反该约束带来的后果就是无法执行更新操作，这就是 `ACID` 特性中的 `C`——一致性（`Consistency`）。

#### 4.3

```
INSERT INTO ShohinSaeki (shohin_id, shohin_mei, hanbai_tanka, ⇨
shiire_tanka, saeki)
SELECT shohin_id, shohin_mei, hanbai_tanka, shiire_tanka, ⇨
hanbai_tanka - shiire_tanka
FROM Shohin;
```

>> 解答

Shohin (商品) 表和 ShohinSaeki (商品利润) 表中定义完全相同的列 shohin\_id (商品编号)、shohin\_mei (商品名称)、hanbai\_tanka (销售单价)、shiire\_tanka (进货单价), 可以通过 SELECT 语句直接从 Shohin (商品) 表取出插入到 ShohinSaeki (商品利润) 表中。只有 Shohin (商品) 表中没有的 saeki (利润) 列的值需要根据 shiire\_tanka 进货单价和 hanbai\_tanka 销售单价进行计算。

#### 4.4 1.

```
-- 下调销售单价
UPDATE ShohinSaeki
   SET hanbai_tanka = 3000
   WHERE shohin_id = '0003';
```

#### 2.

```
-- 重新计算利润
UPDATE ShohinSaeki
   SET saeki = hanbai_tanka - shiire_tanka
   WHERE shohin_id = '0003';
```

#### 5.1

```
-- 创建视图的语句
CREATE VIEW ViewRenshu5_1 AS
SELECT shohin_mei, hanbai_tanka, torokubi
   FROM Shohin
  WHERE hanbai_tanka >= 1000
     AND torokubi = '2009-09-20';
```

#### 5.2 会发生错误。

>> 解答

对视图的更新归根结底是对视图所对应的表进行更新。因此, 该 INSERT 语句实质上和下面的 INSERT 语句相同。

```
INSERT INTO Shohin (shohin_id, shohin_mei, shohin_bunrui, hanbai_tanka,
shiire_tanka, torokubi)
   VALUES (NULL, '刀子', NULL, 300, NULL, '2009-11-02');
```

shohin\_id (商品编号)、shohin\_mei (商品名称)、shohin\_bunrui (商品种类) 3 列在表定义时都被赋予了 NOT NULL 约束<sup>④</sup>。因此, 向 shohin\_id (商品编号) 以及 shohin\_bunrui (商品种类) 中插入 NULL 的 INSERT 语句是无法执行的。

并且 INSERT 语句中只对 shohin\_mei (商品名称)、hanbai\_tanka (销售单价)、torokubi (登记日期) 3 列进行了赋值, 所以剩余的列都会被自动插入 NULL, 于是就发生了错误。

### ④

其实 shohin\_id (商品编号) 是被赋予了主键约束, 但其中默认包含了 NOT NULL 约束。

## 5.3

```
SELECT shohin_id,
       shohin_mei,
       shohin_bunrui,
       hanbai_tanka,
       (SELECT AVG(hanbai_tanka) FROM Shohin) AS hanbai_tanka_all
FROM Shohin;
```

### >> 解答

使用标量子查询来计算销售单价的平均值。由于平均销售单价是 2091.5 这样一个单值, 可以确定为标量值, 所以可以书写在 SELECT 子句之中。

但是有没有读者会想到如下 SELECT 语句呢?

```
SELECT shohin_id,
       shohin_mei,
       shohin_bunrui,
       hanbai_tanka,
       AVG(hanbai_tanka) AS hanbai_tanka_all
FROM Shohin;
```

上述 SELECT 语句会发生错误<sup>⑤</sup>。原因在于 AVG 是一个聚合函数。正如 3-3 节说明的那样, 使用聚合函数时对书写在 SELECT 子句中的要素有很多限制。使用了这种错误方法的读者请重新阅读一下 3-3 节的内容。

### ⑤

虽然在 MySQL 中该 SELECT 语句不会发生错误, 但毕竟这只是基于 MySQL 特定需求的结果, 无法在其他的 DBMS 中使用。并且得到的结果也完全不同。

## 5.4

```
-- 创建视图的语句
CREATE VIEW AvgTankaByBunrui AS
SELECT shohin_id,
       shohin_mei,
       shohin_bunrui,
       hanbai_tanka,
       (SELECT AVG(hanbai_tanka)
        FROM Shohin S2
        WHERE S1.shohin_bunrui = S2.shohin_bunrui
        GROUP BY S1.shohin_bunrui) AS avg_hanbai_tanka
FROM Shohin S1;
```

```
-- 删除视图的语句
DROP VIEW AvgTankaByBunrui;
```

&gt;&gt; 解答

在视图所包含的列中，除了 avg\_hanbai\_tanka 之外的 4 列 (shohin\_id、shohin\_mei、shohin\_bunrui、hanbai\_tanka) 在 Shohin 表中都存在，因此可以直接读取。但是，最后的 avg\_hanbai\_tanka（平均销售单价）则必须使用关联子查询进行结算。使用标量子查询和关联子查询也可以创建出上述视图。

## 6.1 ①的答案

```
shohin_mei|shiire_tanka
-----+-----
打孔器    |          320
擦菜板    |          790
```

&gt;&gt; 解答

对于①的结果应该没有什么疑问。由于要选取的是进货单价 (shiire\_tanka) 为 500 日元、2800 日元、5000 日元之外的商品 (shiire\_mei)，所以会得到 320 日元的打孔器和 790 日元的擦菜板两条记录。此外，不仅是 IN，通常的谓词都无法与 NULL 进行比较，因此进货单价 (shiire\_tanka) 为 NULL 的叉子和圆珠笔都没有出现在结果之中。

②的答案 无法取出任何记录

```
shohin_mei|shiire_tanka
-----+-----
```



>> 解答

②的结果有必要说明一下。②的 SQL 仅仅是在①的 NOT IN 的参数中增加了 NULL。并且①的结果中已经排除了进货单价 (shiire\_tanka) 为 NULL 的记录, 因此大家可能会觉得②的结果也是如此。但让人吃惊的是②的 SQL 却无法选取出任何记录。不仅仅是进货单价为 NULL 的记录, 连从①中选取出打孔器和擦菜板也不见了。

其实这是 SQL 中最危险的陷阱。NOT IN 的参数中包含 NULL 时结果通常会为空, 也就是无法选取出任何记录。

为什么会得到这样的结果呢, 其中的理由十分复杂, 属于中级学习的范畴, 因此本书不会进行详细介绍<sup>①</sup>。这里希望大家了解的是 NOT IN 的参数中不能包含 NULL。不仅仅是指定 NULL 的情况, 使用子查询作为 NOT IN 的参数时, 该子查询的返回值也不能是 NULL。请大家一定要遵守这一规定。

①

想要了解为什么 NOT IN 会得到这样结果的读者, 可以参考拙著《深入浅出 SQL 徹底指南書》(翔泳社出版)中“第1部: 1-3 3值逻辑和 NULL”的内容。

## 6.2

```
SELECT SUM(CASE WHEN hanbai_tanka <= 1000
                THEN 1 ELSE 0 END) AS low_price,
       SUM(CASE WHEN hanbai_tanka BETWEEN 1001 AND 3000
                THEN 1 ELSE 0 END) AS mid_price,
       SUM(CASE WHEN hanbai_tanka >= 3001
                THEN 1 ELSE 0 END) AS mid_price
FROM Shohin;
```

>> 解答

大家发现了吗, 这与我们在 6-3 节中的“CASE 表达式的书写位置”中学过的使用 CASE 表达式进行行列变换是相似的问题。如果能够使用 CASE 表达式创建出 3 个分类条件的话, 之后就可以将其与聚合函数进行组合了。只有计算中间额度商品<sup>①</sup>条件中的 BETWEEN 需要注意一下。

①

此处的“中间额度”是笔者创造出来的词语, 大家应该能理解其中的含义。

## 7.1 如下所示, 会将 Shohin 表中的 8 行记录原封不动地选取出来。

执行结果

shohin_id	shohin_mei	shohin_bunrui	hanbai_tanka	shiire_tanka	torokubi
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	橡皮板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

>> 解答

可能有些读者会对此感到惊讶，心想同时使用 UNION 和 INTERSECT 时，不是 INTERSECT 会优先执行吗……实在对不起了。当然，从执行顺序上来说确实是先 INTERSECT 开始的，但是在此之前，由于对同一张表使用了 UNION 或者 INTERSECT，所以结果并不会发生改变。也就是说，由于 UNION 或者 INTERSECT 未使用 ALL，会排除掉重复的记录，所以对同一张表来说，无论执行多少次操作原表也不会发生改变。

## 7.2 SELECT 语句如下所示。

```
SELECT COALESCE(TS.tenpo_id, '不确定') AS tenpo_id,
       COALESCE(TS.tenpo_mei, '不确定') AS tenpo_mei,
       S.shohin_id,
       S.shohin_mei,
       S.hanbai_tanka
FROM TenpoShohin TS RIGHT OUTER JOIN Shohin S
ON TS.shohin_id = S.shohin_id
ORDER BY tenpo_id;
```

>> 解答

大家想起这个名字有点奇怪的 COALESCE 函数了吗？该函数可以将 NULL 转换为其他的值。虽然名字有些古怪，但使用却很频繁。特别是在希望改变外部连接结果中的 NULL 时，该函数是唯一的选择，因此希望大家能够牢记。

## 8.1 结果如下。

shohin_id	shohin_mei	hanbai_tanka	current_max_tanka	
0001	T恤衫	1000	1000	+ (1000) 的最大值
0002	打孔器	500	1000	+ (1000, 500) 的最大值
0003	运动T恤	4000	4000	+ (1000, 500, 4000) 的最大值

0004	菜刀	3000	4000	← (1000, 500, 4000, 3000) 的最大值
0005	高压锅	6800	6800	
0006	叉子	500	6800	
0007	煎菜板	880	6800	
0008	微波炉	100	6800	

>> 解答

本题中 SELECT 语句的含义是“按照商品编号 (shohin\_id) 的升序进行排序，计算出截至当前行的最高销售单价”。因此，在显示出最高销售单价的同时，窗口函数的返回结果也会变化。这恰好和奥运会等竞技体育的最高记录不断变化相似。随着商品编号越来越大，计算最大值的对象范围也不断扩大。就像随着时代变迁，运动员数量也会逐渐增加，要选出“历代第一”也会越来越难。

此外，在 SQL Server 中将聚合函数作为窗口函数使用时，由于不能使用 ORDER BY，所以答案中的 SELECT 语句会发生语法错误。

8.2 ①和②两种方法都可以实现。

① torokubi 为 NULL 时，显示“1年1月1日”。

```
SELECT torokubi, shohin_mei, hanbai_tanka,
       SUM (hanbai_tanka) OVER (ORDER BY COALESCE(torokubi, CAST('0001-01-01' AS DATE))) AS current_sum_tanka
FROM Shohin;
```

② torokubi 为 NULL 时，将该记录放在最前显示。

```
SELECT torokubi, shohin_mei, hanbai_tanka,
       SUM (hanbai_tanka) OVER (ORDER BY torokubi NULLS FIRST) AS current_
       _sum_tanka
FROM Shohin;
```

两组答案的结果都如下所示。

torokubi	shohin_mei	hanbai_tanka	current_sum_tanka	
	运动T恤	4000	4000	← NULL 的记录会 显示在最前面
2008-04-28	煎菜板	880	4880	
2009-01-15	高压锅	6800	11680	
2009-09-11	打鸡蛋	500	12180	
2009-09-20	T恤衫	1000	13180	
2009-09-20	菜刀	3000	16180	
2009-09-20	叉子	500	16680	
2009-11-11	微波炉	100	16780	

>> 解答

首先来看一下①,这种方法比较简单。使用 COALESCE 函数可以将 NULL 转换为“1年1月1日(公历)”。这样得到的结果就比其他任何日期都早了(即使同为“1年1月1日”也没有关系)。这种“欺骗”DBMS 的方法恐怕很多读者都想到了吧。这也是在所有 DBMS 中通用的方法。

接下来我们再来看一下②,其中包含了本书并未介绍的使用 NULLS FIRST 选项的方法。通过在 ORDER BY 子句中指定该选项,可以明确地给 DBMS 下达指令,在排序时将 NULL 放在最前面。目前该方法也是在支持窗口函数的 DBMS 中通用的方法。

本书之所以并未提及上述功能,是因为该功能并不是标准 SQL 支持的功能,而是依存于 DBMS 实现的。标准 SQL 对于 NULL 的顺序,只规定要“排列在开头或者末尾”,至于到底是开头还是末尾,以及明确地指定方法,都交给 DBMS 来实现了。

因此,大家需要注意,这些随时都有可能由于某个 DBMS 的需求改变而无法继续使用。

此外,在 SQL Server 中将聚合函数作为窗口函数使用时,由于不能使用 ORDER BY,所以答案中的 SELECT 语句会发生语法错误。