ORACLE

Mc Graw Hill Education

Oracle Press 15 years McGraw-Hill

Oracle Database 11*g*: The Complete Reference

Master the Powerful Features of the Latest Database Release

Oracle Database 11*g*

完全参考手册

(美) Kevin Loney 著

Oracle畅销书作者

刘伟琴 张格仙 译

ORIGINAL • AUTHENTIC Oracle Press ONLY FROM OSBORNE

Mc Graw Hill

清华大学出版社

ORACLE

Oracle Database 11*g*完全参考手册

ORACLE DATABASE 11g

Oracle Database 11*g*: The Complete Reference

Oracle Database 11*g*最佳参考手册

本书全面详细地介绍了Oracle Database 11g的强大功能，阐述了如何使用所有的新增功能和工具，如何执行功能强大的SQL查询，如何编写PL/SQL和SQL*Plus语句，如何使用大对象和对象-关系数据库。通过学习本书，您可以了解如何实现最新的安全措施，如何调优数据库的性能，如何部署网格计算技术。附录部分内容丰富、便予参照，包括Oracle命令、关键字、功能以及函数等。

本书主要内容

- 安装Oracle Database 11g或从早期版本升级
- 创建数据库表、序列、索引、视图和用户账户
- 构造SQL语句、过程、查询和子查询
- 使用虚拟专用数据库和透明数据加密优化安全性
- 使用SQL*Loader和Oracle Data Pump导入和导出数据
- 使用SQL重放、变更管理和缓存结果
- 使用闪回和自动撤消管理功能避免人为错误
- 构建和调整PL/SQL触发器、函数和程序包
- 使用Java、JDBC和XML开发数据库应用程序
- 使用Oracle实时应用群集(RAC)优化可用性和可扩展性

作者简介

Kevin Loney是Oracle数据库设计、开发、管理和调整方面的国际知名专家。作为金融界的一名企业数据库架构师，2002年他被Oracle Magazine提名为年度顾问。他的畅销书包括《Oracle Database 11g DBA手册》、*Oracle Advanced Tuning and Administration*和*Oracle SQL & PL/SQL Annotated Archives*。他也为业界的多种杂志撰写了很多技术文章。他经常以贵宾身份出席在北美和欧洲举办的Oracle用户大会。

本书源代码下载

http://www.OraclePressBooks.com

http://www.tupwk.com.cn/downpage

ORIGINAL • AUTHENTIC Oracle Press ONLY FROM OSBORNE

McGraw-Hill
全球智慧中文化
http://www.mheducation.com

ISBN 978-7-302-22192-0
9 787302 221920 >
定价：158.00元

上架建议：数据库/Oracle
读者信箱：wkservice@vip.163.com
投稿信箱：bookservice@263.net

Oracle Database 11g
完全参考手册

(美) Kevin Loney 著

刘伟琴 张格仙 译

清华大学出版社

北 京

Kevin Loney
Oracle Database 11g: The Complete Reference
EISBN: 978-0-07-159875-0

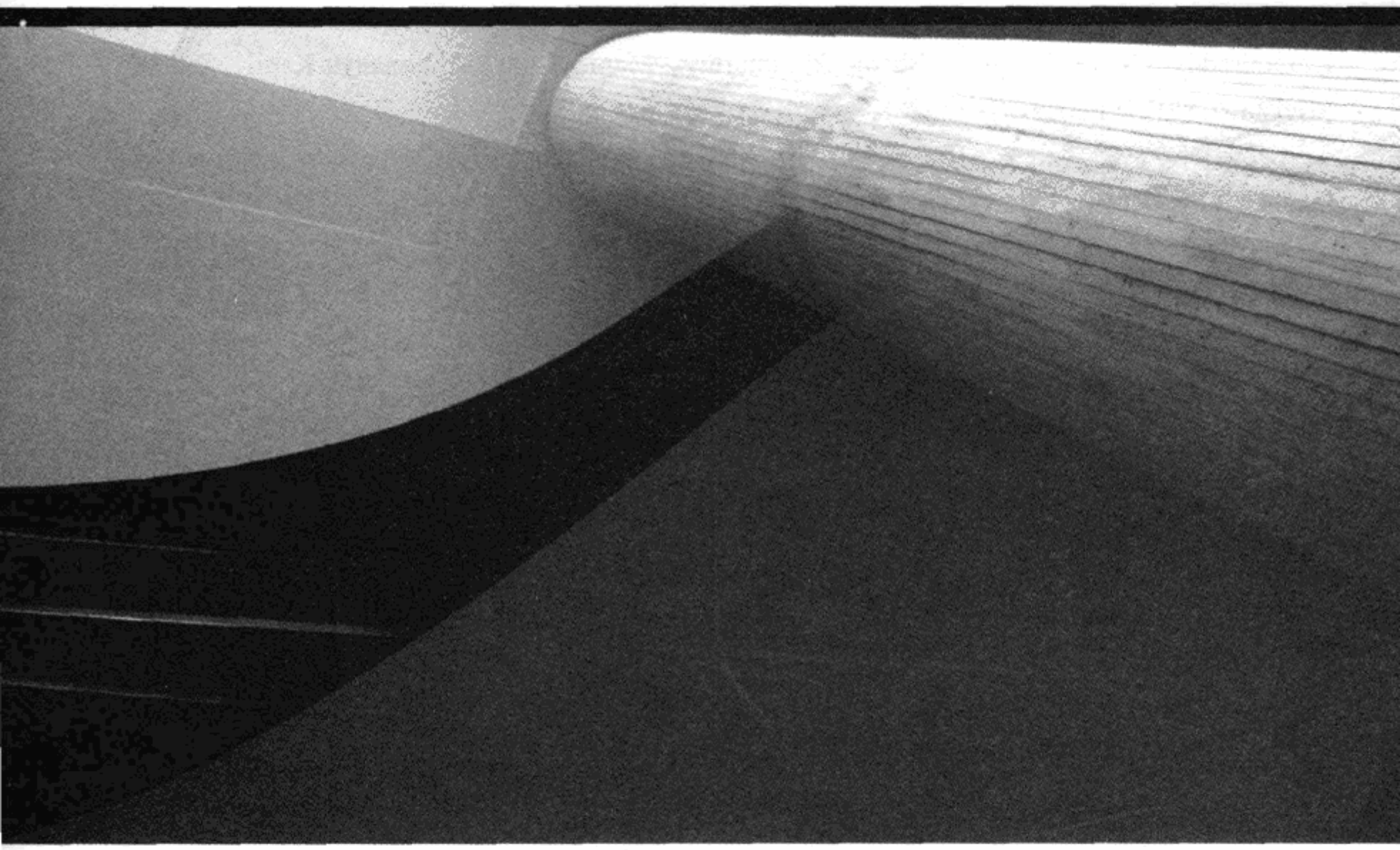
北京市版权局著作权合同登记号　图字：01-2009-2613

图书在版编目(CIP)数据

Oracle Database 11g 完全参考手册/(美)罗尼(Loney, K.)著；刘伟琴，张格仙 译. —北京：清华大学出版社，2010. 6
书名原文：Oracle Database 11g: The Complete Reference
ISBN 978-7-302-22192-0
Ⅰ. ①O… Ⅱ. ①罗… ②刘… ③张… Ⅲ. ①关系数据库—数据库管理系统，Oracle 11g—手册
Ⅳ. ①TP311.138-62
中国版本图书馆 CIP 数据核字(2010)第 034811 号

责任编辑：王　军　谢晓芳
装帧设计：孔祥丰
责任校对：成凤进
责任印制：孟凡玉
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：78.75　**字　数**：1867 千字
版　　次：2010 年 6 月第 1 版　　**印　　次**：2010 年 6 月第1 次印刷
印　　数：1～4000
定　　价：158.00 元

产品编号：030223-01



致　谢

很多人参与了本书的创作过程，并给与了我大力的支持。我的家人、同事和读者都是我开始撰写本书的动力源泉，而且由他们见证了本书的完成。我深深地感谢本书的编辑和出版人员，他们包容了我对本书进度的拖延，并一直关注到本书交稿。

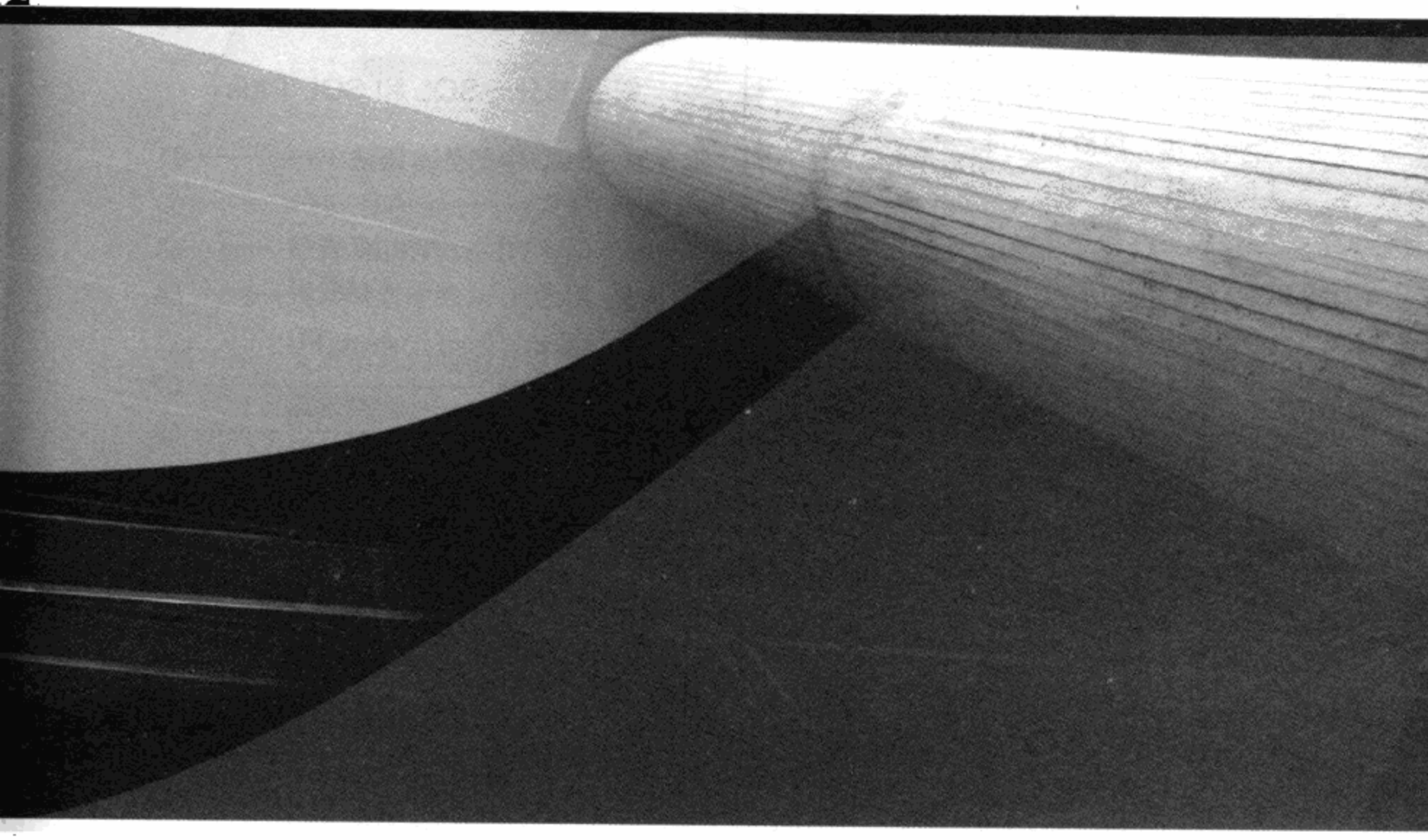
感谢 McGraw-Hill 出版公司的 Lisa McClain、Scott Rogers、Laura Stone、Jennifer Housh、Mandy Canales、Bart Reed、Apollo Publishing Services、Paul Tyler、Marian Selig 和 Jack Lewis 对本书所做的工作和大力支持。特别要感谢技术编辑 Scott Gossett 和 Sreekanth Chintala。

在工作中，我有幸得到同事的认可和支持。同事为本书的更新给予了帮助和支持，尤其是 Joyce Walsh、Brian Albert、Rich Menuchi、Earl Patterson 和 John Bauer。感谢 Phil Steitz、Bernie McGarrigle、Susan Terranova、Robert Brown、Linda Heckert、Susan St. Claire 和在本书创作过程给予我鼓励的其他人，感谢他们的指导和支持。我所在的团队高度的敬业精神和一丝不苟的工作态度时刻感染着我，从而使我能够专心致志地从事这一项目——感谢 Alex

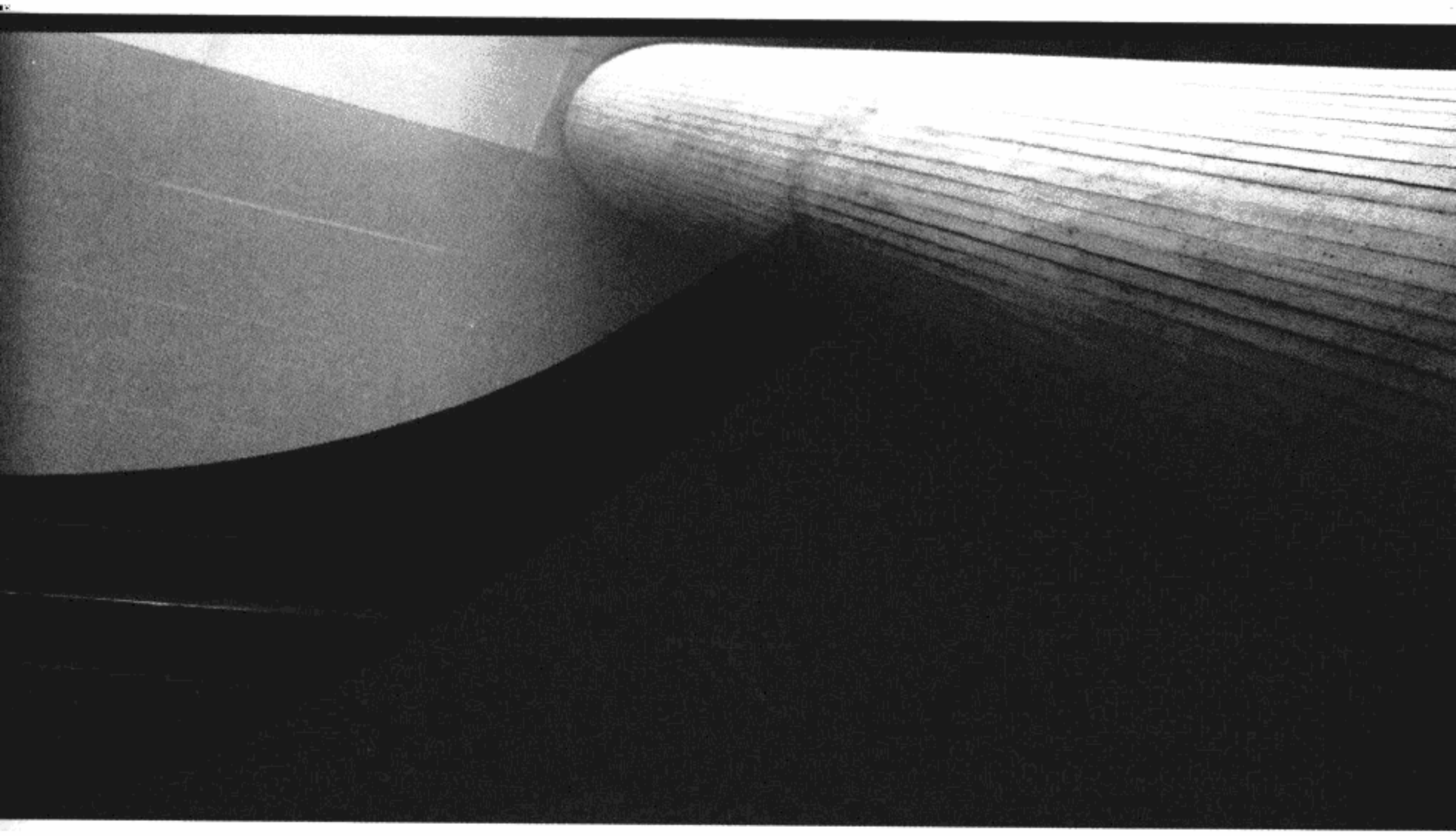
Yankelevich、Kapil Ladha、Tony Price、Rati Mishra、Mike Connolly、Dave Hansen、Uday Kommireddy、Noyal Thomas、Bhanu Thirumurthy、Manish Tare、Abhimanyu Kapil、Saurabh Srivistava 和 Eric Felice。

感谢创建、管理和支持用户组的所有人，包括 TCOUG 的 Monica Penshorn、NEOUG 的 David Teplow 和 RAC SIG 的 Dan Norris，他们为其成员提供了很好的服务。感谢具有无私职业操守的这些人，他们花时间创建、管理、支持和指导用户组、简报、新闻组、书籍和会议。能成为他们的伙伴深感荣幸。

感谢我的家人和所有朋友，你们使我感到撰写本书的过程非常值得。如果由于我的疏忽而未在此处列出您的名字，请相信我仍然将你们铭记于心。

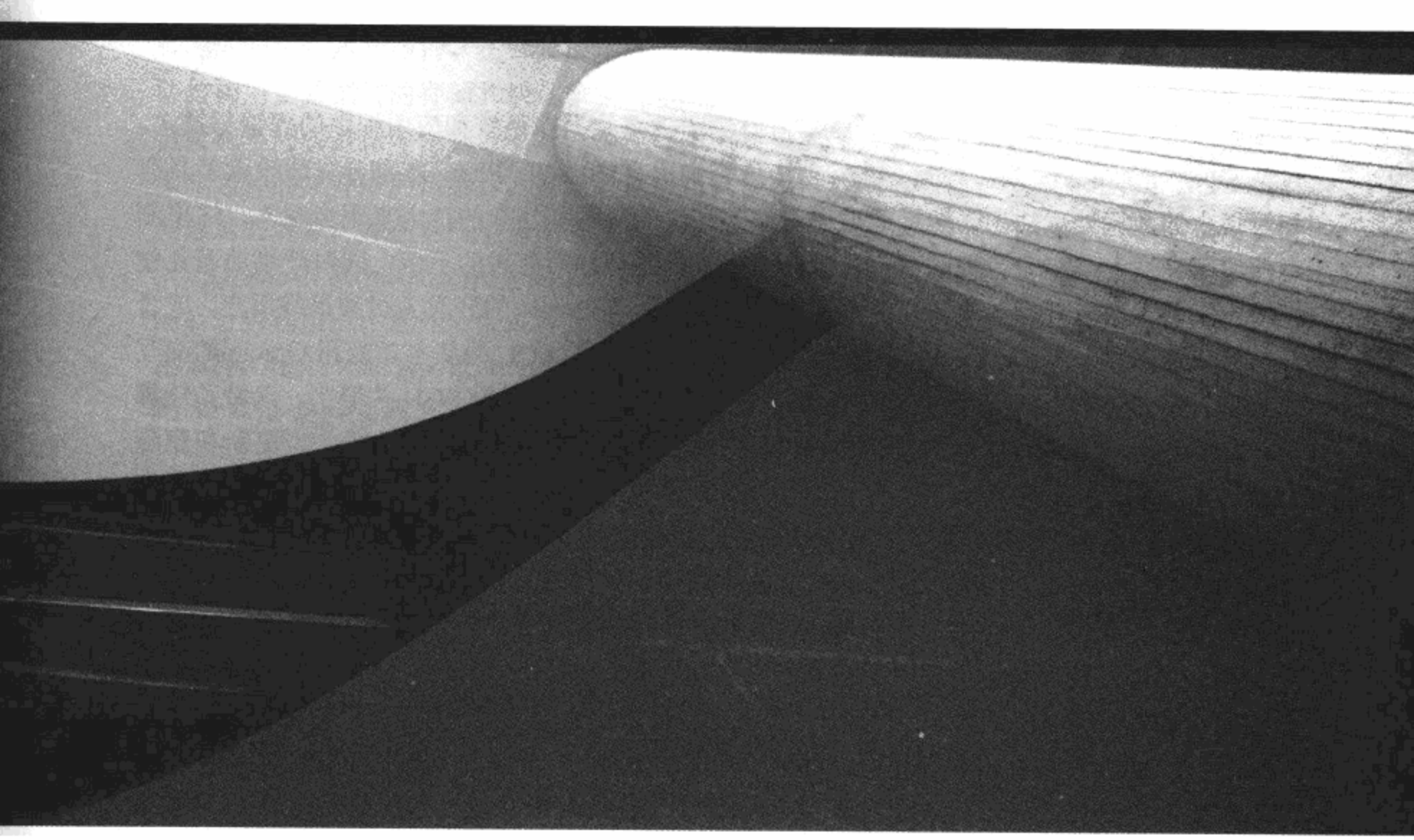


目　录



第 I 部分

关键的数据库概念



第1章

Oracle Database 11g 体系结构

与 Oracle 之前的版本相比，Oracle Database 11g 进行了有意义的升级。新增加的功能使得开发人员、数据库管理员和最终用户能够更好地控制数据的存储、处理和检索。本章将介绍 Oracle Database 11g 体系结构的重要部分。后面的章节将详细讨论这些新的功能，包括 SQL 重放、变更管理和结果缓存等内容。本章的目标是从总体上介绍在 Oracle 应用程序中可以使用的功能以及后面描述这些功能的章节。

本书主要分为 8 个部分。

第 I 部分名为“关键的数据库概念”，概述了不同的 Oracle Database 11g 选项、怎样安装 Oracle 软件、怎样创建或升级数据库，以及怎样规划应用程序的实现。第 I 部分中的这几章形成了一个通用词汇表，这个词汇表帮助最终用户和开发人员连贯巧妙地共享相同的概念，

并确保任何开发工作的成功。本章和第 4 章针对 Oracle 的开发人员和最终用户；第 2~3 章针对数据库管理员。

第 II 部分名为“SQL 和 SQL*Plus”，介绍了关系数据库系统和应用程序的理论与技术，包括 SQL(Structured Query Language，结构化查询语言)和 SQL*Plus。本部分先假设读者具有相对很少的数据处理知识，然后通过一些非常高深的问题和复杂的技术来逐步深入。这里刻意使用了简洁而且符合习惯的语言，同时给出了一些独特而有趣的示例；同时严格地避免使用未定义的术语或行话。该部分主要是针对刚开始学习 Oracle 的开发人员和最终用户，或者是需要快速回顾一下 Oracle 某些功能的读者。逐步介绍了 SQL 的基本功能和 Oracle 的交互式查询工具 SQL*Plus。阅读完这部分后，您应该完全掌握所有的 SQL 关键字、函数和运算符。在 Oracle 数据库中，您应该能够编写复杂的查询、创建数据表以及插入、更新和删除数据。

第III部分名为“高级主题”，讨论了 Oracle 的高级选项，包括虚拟专用数据库(virtual private database)、Data Pump、复制(replication)、文本索引、外部表、变更重放，以及开发人员与数据库管理人员应该如何使用闪回选项。这一部分介绍的大部分功能，最终用户可能都不会直接用到，但他们使用的应用程序却是基于这些功能的。

第IV部分名为“PL/SQL”，讨论了 PL/SQL。讨论的主题包括回顾 PL/SQL 的结构，以及触发器、存储过程和包。本部分同时讨论了标准的 PL/SQL 和本地动态的 PL/SQL。

第 V 部分名为“对象关系数据库”，深入讨论了面向对象功能，如抽象数据类型、方法、对象视图、对象表、嵌套表、可变数组和大对象。

第VI部分名为“Oracle 中的 Java”，讨论了 Oracle 数据库中的 Java 功能。这一部分概述了 Java 的语法以及关于 JDBC 和 Java 存储过程的章节。

第VII部分名为“指南”，概述了 Oracle Database 11g 中的实时应用群集(Real Application Cluster)和可用的网格体系结构以及使用 Oracle 调整工具方面的案例分析，还介绍了客户端高速缓存等新特性，以及数据库管理和如何在 Oracle 中使用 XML。

第VIII部分名为“命令和术语参考”，是 Oracle 服务器的参考——它本身就可构成一本书。阅读这个参考的介绍性内容有助于提高这部分内容的阅读效率，而且能够更好地理解它。该部分包含了 Oracle 的大多数主要命令、关键字、产品、功能和函数的参考，各个主题之间带有大量交叉引用。虽然该参考是为 Oracle 开发人员和用户准备的，但是要求读者对相应的产品比较熟悉。为了能以最高的效率使用该参考中的任意一个条目，阅读其介绍性内容是完全值得的。这些介绍性内容非常详细地说明了参考中包含了什么内容，不包含什么内容，以及怎样阅读这些条目。

在 www.oraclepressbooks.com 网站的下载页面上，包含了本书中用到的所有数据表的表创建语句和行插入内容。正在学习 Oracle 的任何人将这些数据表下载到自己的 Oracle ID 或实际 ID 上之后，就可以方便地试验或扩展这些范例了。

1.1 数据库和实例

每一个 Oracle 数据库都是一个数据的集合，这些数据包含在一个或多个文件中。数据库

有物理和逻辑两种结构。在开发应用程序的过程中，您会创建诸如表和索引这样的结构，这些结构用于数据行的存储和快速检索。可以为对象的名称创建同义词(synonym)，通过数据库链接在不同的数据库中查看对象，并能够限制对象的访问权限。甚至可以使用外部表访问数据库之外的文件中的数据行，其效果就像访问表中的数据行一样。在本书中，您将看到如何创建这些对象，并在这些对象的基础上开发应用程序。

Oracle 实例(instance)由命名为系统全局区(System Global Area，SGA)的内存区域和相应的后台进程组成，这些后台进程负责 SGA 和数据库磁盘文件之间的交互。在 Oracle 实时应用集群(Oracle Real Application Cluster，RAC)中，会有多个实例同时使用同一个数据库(参考第 50 章)。这些实例通常位于不同的服务器上，这些服务器保持高速互连(interconnect)。

1.2　数据库技术

在 Oracle 数据库中，基本的结构是表。Oracle Database 11g 支持多种类型的表，包括如下几种：

- **关系表(relational table)**　使用 Oracle 提供的数据类型(请参考“命令和术语参考”中的“数据类型”)，可以创建存储行数据的表，这些行是由应用程序插入和操作的。表中包含列定义，可以根据应用程序的需求变化添加或删除列。创建表使用 create table 命令。
- **对象关系表(object-relational table)**　为了充分利用诸如类型继承等功能，可以使用 Oracle 的对象关系功能。可以定义自己的数据类型，并在列定义、对象表、嵌套表、可变数组等其他的地方使用这些数据类型。详细内容请参考本书的第 V 部分。
- **索引组织表(index-organized table)**　可以创建一个表，这个表把数据存储在一个索引结构中，使得表中的数据根据索引的值排列。详细内容请参考第 17 章。
- **外部表(external table)**　可以把存储在平面文件中的数据看作是一个表，用户可以对其直接进行查询，并在查询中把它和其他表关联起来。可以使用外部表，在不需要把大量数据导入到数据库中的情况下就可以对这些数据进行访问。请参考第 28 章。注意，Oracle 另外还支持 BFILE 数据类型，它是一个指向外部二进制文件的指针。在创建一个 BFILE 或外部表之前，必须在 Oracle 中(通过 create directory 命令)创建一个目录别名，使这个目录指向外部文件的物理位置。关于 BFILE 和其他大对象数据类型的详细内容，请参考第 40 章。
- **分区表(partitioned table)**　可以把一个表分成多个部分，并单独管理表的每一个部分。可以向表添加新的分区、拆分已经存在的分区并在独立于其他分区的情况下管理某个分区。对表进行分区能够简化维护操作，或者能提高用户查询的效率。可以按照一定的数据范围、数据列表、列数据的哈希值或者这几个条件的某种组合来对表进行分区。详细内容请参考第 18 章。
- **物化视图(materialized view)**　物化视图是由查询检索到的数据的一个副本。用户查询可能会被重定向到物化视图，以避免在查询执行期间访问大型表——Oracle 优化

程序会自动重写查询。您可以创建并管理刷新时间表以便让物化视图中的数据对业务需求保持必要的更新。详细内容请参考第 26 章。

- **临时表(temporary table)** 可以使用 create global temporary table 命令创建一个表，该表允许多个用户在其中插入数据。每个用户只能看到他自己在表中插入的数据。详细内容请参考第 14 章。
- **群集表(clustered table)** 如果两个表通常会被一起查询，那么可以通过群集(cluster)结构把它们存储到物理相邻的位置上。详细内容请参考第 17 章。
- **删除的表(Dropped table)** 可以通过 flashback table to before drop 命令迅速恢复已删除的表。您可以一次把多个表和整个数据库恢复到某个时间点之前的状态。Oracle 支持闪回查询(flashback query)，这种查询将返回表以前某个版本中的数据。

为了访问表，可以使用视图。视图能够处理连接(join)和聚集(aggregation)，限制返回的数据行数，或者更改显示的列。视图可以是只读的，也可以是可更新的，而且它们可以引用本地表或远程表。远程表可以通过数据库链接来访问。可以用同义词隐藏表的物理位置。关于数据库链接的详细内容请参考第 25 章，关于视图的详细内容请参考第 17 章。

为了调整对表的访问，Oracle 支持很多种类型的索引，列举如下：

- **B*树索引(B*-tree index)** B*树索引是 Oracle 中的标准索引类型，它对于选择符合某个等式条件或范围条件的数据非常有用。这种索引由 create index 命令创建。
- **位图索引(bitmap index)** 对于只有极少几个值的列，位图索引也许能够提高查询性能。位图索引只应该在数据被批量加载(对于很多数据仓库或报表应用程序)时使用。
- **反转键索引(reverse key index)** 如果在插入连续数据时会牵涉到 I/O 操作，Oracle 会在存储数据之前动态地反转已经按索引排序好的数据。
- **基于函数的索引(function-based index)** 除了以一列作为索引，如 Name，还可以把基于函数的列作为索引，如 UPPER(Name)。这个基于函数的索引可以为 Oracle 优化器在选择执行路径时提供更多的选择。
- **分区索引(partitioned index)** 可以对索引分区，以支持分区表或者简化索引管理。索引分区可以只作用于表分区，也可以作用于表中的所有行。
- **文本索引(text index)** 可以以文本数据为索引，以支持高级的搜索功能，如扩展单词词干或搜索短语。文本索引是由 Oracle 维护的一系列表和索引值，这些表和索引值能够满足复杂的文本搜索需要。Oracle Database 11g 对文本索引进行了改进，使它的管理和维护变得更简单。

关于上面列出的索引类型(除了文本索引以外)的详细信息，请参考第 17 章和第 46 章。关于文本索引请参考第 27 章。

1.2.1 存储数据

所有这些逻辑结构都必须存储在数据库中的某个地方。Oracle 维护一个数据字典(参考第 45 章)，这个字典中记录了与所有对象(对象所有者、定义、相关的权限等)有关的元数据。对于需要自己的物理空间来存储的对象，Oracle 会在一个表空间(tablespace)中为其分配空间。

1. 表空间

表空间由一个或多个数据文件组成；数据文件是表空间的一部分，而且也只能是一个表空间的一部分。Oracle Database 11g 至少为每个数据库创建两个表空间：SYSTEM 和 SYSAUX，以支持其内部管理的需要。可以使用 Oracle 托管文件(OMF，Oracle managed file)简化数据文件的创建和维护。

您可以创建一种特殊的表空间，称为 bigfile 表空间，其大小可达几千个 TB。借助于 OMF，对 bigfile 的管理使得对表空间的管理对 DBA 完全透明；DBA 可以把表空间作为一个整体来管理，而不用担心空间的大小和底层数据文件的各种结构。

如果一个表空间被指定为临时表空间，则这个表空间本身是永久性的，但是保存在它里面的数据段是临时的。Oracle 使用临时表空间来支持诸如创建索引和连接处理这样的排序操作。临时数据段和永久对象不应该存储在相同的表空间中。

表空间可以按字典的方式进行管理，也可以按本机的方式进行管理。在按字典方式进行管理的表空间中，空间的管理记录在数据字典中。在按本机方式(默认方式)进行管理的表空间中，Oracle 在表空间的每个数据文件中都维护了一个位图，这个位图用于跟踪可用空间的大小。只有存储限额(quota)是在数据字典中进行管理的，这极大地减少了对数据字典表的争用。

2. 自动存储管理

自动存储管理(automatic storage management，ASM)功能将自动完成对数据库所使用的数据文件和其他操作系统级别文件的分布规划，把这些文件分配到所有可用的磁盘空间中。当在 ASM 实例中添加了新的磁盘时，Oracle 会自动在定义好的磁盘阵列的所有磁盘上重新分配数据库文件，以获得更优的性能。ASM 实例的多路复用功能使得数据丢失的可能性降到最低，而且比手工管理关键性文件并把备份保存到不同的物理磁盘上要高效得多。请参考第 51 章。

3. 自动撤消管理

为了支持事务，Oracle 能够动态地创建和管理撤消数据段(undo segment)，这个数据段有助于维护数据块和数据行在修改前的镜像。以前查询过被您修改的数据行的用户仍将看到原来的数据行，就像他们开始查询的时候一样。自动撤消管理(automatic undo management，AUM)使得 Oracle 可以直接管理撤消数据段，而无需数据库管理员进行干预。另外，使用 AUM 还简化了闪回查询的使用。您可以执行闪回查询来查看在一定时间间隔内对数据所进行的各种修改。关于撤消数据段、闪回查询和闪回版本查询的更多内容，请参考第 29 章和第 30 章。

4. 删除的数据

Oracle Database 10g 中新增的回收站(recycle bin)概念使得表空间和数据文件对空间的需求发生了变化。删除一个表的默认动作是保留为它分配的空间；可以通过 RECYCLEBIN 数据字典视图查看它的空间使用情况。如果两次创建一个表并把它删除掉，那么回收站中将会

出现这个表的两个副本。虽然这种体系结构大大简化了恢复偶然删除的数据表的恢复，但是同时也显著地增加了数据库所使用的空间。使用 purge 命令可以删除回收站中原来的内容。查看附录 A 可以了解 purge 命令的语法。

1.2.2 数据保护

您可以完全控制数据的访问权限。可以授予其他用户对对象执行特定操作(如 select、insert 等)的权限。您还可以进行更进一步的授权操作。可以对角色授权，然后再把角色授予用户，从而将权限分成可管理的组。

Oracle 支持一系列非常详细的权限级别；您可以控制哪些行可以访问和在审核期间哪些行将触发审核事件，以便把事件记录下来。如果使用虚拟专用数据库(Virtual Private Database，VPD)选项，那么用户的表查询始终都受到限制，而不管他们用什么方法访问表。

您可以对敏感数据启用列屏蔽功能，也可以对存储在磁盘上的数据进行加密。关于 VPD 实现的详细内容，请参考第 20 章。

除了保护数据访问的安全以外，您还可以审核数据库中的各种事件。可以审核的事件包括授权的操作(如创建用户)、修改数据结构和访问特定的行和表。

1.2.3 可编程的结构

Oracle 支持很多可编程的访问方法。SQL 语言对所有应用程序的开发工作都非常重要，关于它的详细讨论将贯穿在本书中。其他的访问方法列举如下：

- **PL/SQL** 如本书第Ⅳ部分所描述的，PL/SQL 对大多数应用程序的实现都是非常关键的。可以使用 PL/SQL 创建存储过程和函数，而且可以在查询中调用创建的函数。过程和函数可以集中放到程序包中。另外还可以创建触发器，用于告诉数据库当在数据库内部发生不同的事件时应分别采取什么样的步骤。触发器可能会在某些数据库事件(如启动数据库)发生时触发，也可能在对数据行或数据结构进行修改(例如试图删除一个表)时触发。无论哪种情况，您都将在触发事件发生时使用 PL/SQL 来控制数据库或应用程序的行为。
- **动态 SQL** 可以在程序运行的时候产生 SQL 命令并把这些命令传递给过程，然后通过动态 SQL 执行该过程。详细内容请参考第 36 章。
- **SQL*Plus** 正如本书所述，SQL*Plus 为 Oracle 数据库提供了一个简单的接口。SQL*Plus 可以很好地满足基本的报表需求，但是它在脚本编程方面的功能更突出。它为从数据字典中检索数据和创建数据库对象提供了一个统一的接口。
- **Java 和 JDBC** 您将在本书的第Ⅵ部分看到，Oracle 对 Java 和 JDBC 的支持使得您可以用 Java 取代 PL/SQL 来完成很多操作。您甚至可以编写基于 Java 的存储过程。Oracle 中 Java 的功能正随着新版本的发布不断得以扩展和加强。
- **XML** 如第 52 章所述，可以使用 Oracle 的 XML 接口和 XML 类型通过 XML 执行数据的插入和检索。
- **面向对象的 SQL 和 PL/SQL** 可以使用 Oracle 创建和访问面向对象的结构，包括用户定义的数据类型、方法、大对象(LOB)、对象表和嵌套表。详细内容参考第Ⅴ部分。

- **Data Pump**　Oracle Database 10g 中新增的 Data Pump Import 和 Data Pump Export 这两个功能大大增强了原来使用的 Import 实用程序和 Export 实用程序的可管理性和性能。可以在修改模式和数据的时候使用 Data Pump 把数据库中的数据快速提取出来并存储到其他数据库中。关于使用 Data Pump 的详细内容请参考第 24 章。
- **SQL*Loader**　可以使用 SQL*Loader 把平面文件快速加载到 Oracle 数据表中。可以在一次加载过程中把单个平面文件加载到多个数据表中，而且加载是并行完成的。详细内容参考第 23 章。
- **外部程序和过程**　可以把 SQL 嵌入到外部程序中，或者创建过程库，以后可以将这个库链接到 Oracle。详细内容参考第 35 章。
- **UTL_MAIL**　UTL_MAIL 是 Oracle Database 10g 中才引入的一个程序包，它允许 PL/SQL 应用程序开发人员直接发送电子邮件，而不用了解如何使用底层的 SMTP 协议栈。

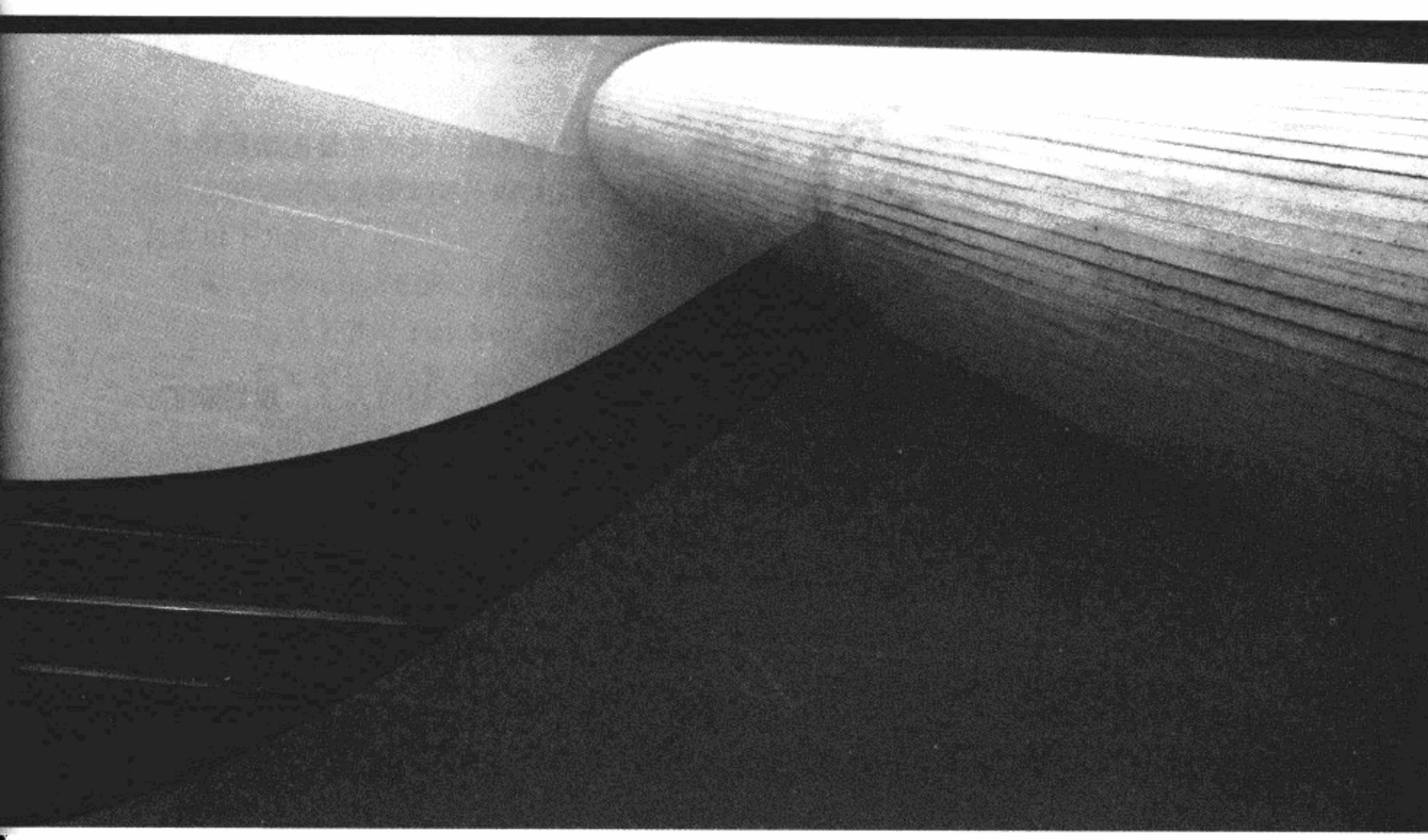
1.3　选择体系结构和选项

Oracle 为开发基于 Oracle Database 11g 的应用程序提供了一整套非常有用的工具。您可以使用 Oracle Database 11g 引入的很多功能，而与您所选择的应用程序体系结构没有关系。

如果您曾经用 Oracle 以前的版本实现过应用程序，那么应该检查一下您的数据库，并找出 Oracle Database 11g 中的新功能可对应用程序加以改进的地方。例如，如果曾实现过物化视图，那么也许可以利用扩展物化视图增量(fast 选项)刷新的新功能。Oracle 提供了一系列的过程，用以帮助您管理物化视图的刷新时间表。例如，您可以执行一个过程来生成描述信息，给出刷新可能性和妨碍您使用最快速选项的配置问题。您可以根据一组示例查询使用 Oracle 提供的另一个过程来生成调整物化视图结构的推荐配置。

有一些新功能稍微发生了一些变化，但这些变化对应用程序或编写代码的方法产生的影响却很大。例如，可以使用变更重放功能来捕获在一个数据库中执行的命令，并在另一个数据库中重放这些命令。一些有意义的新功能还包括“不可见”索引、简化的表维护以及版本对象。应该根据可用的新特性重新评估应用程序原有的体系结构。

在下面的几章中，您将了解到怎样安装 Oracle Database 11g 和怎样把旧版本的数据库升级到 Oracle Database 11g 的数据库。随后几章将简要介绍一下应用程序的规划，然后用大量的篇幅分别介绍 SQL、PL/SQL、Java、面向对象的功能和 XML 的使用方法，从而使您能够最大限度地发挥 Oracle 数据库的作用。随着业务流程的变化，应用程序的体系结构也会不断发生变化。在调整应用程序的体系结构的同时，应该了解 Oracle 数据库最新的功能并确定应用程序如何才能最好地使用它们，以便提高应用程序的功能和性能。

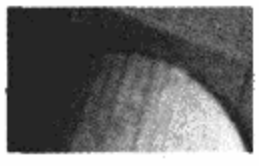


第2章

安装 Oracle Database 11g 和创建数据库

随着新版本的发布，Oracle 的安装程序变得越来越易于使用，您也许很想马上下载安装。如果只是想体验一下数据库的新功能，则也很易于实现；但是如果不希望在一个月之后就得返工甚至重新安装数据库，那就需要好好地规划一下。虽然关于安装 Oracle Database 11g 的全部细节超出了本书的讨论范围，但本书将介绍如何使用 Oracle Universal Installer(OUI)进行基本的安装。无论如何，要想成功地部署 Oracle 数据库，重新仔细阅读一下与您的特定平台

有关的安装指导都是必不可少的一个步骤。



注意：
虽然本章针对数据库管理员初学者，但是因为规划的过程应该包括最终用户、应用程序开发人员和系统管理员的参与，所以关于工作负载和空间需求的数据应该尽量精确。

在开始安装之前应该首先考虑或解决下面这些问题：

- 确定本地数据库的名称，以及由哪个域包含这个数据库。
- 对于使用数据库的第一个项目，估计表和索引的数量以及它们的大小，规划除了 Oracle SYSTEM 表空间及相关的 Oracle 软件和工具以外所需要的磁盘空间。
- 规划物理数据文件在 Oracle 服务器的磁盘上的位置，以便获得最佳的性能和可恢复性。通常，物理磁盘是越多越好。如果数据文件存储到 RAID 或者共享存储区，那么应该考虑使用 Oracle Managed Files 管理数据文件的存放。可以使用自动存储管理(ASM)功能简化存储管理。关于 ASM 的详细内容请参考第 51 章。
- 回顾并理解基本的初始化参数。
- 选择数据库字符集，并选择一个备用字符集。虽然在安装的时候按默认属性设置字符集很容易，但是您可能需要考虑数据库的用户的分布以及他们使用的语言。如果在安装完成之后需要修改字符集，那么所选择的字符集只能是已有的字符集的超集。
- 决定最优的默认数据块大小。默认数据块大小由 DB_BOLCK_SIZE 定义，只有重新安装数据库才能改变这个值。注意 Oracle 在单个数据库中支持多种数据块大小。
- 规划在非 SYSTEM 表空间中存储非 SYSTEM 用户对象。确保为所有非管理员用户分配一个非 SYSTEM 表空间作为他们的默认表空间。
- 规划如何实现自动撤消管理，以简化事务撤消信息的管理。
- 规划一个备份和恢复策略。决定数据库的备份方式和备份频率。规划使用多种方法备份数据库。

熟悉一些关键的 Web 站点是必须的。Oracle 技术网(Oracle Technology Network，OTN)，即 http://otn.oracle.com 网站提供了大量信息，包括白皮书、免费工具、示例代码和在线电子杂志 *Oracle Magazine*。需要在网站上注册，但使用 OTN 是不收费的。您可以从 OTN 网站下载最新版本的 Oracle 软件。

虽然可以先购买一个 Oracle 数据库软件的许可证(license)，但是拥有包含 Web 技术支持的 Oracle 支持合同对成功安装和部署数据库更为关键。使用 Oracle 公司的 Metalink(http://metalink.oracle.com)意味着您也许可以在 Web 浏览器上获得各种帮助，使您的数据库能够良好地启动和运行。在 Metalink 上，您可以提交技术支持请求、搜索其他的技术支持请求、下载补丁和白皮书以及搜索故障数据库(bug database)。

2.1　许可证和安装选项

使用软件的第一步是成功地安装软件。如果不考虑安装 Oracle 的硬件和软件平台，那么您可以选择的安装类型是一样的。虽然在不同的版本之间可选的类型不完全一样，但一般都包含以下几种：

- Enterprise Edition(企业版) 该版本是 Oracle 数据库功能最全、扩展性最好的版本。它包含诸如 Flashback Database(闪回数据库)这样的功能，并允许您添加其他购买了许可证的功能，如 Oracle Spatial、Oracle OLAP、Oracle Label Security 以及 Oracle Data Mining 等。
- Standard Edition(标准版) 该版本提供了 Enterprise Edition 的一个良好的功能子集，通常包含了小公司所需要的功能。
- Personal Edition(个人版) 该版本允许开发将要运行在 Standard Edition 或 Enterprise Edition 上的应用程序。它不能用于开发商业产品。

只有指定的用户或 CPU 能够获得 Oracle 数据库的许可权，而且多个用户不能同时使用同一个许可权。因此，DBA 应该使用初始化参数 LICENSE_MAX_USERS 指定在数据库中可以创建的用户的最大数量。另外，Oracle Management Server(是 Oracle Enterprise Manager(OEM)客户机的后台)可以在安装服务器端或客户机端的时候进行安装。不过，最好还是在数据库的基本安装完成之后再执行这个安装。

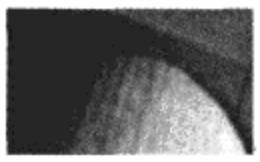
2.2　使用 OUI 安装 Oracle 软件

可以使用 Oracle Universal Installer(OUI)安装和管理 Oracle 服务器端和客户机端的所有组件。也可以使用 OUI 卸载 Oracle 公司的任何产品。

在安装服务器时，您将从前文介绍的安装版本(Enterprise Edition、Standard Edition)或其他在您的平台上可用的安装版本中选择一种 Oracle Database 11g 版本。

最好是在安装的过程中出现创建数据库的提示对话框时创建一个启动器数据库(starter database)。创建启动器数据库可以确保服务器的环境已经正确地设置，而且可以了解 Oracle Database 11g 中的任意新功能。启动器数据库还可以用于存储 OEM 或者 Recovery Manager。

具体的安装流程会由于您的操作环境和 Oracle 版本的不同而有所变化。在 Oracle 软件安装结束时将启动数据库配置助手(Database Configuration Assistant，DBCA)，并启动新的数据库创建过程，此数据库将用在服务器上。



注意：
在 UNIX 环境下，您需要为 DISPLAY 环境变量设置一个合适的值，并在通过 runInstaller 脚本启动 OUI 之前激活 xhost。

启动 OUI 时要求您提供配置信息。如图 2-1 所示，第一个屏幕要求提供 Oracle 软件的基本目录位置、数据库的主目录位置以及安装类型。您也可以选择在 Oracle 软件成功安装之后

创建一个启动器数据库。您需要指定此数据库的名称和密码。

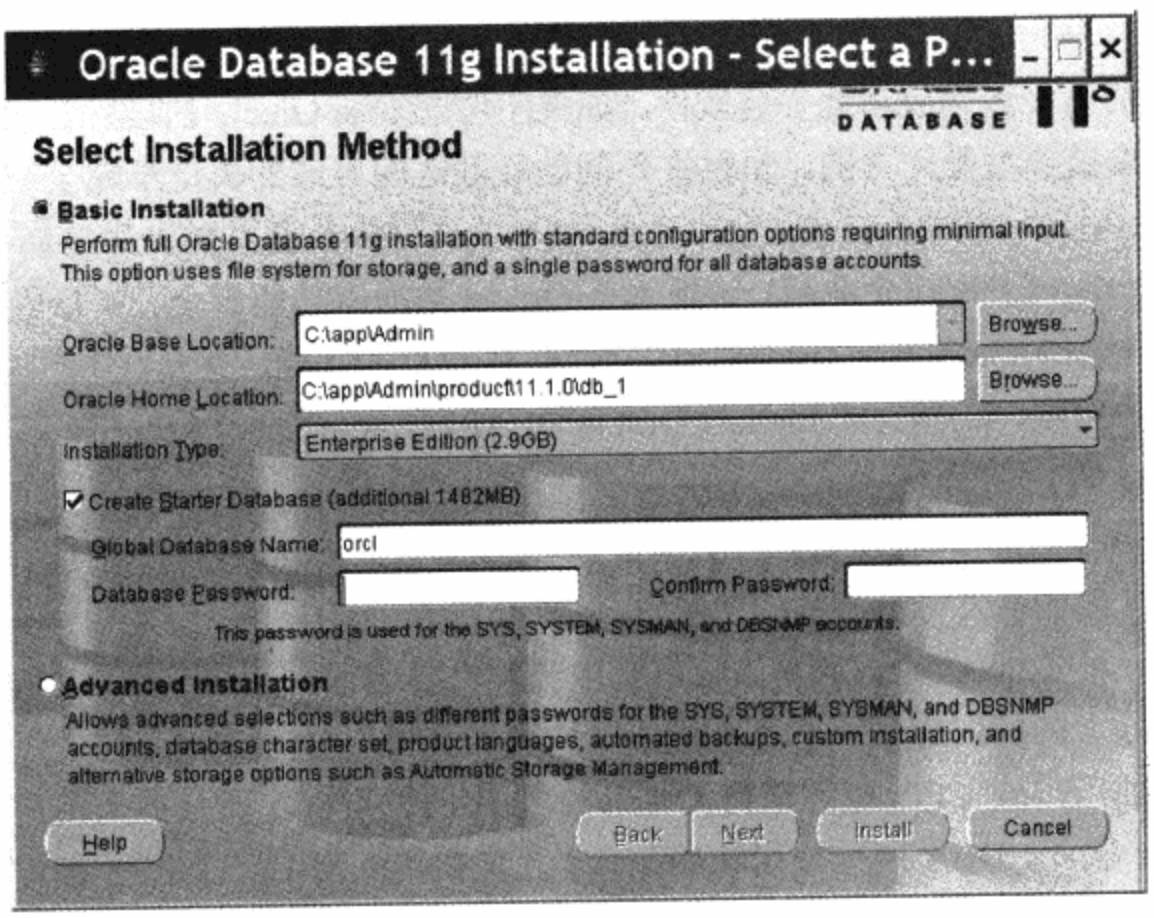


图 2-1 OUI 初始屏幕

接下来 OUI 会执行一系列必要的检查，以确保环境的配置支持 Oracle 安装。这些检查包括基本的网络配置和环境变量设置，如图 2-2 所示。

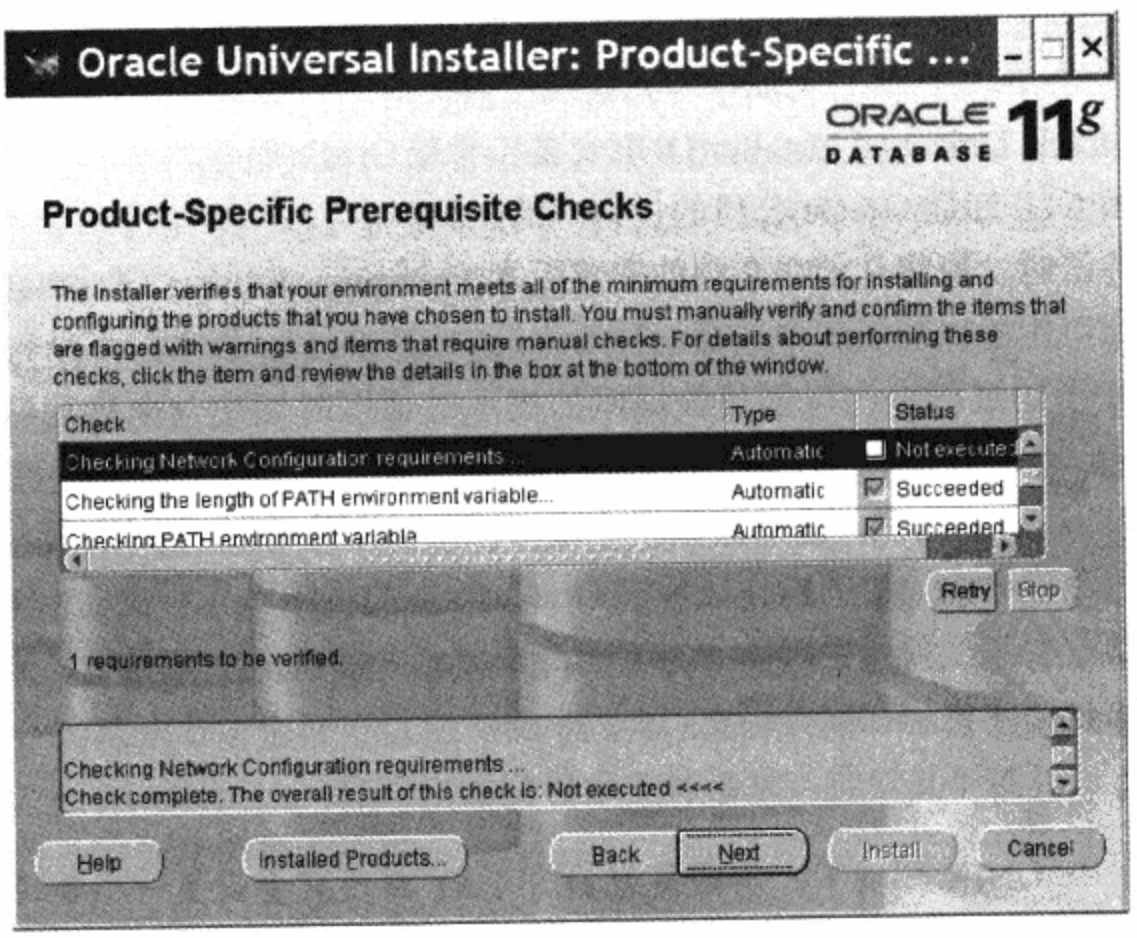


图 2-2 特定于产品必需的检查

可以配置数据库与 Metalink(Oracle 支持)账户相关联。如图 2-3 所示，接下来可以指定使

用的 Metalink 用户名和密码。可以使用 Test Registration 选项来验证您的计算机与 Metalink 站点的连通性。

图 2-3　Oracle 配置管理器注册

此时，安装过程准备继续进行，OUI 会显示选择安装的产品的一个列表。如图 2-4 所示，此列表包括核心软件以及相关的实用程序和脚本。

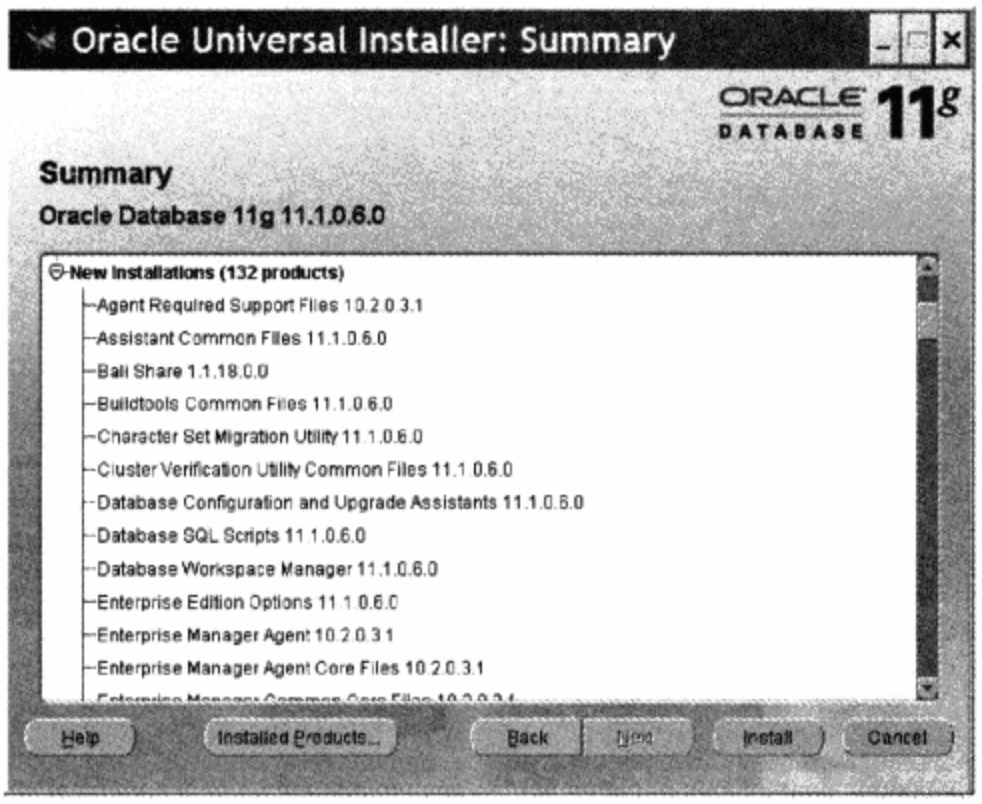


图 2-4　安装产品的列表

现在可以开始安装 Oracle 了。如图 2-5 所示，Oracle 提供一个状态栏来显示安装的进度。安装所需要的时间取决于计算机的处理速度。此时不要在计算机上运行其他进程，因为这些进程可能会影响 Oracle 软件的成功安装。

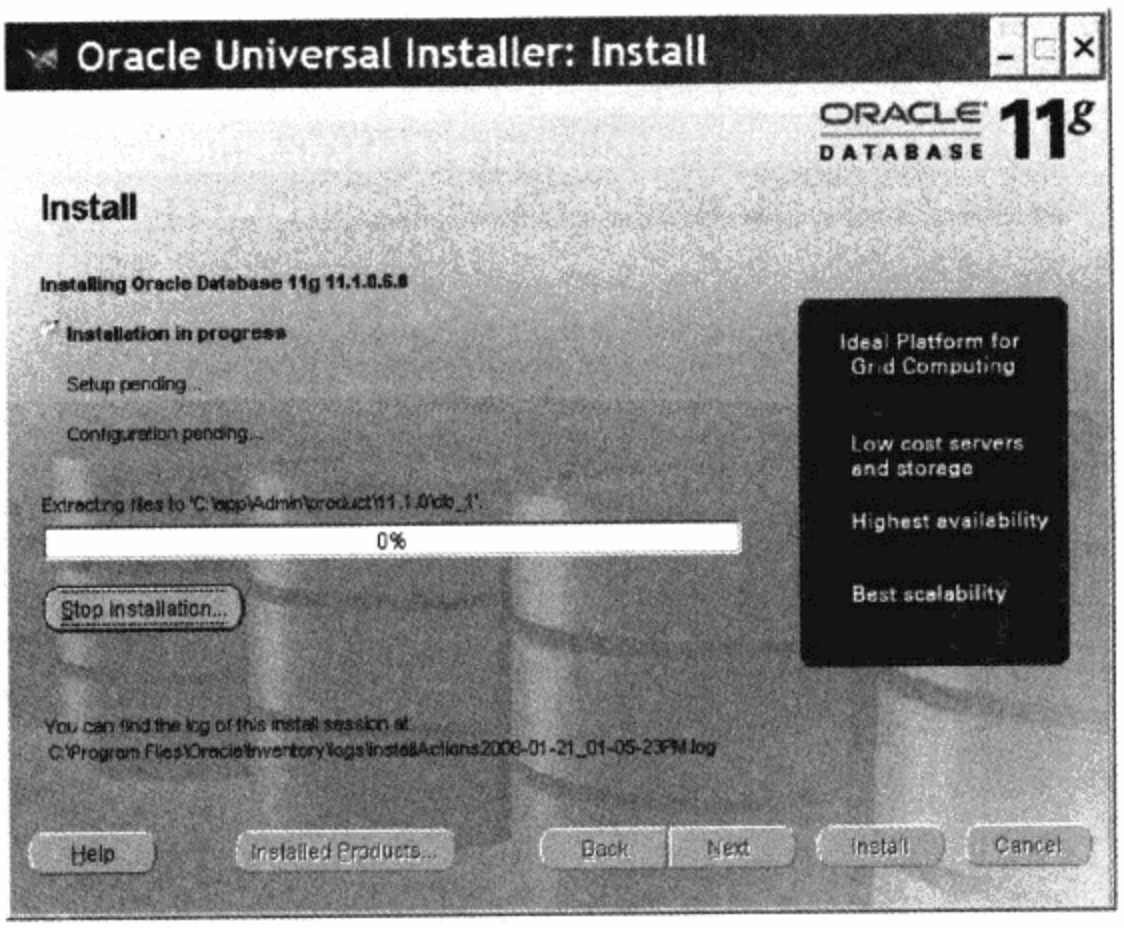


图 2-5 安装过程

Oracle 软件成功安装之后，如果您选择了数据库配置助手这一选项，则数据库配置助手会自动运行。如图 2-6 所示，创建数据库的步骤包括将启动器数据库的数据文件复制到计算机的目标区域，然后创建一个实例。这一步骤完成之后会创建一个功能完整的数据库，可以将这一数据库用于本书的实践练习。

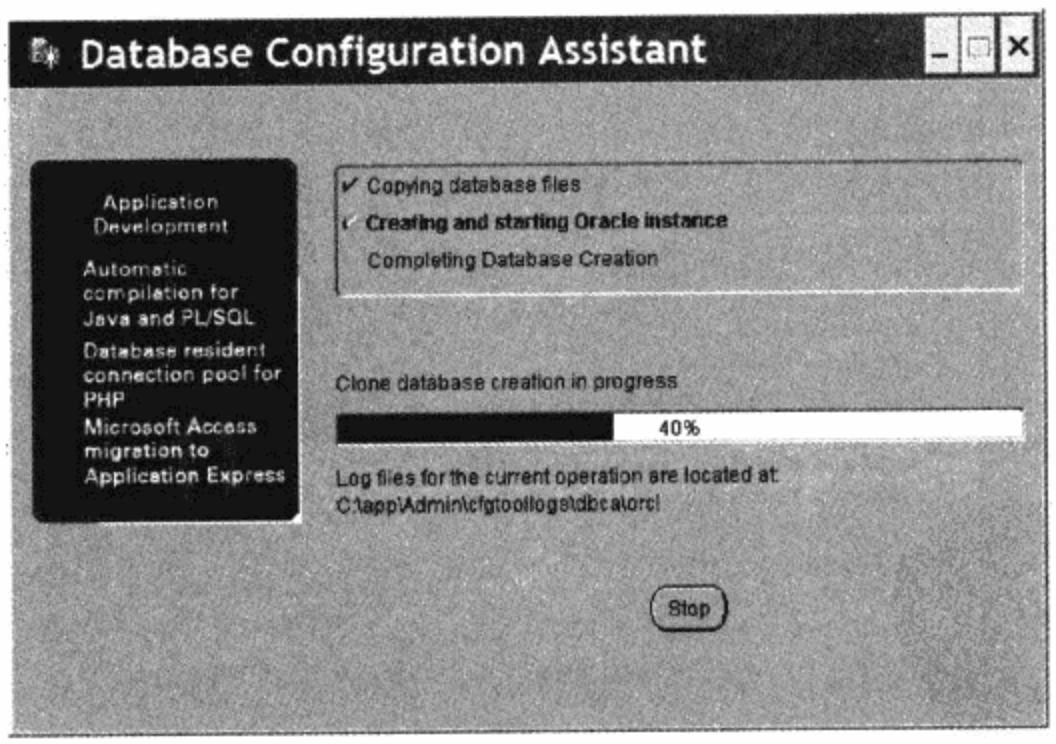


图 2-6 数据库配置助手

如图 2-7 所示，将运行多个配置助手。数据库配置助手(DBCA)创建数据库，而 Oracle Net Configuration Assistant 验证网络配置。到数据库的连接将使用 Oracle Net。您可以选择不运行这些配置助手，也可以选择失败时重试。

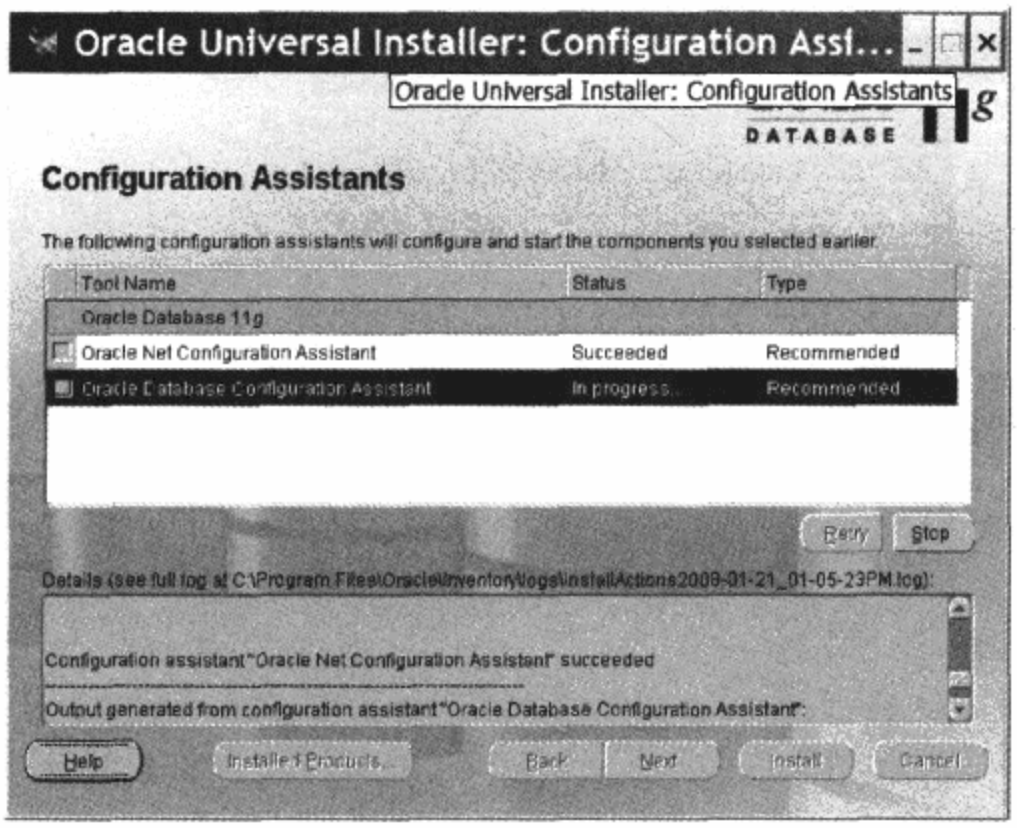


图 2-7　其他的配置助手

验证网络连接配置之后，DBCA 现在可以完成数据库创建了，下图 2-8 所示。

数据库创建完成之后，您将看到一个类似于图 2-9 的小结屏幕。此小结屏幕将列出被创建数据库的名称、数据库参数文件的位置和取消锁定的账户。作为一种安全措施，新的 Oracle 数据库中的大多数账户都被锁定了。取消锁定的账户的密码是在初始创建时设置的那个密码(参见图 2-1)。

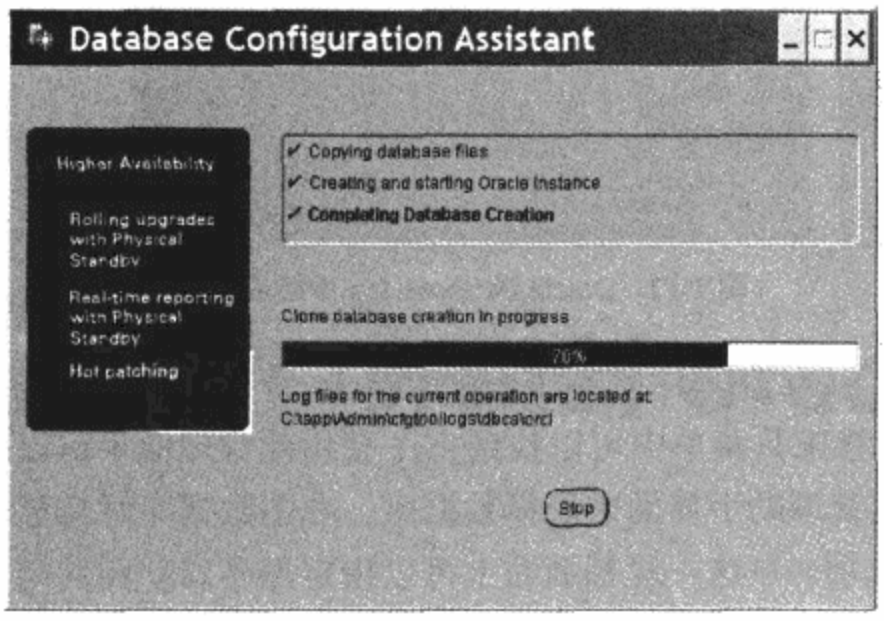


图 2-8　数据库创建过程

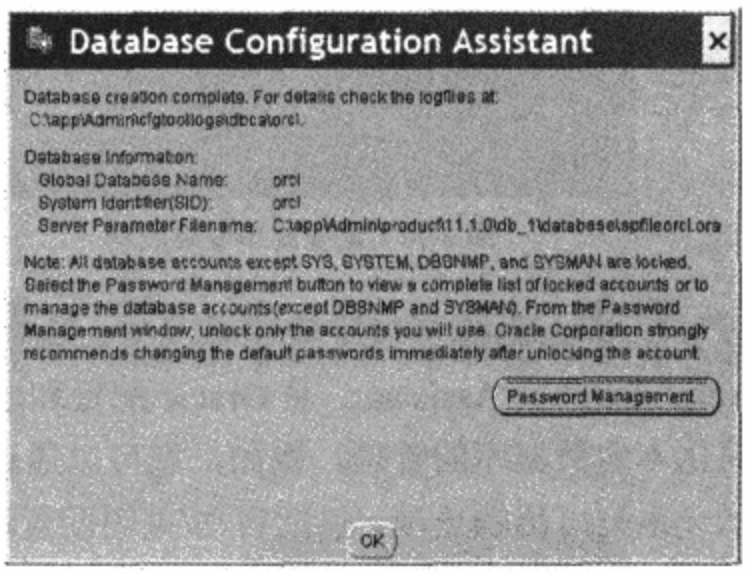


图 2-9　小结屏幕

如果您选择了更改安全设置，则会看到安全管理页面，如图 2-10 所示。您可以选择取消 Oracle 数据库提供的任何标准账户的锁定。您可以为每一个取消锁定的账户指定一个密码。默认情况下，只有取消锁定的账户才能用来管理数据库，如 SYS 和 SYSTEM。

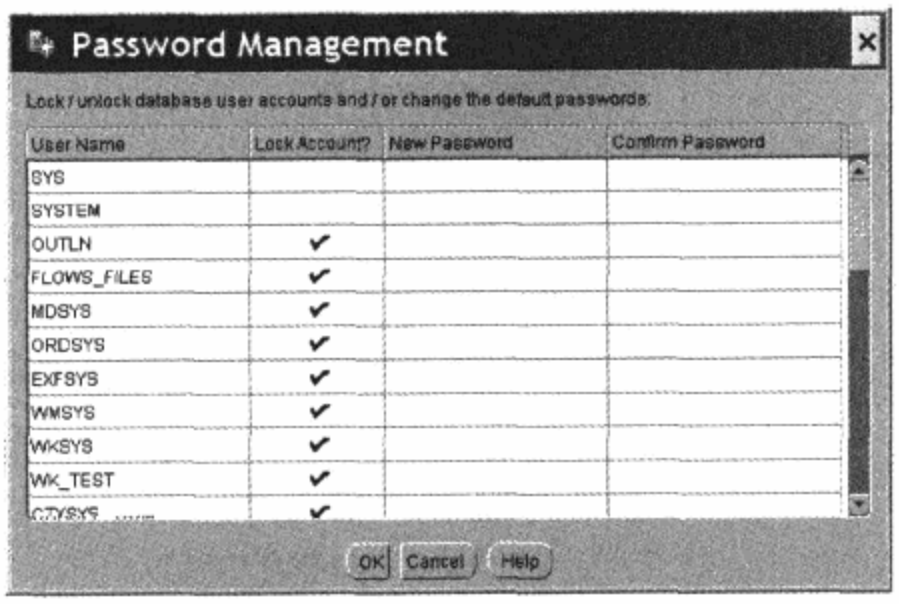


图 2-10 密码管理屏幕

紧接着密码管理屏幕，是最后一组小结屏幕，如图 2-11 和图 2-12 所示，显示单个配置助手成功安装以及整个安装过程成功完成。创建的数据库完全可用，并有一个实例运行在要访问此数据库的本地计算机上。现在可以使用 SQL*Plus 等工具来访问示例数据库了。

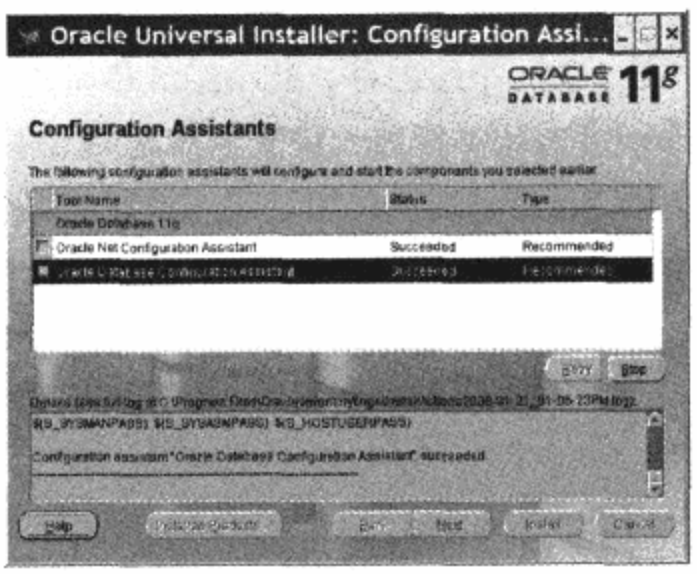


图 2-11 配置助手安装成功

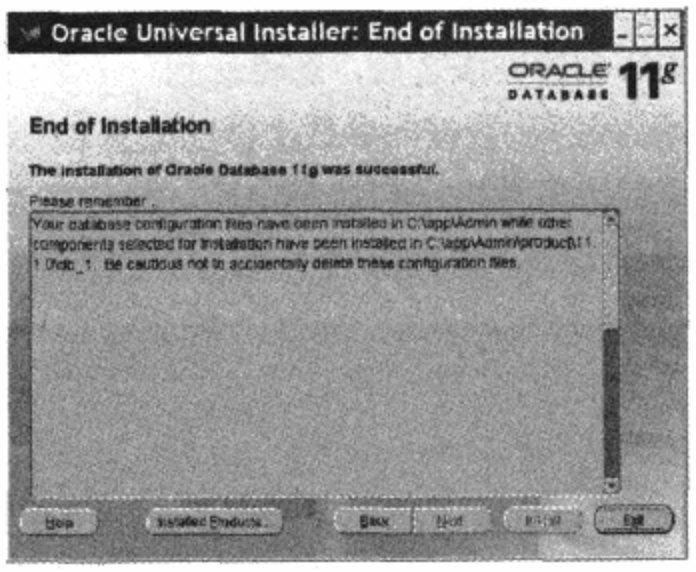


图 2-12 Oracle Database 11g 安装成功

在 Windows 环境中安装时，Oracle 在 Start 菜单中安装了一个 Oracle 管理助手(Oracle Administration Assistant，从 Oracle 配置和迁移工具菜单中可以找到它)。使用此管理助手可以简化本地数据库的管理。例如，您可以通过管理助手界面来选择数据库。右击数据库时会显示一系列可用的选项，包括启动/关闭配置选项。在这一屏幕画面上可以指定每当启动和关闭 Windows 服务时启动和关闭数据库实例，从而简化数据库管理。也可以指定要执行的关闭类型(默认情况下是 Shutdown Normal)。

如果需要以手工方式重新运行 DBCA，则可以通过与管理助手相同的配置和迁移工具菜单结构来完成。建议缺乏经验的数据库管理员(DBA)使用 DBCA。DBA 可以选择使用 DBCA 或执行 create database 命令。create database 命令的语法参见附录 A。

第3章

升级到 Oracle Database 11g

如果您已经安装了 Oracle Database 11g 以前的版本，则可以把它升级到 Oracle Database 11g。Oracle 支持多种升级方法；要根据您当前的 Oracle 软件的版本和数据库的大小等因素来选择恰当的升级方法。本章将详细介绍这些升级方法和它们的使用指导原则。

如果没有使用过 Oracle Database 11g 以前的版本，则您可以跳过这一章。但是如果需要从以前的版本升级到 Oracle Database 11g 或者需要把不同数据库中的数据迁移到 Oracle Database 11g 中，就需要参考本章的内容。

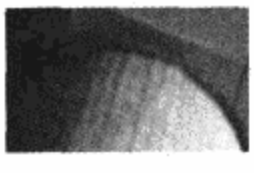
在开始升级之前，应该阅读 Oracle Database 11g 针对您的操作系统的安装指南。成功的安装取决于正确的配置环境——包括操作系统的补丁级别和系统参数设置。与其试图重新开

始没有完全成功的安装，还不如首先计划一下怎样安装和升级。

本章假设您已经成功地完成了 Oracle Database 11g 软件的安装(参考第 2 章)，而且您拥有 Oracle 以前版本的数据库。为了升级数据库，您可以选择如下 4 个选项：

- 使用数据库升级助手(Database Upgrade Assistant)引导和执行原位升级。在这个处理中旧数据库将变成 Oracle 11g 的数据库。
- 手动升级数据库。旧数据库在该过程中将变成 Oracle Database 11g 的数据库。
- 使用 Data Pump Export 实用程序和 Data Pump Import 实用程序把旧版的 Oracle 数据库升级为 Oracle 11g 的数据库。在这个处理过程中将会用到两个数据库——作为导出源的旧数据库，和作为导入目标的新数据库。
- 把 Oracle 以前版本的数据库中的数据复制到 Oracle 11g 的数据库中。在这个处理过程中将会用到两个数据库——作为复制源的旧数据库，和作为复制目标的新数据库。

原位升级数据库(通过 Database Upgrade Assistant 或者手动升级的途径)被称作直接升级。



注意：

直接升级到 Oracle Database 11g 只能从以下几个版本开始：Oracle 9.2.0.4 或更高版本(最好是 9.2.0.8)、10.1.0.2 或更高版本、10.2.0.1 或更高版本。如果使用的是其他版本，那么必须先升级到这几个版本中的某一个，否则就需要使用其他的升级方法。

由于直接升级不需要为正在被升级的数据库创建另一个数据库，因此它完成的速度可能会比较快而且需要的磁盘空间也会比间接升级小。

注意：

请认真规划您的升级过程。在升级到 Oracle Database 11g 之前您可能需要时间进行多次逐步升级(如从 Oracle 9.2.0.3 升级到 9.2.0.8)。

3.1 选择升级方法

如前所述，有两种直接升级方法和两种间接升级方法。本节将详细介绍这些升级方法，并给出使用说明。

一般情况下，直接升级方法的速度最快，因为它们原位升级数据库。其他的方法包括把数据复制到文件系统上的 Data Pump Export 转储文件中，或者通过数据库链接复制数据。对非常大的数据库而言，因为通过间接方法完整地重新创建数据库所需要的时间很长，所以一般不会使用这种方法。

第一种直接升级方法需要借助于 Database Upgrade Assistant(DBUA)完成。DBUA 是一种交互式工具，它将在升级过程中引导您。DBUA 会评估当前数据库的配置并给出在升级过程中执行的推荐修改方案。推荐的方案也许会涉及文件大小的设定和新的 SYSAUX 表空间的设置(如果从 Oracle 10g 的数据库之前的版本进行升级)。如果接受了推荐的方案，则 DBUA

将在后台执行升级而在前台显示一个升级进度条。DBUA 在方法上和 DBCA 非常相似。正如第 2 章所述，DBCA 是一个图形界面，它将引导用户逐步设置所有必需的参数，使升级能够成功完成。

第二种直接升级方法也称手动升级。DBUA 是在后运行脚本，而手动升级却需要数据库管理员自己运行脚本。手动升级提供了很大的控制权，但是这也意味着在升级上增加了更大的风险，因为您必须按正确的顺序执行所有的步骤。

作为一种间接方法，您可以通过 Data Pump Export 工具和 Data Pump Import 工具来升级数据库。使用这种方法时，从旧版的数据库中导出数据然后把这些数据再导入到 Oracle 软件一个新版的数据库中。这个过程中可能需要足够的磁盘空间来保存数据的多个副本——这些副本分别在源数据库中，在转储文件中，以及在目标数据库中。虽然占用了不少磁盘空间，但是这种方法可以灵活地选择迁移哪些数据。可以选择导出特定的表空间、模式、表和行。

使用 Data Pump Export 和 Data Pump Import 方法，原始数据库不会被升级。它的数据被提取出来并迁移到其他地方，而原始数据库可以被删除，或者是和新数据库并行地运行直到新的数据库测试成功。在执行导入/导出的过程中，您可以选择和重新插入数据库中的每一行数据。如果数据库非常大，导入的过程也许会花费很长的时间，这将妨碍您及时地向用户提供升级过的数据库。关于 Data Pump Export 实用程序和 Data Pump Import 实用程序的更多内容请参考第 24 章和第 51 章。

在数据复制方法中，您将使用一系列的 create table as select 或 insert as select 命令，这些命令将通过数据库链接(参考第 25 章)检索源数据。在 Oracle Database 11g 中，表是基于在一个单独的源数据库上的数据查询创建的。这种方法允许您逐步导入数据并限制被迁移的行和列。但是，您必须确保所复制的数据中包含了表之间所有必需的关系。和 Data Pump Export/Import 方法一样，这种方法对大型数据库可能需要很长的时间。

您需要和团队中的技术专家一起讨论需要复制哪些数据，以及迁移数据期间允许数据库停止运行的最长时间，从而选择适当的升级方法。通常情况下，DBUA 适用于大型数据库，而间接方法适用于比较小的数据库。

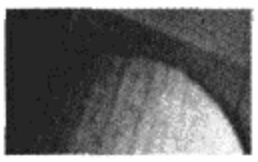
3.2　升级之前的准备

在开始迁移数据库之前，您应该备份现有的数据库和数据库软件。如果由于某种原因迁移失败，而且您无法把数据库或者软件恢复到它以前的状态，那么这时可以恢复以前的备份并重新创建数据库。

应该开发和测试一些脚本，以便在升级完成之后评估数据库的性能和功能。这个评估可能会包括特定的数据库操作的性能或者数据库在较大的用户负载的情况下的总体性能。

在对一个商业数据库进行升级之前，应该先尝试升级一个测试数据库，以免遗漏了某些部件(比如操作系统的补丁)，并能够计算升级所需要的时间。

在进行直接升级之前，应该分析数据字典表。在升级到 Oracle Database 11g 的过程中，如果还没有对数据字典进行分析，那么 Oracle 会先对其进行分析，所以提前进行分析将有助于提高升级的效率。



注意：
升级到 Oracle Database 11g 之后，CONNECT 角色只有 CREATE SESSION 权限，在 Oracle Database 11g 之前版本中授予 CONNECT 角色的其他权限在升级过程中都被取消了。

3.3 运行升级前信息工具(Pre-Upgrade Information Tool)

安装 Oracle 11g 软件之后，在将数据库升级到新版本之前应该对数据库进行检查。对数据库的检查由升级前信息工具(Pre-Upgrade Information Tool)自动完成。如果以手动方式升级数据库，则必须运行升级前信息工具，建议对所有升级运行这一工具。

升级前信息工具是随同 Oracle Database 11g 一同发布的 SQL 脚本，必须将它复制到正被升级的数据库环境中，且从这个环境中运行。完成下面这些步骤来运行升级前信息工具：

(1) 以 Oracle Database 11g 主目录所有者的身份登录系统。

(2) 将升级前信息工具(文件名为 utlu111i.sql)从 ORACLE_HOME/rdbms/admin 目录复制到 Oracle 主目录之外的一个目录中，如系统上的临时目录。记下此文件的新位置。

(3) 以被升级的数据库的 Oracle 主目录所有者的身份登录系统。

(4) 切换到步骤(2)中复制文件的那个目标目录。

(5) 启动 SQL*Plus。

(6) 以具有 SYSDBA 权限的用户身份连接到数据库实例。

(7) 设置系统，通过假脱机操作将脚本结果输出到日志文件，以备日后进行分析。

```
SQL> SPOOL upgrade_info.log
```

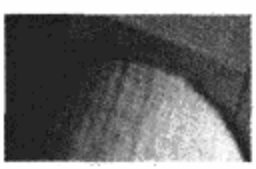
(8) 运行升级前信息工具：

```
SQL> @utlu111i.sql
```

(9) 关闭通过假脱机输出到日志文件的脚本结果：

```
SQL> SPOOL OFF
```

在 upgrade_info.log 中检查升级前信息工具的输出结果，以便在升级前发现问题。



注意：
当升级到 Oracle Database 11g 时，会为缺少统计信息的字典表收集优化器统计信息。对于具有大量字典表的数据库，收集统计信息会花费一些时间，但只有对那些缺少统计信息或升级期间发生很大变化的表才收集统计信息。要减少收集统计信息所导致的停机时间，可以在真正进行数据库升级之前收集统计信息。与 Oracle Database 10g 一样，Oracle 建议使用 DBMS_STATS.GATHER_DICTIONARY_STATS 过程收集这些统计信息。

3.4　使用数据库升级助手(DBUA)

可以通过

```
dbua
```

命令(在 UNIX 环境下)或者选择 Oracle Configuration and Migration Tools 菜单中的 Database Upgrade Assistant 项(在 Windows 环境下)来启动数据库升级助手(Database Upgrade Assistant，DBUA)。

启动时，DBUA 将显示一个欢迎界面，接着显示升级选项的一个列表(以便升级数据库或 ASM 实例)。在下一个界面上，从可用数据库列表中选择需要升级的数据库。一次只能升级一个数据库。然后您可以选择在升级过程中移动数据库文件。

然后 DBUA 提示您闪回还原区域的目标位置，以便存储与备份和还原相关的文件。选择相关文件之后，将会显示数据库管理选项屏幕，提示基本的配置信息，如数据库管理员的电子邮件地址。输入管理配置信息之后，会提示您输入 Oracle 数据库提供的账户的密码。

如果 Oracle 检测到服务器上有多个 Oracle Net 监听器，则会提示您为数据库选择一个监听器，并显示网络配置细节，以便于查看和编辑。

在选择监听器之后，升级过程就开始了。DBUA 将执行升级前的检查(如检查已废弃的初始化参数或者太小的文件)。然后 DBUA 将提示您在升级之后重新编译无效的 PL/SQL 对象(参考本书的第Ⅳ部分)。如果在升级之后没有重新编译这些对象，那么第一次使用到这些对象的用户将被迫等待 Oracle 执行运行时重编译。

接下来 DBUA 将提示您备份数据库，这是升级过程的一个组成部分。如果在启动 DBUA 之前已经备份了数据库，则可以跳过这一步。如果选择让 DBUA 备份数据库，它将关闭数据库并执行脱机备份，把数据文件备份到指定的目录。DBUA 另外还要在这个目录下创建一个批处理文件，用于把数据文件自动恢复到原来的位置上。

最后的小结界面将显示您所做出的升级选择，如果接受这些选择，则升级将马上开始。升级结束后，DBUA 将显示升级结果(Upgrade Results)界面，显示执行了的步骤、相关的日志文件和状态。这个界面上标题为 Password Management 的区域允许您管理密码和加锁/解锁升级过的数据库中账号的状态。

如果对升级的结果不满意，可以选择 Restore 选项。如果在备份时选择的是 DBUA，则恢复将自动完成；否则，您需要手动完成恢复。

在成功升级数据库后退出 DBUA 时，DBUA 将删除网络监听器配置文件中旧数据库的条目，插入升级后数据库的条目，并重新加载这个文件。

3.5　执行手动直接升级

在手动升级中，必须执行 DBUA 所执行的各个步骤。这样一来，在手动升级数据库时您必须负责(同时也控制着)直接升级过程中的每一步。

应该在升级之前使用升级前信息工具分析数据库。这个工具包含在一段 SQL 脚本中，此脚本是和 Oracle Database 11g 软件一起安装的；您需要对待升级的数据库运行该脚本。这个脚本文件的名称是 utlu111i.sql，位于 Oracle Database 11g 主目录下的/rdbms/admin 子目录中。您应该以 SYSDBA 用户的身份在需要升级的数据库中运行这个文件，通过假脱机操作将结果输出到一个日志文件中。这些结果将显示出在升级之前应该解决的潜在问题。

如果在升级之前没有需要解决的问题，那么应该关闭数据库，并执行脱机备份，然后继续进行升级。

备份完成之后，需要的时候就可以还原了，下面就可以开始升级了。由于升级过程有详细描述且是基于脚本的，因此应该阅读针对您的环境和版本的 Oracle 安装和升级文档。升级步骤与操作系统有关，而且取决于现有的配置。

Oracle Database Upgrade Guide 文档中详细描述了手动升级的各个步骤。您应该阅读此文档，以便了解升级时需要遵循哪些步骤。大体包括以下这些步骤：

(1) 关闭 Oracle 数据库和访问它的所有服务。

(2) 备份数据库。

(3) 准备新的 Oracle 软件主目录并编辑所有相关的配置文件。

(4) 使用 startup upgrade 命令启动数据库。

(5) 如果是从 Oracle 9.2 升级，则创建一个 SYSAUX 表空间。

(6) 通过 catupgrd.sql 脚本升级数据字典表。

(7) 启动实例。

(8) 运行升级后状态工具(Post-Upgrade Status Tool) utlu111s.sql 脚本，以便验证所有数据库组件都已正确升级。

(9) 运行 catuppst.sql 脚本，执行额外的升级步骤。

(10) 运行 utlrp.sql 脚本，重新编译程序包和过程。

参阅 Oracle Database Upgrade Guide 文档，以便了解针对您的环境的详细升级步骤，以及在发生错误的情况下如何解决。

3.6 使用 Export 与 Import

Data Pump Export 和 Data Pump Import 以及原来的 Export 和 Import 实用程序是一种间接的升级方法。可以创建一个和现有的数据库并存的 Oracle 11g 数据库，并使用 Data Pump Export 和 Data Pump Import 把数据从旧数据库迁移到新数据库中。当数据迁移完成后，需要让自己的应用程序指向新的数据库而不是原来的那个数据库。您还需要更新所有配置文件、版本特有的脚本和网络连接配置文件(tnsnames.ora 和 listener.ora)，使这些文件指向新数据库。

根据您是从哪个版本进行升级，有可能需要使用原来的 Export 和 Import 实用程序，具体参见下面的描述。

3.6.1 使用哪个 Export 和 Import 版本

在使用 Export 实用程序创建 Export 转储文件时，该文件可以导入到所有新版本的 Oracle 中。因为 Export 转储文件并不是向后兼容的，所以如果需要转换到旧版的 Oracle 中，就需要仔细选择 Export 和 Import 实用程序的版本。表 3-1 列出了在不同的版本之间转换时可以执行

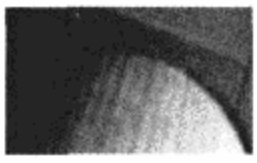
的 Export 和 Import 实用程序的版本。

表 3-1　不同版本之间转换时使用的 Export 和 Import 实用程序的版本

从哪个版本导出	导入到哪个版本	使用的 Export 实用程序版本	使用的 Import 实用程序版本
版本 10.2	版本 11.1	Data Pump Export 版本 10.2	Data Pump Import 版本 11.1
版本 10.1	版本 11.1	Data Pump Export 版本 10.1	Data Pump Import 版本 11.1
版本 9.2	版本 11.1	原来的 Export 版本 9.2	原来的 Import 版本 11.1
版本 8.1.7	版本 11.1	原来的 Export 版本 8.1.7	原来的 Import 版本 11.1
版本 8.0.6	版本 11.1	原来的 Export 版本 8.0.6	原来的 Import 版本 11.1
版本 7.3.4	版本 11.1	原来的 Export 版本 7.3.4	原来的 Import 版本 11.1

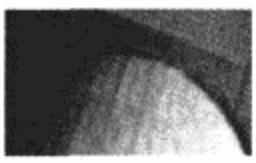
3.6.2　进行升级

使用 3.6.1 节指定的 Export 实用程序的版本从源数据库中导出数据。可以执行一致性的导出，也可以在升级期间或升级后数据库不用于更新时执行导出。



注意：
如果可用的磁盘空间很少，现在您就可以备份并删除已经存在的数据库，然后安装 Oracle Database 11g 软件并创建一个供导入的目标数据库。只要有可能，就应该在升级期间同时维护源数据库和目标数据库。在服务器上同一时间只保存一个数据库的唯一好处是：它们可以共享相同的数据库名称。

安装 Oracle Database 11g 软件并创建目标数据库。在目标数据库中，预先创建存储源数据所需要的用户和表空间。如果源数据库和目标数据库共存于服务器上，则必须注意不能用一个数据库中的数据文件把另一个数据库中的数据文件覆盖掉。Import 实用程序将尝试着执行在 Export 转储文件中发现的 create tablespace 命令，这些命令将包含源数据库中数据文件的名称。默认情况下，如果文件已经存在(虽然这可以通过 Import 实用程序的 DESTROY 参数覆盖掉)，则这些命令的执行将失败。使用正确的数据文件名预先创建表空间就可以避免这个问题。



注意：
可以导出特定的表空间、用户、表和数据行。

一旦准备好数据库，就可以使用 Import 实用程序或者 Data Pump Import 实用程序(参见第 24 章)把数据从 Export 转储文件加载到目标数据库。检查日志文件，确认没有对象导入失败的信息。

3.7　使用数据复制法

数据复制法需要让源数据库和目标数据库共存。这种方法最适用于需要迁移的表比较小

而且数量也不多的情况。您必须预防在数据提取期间和提取之后发生在源数据库中的事务。在这种方法中，使用数据库链接通过查询提取数据。

使用 Oracle Database 11g 软件创建目标数据库，然后预先创建将由源数据库中的数据进行填充的表空间、用户和表。在目标数据库中创建访问源数据库中的账号的数据库链接(参考第 25 章)。使用 insert as select 命令把数据从源数据库迁移到目标数据库。

数据复制方法允许您只复制需要的数据行和列；可通过查询限制待迁移的数据。必须认真处理源数据库中表之间的关系，以便可以在目标数据库中正确地重新创建这些关系。如果可以把应用程序停用很长的时间以便升级，或者您想在迁移的过程中修改数据结构，那么数据复制方法能够很好地满足您的需要。注意这种方法需要一次将数据同时存储在多个地方，这对存储空间有一定的要求。

为了改进这种方法的性能，可以考虑下面的几个选项：

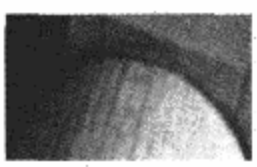
- 在加载所有的数据之前禁用所有的索引和约束条件
- 并行运行多个数据复制的作业
- 使用并行查询选项提高单个查询和插入的性能
- 使用 APPEND 提示提高插入的性能

关于性能调整的其他建议请参考第 46 章。

3.8 升级完成之后的工作

升级完成之后，应该仔细地对照检查与数据库有关的配置和参数文件，特别是在迁移的过程中实例的名称发生了改变的时候。这些文件包括：

- tnsnames.ora 文件
- listener.ora 文件
- 包含有硬编码实例名称的程序文件

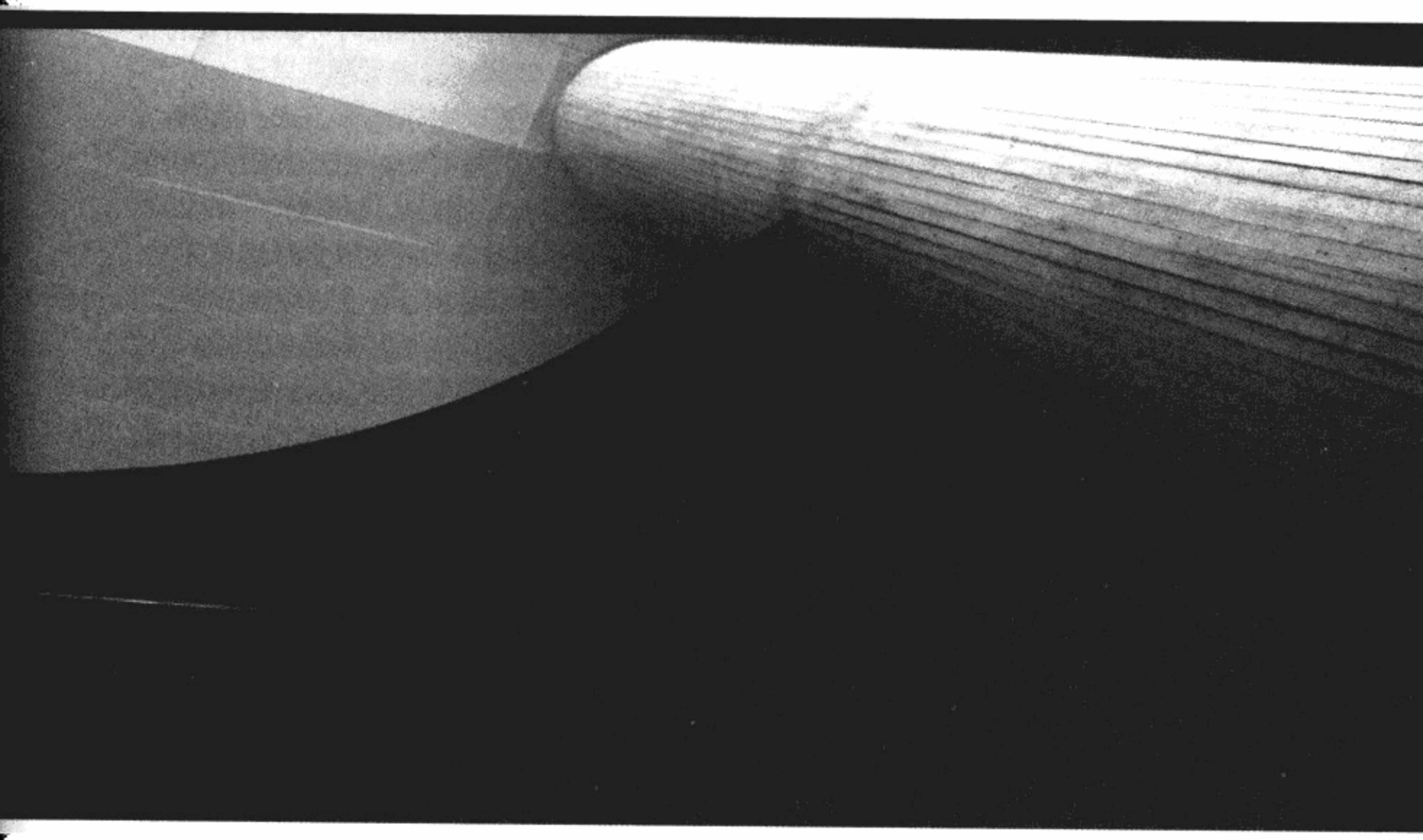


注意：
如果没有使用 DBUA 进行升级，那么需要手动地重新加载修改过的 listener.ora 文件。

升级完成之后还应该验证所有环境变量(如 ORACLE_HOME 和 PATH)的值并对恢复目录进行升级。

应该仔细检查数据库初始化参数，以确保删除所有不主张使用的和过时的参数；在迁移的过程中本应该把这些参数识别出来。一定要重新编译您编写的所有与数据库软件库有关的程序。

一旦升级完成，就应该执行在升级之前就确定下来的功能和性能测试。如果数据库的功能有问题，就要尝试识别可能影响测试结果的所有参数设置或遗漏的对象。如果无法解决问题，则也许需要返回到以前的版本。



第4章

规划 Oracle 应用程序——方法、风险和标准

为了快速并且高效地开发一个Oracle应用程序，用户和开发人员必须用共同的语言交流，并且对商业应用程序和 Oracle 工具有深入的共识。在前面的几章中，本书介绍了 Oracle 产品的总体描述以及安装和升级的步骤。在安装了软件后，现在就可以根据技术人员和业务领域专家对业务和数据达成的共识来开发应用程序了。

在过去，系统分析员首先分析业务需求并构建满足这些需求的应用程序。用户只有在描述业务时，或者在应用程序完成之后检查其功能时才会参与到项目中来。

随着可用的新工具和新方法越来越多(特别是 Oracle)，在创建应用程序时可以更好地符合用户的需求和业务的处理习惯——但是只有大家达成共识时才行得通。

本书明确的目标就是培养这种共识，并为用户和开发人员提供充分利用 Oracle 的各种方法。最终用户知道业务的细节，而开发人员对此却不甚了解。开发人员能理解 Oracle 的内部功能和特性以及计算机的环境，而这对最终用户而言却在技术上太复杂。但是对于 Oracle 最终用户和开发人员可共享的知识，专业性知识相对很少。这里有很大的发挥空间。

“业务”人员和“系统”人员之间存在着冲突早已不是什么秘密。造成这种局面的原因包括知识、文化、专业兴趣和目标之间的差异，而且两个群体之间简单的距离分隔也会造成彼此疏远。公平地说，这种表现并不是数据处理过程所特有的。当财务部门的人员、人力资源部门的人员或高级管理人员分别位于不同的楼层、不同的大楼或者不同的城市时，他们之间同样会疏远。不同部门的人员之间的关系会变得礼仪化、不自然而且不正常。从这种孤立主义中滋生出来的人为阻碍和手续变得越来越牢不可破，这也是导致这种现象的原因之一。

您也许会认为这没有什么不对，而且社会学家对这种现象也许会很感兴趣，但是这和 Oracle 有什么关系呢？

因为 Oracle 不是用只有系统专业人员才能理解的神秘语言来操作的，所以它从根本上改变了业务人员和系统人员之间的关系的本质。任何人都可以理解它。任何人都可以使用它。对于业务人员，那些以前必须由系统内部的人员创建并发布了报表之后才可以直接获得的系统内部信息，现在只要输入英文查询命令就可以了。这改变了游戏规则。

在使用 Oracle 的领域，它将迅速增进两个阵营之间的理解，增加彼此的知识，而且使两个阵营之间的关系正常化。另外这还有利于创建优秀的应用程序，获得一个圆满的结局。

因为从第一个版本开始，Oracle 一直是基于易于理解的关系模型(稍后将详细解释)，所以非程序开发人员也已经理解了 Oracle 的作用以及实现方式。这使得它易于学习和使用。

有些人到现在为止既不接受也不理解这种观点，而且他们也没有意识到“用户”和“系统”之间由来已久的人为障碍不断出现是非常致命的。但是协作开发的出现将极大地改变应用程序和它们的作用。

然而，很多应用程序开发人员使用 Oracle 时却陷入了困境：继续使用以前的系统设计中积累下来的方法没有什么益处。有很多东西需要忘却。以前设计系统时很多必不可少的技术(和限制)在使用 Oracle 进行设计时都不是必需的，它们很可能会产生相反的效果。在解释 Oracle 的过程中，必须消除这些旧的习惯和方法的负面影响。有很多新的可能性。

贯穿本书的目的是通过一种明确而又简单的方法，使用用户和开发人员都理解的术语解释 Oracle。本书将不再提及那些过时的和不合理的设计和管理方法。

4.1 协作方法

Oracle 数据库是对象关系数据库管理系统。对于理解和管理业务中使用的数据，关系数据库是一种非常简单的方法。关系数据库只不过是一组数据表的集合而已。我们每天都会看到很多表——天气预报表、股票图例、体育运动的积分等。这些都是表，它们简单地显示列标题和信息行。即便如此，关系方法对最复杂的业务也是完善而且足够强大的。对象关系数

据库不但支持关系数据库的所有功能而且还支持面向对象的概念和功能。您可以把 Oracle 当作一个关系数据库管理系统(RDBMS)，也可以使用其面向对象的功能。

但是，从关系数据库中受益最多的人——业务用户——通常对它的理解却最少。应用程序开发人员通常会发现难以用简洁的术语向用户解释关系的概念，而他们又必须为用户开发系统。所以要使两类人员协作必须有一种通用的语言。

本书的前两部分将使用已经广为接受的术语解释关系数据库究竟是什么以及如何在业务中高效地使用它。这个讨论看起来好像只对“用户”有利。经验非常丰富的关系应用程序设计人员可能会跳过这几章，只是把这本书当作 Oracle 资料的主要参考书。虽然这些内容看起来好像只是在回顾基础知识，但对应用程序设计人员来说，阅读这些内容可以获得清楚的、一致的、有效的术语，然后可以使用这些术语和用户沟通，了解他们的需求以及怎样迅速地满足这些需求。如果您是应用程序设计人员，这些内容也许还可以帮助您忘却一些没有必要而且可能是无意识的设计习惯。很多这些习惯在介绍关系方法的过程中并不会讨论到。重要的是要意识到即使是 Oracle 的强大功能也可能会由于只适用于非关系型开发的设计方法而大幅削弱。

如果您是最终用户，理解对象关系数据库隐藏的基本思想将有助于您用准确的语言向开发人员描述您的需求，同时有助于您理解怎样才能满足这些需求。从事业务处理的普通人员可以在很短的时间内从初学者变成专业人员。使用 Oracle，您将可以获取并使用信息，轻易地控制报表和数据，对应用程序的功能以及它怎样实现这些功能有清晰的理解。Oracle 能够使用户非常专业地控制应用程序或查询实用工具，而且还能知道自己是否充分利用了 Oracle 的各种灵活性和强大的功能。

另外您还可以让程序员从他们最没有兴趣的工作(编写新的报表)中解脱出来。在大型的组织中，有多达 95%的编程工作都是编写新的报表。由于用几分钟而不是几个月的时间就可以编写出自己的报表，因此您也乐于承担这个责任。

4.2　每个人都有“数据”

图书馆保存着会员、图书和罚款的清单。棒球卡收集者会记录球员的名字、日期、平均分和卡面值。在所有的商业活动中，都会保存客户、产品、价格、财务状况等各方面的信息。这些信息称作数据。

信息学家也许会说数据只有以一种有意义的方式组织起来之后才能成为“信息”。如果真是这样，那么使用 Oracle 还是一种把数据转变成信息的简单途径。Oracle 将对数据进行排序和处理，以便揭示隐藏在数据中的信息——如总计、购买倾向以及一些尚未被发现的关系。您将学习如何去发现这些关系。这里的重点是您拥有数据，并对其进行 3 个基本的操作：获取、存储和检索。

一旦完成了基本的操作，就可以对数据进行计算，把它从一个地方移动到另一个地方，或者对其进行修改。这称作数据处理，而且，它从根本上包含了同样的 3 个步骤，这些步骤影响着数据的组织方式。

可以使用一个雪茄烟盒、一支铅笔和一张白纸来处理数据，但是当数据量增加时，您应

该换一种工具。您也许会使用一个文件柜、几个计算器、几支铅笔和几张白纸。虽然这从某种意义上来说可以避免使用计算机，但是您的工作任务并没有改变。

关系数据库管理系统(relational database management system，RDBMS)——如Oracle——使得您能以一种可以理解的而且相对简单的方式来完成这些任务。Oracle 基本上完成 3 件事情，如图 4-1 所示。

- 让您输入数据
- 保存数据
- 让您获取数据并对其进行处理

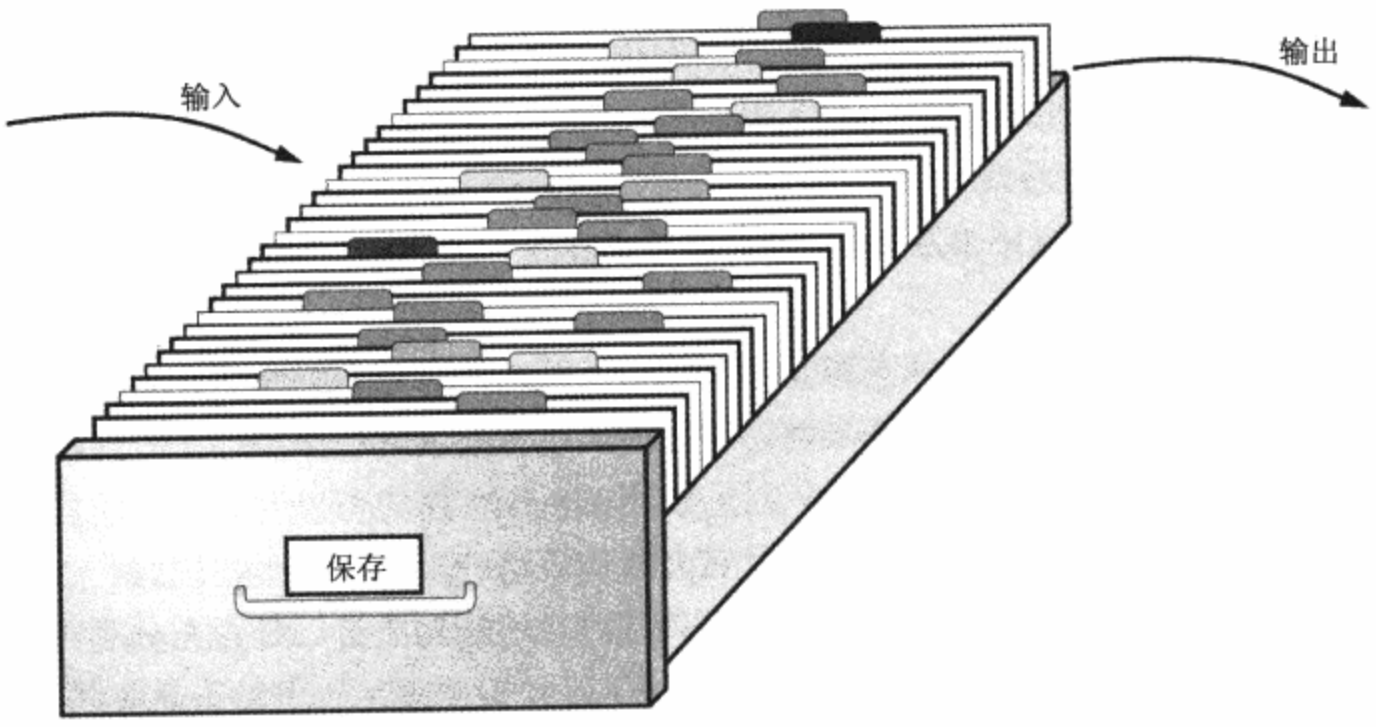


图 4-1 简单的处理过程

Oracle 支持这种输入/保存/输出的方法并提供更智能的工具，这些工具可以让您自由地获取、编辑、修改和输入数据，安全地保存数据，以及取出数据进行操作并制作报表。

4.3 熟悉的 Oracle 语言

存储在 Oracle 中的信息保存在表中——和日报上的天气预报表差不多，如图 4-2 所示。

WEATHER

City	Temperature	Humidity	Condition
Athens.....	97	89	Sunny
Chicago.....	66	88	Rain
Lima.....	45	79	Rain
Manchester...	66	98	Fog
Paris.....	81	62	Cloudy
Sparta.....	74	63	Cloudy
Sydney.....	69	99	Sunny

图 4-2 报纸上的天气预报表

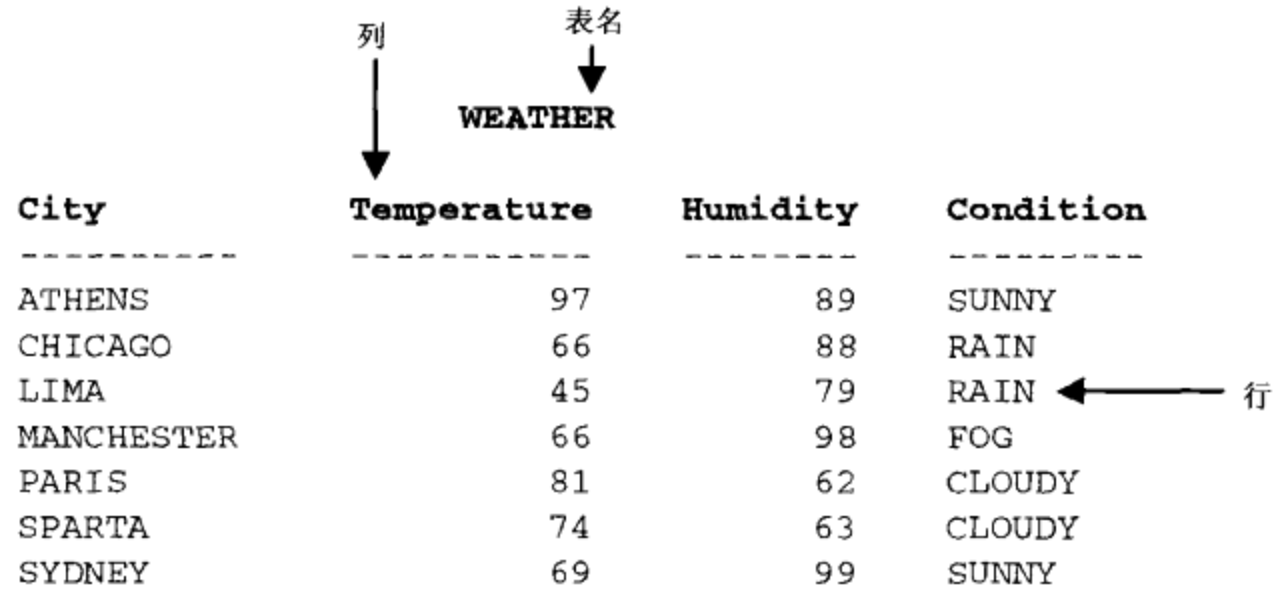
这个表有 4 列：City、Temperature、Humidity 和 Condition。从 Athens 到 Sydney 的每一个城市都对应一行数据。最后，这个表有一个名称：WEATHER。

大多数表具有如下 3 个主要特征：列、行和表名。关系数据库也是这样。任何人都能够理解这些词汇和它们所表示的意思，因为用于描述 Oracle 数据库中表的各个部分所使用的词汇和我们日常对话中所使用的词汇是一样的。这些词汇没有特殊的、不同寻常的或者怪异的含义。

4.3.1　存储信息的表

Oracle 将信息存储在表中，图 4-3 给出了一个示例。每个表有一列或多列。列的标题(如图 4-3 中的 City、Temperature、Humidity 和 Condition)描述了保存在列中的信息的类型。信息按行(每行代表着一个城市)存储。每一组唯一的数据，如城市 Manchester 的温度、湿度和天气情况，都被保存在一个单独的行中。

Oracle 避开了那些专业的、学术性很强的术语，以便让它自己变得更容易学习。在关系理论的研究论文中，列可能称作“属性(attribute)”，行可能称作“元组(tuple)”，而表则可能称作“实体(entity)”。然而，这些术语可能会让最终用户感到迷惑。最糟糕的是，这些术语对我们在日常用语中双方都理解的名字进行了一次毫无必要的重命名。Oracle 使用这些通用的语言，开发人员也可以。必须意识到由于使用一些没有必要的技术行话而造成的不信任和误解。和 Oracle 一样，本书坚持使用“表”、“列”和“行”这些术语。



City	Temperature	Humidity	Condition
ATHENS	97	89	SUNNY
CHICAGO	66	88	RAIN
LIMA	45	79	RAIN
MANCHESTER	66	98	FOG
PARIS	81	62	CLOUDY
SPARTA	74	63	CLOUDY
SYDNEY	69	99	SUNNY

图 4-3　Oracle 中的 WEATHER 表

4.3.2　结构化查询语言

Oracle 是第一家发布使用基于英文的结构化查询语言(或者称为 SQL)的产品的公司。这种语言使得最终用户可以自己提取信息，而不用为每个非常小的报表使用系统组。

Oracle 的查询语言具有结构，就像英语和其他语言也有自己的结构一样。它有语法规则和句法规则，不过这些规则基本上就是普通的英语语法规则，人们很容易理解。

SQL 读作“sequel”或“S－Q－L”，您将会看到它是一个功能非常强大的工具。使用它不需要任何编程经验。

下面给出一个使用 SQL 的示例。如果要找出前面的 WEATHER 表中湿度(humidity)为 89%

的城市，您可以很快就回答是 Athens。如果要找出温度(temperature)是 60° C 的城市，您会回答是“Chicago 和 Manchester”。

Oracle 可以像您一样轻松地回答同样的问题，而且所要回答的查询语句与您提出的问题非常相似。在 Oracle 中使用的查询关键词有 select、from、where 和 order by。它们是 Oracle 理解您的请求并给出正确回答的根据。

4.3.3 简单的 Oracle 查询

如果在 Oracle 数据库中有 WEATHER 这张表，则您的第一个查询语句(用分号告诉 Oracle 执行这个命令)很可能如下所示：

```
select City from WEATHER where Humidity = 89 ;
```

Oracle 将给出如下的回答：

```
City
----------
ATHENS
```

第二个查询应该是这样：

```
select City from WEATHER where Temperature = 66 ;
```

对这个查询，Oracle 给出的回答是：

```
City
-----------
MANCHESTER
CHICAGO
```

正如您所见，每一个查询中都使用了关键词 select、from 和 where。那么 order by 的作用是什么呢？假设您希望让所有的城市按照温度进行排列。只需要输入：

```
select City, Temperature from WEATHER
order by Temperature ;
```

Oracle 将立刻给出回答：

```
City          Temperature
-----------   -----------
LIMA                   45
MANCHESTER             66
CHICAGO                66
SYDNEY                 69
SPARTA                 74
PARIS                  81
ATHENS                 97
```

Oracle 将根据温度迅速地重新排列表(表中首先列出温度最低的城市。第 5 章将学习如何指定按升序还是降序排列)。

虽然关于 Oracle 的查询工具您可能还有很多问题，但是这些简单的示例已经足以向您表明，以对自己最有用的形式从 Oracle 数据库中获得所需要的信息是多么简单的事情。虽然您可以根据多个简短的信息构造复杂的查询，但是无论如何，所使用的方法都是可以理解的。例如，可以同时使用 where 和 order by 两个简单的关键字，前一个关键字告诉 Oracle 选择温度高于 80°C 的城市，后一个关键字告诉 Oracle 把这些城市按温度的升序排列。可以输入如下的命令：

```
select City, Temperature from WEATHER
 where Temperature > 80
 order by Temperature ;
```

Oracle 将立即给出答复：

```
City          Temperature
-----------   -----------
PARIS                  81
ATHENS                 97
```

或者可以指定得更具体一点，选择温度高于 80°C 并且湿度低于 70%的城市：

```
select City, Temperature, Humidity from WEATHER
where Temperature > 80
 and Humidity < 70
order by Temperature ;
```

Oracle 将给出回答：

```
City        Temperature  Humidity
----------- -----------  --------
PARIS                81        62
```

4.3.4　为什么称作“关系”

注意在 WEATHER 表中列举出了不同国家的城市，有些国家列出了不止一个城市。假设您需要知道某个城市位于哪个国家，那么可以创建一个单独的 LOCATION 表，其中包含城市和它们所在的国家，如图 4-4 所示。

对于 WEATHER 表中的任何城市，可以直接查看 LOCATION 表，在 City 列中找到城市名，然后查看同一行的 Country 列，就可以找到国家的名字。

在图 4-4 中有两个完全分开和独立的表。每个表都把它的信息保存在自己的行和列中。这两个表有一个共同点：都包含 City 列。对于 WEATHER 表中的每一个城市名，在 LOCATION 表中都有一个同样的城市名称。

例如，属于 Australian 的城市的当前气温、湿度和气候是什么样的？请看图 4-4 中的两个表，找到答案，然后继续阅读下面的内容。

WEATHER

City	Temperature	Humidity	Condition
ATHENS	97	89	SUNNY
CHICAGO	66	88	RAIN
LIMA	45	79	RAIN
MANCHESTER	66	98	FOG
PARIS	81	62	CLOUDY
SPARTA	74	63	CLOUDY
SYDNEY	69	99	SUNNY

LOCATION

City	Country
ATHENS	GREECE
CHICAGO	UNITED STATES
CONAKRY	GUINEA
LIMA	PERU
MADRAS	INDIA
MADRID	SPAIN
MANCHESTER	ENGLAND
MOSCOW	RUSSIA
PARIS	FRANCE
ROME	ITALY
SHENYANG	CHINA
SPARTA	GREECE
SYDNEY	AUSTRALIA
TOKYO	JAPAN

图 4-4 WEATHER 表和 LOCATION 表

您是如何解决这个问题的？在 LOCATION 表的 Country 列中您只能找到 AUSTRALIA 记录。紧接着，在同一行的 City 列中，找到城市名称：SYDENY。根据 SYDENY 这个名称查找 WEATHER 表中的 City 列。找到这个城市名称之后，您就可以在同一行中找到 Temperature、Humidity 和 Condition 列的值：69、99 和 SUNNY。

虽然这两个表是独立的，但是您很容易看出它们之间是有关系的。一个表中的城市名称和另一个表中的城市名称是关联在一起的(参见图 4-5)。关系数据库正因为这种关系而得名。

WEATHER

City	Temperature	Humidity	Condition
ATHENS	97	89	SUNNY
CHICAGO	66	88	RAIN
LIMA	45	79	RAIN
MANCHESTER	66	98	FOG
PARIS	81	62	CLOUDY
SPARTA	74	63	CLOUDY
SYDNEY	69	99	SUNNY

LOCATION

City	Country
ATHENS	GREECE
CHICAGO	UNITED STATES
CONAKRY	GUINEA
LIMA	PERU
MADRAS	INDIA
MADRID	SPAIN
MANCHESTER	ENGLAND
MOSCOW	RUSSIA
PARIS	FRANCE
ROME	ITALY
SHENYANG	CHINA
SPARTA	GREECE
SYDNEY	AUSTRALIA
TOKYO	JAPAN

关系

图 4-5 WEATHER 表和 LOCATION 表之间的关系

这就是关系数据库的基本思想(有时候也称为关系模型)。数据存储在表中。表有列、行和名称。如果表之间有一列信息是相同的，表之间就可以相互关联起来。

关系的概念就是这样的。它和您想象的一样简单。

4.4　一些通用的、常见的示例

在理解了关系数据库的基本思想之后，您就会发现表、行和列实际上出现在很多地方。并不是以前没有看到过它们，而是以前可能没有用这样的方式思考它们。很多您习以为常的表都可以存储到 Oracle 中。它们可以用来快速地回答您费尽周折来回答的问题。

图 4-6 是一个典型的股市行情报表。该图显示的只是报纸上给出的报表的一小部分，这些报表一般按照字母顺序排列得非常紧密。哪一支股票的交易额最多？哪一支股票的差价最大，无论是涨还是落？在 Oracle 中回答这些问题只需要执行简单的英文查询语句就可以了，远比您在报纸上一列一列地查找要快得多。

Company	Close Yesterday	Close Today	Shares Traded
Ad Specialty	31.75	31.75	18,333,876
Apple Cannery	33.75	36.50	25,787,229
AT Space	46.75	48.00	11,398,323
August Enterprises	15.00	15.00	12,221,711
Brandon Ellipsis	32.75	33.50	25,789,769
General Entropy	64.25	66.00	7,598,562
Geneva Rocketry	22.75	27.25	22,533,944
Hayward Antiseptic	104.25	106.00	3,358,561
IDK	95.00	95.25	9,443,523
India Cosmetics	30.75	30.75	8,134,878
Isaiah James Storage	13.25	13.75	22,112,171
KDK Airlines	80.00	85.25	7,481,566
Kentgen Biophysics	18.25	19.50	6,636,863
LaVay Cosmetics	21.50	22.00	3,341,542
Local Development	26.75	27.25	2,596,934
Maxtide	8.25	8.00	2,836,893
MBK Communications	43.25	41.00	10,022,980
Memory Graphics	15.50	14.25	4,557,992
Micro Token	77.00	76.50	25,205,667
Nancy Lee Features	13.50	14.25	14,222,692
Northern Boreal	26.75	28.00	1,348,323
Ockham Systems	21.50	22.00	7,052,990
Oscar Coal Drayage	87.00	88.50	25,798,992
Robert James Apparel	23.25	24.00	19,032,481
Soup Sensations	16.25	16.75	22,574,879
Wonder Labs	5.00	5.00	2,553,712

图 4-6　股市行情表

图 4-7 是报纸的索引。F 板块有什么？如果从前向后阅读这份报纸，您将以什么样的顺序阅读这篇文章？在 Oracle 中通过一条简单的查询语句就可以获得这些问题的答案。在使用本书的过程中，您将学会怎样编写这些查询，以及创建用于存储信息的表。

Feature	Section	Page
Births	F	7
Bridge	B	2
Business	E	1
Classified	F	8
Comics	C	4
Doctor's In	F	6
Editorials	A	12
Modern Life	B	1
Movies	B	4
National News	A	1
Obituaries	F	6
Sports	D	1
Television	B	7
Weather	C	2

图 4-7 显示报纸板块的表

本书的示例中使用的数据和对象都是在业务和日常生活中经常遇到的。用于练习的数据也应该尽量是常见的东西。在后面的章节中您将使用与日常生活有关的数据源学习如何输入和检索数据。

与所有的新技术和风险投资一样，不仅需要考虑它们带来的收益和机会，还要考虑到成本和风险。如果把关系数据库和一系列功能强大、简单易用的工具结合起来，就像 Oracle 一样，那么可能因为其简单而诱发灾难。添加面向对象的功能和 Web 功能之后，就更增添了危险。下面的章节将讨论开发人员和最终用户都需要考虑的一些危险的情况。

4.5 风险所在

开发关系数据库应用程序的主要风险是：它们确实非常简单。理解表、列和行并不困难。两个表之间的关系在概念上也很简单。即使是规范化(normalization)(也就是分析一个公司的数据的不同元素之间的内在关系或“规范”关系的过程)，学起来也是相当容易的。

遗憾的是，这通常会产生一些速成的“专家”，虽然他们充满自信但是对于如何构建真正的、可作为商品的应用程序毫无经验。对于非常小的市场数据库，或者家庭理财应用程序，这没有太大的关系。程序中包含的错误会被及时发现，开发人员会吸取教训，在下一次开发程序时避免出现同样的错误。然而，在一个非常重要的应用程序中，这肯定会导致灾难。缺乏经验往往会导致一个重要的项目失败。

老的开发方法通常都比较慢，主要是因为老的方法中包含的各种任务(编写代码、提交编

译作业、链接和测试)导致了开发的缓慢。工作周期，特别是牵涉到大型主机的工作周期，通常都非常枯燥，以至于程序员要花费大量的时间在本机上进行检查，以避免由于代码中的错误而延缓另一个周期的工作。

第 4 代开发工具使得很多开发人员仓促地开发产品。代码的修改和实现是如此迅速，以至于测试的周期非常短。本机检查(desk-checking)实质上已经不存在，这就造成了很多问题。当导致本机检查的负面因素(很长的工作周期)消失之后，本机检查也随之消失。很多人的态度好像是："如果应用程序存在缺陷，则我们可以很快修改它。如果数据遭到破坏，则我们可以快速进行更新。如果速度不够快，则我们随即可以调整。让我们提前得到它，先睹为快。"

重要的 Oracle 项目的测试周期应该比传统项目的周期更长、更详细。即使对项目进行了合理的控制，即使有经验丰富的项目经理在指导项目，也必须如此，因为开发人员在本机上的检查比原来少，而且他们都过于自信。测试必须检查数据输入界面和报表的正确性、数据加载和更新的正确性、数据完整性和并发性的正确性、特别是在负载最高峰时事务和存储量的正确性。

因为 Oracle 应用程序确实如所表现的那样简单，所以使用 Oracle 的工具进行应用程序开发的速度快得惊人。但这自动就会减少所进行的测试数量，而测试是开发的一个有机组成部分，为了弥补应用程序的不足，计划好的测试和品质保证工作必须有意识地延长。虽然 Oracle 新手或刚开始使用第 4 代开发工具的人员一般都不会预见到这一点，但是您必须在项目计划中留出相应的预算。

4.6　新视角的重要性

很多人都期待着有一天能够在屏幕上用英语输入"自然的"语言查询，然后在很短的时间内在屏幕上得到答案。

很多人都没有意识到我们距离这个目标已经非常接近。限制因素不再是技术，而是在应用程序的设计中出现的思想僵化。Oracle 能够直接地构建基于英语的系统，这些系统即使对于没有经验的用户也是很容易理解和使用的。潜在的问题是，很多功能在 Oracle 的数据库和工具中都已经可以使用了，但是只有很少的人理解并使用这些功能。

任何 Oracle 应用程序都应该具有清楚易懂的特点。应用程序应该可以通过人类语言来操作，让所有没有编程背景的最终用户能够轻松地理解，并能通过简单的英语查询提供信息。

该怎么做呢？首先，设计工作的一个主要目标是必须使应用程序易于理解和使用。如果您误入歧途，则必须回到这个方向上来，即使这样做意味着占用更多的 CPU 资源和磁盘空间。这种方法的局限性是您可能创建了一个非常易于使用的应用程序，但是这个程序过于复杂以至于无法维护或改进。这同样是一个非常糟糕的错误。但是，反过来也一样，不能由于编写代码的需要而使得最终用户难以适应。

4.6.1 变化的环境

考虑一下以每秒钟运行一百万条指令(MIPS)的方式表示的计算机运行成本，这个成本每年都以 20%的速度下降。而另一方面，劳动力的成本每年都在稳步增长。这就意味着如果可以把某些由人力完成的工作转移到计算机上，就能够节约成本。

我们是否已经把这个令人难以置信的转移过程考虑到应用程序的设计中了？答案可能是“一点点”，但是实际上根本就没有。真正的进展是在操作环境上，如首先在帕洛阿尔托研究中心(Xerox Palo Alto Research Center，PARC)、然后在 Macintosh 上、现在又在基于 Web 的浏览器和其他基于图标的图形化系统上完成的视觉效果方面的工作。这些环境与旧的、基于字符的环境相比，更易于学习和理解。人们在旧的环境下可能要花费数天时间才能做完的工作，而现在只需要几分钟就可以了。某些方面的提高幅度非常大，以至于我们都忘了以前有些任务是如何费尽周折的。

遗憾的是，很多应用程序开发人员都没有真正理解这个容易适应的而且友好的环境的概念。他们虽然在这样的环境中工作，却仍沿袭着陈旧的编程习惯。

4.6.2 代码、缩写和命名标准

旧的编程习惯的问题主要集中在代码、缩写和命名标准上，在考虑最终用户的需求时这些问题几乎不会引起人们的注意。当最终考虑到这 3 个问题的时候，通常只关心系统部门的需要和习惯。这看起来好像是一个必须考虑的枯燥而又没有意思的问题，但是如果做得好就能够获得巨大的成功，使生产效率直线上升，并培养出兴趣高涨的、有效率的用户；如果做得不好，就只能让用户勉强接受，同时收益少得可怜，而且失去耐心、饱受折磨的用户会不断地向开发人员提出要求。

业务记录通常保存在各种分类账册中。每个事件和事务都用自然语言一行一行地记录下来。在我们开发应用程序的时候，会用代码来表示数据的值(比如用“01”表示“Accounts Receivable”，“02”表示“Accounts Payable”等)。键盘输入人员实际上必须知道或者查看大多数这样的代码并在屏幕上相应的字段中输入它们。这是一个极端的示例，但实际上很多应用程序都采用这种方法，每个细节都很难学习或理解。

这个问题在大型的、传统的主机系统开发中尤为突出。在把关系数据库介绍给这些开发人员之后，它们只是被简单地用来替换原来的输入/输出方法，如虚拟存储器存取方法(Virtual Storage Access Method，VSAM)和信息管理系统(Information Management System，IMS)。在这样一种方式下，关系数据库的强大功能实际上全被掩盖了。

1. 为什么使用代码而不是自然语言

为什么要使用代码？通常有两个主要理由：

- 每种类别的数据项都非常多以至于不能用自然语言正确地表示和记录下来
- 节约计算机的磁盘空间

第二个原因现在已经过时了。内存和永久存储器曾经非常昂贵，而且 CPU 的速度也非常慢(还没有现在的手持计算器的速度快)，以至于程序员不得不想方设法把信息以最小的磁盘空间保存下来。数字(用于表示字符的符号)所占的计算机存储空间只有字母的一半，而代码进一步降低了对计算机的要求。

由于计算机比较昂贵，因此开发人员不得不对每件事都使用代码，从而使一切正常运转。这是通过技术来解决经济问题。对于那些必须学习所有这些无意义的代码的用户来说，这个要求显然是很可怕的。因为计算机又慢又贵，无法适应人类，所以要训练人类来适应计算机。这是无法避免的苦差事。

因为经济原因而使用代码的理由已经消失好多年了。计算机现在足够快，也足够便宜，完全能适应人类的工作方式，而且使用的词汇人们也都理解。这个发展来得非常及时。然而，开发人员和设计人员都继续使用代码，而没有认真地考虑一下这个原因的由来。

第一点是更切实的问题，虽然每个类别中的数据项太多，但是并不像它一开始看起来的那么多。有一种观点认为输入由数字表示的代码比输入实际的字符串，如书的标题，要轻松一些(所以成本也更低一些)。这种解释在 Oracle 中是不成立的。不仅因为培训人员了解客户、产品、事务和其他代码的成本更高，纠正错误的成本(尤其是基于代码的系统)更昂贵，而且因为使用代码还意味着没有完全发挥 Oracle 的作用；Oracle 能够根据标题最前面的几个字符自动连带出整个标题。在应用程序的所有地方，对于产品的名称、事务等也是一样的(例如，输入 b 将会自动连带输出 buy，输入 s 将会自动连带输出 sell)。Oracle 通过非常强大的模式匹配功能实现这种功能。

2. 用户反馈的好处

有一个非常直接的好处：由于用户可以立即得到用自然语言返回的他们输入的业务信息的错误，键盘输入造成的错误被降到最低。数字的顺序不会被颠倒；代码也不会被记错；而且，在财务应用程序中，金额几乎不会因为输入错误而减少，节约了大量的成本。

另外，应用程序变得更容易理解了。界面和报表从神秘的包含代码和数字的数组转变成容易阅读和理解的格式。设计方法由面向代码转变到面向自然语言对公司及其员工都有非常深远而且积极的影响。对于曾经饱受代码之苦的用户而言，使用基于自然语言的应用程序是一种巨大的精神解脱。

4.7　如何减少混淆

对“每个类别的数据项都非常多”的另一种解释是产品、客户或者事务类型的数量是如此之多，以至于难以通过名字来区别它们，或者是在一个类别的数据中有很多相同的或是非常相似的数据项(例如，有多个名为 John Smith 的客户)。一个类别中可以有太多的数据项，以至无法记住或者区别它们，但更常见的情况是信息的分类不彻底：很多并不相似的数据都被硬塞到一个范围很大的类型中。与基于代码相反，开发一个强大的基于自然语言方向的应

用程序需要花费大量的时间与用户和开发人员在一起——剖析业务的信息、理解信息的本质关系和类型，然后仔细地创建一个数据库并为方案定义一个简单却可以准确反映这些新分类的名字。

完成这个工作需要 3 个基本的步骤：

(1) 规范化数据

(2) 为表和列选择自然语言的名称

(3) 为数据选择自然语言的单词

本书将按顺序解释上面的每一个步骤。这样做的目的是设计一个应用程序，使这个应用程序中的数据按照有意义的方式组织起来，存储数据的表和列的名字都是用户熟悉的，而且用用户熟悉的术语描述，而不是使用代码。

4.7.1 规范化

不同国家之间、同一个公司的不同部门之间、或者用户和开发人员之间的关系，通常都是特定历史环境的产物，这些环境虽然已经消失很久了，但仍然可能是形成目前种种关系的主要因素。但随着时间的变化，其结果可能是不正常的关系，或者，用现在的说法，是失效的关系。历史和环境通常会对数据产生相同的影响——包括数据的收集、组织和报告方式。同样，数据也会变得不规范而且失效。

规范化(normalization)是正确存放事物，使它们变得规范的一个过程。这个词来源于拉丁词 norma，表示木匠的直角尺，用于测量一个角是不是直角。在几何学中，当一条直线和另一条直线的夹角为直角时，我们说这条直线和另一条直线垂直。在关系数据库中，这个术语有一个特定的数学意义，与怎样把数据的各个元素(如名字、地址或者技能)分割成几个关系紧密的组(affinity groups)，并定义它们之间规范的、或者说“正确的”关系有关。

在这里介绍规范化的基本概念，是为了让用户对他们将要使用的应用程序的设计提出意见，或者让他们能够更好地理解已构建的系统。然而，如果认为规范化过程只能用于设计数据库或者计算机应用程序那就错了。规范化能够让您专注于业务中使用的各种信息以及这些信息中的各种元素是怎样相互关联的。这种方法在数据库和计算机以外的领域同样具有学习的价值。

1. 逻辑模型

分析过程的前期步骤之一是建立逻辑模型(logical model)，该模型只是业务中使用到的数据的一个规范化的图表。了解为什么分割数据以及如何分割数据，对于理解该模型非常关键，而模型对于构建一个可以长期支持业务、不需要其他额外支持的应用程序是非常关键的。

规范化通常用范式表示：第一范式(First Normal Form)、第二范式(Second Normal Form)和第三范式(Third Normal Form)最常用，其中第三范式表示规范化的程度已经非常高。另外也定义了第四范式(Fourth Normal Form)和第五范式(Fifth Normal Form)，但它们已经超出了本书的讨论范围。

考虑一个书架：对每本书，您可以存储与它有关的信息——标题、出版商、作者和该书所从属的多个类别或描述性用语。假设将这种书籍级别的数据转变成 Oracle 中的表设计。表的名称也许是 BOOKSHELF，列的名称可能是 Title、Publisher、Author1、Author2、Author3 和 Caterory1、Caterory2、Caterory3。如果是这样的话，使用这个表的用户已经受到一个限制了：在 BOOKSHELF 表中，用户最多只能列出一本书的 3 个作者和 3 个所属类别。

如果书籍可接受的类别发生了变化该怎么办？必须有人检查 BOOKSHELF 表中的每一行数据并修改所有旧的值。另外如果其中某个作者的名字改变了又该怎么办？同样，所有相关的数据都得修改。如果一本书还有第 4 个作者您将如何处理呢？

这些其实并不是计算机和技术的问题，虽然它们是由于您在设计数据库时才暴露出来的。它们其实是关于怎样合理而且合乎逻辑地组织业务信息的基本问题。规范化所要解决的正是这些问题。通过去年无效的关系并确保规范的关系，逐步把数据元素重新组织到关系紧密的组中来解决这些问题。

2. 规范化数据

重新组织数据的第一步就是把它们放入第一范式中。在第一步中，要把数据放入多个独立的表中，每一个表的数据类型相似，并为每个表指定一个主键(primary key)——唯一的标号或标识符。这样将消除重复的数据行，如书架上的作者名。

每个作者的数据被放到一个单独的表中，每一行存放一个名字和相应的描述，而不是只允许每本书对应 3 个作者。这样 BOOKSHELF 表中的作者就不必是一个变数，比限制在 BOOKSHELF 表中最多只有 3 个作者的设计方法要好。

接下来，定义每个表的主键：唯一标识数据行并允许您按行提取出数据的字段。为简单起见，假设标题和作者的名称是唯一的，所以 AuthorName 就是 AUTHOR 表的主键。

现在必须把 BOOKSHELF 表分割成两个表：Author 表和 BOOKSHELF 表。Author 表包含 AuthorName(主键)和 Comments 两列；BOOKSHELF 表的主键是 Title，另外还包含 Publisher、Category1、Category2、Category3、Rating 和 RatingDescription 列。创建第 3 个表，BOORSHELF_AUTHOR，提供前两个表之间的关系：一本书可以有多个作者，同时一个作者也可以编写多本书——我们把这种关系称作多对多关系。图 4-8 显示了 3 个表的主键和这些关系。

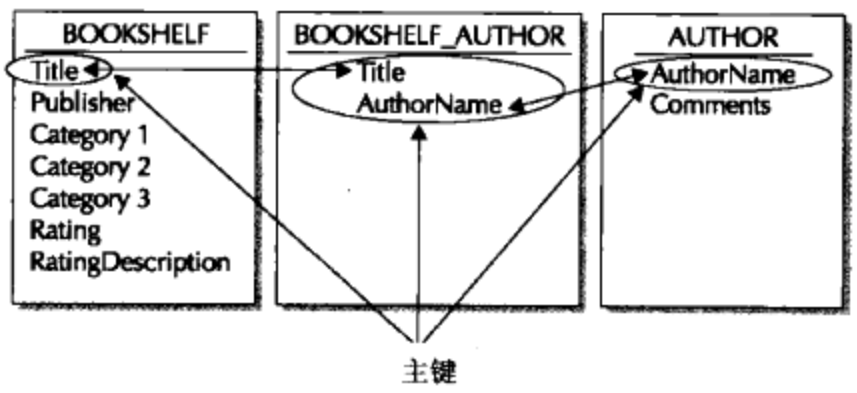


图 4-8　BOOKSHELF 表、AUTHOR 表和 BOOKSHELF_AUTHOR 表

规范化过程的下一步为第二范式，目的是找出只依赖于部分键值的数据。如果有不依赖于整个键的属性，那么应该把这些属性移到另一个表中。在这个示例中，因为 RatingDescription 属性并没有真正依赖于主键 Title，而是依赖于 Rating 列的值，所以应该把它移到另一个单独的表中。

最后一步是第三范式，这意味着剔除表中所有不唯一依赖于主键的数据。在这个示例中，类别之间有相互关联的部分：一个标题不会被同时指定为 Fiction 和 Nonfiction 两种类型，而且在 Adult 类型下的子类别和 Children 类别下的子类别也不会相同。因此把 Category 信息移到另一个单独的表中。图 4-9 显示了满足第三范式的表。

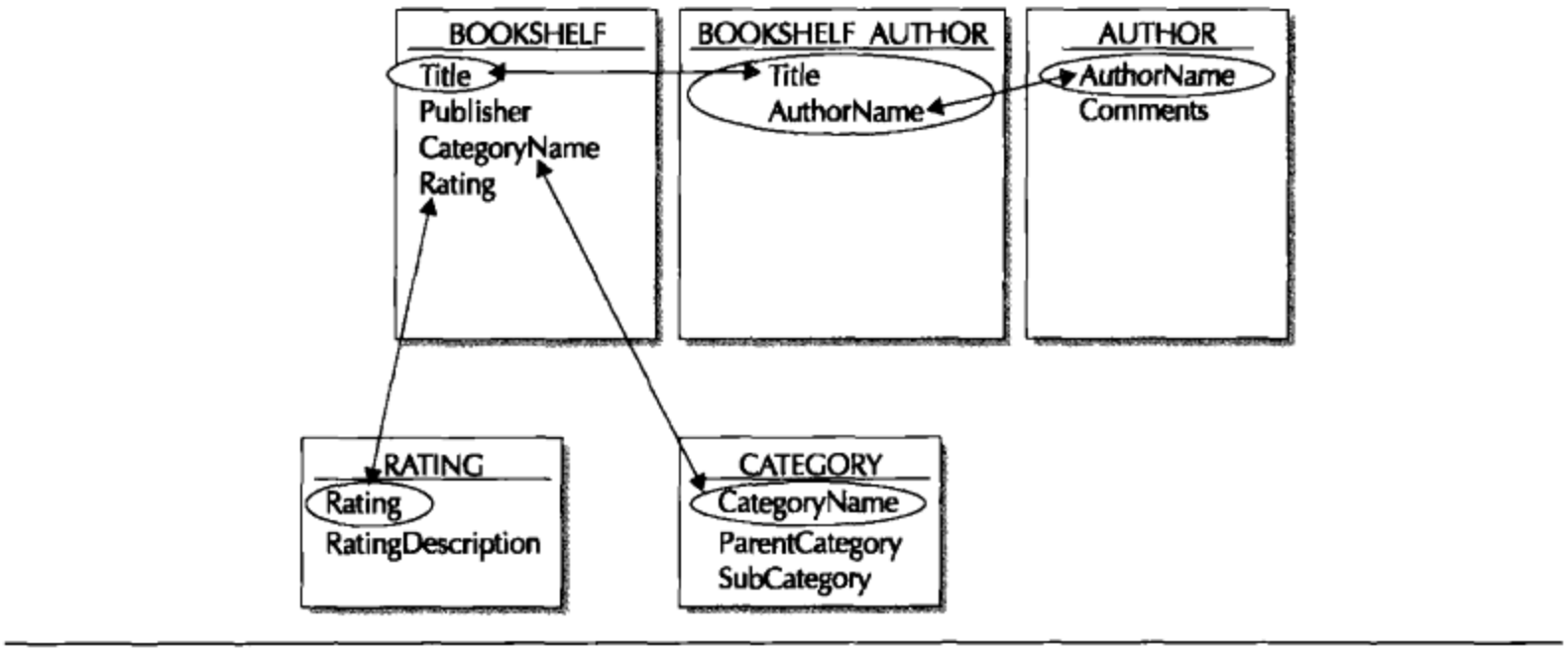


图 4-9 BOOKSHELF 表和相关的表

无论何时，满足第三范式的数据将自动满足第二范式和第一范式。因此，整个规范化的过程并不用在范式之间反复进行。只需要安排好数据，使得表中的所有列，除了主键以外，都仅仅依赖于整个主键。第三范式有时候被描述为“键、整个键和只依赖键”。

3. 在数据之间导航

bookshelf 数据库现在满足第三范式。图 4-10 显示了这些表中可能包含的数据。很容易就可以看出这些表是怎样关联到一起的。根据每个表的键值在表之间导航就可以找出与某个特定的作者有关的信息。每个表中的主键都可以唯一地标识每一行数据。例如，在 AUTHOR 表中可以很容易就找到作者 Stephen Jay Gould 的数据，因为 AuthorName 是这个表的主键。

AUTHOR

AuthorName	Comments
DIETRICH BONHOEFFER	GERMAN THEOLOGIAN, KILLED IN A WAR CAMP
ROBERT BRETALL	KIERKEGAARD ANTHOLOGIST
ALEXANDRA DAY	AUTHOR OF PICTURE BOOKS FOR CHILDREN
STEPHEN JAY GOULD	SCIENCE COLUMNIST, HARVARD PROFESSOR
SOREN KIERKEGAARD	DANISH PHILOSOPHER AND THEOLOGIAN
HARPER LEE	AMERICAN NOVELIST, PUBLISHED ONLY ONE NOVEL
LUCY MAUD MONTGOMERY	CANADIAN NOVELIST
JOHN ALLEN PAULOS	MATHEMATICS PROFESSOR
J. RODALE	ORGANIC GARDENING EXPERT

RATING

Rating	RatingDescription
1	ENTERTAINMENT
2	BACKGROUND INFORMATION
3	RECOMMENDED
4	STRONGLY RECOMMENDED
5	REQUIRED READING

CATEGORY

CategoryName	ParentCategory	SubCategory
ADULTREF	ADULT	REFERENCE
ADULTFIC	ADULT	FICTION
ADULTNF	ADULT	NONFICTION
CHILDRENPIC	CHILDREN	PICTURE BOOK
CHILDRENFIC	CHILDREN	FICTION
CHILDRENNF	CHILDREN	NONFICTION

BOOKSHELF_AUTHOR

Title	AuthorName
TO KILL A MOCKINGBIRD	HARPER LEE
WONDERFUL LIFE	STEPHEN JAY GOULD
INNUMERACY	JOHN ALLEN PAULOS
KIERKEGAARD ANTHOLOGY	ROBERT BRETALL
KIERKEGAARD ANTHOLOGY	SOREN KIERKEGAARD
ANNE OF GREEN GABLES	LUCY MAUD MONTGOMERY
GOOD DOG, CARL	ALEXANDRA DAY
LETTERS AND PAPERS FROM PRISON	DIETRICH BONHOEFFER

图 4-10　BOOKSHELF 表中的样本数据

BOOKSHELF

Title	Publisher	CategoryName	Rating
TO KILL A MOCKINGBIRD	HARPERCOLLINS	ADULTFIC	5
WONDERFUL LIFE	W.W.NORTON & CO.	ADULTNF	5
INNUMERACY	VINTAGE BOOKS	ADULTNF	4
KIERKEGAARD ANTHOLOGY	PRINCETON UNIV PR	ADULTREF	3
ANNE OF GREEN GABLES	GRAMMERCY	CHILDRENFIC	3
GOOD DOG, CARL	LITTLE SIMON	CHILDRENPIC	1
LETTERS AND PAPERS FROM PRISON	SCRIBNER	ADULTNF	4

图 4-10 (续)

查看 BOOKSHELF_AUTHOR 表的 AuthorName 列中的 Harper Lee，您会发现她已经出版了一本小说，小说的名字是 *To Kill A Mockingbird*。然后可以在 BOOKSHELF 表中查看这本书的出版商、类别和等级(rating)。如果想获得等级的描述信息，可以查看 RATING 表。

在 BOOKSHELF 表中查找 *To Kill A Mockingbird* 小说时，是通过主键搜索的。为了找到这本书的作者，可以反转前面的搜索路径，查找 BOOKSHELF_AUTHOR 表中 Title 列的值为 To Kill A Mockingbird 的记录——在 BOOKSHELF_AUTHOR 表中 Title 列是一个外键。如果 BOOKSHELF 表的主键出现在另一个表中，就像它出现在 BOOKSHELF_AUTHOR 表中那样，则此时把它称作外键(foreign key)。

这些表同时还显示了一些现实生活中的特点：有些等级和类别还没有被书架上的书使用到。由于数据是按逻辑组织起来的，所以可以保存一些潜在的类别、等级和作者，尽管目前还没有书籍会使用到这些数据。

无论这种表用在什么地方，这都是一种有意义的、合乎逻辑的数据组织方式。当然，为了把这些表转换成一个真正的数据库还有一些工作需要处理。例如，应该把 AuthorName 分割成 FirstName 和 LastName，而且您可能还希望用某种方式表示哪个作者是主要作者，或者某个人只是编辑而不是作者。

整个这个过程称作规范化。这并没有太多的技巧。虽然一个优秀的设计还包括其他方面的内容，但是正如刚才解释的那样，分析数据中不同元素之间的“规范”关系是非常简单和直接的。就算不涉及关系数据库和计算机，这种组织方式也是很有意义的。

然而，有一点必须注意。规范化是分析过程的一个组成部分。它不是设计。数据库应用程序的设计还要考虑其他很多方面的内容，而且，认为逻辑模型的规范化表就是实际的数据库的“设计”是完全错误的。混淆分析和设计往往是导致某些主要的关系应用程序失败的原因之一。本章后面将向开发人员更详细地解释这些问题。

4.7.2 表和列的英文名称

在理解了应用程序的数据中各个元素之间的关系并把数据元素恰当地分割开之后，就应该集中精力为存放这些数据的表和列选取名称。人们对这个方面的关注非常少，即使是那些

更了解情况的人也是这样。通常在没有征求最终用户的意见而且没有进行严格复查的情况下就确定了表和列的名称。在实际使用应用程序时，这两个缺点都将产生严重的后果。

例如，考虑图 4-10 中的表。这些表和列的名称实际上都很明显。最终用户，即使是对关系的概念和 SQL 都不太熟悉的用户，在理解甚至是复制这样一个查询时也不会有太多的困难：

```
select Title, Publisher
  from BOOKSHELF
order by Publisher;
```

用户理解这个查询语句是因为对这些词语都很熟悉。这里没有任何晦涩的或者定义有误的术语。在定义必须包含很多列的表时，为列选择名称可能会更加困难，但有一些惯用的强制规则会起到很大的帮助作用。设想一下由于没有命名约定而造成的一些常见的困难。假如选择的是这些名称，那会怎样呢？

```
BOOKSHELF     B_A       AUTHS          CATEGORIES
------------- --------  -------------  --------------
title         title     anam           cat
pub           anam      comms          p_cat
cat                                    s_cat
rat
```

遗憾的是，这张表中稀奇古怪的命名方法非常常见。这些表名和列名是根据几个有名的供应商和开发人员所使用的约定来表示的。

下面是理解这些名字时遇到的一些较明显的困难：

- 没有很好的理由使用缩写。这使得记住表名或列名的“拼写”几乎是不可能的事情。这些名称可能也和代码差不多，因为用户将不得不查阅它们代表的是什么。
- 缩写不一致。
- 表或列的作用或含义不明确。缩写除了使得名称的拼写难以记住之外，它们还使得包含在列或者表中的数据的性质变得模糊。P_cat 是什么呢？Comms 又是什么呢？
- 下划线的用法不一致。有时候它被用来分隔一个名称中的不同单词，有时候又不是。人们要怎样才能记住哪个名字里面是否有下划线呢？
- 名词复数的使用不一致。是 CATEGORY 还是 CATEGORIES？是 Comm 还是 Comms？
- 有些规则存在限制。如果用表名的第一个字母命名列，如 Anam 列用于以 A 开头的表中，那么如果必须有另一个同样以 A 开头的表该怎么办？表中的列名也称为 Anam 吗？如果是这样，那么两个表中的列名为什么不直接称为 Name？

以上只是一些最明显的困难。如果表和列的命名都非常糟糕，则用户将无法简单地输入英文查询。这些查询语句不会像 BOOKSHELF 表的查询语句那样给人直观的熟悉的“感觉”，而且这将在很大程度上降低应用程序的接受程度和作用。

以前常常要求程序员创建长度最多为 6~8 个字符的名称。结果，名称中就不可避免地混杂着字符、数字和密码一样的缩写，混乱不堪。和原来的技术造成的其他很多的用户限制一

样，这个限制也解除了。Oracle 允许表和列的名称最长可达 30 个字符。这使得设计人员有足够的空间来创建完整的、含义明确的描述性名称。

这里列出来的困难已经暗含着解决方法，如避免缩写和复数，以及要么不使用下划线，要么一致地使用下划线。这些经验在解决现在非常流行的命名混淆问题的道路上很多帮助。与此同时，命名约定必须简单，易于理解和记忆。在某种意义上，我们需要的是一种规范化的命名。与对数据进行逻辑分析、按目的进行分割从而进行规范化一样，对命名标准也需要在逻辑上给予足够的注意。没有这个标准，应用程序的构建工作将是不正确的。

4.7.3 数据中的英文单词

在讨论了表和列的命名约定这个重要的问题之后，接下来介绍数据本身。毕竟，当表中的数据打印在报表上时，数据的明显性将决定理解报表的难易程度。在 BOOKSHELF 表的示例中，Rating 是一个代码值，而 Category 是多个值的串连。是否可以改进一下呢？如果询问某人一本书的情况，那么您是否想听到它的等级是 AdultNF 中的第 4 等级呢？为什么计算机给出的答案就允许模糊一些呢？

另外，用英语保存数据可以使得编写和理解查询更容易。查询应尽可能类似于自然语言：

```
select Title, AuthorName
  from BOOKSHELF_AUTHOR;
```

4.8 名称和数据中的大写

Oracle 忽略字母的大小写(无论是输入大写字母、小写字母，还是大小写混合)，从而使记住表和列的名称比较容易。它在内部的数据字典中以大写形式存储表和列的名称。在输入查询的时候，它立即把表和列的名称转换成大写形式，然后在字典中查找它们是否存在。其他一些关系系统是区分大小写的。如果用户输入一个列的名称为“Ability”，但若数据库认为它是“ability”或者“ABILITY”(由创建表时选取的名称决定)，那么系统将不能理解这条查询语句。

注意：

虽然可以在 Oracle 中强制创建大小写混合的表和列的名称，但是这样做会使得查询和处理数据变得很麻烦。应使用默认的大写形式。

人们把创建区分大小写表名的功能当作一种优点，这是因为，它允许程序员创建名称相似的表。他们可以创建 worker 表、Worker 表、wORker 表等。这些都是独立的表。一个人(包括程序员)怎样才能记住这些表之间的差别呢？所以这其实是一个缺点，不是优点，Oracle 非常明智，没有掉进这个陷阱。

对于存储在数据库中的数据也是类似的情况。虽然有很多方法可以从数据库中查找信息，无论数据是大写形式的还是小写形式的，但是这些方法都施加了一个不必要的负担。除了极少数内容以外，如法定文本或者格式信件段落，以大写形式在数据库中存储数据会容易得多。这样可以使查询变得更容易，并在报表上提供一个更一致的外观。如果某些数据需要

以小写形式、或者是大小写混合的形式(如信上的称呼和地址)保存，那么将自动调用 Oracle 中执行转换功能的函数。总而言之，以大写形式存储和报告数据省事，而且不容易产生混淆。可以用区分大小写的查询与 Oracle 进行交互，但是通常情况下数据来用一致的大小写会简化应用程序的开发。

回顾本章的内容，您会看到我们并没有遵循这一惯例，直到介绍了这一主题并能在适当的上下文环境中应用时再加以采用。从现在开始，除了一两个表和一些孤立的实例外，数据库中的数据都将以大写形式保存。

4.9　规范化名称

市场上有很多查询工具，这些工具的目的是让您能使用普通的英语单词而不是奇怪的字符组合来编写查询。这些产品的工作方式是在普通的英语单词和那些难以记住、不是英语单词的列名、表名和代码之间建立一个逻辑映射关系。虽然映射需要仔细地思考，但是一旦映射完成之后，它将使得用户和应用程序的交互变得很容易。但是为什么不在一开始时就注意这个问题呢？如果只要在一开始的时候为表和列选择更好的名字就能够避免大多数的混淆，那么为什么会需要另外一个层、另外的产品和多余的工作呢？

出于性能原因，也许应用程序的有些数据必须以代码的形式存储在计算机的数据库中。不论是在输入还是检索数据的时候，这些代码都不应该被暴露给用户，Oracle 能够轻易就把它们隐藏起来。

如果在数据输入的时候用到代码，那么键盘输入发生错误的可能性将增大。如果报表中包含了代码而不是英语，那么在解释代码时将可能产生错误。当用户需要创建新报表或临时的报表时，他们完成任务的速度和准确性将受到代码和那些奇怪的表名称和列名称的重要影响。

Oracle 能够让用户在整个应用程序的工作中都使用英语。若忽略这个机会就等于浪费 Oracle 的功能，而且毫无疑问这会导致开发出来的应用程序的可理解性和效率都比较差。开发人员应该抓住这个机会。用户应该要求实现这样的功能。他们都将从中受益无穷。

4.10　人性化和优秀的设计

到目前为止，如果您是 Oracle 的初学者，您也许马上就想开始使用 Oracle 和 SQL 语言。这将在第 5 章中讨论；本章的剩余部分将把重点放在性能、命名和设计的考虑上。您可以在准备好设计和实现一个应用程序的时候再回顾这一节的内容。

本节将讨论开发项目的方法，该方法要考虑最终用户必须完成的实际业务工作。这使得这种方法区别于很多开发人员和开发方法中更常使用的面向数据的方法。数据规范化和 CASE(Computer Aided Software Engineering，计算机辅助软件工程)技术在关系应用程序开发中占据的位置如此重要，以至于对数据和参照完整性、键、规范化以及表的图表的关注都难以与其相提并论。这些问题和设计经常被混淆——甚至把它们视为是设计——以至于当有人提

醒它们只是分析时，常常让人们感到吃惊。

规范化是分析，而不是设计。而且它只是分析的一部分，对于理解某一业务并创建一个成功的应用程序是必不可少的。毕竟，开发应用程序的目的，就是提高处理业务工作的的速度和效率，以及尽量使人们工作的环境变得有意义和受支持，从而帮助业务更成功地运行。让用户控制他们的信息，并让他们直观地、便捷地访问信息，用户会很欢迎，并可以提高工作效率。如果把数据的控制权交给一个远程用户组，造成代码中信息的模糊和用户界面的不友好，则他们肯定会不高兴，而且工作效率也不会高。

本节描述的方法并不是设计过程的详细说明，而且您处理数据结构或数据流所使用的熟悉的工具也许完全能够很好地完成任务。这里的目的只是揭示一种高效地创建反应灵敏的、正确的而且易于使用的应用程序的方法。

4.10.1 理解应用程序的任务

在构造软件的时候一个很容易忽视的步骤是理解最终用户的任务——这是计算机自动化致力于支持的目标。有时候，这是因为应用程序本身非常特殊；但大多数时候，这是因为设计的方法倾向于面向数据。通常，在分析的过程中需要回答以下几个主要问题：

- 应该捕获什么数据？
- 应该怎样处理数据？
- 应该怎样报告数据？

这几个问题可以扩展成一系列子问题，它们包括诸如输入形式、代码、屏幕布局、计算、记入、错误订正、审核、保留项、存储量、处理周期、报告格式、分配和维护等问题。这些都是至关重要的领域。然而，一个难点是它们都只关注数据。

虽然人们是在使用数据，但是他们是在“完成”任务。有人也许会说虽然这对专业人员可能是必要的，但是键盘输入人员的工作只是简单地把输入表单上的数据输入到计算机中；他们的工作就是面向数据的。现在这样描述这些工作是正确的。但是这是真正需要完成的工作的结果呢？还是计算机应用程序设计的一个特征？把人当作输入设备，特别是输入按一定格式排列的大量的、连续的数据(如表单上的数据)，而且数据的变化非常有限，这是一种既昂贵又过时的数据捕获方法，且很不人性化。就像为了适应计算机的限制而使用代码一样，这是一个落伍的想法。

虽然这听起来也许过于哲学化，但是它们确实渗入到应用程序的设计中。虽然人们使用数据，但是他们是在完成任务。而且他们并不是都从头到尾一次完成一个任务。他们做多个相互包含或者相互交叉的工作，而且是同时并行地完成这些工作。

当设计人员用这种想法来指导应用程序的设计和创建，而不是专注于过去曾经占主导地位的面向数据的方法时，工作的性质将发生显著的改变。为什么窗口环境会如此成功？因为窗口允许用户从一个小任务迅速地跳转到另一个小任务，使所有的任务都保持激活状态，不必为了启动另一个任务而关闭并退出这一个任务。与旧有的“一次完成一个任务”的方式相比，窗口环境更好地映射了人们思考和工作的方式。我们不应该忘记这个教训。应该把它牢记在心里。

理解应用程序的任务意味着不只是简单地识别数据元素、规范化数据、创建界面、处理

程序和生成报告。它意味着需要真正理解用户所做的事情和他们的任务是什么，并设计出能响应这些任务、而不只是捕获与它们有关的数据的应用程序。实际上，当设计方向面向数据时，最终的设计将不可避免地曲解用户的任务而不是支持它们。

如何设计一个响应任务而不是响应数据的应用程序呢？最大的障碍就是要理解：关注业务是必需的。这使得您能以一种全新的视角去分析业务。

分析过程的第一步是理解任务。完成哪些任务用户群的成员真正需要使用计算机？他们提供的服务或者生产的产品到底是什么？这些问题看起来好像很基础，甚至过于简单，但是您会发现有相当多的业务人员对这个问题的答案都不是很清楚。很多业务人员(从医疗保健业到银行业、从航运业到制造业)都认为他们所从事的是数据处理业务，这实在是令人吃惊。毕竟，他们输入数据，处理数据并报告数据，不是吗？这种错觉是我们的系统设计中包含的另一个面向数据的特征，它错误地导致很多公司试图销售他们想象中的“真正的”产品——数据处理，但其中的大多数都因此承受了灾难性的后果。

由此可见了解一个业务应用程序的重要性：您必须保持开放的头脑，而且为了真正了解业务，可能需要不断地挑战一些流行的观点。这是一种健全的过程，虽然有时候会比较困难。

而且，就像业务人员已经成为 SQL 的直接用户并理解关系模型的基本概念是非常关键的那样，应用程序设计人员必须真正理解提供的服务或者生产的产品，以及相关的必要业务，这也是非常重要的。在一个包含最终用户和设计人员的项目小组中，如果最终用户已经学习了 SQL 和关系模型的精髓(如通过阅读本书)，而设计人员对最终用户的需求很敏感，并理解面向任务的、可读性强的应用程序环境的价值，那么这样的项目小组将开发出非常优秀的系统。这个项目小组中的成员将相互检查、支持和促进彼此的工作。

使用这种开发流程的方法之一是开发两个趋同的文档：一个任务文档和一个数据文档。在准备任务文档的过程中将逐步帮助设计小组加深对应用程序的理解。数据文档有助于实现视觉效果，并能确保所有的细节和规则都是经过说明的，但是任务文档确定业务的目的。

4.10.2　任务概要

任务文档是业务用户和应用程序设计人员共同努力的结果。它采用自顶向下的方式列出了所有与业务有关的任务。文档一开始应是业务的基本描述。这应该是一句 3~10 个单词的简单的描述性语句，句子使用主动语气，没有逗号，同时尽量少用形容词，如下所示：

我们卖保险。

但不应该这样写：

Amalgamated Diversified 是一家领先的国际提供商，业务范围包括资金来源、培训、信息处理、事务受理和发布、通信以及客户支持，并就医疗保健、财产保护和汽车债务等领域的风险投资提供专业指导。

人们很可能想把所有关于公司业务和目标的详细内容都放到第一个句子中。但切勿如此。剔除繁冗的描述后得到的简单句子可以集中人们的注意力。如果不能把业务描述精简到

10 个单词，那么说明您还没有理解它。

但是，作为应用程序设计人员，创建这个句子并不是您一个人的责任；而需要和业务用户共同完成，而且它是编写任务文档的首要工作。它为您提供了一个机会，让您认真考虑业务是什么以及怎样完成。这对业务本身来说是一个非常有价值的过程，和正在创建的应用程序并没有什么关系。您会遇到很多后来被证明毫无意义或者作用非常有限的任务、子任务、过程和规则。通常，这些假象很可能是一个以前的问题，很久以前就已经解决了，或者是一个已经辞职了的管理人员提出来的信息或报告要求。

一些人曾经建议处理创建出来的过多报表的方法，就是直接停止创建这些报表并查看是否有人注意到，无论它们是手动创建的还是计算机创建的。这是一个幽默的想法，但它包含的事实却需要包含到任务文档中。实际上，在修复 Y2K 问题的时候这种想法已经被证明是非常有用的——很多程序和报表都不需要修改，因为已经没有人会使用它们了！

通过开发人员和用户共同努力编写任务文档的方法，您可以询问有疑问的问题并考虑那些可能只是假象的问题(同时重新评估这些问题的意义)。然而，必须承认，作为设计人员，您不可能像用户那样详细地理解业务。把握好这一点，就可能使应用程序开发项目正确地认识需要完成什么任务以及这样做的原因；否则，假装比用户更理解“实际的”业务很可能会冒犯用户。

让用户仔细地描述一个任务并解释每一个步骤的理由。如果这个理由很勉强，如“我们一直都是这样做的”，或者“我想他们用这个只是为了某件事情”，那么应该警惕这个理由了。告诉他们您不理解，请他们再解释一遍。如果得到的回答仍不能令人满意，那就需要把这个任务和您的问题放到一个单独的列表中，以待解决。有些问题只需要对这个主题更熟悉的人就可以解答，但是其他问题可能需要和更高级别的管理人员交流，而且很多任务都是以删除告终，因为没有人会再需要它们。一个优秀的分析过程将能够很好地改善现有的过程，而这与新的计算机应用程序无关，而且改善一般都是在实现新的计算机应用程序之前。

1. 任务文档的常规格式

任务文档的常规格式如下：

- 描述业务的总结性语句(3~10 个词)。
- 描述和统计业务中的主要任务数量的总结性语句(简练的句子、简短的词汇)。
- 在每个主要任务中根据需要按不同的详细程度描述任务。

无论如何，都可以根据需要在各级总结性语句的后面增加一个简短的描述性段落，但是不能因此而放弃总结语句的简洁性。通常把主要任务编号为 1.0、2.0、3.0 等，有时也把它们称作 zero-level task(0 级任务)。每个主要任务下面的级别用附加的点号编号，比如 3.1 和 3.1.14。每个主要任务都被分解成一系列的原子任务(atomic task)级别。原子任务不可再分为其他任何子任务，而且一旦启动，要么全部执行，要么全部不执行。原子任务绝对不允许半途而废。

填写支票就是原子任务，而填写美元账目则不是。作为客户服务代表回复一个电话不是原子任务；回复电话并满足客户请求是原子任务。原子任务必须有意义而且必须完成一个动作。

任务在什么级别上是原子任务和任务有关。由 3.1.14 表示的任务可能是原子任务，然而也可能包含另外的子级别。任务 3.2 可以是原子任务，或者任务 3.1.16.4 也可以是原子任务。

重要的不是给任务编号(这只不过是列出任务级别的一种方法)，而是把任务分解到原子级别。原子任务是业务的基本组成块。两个偶尔相互依赖的任务仍然可能是原子任务，当且仅当它们可以而且确实是独立地完成。如果两个任务总是相互依赖，它们就不是原子性的。真正的原子任务将同时包含这两个任务。

在绝大多数业务中，您很快就会发现很多任务都不能被妥当地划分为一个主要任务(0 级别的任务)，而好像是需要分成两个或者更多的任务，而且处于网络或者“虚线”方式。这几乎总是表明主要任务的定义不正确，或者较低级别的任务的原子化还没有完成。分析的目标是把每个任务都变成一个概念上的“对象”，精确地定义它做的是什么(它的终极目标)以及它使用什么资源(数据、计算、思想、纸张、铅笔等)来完成它的目标。

2. 从任务文档中获得的深刻认识

从任务文档中可以获得很多深刻的认识。首先，由于任务文档是面向任务而不是面向数据的，因此它很可能会从根本上改变用户界面的设计方法。它将影响到捕获哪些数据、如何显示这些数据、怎样实现帮助系统以及用户怎样在不同任务之间进行切换。面向任务有助于确保用户在任务之间切换时不需要进行过多的工作。

其次，在发现冲突时主要任务的类别将发生改变，这将影响到设计人员和业务用户如何理解业务。

第三，即使是总结性语句本身也可能发生改变。把一项业务合理地分解成原子任务“对象”必须清除各种假象、混淆的概念以及不需要的依赖性，这些依赖性长久以来给业务增加了不必要的负担。

这一过程并不是一帆风顺的，但是值得投入时间和精力，因为通常在业务的自我理解、过程的清理以及任务的自动化等方面都会得到巨大的好处。如果遇到不合意的问题就问、不正确的假设就修改以及任务文档在完成之前将逐步地进行修改，这种作风在项目小组中建立起来，那么整个项目将受益无穷。

4.11　理解数据

在把任务的分解和描述结合到一起时，每一步所需要的资源，尤其是需要的数据，都将在任务文档中给出描述。这是按任务逐一完成的，所需要的数据将包含在数据文档中。这在概念上是一种不同于经典的数据视图的方法。您将不再只是简单地接受每个任务当前正在使用的表单和界面并记录它们包含的元素。“雪茄烟盒纸片”方法的缺点在于：我们倾向于(即使我们不愿意承认这一点)认为任何印刷在纸张上的东西都是必须的或者是真实的。

在查看每个任务的时候，您应该决定为了完成任务需要什么数据，而不是为了完成任务需要把什么数据元素放到所使用的表单中。根据任务的需要而不是任何已经存在的表单或界面来询问所需数据的定义，您将对任务的真正目的和真正的数据需求进行一次检查。如果完成这项任务的人不知道怎样使用输入的数据，就需列出元素，以待解决。该过程将删除大量的垃圾数据。

一旦识别出当前的数据元素，就必须对它们进行仔细的检查。所有数字和字母代码都可能存在问题。它们把真正的信息掩盖在不直观的、没有意义的符号后面。有些时候或在某些任务中，代码也可能简单易用、易于记忆的，或者纯粹为了填充数据量。但是，在最终的设计中，这种情况应该非常少而且显而易见。否则，您就是“迷路”了。

在对已经存在的数据元素进行详细审查时，对代码应该给予特别的关注。在任何情况下都要问问自己这个元素是否应该是代码。应该用怀疑的眼光观察是否继续把它作为代码使用。必须有很好的依据和充足的理由才能维持这种“伪装”。把代码转变为英文的过程很简单，但它是一个关联的工作。首先按照数据元素把代码和它们的意思列出来，然后由用户和设计人员进行检查，提出简短的英文意思，对其讨论，并暂时使用这些英文。

在这种讨论中，设计人员和用户将决定数据元素的名称。这将作为数据库中的列名，而且通常会在查询中使用，所以这些名称应该具有描述性(除了在业务中常用的缩写以外，避免使用其他缩写)，并且是唯一的。由于列名和列中所包含的数据之间关系很密切，因此应该同时指定两者。一个精心挑选的列名称将极大地简化判定其新数据的过程。

对不是代码的数据元素也要进行严格的检查。到现在为止，您完全有理由相信，所有已经识别出来的数据元素对业务任务都是必须的，但是它们并不一定都已经组织好了。在已有任务中看起来像是一个数据元素的数据项实际上可能是几个混合在一起的元素，需要进行分解。典型的示例包括姓名、地址和电话号码，但是每个应用程序中都有很多其他这样的数据。

例如，在 AUTHOR 表中姓和名被混合到了一起。AuthorName 列中同时包含了姓和名，虽然该表已经满足第三范式了。事实上这是一种非常繁琐的应用程序实现方法，虽然在技术上它已经满足规范化的要求。为了使应用程序切实可行并为英文查询做准备，AuthorName 列至少需要分解成两个新的列：LastName 和 FirstName。在合理划分其他数据元素的工作中通常需要同样的分类过程，而且这个过程在很大程度上独立于规范化操作。

分解的程度取决于如何使用特定的数据元素。很可能会把类别分解得过细，虽然这些类别包含其他独立的类别，但是这样做并不会使它们在新的状态下增加任何价值。在按逐个元素进行的基础上，分解的程度取决于应用程序。一旦完成分解，就需要慎重地为这些新的元素命名，它们将变成列，而且需要仔细地审查它们所包含的数据。对于数量确定的文本数据需要检查其命名情况。这些列的名称和值都是暂定的，与代码的名称和值一样。

4.11.1 原子数据模型

现在规范化的过程开始了，与此同时将描绘出原子数据模型。因为关于这个主题的优秀文章不胜枚举，而且有非常多的分析和设计工具可以加快这个过程，所以本书并不建议某种特定的方法，因为推荐某一种方法可能会事与愿违。

对每个原子事务都应该建立模型，而且要用此模型适用的任务的编号对其进行标记。模型中包含了表的名称、主键和外键以及主要的列。每个规范化的关系都应该有一个描述性的名称，而且每个表中都应该给出估计的行数和事务等级。伴随每个模型的是另外一个清单，其中包含了所有的列和数据类型、它们的数值范围以及表、列和列中指定数值的暂定名称。

4.11.2　原子业务模型

数据文档现在和任务文档绑定到一起。绑定好的文档就是一个业务模型。它的精确性和完整性将由应用程序设计人员和最终用户共同来评审。

4.11.3　业务模型

到现在为止，应用程序设计人员和最终用户应该对业务及其任务和数据有了清晰的认识。一旦修改并批准了业务模型，将任务和数据模型整合成一个总体的业务模型的过程就开始了。过程的这个环节负责把任务之间的公共数据分类，完成最后的、大规模的规范化进程，并决定各个环节使用一致的、确定的名称。

这对于较大的应用程序来说工作量是相当大的，制作出来的支持文档应包含任务、数据模型(建立在完整模型的基础上，使用正确的元素名称)和一个列表，此列表列出所有的表及其列名称、数据类型和内容描述。通过跟踪完整业务模型中每个事务的数据访问路径，以决定事务需要的所有数据都可以用于选择或者插入，而且任务插入的数据没有遗漏对模型的参照完整性起关键作用的元素，从而进行最后的检查。

除了为不同的表、列和公共值恰当地命名之外，到现在为止所进行的工作实际上都是分析，而不是设计。必须不断强调理解业务和它的组成。

4.11.4　数据项

界面设计并不包含在业务模型中。因为它关注的不是表，而是任务，所以创建的界面应支持面向任务并满足必要时在任务之间切换的需要。从实际的角度看，这将映射到任务所使用的主表，以及其他在访问主表时会被查询或者更新的表。

但有时候也会碰到没有主表，而是有很多相互关联的表的情况，所有这些表都将提供或接收数据，从而支持任务。这些界面的外观和行为与很多应用程序中开发的面向表的的界面的外观和行为迥然不同，但它们将显著提高用户的效率和对业务的贡献。而这也正是这种方法的总体目标。

用户和计算机之间的交互很关键；输入和查询界面始终应该面向任务，并具有英文描述性。使用图标和图形界面也非常重要。界面必须反映实际完成工作的方式，而且要使用管理业务的语言来创建。

4.11.5　查询和报告

如果能有任何方法使得关系方法和 SQL 从较传统的应用程序环境中分离出来，那么最终用户很容易学习并执行即席查询。这些报表和一次性查询没有包含在通常随着应用程序的代码一起开发和发布的基本报表集中。

通过使用 Oracle SQL*Plus 实用程序(用于创建和查询数据库对象的命令行实用程序)，最终用户获得了前所未有的数据控制权。这对用户和开发人员都有利：对用户有利是因为他们可以创建报表、分析信息、修改他们的查询并重新执行，这一切只需要几分钟的时间；对开发人员有利是因为他们可以从令人厌烦的创建新报表的需求中解脱出来。

用户现在可以仔细研究他们的数据，对其进行分析，而且快速的响应和详细的分析在几年前都是难以想象的。如果表、列和数据值都是用自然语言精心制作的，那么以后的生产效率就会大大提高。如果因为糟糕的命名约定和毫无意义的代码和缩写影响到设计，那么以后的生产效率就会大打折扣。用户在设计的过程中为对象设定一个一致的描述性名称所花的时间，很快就会得到回报，并因此使整个业务受益。

有一些人，特别是那些没有创建过大型关系应用程序的人，担心让最终用户使用查询工具会影响使用这些工具的计算机。他们担心用户将编写低效率的查询，会消耗大量的 CPU 时间，使得计算机和其他用户的速度都变慢。经验表明一般不会这样。用户很快就能知道什么样的查询运行快，而什么样的查询运行慢。而且，现在有很多业务智能和报表工具都能够估算出一个查询会耗费多少时间，并根据用户、一天中的时间段或者这两个因素一起来限制消耗资源过多的查询。实际上，用户很少过分地要求计算机的资源，但是用户得到的好处远远超过处理过程的开销。实际上无论何时，只要能把由人完成的工作转移到计算机上去完成，就可节约成本。

设计的真正目的是澄清并满足业务和业务用户的需要。如果有什么偏爱，那么一定是倾向于使应用程序更易于理解和使用，特别是不惜以 CPU 和硬盘为代价，但是如果成本的内在复杂性太高，以至于维护和修改都变得困难而缓慢的话，那么尽量不要这样。

4.12 关于对象名称的规范化

命名的基本方法是选择有意义的、容易记忆的和描述性和可读性强的名称，避免使用缩写和代码，而且要么统一使用下划线，要么全部不用。在大型应用程序中，表名、列名和数据名很可能都是由多个单词组成的，如 ReversedSuspenseAccount 或者 Last_GL_Close_Date。慎重选择命名方法的目标就是易用性：名称必须易于记忆，且必须遵循易于解释和应用的原则。下面将介绍一种从某种意义上说比较严格的命名方法，这种方法的最终目标是开发一个正式的对象名称规范化过程。

4.12.1 级别-名称完整性

在关系数据库系统中，对象的层次大到数据库，小到表所有者(table owner)、表、列和数据值。在非常大的系统中，甚至可能有多个数据库，而且可能分布在不同的地区。为简单起见，暂时忽略较高级别的对象，但是这里讨论的内容同样适用于它们。

层次结构中的每一个级别都定义在它的上一个级别范围之内，而且每个级别都应该拥有适合它自己的级别名称，并且不应该和它外面的级别名称交叉。例如，一个表不能拥有两个 Name 列，命名为 George 的账户不能拥有两个 AUTHOR 表。

但是 Oracle 不要求 George 拥有的表的名称在整个数据库中都是唯一的。其他所有者也可以拥有 AUTHOR 表。即使授予 George 访问这些表的权限，也不会造成混淆，因为他可以通过把所有者的名字作为表的前缀来区分它们，如 Dietrich.AUTHOR。如果把 George 的名称作为他的表的前缀，如 GEOAUTHOR、GEOBOOLSHELF 等，这在逻辑上是不一致的。这是把父级名称的一部分放到本级的前面，使得表的名称变得混乱不堪，从而影响了级别-名称

完整性(level-name integrity)。

简洁绝对不能以牺牲清晰为代价。在列名中包含表名的一部分是一种糟糕的技术，因为它违反了这里需要的级别的逻辑概念和级别-名称完整性。它还会使用户产生混淆，使他们在每次编写查询时都必须查看列名。对象名在父级的范围内必须是唯一的，但是不允许渗入超出对象自身级别的名称。

Oracle 对抽象数据类型的支持使得您能够为属性创建一致的名称。如果创建一个 ADDRESS_TY 数据类型，则它在每一次使用时的属性都是一样的。每个属性都会有一个一致的名称、数据类型和长度，使得它们的实现在整个项目中更加一致。但是，以这种方式使用抽象数据类型需要您完成下面的两件事情：

- 在一开始就正确地定义数据类型，避免以后的修改。
- 支持抽象数据类型的语法要求。

4.12.2　外键

使用简短的列名有时候会遇到一个难题，就是有时候表中外键在另一列中的名称和外键列对应的主表(home table)中的名称一样。从长远来看，一种可行解决方案是：允许使用完整的外键名称(包括主表的名称)作为本地表的列名(如把 BOOKSHELF.Title 作为一个列名)。

在实际中，解决同名列的问题需要采取下面几种办法中的一种：

- 使用包含外键源表的名称，该名称中不使用点号(可使用下划线)。
- 使用包含外键源表的缩写的名称。
- 使用与源表中不一样的名称。
- 更改发生冲突的列名。

虽然这些方法都不是非常好，但是如果遇到同名的棘手问题，就需要采取其中的一种方法。

4.12.3　单数名称

一个极易产生前后矛盾和混淆的问题是对象名应该用单数还是复数。应该是 AUTHOR 表还是 AUTHORS 表？是 Name 列还是 Names 列？

有两种方法有助于思考这个问题。首先，考虑几乎会在每个数据库中都出现的列：Name、Address、City、State 和 Zip。除了第一列以外，是否遇到过有些人对其他列使用复数名称？很明显，这些名称描述的是一行的内容，即一个记录。虽然关系数据库是“面向集合的”，但是很明显集合的基本单元是行，而且就是单数的列名所描述的那一行的内容。在设计一个人的名称和地址输入界面时，是不是应该设计成如下格式：

```
Names: _____________________________________________________
Addresses: _____________________________________________________
Cities: _____________________ States __ Zips _____-____
```

这些列名采用单数，因为您一次只接受一个名字和地址，但是告诉用户在编写查询时必须把这些名称都转变成复数吗？显然只使用单数列名称更直观和简单。

如果所有对象的命名都一致，那么无论是您还是用户都不需要刻意记住到底是使用复数规则还是单数规则。这样做的好处是显而易见的。假设我们决定所有的对象从此以后都使用

复数名称。现在在每个对象名的后面都会带有“s”或者“es”，甚至在一个由多个单词组成的对象名中每个单词的后面都可能带有“s”或者“es”。总是输入这些额外的字母会有什么好处？更容易使用吗？更容易理解吗？更容易记忆吗？显然，这些都不是。

因此，最好的解决方案是：所有的对象名都使用单数。唯一的例外是那些在业务中已经广泛使用的、为大家所接受的术语，如“sales”。

4.12.4 简洁

前面已经提到，清晰绝对不能以牺牲简洁性为代价，但是如果两个名称的意思、可记忆性和描述性相同时，总是选择较短的名称。在开发应用程序的时候，可以为用户和开发人员提供一些可选的列名和表名，并让他们选择含义最清晰的那一个。怎样提供可选列表呢？用辞典和字典。在一个致力于开发优秀的、高效的应用程序的项目小组中，每一个团队成员都应该以一本辞典和一本字典作为基本的工具，而且应该反复强调他们为对象慎重命名的重要性。

4.12.5 对象名辞典

关系数据库最终应该包含一个对象名辞典，就像它们包含一个数据字典那样。这个辞典应该遵循公司的命名标准，并确保名称(在用到的地方)的选择和缩写的一致性。

这些标准可能会要求在为对象命名时使用下划线，以便可方便地把对象的名称分解成不同的组成部分。这还有助于强制对下划线的统一使用，而不是像现在的很多应用程序那样以分散地、不一致地使用下划线。

如果您直接和政府机关或大型公司打交道，则该组织很可能已经有它们自己的对象命名标准。大型组织的对象命名标准在过去的几年中已经渗入到商业市场，而且它们可能构成了您的公司正在使用的命名标准的基础。例如，那些标准也许可以指导您在“Corporation”和“Firm”之间做出选择。如果它们没有这些标准，您应该开发自己的命名标准以便保持一致，不但要和这些基础标准一致，还要和本章中给出的标准一致。

4.13 智能键和列值

智能键(intelligent key)之所以有这样的名字，是因为它们包含了联合在一起的非常有价值的信息。这个术语非常具有误导性，因为它意味着某些正面的或有价值的事情。更好的术语应该是过载键(overloaded key)。一般的分类账和产品代码通常都属于这个类型，而且包含了与其他代码有关的难题和其他问题。而且，过载键包含的难题同样存在于包含多个有意义的数据的非键列中。

典型的过载键和列值的描述如下：“第 1 个字符是区域代码。接下来的 4 个字符是类别编号。最后面的一位数字是成本中心代码；若这是一个导入部分，则要在数字的后面加上 *I* 作为标记；若它是一个具有大量数据的项(如螺丝钉)，则只用 3 个数字表示其类型，而且区域代码是 HD。”

消除过载键和过载列值对于优秀的关系设计是非常重要的。所有建立在这种键上的依赖

关系(通常是到其他表的外键)对系统结构的维护都是非常危险的。遗憾的是，很多应用程序都有过载键，这些键已经使用了很多年而且和公司的任务密切相关。其中有些是在早期的自动化工作中创建的，那时使用的数据库不支持由多个列组成的复合键。其他的则是由于时间的增长，通过强制使一个简短的代码(通常是数字)表达并涵盖比一开始设计时更多的含义和情况，从而导致了过载键。由于一些实际的原因，可能不能立即消除已经存在的过载键。这就使得创建一个新的、关系型的应用程序更加困难。

解决这个问题的方案是创建一组新的键，包括主键和外键，这些键能够正确地规范化数据；接着，确保人们只能通过这些新的键访问表。然后就把过载键当作一个附加的，同时也是唯一的表列。用原来的方法(如在一个查询中匹配过载键)访问它仍然是可以的，但是新构造的键将作为访问的优先方法。随着时间的推移，经过适当的培训，用户将被这些新键所吸引。最终，过载键(和其他过载的列值)都可以简单地被赋值为 NULL 或者从表中删除。

如果没能成功删除过载键和值，那么从数据库中提取信息、确认数据值、确保数据的完整性和修改结构等任务都会非常困难，而且系统开销巨大。

4.14　建议

到现在为止，我们已经讨论过了在设计中所有与生产率有关的主要问题。在这里集中总结一下也许是很有帮助的——因此就有了本节。给出这些建议并不是要告诉您需要做些什么，而是希望您能做出合理的判断，并从别人的教训中吸取经验。这里的目的不是描述开发周期，您对它的了解也许已经超出了自己的期望，而是偏向于一种开发方向，这种方向将在很大程度上改变应用程序的外观和使用方式。仔细心地考虑这些思想将极大地提高生产率和应用程序的用户的满意程度。

下面是实现人性化设计的 10 条建议：

(1) 与用户合作。使他们加入到项目小组中并教会他们关系模型和 SQL。

(2) 和用户一起为表、列、键和数据命名。开发一个应用程序标准来确保名称的一致性。

(3) 使用有意义的、易于记忆的、描述性的、简洁的、单数形式的自然语言单词。以统一的方式使用下划线，或者始终不使用。

(4) 在命名时不能混合不同的级别。

(5) 避免代码和缩写。

(6) 尽最大可能使用有意义的键。

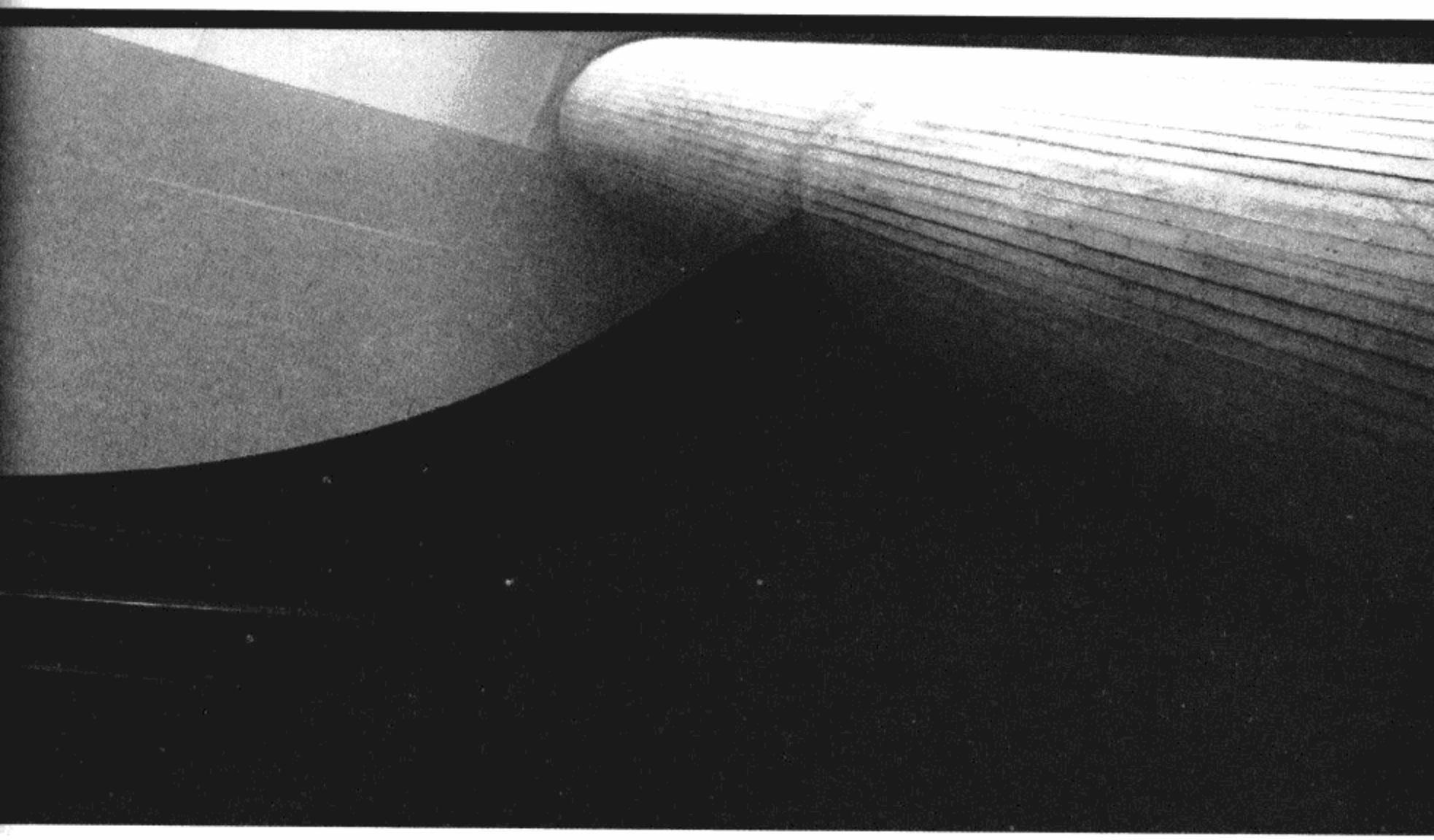
(7) 分解过载键。

(8) 除了对数据进行分析和设计，还要对任务进行分析和设计。记住规范化过程不是设计过程。

(9) 把任务由用户手工操作改为计算机操作。用开发周期和存储空间换取应用程序的易用性是物有所值的。

(10) 不要一味追求开发速度。在分析、设计、测试和调整上多花点时间和心思。

将本章放在介绍命令和函数的前面是有原因的——如果设计非常糟糕，那么后期无论使用什么样的命令，您的应用程序都会受到损害。为功能作计划，为效率作计划，为可恢复性作计划，为安全性作计划，为可用性作计划。总之，为最后的成功作计划。



第 II 部分

SQL 和 SQL*Plus

第 11 章　转换函数与变换函数

第 12 章　分组函数

第 13 章　当一个查询依赖于另一个查询时

第 14 章　一些复杂的技术

第 15 章　更改数据：插入、更新、合并和删除

第 16 章　DECODE 和 CASE：SQL 中的 if-then-else

第 17 章　创建和管理表、视图、索引、群集和序列

第 18 章　分区

第 19 章　Oracle 基本安全

第5章

SQL中的基本语法

通过结构化查询语言(Structured Query Language，SQL)，可以告诉Oracle对哪些信息进行选择(select)、插入(insert)、更新(update)或删除(delete)操作。事实上，这4个动词是向Oracle发布指令的主要单词。您可以用附加命令merge(合并)在一条命令中执行insert和update操作。后面的章节将描述，这些基本命令可以支持闪回版本查询和其他高级功能。

第I部分学习了“关系”意味着什么，如何将表组织成列和行，以及如何命令Oracle从表中选择某些列并且逐行地显示其信息。本章以及接下来的各章将学习对于Oracle所支持的不同的数据类型，如何更加完美地实现上述操作。本部分还将会介绍如何与SQL*Plus进行交互，其中SQL*Plus是Oracle一个功能强大的产品，它基于在适当的位置设置的命令或指

示，获得对 Oracle 的指令，检查其正确性，将其提交给 Oracle，然后修改或重新格式化 Oracle 给出的响应。

在理解 SQL*Plus 所执行任务与 Oracle 所执行任务的差别时，起先可能会有一点迷惑不解，尤其是 Oracle 产生的错误消息只通过 SQL*Plus 传递给用户，但是通读本书后，您就会明白差别所在。开始时，只把 SQL*Plus 看作合作者(coworker)，即遵从用户的指令并且帮助用户更快地完成工作的一个助手。用户通过键盘输入来与这个合作者进行交互。

可以通过键入所示命令来理解本章和后续章节中的示例。Oracle 和 SQL*Plus 程序会作出与示例一样的响应。但是，必须确保将本书中所用的表已经加载到 Oracle 的副本中。

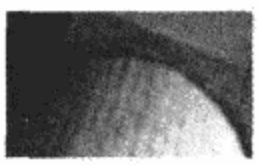
您不需要实际键入相应的内容就可以理解本书所描述的内容。例如，可以将所显示的命令用到自己的表上。但如果将这里所用的表加载到 Oracle 中，并且使用相同的查询，可能更便于学习。访问站点 www.OraclePressBooks.com 可以下载文件。假定已经将演示表加载到一个 Oracle 数据库中，则可以与 SQL*Plus 连接并且通过键入以下命令开始工作：

```
sqlplus
```

如果想从台式客户机运行 SQL*Plus，则可以从 Oracle 软件菜单选项下的 Application Development 菜单选项中选择 SQL Plus 程序选项。这样就启动了 SQL*Plus。请注意，不要键入正式产品名称中间的星号(*)，而且星号也不能出现在程序名中。由于 Oracle 小心防备那些可以访问其存储数据的用户，因此要与它连接必须输入 ID 和口令。Oracle 将显示版本消息，然后要求输入用户名和口令。使用已经创建的、用来保存示例表的账户和口令(如 practice/practice)登录数据库。如果提供了一个有效的用户名和口令，则 SQL*Plus 将宣布已经与 Oracle 连接，然后显示下面的提示：

```
SQL>
```

现在就登录到了 SQL*Plus 中，SQL*Plus 等待您的指令进行下一步操作。



注意：
很多应用程序开发环境直接提供对 Oracle 数据库的 SQL 访问。虽然本章所显示的 SQL 命令可以用于这些工具中，但 SQL*Plus 专用的命令(如 describe)不能用于这些工具中。

如果命令失败，可能有几种原因：Oracle 不在用户的路径内；用户没有被授权使用 SQL*Plus；或者 Oracle 没有正确安装在用户的计算机上。如果得到下面的消息：

```
ERROR: ORA-1017: invalid username/password; logon denied
```

那么可能是输入的用户名或口令不正确，也可能是该用户名还没有在用户的 Oracle 副本上创建。如果试图输入 Oracle 认可的用户名和口令，但是连续 3 次都没有成功，则 SQL*Plus 将终止登录，并显示下面的消息：

```
unable to CONNECT to ORACLE after 3 attempts, exiting SQL*Plus
```

如果得到该消息，则请与公司的数据库管理员联系。假如一切正常，并且已经出现 SQL>

提示符，那么可以开始用 SQL*Plus 进行工作了。

希望退出工作并且离开 SQL*Plus 时，键入下面的命令：

```
quit
```

或者

```
exit
```

5.1　样式

首先是关于样式的一些注释。SQL*Plus 并不在意键入的 SQL 命令是大写还是小写。例如，下面的命令：

```
SeLeCt feaTURE, section, PAGE FROM newsPaPeR;
```

与下面的命令得到完全一样的结果：

```
select Feature, Section, Page from NEWSPAPER;
```

只有在 SQL*Plus 或 Oracle 检查文本数字值的相等性时才会考虑大小写。如果告诉 Oracle 寻找一个满足 Section='f'的行，而 Section 实际等于'F'，则 Oracle 将找不到要求的行(因为 f 和 F 不相等)。除了此用法之外，大小写完全无关紧要(另外需注意，这里所用的'F'称为字面量，这意味着按照字面测试 Section 是否为字母 F，而不是名为 F 的列。单引号将字母括起来告诉 Oracle 它是一个字面量，而不是一个列名)。

就样式来说，本书遵循关于大小写的一些约定，以增加可读性：

- select、from、where、order by、having 和 group by 在正文中始终采用小写形式。
- SQL*Plus 命令也是小写(例如，column、set、save、ttitle 等)。
- IN、BETWEEN、UPPER 和其他 SQL 运算符和函数将采用大写形式。
- 列名是大小写混合形式(如 Feature、EastWest、Longitude 等)。
- 表名用大写形式(如 NEWSPAPER、WEATHER、LOCATION 等)。

在创建自己的查询时可以遵循类似的约定，或者所在的公司已经有了这方面的标准可以使用。甚至可以选择创造自己的标准。但是无论如何，任何标准的目标都是使自己的工作易于阅读和理解。

5.2　创建 NEWSPAPER 表

本书中的示例都是基于合作站点上的脚本创建的表。每个表都是通过 create table 命令来创建，命令中指出表中列名以及这些列的特征。下面是创建 NEWSPAPER 表的 create table 命令，本章的许多示例中都将用到该表：

```
create table NEWSPAPER (
```

```
Feature  VARCHAR2(15) not null,
Section  CHAR(1),
Page     NUMBER
);
```

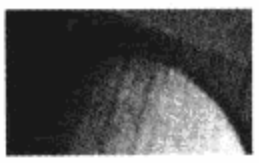
本书后续章节将解释该命令中的所有子句。目前，可以理解为：创建一个名为 NEWSPAPER 的表。它有 3 个列，分别是 Feature(变长字符列)、Section(定长字符列)和 Page(数值列)。Feature 列的值最长可以有 15 个字符，并且每行都必须有一个 Feature 值。Section 值总是只有一个字符的长度。”

在后面的章节中，您还将看到如何扩展该简单命令，以添加约束、索引和存储子句。目前，保持 NEWSPAPER 表简单易读，从而使示例集中在 SQL 上。

5.3 用 SQL 从表中选择数据

图 5-1 显示了一份当地报纸的栏目表。如果这是一个 Oracle 表，而不仅仅是当地报纸前面的纸张和墨水，那么如果键入下面的语句，则 SQL*Plus 将显示如下内容：

```
SQL> select Feature, Section, Page from NEWSPAPER;

FEATURE          S       PAGE
---------------  -  ----------
National News    A           1
Sports           D           1
Editorials       A          12
Business         E           1
Weather          C           2
Television       B           7
Births           F           7
Classified       F           8
Modern Life      B           1
Comics           C           4
Movies           B           4
Bridge           B           2
Obituaries       F           6
Doctor Is In     F           6

14 rows selected.
```



注意：
根据相关配置，该列表可能会出现页面中断。如果出现这种现象，则可以用 set pagesize 命令来增加结果显示页面的尺寸。关于该命令的详细内容，请参考附录 A。

所创建的表和图 5-1 中显示的表有什么区别呢？两个表具有相同的信息，但是格式不同，行的顺序有可能也不同。例如，列标题之间有细微的差别。事实上，它们与 select 语句中所要求的列也有些细微的差别。

Feature	Section	Page
Births	F	7
Bridge	B	2
Business	E	1
Classified	F	8
Comic	C	4
Doctor Is In	F	6
Editorials	A	12
Modern Life	B	1
Movies	B	4
National News	A	1
Obituaries	F	6
Sports	D	1
Television	B	7
Weather	C	2

图 5-1　NEWSPAPER 表

请注意，Section 列只显示一个字母 *S*。此外，尽管命令中键入的列标题混合使用了大小写字母，如下所示：

```
select Feature, Section, Page from NEWSPAPER;
```

但是返回的列标题都是大写字母。

这些变化是由 SQL*Plus 关于如何显示信息的假设导致的。您可以更改这些假设，而且您很可能将更改这些假设，但 SQL*Plus 在默认情况下，假定都将对输入作如下更改：

- 将所有的列标题变为大写
- 列的宽度只能等于 Oracle 中定义的列宽
- 如果列标题是一个函数，则挤出任何空间(详细内容请参阅第 7 章)

第一点是显然的。所使用的列名被转换成大写。第二点不明显。列是如何定义的？要明白这个问题，可以询问 Oracle。使用 describe 命令直接告诉 SQL*Plus 来描述该表，如下所示：

```
describe NEWSPAPER

Name                            Null?     Type
------------------------------- --------  ------------
FEATURE                         NOT NULL  VARCHAR2(15)
SECTION                                   CHAR(1)
PAGE                                      NUMBER
```

所显示的内容是一个描述性表，它列出了 NEWSPAPER 表的列以及各列的定义；describe 命令适用于任何表。请注意该描述中的详细内容与本章前面给出的 create table 命令相匹配。

第 1 列说明所描述表中的列名。

第 2 列(Null?)实际是关于左边相应列的一个规则。在创建 NEWSPAPER 表时，NOT NULL 规则不允许 Oracle 任何用户在表中添加一个 Feature 列为空的新行(NULL 表示空)。当然，在一个表中，如 NEWSPAPER，也可能将相同的规则应用于 3 列中。知道一个栏目的标题，但是不知道它在哪一版以及哪一页上又有什么用呢？但是，从示例的角度考虑，这里只有 Feature 在创建时规定它不能为 NULL。

由于 Section 和 Page 在“Null?”列没有任何内容，因此它们在 NEWSPAPER 表的任意行中都可以为空。

第 3 列(Type)给出了每一列的基本特征。Feature 是一个最长为 15 个字符(字母、数字、符号或空格)的 VARCHAR2(变长字符)列。

虽然 Section 也是一个字符列，但是它只有一个字符的长度。因为表的创建者知道当地报纸中的版号只有一个字母，所以只给列定义了需要的宽度。它是用 CHAR 数据类型(用于定长字符串)来定义的。SQL*Plus 在显示查询

```
select Feature, Section, Page from NEWSPAPER;
```

的结果时，它从 Oracle 知道 Section 最多只有一个字符。同时也假定用户不想使用更多的空间，因此它只用一个字符的宽度来显示该列并且尽可能用列名——这里，只显示字母 *S*。

NEWSPAPER 表中的第 3 列是 Page，它只是一个数值。请注意，Page 列显示为 10 个空格宽度，即使使用的页面没有超过两位数。由于数值通常并不定义最大宽度，因此 SQL*Plus 启动时假定一个最大值。

读者可能也注意到，单独由数字构成的唯一一列的标题(Page)是右对齐的，也就是说，它位于列的右边，但是含有字符的列标题位于左边。这是 SQL*Plus 中列标题的标准对齐方式。至于其他的列特性，将在第 6 章中需要时介绍如何更改对齐方式。

最后，SQL*Plus 将说明它在 Oracle 的 NEWSPAPER 表中找到多少行(请注意显示内容底部的“14 rows selected”)。这称为反馈(feedback)。可以通过设置 feedback 选项使 SQL*Plus 停止显示反馈，如下所示：

```
set feedback off
```

另一个可选的方法是，设置 feedback 作用的最小行数。

```
set feedback 25
```

上一个示例告诉 Oracle，用户在显示的行数超过 25 行时才希望显示反馈信息。除非告诉 SQL*Plus 不同的值，否则 feedback 设为 6。

set 命令是一个 SQL*Plus 命令，也就是一个告诉 SQL*Plus 如何操作的指令。有许多 SQL*Plus 选项可以设置，如 feedback。其中的几个选项将在本章和后续章节中使用。至于完整的列表，请查阅附录 A 中关于 set 的信息。

set 命令有一个名为 show 的副本，它允许查看给 SQL*Plus 下达的指令。例如，可以通过键入下面的指令来检查 feedback 的设置：

```
show feedback
```

SQL*Plus 的响应如下：

```
FEEDBACK ON for 25 or more rows
```

通过 set 命令也可以更改用于显示数值的宽度。键入下面的语句可以检查该宽度：

```
show numwidth
```

SQL*Plus 的回答如下：

```
numwidth 10
```

对于显示那些不超过两位数字的页码来说，由于 10 是一个比较宽的宽度，因此可以通过键入下面的命令来缩短显示宽度：

```
set numwidth 5
```

但是，这意味着所有的数字列都将是 5 位数字的宽度。如果预计有多于 5 位数字宽度的数值，则必须使用大于 5 的数。显示时每一列也可以单独设置。。

```
set numwidth 6

select 123456789123456789 num from DUAL;

   NUM
------
######

set numwidth 7

/

     NUM
-------
1.2E+17
```

5.4　select、from、where 和 order by

从 Oracle 表中选择信息时将用到 SQL 中 4 个基本的关键字：select、from、where 和 order by。任何 Oracle 查询都将用到 select 和 from。

select 关键字告诉 Oracle 用户希望得到的列，而 from 告诉 Oracle 那些列所在表的名字。示例 NEWSPAPER 表说明了如何使用这些关键字。在输入的第一行中，每个列名后面都有一个逗号，但最后一个列名后不加逗号。您将会注意到正确输入的 SQL 查询很像一个英语句子。SQL*Plus 中的查询通常以分号(有时称为 SQL 终止符)结尾。where 关键字告诉 Oracle 想要放在所选信息上的限定符。例如，如果输入：

```
select Feature, Section, Page from NEWSPAPER
```

```
 where Section = 'F';

FEATURE           S    PAGE
---------------   -   -----
Births            F       7
Classified        F       8
Obituaries        F       6
Doctor Is In      F       6
```

则 Oracle 在返回行之前检查 NEWSPAPER 表中的每一行。它跳过那些在 Section 列中没有单个字母 F 的行。它返回 Section 项是 F 的那些行，并且 SQL*Plus 将它们显示出来。

要告诉 Oracle 希望将返回的信息按指定的顺序排列，可以使用 order by。关于排序的要求可以尽可能详细。考虑下面的示例：

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Feature;

FEATURE           S    PAGE
---------------   -   -----
Births            F       7
Classified        F       8
Doctor Is In      F       6
Obituaries        F       6
```

当按照 Page 排序时，上面的显示内容几乎完全逆序，如下所示：

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page;

FEATURE           S    PAGE
---------------   -   -----
Obituaries        F       6
Doctor Is In      F       6
Births            F       7
Classified        F       8
```

在下面的示例中，Oracle 首先按照 Page 排序(仅仅按照 Page 排序时，可以参考前面的程序清单来观察它们的排序)。然后进一步按照 Feature 排序，将 Doctor Is In 显示在 Obituaries 的前面。

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page, Feature;

FEATURE           S    PAGE
---------------   -   -----
```

```
Doctor Is In     F       6
Obituaries       F       6
Births           F       7
Classified       F       8
```

用 order by 也可以将正常顺序逆转，如下所示：

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page desc, Feature;

FEATURE          S    PAGE
---------------  -   -----
Classified       F       8
Births           F       7
Doctor Is In     F       6
Obituaries       F       6
```

desc 关键字表示 descending(降序)。由于它放在 order by 行中的单词 Page 后，因此将页码按降序排列。如果它跟在 order by 行中的单词 Feature 后，对 Feature 列也会有同样的效果。

请注意这些关键字(select、from、where 和 order by)都按照自己的方式组织跟在其后的单词。包含这些关键字的单词组通常称为子句(clause)，如图 5-2 所示。

```
Select Feature, Section, Page          <-- select 字句
from NEWSPAPER                         <-- from 字句
where Section = 'F'                    <-- where 字句
```

图 5-2　子句

5.5　逻辑和值

与 order by 子句可以包含几部分一样，where 子句也可以，只是复杂度高得多。通过在期望返回的内容上对 Oracle 小心地应用逻辑指令，可以控制使用 where 查询数据的范围。这些指令用称为逻辑运算符(logical operator)的数学符号来表示。这只是简要的解释，具体描述请参见附录 A。

下面是一个简单的示例，对 Page 列进行测试，查看是否有等于 6 的值。返回所有满足该条件的行。将跳过任何 Page 不等于 6 的行(换句话说，就是 Page＝6 为 false 的那些行)。

```
select Feature, Section, Page
  from NEWSPAPER
 where Page = 6;

FEATURE          S    PAGE
---------------  -   -----
Obituaries       F       6
```

```
Doctor Is In     F        6
```

等号被称为逻辑运算符(logical operator)，因为它的操作是进行逻辑测试，比较等号两边

NULL 关键字用于测试一列中的某行是否存在数据。如果整列都没有内容，就说该列是 Null。单词 IS 必须与 NULL 以及 NOT NULL 一起使用；等于、大于和小于符号不能和 NULL 以及 NOT NULL 一起使用。

1. 等于、大于、小于和不等于

对于等式和关系值，逻辑测试都可以比较值的大小。下文是对版面号等于 B 的一个的简单测试：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'B';

FEATURE           S    PAGE
---------------   -   -----
Television        B       7
Modern Life       B       1
Movies            B       4
Bridge            B       2
```

下面是页码大于 4 的测试：

```
select Feature, Section, Page
  from NEWSPAPER
 where Page > 4;

FEATURE           S    PAGE
---------------   -   -----
Editorials        A      12
Television        B       7
Births            F       7
Classified        F       8
Obituaries        F       6
Doctor Is In      F       6
```

下面是版面号大于 B(也就是字母表中位于 B 的后面)的测试：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section > 'B';

FEATURE           S    PAGE
---------------   -   -----
Sports            D       1
Business          E       1
Weather           C       2
Births            F       7
Classified        F       8
Comics            C       4
```

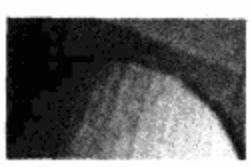
```
Obituaries        F       6
Doctor Is In      F       6
```

与进行大于测试一样，也可以进行小于测试，如下所示(所有页码小于8)：

```
select Feature, Section, Page
  from NEWSPAPER
 where Page < 8;

FEATURE           S        PAGE
---------------   -   ---------
National News     A           1
Sports            D           1
Business          E           1
Weather           C           2
Television        B           7
Births            F           7
Modern Life       B           1
Comics            C           4
Movies            B           4
Bridge            B           2
Obituaries        F           6
Doctor Is In      F           6
```

与等于测试相反的是不等于测试，如下所示：

```
select Feature, Section, Page
  from NEWSPAPER
 where Page <> 1;

FEATURE           S         PAGE
---------------   -   ----------
Editorials        A          12
Weather           C           2
Television        B           7
Births            F           7
Classified        F           8
Comics            C           4
Movies            B           4
Bridge            B           2
Obituaries        F           6
Doctor Is In      F           6
```



注意：

把大于和小于运算符应用于存储在字符数据类型列中的数值时一定要小心。VARCHAR2和CHAR列中的所有值在比较中都视为字符。因此，存储在这些列中的数值作为字符串进行比较，而不是作为数值。如果列的数据类型为NUMBER，则12大于9。如果是字符列，则9大于12，因为字符9比字符1要大。

2. LIKE

SQL 的最强大的功能之一是 LIKE 模式匹配运算符，它能够在数据库列的行中搜索与所描述模式相似的值。它使用两个专有字符来表示要进行哪种匹配：百分号(%)和下划线(_)，百分号称为通配符，下划线称为位标器。例如，要寻找以字母 Mo 开头的所有栏目，可以用下面的命令：

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE 'Mo%';

FEATURE           S         PAGE
---------------   -   ----------
Modern Life       B            1
Movies            B            4
```

这里百分号表示可以接受的任何内容：1 个字符、100 个字符或者 0 个字符。如果前面的字符为 Mo，LIKE 就会发现该 Feature 值。如果查询将搜索条件变为 MO%，则不返回任何行，这是由于 Oracle 区分数据值的大小写。

如果想找到标题的第 3 个字符为 i 的栏目，而并不介意 i 前面的两个字符以及其后的字符集是什么，可以用两个下划线(_ _)来指定那两个位置上的任何字符都是可以接受的。位置 3 必须是小写字母 i，其后的百分号说明可以是任何内容。

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE '__i%';

FEATURE           S         PAGE
---------------   -   ----------
Editorials        A           12
Bridge            B            2
Obituaries        F            6
```

也可以使用多个百分号。为了找到 Feature 标题中含有两个小写字母 o 的那些单词，要使用 3 个百分号，如下所示：

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE '%o%o%';

FEATURE           S         PAGE
---------------   -   ----------
Doctor Is In      F            6
```

为了作比较，下面是一个相同的查询，只不过是寻找两个 i：

```
select Feature, Section, Page from NEWSPAPER
where Feature LIKE '%i%i%';

FEATURE           S         PAGE
---------------   -   ----------
```

```
Editorials          A          12
Television          B           7
Classified          F           8
Obituaries          F           6
```

通过简化对姓名、产品、地址和其他只知道部分内容的项值的搜索，使应用程序更加友好，模式匹配特性在这个过程中充当了重要角色。第 8 章将介绍自 Oracle Database 10g 起如何使用高级的正规表达式查询。

3. NULL 和 NOT NULL

NEWSPAPER 表中没有允许为 NULL 的列，即使在其上所作的 describe 指出允许这样。下面的查询针对 COMFORT 表，除其他数据以外，它还包含 2003 年 4 个指定日期中 San Francisco、California 和 Keene、New Hampshire 的降水量(precipitation)：

```
select City, SampleDate, Precipitation
  from COMFORT;

CITY           SAMPLEDAT PRECIPITATION
-------------  --------- -------------
SAN FRANCISCO  21-MAR-03            .5
SAN FRANCISCO  22-JUN-03            .1
SAN FRANCISCO  23-SEP-03            .1
SAN FRANCISCO  22-DEC-03           2.3
KEENE          21-MAR-03           4.4
KEENE          22-JUN-03           1.3
KEENE          23-SEP-03
KEENE          22-DEC-03           3.9
```

可以找出此查询没有测量出其降水量的城市和日期：

```
select City, SampleDate, Precipitation
  from COMFORT
 where Precipitation IS NULL;

CITY           SAMPLEDAT PRECIPITATION
-------------  --------- -------------
KEENE          23-SEP-03
```

IS NULL 实际上是指示 Oracle 识别缺失数据的列。用户不知道那天的降水量应该是 0、1 还是 5 英寸。由于该值未知，该列中对应的值没有设置为 0；它仍然保留为空。通过使用 NOT，也可以找到那些有对应数据的城市和日期。使用下面的查询：

```
select City, SampleDate, Precipitation
  from COMFORT
 where Precipitation IS NOT NULL;

CITY           SAMPLEDAT  PRECIPITATION
```

```
-------------  ---------  -------------
SAN FRANCISCO  21-MAR-03             .5
SAN FRANCISCO  22-JUN-03             .1
SAN FRANCISCO  23-SEP-03             .1
SAN FRANCISCO  22-DEC-03            2.3
KEENE          21-MAR-03            4.4
KEENE          22-JUN-03            1.3
KEENE          22-DEC-03            3.9
```

虽然 Oracle 允许关系运算符(=、!=等)与 NULL 一起使用，但是这种比较不能返回任何有意义的结果。请使用 IS 和 IS NOT 进行 NULL 值的比较。

5.5.2　值列表的简单测试

如果有测试单一值的逻辑运算符，那么是否有测试许多值(例如，值列表)的运算符？下面关于“值列表逻辑测试”的内容展示了这样一组运算符。

值列表逻辑测试

数值的逻辑测试：

Page IN(1,2,3)	页码为(1,2,3)中某项的页面
Page NOT IN(1,2,3)	页码不为(1,2,3)中某项的页面
Page BETWEEN 6 AND 10	页码为 6~10 之间(包含 6 和 10)某项的页面
Page NOT BETWEEN 6 AND 10	页码小于 6 或者大于 10 的页面

字母(或字符)的逻辑测试：

Section IN('A', 'C', 'F')	属于('A', 'C', 'F')中的版面
Section NOT IN('A', 'C', 'F')	不属于('A', 'C', 'F')中的版面
Section BETWEEN 'B' AND 'D'	版号在'B'~'D'之间(按字母表顺序，含'B'和'D')的版面
Section NOT BETWEEN 'B' AND 'D'	版号小于'B'或大于'D' (字母表顺序)的版面

下面是如何使用这些逻辑运算符的一些示例：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section IN ('A','B','F');

FEATURE          S         PAGE
---------------  -  ----------
National News    A            1
Editorials       A           12
Television       B            7
Births           F            7
Classified       F            8
Modern Life      B            1
Movies           B            4
```

```
Bridge              B              2
Obituaries          F              6
Doctor Is In        F              6

select Feature, Section, Page
  from NEWSPAPER
 where Section NOT IN ('A','B','F');

FEATURE             S            PAGE
---------------     -    ----------
Sports              D              1
Business            E              1
Weather             C              2
Comics              C              4

select Feature, Section, Page
  from NEWSPAPER
 where Page BETWEEN 7 and 10;

FEATURE             S            PAGE
---------------     -    ----------
Television          B              7
Births              F              7
Classified          F              8
```

这些逻辑测试也可以联合使用，如下所示：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
  AND Page > 7;

FEATURE             S            PAGE
---------------     -    ----------
Classified          F              8
```

AND 命令用于合并两个逻辑表达式，要求 Oracle 所检查的任何行都要同时通过两个测试：Section='F'和 Page>7 必须同时为真，才能返回一个行。还有一种合并方法是使用 OR，如果任何一个逻辑表达式证明为真，则它将返回该行：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
    OR Page > 7;

FEATURE             S      PAGE
---------------     -    -----
Editorials          A       12
Births              F        7
Classified          F        8
Obituaries          F        6
```

```
Doctor Is In      F         6
```

这里有一些版号不等于 F 的行，因为它们的页码大于 7，也有一些页码小于或等于 7 但是版号等于 F 的行。

最后，用下面的查询选择 Section F 中页码在 7~10 之间的那些栏目：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
   and Page BETWEEN 7 AND 10;

FEATURE           S     PAGE
---------------   -    -----
Births            F        7
Classified        F        8
```

还有一些用法更为复杂的多值运算符(many-value operator)，它们将在第 9 章中介绍。这些运算符以及上面介绍的那些运算符都可以在本书的“命令和术语参考”部分中找到。

5.5.3　组合逻辑

AND 和 OR 都遵循单词的常见含义。它们实质上能够以任意多种方式组合，但是必须小心使用，因为 AND 和 OR 很容易令人费解。

假如要找到报纸中编辑想隐藏的那些栏目，它们位于 A 版或 B 版的第 2 页以后的某个位置，可以试试下面的查询：

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'A'
    or Section = 'B'
   and Page > 2;

FEATURE           S    PAGE
--------------- -   -----
National News     A        1
Editorials        A       12
Television        B        7
Movies            B        4
```

请注意，从 Oracle 中得到的结果并不是我们所希望的结果。不知何故，A 版的第 1 页也包含在其中。为什么会发生这种事情呢？是否有一种方式能让 Oracle 正确地回答这个问题？尽管 AND 和 OR 都是逻辑连接符，但是 AND 的限制范围更强一点。它以比 OR 更强的方式将其两边的逻辑表达式绑定在一起(从技术的角度来说，AND 的优先级较高)，也就是说，where 子句：

```
where Section = 'A'
   or Section = 'B'
  and Page > 2;
```

解释为：where Section = 'A'，或者 where Section = 'B' 且 Page > 2。如果观察刚才给出的失败的示例，就会明白该解释是如何影响结果的。AND 总是首先起作用。

可以用小括号把希望放在一起解释的表达式括起来，从而打破这种绑定。小括号重写了正常的优先级：

```
select Feature, Section, Page
  from NEWSPAPER
 where Page > 2
   and ( Section = 'A' or Section = 'B' );

FEATURE              S    PAGE
-------------------  -    ----
Editorials           A      12
Television           B       7
Movies               B       4
```

这里的结果就是我们前面希望得到的结果。请注意，该查询的时候也可以先输入版号，结果是一样的，这是因为小括号告诉 Oracle 将什么放在一起解释。请将它与没有使用小括号导致次序改变的第一个示例得到的不同结果作比较。

5.6 where 的另一个用途：子查询

如果前面的“单值逻辑测试”和“值列表逻辑测试”部分中的逻辑运算符不仅可以与单个字面量(如 F)或输入的值列表(如 4、2、7 或 A、C、F)一起使用，还可以与 Oracle 查询返回的值一起使用，会怎么样呢？事实上，这是 SQL 的一项强大的功能。

假设您是“Doctor Is In”栏目的作者，则出版您的专栏的每种报纸都会发送一个包含您文章内容表的副本给您。当然，每个编辑对文章重要性的鉴定都会有一些不同，并据此将文章放在他或她认为合适的版面。如果事先不知道自己栏目的位置，或者不知道和其他什么栏目放在一起，那么如何写一个查询来找到某种本地报纸将您的文章放在何处了呢？可以按照下面的代码查询：

```
select Section from NEWSPAPER
 where Feature = 'Doctor Is In';

S
-
F
```

结果是'F'。知道了这一点，可以执行下面的查询：

```
select FEATURE from NEWSPAPER
 where Section = 'F';

FEATURE
---------------
```

```
Births
Classified
Obituaries
Doctor Is In
```

您的文章与诞辰、讣告以及分类广告放在一起。这两个分开的查询能否合并成一个？答案是肯定的，如下所示：

```
select FEATURE from NEWSPAPER
 where Section = (select Section from NEWSPAPER
                  where Feature = 'Doctor Is In');

FEATURE
---------------
Births
Classified
Obituaries
Doctor Is In
```

5.6.1　从子查询得到单值

实际上，小括号中的 select(称为子查询)返回一个单值 F。然后主查询将该值视为一个字面量' F '，就如同在前面的查询中那样。要记住等号是单值测试。它不能作用于列表，因此如果子查询返回内容多于一行，则会得到一个如下的错误消息：

```
select * from NEWSPAPER
 where Section = (select Section from NEWSPAPER
                  where Page = 1);
where Section = (select Section from NEWSPAPER
                 *
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row
```

所有测试单值的逻辑运算符都可以作用于子查询，只要子查询返回单行。例如，可以查找报纸中版号小于(即字母表中比较靠前的)印有您的专栏版号的所有栏目。select 中的星号是请求表中所有列的一种速记方式，而不需要将它们逐个列出。按照它们在表中创建的次序进行显示：

```
select * from NEWSPAPER
 where Section < (select Section from NEWSPAPER
                  where Feature = 'Doctor Is In');

FEATURE          S    PAGE
---------------  -   -----
FEATURE          S    PAGE
---------------  -   -----
National News    A       1
Sports           D       1
Editorials       A      12
Business         E       1
Weather          C       2
```

```
Television      B       7
Modern Life     B       1
Comics          C       4
Movies          B       4
Bridge          B       2

10 rows selected.
```

这个本地报纸中有 10 个其他栏目排在您的医疗建议前面。

5.6.2 从子查询得到值列表

正如单值逻辑运算符可以作用在子查询上一样，多值运算符也可以。如果子查询返回一行或多个行，则列中的每一行的值都将构成一个列表。例如，假定想知道天气情况为多云的城市和国家。通过下面的查询，您可以创建一个所有城市的完整的天气信息表，以及一个所有城市及其国家的 LOCATION 表：

```
select City, Country from LOCATION;

CITY                        COUNTRY
--------------------------  --------------------------
ATHENS                      GREECE
CHICAGO                     UNITED STATES
CONAKRY                     GUINEA
LIMA                        PERU
MADRAS                      INDIA
MANCHESTER                  ENGLAND
MOSCOW                      RUSSIA
PARIS                       FRANCE
SHENYANG                    CHINA
ROME                        ITALY
TOKYO                       JAPAN
SYDNEY                      AUSTRALIA
SPARTA                      GREECE
MADRID                      SPAIN

select City, Condition from WEATHER;

CITY         CONDITION
-----------  -----------
LIMA         RAIN
PARIS        CLOUDY
MANCHESTER   FOG
ATHENS       SUNNY
CHICAGO      RAIN
SYDNEY       SUNNY
SPARTA       CLOUDY
```

首先将发现哪些城市是多云：

```
select City from WEATHER
 where Condition = 'CLOUDY';

CITY
-----------
PARIS
SPARTA
```

然后，构建一个包含那些城市的列表，并用它来查询 LOCATION 表：

```
select City, Country from LOCATION
 where City IN ('PARIS', 'SPARTA');

CITY                        COUNTRY
--------------------------  --------------------------
PARIS                       FRANCE
SPARTA                      GREECE
```

相同的任务可以通过一个子查询来完成，其中小括号内的 select 构建了一个城市列表，并由 IN 运算符进行测试，如下所示：

```
select City, Country from LOCATION
 where City IN (select City from WEATHER
                  where Condition = 'CLOUDY');

CITY                        COUNTRY
--------------------------  --------------------------
PARIS                       FRANCE
SPARTA                      GREECE
```

其他的多值运算符工作原理类似。基本任务是构建一个子查询，它产生一个能够进行逻辑测试的列表。下面是一些相关的要点：

- 子查询要么只有一列，要么将其选择的列与主查询中小括号内的多列进行比较(第 13 章将介绍)。
- 子查询必须放在小括号中。
- 只产生一行的子查询能够与单值(single-value)运算符或多值(many-value)运算符一起使用。
- 产生多行的子查询只能与多值运算符一起使用。

5.7　组合表

如果已经将数据规范化，则可能需要将两个或多个表组合起来以得到期望的信息。

假设您是特尔斐的神使。雅典人来询问可能影响预料中的斯巴达人攻击力的自然因素以及斯巴达人可能出现的方向：

```
select City, Condition, Temperature from WEATHER;

CITY           CONDITION    TEMPERATURE
-----------    -----------  -----------
LIMA           RAIN                  45
PARIS          CLOUDY                81
MANCHESTER     FOG                   66
ATHENS         SUNNY                 97
CHICAGO        RAIN                  66
SYDNEY         SUNNY                 69
SPARTA         CLOUDY                74
```

您意识到自己的地势比较差，因此查询 LOCATION 表：

```
select City, Longitude, EastWest, Latitude, NorthSouth
  from LOCATION;

CITY                      LONGITUDE E   LATITUDE N
------------------------  --------- - --------  -
ATHENS                       23.43  E     37.58  N
CHICAGO                      87.38  W     41.53  N
CONAKRY                      13.43  W      9.31  N
LIMA                         77.03  W     12.03  S
MADRAS                       80.17  E     13.05  N
MANCHESTER                    2.15  W      53.3  N
MOSCOW                       37.35  E     55.45  N
PARIS                          2.2  E     48.52  N
SHENYANG                    123.27  E     41.48  N
ROME                         12.29  E     41.54  N
TOKYO                       139.46  E     35.42  N
SYDNEY                      151.13  E     33.52  S
SPARTA                       22.27  E     37.05  N
MADRID                        3.41  W     40.24  N
```

这远远超出所需要的内容，而且没有任何天气信息。然而，WEATHER 和 LOCATION 这两个表有一个公共列：City。因此可以将这两个表连接起来从而将这两个表的信息合并在一起。只需要用 where 子句告诉 Oracle 两个表共有的内容：

```
select WEATHER.City, Condition, Temperature, Latitude,
       NorthSouth, Longitude, EastWest
 from WEATHER, LOCATION
where WEATHER.City = LOCATION.City;

CITY          CONDITION     TEMPERATURE LATITUDE  N    LONGITUDE E
-----------   -----------   ----------- --------  -    ---------  -
ATHENS        SUNNY                  97     37.58  N        23.43  E
CHICAGO       RAIN                   66     41.53  N        87.38  W
LIMA          RAIN                   45     12.03  S        77.03  W
MANCHESTER    FOG                    66      53.3  N         2.15  W
```

```
PARIS          CLOUDY          81     48.52 N          2.2 E
SYDNEY         SUNNY           69     33.52 S        151.13 E
SPARTA         CLOUDY          74     37.05 N         22.27 E
```

请注意组合后的表中的行只有两个表中有相同城市的那些行。where 子句仍然执行用户的逻辑，就像它以前在 NEWSPAPER 表中一样。用户给出的逻辑描述了两个表之间的关系。它解读为“选择 WEATHER 表和 LOCATION 表中城市相同的那些行”。如果一个城市只在一个表中，则它在另一个表中没有相同的项。在 select 语句中使用的符号是 TABLE.ColumnName，这里是 WEATHER.City。

select 子句从两个表中选择希望显示的列；表中没有请求的任何列都将被忽略。如果第一行简写为：

```
select City, Condition, Temperature, Latitude
```

则 Oracle 将能确定是哪一个表中的 City。Oracle 将提示列名 City 有多义性。select 子句中的正确措词应该是 WEATHER.City 或者 LOCATION.City。在本例中，用哪一个可选的列并没有太大差别，但在一些复杂的情况中，来自两个或多个表的具有相同名称的列将包含差别很大的数据，对它们进行选择就会比较困难。

where 子句还要求对相同的列名使用表名，从而将表组合起来：“天气.城市=位置.城市”(就是说，WEATHER 表中的 City 列等于 LOCATION 表中的 City 列)。

考虑两个表组合成一个 7 列 7 行的表。用户不想要的内容都去掉了。这里没有 Humidity 列，尽管它是 WEATHER 表的一部分。这里也没有 Country 列，虽然它是 LOCATION 表的一部分。在 LOCATION 表的 14 个城市中，只有包含在 WEATHER 表中的城市保留了下来。where 子句不允许选择其他的城市。

从一个表或多个表中的列构建的表有时称为投影表(projection table)或结果表(result table)。

5.8　创建视图

创建视图不只是为了方便查看。它不仅看起来像一个新表，而且可以命名，同时可以像表一样进行处理。这称为“创建视图”。视图提供了一种隐藏创建连接表的逻辑而只显示表的方式。它的工作方式如下：

```
create view INVASION AS
    select WEATHER.City, Condition, Temperature, Latitude,
           NorthSouth, Longitude, EastWest
     from WEATHER, LOCATION
    where WEATHER.City = LOCATION.City;

    View created.
```

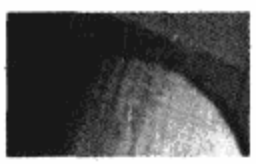
现在可以对 INVASION 进行操作，就像它是一个具有自己的行和列的真正的表。甚至可以要求 Oracle 将它显示出来：

```
describe INVASION

Name                                      Null?    Type
----------------------------------------- -------- ------------
CITY                                               VARCHAR2(11)
CONDITION                                          VARCHAR2(9)
TEMPERATURE                                        NUMBER
LATITUDE                                           NUMBER
NORTHSOUTH                                         CHAR(1)
LONGITUDE                                          NUMBER
EASTWEST                                           CHAR(1)
```

也可以对它进行查询(请注意，不必指定 City 列来自哪个表，因为逻辑隐藏在视图中)：

```
select City, Condition, Temperature, Latitude, NorthSouth,
       Longitude, EastWest
  from INVASION;

CITY         CONDITION    TEMPERATURE  LATITUDE N   LONGITUDE  E
-----------  -----------  -----------  -------- -   ---------  -
ATHENS       SUNNY                 97     37.58 N       23.43  E
CHICAGO      RAIN                  66     41.53 N       87.38  W
LIMA         RAIN                  45     12.03 S       77.03  W
MANCHESTER   FOG                   66      53.3 N        2.15  W
PARIS        CLOUDY                81     48.52 N         2.2  E
SYDNEY       SUNNY                 69     33.52 S      151.13  E
SPARTA       CLOUDY                74     37.05 N       22.27  E
```

虽然有一些 Oracle 函数可以在普通表上使用，但是不能在视图上使用，然而这样的函数非常少，而且绝大多数涉及修改行以及索引表(将在后面的章节讨论)。在大部分情况下，视图都可以跟任何其他表一样进行操作。



注意：
视图不包含任何数据。表包含数据。尽管可以创建包含数据的“物化视图”，但是它们是真实的表，而不是视图。

现在假定您意识到，因为并不需要 Chicago 或者除 Greece 以外的其他城市的信息，所以要更改查询。下面的查询能实现这个目的吗？

```
select  City, Condition, Temperature, Latitude, NorthSouth,
        Longitude, EastWest
 from   INVASION
where   Country = 'GREECE';
```

SQL*Plus 从 Oracle 返回如下消息：

```
where Country = 'GREECE'
        *
ERROR at line 4:
```

```
ORA-00904: "COUNTRY": invalid identifier
```

为什么会这样？其原因是，虽然 Country 是构成 INVASION 视图的其中一个表中的一个真正的列，但是在创建视图的时候它并不在 select 子句中。这样就好像它并不存在。因此，必须返回 create view 语句并且将国家 Greece 包含在其中：

```
create or replace view INVASION as
select WEATHER.City, Condition, Temperature, Latitude,
       NorthSouth, Longitude, EastWest
 from WEATHER, LOCATION
where WEATHER.City = LOCATION.City
  and Country = 'GREECE';

View created.
```

用 create or replace view 命令可以创建一个新版的视图，而不需要首先删除之前的视图。该命令简化了对用户访问视图权限的管理，这将在第 19 章中介绍。

where 子句的逻辑已经得到扩展，可以包括连接两个表，并在其中一个表中的某一列上进行单值测试。现在，来查询 Oracle，将会得到如下响应：

```
select City, Condition, Temperature, Latitude, NorthSouth,
       Longitude, EastWest
from INVASION;

CITY         CONDITION   TEMPERATURE LATITUDE   N LONGITUDE   E
-----------  ----------  ----------- --------   - ---------   -
ATHENS       SUNNY                97     37.58   N      23.43   E
SPARTA       CLOUDY               74     37.05   N      22.27   E
```

这样，您就可以警告雅典人，斯巴达人可能从西南方出现，但是将会因为行军而变得大汗淋漓和疲惫不堪。用一些三角学的知识，甚至可以使 Oracle 计算出他们已经行进了多远。

5.9　扩展视图

视图可以隐藏甚至修改数据，这一功能可以用于多种小场合。可以通过创建一系列简单的视图来生成非常复杂的报表，并且可以限制特定的个人或小组，使他们只看到整个表中的某些内容。

事实上，可以放进查询中的任何限制条件都可以成为视图的一部分。例如，可以让查看工资表的主管只看到他们自己的薪水以及为他们工作的员工的薪水，或者可以限制公司的业务部门只看到他们自己的财政结果，即使该表中包含所有部门的结果。最重要的是，视图并不是过去某一点的数据图。它们是动态的，并且总是反映出底层表中的数据。如果一个表中的即时数据被更改，则基于该表创建的任何视图也跟着更改。

例如，可以创建一个视图，它基于列值进行限制。如下所示，下面的查询限制针对 LOCATION 表的 Country 列，此查询可用于限制通过视图可以看到的那些行：

```
create or replace view PERU_LOCATIONS as
    select * from LOCATION
    where Country = 'PERU';
```

查询 PERU_LOCATION 的用户将不能看到除 Peru 外的任何国家的数据。

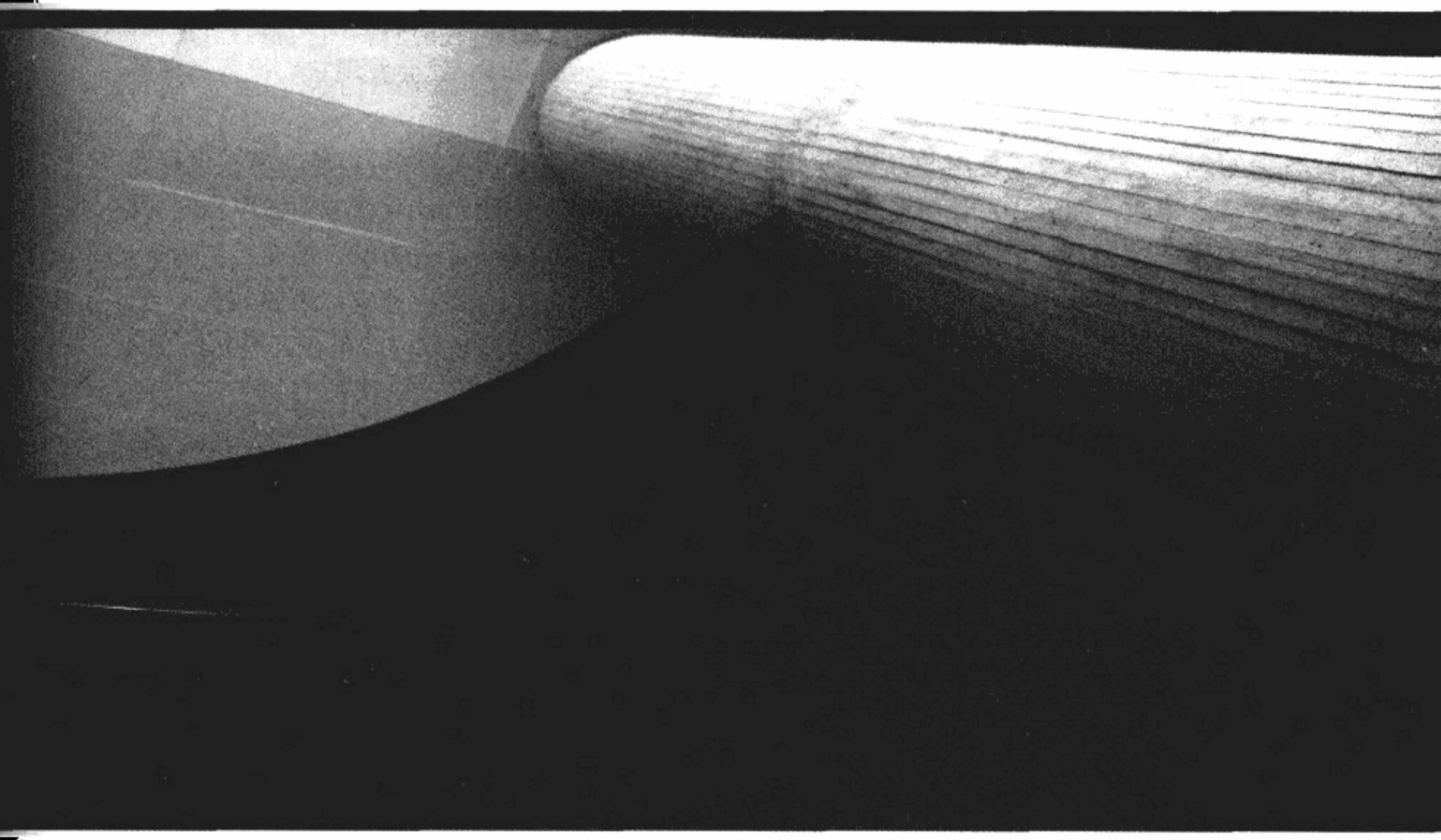
用于定义视图的查询也可以引用伪列(pseudo_column)。伪列是这样一个列：选择它时返回一个值，但它不是表中一个真正的列。选择 User 伪列时将总是返回执行该查询的 Oracle 用户名。因此，如果表中的某一列包含用户名，则可以将这些值与限制它的行的 User 伪列进行比较，如下面的程序清单所示。本例查询 NAME 表。如果其 Name 列的值与输入该查询的用户名相同，则返回相应的行：

```
create or replace view RESTRICTED_NAMES as
    select * from NAME
    where Name = User;
```

在用户要求访问一个表中被选择的行时，这种视图非常有用。它防止用户看到与 Oracle 用户名不匹配的任何行。

视图是功能强大的工具。第 17 章将详细介绍关于视图的知识。

可以基于公共列用 where 子句连接两个表。数据的结果集可以转化成一个视图(具有视图自己的名字)，并且可以像常规表那样进行处理。视图的功能在于：它可以限制或更改用户查看数据的方式，而底层表本身没有受到影响。



第6章

基本的 SQL*Plus 报表和命令

SQL*Plus 通常作为一种操作数据库数据和对数据库执行即席查询的工具，它也可以作为交互式报表编写器。它使用 SQL 从 Oracle 数据库中获取信息，并可创建报表。在创建报表时，您可以轻松地控制标题、列标题、小计和总计，并可以执行数字和文本的重新格式化等操作。它还通过使用 SQL 中的 insert、update、merge 及 delete 命令更改数据库。SQL*Plus 甚至可以作为代码生成器使用，通过一系列 SQL*Plus 命令动态地生成程序并且执行它。

在大多数产品应用程序中，都使用较高级的报表编写器，如基于 Web 的参数驱动报表。SQL*Plus 最常用于简单查询和打印报表。使用 SQL*Plus 可以根据自己的喜欢和需要来格式化报表中的信息，这仅需要少量的 SQL*Plus 命令，即指示 SQL*Plus 如何操作的关键字。这些命令如表 6-1 所示。有关这些命令的详细解释、示例以及附加的功能，请参阅附录 A。

本章将介绍一个用 SQL*Plus 编写的基本报表，以及用于创建报表的功能说明。如果在一开始创建报表时遇到困难，请不要着急。一旦迈出开始的几步，就会发现它们很容易理解，并且很快就会熟悉它们。

可以以交互的方式使用 SQL*Plus 编写 SQL*Plus 报表。也就是说，可以输入有关页标题、列标题、格式、间隔、总计等命令，然后执行 SQL 查询，SQL*Plus 就会立即按照指定的格式规范生成报表。对于快速回答不重复出现的简单问题，这是一个好方法。然而，更为常见的是一些需要定期生成的复杂报表，不仅要在屏幕上显示它，还要打印它。遗憾的是，当退出 SQL*Plus 时，它迅速忘记给它的所有指令。假如仅限制以交互方式使用 SQL*Plus，那么以后再运行这个报表时，必须重新输入所有命令。

另一种方法非常简单。只要把命令逐行地输入到一个文件中即可。然后 SQL*Plus 就会把这个文件当作一个脚本来读取，并且就像正在逐条输入的那样执行您的命令。事实上，这其实创建了一个报表程序，只不过没有程序员和编译器罢了。可使用任何流行的编辑器甚至是(有某种限制的)字处理程序来创建这个文件。

编辑器不是 Oracle 的一部分。编辑器有许多种，并且每个公司和个人都各有所好。Oracle 也意识到这一点，因此决定让用户自己选择编程用的编辑器，而不是在 Oracle 中包装某个编辑器程序，强迫用户使用它。当用户打算使用自己的编辑器程序时，需要挂起 SQL*Plus，跳转到该编辑器程序中，创建或更改 SQL*Plus 报表程序(也称启动文件)，然后再跳转回 SQL*Plus 原来的位置继续运行报表(参见图 6-1)。

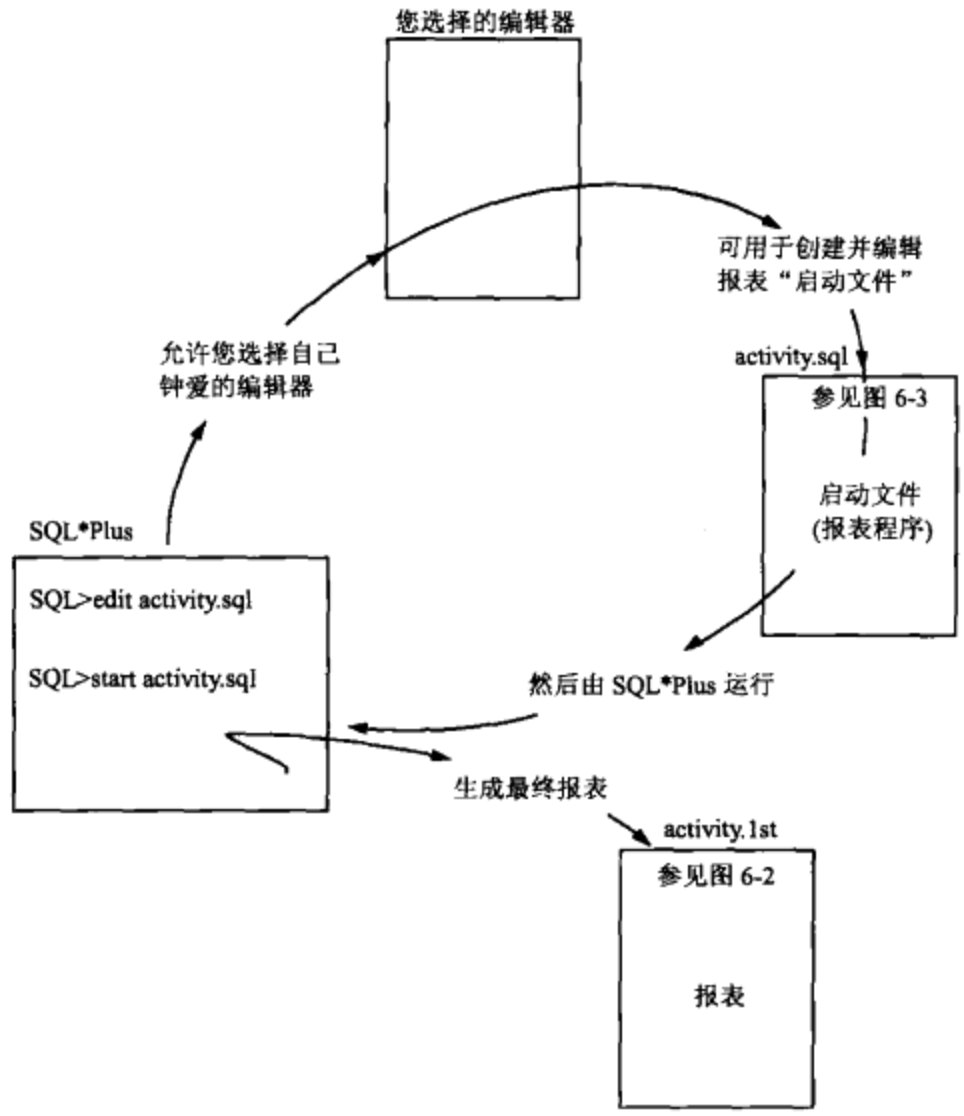


图 6-1 报表创建过程

SQL*Plus 也有自己的内置编辑器，有时称为命令行编辑器(Command Line Editor)，它允许用户不退出 SQL*Plus 就可以迅速修改一个 SQL 查询。有关它的用法将在本章后面介绍。

表 6-1　基本的 SQL*Plus 命令

命　令	定　义
remark	告诉 SQL*Plus 接下来的文字是注释，不是指令
set headsep	标题分隔符标识一个字符，用于告诉 SQL*Plus 将一个标题分为两行或更多行
ttitle	设置报表每一页顶部的标题
btitle	设置报表每一页尾部的标题
column	给出 SQL*Plus 各种关于列的标题、格式和处理的指令
break on	告诉 SQL*Plus 在报表的各部分之间插入空格，或者将小计和总计断开
compute sum	使 SQL*Plus 计算小计
set linesize	为报表的任意行设置最大字符数
set pagesize	为每一页设置最大行数
set newpage	设置页与页之间的空行数
spool	将通常在屏幕上显示的报表导入一个文件中，以便打印
/* */	在 SQL 项内标记注释的开头和结尾。类似于 remark
--	在 SQL 项内标记行内注释的开始。从该标记开始到该行末尾的一切内容都被视为注释。类似于 remark
set pause	使屏幕显示在页与页之间停顿
save	把正在创建的 SQL 查询保存到一个指定的文件中
host	向主机操作系统发送任何命令
start 或@	告知 SQL*Plus 执行已经保存到文件中的指令
edit	使用户迅速离开 SQL*Plus 并进入所选择的编辑器
define_editor	告诉 SQL*Plus 用户所选择的编辑器的名字
exit 或 quit	终止 SQL*Plus

6.1　构建简单的报表

图 6-2 给出了一个显示 3 个月内图书的借阅及归还日期的快速而简单的报表。

Thu Apr 04 page 1

Checkout Log for 1/1/02-3/31/02

NAME	TITLE	CHECKOUTD	RETURNEDD	Days Out
DORAH TALBOT	EITHER/OR	02-JAN-02	10-JAN-02	8.00
	POLAR EXPRESS	01-FEB-02	15-FEB-02	14.00
	GOOD DOG, CARL	01-FEB-02	15-FEB-02	14.00
	MY LEDGER	15-FEB-02	03-MAR-02	16.00
**********				-------
avg				13.00
EMILY TALBOT	ANNE OF GREEN GABLES	02-JAN-02	20-JAN-02	18.00
	MIDNIGHT MAGIC	20-JAN-02	03-FEB-02	14.00
	HARRY POTTER AND THE GOBLET OF FIRE	03-FEB-02	14-FEB-02	11.00
**********				-------
avg				14.33
FRED FULLER	JOHN ADAMS	01-FEB-02	01-MAR-02	28.00
	TRUMAN	01-MAR-02	20-MAR-02	19.00
**********				-------
avg				23.50
GERHARDT KENTGEN	WONDERFUL LIFE	02-JAN-02	02-FEB-02	31.00
	MIDNIGHT MAGIC	05-FEB-02	10-FEB-02	5.00
	THE MISMEASURE OF MAN	13-FEB-02	05-MAR-02	20.00
**********				-------
avg				18.67
JED HOPKINS	INNUMERACY	01-JAN-02	22-JAN-02	21.00
	TO KILL A MOCKINGBIRD	15-FEB-02	01-MAR-02	14.00
**********				-------
avg				17.50
PAT LAVAY	THE SHIPPING NEWS	02-JAN-02	12-JAN-02	10.00
	THE MISMEASURE OF MAN	12-JAN-02	12-FEB-02	31.00
**********				-------
avg				20.50
ROLAND BRANDT	THE SHIPPING NEWS	12-JAN-02	12-MAR-02	59.00
	THE DISCOVERERS	12-JAN-02	01-MAR-02	48.00
	WEST WITH THE NIGHT	12-JAN-02	01-MAR-02	48.00
**********				-------
avg				51.67
				-------
avg				22.58

from the Bookshelf

图 6-2 Bookshelf 借阅报表的输出结果

6.1.1 ①remark

图 6-3 的第一行，即①，是关于启动文件本身的说明文字。该行以下面的词汇开头：

[redacted]rem

它表示 remark。由于 SQL*Plus 忽略以 rem 开头的任何行的内容，因此，用户可以将注释、文档和说明添加到所创建的任何启动文件中。在启动文件的开头用注释的形式给出文件名、文件的创建者以及创建日期、修改者的姓名、修改日期、修改内容以及此文件的目的说明，这不失为一个好主意。随着报表的大量增加并且变得更为复杂，这种方法将被证明是非常有价值的。

```
rem  Bookshelf activity report                                    (1)

set headsep !                                                     (2)

ttitle 'Checkout Log for 1/1/02-3/31/02'                          (3)
btitle 'from the Bookshelf'

column Name format a20                                            (4)
column Title format a20 word_wrapped                              (5)
column DaysOut format 999.99                                      (6)
column DaysOut heading 'Days!Out'                                 (7)

break on Name skip 1 on report                                    (8)
compute avg of DaysOut on Name                                    (9)
compute avg of DaysOut on report


set linesize 80                                                   (10)
set pagesize 60
set newpage 0
set feedback off

spool activity.lst                                                (11)

select Name, Title, CheckoutDate, ReturnedDate,
       ReturnedDate-CheckoutDate as DaysOut /*Count Days*/        (12)
  from BOOKSHELF_CHECKOUT
 order by Name, CheckoutDate;

spool off
```

图 6-3　activity.sql 文件

SQL*Plus 和 SQL

图 6-3 底部的 select 语句从单词“select”开始，并以分号结束，该语句属于结构化查询语言，即用来与 Oracle 数据库进行交互的语言。该页中其他的命令都是 SQL*Plus 命令，用于将 SQL 查询结果变成报表的格式。

SQL*Plus 的 start 命令使 SQL*Plus 读取文件 activity.sql 并执行存储在它内部的指令。回顾一下该启动文件，将看到基本的 SQL*Plus 指令。用户可以用这些指令生成报表或修改与 SQL*Plus 进行交互的方法。这些操作的难易程度取决于用户的经验。它是由一系列简单的 SQL*Plus 指令构成的。

图 6-3 显示了生成该报表的 SQL*Plus 启动文件(这里命名为 activity.sqi)。要在 SQL*Plus 中运行该报表程序，请输入下面的文本：

```
start activity.sql
```

6.1.2 ②set headsep

在图 6-3 的②中，set headsep(headsep 表示 heading separator)后的标点符号告诉 SQL*Plus，在哪里把超出一行的页标题或列标题断开。当第一次激活 SQL*Plus 时，默认的 headsep 字符为竖杠(|)。如果想要在标题中使用竖杠，就会发现它比用其他 headsep 字符简单。这一标题分隔符命令只对 column 命令有效，并不是 select 语句的一部分。

```
set headsep!
```

警告：
选择其他可以出现在标题或列标题中的符号将会导致意想不到的分行。

6.1.3 ③ttitle 和 btitle

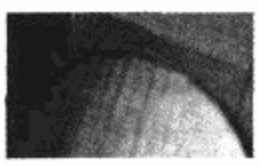
图 6-3 中标号③所指的行

```
ttitle 'Checkout Log for 1/1/02-3/31/02'
```

指示 SQL*Plus 将该顶部标题(top title)置于报表每页的顶端。用户选择的标题必须用单引号括起来。下一行

```
btitle 'from the Bookshelf'
```

类似于 ttitle，只不过它放在每页的底部(单词的首字符 b 表示 bottom，底部)，它也必须用单引号括起来。由于整个标题用单引号括了起来，因此一个撇号(键盘上有相同的符号)将会使 SQL*Plus 误认为该标题已经结束。



注意：
为了在标题中使用撇号或单引号，可在对应的位置上紧挨着放置两个单引号。因为 SQL 和 SQL*Plus 都是用单引号把字符串括起来，所以在 SQL 和 SQL*Plus 中，当需要打印或显示撇号时，都可使用此方法。

当以此方式使用 ttitle 时，SQL*Plus 总是根据所设置的 linesize 使标题居中(linesize 将在本章后面讨论)，并且总是将报表运行的星期、月和日放在页的左上角，页码放在页的右上角。

可以使用 repheader 和 repfooter 命令创建报表的表头和页脚。有关 repheader 和 repfooter 的详细信息，请参阅本书最后的“命令和术语参考”。

6.1.4 column

column 允许更改 select 语句中任何列的标题和格式。看一下图 6-2 所示的报表，第 5 列“Days Out”不是数据库的一列，在图 6-3 所示的查询中把它称为 DaysOut。以下行

```
column DaysOut heading 'Days! Out'
```

给该列重新定义了标签并给该列提供了一个新的标题。此标题将分为两行，因为它的中

间内嵌了 headsep 字符(!)。标记为④的行

```
column Name format  a20
```

将 Name 列的显示宽度设置为 20。a20 中的 a 告知 SQL*Plus 这是一个字母列，而不是数字列。宽度可以设置为任何值，这与数据库中如何定义该列无关。

⑤Name 列定义为 25 个字符宽，使得某些名称可以超过 20 个字符。如果不在报表中定义此列，则任何超过 20 个字符长的名称都将换到下一行。再看一下图 6-2，可以看到有 4 个标题换行，虽然 Title 列定义为 VARCHAR2(100)，但格式化为 a20(参见⑤)。

除了使用 word_wrapped 格式外，还可以选择 truncated，此格式将不显示超过该列的指定格式长度的任意字符。

⑥图 6-3 中标记为⑥的行显示了格式化数据的一个示例：

```
column DaysOut format 999.99
```

此格式定义了一个包含有 5 个数字和一个小数点的列。如果计算一下报表中 DaysOut 列的空间，则将发现有 7 个字符的位置。仅观察 column 命令，或许会以为此列只有 6 个字符的宽度，但是，因为如果数字为负，负号就没有地方放了，所以在数字的左边预留了一个空格。

⑦图 6-3 中的⑦引用了一个在用 SQL*Plus 描述时未出现在该表中的列：

```
column DaysOut heading 'Days!Out'
```

DaysOut 是什么？查看一下图 6-3 底部的 select 语句。DaysOut 出现在以下行中：

```
ReturnedDate - CheckoutDate as DaysOut /*Count Days*/
```

该行告诉 SQL 执行日期算术运算(即计算两个日期之间的天数)，且将计算结果保存在一个列中，并为其指定一个更简单的列名。结果，SQL*Plus 发现一个名为 DaysOut 的列，其所有格式化命令和其他命令都将 DaysOut 列作为表中真正的一列。用于 DaysOut 的 column 命令就是一个示例。“DaysOut”称为列别名(即引用某个列时所用的另一个名字)。

6.1.5　⑧break on

查看图 6-3 中的⑧。请注意，图 6-2 的报表中的每个 Name 的借阅记录是如何组合在一起的。这个结果是由以下行

```
break on Name skip 1 on report
```

以及在该启动文件末尾的 select 语句中的

```
order by Name, CheckoutDate;
```

所产生的。

SQL*Plus 认为每一行都是从 Oracle 返回的，并且在 Name 中跟踪该值。对于前 4 行，由于该值是 DORAH TALBOT，因此 SQL*Plus 显示已经得到的行。在第 5 行，Name 更改为

EMILY TALBOT。SQL*Plus 记住了中断指令，这些指令告诉 SQL*Plus 当 Name 有变化时，应该打破正常的逐行显示，跳过一行再显示。在报表的各 Name 部分间会看到一个空行。仅当名称由 order by 子句排序时，break on 才会在每次 Name 发生变化时跳过一行。这就是为什么 break on 命令必须和 order by 子句一起使用的原因。

您或许注意到了名称 DORAH TALBOT 只显示在其所属部分的第一行，其余的名称也如此。这样做避免了每个名称在每一部分的每一行中重复显示，以便看上去更美观。如果愿意，则可以通过修改 break on 命令强制 SQL*Plus 在每一部分的每一行中重复显示名称，修改后的 break on 语句如下：

```
break on Name duplicate skip 1
```

在图 6-2 中报表输出显示了整个报表中 DaysOut 的平均值。为了得到报表的总计，使用 break on report 命令增加另一个换行。在增加换行时要小心，因为它们都需要通过一个命令来创建，键入两个连续的 break on 命令将导致第一个命令的指令被第二个命令代替。关于 break on 命令在报表中的用法，请参见⑧：

```
break on Name skip 1 on report
```

6.1.6 ⑨compute avg

报表中每个部分的平均值是由⑨中的 compute avg 命令计算的。该命令总是与 break on 命令一起使用，计算的总计值也总是针对由 break on 指定的那部分。将下面这两个相关的命令视为一体是相当明智的：

```
break on Name skip 1 on report
  compute avg of DaysOut on Name
  compute avg of DaysOut on report
```

换句话说，就是告诉 SQL*Plus 计算每个 Name 的 DaysOut 的平均值。SQL*Plus 将首先为 DORAH TALBOT 计算平均值，然后依次为每个 Name 计算平均值。每次 SQL*Plus 遇到一个新的 Name 时，就计算并显示前面的 DaysOut 值的平均值。compute avg 还在 break on 命令所使用的列下面添加一行星号，并且在其下显示 avg 一词。对于有许多列需要累加的报表，每个计算都需要一个单独的 compute avg 语句；如果是用于求和，则每个计算都需要一个单独的 compute sum 语句。在一个大型报表中，可能会有几种不同的换行(例如，对 Name、Title 以及日期等)以及相应的 compute avg 命令。

虽然可以不用 compute sum 命令而单独使用 break on 命令把报表分解成不需要总计的几部分(含有 break on City 的地址就是一个示例)，但是反之不成立。

注意：

每个 compute avg 命令都必须有一个 break on 命令作引导，并且两个命令的 on 部分必须相匹配(如前面示例中的 on Name：break on Name skip 1 on report 和 compute avg of DaysOut on Name)。

以下是基本规则：

- 每个 break on 命令必须有一个相关的 order by 命令。
- 每个 compute avg 命令必须有一个相关的 break on 命令。

这当然是有意义的，但是往往容易忘掉其中的一条。除了 compute avg 以外，还可以在记录集上使用 compute sum、compute count、compute max，或任意其他的 Oracle 的分组函数。

6.1.7　⑩set linesize

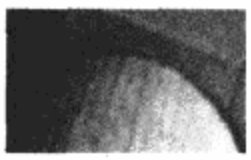
位于图 6-3 中⑩的 4 个命令用于控制报表的总体大小。set linesize 命令控制每行显示的最大字符数。对于稿纸来说，该值通常大约为 70 或 80，除非打印机用的是压缩(窄)字体。

如果在 SQL 查询中放置了过多的信息列，其长度已经超过所分配的 linesize，则 SQL*Plus 会把超过的列换到下一行，并与上一行相应各列对齐。当有大量数据需要显示时，用这种方式确实可以得到较好的效果。

SQL*Plus 还使用 linesize 确定在哪里使 ttitle 居中，在哪里放置日期和页码。日期和页码出现在首行，并且日期的第一个字符到页码的最后一个数字之间的长度总是等于用户所设置的 linesize 的值。

6.1.8　set pagesize

set pagesize 命令设置 SQL*Plus 在每页上打印的总行数，其中包括 ttitle、btitle、column 标题，以及它显示的任意空行。对于稿纸和计算机打印纸来说，该值通常为 66(即 6 行/英寸×11 英寸，美国标准)。set pagesize 和 set newpage 一致。pagesize 的默认值为 14。



注意：

set pagesize 并不设置报表正文的大小(从日期到 btitle 的打印行数)，而是设置页的总长度，以行为单位。

6.1.9　set newpage

set newpage 确切地说应该称为“设置空行”，因为它真正所做的是在报表每页的首行(即日期、页码)前打印空行。这对于在激光打印机上打印单页纸时调整报表的位置，以及对于跳过连续打印纸张之间的小孔很有用。

因此，如果输入下列内容：

```
set pagesize 66
set newpage 9
```

SQL*Plus 就生成一个以 9 个空行开头且后面有 57 行信息(从日期行算起，直到 btitle)的报表。如果增加 newpage 的尺寸，SQL*Plus 就减少每页的信息行，但同时会打印更多的页。

这并不难理解，但是图 6-3 中的⑩做了什么？它的内容如下：

```
set pagesize 60
set newpage 0
```

这是一个奇怪的报表页面尺寸，难道 SQL*Plus 在页之间没有放置空行？不是的，相反，

newpage 后面的 0 打开了一个特别的属性，即 set newpage 0 在每页的日期前生成一个报表页头字符(通常为十六进制 13)。现在的大多数打印机都响应此字符，即立即移动到下一页的页头，然后从那里开始打印报表。set pagesize 60 和 set newpage 0 组合使用生成了一个信息正文为 60 行的报表，并且在每页的开头都含有一个页头字符。这是一种更简洁且更简单地控制页打印的方法。还可以使用 set newpage none 命令，这样会使报表页之间没有空行和换页。

6.1.10 ⑪spool

在计算机早期发展阶段，大多数文件的存储都是在电磁线或磁带中完成的。在那时，把信息写入文件以及使文件假脱机实际上是同义的。假脱机这一术语仍继续使用，现在，假脱机通常指将信息从一个地方移到另一个地方的过程。在 SQL*Plus 中：

```
spool activity.lst
```

告诉 SQL 从 SQL*Plus 中提取所有的输出并将其写入到名为 activity.l st 的文件中。一旦使 SQL*Plus 执行假脱机操作，它就会持续执行直到输入以下命令使它停止为止：

```
spool off
```

例如，这意味着输入了

```
spool work.fil
```

然后输入一个 SQL 查询，如

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F';

FEATURE           S   PAGE
---------------   - ----
Births            F     7
Classified        F     8
Obituaries        F     6
Doctor Is In      F     6
```

或输入一系列的 SQL*Plus 命令，如：

```
set pagesize 60
column Section heading 'My Favorites'
```

或其他命令。无论 SQL*Plus 产生什么样的提示，也无论您得到什么样的错误消息或在假脱机时计算机屏幕上出现什么内容，它都将以文件 work.fil 告终。假脱机不加以区别。它记录从使用 spool 命令到使用 spool off 命令为止的实例中所发生的一切，并返回在图 6-3 中⑪的报表：

```
spool activity.lst
```

这个短语作为命令放在 select 语句之前，并立刻在 select 语句后添加一个 spool off 命令。

如果 spool activity. lst 出现较早，则正在发布的 SQL*Plus 命令将在报表文件的第一页结束。相反，它们进入文件 activity.lst，就是您在图 6-2 所看到的，即 SQL 查询的结果根据用户的指令进行格式化。现在可以随意地打印该文件了，肯定有一个清晰的报表出现在打印机上。

这个命令集将在输出的第一页上打印该 SQL 查询，然后是第二页开始的数据。为了在输出中不显示 SQL 查询，也可以改变命令的顺序，即输入 SQL 查询时不包括分号，按两次回车键，这样命令将仍存储在 SQL*Plus 的缓冲区中，没有被执行。然后可以启动假脱机，并执行命令：

```
(在这里输入 SQL 命令)

spool activity.lst
/
spool off
```

可以在现有的假脱机文件中追加数据。默认的方法是(跟以前的版本一样)创建一个新的输出文件。现在可以利用 spool 命令的 append 或 replace 选项，在与给定名字匹配的现有文件中追加数据，或者分别用新的输出来取代现有的文件。

6.1.11　⑫/* */

图 6-3 中的⑫显示了如何在一个 SQL 语句中嵌入注释。这与前面所讨论的 remark 语句在用法和用途上有所不同。remark(或 rem)必须出现在行首，并且只在其出现的那一行有效。而且，一个多行 SQL 语句中是不允许包含 remark 命令的。例如，下面的代码

```
select Feature, Section, Page
rem this is just a comment
  from NEWSPAPER
 where Section = 'F';
```

是错误的，它将无法运行，并得到一个错误消息。但是，可以用⑫所示的方法在 SQL 中嵌入一个注释，如：

```
select Feature, Section, Page
 /* this is just a comment */
  from NEWSPAPER
 where Section = 'F';
```

其中的奥妙在于/*告诉 SQL*Plus 一个注释已经开始。从/*往后一直到*/为止的一切内容(甚至会持续好几行)SQL*Plus 都认为是注释。用户还可以用字符“--”来开始一个注释，在该行的末尾结束该注释。这种注释与/**/的单行版本有同样的效果，所不同的只是用“--”代替了/**/。

6.1.12　关于列标题的一些说明

下面语句中的对列的重命名

```
ReturnedDate-CheckoutDate as DaysOut
```

与下面语句中给列 DaysOut 提供新标题

```
column DaysOut heading 'Days!Out'
```

之间没有明显的区别，尤其是查看下面的命令：

```
compute avg of DaysOut on Name
```

SQL*Plus 命令只能识别实际出现在 select 语句中的列。每个 column 命令引用 select 语句中的一列。break on 和 compute 命令都只引用 select 语句中的列。column 命令或 compute 命令能够识别列 DaysOut 的唯一理由是，该列从 select 语句本身获得了它自己的名字。通过 SQL 而不是 SQL*Plus 把“ReturnDate-CheckoutDate”重命名为“DaysOut”。

6.2 其他特性

查看启动文件和它所生成的报表，以及查看所有这些格式化和计算是如何完成的并不困难。可以从创建启动文件开始，输入所需要的每一条命令，然后在 SQL*Plus 中运行它，看它是否正确。但是，在首次创建报表时，与 SQL*Plus 进行交互，调整列格式、SQL 查询、标题以及总计，直到用户实际期望的操作开始有眉目，通常都是非常简单的。

6.2.1 命令行编辑器

当输入一条 SQL 语句时，SQL*Plus 会在输入的同时记住每一行，并把它存储在 SQL 缓冲区(存储 SQL 语句的计算机临时存储器)中。假定输入了以下查询：

```
select Featuer, Section, Page
  from NEWSPAPER
 where Section = 'F';
```

SQL*Plus 会做出下列响应：

```
select Featuer, Section, Page
       *
ERROR at line 1:
ORA-00904: "FEATUER": invalid identifier
```

此时您知道把 Feature 拼错了，但是不必重新输入整个查询。命令行编辑器早已就绪并等待指令。首先，要求它列出您的查询：

```
list
```

SQL*Plus 立刻做出如下的响应：

```
1  select Featuer, Section, Page
2  from NEWSPAPER
3* where Section = 'F'
```

请注意，SQL*Plus 一共显示了 3 行并且对每行进行了编号。还在第 3 行编号的旁边加了一个星号，这表示它是编辑命令可以修改的行。但是，如果想更改第 1 行，就输入以下命令，SQL*Plus 会列出相应的内容：

```
list 1
  1* select Featuer, Section, Page
```

第 1 行被显示并且成为当前行。可以输入下列命令来更改它：

```
change /Featuer/Feature

  1* select Feature, Section, Page
```

也可以用 c 取代 change。

可以用 list 或者只用字母 L 再次检查整个查询：

```
list

  1 select Feature, Section, Page
  2 from NEWSPAPER
  3* where Section = 'F'
```

如果用户认为这一次输入正确，则在提示后输入一个斜线(/)。此斜线与 change 命令或编辑器无关，它只是告诉 SQL*Plus 在缓冲区中执行 SQL。

```
/

FEATURE           S   PAGE
---------------   - ----
Births            F     7
Classified        F     8
Obituaries        F     6
Doctor Is In      F     6
```

change 命令要求用斜线或其他字符来标注要更改的文本的开始和结束位置。下一行命令

```
c $Featuer$Feature
```

具有同样的作用。SQL*Plus 查看 change 后面的第一个字符，并假定它是用来标记错误的正文开始和结束位置的字符(这些标记符通常称为分隔符)。还可以按如下所示的方法删除当前行：

```
list

  1  select Feature, Section, Page
  2  from NEWSPAPER
  3* where Section = 'F'

del
```

```
list

  1  select Feature, Section, Page
  2  from NEWSPAPER
```

del 只删除当前行。还可通过指定要删除的第一行和最后一行的行号，给 del 命令传递一个行号范围以便一次性删除多行。例如，要想删除第 3~7 行，可使用 del 3 7 命令。请注意，在要删除的第一行的行号(3)前有一个空格，在所要删除的最后一行的行号(7)前也有一个空格。如果遗漏了 3 和 7 之间的空格，SQL*Plus 会删除第 37 行。要想从第 2 行一直删除到缓冲区的最后一行，可使用 del 2 LAST 命令。list 命令也有同样的语法结构，例如，list 3 7 将列出第 3~7 行。对于 del 命令和 list 命令的所有语法选项，请参阅附录 A。

delete 关键词(拼写完整)将删除所有行并把“delete”作为第 1 行。因为这会引起一些问题，所以要避免输入整个词“delete”。如果想清除整个 select 语句，则请输入下列内容：

```
clear buffer
```

如果想在当前行中追加一些内容，则使用 append 命令(或只使用字母 a)：

```
list 1

  1* select Feature, Section, Page

a  "WhereItIs"

  1* select Feature, Section, Page "WhereItIs"
```

append 命令会把追加的文本放在当前行的末端，中间不留空隙。为了像上面语句那样加入一个空格，可在单词 append 和追加的文本之间输入两个空格。

还可以在当前行后用 input 输入一个全新的行，如下所示：

```
list

  1 select Feature, Section, Page "WhereItIs"
  2* from NEWSPAPER

input where Section = 'A'

list

  1   select Feature, Section, Page "WhereItIs"
  2   from NEWSPAPER
  3* where Section = 'A'
```

为 WhereItIs 列设置列标题：

```
column WhereItIs heading "Where It Is"
```

然后运行这个查询：

```
/

FEATURE           S    Where It Is
---------------   -    -----------
National News     A              1
Editorials        A             12
```

回顾一下，命令行编辑器可以列出(list)输入的 SQL 语句，更改(change)或删除(delete)当前行(由星号标记的行)，把内容追加(append)到当前行的尾部，或在当前行后输入(input)一整行。一旦修正完毕，若在 SQL>提示符后输入斜线(/)，则将执行 SQL 语句。除了 del 命令必须缩写为前 3 个字母外，其他命令都可以缩写成它们对应关键字的第一个字母。

命令行编辑器只能编辑 SQL 语句，不能编辑 SQL*Plus 命令。例如，如果输入了 column Name format a18，并想更改为 column Name format a20，则必须重新输入整个语句(这只是在 SQL*Plus 交互模式下——如果已经在文件中获得了这些命令，则当然可以用自己的编辑器来进行更改)。还要注意，在交互模式下，一旦开始输入 SQL 语句，就必须在输入其他的 SQL*Plus 命令(如 column 格式或 ttitle)前完成它。只要 SQL*Plus 碰到 select 这个词，就认为接下来的所有内容都是 select 语句的组成部分，直到在最后一个 SQL 语句行遇见分号(;)或在最后一个 SQL 语句的下一行开始处遇到斜线为止。

下列两个输入是正确的：

```
select * from LEDGER;

select * from LEDGER
/
```

但是，下面的输入是错误的

```
select * from LEDGER/
```

6.2.2　设置停顿

在新报表的开发中，或者在使用 SQL*Plus 进行数据库快速查询时，将 linesize 设置为 79 或 80，将 pagesize 设置为 24(默认值为 14)，将 newpage 设置为 1 通常是有益处的。下面列出了两个与此相关的命令：

```
set pause 'More. . .'
set pause on
```

该组合的结果将为生成的报表中的每一页产生一个全屏显示，并且在页与页之间停顿以便于查看(“More...”将显示在左下角)，直到按 Enter 键为止。在设计了各种列标题和标题之后，pagesize 可以重新调整纸页，并用下列语句取消停顿：

```
set pause off
```

6.2.3 保存

如果要大范围更改 SQL 语句，或者用户只想在自己的编辑器下工作，则可以通过在交互模式下将 SQL 语句写入文件，来保存迄今为止已经创建的 SQL 语句，如下所示：

```
save fred.sql
```

SQL*Plus 做出如下响应：

```
created file fred.sql
```

现在，SQL 语句(而不是任意 column、ttitle 或其他 SQL*Plus 命令)已经保存在名为 fred.sql(或者是用户自己命名的名字)的文件中了，用户可以使用自己的编辑器编辑它。

如果该文件已经存在，则必须在 save 命令中使用 replace(简写为 rep)选项将新的查询保存到具有此名的文件中。对于本例，语法如下：

```
save fred.sql rep
```

另一种方法是用命令 save fred.sql app 向文件 fred.sql 中追加内容。

6.2.4 存储

可以用 store 命令将当前的 SQL*Plus 环境设置保存到一个文件中。下面将创建一个名为 my_settings.sql 的文件，并且在此文件中存储相应的设置：

```
store set my_settings.sql create
```

如果 my_settings.sql 文件已经存在，则可以用 replace 选项来代替 create，并用新的设置替换旧的文件。还可使用 append 选项向已有文件中添加新的设置。

6.2.5 编辑

每个人都有自己喜爱的编辑器。虽然 SQL*Plus 可以使用字处理程序，但是只有用 ASCII 码格式保存文件时才能使用它(查阅字处理程序手册看是如何实现的)。编辑器本身就是程序。通常只需在操作系统提示符下输入其名字即可调用。在 UNIX 中，通常为如下形式：

```
> vi fred.sql
```

在本例中，vi 是编辑器的名字，fred.sql 表示您想要编辑的文件(在这里，这个原来介绍过的启动文件只作为一个示例，用户可以输入要编辑的文件的真正名字)。虽然其他类型的计算机系统可能没有“>”提示符，但会有其他对应字符。如果用户能用这种方式在计算机上调用编辑器，那么也肯定可以在 SQL*Plus 中调用它，只是不应输入编辑器的名称，而是要输入 edit 命令：

```
SQL> edit fred.sql
```

首先应该告诉 SQL*Plus 编辑器的名称。为此可以在 SQL*Plus 中按如下方法定义该编辑器：

```
define_editor = "vi"
```

注意，在 editor 的 e 前面有一条下划线。SQL*Plus 然后会记住编辑器的名字(直到退出 SQL*Plus 为止)，并允许在任何时候使用它。关于自动地定义编辑器，请参阅 6.2.6 节关于“使用 Login.sql 定义编辑器”的内容。

6.2.6　host

如果上述的编辑命令都不能工作(当然不太可能)，而用户确实想使用自己喜欢的编辑器，则可以通过输入下列内容来调用它：

```
host vi fred.sql
```

host 告诉 SQL*Plus 这是只返回操作系统执行的命令，这和在操作系统提示符下输入 vi fred.sql 作用相同。顺便说一下，使用 host 命令可以从 SQL*Plus 中执行几乎所有的操作系统命令，包括 dir、copy、move、erase、cls 等。

使用 login.sql 定义编辑器

如果想让 SQL*Plus 自动地定义编辑器，则将 define_editor 命令放在 login.sql 文件中。这是一个特殊的文件名，SQL*Plus 在启动时总是先查找该文件。如果 SQL*Plus 发现了 login.sql，就执行此文件中的所有命令，就好像是手工输入这些命令一样。当输入 SQLPLUS 时，它首先查找您所在的目录。

实际上，几乎可以把您在 SQL*Plus 中能使用到的任何命令放进 login.sql 中，其中包括 SQL*Plus 命令和 SQL 语句，所有这些内容都将在 SQL*Plus 给出 SQL>提示符以前执行。这可能是一种以用户喜欢的所有基本布局设置自己的 SQL*Plus 环境的简便方法。下面是一个典型的 login.sql 文件示例：

```
prompt Login.sql loaded.
set feedback off
set sqlprompt 'What now, boss? '
set sqlnumber off
set numwidth 5
set pagesize 24
set linesize 79
define _editor="vi"
```

Oracle Database 10g 提供了 3 个新的预定义环境变量：_DATE、_PRIVILEGE(AS SYSDBA、AS SYSOPER，或空白)和_USER(与 show user 返回值一样)。

另一个文件称为 glogin.sql，用于为所有的数据库用户创建默认的 SQL*Plus 设置。此文件通常存储在 SQL*Plus 的管理目录中，它对于强制多用户的列和环境设置很有用。在 Oracle Database 11g 中，以前存储在 glogin.sql 文件中的 SQL*Plus 设置现在嵌入到可执行文件中。

每个命令的含义都能在附录 A 中找到。

6.2.7 添加 SQL*Plus 命令

把 SQL 语句保存到一个文件中(如 fred.sql)后，就可以向其中添加任何 SQL*Plus 命令。实际上，可以采用类似图 6-3 中的 activity.sql 的风格创建文件。当完成后，可以退出编辑器返回到 SQL*Plus。

6.2.8 启动

一旦返回到 SQL*Plus，就可以通过执行刚才编辑的文件来测试编辑效果：

```
start fred.sql
```

该文件中的所有 SQL*Plus 命令和 SQL 命令将会逐行执行，就好像用户手工输入每条命令一样。如果在文件中包括了 spool 和 spool off 命令，则可以通过编辑器查看其结果。这就是图 6-2 中所示的内容，即启动 activity.sql 并把它的结果假脱机输入到 activity.lst 中。

要创建一个报表，可以按照以下步骤循环进行：

(1) 以交互方式使用 SQL*Plus 创建一个 SQL 查询。当它相对满意时，将其保存在一个文件中，如 test.sql(扩展名.sql，通常专用于保存启动文件，即为了生成报表所执行的脚本)。

(2) 用您喜爱的编辑器编辑 test.sql 文件。在文件中添加 column、break、compute、set 和 spool 命令。通常假脱机到扩展名为.lst 的文件中，如 test.lst。最后退出编辑器。

(3) 返回到 SQL*Plus，启动文件 test.sql。其结果在屏幕上迅速闪过，同时也存储到文件 test.lst 中。打开编辑器检查此文件。

(4) 对 test.sql 做必要的更改，并再次运行它。

(5) 继续此过程，直到报表正确且完美。

6.3 检查 SQL*Plus 环境

从前面的介绍中已经了解，命令行编辑器不能更改 SQL*Plus 命令，因为它只对 SQL 语句——即存储在 SQL 缓冲区的那些行——起作用。此外还了解到，可以保存(save)SQL 语句并把环境设置存储到特定文件中，并使用自己喜爱的编辑器修改文件的内容。

如果想检查某个特定的列是如何定义的，则可以输入以下内容：

```
column DaysOut
```

列名后面不必跟任何内容。然后，SQL*Plus 将列出有关此列的所有指令，如下所示：

```
COLUMN   DaysOut ON
HEADING 'Days!Out' headsep '!'
FORMAT   999.99
```

如果只输入 column 一词，后面没有任何列名，则将显示所有的列。包括 Oracle 所有默认设置的列以及用户自己定义的列：

```
COLUMN    Title ON
```

```
FORMAT    a20
word_wrap

COLUMN    DaysOut ON
HEADING  'Days!Out' headsep '!'
FORMAT    999.99

COLUMN    Name ON
FORMAT    a20
```

ttitle、btitle、break 和 compute 也可以通过只输入其名字显示，后面不必跟任何其他内容。SQL*Plus 根据当前的定义立刻产生响应。下面每个示例中的第一行都是用户输入的内容，后面的行显示了 SQL*Plus 的响应：

```
ttitle
ttitle ON and is the following 31 characters:
Checkout Log for 1/1/02-3/31/02

btitle
btitle ON and is the following 18 characters:
from the Bookshelf

break
break on report nodup
           on Name skip 1 nodup

compute
COMPUTE avg LABEL 'avg' OF DaysOut ON Name
COMPUTE avg LABEL 'avg' OF DaysOut ON report
```

可使用 show 命令查看 set 命令后面的设置(也称为参数)。

```
show headsep
headsep "!" (hex 21)

show linesize
linesize 80

show pagesize
pagesize 60

show newpage
newpage 0
```

关于 set 和 show 命令参数的完全列表，请参阅本书最后的“命令和术语参考”部分。

可以用 ttitle off 和 btitle off 命令来禁用 ttitle 和 btitle 设置。下面显示了这些命令(请注意 SQL*Plus 不对这些命令做出任何响应)：

```
ttitle off
```

```
btitle off
```

对于 columns、breaks 和 computes 的设置，可通过 clear columns、clear breaks 和 clear computes 命令来禁用。下列程序清单中每个示例的第一行都是用户输入的内容；后面的行显示了 SQL*Plus 的响应：

```
clear columns
columns cleared

clear breaks
breaks cleared

clear computes
computes cleared
```

6.4 构件块

本章内容很多，尤其是如果用户不熟悉 SQL*Plus，就会倍感困难。但经过认真的学习，您或许也认为这里所介绍的内容并不困难的。在本章开始时，您看到图 6-3 会感到有些畏惧，但现在再回头看一下，还有什么不理解的行或者有什么不明白的内容吗？如果愿意，则可以把此文件(activity.sql)复制到另一个不同名称的文件中，并根据自己的需要来进行修改，以查询自己的表。归根结底，所生成的任何报表的结构都是非常相似的。

虽然 activity.sql 文件中有许多内容，但它是由一些简单的构件块组成的。这也将是贯穿本书的方法。Oracle 提供了许多构件块，每个块都是可理解和有用的。

前几章学习了如何从数据库中选择数据、选择某些列而忽略其他列，根据设置的逻辑限制来选择某些行，并把两个表组合起来，提供从其中任意一个表都不可得到的信。

本章学习了如何给出 SQL*Plus 遵循的关于格式化并生成完美的报表页和标题的规则。

后几章将对数据进行逐行的更改和格式化。您的专业水平和信心将会随着不断的学习而逐步提高；在第 II 部分结束时，您应该能够在短时间内制作出非常复杂的报表，这对您的公司和个人都是非常有益的。

SQL*Plus 并不是与 Oracle 数据库交互的唯一方法，但对于所有 Oracle 环境，它都是一种常用的方法。本章所介绍的概念——构建 SQL 命令和对这些命令的结果进行格式化——在所有的 Oracle 数据库开发环境中都是常见的概念。

第 7 章

文本信息的收集与更改

本章将介绍串函数(string function)，它是允许用户处理字母串或其他字符串的软件工具。要想快速了解某个函数，请参阅附录 A。本章将集中讨论对文本串的操作。要想搜索单词(其中包括精确查找和模糊匹配)，应该使用第 27 章介绍的 Oracle Text。

Oracle 函数的工作方式有两种。有些函数根据旧对象创建新对象——它们对原来的信息进行修改，如变小写字母为大写字母；另一些函数则告诉用户有关的信息，如一个单词或句子中有几个字符。

注意：
如果您正在使用 PL/SQL，则可以用 creat function 语句创建属于自己的函数。

7.1 数据类型

就像可以基于某些特征(害羞、外向、聪明、愚蠢等)将人分为几种类型一样，也可以根据某些特征把不同的数据分为不同的数据类型(datatype)。

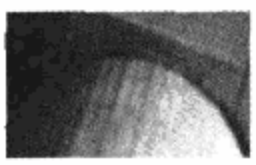
Oracle 的数据类型包括 NUMBER、CHAR(CHARACTER 的简写)、DATE、TIMESTAMP、VARCHAR2、LONG、RAW、LONG RAW、BLOB、CLOB 和 BFILE。前几种类型比较易懂。其余的是特殊的数据类型，您将会在以后遇到。每一种数据类型的完全说明都可以在附录 A 中通过名字或在“数据类型”下找到。每种数据类型也会在后面的章节中详细介绍。与人一样，有些“类型”是重叠的，而有些类型是很少见的。

如果信息为字符类型(VARCHAR2 或 CHAR)，即字母、标点符号、数字和空格的混合形式(又称字母数字)，则需要使用串函数来对它进行修改或给出其相关的信息。对此，Oracle 的 SQL 提供了相当多的工具。

7.2 什么是串

串是一个简单的概念，即排成一行的东西，如房屋、爆米花、珍珠、数字或句子中的字符。

在管理信息中经常遇到串。名字是字符串，如 Juan L'Heureaux。电话号码是数字、破折号，有时还包括小括号组成的串，如(415) 555-2676。甚至一个纯数字，如 5443702，既可被视为一个数字，也可看作是一个字符串。



注意：
仅包含纯数字(根据需要加上小数点和负号)的数据类型称为 NUMBER，通常不作为串来引用。数字可以按照串不能使用的某些方式使用，反之亦然。

包含字母、数字、空格以及其他符号(如标点符号和特殊字符)的串称为字符串，或简称为串。

Oracle 主要有两种串数据类型。CHAR 串始终为定长的。如果设置了长度小于 CHAR 列的串值，则 Oracle 会自动用空格填充。当比较 CHAR 串时，Oracle 用空格将其填充为等长，再进行比较。这意味着如果将不同 CHAR 列中的“character”进行比较，Oracle 就认为这些串是相等的。虽然 VARCHAR2 数据类型为变长的串。虽 VARCHAR 数据类型与 VARCHAR2 类型是同义的，但在今后的 Oracle 版本中也许会有变化，因此应该避免使用 VARCHAR。定长的字符串字段使用 CHAR，而其他所有的字符串字段都使用 VARCHAR2。

本章介绍的简单的 Oracle 串函数如表 7-1 所示。

表 7-1　Oracle 串函数

函 数 名	用　途
||	将两个串连接在一起。“|”符号称为竖线或管道
ASCII	返回数据库字符集中的第一个字符的十进制表示
CHR	返回数据库字符集或国家字符集中与二进制对应的字符
CONCAT	连接两个串(与“||”相同)
INITCAP	首字母大写。也就是使一个单词或一串单词的第一个字母大写
INSTR	在串中定位一个字符
LENGTH	说明串的长度
LOWER	把串中的每个字符转换成小写
LPAD	左填充。在串的左边添加一组字符，使串达到指定长度
LTRIM	左删除。删除串左边的一组字符
NLS_INITCAP	基于 National Language Support(NLS)值使首字母大写
NLS_LOWER	基于 NLS 值转换成小写
NLS_UPPER	基于 NLS 值转换成大写
NLSSORT	基于所选语言进行分类
REGEXP_INSTR、REGEXP_REPLACE、REGEXP_COUNT、REGEXP_LIKE、和 REGEXP_SUBSTR	关于正则表达式的 INSTR、REPLACE、COUNT、LIKE 和 SUBSTR
RPAD	右填充。在串的右边添加一组字符，使串达到指定长度
RTRIM	右删除。删除串右边的一组字符
SOUNDEX	查找发音与所指定示例相似的词
SUBSTR	子串，从一个串中剪切一个子串
TREAT	更改一个表达式的声明类型
TRIM	删除出现在串两边的字符
UPPER	把串的每个字母转换成大写

7.3　表示法

本书中的全部函数用以下格式表示：

```
FUNCTION(string [, option])
```

函数本身大写。它所作用的内容(通常为一个串)以小写显示。单词 string 出现时，它表示一个文本字符串或者表中一个字符列的名字。当实际使用一个串函数时，任何文本都必须用单引号括起来，任何列名却不能带单引号。

每个函数只有一对圆括号。函数所使用到的值以及能够传递给函数的其他信息都放在该圆括号中。

虽然有些函数有选项(option)，可以用来使函数按照您的要求工作，但是这些选项并不总是必需的。选项总是放在方括号[]里。关于如何使用选项的示例，请参阅 7.5.1 节关于 LPAD 和 RPAD 的讨论。

以下是 LOWER 函数的简单格式：

```
LOWER(string)
```

由于单词“LOWER”以及两个圆括号是函数，因此 LOWER 用大写，string 则表示要转换成小写的实际字符串，这里用小写。因此，

```
LOWER('CAMP DOUGLAS')
```

将输出

```
camp douglas
```

字符串‘CAMP DOUGLAS’是一个字面值，即它是 LOWER 函数使用的实际字符串。Oracle 使用单引号标记字面字符串的开始和结束。LOWER 函数中的串还可以是表的列名，在这种情况下，该函数对 select 语句返回的该列中每行的内容进行操作。例如

```
select City, LOWER(City), LOWER('City') from WEATHER;
```

将输出以下结果：

```
CITY          LOWER(CITY)   LOWE
-----------   -----------   ----
LIMA          lima          city
PARIS         paris         city
MANCHESTER    manchester    city
ATHENS        athens        city
CHICAGO       chicago       city
SYDNEY        sydney        city
SPARTA        sparta        city
```

在第 2 列顶端的 LOWER 函数中，CITY 没有用单引号括起来，这告诉 Oracle 它是列名而不是一个字面值。

在第 3 列的 LOWER 函数中，CITY 用单引号括了起来，这表示实际上希望函数 LOWER 直接作用于单词 CITY(即字符串 C-I-T-Y)，而不是对同名的列进行操作。

7.4 连接符(||)

下面的表示法告诉 Oracle 将两个串连接在一起：

```
string || string
```

当然，这两个串可以是列名或字面值。例如：

```
select City||Country from LOCATION;

CITY || COUNTRY
--------------------------------------------------
ATHENSGREECE
CHICAGOUNITED STATES
CONAKRYGUINEA
LIMAPERU
MADRASINDIA
MANCHESTERENGLAND
MOSCOWRUSSIA
PARISFRANCE
SHENYANGCHINA
ROMEITALY
TOKYOJAPAN
SYDNEYAUSTRALIA
SPARTAGREECE
MADRIDSPAIN
```

在这里，城市名的长度在 4～12 个字符之间变化。国家名连接在城市名的右边。这就是连接函数的工作原理：将列或串连接在一起，中间没有任何空格。

当然，这样不易于阅读。为了使其可读性更强，可以用逗号和空格将它们隔开。可以利用一个逗号和一个空格的字面量字符串将 City 列和 Country 列连接起来，如下所示：

```
select City ||', '||Country from LOCATION;

CITY ||','||COUNTRY
--------------------------------------------------
ATHENS, GREECE
CHICAGO, UNITED STATES
CONAKRY, GUINEA
LIMA, PERU
MADRAS, INDIA
MANCHESTER, ENGLAND
MOSCOW, RUSSIA
PARIS, FRANCE
SHENYANG, CHINA
ROME, ITALY
TOKYO, JAPAN
SYDNEY, AUSTRALIA
SPARTA, GREECE
MADRID, SPAIN
```

请注意列标题。关于列标题的介绍，请回顾第 6 章。

还可以使用 CONCAT 函数来连接串。例如，下面的查询

```
select CONCAT(City, Country) from LOCATION;
```

等价于

```
select City||Country from LOCATION;
```

7.5 剪切和粘贴串

本节将学习一些容易混淆的函数：LPAD、RPAD、LTRIM、RTRIM、TRIM、LENGTH、SUBSTR 和 INSTR。这些函数都用于一个共同的目的：剪切和粘贴。

这些函数都进行某种剪切和粘贴任务。例如，LENGTH 返回串中字符的长度。SUBSTR 可以剪切并使用一个子串(substring)，即串的一部分。该子串从串的某一位置开始一直到指定的长度。INSTR 用来查找一组字符在另一个串中的位置。LPAD 和 RPAD 可以在串的左边或右边连接空格或其他字符。LTRIM 和 RTRIM 删除字符串某一端的字符，TRIM 则同时从串两端删除字符。最有意思的是，所有这些函数都可以彼此组合使用，接下来将介绍这种用法。

7.5.1 RPAD 和 LPAD

RPAD 和 LPAD 是十分相似的两个函数。RPAD 允许在列的右边填充一组字符，填充的字符可以为任何字符：空格、句号、逗号、字母或数字、^号、甚至感叹号(!)。LPAD 与 RPAD 的作用差不多，只不过是从左边进行添加。

以下是 RPAD 和 LPAD 的格式：

```
RPAD( string, length [,'set'])

LPAD( string, length [,'set'])
```

这里，string 是数据库中的 CHAR 列或 VARCHAR2 列的名字(或字面是字符串)，length 是填充字符后字符串的总长度(换句话，即宽度)，set 是需要进行填充的字符集，它必须用单引号括起来。方括号表示该字符集(以及其前的逗号)是可选的。如果省略这部分，则函数将自动填充空格。这有时称为默认值，即如果没有告诉函数要使用哪些字符，则它将默认使用空格。

很多用户都会创建带有虚点的表，以便用户从该页的一端移动到另一端。这里将显示 RPAD 如何做到这一点。在本例中，填充后的字符串长度为 35。

```
select RPAD(City,35,'.'), Temperature from WEATHER;

RPAD(CITY,35,'.')                   TEMPERATURE
----------------------------------- -----------
LIMA.................................          45
PARIS................................          81
MANCHESTER...........................          66
ATHENS...............................          97
CHICAGO..............................          66
SYDNEY...............................          69
SPARTA...............................          74
```

请注意这段程序及其结果。RPAD 在 Lima 到 Sparta 的每个城市的右边都连接了虚点，其结果(城市和点)恰好为 35 个字符长。而连接函数(||)则做不到这一点，它只能为每个城市增加固定数量的点，从而导致右边不整齐。

LPAD 的作用相同，所不同的只是在左边添加字符。如果想重新定义城市和温度的格式，使城市右对齐(即都在右边排列)，可使用下面的程序。对于本例，填充后的长度为 11。

```
select LPAD(City,11), Temperature from WEATHER;

LPAD(CITY,1     TEMPERATURE
-----------     -----------
        LIMA              45
       PARIS              81
  MANCHESTER              66
      ATHENS              97
     CHICAGO              66
      SYDNEY              69
      SPARTA              74
```

7.5.2　LTRIM、RTRIM 和 TRIM

LTRIM 和 RTRIM 类似修剪机。它们从串的左边或右边删除不需要的字符。例如，假定在 MAGAZINE 表中有一个包含杂志文章标题的列，但是标题是由不同的人输入的。有些人总是把标题用引号括起来，而有些人只是输入标题；有些人使用句号，而有些人不用；有些人用“The”开始，而有些人不用。该如何修剪这些标题呢？

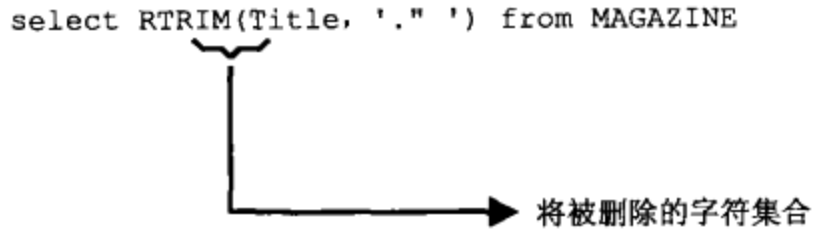
```
select Title from MAGAZINE;

TITLE
-------------------------------------
THE BARBERS WHO SHAVE THEMSELVES.
"HUNTING THOREAU IN NEW HAMPSHIRE"
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
"INTERCONTINENTAL RELATIONS."
```

下面是 RTRIM 和 LTRIM 的格式：

```
RTRIM( string [,' set'])

LTRIM( string [,' set'])
```

这里，string 是数据库中的列名(或一个字面量字符串)，set 是想要删除的字符的集合。如果没有指定字符组，则函数删除空格。

可以一次删除多个字符，为此，只要简单地把要删除的字符组成一个列表(或串)就可以了。首先，去掉右边的引号和句号，如下所示：



上面的查询将产生以下结果：

```
RTRIM(TITLE,'."')
---------------------------------
THE BARBERS WHO SHAVE THEMSELVES
"HUNTING THOREAU IN NEW HAMPSHIRE
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
"INTERCONTINENTAL RELATIONS
```

RTRIM 从每个标题的右边删除双引号和句号。要删除的字符集可以根据需要设为任意长度。Oracle 将反复检查每个标题的右端，直到串中的每个字符都被删除了，也就是说，直到它遇到被删除字符集中未出现的第一个字符时，才停止删除。

7.5.3 组合两个函数

现在做什么呢？可以使用 LTRIM 函数去掉左边的引号。Title 列放在了 RTRIM 函数的中间。本节将学习如何组合函数。

已经知道，当运行下面 select 语句时：

```
select Title from MAGAZINE;
```

返回的结果是 Title 列的内容，如下所示：

```
THE BARBERS WHO SHAVE THEMSELVES.
"HUNTING THOREAU IN NEW HAMPSHIRE"
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
"INTERCONTINENTAL RELATIONS."
```

记住：

```
RTRIM(Title,'."')
```

的目的是接受每个字符串并删除这些串右边的引号，有效地生成一个新列，其内容如下：

```
THE BARBERS WHO SHAVE THEMSELVES
"HUNTING THOREAU IN NEW HAMPSHIRE
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
"INTERCONTINENTAL RELATIONS
```

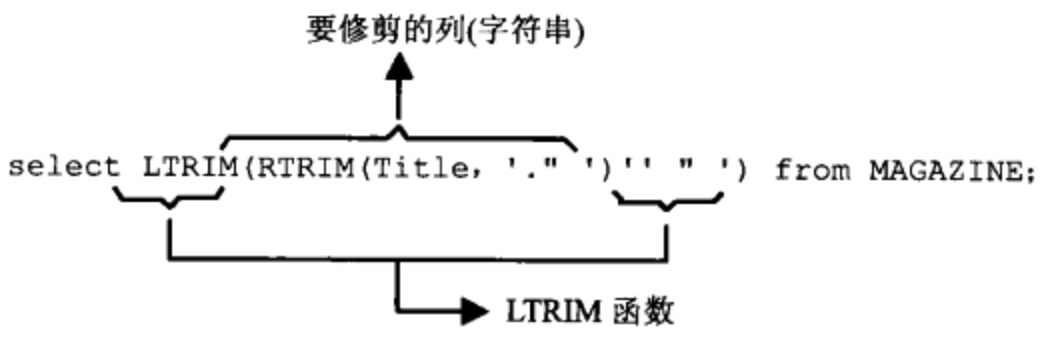
因此，如果假定 RTRIM(Title,'." ')只是一个列名，那么可以用它来代替下面的 string：

```
LTRIM( string,'set')
```

因此，可以简单地输入下面的 select 语句：

```
select LTRIM(RTRIM(Title,'."'),'"') from MAGAZINE;
```

可以用下面的图示来解释上面语句的各部分：



这是您想要的结果吗？该组合函数的结果是什么呢？

```
LTRIM(RTRIM(TITLE,'."'),'"')
---------------------------------
THE BARBERS WHO SHAVE THEMSELVES
HUNTING THOREAU IN NEW HAMPSHIRE
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
INTERCONTINENTAL RELATIONS
```

现在标题被整理整齐了。

第一次看到函数的组合时可能使人迷惑，即使是一个有经验的查询用户也不例外。判断哪个逗号及圆括号与哪个函数相匹配很困难，尤其是当编写的查询不能正确地执行时更是如此；查找哪里少了一个逗号以及哪对圆括号匹配不正确可能真的是一种挑战。

解决上述问题的一个简单方法是将函数拆分成单独的行，一直拆分到使它们能够按您希望的方式工作为止。SQLPLUS 根本不关心在哪里拆分了 SQL 语句，只要不是在一个单词或一个字面量字符串的中间拆开即可。为了更好地理解 RTRIM 和 LTRIM 的组合是如何工作的，可按如下方式输入函数：

```
select LTRIM(
               RTRIM(Title,'."')
                                  ,'"')
  from MAGAZINE;
```

这使您的目的更加明确了，即使该语句分成 4 行输入并且还带有许多空格也可以正常执行。SQLPLUS 会忽略多余的空格。

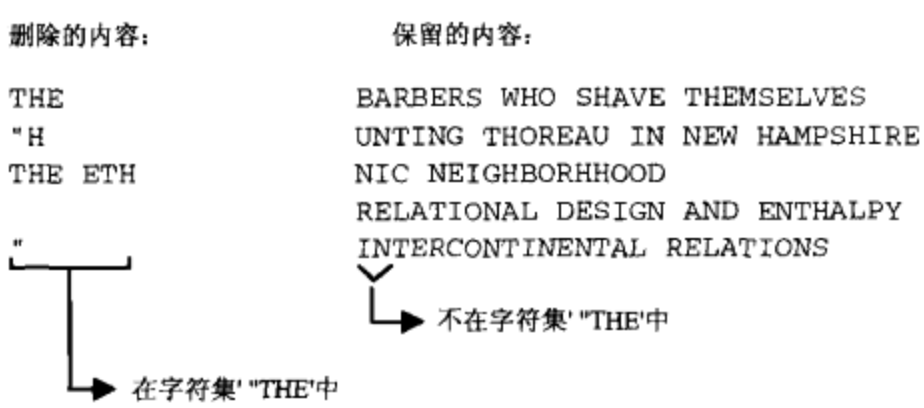
现在，假定要删除两个标题前面的 THE 以及跟在其后的空格(当然，还有之前要删去的双引号)。可以使用以下的代码：

```
select LTRIM(RTRIM(Title,'."'),'"THE ')
  from MAGAZINE;
```

将产生下面的结果：

```
LTRIM(RTRIM(TITLE,'."'),'"THE')
---------------------------------
BARBERS WHO SHAVE THEMSELVES
UNTING THOREAU IN NEW HAMPSHIRE
NIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
INTERCONTINENTAL RELATIONS
```

发生了什么呢？删除了第 2 行和第 3 行不期望删除的内容。为什么呢？因为 LTRIM 忙于查找并删除双引号、T、H、E 或空格。它并不是在查找单词 THE，而是在查找其中的字母。当 LTRIM 第一次发现它所要查找的字母时，它并不退出。而当它发现不属于其字符集中的字符时，才会退出。



换句话说，以下给出的字母组合以及许多其他的字母组合在用于 LTRIM 或 RTRIM 时都会产生同样的效果：

```
'"THE'
'HET"'
'E"TH'
'H"TE'
'ET"H'
```

字符集的字母顺序不影响函数的运行。但是，请注意，字母的大小写至关重要。Oracle 将检查串和字符集中的字母的大小写，只删除完全匹配的字符。

LTRIM 和 RTRIM 用来从串的左边或右边删除字符集中的任何字符。它们并不删除单词。为了做到这一点，需要灵活运用 INSTR、SUBTR 和 DECODE 函数，这些函数将在第 16 章中介绍。

前面的示例清楚地说明了一点：在数据存储到数据库之前，最好先清理和编辑这些数据。如果在输入这些杂志文章的标题时都没有使用引号、句号以及单词 THE，则将大大减少麻烦。

7.5.4 使用 TRIM 函数

上面的示例说明了如何组合使用两个函数，这在处理串操作时是有用的技术。如果要从串的开始和结尾同时删除完全相同的数据，则可以使用 TRIM 函数代替 LTRIM/RTRIM 组合。

TRIM 使用独特的语法。下面的示例说明了 TRIM 函数以及此函数内与之相关的 from 子句的使用方法。在该示例中，我们要删除杂志文章标题的开始和结尾处的双引号。由于双引号是字符串，因此应将其放在两个单引号中：

```
select TRIM('"' from Title) from MAGAZINE;

TRIM('"'FROMTITLE)
----------------------------------
THE BARBERS WHO SHAVE THEMSELVES.
HUNTING THOREAU IN NEW HAMPSHIRE
THE ETHNIC NEIGHBORHOOD
RELATIONAL DESIGN AND ENTHALPY
INTERCONTINENTAL RELATIONS.
```

引号从串的开始和结尾处被删除。如果只想从串的某一端删除字符，则可以使用 leading 或 trailing 子句，如下所示：

```
select TRIM(leading '"' from Title) from MAGAZINE;
select TRIM(trailing '"' from Title) from MAGAZINE;
```

使用 leading 后，TRIM 的作用和 LTRIM 一样；使用 trailing 后，TRIM 的作用和 RTRIM 一样。TRIM 最大的作用是可以同时作用于串的两端，这简化了代码的编写(假设从串的两端删除同样的字符)。

7.5.5　再次使用填充函数

假定打算用虚线和^符号右填充(RPAD)已经删除过部分字符的标题，并要求获取杂志名和页码。可以使用下面的查询：

```
select Name, RPAD(LTRIM(RTRIM(Title,'"'),'."'),47,'-^'), Page
  from MAGAZINE;

NAME             RPAD(LTRIM(RTRIM(TITLE,'"'),'."'),47,'-^')   PAGE
---------------- ---------------------------------------------  -----
BERTRAND MONTHLY THE BARBERS WHO SHAVE THEMSELVES-^-^-^-^-^      70
LIVE FREE OR DIE HUNTING THOREAU IN NEW HAMPSHIRE-^-^-^-^-^     320
PSYCHOLOGICA     THE ETHNIC NEIGHBORHOOD-^-^-^-^-^-^-^-^-^-     246
FADED ISSUES     RELATIONAL DESIGN AND ENTHALPY-^-^-^-^-^-^     279
ENTROPY WIT      INTERCONTINENTAL RELATIONS-^-^-^-^-^-^-^-^     20
```

因为每个函数都用圆括号来括住它要作用的列，所以理解 select 语句中的组合函数的真正诀窍是从外到里同时在左右两边进行阅读，并观察左右两端(甚至计算)圆括号的个数是否相等。

7.5.6　LOWER、UPPER 和 INITCAP

这 3 个相关且非常简单的函数经常一起使用。LOWER 把串或列中的任意字母转换成小写。UPPER 正好相反，将任意字母转换成大写。INITCAP 将串或列中每个单词的首字母转换成大写。

以下是这些函数的格式：

```
LOWER( string)
UPPER( string)
INITCAP( string)
```

返回到 WEATHER 表，注意到每个城市都是用大写字母存储的，如下所示：

```
LIMA
PARIS
ATHENS
CHICAGO
MANCHESTER
SYDNEY
SPARTA
```

因此，

```
select City, UPPER(City), LOWER(City), INITCAP(LOWER(City))
  from WEATHER;
```

将产生以下结果：

```
City        UPPER(CITY)  LOWER(CITY)  INITCAP(LOW
----------  -----------  -----------  -----------
LIMA        LIMA         lima         Lima
PARIS       PARIS        paris        Paris
MANCHESTER  MANCHESTER   manchester   Manchester
ATHENS      ATHENS       athens       Athens
CHICAGO     CHICAGO      chicago      Chicago
SYDNEY      SYDNEY       sydney       Sydney
SPARTA      SPARTA       sparta       Sparta
```

仔细看一看每列所产生的结果，以及在 SQL 语句中产生这些结果的函数。第 4 列说明了如何在 LOWER(City)中应用 INITCAP 函数并以首字母大写来显示，尽管是以大写字母存储的。

另一个示例是 MAGAZINE 表中的 Name 列：

```
NAME
---------------
BERTRAND MONTHLY
LIVE FREE OR DIE
PSYCHOLOGICA
FADED ISSUES
ENTROPY WIT
```

然后用组合的 INITCAP 和 LOWER 函数进行检索，如下所示：

```
select INITCAP(LOWER(Name)) from MAGAZINE;

INITCAP(LOWER(NA
----------------
Bertrand Monthly
Live Free Or Die
Psychologica
Faded Issues
Entropy Wit
```

可将此函数组合应用于 Name 列、整理过的 Title 列和 Page 列(请注意，用户还将对这些

列重命名)：

```
select INITCAP(LOWER(Name)) AS Name,
       INITCAP(LOWER(RTRIM(LTRIM(Title,'"'),'."'))) AS Title,
       Page
  from Magazine;
```

```
NAME              TITLE                                  PAGE
----------------- -------------------------------------- -----
Bertrand Monthly The Barbers Who Shave Themselves          70
Live Free Or Die  Hunting Thoreau In New Hampshire        320
Psychologica      The Ethnic Neighborhood                 246
Faded Issues      Relational Design And Enthalpy          279
Entropy Wit       Intercontinental Relations               20
```

7.5.7 LENGTH

该函数很简单。LENGTH 告诉用户一个串有多长，即串中有多少个字符，其中包括字母、空格和任何其他字符。下面是 LENGTH 的格式：

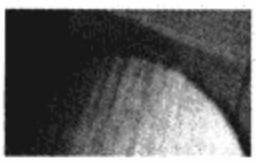
```
LENGTH(string)
```

例如：

```
select Name, LENGTH(Name) from MAGAZINE;
```

```
NAME                 LENGTH(NAME)
---------------      ------------
BERTRAND MONTHLY              16
LIVE FREE OR DIE              16
PSYCHOLOGICA                  12
FADED ISSUES                  12
ENTROPY WIT                   11
```

虽然该函数本身并不是很有用，但它可以作为其他函数的一部分，用于计算某个报表中需要多少空格，或作为 where 或 order by 子句的一部分。



注意：
您不能对一个使用 LONG 数据类型的列使用 LENGTH 之类的函数。

7.5.8 SUBSTR

可以使用 SUBSTR 函数提取出串的一部分。下面是 SUBSTR 函数的格式：

```
SUBSTR( string,start [ ,count])
```

该函数告诉 Oracle 提取出 string 的一个子集，该子集从 start 位置开始，长度为 count 个字符。如果不指定 count，则 SUBSTR 将从 start 开始一直提取到该串的串尾。例如：

```
select SUBSTR(Name,6,4) from MAGAZINE;
```

将得到以下的结果：

```
SUBS
----
AND
FREE
OLOG
 ISS
PY W
```

您可以看到该函数是如何工作的。它从杂志名的第 6 个字符(从左边计数)开始提取出 4 个字符。

将电话号码从通讯簿上分离出来可能更实用。例如，假定有一个 ADDRESS 表，除其他字段外，还包含姓(last name)、名(first name)以及电话号码等字段，如下所示：

```
select LastName, FirstName, Phone from ADDRESS;
```

LASTNAME	FIRSTNAME	PHONE
BAILEY	WILLIAM	213-555-0223
ADAMS	JACK	415-555-7530
SEP	FELICIA	214-555-8383
DE MEDICI	LEFTY	312-555-1166
DEMIURGE	FRANK	707-555-8900
CASEY	WILLIS	312-555-1414
ZACK	JACK	415-555-6842
YARROW	MARY	415-555-2178
WERSCHKY	ARNY	415-555-7387
BRANT	GLEN	415-555-7512
EDGAR	THEODORE	415-555-6252
HARDIN	HUGGY	617-555-0125
HILD	PHIL	603-555-2242
LOEBEL	FRANK	202-555-1414
MOORE	MARY	718-555-1638
SZEP	FELICIA	214-555-8383
ZIMMERMAN	FRED	503-555-7491

假设只想要区号为 415 的电话号码。一种解决方案是分离出一个称为 AreaCode 的列。多考虑一些表和列会免去以后重定格式的诸多麻烦。然而，在本例中，由于区号和电话号码是组合在一个列中的，因此，必须找到一种方法将区号为 415 的号码分离出来。

```
select LastName, FirstName, Phone from ADDRESS
 where Phone like '415-%';
```

LASTNAME	FIRSTNAME	PHONE
ADAMS	JACK	415-555-7530
ZACK	JACK	415-555-6842
YARROW	MARY	415-555-2178

```
WERSCHKY                  ARNY                      415-555-7387
BRANT                     GLEN                      415-555-7512
EDGAR                     THEODORE                  415-555-6252
```

接下来，当给区号为 415 的朋友打电话时，由于区号相同，不必再拨区号，因此可以用另一个 SUBSTR 函数从以上结果中删除这些区号。

```
select LastName, FirstName, SUBSTR(Phone,5) from ADDRESS
 where Phone like '415-%';
```

```
LASTNAME                  FIRSTNAME                 SUBSTR(P
------------------------- ------------------------- --------
ADAMS                     JACK                      555-7530
ZACK                      JACK                      555-6842
YARROW                    MARY                      555-2178
WERSCHKY                  ARNY                      555-7387
BRANT                     GLEN                      555-7512
EDGAR                     THEODORE                  555-6252
```

请注意，这里使用的是 SUBSTR 函数的默认版本。SUBSTR(Phone，5)告诉 SQL 提取出从电话号码的第 5 个字符开始直到串尾的子串。这样就删除了区号。

当然，

```
SUBSTR(Phone,5)
```

与下面的语句作用相同：

```
SUBSTR(Phone,5,8)
```

可以把这个语句与第 6 章讨论的列的重命名技术结合起来，以生成当地朋友的电话号码的列表，如下所示：

```
select LastName ||','||FirstName AS Name,
       SUBSTR(Phone,5) AS Phone
  from ADDRESS
 where Phone like '415-%';
```

```
NAME                                                  PHONE
----------------------------------------------------- --------
ADAMS, JACK                                           555-7530
ZACK, JACK                                            555-6842
YARROW, MARY                                          555-2178
WERSCHKY, ARNY                                        555-7387
BRANT, GLEN                                           555-7512
EDGAR, THEODORE                                       555-6252
```

为了在名字的后面生成虚线，可以添加 RPAD 函数：

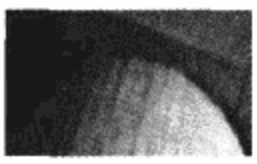
```
select RPAD(LastName ||', '||FirstName,25,'.') AS Name,
       SUBSTR(Phone,5) AS Phone
  from ADDRESS
```

```
 where Phone like '415-%';

NAME                          PHONE
----------------------------  --------
ADAMS, JACK.....              555-7530
ZACK, JACK......              555-6842
YARROW, MARY.....             555-2178
WERSCHKY, ARNY.....           555-7387
BRANT, GLEN.....              555-7512
EDGAR, THEODORE.....          555-6252
```

在 SUBSTR 函数中也可以使用负数。通常，所指定的开始位置值是相对于串的起始位置的。当位置值使用负数时，它相对于串的末尾位置。例如：

```
SUBSTR(Phone,-4)
```

将 Phone 列值从末尾的第 4 个位置作为它的开始点。由于该例中没有指定长度参数，因此返回串的剩余部分。



注意：

这种功能仅用于 VARCHAR2 数据类型的列。不能用于 CHAR 数据类型的列。因为 CHAR 列是定长的列，所以将用空格填充它们的值，将长度扩展到列的全长。在用于 CHAR 列的 SUBSTR 函数中，若其位置值使用负数，则将从列的末尾开始确定相对的开始位置，而不是从字符串的末尾开始确定。

下面的示例说明了在 VARCHAR2 列的 SUBSTR 函数中使用负数时的结果：

```
select SUBSTR(Phone,-4)
  from ADDRESS
 where Phone like '415-5%';

SUBS
----
7530
6842
2178
7387
7512
6252
```

SUBSTR 函数中的 count 值必须为正数，或不指定该值。使用负的 count 值将返回一个 NULL 值。

7.5.9 INSTR

INSTR 函数允许在串中简单地或复杂地搜索一个串成一组字符，除了 INSTR 不能删除任何内容外，它与 LTRIM 和 RTRIM 函数没有什么不同。它只是告诉您查找的内容在串的什么位置。这类似于第 5 章讨论的 LIKE 逻辑运算符，只是 LIKE 只能用于 where 和 having 子

句，而 INSTR 除了不能用在 from 子句外，能用于任何子句。当然，LIKE 能够用于非常复杂的模式搜索，而这些模式搜索即使可能使用 INSTR，也会很困难。下面是 INSTR 的格式：

```
INSTR( string, set [, start [, occurrence ] ])
```

INSTR 在串中寻找特定的字符集。它有两个选项，一个选项包含在另一个选项中。第一个选项为默认值，它将从串的第 1 个位置开始搜索该字符集。如果指定 start 的位置，则它将跳过前面所有的字符到该位置并开始搜索。

第二个选项是 occurrence。字符集可能在一个串中多次出现，而您真正关心的可能只是某些内容是否在串中多次出现。默认情况下，INSTR 将查找字符集第一次出现的位置。通过添加 occurrence 选项，例如把它设置为 3，将强迫 INSTR 跳过前两次与字符集的匹配，给出第 3 次匹配的位置。

下面的示例将解释前面介绍的内容。回顾一下存放杂志文章的表，下面是作者的列表：

```
select Author from MAGAZINE;

AUTHOR
-------------------------
BONHOEFFER, DIETRICH
CHESTERTON, G.K.
RUTH, GEORGE HERMAN
WHITEHEAD, ALFRED
CROOKES, WILLIAM
```

为了查找字母 O 第一次出现的位置，使用 INSTR，而不添加其他选项，字符集设为 'O'(请注意使用单引号，因为这是一个字面值)，如下所示：

```
select Author, INSTR(Author,'O') from MAGAZINE;

AUTHOR                          INSTR(AUTHOR,'O')
-------------------------    -----------------
BONHOEFFER, DIETRICH                           2
CHESTERTON, G.K.                               9
RUTH, GEORGE HERMAN                            9
WHITEHEAD, ALFRED                              0
CROOKES, WILLIAM                               3
```

当然，它与下面的语句是相同的：

```
select Author, INSTR(Author,'O',1,1) from MAGAZINE;
```

如果 INSTR 查找字母 O 第二次出现的位置，则将发现：

```
select Author, INSTR(Author,'O',1,2) from MAGAZINE;

AUTHOR                         INSTR(AUTHOR,'O',1,2)
-------------------------     -------------------
BONHOEFFER, DIETRICH                             5
CHESTERTON, G.K.                                 0
RUTH, GEORGE HERMAN                              0
```

```
WHITEHEAD, ALFRED                                    0
CROOKES, WILLIAM                                     4
```

INSTR 在 Bonhoeffer 名字中的第 5 个字符位置找到了第 2 个 O，并在 Crookes 名字中的第 4 个字符位置找到了第 2 个 O。Chesterton 只有一个 O，因此，它、Ruth 以及 Whitehead 的结果为零，即不成功，因为没有找到第 2 个 O。

为了告诉 INSTR 查找第二次出现的字符，还必须告诉它从哪里开始查找(在本例中为位置 1)。start 的默认值为 1，这表示如果不指定任何值，它就为 1，但是要使用 occurrence 选项，就需要一个 start 值，因此必须同时指定二者。

如果 set 中不止有一个字符而是有几个字符，则 INSTR 给出该字符集中第一个字符的位置，如下所示：

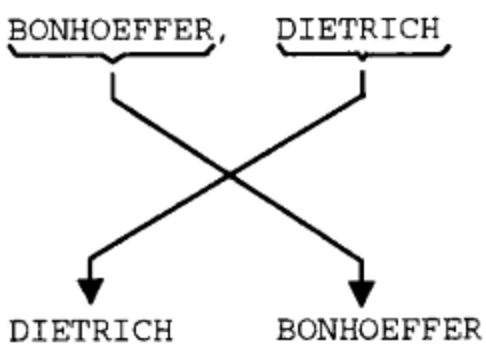
```
select Author, INSTR(Author,'WILLIAM') from MAGAZINE;

AUTHOR                         INSTR(AUTHOR,'WILLIAM')
------------------------ ----------------------
BONHOEFFER, DIETRICH                                 0
CHESTERTON, G.K.                                     0
RUTH, GEORGE HERMAN                                  0
WHITEHEAD, ALFRED                                    0
CROOKES, WILLIAM                                    10
```

还有很多有用的应用。例如，在 MAGAZINE 表中：

```
select Author, INSTR(Author,',') from MAGAZINE;

AUTHOR                          INSTR(AUTHOR,',')
------------------------       -----------------
BONHOEFFER, DIETRICH                            11
CHESTERTON, G.K.                                11
RUTH, GEORGE HERMAN                              5
WHITEHEAD, ALFRED                               10
CROOKES, WILLIAM                                 8
```

在这里，INSTR 搜索作者姓名串中的逗号，然后报告它在每个串中的位置。

假设想重新定义作者姓名的格式，不按照正式的“last name/comma/first name”(姓/逗号/名)方法，而是按日常用法显示它们，如下所示：



为此，可使用 INSTR 函数和 SUBSTR 函数查找逗号的位置，并用该位置告诉 SUBSTR

从哪里开始提取。按此步骤进行，首先在上面的程序清单中查找逗号。

需要两个 SUBSTR 函数，一个提取逗号前的作者的姓，另一个提取从逗号后两个位置开始直到串尾的作者的名。

首先，看一下从位置 1 到逗号前提取内容的函数：

```
select Author, SUBSTR(Author,1,INSTR(Author,',')-1)
  from MAGAZINE;

AUTHOR                         SUBSTR(AUTHOR,1,INSTR(AUT
-----------------------------  -------------------------
BONHOEFFER, DIETRICH           BONHOEFFER
CHESTERTON, G.K.               CHESTERTON
RUTH, GEORGE HERMAN            RUTH
WHITEHEAD, ALFRED              WHITEHEAD
CROOKES, WILLIAM               CROOKES
```

接着，看一下从逗号后两个位置开始直到串尾提取内容的函数：

```
select Author, SUBSTR(Author,INSTR(Author,',')+2) from MAGAZINE;

AUTHOR                         SUBSTR(AUTHOR,INSTR(AUTHO
-----------------------------  -------------------------
BONHOEFFER, DIETRICH           DIETRICH
CHESTERTON, G.K.               G.K.
RUTH, GEORGE HERMAN            GEORGE HERMAN
WHITEHEAD, ALFRED              ALFRED
CROOKES, WILLIAM               WILLIAM
```

看一下这两个函数与连接函数的组合。连接函数在两者之间放一个空格，得到的新列重命名为 ByFirstName：

```
column ByFirstName heading "By First Name"

select Author, SUBSTR(Author,INSTR(Author,',')+2)
                 ||' '||
                 SUBSTR(Author,1,INSTR(Author,',')-1)
                 AS ByFirstName
  from MAGAZINE;

AUTHOR                         By First Name
-----------------------------  -------------------------
BONHOEFFER, DIETRICH           DIETRICH BONHOEFFER
CHESTERTON, G.K.               G.K.CHESTERTON
RUTH, GEORGE HERMAN            GEORGE HERMAN RUTH
WHITEHEAD, ALFRED              ALFRED WHITEHEAD
CROOKES, WILLIAM               WILLIAM CROOKES
```

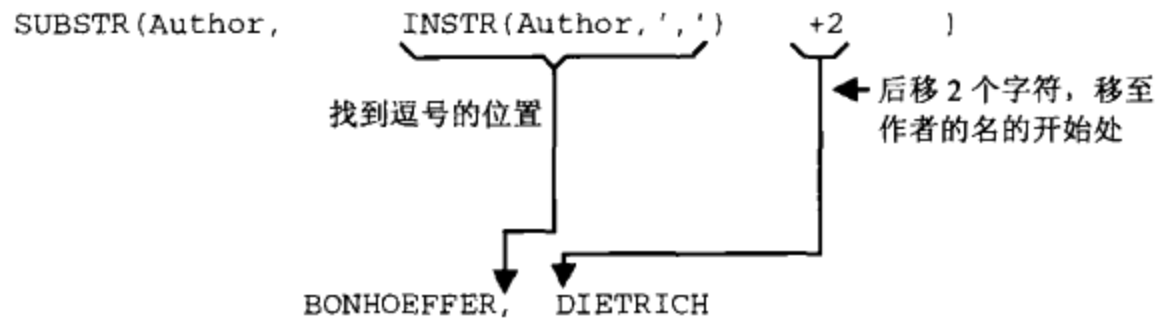
虽然这样的 SQL 语句看起来很复杂，但它是通过简单的逻辑构建的，可以按照相同的方法进行拆分。以 Bonhoeffer 为例。第一部分如下所示：

```
SUBSTR(Author,INSTR(Author,',')+2)
```

该语句告诉 SQL 提取 Author 列中的一个子串，该子串从逗号右边两个位置开始直到串尾。这将提取出 DIETRICH，即作者的名。

作者的名的开始位置是通过定位其姓后面的逗号找到的(INSTR 函数完成的)，然后右移两个位置(这是作者的名开始的地方)。

下图说明了 INSTR 函数(加 2)是如何作为 SUBSTR 函数的 start 参数的值：



以下是组合语句的第二部分：

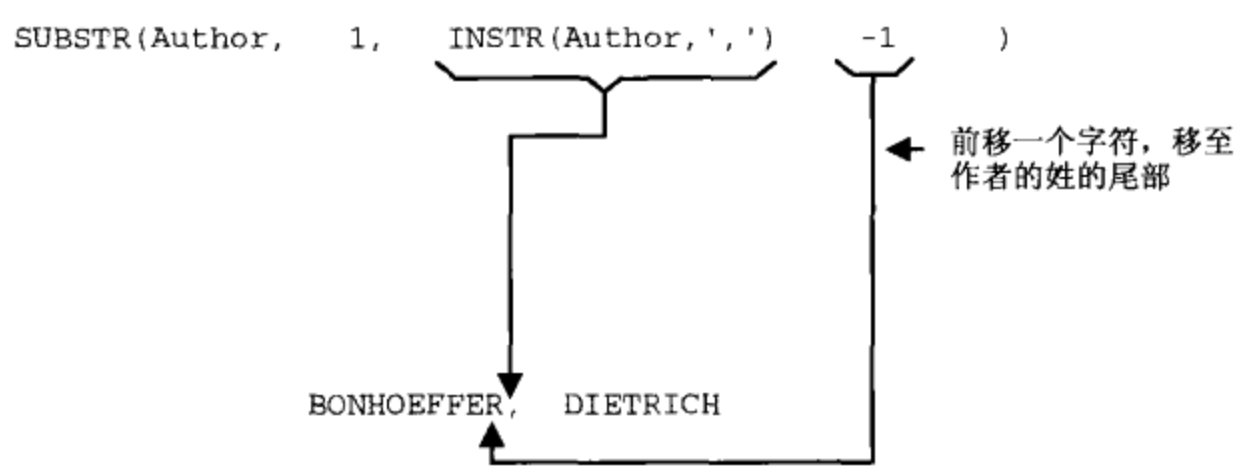
```
||' '||
```

当然，它只告诉 SQL 在中间连接一个空格。

以下是组合语句的第三部分：

```
SUBSTR(Author,1,INSTR(Author,',')-1)
```

它告诉 SQL 提取出从位置 1 开始到逗号前一个位置的子串，即作者的姓：



第 4 部分指定一个列的别名：

```
AS ByFirstName
```

能够完成这种转换是因为 MAGAZINE 表中每个 Author 记录都遵从相同的格式约定。在每个记录中，姓始终是串中的第一个单词并且后面有一个逗号。这样可用 INSTR 函数搜索这个逗号。一旦确定了逗号的位置，就可以确定该串的哪部分为姓，那么串其余的部分为名。

但实际情况通常并不是这样的。姓名很难套用标准格式。姓也许包括前缀(如 von Hagel 中的 von)或后缀(如 Jr.、Sr.、和 III)。如果使用上例中的 SQL 语句，名字 Richards, Jr., Bob 将被转换成 Jr., Bob Richards。

由于缺少标准的格式，因此很多应用程序会分别存储姓和名。头衔(如 MD)经常存储在另一列中。当存储这样的数据时，第二个选项将强制它转换为一个单一的格式，并且在需要

时用 SUBSTR 和 INSTR 处理该数据。

7.5.10　ASCII 和 CHR

ASCII 和 CHR 函数在即席查询中很少使用。CHR 把数值转换为等价的 ASCII 字符串。

```
select CHR(70)||CHR(83)||CHR(79)||CHR(85)||CHR(71)
    as ChrValues
  from DUAL;
CHRVA
-----
FSOUG
```

Oracle 根据数据库的字符集，将 CHR(70)转换成 F，CHR(83)转换成 S 等。

ASCII 函数执行相反的操作。但是，如果给它传递一个串，它只对该串的第一个字符起作用：

```
select ASCII('FSOUG') from DUAL;

 ASCII('FSOUG')
 --------------
             70
```

要查看每个字符的 ASCII 值，可以使用 DUMP 函数。

```
select DUMP('FSOUG') from DUAL;
DUMP('FSOUG')
-----------------------------
Typ=96 Len=5: 70,83,79,85,71
```

7.6　在 order by 和 where 子句中使用串函数

串函数可以用于 where 子句，如下所示：

```
select City
  from WEATHER
 where LENGTH(City) < 7;

CITY
-----------
LIMA
PARIS
ATHENS
SYDNEY
SPARTA
```

它们也可以用于 order by 子句，如下所示：

```
select City
  from WEATHER
```

```
 order by LENGTH(City);

CITY
-----------
LIMA
PARIS
ATHENS
SYDNEY
SPARTA
CHICAGO
MANCHESTER
```

这些都是简单的示例，它们也可以用于更复杂的子句。例如，通过在 where 子句中使用 INSTR 函数，可以查找到名字中含有多个 O 的所有作者：

```
select Author from MAGAZINE
  where INSTR(Author,'O',1,2) > 0;

AUTHOR
------------------------
BONHOEFFER, DIETRICH
CROOKES, WILLIAM
```

这是通过查找在作者名字中的第二个 O 来完成的。“> 0”是一种逻辑方法，请回忆一下，函数通常产生两种不同的结果，一种是创建新对象，另一种是告诉您已有对象的相关信息。

INSTR 函数告诉您关于串的某些信息，尤其是所查找的字符集的位置。在本例中，要求确定 Author 串中第二个 O 的位置。对于至少含有两个 O 的名字，将得到大于 0 的值；而对于只含有一个 O 或不含 O 的名字，将为零值(当 INSTR 找不到任何匹配内容时，结果为零)。因此结果大于零的测试可以检查 INSTR 函数查找第二个 O 是否成功。

使用 INSTR 的 where 子句产生与以下语句相同的结果：

```
where Author LIKE '%O%O%'
```

记住，百分号(%)是一个通配符，这说明它可以代替任何字符。因此，这里的 like 子句告诉 SQL 查找两个 O，其前面、中间或后面可以是任何内容。这比前面的 INSTR 示例更容易理解。

在 Oracle 中经常可以用不同的方法产生同样的结果。有些方法容易理解，有些操作运行速度比较快，有些方法更适合于某种情况，而有些方法则只与个人的风格有关。

7.6.1 SOUNDEX

SOUNDEX 是只用于 where 子句的串函数。它具有查找在发音上类似于其他词语的单词的强大功能，而不论两个单词是怎样拼写的。这在不能确定某个字词或名字是如何拼写时特别有用。下面是 SOUNDEX 的格式：

```
SOUNDEX( string)
```

这里有几个 SOUNDEX 的应用实例：

```
SOUNDEX('menncestr');

CITY        TEMPERATURE CONDITION
----------- ----------- ---------
MANCHESTER           66 FOG

select Author from MAGAZINE
  where SOUNDEX(Author) = SOUNDEX('Banheffer');

AUTHOR
-------------------------
BONHOEFFER, DIETRICH
```

SOUNDEX 将所选列中的项的发音与单引号中的单词的发音进行比较，并查找近似匹配。SOUNDEX 对字母或字母组合在英语中的发音方式做某种假设，并且被比较的两个字词必须有相同的首字母。虽然 SOUNDEX 不一定总能找到要搜索的或拼错的字词，但还是有一定帮助的。

where 子句中的两个 SOUNDEX 函数不必一定要含有字面量，SOUNDEX 可以用来比较两个列中的数据来查找发音相似的数据。

此函数的一个有用的功能是整理邮件列表。许多列表都有在客户名字的拼写或格式上稍有不同的重复项。使用 SOUNDEX 列出所有发音相似的名字，就会发现许多重复项并将删除它们。

将 SOUNDEX 函数用于 ADDRESS 表：

```
select LastName, FirstName, Phone
  from ADDRESS;

LASTNAME                  FIRSTNAME                 PHONE
------------------------- ------------------------- ------------
BAILEY                    WILLIAM                   213-555-0223
ADAMS                     JACK                      415-555-7530
SEP                       FELICIA                   214-555-8383
DE MEDICI                 LEFTY                     312-555-1166
DEMIURGE                  FRANK                     707-555-8900
CASEY                     WILLIS                    312-555-1414
ZACK                      JACK                      415-555-6842
YARROW                    MARY                      415-555-2178
WERSCHKY                  ARNY                      415-555-7387
BRANT                     GLEN                      415-555-7512
EDGAR                     THEODORE                  415-555-6252
HARDIN                    HUGGY                     617-555-0125
HILD                      PHIL                      603-555-2242
LOEBEL                    FRANK                     202-555-1414
MOORE                     MARY                      718-555-1638
SZEP                      FELICIA                   214-555-8383
ZIMMERMAN                 FRED                      503-555-7491
```

要查找重复的名字，必须强制 Oracle 将该表中的每一个 LastName 与表中其他所有的 LastName 进行比较。

通过创建表的别名，第一个表称为 a，第二个表称为 b，将 ADDRESS 表和它本身连接起来。现在就好像有两个表，a 和 b，它们都有相同的列 LastName。

在 where 子句中，排除两个表中的 LastName 列值相同的行。这样可以防止出现与本身相匹配的 LastName。

然后，选择发音相似的行：

```
select a.LastName, a.FirstName, a.Phone
  from ADDRESS a, ADDRESS b
where SOUNDEX(a.LastName)=SOUNDEX(b.LastName);

LASTNAME                  FIRSTNAME                 PHONE
------------------------  ------------------------- ------------
SZEP                      FELICIA                   214-555-8383
SEP                       FELICIA                   214-555-8383
```

还可以用 SOUNDEX 函数在文本项中搜索某个单词。关于这方面的示例以及其他复杂的文本搜索，请参阅第 27 章。

7.6.2 国际语言支持

Oracle 不一定使用英语字符，它可以通过 National Language Support(国际语言支持)的实现以及 NCHAR 和 NVARCHAR2 数据类型使用任何一种语言来表示数据。通过使用比普通字符更长的信息构成的字符，Oracle 能表示日语和其他类似的串。请参阅附录 A 中的 NLSSORT、NLS_INITCAP、NLS_LOWER 和 NLS_UPPER。除了 SUBSTR 函数外，Oracle 还支持 SUBSTRB(用字节代替字符)、SUBSTRC(使用 Unicode 完整字符)、SUBSTR2(使用 UCS2 码点)和 SUBSTR4(使用 UCS4 码点)。

7.6.3 正则表达式支持

串函数 INSTR、REPLACE 和 SUBSTR 的功能已得到扩展，以支持正则表达式。第 8 章将介绍这些文本搜索的高级功能。

7.7 小结

数据可分为不同类型，主要有 DATE、NUMBER 和字符。字符数据主要为字母、数字或其他符号组成的串，这些串通常称作“字符串”或简称为“串”。这些串可以通过串函数来更改或描述。Oracle 有两种字符类型：变长串(VARCHAR2 数据类型)和定长串(CHAR 数据类型)。CHAR 列中的值的长度如果小于定义的列长度，则用空格填充至该列的长度。

有些函数，如 RPAD、LPAD、LTRIM、RTRIM、TRIM、LOWER、UPPER、INITCAP 以及 SUBSTR，可以在显示之前更改串的显示内容。

还有一些函数，如 LENGTH、INSTR 和 SOUNDEX，可以描述串的特征，如串的长度、某个字符的位置或它的发音。

所有这些函数都可以单独使用或组合使用，以选择和显示 Oracle 数据库中的信息。这是一个很简单的过程，即将简单的逻辑步骤组合起来完成非常复杂的任务。

第8章

正则表达式搜索

SUBSTR、INSTR、LIKE、REPLACE 和 COUNT 等函数的功能已得到加强和扩展，以支持正则表达式搜索。正则表达式支持许多标准化的控制和检查，例如，匹配指定的次数值、搜索标点符号或数字等。可以使用这些新函数在串中执行高级搜索。这些新函数分别命名为 REGEXP_SUBSTR、REGEXP_INSTR、REGEXP_LIKE、REGEXP_REPLACE 和 REGEXP_COUNT。

用户若以前使用过 UNIX 的 grep 命令在文本文件中搜索正则表达式，则可能对这里涉及的概念和搜索技术比较熟悉。

8.1 搜索串

从一个示例开始。ADDRESS 表中的电话号码的格式为 123-456-7890。要选出所有的交换机(即号码的中间部分)，可以在电话号码中用 select 选择任何以字符‘-’开始和结束的串。

在 REGEXP_SUBSTR 函数中，需要告诉 Oracle 串从哪里开始。本例中，搜索字符‘-’。用于搜索的正则表达式以下面的代码开始：

```
select REGEXP_SUBSTR('123-456-7890', '-')
```

现在需要告诉 Oracle 继续搜索一直到发现串中的另一个‘-’字符为止。为此，使用[^运算符，该方括号表达式说明，除了程序清单中显示的表达式外，可以接受匹配任何字符的值。现在的命令如下所示：

```
select REGEXP_SUBSTR('123-456-7890', '-[^-]+' )
         "REGEXP_SUBSTR"
  from DUAL;

REGE
----
-456
```

该命令告诉 Oracle 查找两个‘-’字符，中间可以有一个或多个非‘-’字符。请注意，如果在正则表达式的结束处添加一个‘-’字符，那么返回的串中包含‘-’：

```
select REGEXP_SUBSTR('123-456-7890', '-[^-]+-' )
         "REGEXP_SUBSTR"
  from DUAL;

REGEX
-----
-456-
```

如果没有经过训练和实践，则大部分用户(和开发人员)并不习惯输入如下的串：

```
'-[^-]+'
```

但是一旦使用了 RRGEXP_函数，很快就会发现其强大的功能。考虑一下，如果要像前面介绍的那样只使用 SUBSTR 和 INSTR 来得到同样的结果，并且假设两个‘-’字符间串的长度未知，那么需要执行下面的查询：

```
select SUBSTR('123-456-7890',
              INSTR('123-456-7890', '-',1,1),
              INSTR('123-456-7890', '-',1,2)-
              INSTR('123-456-7890', '-',1,1))
from DUAL;
```

通过比较可以看出，REGEXP_SUBSTR 函数简练很多。本章后面的示例将说明，使用正

则表达式搜索，可以在单个函数的调用中编码复杂的搜索模式。

表 8-1 说明了正则表达式运算符及其描述。理解这里所介绍的运算符对于有效地使用正则表达式的搜索功能非常关键。

从一个简单的搜索开始，应用这些运算符和字符类(character class)。首先，搜索一个冒号：

```
SQL> select REGEXP_SUBSTR
  2         ('MY LEDGER: Debits, Credits, and Invoices 1940',
  3          ':' ) "REGEXP_SUBSTR"
  4    from DUAL;

R
-
:
```

现在，用一个标点符号搜索来替代上面的搜索，使用[:punct:]字符类：

```
SQL> select REGEXP_SUBSTR
  2         ('MY LEDGER: Debits, Credits, and Invoices 1940',
  3          '[:punct:]' ) "REGEXP_SUBSTR"
  4    from DUAL;

R
-
:
```

表 8-1　正则表达式运算符

运　算　符	说　　明
\ ①	反斜线字符根据上下文有 4 种不同的含义。它可以表示本身、引用下一个字符、引入一个运算符或者什么都不做
*	匹配零次或多次出现
+	匹配一次或多次出现
?	匹配零次或一次出现
|	指定其他匹配项的运算符
^②	匹配行的开始字符
$②	匹配行的结束字符
.③	匹配除 NULL 以外的受支持的字符集中的任何字符
[ ]④	方括号表达式指定一个匹配列表，该列表匹配列表中显示的任何表达式。非匹配列表表达式以^开始，指定一个列表，除了列表中显示的表达式，该列表可以匹配任何字符
()	分组表达式，看作单个子表达式
{m}	匹配 m 次
{m,}	至少匹配 m 次
{m,n}	至少匹配 m 次，但是不超过 n 次

(续表)

运 算 符	说 明
\n[6]	反向引用(backreference)表达式(n 是 1~9 之间的数字)匹配圆括号和前面的\n 之间的第 n 个子表达式
[..][6]	指定一个对照(collation)元素，可以是多字符元素(例如，西班牙语中的[.ch.])
[: :][7]	指定字符类(例如，[:alpha:])。可以匹配字符类中的任何字符
[= =][8]	指定等价类。例如，[=a=]匹配所有包含基本字母“a”的字符

关于 POSIX 运算符和 Oracle 改进内容的注释：

① 如果反斜线运算符后跟的字符本身也是一个运算符，则该反斜线使其作为普通字符。例如，‘*’是一个星号字符串字面量。

② 字符‘^’和‘$’是 POSIX 锚定(anchoring)运算符。默认情况下，它们只匹配整个字符串的开始或结束。Oracle 允许指定‘^’和‘$’来匹配源串中任意位置的任意行的开始或结束。反过来说，也就是允许把源串看作多行。

③ 在 POSIX 标准中，定义该运算符(.)匹配除 NULL 和换行符以外的任何英文字符。在 Oracle 中，此运算符可以匹配数据库字符集中的任何字符，包括换行符。

④ 在 POSIX 标准中，一个正则表达式中的范围包括范围起点和终点之间的所有对照元素，该范围是当地语言学定义中的范围。因此，正则表达式中的范围是语言学上的范围，而不是字节值范围，并且范围表达式的语义与字符集无关。Oracle 根据 NLS_SORT 初始化参数决定的语言学定义，通过解释范围表达式，实现该无关性。

⑤ 反向引用表达式‘\n’匹配第‘n’个同样的字符串子表达式。字符 n 必须是 1~9 之间的数字，从左到右计算，指定第 n 个子表达式。如果源串中在‘\n’之前没有 n 个子表达式，那么该表达式无效。例如，正则表达式^(.*)\1$ 匹配一个由同一个字符串连续出现两次所组成的行。Oracle 在正则表达式模式和 REGEXP_REPLACE 函数中的替代串中支持反向引用表达式。

⑥ 对照元素是一个对照单元，多数情况下等于一个字符，但是在某些语言中也可以由两个或多个字符组成。正则表达式语法以前不支持含有多字符对照元素的范围，如范围‘a’到‘ch’。POSIX 标准引入对照元素分隔符‘[..]’，用于分隔多字符对照元素如‘[a-[.ch .]]’。Oracle 支持的对照元素由 NLS_SORT 初始化参数的设置来决定。对照元素只有在方括起来的表达式内才有效。

⑦ 在英语正则表达式中，范围表达式通常表示一个字符类。例如，‘[a-z]’表示任何小写字符。在多语言环境中，该约定并不实用，因为给定的字符类的首尾字符在所有的语言中可能并不相同。POSIX 标准引入方便的字符类语法‘[::]’。除了运算符以外，Oracle 也支持下面的字符类，它们基于 NLS 分类数据中的字符类定义：

字符类语法	含义
[:alnum:]	所有的字母和数字字符
[:alpha:]	所有的字母字符
[:blank:]	所有的空格字符
[:cntrl:]	所有的控制字符(不会打印出来)
[:digit:]	所有的数字
[:graph:]	所有的[:punct:]、[:upper:]、[:lower:]和[:digit:]字符
[:lower:]	所有的小写字母
[:print:]	所有可打印的字符
[:punct:]	所有的标点符号

[:space:]	所有的空隔字符(不会打印出来)
[:upper:]	所有的大写字母
[:xdigit:]	所有有效的十六进制字符

该字符类语法可以更好地利用 NLS 字符定义来灵活地编写正则表达式。这些字符类只有在方括号括起来的表达式中才有效。

⑧ Oracle 通过 POSIX 的‘[= =]’语法支持等价类。一个基本字母和它所有的重音字符构成一个等价类。例如，等价类‘[=a=]’匹配 ä 和 â。等价类只有在方括号括起来的表达式中才有效。

请注意关于等价类的一个限制：同一个等价类的组合版本和分解版本并不匹配。例如，‘ä’与‘a’后跟一个变音符号并不匹配。

从串中的指定的地方开始搜索，一直到遇到下一个逗号：

```
select REGEXP_SUBSTR
       ('MY LEDGER: Debits, Credits, and Invoices 1940',
        '[:punct:][^,]+,' ) "REGEXP_SUBSTR"
  from DUAL;

REGEXP_SU
---------
: Debits,
```

可以使用[:digit:]字符类在串中查找数字：

```
select REGEXP_SUBSTR
       ('MY LEDGER: Debits, Credits, and Invoices 1940',
        '[[:digit:]]+' ) "REGEXP_SUBSTR"
  from DUAL;

REGE
----
1940
```

如本例所示，可以使用字符类来实现多值搜索。在进行串搜索时(特别是在您以前没有使用过正则表达式的情况下)，尽可能从最简单的情况入手，然后逐步增加复杂度。

8.2　REGEXP_SUBSTR

正如前面的示例所示，REGEXP_SUBSTR 函数使用正则表达式来指定返回串的起点和终点。REGEXP_SUBSTR 的语法如下面的程序清单所示。REGEXP_SUBSTR 返回与 source_string 字符集中的 VARCHAR2 或 CLOB 数据相同的字符串。

```
REGEXP_SUBSTR( source_string, pattern
                  [, position
                     [, occurrence
                        [, match_parameter ]
```

```
        ]
    ]
)
```

本章中的示例一直在关注 pattern 变量，即正则表达式。正则表达式可以包含 512 个字节。从其语法结构可以看出，也可以指定与 position、occurrence 和 match_parameter 条件有关的参数。

position 变量告诉 REGEXP_SUBSTR 在 source_string 中从哪里开始搜索。默认的 position 值为 1(即第一个字符)。occurrence 变量是一个整数，指出 Oracle 应该在 source_string 中搜索 pattern 指定的哪一次出现。默认的 occurrence 值是 1。在标准的 SUBSTR 函数中没有 position 变量和 occurrence 变量。这些变量表示对标准的 SUBSTR 的功能的重要扩展。

可以使用 match_parameter 变量进一步定制搜索。match_parameter 是一个文本量，允许对函数的默认匹配行为进行更改。它可能的取值有：

- 'i' 用于不区分大小写的匹配
- 'c' 用于区分大小写的匹配
- 'n' 允许句点(.)作为通配符(请参阅表 8-1)去匹配换行符。如果省略该参数，则句点将不匹配换行符
- 'm' 将源串视为多行。即 Oracle 将"^"和"$"分别看作源串中任意位置任何行的开始和结束，而不是仅仅看作整个源串的开始或结束。如果省略该参数，则 Oracle 将源串看作一行

如果为 match_parameter 指定了多个相互矛盾的值，那么 Oracle 使用最后一个值。例如，如果指定'ic'，Oracle 将使用区分大小写的匹配。如果指定了上文中没有出现的字符，则 Oracle 将报错。

如果省略了 match_parameter，则将会进行如下处理：

- 默认区分大小写，这由 NLS_SORT 参数的值决定
- 句点(.)不匹配换行符
- 把源串视为一行

以下执行的 REGEXP_SUBSTR 搜索不区分大小写匹配：

```
select REGEXP_SUBSTR
        ('MY LEDGER: Debits, Credits, and Invoices 1940',
         'my' ,1,1, 'i') "REGEXP_SUBSTR"
  from DUAL;

RE
--
MY
```

现在，更改上例以执行区分大小写的搜索：

```
select REGEXP_SUBSTR
        ('MY LEDGER: Debits, Credits, and Invoices 1940',
         'my' ,1,1,'c') "REGEXP_SUBSTR"
```

```
  from DUAL;

RE
--
```

由于不匹配，因此没有返回任何内容。默认情况下，搜索是区分大小写的。

可以采用与 INSTR 一样的方式来使用 pattern 参数和 occurrence 参数。在下面的示例中，返回了第 2 个数字：

```
select REGEXP_SUBSTR
       ('MY LEDGER: Debits, Credits, and Invoices 1940',
        '[[:digit:]]' ,1,2) "REGEXP_SUBSTR"
  from DUAL;

R
-
9
```

使用 SUBSTR 和 INSTR 编写同样的查询(假设两个数字可能不是连续的)会复杂很多。

8.3 REGEXP_INSTR

REGEXP_INSTR 函数使用正则表达式返回搜索模式的起点和终点。REGEXP_INSTR 的语法如下所示。REGEXP_INSTR 返回一个整数，指出搜索模式的开始或结束的位置，如果没有发现匹配的值，则返回 0。

```
REGEXP_INSTR ( source_string, pattern
               [, position
                  [, occurrence
                     [, return_option
                           [, match_parameter ]
                        ]
                  ]
               ]
             )
```

如前文所示，REGEXP_INSTR 函数通常执行与 INSTR 相关联的一些功能。但 REGEXP_INSTR 添加了独特的功能，使它成为 SQL 工具包的重要补充。与 REGEXP_SUBSTR 一样，它也有变量 pattern、position(开始位置)、occurrence 和 match_parameter；请参阅 8.2 节对这些变量的介绍。新参数 return_option 允许用户告诉 Oracle，模式出现的时候，要返回什么内容。

- 如果 return_option 为 0 则，Oracle 返回第一个字符出现的位置。这是默认值，与 INSTR 的作用相同。
- 如果 return_option 为 1，则 Oracle 返回跟在所搜索字符出现以后下一个字符的位置。

例如，下面的查询返回了在串中发现的第一个数字的位置：

```
select REGEXP_INSTR
       ('MY LEDGER: Debits, Credits, and Invoices 1940',
        '[[:digit:]]') "REGEXP_INSTR"
  from DUAL;

REGEXP_INSTR
------------
          42
```

该串的第一个数字后面的下一个位置是什么？

```
select REGEXP_INSTR
       ('MY LEDGER: Debits, Credits, and Invoices 1940',
        '[[:digit:]]',1,1,1) "REGEXP_INSTR"
  from DUAL;

REGEXP_INSTR
------------
          43
```

即使对于那些并没有复杂模式的搜索，也可以使用 REGEXP_INSTR 来替代 INSTR，以简化在组合使用 INSTR 和 SUBSTR 功能的查询中所用到的数学和逻辑。仔细考虑区分大小写的设置，它通过 match_parameter 参数的设置来指定。

8.4 REGEXP_LIKE

除了前面列出的正则表达式函数，还可以使用 REGEXP_LIKE 函数。REGEXP_LIKE 支持在 where 子句中使用正则表达式。例如，哪些电话号码以 415 开头？

```
select LastName
 from ADDRESS
where REGEXP_LIKE (Phone,'415+');

LASTNAME
----------------------------
ADAMS
ZACK
YARROW
WERSCHKY
BRANT
EDGAR
```

REGEXP_LIKE 的格式如下所示：

```
REGEXP_LIKE( source_string, pattern
                    [ match_parameter ]
           )
```

在该模式中，可以使用本章前面介绍的所有搜索功能(包括字符类定义)作为 REGEXP_LIKE 搜索的一部分。该功能使得非常复杂的搜索变得简单。

例如，如何说明一个列值中含有数字？

```
select LastName
  from ADDRESS
 where REGEXP_LIKE (Phone, '[[:digit:]]');
```

如果由于在值中发现一个标点符号而导致 TO_NUMBER 函数失败，则可以显示导致它失败的行：

```
select LastName
  from ADDRESS
 where REGEXP_LIKE (Phone, '[[:punct:]]');
```

8.5　REPLACE 和 REGEXP_REPLACE

REPLACE 函数是用另外一个值来替代串中的某个值。例如，可以用一个匹配数字来替代字母的每一次出现。REPLACE 的格式如下所示：

```
REPLACE ( char, search_string [, replace_string])
```

如果没有指定 replace_string 变量的值，那么当发现 search_string 变量的值时，就将其删除。输入可以为任何字符数据类型——CHAR、VARCHAR2、NCHAR、NVARCHAR2、CLOB 或 NCLOB。下面是一个示例：

```
REPLACE('GEORGE', 'GE', 'EG') = EGOREG
REPLACE('GEORGE', 'GE', NULL) = OR
```

如果搜索串的长度不为零，则可以知道搜索串在某个串中出现的次数。首先，计算源串的长度：

```
LENGTH('GEORGE')
```

然后，计算源串删除搜索串以后的长度：

```
LENGTH(REPLACE('GEORGE', 'GE', NULL))
```

接着用搜索串的长度除以两次的长度之差，就可以得到搜索串出现的次数：

```
select LENGTH('GEORGE')
         - LENGTH(REPLACE('GEORGE', 'GE', NULL))
         /
         LENGTH('GE') AS Counter
from DUAL;
COUNTER
-------
2
```

REGEXP_REPLACE 函数在几个方面扩展了 REPLACE 函数的功能。它支持在搜索模式中使用正则表达式，也支持本章前面描述的变量，即 position、occurrence 和 match_parameter，从而可以选择只替代某些匹配的值，或者不区分大小写。REGEXP_REPLACE 函数的语法如下所示：

```
REGEXP_REPLACE( source_string, pattern
                [, replace_string
                   [, position
                      [, occurrence
                         [, match_parameter ]
                      ]
                   ]
                ]
               )
```

除了 replace_string，这里所有的变量都已经在本章前面章节作了介绍。replace_string 告诉 Oracle 用什么来替代 source_string 中与 pattern 匹配的部分。occurrence 变量是一个非负整数，它指定操作的次数：如果为 0，则所有的匹配项都被替代；如果指定一个正数，则 Oracle 替代第 n 次匹配。

考虑 ADDRESS 表中的 Phone 列。首先，寻找格式为###-###-#### 的号码。该格式分为 3 部分，分别是 3 个数字的集合、后面是另 3 个数字的一个集合，然后又是 4 个数字的一个集合，中间用‘-’符号隔开。通过在 REGEXP_SUBSTR 函数调用中查找那些数字集合，可以找到与该标准匹配的行：

```
select REGEXP_SUBSTR (Phone,
       '([[:digit:]]{3})-([[:digit:]]{3})-([[:digit:]]{4})'
       ) "REGEXP_SUBSTR"
  from ADDRESS;

REGEXP_SUBST
------------
213-555-0223
415-555-7530
214-555-8383
312-555-1166
707-555-8900
312-555-1414
415-555-6842
415-555-2178
415-555-7387
415-555-7512
415-555-6252
617-555-0125
603-555-2242
202-555-1414
718-555-1638
214-555-8383
```

```
503-555-7491
```

现在，使用 REGEXP_REPLACE 把前 3 个数字放在圆括号内，同时省略第一个‘-’符号。为此，我们将第 1 个数字集称为 \1，第 2 个数据集称为 \2，第 3 个数据集称为\3。

```
select REGEXP_REPLACE (Phone,
        '([[:digit:]]{3})-([[:digit:]]{3})-([[:digit:]]{4})'
      , '(\1) \2-\3'
      ) "REGEXP_REPLACE"
 from ADDRESS;

REGEXP_REPLACE
----------------------------------------
(213) 555-0223
(415) 555-7530
(214) 555-8383
(312) 555-1166
(707) 555-8900
(312) 555-1414
(415) 555-6842
(415) 555-2178
(415) 555-7387
(415) 555-7512
(415) 555-6252
(617) 555-0125
(603) 555-2242
(202) 555-1414
(718) 555-1638
(214) 555-8383
(503) 555-7491
```

输出说明了 REGEXP_REPLACE 函数调用的结果，即区号用圆括号括起来，第一个'-'被去掉。

为了说明 occurrenc 变量的工作原理，下面的 REGEXP_REPLACE 函数调用是用句点来替代电话号码中的第二个'5'：

```
select REGEXP_REPLACE (Phone,
        '5', '.',
       1, 2
       ) "REGEXP_REPLACE"
 from ADDRESS;

REGEXP_REPLACE
----------------------------------------
213-5.5-0223
415-.55-7530
214-5.5-8383
312-5.5-1166
707-5.5-8900
312-5.5-1414
```

```
415-.55-6842
415-.55-2178
415-.55-7387
415-.55-7512
415-.55-6252
617-5.5-0125
603-5.5-2242
202-5.5-1414
718-5.5-1638
214-5.5-8383
503-.55-7491
```

可以进一步修改该查询语句，排除前 3 个可能匹配的数字(开始位置设为 4)，并替代第 4 次出现：

```
select REGEXP_REPLACE (Phone,
       '5', '.',
       4, 4
       ) "REGEXP_REPLACE"
  from ADDRESS;

REGEXP_REPLACE
----------------------------------------
213-555-0223
415-555-7.30
214-555-8383
312-555-1166
707-555-8900
312-555-1414
415-555-6842
415-555-2178
415-555-7387
415-555-7.12
415-555-62.2
617-555-012.
603-555-2242
202-555-1414
718-555-1638
214-555-8383
503-555-7491
```

通过在 where 子句中使用 REGEXP_INSTR，可以限制返回的行。在本例中，只显示那些至少含有 4 个‘5’的行(从第 4 个字符开始)。因为该搜索模式并不复杂，所以这里也可以使用 INSTR 函数。

```
select REGEXP_REPLACE (Phone,
       '5', '.',
       4, 4
       ) "REGEXP_REPLACE"
  from ADDRESS
 where REGEXP_INSTR(Phone, '5',4,4) > 0;
```

```
REGEXP_REPLACE
----------------------------------------
415-555-7.30
415-555-7.12
415-555-62.2
```

- ‘n’ 允许句点(.)作为通配符去匹配换行符。如果省略该参数，则句点将不匹配换行符
- ‘m’ 将源串视为多行。即 Oracle 将^和$分别看作源串中任意位置任何行的开始和结束，而不是仅仅看作整个源串的开始或结束。如果省略该参数，则 Oracle 将源串看作一行。
- ‘x’ 忽略空格字符。默认情况下，空格字符与自身相匹配。

如果为 match_param 指定了多个相互矛盾的值，那么 Oracle 使用最后一个值。

可以用 REGEXP_COUNT 来修改本章前面的 LENGTH 示例。可以将下面的语法

```
select (LENGTH('GEORGE')
       - LENGTH(REPLACE('GEORGE', 'GE', NULL)) )
       /
       LENGTH('GE') AS Counter
from DUAL;

COUNTER
-------
       2
```

用如下语法来代替，得到的结果是相同的：

```
select REGEXP_COUNT('GEORGE','GE',1,'i')
from DUAL;
```

用 REGEXP_COUNT 取代 LENGTH 还有一个好处，即可以进行不区分大小写的搜索，因此，前面的查询也可以写成下面这样：

```
select REGEXP_COUNT('GEORGE','ge',1,'i')
    from DUAL;
```

REGEXP_SUBSTR 、 REGEXP_INSTR 、 REGEXP_LIKE 、 REGEXP_REPLACE 和 REGEXP_COUNT 函数的使用只受限于您开发正则表达式的能力，而正则表达式反映了您的需要。正如本章中的示例所示，可以使用这些函数来修改已有数据的显示、查找复杂的模式以及在模式中返回串。

第9章

数值处理

实际上，我们所做的任何事情(尤其在商业活动中)，经常是用数字进行度量、解释和指导的。虽然 Oracle 不能纠正数字对我们的困扰以及它们经常给我们的控制带来的错觉，但它有助于对数据库中的数据进行有效、彻底的分析。精确的数学分析通常能显示最初看不到的趋势和事实。

9.1 3 类数值函数

Oracle 函数处理 3 类数值，即单值、值组和值列表。和使用串函数一样(第 7 章和第 8 章

讨论过)，这些函数有些是用来更改数值的，而有些仅提供数值的相关信息。数值的类别用以下方法来区分：

单值(single value)只是一个数值，例如：

- 字面数值，如 544.3702
- SQL*Plus 或 PL/SQL 中的变量
- 数据库的一列和一行中的一个数字

Oracle 的单值函数通常通过计算来更改这些值。

值组(group of value)是一系列数据行中的某一列的所有数据，如第 4 章中的股票行情表的所有股票数据行的收盘价。Oracle 的值组函数提供的是整个组的信息(如股票的平均价格)，而不是该组中的单个成员的信息。

值列表(list of value)可以包括以下一系列数值：

- 字面值，如 1、7.3、22 和 86
- SQL*Plus 或 PL/SQL 中的变量
- 表中的列，如 OpeningPrice、ClosingPrice、Bid 和 Ask。

Oracle 的列表函数选择值列表中一个成员的值。

在附录 A 中，您可以看到按类列出的这些函数(数字函数、列表函数、聚集函数和分析函数)以及这些函数的描述和用法。

9.2 表示法

函数采用如下表示法：

```
FUNCTION(value [,option])
```

函数本身用大写，值和选项用小写。无论单词 value 什么时候出现，它都代表以下内容之一：一个字面数字、表中的一个数值列的名字、一个计算结果或一个变量。因为 Oracle 不允许将数字用作列名，所以字面数字不应该使用单引号(像一个文本串在串函数中那样)。列名也不能用单引号。

每个函数只有一对圆括号，函数所使用的值以及传递给此函数的附加信息都放在圆括号中。

虽然有些函数有 option 参数，但不是必需的，使用它们可以提供更多的控制。选项总是出现在方括号([])中，函数的必需部分总是出现在选项部分的前面。

9.3 单值函数

大多数的单值函数都是非常简单的。本节给出了主要函数的一些简要示例，并且显示了函数的结果以及它们都是如何与列、行和列表相对应的。在这些示例后面，您将明白如何组合使用这些函数。所有这些函数的语法都可以在附录 A 中找到。

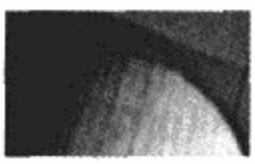
创建一个名为 MATH 的表来显示许多数学函数的计算结果。它只有 4 行 4 列，如下

所示：

```
select Name, Above, Below, Empty from MATH;

NAME             ABOVE   BELOW  EMPTY
------------ ------- ------- -----
WHOLE NUMBER        11      -22
LOW DECIMAL      33.33  -44.44
MID DECIMAL       55.5    -55.5
HIGH DECIMAL   66.666 -77.777
```

此表由于拥有不同特征的值而显得特别有用，这些特征由行的名字进行说明。WHOLE NUMBER 行不包含小数部分，LOWER DECIMAL 行有小于 0.5 的小数，MID DECIMAL 行有等于 0.5 的小数，而 HIGH DECIMAL 有大于 0.5 的小数。此范围在使用 ROUND 函数和 TRUNC(truncate)函数以及理解这些函数如何影响数值时尤其重要。

NAME 列的右边有其他 3 列：Above 列(它只包含大于 0 的数字，即正数)；Below 列(它只包含小于 0 的数字)；Empty 列(它的值是 NULL)。



注意：

在 Oracle 中，数值列可以没有任何值，此时它的值是 NULL 而不是 0，它是空值。不久您就会发现，这在计算中有相当重要的意义。

要说明大多数数学函数的工作原理，并不需要用到 MATH 表中的所有行，这些示例主要使用最后一行，即 HIGH DECIMAL。另外，SQL*Plus 的 column 命令已经显式地显示了计算的精度，使得影响数值精度的函数结果可以清晰地显示出来。为了回顾产生数值结果的 SQL 命令和 SQL*Plus 命令，可以查看针对示例表的 create table 命令的脚本。

9.3.1 加减乘除

以下的查询将用 Above 和 Below 说明 4 种基本的算术函数：

```
select Name, Above, Below, Empty,
       Above + Below AS Plus,
       Above - Below AS Subtr,
       Above * Below AS Times,
       Above / Below AS Divided
  from MATH
 where Name = 'HIGH DECIMAL';

NAME             ABOVE   BELOW  EMPTY     PLUS     SUBTR        TIMES    DIVIDED
------------ ------ ------- ----- ------- ------- ---------- ----------
HIGH DECIMAL 66.666 -77.777         -11.111  144.443  -5185.0815 -.85714286
```

9.3.2 NULL

在下面的示例中，再来进行一次四则运算，只不过上例使用的是 Above 和 Below，而本例使用的是 Above 和 Empty。请注意，任何包含 NULL 值的算术运算都会得到结果 NULL。

计算列(其值为计算的结果)的加、减、乘、除都为空。

```
SQL> select Name, Above, Below, Empty,
            Above + Empty AS Plus,
            Above - Empty AS Subtr,
            Above * Empty AS Times,
            Above / Empty AS Divided
       from MATH
      where Name = 'HIGH DECIMAL';

NAME             ABOVE    BELOW   EMPTY     PLUS     SUBTR    TIMES   DIVIDED
------------- ------ -------- ------ ------- -------- ------- ---------
HIGH DECIMAL   66.666 -77.777
```

这里所看到的代码证实了 NULL 值不能用于计算。NULL 与 0 不同，应当把 NULL 看作是一个未知值，例如，假如您有一个关于朋友的名字和年龄的表，但是 PAT SMITH 的 Age 列为空，因为您不知道他的年龄。他与您的年龄差是多少呢？显然不能用您的年龄减零。您的年龄减一个未知年龄仍是未知的，即 NULL。由于没有结果，因此您不能填写答案。由于不能进行计算，因此答案为 NULL。

这也就是为什么不能在 where 子句中将 NULL 与等号一起使用的原因(请参见第 5 章)。因为 x 和 y 都是未知的，所以 x 等于 y，这样的说法是无意义的。如果 Mrs.Wilkins 和 Mr.Adams 的年龄都未知，并不意味着它们年龄相同。

下面举一些示例说明 NULL 意味着一个值是无关的，如一幢房子的门牌号。在某些时候，门牌号为 NULL 是因为不清楚门牌号是多少(即使它确实存在)，而在另一些情况下门牌号为 NULL 是因为它不存在。9.4.1 节将详细介绍 NULL。

9.3.3 NVL：空值置换函数

9.3.2 节讲述了 NULL 的一般情况，即 NULL 表示一个未知的或无关的值。但是，在特定情况下，虽然某些值是未知的，但您可以做一个合理的假设。例如，如果您是一名包裹运送员，有 30%的客户不知道其货物的重量和体积，那么完全不能估计出将需要多少架运输机吗？当然不是。由于您根据经验会知道这些包裹的平均重量和体积，因此，可以为没有提供这些信息的客户填入这些数据。下面是您的客户所提供的信息：

```
SQL> select Client, Weight from SHIPPING;

CLIENT           WEIGHT
-------------    ------
JOHNSON TOOL         59
DAGG SOFTWARE        27
TULLY ANDOVER
```

以下是用 NULL 值置换函数(NVL)所做的工作：

```
SQL> select Client, NVL(Weight,43) from SHIPPING;
```

```
CLIENT           NVL(WEIGHT,43)
-------------    --------------
JOHNSON TOOL                 59
DAGG SOFTWARE                27
TULLY ANDOVER                43
```

由于现在您知道包裹的平均重量是 43 磅，因此在客户的包裹重量未知的情况下(即列值为空时)可使用 NVL 函数填入 43。在此情形下，虽然 TULLY ANDOVER 客户不知道其包裹的重量，但您仍能够计算出这些包裹的总重量并得到一个合理的估计值。

以下是 NVL 函数的格式：

```
NVL(value, substitute)
```

如果 value 为 NULL，则该函数等于 substitute。如果 value 不为 NULL，则该函数等于 value。注意 substitute 可以是一个数字、另一个列或一个计算结果。如果您真的是一位遇到此问题的包裹搬运工，则甚至可以在 select 语句中创建一个表连接，在该语句中 substitute 来自一个对所有非 NULL 包裹的重量求平均值的视图。

NVL 不只限于数值，虽然它也能用于 CHAR、VARCHAR2、DATE 和其他数据类型，但是 value 和 substitute 必须为相同的数据类型。另外，只有在数据是未知的而不是无关的情况下，NVL 才是有用的。

一个类似的函数是 NVL2，它相对复杂一些。它的格式如下所示：

```
NVL2( expr1 , expr2 , expr3 )
```

在 NVL2 中，expr1 永远不会被返回，返回值是 expr2 或 expr3。如果 expr1 不为 NULL，则 NVL2 返回 expr2。如果 expr1 为 NULL，则 NVL2 返回 expr3。参数 expr1 可以是任意数据类型。参数 expr2 和 expr3 可以是除 LONG 以外的任意数据类型。

对于 BINARY_FLOAT 和 BINARY_DOUBLE 数据类型可以使用 NANVL 函数。NANVL 接收两个变量，并且当第 1 个变量不是数字时返回第 2 个变量。

9.3.4 ABS：绝对值函数

绝对值函数用来度量某些量的幅度。例如，在温度变化和股票指数变化中，变化的幅度本身就很有意义，而不管变化是涨还是跌(当然它也重要)。绝对值总是为正数。

ABS 的格式如下所示：

```
ABS(value)
```

注意下列示例：

```
ABS(146) = 146
ABS(-30) = 30
```

9.3.5 CEIL

CEIL(表示最高限度)只产生大于或等于指定值的最小整数(或是整个数字)。要特别留意

它对负数产生的影响。

以下是 CEIL 的格式及一些示例：

```
CEIL(value)

CEIL(2)      =  2
CEIL(1.3)    =  2
CEIL(-2)     = -2
CEIL(-2.3)   = -2
```

9.3.6 FLOOR

FLOOR 返回等于或小于指定值的最大整数。以下是 FLOOR 的格式以及一些示例：

```
FLOOR(value)

FLOOR(2)      =  2
FLOOR(1.3)    =  1
FLOOR(-2)     = -2
FLOOR(-2.3)   = -3
```

9.3.7 MOD

MOD(modulus)函数主要用于复杂任务(如检查数字)的数据处理，它用来确保一串数字的精确传送。MOD 用一个除数除一个值并给出余数。例如，MOD(23，6)= 5 表示 23 除以 6，商是 3，余数是 5，因此 5 是该模运算的结果。

MOD 的格式如下：

```
MOD(value,divisor)
```

value 和 divisor 都可以是任意实数。假如 divisor 为零或负数，则 MOD 的值为零。请注意以下示例：

```
MOD(100,10)     =  0
MOD(22,23)      = 22
MOD(10,3)       =  1
MOD(-30.23,7)   = -2.23
MOD(4.1,.3)     =   .2
```

第 2 个示例表明，当除数大于被除数时，MOD 所得到的结果等于被除数。还要注意 value 不是整数的重要情况：

```
MOD( value, 1) = 0
```

上面这行代码是测试一个数是否为整数的好办法。

您也可以用类似的方法使用 REMAINDER 函数。虽然 MOD 函数与 REMAINDER 函数相似，但 MOD 在公式中使用的是 FLOOR，而 REMAINDER 使用的是 ROUND。

```
REMAINDER(4,3) = 1
```

9.3.8 POWER

POWER 只用于计算一个值与给定正指数的乘方，如下所示：

```
POWER( value, exponent)

POWER(3,2)          =     9
POWER(3,3)          =    27
POWER(-77.777,2)    = 6049.26173
POWER(3,1.086)      =     3.29726371
POWER(64,.5)        =     8
```

exponent 可以为任何实数。

9.3.9 SQRT：求平方根

Oracle 有单独的平方根函数，它与 POWER(value，.5)的结果相等：

```
SQRT( value)

SQRT(64)        = 8
SQRT(66.666)    = 8.16492498
SQRT(4)         = 2
```

负数的平方根是一个虚数。因为 Oracle 不支持虚数，所以当试图为一个负数开方时，它返回一个错误。

9.3.10 EXP、LN 和 LOG

EXP、LN 和 LOG 函数很少用于商业计算，但在科技工作中却普遍使用。EXP 是 e(2.71828183……)的幂运算；LN 是自然对数，或以 e 为底的对数。前两个函数互为反函数，即 LN(EXP(*i*)) = value。LOG 函数接收一个底数和一个正值。LN(value)与 LOG(2.71828183，value)相同。

```
EXP(value)

EXP(3)        =   20.0855369
EXP(5)        = 148.413159

LN(value)

LN(3)             =  1.09861229
LN(20.0855369)    =  3

LOG(value)

LOG(EXP(1),3)     = 1.09861229
LOG(10,100)       = 2
```

9.3.11 ROUND 和 TRUNC

ROUND 和 TRUNC 是两个相关的单值函数。TRUNC 按精度值截取某个数字。ROUND 则根据给定的精度舍入数值。

以下是 ROUND 和 TRUNC 函数的格式：

```
ROUND( value, precision)
TRUNC( value, precision)
```

在这里，有些属性值得注意。首先，看一个简单的用于 MATH 表 select 的示例。要求使用两位精度(保留小数点后两位数字)。

```
select Name, Above, Below,
       ROUND(Above,2),
       ROUND(Below,2),
       TRUNC(Above,2),
       TRUNC(Below,2)
from MATH;
```

```
                                        ROUND       ROUND      TRUNC       TRUNC
NAME            ABOVE    BELOW  (ABOVE,2) (BELOW,2) (ABOVE,2)    (BELOW,2)
------------ -------  ------- -------- -------- ---------    ---------
WHOLE NUMBER       11      -22         11       -22          11          -22
LOW DECIMAL     33.33   -44.44      33.33    -44.44       33.33       -44.44
MID DECIMAL      55.5    -55.5       55.5     -55.5        55.5        -55.5
HIGH DECIMAL   66.666  -77.777      66.67    -77.78       66.66       -77.77
```

只有最后一行受到了影响，因为只有它在小数点后面有 3 位数字。最后一行的正数和负数都被四舍五入或截断了，66.666 被四舍五入成更大的数字，即 66.67，而 - 77.777 被四舍五入为较小的数字，即 - 77.78。当按 0 位四舍五入时，结果如下：

```
select Name, Above, Below,
       ROUND(Above,0),
       ROUND(Below,0),
       TRUNC(Above,0),
       TRUNC(Below,0)
  from MATH;
```

```
                                        ROUND       ROUND      TRUNC       TRUNC
NAME            ABOVE    BELOW  (ABOVE,0) (BELOW,0) (ABOVE,0)   (BELOW,0)
------------ -------  ------- -------- -------- ---------    ---------
WHOLE NUMBER       11      -22         11       -22          11          -22
LOW DECIMAL     33.33   -44.44         33       -44           3          -44
MID DECIMAL      55.5    -55.5         56       -56          55          -55
HIGH DECIMAL   66.666  -77.777         67       -78          66          -77
```

请注意，当 55.5 变成 56 时，0.5 的小数部分被舍去。这遵循的是最通用的美国的舍入约定(有些舍入约定只在数字大于 0.5 时才向上舍入)。将这些结果与 CEIL 和 FLOOR 的结果进

行比较，它们有明显的区别：

```
ROUND(55.5) = 56   ROUND(-55.5) = -56
TRUNC(55.5) = 55   TRUNC(-55.5) = -55
CEIL(55.5)  = 56   CEIL(-55.5)  = -55
FLOOR(55.5) = 55   FLOOR(-55.5) = -56
```

最后，请注意，ROUND 和 TRUNC 函数也可用于负的精度的情况，它表示移到小数点的左边：

```
select Name, Above, Below,
       ROUND(Above,-1),
       ROUND(Below,-1),
       TRUNC(Above,-1),
       TRUNC(Below,-1)
  from MATH;
```

NAME	ABOVE	BELOW	ROUND (ABOVE,-1)	ROUND (BELOW,-1)	TRUNC (ABOVE,-1)	TRUNC (BELOW,-1)
WHOLE NUMBER	11	-22	10	-20	10	-20
LOW DECIMAL	33.33	-44.44	30	-40	30	-40
MID DECIMAL	55.5	-55.5	60	-60	50	-50
HIGH DECIMAL	66.666	-77.777	70	-80	60	-70

带有负数的舍入在生成像经济报表这样的数据时很有用，因为人口或美元的总数需要舍入到百万(10^6)、十亿(10^9)或万亿(10^{12})。

9.3.12 SIGN

SIGN 函数与绝对值函数反映的是数值的不同方面。ABS 函数给出的是值的大小而不是其符号，而 SIGN 函数则给出值的符号而不是大小。

以下是 SIGN 的格式：

```
SIGN(value)
```

```
示例: SIGN(146) =  1        比较:ABS(146) = 146
      SIGN(-30) = -1              ABS(-30) = 30
```

0 的 SIGN 为 0。

```
SIGN(0)=0
```

SIGN 函数经常与 DECODE 函数联合使用。DECODE 函数将在第 16 章中介绍。

9.3.13 SIN、SINH、COS、COSH、TAN、TANH、ACOS、ATAN、ATAN2 和 ASIN

三角函数正弦、余弦和正切都是科学和技术函数，很少用于商业方面。SIN、COS 和 TAN 给出用弧度(角度*π/180)表示的角度的标准三角函数值。SINH、COSH 及 TANH 给出一个角

度的双曲函数。

```
SIN( value)

SIN(30*3.141592655/180) = .5

COSH( value)

COSH(0) = 1
```

ASIN、ACOS 和 ATAN 函数返回给定值的反正弦值、反余弦值和反正切值(用弧度表示)。ATAN2 也返回反正切值，但它具有两个输入参数，这两个输入值没有大小限制，输出用弧度表示。

9.4 聚集函数

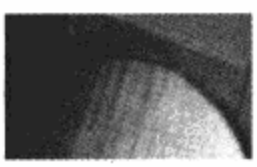
聚集函数或者组值函数是用于统计的函数，如 SUM、AVG、COUNT 以及把一组值作为整体来提供相关信息的函数。如前面所提到的表中的所有朋友的平均年龄，或该组中年龄最大的成员、年龄最小的成员、组中的成员数等。即使其中一个函数提供关于单行的信息(如年龄最大的人)，它也仍然是与组有关的行定义的信息。

您可以对数据使用大量的高级统计函数——包括衰减测试和取样。在下面的讨论中，您将看到关于最常用的组值函数的描述；其他组值函数则请参见附录 A。

9.4.1 组值函数中的 NULL

组值函数对待 NULL 值的方式与单值函数不同。组值函数忽略 NULL 值并计算出结果。

以 AVG 函数为例。假如您有一个 100 个朋友及他们年龄的列表。如果从中随机挑出 20 个人，并计算其年龄的平均值，那么这与您随机选择另 20 个朋友所计算的年龄平均值有什么不同呢？与取 100 人的平均年龄有什么不同呢？实际上，这 3 组年龄平均数非常接近。这意味着 AVG 函数对所缺少的记录不敏感，即使缺少的数据占记录总数的百分比很高时也是如此。



注意：

AVG 并不能完全不受缺少数据的影响，虽然在某些情况下，丢失的数据对它影响很大(如缺少的数据不是随机分布的)，但这种情况不常见。

AVG 对缺少数据的不敏感性是相对的，如与 SUM 比较。20 位朋友的年龄总和与 100 位朋友的年龄总和的接近程度如何？完全不接近。所以，如果您有一张关于朋友的列表，但 100 人中仅 20 人的年龄列不为空，其他 80 人的年龄列都是 NULL，那么对于整组数据而言，哪一种运算更可靠；对于缺少数据的情况，哪一种运算受影响较小——是这 20 个朋友的 AVG 运算还是它们的 SUM 运算。注意，这与基于 20 个朋友的平均年龄值来估计 100 个朋友的总年龄有着天壤之别(实际上是 20 个年龄的平均值乘以 100)。如果您不知道有多少行是 NULL，

则可以用如下语句得出一个相对合理的结果：

```
Select  AVG(Age) from  BIRTHDAY;
```

但是，用下面的程序不能得出合理的结果：

```
select  SUM(Age) from  BIRTHDAY;
```

结果是否合理的测试可以确定其他组函数对 NULL 的反应。STDDEV 和 VARIANCE 是中间趋势的度量，它们也对缺少的数据较不敏感(这些将在 9.4.5 节中介绍)。

MAX 和 MIN 测量数据的两个极限，它们可以大幅度地波动，而 AVG 却保持相对稳定。如果您向 99 个平均年龄为 50 岁的人群中添加一个 100 岁的老人，则平均年龄仅上升为 50.5，但年龄的最大值已经是原来的 2 倍。增加一个新生儿，平均年龄值回到 50，但最小值现在为 0。很明显，由于缺少的或未知的 NULL 值可能对 MAX、MIN 以及 SUM 产生很大影响，因此在使用它们时要小心，特别是绝大部分的数据是 NULL 时更要小心。

有没有能够提供数据的稀疏程度以及 NULL 值的个数与实际值个数的比例，并且能对 MAX、MIN 和 SUM 做出更好的估计的函数？有，但这样的函数是统计预测函数，这些函数必须对特定的数据组作出明确的假设。这不是常规组函数的任务。有些统计学家认为如果这些函数遇到 NULL 值，就应该返回 NULL 值，因为返回任何值都将引起误解。Oracle 返回非空值，但由您决定此结果是否合理。

COUNT 是一个特例。不论用什么方法处理 NULL 值，它总是返回一个数值；它从不对 NULL 求值。COUNT 的格式和用法将在下文给出，但是与其他组函数不同，它将统计某列的所有非空行，或对所有行进行统计。换句话说，如果要求计算 100 个朋友的年龄，则 COUNT 将返回 20 个朋友的年龄(因为 100 人中只有 20 人给出了年龄值)。如果要求对朋友表中的行进行统计而未指定是哪列，则它将返回 100。关于这些区别的示例将在 9.4.6 节中介绍。

9.4.2 单值函数和组值函数的示例

虽然组值函数和单值函数都不难理解，但是，对每个函数如何工作有一个大致的了解将有助于使用某些选项和明白它们的作用。

这些示例中的 COMFORT 表包括了某些城市每年 4 个抽样日的中午和午夜的基本温度数据，这 4 个抽样日为春分和秋分(大约 3 月 21 日和 9 月 23 日)以及夏至和冬至(7 月 22 日和 12 月 22 日)。基于每年这 4 天的温度，您应该知道这些城市的特征。

在本例中，此表只有 8 行，即 San Francisco、California、Keene 和 New Hampshire 在 2003 年这 4 个日期的数据。可以用 Oracle 的数值函数分析这些城市在 2003 年的平均温度、温度的变化，等等。随着年份和城市数据的增加，可以完成整个世纪的温度模式和变化分析。

该表的结构如下所示：

```
describe COMFORT

Name                                      Null?    Type
----------------------------------------- -------- ------------
```

```
CITY                                      NOT NULL    VARCHAR2(13)
SAMPLEDATE                                NOT NULL    DATE
NOON                                                  NUMBER(3,1)
MIDNIGHT                                              NUMBER(3,1)
PRECIPITATION                                         NUMBER
```

它包含以下温度数据：

```
SQL> select City, SampleDate, Noon, Midnight from COMFORT;

CITY            SAMPLEDAT  NOON   MIDNIGHT
--------------  ---------  ----   --------
SAN FRANCISCO   21-MAR-03  62.5        42.3
SAN FRANCISCO   22-JUN-03  51.1        71.9
SAN FRANCISCO   23-SEP-03               61.5
SAN FRANCISCO   22-DEC-03  52.6        39.8
KEENE           21-MAR-03  39.9        -1.2
KEENE           22-JUN-03  85.1        66.7
KEENE           23-SEP-03  99.8        82.6
KEENE           22-DEC-03  -7.2        -1.2
```

9.4.3 AVG、COUNT、MAX、MIM 和 SUM

由于电源故障，因此没有记录 San Francisco 在 9 月 23 日中午的温度。结果可通过以下查询查看：

```
SQL> select AVG(Noon), COUNT(Noon), MAX(Noon), MIN(Noon), SUM(Noon)
  2    from COMFORT
  3   where City = 'SAN FRANCISCO';

AVG(NOON) COUNT(NOON) MAX(NOON) MIN(NOON)  SUM(NOON)
--------- ----------- --------- ---------  ---------
     55.4           3      62.5       51.1       166.2
```

AVG(Noon)是 3 个已知温度的平均值，COUNT(Noon)是 Noon 列不为空的行数。MAX 和 MIN 的意思是显而易见的。SUM(Noon)只是 3 个日期的温度和，因为 9 月 23 日为空。请注意：

```
SQL> SUM(NOON)
  2  ------
  3     166.2
```

绝非偶然与 AVG(Noon)的 3 倍不一致。

9.4.4 组值函数和单值函数的组合

假如想要知道某一天的温度是如何变化的，这就是变化量的测量。首先用中午的温度减去午夜的温度来回答这个问题：

```
select City, SampleDate, Noon-Midnight
  from COMFORT
 where City = 'KEENE';

CITY            SAMPLEDAT   NOON-MIDNIGHT
------------    ---------   -------------
KEENE           21-MAR-03           41.1
KEENE           22-JUN-03           18.4
KEENE           23-SEP-03           17.2
KEENE           22-DEC-03             -6
```

由于只考虑此表的4行数据，因此可以忽略表中的负号。温度变化量实际上是一个幅度，即表示温度变化了多少。由于不应当包含符号，因此-6是不正确的。于是将它包含在进一步的计算(如对一年中平均气温变化的计算)所得出的结果是绝对错误的，如下所示：

```
select AVG(Noon-Midnight)
  from COMFORT
 where City = 'KEENE';

AVG(NOON-MIDNIGHT)
------------------
            17.675
```

正确的答案应该是一个绝对值，如下所示：

```
select AVG(ABS(Noon-Midnight))
  from COMFORT
 where City = 'KEENE';

AVG(ABS(NOON-MIDNIGHT))
-----------------------
                 20.675
```

组合函数的方式与第7章中串函数的组合方式相同。一个完整的函数，如

```
ABS(Noon-Midnight)
```

将只作为另一个函数的值，如

```
AVG(value)
```

将产生

```
AVG(ABS(Noon-Midnight))
```

上例展示了单值函数和组值函数的结合用法。可以把单值函数放在组值函数中。单值函数将计算每一行的结果，而组值函数将把此结果看作是此行的实际值。单值函数可以毫无限制地进行组合(可相互嵌套)。组值函数可用单值函数代替其值。实际上，组值函数可以用多个单值函数代替其值。

组合组值函数结果会怎样？首先，用以下方法嵌套它们没有任何意义：

```
select SUM (AVG(Noon))from COMFORT;
```

接着将产生以下错误：

```
ERROR at line 1:
ORA-00978: nested group function without GROUP BY
```

另外，如果它实际运行，则将产生和以下语句相同的结果：

```
AVG(Noon)
```

因为 AVG(Noon)的结果是一个单值。因为单值的 SUM 也是单值，所以嵌套的组函数是没有意义的。例外是在 select 语句中使用 group by 时，缺少 group by 时 Oracle 将产生错误信息。这方面的内容将在第 12 章中介绍。

将两个或更多的组函数进行加、减、乘、除运算是有意义的。例如：

```
select MAX(Noon) - MIN(Noon)
  from COMFORT
 where City = 'SAN FRANCISCO';

MAX(NOON)-MIN(NOON)
-------------------
               11.4
```

给出了一年中的温度变化范围。实际上，只要再多一些代码就可得到 San Francisco 和 Keene 的快速比较。

```
select City, AVG(Noon), MAX(Noon), MIN(Noon),
       MAX(Noon) - MIN(Noon) AS Swing
  from COMFORT
 group by City;

CITY            AVG(NOON)   MAX(NOON)   MIN(NOON) SWING
------------    ---------   ---------   --------- -----
KEENE                54.4        99.8        -7.2   107
SAN FRANCISCO        55.4        62.5        51.1  11.4
```

此查询是在数据中发现有用信息的好示例。虽然我们发现两个城市的平均温度几乎相等，但是与 San Francisco 比较，Keene 的温度变化大，这说明了两个城市的年温度变化量，以及城市对穿着(或家里的加热和降温)的不同需求。Group by 子句将在第 12 章详细介绍。简而言之，在本例中，它并不是强制组函数作用于整个表，而是作用于城市的温度组。

9.4.5 STDDEV 和 VARIANCE

标准差和标准方差函数用于常规的统计学功能，其格式与所有组函数的格式相同：

```
select MAX(Noon), AVG(Noon), MIN(Noon), STDDEV(Noon),
       VARIANCE(Noon)
  from COMFORT
 where City = 'KEENE';

MAX(NOON) AVG(NOON) MIN(NOON) STDDEV(NOON)  VARIANCE(NOON)
--------- --------- --------- ------------ --------------
     99.8      54.4      -7.2  48.3337701      2336.15333
```

详细内容请参考附录 A 中关于统计函数的语法。

9.4.6 组函数中的 DISTINCT

所有组值函数都有一个 DISTINCT 与 ALL 选项对应。以 COUNT 为例可以很好地说明这些选项是如何工作的。

以下是 COUNT 的格式(注意：|表示“或”)：

```
COUNT( (DISTINCT | ALL) value)
```

下面是一个示例：

```
select COUNT(DISTINCT City), COUNT(City), COUNT(*)
  from COMFORT;

COUNT(DISTINCTCITY) COUNT(CITY) COUNT(*)
------------------- ----------- --------
                  2           8        8
```

该查询显示了一组有趣的结果。首先，DISTINCT 强制 COUNT 只对每个城市名计数一次。如果要对午夜温度以 DISTINCT 方式进行计数，则返回 7，因为 8 个温度中有两个是相同的。当 COUNT 用在 City 列且不使用 DISTINCT 时，结果为 8。

这也表明 COUNT 可以作用于字符列。它不像 SUM 和 AVG 那样对列的值进行计算，仅仅对指定的列中有值的数据行进行计数。

COUNT 有另一个独特的属性，即 value 可以为星号。COUNT 将告诉您该表中有多少行，而不管指定的任何列是否为 NULL，即使一行的所有字段为 NULL，也将对这一行计数。

其他的组函数既不能共享COUNT使用星号的功能，也没有把字符列作为value的功能(虽然 MAX 和 MIN 可以)。它们都可使用 DISTINCT 参数，这使得它们对每个唯一的值只作用一次。具有值的一个表，例如：

```
select FirstName, Age from BIRTHDAY;

FIRSTNAME          AGE
---------------    ----
GEORGE               42
ROBERT               52
NANCY                42
VICTORIA             42
```

```
FRANK                42
```

会产生这样的结果：

```
select  AVG(DISTINCT Age) AS Average,
        SUM(DISTINCT Age) AS Total
  from  BIRTHDAY;

AVERAGE   TOTAL
------- ------
     47      94
```

如果想知道朋友的平均年龄，则上面的结果不是正确的答案。DISTINCT 除了在 COUNT 函数中使用外，在其他地方很少使用，也许可用在某些统计计算中。无论是否有 DISTINCT，MAX 和 MIN 都产生同样的结果。

DISTINCT 对应的选项是 ALL，这是默认选项。ALL 告诉 SQL 检查每一行，即使这行的值与另一行的值相同。您不需要输入 ALL，如果不输入 DISTINCT，就自动使用 ALL。

9.5 列表函数

与组值函数(作用于一组行上)不同，列表函数作用于一行的一组列上，这些列可以是实际值或计算值。换句话说，列表函数将各列的值进行比较，从而选出列表中的最大值和最小值。以 COMFORT 表为例，如下所示：

```
select City, SampleDate, Noon, Midnight from COMFORT;

CITY            SAMPLEDAT NOON   MIDNIGHT
--------------  --------- ----   --------
SAN FRANCISCO   21-MAR-03 62.5        42.3
SAN FRANCISCO   22-JUN-03 51.1        71.9
SAN FRANCISCO   23-SEP-03              61.5
SAN FRANCISCO   22-DEC-03 52.6        39.8
KEENE           21-MAR-03 39.9        -1.2
KEENE           22-JUN-03 85.1        66.7
KEENE           23-SEP-03 99.8        82.6
KEENE           22-DEC-03 -7.2        -1.2
```

现在将此查询结果与下面的结果进行比较，特别注意 San Francisco 的 6 月和 9 月以及 Keene 的 12 月的数据：

```
select City, SampleDate, GREATEST(Midnight,Noon) AS High,
        LEAST(Midnight,Noon) AS Low
  from COMFORT;

CITY            SAMPLEDAT  HIGH    LOW
--------------  ---------  ----  ----
SAN FRANCISCO   21-MAR-03  62.5  42.3
```

```
SAN FRANCISCO   22-JUN-03  71.9  51.1
SAN FRANCISCO   23-SEP-03
SAN FRANCISCO   22-DEC-03  52.6  39.8
KEENE           21-MAR-03  39.9  -1.2
KEENE           22-JUN-03  85.1  66.7
KEENE           23-SEP-03  99.8  82.6
KEENE           22-DEC-03  -1.2  -7.2
```

San Francisco 的 9 月份数据为 NULL，原因在于 GREATEST 和 LEAST 不能将一个实际的午夜温度与一个未知的中午温度进行比较。在另两个实例中，午夜的实际温度高于中午的温度。

下面是 GREATEST 和 LEAST 的格式：

```
GREATEST( value1,value2,value3…)
LEAST( value1, value2,value3…)
```

GREATEST 和 LEAST 都可以使用多个值，并且这些值可以是列、字面数值、计算值或者其他列的组合。GREATEST 和 LEAST 也可用于字符列。例如，它们可以按字母顺序选出最后一个名字(GREATEST)或第一个名字(LEAST)。

```
Isaiah
LEAST('Bob','George','Andrew','Isaiah') = Andrew
```

可以使用 COALESCE 函数计算多个值的非空值。给定一串值，COALESCE 将返回遇到的第一个非空值。如果所有值都是空，则返回 NULL。

在 COMFORT 表中，San Francisco 的一个 Noon 值为 NULL。如以下查询：

```
select COALESCE(Noon, Midnight) from COMFORT
 where City = 'SAN FRANCISCO';

COALESCE(NOON,MIDNIGHT)
-----------------------
                  62.5
                  51.1
                  61.5
                  52.6
```

在输出的前两个记录中，显示的是 Noon 值。在第 3 个记录中，由于 Noon 值为 NULL，因此代之以 Midnight 的值返回。Oracle 的 DECODE 和 CASE 函数提供了相似的功能，详细内容将在第 16 章介绍。

9.6 使用 MAX 或 MIN 函数查找行

哪一个城市有最高的温度记录？在哪一天？虽然只看看上述 8 行，答案是很容易得到的，但是如果您有该国家每个城市在过去 50 年中每天的数据，那么多的数据该怎么查看呢？现在假设本年的最高温度出现在更接近中午而不是午夜的时候。以下语句将不起作用：

```
select City, SampleDate, MAX(Noon)
  from COMFORT;
```

Oracle 标出 City 列并给出错误消息：

```
select City, SampleDate, MAX(Noon)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

该错误消息有点晦涩难懂。它的意思是 Oracle 已经检测到该问题的逻辑错误。查询列操作表示您要显示不同行，而 MAX 这个组函数意味着需要所有行的一个组结果。这是两个不同的请求。因为第一个请求一组行，第二个只请求一个已计算的行，所以产生了冲突。下面说明如何构造查询：

```
select City, SampleDate, Noon
  from COMFORT
 where Noon = (select MAX(Noon) from COMFORT);

CITY            SAMPLEDAT  NOON
------------  ---------- -----
KEENE           23-SEP-03  99.8
```

这只产生了一行。因此，您也许会认为 City 列和 SampleDate 列以及查询中午温度最大值的请求组合与刚才提到的不矛盾。但是如果您需要最小温度值呢？

```
select City, SampleDate, Midnight
  from COMFORT
 where Midnight = (select MIN(Midnight) from COMFORT);

CITY            SAMPLEDAT MIDNIGHT
------------- --------- --------
KEENE           21-MAR-03     -1.2
KEENE           22-DEC-03     -1.2
```

两行！因为有多个满足 MIN 请求的值，所以，在把一个常规的列请求与组函数进行组合时会发生冲突。

也可以使用两个子查询，每个子查询使用一个组值函数(或者两个子查询，其中一个子查询有组函数，另一个没有组函数)。假定您想知道每年中午的最高和最低温度，则可以使用如下查询：

```
select City, SampleDate, Noon
  from COMFORT
 where Noon = (select MAX(Noon) from COMFORT)
    or Noon = (select MIN(Noon) from COMFORT);

CITY            SAMPLEDAT NOON
------------- --------- -----
KEENE           23-SEP-03 99.8
```

```
KEENE          22-DEC-03 -7.2
```

9.7 优先级和圆括号的应用

当在单值计算中使用多个算术运算符或逻辑运算符时，先执行哪个运算呢？这与它们所在的顺序有关系吗？看一下下面关于 DUAL 表的查询(即 Oracle 提供的一行一列的表)：

```
select 2/2/4 from DUAL;

    2/2/4
    -----
    .25
```

当引入了圆括号时，虽然数字和运算符(除号)都一样，但答案彻底改变了：

```
select 2/(2/4) from DUAL;

    2/(2/4)
    -------
          4
```

其原因是优先级(precedence)发生了变化。优先级定义了数学运算的执行顺序，通常不仅在 Oracle 中，在数学运算中也如此。规则很简单：圆括号优先级最高，乘法和除法次之，然后为加法和减法。当计算一个等式时，括号内的任何计算最先进行，接着计算乘法和除法，最后完成加法和减法运算。当执行优先级相等的运算时，总是从左到右地执行。下面是几个示例：

```
2*4/2*3 = 12                    (等同于((2*4)/2 )*3)
(2*4)/(2*3) = 1.333
4-2*5 = -6                      (等同于 4 - (2*5))
(4-2)*5 = 10
```

AND 和 OR 也遵守优先级规则，AND 的优先级较高。在下面两个查询中，观察 AND 的效果，以及从左到右的顺序：

```
select * from NEWSPAPER
 where Section = 'B' AND Page = 1 OR Page = 2;

FEATURE           S    PAGE
---------------   -   -----
Weather           C       2
Modern Life       B       1
Bridge            B       2

3 rows selected.

select * from NEWSPAPER
 where Page = 1 OR Page = 2 AND Section = 'B';
```

```
FEATURE           S    PAGE
---------------   -    -----
National News     A         1
Sports            D         1
Business          E         1
Modern Life       B         1
Bridge            B         2

5 rows selected.
```

如果真的要查找 Section B 的第 1 页或第 2 页，就必须用圆括号来改变 AND 的优先级。圆括号优先于任何其他运算。

```
select * from NEWSPAPER
 where Section = 'B' AND (Page = 1 OR Page = 2);

FEATURE           S    PAGE
---------------   - -----
Modern Life       B      1
Bridge            B      2

2 rows selected.
```

事实上，即使经验丰富的程序员和数学家，在编写查询或等式时也会忘记先执行什么。聪明的做法是使 Oracle 所遵从的顺序一目了然。无论在什么时候，在可能会产生误解时，请使用圆括号。

9.8 小结

单值函数以逐行方式对值进行操作。列表函数比较每一列同时只选择一个值，它也是以逐行的方式进行操作的。单值函数几乎总是更改列的值。当然，这并不意味着它们修改了数据库的值，而是对这个值进行了计算，并且结果与原来的值不同。

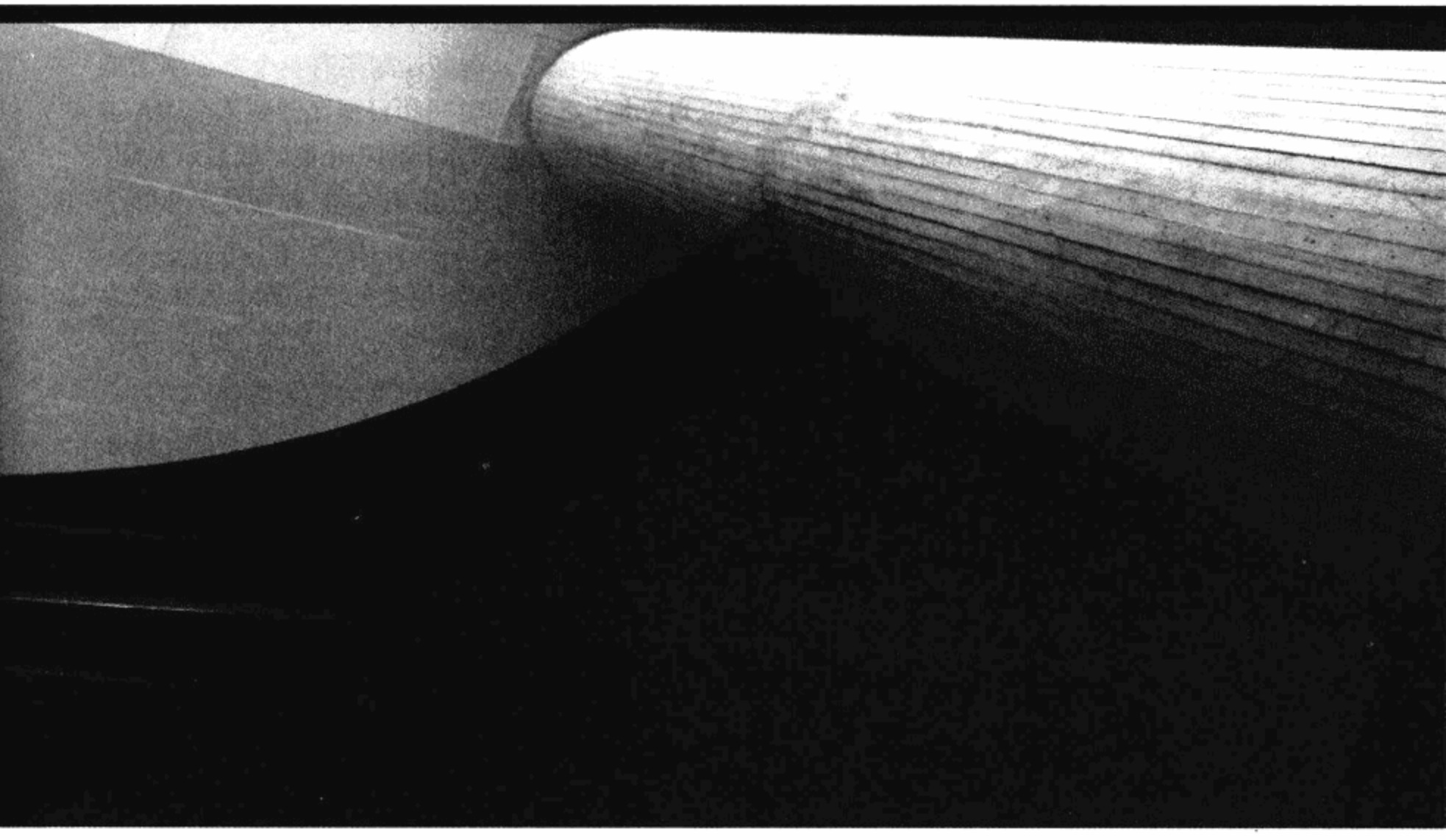
列表函数不以这种方式更改值，它们仅选择(或报告)某行中的一系列值的最大值(GREATEST)或最小值(LEAST)。单值函数和列表函数在遇到 NULL 值时都不会产生结果。

单值函数和列表函数能用于任何表达式中，如 select 子句和 where 子句。

组值函数提供关于整组数据(一个集合中的所有行)的信息。组值函数告诉您这些数字的平均值、最大值、数字的个数，或这些值的标准差等。组函数忽略 NULL 值，这一点在报告组值时必须牢记，否则在分析数据时将产生错误的结论。

组值函数也能报告表中子组的信息，或用来创建来自一个或多个表信息的总体视图。第 12 章将详细介绍这些附加的功能。

最后，算术和逻辑优先级将影响查询求值的顺序，这对查询结果有显著的影响。要养成使用圆括号的习惯，从而使执行顺序更清晰和易于理解。



第10章

日期：过去、现在及日期的差

Oracle 的一项强大功能是可以存储和计算日期，计算日期之间的秒、分钟、小时、天、

的功能。DATE 数据类型以特殊的 Oracle 内部格式存储，这种格式不仅包括月、日、年，而且包含小时、分钟、秒。这种格式的好处应该是不言而喻的。例如，如果有一个客户帮助桌面系统，则对于每次登录调用，Oracle 都可以自动地将调用的时间和日期存储到某个 DATE 列中。可在一个报表上格式化 DATE 列，使其只显示日期，或显示日期和小时，或显示世纪、日期、小时和分钟，或者显示日期、小时、分钟和秒。可利用 TIMESTAMP 数据类型存储小数部分的秒。详细内容请参考 10.7 节。

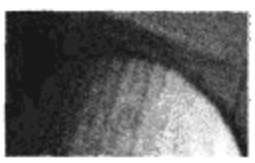
SQL*Plus 和 SQL 能够识别 DATE 数据类型的列，并能理解对这些列进行算术运算(称为日期运算而不是常规的数学计算)的指令。例如，给日期加 1 将得出另一个日期，即下一天；从一个日期中减去另一个日期会得出一个数字，这个数字为这两个日期间的天数差。

然而，因为 Oracle 的日期可以包括小时、分钟和秒，所以日期运算是比较复杂的，比如 Oracle 会得出今天与明天之间的差可能会是 0.516 天(这将在本章后面介绍)。

10.1.1 SYSDATE、CURRENT_DATE 及 SYSTIMESTAMP

Oracle 使用计算机操作系统中的当前日期和时间。它通过一个称为 SYSDATE 的特殊函数来达到这个目的。可将 SYSDATE 视为一个其结果始终为当前日期和时间的函数，在任何可以使用 Oracle 函数的其他地方都可以使用 SYSDATE。也可以将它视为每个表的一个隐藏列或伪列。下面的 SYSDATE 显示当天的日期：

```
Select SysDate from DUAL;

SYSDATE
-----------------
28-FEB-08
```



注意：
DUAL 是为了测试函数或进行快速计算而创建的一个小型但有用的 Oracle 表。10.1.2 节“用于快速测试和计算的 DUAL 表”部分将介绍 DUAL 表。

另一个函数 CURRENT_DATE 报告会话的时区中的系统日期(您可以设置自己会话的时区，以区别于数据库的时区)：

```
select Current_Date from DUAL;

CURRENT_D
---------
28-FEB-08
```

还有一个函数 SYSTIMESTAMP，它报告 TIMESTAMP 数据类型格式的系统日期：

```
select SysTimeStamp from DUAL;

SYSTIMESTAMP
---------------------------------------------
```

```
28-FEB-08 04.49.31.718000 PM -05:00
```

关于 TIMSTAMP 数据类型和时区格式及函数，请参考 10.7 节。以下几节将重点介绍 DATE 数据类型的用法，因为 DATE 能满足大多数数据处理的需求。

10.1.2　两个日期的差

HOLIDAY 是记录美国 2004 年中节假日的一个表：

```
select Holiday, ActualDate, CelebratedDate from HOLIDAY;

HOLIDAY                   ACTUALDAT  CELEBRATE
------------------------  ---------  ---------
NEW YEARS DAY             01-JAN-04  01-JAN-04
MARTIN LUTHER KING, JR.   15-JAN-04  19-JAN-04
LINCOLNS BIRTHDAY         12-FEB-04  16-FEB-04
WASHINGTONS BIRTHDAY      22-FEB-04  16-FEB-04
FAST DAY, NEW HAMPSHIRE   22-FEB-04  22-FEB-04
MEMORIAL DAY              30-MAY-04  31-MAY-04
INDEPENDENCE DAY          04-JUL-04  04-JUL-04
LABOR DAY                 06-SEP-04  06-SEP-04
COLUMBUS DAY              12-OCT-04  11-OCT-04
THANKSGIVING              25-NOV-04  25-NOV-04
```

这些节日中哪些在 2004 年中不是在当天进行庆祝的呢？可以通过从 CelebrateDate 减去 ActualDate 得出答案。如果结果不为 0，则表示两个日期不同：

```
select Holiday, ActualDate, CelebratedDate
  from Holiday
 where CelebratedDate - ActualDate != 0;

HOLIDAY                   ACTUALDAT  CELEBRATE
------------------------  ---------  ---------
MARTIN LUTHER KING, JR.   15-JAN-04  19-JAN-04
LINCOLNS BIRTHDAY         12-FEB-04  16-FEB-04
WASHINGTONS BIRTHDAY      22-FEB-04  16-FEB-04
MEMORIAL DAY              30-MAY-04  31-MAY-04
COLUMBUS DAY              12-OCT-04  11-OCT-04
```

用于快速测试和计算的 DUAL 表

DUAL 表是 Oracle 提供的一个小型表，它只有一行和一列：

```
Describe DUAL
Name                          Null?          Type
----------------------------  -------------  ----------------
DUMMY                                        VARCHAR(1)
```

因为 Oracle 的许多函数既处理列又处理文本量，利用 DUAL 能了解某些只处理文本量的函数。在下面的示例中，select 语句不关心表中有哪些列，而且只用一行数据就足以说明问题了。例如，假定希望快速地计算 POWER(4,3)，即计算 4 的立方。可用如下程序：

```
select power(4,3) from DUAL;
POWER(4,3)
----------
64
```

DUAL 表中实际的列不起作用。这表示可以利用 DUAL 表和日期函数设置日期格式及运算，从而理解它们如何起作用。然后，可把这些函数应用到表中的实际日期上。

10.1.3 添加月份

如果在新汉普郡 2 月 22 日是“斋戒日”，那么 6 个月后应当作为“斋戒日”庆祝。那么“斋戒日”庆祝日具体是哪一天呢？利用 ADD_MONTHS 函数添加 6 个月的数量即可计算出来：

```
column FeastDay heading "Feast Day"

select ADD_MONTHS(CelebratedDate,6) AS FeastDay
  from HOLIDAY
 where Holiday like 'FAST%';

Feast Day
---------
22-AUG-04
```

10.1.4 减少月份

假如必须至少在 Columbus Day 前 6 个月预定野餐地点，那么预定的最后期限是哪一天呢？从表中取出 Columbus Day 的 CelebratedDate，利用 ADD_MONTHS 函数添加负 6 个月(这相当于减去 6 个月)。这将得出在 Columbus Day 前 6 个月的日期。然后再减去一天即是预定的最后期限：

```
column LastDay heading "Last Day"

select ADD_MONTHS(CelebratedDate,-6) - 1 AS LastDay
  from HOLIDAY
 where Holiday = 'COLUMBUS DAY';

Last Day
---------
10-APR-04
```

10.1.5 GREATEST 和 LEAST

对于每个节日的实际日期和庆祝日期，哪个日期在前面呢？LEAST 函数从一个日期(不管是列还是日期字面值)表中选择最早的日期，而 GREATEST 函数从一个日期表中选择最近的一个日期。GREATEST 函数和 LEAST 函数用于数值和字符串时，其作用是相同的：

```
select Holiday, LEAST(ActualDate, CelebratedDate) AS First,
       ActualDate, CelebratedDate
  from HOLIDAY
 where ActualDate - CelebratedDate != 0;
```

```
HOLIDAY                   FIRST      ACTUALDAT CELEBRATE
------------------------  ---------  --------- ---------
MARTIN LUTHER KING, JR.   15-JAN-04  15-JAN-04 19-JAN-04
LINCOLNS BIRTHDAY         12-FEB-04  12-FEB-04 16-FEB-04
WASHINGTONS BIRTHDAY      16-FEB-04  22-FEB-04 16-FEB-04
MEMORIAL DAY              30-MAY-04  30-MAY-04 31-MAY-04
COLUMBUS DAY              11-OCT-04  12-OCT-04 11-OCT-04
```

其中，LEAST 效果很好，因为它作用于表的 DATE 列。那么，如果处理的是日期字面值又会怎样呢？

```
Select  LEAST('20-JAN-04','20-DEC-04') from DUAL;

LEAST('20
--------------
20-DEC-04
```

在此例中，两个字符串的前 3 个字符(20-)是相同的，第 4 个字符是基本的比较字符，在字母表中，D 位于 J 的前面。

LEAST 不知道将这两个字符串作为日期处理。利用 TO_DATE 函数可将这些日期字面值转换为 Oracle 能够用于面向日期的函数的内部 DATE 格式：

```
select  LEAST(TO_DATE('20-JAN-04'),TO_DATE('20-DEC-04'))
  from  DUAL;

LEAST(TO
---------
20-JAN-04
```

关于 GREATEST 和 LEAST 的警告

与许多其他 Oracle 函数和逻辑运算符不一样，GREATEST 和 LEAST 函数不会将日期格式的日期字面值作为日期处理。这些日期还是作为字符串处理。

```
select Holiday,CelebratedDate
  From HOLIDAY
where CelebrateDate=LEAST('19-JAN-04','06-SEP-04');

HOLIDAY          CELEBRATE
---------------  ----------
LABOR DAY        06-SEP-04
```

这就大错特错了，这样做的结果好像使用的是 GREATEST 而不是 LEAST。2004 年 10 月 20 日不是在 2004 年 1 月 20 日之前。为什么会发生这样的事情呢？原因是 LEAST 将这些日期字面值作为串处理了。

为使 LEAST 和 GREATEST 正常工作，必须对日期字面值串使用 TO_DATE 函数：

```
select Holiday,CelebratedDate
  from HOLIDAY
```

```
where CelebratedDate=LEAST( TO_DATE('19-JAN-04'),
                            TO_DATE('06-SEP-04') );


HOLIDAY                                    CELEBRATE
------------------------------------------ --------------------
MARTIN LUTHER KING,  JR                    19-JAN-04
```

10.1.6 NEXT_DAY

在给出某一天是星期几(即 Sunday、Monday、Tuesday、Wednesday、Thursday、Friday 或 Saturday)之后，NEXT_DAY 将计算下一个这样的星期几是哪一天。例如，假如每个发薪日都是每月 15 日之后的第一个星期五。在下面的表 PAYDAY 中包含全年的发薪日，每个发薪日都是当月的 15 号，2004 年的每个月占一行：

```
select CycleDate from PAYDAY;

CYCLEDATE
---------
15-JAN-04
15-FEB-04
15-MAR-04
15-APR-04
15-MAY-04
15-JUN-04
15-JUL-04
15-AUG-04
15-SEP-04
15-OCT-04
15-NOV-04
15-DEC-04
```

那么实际的发薪日是哪一天呢？

```
column Payday heading "Pay Day"

select NEXT_DAY(CycleDate,'FRIDAY') AS Payday
  from PAYDAY;

Pay Day
---------
16-JAN-04
20-FEB-04
19-MAR-04
16-APR-04
21-MAY-04
18-JUN-04
16-JUL-04
20-AUG-04
17-SEP-04
22-OCT-04
19-NOV-04
```

```
17-DEC-04
```

这个答案基本正确，除了在 10 月有一些误差，因为 NEXT_DAY 是在每个发薪日之后的第一个星期五。由于 2004 年 10 月 15 日本身就是星期五，但是却(错误地)给出了下一个星期五。正确的方法如下所示：

```
column payday heading "Pay Day"

select NEXT_DAY(CycleDate-1,'FRIDAY') As PayDay
  from PAYDAY;
```

NEXT_DAY 实际是一种“大于”函数。它要求下一个日期大于属于本周中某一天的指定日期。为了找到那些已经是星期五的日期，可从发薪日中减去一天。这使每个发薪日都比 NEXT_DAY 早一天，从而得出的发薪日期总是正确的星期五。

10.1.7　LAST_DAY

LAST_DAY 计算每个月最后一天是哪一天。假定佣金和红利总是在每月的最后一天发放，那么 2004 年这样的日期有哪些呢？

```
column EndMonth heading "End Month"

select LAST_DAY(CycleDate) AS EndMonth
  from PAYDAY;

End Month
---------
31-JAN-04
29-FEB-04
```

```
VICTORIA        LYNN            20-MAY-49
FRANK           PILOT           11-NOV-42
```

为了算出每个人的年龄，必须计算今天的日期和他们的生日之间相差的月份，然后除以12得到年数：除法的结果将会显示带小数部分的年龄。因为大部分7岁以上的人不需要报出年龄的小数部分，所以可对计算结果使用FLOOR函数。

```
select FirstName, LastName, BirthDate,
       FLOOR(
            MONTHS_BETWEEN(SysDate,BirthDate)/12
            ) AS Age
  from BIRTHDAY;

FIRSTNAME       LASTNAME        BIRTHDATE        AGE
--------------- --------------- ---------   -------
GEORGE          SAND            12-MAY-46         61
ROBERT          JAMES           23-AUG-37         70
NANCY           LEE             02-FEB-47         61
VICTORIA        LYNN            20-MAY-49         58
FRANK           PILOT           11-NOV-42         65
```

10.1.9 组合日期函数

假设您在2008年2月28日得到一份新工作，虽然开始的薪水比期望的低，但是公司承诺6个月后加薪。如果当前的日期是2008年2月28日，那么评审加薪的日期是哪一天呢？

```
select SysDate AS Today,
       LAST_DAY(ADD_MONTHS(SysDate,6)) + 1 Review
  from DUAL;

TODAY       REVIEW
---------   ---------
28-FEB-08   01-SEP-08
```

ADD_MONTHS在SysDate的基础上加6个月。LAST_DAY得到ADD_MONTHS的结果月份并计算当月的最后一天。接着在LAST_DAY的结果上加1得到下一个月的第一天。那么在评审加薪前还有多少天呢？只要从前面的结果中减去今天的日期即可。注意，应使用圆括号以保证正确的计算顺序：

```
select (LAST_DAY(ADD_MONTHS(SysDate,6)+1)-SysDate Wait
  from DUAL;

      WAIT
--------
       186
```

10.2 日期计算中的 ROUND 和 TRUNC

下面是今天的SysDate时间：

```
SYSDATE
---------
28-FEB-08
```

本章开始时提及过 Oracle 可从一个日期减去另一个日期(如明天减今天)，结果却不一定是一个整数。如下所示：

```
select TO_DATE('29-FEB-08') - SysDate from DUAL;
TO_DATE('29-FEB-08')-SYSDATE
----------------------------
                        .516
```

明天和今天的日期之差为小数的原因在于 Oracle 保存了日期的小时、分钟、秒，SysDate 总是精确到当前时间的秒数。显而易见，距离明天不到一整天。

为了减少使用非整数天数所带来的麻烦，Oracle 对日期做了以下几个假定：

- 以日期字面值输入的日期(如 29-FEB-08)的默认时间为当天的开始时间 0 时(即午夜)。
- 通过 SQL*Plus 输入的日期，除非特别指定时间，否则将它设置为当天的开始时间 0 时(即午夜)。
- 除非特意舍入，否则 SysDate 总是包括日期和时间。如果某天的时间在中午之前，则使用 ROUND 函数将该天的时间设置为零点；如果其时间为中午之后，则设置为次日的零点。TRUNC 函数的功能与之类似，只不过它将任何时间，包括午夜前一秒也设置为当天的零点。

为了得到今天和明天之间天数的舍入数，可使用如下方法：

```
select TO_DATE('29-FEB-08') - ROUND(SysDate) from DUAL;
TO_DATE('29-FEB-08')-ROUND(SYSDATE)
-----------------------------------
                                  1
```

如果所用的当前时间在下午，则时间差的舍入为 0 天。

如果未指定 format(参考附录 A 中的“日期函数”)，则 ROUND 总是将日期舍入为最接近一天的零点。如果所使用的日期包含非正午时间，则在计算中或者使用 ROUND 或者接受可能的小数结果。虽然 TRUNC 的作用与之类似，但它将时间设置为当天的午夜 0 时。

10.3　使用 TO_DATE 和 TO_CHAR 设置日期格式

从都具有很强的格式化功能这个角度来说，TO_DATE 和 TO_CHAR 是类似的。但它们在功能上正好相反，TO_DATE 将字符串或数字转换为 Oracle 日期，而 TO_CHAR 将 Oracle 日期转换为字符串。这两个函数的格式如下所示：

```
TO_CHAR( date[,' format'[,' NLSparameters']])
TO_DATE( string[,' format'[,' NLSparameters']])
```

date 必须是一个定义为 Oracle 中 DATE 类型的列。它不能是一个串，即使这个串的格式

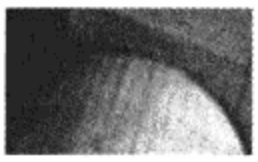
为最常用的日期格式 DD-MON-YY 也不行。使用 date 出现在 TO_CHAR 函数内的串的唯一方法是把它括在 TO_CHAR 函数内。

string 为一个日期字面值的串、日期字面值的数值或一个包含串或数值的数据库列。在各种情况下，string 的格式必须符合 format 的描述。只有在 string 为默认格式时可以省略 format。虽然默认情况下以 DD-MON-YY 开始，但对于给定的 SQL 会话或带 NLS_DATE_FORMAT init.ora 参数的会话，可以用如下形式更改：

```
alter session set NLS_DATE_FORMAT="DD/MON/YYYY";
```

format 是许多选项的一个集合，该组合可组合成各种各样的形式(参见本书附录 A 中的“日期格式”)。一旦理解了使用这些选项的基本方法，就会发现在实际中使用它们是很简单的。

NLSparameters 是一个设置 NLS_DATE_LANGUAGE 选项为某种特殊语言而不是当前 SQL 会话使用的语言的串。不应该经常使用这个选项。Oracle 将以使用 alter session 的会话的语言返回日期和月份的名称。



注意：
在很多情况下，您可以使用 EXTRACT 函数代替 TO_CHAR。实际的示例请参考 10.6 节。

以 TO_CHAR 为例来说明怎样使用这些选项。首先为 TO_CHAR 函数结果定义一个格式，因为不这样做，TO_CHAR 将在 SQL*Plus 中生成近 100 个字符宽的列标题。通过对列重新命名(使其标题好理解)，并设置其格式为 30 个字符，可以产生一种便于理解的显示：

```
column Formatted format a30 word_wrapped
select BirthDate,
       TO_CHAR(BirthDate,'MM/DD/YY') AS Formatted
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTHDATE  FORMATTED
---------  ------------------------------
20-MAY-49  05/20/49
```

BirthDate 显示出了默认的 Oracle 日期格式：DD-MON-YY，即月份中的日期、-、3 个字符的缩写月份、-、两个数字的年份。select 语句中的 TO_CHAR 函数意思很明显，就不作解释了。TO_CHAR 语句中的 MM、DD 和 YY 是 Oracle 中格式化日期的关键符号。斜杠(/)为标点符号，Oracle 认可各种标点符号：

```
select BirthDate, TO_CHAR(BirthDate,'YYMM>DD') Formatted
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTHDATE FORMATTED
--------- ------------------------------
20-MAY-49 4905>20
```

除了标准的标点外，Oracle 允许在 format 中插入文本。这通过将所需文本括在双引号中来完成：

```
select BirthDate,
       TO_CHAR(BirthDate,'Month, DDth "in, um,"
                YyyY') AS Formatted
  from BIRTHDAY ;

BIRTHDATE FORMATTED
--------- ------------------------------
12-MAY-46 May       ,      12TH in, um,
           1946
23-AUG-37 August    ,      23RD in, um,
           1937
02-FEB-47 February  ,      02ND in, um,
           1947
20-MAY-49 May       ,      20TH in, um,
           1949
11-NOV-42 November  ,      11TH in, um,
           1942
```

在这里要注意 format 的几种结果。完整的单词 Month 告诉 Oracle 在显示时使用月份的全名。由于输入时它的第一个字符为大写，其余部分为小写，因此结果中的每个月都格式化为相同的形式。月份的选项如下所示：

格 式	结 果
Month	August
Month	August
Mon	Aug
Mon	Aug

按 format 中的 DD 格式生成月份中的天。DD 的后缀 th 告诉 Oracle 对数值使用序数后缀，如 TH、RD 和 ND 等。虽然在下面的示例中，这个后缀也是区分大小写的，但它们的大小写由 DD 而不是 th 确定：

格 式	结 果
DDth 或 DDTH	11TH
Ddth 或 DdTH	11Th
Ddth 或 ddTH	11th

同样的方法对 format 中所有数值都一样。这些数值包括世纪、年、季度、月、周、月份中的天(DD)，公历日、小时、分钟和秒。

双引号之间的词插在放置它们的地方。这些格式之间的空格也在结果中重新生成(请参考前面示例中单词“in”之前的 3 个空格)。包括 YyyY 只是为了表明大小写没有关系，除非使用诸如 Th 这样的后缀。为简单起见，考虑下面的格式请求：

```
select BirthDate, TO_CHAR(BirthDate,'Month, ddth, YyyY')
          AS Formatted
  from BIRTHDAY;

BIRTHDATE FORMATTED
--------- ------------------------------
12-MAY-46 May       ,  12th, 1946
23-AUG-37 August    ,  23rd, 1937
02-FEB-47 February  ,  02nd, 1947
20-MAY-49 May       ,  20th, 1949
11-NOV-42 November  ,  11th, 1942
```

这样就合理地规范了格式。天数全都是对齐的，这样使得比较每一行很方便。这是默认的对齐方式，Oracle 通过在月份名的右边填充最多 9 个空格来完成对齐。在某些情况下，规范地格式化日期很重要，如在信封上面。月份和逗号之间的空格看上去有点怪。为了删除这些空格，可对单词“month”或“day”使用前缀 fm:

格 式	结 果
Month, ddth	August , 20th
fmMonth, ddth	August, 20th
Day, ddth	Monday , 20th
fmDay, ddth	Monday, 20th

举例说明如下：

```
select BirthDate, TO_CHAR(BirthDate,'fmMonth, ddth, YyyY')
          AS Formatted
  from BIRTHDAY;

BIRTHDATE FORMATTED
--------- ------------------------------
12-MAY-46 May, 12th, 1946
23-AUG-37 August, 23rd, 1937
02-FEB-47 February, 2nd, 1947
20-MAY-49 May, 20th, 1949
11-NOV-42 November, 11th, 1942
```

结合这些格式控制并加上小时和分钟，可生成一个出生通知：

```
select FirstName, Birthdate, TO_CHAR(BirthDate,
'"Baby Girl on" fmMonth ddth, YYYY, "at" HH:MI "in the Morning"')
        AS Formatted
  from BIRTHDAY
 where FirstName = 'VICTORIA';

FIRSTNAME       BIRTHDATE  FORMATTED
--------------  ---------  ------------------------------
VICTORIA        20-MAY-49  Baby Girl on May 20th, 1949,
                           at 3:27 in the Morning
```

假如看了上面的通知后，决定把日期拼写出来。可利用如下的 sp 控制：

```
select FirstName, BirthDate, TO_CHAR(BirthDate,
'"Baby Girl on the" Ddsp "of" fmMonth, YYYY, "at" HH:MI')
          AS Formatted
   from BIRTHDAY
  where FirstName = 'VICTORIA';

FIRSTNAME      BIRTHDATE  FORMATTED
-------------  ---------  ------------------------------
VICTORIA       20-MAY-49  Baby Girl on the Twenty of
                          May, 1949, at 3:27
```

虽然现在 20 被拼写出来了，但是觉得还不够。于是在 sp 后面添加 th 后缀：

```
select FirstName, BirthDate, TO_CHAR(BirthDate,
'"Baby Girl on the" Ddspth "of" fmMonth, YYYY, "at" HH:MI')
       AS Formatted
   from BIRTHDAY
  where FirstName = 'VICTORIA';

FIRSTNAME      BIRTHDATE  FORMATTED
-------------- ---------  ------------------------------
VICTORIA       20-MAY-49  Baby Girl on the Twentieth of
                          May, 1949, at 3:27
```

但是，究竟是 3:27 A.M.还是 3:27 P.M.呢？可在双引号中添加它们。但这样一来结果总是显示 A.M.或 P.M.，而不管实际时间是多少(因为双引号括住的是一个日期字面值)。所以，虽然 Oracle 允许在时间后添加 A.M.或 P.M.，但不是在双引号中添加。Oracle 会将其理解为一种显示请求。请注意，select 如何得到这种以 P.M.为输入但显示的却是 A.M.的格式控制，因为出生时间是早晨：

```
select FirstName, BirthDate, TO_CHAR(BirthDate,
'"Baby Girl on the" Ddspth "of" fmMonth, YYYY, "at" HH:MI P.M.')
         AS Formatted
   from BIRTHDAY
  where FirstName = 'VICTORIA';

FIRSTNAME      BIRTHDATE  FORMATTED
-------------- ---------  ------------------------------
VICTORIA       20-MAY-49  Baby Girl on the Twentieth of
                          May, 1949, at 3:27 A.M.
```

所有可能的日期选项可参见本书附录“命令和术语参考”中的“日期格式”。那么，怎样为路易九世统治时期的第 782 年构造一个日期格式呢？可利用日期运算将年份从 A.D.换成 A.L.(由于路易王朝从 1226 开始，因此应该从当前年份中减去 1226)，然后再利用 TO_CHAR 格式化这个结果。

10.3.1 最常见的 TO_CHAR 错误

在使用 TO_CHAR 函数时经常需要检查日期的格式。在格式化日期的时间部分时，最为常见的错误是互换 MM(月)格式和 MI(分钟)格式。

例如，为了查看当前时间，可以用 TO_CHAR 函数来查询 SysDate 的时间部分：

```
select TO_CHAR(SysDate,'HH:MI:SS') Now
  from DUAL;

NOW
--------
05:11:48
```

这个示例是正确的，因为它使用 MI 来表示分钟。但是用户经常会错误地使用 MM，有时是因为他们还选择了双字符的其他两对 HH 和 SS。选择 MM 将返回月份而不是分钟：

```
select TO_CHAR(SysDate,'HH:MM:SS') NowWrong
  from DUAL;

NOWWRONG
--------
05:02:48
```

这个时间是错误的，因为在分钟的位置选择了月份。由于 Oracle 非常灵活，支持许多不同的日期格式，因此它不会阻止您犯这样的错误。

10.3.2 NEW_TIME：切换时区

NEW_TIME 函数显示其他时区中某个日期列或日期字面值的时间和日期。其格式如下所示：

```
NEW_TIME(date, 'this', 'other')
```

date 是 this 时区内的日期(和时间)。This 将被 3 个字母组成的缩写代替，表示当前时区。Other 将被需要知道其时间和日期的其他时区的 3 个字母缩写代替(参见本书附录 A 中的“日期函数”)。要比较东部标准时间和夏威夷标准时间的维多利亚生日，只比较日期，不显示时间，可用如下方法：

```
select Birthdate, NEW_TIME(Birthdate,'EST','HST')
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTHDATE NEW_TIME(
--------- ---------
20-MAY-49 19-MAY-49
```

但是维多利亚怎么可能有两个不同的生日呢？由于存储在 Oracle 中的每个日期包含一个时间，因此完全可以通过 TO_CHAR 和 NEW_TIME 来得出两个时区间的日期与时间差，如下所示：

```
select TO_CHAR(Birthdate,'fmMonth Ddth, YYYY "at" HH:MI AM') AS Birth,
       TO_CHAR(NEW_TIME(Birthdate,'EST','HST'),
       'fmMonth ddth, YYYY "at" HH:MI AM') AS Birth
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTH                          BIRTH
------------------------------ ------------------------------
May 20th, 1949 at 3:27 AM      May 19th, 1949 at 10:27 PM
```

10.3.3　TO_DATE 计算

虽然 TO_DATE 与 TO_CHAR 有相同的格式约定，但有一些限制。TO_DATE 的用途是将一个日期字面值串(如 MAY 20，1949)转换为 Oracle 日期格式。这样能将日期用于日期计算。

TO_DATE 的格式如下所示：

```
TO_DATE(String[,'format'])
```

例如，可按下面这样将串 28-FEB-08 转换为 Oracle 日期格式：

```
select TO_DATE('28-FEB-08','DD-MON-YY') from DUAL;

TO_DATE('
---------
28-FEB-08
```

注意，28-FEB-08 格式已经是 Oracle 显示和支持的默认日期格式了。当一个日期字面值串以这样的格式表示时，省略 TO_DATE 中的 format 也能得到相同的结果：

```
select TO_DATE('28-FEB-08') from DUAL;

TO_DATE('
---------
28-FEB-08
```

注意标点被忽略了。即使默认日期格式是 28/FEB/08，查询也仍将执行，只不过返回的日期格式是 28/FEB/08。

但日期中的世纪怎么处理呢？它是 1908 还是 2008？如果未指定年份的完整的 4 位数字表示，则需要根据数据库默认值设置为适当的世纪值。

如果该串的格式类似，但不是默认的 Oracle 格式 DD-MON-YY，则 TO_DATE 失败：

```
select TO_DATE('02/28/08') from DUAL;

select TO_DATE('02/28/08') from DUAL;
               *
ERROR at line 1:
ORA-01843: not a valid month
```

在格式与日期字面值串相匹配时，该串被成功地转换为一个日期，并以默认日期格式显示：

```
select TO_DATE('02/28/08','MM/DD/YY') from DUAL;

TO_DATE('
---------
28-FEB-08
```

假定需要知道 3 月 17 日是星期几，这时就不能使用 TO_CHAR 函数，即使对于具有适当格式的日期字面值串也是如此，原因在于 TO_CAHR 需要一个日期(请参考 10.3 节最开始的内容)。

```
select TO_CHAR('17-MAR-08','Day') from DUAL;

ERROR at line 1:
ORA-01722: invalid number
```

这个消息有时会误导，不过事实上是这个查询失败了。虽然您可以使用 EXTRACT 函数，但如果事先把串转换成日期，则这个查询也不会出错了。可结合使用 TO_CHAR 和 TO_DATE 这两个函数来达到这个目的：

```
select TO_CHAR( TO_DATE('17-MAR-08'), 'Day') from DUAL;

TO_CHAR(T
---------
Monday
```

TO_DATE 不仅可以接收串日期值，也可以接收数值(没有单引号)，只要对它们进行一致的格式化即可。示例如下所示：

```
select TO_DATE(11051946,'MMDDYYYY') from DUAL;

TO_DATE(1
---------
05-NOV-46
```

虽然 format 中的标点符号被忽略，但数值必须遵循格式控制的顺序。数值自身一定不能有标点符号。

TO_DATE 中的格式控制可以复杂到什么程度呢？假如颠倒前面的 TO_DATE select 语句，将其结果放入 TO_DATE 的 string 部分，并保持其格式与 TO_CHAR 相同：

```
select TO_DATE('Baby Girl on the Twentieth of May, 1949,
        at 3:27 A.M.',
        '"Baby Girl on the" Ddspth "of" fmMonth, YYYY,
        "at" HH:MI P.M.')
        AS Formatted
  from BIRTHDAY
 where FirstName = 'VICTORIA';

ERROR at line 1:
ORA-01858: a non-numeric character was found
 where a numeric was expected
```

这显然是失败的。原来，只能使用有限数目的格式控制。下面是控制 TO_DATE 的 format

```
------------------------   ---------
NEW YEARS DAY              01-JAN-04
FAST DAY, NEW HAMPSHIRE    22-FEB-04
```

如果不想让世纪的默认值为 2000，则可以在 IN 运算符中用 TO_DATE 函数指定日期的世纪值：

```
select Holiday, CelebratedDate
  from HOLIDAY
 where CelebratedDate IN
       (TO_DATE('01-JAN-2004','DD-MON-YYYY'),
        TO_DATE('22-FEB-2004','DD-MON-YYYY'));

HOLIDAY                    CELEBRATE
------------------------   ---------
NEW YEARS DAY              01-JAN-04
FAST DAY, NEW HAMPSHIRE    22-FEB-04
```

LEAST 和 GREATEST 不起作用，原因在于它们假定日期字面值串是串而不是日期。关于 LEAST 和 GREATEST 的说明，请参考 10.1.4 节中“关于 LEAST 和 GREATEST 的警告”的内容。

10.5 处理多个世纪

如果应用程序只使用两位数字的年份值，则可能会遇到与 2000 年相关的问题。如果年份只指定两位数字(如用“98”表示“1998”)，则插入记录时，世纪值(19)由数据库来指定。如果给出 2000 年以前的日期(如出生日期)，则为数据分配的世纪值可能会出问题。

在 Oracle 中，所有日期值都有世纪值。如果只指定年份值的最后两位数字，则在插入记录时，Oracle 将默认使用当前世纪作为世纪值。例如，下面的程序清单显示了 BIRTHDAY 表的一个 insert 语句：

```
insert into BIRTHDAY
 (FirstName, LastName, BirthDate)
values
('ALICIA', 'ANN', '21-NOV-39');
```

上面的示例中未指定 BirthDate 列的世纪值，也没有指定年龄。如果对 BirthDate 列使用 TO_CHAR 函数，则将看到 Oracle 插入的完整的出生日期，它默认的世纪为当前世纪：

```
select TO_CHAR(BirthDate,'DD-MON-YYYY') AS Bday
  from BIRTHDAY
 where FirstName = 'ALICIA'
   and LastName = 'ANN';

BDAY
-----------
21-NOV-2039
```

对于默认世纪为当前世纪的日期，使用默认值不会出现问题。Alicia 的 BirthDate 值为 21-NOV-2039，差了 100 年！所以，不论在哪里插入日期值时，都应该指定完整的 4 位数字的年份值。

如果要支持多个世纪的日期，则可以考虑使用“RR”日期格式来代替“YY”。年采用“RR”格式将会根据年的值来确定世纪值。因此，前半个世纪的年值，其世纪值是 20；而后半个世纪的年值，其世纪值是 19。使用“RR”日期格式可以在数据库级别指定，也可以在 SQL 语句级别指定(如在 insert 操作期间)。

10.6　使用 EXTRACT 函数

您可以使用 EXTRACT 函数代替 TO_CHAR 函数来选择日期值的某一部分(如从一个日期中选择月份和天)。EXTRACT 函数的语法如下所示：

```
EXTRACT
    ( { { YEAR
        | MONTH
        | DAY
        | HOUR
        | MINUTE
        | SECOND
        }
      | { TIMEZONE_HOUR
        | TIMEZONE_MINUTE
        }
    | { TIMEZONE_REGION
      | TIMEZONE_ABBR
```

分之一秒的日期。

时间戳值的基本数据类型称为 TIMESTAMP。与 DATE 一样，它存储年、月、日、小时、分钟、秒。TIMESTAMP 还包括 fractional_seconds_precision 设置，用来确定秒字段小数部分的数字位数。默认情况下，精度为 6，有效值为 0～9。

在下面的示例中，用 TIMESTAMP 数据类型创建一个表，通过 SYSTIMESTAMP 函数来填充表：

```
create table X1
(tscol TIMESTAMP(5));

insert into X1 values (SYSTIMESTAMP);
```

现在从表中选择值：

```
select * from X1;
TSCOL
---------------------------------
28-FEB-08 05.27.32.71800 PM
```

输出结果显示插入行时的秒，精确到小数点后 5 位数字。

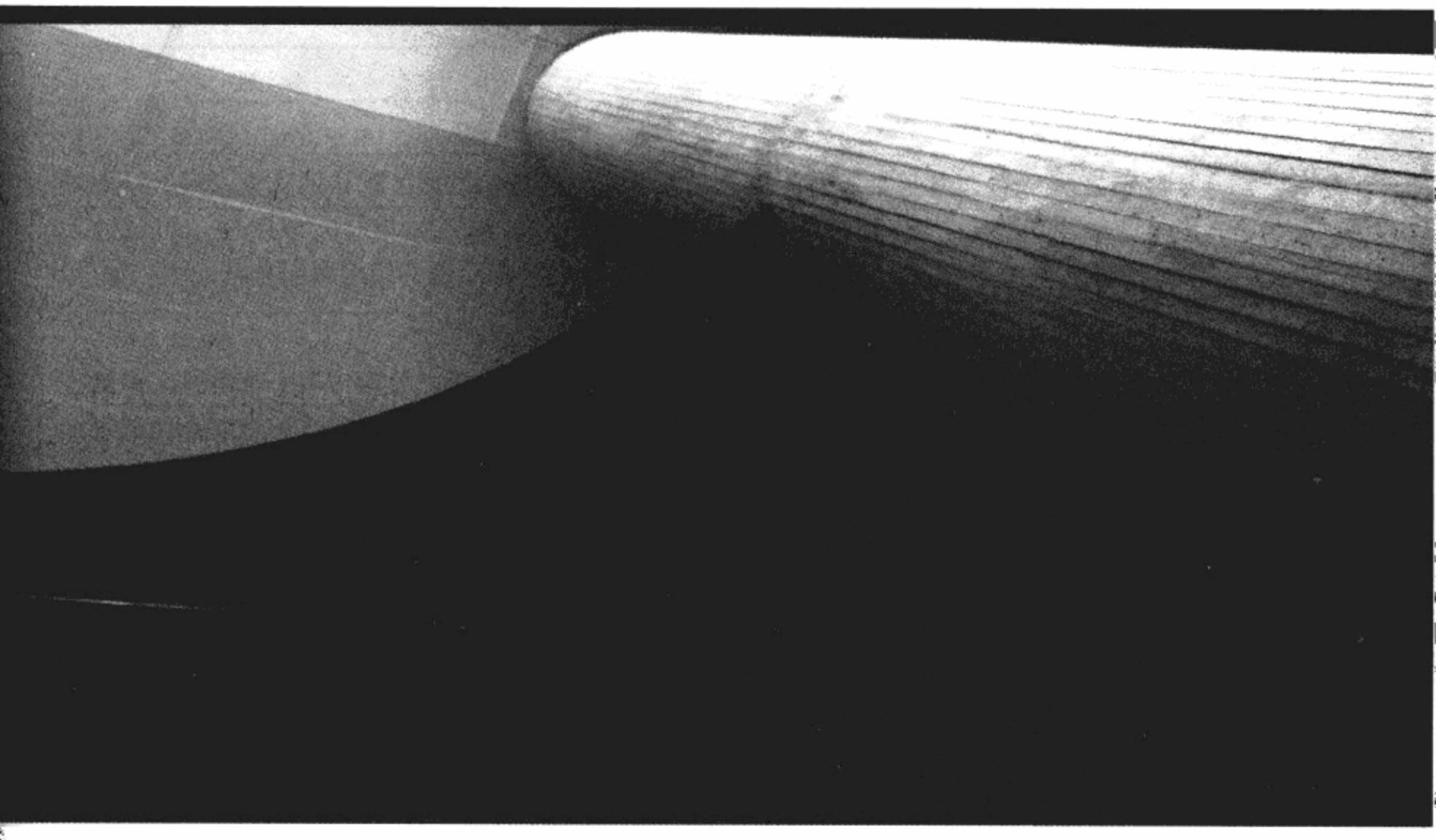
SYSTIMESTAMP 函数以 TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE 数据类型的形式返回数据。插入到用 TIMESTAMP(5) WITH TIME ZONE 数据类型定义的列中的完全相同的行，以下面的格式返回上述数据：

```
create table X2
(tscol TIMESTAMP(5) WITH TIME ZONE);

insert into X2 values (SYSTIMESTAMP);

select * from X2;

TSCOL
---------------------------------
28-FEB-08 05.29.11.64000 PM -05:00
```

在这个输出结果中，时区显示为国际标准时间(Coordinated Universal Time，UTC)的一个偏移值。此数据库当前设置的时区比 UTC 早 5 个小时。

Oracle 还支持 TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE 数据类型，此数据类型类似于 TIMESTAMP WITH TIME ZONE。不同之处在于数据在存储到数据库中时被规范化为数据库时区，检索时用户看到的是以会话时区为标准的数据。

除了 TIMESTAMP 数据类型外，Oracle 还支持两种时间间隔数据类型：INTERVAL YEAR (year_precision) TO MONTH 和 INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)。INTERVAL YEAR TO MONTH 以年和月存储时间间隔，其中精度为 YEAR 字段中的数字位数(范围为 0～9，默认为 2 位数字)。INTERVAL DAY TO SECOND 数据类型以天、小时、分钟、秒存储时间间隔，天和秒的精度值为 0～9。INTERVAL 数据类型在统计分析和数据挖掘中经常用到。



第11章

转换函数与变换函数

本章将介绍转换函数或变换函数，它们将一种数据类型转换为另一种数据类型。迄今为止介绍的4种主要数据类型及与之相关的函数包括：

- CHAR(定长字符串)和 VARCHAR2(可变长字符串)包括字母表中任意字母、任意数字以及键盘上的任何符号。字符字面值必须用单引号括起来，如

  ```
  'Sault Ste. Marie!'
  ```

- NUMBER 只包含数字 0～9。如果有必要，还可以包含小数点以及负号。NUMBER 字面值不需要括在引号内，如

```
246.320
```

NUMBER 也可以用浮点格式显示，如

```
2.4632E+2
```

- DATE 是一种包含日期和时间信息的特殊数据类型。虽然其默认格式为 DD-MON-YY(取决于 NLS_DATE_FORMAT 参数的设置)，但正如第 10 章所述，可通过 TO_CHAR 函数以多种格式将其显示出来。DATE 字面值必须用单引号括起来，如

```
'26-AUG-81'
```

每种数据类型都有一组专门针对该类型数据进行处理的函数，如第 7~10 章所述。串函数用于字符列或字面值，算术函数用于 NUMBER 列或字面值，日期函数用于 DATE 列或字面值。大多数分组和其他函数能使用所有这些数据类型。有一些函数会更改它们所涉及的对象(无论是 CHAR、VARCHAR2、NUMBER 还是 DATE)，而其他函数则只报告相关对象的信息。

从某种意义上讲，目前所研究的大多数函数都是转换函数，这表示它们都对其对象进行更改。然而，本章所介绍的函数以一种不寻常的方式更改它们的对象，即它们将数据从一种类型变换为另一种类型，或者说它们对其中的数据做一种复杂的变换。表 11-1 描述了这些函数。

表 11-1 转 换 函 数

函 数 名	定 义
ASCIISTR	转换任意字符集的串，并返回本数据库字符集的 ASCII 串
BIN_TO_NUM	将二进制值转换为其等价的数字值
CAST	将一种内部或集合类型强制转换为另一种内部或集合类型；一般用于嵌套表和可变数组
CHARTOROWID	更改字符串，使其可用作 Oracle 内部行标识符或 ROWID
COMPOSE	将任意类型的串转为 Unicode 标准格式的串，且字符集与输入串相同
CONVERT	将字符串从一种国家语言字符集转换为另一种国家语言字符集。返回格式是 VARCHAR2
DECOMPOSE	将任意数据类型的串转换为用与输入相同的字符集规范分解后的 Unicode 串
HEXTORAW	将十六进制数的字符串转换为二进制
NUMTODSINTERVAL	将 NUMBER 转换为 INTERVAL DAY TO SECOND 字面值
NUMTOYMINTERVAL	将 NUMBER 转换为 INTERVAL YEAR TO MONTH 字面值
RAWTOHEX	将二进制数字串转换为十六进制数字串
RAWTONHEX	将 RAW 转换为包含其十六进制等价形式的 NVARCHAR2 字符值
ROWIDTOCHAR	将 Oracle 的内部行标识符或 ROWID 转换为字符串
ROWIDTONCHAR	将 ROWID 值转换为 NVARCHAR2 数据类型的值
SCN_TO_TIMESTAMP	将系统改变号(SCN)转换为接近的时间戳
TIMESTAMP_TO_SCN	将时间戳转换为接近的系统改变号(SCN)

(续表)

函 数 名	定　义
TO_BINARY_DOUBLE	返回双精度浮点数
TO_BINARY_FLOAT	返回单精度浮点数
TO_CHAR	将 NUMBER 或 DATE 类型的值转换为字符串
TO_CLOB	将 LOB 列中的 NCLOB 值或者其他字符串转换为 CLOB 值
TO_DATE	将 NUMBER、CHAR 或 VARCHAR2 转换为 DATE(一种 Oracle 数据类型)类型值
TO_DSINTERVAL	将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 INTERVAL DAY TO SECOND 类型
TO_LOB	将 LONG 转换为 LOB，作为 insert as select 的一部分
TO_MULTI_BYTE	将字符串中的单字节字符转换为多字节字符
TO_NCHAR	将字符串、NUMBER 或 DATE 从数据库字符集转换为国家语言字符集
TO_NCLOB	将 LOB 列中的 CLOB 值或者其他字符串转换为 NCLOB 值
TO_NUMBER	将 CHAR 或 VARCHAR2 转换为数值
TO_SINGLE_BYTE	将 CHAR 或 VARCHAR2 中的多字节字符转换为单字节字符
TO_TIMESTAMP	将字符串转换为 TIMESTAMP 数据类型的值
TO_TIMESTAMP_TZ	将字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型的值
TO_YMINTERVAL	将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 INTERVAL YEAR TO MONTH 类型
TRANSLATE…USING	将串中的 TRANSLATE 字符变换为不同的字符
UNISTR	将串转换为数据库 Unicode 字符集中的 Unicode 字符

11.1　基本的转换函数

虽然表 11-1 中列出了许多转换函数，但最常用的是下面 3 个转换函数，其用途是将一种数据类型转换成另一种数据类型：

- TO_CHAR　将 DATE 或 NUMBER 转换为字符串。
- TO_DATE　将 NUMBER、CHAR 或 VARCHAR2 转换为 DATE。当用到时间戳时，可以用 TO_TIMESTAMP 或 TO_TIMESTAMP_TZ。
- TO_NUMBER　将 CHAR 或 VARCHAR2 转换为 NUMBER。

为什么这些变换很重要呢？TO_DATE 对于完成日期运算显然是很必要的。TO_CHAR 使您能利用串函数处理数，就好像它是串一样。TO_NUMBER 使您能够使用仅包含数字的串，就像它是一个数一样。利用它，可以进行加、减、乘，除等运算。

例如，假如把一个有 9 位数字的邮政编码(ZIP)以数字的形式存储，则可以首先将它变换为串的形式，然后用 SUBSTR 和连接添加一个-(与在信封上打印地址时一样)：

```
select SUBSTR(TO_CHAR(948033515),1,5)||'-'||
       SUBSTR(TO_CHAR(948033515),6) AS Zip
  from DUAL;

ZIP
----------------------------------------
94803-3515
```

其中，TO_CHAR 函数将纯数值 948033515(注意，该数字没有像 CHAR 和 VARCHAR2 串那样必须用单引号括起来)转换成一个字符串。然后 SUBSTR 将该字符串的前 5 个数字提取出来，得到 94803。在这个串的右边连接一个-，接着用另一个 TO_CHAR 函数生成了另一个串，然后用另一个 SUBSTR 从这个串的第 6 位数字到末尾提取出第 2 个串。所提取出来的第 2 个串是 3515，它被连接到-之后。重新生成的这个串就是重新标出的邮编，Oracle 将其显示为 94803-3515。TO_CHAR 函数允许用户对数值(以及日期)使用串处理函数，就像它们是串一样。确实很方便。但是请看下面的示例：

```
select SUBSTR(948033515,1,5)||'-'||
       SUBSTR(948033515,6) AS Zip
  from DUAL;

ZIP
----------------------------------------
94803-3515
```

该语句此时就不起作用了，原因在于 948033515 此时是一个 NUMBER 类型的数据，而不是字符串。因此，字符串函数 SUBSTR 显然要出错。那么，它对实际的 NUMBER 数据库列有作用吗？在下面的表中，Zip 为 NUMBER 类型：

```
describe ADDRESS

Name                              Null?    Type
--------------------------------  -------- ------------
LASTNAME                                   VARCHAR2(25)
FIRSTNAME                                  VARCHAR2(25)
STREET                                     VARCHAR2(50)
CITY                                       VARCHAR2(25)
STATE                                      CHAR(2)
ZIP                                        NUMBER
PHONE                                      VARCHAR2(12)
EXT                                        VARCHAR2(5)
```

在表中选出所有名字为 Mary 的 Zip 代码：

```
select SUBSTR(Zip,1,5)||'-'||
       SUBSTR(Zip,6) AS Zip
```

```
  from ADDRESS
 where FirstName = 'MARY';

ZIP
----------------------------------------
94941-4302
60126-2460
```

这里，即使 ADDRESS 表中的 Zip 是一个 NUMBER 列，SUBSTR 也是有效的，就像处理串一样。那么其他的串函数也可以这么用吗？

```
select Zip, RTRIM(Zip,20)
  from ADDRESS
 where FirstName = 'MARY';

       ZIP RTRIM(ZIP,20)
---------- ----------------------------------------
 949414302 9494143
 601262460 60126246
```

左边的列说明 Zip 是一个 NUMBER，这一列是右对齐的，一般数值列默认都是右对齐的。但 RTRIM 列就像串一样是左对齐的，而且还把 0 和 2 从 Zip 码的右边删掉了。该例还有一些特别之处。请回顾一下第 7 章中 RTRIM 的格式，如下所示：

```
RTRIM(string[,'set'])
```

从 string 中删除的 set 是括在单引号内的，但在下例中：

```
RTRIM(Zip, 20)
```

没有引号。为什么会这样呢？

11.1.1　数据类型的自动转换

Oracle 将会自动地将 Zip 和 20 这些数字转换为串，就像在它们前面有 TO_CHAR 函数一样。事实上，除个别明显的例外情况，Oracle 将自动地根据要使用的函数对数据进行数据类型转换。如果使用串函数，则 Oracle 会立即将 NUMBER 或 DATE 转化为串，然后该串函数就可以工作了。如果使用 DATE 函数，并且列和字面值是一个 DD-MON-YY 格式的串，则 Oracle 会把它们转换为 DATE 类型。如果使用的是一个算术函数并且列或字面值是一个字符串，则 Oracle 会首先将其转换为 NUMBER 然后再计算。

这种自动转换的方法是否永远起作用呢？答案是否定的。如果要使 Oracle 自动将一种数据类型转换为其他数据类型，则第一个数据类型必须“看上去”和要转换为的数据类型相似。基本的原则如下：

- 任何 NUMBER 或 DATE 都能转换为字符串。任何串函数都能用于 NUMBER 和 DATE 列。NUMBER 类型的字面值用于串函数时不需要用单引号括起来，DATE 类型的字面值也一样。

- 在仅包含 NUMBER、小数点或数左边的负号的情况下，CHAR 或 VARCHAR2 值可以转换为 NUMBER 类型的数据。
- CHAR 或 VARCHAR2 值能转换成 DATE 类型的数据，只要它的格式是默认格式(通常是 DD-MON-YY)即可。这几乎适用于所有函数，但也有例外。它对 GREATEST 和 LEAST 函数来说不适用，这两个函数会把值作为一个串来处理，而对于 BETWEEN 函数来说，只在 BETWEEN 后左边的列是一个 DATE 类型的值时才可以。否则，必须以恰当的格式使用 TO_DATE。

由于这些原则可能让人有些迷惑不解，因此为了正确地使用 TO_DATE 和其他几个转换函数来正确地处理值，下面用几个示例来阐述这些原则。下面是几个随机挑选出来的作用于 NUMBER 和 DATE 的串函数的结果：

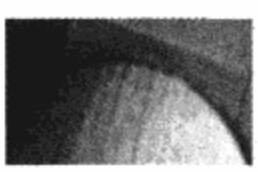
```
select INITCAP(LOWER(SysDate)) from DUAL;

INITCAP(LOWER(SYSDATE))
-----------------------
28-Feb-08
```

请注意，INITCAP 函数把“feb”的第一个字母换成了大写，虽然“FEB”位于串“28-FEB-08”的中间。尽管这是第一次说明它，不过要说明的是，这是 INITCAP 函数的并不仅限于日期函数的一个功能。它起作用，是因为下面的语句起作用：

```
select INITCAP('this-is_a.test,of:punctuation;for+initcap')
  from DUAL;

INITCAP('THIS-IS_A.TEST,OF:PUNCTUATION;FO
-----------------------------------------
This-Is_A.Test,Of:Punctuation;For+Initcap
```

INITCAP 把每个单词的第一个字母都转换为大写了。它根据前面的非字母字符来确定一个单词。还可以用串函数剪切和粘贴日期，就像它们是串一样：

```
select SUBSTR(SysDate,4,3) from DUAL;

SUB
---
FEB
```



注意：
也可以使用 TO_CHAR 或 EXTRACT 函数从一个日期中返回月份值。

下面用 9 填充 DATE 的左边使其总长度为 20：

```
select LPAD(SysDate,20,'9') from DUAL;

LPAD(SYSDATE,20,'9')
```

```
--------------------
9999999999922-FEB-08
```

LPAD 或其他串函数都可以作用于 NUMBER，无论是字面值(如这里所示)还是列：

```
select SysDate / 2 from DUAL;
           *
ERROR at line 1:
ORA-00932: inconsistent datatypes: expected NUMBER got DATE
```

最后，NUMBER 型的数据决不会自动转换为 DATE 型数据的，因为一个纯数字不具备 DATE 数据的默认格式，即不具备 DD-MON-YY 格式：

```
select NEXT_DAY(022808,'FRIDAY') from DUAL;
       *

ERROR at line 1:
ORA-00932: inconsistent data types: expected DATE got NUMBER
```

如果需要在日期函数中用 NUMBER 型数据，则必须使用 TO_DATE 函数。

11.1.2 关于自动转换的注意事项

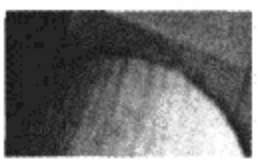
关于允许 SQL 自动进行数据类型转换是否是个好主意有两种看法。一方面，这种做法极大地简化并减少了使用 select 语句时所必需的函数。另一方面，如果关于列中内容的假定有错(比如认为某个特定的字符列里面总是包含数字，也就是说可以将它用于计算)，那么有时查询将会终止，Oracle 将产生错误，并且必须花时间找出错误。此外，阅读 select 语句的人会被对数字和字符使用不恰当的函数这一假象所迷惑。使用 TO_NUMBER 简化了这些问题，即使列使用的是 VARCHAR2 数据类型，但预期的总是数值型值。如果 TO_NUMBER 函数作用于列中的非数值型值，那么仍然会产生错误。

隐式转换的最大问题是转换时列索引无效。如果有一个 VARCHAR2 类型的列，如 ID 或 Order_Num，然后您创建了一个 PL/SQL 过程来访问此列数据，并在过程中传递一个数，那么 ID 或 Order_Num 上建立的索引将不起作用，因此性能会受到影响。

一个简单的经验规则就是最好在相对安全的地方使用函数，比如对数值使用串处理函数，而不对串使用算术运算函数。为了保证您自己以及其他使用您的程序的任何人的利益，最好在 select 语句的旁边标出数据类型的自动转换。

11.2 特殊的转换函数

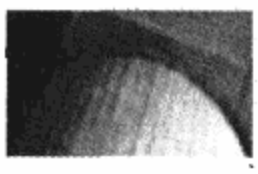
如表 11-1 所示，Oracle 有几个特殊的转换函数。如果打算利用 SQL*Plus 和 Oracle 简单地生成报表，则可能不需要这些函数。但是，如果打算使用 SQL*Plus 来更新数据库，如果期望构建 Oracle 应用程序，或者，如果您正使用 National Language Support，下面的信息就是有价值的。这些函数可按名字在附录 A 中找到。



注意：
CAST 函数用于嵌套表和可变数组，详细内容请参考第 39 章。DECODE 函数将在第 16 章介绍。

转换函数一般接收单个值作为输入，返回一个转换过的值作为输出。例如，BIN_TO_NUM 函数将二进制值转换为十进制值。它的输入值是一个二进制的数字列表，用逗号分隔，作为一个输入串处理：

```
select BIN_TO_NUM(1,1,0,1) from DUAL;

BIN_TO_NUM(1,1,0,1)
-------------------
                 13

select BIN_TO_NUM(1,1,1,0) from DUAL;

BIN_TO_NUM(1,1,1,0)
-------------------
                 14
```



注意：

可以用 TO_BINARY_DOUBLE 和 TO_BINARY_FLOAT 函数分别将值转换为双精度浮点数和单精度浮点数。

当使用闪回操作时(参考第 29~30 章)，可以通过 SCN_TO_TIMESTAMP 函数将系统改变号(SCN)转换为时间戳值；TIMESTAMP_TO_SCN 返回一个特定时间戳的 SCN 值。

11.3　变换函数

虽然从某种意义上说，任何改变其对象的函数都可以叫做变换函数，但有两个不寻常的函数可以用不同的方法根据输入控制输出，而不是简单地变换它。这两个函数就是 TRANSLATE 和 DECODE。

11.3.1　TRANSLATE

TRANSLATE 是一个简单的函数，它在字符串中进行逐字符的替换。其格式如下所示：

```
TRANSLATE(string,if,then)
```

TRANSLATE 查看 string 中的每个字符，然后检查 if 以确定该字符是否存在。如果存在，就在 if 中标出找到字符的位置，然后查看 then 中相同的位置。TRANSLATE 将用 then 中该位置的字符替换 string 中相应的字符。通常这个函数写为单独的一行，如下所示：

```
select TRANSLATE(7671234,234567890,'BCDEFGHIJ')
  from DUAL;

TRANSLA
-------
GFG1BCD
```

但是把它写成两行通常会更好理解(当然，SQL 不会介意将代码写成一行还是多行)：

```
select TRANSLATE(7671234,234567890,
                                 'BCDEFGHIJ')
  from DUAL;

TRANSLA
-------
GFG1BCD
```

当 TRANSLATE 在 string 中看到 7 时，它在 if 中寻找 7，并将其转换成 then 中同样位置的字符(在本例中，是大写的 G)。如果这个字符不在 if 中，则不进行转换(请观察 TRANSLATE 对 1 所做的工作)。

虽然从技术上来说，TRANSLATE 是一个串函数，但是正如您所见，它会自动进行数据转换并作用于串和数字的混合形式。下面是一个非常简单的代码加密程序示例，其中字母表中的每个字符都移动了一个位置。多年以前，间谍用这样的字符替换方法在发送消息前对其进行加密。接收者只需颠倒这个过程即可。您还记得电影《2001 太空漫游》中那个讲话平和的计算机 HAL 吗？如果用 TRANSLATE 在字母表中对 HAL 进行一个字符的位移，则将得到：

```
select TRANSLATE('HAL','ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                                 'BCDEFGHIJKLMNOPQRSTUVWXYZA') AS Who
  from DUAL;

WHO
---
IBM
```

关于用正则表达式处理字符串的详细内容，请参考 8.5 节关于 REGEXP_REPLACE 函数的讨论。

11.3.2 DECODE

如果 TRANSLATE 是逐字符地替换，那么可以认为 DECODE 是逐值的替换，DECODE 会为某个字段中的每个值在一系列的 if/then 测试中查找匹配值。DECODE 是一个功能很强的函数，它的用途很广。第 16 章会详细介绍 DECODE 和 CASE 的高级应用。

DECODE 的格式如下所示：

```
DECODE(value,if1,then1,if2,then2,if3,then3,... ,else)
```

虽然这里只列出了 3 个 if/then 的组合，但实际上它是没有限制的。为理解这个函数是怎样工作，可回想一下前面章节中见过的 NEWSPAPER 表：

```
select * from NEWSPAPER;

FEATURE           S          PAGE
---------------   -   ----------
National News     A             1
Sports            D             1
Editorials        A            12
Business          E             1
```

```
Weather          C          2
Television       B          7
Births           F          7
Classified       F          8
Modern Life      B          1
Comics           C          4
Movies           B          4
Bridge           B          2
Obituaries       F          6
Doctor Is In     F          6
```

下面的示例对页码进行解码。如果为 1，则将其替换为 Front Page。如果是别的页码，那么把词 Turn to 连接到的前面。这个示例说明 else 可以是一个函数、一个字面值或者其他列。

```
select  Feature, Section,
          DECODE(Page,'1','Front Page','Turn to '||Page)
  from NEWSPAPER;
```

```
FEATURE          S    DECODE(PAGE,'1','FRONTPAGE','TURNTO'||PAGE)
---------------  -    -------------------------------------------
National News    A    Front Page
Sports           D    Front Page
Editorials       A    Turn to 12
Business         E    Front Page
Weather          C    Turn to 2
Television       B    Turn to 7
Births           F    Turn to 7
Classified       F    Turn to 8
Modern Life      B    Front Page
Comics           C    Turn to 4
Movies           B    Turn to 4
Bridge           B    Turn to 2
Obituaries       F    Turn to 6
Doctor Is In     F    Turn to 6
```

在 if 和 then 子句的列表中，对数据类型有一些限制，这些限制将在第 16 章中介绍。

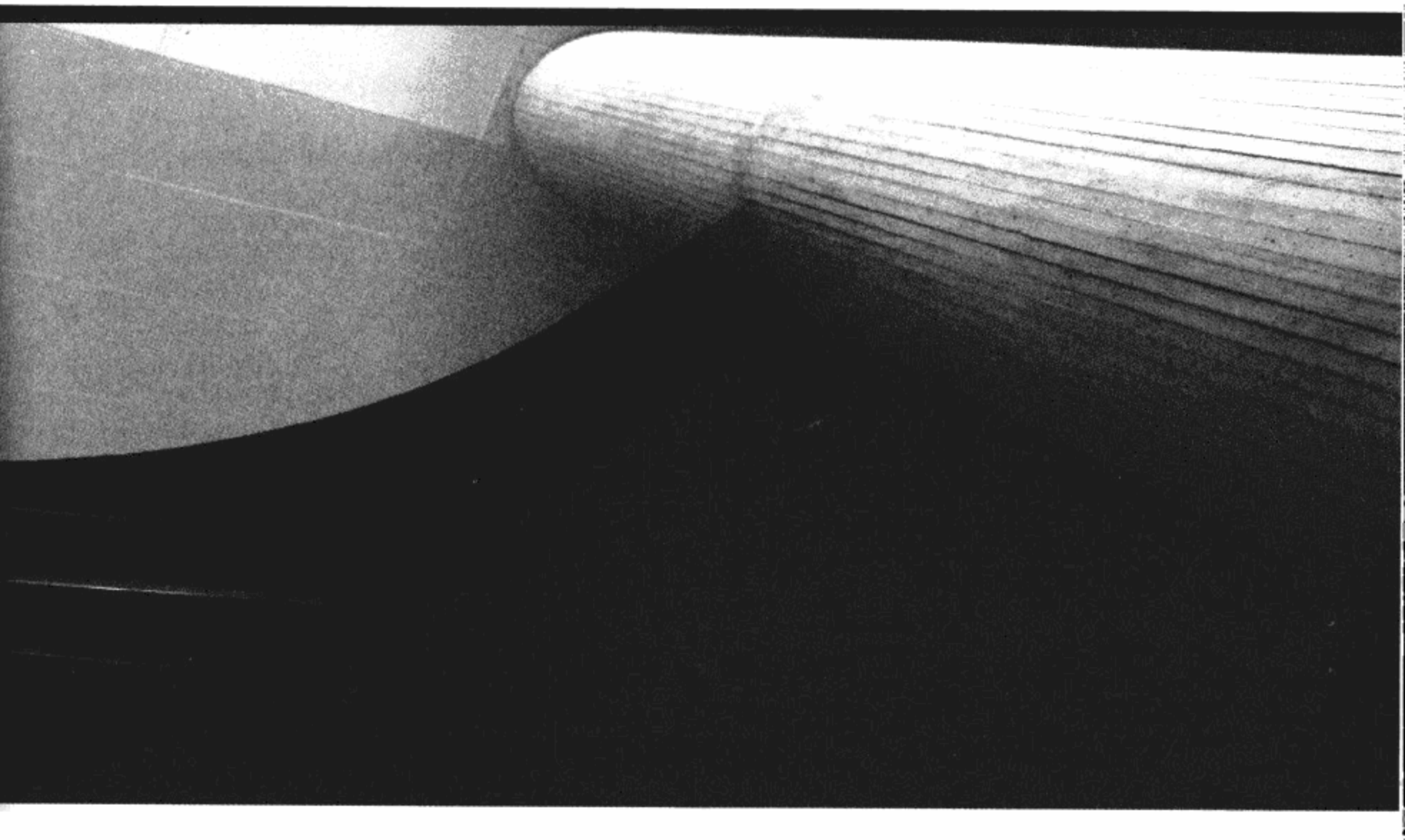
11.4　小结

虽然 Oracle 的大部分函数都是为特定的数据类型而设计的，如 CHAR、VARCHAR2、NUMBER 和 DATE，但它们也可以使用其他数据类型。这些函数通过数据类型的自动转换来使用其他数据类型。虽然只要被转换的数据看起来“像”这个函数所需的数据类型，它们就能进行处理，但也有几个例外，希望将来能解决。

字符函数能转换任何 NUMBER 型数据或者 DATE 型数据。NUMBER 函数能转换只包含数字 0~9、一个小数点或左边有一个负号的 CHAR 和 VARCHAR2 型数据类型。NUMBER 函数不能转换 DATE 型数据。虽然 DATE 函数能转换格式是 DD-MON-YY 的字符串，但它们

不能转换 NUMBER 型数据。

TRANSLATE 和 DECODE 这两个函数，将从根本上改变它们所处理的数据。TRANSLATE 将根据所指定的模式进行字符替换，而 DECODE 将根据所指定的模式进行值的替换。



第12章

分组函数

目前为止，本书已经介绍了 SQL 如何从数据库表中选择(select)信息行，where 子句如何限定返回满足所定义条件的行的数量，以及如何用 order by 将返回的行按升序或降序进行排列。还介绍了如何用字符、NUMBER 和 DATE 函数来修改列的值，以及分组函数如何得出整组行的某些内容。

除分组函数以外，还有两个分组子句：having 和 group by。它们与 where 和 order by 子句作用相似，只不过 having 和 group 是对组而不是单独的行起作用。使用这两个子句能对数据进行更深入的观察。

12.1 group by 和 having 的用法

如果希望统计书架上的书，这些书按书的类型分类，则可编写如下的查询：

```
select CategoryName, COUNT(*)
  from BOOKSHELF
 group by CategoryName;
```

Oracle 将响应如下内容：

```
CATEGORYNAME                 COUNT(*)
--------------------     ----------
ADULTFIC                          6
ADULTNF                          10
ADULTREF                          6
CHILDRENFIC                       5
CHILDRENNF                        1
CHILDRENPIC                       3
```

注意在 select 子句中混合使用了一个列名 CategoryName 和分组函数 COUNT。这样混合使用是合适的，因为在 group by 子句中引用了 CategoryName。否则，将会出现第 9 章中遇到的难以理解的错误消息：

```
select CategoryName, COUNT(*) from BOOKSHELF;

select CategoryName, COUNT(*) from BOOKSHELF
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

产生这种结果的原因是分组函数(如 SUM 和 COUNT)是用来说明表中的一组行而不是单个行的信息的。在 group by 子句中使用 CategoryName 避免了这个错误，它强制 COUNT 函数统计每个 CategoryName 列内的行。

Having 子句的功能和 where 子句很相似，只是它的逻辑仅和分组函数的结果有关，而不与单个行的列或表达式有关，单个行仍然可被 where 子句选择。上例中的行可进一步限制为只选择多于 5 本书的那些类型：

```
select CategoryName, COUNT(*)
  from BOOKSHELF
 group by CategoryName
having COUNT(*) > 5;

CATEGORYNAME                 COUNT(*)
--------------------     ----------
ADULTFIC                          6
ADULTNF                          10
ADULTREF                          6
```

为了确定种类的平均等级，可使用 AVG 函数，如下所示：

```
select CategoryName, COUNT(*), AVG(Rating)
  from BOOKSHELF
 group by CategoryName;

CATEGORYNAME              COUNT(*)   AVG(RATING)
--------------------    ----------  -----------
ADULTFIC                         6   3.66666667
ADULTNF                         10          4.2
ADULTREF                         6   3.16666667
CHILDRENFIC                      5          2.8
CHILDRENNF                       1            3
CHILDRENPIC                      3            1
```

虽然 Rating 是一个字符列，定义为 VARCHAR2 类型，但由于它包含数字类型的值，因此 Oracle 在其上使用数值函数(参考第 11 章)。总的平均等级是什么呢？

```
select AVG(Rating) from BOOKSHELF;

AVG(RATING)
-----------
3.32258065
```

在这个示例中没有 group by 子句，因为 BOOKSHELF 表中整组行是作为一个组处理的。现在可将此结果作为一个更大的查询的一部分，这个查询是：哪些种类的平均等级大于所有书的平均等级？

```
select CategoryName, COUNT(*), AVG(Rating)
  from BOOKSHELF
 group by CategoryName
having AVG(Rating) >
      (select AVG(Rating) from BOOKSHELF);

CATEGORYNAME              COUNT(*)     AVG(RATING)
--------------------  ----------    -----------
ADULTFIC                         6      3.66666667
ADULTNF                         10             4.2
```

查看前面的程序清单，这个结果是正确的，只有两个组的平均等级大于总的平均等级。

虽然此结果是按 CategoryName 列排序的，但 group by 的目的不是产生想要的顺序，而是将“类似”的行排在一起。次序是 group by 的一个副产品，group by 并不用来改变排列顺序。

12.1.1　添加一个 order by

改变显示顺序的方法是在 having 子句后再添加一个 order by 子句，如下所示：

```
order by CategoryName desc
```

这将颠倒列表的顺序：

```
select CategoryName, COUNT(*)
  from BOOKSHELF
 group by CategoryName
 order by CategoryName desc;

CATEGORYNAME              COUNT(*)
--------------------  ----------
CHILDRENPIC                    3
CHILDRENNF                     1
CHILDRENFIC                    5
ADULTREF                       6
ADULTNF                       10
ADULTFIC                       6
```

或者可以使用以下代码：

```
order by COUNT(*) desc
```

它产生以下结果：

```
CATEGORYNAME              COUNT(*)
--------------------  ----------
ADULTNF                       10
ADULTFIC                       6
ADULTREF                       6
CHILDRENFIC                    5
CHILDRENPIC                    3
CHILDRENNF                     1
```

虽然可以用列别名作为 order by 子句的一部分，但不能将列别名作为 having 子句的一部分。给 COUNT(*)一个别名 Counter，并试图在这个查询中使用 having Counter > 1 作为子句将会产生“无效列名”错误：

```
select CategoryName, COUNT(*) as Counter
  from BOOKSHELF
 group by CategoryName
having Counter > 1
 order by COUNT(*) desc;

having Counter > 1
       *
ERROR at line 4:
ORA-00904: "COUNTER": invalid identifier
```

12.1.2 执行顺序

上述查询语句中有几个竞争的子句。下面列出 Oracle 用来执行这些子句的规则以及执行的顺序：

(1) 根据 where 子句选择行。

(2) 根据 group by 子句将这些行分组。

(3) 为每一组计算分组函数的结果。

(4) 根据 having 子句选择和排除组。

(5) 根据 order by 子句中分组函数的结果对组进行排序。order by 子句必须使用分组函数，或者使用在 group by 子句中指定的列。

执行顺序很重要，因为它对查询性能具有直接的影响。一般来说，通过 where 子句排除的记录越多，查询执行的速度就越快。这种性能上的好处是由于它减少了在 group by 操作中必须处理的行数。

如果编写一条利用 having 子句排除组的查询语句，那么应该检查是否可以用 where 子句重写 having 条件。在很多情况下，不能进行这种重写。只有在用 having 子句根据分组列来排除组时才能进行这种重写。

例如，如果有下面的查询：

```
select CategoryName, COUNT(*), AVG(Rating)
  from BOOKSHELF
 where Rating > 1
 group by CategoryName
having CategoryName like 'A%'
 order by COUNT(*) desc;

CATEGORYNAME            COUNT(*)    AVG(RATING)
-------------------- ----------    -----------
ADULTNF                     10            4.2
ADULTFIC                     6     3.66666667
ADULTREF                     6     3.16666667
```

那么执行顺序为：

(1) 基于 where Rating > 1 排除行。

(2) 基于 group by CategoryName 分组剩下的行。

(3) 为每个 CategoryName 计算 COUNT(*)。

(4) 基于 having CategoryName like 'A%'排除组。

(5) 对剩下的组进行排序。

如果第(4)步中排除的组能在第(1)步中作为行被排除掉，则这个查询将执行得更快。如果这些行在第(1)步中被排除，则第(2)步中分组的行会少一些，第(3)步所执行的计算会更少，第(4)步将没有组被排除了。因此这些步骤的执行速度就会越来越快。

由于本例中的 having 条件不是基于计算出来的列而得出的，因此很容易更改为 where 条件：

```
select CategoryName, COUNT(*), AVG(Rating)
  from BOOKSHELF
 where Rating > 1
   and CategoryName like 'A%'
```

```
group by CategoryName
order by COUNT(*) desc;
```

在修改过的这个版本中，需要被分组的行比较少，结果是提高了性能。随着表中行数的增加，尽早排除行将会很大地提高性能。

这个优化示例似乎微不足道，因为表中只有几行。但即使是这样一个小小的查询，在产品应用程序中也会产生很大的影响。可以举出很多这样的产品应用程序的例子，由于大量执行看起来很小的查询，从而使得它们的性能受到了很大的影响。当这些小查询每天执行成千上万次的时候，它们就成为数据库中最消耗资源的查询。当为应用程序规划 SQL 访问路径时，即使是一些小查询，也要优化处理。

12.2 分组视图

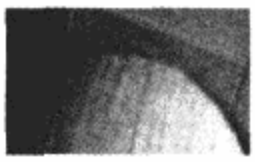
在第 5 章中，INVASION 视图是在 Delphi 中为 Oracle 创建的，它把 WEATHER 表和 LOCATION 表连接了起来。虽然这个视图本身似乎就是一个表，具有列和行，但它的每个行包含的列实际是来自于两个单独的表。

创建视图的过程同样可用于分组。不同的是每一行将包含一组行的信息，这是一种分类汇总表。请考虑下面这个分组查询：

```
select CategoryName, COUNT(*)
  from BOOKSHELF
 group by CategoryName;
```

可根据这个查询创建一个视图，然后查询此视图：

```
create or replace view CATEGORY_COUNT as
select CategoryName, COUNT(*) AS Counter
  from BOOKSHELF
 group by CategoryName;

desc CATEGORY_COUNT

Name                                      Null?    Type
----------------------------------------- -------- ------------
CATEGORYNAME                                       VARCHAR2(20)
COUNTER                                            NUMBER

select * from CATEGORY_COUNT;

CATEGORYNAME         COUNTER
-------------------- ----------
ADULTFIC                      6
ADULTNF                      10
ADULTREF                      6
CHILDRENFIC                   5
CHILDRENNF                    1
CHILDRENPIC                   3
```



注意：
因为 COUNT(*)列是一个函数，所以在使用查询作为视图的基础时，必须给它一个列别名(在本例中为 Counter)。

12.3　用别名重命名列

请注意上例 select 子句中的名字 Counter。AS Counter 子句重命名了它前面的列。这些新名字称为别名(Alias)，可用来隐藏基础列的真名(因为这些基础列具有函数，所以显得比较复杂)。

在查询此视图时，可以(而且必须)使用新的列名：

```
select CategoryName,Counter from CATEGORY_COUNT;
```

Counter 被称为列别名(引用一个列时使用的另一个名字)。在视图的描述以及在查询中，没有执行分组函数，只使用了 Counter 列名。就好像视图 CATEGORY_COUNT 是一个具有每月合计的真实的表一样。为什么呢？

Oracle 会自动地接收没有引号的单个词，并用它重命名它前面的列。在重命名列时，Oracle 会强制别名为大写，而不管它是以大写或小写输入的。比较 create view 和 describe 命令中的列名，可看到这一点。创建视图时，绝对不要给列的别名加双引号。creat view 语句中的别名始终是没有引号的。这样使它们得以用大写形式存储，从而便于 Oracle 找到它们。请参考“视图创建中的别名”部分中关于别名的提示。

视图创建中的别名

在内部，Oracle 将所有的列名和表名存储为大写形式。这些列名和表名在数据字典中就是以大写形式存储的，而且 Oracle 也希望它们是大写。在输入别名以创建一个视图时，别名不应该用引号括起来。给别名加双引号会使 Oracle 内部存储的列名变为大小写混合的形式。如果这样做，除非在所有查询中给列名加上双引号，否则执行 select 语句时 Oracle 将找不到列名。

在创建视图的别名时绝对不要使用双引号。

现在，在一个视图中得到了 Category 的统计。可利用 BOOKCOUNT 作为视图名和 COUNT(*)的列别名创建整个书架的合计视图：

```
create or replace view BOOKCOUNT as
  select COUNT(*) BOOKCOUNT
    from BOOKSHELF;

  View created.
```

如果查询此视图，则将发现它只有一个记录：

```
select BOOKCOUNT
  from BOOKCOUNT;
```

```
BOOKCOUNT
---------
        31
```

因为添加了新的行并提交给 BOOKSHELF 表，所以 BOOKCOUNT 视图和 CATEGORY_COUNT 视图将会反映统计中的这种改变。

12.4 分组视图的功能

现在，您看到关系数据库的强大功能了。我们已经创建了两个视图：一个视图按照书的 Category 进行计数，另一个视图显示整个表中书的数量。现在可以将这些视图连接起来(就像第 5 章中将表连接起来一样)，以反映一些以前无法看出的信息。例如，可以计算每种类别的书所占的百分比：

```
select CategoryName, Counter, (Counter/BookCount)*100 as Percent
  from CATEGORY_COUNT, BOOKCOUNT
 order by CategoryName;

CATEGORYNAME              COUNTER      PERCENT
--------------------  ----------   ----------
ADULTFIC                       6   19.3548387
ADULTNF                       10   32.2580645
ADULTREF                       6   19.3548387
CHILDRENFIC                    5   16.1290323
CHILDRENNF                     1   3.22580645
CHILDRENPIC                    3   9.67741935
```

虽然这个查询的 from 子句列出了两个视图，但并没有将它们在 where 子句中连接起来。为什么不这样做呢？在这种特殊的情况下，不需要使用 where 子句，因为视图 BOOKCOUNT 只返回一行(如前面的程序清单所示)。BOOKCOUNT 视图中的这一行与 CATEGORY_COUNT 视图中的每行连接，为 CATEGORY_COUNT 视图中的每行产生一行输出。虽然相同的结果也可以通过直接连接 BOOKSHELF 表和 BOOKCOUNT 视图得到，但正如所见，此查询更为复杂，更难理解，因为组的数目更多了，所以查询变得更复杂了：

```
select CategoryName, COUNT(*),
       (COUNT(*)/MAX(BookCount))*100 as Percent
  from BOOKSHELF, BOOKCOUNT
 group by CategoryName
order by CategoryName;
```

请注意百分数的计算：

```
    (COUNT(*)/MAX(BookCount))*100 as Percent
```

由于这个结果为分组函数的一部分，因此每个值必须分组。因此，如下的计算将会失败：

```
    (COUNT(*) /BookCount))*100 as Percent
```

因为 BookCount 不分组。由于 BOOKCOUNT 视图中只有一行，因此可对它执行 MAX 函数，返回该行，根据它进行分组。

为了创建以比较一个分组的行与另一个分组的行的查询，至少有一个分组必须为视图，或者为在查询的 from 子句中创建的“内联视图”。不过，除了这个技术限制外，用视图进行查询确实更简单，也更容易理解。请比较最后的这两个示例，它们的差异很明显。视图隐藏了复杂性。

为了使用内联视图方法，只要将视图的文本放入 from 子句，并给予其相应的列别名即可：

```
select CategoryName, Counter, (Counter/BookCount)*100 as Percent
  from CATEGORY_COUNT,
       (select COUNT(*) as BookCount from BOOKSHELF)
 order by CategoryName;
```

在这个示例中，BOOKCOUNT 视图已经从 from 子句中删除了，并被替换为其基本查询。在该查询中，给出了 BookCount 别名，提供针对 BOOKSHELF 表执行的 COUNT(*)的结果。在主查询中，BookCount 别名用作计算的一部分。使用这种方法编写代码，就不需要创建 BOOKCOUNT 视图了。在同一查询中使用多个分组层次时要小心。创建视图一般有助于简化代码的编写和维护。

12.4.1　在视图中使用 order by

虽然从理论上说，不能将 order by 子句存储在视图中，但可在查询视图时使用 order by 子句。Oracle 支持视图内部的 order by 子句，如下所示：

```
create view BOOKSHELF_SORTED
  as select * from BOOKSHELF
  order by Title;
```

把数据存储在视图中会简化应用程序的开发。例如，如果代码单步调试一组记录，则把这些记录预先排序可能会使处理和错误检查更为简单。在应用程序开发中，您将会知道返回的数据总是以某种方式进行了排序。下面的查询选择 Title 值，利用 RowNum 伪列限制输出为 9 个记录：

```
select Title from BOOKSHELF_SORTED
 where Rownum < 10;

TITLE
-------------------------------------
ANNE OF GREEN GABLES
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EITHER/OR
EMMA WHO SAVED MY LIFE
GOOD DOG, CARL
GOSPEL
```

```
HARRY POTTER AND THE GOBLET OF FIRE
```

BOOKSHELF_SORTED 视图所做的工作不仅是从 BOOKSHELF 表中进行查询，还能执行排序操作，以特定的顺序返回行。如果您不需要以调整后的顺序返回这些行，就没有充分利用数据库给应用程序带来的好处。

视图还提供了任意使用多个不同字符、NUMBER 和 DATE 数据类型的强大功能，不用担心诸如月份以字母顺序显示等问题。

12.4.2 having 子句中的逻辑

在 having 子句中，分组函数的选择以及它所作用的列与 select 子句中的列或分组函数可能毫无关系：

```
select CategoryName, COUNT(*),
       (COUNT(*)/MAX(BookCount))*100 as Percent
  from BOOKSHELF, BOOKCOUNT
 group by CategoryName
having Avg(Rating) > 4
 order by CategoryName;

CATEGORYNAME          COUNT(*)    PERCENT
-------------------- ---------- ----------
ADULTNF                      10  32.2580645
```

其中，having 子句仅选择平均等级大于 4 的那些种类(group by 把所有的行收集到按 CategoryName 划分的组中)。所有其他的组都被删除。对于满足条件的分组，计算合计的百分数。

having 子句对于确定表中特定的列中哪些行具有重复值非常有效。例如，如果试图在表的某个列(或某组列)上创建一个新的唯一索引，并且该索引的创建由于数据不唯一而产生错误，那么可以很容易地找到出问题的行。

首先，选择希望其唯一的列，后面是 COUNT(*)列。在希望其唯一的列上执行 group by，再使用 having 子句只返回那些 COUNT(*)>1 的分组。返回的记录将是重复的。下面的查询说明了对表 AUTHOR 的 AuthorName 列执行的这种检查：

```
select AuthorName, COUNT(*)
  from AUTHOR
 group by AuthorName
having COUNT(*)>1
 order by AuthorName;

no rows selected
```

哪些书有多个作者？从 BOOKSHEFL_AUTHOR 表中选择其分组(按 Title)有多个成员的书名：

```
column Title format a40

select Title, COUNT(*)
  from BOOKSHELF_AUTHOR
group by Title
```

```
having COUNT(*)>1;

TITLE                                        COUNT(*)
----------------------------------------   ----------
COMPLETE POEMS OF JOHN KEATS                        2
JOURNALS OF LEWIS AND CLARK                         4
KIERKEGAARD ANTHOLOGY                               2
RUNAWAY BUNNY                                       2
```

那10个作者分别是谁？可基于此查询创建一个视图，或尝试用内联视图：

```
column Title format a40
column AuthorName format a30

select Title, AuthorName
from BOOKSHELF_AUTHOR,
  (select Title as GroupedTitle, COUNT(*) as TitleCounter
    from BOOKSHELF_AUTHOR
  group by Title
  having COUNT(*) > 1)
where Title = GroupedTitle
order by Title, AuthorName;

TITLE                              AUTHORNAME
-------------------------------    --------------------
COMPLETE POEMS OF JOHN KEATS       JOHN BARNARD
COMPLETE POEMS OF JOHN KEATS       JOHN KEATS
JOURNALS OF LEWIS AND CLARK        BERNARD DE VOTO
JOURNALS OF LEWIS AND CLARK        MERIWETHER LEWIS
JOURNALS OF LEWIS AND CLARK        STEPHEN AMBROSE
JOURNALS OF LEWIS AND CLARK        WILLIAM CLARK
KIERKEGAARD ANTHOLOGY              ROBERT BRETALL
KIERKEGAARD ANTHOLOGY              SOREN KIERKEGAARD
RUNAWAY BUNNY                      CLEMENT HURD
RUNAWAY BUNNY                      MARGARET WISE BROWN
```

虽然这个查询看上去可能有点复杂(使用视图会使它读起来简单一些)，但它是基于本章介绍的概念：内联视图执行group by函数，并使用having子句只返回那些具有多个作者的书名。然后使用这些书名作为对BOOKSHELF_AUTHOR表执行查询的基础。在单个查询中，查询BOOKSHELF_AUTHOR表，以得出分组的数据和个别的行数据。

12.4.3 对列和分组函数进行排序

order by子句在where、group by和having子句之后执行。它能够使用分组函数或来自group by的列，或者两者同时使用。如果它使用一个分组函数，此函数在分组上执行，则由order by按顺序排列这个函数的结果。如果order by使用来自group by的一个列，则它会基于该列对返回的行进行排序。分组函数和单一列(只要这些列在group by内)可在order by内组合使用。

在order by子句中，可以指定一个分组函数和它所涉及的列，即使它们与select、group by或having子句中的分组函数或列没有任何关系也可以。另一方面，如果在order by子句中指

定了不是分组函数一部分的一个列，则它必须在 group by 子句中。现在考虑最后一个示例，并修改其 order by 子句：

```
order by TitleCounter desc,Title,AuthorName
```

现在，书名和作者先根据作者的数目进行排序(作者数多的排在最前面)，然后再按 Title 和 AuthorName 进行排序：

```
TITLE                                    AUTHORNAME
---------------------------------------- --------------------
JOURNALS OF LEWIS AND CLARK              BERNARD DE VOTO
JOURNALS OF LEWIS AND CLARK              MERIWETHER LEWIS
JOURNALS OF LEWIS AND CLARK              STEPHEN AMBROSE
JOURNALS OF LEWIS AND CLARK              WILLIAM CLARK
COMPLETE POEMS OF JOHN KEATS             JOHN BARNARD
COMPLETE POEMS OF JOHN KEATS             JOHN KEATS
KIERKEGAARD ANTHOLOGY                    ROBERT BRETALL
KIERKEGAARD ANTHOLOGY                    SOREN KIERKEGAARD
RUNAWAY BUNNY                            CLEMENT HURD
RUNAWAY BUNNY                            MARGARET WISE BROWN
```

12.4.4 连接列

正如第 5 章所述，连接两个表时要求它们具有由一个相同列定义的关系。对于视图连接或者表和视图的连接也是如此。唯一的例外就是当其中一个表或视图仅有唯一一行时不要求这样，如 BOOKCOUNT 表。在这种情况下，SQL 把这唯一的一行连接到另一个表或视图的每一行，并且在查询的 where 子句中不需要引用连接列。

如果要连接两个表，每一个表都具有多行，并且没有在 where 子句中指定连接列，则将会产生所谓的笛卡尔积(Cartesian Product)，这个积一般会是一个很大的结果，即一个表的每一行都与另一个表的每一行连接。一个 80 行的小表用这种方式连接到一个 100 行的小表上，将会产生一个 8000 行的表，这些行中没几行是有意义的。

12.5 更多分组可能性

除了本章所介绍的操作外，还可以执行复杂的行分组——创建交叉报表，在数据中加入分层等。这些分组函数，以及相关的函数和子句(比如 connect by、ROLLUP、GROUPING 以及 CUBE)将在第 14 章介绍。

当您在应用程序开发环境中使用这些分组函数时，通常会发现使用视图会使复杂的查询变简单。您可以使用视图来表示行的逻辑分组，这种表示方法对最终用户编写报表很有帮助，同时保持底层表的结构不发生变化。数据以一种用户理解的格式来表示，不但对用户有益，而且可以保持数据库设计的完整性。

第 13 章

当一个查询依赖于另一个查询时

本章和第 14 章将介绍的概念比前面章节中介绍的概念要复杂一些，虽然其中很多概念在一般的运行查询和生成报表的过程中很少用到，但它们有时却很有用处。如果您在学习它们的过程中遇到困难，那么无论如何要坚持下去。因为这些方法在需要的时候很有用，您以后肯定会用到它们。

13.1 高级子查询

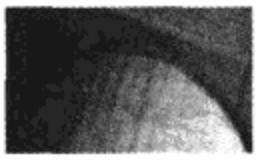
前面的章节提到过子查询(subquery)，它们是某些 select 语句中 where 子句的组成部分。

子查询也可以用在 insert、update 和 delete 语句之中。在 insert、update 和 delete 语句中的用法将在第 15 章中介绍。

通常，子查询可以提供查询的另一种方法。例如，假如希望知道哪种种类的书籍已经借出去了。下面的三表连接提供了这个信息：

```
select distinct C.ParentCategory, C.SubCategory
  from CATEGORY C, BOOKSHELF B, BOOKSHELF_CHECKOUT BC
 where C.CategoryName = B.CategoryName
   and B.Title = BC.Title;

PARENTCA  SUBCATEGORY
--------  --------------------
ADULT     FICTION
ADULT     NONFICTION
ADULT     REFERENCE
CHILDREN  FICTION
CHILDREN  PICTURE BOOK
```

3 个表的连接方法与两个表的连接方法相同。在 where 子句中，共同列设置为彼此相等，如上面的程序清单所示。要连接 3 个表，就必须把其中的两个表与第 3 个表连接起来。在本例中，CATEGORY 表连接到 BOOKSHELF 表，它们连接的结果再与 BOOKSHELF_CHECKOUT 表连接。distinct 子句告诉 Oracle 只返回 ParentCategory 和 SubCategory 的不同的合并结果。



注意：

并不是每一个表都与其他每个表相连接。事实上，表之间的连接数通常比被连接的表的数量少 1 。

一旦将这些表连接起来(如 where 子句的前两行所示)，就可以按父类别和子类别确定借出书的数目了。

13.1.1 相关子查询

还有执行多表连接的其他方法吗？回忆一下，where 子句可以包含一个子查询 select。子查询 select 可以嵌套，即一个子查询的 where 子句中还可以包含具有一个子查询的 where 子句，这个子查询的 where 子句中又可以包含另一个子查询，向下可以包含很多层子查询，直到满足需要为止。下面给出了 3 个 select 子查询，每个 select 子查询都是通过一个 where 子句连接到另一个子查询：

```
select distinct C.ParentCategory, C.SubCategory
  from CATEGORY C
 where CategoryName in
       (select CategoryName from BOOKSHELF
         where Title in
                    (select Title from BOOKSHELF_CHECKOUT)
       );

PARENTCA  SUBCATEGORY
--------  --------------------
```

```
ADULT     FICTION
ADULT     NONFICTION
ADULT     REFERENCE
CHILDREN  FICTION
CHILDREN  PICTURE BOOK
```

这些查询选择的任何种类包含已经借出的那些书籍。它通过请求其书名在 BOOKSHELF 表中、且借出记录在 BOOKSHELF_CHECKOUT 表中的书籍来完成这项工作。在子查询中，Oracle 假定列来自第一个 select 语句，这个 select 语句在其 where 子句中包含子查询。这称为嵌套子查询，因为对于主(外部)查询中每个 CategoryName 来说，CategoryName 在第二个 where

虽然乍一看好像很复杂，但细读一下此代码，会发现它很容易理解。从最里层的查询开始阅读：得到 Fred Fuller 已经借出的书籍的书名清单。根据这些的书名，进入 BOOKSHELF 表，得到指向这些书籍的不同的 CategoryName 值列表。再次进入 BOOKSHELF 表，得到那些种类中所有书籍的书名。根据这些书名，进入 BOOKSHELF_AUTHOR 表，生成作者清单。结果如下所示：

```
AUTHORNAME
----------------------
BERNARD DE VOTO
BERYL MARKHAM
DANIEL BOORSTIN
DAVID MCCULLOUGH
DIETRICH BONHOEFFER
G. B. TALBOT
JOHN ALLEN PAULOS
MERIWETHER LEWIS
STEPHEN AMBROSE
STEPHEN JAY GOULD
WILLIAM CLARK
```

Fred 要求推荐新作者，那么先排除他已经阅读过其作品的作者。为了知道 Fred 已经读过哪些作者的作品，只要查询 BOOKSHELF_CHECKOUT 表和 BOOKSHELF_AUTHOR 表即可：

```
select distinct AuthorName
  from BOOKSHELF_AUTHOR ba, BOOKSHELF_CHECKOUT bc
 where ba.Title = bc.Title
   and bc.Name = 'FRED FULLER';

AUTHORNAME
------------------------
DAVID MCCULLOUGH
```

现在，就可以从准备提供的清单中排除这些作者。在相应的查询中添加另外一条 and 子句就可完成这项工作了：

```
select distinct AuthorName from BOOKSHELF_AUTHOR
  where Title in
    (select Title from BOOKSHELF
     where CategoryName in
       (select distinct CategoryName from BOOKSHELF
        where Title in
            (select Title
               from BOOKSHELF_CHECKOUT bc
              where BC.Name = 'FRED FULLER')))
   and AuthorName not in
    (select AuthorName
       from BOOKSHELF_AUTHOR ba, BOOKSHELF_CHECKOUT bc
      where ba.Title = bc.Title
        and bc.Name = 'FRED FULLER');
```

```
AUTHORNAME
--------------------
BERNARD DE VOTO
BERYL MARKHAM
DANIEL BOORSTIN
DIETRICH BONHOEFFER
G. B. TALBOT
JOHN ALLEN PAULOS
MERIWETHER LEWIS
STEPHEN AMBROSE
STEPHEN JAY GOULD
WILLIAM CLARK
```

虽然 and 子句跟在子查询之后，但这个 and 子句是主查询的一部分。还要注意，在脚本中多个地方查询的某些表，这些查询都作为对表的单独访问。

13.1.3　EXISTS 及其相关子查询的使用

EXISTS 用来测试存在状态。它以子查询可能放置 IN 的方式放置，不同之处在于，它是对从一个查询返回的行的逻辑测试，而不是对行本身的逻辑测试。

书架上有多少作者写了多本书？

```
select AuthorName, COUNT(*)
    from BOOKSHELF_AUTHOR
  group by AuthorName
having COUNT(*) > 1;
```

```
AUTHORNAME                    COUNT(*)
----------------------------- ----------
DAVID MCCULLOUGH                       2
DIETRICH BONHOEFFER                    2
E. B. WHITE                            2
SOREN KIERKEGAARD                      2
STEPHEN JAY GOULD                      2
W. P. KINSELLA                         2
WILTON BARNHARDT                       2
```

然而想同时找到 AuthorName 和 Title 是行不通的，因为 COUNT(*)所必需的 group by 是建立在 BOOKSHELF_AUTHOU 表(AuthorName，Title)的主键之上。因为按照定义，每个主键只能唯一标识一行，对于该行的书名统计结果永远不会大于 1，所以 having 子句的测试结果总是假的，它找不到任何行：

```
select AuthorName, Title, COUNT(*)
    from BOOKSHELF_AUTHOR
 group by AuthorName, Title
having COUNT(*) > 1;

no rows selected.
```

EXISTS 提供了一种解决方法。对于外部查询中选择的每个 AuthorName，下面的子查询会询问在 BOOKSHELF_AUTHOR 表中是否存在 Title 数大于 1 的 AuthorName。如果对于某

个给定的名字，答案是存在，则 EXISTS 测试结果为真，外部查询选择一个 AuthorName 和 Title。作者名与第一个 BOOKSHELF_AUTHOR 表的别名 BA 相关联。

```
column AuthorName format a25
column Title format a30

select AuthorName, Title
  from BOOKSHELF_AUTHOR BA
where EXISTS
       (select 'x'
          from BOOKSHELF_AUTHOR BA2
         where BA.AuthorName = BA2.AuthorName
         group by BA2.AuthorName
        having COUNT(BA2.Title) > 1)
order by AuthorName, Title;

AUTHORNAME              TITLE
--------------------    ------------------------------
DAVID MCCULLOUGH        JOHN ADAMS
DAVID MCCULLOUGH        TRUMAN
DIETRICH BONHOEFFER     LETTERS AND PAPERS FROM PRISON
DIETRICH BONHOEFFER     THE COST OF DISCIPLESHIP
E. B. WHITE             CHARLOTTE'S WEB
E. B. WHITE             TRUMPET OF THE SWAN
SOREN KIERKEGAARD       EITHER/OR
SOREN KIERKEGAARD       KIERKEGAARD ANTHOLOGY
STEPHEN JAY GOULD       THE MISMEASURE OF MAN
STEPHEN JAY GOULD       WONDERFUL LIFE
W. P. KINSELLA          BOX SOCIALS
W. P. KINSELLA          SHOELESS JOE
WILTON BARNHARDT        EMMA WHO SAVED MY LIFE
WILTON BARNHARDT        GOSPEL
```

这两个查询是相关联的。请注意：子查询引用了 BA.AuthorName 列，即使该列位于外部查询中而不是在子查询中也可以。虽然在子查询内，不需要别名 BA2，但它可以使代码更容易维护。

可利用 IN 和列名上的测试构建相同的查询。此时不需要相关子查询：

```
select AuthorName, Title
   from BOOKSHELF_AUTHOR BA
  where AuthorName in
        (select AuthorName
           from BOOKSHELF_AUTHOR
         group by AuthorName
        having COUNT(Title) > 1)
order by AuthorName, Title;
```

13.2 外部连接

Oracle9i 对外部连接的语法做了很大的改动。在下面的示例中，将会看到 Oracle9i 的语

法和 Oracle9i 以前版本的语法。虽然 Oracle9i 仍然支持 Oracle9i 以前版本的语法，但现在不应该再使用它了。新的开发应该使用新的语法，新的语法符合 ANSI SQL 标准，而旧语法不符合此标准。这里讨论旧的语法是因为很多第三方工具还在使用它。

13.2.1　Oracle9i 以前版本中的外部连接语法

在 BOOKSHELF_CHECKOUT 表中记录的时间段内，哪些书籍被借出去了？

```
column Title format a40

select distinct Title
  from BOOKSHELF_CHECKOUT;

TITLE
----------------------------------------
ANNE OF GREEN GABLES
EITHER/OR
GOOD DOG, CARL
HARRY POTTER AND THE GOBLET OF FIRE
INNUMERACY
JOHN ADAMS
MIDNIGHT MAGIC
MY LEDGER
POLAR EXPRESS
THE DISCOVERERS
THE MISMEASURE OF MAN
THE SHIPPING NEWS
TO KILL A MOCKINGBIRD
TRUMAN
WEST WITH THE NIGHT
WONDERFUL LIFE
```

虽然这是一个正确的报表，但没有显示统计结果 0，即没有显示未借出的书籍。如果您想查看所有书籍及借出列表的清单，那么需要将 BOOKSHELF_CHECKOUT 表与 BOOKSHELF 表进行连接：

```
select distinct B.Title
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title;
```

但是，此查询只返回完全相同的记录，即 BOOKSHELF 表中仅仅那些已经借出的行记录能满足连接条件。为了列出其余的书籍，就要使用一种外部连接，告诉 Oracle 即使连接不匹配也返回一行。Oracle 完全支持两个版本的外部连接语法。在 Oracle9i 以前的版本，外部连接的语法在这种连接的旁边使用一个(+)，它将返回额外的行。在本例下，返回 BOOKSHELF_CHECKOUT 表。下面的查询显示了每本书能借出的最长时间：

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
         "Most Days Out"
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
 where BC.Title (+) = B.Title
 group by B.Title;
```

```
TITLE                                    Most Days Out
---------------------------------------- -------------
ANNE OF GREEN GABLES                                18
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EITHER/OR                                            8
EMMA WHO SAVED MY LIFE
GOOD DOG, CARL                                      14
GOSPEL
HARRY POTTER AND THE GOBLET OF FIRE                 11
INNUMERACY                                          21
JOHN ADAMS                                          28
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
MIDNIGHT MAGIC                                      14
MY LEDGER                                           16
POLAR EXPRESS                                       14
PREACHING TO HEAD AND HEART
RUNAWAY BUNNY
SHOELESS JOE
THE COST OF DISCIPLESHIP
THE DISCOVERERS                                     48
THE GOOD BOOK
THE MISMEASURE OF MAN                               31
THE SHIPPING NEWS                                   59
TO KILL A MOCKINGBIRD                               14
TRUMAN                                              19
TRUMPET OF THE SWAN
UNDER THE EYE OF THE CLOCK
WEST WITH THE NIGHT                                 48
WONDERFUL LIFE                                      31
```

上述程序返回 BOOLSHELF 表中的所有书名，即使那些不满足连接条件的也返回。如果显示 BOOKSHELF_CHECKOUT.Title 的值，则将看到这些值为 NULL。考虑(+)，它必须紧跟在较短的表的连接列之后，因为“任何时候添加 BC.Title 的一个额外的(NULL)行，对 B.Title 来说都不存在匹配”。

13.2.2 现在的外部连接语法

现在可以使用外部连接的 ANSI SQL 标准语法。在 from 子句中，可以让 Oracle 执行 left、right 和 full outer 连接。从 13.2.1 节的示例开始讨论：

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
           "Most Days Out"
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title (+) = B.Title
group by B.Title;
```

在本例的连接过程中，即使找不到任何匹配，也从 BOOKSHELF_CHECKOUT 表有返回

行。这条查询语句也可以重写成这样：

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
          "Most Days Out"
  from BOOKSHELF_CHECKOUT BC right outer join BOOKSHELF B
    on BC.Title = B.Title
group by B.Title;
```

注意作为外部连接语法一部分的 on 子句的使用情况。注意：

```
from BOOKSHELF_CHECKOUT BC right outer join BOOKSHELF B
```

等同于

```
from BOOKSHELF B left outer join BOOKSHELF_CHECKOUT BC
```

这里的 on 子句可以用 using 子句和表公用的列名来代替，不要用表名或表别名来限制此列名。

```
select Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
         "Most Days Out"
  from BOOKSHELF_CHECKOUT BC right outer join BOOKSHELF B
using (Title)
group by Title;
```

注意不能为 using 子句中列出的列指定表别名，即使在 group by 和 select 子句中也是如此。

正如旧的语法一样，作为外部连接驱动表的左右侧是不同的，执行 left outer join 不会返回所有的书名。

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
        "Most Days Out"
  from BOOKSHELF_CHECKOUT BC left outer join BOOKSHELF B

    on BC.Title = B.Title
  group by B.Title;
```

```
TITLE                                    Most Days Out
---------------------------------------  -------------
ANNE OF GREEN GABLES                                18
EITHER/OR                                            8
GOOD DOG, CARL                                      14
HARRY POTTER AND THE GOBLET OF FIRE                 11
INNUMERACY                                          21
JOHN ADAMS                                          28
MIDNIGHT MAGIC                                      14
MY LEDGER                                           16
POLAR EXPRESS                                       14
THE DISCOVERERS                                     48
THE MISMEASURE OF MAN                               31
THE SHIPPING NEWS                                   59
TO KILL A MOCKINGBIRD                               14
TRUMAN                                              19
WEST WITH THE NIGHT                                 48
```

```
WONDERFUL LIFE                                    31

16 rows selected.
```

第 3 种选择，使用 full outer join 返回两个表中的所有行。不满足 on 条件的行返回值为 NULL。在这个示例中，因为没有行满足上述条件，所以该查询返回了和 right outer join 相同的 31 行。

```
select B.Title, MAX(BC.ReturnedDate - BC.CheckoutDate)
          "Most Days Out"
  from BOOKSHELF_CHECKOUT BC full outer join BOOKSHELF B
    on BC.Title = B.Title
group by B.Title;
```

13.2.3 用外部连接代替 NOT IN

哪些书籍没有借出？可编写如下的查询：

```
select Title
  from BOOKSHELF
  where Title not in
        (select Title from BOOKSHELF_CHECKOUT)
  order by Title;

TITLE
----------------------------------------
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EMMA WHO SAVED MY LIFE
GOSPEL
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
PREACHING TO HEAD AND HEART
RUNAWAY BUNNY
SHOELESS JOE
THE COST OF DISCIPLESHIP
THE GOOD BOOK
TRUMPET OF THE SWAN
UNDER THE EYE OF THE CLOCK
```

这是编写这类查询的典型方式。出于性能考虑，优化程序可在内部将 NOT IN 转换为以下具有同样功能的方法之一。下面的查询使用一个外部连接并产生相同的结果。

```
select distinct B.Title
    from BOOKSHELF_CHECKOUT BC right outer join BOOKSHELF B
      on BC.Title = B.Title
  where BC.Title is NULL
  order by B.Title;

TITLE
```

```
----------------------------------------
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EMMA WHO SAVED MY LIFE
GOSPEL
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
PREACHING TO HEAD AND HEART
RUNAWAY BUNNY
SHOELESS JOE
THE COST OF DISCIPLESHIP
THE GOOD BOOK
TRUMPET OF THE SWAN
UNDER THE EYE OF THE CLOCK
```

为什么此查询有效并给出和 NOT IN 相同的结果呢？这是因为两个表之间的外部连接确保所有行都可以用于测试，包括那些在 BOOKSHELF_CHECKOUT 表中未列出借出记录的书名。下面的代码行

```
where BC.Title is NULL
```

只列出那些没有出现在 BOOKSHELF_CHECKOUT 表中的书名(因此，由 Oracle 作为 NULL 书名返回)。虽然这里的逻辑不明显，但很有效。使用这种技术最好的方法是模仿。

13.2.4　用 NOT EXISTS 代替 NOT IN

执行这种查询的一种较常见的方法是使用 NOT EXISTS 子句。NOT EXISTS 子句一般用来判定一个表中哪些值在另一个表中没有匹配的值。在使用时，它与 EXISTS 子句完全相同。在下面的示例中，可以看到在查询逻辑和返回记录方面的差异。

NOT EXISTS 允许使用相关子查询从一个表中排除那些可能已经成功连接到另一个表的所有记录。在这个示例中，这表示可以从 BOOKSHELF 表中排除所有出现在 BOOKSHELF_CHECKOUT 表的 Title 列中的书名。下面的查询可以说明了如何完成这项工作：

```
select B.Title
    from BOOKSHELF B
  where not exists
        (select 'x' from BOOKSHELF_CHECKOUT BC
        where BC.Title = B.Title)
  order by B.Title;

TITLE
----------------------------------------
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EMMA WHO SAVED MY LIFE
GOSPEL
JOURNALS OF LEWIS AND CLARK
```

```
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
PREACHING TO HEAD AND HEART
RUNAWAY BUNNY
SHOELESS JOE
THE COST OF DISCIPLESHIP
THE GOOD BOOK
TRUMPET OF THE SWAN
UNDER THE EYE OF THE CLOCK
```

如前面通过 NOT IN 和外部连接方法所示的那样，这个查询给出了未借出的书籍。这个查询是怎样工作的呢？

对于 BOOKSHELF 表中的每个记录，都要检查 NOT EXISTS 子查询。如果该记录与 BOOKSHELF_CHECKOUT 表的连接返回一行，那么这个子查询的结果 EXIST。NOT EXISTS 告诉子查询颠倒返回的代码，这样外部连接就不会返回 BOOKSHELF 表中可成功连接到 BOOKSHELF_CHECKOUT 表的行。唯一留下的行就是在 BOOKSHELF_CHECKOUT 表中无匹配行的 BOOKSHELF 行。

NOT EXISTS 是完成这种查询的一种有效的方法，特别是在利用多列进行连接，且在连接列上创建了索引的情况下更是如此。

13.3 自然连接和内部连接

可利用 natural 关键字指出一个连接应该基于被连接的两个表中具有相同名称的列来执行。例如，BOOK_ORDER 中的哪些书名与 BOOKSHELF 中已经存在的书名匹配？

```
select Title
  from BOOK_ORDER natural join BOOKSHELF;

TITLE
----------------------------------------
SHOELESS JOE
GOSPEL
```

自然连接(natural join)返回了与如下查询相同的结果。

```
select BO.Title
  from BOOK_ORDER BO, BOOKSHELF
where BO.Title = BOOKSHELF.Title
  and BO.Publisher = BOOKSHELF.Publisher
  and BO.CategoryName = BOOKSHELF.CategoryName;
```

此连接基于两个表公有的列执行。

内部连接(inner join)是默认设置的，它们返回两个表公有的行，是除外部连接外的另一种办法。注意，由于它支持 on 和 using 子句，因此可以用如下程序清单指定连接条件：

```
select BO.Title
  from BOOK_ORDER BO inner join BOOKSHELF B
    on BO.Title = B.Title;

TITLE
----------------------------------------
GOSPEL
SHOELESS JOE
```

13.4　UNION、INTERSECT 和 MINUS

有时，需要把多个表中相似类型的信息合并起来。一个典型的示例是在发送之前合并两个或多个邮件列表。根据特定邮件的目的，您可能想把信发送给以下任意类型的人：

- 发送给两个列表中的每个人(同时避免给两个列表中都存在的某个人发送两封信)。
- 只发送给两个列表中都列出的那些人。
- 发送给只在一个列表中列出的那些人。

列表的这 3 种组合就是 Oracle 中所谓的 UNION(并)、INTERSECT(交)、MINUS(差)。在下面的示例中，将看到怎样利用这 3 个子句处理多个查询的结果。这些示例将比较现有的(BOOKSHELF)表上的书籍与订单上的(BOOK_ORDER)书籍。

为了查看所有的书籍，UNION 这两个表。为了减少输出量，仅选择字母表的前一半的 BOOKSHELF 项。下面的 select 语句将返回 14 行：

```
select Title from BOOKSHELF
 where Title < 'M%';
```

下面的 Select 语句返回 6 行：

```
select Title from BOOK_ORDER;
```

如果把它们 UNION 起来那么将返回多少行呢？

```
select Title from BOOKSHELF
  where Title < 'M%'
union
select Title from BOOK_ORDER;

TITLE
----------------------------------------
ANNE OF GREEN GABLES
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EITHER/OR
EMMA WHO SAVED MY LIFE
GALILEO'S DAUGHTER
GOOD DOG, CARL
GOSPEL
HARRY POTTER AND THE GOBLET OF FIRE
```

```
INNUMERACY
JOHN ADAMS
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
LONGITUDE
ONCE REMOVED
SHOELESS JOE
SOMETHING SO STRONG

19 rows selected.
```

其余的记录到哪里去了呢？问题是 BOOK_ORDER 中的某个 Title 值已经出现在 BOOKSHELF 表中。为说明这个重复现象，使用 UNION ALL 代替 UNION：

```
select Title from BOOKSHELF
  where Title < 'M%'
union all
select Title from BOOK_ORDER
  order by Title;

TITLE
----------------------------------------
ANNE OF GREEN GABLES
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EITHER/OR
EMMA WHO SAVED MY LIFE
GALILEO'S DAUGHTER

GOOD DOG, CARL
GOSPEL
GOSPEL
HARRY POTTER AND THE GOBLET OF FIRE
INNUMERACY
JOHN ADAMS
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
LONGITUDE
ONCE REMOVED
SHOELESS JOE
SOMETHING SO STRONG

20 rows selected.
```

现在，重复的书名列出了两遍。

下面，对书的两个清单求交集。这个清单仅包含在两个基表中都出现的那些名称(注意，这个示例中取消了对 Title < 'M%'的限制)：

```
select Title from BOOKSHELF
intersect
```

```
select Title from BOOK_ORDER
  order by Title;

TITLE
----------------------------------------
GOSPEL
SHOELESS JOE
```

接着，利用 MINUS 运算符生成新书(出现在 BOOK_ORDER 表中，但不出现在 BOOKSHELF 表中的书籍)的清单：

```
select Title from BOOK_ORDER
minus
select Title from BOOKSHELF
  order by Title;

TITLE
----------------------------------------
GALILEO'S DAUGHTER
LONGITUDE
ONCE REMOVED
SOMETHING SO STRONG
```

还可以利用 MINUS 来显示哪些书还未借出：

```
select Title from BOOKSHELF
  minus
select Title from BOOKSHELF_CHECKOUT;

TITLE
------------------------------
BOX SOCIALS
CHARLOTTE'S WEB
COMPLETE POEMS OF JOHN KEATS
EMMA WHO SAVED MY LIFE
GOSPEL
JOURNALS OF LEWIS AND CLARK
KIERKEGAARD ANTHOLOGY
LETTERS AND PAPERS FROM PRISON
PREACHING TO HEAD AND HEART
RUNAWAY BUNNY
SHOELESS JOE
THE COST OF DISCIPLESHIP
THE GOOD BOOK
TRUMPET OF THE SWAN
UNDER THE EYE OF THE CLOCK

15 rows selected.
```

以上简单地介绍了 UNION、INTERSECT 和 MINUS 的基础知识。下面进行更为详细的讨论。在合并两个表时，Oracle 并不关心合并运算符任何一边的列名，也就是说虽然 Oracle 要求每一个 select 语句都是有效的并有对它自己的表有效的列，但第一个 select 语句中的列

名不必和第二个 select 语句中的列名相同。Oracle 有这样一些约定：

- select 语句必须具有相同的列数。如果被查询的两个表具有不同数目的选择列，则可选择串代替列，以便使两个查询的列的列表匹配。
- select 语句中相应的列必须有相同的数据类型(长度可以不同)。

在对输出进行排序时，Oracle 使用第一个 select 语句中的列名给出查询结果。因此，只有第一个 select 语句中的列名可用于 order by 中。

虽然可以对两个或多个表使用合并运算符，但如果这样做，优先级次序就会出现问题，特别在使用 INTERSECT 和 MINUS 时更是如此。应该使用圆括号来强制所需的执行顺序。

13.4.1 IN 子查询

虽然在子查询中可以使用合并运算符，但必须注意优先级。以下查询就容易让人产生误解：

```
select ColA from TABLE_A
 where ColA in
(select Col1 from TABLE_1)
 union
(select Col2 from TABLE_2);
```

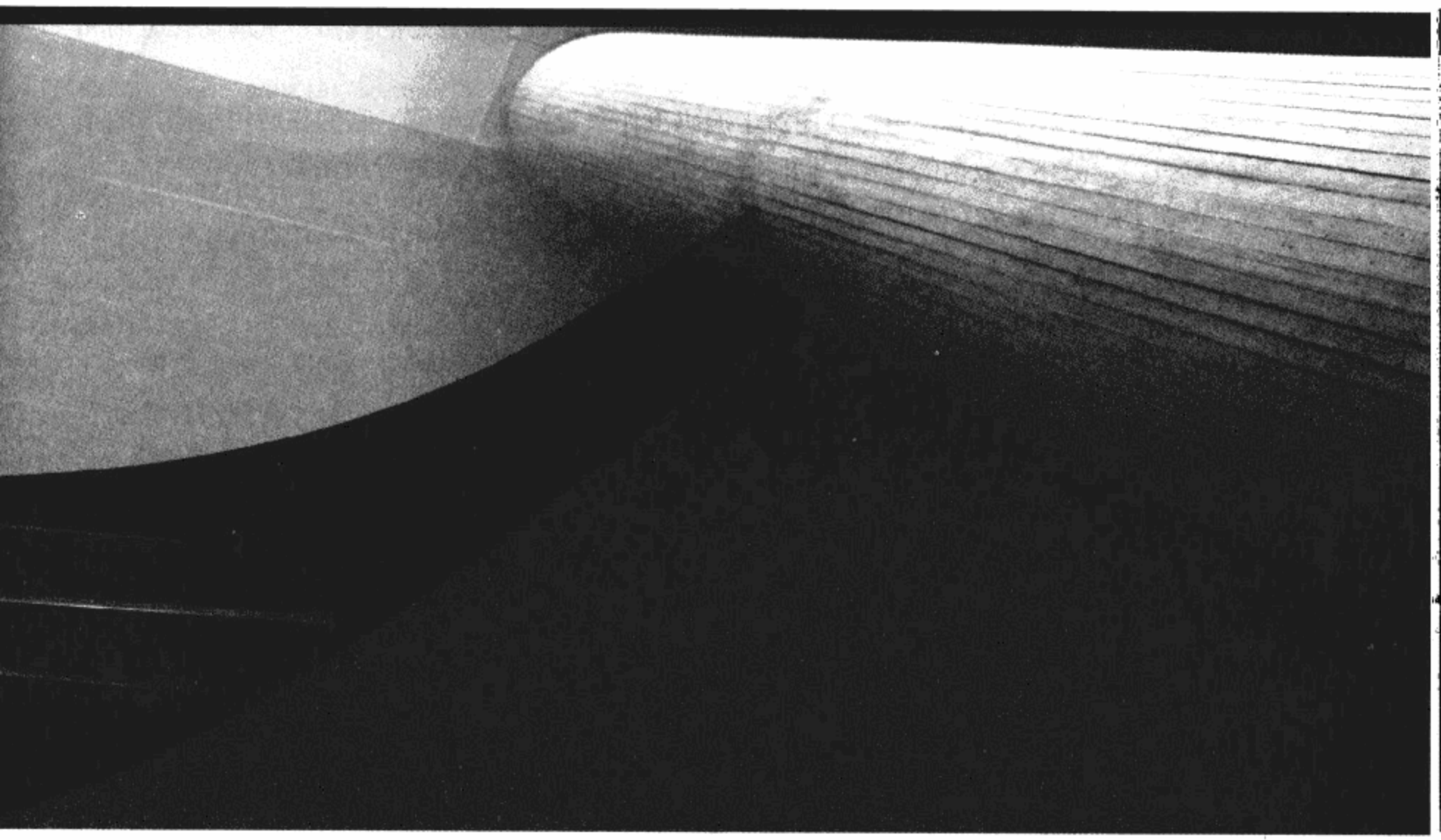
应该首先执行哪一个子查询呢？是执行两个查询的 union 作为单个 where 子句的一部分，还是执行基于 TABLE_1 查询的 in 子句，然后是该结果与 TABLE_2 的查询的 union？应该利用圆括号使意思清楚，并强制合适的运算优先级。除非用圆括号改变查询执行的方式，否则 in 语句的优先级总是比 union 要高。如果希望 union 具有更高的优先级，则可按如下方法使用圆括号：

```
select ColA from TABLE_A
 where ColA in
(select Col1 from TABLE_1
 union
 select Col2 from TABLE_2);
```

13.4.2 UNION、INTERSECT 和 MINUS 的限制

在 where 子句中使用 UNION、INTERSECT 或 MINUS 的查询必须在其 select 列表中具有相同数量和相同类型的列。注意，等价的 IN 结构没有这种限制。

用组合运算符代替 IN、AND 和 OR 只不过是一种个人喜好。大多数 SQL 用户认为 IN、AND 和 OR 比组合运算更清晰、更容易理解。



第 14 章

一些复杂的技术

本章继续学习更复杂的 Oracle 函数和功能。本章重点介绍创建可转变为视图的简单查询和分组查询，以及计算总计的应用，并创建可显示树型结构的报表。与第 13 章介绍的技术一样，这些技术对于多数报表来说并不是必需的。虽然它们看起来有点难，但不要担心。如果您是使用 Oracle 及其查询工具的新手，那么知道有这些功能并且在需要的时候能够找到它们就已经足够了。

14.1 复杂的分组

视图可以互为基础来创建。第 12 章介绍了创建一个表中分组行的视图的概念。如第 12

章所示，您可以很容易地将视图连接到其他视图和表，以生成另外的视图，从而简化查询和报表的任务。

当分组工作越来越复杂时，视图对于编码工作来说就越有价值。它们在应用程序内的不同分组级别上简化了数据的表示。同时还能更容易地使用更高级的分析函数。

考虑 CATEGORY_COUNT 视图，第 12 章已经介绍过：

```
create or replace view CATEGORY_COUNT as
select CategoryName, COUNT(*) as Counter
   from BOOKSHELF
  group by CategoryName;

select * from CATEGORY_COUNT
  order by CategoryName;

CATEGORYNAME              COUNTER
------------------- ----------
ADULTFIC                       6
ADULTNF                       10
ADULTREF                       6
CHILDRENFIC                    5
CHILDRENNF                     1
CHILDRENPIC                    3
```

根据 Counter 列值的大小进行排列，最大列值的排在前面：

```
select * from CATEGORY_COUNT
 order by Counter desc;

CATEGORYNAME              COUNTER
------------------- ----------
ADULTNF                       10
ADULTFIC                       6
ADULTREF                       6
CHILDRENFIC                    5
CHILDRENPIC                    3
CHILDRENNF                     1
```

输出结果显示了种类的顺序。按照书的数目，ADULTNF 类排在第一。不显示这个清单，也可以确定不同的 Counter 值排在何处。为此，可以使用 RANK 函数。如下面的程序清单所示，RANK 函数接收一个值作为输入，并且具有几个不同的子句，即 within group 子句和 order by 子句，它们告诉 Oracle 怎样进行排序。值为 3 的 Counter 应排在何处？

```
select RANK(3) within group
(order by Counter desc)
  from CATEGORY_COUNT;

RANK(3)WITHINGROUP(ORDERBYCOUNTERDESC)
--------------------------------------
                                     5
```

值为 3 的 Counter 排在第 5。那么，Counter 值为 8 应该排在何处呢？

```
select RANK(8) within group
(order by Counter desc)
  from CATEGORY_COUNT;

RANK(8)WITHINGROUP(ORDERBYCOUNTERDESC)
-------------------------------------
                                     2
```

增加这 5 本书后，该种类将提升到第 2 位。从百分比的角度来看，该种类的百分比的排序是怎样的呢？

```
select PERCENT_RANK(8) within group
(order by Counter desc)
  from CATEGORY_COUNT;

PERCENT_RANK(8)WITHINGROUP(ORDERBYCOUNTERDESC)
---------------------------------------------
                                   .166666667
```

正如所料，它在种类前 1/6 处。

利用这种汇总视图和分析函数的技术，可创建包括加权平均、实际收益率、占总数的百分比、占小计的百分比以及许多相似计算的视图和报表。对于每个视图上可以相互创建多少视图没有具体的限制，不过，即使是最复杂的计算也很少需要在视图上创建多于 3 层或 4 层视图。请注意，如第 12 章所述，还可以在 from 子句中创建内联视图。

14.2　使用临时表

可以为自己的会话创建一个独立存在的表，或者创建一个数据在事务处理期间可以永久存在的表。可以使用临时表来支持专门的统计或支持特定的应用处理要求。

可以使用 create global temporary table 命令创建临时表。在创建临时表时，可以指定它是否在整个会话期间都存在(通过 on commit preserve rows 子句)，或者在事务处理完成时是否删除它的行(通过 on commit delete rows 子句)。

与永久表不同的是，在创建临时表时不会自动分配空间。表的空间是在插入行时动态分配的：

```
create global temporary table YEAR_ROLLUP (
    Year NUMBER(4),
    Month VARCHAR2(9),
    Counter NUMBER)
on commit preserve rows;
```

通过为这个表查询 USER_TABLES 的 Duration 列，将会看到 YEAR_ROLLUP 表中的数据持续时间。这里，Duration 的值为 SYS$SESSION。如果指定用 on commit delete rows 代替，

则 Duration 的值将变成 SYS$TRANSACTION。

既然 YEAR_ROLLUP 表已经存在，就可以为它填充数据了，比如利用一个复杂的查询通过 insert as select 命令进行填充。之后可以作为与其他表连接的一部分来查询 YEAR_ROLLUP 表。您会发现这种方法比前面介绍的方法更容易实现。

14.3 使用 ROLLUP、GROUPING 和 CUBE

怎样才能在单个 SQL 语句内而不是通过 SQL*Plus 命令完成分组操作(比如进行合计)呢？可以利用 ROLLUP 和 CUBE 函数增强查询中进行的分组操作。我们来看看它们怎样支持管理与书籍归还有关的数据。因为书籍借出的数量越来越多，所以借出的时间限定为 14 天，每超期一天收取 0.2 美元。下面的报表显示了每个人应支付的滞纳金。

```
set headsep !
column Name format a20
column Title format a20 word_wrapped
column DaysOut format 999.99 heading 'Days!Out'
column DaysLate format 999.99 heading 'Days!Late'

break on Name skip 1 on report
compute sum of LateFee on Name
set linesize 80
set pagesize 60
set newpage 0

select Name, Title, ReturnedDate,
       ReturnedDate-CheckoutDate as DaysOut /*Count days*/,
       ReturnedDate-CheckoutDate -14 DaysLate,
       (ReturnedDate-CheckoutDate -14)*0.20 LateFee
  from BOOKSHELF_CHECKOUT
 where ReturnedDate-CheckoutDate > 14
 order by Name, CheckoutDate;
```

NAME	TITLE	RETURNEDD	Days Out	Days Late	LATEFEE
DORAH TALBOT	MY LEDGER	03-MAR-02	16.00	2.00	.4
**********					--------
sum					.4
EMILY TALBOT	ANNE OF GREEN GABLES	20-JAN-02	18.00	4.00	.8
**********					--------
sum					.8
FRED FULLER	JOHN ADAMS	01-MAR-02	28.00	14.00	2.8
	TRUMAN	20-MAR-02	19.00	5.00	1
**********					--------
sum					3.8
GERHARDT KENTGEN	WONDERFUL LIFE	02-FEB-02	31.00	17.00	3.4
	THE MISMEASURE OF MAN	05-MAR-02	20.00	6.00	1.2

```
**********                                                    --------
sum                                                              4.6

JED HOPKINS        INNUMERACY              22-JAN-02    21.00    7.00        1.4
**********                                                    --------
sum                                                              1.4

PAT LAVAY          THE MISMEASURE OF MAN   12-FEB-02    31.00   17.00        3.4
**********                                                    --------
sum                                                              3.4

ROLAND BRANDT      THE SHIPPING NEWS       12-MAR-02    59.00   45.00          9
                   THE DISCOVERERS         01-MAR-02    48.00   34.00        6.8
                   WEST WITH THE NIGHT     01-MAR-02    48.00   34.00        6.8
**********                                                    --------
sum                                                             22.6
```

可以取消显示 DaysOut，只集中观察滞纳金，显示每个返还日期应支付的费用：

```
clear compute
clear break
select ReturnedDate, Name,
       SUM((ReturnedDate-CheckoutDate -14)*0.20) LateFee
   from BOOKSHELF_CHECKOUT
  where ReturnedDate-CheckoutDate > 14
  group by ReturnedDate, Name
  order by ReturnedDate, Name;

RETURNEDD  NAME                     LATEFEE
---------  -------------------- ----------
20-JAN-02  EMILY TALBOT                .8
22-JAN-02  JED HOPKINS                1.4
02-FEB-02  GERHARDT KENTGEN           3.4
12-FEB-02  PAT LAVAY                  3.4
01-MAR-02  FRED FULLER                2.8
01-MAR-02  ROLAND BRANDT             13.6
03-MAR-02  DORAH TALBOT                .4
05-MAR-02  GERHARDT KENTGEN           1.2
12-MAR-02  ROLAND BRANDT                9
20-MAR-02  FRED FULLER                  1
```

然后，进一步修改该程序，按月对滞纳金分组：

```
select TO_CHAR(ReturnedDate,'MONTH'), Name,
       SUM((ReturnedDate-CheckoutDate -14)*0.20) LateFee
   from BOOKSHELF_CHECKOUT
  where ReturnedDate-CheckoutDate > 14
  group by TO_CHAR(ReturnedDate,'MONTH'), Name;

TO_CHAR(R  NAME                     LATEFEE
---------  -------------------- ----------
FEBRUARY   PAT LAVAY                   3.4
FEBRUARY   GERHARDT KENTGEN            3.4
JANUARY    JED HOPKINS                 1.4
```

```
JANUARY     EMILY TALBOT                .8
MARCH       FRED FULLER                3.8
MARCH       DORAH TALBOT                .4
MARCH       ROLAND BRANDT              22.6
MARCH       GERHARDT KENTGEN            1.2
```

除了可以按照 Month 和 Name 进行分组外，还可利用 ROLLUP 函数生成小计和总计。在下面的示例中，可修改 group by 子句，使它包括一个 ROLLUP 函数调用。请注意，结果集结尾处和每个月之后生成的额外的行。

```
select TO_CHAR(ReturnedDate,'MONTH'), Name,
       SUM((ReturnedDate-CheckoutDate -14)*0.20) LateFee
  from BOOKSHELF_CHECKOUT
 where ReturnedDate-CheckoutDate > 14
 group by ROLLUP(TO_CHAR(ReturnedDate,'MONTH'), Name);
```

```
TO_CHAR(R  NAME                 LATEFEE
---------  -------------------- ----------
FEBRUARY   PAT LAVAY                   3.4
FEBRUARY   GERHARDT KENTGEN            3.4
FEBRUARY                               6.8
JANUARY    JED HOPKINS                 1.4
JANUARY    EMILY TALBOT                 .8
JANUARY                                2.2
MARCH      FRED FULLER                 3.8
MARCH      DORAH TALBOT                 .4
MARCH      ROLAND BRANDT              22.6
MARCH      GERHARDT KENTGEN            1.2
MARCH                                    28
                                         37
```

Oracle 计算了每个月总的滞纳金，并用名为 NULL 的值显示它。输出结果在 February 中给出了两个单独的 3.40 美元，而当月合计为 6.80 美元。该季度滞纳金的合计为 37.00 美元。虽然可以利用 SQL*Plus 命令来进行计算(参考第 6 章)，但这种方法允许利用单个 SQL 命令生成这些合计，而不管查询数据库所用的工具是什么。

下面来优化一下这个报表的外观。可利用 GROUPING 函数确定行是总计还是小计(由 ROLLUP 生成)，或者对应于数据库中的一个 NULL 值。在 select 子句中，将用如下方式选择 Name 列：

```
select DECODE(GROUPING(Name),1, 'All names',Name),
```

如果此列的值是由 ROLLUP 操作产生的，则 GROUPING 函数将返回 1。这个查询使用 DECODE(第 16 章将详细介绍)查看 GROUPING 函数的结果。如果 GROUPING 输出的值是 1，那么可知道该值是由 ROLLUP 函数生成的，Oracle 将输出所有的人名；否则将打印 Name 列的值。我们将对 Date 列应用类似的逻辑。下面的程序清单给出完整的查询及输出结果：

```
select DECODE(GROUPING(TO_CHAR(ReturnedDate,'MONTH')),1,
                 'All months',TO_CHAR(ReturnedDate,'MONTH')),
              DECODE(GROUPING(Name),1, 'All names',Name),
```

```
       SUM((ReturnedDate-CheckoutDate -14)*0.20) LateFee
  from BOOKSHELF_CHECKOUT
 where ReturnedDate-CheckoutDate > 14
 group by ROLLUP(TO_CHAR(ReturnedDate, 'MONTH'), Name);

DECODE(GRO  DECODE(GROUPING(NAME),1,'     LATEFEE
----------  ------------------------ ----------
FEBRUARY    PAT LAVAY                       3.4
FEBRUARY    GERHARDT KENTGEN                3.4
FEBRUARY    All names                       6.8
JANUARY     JED HOPKINS                     1.4
JANUARY     EMILY TALBOT                     .8
JANUARY     All names                       2.2
MARCH       FRED FULLER                     3.8
MARCH       DORAH TALBOT                     .4
MARCH       ROLAND BRANDT                  22.6
MARCH       GERHARDT KENTGEN                1.2
MARCH       All names                        28
All months  All names                        37
```

可以使用 CUBE 函数为在 group by 子句中的值的组合生成小计。下面的查询使用 CUBE 函数生成这些信息：

```
select DECODE(GROUPING(TO_CHAR(ReturnedDate,'MONTH')),1,
          'All months',TO_CHAR(ReturnedDate,'MONTH')),
       DECODE(GROUPING(Name),1, 'All names',Name),
       SUM((ReturnedDate-CheckoutDate -14)*0.20) LateFee
  from BOOKSHELF_CHECKOUT
 where ReturnedDate-CheckoutDate > 14
 group by CUBE(TO_CHAR(ReturnedDate,'MONTH'), Name);

DECODE(GRO  DECODE(GROUPING(NAME),1,'     LATEFEE
----------  ------------------------ ----------
All months  All names                        37
All months  PAT LAVAY                       3.4
All months  FRED FULLER                     3.8
All months  JED HOPKINS                     1.4
All months  DORAH TALBOT                     .4
All months  EMILY TALBOT                     .8
All months  ROLAND BRANDT                  22.6
All months  GERHARDT KENTGEN                4.6
FEBRUARY    All names                       6.8
FEBRUARY    PAT LAVAY                       3.4
FEBRUARY    GERHARDT KENTGEN                3.4
JANUARY     All names                       2.2
JANUARY     JED HOPKINS                     1.4
JANUARY     EMILY TALBOT                     .8
MARCH       All names                        28
MARCH       FRED FULLER                     3.8
MARCH       DORAH TALBOT                     .4
MARCH       ROLAND BRANDT                  22.6
MARCH       GERHARDT KENTGEN                1.2
```

CUBE 函数提供了由 ROLLUP 选项生成的合计，并且按 Name 显示了 All months 种类的合计。在标准的 SQL 中执行合计的功能极大地增强了用户选择最好的报表工具的能力。

14.4 家族树和 connect by

Oracle 的一个更为有趣但很少使用或不太为人所知的工具是 connect by 子句。简单地说，它是一种以家族树的分支顺序生成报表的方法。我们经常会遇到这样的树，如人类家族的血统、牲畜、马的谱系，企业管理、公司部门、制造业，文献、思想、演化、科学研究和理论(甚至创建在视图之上的视图)的体系等。

connect by 子句提供了对树中所有成员进行报告的手段。它能够排除家族树的分支或者个别成员，可以自顶向下或自底向上遍历树，在遍历树的过程中报告遇到的成员。

树中最早的祖先在技术上称为根节点(root node)，在日常用语中，也称为树干。从树干延伸出来的是分支，分支之外又有分支。自一个较大的分支处又分裂出一个或多个分支的分叉叫做节点(node)。分支的终端叫做叶或叶节点。图 14-1 显示了这样一棵树的结构图。

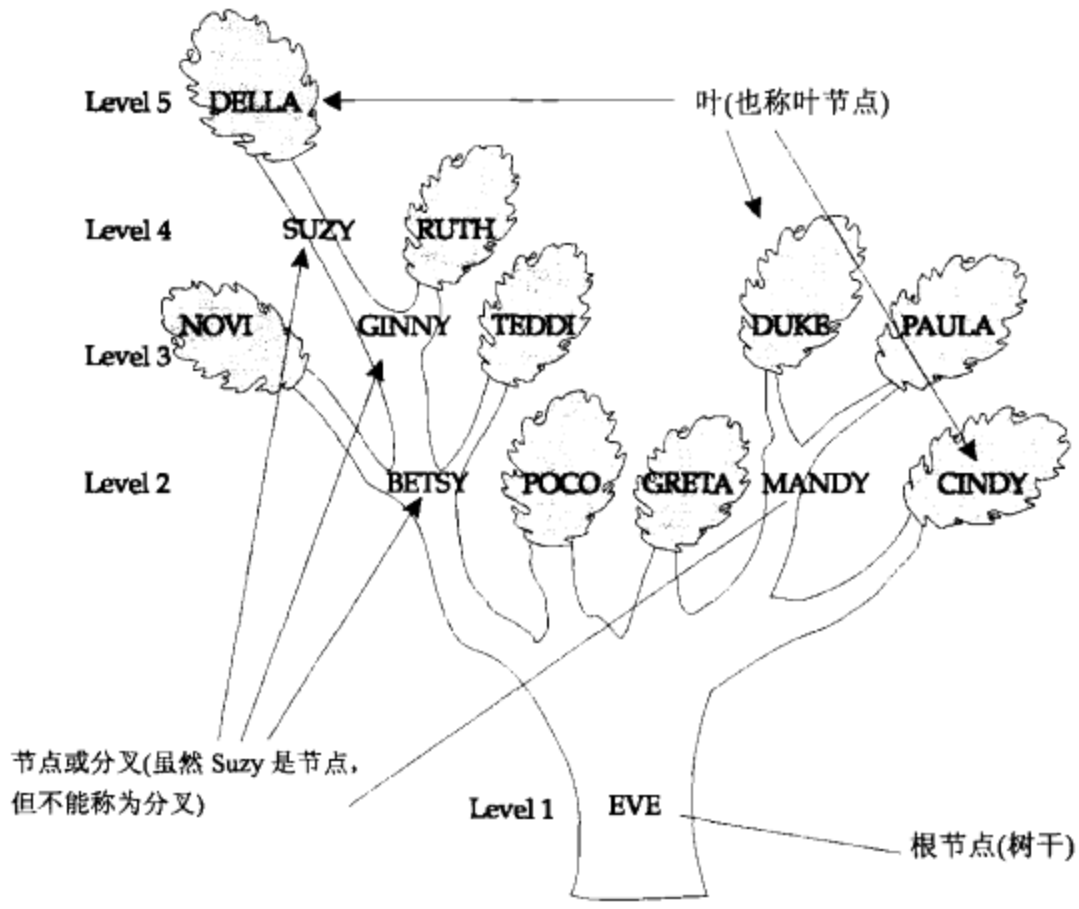


图 14-1 Eve 的后裔

下面是一张从 1900 年 1 月到 1908 年 10 月间出生的母牛和公牛的表。每出生一个后代，都把它以及它的性别、父母(母牛和公牛)和它的出生日期一起作为一行输入该表中。如果把这个表中的 Cow 和 Offspring 与图 14-1 进行比较，则将发现它们是对应的。EVE 没有记录父母，因为它出生在另一个农场，ADAM 和 BANDIT 是为繁殖而引进的公牛，它们在表中也没有双亲。

```
column Cow format a6
column Bull format a6
```

```
column Offspring format a10
column Sex format a3

select * from BREEDING
  order by Birthdate;

OFFSPRING  SEX  COW     BULL    BIRTHDATE
---------  ---  ------  ------  ---------
BETSY      F    EVE     ADAM    02-JAN-00
POCO       M    EVE     ADAM    15-JUL-00
GRETA      F    EVE     BANDIT  12-MAR-01
MANDY      F    EVE     POCO    22-AUG-02
CINDY      F    EVE     POCO    09-FEB-03
NOVI       F    BETSY   ADAM    30-MAR-03
GINNY      F    BETSY   BANDIT  04-DEC-03
DUKE       M    MANDY   BANDIT  24-JUL-04
TEDDI      F    BETSY   BANDIT  12-AUG-05
SUZY       F    GINNY   DUKE    03-APR-06
PAULA      F    MANDY   POCO    21-DEC-06
RUTH       F    GINNY   DUKE    25-DEC-06
DELLA      F    SUZY    BANDIT  11-OCT-08
ADAM       M
EVE        F
BANDIT     M
```

下面编写一个查询来形象地说明这个家族关系。这个查询使用了 LPAD 以及随 connect by

```
EVE        POCO          CINDY                    F 09-FEB-03
EVE        BANDIT        GRETA                    F 12-MAR-01
EVE        POCO          MANDY                    F 22-AUG-02
MANDY      BANDIT              DUKE               M 24-JUL-04
MANDY      POCO                PAULA              F 21-DEC-06
EVE        ADAM          POCO                     M 15-JUL-00
```

请注意，这个结果是图 14-1 按顺时针方向旋转而得到的。虽然 EVE 不是中心，但它是该树的根节点(树干)。它的孩子是 BETSY、CINDY、GRETA、MANDY 和 POCO。BETSY 的孩子是 GINNY、NOVI 和 TEDDI，而 GINNY 的孩子是 RUTH 和 SUZY，SUSY 的孩子是 DELLA。MANDY 也有两个孩子，分别是 DUKE 和 PAULA。

这个树以 EVE 作为第一代。如果 SQL 语句用 start with 指明以 MANDY 开始，则只有 MANDY、DUKE 和 PAULA 被选择。start with 定义了将要显示的那部分树的开始，而且它只包括从 start with 指定的个体开始的分支。start with 的行为顾名思义。

select 语句中的 LPAD 可能有点令人困惑。回顾一下第 7 章，LPAD 的格式是：

```
LPAD( string, length [,' set'])
```

即接收指定的 string 并且用特定的字符 set(字符集)从左开始填充，填充长度为 length。如果未给出 set，则左边填充空格串。

请将下面的 LPAD 语法与上面 select 语句中的 LPAD 进行比较：

```
LPAD(' ',6*(Level-1))
```

在本例中，string 是单个字符空格(由单引号括起来的空字符表示)。6*(level-1)为 length，因为没有给出 set，所以将使用空格。也就是说，它告诉 SQL 接收这个由一个空格组成的字符串，从左开始填充由 6*(level-1)所确定的空格数。计算过程是先将 Level 的值减去 1，再把得到的结果乘以 6。对 EVE 来说，因为其 Level 值为 1，所以 6*(1-1)为 0，即填充 0 个空格。对 BETSY 来说，由于 Level(表示第几代)值是 2，因此就要 6 次左填充。因此，每增加一代，就要在 Offspring 列的左边连接 6 个空格。这在上面所显示的结果中很明显，每个 Offspring 的名称都会缩进一些，因为在其左边填充了相应于其 Level 或第几代的空格数。

为什么要这样做，而不是简单地把 LPAD 直接用于 Offspring 呢？有两个原因，首先，在 Offspring 上直接使用 LPAD 会使 Offspring 的名字右对齐。每一代的各个名字的最后一个字母将排在同一列上(即最后一个字母对齐)。其次，如果 Level-1 等于 0(对于 EVE)，则 EVE 的 LPAD 结果将是 0 个字符宽，从而会导致 EVE 消失：

```
select Cow, Bull, LPAD(Offspring,6*(Level-1),' ') AS Offspring,
       Sex, Birthdate from BREEDING
  start with Offspring = 'EVE'
connect by Cow = PRIOR Offspring;

COW    BULL     OFFSPRING                S BIRTHDATE
------ -------- ------------------------ - ---------
                                         F
EVE    ADAM     BETSY                    F 02-JAN-00
BETSY  BANDIT         GINNY              F 04-DEC-03
```

```
GINNY   DUKE                    RUTH            F 25-DEC-06
GINNY   DUKE                    SUZY            F 03-APR-06
SUZY    BANDIT                        DELLA     F 11-OCT-08
BETSY   ADAM            NOVI                    F 30-MAR-03
BETSY   BANDIT          TEDDI                   F 12-AUG-05
EVE     POCO    CINDY                           F 09-FEB-03
EVE     BANDIT  GRETA                           F 12-MAR-01
EVE     POCO    MANDY                           F 22-AUG-02
MANDY   BANDIT          DUKE                    M 24-JUL-04
MANDY   POCO            PAULA                   F 21-DEC-06
EVE     ADAM    POCO                            M 15-JUL-00
```

因此，为了使每一代都有适当的空间，以确保 EVE 的出现并使名字左对齐，就要把 LPAD 和连接函数一起使用，而不是直接作用于 Offspring 列。

现在来看看 connect by 是如何工作的？请再看一下图 14-1。从 NOVI 开始向下遍历，哪些母牛是 NOVI 的前辈呢？第一个是 BETSY，而 BETSY 的母亲又是 EVE。这有点不太好懂，下面的子句：

```
connect by Cow = PRIOR Offspring
```

告诉 SQL 查找下一行，其中 COW 列的值和前面一行的 Offspring 列的值相等。看一看上面的表就知道这是真的。

14.4.1　排除个体和分支

有两种方法可以从报表中排除母牛。一种方法是使用 where 子句，另一种方法是使用 connect by 子句。不同之处在于，使用 connect by 子句不仅排除所提到的母牛，而且排除它的所有孩子。如果使用 connect by 子句排除 BETSY，则 NOVI、GINNY、TEDDI、SUZY、RUTH 和 DELLA 也将全部消失。connect by 子句实际上是对树结构进行追踪。如果 BETSY 没有出生，那么它也不会有子孙。在下面的示例中，用 and 子句修改 connect by 子句：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
          Sex, Birthdate
    from BREEDING
  start with Offspring = 'EVE'
connect by Cow = PRIOR Offspring
  and Offspring != 'BETSY';

COW     BULL    OFFSPRING                       SEX BIRTHDATE
------  ------  ------------------------------  --- ---------
                EVE                             F
EVE     POCO        CINDY                       F   09-FEB-03
EVE     BANDIT      GRETA                       F   12-MAR-01
EVE     POCO        MANDY                       F   22-AUG-02
MANDY   BANDIT              DUKE                M   24-JUL-04
MANDY   POCO                PAULA               F   21-DEC-06
EVE     ADAM        POCO                        M   15-JUL-00
```

where 子句只删除它所提及的母牛。如果 BETSY 死了，则从表中删除它，但它的子孙不

被删除。事实上，请注意，作为 NOVI、CINNY 和 TEDDI 的妈妈，母牛 BETSY 仍然在 COW 列下面：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
  from BREEDING
 where Offspring != 'BETSY'
start with Offspring = 'EVE'
 connect by Cow = PRIOR Offspring;

COW     BULL      OFFSPRING                      S  BIRTHDATE
------- --------- ------------------------------ -  ---------
                  EVE                            F
BETSY   BANDIT                GINNY              F  04-DEC-03
GINNY   DUKE                        RUTH         F  25-DEC-06
GINNY   DUKE                        SUZY         F  03-APR-06
SUZY    BANDIT                            DELLA  F  11-OCT-08
BETSY   ADAM                  NOVI               F  30-MAR-03
BETSY   BANDIT                TEDDI              F  12-AUG-05
EVE     POCO          CINDY                      F  09-FEB-03
EVE     BANDIT        GRETA                      F  12-MAR-01
EVE     POCO          MANDY                      F  22-AUG-02
MANDY   BANDIT                DUKE               M  24-JUL-04
MANDY   POCO                  PAULA              F  21-DEC-06
EVE     ADAM          POCO                       M  15-JUL-00
```

如图 14-1 所示，在使用 connect by 时，家族树显示的顺序基本上是一层一层、从左到右、从最低层(即第一层)开始的。

14.4.2 向根遍历

到目前为止，在家族树的报表中遍历的方向都是从父代向子代进行的。能否从一个孩子开始，向后遍历到父母、祖父母、曾祖父母呢？要做到这一点，只需把词 prior 移到等号的另一边即可。在下面的示例中，Offspring 设置为等于 prior Cow 的值；而在前面的示例中，Cow 设置为等于 prior Offspring 的值。下面就是要找出 DELLA 的祖先：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
  from BREEDING
 start with Offspring = 'DELLA'
connect by Offspring = PRIOR Cow;

COW    BULL    OFFSPRING                      SEX  BIRTHDATE
------ ------  ------------------------------ ---  ---------
SUZY   BANDIT  DELLA                          F    11-OCT-08
GINNY  DUKE           SUZY                    F    03-APR-06
BETSY  BANDIT               GINNY             F    04-DEC-03
EVE    ADAM                       BETSY       F    02-JAN-00
                                         EVE   F
```

虽然此例显示了 DELLA 自己的根，但是与前面的显示相比有点混乱。看起来好像 DELLA

是祖先，而 EVE 是后代。虽然可以为 Birthdate 添加一个 order by 子句作为帮助，但 EVE 仍然位于最右边：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
   from BREEDING
  start with Offspring = 'DELLA'
connect by Offspring = PRIOR Cow
  order by Birthdate;

COW    BULL   OFFSPRING                      SEX BIRTHDATE
------ ------ ------------------------------ --- ---------
EVE    ADAM                        BETSY      F   02-JAN-00
BETSY  BANDIT              GINNY              F   04-DEC-03
GINNY  DUKE          SUZY                     F   03-APR-06
SUZY   BANDIT DELLA                           F   11-OCT-08
                                        EVE   F
```

解决方法很简单，只要在 LPAD 项中改变计算即可：

```
select Cow, Bull, LPAD(' ',6*(5-Level))||Offspring Offspring,
       Sex, Birthdate
   from BREEDING
  start with Offspring = 'DELLA'
connect by Offspring = PRIOR Cow
  order by Birthdate;

COW    BULL   OFFSPRING                  SEX BIRTHDATE
------ ------ ------------------------   --- ---------
EVE    ADAM       BETSY                  F   02-JAN-00
BETSY  BANDIT           GINNY            F   04-DEC-03
GINNY  DUKE                   SUZY       F   03-APR-06
SUZY   BANDIT                       DELLA F   11-OCT-08
             EVE                          F
```

最后，看一下在使用 connect by 找出 BULL 的父辈时生成的报表有何不同。以下是查找 ADAM 的后代的程序清单：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
   from BREEDING
  start with Offspring = 'ADAM'
connect by PRIOR Offspring = Bull;

COW    BULL   OFFSPRING                  SEX BIRTHDATE
------ ------ ------------------------   --- ---------
             ADAM                       M
EVE    ADAM       BETSY                  F   02-JAN-00
BETSY  ADAM       NOVI                   F   30-MAR-03
EVE    ADAM       POCO                   M   15-JUL-00
EVE    POCO            CINDY             F   09-FEB-03
EVE    POCO            MANDY             F   22-AUG-02
MANDY  POCO            PAULA             F   21-DEC-06
```

ADAM和BANDIT是这群牛中最早的公牛。要创建一个能报告ADAM的子孙和BANDIT的子孙的树，就必须为它们“创造”一个父亲作为该树的根。这些树比前面显示的那种树更优越的地方在于，它能够以多种方式对许多继承组(如家族、项目到公司内的部门等)进行准确的描述：

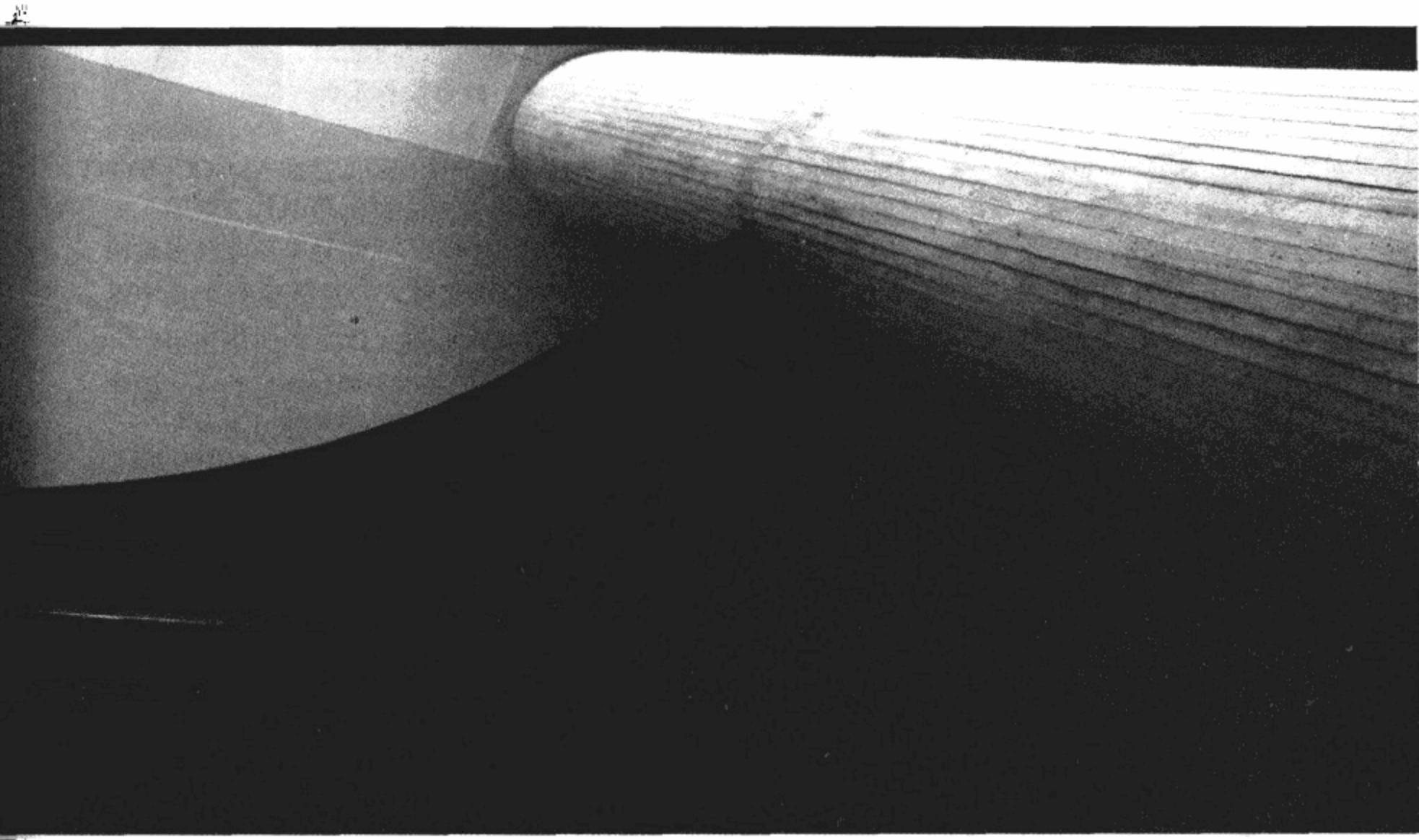
```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
  from BREEDING
 start with Offspring = 'BANDIT'
connect by PRIOR Offspring = Bull;

COW    BULL   OFFSPRING                      SEX BIRTHDATE
------ ------ ------------------------------ --- ---------
              BANDIT                         M
SUZY   BANDIT       DELLA                    F   11-OCT-08
MANDY  BANDIT       DUKE                     M   24-JUL-04
GINNY  DUKE               RUTH               F   25-DEC-06
GINNY  DUKE               SUZY               F   03-APR-06
BETSY  BANDIT       GINNY                    F   04-DEC-03
EVE    BANDIT       GRETA                    F   12-MAR-01
BETSY  BANDIT       TEDDI                    F   12-AUG-05
```

14.4.3 基本规则

使用 connect by 和 start with 来创建类似于树的报表并不难，只要遵循以下基本规则即可：

- 使用 connect by 时各子句的顺序应为：
  (1) select
  (2) from
  (3) where
  (4) start with
  (5) connect by
  (6) order by
- prior 强制报表的顺序变为从根到叶(如果 prior 列是父辈)或从叶到根(如果 prior 列是后代)
- 虽然 where 子句可以从树中排除个体，但不排除它们的子孙(或者祖先，如果 prior 在等号的右边)。
- connect by 中的条件(尤其是不等于)可以排除个体和它所有的子孙(或祖先，取决于怎样跟踪树)。
- connect by 不能和 where 子句中的表连接使用。

这是一组很少有人能够正确记住的特殊命令。但是，有了对树和继承的基本理解，构造一条合适的 select 语句以生成树型报表应该很简单，只需要正确引用本章介绍的语法即可。



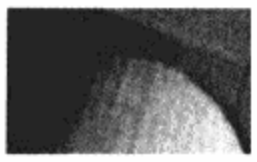
第15章

更改数据：插入、更新、合并和删除

到目前为止，本书介绍的关于 Oracle、SQL 和 SQL*Plus 的一切内容实际上都与从数据库的表中选取数据有关。本章将介绍如何更改表中的数据，包括如何插入新行，如何更新行中各列的值，以及删除整行。虽然这些内容还没有明确介绍，但是由于您已经了解的关于 SQL 的所有内容几乎都可以在此使用，其中包括数据类型、计算、串格式化、where 子句等内容，因此并没有多少新内容。Oracle 提供了一种透明的分布式数据库的能力，即使在远程数据库中也能插入、更新和删除数据(请参考第 25 章)。正如您将在本章中所看到的，Oracle 允许您通过以下两种方法合并这些命令的功能，即多表插入和 merge 命令(在单个命令中执行插入和更新)。

15.1 插入

SQL 的 insert 命令把一行信息直接插入到一个表中(或通过一个视图间接插入)。COMFORT 表记录每个城市一年中 4 个抽样日的中午和午夜的温度以及日降雨量



注意：
为了存储秒的小数部分，可以使用 TIMESTAMP 和 TIMESTAMP WITH TIME ZONE 数据类型，如第 10 章所述。

如果一开始就(在单词 values 之前)列出了数据的顺序的话，则列也可以不按照创建表时所描述的顺序进行插入。这样做并不改变表中各列的基本顺序，只是以不同的顺序列出各数据字段(关于把数据插入到用户定义的数据类型中的信息，请参考第 V 部分)

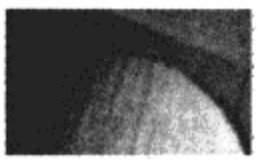
也可以插入 NULL 值。这只表示某行中的该列为空，如下所示：

```
insert into COMFORT
        (SampleDate, Precipitation, City, Noon, Midnight)
values ('23-SEP-03', NULL,
        'WALPOLE', 86.3, 72.1);

1 row created.
```

15.1.2　用 select 插入

您还可以插入从某个表中选中的信息。以下插入的是选自 COMFORT 表的列以及 SampleDate(22-DEC-03)和 City(WALPOLE)字面值的混合形式。请注意 select 语句中的 where 子句，它只能检索出一行。如果 select 语句检索出 5 行，则将插入 5 个新行；如果检索出 10 行，则将新插入 10 行，依此类推：

```
insert into COMFORT
        (SampleDate, Precipitation, City, Noon, Midnight)
select '22-DEC-03', Precipitation,
       'WALPOLE', Noon, Midnight
  from COMFORT
where City = 'KEENE'
   and SampleDate = '22-DEC-03';

1 row created.
```



注意：
除非您使用 TO_LOB 函数将 LONG 型数据插入到 LOB 列中，否则不能把 insert into…select from 语句和 LONG 数据类型一起使用。

当然，在选中的列中不仅可以插入值，还可以在 select 语句中使用适当的串函数、日期函数或数值函数来修改列。这些函数的结果就是将要插入的内容。您可以试着将一个超过列宽的值(如字符数据类型)或超过列值范围的值(如数值数据类型)插入列中。由于必须遵守定义列时所给出的约束，因此这样做将产生一条“value too large for column(列值太大)”或“mismatched datatype(数据类型不匹配)”的错误消息。如果现在查询 COMFORT 表中 Walpole 城市的信息，则结果将是您所插入的那些记录：

```
select * from COMFORT
  where City = 'WALPOLE';
```

```
CITY        SAMPLEDAT       NOON    MIDNIGHT   PRECIPITATION
----------  ---------  ---------- ----------   -------------
WALPOLE     21-MAR-03        56.7       43.8               0
WALPOLE     22-JUN-03        56.7       43.8               0
WALPOLE     22-JUN-03        56.7       43.8               0
WALPOLE     23-SEP-03        86.3       72.1
WALPOLE     22-DEC-03        -7.2       -1.2             3.9

5 rows selected.
```

有两个显示为 22-JUN-03 的记录，因为其中一个有时间元素。使用 TO_CHAR 函数可以查看该时间：

```
select City, SampleDate,
       TO_CHAR(SampleDate,'HH24:MI:SS') AS TimeofDay, Noon
  from COMFORT
 where City = 'WALPOLE';

CITY          SAMPLEDAT  TIMEOFDA        NOON
------------- ---------  -------- ----------
WALPOLE       21-MAR-03  00:00:00        56.7
WALPOLE       22-JUN-03  00:00:00        56.7
WALPOLE       22-JUN-03  01:35:00        56.7
WALPOLE       23-SEP-03  00:00:00        86.3
WALPOLE       22-DEC-03  00:00:00        -7.2
```

15.1.3 使用 APPEND 提示改善插入性能

正如第 46 章将介绍的，Oracle 使用一个优化程序来确定执行每个 SQL 命令的最有效方法。对于 insert 语句来说，Oracle 试图把每个新的记录插入到已分配给表的已有数据块中。这个执行计划优化了存储数据所需的空间的使用。但是，对于带有插入多行的 select 命令的 insert 命令来说，它也许不能提供足够的性能。可以通过使用 APPEND 提示来修正执行计划从而改善大量数据插入操作的性能。APPEND 提示将使数据库去查找已经被插入表中数据的最后一个块，新的记录将从最后一个块的后面开始插入。此外，插入的数据将直接写入数据文件中，而不先进入数据块缓存。因此，在插入期间对数据库进行的空间管理工作就很少了。于是，在使用 APPEND 提示时，插入操作会进行得很快。

可以在 insert 命令中定义 APPEND 提示。提示看起来就像一个注释，因为它也以/*开始并以*/结束。唯一不同的是，字符集开始时会在提示名前加一个“＋”号。下面的示例显示了一个 insert 命令，其数据被添加到表中：

```
insert /*+ APPEND */ into BOOKSHELF (Title)
select Title from BOOK_ORDER
 where Title not in (select Title from BOOKSHELF);
```

来自 BOOK_ORDER 表的记录将插入到 BOOKSHELF 表中。新的记录将放在表的物理存储空间的尾部，而不是重用 BOOKSHELF 表以前所用过的空间。

因为新的记录不会重新使用该表已经使用过的可用空间，所以对 BOOKSHELF 表的空间需求会有所增加。通常，只有在把大量的数据插入到具有较少的可重用空间的表中时，才使用 APPEND 提示。插入追加记录的点称为表的高水位标记(high-water mark)，重置高水位标记的唯一方法是把表截断(truncate)。因为 truncate 将删除所有的记录并且不能回滚，所以应该确保在执行 truncate 操作之前该表有数据备份。关于 truncate 的详细内容，请参考附录 A。

15.2　rollback、commit 和 autocommit 命令

当对数据库进行插入、更新或删除数据的操作时，可以后退或回滚(roll back)已经完成的工作。这在发现错误时很重要。提交或者回滚的过程受两个 SQL*Plus 命令控制：即 commit 和 rollback。另外，SQL*Plus 具有自动提交工作的能力，而不需要用户明确地告诉它要做什么。这个功能由 set 命令的 autocommit 功能控制。像其他的 set 功能一样，您可以按如下方法显示它：

```
show autocommit

autocommit OFF
```

OFF 是默认值，在几乎所有的程序中这都是推荐的模式。也可以为 autocommit 指定一个数值，该值将确定在几条命令后 Oracle 将执行 commit。autocommit off 意味着直到 commit 后，insert、update 和 delete 操作才算最后完成。

```
commit;

commit complete
```

在 commit 操作之前，只有您才能看到所做的工作对表的影响。访问这些表的其他任何人仍将获得旧信息。无论何时您从这些表中选择数据时，都会看到新信息。实际上，您的工作是在“分段”区域进行，您可以与之交互，直到 commit 操作完成为止。您可以执行大量的 insert、update 和 delete 操作，并通过使用以下命令撤消这些工作(即将该表返回到以前的状态)：

```
rollback;

rollback complete
```

然而，消息“rollback complete”(回滚完成)可能会引起误解。它只表示 Oracle 回滚了未被提交的那些工作。如果您提交了一系列的事务，无论是显式使用 commit 命令还是在另一个操作隐式使用中，“rollback complete”消息都将无任何意义。

15.2.1　使用 savepoint

您可以使用 savepoint 回滚当前事务集的部分事务。参考以下命令：

```
insert into COMFORT
```

```
values ('WALPOLE', '22-APR-03',50.1, 24.8, 0);

savepoint A;

insert into COMFORT
values ('WALPOLE', '27-MAY-03',63.7, 33.8, 0);

savepoint B;

insert into COMFORT
values ('WALPOLE', '07-AUG-03',83.0, 43.8, 0);
```

现在，从 COMFORT 表中选择 Walpole 的数据：

```
select * from COMFORT
  where City = 'WALPOLE';
```

```
CITY         SAMPLEDAT      NOON    MIDNIGHT  PRECIPITATION
-----------  --------- ---------- ----------  -------------
WALPOLE      21-MAR-03       56.7        43.8              0
WALPOLE      22-JUN-03       56.7        43.8              0
WALPOLE      22-JUN-03       56.7        43.8              0
WALPOLE      23-SEP-03       86.3        72.1
WALPOLE      22-DEC-03       -7.2        -1.2            3.9
WALPOLE      22-APR-03       50.1        24.8              0
WALPOLE      27-MAY-03       63.7        33.8              0
WALPOLE      07-AUG-03         83        43.8              0
```

该输出显示了 5 个以前的记录和 3 个新增的记录。现在，回滚最后一个插入操作：

```
rollback to savepoint B;
```

如果您现在查询 COMFORT，则将看到虽然已经回滚了最后一个 insert，但其他的 insert 操作结果仍然存在：

```
select * from COMFORT
  where City = 'WALPOLE';
```

```
CITY         SAMPLEDAT      NOON     MIDNIGHT  PRECIPITATION
-----------  --------- ---------   ----------  -------------
WALPOLE      21-MAR-03      56.7         43.8              0
WALPOLE      22-JUN-03      56.7         43.8              0
WALPOLE      22-JUN-03      56.7         43.8              0
WALPOLE      23-SEP-03      86.3         72.1
WALPOLE      22-DEC-03      -7.2         -1.2            3.9
WALPOLE      22-APR-03      50.1         24.8              0
WALPOLE      27-MAY-03      63.7         33.8              0
```

最后两个记录仍然未提交，您应该执行 commit 命令或其他命令强制提交发生。您仍然可以回滚第 2 个插入操作(到 savepoint A)或回滚所有的插入操作(通过 rollback 命令)。

15.2.2　隐式提交

即使没有直接下达 commit 指令，有些操作(如 quit、exit(等价于 quit))以及任意数据定义语言(DDL)的命令也会强制提交发生。使用这些命令会强制提交事务。

15.2.3　自动回滚

如果您虽然已完成了一系列的 insert、update 或者 delete 操作，但是还没有显式或隐式地提交它们，并且遇到了严重的问题(如计算机故障)，则 Oracle 将会自动回滚任意尚未提交的工作。如果机器或数据库发生故障，则 Oracle 将自动回滚，完成下一次数据库的恢复的清理工作。

15.3　多表插入

您可以在一个命令中执行多个插入。您可以无条件地执行所有的插入，也可以指定条件执行插入(用 when 子句告诉 Oracle 怎样控制多个插入)。如果指定 all，则将判定所有的 when 子句；若指定 first，则告诉 Oracle，在它找到一个判定结果为真的 when 子句后将忽略后面的 when 子句。还可以用 else 子句告诉 Oracle，如果 when 子句的判定结果没有一个为真时应该进行什么操作。

为了说明此功能，下面创建一个在结构上与 COMFORT 表稍有不同的新表：

```
drop table COMFORT_TEST;
create table COMFORT_TEST (
City          VARCHAR2(13) NOT NULL,
SampleDate    DATE NOT NULL,
Measure       VARCHAR2(10),
Value         NUMBER(3,1)
);
```

对应 COMFORT 表的每个记录，COMFORT_TEST 表将有多个记录，即它的 Measure 列将有值 Midnight、Noon 和 Precip，允许我们为每个城市的每个采样日期存储更多的测量数据。

现在用 COMFORT 表中的数据无条件地填充 COMFORT_TEST 表：

```
insert ALL
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'NOON', Noon)
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'MIDNIGHT', Midnight)
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'PRECIP', Precipitation)
select City, SampleDate, Noon, Midnight, Precipitation
   from COMFORT
  where City = 'KEENE';

12 rows created.
```

该查询告诉 Oracle 为从 COMFORT 表中返回的每一行插入多行。COMFORT 表的查询返

回了 4 行。第 1 个 into 子句插入 Noon 值，并在 Measure 列中插入‘NOON’文本串。第 2 个 into 子句插入 Midnight 值，第 3 个 into 子句插入 Precipitation 值，如 insert 之后的 COMFORT_TEST 表的查询所示：

```
select * from COMFORT_TEST;

CITY        SAMPLEDAT  MEASURE          VALUE
----------- ---------  ---------- ----------
KEENE       21-MAR-03  NOON             39.9
KEENE       22-JUN-03  NOON             85.1
KEENE       23-SEP-03  NOON             99.8
KEENE       22-DEC-03  NOON             -7.2
KEENE       21-MAR-03  MIDNIGHT         -1.2
KEENE       22-JUN-03  MIDNIGHT         66.7
KEENE       23-SEP-03  MIDNIGHT         82.6
KEENE       22-DEC-03  MIDNIGHT         -1.2
KEENE       21-MAR-03  PRECIP            4.4
KEENE       22-JUN-03  PRECIP            1.3
KEENE       23-SEP-03  PRECIP
KEENE       22-DEC-03  PRECIP            3.9

12 rows selected.
```

如果您用 first 关键字取代 all 关键字会怎样呢？除非使用 when 子句，否则不能使用 first 关键字。下面的示例说明了如何用 when 子句限制在多个插入命令中执行哪个 insert 操作。例如，下面的示例使用了 all 关键字：

```
delete from COMFORT_TEST;
commit;

insert ALL
        when Noon > 80 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
        values (City, SampleDate, 'NOON', Noon)
        when Midnight > 70 then
          into COMFORT_TEST (City, SampleDate, Measure, Value)
        values (City, SampleDate, 'MIDNIGHT', Midnight)
        when Precipitation is not null then
          into COMFORT_TEST (City, SampleDate, Measure, Value)
        values (City, SampleDate, 'PRECIP', Precipitation)
select City, SampleDate, Noon, Midnight, Precipitation
  from COMFORT
where City = 'KEENE';

6 rows created.
```

在 6 行中插入了什么内容？有满足以下条件的两个 Noon 值：

```
    when Noon > 80 then
```

和满足以下条件的 1 个 Midnight 值：

```
  when Midnight > 70 then
```

以及满足以下条件的 3 个 Precipitation 值：

```
  when Precipitation is not null then
```

您可以在 COMFORT_TEST 表中查看该结果：

```
select * from COMFORT_TEST;

CITY            SAMPLEDAT  MEASURE          VALUE
------------    ---------  ----------   ----------
KEENE           22-JUN-03  NOON               85.1
KEENE           23-SEP-03  NOON               99.8
KEENE           23-SEP-03  MIDNIGHT           82.6
KEENE           21-MAR-03  PRECIP              4.4
KEENE           22-JUN-03  PRECIP              1.3
KEENE           22-DEC-03  PRECIP              3.9
```

如果使用 first 会发生什么情况？

```
delete from COMFORT_TEST;
commit;

insert FIRST
       when Noon > 80 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'NOON', Noon)
       when Midnight > 70 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'MIDNIGHT', Midnight)
       when Precipitation is not null then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'PRECIP', Precipitation)
select City, SampleDate, Noon, Midnight, Precipitation
  from COMFORT
where City = 'KEENE';

4 rows created.
```

在此例中，插入了 4 行：

```
select * from COMFORT_TEST;

CITY           SAMPLEDAT  MEASURE          VALUE
-------------- ---------  ---------- ----------
KEENE          21-MAR-03  PRECIP              4.4
KEENE          22-DEC-03  PRECIP              3.9
KEENE          22-JUN-03  NOON               85.1
```

```
KEENE          23-SEP-03  NOON              99.8
```

MIDNIGHT 值怎么了？唯一满足 MIDNIGHT 值的 when 子句：

```
    when Midnight > 70 then
```

该记录也满足 NOON 值的 when 子句：

```
    when Noon > 80 then
```

因此，其 Noon 值(99.8)被插入。因为使用了 first 关键字，且判定的第一个(Noon)条件为真，所以 Oracle 不对该行的其余条件进行检查。相同的过程也发生在 PRECIP 测量数据中，另一个非空的 Precipitation 值在 Noon 读数为 85.1 的相同记录中。

如果不满足任何条件会怎样呢？为了说明该选项，创建第 3 个表 COMFORT2，该表的结构与 COMFORT 表相同：

```
create table COMFORT2 (
City           VARCHAR2(13) NOT NULL,
SampleDate     DATE NOT NULL,
Noon           NUMBER(3,1),
Midnight       NUMBER(3,1),
Precipitation NUMBER
);
```

现在，我们对所有的城市执行 insert(使用 first 子句)命令，并用一个 else 子句指定不满足条件的任意行都放在 COMFORT2 表中。在这个示例中，更改 when 条件用来限制符合条件的行数：

```
delete from COMFORT_TEST;
delete from COMFORT2;
commit;

insert FIRST
       when Noon > 80 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'NOON', Noon)
       when Midnight > 100 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'MIDNIGHT', Midnight)
       when Precipitation > 100 then
         into COMFORT_TEST (City, SampleDate, Measure, Value)
       values (City, SampleDate, 'PRECIP', Precipitation)
     else
        into COMFORT2
select City, SampleDate, Noon, Midnight, Precipitation
  from COMFORT
where City = 'KEENE';

4 rows created.
```

反馈信息只告诉您创建了多少行，而没有告诉您这些行创建在哪个表中。报告的总数是

针对所有插入命令的。在这里，有两个记录插入到 COMFORT_TEST 表中，另两个记录由于满足 else 条件插入到 COMFORT2 表中：

```
select * from COMFORT_TEST;

CITY            SAMPLEDAT MEASURE          VALUE
------------ --------- ---------- ----------
KEENE           22-JUN-03 NOON               85.1
KEENE           23-SEP-03 NOON               99.8

select * from COMFORT2;

CITY            SAMPLEDAT        NOON    MIDNIGHT  PRECIPITATION
------------ --------- ---------- ---------- -------------
KEENE           21-MAR-03        39.9        -1.2             4.4
KEENE           22-DEC-03        -7.2        -1.2             3.9
```

15.4　delete 命令

从一个表中删除一行或多行要用到 delete 命令，如 15.3 节的示例所示。对于删除指定的行来说，where 子句是至关重要的。如果没有 where 子句，则 delete 操作将完全清空表。以下命令删除了 COMFORT 表中的 Walpole 项：

```
delete from COMFORT where City = 'WALPOLE';
```

当然，delete 语句中的 where 子句(就像在 update 中或作为 insert 语句一部分的 select 中的 where 一样)，可以包含和任意 select 语句一样多的逻辑，并可包含子查询、并、交等运算。虽然可以回滚一个错误的 insert、update 或 delete 操作，但是在实际更改数据库之前，您应当用 select 语句试验一下，以确保所做的工作是正确的。

既然已经删除了满足 City=‘WALPOLE’的行，就可以用一个简单的查询来测试 delete 的效果：

```
select * from COMFORT
 where City = 'WALPOLE';

no rows selected
```

现在，回滚此 delete 操作并运行同样的查询：

```
rollback;
Rollback complete

select * from COMFORT
 where City = 'WALPOLE';

CITY            SAMPLEDAT        NOON    MIDNIGHT  PRECIPITATION
------------ --------- ---------- ---------- -------------
WALPOLE         21-MAR-03        56.7        43.8             0
```

```
WALPOLE         22-JUN-03        56.7        43.8           0
WALPOLE         22-JUN-03        56.7        43.8           0
WALPOLE         23-SEP-03        86.3        72.1
WALPOLE         22-DEC-03        -7.2        -1.2         3.9
WALPOLE         22-APR-03        50.1        24.8           0
WALPOLE         27-MAY-03        63.7        33.8           0

7 rows selected.
```

这说明可以进行恢复，就好像 commit 操作没有发生一样。如果上述更改已经提交，则可以用闪回查询(见第 29 章)来检索数据。

删除记录的另一个命令是 truncate，它与 delete 不同。delete 可以提交或回滚删除操作，而 truncate 自动地删除表中的所有记录。truncate 命令的操作结果不能被回滚或提交，截除的记录不能被恢复。不能靠执行闪回查询查看被截除的数据。关于 truncate 命令的详细内容，请参考附录 A。

15.5 update 命令

update 需要为要更改的每一列设置指定的值，并且通过精心构造 where 子句来指定应当修改哪一行或哪几行：

```
update COMFORT set Precipitation = .5, Midnight = 73.1
  where City = 'KEENE'
    and SampleDate = '22-DEC-03';

1 row updated.
```

结果显示在 22-DEC-03 记录中：

```
select * from COMFORT
  where City = 'KEENE';

CITY            SAMPLEDAT       NOON    MIDNIGHT  PRECIPITATION
------------    ---------  ---------  ---------  -------------
KEENE           21-MAR-03        39.9       -1.2            4.4
KEENE           22-JUN-03        85.1       66.7            1.3
KEENE           23-SEP-03        99.8       82.6
KEENE           22-DEC-03        -7.2       73.1             .5

4 rows selected.
```

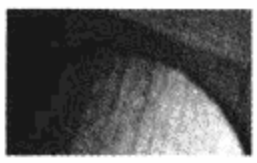
若后来发现 Keene 所使用的温度计报告的温度始终高 1°C 该怎么办？您可以进行计算，可使用串函数以及用于设置 update 值的几乎所有合法函数(就像 insert 或者 delete 命令的 where 子句那样)。下面，Keene 测量的每个温度值都减少了 1°C：

```
update COMFORT set Midnight = Midnight - 1, Noon = Noon - 1
```

```
 where City = 'KEENE';

4 rows updated.
```

下面是 update 的结果：

```
select * from COMFORT
 where City = 'KEENE';

CITY           SAMPLEDAT        NOON    MIDNIGHT PRECIPITATION
-------------- --------- ---------- ---------- -------------
KEENE          21-MAR-03        38.9       -2.2           4.4
KEENE          22-JUN-03        84.1       65.7           1.3
KEENE          23-SEP-03        98.8       81.6
KEENE          22-DEC-03        -8.2       72.1            .5
```



注意：

如果 update 违反列定义或约束就会失败。在本例中，设置 Noon 或 Midnight 为 100 或更大的值将违反列的数值范围定义。

像 delete 一样，where 子句是至关重要的。如果没有它，则数据库表中的每一行都将被更新。如果 where 子句用得不当，所更新的行就不是用户想更新的，这通常很难发现或修正，尤其是在所做的工作已经被提交了的情况下。在进行 update 操作时始终要进行 set feedback on 操作，检查反馈信息，以确保要更新的行数与您的预期相符。在 update 操作之后对查询这些行，查看所发生的更改是否符合预期的要求。

15.5.1　用嵌入式 select 进行更新

可以在 update 语句中间嵌入 select 语句来设定值。请注意，这个 select 语句有其自己的 where 子句，用于从 WEATHER 表中选取 MANCHESTER 城市的温度，并且 update 语句有其自己的 where 子句，它只在某一天对 Keene 城市产生影响：

```
update COMFORT set Midnight =
        (select Temperature
           from WEATHER
          where City = 'MANCHESTER')
  where City = 'KEENE'
    and SampleDate = '22-DEC-03';

1 row updated.
```

下面显示的是 update 命令的结果：

```
select * from COMFORT
  where City = 'KEENE';

CITY           SAMPLEDAT        NOON    MIDNIGHT PRECIPITATION
-------------- --------- ---------- ---------- -------------
KEENE          21-MAR-03        38.9       -2.2           4.4
```

```
KEENE          22-JUN-03       84.1          65.7           1.3
KEENE          23-SEP-03       98.8          81.6
KEENE          22-DEC-03       -8.2            66            .5

4 rows selected.
```

当在 update 命令中使用子查询时，必须确定该子查询只为每个要更新的记录返回仅一个记录；否则，update 就会失败。相关查询的详细内容请参考第 13 章。

还可以使用一个嵌入的 select 语句同时更新多列。这些列必须放在圆括号中，并用逗号分开，如下所示：

```
update COMFORT set (Noon, Midnight) =
          (select Humidity, Temperature
             from WEATHER
            where City = 'MANCHESTER')
 where City = 'KEENE'
   and SampleDate = '22-DEC-03';

1 row updated.
```

以下显示了查询的结果：

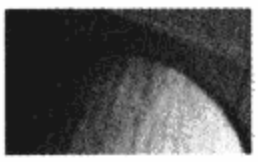
```
select * from COMFORT
 where City = 'KEENE';

CITY           SAMPLEDAT       NOON    MIDNIGHT PRECIPITATION
-------------- --------- ---------- ---------- -------------
KEENE          21-MAR-03       38.9         -2.2           4.4
KEENE          22-JUN-03       84.1         65.7           1.3
KEENE          23-SEP-03       98.8         81.6
KEENE          22-DEC-03         98           66            .5

4 rows selected.
```

15.5.2 用 NULL 更新

还可以更新一个表并设置某一列为 NULL。这是 NULL 与等号而不是与单词 is 一起使用的唯一示例。例如：

```
update COMFORT set Noon = NULL
 where City = 'KEENE'
   and SampleDate = '22-DEC-03';

1 row updated.
```

将把 2003 年 12 月 22 日 Keene 的中午温度设为 NULL。



注意：

使用 insert、update 和 delete 语句时的主要问题是要小心地构造 where 子句，使之只影响(或插入)所要求的行，并在 insert、update 和 delete 操作中正确使用 SQL 函数。在提交工作之前必须先对所做的工作进行仔细检查，这一点极其重要。这些命令使 Oracle 的功能得以扩展，使之不仅可以进行简单的查询，并且可以对数据进行直接操作。

15.6　使用 merge 命令

可以在单个命令中使用 merge 命令对单个表执行 insert 和 update 操作。根据您指定的条件，Oracle 将接收源数据(可以是表、视图或查询)，如果条件满足，则更新已有的值。如果不满足条件，则将插入此行。

例如，更改本章前面创建的 COMFORT2 表中的行：

```
delete from COMFORT2;

insert into COMFORT2

values ('KEENE', '21-MAR-03',55, -2.2, 4.4);
insert into COMFORT2
values ('KEENE', '22-DEC-03',55, 66, 0.5);
  insert into COMFORT2
values ('KEENE', '16-MAY-03', 55, 55, 1);
commit;
```

COMFORT2 表中的数据如下所示：

```
select * from COMFORT2;

CITY           SAMPLEDAT       NOON    MIDNIGHT PRECIPITATION
-------------- --------- ---------- ---------- -------------
KEENE          21-MAR-03          55       -2.2           4.4
KEENE          22-DEC-03          55         66            .5
KEENE          16-MAY-03          55         55             1
```

COMFORT 表中的 Keene 的数据为：

```
select * from COMFORT where City = 'KEENE';

CITY           SAMPLEDAT       NOON    MIDNIGHT  PRECIPITATION
-------------- ---------  ---------- ---------- -------------
KEENE           21-MAR-03       38.9       -2.2           4.4
KEENE           22-JUN-03       84.1       65.7           1.3
KEENE           23-SEP-03       98.8       81.6
KEENE           22-DEC-03                    66            .5
```

用 COMFORT2 表作为数据源，现在可以在 COMFORT 表中执行 merge 操作。对于匹配 21-MAR-03 和 22-DEC-03 项的行来说，将更新 COMFORT 表中的 Noon 值。仅存在于 COMFORT2 中的行(16-MAY-03 行)将插入到 COMFORT 表中。以下程序清单显示了该命令的使用方法：

```
merge into COMFORT C1
    using (select City, SampleDate, Noon, Midnight,
                Precipitation from COMFORT2) C2
      on (C1.City = C2.City and C1.SampleDate=C2.SampleDate)
     when matched then
          update set Noon = C2.Noon
     when not matched then
          insert (C1.City, C1.SampleDate, C1.Noon,
                  C1.Midnight, C1.Precipitation)
          values (C2.City, C2.SampleDate, C2.Noon,
                  C2.Midnight, C2.Precipitation);

    3 rows merged.
```

虽然从该输出可以看出在源表中处理的行数，但是它没有说明您插入或更新了多少行。可以通过查询 COMFORT 表查看这些变化(请注意 Noon = 55 的记录)：

```
select * from COMFORT where City = 'KEENE';
```

CITY	SAMPLEDAT	NOON	MIDNIGHT	PRECIPITATION
KEENE	16-MAY-03	55	55	1
KEENE	21-MAR-03	55	-2.2	4.4
KEENE	22-JUN-03	84.1	65.7	1.3
KEENE	23-SEP-03	98.8	81.6	
KEENE	22-DEC-03	55	66	.5

看看此命令是如何完成的。在第一行，命名目标表并给出别名 C1：

```
merge into COMFORT    C1
```

在接下来的两行中，using 子句提供了 merge 操作的源数据，其别名为 C2：

```
using (select City, SampleDate, Noon, Midnight,
                Precipitation from COMFORT2) C2
```

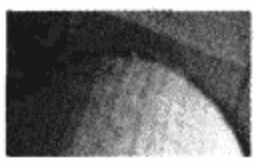
然后，在 on 子句中指定合并的条件。如果源数据的 City 和 SampleDate 的值与目标表中的相应值相匹配，则更新该数据。update 命令前面的 when matched then 子句告诉 Oracle 要更新源表中的哪些列：

```
  on (C1.City = C2.City and C1.SampleDate=C2.SampleDate)
 when matched then
      update set Noon = C2.Noon
```

如果不匹配，则应该插入该行，即在 when not matched 子句中指定的内容：

```
when not matched then
        insert (C1.City, C1.SampleDate, C1.Noon,
                C1.Midnight, C1.Precipitation)
        values (C2.City, C2.SampleDate, C2.Noon,
                C2.Midnight, C2.Precipitation);
```

对于要从一个源表中插入和更新许多行的操作，可以使用 merge 命令进行简化。与所有 update 操作一样，应该非常小心地在 merge 命令的 using 子句中使用恰当的 where 子句。



注意：
不能更新 on 条件子句中引用的列。

在 merge 命令中，可以指定一个 update 子句，一个 insert 子句，或者同时指定这两个子句。

为了把所有的源数据行插入到表中，可以在 on 子句的条件中使用一个常量过滤器(constant filter)谓词。一个关于常量过滤器谓词的示例就是 on (0=1)。Oracle 能识别这样的谓词，可以将所有的源数据行无条件地插入到表中。如果存在一个常量过滤器谓词，则在 merge 命令执行期间最好不要执行任何一种连接。

merge 命令的 update 部分可以包含一个 delete 子句。delete 子句(有其自己的 where 子句)可以删除目标表中被 merge 更新的行。delete…where 子句可以计算更新值，而不是目标表中的初始值。如果目标表中的行符合 delete…where 条件但不在 merge 所作用(就如 on 条件所定义的)的行集范围内，就不会被删除。以下是一个这样的示例：

```
merge into COMFORT C1
    using (select City, SampleDate, Noon, Midnight,
                  Precipitation from COMFORT2) C2
       on (C1.City = C2.City and C1.SampleDate=C2.SampleDate)
     when matched then
          update set Noon = C2.Noon
          delete where ( Precipitation is null )
     when not matched then
          insert (C1.City, C1.SampleDate, C1.Noon,
                  C1.Midnight, C1.Precipitation)
          values (C2.City, C2.SampleDate, C2.Noon,
                  C2.Midnight, C2.Precipitation);
```

显然，这些命令很快会变得非常复杂。在该示例中，对更新的行执行有条件的 update 和无条件的 delete，同时对其他行执行有条件的 insert。可以将很多复杂的业务逻辑合并到单个命令中，充分增强标准 insert、update 和 delete 命令的功能。

15.7 处理错误

默认情况下，单个事务中的所有行要么成功要么失败。如果在单个 insert as select 操作中插入大量的行，那么一行中的错误将导致整个插入操作失败。您可以避免发生这种行为，方法是使用错误日志表，并在命令中使用特定语法告诉数据库如何处理错误。之后 Oracle 会自动将失败条目以及它们的失败原因记录到日志中，尽管事务本身会成功。在某种意义上，这是一个多表插入——可插入的行被添加到目标表中，而失败的行被插入到错误日志表中。

为了使用这种方法，必须先创建一个错误日志表，它的结构基于您将要更新的表。Oracle 在 DBMS_ERRLOG 程序包中提供了一个名为 CREATE_ERROR_LOG 的过程，此过程为每个指定的表创建一个错误日志。CREATE_ERROR_LOG 过程的格式是：

```
DBMS_ERRLOG.CREATE_ERROR_LOG (
    dml_table_name                IN VARCHAR2,
    err_log_table_name            IN VARCHAR2 := NULL,
    err_log_table_owner           IN VARCHAR2 := NULL,
    err_log_table_space           IN VARCHAR2 := NULL,
    skip_unsupported              IN BOOLEAN := FALSE);
```

CREATE_ERROR_LOG 过程的参数如表 15-1 表示。

表 15-1 CREATE ERROR_LOG 过程的参数

参 数	描 述
dml_table_name	在其上创建错误日志表的那个表的名称。此名称可以被完全限定(例如，BOOKSHELF 或 PRACTICE.BOOKSHELF)。如果名称用双引号引起来，则它将是大写字母形式
err_log_table_name	将要创建的错误日志表的名称。默认名称是 DML 表名的前 25 个字符，且使用“ERR$_”前缀(如 ERR$_BOOKSHELF)
err_log_table_owner	错误日志表的所有者的名称。您可以在 dml_table_name 参数中指定此所有者。否则，使用当前连接用户的模式
err_log_table_space	将要在其中创建错误日志表的表空间。如果未指定此参数，则使用拥有错误日志表的用户的默认表空间
skip_unsupported	当此参数设置为 TRUE 时，忽略错误日志不支持的列类型，且不添加到错误日志表中。当此参数设置为 FALSE(默认设置)时，不支持的列类型将会导致该过程终止

对于 BOOKSHELF 表，可以使用下面的命令创建一个错误日志表：

```
execute DBMS_ERRLOG.CREATE_ERROR_LOG('bookshelf','errlog');
```

当将行插入到 BOOKSHELF 表中时，可以使用 log errors 子句对错误进行重定向：

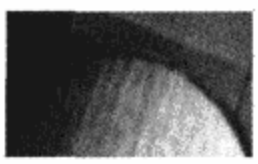
```
insert into BOOKSHELF (Title, Publisher, CategoryName)
select * from BOOR_ORDER
log errors into errlog ('fromorder') reject limit 10;
```

可以使用 reject limit 子句来指定在 insert 命令失败之前可以记录到日志中的错误数量的上限。默认的 reject limit 是 0，也可以将它设置为 unlimited。

如果在 insert 执行期间遇到错误，那么这些错误将会被写到 ERRLOG 表中。并按照程序清单中 log errors 子句的指定，用“fromorder”值标记它们。除 BOOKSHELF 表列之外，ERRLOG 表还有另外两列：

- ORA_ERR_MESG$：它包含 Oracle 错误消息号和错误的消息文本。
- ORA_ERR_TAG$：它包含 into 子句中指定的标记，如前面的程序清单所示。

在 ERRLOG 表中，对每一个错误行，都会有一个相应的条目。整个错误行的值将存储在 ERRLOG 表中。



注意：
此命令的标记可以是字面量、数字、绑定变量、或函数表达式，如 TO_CHAR(SYSDATE)。

如果有错误，且插入操作完成，则 Oracle 不会返回任何错误代码。您需要检查错误日志表来确定在插入期间是否有失败的行。

对于下面的条件，如果没有调用错误日志功能，则命令将会失败：

- 违反延迟约束。
- 导致违反唯一约束或唯一索引的任何直接路径 insert 或 merge 操作。
- 导致违反唯一约束或唯一索引的任何 update 或 merge 操作。

不能在错误日志表中记录 LONG、LOB 或对象类型列的错误。但是，DML 操作的目标表可以包含这些类型的列。

第16章

DECODE和CASE：SQL中的if-then-else

DECODE函数无疑是Oracle的SQL中功能最强大的函数之一。它也是Oracle对标准SQL语言增加的几个扩展函数之一。本章将探讨使用DECODE的大量方法，包括“交叉表”报表的生成方法。您还可以在SQL语句中使用CASE函数和COALESCE函数执行复杂的逻辑测试。

DECODE和CASE常常用来旋转数据——也就是说将数据行转换为报表的列。本章将介绍如何使用DECODE、CASE和Oracle 11g中新引入的PIVOT运算符生成交叉表报表。

16.1 if-then-else

在程序设计和逻辑中，问题的常见结构为“if-then-else”这样的模式。例如，if(如果)今天是星期六，then(则)Adah 应该在家里；if 今天是星期天，then Adah 将去她祖父家；if 今天是假期，then Adah 将回到姑母 Dora 家；else(否则)，Adah 应该去学校。在每种情况下，都要对“今天”进行测试，如果它属于特定日期列表中的某一天，则产生相应的结果，否则(如果它不是这些特定日期中的某天)，将产生另一种结果。DECODE 函数遵循这种逻辑。第 11 章初步介绍了 DECODE 函数的基本结构和用途。

DECODE 函数的格式为：

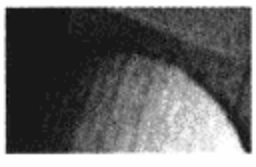
```
DECODE(value, if1, then1, if2, then2, if3, then3,. . . , else)
```

其中，value 表示表中的任意一列(不考虑数据类型)或者任意一个计算结果(如一个日期减去另一个日期、字符列的 SUBSTR、一个数乘以另一个数等)。每一行都对 value 进行测试，如果 value 符合条件 if1，则 DECODE 的结果为 then1；如果 value 符合条件 if2，则 DECODE 的结果为 then2，等等。事实上，可以构造多对 if-then。如果 value 与任意一个 if 都不符，则 DECODE 的结果为 else。每个 if、then 和 else 都可以是表列或者一个函数或计算的结果。最多可以在圆括号中包含 255 个元素。

我们考虑一下书架的借阅历史记录，这些历史记录保存在 BOOKSHELF_CHECKOUT 表中：

```
column Name format a16
column Title format a24 word_wrapped
set pagesize 60

select * from BOOKSHELF_CHECKOUT;

NAME             TITLE                    CHECKOUTD  RETURNEDD
---------------- ------------------------ ---------  ---------
JED HOPKINS      INNUMERACY               01-JAN-02  22-JAN-02
GERHARDT         KENTGEN WONDERFUL LIFE   02-JAN-02  02-FEB-02
DORAH TALBOT     EITHER/OR                02-JAN-02  10-JAN-02
EMILY TALBOT     ANNE OF GREEN GABLES     02-JAN-02  20-JAN-02
PAT LAVAY        THE SHIPPING NEWS        02-JAN-02  12-JAN-02
ROLAND BRANDT    THE SHIPPING NEWS        12-JAN-02  12-MAR-02
ROLAND BRANDT    THE DISCOVERERS          12-JAN-02  01-MAR-02
ROLAND BRANDT    WEST WITH THE NIGHT      12-JAN-02  01-MAR-02
EMILY TALBOT     MIDNIGHT MAGIC           20-JAN-02  03-FEB-02
EMILY TALBOT     HARRY POTTER AND THE     03-FEB-02  14-FEB-02
                 GOBLET OF FIRE
PAT LAVAY        THE MISMEASURE OF MAN    12-JAN-02  12-FEB-02
DORAH TALBOT     POLAR EXPRESS            01-FEB-02  15-FEB-02
DORAH TALBOT     GOOD DOG, CARL           01-FEB-02  15-FEB-02
GERHARDT KENTGEN THE MISMEASURE OF MAN    13-FEB-02  05-MAR-02
FRED FULLER      JOHN ADAMS               01-FEB-02  01-MAR-02
```

```
FRED FULLER       TRUMAN                    01-MAR-02  20-MAR-02
JED HOPKINS       TO KILL A MOCKINGBIRD     15-FEB-02  01-MAR-02
DORAH TALBOT      MY LEDGER                 15-FEB-02  03-MAR-02
GERHARDT KENTGEN  MIDNIGHT MAGIC            05-FEB-02  10-FEB-02
```



注意：
由于自动换行，因此输出中可能包含一个额外的空白行。为了增加可读性，额外的空白行在这里不显示。

当您浏览借阅列表时，会看到有些书借出很长时间了。可以按借出的天数对列表排序，以突出显示借书时间最长的读者：

```
select Name, Title, ReturnedDate-CheckOutDate DaysOut
  from BOOKSHELF_CHECKOUT
 order by DaysOut desc;
```

```
NAME              TITLE                          DAYSOUT
----------------- ------------------------------ ----------
ROLAND BRANDT     THE SHIPPING NEWS                      59
ROLAND BRANDT     THE DISCOVERERS                        48
ROLAND BRANDT     WEST WITH THE NIGHT                    48
GERHARDT KENTGEN  WONDERFUL LIFE                         31
PAT LAVAY         THE MISMEASURE OF MAN                  31
FRED FULLER       JOHN ADAMS                             28
JED HOPKINS       INNUMERACY                             21
GERHARDT KENTGEN  THE MISMEASURE OF MAN                  20
FRED FULLER       TRUMAN                                 19
EMILY TALBOT      ANNE OF GREEN GABLES                   18
DORAH TALBOT      MY LEDGER                              16
EMILY TALBOT      MIDNIGHT MAGIC                         14
DORAH TALBOT      POLAR EXPRESS                          14
DORAH TALBOT      GOOD DOG, CARL                         14
JED HOPKINS       TO KILL A MOCKINGBIRD                  14
EMILY TALBOT      HARRY POTTER AND THE                   11
                  GOBLET OF FIRE
PAT LAVAY         THE SHIPPING NEWS                      10
DORAH TALBOT      EITHER/OR                               8
GERHARDT KENTGEN  MIDNIGHT MAGIC                          5
```

但是，出现这类问题是由于读者自身的原因，还是因为某些书籍的阅读的确要花较长时间呢？其中涉及多种因素，如果孤立地看待它们就会导致不正确的判断。那么，每种类别的图书的平均借出时间是多少呢？

```
select B.CategoryName,
       MIN(BC.ReturnedDate-BC.CheckOutDate) MinOut,
       MAX(BC.ReturnedDate-BC.CheckOutDate) MaxOut,
       AVG(BC.ReturnedDate-BC.CheckOutDate) AvgOut
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title
group by B.CategoryName
```

```
order by B.CategoryName;

CATEGORYNAME              MINOUT     MAXOUT       AVGOUT
------------------- ---------- ----------   ----------
ADULTFIC                    10         59   27.6666667
ADULTNF                     16         48   29.1111111
ADULTREF                     8          8            8
CHILDRENFIC                  5         18           12
CHILDRENPIC                 14         14           14
```

虽然上述数据更有用，但是它不能反映不同的人借阅各类图书的情况。为了反映这一点，虽然可以使用另一个分组，但是创建一个交叉报表结果将更容易使用。以下程序清单生成了一个显示每个图书分类中每个人的最短、最长和平均借阅时间的报表。该查询使用DECODE函数进行计算。为说明这个示例的目的使用了3个借阅者：

```
column CategoryName format a11

select B.CategoryName,
       MAX(DECODE(BC.Name, 'FRED FULLER',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxFF,
       AVG(DECODE(BC.Name, 'FRED FULLER',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgFF,
       MAX(DECODE(BC.Name, 'DORAH TALBOT',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxDT,
       AVG(DECODE(BC.Name, 'DORAH TALBOT',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgDT,
       MAX(DECODE(BC.Name, 'GERHARDT KENTGEN',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxGK,
       AVG(DECODE(BC.Name, 'GERHARDT KENTGEN',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgGK
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title
group by B.CategoryName
order by B.CategoryName;

CATEGORYNAM      MAXFF      AVGFF      MAXDT      AVGDT      MAXGK      AVGGK
----------- ---------- ----------  --------  ---------- -------- ---------
ADULTFIC
ADULTNF              28       23.5        16         16         31      25.5
ADULTREF                                  8          8
CHILDRENFIC                                                     5         5
CHILDRENPIC                              14         14
```

现在，该输出在表的顶部显示借阅者，Fred Full的最长借阅时间是MaxFF列，Dorah Talbot的最长借阅时间是MaxDT列，而Gerhardt Kentgen的最长借阅时间是MaxGK列。该输出说明，对于每个借阅者，AdultNF类图书的平均借阅时间最长，其中Gerhardt Kentgen的平均借阅时间明显大于他借阅其他类图书的平均时间。

这个报表是如何生成的？在该查询中是通过CatrgoryName进行分组的。MaxFF列的查

询如下所示：

```
select B.CategoryName,
       MAX(DECODE(BC.Name, 'FRED FULLER',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxFF,
```

DECODE 正对数据执行 if-then 测试。如果某行的 BC.Name 列的值为 FRED FULLER，则计算 ReturnedDate 和 CheckOutDate 之差；否则返回一个 NULL 值。然后计算该列表的值并返回最大值。返回 Fred Fuller 的平均借出时间的操作方法也类似：

```
AVG(DECODE(BC.Name, 'FRED FULLER',
       BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgFF,
```

16.2　通过 DECODE 替换值

在上例中，DECODE 依据借书人的 Name 有条件地返回值。还可以使用 DECODE 替换一个列表中的值。例如，从 BOOKSHELF 表中选择分类名将产生以下结果：

```
select distinct CategoryName from BOOKSHELF
  order by CategoryName;

CATEGORYNAM
-----------
ADULTFIC
ADULTNF
ADULTREF
CHILDRENFIC
CHILDRENNF
CHILDRENPIC
```

为了替换这些名字，可以将 BOOKSHELF 表与 CATEGORY 表连接，并从 CATEGORY 表中选择 ParentCategory 和 SubCategory。如果分类的列表是静态的，则可以不用与 CATEGORY 表连接，并通过 DECODE 函数进行替换，如以下程序清单所示。请注意 DECODE 是如何在一个调用中支持多个 if-then 组合的。

```
select distinct
       DECODE(CategoryName,'ADULTFIC','Adult Fiction',
                           'ADULTNF','Adult Nonfiction',
                           'ADULTREF','Adult Reference',
                           'CHILDRENFIC','Children Fiction',
                           'CHILDRENNF','Children Nonfiction',
                           'CHILDRENPIC','Children Picturebook',
                           CategoryName)
 from BOOKSHELF
order by 1;

DECODE(CATEGORYNAME,
```

```
--------------------
Adult Fiction
Adult Nonfiction
Adult Reference
Children Fiction
Children Nonfiction
Children Picturebook
```

在本例中，数据是静态的。对于动态的数据，把替换值硬编码为应用程序的代码是不可行的。这里介绍的技术适用于调用最少的数据库的事务处理系统。在本例中，没有将 ADULTNF 更改为 Adult Nonfiction 的数据库访问，该更改出现在 DECODE 函数的调用中。请注意，如果还发现任何其他的种类，则 DECODE 函数中的 else 条件返回原始的 CategoryName 列的值。

16.3 DECODE 中的 DECODE

可以在一个 DECODE 函数调用中调用另一个 DECODE 函数。假设您想收取滞纳金，并且不同种类的书籍有不同的滞纳金比率。虽然 Adult 类的图书借阅的时间可以较长，但对逾期不还的罚款也较高。

对借阅时间大于 14 天的书籍，开始时规定每天的基本收费为 0.20 美元：

```
column Name format a16
column Title format a20 word_wrapped
column DaysOut format 999.99 heading 'Days Out'
column DaysLate format 999.99 heading 'Days Late'
set pagesize 60
break on Name

select Name, Title, ReturnedDate,
       ReturnedDate-CheckoutDate as DaysOut /*Count days*/,
       ReturnedDate-CheckoutDate -14 DaysLate,
       (ReturnedDate-CheckoutDate -14)*0.20 LateFee
  from BOOKSHELF_CHECKOUT
where ReturnedDate-CheckoutDate > 14
  order by Name, CheckoutDate;
```

NAME	TITLE	RETURNEDD	Days Out	Days Late	LATEFEE
DORAH TALBOT	MY LEDGER	03-MAR-02	16.00	2.00	.4
EMILY TALBOT	ANNE OF GREEN GABLES	20-JAN-02	18.00	4.00	.8
FRED FULLER	JOHN ADAMS	01-MAR-02	28.00	14.00	2.8
	TRUMAN	20-MAR-02	19.00	5.00	1
GERHARDT KENTGEN	WONDERFUL LIFE	02-FEB-02	31.00	17.00	3.4
	THE MISMEASURE OF MAN	05-MAR-02	20.00	6.00	1.2
JED HOPKINS	INNUMERACY	22-JAN-02	21.00	7.00	1.4

```
PAT LAVAY          THE MISMEASURE OF     12-FEB-02     31.00       17.00         3.4
                   MAN
ROLAND BRANDT      THE DISCOVERERS       01-MAR-02     48.00       34.00         6.8
                   THE SHIPPING NEWS     12-MAR-02     59.00       45.00           9
                   WEST WITH THE NIGHT   01-MAR-02     48.00       34.00         6.8
```

Adult 类图书的借阅时间延长到 21 天。虽然这样不更改借出的天数，但是要更改 DaysLate 列的计算结果。由于 CategoryName 不是 BOOKSHELF_CHECKOUT 表中的列，因此该修改还需要将 BOOKSHELF 表添加到 from 子句中。

```
select BC.Name, BC.Title, BC.ReturnedDate,
       BC.ReturnedDate-BC.CheckoutDate as DaysOut /*Count days*/,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          BC.ReturnedDate-BC.CheckoutDate -21,
          BC.ReturnedDate-BC.CheckoutDate -14 ) DaysLate,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.30,
          (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title
  and BC.ReturnedDate-BC.CheckoutDate >
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',21,14)
order by BC.Name, BC.CheckoutDate;

NAME               TITLE                    RETURNEDD Days Out Days Late    LATEFEE
---------------    ---------------------    --------- -------- ---------    ---------
EMILY TALBOT       ANNE OF GREEN GABLES     20-JAN-02     18.00       4.00          .8
FRED FULLER        JOHN ADAMS               01-MAR-02     28.00       7.00         2.1
GERHARDT KENTGEN   WONDERFUL LIFE           02-FEB-02     31.00      10.00           3
PAT LAVAY          THE MISMEASURE OF        12-FEB-02     31.00      10.00           3
MAN
ROLAND BRANDT      THE DISCOVERERS          01-MAR-02     48.00      27.00         8.1
                   WEST WITH THE NIGHT      01-MAR-02     48.00      27.00         8.1
                   THE SHIPPING NEWS        12-MAR-02     59.00      38.00        11.4
```

在 select 子句中，DaysLate 列的查询逻辑如下所示：

```
DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
       BC.ReturnedDate-BC.CheckoutDate -21,
       BC.ReturnedDate-BC.CheckoutDate -14 ) DaysLate
```

依据 DECODE 的 if-then-else 格式，该例表示：“如果 CategoryName 列的前 5 个字符与

现在，增加另外一个规则：对于 Adult 小说类的书籍，其滞纳金是其他 Adult 类书籍的两倍。在上一个查询中，LateFee 的计算为：

```
DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
   (BC.ReturnedDate-BC.CheckoutDate -21)*0.30,
   (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
```

新的规则要求在 DECODE 函数中有一个附加的种类检查。因此，新的 LateFee 的计算如下所示：

```
DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
   DECODE(SUBSTR(CategoryName,6,3),'FIC',
   (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
   (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
   (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
```

在上述程序清单中，第一个 DECODE 函数告诉 Oracle，如果 CategoryName 的前 5 个字母为 ADULT，则执行第二个 DECODE 函数。第二个 DECODE 告诉 Oracle，如果 CategoryName 的第 6～8 个字母为 FIC，则 21 天后每天的滞纳金为 0.60 美元；否则，21 天后每天的滞纳金为 0.30 美元。到此为止，内层的 DECODE 函数完成。对于第一个 DECODE 函数，else 子句(对于非 ADULT 类的图书)指定滞纳金的计算。查询和结果如以下程序清单所示。通过比较该报表和上一个报表中标题为 THE SHOPPING NEWS 的 LatelFee 列，可以看到其效果。

```
select BC.Name, BC.Title, BC.ReturnedDate,
       BC.ReturnedDate-BC.CheckoutDate as DaysOut /*Count days*/,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          BC.ReturnedDate-BC.CheckoutDate -21,
          BC.ReturnedDate-BC.CheckoutDate -14 ) DaysLate,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          DECODE(SUBSTR(CategoryName,6,3),'FIC',
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
          (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title
  and BC.ReturnedDate-BC.CheckoutDate >
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',21,14)
order by BC.Name, BC.CheckoutDate;

NAME             TITLE                    RETURNEDD  Days Out  Days Late LATEFEE
---------------- ------------------------ ---------  --------  --------- -------
EMILY TALBOT     ANNE OF GREEN GABLES     20-JAN-02     18.00      4.00      .8
FRED FULLER      JOHN ADAMS               01-MAR-02     28.00      7.00     2.1
GERHARDT KENTGEN WONDERFUL LIFE           02-FEB-02     31.00     10.00       3
PAT LAVAY        THE MISMEASURE OF MAN    12-FEB-02     31.00     10.00       3
ROLAND BRANDT    THE DISCOVERERS          01-MAR-02     48.00     27.00     8.1
                 WEST WITH THE NIGHT      01-MAR-02     48.00     27.00     8.1
                 THE SHIPPING NEWS        12-MAR-02     59.00     38.00    22.8
```

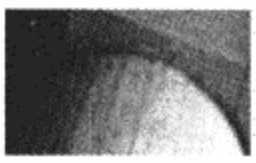
可以在一个 DECODE 中嵌套另一个 DECODE，以支持数据处理中的复杂逻辑。例如，可以选择打印具有非 NULL 值的列，并且如果第一列为 NULL 则打印第二列。如果使用 DECODE，就是一个简单的函数调用：

```
DECODE(Column1, NULL, Column2, Column1)
```

如果 Column1 为 NULL，则返回 Column2；否则，返回 Column1 的非 NULL 值。还可以使用 NVL 或 COALESCE 和 NULLIF 函数执行类似的逻辑。

COALESCE 将返回在值列表中遇到的第一个非 NULL 值。最后一个 DECODE 示例可重写为：

```
COALESCE(Column1, Column2)
```



注意：
虽然 COALESCE 可以有两个以上的参数，但 NVL 不能。

由于 COALESCE 的名字不能清楚地表达其用法，因此必须在代码中解释所执行的逻辑测试。

16.4　DECODE 中的大于和小于

虽然 DECODE 支持逻辑检查，但是如何用此格式进行数值比较呢？最简单的方法是使用 SIGN 函数。如果数值为正数，则 SIGN 返回 1；如果数值为 0，则 SIGN 返回 0；如果数值为负数，则 SIGN 返回 - 1。由于 SIGN 的操作对象是数值，因此可以对返回一个数值(包括日期运算)的任何函数求值。

让我们再次修改 LateFee 的业务规则。利用同样的基本计算对结果进行修改，我们将不收取小于或等于 4.00 美元的滞纳金。以下是原来的 LateFee 的计算：

```
DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
     DECODE(SUBSTR(CategoryName,6,3),'FIC',
     (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
     (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
     (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
```

如果该值小于 4，则将返回 0；否则，将返回计算值。为了实现该要求，要从 LateFee 的值中减去 4。如果结果为正(即 SIGN 值为 1)，则返回该结果；否则，返回 0。

```
DECODE(SIGN(
    DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
        DECODE(SUBSTR(CategoryName,6,3),'FIC',
        (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
        (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
        (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) -4),
    1,
```

```
    DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
       DECODE(SUBSTR(CategoryName,6,3),'FIC',
       (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
       (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
       (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ),
    0) LateFee
```

从一个简单的基础开始构造，这一系列 DECODE 函数调用能帮您完成用于 LateFee 计算的非常复杂的逻辑。第一个 DECODE 计算下一个 DECODE 函数的结果减去 4 的 SIGN。如果该值为正，则返回计算出的滞纳金；否则，返回 0。以下程序清单显示了该计算以及输出结果。

```
select BC.Name, BC.Title, BC.ReturnedDate,
       BC.ReturnedDate-BC.CheckoutDate as DaysOut /*Count days*/,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          BC.ReturnedDate-BC.CheckoutDate -21,
          BC.ReturnedDate-BC.CheckoutDate -14 ) DaysLate,
DECODE(SIGN(
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          DECODE(SUBSTR(CategoryName,6,3),'FIC',
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
          (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) -4),
        1,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          DECODE(SUBSTR(CategoryName,6,3),'FIC',
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.60,
          (BC.ReturnedDate-BC.CheckoutDate -21)*0.30),
          (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ),
       0) LateFee
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
where BC.Title = B.Title
  and BC.ReturnedDate-BC.CheckoutDate >
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',21,14)
order by BC.Name, BC.CheckoutDate;
```

NAME	TITLE	RETURNEDD	Days Out	Days Late	LATEFEE
EMILY TALBOT	ANNE OF GREEN GABLES	20-JAN-02	18.00	4.00	0
FRED FULLER	JOHN ADAMS	01-MAR-02	28.00	7.00	0
GERHARDT KENTGEN	WONDERFUL LIFE	02-FEB-02	31.00	10.00	0
PAT LAVAY	THE MISMEASURE OF MAN	12-FEB-02	31.00	10.00	0
ROLAND BRANDT	THE DISCOVERERS	01-MAR-02	48.00	27.00	8.1
	WEST WITH THE NIGHT	01-MAR-02	48.00	27.00	8.1
	THE SHIPPING NEWS	12-MAR-02	59.00	38.00	22.8

可以通过对 where 子句作相同的修改删除前 4 行的显示(滞纳金为 0 美元)。

16.5　使用 CASE

可以用 CASE 函数代替 DECODE。CASE 函数使用关键字 when、then、else 和 end 说明后面的逻辑路径。通常，与使用等价的 DECODE 相比，虽然使用 CASE 函数的 SQL 显得比较冗长，但可以使程序可读性强，而且更容易维护。

考虑本章前面的一个简单的 DECODE 示例。

```
select distinct
       DECODE(CategoryName,'ADULTFIC','Adult Fiction',
                           'ADULTNF','Adult Nonfiction',
                           'ADULTREF','Adult Reference',
                           'CHILDRENFIC','Children Fiction',
                           'CHILDRENNF','Children Nonfiction',
                           'CHILDRENPIC','Children Picturebook',
                           CategoryName)
  from BOOKSHELF;
```

等价的 CASE 函数为：

```
select distinct
       CASE CategoryName
         when 'ADULTFIC' then 'Adult Fiction'
         when 'ADULTNF' then 'Adult Nonfiction'
         when 'ADULTREF' then 'Adult Reference'
         when 'CHILDRENFIC' then 'Children Fiction'
         when 'CHILDRENNF' then 'Children Nonfiction'
         when 'CHILDRENPIC' then 'Children Picturebook'
         else CategoryName
       end
  from BOOKSHELF
 order by 1;

CASECATEGORYNAMEWHEN
--------------------
Adult Fiction
Adult Nonfiction
Adult Reference
Children Fiction
Children Nonfiction
Children Picturebook
```

CASE 判定第一个 when 子句，如果满足限定的条件，则返回一个匹配值。与 DECODE 相同，您可以嵌套 CASE 函数调用。那么，如何用 CASE 完成 LateFee 列的计算呢？

DECODE 函数从计算 ADULT 类图书的更高比率的滞纳金开始：

```
  DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
       (BC.ReturnedDate-BC.CheckoutDate -21)*0.30,
```

```
        (BC.ReturnedDate-BC.CheckoutDate -14)*0.20 ) LateFee
```

等价的 CASE 语句是：

```
CASE SUBSTR(CategoryName,1,5)
     when 'ADULT' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
     else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
end
```

第二个规则(ADULTFIC 类的书籍要双倍的滞纳金)需要一个嵌套的 CASE 命令：

```
CASE SUBSTR(CategoryName,1,5)
     when 'ADULT' then
          CASE SUBSTR(CategoryName,6,3)
             when 'FIC' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.60
             else (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
             end
     else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
end
```

虽然这个命令更复杂，但是逻辑更容易理解，通常比等价的 DECODE 子句更容易维护。现在考虑最后的条件，即如果计算得到的滞纳金小于或等于 4 美元，则返回 0。由于我们使用了 CASE 函数，因此不需要使用 SIGN 函数，我们可以使用“<”比较操作作为命令处理的一部分。以下示例说明了如何添加这个检查。可以添加一个额外的 CASE 函数，这个内层的 CASE 函数调用括在圆括号中。结果与 4 美元作比较：

```
CASE when
     (CASE SUBSTR(CategoryName,1,5)
      when 'ADULT' then
          CASE SUBSTR(CategoryName,6,3)
             when 'FIC' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.60
             else (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
             end
      else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
     end)
     < 4 then 0
```

该 CASE 函数调用的 else 子句与内层的 CASE 函数的执行任务相同，即计算滞纳金：

```
else
     CASE SUBSTR(CategoryName,1,5)
      when 'ADULT' then
          CASE SUBSTR(CategoryName,6,3)
             when 'FIC' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.60
             else (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
             end
      else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
   end
end
```

完整的查询以及输出结果如以下程序清单所示：

```
column LateFee format 999.99

select BC.Name, BC.Title, BC.ReturnedDate,
       BC.ReturnedDate-BC.CheckoutDate as DaysOut /*Count days*/,
       DECODE(SUBSTR(CategoryName,1,5), 'ADULT',
          BC.ReturnedDate-BC.CheckoutDate -21,
          BC.ReturnedDate-BC.CheckoutDate -14 ) DaysLate,
CASE when
    (CASE SUBSTR(CategoryName,1,5)
     when 'ADULT' then
        CASE SUBSTR(CategoryName,6,3)
           when 'FIC' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.60
       else (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
     end
  else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
  end)
  < 4 then 0 else
  CASE SUBSTR(CategoryName,1,5)
     when 'ADULT' then
        CASE SUBSTR(CategoryName,6,3)
          when 'FIC' then (BC.ReturnedDate-BC.CheckoutDate -21)*0.60
          else (BC.ReturnedDate-BC.CheckoutDate -21)*0.30
          end
     else (BC.ReturnedDate-BC.CheckoutDate -14)*0.20
    end
  end AS LateFee
    from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
  where BC.Title = B.Title
    and BC.ReturnedDate-BC.CheckoutDate >
         DECODE(SUBSTR(CategoryName,1,5), 'ADULT',21,14)
  order by BC.Name, BC.CheckoutDate;
```

NAME	TITLE	RETURNEDD	Days Out	Days Late	LATEFEE
EMILY TALBOT	ANNE OF GREEN GABLES	20-JAN-02	18.00	4.00	.00
FRED FULLER	JOHN ADAMS	01-MAR-02	28.00	7.00	.00
GERHARDT KENTGEN	WONDERFUL LIFE	02-FEB-02	31.00	10.00	.00
PAT LAVAY	THE MISMEASURE OF MAN	12-FEB-02	31.00	10.00	.00
ROLAND BRANDT	THE DISCOVERERS	01-MAR-02	48.00	27.00	8.10
	WEST WITH THE NIGHT	01-MAR-02	48.00	27.00	8.10
	THE SHIPPING NEWS	12-MAR-02	59.00	38.00	22.80

将 CASE 版本的示例与本章前面的 DECODE 版本的示例进行比较，可以发现虽然 CASE 会令多了 6 行，但是阅读和维护起来更简单。CASE 提供了 DECODE 的另一种功能强大的选

16.6 使用 PIVOT

在 Oracle Database 11g 中，可以使用 PIVOT 运算符和 UNPIVOT 运算符处理交叉表数据。在交叉表报表中，行数据显示为单独的列。例如，在本章前面的内容中，行数据中的值(name)用来填充报表的列。在此查询中，第二列和第三列是 Fred Fuller 列，第四列和第五列是 Dorah Talbot 列，等等。在第二列中，DECODE 函数检查 Name 列的值是否为 FRED FULLER。如果是，则执行计算；否则，返回一个 NULL 值。

```
column CategoryName format a11

select B.CategoryName,
       MAX(DECODE(BC.Name, 'FRED FULLER',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxFF,
       AVG(DECODE(BC.Name, 'FRED FULLER',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgFF,
       MAX(DECODE(BC.Name, 'DORAH TALBOT',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxDT,
       AVG(DECODE(BC.Name, 'DORAH TALBOT',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgDT,
       MAX(DECODE(BC.Name, 'GERHARDT KENTGEN',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxGK,
       AVG(DECODE(BC.Name, 'GERHARDT KENTGEN',
                  BC.ReturnedDate-BC.CheckOutDate,NULL)) AvgGK
  from BOOKSHELF_CHECKOUT BC, BOOKSHELF B
 where BC.Title = B.Title
 group by B.CategoryName
 order by B.CategoryName;

CATEGORYNAM      MAXFF      AVGFF      MAXDT      AVGDT      MAXGK      AVGGK
----------- --------- --------- --------- ---------- -------- ---------
ADULTFIC
ADULTNF             28       23.5         16         16         31       25.5
ADULTREF                                  8          8
CHILDRENFIC                                                     5          5
CHILDRENPIC                               14         14
```

在输出中，数据已经旋转了——行记录(例如，Name 列中包含 FRED FULLER 的那些行)已经被 DECODE 函数转换了，单独的行(具有不同的名称)现在显示为单独的列。下面的示例通过消除与 BOOKSHELF 表的连接，从而进一步说明了旋转功能。

```
column CategoryName format a11

select MAX(DECODE(BC.Name, 'FRED FULLER',
           BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxFred,
       MAX(DECODE(BC.Name, 'DORAH TALBOT',
           BC.ReturnedDate-BC.CheckOutDate,NULL)) MaxDorah
```

```
  from BOOKSHELF_CHECKOUT BC;

   MAXFRED   MAXDORAH
---------- ----------
        28         16
```

PIVOT 运算符简化了这类报表的生成过程。

```
select * from (
  select Name, ReturnedDate, CheckOutDate
    from BOOKSHELF_CHECKOUT BC
)
pivot
(
   MAX(ReturnedDate-CheckOutDate)
   for Name in ('FRED FULLER','DORAH TALBOT')
)
/

'FRED FULLER'   'DORAH TALBOT'
------------   --------------
          28                16
```

PIVOT 查询从 BOOKSHELF_CHECKOUT 表选择数据，然后旋转数据，在 for 子句中使用限制条件来确定计算哪些行。在此例中，选择 Name 为 FRED FULLER 的行，并计算 ReturnedDate 和 CheckOutDate 之间的最大差，并将此差作为第一列。与此相似，将 Name 为 DORAH TALBOT 的行用作第二列。

上面这两个查询的结果是相同的——两个人借出书的最长时间在报表中显示为单独的列。使用 PIVOT 运算符的代码比较容易阅读，也比较容易管理。例如，如果您想要查看更多人的记录，则只需要简单地修改 PIVOT 的代码——只需要将名字添加到 in 子句中，新的列就会显示在输出中。而在 DECODE 版本的代码中，则需要为每个新的列分别添加代码。

除了 PIVOT 运算符以外，Oracle Database 11g 还提供了 UNPIVOT 运算符。正如您所料，UNPIVOT 执行与 PIVOT 相反的功能，是将列转换为行。需要注意的是，UNPIVOT 并非总是能够取消 PIVOT 操作。例如，如果 PIVOT 操作执行一个聚集函数，您就不能用 UNPIVOT 生成已聚集的详细行。

下面这个例子(此示例来自于 Oracle 公司的 Lucas Jellema)说明了 UNPIVOT 运算符的用法。在此例中，从 DUAL 表中选择 5 个不同的值——每个元音字母对应一个值。每个选择一般显示为单独的列(在此示例中，它们分别被命名为 V1、V2、V3、V4 和 V5)。然后，UNPIVOT 运算符将这些列转换为行，并输出。

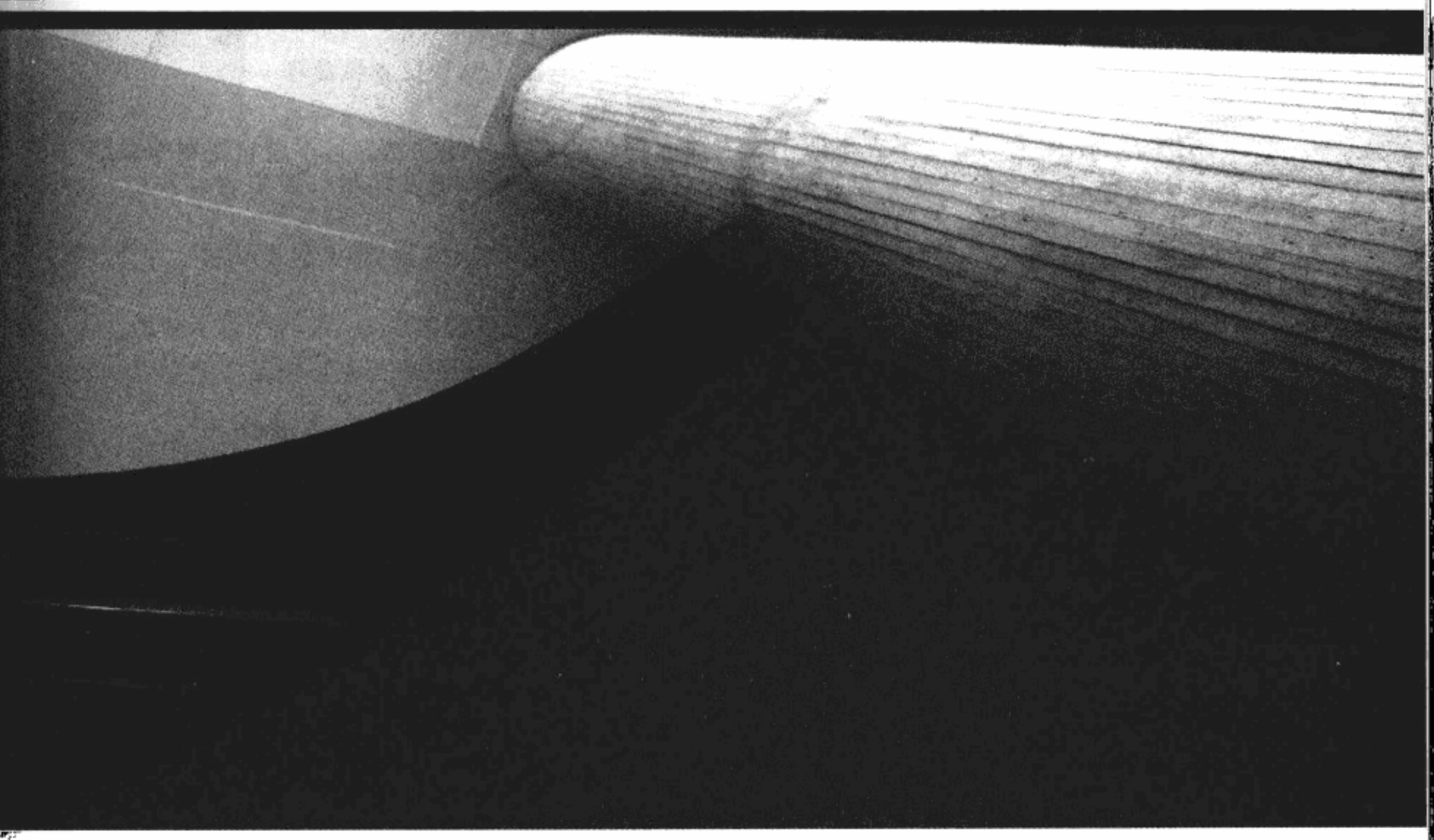
```
select value
from
(
   (
      select
         'a' v1,
```

```
            'e' v2,
            'i' v3,
            'o' v4,
            'u' v5
        from DUAL
    )
    unpivot
    (
        value
        for value_type in
            (v1,v2,v3,v4,v5)
    )
)
/
```

此查询的输出如下所示：

```
V
-
a
e
i
o
u
```

使用 PIVOT 和 UNPIVOT 运算符可以将列转换为行或将行恢复为列，如前面这些示例所示。将 PIVOT 和 UNPIVOT 运算符与 DECODE 和 CASE 操作符联合起来使用，可以为数据的显示和操纵提供强大的工具。



第 17 章

创建和管理表、视图、索引、群集和序列

到此为止，本书的重点是介绍如何使用表。而本章的主要内容是创建、删除、更改表，创建视图，以及使用选项(如索引编排表)。您已经看到了大量的 create table 命令，本章将完善那些示例并说明如何使用最新的选项。本章还将给出一些创建和管理视图、索引、群集和序列的命令。

17.1 创建表

考虑 TROUBLE 表，这个表类似于本书前面提到的 COMFORT 表，只不过它是用来跟踪有异常天气状况的城市。

```
describe TROUBLE

Name                            Null?      Type
------------------------------- --------   ------------
CITY                            NOT NULL   VARCHAR2(13)
SAMPLEDATE                      NOT NULL   DATE
NOON                                       NUMBER(3,1)
MIDNIGHT                                   NUMBER(3,1)
PRECIPITATION                              NUMBER
```

TROUBLE表中的列表示Oracle中3种主要的数据类型，即VARCHAR2、DATE和NUMBER。下面创建该 Oracle 表的 SQL 语句：

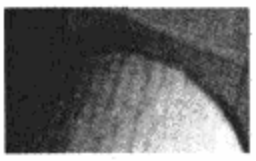
```
create table TROUBLE (
City           VARCHAR2(13) NOT NULL,
SampleDate     DATE NOT NULL,
Noon           NUMBER(3,1),
Midnight       NUMBER(3,1),
Precipitation  NUMBER
);
```

下面是此命令的基本元素：

- 关键词 create table
- 表的名称
- 左圆括号
- 列定义
- 右圆括号
- SQL 终止符

各列的定义由逗号分隔，最后一个列定义后面无逗号。虽然表名和列名必须以字母表中的字母开始，但可以包括字母、数字和下划线。名称长度为 1～30 个字符，并且该名称在表中必须是唯一的，不能用 Oracle 的保留字(请参阅本书“命令和术语参考”中的 RESERVED WORDS 部分)。

如果名称没有用双引号括起来，则在命名表时不用考虑字母的大小写。DATE 数据类型没有选项。变长字符数据类型必须指定最大长度。NUMBER 可以是高精度(最高到 38 位)，也可以是指定的精度，这要根据数字的最大位数和允许小数点后有几位小数来确定(例如，使用美元的 Amount 字段仅可以有两位小数)。



注意：
不要用双引号把表名和列名括起来，否则字母的大小写将会出问题，这会给用户和开发人员造成麻烦。

关于对象关系功能的其他 create table 选项，请参阅本书第 V 部分。

17.1.1　字符宽度和数值精度

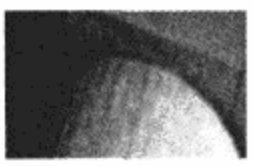
在设计表的时候，必须考虑为字符列(CHAR 和 VARCHAR2)指定最大宽度，为 NUMBER 列指定精度。虽然以后可以用 alter table 命令来修正不正确的定义，但这种修改是很麻烦的。

1. 确定合适的宽度

如果一个字符列的宽度小于要存放的数据的宽度，则会造成 insert 操作的失败并产生如下错误消息：

```
ERROR at line 1:
ORA-12899: value too large for column "PRACTICE"."TROUBLE"."CITY"
(actual: 16, maximum: 13)
```

CHAR(定长字符)列的最大字符宽度是 2 000 个字符。VARCHAR2(变长字符)列可以有最多 4 000 个字符。在设计列的宽度时，应该分配足够的空间以适应将来可能出现的情况。例如，将城市名设置为 CHAR(15)会在以后带来麻烦。因为您可能不得不更改这个表，或者截短某些城市的名称。



注意：
在 Oracle 中定义较宽的 VARCHAR2 列不会产生什么问题。Oracle 很“聪明”，不会在 VARCHAR2 列的末尾存储空格。例如，即使将列定义为 VARCHAR2(50)，城市名 SAN FRANCISCO 也只占 13 个字符的空间。而且，如果一个列中没有任何内容(NULL)，则 Oracle 也不会在列中存储任何内容，甚至连一个空格也不存储(虽然它确实存储了几个字节的内部数据库控制信息，但它们不受为列定义的尺寸影响)。定义过高的值所产生的唯一影响是会对默认的 SQL*Plus 列格式产生影响。SQL*Plus 将会创建一个与 VARCHAR2 定义的宽度相同的默认标题。

2. 选择 NUMBER 的精度

精度不正确的 NUMBER 列将会造成两种后果，即 Oracle 要么拒绝插入一行数据要么把数据的精度降低。下面是要插入 Oracle 表的 4 行数据：

```
insert into TROUBLE values
       ('PLEASANT LAKE','21-MAR-03',
       39.99, -1.31, 3.6);
```

```
insert into TROUBLE values
    ('PLEASANT LAKE','22-JUN-03',
    101.44, 86.2, 1.63);

insert into TROUBLE values
    ('PLEASANT LAKE','23-SEP-03',
    92.85, 79.6, 1.00003);

insert into TROUBLE values
    ('PLEASANT LAKE','22-DEC-03',
    -17.445, -10.4, 2.4);
```

下面是产生的结果：

```
insert into TROUBLE values
    ('PLEASANT LAKE','21-MAR-03',
    39.99, -1.31, 3.6);

1 row created.

insert into TROUBLE values
    ('PLEASANT LAKE','22-JUN-03',
    101.44, 86.2, 1.63);
    *
ERROR at line 3:
ORA-01438: value larger than specified precision allows for this column

insert into TROUBLE values
    ('PLEASANT LAKE','23-SEP-03',
    92.85, 79.6, 1.00003);

1 row created.

insert into TROUBLE values
    ('PLEASANT LAKE','22-DEC-03',
    -17.445, -10.4, 2.4);

1 row created.
```

第 1 行、第 3 行和第 4 行成功地插入表中，而第 2 行则插入失败，因为 101.44 超出了 create table 语句设置的精度，其中 Noon 定义为 NUMBER(3，1)。其中的 3 指出了 Oracle 最多将要存储的数字位数。1 表示这 3 位数字中有一位在小数点的右边。因此，12.3 是合法的，但 123.4 不合法。

请注意，该行的错误是由 101 引起的，而不是由 0.44 引起的，因为 NUMBER(3，1)只允许小数点前面有两位数字。0.44 将不会产生“value larger than specified precision(值超过指定精度)”的错误。它将被简单地四舍五入为小数点后一位数。稍后会说明这一点，下面首先来看一下插入这 4 行的查询的结果：

```
select * from TROUBLE;

CITY            SAMPLEDAT      NOON   MIDNIGHT  PRECIPITATION
--------------  --------- --------   --------  -------------
PLEASANT LAKE   21-MAR-03       40       -1.3            3.6
PLEASANT LAKE   23-SEP-03     92.9       79.6         1.00003
PLEASANT LAKE   22-DEC-03    -17.4      -10.4            2.4
```

成功地插入了 3 行，有问题的那行没有插入。Oracle 会自动取消失败的 insert 语句。

17.1.2　在插入时进行舍入

假如修正 create table 语句并且增加 Noon 和 Midnight 可用的数字位数，则可采用如下所示的方法：

```
drop table TROUBLE;
create table TROUBLE (
City            VARCHAR2(13) NOT NULL,
SampleDate      DATE NOT NULL,
Noon            NUMBER(4,1),
Midnight        NUMBER(4,1),
Precipitation   NUMBER
);
```

现在 4 条 insert 语句都会成功。现在查询显示为：

```
insert into TROUBLE values
    ('PLEASANT LAKE','21-MAR-03',
    39.99, -1.31, 3.6);

insert into TROUBLE values
    ('PLEASANT LAKE','22-JUN-03',
    101.44, 86.2, 1.63);

insert into TROUBLE values
    ('PLEASANT LAKE','23-SEP-03',
    92.85, 79.6, 1.00003);

insert into TROUBLE values
    ('PLEASANT LAKE','22-DEC-03',
    -17.445, -10.4, 2.4);

select * from TROUBLE;

CITY            SAMPLEDAT      NOON   MIDNIGHT   PRECIPITATION
--------------  --------- --------   --------   -------------
PLEASANT LAKE   21-MAR-03       40       -1.3            3.6
PLEASANT LAKE   22-JUN-03    101.4       86.2            1.63
PLEASANT LAKE   23-SEP-03     92.9       79.6         1.00003
PLEASANT LAKE   22-DEC-03    -17.4      -10.4            2.4
```

请看第一个 insert 语句。Noon 的值为 39.99。在此查询中，它被四舍五入为 40。Midnight 的值插入时为 - 1.31。在查询中，它的值被舍入为 - 1.3。Oracle 会根据指定的小数点右边的精度位数进行四舍五入。表 17-1 列出了几个示例的精度效果。关于 ROUND 函数的示例，请参见第 9 章。

表 17-1 插入时的精度效果

Insert 语句中的值	表中的实际值
精度为 NUMBER(4,1)	
123.4	123.4
123.44	123.4
123.45	123.5
123.445	123.4
1234.5	插入失败
精度为 NUMBER(4)	
123.4	123
123.44	123
123.45	123
123.445	123
1234.5	1235
12345	插入失败
精度为 NUMBER(4, - 1)	
123.4	120
123.44	120
123.45	120
123.445	120
125	130
1234.5	1230
12345	插入失败
精度为 NUMBER	
123.4	123.4
123.44	123.44
123.45	123.45
123.445	123.445
125	125
1234.5	1234.5
12345.6789012345678	12345.6789012345678

17.1.3　create table 的约束

create table 语句能够对一个表施加几种不同的约束，其中包括：候选键、主键、外键以及检查条件。constraint(约束)子句可以约束表中的一列或一组列。这些约束的作用就是使 Oracle 完成维护数据库完整性的大多数工作。表定义添加的约束越多，应用程序中所做的维护数据的工作就越少。另一方面，表中的约束越多，更新数据所花的时间就越长。

有两种指定约束的方法，一种方法是作为列定义的一部分定义(称为列约束)，另一种方法是在 create table 语句的末尾定义(称为表约束)。限制多个列的子句必须是表约束。

1. 候选键

候选键(candidate key)是一列或多列的组合，其值唯一地标识了表中的一行。下面的程序清单显示了创建表 TROUBLE 的一个 UNIQUE 约束：

```
drop table TROUBLE;

create table TROUBLE (
City            VARCHAR2(13) NOT NULL,
SampleDate      DATE NOT NULL,
Noon            NUMBER(4,1),
Midnight        NUMBER(4,1),
Precipitation   NUMBER,
constraint TROUBLE_UQ UNIQUE (City, SampleDate)
);
```

这个表中的键是 City 和 SampleDate 的组合。请注意，这两个列都声明为 NOT NULL。这一特性能够阻止无数据的列插入表中。很显然，如果不知道是在哪里或在什么时候收集到的数据，则光有温度和降雨量信息是没有用的。虽然这种技术对于作为表的主键的那些列是很常用的，但是如果某些列是使数据行有意义的关键列，则使用这一技术也很有益处。如果指定出 NOT NULL，则相应的列可具有 NULL 值。

创建 UNIQUE 约束时，Oracle 将创建一个唯一的索引来强制值的唯一性。

2. 主键

表的主键(primary key)是具有某些特殊性质的候选键之一。可以仅有一个主键，并且主键列不能包含 NULL 值。

除了可以有几个 UNIQUE 约束但只能有一个 PRIMARY KEY 约束外，下面的 create table 语句与前面的语句具有相同的作用：

```
drop table TROUBLE;
create table TROUBLE (
City            VARCHAR2(13),
SampleDate      DATE,
Noon            NUMBER(4,1),
Midnight        NUMBER(4,1),
Precipitation   NUMBER,
```

```
constraint TROUBLE_PK PRIMARY KEY (City, SampleDate)
);
```

对于单列的主键或候选键来说，可以在具有列约束而不是表约束的列上定义键：

```
create table AUTHOR
(AuthorName VARCHAR2(50) primary key,
Comments     VARCHAR2(100));
```

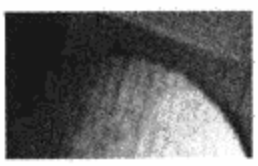
在本例中，AuthorName 列是主键，并且 Oracle 会为这个 PRIMARY KEY 约束生成一个名称。正如 17.1.5 节将介绍的那样，如果希望为键应用某种常见的命名标准，则不推荐您这样做。

17.1.4 指定索引表空间

UNIQUE 约束和 PRIMARY KEY 约束创建索引。除非您告诉 Oracle 将那些索引存储在其他地方，否则那些索引将存储在默认表空间中(表空间管理将在第 22 章全面介绍)。为了指定另一个表空间，可使用 create table 命令的 using index tablespace 子句，如下面的程序清单所示：

```
create table AUTHOR2
(AuthorName VARCHAR2(50),
Comments    VARCHAR2(100),
constraint  AUTHOR_PK primary key (AuthorName)
using index tablespace USERS);
```

与 AUTHOR_PK 主键约束相关的索引将存储在 USERS 表空间中。有关表空间管理的详细内容，请参见第 22 章。



注意：
在大多数默认安装中，创建 USERS 表空间，它是默认表空间。

1. 外键

外键(foreign key)是基于另一个表的主键值的列的组合。外键约束也称为参照完整性约束，它指明外键的值对应于另一个表中的主键的实际值。例如，在 BOOKSHELF 表中，CategoryName 列引用了 CATEGORY 表中的 CategoryName 列的值：

```
create table BOOKSHELF
(Title         VARCHAR2(100) primary key,
Publisher      VARCHAR2(20),
CategoryName   VARCHAR2(20),
Rating         VARCHAR2(2),
constraint CATFK foreign key (CategoryName)
references CATEGORY(CategoryName));
```

即使是在同一个表中，也可以引用主键或唯一键。但是，不能在 references 子句中引用

一个远程数据库中的表。可以用表的形式(以前用来在 TROUBLE 表上创建一个 PRIMARY KEY)而不是用列的形式来指定包含几个列的外键。

有时在删除相关行所依赖的行时，可能希望删除这些相关行。在 BOOKSHELF 表和 CATEGORY 表的示例下，如果想删除 CATEGORY 表的某个 CategoryName 列，则可以使匹配的 BOOKSHELF 表的 CategoryName 列值为 NULL。在另外一种情况下，您可能想删除这整个行。添加到 references 子句中的 on delete cascade 子句将告诉 Oracle 在您删除父表的相应行时，也删除子表相关的行。这种操作会自动维护参照完整性。关于 on delete cascade 子句和 references 子句的更多信息，请参阅附录 A 中的“INTEGRITY CONSTRAINT”部分。

2. CHECK 约束

许多列必须具有处于某个范围或满足某些条件的值。利用 CHECK 约束，您可以指定一个表达式，这个表达式对于表中的每一行必须始终为真。例如，RATING 表存储有效的等级；为了限定可用的值不超过列定义的限制值，可以使用 CHECK 约束，如下面的程序清单所示：

```
create table RATING_WITH_CHECK
(Rating VARCHAR2(2) CHECK (Rating <= 9),
RatingDescription VARCHAR2(50));
```

一个列级的 CHECK 约束不能引用其他行的值，不能使用伪列如 SysDate、User、CurrVal、NextVal、Level 和 RowNum 等。您可以使用表约束形式(相对于列约束形式)来引用 CHECK 约束中的多个列。

17.1.5　命名约束

可以给约束命名。如果为约束名使用一种有效的命名模式，就能够更好地识别和管理约束。约束名应该标识出它所作用的表和它所表示的约束类型。例如，TROUBLE 表的主键可以命名为 TROUBLE_PK。

在创建一个约束时，可以给这个约束指定一个名称。如果没有为约束指定名称，则 Oracle 将会生成一个名称。Oracle 所生成的大多数约束名都是 SYS_C######的形式。例如，SYS_C000145。因为系统生成的约束名并不提供关于表和约束的任何信息，所以您最好命名自己的约束。

在下面的示例中，作为 TROUBLE 表的 create table 命令的一部分，创建和命名了 PRIMARY KEY 约束。请注意下面的 constraint 子句：

```
create table TROUBLE (
City           VARCHAR2(13),
SampleDate     DATE,
Noon           NUMBER(4,1),
Midnight       NUMBER(4,1),
Precipitation NUMBER,
constraint TROUBLE_PK PRIMARY KEY (City, SampleDate)
);
```

create table 命令的 constraint 子句为约束命名(在此例中命名为 TROUBLE_PK)。以后，

在启用或禁用约束时，可以使用这个约束名。

17.2 删除表

删除表的操作很简单。使用关键词 drop table 和表名就可以了。如下所示：

```
drop table TROUBLE;
```

只有当不再需要表的时候才能删除它。在 Oracle 中，truncate 命令可以删除表中的所有行，并回收空间以便其他的用途，而不从数据库中删除表的定义。

从 Oracle Database 10g 开始，删除表时仍保留了表的空间，它们只是暂时通过“回收站”来访问。如果要在删除表的同时把它从“回收站”中清除，可以使用 drop table 命令中的 purge 子句。

```
drop table TROUBLE purge;
```

如果表已经被删除了，则可以从“回收站”中清除所占的空间。

```
purge table TROUBLE;
```

使用以下命令可以清除“回收站”中的所有内容：

```
purge recyclebin;
```

截断表的操作也很简单：

```
truncate table TROUBLE;

Table truncated.
```

截断操作不能回滚。如果有删除行的触发器，且所删除的行依赖于这个表中的行，则 truncate 命令将不触发这些触发器。所以，在执行 truncate 命令前，必须保证确实是想执行 truncate。您可以选择释放表的存储空间(通过默认的 drop storage 子句)或保留它。

17.3 更改表

表定义的更改有多种方法：给已有的表添加列，更改列的定义，或者删除列。数据库管理员们通常以其他方式更改表来管理空间和内存的使用。添加一列的操作很简单，类似于创建一个表。假如您打算为 TROUBLE 表新添加两列：Condition(应该是 NOT NULL)和 Wind(表示风速)。首先创建表，然后用一些记录填充该表：

```
drop table TROUBLE;

create table    TROUBLE (
City            VARCHAR2(13),
SampleDate      DATE,
```

```
Noon            NUMBER(4,1),
Midnight        NUMBER(4,1),
Precipitation   NUMBER);

insert into     TROUBLE values
    ('PLEASANT LAKE','21-MAR-03',
    39.99, -1.31, 3.6);

insert into TROUBLE values
    ('PLEASANT LAKE','22-JUN-03',
    101.44, 86.2, 1.63);

insert into TROUBLE values
    ('PLEASANT LAKE','23-SEP-03',
    92.85, 79.6, 1.00003);

insert into TROUBLE values
    ('PLEASANT LAKE','22-DEC-03',
    -17.445, -10.4, 2.4);
```

添加上述两个新列的第一个尝试如下所示：

```
alter table TROUBLE add (
Condition   VARCHAR2(9) NOT NULL,
Wind        NUMBER(3)
);

alter table TROUBLE add (
            *
ERROR at line 1: ORA-01758: table must be empty to add
mandatory (NOT NULL) column
```

这里出现了错误消息，因为不能添加定义为 NOT NULL 的列，在试图添加这个列时，其中不具有任何内容。表中的每行将有一个定义为 NOT NULL 的新的空列，这是互相矛盾的。

在 Oracle 11g 之前，如果表是空的，则 alter table 命令中的 add 子句将使用一个 NOT NULL 列，但在通常情况下，把表中所有的行清空仅仅为了添加一个 NOT NULL 列是不现实的。

另一种办法是首先通过添加没有 NOT NULL 限制的列来更改表：

```
alter table TROUBLE add (
Condition    VARCHAR2(9),
Wind         NUMBER(3)
);

Table altered.
```

然后，为每一行用数据填充这个列(用合法的数据，或用占位符，直到获得合法的数据)：

```
update TROUBLE set Condition = 'SUNNY';
```

最后，再次更改这个表，修改列定义为 NOT NULL：

```
alter table TROUBLE modify (
Condition     VARCHAR2(9) NOT NULL,
City          VARCHAR2(17)
);

Table altered.
```

您可以向已经包含行的表添加 NOT NULL 列。为此，必须为新的列指定一个默认值。但是，如果是大型数据表，这就会产生新的问题，因为数百万行可能需要更新，以便包含新的列值。Oracle 11g 自动处理这种情况，从而避免对用户产生大的影响。

再来看一下只向 TROUBLE 表添加 Condition 列：

```
alter table TROUBLE add (Condition VARCHAR2(9) NOT NULL);
```

如果 TROUBLE 表已经有行，则不能添加 Condition 列，因为现有行的 Condition 列有 NULL 值。解决方案是为 Condition 列添加一个 DEFAULT 约束：

```
alter table TROUBLE add (Condition VARCHAR2(9)
default 'SUNNY' NOT NULL);
```

虽然 Oracle 现在将创建新列，但不会更新现有的行。如果用户选择 Condition 列有空值的一行，则会返回默认值。Oracle 的处理方法解决了两个问题：您可以添加 NOT NULL 列；避免了必须更新现有的每一行。在大型数据表中，这就意味着大大节省了支持事务规模所需要的空间。

请注意，City 列也被修改了，其字符数增加为 17 个(仅仅是为了说明怎样进行这项工作)。在说明该表时，显示如下内容：

```
describe TROUBLE

Name                          Null?     Type
----------------------------- --------  ------------
CITY                                    VARCHAR2(17)
SAMPLEDATE                              DATE
NOON                                    NUMBER(4,1)
MIDNIGHT                                NUMBER(4,1)
PRECIPITATION                           NUMBER
CONDITION                     NOT NULL  VARCHAR2(9)
WIND                                    NUMBER(3)
```

此表包含下面的信息：

```
select * from TROUBLE;

CITY            SAMPLEDAT   NOON   MIDNIGHT PRECIPITATION  CONDITION WIND
--------------- --------- ------   -------- -------------  --------- ----
PLEASANT LAKE   21-MAR-03     40       -1.3            3.6  SUNNY
PLEASANT LAKE   22-JUN-03  101.4       86.2           1.63  SUNNY
PLEASANT LAKE   23-SEP-03   92.9       79.6        1.00003  SUNNY
PLEASANT LAKE   22-DEC-03  -17.4      -10.4            2.4  SUNNY
```

从该表中可以看到更改的效果。City 现在是 17 个字符宽而不是 13 个字符宽。Condition 作为 NOT NULL 添加到表中，并且值为 SUNNY(暂时的)。Wind 也添加到表中，且值为 NULL。

为了使 NOT NULL 列可空，可使用带 NULL 子句的 alter table 命令，如下所示：

```
alter table TROUBLE modify
(Condition NULL);
```

17.3.1　添加或修改列的规则

下面是修改列的规则：

- 可以随时增加字符列的宽度。
- 可以随时增加 NUMBER 列的数字位数。
- 可以随时增加或减少 NUMBER 列中的小数位数。

此外，如果表中的每一行的某一列都为 NULL，则可以作下列更改：

- 可以更改该列的数据类型。
- 可以减少字符列的宽度。
- 可以减少 NUMBER 列的数字位数。

如果该列在表的所有行中都为空(NULL)，则您只能更改一列的数据类型。

数据类型更改的限制有一个例外情况。Oracle 支持将 LONG 数据类型列更改为 LOB 数据类型，即使 LONG 列中已经存在数据。下面的程序清单说明了此功能：

```
create table LONGTEST
(Col1 NUMBER,
Longcol LONG);

Table created.

insert into LONGTEST values (1,'This is a LONG value');

1 row created.

alter table LONGTEST modify (Longcol CLOB);

Table altered.

desc LONGTEST

Name                                      Null?    Type
----------------------------------------- -------- --------
COL1                                               NUMBER
LONGCOL                                            CLOB

select Longcol from LONGTEST;
LONGCOL
-----------------------------------
This is a LONG value
```

17.3.2 创建只读表

在 Oracle 11g 中，您可以将表更改为只读或读、写状态。这样您就可以阻止表一级的 insert、update 和 delete 操作(在 Oracle 11g 以前的版本中，这一功能存在于表空间一级，否则就需要使用视图)。

要将表设置为只读状态，可以使用 alter table 命令：

```
alter table LONGTEST read only;
```

试图更改 LONGTEST 中的记录会导致错误。要将表恢复到以前的状态，需要在 alter table 命令中使用 read write 子句，如下所示：

```
alter table LONGTEST read write;
```

要查看表的状态，查询 USER_TABLES 数据字典视图的 Read_Only 列。

当表处于只读模式时，可以继续对它执行 DML 操作。

17.3.3 更改当前使用的表

当您发出 alter table 命令时，Oracle 试图获取表上的 DDL 锁。如果此时其他人正在访问此表，则命令会失败——当您更改表的结构时需要对表具有排他的访问权。您可能需要反复执行命令才能获得需要的锁。

从 Oracle 11g 开始，可以使用 DDL 锁超时选项来处理这一问题。可以执行 alter session 命令来设置 ddl_lock_timeout 参数的值，指定 Oracle 不断地重试命令所应该持续的秒数。当命令成功执行或者到达设定的超时值时，不再继续重试命令。

要在 60 秒内反复尝试执行命令，则使用如下命令：

```
alter session set ddl_lock_timeout=60;
```

DBA 可以通过 alter system 命令在数据库级别启用它，如下所示：

```
alter system set ddl_lock_timeout=60;
```

17.3.4 创建虚拟列

从 Oracle 11g 开始，您可以在表中创建虚拟列，而不是存储派生的数据。虚拟列可以基于同一行中的其他值(例如，将两列加起来)。在以前的版本中，您需要创建视图来执行创建列值必需的函数。在 Oracle 11g 中，可以将函数指定为表定义的一部分，这样就可以对虚拟列创建索引，并通过虚拟列创建分区表。

考虑下面的 AUTHOR 表：

```
create table AUTHOR
    (AuthorName  VARCHAR2(50) primary key,
    Comments  VARCHAR2(100));
```

当把值插入到 AUTHOR 表中时，它们是大写字母形式：

```
insert into AUTHOR values
('DIETRICH BONHOEFFER', 'GERMAN THEOLOGIAN, KILLED IN A WAR CAMP');

insert into AUTHOR values
('ROBERT BRETALL','KIERKEGAARD ANTHOLOGIST');
```

您可以创建另外一个大小写字母混合的列，使用 generated always 子句告诉 Oracle 创建虚拟列：

```
create table AUTHOR2
(AuthorName VARCHAR2(50) primary key,
Comments VARCHAR2(100),
MixedName VARCHAR2(50)
generated always as
(initcap(AuthorName) ) virtual
);
```

创建虚拟列的代码是复杂的，包括 case 语句和其他函数。需要注意的是，您不必创建触发器或其他机制来填充列。Oracle 针对表中的每一行将此列作为一个虚拟值来维护。值并没有物理地存储于表中，而是只在需要的时候才生成。

注意：
试图在插入操作期间为虚拟列提供值将会产生错误。

与在表中的标准列上创建索引一样，您可以在虚拟列上创建索引。另外，可以根据此列创建分区表(具体介绍参见第 18 章)。

17.3.5　删除列

可以从表中删除一列。删除列比添加列或修改列更为复杂，这是因为 Oracle 必须执行一些额外的工作。仅从表的多个列中删除一列(从而在该表中执行 select *时不显示它)是很简单的。真正复杂的是恢复这些列值占用的空间，对数据库而言这可能很耗时。因此，可以立即删除一列，或给该列标记为“unused”，稍后再删除它。如果立即删除一列，则该操作可能会影响性能。如果将该列标记为未使用，则对性能不会产生影响。该列可以在今后不频繁使用数据库时真正地删除。

为了删除一列，使用 alter table 命令的 set unused 子句或 drop 子句。但是，不能删除伪列、嵌套表的列或分区键列。

下面的示例从 TROUBLE 表中删除 Wind 列：

```
alter table TROUBLE drop column Wind;
```

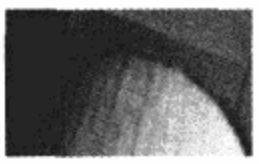
也可以将 Wind 列标记为 unused：

```
alter table TROUBLE set unused column Wind;
```

若标记一个列为 unused，则在删除该列之前不会释放此列以前所占用的空间：

```
alter table TROUBLE drop unused columns;
```

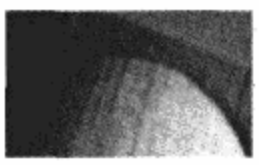
可以查询 USER_UNUSED_COL_TABS、ALL_UNUSED_COL_TABS 以及 DBA_UNUSED_COL_TABS 来查看所有其中有标记为 unused 的列的表。



注意：
一旦将某列标记为 unused，则不能访问该列。

可以利用一条命令删除多列，如下面的程序清单所示：

```
alter table TROUBLE drop (Condition, Wind);
```



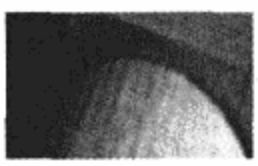
注意：
在删除多列时，不应该使用 alter table 命令的 column 关键字；否则将导致一个语法错误。如上面的程序清单所示，多个列名必须括在圆括号内。

如果删除的列是主键或唯一约束的一部分，那么在 alter table 命令中必须使用 cascade constraints 子句。如果删除属于主键的列，则 Oracle 将同时删除该列和相应的主键索引。

17.4 根据一个表创建另一个表

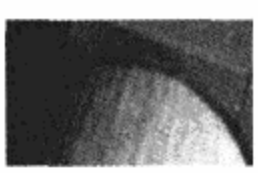
运行时，Oracle 可以根据在一个已有的表上的 select 语句快速地创建一个新表：

```
create table RAIN_TABLE as
  select City, Precipitation
    from TROUBLE;

  Table created.
```



注意：
如果所选择的其中一列为 LONG 数据类型，则 create table…as select…命令不起作用。

当描述这个新表时，它显示从 TROUBLE 表中“继承”了列定义。以这种方式创建的表能包括另一个表的所有列(如果喜欢可以使用星号)或者列的子集。新表也可以包括“创造”的列，这些列就像视图中的列一样由函数或其他列的组合产生。字符列的大小将进行调整，以容纳创造出来的列中的数据。NUMBER 列在源表中具有指定精度，但在创建新列的过程中经过计算的 NUMBER 列将成为新表中没有指定精度的 NUMBER 列。



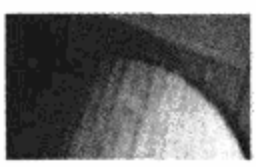
注意：
不会为新表自动创建源表上的约束，包括 NOT NULL 约束。

在描述 RAIN 表时，它的列定义显示如下：

```
describe RAIN_TABLE

Name                                      Null?    Type
----------------------------------------  -------- ------------
CITY                                               VARCHAR2(17)
PRECIPITATION                                      NUMBER
```

在查询 RAIN_TABLE 表时，它仅包含了从 TROUBLE 表中选定的列和数据，如下所示(因为创建 RAIN_TABLE 表时它已经存在)：

```
select * from RAIN_TABLE;

CITY            PRECIPITATION
--------------- -------------
PLEASANT LAKE             3.6
PLEASANT LAKE            1.63
PLEASANT LAKE         1.00003
PLEASANT LAKE             2.4
```



注意：
如果查询语句中的一列是通过公式生成的，则必须在创建表时为 Oracle 指定该列的一个列别名。

通过构建一个不从旧表中选择行的 where 语句，也可以利用这种方法创建列定义与源表中的列定义相同的表，只不过所创建的表中没有任何行：

```
create table RAIN_EMPTY as
select City, Precipitation
  from TROUBLE
 where 1=2;

Table created.
```

查询这个表时将不显示任何内容：

```
select * from RAIN_EMPTY;

no rows selected.
```

可以基于查询创建一个表而不生成重做日志(redo log)项(在数据库恢复期间使用的按时间排列的数据库操作的记录)。避免生成重做日志项是通过在 create table 命令中使用 nologging 关键字来实现的。在以这种方法避免生成重做日志项时，create table 命令的性能将得到改善(因为所做的工作减少了)。表越大，对性能的影响就越大。但是，因为新表的创建未写入重做日志文件(重做日志文件记录重做日志项)，所以如果以后数据库出错，用重做日志文件恢复数据库时，就不会重新创建这个表。因此，如果希望能恢复这个新表，那么在使用 nologging 选项后不久，应当考虑执行数据库的备份。

下面的示例说明了在查询的基础上创建表时如何使用 nologging 关键字(默认情况下，基于查询创建表时将生成重做日志项)：

```
create table RAIN_NOLOG
nologging
as
select * from TROUBLE;

Table created.
```

请注意，可以在分区级和 LOB 级指定 nologging(参见第 18 章和第 40 章)。nologging 的使用将改善向表中加载数据操作的性能。

17.5 创建索引编排表

索引编排表(index-organized table)能根据表的主键列值对数据进行排序。索引编排表存储数据时就像整个表存储在一个索引中那样。索引有两个主要用途：

- **强调唯一性** 在创建一个 PRIMARY KEY 或 UNIQUE 约束时，Oracle 创建索引来强调索引列的唯一性。
- **改善性能** 当某个查询使用索引时，此查询的性能将得到显著的改善。关于查询在什么条件下可以使用索引的更多信息，请参阅第 46 章。

索引编排表允许在索引中存储整个表的数据。普通的索引只存储索引列，而索引编排表则在索引中存储表中的所有列。

为了创建一个索引编排表，可使用 create table 命令的 organization index 子句，如下例所示：

```
create table TROUBLE_IOT (
City           VARCHAR2(13),
SampleDate     DATE,
Noon           NUMBER(4,1),
Midnight       NUMBER(4,1),
Precipitation NUMBER,
constraint TROUBLE_IOT_PK PRIMARY KEY (City, SampleDate))
organization index;
```

为了把 TROUBLE 表创建成索引编排表，必须在其上创建一个 PRIMARY KEY 约束。

如果您经常通过 City 列和 SampleDate 列访问 TROUBLE 表的数据(在查询的 where 子句中)，那么创建一个索引编排表是很适合的。为了使索引所需的有效管理量最小，应该仅在表中的数据上静态的情况下使用索引编排表。如果表中的数据频繁更改，则应该使用普通的表以及相应的索引。

通常情况下，当主键由表中大部分列构成时，使用索引编排表是最有效的。如果表中包含了许多需要频繁访问但又不属于主键的列，则索引编排表需要重复访问它的溢出区。虽然这是个缺点，但还是可以选择使用索引编排表以便利用那些标准表中没有的重要功能，如使用 alter table 命令的 move online 选项的功能等。在表被 insert、update 和 delete 操作访问时，

可以使用这个选项把该表从一个表空间移动到另一个表空间。分区的索引编排表不能使用 move online 选项。分区的概念将在第 18 章介绍。

17.6　创建视图

前面章节已经介绍了创建视图的方法，这里就不重复了。但本节将介绍另外几个证明视图有用的要点。

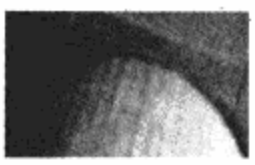
如果一个视图基于单个底层的表，那么可以在此视图中对行进行 insert、update 或 delete 操作。这实际将在底层的表中 insert、update 或 delete 行。这样做存在一些限制，不过这些限制相当合理：

- 如果底层的表具有未出现在视图中的任意 NOT NULL 列，则不能进行 insert 操作。
- 如果在 insert 或 update 中引用的任意一个视图的列包含了函数或计算，则不能进行 insert 或 update 操作。
- 如果视图包括 group by、distinct 或对伪列 RowNum 的引用，则不能进行 insert、update 或 delete 操作。

如果 Oracle 能确定要插入的适当的行，那么可以在基于多个表的视图上进行 insert 操作。在一个多表视图中，Oracle 可以确定哪些表是被键保留的。如果一个视图包含了标识该表的主键的足够多的列，则这个键被保留，并且 Oracle 可以通过此视图向表中插入行。

17.6.1　视图的稳定性

请回忆一下，在执行一个视图的查询时，会立即从一个表(或多个表)产生查询结果。但此时，视图还是不能像表那样拥有自己的数据。它仅仅是对从其他表中得到什么信息以及如何组织这些信息的描述(SQL 语句)。结果，如果表被删除，则视图的有效性就会遭到破坏。试图查询底层表已删除的视图将会产生一条关于视图的错误消息。



注意：
这个规则的唯一例外是物化视图的使用。物化视图实际上是一个表，它存储通常通过视图进行查询的数据。物化视图将在第 26 章详细介绍。

下面，首先在一个已有表上创建一个视图，然后删除该表，最后查询这个视图。

首先，创建视图：

```
create view RAIN_VIEW as
select City, Precipitation
  from TROUBLE;

View created.
```

删除底层的表：

```
drop table TROUBLE;

Table dropped.
```

查询这个视图：

```
select * from RAIN_VIEW;
                *
ERROR at line 1:
ORA-04063: view "PRACTICE.RAIN_VIEW" has errors
```

类似地，可以利用星号创建视图：

```
create or replace view RAIN_VIEW as
select * from TROUBLE;

View created.
```

然后更改底层的表：

```
alter table TROUBLE add (
Warning      VARCHAR2(20)
);

Table altered.
```

尽管更改了视图的基表，该视图仍然有效，不过警告列却不可见了。在更改表后，要使警告列可见就必须替换视图。为了重新创建一个视图，同时保留授予它的所有权限，可使用 create or replace view 命令，如下面的程序清单所示。这个命令将用新的视图文本替换现有视图的视图文本，同时不影响原来授予该视图的权限。

```
create or replace view RAIN_VIEW as
    select * from TROUBLE;
```

17.6.2 视图中的 order by

用户可以在 create view 语句中使用 order by，如下面的基于重新创建的 TROUBLE 表的程序清单所示(请注意，由于要执行附加的分类活动，因此该针对视图的查询将导致性能的损失)：

```
create view TROUBLE_SORTED
     as select * from TROUBLE
   order by City, SampleDate;

View created.

select * from TROUBLE_SORTED;

CITY            SAMPLEDAT       NOON   MIDNIGHT PRECIPITATION
--------------  --------- --------- ---------- -------------
PLEASANT LAKE   21-MAR-03         40       -1.3           3.6
PLEASANT LAKE   22-JUN-03      101.4       86.2          1.63
PLEASANT LAKE   23-SEP-03       92.9       79.6       1.00003
PLEASANT LAKE   22-DEC-03      -17.4      -10.4           2.4
```

17.6.3　创建只读视图

可以使用 create view 命令的 with read only 子句来防止用户通过视图操纵记录。考虑前面创建的 RAIN_VIEW 视图：

```
create or replace view RAIN_VIEW as
  select * from TROUBLE;
```

如果用户能够从 TROUBLE 表中删除记录，那么用户可以通过 RAIN_VIEW 视图删除 TROUBLE 表中的记录：

```
delete from RAIN_VIEW;
```

用户还可以通过对 RAIN_VIEW 视图执行 insert 或 update 操作，从而插入或更新 TROUBLE 表中的记录。如果视图基于多个表的连接，那么用户更新视图的记录的能力将受到限制。除非更新只涉及一个表且视图的列中包括了已更新的表的整个主键，否则不能更新视图的基表。

为了防止通过视图修改基表，可以使用 create view 命令的 with read only 子句。如果在创建视图时使用了 with read only 子句，则用户只能从视图中 select 记录。即使这个视图基于单个表，用户也不能操纵从视图获得的记录：

```
create or replace view RAIN_READ_ONLY as
  select * from TROUBLE
    with read only;
```

也可以使用 INSTEAD OF 触发器来管理针对视图执行的数据操纵命令。关于 INSTEAD OF 触发器的详细内容，请参阅第 38 章。

17.7　索引

索引是一个简单的概念，它是有关于某个主题以及其信息位置的关键字的列表。例如，要想查找关于索引的信息，可以在书籍后面所附的索引中查找单词“indexes”。它将给出您现在正在阅读的页的页码。“indexes”是键，页码指出该书中关于索引的讨论位置。

当您在查找有关索引的信息时，虽然可以通过阅读整本书查找到有关索引的信息页，但会很慢并且很费时。由于书籍后面的索引是按字母顺序排列的，因此可以快速地找到索引中“index”的适当位置(而不是阅读每个条目)。即使这个单词不常用，这样也比从头到尾阅读整本书来查找要快得多。相同的原理也可以应用于 Oracle 索引。例如，如果要查找某本特定的书，则可用 Title 列上的一个限定条件来进行查询：

```
select * from BOOKSHELF
  where Publisher = 'SCHOLASTIC';
```

如果 BOOKSHELF 表在 Publisher 列上没有索引，那么 Oracle 必须阅读该表的每一行直到找到与查询的 where 子句相匹配的所有出版社为止。如果表较小，这还不会导致性能问题。随着表大小的增加，返回所有匹配行所需的时间可能会影响应用程序及其所支持的业务流程的性能。

为了加快数据检索，可在 Publisher 列上创建一个索引。这样，当执行同样的查询时，Oracle 首先在索引中查找，而索引是按顺序存储的，因而很快就会找到名为 SCHOLASTIC 的出版社(Oracle 不用读取每一项，而是直接跳到这个名称的附近，这很像是通过书的索引进行浏览)。然后相应的索引项向 Oracle 提供了该出版社在表(磁盘)中的行的确切位置。Oracle 中的标准索引类型称为 B 树索引，是将列值与其相关的 RowID 相匹配。

对一个重要的列(可能是出现在 where 子句中的列)创建索引一般会加快 Oracle 对查询的响应速度。如果对两个连接表中的相关列创建索引(通过 where 子句)，则索引同样能加速这两个连接表之间的查询。这些是索引的基础内容，本节其余的部分将说明与提高索引的工作效率有关的其他功能和问题。关于索引对查询优化的影响，请参阅第 46 章。

17.7.1 创建索引

可以通过 create index 命令创建一个索引。在表的创建或维护过程中，如果指定了一个主键或唯一列，则 Oracle 将自动创建一个唯一的索引以支持该约束。create index 命令的完整语法请参照附录 A。它最常用的格式如下所示：

```
create [bitmap | unique] index index
    on table( column [, column]. . .) [reverse];
```

index 的名称必须是唯一的，并且遵循 Oracle 列的命名约定。table 是创建索引的基础表的名称，column 是索引所在列的列名。

位图索引可以在只有很少几个不同值的列上创建有用的索引，详细内容请参阅 17.7.4 节。reverse 关键字告诉 Oracle 反转索引值的字节，这在插入许多有序的数据值时可以改善数据库 I/O 的分布。

可在多个列上创建单个索引，这些列必须一个接一个地列出，中间用逗号分开。在下面的示例中，将创建一个没有主键的 BOOKSHELF_AUTHOR 表：

```
create table BOOKSHELF_AUTHOR
(Title VARCHAR2(100),
AuthorName VARCHAR2(50),
constraint TitleFK Foreign key (Title) references BOOKSHELF(Title),
constraint AuthorNameFK Foreign key (AuthorName)
references AUTHOR(AuthorName));
```

17.7.2 实施唯一性

请回忆一下第 I 部分的内容，如果每个表中行的所有列都仅依赖于主键，那么认为这组表是符合第三范式的。在 BOOKSHELF_AUTHOR 表中，主键是 Title 列和 AuthorName 列的组合。在其他表中，主键可以是雇员 ID、客户 ID、账号或银行中的支行号和账号的组合。

在这些情形下，主键的唯一性是至关重要的。具有重复账号的银行或具有重复客户 ID 的账单系统都会造成混乱，因为事务会被计入属于其他用户但主键相同的账户中(这就是为什么通常不用名称作为主键的原因，因为有太多的重复名称)。为了避免这种危险，要让数据库阻止创建重复的主键。Oracle 提供了两种有用的措施：

- 可以通过索引或约束来保证键的唯一性。
- 可以使用序列生成器(将在本章后面介绍)。

17.7.3　创建唯一索引

可用 3 种方法对 BOOKSHELF_AUTHOR 表中的 Title 和 AuthorName 的组合创建一个唯一索引。这 3 种方法分别是：创建主键约束、创建唯一约束、创建唯一索引。如果创建一个约束，就能创建一个引用该约束的外键。如果首先创建一个唯一索引，则仍然能够在该表上创建一个主键，Oracle 将使用现有的索引作为主键索引。

下面的程序清单显示了此多列索引的 create index 命令：

```
create unique index BA$TITLE_AUTHOR
   on BOOKSHELF_AUTHOR(Title, AuthorName);
```

创建主键的方法如下所示：

```
alter table BOOKSHELF_AUTHOR
  add constraint BA_PK primary key (Title, AuthorName);
```

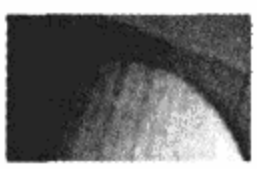
为了创建唯一约束，只需用 unique 替换 primary key 子句即可。当试图在已经有数据的表上创建唯一索引，如果数据有重复，则该命令将失败。如果此 create unique index 语句成功，则以后试图插入(或更新)会生成重复键的行的行为都会失败，并且引发如下错误消息：

```
ERROR at line 1:
ORA-00001: unique constraint (BOOKSHELF_AUTHOR.BA_PK) violated
```

在创建索引时，需要一定的存储空间。关于所创建索引位置的详细情况，请参阅 17.7.9 节。

17.7.4　创建位图索引

使用位图索引可以调整在限制条件中使用非选择性列的查询。位图索引应该只在数据不频繁更新的情况下使用，因为位图索引会增加在它们索引的表上的所有数据操纵事务的系统开销。



注意：
那些与联机事务处理应用程序(这些应用程序取决于 Oracle 用来维护它们的内在机制)相关的表最好不要使用位图索引。另外，和批处理事务相关的表也限制使用位图索引。

在用非选择性列作为查询的限制条件时，使用位图索引是适合的。例如，如果在一个大型的 BOOKSHELF 表中只有很少几个不同的 Rating 值，那么通常不要在 Rating 上创建传统的 B 树索引，即使它经常在 where 子句中使用也是如此。但是，Rating 可以利用位图索引。

位图索引在内部将不同的列值映射到每条记录上。例如，假设在一个大型的 BOOKSHELF 表中只有 5 个 Rating 值(1、2、3、4 和 5)。由于只有 5 个 Rating 值，因此对于 Rating 位图索引有 5 个单独的位图索引项可以使用。如果此表的前 5 行有一个 Rating 值为 1，后 5 行有一个 Rating 值为 2，那么 Rating 位图的项将类似于下面的程序清单所示：

```
Rating bitmaps:
      1: 1 1 1 1 1 0 0 0 0 0
      2: 0 0 0 0 0 1 1 1 1 1
      3: 0 0 0 0 0 0 0 0 0 0
      4: 0 0 0 0 0 0 0 0 0 0
      5: 0 0 0 0 0 0 0 0 0 0
```

在上面的程序清单中，每个 0 和 1 的列表示 BOOKSHELF 表的一行。因为考虑了 10 行，所以显示 10 个位图值。阅读 Rating 的位图，前 5 个记录具有为 1 的 Rating 值(1 值)，后 5 行没有值(0 值)。每个可能的值都有一个单独的位图项。

在查询过程中，Oracle 优化程序能够动态地将位图索引项转换为 RowID。这种转换功能允许优化程序在有许多不同值的列上(通过 B*树索引)和在不同值很少的列上(通过位图索引)使用索引。

为了创建位图索引，可用如下面的程序清单所示的 create index 命令的 bitmap 子句。应该在索引名称中指出它的属性为位图索引，以便在调整操作中很容易检测到它：

```
create bitmap index BOOKSHELF$BITMAP_RATING
    on BOOKSHELF(Rating);
```

如果选择使用位图索引，则应该将查询中的性能益处与数据操纵命令中的性能损失进行权衡。表中的位图索引越多，每个事务处理过程中的系统开销就越大。对于需要频繁添加新数据的列不要使用位图索引，因为在 Rating 列中每添加一个新值就需要创建一个对应的新位图。

17.7.5 何时创建索引

索引在大型表、在那些可能在 where 子句中以两边相等的形式出现的列上时是非常有效的，如下所示：

```
where AuthorName = 'STEPHEN JAY GOULD'
  and Rating = '5'
```

它在连接中也非常有用，如下所示：

```
where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title
```

除了支持 where 子句和连接外，索引还支持 order by 子句和 MAX、MIN 函数。在一些情况下，Oracle 会选择扫描索引而不是扫描整个表。关于优化程序如何使用索引的详细内容，

请参阅第 46 章。

17.7.6　创建不可见索引

创建新索引可能会改变现有的应用程序的工作方式。在大多数情况下，这种行为是有意识的：创建索引是为了改变查询选择的执行路径。新索引应该支持已知的查询执行路径，这将会降低 insert 和其他 DML 操作的性能。从 Oracle 11g 开始，可以分阶段地将索引引入到环境中，先创建索引，然后使它对优化程序可见。

创建索引之后，可以通过 alter index 命令告诉优化程序生成执行计划时不要考虑它：

```
alter index BA$TITLE_AUTHOR invisible;
```

使索引处于“不可见”状态，现在可以测试命令以确定索引将会如何影响操作。只有在使用明确地命名了索引的 INDEX 提示(参见第 46 章)时才使用不可见索引。当索引处于不可见状态时，优化程序不会出于一般的用法而考虑索引。从 USER_INDEXES 数据字典视图中选择 Visibility 列，确定索引是否已被设置为对优化程序不可见。无论索引是否处于不可见状态，都会继续占用空间并影响 DML 操作。

17.7.7　索引列的变化

传统的索引(B*树索引)对于包含有大量可变数据的列是非常有用的。例如，一个用 Y 或 N 来指出一个公司是否是当前客户的列不宜使用传统索引，而且实际上可能会降低查询的速度；位图索引将是更好的选择。电话号码列很适合使用 B*树索引，区号列则不是很适合，它取决于表中区号值的分布是否唯一。

在一个多列索引中，应该首先考虑最经常访问的列。即使首要的列未在查询中提到，也可以利用优化程序的跳动扫描特性来使用多列索引。关于索引用法的示例，请参阅第 46 章。

小型表最好不要创建索引，除非要在主键中实施唯一性。小型表是数据库块很少的表。Oracle 用一次物理读操作就能遍历所有的数据。除此之外，使用索引几乎都能提高效率。一次物理上的读操作能遍历的数据库的块数目是通过 DB_FILE_MULTIBLOCK_READ_COUNT 初始化参数来设置的。

位图索引给只有极少几个不同值的列提供了一种更为可行的索引方式。位图索引通常用于限制了值(如 Y 和 N)的“标志”列。当多个位图索引都在同一个查询中使用时，位图索引特别有效。优化程序能够快速判定位图并确定哪些行满足使用位图索引的所有条件。

17.7.8　一个表能使用多少个索引

可以在一个表中创建很多索引，一个索引中可以有很多列。索引的列太多会降低插入新行的速度，即在执行 insert 操作时，每个索引也必须拥有一个新项。如果表主要用于查询，那么索引多列(当然，这些列有变化，并将用于 where 子句)的唯一代价是要使用额外的磁盘空间。严格地从数据加载性能这个角度来看，拥有较少的索引但每个索引中列数较多比拥有较多的索引但每个索引中列数较少更好。

除了在群集索引(将在本章后面介绍)中，NULL 列将不出现在索引中。如果任何列都不为

NULL，则基于多列的索引将有一个项。如果某行的所有索引列都为 NULL，则此行在索引中没有项。

17.7.9 在数据库中放置索引

通过给索引分配一个指定的表空间来指定表的索引所放置的位置。正如在第 22 章将详细描述的那样，表空间是数据库的一个逻辑分区，它对应于一个或多个数据文件。数据文件提供数据库使用的物理存储(存储表和索引的磁盘扇区)。数据库有几个表空间，每个表空间都具有自己的名称。为了提高可用性和管理选项，表的索引应该放置在物理上把磁盘驱动器和相应的表分隔开的表空间中。

为了指定放置索引的表空间，应该在普通的 create index 语句后使用关键词 tablespace 和表空间名，如下所示：

```
create unique index BA$TITLE_AUTHOR
    on BOOKSHELF_AUTHOR(Title, AuthorName)
tablespace BA_INDEXES;
```

在这个示例中，BA_INDEXS 是数据库管理员以前创建的一个表空间的名称。在 create index 语句中使用 tablespace 选项允许从物理上分开表和关联的索引。

在创建一个主键或唯一约束时，Oracle 将会自动创建一个索引来加强唯一性。除非另有指定，否则索引将创建在该约束所修改的表所处的同一个表空间中，并为这个表空间使用默认的存储参数。由于通常不希望使用这个存储位置，因此在创建主键和唯一约束时应该优先使用 using index 子句。

using index 子句允许为约束创建的索引指定存储参数和表空间位置。在下例中，在 BOOKSHELF_AUTHOR 表上创建一个主键，并放置在 BA_INDEXS 表空间中。这个示例假定在指定的列上不存在现有索引。

```
alter table BOOKSHELF_AUTHOR
  add constraint BA_PK primary key (Title, AuthorName)
using index tablespace BA_INDEXES;
```

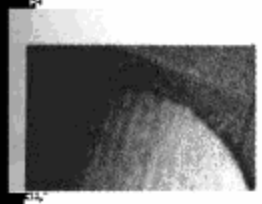
关于 using index 子句的更多选项，请参阅附录 A 中的 Integrity Constraint 部分。关于与性能有关的索引创建选项，请参阅附录 A 中的 create index 部分。

17.7.10 重建索引

Oracle 提供了一种快速的索引重建功能，这种功能允许重新创建一个索引而不必删除已有的索引。当前可用的索引作为该索引的数据源，而不是使用表作为数据源。在重建索引的过程中，可以更改其 storage 参数和 tablespace 值。

在下面的示例中，将重建 BA_PK 索引(通过 rebuild 子句)。在 BA_INDEXES 表空间中，它的存储参数更改为使用 8MB 的初始区以及 4MB 的下一扩展区。

```
alter index BA_PK rebuild
storage (initial 8M next 4M pctincrease 0)
tablespace BA_INDEXES;
```



注意：
在重建 BA_PK 索引时，必须有足够的空间用来同时容纳旧索引和新索引。创建新索引后，将删除旧索引。

当基于以前的索引列创建一个索引时，Oracle 可以使用已有的索引作为新索引的数据源。由于 Oracle 优化程序会根据需要使用已经存在的复合索引的一部分来进行查询，因此不需要创建太多的索引就可以支持大部分常见的查询。

可以在使用 alter index 命令的 rebuild online 子句访问索引时重建这些索引。

17.7.11　基于函数的索引

现在可以创建基于函数的索引。在一个列上执行函数的查询一般不能使用该列的索引。因而，以下这个查询将不能使用 Title 列的索引：

```
select * from BOOKSHELF
where UPPER(Title) = 'MY LEDGER';
```

但是下面这个查询能使用 Title 列的索引，因为它没有在 Title 列上执行 UPPER 函数。

```
select * from BOOKSHELF
where Title = 'MY LEDGER';
```

可以创建由索引访问支持的索引，该索引允许基于函数访问。不要在 Title 列上创建索引，而可以在列表达式 UPPER(Title)上创建索引，如下面的程序清单所示：

```
create index BOOKSHELF$UPPER_TITLE on
BOOKSHELF(UPPER(Title));
```

虽然基于函数的索引是非常有用的，但当创建索引时必须考虑下列问题：

- 能够限制在该列上使用的函数吗？如果能，能够限制在该列上执行的所有函数吗？
- 对于额外的索引有足够的存储空间吗？
- 在删除表时，将比以前删除更多的索引(因此删除更多的区)吗？这会对删除表所需要的时间产生怎样的影响？

虽然基于函数的索引很有用，但应该小心地创建它们。在表上创建的索引越多，insert、update 和 delete 操作所花费的时间就越长。

17.8　群集

群集是存储表的一种方法，这些表密切相关，并经常一起连接到磁盘的同一区域。例如，表BOOKSHELF 和表BOOKSHELF_AUTHOR的数据行可以交错插入到称为群集的单个区域中，而不是将两个表放在磁盘上的不同扇区上。群集键可以是一列或多列，通过这些列通常可以将这些表在查询中连接起来(例如，BOOKSHELF 表和 BOOKSHELF_AUTHOR 表中的 Title 列)。为了将表聚集在一起，必须拥有这些将要聚集在一起的表。

下面是 create cluster 命令的基本格式：

```
create cluster cluster
( column datatype [, column datatype]. . .) [ other options];
```

群集名遵循表命名的约定，列数据类型是将作为群集键使用的名称和数据类型。列名可以与将要放进该群集中的表的一个列名相同，或者为其他任何有效的名称。下面是一个示例：

```
create cluster BOOKandAUTHOR (Col1 VARCHAR2(100));
Cluster created.
```

这样就创建了一个没有任何内容的群集(像给表分配了一块空间一样)。Col1 的使用对于群集键是不相关的，您不会再使用它。但是，它的定义应该与要添加的表的主键匹配。接下来，创建要包含在该群集中的表：

```
create table BOOKSHELF
(Title          VARCHAR2(100) primary key,
Publisher       VARCHAR2(20),
CategoryName    VARCHAR2(20),
Rating          VARCHAR2(2),
constraint CATFK foreign key (CategoryName)
references CATEGORY(CategoryName)
)
cluster BOOKandAUTHOR (Title);
```

在向 BOOKSHELF 表中插入行之前，必须创建一个群集索引：

```
create index BOOKandAUTHORndx
on cluster BOOKandAUTHOR;
```

回顾一下，由于使用了 cluster 子句，因此就不必再使用 tablespace 或 storage 子句。请注意此结构与标准的 create table 语句的不同之处：

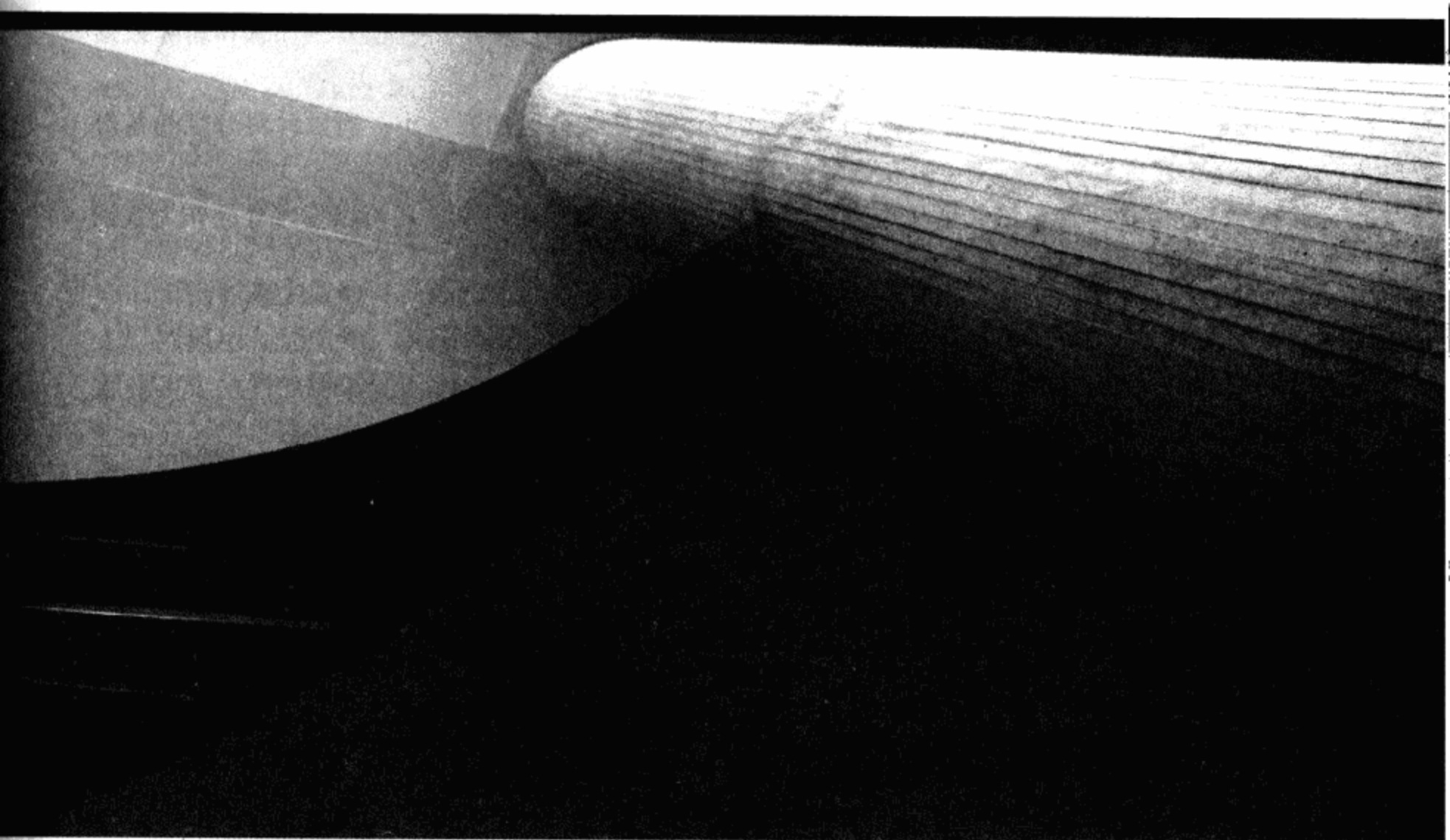
```
create table BOOKSHELF
(Title          VARCHAR2(100) primary key,
Publisher       VARCHAR2(20),
CategoryName    VARCHAR2(20),
Rating          VARCHAR2(2),
constraint CATFK foreign key (CategoryName)
references CATEGORY(CategoryName)
);
```

在第一个 create table 语句中，群集 BOOKandAUTHOR (Title)子句放在表中创建的列的列表的右圆括号的后面。BOOKandAUTHOR 是前面创建的群集的名称。Title 是将存储到群集键 Col1 中的该表的列。create cluster 语句中可能会有多个群集键，并且在 create table 语句中可能有多个列存储在这些键中。请注意，没有任何语句显式地说明 Title 列存储到群集键 Col1 中。这种匹配仅仅是通过位置做到的，即 Col1 和 Title 都是在它们各自的群集语句中提到的第一个对象。多个列和群集键是第一列与第一个群集键匹配，第二列与第二个群集键匹

配，第三列与第三个群集键匹配，依此类推。现在，把第二个表添加到群集中：

```
create table BOOKSHELF_AUTHOR
(Title VARCHAR2(100),
AuthorName VARCHAR2(50),
constraint TitleFK Foreign key (Title) references BOOKSHELF(Title),
```

也可以为快速访问而在内存中缓存序列值，一旦达到序列的最大值，可以使序列循环返回到开始值。在 RAC 环境中，Oracle 建议每个实例缓存 20 000 个序列值，以避免写操作期间的争用。对于非 RAC 环境，至少应该缓存 1 000 个序列值。需要注意的是，如果刷新实例的共享池部分，或者关闭并重新启动数据库，则任意缓存的序列值将会丢失，数据库中存储的序列号将不连贯。请参阅附录 A 中的 create sequence 部分。



- 表更容易管理。因为分区表的数据存储在多个部分中，所以按分区加载和删除数据比在大表中加载和删除数据更容易。
- 备份和恢复操作会执行得更好。因为分区比被分区的表要小，所以针对分区的备份和恢复方法要比备份和恢复整个表的方法多。

Oracle 优化程序将知道哪个表已被分区。正如本章稍后所示，还可以指定分区作为查询的 from 子句的组成部分。

18.1 创建分区表

为创建分区表，需要指定如何创建作为 create table 命令的组成部分的表数据的分区。通常按值的范围对表进行分区(称作范围分区)。

考虑 BOOKSHELF 表：

```
create table BOOKSHELF
    (Title          VARCHAR2(100) primary key,
    Publisher       VARCHAR2(20),
    CategoryName    VARCHAR2(20),
    Rating          VARCHAR2(2),
    constraint CATFK foreign key (CategoryName)
    references CATEGORY(CategoryName));
```

如果要在 BOOKSHELF 表中存储大量的记录，则可以将这些记录分成多个分区。为了对表的记录进行分区，可使用 create table 命令的 partition by range 子句，如下所示。值的范围将决定存储在每个分区中的值。

```
create table BOOKSHELF_RANGE_PART
    (Title          VARCHAR2(100) primary key,
    Publisher       VARCHAR2(20),
    CategoryName    VARCHAR2(20),
    Rating          VARCHAR2(2),
    constraint CATFK2 foreign key (CategoryName)
    references CATEGORY(CategoryName)
    )
    partition by range (CategoryName)
    (partition PART1 values less than ('B')
       tablespace PART1_TS,
    partition PART2 values less than (MAXVALUE)
       tablespace PART2_TS);
```

BOOKSHELF 表将按 CategoryName 列中的值进行分区：

```
partition by range (CategoryName)
```

如果 Category 值小于 'B' (ADULT 目录)，则记录将存储在名为 PART1 的分区中。PART1 分区将被存储在 PART1_TS 表空间中(关于表空间的详细介绍，请参阅第 22 章)。其他分类将

存储在 PART2 分区中。请注意 PART2 分区的定义，它的范围子句为：

```
partition PART2 values less than (MAXVALUE)
```

不必为最后一个分区指定最大值，maxvalue 关键字会告诉 Oracle 使用这个分区来存储在前面几个分区中不能存储的数据。

您可以创建多个分区，每个分区都定义了自己的上限值。对于每一个分区来说，只需指定其范围的最大值即可。范围的最小值由 Oracle 通过前面分区的定义隐式指定。

除范围分区之外，Oracle 还支持散列分区。散列分区通过在分区键值上执行一个散列函数来决定数据的物理位置。在范围分区中，分区键的连续值通常存储在相同的分区中。而在散列分区中，连续的分区键值不必存储在相同的分区中。散列分区把记录分布在比范围分区更多的分区上，这减少了 I/O 争用的可能性。

为了创建一个散列分区，应该用 partition by hash 子句替代 partition by range 子句，如下所示：

```
create table BOOKSHELF_HASH_PART
(Title          VARCHAR2(100) primary key,
Publisher       VARCHAR2(20),
CategoryName    VARCHAR2(20),
Rating          VARCHAR2(2),
constraint CATFK_HASH foreign key (CategoryName)
references CATEGORY(CategoryName)
)
partition by hash (CategoryName)
partitions 16;
```

注意：

考虑到 Oracle 中分区映射的实现方式，建议将表中的分区数设置为 2 的乘方(例如，2、4、8、16)，以便使数据均匀分布。

就像对范围分区所做的一样，可以为每个分区命名并指定其表空间，如下所示：

```
create table BOOKSHELF_HASH_PART
(Title          VARCHAR2(100) primary key,
Publisher       VARCHAR2(20),
CategoryName    VARCHAR2(20),
Rating          VARCHAR2(2),
constraint CATFK_HASH foreign key (CategoryName)
references CATEGORY(CategoryName)
)
partition by hash (CategoryName)
partitions 2
store in (PART1_TS, PART2_TS);
```

在 partition by hash(CategoryName)行的后面，有两种格式可供选择：

- 如前面的程序清单所示，您可以指定所使用的分区数目和表空间：

```
partitions 2
    store in (PART1_TS, PART2_TS);
```

这种方法将创建 SYS_Pnnn 格式的带系统名称的分区。在 store in 子句中指定的表空间的数目不必等于分区的数目。如果指定的分区数目比表空间数目多，则分区将会以循环的方式分配到表空间中。

- 可以指定已命名的分区：

```
partition by hash (CategoryName)
    (partition PART1 tablespace PART1_TS,
     partition PART2 tablespace PART2_TS);
```

用这种方法可以给每个分区一个名称和表空间，还可以选择使用 lob 或 varray 存储子句(请参阅第 39 章和 40 章)。这种方法不仅对分区位置提供了更多控制，而且给每个分区指定了有意义的名称。

18.2 列表分区

您可以使用列表分区代替范围分区和散列分区。在列表分区中，告诉 Oracle 所有可能的值，并指定应该插入相应行的分区。以下示例显示了按 CategoryName 分区的 BOOKSHELF 项列表：

```
create table BOOKSHELF_LIST_PART
    (Title          VARCHAR2(100) primary key,
    Publisher       VARCHAR2(20),
    CategoryName    VARCHAR2(20),
    Rating          VARCHAR2(2),
    constraint CATFK4 foreign key (CategoryName)
    references CATEGORY(CategoryName)
    )
    partition by list (CategoryName)
      (partition PART1 values ('ADULTFIC', 'ADULTNF', 'ADULTREF')
         tablespace PART1_TS,
      partition PART2 values
          ('CHILDRENFIC', 'CHILDRENNF', 'CHILDRENPIC')
         tablespace PART2_TS);
```

另外，可以使用 default 关键字来创建列表分区，以处理没有列出的分类名。使用下面的子句添加分区：

```
partition category_other values (DEFAULT)
```

与其他分区定义中指定的 CategoryName 值不匹配的值将存储在此分区中。

18.3　创建子分区

也可以创建子分区，即分区的分区。可以用子分区将范围分区、列表分区和散列分区等各种类型的分区结合在一起。例如，可以把范围分区和散列分区结合使用，创建范围分区的散列分区。对于非常大的表来说，这种组合分区是一种把数据分成可管理和可调整的组成部分的有效方法。

在下面的示例中，按 Title 列对 BOOKSHELF 表进行了范围分区，并且按 CategoryName 值对 Title 分区进行了散列分区：

```
create table BOOKSHELF_RANGE_HASH_PART
(Title          VARCHAR2(100) primary key,
Publisher      VARCHAR2(20),
CategoryName   VARCHAR2(20),
Rating         VARCHAR2(2),
constraint CATFK3 foreign key (CategoryName)
references CATEGORY(CategoryName)
)
partition by range (Title)
subpartition by hash (CategoryName)
subpartitions 6
  (partition PART1 values less than ('M')
     tablespace PART1_TS,
   partition PART2 values less than (MAXVALUE)
     tablespace PART2_TS);
```

通过为已命名分区指定的 Title 值范围，将 BOOKSHELF 表按范围分为两个分区。每一个分区又按 CategoryName 列进行散列分区。

18.4　创建范围和间隔分区

Oracle 11g 对分区有两个比较大的改善，即增加了引用分区和间隔分区。引用分区可以根据这样一个列来创建分区：虽然此列不在被分区的表中，但它是另一个表的外键引用。这对于以相关的方式分区相关的表是很有用的，即使它们没有相同的列。

例如，INVOICE_HEADERS 表可能包含一个 Invoice_Date 列，用来记录开发票的日期。发票的行条目可能存储在 INVOICE_LINE_ITEMS 表中——从规范化的视角来看，在行条目级别存储 Invoice_Date 是不合适的。但是，如果 Invoice_Date 列不在 INVOICE_LINE_ITEMS 表中，那么您如何能够根据发票日期对 INVOICE_LINE_ITEMS 表进行分区呢？

您可以按照 Invoice_Date 列以月为单位对 INVOICE_HEADERS 表进行范围分区。假定虽然 INVOICE_LINE_ITEMS 表没有 DATE 数据类型列，但 Invoice_Number 列上有一个外键可以引用 INVOICE_HEADERS 表。如何对它分区呢？

在 Oracle 11g 中，可以使用引用分区解决这一问题。使用引用分区的 INVOICE_LINE_

ITEMS 表的创建(假定 INVOICE_HEADERS 表已经存在，并且已被分区)如下面的程序清单所示：

```
create table INVOICE_LINE_ITEMS
    (
       InvoiceNum number not null,
       Line_id        number not null,
       Sales_amt      number,
       constraint     fk_inv_01
         foreign key (InvoiceNum)
         references INVOICE_HEADERS
    )
    partition by reference (fk_inv_01);
```

当创建 INVOICE_LINE_ITEMS 表时，根据对 INVOICE_HEADERS 表分区所使用的列进行分区，即使此列(Invoice_Date)不在 INVOICE_LINE_ITEMS 表中。这两个表将被同步分区。

虽然这似乎只是向前迈了一小步，但对于有效地管理老化的数据有很大的意义。随着时间的流逝，我们需要能够以一致的方式使来自多个表的相关数据一同老化。需要能够删除数据而不必担心在此期间有挂起的引用将使参照完整性无效。通过使用引用分区，您正在将跨多个表划分数据的方式联合起来，这就大大地减轻了维护工作的负担。

在 Oracle 11g 中可用的另一种新的分区类型是间隔分区。在间隔分区中，不必为每个分区指定特定的范围值，相反是指定间隔持续的时间。也就是说，不是指定分区 1 在 1 月 31 日结束，分区 2 在 2 月 29 日结束，而是指定每个分区为 1 个月长的时间。当插入新行时，Oracle 将根据间隔定义确定将此行插入到哪个分区。如果没有为那个月创建分区，则数据库将自动创建一个新的分区。

使用间隔分区需要谨慎。您需要在插入新行之前自己执行数据值约束检查——如果不这样，仅仅因为数据录入人员将年份“2008”错误地输入为“2098”，就可能会创建不想要的分区。INVOICE_HEADERS 表的间隔分区版本如下面的程序清单所示：

```
create table INVOICE_HEADERS
    (
       InvoiceNum number,
       CustomerNum number,
       Invoice_Date date
    )
    partition by range (Invoice_Date)
    interval (numtoyminterval(1,'MONTH'))
    (
      partition p0701 values
        less than (to_date('2007-02-01','yyyy-mm-dd'))
    );
```

需要注意的是，如果依靠间隔分区自动为您创建分区，则应用程序开发人员不能得到一致的分区名称，因为 Oracle 为它自动创建的每个分区创建一个系统生成的名称。

18.5　索引分区

在创建一个分区表时，应该在这个表上创建一个索引。索引也可以按照与对表进行分区时所用的相同范围的值来分区。在下面的程序清单中，给出了 BOOKSHELF_LIST_PART(分区的列表)表的 create index 命令。索引分区放在 PART1_NDX_TS 表空间和 PART2_NDX_TS 表空间中。

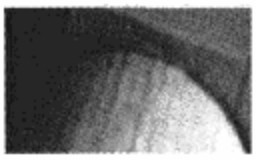
```
create index BOOKSHELF__LIST_CATEGORY
    on BOOKSHELF_LIST_PART(CategoryName)
      local
      (partition PART1
        tablespace PART1_NDX_TS,
      partition PART2
        tablespace PART2_NDX_TS);
```

请注意 local 关键字。在这个 create index 命令中没有指定范围，而是由 local 关键字告诉 Oracle 为 BOOKSHELF_LIST_PART 表的每个分区创建一个单独的索引。因为在 BOOKSHELF_LIST_PART 表上创建了两个分区，所以此索引将创建两个独立的索引分区，每个表分区对应一个索引分区。因为每个分区有一个索引，所以这些索引分区对于分区来说是“局部”的。

也可以创建“全局”索引。全局索引可以包含多个分区的值。

```
create index BOOKSHELF_LIST_CATEGORY_G
    on BOOKSHELF_LIST_PART(Publisher)
     global;
```

该 create index 命令中的 global 子句允许创建非分区索引(如这里所示)，或指定与表的分区范围不同的索引值的范围。虽然局部索引比全局索引容易管理，但是全局索引进行唯一性检查的速度可能会比局部索引(分区索引)快。



注意：
不能为散列分区或子分区创建全局索引。

18.6　管理分区表

可以使用 alter table 命令来对分区进行 add、drop、exchange、move、modify、rename、split 和 truncate 等操作。这些 alter table 命令选项允许更改已有的分区结构，因为一个分区表在频繁使用后可能需要更改其分区结构。例如，分区表中 CategoryName 值的分布可能已经被修改，或者最大值可能增加了。关于这些选项的信息，请参阅“命令和术语参考”中的 ALTER TABLE 项。

在对分区表进行插入操作时，Oracle 使用分区的定义来确定记录应该插入哪个分区中。因此，您可以像使用表一样使用分区表，并依靠 Oracle 来管理数据的内部分离。

分区的一种常见用法是可以使应用程序的停机时间最小化。假定您有一个数据表，它是按天批量加载和分区。当新一天的数据产生时，您创建一个新表来存储这些数据。使新表的结构类似于现有分区表的分区。将数据加载到新表，然后索引此新表。这可以避免数据加载期间与存储索引相关的性能损失。

接下来分析此新表。在分区表中，首先为今天的数据创建一个新的分区，然后使用 alter table exchange partition 命令将这一空的分区交换成新加载的表。遵循这一过程，当您加载、索引和分析分区表时，就可以阻止数据加载对用户访问分区表造成大的影响。

第 19 章

Oracle 基本安全

虽然信息是取得成功的关键，但是若信息遭到破坏或者被窃取，就会对成功造成威胁。Oracle 提供了丰富的安全功能，来保护信息免遭窃取以及有意或无意的破坏。这种安全功能是通过针对每个人逐个授予或取消权限来实现的，不仅是在计算机系统已有的安全功能之外的安全措施，并且独立于计算机系统已有的安全功能。Oracle 利用 create user、create role 和 grant 命令控制数据访问。

本章将介绍 Oracle 中可用的基本安全特性。虚拟专用数据库等 Oracle 高级安全特性参见第 20 章。

19.1 用户、角色和权限

每个 Oracle 用户都有自己的名字和密码，并且拥有他们自己所创建的任意表、视图和其他资源。Oracle 角色是一组权限(或每个用户需要的访问类型，这取决于用户的地位和职责)。您可以授予或给予角色一些特殊的权限，然后将角色分配给适当的用户。一个用户也可以直接授权给其他的用户。

数据库系统权限(database system privilege)允许您执行一组特定的命令。例如，CREATE TABLE 权限允许您创建表。GRANT ANY PRIVILEGE 权限允许您授予任何系统权限。

数据库对象权限(database object privilege)赋予了您在不同的对象上执行某些操作的能力。例如，DELETE 权限允许您从表和视图中删除行。SELECT 权限允许您用 select 命令对表、视图、序列和图(物化视图)进行查询。

关于系统和对象权限的完整列表，请参阅附录 A 中的 PRIVILEGE 部分。

19.1.1 创建用户

Oracle 系统本身带有多个已经创建的用户，包括 SYSTEM 和 SYS。SYS 用户拥有 Oracle 用来管理数据库的核心内部表，而 SYSTEM 用户则拥有其他的表和视图。您可以用 SYSTEM 用户登录来创建其他用户，因为 SYSTEM 用户拥有这个权限。

在安装 Oracle 时，您(或系统管理员)可以首先为自己创建一个用户。下面是 create user 命令的最简单的格式：

```
create user user identified
        {by password | externally | globally as 'extnm'};
```

很多其他的账户特性也可以通过此命令来设置。详细内容请参阅本书附录 A 中的 create user 命令。

为了将计算机系统的用户 ID 和密码与 Oracle 的安全机制结合起来，以便基于主机的用户只需登录一次，可以使用 externally，而不是给出密码。系统管理员(有很多权限)可能需要一个独立的密码以便更加安全。在下面的示例中，我们称这个系统管理员为 Dora：

```
create user Dora identified by avocado;
```

现在，Dora 的账户已经存在并且由一个密码保护。

您也可以创建具有特定的表空间以及空间和资源限额(限制)的用户。关于表空间以及资源的介绍，请参阅附录 A 中的 CREATE USER 部分以及第 17 和 22 章中有关于此问题的讨论。

可使用 alter user 命令来修改密码：

```
alter user Dora identified by psyche;
```

现在 Dora 的密码是 psyche 而不是 avocado。但是，除非 Dora 首先具有 CREATE SESSION 的系统权限，否则她不能登录到她的账户中去：

```
grant CREATE SESSION to Dora;
```

在本章的后面，您将看到关于授予系统权限的其他示例。

19.1.2　密码管理

密码会过期，并且账户有可能会由于多次连续失败而被锁定。在您修改了密码之后，需要维护密码历史记录，以防止重用以前的密码。

账户密码到期的特性是由分配给账户的配置文件决定的。配置文件由 create profile 命令创建，并由 DBA(数据库管理员，将在本章后面介绍)管理。关于 create profile 命令的详细内容，请参阅附录 A 中的 CREATE PROFILE 部分。

关于密码和账户的访问，配置文件规定了下列内容：

- 密码的生存期(Lifetime)，它决定了多长时间就必须更改密码。
- 密码的“过期日期”后的宽限期，在此期间您可以修改密码。
- 在账户自动“锁定”之前允许连续失败的次数。
- 账户保持锁定状态的天数。
- 在重用一个密码前必须经过的天数。
- 在重用一个密码前该密码必须经历的更改次数。

附加的密码管理功能允许使用长度和复杂性均最小的密码。

除了 alter user 命令之外，还可以在 SQL*Plus 中使用 password 命令来修改您的密码。如果使用 password 命令，那么输入的新密码不会显示在屏幕上。数据库管理员可以利用 password 命令修改任何用户的密码，而其他的用户只能修改自己的密码。

在您输入 password 命令时，您将被提示输入旧密码和新密码，如下面的程序清单所示：

```
connect dora/psyche
password
Changing password for dora
Old password:
New password:
Retype new password:
```

当成功地修改密码以后，您将会收到如下反馈信息：

```
Password changed
```

通过 DBA 账号您可以设置另外一个用户的密码。只需要简单地将用户名追加到 password 命令之后即可。而且还不要求您输入旧密码：

```
password dora
Changing password for dora
New password:
Retype new password:
```

1. 密码过期机制

您可以使用配置文件来管理密码的过期、重用和复杂性。您可以限定一个密码的生存期，并锁定密码使用太久的账户。您还可以使密码的复杂性尽可能地降低，并将已经多次登录失败的账户锁住。

例如，如果您将用户的配置文件中的 FAILED_LOGIN_ATTEMPTS 资源设置为 5，则表示只允许该账户连续 4 次登录失败，第 5 次失败该账户将被锁定。

注意：
如果第 5 次给出了正确的密码，则 "failed login attempt count" 将重新置为 0，即在账户锁定之前还允许另外 5 次连续的失败登录。

在以下程序清单中，创建了 LIMITED_PROFILE 配置文件，该文件由用户 JANE 使用：

```
create profile LIMITED_PROFILE limit
FAILED_LOGIN_ATTEMPTS 5;

create user JANE identified by EYRE
profile LIMITED_PROFILE;

grant CREATE SESSION to JANE;
```

如果 JANE 账户连续 5 次连接失败，则该账户将被 Oracle 自动锁定。然后，当您用正确的密码登录 JANE 的账户时，将收到一条错误消息：

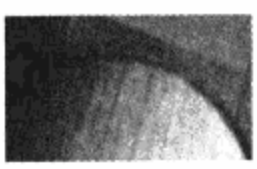
```
connect jane/eyre
ERROR:
ORA-28000: the account is locked
```

要想将账户解锁，可以使用 alter user 命令的 account unlock 子句(用 DBA 账户)，如以下程序清单所示：

```
alter user JANE account unlock;
```

在该账户解锁之后，将允许再次连接到 JANE 账户。您可以通过 alter user 命令的 account lock 子句手工锁定一个账户。

```
Alter user JANE account lock;
```



注意：
您可以指定 account lock 为 create user 命令的一部分。

如果一个账户由于多次连接失败而被锁定，则当超过配置文件设置的 PASSWORD_LOCK_TIME 值时，它将自动解锁。例如，如果 PASSWORD_LOCK_TIME 设置为 1，则上例中的 JANE 账户将被锁定一天，一天之后该账户自动解锁。

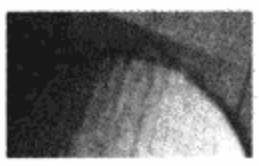
可以通过配置文件中的 PASSWORD_LIFE_TIME 资源设置密码的最大寿命。例如，可以使用 LIMITED_PROFILE 配置文件强制用户每隔 30 天修改一次密码。

```
Alter profile LIMITED_PROFILE limit
    PASSWORD_LIFE_TIME 30;
```

在这个示例中，使用 alter profile 命令来修改 LIMITED_PROFILE 配置文件。由于 PASSWORD _LIFE_TIME 的值设置为 30，因此，使用该配置文件的每个账户将在 30 天后密码到期。如果您的密码已经到期，则您必须在下一次登录时修改它，除非配置文件对到期的密码有一个特定的宽限期。宽限期参数称为 PASSWORD_GRACE_TIME。如果密码在宽限期内未修改，则账户将过期。

注意：
如果打算使用 PASSWORD_LIFE_TIME 参数，则需要给用户提供一种修改其密码的简单方法。

“过期”的账户与“锁定”的账户不同。如本节前面所述，锁定的账户可在一段时间后自动解锁，但是，过期的账户需要由 DBA 手工设置才能重新生效。



注意：
如果要使用密码过期功能，就要确保拥有应用程序的各个账户有不同的配置文件设置；否则，这些账户可能被锁定，并且不能使用应用程序。

为了使到期的账户重新启用，应该执行 alter user 命令。在如下示例中，DBA 手工使 JANE 的密码过期：

```
alter user jane password expire;

User altered.
```

接着，JANE 试图连接她的账户。当她给出密码时，系统立即提示她输入该账户的新密码。

```
connect jane/eyre
ERROR:
ORA-28001: the password has expired

Changing password for jane
New password:
Retype new password:
Password changed
Connected.
```

您还可以强制用户在第一次访问他们的账户时修改自己的密码，通过 create user 命令的 password expire 子句来修改密码。但是，create user 命令不允许您给用户设置的新密码设定到期日期；要想为用户设置的新密码设置到期日期，必须使用前面示例中所示的 PASSWORD_LIFE_TIME 配置文件的参数。

为了查看任意账户的密码到期日期，可查询 DBA_USERS 数据字典视图的 Expiry_Date 列。想要查看账户的密码到期日期的用户可以查询 USE_USERS 数据字典视图的 Expiry_Date 列(通过 SQL*Plus 或基于客户端的查询工具)。

2. 密码重用限制

为了防止密码被重用，可以使用配置文件中的两个参数：PASSWORD_REUSE_MAX 或 PASSOWRD_REUSE_TIME。这两个参数是相互排斥的，即如果您设置了其中一个，则另一个必须设置为 UNLIMITED。

PASSWORD_REUSE_TIME 参数指定在一个密码可以重用前必须间隔的天数。例如，如果设置 PASSWORD_REUSE_TIME 为 60，则在 60 天内您不能重用同样的密码。

PASSWORD_REUSE_MAX 参数指定在一个密码重用前密码必须变化的次数。如果您试图在到达该限定值前重用该密码，则 Oracle 将拒绝您修改密码。

例如，您可以为本章前面创建的 LIMITED_PROFILE 配置文件设置 PASSWORD_REUSE_MAX 参数：

```
alter profile LIMITED_PROFILE limit
PASSWORD_REUSE_MAX 3
PASSWORD_REUSE_TIME UNLIMITED;
```

如果用户 JANE 现在试图重用一个近期的密码，则密码变更将失败。例如，假设她修改了自己的密码，如下面的程序行所示：

```
alter user JANE identified by austen;
```

然后她再次修改密码：

```
alter user JANE identified by whitley;
```

当再次修改密码时，她试图重用一个近期的密码，但重用失败：

```
alter user jane identified by austen;
alter user jane identified by austen
*
ERROR at line 1:
ORA-28007: the password cannot be reused
```

她不能重用任何最近的密码，她必须使用一个新密码。

19.1.3 标准角色

本章前面创建了一个名为 Dora 的用户。既然 Dora 已经有一个账户，那么她能在 Oracle 中干什么呢？此时，她什么也不能做，因为 Dora 除了 CREATE SESSION 以外没有其他系统权限。在大多数情况下，应用程序用户通过角色来获得权限。您可以依据应用程序用户的需要将系统权限和对象访问融入角色当中。可以为应用程序访问创建您自己的角色，还可以为一些系统访问需求而使用 Oracle 的默认角色。在数据库创建期间创建的最重要的标准角色

如表 19-1 所示。

表 19-1　数据库创建期间最重要的标准角色

角　　色	说　　明
CONNECT、RESOURCE 和 DBA	为与 Oracle 数据库软件的以前版本兼容而提供的角色
DELETE_CATALOG_ROLE、 EXECUTE_CATALOG_ROLE、 SELECT_CATALOG_ROLE	这些角色是为访问数据字典视图和包而提供的
EXP_FULL_DATABASE、 IMP_RULL_DATABASE	这些角色是为了方便地使用输入输出实用程序而提供的
AQ_USER_ROLE、 AQ_ADMINISTRATOR_ROLE	这些角色是 Oracle 高级查询所需的
SNMPAGENT	这个角色用于 Enterprise Manager Intelligent Agent
RECOVERY_CATALOG_OWNER	这个角色是创建一个恢复目录模式拥有者所需的
HS_ADMIN_ROLE	这个角色是支持异类服务所需的
SCHEDULER_ADMIN	这个角色允许被授权者执行 DBMS_SCHEDULER 包的过程；应当限于 DBA

因为 CONNECT、RESOURCE 和 DBA 是为了向后兼容而提供的，所以不应再使用了。CONNECT 向用户提供了登录和执行基本函数的能力。在 Oracle 11g 中，拥有 CONNECT 角色的用户可以不再创建表、视图、序列、群集、同义词以及数据库链接。拥有 RESOURCE 角色的用户可以创建自己的表、序列、过程、触发器、数据类型、运算符、索引类型、索引和群集。拥有 RESOURCE 角色的用户还可以获得 UNLIMITED TABLESPACE 系统权限，这将允许这些用户忽略所有表空间上的限额。拥有 DBA 角色的用户可以执行数据库管理功能，包括创建和更改用户、表空间和对象。

为了替换 CONNECT、RESOURCE 和 DBA，应当创建您自己的角色以便有权限执行特定的系统权限。在下面几节中，您将会看到如何为用户和角色授予权限。

19.1.4　grant 命令的格式

以下是系统权限的 grant 命令的通用格式(请参阅附录 A 以了解完整的语法)：

```
grant {system privilege | role | all [privileges] }
      [, {system privilege | role | all [privileges] }. . .]
      to {user | role} [, {user | role}]. . .
      [identified by password ]
      [with admin option]
```

您可以将任何系统权限或角色赋予其他用户、其他角色，或者授予 public。with admin option 子句能够保证被授权者将权限或角色授予其他用户或角色。all 子句授予用户或角色除 SELECT ANY DICTIONARY 系统权限以外的所有权限。

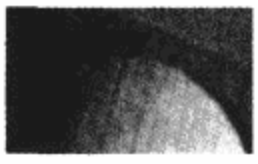
授权者也可以从用户那里撤消一个角色。

19.1.5 撤消权限

可以撤消已经授予的权限。revoke 命令的语法类似于 grant 命令：

```
revoke {system privilege | role | all [privileges] }
       [, {system privilege | role | all [privileges] }. . .]
       from {user | role} [, {user | role}]. . .
```

具有 DBA 角色的个人可以撤消任何人(包括另一个 DBA)的 CONNECT、RESOURCE、DBA 或其他权限或角色。当然，这是非常危险的，这就是为什么除了极少部分确实需要这个权限的人外，不轻易给人授予 DBA 权限的原因所在。

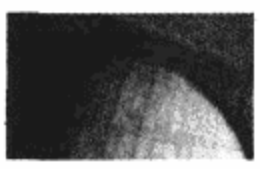


注意：
从一个指定用户那里撤消所有权限并没有从 Oracle 中删除那个用户，也没有销毁该用户所创建的表，而只是禁止该用户访问这些表。对这些表有访问权的其他用户仍然具有原来的访问权限。

为了删除一个用户以及该用户所拥有的所有资源，可使用 drop user 命令，如下所示：

```
drop user username [cascade];
```

cascade 选项可以删除这个用户及其拥有的所有对象，其中包括参照完整性约束。cascade 选项使得引用已删除的用户模式中对象的视图、同义词、存储过程、函数或程序包失效。如果您不使用 cascade 选项并且用户拥有的对象仍然存在，则 Oracle 不会删除该用户，相反还会返回一个错误消息。



注意：
关于 grant 命令的所有选项，请参见附录 A 中的 Privileges 条目。

19.2 可以授予用户何种权限

用户可以给他所拥有的任何对象授权。数据库管理员可以授予任何系统权限。

假设用户 Dora 拥有 COMFORT 表并且是数据库管理员。她用这些权限创建了两个新用户：Bob 和 Judy：

```
create user Judy identified by sarah;

User created.

grant CREATE SESSION to Judy;

Grant succeeded.

create user Bob identified by carolyn;
```

```
User created.

grant CREATE SESSION, CREATE TABLE, CREATE VIEW,
CREATE SYNONYM to bob;

Grant succeeded.

alter user bob
default tablespace users
quota 5m on users;

User altered.
```

以上的命令序列同时让 Judy 和 Bob 能够连接到 Oracle，并且又赋予 Bob 一些额外的权限。但是，他们都能对 Dora 的表进行访问吗?不行，因为他们没有显式的访问权。

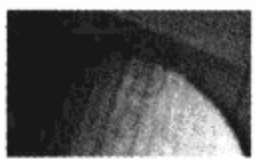
为了使其他人能访问您的表，应该使用 grant 命令的另一种格式：

```
grant { object privilege | all [privileges] }
    [(column [, column] . . .)]
    [, { object privilege | all [privileges] }
    [(column [, column] . . .)] ] . . .
    on object to {user | role}
    [with grant option]
    [with hierarchy option];
```

用户可以授予的权限包括：

- 在用户的表、视图和物化视图上进行如下操作：

  FLASHBACK
  INSERT
  UPDATE(所有列或特定的列)
  DELETE
  SELECT



注意：
如果物化视图是可更新的，那么 INSERT、UPDATE 和 DELETE 权限只能授予物化视图。有关创建物化视图的详细内容，请参阅第 26 章。

- 对于表，您还可以授予以下权限：

  ALERT(表——所有列或特定的列)
  DEBUG
  REFERENCES
  INDEX(表中的列)
  ON COMMIT REFRESH
  QUERY REWRITE
  ALL(前面列出的所有条目)

- 在过程、函数、程序包、抽象数据类型、库、索引类型和运算符上可以执行如下操作：

  EXECUTE

```
DEBUG
```

- 在序列上可以执行如下操作：

```
SELECT
ALTER
```

- 在目录(用于 BFILE LOB 数据类型和外部表)上可以执行如下操作

```
READ
WRITE
```

- 在对象类型和视图上可以执行如下操作：

```
UNDER(可以在视图上创建子视图或在类型上创建子类型)
```

当您在执行另一个用户的过程或函数时，它一般利用自己的权限执行。这就意味着您不需要对过程或函数使用的数据进行显示访问。您只能看到执行的结果，而看不到基础数据。您还可以创建存储过程，该存储过程在调用用户的权限下，而不是过程拥有者的权限下执行。

Dora 授予 Bob 访问 COMFORT 表的 SELECT 权限：

```
grant select On COMFORT to Bob;

Grant succeeded.
```

grant 命令中的 with grant option 子句允许权限的接收者将授予他的权限再授予其他用户。如果用户 Dora 用 with grant option 子句将其表上的权限授予用户 Bob，则 Bob 可以再将从 Dora 的表上得到的权限授予其他用户(Bob 只能转授那些已经授予他的权限，如 SELECT)。如果您打算基于另一个用户的表创建视图并且把这些视图的访问权授予其他用户，则您必须具有对基表的 with grant option 权限。

19.2.1 利用 connect 移动到另一个用户

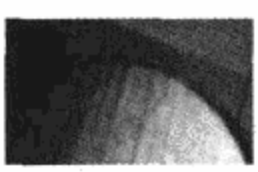
为了测试授权是否成功，Dora 用 connect 命令连接到 Bob 的用户名。该命令可以通过以下任意一种方法使用：

- 在命令行上，输入该用户名和密码
- 先输入命令然后对提示信息作出响应
- 先输入命令和用户名，然后响应密码的提示

由于后两种方法都不显示密码，因此比较安全。下面给出一个连接到数据库的示例：

```
connect Bob/carolyn

Connected.
```

一旦连接完毕，Dora 就可以从 Bob 已经具备 SELECT 访问权限的表中进行选择。



注意：
除非使用同义词，否则表名必须跟在表的拥有者的用户名之后。否则，Oracle 会认为这个表不存在。

```
select * from Dora.COMFORT;

CITY            SAMPLEDAT   NOON   MIDNIGHT PRECIPITATION
--------------  ---------  -----  -------- -------------
SAN FRANCISCO   21-MAR-03   62.5       42.3            .5
SAN FRANCISCO   22-JUN-03   51.1       71.9            .1
SAN FRANCISCO   23-SEP-03   61.5                       .1
SAN FRANCISCO   22-DEC-03   52.6       39.8           2.3
KEENE           21-MAR-03   39.9       -1.2           4.4
KEENE           22-JUN-03   85.1       66.7           1.3
KEENE           23-SEP-03   99.8       82.6
KEENE           22-DEC-03   -7.2       -1.2           3.9
```

下面的代码创建了一个名为 COMFORT 的视图，它只是直接从 Dora.COMFORT 表中选取数据。

```
create view COMFORT as select * from Dora.COMFORT;

View created.
```

从这个视图中进行选择所产生的结果和从 Dora.COMFORT 表中进行选择所产生的结果完全一样：

```
select * from COMFORT;

CITY            SAMPLEDAT   NOON   MIDNIGHT PRECIPITATION
--------------  ---------  -----  -------- -------------
SAN FRANCISCO   21-MAR-03   62.5       42.3            .5
SAN FRANCISCO   22-JUN-03   51.1       71.9            .1
SAN FRANCISCO   23-SEP-03              61.5            .1
SAN FRANCISCO   22-DEC-03   52.6       39.8           2.3
KEENE           21-MAR-03   39.9       -1.2           4.4
KEENE           22-JUN-03   85.1       66.7           1.3
KEENE           23-SEP-03   99.8       82.6
KEENE           22-DEC-03   -7.2       -1.2           3.9
```

现在，Dora 返回自己的用户名并创建一个只选择部分 COMFORT 表的视图：

```
connect Dora/psyche
Connected.

create view SOMECOMFORT as
select * from COMFORT
 where City = 'KEENE';

View created.
```

然后她将这个视图上的 SELECT 和 UPDATE 权限授予 Bob，并且撤消 Bob(通过 revoke 命令)对整个 COMFORT 表的所有权限：

```
grant select, update on SOMECOMFORT to Bob;

Grant succeeded.

revoke all on COMFORT from Bob;

Revoke succeeded.
```

然后 Dora 重新连接到 Bob 的用户名以测试更改的效果：

```
connect Bob/carolyn
Connected.

select * from COMFORT;
                    *
ERROR at line 1:
ORA-04063: view "BOB.COMFORT" has errors
```

尝试选择 COMFORT 视图的操作失败了，这是因为在 Bob 的视图中，命名的基表是 Dora.COMFORT 表。毫不奇怪，若对 Dora.COMFORT 表进行选择则将出现同样的消息。接下来，看一看对 Dora.SOMECOMFORT 表进行选择的结果：

```
select * from Dora.SOMECOMFORT;

CITY           SAMPLEDAT   NOON   MIDNIGHT  PRECIPITATION
-------------- ---------  -----   --------  -------------
KEENE          21-MAR-03   39.9      -1.2            4.4
KEENE          22-JUN-03   85.1      66.7            1.3
KEENE          23-SEP-03   99.8      82.6
KEENE          22-DEC-03   -7.2      -1.2            3.9
```

即使对 COMFORT 表的直接访问权限已经被撤消，这项工作也会完成得很好，这是因为 Dora 赋予 Bob 通过 SOMECOMFORT 视图访问部分 COMFORT 表的权限。这仅仅是表中关于 KEENE 的部分。

这说明了 Oracle 强大的安全特性：您可以在列中使用任何约束或任何计算来创建视图，然后把访问视图而不是访问基表的权限授予其他用户。这些用户只能看到视图显示的信息。这种安全机制甚至可以扩展到用户指定的内容。19.2.16 节将对这一特性进行完整详细的讨论。

现在，以 Bob 的用户名、基于 SOMECOMFORT 视图创建 LITTLECOMFORT 视图：

```
create view LITTLECOMFORT as select * from Dora.SOMECOMFORT;

View created.
```

然后更新 23-SEP-03 那一行：

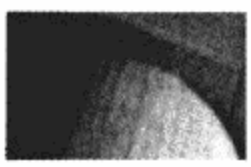
```
update LITTLECOMFORT set Noon = 88
 where SampleDate = '23-SEP-03';
```

```
1 row updated.
```

当查询视图 LITTLECOMFORT 时，它将显示 update 后的结果：

```
select * from LITTLECOMFORT;

CITY           SAMPLEDAT   NOON    MIDNIGHT PRECIPITATION
-------------- ---------   -----   -------- -------------
KEENE          21-MAR-03    39.9      -1.2           4.4
KEENE          22-JUN-03    85.1      66.7           1.3
KEENE          23-SEP-03      88      82.6
KEENE          22-DEC-03    -7.2      -1.2           3.9
```

对 Dora.SOMECOMFORT 进行的查询将会显示相同的结果，就像 Dora 在自己的用户名上进行 COMFORT 表的查询一样。这个 update 对于基表也可成功执行，即使这项操作是通过两个视图(LITTLECOMFORT 和 SOMECOMFORT)实现的。



注意：
您需要授予用户对任意表执行 SELECT 操作的访问权限，这样才能使用户更新和删除记录。这遵守了 ANSI 标准，并且反映了拥有表的 UPDATE 或 DELETE 权限的用户能够使用数据库的反馈信息来发现关于基础数据的信息这一事实。

19.2.2　创建同义词

创建一个视图，使它包含其他用户的整个表或视图的另一种方法是创建同义词：

```
create synonym LITTLECOMFORT for Dora.SOMECOMFORT;
```

这个同义词完全可以像视图一样使用。请参阅附录 A 中的 create synonym 命令。

19.2.3　使用未授权的权限

下面假设打算删除一个刚刚更新过的行：

```
delete from LITTLECOMFORT where SampleDate = '23-SEP-03';
                *
ERROR at line 1:
ORA-01031: insufficient privileges
```

由于 Dora 没有授予 Bob DELETE 权限，因此此操作失败。

19.2.4　权限的传递

虽然 Bob 可授权其他用户访问他的表，但是不能授权其他用户访问那些不属于他的表。下面是 Bob 试图给 Judy 授予 INSERT 权限的示例：

```
grant insert on Dora.SOMECOMFORT to Judy;
        *
```

```
ERROR at line 1:
ORA-01031: insufficient privileges
```

因为 Bob 没有 INSERT 权限，所以以上操作失败了。接下来，Bob 试图传递他具有的 SELECT 权限：

```
grant select on Dora.SOMECOMFORT to Judy;
       *
ERROR at line 1:
ORA-01031: insufficient privileges
```

他也不能授予这个权限，因为视图 SOMECOMFORT 并不属于他。如果使用子句 with grant option 授予他访问 SOMECOMFORT 视图的权限，上面的 grant 命令就会成功。由于视图 LITTLECOMFORT 确实属于他，因此他可以将此视图的权限传递给 Judy。

```
grant select on LITTLECOMFORT to Judy;

ERROR at line 1:
ORA-01720: grant option does not exist for 'DORA.SOMECOMFORT'
```

因为 LITTLECOMFORT 视图依赖于 Dora 的一个视图，并且 Bob 未被使用 with grant option 子句授予那个视图上的 SELECT 权限，所以 Bob 的授权失败了。

此外，可以用 Bob 的 LITTLECOMFORT 视图中的当前信息创建并加载属于 Bob 的一个新表：

```
create table NOCOMFORT as
select * from LITTLECOMFORT;

Table created.
```

NOCOMFORT 表上的 SELECT 权限也被授予 Judy：

```
grant select on NOCOMFORT to Judy;

Grant succeeded.
```

可以从 Judy 的账户中查询由 Bob 向 Judy 授予了 SELECT 权限的 NOCOMFORT 表：

```
connect Judy/sarah
select * from Bob.NOCOMFORT;

CITY            SAMPLEDAT   NOON   MIDNIGHT PRECIPITATION
------------    ---------  -----   -------- -------------
KEENE           21-MAR-03   39.9      -1.2            4.4
KEENE           22-JUN-03   85.1      66.7            1.3
KEENE           23-SEP-03     88      82.6
KEENE           22-DEC-03   -7.2      -1.2            3.9
```

这次授权成功。用 Bob 的用户名创建了一个 NOCOMFORT 表，此表为 Bob 所拥有。Bob 可以成功地授予这个表的访问权。

如果 Dora 希望 Bob 能够将他的权限转授给其他人，则她可以给 grant 语句添加另一个子句：

```
connect Dora/psyche
grant select, update on SOMECOMFORT to Bob with grant option;

Grant succeeded.
```

如果您是一位 DBA，或者被授予了 GRANT ANY PRIVILEGE 系统角色，那么您可以将系统权限(如 CREATE SESSION、CREATE SYNONYM 和 CREATE VIEW 等)授予角色。然后，这些权限对角色的任何用户都是可用的。

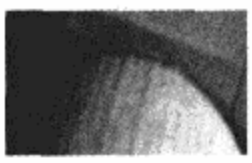
登录数据库的功能是通过 CREATE SESSION 系统权限授予的。下面的示例将这一权限授予了 CLERK 角色。这一权限和 CREATE VIEW 系统权限也授予了 MANAGER 角色。

```
grant CREATE SESSION to CLERK;
grant CREATE SESSION, CREATE VIEW to MANAGER;
```

19.2.7 将一个角色授予另一个角色

角色可以授予给另外的角色。可以通过 grant 命令达到这个目的，如下所示：

```
grant CLERK to MANAGER;
```



注意：
因为不能进行循环授权，所以现在就不能将 MANAGER 授予 CLERK。

在这个示例中，CLERK 角色被授予角色 MANAGER。即使您没有给 MANGER 角色直接授予任何表的权限，角色 MANAGER 也会继承授予角色 CLERK 的所有权限。以这种方式组织角色是一种常见的层次结构设计方式。

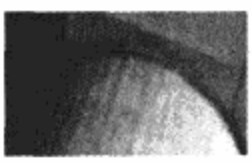
在将一个角色授予另一个角色(或用户，后面将介绍)时，您可以使用 with admin option 子句为角色授权，如下所示：

```
grant CLERK to MANAGER with adnin option;
```

如果使用了 with admin option 子句，那么被授予者有权将这个角色授予其他的用户或角色。被授予者也能更改和删除这个角色。

19.2.8 为用户授予角色

可以将角色授予用户。当授予用户后，角色可看作命名的一组权限。不是将每个权限授予每个用户，而是将这些权限统一授予一个角色，然后再把这个角色授予每个用户。这样极大地简化了权限管理工作。



注意：
在对视图、过程、函数、程序包或者外键进行操作时，不能使用通过角色授予用户权限。在创建了这些类型的数据库对象后，您必须依靠所需权限的直接授权。

可以通过 grant 命令给用户进行角色的授权，如下所示：

```
grant CLERK to Bob;
```

本例中的用户 Bob 将拥有授予角色 CLERK 的所有权限(COMFORT 表上的 CREATE SESSION 和 SELECT 权限)。

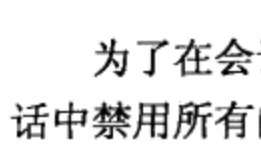
在将一个角色授予用户时，您可以使用 with admin option 子句授予角色，如下所示：

认情况下，角色没有密码。

```
alter role MANAGER not identified;
```

19.2.11 启用和禁用角色

在用户的账户被更改时，可以通过 alter user 命令的 default role 子句来为该用户创建默认角色列表。这个命令的默认动作是将用户的所有角色设置为默认角色，每当该用户登录时启用所有这些角色。



注意：
在任何时候，用户可启用的角色的最大数目是通过MAX_ENABLED_ROLES数据库初始化参数设置的。

alter user 命令的这部分的语法如下所示：

```
alter user username
    default role {[role1, role2]
    [all | all except role1, role2][NONE]};
```

如该语法所示，在默认情况下，用户可选择启用特定的角色、启用所有角色、启用除特定角色以外的所有角色，或者不启用任何角色。例如，下面的 alter user 命令将在 Bob 登录的任何时候都启用 CLERK 角色：

```
alter user Bob
    default role CLERK;
```

可以使用 set role 命令来启用一个非默认角色，如下所示：

```
set role CLERK;
```

如需查看在您当前会话中启用了哪些角色，可以从 SESSION_ROLES 数据字典视图中选择相关信息。查询 SESSION_PRIVS 可以知道当前启用的系统权限。

您也可以使用在 alter user 命令中可用的 all 子句和 all except 子句：

```
set role all;
    set role all except CLERK;
```

如果角色有一个相关的密码，则该密码必须通过 identified by 子句指定：

```
set role MANAGER identified by cygnusxi;
```

为了在会话中禁用一个角色，您可以使用如下的 set role none 命令。此命令将在当前会话中禁用所有的角色。在禁用了所有角色之后，可以重新启用想要的角色。

```
set role none;
```

因为您将会发现需要不断地执行 set role none 命令，所以可以将 CREATE SESSION 权限直接授予用户而不是通过角色授予用户。

19.2.12　撤消角色的权限

可以使用本章前面介绍过的 revoke 命令来撤消角色的权限。用下例所示方法指定权限、对象名(如果它是对象权限)和角色名：

```
revoke SELECT on COMFORT from CLERK;
```

CLERK 角色的用户以后就不能查询 COMFORT 表了。

19.2.13　删除角色

可以使用 drop role 命令来删除角色，如下所示：

```
drop role MANAGER;
drop role CLERK;
```

您所指定的角色及其相关的权限将全部从数据库中删除。系统和对象权限的 grant 和 revoke 操作会立即生效。而角色的 grant 和 revoke 操作仅当当前用户发出 set role 命令或启动一个新的用户会话时才会生效。

19.2.14　给指定的列授予 UPDATE 权限

您可能希望授予用户 SELECT 权限的列比授予用户 UPDATE 权限的列能更多一些。因为 SELECT 列可以通过视图进行控制，从而进一步限制可更新的列需要一种特殊的用户 grant 命令的形式。下面是用于 COMFORT 表的两列的实例：

```
grant update (Noon, Midnight) on COMFORT to Judy;
```

19.2.15　撤消对象权限

如果可以授予对象权限，那么也可以撤消这些权限。这和 grant 命令类似：

```
revoke { object privilege | all [privileges]}
       [(column [, column]. . . )]
[, { object privilege | all [privileges]}
       [(column [, column]. . . )]]. . .]
on object
from {user | role} [,{user | role}]
[cascade constraints] [force];
```

revoke all 将删除以前列出的所有权限，从 SELECT 到 INDEX；撤消个别权限后将留下其他被授予的权限。with grant option 将与它所依附的权限一起被撤消；然后 revoke 操作将级联进行，导致那些通过 with grant option 用户获得访问权限的用户也将撤消他们的权限。

如果用户在对象上定义了参照完整性约束，那么使用 cascade constraints 选项来取消对象上的权限，会使 Oracle 删除这些约束。force 选项适用于对象类型。force 选项将撤消带有表或类型依赖关系的对象类型上的 EXECUTE 权限；所有依赖的对象都将标记为无效，且相关表的数据将不可访问，除非重新授予必要的权限。

19.2.16 用户安全性

可以具体地、逐个表、逐个视图地把表的访问权限授予每个用户或角色。但是，在某种情况下可以应用简化这一过程的技术。请回忆一下 BOOKSHELF_CHECKOUT 记录：

```
select Name, SUBSTR(Title,1,30) from BOOKSHELF_CHECKOUT;

NAME                      SUBSTR(TITLE,1,30)
------------------------- ------------------------------
JED HOPKINS               INNUMERACY
GERHARDT KENTGEN          WONDERFUL LIFE
DORAH TALBOT              EITHER/OR
EMILY TALBOT              ANNE OF GREEN GABLES
PAT LAVAY                 THE SHIPPING NEWS
ROLAND BRANDT             THE SHIPPING NEWS
ROLAND BRANDT             THE DISCOVERERS
ROLAND BRANDT             WEST WITH THE NIGHT
EMILY TALBOT              MIDNIGHT MAGIC
EMILY TALBOT              HARRY POTTER AND THE GOBLET OF
PAT LAVAY                 THE MISMEASURE OF MAN
DORAH TALBOT              POLAR EXPRESS
DORAH TALBOT              GOOD DOG, CARL
GERHARDT KENTGEN          THE MISMEASURE OF MAN
FRED FULLER               JOHN ADAMS
FRED FULLER               TRUMAN
JED HOPKINS               TO KILL A MOCKINGBIRD
DORAH TALBOT              MY LEDGER
GERHARDT KENTGEN          MIDNIGHT MAGIC
```

为了使每个借阅者都能够访问这个表，但限制只能访问每个借阅者自己的单行视图，您可以创建 19 个独立的视图，每个视图在 where 子句中有一个不同的名字，并且您可以针对每个借阅者对每个视图进行单独的授权。此外，您还可以创建其 where 子句中包含 User(伪列)的一个视图，如下所示：

```
create or replace view MY_CHECKOUT as
select * from BOOKSHELF_CHECKOUT
  where SUBSTR(Name,1,INSTR(Name,' ')-1) = User;
```

BOOKSHELF_CHECKOUT 表的所有者可以创建该视图并将其 SELECT 权限授予用户。然后，一个用户可以通过 MY_CHECKOUT 视图查询 BOOKSHELF_CHECKOUT 表。where 子句将在 Name 列中找到其用户名(参见 where 子句)，并只返回该用户的记录。

在下例中，将创建一个用户 jed，并授予他对 MY_CHECKOUT 视图的访问权限。从该视图中进行选择将只得到由他所借阅的书籍。下面的示例假定拥有 BOOKSHELF_CHECKOUT 表的模式命名为 PRACTICE，并且该用户是一名数据库管理员。

```
create user jed identified by hop;

User created.
```

```
grant CREATE SESSION to jed;

Grant succeeded.

grant select on MY_CHECKOUT to jed;

Grant succeeded.

connect jed/hop

Connected.

select * from Practice.MY_CHECKOUT;

NAME
-------------------------
TITLE
-----------------------------------------------------------
CHECKOUTD  RETURNEDD
---------  ---------
JED        HOPKINS
INNUMERACY
01-JAN-02 22-JAN-02

JED HOPKINS
TO KILL A MOCKINGBIRD
15-FEB-02 01-MAR-02
```

当查询 MY_CHECKOUT 时，将返回依赖于伪列 User(在查询该视图时的 SQL*Plus 的用户)的记录。

19.2.17　给公众授予访问权

我们不仅可以将访问权限授予每一个用户，而且还可以使用 grant 命令将访问权限授予公众(public)：

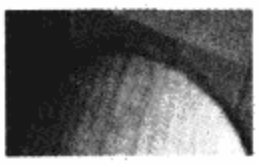
```
grant select on MY_CHECKOUT to public;
```

此命令将访问权限授予每个用户，包括在此 grant 操作之后创建的用户。但是，每个用户仍然必须用所有者的用户名作为前缀来访问这个表。为了避免这种情况，DBA 可以创建一个公用同义词(即创建一个代表 Practice.MY_CHECKOUT 的所有用户可访问的名字)：

```
create public synonym MY_CHECKOUT for Practice.MY_CHECKOUT;
```

从此以后，任何人都可以访问 MY_CHECKOUT，而不必用模式所有者(本例中为 Practice)作前缀了。这种方法提供了很大的安全灵活性。例如，员工在一个包含所有人工资的表中只能看到他们自己的工资。但是，如果某个用户创建了与公用同义词同名的一个表或视图，则这个用户以后发出的任何 SQL 语句都将作用于这个新表或视图，而不是作用于公众可以访问

的同名表或视图。



注意：
所有这些访问(以及所有系统级操作，如登录)都可以进行审计。关于 Oracle 的安全性审计功能的介绍，请参阅附录 A 中的 AUDIT 部分。

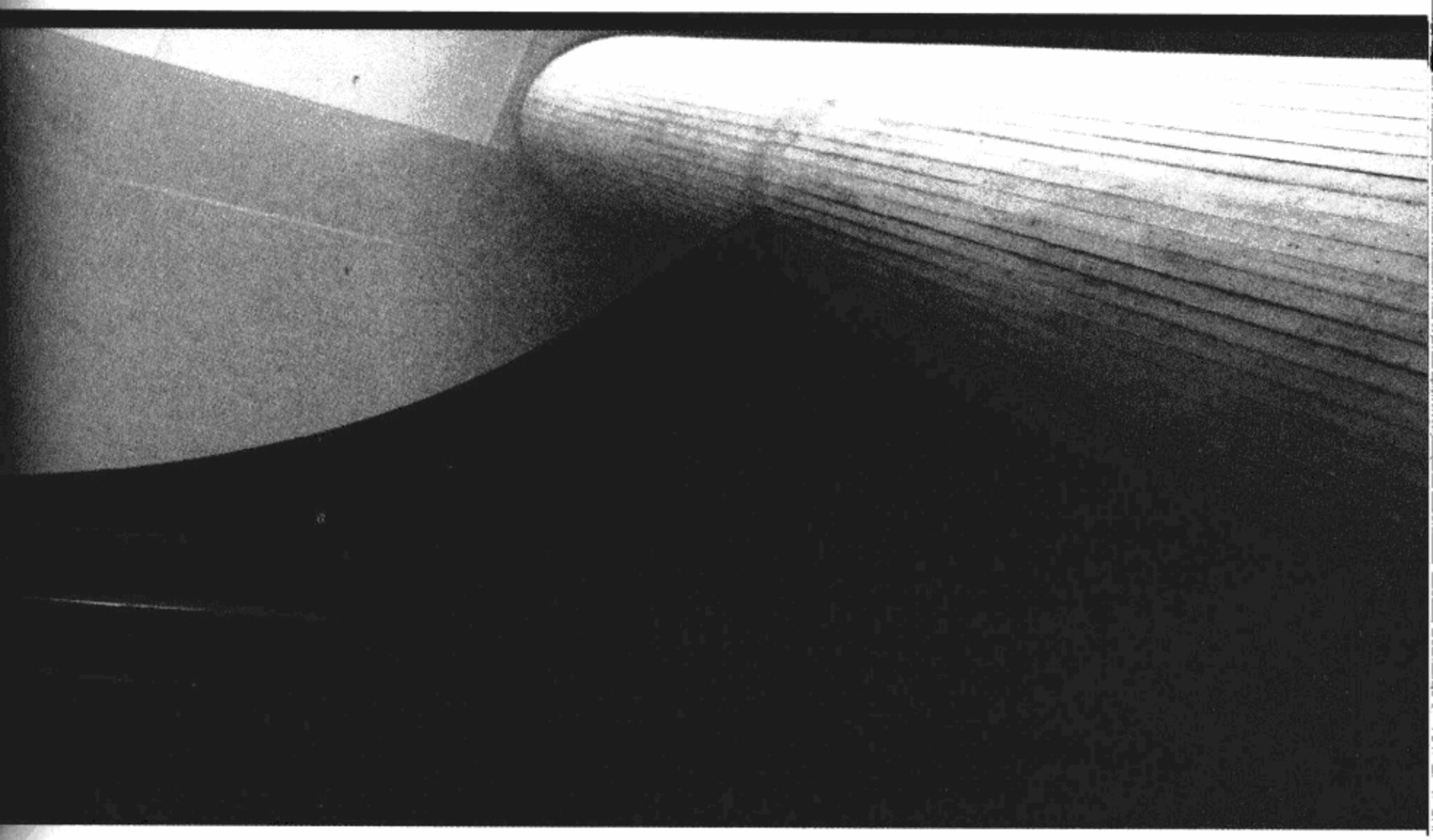
19.3 有限资源的授权

在 Oracle 数据库中分配资源限额时，可以使用 create user 或 alter user 命令中的 quota 参数，如下面的程序清单所示。在这个示例中，给 Bob 分配了 USERS 表空间中 100MB 的限额。

```
alter user Bob
    quota 100M on USERS;
```

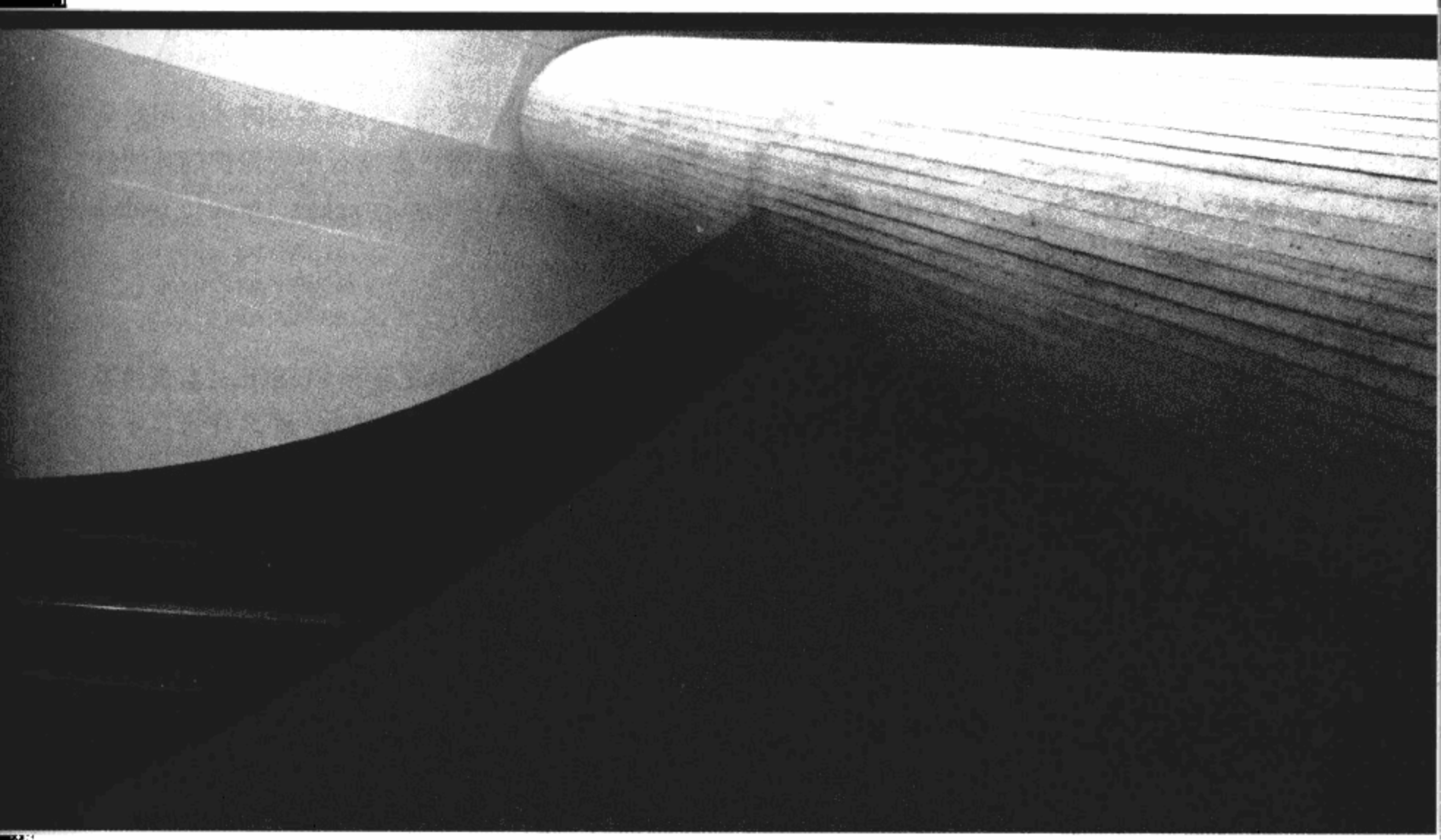
用户的空间限额可在通过 create user 命令创建用户时设置。如果想使撤消一个用户的空间限额，以便不受限制，那么可以将 UNLIMITED TABLESPACE 系统权限授予该用户。关于这些命令的详细内容，请查阅附录 A 中的 create user 和 alter user 命令。

您也可以使用配置文件来施加其他的资源限制，例如规定用户对 Oracle 的请求可占用的 CPU 时间或空闲时间量。可创建一个详细规定这些资源限额的配置文件，并将它分配给一个或多个用户。详细内容请参阅附录 A 中的 create profile 和 alter user 命令项。



第Ⅲ部分

高 级 主 题



第20章

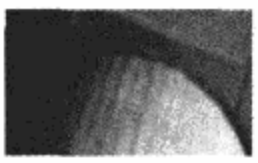
高级安全性——虚拟专用数据库

前面的章节介绍了如何使用视图来加强安全性，这些视图仅仅显示那些符合指定标准的数据行。本章将介绍如何使用虚拟专用数据库(Virtual Private Database，VPD)在您的应用表上提供记录级类型的安全性。在 VPD 中，您可以直接在表、视图和同义词上实施安全策略，这样用户就不可能忽略您的安全设置了。

在 VPD 中，任何受 VPD 策略保护的同时用于访问表、视图或者同义词的 SQL 都被动态地修改，并包含了一个限制条件——一个 where 子句或者一个 and 子句。修改过程是透明的，并且用户只能看到满足 where 子句限制条件的数据。这种细粒度的访问允许您对所有应用表的访问执行大量的控制操作。VPD 策略可以应用于 select、insert、update、index 和 delete 命

令。您还可以对不同类型的访问创建不同的安全策略——如为 select 访问创建一个策略，再为 insert 访问创建另外一个策略，等等。

VPD 包括列级别的 VPD，在这种数据库中，仅当访问指定的一列或多列时才应用安全策略。Oracle 还支持列屏蔽，这种特性可以使得在返回行数据时，数据行中受保护的列的值显示为 NULL。另外，因为审计功能也得以加强，所以 DBA 可以执行 select、insert、update 和 delete 命令的细粒度审计了。



注意：
实现 VPD 需要使用一些高级的特性——过程、触发器和 PL/SQL。本书的第Ⅳ部分将讲述这些特性。如果您对这些特性还不是很熟悉，那么建议在学习本章后续内容之前了解一下本书第Ⅳ部分的内容。

20.1 初始配置

VPD 允许您为不同的用户查看相同的表定制不同的方式。为了能顺利运行本章所示的所有示例，您需要一个拥有 SYSDBA 权限的账户。

VPD 要求使用一个名为 DBMS_RLS 的程序包。通过 SYSDBA 权限的账户，为所有的用户授予对 DBMS_RLS 程序包的 EXECUTE 权限。同时，创建将在示例中使用的用户：PRACTICE(将拥有这些表)、ADAMS、BURLINGTON，如下面的程序清单所示：

```
connect system/password as SYSDBA

grant execute on DBMS_RLS to public;

create user ADAMS identified by john7;
create user BURLINGTON identified by newj2;
grant CREATE SESSION to ADAMS, BURLINGTON;
```

通过 SYSDBA 权限的账户创建 PRACTICE 用户(如果这个用户还不存在)，并授予他 CREATE ANY CONTEXT 和 CREATE PUBLIC SYNONYM 系统权限：

```
grant CREATE ANY CONTEXT, CREATE PUBLIC SYNONYM to PRACTICE;
```

在 PRACTICE 用户中，您现在应创建两个表：STOCK_ACCOUNT 表和 STOCK_TRX 表。STOCK_ACCOUNT 表中针对每个可以进行股票交易的账户都有一行相应的数据。STOCK_TRX 表中针对此账户持有者每一笔买卖的股票都有一行相应的数据。

```
connect practice/practice

create table STOCK_ACCOUNT
(Account NUMBER(10),
AccountLongName VARCHAR2(50));

insert into STOCK_ACCOUNT values (
```

```
1234, 'ADAMS');
insert into STOCK_ACCOUNT values (
7777, 'BURLINGTON');

create table STOCK_TRX (
Account NUMBER(10),
Symbol VARCHAR2(20),
Price NUMBER(6,2),
Quantity NUMBER(6),
Trx_Flag VARCHAR2(1));

insert into STOCK_TRX values (
  1234, 'ADSP', 31.75, 100, 'B');
insert into STOCK_TRX values (
  7777, 'ADSP', 31.50, 300, 'S');
insert into STOCK_TRX values (
  1234, 'ADSP', 31.55, 100, 'B');
insert into STOCK_TRX values (
  7777, 'OCKS', 21.75, 1000, 'B');
commit;
```

一旦创建了这些表之后，就授予 ADAMS 用户和 BURLINGTON 用户访问表的权限：

```
grant SELECT, INSERT on STOCK_TRX to ADAMS, BURLINGTON;
grant SELECT on STOCK_ACCOUNT to ADAMS, BURLINGTON;
```

您现在已经将应用配置设置好了——PRACTICE 模式拥有数据表(STOCK_TRX)，并有一个您可以用来将用户名映射到账号的表(STOCK_ACCOUNT)。在完成以上配置之后，您可以创建一个应用程序上下文，相关内容将在 20.2 节介绍。

20.2　创建应用程序上下文

VPD 配置的下一步是创建应用程序上下文。上下文的作用是定义用来区分用户的规则。上下文将是每一个会话特征的一部分。

首先，使用 create context 命令指定将用于设置规则的程序包的名称：

```
connect PRACTICE/PRACTICE

create context PRACTICE using PRACTICE.CONTEXT_PACKAGE;
```

接下来，创建同名的程序包(CONTEXT_PACKAGE)。关于程序包用法的详细内容，请参阅第 35 章。这个程序包只有一个单独的过程调用：

```
create or replace package CONTEXT_PACKAGE AS
  procedure SET_CONTEXT;
end;
/
```

请注意程序包的头部已经创建，将要创建的是程序包的主体。以下程序清单显示了程序

包的主体，其将为每一个会话设置上下文。

```
create or replace package body CONTEXT_PACKAGE is
  procedure Set_Context IS
    v_user VARCHAR2(30);
    v_id NUMBER;
  begin
     DBMS_SESSION.SET_CONTEXT('PRACTICE','SETUP','TRUE');
     v_user := SYS_CONTEXT('USERENV','SESSION_USER');
     begin
       select Account
         into v_id
         from STOCK_ACCOUNT
       where AccountLongName = v_user;
     DBMS_SESSION.SET_CONTEXT('PRACTICE','USER_ID', v_id);
   exception
     WHEN NO_DATA_FOUND then
       DBMS_SESSION.SET_CONTEXT('PRACTICE','USER_ID', 0);
   end;
   DBMS_SESSION.SET_CONTEXT('PRACTICE','SETUP','FALSE');
 end SET_CONTEXT;
end CONTEXT_PACKAGE;
/
```

这个程序包主体的核心是一个单独的查询：

```
select Account
    into v_id
    from STOCK_ACCOUNT
  where AccountLongName = v_user;
```

STOCK_ACCOUNT 表将针对每个用户(Adams、Burlington 等)包含相应的一行数据。这个查询获得了已经登录的用户名称，并且将其与 STOCK_ACCOUNT 表中的 AccountLongName 值进行比较。如果两者匹配，那么返回 Account 值。如果两者不匹配，那么该程序包的另外一部分将返回 0 来表示账号。Account 值将通过会话的 User_ID 变量来设置：

```
DBMS_SESSION.SET_CONTEXT("PRACTICE" ,"USER_ID",v_id);
```

您将可以在程序中引用这个上下文的值，而且 VPD 约束也将基于此上下文设置。

由于程序包已经创建了，因此为其授予公共访问权：

```
grant EXECUTE on PRACTICE.CONTEXT_PACKAGE to public;
create public synonym CONTEXT_PACKAGE for PRACTICE.CONTEXT_PACKAGE;
```

为会话创建上下文的规则现在将被激活，如 20.3 节所介绍。

20.3　创建登录触发器

为了在每个用户的会话内设置上下文，您必须创建触发器，在用户每次登录到数据库时此触发器将被执行(关于触发器的详细内容，请参阅第 34 章)。

```
rem NOTE: connect to a SYSDBA account before running this
rem
create or replace trigger PRACTICE.SET_SECURITY_CONTEXT
after logon on database
begin
  PRACTICE.CONTEXT_PACKAGE.SET_CONTEXT;
end;
/
```

每次用户登录的时候，Oracle 都将在 PRACTICE 用户的 CONTEXT_PACKAGE 程序包内执行 SET_CONTEXT 过程。这个过程将查询 STOCK_ACCOUNT 表，并判断是否有匹配的账户名称；如果存在匹配的账户名称，那么将会修改话的上下文，以反映 Account 值。

在 VPD 的实现中，当您进行到这一步骤之前，应当测试上下文设置。您可以使用 SYS_CONTEXT 函数来判断配置是否运行正常，如下面的程序清单所示：

```
connect adams/john7
select SYS_CONTEXT('USERENV', 'SESSION_USER') Username,
       SYS_CONTEXT('PRACTICE', 'USER_ID') ID
from DUAL;

USERNAME
------------------------------------------------------------
ID
------------------------------------------------------------
ADAMS
1234
```

SYS_CONTEXT 函数的输出显示了 VPD 配置的强大作用。对每一个登录的用户，Oracle 都将检查 STOCK_ACCOUNT 表，并使用匹配的 Account 值来设置会话的 User_ID 值。在上面的示例中，Oracle 在 STOCK_ACCOUNT 表中查找 ADAMS 用户，找到了其 Account 值(1234)，并将其设置为会话的 User_ID 变量的值。如果您知道了这个信息，就可以根据 STOCK_ACCOUNT 表中的 Account 值来限制对其他表中数据行的访问。VPD 允许自动地限制这样的访问，不需要创建视图。每次用户试图访问一个表时，无论其使用的工具如何，访问都会被限制。

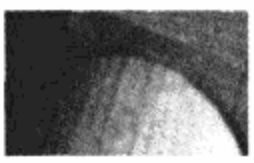
如果以用户身份登录，而这位用户没有在 STOCK_ACCOUNT 表中列出(触发了 NO _DATA_FOUND 异常)，那么您的 User_ID 值将为 0，如下面的过程所示：

```
  WHEN NO_DATA_FOUND then
    DBMS_SESSION.SET_CONTEXT('PRACTICE','USER_ID', 0);
```

(续表)

参　数	说　明
SCHEMAS	命名 Schema 模式将要导出的模式
STATUS	显示 Data Pump 作业的详细状态
TABLES	列出 Table 模式将要导出的表和分区
TABLESPACES	列出将要导出的表空间
TRANSPORT_FULL_CHECK	指定是否应该先验证导出的表空间是独立集
TRANSPORT_TABLESPACES	指定使用 Transportable Tablespace 模式导出
TRANSPORTABLE	指定在表模式导出(用 TABLES 参数指定)期间是否应该使用可移动选项为特定表、分区和子分区导出元数据。默认值是 NEVER。ALWAYS 值告诉 Data Pump 使用可移动选项
VERSION	指定将要创建的数据库对象的版本，这样转储文件集就能够与 Oracle 早期发布的版本兼容。选项为 COMPATIBLE、LATEST 和数据库版本号(不低于 10.0.0)

正如表 24-1 所示，支持 Data Pump 导出的 5 种模式。Full(完全)导出抽取所有数据库的数据和元数据。Schema(模式)导出抽取指定用户模式的数据和元数据。Tablespace(表空间)导出抽取表空间的数据和元数据，而 Table(表)导出抽取表及其分区的数据和元数据。Transportable Tablespace(可移动表空间)导出抽取特定表空间的元数据。



注意：

必须具有 EXP_FULL_DATABASE 系统权限，才能执行完全导出或可移动表空间导出。

当您提交作业时，Oracle 将给该作业指定一个系统生成的名称。如果通过 JOB_NAME 参数自己为作业指定一个名字，就必须确定这个作业名不会与模式中的任何表或视图的名称发生冲突。在 Data Pump 作业期间，Oracle 将创建并维护一个主表。主表将具有与 Data Pump 作业相同的名称，这样主表的名称就不会与现有的对象发生冲突了。

当作业运行时，您可以通过 Data Pump 的界面执行表 24-2 列出的命令。

表 24-2　交互模式 Data Pump Export 的参数

参　数	说　明
ADD_FILE	添加转储文件
CONTINUE_CLIENT	退出交互模式并进入日志模式
EXIT_CLIENT	退出客户端会话，但保留服务器端 Data Pump Export 作业的运行
FILESIZE	重新为随后的转储文件定义默认大小
HELP	显示导入的在线帮助

(续表)

参　数	说　明
KILL_JOB	关闭当前作业并分离相关的客户端会话
PARALLEL	更改参与 Data Pump Export 作业的人数
START_JOB	重新启动附加的作业
STATUS	显示 Data Pump 作业的详细状态
STOP_JOB	停止作业以便稍后重启

如表 24-2 所提示，可以通过交互式命令模式更改运行的 Data Pump Export 作业的很多特性。如果转储区域的空间用完了，那么可以附加作业、添加文件，还可以及时地重启作业；并且这样做还无需关闭作业或者从头开始重新执行作业。您既可以通过 STATUS 参数也可以通过 USER_DATAPUMP_JOBS 和 DBA_DATAPUMP_JOBS 数据字典视图或者 V$SESSION_LONGOPS 视图在任何时候显示作业的状态。

24.3　启动 Data Pump Export 作业

可以在参数文件中存储作业参数，这些参数可以通过 expdp 的 PARFILE 参数引用。例如，可以创建包含如下条目的 dp1.par 文件：

```
DIRECTORY=dtpump
DUMPFILE=metadataonly.dmp
CONTENT=METADATA_ONLY
```

然后启动 Data Pump Export 作业：

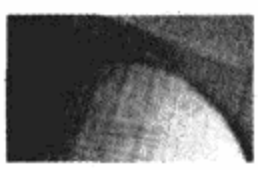
```
expdp practice/practice PARFILE=dp1.par
```

Oracle 将会把 dp1.par 项传递给 Data Pump Export 作业。这时将执行模式类型(默认类型)的 Data Pump Export，并且将输出(元数据清单，而不是数据)写入到前面定义的 dtpump 目录中的一个文件中。当您执行 expdp 命令时，输出将以如下所示的格式(为每个主要的对象类型——表、授权、索引等——分行显示)显示：

```
Starting "PRACTICE"."SYS_EXPORT_SCHEMA_01": practice/******** parfile=dp1.par
Processing object type SCHEMA_EXPORT/SE_PRE_SCHEMA_PROCOBJACT/PROCACT_SCHEMA
Processing object type SCHEMA_EXPORT/TYPE/TYPE_SPEC
Processing object type SCHEMA_EXPORT/TABLE/TABLE
Processing object type SCHEMA_EXPORT/TABLE/GRANT/OBJECT_GRANT
Processing object type SCHEMA_EXPORT/TABLE/INDEX/INDEX
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
Processing object type SCHEMA_EXPORT/TABLE/COMMENT
Processing object type SCHEMA_EXPORT/VIEW/VIEW
Processing object type SCHEMA_EXPORT/PACKAGE/PACKAGE_SPEC
```

```
Processing object type SCHEMA_EXPORT/PACKAGE/PACKAGE_BODY
Processing object type SCHEMA_EXPORT/PACKAGE/GRANT/OBJECT_GRANT
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/REF_CONSTRAINT
Processing object type SCHEMA_EXPORT/SE_EV_TRIGGER/TRIGGER
Master table "PRACTICE"."SYS_EXPORT_SCHEMA_01" successfully loaded/unloaded
******************************************************************************
Dump file set for PRACTICE.SYS_EXPORT_SCHEMA_01 is:
  E:\DTPUMP\METADATAONLY.DMP
Job "PRACTICE"."SYS_EXPORT_SCHEMA_01" successfully completed at 17:30
```

如程序清单所示，输出文件被命名为 metadataonly.dmp。该输出转储文件包含了为 Practice 模式重新创建结构的 XML 条目。在导出期间，Data Pump 创建并使用了一个名为 SYS_EXPORT_SCHEMA_01 的外部表。



注意：
转储文件不会覆盖同一目录下以前已经存在的转储文件。

可以为一个 Data Pump Export 使用多个目录和转储文件。在 DUMPFILE 参数设置中，列出目录和文件名，格式如下所示：

```
DUMPFILE=directory1:file1.dmp,
         directory2:file2.dmp
```

24.3.1 停止和重新启动运行的作业

在启动了 Data Pump Export 作业之后，可以关闭用来启动作业的客户端窗口。因为作业是基于服务器的，所以导出操作将继续运行。然后就可以连接到作业、检查作业状态并更改它。例如，可以通过 expdp 启动作业：

```
expdp practice/practice PARFILE=dp1.par
```

按下 CTRL+C 组合键可以关闭日志显示，并且将从 Data Pump 返回到 Export 提示：

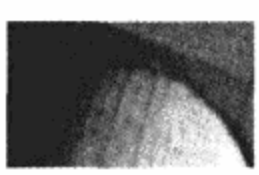
```
Export>
```

可通过 EXIT_CLIENT 命令退出操作系统：

```
Export> EXIT_CLIENT
```

然后就可以重启客户端，并连接到当前正在模式下运行的作业：

```
expdp practice/practice attach
```



注意：
这个示例导出很快就会完成，于是在您重新连接到作业之前，作业可能已经完成了。

如果您指定了 Data Pump Export 作业的名称，那么应当将该名称指定为 ATTACH 参数调用的一部分。例如，如果已经将作业命名为 PRACTICE_JOB，那么通过名称连接到该作业：

```
expdp practice/practice attach=PRACTICE_JOB
```

当连接到一个正在运行的作业时，Data Pump 将显示该作业的状态——它的基本配置参数和当前的状态。然后您就可以使用 CONTINUE_CLIENT 命令来查看生成的日志项，或者可以更改正在运行的作业：

```
Export> CONTINUE_CLIENT
```

可以通过 STOP_JOB 选项停止一项作业：

```
Export> STOP_JOB
```

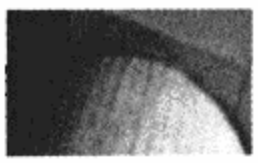
在作业停止时，就可以通过 ADD_FILE 选项在新的目录中添加其他转储文件。然后您可以重新启动作业：

```
Export> START_JOB
```

您可以通过 LOGFILE 参数为导出日志指定一个日志文件位置。如果没有指定 LOGFILE 值，那么日志文件将写入与转储文件相同的目录中。

24.3.2　从另一个数据库中导出

可以使用 NETWORK_LINK 参数来从另一个数据库中导出数据。如果登录到 HQ 数据库并且您有到另一个数据库的数据库链接，那么 Data Pump 可以使用这个链接来连接数据库并提取数据库中的数据。



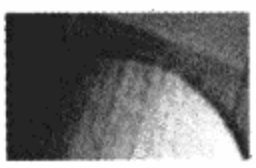
注意：
如果源数据库是只读的，那么源数据库上的用户必须拥有一个用作临时表空间的本地管理的表空间；否则，作业将失败。

在参数文件中(或者在 expdp 命令行上)，设置 NETWORK_LINK 参数等于数据库链接的名称。Data Pump Export 将会把数据从远程数据库写入本地数据库定义的目录中。

24.3.3　使用 EXCLUDE、INCLUDE 和 QUERY

可以通过 EXCLUDE 和 INCLUDE 选项从 Data Pump Export 中排除或包括表集合。您可以按类型或按名称排除对象。如果对象被排除，那么依赖该对象的所有对象也会被排除。EXCLUDE 选项的格式如下：

```
EXCLUDE=object_type [: name_clause] [, …]
```



注意：
如果您指定 CONTENT=DATA_ONLY，就不能指定 EXCLUDE。

例如，为了从完全导出中排除 PRACTICE 模式，EXCLUDE 选项的格式如下：

```
EXCLUDE=SCHEMA:"='PRACTICE'"
```

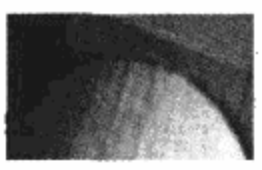
注意：
可以在同一个 Data Pump Export 作业中指定多个 EXCLUDE 选项。

前面程序清单中的 EXCLUDE 选项包含了一个限制条件(='PRACTICE')，此条件用双引号引起来。object_type 变量可以是任何 Oracle 对象类型，包括授权、索引和表。name_clause 变量用来限制返回的值。例如，当需要从导出中排除所有名称以'TEMP'开头的表时，就可以使用如下语句：

```
EXCLUDE=TABLE:"LIKE "TEMP%" "
```

当在命令行输入以上语句时，可能需要使用转义字符，以便将引号和其他特殊的符号正确地传递给 Oracle。expdp 命令格式如下：

```
expdp practice/practice EXCLUDE=TABLE:\ "LIKE\'TEMP%\'\"
```



注意：
以上这个示例只展示了一部分语法，而不是该命令的所有语法。

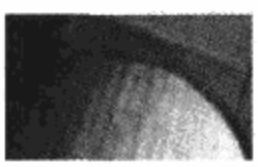
如果没有提供 name_clause 值，那么指定类型的所有对象都将被排除。例如，要排除所有的索引，可以使用如下语句：

```
expdp practice/practice EXCLUDE=INDEX
```

为了得到您可以筛选的对象列表，可以查询 DATABASE_EXPORT_OBJECTS、SCHEMA_EXPORT_OBJECTS 和 TABLE_EXPORT_OBJECTS 数据字典视图。如果 object_type 值是 CONSTRAINT，那么 NOT NULL 约束将不会被排除。而且，那些需要用来成功创建表的约束(如索引组织表的主键约束)也不能排除。如果 object_type 值是 USER，那么用户定义将被排除，但是在用户模式内的对象还将被导出。如前面的示例所示，可以使用 SCHEMA object_type 来排除用户和该用户的所有对象。如果 object_type 值是 GRANT，那么所有的对象授权和系统权限授权将被排除。

第二个选项 INCLUDE 也可用。当使用 INCLUDE 时，只有那些满足条件的对象才被导出，其余的所有对象都将排除。INCLUDE 和 EXCLUDE 是互相排斥的。INCLUDE 的格式如下：

```
INCLUDE = object_type [: name_clause] [, …]
```



注意：
如果您指定了 CONTEXT=DATA_ONLY，就不能指定 INCLUDE。

例如，要导出两个表和所有过程，参数文件可能包含如下两行：

```
INCLUDE=TABLE:"IN ('BOOKSHELF','BOOKSHELF_AUTHOR')"
INCLUDE=PROCEDURE
```

对于满足 EXCLUDE 或 INCLUDE 条件的对象，将会导出哪些行呢？在默认情况下，会导出每个表的所有行。您可以使用 QUERY 选项来限制返回的行数。QUERY 选项的格式如下：

```
QUERY = [schema.] [table_name:] query_clause
```

如果没有指定 schema 变量和 table_name 变量的值，那么 query<->_clause 将应用于所有导出的表。因为 query<->_clause 通常包含特定的列名称，所以在选择导出所包含的表时，应当非常地小心。

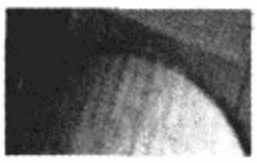
您可以为一个表指定 QUERY 值，如下面的程序清单所示：

```
QUERY=BOOKSHELF:'"WHERE Rating > 2"'
```

结果是，转储文件将只包含满足 QUERY 条件以及 INCLUDE 或 EXCLUDE 条件的行。您还可以在随后的 Data Pump Import 中应用这些限制，24.4 节将描述这些内容。

24.4　Data Pump Import 选项

如果要导入一个由 Data Pump Export 导出的转储文件，那么可以使用 Data Pump Import。与导出过程一样，导入过程作为一个基于服务器的作业来运行，这样就可以在它执行时进行管理。可以通过命令行接口、参数文件和交互式接口与 Data Pump Import 交互。表 24-3 列出了命令行接口使用的参数。



注意：
转储文件和日志文件的目录必须已经存在；请参阅 24.1 节。

与 Data Pump Export 一样，Data Pump Import 支持 5 种模式：Full(完全)、Schema(模式)、Table(表)、Tablespace(表空间)和 Transportable Tablespace(可移动表空间)。如果没有指定任何模式，那么 Oracle 将试图加载整个转储文件。

表 24-3　Data Pump Import 的命令行参数

参　数	说　明
ATTACH	将客户端连接至服务器会话并置于交互模式
CONTENT	过滤导入的数据：ALL、DATA_ONLY 或者 METADATA_ONLY
DATA_OPTIONS	提供了这样的选项，即导出和导入期间如何处理某些类型的数据。对于导入操作，DATA_OPTIONS 参数唯一有效的选项是 SKIP_CONSTRAINT_ERRORS
DIRECTORY	指定转储文件集的位置，并为日志文件和 SQL 文件指定目标目录
DUMPFILE	为转储文件集指定名称，还可以根据需要为转储文件集指定目录

(续表)

参　数	说　明
ENCRYPTION_PASSWORD	指定访问转储文件集中的加密列数据所需要的密码
ESTIMATE	决定用来评估转储文件大小(BLOCKS 或者 STATISTICS)的方法
EXCLUDE	导入时排除对象和数据
FLASHBACK_SCN	在导入期间闪回的数据库的 SCN(请参阅第 27 章)
FLASHBACK_TIME	在导入期间闪回的数据库的时间戳(请参阅第 27 章)
FULL	Y/N 标记，用于指明您是否想要导入全部转储文件
HELP	显示导入的联机帮助
INCLUDE	为将要导入的对象指定条件
JOB_NAME	为作业指定名称；由系统生成默认名称
LOGFILE	导入日志的名称和可选的目录名称
NETWORK_LINK	为导入一个远程数据库的 Data Pump 作业指定源数据库链接
NOLOGFILE	Y/N 标记，用于抑制日志文件的创建
PARALLEL	为 Data Pump Import 作业设置参与工作的人数
PARFILE	如果有要使用的参数文件，则命名参数文件
PARTITION_OPTIONS	指定在导入操作期间应该如何创建表分区。DEPARTITION 值将每个分区或子分区提升为一个新表。MERGE 值将所有分区和子分区合并成为一个新表
QUERY	在导入期间从表中过滤数据行
REMAP_DATA	允许在将数据插入到新数据库中时重新映射数据。常见的用法是，当将一个表导入到目标数据库中预先存在的一个表时，重新生成主键以避免冲突。相同的函数可以应用于被转储的多列。如果您想在重新映射引用约束的子列和父列时保证一致，这一点就很有用
REMAP_DATAFILE	在导入期间用 create library、create tablespace 和 create directory 命令将源数据文件名称更改为目标数据文件
REMAP_SCHEMA	将从源模式导出的数据导入到目标模式
REMAP_TABLESPACE	将从源表空间导出的数据导入到目标表空间
REUSE_DATAFILES	指定在 Full 模式导入时 create tablespace 命令是否应该重用已经存在的数据文件
SCHEMAS	为 Schema 模式将要导入的模式命名
SKIP_UNUSABLE_IINDEXES	Y/N 标记。如果设置为 Y，那么导入不会将数据加载到其索引设置为 Index Unusable 状态的表
SQLFILE	命名将要写入 DDL 的文件。数据和元数据不会加载到目标数据库中
STATUS	显示 Data Pump 作业的详细状态
STREAMS_CONFIGURATION	用于指定 Streams 配置信息是否应当导入的 Y/N 标记

(续表)

参　数	说　明
TABLE_EXISTS_ACTION	如果将要导入的表已经存在，则指示 Import 如何处理。值包括 SKIP、APPEND、TRUNCATE 和 REPLACE。如果 CONTENT=DATA_ ONLY，则默认值为 APPEND；否则默认值为 SKIP
TABLES	列出以 Table 模式导入的表
TABLESPACES	列出以 Tablespace 模式导入的表
TRANSFORM	指示导入期间段属性或存储的变化
TRANSPORT_DATAFILES	列出以 Transportable Tablespace 模式导入的数据文件
TRANSPORT_FULL_CHECK	指定是否应该首先验证导入的表空间为独立集
TRANSPORT_TABLESPACES	列出以 Transportable Tablespace 模式导入的表空间
TRANSPORTABLE	指定在执行表模式导入(用 TABLES 参数指定)时是否应该使用可移动选项
VERSION	指定将要创建的数据库对象的版本，这样转储文件集就能够与 Oracle 早期发布的版本兼容。选项为 COMPATIBLE、LATEST 和数据库版本号(不低于 10.0.0)。只对 NETWORK_LINK 和 SQLFILE 有效

表 24-4 列出了 Data Pump Import 的交互模式中有效的参数。

表 24-4　Data Pump Import 的交互参数

参　数	说　明
CONTINUE_CLIENT	退出交互模式并且进入登录模式。如果作业闲置，则将重新启动
EXIT_CLIENT	退出客户端会话，但是保持服务器端 Data Pump Import 作业的运行
HELP	显示导入的联机帮助
KILL_JOB	关闭当前的作业并分离相关的客户端会话
PARALLEL	更改参与 Data Pump Import 作业的人数
START_JOB	重新启动附加的作业
STATUS	显示 Data Pump 作业的详细状态
STOP_JOB	停止作业，稍以后重启

很多 Data Pump Import 的参数与 Data Pump Export 的参数类似。下面几节将介绍如何启动导入作业，并且将会描述 Data Pump Import 特有的主要选项。

24.5　启动 Data Pump Import 作业

在 Oracle Datebase 11g 中，可以通过 impdp 可执行程序来启动 Data Pump Import 作业。可以使用命令行参数来指定导入模式和所有文件的位置。可以在参数文件中存储参数值，然

后通过 PARFILE 选项引用参数文件。

在本章的第一个导出示例中，文件名为 dp1.par 的参数文件包含了如下项：

```
DIRECTORY=dtpump
DUMPFILE=metadataonly.dmp
CONTENT=METADATA_ONLY
```

该导入将以不同的模式创建 PRACTICE 模式的对象。REMAP_SCHEMA 选项允许您将对象导入到与导出不同的模式。如果您还想同时更改对象的表空间分配，则可以使用 REMAP_TABLESPACE 选项。REMAP_SCHEMA 的格式如下：

```
REMAP_SCHEMA=source_schema : target_schema
```

创建新的用户账户来保存对象：

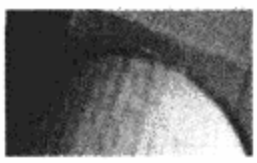
```
create user Newpractice identified by newp;
grant CREATE SESSION to Newpractice;
grant CONNECT, RESOURCE to Newpractice;
grant CREATE TABLE to Newpractice;
grant CREATE INDEX to Newpractice;
```

现在就可以将 REMAP_SCHEMA 行添加到 dp1.par 参数文件：

```
DIRECTORY=dtpump
DUMPFILE=metadataonly.dmp
CONTENT=METADATA_ONLY
REMAP_SCHEMA=Practice:Newpractice
```

现在可以启动导入作业。因为正在更改数据的拥有模式，所以必须具有 IMP_FULL_DATABASE 系统权限。Data Pump Import 作业可以通过 impdp 可执行命令来启动。以下程序清单显示了使用 dp1.par 参数文件创建 Data Pump Import 作业的示例：

```
impdp system/passwd parfile=dp1.par
```



注意：

所有的转储文件必须在作业启动时指定。

随后，Oracle 将执行导入并显示其进程。因为没有指定 NOLOGFILE 选项，所以导入的日志文件将存储与转储文件相同的目录中，并且将被命名为 import.log。可以登录到 NEWPRACTICE 模式来验证导入的成功。NEWPRACTICE 模式应当有所有有效对象的一个副本，这些对象是以前在 PRACTICE 模式中创建的。

如果导入的表已经存在，结果会怎么样呢？在这个示例中，因为 CONTENT 选项被设置为 METADATA_ONLY，所以默认情况下表将会被忽略。如果 CONTENT 选项设置为 DATA_ONLY，那么新的数据将会被追加到现有的表数据之后。为了改变这种行为，可以使用 TABLE_EXISTS_ACTION 选项。TABLE_EXISTS_OPTION 的有效值包括 SKIP、APPEND、TRUNCATE 和 REPLACE。

24.5.1　停止和重新启动运行的作业

在启动了 Data Pump Import 作业后，可以关闭先前用来启动作业的客户端窗口。因为导入操作是基于服务器的，所以导入操作会继续运行。然后就可以连接到该作业，查看作业的状态并更改它：

```
impdp system/passwd PARFILE=dp1.par
```

按下 CTRL＋C 组合键退出日志显示，然后 Data Pump 将返回 Import 提示：

```
Import>
```

可以通过 EXIT_CLIENT 命令退出操作系统：

```
Import> EXIT_CLIENT
```

随后可以重新启动客户端，并连接到当前正在模式下运行的作业：

```
impdp system/passwd attach
```

如果对 Data Pump Import 作业进行了命名，那么可以指定这个名称作为 ATTACH 参数调用的一部分。当您连接到一个正在运行的作业时，Data Pump 将显示作业的状态——作业的基本配置参数和当前状态。然后还可以使用 CONTINUE_CLIENT 命令来查看生成的日志项，或者可以更改正在运行的作业：

```
Import> CONTINUE_CLIENT
```

可以通过 STOP_JOB 选项停止作业：

```
Import> STOP_JOB
```

当作业停止时，可以通过 PARALLEL 选项来提高并行性，然后就可以重新启动该作业了：

```
Import> START_JOB
```

24.5.2　EXCLUDE、INCLUDE 和 QUERY

和 Data Pump Export 一样，Data Pump Import 允许使用 EXCLUDE、INCLUDE 和 QUERY 选项来限制处理的数据，本章前面已有描述过。因为在导出和导入中都可以使用这些选项，所以就可以很灵活地处理导入操作了。例如，您也许会选择导出整个表，而只导入表的一部分——导入那些匹配 QUERY 条件的行。可以选择导出整个模式而通过只导入最需要的表来恢复数据库，这样应用程序的停机时间可能会最少。在导出和导入作业的过程中，EXCLUDE、INCLUDE 和 QUERY 选项为开发人员和数据库管理员提供了强大的功能。

24.5.3　转换导入的对象

除了在导入期间可以更改或选择模式、表空间、数据文件和行之外，您还可以通过 TRANSFORM 选项来更改段属性和存储要求。TRANSFORM 格式如下：

```
TRANSFORM = transform_name : value [: object_type]
```

变量 transform_name 的值可以是 SEGMENT_ATTRIBUTES 或者 STORAGE。可以使用 value 变量来包含或者排除段属性(物理属性、存储属性、表空间和登录)。object_type 变量是可选的，如果需要使用它，则它的值可以是 TABLE 或者 INDEX。

例如，对象存储要求在导入/导出期间可能需要更改——您可能会使用 QUERY 选项来限制导入的行，或者可能只导入元数据而不导入表数据。如果需要从导入的表中排除导出的存储子句，那么可以在参数文件中添加如下语句：

```
TRANSFORM = STORAGE : n : table
```

为了从所有的表和索引中排除导出的表空间和存储子句，可以使用如下语句：

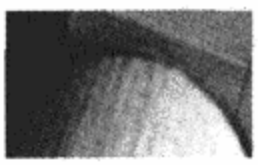
```
TRANSFORM = SEGMENT_ATTRIBUTES : n
```

当导入对象时，这些对象将被分配给用户的默认表空间，并且将使用这个表空间的默认存储参数。

24.5.4 生成 SQL

除了导入数据和对象，也可以为对象(不是数据)生成 SQL，并将其存储在操作系统的一个文件中。该文件将通过 SQLFILE 选项写入指定的目录和文件。SQLFILE 选项的格式如下：

```
SQLFILE = [directory_object : ] file_name
```



注意：
如果没有为 directory_name 变量指定值，那么将在转储文件目录中创建文件。

以下程序清单显示了使用 SQLFILE 选项导入的样本参数文件。请注意这里没有指定 CONTENT 选项。输出将被写入 dtpump 目录。

```
DIRECTORY=dtpump
DUMPFILE=metadataonly.dmp
SQLFILE=sql.txt
```

然后就可以运行导入来填充 sql.txt 文件：

```
impdp practice/practice parfile=dp1.par
```

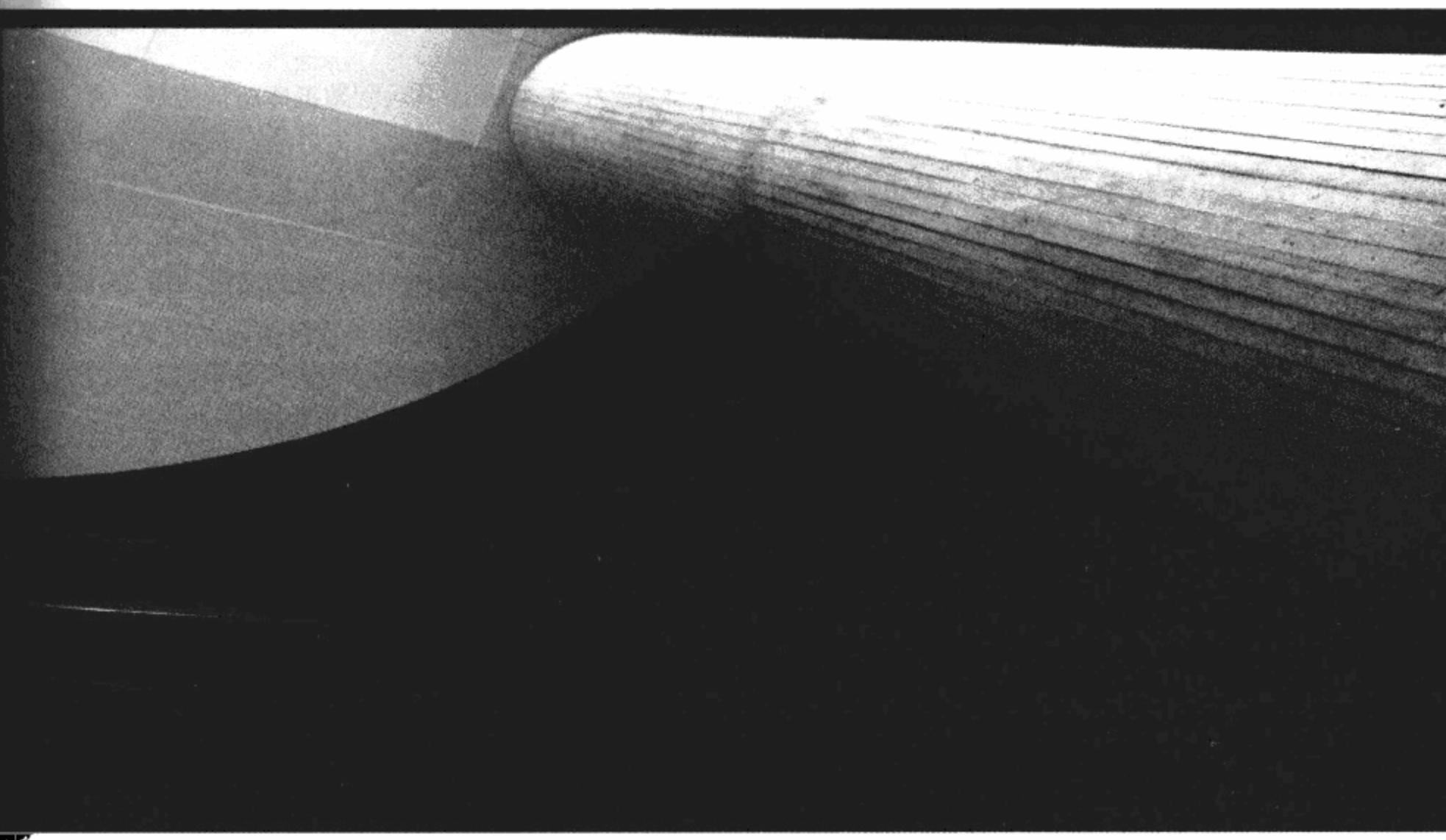
在该导入创建的 sql.txt 文件中，您将看到该模式中每一种对象类型的项。输出文件的格式将与以下的程序清单一样，只是对象的 ID 和 SCN 将因环境的不同而不同。简单来说，不是文件中的所有项都显示于此。

```
-- CONNECT PRACTICE
-- new object type path is: SCHEMA_EXPORT/
SE_PRE_SCHEMA_PROCOBJACT/PROCACT_SCHEMA
```

```
BEGIN
sys.dbms_logrep_imp.instantiate_schema(schema_name=>'PRACTICE',
export_db_name=>'ORCL', inst_scn=>'3377908');
COMMIT;
END;
/

-- new object type path is: SCHEMA_EXPORT/TYPE/TYPE_SPEC
CREATE TYPE "PRACTICE"."ADDRESS_TY"
  OID '48D49FA5EB6D447C8D4C1417D849D63A' as object
(Street VARCHAR2(50),
  City   VARCHAR2(25),
  State  CHAR(2),
  Zip    NUMBER);
/

CREATE TYPE "PRACTICE"."CUSTOMER_TY"
  OID '8C429A2DD41042228170643EF24BE75A' as object
(Customer_ID NUMBER,
  Name      VARCHAR2(25),
  Street    VARCHAR2(50),
  City      VARCHAR2(25),
  State     CHAR(2),
  Zip       NUMBER);
/

CREATE TYPE "PRACTICE"."PERSON_TY"
  OID '76270312D764478FAFDD47BF4533A5F8' as object
(Name VARCHAR2(25),
 Address ADDRESS_TY);
/

-- new object type path is: SCHEMA_EXPORT/TABLE/TABLE
CREATE TABLE "PRACTICE"."CUSTOMER"
    ( "CUSTOMER_ID" NUMBER,
      "NAME" VARCHAR2(25),
      "STREET" VARCHAR2(50),
      "CITY" VARCHAR2(25),
      "STATE" CHAR(2),
      "ZIP" NUMBER
    ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

CREATE TABLE "PRACTICE"."CUSTOMER_CALL"
    ( "CUSTOMER_ID" NUMBER,
      "CALL_NUMBER" NUMBER,
      "CALL_DATE" DATE
```

```
  ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

CREATE TABLE "PRACTICE"."STOCK_TRX"
    ( "ACCOUNT" NUMBER(10,0),
      "SYMBOL" VARCHAR2(20),
      "PRICE" NUMBER(6,2),
      "QUANTITY" NUMBER(6,0),
      "TRX_FLAG" VARCHAR2(1)
  ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

CREATE TABLE "PRACTICE"."STOCK_ACCOUNT"
    ( "ACCOUNT" NUMBER(10,0),
      "ACCOUNTLONGNAME" VARCHAR2(50)
  ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;

-- new object type path is: SCHEMA_EXPORT/TABLE/GRANT/OBJECT_GRANT
GRANT SELECT ON "PRACTICE"."STOCK_TRX" TO "ADAMS";
GRANT SELECT ON "PRACTICE"."STOCK_TRX" TO "BURLINGTON";
GRANT SELECT ON "PRACTICE"."STOCK_ACCOUNT" TO "ADAMS";
GRANT SELECT ON "PRACTICE"."STOCK_ACCOUNT" TO "BURLINGTON";
GRANT INSERT ON "PRACTICE"."STOCK_TRX" TO "ADAMS";

-- new object type path is: SCHEMA_EXPORT/TABLE/INDEX/INDEX
CREATE UNIQUE INDEX "PRACTICE"."CUSTOMER_PK" ON "PRACTICE"."CUSTOMER"
("CUSTOMER_ID")
  PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" PARALLEL 1 ;

  ALTER INDEX "PRACTICE"."CUSTOMER_PK" NOPARALLEL;
```

由于 SQLFILE 输出的是一个纯文本文件，因此可以编辑该文件，在 SQL*Plus 中使用它，或者将它保存为应用程序的数据库结构文档。



第 25 章

访问远程数据

当数据库在大小和数量上不断增长时，您可能会迫切地需要在数据库之间共享数据。数据共享要求使用一种查找并访问数据的方法。在 Oracle 中，使用数据库链接可以进行诸如查询和更新的远程数据访问。如本章将介绍的，数据库链接让用户将一组分布式数据库作为一个单独的、一体化的数据库来对待。本章还将介绍关于直接连接远程数据库的信息，如用于客户-服务器应用程序的信息。

25.1 数据库链接

数据库链接(Database Link)告诉 Oracle 如何从一个数据库到达另一个数据库。如果要频繁使用到远程数据库的同一个连接，那么数据库链接是最合适的方法。

25.1.1 数据库链接是如何工作的

数据库链接要求 Oracle Net(以前称为 SQL*Net 和 Net8)在涉及远程数据库访问的每一台计算机(主机)上运行。Oracle Net 通常由数据库管理员(DBA)或系统管理员启动。图 25-1 显示了使用数据库链接进行远程访问的一个样例体系结构。该图显示了两台主机，每台都运行 Oracle Net。每台主机上都有一个数据库。数据库链接定义了从第一个数据库(名为 LOCAL，在 Branch 主机上)到第二个数据库(名为 REMOTE，在 Headquarters 主机上)的连接。图 25-1 显示的数据库链接位于 LOCAL 数据库中。

为了支持两个数据库之间的通信，Oracle Net 配置必须包含数据库的侦听器和数据库的服务名。Oracle Net 配置文件包括 tnsnames.ora(用于把数据库服务名转化为数据库、主机和端口)和 listener.ora(用于指定数据库在本地主机上的连接信息)。

数据库链接指定以下连接信息：

- 在连接期间使用的通信协议(如 TCP/IP)
- 远程数据库所在的主机
- 远程主机上的数据库名
- 远程数据库中的有效账户名(可选)
- 该账户的密码(可选)

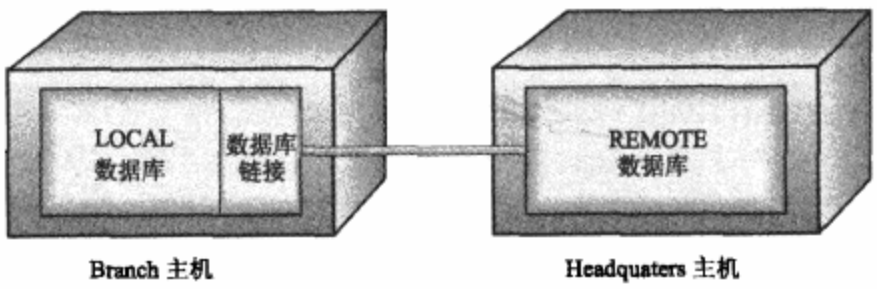


图 25-1 数据库链接的样例体系结构

在使用时，数据库链接实际作为远程数据库中的用户进行登录，且在会话期间数据库链接仍然打开，直到您退出会话，或者执行 alter session close database link 命令。数据库链接可以是私有的，即归单个用户所有，或者为公有的，即此时 LOCAL 数据库中的所有用户都可使用此链接。

25.1.2 利用数据库链接进行远程查询

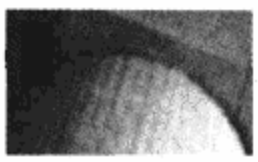
如果您是图 25-1 所示的 LOCAL 数据库的用户，则可以通过一个数据库链接访问 REMOTE 数据库中的对象。为此，只要将数据库链接名追加到远程账户可访问的任何表或视图上即可。在将数据库链接名追加到表名或视图名上时，必须在数据库链接名前使用一个@符号。

对于本地表，应该在 from 子句中引用表名：

```
select *
  from BOOKSHELF;
```

对于远程表，可使用名为 REMOTE_CONNECT 的数据库链接。在 from 子句中，引用的表名后跟上@REMOTE_CONNECT：

```
select *
    from BOOKSHELF@REMOTE_CONNECT;
```



注意：

如果数据库初始化参数指定 GLOBAL_NAMES=TRUE，那么数据库链接的名称必须与您正在连接的远程实例的名称相同。

当使用上述查询中的数据库链接时，Oracle 将使用该链接提供的用户名和密码登录到由该数据库链接指定的数据库中。然后在该账户中查询 BOOKSHELF 表，并将数据返回给启动此查询的用户。这个过程可以用图 25-2 来表示。图 25-2 所示的 REMOTE_CONNECT 数据库链接位于 LOCAL 数据库中。

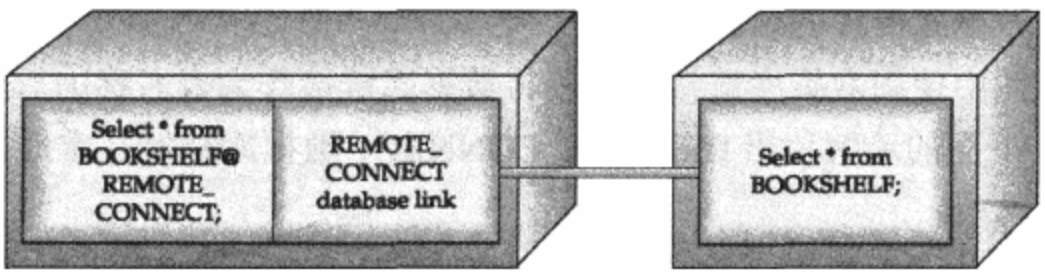
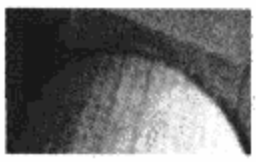


图 25-2　利用数据库链接进行远程查询

如图 25-2 所示，登录到 LOCAL 数据库，并在 from 子句中使用 REMOTE_CONNECT 数据库链接。返回的结果与直接登录到远程数据库并执行查询(不使用数据库链接)的结果相同。它使得远程数据库就像本地数据库一样。



注意：

可以通过该数据库的初始化参数文件中的 OPEN_LINK 参数，设置可在单个会话中使用的数据库链接数量的最大值。

对于使用数据库链接执行的查询有一些限制。应该避免在具有 connect by、start with 和 prior 关键字的查询中使用数据库链接。虽然在具有这些关键字的查询中可以使用数据库链接(例如，如果 prior 没有用在 connect by 子句的外面，并且 start with 不使用子查询的话)，但大多数树型结构的查询在使用数据库链接时会失败。

25.1.3　对同义词和视图使用数据库链接

可以创建引用远程对象的本地同义词和视图。为此，只要在引用远程表的地方引用数据库链接名并在前面添加一个@号即可。下面的示例说明了如何对同义词使用数据库链接。本例中的 create synonym 命令从 LOCAL 数据库中的一个账户开始执行。

```
create synonym BOOKSHELF_SYN
    for BOOKSHELF@REMOTE_CONNECT;
```

在本例中，名为 BOOKSHELF_SYN 的同义词是为通过 REMOTE_CONNECT 数据库链接访问的 BOOKSHELF 表创建的。每当此同义词在某个查询的 from 子句中使用时，都会查询远程数据库。这与前面所示的远程查询非常类似；唯一的区别是现在的数据库链接是作为

本地对象的一部分定义的(本例中为同义词)。

如果此数据库链接访问的远程账户没有被引用的表怎么办？在这种情况下，可用于该远程账户的任何同义词(无论是私有的还是公有的)都能够使用。如果远程账户有权访问的表不存在这样的同义词，就必须在查询中指定该表的所有者的名字，如下例所示：

```
create  synonym  BOOKSHELF_SYN
    for Practice.BOOKSHELF@REMOTE_CONNECT;
```

在这个示例中，数据库链接使用的远程账户既没有 BOOKSHELF 表，也没有称为 BOOKSHELF 的同义词。但是，它具有 REMOTE 数据库中远程用户 Practice 在 BOOKSHELF 表上拥有的权限。因此，应指定所有者和表名；二者都在 REMOTE 数据库中解释。这些查询和同义词的语法几乎相同，就好像所有数据都是在本地数据库中一样；唯一不同的是比在本地数据库中多了数据库链接的名称。

为了能够在视图中使用数据库链接，只要将数据库链接作为表名称的后缀添加到 create view 命令中即可。下面的示例使用 REMOTE_CONNECT 数据库链接在本地数据库中创建一个远程表的视图：

```
create view LOCAL_BOOKSHELF_VIEW
as
select * from BOOKSHELF@REMOTE_CONNECT
  where Title > 'M';
```

本示例中的 from 子句指向 BOOKSHELF@REMOTE_CONNECT。因此，该视图的基表与视图不在同一个数据库中。另外，请注意该查询中的 where 子句，它用来限制查询返回视图的记录数。

这个视图现在与本地数据库中的任何其他视图相同。此视图的访问权可授予其他用户，只要这些用户也具有对 REMOTE_CONNECT 数据库链接的访问权。

25.1.4 利用数据库链接进行远程更新

用于远程更新操作的数据库链接语法与用于远程查询的语法相同。只要将数据库链接名追加到要更新的表名后面即可。例如，为了修改远程表 BOOKSHELF 中图书的 Rating 值，应执行 update 命令，如下面的程序清单所示：

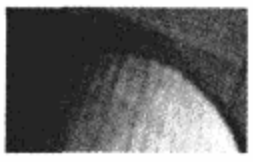
```
update BOOKSHELF@REMOTE_CONNECT
    set Rating = '5'
 where Title = 'INNUMERACY';
```

此 update 命令将使用 REMOTE_CONNECT 数据库链接登录到远程数据库。然后根据指定的 set 条件和 where 条件，更新该数据库的 BOOKSHELF 表。

您可以在 update 命令的 set 部分中使用子查询(请参阅第 15 章)。这个子查询中的 from 子句或者引用本地数据库或者远程数据库。要想在子查询中引用远程数据库，可将该数据库链接名追加到子查询的 from 子句中的表名后面。这方面的示例如下面的程序清单所示：

```
update BOOKSHELF@REMOTE_CONNECT /*in remote database*/
    set Rating =
        (select Rating
```

```
        from BOOKSHELF@REMOTE_CONNECT /*in remote database*/
         where Title = 'WONDERFUL LIFE')
where Title = 'INNUMERACY';
```



注意：
在 update 子查询的 from 子句中，如果没有把数据库链接名追加到表名后面，则将使用本地数据库中的表。即使要更新的表在远程数据库中，结果也是如此。

在本示例中，远程 BOOKSHELF 表依据远程 BOOKSHELF 表的 Rating 值进行更新。如果没有在子查询中使用数据库链接，同下面的示例一样，那么将使用本地数据库中的 BOOKSHELF 表。如果这是无意识的，那么它将导致本地数据混杂到远程数据库表中。如果刻意这样操作，则应十分小心。

```
update BOOKSHELF@REMOTE_CONNECT    /*in remote database*/
  set Rating =
      (select Rating
        from BOOKSHELF             /*in local database*/
        where Title = 'WONDERFUL LIFE')
where Title = 'INNUMERACY';
```

25.1.5　数据库链接的语法

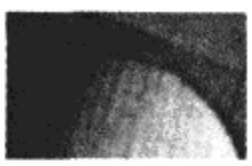
您可以使用以下命令创建数据库链接：

```
create [shared] [public] database link REMOTE_CONNECT
connect to {current_user | username
identified by password [authentication clause]}
  using 'connect string';
```

在创建数据库链接时使用的特定语法取决于以下两个条件：

- 数据库链接的“公有”或“私有”状态
- 对远程数据库使用默认或显式登录

这些条件及其相关的语法依次在以下各节中描述。



注意：
为了创建数据库链接，必须拥有 CREATE DATABASE LINK 系统权限。远程数据库中将要连接的账户必须拥有 CREATE SESSION 系统权限。

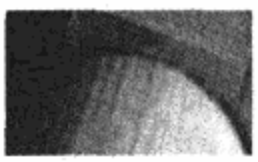
如果 GLOBAL_NAMES 初始化参数的值是 TRUE，则数据库链接名必须与它连接的数据库名相同。如果 GLOBAL_NAMES 的值是 FALSE，且您已经改变了数据库的全局名，则可以指定全局名。

1. 公有数据库链接与私有数据库链接

如前所述，在一个数据库中，公有数据库链接对数据库中所有用户可用。反之，私有数据库链接只对创建该链接的用户可用。因此，一个用户不能将私有数据库链接的访问权限授予另一个用户。数据库链接要么是公有的(对所有用户可用)，要么是私有的。

要想将数据库链接指定为公有的，可在 create database link 命令中使用 public 关键字，如下面的示例所示：

```
create public database link REMOTE_CONNECT
  connect to username identified by password
  using 'connect string';
```



注意：
要想创建公有数据库链接，必须具有 CREATE PUBLIC DATABASE LINK 系统权限。此权限包含在 Oracle 的 DBA 角色中。Oracle 中的 DBA 角色仅用于与先前版本的向后兼容。

2. 默认登录与显式登录

在创建数据库链接时，可以使用 connect to current_user 来代替 connect to…identified by…子句。如果使用 current_user 选项，那么当使用该链接时，它将试图在远程数据库中打开一个与本地数据库账户有相同用户名和密码的会话。这称为默认登录，因为用户名/密码组合的默认值就是正在使用的本地数据库中的用户名/密码组合。

下面的程序清单显示了用默认登录(有关默认登录的详细介绍，请参阅 25.3 节)创建的公有数据库链接的示例：

```
create public database link REMOTE_CONNECT
  connect to current_user
   using 'HQ';
```

在使用此数据库链接时，它将试图用当前用户的用户名和密码登录到由 HQ 服务名识别的远程数据库。如果当前用户名在远程数据库中无效，或者密码不同，则登录操作失败。这一失败将导致使用此链接的 SQL 语句失败。虽然使用默认登录具有更高的安全性，因为远程账户的密码没有存储在本地数据库中，但是默认登录使得账户的维护更加复杂。如果在本地数据库中更改账户的密码，那么也需要在通过数据库链接访问的远程数据库中将它改为相同的密码；否则，链接就可能失败。

显式登录指定将要在连接远程数据库时数据库链接将使用的用户名和密码。无论哪个本地账户使用该链接，都将使用相同的远程账户。下面的程序清单说明了用显式登录创建数据库链接的过程：

```
create public database link REMOTE_CONNECT
  connect to WAREHOUSE identified by ACCESS339
  using 'HQ';
```

这个示例说明了在数据库链接中显式登录的常用方法。在远程数据库中，创建了一个名为 Warehouse 的用户，其密码为 ACCESS339。然后可以授予 Warehouse 账户对特定表的 SELECT 访问权，只通过数据库链接使用。接下来 REMOTE_CONNECT 数据库链接为所有的本地用户提供对远程 Warehouse 账户的访问。

3. 连接串的语法

Oracle Net 使用服务名(Service Name)标识远程连接。这些服务名详细的连接信息包含在分布在网络上的每台主机上的文件中。当遇到服务名时，Oracle 将检查本地 Oracle Net 的配置文件(称为 tnsnames.ora)以确定在连接中使用哪个协议、主机名和数据库名。连接的所有信息都可在外部文件中找到。

使用 Oracle Net 时，必须知道指向远程数据库的服务名。例如，如果服务名 HQ 指定所需数据库的连接参数，那么 HQ 应该用作 create database link 命令中的连接串。以下示例说明了一个私有数据库链接，该连接使用默认登录和 Oracle Net 服务名：

```
create database link REMOTE_CONNECT
connect to current_user
  using 'HQ';
```

使用此连接时，Oracle 检查本地主机中的 tnsnames.ora 文件以确定要连接到哪个数据库。当它试图登录到该数据库时，它使用当前用户的用户名和密码。

数据库网络的 tnsnames.ora 文件应该由管理这些数据库的 DBA 来调整。以下程序清单显示了 tnsnames.ora 文件中的简化版本的条目(用于使用 TCP/IP 协议的网络)：

```
HQ =(DESCRIPTION=
      (ADDRESS_LIST =
            (ADDRESS = (PROTOCOL=TCP)
            (HOST=host1)
            (PORT=1521))
      )
      (CONNECT DATA=
      (SERVICE_NAME = HQ.host1)
    )
  )
```

在上述程序清单中，HQ 服务名映射到一个连接描述符上，该连接描述符告诉数据库要使用哪个协议(TCP/IP)，以及要连接哪个主机(host1)和数据库(HQ)。“port”信息指主机中用于连接的端口；这些数据是环境特有的。虽然不同的协议将有不同的关键字，但它们都必须传递相同的内容。

4. 使用共享数据库链接

如果使用数据库链接的 Shared Server 选项，而且应用程序将使用许多并发的数据库链接的连接，那么能够从共享数据库链接(Shared Database Link)中得到许多好处。共享数据库链接使用共享服务器连接支持数据库链接的连接。如果有访问一个远程数据库的多个并发的数据库链接，则可使用共享数据库链接来减少所需服务器连接的数目。

为了创建一个共享数据库链接，可以使用 create database link 命令的 shared 关键字。如下面的程序清单所示，您还应该为远程数据库指定模式和密码：

```
create shared database link HR_LINK_SHARED
connect to current_user
authenticated by HR identified by puffin55556d
using 'hq';
```

在访问 hq 数据库时，可以通过 connect to current_user 子句指定 HR_LINK_SHARED 数据库链接使用所连接用户的用户名和密码。为了防止未授权用户使用共享链接，共享链接需要 authenticated by 子句。在这个示例中，虽然用于身份验证的账户是一个应用程序账户，但是您也可以使用一个空模式进行身份验证。经过身份验证的账户必须具有 CREATE SESSION 系统权限。在使用 HR_LINK_SHARED 链接期间，连接试图包含针对 HR 链接账户的身份验证。

如果要修改经过身份验证的账户的密码，那么应该删除并重新创建引用它的每个数据库链接。为了简化维护工作，应创建一个只用于验证共享数据库链接连接的账户。该账户应该只具有 CREATE SESSION 系统权限，对任何应用程序表不应该具有任何权限。

如果应用程序很少使用数据库链接，则应该使用不带 shared 子句传统的数据库链接。如果不使用 shared 子句，那么每个数据库链接的连接都需要一个到远程数据库的单独连接。

25.2 为位置透明性使用同义词

在应用程序的整个生命期中，其数据很可能会从一个数据库移动到另一个数据库，或从一台主机移动到另一台主机。因此，如果将一个数据库对象的准确物理位置屏蔽起来避免用户(或应用程序)直接访问，则可以简化应用程序的维护。

实现这种位置透明性的最好方法是使用同义词。您不需要编写应用程序(或 SQL*Plus 报表)来包含指定表的所有者的查询，如：

```
select *
  from Practice.BOOKSHELF;
```

而是应该创建该表的同义词，然后在查询中引用该同义词，如下所示：

```
create synonym BOOKSHELF
    for Practice.BOOKSHELF;

select *
  from BOOKSHELF;
```

这样，查找数据所需的逻辑被移出了应用程序并移入到数据库中。将表位置逻辑移动到数据库对任何时候在模式之间移动该表都有好处。

除了隐藏应用程序中表的所有权外，还可以使用数据库链接和同义词来隐藏数据的物理位置。通过为远程表使用本地同义词，可以将另一层逻辑移出应用程序并移入到数据库中。例如，本地同义词 BOOKSHELF(如下面的程序清单所定义)引用位于不同主机的不同数据库中的一个表。如果移动此表，那么只有该链接需要修改；而使用同义词的应用程序代码不变。

```
create synonym BOOKSHELF
    for BOOKSHELF@REMOTE_CONNECT;
```

如果由数据库链接使用的远程账户不是被引用的对象的所有者，则有两种选择。首先，可以引用远程数据库中的一个可用的同义词：

```
create synonym BOOKSHELF
    for BOOKSHELF@REMOTE_CONNECT;
```

其中，数据库链接使用的远程账户中的 BOOKSHELF 是另一个用户的 BOOKSHELF 表的同义词。

其次，您可以在创建本地同义词时包含远程所有者的名字，如下面的程序清单所示：

```
create synonym BOOKSHELF
    for Practice.BOOKSHELF@REMOTE_CONNECT;
```

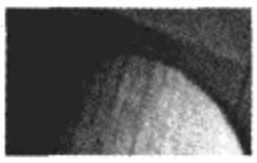
虽然这两个示例将产生同样的查询功能，但是它们之间存在区别。第二个示例，即包含所有者名字的示例维护起来更为困难，因为您没有在远程数据库中使用一个同义词，并且远程对象也可能在稍后的某个时间被移除，而使得您的本地同义词无效。

25.3　在视图中使用 User 伪列

User 伪列在使用远程数据访问方法时非常有用。例如，您可能不想让所有远程用户都看到表中的所有记录。为了解决此问题，必须将远程用户看作数据库中的特殊用户。为了施加此数据约束，您必须创建一个远程账户可访问的视图。但是在 where 子句中应使用什么内容来恰当地限制记录呢？将选中的用户名与 User 伪列组合起来，可以施加此约束。

回忆一下第 17 章的内容，用来定义视图的查询也可以引用伪列。虽然伪列在被选中时，能返回一个值，但它不是表中的实际列。User 伪列在选中时总是返回执行此查询的 Oracle 用户名。因此，如果该表中的某列包含用户名，那么这些值可以与 User 伪列的值进行比较以限制其记录，如下例所示。这个示例将查询 NAME 表。如果 Name 列的第一部分的值与输入此查询的用户的名字相同，则返回记录。

```
create view MY_CHECKOUT as
    select * from BOOKSHELF_CHECKOUT
    where SUBSTR(Name,1,INSTR(Name,' ')-1) = User;
```



注意：
我们需要转移此讨论的视角。由于这里的讨论涉及查询拥有表的数据库的操作，因此该数据库将称为“本地”数据库，而其他数据库的用户称为“远程”用户。

在限制对表记录的远程访问时，首先应该考虑哪些列最合适用于此约束。对表中的数据通常能进行逻辑划分，如 Department 或 Region。对于每个不同的划分，应在本地数据库中创建一个独立的用户账户。对本示例来说，我们将在 BOOKSHELF 表中添加一个 Region 列。现在就能够从一个表分布的多个位置记录书籍清单了：

```
alter table BOOKSHELF
    add
    (Region VARCHAR2(10));
```

假定 BOOKSHELF 表有 4 个主要的区域，并且已经为每个区域创建了一个 Oracle 账户。这时，您就可以创建每个远程用户的数据库链接，以便在本地数据库中使用特定的用户账户。

在本例中，假定区域分别为 NORTH、EAST、SOUTH 和 WEST。对于每个区域来说，将创建特定的数据库链接。例如，SOUTH 部门的成员将使用如以下程序清单所示的数据库链接：

```
create database link SOUTH_LINK
  connect to SOUTH identified by PAW
    using 'HQ';
```

本例所示的数据库链接是显式登录到远程数据库中的 SOUTH 账户的私有数据库链接。

当远程用户通过其数据库链接(如上例中的 SOUTH_LINK)进行查询时，他们将登录到 HQ 数据库中，并且使用 Department 名(如 SOUTH)作为他们的用户名。这样，对于用户查询的任何表，其 User 列的值都将为 SOUTH。

现在，创建基表的视图，在该视图的 where 子句中将 User 伪列与 Department 列的值进行比较：

```
create or replace view RESTRICTED_BOOKSHELF
    as select *
  from BOOKSHELF
where Region = User;
```

通过 SOUTH_LINK 数据库链接进行连接的用户——因而作为 SOUTH 用户登录——将只能看见 Region 值等于 SOUTH 的 BOOKSHELF 记录。如果用户从远程数据库访问您的表，他们的登录将通过数据库链接实现——您知道他们使用的本地账户，因为该账户是您创建的。

这种限制也可以在远程数据库中实现，而不是在表所驻留的数据库中实现。远程数据库中的用户可以在他们的数据库中创建视图，如下所示：

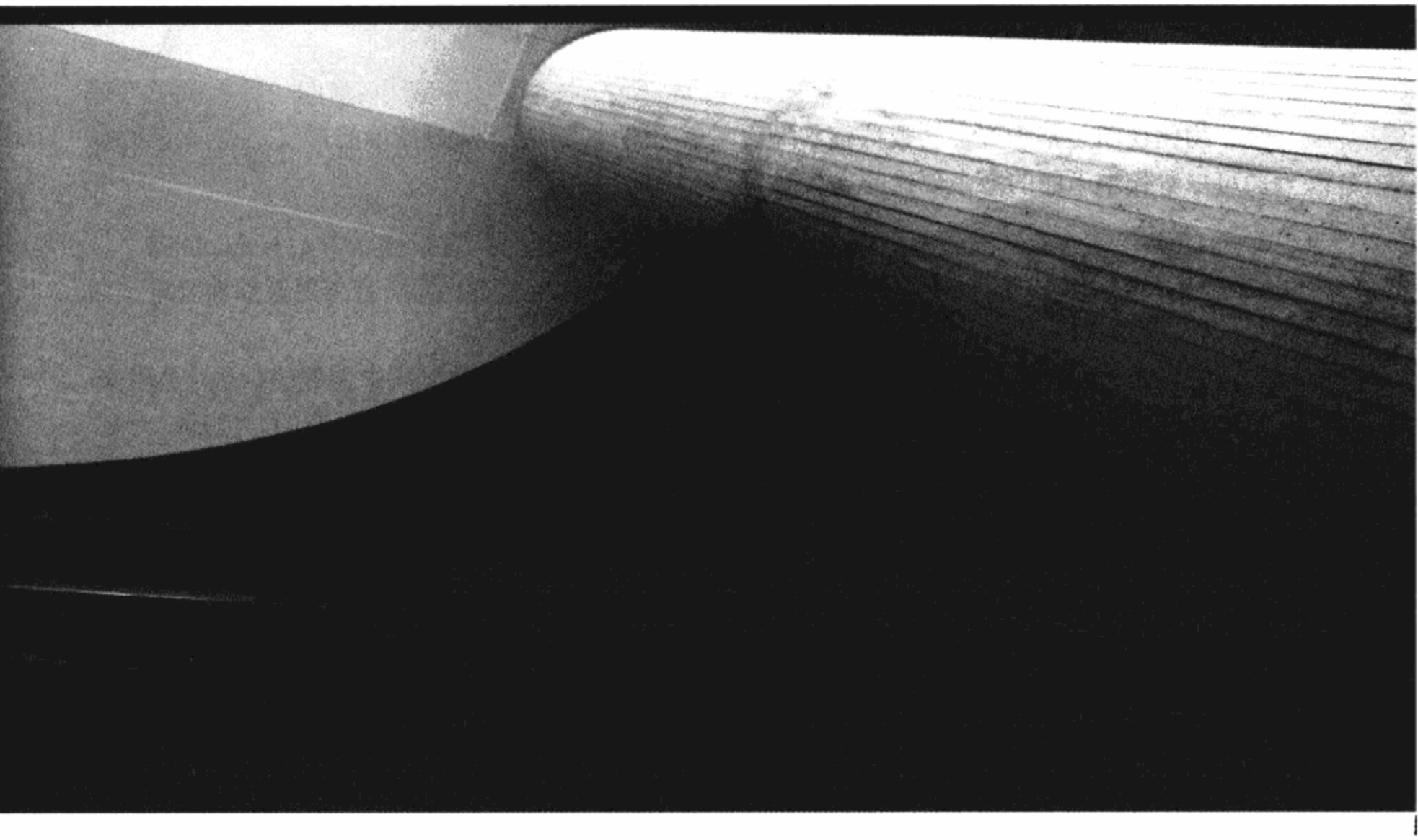
```
create or replace view SOUTH_BOOKSHELF
    as select *
  from BOOKSHELF@REMOTE_CONNECT
where Region = 'SOUTH';
```

在这个示例中，虽然 Region 限制仍然有效，但它是本地管理的，并且 Region 限制已经编码到该视图的查询中了。在这两种限制(本地或远程)之间的选择取决于需要实加限制的账户的数量。

为了保护产品数据库的安全，应该限制有权使用数据库链接的账户的权限。权限的授予通过角色完成，并使用视图(具有 with read only 或 with check option 子句)进一步限制所用的这些账户的能力，使其不能未经授权就更改数据。

这些示例表明，对 Branch 主机几乎没有什么硬件需求。它必须支持的只是前端工具和 Oracle Net。客户机(如 Branch 主机)主要用于通过数据库访问工具来显示数据。而服务器端(如 Headquarters 主机)则用于维护数据并处理用户的数据访问请求。

不管可用的配置和配置工具如何，都需要告诉 Oracle Net 如何查找远程数据库。要与 DBA 协同工作从而确保远程服务器配置正确，能够监听到新的连接请求，并且确保客户机也配置正确，能够发送这些请求。



第 26 章

使用物化视图

为了改善应用程序的性能，您可为使用分布式数据的远程表创建本地副本，或者基于 group by 操作创建汇总表。Oracle 用物化视图(Materialized View)来存储数据副本或聚集。物化视图可用来复制一个表的全部或部分，或者复制对多个表进行查询的结果；数据库可在给定的时间间隔内自动完成被复制数据的刷新。本章将介绍物化视图的常规用法，包括刷新策略，之后将介绍一些可用的优化策略。

26.1 功能

物化视图是基于查询的数据副本(即通常所说的复制品)。从最简单的形式来说，物化视图可以看作由下面的命令创建的表：

```
create table LOCAL_BOOKSHELF
as
select * from BOOKSHELF@REMOTE_CONNECT;
```

在这个示例中，名为 LOCAL_BOOKSHELF 的表是在本地数据库中创建的，并且是用来自远程数据库(由名为 REMOTE_CONNECT 的数据库链接定义)的数据填充的。但是，一旦创建了 LOCAL_BOOKSHELF 表，它的数据就可能会与主表(BOOKSHELF@REMOTE_CONNECT)的数据不同步。而且，LOCAL_BOOKSHELF 表也可以由本地用户更新，这使它与主表的同步更加复杂了。

尽管有这些同步问题，但以这种方式复制数据仍然有好处。创建远程数据的本地副本可以改善分布式查询的性能，尤其是在主表的数据不是经常改变时更是如此。也可以使用本地表的创建过程来限制返回的行、限制返回的列或生成新的列(如对所选择的值使用函数)。这对于决策支持环境是一种常用的策略，该策略利用复杂的查询定期将数据“积累”到分析时所用的汇总表。

物化视图使数据的复制和刷新过程自动进行。在创建物化视图时，可创建刷新间隔(Refresh Interval)来安排被复制数据的刷新周期。可阻止本地更新，并使用基于事务的刷新。基于事务的刷新(可用于很多种类型的物化视图)仅从主数据库发送那些已经在物化视图中更改过的行。这种功能将在本章后面介绍，它可能会极大地改进刷新的性能。

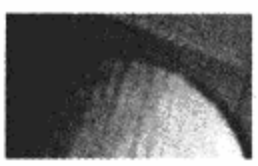
26.2 必需的系统权限

为了创建物化视图，必须具有创建这些基本对象所需的权限。必须拥有 CREATE MATERIALIZED VIEW 权限，以及 CREATE TABLE 或 CREATE ANY TABLE 系统权限。此外，还必须具有 UNLIMITED TABLESPACE 系统权限或在本地表空间中有足够的空间限额。为了创建一个提交刷新(refresh-on-commit)方式的物化视图，您还必须在自己没有的表上具有 ON COMMIT REFRESH 系统权限，或者必须具有 ON COMMIT REFRESH 系统权限。

远程表的物化视图需要查询远程表；因此，您必须具有使用(访问远程数据库的)数据库链接的权限。所使用的这个链接可以是公有的，或者是私有的。如果该数据库链接是私有的，则必须拥有 CREATE DATABASE LINK 系统权限才能创建数据库链接。关于数据库链接的详细信息请参阅第 25 章。

如果要创建利用了查询重写(Query Rewrite)功能(优化程序动态地从物化视图而不是从基表中选择数据)的物化视图，则必须具有 QUERY REWRITE 权限。如果表在其他用户的模式中，则必须拥有 GLOBAL QUERY REWRITE 权限。如果物化视图通过指定的 on commit refresh 创建，那么必须具有 ON COMMIT REFRESH 系统权限，或者在您的模式之外的每个

表上具有 ON COMMIT REFRESH 对象权限。



注意：
在 Oracle 11g 中，引用远程表的查询支持查询重写。

在 Oracle 11g 中，查询重写功能得到增强，支持包含内联视图的查询。在 Oracle 11g 之前，包含内联视图的查询只有在与物化视图的内联视图有精确的文本匹配时才能重写。

26.3　必需的表权限

当创建物化视图时，可以通过数据库链接引用远程数据库中的表。数据库链接在远程数据库中使用的账户必须有权访问此数据库链接所用的表和视图。您不能基于 SYS 用户拥有的对象来创建物化视图。

在本地数据库内，可以将一个物化视图的 SELECT 权限授予其他本地用户。因为大多数物化视图是只读的(尽管它们可以是可更新的)，所以不需要额外的授权。如果您创建一个可更新的物化视图，则必须授予用户 UPDATE 权限，用来更新物化视图和物化视图访问的本地基表。

26.4　只读物化视图与可更新的物化视图

只读物化视图不能将数据的变化传递给它的主表。可更新的物化视图能将数据的变化传递给它的主表。

尽管这看起来是一个简单的区别，但这两种物化视图之间的基本差异并不简单。只读物化视图是通过 create table as select 命令实现的。当进行事务处理时，事务处理仅在主表内进行；以后才可以选择性地把这些事务发送到只读物化视图。因此，物化视图中行更改的方式是可以控制的——物化视图的行仅随着其主表的更改而更改。

在可更新的物化视图中，物化视图中行的更改方法的比较灵活。行可以基于主表的变化而变化，或者行可由物化视图的用户直接更改。结果，您必须将记录从主表发送到物化视图，或者从物化视图发送到主表。因为存在多个更改来源，所以存在多个主站，称为多主站配置(multimaster configuration)。

在记录从物化视图传输到主站的过程中，您需要决定怎样调节冲突。例如，如果 ID=1 的记录在物化视图站点被删除，而在主站点上一个独立的表中创建了一个引用 ID=1 的记录(通过一个外键)，那么结果会怎么样呢？您将不能从主站点删除 ID=1 的记录，因为这个记录有一个与其相关的子记录。也不能在物化视图站点中插入子记录，因为其父记录(ID=1)已被删除。那么，该怎样解决这样的冲突呢？

只读物化视图使所有的事务处理在受控制的主表中进行，以避免产生冲突。虽然这样做会限制您的功能，但它对于绝大多数复制需求来说是一种理想的解决方案。如果需要多主机

复制，那么有关的准则和详细的实现说明可参阅 *Advanced Replication Guide*。

26.5 创建物化视图的语法

创建物化视图的基本语法如下所示。完整的命令语法请参阅附录 A。在描述了命令之后，给出了几个示例，用来说明创建远程数据的本地副本的方法。

```
create materialized view [user.]name
    [ organization index iot_clause]
    [ { { segment attributes clauses }
      | cluster cluster (column [, column] ...) }
      [ {partitioning clause | parallel clause | build clause } ]
      | on prebuilt table [ {with | without} reduced precision ] ]
    [ using index
     [ { physical attributes clauses| tablespace clause }
      [ physical attributes clauses| tablespace clause ]
      | using no index ]
    [ refresh clause ]
    [ for update ] [{disable | enable} query rewrite]
     as subquery;
```

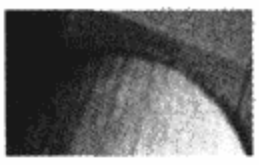
create materialized view 命令有 4 个主要部分。第一部分为标题，在标题中给物化视图命名(程序清单的第一行)：

```
create materialized view [user.] name
```

除非在标题中指定不同的用户名，否则会在您的用户账户(模式)内创建该物化视图。第 2 部分设置存储参数：

```
[ organization index iot_clause]
[ { { segment attributes clauses }
  | cluster cluster (column [, column] ...) }
  [ {partitioning clause | parallel clause | build clause } ]
  | on prebuilt table [ {with | without} reduced precision ] ]
[ using index
 [ { physical attributes clauses| tablespace clause }
  [ physical attributes clauses| tablespace clause ]
  | using no index ]
```

存储参数应用于将在本地数据库中创建的表。关于可用的存储参数的信息，请参阅附录 A 中的 STORAGE 部分。如果数据已经复制到了本地表，则可使用 on prebuilt table 子句告诉 Oracle 将该表作为一个物化视图使用。



注意：
可以指定用来在物化视图上自动创建索引的存储参数。

第 3 部分设置刷新选项：

```
[refresh clause]
```

refresh clause 的语法为：

```
{ refresh
  { { fast | complete | force }
  | on { demand | commit }
  | { start with | next } date
  | with { primary key | rowid }
  | using
    { default [ master | local ] rollback segment
    | [ master | local ] rollback segment rollback_segment
    }
    [ default [ master | local ] rollback segment
    | [ master | local ] rollback segment rollback_segment
    ]...
  }
  [ { fast | complete | force }
  | on { demand | commit }
  | { start with | next } date
  | with { primary key | rowid }
  | using
    { default [ master | local ] rollback segment
    | [ master | local ] rollback segment rollback_segment
    }
    [ default [ master | local ] rollback segment
    | [ master | local ] rollback segment rollback_segment
    ]...
  ]...
  } never refresh
  }
```

refresh 选项指定在刷新物化视图时 Oracle 应该使用的机制。可用的 3 个选项为 fast、complete 和 force。快速(fast)刷新仅在 Oracle 物化视图中的行直接与基表中的行匹配时可用，它们使用物化视图日志(materialized view log)表将特定的行从主表发送到物化视图。完全(complete)刷新将会截断数据并重新执行物化视图的基本查询来重新填充数据。force 选项告诉 Oracle：如果可用就使用快速刷新；否则，使用完全刷新。如果您已经创建了一个简单的物化视图，但希望使用完全刷新，那么应该在 create materialized view 命令中指定 refresh complete。26.8 节进一步描述了刷新选项。在 create materialized view 命令的这一部分中，也可以指定用来将物化视图中的值与主表相关联的机制——不管使用的是 RowID 还是主键值。默认情况下，使用主键。

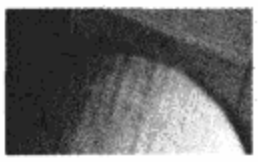
如果物化视图的主查询引用了连接或单表聚集，就可以使用 on commit 选项来控制更改的复制。如果您使用 on commit，那么当在主表上提交更改时，更改将从主表发送到复制表。如果您使用 on demand，则将在手工执行刷新命令时进行刷新。

create materialized view 命令的第 4 部分是物化视图将要使用的查询：

```
[for update] [{disable | enable} query rewrite]
as subquery
```

如果使用 for update，则物化视图是可更新的；否则，物化视图是只读的。大多数物化视

图都是主数据的只读副本。如果使用可更新的物化视图，则应该关心以下问题：更改的双向复制和协调数据更改的冲突等。可更新的物化视图是多主机复制的一个示例；关于实现一个多主机复制环境的详细内容，请参阅 *Advanced Replication Guide*。



注意：
构成物化视图基础的查询不应该使用 User 或 SysDate 伪列。

下面的示例基于一个可通过 REMOTE_CONNECT 数据库链接访问的、名为 BOOKSHELF 的远程表，在本地数据库中创建了一个名为 LOCAL_BOOKSHELF 的只读物化视图。此物化视图存储在 USERS 表空间中。

```
create materialized view LOCAL_BOOKSHELF
    tablespace USERS
    refresh force
    start with SysDate next SysDate+7
    with primary key
    as
    select * from BOOKSHELF@REMOTE_CONNECT;
```

Oracle 的响应为：

```
Materialized view created.
```

这个示例中的命令将创建一个名为 LOCAL_BOOKSHELF 的只读物化视图。它的基表将在名为 USERS 的表空间中创建。您可以将物化视图日志存储在表空间中，与其支持的物化视图分开存储。因为物化视图的基表上不存在任何物化视图日志，所以指定刷新选项 force；Oracle 将试图使用快速刷新，但在创建物化视图日志之前将使用完全刷新。此物化视图的查询指出整个 BOOKSHELF 表将被原样复制到本地数据库。一旦创建了 LOCAL_BOOKSHELF 物化视图，它的基表就用 BOOKSHELF 表的数据来填充。此后，此物化视图将会每隔七天刷新一次。未指定的存储参数将使用 USERS 表空间默认的参数值。

下面的示例基于通过 REMOTE_CONNECT 数据库链接访问的 BOOKSHELF 远程表，在本地数据库中创建了一个名为 LOCAL_CATEGORY_COUNT 的物化视图。

```
create materialized view LOCAL_CATEGORY_COUNT
    tablespace USERS
    refresh force
    start with SysDate next SysDate+7
      as
    select CategoryName, COUNT(*) CountPerCat
      from BOOKSHELF@REMOTE_CONNECT
    group by CategoryName;
```

LOCAL_CATEGORY_COUNT 物化视图中的查询统计远程 BOOKSHELF 表中的每类图书的数量。

本节的这两个示例有几点需要注意：

- LOCAL_CATEGORY_COUNT 物化视图中使用的 group by 查询可在 SQL*Plus 中针对 LOCAL_BOOKSHELF 物化视图执行。也就是说，group by 操作可以在此物化视图外执行。
- 因为 LOCAL_CATEGORY_COUNT 使用了一个 group by 子句，所以它是一个复杂的物化视图。因此，只能使用完全刷新。作为一个简单的物化视图，LOCAL_BOOKSHELF 可使用快速刷新。

上面示例中的两个物化视图引用了相同的表。由于这两个物化视图中有一个是简单的物化视图，它复制了主表的全部行和全部列，因此第二个物化视图乍看上去似乎是多余的。但有时第二个物化视图，即复杂的物化视图可能是两个视图中更有用的。

为什么会这样呢？首先，请记住这些物化视图用于方便本地用户的查询。如果这些用户总是在他们的查询中执行 group by 操作，并且他们的分组列是固定的，那么 LOCAL _CATEGORY_COUNT 对他们来说可能更有用。其次，如果主 BOOKSHELF 表上的事务非常多，或者主 BOOKSHELF 表非常小，则快速刷新和完全刷新之间的刷新时间就不会有太大的差别。对用户来说，最适合的物化视图是效率最高的那一个。

26.5.1　物化视图的类型

前面示例中的物化视图说明了物化视图的两种类型。第一，物化视图没有聚集就创建了

- **系统稳定性** 如果主站不稳定，那么可能需要执行涉及主表的数据库恢复。在您使用 Oracle 的 Data Pump Export 和 Data Pump Import 实用程序进行恢复时，行的 RowID 值会改变。如果系统经常需要导出和导入，那么应该使用基于主键的物化视图。
- **物化视图日志表的大小** Oracle 允许将主表的更改存储在独立的表中，这些表被称为物化视图日志(本章后面将介绍)。如果主键由许多列组成，则基于主键的物化视图的物化视图日志表将比同样基于 RowID 的物化视图的物化视图日志大得多。
- **参照完整性** 为了使用基于主键的物化视图，必须在主表上定义一个主键。如果您不能在主表上定义一个主键，则必须使用基于 RowID 的物化视图。

26.5.3 使用预建表

当创建物化视图时，可以使用 build immediate 来立即填充物化视图，或者使用 build deferred 稍后再填充物化视图(通过完全刷新)。如果您不得不小心地管理最初填充物化视图的那些事务，那么可以创建一个与物化视图结构相同的表，然后再填充该表。当这个表完全加载并且创建了正确的索引时，可以使用 create materalized view 命令的 on prebuilt table 子句。这个表和物化视图必须具有相同的名称，并且该表还必须具有与物化视图相同的列和数据类型(您可以使用 with reduced precision 来调节精度的差别)。表可以包含其他未管理的列。

一旦将表注册为物化视图，就可以通过刷新来维护它，而且优化程序也可以在查询重写操作中使用它。为了让查询重写在预建表上正确地工作，必须将 QUERY_ REWRITE_ INTEGRITY 初始化参数设置为 STALE_TOLERATED 或者 TRUSTED。

26.5.4 为物化视图表创建索引

当创建了物化视图后，Oracle 就会创建一个本地基表，此基表包含满足基本查询的数据。因为复制数据有明确的目的(通常是提高数据库或者网络的性能)，所以在创建了物化视图之后按照这个目的进行下去就显得非常重要。使用索引通常可以提高查询的性能。应该为在查询的 where 子句中经常使用的列创建索引；如果查询经常访问一组列，那么可以在这些列上创建一个连接索引(关于 Oracle 优化程序的更多信息，请参阅第 46 章)。

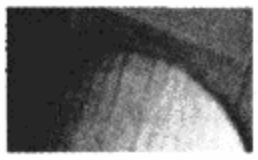
除了在主键上以外，Oracle 不会自动在复杂的物化视图的列上创建索引，您需要手工创建这些索引。为了在本地基表上创建索引，可使用 create index 命令(参见附录 A)。不应该在物化视图的本地基表上创建任何约束；Oracle 将维护主表上的基于约束的关系。

因为在用户可能从物化视图中查询的列上没有创建索引，所以应该在物化视图的本地基表上创建索引。

26.6 用物化视图更改查询执行路径

对一个大型数据库来说，物化视图可以提供几种性能优势。可以用物化视图影响优化程序从而改变查询的执行路径。这种功能称为查询重写(query rewrite)，它使优化程序能够用一个物化视图代替物化视图查询的表，即使在查询中未指定物化视图也是这样。例如，如果有一个大型表 SALES，就可以创建一个按地区对 SALES 数据求和的物化视图。如果用户查询

SALES 表以汇总某个地区的 SALES 数据，那么 Oracle 可改变该查询的路径，用物化视图代替 SALES 表。这样，就能够减少对大型表的访问次数，提高系统性能。而且，因为物化视图中的数据已经按地区分组，所以在执行查询时不必求和。



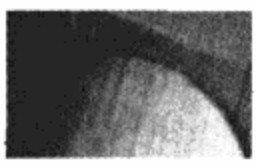
注意：
对于用作查询重写操作的组成部分的视图，必须在物化视图的定义中指定 enable query rewrite。

为了有效地利用查询重写的功能，应该创建一个维度来定义表内数据的层次结构。要执行 create dimension 命令，必须具有 CREATE DIMENSION 系统权限。可以创建一个维度用来支持 COUNTRY 示例表和 CONTINENT 示例表之间的层次结构：

```
create dimension GEOGRAPHY
    level COUNTRY_ID is COUNTRY.Country
    level CONTINENT_id is CONTINENT.Continent
    hierarchy COUNTRY_ROLLUP (
      COUNTRY_ID child of
      CONTINENT_ID
    join key COUNTRY.Continent references CONTINENT_id);
```

为了使物化视图支持查询重写，此物化视图的所有主表必须处于此物化视图的模式中，而且您必须具有 QUERY REWRITE 系统权限。如果视图和表位于不同的模式中，那么您必须具有 GLOBAL QUERY REWRITE 系统权限。一般来说，您应该在与基表相同的模式中创建物化视图；否则，必须管理用来创建和维护物化视图所需的权限和授权。

您可以通过 REWRITE 和 NOREWRITE 提示在 SQL 语句级别上启用或者禁用查询重写。当使用 REWRITE 提示时，可以考虑为优化程序指定物化视图。



注意：
因为基于不同执行路径的成本来选择查询重写功能，所以应当保持最新的统计数据。

要使用查询重写，必须设置如下初始化参数：

- OPTIMIZER_MODE = ALL_ROWS 或 FIRST_ROWS
- QUERY_REWRITE_ENABLED = TRUE
- QUERY_REWRITE_INTEGRITY = STALE_TOLERATED、TRUSTED 或 ENFORCED

在默认情况下，QUERY_REWRITE_INTEGRITY 设置为 ENFORCED；在这种模式中所有的约束都有效。优化程序仅使用来自物化视图的刷新数据，并且只使用基于 ENABLED VALIDATED 主键约束、唯一约束或者外键约束的关系。在 TRUSTED 模式中，优化程序相信物化视图中的数据是刷新过的，并且在维度和约束中声明的关系是正确的。在 STALE_TOLERATED 模式中，优化程序使用有效但包含老数据的物化视图，也使用包含最新数据的物化视图。

如果将 QUERY_REWRITE_ENABLED 设置为 FORCE，那么优化程序将重写查询以使用物化视图，即使在原有查询的预计查询成本比较低时也是如此。

如果执行了查询重写，则查询的说明计划(请参阅第 46 章)将把物化视图列为被访问的对象之一，同时作为“MAT_VIEW REWRITE ACCESS”的一项操作列出。可以使用 DBMS_MVIEW.EXPLAIN_REWRITE 过程来查看一个查询是否可以重写，以及将会涉及哪些物化视图。如果查询不能重写，那么该过程会将不能重写的原因编写成文档。

EXPLAIN_REWRITE 具有 3 个输入参数——查询、物化视图名称和语句标识符，它们可以将输出存储在一个中。在 Oracle 软件主目录下的/rdbms/admin 目录中的名为 utlxrw.sql 的脚本中，Oracle 为输出的表提供 create table 命令。utlxrw.sql 脚本创建了一个名为 REWRITE_TABLE 的表。

可以查询 REWRITE_TABLE 表来查看原来的成本、重写后的成本以及优化程序的决定。Message 列将显示优化程序作决定的原因。

有关查询重写的详细内容和约束，请参阅 *Data Warehousing Guide*。

26.7 使用 DBMS_ADVISOR

可以使用 SQL Access Advisor 来生成关于物化视图的创建和索引的建议。SQL Access Advisor 可以推荐特定的索引(和索引类型)来提高连接和其他查询的性能。SQL Access Advisor 还可以生成关于修改物化视图的建议，以便它支持查询重写或快速刷新。可以在 Oracle Enterprise Manager 内或者通过 DBMS_ADVISOR 程序包来执行 SQL Access Advisor。

注意：
为了从 DBMS_ADVISOR 程序包得到最佳的效果，应当在生成建议之前收集有关所有表、索引和连接列的统计数据。

要使用 SQL Access Advisor，无论是从 Oracle Enterprise Manager，还是通过 DBMS_ADVISOR，都应当执行如下 4 个步骤：

(1) 创建任务

(2) 定义工作负载

(3) 生成建议

(4) 查看和实施建议

您可以选择以下两种方式来创建任务：要么执行 DBMS_ADVISOR.CREATE_TASK 过程，要么使用 DBMS_ADVISOR.QUICK_TUNE 过程(26.8 节将介绍)。

工作负载由一条或多条 SQL 语句加上与这些语句有关的统计数字和属性组成。工作负载可以包括所有用于应用程序的 SQL 语句。SQL Access Advisor 根据统计数据和业务的重要性将工作负载中的条目归类。可以通过 DBMS_ADVISOR.CREATE_SQLWKLD 过程来创建工作负载。要将工作负载与父 Advisor 任务关联起来，可以使用 DBMS_ADVISOR.ADD_SQLWKLD_ REF 过程。如果没有提供工作负载，那么 SQL Access Advisor 可以根据在您的模

式中定义的维度来生成并使用一个假定的工作负载。

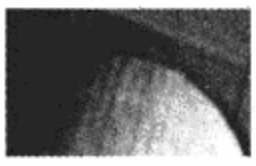
一旦任务存在并且工作负载与任务关联，就可以通过 DBMS_ADVISOR.EXECUTE_TASK 过程来生成建议。SQL Access Advisor 将考虑工作负载和系统统计数据，并试图为调整应用程序而生成建议。可以执行 DBMS _ADVISOR.GET_TASK_SCRIPT 函数或通过数据字典视图来查看建议。每条建议都可以通过 USER_ADVISOR_RECOMMENDATIONS 视图来查看(此视图也有 ALL 和 DBA 两个版本可用)。为了将建议和 SQL 语句联系起来，必须使用 USER_ADVISOR_SQLA_WK_STMTS 视图和 USER_ADVISOR_ ACTIONS。

当执行 GET_TASK_SCRIPT 过程时，Oracle 生成一个可执行的 SQL 文件，这个文件包含了用来创建、修改或删除建议对象的命令。应当在执行生成的脚本之前浏览其内容，特别需要注意表空间的说明。26.8 节将介绍如何使用 QUICK_TUNE 过程来为单个命令简化调整指导过程。

要调整一条单独的 SQL 语句，可以使用 DBMS_ADVISOR 程序包的 QUICK_TUNE 过程。QUICK_TUNE 有两个输入参数——任务名和 SQL 语句。使用 QUICK_TUNE 将使得用户不能通过 DBMS_ADVISOR 来创建工作负载和任务。

例如，下面的过程调用执行一个查询：

```
execute DBMS_ADVISOR.QUICK_TUNE(DBMS_ADVISOR.SQLACCESS_ADVISOR, -
    'MV_TUNE','SELECT PUBLISHER FROM BOOKSHELF');
```



注意：
执行此命令的用户需要具有 ADVISOR 系统权限。

由 QUICK_TUNE 生成的建议可以通过 USER_ADVISOR_ACTIONS 来查看，但是如果您使用 DBMS_ADVISOR 过程生成一个脚本文件的话，就更容易阅读了。该建议是创建物化视图来支持查询。因为只提供一条 SQL 语句，所以给出的建议是孤立的，并没有考虑到数据库或应用程序的任何其他方面。

您可以使用 CREATE_FILE 过程来自动生成一个文件，该文件包含了实施建议所需脚本的文件。首先，创建一个存储文件的目录对象：

```
create directory scripts as 'e:\scripts';
grant read on directory scripts to public;
grant write on directory scripts to public;
```

接下来，执行 CREATE_FILE 过程。这里有 3 个输入变量：脚本(通过 GET_TASK_SCRIPT 生成并且把任务名传递给该脚本)、输出目录和将要创建的文件的名称：

```
execute DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('MV_TUNE'),-
'SCRIPTS','MV_TUNE.sql');
```

由 CREATE_FILE 过程创建的 MV_TUNE.sql 文件将包含一些命令，这些命令与下面的程序清单中显示的命令相似。根据 Oracle 的不同版本，建议可能有所不同：

```
Rem Username:   PRACTICE
Rem Task:       MV_TUNE
Rem

set feedback 1
set linesize 80
set trimspool on
set tab off
set pagesize 60

whenever sqlerror CONTINUE

CREATE MATERIALIZED VIEW "PRACTICE"."MV$$_021F0001"
   REFRESH FORCE WITH ROWID
   ENABLE QUERY REWRITE
   AS SELECT PRACTICE.BOOKSHELF.ROWID C1,
"PRACTICE"."BOOKSHELF"."PUBLISHER" M1
FROM PRACTICE.BOOKSHELF;

begin
  dbms_stats.gather_table_stats('"PRACTICE"',
'"MV$$_021F0001"',NULL,dbms_stats.auto_sample_size);
end;
/

whenever sqlerror EXIT SQL.SQLCODE

begin
dbms_advisor.mark_recommendation('MV_TUNE',1,'IMPLEMENTED');
end;
/
```

MARK_RECOMMENDATION 过程允许您给建议添加注释，这样就能够在后续的脚本生成时跳过已注释的建议。MARK_RECOMMENDATION 的有效操作包括 ACCEPT、IGNORE、IMPLEMENTED 和 REJECT。

可以使用DBMS_ADVISOR程序包的TUNE_MVIEW过程生成关于物化视图的重新配置的建议。TUNE_MVIEW 生成两个输出结果集——一个用于创建新的物化视图，另一个用于删除以前创建的物化视图。最终结果应当是一个可以快速刷新的物化视图集，用来替换不能快速刷新的物化视图。

您可以通过 USER_TUNE_MVIEW 数据字典视图来查看 TUNE_MVIEW 的输出，或者可以通过在前面程序清单中出现的 GET_TASK_SCRIPT 过程和 CREATE_FILE 过程来生成脚本。

26.8 刷新物化视图

物化视图中的数据可以一次复制完毕(在创建视图时)，也可以定期地复制。create materialized view 命令允许设置刷新的时间间隔，委托数据库来安排和执行刷新任务。下面几节将介绍如何进行人工刷新和自动刷新。

26.8.1　可执行何种刷新

要知道物化视图能使用何种刷新功能和重写功能，可查询 MV_CAPABILITIES_TABLE 表。这些功能随版本不同可能会有不同，因此，您应该随着 Oracle 软件版本的升级，重新评估刷新功能。为了创建这个表，可执行 Oracle 软件主目录下的/rdbms/admin 目录中的 utlxmv.sql 脚本。

MV_CAPABILITIES_TABLE 表的列为：

```
desc MV_CAPABILITIES_TABLE

Name                                      Null?    Type
----------------------------------------- -------- ----------------
STATEMENT_ID                                       VARCHAR2(30)
MVOWNER                                            VARCHAR2(30)
MVNAME                                             VARCHAR2(30)
CAPABILITY_NAME                                    VARCHAR2(30)
POSSIBLE                                           CHAR(1)
RELATED_TEXT                                       VARCHAR2(2000)
RELATED_                                           NUM NUMBER
MSGNO                                              NUMBER(38)
MSGTXT                                             VARCHAR2(2000)
SEQ                                                NUMBER
```

为了填充 MV_CAPABILITIES_TABLE 表，可以执行 DBMS_MVIEW.EXPLAIN_MVIEW 过程，用物化视图名作为输入参数，如下所示：

```
execute DBMS_MVIEW.EXPLAIN_MVIEW('local_category_count');
```

utlxmv.sql 脚本提供了关于列值的说明，如下面的程序清单所示：

```
CREATE TABLE MV_CAPABILITIES_TABLE
  (STATEMENT_ID    VARCHAR(30),   -- Client-supplied unique statement identifier
  MVOWNER          VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
  MVNAME           VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
  CAPABILITY_NAME  VARCHAR(30),   -- A descriptive name of the particular
                                  -- capability:
                                  -- REWRITE
                                  -- Can do at least full text match
                                  -- rewrite
                                  -- REWRITE_PARTIAL_TEXT_MATCH
                                  -- Can do at least full and partial
                                  -- text match rewrite
                                  -- REWRITE_GENERAL
                                  -- Can do all forms of rewrite
                                  -- REFRESH
                                  -- Can do at least complete refresh
                                  -- REFRESH_FROM_LOG_AFTER_INSERT
                                  -- Can do fast refresh from an mv log
                                  -- or change capture table at least
                                  -- when update operations are
                                  -- restricted to INSERT
```

```
                                -- REFRESH_FROM_LOG_AFTER_ANY
                                -- can do fast refresh from an mv log
                                -- or change capture table after any
                                -- combination of updates
                                -- PCT
                                -- Can do Enhanced Update Tracking on
                                -- the table named in the RELATED_NAME
                                -- column. EUT is needed for fast
                                -- refresh after partitioned
                                -- maintenance operations on the table
                                -- named in the RELATED_NAME column
                                -- and to do non-stale tolerated
                                -- rewrite when the mv is partially
                                -- stale with respect to the table
                                -- named in the RELATED_NAME column.
                                -- EUT can also sometimes enable fast
                                -- refresh of updates to the table
                                -- named in the RELATED_NAME column
                                -- when fast refresh from an mv log
                                -- or change capture table is not
                                -- possible.
POSSIBLE          CHARACTER(1), -- T = capability is possible
                                -- F = capability is not possible
RELATED_TEXT      VARCHAR(2000),-- Owner.table.column, alias name, etc.
                                -- related to this message. The
                                -- specific meaning of this column
                                -- depends on the MSGNO column. See
                                -- the documentation for
                                -- DBMS_MVIEW.EXPLAIN_MVIEW() for details
RELATED_          NUM NUMBER,   -- When there is a numeric value
                                -- associated with a row, it goes here.
                                -- The specific meaning of this column
                                -- depends on the MSGNO column. See
                                -- the documentation for
                                -- DBMS_MVIEW.EXPLAIN_MVIEW() for details
MSGNO             INTEGER,      -- When available, QSM message #
                                -- explaining why not possible or more
                                -- details when enabled.
MSGTXT            VARCHAR(2000),-- Text associated with MSGNO.
SEQ               NUMBER);      -- Useful in ORDER BY clause when
                                -- selecting from this table.
```

一旦执行了 EXPLAIN_MVIEW 过程，就可以查询 MV_CAPABILITIES_TABLE 来决定您的选项。

```
select Capability_Name, Msgtxt
  from MV_CAPABILITIES_TABLE
where Msgtxt is not null;
```

对于 LOCAL_BOOKSHELF 物化视图，查询将返回：

```
CAPABILITY_NAME
------------------------------
```

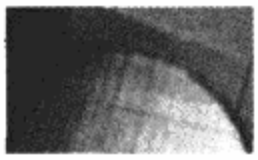
```
MSGTXT
--------------------------------------------------------------------
PCT_TABLE
relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT
the detail table does not have a materialized view log

REFRESH_FAST_AFTER_ONETAB_DML
see the reason why REFRESH_FAST_AFTER_INSERT is disabled

REFRESH_FAST_AFTER_ANY_DML
see the reason why REFRESH_FAST_AFTER_ONETAB_DML is disabled

REFRESH_FAST_PCT
PCT is not possible on any of the detail tables in the
materialized view

REWRITE_FULL_TEXT_MATCH
query rewrite is disabled on the materialized view

REWRITE_PARTIAL_TEXT_MATCH
query rewrite is disabled on the materialized view

REWRITE_GENERAL
query rewrite is disabled on the materialized view

REWRITE_PCT
general rewrite is not possible or PCT is not possible on
any of the detail tables

PCT_TABLE_REWRITE
relation is not a partitioned table

10 rows selected.
```

因为在创建物化视图时未指定 query rewrite 子句，所以 LOCAL_BOOKSHELF 物化视图禁用查询重写功能。因为基表没有物化视图日志，所以也不支持快速刷新功能。如果更改了物化视图或其基表，则为了查看新增功能应该重新生成 MV_CAPABILITIES_TABLE 中的数据。

如上面的程序清单所示，LOCAL_BOOKSHELF 物化视图不能使用快速刷新，因为它的基表没有物化视图日志。还有其他的约束会限制您使用快速刷新功能：

- 物化视图绝对不能引用非重复表达式，如 SysDate 和 RowNum。
- 物化视图绝对不能引用 RAW 或者 LONG RAW 数据类型。
- 对基于连接的物化视图，在 from 列表中的所有表的 RowID 必须是 select 列表的一部分。
- 如果有外部连接，那么所有的连接必须由 and 连接，where 子句不允许有选择集，而且在内部连接表的连接列上必须存在唯一约束。
- 对基于聚集的物化视图，物化视图日志必须包含被引用表的所有列，必须使用 rowid 和 including new values 子句，而且必须使用 sequence 子句。

有关复杂聚集的快速刷新的更多限制条件，请参阅 *Data Warehousing Guide*。



注意：
可以在 create materialized view 命令中使用 order by 子句。order by 子句将只影响物化视图的初始创建；它不会影响任何刷新。

26.8.2 用 CONSIDER FRESH 快速刷新

您可能需要在物化视图中积累历史信息，即使详细信息已不在基表中。在 Oracle 11g 中，可以使用 alter materialized view 命令的 consider fresh 子句来通知数据库，物化视图的内容正确地反映了基表，虽然这并非易事。

例如，您可能有一个物化视图，它汇总了详细的事务表中的数据。如果您从事务表中删除了历史数据，则应该刷新物化视图，因为 Oracle 将知道物化视图的数据过时了。可以使用 consider fresh 子句来改变物化视图的状态，使物化视图能够用于查询重写。然后在安排物化视图刷新时可以继续利用查询重写功能，这样，物化视图将准确地反映基本数据。

26.8.3 自动刷新

考虑一下前面介绍的物化视图 LOCAL_BOOKSHELF。其刷新计划的设置由 create materialized view 命令定义，如下面程序清单中粗体部分所示：

```
create materialized view LOCAL_BOOKSHELF
    tablespace USERS
    refresh force
    start with SysDate next SysDate+7
    with primary key
    as
    select * from BOOKSHELF@REMOTE_CONNECT;
```

刷新计划包含 3 个成分。首先，指定刷新的类型(fast、complete、never 或 force)。快速刷新利用物化视图日志(本章后面描述)将更改过的行从主表发送到物化视图。完全刷新将删除物化视图的所有行并重新填充。刷新的 force 选项告诉 Oracle，如果可用就使用快速刷新；否则，使用完全刷新。

start with 子句告诉数据库何时执行从主表到本地基表的第一次复制。它必须等于未来的某个时间点。如果不指定一个 start with 时间而指定了一个 next 值，那么 Oracle 将使用 next 子句来确定开始时间。为了控制复制刷新时间表，最好为 start with 子句指定一个值。

next 子句告诉 Oracle 在两次刷新之间等待多长时间。因为每次刷新物化视图时将应用不同的基础时间，所以 next 子句给出的是一个日期表达式而不是一个固定的日期。在前面的示例中，这个表达式为：

```
next SysDate+7
```

每次刷新物化视图时，下一次刷新将安排在 7 天之后。虽然这个示例中的刷新时间表比较简单，但是您可以用 Oracle 的许多日期函数来定制更高级的刷新时间表。例如，如果希望在每个星期一的中午进行刷新而不考虑目前的日期，那么可以设置 next 子句为：

```
NEXT_DAY(TRUNC(SysDate), "MONDAY")+12/24
```

此示例将查找当前系统日期之后的下一个星期一；日期中的时间部分将会被截掉，并且会给日期加 12 个小时(关于 Oracle 中的日期函数，请参阅第 10 章)。

为了自动对物化视图进行刷新，数据库至少必须运行一个后台快照刷新进程。此刷新进程会定期“启动”，并检查数据库中是否有物化视图需要刷新。数据库运行的进程的数目是由 JOB_QUEUE_PROCESSES 初始化参数确定。必须设置(在初始化参数文件中)这个参数的值大于 0；大多数情况下，值为 1 就够了。协调器进程按需要启动作业队列进程。

如果数据库没有运行作业队列进程，则您必须使用人工刷新方法，26.8.4 节将介绍相关内容。

26.8.4　人工刷新

除了数据库的自动刷新外，还可以对物化视图进行人工刷新。这时不考虑通常的刷新计划，新的 start with 值将基于人工刷新的时间。

为了刷新单个物化视图，可使用 DBMS_MVIEW.REFRESH。它的两个主要参数分别为要刷新的物化视图的名称和所用的方法。对于这个方法，可指定‘c’表示完全刷新、‘f’表示快速刷新、‘p’表示使用 Partition Change Tracking(PCT)的快速刷新、‘?’表示强制刷新。例如：

```
execute DBMS_MVIEN.REFRESH('local_bookshelf', 'c');
```

注意：

当在物化视图引用的表上执行分区维护操作时，就会有 Partition Change Tracking(PCT)。在 PCT 中，Oracle 重新计算受详细表中更改分区影响的物化视图中的行，并执行刷新，这样避免了执行完全刷新。

如果执行一次 DBMS_MVIEW.REFRESH 同时刷新多个物化视图，则应该在第一个参数中列出所有的物化视图名，在第二个参数中列出相应的刷新方法，如下所示：

```
execute DBMS_MVIEW.REFRESH('local_bookshelf, local_category_count', '?c');
```

在此示例中，名为 LOCAL_BOOKSHELE 的物化视图将通过强制刷新方式被刷新，而第二个物化视图将使用完全刷新。

您可利用 DBMS_MVIEW 程序包中一个独立的过程来刷新所有要进行自动刷新的物化视图。这个过程名为 REFRESH_ALL，将分别地刷新每个物化视图。它不接受任何参数。下面的程序清单显示了执行该过程的一个示例：

```
execute DBMS_MVIEW.REFRESH_ALL_MVIEWS;
```

因为这些物化视图将通过 REFRESH_ALL 不断地刷新，所以它们并不是全部同时刷新的。因此，在此过程的执行中出现的数据库或服务器故障可能会导致本地物化视图与其他物化视图不同步。如果出现这种情况，那么只需在恢复数据库后重新运行此过程即可。作为一种选择，您可用 26.9 节所述的方法创建刷新组。

DBMS_MVIEWS 程序包内的另一个过程 REFRESH_ALL_MVIEWS 用来刷新具有下列性质的所有物化视图：

- 所依赖的主表或主物化视图从最近一次更改以来未刷新过的物化视图。
- 物化视图及其所依赖的所有主表或主物化视图都是本地的。
- 位于视图 DBA_MVIEWS 中的物化视图。

如果您想创建嵌套的物化视图，就可以使用 DBMS_MVIEW.REFRESH_DEPENDENT 过程来确保在树内的所有物化视图都被刷新。

26.9 创建物化视图日志的语法

物化视图日志是一个记录主表中行的更改和物化视图的复制历史的表。这样可在刷新期间可以利用更改行的记录，只将那些主表中更改过的行发送给物化视图。基于同一个表的多个物化视图可使用相同的物化视图日志。

create materialized view log 命令的完整语法在附录 A 中给出。下面的程序清单显示了这个语法的一部分；如语法所示，它的所有参数一般都与表相关：

```
create materialized view log on [schema .] table
  [{ physical_attributes_clause
   | tablespace tablespace
   | { logging | nologging }
   | { cache | nocache }
   }
   [ physical_attributes_clause
   | tablespace tablespace
   | { logging | nologging }
   | { cache | nocache }
   }...
]
[parallel_clause] [partitioning_clauses]
[with
  { object id | primary key | rowid | sequence | ( column [, column]... ) }
  [, { object id | primary key | rowid | sequence | ( column [, column]... ) }]
    ...]
[{ including | excluding } new values];
```

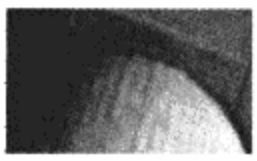
create materialized view log 命令在主表的数据库中执行，通常由主表的所有者执行。不应该为只在复杂的物化视图中涉及的表创建物化视图日志(因为它们不会被使用)。不需要给物化视图日志指定名称。

可通过下面的命令创建 BOOKSHELF 表的物化视图日志，并从拥有此表的账户内执行此下面的命令：

```
create materialized view log on BOOKSHELF
with sequence,ROWID
(Title, Publisher, CategoryName, Rating)
including new values;
```

这里需要用 with sequence 子句来支持对于多个基表进行的兼具 DML 操作的复制。

因为物化视图日志在产品数据库中会随时间的流逝而不断增大，这种增大是不可预测的，所以应当考虑将物化视图的相关对象存储到专门用于物化视图日志的表空间中。



注意：
在 Oracle 11g 中，可禁止单个会话将变更记入物化视图日志中，而对其他会话所做的变更则继续记入日志。

为了创建物化视图日志，必须具有 CREATE TABLE 和 CREATE TRIGGER 系统权限。如果以没有主表的某个用户账户创建物化视图日志，则应该具有 CREATE ANY TABLE、COMMENT ANY TABLE 和 CREATE ANY TRIGGER 系统权限，以及物化视图主表上的 SELECT 权限。

26.10　更改物化视图和日志

可以更改已有物化视图的存储参数、刷新选项和刷新时间表。如果不能肯定快照的当前设置，那么可查看 USER_MVIEWS 数据字典视图。

alter materlalized view 命令的语法在附录 A 中给出。下面的程序清单中列出的命令更改了 LOCAL_BOOKSHELF 物化视图所用的刷新选项：

```
alter materialized view LOCAL_BOOKSHELF
refresh complete;
```

LOCAL_BOOKSHELF 所有未来的刷新操作将刷新整个本地基表。

为了更改一个物化视图，您必须拥有该物化视图，或者拥有 ALTER ANY MATERIALIZED VIEW 系统权限。

为了更改一个物化视图日志，您必须拥有相应的表，具有该表的 ALTER 权限，或者具有 ALTER ANY TABLE 系统权限。

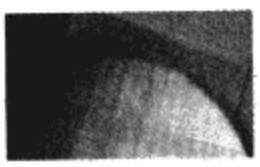
如果没有使用 RowID 或 sequence 子句来创建物化视图日志，那么可以在使用过了 alter materialized view 命令之后再添加。

26.11　删除物化视图和日志

为了删除物化视图，必须具有删除物化视图及其所有相关对象的系统权限。如果对象位于您的模式中，则应该具有 DROP MATERIALIZED VIEW 权限；如果物化视图不在您的模式内，那么需要具有 DROP ANY MATERIALIZED VIEW 系统权限。

下面的命令将删除本章前面创建的 LOCAL_CATEGORY_COUNT 物化视图：

```
drop materialized view LOCAL_CATEGORY_COUNT;
```

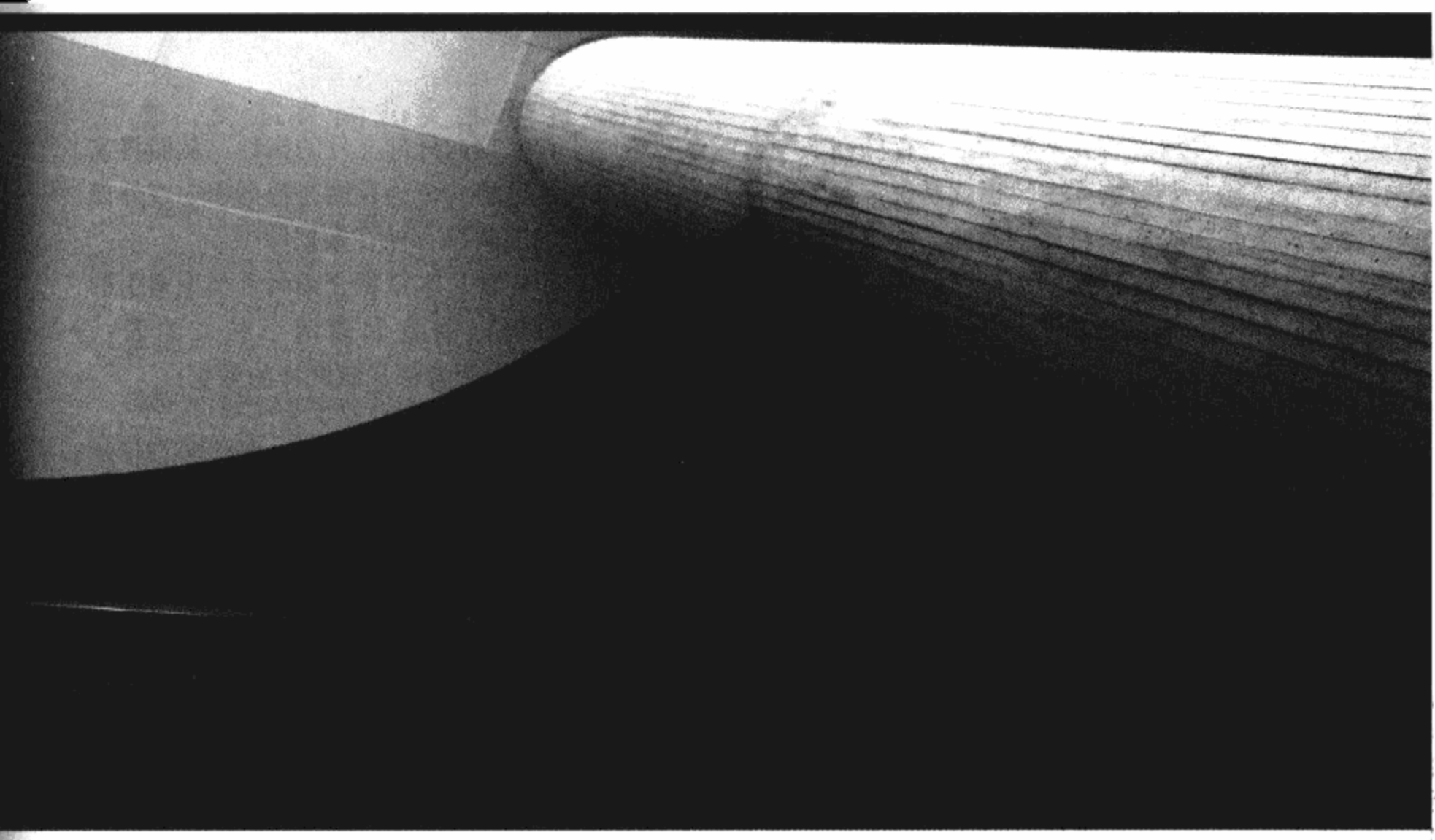


注意：
当您删除一个在预建表上创建的物化视图时，表仍然存在，但物化视图将被删除。

物化视图日志可通过 drop materialized view log 命令删除。一旦从主表中删除了物化视图日志，对于基于该表的简单的物化视图就不能执行快速刷新了。在没有简单的物化视图基于主表时，应该删除物化视图日志。下面的命令将删除本章前面在 BOOKSHELF 表上创建的物化视图日志：

```
drop materialized view log on BOOKSHELF;
```

为了删除一个物化视图日志，您必须具有删除该物化视图日志及其相关对象的能力。如果您拥有该物化视图日志，则必须具有 DROP TABLE 和 DROP TRIGGER 系统权限。如果您不拥有该物化视图日志，那么执行这条命令需要 DROP ANY TABLE 和 DROP ANY TRIGGER 系统权限。



第 27 章

使用 Oracle Text 进行文本搜索

随着数据库中文本量的不断增加，针对数据库进行的文本查询也越来越复杂。不仅需要执行字符串匹配，您还将需要新的文本搜索功能——如在多项搜索中给某些项增加权重或给文本搜索结果归类等。

可使用 Oracle Text 来完成基于文本的搜索。文本搜索功能包括通配符搜索、模糊匹配、关联分类、近似搜索、项加权和单词扩展。本章将介绍如何配置和使用 Oracle Text。

27.1 将文本添加到数据库中

可通过将文本物理地址存储在表中，或在数据库中存储指向外部文件的指针，从而将文本添加到数据库中。也就是说，对于书架上的书籍来说，可将其评论存储在数据库中或外部文件中。如果将评论存储在外部文件中，则应该将相应的文件名存储在数据库中。

为了在数据库中存储评论，需要创建 BOOK_REVIEW 表。本章将介绍两种类型索引的示例：CONTEXT 和 CTXCAT。为了支持这两个示例，需要创建两个独立的表：BOOK_REVIEW_CONTEXT 和 BOOK_REVIEW_CTXCAT，两者都加载同样的数据：

```
create table BOOK_REVIEW_CONTEXT
(Title        VARCHAR2(100) primary key,
Reviewer     VARCHAR2(25),
Review_Date  DATE,
Review_Text  VARCHAR2(4000));

insert into BOOK_REVIEW_CONTEXT values
('MY LEDGER', 'EMILY TALBOT', '01-MAY-02',
'A fascinating look into the transactions and finances of G. B. Talbot
and Dora Talbot as they managed a property in New Hampshire around 1900.
The stories come through the purchases - for medicine, doctor visits and
gravesites - for workers during harvests - for gifts at the general store
at Christmas. A great read. ');

create table BOOK_REVIEW_CTXCAT
(Title        VARCHAR2(100) primary key,
Reviewer     VARCHAR2(25),
Review_Date  DATE,
Review_Text  VARCHAR2(4000));

insert into  BOOK_REVIEW_CTXCAT values
('MY LEDGER', 'EMILY TALBOT', '01-MAY-02',
'A fascinating look into the transactions and finances of G. B. Talbot
and Dora Talbot as they managed a property in New Hampshire around 1900.
The stories come through the purchases - for medicine, doctor visits and
gravesites - for workers during harvests - for gifts at the general store
at Christmas. A great read. ');
```

BOOK_REVIEW 表的 Review_Text 列定义为 VARCHAR2(4000)数据类型。对于更长的值，可考虑使用 CLOB 数据类型。关于 CLOB 数据类型的细节，请参阅第 40 章。

可从数据库中选择评论的文本：

```
set linesize 74

select Review_Text
   from BOOK_REVIEW_CONTEXT
  where Title = 'MY LEDGER';

REVIEW_TEXT
---------------------------------------------------------------------------
A fascinating look into the transactions and finances of G. B. Talbot
and Dora Talbot as they managed a property in New Hampshire around 1900.
The stories come through the purchases - for medicine, doctor visits and
gravesites - for workers during harvests - for gifts at the general store
```

```
at Christmas. A great read.
```

27.2　文本查询和文本索引

查询文本不同于查询数据，因为单词有含义上的差别、与其他词的关系以及反义词等。用户可能希望搜索互相接近的词，或搜索互相关联的词。如果仅使用标准的关系运算符，那么这类查询将非常困难。通过对 SQL 进行扩充，使其包含文本索引，Oracle Text 允许用户就文本提出非常复杂的问题。

为了使用 Oracle Text，需要对存储文本的列创建文本索引。文本索引(text index)是一个有点含糊的词——它实际上是一个表和索引的集合表和索引的集合，这些表和索引存储关于存储在列中的文本的信息。

本章将介绍 CONTEXT 和 CTXCAT 文本索引的示例。还可以使用第 3 种类型的索引 CTXRULE 构建一个基于内容的文档分类应用程序。关于 CTXRULE 索引的详细介绍，请参阅 *Oracle Text Application Developer's Guide*。

注意：
在某个表上创建文本索引之前，如果该表没有主键，那么必须为该表创建一个主键。

可以利用 create index 命令的某个特殊版本来创建文本索引。对于 CONTEXT 索引，可使用如下程序清单在 indextype 子句中指定 Ctxsys.Context 索引类型：

```
create index Review_Context_Index
    on BOOK_REVIEW_CONTEXT(Review_Text)
indextype is ctxsys.context;
```

在创建了文本索引后，Oracle 会在用户的模式中创建大量的索引和表以支持文本查询。就像对其他索引一样，可以利用 alter index 命令重新创建文本索引。

可以利用 CTXCAT 索引类型代替 CONTEXT 索引类型：

```
create index Review_CtxCat_Index
    on BOOK_REVIEW_CTXCAT(Review_Text)
indextype is ctxsys.ctxcat;
```

CTXCAT 索引类型支持基表(BOOK_REVIEW_CTXCAT)和其文本索引之间数据的事务同步。对于 CONTEXT 索引，在基表中更改数据后，需要手工告诉 Oracle 更新文本索引中的值。虽然 CTXCAT 索引类型在文本查询中不需要生成“得分”值(而 CONTEXT 索引类型需要)，但这两种类型的查询语法极为相似。下面各节说明了通过 Oracle Text 可执行的文本查询的类型。

27.2.1　文本查询

一旦在 BOOK_REVIEW_CONTEXT 表的 Review_Text 列上创建了文本索引，文本搜索功能就会得到极大的提高。现在可以查询任何包含单词“property”的书籍评论：

```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property') >0;
```

注意：
无论在文本索引列中，单词“property”的大小写情况怎样，都会返回匹配行。

CONTAINS 函数有两个参数——列名和搜索串，它们检查 Review_Text 列的文本索引。如果在 Review_Text 列的文本索引中找到单词“property”，则数据库返回的得分大于 0，并返回匹配的 Title 值。得分是返回的记录与 CONTAINS 函数中给出的条件匹配程度的计算值。

如果要创建一个 CTXCAT 索引，那么可以使用 CATSEARCH 函数取代 CONTAINS 函数。CATSEARCH 函数接受 3 个参数：列名、搜索串以及索引集名。索引集在 27.3 节中介绍。因为下面的示例中没有索引集，所以相应的参数设置为 NULL：

```
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property', NULL) >0;
```

CATSEARCH 不计算得分，而使用“>0”语法，这样可以简化从 CONTEXT 索引到 CTXCAT 索引的移植。CTXCAT 索引支持使用索引集，这将在本章后面介绍。

当在查询中使用 CONTAINS 或 CATSEARCH 这样的函数时，查询的文本部分由 Oracle Text 处理。查询的其他部分的处理方法类似数据库中的普通查询的处理方法。文本查询处理和普通查询处理的结果合并为一组记录返回给用户。

27.2.2 可用的文本查询表达式

如果 Oracle Text 只允许进行单词精确匹配的搜索，则其功能将相当有限。不过 Oracle Text 提供了一组广泛的文本搜索功能，使用户可以定制自己的查询。大多数文本搜索功能可通过 CONTAINS 和 CATSEARCH 函数启用，这两个函数只能出现在 select 语句或子查询的 where 子句中，不能用在 insert、update 或 delete 语句的 where 子句中。

CONTAINS 函数中的运算符允许您执行下面的文本搜索：

- 单词或短语的精确匹配
- 多个单词的精确匹配(使用布尔逻辑组合几项搜索)
- 根据词汇在文本中的接近程度进行搜索
- 搜索具有相同词根的词
- 单词的“模糊”匹配
- 搜索发音相似的词

CATSEARCH 支持精确匹配搜索功能和索引集的创建，索引集将在本章后面介绍。下面几节将介绍文本搜索的相关示例，以及可用来定制文本搜索的运算符的信息。

27.2.3　一个单词精确匹配的搜索

以下对 BOOK_REVIEW 表的查询返回包含单词“property”的所有评论的标题：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property')>0;
REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property', NULL)>0;
```

在函数调用中，“>”符号称为阈值(Threshold)运算符。上面的文本搜索可转换如下：

```
select all the Title column values
from the BOOK_REVIEW_CONTEXT table
where the score for the text search of the Review_Text column
for an exact match of the word 'property'
exceeds a threshold value of 0.
```

阈值分析将得分——在执行文本搜索时 Oracle 计算的内部得分——与给出的阈值进行比较。搜索串每次在文本中出现时，各搜索的得分范围在 0~10 之间变化。对于 CONTEXT 索引，可以将得分作为查询的一部分显示。

为显示文本搜索的得分，可以使用 SCORE 函数，它有一个参数，该参数是在文本搜索内分配给得分的标签：

```
column Title format a30

select Title, SCORE(10)
  from BOOK_REVIEW_CONTEXT
where CONTAINS(Review_Text, 'property', 10)>0;

TITLE                          SCORE(10)
------------------------------ ----------
MY LEDGER                               3
```

在这个程序清单中，CONTAINS 函数的参数修改为包含用于执行文本搜索操作的标签(10)。SCORE 函数将显示与该标签相关联的文本搜索的得分。得分是根据索引文本与搜索条件之间的匹配程度进行的内部计算。

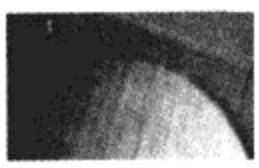
对于 CONTEXT 索引，可以在 select 语句的列表(如前面的查询所示)中或者在 group by 或 order by 子句中使用 SCORE 函数。

27.2.4　多个单词精确匹配的搜索

如果想搜索多个单词的文本，应该怎么做呢？可以用布尔逻辑(AND 和 OR)在一个查询中组合多个文本搜索的结果。还可以在同一个函数调用中搜索多项，并让 Oracle 分析搜索的结果。

例如，如果想要搜索在评论文本中具有单词“property”和“harvests”的评论，那么可输入下列查询：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
where CONTAINS(Review_Text, 'property AND harvests')>0;

REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property AND harvests', NULL)>0;
```



注意：
此搜索不是查找短语“property and harvests”，而是在搜索文本中分别查找这两个单词。27.2.5 节将介绍短语搜索的语法。

可以不在 CONTEXT 索引查询中使用 AND，取而代之的是使用“&”符号。在 SQL*Plus 中使用这种方法前，应当执行 set define off，以避免“&”符号被视为变量名的一部分：

```
set define off

REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property & harvests')>0;
```

对于 CTXCAT 索引，关键字 AND 可以完全省略：

```
REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property harvests', NULL)>0;
```

因为“&”符号或关键字 AND 都表示 AND 操作——所以 CONTAINS 函数只有当评论文本中同时含有单词“property”和“harvests”时，才返回一行。每次搜索都要符合为搜索得分定义的阈值条件。如果想要搜索的单词多于两个，那么只要把它们添加到 CONTAINS 或 CATSEARCH 子句中即可，如下面的程序清单所示：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property AND harvests AND workers')>0;

REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
where CATSEARCH(Review_Text, 'property harvests workers', NULL)>0;
```

此程序清单中的查询只有当“property”、“harvests”和“workers”的搜索得分都大于 0

时，才返回一行。

除了 AND 之外，还可以使用 OR 运算符。在使用 OR 运算符时，只要任意一个搜索条件满足定义的阈值条件，就返回一条记录。在 Oracle Text 中，由于 OR 的符号是一条垂直线(|)，因此下面的两个查询在处理方法上是相同的：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property OR harvests')>0;

select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property | harvests')>0;
```

在执行这些查询时，如果两个搜索(分别是针对“property”和“harvests”的搜索)中的任意一个返回一个大于 0 的得分，则返回一条记录。

```
REM CATSEARCH method for CTXCAT indexes:

select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property OR harvests', NULL)>0;

select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property | harvests', NULL)>0;
```

ACCUM(累加)运算符提供了另一种组合搜索的方法。ACCUM 把单个搜索的得分加到一起，然后将此总分与阈值进行比较。因为 ACCUM 的符号是一个逗号(,)，所以下面的两个查询是等价的：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property ACCUM harvests')>0;

select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property , harvests')>0;
```

虽然CATSEARCH函数调用支持ACCUM语法，但建议用户不要使用，因为CATSEARCH不计算用来与阈值进行比较的得分。

也可以在比较结果与阈值前，用 Oracle Text 从多项搜索中减去得分。CONTAINS 中的 MINUS 运算符从第 1 项搜索的得分中减去第 2 项搜索的得分。下面程序清单中的查询将确定“property”的搜索得分，并从中减去“house”的搜索得分，最后将结果与阈值进行比较。

在下面的示例中，第 2 个搜索项(“house”)在索引文本中没有找到。如果第二项在文本中(例如：“harvests”)，那么将不返回任何行。

```
REM CONTAINS method for CONTEXT indexes:
```

```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property MINUS house')>0;

select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property - house')>0;
```

可以使用“－”符号代替 MINUS 运算符，如上面的程序清单所示。

对于 CONTEXT 索引，虽然“－”运算符在将整体的搜索得分与阈值进行比较时减少得分值，但不考虑排除行。为了根据 CONTEXT 索引中的搜索项排除行，应该使用“~”符号作为 NOT 运算符。

对于 CTXCAT 索引，符号“－”与 CONTEXT 索引具有不同的含义。对于 CTXCAT 索引，“－”告诉 Oracle Text：如果在“－”后发现了搜索项，就不返回行(与 CONTEXT 索引中的“~”相同)。如果发现第 2 项，CATSEARCH 就不返回行。对于 CATSEARCH 查询，可以用词“NOT”代替“－”。

```
REM CATSEARCH method for CTXCAT indexes:

select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property - harvests', NULL)>0;

select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property NOT harvests', NULL)>0;
```

可以使用圆括号来阐明搜索条件内的逻辑。如果搜索同时使用了“AND”与“OR”，则应该用圆括号来阐明处理记录的方法。例如，如果被搜索的文本包含词“house”或者同时包含词“workers”与“harvests”，则下面的查询将返回一行：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'house OR (workers AND harvests)')>0;
```

CATSEARCH 在“workers”和“harvests”之间不需要单词“AND”：

```
REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'house | (workers harvests)', NULL)>0;
```

如果改变其中圆括号的位置，就会改变文本搜索的逻辑。如果搜索的文本包含单词“house”或“workers”，同时还包含单词“harvests”时，那么下面的查询将返回一行：

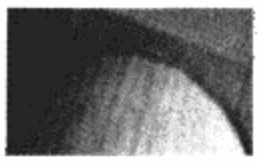
```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '(house OR workers) AND harvests')>0;
```

在计算 CONTEXT 索引多项搜索的得分值时，可以告诉 Oracle Text：使某些搜索的得分的权重比其他搜索的得分权重更大。例如，如果希望“harvests”的搜索得分在与阈值得分进行比较时有双倍的权重，那么可以使用星号(*)指出其搜索得分应该乘上的因子。

下面的查询将使“harvests”的搜索得分的计算在使用 OR 条件时加倍：

```
select Title, SCORE(10)
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'harvests*2 OR property*1',10)>5;
```

使用 AND、OR、ACCUM 和 MINUS 运算符，应该能够搜索单词匹配的任意组合。



注意：

EQUIV 运算符可以在搜索分数时，将两个搜索项一视同仁。EQUIV(可以用“=”代替)可以在比较搜索分数时的运算符组合中使用。

27.2.5 节将介绍如何搜索短语。

27.2.5　短语精确匹配的搜索

在进行短语精确匹配的搜索时，应当将整个短语作为搜索串的一部分。如果该短语含有保留字(如“AND”、“OR”或“MINUS”)，则需要使用本节讲述的转义符，这样才能正确地执行搜索。如果搜索短语包含 Oracle Text 内的保留字，则必须使用大括号({})括住相应的保留字。

下面的查询搜索其评论包含短语“doctor visits”的标题。

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'doctor visits')>0;

REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'doctor visits',NULL)>0;
```

下面的查询搜索短语“transactions and finances”，单词“and”放在大括号中。

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'transactions {and} finances')>0;

REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'transactions {and} finances',NULL)>0;
```

“transactions {and} finances”查询与“transactions and finances”查询不同。“transactions {and} finances”查询仅当短语“transactions and finances”出现在搜索文本中时，返回一条记

录。而“transactions and finances”查询在单词“transactions”的搜索得分和单词“finances”的搜索得分都高于阈值(或者在 CTXCAT 索引中，两者都存在)时，返回一条记录。

可以把整个短语括在大括号内，在这种情况下，短语中的保留字将作为搜索条件的组成部分，如下例所示：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '{transactions and finances}')>0;
```

27.2.6 搜索互相接近的单词

可以使用 Oracle Text 的接近搜索(proximity search)功能，根据搜索文档内单词之间的接近程度进行文本搜索。接近搜索为相互接近的单词返回一个高分，为完全不接近的单词返回一个低分。如果单词之间相互紧挨着，则返回 100 分。

为了对 CONTEXT 索引使用接近搜索，可以使用关键字 NEAR，如下面的示例所示：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'workers NEAR harvests')>0;
```

可以用与 NEAR 运算符等价的“;”分号代替 NEAR 运算符。修改过的查询如下面的程序清单所示：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'workers ; harvests')>0;
```

在 CONTEXT 索引查询中，可以指定搜索项之间单词的最大数目。例如，相互之间的单词在 10 个以内，可使用搜索串“NEAR((workers, harvests), 10)”。

可以使用本章介绍过的短语搜索和单词搜索来搜索精确匹配的单词和短语，以及执行精确的单词和短语的接近搜索。到目前为止，所有搜索采用的都以搜索项的精确匹配作为搜索的基础。下面的 4 节将介绍如何使用 4 种方法扩展搜索项，这 4 种方法分别是：通配符、词根、模糊匹配以及 SOUNDEX 搜索。

27.2.7 在搜索中使用通配符

在本章前面的示例中，查询选择了那些与给定条件精确匹配的文本值。例如，搜索项包含“workers”，而不是“worker”。可以用通配符来扩展查询中使用的有效搜索项的列表。

与正常的文本串通配符处理一样，有两种可用的通配符如表 27-1 所示。

表 27-1 两种可用的通配符

符 号	说 明
%	百分号；多字符的通配符
_	下划线；单字符的通配符

下面的查询将搜索所有以字符“worker”开头的文本：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'worker%')>0;
```

下面的查询将文本串的扩展限定为 3 个字符。可以使用 3 个下划线(_ _ _)取代上面查询中的%。因为下划线是单字符的通配符，所以在搜索时，文本串的扩展字符不能超过 3 个。例如，下面的文本搜索可以返回单词“workers”，但“workplace”太长，不能返回。

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'work___')>0;
```

在能够肯定搜索串中的某些字符时，应当使用通配符。若不能确定搜索串，则应该使用下面几节介绍的匹配方法之一，即词根匹配、模糊匹配和 SOUNDEX 匹配。

27.2.8　搜索具有相同词根的单词

除了使用通配符，还可以使用词根扩展(stem-expansion)功能来扩展文本串的列表。如果给出一个单词的“词根”，Oracle 将扩展要搜索的单词列表，使其包含具有相同词根的所有单词。举例如表 27-2 所示：

表 27-2　词根扩展的示例

词　根	扩 展 示 例
play	plays、played、playing、playful
works	working、work、worked、workman、workplace
have	had、has、haven't、hasn't
story	stories

因为“works”与“work”具有相同的词根，所以单词“works”的词根扩展搜索将返回包含单词“work”的文本。

为了在查询中使用词根扩展，应当使用美元符号($)。在搜索串中，“$”应放在要扩展的单词之前，在“$”与单词之间不能有空格。

下面的程序清单对 BOOK_REVIEW_CONTEXT 表进行查询，并显示所有包含与单词“manage”词根相同的单词的评论：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '$manage')>0;
```

在执行上面的查询时，Oracle 扩展单词“$manage”，以包含具有相同词根的所有单词，然后执行搜索。如果某个评论包含了与“manage”词根相同的单词，则相应的记录将返回给用户。

通过词根对项进行的扩展简化了用户的查询处理。用户在输入文本时，不必再操心动词或名词的使用形式，因为所有形式都将用作搜索的基础。也不必像使用精确匹配或通配符的查询那样给出特定的文本串。相反只要指定一个单词，Oracle Text 就能根据所给出的词动态地确定应该搜索的所有单词。

27.2.9 模糊匹配搜索

模糊匹配把特定的搜索项扩展为包括拼写类似但不一定具有相同词根的单词。模糊匹配在文本拼写错误时最有用。在查询过程中，搜索文本或用户给出的搜索串可能包含错误的拼写。

例如，下面的查询将不返回“MY LEDGER”，因为其评论并不包含词“hardest”：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'hardest')>0;
```

但它确实包括单词“harvest”。模糊匹配将返回包含单词“harvest”的评论，即使“harvest”与搜索项使用的词根不同。

为使用模糊匹配，可以在搜索项前面加一个问号，在问号与搜索项开头之间不加空格。下面的示例说明了如何使用模糊匹配功能。

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '?hardest')>0;
```

另一种可供选择的方法是使用 FUZZY 运算符，如下所示：

```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'fuzzy(hardest,60,100,w')>0;
```

FUZZY 运算符的参数按顺序依次为项、得分、结果数和权重。下面给出这些参数的解释：

- term(项) 用来指定要执行 FUZZY 扩展的单词。Oracle Text 将项扩展为包含仅在索引中出现的单词。只有单词不少于 3 个字符，FUZZY 运算符才会处理。
- score(得分) 用来指定相似性得分。将得分低于此数值的扩展项丢弃。数值范围是 1~80。默认值是 60。
- numresults(结果数) 用来指定项扩展中最多能够使用的项的数量。数值范围是 1~5 000。默认值是 100。
- weight(权重) 若指定为 WEIGHT(或 W)，则表示根据相似性得分对结果加权；若指定为 NOWEIGHT(或 N)，则不对结果加权。

27.2.10　搜索发音相似的词

词根扩展搜索基于单词的词根将一个搜索项扩展为多个搜索项。模糊匹配基于文本索引中类似的单词扩展搜索项。第 3 种搜索项扩展 SOUNDEX 是基于单词的发音扩展搜索项。SOUNDEX 扩展方法通过 SQL 中 SOUNDEX 函数使用相同的文本匹配逻辑。

为了使用 SOUNDEX 选项，需要在搜索项前面放置一个感叹号(!)。在搜索过程中，Oracle 计算文本索引中各项的 SOUNDEX 值，并搜索那些具有相同 SOUNDEX 值的单词。

如下面的查询所示，可以搜索所有包含单词“great”的评论，该查询使用 SOUNDEX 匹配技术：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '!grate')>0;
```

结果返回了“MY LEDGER”评论，因为词“grate”和“great”具有相同的 SOUNDEX 值。

还可以嵌套运算符，从而能在模糊匹配返回的项上进行词根扩展。在下面的示例中，在单词“stir”上进行模糊匹配，然后使用词根扩展对模糊匹配返回的项进行扩展：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '$?stir')>0;
```

表 27-3 总结了 CONTAINS 的主要搜索选项。关于所支持的所有语法的列表(包括同义词词典使用、同义词扩展以及 XML 支持)，请参阅 *Oracle Text Reference*。

表 27-4 总结了 CATSEARCH 的搜索选项。表 27-4 中并未列出不赞成使用的 CONTAINS 特性，如 ACCUM，这些特性没有作为 CATSEARCH 支持的特性而归档。

表 27-3　主要的 CONTAINS 选项

运 算 符	说　明
OR	如果任意一个搜索项的得分超过阈值，则返回一条记录
|	与 OR 作用相同
AND	如果两个搜索项的得分都超过阈值，则返回一条记录
&	与 AND 作用相同
ACCUM	如果搜索项的得分之和超过阈值，则返回一条记录
,	与 ACCUM 作用相同
MINUS	如果第一个搜索的得分减去第二个搜索的得分超过阈值，则返回一条记录
-	与 MINUS 作用相同
*	给搜索的得分指定不同的权重

(续表)

运 算 符	说 明
NEAR	依赖于搜索项在被搜索文本中互相接近的程度
;	与 NEAR 作用相同
NOT	如果发现 NOT 后的项，则排除该行
~	与 NOT 作用相同
EQUIV	在搜索计分中将某两项一视同仁(第 1 项等于第 2 项)
=	与 EQUIV 作用相同
{}	如果保留字(如 AND)是搜索项的组成部分，则用{}将它们括起来
%	多字符的通配符
_	单字符的通配符
$	在执行搜索前，执行搜索项的词根扩展
?	在执行搜索前，执行搜索项的模糊匹配
!	执行 SOUNDEX 搜索
()	指定判断搜索条件的顺序

表 24-4 CATSEARCH 选项

运 算 符	说 明
|	如果找到任意一个搜索项，就返回一条记录
AND	如果同时找到两个搜索项，就返回一条记录。因为这是默认动作，所以(a b)被认为(a AND b)
-	如果在连字符周围有空格，则 CATSEARCH 返回这样的行，它们包含“-”之前的项，但不包含其后的项。不带空格的连字符可看作常规字符
NOT	与“-”相同
""	括住短语
()	指定判断搜索条件的顺序
*	匹配多字符的通配符。可以放在在搜索项的结尾处或在字符中间

27.2.11 使用 ABOUT 运算符

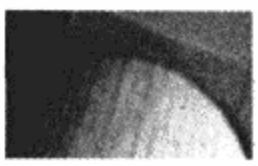
在 Oracle Text 中，可以搜索文档的主题。主题搜索与文本项搜索结合在一起。可使用 ABOUT 运算符来搜索与文档主题而不是文档中特定的项有关的项。例如：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'ABOUT(medicine)')>0;

REM CATSEARCH method for CTXCAT indexes:
select Title
```

```
 from BOOK_REVIEW_CTXCAT
where CATSEARCH(Review_Text, 'ABOUT(medicine)', NULL)>0;
```

27.2.12　索引同步

默认情况下，在使用 CONTEXT 索引时，将不得不管理文本索引的内容；而在基表更新时，文本索引并不更新。一旦更新 Review_Text 的值，其文本索引就与基表不同步了。为了使索引与基表同步，应当执行 CTX_DDL 程序包的 SYNC_INDEX 过程，如下面的程序清单所示：



注意：
授予 PRACTICE 用户在 CTX_DDL 程序包上的 EXECUTE 权限，以支持这些维护操作。

```
execute CTX_DDL.SYNC_INDEX('REVIEW_CONTEXT_INDEX');
```

CONTEXT 索引可以在提交时或在指定的时间间隔内自动维护。作为 CONTEXT 索引的 create index 命令的一部分，可以使用 sync 子句：

```
[[METADATA] SYNC (MANUAL | EVERY "interval" | ON COMMIT)]
```

例如：

```
drop index Review_Context_Index;

create index Review_Context_Index
   on BOOK_REVIEW_CONTEXT(Review_Text)
indextype is ctxsys.context
parameters ('sync (on commit)');
```

27.3　索引集

以前，当其他条件作为 where 子句的一部分用于文本搜索时，文本索引的查询就会出现问题。例如，在文本搜索完成之前或之后，需要应用非文本列上的 where 子句吗？该怎样正确地对结果排序？为改善“混合”查询的功能，可以使用索引集。索引集中的索引可以创建在结构关系列或文本列上。

为创建索引集，可以用 CTX_DDL 程序包来创建索引集，然后把索引添加到索引集中。在创建文本索引时，可以指定它所属的索引集。要执行此示例，首先应删除本章前面在 BOOK_REVIEW_CTXCAT 表上创建的 REVIEW_CTXCAT_INDEX 文本索引：

```
drop index REVIEW_CTXCAT_INDEX;
```

为了创建一个名为 Reviews 的索引集，可以使用 CREATE_INDEX_SET 过程：

```
execute CTX_DDL.CREATE_INDEX_SET('Reviews');
```

现在，可以通过 ADD_INDEX 过程把索引添加到索引集中。首先，添加一个标准的非文本

的索引：

```
execute CTX_DDL.ADD_INDEX('Reviews', 'Reviewer');
execute CTX_DDL.ADD_INDEX('Reviews', 'Review_Date');
```

现在，创建 CTXCAT 文本索引。指定 Ctxsys.Ctxcat 作为索引类型，并在 parameters 子句中列出索引集：

```
create index REVIEW_CTXCAT_INDEX
    on BOOK_REVIEW_CTXCAT(Review_Text)
  indextype is CTXSYS.CTXCAT
parameters ('index set Reviews');
```

现在，可以合并索引集搜索的结果来对结果进行排序：

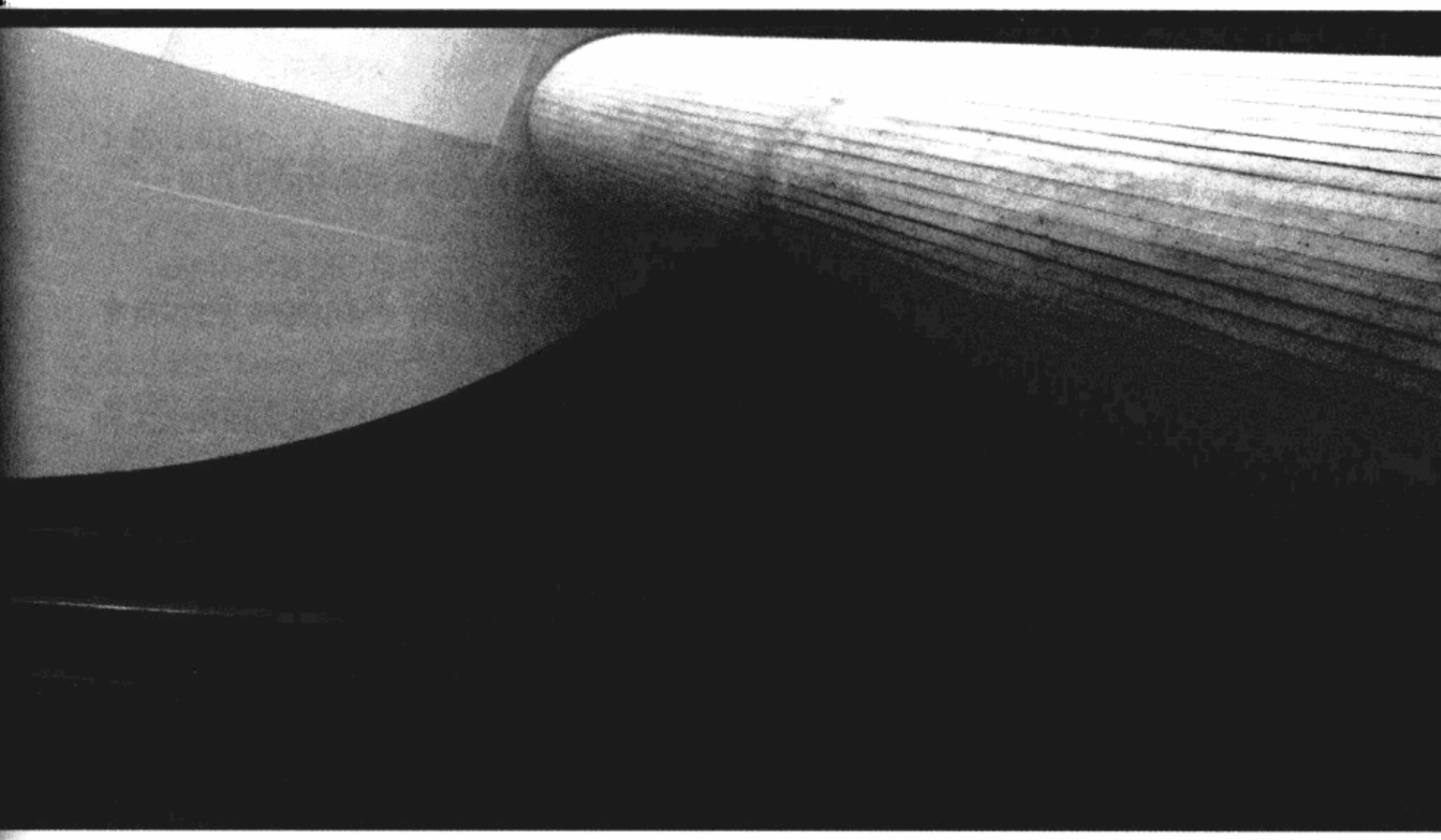
```
select * from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'great',
                 'Reviewer=''EMILY TALBOT''
                 order by Review_Date desc')>0;
```

为解析此查询，Oracle Text 将使用索引集，这样就可以正确地对结果进行排序。请注意，字符串“EMILY TALBOT”的两边有两个引号，因为这是一个文本串搜索——在执行查询时，它将转换成为一组单引号。

索引集在 NUMBER、DATE、CHAR 和 VARCHAR2 数据类型上最多可包含 99 个索引。索引集的索引中的列不能超过 30 个字节(所以在这个示例中，Title 列不能作为一个索引集的组成部分而创建索引)。索引列一定不能包含 NULL 值。

Oracle Text 包括跟踪功能，有助于在索引和查询中识别瓶颈。使用 CTX_OUTPUT 程序包的 ADD_TRACE 过程来启用跟踪功能。有用的跟踪功能包括在不同搜索组件上花费的时间、读取的字节的数量以及处理的行的数量等。

关于索引集、文本索引管理、文本应用程序开发以及跟踪的详细内容，请参阅 *Oracle Text Application Developer's Guide* 和 *Oracle Text Reference*，两者都是标准的 Oracle 文档集。



第28章

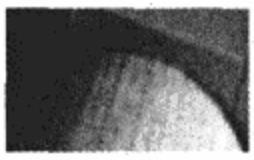
使用外部表

用户可以使用外部表的功能访问外部文件，就像它们是数据库中的表一样。创建外部表时，要在 Oracle 中定义外部表的结构和位置。在查询该表时，Oracle 读取外部表并返回结果，就像数据存储在数据库中一样。但因为这些数据在数据库的外部，所以不必关心把它们加载到数据库的过程——这对于数据仓库和大型数据库有很大好处。

外部表有一些限制——不能从 Oracle 内更新或删除外部表的行，不能对外部表创建索引。因为外部表是数据库应用程序的组成部分，所以在备份和恢复过程中必须考虑它们。虽然需要考虑这些复杂的内容，但外部表确实极大地丰富了对数据库体系结构的规划。此外，可以通过 ORACLE_DATAPUMP 访问驱动程序使用外部表把数据从表中卸载到外部文件。

28.1 访问外部数据

为了从 Oracle 内访问外部文件，首先必须使用 create directory 命令定义一个指向外部文件的位置的目录对象。需要访问外部文件的用户必须具有访问相应目录的 READ 权限。



注意：
在开始访问之前，应当验证一下外部文件是否存在，执行 create directory 命令的用户要具有 CREATE ANY DIRECTORY 系统权限。

下面的示例将创建一个名为 BOOK_DIR 的目录并授予对 PRACTICE 模式的 READ 和 WRITE 权限：

```
create directory BOOK_DIR as 'e:\oracle\external';
grant read on directory BOOK_DIR to practice;
grant write on directory BOOK_DIR to practice;
```

PRACTICE 用户现在可以读取 e:\oracle\external 目录中的文件，就像这些文件是数据库内部的文件一样。因为已经授予 PRACTICE 用户在该目录上的 WRITE 权限，所以 PRACTICE 用户可在此目录内创建日志文件、废弃文件和坏文件，就像用户执行 SQL*Loader 实用程序一样(参阅第 23 章)。

下面的程序清单为样本数据生成两个文件，一个来自 BOOKSHELF 表，另一个来自 BOOKSHELF_AUTHOR 表。请注意，spool 命令不能使用由 create directory 命令创建的目录名；应当指定完整的操作系统目录名。

```
connect practice/practice

set pagesize 0 newpage 0 feedback off
select Title||'~'||Publisher||'~'||CategoryName||'~'||Rating||'~'
  from BOOKSHELF
 order by Title

spool e:\oracle\external\bookshelf_dump.lst
/
spool off
select Title||'~'||AuthorName||'~'
  from BOOKSHELF_AUTHOR
 order by Title

spool e:\oracle\external\book_auth_dump.lst
/
spool off
```

除了数据外，输出文件还包含两行：一行在最开始，带有“SQL>/”；另一行为结束行“SQL>spool off”。为了简化示例，处理数据之前应先在操作系统级别上手工编辑文件，以删除这些多余的行。

如果另一个用户要访问 bookshelf_dump.1st 文件和 book_auth_dump.1st 文件中的数据，则必须授予该用户在 BOOK_DIR 目录上的 READ 权限：

```
grant read on directory BOOK_DIR to another_user;
```

而且，文件本身必须在操作系统级别可供 Oracle 用户读取。

28.2　创建外部表

既然外部数据可用并可以访问，就可以创建一个访问它的表结构。为了完成此工作，需要使用 create table 命令的 organization external 子句。在该子句内，可以指定数据结构，这与使用 SQL*Loader 控制文件的结构非常类似。下面的程序清单(基于 28.1 节创建的 bookshelf.1st 假脱机文件中的数据)说明了 BOOKSHELF_EXT 表的创建过程：

```
set feedback on heading on newpage 1 pagesize 60

create table BOOKSHELF_EXT
(Title         VARCHAR2(100),
Publisher      VARCHAR2(20),
CategoryName   VARCHAR2(20),
Rating VARCHAR2(2)
)
organization external
(type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
 fields terminated by "~"
                    (Title         CHAR(100),
                     Publisher     CHAR(20),
                     CategoryName  CHAR(20),
                     Rating CHAR(2)
                    ))
  location ('bookshelf_dump.lst')
);
```

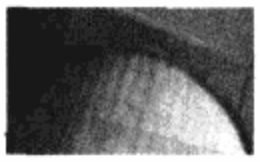
Oracle 的响应为：

```
Table created.
```

虽然在 Oracle 数据库中没有创建任何实际的数据。

类似地，可以基于 book_auth_dump.1st 假脱机文件创建一个表：

```
create table BOOKSHELF_AUTHOR_EXT
(Title       VARCHAR2(100),
AuthorName   VARCHAR2(50)
)
organization external
(type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
                    fields terminated by "~"
                   (Title        CHAR(100),
                    AuthorName   CHAR(50)
                   ))
  location ('book_auth_dump.lst')
);
```



注意：
在创建外部表时，Oracle 只进行粗略的有效性检查。在试图查询该表之前，不会看到大多数错误。访问参数的语法是特定的，访问定义中的很小的错误(包括子句的顺序)都可能会阻止对所有行的访问。

可以通过查询外部表，并将它们的内容与源表进行比较来验证外部表的内容，如下面的程序清单所示：

```
select Title from BOOKSHELF
  where CategoryName = 'CHILDRENPIC';

TITLE
----------------------------------------
GOOD DOG, CARL
POLAR EXPRESS
RUNAWAY BUNNY

3 rows selected.

select Title from BOOKSHELF_EXT
  where CategoryName = 'CHILDRENPIC';

TITLE
----------------------------------------
GOOD DOG, CARL
POLAR EXPRESS
RUNAWAY BUNNY

3 rows selected.

select COUNT(*) from BOOKSHELF_AUTHOR;

   COUNT(*)
----------
        37

select COUNT(*) from BOOKSHELF_AUTHOR_EXT;

   COUNT(*)
----------
        37
```

可以将“内部”表 BOOKSHELF_AUTHOR 和其外部对应的表 BOOKSHELF_AUTHOR_EXT 进行连接，以确认没有遗漏行，也没有添加行：

```
select * from BOOKSHELF_AUTHOR BA
where not exists
(select 'x' from BOOKSHELF_AUTHOR_EXT BAE
  where BA.Title = BAE.Title
    and BA.AuthorName = BAE.AuthorName);

no rows selected
```

BOOKSHELF_AUTHOR_EXT 表指向 book_auth_dump.1st 文件。如果更改了此文件中的

数据，则 BOOKSHELF_AUTHOR_EXT 中的数据也会改变。如本例所述，查询外部表的方式与查询(作为视图的一部分连接的)标准表的方式相同。在查询中，还可以在外部表的列上执行函数，就像处理标准表一样。

可以查询 USER_EXTERNAL_TABLE 数据字典视图，得到外部表的相关信息，这些信息包括默认目录和访问定义：

```
desc USER_EXTERNAL_TABLES

Name                                      Null?    Type
----------------------------------------- -------- --------------
TABLE_NAME                                NOT NULL VARCHAR2(30)
TYPE_OWNER                                         CHAR(3)
TYPE_NAME                                 NOT NULL VARCHAR2(30)
DEFAULT_DIRECTORY_OWNER                            CHAR(3)
DEFAULT_DIRECTORY_NAME                    NOT NULL VARCHAR2(30)
REJECT_LIMIT                                       VARCHAR2(40)
ACCESS_TYPE                                        VARCHAR2(7)
ACCESS_PARAMETERS                                  VARCHAR2(4000)
PROPERTY                                           VARCHAR2(10)
```

例如，BOOKSHELF_AUTHOR_EXT 表使用 BOOK_DIR 作为其默认目录，如下面的程序清单所示：

```
set long 500

select Default_Directory_Name,
       Access_Parameters
  from USER_EXTERNAL_TABLES
where Table_Name = 'BOOKSHELF_AUTHOR_EXT';

DEFAULT_DIRECTORY_NAME
------------------------------
ACCESS_PARAMETERS
--------------------------------------------------
BOOK_DIR
records delimited by newline
                      fields terminated by "~"
                     (Title        CHAR(100),
                      AuthorName   CHAR(50)
                     )
```

USER_EXTERNAL_TABLES 不显示表所引用的外部文件名。为了查看外部文件的信息，应查询 USER_EXTERNAL_LOCATIONS：

```
select * from USER_EXTERNAL_LOCATIONS;

TABLE_NAME
------------------------------
LOCATION
----------------------------------------------
DIR DIRECTORY_NAME
--- ------------------------------
BOOKSHELF_EXT
```

```
bookshelf_dump.lst
SYS BOOK_DIR

BOOKSHELF_AUTHOR_EXT
book_auth_dump.lst
SYS BOOK_DIR
```

28.2.1 外部表创建选项

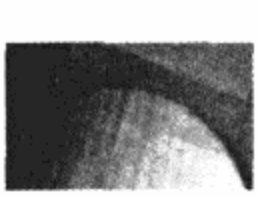
在 organization external 子句中，有 4 个主要子选项：type、default directory、access parameters 和 location。在创建外部表时，可以使用这些子句定制 Oracle 查看外部数据的方式。

1. type 和 default directory

type 组件的语法为：

```
( [type access_driver_type] external_data_properties )
[reject limit { integer | unlimited }]
```

对于外部表，访问驱动程序是用来转换外部数据的 API。如本章前面的示例所示，可将 type ORACLE_LOADER 用于外部表。如果正在使用 Data Pump(参阅第 24 章)，或者在创建外部表时正在加载外部表，则可以使用 ORACLE_DATAPUMP。如果使用 as subquery 子句从数据库卸载数据，然后重新加载，就必须使用 ORACLE_DATAPUMP 访问驱动程序。默认的驱动程序是 type ORACLE_LOADER 访问驱动程序。



注意：
因为访问驱动程序是 Oracle 软件的组成部分，所以只有那些数据库可访问的文件才能够作为外部表被访问。Oracle 用户不能访问的文件不能作为外部表。

在 type 声明后面，可设置一个“拒绝限制”值。默认情况下，不拒绝任何行——任何行的任何问题都会导致 select 语句返回一个错误。现在在另一个文件中再次生成 BOOKSHELF 数据的一个副本，这一次保留了在 spool 操作中 SQL*Plus 插入的其他行：

```
set pagesize 0 newpage 0 feedback off
select Title||'~'||Publisher||'~'||CategoryName||'~'||Rating||'~'
  from BOOKSHELF
 order by Title

spool e:\oracle\external\bookshelf_dump_2.lst
/
spool off
```

现在，创建一个引用此假脱机文件的新表，告诉 Oracle 跳过第一条记录(skip 1)并允许一个错误(reject limit 1)。这将说明第一行中的“/”字符和最后一行中的“SQL>spool off”：

```
set feedback on heading on newpage 1 pagesize 60

create table  BOOKSHELF_EXT_2
(Title        VARCHAR2(100),
Publisher     VARCHAR2(20),
CategoryName  VARCHAR2(20),
```

```
Rating          VARCHAR2(2)
)
organization external
(type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
                    skip 1
                    fields terminated by
                    "~"
                    (Title CHAR(100),
                    Publisher CHAR(20),
                    CategoryName CHAR(20),
                    Rating CHAR(2)
                   ) )
  location ('bookshelf_dump_2.lst')
)
reject limit 1
;
```

现在，可以验证表中行的数目：

```
set feedback on heading on newpage 1 pagesize 60
select COUNT(*) from BOOKSHELF_EXT_2;

   COUNT(*)
----------
         31
```

default directory 子句指出用于未指出其他目录的所有数据文件的目录对象。如果使用位于多个目录中的多个外部文件，那么可以指定其中一个作为默认目录，并利用 location 子句的目录名指出其他目录。在 location 子句中必须使用目录对象名(如 BOOK_DIR)，而不是完整的目录路径名。

2. access parameters

access parameters 子句告诉 Oracle 如何将文件中的行映射到表中的行。其语法如图 28-1 所示。

在 access parameters 子句内，首先告诉 Oracle 怎样创建一个记录——记录长度是固定的还是可变的、行怎样分隔等。在 BOOKSHELF_EXT 示例中，记录之间由换行符分隔。如果一行有多个记录，那么可使用字符串作为记录之间的分隔符。因为外部数据可能来自非 Oracle 数据库，所以 Oracle 支持多种字符集和串大小。

与使用 SQL*Loader 一样，可以指定 when 子句来限制选择哪些行。下面的程序清单创建表 BOOKSHELF_EXT_3，用一个 when 子句(以粗体显示)限制其只包含 CHILDRENPIC 种类的图书。

```
create table BOOKSHELF_EXT_3
(Title          VARCHAR2(100),
Publisher       VARCHAR2(20),
CategoryName    VARCHAR2(20),
Rating          VARCHAR2(2)
)
```

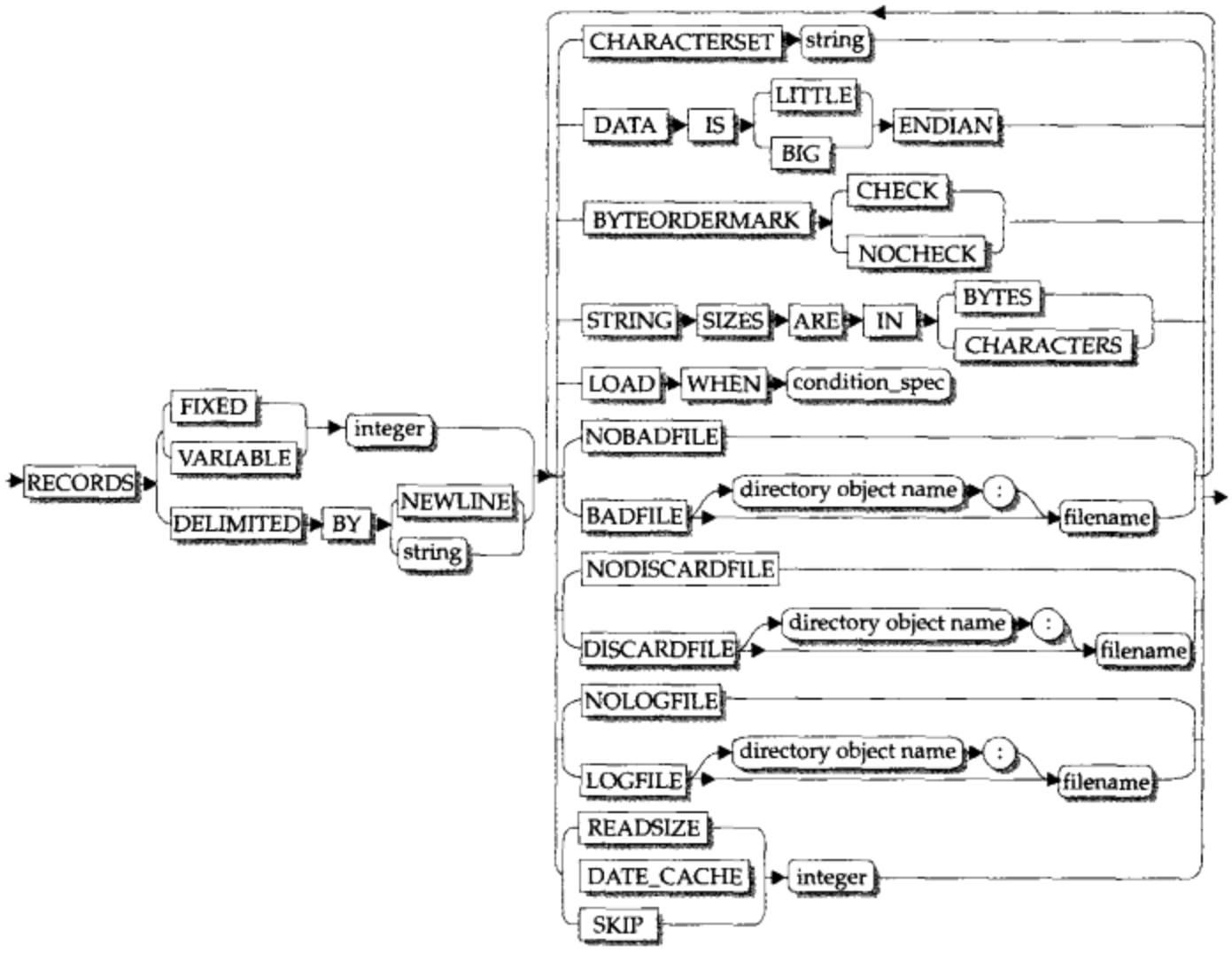


图 28-1 access parameters 子句的语法

```
organization external
 (type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
                    load when CategoryName = 'CHILDRENPIC'
                                        skip 1
                    fields terminated by "~"
                    (Title          CHAR(100),
                     Publisher      CHAR(20),
                     CategoryName   CHAR(20),
                     Rating         CHAR(2)
                   ) )
  location ('bookshelf_dump_2.lst')
)
reject limit 1
;
```

结果如下：

```
select SUBSTR(Title, 1,30), CategoryName
  from BOOKSHELF_EXT_3;

SUBSTR(TITLE,1,30)              CATEGORYNAME
------------------------------  --------------------
GOOD DOG, CARL                  CHILDRENPIC
POLAR EXPRESS                   CHILDRENPIC
RUNAWAY BUNNY                   CHILDRENPIC

3 rows selected.
```

虽然 BOOKSHELF_EXT_3 访问的文件与 BOOKSHELF_EXT_2 相同，但由于使用了 load when 子句，因此它只给出 CHILDRENPIC 种类的记录。

与使用 SQL*Loader 一样，用户可以创建日志文件、坏文件以及废弃文件。不符合 load when 条件的行将写入废弃文件。不符合 access parameters 条件的行将被写入坏文件，而加载的详细情况将被写入日志文件。对这 3 种类型的文件，都可以指定目录对象及文件名，以便将输出写到不同于输入数据文件目录的另一个目录。可以指定 nodiscardfile、nobadfile 和 nologfile，以阻止创建这些文件。在指定废弃文件、坏文件和日志文件的位置时，应当使用目录对象名(如本章示例中的 BOOK_DIR)。如果不指定日志文件、坏文件和废弃文件的位置，Oracle 将在默认目录中用系统生成的名称创建它们。

在 access parameters 子句内，还要指定字段定义和分隔符，如：

```
fields terminated by "~"
(Title              CHAR(100),
 Publisher          CHAR(20),
 CategoryName       CHAR(20),
 Rating CHAR(2))
```

虽然可以使用 missing field values are null 子句设置 NULL 列的值，但在使用该选项时必须小心。例如，AUTHOR 表在其 Comments 列中拥有 NULL 值。创建 AUTHOR_EXT 的外部表的程序清单如下所示：

```
set pagesize 0 newpage 0 feedback off
select AuthorName||'~'||Comments||'~'
  from AUTHOR
 order by AuthorName

spool e:\oracle\external\author_dump.lst
/
spool off

set feedback on heading on newpage 1 pagesize 60
create table AUTHOR_EXT
(AuthorName    VARCHAR2(50),
 Comments      VARCHAR2(100)
)
organization external
(type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
                    skip 1
                    fields terminated by "~"
                    missing field values are null
                    (AuthorName CHAR(50),
                     Comments CHAR(100)
                   ) )
  location ('author_dump.lst')
)
reject limit 1
;
```

但这并不正确，如果从 AUTHOR_EXT 中选择 AuthorName 值，就会看到该值包括：

```
select AuthorName from AUTHOR_EXT
```

```
  where AuthorName like 'S%';

AUTHORNAME
--------------------------------------------------
SOREN KIERKEGAARD
STEPHEN AMBROSE
STEPHEN JAY GOULD
SQL> spool off

4 rows selected.
```

由于 missing field values are null 子句，因此程序清单末尾的“SQL>spool off”行作为一个 AuthorName 值读入，同时还读入一个为 NULL 的 Comments 值。这在加载程序定义中加重了编码异常问题——应保证完全理解源数据以及加载程序的处理方式。在大多数情况下，强制使行失败(进入坏文件或废弃文件)并判断除一般加载的行之外失败的行，可以更好地保证数据的完整性。

关于 access parameters 子句的完整语法，请参阅附录 A 中的 SQL*Loader 项。

3. location

在 location 子句中，可以指定用作表的源数据的数据文件。可以在 location 子句中命名多个文件，条件是如果它们全都存在于用户具有 READ 权限的目录对象中。下面的示例将两个独立的 BOOKSHELF 假脱机文件组合起来，以说明将多个文件合并到一个外部表中的功能：

```
create table BOOKSHELF_EXT_4
(Title        VARCHAR2(100),
Publisher    VARCHAR2(20),
CategoryName VARCHAR2(20),
Rating VARCHAR2(2)
)
organization external
(type ORACLE_LOADER
 default directory BOOK_DIR
 access parameters (records delimited by newline
                    skip 1
                    fields terminated by "~"
                    (Title CHAR(100),
                     Publisher CHAR(20),
                     CategoryName CHAR(20),
                     Rating CHAR(2)
                   ) )
  location ('bookshelf_dump_2.lst', 'bookshelf_dump.lst')
)
reject limit 1
;
```

文件的顺序很重要，skip 1 作用于第一个文件而不是第二个文件。第二个文件 bookshelf_dump.1st 在前面编辑过，消除了其中第一行和最后一行的非数据行。结果(反映了两个文件中的行)如下面的程序清单所示：

```
select COUNT(*) from BOOKSHELF_EXT_4;
```

```
  COUNT(*)
----------
        62
```

28.2.2　创建时加载外部表

可以创建一个用 create table as select 命令填充的外部表。当创建此外部表时，Oracle 将执行特定的查询，并用此查询结果创建一个外部文件，外部表的格式基于此查询的格式。要加载外部文件，需要用 ORACLE_DATAPUMP 驱动程序取代 ORACLE_LOADER。

当使用 ORACLE_DATAPUMP 驱动程序时，Oracle 创建一个包含表数据的转储文件。您可以将这一转储文件用作其他外部文件的基础。一旦表已填充，就不能通过 SQL 对外部数据执行其他插入或更新操作。

在 Oracle Database 11g 中，您可以在表的创建过程中压缩和加密存储在外部文件中的数据。默认情况下，压缩和加密都是禁用的。

下面的程序清单说明了使用 ORACLE_DATAPUMP 驱动程序来填充转储文件：

```
create table BOOKSHELF_XT
organization external
  (
   type ORACLE_DATAPUMP
   default directory BOOK_DIR
   location ('BK_XT.DMP')
  )
as select * from BOOKSHELF;
```

现在可以将 BK_XT.DMP 文件用作另一个外部表的源文件，或者也可以直接查询 BOOKSHELF_XT 表。

为了启用压缩功能，需要修改 access parameters 设置，如下面的程序清单所示：

```
create table BOOKSHELF_XT
organization external
  (
   type ORACLE_DATAPUMP
   default directory BOOK_DIR
   access parameters (compression enabled)
   location ('BK_XT.DMP')
  )
as select * from BOOKSHELF;
```

为了在向转储文件写入数据时加密转储文件，需要使用 access parameters 子句的 encryption 选项，如下面的程序清单所示：

```
create table BOOKSHELF_XT
organization external
  (
   type ORACLE_DATAPUMP
   default directory BOOK_DIR
   access parameters (encryption enabled)
   location ('BK_XT.DMP')
  )
as select * from BOOKSHELF;
```

28.3 更改外部表

可以更改外部表的定义，以改变 Oracle 解释平面文件的方式。下面各节详细介绍了可用的选项。

28.3.1 Access Parameters 子句

无需删除和重建外部表的定义，就可以修改访问参数，从而保护授权、文件定义等。例如，以下示例显示了如何增加 BOOKSHELF_EXT_4 表中可跳过的记录数：

```
alter table BOOKSHELF_EXT_4
access parameters (records delimited by newline
                   skip 10
                   fields terminated by "~"
                   (Title         CHAR(100),
                    Publisher     CHAR(20),
                    CategoryName  CHAR(20),
                    Rating        CHAR(2)
                  ) );
select COUNT(*) from BOOKSHELF_EXT_4;

   COUNT(*)
----------
        53
```

28.3.2 Add Column 子句

可以使用 alter table 命令的 add column 子句向外部表中添加列，使用的语法与在标准表中使用的一样。

28.3.3 Default Directory 子句

可以使用 alter table 命令的 default directory 子句来修改表访问的外部文件的默认目录。该目录必须通过 create directory 命令创建。

28.3.4 Drop Column 子句

可以使用 alter table 命令的 drop column 子句从外部表中删除列，使用的语法与在标准表中使用的一样。文件中的数据保持不变。

28.3.5 Location 子句

可以通过 alter table 命令的 location 子句来修改外部表访问的文件。可以使用该选项把新文件添加至列表，或者修改外部表访问文件的顺序。

28.3.6 Modify Column 子句

可以使用 alter table 命令的 modify column 子句来更改外部表中的列，使用的语法与在标准表中使用的一样。

28.3.7　Parallel 子句

可以使用 alter table 命令的 parallel 子句来修改外部表的并行度，使用的语法与在标准表中使用的一样。

28.3.8　Project Column 子句

alter table 命令的 project column 子句告诉访问驱动程序如何使后续查询中的行有效。如果使用 project column referenced 选项，那么访问驱动程序只处理由查询选择的列。然后，如果从外部表查询不同的列集，那么结果将和第一次查询的结果不一样。如果使用 project column all 选项，那么访问驱动程序将会处理在外部表中定义的所有列，从而得到前后一致的查询结果集。默认选项是 project column referenced。

28.3.9　Reject Limit 子句

可以使用 alter table 命令的 reject limit 子句来修改外部表中允许拒绝的行数。如下所示：

```
alter table BOOKSHELF_EXT_3
    reject limit 5;
```

28.3.10　Rename To 子句

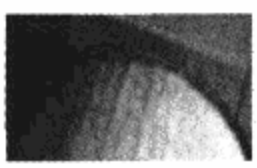
可以使用 alter table 命令的 rename to 子句来修改外部表的名称，使用的语法与在标准表中使用的一样。如下所示：

```
alter table BOOKSHELF_EXT_3
    rename to BE3;
```

28.4　外部表的优缺点和潜在用途

外部表的局限性使其不适合用于某些联机事务处理应用程序。不能对外部表执行任何 update 或 delete 操作。表的动态性越大，可能就越不适合使用外部文件。如本章前面的示例所示，可在操作系统级别动态地更改相应的文件。如果用户的应用程序只生成 insert 语句，就能将那些要插入的记录写入一个外部文件而不是数据库表。

不能对外部表创建索引。外部表没有索引不一定会对应用程序性能产生负面影响。外部表的查询速度非常快，即使每次访问需要扫描整个表也是如此。虽然外部表涉及 I/O，但现代 I/O 系统使用缓存和 RAID 技术，极大地降低了重复扫描相同文件所造成的性能损失。



注意：
要分析外部表，可以使用 DBMS_STATS 程序包。不可以通过 analyze 命令来分析外部表。

不能对外部表使用约束，甚至也不能创建 NOT NULL 约束或外键约束：

```
alter table BOOKSHELF_EXT add constraint CATFK
foreign key (CategoryName) references CATEGORY(CategoryName);

foreign key (CategoryName) references CATEGORY(CategoryName)
*
ERROR at line 2:
ORA-30657: operation not supported on external organized table
```

尽管有这些限制，外部表还提供了许多有用的功能。用户可以连接外部表(相互连接，或与其他标准表连接)，可以使用提示来强制优化程序选择不同的连接路径，还可以在查询执行路径中查看结果(关于提示和 Oracle 优化程序的详细内容，请参阅第 46 章)。

作为数据加载的另一种方案，外部表使 DBA 和应用程序开发人员无需支持长时间运行的加载程序，就可以访问数据。因为可在操作系统级别编辑文件，所以用户可快速地替换表的数据，不用担心修改表的重要事务。例如，可以使用这种功能创建多个外部表，并在这些外部表上创建一个 union all 视图，或者在多个文件上创建一个分区视图。然后，可在文件系统级别分别管理每个表的数据，并根据需要替换它们的内容。

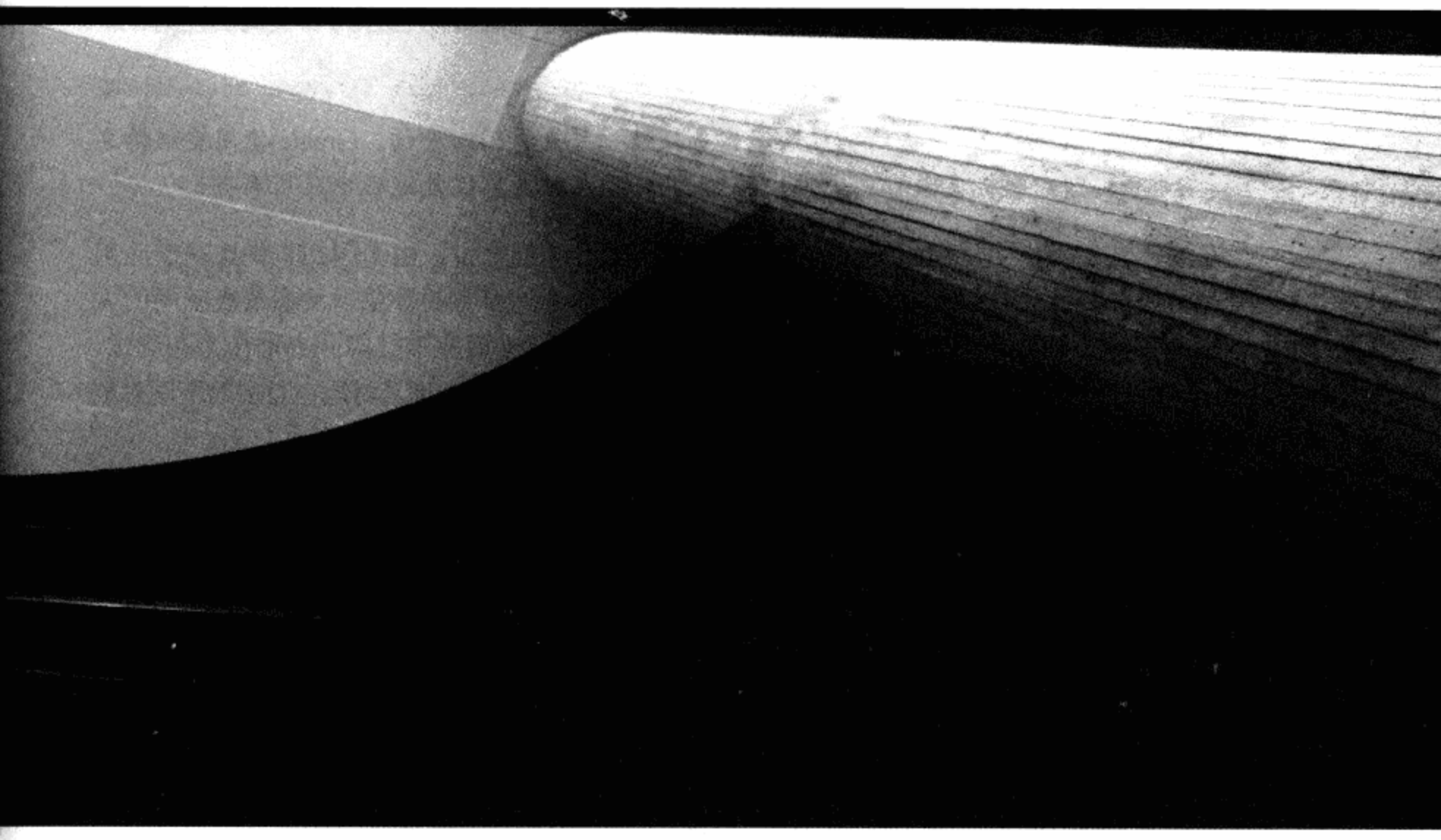
因为可以查询外部表，所以可以使用外部表作为 insert as select 命令的数据源。在此操作中，Oracle 将试图并行加载外部文件，这样可以改善性能。为进一步改善 insert as select 操作的性能，应当使用 APPEND 提示强制进行块级插入。当指定 insert as select 操作的并行度时，Oracle 启动多个 ORACLE_LOADER 访问驱动程序来并行处理数据。为进一步增强加载的性能，应避免使用变长字段、分隔字段、字符集转换、NULLIF、DEFAULTIF 和数据类型转换操作。关闭 badfile(用 nobadfile)将会消除与文件创建及维护原始行的上下文相关的成本。

在 insert as select 操作期间，可以在其处理的数据上执行函数。可以在 insert as select 命令语法中或在外部表的定义中执行函数。这种功能是外部表的一个重要优势——可以把数据的表示和处理需求集中起来，在表的定义中构建转换例程。处理的数据不存储在 SQL*Loader 控制文件或 PL/SQL 例程中。所有逻辑都创建在表的定义之中，可通过 USER_EXTERNAL_TABLES 访问。

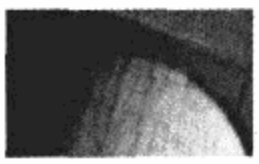
在查询过程中，外部表允许您选择特定的数据集(如本章所述，通过 load when 子句)。如果数据仓库的加载有多个数据源，那么可以选择使那些数据可用的数据源，即使这些数据位于数据库外部也行。在数据加载中，可利用此功能来维护应用程序的可用性。如果外部文件具有 FIXED 文件格式，那么这些加载可并行进行。

限制访问功能还允许实施关于数据访问的复杂的安全规则。例如，可以把敏感数据保存在数据库外部某个安全的目录中。对该目录具有 READ 权限的用户应该能使用外部表并将其与其他表连接；没有访问权限的用户将只能使用可访问的公有表中的数据。安全性较高的数据或不经常访问的动态数据在使用前不必插入数据库，或者根本不用插入数据库。

如果在数据库体系结构中使用外部表，则必须说明这些文件以及数据库的其他成分以保证备份和恢复计划。如果外部文件比数据库文件变化得更快，就应当经常地备份它们，以便利用 Oracle 的完全恢复功能。

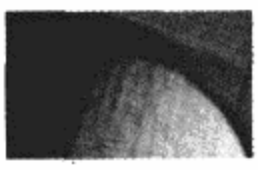


保留时间设置和撤消表空间中的可用空间的大小将极大地影响成功执行闪回查询的能力。



注意：
Oracle 使用撤消功能回滚事务处理，并支持闪回查询。Oracle 使用重做信息(在联机重做日志文件中捕获的信息)在数据库恢复过程中应用事务。

在局部恢复操作中，闪回查询是非常重要的工具。一般来说，由于闪回查询取决于应用程序开发人员控制之外的系统元素(例如，在某个时间段中的事务的数量以及撤消表空间的大小)，因此，不应该依赖于闪回查询设计应用程序的某一部分。而应该把闪回查询作为在测试、支持和数据恢复这些关键时期的一个选项。例如，可以使用闪回查询及时地在过去的不同时刻创建表的副本，以重构改变的数据。



注意：
为了使用闪回查询的某些功能，必须拥有对 DBMS_FLASHBACK 程序包的 EXECUTE 权限。大多数用户并不需要对该程序包拥有权限。

29.1 基于时间的闪回示例

BOOK_ORDER 表有 6 条记录，如以下的清单所示：

```
column Title format a30
select * from BOOK_ORDER;

TITLE                          PUBLISHER          CATEGORYNAME
------------------------------ -------------      ------------
SHOELESS JOE                   MARINER            ADULTFIC
GOSPEL                         PICADOR            ADULTFIC
SOMETHING SO STRONG            PANDORAS           ADULTNF
GALILEO'S DAUGHTER             PENGUIN            ADULTNF
LONGITUDE                      PENGUIN            ADULTNF
ONCE REMOVED                   SANCTUARY PUB      ADULTNF

6 rows selected.
```

一批货物到达后，旧的 BOOK_ORDER 记录被删除，且删除操作被提交。遗憾的是，由于并非所有的图书都在这批货中，因此使用 delete 操作是不合适的：

```
delete from BOOK_ORDER;
commit;
```

既然手中只有收到的书，那么如何从 BOOK_ORDER 表重构那些未收到书的记录呢？可以使用 Data Pump Import(参阅第 24 章)执行数据库恢复，从而还原表；或者也可以执行物理数据库恢复及时地将数据库复原至 delete 操作前的某个状态。但是，若使用闪回查询，则不必执行这些恢复操作。

首先，查询数据库中的旧数据。可以使用 select 命令的 as of timestamp 和 as of scn 子句

来指定 Oracle 闪回多长时间以前的数据。

```
select COUNT(*) from BOOK_ORDER;

   COUNT(*)
----------
         0

select COUNT(*) from BOOK_ORDER
  as of timestamp (SysDate - 5/1440);

   COUNT(*)
----------
         6
```

在执行闪回查询时，只改变数据的状态。使用了当前的系统时间(查询 SysDate 值可以看到)和当前的数据字典。如果表的结构发生变化，查询就会失败。

29.2　保存数据

如 BOOK_ORDER 示例所示，闪回查询易于实现，只要数据库的撤消管理适当地配置且撤消信息可用即可。但是，怎样用闪回数据工作呢？

最简单的方法是将数据保存在单独的表中。因为每天有 1 440 分钟，所以“SysDate - 5/1440”语句将把数据库置于 5 分钟之前的状态。请注意：如果提交后至少经过了 5 分钟，那么查询将不会返回任何行。

```
create table BOOK_ORDER_OLD
as select * from BOOK_ORDER
  as of timestamp (SysDate - 5/1440);

Table created.

select COUNT(*) from BOOK_ORDER_OLD;

   COUNT(*)
----------
         6
```

可以这样验证数据的正确性：

```
column Title format a30

select * from BOOK_ORDER_OLD;

TITLE                          PUBLISHER          CATEGORYNAME
------------------------------ ------------------ ------------
SHOELESS JOE                   MARINER            ADULTFIC
GOSPEL                         PICADOR            ADULTFIC
SOMETHING SO STRONG            PANDORAS           ADULTNF
GALILEO'S DAUGHTER             PENGUIN            ADULTNF
LONGITUDE                      PENGUIN            ADULTNF
ONCE REMOVED                   SANCTUARY PUB      ADULTNF
```

```
6 rows selected.
```

这样就可以使用这些数据来还原丢失的数据、执行选择性的更新、仅插入错误删除的行或者满足其他操作的需要。

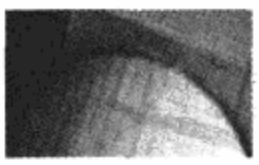
新表 BOOK_ORDER_OLD 没有索引，也没有参照完整性约束。如果需要将其连接至其他表，那么可能需要创建多个表的闪回副本，以维护数据的参照完整性。同时，请注意每一个查询都是在不同的时间点执行的——于是 as of timestamp 子句使用的相关时间将会导致混乱或不一致的结果。

可以直接使用旧的数据——但是，用户依赖对其事务可用的旧数据。从一般的、更安全的角度来说，应当创建一个表并且将旧数据暂时存储进去。

29.3 基于 SCN 的闪回示例

执行基于时间的闪回操作时，实际在执行基于 SCN 的闪回操作；用户正在依靠 Oracle 查找接近于所指定时间的 SCN。如果知道准确的 SCN，则可以精确地执行闪回操作。

为启动基于 SCN 的闪回，必须首先知道事务的 SCN。为了得到最近的更改号，可以使用 commit 命令，然后使用 select 命令的 as of scn 子句。可以在执行事务前，执行 DBMS_FLASHBACK 程序包的 GET_SYSTEM_CHANGE_NUMBER 函数来查找当前的 SCN。



注意：
在执行以下示例之前，必须拥有 DBMS_FLASHBACK 程序包的 EXECUTE 权限。

以下示例说明了该过程是针对 BOOK_ORDER_OLD 表的事务的一部分，该表在本章的第一部分创建并填充。首先，将当前的 SCN 分配给变量 SCN_FLASH，并通过 SQL*Plus 的 print 命令显示：

```
commit;

variable SCN_FLASH number;
execute :SCN_FLASH :=DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;
PL/SQL procedure successfully completed.

print SCN_FLASH

 SCN_FLASH
----------
    529732
```

接下来，使用 delete 命令并提交结果：

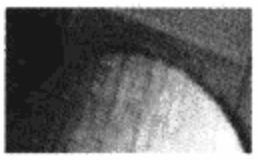
```
delete from BOOK_ORDER_OLD;

6 rows deleted.

commit;
```

现在，可以查询闪回数据。虽然已知 SCN 的值，但如果您是在同一个会话中，那么可以继续使用 SCN_FLASH 变量的值：

```
select COUNT(*) from BOOK_ORDER_OLD;

   COUNT(*)
----------
         0

select COUNT(*) from BOOK_ORDER_OLD
  as of scn (:scn_flash);

   COUNT(*)
----------
         6
```

现在，通过 as of scn 子句的访问，可以使用 BOOK_ORDER_OLD 中的闪回数据用旧值填充表：

```
insert into BOOK_ORDER_OLD
select * from BOOK_ORDER_OLD
  as of scn (:scn_flash);

6 rows created.

commit;
```



注意：

更改表结构的 DDL 操作使表原来撤消的数据无效，并且从执行 DDL 以后闪回功能在时间上受到限制。与空间有关的变化(如改变 pctfree)不会使旧的撤消数据无效。有关 flashback table 命令和 flashback database 命令的详细信息，请参阅第 30 章。

29.4　闪回查询失败的后果

如果在撤消表空间中没有足够的空间来维持闪回查询所需的所有数据，查询就会失败。即使 DBA 创建了一个巨大的撤消表空间，一系列大型事务可能还会使用全部可用的空间。每个失败的查询的一部分信息会写入数据库的警告日志中。

从用户试图恢复旧数据的角度来说，应当尽可能正确地和及时地恢复数据。一般来说，可能需要执行多个闪回查询来确定返回多长时间之前成功查询的数据，然后保存可以访问的最早的数据和最接近出错时刻的数据。一旦最早的数据从撤消表空间中删除，就再也不能使用闪回查询重新得到了。

如果不可能通过闪回取得很早之前的数据，就需要执行某种数据库恢复操作——闪回整个数据库或者通过传统的 DBA 方法恢复特定的表和表空间。如果总是出现问题，则应当增加撤消保留时间，增加分配给撤消表空间的空间，并检查应用程序的使用情况以确定为什么

有问题的事务总是不断发生。

29.5 什么 SCN 与每一行关联

可以查看数据库中与每一行关联的最接近的 SCN。

下面使用恢复到 BOOK_ORDER_OLD 表的数据来重新填充 BOOK_ORDER 表：

```
delete from BOOK_ORDER;

0 rows deleted.

insert into BOOK_ORDER
select * from BOOK_ORDER_OLD;

6 rows created.

commit;
```

现在，使用 ORA_ROWSCN 来查看与每一行关联的 SCN：

```
select Title, ORA_ROWSCN from BOOK_ORDER;

TITLE                          ORA_ROWSCN
------------------------------ ----------
SHOELESS JOE                       553531
GOSPEL                             553531
SOMETHING SO STRONG                553531
GALILEO'S DAUGHTER                 553531
LONGITUDE                          553531
ONCE REMOVED                       553531
```

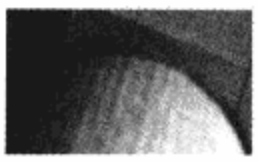
所有的行都是同一事务的一部分，都发生在 SCN 553531 上(数据库中的 SCN 与本示例中的 SCN 不同)。现在等待数据库中其他事务的发生，然后插入一条新的记录：

```
insert into BOOK_ORDER
values ('INNUMERACY','VINTAGE BOOKS','ADULTNF');

1 row created.

commit;
```

现在，新的更改已经提交，可以查看与之关联的 SCN 了：

```
select Title, ORA_ROWSCN from BOOK_ORDER;

TITLE                          ORA_ROWSCN
------------------------------ ----------
SHOELESS JOE                       553531
GOSPEL                             553531
SOMETHING SO STRONG                553531
```

```
GALILEO'S DAUGHTER              553531
LONGITUDE                       553531
ONCE REMOVED                    553531
INNUMERACY                      553853
```



注意：
ORA_ROWSCN 伪列的值不是绝对准确的，因为 Oracle 通过行所在的块提交的事务处理来跟踪 SCN。块中的新事务处理可能更新块中所有行的 ORA_ROWSCN 值。

SCN 映射到什么时间？可以使用 SCN_TO_TIMESTAMP 函数来显示变化发生的日期：

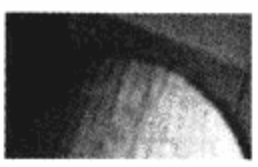
```
select SCN_TO_TIMESTAMP(555853) from DUAL;

SCN_TO_TIMESTAMP(555853)
---------------------------------------------
20-FEB-04 03.11.28.000000000 PM
```

可以合并这两个查询来查看每行最新的事务时间：

```
select Title, SCN_TO_TIMESTAMP(ORA_ROWSCN)
  from BOOK_ORDER;
```

29.6　闪回版本查询

可以显示在指定的时间间隔内存在的行的不同版本。如本章前面的示例所示，因为版本变化依赖于 SCN，所以只有提交的更改才会显示。



注意：
闪回版本查询要求 DBA 已经把 UNDO_RETENTION 初始化参数设置为一个非零值。如果 UNDO_RETENTION 值很小，那么可能遇到 ORA-30052 错误。

为了准备这些示例，首先要删除 BOOK_ORDER 表中的旧行：

```
delete from BOOK_ORDER;
```

接下来，重新填充 BOOK_ORDER 表：

```
select SysTimeStamp from DUAL;

insert into BOOK_ORDER
select * from BOOK_ORDER_OLD;

select SysTimeStamp from DUAL;
```

然后，等待几分钟，再更新所有的行：

```
select SysTimeStamp from DUAL;
```

```
update BOOK_ORDER set CategoryName = 'ADULTF';

select SysTimeStamp from DUAL;
```

要执行闪回版本查询，可以使用 select 命令的 versions between 子句。可以指定时间戳或者 SCN。在此示例中，时间戳子句的格式基于 Oracle 的标准格式(对于当前值应该是 select SysTimeStamp from DUAL)：

```
select *
  from BOOK_ORDER
versions between timestamp
  to_timestamp('20-FEB-04 16.00.20','DD-MON-YY HH24.MI.SS')
  and
  to_timestamp('20-FEB-04 16.06.20','DD-MON-YY HH24.MI.SS') ;
```

执行该查询时，Oracle 将为每行的每个版本返回一行，该版本出现在 versions between 子句中指定的起点和终点之间。对那些返回的行，可以查询其他伪列，如表 29-1 所示。

表 29-1 返回的行可以查询的其他伪列

伪 列	说 明
VERSIONS_STARTSCN	数据第一次使值反射时的 SCN。如果为 NULL，则行在查询的时间下限之前创建
VERSIONS_STARTTIME	数据第一次使值反射时的时间戳。如果为 NULL，则行在查询的时间下限之前创建
VERSIONS_ENDSCN	行版本过期时的 SCN。如果为 NULL，则行是当前版本，或已经被删除
VERSIONS_ENDTIME	行版本过期时的时间戳。如果为 NULL，则行是当前版本，或已经被删除
VERSIONS_XID	创建行版本的事务标识符
VERSIONS_OPERATION	执行事务的操作(I 表示插入，U 表示更新，D 表示删除)

以下示例显示了对闪回版本查询的使用。当前行的 Versions_StartSCN 值为 NULL；旧行已经被更新。时间戳将因平台的不同而不同。

```
select Title, Versions_StartSCN, Versions_Operation
  from BOOK_ORDER
versions between timestamp
   TO_TIMESTAMP('20-FEB-04 16.00.20','DD-MON-YY HH24.MI.SS')
   and
   TO_TIMESTAMP('20-FEB-04 16.06.20','DD-MON-YY HH24.MI.SS') ;

TITLE                          VERSIONS_STARTSCN  V
------------------------------ -----------------  -
ONCE REMOVED                              568127  U
LONGITUDE                                 568127  U
GALILEO'S DAUGHTER                        568127  U
SOMETHING SO STRONG                       568127  U
GOSPEL                                    568127  U
SHOELESS JOE                              568127  U
SHOELESS JOE
```

```
GOSPEL
SOMETHING SO STRONG
GALILEO'S DAUGHTER
LONGITUDE
ONCE REMOVED
```

versions between 子句可以用在 DML 命令和 DDL 命令的子查询中。

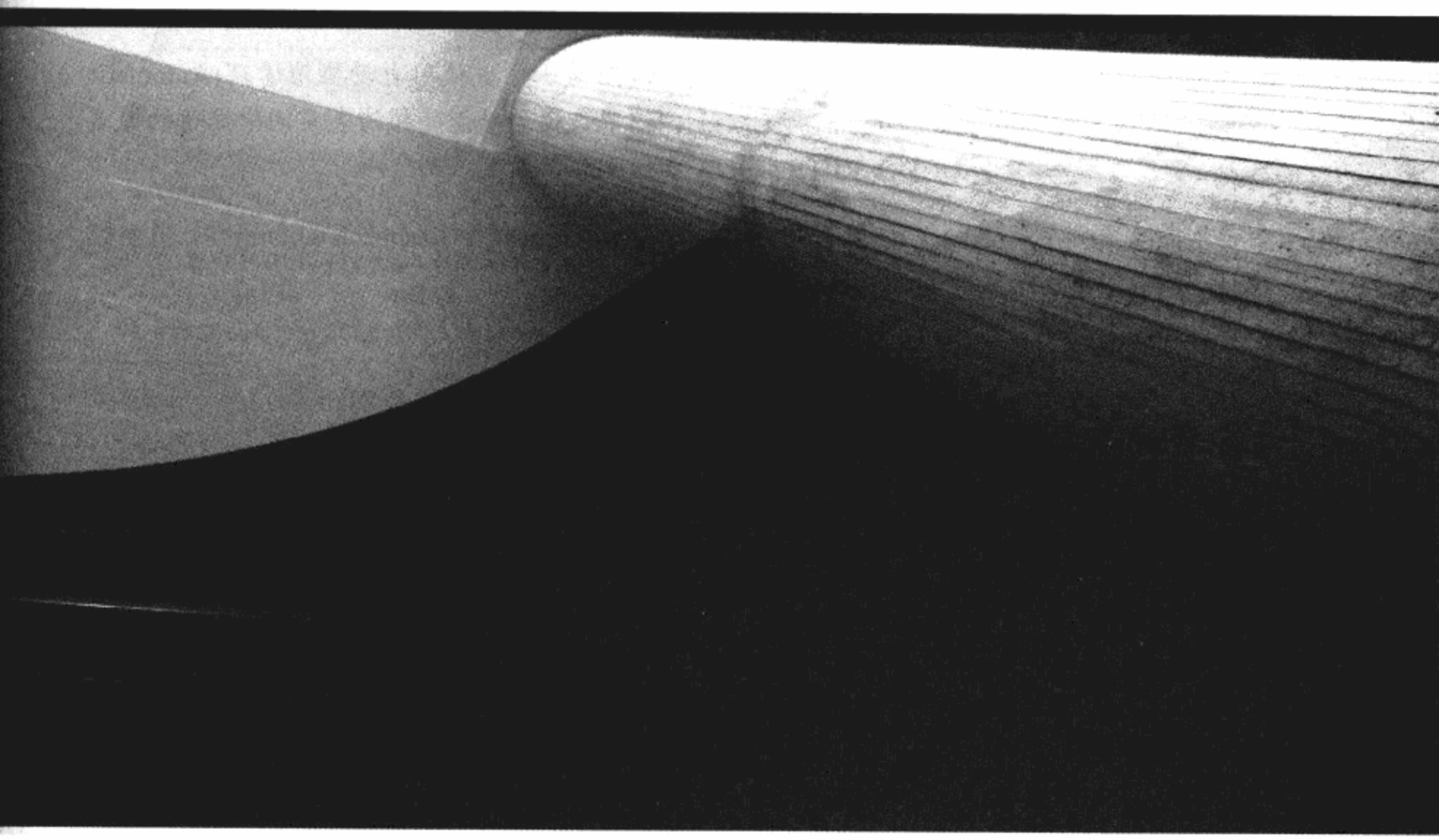
可以使用 FLASHBACK_TRANSACTION_QUERY 数据字典视图来跟踪由特殊事务产生的变化。对于一个给定的事务，FLASHBACK_TRANSACTION_ QUERY 可以显示执行事务的用户的名称、所执行的操作、应用事务的表、开始和结束的 SCN 和时间戳以及用来撤消事务的 SQL。

29.7　闪回计划

对于 DBA 来说，闪回查询可以提供快速执行部分恢复操作的方法。如果数据能通过闪回查询重构并且数据量不大的话，那么可以将多个闪回查询的结果保存在不同的表中。然后，可以通过前几章给出的 SQL 选项(相关的子查询、exists、not exists、minus，等等)来比较表中的数据。如果不能避免使用恢复操作，那么应当确定基于时间的恢复操作所使用的时间段。

在第 30 章将看到，Oracle 支持更多的选项——flashback database 命令和 flashback table 命令。

对于应用程序开发人员和应用程序管理员来说，闪回查询提供了重构数据的一个重要的工具。闪回查询对于测试和支持操作尤其重要。不要把闪回查询作为产品应用程序设计的一部分，而应该将闪回查询作为一种备选方案，以支持在数据受影响之前不能解决的情况。



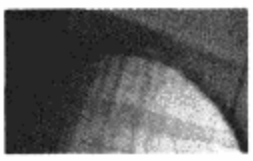
第 30 章

闪回：表和数据库

可以使用 flashback table 命令和 flashback database 命令简化和增强数据恢复工作。flashback table 命令可以自动把整个表恢复至原来的状态。而 flashback database 命令闪回整个数据库，并修改数据库的状态和日志。以下各节将介绍这些选项的详细内容。

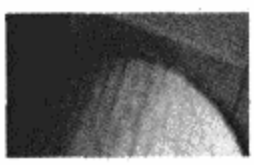
30.1 flashback table 命令

flashback table 命令在人为事件或应用程序出现错误时，将表恢复至原来的状态。但是 Oracle 不能通过任何改变表结构的 DDL 操作来将表恢复至原来的状态。



注意：
为使 flashback table 正常工作，数据库应当使用自动撤消管理(Automatic Undo Management，AUM)。闪回旧数据的功能受撤消表空间中保留的撤消次数和 UNDO_RETENTION 初始化参数设置限制。

不能回滚一个 flashback table 语句。但是可以使用另外一个 flashback table 语句，并且指定一个当前时间之前的时间。



注意：
在使用 flashback table 命令之前，应先记录当前的 SCN。

30.1.1 必需的权限

用户必须具有对表的 FLASHBACK 对象权限或者 FLASHBACK ANY TABLE 系统权限，还必须具有对表的 SELECT、INSERT、DELETE 和 ALTER 对象权限。闪回列表中的所有表都必须启用行移动功能。在将一个表闪回到 drop table 操作之前，只需要有删除表所要求的权限即可。

30.1.2 恢复删除的表

下面看一下 AUTHOR 表：

```
describe AUTHOR
```

```
Name                 Null?     Type
-------------------- --------  ----------------------------
AUTHORNAME           NOT NULL  VARCHAR2(50)
COMMENTS                       VARCHAR2(100)
```

现在删除该表：

```
drop table AUTHOR cascade constraints;

Table dropped.
```

该表该如何恢复呢？默认情况下，被删除的表不会完全消失。该表的块还依然存在于表空间中，并且还占据一定的空间定额。可以通过查询 RECYCLEBIN 数据字典视图来查看被删除的对象。请注意：Object_Name 列的格式因版本不同而不同。

```
select * from RECYCLEBIN;

OBJECT_NAME                    ORIGINAL_NAME                    OPERATION
------------------------------ -------------------------------- ---------
TYPE                           TS_NAME                          CREATETIME
------------------------------ ------------------------------ -------------------
```

```
DROPTIME              DROPSCN PARTITION_NAME                          CAN  CAN
------------------- --------- ------------------------------------- --- ---
   RELATED  BASE_OBJECT   PURGE_OBJECT       SPACE
---------  -----------  ----------- ----------
BIN$yrMKlZaVMhfgNAgAIYowZA==$0 AUTHOR                                  DROP
TABLE                        USERS
2004-02-23:16:10:58
   2004-02-25:14:30:23 720519                                          YES  YES
        48448       48448      48448                  8
BIN$DBo9UChtZSbgQFeMiThArA==$0 SYS_C004828 DROP
INDEX                        USERS
2004-02-23:16:10:58
   2004-02-25:14:30:23 720516                                          NO   YES
        48448       48448      48449                  8
```

RECYCLEBIN 是 USER_RECYCLEBIN 数据字典视图的公有同义词，显示了当前用户的回收站条目。DBA 可以通过 DBA_RECYCLEBIN 数据字典视图查看所有被删除的对象。

如前面的程序清单所示，Oracle 已经删除了 AUTHOR 表和与其相关联的主键索引。虽然它们已经被删除，但是它们还能够用于闪回。请注意回收站程序清单显示了用于删除基本对象的 drop 命令的 SCN。

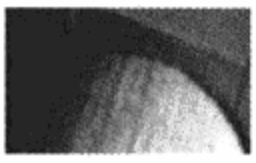
可以使用 flashback table to before drop 命令从回收站中恢复表：

```
flashback table AUTHOR to before drop;

Flashback complete.
```

现在表已经被还原，包括表中的行、索引和统计信息，如下所示：

```
select COUNT(*) from AUTHOR;

  COUNT(*)
----------
        31
```

如果删除 AUTHOR 表，再重新创建它，然后又删除这个表，那么结果将会怎样呢？回收站中将包含两个表。回收站的每个条目都能通过它的 SCN 和删除操作的时间戳来标识。



注意：
flashback table to before drop 命令不能恢复参照性约束。

要从回收站中清除旧的条目，可以使用 purge 命令。可以清除所有被删除的对象、在数据库中被删除的所有对象(如果您是 DBA)、特定表空间中的所有对象或者在特定表空间中特殊用户的所有对象。有关 purge 命令的完整语法和命令选项，请参阅附录 A。

可以使用 flashback table 命令的 rename to 子句重命名闪回的表。

30.1.3 启用和禁用回收站

可以使用 RECYCLEBIN 初始化参数来启用和禁用回收站。默认情况下，此参数设置为 ON。当回收站被禁用时，删除的表及其相关对象并不放在回收站中；而是直接被删除，必须通过其他方法来恢复它们(例如，从备份恢复它们)。

可以在系统级别禁用回收站：

```
alter system set recyclebin = off;
```

也可以在会话级别禁用回收站：

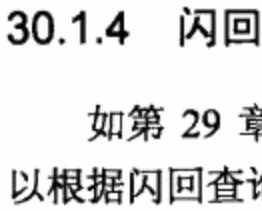
```
alter session set recyclebin = off;
```

通过将相应的值设置为 ON，可以重新启用回收站：

```
alter system set recyclebin = on;
```

下面是如何在会话级别重新启用回收站：

```
alter session set recyclebin = on;
```



注意：
禁用回收站并不会清除或影响回收站中已经存在的对象。

30.1.4 闪回 SCN 或者时间戳

如第 29 章所述，可以在一个单独的表中保存闪回查询的结果并保持主表不受影响。还可以根据闪回查询的结果来更新表的行。可以使用 flashback table 命令将表闪回以前的版本——清除自指定闪回点以来所做的更改。

在 flashback table 操作期间，Oracle 需要在所有指定的表上具有独占的 DML 块。该命令可以在跨越所有表的单个事务中执行；如果其中的任何一个表失败，则整个语句也将失败。可以将表闪回特定的 SCN 或时间戳。

注意：
flashback table to scn 或 to timestamp 不能保存 RowID。

以下的 update 命令试图更新 Clement Hurd 条目的注释。但是，没有使用 where 子句；更新的所有行都具有相同的注释，并且提交该变化。

```
update AUTHOR
    set Comments = 'ILLUSTRATOR OF BOOKS FOR CHILDREN';

31 rows updated.
```

```
commit;
```

在此示例中，几乎整个表都有不正确的数据，根据需要并且已知没有正确执行的事务。为了恢复正确的数据，可以将数据库闪回先前的状态，然后使用任何新命令使数返回当前新状态。

首先，确保如果需要返回当前状态的话，那么已知当前的 SCN：

```
commit;

variable SCN_FLASH number;
execute :SCN_FLASH :=DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER;

PL/SQL procedure successfully completed.

print SCN_FLASH

 SCN_FLASH
----------
    720880
```

现在，将表闪回至更新前的某个时间点的状态。首先，必须启用表的行移动功能：

```
alter table AUTHOR enable row movement;

Table altered.
```

然后就可以闪回表了：

```
flashback table AUTHOR to timestamp (systimestamp - 5/1440);
```

如果希望指定 SCN 而不是时间戳，那么可以使用 to scn 子句。

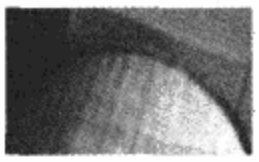
30.1.5　索引和统计信息

当闪回表时，不闪回其统计信息。存在于表中的索引将被还原，并能够反映表在闪回点时的状态。闪回点以后删除的索引将不能还原。而闪回点之后创建的索引将继续存在并且将被更新，以反映更旧的数据。

当从回收站恢复时，索引将保留它们在回收站中的名称。通过 alter index … rename to … 命令可以恢复这些索引原来的名称。

30.2　flashback database 命令

flashback database 命令将数据库返回至过去某个时间的状态或者 SCN，提供了另外一种快速的执行不完整数据库恢复的方法。在 flashback database 操作之后，为了对闪回数据库进行写访问，必须用 alter database open resetlogs 命令重新打开它。用户必须具有 SYSDBA 系统权限，才能使用 flashback database 命令。



注意：
必须通过 alter database flashback on 命令将数据库置于闪回模式。在执行此命令时，必须以独占的模式安装数据库，但不打开它。

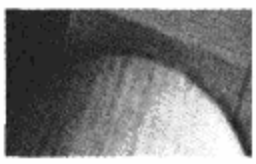
flashback database 命令的语法如下：

```
flashback [standby] database [database]
    { to {scn | timestamp} expr
    | to before {scn | timestamp } expr
    }
```

可以使用 to scn 或者 to timestamp 子句来设置整个数据库应闪回的状态点。可以闪回一个关键点之前的某个状态(如为多个表并产生随意结果的一个事务)。可以使用 ORA_ROWSCN 伪列来查看最近作用于行的事务的 SCN。

为使用 flashback database，首先必须在安装了数据库但没有打开的情况下，对数据库进行更改。如果还没有这么做，则需要关闭数据库，然后在启动数据库的过程中启用闪回功能：

```
startup mount exclusive;
alter database archivelog;
alter database flashback on;
alter database open;
```



注意：
在执行 alter database flashback on 命令之前，必须通过 alter database archivelog 命令启用介质恢复。

两个初始化参数设置控制闪回数据保留在数据库中的数量。DB_FLASHBACK_RETENTION_TARGET 初始化参数设置了数据库可以闪回的时间上限(以分钟表示)。DB_RECOVERY_FILE_DEST 初始化参数设置了闪回恢复区域的大小。请注意：flashback table 命令使用撤消表空间，而 flashback database 命令依赖于存储在闪回恢复区域中的闪回日志。

闪回恢复区域可包含控制文件、联机重做日志、归档重做日志、闪回日志和 RMAN 备份。恢复区域中的文件或者是永久文件或者是临时文件。永久文件是由数据库实例使用的活动文件。除永久文件之外的所有文件都是临时文件。一般而言，对于跟不上备份保留策略变化的临时文件，或者是已经备份到磁带的临时文件，Oracle Database 最终会删除它们。

表 30-1 描述了恢复区域中的文件，以及每个文件的分类是永久的还是临时的，另外还描述了如何影响数据库的可用性。

表 30-1 闪回恢复区域中文件的类型

文　件	类　型	闪回恢复区域不可访问时的数据库行为
当前控制文件的多元复用副本	永久文件	如果数据库不能写到存储在闪回恢复区域中的控制文件的多元复用副本，则实例会失败。即使可访问的多元复用副本位于恢复区域的外面，也会失败

(续表)

文　件	类　型	闪回恢复区域不可访问时的数据库行为
联机重做日志文件	永久文件	如果联机重做日志的镜像副本存在于闪回恢复区域之外的可访问位置，则实例可用性不受影响；否则，实例失败
归档重做日志文件	临时文件	如果日志被归档到闪回恢复区域之外的可访问位置，则实例可用性不受影响；否则，数据库最终会停止，因为它不能归档联机重做日志
外部归档重做日志文件	临时文件	实例可用性不受影响
数据文件和控制文件的镜像副本	临时文件	实例可用性不受影响
备份块	临时文件	实例可用性不受影响
闪回日志	临时文件	实例可用性不受影响，因为数据库自动禁用 flashback database、自动将消息写入报警日志、并继续处理数据库

可以查询 V$FLASHBACK_DATABASE_LOG 视图来确定可以闪回数据库的程度。保留在数据库中的闪回数据的数量由初始化参数和闪回恢复区域的大小所控制。下面的程序清单显示了 V$FLASHBACK_DATABASE_LOG 中可用的列和示例内容：

```
desc V$FLASHBACK_DATABASE_LOG
```

```
Name                                      Null?    Type
----------------------------------------- -------- -------
OLDEST_FLASHBACK_SCN                               NUMBER
OLDEST_FLASHBACK_TIME                              DATE
RETENTION_TARGET                                   NUMBER
FLASHBACK_SIZE                                     NUMBER
ESTIMATED_FLASHBACK_SIZE                           NUMBER

select * from V$FLASHBACK_DATABASE_LOG;

OLDEST_FLASHBACK_SCN OLDEST_FL RETENTION_TARGET FLASHBACK_SIZE
-------------------- --------- ---------------- --------------
ESTIMATED_FLASHBACK_SIZE
------------------------
              722689  25-FEB-04             1440           8192000
                           0
```

可以查询 V$DATABASE 来验证数据库的闪回状态；如果数据库已经启用了闪回功能，Flashback_On 列就会有 YES 值。

```
select Current_SCN, Flashback_On from V$DATABASE;

CURRENT_SCN       FLA
```

```
---------------- ---
          723649 YES
```

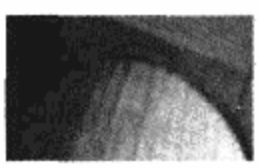
将数据库打开一个小时以后，验证闪回数据是可用的，然后将数据闪回——您将丢失发生在那段时间内的所有事务：

```
shutdown;
startup mount exclusive;
flashback database to timestamp sysdate-1/24;
```

请注意 flashback database 命令要求数据库以独占模式加载，该模式将影响它参与任何 RAC 群集的能力(参阅第 50 章)。

执行 flashback database 命令时，Oracle 检查并确保所需的全部归档重做日志文件和联机重做日志文件是可用的。如果日志可用，则联机数据文件将被还原至指定的时间或者 SCN 的状态。

如果在存档日志和闪回区域中没有足够的联机数据，则必须使用传统的数据库恢复方法来恢复数据。比如，可能需要在回滚数据之前使用文件系统恢复方法。



注意：
有些数据库操作，如删除表空间或缩小数据文件，不能用 flashback database 还原。在这种情况下，闪回数据库窗口紧接着此操作立即启动。

一旦完成了闪回，就必须通过 resetlogs 选项来打开数据库，以便有权对数据库进行写访问：

```
alter database open resetlogs;
```

为了关闭闪回数据库选项，可以在安装数据库被装配而没有打开时，执行 alter database flashback off 命令：

```
startup mount exclusive;
alter database flashback off;
alter database open;
```

如本章和第 29 章所述，可以使用闪回选项来执行一组操作——恢复旧数据、还原表中原来的数据、维护行的修改历史以及快速恢复整个数据库。如果数据库的配置支持 Automatic Undo Management，那么所有这些操作都将极大地简化。而且请注意：flashback database 命令需要修改数据库的状态。虽然这些要求给 DBA 增加了额外的负担，但需要的恢复次数明显减少，同时完成这些恢复的速度大大提高。

第31章

SQL 重 放

在 Oracle Database 11g 中，可以使用“重放”功能捕获在数据库中执行的命令，然后在其他地方重放。可以在一个数据库中捕获这些操作，并在另一个数据库中重放。可以使用捕获和重放功能执行诊断操作，或在产品环境下测试应用程序。

当评估变更对多用户打包的应用程序的影响时，如果无权直接访问此应用程序的所有代码，则重放功能非常有用。无论命令源自何处，Oracle 都将捕获在数据库中执行的命令。捕获行为独立于原始程序。

31.1 高级别配置

重放由以下进程和组件组成。

源数据库：

- 工作负载捕获进程记录在数据库中执行的 SQL

目标数据库：

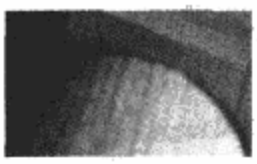
- 工作负载预处理步骤准备要执行的日志
- 重放客户在数据库中创建会话
- 工作负载重放进程通过重放客户在新数据库中执行 SQL

当工作负载在目标数据库中重放时，可以分析数据库，并生成性能调整建议。下面几节将介绍如何执行工作负载捕获进程和重放。但首先需要确保系统配置支持这些进程，如 30.1.1 节所描述。

31.1.1 分离和连接

开始之前，应该配置目标数据库(捕获的重放将在这里执行)，尽量减少它与外部文件和对象的交互。在源数据库中，开始捕获之前，应该仔细检查数据库链接和外部文件的使用情况。在重放期间，Oracle 将在目标数据库中执行从源数据库捕获的命令。如果目标数据库与源数据库有相同的链接和访问权限，则在目标数据库中可以通过事务的重复来操纵远程对象。

如果远程数据库只用于数据查找，这也许就没有什么问题。如果远程数据库被源数据库更新，则在目标数据库重放期间重复这些事务可能会在远程数据库中生成不希望的事务。例如，如果源数据库包含一个在测试期间执行刷新的物化视图，且在重放执行期间，从复制的测试数据库执行相同的命令，则极有可能产生不希望的结果。对重放环境最好的控制方法是分离它。尽可能地避免使用目录对象、URL、UTL_FILE 和生成电子邮件的程序。



注意：

即使是重放期间最好的结果，可能也需要改变目标系统上的系统时间。因此，重放最适用于目标系统是一个测试环境的情况，而不适合于在测试环境中生成重放，然后在产品数据库中执行的情况。

在源数据库上开始捕获之前，应该确保数据库中没有任何活动事务。如果您具有 DBA 权限，则可以终止任何打开的活动会话，从而完成这一任务；另一种方法是，可以关闭并重新启动数据库。然后应该像数据存在于源数据库中一样捕获数据(通过备份方法，如 Data Pump Export/Import 或 RMAN)。当在目标系统上重放工作负载时，实际上希望数据环境与源数据库中的环境一样。

为了进一步控制源系统中允许的事务，可以选择以受限模式启动它。因为这会限制工作负载捕获期间连接到数据库的会话，所以需要确定这样的受限行为是否准确描述了应用程序的标准用法。

31.1.2 创建工作负载目录

SQL 工作负载的日志文件将写到源服务器上的目录中。为了指定此目录的物理位置，必须在数据库中创建一个目录对象(是一个指向物理目录的逻辑指针)。如下所示，使用 create directory 命令创建一个目录对象：

```
create directory workload_dir as '/u01/workload';
```

现在可以将 workload_dir 目录用作 SQL 日志文件的目标目录。确保指向的文件系统有可用空间，且其空间使用情况受到动态监视。

31.2　捕获工作负载

可以通过 Oracle 企业管理器(Oracle Enterprise Manager，OEM)控制台或存储过程调用来执行创建目录和捕获工作负载所需要的所有命令。在本章的示例中，您将看到执行命令所需要的命令行条目，因为并非所有用户都有权访问 OEM。

工作负载捕获进程通过 DBMS_WORKLOAD_CAPTURE 程序包来控制。使用 DBMS_WORKLOAD_CAPTURE 程序包可以控制用于工作负载的过滤器、启动捕获、停止捕获和导出捕获的自动工作负载存储库(Automated Workload Repository，AWR)数据。

DBMS_WORKLOAD_CAPTURE 程序包中的函数和过程如表 31-1 所示。

表 31-1　DBMS_WORKLOAD_CAPTURE 程序包中的函数和过程

子 程 序	描　述
ADD_FILTER	添加一个过滤器
DELETE_CAPTURE_INFO	删除 DBA_WORKLOAD_CAPTURES 视图和 DBA_WORKLOAD_FILTERS 视图中的行
DELETE_FILTER	删除指定的过滤器
EXPORT_AWR	导出与指定捕获的 ID 相关的 AWR 快照
FINISH_CAPTURE	结束工作负载捕获或将数据库恢复为正常的操作模式
GET_CAPTURE_INFO	检索工作负载捕获的所有信息，将信息导入到 DBA_WORKLOAD_CAPTURES 视图和 DBA_WORKLOAD_FILTERS 视图中，并返回合适的 DBA_WORKLOAD_CAPTURES.ID
IMPORT_AWR	导入与指定捕获的 ID 相关的 AWR 快照
REPORT	根据捕获的工作负载生成报表
START_CAPTURE	启动捕获进程

31.2.1　定义过滤器

默认情况下，捕获进程捕获数据库中的所有活动。过滤器允许捕获工作负载的指定子集。要添加过滤器，使用 ADD_FILTER 进程。

ADD_FILTER 进程的格式如下所示：

```
DBMS_WORKLOAD_CAPTURE.ADD_FILTER (
    fname          IN  VARCHAR2 NOT NULL,
    fattribute     IN  VARCHAR2 NOT NULL,
    fvalue         IN  VARCHAR2 NOT NULL);
```

fname 变量是过滤器的名称。fattribute 变量指定需要应用到过滤器的属性。fattribute 的可能值是 INSTANCE_NUMBER、USER、MODULE、ACTION、PROGRAM 和 SERVICE。然后可以通过 fvalue 变量传递属性值，如下所示：

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.ADD_FILTER (
                fname => 'user_practice',
                fattribute => 'USER',
                fvalue => 'PRACTICE');
END;
/
```

通过将过滤器名传递给 DELETE_FILTER 进程可以删除过滤器。DELETE_FILTER 进程的语法如下所示：

```
DBMS_WORKLOAD_CAPTURE.DELETE_FILTER (
   filter_name IN VARCHAR2(40) NOT NULL);
```

当启动捕获时，告诉 Oracle 如何使用过滤器。要么将捕获限制为只满足过滤器标准的值，要么捕获不满足过滤器标准的所有值。

31.2.2 启动捕获

要启动捕获，可以执行 START_CAPTURE 进程。如果还没有指定一个过滤器，则捕获适用于在数据库中执行的所有命令。

START_CAPTURE 的语法如下所示：

```
DBMS_WORKLOAD_CAPTURE.START_CAPTURE (
  name              IN VARCHAR2,
  dir               IN VARCHAR2,
  duration          IN NUMBER DEFAULT NULL,
  default_action    IN VARCHAR2 DEFAULT 'INCLUDE',
  auto_unrestrict   IN BOOLEAN DEFAULT TRUE);
```

START_CAPTURE 的输入参数如下所示：

- name 参数是分配给工作负载捕获的名称。
- dir 是创建的目录对象的名称，此目录对象用来存储工作负载捕获的文件
- 可选的 duration 参数允许指定捕获的持续时间，单位是秒。默认情况下，duration 值设置为 NULL，且在通过 FINISH_CAPTURE 过程调用明确结束捕获之前，捕获会一直运行。
- default_action 参数告诉捕获进程如何使用过滤器。如果该参数设置为 INCLUDE(默认值)，则除了过滤器中指定的那些命令外，所有命令都将被捕获。如果该参数设置为 EXCLUDE，则只捕获通过过滤器的那些命令。
- 当 auto_restrict 参数设置为 TRUE(默认值)时，告诉 Oracle 成功开始捕获时以非受限模式启动数据库。

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.START_CAPTURE
```

```
        (name => 'practice_capture',
           dir => 'workload_dir',
   default_action => 'EXCLUDE');
END;
/
```

然后 Oracle 开始捕获在数据库中执行的工作负载数据。

通过执行 GET_CAPTURE_INFO 函数，可以确定捕获的哪一个 ID 被分配给捕获，如下所示：

```
DBMS_WORKLOAD_CAPTURE.GET_CAPTURE_INFO
  dir    IN    VARCHAR2)
RETURN  NUMBER;
```

为了在捕获结束时导出 AWR 统计信息，需要知道捕获的 ID。

31.2.3　停止捕获

如果没有指定持续时间，则必须以手工方式停止捕获。要停止捕获，需要执行 FINISH_CAPTURE 过程，如下所示：

```
BEGIN
  DBMS_WORKLOAD_CAPTURE.FINISH_CAPTURE ();
END;
/
```

31.2.4　导出 AWR 数据

为了导出捕获的 AWR 数据(以便以后分析时使用)，需要执行 EXPORT_AWR 过程：

```
DBMS_WORKLOAD_CAPTURE.EXPORT_AWR (
    capture_id IN NUMBER);
```

31.3　处理工作负载

捕获工作负载之后，就需要将捕获的所有工作负载日志移动到目标系统中。如前所述，目标数据库应该与产品数据库分开，而且应该尽可能与产品数据库访问的外部结构分离。目标数据库应该与源数据库运行相同的 Oracle 版本。

在预处理步骤中，将工作负载日志传送到重放文件中，并创建元数据。在重放捕获的工作负载之前，必须针对其中的每个负载执行预处理步骤。如果目标数据库是一个 RAC 集群，则可以将集群的一个实例用于重放预处理。

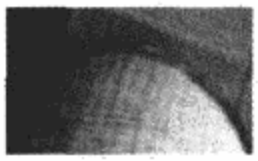
可以通过 OEM 或通过 DBMS_WORKLOAD_REPLAY 程序包启动预处理步骤，如下面的程序清单所示：

```
BEGIN
  DBMS_WORKLOAD_REPLAY.PROCESS_CAPTURE
    (capture_dir => 'workload_dir');
END;
/
```

capture_dir 变量值必须是数据库中目录对象的名称，它指向存储工作负载日志的物理目录。预处理步骤完成之后，就可以按照 31.4 小节的描述，在测试数据库上重放工作负载。

31.4 重放工作负载

已被预处理过的工作负载文件可以针对目标数据库重放。本章前面曾提到过，目标数据库应包含源数据库对象和源数据的一份副本。在可能的地方，目标数据库应该是开始捕获前产品系统的一个副本。目标数据库应该隔离，无权访问产品数据库使用的外部对象(例如，外部文件、数据库链接或远程物化视图的源表)。需要注意的是，在目标数据库中的事务重放不会在其他数据库中生成无效的数据。



注意：
因为工作负载重放的某些方面可能依赖于时间(例如，执行基于时间调度的作业，或使用 SYSDATE 函数)，所以建议在开始重放之前重置测试服务器上的系统时间。

目标数据库必须有一个目录对象，它指向存储工作负载文件的物理目录。如果目标数据库用于预处理步骤，则重放可以使用这一相同的目录。如果这样的目录不存在，则使用 create directory 命令创建一个目录。

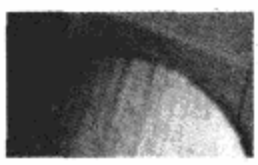
重放执行时，试图连接到产品数据库。在目标服务器上，必须将这些连接重定向到目标数据库。例如，如果产品服务器被 PROD 服务名访问，则在目标服务器上，PROD 服务名必须指向目标数据库。

31.4.1 控制和启动重放客户

重放由重放客户执行。重放客户是一个多线程程序(其名称是 wrc)，它的每一个线程都提交一个来自捕获的会话的工作负载。在重放开始之前，必须将重放客户连接到目标数据库。必须在目标数据库中为 DBA 权限的账户(除 SYS 外)提供用户名和密码。重放客户必须有权访问存储处理过的工作负载文件的目录。

重放客户可执行程序 wrc.exe 位于 Oracle 软件主目录下的/bin 子目录中。wrc 的语法如下：

```
wrc [user/password[@server]] MODE=[value] [keyword=[value]]
```



注意：
如果输入无参数的 wrc，则 Oracle 将显示每个参数可用选项的完整列表。

user 和 password 参数告诉工作负载客户如何连接到将重放工作负载的目标数据库。server 参数指定运行重放客户的服务器名。mode 参数告诉 Oracle 如何运行工作负载，有效的值是 replay(默认值)、calibrate 和 list_hosts。keyword 参数告诉 Oracle 如何配置客户。

因为一个的重放客户可以启动多个数据库会话，所以不需要每个会话启动一个客户。如果 mode 参数使用 calibrate 选项，如下所示，则 Oracle 会提供要启动的客户数量的建议：

```
wrc mode=calibrate replaydir=/u01/workload
```

replaydir 参数指定处理过的重放文件所在的物理目录。wrc 输出将建议要启动的客户数量以及最多可涉及的并发会话的数量。

要启动重放客户，使用 mode=replay 选项，如下所示：

```
wrc dbacct/pass@target mode=replay replaydir=/u01/workload
```

工作负载重放时，重放客户仍然运行。重放完成时，重放客户本身终止。

31.4.2　初始化和运行重放

处理了工作负载数据和配置了重放客户之后，现在可以通过 OEM 或 DBMS_WORKLOAD_REPLAY 程序包进行重放了。DBMS_WORKLOAD_REPLAY 程序包可用的过程和函数如表 31-2 所示。

表 31-2　DBMS_WORKLOAD_REPLAY 程序包中的函数和过程

子 程 序	描　述
CALIBRATE	在处理过的工作负载捕获目录上操作，估计重放所需要的主机和工作负载重放客户的数量
CANCEL_REPLAY	取消正在进行的重放
DELETE_REPLAY_INFO	删除 DBA_WORKLOAD_REPLAY 程序包中与指定的工作负载重放 ID 相对应的行
EXPORT_AWR	导出与指定的重放 ID 相关联的 AWR 快照
GET_REPLAY_INFO	检索工作负载捕获的有关信息和重放的历史信息
IMPORT_AWR	导入与指定的重放 ID 相关联的 AWR 快照
INITIALIZE_REPLAY	初始化重放
PREPARE_REPLAY	将 RDBMS 置于特定的“准备”模式
PROCESS_CAPTURE	处理工作负载捕获
REMAP_CONNECTION	将捕获的连接重新映射到新的连接，以便在工作负载重放期间用户会话可以以期望的方式连接到数据库
REPORT	生成关于重放的报表
START_REPLAY	启动重放

初始化步骤将元数据加载到测试系统，并为启动重放准备好数据库。INITIALIZE_REPLAY 的语法如下所示：

```
DBMS_WORKLOAD_REPLAY.INITIALIZE_REPLAY (
    replay_name   IN VARCHAR2,
    replay_dir    IN VARCHAR2);
```

replay_dir 参数是目标数据库中目录对象的名称，此目录对象指向存储处理过的工作负载文件的物理目录。

在捕获的工作负载中使用的连接字符串可能需要重新映射。当工作负载 SQL 试图连接到外部数据库时(通过捕获的工作负载中的连接字符串)，要么这些远程数据库必须可用，从而

使会话连接到默认主机，要么使用 REMAP_CONNECTION 过程指定连接应该如何重定向。前面曾提到，在理想情况下，您应该工作在一个独立的环境中，默认方法是这些连接默认连接到目标主机。

工作负载初始化和重放客户准备好之后，现在可以准备启动重放了。为了准备重放，需要使用 PREPART_REPLAY 进程：

```
DBMS_WORKLOAD_REPLAY.PREPARE_REPLAY (
  synchronization           IN BOOLEAN DEFAULT TRUE,
  connect_time_scale        IN NUMBER DEFAULT 100,
  think_time_scale          IN NUMBER DEFAULT 100,
  think_time_auto_correct   IN BOOLEAN DEFAULT TRUE);
```

当 synchronization 参数设置为 TRUE 时，在重放中保存提交顺序。connect_time_scale 和 think_time_scale 是在工作负载重放期间用来增加或减少并发的用户数量的百分比。当完成用户调用在重放期间所花的时间比初始化捕获期间所花的时间长时，think_time_auto_correct 参数自动纠正调用之间的“判断时间(think time)”。当目标系统性能与源系统性能不匹配时，自动纠正是必需的。

如果 connect_time_scale 的值是 10，则重放期间会话连接之间的间隔会减少到原始值的 10%。如果 think_time_scale 参数设置为 20，则重放期间操作之间的“判断时间”会减少到原始值的 20%。如果 think_time_scale 参数设置为 200%，则操作之间的“判断时间”在重放期间会翻倍。

要开始重放，使用 START_REPLAY 进程，如下面的程序清单所示：

```
BEGIN
  DBMS_WORKLOAD_REPLAY.START_REPLAY ();
END;
/
```

要停止重放，使用 CANCEL_REPLAY 进程，如下面的程序清单所示：

```
BEGIN
  DBMS_WORKLOAD_REPLAY.CANCEL_REPLAY ();
END;
/
```

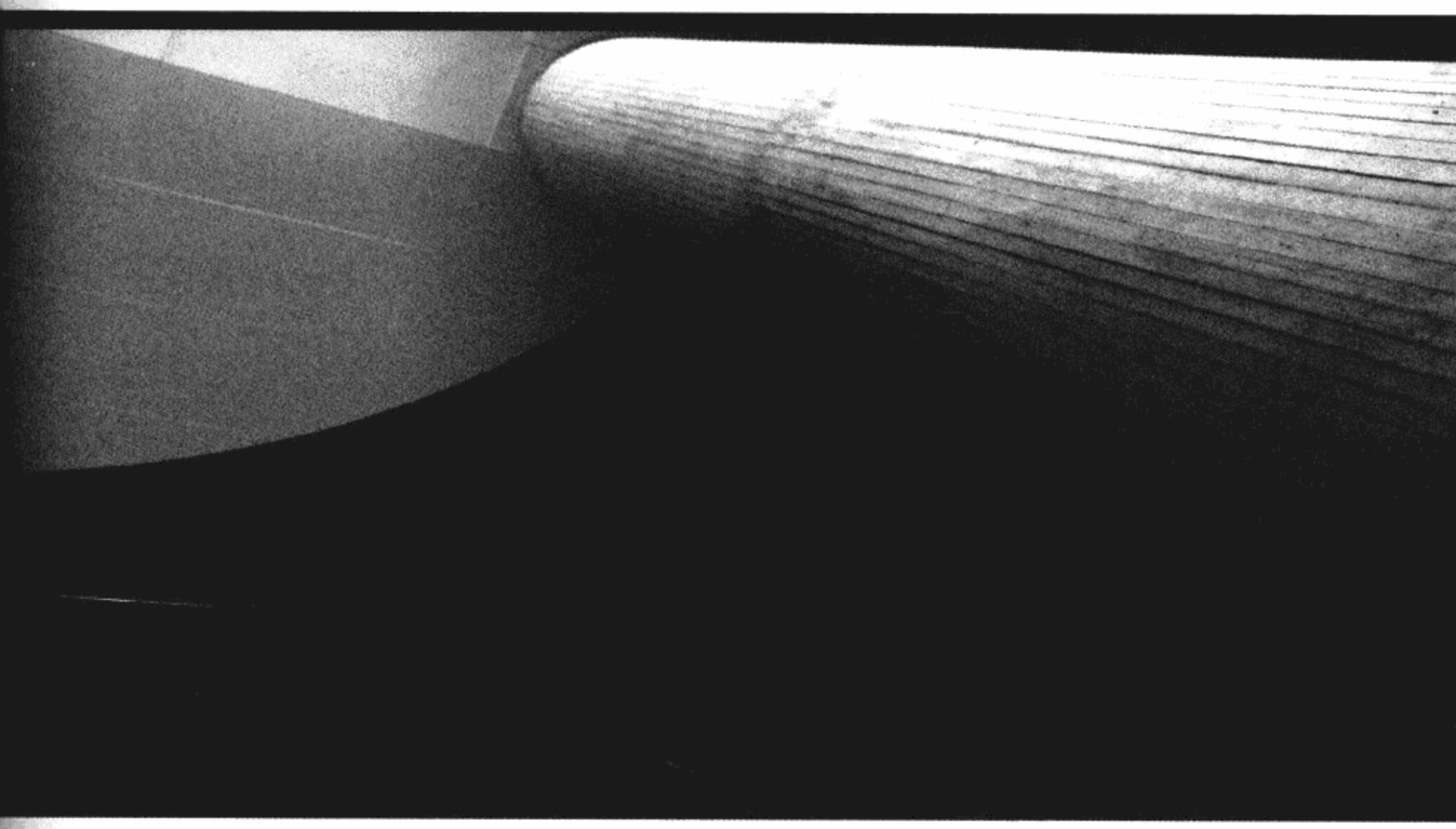
31.4.3 导出 AWR 数据

通过 EXPORT_AWR 过程可以导出重放的 AWR 数据。它的唯一参数是 replay_id，如下面的程序清单所示：

```
BEGIN
  DBMS_WORKLOAD_REPLAY.EXPORT_AWR (replay_id => 1);
END;
/
```

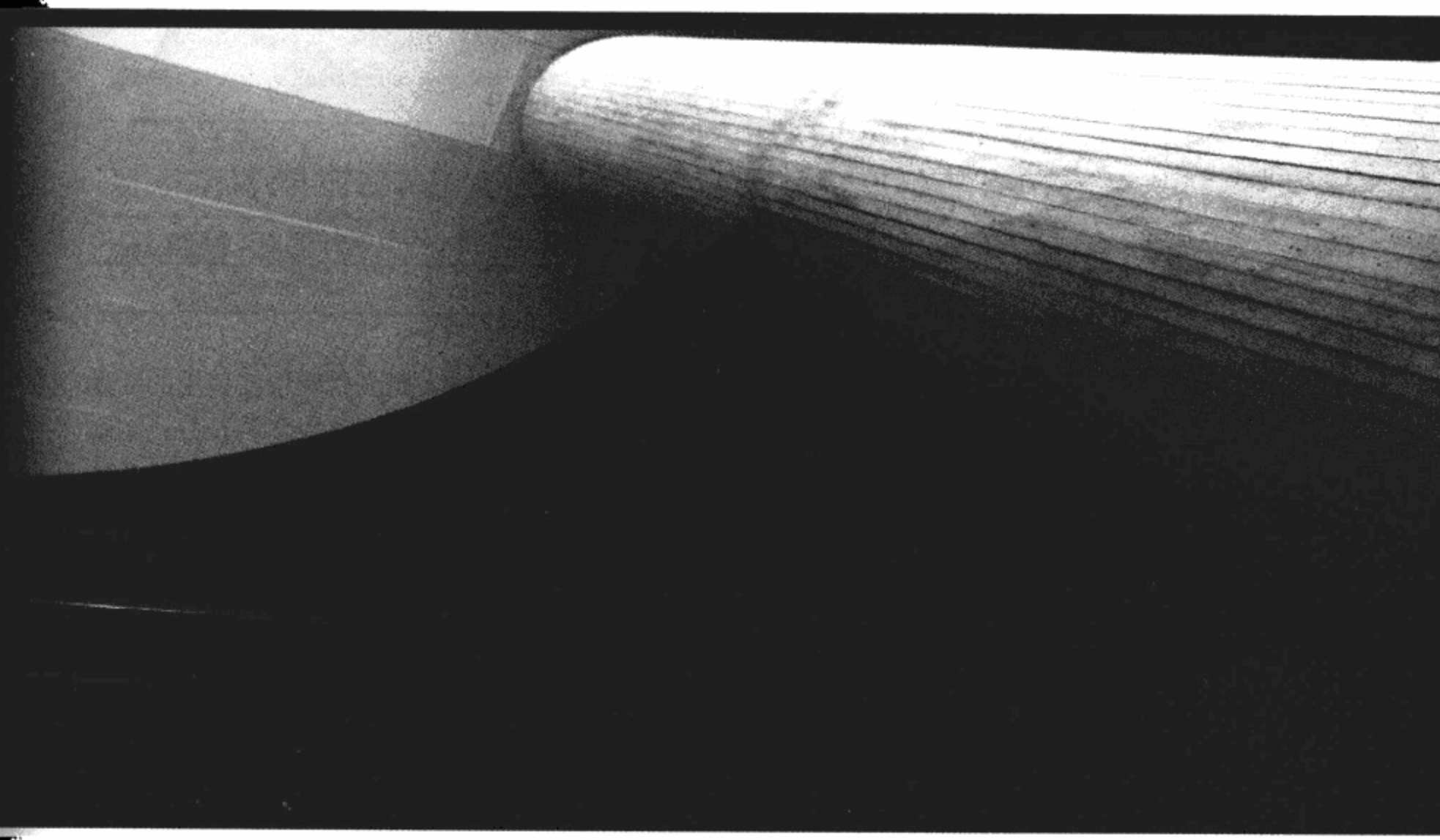
replay_id 为 1 的任何 AWR 快照都将被导出。然后可以将重放执行的 AWR 报告与产品环境中相同工作负载的 AWR 报告进行比较，以便评估数据库之间的性能差异和执行行为变化。

重放可以执行多次，只需要先在测试数据库中重新创建对象并每次重新设置系统时间即可。因此可以使用重放工具来确定在测试环境中参数变化的影响，然后再具体运用到产品环境中。



第Ⅳ部分

PL/SQL



第32章

PL/SQL 简 介

PL/SQL 是 Oracle 对结构化查询语言(SQL)的过程语言(PL)的扩展。可以用 PL/SQL 完成如下操作，如通过创建存储过程和程序包编写业务规则的代码、触发数据库事件以及给 SQL 命令的执行添加程序逻辑等。

创建触发器、存储过程和程序包所涉及的步骤将在本部分的后面章节中讨论。本章将介绍 PL/SQL 的基本结构和语法。

32.1 PL/SQL 概述

PL/SQL 代码包含在称之为块的结构中。如果要创建一个存储过程或程序包，应当给 PL/SQL 代码块起一个名字；如果没有给 PL/SQL 代码块起名字，该代码块就被称为匿名块。本章的示例反映了匿名块的特性；第 IV 部分其余的章节将说明命名块的创建过程。

PL/SQL 代码块包含 3 部分如表 32-1 所示。

表 32-1 PL/SQL 代码块的 3 部分

组 成 部 分	说 明
声明(Declaration)	定义并初始化块中使用的变量和游标
可执行命令(Executable Command)	用流控制命令(如 if 命令和循环)执行命令并给声明的变量赋值
异常处理(Exception Handling)	提供对错误情况的定制处理

在 PL/SQL 代码块中，第一部分是声明部分。在声明部分中，定义代码块要使用的变量和游标。声明部分以关键字 declare 开头，并在可执行命令部分开始时(以关键字 begin 指出)结束。可执行命令部分后面紧接着是异常处理部分；关键字 exception 标志着异常处理部分的开始。PL/SQL 块以 end 关键字结束。

典型的 PL/SQL 块结构如下所示：

```
declare
  <declarations section>
begin
  <executable commands>
exception
  <exception handling>
end;
```

本章后续部分将说明 PL/SQL 块的每一部分。

32.2 声明部分

声明部分是 PL/SQL 块的开始。声明部分以 declare 关键字开始，后面是变量和游标的定义列表。用户可以定义具有常量值的变量，并且变量能够继承已存在的列和查询结果中的数据类型，如下面的示例所示。

下面的程序清单计算一个圆的面积，结果存储到一个名为 AREAS 的表中。AREAS 表有两列，分别用于存储半径和面积。计算圆面积的公式为半径的平方乘以常量 π。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  radius := 3;
  area := pi*power(radius,2);
  insert into AREAS values (radius, area);
end;
/
```

“end;”行是 PL/SQL 块的结尾标志；根据所使用的工具，可能需要添加“/”来执行

PL/SQL 块。执行 PL/SQL 块时，将会从 Oracle 得到以下响应信息：

```
PL/SQL procedure successfully completed.
```

为了验证 PL/SQL 块是否正确执行，可以从数据库中选择由 PL/SQL 代码插入的行。下面的程序清单中的查询和结果显示了在 AREAS 表中 PL/SQL 块创建的行：

```
select *
  from AREAS;

    RADIUS         AREA
---------- ----------
         3        28.27
```

输出结果表明 PL/SQL 块在 AREAS 表中插入了一行。此程序只创建了一行，因为只给出了一个 Radius 值。

在下面的程序清单中，PL/SQL 块的第一部分将声明 3 个变量。在 PL/SQL 块的可执行命令部分使用的变量必须事先声明。

```
declare
  pi      constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area    NUMBER(14,2);
```

所声明的第一个变量是 π，它通过 constant 关键字设置为一个常量值。由于该变量声明为 constant，因此变量 π 的值在可执行命令部分不能改变。该值通过“:=”运算符赋值：

```
pi  constant  NUMBER(9, 7) := 3.1415927;
```

还可以通过 default 关键字赋予常量值：

```
pi  constant  NUMBER(9, 7) DEFAULT 3.1415927;
```

接下来定义了两个变量，但未赋予默认值：

```
radius INTEGER(5);
 area NUMBER(14,2);
```

可以在声明部分给变量赋一个初始值。要给变量设置初始值，只要在说明数据类型之后赋予相应的值即可，如下面的程序清单所示：

```
radius INTEGER(5) := 3;
```

在本例中，数据类型包括 NUMBER 和 INTEGER。PL/SQL 的数据类型包括所有有效的 SQL 数据类型和基于查询结构的复杂的数据类型。表 32-2 列出了 PL/SQL 支持的数据类型。

表 32-1 PL/SQL 数据类型

标量数据类型			
BINARY_DOUBLE	FLOAT	NUMBER	SMALLINT
BINARY_FLOAT	INT	NUMERIC	STRING
BINARY_INTEGER	INTEGER	NVARCHAR2	TIMESTAMP
BOOLEAN	INTERVAL DAY TO SECOND	PLS_INTEGER	TIMESTAMP WITH LOCALTIME ZONE
CHAR	INTERVAL YEAR TO MONTH	POSITIVE	TIMESTAMP WITH TIME ZONE
CHARACTER	LONG	POSITIVEN	UROWID
DATE	LONG RAW	RAW	VARCHAR
DEC	NATURAL	REAL	VARCHAR2
DECIMAL	NATURALN	ROWID	
DOUBLE PRECISION	NCHAR	SIGNTYPE	
组合类型			
RECORD	TABLE	VARRAY	
引用类型			
REF CURSOR	REF *object_type*		
LOB 类型			
BFILE	BLOB	CLOB	NCLOB

下面的示例声明一个游标来检索 RADIUS_VALS 表中的记录。RADIUS_VALS 表只有一列——Radius 列，它用来保存这些示例使用的半径。该游标在声明部分声明，rad_val 变量被声明为基于游标结果的数据类型。

注意：
在本示例中，RADIUS_VALS 表中只包含一行，此行的 Radius 值为 3。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
     select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
  fetch rad_cursor into rad_val;
```

```
    area := pi*power(rad_val.radius,2);
    insert into AREAS values (rad_val.radius, area);
  close rad_cursor;
end;
/
```

在声明部分的开始，定义了 π 和 area 两个变量，就像在本章前面的示例中所定义的那样。没有定义 radius 变量，而是定义了一个名为 rad_cursor 的游标。该游标的定义包括一个游标名称(rad_cursor)和一个查询(select * from RADIUS_VALS)。游标保存了在 PL/SQL 块中其他命令执行的查询结果。

```
declare
  pi      constant NUMBER(9,7) := 3.1415927;
  area    NUMBER(14,2);
  cursor rad_cursor is
      select * from RADIUS_VALS;
```

最后的声明创建一个变量，并通过游标的结果集设定其结构。

```
rad_val rad_cursor%ROWTYPE;
```

rad_val 变量能够引用该查询的结果集的每一列。在此示例中，查询只返回一列，但如果该表包含多列，则可以通过 rad_val 变量引用它们。

除了%ROWTYPE 声明外，还可以用%TYPE 声明继承数据类型信息。如果使用%ROWTYPE 声明，那么变量将使游标的结果集中的所有列都继承该列和相应的数据类型信息。如果使用%TYPE 声明，则变量只能继承用于定义它的列的定义。甚至能够在游标中使用%TYPE 定义，如下所示：

```
cursor rad_cursor is
       select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
  rad_val_radius rad_val.Radius%TYPE;
```

在以上程序清单中，变量 rad_val 继承了游标 rad_cursor 的结果集的数据类型。变量 rad_val_radius 继承了变量 rad_val 中 Radius 列的数据类型。

采用%ROWTYPE 定义和%TYPE 定义设定数据类型的优点是，它可以使 PL/SQL 代码中数据类型的定义与基础数据结构无关。如果 RADIUS_VALS 表的 Radius 列从 NUMBER(5)数据类型变为 NUMBER(4，2)类型，则不需要修改 PL/SQL 代码；相关变量的数据类型在运行时动态分配。

32.3　可执行命令部分

可执行命令部分处理在 PL/SQL 块的声明部分中声明的变量和游标。可执行命令部分总是以关键字 begin 开头。下面的程序清单重复了 32.2 节的第一个 PL/SQL 块的示例：计算圆的面积，并将结果插入 AREAS 表中。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  radius := 3;
  area := pi*power(radius,2);
  insert into AREAS values (radius, area);
end;
/
```

在上面的程序清单中，可执行命令部分如下：

```
begin
  radius := 3;
  area := pi*power(radius,2);
  insert into AREAS values (radius, area);
end;
```

在关键字 begin 之后，PL/SQL 块开始工作。首先，给 radius 变量赋值，radius 变量和 π 常量决定 area 变量的值。然后，将 Radius 和 Area 的值插入 AREAS 表中。

PL/SQL 块的可执行命令部分包含的命令执行在声明部分中声明的游标。在下面的示例中(来自 32.2 节)，可执行命令部分包含几个引用 rad_cursor 游标的命令：

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
     select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
  fetch rad_cursor into rad_val;
    area := pi*power(rad_val.radius,2);
    insert into AREAS values (rad_val.radius, area);
  close rad_cursor;
end;
/
```

在涉及游标的第 1 个命令中，使用了 open 命令：

```
open rad_cursor;
```

当打开 rad_cursor 游标时，执行声明该游标的查询，并且识别将要返回的记录。接着，从游标中取出记录：

```
fetch rad_cursor into rad_val;
```

在声明部分中，声明变量 rad_val，用于设定它的数据类型为 rad_cursor 游标的数据类型：

```
cursor rad_cursor is
        select * from RADIUS_VALS;
    rad_val rad_cursor%ROWTYPE;
```

从游标中取出记录并放入 rad_val 变量时，仍能访问通过游标的查询选择的每一列的值。当不再需要游标的数据时，可以关闭游标，如下面的程序清单所示：

```
close rad_cursor;
```

可执行命令部分可以包含条件逻辑，如 if 命令和循环。下面几节将介绍 PL/SQL 中允许的几种主要的流控制操作的示例。可以使用流控制操作改变取出的记录和可执行命令的管理方式。

32.3.1　条件逻辑

在 PL/SQL 中，可以使用 if、else、elsif、continue 和 case 命令来控制可执行命令部分的命令流。if 子句的格式如下所示：

```
if <some condition>
  then <some command>
elsif <some condition>
  then <some command>
else <some command>
end if;
```

if 条件可以相互嵌套，如下所示：

```
if <some condition>
  then
    if <some condition>
      then <some command>
    end if;
else <some command>
end if;
```

通过嵌套 if 条件，可以快速地在可执行命令部分开发出复杂的逻辑流程。在嵌套 if 条件时，不要使流控制过于复杂，应当经常检查逻辑条件是否能够组合成更简单的形式。

现在可修改 32.3 节中“圆面积”的示例，使其包含条件逻辑。在下面的程序清单中，修改后的 PL/SQL 块的可执行命令部分包括 if 命令和 then 命令：

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
       select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
  fetch rad_cursor into rad_val;
```

```
    area := pi*power(rad_val.radius,2);
    if area > 30
      then
        insert into AREAS values (rad_val.radius, area);
    end if;
  close rad_cursor;
end;
/
```

在上述程序清单中，用 if 条件控制 PL/SQL 块中可执行命令部分的命令流。程序清单中的流控制命令如下所示：

```
if  area > 30
  then
    insert into AREAS values (rad_val.radius, area);
end if;
```

该示例中的流控制命令在 area 变量的值确定以后开始执行。如果该值大于 30，则 1 个记录将被插入 AREAS 表中。如果该值小于 30，就不在表中插入任何记录。可以基于 if 条件使用这种流控制，从而决定执行哪些 SQL 语句。

下面的程序清单显示了包含 SQL 命令的流控制的语法：

```
if area>30
  then <some command>
elsif area<10
  then <some command>
else <some command>
end if;
```

下面关于循环类型的示例将介绍流控制的 case 命令的用法。

32.3.2 循环

可以在一个 PL/SQL 块中用循环处理多条记录。PL/SQL 支持 3 种类型的循环：

- **简单循环** 直到在循环中遇到 exit 或 exit when 语句时，才跳出循环
- **FOR 循环** 指定循环次数的循环
- **WHILE 循环** 在满足某个条件时循环

在下面几节中，将会看到以上各种循环类型的示例。循环的示例可作为本章前面用过的 PL/SQL 块的起点。可以用循环处理来自一个游标的多条记录。

1. 简单循环

在下面的程序清单中，用一个简单循环在表 AREAS 中生成多行。此循环以 loop 关键字开始，exit when 子句确定何时退出循环。end loop 子句标识循环的结束。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
```

```
  area NUMBER(14,2);
begin
  radius := 3;
  loop
    area := pi*power(radius,2);
      insert into AREAS values (radius, area);
    radius := radius+1;
    exit when area >100;
  end loop;
end;
/
```

以上示例中的循环部分为 PL/SQL 块的可执行命令部分的命令创建了流控制。循环中的步骤在下面这个带注释的 loop 命令中描述：

```
loop
      /* Calculate the area, based on the radius value.   */
    area := pi*power(radius,2);
      /* Insert the current values into the AREAS table.  */
    insert into AREAS values (radius, area);
      /* Increment the radius value by 1.                 */
    radius := radius+1;
      /* Evaluate the last calculated area. If the value  */
      /* exceeds 100, then exit. Otherwise, repeat the    */
      /* loop using the new radius value.                 */
    exit when area >100;
      /* Signal the end of the loop.                      */
end loop;
```

此循环应该在 AREAS 表中生成多条记录。第一条记录是值为 3 的 Radius 所生成的记录。一旦 area 的值超过 100，就不会再有记录插入到 AREAS 表中了。

此 PL/SQL 块的执行结果如下所示：

```
select *
  from AREAS
order by Radius;

   RADIUS       AREA
--------- ----------
        3      28.27
        4      50.27
        5      78.54
        6      113.1
```

因为当 Radius 的值为 6 时，area 的值超过 100，所以不再处理 Radius 的值，PL/SQL 块结束。

2. 简单的游标循环

可以使用游标的属性(如是否还有可取出的行等)作为退出循环的条件。在下面的示例中，

游标执行到查询不再返回任何行时，要确定游标的状态，就需要检查游标的属性。游标有 4 个可以在程序中使用的属性：

- %FOUND 可从游标中取出 1 条记录
- %NOTFOUND 不能从游标中再取到记录
- %ISOPEN 游标已经打开
- %ROWCOUNT 迄今为止从游标中取出的行数

游标的%FOUND、%NOTFOUND 和%ISOPEN 属性是布尔型的；它们被设置成 TRUE 或 FALSE。因为它们是布尔型的属性，所以可以直接判断它们的设置而不用显式地将其与 TRUE 或 FALSE 进行匹配。例如，下面的命令在 rad_cursor%NOTFOUND 为 TRUE 时，会导致退出：

```
exit when rad_cursor%NOTFOUND;
```

下面的程序清单使用 1 个简单循环来处理来自 1 个游标的多行记录：

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
       select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
  loop
    fetch rad_cursor into rad_val;
  exit when rad_cursor%NOTFOUND;
    area := pi*power(rad_val.radius,2);
    insert into AREAS values (rad_val.radius, area);
  end loop;
  close rad_cursor;
end;
/
```

此 PL/SQL 块的循环部分使用 RADIUS_VALS 表中的值作为其输入。退出条件不是基于 Area 值，而是检查游标的%NOTFOUND 属性。如果在游标中找不到更多的行，则%NOTFOUND 为 TRUE，于是退出循环。此循环带注释的程序清单如下：

```
loop
      /* Within the loop, fetch a record.            */
    fetch rad_cursor into rad_val;
      /* If the fetch attempt reveals no more        */
      /* records in the cursor, then exit the loop.  */
    exit when rad_cursor%NOTFOUND;
      /* If the fetch attempt returned a record,     */
      /* then process the Radius value and insert    */
      /* a record into the AREAS table.              */
    area := pi*power(rad_val.radius,2);
```

```
    insert into AREAS values (rad_val.radius, area);
     /* Signal the end of the loop.                    */
end loop;
```

当执行上述 PL/SQL 块时，将对 RADIUS_VALS 表中的每条记录进行循环处理。到目前为止，RADIUS_VALS 表中只包含了一条记录——Radius 的值为 3 的记录。

注意：
在执行本节的 PL/SQL 块之前，向 RADIUS_VALS 表中添加两个新的 Radius 值——4 和 10。

下面的程序清单显示了如何向 RADIUS_VAlS 表中添加新记录：

```
insert into RADIUS_VALS values (4);
insert into RADIUS_VALS values (10);
commit;

select *
   from RADIUS_VALS
  order by Radius;

    RADIUS
----------
         3
         4
        10
```

一旦向 RADIUS_VALS 表中添加了新记录，再执行本节前面的 PL/SQL 块时，输出将如下所示：

```
select *
    from AREAS
  order by Radius;

    RADIUS       AREA
---------- ----------
         3      28.27
         4      50.27
        10     314.16
```

AREAS 表的该查询结果表明 RADIUS_VALS 表中的每一条记录都从游标中取出并进行了相应处理。当游标中没有记录可供处理时，退出循环，PL/SQL 块执行完毕。

3. FOR 循环

在简单循环中，当满足 exit 条件时循环结束。而在 FOR 循环中，循环执行指定的次数。下面的程序清单给出了一个 FOR 循环的示例。FOR 循环的开始由关键字 for 指定，后面是用来决定循环过程何时完成以及循环何时退出的条件。由于循环执行的次数在循环开始时设

置，因此在循环中不再需要 exit 命令。

在下面的示例中，分别计算 Radius 的值为 1~7 的圆面积。

```
delete from AREAS;

declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  for radius in 1..7 loop
    area := pi*power(radius,2);
    insert into AREAS values (radius, area);
  end loop;
end;
/
```

循环的处理步骤如下面的程序清单所示，其中包含一些注释：

```
    /* Specify the criteria for the number of loop      */
    /* executions.                                      */
  for radius in 1..7 loop
    /* Calculate the area using the current Radius      */
    /* value. */
    area := pi*power(radius,2);
    /* Insert the area and radius values into the AREAS */
    /* table.                                           */
    insert into AREAS values (radius, area);
    /* Signal the end of the loop.                      */
  end loop;
```

请注意，此 FOR 循环中没有下面这一行：

```
radius := radius + 1 ;
```

因为循环指定了下面的说明：

```
for radius in 1..7 loop
```

这样，Radius 值已被指定。对每一个值，都要执行循环中的所有命令(这些命令可以包括其他条件逻辑，如 if 条件)。在循环处理完一个 Radius 值后，检查对 for 子句的限制，然后或者使用下一个 Radius 值，或者循环执行完毕。

此 FOR 循环的执行结果如下：

```
select *
   from AREAS
  order by Radius;

    RADIUS       AREA
---------- ----------
```

```
         1         3.14
         2        12.57
         3        28.27
         4        50.27
         5        78.54
         6        113.1
         7       153.94

7 rows selected.
```

4. 游标 FOR 循环

在 FOR 循环中，循环执行指定的次数。而在游标 FOR 循环中，循环执行的次数由查询的结果动态决定。在游标 FOR 循环中，打开游标、取出游标和关闭游标隐式完成，不需要显式指定这些操作。

下面是游标 FOR 循环的一个示例，它查询 RADIUS_VALS 表，并向 AREAS 表中插入记录：

```
delete from AREAS;

declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
     select * from RADIUS_VALS;
begin
  for rad_val in rad_cursor
    loop
      area := pi*power(rad_val.radius,2);
      insert into AREAS values (rad_val.radius, area);
    end loop;
end;
/
```

在游标 FOR 循环中，没有 open 和 fetch 命令。下面的命令

```
for rad_val in rad_cursor
```

隐式地打开 rad_cursor 游标，并将取出的值放入 rad_val 变量中。当该游标中没有记录时，退出循环，游标关闭。在游标 FOR 循环中，不需要 close 命令。请注意：rad_val 没有在块中显式地声明。

下面的程序清单列出了此 PL/SQL 块的循环部分，其中的注释说明了控制流。此循环通过 rad_cursor 游标中是否存在可取记录来控制。这里不需要检查游标的%NOTFOUND 属性——它通过游标 FOR 循环自动检查。

```
    /* If a record can be fetched from the cursor,      */
    /* then fetch it into the rad_val variable. If      */
    /* no rows can be fetched, then skip the loop.      */
  for rad_val in rad_cursor
```

```
    /* Begin the loop commands.                                    */
    loop
       /* Calculate the area based on the Radius value  */
       /* and insert a record into the AREAS table.     */
      area := pi*power(rad_val.radius,2);
      insert into AREAS values (rad_val.radius, area);
    /* Signal the end of the loop commands.                        */
    end loop;
```

下面的程序清单给出了样本输出；在此示例中，RADIUS_VALS 表有 3 条记录，Radius 值分别为 3、4 和 10。

```
select *
  from RADIUS_VALS
order by Radius;

    RADIUS
----------
         3
         4
        10
```

执行具有游标 FOR 循环的 PL/SQL 块分将在 AREAS 表中生成如下记录：

```
select *
  from AREAS
order by Radius;

    RADIUS       AREA
---------- ----------
         3      28.27
         4      50.27
        10     314.16
```

5. WHILE 循环

直到满足退出条件，WHILE 循环才会结束。在此种循环中，不再用 exit 命令指定退出循环的条件，而是在循环的开始用 while 命令指定退出循环的条件。

在下面的程序清单中，用 WHILE 循环处理多个 Radius 值。如果 Radius 变量当前的值满足循环中规定的 while 条件，就执行循环的命令。一旦 Radius 值不满足循环的条件，循环就结束。

```
delete from AREAS;

declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  radius := 3;
  while radius<=7
```

```
    loop
      area := pi*power(radius,2);
      insert into AREAS values (radius, area);
    radius := radius+1;
  end loop;
end;
/
```

WHILE 循环和简单循环在结构上很相似，都是基于变量的值来决定是否终止循环。下面的程序清单显示了循环中的步骤和包含的注释：

```
    /* Set an initial value for the Radius variable.      */
  radius := 3;
    /* Establish the criteria for the termination of      */
    /* the loop. If the condition is met, execute the     */
    /* commands within the loop. If the condition is      */
    /* not met, then terminate the loop.                  */
  while radius<=7
     /* Begin the commands to be executed.                */
    loop
       /* Calculate the area based on the current        */
       /* Radius value and insert a record in the        */
       /* AREAS table.                                   */
       area := pi*power(radius,2);
       insert into AREAS values (radius, area);
        /* Set a new value for the Radius variable. The  */
       /* new value of Radius will be evaluated against  */
       /* the termination criteria and the loop commands */
       /* will be executed for the new Radius value or   */
       /* the loop will terminate.                       */
       radius := radius+1;
       /* Signal the end of the commands within the loop. */
    end loop;
```

执行过程中，上述程序清单中的 PL/SQL 块将在 AREAS 表中生成记录。此 PL/SQL 块的输出结果如下：

```
select *
  from AREAS
order by Radius;

    RADIUS         AREA
---------- ----------
         3        28.27
         4        50.27
         5        78.54
         6        113.1
         7       153.94
```

由于在循环开始之前给 Radius 变量赋过值，因此循环至少应执行一次。应当保证赋给变

量的值满足循环执行的条件。

6. 使用 CONTINUE 和 CONTINUE WHEN 语句

可以在循环中使用 continue 语句退出循环的当前迭代，并将控制权转交给下一次迭代。例如，如果通过一组计数器值进行迭代，就可以跳过指定的值而不完全退出循环。continue 语句有 continue 和 continue when 两种形式.

下面的程序清单说明了 continue 语句的用法。在此例中，第一次循环迭代从值 1 开始。当计算 if 子句的值时，小于 5 的任何值都会触发 continue 语句，并开始下一次循环迭代。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  radius := 1;
  loop
    if radius < 5 then
      continue;
    end if;
      area := pi*power(radius,2);
        insert into AREAS values (radius, area);
      radius := radius+1;
      exit when area >100;
    end loop;
end;
/
select *
   from AREAS
  order by Radius;

   RADIUS       AREA
--------- ----------
        5      78.54
        6      113.1
```

也可以在循环中使用 continue when 语句。when 部分提供了要判断的条件。如果条件为真，则当前的循环迭代完成，控制权转交给下一次迭代。下面的程序清单说明了 continue when 语句的用法。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
begin
  radius := 1;
  loop
  continue when radius < 5;
    area := pi*power(radius,2);
      insert into AREAS values (radius, area);
    radius := radius+1;
```

```
    exit when area >100;
  end loop;
end;
/
```

可以使用 continue 语句强制循环只处理偶数次迭代(其中 mod(radius,2)=0)或者想要应用的任何定制条件。

32.3.3　CASE 语句

可以使用 case 语句控制 PL/SQL 块中的分支逻辑。例如，可以使用 case 语句按条件赋值，或者在插入值之前进行转换。在下例中，case 表达式使用 Radius 值确定将哪些行插入到 AREAS 表中：

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
     select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
  loop
    fetch rad_cursor into rad_val;
    exit when rad_cursor%NOTFOUND;
    area := pi*power(rad_val.radius,2);
    case
    when rad_val.Radius = 3
    then
      insert into AREAS values (rad_val.radius, area);
    when rad_val.Radius = 4
    then
      insert into AREAS values (rad_val.radius, area);
    when rad_val.Radius = 10
    then
      insert into AREAS values (0, 0);
    else raise CASE_NOT_FOUND;
   end case;
  end loop;
  close rad_cursor;
end;
/
```

该代码块产生以下输出结果。当 Radius 的值为 10 时，case 子句插入 Radius 的值和 Area 的值都置 0 的一行：

```
select * from AREAS;

    RADIUS         AREA
---------- ----------
         3        28.27
```

```
             4          50.27
             0             0
```

关键字 case 表示子句的开始：

```
case
  when rad_val.Radius = 3
   then
     insert into AREAS values (rad_val.radius, area);
```

when 子句按顺序判断。case 子句中的 else 关键字与 if-then 子句中的 else 关键字作用相似。如果省略 else 关键字，则 PL/SQL 隐式地增加以下 else 子句：

```
else raise CASE_NOT_FOUND;
```

end case 子句结束 case 子句。case 子句经常用于把一列值转换成相应的说明，如下例用于转换书架的分类名。下面的程序清单显示了 CategoryName 值如何被转换成作为 SQL 命令一部分的其他值：

```
case CategoryName
     when 'ADULTFIC' then 'Adult Fiction'
     when 'ADULTNF' then 'Adult Nonfiction'
     when 'ADULTREF' then 'Adult Reference'
     when 'CHILDRENFIC' then 'Children Fiction'
     when 'CHILDRENNF' then 'Children Nonfiction'
     when 'CHILDRENPIC' then 'Children Picturebook'
     else CategoryName
end
```

case 子句的复杂用法示例请参见第 16 章。

32.4 异常处理部分

当遇到用户定义或与系统有关的异常(错误)时，PL/SQL 块的控制转交给异常处理部分。在异常处理部分，用 when 子句判断触发哪个异常，即执行哪段代码。

如果在 PL/SQL 块的可执行命令部分触发一个异常，则命令流立刻退出可执行命令部分，并在异常处理部分中搜索与遇到的错误相匹配的异常。PL/SQL 提供一组系统定义的异常，并允许添加用户自定义的异常。关于用户自定义异常的示例，参见第 34 章和第 35 章。

异常处理部分总是以 exception 关键字开始，并放在终止 PL/SQL 块的可执行命令部分的 end 命令之前。PL/SQL 块中异常处理部分的顺序如下所示：

```
declare
  <declarations section>
begin
  <executable commands>
exception
  <exception handling>
```

```
end;
```

PL/SQL块中的异常处理部分是可选的——本章前面的PL/SQL块示例程序中没有一个包括异常处理部分。但是，本章的示例基于几个已知的输入值，所完成的处理也非常有限。

在下面的程序清单中，显示了计算圆面积的一个简单循环，只是作了两处改动。在声明部分声明了一个新的名为 some_variable 的变量，在可执行命令部分通过计算确定该变量的值。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
  some_variable NUMBER(14,2);
begin
  radius := 3;
  loop
    some_variable := 1/(radius-4);
    area := pi*power(radius,2);
      insert into AREAS values (radius, area);
    radius := radius+1;
    exit when area >100;
  end loop;
end;
/
```

因为变量 some_variable 的计算中包含除法，所以可能会遇到除以零的情况——这是一个错误情况。第一次循环迭代时，处理 radius 变量(初始值为 3)，并且向 AREAS 表中插入一条记录。第二次循环迭代时，radius 变量的值为 4——此时 some_variable 变量的计算遇到一个错误：

```
declare
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 9
```

由于产生了错误，因此插入表 AREAS 的第一行被回滚，并且 PL/SQL 块中止。

可以通过在 PL/SQL 块中添加异常处理部分来修改错误情况的处理，如下面的程序清单所示：

```
delete from AREAS;

declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
  some_variable NUMBER(14,2);
begin
  radius := 3;
  loop
    some_variable := 1/(radius-4);
```

```
    area := pi*power(radius,2);
     insert into AREAS values (radius, area);
    radius := radius+1;
    exit when area >100;
  end loop;
exception
  when ZERO_DIVIDE
    then insert into AREAS values (0,0);
end;
/
```

下面的程序清单重复了 PL/SQL 块的异常处理部分：

```
exception
  when ZERO_DIVIDE
  then insert into AREAS values (0,0);
```

当 PL/SQL 块遇到错误时，它会扫描异常处理部分，以便找到所定义的异常。在本例中，它找到 ZERO_DIVIDE 异常，它是 PL/SQL 中一个系统定义的异常。除系统定义的异常和用户定义的异常外，还可以使用 when others 子句处理在异常处理部分中未定义过的所有异常。执行异常处理部分中此异常的相应命令，并在 AREAS 表中插入一条记录。此 PL/SQL 块的输出如下所示：

```
select *
  from AREAS;

    RADIUS       AREA
---------- ----------
         3      28.27
         0          0
```

此输出表明第一个 Radius 的值(3)被处理了，而在第二次循环时出现异常。

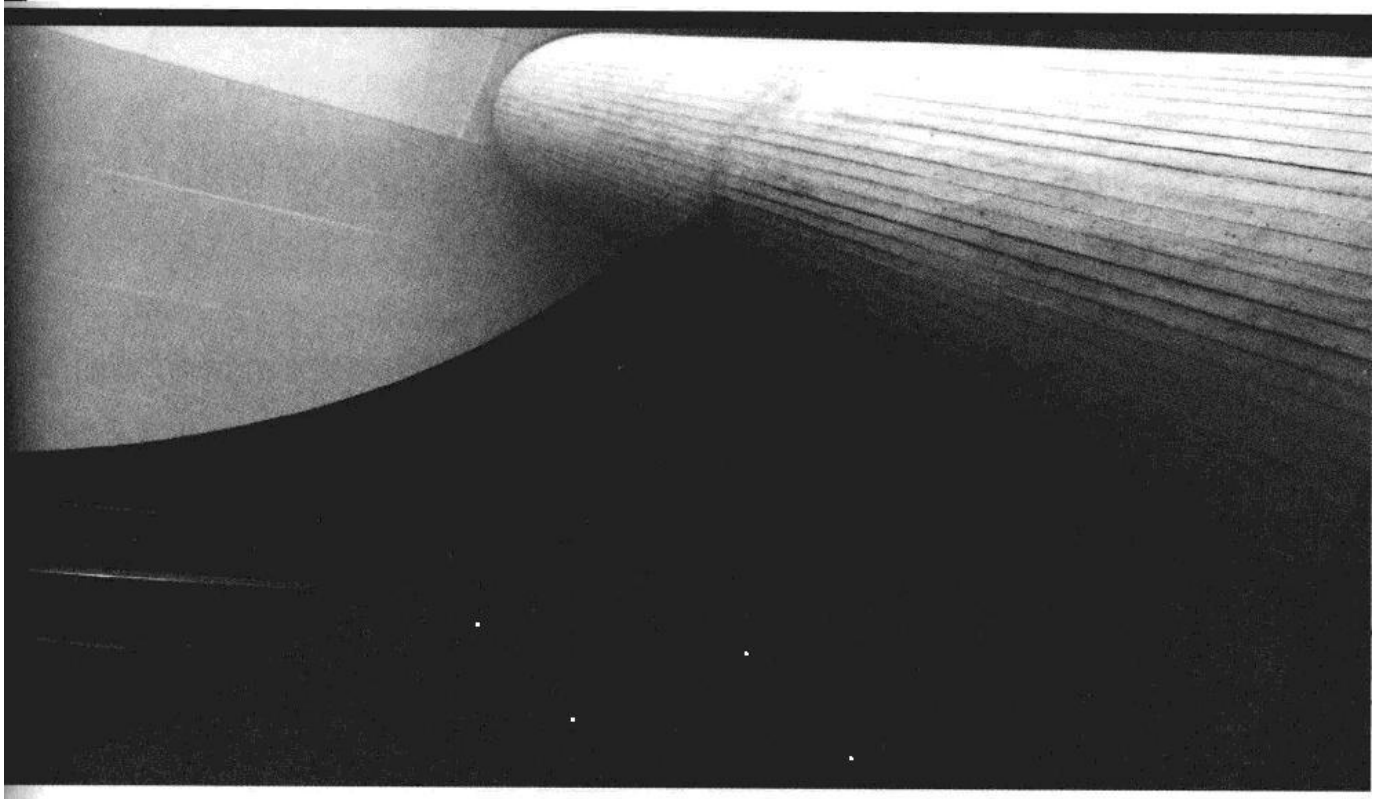
注意：

一旦遇到异常，就不能返回到可执行命令部分中正常的命令处理流了。如果需要使控制保持在可执行命令部分中，就应当在程序遇到可能的异常之前用 if 条件去测试它们，或创建一个拥有本地异常处理的嵌套块。

可用的系统定义的异常在附录 A 的“EXCEPTION”条目下可以找到。用户定义的异常的示例参见第 34 章和第 35 章。

有关动态 PL/SQL 的详细信息，请参阅第 37 章。动态 PL/SQL 允许动态创建在 PL/SQL 块中执行的命令。

PL/SQL 支持那些使用 Oracle 的 SQL 最新特性的复杂查询。例如，从 Oracle Database 10g 开始，在 PL/SQL 中支持闪回版本查询(参阅第 27 章)和正则表达式函数(参阅第 8 章)。有关这些新特性的详细介绍，请参阅相关章节；它们都可以引入到 PL/SQL 中，用来简化和增强处理能力。



第33章

应用程序在线升级

随着业务需求的变化，支持业务需求的应用程序也需要变化。Oracle 提供了很多种方法来修改已存在的应用程序，可以使用 Oracle 11g 中引入的特性来修改应用程序，同时保持应用程序用户要求的高可用性。

本章将介绍如何在支持变更需求的同时最小化对应用程序造成的影响。其中的很多方法也会在其他章节中介绍。

33.1 高可用数据库

如果应用程序依赖于可用的数据库，则数据库体系结构的整个设计必须支持高可用性。在数据库级别，这涉及从很多配置选项中进行选择：

- RAC 数据库 在这种数据库中，多个实例访问一个共享的数据库。RAC 数据库可以在节点故障的情况下提供透明应用程序故障转移。
- 复制的数据库 在这种数据库中，不同人的数据库作为数据的“主”站点，复制的数据库同时提供数据库和实例的完全故障转移功能。在确定如何解决不同数据库之间在数据级别的冲突时，复制的数据库会增加设计的复杂性。
- 备用数据库 在这种数据库中，一个数据库是主数据库，另一个不同的数据库充当辅助数据库。备用数据库在主数据库发生故障时可用。

这些选项都支持高可用性需求。对环境的正确选择取决于特定应用程序的需求。当评估高可用性选项时，评估标准通常基于灾难场景、丢失的数据量以及恢复所需要的时间。但是也应该考虑日常的和普通的场景：对已存在的数据库表进行变更的需要。

在 RAC 解决方案中，只有一个表被改变，因为不同的实例共享相同的数据库。在表上创建 DDL 锁的任何操作都会影响所有实例。

在复制的数据库环境中，对象通常保持高度同步。对一个环境做出任何改变，都必须在复制的数据库中实施相应的改变。在一个数据库中 DDL 改变造成的中断一般都会导致复制数据库的中断。

在备用数据库解决方案中，备用数据库可以是精确的物理复制数据库，也可以是逻辑备用数据库，备用数据库具有不同于主数据库的对象。鉴于这种体系结构，可以使用备用数据库实现应用程序对象的滚动升级——先在一个环境中改变它们，然后在第二个环境中改变，应用程序不会长时间的中断。您需要与 DBA 密切协作，来配置和管理备用环境，从而支持这种方法。对于要求高可用性同时又需要维护的应用程序，这种体系结构具有很大的好处。

备用数据库是通过 Oracle Data Guard(Oracle 数据卫士)技术得到支持的。虽然创建和配置 Data Guard 数据库是 DBA 的任务，但开发人员也应该知道涉及的体系结构和特性。例如，备用数据库可以以只读方式启动，以支持用户，然后再返回到备用方式。对主数据库的改变可自动从主数据库重放到备用数据库，并可确保处理过程中无数据丢失。备用数据库服务器可以在物理上与主服务器分开。

以下各节将介绍如何配置 Oracle Data Guard 环境，然后介绍几种在对象改变期间使停机时间最短的 Oracle 功能。

33.1.1 Oracle Data Guard(数据卫士)体系结构

在 Data Guard 的实现中，将一个运行在 ARCHIVELOG 模式下的数据库指定为一个应用程序的主数据库。通过 Oracle Net(Oracle 网络)可访问的一个或多个备用数据库提供故障转移功能。Data Guard 自动将重做信息发送到应用此信息的备用数据库。因此，备用数据库在事务上保持一致。根据重做应用程序过程的配置方式，备用数据库可能与主数据库同步，也可能滞后于主数据库。

通过初始化参数设置，可以按照定义自动将重做日志的数据传送到备用数据库。在备用服务器上，接收和应用重做日志的数据。

1. 物理备用数据库与逻辑备用数据库

有两种类型的备用数据库：物理备用数据库和逻辑备用数据库。物理备用数据库具有和主数据库相同的结构。逻辑备用数据库具有不同的内部结构(如用于报表的其他索引)。通过将重做数据转换为依据备用数据库执行的 SQL 语句，可以使逻辑备用数据库和主数据库同步。

物理备用数据库和逻辑备用数据库用于不同的目的。由于物理备用数据库是主数据库所有块的一个副本，因此它可以用作替代主数据库的数据库备份。在灾难恢复过程中，物理备用数据库看起来就像是它替代的主数据库。

由于逻辑备用数据库支持其他数据库结构，因此可以更容易地用于支持特定的报表需求，否则这种需求会加重主数据库的负担。另外，当使用逻辑备用数据库时，能够以最少的停机时间执行主数据库和备用数据库的滚动更新。根据需要使用的备用类型有所不同，许多环境最开始将物理备用数据库用于灾难恢复，然后添加其他的逻辑备用数据库来支持特定的报表和业务需求。

注意：

主位置和备用位置上的操作系统和平台的体系结构必须相同。虽然目录结构和操作系统补丁级别可以有所不同，但是应最小化这种区别来简化管理和故障转移过程。如果备用数据库和主数据库位于同一个服务器上，则这两个数据库必须使用不同的目录结构，并且它们不能共享一个归档日志目录。

2. 数据保护模式

当配置主数据库和备用数据库时，将需要确定业务可以接受的数据丢失的程度。在主数据库中，需要定义归档日志的目标区域，并且至少有一个目标区域将引用备用数据库使用的远程站点。备用数据库的 LOG_ARCHIVE_DEST_*n* 参数设置的 ASYNC、SYNC、ARCH、LGWR、NOAFFIRM 和 AFFIRM 属性将指导 Oracle Data Guard 在多种操作模式中做出选择：

- 在最大保护(或“无数据丢失”)模式下，在主数据库中提交一个事务之前，事务必须至少写入一个备用位置。如果备用数据库的日志存放位置不可用，则主数据库关闭。
- 在最大可用性模式下，在主数据库中提交一个事务之前，事务必须至少写入一个备用位置。如果备用位置不可用，则主数据库不关闭。当纠正了错误后，出错之后生成的重做数据会传送并应用到备用数据库上。
- 在最大性能模式(默认模式)下，可以在将它们的重做信息传送到备用位置之前提交事务。一旦事务写入本地联机重做日志，就可以在主数据库中进行提交。

一旦已经为配置确定了备用类型和数据保护模式，就可以创建备用数据库。

33.1.2 创建备用数据库配置

可以使用 SQL*Plus、Oracle Enterprise Manager(OEM)或 Data Guard 特有的工具来配置和管理 Data Guard 配置。设置的参数将依赖于选择的配置。

如果主数据库和备用数据库在同一个服务器上，就需要为 DB_UNIQUE_NAME 初始化参数设置一个值。由于这两个数据库的目录结构就是不同的，因此必须手工重新命名文件或者为备用数据库中的 DB_FILE_NAME_CONVERT 和 LOG_FILE_NAME_CONVERT 参数定义值。必须通过 SERVICE_NAMES 初始化参数为主数据库和备用数据库设置唯一的服务名。

如果主数据库和备用数据库位于分离的服务器上，则可以将相同的目录结构用于每个数据库，从而不需要文件名转换参数。如果将一个不同的目录结构用于数据库文件，则需要为备用数据库中的 DB_FILE_NAME_CONVERT 参数和 LOG_FILE_NAME_CONVERT 参数定义值。

在物理备用数据库中，所有的重做(数据)来自主数据库。当物理备用数据库在只读模式下打开时，不会生成重做数据。然而，Oracle Data Guard 确实要使用归档的重做日志文件来支持数据和 SQL 命令的复制，这些数据和 SQL 命令用于更新备用数据库。

注意：
对每个备用数据库，应该创建一个备用重做日志文件来存储从主数据库接收到的重做数据。

1. 创建逻辑备用数据库

创建逻辑备用数据库与创建物理备用数据库有很多步骤是相同的。因为逻辑备用数据库依赖于 SQL 命令的重新执行，所以在使用上逻辑备用数据库有更大的限制。如果主数据库中的任何一个表使用如下的数据类型，那么在重做应用程序过程中将会忽略它们：

- BFILE
- ROWID
- UROWID
- 用户定义的数据类型
- 对象类型
- REF
- 可变数组
- 嵌套表
- XML 类型

注意：
逻辑备用数据库不支持加密列。

此外，在重做应用过程中会忽略采用表压缩的表和使用 Oracle 软件安装的模式。DBA_LOGSTDBY_UNSUPPORTED 视图列出了逻辑备用数据库不支持的对象。DBA_LOGSTDBY_

SKIP 视图列出了将会忽略的模式。

逻辑备用数据库不同于主数据库。在逻辑备用数据库中执行的每个事务必定逻辑上等价于主数据库中已执行的事务。因此，应该确保表对逻辑备用数据库施加了适当的限制——主键、唯一约束、检查约束和外键——因此能够选取正确的行用于逻辑备用数据库中的更新。可以查询 DBA_LOGSTDBY_NOT_UNIQUE 来列出在主数据库中缺少主键或唯一约束的表。

2. 使用实时应用

默认情况下，直到已经归档了备用重做日志文件后，才将重做数据应用于备用数据库。当使用实时应用功能时，在接收重做数据的同时将它应用于备用数据库，从而减少了数据库之间的时间延迟并缩短了故障转移到备用数据库所需的时间。

为了在物理备用数据库中启用实时应用，在备用数据库内执行如下的命令：

```
alter database recover managed standby database
using current logfile;
```

对于逻辑备用数据库，使用的命令是：

```
alter database start logical standby apply immediate;
```

如果已经启用了实时应用，那么 V$ARCHIVE_DEST_STATUS 视图的 Recovery_Mode 列将有一个“MANAGED REAL-TIME APPLY”值。

如本章前面所述，可以通过如下的命令在物理备用数据库上启用重做应用程序：

```
alter database recover managed standby database
  disconnect from session;
```

在与 Oracle 会话断开连接后，disconnect from session 子句允许该命令在后台运行。当启动了一个前台会话并发出相同的不带 disconnect from session 子句的命令时，命令提示符不会返回控制指令，直到另一个会话取消了恢复。为了在物理备用数据库中停止重做应用程序——无论是在后台会话还是在前台会话中——使用如下命令：

```
alter database recover managed standby database cancel;
```

对于逻辑备用数据库，停止日志应用服务的命令是：

```
alter database stop logical standby apply;
```

33.1.3　管理角色——切换和故障转移

参与 Data Guard 配置的每个数据库都有一个角色——或者是主数据库，或者是备用数据库。在某些时候，这些角色可能需要改变。例如，如果在主数据库的服务器上有一个硬件故障，那么主数据库可能将故障转移到备用数据库。根据配置选择，在故障转移过程中可能有一些数据丢失。

第二类角色改变称为“切换(switchover)”，这种情况出现在主数据库和备用数据库切换角色并且备用数据库变成新的主数据库时。在切换过程中，应该不存在任何数据丢失。切换和故障转移都需要数据库管理员的手工干预。

1. 切换

切换是有计划的角色改变，通常考虑在主数据库服务器上执行维护活动。当选择一个备用数据库充当新的主数据库时，会发生切换，并且应用程序把它们的数据写入新的主数据库中。在以后的某个时间点，可以把数据库切换回它们原始的角色。

注意：
可以使用一个逻辑备用数据库或一个物理备用数据库来执行切换，物理备用数据库是首选的选项。

如果已经定义了多个备用数据库，那么该怎么办呢？当其中一个物理备用数据库变为新的主数据库时，其他备用数据库必须能够从新的主数据库接收它们的重做日志数据。在这种配置中，必须定义 LOG_ARCHIVE_DEST_*n* 参数，以便允许这些备用站点从新的主数据库接收数据。

注意：
要验证将变成新的主数据库的数据库正运行在 ARCHIVELOG 模式下。

以下各节将详细介绍执行切换到备用数据库所需的步骤。在切换之前，备用数据库应该主动地应用重做日志数据，因为这会最小化完成切换所需的时间。

2. 切换到物理备用数据库

在主数据库上启动切换并在备用数据库上完成切换。本节将介绍切换到物理备用数据库的步骤。

从验证主数据库是否能够执行切换开始。查询 V$DATABASE 来了解 Switchover_Status 列的值：

```
select Switchover_Status from V$DATABASE;
```

如果 Switchover_Status 列的值不是 TO STANDBY，就不可能执行切换(通常由于配置或硬件问题)。如果该列的值是 SESSIONS ACTIVE，则应该终止当前的用户会话。Switchover_Status 列的有效值如表 33-1 所示。

表 33-1 Switchover_Status 的值

Switchover_Status 的值	说 明
NOT ALLOWED	当前的数据库不是带有备用数据库的主数据库
PREPARING DICTIONARY	该逻辑备用数据库正在向一个主数据库和其他备用数据库发送它的重做数据，以便为切换做准备

(续表)

Switchover_Status 的值	说　明
PREPARING SWITCHOVER	接受用于切换的重做数据时，逻辑备用配置会使用它
RECOVERY NEEDED	备用数据库还没有接收到切换请求
SESSIONS ACTIVE	在主数据库中存在活动的 SQL 会话；在继续执行之前必须断开这些会话
SWITCHOVER PENDING	适用于那些接收到主数据库切换请求但是还没有处理该请求的备用数据库
SWITCHOVER LATENT	切换没有完成并返回到主数据库
TO LOGICAL STANDBY	主数据库接收到来自逻辑备用数据库的完整的字典
TO PRIMARY	该备用数据库可以转换为主数据库
TO STANDBY	该主数据库可以转换为备用数据库

在主数据库中，可以使用如下命令把它转换到物理备用数据库角色：

```
alter database commit to switchover to physical standby;
```

在执行该命令的同时，Oracle 会将当前主数据库的控制文件备份到一个跟踪文件上。此时，应该关闭主数据库并安装它：

```
shutdown immediate;
startup mount;
```

主数据库做好了切换的准备，现在应该回到将充当新的主数据库的物理备用数据库。

在物理备用数据库中，检查 V$DATABASE 视图中的切换状态，它的状态应该为 TO PRIMARY(参见表 33-1)。现在可以通过如下命令将物理备用数据库切换为主数据库：

```
alter database commit to switchover to primary;
```

关闭并启动数据库，数据库将完成到主数据库角色的转换。数据库启动之后，首先通过 alter system switch logfile 命令强制日志切换，然后，如果重做应用服务还没有在后台运行，则在备用数据库上启动重做应用服务。

3. 切换到逻辑备用数据库

在主数据库上启动切换并在备用数据库上完成切换。本节将了解切换到逻辑备用数据库所需的步骤。

从检验主数据库是否能够执行切换开始，查询 V$DATABASE 来了解 Switchover_Status 列的值：

```
select Switchover_Status from V$DATABASE;
```

为了完成切换，该状态必须是 TO STANDBY、TO LOGICAL STANDBY 或 SESSIONS ACTIVE。

在主数据库中，执行如下命令以便使主数据库准备好切换：

```
alter database prepare to switchover to logical standby;
```

在逻辑备用数据库中，执行如下命令：

```
alter database prepare to switchover to primary;
```

在这个时候，逻辑备用数据库将开始向当前的主数据库和配置中的其他备用数据库传送它的重做数据。此时，传送逻辑备用数据库中的重做数据，但是没有应用这些数据。

在主数据库中，现在必须验证从逻辑备用数据库接收了字典数据。在能够继续执行下一步骤之前，V$DATABASE 中的 Switchover_Status 列的值必须读取主数据库中的 TO LOGICAL STANDBY。当该状态值显示在主数据库中时，将主数据库切换到逻辑备用角色：

```
alter database commit to switchover to logical standby;
```

不需要关闭和重启旧的主数据库。现在应该返回最初的逻辑备用数据库并检验它在 V$DATABASE 中的 Switchover_Status 值(应该是 TO PRIMARY)。然后可以完成切换，在原始的逻辑备用数据库中，执行如下命令：

```
alter database commit to switchover to primary;
```

现在原始的逻辑备用数据库成为主数据库。在新的主数据库中，执行日志切换：

```
alter system archive log current;
```

在新的逻辑备用数据库(旧的主数据库)中，启动重做应用过程：

```
alter database start logical standby apply;
```

到此完成了切换。

33.2 最小化 DDL 变更的影响

如果只有一个数据库可用，那么改变表和索引时就需要小心谨慎。当创建或修改对象时，必须先获得对象上的 DDL 锁，以阻止对对象进行其他类型的访问。因为这种锁的问题，所以当用户正在访问对象时应该避免改变这些对象。而且，改变对象会使引用对象的存储过程无效。

虽然理想情况下应该在用户不访问对象时对对象进行更改，但维护时间窗口可能不够大，以致无法支持这种要求。始终应该规划足够的时间来实施更改、测试更改以及重新编译更改所影响到的任何对象。下面几节将介绍几种实施变更的方法，这些方法使需要的维护窗口的尺寸最小化。

33.2.1 创建虚拟列

在 Oracle 11g 中，您可以在表中创建虚拟列，而不用存储导出的数据。表中可以有根据

同一行中其他值而导出的虚拟列，例如，将两列相加得到新的虚拟列。在 Oracle 11g 中，可以将这一函数指定为表定义的一部分，这允许为此虚拟列创建索引，并根据此列对表进行分区。

从变更的角度来看，这一特性的好处在于可以避免在数据库中创建物理列。不必向一个非常大的表添加一列导出数据，相反，可以创建一个虚拟列——即使没有费很多时间执行 DDL 命令，数据对用户仍然是可用的。

考虑 AUTHOR 表：

```
create table AUTHOR
(AuthorName VARCHAR2(50) primary key,
Comments VARCHAR2(100));
```

当值插入到 AUTHOR 表中时，它们是以大写字母形式插入的：

```
insert into AUTHOR values
('DIETRICH BONHOEFFER', 'GERMAN THEOLOGIAN, KILLED IN A WAR CAMP');

insert into AUTHOR values
('ROBERT BRETALL','KIERKEGAARD ANTHOLOGIST');
```

可以使用 generated always 子句创建另外一个大小写字母混合的列，告诉 Oracle 创建虚拟列：

```
AuthorName VARCHAR2(50) primary key,
Comments VARCHAR2(100),
MixedName VARCHAR2(50)
  generated always as
  (initcap(AuthorName) ) virtual
);
```

创建虚拟列使用的代码比较复杂，包括 case 语句和其他函数。需要注意的是，不必创建触发器或其他机制来填充列，对于表中的每一行，Oracle 将此列作为虚拟值来进行维护。此值在物理上并不存储在表中，只在需要的时候才临时生成。

注意：
试图在插入期间为虚拟列提供值会产生错误。

可以在虚拟列上创建索引，这与在表的标准列上创建索引是一样的。另外，可以根据此虚拟列对表进行分区(本章稍后加以描述)。

33.2.2　改变正在使用的表

当您发出 alter table 命令时，Oracle 试图获取表上的 DDL 锁。如果其他人此时正在访问此表，则您发出的命令会失败——当您改变表的结构时，需要对表具有排他的访问权。为了获取您所需要的锁，可能需要反复尝试执行此命令。

在 Oracle 11g 中，可以使用 DDL 锁超时选项来解决这一问题。可以执行 alter session 命令为 ddl-_lock_timeout 参数设置一个值，指定 Oracle 应该持续重试命令的秒数。重试会持续

进行，直到命令成功执行，或者到了设定的超时值。无论首先满足这两个条件中的哪一个，重试都会停止。

要持续重试命令 60 秒，可以发出如下命令：

```
alter session set ddl_lock_timeout=60;
```

DBA 可以通过 alter system 命令在数据库级别启用它，如下所示：

```
alter system set ddl_lock_timeout=60;
```

33.2.3 添加 NOT NULL 列

在 Oracle 11g 中，可以向已经包含行的表中添加 NOT NULL 列。为此，必须为该新列指定一个默认值。对于大型表，这可能有问题，因为可能需要更新数百万行记录，以便包含此新列的值。Oracle 11g 自动处理这一问题，可以避免对用户造成大的影响。

我们再来看一下试图向 TROUBLE 表添加 Condition 列：

```
alter table TROUBLE add (Condition VARCHAR2(9) NOT NULL);
```

如果 TROUBLE 表已经有行，则不能添加 Condition 列，因为已有行的该列值为 NULL。解决方案是为 Condition 列添加一个 DEFAULT 约束：

```
alter table TROUBLE add (Condition VARCHAR2(9)
    default 'SUNNY' NOT NULL);
```

Oracle 现在将创建新的列，但不更新已经存在的行。如果用户选择了 Condition 列值为空的一行，则返回默认值。Oracle 的方法解决了两个问题：不仅允许添加 NOT NULL 的列，而且避免了必须更新已存在的每一行。在大型表中，这会大大节省支持事务大小所需的空间。

33.2.4 在线对象重新组织

在线移动对象时，有几种在线选项可用。对于索引，当索引的基表正在被访问时，可以使用 alter index rebuild online 子句修改索引。可以使用 alter index rebuild online 命令移动现有索引的分区。

对于索引组织的表，可以使用 alter table 命令的 move online 选项。

对于分区表，可以通过 alter table 命令的 partition 子句移动分区。

通过 DBMS_REDEFINITION 程序包可以在线地重新组织大多数类型的表。DBMS_REDEFINITION 方法创建表的一个复制品，并创建触发器和日志以保持复制表与原始表的同步(就如同它是物化视图一样)。您可以在希望的任何时间移动表数据，可以手工刷新复制表中的数据。当您准备好将变更运用于系统时，FINISH_REDER_TABLE 完成重新定义过程，复制表取代原始表。需要注意的是，原始表仍然存在，且包含所有原始数据，直到您手工删除原始表。

重新定义过程分为 5 个步骤：

(1) 验证表可以在线重新构建。

(2) 创建一个临时表及其索引、授权、约束和触发器。

(3) 开始重新定义过程。

(4) 同步源表和目标表(可选)。

(5) 终止或完成重新定义过程。

下面这些步骤说明了 PRACTICE.EMP 表从 USERS 表空间到 USERS_DEST 表空间的重新定义。在此过程中，我们也可以动态地对 EMP 表进行分区。

1. 验证表可以在线重新构建

要验证表可以在线地重新构建，可以执行 CAN_REDEF_TABLE 过程，作为输入提供模式所有者和表名：

```
execute DBMS_REDEFINITION.CAN_REDEF_TABLE('SCOTT','EMP');
```

如果报告错误栈，则列出来的第一个错误会说明问题是什么。以后会忽略关于 DBMS_REDEFINITION 程序包的错误。

2. 创建临时表

您希望 EMP 在新的表空间中看起来什么样？创建一个临时表，在重新定义过程中将使用此临时表。为简便起见，保持此临时表的列名和顺序与源 EMP 表相同。

```
create table EMP_DEST
(EMPNO        NUMBER(4) PRIMARY KEY,
 ENAME        VARCHAR2(10),
 JOB          VARCHAR2(9),
 MGR          NUMBER(4),
 HIREDATE     DATE,
 SAL          NUMBER(7,2),
 COMM         NUMBER(7,2),
 DEPTNO       NUMBER(2))
partition by range (DeptNo)
(partition PART1
       values less than ('30'),
partition PART2
       values less than (MAXVALUE))
tablespace USERS_DEST;
```

在本示例中，EMP_DEST 表在重新定义期间作为临时表。一旦完成重新定义过程，EMP_DEST 就会代替旧的 EMP 表，并且 EMP 表可用的唯一的索引、约束、触发器和授权将成为 EMP_DEST 表的相应索引、索束、触发器和授权。在完成重新定义过程以后，可以选择性地创建索引、触发器、授权和约束，但是现在创建它们避免了以后在活动表上需要 DDL 锁。

为了在目标表(EMP_DEST)上创建源表上已有的所有索引、触发器、约束、权限和统计信息，可以执行 COPY_TABLE_DEPENENTS 过程。COPY_TABLE_DEPENDENTS 的格式如下面的程序清单所示：

```
DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS(
    uname              IN   VARCHAR2,
    orig_table         IN   VARCHAR2,
    int_table          IN   VARCHAR2,
    copy_indexes       IN   PLS_INTEGER := 1,
    copy_triggers      IN   BOOLEAN     := TRUE,
    copy_constraints   IN   BOOLEAN     := TRUE,
    copy_privileges    IN   BOOLEAN     := TRUE,
```

```
    ignore_errors       IN   BOOLEAN      := FALSE,
    num_errors          OUT  PLS_INTEGER,
    copy_statistics     IN   BOOLEAN      := FALSE,
    copy_mvlog          IN   BOOLEAN      := FALSE);
```

3. 启动重新定义过程

START_REDEF_TABLE 过程启动重新定义过程。现在用 EMP 表中提交的数据填充 EMP_DEST 表，因此，这一步骤的时间和撤消需求取决于 EMP 表的大小。作为参数传递模式所有者、旧表名和临时表名：

```
execute DBMS_REDEFINITION.START_REDEF_TABLE -
  ('SCOTT','EMP','EMP_DEST');
```

如果 EMP_DEST 表有一个不同于 EMP 表的列列表，则此列表会作为第 4 个参数传递给 START_REDEF_TABLE 过程。一旦此过程完成，就可以查询 EMP_DEST 表以验证它的数据内容。

4. 终止重新定义过程(可选)

如果此时需要终止重新定义过程，则使用 ABORT_REDEF_TABLE 过程，将模式所有者、源表和临时表作为输入参数：

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE -
  ('SCOTT','EMP','EMP_DEST');
```

终止重新定义过程之后，也应该考虑截断临时表(EMP_DEST)。

5. 对表进行同步(可选)

在重新定义过程结束时，Oracle 将同步源表和临时表之间的数据。为缩短这一过程所需要的时间，可以在最后一个步骤之前对表进行同步。这一可选的步骤允许在临时表中实例化最新的产品数据，以最小化以后对在线用户造成的影响。

为了同步源表和临时表，使用 SYNC_INTERIM_TABLE 过程，将模式所有者、源表和临时表作为输入参数：

```
execute DBMS_REDEFINITION.SYNC_INTERIM_TABLE -
  ('SCOTT','EMP','EMP_DEST');
```

6. 完成重新定义过程

为了完成重新定义过程，执行 FINISH_REDEF_TABLE 过程，如下面的示例所示。输入参数是所有者、源表和临时表名。

```
execute DBMS_REDEFINITION.FINISH_REDEF_TABLE -
  ('SCOTT','EMP','EMP_DEST');
```

7. 验证重新定义过程

因为 EMP 表现在应该分区了，并被移动到 USERS_DEST 表空间中，所以通过查询 DBA_TAB_PARTITIONS 可以验证它的重新定义过程：

```
select Table_Name, Tablespace_Name, High_Value
    from DBA_TAB_PARTITIONS
  where Owner = 'SCOTT';

TABLE_NAME     TABLESPACE_NAME   HIGH_VALUE
------------   ---------------   ----------
EMP            USERS_DEST        MAXVALUE
EMP            USERS_DEST        '30'
```

如上面的程序清单所示，EMP 表的两个分区都驻留在 USERS_DEST 表空间中。此时，应该验证 EMP 表上的外键已启用、所有必需的授权已授予、所有索引已创建，以及所有触发器已启用。

33.2.5　删除列

可以使用表重新组织选项删除列(因为源表和目标表有不同的列定义)。另一种可供选择的方法是，在列正常使用期间可以将列标记为“unused”状态，然后在有更大的维护窗口可用时删除这些列。

例如，可以将 Wind 列标记为 unused：

```
alter table TROUBLE set unused column Wind;
```

将一个列标记为“unused”并不会释放该列以前使用的空间，直到删除未使用的列：

```
alter table TROUBLE drop unused columns;
```

可以查询 USER_UNUSED_COL_TABS、ALL_UNUSED_COL_TABS 和 DBA_UNUSED_COL_TABS 来查看具有 unused 标记列的所有表。

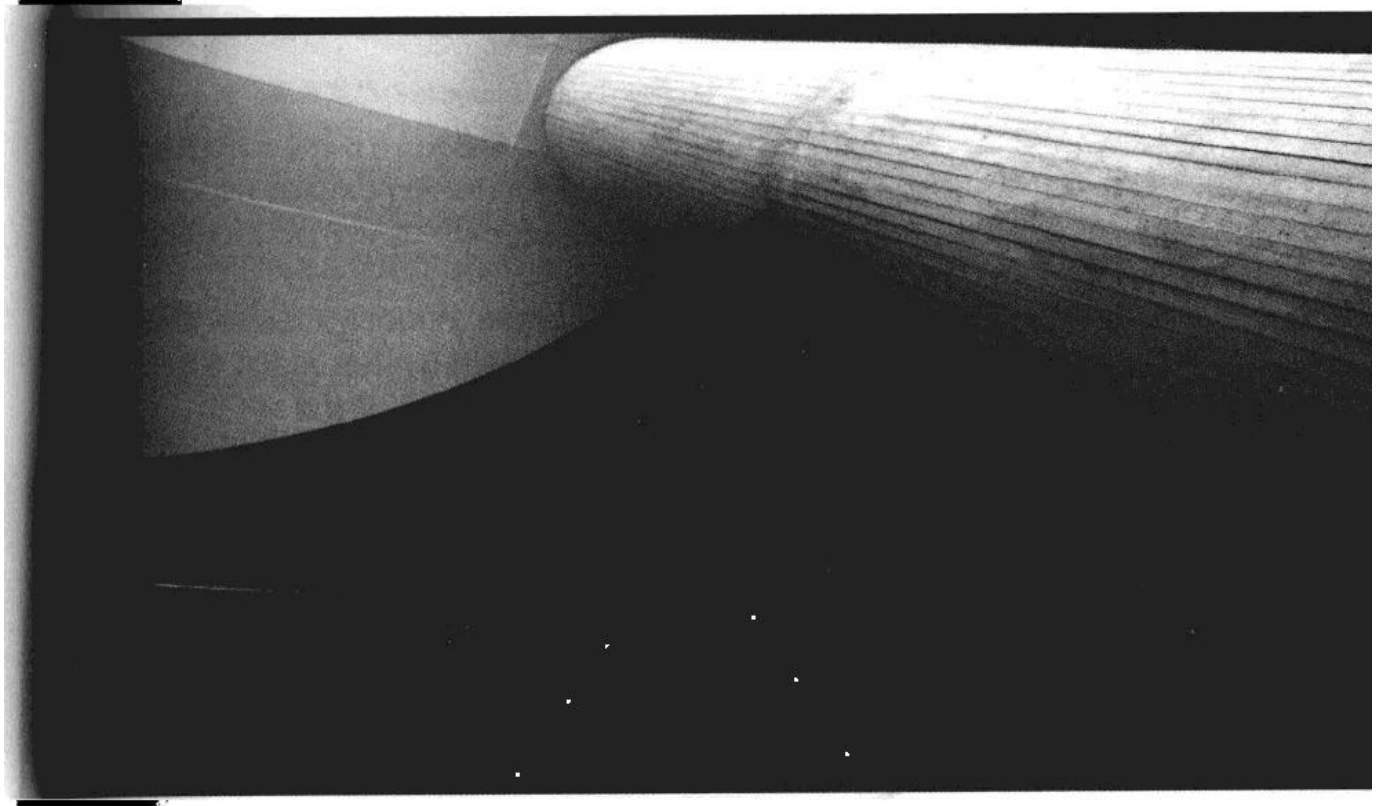
注意：
一旦将某个列标记为“unused”，就不能再访问该列。

可以用一个命令删除多列，如下面的程序清单所示：

```
alter table TROUBLE drop (Condition, Wind);
```

注意：
删除多列时，不应该使用 alter table 命令的 column 关键字；否则会导致语法错误。多个列名必须用圆括号括起来，如上面的程序清单所示。

如果删除的列是主键或唯一约束的一部分，就需要在 alter table 命令中使用 cascade constraints 子句。如果删除属于主键某一列，则 Oracle 同时删除此列和主键索引。



第34章

触 发 器

触发器定义与数据库有关的某个事件发生时数据库将要执行的操作。触发器可用来补充声明的参照完整性，强制实施复杂的业务规则，或审计数据的变化。触发器内的代码，称为触发器体(trigger body)，由 PL/SQL 块(参见第 32 章)构成。

触发器的执行对用户透明。当数据库在特定的表上执行特定的数据操作命令时，执行触发器。这样的命令包括 insert、update 和 delete 等。特定列的更新也可以作为触发事件。触发事件还可包括 DDL 命令和数据库事件，如关闭和登录等。

因为触发器具有灵活性，所以可用来补充参照完整性；但不应该用它代替参照完整性。在应用程序中强制实施某些业务规则时，首先应该依赖 Oracle 声明的参照完整性，然后才用触发器来强制实施那些不能通过参照完整性进行编码的规则。

34.1 必需的系统权限

为了在表上创建触发器，必须能够更改该表。因此，必须要么拥有该表要么对该表有ALTER 权限，或者有 ALTER ANY TABLE 系统权限。此外，还必须有 CREATE TRIGGER 系统权限；为了在其他用户的模式中创建触发器，还必须具有 CREATE ANY TRIGGER 系统权限。

要更改触发器，要么拥有该触发器，要么具有 ALTER ANY TRIGGER 系统权限。还可以通过更改触发器所基于的表来启用或禁用触发器，这要求要么拥有该表的 ALTER 权限，要么具有 ALTER ANY TABLE 系统权限。有关更改触发器的详细介绍，请参阅 34.5 节。

为了在数据库级的事件上创建触发器，必须具有 ADMINISTER DATABASE TRIGGER 系统权限。

34.2 必需的表权限

除了启动触发事件的表外，触发器还可以引用表。例如，如果使用触发器审计 BOOKSHELF 表中数据的变化，则可以在每次更改 BOOKSHELF 表中的记录时，在另一个表(如 BOOKSHELF_AUDIT)中插入一条记录。为了实现上述操作，需要有权对 BOOKSHELF_AUDIT 表进行插入操作(以便执行所触发的事务)。

注意：
触发的事务所需的权限不能来自角色；这些权限必须直接授予触发器的创建者。

34.3 触发器类型

触发器的类型由触发事务的类型和执行该触发器的级别来定义。以下几节将介绍触发器的分类和相关约束条件。

34.3.1 行级触发器

行级触发器(Row-Level Trigger)对受 DML 语句影响的每行执行一次。对于前面提及的 BOOKSHELF 表的审计示例，在 BOOKSHELF 表中更改的每一行都能由触发器处理。行级触发器是最常用的一种触发器，它们通常用于数据审计应用程序中。行级触发器对于同步地保存分布式数据也很有用。第 26 章介绍的物化视图就为此目的而使用了内部行级触发器。

可在 create trigger 命令中使用 for each row 子句创建行级触发器。触发器的语法将在 34.4 节中讲述。

34.3.2 语句级触发器

语句级触发器(Statement-Level Trigger)对每个 DML 语句执行一次。例如，如果一条 INSERT 语句在 BOOKSHELF 表中插入 500 行，那么该表上的语句级触发器只执行一次。因而，语句级触发器不常用于与数据相关的活动；它们通常用于强制实施可能在一个表上执行的各种操作的其他安全措施。

语句级触发器是由 create trigger 命令创建的触发器的默认类型。触发器的语法将在 34.4 节中讲述。

34.3.3 BEFORE 和 AFTER 触发器

由于触发器由事件驱动，因此可以设置触发器为在这些事件之前或之后立即执行。既然执行触发器的事件包括数据库的 DML 语句，触发器也就可以在 insert、update 和 delete 之前或之后立刻执行。对于数据库级的事件，需要应用一些额外的限制；您不能触发在登录或启动之前发生的事件。

在触发器中，可以引用 DML 语句中涉及的旧值和新值。新旧数据所需的访问可以决定所需的触发器类型。“旧”是指在 DML 语句前存在的数据；update 和 delete 通常引用旧值。“新”值是指由 DML 语句创建的数据值(如插入记录中的列)。

如果需要通过触发器在插入的行中设置一个列值，则应该使用 BEFORE INSERT 触发器访问“新”值。使用 AFTER INSERT 触发器不允许设置插入的值，因为该行已经被插入表中。

审计应用程序经常使用 AFTER 行级触发器，因为直到行被修改时才触发它们。行的成功修改表明此行满足为该表定义的参照完整性约束。

34.3.4 INSTEAD OF 触发器

可以使用 INSTEAD OF 触发器告诉 Oracle 要做的工作，而不是执行调用触发器的操作。例如，可以使用某个视图上的 INSTEAD OF 触发器将 insert 重定向到一个表，或者 update 构成一个视图的多个表。可以在对象视图(参阅第 38 章)或者关系视图上使用 INSTEAD OF 触发器。

例如，如果一个视图涉及两个表的连接，则在视图中的记录上使用 update 命令的权限会受到限制。但是，如果使用 INSTEAD OF 触发器，就可以在用户试图通过视图更改值时，告诉 Oracle 怎样 update、delete 或 insert 视图的基表中的记录。执行 INSTEAD OF 触发器中的代码来代替输入 insert、update 或 delete 命令。

注意：
可以在 BEFORE 和 INSTEAD OF 触发器中访问或更改 LOB 数据。

本章将学习如何实现基本的触发器。INSTEAD OF 触发器最初是为了支持对象视图引入的，有关的内容将在第 38 章中介绍。

34.3.5 模式触发器

可以在模式级的操作上创建触发器，这些操作包括 create table、alter table、drop table、audit、rename、truncate 和 revoke 等。甚至可以创建一个 before ddl 触发器。在大多数情况下，模式级触发器主要提供两种功能：阻止DDL操作并且在发生DDL操作时提供额外的安全监控。

34.3.6 数据库级触发器

可以创建在数据库级的事件上触发的触发器，包括错误、登录、注销、关闭和启动。可以用这种类型的触发器自动进行数据库维护或审计操作。第 20 章介绍的虚拟专用数据库就依靠数据库级触发器来建立会话上下文变量值。

34.3.7 复合触发器

在 Oracle 11g 中，您可以将多种类型的触发器组合成一种复合触发器。可以用一种复合触发器执行计算，完成分别执行多个触发器不可能处理的任务。例如，有一个 INSERT 触发器需要计算正在插入行的表的一个平均值。一般来说，这会产生一个“变异表”错误——插入的行改变了表，从而改变了平均值。您可以使用复合触发器在不同的触发器命令之间共享变量值，从而避免变异表错误。

34.4 触发器语法

附录 A 详细介绍了 create trigger 命令的完整语法。下面的程序清单是此命令语法的一个简写版本：

```
create [or replace] trigger [schema .] trigger
  { before | after | instead of }
  { dml_event_clause
  | { ddl_event [or ddl_event]...
    | database_event [or database_event]...
    }
    on { [schema .] schema | database }
  }
  [when ( condition ) ]
  { pl/sql_block | call_procedure_statement }
```

可用的语法选项根据所使用触发器的类型有所不同。例如，DML 事件上的触发器将遵循以下语法使用 dml_event_clause：

```
{ delete | insert | update [of column [, column]...] }
[or { delete | insert | update [of column [, column]...] }]...
on { [schema .] table | [nested table nested_table_column of]
[schema .] view }
[referencing_clause] [for each row]
```

很明显，在设计触发器时有很大的灵活性。before 和 after 关键字指出触发器应该在触发事件之前还是之后执行。如果使用了 instead of 子句，就可以执行触发器的代码来代替执行导

致调用触发器的事件。delete、insert 和 update 关键字(update 关键字可包括一个列列表)指定构成一个触发事件的数据操作类型。在引用新旧列值时，可以使用默认值(“新”和“旧”)，或者使用 referencing 子句指定其他的名称。

当使用 for each row 子句时，触发器应为行级触发器；否则，触发器为语句级触发器。可以使用 when 子句来进一步限定何时执行触发器。在 when 子句中强制实施的限定条件可以包括新旧数据值的检查。

例如，当等级值下降时，假定希望跟踪 BOOKSHELF 表中 Rating 值的变化。首先创建一个存储审计记录的表：

```
drop table BOOKSHELF_AUDIT;
create table BOOKSHELF_AUDIT
(Title         VARCHAR2(100),
Publisher     VARCHAR2(20),
CategoryName  VARCHAR2(20),
Old_Rating    VARCHAR2(2),
New_Rating    VARCHAR2(2),
Audit_Date    DATE);
```

仅当 Rating 值下降时，才执行以下的行级 BEFORE UPDATE 触发器。该示例还说明了 new 关键字的使用方法，它引用该列的新值，而 old 关键字引用该列的旧值。

```
create or replace trigger BOOKSHELF_BEF_UPD_ROW
before update on BOOKSHELF
for each row
when (new.Rating < old.Rating)
begin
  insert into BOOKSHELF_AUDIT
    (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
    (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
end;
/
```

下面把该 create trigger 命令分解成容易理解的几个组成部分。首先，命名触发器：

```
create or replace trigger BOOKSHELF_BEF_UPD_ROW
```

触发器的名称包含它所作用的表名和触发器的类型(参阅 34.4.6 节以获得命名规则的详细信息)。

该触发器作用于 BOOKSHELF 表；它将在 update 事务提交到数据库之前执行：

```
before update on BOOKSHELF
```

因为使用了 for each row 子句，所以该触发器将作用于被 update 语句修改的所有行上。如果不使用该子句，则该触发器将在语句级执行。

```
for each row
```

when 子句进一步限制了触发的条件。触发事件不仅必须是 BOOKSHELF 表的一个 update 操作，而且必须反映 Rating 值的下降趋势：

```
when (new.Rating < old.Rating)
```

下面的程序清单给出的 PL/SQL 代码是触发器体。其中的命令针对 BOOKSHELF 表中每一个满足 when 条件的 update 操作而执行。为确保成功，BOOKSHELF_AUDIT 表必须存在，且触发器的所有者必须已经授予在该表上操作的权限(直接授权，而不是通过角色授权)。在更新 BOOKSHELF 记录前，该示例将 BOOKSHELF 记录中的旧值插入 BOOKSHELF_AUDIT 表中。

```
begin
  insert into BOOKSHELF_AUDIT
    (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
    (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
end;
```

注意：
在 PL/SQL 块中引用 new 和 old 关键字时，应该在它们前面加冒号(:)。

该示例是一个典型的审计触发器。审计活动对于在 BOOKSHELF 表上实施 update 操作的用户完全透明。但是，BOOKSHELF 表的事务处理依赖于该触发器的成功执行。

34.4.1 DML 触发器类型的组合

一个表上针对多个 insert、update 和 delete 命令的触发器可以组合为一个触发器，前提是它们在同一级别上(行级或语句级)。下面的示例显示了一个触发器，当发生 insert 或 update 时，此触发器就被执行。该示例中有几点(以黑体显示)需要注意的地方：

- 触发器的 update 部分仅在更新 Rating 列值时发生。
- 在 PL/SQL 块中使用 if 子句来决定两个命令中哪个命令调用触发器。
- 如前面示例所示，将要更改的列在 dml_event_clause 中指定，而不是在 when 子句中指定。

```
drop trigger BOOKSHELF_BEF_UPD_ROW;

create or replace trigger BOOKSHELF_BEF_UPD_INS_ROW
before insert or update of Rating on BOOKSHELF
for each row
begin
  if INSERTING then
    insert into BOOKSHELF_AUDIT
      (Title, Publisher, CategoryName,
      New_Rating, Audit_Date)
    values
      ( :new.Title, :new.Publisher, :new.CategoryName,
        :new.Rating, Sysdate);
  else -- if not inserting then we are updating the Rating
    insert into BOOKSHELF_AUDIT
      (Title, Publisher, CategoryName,
```

```
      Old_Rating, New_Rating, Audit_Date)
    values
      ( :old.Title, :old.Publisher, :old.CategoryName,
        :old.Rating, :new.Rating, Sysdate);
  end if;
end;
/
```

再来看一下触发器的组成部分。首先，它被命名和标识为(Rating 的)BEFORE INSERT 和 BEFORE UPDATE 触发器，针对每一行(for each row)执行：

```
create or replace trigger BOOKSHELF_BEF_UPD_INS_ROW
before insert or update of Rating on BOOKSHELF
for each row
```

接下来是触发器体。在触发器体的第一部分中(如下面的程序清单所示)，通过 if 子句检查事务处理的类型。有效的事务处理类型是 INSERTING、DELETING 和 UPDATING。在该示例中，触发器检查记录是否正在插入 BOOKSHELF 表中。如果已经插入表中，则执行触发器体的第一部分。触发器体的 INSERTING 部分将记录的新值插入 BOOKSHELF_AUDIT 表中。

```
begin
  if INSERTING then
    insert into BOOKSHELF_AUDIT
      (Title, Publisher, CategoryName,
      New_Rating, Audit_Date)
    values
      (:new.Title, :new.Publisher, :new.CategoryName,
      :new.Rating, Sysdate);
```

然后，检查其他的事务处理类型。在此示例中，由于触发器被执行，因此事务处理必须是 Rating 列的 insert 或 update 操作。因为触发器体前半部分中的 if 子句检查是否是 insert 操作，且触发器只对 insert 和 update 操作执行，所以应该执行触发器体后半部分的唯一条件是 Rating 的 update 操作。因此，不需要任何其他的 if 子句，就可以确定 DML 事件类型。触发器体的这一部分与前面的示例相同：在更新前，行中的旧值写入 BOOKSHELF_AUDIT 表。

```
  else -- if not inserting then we are updating the Rating
    insert into BOOKSHELF_AUDIT
      (Title, Publisher, CategoryName,
      Old_Rating, New_Rating, Audit_Date)
    values
      (:old.Title, :old.Publisher, :old.CategoryName,
      :old.Rating, :new.Rating, Sysdate);
```

以这种方式组合触发器类型有助于在多个开发人员中协调触发器的开发任务，因为它可以合并依赖于一个表的所有数据库事件。

34.4.2 设置插入值

用户可以在 insert 和 update 操作过程中使用触发器设置列值。本章前面的示例将 BOOKSHELF_AUDIT.Audit_Date 值设置为 SYSDATE 函数的结果。还可以使用 update 操作支持不同的应用需求，如存储某个值的大写版本以及由用户输入的大小写混合版本。在这种

情况下，可能已经使表部分地非标准化，以便包括导出数据的列。以大写格式存储该数据(对于该示例在 UpperPerson 列中)允许在查询中使用大写列时，以自然的格式向用户显示数据。

因为该值的大写版本是导出的数据，所以可能与用户输入的列不同步。也就是说，除非应用程序在 insert 操作中提供了某个大写的值，否则当新行输入时该列的值将为 NULL。

为了避免同步问题，可以使用表级触发器。在表上放置一个 BEFORE INSERT 触发器和一个 BEFORE UPDATE 触发器；它们将在行级起作用。如下面的程序清单所示，每次在 BOOKSHELF_ CHECKOUT 中更改 Name 列的值时，此方法将为 UpperName 新设置一个值：

```
alter table BOOKSHELF_CHECKOUT add (UpperName VARCHAR2(25));

create or replace trigger BOOKSHELF_CHECKOUT_BUI_ROW
before insert or update of Name on BOOKSHELF_CHECKOUT
for each row
begin
   :new.UpperName := UPPER(:new.Name);
end;
/
```

在此示例中，触发器体使用 Name 列上的 UPPER 函数确定 UpperName 的值。每次在 BOOKSHELF_CHECKOUT 表中插入一行和每次更新 Name 列时，该触发器就会执行。从而保持 Name 列和 UpperName 列的同步。

34.4.3 维护复制的数据

34.2 节所示的通过触发器设置值的方法可以与第 25 章介绍的远程数据访问方法相结合。就像物化视图一样，可以复制表中的所有行或部分行。

例如，用户可能想要创建并维护应用程序的审计日志的另一个副本。这样做，能够防止单个应用程序删除由多个应用程序创建的审计日志记录。复制审计日志通常用于安全监控。

考虑本章示例使用的 BOOKSHELF_AUDIT 表。可创建另一个表 BOOKSHELF_AUDIT _ DUP，它可能在一个远程数据库中。对于该示例，假定使用一个名为 AUDIT_LINK 的数据库链接将用户与 BOOKSHELF_AUDIT_DUP 表驻留的数据库连接起来(有关数据库链接的详细信息，请参阅第 25 章)。

```
drop table BOOKSHELF_AUDIT_DUP;
create table BOOKSHELF_AUDIT_DUP
(Title           VARCHAR2(100),
Publisher        VARCHAR2(20),
CategoryName     VARCHAR2(20),
Old_Rating       VARCHAR2(2),
New_Rating       VARCHAR2(2),
Audit_Date       DATE);
```

为了自动填充 DOOKSHELF_AUDIT_DUP 表，可将以下触发器放在 BOOKSHELF_ AUDIT 表中：

```
create or replace trigger BOOKSHELF_AUDIT_AFT_INS_ROW
after insert on BOOKSHELF_AUDIT
for each row
begin
```

```
    insert into BOOKSHELF_AUDIT_DUP@AUDIT_LINK
      (Title, Publisher, CategoryName,
       New_Rating, Audit_Date)
      values (:new.Title, :new.Publisher, :new.CategoryName,
       :new.New_Rating, :new.Audit_Date);
  end;
  /
```

如该触发器的标题所示，该触发器对插入 BOOKSHELF_AUDIT 表的每一行执行。它向由 AUDIT_LINK 数据库链接定义的数据库的 BOOKSHELF_AUDIT_DUP 表插入一个记录。AUDIT_LINK 可能指向一个位于远程服务器上的数据库。如果在通过触发器启动的远程插入操作中出现任何问题，那么本地插入将会失败。

对于异步复制需求，则考虑使用物化视图(请参阅第 24 章)。

通过将一行插入 BOOKSHELF 表，可以查看这些触发器的操作：

```
insert into BOOKSHELF
(Title, Publisher, CategoryName, Rating) values
('HARRY POTTER AND THE CHAMBER OF SECRETS',
'SCHOLASTIC','CHILDRENFIC','4');

1 row created.

select Title from BOOKSHELF_AUDIT;

TITLE
--------------------------------------------------
HARRY POTTER AND THE CHAMBER OF SECRETS

select Title from BOOKSHELF_AUDIT_DUP;

TITLE
--------------------------------------------------
HARRY POTTER AND THE CHAMBER OF SECRETS
```

34.4.4 定制错误条件

用户可以在单个触发器内定义不同的错误条件。对于定义的每一个错误条件，可以选取一个在发生错误时显示的错误消息。显示给用户的错误数量和错误消息可以通过 RAISE_APPLICATION_ERROR 过程来设置，此过程可以从任何一个触发器中调用。

下面的示例显示了 BOOKSHELF 表中一个语句级的 BEFORE DELETE 触发器。在用户试图从 BOOKSHELF 表中删除记录时，该触发器被执行，并且检查两个系统条件：第一个条件是该日期既不是星期六也不是星期天，第二个条件是执行 delete 操作的账户的 Oracle 用户名以字母“LIB”开始。下面的程序清单描述了该触发器的组成部分：

```
create or replace trigger BOOKSHELF_BEF_DEL
before delete on BOOKSHELF
declare
  weekend_error EXCEPTION;
  not_library_user EXCEPTION;
begin
if TO_CHAR(SysDate,'DY') = 'SAT' or
   TO_CHAR(SysDate,'DY') = 'SUN' THEN
```

```
  RAISE weekend_error;
end if;
if SUBSTR(User,1,3) <> 'LIB' THEN
  RAISE not_library_user;
end if;
EXCEPTION
  WHEN weekend_error THEN
    RAISE_APPLICATION_ERROR (-20001,
      'Deletions not allowed on weekends');
  WHEN not_library_user THEN
    RAISE_APPLICATION_ERROR (-20002,
      'Deletions only allowed by Library users');
end;
/
```

该触发器的标题将触发器定义为一个语句级的 BEFORE DELETE 触发器：

```
create or replace trigger BOOKSHELF_BEF_DEL
before delete on BOOKSHELF
```

由于在该触发器中没有 when 子句，因此对于所有的删除操作都执行该触发器体。

触发器的下一部分声明在该触发器中定义的两个异常的名字：

```
declare
  weekend_error EXCEPTION;
  not_library_user EXCEPTION;
```

触发器体的第一部分包含一个在 SYSDATE 函数上使用 TO_CHAR 函数的 if 子句。如果当前日期是星期六或星期日，则触发 WEEKEND_ERROR 错误条件。该错误条件称为异常，必须在触发器体内定义。

```
begin
if TO_CHAR(SysDate,'DY') = 'SAT' or 'SAT' or
  TO_CHAR(SysDate,'DY') = 'SUN' THEN
  RAISE weekend_error;
end if;
```

第二个 if 子句检查 User 伪列的前 3 个字符是否是“LIB”。如果用户名不以“LIB”开头，则会触发 NOT_LIBRARY_USER 异常。在本例中，使用了运算符“<>”，它等同于“!=”(表示“不等于”)。

```
if SUBSTR(User,1,3) <> 'LIB' THEN
   RAISE not_library_user;
end if;
```

触发器体的最后一部分告诉触发器如何处理异常。它以关键字 exception 开头，后面是 when 子句，针对每个异常有一个 when 子句。此触发器中的每个异常都调用 RAISE_APPLICATION_ERROR 过程，它接受两个输入参数：错误数量(必须在 - 20 001~ - 20 999 之间)和将要显示的错误消息。在这个示例中，定义了两个不同的错误消息，定义的每个异常显示一个错误消息：

```
EXCEPTION
  WHEN weekend_error THEN
    RAISE_APPLICATION_ERROR (-20001,
```

```
    'Deletions not allowed on weekends');
  WHEN not_library_user THEN
    RAISE_APPLICATION_ERROR (-20002,
      'Deletions only allowed by Library users');
```

在管理可能在触发器中遇到的错误条件时，使用 RAISE_APPLICATION_ERROR 过程会提供很大的灵活性。关于这些过程的详细描述，请参见第 35 章。

注意：
即使 delete 操作没有发现要删除的行，异常也会触发。

当一个非“LIB”用户试图删除 BOOKSHELF 表中日期是工作日的行时，将导致以下结果：

```
delete from BOOKSHELF
where Title = 'MY LEDGER';

delete from BOOKSHELF
            *
ERROR at line 1:
ORA-20002: Deletions only allowed by Library users
ORA-06512: at "PRACTICE.BOOKSHELF_BEF_DEL", line 17
ORA-04088: error during execution of trigger 'PRACTICE.BOOKSHELF_BEF_DEL'
```

如果试图删除日期是星期六或者星期天的行，则会返回 ORA-20001 错误。一旦出现异常，Oracle 就跳出触发器的主要可执行命令部分，去处理异常。用户只能看到遇到的第一个异常。

34.4.5 在触发器中调用过程

无需在触发器体内创建较大的代码块，只需将代码保存为一个存储过程，并在触发器内使用 call 命令调用此存储过程即可。例如，如果您创建一个 INSERT_BOOKSHELF_AUDIT_ DUP 过程，该过程将行插入 BOOKSHELF_ADUIT_DUP 中，则可以用如下程序清单从 BOOKSHELF_AUDIT 表的触发器中调用它：

```
create or replace trigger BOOKSHELF_AFT_INS_ROW
after insert on BOOKSHELF_AUDIT
for each row
begin
call INSERT_BOOKSHELF_AUDIT_DUP(:new.Title, :new.Publisher,
   :new.CategoryName, :new.Old_Rating, :new.New_Rating,
   :new.Audit_Date);
end;
/
```

34.4.6 命名触发器

触发器名应该清楚地标出它所应用的表、触发它的 DML 命令、它的 before/after 状态以及它是行级触发器还是语句级触发器。因为一个触发器的名称不能超过 30 个字符，所以在命名触发器的时候需要使用一组标准的缩写。一般说来，触发器的名称应尽量包含表名。因

此，当在一个名为 BOOKSHELF_CHECKOUT 的表上创建一个行级触发器 BEFORE UPDATE 时，不应该命名此触发器为 BEFORE_UPDATE_ROW_LEVEL_BC。使用名称 BOOKSHELF_CHECKOUT_BEF_UPD_ROW 会更好一些。

34.4.7 创建 DDL 事件触发器

用户可以创建在 DDL 事件发生时执行的触发器。如果仅是出于安全目的而使用该功能，则应当考虑使用 audit 命令。例如，可以使用 DDL 事件触发器执行在群集、函数、索引、程序包、过程、角色、序列、同义词、表、表空间、触发器、类型、用户或视图上执行的 create、alter 和 drop 命令的触发器代码。如果使用 on schema 子句，则触发器将对在模式中创建的任何一个新的数据字典对象执行。只要对象是在用户模式中创建的，下面的示例都将执行一个名为 INSERT_AUDIT_RECORD 的过程：

```
create or replace trigger CREATE_DB_OBJECT_AUDIT
after create on schema
begin
   call INSERT_AUDIT_RECORD (ora_dict_obj_name);
end;
/
```

如该示例所示，可以引用诸如对象名这样的系统属性。表 34-1 列出了可用的属性。

表 30-1 系统定义的事件属性

属 性	类 型	说 明	示 例
ora_client_ip_address	VARCHAR2	当基本协议为 TCP/IP 时，返回 LOGON 事件中客户机的 IP 地址	if (ora_sysevent = 'LOGON') then addr := ora_client_ip_address; end if;
ora_database_name	VARCHAR2(50)	数据库名	Declare db_name VARCHAR2(50); begin db_name := ora_database_name; end;
ora_des_encrypted_password	VARCHAR2	正被创建或更改的用户的 DES 加密密码	if (ora_dict_obj_type = 'USER') then insert into event_table (ora_des_encrypted_password); end if;
ora_dict_obj_name	VARCHAR(30)	发生 DDL 操作的字典对象的名称	insert into event_table ('Changed object is ' || ora_dict_obj_name');
ora_dict_obj_name_list(name_list OUT ora_name_list_t)	BINARY_INTEGER	返回在事件中正在被更改对象的对象名列表	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_modified := ora_dict_obj_name_list (name_list); end if;
ora_dict_obj_owner	VARCHAR(30)	发生 DDL 操作的字典对象的所有者	insert into event_table ('object owner is' || ora_dict_obj_owner');

(续表)

属 性	类 型	说 明	示 例
ora_dict_obj_owner_list(owner_list OUT ora_name_list_t)	BINARY_INTEGER	返回在事件中正在被修改对象的对象所有者列表	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_of_modified_objects := ora_dict_obj_owner_list(owner_list); end if;
ora_dict_obj_type	VARCHAR(20)	发生DDL操作的字典对象的类型	insert into event_table ('This object is a ' || ora_dict_obj_type);
ora_grantee(user_list OUT ora_name_list_t)	BINARY_INTEGER	返回OUT参数中一个授权事件的被授予者；返回返回值中被授予者的数量	if (ora_sysevent = 'GRANT') then number_of_users := ora_grantee(user_list); end if;
ora_instance_num	NUMBER	实例编号	if (ora_instance_num = 1) then insert into event_table ('1'); end if;
ora_is_alter_column (column_name IN VARCHAR2)	BOOLEAN	如果指定的列被更改，则返回TRUE	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then alter_column := ora_is_alter_column('FOO'); end if;
ora_is_creating_nested_table	BOOLEAN	如果当前事件正在创建一个嵌套表，则返回TRUE	if (ora_sysevent = 'CREATE' and ora_dict_obj_type = 'TABLE' and ora_is_creating_nested_table) then insert into event_tab values ('A nested table is created'); end if;
ora_is_drop_column (column_name IN VARCHAR2)	BOOLEAN	如果指定的列被删除，则返回TRUE	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_drop_column('FOO'); end if;
ora_is_servererror	BOOLEAN	如果指定的错误在错误栈中，就返回TRUE；否则返回FALSE	if (ora_is_servererror(error_number)) then insert into event_table ('Server error!!'); end if;
ora_login_user	VARCHAR2(30)	登录用户名	select ora_login_user from dual;
ora_partition_pos	BINARY_INTEGER	在CREATE TABLE的INSTEAD OF触发器中，可以在SQL文本中插入PARTITION子句的位置	-- Retrieve ora_sql_txt into -- sql_text variable first. n := ora_partition_pos; new_stmt := substr(sql_text, 1, n-1) || ' ' || my_partition_clause || ' ' || substr(sql_text, n));
ora_privilege_list (privilege_list OUT ora_name_list_t)	BINARY_INTEGER	返回被授予者授予的权限列表或OUT参数中被撤消者被撤消的权限列表；返回返回值中的权限数目	if (ora_sysevent = 'GRANT' or ora_sysevent = 'REVOKE') then number_of_privileges := ora_privilege_list(priv_list); end if;

(续表)

属 性	类 型	说 明	示 例
ora_revokee (user_list OUT ora_name_list_t)	BINARY_INTEGER	返回OUT参数中撤消事件的被撤消者；返回返回值中的被撤消者的数目	`if (ora_sysevent = 'REVOKE') then number_of_users := ora_revokee(user_list);`
ora_server_error	NUMBER	给定一个位置(1为栈顶)，它返回错误栈中该位置的错误号	`insert into event_table ('top stack error ' || ora_server_error(1));`
ora_server_error_depth	BINARY_INTEGER	返回错误栈中错误消息的总数	`n := ora_server_error_depth; -- This value is used with -- other functions such as -- ora_server_error`
ora_server_error_msg (position in binary_integer)	VARCHAR2	给定一个位置(1为栈顶)，它返回错误栈中该位置的错误消息	`insert into event_table ('top stack error message' || ora_server_error_msg(1));`
ora_server_error_num_params (position in binary_integer)	BINARY_INTEGER	给定一个位置(1为栈顶)，它返回利用格式“%s”已经替换为错误消息的串数目	`n := ora_server_error_num_params(1);`
ora_server_error_param (position in binary_integer, param in binary_integer)	VARCHAR2	给定一个位置(1为栈顶)和一个参数号码，它返回错误消息中与“%s”、“%d”等匹配的替换值	`-- E.g. the 2nd %s in a message -- like "Expected %s, found %s" param := ora_server_error_param(1,2);`
ora_sql_txt (sql_text outora_name_list_t)	BINARY_INTEGER	返回OUT参数中触发语句的SQL文本。如果该语句很长，则分成多个PL/SQL表元素。该函数的返回值指定PL/SQL表中有多少元素	`--... -- Create table event_table create table event_table (col VARCHAR2(2030)); --... DECLARE sql_text DBMS_STANDARD.ora_name_list_t; n PLS_INTEGER; v_stmt VARCHAR2(2000); BEGIN n := ora_sql_txt(sql_text); FOR i IN 1..n LOOP v_stmt := v_stmt || sql_text(i); END LOOP; INSERT INTO event_table VALUES ('text of triggering statement: ' || v_stmt); END;`
ora_sysevent	VARCHAR2(20)	激活触发器的系统事件：事件名与语法中的名称相同	`Insert into event_table (ora_sysevent);`
ora_with_grant_option	BOOLEAN	如果权限用 with grant option 授予，则返回TRUE	`If (ora_sysevent = 'GRANT' and ora_with_grant_option = TRUE) then insert into event_table ('with grant option'); end if;`

(续表)

属 性	类 型	说 明	示 例
space_error_info(error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)	BOOLEAN	如果错误与空间不足有关，则返回 TRUE，并将引起该错误的对象的相关信息填入 OUT 参数中	IF (space_error_info(eno,typ,owner,ts,obj,subobj) =TRUE) then.put_line('The object ' || run out of space.'); dbms_outputobj || ' owned by ' || owner || ' has end if;

为了保护模式中的对象，用户可能想为每个 drop table 命令的执行创建一个触发器。该触发器必定是一个 BEFORE DROP 触发器：

```
create or replace trigger PREVENT_DROP
  before drop on Practice.schema
begin
  if ora_dict_obj_owner = 'PRACTICE'
    and ora_dict_obj_name like 'BOO%'
    and ora_dict_obj_type = 'TABLE'
  then
    RAISE_APPLICATION_ERROR (
      -20002, 'Operation not permitted.');
  end if;
end;
/
```

需要注意的是，触发器在其 if 子句内引用事件属性。如果试图在 Practice 模式下删除名称以 BOO 开头一个的表，则结果如下所示：

```
drop table BOOKSHELF_AUDIT_DUP;

drop table BOOKSHELF_AUDIT_DUP
*
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20002: Operation not permitted.
ORA-06512: at line 6
```

应该验证预期的 drop 操作是否成功完成。

可以使用 RAISE_APPLICATION_ERROR 过程来进一步定制显示给用户的错误消息。

34.4.8 创建数据库事件触发器

像 DML 事件一样，数据库事件也能执行触发器。在数据库事件发生(关闭、启动或出错)时，可以执行引用事件属性的触发器(就像对 DDL 事件一样)。可以在每个数据库启动后，立

即用数据库事件触发器执行系统维护功能。

例如，下面的触发器把程序包固定到每个数据库的启动上。固定程序包(pinning package)是一种在内存共享池中保存大 PL/SQL 对象的有效方式，它可以改善数据库的性能并增强数据库的稳定性。该触发器 PIN_ON_STARTUP 将在数据库每次启动时运行。只有以具有 ADMINISTER DATABASE TRIGGER 权限的用户连接数据库后，才能创建该触发器。

```
rem while connected as a user with the
rem ADMINISTER DATABASE TRIGGER system privilege:

create or replace trigger PIN_ON_STARTUP
after startup on database
begin
  DBMS_SHARED_POOL.KEEP (
    'SYS.STANDARD', 'P');
end;
/
```

此示例给出了一个将在数据库启动后立即执行的简单触发器。可以修改触发器体中的程序包列表，以便包含应用程序经常使用的那些程序包。

启动触发器和关闭触发器可以访问表 34-1 中列出的 ora_instance_num、ora_database_name、ora_login_user 和 ora_sysevent 属性。有关 createt trigger 命令的完整语法，请参阅附录 A。

34.4.9 创建复合触发器

从 Oracle 11g 开始，可以创建这样的触发器，即它针对下面每个时间点都有一部分：语句之前(BEFORE STATEMENT)、每行之前(BEFORE EACH ROW)、每行之后(AFTER EACH ROW)和语句之后(AFTER STATEMENT)。将语句级和行级触发器命令综合起来可以使用户执行复杂的命令，在执行处理期间，这些命令共享语句级和行级的数据。从而可以避免变异表错误，并能提高性能。

复合触发器对它的每个组件区域(语句之前、语句之后、行之前、行之后)都有一部分。所有这些部分都可以访问共享的 PL/SQL 状态信息，如变量值。当触发语句完成时，这些状态信息不再可用。

要创建复合触发器，可以使用 create trigger 命令的 COMPOUND TRIGGER 子句，如下面的程序清单所示：

```
create or replace trigger BOOKSHELF_AUDIT_COMPOUND
after insert on BOOKSHELF_AUDIT
compound trigger
--declare and set variable values here
BEFORE STATEMENT IS
-- statement level commands
end BEFORE STATEMENT;
AFTER EACH ROW IS
begin
  insert into BOOKSHELF_AUDIT_DUP@AUDIT_LINK
    (Title, Publisher, CategoryName,
     New_Rating, Audit_Date)
    values (:new.Title, :new.Publisher, :new.CategoryName,
     :new.New_Rating, :new.Audit_Date);
end;
/
```

您可以声明将要在复合触发器的所有组件中共享的变量。例如，BEFORE STATEMENT触发器可以执行一个计算某一列的平均值的查询。然后通过共享变量将此平均值传递给同一触发器的 AFTER EACH ROW 部分。

注意：
复合触发器只能被 DML 语句触发。

要使用复合触发器，必须遵守下面的限制条件：

- 复合触发器的主体必须是复合触发器块。
- 复合触发器必须是 DML 触发器。
- 复合触发器必须定义在表或视图上。
- 声明部分不能包含 PRAGMA AUTONOMOUS_TRANSACTION。
- 由于复合触发器的主体不能有初始化块，因此它不能有异常部分。这不是什么问题，因为 BEFORE STATEMENT 部分始终是在任何其他时间点执行之前执行一次。
- 在某一部分发生的异常必须在这一部分进行处理，不能将控制权移交给另一部分。
- 如果某一部分包含 GOTO 语句，则 GOTO 语句的目标必须在同一部分中。
- “:OLD”、“:NEW”和“:PARENT”不能出现在声明部分、BEFORE STATEMENT 部分或 AFTER STATEMENT 部分。
- 只有 BEFORE EACH ROW 部分能改变:NEW 的值。
- 如果激活复合触发器之后，触发语句由于 DML 异常而回滚，则：
  - 复合触发器部分中声明的本地变量被重新初始化，于是迄今为止计算的任何值都会丢失。
  - 由激活复合触发器而产生的副作用不回滚。
- 复合触发器的激活顺序没有保证。复合触发器的激活可以与简单触发器的激活交替进行。
- 如果用 FOLLOWS 选项指定了激活复合触发器的顺序，并且如果 FOLLOWS 的目标不包含作为源代码的相应部分，则可以忽略这一顺序。

34.5 启用和禁用触发器

与声明完整性约束(如 NOT NULL 和 PRIMARY KEY)不同，触发器并不一定影响表中所有的行。它们只影响特定类型的操作，并且仅在启用触发器时影响。在创建触发器前创建的任何数据都不受该触发器的影响。

在默认情况下，触发器在创建时即被启用。可是，在有些情况下，用户可能希望禁用触发器。两个最常见的原因都涉及数据加载。在大型数据加载过程中，用户可能希望禁用在加载期间执行的触发器。在数据加载期间禁用触发器，可以极大地提高加载性能。在数据加载完毕后，必须手工执行数据处理工作。如果在数据加载时启用触发器，则这些数据处理工作将由触发器完成。

在数据加载失败且必须进行第二次加载时，也需要禁用触发器，这是与数据加载有关的禁用触发器的第二个原因。在这种情况下，有可能部分数据已成功加载——因此触发器将由于部分记录的加载而被触活。在随后的加载过程中，可能会插入相同的记录。所以有可能会为相同的插入操作两次执行同一个触发器(当该插入操作在两次加载过程中都出现时)。根据操作和触发器的性质，这并不理想。如果在失败的加载过程中启用了触发器，则在第二次数据加载开始前可能需要禁用它。在数据加载完成后，必须手工完成相应的数据处理工作。如果在数据加载中启用触发器，则这些数据处理工作将由触发器来完成。

为启用触发器，可使用带 enable 关键字的 alter trigger 命令。要使用该命令，必须拥有表，或者拥有 ALTER ANY TRIGGER 系统权限。alter trigger 样本命令如下所示：

```
alter trigger BOOKSHELF_BEF_UPD_INS_ROW enable;
```

启用触发器的第二种方法是使用带有 enable all triggers 子句的 alter table 命令。使用此命令有可能无法启用特定的触发器；还必须使用 alter trigger 命令。下面的示例显示了 alter table 命令的用法：

```
alter table BOOKSHELF enable all triggers;
```

为使用 alter table 命令，必须拥有表，或者拥有 ALTER ANY TABLE 系统权限。

也可以使用相同的基本命令禁用触发器(需要相同的权限)，但子句需要修改。对于 alter trigger 命令，可使用 disable 子句：

```
alter trigger BOOKSBELF_BEF_UPD_INS_ROW disable;
```

对于 alter table 命令，可使用 disable all triggers 子句：

```
alter table BOOKSHELF disable all triggers;
```

用户可以手工编译触发器。可以使用 alter trigger compile 命令手工编译现有的但已经失效的触发器。因为触发器有依赖性，所以如果触发器所依赖的某个对象改变，触发器就会失效。在触发器重新编译时，alter trigger debug 命令允许重新生成 PL/SQL 信息。

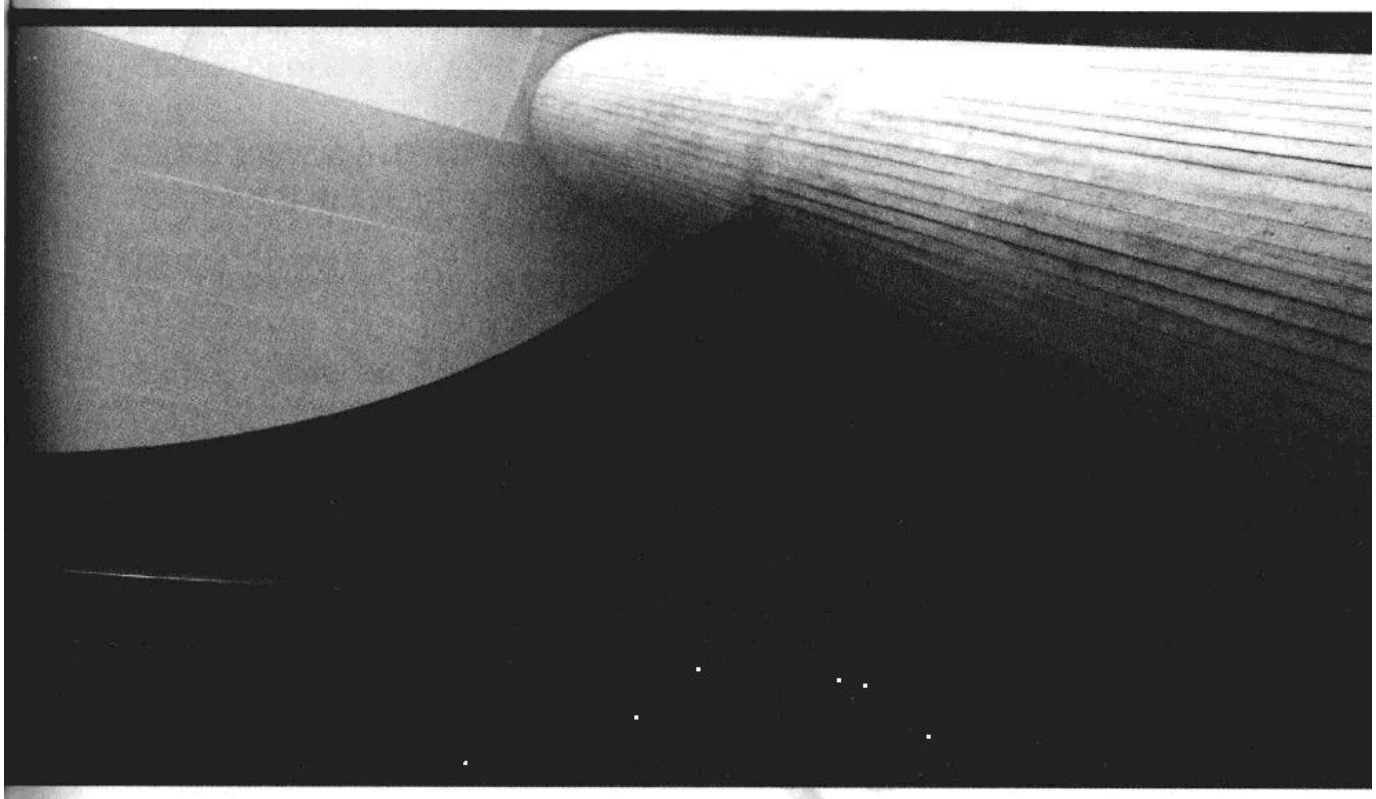
34.6 替换触发器

触发器的状态是唯一可更改的部分。为了更改触发器的主体，必须重新创建或替换此触发器。在替换触发器时，使用 create or replace trigger 命令(请参阅 34.4 节和相关示例)。

34.7 删除触发器

触发器可通过 drop trigger 命令来删除。为删除触发器，必须拥有该触发器，或者具有 DROP ANY TRIGGER 系统权限。下面是该命令的一个示例：

```
drop trigger BOOKSHELF_BEF_UPD_INS_ROW;
```



第35章

过程、函数与程序包

复杂的业务规则和应用逻辑可以存储为 Oracle 内的过程。存储过程(成组的 SQL、PL/SQL 和 Java 语句)能够将实现业务规则的代码从应用程序中移植到数据库中。因此，多个应用程序所使用的代码只需存储一次即可。因为 Oracle 支持存储过程，所以应用程序内的代码更加一致，也更容易维护。

注意：

关于 Java 及其 Java 在存储过程中应用的详细内容，请参阅第 42~44 章。本章将集中讨论 PL/SQL 过程。

用户可将过程和其他 PL/SQL 命令组织成程序包(Package)。以下几节将介绍程序包、过程和函数(可向用户返回值的过程)的实现细节和建议。

使用过程可以获得性能上的改善，两点原因如下：

- 能够在数据库内(从而可在服务器上)进行复杂的业务规则处理。在客户端/服务器或 3 层体系结构的应用程序中，把复杂的处理从应用程序(在客户端上)转移到数据库(在服务器上)中会极大地改善性能。
- 由于过程化代码存储在数据库中，而且相对处于静态，因此在数据库内复用相同查询有助于提高系统性能。系统全局区(SGA)中的共享的 SQL 区将存储解析版本已执行的命令。因此，当再次执行一个过程时，它能够利用前面完成的解析，从而改善过程的执行性能。

除了上述优点外，应用程序开发工作也将大大受益。合并进数据库中的业务规则不再需要写入每个应用程序中，从而节省了应用程序创建的时间，并简化了维护过程。

35.1 必需的系统权限

要创建过程对象，必须拥有 CREATE PROCEDURE 系统权限。如果过程对象位于其他用户模式中，则必须拥有 CREATE ANY PROCEDURE 系统权限。

一旦创建了过程对象，就可以执行它。在执行过程对象时，它可以依赖其所有者的表权限，或者依赖于执行它的用户的权限。当用定义者的权限创建过程时，执行过程的用户无需具有过程所访问的表的访问权限。如果过程依靠调用者的权限，则用户必须有权访问该过程所访问的所有数据库对象。除非另行说明，否则本章的示例都假定用户在所有者权限下执行过程。

为了允许其他用户执行过程对象，应该给他们授予在该对象上的 EXECUTE 权限，如下例所示：

```
grant execute on  MY_PROCEDURE to Dora;
```

现在，用户 Dora 能够执行名为 MY_PROCEDURE 的过程。如果没有将 EXECUTE 权限授予用户，则必须具有 EXECUTE ANY PROCEDURE 系统权限才能执行相应的过程。

在执行过程时，一般要给过程传递一些变量。例如，与 BOOKSHELF 表交互的某个过程可能接受 Title 列的一个值作为其输入。在这个示例中，假定该过程在接收到一本新书(来自 BOOK_ORDER 表的图书列表)后，在表 BOOKSHELF 中新建一条记录。这个过程可从数据库中的任意应用程序调用(只要调用此过程的用户已经被授予相应的 EXECUTE 权限)。

用来执行过程的语法取决于调用该过程的环境。在 SQL*Plus 中，可使用 execute 命令来执行一个过程，execute 命令后面是一个过程名。传递给该过程的任意参数必须括在该过程名后的圆括号中，如下例所示(该示例使用了一个名为 NEW_BOOK 的过程)：

```
execute NEW_BOOK( 'ONCE REMOVED' );
```

此命令将执行 NEW_BOOK 过程，将值 ONCE REMOVED 传递给它。

不使用 execute 命令，也可以从其他过程、函数、程序包或触发器内调用此过程。如果从 BOOK_ORDER 表上的某个触发器调用 NEW_BOOK 过程，则该触发器的主体可以包含下面的命令：

```
NEW_BOOK(: new.Title );
```

该命令将以 Title 列的新值作为输入，来执行 NEW_BOOK 过程(关于触发器中旧值和新值的使用，请参阅第 34 章)。

为执行另一个用户拥有的过程，必须创建该过程的一个同义词，或者在执行期间引用所有者的姓名，如下面的程序清单所示：

```
execute Practice.NEW_BOOK('ONCE REMOVED' );
```

此命令将执行 Practice 模式中的 NEW_BOOK 过程。

另外，也可以使用下面的命令创建该过程的一个同义词：

```
create synonym NEW_BOOK for Practice.NEW_BOOK;
```

这样，此同义词的所有者在执行此过程时就不再需要引用过程的所有者了。简单地输入以下命令即可：

```
execute NEW_BOOK( 'ONCE REMOVED' );
```

其中的同义词将指向相应的过程。

在执行远程过程时，必须指定数据库链接名(关于数据库链接的信息，请参阅第 25 章)。数据库链接名必须紧跟在过程名之后，变量之前，如下例所示：

```
execute NEW_BOOK@REMOTE_CONNECT( 'ONCE REMOVED' );
```

此命令使用 REMOTE_CONNECT 数据库链接访问远程数据库中名为 NEW_BOOK 的过程。

为了使过程的位置对用户透明，可以为远程过程创建同义词，如下面的示例所示：

```
create synonym NEW_BOOK
    for NEW_BOOK@REMOTE_CONNECT;
```

一旦创建该同义词，用户就可以使用此同义词的名称来引用远程过程了。Oracle 假定所有远程过程调用都涉及对远程数据库的更新。

35.2　必需的表权限

过程对象可以引用表。为了正确运行对象，过程的所有者、程序包或正在执行的函数必须在所使用的表上具有相应的权限。除非使用调用者权限，否则执行相应过程对象的用户不需要在这些基表上具有权限。

注意：

过程、程序包和函数所需的权限不能来自角色；必须将权限直接授予过程、程序包或函数的所有者。

35.3 过程与函数

过程不一定给调用者返回值。而函数可以给调用者返回值，并且函数可以在查询中直接引用。该值是通过在函数内使用 return 关键字返回的。函数的示例在 35.6 节中给出。

35.4 过程与程序包

程序包是由成组的过程、函数、变量和 SQL 语句组成的一个单元。要执行程序包中的某个过程，首先必须列出程序包名，然后列出过程名，如下例所示：

```
execute BOOK_INVENTORY.NEW_BOOK( 'ONCE REMOVED' );
```

这里，执行了 BOOK_INVENTORY 程序包中的 NEW_BOOK 过程。

程序包允许多个过程使用相同的变量和游标。程序包中的过程可以是公有的(如前面示例中的 NEW_BOOK 过程)也可以是私有的。在私有情况下，只能通过程序包中的命令访问它们(如从其他过程调用)。程序包的示例在 35.7 节中给出。

程序包还包括每次调用该程序包时所执行的命令，而不管在程序包内调用的是过程还是函数。因此，程序包不但可以组合过程，而且还提供了执行非特定过程命令的能力。35.7 节给出了每次调用程序包时都会执行的一个代码示例。

35.5 create procedure 语法

附录 A 给出了 create procedure 命令的完整语法。此命令的简短语法如下：

```
create [or replace] procedure [schema .] procedure
[( argument [ in | out | in out ] [nocopy] datatype
   [, argument [ in | out | in out ] [nocopy] datatype]...
)]
[authid { current_user | definer }]
{ is | as } { pl/sql_subprogram_body |
  language { java name 'string'| c [name name] library lib_name
}};
```

过程的标题和主体都由该命令创建。

NEW_BOOK 过程是由下面的程序清单创建的：

```
create or replace procedure NEW_BOOK (aTitle IN VARCHAR2,
  aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
as
begin
  insert into BOOKSHELF (Title, Publisher, CategoryName, Rating)
```

```
    values (aTitle, aPublisher, aCategoryName, NULL);
  delete from BOOK_ORDER
    where Title = aTitle;
end;
/
```

NEW_BOOK 过程以书名、出版商和分类作为其输入。可以从任何一个应用程序调用此过程。此过程把一条记录插入 BOOKSHELF 表中，Rating 列有一个 NULL 值，并从 BOOK_ORDER 表中删除相匹配的 Title。

如果一个过程已经存在，则可以通过 create or replace procedure 命令替换它。使用该命令(而不是删除和重建旧过程)的好处是以前在此过程上授予的 EXECUTE 权限将保持不变。

调用此过程时，对必须指定值的参数要使用 IN 限定符。在 NEW_BOOK 示例中，aTitle、aPublisher 和 aCategoryName 参数都声明为 IN。OUT 限定符表明该过程将通过此参数把一个值返回给调用者。IN OUT 限定符表明参数同时为 IN 和 OUT：在调用过程时必须给此参数指定一个值，且此过程将通过该参数把一个值返回给调用者。如果不指定限定符的类型，则默认值为 IN。

在 create procedure 语法中，authid 指身份验证的类型，即定义者权限(默认)或调用者权限(current_user)。nocopy 关键字告诉 Oracle 将变量的值尽快传递给用户。

在默认条件下，过程由一个用 PL/SQL 编写的代码块组成(代码块指的是调用时，过程将执行的代码)。在 NEW_BOOK 示例中，代码块如下所示：

```
begin
  insert into BOOKSHELF (Title, Publisher, CategoryName, Rating)
    values (aTitle, aPublisher, aCategoryName, NULL);
  delete from BOOK_ORDER
    where Title = aTitle;
end;
```

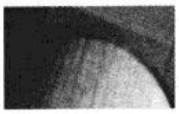
这里给出的 PL/SQL 代码块非常简单，由一条 SQL 语句组成。过程内的 PL/SQL 代码块可以包括任何 DML 语句；PL/SQL 代码块不能用于 DDL 语句(如 create view)。

此外，language 指用来编写代码的语言。关于 Java 的示例，请参阅第 44 章。

以下程序清单说明了 NEW_BOOK 过程的执行结果，以及 BOOK_ORDER 表和 BOOKSHELF 表相应的变化。

```
select Title from BOOK_ORDER;

TITLE
--------------------------------------------------------------------
GALILEO'S DAUGHTER
GOSPEL
LONGITUDE
ONCE REMOVED
SHOELESS JOE
SOMETHING SO STRONG
```

```
execute NEW_BOOK('ONCE REMOVED','SANCTUARY PUB','ADULTNF');

PL/SQL procedure successfully completed.

select Title from BOOK_ORDER;
TITLE
--------------------------------------------------------------
GALILEO'S DAUGHTER
GOSPEL
LONGITUDE
SHOELESS JOE
SOMETHING SO STRONG

select * from BOOKSHELF
  where Title = 'ONCE REMOVED';

TITLE
--------------------------------------------------------------
PUBLISHER          CATEGORYNAME          RA
------------------ --------------------  --
ONCE REMOVED
SANCTUARY PUB      ADULTNF
```



注意：
用户可以通过创建外部过程来使外部 C 过程和 Java 库在 PL/SQL 内可用。必须在网络配置文件中建立一个代理(默认名为 extproc)。如果需要引用外部库，则请与数据库管理员一同配置网络环境和所需的代理，并在数据库中创建目录和库。

35.6 create function 语法

create function 命令的语法比 create procedure 命令的语法更复杂，create function 命令的语法大体如下：

```
create [or replace] function [schema .] function
[( argument [ in | out | in out ] [nocopy] datatype
   [, argument [ in | out | in out ] [nocopy] datatype]...
)]
return datatype
[{ invoker_rights_clause | deterministic | parallel_enable_clause }
  [ invoker_rights_clause | deterministic | parallel_enable_clause ]...
]
{ { aggregate | pipelined } using [schema .] implementation_type
| [pipelined] { is | as } { pl/sql_function_body | call_spec }};
```

函数头和函数体都通过此命令创建。

return 关键字指定函数的返回值的数据类型。它可以是任何有效的 PL/SQL 数据类型(参见第 32 章)。每个函数必须有一个 return 子句。因为根据定义，函数必须给调用环境返回一个值。

下面的示例说明了一个名为 OVERDUE_CHARGES 的函数，它根据对 BOOKSHELF _ CHECKOUT 表的计算结果，返回某人的过期图书滞纳金。输入为此人的名字，而输出为此人的结算结查。

```
create or replace function OVERDUE_CHARGES (aName IN VARCHAR2)
  return NUMBER
  is
  owed_amount NUMBER(10,2);
begin
   select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
   into owed_amount
      from BOOKSHELF_CHECKOUT
    where Name = aName;
   RETURN(owed_amount);
end;
/
```

首先，给此函数命名并指定其输入：

```
create or replace function  OVERDUE_CHARGES(aName IN VARCHAR2)
```

接下来，定义将要返回的值的特征。对于要返回其值的变量，它的定义包括 3 部分：数据类型、变量名和长度。此示例中的数据类型通过 return NUMBER 子句设置。然后，命名此变量为 owed_amount，且定义为 NUMBER(10,2)。这样，变量的 3 部分：数据类型、变量名和长度就定义好了。

```
return NUMBER
is
owed_amount NUMBER(10,2);
```

接着是 PL/SQL 块。在 SQL 语句中，计算所有滞纳金的和(超过 14 天，每天为 0.20 美元)。计算结果放入 owed_amount(前面已定义)变量中。然后，RETURN(owed_amount)命令行将 owed_amount 变量的值返回给主调程序。

```
begin
   select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
   into owed_amount
      from BOOKSHELF_CHECKOUT
    where Name = aName;
   RETURN(owed_amount);
end;
```

如果函数已经存在，就可以使用 create or replace function 命令来替换它。如果使用 or replace 子句，则以前在函数上所具有的任何 EXECUTE 权限都将保持不变。

如果函数在其他模式中创建，则必须拥有 CREATE ANY PROCEDURE 系统权限。如果不指定任何模式，则函数将在用户的模式中创建。要在模式中创建一个函数，必须具有 CREATE PROCEDURE 系统权限(该权限是 RESOURCE 角色的组成部分)。拥有了创建过程的权限，就拥有了创建函数和程序包的权限。

可以通过创建一个变量并设置其值等于函数的返回值来检查该函数：

```
variable owed number;
execute :owed := OVERDUE_CHARGES('FRED FULLER');

PL/SQL procedure successfully completed.
```

可以通过输出该变量值来验证结果：

```
print owed

      OWED
----------
       3.8
```

35.6.1 在过程中引用远程表

使用过程中的 SQL 语句，可以访问远程表。通过过程中的数据库链接，可以查询远程表，如下面的示例所示。在此示例中，NEW_BOOK 过程将一条记录插入由 REMOTE_CONNECT 数据库链接定义的数据库中的 BOOKSHELF 表中，同时从 BOOK_ORDER 表的本地副本中删除该记录。

注意：
下面的示例要求 REMOTE_CONNECT 数据库链接存在，且远程数据库包含被引用对象。

```
create or replace procedure NEW_BOOK (aTitle IN VARCHAR2,
    aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
as
begin
    insert into BOOKSHELF@REMOTE_CONNECT
      (Title, Publisher, CategoryName, Rating)
       values (aTitle, aPublisher, aCategoryName, NULL);
    delete from BOOK_ORDER
       where Title = aTitle;
end;
/
```

过程还可以使用本地同义词。例如，可以为远程表创建一个本地同义词，如下所示：

```
create synonym BOOKSHELF_REMOTE for BOOKSHELF@REMOTE_CONNECT;
```

然后，可以重写过程，以删除数据库链接说明：

```
create or replace procedure NEW_BOOK (aTitle IN VARCHAR2,
    aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
as
begin
    insert into BOOKSHELF_REMOTE
      (Title, Publisher, CategoryName, Rating)
       values (aTitle, aPublisher, aCategoryName, NULL);
    delete from BOOK_ORDER
       where Title = aTitle;
end;
/
```

从过程中删除数据库链接名允许用户从过程中删除表的物理位置中的所有细节。如果表改变位置，则只有同义词会改变，而过程仍然有效。

35.6.2　调试过程

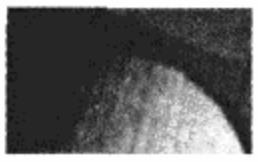
SQL*Plus 的 show errors 命令显示所有与最近创建的过程对象有关的错误消息。该命令检查 USER_ERRORS 数据字典视图找出与最近试图编译该对象有关的错误。show errors 显示每个错误的行号和列号，以及错误消息的文本。

要查看与以前创建的过程对象有关的错误，可以直接查询 USER_ERRORS 数据字典视图，如下面的程序清单所示。此示例查询 USER_ERRORS 数据字典视图，以找出在本章前面创建 OVERDUE_CHARGES 函数期间遇到的错误消息。如果发现任何错误，则查询将会返回导致错误条件的代码行。

```
select Line,      /*Line number of the error. */
       Position,  /*Column number of the error.*/
       Text /*Text of the error message.*/
  from USER_ERRORS
where Name = 'OVERDUE_CHARGES'
  and Type = 'FUNCTION'
order by Sequence;
```

Type 列的有效值为 VIEW、PROCEDURE、PACKAGE、FUNCTION 和 PACKAGE BODY。

另外两个数据字典视图级别——ALL 和 DBA——也可以用来检索与过程对象相关的错误信息。关于这些视图的信息，请参阅第 45 章。



注意：

从 Oracle 11g 开始，您可以使用 PL/SQL Hierarchical Profiler 按子程序调用来报告动态执行时间。它的输出显示除去 SQL 执行时间之外执行 PL/SQL 步骤的时间。配置文件程序包 DBMS_HPROF 为每个函数调用生成相应的输出。有关输出的生成和解释的详细信息请参见此程序包的文档。有关 PL/SQL 过程性能调整的详细信息请参见第 37 章。

除了 show errors 命令所提供的调试信息之外，还可以使用 DBMS_OUTPUT 程序包，这是创建 Oracle 数据库时自动安装的程序包之一。

为了使用 DBMS_OUTPUT，必须在执行将要调试的过程对象之前，执行 set serveroutput on 命令。

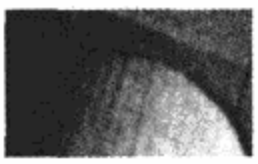
DBMS_OUTPUT 允许在程序包中使用 3 个调试函数：

- PUT　　多个输出结果在同一行显示
- PUT_LINE　　每个输出结果占一行
- NEW_LINE　　与 PUT 一起使用；表示当前输出行的结束

PUT 和 PUT_LINE 用于生成用户想要显示的调试信息。例如，如果正在调试一个包含循环(参阅第 32 章)的过程，那么可能想要跟踪每一次循环中变量的变化情况。为跟踪变量的值，可以使用类似于下面程序清单中的命令。在下例中，输出 Owed_Amount 列的值，用文字串“Owed:”作为前缀。

```
DBMS_OUTPUT.PUT_LINE('Owed:' || Owed_Amount );
```

要使用 DBMS_OUTPUT，首先应该在会话中使用 set serveroutput on 命令。虽然也可以在循环外使用 PUT 和 PUT_LINE，但是，这种用法在函数使用 return 命令时可能会完成得更好(请参阅 35.6 节)。



注意：

对于附加的调试功能，如果具有 DEBUG CONNECT SESSION 系统权限，就可以使用 DBMS_DEBUG 过程。DEBUG ANY PROCEDURE 系统权限等同于对数据库中所有对象具有 DEBUG 权限。

35.6.3 创建自己的函数

除了通过 execute 命令调用自定义函数外，还可以在 SQL 表达式中使用这些函数。这样能够扩展 SQL 的功能，根据需要定制相应的功能。这种函数的使用方法与 Oracle 提供的 SUBSTR 和 TO_CHAR 等函数的使用方式相同。自定义函数不能在 CHECK 或 DEFAULT 约束中使用，而且不能处理任何数据库值。

用户可以调用独立函数(通过前几节介绍的 create function 命令创建)或者在程序包的说明中声明的函数(将会在 35.7 节介绍)。虽然过程不能从 SQL 中直接调用，但是可以通过自己创建的函数调用。

例如，考虑本章前面所示的 OVERDUE_CHARGES 函数，它计算过期图书的滞纳金总额。它有一个输入变量，即人名。但是，为了查看所有借阅者的 OVERDUE_CHARGES 结果，通常需要为 BOOKSHELF_CHECKOUT 表中的每条记录执行一次此过程。

可以改进滞纳金的计算过程。考虑下面的查询：

```
create view BOOK_BORROWERS
  as select distinct Name from BOOKSHELF_CHECKOUT;

select Name,
       OVERDUE_CHARGES(Name)
from BOOK_BORROWERS;
```

Oracle 返回以下输出结果：

```
NAME                      OVERDUE_CHARGES(NAME)
------------------------- --------------------
DORAH TALBOT                                -.8
EMILY TALBOT                                 .2
FRED FULLER                                 3.8
GERHARDT KENTGEN                            2.8
JED HOPKINS                                 1.4
PAT LAVAY                                   2.6
ROLAND BRANDT                              22.6

7 rows selected.
```

此查询使用自定义 OVERDUE_CHARGES 函数计算所有借阅者的滞纳金。结果不是非常准确，因为该函数把以前已经还的书也计算在内了。可以使用 create or replace function 命令把 where 子句添加到函数的查询中：

```
create or replace function OVERDUE_CHARGES (aName IN VARCHAR2)
  return NUMBER
  is
  owed_amount NUMBER(10,2);
begin
   select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
   into owed_amount
     from BOOKSHELF_CHECKOUT
    where Name = aName
    and ReturnedDate-CheckoutDate > 14;
  RETURN(owed_amount);
end;
/
```

正确的输出结果如下：

```
select Name,
       OVERDUE_CHARGES(Name)
  from BOOK_BORROWERS;

NAME                      OVERDUE_CHARGES(NAME)
------------------------- --------------------
DORAH TALBOT                                .4
EMILY TALBOT                                .8
FRED FULLER                                3.8
GERHARDT KENTGEN                           4.6
JED HOPKINS                                1.4
PAT LAVAY                                  3.4
ROLAND BRANDT                             22.6
```

此查询从 BOOK_BORROWERS 表中选择每一个名字；对于每个名字，它都执行 OVERDUE_CHARGES 函数，该函数从 BOOKSHELF_CHECKOUT 表中选择数据。

为了利用该特性，函数必须遵循与 Oracle 函数相同的指导原则。最值得注意的是，它们不能更新数据库，且只能包含 IN 参数。

35.6.4 定制错误条件

用户可以在过程对象中定义不同的错误条件(关于在触发器中定制错误条件的示例，请参阅第 34 章)。对于定义的每一个错误条件，可以选择错误发生时显示的错误消息。可以通过 RAISE_APPLICATION_ERROR 过程设置用于向用户显示的错误编号和错误消息，该过程可以在任何过程对象中调用。

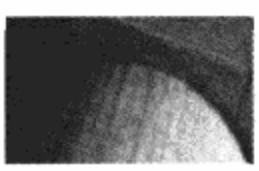
可以从过程、程序包和函数中调用 RAISE_APPLICATION_ERROR。它要求两个输入：消息编号和消息文本。可以指定显示给用户的消息编号和消息文本。这极大地丰富了 PL/SQL 中可用的标准异常(请参阅附录 A 中的“EXCEPTION”项)。

下面的示例显示了本章前面定义的 OVERDUE_CHARGES 函数。可是，现在它另外增加了一部分(以粗体字显示)。标题为 EXCEPTION 的部分告诉 Oracle 如何处理非标准过程。在这个示例中，通过 RAISE_APPLICATION_ERROR 过程重写 NO_DATA_FOUND 异常的标准消息。

```
create or replace function OVERDUE_CHARGES (aName IN VARCHAR2)
  return NUMBER
  is
  owed_amount NUMBER(10,2);
begin
    select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
    into owed_amount
    from BOOKSHELF_CHECKOUT
  where Name = aName
    and (ReturnedDate-CheckoutDate) > 14;
  RETURN(owed_amount);
EXCEPTION
  when NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20100,
    'No books borrowed.');
end;
/
```

前面的示例使用了 NO_DATA_FOUND 异常。如果要定义一个自定义的异常，则必须在过程的声明部分(即 begin 命令的前面)命名异常。如下面的程序清单所示，这一部分应该包括已经定义的每一个自定义异常项，并作为 EXCEPTION 类型显示出来：

```
declare
some_custom_error EXCEPTION;
```



注意：

如果使用的是已经在 PL/SQL 中定义的异常，则不需要在过程对象的声明部分列出它们。关于预定义异常列表，请参见附录 A 中的“EXCEPTION”项。

在过程对象的代码的 exception 部分，告诉数据库如何处理异常。异常处理部分以关键字 exception 开始，后面是每个异常的 when 子句。每个异常都可以调用 RAISE_APPLICATION_ERROR 过程，此过程接受两个输入参数：错误编号(必须在 - 20 001~ - 20 999 之间)和要显示的错误消息。前面的示例只定义了一个异常。下面的程序清单定义了多个异常；可以使用 when others 子句处理所有未指定的异常：

```
EXCEPTION
    when NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20100,
      'No books borrowed.');
    when some_custom_error THEN
        RAISE_APPLICATION_ERROR (-20101,
           'Some custom error message.');
```

当处理在过程对象中遇到的错误条件时，使用 RAISE_APPLICATION_ERROR 过程将带来极大的灵活性。

35.6.5　命名过程和函数

过程和函数应该根据它们执行的业务功能或它们遵循的业务规则来命名。关于它们的目的不应该有任何歧义。

前面给出的 NEW_BOOK 过程就应该重新命名。NEW_BOOK 过程执行一个业务功能——把记录插入 BOOKSHELF 表中，并从 BOOK_ORDER 表中删除它们，因此它的名字应该反映此项功能。更好的名称是 RECEIVE_BOOK_ORDER。因为它执行一项功能，所以选用的动词(在此示例中是 receive)必须描述它所做的事情。过程名还应该包括它所涉及的主表名。如果对这些表进行了适当的命名，则表名应该是动词所作用的直接对象(在这个示例中是 BOOK_ORDER 表或 BOOKSHELF 表)。为了保持一致性，本章的剩余部分将继续称此过程为 NEW_BOOK。

35.7　create package 语法

在创建程序包时，程序包说明和程序包体是分别创建的。这样，需要分别使用两个命令：用 create package 创建程序包说明，用 create package body 创建程序包体。这两个命令都要求用户具有 CREATE PROCEDURE 系统权限。如果要在一个不属于用户的模式中创建程序包，则必须具有 CREATE ANY PROCEDURE 系统权限。

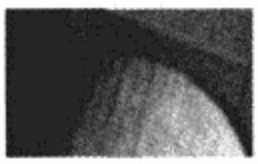
下面是创建程序包说明的语法：

```
create [or replace] package [user.] package
[authid {definer | current_user} ]
{is | as}
package specification;
```

程序包说明(Package Specification)由函数、过程、变量、常量、游标和异常组成，并且这些组成部分对程序包的用户是可用的。

create package 命令的示例如下面的程序清单所示。此示例创建了 BOOK_MANAGEMENT 程序包。本章前面提到的 OVERDUE_CHARGES 函数和 NEW_BOOK 过程都包含在这个程序包中。

```
create or replace package BOOK_MANAGEMENT
    as
      function OVERDUE_CHARGES(aName IN VARCHAR2) return NUMBER;
      procedure NEW_BOOK (aTitle IN VARCHAR2,
          aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2);
    end BOOK_MANAGEMENT;
    /
```



注意：
可以将过程对象名追加到 end 子句中，这样会使在代码中协调逻辑关系的工作变得更容易。

程序包体包含了程序包说明中列出的所有公有对象的代码块和说明。程序包体还可以包括没有在程序包说明中列出的对象；这些对象被称为私有对象，对程序包的用户不可用。私有对象只能由同一程序包体中的其他对象调用。程序包体还可以包含每次调用程序包时运行的代码，不管执行的是程序包的哪一部分——相应的示例请参阅 35.7 节。

创建程序包体的语法如下：

```
create [or replace] package body [user.] package body
    {is | as}
    package body;
```

此程序包体的名称应该与程序包说明的名称相同。

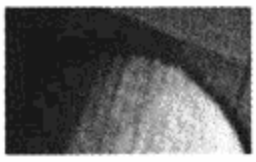
继续以 BOOK_MANAGEMENT 程序包为例，可以通过 create package body 命令创建它的程序包体，如下面的程序清单所示：

```
create or replace package body BOOK_MANAGEMENT
    as
    function OVERDUE_CHARGES (aName IN VARCHAR2)
      return NUMBER
      is
      owed_amount NUMBER(10,2);
    begin
         select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
         into owed_amount
           from BOOKSHELF_CHECKOUT
         where Name = aName
           and (ReturnedDate-CheckoutDate) > 14;
         RETURN(owed_amount);
       EXCEPTION
         when NO_DATA_FOUND THEN
           RAISE_APPLICATION_ERROR(-20100,
           'No books borrowed.');
```

```
end OVERDUE_CHARGES;
procedure NEW_BOOK (aTitle IN VARCHAR2,
    aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
is
begin
    insert into BOOKSHELF
      (Title, Publisher, CategoryName, Rating)
       values (aTitle, aPublisher, aCategoryName, NULL);
    delete from BOOK_ORDER
      where Title = aTitle;
end NEW_BOOK;
end BOOK_MANAGEMENT;
/
```

上面这个示例中的 create or replace package body 命令结合了 OVERDUE_CHARGES 函数的 create function 命令和 NEW_BOOK 过程的 create procedure 命令。end 子句具有附加到其上的全部相关对象名(上面程序清单中显示的粗体部分)。

虽然可以在程序包体中定义其他的函数、过程、异常、变量、游标和常量，但是除非在程序包说明内对它们进行了声明(通过 create package 命令)，否则它们不是公有的。如果某个用户在程序包上具有 EXECUTE 权限，该用户就可以访问程序包说明中声明的任意一个公有对象。



注意：

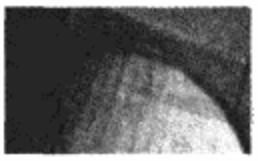
可以使用 PL/SQL 包装器(wrapper)来隐藏应用程序代码的内部情况。WRAP 可执行命令(可在 Oracle 软件主目录的/bin 目录中找到)利用输入文件名作为输入并生成包含对象代码的输出文件。不能将已经包装好的程序包再打开。

程序包可以包含用户在每一次会话中第一次执行此程序包的函数或过程时运行的代码。在下面的示例中，修改 BOOK_MANAGEMENT 程序包体，使其包括一条 SQL 语句，该语句记录当前用户的用户名和在此会话中某个程序包组件第一次执行的时间戳。要记录这些值，还必须在程序包体中声明两个新的变量。

因为这两个新的变量是在程序包体中声明的，所以它们不是公有变量。在程序包体内，它们独立于函数和过程。程序包的初始化代码如下面程序清单中的粗体部分所示：

```
create or replace package body BOOK_MANAGEMENT
    as
    User_Name VARCHAR2(30);
    Entry_Date DATE;
    function OVERDUE_CHARGES (aName IN VARCHAR2)
      return NUMBER
      is
      owed_amount NUMBER(10,2);
    begin
        select SUM(((ReturnedDate-CheckoutDate) -14)*0.20)
        into owed_amount
          from BOOKSHELF_CHECKOUT
```

```
      where Name = aName
        and (ReturnedDate-CheckoutDate) > 14;
    RETURN(owed_amount);
  EXCEPTION
    when NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20100,
      'No books borrowed.');
end OVERDUE_CHARGES;
procedure NEW_BOOK (aTitle IN VARCHAR2,
    aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
is
begin
    insert into BOOKSHELF
      (Title, Publisher, CategoryName, Rating)
      values (aTitle, aPublisher, aCategoryName, NULL);
    delete from BOOK_ORDER
      where Title = aTitle;
end NEW_BOOK;
begin
  select User, SysDate
    into User_Name, Entry_Date
    from DUAL;
end BOOK_MANAGEMENT;
/
```



注意：
在执行程序包的某个组件时，首次运行的代码存储在程序包体底部自己的PL/SQL块中。它没有自己的end子句，它使用程序包体的end子句。

首次在用户的会话中执行BOOK_MANAGEMENT程序包的组件时，可以使用上面程序清单中显示的查询来填充User_Name变量和Entry_Date变量。这样，程序包内的函数和过程就可以使用这两个变量了。但是为了简单起见，本示例没有使用这两个变量。

为了执行程序包内的过程或函数，需要在 execute 命令中指定程序包名和过程或函数的名称，如下所示：

```
execute BOOK_MANAGEMENT.NEW_BOOK('SOMETHING SO STRONG','PANDORAS','ADULTNF');
```

35.8 查看过程对象的源代码

已存在的过程、函数、程序包和程序包体的源代码可以从下面的数据字典视图中查询：

- USER_SOURCE　用于用户所拥有的过程对象
- ALL_SOURCE　用于用户所拥有的过程对象或用户已被授权访问的过程对象
- DBA_SOURCE　用于数据库中的所有过程对象

可通过与下面程序清单中类似的查询，从 USER_SOURCE 视图中选择信息。在该示例中，选择了Text列，按照Line编号排序。对象的Name和Type用来指定待显示对象的源代

码。下面的示例使用了本章前面给出的 NEW_BOOK 过程：

```
select Text
    from USER_SOURCE
  where Name = 'NEW_BOOK'
    and Type = 'PROCEDURE'
  order by Line;

TEXT
------------------------------------------------------------------
procedure NEW_BOOK (aTitle IN VARCHAR2,
   aPublisher IN VARCHAR2, aCategoryName IN VARCHAR2)
as
begin
   insert into BOOKSHELF (Title, Publisher, CategoryName, Rating)
     values (aTitle, aPublisher, aCategoryName, NULL);
   delete from BOOK_ORDER
     where Title = aTitle;
end;
```

如该示例所示，USER_SOURCE 视图针对 NEW_BOOK 过程的每一行包含一条相应的记录。行的顺序由 Line 列维护。因此，应当在 order by 子句中使用 Line 列。

Type 列的有效值是 PROCEDURE、FUNCTION、PACKAGE、PACKAGE BODY、JAVA SOURCE、TYPE 和 TYPE BODY。

35.9　编译过程、函数和程序包

Oracle 在创建过程对象时对它们进行编译。但是，如果过程对象引用的数据库对象发生变化，则这些过程对象可能会失效。下一次执行这些过程对象时，数据库将对它们进行重新编译。

可以显式地重新编译过程、函数和程序包，以避免这种运行时编译——同时避免其导致的性能下降。为了重新编译一个过程，可以使用 alter procedure 命令，如下面的程序清单所示。compile 子句是该命令唯一有效的选项。

```
alter procedure NEW_BOOK compile;
```

为重新编译一个过程，必须拥有该过程或者具有 ALTER ANY PROCEDURE 系统权限。

要重新编译一个函数，使用带有 compile 子句 alter function 命令：

```
alter function OVERDUE_CHARGES compile;
```

要重新编译一个函数，必须拥有该函数或者具有 ALTER ANY PROCEDURE 系统权限。

在重新编译程序包时，要么重新编译程序包说明和程序包体，要么只重新编译程序包体。在默认情况下，程序包说明和程序包体都要重新编译。不能使用 alter function 或 alter procedure 命令来重新编译存储在程序包内的函数和过程。

如果程序包体中过程或函数的源代码已经改变，但是程序包说明没有改变，那么只需重新编译程序包体即可。在大多数情况下，最好是将程序包说明和程序包体都重新编译。

下面是 alter package 命令的语法：

```
alter package [user.] package_name

compile [debug] [package | body | specification];
```

要重新编译程序包，可以使用带有 compile 子句的 alter package 命令，如下所示：

```
alter package BOOK_MANAGEMENT compile;
```

要重新编译程序包，必须拥有程序包或者具有 ALTER ANY PROCEDURE 系统权限。由于前面的示例既没有指定 PACKAGE 也没有指定 BODY，因此默认使用 PACKAGE，从而导致程序包说明和程序包体都被重新编译了。

35.10 替换过程、函数和程序包

过程、函数和程序包可以分别使用其 create or replace 命令替换。使用 or replace 子句可以使这些对象已有的授权保持不变。如果选择删除并重新创建过程对象，则必须重新授予前面授予的任何 EXECUTE 权限。

35.11 删除过程、函数和程序包

删除过程可以使用 drop procedure 命令，如下所示：

```
drop procedure NEW_BOOK;
```

要删除一个过程，必须拥有该过程或具有 DROP ANY PROCEDURE 系统权限。

要删除函数，可以使用 drop function 命令，如下所示：

```
drop function OVERDUE_CHARGES;
```

要删除一个函数，必须拥有该函数或拥有 DROP ANY PROCEDURE 系统权限。

要删除程序包体，可以使用带 body 子句的 drop package 命令，如下所示：

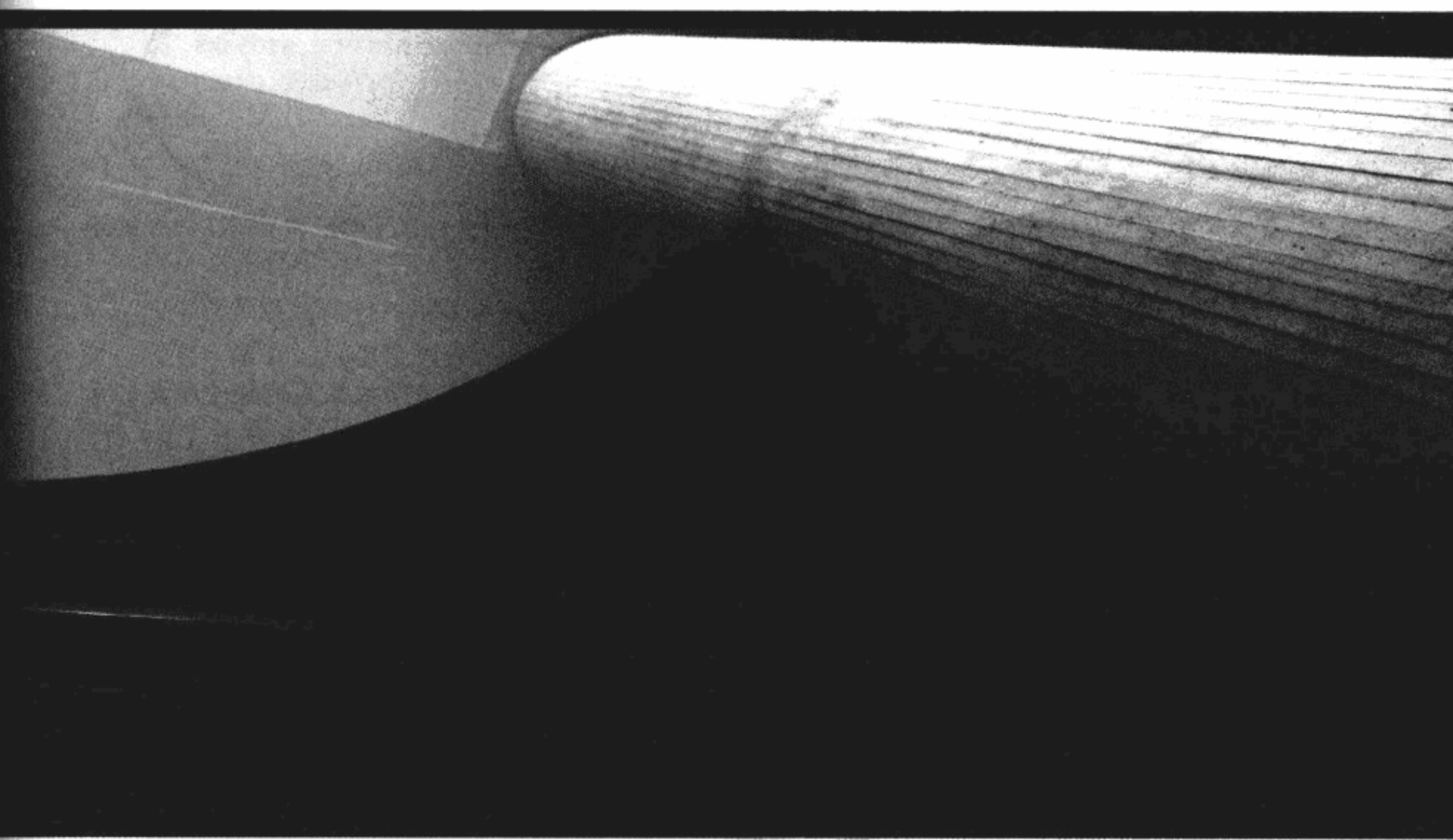
```
drop package body BOOK_MANAGEMENT;
```

要删除一个程序包体，必须拥有该程序包或者具有 DROP ANY PROCEDURE 系统权限。

要删除一个程序包(包括程序包说明和程序包体)，可以使用 drop package 命令，如下所示：

```
drop package BOOK_MANAGEMENT;
```

要删除程序包，必须拥有该程序包或者拥有 DROP ANY PROCEDURE 系统权限。



第36章

使用本地动态 SQL 和 DBMS_SQL

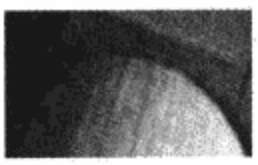
用户可以使用本地动态 SQL(native dynamic SQL)创建通用过程，并可以在 PL/SQL 中执行 DDL 命令。可以在存储的过程对象内(如程序包)和在匿名的 PL/SQL 块中使用动态 SQL 执行在运行时而非创建程序时构建的 SQL 命令。比如，可以创建一个用来创建表的过程，该过程将表名称或者 where 子句作为输入参数。

36.1 使用 EXECUTE IMMEDIATE

下面示例中的脚本创建一个名为 ANYSTRING 的过程。ANYSTRING 过程只有一个输入变量：SQL 命令。执行 ANYSTRING 过程时，将要执行的 SQL 命令传递给它；SQL 命令可以是 DML 命令(如 select 或 insert)或者 DDL 命令(如 create table)。

```
create or replace procedure ANYSTRING(String IN VARCHAR2) AS
BEGIN
   EXECUTE IMMEDIATE (String);
END;
/
```

可以通过 execute 命令来执行 ANYSTRING 过程，如将要介绍的 DDL 和 DML 命令的示例所示。请注意输入的字符串不能以分号结尾。



注意：
如果打算通过动态 SQL 来运行 DDL 命令，则您必须被显式地授予所需的系统权限——而不是通过角色来授权。如果您没有被显式地授予 CREATE TABLE 系统权限，则以下命令将会失败，并显示“ORA-01031:insufficient privileges”错误：

```
execute ANYSTRING('create table CD (Artist VARCHAR2(25),
  Title VARCHAR2(25)) ');

execute ANYSTRING('insert into CD values(''NEIL FINN'',
  ''TRY WHISTLING THIS'') ');
```

以上程序清单中的第一个命令执行了以下 SQL 命令：

```
create table CD
(Artist VARCHAR2(25),
 Title VARCHAR2(25));
```

程序清单中的第二个命令将一条记录插入 CD 表：

```
insert into CD values ( 'NEIL FINN', 'TRY WHISTLING THIS' );
```

如果传递给 ANYSTRING 的 SQL 语句包含单引号，则每个单引号都应当被替换为一对单引号，如前面执行 ANYSTRING 的 insert 命令一样。在执行完每个命令后，将接收到以下响应信息：

```
PL/SQL procedure successfully completed.
```

可以通过查询新表来验证命令的执行是否成功：

```
select * from CD;

ARTIST                    TITLE
------------------------  ------------------------
NEIL FINN                 TRY WHISTLING THIS
```

可以为将传递给 SQL 命令的输入参数编写本地动态 SQL：

```
declare
  Symbol VARCHAR2(6) := 'WOND';
```

```
begin
    EXECUTE IMMEDIATE 'delete from STOCK where Symbol = :symbol'
    USING Symbol;
end;
/
```

设置 symbol 变量等于“WOND”，然后通过 using 子句将该值传递给 delete 命令。

在下面的程序清单中，名为 DELETE_ROWS 的过程将从作为变量传递的表中删除所有行。如果在该过程执行时输入 where_clause 的一个值，则该值将动态地添加到 delete 命令后面；否则，所有的行将被删除。

```
create or replace procedure DELETE_ROWS (
    Table_Name in VARCHAR2,
    Condition in VARCHAR2 default NULL) as
    Where_Clause VARCHAR2(100) := ' where ' || Condition;
begin
    if Condition is NULL then Where_Clause := NULL; end if;
    execute immediate 'delete from ' || Table_Name || Where_Clause;
end;
/
```

可以在想从表中删除行的任何时候调用 DELETE_ROWS 过程。无需授予任何人直接在 SQL*Plus 中执行 delete 命令的权限，即可在另外一个过程或在应用程序中调用此过程。以下的命令将删除 DEPT 表中的所有行：

```
execute DELETE_ROWS( 'STOCK' );
```

除了调用 SQL 语句之外，还可以通过本地动态 SQL 来执行过程和 PL/SQL 块。

36.2　使用绑定变量

本地动态 SQL 的一个优点就是可以在代码中使用绑定变量。例如，考虑下面这个在本章前面出现过的示例：

```
declare
  Symbol VARCHAR2(6) := 'WOND';
begin
    EXECUTE IMMEDIATE 'delete from STOCK where Symbol = :symbol'
    USING Symbol;
end;
/
```

在上面的程序清单中，“:symbol”是绑定变量。这样，symbol 变量后续的值就能够传递给 EXECUTE IMMEDIATE 命令了。由于使用了绑定变量，因此命令就不用被重复解析了。如果在使用应用程序时监视数据库的状态，就会在 V$SQL 中看见该命令的一个条目。如果没有使用绑定变量，则将在 V$SQL 中看见每个命令的执行都分别有相应的条目。也就是说，

将不会在库缓存中看到：

```
delete from STOCK where Symbol = :symbol
```

而将看到每一个单独的删除操作对应的一个的条目，例如：

```
delete from STOCK where Symbol = 'WOND'
```

这些命令全部是分开解析的，并且每一条命令都要占据内存空间。因此，未能使用绑定变量可能导致共享池中存在大量相似的命令——这不仅浪费了数据库本应充分利用的资源而且导致了潜在的性能问题。虽然数据库管理员可以通过将 CURSOR_SHARING 初始化参数设置为 SIMILAR 来解决部分问题，但是从本质上说，该问题应当通过正确的开发方法来解决。

当查询运行时，绑定参数将它们相应的占位符放入字符串中。每个占位符都必须与 using 子句或 returning into 子句中的一个绑定变量相关联。绑定变量可以是数字、字符或者字符串字面值。

下面的示例说明了在动态 update 语句中绑定变量的用法：

```
declare
    sql_stmt    VARCHAR2(200);
    Symbol      VARCHAR2(6) := 'ADSP';
    CloseToday  NUMBER(6,2);
begin
sql_stmt := 'update STOCK set CloseToday = 32 WHERE Symbol = :1
      RETURNING CloseToday INTO :2';
   EXECUTE IMMEDIATE sql_stmt USING Symbol RETURNING INTO CloseToday;
end;
/
```

returning into 子句只用于具有 returning 子句的 insert、update 和 delete 命令。对该命令返回的每个值，都必须在 returning into 子句中有一个相应的变量。

36.3 使用 DBMS_SQL

在 Oracle9i 之前，动态 SQL 需要使用 DBMS_SQL 程序包。现在还可以使用 DBMS_SQL 方法，如本节所示。虽然使用 DBMS_SQL 可以提高用户对动态 SQL 内的处理流的控制能力，但是它通常比本章前面所示的本地动态 SQL 的编写要复杂许多。

以下示例中的脚本将创建 DBMS_SQL 版本的 ANYSTRING 过程(本章前面出现的)。ANYSTRING 过程只有一个输入变量：一条 SQL 命令。当用户执行 ANYSTRING 过程时，把要执行的 SQL 命令传递给它；SQL 命令可以是 DML 命令(如 select 或 insert)或者 DDL 命令(如 create table)。

```
create or replace procedure ANYSTRING(String IN VARCHAR2) AS
    Cursor_Name INTEGER;
    Ret INTEGER;
BEGIN
```

```
    Cursor_Name := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(Cursor_Name, String, dbms_sql.Native);
    Ret := DBMS_SQL.EXECUTE(Cursor_Name);
    DBMS_SQL.CLOSE_CURSOR(Cursor_Name);
END;
/
```

如更早的版本所示，可以通过 execute 命令执行 ANYSTRING 过程，如下面关于 DDL 和 DML 命令的示例所示：

```
execute ANYSTRING('drop table CD');

execute ANYSTRING('create table CD (Artist VARCHAR2(25),
  Title VARCHAR2(25))');

execute ANYSTRING('insert into CD values(''NEIL FINN'',
  ''TRY WHISTLING THIS'')');
```

如果传递给 ANYSTRING 的 SQL 语句包含单引号，则每个单引号都将替换成一对单引号，如前面执行 ANYSTRING 的 insert 命令所示。在执行完每个命令后，将接收到如下响应信息：

```
PL/SQL procedure successfully completed.
```

可以通过查询新表来验证命令的执行是否成功，如下面的示例所示：

```
select * from CD;

ARTIST                         TITLE
-------------------------      -------------------------
NEIL FINN                      TRY WHISTLING THIS
```

下面的几节将介绍在动态 SQL 语句中使用的主要函数和过程。

36.3.1　OPEN_CURSOR

DBMS_SQL 程序包的 OPEN_CURSOR 函数打开一个游标，并将游标的 ID 号返回给程序。因此，在 ANYSTRING 过程中，cursor_name 变量被定义为 INTEGER 数据类型：

```
Cursor_Name INTEGER;
```

在可执行命令部分，将 cursor_name 变量设置为执行 OPEN_CURSOR 函数的输出：

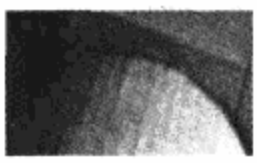
```
Cursor_Name := DBMS_SQL.OPEN_CURSOR;
```

36.3.2　PARSE

语句的解析过程将检查语句的语法，并将语句与游标相关联。要解析一个语句，可以使用 DBMS_SQL 程序包的 PARSE 过程。例如，在 ANYSTRING 过程中，下面的命令将解析 SQL 命令。PARSE 接受 3 个参数：游标 ID、待解析的命令和语言标记。

```
DBMS_SQL.PARSE(Cursor_Name, String, dbms_sql.Native);
```

语言标记指定语句的表现形式。在本示例中显示的“Native”设置根据当前的数据库版本指定了语句的表现形式。



注意：
DDL 命令(如 create table)解析后才会执行。

36.3.3 BIND_VARIABLE 和 BIND_ARRAY

如果动态 SQL 过程执行了数据操作，就需要使用绑定变量。任何插入、更新或删除记录的动态 SQL 命令都需要在 DBMS_SQL 内使用 BIND_VARIABLE 或者 BIND_ARRAY 过程作为处理的一部分。必须调用其中的一个过程来为占位符提供程序中变量的值。然后当执行 SQL 语句时，Oracle 将会使用程序放置在输出、输入或者绑定变量中的数据。

例如，以下程序清单中执行的 PARSE 将解析一个 SQL 命令，该命令中的一个值未知，即该过程使用的绑定变量的值：

```
DBMS_SQL.PARSE(Cursor_Name, 'delete from CD where Artist > :artist',
                    dbms_sql.Native);
```

在执行了 PARSE 之后，必须将“:artist”绑定变量绑定到传递给该过程的 artist_name 变量。当用户执行该过程时，将会为 artist_name 变量指定一个值。以下程序清单所示的 BIND_VARIABLE 过程将 artist_name 值与将被解析的命令串所使用的“:artist”绑定变量进行绑定：

```
DBMS_SQL.BIND_VARIABLE(Cursor_Name, ':artist', artist_name);
```

可以使用动态 SQL 多次执行一个 DML 语句——每次都有一个不同的绑定变量。可以使用 BIND_ARRAY 过程绑定一组值，每个 EXECUTE 每次只能使用其中的一个值作为输入变量。

36.3.4 EXECUTE

要执行动态创建的 SQL 语句，可以使用 DBMS_SQL 程序包的 EXECUTE 函数。下面的程序清单来自 ANYSTRING 过程，说明了 EXECUTE 函数的用法：

```
Ret := DBMS_SQL.EXECUTE(Cursor_Name);
```

使用 EXECUTE 函数时，将游标 ID 值(在本示例中为 cursor_name 变量)作为输入。ret 变量存储 EXECUTE 函数的输出，该变量表示处理过的行数(若执行的语句运行的是 DML)。

36.3.5 DEFINE_COLUMN

如果游标执行了一个查询，就必须为每个被选择的列执行一次 DEFINE_COLUMN 函数。如果该列被定义为 LONG 数据类型，则必须使用 DEFINE_COLUMN_LONG 函数来代替 DEFINE_COLUMN 函数。如果为一个 PL/SQL 数组选择相应的值，就必须使用 DEFINE_ARRAY 函数

代替 DEFINE_COLUMN 函数。

例如，考虑在下面的命令中解析的查询：

```
DBMS_SQL.PARSE(Cursor_Name,'select Artist, Title from CD',
                    dbms_sql.Native);
```

在执行游标之前，必须在查询中根据位置定义列，如下面的程序清单所示：

```
DBMS_SQL.DEFINE_COLUMN(Cursor_Name, 1, Artist, 25);
DBMS_SQL.DEFINE_COLUMN(Cursor_Name, 2, Title, 25);
```

一旦定义了列，就可以使用 EXECUTE 函数执行游标了。

36.3.6　FETCH_ROWS、EXECUTE_AND_FETCH 和 COLUMN_VALUE

如果动态 SQL 命令执行一个查询，则可以调用 FETCH_ROWS 函数来检索该查询的行。而 EXECUTE_AND_FETCH 函数既可以执行又可以取出行。

例如，假定有一个名为 CD_COPY 的表，其中的列定义与本章前面创建的 CD 表相同。可以使用动态 SQL 从 CD 表中选择记录，并将这些记录插入 CD_COPY 表中。在此示例中，Get_CD_Cursor 语句将从 CD 表中检索行，而 Insert_CD_Copy_Cursor 语句将检索到的值插入 CD_COPY 表中。

在此示例中，Get_CD_Cursor 游标已经解析过。样本代码执行 FETCH_ROWS 函数。如果检索的行数大于零，那么将处理记录。

```
if DBMS_SQL.FETCH_ROWS(Get_CD_Cursor)>0 then
```

对于检索的记录，为选择的每一列执行一次 COLUMN_VALUE 过程：

```
DBMS_SQL.COLUMN_VALUE(Get_CD_Cursor, 1, Artist);
DBMS_SQL.COLUMN_VALUE(Get_CD_Cursor, 2, Title);
```

既然查询的值已经绑定到变量，就可以将这些值与把记录插入 CD_COPY 表中的游标绑定：

```
DBMS_SQL.BIND_VARIABLE(Insert_CD_Copy_Cursor, 'artist', Artist);
DBMS_SQL.BIND_VARIABLE(Insert_CD_Copy_Cursor, 'title', Title);
```

现在可以执行将值插入 CD_COPY 表的游标：

```
Ret := DBMS_SQL.EXECUTE(Insert_CD_COPY_Cursor);
```

ret 变量用来存储插入 CD_COPY 表中的行数。如果从一个匿名的 PL/SQL 块中调用过程，就可以执行 VARIABLE_VALUE 函数来检索赋予输出变量的值。

36.3.7　CLOSE_CURSOR

可以通过执行 DBMS_SQL 程序包的 CLOSE_CURSOR 过程来关闭已打开的游标。下面程序清单中的示例来自 ANYSTRING 过程，显示了 CLOSE_CURSOR 过程使用游标 ID 值作为它唯一的输入参数：

```
DBMS_SQL.CLOSE_CURSOR(Cursor_Name);
```

关闭游标将释放游标使用的内存，并减少在会话中同时打开的游标数。

除非需要详细控制执行动态 SQL 所涉及的步骤，否则就应该使用本章前面介绍的 EXECUTE IMMEDIATE 方法。如果必须使用 DBMS_SQL 程序包方法，请参阅 Oracle 文档中的“PL/SQL Packages and Types Reference”，它包括所有 DBMS_SQL 程序包的过程和函数的详细描述。

第37章

PL/SQL 调 整

调整 PL/SQL 代码涉及多个步骤：调整 SQL、调整代码内部的过程步骤、利用 PL/SQL 特性。本章将介绍如何处理 PL/SQL 调整，如何使用 PL/SQL 中可用的配置工具来确定性能问题产生的原因，如何使用 PL/SQL 特性，如 FORALL 运算符号。

37.1 调整 SQL

假定有一个性能糟糕的 PL/SQL 过程，调整它的第一步是查看程序包中的 SQL。由于执行 SQL 通常占了大部分的过程执行时间，因此，对 SQL 的任何改进都会对过程的性能产生巨大的影响。关于调整 SQL 语句的详细介绍参见第 46 章。

根据过程的结构，也许不能提取与执行的查询精确匹配的 SQL。例如，SQL 命令可能通

过动态 SQL 汇编，或可能在 where 子句中使用绑定变量。在这样的情况下，就需要监控程序，以准确地看清执行的 SQL 是什么。可以通过 trace 实用程序或通过动态性能视图(包括 V$SQL)来监控程序，这些动态性能视图展示了每个会话执行的 SQL 命令。执行大量物理读(通常是由于缺失索引)或大量逻辑读(由于处理效率低下或索引选择不佳)的 SQL 会对调用它的任何 PL/SQL 块的性能产生负面影响。

如果能准确地复制正在执行的 SQL，则使用第 46 章描述的 explain plan 或 set qutotrace 命令来确定优化程序选择的执行路径以及预期的成本。

37.2 调整 PL/SQL 的步骤

即使 PL/SQL 代码中的 SQL 效率很高，也仍然能够改进代码。在编写 PL/SQL 代码时，应该遵循下面这些最佳实践：

1. 使用提供的函数和过程

避免自己编写 Oracle 已提供的函数。使用 Oracle 的搜索和排序例程，而不要自己编写这些例程。

2. 使用可用的快捷方法

利用 Oracle 执行控制逻辑判断的方法。如果有多个条件来控制分支，则 Oracle 会按提供的顺序对这些条件进行判断。如果不是需要，Oracle 就不会对所有条件进行判断，因此，也许可以对要判断的条件进行合理排序使 Oracle 可以走捷径。如果有多个条件需要判断，例如：

```
if Salary > 1000 or Bonus>1000
```

限制性最强的条件应该放在前面。如果 Salary 的值大于 1 000，就不计算 Bonus 的值。如果其中的一个条件执行用户定义的函数调用，例如：

```
if Salary > 1000 or range_balance(Bonus)>1000
```

则将此条件放在第二个位置会限制它的调用次数。如下面这样改变条件的顺序，使每次执行 if 子句时，Oracle 都会计算此函数调用的值：

```
if range_balance(Bonus)>1000 or Salary > 1000
```

3. 避免动态数据类型转换

避免将具有不同数据类型的变量进行比较。如果在比较之前将数据类型转换为一致的集合，那么可以节省每次执行数据类型转换所花的时间。

4. 正确地确定 VARCHAR2 变量的大小

开发人员常常为 VARCHAR2 变量分配更多的存储空间。例如，您可能有一个标准，它强制创建变量如 VARCHAR2(200)，即使变量的最大长度小于 200。Oracle 在 PL/SQL 中的内

存分配包含一个用来确定存储空间大小的算法，创建占用内存较多的 VARCHAR2 变量时应该考虑此算法。

根据 VARCHAR2 变量的长度，Oracle 分配的内存有所不同：

- 如果 VARCHAR2 变量的长度大于或等于 2 000，则 PL/SQL 只动态地分配足够的内存来存储实际值。
- 如果 VARCHAR2 变量的长度小于 2 000，则 PL/SQL 分配足够的内存来存储变量声明的长度。

如果在两个不同的 VARCHAR2 变量中存储一个 500 字节的值，那么对内存使用情况会产生什么影响呢？如果此值存储在 VARCHAR2(1000)变量中，则 PL/SQL 将分配 1 000 字节的内存。如果此值存储在 VARCHAR2(2000)变量中，则 PL/SQL 只分配实际值大小的内存，即 500 字节。对变量使用较多的内存使得分配的内存从 1 000 字节减少到了 500 字节。

下面的几节将介绍如何识别执行时间最长的代码行，以及如何使用 PL/SQL 选项来调整大批量操作。

37.3　使用 DBMS_PROFILE 识别问题

可以使用 DBMS_PROFILER 程序包来识别执行时间最长的代码行。虽然 DBMS_PROFILER 程序包在较早的 Oracle 版本中可用，但并非总是默认安装它。如果数据库从较早的 Oracle 版本升级而来，则应该检查确认已安装了 DBMS_PROFILER 程序包。如果还没有安装此程序包，则在 Oracle 主目录下的/rdbms/admin 目录中查看 profload.sql 脚本。当以 SYS 连接时执行 profload.sql 脚本，以创建配置所需要的程序包。

Oracle 另外还提供了两个脚本：proftab.sql 和 profrep.sql。使用配置程序(profiler)的每个人都需要在他们的模式中有一组表，以捕获配置统计信息。配置表应该通过在该用户的模式中执行 proftab.sql 脚本来创建。profrep.sql 脚本应该运行在相同的模式中，它创建的视图可用来显示配置统计信息。根据所使用的 Oracle 版本，profrep.sql 脚本内容可以集成到其他脚本中。

当使用配置程序时，需要知道代码中的行编号。下面从一个简单的循环示例开始，它计算圆的面积并向 AREAS 表中插入行：

```
SQL> Declare
  2 Pi constant NUMBER(9,7) := 3.141593;
  3 Radius integer(5);
  4 Area number(14,2);
  5 Begin
  6 Radius :=3;
  7 Loop
  8 Area:=pi*power(radius,2);
  9 Insert into AREAS values (radius, area);
 10 Radius := radius+1;
 11 Exit when area>100;
 12 End loop;
```

```
 13 End;
 14 /

PL/SQL procedure successfully completed.
```

为验证正确执行了此 SQL，查询 AREAS 表：

```
SQL> select * from areas;

    RADIUS       AREA
---------- ----------
         3      28.27
         4      50.27
         5      78.54
         6      113.1
```

这两个命令——PL/SQL 块和 AREAS 表的查询——现在都已经被解析并存储在共享池中。因此，解析命令所需的时间不影响接下来的性能配置。

从 AREAS 表中删除行，准备测试表：

```
SQL> delete from areas;

4 rows deleted.

SQL> commit;

Commit complete.
```

接下来，执行 DBMS_PROFILER 程序包的 START_PROFILER 过程。执行 START_PROFILER 过程之后，Oracle 将收集被运行命令的统计信息。

```
SQL> exec dbms_profiler.start_profiler;

PL/SQL procedure successfully completed.
```

现在再次运行 PL/SQL 块：

```
SQL> Declare
  2  Pi constant NUMBER(9,7) := 3.141593;
  3  Radius integer(5);
  4  Area number(14,2);
  5  Begin
  6  Radius :=3;
  7  Loop
  8  Area:=pi*power(radius,2);
  9  Insert into AREAS values (radius, area);
 10  Radius := radius+1;
 11  Exit when area>100;
 12  End loop;
 13  End;
```

```
  14   /

PL/SQL procedure successfully completed.
```

停止配置程序，如下面的代码行所示：

```
SQL> exec dbms_profiler.stop_profiler;

PL/SQL procedure successfully completed.
```

如果您已经在自己的模式中执行了 profrep.sql 脚本，则现在能够从提供的视图中查询配置程序的统计信息。PLSQL_PROFILER_RUNS 显示了模式名和执行代码的日期，如下面的程序清单所示：

```
SQL> set pagesize 1000

SQL> select * from plsql_profiler_runs

     RUNID   RELATED_RUN  RUN_OWNER                          RUN_DATE
 ---------  -----------  ------------------------------  --------
RUN_COMMENT
-----------------------------------------------------------------
RUN_TOTAL_TIME
--------------
RUN_SYSTEM_INFO
-----------------------------------------------------------------
RUN_COMMENT1
-----------------------------------------------------------------
SPARE1
-----------------------------------------------------------------
         1             0 PRACTICE                           08-NOV-07
08-NOV-07
   1.7515E+10
```

PLSQL_PROFILER_DATA 视图显示每行的时间，单位是纳秒。如果相同的代码执行多次，则显示每行的总时间、最长时间和最短时间。

```
SQL> select * from plsql_profiler_data;
     RUNID  UNIT_NUMBER      LINE#  TOTAL_OCCUR  TOTAL_TIME  MIN_TIME   MAX_TIME
---------  -----------  ---------  -----------  ----------  --------  ---------
   SPARE1       SPARE2      SPARE3      SPARE4
---------  ----------  ---------  ----------

         1           1          1           1        1400      1400       1400

         1           2          1           0        8900      8900       8900

         1           2          2           1        3000      3000       3000

         1           2          6           1         700       700        700
```

```
         1          2          8          4      19200       2000      11800
         1          2          9          4     444700      26400     357500
         1          2         10          4       5200        700       2900
         1          2         11          4       3100        400       1900
         1          2         13          1       3100       3100       3100
         1          3          1          2      37600       1800      30100

10 rows selected.
```

如上面的程序清单所示，时间最密集的步骤是第 9 行，即 444 700 纳秒。从上述代码中可以看到第 9 行是：

```
9 Insert into AREAS values (radius, area);
```

相比之下，第 8 行(计算 Area)所花的时间不到第 9 行的 5%。调整 PL/SQL 块的任何工作都应该从调整目前系统开销最大的步骤开始，即一次一行地将所有行插入表中的 SQL 命令。

代码如何扩展呢？如果插入很多行结果会怎么样呢？使用配置程序来查看每一步的性能如何改变。为了确保前面的统计信息意义明确，执行 proftab.sql 脚本，删除并重新创建模式中的统计表：

```
SQL> @proftab
Table dropped.

Table dropped.

Table dropped.

Sequence dropped.

Table created.

Comment created.

Table created.

Comment created.

Table created.

Comment created.

Sequence created.
```

接下来，启动配置程序：

```
SQL> exec dbms_profiler.start_profiler;

PL/SQL procedure successfully completed.
```

再次执行 AREAS PL/SQL 块，但这次对第 11 行做了修改，使得退出条件为 Area>1000：

```
SQL> Declare
  2  Pi constant NUMBER(9,7) := 3.141593;
  3  Radius integer(5);
  4  Area number(14,2);
  5  Begin
  6  Radius :=3;
  7  Loop
  8  Area:=pi*power(radius,2);
  9  Insert into AREAS values (radius, area);
 10  Radius := radius+1;
 11  Exit when area>1000;
 12  End loop;
 13  End;
 14  /

PL/SQL procedure successfully completed.
```

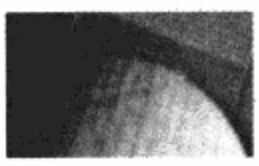
接下来，停止配置程序：

```
SQL> exec dbms_profiler.stop_profiler;

PL/SQL procedure successfully completed.
```

然后检查报表视图，以查看统计信息：

```
SQL> select * from plsql_profiler_lines_cross_run
  2 order by total_time;

UNIT_OWNER                     UNIT_NAME
------------------------------ ------------------------------
UNIT_TYPE                         LINE# TOTAL_OCCUR TOTAL_TIME    MIN_TIME
------------------------- --------- ---------- ----------  ---------
  MAX_TIME
----------
<anonymous>                    <anonymous>
ANONYMOUS BLOCK                        6          1         300         300
        300

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                        2          1        2500        2500
       2500

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                       13          1        3100        3100
       3100

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                       11         16        6900         300
       1800
```

```
<anonymous>                    <anonymous>
ANONYMOUS BLOCK                        10          16        13600         700
       2800

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                         8          16        45700        2000
      12600

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                         1           3        48300        1200
      31100

<anonymous>                    <anonymous>
ANONYMOUS BLOCK                         9          16      1787400       26200
    1380900

8 rows selected.
```



注意：
在 profrep.sql 脚本可用的地方，PLSQL_PROFILER_LINES_CROSS_RUN 表是通过 profrep.sql 脚本创建的。

如程序清单中的 Total_Occur 列所示，PL/SQL 块的不同行执行的次数不同。循环内的行执行多次，程序清单中显示的 Total_Time 是这些行被执行的所有次数的累计执行时间。执行次数的变化对每次执行的平均时间有什么影响呢？

第 9 行——即 insert 行——在第一次测试时总计花了 444 700 纳秒。此测试要求插入 4 行，因此，这些插入的平均值是每行 111 175 纳秒。在第二次测试期间，在 1 787 400 纳秒内插入 16 行，即平均每行是 111 702 纳秒。insert 操作的性能是线性的，平均算来，无论插入次数的多少，每个插入所花的时间几乎都完全相同。虽然这种性能是可以预料的，但不能很好地调整。理想情况下，我们希望每次新执行一个步骤，都不再额外地花时间。

如果试图调整执行很多独立事务的 PL/SQL 代码，则应该考虑使用批量的插入和选择操作，如下面几节所示。

37.4 将 PL/SQL 特性用于批量操作

PL/SQL 提供了很多特性来增强执行很多独立查询或很多独立事务的代码的性能。通过将多个小操作合并起来，也许可以提高性能，尤其是在 CPU 利用率上。通过将多个命令合并成一个命令，可以减少解析操作的次数和相关的处理成本。虽然事务数量较少时，其效果并不明显，但随着事务数量的增加，收效将是很可观的。下面几节将简要介绍并举例说明两个这样的方法：forall 和 bulk collect。

37.4.1 forall 操作

forall 运算符用一系列动态或静态的 DML 命令替代 for 循环。可以为 forall 命令处理期间

它所迭代的值指定上限和下限。Forall 运算符使用批量绑定，允许在单个命令中将数据集传递到数据库中。查询可以返回多个结果，无需单独取出每一行。

考虑下面的示例。首先，新创建一个测试期间要使用的表：

```
create table BOOKSHELF_TEMP
as select * from BOOKSHELF;
```

现在，删除 BOOKSHELF_TEMP 表中 Rating 值为 1、2 或 3 的所有行。可以使用 PL/SQL 遍历每一个 Rating，并对每个 Rating 发出单独的 delete 命令(在 for 循环内)。下面的程序清单说明了如何使用 forall 以批量方式执行删除操作：

```
DECLARE
   TYPE NumList IS VARRAY(20) OF NUMBER;
   Ratings NumList := NumList(1, 2, 3);
BEGIN
   FORALL i IN Ratings.FIRST..Ratings.LAST
      delete from BOOKSHELF_TEMP
      where Rating = Ratings(i);
   COMMIT;
END;
/
```

下面这行代码：

```
Ratings NumList := NumList(1, 2, 3);
```

定义了 Rating 集合。它使用 NumList 数据类型(最多具有 20 个值的可变数组)。然后可以使用 Rating 集合的值控制要执行的命令。在此示例中，从第一个条目遍历到最后一个条目：

```
FORALL i IN Ratings.FIRST..Ratings.LAST
```

对于其中的每一个评定等级，执行相同的删除：

```
   delete from BOOKSHELF_TEMP
   where Rating = Ratings(i);
```

下面的示例来自 Oracle，它快速地比较 inserts 期间 forall 命令的性能。要查看 DBMS_OUTPUT 调用的输出，首先应该使用 set serveroutput on 命令：

```
set serveroutput on
```

启用输出显示，下面的命令说明了 forall 带来的性能改进：

```
CREATE TABLE parts1 (pnum INTEGER, pname VARCHAR2(15));
CREATE TABLE parts2 (pnum INTEGER, pname VARCHAR2(15));
DECLARE
   TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
   TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
   pnums NumTab;
```

```
    pnames NameTab;
    iterations CONSTANT PLS_INTEGER := 500;
    t1 INTEGER;
    t2 INTEGER;
    t3 INTEGER;
BEGIN
    FOR j IN 1..iterations LOOP -- load index-by tables
        pnums(j) := j;
        pnames(j) := 'Part No. ' || TO_CHAR(j);
    END LOOP;
    t1 := DBMS_UTILITY.get_time;
    FOR i IN 1..iterations LOOP -- use FOR loop
        INSERT INTO parts1 VALUES (pnums(i), pnames(i));
    END LOOP;
    t2 := DBMS_UTILITY.get_time;
    FORALL i IN 1..iterations -- use FORALL statement
        INSERT INTO parts2 VALUES (pnums(i), pnames(i));
    t3 := DBMS_UTILITY.get_time;
    DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
    DBMS_OUTPUT.PUT_LINE('---------------------');
    DBMS_OUTPUT.PUT_LINE('FOR loop: ' || TO_CHAR((t2 - t1)/100));
    DBMS_OUTPUT.PUT_LINE('FORALL: ' || TO_CHAR((t3 - t2)/100));
    COMMIT;
END;
/
```

此示例代码通过两个独立的循环插入数据。第一个循环通过传统的 for 循环将数据插入 PARTS1 表中：

```
FOR i IN 1..iterations LOOP -- use FOR loop
    INSERT INTO parts1 VALUES (pnums(i), pnames(i));
END LOOP;
```

第二个循环使用 forall 方法：

```
FORALL i IN 1..iterations -- use FORALL statement
    INSERT INTO parts2 VALUES (pnums(i), pnames(i));
```

由于此示例只插入 500 行，因此性能差别是有限的：

```
iterations CONSTANT PLS_INTEGER := 500;
```

因为数据集很小，所以性能上的任何差别都是有意义的。它表明随着迭代次数的增加，性能的趋势如何改变。在非常健壮的服务器上运行代码，会得到下面的结果：

```
FOR loop: 0.03 sec
FORALL: 0.01 sec
```

随着行数的增加，节省的时间也会增加。

当使用 forall 时，应该仔细考虑它将遍历的值。可以设置不同的值列表(如前面的 delete

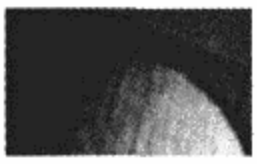
示例所示)，或告诉 Oracle 遍历指定数量的值。forall 可用的完整选项列表参见 *Oracle PL/SQL Users Guide and Reference*。

37.4.2　bulk collect 操作

与 forall 一样，bulk collect 高效地作用于记录集。forall 支持数据操作，而 bulk collect 支持查询。bulk collect 不遍历游标的每一行，相反，可以使用 bulk collect 在单个操作中将查询结果存储在集合中。bulk collect 运算符支持使用 select into、fetch into 和 returning into，如下面的示例所示。

在 bulk collect 查询中，接受所选择数据的变量必须是集合(嵌套表或可变数组)。bulk collect 操作的基本格式如下面的示例所示，此示例来自 Oracle(此示例假定您的模式中存在 EMPLOYEES 表)：

```
DECLARE
    TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
    TYPE NameTab IS TABLE OF employees.last_name%TYPE;
    enums NumTab; -- No need to initialize the collections.
    names NameTab; -- Values will be filled in by the SELECT INTO.
    PROCEDURE print_results IS
    BEGIN
      IF enums.COUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('No results!');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Results:');
        FOR i IN enums.FIRST .. enums.LAST
        LOOP
          DBMS_OUTPUT.PUT_LINE(' Employee #' || enums(i) || ': ' || names(i));
        END LOOP;
      END IF;
    END;
BEGIN
  -- Retrieve data for employees with Ids greater than 1000
    SELECT employee_id, last_name
      BULK COLLECT INTO enums, names FROM employees WHERE employee_id > 1000;
-- The data has all been brought into memory by BULK COLLECT
-- No need to FETCH each row from the result set
    print_results();
-- Retrieve approximately 20% of all rows
    SELECT employee_id, last_name
      BULK COLLECT INTO enums, names FROM employees SAMPLE (20);
    print_results();
END;
/
```



注意：

集合自动初始化。如果使用可变数组，则所有返回值必须符合可变数组声明的大小。

与 forall 示例一样，第一步包括声明变量：

```
DECLARE
   TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
   TYPE NameTab IS TABLE OF employees.last_name%TYPE;
   enums NumTab;    -- No need to initialize the collections.
   names NameTab;   -- Values will be filled in by the SELECT INTO.
```

在可执行命令块中，首先通过对嵌套表使用 COUNT 属性来统计结果的数量：

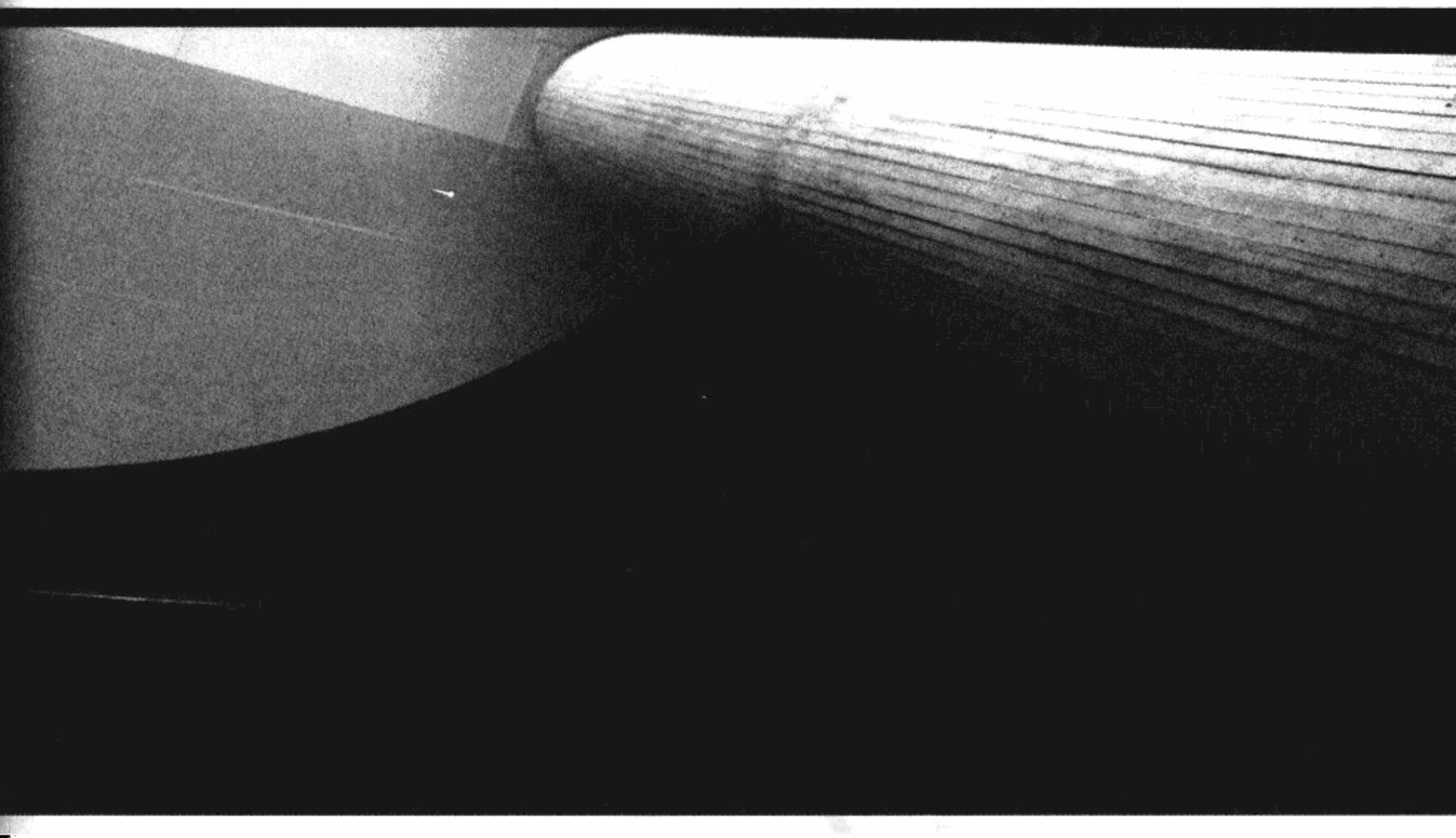
```
BEGIN
  IF enums.COUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE('No results!');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Results:');
    FOR i IN enums.FIRST .. enums.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE(' Employee #' || enums(i) || ': ' || names(i));
    END LOOP;
  END IF;
END;
```

可以使用 bulk collect 运算符来选择在前面的程序清单中声明的嵌套表中的数据：

```
-- Retrieve data for employees with Ids greater than 1000
   SELECT employee_id, last_name
     BULK COLLECT INTO enums, names FROM employees WHERE employee_id > 1000;
-- The data has all been brought into memory by BULK COLLECT
-- No need to FETCH each row from the result set
   print_results();
```

bulk collect 操作和 forall 操作的完整功能描述参见 *Oracle PL/SQL User's Guide and Reference*。从规划的视角来看，当设计定期处理大量数据的代码时，应该考虑批量操作。避免逐行操作，因为逐行操作很难扩展(如前面配置程序示例所示)。

PL/SQL 过程的合理设计涉及理解 SQL、控制路径和数据量。调整工作应该遵循相同的步骤——评估 SQL、分析控制路径的成本、修改编码方法，从而正确地支持数据量。随着应用程序和数据库的成熟，需要定期重新考虑代码，确保它继续满足业务的性能需要。



第V部分

对象关系数据库

第 38 章

实现对象类型、对象视图和方法

本章将详细说明对象类型的用法。本章涵盖了对象类型的安全管理和对象类型属性的索引等内容。还介绍了对象类型的创建方法以及如何使用对象视图和 INSTEAD OF 触发器。

要使用本章提供的信息，首先应当熟悉数据类型(请参阅第 5 章)、视图(请参阅第 17 章)、grant 命令(请参阅第 20 章)、索引(请参阅第 17 章)以及 PL/SQL 过程和触发器(请参阅第 32、34 和 35 章)。每一个主题的元素都是实现对象关系数据库的一个组成部分。

38.1 使用对象类型

可以使用对象类型将相关的列组成对象。例如，通过 create type 命令，可以将构成地址信息的列组成一个 ADDRESS_TY 数据类型。对于此示例，可以使用一个名为 Dora 的账户，

该账户具有 CONNECT 角色和 RESOURCE 角色：

```
rem while connected as Dora:

create type ADDRESS_TY as object
(Street   VARCHAR2(50),
 City     VARCHAR2(25),
 State    CHAR(2),
 Zip      NUMBER);
/
```

上述程序清单中的 create type 命令创建一个 ADDRESS_TY 对象类型。在创建其他的数据库对象时，可以使用 ADDRESS_TY 数据类型。例如，下面的 create type 命令创建 PERSON_TY 数据类型，并将 ADDRESS_TY 数据类型用作 Address 列的数据类型：

```
create type PERSON_TY as object
(Name     VARCHAR2(25),
 Address  ADDRESS_TY);
/
```

由于 PERSON_TY 数据类型的 Address 列使用了 ADDRESS_TY 数据类型，因此，它不是只保存 1 个值，而是保存 4 个值，这 4 个值构成 ADDRESS_TY 数据类型的 4 列。

38.1.1 对象类型的安全性

上述示例假定 ADDRESS_TY 数据类型和 PERSON_TY 数据类型由同一个用户所拥有。但是如果 PERSON_TY 数据类型的所有者与 ADDRESS_TY 数据类型的所有者不同，那么结果会怎样呢？

例如，如果账户名为 Dora 的用户拥有 ADDRESS_TY 数据类型，而账户为 George 的用户试图创建 PERSON_TY 数据类型，那么结果会怎样呢？George 执行如下的命令：

```
rem while connected as George:

create type PERSON_TY as object
(Name VARCHAR2(25),
 Address ADDRESS_TY);
/
```

如果 George 没有 ADDRESS_TY 对象类型，则 Oracle 将以如下消息响应 create type 命令：

```
Warning: Type created with compilation errors.
```

此命令的编译错误是由创建构造函数方法(Constructor Method)——创建此数据类型时由 Oracle 创建的一种特殊的方法——的问题造成的。Oracle 不能解析对 ADDRESS_TY 数据类型的引用，因为 George 没有 ADDRESS_TY 数据类型。George 可以再次执行 create type 命令(使用 or replace 子句)来明确地引用 Dora 的 ADDRESS_TY 数据类型：

```
create or replace type PERSON_TY as object
(Name     VARCHAR2(25),
 Address  Dora.ADDRESS_TY);
/

Warning: Type created with compilation errors.
```

要查看与数据类型的创建相关的错误，可以使用 show errors 命令：

```
show errors
Errors for TYPE PERSON_TY:

LINE/COL     ERROR
--------     -----------------------------------------------------------
0/0          PL/SQL: Compilation unit analysis terminated
3/11         PLS-00201: identifier 'DORA.ADDRESS_TY' must be declared
```

除非 Dora 首先授予 George 在 ADDRESS_TY 数据类型上的 EXECUTE 权限，否则 George 仍然不能创建 PERSON_TY 数据类型(包括 ADDRESS_TY 数据类型)。以下程序清单显示了此授权命令：

```
rem while connected as Dora:

grant EXECUTE on ADDRESS_TY to George;
```

正确地授权以后，George 就可以创建一个基于 Dora 的 ADDRESS_TY 数据类型的数据类型了，如下面的程序清单所示：

```
rem while connected as George:

create or replace type PERSON_TY as object
(Name VARCHAR2(25),
 Address Dora.ADDRESS_TY);
/
```

使用另一个用户的对象类型有时很重要。例如，在插入操作期间中，必须指明每个类型的所有者或者使用一个同义词。George 可以创建基于其数据类型 PERSON_TY(包括 Dora 的 ADDRESS_TY 对象类型)的表，如下面的程序清单所示：

```
create table CUSTOMER
(Customer_ID  NUMBER,
 Person       PERSON_TY);
```

如果 George 拥有 PERSON_TY 和 ADDRESS_TY 数据类型，则对 CUSTOMER 表进行 insert 操作时可以使用下面的格式：

```
insert into CUSTOMER values
(1,PERSON_TY('SomeName',
  ADDRESS_TY('StreetValue','CityValue','ST',11111)));
```

因为 George 没有 ADDRESS_TY 数据类型，所以上述命令将失败。在插入操作期间，使用了 ADDRESS_TY 的构造函数方法——Dora 拥有该方法。因此，必须修改 insert 命令，以指定 Dora 为 ADDRESS_TY 的所有者。下面的示例显示了改正后的 insert 语句，对 Dora 的引用如粗体部分所示：

```
insert into CUSTOMER values
  (1,PERSON_TY('SomeName',
    Dora.ADDRESS_TY('StreetValue','CityValue','ST',11111)));
```

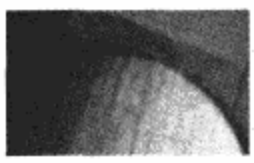
George 可以创建一个名为 ADDRESS_TY 的同义词来简化 SQL：

```
create synonym ADDRESS_TY for Dora.ADDRESS_TY;
```

该同义词可以用于查询、DML 和 DDL。

在查询与插入中，可以将同义词用于另外一个用户的对象类型。例如，如果 ADDRESS_TY 同义词存在，则 George 的插入操作可以执行：

```
insert into CUSTOMER values
  (1,PERSON_TY('SomeName',
     ADDRESS_TY('StreetValue','CityValue','ST',11111)));
```



注意：

创建同义词时，Oracle 不检查正在为其创建同义词的对象的有效性。执行 create synonym x for y 时，Oracle 也不检查 y 是否为有效的对象名或为有效的对象类型。只有在通过该同义词访问此对象时才检查此对象的有效性。

在纯关系的 Oracle 实现中，您可以授予在过程对象上的(如过程和程序包)EXECUTE 权限。在 Oracle 的对象关系实现中，EXECUTE 权限还加以扩展而涵盖对象类型。因为对象类型可以包含方法(即在此数据类型上运行的 PL/SQL 的函数和过程)，所以 EXECUTE 权限是合适的。如果授予某人使用对象类型的权限，也就同时授权了该用户在此对象类型上执行所定义方法的权限。尽管 Dora 还没有在 ADDRESS_TY 数据类型上定义任何方法，但 Oracle 可自动创建访问该数据的构造函数方法。使用 ADDRESS_TY 对象类型的任何对象(如 PERSON_TY)均可使用与 ADDRESS_TY 相关的构造函数方法。即使没有为对象类型创建任何方法，也仍然有与之相关的过程。

可以用如下方法描述 CUSTOMER：

```
describe CUSTOMER

 Name                                      Null?    Type
 ----------------------------------------- -------- ---------
 CUSTOMER_ID                                        NUMBER
 PERSON                                             PERSON_TY
```

可以使用 SQL*Plus 中的 set describe depth 命令显示使用对象-关系特性的表的数据类型属性。可以指明深度值，其范围是 1~50：

```
set describe depth 2

desc CUSTOMER
  Name                                      Null?    Type
  ----------------------------------------- -------- ---------------
  CUSTOMER_ID                                        NUMBER
  PERSON                                             PERSON_TY
    NAME                                             VARCHAR2(25)
    ADDRESS                                          DORA.ADDRESS_TY

set describe depth 3

desc customer

  Name                                      Null?    Type
  ----------------------------------------- -------- ---------------
  CUSTOMER_ID                                        NUMBER
  PERSON                                             PERSON_TY
    NAME                                             VARCHAR2(25)
    ADDRESS                                          DORA.ADDRESS_TY
      STREET                                         VARCHAR2(50)
      CITY                                           VARCHAR2(25)
      STATE                                          CHAR(2)
      ZIP                                            NUMBER
```

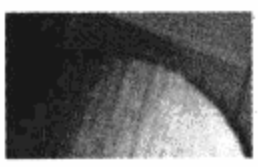
38.1.2　索引对象类型属性

如前面的程序清单所示，CUSTOMER 表包括一个普通列(Customer_ID)和一个由 PERSON_TY 对象类型定义的 Person 列。

从本章前面部分所示的数据类型定义中，可以看到：在由 ADDRESS_TY 数据类型定义的 Address 列之前，PERSON_TY 还有一列(Name)。

在查询过程中，在引用对象类型中的列时，要完全限定数据类型的属性，通过在属性名前加列名并在列名前加关联变量来实现。例如，下面的查询返回 Customer_ID 列和 Name 列。因为 Name 列是定义 Person 列的数据类型的一个属性，所以可以称该属性为 Person.Name：

```
select Customer_ID, C.Person.Name
    from CUSTOMER C;
```



注意：

当查询一个对象类型的属性时，必须使用表的关联变量，如本示例所示。

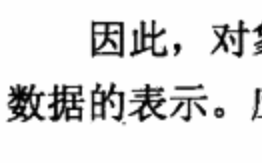
为了设置 SQL*Plus 中对象类型的属性的显示属性，可以使用 column 命令指定数据类型名和属性：

```
column Customer_ID format 9999999
column Person.Name format A25
```

通过指定相关列的完整路径，可以引用 ADDRESS_TY 对象类型中的属性。例如，Street 列被称为 Person.Address.Street，它充分地描述了该列在表结构中的位置。在下面的示例中，City 列被引用两次——一次是在要选择列的列表中，另一次是在 where 子句中。每次引用都需要一个关联变量(该示例中为 C)。

```
select C.Person.Name,
       C.Person.Address.City
  from CUSTOMER C
where C.Person.Address.City like 'C%';
```

由于 City 列用在 where 子句的范围搜索中，因此 Oracle 优化程序在解析此查询时能够使用索引。如果索引在 City 列可用，Oracle 就能迅速找到 City 值以字母 C 开头的所有行，如上面的示例所示。

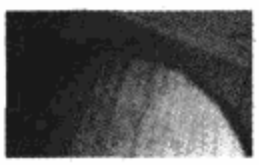


注意：
有关索引列的详细内容，请参阅第 46 章。

为创建对象类型中某一列的索引，应当在 create index 命令中指明该列的完整路径。为创建 City 列(它是 Address 列的一部分)的索引，可以执行以下命令：

```
create index I_CUSTOMER$CITY
on CUSTOMER(Person.Address.City);
```

此命令将在 Person.Address.City 列上创建一个名为 I_CUSTOMER$CITY 的索引。无论何时访问 City 列，Oracle 优化程序都将评估用于访问数据的 SQL，并确定新的索引是否能提高该查询的性能。



注意：
在 create index 命令中，引用对象类型的属性时，一般不使用关联变量。

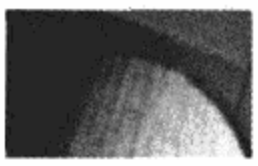
创建基于对象类型的表时，应当考虑如何访问对象类型中的列。和上例中的 City 列一样，如果特定列经常作为查询中的限定条件，则应该对它们创建索引。从这一点来说，单个对象类型中多列的表示方式将会妨碍应用程序的性能，因为它将掩藏在对象类型内为特定列创建索引的需求。使用对象类型时，应当将一组列作为单个实体来对待，如 Address 列或 Person 列。判定查询访问路径时，记住优化程序将单独考虑这些列十分重要。另外，还要记住，在使用 ADDRESS_TY 数据类型的表中索引 City 列不会影响另一个使用 ADDRESS_TY 对象类型的表中的 City 列。例如，如果存在另一个名为 BRANCH、使用 ADDRESS_TY 对象类型的表，则其 City 列不会被索引到，除非为它创建了索引。CUSTOMER 表中的 City 列上有索引这一事实不影响 BRANCH 表中的 City 列。

因此，对象类型的优点之一——提高遵循逻辑数据表示标准的能力——不会扩展到物理数据的表示。应当分别处理每个物理实现。

38.2　实现对象视图

38.1 节中创建的 CUSTOMER 表假定 PERSON_TY 数据类型已经存在。当实现对象关系数据库的应用程序时，应当先使用本书第一部分所描述的关系数据库的设计方法。在数据库设计完全规范化后，应当寻找构成对象类型的表示的列组(或单个的列)。然后，可以基于对象类型创建表，如本章前面所示。

但是，如果表已经存在了，应该怎么办？如果以前已经创建了一个关系数据库应用程序，而现在想要在该应用程序中实现对象-关系的概念，但又不想重建整个应用程序，那么应该怎么办呢？为了做到这一点，需要在已有关系表上覆盖面向对象(OO)的结构，如对象类型。Oracle 用对象视图作为定义已有关系表使用的对象的一种方法。



注意：
在本书的所有示例中，对象视图的名称总是以后缀 OV 结尾。

在本章前面的示例中，操作的顺序如下：

(1) 创建 ADDRESS_TY 数据类型。

(2) 使用 ADDRESS_TY 数据类型创建 PERSON_TY 数据类型。

(3) 使用 PERSON_TY 数据类型创建 CUSTOMER 表。

若 CUSTOMER 表已经存在，就可以创建 ADDRESS_TY 数据类型和 PERSON_TY 数据类型，并可以使用对象视图将它们与 CUSTOMER 表联系起来。在下面的程序清单中，CUSTOMER 表作为关系表创建，只使用内置的数据类型。为了使这些示例分隔开来，应在新的模式下创建这些对象：

```
rem connected as a new user:

drop table CUSTOMER;
create table CUSTOMER
(Customer_ID NUMBER primary key,
 Name        VARCHAR2(25),
 Street      VARCHAR2(50),
 City        VARCHAR2(25),
 State       CHAR(2),
 Zip         NUMBER);
```

如果想要创建另外的表或应用程序来存储关于人员和地址的信息，则可以选择创建 ADDRESS_TY 和 PERSON_TY 对象类型。但是，为了保持一致，也应该将它们应用到 CUSTOMER 表上。

由于已经创建了 CUSTOMER 表(并可能包含数据)，因此应当创建对象类型。首先，创建 ADDRESS_TY 对象类型：

```
drop type PERSON_TY;
drop type ADDRESS_TY;
drop synonym ADDRESS_TY;

create or replace type ADDRESS_TY as object
(Street  VARCHAR2(50),
 City    VARCHAR2(25),
 State   CHAR(2),
 Zip     NUMBER);
/
```

接下来，创建使用 ADDRESS_TY 数据类型的 PERSON_TY 数据类型：

```
create or replace type PERSON_TY as object
(Name VARCHAR2(25),
 Address ADDRESS_TY);
/
```

现在，使用已经定义的对象类型创建基于 CUSTOMER 表的对象视图。对象视图以 create view 命令开始创建。在 create view 命令中，要指定构成该视图基础的查询。可以使用刚刚创建的数据类型。创建 CUSTOMER_OV 对象视图的程序清单显示如下：

```
create view CUSTOMER_OV (Customer_ID, Person) as
select Customer_ID,
       PERSON_TY(Name,
       ADDRESS_TY(Street, City, State, Zip))
  from CUSTOMER;
```

下面来仔细分析此命令。第一行为对象视图命名：

```
create view CUSTOMER_OV (Customer_ID, Person) as
```

CUSTOMER_OV 视图将有两列：Customer_ID 列和 Person 列(后者通过 PERSON_TY 对象类型定义)。请注意，不能在 create view 命令中指定“对象”为一个选项。

在 create view 命令的下一部分中，可以创建构成该视图基础的查询：

```
select Customer_ID,
       PERSON_TY(Name,
       ADDRESS_TY(Street, City, State, Zip))
  from CUSTOMER;
```

该示例中出现了几个重要的语法问题。当在已有对象类型上构建表时，可以在该表中通过引用列名(如前面示例中的 Person 和 Address)而不是引用它们的构造函数方法来选择列值。但是，当创建对象视图时，还是要引用构造函数方法(PERSON_TY 和 ADDRESS_TY)的名称。

可以在构成对象视图基础的查询中使用 where 子句。在下面的示例中，修改 CUSTOMER_OV 使其包含一个 where 子句，该子句限定它只显示在 State 列中含有“DE”的 CUSTOMER 值：

```
create or replace view CUSTOMER_OV (Customer_ID, Person) as
```

```
select Customer_ID,
       PERSON_TY(Name,
       ADDRESS_TY(Street, City, State, Zip))
  from CUSTOMER
 where State = 'DE';
```

请注意，当创建对象视图 CUSTOMER_OV 时，该视图基本查询的 where 子句不引用对象类型。相反，它直接引用 CUSTOMER 表。where 子句直接引用 State 列，因为该查询直接引用 CUSTOMER 表，该表是一个不基于任何对象类型的关系表。

要创建基于关系表的对象视图，其操作顺序如下：

(1) 创建 CUSTOMER 表(如果它不存在)。

(2) 创建 ADDRESS_TY 数据类型。

(3) 用 ADDRESS_TY 数据类型创建 PERSON_TY 数据类型。

(4) 用已经定义的数据类型创建 CUSTOMER_OV 对象视图。

使用对象视图有两大好处。首先，它允许在已有表中创建对象类型。由于可以在应用程序的多个表中使用相同的对象类型，因此可以提高应用程序的数据表示标准的一致性和重用已有对象的能力。由于可以定义对象类型的方法，因此这些方法将应用到任何新表及已有表的数据中。

其次，对象视图用两种不同的方法将数据输入基表中。对象视图的数据操作灵活性(能够视基表为关系表或对象表)给应用程序开发人员带来了很大的好处。38.2.1 节将介绍对象视图可用的数据操作选项。

38.2.1　通过对象视图操作数据

可以通过 CUSTOMER_OV 对象视图更新 CUSTOMER 表中的数据，或者直接更新 CUSTOMER 表。通过将 CUSTOMER 视为一个表，可以用普通的 SQL insert 命令将数据插入其中，如下例所示：

```
insert into CUSTOMER values
  (123,'SIGMUND','47 HAFFNER RD','LEWISTON','NJ',22222);
```

此 insert 命令在 CUSTOMER 表中插入一条记录。即使已经在该表上创建了一个对象视图，也仍然应将 CUSTOMER 看作一个正常的关系表。

由于已经在 CUSTOMER 表上创建对象视图，因此可以通过该视图使用的构造函数方法向 CUSTOMER 表插入数据。如本章前面所描述的，当将数据插入使用对象类型的对象中时，应当在 insert 命令内指明构造函数方法的名称。下面的程序清单所示的示例使用 CUSTOMER_TY、PERSON_TY 和 ADDRESS_TY 构造函数方法，将一条记录插入 CUSTOMER_OV 对象视图中：

```
insert into CUSTOMER_OV values
    (234,
    PERSON_TY('EVELYN',
    ADDRESS_TY('555 HIGH ST','LOWLANDS PARK','NE',33333)));
```

因为可以使用这两种方法将值插入 CUSTOMER_OV 对象视图中，所以可以使应用程序执行数据操作的方法标准化。如果 insert 操作都基于对象类型，则可以使用同样类型的 insert 命令代码，而无须考虑相应的对象类型是在表前还是表后创建的。

虽然可以查询刚插入 CUSTOMER 或者 CUSTOMER_OV 的行；但是因为此视图含有一个 where 子句，所以 CUSTOMER_OV 只显示那些满足限制条件的行。

38.2.2 使用 INSTEAD OF 触发器

如果创建一个对象视图，则可以使用 INSTEAD OF 触发器来告诉 Oracle 如何更新作为视图一部分的基表。可以在对象视图或标准关系视图中使用 INSTEAD OF 触发器。

例如，如果一个视图涉及两个表的连接，那么用户在视图中更新记录的能力有限。但是，如果使用 INSTEAD OF 触发器，那么当用户试图通过该视图更改值时，可以告诉 Oracle 如何在表中更新、删除或插入记录。INSTEAD OF 触发器中的程序代码代替了输入的 update、delete 或 insert 命令。

例如，有一个连接 BOODSHELF 表与 BOOKSHELF_AUTHOR 表的视图：

```
create or replace view AUTHOR_PUBLISHER as
select BA.AuthorName, Title, B.Publisher
  from BOOKSHELF_AUTHOR BA inner join BOOKSHELF B
using (Title);
```

可以从该视图中选择相应的值——而且如果使用 INSTEAD OF 触发器，就可以通过该视图执行数据操作。考虑以下记录：

```
select AuthorName, Publisher from AUTHOR_PUBLISHER
  where AuthorName = 'W. P. KINSELLA';

AUTHORNAME                                         PUBLISHER
-------------------------------------------------- ----------
W. P. KINSELLA                                     MARINER
W. P. KINSELLA                                     BALLANTINE
```

如果试图更新 Publisher 的值，则更新会失败：

```
update AUTHOR_PUBLISHER
   set Publisher = 'MARINER'
 where AuthorName = 'W. P. KINSELLA';

set Publisher = 'MARINER'
    *
ERROR at line 2:
ORA-01779: cannot modify a column which maps to a
       non key-preserved table
```

问题是 Oracle 不能确定 BOOKSHELF 表中哪些记录中的哪些出版商要更新。为了通过该视图执行更新，需要使用 INSTEAD OF 触发器。

在下面的程序清单中，创建了 AUTHOR_PUBLISHER 视图的一个 INSTEAD OF 触发器：

```
create or replace trigger AUTHOR_PUBLISHER_UPDATE
  instead of UPDATE on AUTHOR_PUBLISHER
  for each row
  begin
    if :old.Publisher <> :new.Publisher
    then
      update BOOKSHELF
        set Publisher = :new.Publisher
      where Title = :old.Title;
    end if;
    if :old.AuthorName <> :new.AuthorName
    then
      update BOOKSHELF_AUTHOR
        set AuthorName = :new.AuthorName
      where Title = :old.Title;
    end if;
  end;
  /
```

该触发器的第一部分命名触发器，并通过 instead of 子句描述其用途。顾名思义该触发器的功能很明显：用于支持对 AUTHOR_PUBLISHER 视图执行的 update 命令。它是一个行级触发器；可以处理每个变化的行，如下所示：

```
create trigger AUTHOR_PUBLISHER_UPDATE
  instead of UPDATE on AUTHOR_PUBLISHER
  for each row
```

该触发器的下一部分告诉 Oracle 如何处理 update 操作。首先要检查的是触发器体内部的 Publisher 值。如果旧的 Publisher 值等于新的 Publisher 值，则不做任何修改。如果两个值不相同，则 BOOKSHELF 表被更新为新的 Publisher 值：

```
begin
    if :old.Publisher <> :new.Publisher
    then
      update BOOKSHELF
        set Publisher = :new.Publisher
      where Title = :old.Title;
    end if;
```

触发器的下一部分判断 AuthorName 的值是否有变化。如果 AuthorName 的值被修改，则更新 BOOKSHELF 表，以反映新的 AuthorName 值：

```
if :old.AuthorName <> :new.AuthorName
  then
    update BOOKSHELF_AUTHOR
       set AuthorName = :new.AuthorName
     where Title = :old.Title;
  end if;
```

因此，该视图依赖两个表——BOOKSHELF 和 BOOKSHELF_AUTHOR——并且该视图

的update操作可以更新一个表也可以同时更新两个表。为了支持对象视图而引入的INSTEAD OF触发器是应用程序开发的有力工具。

现在可以直接更新 AUTHOR_PUBLISHER 视图并使触发器恰当地更新基表。例如，以下命令将更新 BOOKSHELF 表：

```
update AUTHOR_PUBLISHER
   set Publisher = 'MARINER'
 where AuthorName = 'W. P. KINSELLA';

2 rows updated.
```

可以通过查询 BOOKSHELF 表，来验证 update 操作：

```
select Publisher from BOOKSHELF
 where Title in
(select Title from BOOKSHELF_AUTHOR
 where AuthorName = 'W. P. KINSELLA');

PUBLISHER
--------------------
MARINER
MARINER
```

INSTEAD OF 触发器的功能十分强大。如本示例所示，使用 PL/SQL 内可用的流控制逻辑，可以使用该触发器在不同的数据库表上进行操作。在处理对象视图时，可以使用 INSTEAD OF 触发器将对象视图的 DML 重定位到这些视图的基表上。

38.3 方法

在 CUSTOMER 表上创建对象视图时，定义了该表所使用的对象类型。虽然这些数据类型有助于数据表示的标准化，但也可用于另一个目的。可以定义应用于数据类型的方法，并通过将这些数据类型应用于已有的表，来将这些方法用于那些表中的数据。

考虑 CUSTOMER_OV 对象视图：

```
create or replace view CUSTOMER_OV (Customer_ID, Person) as
select Customer_ID,
       PERSON_TY(Name,
       ADDRESS_TY(Street, City, State, Zip))
  from CUSTOMER;
```

该视图将 PERSON_TY 和 ADDRESS_TY 对象类型应用于 CUSTOMER 表的数据。如果任何方法与这些对象类型有关，它们就应用于表中的数据。创建对象类型时，使用命令 create type。而创建方法时，使用 create type body 命令。

38.3.1 创建方法的语法

在为方法创建主体之前，必须在数据类型的声明中命名此方法。例如，可以先创建一个

新的数据类型 ANIMAL_TY。该数据类型的定义如下面的程序清单所示：

```
create or replace type ANIMAL_TY as object
 (Breed     VARCHAR2(25),
  Name      VARCHAR2(25),
  BirthDate DATE,
 member function AGE (BirthDate IN DATE) return NUMBER);
 /
```

数据类型 ANIMAL_TY 可以作为第 14 章中使用的 BREEDING 表的对象视图的一部分。在 BREEDING 表中，记录了某个后代的姓名、性别、父母及它的出生日期。

在 create type 命令中，以下这个命令行：

```
member function AGE (BirthDate IN DATE) return NUMBER)
```

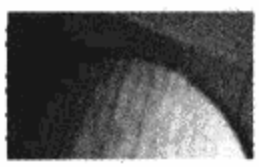
命名作为 ANIMAL_TY 数据类型成员的函数。由于 AGE 将返回一个值，因此它是一个函数(请参阅第 35 章)。要定义 AGE 函数，可使用 create type body 命令，其全部语法可参见附录 A。

下面创建 AGE 函数，作为数据类型 ANIMAL_TY 中的一个成员函数。AGE 函数将返回动物的年龄值，以日期表示：

```
create or replace type body ANIMAL_TY as
 member function Age (BirthDate DATE) return NUMBER is
    begin
  RETURN ROUND(SysDate - BirthDate);
    end;
 /
```

如果还需要编写其他的函数或过程，则在最后一个 end 子句前，可在同一个 create type body 命令中指定它们。

既然已经创建了 AGE 函数，它与对象类型 ANIMAL_TY 就相关联了。如果某个表使用此对象类型，它就可以使用该 AGE 函数。



注意：
不能删除或重新创建某个表正在使用的数据类型。

可以在查询内使用 AGE 函数。如果将 ANIMAIL_TY 用作 ANIMAL 表中 Animal 列的对象类型，则该表的结构如下所示：

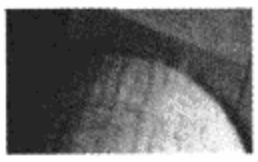
```
create table ANIMAL
 (ID NUMBER,
  Animal ANIMAL_TY);

insert into ANIMAL values
(11,ANIMAL_TY('COW','MIMI',TO_DATE('01-JAN-1997','DD-MON-YYYY')));
```

现在，可以调用 AGE 函数，它是 ANIMAL_TY 对象类型的一部分。和对象类型的属性

一样，成员方法通过使用该数据类型的列名来引用(本例中为 Animal.AGE)。执行查询时，结果将反映当时系统的日期：

```
select A.Animal.AGE(A.Animal.BirthDate)
  from ANIMAL A;

A.ANIMAL.AGE(A.ANIMAL.BIRTHDATE)
--------------------------------
                            2694
```

A.Animal.AGE(A.Animal.BirthDate)调用执行数据类型 ANIMAL_TY 中的 AGE 方法。因而，不必非要在数据库中存储诸如年龄之类的可变数据。相反，可以在数据库中存储静态数据(如出生日期)，并使用函数和过程调用导出可变数据。



注意：
在本示例中，需要在两个地方使用关联变量。一个是在调用 AGE 方法时；另一个是向该方法传递 BirthDate 列名时。

38.3.2 管理方法

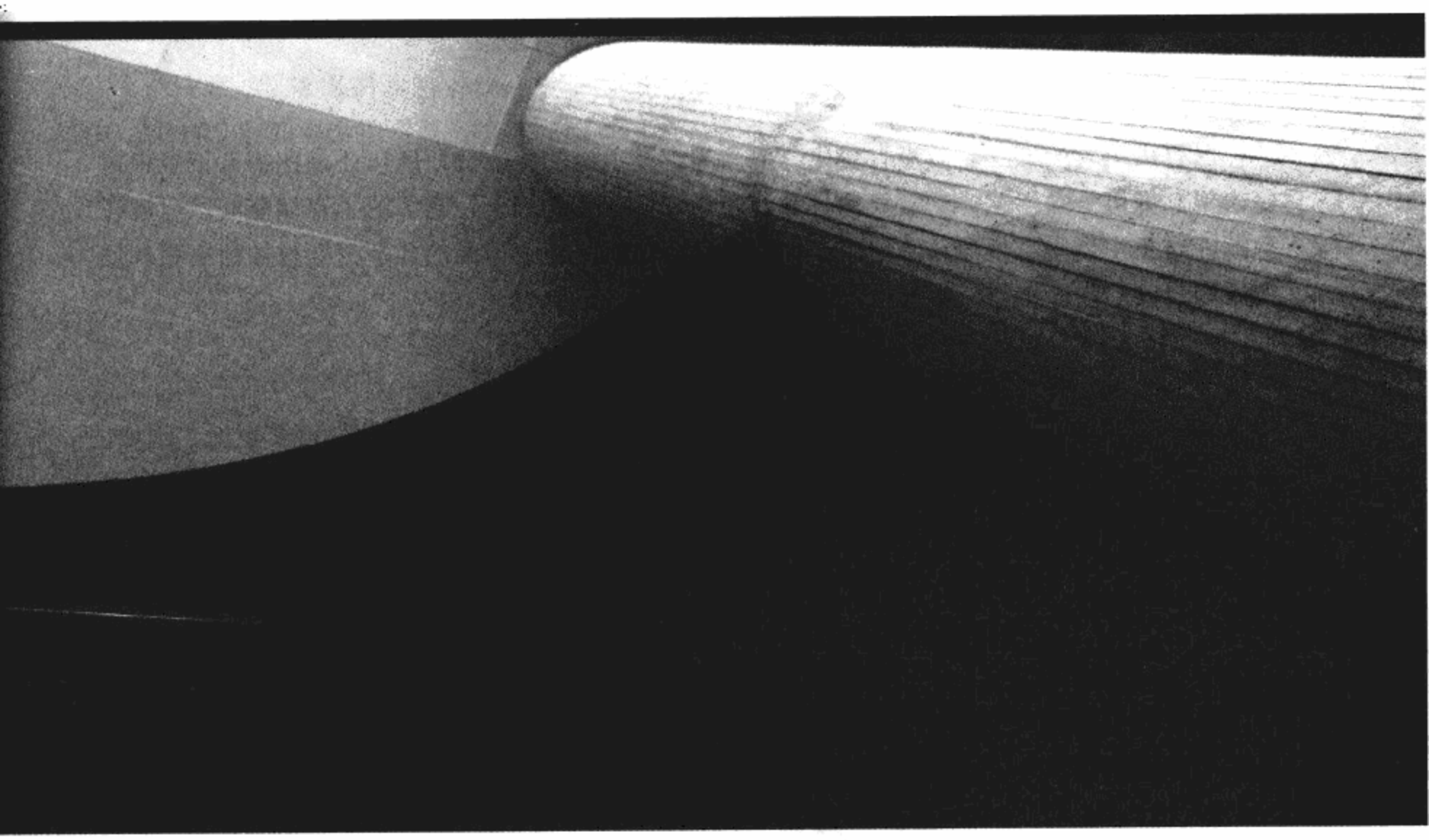
通过更改对象类型(用 alter type 命令)，可以向对象类型添加新的方法。更改对象类型时，应列出此对象类型的所有方法，包括旧的方法的和新的方法。在更改对象类型之后，再更改对象类型体，应当用一个命令列出该对象类型的所有方法。

对于对象类型的成员函数和过程，不需要授予 EXECUTE 权限。如果授予另一个用户在 ANIMAL_TY 对象类型上的 EXECUTE 权限，则该用户将自动拥有在该对象类型的方法上的 EXECUTE 权限。因此，对象类型的访问授权包括数据表示方法和数据访问方法，这与 OO 的概念一致。

在创建成员函数时，可以指定它们为映射方法或排序方法(或均不指定，如同 ANIMAL_TY.Age 成员函数的情况一样)。映射方法(map method)返回一个给定记录的相对位置，即返回对该对象内所有记录排序后给定记录在其中的位置。类型体可只包含一个映射方法，该映射方法必须是一个函数。

排序方法(order method)是一个成员函数，它将该对象中的一条记录作为一个显式的参数，并返回一个整数值。如果返回值为负数、零或正数，则表示该函数隐含的“自身”参数分别小于、等于或大于明确指定的记录。当一个对象中的多行在 order by 子句中进行比较时，排序方法自动将返回的行进行排序。一个对象类型说明可以只包含一个排序方法，该排序方法必须是一个函数，且返回类型是 INTEGER。

当在一个对象类型的多个数据行内考虑收集器(collector，即每行保存多个值的数据类型)的实现时，排序概念可能很有用。Oracle 有两种类型的收集器，分别称为嵌套表和可变数组。第 39 章将介绍如何实现这些收集器类型。



第39章

收集器(嵌套表和可变数组)

收集器使用对象类型。在使用可变数组和嵌套表之前，应当熟悉对象类型的创建和实现(请参阅第38章)。

39.1 可变数组

可变数组允许在一行中重复存储一条记录的属性。例如，假定想要了解哪些工具借给了哪些邻居，可以在关系数据库中创建BORROWER表来模拟这项工作：

```
create table BORROWER
```

```
(Name     VARCHAR2(25),
 Tool     VARCHAR2(25),
constraint BORROWER_PK primary key (Name, Tool));
```

尽管借用者的 Name 值没有变化，但因为它是主键的一部分，所以还是在每条记录中重复出现。像可变数组这样的收集器只允许重复那些变化的列值，它节约了存储空间。可以在数据库对象中使用收集器精确地表示对象类型之间的关系。下面的几节将探讨如何创建和使用收集器。

39.1.1 创建可变数组

可以基于对象类型或者 Oracle 的某种标准数据类型(如 NUMBER)创建可变数组。例如，可以按如下程序清单所示方法创建一个新的对象类型 TOOL_TY，它只有一列；应当限制可变数组为一列。如果需要在一个数组中使用多列，则应考虑使用嵌套表(39.2 节将会介绍)。

```
create type TOOL_TY as object
(ToolName VARCHAR2(25));
/
```

在创建可变数组(名为 TOOLS_VA)前，不必先创建基本类型(在本例中为 TOOL_TY)。因为可变数组只基于一列，所以用一个步骤就可以创建 TOOLS_VA 类型。为了创建该可变数组，应当在 create type 命令中使用 as varray()子句：

```
create or replace type TOOLS_VA as varray(5) of VARCHAR2(25);
/
```

执行此 create type 命令时，创建了名为 TOOLS_VA 的类型。而 as varray(5)子句告诉 Oracle 正在创建可变数组，并且每个记录最多有 5 项。该可变数组的名称为复数的 TOOLS 而不是 TOOL，以表明此类型将用于保存多条记录。后缀 VA 清楚地表明该类型是可变数组。现在，可以将此类型作为列定义的基础，如下面的程序清单所示：

```
drop table BORROWER;

create table BORROWER
(Name           VARCHAR2(25),
 Tools          TOOLS_VA,
constraint BORROWER_PK primary key (Name));
```

39.1.2 描述可变数组

BORROWER 表将为每个借用者创建一条记录，即使借用者有多个工具也是如此。多个工具将存储在 Tools 列中，使用 TOOLS_VA 可变数组。可以按如下方式描述 BORROWER 表：

```
desc BORROWER

Name                            Null?     Type
------------------------------  --------  ------------
NAME                            NOT NULL  VARCHAR2(25)
```

```
TOOLS TOOLS_VA
```

请注意，第 38 章所示的 set describe depth 命令不会影响此输出结果。可以从 USER_TAB_COLUMNS 数据字典视图查询类似的信息：

```
column Data_Type format a12

select Column_Name,
       Data_Type
  from USER_TAB_COLUMNS
 where Table_Name = 'BORROWER';

COLUMN_NAME                    DATA_TYPE
------------------------------ ------------
NAME                           VARCHAR2
TOOLS                          TOOLS_VA
```

USER_TYPES 说明 TOOLS_VA 是一个集合，而且没有属性：

```
select TypeCode,
       Attributes
  from USER_TYPES
 where Type_Name = 'TOOLS_VA';

TYPECODE                       ATTRIBUTES
------------------------------ ----------
COLLECTION                              0
```

关于 TOOLS_VA 的详细信息，请查询 USER_COLL_TYPES 数据字典视图，如下面的程序清单所示：

```
select Coll_Type,
       Elem_Type_Owner,
       Elem_Type_Name,
       Upper_Bound,
       Length
  from USER_COLL_TYPES
 where Type_Name = 'TOOLS_VA';

COLL_TYPE                      ELEM_TYPE_OWNER
------------------------------ ------------------------------
ELEM_TYPE_NAME                 UPPER_BOUND      LENGTH
------------------------------ ----------- ----------
VARYING ARRAY
VARCHAR2                                 5         25
```

如果可变数组基于已有的对象类型，则该对象类型的所有者将显示在 Elem_Type_Owner 列中，并且对象类型的名称将显示在 Elem_Type_Name 列中。由于 TOOLS_VA 使用 VARCHAR2 而不是对象类型，因此 Elem_Type_Owner 列的值为 NULL。

39.1.3 向可变数组中插入记录

由于可变数组是一个对象类型，因此需要使用构造函数方法向使用可变数组的表中插入记录。此外，因为可变数组本身是一个对象类型，所以可能需要嵌套调用多个构造函数方法，以便向使用可变数组的表中插入记录。

下面的命令将向 BORROWER 表中插入一条记录。在下例中，该记录有 Name 列的 1 个值和 Tools 列的 3 个值。

```
insert into BORROWER values
('JED HOPKINS',
 TOOLS_VA('HAMMER','SLEDGE','AX'));
```

此 insert 命令首先指定 Name 列的值。因为 Name 列不是任何对象类型的一部分，所以该值的插入方法与向任何关系列中插入数值的方法相同：

```
insert into BORROWER values
('JED HOPKINS',
```

insert 命令的下一部分把记录插入 Tools 列中。由于 Tools 列使用 TOOLS_VA 可变数组，因此应当使用 TOOLS_VA 构造函数方法：

```
TOOLS_VA('HAMMER','SLEDGE','AX'));
```

由于这是一个可变数组，因此可以将多个值传递给 TOOLS_VA 构造函数方法。如上面的程序清单所示，在一条 insert 命令中列出了三件工具，它们对应于一个借用者名称。如果可变数组基于 TOOL_TY 对象类型，则在 TOOLS_VA 调用中，将嵌套调用 TOOLS_TY：

```
insert into BORROWER values
('JED HOPKINS',
 TOOLS_VA(
    TOOL_TY('HAMMER'),
    TOOL_TY('SLEDGE'),
    TOOL_TY('AX')));
```

可变数组不是基于用户定义的对象类型，而是基于 Oracle 提供的数据类型(仅仅使用一个单独的列)，这一点消除了对其他构造函数方法的嵌套调用，从而简化了 insert 命令。

定义的可变数组 TOOLS_VA 最多包含 5 个值。但在以下 insert 命令中，该行仅输入了 3 个值，而其他两个值未初始化。如果想要将未初始化的值设置为 NULL，则可在 insert 命令中指定：

```
insert into BORROWER values
('JED HOPKINS',
 TOOLS_VA('HAMMER','SLEDGE','AX',NULL,NULL));
```

如果您指定过多的值插入该可变数组中，则 Oracle 将做出如下响应：

```
ORA-22909: exceeded maximum VARRAY limit
```

不能向包含可变数组的表中插入记录，除非了解该数组的结构和数据类型。本示例假定由 BORROWER 表使用的所有数据类型均处于创建 BORROWER 表所用的模式中。若这些对象类型隶属于另一个模式，那么 BORROWER 表的创建者必须具有在这些数据类型上的 EXECUTE 权限。关于使用在其他模式下创建的对象类型的示例，请参阅第 38 章。

39.1.4　从可变数组中选择数据

可以从下表中直接查询可变数组数据：

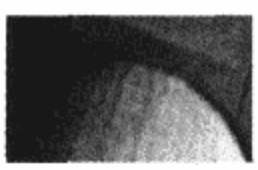
```
select Tools from BORROWER;

TOOLS
----------------------------------------
TOOLS_VA('HAMMER', 'SLEDGE', 'AX')
```

可以使用 from 子句中的 TABLE 函数来显示可变数组中不同行的数据。考虑下面的查询及其输出结果：

```
select B.Name, N.*
  from BORROWER B, TABLE(B.Tools) N;

NAME COLUMN_                  VALUE
---------------------- ------------------------
JED HOPKINS                   HAMMER
JED HOPKINS                   SLEDGE
JED HOPKINS                   AX
JED HOPKINS
JED HOPKINS
```

这是如何工作的呢？TABLE 函数以可变数组的名称作为输入，然后以别名 N 作为输出。选择 N 中的值，这将生成输出结果的第二列。可变数组中的 5 个条目全部显示出来了，包括在 insert 期间未指定的两个 NULL 值。

要在其他行中显示可变数组数据，同样可以使用 PL/SQL。可以调用 LIMIT 查询可变数组以确定每行的最大项数，并调用 COUNT 确定每行的当前项数。为了从可变数组中检索 COUNT、LIMIT 和数据，可以使用一组嵌套的 Cursor FOR 循环，如下面的程序清单所示：



注意：
关于 PL/SQL 和 Cursor FOR 循环的介绍，请参阅第 32 章。

```
set serveroutput on
declare
  cursor borrower_cursor is
    select * from BORROWER;
begin
  for borrower_rec in borrower_cursor
  loop
```

```
    dbms_output.put_line('Contact Name: '||borrower_rec.Name);
    for i in 1..borrower_rec.Tools.Count
    loop
      dbms_output.put_line(borrower_rec.Tools(i));
    end loop;
  end loop;
end;
/
```

下面是该 PL/SQL 脚本的输出结果：

```
Contact Name: JED HOPKINS
HAMMER
SLEDGE
AX

PL/SQL procedure successfully completed.
```

该脚本使用几个 PL/SQL 特性检索可变数组中的数据。由于该数据将通过 DBMS_OUTPUT 程序包(请参阅第 35 章)的 PUT_LINE 过程来显示，因此必须首先能够显示 PL/SQL 的输出结果：

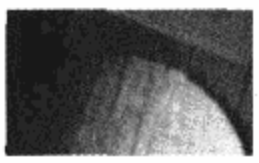
```
set serveroutput on
```

在 PL/SQL 块的声明部分中声明了一个游标。该游标的查询从 BORROWER 表中选择所有的值：

```
declare
    cursor borrower_cursor is
      select * from BORROWER;
```

该 PL/SQL 块的可执行命令部分包含了两个嵌套的 FOR 循环。在第一个循环中，判断游标的每个记录，并通过 PUT_LINE 过程输出 Name 值：

```
begin
    for borrower_rec in borrower_cursor
    loop
      dbms_output.put_line('Contact Name: '||borrower_rec.Name);
```



注意：

通过 PUT_LINE，只能输出字符数据。PL/SQL 将通过 PUT_LINE 输出数值，但是在输出前不能将其与表达式连接。因此，在调用 PUT_LINE 输出数值之前，应当使用 TO_CHAR 函数处理数值数据。

在 PL/SQL 块接下来的部分中，进一步判断行以检索存储在可变数组中的数据。执行第二个 FOR 循环，它嵌套在第一个循环中。该内层 FOR 循环的次数由数组中值的个数限定，循环次数由 COUNT 方法确定(数组中值的最大个数通过 LIMIT 方法得到)。

Tools 数组的值通过该循环逐个地输出：

```
    for i in 1..borrower_rec.Tools.Count
     loop
      dbms_output.put_line(borrower_rec.Tools(i));
```

符号

```
    Tools(i)
```

告诉 Oracle 输出与 i 值相对应的数组值。因此，第一次遍历此循环时，输出数组中的第一个值；然后 i 值增加，输出数组的第二个值。

PL/SQL 块的最后一部分结束这两个嵌套循环，并结束该块：

```
        end loop;
      end loop;
end;
```

由于此方法涉及 PL/SQL 内的嵌套循环，因此从可变数组中选择数据非常有意义的。

请注意，每个借用者可以得到不同数量的工具，范围是 0~5 个。例如，以下程序将在可变数组中插入只包含一项的一行：

```
insert into BORROWER values
('FRED FULLER',
 TOOLS_VA('HAMMER'));
```

可以通过 PL/SQL、TABLE 函数或 SQL 选择数据：

```
select B.Name, N.*
  from BORROWER B, TABLE(B.Tools) N;

NAME COLUMN_                   VALUE
-------------------------- --------------------------
FRED FULLER                HAMMER
JED HOPKINS                HAMMER
JED HOPKINS                SLEDGE
JED HOPKINS                AX
JED HOPKINS
JED HOPKINS
```

为了更灵活地从收集器中选择数据，应当考虑使用 39.2 节将要介绍的嵌套表。

注意：
可以使用 alter type 命令的 modify limit 子句改变可变数组中允许的最大条目数量。

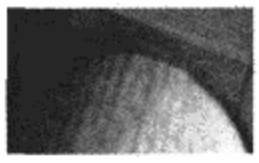
在以下使用嵌套表的示例中，会看到针对变数组和嵌套表可使用的其他功能。

39.2 嵌套表

虽然可变数组有项数限制，但另一种类型的收集器，即嵌套表，却不限制每行中的项数。顾名思义，嵌套表是表中的表。在这种情况下，它表示为另一个表中某一列的一个表。对应主表的每一行，嵌套表可以有多行。

例如，如果有一个关于动物饲养者的表，当然也就会有关于其动物的数据。使用一个嵌套表，就可以将饲养者和他们的动物的信息存储在嵌套表中。考虑第 38 章介绍过的 ANIMAL_TY 数据类型：

```
create or replace type ANIMAL_TY as object
 (Breed    VARCHAR2(25),
  Name     VARCHAR2(25),
  BirthDate DATE);
/
```



注意：
为了简单起见，这里的 ANIMAL_TY 数据类型不包含任何方法。对于该示例，应当删除 ANIMAL 表，如果它在实际模式中存在的话。

ANIMAL_TY 数据类型为每个动物创建一条记录——包括其饲养者、名字和出生日期。为了将该数据类型作为嵌套表的基础，应当创建一个新的数据类型：

```
create type ANIMALS_NT as table of ANIMAL_TY;
/
```

该 create type 命令中的 as table of 子句告诉 Oracle 将使用此类型作为嵌套表的基础。ANIMALS_NT 是类型名，ANIMALS 采用复数形式表明它可以存储多行，后缀 NT 则表示它是一个嵌套表。

现在，可以使用该 ANIMALS_TY 数据类型创建一个关于饲养者的表了：

```
create table BREEDER
 (BreederName    VARCHAR2(25),
  Animals        ANIMALS_NT)
  nested table ANIMALS store as ANIMALS_NT_TAB;
```

在该 create table 命令中，创建 BREEDER 表。该表的第一列为 BreederName 列；第二列为 Animals 列，它定义成一个嵌套表 ANIMALS_NT。

在创建包含嵌套表的表时，必须指定用来存储嵌套表的数据的表名，即嵌套表的数据不与该表的其他数据“内联”地存储在一起。相反，它与主表分开存储。因此，Animals 列的数据将存储在一个表中，而 BreederName 列的数据存储在另一个表中。Oracle 将维护表之间的指针。在该示例中，嵌套表的“外部”数据存储在名为 ANIMALS_NT_TAB 的表中：

```
nested table ANIMALS store as ANIMALS_NT_TAB;
```

该例强调了收集器命名标准的重要性。如果列名使用复数形式(Animals 而非 Animal)，

并且始终使用后缀(如 TY、NT 等)，那么只要观察其名称，就可以知道它是什么对象。此外，因为嵌套表和可变数组可以直接以前面定义的数据类型为基础，所以它们的名称将反映它们所基于的数据类型的名称。这样，ANIMAL_TY 数据类型可以作为 ANIMALS_VA 可变数组的基础，或者作为 ANIMALS_NT 嵌套表的基础。该嵌套表将有一个相关联的名为 ANIMALS_NT_TAB 的外部表。如果您知道有一个名为 ANIMALS_NT_TAB 的表并且您一贯遵守对象命名约定，就自然知道它存储了基于 ANIMAL_TY 数据类型的数据。

39.2.1　指定嵌套表的表空间

当创建具有嵌套表 ANIMALS 的 BREEDER 表时，嵌套表也将存储在同一个表中(即父表)。如果在一个嵌套表中再创建一个嵌套表，这个子表就会创建在同一表空间中(即作为它的父表)，除非指定了表空间。

如下面的程序清单所示，可以用 create table 命令为嵌套表指定表空间。在此示例中，BREEDER 表存储在用户的默认表空间中，而 ANIMALS 嵌套表存储在 USERS 表空间中：

```
drop table BREEDER;

create table BREEDER
(BreederName   VARCHAR2(25),
 Animals       ANIMALS_NT)
 nested table ANIMALS store as ANIMALS_NT_TAB (tablespace USERS);
```

可以通过 alter table 命令中的 move 子句移动嵌套表。如果在 BREEDER 表上执行 alter table move 命令，则它的嵌套表不会被移动。要移动嵌套表，可使用 create table 命令中列出的存储表名：

```
alter table ANIMALS_NT_TAB move tablespace PRACTICE;
```

39.2.2　向嵌套表中插入记录

通过使用其数据类型的构造函数方法，可以向嵌套表中插入记录。因为 Animals 列的数据类型为 ANIMALS_NT；所以可以使用构造函数方法 ANIMALS_NT。依此类推，ANIMALS_NT 类型使用 ANIMAL_TY 数据类型。如下例所示，向 BREEDER 表中插入记录要求同时使用 ANIMALS_NT 和 ANIMAL_TY 两个构造函数方法。在下例中，饲养者 Jane James 有 3 个动物：

```
insert into BREEDER values
    ('JANE JAMES',
      ANIMALS_NT(
           ANIMAL_TY('DOG', 'BUTCH', '31-MAR-01'),
           ANIMAL_TY('DOG', 'ROVER', '05-JUN-01'),
           ANIMAL_TY('DOG', 'JULIO', '10-JUN-01')
    ));
```

此 insert 命令首先指定饲养者的姓名：

```
insert into BREEDER values
    ('JANE JAMES',
```

接下来，输入 Animals 列的值。由于 Animals 列使用 ANIMALS_NT 嵌套表，因此调用 ANIMALS_NT 构造函数方法：

```
ANIMALS_NT(
```

因为 ANIMALS_NT 嵌套表使用 ANIMAL_TY 数据类型，所以对于每个要插入的值均须调用构造函数方法 ANIMAL_TY：

```
ANIMAL_TY('DOG', 'BUTCH', '31-MAR-01'),
     ANIMAL_TY('DOG', 'ROVER', '05-JUN-01'),
     ANIMAL_TY('DOG', 'JULIO', '10-JUN-01')
));
```

通过 describe 命令，可以看到构成该表及其嵌套表的数据类型。使用 SQL*Plus 的 set describe depth 命令可以显示数据类型的属性。

```
set describe depth 2

desc breeder

Name                                      Null?    Type
----------------------------------------- -------- ------------
BREEDERNAME                                        VARCHAR2(25)
ANIMALS                                            ANIMALS_NT
  BREED                                            VARCHAR2(25)
  NAME                                             VARCHAR2(25)
  BIRTHDATE                                        DATE
```

如同本章前面的示例，可以从 USER_TAB_COLUMNS、USER_TYPES、USER _COLL_TYPES 和 USER_TYPE_ATTRS 数据字典视图中查询该元数据。

39.2.3 操作嵌套表

嵌套表支持各种各样的查询。嵌套表是表中的一列。为了支持对嵌套表的行和列的查询，可以使用本章前面引入的 TABLE 函数。

为了弄清如何使用 TABLE 函数，首先考虑嵌套表本身。如果它是关系表的一个普通列，则可以通过常规的 select 命令来查询它：

```
select BirthDate /* This won't work.*/
  from ANIMALS_NT
 where Name = 'JULIO';
```

ANIMALS_NT 不是一个普通表；它是一个数据类型。为了从嵌套表中选择列(如 BirthDate)，首先要使表“平面化”，以便查询。首先，如下面的程序清单所示，将 TABLE 函数应用于嵌套表的列(BREEDER.Animals)：

```
SQL> select BreederName, N.Name, N.BirthDate
  2    from BREEDER, TABLE(BREEDER.Animals) N;

BREEDERNAME               NAME                      BIRTHDATE
------------------------- ------------------------- ---------
JANE JAMES                BUTCH                     31-MAR-01
JANE JAMES                ROVER                     05-JUN-01
JANE JAMES                JULIO                     10-JUN-01
```

使用这种格式的数据，检索相应的行很简单：

```
SQL> select BreederName, N.Name, N.BirthDate
  2    from BREEDER, TABLE(BREEDER.Animals) N
  3  where N.Name = 'JULIO';

BREEDERNAME               NAME                      BIRTHDATE
------------------------- ------------------------- ---------
JANE JAMES                JULIO                     10-JUN-01
```

无论何时需要对嵌套表直接执行 insert 或 update 时，都使用 TABLE 函数。例如，随着时间的推移，饲养者可能将有更多的动物，因此，应当在 BREEDER 表中把新的记录添加到 ANIMALS_NT 嵌套表中。必须将记录插入嵌套表中而不必向 BREEDER 表添加新的记录，并且需要将新的 Animal 记录关联到 BREEDER 表中已有的饲养者。

下面的 insert 语句把一条新的记录添加到 Jane James 的 BREEDER 记录的嵌套表中：

```
SQL> insert into TABLE(select Animals
  2                      from BREEDER
  3                     where BreederName = 'JANE JAMES')
  4  values
  5  (ANIMAL_TY('DOG','MARCUS','01-AUG-01'));
```

该 insert 语句不提供 BreederName 列的值；只把新的记录插入一个饲养者记录的嵌套表中。再次使用 TABLE 函数使 Animals 列(对于 Jane 的行)的嵌套表“平面化”，并插入新值。注意在 TABLE 函数调用中 where 子句的用法，它指定要作用的行。

还可以更新狗的生日。Julio 的生日原来输入为 10-JUN-01，但是后来发现是 01-SEP-01。下面的 update 修改嵌套表中 Julio 的 BirthDate 值。它的语法和结构与本章前面所示的查询示例很相近。

```
SQL> update TABLE(select Animals
  2                   from BREEDER
  3                  where BreederName = 'JANE JAMES') N
  4    set N.BirthDate = '01-SEP-01'
  5  where N.Name = 'JULIO';
```

可以通过查询嵌套表的数据验证此修改：

```
SQL> select BreederName, N.Name, N.BirthDate
  2    from BREEDER, TABLE(BREEDER.Animals) N;
```

```
BREEDERNAME               NAME                      BIRTHDATE
------------------------- ------------------------- ---------
JANE JAMES                BUTCH                     31-MAR-01
JANE JAMES                ROVER                     05-JUN-01
JANE JAMES                JULIO                     01-SEP-01
JANE JAMES                MARCUS                    01-AUG-01
```

还可以删除嵌套表的项：

```
delete TABLE(select Animals
               from BREEDER
              where BreederName = 'JANE JAMES') N
  where N.Name = 'JULIO';

1 row deleted.

select BreederName, N.Name, N.BirthDate
  from BREEDER, TABLE(BREEDER.Animals) N;

BREEDERNAME               NAME                      BIRTHDATE
------------------------- ------------------------- ---------
JANE JAMES                BUTCH                     31-MAR-01
JANE JAMES                ROVER                     05-JUN-01
JANE JAMES                MARCUS                    01-AUG-01
```

39.3 嵌套表与可变数组的附加函数

除了本章所示的一些函数外，Oracle 还提供了很多专门用于收集器的函数。CARDINALITY 函数可以返回嵌套表或可变数组中的元素个数：

```
select CARDINALITY(Animals) from BREEDER;

CARDINALITY(ANIMALS)
--------------------
                   3
```

MULTISET EXCEPT 函数以两个嵌套表作为输入，并返回第一个嵌套表(而不是第二个嵌套表)中的记录集(类似于一个 minus 运算符)。MULTISET INTERSECT 返回两个嵌套表共有的行。MULTISET UNION 计算两个嵌套表的并集，可以指定 MULTISET UNION ALL(默认)或 MULTISET UNION DISTINCT。

SET 函数可以通过删除副本，将一个嵌套表转换成一个集合，并返回一个拥有不同值的嵌套表：

```
set linesize 60
select SET(Animals) from BREEDER;
```

```
SET(ANIMALS)(BREED, NAME, BIRTHDATE)
------------------------------------------------------------
ANIMALS_NT(ANIMAL_TY('DOG', 'BUTCH', '31-MAR-01'), ANIMAL_TY
('DOG', 'ROVER', '05-JUN-01'), ANIMAL_TY('DOG', 'MARCUS', '0
1-AUG-01'))
```

关于这些函数以及 COLLECT、POWERMULTISET 和 POWERMULTISET_BY_CARDINALITY 函数的语法，请参阅附录 A。

39.4　嵌套表和可变数组的管理问题

存储与表有关的数据时，可以选择以下 3 种方法：可变数组、嵌套表或独立表。如果行数有限，则选择可变数组比较合适。但是，当行数增加时，在访问可变数组时就可能会遇到性能问题。这些问题可能是由可变数组的特性引起的：即它们不能被索引。但是嵌套表和标准关系表可以被索引。因此，Varray 收集器的性能可能会随着行数的增长而恶化。

从可变数组中查询数据比较困难，这进一步加重了可变数组的负担。如果对于应用程序来说，可变数组不是良好的解决方案，那么应该使用哪一种方法呢？嵌套表还是独立的关系表？这需要视具体情况而定。由于嵌套表和关系表有不同的用途，因此答案取决于打算使用这些数据做什么。关键区别如下：

- 嵌套表是对象类型，通过 create type 命令创建。因而，它们具有与之相关联的方法。如果打算将数据与方法联系在一起，就应当使用嵌套表而不是关系表。或者，也可以考虑使用带有方法的对象视图，如第 38 章所介绍的那样。
- 关系表容易与其他关系表相关联。如果数据可以与许多其他表关联，则最好不要在一个表中嵌套数据。在自己的表中存储这些数据将会为管理数据关系带来极大的灵活性。

39.4.1　收集器的可变性

可以使用对象类型加强数据库的标准化。然而，用户可能意识不到在使用收集器时规范化同样会有好处。考虑一下本章前面介绍的 TOOLS_VA 可变数组。如果另一个应用程序或表想要使用 TOOLS_VA 可变数组，则应用程序必须使用相同的数组定义，值的最大个数也相同。然而，新的应用程序对于工具的最大数量，可能具有不同的值。这样，必须改变数组，使其支持在任何数组实现中可能包含的值的数量。因此，数组的限定能力的有效性降低了。为了支持多个数组的限定条件，需要创建多个数组，但是这样就失去了在创建可变数组时获得的重用对象的好处。然后必须管理不同数组的限定条件，并了解每个数组的限定条件是什么。

还会发现多个嵌套表相似，其区别只在一两列上。用户可能试图创建一个单独的、包含在多个表中使用的所有嵌套表类型的表，但该表也将会遇到管理问题。需要知道嵌套表的哪些列对哪些实现有效。如果无法使用同样的列结构，使每个实例中这些列都具有同样的内涵和外延，就不应该使用共享的嵌套表对象类型。如果希望在某环境中使用嵌套表，则应该为每个将使用嵌套表的实例创建一个单独的嵌套表对象类型，并且分别管理它们。具有多个嵌

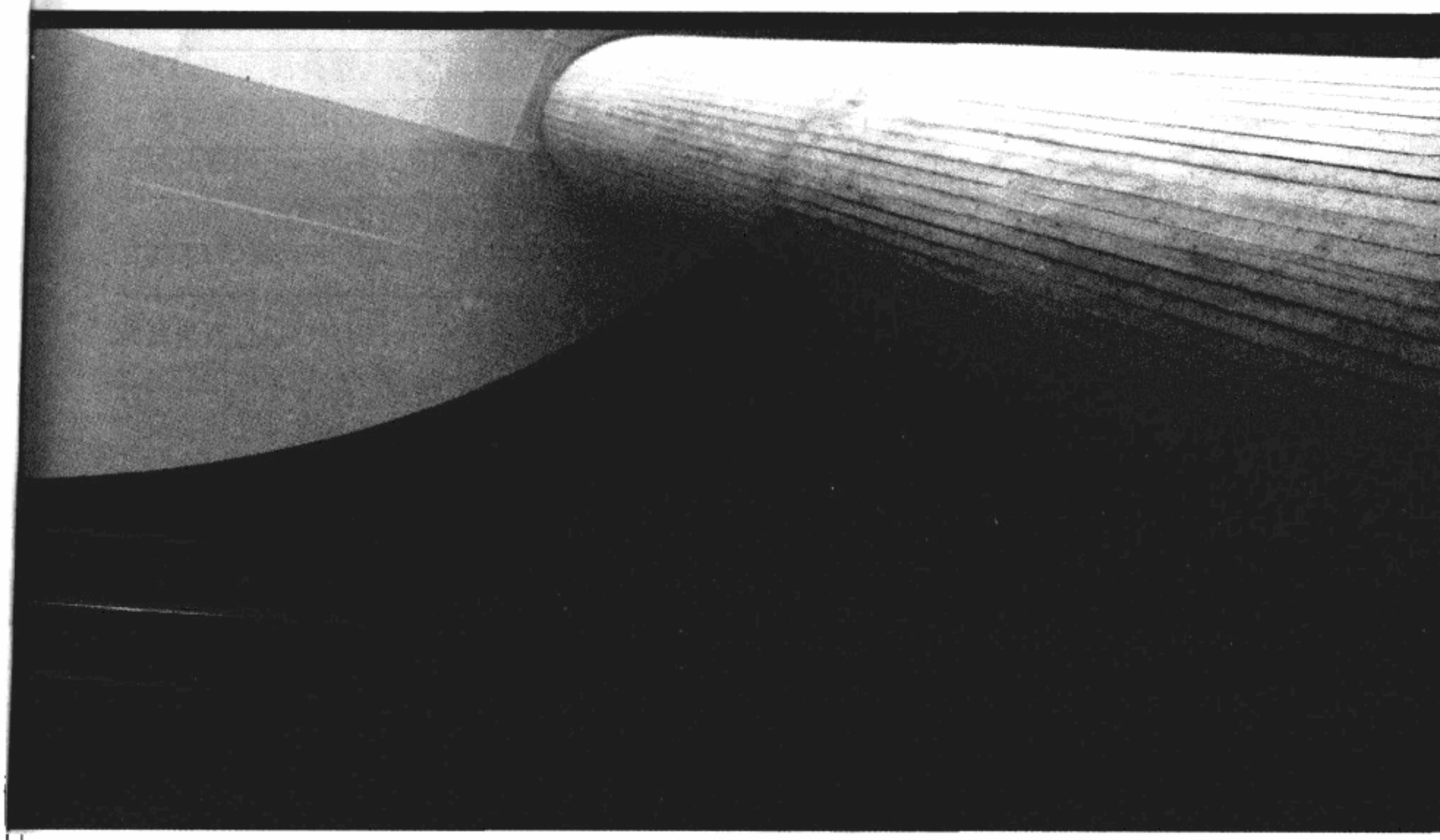
套表对象类型将使数据库的管理复杂化。

39.4.2 数据的位置

在可变数组中，数组中的数据与该表的其他数据存储在一起。但是，在嵌套表中，数据则存储在“外部”(如本章前面的示例所示，数据可能会存储在另一个表空间中)。因此，对于嵌套表，在不涉及收集器数据的查询过程中，扫描到的数据量比可变数组的要少。也就是说，在查询过程中，数据库不需要将嵌套表的数据读一遍；如果使用可变数组，则数据库需要将可变数组的数据读一遍，以查找所需内容。像嵌套表这样使用外部数据，可以改善查询的性能。

外部收集器的数据模拟了数据存储在普通关系数据库应用程序中的方式。例如，在一个关系应用程序中，将在多个独立的表中存储相关的数据(如书籍和作者)，并且这些表在物理上是分开的。虽然外部嵌套表将其数据与主表分开存储，但保持与主表的关系。外部数据通过将针对主表执行的 I/0 分散在多个表上，这样可以改善查询性能。

外部数据还用于存储在内部存储(与 BFILE 相反，BFILE 是指向数据库外部的 LOB 的指针)的大对象(LOB)。第 40 章将介绍如何创建和操作 LOB 值。对象类型、对象视图、收集器和 LOB 的组合为对象-关系数据库应用程序的实现奠定了坚实的基础。



第 40 章

使用大对象

用户可以用 LOB 数据类型(BLOB、CLOB、NCLOB 和 BFILE)代替 LONG 和 LONG RAW，来存储长数据。虽然 Oracle 仍然允许创建 LONG 和 LONG RAW 列，但 Oracle 建议使用 LOB 数据类型创建新列，并建议将现有的 LONG 列和 LONG RAW 列分别转换为 CLOB 列或 BLOB 列。如果使用其中的一种数据类型来存储大对象(LOB)，就可以利用新功能来查看和操作数据。还可以使用 Oracle Text(参见第 27 章)对 CLOB 数据执行文本搜索。

本章将介绍如何使用 LOB 数据类型和如何操作那些甚至不必存储在数据库中的数据。

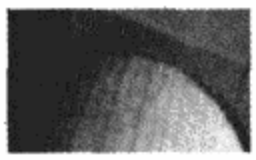
40.1 可用的数据类型

Oracle 支持以下 4 种 LOB 数据类型，如表 40-1 所示。

表 40-1 Oracle 支持的 4 种 LOB 数据类型

LOB 数据类型	说 明
BLOB	二进制 LOB。存储在数据库中的二进制数据
CLOB	字符 LOB。存储在数据库中的字符数据
BFILE	二进制文件。存储在数据库外部的只读二进制数据，其长度受操作系统限制
NCLOB	支持多字节字符集的 CLOB 列

可以在一个表中创建多个 LOB 数据类型。例如，假设想创建一个 PROPOSAL 表来跟踪提交的正式建议。建议记录由一系列字处理文件和电子表格组成，这些文件和表格用来归档和评价所建议的工作。该 PROPOSAL 表将包含 VARCHAR2 数据类型(如包含该建议接受者名字的列)和 LOB 数据类型(包含字处理文件和电子表格文件)。



注意：
不能在一个表中创建多个 LONG 列或 LONG RAW 列。

下列程序清单中的 create table 命令创建 PROPOSAL 表：

```
create table PROPOSAL
 (Proposal_ID          NUMBER(10),
  Recipient_Name       VARCHAR2(25),
  Proposal_Name        VARCHAR2(25),
  Short_Description    VARCHAR2(1000),
  Proposal_Text        CLOB,
  Budget               BLOB,
  Cover_Letter         BFILE,
 constraint PROPOSAL_PK primary key (Proposal_ID));
```

由于可能向同一个接受者提交多项建议，因此可以给每项建议指定一个编号 Proposal_ID。PROPOSAL 表存储建议的接受者、建议的名称和简短说明。然后，利用 Oracle 中可用的 LOB 数据类型，PROPOSAL 表包含以下 3 列：

- Proposal_Text 包含建议文本的 CLOB
- Budget 包含一个电子表格，给出所建议工作的成本和利润计算结果的 BLOB
- Cover_Letter 存储在数据库外部的二进制文件(BFILE)，包含该建议的封面文字

这样，PROPOSAL 表的每一行最多可存储 3 个 LOB 值。虽然 Oracle 将使用普通的数据库功能支持 Proposal 项和 Budget 项的数据完整性和并发性，但不包括 Cover_Letter 的值。因为 Cover_Letter 列使用 BFILE 数据类型，所以它的数据存储在该数据库的外部。对内部来说，数据库只存储用来查找其外部文件的定位器值，从而减少数据库的空间需求和管理需求。Oracle 无法保证存储在数据库外部的 BFILE 文件的数据完整性。并且，当使用 BFILE 数据类型插入记录时，Oracle 也不验证该文件是否存在。数据并发性和完整性只用于维护内部存储的 LOB。

LOB 列的数据可能和 PROPOSAL 表在物理上完全分离。在 PROPOSAL 表中，Oracle

存储指向数据位置的定位器值。对于 BFILE 数据类型，该定位器指向外部文件；而对于 BLOB 和 CLOB 数据类型，定位器指向数据库创建的、用来保存 LOB 数据的独立数据位置。因此，LOB 数据的数据位置可能与 PROPOSAL 表的其他行数据存储在一起，也可能存储在外部。在默认情况下，如果 LOB 数据的长度近似地小于 4 000 字节，它就和其他数据存储在一起。

将数据存储在外部，可以使数据库在读取数据库中的多行时不用每次都扫描 LOB 数据。而 LOB 数据只在需要的时候才读取；否则只读取其定位器值。还可以为存储 LOB 数据的区域指定存储参数(表空间和尺寸)，这些内容将在 40.2 节介绍。

40.2 为 LOB 数据指定存储参数

创建包含 LOB 列的表时，可以为 LOB 数据使用的区域指定存储参数。因此，对于没有 LOB 列的表来说，在 create table 命令中可能会有一个 storage 子句(参见第 17 章)。若该表至少有一个 LOB 列，则需要在 create table 命令中附加一个 lob 子句。

考虑 PROPOSAL 表，这次使用 tablespace 子句：

```
create table PROPOSAL
 (Proposal_ID          NUMBER(10),
  Recipient_Name       VARCHAR2(25),
  Proposal_Name        VARCHAR2(25),
  Short_Description    VARCHAR2(1000),
  Proposal_Text        CLOB,
  Budget               BLOB,
  Cover_Letter         BFILE,
 constraint PROPOSAL_PK primary key (Proposal_ID))
  tablespace PROPOSALS;
```

因为该表有 3 个 LOB 列，所以需要修改 create table 命令，从而包含外部 LOB 数据的存储说明(如果要存储外部 LOB 数据)。其中的两个 LOB 列，即 Proposal_Text 和 Budget，将存储在数据库内部。修改后的 create table 命令带有指定的 LOB 存储子句，如下面的程序清单所示：

```
create table PROPOSAL
 (Proposal_ID          NUMBER(10),
  Recipient_Name       VARCHAR2(25),
  Proposal_Name        VARCHAR2(25),
  Short_Description    VARCHAR2(1000),
  Proposal_Text        CLOB,
  Budget               BLOB,
  Cover_Letter BFILE,
 constraint PROPOSAL_PK primary key (Proposal_ID))
   tablespace PROPOSALS
 lob (Proposal_Text, Budget) store as
 (tablespace Proposal_Lobs
    storage (initial 100K next 100K pctincrease 0)
```

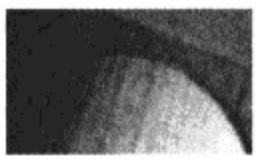
```
                chunk 16K pctversion 10 nocache logging);
```

该 create table 命令的最后 4 行告诉 Oracle 如何存储外部 LOB 数据，其指令如下所示：

```
    lob (Proposal_Text, Budget) store as
     (tablespace Proposal_Lobs
        storage (initial 100K next 100K pctincrease 0)
                 chunk 16K pctversion 10 nocache logging);
```

该 lob 子句告诉 Oracle，下一组命令将处理该表的 LOB 列的外部存储说明。显式地列出了两个 LOB 列(Proposal_Text 和 Budget)。当这两列的值存储在数据库外部时，它们存储在由 lob 子句指定的段中：

```
    lob (Proposal_Text, Budget) store as
```



注意：
如果只有一个 LOB 列，则可以为 LOB 段指定一个名称。应当紧接着上述命令中的 store as 子句命名该段。

接下来的两行为外部 LOB 存储指定表空间和存储参数。tablespace 子句将外部数据分配给 PROPOSAL_LOBS 表空间，而可选的 storage 子句用来指定外部数据存储将要使用的 initial、next 和 pctincrease 值。关于段可用的存储参数的说明，请参见命令和术语参考中的“STORAGE”子句条目。

```
    (tablespace Proposal_Lobs
```

因为这些段用来存储 LOB 数据，所以另外几个参数也可用，它们都在 lob 子句中指定：

```
    chunk 16K pctversion 10
```

LOB 存储参数 chunk 告诉 Oracle 在每个 LOB 值的操作过程中为其分配多大的空间。默认的 chunk 值为 1KB，最大可达 32KB。

pctversion 参数是为创建新的 LOB 所使用的全部 LOB 存储空间的最大百分比。默认情况下，除非使用了 10%的 LOB 可用存储空间，否则将不重写旧的 LOB 数据。

LOB 存储空间的最后两个参数告诉 Oracle 在读写过程中如何处理 LOB 数据：

```
    nocache logging);
```

lob 存储子句中的 nocache 参数表明在查询期间，LOB 值没有存储在内存中是为了快速访问。默认的 lob 存储设置是 nocache；其相反设置为 cache。

logging 参数指定 LOB 数据的所有操作都将记录在数据库的重做日志文件中。logging 的相反值为 nologging，表示不在重做日志文件中记录这些操作，从而提高了表创建操作的性能。还可以为 LOB 索引指定 storage 参数和 tablespace 参数。关于 LOB 索引的相关语法，请参见附录 A 中的“CREATE TABLE”命令项。

如果指定 cache 参数而省略 nologging/logging 子句，logging 就会自动实现，因为不能指

定 cache　nologging。如果没有指定 cache 参数，并且省略 nologging/logging 子句，则将从存储有 LOB 值的表空间中获得 logging 的值。

既然已经创建了表，现在就可以输入数据了。40.3 节将介绍如何向 PROPOSAL 表中插入 LOB 值，以及如何使用 DBMS_LOB 程序包处理这些值。

有多少 LOB 列就可以有多少 lob store as 列。只要将 LOB 列名指定为 lob 子句的一部分即可，如下所示：

```
lob (Proposal_Text, Budget) store as
    (tablespace Proposal_Lobs
       storage (initial 100K next 100K pctincrease 0)
                chunk 16K pctversion 10 nocache logging)
```

40.3 LOB 值的操作和选择

可以用几种方法选择或操作 LOB 数据。可对 LOB 数据类型使用本打算用于 VARCHAR2 列的字符串函数。对于较大的 LOB 值(长度大于或等于 32KB)，或对于更复杂的数据操作来说，应当通过 DBMS_LOB 程序包来操作 LOB 数据。操作 LOB 数据的其他方法还包括使用应用程序接口(API)和 Oracle 调用接口(Oracle Call Interface，OCI)程序。本节将介绍串函数的使用以及 DBMS_LOB 程序包的概述。从给出的示例中不难发现，在数据操作的可能性上，LOB 比 LONG 数据类型更为灵活。

表 40-2 列出了 DBMS_LOB 程序包内可用的过程及函数。

表 40-2　DBMS_LOB 程序包内函数和过程

子　程　序	说　　明
APPEND 过程	将源 LOB 内容追加到目的 LOB
CLOSE 过程	关闭前面打开的内部或外部 LOB
COMPARE 函数	比较两个 LOB 的部分或全部
COPY 过程	将源 LOB 的全部或部分内容复制到目的 LOB
CONVERTTOBLOB 过程	从源 CLOB 或 NCLOB 实例中读取字符数据，将字符数据转换为指定的字符，以二进制格式将转换后的数据写入目的 BLOB 实例，并返回新的偏移量
CONVERTOCLOB 过程	从源 CLOB 或 NCLOB 实例中读取字符数据，将字符数据转换为指定的字符，以二进制格式将转换后的数据写入目的 BLOB 实例，并返回新的偏移量
CREATETEMPORARY 过程	创建临时的 BLOB 或 CLOB 及其在用户的默认临时表空间中的索引
ERASE 过程	删除 LOB 的全部或部分内容
FILECLOSE 过程	关闭文件
FILECLOSEALL 过程	关闭以前打开的所有文件
FILEEXISTS 函数	检查文件是否在服务器上
FILEGETNAME 过程	获取目录别名和文件名
FILEISOPEN 函数	检查文件是否已经使用输入 BFILE 定位器打开

(续表)

子 程 序	说 明
FILEOPEN 过程	打开文件
FRAGMENT_DELETE 过程	删除 LOB 中给定长度给定偏移量的数据
FRAGMENT_INSERT 过程	将给定偏移量的数据(限制为 32KB)插入 LOB 中
FRAGMENT_MOVE 过程	将字节(BLOB)或字符(CLOB)数量从给定偏移量移动到新指定的偏移量
FRAGMENT_REPLACE 过程	将给定偏移量的数据更换为给定数据(不超过 32KB)
FREETEMPORARY 过程	释放用户默认临时表空间的临时 BLOB 或 CLOB
GETCHUNKSIZE 函数	返回在 LOB 块中用来存储 LOB 值的空间量
GETLENGTH 函数	获取 LOB 值的长度
GET_OPTIONS 函数	获取特定 LOB 与 option_types 字段相对应的设置
GET_STORAGE_LIMIT 函数	返回数据库配置中 LOB 的存储限制
INSTR 函数	返回模式在 LOB 中出现的第 n 个匹配的位置
ISOPEN 函数	确定 LOB 是否已经用输入定位器打开
ISTEMPORARY 函数	检查定位器是否指向一个临时 LOB
LOADFROMFILE 过程	把 BFILE 数据加载到某个内部 LOB
LOADBLOBFROMFILE 过程	把 BFILE 数据加载到某个内部 BLOB
LOADCLOBFROMFILE 过程	把 BFILE 数据加载到某个内部 CLOB
OPEN 过程	以指定的方式打开 LOB(内部、外部或临时)
READ 过程	从指定的偏移量处开始读取 LOB 数据
SETOPTIONS 过程	启用基于每个 LOB 的 CSCE 功能，重写默认的 LOB 列设置
SUBSTR 过程	从指定的偏移量开始返回部分 LOB 值
TRIM 过程	将 LOB 值修剪为指定的长度
WRITE 过程	从指定的偏移值处将数据写入 LOB
WRITEAPPEND 过程	将某个缓冲区写入一个 LOB 的末尾

40.3.1 初始化值

再次考虑 PROPOSAL 表。Proposal_ID 列是 PROPOSAL 表的主键列。PROPOSAL 表包含 3 个 LOB 列，1 个 BLOB 列，1 个 CLOB 列和 1 个 BFILE 列。对于每个 LOB 列，Oracle 将存储一个定位器值(locator value)，以便找到该记录存储的任何外部数据。

在向包含 LOB 的表中插入记录时，可以用函数告诉 Oracle 为内部存储的 LOB 列创建一个空的定位器值。空的定位器值与 NULL 值不同。如果一个内部存储的 LOB 列的值为 NULL，那么在它更新为非 NULL 值前，必须先将其设置为空的定位器值。

假如开始起草一份建议书，并在 PROPOSAL 表中输入一条记录。这时，既没有预算电子表格，也没有封面文字。insert 命令如下面的程序清单所示：

```
insert into PROPOSAL
    (Proposal_ID, Recipient_Name, Proposal_Name, Short_Description,
     Proposal_Text,
     Budget, Cover_Letter)
    values
    (1, 'DOT RODALE', 'MAINTAIN ORGANIC GARDEN', NULL,
     'This is the text of a proposal to maintain an organic garden.',
     EMPTY_BLOB(),NULL);
```

所插入记录的 Proposal_ID 值为 1，Recipient_Name 值为 DOT RODALE。Proposal_Name 值为 MAINTAIN ORGANIC GARDEN，而 Short_Description 列现在仍然为 NULL。Proposal _ Text 列暂时设置为一个短字符串。为了将 Budget 列设为空的定位器值，可以使用函数 EMPTY_ BLOB 函数。如果想设置 CLOB 数据类型的列等于空的定位器值，则可使用 EMPTY_CLOB 函数。由于 Cover_Letter 列是一个外部存储的 BFILE 值，因此设置为 NULL。

为了设置 LOB 列为空的定位器值，必须知道以下数据类型：

- BLOB　　使用 EMPTY_BLOB()
- CLOB　　使用 EMPTY_CLOB()
- NCLOB　　使用 EMPTY_CLOB()
- BFILE　　使用 BFILENAME

可以使用 BFILENAME 过程指向一个目录和文件名的组合。在向 BFILENAME 函数中输入一个目录的值之前，具有 DBA 角色或 CREATE ANY DIRECTORY 系统权限的用户必须创建该目录。要创建目录，可用如下所示的 create directory 命令：

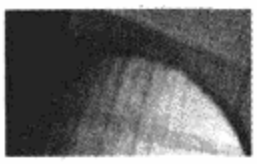
```
create directory proposal_dir as '/u01/proposal/letters';
```

在插入 BFILE 类型的项时，应当引用逻辑目录名(如 proposal_dir)，而不应当引用操作系统的物理目录名。具有 DBA 权限的用户可以把对目录名的 READ 访问权授予用户。详细的内容请参阅附录 A 中的“CREATE DIRECTORY”命令。

现在，可以输入另一个 PROPOSAL 记录了；此记录具有 Cover_Lever 值：

```
insert into PROPOSAL
    (Proposal_ID, Recipient_Name, Proposal_Name, Short_Description,
     Proposal_Text, Budget,
    Cover_Letter)
    values
    (2, 'BRAD OHMONT', 'REBUILD FENCE', NULL,
     EMPTY_CLOB(), EMPTY_BLOB(),
     BFILENAME('proposal_dir','P2.DOC'));
```

在上述程序清单中，第二个建议，即重建栅栏(REBUILD FENCE)，插入 PROPOSAL 表中。可以选择性地使用 EMPTY_CLOB()函数和 EMPTY_BLOB()函数在表中插入空的定位器值。函数 BFILENAME 准确地告诉 Oracle 在哪里可以找到 Cover_Lever——它在目录 proposal_dir 中，名为 P2.DOC。在 BFILENAME 函数中，第一个参数始终为目录名，而第二个参数是该目录中的文件名。



注意：
为了成功地插入记录，P2.DOC 文件一定要存在。

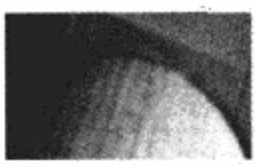
从 LOB 列中选择数据时，Oracle 用定位器值查找与该 LOB 数据相关联的数据。不需要了解或指定定位器值。如下面各节将要介绍的，用 LONG 数据类型不能实现的操作可以用 LOB 值来实现。

40.3.2 用子查询插入数据

如果想要复制一条建议记录，该怎么办？例如，假设开始着手考虑第三个建议，它与第一个建议非常相似：

```
insert into PROPOSAL
    (Proposal_ID, Recipient_Name, Proposal_Name, Short_Description,
     Proposal_Text, Budget, Cover_Letter)
    select 3, 'SKIP GATES', 'CLEAR GATES FIELD', NULL,
              Proposal_Text, Budget, Cover_Letter
     from PROPOSAL
    where Proposal_ID = 1;
```

这条 insert 命令告诉 Oracle 查找 Proposal_ID 值等于 1 的 PROPOSAL 记录。获取该记录的 Proposal_Text 列、Budget 列和 Cover_Letter 列的值，以及指定的字面值(如 3、SKIP GATES 等)，并且将一条新记录插入 PROPOSAL 表中。请注意，插入的 LOB 列将设置为空的定位器值。这种基于查询执行 insert 操作的能力是使用 LOB 数据类型的一个显著优点——如果 LONG 列是查询的组成部分，则不能执行这种 insert 操作。



注意：
如果用 insert as select 插入 LOB 数据值，则可能会有多个指向同一个外部文件的 BFILE 值。这时，需要更新这些新值，使其指向正确的外部文件。Oracle 不维护外部文件的数据完整性。

40.3.3 更新 LOB 值

在创建的第 3 条记录中，Proposal_Text 值从第一条记录复制。要更新这个 Proposal_ID 为 3 的记录的 Proposal_Text 值，可执行以下命令：

```
update PROPOSAL
   set Proposal_Text = 'This is the new proposal text.'
  where Proposal_ID = 3;
```

还可以更新 Cover_Letter 列，使其指向正确的封面文字。可以用 BFILENAME 函数使其指向正确的文件：

```
update PROPOSAL
   set Cover_Letter = BFILENAME('proposal_dir', 'P3.DOC')
 where Proposal_ID = 3;
```

可以更新 BFILE 列的 NULL 值，而不用先设置 BFILE 列值为空的定位器值。

40.3.4　使用串函数处理 LOB 值

可以对 CLOB 值使用 Oracle 提供的串函数。例如，可以使用 SUBSTR、INSTR、LTRIM 和 INITCAP 修改串值。关于这些内置的串函数，请参阅第 7 章。

在 PROPOSAL 表中，Proposal_Text 列定义为 CLOB 数据类型。下面给出了 3 个 Proposal_Text 值(第二个为 NULL)：

```
select Proposal_Text from PROPOSAL;

PROPOSAL_TEXT
-------------------------------------------------------------
This is the text of a proposal to maintain an organic garden.

This is the new proposal text.
```

对第三条记录使用 INITCAP，得出如下结果：

```
select INITCAP(Proposal_Text) from PROPOSAL
 where Proposal_ID = 3;

INITCAP(PROPOSAL_TEXT)
-----------------------------------------------
This Is The New Proposal Text.
```

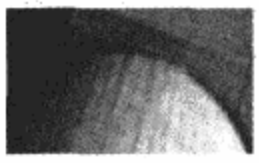
对所有 3 条记录使用 INSTR 后，返回的预期结果为：

```
select INSTR(Proposal_Text,'new',1,1) from PROPOSAL;

INSTR(PROPOSAL_TEXT,'NEW',1,1)
------------------------------
                             0
                             0
                            13
```

可以使用 SUBSTR 返回文本的前 10 个字符：

```
select SUBSTR(Proposal_Text,1,10) from PROPOSAL
  where Proposal_ID = 3;

SUBSTR(PROPOSAL_TEXT,1,10)
----------------------------------
This is th
```

对于复杂的操作，应当如以下几节介绍的那样，使用 DBMS_LOB 程序包。



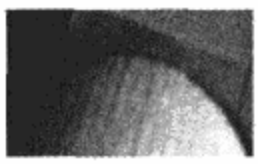
注意：
对 LOB 数据类型可以使用正则表达式函数(参见第 8 章)。

40.3.5 使用 DBMS_LOB 操作 LOB 值

可以使用 DBMS_LOB 程序包更改或选择 LOB 值。下面几节将介绍如何对 LOB 值执行 SUBSTR 和 INSTR 等 SQL 函数，以及如何追加、比较和读取 LOB 值。

表 40-2 列出了 DBMS_LOB 程序包内可用的串比较和串操作的过程及函数。

下面几节将讲述 DBMS_LOB 程序包内可用的主要过程及函数。当使用 BFILE 时，最多可并行打开的文件个数由数据库的系统参数文件中 SESSION_MAX_OPEN_FILES 参数的设置决定。默认情况下，最多可并行打开的 BFILE 文件个数为 10；最多可打开的文件个数为 50 或为 MAX_OPEN_ FILES 参数的值，这两个值都比较小。



注意：
所有 DBMS_LOB 子程序的完整程序清单请参见 Oracle 文档集。本章将介绍开发人员最常使用的那些函数和过程。

在下面的示例中，Proposal_Text CLOB 列用来说明主要的过程和函数的影响。

1. READ 过程

READ 过程读取 LOB 值的一部分。在 PROPOSAL 表的示例中，第一条记录中的 Proposal_Text 如下：

```
select Proposal_Text
  from PROPOSAL
 where Proposal_ID = 1;

PROPOSAL_TEXT
--------------------------------------------------------------
This is the text of a proposal to maintain an organic garden.
```

因为这是一个相当简短的描述，所以本应该用 VARCHAR2 数据类型列来存储。尽管如此，因为它存储在 CLOB 类型列中，所以其最大长度远远大于它存储在 VARCHAR2 列中的长度。BLOB 和 CLOB 的最大长度由公式(4GB - 1)×数据库块大小来计算，其结果在 8~128TB 之间。为了简单起见，我们在这些示例中就使用这一比较短的串；对于较长的 CLOB 串，同样也可以使用这些函数和操作。

READ 过程有 4 个参数，它们必须按下列顺序指定：

(1) LOB 定位器(用于 BLOB、CLOB 或 BFILE)

(2) 将要读取的字节数或字符数

(3) 相对于 LOB 起始值的偏移量(即读取的开始位置)

(4) 从 READ 过程中输出的数据

如果在读取指定的字节数之前到达了 LOB 值的末尾，则 READ 过程将返回一个错误。

因为 READ 过程需要将 LOB 定位器作为输入，所以 READ 过程在 PL/SQL 块内执行。首先应当选择 LOB 定位器值作为 READ 过程的输入，然后通过 DBMS_OUTPUT 程序包显示读取的文本。

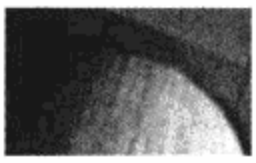
关于 DBMS_OUTPUT 程序包的说明

如第 35 章所述，可以用 DBMS_OUTPUT 程序包显示 PL/SQL 块中的变量值。DBMS_OUTPUT 程序包中的 PUT_LINE 过程将指定的输出内容显示在一行中。在使用 DBMS_OUTPUT 程序包前，应当首先执行 set serveroutput on 命令。下一节将介绍如何创建能从 LOB 中读取数据的 PL/SQL 块。

READ 示例

为了使用 READ 过程，应当知道想要读取的 LOB 的定位器值。此定位器值必须从包含 LOB 的表中选择。因为必须提供定位器值作为 READ 过程的输入，所以应该使用 PL/SQL 变量来保存定位器值。READ 过程将依次把它的输出结果放在一个 PL/SQL 变量中。可以使用 DBMS_ OUTPUT 程序包显示该输出值。该示例中的 PL/SQL 块的结构如下所示：

```
declare
    variable to hold locator value
    variable to hold the amount (the number of characters/bytes to read)
    variable to hold the offset
    variable to hold the output
begin
  set value for amount_var;
  set value for offset_var;
  select locator value into locator var from table;
  DBMS_LOB.READ(locator var, amount var, offset var, output var);
  DBMS_OUTPUT.PUT_LINE('Output:' || output var);
end;
```



注意：

关于 PL/SQL 块及其组成部分的概述，请参见第 32 章。

对于 PROPOSAL 表，从 Proposal_Text 列中选择前 10 个字符的程序清单如下所示：

```
declare
    locator_  var CLOB;
    amount_   var INTEGER;
    offset_   var INTEGER;
    output_   var VARCHAR2(10);
begin
```

```
  amount_     var := 10;
  offset_     var := 1;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  DBMS_LOB.READ(locator_var, amount_var, offset_var, output_var);
  DBMS_OUTPUT.PUT_LINE('Start of proposal text: ' || output_var);
end;
/
```

在执行上面的 PL/SQL 块时，输出结果与前面的 SQL SUBSTR 示例相匹配：

```
Start of proposal text: This is th

PL/SQL procedure successfully completed.
```

此输出结果显示了 Proposal_Text 值的前 10 个字符，从 LOB 值的第一个字符开始。

上例中的 PL/SQL 块首先声明了要使用的变量：

```
declare
   locator_var  CLOB;
   amount_var   INTEGER;
   offset_var   INTEGER;
   output_var   VARCHAR2(10);
```

接下来，给这些变量赋值：

```
begin
  amount_var := 10;
  offset_var := 1;
```

然后从表中选择定位器值，并将其存储在变量 locator_var 中：

```
select Proposal_Text into locator_var
   from PROPOSAL
  where Proposal_ID = 1;
```

接着，使用已经赋值过的变量调用 READ 过程：

```
DBMS_LOB.READ(locator_var, amount_var, offset_var, output_var);
```

作为 READ 过程的执行结果，output_var 变量用读取的数据进行赋值。PUT_LINE 过程显示了该数据：

```
DBMS_OUTPUT.PUT_LINE('Start of proposal text: ' || output_var);
```

PL/SQL 块的这种格式将在本章中的后面部分使用。在声明变量时，可以有选择地设置变量值。

如果想选择 Proposal_Text 列的 CLOB 数据的其他部分，则只需要改变个数和偏移量这两个变量即可。如果要更改读取的字符个数，则还应当更改输出变量的长度。在下例中，从 Proposal_Text 列的第 10 个字符开始，读取 12 个字符：

```
declare
    locator_var    CLOB;
    amount_var     INTEGER;
    offset_var     INTEGER;
    output_var     VARCHAR2(12);
begin
  amount_var := 12;
  offset_var := 10;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  DBMS_LOB.READ(locator_var, amount_var, offset_var, output_var);
  DBMS_OUTPUT.PUT_LINE('Part of proposal text: ' || output_var);
end;
/
```

此 PL/SQL 块的输出结果为：

```
Part of proposal text: he text of a

PL/SQL procedure successfully completed.
```

如果 Proposal_Text 已创建为 VARCHAR2 列，则可以使用 SUBSTR 函数选择该数据。不过，这种数据类型的列的最大长度将被限制为 4 000 个字符。此外，请注意 LOB 列可以包含二进制数据(BLOB 数据类型)，并且可以用从 CLOB 中选择数据的同样方法，使用 READ 过程从 BLOB 中选择数据。对于 BLOB，应当声明使用 RAW 数据类型的输出缓冲区。在 PL/SQL 中，RAW 数据类型和 VARCHAR2 数据类型(用于 CLOB 和 BLOB 读取的输出变量)的最大长度均为 32 767 个字符。

对于 READ 过程来说，可能出现的异常有：VALUE_ERROR、INVALID_ARGVAL、NO_DATA _FOUND、UNOPENED_FILE、NOEXIST_DIRECTORY、NOPRIV_DIRECTORY、INVALID_ DIRECTORY、INVALID_OPERATION 以及 BUFFERING_ENABLED。

2. SUBSTR 函数

DBMS_LOB 程序包中的 SUBSTR 函数执行关于 LOB 值的 SQL SUBSTR 函数。该函数有 3 个输入参数，必须按如下顺序指定：

(1) LOB 定位器

(2) 将要读取的字节数或字符数

(3) 相对于 LOB 起始值的偏移量(读取的起点)

因为 SUBSTR 是一个函数，所以没有像 READ 过程那样的从 SUBSTR 直接返回任何输出变量的值。对于 READ 过程，声明了一个输出变量并在 READ 过程调用中为其赋值。READ 过程使用的 PL/SQL 块如下面的程序清单所示，与输出变量有关的行用粗体标出：

```
declare
    locator_var     CLOB;
    amount_var      INTEGER;
```

```
    offset_var      INTEGER;
    output_var      VARCHAR2(12);
begin
  amount_var := 12;
  offset_var := 10;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  DBMS_LOB.READ(locator_var, amount_var, offset_var, output_var);
  DBMS_OUTPUT.PUT_LINE('Section of proposal text: ' || output_var);
end;
/
```

由于 SUBSTR 是一个函数，因此可以以不同方式对输出变量赋值。格式为：

```
output_var := DBMS_LOB.SUBSTR(locator_var, amount_var, offset_var);
```

前面的 PL/SQL 语句使用 DBMS_LOB 程序包中的 SUBSTR 函数，从 Proposal_Text LOB 列选择了 12 个字符，开始位置为第 10 个字符。完整的 PL/SQL 块如下所示：

```
declare
    locator_var     CLOB;
    amount_var      INTEGER;
    offset_var      INTEGER;
    output_var      VARCHAR2(12);
begin
  amount_var := 12;
  offset_var := 10;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  output_var := DBMS_LOB.SUBSTR(locator_var, amount_var, offset_var);
  DBMS_OUTPUT.PUT_LINE('Section of proposal text: ' || output_var);
end;
/
```

输出结果如下所示：

```
Section of proposal text: he text of a
PL/SQL procedure successfully completed.
```

等价的 SQL 函数调用为

```
select SUBSTR(Proposal_Text,10,12)
  from PROPOSAL
 where Proposal_ID = 1;
```

SUBSTR 函数和 READ 过程只有两点不同，即函数调用名和给输出变量赋值的方式不同。如 READ 过程的示例一样，DBMS_OUTPUT 程序包中的 PUT_LINE 过程用来显示结果。

一般情况下，无论何时选择 LOB 值的某个特定部分时，都应当使用 SUBSTR 函数。虽

然 READ 函数也可以用来选择文本串的某一部分(如上面示例所示)，但通常更多地用于循环中。例如，假设 LOB 值大于 PL/SQL 中允许的 RAW 或 VARCHAR2 变量的最大值(32 767 个字符)。这时，可以在循环中使用 READ 过程从 LOB 值中读取所有数据。关于 PL/SQL 中可用的循环选项的详细信息，请参阅 32 章。

对于 SUBSTR 函数，可能出现的异常有：UNOPENED_FILE、NOEXIST_DIRECTORY、NOPRIV_DIRECTORY、INVALID_DIRECTORY、INVALID_OPERATION 以及 BUFFERING_ENABLED。

3. INSTR 函数

DBMS_LOB 程序包中的 INSTR 函数对 LOB 值执行 SQL INSTR 函数。

INSTR 函数有 4 个输入参数，它们必须按如下顺序指定：

(1) LOB 定位器

(2) 测试模式(对于 BLOB 为 RAW 字节；对于 CLOB 为字符串)

(3) LOB 值相对于起始点的偏移量(即开始读取的位置)

(4) 在 LOB 值中模式出现的次数

DBMS_LOB 程序包中的 INSTR 函数为特定模式的字节或字符搜索 LOB 数据。出现次数变量允许指定在搜索值中应该返回哪一次出现的模式。INSTR 函数的输出结果为搜索的字符串中模式的开始位置。例如，在 SQL 中，

```
INSTR('ABCABC','A',1,1) = 1
```

表示在字符串 ABCABC 中，要搜索的模式为 A，从该串的第 1 个字符开始搜索，查询模式第一次出现的位置。A 在该搜索串的第一个字符位置找到。如果从串的第 2 个字符开始搜索 ，则会得出不同的结果：

```
INSTR('ABCABC','A',2,1) = 4
```

因为此 INSTR 从第二个字符位置开始，所以第一个 A 不属于搜索值的一部分，其实第一次找到的 A 应在第 4 个字符位置。

如果更改 SQL 的 INSTR 函数的最后一个参数，为了查找第二次出现的 A，且从第二个字符位置开始，则找不到此模式：

```
INSTR('ABCABC','A',2,2) = 0
```

如果 SQL 的 INSTR 函数找不到相匹配的模式，则返回 0。DBMS_LOB 程序包中的 INSTR 函数以同样的方式操作。

因为 INSTR 是一个函数，所以应当给它的输出变量赋值，其赋值方式与 SUBSTR 函数的输出变量的赋值方式相同。在下面的程序清单中，在 Proposal_Text 列的 CLOB 值中搜索字符串 “pr”。声明变量 position_var，用来存储 INSTR 函数的输出结果。

```
declare
    locator_var     CLOB;
    pattern_var     VARCHAR2(2);
```

```
    offset_var      INTEGER;
    occur_var       INTEGER;
    position_var    INTEGER;
begin
  pattern_var := 'pr';
  offset_var := 1;
  occur_var := 1;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  position_var := DBMS_LOB.INSTR(locator_var, pattern_var,
offset_var, occur_var);
  DBMS_OUTPUT.PUT_LINE('Found string at position: ' || position_var);
end;
/
```

其输出结果如下：

```
Found string at position: 23

PL/SQL procedure successfully completed.
```

此输出结果表明，在 Proposal_Text 中找到要搜索的字符串“pr”，该字符串从 LOB 值的第 23 个字符开始。因为 Oracle 支持作用于 CLOB 列的 SQL 串函数，所以该函数调用等价于下面的查询：

```
select INSTR(Proposal_Text ,'pr',1,1)
    from PROPOSAL
  where Proposal_ID = 1;
```

对于 INSTR 函数，可能的异常包括：UNOPENED_FILE、NOEXIST_DIRECTORY、NOPRIV_DIRECTORY、INVALID_DIRECTORY、INVALID_OPERATION 以及 BUFFERING_ENABLED。

4. GETLENGTH 函数

DBMS_LOB 程序包中的 GETLENGTH 函数返回 LOB 值的长度。虽然 DBMS _LOB.GETLENGTH 函数与 SQL 的 GETLENGTH 函数在功能上相似，但它只用于 LOB 值。

DBMS_LOB 程序包中的 GETLENGTH 函数仅有一个输入参数，即 LOB 的定位器值。在下面的程序清单中，先声明用来存储定位器值和输出结果的变量。然后执行 GETLENGTH 函数，其结果通过 PUT _LINE 过程报告：

```
declare
    locator_var CLOB;
    length_var INTEGER;
begin
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  length_var := DBMS_LOB.GETLENGTH(locator_var);
```

```
  DBMS_OUTPUT.PUT_LINE('Length of LOB: ' || length_var);
end;
/
```

下面是 GETLENGTH 的输出结果：

```
Length of LOB: 61

PL/SQL procedure successfully completed.
```

如果 LOB 值为 NULL，则 GETLENGTH 函数将返回一个 NULL 值。

与 CLOB 值等价的 SQL 函数为：

```
select LENGTH(Proposal_Text)
  from PROPOSAL
 where Proposal_ID = 1;
```

5. COMPARE 函数

DBMS_LOB 程序包中的 COMPARE 函数比较两个 LOB 值。如果两个 LOB 值相同，则 COMPARE 函数返回值 0；否则将返回一个非零整数值(通常为 1 或 - 1)。为了执行 COMPARE 函数，DBMS_LOB 程序包实际上执行了两个 READ 函数，并比较其结果。因此，对于每个要比较的 LOB 值来说，应当提供一个定位器值和一个偏移量值。对于两个 LOB 值来说，要比较的字符数或字节数相等。只能对相同数据类型的 LOB 进行比较。

COMPARE 函数有 5 个输入参数，它们必须按如下顺序指定：

(1) 第一个 LOB 的 LOB 定位器

(2) 第二个 LOB 的 LOB 定位器

(3) 数量变量(要比较的字节数或字符数)

(4) 相对第一个 LOB 值的偏移量(即开始读取的位置)

(5) 相对第二个 LOB 值的偏移量(即开始读取的位置)

由于比较的两个 LOB 有不同的偏移值，因此可以将一个 LOB 值的第一部分与另一个 LOB 值的第二部分进行比较。在下面的程序清单中，将 Proposal_ID 为 1 和 Proposal_ID 为 3 的这两个记录的 Proposal_Text 值的前 25 个字符进行比较：

```
declare
    first_locator_var      CLOB;
    second_locator_var     CLOB;
    amount_var             INTEGER;
    first_offset_var       INTEGER;
    second_offset_var      INTEGER;
    output_var             INTEGER;
begin
  amount_var := 25;
  first_offset_var := 1;
  second_offset_var := 1;
  select Proposal_Text into first_locator_var
    from PROPOSAL
```

```
   where Proposal_ID = 1;
select Proposal_Text into second_locator_var
  from PROPOSAL
 where Proposal_ID = 3;
output_var :=DBMS_LOB.COMPARE(first_locator_var, second_locator_var,
       amount_var, first_offset_var, second_offset_var);
DBMS_OUTPUT.PUT_LINE('Comparison value (0 if the same): '||
 output_var);
end;
/
```

下面是输出结果：

```
Comparison value (0 if the same): 1

PL/SQL procedure successfully completed.
```

如果两个字符串相同，COMPARE 函数将返回 0。

在下面的程序清单中，比较两个相同的 LOB，但这次只比较前 5 个字符。因为这两个字符串都以“This” 开始，所以 COMPARE 返回 0：

```
declare
    first_locator_var     CLOB;
    second_locator_var    CLOB;
    amount_var            INTEGER;
    first_offset_var      INTEGER;
    second_offset_var     INTEGER;
    output_var            INTEGER;
begin
  amount_var := 5;
  first_offset_var := 1;
  second_offset_var := 1;
  select Proposal_Text into first_locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  select Proposal_Text into second_locator_var
    from PROPOSAL
   where Proposal_ID = 3;
  output_var :=DBMS_LOB.COMPARE(first_locator_var, second_locator_var,
          amount_var, first_offset_var, second_offset_var);
  DBMS_OUTPUT.PUT_LINE('Comparison value (0 if the same): ' ||
   output_var);
end;
/

Comparison value (0 if the same): 0

PL/SQL procedure successfully completed.
```

COMPARE 函数既可以用于字符串之间的比较，也可用于二进制数之间的比较。如果在

BLOB 列上使用 COMPARE 函数，则应定义定位器变量为 RAW 数据类型。

对于 COMPARE 函数，可能的异常包括：UNOPENED_FILE、NOEXIST_DIRECTORY、NOPRIV_DIRECTORY、INVALID_DIRECTORY、INVALID_OPERATION 和 BUFFERING_ENABLED。

6. WRITE 过程

DBMS_LOB 程序包中的 WRITE 过程允许将数据写入 LOB 的特定位置。例如，可以将二进制数据写入 BLOB 的某个部分，并重写该位置已有的数据。使用 WRITE 过程，可以将字符数据写入 CLOB 字段。此过程有 4 个输入参数，必须按以下顺序指定：

(1) LOB 的 LOB 定位器

(2) 数量变量(要写入的字节数或字符数)

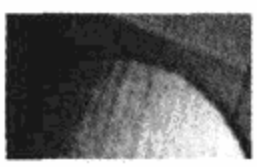
(3) 相对于 LOB 起始值的偏移量(即开始写入的位置)

(4) 分配给要添加的字符串或二进制数据的缓冲区变量

因为 WRITE 过程更新 LOB 值，所以应当首先通过 select for update 锁定该行，如下面程序清单中的粗体部分所示。在此示例中，选择并锁定了一条记录。文本 ADD NEW TEXT 接着被写入 LOB 值，从位置 10(由偏移变量定义)开始重写数据。

```
declare
      locator_var     CLOB;
      amount_var      INTEGER;
      offset_var      INTEGER;
      buffer_var      VARCHAR2(12);
begin
  amount_var := 12;
  offset_var := 10;
  buffer_var := 'ADD NEW TEXT';
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 3
     for update;
  DBMS_LOB.WRITE(locator_var, amount_var, offset_var, buffer_var);
commit;
end;
/
```

因为 WRITE 过程更改数据，所以在 PL/SQL 块的 end 子句之前，应添加 commit 命令。



注意：

先不使用 commit 进行测试，就可以看到在向数据库提交变化之前过程调用的影响。

对于 WRITE 过程，没有任何输出。为查看更改后的数据，应当再一次调用 READ 过程(在同一个 PL/SQL 块中)或从表中选择 LOB 值：

```
select Proposal_Text
  from PROPOSAL
 where Proposal_ID = 3;

PROPOSAL_TEXT
------------------------------------------------------------
This is tADD NEW TEXTsal text.
```

WRITE 过程重写了 12 个字符，从 Proposal_Text 列的第 10 个字符开始。可以使用过程 WRITEAPPEND 在已有的 LOB 值的尾部添加新数据。

对于 CLOB 数据，可以使用 update 语句，结合 SUBSTR 和连接函数重写这个示例。

对于 WRITE 函数，可能的异常包括：VALUE_ERROR、INVALID_ARGVAL、QUERY_WRITE、BUFFERING_ENABLED 以及 SECUREFILE_OUTOFBOUNDS。

7. APPEND 过程

DBMS_LOB 程序包中的 APPEND 过程可从一个 LOB 向另一个 LOB 中添加数据。由于这相当于对 LOB 值进行更新，因此在执行 APPEND 过程前应锁定要更新的记录。

APPEND 过程接受两个参数，即目的 LOB 的定位器值和源 LOB 的定位器值。如果其中一个参数为 NULL，则 APPEND 过程返回错误消息。

在下例中，定义了两个定位器变量。第一个定位器变量命名为 dest_locator_var，这是要添加数据的 LOB 的定位器值。第二个定位器变量为 source_locator_var，是要添加到目的 LOB 的源数据的定位器值。由于目的 LOB 将被更新，因此需用 select for update 命令锁定它。在此示例中，将 Proposal_ID 等于 1 的记录的 Proposal_Text 的值追加到 Proposal_ID 等于 3 的记录的 Proposal_Text 的值后面：

```
declare
  dest_locator_var    CLOB;
  source_locator_var  CLOB;
begin
  select Proposal_Text into dest_locator_var
    from PROPOSAL
   where Proposal_ID = 3
     for update;
  select Proposal_Text into source_locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  DBMS_LOB.APPEND(dest_locator_var, source_locator_var);
commit;
end;
/
```

要查看更改后的数据，可以从 PRPOSAL 表中选择记录：

```
select Proposal_Text
  from PROPOSAL
 where Proposal_ID = 3;
```

```
PROPOSAL_TEXT
--------------------------------------------------------------
This is tADD NEW TEXTsal text.This is the text of a proposal to
maintain an orga
```

剩下的文本发生了什么变化呢？默认情况下，在查询过程中，LONG 值和 LOB 值只显示 80 个字符。要查看剩下的文本，应当使用 set long 命令：

```
set long 1000

select Proposal_Text
  from PROPOSAL
 where Proposal_ID = 3;

PROPOSAL_TEXT
--------------------------------------------------------------
This is tADD NEW TEXTsal text.This is the text of a proposal to
maintain an organic garden.
```

此示例显示了 LOB 值的前 1 000 个字符。如果想了解 LOB 的长度，则可以使用本章前面介绍的 GETLENGTH 函数。

APPEND 过程不仅允许添加 CLOB 值，还允许添加 BLOB 值。在添加 BLOB 数据时，应当了解该 BLOB 数据的格式和在该 BLOB 值末尾添加新数据所产生的影响。对于 BLOB 值，APPEND 函数会非常有用，特别是对于 BLOB 中仅包含原始数据(如数字数据传输)而非格式化的二进制数据(如通过程序格式化的电子表格文件)的应用程序，更是如此。

对于 CLOB 数据，可以使用 update 语句重写该示例，设置 Proposal_Text 的值等于多个值的连接。

对于 APPEND 过程，定义的异常有 VALUE_ERROR、QUERY_WRITE 和 BUFFERING_ENABLED。

8. ERASE 过程

使用 DBMS_LOB 程序包中的 ERASE 过程可以擦除 LOB 中任何部分的字符或字节。也可以用 ERASE 擦除整个 LOB 值，或擦除指定的 LOB 部分。ERASE 在功能上与 WRITE 函数相似。如果擦除 BLOB 值中的数据，则该数据将用 0 字节填充器代替。如果擦除 CLOB 值中的数据，则用空格填充。

因为 ERASE 更新 LOB 值，所以应当遵循 LOB 更新数据的标准过程，即先锁定相应的行，并在完成此过程后擦除提交。

在下面的 PL/SQL 块中，将擦除 Proposal_ID 等于 3 的记录的 Proposal_Text 值的部分内容。擦除的字符从第 10 个字符开始，并连续擦除 17 个字符。ERASE 过程指定参数的顺序如下：

(1) LOB 使用的 LOB 定位器

(2) 数量变量(要擦除的字节数或字符数)

(3) 相对于 LOB 起始值的偏移量(即开始擦除的位置)

```
declare
    locator_var    CLOB;
    amount_var     INTEGER;
    offset_var     INTEGER;
begin
  amount_var := 17;
  offset_var := 10;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 3
    for update;
  DBMS_LOB.ERASE(locator_var, amount_var, offset_var);
commit;
end;
/
```

(4) 通过查询 Proposal_Text 值，可以查看记录的变化：

```
select Proposal_Text
  from PROPOSAL
 where Proposal_ID = 3;

PROPOSAL_TEXT
--------------------------------------------------------------------
This is t               ext.This is the text of a proposal to
maintain an organic garden.
```

注意：
擦除数据后留下的空间由空格填充，LOB 中余下的数据位置不改变。

对于 CLOB 数据，可以使用 update 语句重写该操作，通过连接 SUBSTR 的结果和空格串，设置 Proposal_Text 的一个新值。

对于 ERASE 过程，可能的异常包括：VALUE_ERROR、INVALID_ARGVAL、QUERY_WRITE 和 BUFFERING_ENABLED。

9. TRIM 过程

使用 DBMS_LOB 程序包中的 TRIM 过程，通过去掉 LOB 值末尾的字符或字节(类似于 SQL 的 RTRIM 函数)，可以减少 LOB 值的长度。在执行 TRIM 过程时，需要指定 LOB 的定位器值和该 LOB 后来的长度。

在下面的程序清单中，把 Proposal_ID 值为 3 的记录的 Proposal_Text 值截断到前 10 个字符。 由于 TRIM 过程更新数据，因此在 PL/SQL 块的 end 子句前应增加 commit 命令。

```
declare
    locator_var      CLOB;
    new_length_var   INTEGER;
begin
```

```
  new_length_var := 10;
  select Proposal_Text into locator_var
    from PROPOSAL
   where Proposal_ID = 3
     for update;
  DBMS_LOB.TRIM(locator_var, new_length_var);
commit;
end;
/
```

对于 TRIM 过程，没有任何输出结果。要查看更改后的数据，应当再次从表中选择 LOB 值：

```
select Proposal_Text
    from PROPOSAL
  where Proposal_ID = 3;

PROPOSAL_TEXT
--------------------------------------------------------------------
This is t
```

此样例的输出表明，TRIM 过程的结果与前一节执行的 ERASE 过程类似。对于 CLOB 数据，可以重写为一个简单的 update：

```
update PROPOSAL
    set Proposal_Text = SUBSTR(Proposal_Text,1,10)
  where Proposal_ID = 3;
```

对于 TRIM 过程，可能的异常包括：VALUE_ERROR、INVALID_ARGVAL、QUERY_WRITE 和 BUFFERING_ENABLED。

10. COPY 过程

可以使用 DBMS_LOB 程序包中的 COPY 过程将一个 LOB 的一部分数据复制到另一个 LOB 中。与 APPEND 过程不同，不必将一个 LOB 值的全部文本复制到另一个 LOB 值中。在 COPY 过程中，可以指定读取数据的偏移量和写入数据的偏移量。COPY 过程有 5 个参数，依次为：

(1) 目的 LOB 定位器
(2) 源 LOB 定位器
(3) 数量(要复制的字符或字节数)
(4) 目的 LOB 值中开始写入点的偏移量
(5) 源 LOB 值中开始读取点的偏移量

COPY 过程同时具有 READ 功能和 WRITE 功能。与 WRITE 操作相同，COPY 过程也更改 LOB 值。因此，应先锁定记录，并且 PL/SQL 块必须包括 commit 命令。在下例中，Proposal_ID 等于 1 的记录中的前 61 个字符被复制到了 Proposal_ID 等于 3 的记录中。由于目的 LOB 使用的偏移量为 1，因此被复制的数据将重写目的 LOB 内的数据。

```
declare
    dest_locator_var     CLOB;
    source_locator_var   CLOB;
    amount_var           INTEGER;
    dest_offset_var      INTEGER;
    source_offset_var    INTEGER;
begin
  amount_var := 61;
  dest_offset_var := 1;
  source_offset_var := 1;
  select Proposal_Text into dest_locator_var
    from PROPOSAL
   where Proposal_ID = 3
    for update;
  select Proposal_Text into source_locator_var
    from PROPOSAL
   where Proposal_ID = 1;
  DBMS_LOB.COPY(dest_locator_var, source_locator_var,
      amount_var, dest_offset_var, source_offset_var);
commit;
end;
/
```

COPY 过程不显示任何输出结果。要查看修改过的数据，应当再次从表中选择该 LOB 值：

```
select Proposal_Text
    from PROPOSAL
  where Proposal_ID = 3;

PROPOSAL_TEXT
-------------------------------------------------------------
This is the text of a proposal to maintain an organic garden.
```

对于 COPY 过程，可能的异常包括：VALUE_ERROR、INVALID_ARGVAL、QUERY_WRITE 和 BUFFERING_ENABLED。

11. 使用 BFILE 函数和过程

由于 BFILE 值存储在外部，因此在对 BFILE LOB 执行 READ、SUBSTR、INSTR、GETLENGTH、COMPARE 操作之前，需要使用几个过程和函数来操作这些文件。DBMS_LOB 中与 BFILE 相关的过程和函数(参见前面的表 40-2)可以打开文件、关闭文件、获取文件名、检查文件是否存在并测试文件是否打开。存在性检测函数十分必要，因为 Oracle 不维护存储在外部文件中的数据，Oracle 只维护在插入或更新记录时通过 BFILENAME 过程创建的指针。应当使用表 40-2 中的过程和函数来控制 PL/SQL 代码中使用的文件。例如，要从 Proposal_ID 等于 2 的记录的 Cover_Letter 列的 BFILE 值中读取前 10 个字节，需要先打开文件，然后执行 READ 过程，再调用 DBMS_OUTPUT 程序包中的 PUT_LINE 过程显示读取的数据。

有关这些访问方法的更多代码示例和描述，请参见 *Oracle Database SecureFiles and Large*

Objects Developer's Guide。

```
/* Checking if a pattern exists in a BFILE using instr
/* Procedure compareBFILEs_proc is not part of DBMS_LOB package: */

CREATE OR REPLACE PROCEDURE compareBFILEs_proc IS
   /* Initialize the BFILE locator: */
   file_loc1     BFILE := BFILENAME('MEDIA_DIR', 'keyboard.jpg');
   file_loc2     BFILE;
   Retval         INTEGER;
BEGIN
   DBMS_OUTPUT.PUT_LINE('------------ LOB COMPARE EXAMPLE ------------');
   /* Select the LOB: */
   SELECT ad_graphic INTO File_loc2 FROM print_media
      WHERE Product_ID = 3060 AND ad_id = 11001;
   /* Open the BFILEs: */
   DBMS_LOB.OPEN(File_loc1, DBMS_LOB.LOB_READONLY);
   DBMS_LOB.OPEN(File_loc2, DBMS_LOB.LOB_READONLY);
Retval := DBMS_LOB.COMPARE(File_loc2, File_loc1, DBMS_LOB.LOBMAXSIZE, 1, 1);
   /* Close the BFILEs: */
   DBMS_LOB.CLOSE(File_loc1);
   DBMS_LOB.CLOSE(File_loc2);
END;
/
```

该示例显示的方法和前面的 COMPARE 示例很相似。由于操作的是 BFILE，因此它还有另外两部分，即用来打开文件的 DBMS_LOB.OPEN 调用和当 COMPARE 结束时用来关闭文件的 DBMS_LOB.CLOSE 调用。

从 BFILE 中读取数据需要遵循前面讲到的 READ 函数的操作步骤，另外还要其他调用来打开和关闭文件。READ 过程通过 Oracle 在/rdbms/demo/logs/plsql 子目录中提供的 fread.sql 演示文件来说明：

```
/* This file is installed in the following path when you install */
/* the database: $ORACLE_HOME/rdbms/demo/lobs/plsql/fread.sql */
/* Reading data from a BFILE. */
/* Procedure readBFILE_proc is not part of DBMS_LOB package: */
CREATE OR REPLACE PROCEDURE readBFILE_proc IS
   file_loc       BFILE;
   Amount         INTEGER := 32767;
   Position       INTEGER := 1;
   Buffer         RAW(32767);
BEGIN
   DBMS_OUTPUT.PUT_LINE('------------ BFILE READ EXAMPLE ------------');
   /* Select the LOB: */
   SELECT ad_graphic INTO File_loc FROM print_media
      WHERE product_id = 3060 AND ad_id = 11001;
   /* Open the BFILE: */
   DBMS_LOB.OPEN(File_loc, DBMS_LOB.LOB_READONLY);
   /* Read data: */
```

```
    DBMS_LOB.READ(File_loc, Amount, Position, Buffer);
    /* Close the BFILE: */
    DBMS_LOB.CLOSE(File_loc);
END;
/
```

如果在 PL/SQL 块中使用了 Exception Handling 部分，则必须要确保在声明的异常中包含 FILECLOSE 过程调用(关于 PL/SQL 块中异常处理的详细介绍，请参阅第 32 章)。

为了将数据从外部文件移到 BLOB 数据类型中，可以在表中创建 1 个 BFILE 数据类型，并使用 BFILENAME 函数创建 1 个指向外部文件的 LOB 定位器。BFILENAME 函数接受两个参数：目录名和文件名。如下例所示：

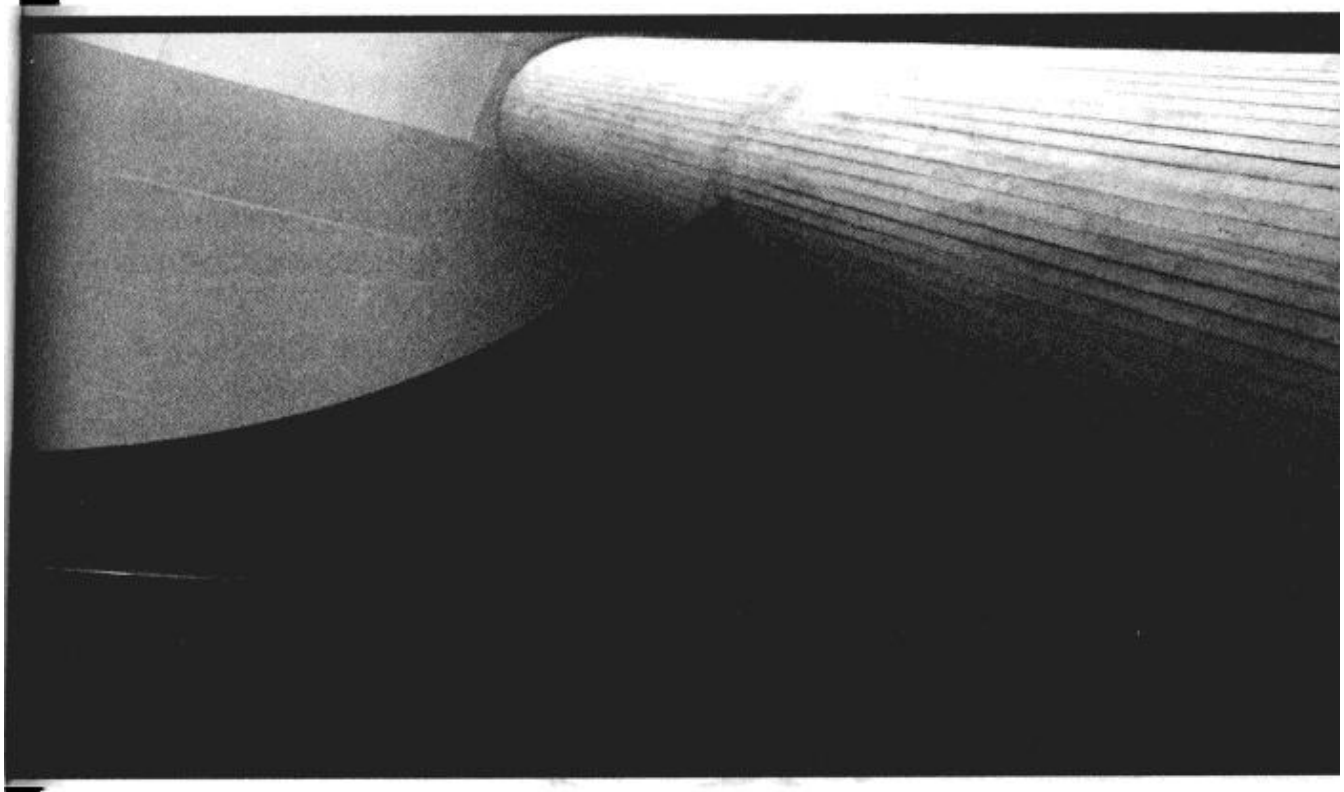
```
insert into PROPOSAL (Proposal_ID, Cover_Letter)
values (4, BFILENAME ('proposal_dir','extfile.doc'));
```

然后，可以打开文件并使用 LOADFROMFILE、LOADBLOBFROMFILE 或 LOADCLOBFROMFILE 过程将外部文件中的数据加载到 BLOB 数据类型中。

如果当前的数据为 LONG 数据类型，则可以使用 alter table 命令把它更改为 CLOB 或 NCLOB 列。还可以使用 alter table 将 LONG RAW 列更改为 BLOB 数据类型的列。对于 insert as select 操作，可以使用 TO_LOB 这个 SQL 函数将数据从 LONG 列移动到 BLOB 列。

40.3.6 删除 LOB

删除 LOB 值时，定位器值也一并删除。如果所删除的值是内部 LOB 数据(BLOB、CLOB 或 NCLOB)，则定位器和 LOB 值均被删除；如果所删除的值是外部 LOB(BFILE)，则仅删除定位器值；需要手工删除该定位器所指向的外部文件。



第41章

面向对象的高级概念

到此为止，本书中 Oracle 所有的面向对象程序设计(OOP)特性均包含两个特征，即它们是嵌入对象和列对象。嵌入对象(embedded object)是完全包含在另一对象中的对象。例如，因为嵌套表包含在另一个表中，所以它是嵌入对象。尽管嵌套表的数据与主表是分开存储的，但其数据只能通过主表访问。列对象(column object)是以表中的列表示的对象。例如，因为可变数组用表中的一列表示，所以它是一个列对象。

为了充分利用 OOP 的功能，数据库还必须支持行对象(row object)，即用行而不是列表示的对象。行对象不是嵌入对象，而是引用对象(referenced object)，它可以通过引用其他对象来访问。本章将介绍如何创建和引用行对象。

面向对象的结构可能在许多应用程序中都不适合——可能会受到工具包、接口、或数据库中已有的关系的限制。本章将介绍 Oracle 如何实现行对象和如何利用对象视图将面向对象和关系体系结构联系起来。除了行对象以外，本章还将介绍 PL/SQL 的对象应用，即创建对

象 PL/SQL。

41.1 行对象和列对象

行对象和列对象之间的区别十分重要。首先，考虑列对象。列对象基于数据库中已经存在的特性进行扩展。对象类型或可变数组可作为表中的一列嵌入表中。嵌套表或大对象(LOB)可以存储在表的一列中。在每种情况下，对象都被表示为表中的一列。

但是，嵌套表和 LOB 涉及与主表分开存储的外部数据。在使用 LOB 时(请参阅 40 章)，指定将要存储 LOB 数据的表的存储值，并且如果 LOB 数据是分开存储的外部数据，则 Oracle 将创建 LOB 定位器，它从主表指向 LOB 表中的行。在创建嵌套表时，指定将存储嵌套表的记录的表名，同时 Oracle 创建从主表指向嵌套表中记录的指针。因此，主表与其嵌入的嵌套表相关联。

在行对象中，对象表示为行而不是列。数据并不嵌入到主表中，相反，主表包含对另一个表的引用。由于嵌套表与主表分开存储，因此它有助于理解行对象。如果嵌套表中的数据未被嵌入主表中并且每一行都是行对象时，该怎么办呢？此时，主表将定义对相关数据的引用。

前几章介绍的 Oracle 的对象关系特性依赖于列对象。本章将集中介绍高级的 OOP 功能，它们基于行对象。使用行对象可以创建不同表的数据行之间的引用。

41.2 对象表和 OID

在一个对象表(object table)中，每一行都是一个行对象。对象表与普通的关系表有两点不同。首先，对象表中的每一行均有一个 OID(即对象标识符值)，它是创建行时 Oracle 分配的。其次，对象表中的行可由该数据库中的其他对象引用。

可以通过 create table 命令创建一个对象表。考虑前几章介绍过的 ANIMAL_TY 数据类型。为避免与前几章创建的对象发生冲突，在一个新的模式中创建该类型。

```
create or replace type ANIMAL_TY as object
 (Breed      VARCHAR2(25),
  Name       VARCHAR2(25),
  BirthDate  DATE);
/
```

注意：
为简单起见，创建 ANIMAL_TY 数据类型时，没有使用任何成员函数。

为创建 ANIMAL_TY 数据类型的对象表，可以执行以下的 create table 命令：

```
create table ANIMAL of ANIMAL_TY;
```

```
Table created.
```

请注意，该命令有一个特殊的语法，即创建了一个对象类型的表。该表初看起来像一个普通的关系表：

```
describe ANIMAL

Name                           Null?    Type
------------------------------ -------- ------------
BREED                                   VARCHAR2(25)
NAME                                    VARCHAR2(25)
BIRTHDATE                               DATE
```

ANIMAL 表的列映射到 ANIMAL_TY 对象类型的属性上。然而，由于它是一个对象表，因此关于如何使用该表有明显的区别。如本节前面所述，对象表中的每一行均有一个 OID 值，而且这些行可以作为对象被引用。

41.2.1　把行插入对象表

因为 ANIMAL 是一个对象表，所以在向表中插入数据时，可以使用其对象类型的构造函数方法。由于 ANIMAL 表基于 ANIMAL_TY 对象类型，因此在向 ANIMAL 表中插入值时，可以使用 ANIMAL_TY 构造函数方法。

注意：
关于构造函数方法的详细讨论，请参阅第 38 章。

在下面的程序清单中，把 3 行数据插入 ANIMAL 对象表中。每个命令均调用了 ANIMAL_TY 构造函数方法。

```
insert into ANIMAL values
       (ANIMAL_TY('MULE','FRANCES', '01-APR-02'));
insert into ANIMAL values
       (ANIMAL_TY('DOG','BENJI', '03-SEP-01'));
insert into ANIMAL values
       (ANIMAL_TY('CROCODILE','LYLE', '14-MAY-00'));
```

如果对象表所基于的对象类型本身又基于另一个对象类型，则应该在 insert 命令中嵌套对多个构造函数方法的调用(关于嵌套对象类型的示例，请参阅第 39 章)。

注意：
如果对象表基于不包含任何嵌套数据类型的数据类型，则也可使用用于关系表的普通 insert 命令语法将记录插入该对象表中。

向对象表中插入一行时，Oracle 给该行分配了一个 OID 值。有了 OID 值才可以将该行作为一个可引用的对象来使用。不能复用 OID 值。

41.2.2 从对象表中选择值

在创建 ANIMAL 表时，它完全基于 ANIMAL_TY 数据类型：

```
create table ANIMAL of ANIMAL_TY;
```

在从包含对象类型的表中进行选择值时，可以像引用表的列一样引用对象类型的列。也就是说，如果使用了 ANIMAL_TY 数据类型作为 Animal 列的基础，则通过从该表中选择 Animal.Name 可以选择动物名称。

不过，对象表是基于对象类型的——它没有其他的列。因此，在访问列时，不需要引用该对象类型。要从 ANIMAL 表中选取名字，只要直接查询属性即可。

```
select Name
  from ANIMAL;

NAME
-------------------------
FRANCES
BENJI
LYLE
```

可以在 where 子句中引用列，就像 ANIMAL 表是关系表一样。

```
select Name
from ANIMAL
where Breed = 'CROCODILE';
NAME
-------------------------
LYLE
```

如果 ANIMAL_TY 对象类型为其中一列使用了另一个对象类型，则该数据类型的列将在所有的 select、update 和 delete 操作中引用。

41.2.3 从对象表中更新和删除数据

如果对象表基于一个不使用其他对象类型的对象类型，那么对象表的更新或删除命令可以使用与关系表一样的格式。因为 ANIMAL_TY 数据类型不对其任何一列使用其他的对象类型，所以 ANIMAL 对象表中的 update 和 delete 操作与 ANIMAL 为关系表时的操作相同。

以下命令更新 ANIMAL 表中的一行：

```
update ANIMAL
   set BirthDate = '01-MAY-00'
 where Name = 'LYLE';

1 row updated.
```

注意，在更新过程中引用了该对象表的列，就像 ANIMAL 表是关系表一样。在删除过程中，可以使用相同的方法在 where 子句中引用该对象表的列：

```
delete from ANIMAL
  where Name = 'LYLE';

1 row deleted.
```

将鳄鱼与骡子和狗放在一起，无论如何都不是一个好主意。

41.2.4　REF 函数

REF 函数可以引用已有的行对象。例如，ANIMAL 表(已删除一行，如 41.2.3 节所示)有两个行对象，每个行对象都分配了一个 OID 值。通过使用 REF 函数，可以看到分配给每行的 OID 值，如下所示：

```
select REF(A)
  from ANIMAL A
 where Name = 'FRANCES';

REF(A)
--------------------------------------------------
0000280209F525172CC0F84F8ABABF0F0F46ECB7209A233BD5F
FD34A7E82D8C0C5CFB88FC0010002270000
```

注意：
用户的 REF(A)值可能与本例给出的不同。该 REF(A)值在程序清单中占据了多行。

在前面的程序清单中，ANIMAL 表具有别名 A，并且此别名可以作为 REF 函数的输入。输出结果为满足查询限定条件的行对象的 OID 值。

注意：
REF 函数以对象表的别名作为输入。由于本章后面给出的其他函数也以别名作为输入，因此应当熟悉这种语法格式。

由于 REF 函数只能引用行对象，因此不能用它来引用列对象。列对象包括对象类型、LOB 和收集器。

正如所见，REF 函数本身不提供任何有用的信息。应当使用另一个函数 DEREF 将 REF 的输出转换为值，然后才可以使用 REF 和 DEREF 来引用行对象的值，如 41.2.5 节所述。

41.2.5　使用 DEREF 函数

REF 函数以行对象作为其参数，并返回一个引用值。而 DEREF 函数正好相反，它接受一个引用值(即为引用而生成的 OID 值)，并返回该行对象的值。

在本章前面，ANIMAL 对象表是用 ANIMAL_TY 对象类型创建的：

```
create table ANIMAL of ANIMAL_TY;
```

为了解 REF 和 DEREF 如何协同工作，可以考虑一个关联到 ANIMAL 表的表。KEEPER 表将记录那些照顾动物的管理员。它有一个关于管理员姓名的列(KeeperName)和对 ANIMAL 对象表(AnimalKept)的一个引用，如下面的程序清单所示：

```
create table KEEPER
(KeeperName VARCHAR2(25),
 AnimalKept REF ANIMAL_TY);
```

虽然该 create table 命令的第一部分看起来很普通：

```
create table KEEPER
 (KeeperName     VARCHAR2(25),
```

但最后一行显示了一个新特性：

```
AnimalKept      REF ANIMAL_TY);
```

AnimalKept 列引用存储在其他地方的数据。REF 函数将 AnimalKept 列指向 ANIMAL_TY 数据类型的行对象。由于 ANIMAL 是 ANIMAL_TY 数据类型的对象表，因此 AnimalKept 列可以指向 ANIMAL 对象表中的行对象。可以将 REF 指向该类型的任何对象表。

在描述 KEEPER 表时，可以看到它的 AnimalKept 列依赖于 REF：

```
describe KEEPER

Name                           Null?     Type
------------------------------ --------  ----
KEEPERNAME                               VARCHAR2(25)
ANIMALKEPT                               REF OF ANIMAL_TY
```

为了查看全部细节，使用一个更高的 describe 深度值：

```
set describe depth 2

desc KEEPER

Name                                      Null?     Type
----------------------------------------- --------  ----------------
KEEPERNAME                                          VARCHAR2(25)
ANIMALKEPT                                          REF OF ANIMAL_TY
BREED                                               VARCHAR2(25)
NAME                                                VARCHAR2(25)
BIRTHDATE                                           DATE
```

现在，向 KEEPER 表中插入一条记录。需要使用 REF 函数将 ANIMAL 引用存储在 AnimalKept 列中。

```
insert into KEEPER
select 'CATHERINE WEILZ',
       REF(A)
```

```
  from ANIMAL A
where Name = 'BENJI';
```

下面仔细看看这条 insert 命令。首先，KeeperName 值是作为字面值从 ANIMAL 表中选择的：

```
insert into KEEPER
select 'CATHERINE WEILZ',
```

接着，必须指定 AnimalKept 列的值。但因为 AnimalKept 列的数据类型为 REF OF ANIMAL_TY，所以必须从 ANIMAL 对象表中选择该引用，以插入 AnimalKept 列的值：

```
   REF(A)
  from ANIMAL A
where Name = 'BENJI';
```

接着会发生什么？首先，查询 ANIMAL 对象表，REF 函数返回选中行对象的 OID。然后，该 OID 作为指向 ANIMAL 对象表中行对象的指针，而存储在 KEEPER 表中。

KEEPER 表包含动物的信息吗？不包含，它包含管理员的名字和对 ANIMAL 表中的一个行对象的引用。通过查询 KEEPER，可以看到 OID 引用：

```
select * from KEEPER;

KEEPERNAME
-------------------------
ANIMALKEPT
--------------------------------------------------
CATHERINE WEILZ
00002202088817DC34109A4A929C632C0FAF1F78EF9A233BD5F
FD34A7E82D8C0C5CFB88FC0
```

注意：

AnimalKept 列的值和文本的换行可能与本例显示的结果不同。

该程序清单中的 AnimalKept 列包含行对象的引用，而不是存储在该行对象中的数据值。

除非使用 DEREF 函数，否则无法看到引用值。在从 KEEPER 表中选择时，Oracle 将使用该 OID 评估该引用(REF)。DEREF 函数将接受该引用，并返回一个值。

在下面的查询中，KEEPER 表中的 AnimalKept 列的值是传递给 DEREF 函数的参数。DEREF 函数将从 AnimalKept 列中获取 OID，并查找引用的对象，此函数将评估该引用，并将值返回给用户。

```
select DEREF(K.AnimalKept)
  from KEEPER K
 where KeeperName = 'CATHERINE WEILZ';
```

```
DEREF(K.ANIMALKEPT)(BREED, NAME, BIRTHDATE)
-------------------------------------------------
ANIMAL_TY('DOG', 'BENJI', '03-SEP-01')
```

注意：
DEREF 函数的参数是 REF 列的列名，而非表名。

结果表明，KEEPER 表中的 CATHERINE WEILZ 记录引用了 ANIMAL 表中名为 BENJI 的动物记录。该查询有几个方面值得注意：

- 查询使用了从一个表(KEEPER)到另一个表(ANIMAL 对象表)的行对象引用。因此，在后台执行了一个连接，而无须指定连接条件。
- 该查询中没有提及对象表本身。在该查询中列出的唯一表是 KEEPER 表。对对象表的值进行 DEREF 操作，一般无须知道对象表的名称。
- 返回所有引用的行对象，而不只是该行的一部分。

这是对象查询与关系查询的重要区别。因此，在查询表时，应当知道创建关系的方式。关系是基于外键和主键，还是基于对象表和 REF 数据类型？如果要使用 Oracle 的面向对象功能，就必须能够适应 SQL 的变化和创建关系的方式。为了帮助完成关系方法和面向对象方法之间的平滑转换，Oracle 允许创建包含 REF 的对象视图，该 REF 是在已有的关系表上添加的。关于这方面的内容，请参阅 41.3 节。

41.2.6 VALUE 函数

DEREF 函数应用于关系表中，本例中为 KEEPER 表。DEREF 函数返回从关系表到对象表的引用值。

那么，从对象表中进行查询会怎样呢？能从 ANIMAL 表中选择吗？

```
select * from ANIMAL;

BREED                    NAME                     BIRTHDATE
------------------------ ------------------------ ---------
MULE                     FRANCES                  01-APR-02
DOG                      BENJI                    03-SEP-01
```

尽管 ANIMAL 表一个对象表，但是仍可以像关系表那样从中选择，这与本章前面介绍的插入和选择操作的示例一致。然而，这不是通过 DEREF 函数显示的内容。DEREF 函数显示了 ANIMAL 对象表使用的对象类型的完整结构。

```
select DEREF(K.AnimalKept)
  from KEEPER K
 where KeeperName = 'CATHERINE WEILZ';

DEREF(K.ANIMALKEPT)(BREED, NAME, BIRTHDATE)
-------------------------------------------------
ANIMAL_TY('DOG', 'BENJI', '03-SEP-01')
```

为了从 ANIMAL 对象表的查询中查看相同的结构，可使用 VALUE 函数。如下面的程序清单所示，VALUE 函数采用 DEREF 函数将使用的相同格式显示数据。VALUE 函数的参数是表的别名。

```
select VALUE(A)
  from ANIMAL A
 where Name = 'BENJI';

VALUE(A)(BREED, NAME, BIRTHDATE)
----------------------------------------
ANIMAL_TY('DOG', 'BENJI', '03-SEP-01')
```

如 41.4 节将要介绍的，在调试引用时，以及在 PL/SQL 中，VALUE 函数都十分有用。由于它可以从对象表中直接查询格式化的值，因此可以选择那些值，而不必使用 KEEPER 表的 AnimalKept 列的 DEREF 查询。

41.2.7　无效引用

可以删除某个引用指向的对象。例如，可以从由 KEEPER 记录指向的 ANIMAL 对象表中删除一行。

```
delete from ANIMAL
  where Name = 'BENJI';
```

引用该 ANIMAL 记录的 KEEPER 表中的记录现在将具有一个悬挂(dangling)的 REF。如果为名为 BENJI 的动物插入一个新 ANIMAL 行，那么该行并不能视为先前已经创建的相同引用的一部分。因为在第一次插入 BENJI 行时，Oracle 便为该行对象生成了一个 OID，就是 KEEPER 列引用的内容。当删除该行对象时，OID 值消失了，即 Oracle 不重用 OID 号。因此，当输入新的 BENJI 记录时，就得到一个新的 OID 值，但 KEEPER 记录仍指向旧的 OID 值。

这是关系系统和 OOP 系统之间的关键区别。在关系系统中，两表之间的连接仅取决于当前数据。但是在 OOP 系统中，由于连接存在于对象之间，因此有相同数据的两个对象并不意味着它们相同。

41.3　具有 REF 的对象视图

使用对象视图可以在已有的关系表(请参阅 38 章)上添加 OOP 结构。例如，可以创建对象类型，并在已有表的对象视图中使用它们。使用对象视图允许通过关系命令语法或对象类型语法来访问表。结果，对象视图是已有的关系应用程序和对象关系应用程序之间沟通的一个重要技术桥梁。

41.3.1　对象视图的简要回顾

第 38 章的这个示例将作为本章中高级对象视图的基础部分。首先，创建 CUSTOMER 表，以 Customer_ID 列作为其主键：

```
create table CUSTOMER
(Customer_ID NUMBER constraint CUSTOMER_PK primary key,
 Name        VARCHAR2(25),
 Street      VARCHAR2(50),
 City        VARCHAR2(25),
 State       CHAR(2),
 Zip         NUMBER);
```

接着，创建两个对象类型。第 1 个为 ADDRESS_TY，包括地址属性 Street、City、State 和 Zip。第 2 个对象类型为 PERSON_TY，包括一个 Name 属性和使用 ADDRESS_TY 数据类型的 Address 属性，如下面的程序清单所示：

```
create or replace type ADDRESS_TY as object
(Street   VARCHAR2(50),
 City     VARCHAR2(25),
 State    CHAR(2),
 Zip      NUMBER);
/

create or replace type PERSON_TY as object
(Name VARCHAR2(25),
 Address ADDRESS_TY);
/
```

因为创建 CUSTOMER 表时未使用 ADDRESS_TY 和 PERSON_TY 数据类型，所以为了通过基于对象的访问(如方法)来访问 CUSTOMER 数据，需要使用对象视图。可以创建一个对象视图，它指定用于 CUSTOMER 表的对象类型。下面的程序清单创建 CUSTOMER_OV 对象视图。

```
create view CUSTOMER_OV (Customer_ID, Person) as
select Customer_ID,
       PERSON_TY(Name,
       ADDRESS_TY(Street, City, State, Zip))
  from CUSTOMER;
```

在创建 CUSTOMER_OV 对象视图的过程中，指定了两个对象类型(ADDRESS_TY 和 PERSON_TY)的构造函数方法。现在可以直接访问(作为一个关系表)或通过对象类型的构造函数方法访问 CUSTOMER 表。

CUSTOMER 表将在本章下面的示例中使用。

41.3.2 包含引用的对象视图

如果 41.3.1 节给出的 CUSTOMER 表与另一个表相关，则可以使用对象视图在两个表之间创建一个引用。即 Oracle 将使用已有的主键/外键关系来模拟在两个表之间 REF 使用的一些 OID。这样，就可以像关系表或对象一样访问表了。当把表看做对象时，可以使用 REF 自动执行表的连接(请参阅 41.2.4 节)。

CUSTOMER 表的主键为 Customer_ID。下面创建一个包含对 Customer_ID 列的外键引用的小表。下面的程序清单创建 CUSTOMER_CALL 表。CUSTOMER_CALL 表的主键为 Customer_ID 与 Call_Number 的组合。CUSTOMER_CALL 表的 Customer_ID 列是返回 CUSTOMER 表的外键，即不能对在 CUSTOMER 表中已经没有记录的客户记录一个调用。Call_Date 是在 CUSTOMER_CALL 表中创建的唯一的非键属性。

```
create table CUSTOMER_CALL
(Customer_ID     NUMBER,
 Call_Number     NUMBER,
 Call_Date       DATE,
 constraint CUSTOMER_CALL_PK
      primary key (Customer_ID, Call_Number),
 constraint CUSTOMER_CALL_FK foreign key (Customer_ID)
   references CUSTOMER(Customer_ID));
```

例如，在 CUSTOMER 表和 CUSTOMER_CALL 表可以插入以下的记录。

```
insert into CUSTOMER values
(123,'SIGMUND','47 HAFFNER RD','LEWISTON','NJ',22222);
insert into CUSTOMER values
(234,'EVELYN','555 HIGH ST','LOWLANDS PARK','NE',33333);
insert into CUSTOMER_CALL values

(123,1,TRUNC(SysDate)-1);
insert into CUSTOMER_CALL values
(123,2,TRUNC(SysDate));
```

从 CUSTOMER_CALL 表到 CUSTOMER 表的外键定义两个表之间的关系。从 OOP 的观点来看，CUSTOMER_CALL 表中的记录引用 CUSTOMER 表中的记录。因此，必须找到一种将 OID 值赋予 CUSTOMER 表中记录的方式，然后在 CUSTOMER_CALL 表中生成引用。

1. 怎样生成 OID

首先，用对象视图将 OID 值分配给 CUSTOMER 表中的记录。记住，OID 值被分配给对象表中的记录，且一个对象表依次基于一种对象类型。因此，首先要创建一个与 CUSTOMER 表有相同结构的对象类型：

```
create or replace type CUSTOMER_TY as object
(Customer_ID NUMBER,
 Name        VARCHAR2(25),
 Street      VARCHAR2(50),
 City        VARCHAR2(25),
 State       CHAR(2),
 Zip         NUMBER);
/
```

现在，创建基于 CUSTOMER_TY 类型的对象视图，同时将 OID 值分配给 CUSTOMER 表中的记录：

```
create or replace view CUSTOMER_OV of CUSTOMER_TY
with object identifier (Customer_ID) as
select Customer_ID, Name, Street, City, State, Zip
  from CUSTOMER;
```

此 create view 命令的第一部分为该视图命名(CUSTOMER_OV)，并告诉 Oracle 该视图的结构基于 CUSTOMER_TY 数据类型：

```
create or replace view CUSTOMER_OV of CUSTOMER_TY
```

create view 命令的下一部分告诉数据库如何为 CUSTOMER 表中的行构造 OID 值。with object identifier 子句后面是用于 OID 的列，在本例中为 Customer_ID 值。这将允许用户对 CUSTOMER 表中的行进行寻址，就像它们是对象表中可引用的行对象一样。

```
with object identifier (Customer_ID) as
```

create view 命令的最后一部分给出该视图的数据访问所基于的查询。该查询中的列必须与视图中基本数据类型相匹配：

```
select Customer_ID, Name, Street, City, State, Zip
  from CUSTOMER;
```

现在，通过 CUSTOMER_OV 视图可以访问 CUSTOMER 表的行，就像它们是行对象一样。CUSTOMER_OV 视图的行所生成的 OID 值称为 pkOID，因为它们基于 CUSTOMER 表的主键值。如果为关系表创建了对象视图，关系表就可以作为行对象来访问。

2. 怎样生成引用

CUSTOMER_CALL 表中的行引用 CUSTOMER 表中的行。从关系的观点来看，关系由从 CUSTOMER_CALL.Customer_ID 列指向 CUSTOMER.Customer_ID 列的外键决定。既然已经创建了 CUSTOMER_OV 对象视图，并且 CUSTOMER 表中的行可以通过 OID 访问，就需要在 CUSTOMER_CALL 表中创建引用 CUSTOMER 表的引用值。一旦 REF 函数准备好，就可以使用 DEREF 函数(见 41.2.5 节)从 CUSTOMER_CALL 表中访问 CUSTOMER 表的数据。

CUSTOMER_CALL 表的对象视图的 create view 命令在下面的程序清单中给出。它使用一个新的函数 MAKE_REF，如下面的程序清单所示。

```
create view CUSTOMER_CALL_OV as
select MAKE_REF(CUSTOMER_OV, Customer_ID) Customer_ID,
       Call_Number,
       Call_Date
  from CUSTOMER_CALL;
```

除了 MAKE_REF 操作外，此 create view 命令看起来像一个普通的 create view 命令。MAKE_ REF 操作如下所示：

```
select MAKE_REF(CUSTOMER_OV, Customer_ID) Customer_ID,
```

MAKE_REF 函数以要引用的对象视图名和构成本地表中外键的列名作为参数。在本例中，CUSTOMER_CALL 表的 Customer_ID 列引用在 CUSTOMER_OV 对象视图中作为生成 OID 基础的列。因此，要传递给 MAKE_REF 函数的两个参数为：CUSTOMER_OV 和 Customer_ID。为 MAKE_REF 操作的结果指定列的别名 Customer_ID。因为此命令创建了视图，所以为该操作的结果指定一个列的别名。

MAKE_REF 函数用来做什么呢？它创建从 CUSTOMER_CALL_OV 视图到 CUSTOMER_OV 视图的引用(称为 pkREF，因为它们基于主键)。现在可以查询这两个视图，就好像 CUSTOMER_OV 视图是一个对象表，和 CUSTOMER_CALL_OV 视图是包含引用 CUSTOMER_OV 视图的 REF 数据类型的一个表一样。

3. 查询对象视图

查询具有 REF 的对象视图的查询来反映表 REF 的查询结构。可以用 DEREF 函数选择被引用的数据值，如本章前面所示。使用对象视图以后，该查询如下所示：

```
select DEREF(CCOV.Customer_ID)
  from CUSTOMER_CALL_OV CCOV
 where Call_Date = TRUNC(SysDate);

DEREF(CCOV.CUSTOMER_ID)(CUSTOMER_ID, NAME, STREET, CITY, STATE,ZIP)
--------------------------------------------------------------------
CUSTOMER_TY(123, 'SIGMUND', '47 HAFFNER RD', 'LEWISTON','NJ',22222)
```

此查询在 CUSTOMER_CALL 表中查找 Call_Date 值为当前系统日期的记录。然后从该记录中获取了 Customer_ID 值，并评估其引用。从 MAKE_REF 函数得到的 Customer_ID 值指向 CUSTOMER_OV 对象视图中的 pkOID 值。CUSTOMER_OV 对象视图返回其 pkOID 值与被引用值相匹配的记录。然后 DEREF 函数返回被引用行的值。因此，该查询返回来自 CUSTOMER 表的行，尽管用户只查询 CUSTOMER_CALL 表。

列对象的对象视图允许像使用关系表和对象关系表那样使用表。当扩展到行对象时，对象视图允许基于已建立的外键/主键关系生成 OID 值。对象视图允许继续使用已有的约束和标准的 insert、update、delete 和 select 命令。它们还允许使用 OOP 功能，如对象表的引用。因此，对象视图为向 OOP 数据库结构的移植提供了重要的技术桥梁。

如本章前面所述，Oracle 执行连接，它们解析数据库中所定义的引用。当检索到被引用的数据时，它将返回被引用的整个行对象。为了引用该数据，需要在关系的“主键”表中创建 pkOID，并用 MAKE_REF 函数在关系的“外键”表中生成引用。然后可以像数据存储在对象表中那样使用该数据。

41.4　对象 PL/SQL

PL/SQL 程序可以使用已经创建的对象类型。尽管 PL/SQL 的早期版本只能使用 Oracle 提供的数据类型(如 DATE、NUMBER 和 VARCHAR2 等)，但是现在也可以使用属于自己的

对象类型，其结果是 SQL、过程逻辑和 OOP 扩展的相互融合，这种整合称为对象 PL/SQL。

以下匿名的 PL/SQL 块就使用了对象 PL/SQL 的概念。Cust1 变量的数据类型为 CUSTOMER_TY 对象类型，其值通过 CUSTOMER_OV 对象视图的查询赋予。

```
set serveroutput on
declare
   Cust1   CUSTOMER_TY;
begin
   select VALUE(COV) into Cust1
     from CUSTOMER_OV COV
    where Customer_ID = 123;
   DBMS_OUTPUT.PUT_LINE(Cust1.Name);
   DBMS_OUTPUT.PUT_LINE(Cust1.Street);
end;
/
```

此 PL/SQL 块的输出为：

```
SIGMUND
47 HAFFNER RD
```

此 PL/SQL 块的第一部分使用 CUSTOMER_TY 数据类型定义了一个变量：

```
declare
    Cust1       CUSTOMER_TY;
```

Cust1 变量的值选自 CUSTOMER_OV 对象视图(在 41.3 节已创建)。VALUE 函数用来在对象类型的结构中检索数据。由于数据在对象类型的格式中选择，因此需要使用在 CUSTOMER 表上创建的 CUSTOMER_OV 对象视图。

```
begin
    select VALUE(COV) into Cust1
      from CUSTOMER_OV COV
     where Customer_ID = 123;
```

然后，从 Cust1 变量的属性中检索并显示 Cust1.Name 和 Cust1.Street 的值。但无法将整个 Cust1 变量传递给 PUT_LINE 过程。

```
   DBMS_OUTPUT.PUT_LINE(Cust1.Name);
   DBMS_OUTPUT.PUT_LINE(Cust1.Street);
end;
```

虽然这是一个精心设计的简单示例，但它显示了对象 PL/SQL 的功能。在所有使用对象类型的地方，均可以使用对象 PL/SQL。这样，PL/SQL 将不再局限于 Oracle 提供的数据类型，并且可以更为精确地反映数据库中的对象。在此例中，查询了一个对象视图，以说明查询可以访问列对象或行对象。接着，可以选择对象类型的属性，并操作或显示它们。如果已经给对象类型定义了方法，那么也可以使用它们。

例如，在 PL/SQL 块中可以调用对象类型的构造函数方法。在下例中，NewCust 变量使用 CUSTOMER_TY 数据类型来定义。接着，使用 CUSTOMER_TY 构造函数方法设置 NewCust

变量等于一组值。然后通过 CUSTOMER_TY 对象视图插入 NewCust 变量的值。

```
declare
   NewCust CUSTOMER_TY;
begin
   NewCust :=
     CUSTOMER_TY(345,'NewCust','StreetVal', 'City','ST',00000);
   insert into CUSTOMER_OV
   values (NewCust);
end;
/
```

通过查询 CUSTOMER_OV 对象视图，可以看到该 insert 操作的结果：

```
select Customer_ID, Name from CUSTOMER_OV;

CUSTOMER_ID  NAME
-----------  -------------------------
        123  SIGMUND
        234  EVELYN
        345  NewCust
```

除调用构造函数方法外，还可以调用已经在对象类型上创建的方法。如果试图比较使用对象类型的变量值，则应当为数据类型定义映射方法或排序方法。这样可以进一步扩展对象 PL/SQL，即在数据库级定义数据类型和函数，并且它们可以在任何 PL/SQL 程序中调用。对象 PL/SQL 显著增强了传统的 PL/SQL。

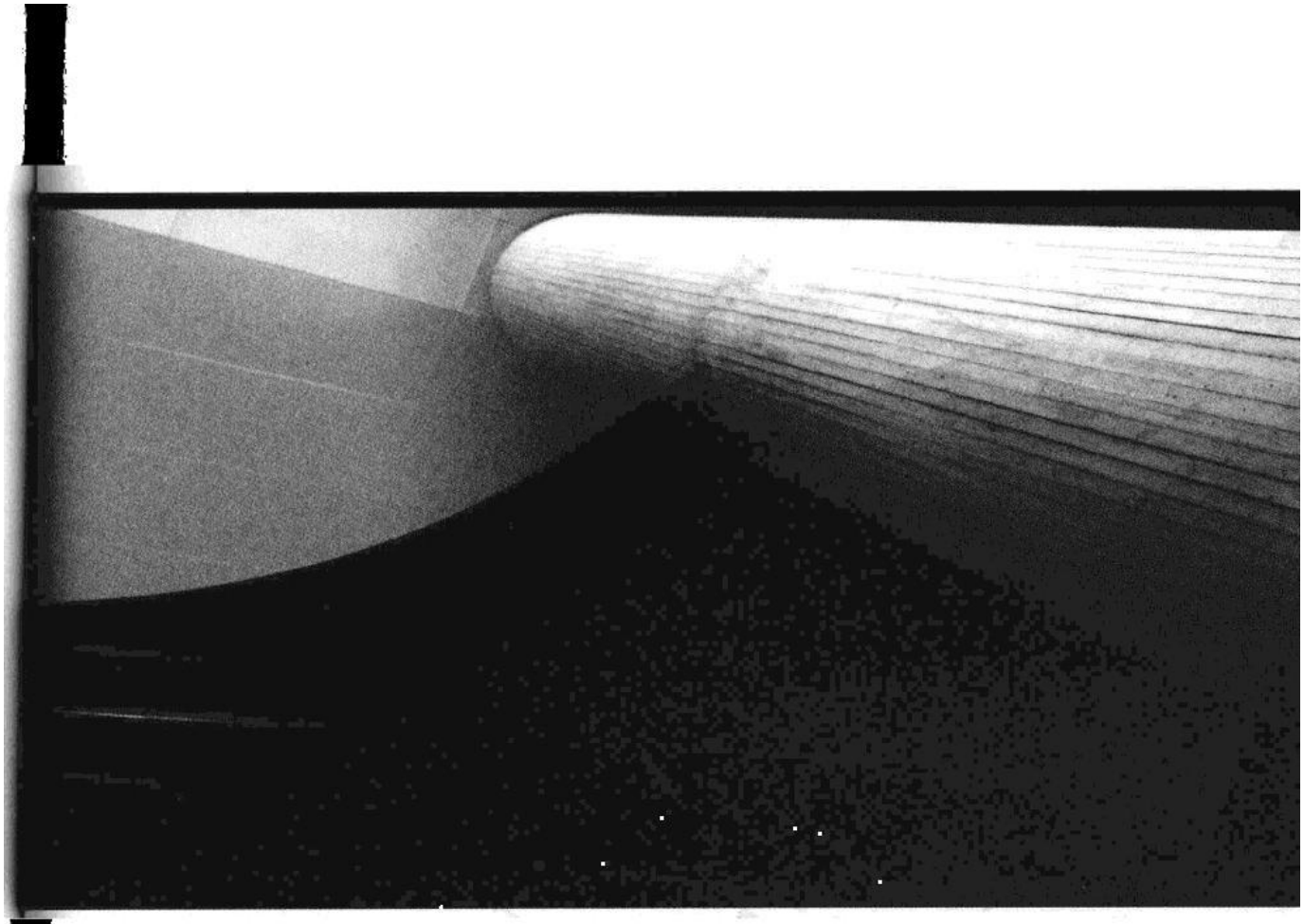
41.5　数据库中的对象

Oracle 中可用的特性，即列对象、行对象以及 PL/SQL 的对象扩展，允许在数据库中实现对象，而无需牺牲在分析和设计阶段已经做出的投资。用户可以继续基于关系设计技术创建系统，并且基于关系访问方法调整它们。使用 Oracle 提供的工具，可以在关系表上创建一个 OOP 层。一旦有了合适的 OOP 层，就可以访问关系数据，如同它是存储在整个 OOP 数据库中一样。

有了 OOP 层，就可以体会到 OOP 系统的某些优点，如抽象和封装。可以将每一个对象类型的方法应用于一组实现方法一致的对象，并可从对象的重用和标准实现中获得好处。同时，也能从 Oracle 的关系特性中获益。在一个应用程序中同时使用关系和对象技术允许在数据库中为相应的作业使用相应的工具。

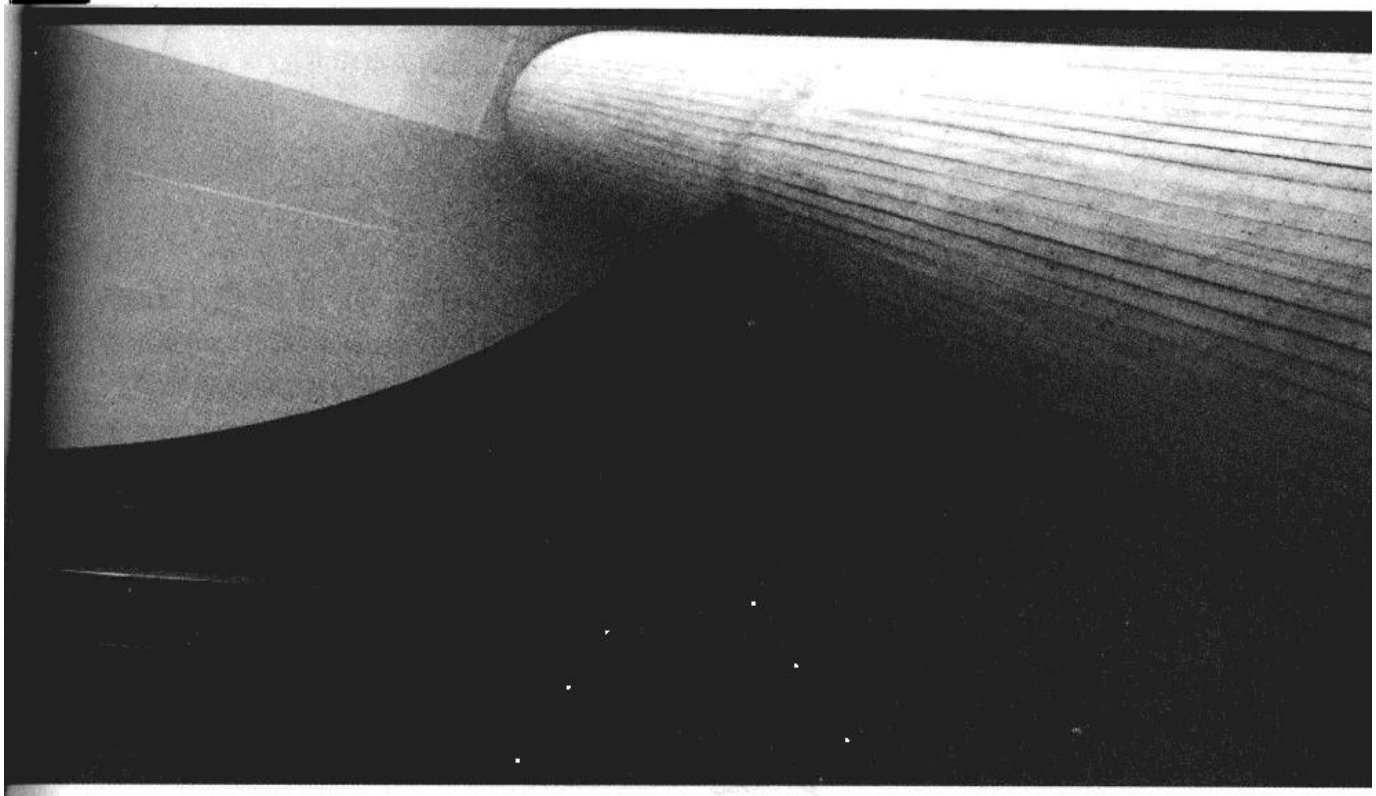
要注意，实现对象关系特性可能促使用户改变处理数据关系的方法和访问数据所使用的工具。如果使用行对象，那么表之间的关系存在于行对象之间，而不是存在于行对象的数据之间，即可能导致悬挂的 REF。此外，除非使用对象视图，否则对查询和 DML 操作需要采用新的 SQL 语法。

实现对象关系数据库的对象部分时，从定义对象类型(业务的核心组件)开始。每一个对象关系特性(不管与列对象还是与行对象相关)都基于一个对象类型。对象类型及其方法定义得越好，对象的实现就越好。根据需要应当嵌套对象，以便使同一个核心对象类型有多种变化。结果将获得一个设计恰当的数据库，以便利用关系和面向对象特性。



第VI部分

Oracle 中的 Java



第 42 章

Java 简介

本章将简要介绍与 Oracle 数据库应用程序相关的 Java。Java 除了用于 Oracle 应用程序之外，还有很多别的用途，Java 语言有许多功能，大多数 Oracle 开发人员并不使用这些功能。本章旨在为具有 SQL 和 PL/SQL 背景知识的开发人员提供 Java 语言结构的基本知识。这里并不详细地回顾 Java 语言(有很多这方面的书籍可供参考)，而是简要地概述 Oracle 开发人员最常使用的 Java 语言组件。

尽管 PL/SQL 和 Java 有非常相关，但它们在术语和用法上还是存在许多差别。因为它们是独立开发出来的，所以存在差异不足为奇。此外，PL/SQL 的面向对象功能并不是与生俱来的，而 Java 天生就具有面向对象特性。为了有效地使用 Java，用户应当使用不同于过去使用 PL/SQL 的方法。

第 32 章介绍了 PL/SQL 语言的结构和流控制。本章将简要介绍 Java 的结构和基本流程

控制方法，并根据需要介绍与之相对应的 PL/SQL 结构。在阅读本章之前，用户应当对 PL/SQL(第 32 章)、程序包、函数、过程(第 35 章)以及用户定义的类型和方法(第 38 章)有所了解。

42.1 Java 与 PL/SQL 概述

比较一下 PL/SQL 与 Java，除了两种语言的术语有很大的差别之外，最大的差异莫过于基本的 PL/SQL 结构——块。在 PL/SQL 中，块是一个拥有声明(Declaration)部分、可执行命令(Executable Command)部分以及异常处理(Exception Handling)部分的结构化代码块。PL/SQL 块的结构如下所示：

```
declare
  <declarations section>
begin
  <executable commands>
exception
  <exception handling>
end;
```

在 Java 中，术语“块”指的是很小的代码子集。在 Java 中，块是由一对花括号“{”和“}”括起来的语句集合。例如，下面的伪代码含有两个 Java 块：

```
if (some condition) {
  block of code to process if condition is met
}
else {
  block of code to process if condition is not met
}
```

在 PL/SQL 中，上面的全部代码部分仅仅是 1 个较大块的组成部分。而在 Java 中，它却包含了两个独立的块。在本章中，如果没有特别说明，则“块”都指的是 Java 块。

PL/SQL 的过程以 begin 和 end 为界，而 Java 的方法则以起始花括号和闭合花括号为界。这些花括号还用于隔离作用域内不同的代码块，或作为条件段开始和结束的指示器。

PL/SQL 和 Java 之间的另一个主要差别在于变量的声明。在 PL/SQL 块中，在可执行命令部分之前定义变量；而在 Java 程序中，可以在程序中根据需要定义变量。Java 程序没有 PL/SQL 块中使用的声明部分。

本章还将提到 Java 和 PL/SQL 之间的很多不同之处。但重要的是记住，Java 并不能取代应用程序中的 PL/SQL——您还可以继续将 PL/SQL 用于 Oracle 应用程序。对于数据检索操作时，在决定使用哪种技术语言之前，应当先对这两种语言的性能进行测试。

42.2 开始

为了使用本节的示例，需要 Java Development Kit(JDK)的一个副本，此副本可以从

http://java.sun.com/网站免费下载。并将它安装到将要运行 Java 程序的服务器上。

注意：
JDK 可作为 Oracle Database 安装程序的一部分。请确保同时安装了数据库核心软件和相关的客户端软件。

42.3 声明

在 Java 中，可以根据需要对变量进行声明。变量具有变量名和数据类型，在程序运行时用来存储数据值。变量也有一个作用域，该作用域决定变量是公有的还是私有的——类似于程序包中的过程可以拥有私有变量。

下面是声明 Java 变量的示例：

```
char aCharVariable = 'J';
boolean aBooleanVariable = false;
```

按照约定，Java 变量总是以小写字母开头，如本例所示。在此示例中，其数据类型是 CHAR 和 BOOLEAN，而且每个变量都赋予了初始值。请注意，Java 中的赋值符号是“=”，而在 PL/SQL 中相应的符号是“:=”或者“=>”。每个声明都以分号结束。

Java 中可用的基本数据类型如表 42-1 所示。

表 42-1 Java 中的基本数据类型

数 据 类 型	说 明
Byte	字节长度的整数
Short	短整数
Int	整数
Long	长整数
Float	单精度浮点实数
Double	双精度浮点实数
Char	单个 Unicode 字符
Boolean	Boolean 值，真或假

除表 42-1 中列出的基本数据类型外，还可以使用引用数据类型，这根据变量的内容而定。值得注意的是，没有针对可变长度字符串的基本数据类型，Java 为此提供了一个 String 类。关于类的概念将在本章后面进行讨论。

42.4 可执行命令

可以通过表达式和语句给变量赋值。Java 支持的算术运算符如表 42-2 所示。

表 42-2 Java 支持的算术运算符

运 算 符	说 明
*	乘法
/	除法
+	加法
-	减法
%	取模

除了这些运算符外，还可以使用 Java 中的一元运算符来简化编码。例如，可以利用“++”运算符对变量进行递增运算，如下所示：

```
aLoopCounter = aLoopCounter++;
```

或者可以简单地写为：

```
aLoopCounter++
```

注意：

“++”运算符之所以被称为一元运算符，是因为它只需要 1 个操作数。如果某个运算符需要两个操作数(如*)，就将其称为二元运算符。

在 PL/SQL 中，上面的语句将写为：

```
aLoopCounter := aLoopCounter +1;
```

如果将“++”运算符放在变量名的前面，它就是一个先递增的运算符而不是后递增的运算符：

```
int anotherCounter = ++aLoopCounter;
```

在此例中，变量 anotherCounter 被设置为 aLoopCounter 的递增值。而

```
int anotherCounter = aLoopCounter++;
```

在 aLoopCounter 递增之前就将值设置为 anotherCounter。还可以通过一元运算符“——”来递减变量：

```
aLoopCounter = aLoopCounter--;
```

还可以将赋值与运算符组合起来。例如：

```
aNumberVariable = aNumberVariable * 2;
```

可写成

```
aNumberVariable *= 2;
```

还可以用/=、%=、+=以及-=来执行类似的组合运算。

如果执行多个算术运算，则应当用圆括号清晰地指出执行运算的顺序，如下例所示：

```
aNumberVariable = (aNumberVariable*2) +2;
```

用分号结束这些运算，使它们成为完整的执行单元，即语句。

42.4.1 条件逻辑

可以使用表达式和语句对变量求值。为了求值，可能需要使用 Java 提供的 1 个或多个基本类。完全列出这些方法超出了本书的范围。关于当前提供的功能，请参阅 Sun 公司的 Java 文档网站或参阅有关的 Java 书籍。

注意：
由于 Java 版本不同，所提供类的数量也迥然不同。

在下面的程序清单中，可以使用 Character.isUpperCase 方法对前面声明过的变量 aCharVariable 进行判断：

```
if (Character.isUpperCase(aCharVariable)) {
  some commands to execute
}
```

如果 aCharVariable 值为大写，则执行后面的块中的命令；否则，将继续执行程序的下一部分。

下面是 if 子句的一般语法：

```
if ( expression) {
  statement
}
```

请注意，其中没有 then 子句。如果表达式判定为真，则自动执行后面的块。注意也没有 end if 子句。Java 块简单地以语句的闭合花括号结尾。

还可以用 else 子句判定多个不同的条件，如下面程序清单所示：

```
if (expression) {
  statement
}
else {
  statement
}
```

如果需要检查多个条件，则可以用 else if 子句依次进行判断，以 else 子句结束。下面的程序清单说明了 else if 子句的用法：

```
if (aCharVariable == 'V') {
  statement
}
else if (aCharVariable == 'J') {
  statement
}
else {
  statement
}
```

注意：
PL/SQL 支持 elsif 子句对可选条件进行选择。一个 if 子句可以有多个 elsif 子句。

除了支持条件逻辑的 if 子句外，Java 还提供了 switch 子句。它与 break 子句和语句标号一起使用，switch 子句的作用和 goto 子句大致相同(Java 不支持 goto 语句)。首先看一个 switch 子句的简单示例。使用 case 语句判断 aMonth 变量的多个值。根据 aMonth 的值，显示月份的文字名称，并通过 break 跳出 switch 块。

```
int aMonth=2;
switch (aMonth) {
  case 1:
       System.out.println("January");
       break;
  case 2:
       System.out.println("February");
       break;
  case 3:
       System.out.println("March");
       break;
  default:
       System.out.println("Out of range");
       break;
}
```

注意：
在 Java 中，case 语句的参数必须是整数。

虽然编写这种代码比简单的 TO_CHAR 要复杂得多，但它说明了 swtich 的用法。switch 运算符将 aMonth 变量作为输入并判定其值。如果 aMonth 的值与某个 case 值匹配，则调用 System.out.println 方法输出相应月份的名称，且通过 break 决定何时跳出 switch 块。如果 aMonth 的值和 case 子句的任何值都不匹配，则执行 default 选项。

那么控制权到底转到哪里去了呢？默认情况下，将转向程序的下一部分。但是，可以选择为代码段建立标号，并把标号名传递给 break 子句。下面的程序清单给出了标号和 break 子句的示例：

```
somelabel:
if (aCharVariable == 'V') {
  statement
}
else {
  statement
}
int aMonth=2;
switch (aMonth) {
```

```
    case 1:
       System.out.println("January");
       break somelabel;
    case 2:
       System.out.println("February");
       break someotherlabel;
    case 3:
       System.out.println("March");
       break somethirdlabel;
    default:
       System.out.println("Out of range");
       break;
}
```

在此示例中，首先判断 if 子句，接着判断 switch 子句。请注意，aMonth 变量直到使用前才声明。就本示例而言，由于变量 aMonth 的赋值为 2，因此程序将显示“February”，并将分支转向 someotherlable 标号标出的代码段。如果 aMonth 值为 1，则将重新执行位于代码段前面的 if 子句。

Java 还支持三元运算符——具有 3 个操作数的运算符，其功能类似于 DECODE 的功能。其作用就像内联的 if-else 组合一样，三元运算符将会判断一个表达式的值。如果该表达式的值为真，则返回第 2 个操作数；否则，返回第 3 个操作数。其语法如下：

```
expression ? operand1 : operand2
```

下面的程序清单给出了三元运算符的一个示例：

```
aStatusVariable = (aCharVariable == 'V') ? "OK" : "Invalid";
```

此示例对表达式

```
(aCharVariable == 'V')
```

进行判定。如果其值为真，则返回“OK”；否则，返回“Invalid”。

可以用该三元运算符模拟 GREATEST 函数。

```
double greatest = (a > b) ? a : b;
```

此示例判断表达式(a > b)，其中 a 和 b 是两个变量名。如果此表达式的值为真，则返回 a 的值；否则，返回 b 的值。

注意：

Java 有一个用于数值的“最大(greatest)”方法，即 java.lang.Math 类中的 Max 方法。前面程序清单中的三元运算符的示例可重写为 double greatest = Math.Max(a,b)。

还可以使用 AND 和 OR 运算来组合多个逻辑检查。在 Java 中，AND 运算用“&&”运算符表示。

```
if (aCharVariable == 'V' && aMonth == 3) {
   statement
```

```
}
```

OR 运算用“||”表示，如以下面的程序清单所示：

```
if (aCharVariable == 'V' || aMonth == 3) {
  statement
}
```

出于性能的原因，仅在需要时才执行“&&”和“||”运算；如果第 1 个表达式决定了结果，则不计算第 2 个表达式。对于 AND 运算符，如果第 1 个值为假，则计算结束并返回布尔值假。对于 OR 运算符，如果第 1 个表达式为假，则还要计算第 2 个表达式，因为只要其中一个测试条件为真就会返回 Boolean 值真。如果 OR 运算符的第 1 个值为真，则不需要判断第 2 个表达式，直接返回真。

“&”和“|”运算符是按位 AND 和 OR，它们不像“&&”和“||”那样简单。要判断所有表达式。如果需要测试方法调用的返回值并保证执行这些方法，那么此功能非常重要。

42.4.2 循环

Java 支持 3 种主要的循环：WHILE 循环、DO-WHILE 循环和 FOR 循环。本节将介绍每种循环的示例。因为 Java 不是 SQL 的扩展，所以它不像 PL/SQL 那样支持游标 FOR 循环。

1. WHILE 循环和 DO-WHILE 循环

while 子句判断一个条件。如果此条件为真，则执行相应的语句块。WHILE 循环的语法形式为：

```
while (expression) {
  statement
  statement
}
```

WHILE 循环重复执行相应的语句块(程序中花括号“{”和“}”之间的部分)直到表达式(expression)的值不再为真。

```
int aNumberVariable = 3;
while (aNumberVariable < 7) {
  some_processing_statement
  aNumberVariable++;
}
```

在此示例中，创建计数器变量(aNumberVariable)，并进行初始化。通过 while 子句判断该变量的值。如果该值小于 7，就执行相应的语句块。作为块的一部分，递增该变量。当块中的语句执行完毕后，再次判断变量的值以决定循环是否继续。

注意：
关于 WHILE 循环处理的示例，请参阅本章 42.5 节。

也可以将该循环写成 DO-WHILE 循环：

```
int aNumberVariable = 3;
do {
  some_processing_statement
  aNumberVariable++;
} while (aNumberVariable < 7);
```

在 DO-WHILE 循环中，块至少循环一次，才会判断表达式的值。

2. FOR 循环

可以使用 FOR 循环反复执行一个代码块。FOR 循环指定初始值、结束值和循环计数器的递增量。循环计数器每执行一次循环都会递增指定的量，直到到达结束值为止。下面是 FOR 循环的语法：

```
for ( initialization; termination; increment) {
  statement;
}
```

前一小节的 WHILE 循环示例也可以用 for 子句重写，如下面的程序清单所示：

```
for (int aNumberVariable=3; aNumberVariable<7; aNumberVariable++) {
  some_processing_statement
}
```

用 for 子句编写的示例比用 while 子句编写的要简短得多。在此示例的 FOR 循环中，声明了变量 aNumberVarible，并对它进行初始化。只要此变量的值不超过 7，块就会反复执行。每执行一次循环，变量的值都会由一元运算符“++”递增 1。

在循环内部，可以使用 continue 子句直接跳转到循环内的另一条语句。如果仅仅使用 continue 子句本身，则处理流程将直接跳到循环体的最后，并判断测试循环结束的条件。如果使用带标号(如 42.4.1 节关于 switch 子句的标号)的 continue，那么将在带标号的循环的下一次迭代时继续执行。

3. 游标和循环

在 PL/SQL 内，可以使用游标 FOR 循环遍历查询的结果。可以使用 Java 的 WHILE 和 FOR 循环模拟 PL/SQL 的游标 FOR 循环的功能。

Oracle 的软件安装程序同时提供了示例文件，用大量示例演示说明标准的编码方法。客户安装程序使用 Oracle 通用安装程序(Oracle Universal Installer)加载软件驱动程序和演示文件，包括那些支持 JDBC 开发的文件。根据客户软件的安装程序，JDBC 示例将存储在 Oracle 软件主目录下与 Java 相关的子目录中。不同的版本所提供的文件可能会有所不同。

注意：

Oracle 软件主目录的/jdbc 子目录下的 README 文件提供了一些扩展信息，包括支持的版本、平台的环境变量设置和支持的特性。为了有效地在您的平台上对 Oracle 使用 Java，阅读并遵循这些提示信息是非常重要的。

下面的示例显示了 Oracle 客户安装程序提供的一个示例中的一段代码：

```
Statement stmt conn.createStatement();
ResultSet rset = stmt.executeQuery ("select FIRST_NAME, "
                                    + "LAST_NAME from EMPLOYEES");
   // Iterate through the result and print the employee names
   while (rset.next ())
     System.out.println (rset.getString (1) + " " + rset.getString (2));
```

此示例假设已经建立了数据库连接。示例的第 1 行根据连接变量 conn(在这部分程序清单中没有显示)创建变量 stmt。示例的第 2 行根据查询 stmt 将返回的结果集创建变量 rset。注释(以“//”为前缀)之后的 while 子句取出结果集中的每一行。如果从结果集中检索到一行并取出该行，则输出该 EMP 行的 ENAME；否则，循环结束。

从程序设计的观点来看，用户应该清楚返回的数据(像前面的示例一样)，指定要返回列的名称。

Oracle 提供的 JDBC(Java Database Connectivity，参阅第 43 章)演示给出了涉及过程调用、LOB 和 DML 的查询示例。在大多数情况下，Java 代码的执行方式与前面示例相似，即用 Java 编写进程流控制语句，并传递给它一条要执行的语句。基于该语句的输出，可以使用 while、for、if、break、continue 和 switch 子句转到不同的代码行，如本章前面所示。

42.4.3 异常处理

在 PL/SQL 过程中，有一个 EXCEPTION 块。在 Java 中，可以具有任意多的 try/catch 块。Java 提供一组可靠的错误处理例程，允许用户创建复杂的错误处理过程。在 PL/SQL 中，raise 操作抛出一个异常；而在 Java 中，throw 操作抛出多个异常。如果抛出异常，则应当使用 Java 的 catch 子句捕获并正确地处理该异常。

Java 的异常处理语法基于 3 个块：try、catch 和 finally。try 块包含可能抛出异常的语句。紧跟在 try 块后面的 catch 块将异常处理程序和 try 块可能抛出的异常相关联。finally 块则清除在 catch 块处理中没有正确清除的系统资源。其一般结构为：

```
try {
  statements
}
catch ( ) {
  statements
}
catch ( ) {
  statements
}
finally {
  statements
}
```

例如，考虑下面的 Java 代码：

```
try {
```

```
    Statement stmt = conn.createStatement ();

    stmt.execute ("drop table plsqltest");
}
catch (SQLException e) {
}
```

此示例创建变量 stmt 来执行 drop table 命令。如果执行失败——例如，该表不存在，那么如何处理这个错误呢？catch 子句告诉 Java 通过异常对象 SQLException(Java 的 SQL 实现的标准部分)对它进行处理。如该示例所示，catch 子句接受两个参数(异常对象和变量名)，可以选择性在后面跟一个将要执行的语句块。

try 块可以包含多条语句，或者可以为每条语句创建独立的 try 块。一般来说，如果和 catch 块联合使用，则管理异常处理更为容易，因此包含 try 块将有助于在更改和增加代码时管理代码。PL/SQL 中的 when others 子句和 Java 中的 catch 子句(Throwable th)本质上一致。

finally 块在将控制权传递给程序的后续部分之前，清除当前代码段的状态。在运行时，无论 try 块的输出怎样，都会执行 finally 块的内容。例如，可以使用 finally 块关闭数据库连接。

42.4.4 保留字

表 42-3 列出了 Java 中的保留字。不能将这些保留字用作类、方法或变量的名称。

表 42-3 Java 的保留字

abstract	else	interface	super
boolean	extends	long	switch
break	false	native	synchronized
byte	final	new	this
case	finally	null	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	true
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while
double	int	strictfp	

42.5 类

本章前面几节介绍了 Java 的基本语法结构。本节将介绍怎样使用这些语法创建并使用对象。下面的程序清单说明了如何输出单词“Oracle”：

```
public class HelloOracle {
  public static void main (String[] args) {
    System.out.println("Oracle");
  }
}
```

该示例创建了类 HelloOracle。在此类中，创建了公有方法 main。main 方法输出单词“Oracle”。这比

```
select 'Oracle' from dual;
```

更为复杂，但 Java 具有 SQL 所没有的特点。由于 HelloOracle 是类，因此可以从其他类中调用其方法。HelloOracle 可拥有多个方法。在此示例中，它只有 main 方法。

main 方法的声明(public static void main)具有很多用户熟悉的关键字：

- **public** 定义类为 public，允许所有类调用此方法
- **static** 声明这是一个类方法，可以用 static 声明类方法。另一个可选的关键字 final 用于声明那些值不能更改的方法和变量(类似于 PL/SQL 的 constant 选项)
- **void** 指出从过程返回的值的数据类型。因为该过程没有返回值，所以其类型为 void

在本示例中，给方法命名为 main，这样可以简化类方法的执行。既然已经创建了类，就可以编译和加载它了。

注意：

为了编译和执行 Java 类，必须在服务器上安装 JDK。下面的示例假设作为此软件包组成部分的二进制程序位于通过 PATH 环境变量自动搜索到的目录之下，而且包含类文件的目录包含在 CLASSPATH 环境变量中(关于环境变量设置的详细内容，请参见/jdc/README 文件)。这里涉及的程序(javac 和 java)位于 JDK 软件目录的/bin 子目录中。

首先，把该程序另存为名为 HelloOracle.java 的纯文本文件。接着进行编译：

```
javac HelloOracle.java
```

它将在服务器上创建名为 HelloOracle.class 的新文件。然后可以运行此类：

```
java HelloOracle
```

相应的输出为：

```
Oracle
```

如 HelloOracle 示例所示，首先为类命名并指定与继承其他类的属性相关的可选属性。然后在类体中定义所有类变量和方法。与 Oracle 中的抽象数据类型一样，Java 类也具有与自身

相关的方法，用于操作相关的数据。

现在考虑一个更为复杂的示例。下面的程序清单是计算圆面积的 Java 程序，半径值为给定的输入(关于该程序的 PL/SQL 版本，请参阅第 32 章)：

```
// AreaOfCircle.java
//
public class AreaOfCircle {
  public static void main(String[] args) {
    try {
         int input = Integer.parseInt(args[0]);
         double result=area(input);
         System.out.println(result);
    }
    catch (ArrayIndexOutOfBoundsException oob) {
         System.out.println("You did not provide the radius of the circle.");
         System.out.println("Usage: java AreaOfCircle 10");
    }
    catch (NumberFormatException nfe) {
         System.out.println("Enter a valid number for the radius.");
         System.out.println("Usage: java AreaOfCircle 10");
    }
  } // main

  public static double area (int rad) {
    double pi=3.1415927;
    double areacircle=pi*rad*rad;
    return areacircle;
  } // area
} //AreaOfCircle
```

注意：

每个块一定要包括一个闭合花括号“}”。在本章给出的示例中，闭合花括号“}”总是单独占一行，以便简化调试和处理可能产生的流控制错误。

首先应该注意，类和文件名必须相同。这里，因为该类命名为 AreaOfCircle，所以此文本就存储为一个名为 AreaOfCircle.java 的文件中。该类有两个方法，分别是 main 和 area。在执行该类时，将自动地执行 main 方法。main 方法将提供的字符串作为输入，并将它解析成整数，放入到 input 变量中。

```
int input=Integer.parseInt(args[0]);
```

在执行此代码时，至少会抛出两个异常。第 1 个异常 ArrayIndexOutOfBoundsException 在试图引用一个数组中不存在的某个元素时出现。因为代码显式引用 args 数组的第 1 个元素(args[0])，所以如果没有数据传递给该程序，就会抛出异常。如果传入非数值的值，则会出现另一个可能的错误 NumberFormatException 异常。例如，如果输入“java AreaOfCircle XYZ”而不是“java AreaOfCircle 123”来执行此程序，则会抛出 NumberFormatException 异常，因

为 XYZ 不是有效的数字。不管什么原因引起的错误，这两个异常都通过两个 catch 语句来处理，向用户提供有用的反馈信息：

```
catch (ArrayIndexOutOfBoundsException oob) {
    System.out.println("You did not provide the radius of the circle.");
    System.out.println("Usage: java AreaOfCircle 10");
}
catch (NumberFormatException nfe) {
    System.out.println("Enter a valid number for the radius.");
    System.out.println("Usage: java AreaOfCircle 10");
}
```

接着，声明变量 result，并使它等于调用 area(input)方法的返回值：

```
double result=area(input);
```

为了处理这条指令，将 input 变量的值作为它的输入，从而执行 area 方法。area 方法如下：

```
public static double area (int rad) {
  double pi=3.1415927;
  double areacircle=pi*rad*rad;
  return areacircle;
}
```

在其定义中，area 方法指出它将接受整型变量 rad。然后处理变量的值(来自 main 方法中的 input 变量)，计算变量 areacircle，并返回给 main 方法。Main 方法将输出该值：

```
double result=area(input);
System.out.println(result);
```

此程序的测试很简单：

```
javac AreaOfCircle.java
java AreaOfCircle 10
```

输出的结果为：

```
314.15927
```

扩展此示例，使其包含 WHILE 循环过程。此示例将反复调用 area 方法，直到最终的 input 的值大于给定的值为止：

```
// AreaOfCircleWhile.java
//
public class AreaOfCircleWhile {
  public static void main(String[] args) {
    try {
      int input=Integer.parseInt(args[0]);
      while (input < 7) {
        double result=area(input);
        System.out.println(result);
        input++;
```

```
        }
      }
      catch (ArrayIndexOutOfBoundsException oob) {
         System.out.println("You did not provide the radius of the circle.");
         System.out.println("Usage: java AreaOfCircle 10");
      }
      catch (NumberFormatException nfe) {
         System.out.println("Enter a valid number for the radius.");
         System.out.println("Usage: java AreaOfCircle 10");
      }
    } // main
    public static double area (int rad) {
      double pi=3.1415927;
      double areacircle=pi*rad*rad;
      return areacircle;
    } // area
  } // AreaOfCircleWhile
```

在此程序清单中，area 方法和异常处理块与前面的示例相比，没有什么大的变化。唯一的变化出现在 main 方法中：

```
int input=Integer.parseInt(args[0]);
while (input < 7) {
   double result=area(input);
   System.out.println(result);
   input++;
}
```

当 input 的值小于 7 时，将会对 input 的值进行循环处理。在计算给定半径的圆面积并输出面积后，递增 input 的值：

```
input++;
```

再一次判断是否进行循环。下面的程序清单给出了示例的输出：

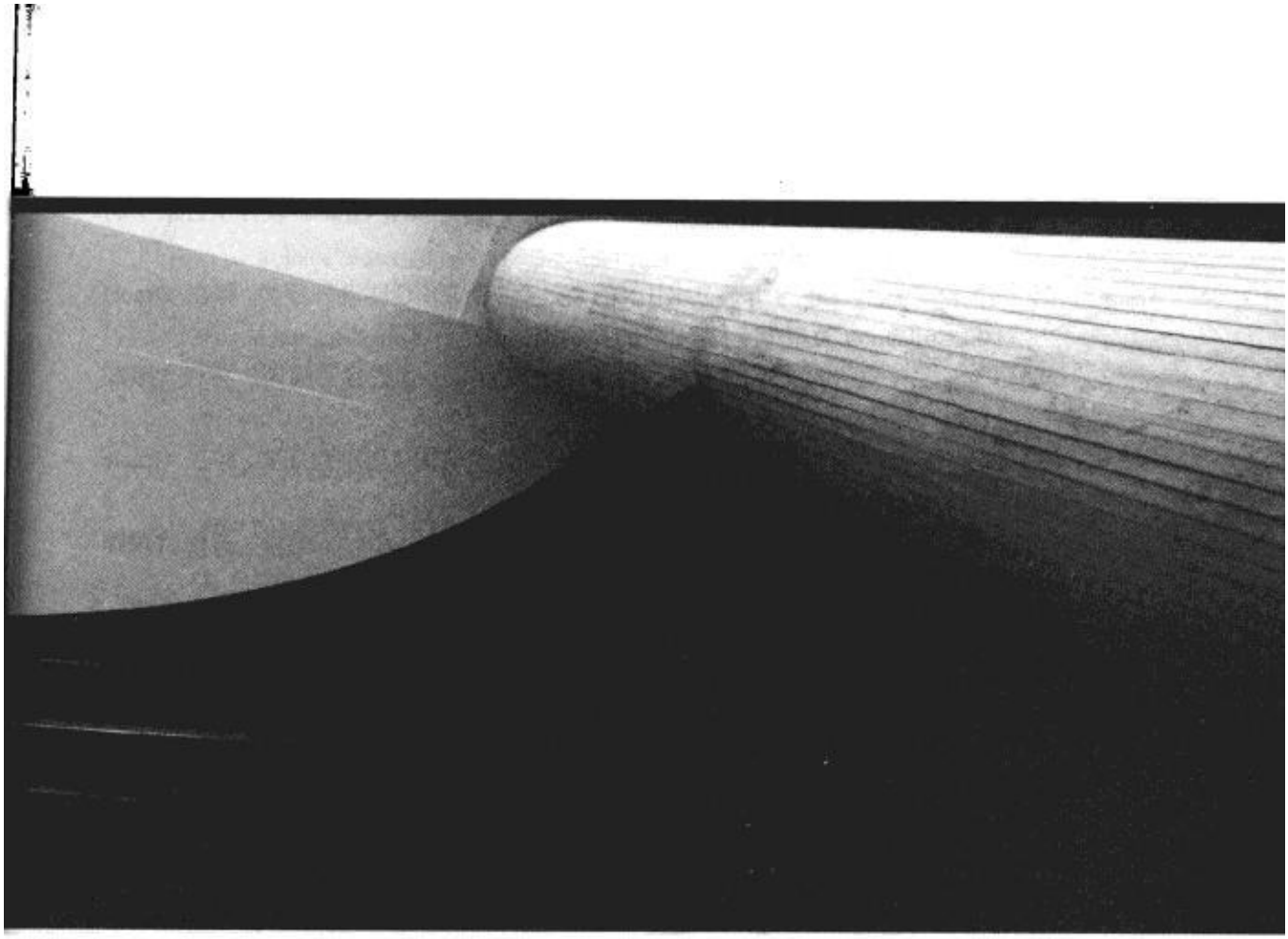
```
javac AreaOfCircleWhile.java
java AreaOfCircleWhile 4

50.2654832
78.5398175
113.0973372
```

此输出结果显示了输入半径值分别为 4、5、6 时的圆面积。

该示例并不是一个完整的程序——例如，可能需要其他处理非整数的异常，但是更重要的是学习这个示例的结构。除了该示例给出的方法外，还可以为 AreaOfCircleWhile 类创建一个构造函数。所有 Java 类都有构造函数(参阅第 38 章)，以初始化基于该类的新对象。构造函数的名称与该类的名称相同。如果创建一个没有构造函数的类，则 Java 会在运行时创建一个默认的构造函数。与方法类似，构造函数体可以包含局部变量声明、循环以及语句块。构造

函数为类初始化这些变量。

下面几章将介绍怎样在 Oracle 内实现 Java(在存储过程内且通过 JDBC)。这几章都假设读者具有本章所介绍的 Java 的基本知识。关于 Java 语言的详细信息，请参阅 Sun Microsystems Web 站点和 Java 的各种专著。



第43章

JDBC程序设计

JDBC(Java Database Connectivity，Java数据库连接)建立在本书前面描述的Java和编程基础上。本章的讨论和示例假定用户熟悉第42章所描述的Java语法和结构。本章并非面面俱到地介绍JDBC，而是主要介绍其基本用法。

可以使用JDBC在Java程序中执行动态SQL语句。Oracle软件安装程序提供了示例文件。创建自己的JDBC程序时可以使用这些示例文件作为范例。

注意：

Oracle 11g不支持J2SE 1.2、1.3和1.4。支持这些J2SE版本的.zip和.jar文件不包含在Oracle JDBC 11g中。如果仍在使用J2SE 1.2、1.3或1.4，则可以继续使用有这些Java版本的Oracle JDBC 10.2.0.1.0。仍支持将Oracle 10.2用于这些版本的Java。如果想要使用Oracle JDBC 11g，则必须使用J2SE 5.0或更高的版本。

下面的库和文件是 Oracle 11g 安装程序的一部分，它们可以从 Oracle 软件主目录下的 /jdbc/lib 目录中获得：

- ojdbc5.jar——此文件包含用于 JDK 1.5 的类。它包含 JDBC 驱动程序类，和在 Oracle Object 和 Collection 类型中支持 NLS 的异常类。
- ojdbc5_g.jar——与 ojdbc5.jar 一样，除了类已用 javac –g 进行了编译，并包含跟踪代码。
- ojdbc6.jar——此文件包含用于 JDK 1.6 的类。它包含 JDBC 驱动程序类，和在 Oracle Object 和 Collection 类型中支持 NLS 的异常类。
- ojdbc6_g.jar——与 ojdbc6.jar 一样，除了类已用 javac –g 进行了编译，并包含跟踪代码。

Oracle JDBC 的公共类的公共 API 的 JDBC Javadoc 可以在 ORACLE_HOME/jdbc/doc/javadoc.tar 文件中找到。示例 JDBC 程序在 ORACLE_HOME/jdbc/demo/demo.tar 中提供。对于国家语言支持(NLS)，文件 ORACLE_HOME/jlib/orai18n.jar 提供了用于 JDK 1.5 和 JDK1.6 的 NLS 类。

当通过 JDBC 执行 SQL 语句时，直到运行时才检查 SQL 语句的错误。即使 SQL 语句无效，包含 SQL 命令的 JDBC 程序也将编译。

43.1 使用 JDBC 类

JDBC 在 Oracle 中通过 Oracle 提供的、名称以 oracle.sql 开头的一组类来实现，而 JDK 提供的标准 JDBC 类以 java.sql 开头。JdbcCheckup.java 类(如下面的程序清单所示)为 JDBC 程序设计的初学者提供了很好的指导。JdbcCheckup.java 类假定用户使用 Oracle Net 连接到数据库。

注意：
下面的代码由 Oracle 提供。按用户自己的编程标准实现这些命令可能会有所不同，如调用 System.out.println 来代替 System.out.print，或者把花括号放在不同的位置。

```
/*
 * This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information. It will select
 * "Hello World" from the database.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// We import java.io to be able to read from the command line
import java.io.*;

import oracle.jdbc.pool.OracleDataSource;
```

```
class JdbcCheckup
{
  public static void main (String args [])
       throws SQLException, IOException
  {
    // Prompt the user for connect information
    System.out.println ("Please enter information to test connection to the
database");
    String user;
    String password;
    String database;

    user = readEntry ("user: ");
    int slash_index = user.indexOf ('/');
    if (slash_index != -1)
    {
      password = user.substring (slash_index + 1);
      user = user.substring (0, slash_index);
    }
    else
      password = readEntry ("password: ");
    database = readEntry ("database (a TNSNAME entry, name-value pairs): ");

    // Create an OracleDataSource and set URL
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:oci8:" + user + "/" + password + "@" + database);

    System.out.print ("Connecting to the database...");
    System.out.flush ();

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.

    System.out.println ("Connecting...");

    Connection conn = ods.getConnection();
    System.out.println ("connected.");

    // Create a statement
    Statement stmt = conn.createStatement ();

    // Do the SQL "Hello World" thing
    ResultSet rset = stmt.executeQuery ("select 'Hello World' from dual");

    while (rset.next ())
      System.out.println (rset.getString (1));

    System.out.println ("Your JDBC installation is correct.");

    // close the resultSet
    rset.close();

    // Close the statement
    stmt.close();

    // Close the connection
    conn.close();
  }
```

```
// Utility function to read a line from standard input
static String readEntry (String prompt)
{
  try
  {
    StringBuffer buffer = new StringBuffer ();
    System.out.print (prompt);
    System.out.flush ();
    int c = System.in.read ();
    while (c != '\n' && c != -1)
    {
      buffer.append ((char)c);
      c = System.in.read ();
    }
    return buffer.toString ().trim ();
  }
  catch (IOException e)
  {
    return "";
  }
 }
}
```

注意：
产品化的应用程序比这个简短的示例包括更多的异常处理部分。关于异常处理的示例，请参阅第 42 章。

首先，该脚本导入 Sun Microsystems 提供的 java.sql 类：

```
import java.sql.*;
```

然后导入 Oracle 提供的数据源驱动程序：

```
import oracle.jdbc.pool.OracleDataSource;
```

在提示用户输入信息时，调用 readEntry 函数来给 user 变量赋值：

```
user = readEntry ("user: ");
```

readEntry 函数该类的后面定义，开头部分如下所示：

```
// Utility function to read a line from standard input
static String readEntry (String prompt)
```

一旦输入连接数据，就会基于 OracleDataSource 定义创建连接对象(connection object)，以维持数据库会话的状态。下面的程序清单中的 getConnection 调用启动此连接：

```
Connection conn = ods.getConnection();
```

如果连接期间发生错误，通常就是这一步出错。不正确的用户名/口令组合、不匹配的驱动程序集或安装和配置 Oracle Net 故障都会导致这一步失败。

默认情况下，连接对象在 autocommit 开启的情况下创建——每项事务都自动提交。要更改此设置，可使用下面的命令(如本示例中，假定该连接对象被命名为 conn)：

```
conn.setAutoCommit(false);
```

接着，该类创建一条语句，执行一个硬编码的查询，并输出结果：

```
    // Create a statement
    Statement stmt = conn.createStatement ();

    // Do the SQL "Hello World" thing
    ResultSet rset = stmt.executeQuery ("select 'Hello World' from dual");

    while (rset.next ())
      System.out.println (rset.getString (1));
```

然后，关闭结果集、语句和连接，程序完成。请注意，在执行该语句时，有两个步骤：首先调用 createStatement 方法创建它，然后利用 executeQuery 方法执行它。可以用 execute 代替 executeQuery 方法，如果运行 insert、update 或 delete 等 SQL 语句，则可用 executeUpdate 代替 executeQuery 方法。如果选择列值(而不仅仅是 1 个文本字符串)，则应当在输出命令中指出该列值，如下面的程序清单所示：

```
ResultSet rset =stmt.executeQuery("select User from dual");
while (rset.next ()) {
   System.out.println (rset.getString ("USER"));
```

43.2 使用 JDBC 进行数据操作

下面来梳理一下迄今为止已经介绍过的内容，即本章的基本连接语法和第 42 章介绍的 Java 类。本节的示例将查询 RADIUS_VALS 表，计算对应每个值的面积，并将这些值插入 AREAS 表中。因此，需要使用与游标 FOR 循环等价的 Java 命令，并支持可执行命令和 insert。

为实现该示例，将 3 条记录放入 RADIUS_VALS 表中，并删除 AREAS 表的所有行：

```
delete from RADIUS_VALS;
insert into RADIUS_VALS(Radius) values (3);
insert into RADIUS_VALS(Radius) values (4);
insert into RADIUS_VALS(Radius) values (10);
delete from AREAS;
commit;
```

在下面的程序清单中，JdbcCircle.java 包含来自 JdbcCheckup.java 的连接组件。在 RetrieveRadius 部分开始运行用于计算圆面积的可执行命令，如程序清单中粗体部分所示：

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
// We import java.io to be able to read from the command line
import java.io.*;

import oracle.jdbc.pool.OracleDataSource;

class JdbcCircle {

  public static void main (String args [])
```

```
    throws SQLException, IOException {

// Prompt the user for connect information
System.out.println ("Please enter information for connection");
String user;
String password;
String database;

user = readEntry ("user: ");
int slash_index = user.indexOf ('/');
if (slash_index != -1) {

  password = user.substring (slash_index + 1);
  user = user.substring (0, slash_index);
}
else {
  password = readEntry ("password: ");
}
database = readEntry ("database (a TNSNAME entry): ");
System.out.println ("Connecting to the database...");

// Create an OracleDataSource and set URL
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci8:" + user + "/" + password + "@" + database);
System.out.flush ();

// Connect to the database
// You can put a database name after the @ sign in the connection URL.

Connection conn = ods.getConnection();

System.out.println ("connected.");

// Create a statement
Statement stmt = conn.createStatement ();

// RetrieveRadius
ResultSet rset =stmt.executeQuery("select Radius from RADIUS_VALS");

while (rset.next ()) {
  // if you wish to print the values:
  // System.out.println (rset.getInt ("RADIUS"));
  int radInput = rset.getInt ("RADIUS");

  // Retrieve Radius value, calculate area:
  double result = area(radInput);

  // insert the value into AREAS
  String sql = "insert into AREAS values ("+radInput+","+result+ ")";
  // If you want to print the SQL statement:
  // System.out.println(sql);

  // Create a statement for the insert:
  Statement insArea = conn.createStatement();
  // Execute the insert:
  boolean insertResult = insArea.execute(sql);
}

// close the resultSet
```

```
    rset.close();
    // Close the statement
    stmt.close();
    // Close the connection
    conn.close();
  }

  // Utility function to calculate the area
  public static double area (int rad) {
    double pi=3.1415927;
    double areacircle=pi*rad*rad;
    return areacircle;
  }

  // Utility function to read a line from standard input
  static String readEntry (String prompt) {

    try {
      StringBuffer buffer = new StringBuffer ();
      System.out.print (prompt);
      System.out.flush ();
      int c = System.in.read ();
      while (c != '\n' && c != -1) {

        buffer.append ((char)c);
        c = System.in.read ();
      }
      return buffer.toString ().trim ();
    }
    catch (IOException e) {
      return "";
    }
  }
}
```

在 RetrieveRadius 部分，要执行的查询被传递给 executeQuery 方法。查询的结果存储在变量 rset 中。

```
// RetrieveRadius
  ResultSet rset =stmt.executeQuery("select Radius from RADIUS_VALS");
```

如第 42 章所述，可以使用 Java 中的 while 循环模拟 PL/SQL 的游标 FOR 循环的功能。RetrieveRadius 部分中的以下代码使用 getInt 方法从结果集中检索整型 Radius 的值。如果此值为字符串，则使用 getString 方法。

```
while (rset.next ()) {
  // if you wish to print the values:
  // System.out.println (rset.getInt ("RADIUS"));
  int radInput = rset.getInt ("RADIUS");
```

在前面程序清单中被赋值的 radInput 变量，现在可用作面积计算的输入。面积计算通过第 42 章给出的代码示例的实用程序函数(请参阅完整的 JdbcCircle 程序清单)来完成。

```
// Retrieve Radius value, calculate area:
  double result = area(radInput);
```

这样，在填充 AREAS 表时，就可以使用半径值(radInput 变量)和面积值(result 变量)。下面构建 insert 语句来完成插入功能。首先，使用 Java 中的字符串连接函数连接文本和变量值。确保生成的命令串的末尾不含分号。

```
// insert the value into AREAS
String sql = "insert into AREAS values ("+radInput+","+result+ ")";
// If you want to print the SQL statement:
// System.out.println(sql);
```

接着，创建一条语句并执行 insert 命令(通过调用包含该命令文本的 sql 变量)：

```
// Create a statement for the insert:
Statement insArea = conn.createStatement();
// Execute the insert:
boolean insertResult = insArea.execute(sql);
```

它是否起作用呢？编译并执行 JdbcCircle.java 文件：

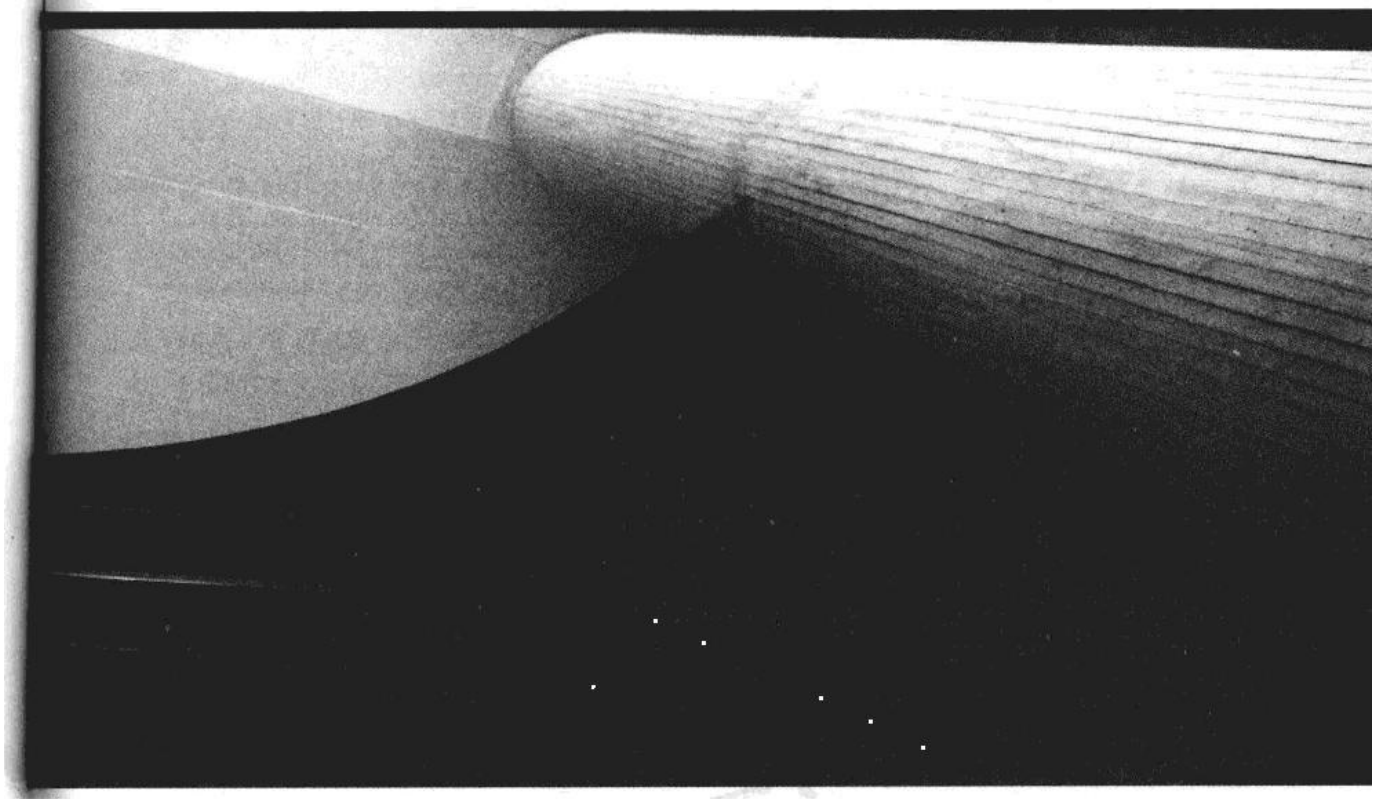
```
javac JdbcCircle.java
java JdbcCircle
```

提示时，输入表 RADIUS_VALS 表和 AREAS 表的所有者的用户名和口令。程序完成后，检查 AREAS 表：

```
select * from AREAS
 order by Radius;

    RADIUS       AREA
---------- ----------
         3      28.27
         4      50.27
        10     314.16
```

虽然该示例基于简单的圆面积计算，但它说明了 JDBC 开发的关键概念：连接到数据库、检索数据、实现游标 FOR 循环、给变量赋值、生成动态 SQL 语句以及执行 DML 等。

JDBC 支持可扩展的编程功能，包括语句缓存、连接缓存、连接故障转移、分布式事务、支持 LOB 等等。关于 JDBC 用法的更详细信息，请参阅 *Oracle Database JDBC Development's Guide and Reference*。



第44章

Java存储过程

用户可以编写调用Java类的存储过程、触发器、对象类型方法以及函数。本章将介绍Java过程的示例集和执行过程所需的命令。为了更好地理解本章，首先应当熟悉Java结构(参见第42章)和Java类、JDBC的用法(参见第43章)。其次还应当熟悉存储过程的基本语法和用法(参见第35章)。

为了重点讨论Java类的数据库访问操作，本章结合简要的示例进行介绍。下面的类命名为BookshelfDML，包含向BOOKSHELF表中插入记录和从该表中删除记录的方法。在该示例中，不进行面积的计算，而所有必需的值都在insert操作期间提供。下面的程序清单显示了BookshelfDML.java类文件：

```
import java.sql.*;
import java.io.*;
```

```
import oracle.jdbc.*;

public class BookshelfDML {
  public static void insertBookshelf (String title, String publisher,
    String categoryname, String rating) throws SQLException {
      String sql = "INSERT INTO BOOKSHELF VALUES (?,?,?,?)";
      try {
        Connection conn =
          DriverManager.getConnection("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, title);
        pstmt.setString(2, publisher);
        pstmt.setString(3, categoryname);
        pstmt.setString(4, rating);
        pstmt.executeUpdate();
        pstmt.close();
      } catch (SQLException e) {System.err.println(e.getMessage());}
  }

public static void updateBookshelf (String rating,
    String title) throws SQLException {
      String sql = "UPDATE BOOKSHELF SET Rating = ? " +
          "WHERE Title = ? ";
      try {
        Connection conn =
          DriverManager.getConnection("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, rating);
        pstmt.setString(2, title);
        pstmt.executeUpdate();
        pstmt.close();
      } catch (SQLException e) {System.err.println(e.getMessage());}
  }

public static void deleteBookshelf (String title) throws SQLException {
      String sql = "DELETE FROM BOOKSHELF WHERE TITLE = ?";
      try {
        Connection conn =
          DriverManager.getConnection("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, title);
        pstmt.executeUpdate();
        pstmt.close();
      } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

BookshelfDML.java 文件定义 BookshelfDML 类。在 BookshelfDML 类中，有几个单独的方法来处理 insert、update 和 delete。在每个方法的说明中，都指定了与表 BOOKSHELF 相关联的数据类型(在本示例中，它们都是 VARCHAR2 数据类型)。

这些参数使用 Oracle 的类库提供的一部分数据类型。如果使用标准的 Java 数据类型，则在类的执行期间将遇到与数据类型相关的问题。

插入方法的下一部分将值插入 BOOKSHELF 表中：

```
String sql = "INSERT INTO BOOKSHELF VALUES (?,?,?,?)";
```

通过 setString 定义 4 个变量(如果为整数，则用 setInt)：

```
pstmt.setString(1, title);
pstmt.setString(2, publisher);
pstmt.setString(3, categoryname);
pstmt.setString(4, rating);
```

注意，变量名区分大小写——必须和前面声明的变量名严格匹配。

删除方法在结构上与插入方法非常相似。它的 SQL 语句是：

```
String sql = "DELETE FROM BOOKSHELF WHERE TITLE = ?";
```

注意，与第 43 章的示例不同，类的组件不创建数据库连接。用来调用该类的连接将为类的方法提供连接环境。

通过 Java 编译器处理该类，可以验证该类是否正确编译。注意这一步不能验证 SQL 的有效性：

```
javac BookshelfDML.java
```

这一步应当成功完成。如果在编译步骤期间遇到错误，则应当检查环境配置(参照 Oracle 软件提供的 README 文件)和类文件。

44.1　将类加载到数据库中

可以使用 loadjava 实用程序将类加载到数据库中。loadjava 实用程序的语法如下：

```
loadjava {-user | -u} username/password[@database]
[-option_name -option_name ...] filename filename ...
```

loadjava 实用程序的可用选项如表 44-1 所示。

表 44-1　loadjava 实用程序的选项

参　数	说　明
filenames	可以指定任意数量和组合的.java、.class、.sqlj、.ser、.jar、.zip 和资源文件名参数

-proxy host:port	如果不能物理上访问服务器主机或 loadjava 客户端来加载类、资源和 java 源文件，则可以使用 HTTP URL 与 loadjava 工具指定 JAR、类文件或资源，然后从远程服务器上加载类。host 是主机名或地址，port 是代理服务器正在使用的端口。URL 实现必须使用 host 和 port，以便 loadjava 工具能够确定要加载的文件的类型——即 JAR、类、资源或 Java 源文件。例如：
	`loadjava -u scott -r -v -proxy proxy_server:1020`
	`http://my.server.com/this/is /the/path/my.jar`
	`Password: password`
	当在服务器内使用 URL 支持时，您应该具有访问远程源文件的合适 Java 权限。URL 支持也包含 ftp: URL 和 file: URL

(续表)

参 数	说 明
-casesensitivepub	创建区分大小写的名称。除非名称已经都是大写字母，否则在 PL/SQL 中通常需要用引号将名称引起来
-cleargrants	-grant 选项导致 loadjava 工具将 EXECUTE 权限授予类、源文件和资源，但不撤消任何权限。如果指定-cleargrants，则 loadjava 工具在将执行权限授予-grant 操作数指定的用户和角色之前，loadjava 工具将撤消任何已授予的执行权限。例如，如果只想将执行权限授予 SCOTT，则正确的选项是： `-grant SCOTT -cleargrants`
-debug	启用 SQL 登录
-definer	默认情况下， 类模式对象用其调用者的权限运行。而此选项在类上授予定义者权限。这一选项在概念上与 UNIX 的 setuid 工具相似
-dirprefix prefix	对于以 prefix 前缀开头的任何文件或 JAR 条目，在确定模式对象的名称之前将从名称中删除此前缀。对于类和源文件，模式对象的名称由它们的内容确定。因此，此选项只对资源有影响
-encoding	为编译器识别源文件编码，重写 JAVA$OPTIONS 中的匹配值(如果有匹配值)。值与 javac-encoding 选项的值是相同的。如果未在命令行或 JAVA$OPTIONS 中指定编码，则假设编码是 System.getProperty("file.encoding")语句返回的值。此选项只在加载源文件时才有用
-fileout file	把所有消息显示到指定的文件中
-force	强制加载文件，即使文件与摘要表条目相匹配
-genmissing	确定要求 loadjava 工具处理的类引用了哪些类和方法。在数据库或文件参数中未找到的任何类都称为缺失类。此选项为包含所有被引用方法的缺失类生成虚定义。然后将生成的类加载到数据库中。此处理过程发生在解析类之前。 因为从源文件检测引用比从类文件检测引用复杂，同时因为源文件一般不用于分布式库，所以 loadjava 工具不尝试为源文件执行这种处理。在缺失类中加载的模式不是-user 选项指定的模式，即使在其他模式中创建引用类也如此。标记创建的类，以便工具可以识别它们。特别需要说明的是，这很有必要，以便验证者可以识别生成的类
-genmissingjar jar_file	此选项执行的操作与-genmissing 相同。此外，它还创建一个 JAR 文件 jar_file，此文件包含生成的任何类的定义
-grant	将被加载类上的 EXECUTE 权限授予列出的用户。可以指定任何数量和组合的用户名，中间用逗号分隔开，但无空格。 授予在另一个模式中的对象上的 EXECUTE 权限要求：原始的 CREATE PROCEDURE 权限是通过 WITH GRANT 选项授予的。 **注意：** -grant 是一个累积选项。用户被添加到具有 EXECUTE 权限的列表中。要删除权限，使用-cleargrants 选项。模式名应该是大写字母

(续表)

参　数	说　明
-help	显示关于如何使用 loadjava 工具及其选项的帮助消息
-javaresource	不解压缩 JAR 文件和加载其中的每个类，此参数将整个 JAR 文件作为一个资源加载到模式中
-noaction	不对文件进行任何操作。操作包括创建模式对象和授予可执行权限等。此参数一般用在选项文件中，用来阻止在 JAR 中创建特定的类。当此参数用于命令行时，除非在选项文件中被重写，否则会使 loadjava 工具忽略所有文件，但仍会检查 JAR 文件，以确定它们是否包含 META-INF/loadjava 选项条目。如果包含，则处理选项文件。选项文件中的-action 选项将重写命令行中指定的-noaction 选项
-norecursivejars	将其他 JAR 文件中包含的 JAR 文件视为资源。这是默认行为。此选项用来覆盖-recursivejars 选项
-nosynonym	不创建类的公共同义词。这是默认行为。此选项重写-synonym 选项
-nousage	如果未指定任何选项或指定-help 选项，则不显示使用消息
-noverify	使已加载的类不进行字节码验证。为了使用这个选项，必须授予 oracle.aurora.security.JServerPermission(Verifier)权限。此外，该选项必须和-resolve 一起使用
-oci | -oci8	指示 loadjava 工具与使用 JDBC OCI 驱动程序的数据库通信。-oci 与-thin 是互相排斥的。如果未指定是-oci 还是-oci8，则默认使用-oci。选择-oci 隐含了-user 值的语法。不需要提供 URL
-publish package	程序包由 loadjava 工具创建或替换。合格方法的包装器将在此程序包中定义。通过使用选项文件，调用一次 loadjava 工具就可以创建多个程序包。每个程序包都将经历与方法相同的名称转换
-pubmain number	适用于只有一个参数的方法的一个特例，此参数的类型是 java.lang.String。将创建 SQL 过程或函数的多个变体，每个变体接受不同数量的 VARCHAR 类型的参数。特别要说明的是，创建变体要使用所有的参数，并且包括 number。number 的默认值是 3。此选项适用于 main 方法和恰好有一个 java.lang.String 类型参数的任何方法。
-recursivejars	一般来说，如果 loadjava 工具在 JAR 中遇到一个具有.jar 扩展名的条目，则会作为资源加载该条目。如果指定了此选项，则 loadjava 工具像处理顶层 JAR 文件一样来处理包含的 JAR 文件。也就是说，loadjava 工具将读取 JAR 文件的条目，并加载类、源文件和资源
-resolve	命令行中的所有类加载之后，编译并根据需要解析类中的外部引用。如果不指定该选项，则 loadjava 工具只加载文件，但不编译和解析文件
-resolver	显式指定与新加载的类绑定在一起的解析程序说明。如果未指定该选项，则使用默认的解析程序说明，它包含当前用户的模式和 PUBLIC

(续表)

参 数	说 明
-resolveonly	使 loadjava 工具跳过最初的创建步骤。仍执行授权、解析、创建同义词等
-schema	指定创建模式对象所在的模式。如果未指定此选项，则使用-user 模式。要在不属于自己的模式中创建模式对象，必须具有 CREATE PROCEDURE 或 CREATE ANY PROCEDURE 权限。必须具有 CREATE TABLE 或 CREATE ANY TABLE 权限。最后，必须具有类的 JServerPermission loadLibraryInClass
-stdout	使输出指向标准输出文件而不是标准错误文件
-stoponerror	一般来说，当 loadjava 工具处理文件时发生错误，将会发出一条消息，并继续处理其他类。当出现错误时，此选项停止。此外，它报告适用于 Java 对象且包含在类被加载的模式的 USER_ERROR 表中的所有错误，但不报告 ORA-29524 错误。这些错误是在类不能解析时产生的，类不能解析的原因是被引用的类不能解析。因此，这些错误是被引用的类不能解析的副面影响
-synonym	为加载的类创建 PUBLIC 同义词，使它们可访问所加载模式的外部。要指定这一选项，必须拥有 CREATE PUBLIC SYNONYM 权限。如果为源文件指定-synonym，则从源文件编译的类可看作用-synonym 加载的类
-tableschema schema	在指定的模式而不是 Java 文件目的模式下创建 loadjava 工具内部表
-thin	指导 loadjava 工具与使用 JDBC 瘦驱动程序的数据库通信。选择-thin 隐含了-user 值的语法。不需要通过-user 选项指定合适的 URL
-unresolvedok	当与-resolve 一起使用时，该参数忽略未解析的错误
-user	指定用户名、口令和数据库连接字符串。文件将被加载到此数据库实例中
-verbose	在运行时指导 loadjava 工具显示详细的状态消息。使用此选项可以了解 loadjava 工具何时由于匹配摘要表条目而不加载文件
-jarsasdbobjects	指出当前 loadjava 工具命令处理的 JAR 将作为数据库常驻程序 JAR 存储在数据库中
-prependjarnames	与-jarsasdbobjects 选项一起使用。此选项使来自不同 JAR 的具有相同名称的类能够共存于相同的模式中。具体实现方法是将一种形式的 JAR 名作为前缀加在类名前，以生成独一无二的数据库对象名

例如，要将 BookshelfDML.java 文件加载到 Practice 模式中(该模式位于“orcl”实例中)，可以使用以下命令：

```
loadjava -user Practice/practice@orcl.world -verbose BookshelfDML.java
```

注意：

根据具体配置，可能还需要使用-encoding 参数。

将会看到关于把该类加载到数据库中的一系列注释。出现任何错误，加载都不会完成。

注意：
类名不应该与模式中任何已存在的对象名相同。

现在把 AreasDML 类加载到数据库中。在加载过程中，Oracle 将在用户的模式下创建多个表和 Java 对象。

注意：
可以通过 create java [source | class | resource]命令，从本地 BFILE 或 LOB 列手动加载 Java 文件。

如果在单独的服务器上运行数据库，而且还没有为数据库定义服务名，则需要为 loadjava 程序调用指定完整的连接说明。格式为：

```
loadjava -user practice/practice@localhost:1521:orcl -v BookshelfDML.java
```

44.2　如何访问类

既然已经将 BookshelfDML 类加载到了数据库中，就必须创建调用每个方法的过程。下面的程序清单中的 create procedure 命令创建了过程 InsertBookshelfViaJava：

```
create or replace procedure InsertBookshelfViaJava
(Title VARCHAR2, Publisher VARCHAR2,
 CategoryName VARCHAR2, Rating VARCHAR2 )
as
language java
name 'BookshelfDML.insertBookshelf(java.lang.String, java.lang.String,
java.lang.String, java.lang.String)';
/
```

这个简单的过程有 4 个参数——BOOKSHELF 表的 4 列。其 language java 子句告诉 Oracle 该过程正在调用 Java 方法。方法的名称在 name 子句中指定(确保与您使用的大小写一致)。接着指定 Java 方法调用的格式。

一个单独的过程将通过方法调用删除 BOOKSHELF 表中的记录：

```
create or replace procedure DeleteBookshelfViaJava
(Title VARCHAR2 )
as
```

```
language java
name 'BookshelfDML.deleteBookshelf(java.lang.String)';
/
```

第 3 个过程将基于 Title 更新 Rating 值：

```
create or replace procedure UpdateBookshelfViaJava
(Rating VARCHAR2, Title VARCHAR2 )
as
language java
name 'BookshelfDML.updateBookshelf(java.lang.String, java.lang.String)';
/
```

现在，进入 SQL*Plus，并使用 call 命令执行 Java 类：

```
call InsertBookshelfViaJava('TEST','TESTPub','ADULTNF','2');
```

将会看到如下响应：

```
Call completed.
```

除了使用 call，还可以在 PL/SQL 块中执行该过程：

```
begin
  InsertBookshelfViaJava('TEST','TESTPub','ADULTNF','2');
end;
/
```

call 命令将把列值作为参数传递给 InsertBookshelfViaJava 存储过程。它还会根据 name 子句中的说明将列值进行格式化，并把它们传递给 BookshelfDML.insertBookshelf 方法。此方法会使用当前的连接执行其命令，并插入该行。大多数库和类至今还没有加载到数据库中，还有许多步骤需要完成，因此第 1 次运行 Java 存储过程时性能将受到影响，但是后面的执行将会快得多。可以查询 BOOKSHELF 表验证其成功与否：

```
column Title format a10
select * from BOOKSHELF where Title like 'TEST';

TITLE      PUBLISHER          CATEGORYNAME          RA
---------- ------------------ --------------------  --
TEST       TESTPub            ADULTNF               2
```

可以通过 UpdateBookshelfViaJava 过程更新 rating：

```
begin
  UpdateBookshelfViaJava('3','TEST');
end;
/
```

```
select * from BOOKSHELF where Title like 'TEST';

TITLE       PUBLISHER              CATEGORYNAME           RA
----------  --------------------   --------------------   --
TEST        TESTPub                ADULTNF                3
```

调用 DeleteBookshelfViaJava 过程删除记录：

```
begin
  DeleteBookshelfViaJava('TEST');
end;
/

select * from BOOKSHELF where Title like 'TEST';

no rows selected
```

调用 DeleteBookshelfViaJava 将比调用 InsertBookshelfViaJava 完成得快得多，因为 Java 库已经在第 1 次执行 InsertBookshelfViaJava 时已加载了。

如果不接受插入的内容(例如，如果试图将字符类型的字符串插入 NUMBER 数据类型的列中)，则 Oracle 将显示错误。当调试 Java 存储过程时，可以在 Java 代码中调用 System.err.println(参见本章前面的 BookshelfDML.java 程序清单)。为了在 SQL*Plus 会话中显示输出，必须首先执行下面的命令：

```
set serveroutput on
call DBMS_JAVA.SET_OUTPUT(10000);
```

调用 DBMS_JAVA.SET_OUTPUT 存储过程将把 System.out.println 的输出重定向到 DBMS_OUTPUT.PUT_LINE 存储过程。

44.2.1　直接调用 Java 存储过程

Oracle 提供了应用程序编程接口(API)，以支持对静态的 Java 存储过程的直接调用。由于该方法的类可以在 oracle.jpub.reflect 程序包中找到，因此必须将此程序包导入到客户端代码中。API 的 Java 接口如下所示：

```
public class Client
{
  public static String getSignature(Class[]);
  public static Object invoke(Connection, String, String,
                              String, Object[]);
  public static Object invoke(Connection, String, String,
                              Class[], Object[]);
}
```

通过该 API 进行访问时，过程的参数不能为 OUT 或 IN OUT。返回的值都必须为函数结果的一部分。注意：方法调用将使用调用者的权限。关于 API 的详细内容请查看随 Oracle 软件安装程序一起提供的 Java 开发人员的指南。

44.2.2 在何处执行命令

BookshelfDML 示例旨在概述 Java 存储过程的实现。假设现在想使这些过程更复杂，包括更多的表和计算。如前面几章所述，可以在 Java 或 PL/SQL 中执行这些计算。

选择的数据处理平台和语言将会影响性能。一般来说，第 1 次执行 Java 存储过程将会造成性能损失(因为要加载类)，但后面的执行就没有性能损失了。如果重复执行相同的命令，则第 1 次执行存储过程对性能的影响将很微小。

退出并重新登录将会影响性能，但是不如第 1 次执行过程时的影响大。使用 Java 存储过程时，必须知道应用程序和 Oracle 的交互方式。如果用户经常退出系统并重新登录，则在每次登录后第 1 次执行过程时会造成性能损失。此外，数据库启动后第 1 次执行过程会造成严重的性能损失。这些活动对性能的影响必须在应用程序开发的设计阶段就加以考虑。

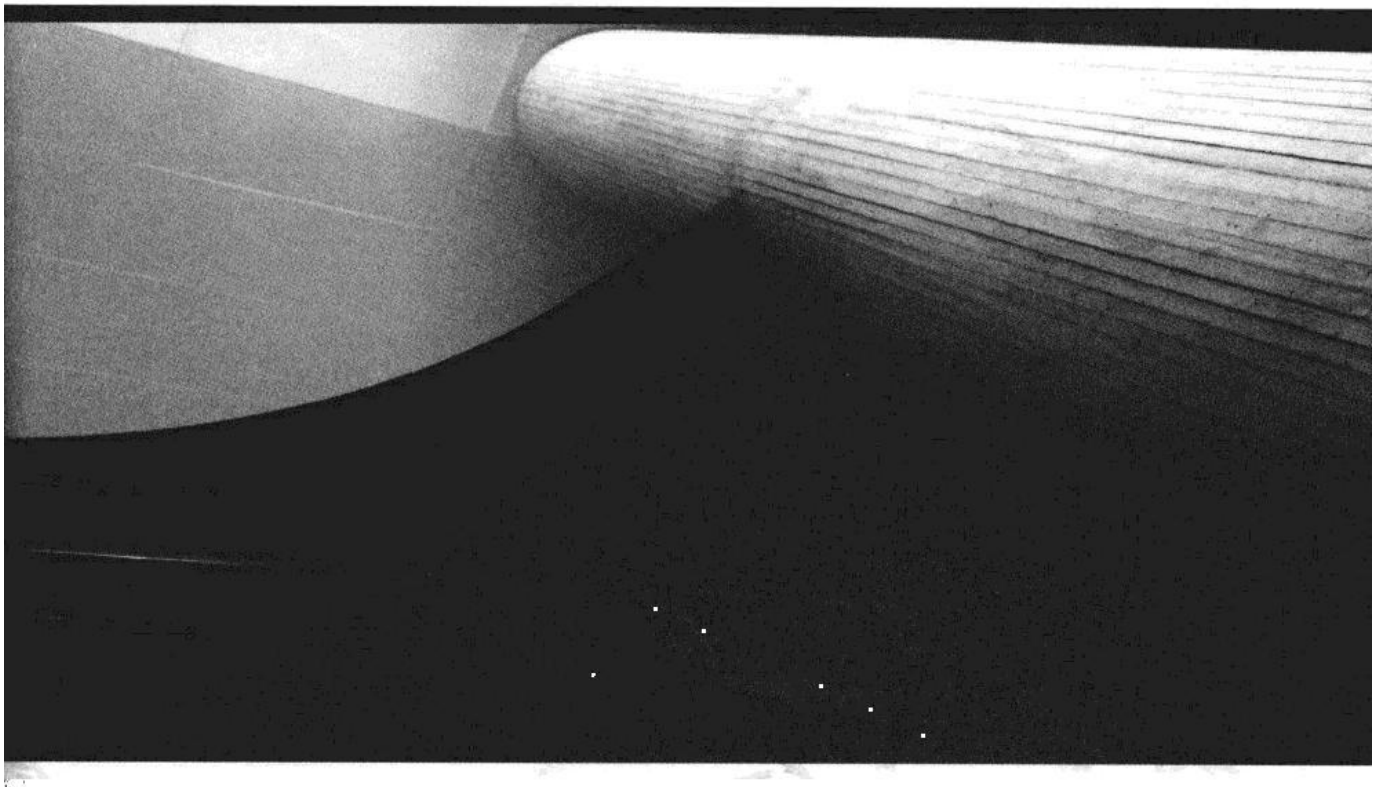
下面的提示可以减少与 Java 存储过程调用有关的性能损失：

- 在数据库启动后执行每个 Java 存储过程。在 BookshelfDML 示例中，这涉及每次启动后插入和删除测试记录。
- 减少短期数据库连接的数量(例如，通过连接池)
- 通过标准 SQL 和 PL/SQL，执行大多数直接数据库操作

Java 是适用于许多计算处理和显示操作的技术。一般情况下，对于数据库连接和数据操作，SQL 和 PL/SQL 比 Java 更有效一些。但是，在 Oracle 内 Java 的性能优越性使其成为 PL/SQL 的一种强有力的替代品。Java 存储过程性能损失的主要原因在于它通过 PL/SQL 包装器(wrapper)所需要的时间。在设计集成 Java、SQL 和 PL/SQL 的应用程序时，应该根据各自的优点来使用这几种语言。

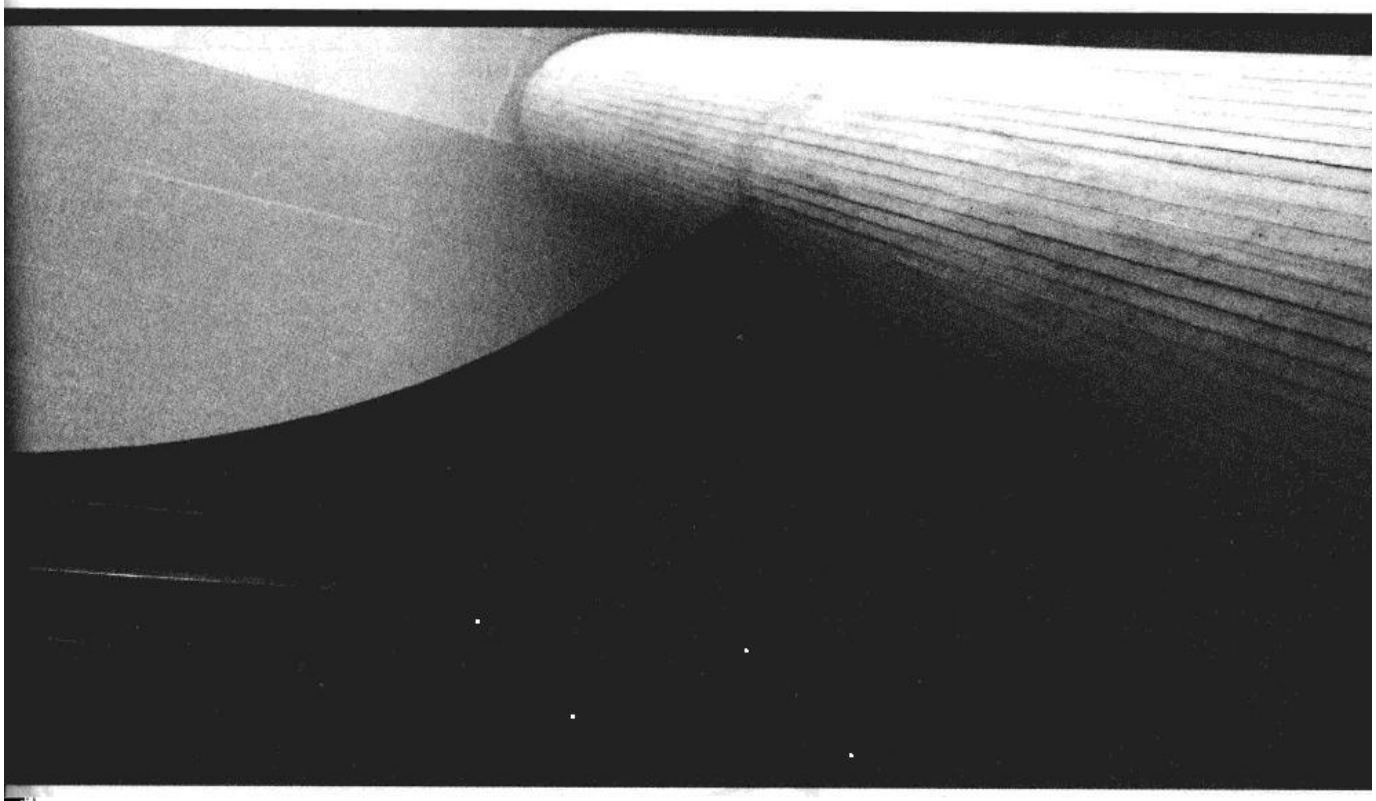
如果代码涉及许多循环和数据的操作，则使用 Java 在性能上比 PL/SQL 更好。使用 Java 的代价(创建 Java 对象与 PL/SQL 头交互)将由实际工作中 Java 所带来的性能改善得到补偿。注意 Oracle Database 11g 中可用的编译过的本地 PL/SQL 将在性能方面为 Java 提供一个强有力的选择。

通常，最好由 PL/SQL 和 SQL 处理那些包含从数据库中选择并操作数据的操作。而用 Java 计算所选的数据，则处理速度会更快一些。在设计应用程序时，必须使用合适的工具来完成相应的功能，以避免不必要的性能损失。



第Ⅶ部分

指　南



第45章

Oracle数据字典指南

Oracle的数据字典存储用来管理数据库中对象的所有信息。尽管数据字典通常由数据库管理员(DBA)管理，但它也是开发人员和最终用户有价值的信息来源。

本章将从最终用户的角度出发介绍数据字典，即Oracle的内部字典。数据字典表和视图不按字母顺序排列，而是按其功能(表、安全性等)进行分组。这样安排是为了引导用户快速找到所需要的信息。本章给出了最常用的数据字典视图及其用法的示例。

根据使用的Oracle配置选项，有些分组对用户的数据库将不起作用。下面先列出最常用的视图。本章所用的视图按顺序分组如下：

- Oracle Database 11g中引入的新视图
- 导航图：DICTIONARY(DICT)和DICT_COLUMNS

- 可选择的内容：表(和列)、视图、同义词和序列
- 回收站：USER_RECYCLEBIN 和 DBA_RECYCLEBIN
- 约束和注释
- 索引和群集
- 抽象数据类型、与 ORDBMS 相关的结构和 LOB
- 数据库链接和物化视图
- 触发器、过程、函数和程序包
- 维度
- 空间分配和使用，包括分区和子分区
- 用户和权限
- 角色
- 审核
- 其他功能
- 监控：V$动态性能表

注意：
随着新特性的增加，每个 Oracle 的新版本都可能会修改数据字典。这里并不打算详尽地列出所有数据字典视图，而只列出用户查询最经常访问的那些数据字典视图。

45.1 关于名称的说明

除了某些特殊情况外，Oracle 数据字典中对象的名称都以下面 3 个前缀开头：“USER”、“ALL”或“DBA”。“USER”视图中的记录通常显示执行查询的账户所拥有对象的信息。“ALL”视图中的记录包含 USER 记录和其权限已经授予 PUBLIC 或用户的对象的信息。“DBA”视图包含所有的数据库对象，而不论所有者是谁。对大多数数据库对象来说，“USER”、“ALL”和“DBA”视图都是可用的。

考虑到用户的实际需求，我们将把重点放在“USER”视图或所有用户都可以访问的那些视图中。ALL 和 DBA 视图将在用到它们的时候介绍。

45.2 Oracle Database 11g 中引入的新视图

对于熟悉 Oracle 10g 数据字典的用户，表 45-1 列出了 Oracle Database 11g 中引入的新视图。

表 45-1　Oracle Database 11g 中的新的数据字典视图

ALL_视图	DBA_视图	USER_视图
ALL_ASSEMBLIES	DBA_ADDM_FDG_BREAKDOWN	USER_ADDM_FDG_BREAKDOWN
	DBA_ADDM_FINDINGS	USER_ADDM_FINDINGS
	DBA_ADDM_INSTANCES	USER_ADDM_INSTANCES
	DBA_ADDM_SYSTEM_DIRECTIVES	
	DBA_ADDM_TASK_DIRECTIVES	USER_ADDM_TASK_DIRECTIVES
	DBA_ADDM_TASKS	USER_ADDM_TASKS
	DBA_ADVISOR_DIR_DEFINITIONS	
	DBA_ADVISOR_DIR_INSTANCES	
	DBA_ADVISOR_DIR_TASK_INST	USER_ADVISOR_DIR_TASK_INST
	DBA_ADVISOR_EXEC_PARAMETERS	USER_ADVISOR_EXEC_PARAMETERS
	DBA_ADVISOR_EXECUTION_TYPES	
	DBA_ADVISOR_EXECUTIONS	USER_ADVISOR_EXECUTIONS
	DBA_ADVISOR_FDG_BREAKDOWN	USER_ADVISOR_FDG_BREAKDOWN
	DBA_ADVISOR_FINDING_NAMES	
	DBA_ADVISOR_SQLA_COLVOL	USER_ADVISOR_SQLA_COLVOL
	DBA_ADVISOR_SQLA_TABLES	USER_ADVISOR_SQLA_TABLES
	DBA_ADVISOR_SQLA_TABVOL	USER_ADVISOR_SQLA_TABVOL
	DBA_ADVISOR_SQLA_WK_SUM	USER_ADVISOR_SQLA_WK_SUM
	DBA_ADVISOR_SQLPLANS	USER_ADVISOR_SQLPLANS
	DBA_ADVISOR_SQLSTATS	USER_ADVISOR_SQLSTATS
	DBA_ARGUMENTS	
	DBA_ASSEMBLIES	USER_ASSEMBLIES
	DBA_AUTOTASK_CLIENT	
	DBA_AUTOTASK_CLIENT_HISTORY	
	DBA_AUTOTASK_CLIENT_JOB	
	DBA_AUTOTASK_OPERATION	
	DBA_AUTOTASK_SCHEDULE	
	DBA_AUTOTASK_TASK	
	DBA_AUTOTASK_WINDOW_CLIENTS	
	DBA_AUTOTASK_WINDOW_HISTORY	
ALL_CHANGE_PROPAGATION_SETS		
ALL_CHANGE_PROPAGATIONS		
ALL_CHANGE_SETS		
ALL_CHANGE_SOURCES		
ALL_CHANGE_TABLES		
ALL_COL_PENDING_STATS	DBA_COL_PENDING_STATS	USER_COL_PENDING_STATS
	DBA_COMPARISON	USER_COMPARISON
	DBA_COMPARISON_COLUMNS	USER_COMPARISON_COLUMNS
	DBA_COMPARISON_ROW_DIF	USER_COMPARISON_ROW_DIF
	DBA_COMPARISON_SCAN	USER_COMPARISON_SCAN
	DBA_COMPARISON_SCAN_SUMMARY	USER_COMPARISON_SCAN_SUMMARY
	DBA_COMPARISON_SCAN_VALUES	USER_COMPARISON_SCAN_VALUES
	DBA_CPOOL_INFO	
	DBA_CQ_NOTIFICATION_QUERIES	USER_CQ_NOTIFICATION_QUERIES
ALL_CUBE_ATTR_VISIBILITY	DBA_CUBE_ATTR_VISIBILITY	USER_CUBE_ATTR_VISIBILITY

（续表）

ALL_视图	DBA_视图	USER_视图
ALL_CUBE_ATTRIBUTES	DBA_CUBE_ATTRIBUTES	USER_CUBE_ATTRIBUTES
ALL_CUBE_BUILD_PROCESSES	DBA_CUBE_BUILD_PROCESSES	USER_CUBE_BUILD_PROCESSES
ALL_CUBE_CALCULATED_MEMBERS	DBA_CUBE_CALCULATED_MEMBERS	USER_CUBE_CALCULATED_MEMBERS
ALL_CUBE_DIM_LEVELS	DBA_CUBE_DIM_LEVELS	USER_CUBE_DIM_LEVELS
ALL_CUBE_DIM_MODELS	DBA_CUBE_DIM_MODELS	USER_CUBE_DIM_MODELS
ALL_CUBE_DIM_VIEW_COLUMNS	DBA_CUBE_DIM_VIEW_COLUMNS	USER_CUBE_DIM_VIEW_COLUMNS
ALL_CUBE_DIM_VIEWS	DBA_CUBE_DIM_VIEWS	USER_CUBE_DIM_VIEWS
ALL_CUBE_DIMENSIONALITY	DBA_CUBE_DIMENSIONALITY	USER_CUBE_DIMENSIONALITY
ALL_CUBE_DIMENSIONS	DBA_CUBE_DIMENSIONS	USER_CUBE_DIMENSIONS
ALL_CUBE_HIER_LEVELS	DBA_CUBE_HIER_LEVELS	USER_CUBE_HIER_LEVELS
ALL_CUBE_HIER_VIEW_COLUMNS	DBA_CUBE_HIER_VIEW_COLUMNS	USER_CUBE_HIER_VIEW_COLUMNS
ALL_CUBE_HIER_VIEWS	DBA_CUBE_HIER_VIEWS	USER_CUBE_HIER_VIEWS
ALL_CUBE_HIERARCHIES	DBA_CUBE_HIERARCHIES	USER_CUBE_HIERARCHIES
ALL_CUBE_MEASURES	DBA_CUBE_MEASURES	USER_CUBE_MEASURES
ALL_CUBE_VIEW_COLUMNS	DBA_CUBE_VIEW_COLUMNS	USER_CUBE_VIEW_COLUMNS
ALL_CUBE_VIEWS	DBA_CUBE_VIEWS	USER_CUBE_VIEWS
ALL_CUBES	DBA_CUBES	USER_CUBES
	DBA_EPG_DAD_AUTHORIZATION	USER_EPG_DAD_AUTHORIZATION
	DBA_FLASHBACK_ARCHIVE	USER_FLASHBACK_ARCHIVE
	DBA_FLASHBACK_ARCHIVE_TABLES	USER_FLASHBACK_ARCHIVE_TABLES
	DBA_FLASHBACK_ARCHIVE_TS	
	DBA_FLASHBACK_TXN_REPORT	USER_FLASHBACK_TXN_REPORT
	DBA_FLASHBACK_TXN_STATE	USER_FLASHBACK_TXN_STATE
	DBA_HIST_BASELINE_DETAILS	
	DBA_HIST_BASELINE_METADATA	
	DBA_HIST_BASELINE_TEMPLATE	
	DBA_HIST_CLUSTER_INTERCON	
	DBA_HIST_COLORED_SQL	
	DBA_HIST_EVENT_HISTOGRAM	
	DBA_HIST_IC_CLIENT_STATS	
	DBA_HIST_IC_DEVICE_STATS	
	DBA_HIST_INTERCONNECT_PINGS	
	DBA_HIST_IOSTAT_FILETYPE	
	DBA_HIST_IOSTAT_FILETYPE_NAME	
	DBA_HIST_IOSTAT_FUNCTION	
	DBA_HIST_IOSTAT_FUNCTION_NAME	
	DBA_HIST_MEM_DYNAMIC_COMP	
	DBA_HIST_MEMORY_RESIZE_OPS	
	DBA_HIST_MEMORY_TARGET_ADVICE	
	DBA_HIST_MUTEX_SLEEP	
	DBA_HIST_PERSISTENT_QUEUES	
	DBA_HIST_PERSISTENT_SUBS	
	DBA_HIST_RSRC_CONSUMER_GROUP	
	DBA_HIST_RSRC_PLAN	

（续表）

ALL_视图	DBA_视图	USER_视图
ALL_IDENTIFIERS	DBA_IDENTIFIERS	USER_IDENTIFIERS
ALL_IND_PENDING_STATS	DBA_IND_PENDING_STATS	USER_IND_PENDING_STATS
	DBA_INVALID_OBJECTS	
ALL_JAVA_COMPILER_OPTIONS	DBA_JAVA_COMPILER_OPTIONS	USER_JAVA_COMPILER_OPTIONS
	DBA_LOGSTDBY_UNSUPPORTED_TABLE	
ALL_MEASURE_FOLDER_CONTENTS	DBA_MEASURE_FOLDER_CONTENTS	USER_MEASURE_FOLDER_CONTENTS
ALL_MEASURE_FOLDERS	DBA_MEASURE_FOLDERS	USER_MEASURE_FOLDERS
ALL_MINING_MODEL_ATTRIBUTES	DBA_MINING_MODEL_ATTRIBUTES	USER_MINING_MODEL_ATTRIBUTES
ALL_MINING_MODEL_SETTINGS	DBA_MINING_MODEL_SETTINGS	USER_MINING_MODEL_SETTINGS
ALL_MINING_MODELS	DBA_MINING_MODELS	USER_MINING_MODELS
ALL_MVIEW_DETAIL_PARTITION	DBA_MVIEW_DETAIL_PARTITION	USER_MVIEW_DETAIL_PARTITION
ALL_MVIEW_DETAIL_SUBPARTITION	DBA_MVIEW_DETAIL_SUBPARTITION	USER_MVIEW_DETAIL_SUBPARTITION
	DBA_NETWORK_ACL_PRIVILEGES	USER_NETWORK_ACL_PRIVILEGES
	DBA_NETWORK_ACLS	
	DBA_OLDIMAGE_COLUMNS	USER_OLDIMAGE_COLUMNS
	DBA_REGISTRY_DATABASE	
	DBA_REGISTRY_DEPENDENCIES	
	DBA_REGISTRY_PROGRESS	
	DBA_RSRC_CAPABILITY	
	DBA_RSRC_CATEGORIES	
	DBA_RSRC_INSTANCE_CAPABILITY	
	DBA_RSRC_IO_CALIBRATE	
	DBA_RSRC_STORAGE_POOL_MAPPING	
ALL_SCHEDULER_CREDENTIALS	DBA_SCHEDULER_CREDENTIALS	USER_SCHEDULER_CREDENTIALS
	DBA_SCHEDULER_JOB_ROLES	
ALL_SCHEDULER_REMOTE_DATABASES	DBA_SCHEDULER_REMOTE_DATABASES	
ALL_SCHEDULER_REMOTE_JOBSTATE	DBA_SCHEDULER_REMOTE_JOBSTATE	USER_SCHEDULER_REMOTE_JOBSTATE
	DBA_SQL_MANAGEMENT_CONFIG	
	DBA_SQL_PATCHES	
	DBA_SQL_PLAN_BASELINES	
ALL_STAT_EXTENSIONS	DBA_STAT_EXTENSIONS	USER_STAT_EXTENSIONS
ALL_STREAMS_COLUMNS	DBA_STREAMS_COLUMNS	
	DBA_STREAMS_TP_COMPONENT	
	DBA_STREAMS_TP_COMPONENT_LINK	
	DBA_STREAMS_TP_COMPONENT_STAT	
	DBA_STREAMS_TP_DATABASE	
	DBA_STREAMS_TP_PATH_BOTTLENECK	
	DBA_STREAMS_TP_PATH_STAT	
	DBA_SUBSCR_REGISTRATIONS	USER_SUBSCR_REGISTRATIONS
ALL_SYNC_CAPTURE	DBA_SYNC_CAPTURE	
ALL_SYNC_CAPTURE_PREPARED_TABS	DBA_SYNC_CAPTURE_PREPARED_TABS	
ALL_SYNC_CAPTURE_TABLES	DBA_SYNC_CAPTURE_TABLES	

（续表）

ALL_视图	DBA_视图	USER_视图
ALL_TAB_HISTGRM_PENDING_STATS	DBA_TAB_HISTGRM_PENDING_STATS	USER_TAB_HISTGRM_PENDING_STATS
ALL_TAB_PENDING_STATS	DBA_TAB_PENDING_STATS	USER_TAB_PENDING_STATS
ALL_TAB_STAT_PREFS	DBA_TAB_STAT_PREFS	USER_TAB_STAT_PREFS
	DBA_TEMP_FREE_SPACE	
ALL_TRIGGER_ORDERING	DBA_TRIGGER_ORDERING	USER_TRIGGER_ORDERING
	DBA_USERS_WITH_DEFPWD	
	DBA_WORKLOAD_CAPTURES	
	DBA_WORKLOAD_CONNECTION_MAP	
	DBA_WORKLOAD_FILTERS	
	DBA_WORKLOAD_REPLAY_DIVERGENCE	
	DBA_WORKLOAD_REPLAYS	

45.3 路线图：DICTIONARY(DICT)和 DICT_COLUMNS

组成 Oracle 数据字典的对象描述可通过 DICTIONARY 视图进行访问。该视图还可通过公有同义词 DICT 访问，它查询数据库以确定用户能够查看哪些数据字典视图。它还为这些视图搜索公有同义词。

下面的示例查询 DICT 视图，以查看名称中包含“MVIEWS”串的所有数据字典视图。如下面的程序清单所示，通过非 DBA 账户从 DICT 视图中进行选择，返回与搜索条件相匹配的每个数据字典对象的对象名和注释。该视图中只有两列，即 Table_Name 列和与表相关联的 Comments 列。

```
column Comments format a35 word_wrapped
column Table_Name format a25

select Table_Name, Comments
   from DICT
  where Table_Name like '%MVIEWS%'
  order by Table_Name;

TABLE_NAME                COMMENTS
------------------------  -----------------------------------
ALL_BASE_TABLE_MVIEWS     All materialized views with log(s)
                          in the database that the user can see
ALL_MVIEWS                All materialized views in the database
ALL_REGISTERED_MVIEWS     Remote materialized views of local
                          tables that the user can see
USER_BASE_TABLE_MVIEWS    All materialized views with log(s)
                          owned by the user in the database
USER_MVIEWS               All materialized views in the database
USER_REGISTERED_MVIEWS    Remote materialized views of local
                          tables currently using logs owned
                          by the user

6 rows selected.
```

可以通过 DICT_COLUMNS 视图查询字典视图的列。与 DICTIONARY 视图一样：DICT_COLUMNS 视图显示已经输入数据库中的关于数据字典视图的注释。DICT_COLUMNS 视图有 3 列：Table_Name、Column_Name 和 Comments。查询 DICT_COLUMNS 视图可以确定哪些数据字典视图最符合要求。

例如，如果想查看数据库对象的空间分配和使用信息，而又不能肯定哪些数据字典视图存储这些信息，可以查询 DICT_COLUMNS 视图，如下例所示，它查找包含 BLOCKS 列的所有字典表：

```
select Table_Name
    from DICT_COLUMNS
  where Column_Name = 'BLOCKS'
    and Table_Name like 'USER%'
  order by Table_Name;

TABLE_NAME
--------------------
USER_ALL_TABLES
USER_EXTENTS
USER_FREE_SPACE
USER_OBJECT_TABLES
USER_SEGMENTS
USER_TABLES
USER_TAB_PARTITIONS
USER_TAB_PENDING_STATS
USER_TAB_STATISTICS
USER_TAB_SUBPARTITIONS
USER_TS_QUOTAS
```

要列出在上例中可能用到的所有可用的列名，应当查询 DICT_COLUMNS 视图：

```
select distinct Column_Name
    from DICT_COLUMNS
  order by Column_Name;
```

当不能肯定在哪里能找到需要的数据时，只需查看 DICTIONARY 视图和 DICT_COLUMNS 视图即可。如果觉得大量的视图都有用，则可查询 DICTIONARY 视图(如第一个示例所示)，以查看每个视图的注释。

45.4　从表、列、视图、同义词和序列中选择

用户目录列出了用户可以从中选择记录的所有对象，即可在查询的 from 子句中列出的任何对象。尽管不能直接在 from 子句中引用序列，但 Oracle 还是把它们包含在目录中。本节将介绍如何检索相关表、列、视图、同义词、序列和用户目录的信息。

45.4.1　目录：USER_CATALOG(CAT)

查询 USER_CATALOG 目录可显示用户拥有的所有表、视图、同义词和序列。Table_Name 列显示对象名(即使不是一个表)，而 Table_Type 列显示对象的类型：

```
select Table_Name, Table_Type
    from USER_CATALOG
  where Table_Name like 'T%';
```

USER_CATALOG 也可以被公有同义词 CAT 引用。

此外，还有两个有用的目录。ALL_CATALOG 目录列出 USER_CATALOG 视图中的所有内容，以及您或者 PUBLIC 用户可以访问的任何对象。DBA_CATALOG 是显示数据库中的所有表、视图、序列和同义词的 DBA 级目录。除了在 USER_CATALOG 目录查询中显示的 Table_Name 列和 Table _Type 列外，ALL_CATALOG 目录和 DBA_CATALOG 目录还包含 Owner 列。

45.4.2 对象：USER_OBJECTS(OBJ)

USER_CATALOG 目录只显示表、视图、序列和同义词的信息。要检索关于所有对象类型的信息，可查询 USER_OBJECTS 视图。可以使用该视图查找许多类型的对象，包括群集、数据库链接、目录、函数、索引、库、程序包、程序包体、Java 类、抽象数据类型、资源计划、序列、同义词、表、触发器、物化视图、LOB 和视图。下面的程序清单列出了 USER_OBJECTS 视图的列。

```
COLUMN_NAME
------------------------------
OBJECT_NAME
SUBOBJECT_NAME
OBJECT_ID
DATA_OBJECT_ID
OBJECT_TYPE
CREATED
LAST_DDL_TIME
TIMESTAMP
TEMPORARY
GENERATED
SECONDARY
NAMESPACE
EDITION_NAME
```

USER_OBJECTS(以其公有同义词 OBJ 而为人们熟知)包含了几条重要的信息，这些信息在其他数据字典视图中找不到。它记录了对象的创建日期(Created 列)和上次更改对象的时间(Last_DDL_Time 列)。在试图协调同一应用程序中不同的对象组时，这些列非常有用。

注意：

如果以某种方式重新创建对象，如通过 Import 实用程序，则它们的 Created 值将更改为上次创建的时间。

另外两个对象列表也可用。ALL_OBJECTS 列出了 USER_OBJECTS 视图中的所有内容以及您或 PUBLIC 用户可以访问的任何对象。DBA_OBJECTS 视图是一个 DBA 级的对象列表，显示了数据库中的所有对象。除了 USER_OBJECTS 视图中显示的列以外，ALL_OBJECTS 和 DBA_OBJECTS 都包含一个 Owner 列。

45.4.3　表：USER_TABLES(TABS)

尽管所有的用户对象都显示在 USER_OBJECTS 视图中，但是这些对象的属性几乎没有显示出来。为了得到对象的更多信息，需要查看特定于此种对象的视图。对于表，相应的视图为 USER_TABLES。它也可以通过公有同义词 TABS 进行引用。

注意：
Oracle 的早期版本包含了一个名为 TAB 的视图。该视图的功能类似于 TABS，目前仍然支持该视图，因为 Oracle 的产品还要引用它。但是，TAB 不包含 TABS 所包含的列。可以在数据字典查询中使用 TABS。

USER_TABLES 视图中的列主要可分为 4 类(标识列、与空间相关的列、与统计信息相关的列以及其他列)，下面的清单显示了 USER_TABLES 视图的列：

```
TABLE_NAME
TABLESPACE_NAME
CLUSTER_NAME
IOT_NAME
STATUS
PCT_FREE
PCT_USED
INI_TRANS
MAX_TRANS
INITIAL_EXTENT
NEXT_EXTENT
MIN_EXTENTS
MAX_EXTENTS
PCT_INCREASE
FREELISTS
FREELIST_GROUPS
LOGGING
BACKED_UP
NUM_ROWS
BLOCKS
EMPTY_BLOCKS
AVG_SPACE
CHAIN_CNT
AVG_ROW_LEN
AVG_SPACE_FREELIST_BLOCKS
NUM_FREELIST_BLOCKS
DEGREE
INSTANCES
CACHE
TABLE_LOCK
SAMPLE_SIZE
LAST_ANALYZED
PARTITIONED
IOT_TYPE
TEMPORARY
SECONDARY
NESTED
```

```
BUFFER_POOL
ROW_MOVEMENT
GLOBAL_STATS
USER_STATS
DURATION
SKIP_CORRUPT
MONITORING
CLUSTER_OWNER
DEPENDENCIES
COMPRESSION
COMPRESS_FOR
DROPPED
READ_ONLY
```

表名在显示 Table_Name 列中。Backed_Up 列说明该表自最后一次更改以来是否备份过。如果该表已经被分区，则分区的列值为 YES(与分区有关的数据字典视图将在 45.12 节中介绍)。

与空间相关的列，如 Pct_Free、Initial_Extent、Max_Extents 以及 Pct_Increase，请参见附录 A 中的 STORAGE 条目。与统计信息相关的列，如 Num_Rows、Blocks、Empty_Blocks、Avg_Row_Len 以及 Last_Analyzed，在分析表时填充。

注意：
可以使用 USER_TAB_STATISTICS 视图访问表的统计信息。可以使用 USER_TAB_PENDING 挂起统计信息。

从 USER_TABLES 视图中查询 Table_Name 列，将显示当前账户中所有表的名称。下面的示例列出了以字母 L 开头的所有表名。

```
select Table_Name from USER_TABLES
  where Table_Name like 'L%';
```

ALL_TABLES 视图显示用户拥有的所有表以及授权该用户访问的表(直接或者通过授权给 PUBLIC)。列出查询可用的表的大多数第三方报表工具都可以通过查询 ALL_TABLES 视图获得。因为 ALL_TABLES 视图能够包含多个用户的项，所以除了 USER_TABLES 视图中的列，还包括一个 Owner 列。DBA_TABLES 视图列出了数据库中的所有表，它包含与 ALL_TABLES 视图相同的列定义。

USER_TABLES 视图中的 Degree 列和 Instance 列关系到如何在并行查询期间查询表。关于 Degree、Instance 和其他表参数的信息，请参阅附录 A 中的 CREATE TABLE 项。

注意：
对于外部表，请参阅 USER_EXTERNAL_TABLES 和 USER_EXTERNAL_LOCATIONS。

USER_TABLES 视图和其他数据字典视图包含一个新的名为 Dropped 的列，该列具有一个为 YES 或 NO 的值。关于删除对象的信息介绍，请参阅有关回收站的相关内容。

45.4.4　列：USER_TAB_COLUMNS(COLS)

尽管用户不从列中进行查询，但显示列的数据字典视图与表的数据字典视图仍然密切相关。USER_TAB_COLUMNS 视图列出了特定于列的信息。也可以通过公有同义词 COLS 查询 USER_TAB_COLUMNS 视图。

可从 USER_TAB_COLUMNS 视图查询的列可以主要分为 3 类：

- 标识列，如 Table_Name、Column_Name 及 Column_ID
- 与定义有关的列，如 Data_Type、Data_Length、Data_Precision、Data_Scale、Nullable 及 Default_Length
- 与统计信息有关的列，如 Num_Distinct、Low_Value、High_Value、Density、Num_Nulls 及其他列

Table_Name 列和 Column_Name 列包含表和列的名称。与定义有关的列的用法在附录 A 中的“DATATYPE”项中描述。与统计信息有关的列在分析表时填充。列的统计信息也在 USER_TAB_COL_STATISTICS 视图(稍后介绍)中提供。

要查看表的列定义，可查询 USER_TAB_COLUMNS 视图，并在 where 子句中指定 Table_Name：

```
select Column_Name, Data_Type
    from USER_TAB_COLUMNS
  where Table_Name = 'NEWSPAPER';

COLUMN_NAME    DATA_TYPE
-------------- ---------
FEATURE        VARCHAR2
SECTION        CHAR
PAGE           NUMBER
```

本示例中的信息还可通过 SQL*Plus 中的 describe 命令获得。但是，describe 命令不提供查看列的默认值和统计信息的选项。ALL_TAB_COLUMNS 视图显示用户拥有的和授权该用户(直接授权或者通过授权给 PUBLIC(所有用户))访问的所有表和视图的列。由于 ALL_TAB_COLUMNS 能包含多个用户的项，因此，除了 USER_TAB_COLUMNS 视图中的列，它还包含一个 Owner 列。DBA_TAB_COLUMNS 视图与 ALL_TAB_COLUMNS 视图有相同的列定义，DBA _TAB_COLUMNS 视图列出了数据库中所有表和视图的列定义。

1. 列统计信息

大多数的列统计信息可从 USER_TAB_COLUMNS 视图(以前的存放地点)和 USE_TAB_COL_ STATISTICS 视图中得到。用于 USER_TAB_COL_STATISTICS 视图的列就是 USER_TAB_COLUMNS 视图中与统计信息有关的列，以及 Table_Name 列和 Column_Name 列。

USER_TAB_COL_STATISTICS 视图包含由 USER_TAB_COLUMNS 视图提供的向后兼容的统计信息列。可以通过 USER_TAB_COL_STATISTICS 视图访问这些列。

2. 列值直方图

可以使用直方图改善基于成本的优化程序所使用的分析。USER_TAB_HISTOGRAMS 视图包含每一列的直方图信息，包括 Table_Name、Column_Name、Endpoint_Number、Endpoint_Value 和 Endpoint_Actual_Value。USER_TAB_HISTOGRAMS 视图中的值由优化程序使用，以确定表中列值的分布。USER_TAB_HISTOGRAMS 视图的“ALL”版本和“DBA”版本都是可用的版本。

3. 可更新的列

可以在视图中更新记录，该视图包含它们在视图查询中的连接，只要该连接满足某种特定的条件即可。USER_UPDATABLE_COLUMNS 视图列出了可更新的所有列。可以查询该列的 Owner、Table_Name 和 Column_Name。如果列能够更新，则 Updatable 列有一个值为 YES；如果列不能更新，则有一个值为 NO。还可以进行查询，以确定是否能通过 Insertable 列和 Deletable 列在视图中插入或删除记录。

45.4.5 视图：USER_VIEWS

通过 USER_VIEWS 数据字典视图，可以访问视图的基本查询，USER_VIEWS 数据字典视图包含 10 列，其中主要的 3 列如下：

- View_Name　　视图名
- Text_Length　　视图的基本查询的长度，以字符为单位
- Text　　视图使用的查询

其他列主要与对象视图和版本有关，稍后将在本节介绍。

注意：
本节只适用于传统视图。对于物化视图，详细内容见 45.9 节。

Text 列的数据类型为 LONG。当通过 SQL*Plus 查询 USER_VIEWS 视图时，这可能会产生问题，因为 SQL*Plus 会截断 LONG 数据。但是，截断的位置可以通过 set long 命令改变。USER_VIEWS 视图提供恰当设置 LONG 截断点的机制，下面的示例将介绍这个机制。

Text_Length 列显示视图的查询的长度。因此，SQL*Plus 的 LONG 截断点必须为一个等于或大于视图的 Text_Length 值的值。例如，下面的程序清单显示了 View_Name 列为 AGING、Text_Length 列为 355 的视图。

```
select View_Name, Text_Length
  from USER_VIEWS
 where View_Name = 'AGING';

View_Name       Text_Length
--------------- -------------
AGING           355
```

由于该视图的文本长度为 355 个字符，因此可以用 set long 命令将 LONG 截断点至少增加到 355(默认值为 80)，以查看视图的查询的全部文本。

```
set long 355
```

然后，可以查询 USER_VIEWS 视图，以获得视图的 Text 列，使用的查询如下面的程序清单所示：

```
select Text
  from USER_VIEWS
 where View_Name = 'AGING';
```

如果没有使用 set long 命令，则输出结果将截断为 80 个字符，并且没有任何消息说明为什么会截断。在查询其他视图前，应当重新检查视图的 Text_Length 值。

注意：
可以从 USER_TAB_COLUMNS 视图中查询视图的列定义，也可以用该视图查询表的列定义。

如果在视图中使用了列别名，并且列别名是视图的查询的一部分，那么视图信息的数据字典查询将得以简化。由于该视图查询的整个文本显示在 USER_VIEWS 视图中，因此也将显示列别名。

可以使用下面的格式创建视图：

```
create view NEWSPAPER_VIEW (SomeFeature, SomeSection)
  as select Feature, Section
 from NEWSPAPER;
```

在 create view 命令的开头部分列出的列名从查询中删除了列别名，从而防止通过 USER_VIEWS 视图查看它们。查看视图的列名的唯一方法是查询 USER_TAB_COLUMNS 视图。如果列名在查询语句中，则对于该查询和列名来说，只需要查询一个数据字典视图(USER_VIEWS)即可。

例如，已知上例中创建的 NEWSPAPER_VIEW 视图，如果查询 USER_VIEWS 视图，就会看到：

```
select Text
  from USER_VIEWS
 where View_Name ='NEWSPAPER_VIEW';

TEXT
----------------------------------------
select Feature, Section from NEWSPAPER
```

此查询没有显示用户赋予的新列名，因为没有将这些列名作为该视图查询的一部分。为使这些列名显示在 USER_VIEWS 视图中，可以在该视图的查询中添加它们，以作为列别名：

```
create view NEWSPAPER_VIEW
  as select Feature SomeFeature, Section SomeSection
 from NEWSPAPER;
```

现在，如果查询 USER_VIEWS 视图，则列别名将作为视图的查询文本的一部分显示出来：

```
select Text
  from USER_VIEWS
 where View_Name ='NEWSPAPER_VIEW';

TEXT
------------------------------------------------
select Feature SomeFeature, Section SomeSection
  from NEWSPAPER
```

为了支持对象视图，USER_VIEWS 视图包含下面的列：

- Type_Text　　该类型视图的 type 子句
- Type_Text_Length　　该类型视图的 type 子句的长度
- OID_Text　　该类型视图的 WITH OID 子句
- OID_Text_Length　　该类型视图的 WITH OID 子句的长度
- View_Type_Owner　　该类型视图的视图类型的所有者
- View_Type　　视图类型

关于对象视图和类型的详细信息，请参阅第 38 章和第 41 章。

ALL_VIEWS 视图列出了用户所拥有的全部视图以及授权用户(直接授权或者授权给 PUBLIC)访问的视图。由于 ALL_VIEWS 视图包含多个用户的项，因此，除了本节前面列出的列外，它还包含一个 Owner 列。DBA_VIEWS 视图列出了数据库中的全部视图，DBA_VIEWS 视图与 ALL_VIEWS 视图有相同的列定义。

45.4.6 同义词：USER_SYNONYMS(SYN)

USER_SYNONYMS 视图列出了用户所拥有的全部同义词。这些列如下所示：

- Synonym_Name　　同义词的名称
- Table_Owner　　同义词所指的表的所有者
- Table_Name　　同义词所指的表名
- DB_Link　　同义词使用的数据库链接名

当在应用程序中调试程序或解决用户访问的对象所遇到的问题时，USER_SYNONYMS 视图非常有用。USER_SYNONYMS 视图也简写为公有同义词 SYN。

如果同义词不使用数据库链接，则 DB_Link 列将为 NULL。因此，如果想通过自己的账户所拥有的同义词来查看当前使用的数据库链接的列表，则可以执行下面的查询：

```
select distinct DB_Link
  from USER_SYNONYMS
 where DB_Link is not null;
```

ALL_SYNONYMS 视图列出了该用户拥有的所有同义词、公有同义词和授权该用户访问的所有同义词。由于 ALL_SYNONYMS 视图可包含多个用户的项，因此，除了本节前面所显示

的列，它还包含一个 Owner 列。DBA_SYNONYMS 视图列出了数据库中的所有同义词，它与 ALL_SYNONYMS 视图具有相同的列定义。

45.4.7　序列：USER_SEQUENCES(SEQ)

要显示序列的属性，可以查询 USER_SEQUENCES 数据字典视图。该视图也能用公有同义词 SEQ 进行查询。USER_SEQUENCES 视图的列如下所示：

- Sequence_Name　序列名
- Min_Value　序列的最小值
- Max_Value　序列的最大值
- Increment_By　序列值之间的增量
- Cycle_Flag　一个标志，一旦达到 Max_Value 值，用来表明该值能否再循环回 Min_Value 值
- Order_Flag　一个标志，用来表明序列号是否按顺序生成
- Cache_Size　内存中缓存的序列项的编号
- Last_Number　使用的或缓存的最后一个序列号，一般大于缓冲区中的最后一个值

Last_Number 列在普通的数据库操作过程中不被更新，它在数据库的重新启动/恢复操作中使用。

ALL_SEQUENCES 视图列出了用户拥有的所有序列，或授权该用户或 PUBLIC 所有用户访问的序列。由于 ALL_SEQUENCES 视图包含多个用户的项，因此除已经列出的列外，它还包含一个 Sequence_Owner 列。DBA_SEQUENCES 视图列出了数据库的全部序列，它与 ALL_SEQUENCES 视图具有相同的列定义。

45.5　回收站：USER_RECYCLEBIN 和 DBA_RECYCLEBIN

可以使用 flashback table 命令恢复已经删除的表和相关对象。为了查看回收站中当前的对象，可以查询 USER_RECYCLEBIN 视图。在查询中，公有同义词 RECYCLEBIN 可用来代替 USER_RECYCLEBIN 视图。无论对象是否可以恢复(Can_Undrop 列)，或者对象是否由于与其相关的基对象被删除而删除(Base_Object 列值)，都可以查看源对象名。

对象一直保存在回收站中，直到被清除。如果在对象被删除的表空间中不再有空闲空间，则这些对象可以从回收站自动清除。通过 DBA_RECYCLEBIN 视图，DBA 可以查看所有用户的回收站中的所有对象。本视图无“ALL”版本。关于从回收站删除对象的详细介绍，请参阅附录 A 中的“purge”命令。

45.6　约束和注释

约束和注释有助于理解表和列之间的相互联系。注释只是信息，不能对存储在所描述对

象中的数据施加任何条件。而约束定义了数据有效的条件。典型的约束包括 NOT NULL、UNIQUE、PRIMARY KEY 和 FOREIGN KEY。以下几节将介绍如何从数据字典中检索有关约束和注释的数据。

45.6.1 约束：USER_CONSTRAINTS

通过 USER_CONSTRAINTS 视图，可以访问约束信息。此信息在试图更改数据约束或解决应用程序的数据问题时非常有用。下面的清单列出了该视图的列。

```
COLUMN_NAME
------------------------------
OWNER
CONSTRAINT_NAME
CONSTRAINT_TYPE
TABLE_NAME
SEARCH_CONDITION
R_OWNER
R_CONSTRAINT_NAME
DELETE_RULE
STATUS
DEFERRABLE
DEFERRED
VALIDATED
GENERATED
BAD
RELY
LAST_CHANGE
INDEX_OWNER
INDEX_NAME
INVALID
VIEW_RELATED
```

尽管这是 1 个“USER”视图，但却包含 1 个 Owner 列。在此视图中，Owner 指约束的所有者，而不是表的所有者(用户)。

Constraint_Type 列的有效值如下：

- C CHECK 约束，包括一些 NOT NULL
- P 主键约束
- R 外键(引用)约束
- U 唯一约束
- V WITH CHECK OPTION 约束(用于视图)
- O WITH READ ONLY 约束(用于视图)

要想获取关于约束的有用信息，理解约束类型很重要。

外键约束总是具有 R_Owner 和 R_Constraint_Name 这两列的值。这两列指出外键引用了哪个约束。一个外键引用另一个约束，而不是另一列。由于列的 NOT NULL 约束将存储为 CHECK 约束，因此它们的 Constrain_Type 为 C。

查询 USER_CONSTRAINTS 视图将会得到表中所有约束的名称。这在解释只提供违反约束的约束名的错误消息时非常有用。

一旦知道了约束的名称和类型，就能通过 USER_CONS_COLUMNS 视图检查与之相关的列，该内容将在 45.6.2 节介绍。

如果在创建约束时未给约束赋予名称，则 Oracle 将生成一个唯一的约束名。详细信息请参阅附录 A 中的“INTEGRITY CONSTRAINTS ”部分。如果约束名由系统生成，则通过 Generated 列指出该事实。

如果延迟一个约束(由 Deferred 列指出)，则该约束不能在事务处理期间实施。例如，如果在相关表中执行大量的更新操作，并且不能保证事务处理的顺序正确，就可以决定延迟在表上检查的约束，直到所有的更新操作完成为止。

ALL_CONSTRAINTS 视图列出了用户或 PUBLIC 能访问的所有表的约束。DBA_CONSTRAINTS 视图列出了数据库中的所有约束。这些视图都有与 USER_CONSTRAINTS 视图相同的列组(因为它们都包含 Owner 列)。

45.6.2　约束列：USER_CONS_COLUMNS

可以通过 USER_CONS_COLUMNS 数据字典视图查看与约束相关的列。如果已经查询 USER_CONSTRAINTS 视图，以得到所涉及约束的类型和名称，就可以使用 USER_CONS_COLUMNS 视图来确定约束涉及哪些列。此视图中的列如下所示：

- Owner　　约束的所有者
- Constraint_ Name　　约束名
- Table_Name　　与约束关联的表名
- Column_Name　　与约束关联的列名
- Position　　约束定义中列的顺序

USER_CONS_COLUMNS 中只有两列是 USER_CONSTRAINTS 视图中没有的，即 Column _Name 列和 Position 列。该表中的示例查询如下面的程序清单所示(您的约束名会有所不同)：

```
select Column_Name, Position
    from USER_CONS_COLUMNS
  where Constraint_Name = 'SYS_C0008791';

COLUMN_NAME    POSITION
------------   --------
FIRSTNAME             1
LASTNAME              2
```

如上述程序清单所示，FirstName 和 LastName 结合在一起形成了该约束(本例中是一个主键)。

Postion 列非常重要。在创建唯一约束或主键约束时，Oracle 自动在指定的列组上创建一个唯一约束。此索引基于指定的列顺序创建。列顺序又影响索引的性能。如果查询的 where 子句使用了索引的首列(Postion＝1)，则使用多列组成的索引最为有效。关于优化程序的详细信息，请参阅第 46 章。

ALL_CONS_COLUMNS 视图和 DBA_CONS_COLUMNS 视图与 USER_CONS_COLUMNS 有相同的列定义。ALL_CONS_COLUMNS 视图能够显示用户可以访问的所有表上约束的列

信息，而不管所有者是谁。DBA_CONS_COLUMNS 视图列出了整个数据库的列级约束信息。

45.6.3 约束异常：EXCEPTIONS

在启用已包含数据的表约束时，可能会遇到数据内的约束冲突问题。例如，您可能想在某一列上创建主键约束使该列的多条记录具有相同的值，但在这样的列上创建主键约束将会失败(因为违反了约束的唯一性)。

可以捕获导致约束创建失败的行的信息。首先，在模式中创建一个名为 EXCEPTIONS 的表，用于创建该表的 SQL 脚本命名为 utlexcpt.sql，此脚本通常位于 Oracle 主目录下的 /rbdms/admin 目录中。

EXCEPTIONS 表包含 4 列：Row_ID(违反约束的每一行的 ROWID)、Owner(违反约束的所有者)、Table_Name(在其上创建违反约束的表)和 Constraint(行违反的约束)。在创建 EXCEPTIONS 表后，应启用一个主键约束：

```
alter table NEWSPAPER enable PRIMARY KEY
exceptions into EXCEPTIONS;

ORA-02437: cannot enable (NEWSPAPER.SYS_C00516) - primary key violated
```

此约束创建失败，并且违反约束的所有行的引用都被放置在 EXCEPTIONS 表中。例如，如果上一个示例中的主键约束产生异常，则应当查询 EXCEPTIONS 表，如下面的程序清单所示：

```
select Owner, Table_Name, Constraint from EXCEPTIONS;

OWNER       TABLE_NAME  CONSTRAINT
----------  ----------  ----------
PRACTICE    NEWSPAPER   SYS_C00516
PRACTICE    NEWSPAPER   SYS_C00516
```

有两行违反了 SYS_C00516 约束(在本例中，它是给予 NEWSPAPER 表的主键约束的约束名)。通过连接 EXCEPTIONS 表的 Row_ID 列与放置约束的表(本例中为 NEWSPAPER 表)的 ROWID 伪列，可以确定 NEWSPAPER 表的哪些行对应于这些异常：

```
select *
  from NEWSPAPER
 where RowID in
       (select Row_ID from EXCEPTIONS);
```

然后，将显示那些导致约束违反的行。

注意：

Row_ID 是 EXCEPTOINS 表中的一个实际列，数据类型为 ROWID。它与产生异常的表中的 ROWID 伪列连接。

闪回操作和恢复操作可以改变先前插入表中的行的 ROWID。

45.6.4 表注释：USER_TAB_COMMENTS

可以在创建表、视图或列后，再为它们添加注释。数据字典视图中的注释是通过

DICTIONARY 视图和 DICT_COLUMNS 视图显示记录的基础。要显示自己拥有的表的注释，可以使用 USER_TAB_COMMENTS 视图。

USER_TAB_COMMENTS 视图包含下面 3 列：

- Table_Name　　表名或视图名
- Table_Type　　对象类型(表、对象表或视图)
- Comments　　已经为该对象输入的注释

为了给一个表添加注释，可以使用 comment 命令，如下面的程序清单所示：

```
comment on table BIRTHDAY is 'Birthday list for Blacksburg
employees';
```

通过指定希望看到其注释的 Table_Name 列，就可以查询 USER_TAB_COMMENTS 视图，如下面的程序清单所示：

```
select Comments
   from USER_TAB_COMMENTS
  where Table_Name = 'BIRTHDAY';

COMMENTS
------------------------------------
Birthday list for Blacksburg
     employees
```

要删除注释，可将该注释设置为之间没有空格的两个单引号：

```
comment on table BIRTHDAY is '';
```

可以通过 ALL_TAB_COMMENTS 视图，查看可访问的所有表上的注释。ALL_TAB_COMMENTS 视图的列 Owner 指定表的所有者。DBA_TAB_COMMENTS 视图列出了数据库中的所有表，它也有一个 Owner 列。

45.6.5　列注释：USER_COL_COMMENTS

USER_COL_COMMENTS 视图显示已经为表中的列输入的注释。这些注释通过 comment 命令添加到数据库中。USER_COL_COMMENTS 视图包含 3 列：

- Table_Name　　表名或视图名
- Column_Name　　列名
- Comments　　已经为该列输入的注释

为给一列添加注释，可以使用 comment 命令，如下面的程序清单所示：

```
comment on column BIRTHDAY.AGE is 'Age in years';
```

通过指定希望看到其注释的 Table_Name 列和 Column_Name 列，可以查询 USER_COL_COMMENTS 视图：

```
select Comments
  from USER_COL_COMMENTS
 where Table_Name = 'BIRTHDAY'
   and Column_Name = 'AGE';
```

要删除注释，可将该注释设置为之间没有空格的两个单引号：

```
comment on column BIRTHDAY.AGE is '';
```

可以通过 ALL_COL_COMMENTS 视图查看可以访问的所有表上的列注释。ALL_COL_COMMENTS 视图的列 Owner 指定表的所有者。DBA_COL_COMMENTS 视图列出数据库中所有表的所有列，它也有一个 Owner 列。

45.7 索引和群集

虽然索引和群集不改变存储在表中的数据；但是，它们确实改变数据的访问方式。下面各节将介绍描述索引和群集的数据字典视图。45.12 节将介绍与分区索引相关的数据字典视图。

45.7.1 索引：USER_INDEXES(IND)

在 Oracle 中，索引与约束紧密相关。主键约束和唯一约束总是与索引相关联。与约束一样，USER_INDEXES 和 USER_IND_COLUMNS 这两个数据字典视图用于查询关于索引的信息。USER_INDEXES 视图也因其公有同义词 IND 而为人们所熟知。

USER_INDEXES 视图中的列可以分为 4 类(标识列、与空间相关的列、与统计信息相关的列和其他列)。USER_INDEXES 视图的列如下面的清单所示：

```
INDEX_NAME
INDEX_TYPE
TABLE_OWNER
TABLE_NAME
TABLE_TYPE
UNIQUENESS
COMPRESSION
PREFIX_LENGTH
TABLESPACE_NAME
INI_TRANS
MAX_TRANS
INITIAL_EXTENT
NEXT_EXTENT
MIN_EXTENTS
MAX_EXTENTS
PCT_INCREASE
PCT_THRESHOLD
INCLUDE_COLUMN
FREELISTS
FREELIST_GROUPS
PCT_FREE
LOGGING
BLEVEL
LEAF_BLOCKS
DISTINCT_KEYS
```

```
AVG_LEAF_BLOCKS_PER_KEY
AVG_DATA_BLOCKS_PER_KEY
CLUSTERING_FACTOR
STATUS
NUM_ROWS
SAMPLE_SIZE
LAST_ANALYZED
DEGREE
INSTANCES
PARTITIONED
TEMPORARY
GENERATED
SECONDARY
BUFFER_POOL
USER_STATS
DURATION
PCT_DIRECT_ACCESS
ITYP_OWNER
ITYP_NAME
PARAMETERS
GLOBAL_STATS
DOMIDX_STATUS
DOMIDX_OPSTATUS
FUNCIDX_STATUS
JOIN_INDEX
IOT_REDUNDANT_PKEY_ELIM
DROPPED
VISIBILITY
DOMIDX_MANAGEMENT
```

索引名显示在 Index_Name 列中。被索引表的所有者和表名显示在 Table_Owner 列和 Table_Name 列中。唯一索引的 Uniqueness 列设置为 UNIQUE，而非唯一索引的 Uniqueness 列设置为 NONUNIQUE。Table_Type 记录此索引是基于 TABLE 还是基于 CLUSTER。Dropped 列用于标识回收站中的索引。Visibility 列用于标识优化程序可见的索引。

与空间相关的列将在附录 A 中的“STORAGE”项中介绍。

与统计信息相关的列在分析表时进行填写(参见附录 A 中的 ANALYZE 命令)。Degree 列和 Instances 列指的是创建索引时使用的并行度。

注意：

可以使用 USER_IND_STATISTICS 视图访问索引的统计信息。

要查看表中的全部索引，可在 where 子句中使用 Table_Owner 列和 Table Name 列查询 USER_INDEXES 视图，如下面的程序清单所示：

```
select Index_Name, Uniqueness
    from USER_INDEXES
  where Table_Owner = 'PRACTICE'
     and Table_Name = 'BIRTHDAY';
```

```
INDEX_NAME        UNIQUENES
----------------- ---------
PK_BIRTHDAY       UNIQUE
```

虽然 Clustering_Factor 列不直接与群集相关，但表示表中行的排序程度。行的排序越有规律，范围查询的效率就越高(范围查询是指在其中给定某一列的取值范围，如 where LastName like 'A%')。

要找出哪些列是索引的成分，及其在索引中的顺序，应当查询 USER_IND_COLUMNS 视图，此视图将在 45.7.2 节描述。

ALL_INDEXES 视图显示用户拥有的全部索引，以及在授予该用户或 PUBLIC 访问权的表上创建的任何索引。由于 ALL_INDEXES 视图包含多个用户的项，因此它除了 USER_INDEXES 视图中的列外，还包含一个 Owner 列。DBA_INDEXES 视图列出了数据库中的所有索引，它与 ALL_INDEXES 视图具有相同的列定义。

注意：
对于基于函数的索引，请参考 USER_IND_EXPRESSIONS。

45.7.2 索引列：USER_IND_COLUMNS

通过查询 USER_IND_COLUMNS 视图，可以确定哪些列在索引中。该视图可用的列如下：

- Index_Name　　索引名
- Table_Name　　被索引的表名
- Column_Name　　索引中的列名
- Column_Position　　索引中列的位置
- Column_Length　　列的索引长度
- Char_Length　　该列最大的码点长度(Unicode)
- Descend　　指定该列是否按照降序排列的 Y/N 标志

本视图中有 5 列不在 USER_INDEXES 视图中，它们分别是 Column_Name、Column_Position、Column_Length、Char_Length 和 Descend。像 USER_INDEXES 视图中与统计信息相关的列一样，Column_ Length 列在分析索引的基表时填写。此表的示例查询如下面的程序清单所示，它使用 USER_INDEXES 示例中的 Index_Name 列(在本例中，Column_Position 列的别名为 Pos)。

```
select Column_Name, Column_Position Pos
    from USER_IND_COLUMNS
  where Index_Name = 'PK_BIRTHDAY';

COLUMN_NAME    POS
------------   ---
FIRSTNAME        1
LASTNAME         2
```

如此查询所示，PK_BIRTHDAY 索引由两列组成，这两列的顺序为 FirstName 和 LastName。关于优化程序怎样使用多列索引的详细内容，请参阅第 46 章。

ALL_IND_COLUMNS 视图可以用来显示用户能访问的所有表的索引的列信息，而不管所有者是谁。DBA_IND_COLUMNS 视图列出了整个数据库的列级索引信息。

如果创建了位图连接索引(在 Oracle9i 中引入的)，则可以查询 USER_JOIN_IND_COLUMNS 视图获得连接的详细信息。USER_JOIN_IND_COLUMNS 视图记录了连接中涉及的表名，以及内层和外层连接列。

45.7.3　群集：USER_CLUSTERS(CLU)

与群集相关联的存储参数和统计参数可通过 USER_CLUSTERS 视图(也因它的公有同义词 CLU 而为人们所熟知)进行访问。此数据字典视图中的列如下面的清单所示：

```
COLUMN_NAME
------------------------------
CLUSTER_NAME
TABLESPACE_NAME
PCT_FREE
PCT_USED
KEY_SIZE
INI_TRANS
MAX_TRANS
INITIAL_EXTENT
NEXT_EXTENT
MIN_EXTENTS
MAX_EXTENTS
PCT_INCREASE
FREELISTS
FREELIST_GROUPS
AVG_BLOCKS_PER_KEY
CLUSTER_TYPE
FUNCTION
HASHKEYS
DEGREE
INSTANCES
CACHE
BUFFER_POOL
SINGLE_TABLE
DEPENDENCIES
```

Cluster_Name 列包含群集名。Cluster_Type 指定群集使用标准的 B*树索引还是散列函数索引。

与空间相关的列的用法将在附录 A 中的“STORAGE”项中介绍。与统计信息相关的列在分析表时填写。

ALL_CLUSTERS 视图和 DBA_CLUSTERS 视图都有一个附加列 Owner，因为它们能列出多个模式中的群集。

45.7.4　群集列：USER_CLU_COLUMNS

为查看表列到群集列的映射，查询 USER_CLU_COLUMNS 视图，它的列如下所示：

- Cluster_Name 群集名
- Clu_Column_Name 群集中键列的名称
- Table_Name 群集中表的名称
- Tab_Column_Name 表中键列的名称

由于单个群集能存储多个表中的数据，因此 USER_CLU_COLUMNS 视图可用于确定哪些表的哪些列映射到群集列上。

此视图没有“ALL”版本。只有 USER_CLU_COLUMNS 视图可用于查看用户的群集列，并且用 DBA_CLU_CILUMNS 视图显示数据库中所有群集的列映射。

45.8 抽象数据类型和 LOB

下面几节将介绍与对象-关系结构(如抽象数据类型、方法和大对象(LOB))相关的数据字典视图。这些数据字典视图的“ALL”版本和“DBA”版本都可用。由于它们包含多个所有者的记录，因此这些视图的“ALL”版本和“DBA”版本都包含 Owner 列以及本节列出的列。

45.8.1 抽象数据类型：USER_TYPES

USER_TYPES 视图列出了在模式中创建的抽象数据类型，该视图包含的列有 Type_Name、属性个数(Attributes)以及为数据类型定义的方法个数(Methods)。

例如，ANIMAL_TY 数据类型有 3 个属性和 1 个方法(方法通过 create type body 命令定义)，如下面的程序清单所示：

```
select Type_Name,
       Attributes,
       Methods
  from USER_TYPES
 where Type_Name = 'ANIMAL_TY';

TYPE_NAME                 ATTRIBUTES     METHODS
------------------------- ----------  ----------
ANIMAL_TY                          3           1
```

1. 数据类型属性：USER_TYPE_ATTRS

为查看数据类型的属性，需要查询 USER_TYPE_ATTRS 视图。USER_TYPE_ATTRS 视图中的列如下面的清单所示：

```
COLUMN_NAME
------------------------------
TYPE_NAME
ATTR_NAME
ATTR_TYPE_MOD
ATTR_TYPE_OWNER
ATTR_TYPE_NAME
LENGTH
```

```
PRECISION
SCALE
CHARACTER_SET_NAME
ATTR_NO
INHERITED
```

可以查询 USER_TYPE_ATTRS 视图以查看嵌套的抽象数据类型之间的关系。例如，PERSON_TY 数据类型使用 ADDRESS_TY 数据类型，如下面的程序清单所示：

```
select Attr_Name,
       Length,
       Attr_Type_Name
  from USER_TYPE_ATTRS
 where Type_Name = 'PERSON_TY';

ATTR_NAME                      LENGTH ATTR_ TYPE_NAME
------------------------------ ------------ ------------------
NAME                                   25   VARCHAR2
ADDRESS                                     ADDRESS_TY
```

2. 数据类型方法：USER_TYPE_METHODS 和 USER_METHOD_PARAMS

如果一个类型有为其定义的方法，就可以查询 USER_TYPE_METHODS 视图来确定方法名。USER_TYPE_MOTHODS 视图包含相应的列来显示类型名(Type_Name)、方法名(Method_Name)、方法编号(Method_No，用于重载的方法)以及方法的类型(Method_Type)。USER_TYPE_MOTHODS 视图还包括显示方法返回的参数个数(Parameters)和结果个数(Results)的列。

例如，ANIMAL_TY 数据类型有一个方法，它是一个名为 AGE 的成员函数。AGE 方法有一个输入参数(Birthdate)和一个输出结果(年龄，以日期表示)。查询 USER_ TYPE_METHODS 视图表明，为 AGE 函数定义了两个参数，而这两个参数都不是用户所期望的：

```
select Parameters,
       Results
  from USER_TYPE_METHODS
 where Type_Name = 'ANIMAL_TY'
   and Method_Name = 'AGE';

PARAMETERS    RESULTS
---------- ----------
         2          1
```

如果 AGE 函数只定义了 1 个输入参数，那么为什么会有两个参数呢?为找到答案，查询 USER_METHOD_PARAMS 视图，它描述了方法的参数：

```
select Param_Name, Param_No, Param_Type_Name
  from USER_METHOD_PARAMS
 order by Param_No;

PARAM_NAME                     PARAM_NO   PARAM_TYPE_NAME
------------------------------ ---------- ---------------
SELF                                    1  ANIMAL_TY
BIRTHDATE                               2  DATE
```

USER_METHOD_PARAMS 视图显示每种方法的参数，Oracle 创建了一个隐式的 SELF

参数。方法的结果在 USER_METHOD_RESULTS 视图中给出：

```
select Method_Name,
       Result_Type_Name
  from USER_METHOD_RESULTS
 where Type_Name = 'ANIMAL_TY';

METHOD_NAME                    RESULT_TYPE_NAME
------------------------------ ------------------------------
AGE                            NUMBER
```

3. 其他数据类型：USER_REFS、USER_COLL_TYPES 和 USER_NESTED_TABLES

如果使用 REF(请参阅第 41 章)，就可以查询 USER_REFS 数据字典视图，以显示已经定义的 REF。USER_REFS 数据字典视图将显示包含 REF 列的表的表名(Table_Name)和对象列的列名(Column_Name)。REF 的属性(例如，它的存储是否有 ROWID)也可以通过 USER_REFS 数据字典视图访问。

收集器类型(嵌套表和可变数组)通过 USER_COLL_TYPES 数据字典视图描述。USER_COLL_TYPES 数据字典视图的列包括 Type_Name、Upper_Bound(针对可变数组)以及元素的长度和精度。可以将 USER_COLL_TYPES 数据字典视图与本节前面介绍的抽象数据类型数据字典视图一起使用，以确定收集器的类型结构。还可以查询 USER_NESTED_TABLES 视图和 USER_VARRAYS 视图来查看自己收集的详细信息。

45.8.2 LOB：USER_LOBS

如第 40 章所述，可以在数据库中存储大对象。USER_LOBS 视图提供在表中定义的 LOB 的信息，如下面的程序清单所示：

```
column column_name format A30

select Table_Name,
       Column_Name
  from USER_LOBS;

TABLE_NAME                     COLUMN_NAME
------------------------------ ------------------------------
PROPOSAL                       PROPOSAL_TEXT
PROPOSAL                       BUDGET
```

USER_LOBS 视图还显示了当 LOB 数据变大时，用来存放 LOB 数据的段名。但是，它不显示 LOB 列的数据类型。要查看 LOB 列的数据类型，可以描述包含 LOB 的表或查询 USER_TAB_COLUMNS 视图(如本章前面所介绍的那样)。

为了对 LOB 使用 BFILE 数据类型，必须创建目录(参阅附录 A 中的 creat directory 命令)。ALL_DIRECTORIES 数据字典显示了授权用户访问的每个目录的 Owner、Directory_Name 和 Directory_Path 列。此数据字典视图的“DBA”版本也是可用的。此视图没有“USER”版本。

45.9　数据库链接和物化视图

数据库链接和物化视图用来管理对远程数据的访问。依据所使用的物化视图的类型，或许能够使用物化视图日志。下面几节将介绍可用来显示数据库链接和物化视图信息的数据字典视图。关于数据库链接的详细信息，请参阅第 25 章。关于物化视图的详细信息，请参阅第 26 章。

45.9.1　数据库链接：USER_DB_LINKS

要查看在自己账户下创建的数据库链接，可以查询 USER_DB_LINKS 视图。此视图中的列包括链接名(DB_Link)、要链接的用户名(Username)、账户密码(Password)和链接字符串(Host)，它们显示了将用该链接创建的远程连接的信息。Username 和 Password 的值将用于登录由 Host 的值定义的远程数据库。

Host 列存储 Oracle Net 的服务名。该列存储在执行 create database link 命令期间指定的字符串，而且不改变其大小写。因此，在创建数据库链接时应当注意所用的大小写，否则 USER_DB_LINKS 视图的查询将不得不考虑 Host 列的大小写不一致性。例如，为了查找使用 HQ 服务描述符的数据库链接，需要输入以下命令：

```
select * from USER_DB_LINKS
where upper(Host)='HQ';
```

因为可能会有 Host 值为“hq”而非“HQ”的项。

注意：
如果使用默认值登录到远程数据库，则 Password 将为 NULL。

ALL_DB_LINKS 视图列出用户拥有的所有数据库链接或 PUBLIC 数据库链接。DBA_DB_ LINKS 视图列出数据库中的所有数据库链接。ALL_DB_LINKS 视图和 DBA_DB_LINKS 视图与 USER_DB LINKS 视图的列定义大致相同，只是 ALL_DB_LINKS 视图和 DBA_DB_LINKS 视图用 Owner 列取代了 Password 列。

关于使用数据库链接的详细信息，请参阅第 25 章。

45.9.2　物化视图

可以查询 USER_MVIEWS 视图来显示您的账户拥有的物化视图的信息。此视图显示了物化视图的结构信息及其刷新时间表，它的列如下面的清单所示。

注意：
关于物化视图的详细内容，请参阅第 26 章。

```
OWNER
MVIEW_NAME
```

```
CONTAINER_NAME
QUERY
QUERY_LEN
UPDATABLE
UPDATE_LOG
MASTER_ROLLBACK_SEG
MASTER_LINK
REWRITE_ENABLED
REWRITE_CAPABILITY
REFRESH_MODE
REFRESH_METHOD
BUILD_MODE
FAST_REFRESHABLE
LAST_REFRESH_TYPE
LAST_REFRESH_DATE
STALENESS
AFTER_FAST_REFRESH
UNKNOWN_PREBUILT
UNKNOWN_PLSQL_FUNC
UNKNOWN_EXTERNAL_TABLE
UNKNOWN_CONSIDER_FRESH
UNKNOWN_IMPORT
UNKNOWN_TRUSTED_FD
COMPILE_STATE
USE_NO_INDEX
STALE_SINCE
NUM_PCT_TABLES
NUM_FRESH_PCT_REGIONS
NUM_STALE_PCT_REGIONS
```

物化视图的名称可以在 USER_MVIEWS 视图的 Mview_Name 列中找到。视图的本地基表为 Container_Name 列。为了确定物化视图使用哪些数据库链接，可查询 Master_Link 列，如下例所示：

```
select Master_Link
  from USER_MVIEWS;
```

该查询返回的数据库链接名可用作 USER_DB_LINKS 视图查询的输入。此查询将显示在物化视图中可用的数据库链接的所有信息：

```
select *
  from USER_DB_LINKS
 where DB_Link in
    (select Master_Link
       from USER_MVIEWS);
```

第 26 章提供了在管理物化视图中其他有用的查询。

ALL_MVIEWS 视图和 DBA_MVIEWS 视图与 USER_MVIEWS 视图具有相同的列定义。可以使用 ALL_MVIEWS 视图显示用户能访问的所有物化视图的信息，而不管所有者是谁。DBA_MVIEWS 视图列出了数据库中所有用户的物化视图信息。

两个与物化视图相关的视图(USER_REFRESH 和 USER_REFRESH_CHILDREN)显示刷新组的信息。USER_MVIEW_REFRESH_TIMES 视图给出每个物化视图最近一次刷新的时间。可以通过 USER_MVIEW_COMMENTS 视图访问物化视图上的注释，并可以通过 USER_REWRITE_ EQUIVALENCES 视图来查询重写的详细信息。

可以查询 USER_MVIEW_ANALYSIS 视图，以查看支持查询重写的物化视图。如果物化视图包含对远程表的引用，则它不会在该视图中列出。可以查询物化视图的所有者、名称(Mview_Name)及基表的所有者(Mview_Table_Owner)。该视图中的许多列为标志，如 Summary(如果视图包含一个聚合，则为 Y)、Known_Stale(如果视图的数据与基表不一致，则为 Y)和 Contains_Views(如果物化视图引用了一个视图，则为 Y)。

如果物化视图包含聚合，则可查询 USER_MVIEW_AGGREGATES 视图得到关于聚合的详细信息。视图中的列如下所示：

- Owner　　物化视图的所有者
- Mview_Name　　物化视图的名称
- Position_in_Select　　在查询中的位置
- Container_Column　　列名
- Agg_Function　　聚合函数
- DistinctFlag　　如果聚合使用 DISTINCT 函数，则为 Y
- Measure　　量度的 SQL 文件，不包含聚合函数

可以从 USER_MVIEW_DETAIL_RELATIONS 数据字典视图和 USER_MVIEW_KEYS 数据字典视图中查询物化视图内关系的细节。如果物化视图基于连接，则可查看 USER_MVIEW_JIONS 视图获得连接的详细内容。通常 USER_MViEW_ANANLYSIS 是与物化视图有关的最常用的数据字典视图。

45.9.3　物化视图日志：USER_MVIEW_LOGS

物化视图日志可由许多物化视图用来确定主表中的哪些记录需要在该表的物化视图中刷新。关于用户日志的信息，可以从 USER_MVIEW_LOGS 数据字典视图中查询，日志的信息包括主表名(Master)、保存日志记录的表(Log_Table)和物化视图是基于主键还是基于 ROWID (Primary_Key 列和 ROWID 列)。查询 USER_MVIEW_LOGS 数据字典视图通常是出于维护目的，例如为了确定用于创建物化视图日志记录的触发器的名称等。

DBA_MVIEW_LOGS 数据字典视图与 USER_MVIEW_LOGS 数据字典视图具有相同的列定义，它显示了数据库中所有的物化视图日志。可查询 USER_BASE_TABLE_MVIEWS 数据字典视图，得到使用物化视图日志的物化视图主表的清单。

45.10　触发器、过程、函数和程序包

可以使用过程、程序包和触发器(存储在数据库中的 PL/SQL 代码块)实施业务规则或执行复杂的处理。触发器在第 34 章中介绍过。过程、函数和程序包在第 35 章介绍过。下面几节将介绍怎样查询数据字典，以获取触发器、过程、程序包和函数的信息。

45.10.1 触发器：USER_TRIGGERS

USER_TRIGGERS 视图包含了用户账户所拥有的触发器的信息。该视图显示了触发器类型和触发器主体，此视图的各列如下面的清单所示：

```
TRIGGER_NAME
TRIGGER_TYPE
TRIGGERING_EVENT
TABLE_OWNER
BASE_OBJECT_TYPE
TABLE_NAME
COLUMN_NAME
REFERENCING_NAMES
WHEN_CLAUSE
STATUS
DESCRIPTION
ACTION_TYPE
TRIGGER_BODY
CROSSEDITION
```

ALL_TRIGGERS 视图列出了可以访问的所有表的触发器。DBA_TRIGGERS 视图列出了数据库中的所有触发器。这两个视图都包含一个附加列 Owner，它记录触发器的所有者。

与触发器有关的另一个数据字典视图 USER_TRIGGER_COLS 显示触发器怎样使用列。它列出触发器所影响的每一列的名称，以及如何使用触发器。与 USER_TRIGGERS 数据字典视图一样，该数据字典视图的“ALL”版本和“DBA”版本都可用。

在 Oracle 11g 中，可以查询 USER_TRIGGER_ORDERING 数据字典视图。USER_TRIGGER_ORDERING 数据字典视图的列包括触发器名、它引用的触发器以及排序类型(在引用的触发器之后或之前)。

45.10.2 过程、函数和程序包：USER_SOURCE

已有过程、函数、程序包和程序包体的源代码可从 USER_SOURCE 数据字典视图中进行查询。USER_SOURCE 数据字典中的 Type 列将过程对象标识为 PROCEDURE、FUNCTION、PACKAGE、PACKAGE BODY、TRIGGER、TYPE、TYPE BODY 或 JAVA SOURCE。每一行代码都存储在 USER_SOURCE 数据字典中单独的一个记录中。

可以使用与以下程序清单相似的查询，从 USER_SOURCE 数据字典中选择信息。在本例中，选择 Text 列，并且按行号(Line)排序。对象名(Name)和对象类型(Type)用于指定显示哪些对象的源代码。

```
select Text
    from USER_SOURCE
  where Name = '&procedure_name'
    and Type = 'PROCEDURE'
order by Line;
```

注意：

由于行的次序由 Line 列维护，因此 Line 列应该用在 order by 子句中。

ALL_SOURCE 视图和 DBA_SOURCE 视图具有 USER_SOURCE 数据字典视图中的所有列和一个附加的 Owner 列(对象的所有者)。ALL_SOURCE 数据字典视图可用于显示用户能够访问的所有过程对象的源代码，而不管所有者是谁。DBA_SOURCE 数据字典视图列出了数据库中所有用户的源代码。

注意：
要查看 PL/SQL 单元不变的参数设置，可以查询 USER_STORED_ SETTINGS 视图。

1. 代码错误：USER_ERRORS

SQL*Plus 中的 show errors 命令检查 USER_ERRORS 数据字典视图，以显示与过程对象最近的编译有关的错误。show errors 将显示每个错误所在的行号和列号，以及错误消息的文本。

要查看与以前创建的过程对象有关的错误，可以直接查询 USER_ERRORS 数据字典视图。在查看与程序包体相关的错误时，必须这样做，因为在执行 show error 命令时，导致错误的程序包编译可能不显示程序包体的错误。当遇到与多个过程对象有关的编译错误时，也应该查询 USER_ERRORS 数据字典视图。

USER_ERRORS 数据字典视图中可用的列如下：

- Name　　过程对象的名称
- Type　　对象类型(PROCEDURE、FUNCTION、PACKAGE、PACKAGE BODY、TRIGGER、TYPE、TYPE BODY、VIEW、JAVA CLASS 或 JAVA SOURCE)
- Sequence　　行序列号，在查询的 order by 子句中使用
- Line　　出现错误的源代码的行号
- Position　　行中出现错误的位置
- Text　　错误消息的文本
- Attribute　　标志，指出选中的行是错误(ERROR)，还是警告(WARNING)
- Message_Number　　数值型错误号码，没有前缀

此视图的查询应总是在 order by 子句中包括 Sequence 列。此视图的“ALL”版本和“DBA”版本都可用。它们也有一个附加列 Owner，用来记录对象的所有者。

2. 代码大小：USER_OBJECT_SIZE

可以从 USER_OBJECT_SIZE 数据字典视图中查询过程对象在 SYSTEM 表空间中所使用的空间量。如下列程序清单所示，可以将 4 个不同大小的区域累加起来，确定在 SYSTEM 数据字典表中存储对象所用的总空间。这 4 个 Size 列与 Name 列和 Type 列一起构成了该视图的所有列。

```
select Source_Size+Code_Size+Parsed_Size+Error_Size Total
  from USER_OBJECT_SIZE
 where Name = '&procedure_name'
   and Type = 'PROCEDURE';
```

该视图还有一个可用的“DBA”版本，DBA_OBJECT_SIZE 视图列出了数据库中所有对象的大小。

45.11 维度

可以查询 USER_DIMENSIONS 视图的 Dimension_Name 列来查看维度的名称。USER_DIMENSIONS 视图的列还包含维度的所有者(Owner 列)、状态(Invalid 列，设置为 Y 或 N)以及修正级别(Revision 列)。维的属性可以通过其他数据字典视图进行访问。

要查看维度内部的层次结构，可以查询 USER_DIM_HIERARCHIES 视图。该视图只有 3 列，即 Owner、Dimension_Name 和 Hierarchy_Name。从 USER_DIM_ HIERARCHIES 数据字典视图中查询 GEOGRAPHY 维度，将返回其层次结构的名称：

```
select Hierarchy_Name
  from USER_DIM_HIERARCHIES;

HIERARCHY_NAME
------------------------------
COUNTRY_ROLLUP
```

可以查询 USER_DIM_CHILD_OF 来查看 COUNTRY_ROLLUP 层次结构的细节，如下面的程序清单所示。第一条命令创建一个名为 GEOGRAPHY 的维度，该维度负责记录国家和大洲的层次结构。

```
create dimension GEOGRAPHY
    level COUNTRY_ID     is COUNTRY.Country
    level CONTINENT_ID   is CONTINENT.Continent
    hierarchy COUNTRY_ROLLUP (
       COUNTRY_ID           child of
       CONTINENT_ID
    join key COUNTRY.Continent references CONTINENT_id);

column join_key_id format a4

select Child_Level_Name,
       Parent_Level_Name,
       Position, Join_Key_Id
  from USER_DIM_CHILD_OF
where Hierarchy_Name = 'COUNTRY_ROLLUP';

CHILD_LEVEL_NAME        PARENT_LEVEL_NAME               POSITION   JOIN
----------------------  ------------------------------  --------   ----
COUNTRY_ID              CONTINENT_ID                           1      1
```

要查看某个层次结构的连接键，可查询 USER_DIM_JION_KEY 数据字典视图：

```
select Level_name, Child_Join_Column
   from USER_DIM_JOIN_KEY
  where Dimension_Name = 'GEOGRAPHY'
    and Hierarchy_Name = 'COUNTRY_ROLLUP';

LEVEL_NAME                     CHILD_JOIN_COLUMN
------------------------------ ------------------------------
CONTINENT_ID                   CONTINENT
```

查询 USER_DIM_LEVELS 数据字典视图可以查看维度的级别，并且可以通过 USER_DIM_LEVEL_KEY 数据字典视图查看键列的层次。维的属性信息可通过 USER_DIM_ATTRIBUTES 数据字典视图进行访问。

与维度有关的所有数据字典视图都有“ALL”版本和“DBA”版本的视图。因为维度的“USER”视图包含一个 Owner 列，所以在相应的“ALL”视图和“DBA”视图中没有附加列。

45.12　包括分区和子分区的空间分配和使用情况

可以查询数据字典，以确定可用并可分配给数据库对象的空间。下面几节将介绍怎样为对象设置默认存储参数、空间使用限额、可用空间，以及对象的物理存储方式。关于 Oracle 存储数据的方法，请参阅第 22 章和第 51 章。

45.12.1　表空间：USER_TABLESPACES

可以查询 USER_TABLESPACES 数据字典视图，以确定有权访问的表空间和每个表空间中的默认存储参数。表空间的默认存储参数将用于存储该表空间中的每个对象，除非该对象的 create 或 alter 命令指定了自己的存储参数。下面的清单显示了 USER_TABLESPACES 数据字典视图中与存储相关的列，它们非常类似于 USER_TABLES 视图中与存储相关的列。更详细的信息请参阅附录 A 中的 STORAGE 项。

```
TABLESPACE_NAME
BLOCK_SIZE
INITIAL_EXTENT
NEXT_EXTENT
MIN_EXTENTS
MAX_EXTENTS
MAX_SIZE
PCT_INCREASE
MIN_EXTLEN
STATUS
CONTENTS
LOGGING
FORCE_LOGGING
EXTENT_MANAGEMENT
ALLOCATION_TYPE
SEGMENT_SPACE_MANAGEMENT
DEF_TAB_COMPRESSION
RETENTION
BIGFILE
PREDICATE_EVALUATION
ENCRYPTED
COMPRESS_FOR
```

此视图没有“ALL”版本。DBA_TABLESPACES 数据字典视图显示了所有表空间的存储参数。

45.12.2　空间限额：USER_TS_QUOTAS

USER_TS_QUOTAS 是一个很有用的视图，它用来确定当前已分配给用户的空间量和通

过表空间可用的最大空间量。下列程序清单给出了USER_TS_QUOTAS视图的一个示例查询：

```
select * from USER_TS_QUOTAS;

TABLESPACE_NAME        BYTES   MAX_BYTES     BLOCKS   MAX_BLOCKS
---------------   ---------  ---------- ----------  ----------
USERS                  67584           0          33           0
```

USER_TS_QUOTAS 视图为每个 Tablespace_Name 列包含一条记录。Bytes 列反映分配给用户拥有的对象的字节数。Max_Bytes 列是用户在表空间中可拥有的最大字节数。如果表空间没有限额，则 Max_Bytes 的值将为 0。Bytes 列和 Max_Bytes 列分别转换为 Blocks 列和 Max_Blocks 列中的 Oracle 块。

此视图没有“ALL”版本。DBA_TS_QUOTAS 视图显示了所有表空间的所有用户的存储限额，它是列出整个数据库中空间使用情况的一种非常有效的手段。

45.12.3 段和区：USER_SEGMENTS 和 USER_EXTENTS

如第 22 章所述，空间以段(segment)为单位分配给对象(如表、群集和索引)，它们在物理上对应于在数据库中创建的逻辑对象。可以查询 USER_SEGMENTS 视图来查看当前的存储参数和段内有效空间的使用情况。当超出某一存储限制时，USER_SEGMENTS 视图将非常有用，它的列如下面的清单所示。

```
SEGMENT_NAME
PARTITION_NAME
SEGMENT_TYPE
SEGMENT_SUBTYPE
TABLESPACE_NAME
BYTES
BLOCKS
EXTENTS
INITIAL_EXTENT
NEXT_EXTENT
MIN_EXTENTS
MAX_EXTENTS
MAX_SIZE
RETENTION
MINRETENTION
PCT_INCREASE
FREELISTS
FREELIST_GROUPS
BUFFER_POOL
```

段由称为区(extent)的连续区域组成。构成段的区将在 USER_EXTENTS 视图中描述。在 USER_EXTENTS 视图中，将看到段内每个区的实际大小，这对跟踪 next 和 pctincrease 设置(在字典托管的表空间中)的变化所产生的影响很有帮助。除了 Segment_Name 列、Segment_Type 列和 Tablespace_Name 列之外，USER_EXTENTS 视图还有 3 个新列，即 Extent_ID(识别段内的区)，Bytes(区的大小，以字节为单位)、Blocks(区的大小，以 Oracle 块为单位)。

USER_SEGMENTS 视图和 USER_EXTENTS 视图都有“DBA”版本，该版本有助于列出所有者的对象的空间使用情况。DBA_SEGMENTS 视图和 DBA_EXTENTS 视图都有一个附加的 Owner 列。如果想要列出在表空间中拥有段的全部所有者，则可以在 DBA_SEGMENTS 视图中基于 Tablespace_ Name 列进行查询，然后列出表空间中段的全部所有者。这些视图都没有“ALL”版本。

45.12.4　分区和子分区

单个表的数据可以存储多个分区上。关于分区的示例和对可用索引选项的描述，请参阅第 18 章。要查看一个表如何分区，应当查询 USER_PART_TABLES 数据字典视图，其列如下面的清单所示。

```
TABLE_NAME
PARTITIONING_TYPE
SUBPARTITIONING_TYPE
PARTITION_COUNT
DEF_SUBPARTITION_COUNT
PARTITIONING_KEY_COUNT
SUBPARTITIONING_KEY_COUNT
STATUS
DEF_TABLESPACE_NAME
DEF_PCT_FREE
DEF_PCT_USED
DEF_INI_TRANS
DEF_MAX_TRANS
DEF_INITIAL_EXTENT
DEF_NEXT_EXTENT
DEF_MIN_EXTENTS
DEF_MAX_EXTENTS
DEF_MAX_SIZE
DEF_PCT_INCREASE
DEF_FREELISTS
DEF_FREELIST_GROUPS
DEF_LOGGING
DEF_COMPRESSION
DEF_COMPRESS_FOR
DEF_BUFFER_POOL
REF_PTN_CONSTRAINT_NAME
INTERVAL
```

USER_PART_TABLES 数据字典视图中的大多数列定义了表分区的默认存储参数。当向表中添加分区时，它将默认使用在 USER_PART_TABLES 数据字典视图中给出的存储参数。USER_PART_TABLES 数据字典视图还显示了表中的分区数(Partition_Count)、分区键中的列数(Partitioning_Key_Count)，以及分区的类型(Partitioning_Type)。

USER_PART_TABLES 数据字典视图为已经分区的每个表存储一行。要查看属于表的每个分区的信息，可以查询 USER_TAB_PARTITIONS 数据字典视图。在 USER_TAB_PARTITIONS 数据字典视图中，您将看到每个表的分区对应一行。USER_TAB_PARTITIONS 数据字典视图的列如下面的清单所示。

```
TABLE_NAME
COMPOSITE
PARTITION_NAME
SUBPARTITION_COUNT
HIGH_VALUE
HIGH_VALUE_LENGTH
PARTITION_POSITION
TABLESPACE_NAME
PCT_FREE
PCT_USED
INI_TRANS
MAX_TRANS
INITIAL_EXTENT
NEXT_EXTENT
```

```
MIN_EXTENT
MAX_EXTENT
MAX_SIZE
PCT_INCREASE
FREELISTS
FREELIST_GROUPS
LOGGING
COMPRESSION
COMPRESS_FOR
NUM_ROWS
BLOCKS
EMPTY_BLOCK
AVG_SPACE
CHAIN_CNT
AVG_ROW_LEN
SAMPLE_SIZE
LAST_ANALYZED
BUFFER_POOL
GLOBAL_STATS
USER_STATS
```

USER_TAB_PARTITIONS 数据字典视图包含标识分区所属表的列，并显示分区的存储参数和统计信息。在分析表时填充与统计相关的列。标识列显示了用来定义分区的范围的上限值(High_Value)和表中分区的位置(Partition_Position)。

分区键使用的列可以通过 USER_PART_KEY_COLUMNS 数据字典视图访问。USER_PART_KEY_COLUMNS 数据字典视图只包含 4 列：

- Name　　分区表或索引的名称
- Object_Type　　对象类型(TABLE 或 INDEX)
- Column_Name　　属于分区键的列名
- Column_Position　　在分区键中列的位置

分区列的统计信息可以通过 USER_PART_COL_STATISTICS 数据字典视图进行访问。USER_PART_COL_STATISTICS 数据字典视图中的列反映了 USER_TAB_COL_STATISTICS 数据字典视图中的列。

分区数据的直方图信息可以通过 USER_PART_HISTOGRAMS 数据字典视图进行访问。该视图的列显示了直方图中每个存储段的终点值。该视图的列包括 Table_Name、Partition_Name、Column_Name、Bucket_Number、Endpoint_Value 以及 Endpoint_Actual_Value。

因为索引可以分区，所以 USER_IND_PARTITIONS 数据字典视图可用。USER_IND_PARTITIONS 数据字典视图中的列与 USER_INDEXES 视图中的列类似，只是标识列稍有不同。分区索引的标识列包括了分区名、分区的上限值和分区在表中的位置。当分析分区时填充与统计信息相关的列。与空间相关的列描述了分配给索引的空间，关于存储参数的信息，请参阅附录 A 中的“STORAGE”条目。

如果一个分区有子分区，则可通过数据字典视图查看子分区的详细情况。USER_IND_SUBPARTITIONS 数据字典视图包含了 USER_IND_PARTITIONS 数据字典视图中与空间相关的和与统计信息相关的列，以及标识子分区的列(Subpartition_Name 和 Subpartition_Position)。类似地，USER_TAB_SUBPARTITIONS 数据字典视图也包含了 USER_TAB_PARTITIONS 数据字典视图中与空间相关的列和与统计信息相关的列，以及 Subpartition_Name 列和 Subpartiton_Position 列。这样，就可以确定每个分区和子分区的空间定义了。

如本节前面所示，可查询 USER_PART_COL_STATISTICS 数据字典视图和 USER_PART_HISTOGRAMS 数据字典视图数据字典视图，来查看关于分区的统计信息。对于子分区，可以查询 USER_ SUBPART _COL_STATISTICS 数据字典视图和 USER_SUBPART_HISTOGRAMS 数据字典视图，它们的结构反映了分区统计视图的结构。要查看子分区键列，可查询 USER_SUBPART_
KEY_COLUMNS 数据字典视图，其列结构与 USER_PART_KEY_COLUMNS 数据字典视图的列结构相同。

45.12.5　可用空间：USER_FREE_SPACE

除了查看已用空间外，还可以查询数据字典，来查看当前有多少可用空间。USER_FREE_SPACE 视图列出了用户可以访问的所有表空间中的可用区。它按照 Tablespace_Name 列出了 File_ID、Block_ID 和可用区起始点的相对文件号。可用区的大小以字节和块为单位列出。DBA_FREE_SPACE 视图经常被 DBA 用来监测可用空间数量和空间分割的程度。

45.13　用户和权限

用户及其权限记录在数据字典中。下面几节将介绍如何查询数据字典，以获取用户账户、资源限制和用户权限的信息。

45.13.1　用户：USER_USERS

可以查询 USER_USERS 视图，列出自己的账户信息。USER_USERS 视图包含了 Username、User_ID(由数据库分配的一个号码)、Default_Tablespace、Temporary_Tablespace 和创建日期(创建账户的日期)。USER_USERS 视图的 Account_Status 列显示了账户的状态，即它是处于锁定、解锁(OPEN)还是过期状态。如果账户被锁定，则 Lock_Date 列将显示锁定账户的日期，Expiry_Date 列将显示账户到期的日期。也可以查询 DBA 分配给用户的资源用户组(Initial_Rsrc_ Consumer_Group)信息和 External_Name 值。

虽然 ALL_USERS 视图只包含 USER_USERS 视图中 Usemame 列、User_ID 列和 Created 列，但是它列出了数据库中所有账户的信息。在需要知道可用的用户名时(例如，在 grant 命令中)，ALL_USERS 视图非常有用。DBA_USERS 视图包含 USER_USERS 视图中的全部列，以及两个附加的列：Password(账户的加密口令)和 Profile(用户的资源预置文件)。DBA_USERS 视图列出了数据库中所有用户的这些信息。

45.13.2　资源限制：USER_RESOURCE_LIMITS

在 Oracle 中，配置文件用来存放用户可用的系统资源和数据库资源数量的限制。如果在数据库中没有创建配置文件，则将使用默认的配置文件，它为所有用户指定无限制的资源。可限制的资源在附录 A 中的“create profile”项中介绍。配置文件施加了额外的安全措施，如账户的到期日期和口令的最小长度等。

要查看当前会话中的限制，可以查询 USER_RESOURCE_LIMITS 视图。其列如下：

- Resource_Name 资源名(如 SESSIONS_PER_USER)
- Limit 在该资源上的限制

USER_PASSWORD_LIMITS 视图描述了用户的口令配置文件参数。它具有与 USER_RESOURCE_ LIMITS 视图相同的列。

此视图没有“ALL”版本和“DBA”版本。此视图仅限于用户的当前会话。要查看每个可用资源的相关成本，可以查询 RESOURCE_COST 视图。DBA 可以访问 DBA_PROFILES 视图，以查看所有配置文件的资源限制。DBA_PROFILES 视图中的 Resource_Type 列指出了资源配置文件为 PASSWORD 配置文件还是 KERNEL 配置文件。

45.13.3 表的权限：USER_TAB_PRIVS

要查看被授予者、授予者或对象所有者的权限，可以查询 USER_TAB_PRIVS 视图(用户表权限)。除了 Grantee 列、Grantor 列和 Owner 列外，此视图所包含的列还有 Table_Name、Hierarchy、Privilege 和一个标志(设置为 YES 或 NO)，该标志用来指出权限是否可以用 with admin option(Grantable)授予。

USER_TAB_PRIVS_MADE 视图显示了用户就是所有者的 USER_TAB_PRIVS 记录(因此缺少 Owner 列)。USER_TAB_PRIVS_RECD 视图(接收的用户表权限)显示了用户为被授权者的 USER_TAB_PRIVS 记录(因此缺少 Grantee 列)。由于 USER_TAB_PRIVS_MADE 视图和 USER_TAB_ PRIVS_RECD 视图都是 USER_TAB_PRIVS 视图的子集，因此可以用一个相应的 where 子句查询 USER_ TAB_PRIVS 视图，从而简单地复制其功能，以查看相应的子集。

USER_TAB_PRIVS 视图、USER_TAB_PRIVS_MADE 视图和 USER_TAB_PRIVS_RECD 视图都有“ALL”版本。“ALL”版本列出了用户或 PUBLIC 为被授予者或授予者的那些对象。USER_TAB_PRIVS 视图有一个名为 DBA_TAB_PRIVS 的“DBA”版本，它列出了授权给数据库中所有用户的所有对象权限。DBA_TAB_PRIVS 视图具有与 USER_TAB_PRIVS 视图相同的列定义。ALL_TAB_PRIVS 视图没有 Owner 列，而具有显示对象模式的 Table_Schema 列。

45.13.4 列权限：USER_COL_PRIVS

除了在表上进行授权外，还可以在列级别授权。例如，可以授予用户只更新表中某些列的权限。详细信息，请参阅第 19 章和附录 A 中的“grant”命令。

用于显示列权限的数据字典视图在设计上与上一节所介绍的表权限视图几乎相同。每个 COL 视图中都有一个附加列 Column_Name，而没有 Hierarchy 列。USER_COL_PRIVS 视图类似于 USER_TAB_PRIVS 视图，USER_COL_PRIVS_MADE 视图类似于 USER_TAB_PRIVS_ MADE 视图，而 USER_COL_PRIVS_RECD 视图类似于 USER_TAB_PRIVS_RECD 视图。

所有的列权限视图都有“ALL”版本。DBA_COL_PRIVS 视图列出了授予数据库中用户的所有列权限(就像 DBA_TAB_PRIVS 视图列出了授予用户的所有表权限一样)。

45.13.5　系统权限：USER_SYS_PRIVS

USER_SYS_PRIVS 视图列出已经授予用户的系统权限。它的列包括 Username、Privilege 和 Admin_Option(设置为 YES 或 NO 的一个标志，用来指出是否用 with admin option 授予权限)，直接授予用户的所有系统权限都可以通过该视图显示。通过角色授予用户的系统权限不能在此视图中显示。下面的查询显示了 PRACTICE 账户的示例输出：

```
select Username, Privilege * from USER_SYS_PRIVS;

USERNAME                       PRIVILEGE
------------------------------ ----------------------
PRACTICE                       CREATE DIMENSION
PRACTICE                       UNLIMITED TABLESPACE
```

此视图没有“ALL”版本。要查看授予数据库中所有用户的系统权限，可以查询 DBA_SYS_PRIVS 视图，它与 USER_SYS_PRIVS 视图具有几乎相同的列定义(“USER”版本有 Username 列，而“DBA”版本有 Grantee 列)。

45.14　角色

除了直接授予用户的权限外，还可以将权限集分成角色。角色可以授予用户或其他角色，也可以由对象和系统权限组成。关于角色使用和管理的信息，请参阅第 19 章。

要查看已经授予了用户哪些角色，可以查询 USER_ROLE_PRIVS 数据字典视图。已经授予 PUBLIC 的任何角色都会在此列出。USER_ROLE_PRIVS 数据字典的视图的可用列如下：

- Usemame　用户名(可能为 PUBLIC)
- Granted_Role　授予用户的角色名
- Admin_Option　值为 YES 或 NO 的一个标志，用来指出是否用 with admin option 授权角色
- Default_Role　值为 YES 或 NO 的一个标志，用来指出角色是否为用户的默认角色
- OS_Granted　值为 YES 或 NO 的一个标志，用来指出操作系统是否用于管理角色

要列出数据库中所有可用的角色，首先必须具有 DBA 权限；然后才可以查询 DBA_ROLES 视图，列出所有角色。DBA_ROLE_PRIVS 视图列出了分配给数据库中所有用户的角色。

角色可以接受 3 种类型的授权，并且每一种授权都有一个相应的数据字典视图：

- 表/列授权　ROLE_TAB_PRIVS 数据字典视图。类似于 USER_TAB_PRIVS 数据字典视图和 USER_COL_PRIVS 数据字典视图，只是 ROLE_TAB_PRIVS 数据字典视图有一个 Role 列而不是 Grantee 列，而且没有 Grantor 列
- 系统权限　ROLE_SYS_PRIVS 数据字典视图。类似于 USER_SYS_PRIVS 数据字典视图，只是 ROLE_SYS_PRIVS 数据字典视图有一个 Role 列而没有 Username 列
- 角色授权　ROLE_ROLE_PRIVS 数据字典视图。列出已经授予其他角色的所有角色

注意：
如果用户不是 DBA，则这些数据字典视图只能列出授予用户的角色。

除了这些视图外，还有两个视图，它们都只有一列，列出了当前会话启用的权限和角色：

- SESSION_PRIVS　Privilege 列列出了会话可用的所有系统权限，不管是直接授予的还是通过角色授权的
- SESSION_ROLES　Role 列列出了当前会话启用的所有角色

SESSION_PRIVS 视图和 SESSION_ROLES 视图对所有用户都可用。

注意：
关于启用和禁用角色以及设置默认角色的详细内容，请参阅第 19 章。

45.15 审计

作为 Oracle 数据库中的非 DBA 用户，用户不能启用数据库的审计功能。但是，如果已经启用了审计功能，就存在任何人都可以用来查看审计跟踪的数据字典视图。

许多不同的审计跟踪数据字典视图都可用。这些视图的大部分都基于数据库中单个审计跟踪表(SYS.AUD$)。最常见的审计跟踪视图为 USER_AUDIT_TRAIL。由于该视图显示了许多不同行为的审计记录，因此有许多列并非对任意行都适用。此视图的“DBA”版本(DBA_AUDIT_TRAIL)列出了审计跟踪表中的所有项，USER_AUDIT_TRAIL 视图只列出了与用户相关的那些项。

大量审计功能都可用(请查阅附录 A 中的 audit 命令)。每一类审计功能都可以通过其自身的数据字典视图进行访问。可用的视图如下：

- USER_AUDIT_OBJECT　用于涉及对象的语句
- USER_AUDIT_SESSION　用于连接和断开连接
- USER_AUDIT_STATEMENT　用于用户发布的 grant、revoke、audit、noaudit 和 alter system 命令

上述列表中的每个“USER”视图都有对应的“DBA”版本，“DBA”版本显示了适合相应视图类型的所有审计跟踪记录。

查询 USER_OBJ_AUDIT_OPTS 视图，可以浏览当前对对象有效的审计选项。对于在 USER_OBJ_AUDIT_OPTS 视图中列出的每个对象来说，每个命令的审计选项都可在 USER_OBJ_AUDIT_OPTS 视图列出的对象(由 Object_Name 列和 Object_Type 列标识)上执行。USER_OBJ_AUDIT_OPTS 视图的列名对应于相应命令的前 3 个字母(例如，Alt 表示 alter，Upd 表示 update 等)。每一列将记录在命令成功(S)、不成功(U)或既有成功也有失败时是否为该对象审计命令。通过 ALL_DEF_AUDIT_OPTS 视图，可以显示数据库中对任何新对象有效

的默认审计选项，该视图与 USER_OBJ_AUDIT_OPTS 视图有相同的列命名约定。USER_OBJ_AUDIT_OPTS 视图包括 Object_Name 列和 Object_Type 列。

可审计的命令存储在名为 AUDIT_ACTIONS 的引用表中，它有两列，即 Action(动作的数字代码)和 Name(动作或命令的名称)。Action 列和 Name 列对应于 USER_AUDIT_TRAIL 视图中的 Action 列和 Action_Name 列。

DBA 能使用几个另外的、没有“USER”对应版本的审计视图，其中包括 DBA_AUDIT_EXITS、DBA_PRIV_AUDIT_OPTS、DBA_STMT_AUDIT_OPTS 和 STMT_AUDIT_OPTION_MAP。关于这些只属于 DBA 的视图的详细信息，请参阅 *Oracle Database Reference* 一书。

45.16　其他视图

除了本章前面介绍的数据字典视图外，在数据字典中还有其他一些视图和表。这些视图和表包括只属于 DBA 的视图和 explain plan 命令执行时使用的表。下面几节将简单介绍其他类型的视图。

45.17　监控：V$动态性能表

监控数据库环境性能的视图称为动态性能视图(dynamic performance view)。动态性能视图通常被称为 V$表，因为它们都是以字母 V 后跟一个美元符号($)开头。

监控视图中列的定义和用法依赖于每个数据库版本的变化。正确解释视图的即席查询的结果通常需要参考 *Oracle Database Administrator's Guide*。V$表通常只由 DBA 使用。关于 V$视图的详细信息，请参阅 *Oracle Database Reference* 和 *Oracle DBA Handbook*。

45.17.1　CHAINED_ROWS

analyze 命令用于生成表内链接的或迁移的行的列表。该链接行的列表可以存储在 CHAINED_ROWS 表内。要在用户模式中创建 CHAINED_ROWS 表，需要运行 utlchain.sql 脚本(通常可在 Oracle 主目录下的/rdbms/admin 子目录中找到)。

要填充 CHAINED_ROWS 表，需要使用 analyze 命令的 list chained rows into 子句，如下面的程序清单所示：

```
analyze TABLE BIRTHDAY list chained rows into CHAINED_ROWS;
```

CHAINED_ROWS 表列出了 Owner_Name、Table_Name、Cluster_Name(如果该表在群集中)、Partition_Name(如果表被分区)、Subpartition_Name(如果表包含子分区)、Head_RowID(该行的 ROWID)和 Analyze_TimeStamp 列(该列显示最后一次分析表或群集的时间)。可以在 CHAINED_ROWS 表中查询基于 Head_RowID 值的表，如下例所示：

```
Select * from BIRTHDAY
    where RowID in
          (select Head_RowID
```

```
from CHAINED_ROWS
where Table_Name = 'BIRTHDAY');
```

如果链接的行长度很短，那么可以通过删除和重新插入行的操作来消除该链接。

45.17.2 PLAN_TABLE

在调整 SQL 语句时，用户可能希望确定优化程序执行查询的步骤。要查看该查询的路径，必须首先在名为 PLAN_TABLE 的模式中创建一个表。用于创建该表的脚本称为 utlxplan.sql，它通常存储在 Oracle 软件主目录下的/rdbms/admin 子目录中。

在模式中创建 PLAN_TABLE 表后，可以使用 explain plan 命令(它将在 PLAN_TABLE 表中生成记录)为希望解释的查询添加指定的 Statement_ID 值。关于生成 explain plan 列表的细节，请参阅第 46 章。

PLAN_TABLE 表中的 ID 列和 Parent_ID 列确定了执行查询时优化程序将遵循的步骤(Operations)的层次结构。关于 Oracle 优化程序和 PLAN TABLE 记录的解释，请参阅第 46 章。

45.17.3 相互依赖性：USER_DEPENDENCIES 和 IDEPTREE

Oracle 数据库中的对象可以互相依赖。例如，一个存储过程可能依赖于一个表，或者一个程序包可能依赖一个程序包体。当数据库中的对象发生变化时，依赖它的任何过程对象都必须重新编译。重新编译可以在运行时自动进行(会产生相应的性能损失)或者手动进行(编译过程对象的详细信息，请参阅第 35 章)。

有两组数据字典视图可用于帮助跟踪依赖关系。第一组为 USER_DEPENDENCIES，它列出了对象的所有直接依赖关系。然而，这仅仅是进入了依赖关系树中的一层。为充分判定依赖关系，必须在模式中创建递归的依赖关系跟踪对象。要创建这些对象，需要运行 utldtree.sql 脚本(通常位于 Oracle 主目录下的/rdbms/admin 子目录中)。此脚本创建两个可以查询的对象：DEPTREE 和 IDEPTREE。虽然它们包含相同的信息(虽然它们的列定义不同)，但是由于 IDEPTREE 基于伪列级别缩进排列，因此更容易阅读和解释。

45.17.4 只属于 DBA 的视图

由于本章面向开发人员和最终用户，因此只有 DBA 可用的数据字典视图不是本章所讨论的内容。只属于 DBA 的视图通常用来提供分布式事务、锁争用、回滚段和其他内部数据库函数的信息。关于只属于 DBA 的视图的使用信息，请参阅 *Oracle Database Administrator's Guide* 一书。

45.17.5 Oracle Label Security

Oracle Label Security 的用户可以查看其他数据字典视图，包括 ALL_SA_GROUPS、ALL_SA_POLICIES、ALL_SA_USERS 和 ALL_SA_USER_PRIVS。关于这些视图的详细用法，请参阅 *Oracle Label Security Administrator's Guide* 一书。

45.17.6　SQL*Loader 直接加载视图

为了在 SQL*Loader 中管理直接加载选项，Oracle 维护许多数据字典视图。查询这些视图通常是出于调试目的，这依赖于 Oracle Customer Support 的请求。SQL*Loader 直接加载选项将在附录 A 中的“SQLLDR”项中介绍，它支持的数据字典视图如下所示：

- LOADER_COL_INFO
- LOADER_CONSTRAINT_INFO
- LOADER_FILE_TS
- LOADER_PARAM_INFO
- LOADER_PART_INFO
- LOADER_REF_INFO
- LOADER_TAB_INFO
- LOADER_TRIGGER_INFO

关于这些视图的用法的详细信息，请参阅 catldr.sql 脚本，它通常位于 Oracle 主目录下的/rdbms/admin 子目录中。

45.17.7　全球支持视图

有 3 个数据字典视图用于显示数据库中当前有效的 Globalization Support 参数的信息。NLS 参数(如 NLS_DATE_FORMAT 和 NLS_SORT)的非标准值可以通过数据库的参数文件或通过 alter session 命令设置(关于 NLS 设置的详细信息，请参阅附录 A 中的 alter session 命令)。为了查看会话、实例和数据库的当前 NLS 设置，可以分别查询 NLS_ SESSION _ PARAMETERS、NLS_INSTANCE_PARAMETERS 和 NLS _ DATABASE _ PARAMETERS。

45.17.8　库

PL/SQL 例程(请参阅第 35 章)能够调用外部 C 程序。要查看自己拥有哪些外部 C 程序库，可以查询 USER_LIBRARIES 视图，它显示了库的名称(Library_Name)、相关文件(File_Spec)、库是否可动态加载(Dynamic)，以及库的状态(Status)。ALL_LIBRARIES 视图和 DBA_ LIBRARIES 视图也可用，它们包含一个附加的 Owner 列来指出库的所有者。关于库的详细信息，请参阅附录 A 中的 create library 命令。

45.17.9　异构服务

为支持异构服务的管理，Oracle 提供了许多数据字典视图。这种类型的所有视图都以字母 HS 而非 DBA 开头。通常，这些视图主要由 DBA 使用。关于 HS 视图的详细信息，请参阅 *Oracle Database Reference* 一书。

45.17.10　索引类型和运算符

运算符和索引类型密切相关。可以使用 create operator 命令创建新的运算符，并定义其绑定情况。可以在索引类型和 SQL 语句中引用运算符。运算符可以依次引用函数、程序包、类

型和用户定义的其他对象。

可以查询 USER_OPERATORS 视图来查看每个运算符的 Owner、Operator_Name 和 Number_of_Binds 值。运算符的辅助信息可以通过 USER_OPANCILLARY 视图进行访问，并且可以查询 USER_OPARGUMENTS 视图来查看运算符的参数。可查询 USER_OPBINDINGS 视图来查看运算符的绑定情况。

USER_INDEXTYPE_OPERATORS 视图列出了索引类型支持的运算符。而索引类型可以通过 USER_INDEXTYPES 视图显示。所有的运算符视图和索引类型视图都有“ALL” 版本和“DBA”版本。可以通过 USER_INDEXTYPE_ARRAYTYPES 视图来查看索引类型的数组类型。

45.17.11 概要

使用存储概要时，可以通过 USER_OUTLINES 数据字典视图，检索其名字和细节。要查看构成概要的提示，可以查询 USER_OUTLINE_HINTS 视图。USER_OUTLINES 视图和 USER_OUTLINE_HINTS 视图都有“ALL”版本和“DBA” 版本。

45.17.12 顾问程序

用户可以访问与优化建议相关的数据字典视图。这些视图以“USER_ADVISOR_”开头，并在“DBA_”级有一组匹配的视图，还有仅属于 DBA 的顾问程序视图(如 DBA_ADVISOR_DEFINITIONS)。

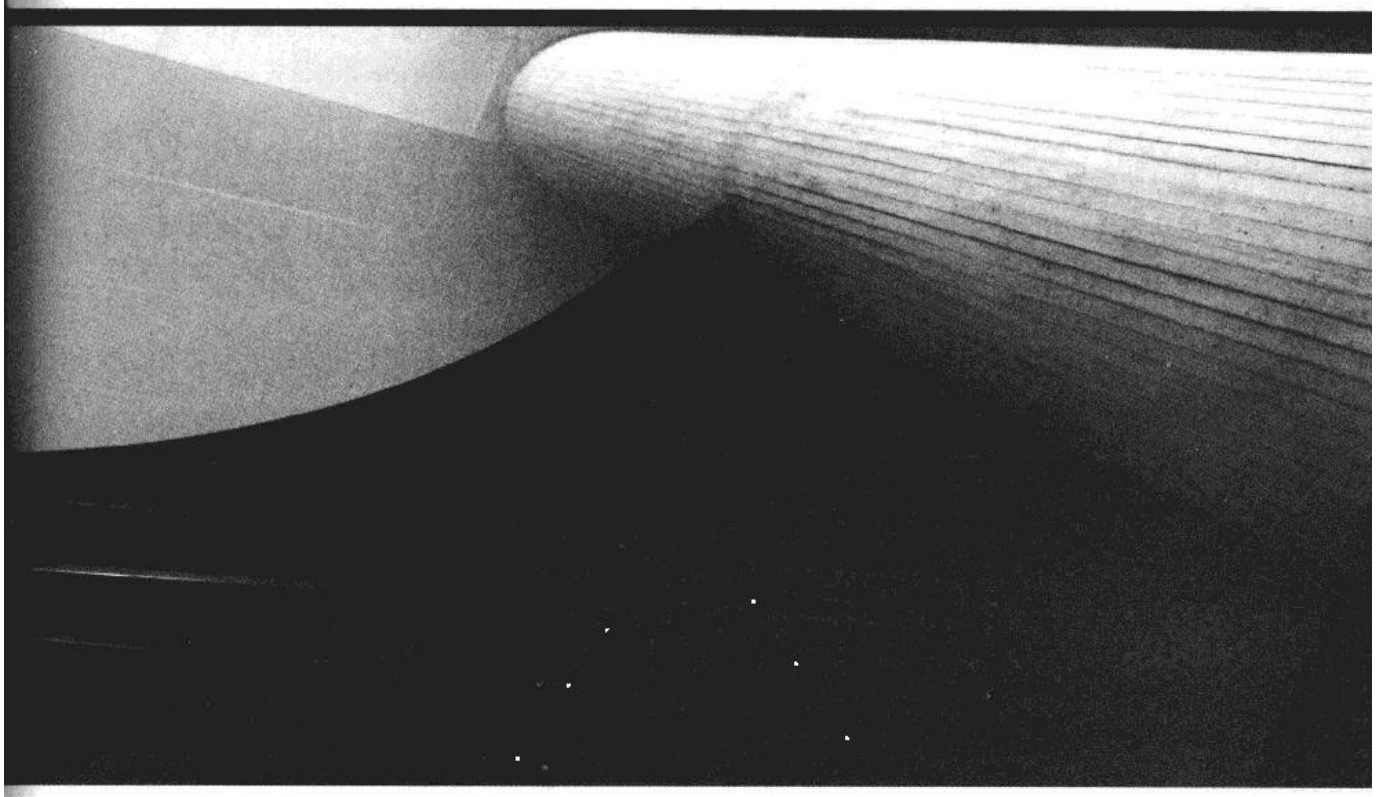
例如，USRE_ADVISOR_ACTIONS 视图包含数据库中与所有建议相关联的动作的数据。每一个动作由 Command 列与 6 个相关属性列指定。USER_ADVISOR_LOG 视图显示关于所有任务当前状态的信息以及关于执行情况的数据(如进程监视和完成状态)。顾问程序的参数可通过 USER_ADVISOR_ PARAMETERS 视图来访问；可以通过 USER_ADVISOR_RATIONALE 视图查看建议的理由。

关于数据库中所有建议的分析结果，可以查看 USER_ADVISOR_RECOMMENTDATIONS 视图。建议可以有与其相关的多个动作(请参阅 USER_ADVISOR_ACTIONS 视图)，这取决于使用的理由。

Oracle 11g 新增的与顾问程序相关的数据字典视图的完整列表请参见表 45-1。对于绝大多数用户来说，这些视图只通过 Oracle 企业管理器(Oracle Enterprise Manager)或其他图形界面来访问。

45.17.13 调度程序

您可以访问一组与数据库中调度程序作业相关的视图。这些视图包括 USER_SCHEDULER_JOBS(用户拥有的所有调度程序作业)、USER_SCHEDULER_PROGRAMS(调度程序)和 USER_SCHEDULER_PROGRAM_ARGS(调度程序的参数)。可以通过 USER_SCHEDULER_JOB_LOG 视图、USER_SCHEDULER_JOB_RUN_DETAILS 视图和 USER_SCHEDULER_ RUNNING_JOBS 视图来查看作业的执行细节。可以通过 USER_SCHEDULER_SCHEDULES 视图来查看调度情况。



第46章

应用程序和SQL调整指南

Oracle在其每次新发布的产品中都添加了新的功能，以增强应用程序的性能。这些功能包括必须手动实现的新选项(如分区)，或者内在的改动(不需要对应用程序进行任何修改，就可以提高性能)。

不要在应用程序投入生产或在系统测试完成之后才开始优化调整。如所看到的那样，应用程序的性能规划必须在设计过程中完成。

本章不仅描述了Oracle Database 11g引入的主要新增调整修改，而且介绍了与调整Oracle应用程序相关的一组准则和最佳实践。本章最后一节将这些功能与推荐的最佳实践结合起来。此外，本章还介绍了生成和解释说明计划(explain plan)和存储概要(stored outline)的准则。

46.1 Oracle Database 11g 新增的调整功能

Oracle Database 11g 引入了许多新增功能，用于简化和自动完成调整过程。DBA 和开发人员以前需要手动完成的调整步骤，现在数据库可以自动地完成这些步骤。注意：数据库不能重新设计应用程序；它只是尽可能快地执行接收到的命令。关于性能设计的经验原则，请参阅 46.3 节。因为本书面向所有用户，所以没有关注特定于 DBA 的工具；相反，本书关注如何设计和开发性能易于管理的系统。

下面与性能相关的功能是 Oracle Database 11g 新增的功能。与调整相关的所有功能的详细信息，请参阅 *Performance Tuning Guide*(来自 Oracle 文档集)。

46.2 Oracle 11g 新增的调整功能

Oracle Database 11g 版本 1(11.1)新增的和更新的性能调整功能包括：

- 活动会话历史记录(ASH)增强——增强 ASH 统计信息，用来提供被捕获的每条 SQL 语句的行级活动信息。这些信息可用来识别执行哪部分 SQL 时占用的时间最多。ASH 视图提供了关于数据库中活动会话的历史信息，DBA 可以访问这一视图。
- 数据库自动诊断监视(ADDM)增强——DBA 和开发人员可以使用 ADDM 分析数据库，并生成性能调整建议。ADDM 功能在 Oracle 11g 中得到了增强，它可以在指定的任何时间段内，在各种不同的粒度级别(例如，数据库群集、数据库实例或特定目标)上对数据库群集进行分析。
- SQL 自动调整——DBA 可以使用 SQL Tuning Advisor 生成调整 SQL 命令的建议。可以在维护窗口期间使用自动维护任务(Automated Maintenance Task)自动调度要运行的 SQL Tuning Advisor 任务。
- AWR 基线——自动的工作负荷存储库(AWR)报表提供了数据库的基线统计信息，并显示了特定时间间隔内数据库的性能。基线包括特定时间段的性能数据，保存这些数据是为了在出现性能问题时与其他相似的工作负荷时间段进行比较。在 Oracle Database 中几种可用的基线类型是：修正的基线、移动窗口基线和基线模板。
- 数据库重放——可以在产品系统上捕获数据库工作负荷，并在测试系统上重放它，以确保系统变化(例如，数据库升级)会产生期望的结果(参见第 31 章)。
- 增强了 I/O 统计信息——针对 Oracle Database 执行的所有 I/O 调用，在使用者组、数据库文件和数据库功能等几个维度上，收集 I/O 统计信息。
- 增强了对优化程序统计信息的维护——统计信息的收集和发布现在是分开进行的。可以收集优化程序统计信息，将它们存储为挂起的统计信息，测试它们，然后只在统计信息有效时发布它们。
- 扩展了统计信息——优化程序现在收集表中一组列的信息(多列的统计信息)，以便分析列组中列的依赖关系。优化程序也收集关于列中表达式的信息。

- I/O 标准化——使用 I/O 标准化功能可以评估存储子系统的性能，并确定 I/O 性能问题是否是由数据库或存储子系统引起的。
- 查询结果缓存——可以将频繁运行的 SQL 查询的结果存储在服务器端和客户机端的结果缓存中。这可以确保以后执行这些查询时有更快的响应时间(参见第 47 章)。
- 实时 SQL 监控——实时 SQL 监控功能允许在执行长期运行的 SQL 语句时对它们进行监控。游标级别的统计信息(例如，CPU 时间和 I/O 时间)和执行计划统计信息(例如，中间结果的基数、内存以及计划中每个操作符使用的临时空间)几乎在语句执行时实时更新。这些统计信息由 V$SQL_MONITOR 和 V$SQL_PLAN_MONITOR 这两个新修正的视图来提供。
- SQL 性能分析器——通过在测试系统上使用 SQL 工作负荷来测试系统变化，测试 SQL 性能分析器可使 DBA 预测这些系统变化对 SQL 性能的影响。
- SQL 计划基线——改变 SQL 语句的执行计划会导致服务器性能的下降。Oracle Database 可以捕获、选择和完善 SQL 计划基线，在验证新计划比现有计划效率更高之前，阻止优化程序使用新计划。此功能取代了以前 Oracle Database 版本中的计划稳定性功能。
- SQL 测试用例生成器——SQL 测试用例生成器提供了生成可再现的测试用例的功能，实现这些功能的方式是收集和打包与 SQL 事件相关的必要信息，以便可以在另一个系统上再现这些信息。

46.3　调整—— 最佳实践

保守估计，至少 50%的时间用于调整应用程序的性能问题。在应用程序和相关数据库结构的设计期间，应用程序体系结构设计人员可能不知道随着时间的过去，企业将使用应用程序数据的所有方式。因此，最初阶段总有一些组件的性能很差，而随着企业使用应用程序的方式发生变化，还会出现一些其他问题。

在某些情况下，修正相对简单——更改初始化参数，添加索引，或重新调度大型操作。在另一些情况下，必须改变应用程序的体系结构才能修正该问题。例如，对于所有的数据访问，应用程序可能大量地重用函数——因此即使执行最简单的数据库操作，函数也会调用其他函数，其他函数又调用更多的函数。结果，单个数据库调用可能导致成千上万个函数调用和数据库访问。这样的应用程序通常不能很好地扩展；当更多用户添加到系统中时，每个用户执行的负载量将降低各个用户的性能。调整应用程序执行的个别 SQL 语句，几乎没有任何性能提高(可能已经很好地调整了语句本身)。相反，导致性能问题的原因是执行的负载量过多。

下面的最佳实践可能看起来过于简单——但它们在数据库应用程序中多次违反规则，这些违规直接导致性能问题。规则总有例外，因为软件或环境接下来的更改可能允许在不影响性能的情况下违反规则。但是，一般而言，遵循这些规则将在扩展应用程序的用途时满足性能需求。

46.3.1 尽可能少做

一般而言，最终用户通常不会关心基本数据库结构是否完全符合第三范式，或其布局是否遵循面向对象的标准。用户想执行业务过程，而数据库应用程序应作为一个工具，该工具用来帮助业务过程尽快完成。设计的重心不应追求完美的理论设计；而应提高最终用户完成作业的能力。应用程序的每一步涉及的过程都要简化。

1. 在应用程序设计中，尽量消除逻辑读

过去，主要关注于消除物理读——现在仍然是个好主意，除非逻辑读需要物理读，否则不使用物理读。

请看一个简单的示例。从 DUAL 选择当前时间。如果选择到秒级，则该值每天将修改 86 400 次。然而很多应用程序设计人员重复地执行该查询，每天执行上百万次。这样一个查询几乎一整天可能都不执行物理读——因此如果仅仅关注于调整物理 I/O，则可能会忽视它。但是，它严重地影响应用程序的性能。怎样影响的呢？通过使用可用的 CPU 资源。每次执行查询，都将迫使 Oracle 执行任务，使用处理功能查询并返回正确的数据。随着越来越多的用户不断地执行该命令，该查询所使用的逻辑读次数将超过所有其他查询。在某些情况下，服务器上的多处理器致力于服务这种类型的重复的简单查询。它们产生什么商业利益呢？完全没有。

考虑下面这个现实中的示例。程序员想在程序中实现一个暂停，在两个步骤完成之间等待 30 秒。由于环境的性能随着时间的推移不可能保持一致，因此编程人员需要编写如下格式的例程(用伪代码显示)：

```
perform Step 1
select SysDate from DUAL into a StartTime variable
begin loop
   select SysDate from DUAL in a CurrentTime variable;
   Compare CurrentTime with the StartTime variable value.
   If 30 seconds have passed, exit the loop;
      Otherwise repeat the loop, calculating SysDate again.
end loop
perform Step 2.
```

这绝对不是个合理的方法！虽然它可以实现开发人员所期望的功能，但应用程序的系统开销很大，而且数据库管理员根本不能改进其性能。在这种情况下，系统开销不是由 I/O 活动造成的——DUAL 表将驻留在实例的内存区域——而是由 CPU 活动造成的。每个用户每次运行该程序时，数据库将花费 30 秒时间使用系统能支持的很多 CPU 资源。在这种特殊情况下，select SysDate from DUAL 查询占用了应用程序使用的全部 CPU 时间的 40%以上。这些 CPU 时间都被浪费了。调整个别 SQL 语句没有任何帮助，必须重新设计应用程序，消除不必要执行的命令。更好的解决方案是设计一种方法来跟踪第一步和随后的操作的完成；然后启动第二步，不计算嵌入等待时间。

对于喜欢基于缓冲区缓存器使用率进行调整的人来说，该数据库的使用率基本上是 100%，这是由于大量完全不必要的没有相关物理读的逻辑读操作造成的。缓冲区缓存器使用率比较逻辑读与物理读的次数；如果 10%的逻辑读需要物理读，则缓冲区缓存器使用率是 90%。低使用率标识执行大量物理读操作的数据库；而本示例中的非常高的使用率标识执行了过多的逻辑读操作的数据库。

2. 在应用程序设计中，尽量避免访问数据库

记住：是在调整应用程序，而不是调整查询。可能需要把多个查询合并为一个过程，这样，每个屏幕就可以一次(而不是多次)访问数据库。绑定的查询方法尤其与依赖多个应用程序层的“瘦客户端”应用程序相关。寻找基于其返回值且相互关联的查询，并注意是否有机会将它们转换为单个代码块。这样做的目的不是生成一个永远也不能完成的单块查询，而是为了避免不必要的工作。在这种情况下，数据库服务器、应用程序服务器和最终用户的计算机之间定期频繁的通信就是需要调整的对象。

在复杂的数据输入格式(显示在屏幕中的每个字段通过单个查询填充)中常遇见这个问题。每个查询都单独地访问数据库。如前一节的示例所示，这促使数据库执行大量相关的查询。即使调整每个查询，命令(命令数量乘以用户数量)的负荷也将消耗服务器上可用的 CPU 资源。虽然这样的设计也影响网络的使用，但是网络通常不是问题——问题是访问数据库的次数。

在程序包和过程中，应当尽量消除不必要的数据库访问。把常用的值存储在本地变量中，而不是反复地查询数据库。如果不必访问数据库来获取信息，就不要访问它。虽然这个建议听起来简单，但令人惊讶的是应用程序开发人员经常不考虑这一建议。

没有任何初始化参数能使该更改生效。它是个设计问题，在应用程序性能规划和调整过程中，需要开发人员、设计人员、DBA、应用程序的用户共同努力。

3. 对于报表系统，按照用户查询的方式存储数据

如果知道将要执行的查询(如通过参数化的报表)，就应该尽量按照呈现给用户的格式存储数据。这样，Oracle 几乎不需要做什么工作就可以将表中的数据格式转换成用户看到的格式。这可能需要创建并维护物化视图或报表。维护物化视图或报表显然是数据库额外完成的工作——但它以批处理的方式完成，不直接影响最终用户。另一方面，最终用户受益于这种更快的查询能力。因为与最终用户查询视图相比，访问基本表来填充和更新物化视图的执行次数很少，所以数据库整体上将执行较少的逻辑读和物理读操作。

4. 避免反复连接数据库

打开数据库连接可能是一个速度很慢的操作，而且在很大程度上是可以避免的。如果需要连接数据库，则保持连接处于打开状态，并重用该连接。在应用程序级，可以使用连接池支持该需求。在数据库中，在执行处理操作时，可以使用存储过程、程序包和其他方法保持连接。

另一个现实中的示例：应用程序设计人员想要验证在执行报表之前，数据库已在运行。解决方案是打开会话并执行下面查询：select count(*) from DUAL。如果查询返回正确的结果(1)，那么数据库正在运行，将打开新连接，并执行报表查询。这种方法有什么错误呢？在小型系统中，这种设计是可以的。在具有大量并发用户的 OLTP 系统中，因为数据库大部分的时间消耗在打开关闭连接上，所以用户将会遇到重大的性能问题。在数据库中，select count(*) from DUAL 这一查询每天将执行上百万次——Oracle 将把大量应用程序可用资源消耗在打开和关闭连接以及把数据库的状态返回给应用程序上。虽然查询几乎不执行 I/O，但其影响表现在它所占用的 CPU 和不断地打开连接这两个方面。

为什么需要这样的一步呢？如果正确地处理报表查询本身的错误，那么显然发生故障时数据库连接不能正确运行。因为不能重用相同的连接，所以不必要的数据库可用性检查会使情况更坏。DBA 不能改正它。必须重新设计应用程序，这样才能正确地重用连接。

5. 使用正确的索引

为消除物理读操作，有些应用程序开发人员在每个表上创建了很多索引。除了它们对数据加载时间的影响(在 46.3.6 节中讨论)，可能很多索引永远也用不到。在 OLTP 应用程序中，不能使用位图索引；如果列几乎没有不同的值，则考虑不要为它创建索引。优化程序支持跳跃式扫描索引访问，即使索引的第一列不是查询的限制条件，也可以在一组列中使用一个索引。

46.3.2 尽可能简单地完成

既然已经消除了不必要的逻辑读操作、不必要的数据库访问、难以管理的连接和不合适的索引等造成的性能开销，下面就讨论其余的事项。

1. 变为原子组件

可以使用 SQL 将许多步骤合并成一个大型查询。在某些情况下，这有利于应用程序——可以创建存储过程并重用代码，从而减少访问数据库的次数。但用户有可能滥用这些性能，创建了无法足够快速完成的大型查询。这些查询通常包括多个分组操作集、内联视图和针对数百万个记录的复杂的多行计算。

如果执行批处理操作，则可以把这样一个查询分解成原子组件，创建临时表来存储每一步的数据。如果某个操作需要几个小时才能完成，则一般总能找到一种方法将它分解成更小的组件部分。可以使用分而治之的方法处理性能问题。

例如，批处理操作可以合并多个表的数据，执行连接和排序，然后把结果插入表中。它在小型系统中完成得很好。在大型系统中，可能必须把该操作分成多个步骤：

(1) 创建工作表。向表中插入行(这些行来自查询的某个源表)，选择那些在后面的处理中仍然需要的行和列。

(2) 为来自第 2 个表的行和列创建另外一个工作表。

(3) 创建工作表中需要的任何索引。注意到目前为止的所有步骤都是并行的——插入、查询源表，以及创建索引。

(4) 执行连接，再次并行执行。连接的输出可以成为另一个工作表的输入。

(5) 执行需要的排序。尽量减少排序的数据。

(6) 把数据插入目标表中。

为什么要检查所有这些步骤呢？因为可以分别调整它们，分别调整它们要比 Oracle 把它们作为单个命令完成得更快。对于批处理操作，要考虑使步骤尽量简单。尽管需要管理分配给工作表的空间，但这种方法能显著提高批处理操作的性能。

2. 消除不必要的排序

作为前一节示例中的一部分，排序操作最后执行。通常，排序操作不适合 OLTP 应用程序。直到对整个行集完成排序，排序操作才把行返回给用户。另一方面，只要行可用，行操作就立即将行返回给用户。

考虑下面这个简单的测试：对一个大型表执行全表扫描。只要开始执行查询，就立即显示第 1 行。现在，虽然执行同一个全表扫描，但在没有索引的列上添加 order by 子句。直到所有行完成排序之后，再显示行。为什么会发生这样的情况？因为对于第 2 个查询，Oracle 在全表扫描的结果上执行了 SORT ORDER BY 操作。因为它是一组操作，所以必须完成这一组操作，才能执行下一个操作。

现在，设想一个应用程序，在它的某个过程中要执行很多查询。每个查询都有一个 order by 子句。这变成了一系列嵌套排序——直到它前面的操作完成后，才能开始下一个操作。

注意：union 操作执行排序。如果对于业务逻辑合适，则应该用 union all 操作代替 union 操作，因为 union all 不执行排序(因为它不消除重复)。

在创建索引的过程中，可以通过 create index 命令的 compute statistics 子句消除随后的排序，并在创建索引时收集统计信息。

3. 消除对查询撤消段的需求

执行查询时，Oracle 需要维护所查询行的读一致性映像。如果另一个用户修改了某一行，那么数据库将需要从撤消段中检索较早版本的块，以查看该行(因为它在用户开始查询时存在)。应用程序的设计要求在查询频繁地访问数据的同时，其他用户可以修改数据，这迫使数据库做更多工作——它不得不在多个位置查询一条数据。同样，这也是设计问题。虽然 DBA 可能可以配置撤消段区域的大小，以减少查询可能遇到的错误，但是纠正根本问题需要对应用程序的设计进行修改。

46.3.3　告诉数据库需要知道什么

在查询执行期间，当 Oracle 的优化程序判断数千种可能采用的路径时，它依赖于统计信息。如何管理这些统计信息将对查询的性能产生重要影响。

1. 不断更新统计信息

多久收集一次统计信息呢？每次对表中的数据进行较大的修改时，应该重新分析这些表。如果已经对表进行分区，则可以基于每个分区进行分析。可以使用统计信息自动收集(Automatic Statistics Gathering)功能自动收集统计信息。默认情况下，该过程在从每晚 10 点到早上 6 点以及周末全天的维护窗口期间收集统计信息。

因为分析工作通常是执行好几小时的批处理操作，所以可以通过在会话级别上改进排序和全表扫描的性能来调整它。因为分析执行的排序和全表扫描将极大地增强性能。

2. 在需要的地方提示

在大多数情况下，基于成本的优化程序(cost-based optimizer, CBO)为查询选择最有效的执行路径。但是，用户可能知道一条更好的路径。例如，您可能将一个本地表与远程数据库中的一个表连接起来，在这种情况下，您可能想避免重复地从远程数据库中查询个别行；若几乎从远程数据库检索中行，则极有可能获得好处。给 Oracle 一个提示以影响连接操作、整体查询目标、使用的指定索引或查询的并行性。关于主要提示的概述，请参阅 46.5.3。

46.3.4 最大化环境中的吞吐量

在理想环境中，永远不需要查询数据库外的信息；所有的数据始终都存储在内存中。但是，除非使用非常小的数据库，否则这是不现实的方法。本节概括了最大化环境的吞吐量的准则。

1. 使用磁盘缓存

如果 Oracle 在数据库中不能找到需要的数据，就执行物理读操作。但是有多少物理读操作真正到达磁盘呢？如果使用磁盘缓存，那么能够阻止 90%甚至更多对最需要的块的访问请求。如果数据库缓冲区缓存使用率是 90%，那么有 10%的时间访问磁盘——如果磁盘缓存阻止 90%请求到达磁盘，则有效的使用率就是 99%。Oracle 内部的统计信息对其提高性能没有影响；必须与磁盘管理员合作来配置并监控磁盘缓存。

2. 使用内存空间较大的数据库块

对于新数据库，唯一一个不在环境中使用内存空间最大的可用块的理由是：如果用户不能支持大量的用户对单个块执行更新和插入操作。除了这个原因，增加数据库块的大小将提高应用程序中几乎所有事务的性能。更大的数据块不仅有助于阻止索引拆分级别，而且有助于将更多的数据长时间地保存在内存中。为了支持很多并发插入和更新，请增加对象级别的 pctfree 参数的设置。

3. 在块级别有效地存储数据

Oracle 在内存中存储数据块。最应该关注的是确保这些块的数据尽可能地密集。如果在块级别的数据存储效率低，就不能像从数据库可用的缓存中那样获得好处。

如果应用程序中的行不打算进行更新，就设置尽可能低的 pctfree。对于分区的表，为每个分区设置 pctfree 值，以最大化块中的行存储器。默认情况下，每个分区将使用表的 pctfree 设置。为索引设置低 pctfree 值。

如果 pctfree 设置得太低，则更新可能迫使 Oracle 移动行(migrated row，迁移的行)。在某些情况下，行链接不可避免，例如当行的长度大于数据库块的大小时。当发生行链接和行迁移时，每次访问一行将需要访问多个数据块，从而影响每条命令需要的逻辑读次数。通过分析表，然后通过 USER_TABLES 检查此表的统计信息，从而可以检测行链接。当分析表时，被链接行的数量将填充到 Chain_Cnt 列中。

4. 设计吞吐量，而不是磁盘空间

把一个运行在 8 个 9GB 磁盘上的应用程序移动到一个 72GB 的磁盘上。该应用程序运行得更快了还是更慢了？通常，它运行得更慢了，因为该单磁盘的吞吐量不可能等于 8 个不同的磁盘总的吞吐量。设计磁盘布局时不是基于可用的空间(这是常用方法)，而是基于可用的磁盘吞吐量。用户可能决定只使用每个磁盘的一部分。除非该磁盘的可用吞吐量提高，否则产品应用程序不会使用剩余的磁盘空间。

5. 避免使用临时段

只要可能，请在内存中执行所有排序。任何写入临时段的操作都可能浪费资源。当 PGA 中的排序区域不能分配足够的内存来支持操作的排序需求时，Oracle 将使用临时段。排序操作包括索引的创建、order by 子句、统计信息的收集、group by 操作和某些连接。如本章前面所提到的，应该尽量减少排序的行。当执行其余的排序时，请在内存中执行。

注意：
Oracle 不建议使用 SORT_AREA_SIZE 参数，除非实例是用共享的服务器选项配置的。Oracle 建议通过设置 PGA_AGGREGATE_TARGET 来启用 SQL 工作区域大小的自动调整。保留 SORT_AREA_SIZE 是为了向后兼容。

6. 使用更少、更快的处理器

如果可以选择，就用少量快速的处理器代替大量较慢的处理器。操作系统将管理更少的处理队列，从而完成得更好任务。

46.3.5　分开处理数据

如果不能避免在数据库上执行大量操作，则可以尝试把工作拆分为更好管理的块。通常可以严格地限制操作所作用的行数，以大大提高性能。

1. 使用分区

分区有益于最终用户、DBA、和应用程序支持人员。对于最终用户，有两个潜在的益处：提高查询性能和提高数据库的可用性。由于分区消除，因此查询性能可以提高。优化程序知道什么分区可能包含查询所请求的数据。结果，不包含所请求数据的分区从查询过程中消除。由于几乎不需要逻辑读和物理读操作，因此查询完成得较快。

由于分区对 DBA 和应用程序支持人员有益，因此提高了可用性。很多管理功能可以在单个分区上执行，使表中其他部分不受影响。例如，可以截断表中的单个分区。可以拆分分区，把它移动到不同表空间，也可以与已有的表交换(这样以前独立的表被看成分区)。可以每次收集一个分区的统计信息。所有这些功能缩小了管理功能的范围，从整体上降低了它们对数据库的可用性的影响。

2. 使用物化视图

可以使用物化视图划分用户对表执行的操作类型。当创建物化视图时，可以把用户的查询直接定向到物化视图，也可以依赖于 Oracle 查询重写功能，把查询重定向到物化视图。这样的结果是将有两个数据副本——一个副本为新事务数据的输入提供服务，另一个副本是为查询提供服务的物化视图。这样可以保持一个副本脱机以便进行维护，而不影响另一个副本的可用性。同样注意，可以分别索引基表和物化视图(它们具有打算使用的指定数据访问类型需要的结构)。

3. 使用并行操作

几乎每个主要操作都可以并行化——包括查询、插入、对象创建和数据加载。并行选项允许单个命令的执行涉及多个处理器，有效地把命令划分成多个更小的并列的命令。这样的结果是，命令执行得更好。可以在对象级别指定并行操作的程度，并可以通过查询中的提示重写它。

注意：
当操作涉及大型数据集时，并行操作的性能最好。

46.3.6 正确测试

在大多数开发方法中，应用程序测试分为多个阶段，包括模块测试、全系统测试以及性能压力测试。很多时候，由于时间限制(差不多接近应用程序的交付期限)，不能充分执行全系统测试和性能压力测试。这样的结果就是：在不能保证应用程序整体上的功能和性能满足用户需要的情况下，应用程序就作为产品而发布了。这是个重大的缺陷，任何应用程序使用者都不能容忍。用户不仅仅需要应用程序的某个组件正确运行，而且需要整个应用程序正确地工作，以支持业务处理。如果他们不能按时完成当天的业务，应用程序就是失败的。

用来判断应用程序是否需要调整的一个关键原则是：如果应用程序减慢了业务处理的速度，它就需要调整。执行的测试必须能够确定在预期的生产负载下，应用程序是否阻碍了业务处理的速度。

1. 测试大量数据

如本章前面所述，数据库中的对象在使用过一段时间后会发挥不同功能。例如，pctfree 设置可能会导致只有一半的块被使用，或者行被链接。每个设置都会引起性能问题，而这些问题有时只有在应用程序使用一段时间后才能发现。

数据量的深层问题涉及索引。当 B*树索引增大时，它们可能内部拆分——索引中的记录项级别增加。这样的结果是，可以把新级别描述成索引中的索引。索引中的附加级别增加了索引对数据加载速率的影响。直到索引被拆分后，才会看到这个影响。应用程序在第 1 个星期或第 2 个星期都能正确地工作，只有在数据量达到临界级别之后才突然失效，不再支持业

务需要。在测试中，尽管表中已经包含了大量的数据，但还是没有方法代替以产品速率加载的产品数据。

2. 用多个并发用户测试

单个用户测试不能反映大多数数据库应用程序的预期生产使用。必须能够确定并发用户是否会遇到死锁、数据一致性问题或性能问题。例如，假设应用程序模块在其处理过程中使用工作表。在表中插入行、并对行操作然后查询。不同的应用程序模块做类似的处理——使用相同的表。当同时执行时，这两个过程试图使用彼此的数据。除非多个用户同时执行多个应用程序的功能进行测试，否则不可能发现这个问题，也不可能发现它将产生的业务数据错误。

用很多并发用户测试将有助于识别应用程序的哪些区域是用户经常使用撤消段来完成其查询的，这些区域影响系统的性能。

3. 测试索引对加载次数的影响

索引列的每个 insert、update 或 delete 操作都要比没有索引的表的相同事务处理慢 3 倍左右。虽然也有一些例外——例如，排序的数据其影响要小得多——但是该规则通常是正确的。如果每秒能够将 3 000 行记录加载到环境中没有索引的表中，那么向表中添加单个索引将会使插入速度减慢到每秒 1 000 行左右。影响由操作系统环境、包含的数据结构和数据的排序程度来决定。

在用户环境中，每秒能插入多少行？执行一系列简单的测试。创建一个没有索引的表，并向其中插入大量行。重复该测试，以降低物理读操作对定时结果的影响。计算每秒钟插入的行数。在大多数环境下，每秒钟可以向数据库中插入上万行记录。在用户的其他数据库环境中执行相同的测试，这样可以识别出任何与其他环境显著不同的内容。

现在考虑应用程序。能否以接近刚才计算的速率，通过应用程序将行插入表中？很多应用程序运行的速率比环境支持的速率慢 5%。它们由于不必要的索引或本章前面描述的代码设计问题而变慢。如果它们的加载速率降低(例如，从每秒 40 行降低到每秒 20 行)，调整将不仅仅应该关注于为什么出现速率降低，而且也应关注于为什么在能够支持每秒上千行插入的环境，现在应用程序每秒只能插入 40 行。

4. 重复测试

大多数规范的产业都有测试标准。它们的标准是如此合理，以至于所有测试工作都应该遵循它们。在标准中要求所有的测试必须是可重复的。为了遵从这些标准，必须能够重新创建使用的数据集、执行的完全相同的操作、预期的完全相同的结果以及看到并记录下来的完全相同的结果。生产前应用程序有效性的测试必须在产品硬件上执行。把应用程序移动到不同的硬件上，需要重新测试应用程序。测试人员和业务用户必须停止所有的测试。

当听到这些限制时，大多数人都会认为它们在任何测试过程中都是良好的措施。事实上，业务用户可能期望开发应用程序的人员遵循这些标准(即使业界不需要这些标准)。但是，他们遵循这些标准了吗？如果没有，为什么没有？通常会列举出两个不遵循该标准的原因：时间和成本。这样的测试需要规划、人员资源、业务用户参与，以及执行和归档的时间。在产

品水准的硬件上测试需要购买额外的服务器。这些是最明显的成本—— 但是不执行这样的测试，有什么样的业务成本呢？在美国制药工业需要执行系统有效性的测试，因为这些系统直接影响到重要产品的完善，如血液供应的安全性。如果业务有重要组件由应用程序来完成 (如果不是这样，那么为什么要构建该应用程序呢？)，那么必须考虑到不充足的成本、匆忙的测试，并把那些潜在的成本与业务用户进行交流。业务用户必须参与对不正确的数据或慢得无法接受的性能的风险评估。这样可能导致延长产品的最终交付期限，以支持正确的测试。

在很多情况下，由于在项目开始时没有合适的测试标准，所以出现了匆忙的测试周期。当项目开始时，如果在企业级有一致的、全面的、归档良好的测试标准，那么测试周期在最终执行时将会变短。测试人员将提前很久知道所需要的重复数据集。他们将提供测试模板。如果任何测试结果有问题，或者应用程序因为修改而需要重新测试，测试就能重复进行。应用程序的用户将知道该测试足够健壮可以模拟应用程序的生产使用。如果因为性能原因导致系统测试失败，则问题可能出在设计上(如前面几节所述)，或者是个别查询的问题。下面几小节将介绍如何显示 SQL 语句的执行路径、包含的主要操作，以及调整 SQL 时使用的相关提示。

46.4 生成并读取说明计划(explain plan)

可以用下面两种方法显示查询的执行路径：

- set autotrace on 命令
- explain plan 命令

以下几节将对这两个命令进行说明；本章剩余部分将使用 set autotrace on 命令说明优化程序提出的执行路径。

46.4.1 使用 set autotrace on

可以自动显示在 SQL*Plus 中执行的每个事务的执行路径。set autotrace on 命令将导致每个查询执行之后显示其执行路径和关于分析查询所涉及处理的高级跟踪信息。

必须首先在账户中创建 PLAN_TABLE 表，才能使用 set autotrace on 命令。由于 PLAN_TABLE 结构可能随着 Oracle 版本的不同而不同，因此每次 Oracle 更新后，都需要删除并重新创建 PLAN_TABLE 表的副本。以下程序清单中显示的命令将删除任何现存的 PLAN_TABLE 表，并用当前版本代替它。

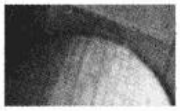
注意：

为了使用 set autotrace on，DBA 必须首先在数据库中创建 PLUSTRACE 角色，并把该角色授权给账户。PLUSTRACE 角色能够访问 Oracle 数据字典中与性能相关的基本视图。创建 PLUSTRACE 角色的脚本是 plustrce.sql，它通常在 Oracle 软件主目录的/sqlplus/admin 目录中。

创建 PLAN_TABLE 表的文件位于 Oracle 软件主目录下的/rdbms/admin 目录中。

```
drop table PLAN_TABLE;
@$ORACLE_HOME/rdbms/admin/utlxplan.sql
```

使用 set autotrace on 命令时，记录被插入 PLAN_TABLE 表的副本中，以显示操作的执行顺序。查询完成后，显示所选择的数据。显示查询的数据之后，显示操作的顺序以及有关查询处理的统计信息。以下对 set autotrace on 的解释重点在于显示操作顺序的输出部分。



注意：
为了在不运行查询时显示 explain plan 的输出，使用 set autotrace on trace explain 命令。

如果使用 set autotrace on 命令的默认选项，那么直到查询完成之后，才能看到查询的说明计划。explain plan 命令(接下来将介绍)不需要首先运行查询，就可以显示执行路径。因此，如果查询的性能未知，则可以在运行该查询前选择使用 explain plan 命令。如果非常肯定查询的性能满足要求，那么使用 set autotrace on 验证其执行路径。

注意：
使用 set autotrace on 命令时，一旦显示了执行路径，Oracle 就将自动删除它插入 PLAN_ TABLE 表中的记录。

如果使用并行查询选项或查询远程数据库，则 set autotrace on 输出的其他部分将显示通过并行查询服务器进程执行的查询或者在远程数据库中执行的查询的文本。

可以使用 set autotrace off 命令禁用自动跟踪功能。

要使用 set autotrace，必须能够通过 SQL*Plus 访问数据库。如果可以通过 SQL 而不是 SQL*Plus 进行访问，那么可以使用 explain plan(如以下部分所描述)。

在下面的示例中，执行了 BOOKSHELF 表的全表扫描。为了简便起见，输出的行没有显示在此输出中。在查询下面显示了操作的顺序。

```
select *
  from BOOKSHELF;

Execution Plan
----------------------------------------------------------
  0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=31 Bytes=12
        40)
  1 0    TABLE ACCESS (FULL) OF 'BOOKSHELF' (TABLE) (Cost=4 Card=31
        Bytes=1240)
```

“Execution Plan”显示了优化程序将用来执行查询的步骤。每一步都被赋予一个 ID 值(以 0 开始)。第 2 个数字显示了当前操作的“父”操作。因此，对于前面的示例，第 2 个操作——TABLE ACCESS (FULL) OF ‘BOOKSHELF’—— 具有父操作(select 语句本身)。每一步了显示该步骤和它所有的子步骤的累计成本。

也可以为 DML 命令生成操作顺序。在下面的示例中，显示了 delete 语句的执行路径：

```
delete
   from BOOKSHELF_AUTHOR;
```

```
Execution Plan
----------------------------------------------------------
   0       DELETE STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=37 Bytes=12
           58)
   1    0   DELETE OF 'BOOKSHELF_AUTHOR'
   2    1     TABLE ACCESS (FULL) OF 'BOOKSHELF_AUTHOR' (TABLE) (Cost=
           4 Card=37 Bytes=1258)
```

正如所预期的那样，delete 命令包含了全表扫描。如果已经对表进行了分析，则 Execution Plan 列的输出将显示每个表的行数、每一步的相关成本以及操作的总成本。步骤中显示的成本是累计值；它们是该步骤加上它所有的子步骤的成本之和。可以使用该信息精确地指出在查询的处理过程中成本最高的操作。

在下面的示例中，执行稍微复杂的查询。对 BOOKSHELF 表执行基于索引的查询，使用其主键索引。

```
select /*+ INDEX(bookshelf) */ *
    from BOOKSHELF
  where Title like 'M%';

Execution Plan
----------------------------------------------------------
   0       SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=2 Bytes=80)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'BOOKSHELF' (TABLE) (Cost
           =3 Card=2 Bytes=80)
   2    1     INDEX (RANGE SCAN) OF 'SYS_C004834' (INDEX (UNIQUE)) (Co
           st=1 Card=2)
```

该程序清单包括 3 个操作。操作#2，即 BOOKSHELF 主键索引(其名字是由系统生成)的 INDEX RANGE SCAN，为操作#1(即 TABLE ACCESS BY INDEX ROWID 操作)提供数据。从 TABLE ACCESS BY INDEX ROWID 返回的数据是查询(即操作#0)所需的数据。

上述输出也显示了优化程序在 execution plan 中自动缩进每个相继的步骤。通常，可以从里到外、从上到下读取操作列表。因此，如果列出两个操作，则通常首先执行缩进最多的那个操作。如果两个操作具有相同的缩进级别，那么首先执行最先列出的那个操作(具有最小的操作号)。

注意：

当选择查询的执行路径时，有很多因素会影响优化程序所做的选择。当在数据库中执行这里所展示的示例时，优化程序所选择的执行路径可能与这些示例有所不同，这没有关系。优化程序在确定选择哪条路径时要考虑数据库、环境、参数设置和选项。

在下面的示例中，没有使用索引，对 BOOKSHELF 表和 BOOKSHELF_AUTHOR 表进行连接。Oracle 将使用合并连接(merge join)操作。合并连接在连接两个排序的行集之前，分别对每个表进行排序。

```
select /*+ USE_MERGE (bookshelf, bookshelf_author)*/
        BOOKSHELF_AUTHOR.AuthorName
```

```
   from BOOKSHELF, BOOKSHELF_AUTHOR
  where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;

Execution Plan
----------------------------------------------------------
   0        SELECT STATEMENT Optimizer=ALL_ROWS (Cost=7 Card=5 Bytes=320)
   1    0   MERGE JOIN (Cost=7 Card=5 Bytes=320)
   2    1     INDEX (FULL SCAN) OF 'SYS_C004834' (INDEX (UNIQUE)) (Cos
             t=1 Card=31)
   3    1     SORT (JOIN) (Cost=5 Card=37 Bytes=1258)
   4    3      TABLE ACCESS (FULL) OF 'BOOKSHELF_AUTHOR' (TABLE) (Cos
             t=1 Card=37 Bytes=1258)
```

这里的缩进起先看起来可能有点混乱，但是操作编号提供的操作父系信息让操作顺序变得清晰。最里面的操作最先执行—— TABLE ACCESS FULL 和 INDEX FULL SCAN 操作。接下来，全表扫描的数据通过 SORT JOIN 操作(在操作#4 中)处理，而索引扫描(已经被排序)的输出用作 TABLE ACCESS BY INDEX ROWID(步骤 2)的基础。步骤 2 和步骤 4 都是操作#1(MERGE JOIN)的子操作。MERGE JOIN 操作通过 select 语句提供返回给用户的数据。

如果使用嵌套循环(NESTED LOOPS)连接运行同样的查询，就会产生不同的执行路径。如下面的程序清单所示，NESTED LOOPS 连接将能够利用 BOOKSHELF 表的 Title 列上的主键索引。

注意：

根据表的大小和可用的统计信息，Oracle 可以选择执行另一种类型连接——散列连接(hash join)来代替 NESTED LOOPS 连接。

```
select /*+ INDEX(bookshelf) */
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=4 Card=37 Bytes=19
          24)
   1    0  NESTED LOOPS (Cost=4 Card=37 Bytes=1924)
   2    1    TABLE ACCESS (FULL) OF 'BOOKSHELF_AUTHOR' (TABLE) (Cost=
          4 Card=37 Bytes=1258)
   3    1     INDEX (UNIQUE SCAN) OF 'SYS_C004834' (INDEX (UNIQUE)) (C
          ost=1 Card=1 Bytes=18)
```

NESTED LOOPS 连接是极少数不遵循缩进的执行路径的“从内到外读取”规则的执行路径之一。为了正确地读取 NESTED LOOPS 执行路径，请检查直接为 NESTED LOOPS 操作提供数据的操作顺序(在本例中，是操作#2 和操作#3)。在这两个操作中，首先执行具有最小编号的操作(本示例中是#2)。因此首先执行 BOOKSHELF_AUTHOR 表的 TABLE ACCESS FULL (BOOKSHELF_AUTHOR 是查询的驱动表)。

一旦建立了查询的驱动表，其他的执行路径就可以从内到外、从上到下读取。执行的第 2 个操作是 BOOKSHELF 主键索引的 INDEX UNIQUE SCAN。之后，NESTED LOOPS 操作可以将行返回给用户。

如果使用散列连接代替 NESTED LOOPS 连接，则执行的路径是：

```
select /*+ USE_HASH (bookshelf) */
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=37 Bytes=19
         24)
   1  0    HASH JOIN (Cost=6 Card=37 Bytes=1924)
   2  1       INDEX (FULL SCAN) OF 'SYS_C004834' (INDEX (UNIQUE)) (Cos
         t=2 Card=31 Bytes=558)
   3  1       TABLE ACCESS (FULL) OF 'BOOKSHELF_AUTHOR' (TABLE) (Cost=
         1 Card=37 Bytes=1258)
```

该散列连接的执行路径显示执行了两个单独的扫描。由于这两个操作以相同的缩进级别列出来，因此 BOOKSHELF 表的主键索引首先被扫描(因为它具有较小的操作编号)。

46.4.2 使用 explain plan

可以在不先运行查询的情况下，使用 explain plan 命令生成查询的执行路径。为了使用 explain plan 命令，必须首先在用户模式下创建 PLAN_TABLE 表(参阅前面关于创建表的指令)。为了确定查询的执行路径，在查询前面加上下面的 SQL：

```
explain plan
for
```

下面的程序清单显示了执行 explain plan 命令的一个示例：

```
explain plan
for
select /*+ USE_HASH (bookshelf) */
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;
```

当执行 explain plan 命令时，记录被插入 PLAN_TABLE 表中。可以直接查询 PLAN_TABLE 表，或者使用 DBMS_XPLAN 程序包格式化结果。下面的程序清单显示了 DBMS_XPLAN 程序包的使用方法：

```
select * from table(DBMS_XPLAN.Display);
```

注意：
utlxplp.sql 脚本位于$ORACLE_HOME/rdbms/admin 目录中，此脚本通过调用 DBMS_XPLAN.Display 过程来显示最近执行的命令的说明计划。

格式化的 explain plan 的清单为：

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
| Id | Operation          | Name              |Rows |Bytes |Cost(%CPU) | Time     |
-------------------------------------------------------------------------------
| 0  | SELECT STATEMENT   |                   | 37  |1924  |    4 (25) | 00:00:01 |
|*1  | HASH JOIN          |                   | 37  |1924  |    4 (25) | 00:00:01 |
| 2  | INDEX FULL SCAN    | SYS_C004834       | 32  |608   |    1(100) | 00:00:01 |
| 3  | TABLE ACCESS FULL  | BOOKSHELF_AUTHOR  | 37  |1258  |    4 (25) | 00:00:01 |

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - access("BOOKSHELF"."TITLE"="BOOKSHELF_AUTHOR"."TITLE")
```

输出包括 Cost 列，显示每一步骤及其子步骤的相关“成本”。它也包括 ID 值，缩进显示了各步骤之间的关系，附加部分列出了每一步骤的限制条件。

当调整查询时，应当注意那些扫描很多行、但只返回很少行的步骤。例如，应当避免执行这样的操作：在上百万行的表中，为了返回 3 行而执行全表扫描。可以使用 explain plan 的输出识别那些处理最多行数的步骤。

46.5　Explain Plan 中的主要操作

如前面的程序清单所述，查询的说明计划详细说明了 Oracle 用来检索和处理行的方法。接下来的几节将描述最常用的操作。

46.5.1　TABLE ACCESS FULL

全表扫描依次读取表的每一行。优化程序调用在全表扫描过程中使用的操作 TABLE ACCESS FULL。为了优化全表扫描的性能，Oracle 在读取每个数据库期间读取多块。

无论何时查询中没有 where 子句时，都可以使用全表扫描。例如，下面的查询从 BOOKSHELF 表中选择所有的行：

```
select *
  from BOOKSHELF;
```

为了解析上述查询，Oracle 将对 BOOKSHELF 表执行全表扫描。如果 BOOKSHELF 表比较小，则 BOOKSHELF 的全表扫描可能比较快，几乎不需要性能开销。但是，当 BOOKSHELF 变大时，执行全表扫描的成本将增加。如果多个用户执行 BOOKSHELF 表的全表扫描，则全表扫描的相关成本将增加得更快。

如果正确规划，全表扫描就不会成为性能问题。可以与数据库管理员合作，确保数据库的配置可以利用并行查询(Parallel Query)和多块读取等功能。如果没有正确地配置全表扫描的环境，就需要小心监控它们的使用。

注意：

根据所选择的数据，优化程序可以选择使用全索引扫描代替全表扫描。

46.5.2 TABLE ACCESS BY INDEX ROWID

为了提高表访问的性能，可以使用通过行记录的ROWID值访问行的Oracle操作。ROWID记录存储行的物理位置。Oracle使用索引将数据值与ROWID值相关联——从而也将数据值与数据的物理位置相关联。如果给定某一行的ROWID，Oracle就可以使用TABLE ACCESS BY INDEX ROWID操作对行进行检索。

当知道ROWID时，就确切地知道行的物理位置。如46.5.4节所述，可以使用索引访问ROWID信息。因为索引提供了对ROWID值的快速访问，所以它们有助于提高那些利用索引列的查询的性能。

46.5.3 相关提示

可以在查询中指定提示，用于在查询的处理中对CBO进行指导。使用下面示例中所显示的语法指定提示。在select关键字之后，立即输入下面的字符串：

```
/*+
```

接下来，添加提示，例如：

```
FULL(bookshelf)
```

用下面的字符串结束提示：

```
*/
```

在查询中，提示使用Oracle的注释语法，在提示开始处加上“+”符号。本章将介绍与每个操作相关的提示。对于表访问，FULL提示告诉Oracle在所列出的表中执行全表扫描(TABLE ACCESS FULL操作)，如下面的程序清单所示：

```
select /*+ FULL(bookshelf) */ *
  from BOOKSHELF
 where Title like 'T%';
```

如果不使用FULL提示，则Oracle通常计划使用Title列上的主键索引来解析该查询。因为该表目前很小，所以全表扫描成本不大。随着表的增大，对该查询最好使用基于ROWID的访问。

46.5.4 使用索引的操作

在Oracle中，主要有两类索引：唯一索引(unique indexes)和非唯一索引(nonunique indexes)。在唯一索引中，索引表的每一行都包含索引列的唯一值；在非唯一索引中，行的索引值可以重复。用来从索引中读取数据的操作取决于所使用的索引类型和访问索引的查询的编写方式。

考虑BOOKSHELF表：

```
create table BOOKSHELF
(Title        VARCHAR2(100) primary key,
```

```
Publisher      VARCHAR2(20),
CategoryName   VARCHAR2(20),
Rating         VARCHAR2(2),
constraint CATFK foreign key (CategoryName)
references CATEGORY(CategoryName));
```

Title 列是表 BOOKSHELF 的主键——即它唯一标识了每一行，每个属性都依赖于 Title 列的值。

无论何时创建 PRIMARY KEY 或 UNIQUE 约束时，Oracle 都创建唯一索引，以强制列中的值的唯一性。按照 create table 命令的定义，将在 BOOKSHELF 表上创建 PRIMARY KEY 约束。因为不是显式指定约束，所以支持主键的索引的名字由系统生成。

可以手动在 BOOKSHELF 表的其他列上创建索引。例如，可以通过 create index 命令在 CategoryName 列上创建非唯一索引。

```
create index I_BOOKSHELF_CATEGORY
   on BOOKSHELF(CategoryName)
tablespace INDEXES
compute statistics;
```

现在，BOOKSHELF 表有两个索引：Title 列上的唯一索引和 CategoryName 列上的非唯一索引。在查询解析期间，可以使用 1 个或多个索引，这取决于如何编写和执行查询。作为索引创建的一部分，它的统计信息通过 compute statistics 子句收集。因为表已经用行填充，所以不需要执行单独的命令来分析索引。

1. INDEX UNIQUE SCAN

为了在查询期间使用索引，所编写的查询必须允许使用索引。在大多数情况下，允许优化程序通过查询的 where 子句使用索引。例如，下面的查询可以在 Title 列上使用唯一索引。

```
select *
  from BOOKSHELF
 where Title = 'INNUMERACY';
```

在内部，上述查询的执行分为两步。首先，通过 INDEX UNIQUE SCAN 操作访问 Title 列上的索引。将从索引中返回匹配“INNUMERACY”的 ROWID 值；然后通过 TABLE ACCESS BY INDEX ROWID 操作，使用 ROWID 值查询 BOOKSHELF 表。

如果查询选择的所有列都已经包含在索引中，Oracle 就不需要使用 TABLE ACCESS BY INDEX ROWID 操作；因为数据将在索引中，而索引是查询需要的所有内容。因为查询从 BOOKSHELF 表中选择所有的列，而索引不包含 BOOKSHELF 表的所有列，所以需要 TABLE ACCESS BY INDEX ROWID 操作。

2. INDEX RANGE SCAN

如果根据某一范围值查询数据库，或使用非唯一索引查询，则使用 INDEX RANGE SCAN 操作查询索引。

再次考虑 BOOKSHELF 表，在其 Title 列上具有唯一索引。以下查询

```
select Title
  from BOOKSHELF
 where Title like 'M%';
```

将返回以“M”开始的所有 Title 值。因为 where 子句使用 Title 列，所以可以使用 Title 列上的主键索引解析该查询。但是，在 where 子句中没有指定唯一值，而是指定了某一范围的值。因此，将通过 INDEX RANGE SCAN 操作访问该唯一主键索引。由于 INDEX RANGE SCAN 操作需要从索引中读取多个值，因此它们的效率没有 INDEX UNIQUE SCAN 操作高。

在上述示例中，查询只选择了 Title 列。因为 Title 列的值存储在正在扫描的主键索引中，所以在查询执行期间数据库不需要直接访问 BOOKSHELF 表。解析查询所需的唯一操作是主键索引的 INDEX RANGE SCAN。

BOOKSHELF 表的 CategoryName 列在该列的值上具有非唯一索引。如果在查询的 where 子句中对 CategoryName 值指定限制条件，则执行 CategoryName 索引的 INDEX RANGE SCAN。因为 BOOKSHELF$CATEGORY 索引是非唯一索引，所以即使在查询中 CategoryName 等于一个单一值，数据库也不能在 BOOKSHELF$CATEGORY 上执行 INDEX UNIQUE SCAN。

46.5.5 何时使用索引

因为索引对查询的性能有着极大的影响，所以应当知道在什么条件下使用索引解析查询。下面几节描述了在解析查询时，引起索引使用的条件。

1. 如果设置索引列等于某个值

在 BOOKSHELF 表中，CategoryName 列具有非唯一索引 BOOKSHELF$CATEGORY。比较 CategoryName 列与某值的查询将能够使用 BOOKSHELF$CATEGORY 索引。

下面查询比较 CategoryName 列与值“ADULTNF”：

```
select Title
  from BOOKSHELF
 where CategoryName = 'ADULTNF';
```

由于 BOOKSHELF$CATEGORY 索引是非唯一索引，因此该查询可能返回多行，当从它读取数据时，可能使用 INDEX RANGE SCAN 操作。根据表的统信息，Oracle 可以选择执行全表扫描。

如果它使用索引，则上述查询的执行包括两个操作：BOOKSHELF$CATEGORY 的 INDEX RANGE SCAN(得到 CategoryName 列中值为“ADULTNF”的所有行的 ROWID 值)，接下来是 BOOKSHELF 表的 TABLE ACCESS BY INDEX ROWID(检索 Title 列的值)。

如果在某一列上创建唯一索引，用“=”号比较该列与某一值，则用 INDEX UNIQUE SCAN 代替 INDEX RANGE SCAN。

2. 如果为索引列指定某一范围值

不需要为所使用的索引顺序地指定明确的值。INDEX RANGE SCAN 操作可以扫描索引，得到某一范围值。在下面的查询中，查询 BOOKSHELF 表的 Title 列，得到一系列值(以 M 开

头的值)：

```
select Title
  from BOOKSHELF
 where Title like 'M%';
```

当使用“>”或“<”运算符时，也执行范围扫描：

```
select Title
  from BOOKSHELF
 where Title > 'M';
```

当指定某一列的某一范围值时，如果第 1 个指定的字符是通配符，则不使用索引解析查询。下面的查询将不在可用的索引上执行 INDEX RANGE SCAN：

```
select Publisher
  from BOOKSHELF
 where Title like '%M%';
```

因为用于比较的字符串的第 1 个字符是通配符，所以不能使用索引快速地找到相关的数据。因此，取而代之的是将执行全表扫描(TABLE ACCESS FULL 操作)。根据表和索引的统计信息，Oracle 可以选择执行索引全局扫描。在本示例中，如果所选择的列是 Title 列，则优化程序选择执行主键索引的全局扫描，而不是 BOOKSHELF 表的全局扫描。

3. 如果 where 子句中的列没有执行函数

下面的查询使用 BOOKSHELF$CATEGORY 索引：

```
select COUNT(*)
  from BOOKSHELF
 where CategoryName = 'ADULTNF';
```

注意：

考虑到 BOOKSHELF 表的大小，优化程序通常选择执行全表扫描，而不对此查找使用索引。对于大型表，应该选择基于索引的方法。

如果不知道 CategoryName 列值的存储格式是大写、大小写混合还是小写，则会怎么样呢？在这种情况下，可以编写如下查询：

```
select COUNT(*)
  from BOOKSHELF
 where UPPER(CategoryName) = 'ADULTNF';
```

UPPER 函数先把 Manager 值修改成大写，然后把它们与值“ADULTNF”进行比较。但是，在列上使用函数，会阻止优化程序使用列上的索引。除非在 UPPER(CategoryName)上创建基于函数的索引，否则上述查询(使用 UPPER 函数)将执行 BOOKSHEL 表的 TABLE ACCESS FULL；关于基于函数的索引，请参阅第 17 章。

如果把两列连接在一起，或者把字符串连接到列上，就不能使用这些列上的索引。索引存储列的实际值，关于该值的任何修改都会阻止优化程序使用索引。

4. 如果索引的列使用 NO IS NULL 或 IS NOT NULL 检查

NULL 值不存储在索引中。因此，下面的查询不使用索引；使用索引对解析查询没有任何帮助：

```
select Title
  from BOOKSHELF
 where CategoryName is null;
```

因为 CategoryName 是查询中唯一具有限制条件的列，而且限制条件是 NULL 检查，所以将不会使用 BOOKSHELF$CATEGORY 索引，而是 TABLE ACCESS FULL 操作将用来解析查询。

如果在列上执行 IS NOT NULL 操作，则会怎么样呢？列上所有非空值都存储在索引中；但是，索引搜索效率不高。为了解析该查询，优化程序需要从索引中读取每个值，并访问表中索引返回的每行。在大多数情况下，执行全表扫描比为索引返回的所有值执行索引扫描(用相关的 TABLE ACCESS BY INDEX ROWID 操作)效率更高。因此，以下查询不使用索引：

```
select Title
  from BOOKSHELF
 where CategoryName is not null;
```

如果所选择的列位于索引中，则优化程序选择执行全索引扫描代替全表扫描。

5. 如果使用等于条件

在前面部分的示例中，用“=”符号把 Title 值与某个值进行比较，如下面查询所示：

```
select *
  from BOOKSHELF
 where Title = 'INNUMERACY';
```

如果想选择 Title 不是“INNUMERACY”的所有记录，那么应该怎么做呢？可以用“!=”代替“=”，现在的查询为：

```
select *
  from BOOKSHELF
 where Title != 'INNUMERACY';
```

当解析修改后的查询时，优化程序不使用索引。当这些值精确地与另一个值比较时——当限制条件是“等于”或“不等于”时，才使用索引。在本示例中，如果全索引扫描(加上 TABLE ACCESS BY INDEX ROWID 操作，以获得所有列)比全表扫描快，优化程序就只能选择索引。

“不等于”的另一个示例是子查询中使用的 not in 子句，下面的查询从 BOOKSHELF 表中选择不是由 Stephan Jay Gould 编写的书：

```
select *
  from BOOKSHELF
 where Title NOT IN
(select Title
```

```
    from BOOKSHELF_AUTHOR
 where AuthorName = 'STEPHEN JAY GOULD');
```

在某些情况下，上述程序清单中的查询将不能使用 BOOKSHELF 表中 Title 列上的索引，因为没有设置它等于任何值。相反，使用值 BOOKSHELF.Title 与 not in 子句消除那些与子查询返回值匹配的行。要使用索引，必须设置索引列等于某个值。在很多情况下，Oracle 内在地把 not in 重写成 not exists 子句，允许查询使用索引。下面使用 in 子句的查询，可以在 BOOKSHELF.Title 列上使用索引，或者在表之间执行非索引连接；优化程序将根据可用的统计信息选择成本最低的路径。

```
select *
  from BOOKSHELF
 where Title IN
(select Title
  from BOOKSHELF_AUTHOR
 where AuthorName = 'STEPHEN JAY GOULD');
```

6. 如果多列索引的第 1 列设置为等于某个值

可以在一列或多行上创建索引。在多列上创建索引时，如果索引的第 1 列在查询的限制条件中使用，就使用该索引。

如果查询仅仅指定索引的第一列以外的值，可以通过 Oracle9i 中引入的索引跳跃式扫描(index skip-scan)功能，使用索引。即使它的第 1 列没有列在 where 子句中，跳跃式扫描索引访问也使优化程序可以使用连接索引。可能需要使用 INDEX 提示(46.5.6 节将介绍)告诉优化程序对跳跃式扫描访问使用索引。可以使用 INDEX_SS 提示建议使用跳跃式扫描索引访问。

7. 如果使用 MAX 或 MIN 函数

如果选择索引列中的 MAX 或 MIN 值，优化程序就可以使用索引，快速地找到列的最大值或最小值。

8. 如果索引可选

前面所有确定是否使用索引的规则，都需要考虑所执行查询的语法和可用索引的结构。优化程序可以使用索引的选择性来判断使用索引是否可以降低查询的执行成本。

在高选择性索引中，每个不同的列值对应不同的记录。例如，如果表中有 100 条记录，而该表中的某列有 80 个不同的值，则该列上索引的选择性为 80/100=0.8。选择性越高，列中每个不同值返回的行数越少。

在范围扫描中，每个不同的值返回的行数很重要。如果索引具有较低的选择性，那么用很多 INDEX RANGE SCAN 操作和 TABLE ACCESS BY INDEX ROWID 操作检索数据，要比表的 TABLE ACCESS FULL 包含更多的工作。

除非对索引进行分析，否则优化程序不会考虑索引的选择性。优化程序使用直方图判断表中数据的分布。例如，如果数据值严重偏离，以致于大多数据值在很小的数据范围内，那么优化程序将避免使用索引查询范围内的值，而是使用索引查询范围外的值。

9. 相关提示

有几个提示可以用来指导优化程序使用索引。INDEX 提示是最常用的与索引相关的提示。INDEX 提示告诉优化程序，在指定的表上使用基于索引的扫描。虽然可以列出选择的具体索引，但是使用 INDEX 提示时，并不需要提到索引名。

例如，下面的查询使用 INDEX 提示，建议在解析查询时使用 BOOKSHELF 表上的索引：

```
select /*+ index(bookshelf bookshelf$category) */ Title
  from BOOKSHELF
 where CategoryName = 'ADULTNF';
```

依据本节前面所提供的规则，上述查询可以在不需要提示的情况下使用索引。但是，如果索引不可选择或者表很小，则优化程序可能选择忽略索引。如果对于给定的数据索引可选，则可以使用 INDEX 提示强制使用基于索引的数据访问路径。

在提示的语法中，需要指定表名(如果给出了表的别名，则也可用其别名)，还可以选择性地指定所建议的索引的名称。优化程序可以选择忽略所提供的任何提示。

如果没有在 INDEX 提示中列出特定的索引，而表中具有多个可用索引，那么优化程序判断这些可用的索引并选择扫描成本最低的索引。优化程序也可以选择扫描多个索引，并通过 AND-EQUAL 操作合并它们。

第 2 个提示，INDEX_ASC，与 INDEX 提示的功能相似：它要求按照升序扫描索引以解析指定表的查询。第 3 个基于索引的提示，INDEX_DESC，告诉优化程序以降序扫描索引(从其最大值到最小值)。使用 INDEX_FFS 提示指定全索引扫描。ROWID 提示与 INDEX 提示类似，要求在指定的表中使用 TABLE ACCESS BY INDEX ROWID 方法。AND_EQUAL 提示建议优化程序合并多索引扫描的结果。

10. 其他的索引调整问题

在表上创建索引时，通常会引起两个问题：应该使用多个索引还是单个连接索引？如果使用连接索引，哪一列将是索引的第一列？

通常，优化程序扫描单个连接索引要比扫描并合并两个不同的索引快。扫描返回的行越多，连接索引扫描的性能越好(与合并两个索引扫描的性能相比)。随着更多列添加到连索引中，范围扫描效率变低。

对于连合并索引，哪一列应该是索引的第一列呢？第一列应该是表中限制条件最常使用的列，它应该是高选择性的。在连接索引中，优化程序将基于索引第一列的选择性来估计索引的选择性(即被使用的可能性)。对于这两个标准(即在限制条件中使用和选择性最高的列)，第 1 个标准更重要。

永远也不会使用那些基于从来不在限制条件中使用的列的高选择性索引。经常在限制条件中使用的列的低选择性索引对性能提高没有什么帮助。如果不能创建一个既具有高选择性又经常使用的索引，就要考虑对被索引的列分别创建不同的索引。

很多应用程序侧重于在线事务处理而不是批处理；这样将会存在很多并发在线用户和少量的并发批处理用户。通常，基于索引的扫描比执行全表扫描允许在线用户更快地访问数据。

当创建应用程序时，必须知道在应用程序中执行的查询类型，以及查询中的限制条件。如果熟悉在数据库上执行的查询，则可以对表进行索引，这样，在线用户可以快速地检索到他们所需的数据。当数据库性能直接影响到在线业务处理时，应用程序应该执行尽量少的数据库访问。

46.5.6　操纵数据集的操作

一旦从表或索引中返回数据，就可以操纵这些数据。可以对记录进行分组、排序、统计、锁定，或把查询的结果与其他查询的结果合并(通过 union、minus 和 intersect 运算符)。以下部分将说明如何使用数据操纵操作。

操纵记录集的大多数操作不返回记录给用户，直到完成整个操作。例如，消除重复记录时对记录排序(SORT UNIQUE NOSORT 操作)不能返回记录给用户，直到所有的记录都唯一。另一方面，索引扫描操作和表访问操作找到记录后，立即返回该记录给用户。

当执行 INDEX RANGE SCAN 操作时，查询返回的第 1 行传递查询设置的限制条件标准—— 在显示第 1 条记录前不必判断返回的下一个记录。如果执行集合操作——如排序操作，则不会立即显示记录。在集合操作中，用户需要等待操作处理完所有行。因此，需要限制在线用户使用的查询执行的集合操作数量(限制应用程序的认知响应时间)。排序和分组操作在大型报表和批处理事务中最常用。

1. 对行进行排序

Oracle 有 3 个内部操作不需要对行分组，就可以对行进行排序。第 1 个是 SORT ORDER BY 操作，在查询中使用 order by 子句时，使用该操作。例如，查询 BOOKSHELF 表，按 Publisher 排序：

```
Select Title from BOOKSHELF
 order by Publisher;
```

执行上述查询时，优化程序将通过 TABLE ACCESS FULL 操作，从 BOOKSHELF 表中检索数据(因为查询没有限制条件，所以返回所有行)。检索到的记录不会立即显示给用户；在用户看到任何结果之前，SORT ORDER BY 操作将对记录排序。

有时候，对记录进行排序时，需要排序操作消除重复的部分。例如，如果想看到 BOOKSHELF 表中独一无二的 Publisher 值，该怎么做呢？查询如下所示：

```
select DISTINCT Publisher from BOOKSHELF;
```

和前面的查询一样，因为该查询没有限制条件，所以使用 TABLE ACCESS FULL 操作从 BOOKSHELF 表中检索记录。但是 distinct 关键字告诉优化程序，仅仅返回 Publisher 列的独一无二的值。

为解析该查询，优化程序获得 TABLE ACCESS FULL 操作返回的记录，并通过 SORT UNIQUE NOSORT 操作对它们进行排序。直到处理完所有记录之后，才把结果显示给用户。

除了 distinct 关键字使用之外，当使用 minus、intersect、和 union(不是 union all)函数时，也会调用 SORT UNIQUE NOSORT 操作。

第 3 个排序操作 SORT JOIN，总是作为 MERGE JOIN 操作的一部分使用，从来不单独使用。46.5.7 节将描述 SORT JOIN 对连接的性能的影响。

2. 对行进行分组

Oracle 的两个内部操作在对记录分组的同时，对行进行排序。这两个操作(SORT AGGREGATE 和 SORT GROUP BY)与分组函数(例如 MIN、MAX 和 COUNT)联合使用。查询语法确定使用哪个操作。

在下面的查询中，从 BOOKSHELF 表中选择按字母排序的最大的 Publisher 值：

```
select MAX(Publisher)
  from BOOKSHELF;
```

为了解析该查询，优化程序将执行两个不同的操作。首先，TABLE ACCESS FULL 操作从表中选择 Publisher 值。其次，通过 SORT AGGREGATE 操作，对行进行分析，返回最大的 Publisher 值给用户。

如果已对 Publisher 列创建了索引，则可以使用索引来解析关于索引的最大值或最小值的查询(如 46.5.4 节所述)。因为 Publisher 列没有索引，所以需要排序操作。直到读取完所有记录并完成 SORT AGGREGATE 操作后，查询才返回最大的 Publisher 值。

因为该查询中没有 group by 子句，所以在上述示例中使用 SORT AGGREGATE 操作。使用 group by 子句的查询将使用名为 SORT GROUP BY 的内部操作。

如果想知道每个人发行的书的数量，该怎么做呢？下面的查询使用 group by 子句从 BOOK SHELF 表中选择每个 Publisher 值发行的书的数量：

```
select Publisher, COUNT(*)
  from BOOKSHELF
 group by Publisher;
```

该查询将对每个不同的 Publisher 值返回一条记录。对每个 Publisher 值，将计算出它在 BOOKSHELF 表中出现的次数，并显示在 COUNT(*)列中。

为了解析该查询，Oracle 首先执行全表扫描(查询没有任何限制条件)。因为使用了 group by 子句，所以从 TABLE ACCESS FULL 操作返回的行将被 SORT GROUP BY 操作处理。一旦对所有的行排序分组并计算出每组的数目，就把记录返回给用户。与其他排序操作一样，直到所有记录都处理完后，才将记录返回给用户。

到目前为止，操作已经包含了简单示例—— 全表扫描、索引扫描和排序操作。大多数访问单个表的查询使用前面所描述的操作。当对在线用户调整查询时，尽量避免使用排序和分组操作，因为它们会强制用户等待处理记录。如果可能，编写允许应用程序用户在查询解析时快速接收记录的查询。执行的排序和分组操作越少，把第 1 个记录返回给用户的速度就越快。在批处理事务中，查询的性能由它完成的总时间，而不是由返回第 1 行的时间来判断。所以，批处理事务可以使用排序和分组操作，而不会影响应用程序认知性能。

如果应用程序不需要在显示查询结果之前排序所有的行，则可以考虑使用 FIRST_ROWS 提示。FIRST_ROWS 告诉优化程序选择不执行集合操作的执行路径。

3. UNION、MINUS 和 INTERSECT

union、minus 和 intersect 子句对多个查询的结果进行处理和比较。每个函数具有一个关联的操作—— 操作的名称为 UNION、MINUS、和 INTERSECT。

以下查询从 BOOKSHELF 表和 BOOK_ORDER 表中选择所有的 Title 值：

```
select Title
  from BOOKSHELF
 UNION
select Title
  from BOOK_ORDER;
```

执行上述查询时，优化程序将分别执行每个查询，然后合并结果。第 1 个查询是：

```
select Title
  from BOOKSHELF
```

因为该查询中没有限制条件，而且索引了 Title 列，所以将扫描 BOOKSHELF 表中的主键索引。

第 2 个查询是：

```
select Title
  from BOOK_ORDER
```

因为该查询中没有限制条件，而且索引了 Title 列，所以将扫描 BOOK_ORDER 表中的主键索引。

因为查询执行了两个查询结果的 union，所以通过 UNION-ALL 操作对两个结果集进行合并。因为使用 union 运算符强制 Oracle 消除重复的记录，所以在把记录返回给用户之前，通过 SORT UNIQUE NOSORT 操作对结果集进行处理。如果查询使用 union all 子句而不是 union，则不需要 SORT UNIQUE NOSORT 操作。查询将是：

```
select Title
  from BOOKSHELF
 UNION ALL
select Title
  from BOOK_ORDER;
```

处理这个修改的查询时，优化程序将执行这两个查询要求的扫描，然后执行 UNION-ALL 操作。因为 union all 子句不消除重复记录，所以不需要 SORT UNIQUE NOSORT 操作。

处理 union 查询时，优化程序分别处理每个需要合并(union)的查询。虽然上述程序清单中所显示的所有示例均涉及全表扫描的简单查询，但 union 查询非常复杂，具有相对复杂的执行路径。直到所有记录被处理完后，才将结果返回给用户。

注意：

UNION ALL 是行操作。UNION 是集合操作，它包括 SORT UNIQUE NOSORT。

当使用 minus 子句时，处理查询的方式与 union 示例中使用的执行路径非常类似。在下

面的查询中，对来自 BOOKSHELF 表和 BOOK_ORDER 表中的 Title 值进行比较。如果 Title 值存在于 BOOK_ORDER 表中，而不存在于 BOOKSHELF 表中，则查询将返回该值。换句话说，希望看到订单中所有现在还没有的 Title。

```
select Title
  from BOOK_ORDER
MINUS
select Title
  from BOOKSHELF;
```

执行查询时，分别执行两个需要 minus 的查询。第 1 个查询

```
select Title
  from BOOK_ORDER
```

需要基于 BOOK_ORDER 主键的索引扫描。第 2 个查询

```
select Title
  from BOOKSHELF
```

需要 BOOKSHELF 主键索引的全索引扫描。

为了执行 minus 子句，全表扫描返回的每个记录集都通过 SORT UNIQUE NOSORT 操作进行排序(进行行排序，并消除重复的值)。MINUS 操作处理排序后的行集合。直到每个查询返回的每个记录集被排序后，才执行 MINUS 操作。直到排序操作完成后，才返回两个排序操作的结果，因此知道两个 SORT UNIQUE NOSORT 操作都完成后，才开始 MINUS 操作。与 union 查询示例一样，用于说明 MINUS 操作的示例查询对于在线用户执行效果不好(在线用户根据查询返回第 1 行的速度来评价性能)。

intersect 子句比较两个查询的结果并确定它们具有的相同行。以下查询确定同时存在于 BOOKSHELF 表和 BOOK_ORDER 表中的 Title 值：

```
select Title
  from BOOK_ORDER
INTERSECT
select Title
  from BOOKSHELF;
```

为处理 INTERSECT 查询，优化程序首先分别判断每个查询。第 1 个查询

```
select Title
  from BOOK_ORDER
```

需要基于 BOOK_ORDER 主键索引的全索引扫描。第 2 个查询

```
select Title
  from BOOKSHELF
```

需要 BOOKSHELF 主键索引的全索引扫描。两个表扫描的结果分别通过 SORT UNIQUE NOSORT 操作进行处理。换句话说，对来自 BOOK_ORDER 表的行进行排序，对来自 BOOKSHELF 表的行也进行排序。INTERSECT 操作比较这两个已经排序过的结果，通过

intersect 子句返回这两个排序操作返回的 Title 值。

使用 intersect 子句的查询的执行路径需要使用 SORT UNIQUE NOSORT 操作。因为直到排序完所有行集合后，SORT UNIQUE NOSORT 操作才返回记录给用户，所以在 INTERSECT 操作执行之前，使用 intersect 子句的查询将不得不等待这两个排序完成。由于对排序操作的依赖，使用 intersect 子句的查询将不返回任何记录给用户，直到排序完成。

union、minus 和 intersect 子句在返回任何行给用户之前，都包含对行集合进行的处理。即使调整所包含的表访问，应用程序的在线用户可能也会感觉到使用这些函数的查询执行效果差。对排序操作的依赖将影响第 1 行返回用户的速度。

4. 从视图中选择

创建视图时，Oracle 将存储该视图所基于的查询。例如，下面的视图基于 BOOKSHELF 表：

```
create or replace view ADULTFIC as
select Title, Publisher
  from BOOKSHELF
 where CategoryName = 'ADULTFIC';
```

从 ADULTFIC 视图中选择时，优化程序将利用查询中的条件，并把它们与视图的查询文本进行合并。如果在视图的查询中指定限制条件，则那些限制条件将(如果有可能)应用于视图的查询文本。例如，如果执行查询：

```
select Title, Publisher
  from ADULTFIC
 where Title like 'T%';
```

则优化程序将把限制条件

```
where Title like 'T%';
```

与视图的查询文本合并，它将执行查询：

```
select Title, Publisher
  from BOOKSHELF
 where CategoryName = 'ADULTFIC'
   and Title like 'T%';
```

在本示例中，视图对查询的性能没有任何影响。当视图的文本与查询的限制条件合并时，优化程序的可用选项增加；它可以选择更多的索引和更多的数据访问路径。

视图处理的方式取决于视图基于的查询。如果视图的查询文本不能与使用视图的查询合并，则在应用其他条件之前，首先解析视图。考虑下面的视图：

```
create or replace view PUBLISHER_COUNT as
select Publisher, COUNT(*) Count_Pub
  from BOOKSHELF
 group by Publisher;
```

PUBLISHER_COUNT视图将显示BOOKSHELF表中的每个不同的Publisher值所对应的行，以及具有该值的记录数量。PUBLISHER_COUNT 视图的 Count_Pub 列记录每个不同的Publisher值的数量。

优化程序如何处理下列对 PUBLISHER_COUNT 视图的查询？

```
select *
  from PUBLISHER_COUNT
 where Count_Pub > 1;
```

查询引用视图的 Count_Pub 列。但是，查询的 where 子句不能与视图的查询文本合并，因为 Count_Pub 是通过分组操作创建的。直到来自于 PUBLISHER_COUNT 视图的结果集解析完之后，才能应用 where 子句。

在应用查询的其他条件之前，解析包含分组操作的视图。就像排序操作一样，包含分组操作的视图不返回任何记录，直到处理完整个结果集。如果视图不包含分组操作，查询文本就可以与从视图中选择的查询的限制条件合并。这样的结果是，包含分组操作的视图限制了优化程序可用选择的数量，并且直到处理完所有行后，才返回记录—— 在线用户查询这样的视图时，执行效果不好。

处理查询时，优化程序首先解析视图。因为视图的查询是

```
select Publisher, COUNT(*) Count_Pub
  from BOOKSHELF
 group by Publisher;
```

优化程序将通过 TABLE ACCESS FULL 操作从 BOOKSHELF 表中读取数据。由于使用了 group by 子句，因此 SORT GROUP BY 操作将处理来自于 TABLE ACCESS FULL 操作的行。接着，另外的操作(FILTER)将处理数据。FILTER 操作用于消除基于查询中条件的行：

```
where Count_Pub > 1
```

如果使用具有 group by 子句的视图，则直到视图处理完所有行之后，才从视图中返回行。这样的结果是，需要很长时间才能得到查询返回的第 1 行，在线用户感觉到视图的性能是不能接受的。如果可以从视图中删除排序和分组操作，则将增加视图文本与调用该试图的查询文本合并的可能性——这样可能促进性能的提高(尽管查询可能使用其他集合操作，这些集合操作对性能有负面影响)。

当查询文本与视图文本合并时，优化程序的可用选项将增加。例如，合并查询的限制条件与视图的限制条件使前面不可用的索引可以在查询执行过程中使用。

通过 NO_MERGE 提示，可以禁止查询文本和视图文本的自动合并。PUSH_PRED 提示强制把谓词连接到视图中；PUSH_SUBQ 引起非合并子查询在执行路径上尽快地执行。

5. 从子查询中选择

只要可能，优化程序就会合并子查询的文本与查询的其他部分的文本。例如，考虑下面查询：

```
select Title
  from BOOKSHELF
 where Title in
(select Title from BOOK_ORDER);
```

优化程序在判断上述查询时，将确定该查询与下面 BOOKSHELF 表和 BOOK_ORDER 表的连接功能相等：

```
select BOOKSHELF.Title
  from BOOKSHELF, BOOK_ORDER
 where BOOKSHELF.Title = BOOK_ORDER.Title;
```

因为现在查询写成连接，所以优化程序有很多可用的操作能够处理数据(如 46.5.7 节所述)。

如果子查询不能解析为连接，则在其他查询文本处理之前对它进行解析——这与 FILTER 操作在视图上使用的方式类似。事实上，如果子查询不能与查询的其他部分合并，则子查询可以使用 FILTER 操作！

依赖分组操作的子查询与包含分组操作的视图具有同样的调整问题。在应用查询的其他限制条件之前，必须完全处理完来自子查询的行。

46.5.7　执行连接的操作

通常，单个查询需要从多个表中选择列。为了从多个表选择数据，在 SQL 语句中对表进行连接——表在 from 子句中列出，连接条件在 where 子句中列出。在下面示例中，基于 Title 列的值对 BOOKSHELF 表和 BOOK_ORDER 表进行连接：

```
select BOOKSHELF.Title
  from BOOKSHELF, BOOK_ORDER
 where BOOKSHELF.Title = BOOK_ORDER.Title;
```

它可以重写为：

```
select Title
  from BOOKSHELF inner join BOOK_ORDER
 using (Title);
```

或者：

```
select Title
  from BOOKSHELF natural inner join BOOK_ORDER;
```

连接条件与连接的限制条件功能一样。因为在第 1 个程序清单的 where 子句中，BOOKSHELF.Title 列等于某个值，所以优化程序可以在查询的执行过程中使用 BOOKSHELF.Title 列上的索引。如果 BOOK_ORDER.Title 列上的索引可用，则优化程序也会考虑使用该索引。

Oracle 有 3 个主要方法处理连接：MERGE JOIN 操作、NESTED LOOPS 操作以及 HASH JOIN 操作。根据查询中的条件、可用的索引、可用的统计信息，优化程序将选择使用哪个连

接操作。根据应用程序和查询的特性，可以强制优化程序使用与它首选的连接方法不同的方法。下面几节将介绍不同连接方法的特征，以及每个方法分别在什么条件下最有用。

46.5.8 Oracle 如何处理两个以上表的连接

如果查询连接两个以上的表，优化程序就把该查询作为多个连接组对待。例如，如果查询连接 3 个表，那么优化程序将这样执行连接：先把两个表连接在一起，然后把连接的结果集与第 3 个表连接。初始连接的结果集的大小将影响其余连接的性能。如果初始连接的结果集太大，那么第 2 个连接将处理很多行。

如果查询连接 3 个大小不同的表(如很小的表 SMALL，中等大小的表 MEDIUM，以及大型表 LARGE)，则必须注意表的连接顺序。如果连接 MEDIUM 表和 LARGE 表返回很多行，那么连接该连接的结果集与 SMALL 表将需要执行大量工作。另一选择是，如果首先连接 SMALL 表和 MEDIUM 表，然后把 SMALL-MEDIUM 连接的结果集与 LARGE 表连接，则使查询执行的工作量最少。

1. MERGE JOIN

在 MERGE JOIN 操作中，分别处理连接的两个输入，对它们进行排序，然后连接。在查询的限制条件没有可用的索引时，通常使用 MERGE JOIN 操作。

在下面的查询中，连接 BOOKSHELF 表和 BOOKSHELF_AUTHOR 表。如果两个表在其 Title 列上都没有索引，那么查询过程中没有可以使用的索引(因为查询中没有其他限制条件)。该示例使用提示强制使用合并连接：

```
select /*+ USE_MERGE (bookshelf, bookshelf_author)*/
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title
  and BOOKSHELF.Publisher > 'T%';
```

或者

```
select /*+ USE_MERGE (bookshelf, bookshelf_author)*/
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF inner join BOOKSHELF_AUTHOR
   on BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title
 where BOOKSHELF.Publisher > 'T%';
```

为了解析该查询，优化程序选择执行表的 MERGE JOIN。为了执行 MERGE JOIN，分别读取每个表(通常通过 TABLE ACCESS FULL 操作或全索引扫描)。通过 SORT JOIN 操作，对从 BOOKSHELF_AUTHOR 表全表扫描返回的行集合进行排序。从 BOOKSHEL 表的 INDEX FULL SCAN 和 TABLE ACCESS BY INDEX ROWID 返回的行集合已经进行了排序(在索引中)，因此，在这部分执行路径上不需要其他的 SORT JOIN 操作。接着，通过 MERGE JOIN 操作对来自于 SORT JOIN 操作的数据进行合并。

当使用 MERGE JOIN 操作连接两组记录时，在连接之前，分别对每组记录进行处理。直

到接收到两个 SORT JOIN 操作的数据之后(为 MERGE JOIN 操作提供输入)，MERGE JOIN 操作才开始。SORT JOIN 操作将不提供数据给 MERGE JOIN 操作，直到排序完所有行。如果索引用作数据源，则不进行 SORT JOIN 操作。

如果 MERGE JOIN 操作必须等待两个独立的 SORT JOIN 操作完成，那么使用 MERGE JOIN 操作的连接通常对在线用户执行效果很差。感觉到性能差是因为延迟了将连接的第 1 行返回给用户。随着表的不断增大，完成排序所需要的时间也大大地增加。如果两个表的大小相差很大，那么执行在较大表上执行的排序操作将对整个查询的性能起负面影响。

因为 MERGE JOIN 操作包含全表扫描和所涉及表的排序，所以只有在两个表都很小或者都很大的时候，才使用 MERGE JOIN 操作。如果两个表都很小，则将很快完成表的扫描和排序过程。如果两个表都很大，那么 MERGE JOIN 操作所需要的排序和扫描操作可以利用 Oracle 的并行选项。

Oracle 可以并行化操作，允许多个处理器参与单个命令的执行。其中可以并行的操作是 TABLE ACCESS FULL 和排序操作。由于 MERGE JOIN 使用 TABLE ACCESS FULL 和排序操作，因此它可以很好地利用 Oracle 的并行选项。包含 MERGE JOIN 操作的并行化查询往往会提高查询的性能(需要足够可用的系统资源支持并行操作)。

2. NESTED LOOPS

NESTED LOOPS 操作通过循环方法连接两个表：对一个表中的记录进行检索，且对于每个检索的记录，执行对第两个表的访问。通过基于索引的访问执行对第 2 个表的访问。

下面的程序清单显示了来自于前面 MERGE JOIN 部分的查询形式。使用提示推荐对 BOOKSHELF 表进行基于索引的访问：

```
select /*+ INDEX(bookshelf) */
       BOOKSHELF.Publisher,
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;
```

因为 BOOKSHELF 表的 Title 列用作查询中连接条件的一部分，所以主键索引可以解析该连接。当执行查询时，使用 NESTED LOOPS 操作执行该连接。

为了执行 NESTED LOOPS 连接，优化程序必须首先选择连接的驱动表(driving table)，驱动表是首先读取(虽然经常看见索引扫描，但通常通过 TABLE ACCESS FULL 操作)的表。对于驱动表中的每个记录，将对连接中的第 2 个表进行查询。示例查询基于 Title 列上的值，连接 BOOKSHELF 表和 BOOKSHELF_AUTHOR 表。在 NESTED LOOPS 执行期间，操作将从 BOOKSHELF_AUTHOR 表中选择所有记录。将对 BOOKSHELF 表上的主键索引进行探查，以确定它是否包含来自于 BOOKSHELF_AUTHOR 表的与当前记录值相同的项。如果发现匹配的项，Title 值将从 BOOKSHELF 主键索引返回。如果需要 BOOKSHELF 表的其他列，则将通过 TABLE ACCESS BY INDEX ROWID 操作，从 BOOKSHELF 表中选择行。

在 NESTED LOOPS 连接中至少包含两个数据访问操作：对驱动表的访问和被驱动表的访问，后者通常基于索引。最常使用的数据访问方法——TABLE ACCESS FULL、TABLE

ACCESS BY INDEX ROWID 以及索引扫描——只要找到记录后，立即把记录返回下一个操作；它们不等到选择完整个记录集。因为这些操作可以很快把最先匹配的行提供给用户，所以在线用户经常执行的连接通常使用 NESTED LOOPS 连接。

当实现 NESTED LOOPS 连接时，需要考虑驱动表的大小。如果驱动表比较大，且通过全表扫描对它进行读取，则在它上面执行的 TABLE ACCESS FULL 操作对查询的性能就会有负面影响。如果连接双方都有可用的索引，则 Oracle 将选择查询的驱动表。优化程序检查表的大小的统计信息和索引的选择性，选择总成本最低的路径。

当把 3 个表连接在一起时，Oracle 执行两个不同的连接：连接两个表，生成一组记录，然后把这组记录与第 3 个表连接起来。如果使用 NESTED LOOPS 连接，则表的连接顺序将很严格。第 1 个连接的输出产生 1 组记录，该组记录作为第 2 个连接的驱动表。

第 1 个连接返回的记录集的大小影响第 2 个连接的性能——因此对整个查询的性能有显著影响。尽量首先连接选择性最高的表，这样，这些连接对后来连接的影响将可以忽略。如果在多连接的查询中首先对大型表进行连接，那么表的大小将影响每个后续的连接，对整个查询的性能有负面影响。优化程序将选择合适的连接路径；可以通过生成说明计划，确认所用成本和连接路径。

当连接中的表的大小不一时，使用 NESTED LOOPS 连接效率最高—— 可以把较小的表作为驱动表，通过基于索引的访问对大型表进行选择。索引的选择性越高，就可以越快地完成查询。

3. HASH JOIN

优化程序可以动态地选择使用 HASH JOIN 操作代替 MERGE JOIN 或 NESTED LOOPS，来执行连接。HASH JOIN 操作在内存中对两个表进行比较。在散列连接期间，扫描第 1 个表，数据库把“散列”函数应用到数据，以便为连接准备表，然后读取第 2 个表的值(通常通过 TABLE ACCESS FULL 操作)，散列函数比较第 2 个表与第 1 个表。将匹配的行返回给用户。

即使有可用的索引，优化程序也可以选择执行散列连接。在下面的程序清单显示的样本查询中，基于 Title 列，对 BOOKSHELF 表和 BOOKSHELF_AUTHOR 表进行连接：

```
select /*+ USE_HASH (bookshelf) */
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;
```

BOOKSHELF 表在其 Title 列上有唯一索引。因为索引可用，并可用于判断连接的条件，所以优化程序可以选择执行两个表的 NESTED LOOPS 连接(如上一部分所示)。如果执行散列连接，那么通过不同的操作对每个表进行读取。来自表和索引扫描的数据将作为 HASH JOIN 操作的输入。散列连接不依赖于处理行集合的操作。包含在散列连接中的操作快速地把记录返回给用户。如果表比较小且可以快速扫描，那么在线用户执行的查询使用散列连接最为合适。

4. 处理外部连接

当处理外部连接时，优化程序将使用前面描述的 3 种方法之一。例如，如果示例查询执行 BOOK_ORDER 和 CATEGORY 之间的外部连接(查看没有定购哪些种类的书)，则使用 NESTED LOOPS OUTER 操作代替 NESTED LOOPS 操作。在 NESTED LOOPS OUTER 操作中，外部连接的“外部”表通常作为查询的驱动表；当扫描内部表的记录以查找匹配的记录时，如果没有匹配的行，则返回 NULL 值。

5. 相关的提示

可以使用提示重写优化程序选择的连接方法。提示允许指定所使用连接方法的类型或者连接方法的目标。

除了本部分描述的提示之外，还可以使用前面描述的 FULL 和 INDEX 提示，来影响连接的处理方式。例如，如果使用提示强制使用 NESTED LOOPS 连接，那么也可以使用 INDEX 提示指定在 NESTED LOOPS 连接中使用的索引，以及通过全表扫描访问的表。

6. 关于目标的提示

可以指定提示，指导优化程序执行指定目标的查询。与连接相关的可用目标如下：

- ALL_ROWS　执行查询，以便尽快返回所有行。
- FIRST_ROWS　执行查询，以便尽快返回第 1 行。

默认情况下，优化程序将选择解析查询所需要的总时间最短的执行路径去执行查询。因此，默认使用 ALL_ROWS 作为目标。如果优化程序仅仅关注于查询返回所有行所需要的总时间，那么可以使用基于集合的操作，如排序和 MERGE JOIN。但是 ALL_ROWS 目标不是永远都能令人满意。例如，在线用户趋向于根据查询返回第 1 行数据所需的时间来判断系统的性能。因此用户选择 FIRST_ROWS 作为其主要目标，把返回所有行所需的时间作为第 2 目标。

可用的提示模仿这些目标：ALL_ROWS 提示允许优化程序选择所有可用的操作，以最小化查询所需的总的处理时间；FIRST_ROWS 提示告诉优化程序，选择能够使第 1 行返回用户所需的时间最短的执行路径。

如果使用 FIRST_ROWS 提示，则优化程序将较少地使用 MERGE JOIN，而较多地使用 NESTED LOOPS。

7. 关于方法的提示

当判断可选的连接方法时，除了指定优化程序使用的目标之外，还可以列出要使用的指定操作以及这些操作用在哪些表上。如果查询仅包含两个表，当为要使用的连接方法提供提示时，就不需要指定连接的表。

USE_NL 提示告诉优化程序，使用 NESTED LOOPS 操作对表进行连接。在下面的示例中，为连接查询示例指定 USE_NL 提示。在该提示中，列出 BOOKSHELF 表作为连接的内部表。

```
select /*+ USE_NL(bookshelf) */
       BOOKSHELF.Publisher,
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;
```

如果希望多表查询中的所有连接都使用 NESTED LOOPS 操作，则指定没有表引用的 USE_NL 提示即可。通常，因为不知道将来如何使用该查询，所以每当使用提示指定连接方法时，都应该指定表名。甚至不知道当前数据库对象是如何创建的——例如，from 子句中的其中一个对象可能是使用 MERGE JOIN 操作调整过的视图。

如果在提示中指定表，则应当引用该表的别名或非限定的表名。也就是说，如果 from 子句引用表。

```
from FRED.HIS_TABLE, ME.MY_TABLE
```

就不能指定像这样的提示

```
/*+ USE_NL(ME.MY_TABLE) */
```

相反，应当通过表名引用该表，而不需要所有者。

```
/*+ USE_NL(my_table) */
```

如果多个表具有相同的名称，则应当把表别名赋值给这些表，并在提示中引用这些别名。例如，如果连接表自身，那么 from 子句可以包括下面程序清单中显示的文本：

```
from BOOKSHELF B1, BOOKSHELF B2
```

编写强制 BOOKSHELF-BOOKSHELF 连接使用 NESTED LOOPS 的提示时将使用表别名，如下面的程序清单所示：

```
/*+ USE_NL(b2) */
```

优化程序将忽略任何语法不正确的提示。用不正确语法编写的提示将作为注释处理(因为它包含在“/*”和“*/”字符之间)。

如果正在使用 NESTED LOOPS 连接，则需要注意表的连接顺序。ORDERED 提示(与 NESTED LOOPS 连接一起使用时)影响表的连接顺序。

当使用 ORDERED 提示时，表将按照其在查询的 from 子句中列出的顺序进行连接。如果 from 子句包含 3 个表，例如

```
from BOOK_ORDER, BOOKSHELF, BOOKSHELF_AUTHOR
```

那么第 1 个连接将连接前面两个表，再对该连接的结果与第 3 个表进行连接。

因为连接的顺序对于 NESTED LOOPS 连接的性能至关重要，所以 ORDERED 提示经常与 USE_NL 提示联合使用。如果使用提示指定连接的顺序，则需要确保随着时间的推移，连接的表中值的相对分布不发生显著变化；否则，指定的连接顺序今后可能导致性能问题。

可以使用 USE_MERGE 提示告诉优化程序执行指定表之间的 MERGE JOIN。在以下程

序清单中，该提示指导优化程序在 BOOKSHELF 表和 BOOKSHELF_AUTHOR 表之间执行 MERGE JOIN 操作：

```
select /*+ USE_MERGE (bookshelf, bookshelf_author)*/
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title
   and BOOKSHELF.Publisher > 'T%';
```

可以使用 USE_HASH 提示告诉优化程序，考虑使用 HASH JOIN 方法。如果没有指定任何表，那么优化程序将根据可用的统计信息选择要扫描的第 1 个表到内存中。

```
select /*+ USE_HASH (bookshelf) */
       BOOKSHELF.Publisher,
       BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title = BOOKSHELF_AUTHOR.Title;
```

8. 其他调整问题

正如在 NESTED LOOPS 和 MERGE JOIN 操作的讨论中所提到的，采用不同的操作，查询返回第 1 行的时间也不同。由于 MERGE JOIN 操作依赖于基于集合的操作，因此直到处理完所有行之后才把记录返回用户。另一方面，NESTED LOOPS 立即把可用的行返回用户。

因为 NESTED LOOPS 连接能够快速地把行返回给用户，所以它们常用于在线用户频繁执行的查询。但是，它们返回第 1 行的效率通常受到应用在所选择行上且基于集合的操作的影响。例如，当把 order by 子句添加到查询中时，就在查询处理的最后添加了 SORT ORDER BY 操作——直到排序完所有行后，才把行显示给用户。

如 46.5.7 所述，在列上使用函数将阻止数据库在数据搜索时使用该列上的索引，除非创建基于函数的索引。

在 Oracle 早前版本中使用的某些禁用索引的技术，已经不再可用。例如，开发人员在连接所涉及的每一列中添加一个空字符串，以禁止索引的使用：

```
select BOOKSHELF_AUTHOR.AuthorName
  from BOOKSHELF, BOOKSHELF_AUTHOR
 where BOOKSHELF.Title||'' = BOOKSHELF_AUTHOR.Title||'';
```

从 Oracle Database 10g 开始，优化程序不允许空字符串阻止索引的使用—— 查询使用 BOOKSHELF.Title 列上的索引作为 HASH JOIN 执行的一部分。

正如在 NESTED LOOPS 操作中提到的，连接的顺序和所选择的连接方法同样重要。如果大型的或非选择性的连接是一系列连接中的第 1 个连接，那么所返回的大型数据集合将对查询中后面的连接以及整个查询的性能有负面影响。

根据提示、优化程序目标以及统计信息，优化程序可以在同一查询中选择使用不同的连接方法。例如，对于 3 个表的查询，优化程序对其中两个表使用 NESTED LOOPS 连接，然后对 NESTED LOOPS 的输出和第 3 个表进行 MERGE JOIN。当使用 ALL_ROWS 优化程序

目标时，通常应用这样的连接类型进行合并。

为了查看操作的顺序，可以使用 set autotrace on 命令查看执行路径(如本章前面所述)。

46.5.9 并行化和缓存问题

除了前面几节所列出的操作和提示之外，还可以使用其他两组提示来影响查询的处理。其他提示允许控制查询的并行化和数据缓存中缓存的数据。

如果服务器具有多个处理器，而被查询的数据分布在多个设备上，那么可以通过并行化数据访问和排序操作，改善查询的处理。当查询并行化后，多个处理器将并行工作，每个处理器都对数据进行访问或排序。查询协调程序对工作负载进行分配并将查询的结果收集返回给用户。

并行度(degree of parallelism)——为某个查询启动的扫描或排序进程的数量——可以设置成表级(请参阅附录 A 中的“create table”命令)。优化程序将检测并行度的表级设置，确定查询中使用的查询服务器进程的数量。优化程序将动态检查处理器的数量和查询涉及的设备，并根据可用的资源做出并行化决策。

可以通过 PARALLEL 提示，影响查询的并行化。在下面的示例中，BOOKSHELF 表的查询的并行度设置为 4：

```
select /*+ PARALLEL (bookshelf,4) */ *
 from BOOKSHELF;
```

在上述查询中，添加了第 2 个操作——order by Publisher 子句的 SORT ORDER BY 操作。由于排序操作也可以并行化，因此查询使用 9 个查询服务器进程而不是 5 个(4 个用于全表扫描，4 个用于排序，1 个用于查询协调程序)。优化程序将根据表的设置、数据库布局以及可用的系统资源，动态确定并行度。

如果希望禁用查询的并行化，则可以使用 NO_PARALLEL 提示(在 Oracle Database 10g 之前的版本中称为 NOPARALLEL)。如果希望在那些通常都以并行化方式执行查询的表上串行地执行查询，则可以使用 NO_PARALLEL 提示。

如果通过全表扫描读取小型的、经常访问的表，则该表的数据可尽可能长时间地保存在 SGA 中。可以使用 create table 或 after table 命令的 buffer_pool 子句标记该表属于“保持(Keep)”池。如果已经指定表存储在 Keep 池中，则它在内存中的块从主缓存器中分离。如果 Keep 池足够大，则这些块将保留在内存中，直到数据库关闭。Keep 池必须由 DBA 配置，且必须配置得足够大，以便容纳使用它的那些表。如果 Keep 池创建得太小，则会适得其反——即对于大多数活动表，需要更多的物理读。

如果表经常被很多查询或用户使用，而且查询没有适当的索引模式，则可以使用 Keep 池提高该表的访问性能。该方法对静态表尤其有用。

46.6 实现存储概要

当从一个数据库移动到另一个数据库时，查询的执行路径可能会改变。执行路径因为以

下几个原因而修改：

- 在不同数据库上可以使用不同的优化程序(在一个数据库上基于成本，在另一个数据库上基于规则)。
- 在不同数据库上可以启用不同的优化程序功能。
- 不同数据库之间被查询的表的统计信息不同。
- 不同数据库之间收集统计信息的频率不同。
- 不同数据库可能运行不同版本的 Oracle 内核。

这些不同对执行路径的影响巨大，当移动或更新应用程序时对查询性能有显著的负面影响。为了使这些不同对查询性能的影响最小化，Oracle 在 Oracle8i 中引入了存储概要(stored outline)特性。

存储概要存储查询的一组提示。每次执行该查询时，都使用这些提示。使用存储提示将增加查询每次使用相同执行路径的可能性。虽然提示不指定执行路径(它们是提示，而不是命令)，但是减少了数据库移动对查询性能的影响。

为了创建概要，必须具有 CREATE ANY OUTLINE 系统权限。使用 create outline 命令创建查询的概要，如以下程序清单所示：

```
create outline REORDERS
    for category DEVELOPMENT
     on

select BOOKSHELF.Title
  from BOOKSHELF, BOOK_ORDER
 where BOOKSHELF.Title = BOOK_ORDER.Title;
```

如果不指定概要的名称，则系统将自动生成概要的名称。

注意：
可以为 DML 命令和 create table as select 命令创建概要。

一旦创建了概要，就可以更改它。例如，可能需要改变概要，以反映数据量和分布上的显著变化。可以使用 alter outline 命令的 rebuild 子句重新生成查询执行期间使用的提示：

```
alter outline REORDERS rebuild;
```

注意：
必须具有 ALTER ANY OUTLINE 系统权限，才能使用该命令。

也可以通过 alter outline 命令的 rename 子句，重命名概要：

```
alter outline REORDERS rename to BOOK_REORDERS;
```

可以通过 alter outline 命令的 change category 子句，修改概要的类别。

可以通过设置会话中(通过 alter session)的 USE_STORED_ OUTLINES 参数为 TRUE，启用存储概要。

为管理存储概要，使用 OUTLN_PKG 程序包，它提供 3 种功能：

- 删除从未使用过的概要
- 删除指定类别中的概要
- 在类别之间移动概要

这 3 个功能都在 OUTLN_PKG 程序包中有对应的过程。执行 DROP_UNUSED 过程删除从未使用过的概要，如下面的程序清单所示：

```
execute OUTLN_PKG.DROP_UNUSED;
```

执行 DROP_BY_CAT 过程，删除指定类别中的所有概要(该过程把该类别的名称作为其唯一的输入参数)。下面示例删除 DEVELOPMENT 类别中的所有概要：

```
execute OUTLN_PKG.DROP_BY_CAT -
    (cat => 'DEVELOPMENT');
```

使用 UPDATE_BY_CAT 过程把概要从旧类别重新分配到新类别中，如下面示例所示：

```
execute OUTLN_PKG.UPDATE_BY_CAT -
  (old_cat => 'DEVELOPMENT', -
  new_cat => 'TEST');
```

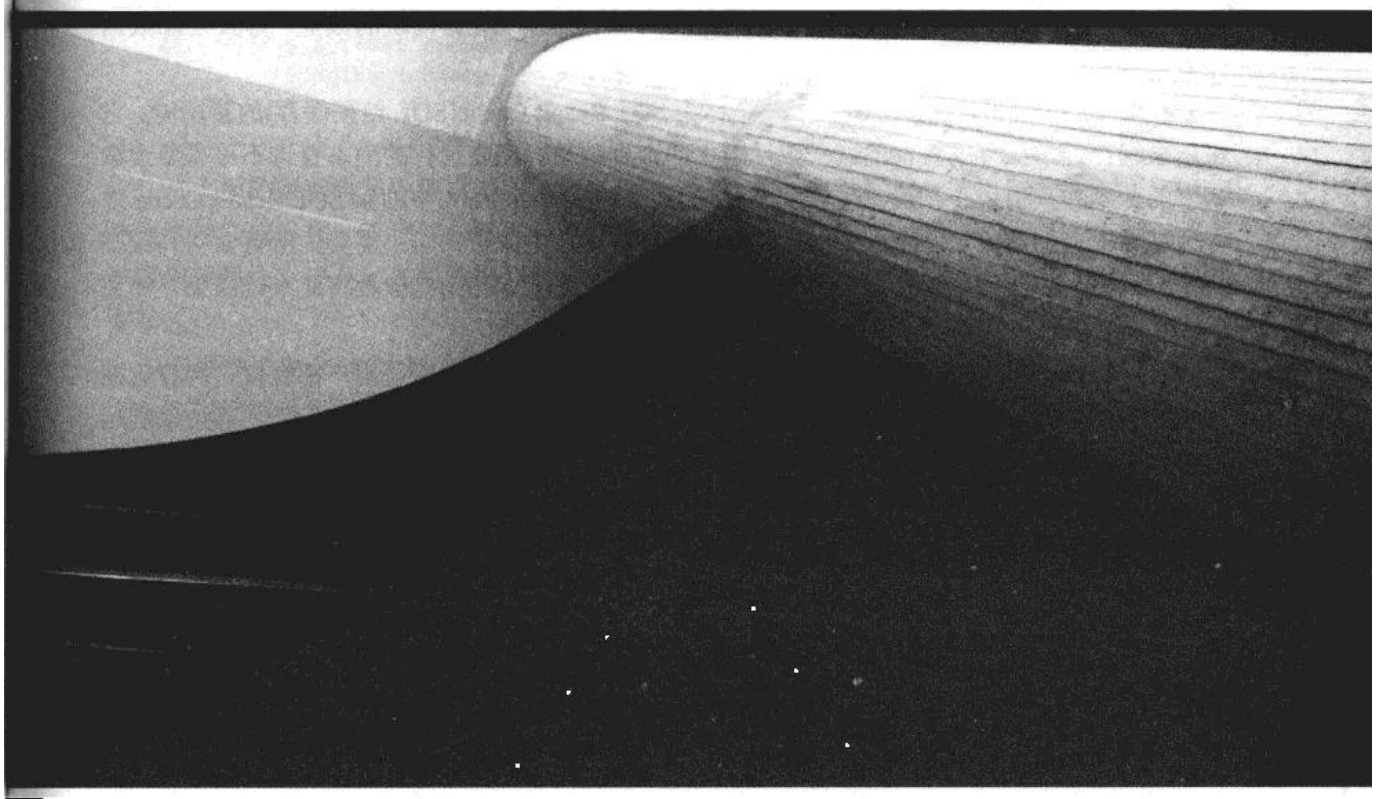
使用 drop outline 命令删除指定的概要。

46.7 小结

前面几节描述了优化程序访问和操作数据所使用的方法。当调整查询时，请牢记下面几个因素：

- **用户的目标** 用户希望快速地返回行，还是更加关注查询所需的总吞吐时间？
- **应用程序设计** 查询如何放置在应用程序的整个设计中？它如何执行？是否还有外部因素影响其性能？
- **连接使用的类型** 表是否正确地创建索引？
- **表的大小** 所查询的表有多大？如果是多个连接，那么每个连接返回的数据集是多大？
- **数据的选择性** 使用的索引的选择性是多大？
- **排序操作** 执行了多少排序操作？是否执行了嵌套排序操作？

如果可以小心处理这些与性能相关的问题，则数据库中的“问题”查询的数量将会减少。如果查询占用更多的资源(与它应该占用的资源相比)，则应使用 set autotrace on 或 explain plan 命令确定查询使用的操作顺序。可以使用本章所提供的提示和讨论对操作或操作组进行调整。



第47章

SQL结果缓存和客户端查询缓存

设计数据库应该程序的主要目的是尽快返回正确的结果。应用程序开发人员花费大量的时间努力编写高效率的SQL代码，Oracle DBA调整数据库内存结构以便能够高效率地访问满足查询需要的数据和索引块。

调整应用程序通常涉及寻找访问 Oracle 数据库缓冲区缓存中的数据和/或索引块的最佳路径。访问内存中的数据总是比访问磁盘数据快得多。Oracle 使用算法将使用率高的数据块存储在内存中，以减少读取物理磁盘的次数。对于使用率高的表和索引，Oracle DBA 可以增

加默认缓冲区缓存的大小，或利用 KEEP 缓冲池增加内存中驻留的块的数量。即使全部数据和索引块在内存中，查询也必须读它们才能获得结果集。

在 Oracle 11g 中，Oracle 引入了一个称为 SQL Result Cache(SQL 结果缓存)的新功能。它可用来缓存经常执行的查询的读一致性版本的查询结果。反复运行的静态查询不再需要读取基本的数据和索引块，相反，可以从 SGA 的共享池(Shared Pool)中的新 SQL Result Cache 内存区域读取查询结果。运行一次 SQL 语句，就可以读取生成查询结果需要的所有索引和数据块，可以根据需要通过 SQL 语句中的提示或通过设置数据库初始化参数将此查询结果存储在 SQL Result Cache 中。

SQL Result Cache 存储在 SGA 中，以便连接到数据库实例的所有用户都有权访问存储在这里的查询结果。执行包含 RESULT_CACHE 提示的相同 SQL 语句的任何用户都可以利用 SQL Result Cache，且不必执行物理和逻辑 I/O 来生成查询输出。

下面来看看 SQL Result Cache 是如何工作的。下面的查询执行全表扫描来生成输出。无论是相同的用户还是不同的用户，每次运行此查询时都必须访问所有相同的数据块。

```
SQL> SET AUTOT ON
SQL> select state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST    PROD_CODE     SUM(AMT)
--  ---------- ----------
IL       100860      16520
IL       100861     189180
...
MI       200380        208
55 rows selected.
Elapsed: 00:00:00.45
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
--------------------------------------------------------------------------------
|Id |Operation          |Name  | Rows | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
| 0 | SELECT STATEMENT  |      | 66   | 792   |     317 (9) | 00:00:04 |
| 1 | SORT GROUP BY     |      | 66   | 792   |     317 (9) | 00:00:04 |
|* 2| TABLE ACCESS FULL |SALES |21775 | 255K  |     312 (7) | 00:00:04 |
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR
              "STATE_CODE"='MI')

Statistics
----------------------------------------------------------
         0 recursive calls
         0 db block gets
      1071 consistent gets
         0 physical reads
         0 redo size
      1737 bytes sent via SQL*Net to client
```

```
       453  bytes received via SQL*Net from client
         5  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
        55  rows processed
```

从上面的跟踪清单中可以看到，当对 Sales 表执行全表扫描 sales 表时，查询执行了 1 071 个连续的获取操作。查询的总时间是 0.45 秒。对于一个包含 225 000 条记录的表，这一查询速度似乎很快；但是，如果数据库的用户必须反复执行此查询，则所需时间肯定会增加到很多秒。

使用 SQL Result Cache 的最简单的方法是在 SQL 语句中包含 RESULT_CACHE 提示，如下面的程序清单所示。第一次运行此查询时，它必须访问所有与前面的查询相同的数据块，但现在查询结果存储在 SGA 中，以供其他人使用。

```
SQL> select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST  PROD_CODE    SUM(AMT)
-- ---------- ----------
IL      100860      16520
IL      100861     189180
...
MI      200380        208
55 rows selected.
Elapsed: 00:00:00.45
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
-------------------------------------------------------------------------------
|Id | Operation        | Name                        |Rows  |Bytes |Cost
(%CPU)|Time |
-------------------------------------------------------------------------------
---------------
| 0 | SELECT STATEMENT |                             |   66 |   792| 317
(9)| 00:00:04 |
| 1 | RESULT CACHE     |gx61vnrf7aqs046gr4r3m70kqs|      |      |      |
|
| 2 | SORT GROUP BY    |                             |   66 |   792| 317
(9)| 00:00:04 |
|* 3| TABLE ACCESS FULL| SALES                       |21775 | 255K | 312
(7)| 00:00:04 |
-------------------------------------------------------------------------------
---------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR "STATE_CODE"='MI')
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=3; dependencies=(DEMO.SALES); parameters=(nls);
name="select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
where state_code in ('IL','IN','MI')
GROUP BY STATE_CODE,"
Statistics
------------------------------------------------------------
```

```
      1 recursive calls
      0 db block gets
   1071 consistent gets
      0 physical reads
      0 redo size
   1737 bytes sent via SQL*Net to client
    453 bytes received via SQL*Net from client
      5 SQL*Net roundtrips to/from client
      1 sorts (memory)
      0 sorts (disk)
     55 rows processed
```

连续获取操作的数量(1 071)和此查询的执行时间(0.45 秒)与第一个查询完全相同。需要注意的是，虽然两个执行计划的 Plan Hash 值(3229864837)是相同的，但执行计划是不同的。在第二个执行计划中另外有一行将此操作列为 RESULT CACHE，并给出了一个十六进制的名称。因为这是第一次执行带有 RESULT_CACHE 提示的这种语句，所以 Oracle 将结果存储在数据库实例的 SQL Result Cache 中。在此数据库实例上执行相同 SQL 语句的任何人都可以利用缓存中的这一查询结果。如果是 RAC 数据库，则每个实例有自己的 SQL Result Cache；如果前面的查询(包含提示)在一个实例上执行，则结果只在此实例可用。

现在再次执行此查询，以了解它是否利用了 SQL Result Cache：

```
SQL> select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST  PROD_CODE    SUM(AMT)
-- ---------- ----------
IL      100860      16520
IL      100861     189180
...
MI      200380        208
55 rows selected.
Elapsed: 00:00:00.12
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
-------------------------------------------------------------------------------
| Id  | Operation         | Name                        | Rows |Bytes | Cost
(%CPU)| Time |
-------------------------------------------------------------------------------
----------------
| 0   | SELECT STATEMENT  |                             |   66 | 792  | 317
(9)| 00:00:04 |
|  1  | RESULT CACHE      | gx61vnrf7aqs046gr4r3m70kqs |      |      |     |
|
|  2  | SORT GROUP BY     |                             | 66   | 792  | 317
(9)| 00:00:04 |
|* 3  | TABLE ACCESS FULL | SALES                       |21775 | 255K | 312
(7)| 00:00:04 |
-------------------------------------------------------------------------------
----------------
Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   3 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR "STATE_CODE"='MI')
Result Cache Information (identified by operation id):
------------------------------------------------------

   1 - column-count=3; dependencies=(DEMO.SALES); parameters=(nls);
name="select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
where state_code in ('IL','IN','MI') GROUP BY STATE_CODE,"
Statistics
----------------------------------------------------------

         0 recursive calls
         0 db block gets
         0 consistent gets
         0 physical reads
         0 redo size
      1737 bytes sent via SQL*Net to client
       453 bytes received via SQL*Net from client
         5 SQL*Net roundtrips to/from client
         0 sorts (memory)
         0 sorts (disk)
        55 rows processed
```

从上面的跟踪清单中可以看到，执行计划与将数据加载到 SQL Result Cache 中的查询完全相同，计划散列值与前两个查询完全相同。不同的是执行时间。查询的运行时间不再是 0.45 秒，而是 0.12 秒。从不同的会话多次运行查询，每种情况的运行时间都是 0.12 秒，连续获取(逻辑读)始终是零。

如果表被更新，会怎么样呢？Oracle 会继续使用 SQL Result Cache 中存储的查询的结果，直到提交变更。

为测试这一点，在第二个 SQL*Plus 会话中更新 sales 表：

```
SQL> rem now do an update in another session;
SQL> update sales set amt =amt + 1 where state_code = 'IL'
and prod_code = 100860;
```

现在在第一个会话中再次运行此查询，注意输出是相同的，因为变更还没有提交：

```
SQL> select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST    PROD_CODE    SUM(AMT)
--  ---------- ----------
IL      100860       16520
IL      100861      189180
...
MI      200380         208
55 rows selected.
Elapsed: 00:00:00.15
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
-------------------------------------------------------------------------------
|Id | Operation         | Name                      |Rows | Bytes | Cost
(%CPU)| Time |
-------------------------------------------------------------------------------
----------------
```

```
| 0 | SELECT STATEMENT |                                | 66 | 792  | 317
(9)| 00:00:04 |
| 1 | RESULT CACHE     |gx61vnrf7aqs046gr4r3m70kqs|     |      |     |
|
| 2 |SORT GROUP BY     |                                | 66 |  792 | 317
(9)| 00:00:04 |
|* 3 |TABLE ACCESS FULL | SALES                        |21775 | 255K| 312
(7)| 00:00:04 |
--------------------------------------------------------------------------------
---------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR "STATE_CODE"='MI')
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=3; dependencies=(DEMO.SALES); parameters=(nls);
name="select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
where state_code in ('IL','IN','MI') GROUP BY STATE_CODE,"
Statistics
----------------------------------------------------------
       0 recursive calls
       0 db block gets
       0 consistent gets
       0 physical reads
       0 redo size
    1737 bytes sent via SQL*Net to client
     453 bytes received via SQL*Net from client
       5 SQL*Net roundtrips to/from client
       0 sorts (memory)
       0 sorts (disk)
      55 rows processed
```

从上面的跟踪清单中可以看到，SQL Result Cache 按照预期被使用，查询输出提供读一致性结果，这在我们的预料之中，因为第二个会话还没有提交变更。

现在我们看一下在第二个会话中提交变更之后，当在第一个会话中运行查询时会怎么样：

```
SQL> rem now commit in second session
SQL> commit;
Commit complete
SQL> select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST   PROD_CODE     SUM(AMT)
-- ---------- ----------
IL      100860      16672
IL      100861     189180
...
MI      200380        208
55 rows selected.
Elapsed: 00:00:00.55
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
--------------------------------------------------------------------------------
---------------
```

```
| Id |Operation          | Name                        | Rows  |Bytes | Cost
(%CPU)| Time |
-------------------------------------------------------------------------------
----------------
| 0 |SELECT STATEMENT   |                             | 66    | 792  | 317
(9)| 00:00:04 |
| 1 |RESULT CACHE       |gx61vnrf7aqs046gr4r3m70kqs|       |      |      |
| 2 |SORT GROUP BY      |                             | 66    | 792  | 317
(9)| 00:00:04 |
|* 3|TABLE ACCESS FULL | SALES                        |21775 |255K | 312
(7)| 00:00:04 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR "STATE_CODE"='MI')
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=3; dependencies=(DEMO.SALES); parameters=(nls);
name="select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
where state_code in ('IL','IN','MI') GROUP BY STATE_CODE,"
Statistics
----------------------------------------------------------
        0 recursive calls
        0 db block gets
     1071 consistent gets
        0 physical reads
        0 redo size
     1737 bytes sent via SQL*Net to client
      453 bytes received via SQL*Net from client
        5 SQL*Net roundtrips to/from client
        1 sorts (memory)
        0 sorts (disk)
       55 rows processed
```

Oracle 确定被查询的表已更新，且变更已提交。因此它强迫 SQL*Plus 会话重新查询数据，并将数据存放在结果集中。如我们所料，更新后的值现在显示在查询输出中。需要注意的是，连续读的次数回到第一次运行时的 1 071，因为必须再次访问所有数据块，所以时间从 0.12 秒增加到 0.55 秒。

再次运行查询，得到的结果相同，但连续读的次数是零，因为正在使用 SQL Result Cache，如下所示：

```
SQL> select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
  2 where state_code in ('IL','IN','MI')
  3 GROUP BY STATE_CODE, PROD_CODE ORDER BY 1,2;

ST  PROD_CODE    SUM(AMT)
-- ---------- ----------
IL      100860      16672
IL      100861     189180
...
MI      200380        208
55 rows selected.

Elapsed: 00:00:01.21
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3229864837
-------------------------------------------------------------------------------
---------------
|Id  | Operation         | Name                          | Rows |Bytes | Cost
(%CPU)| Time |
-------------------------------------------------------------------------------
| 0 |SELECT STATEMENT  |                                |   66 | 792  | 317
(9)| 00:00:04 |
| 1 |RESULT CACHE      |gx61vnrf7aqs046gr4r3m70kqs|      |      |      |
|
| 2 |SORT GROUP BY     |                                |   66 | 792  | 317
(9)| 00:00:04 |
|* 3 |TABLE ACCESS FULL |SALES                          |21775 | 255K | 312
(7)| 00:00:04 |
-------------------------------------------------------------------------------
--------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("STATE_CODE"='IL' OR "STATE_CODE"='IN' OR "STATE_CODE"='MI')
Result Cache Information (identified by operation id):
------------------------------------------------------
  1 - column-count=3; dependencies=(DEMO.SALES); parameters=(nls);
name="select /*+ result_cache */ state_code, prod_code, sum(amt) from sales
where state_code in ('IL','IN','MI') GROUP BY STATE_CODE,"
Statistics
----------------------------------------------------------
       0 recursive calls
       0 db block gets
       0 consistent gets
       0 physical reads
       0 physical reads
       0 redo size
    1737 bytes sent via SQL*Net to client
     453 bytes received via SQL*Net from client
       5 SQL*Net roundtrips to/from client
       0 sorts (memory)
       0 sorts (disk)
      55 rows processed
```

47.1 SQL 结果缓存的数据库参数设置

SQL 结果缓存(SQL Result Cache)是共享池区域(Shared Pool Area)的一部分，Oracle 提供了很多与此功能相关的参数。

RESULT_CACHE_MODE 参数可用来在数据库级别打开或关闭 SQL Result Cache。此参数可设置为下面的值：

- AUTO　Oracle 优化程序根据重复使用情况确定哪些结果集应该存储在缓存中。
- Manual　RESULT_CACHE 提示必须用在 SQL 语句中，查询结果集才存储在缓存中。这是默认值。

- Force　所有查询的所有结果都存储在缓存中。

在数据库初始化 RESULT_CACHE_MODE 参数的设置中，优先使用 RESULT_CACHE 或 NORESULT_CACHE 提示。

RESULT_CACHE_MAX_SIZE 参数用来确定共享池区域中 SQL 结果缓存的大小。下面是要牢记的一些要点：

- 如果此参数设置为 0，则禁用 SQL 结果缓存。
- 此参数不能大于共享池区域大小的 75%。
- 默认值取决于其他内存设置，默认值与特定平台有关，默认值是从 MEMORY_TARGET、SGA TARGET 和 SHARED_POOL_SIZE 参数导出的。

RESULT_CACHE_MAX_RESULT 参数用来确定 SQL 结果缓存中最大的结果集。默认值是 RESULT_CACHE_MAX_RESULT 参数指定的大小的 5%。

RESULT_CACHE_REMOTE_EXPIRATION 参数用来指定使用远程对象的结果集在下一个执行相应 SQL 语句的用户重新查询之前，此结果集有效的分钟数。下面是要牢记的一些要点：

- RESULT_CACHE_REMOTE_EXPIRATION 参数设置为 0 表明使用远程数据库对象的结果集不应该缓存。
- 如果远程表已经变化，则此参数设置为非 0 值会产生不正确的结果。
- 默认值是 0。

需要牢记的是，SQL 结果缓存是共享池的一部分，如果在参数文件中设置 SHARED_POOL_SIZE，则当从 Oracle 11g 以前的版本迁移到 Oracle 11g 版本时，您可能想要增加共享池的大小。

47.2　DBMS_RESULT_CACHE 程序包

DBMS_RESULT_CACHE 程序包包含很多过程和函数，这些过程和函数可用来监控和管理共享池中的 SQL 结果缓存。

DBMS_RESULT_CACHE.MEMORY REPORT 过程生成详细的 SQL 结果缓存的内存使用情况报告。

```
SQL> set serveroutput on
SQL> exec dbms_result_cache.memory_report;
R e s u l t C a c h e M e m o r y R e p o r t
  [Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 1568K bytes (1568 blocks)
Maximum Result Size = 78K bytes (78 blocks)
  [Memory]
Total Memory = 100300 bytes [0.057% of the Shared Pool]
... Fixed Memory = 5132 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 95168 bytes [0.054% of the Shared Pool]
....... Overhead = 62400 bytes
....... Cache Memory = 32K bytes (32 blocks)
```

```
..... Unused Memory = 27 blocks
..... Used Memory = 5 blocks
..... Dependencies = 1 blocks (1 count)
..... Results = 4 blocks
..... SQL = 2 blocks (1 count)
..... Invalid = 2 blocks (1 count)
PL/SQL procedure successfully completed.
```

DBMS_RESULT_CACHE.FLUSH 过程可用来刷新结果缓存使用的内存。您可以包含要保持结果缓存当前大小和要保持当前统计信息的参数。这两个参数的默认值都是 false。下面的示例展示了刷新过程执行之后动态内存使用率是 0：

```
SQL> execute dbms_result_cache.flush;
PL/SQL procedure successfully completed.
SQL> exec dbms_result_cache.memory_report;

R e s u l t C a c h e M e m o r y R e p o r t
  [Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 1568K bytes (1568 blocks)
Maximum Result Size = 78K bytes (78 blocks)
  [Memory]
Total Memory = 5132 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5132 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
PL/SQL procedure successfully completed.
```

DBMS_RESULT_CACHE.STATUS 函数可用来确定结果缓存的状态：

```
SQL> select dbms_result_cache.status from dual;

STATUS
--------------------------------------------------------------------------------
ENABLED
```

DBMS_RESULT_CACHE.INVALIDATE 函数可用来使依赖于特定表的查询无效：

```
SQL> execute dbms_result_cache.invalidate ('DEMO','SALES');
```

47.3 SQL 结果缓存的字典视图

下面几个数据字典视图提供了 SQL 结果缓存的详细信息：

- V$RESULT_CACHE_DEPENDENCY 显示了缓存的结果与其依赖项之间的依赖关系。
- V$RESULT_CACHE_MEMORY 显示了所有内存块及其状态。
- V$RESULT_CACHE_OBJECTS 显示了所有对象(缓存的结果及其依赖项)及其属性。
- V$RESULT_CACHE_STATISTICS 显示了各种结果缓存设置和使用情况统计信息。

47.4　SQL 结果缓存的更多细节

SQL 结果缓存也可用于下面这些方面：

- 子查询和内联查询，即使外部查询发生了变化
- 向主调语句返回值的函数(使用 RESULT CACHE 关键字，并可以有选择性地使用“relies on”子句指定任何依赖性，列出应该使结果集无效的基本表)
- 闪回查询(它们也可缓存)

SQL 结果缓存对下面对象自动禁用：

- 临时表和字典表
- CURRVAL 和 NEXTVAL 序列
- 包含 CURRENT_DATE 或 SYSDATE 的查询

内存不会从结果缓存自动释放。可以使用 DBMS_RESULT_CACHE.INVALIDATE 释放依赖于特定表的 SQL 语句所使用的内存。也可以使用 DBMS_RESULT_CACHE.FLUSH 过程释放整个结果缓存。

47.5　Oracle 调用接口(OCI)客户端查询缓存

除 SQL 结果缓存以外，Oracle 还实现了客户端查询缓存，它们可用来为使用 OCI 调用的应用程序缓存客户系统内存中的数据。经常执行的查询集的查询结果缓存在本地，因此不需要往返于数据库来检索经常使用的应用程序数据。从而消除了数据库服务器上的 CPU 和进程负载以及相应的网络流量。

客户查询缓存最适用于应该程序使用的静态结果集和经常执行的查询。其好处是增强了查找表的性能和具有将服务器端查询缓存扩展到客户端的功能。如果数据库中的结果缓存发生变化，则客户端查询缓存自动刷新。

为了利用客户端缓存，Oracle DBA 必须使用下面的参数在数据库的 SGA 中设置另外一个缓存：

- CLIENT_RESULT_CACHE_SIZE　所有 OCI 客户端进程获取最大的缓存。它可以被 sqlnet.ora 配置参数 OCI_RESULT_CACHE_MAX_SIZE 所重写。此参数的默认值是 0，这是一个静态参数。
- CLIENT_RESULT_CACHE_LAG　指定自从上一次往返于服务器以来的最长时间(单位是毫秒)，OCI 客户端查询需要在此时间之前往返于服务器，以获取与客户端缓存的查询相关的任何数据库变化。此参数的默认值是 5 000 毫秒，它不是一个动态参数。

47.6 Oracle 调用接口(OCI)客户端查询缓存的限制

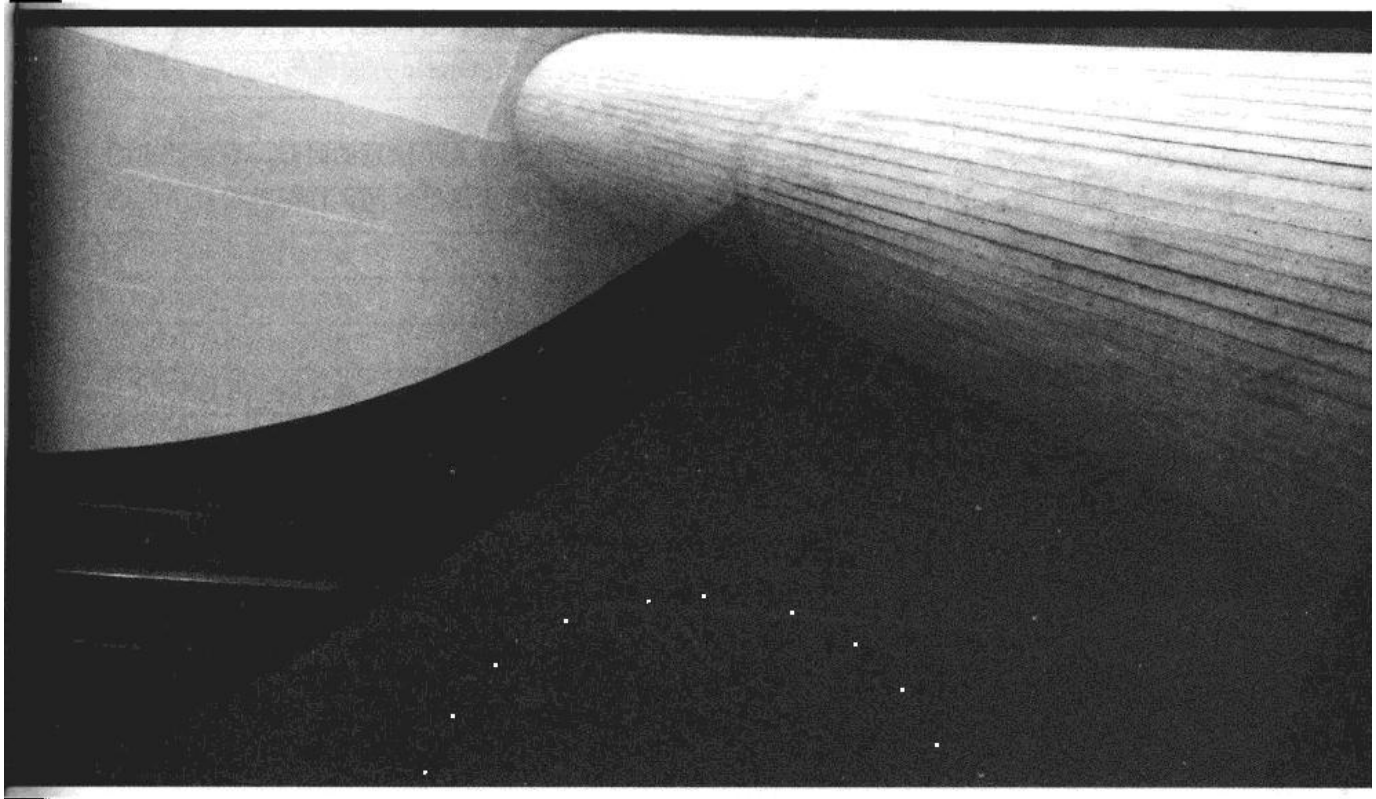
用 OCI 编写的查询必须包含结果缓存提示，以表明结果存储在查询结果缓存中。应用程序必须重新与 Oracle 11g 连接，客户端必须连接到 Oracle 11g 或更高版本的数据库。

客户端查询缓存最适用于数据库对象是只读或大部分读的情况。此外，如果结果缓存提示是针对返回大型结果集的查询，则将使用大量的客户结果缓存。

下面的参数可用在客户配置文件中或在 sqlnet.ora 文件中，以重写数据库参数文件中设置的缓存参数：

- OCI_RESULT_CACHE_MAX_SIZE 为此客户指定 CLIENT_RESULT_CACHE_SIZE 的最大大小
- OCI_RESULT_CACHE_MAX_RSET_SIZE 指定单个结果集的最大大小，单位是字节
- OCI_RESULT_CACHE_MAX_RSET_ROWS 指定单个结果的最大行数

可以查询V$CLIENT_RESULT_CACHE_STATISTICS视图来显示各种结果缓存设置和使用情况统计信息。



第 48 章

关于调整的示例分析

正如第 46 章所述，性能调整从设计开始。在很多情况下，数据库应用程序的核心性能问题已经设计到系统中。必须能够识别应用程序资源最密集的部分，把调整工作集中到特定的数据库参数或 SQL 语句上。本章将描述与不同性能问题相关的示例分析及其解决方案。示例分析从应用程序调整人员的角度介绍了 3 个通用问题区域(环境问题、问题查询以及应用程序体系结构)。

48.1 示例分析 1：等待、等待、再等待

随着并发用户数量的增加，大型在线事务处理系统遇到了显著的性能问题。有时在密集的批处理活动(例如，月末的关闭进程)中，系统性能对在线用户来说不能令人满意。问题是

环境、应用程序，还是在环境中应用程序的实现呢？为了确定核心问题并开发能够提高性能的路径，必须重新检查数据库的统计信息。系统很复杂，但是 Oracle 方面的核心问题与第 46 章所描述的问题类似——要求数据库执行大量的低速命令。

调整数据库的第 1 步包括在应用程序的性能令人满意和令人不满意的时候，查看动态的性能统计信息。可以使用 AWR 实用程序收集统计信息。对于这些区间，详细地检查下面的统计信息：

- 每秒执行的次数
- 每秒逻辑读的次数
- 每物理读的等待次数
- 数据写期间的等待次数
- 撤消段头和块的等待次数
- 每秒物理读的次数
- 在磁盘和内存中执行的排序的次数

捕获这些统计信息，无论是对于识别出现的问题，还是对于测量在调整过程中数据库环境和应用程序 SQL 的改变对于性能的影响，都很重要。每个统计信息(或者其组件)都通过 AWR(及其以前的 Statspack)报告。可以选择数据库的统计信息，因为它最后从动态性能视图(如 V$SYSSTAT 和 V$WAITSTAT)启动，但是这些统计信息将由于数据库中与该应用程序不相关的其他活动而产生偏差。表 48-1 描述了在 3 个小时的测试中收集的初始统计信息。结果显示如下：

表 48-1 在 3 个小时的测试中收集的初始统计信息

统 计 信 息	值
测试持续时间	10 800 秒(3 小时)
每秒执行的次数	1 700
每秒逻辑读的次数	67 000
每秒物理读的次数	6 300
缓冲区忙等待次数	600 000
撤消段头等待次数	300
内存中的排序次数	800 000
磁盘中的排序次数	200
物理读等待次数：Db 文件顺序读(读 1 块)	32 700 000
Db 文件离散读(读多块)	4 500 000
每秒处理的事务	10
每秒的物理写	100

已知这些信息，应该最先关注哪个区域呢？系统非常活跃——每秒执行 1 700 次和每秒 67000 个逻辑读表明在 3 个小时内持续执行大量数据库请求。为了正确分析这些统计信息，应该首先把它们转换成一致性的单位。在本示例中，把所有活动用每秒表示。把原来在区间级表示的统计信息除以 10 800，查看它们每秒的值。结果如表 48-2 所示。

表 48-2　每秒钟的统计信息

统 计 信 息	值(每秒)
执行次数	1 700
逻辑读次数	67 000
物理读次数	6 300
缓冲区忙等待次数	55
撤消段头等待次数	0.03
内存中的排序次数	74
磁盘中的排序次数	0.02
物理读等待：Db 文件顺序读(读 1 块)	3 028
Db 文件离散读(读多块)	416
事务处理	10
物理写次数	100

当所有统计信息用同一单位表示时，就可以对数据库中平均每秒发生的情况有个大致的了解。具有代表性的每秒中出现的物理读等待总次数是 3 028＋416=3 444。当数据库试图解析物理读的请求时引起这些等待——在具有代表性的每秒中，有 6 300 个这样的请求。因此，物理读事件不得不平均等待大约 55%的时间(3 444 / 6 300)。不需要等待时，只能完成 45%的读请求。

同时，写也在等待。当数据库试图执行 100 次物理写时，每秒有 55 个数据块等待——同样是 55%的等待率。查看数据文件 I/O 统计信息(通过 Statspack、AWR 报告和对 V$FILESTAT 的查询提供)，可以看到数据库 I/O 均匀分布在最活跃的表空间上。如果不平均分配 I/O，那么单个磁盘或磁盘组遇到的问题将可能是 I/O 等待的主要来源。

如何降低物理等待的次数？如果仅仅考虑执行全表扫描的查询是导致性能问题的可能原因，那么将很可能几乎不影响该应用程序。全表扫描导致离散读等待(如表 48-2 所示)——它们仅占所发生物理等待的 12%(416 / 3 444)。如果完全删除全表扫描产生的所有等待，则对于应用程序的性能也起不到显著的影响。

注意：

如果全表扫描通过并行查询来执行，则它们始终从磁盘而不是从内存读取。

基于这些原因，可以提出下面这些假设：

- 应用程序需求超过系统配置的 I/O 容量。
- 应用程序导致了大量 SQL 命令的执行。
- 执行的大多数 SQL 命令不产生物理读。数据块缓冲器缓存占有率超过 90%(在这种情况下，是 1 - 6 300 / 67 000)
- 应用程序也消耗大量 CPU 资源，因为它执行很多命令，并很主动地管理 I/O 队列。

既然给出了这些假设，那么调整工作应该从何处开始？应用程序配置文件是读操作密集的应用程序(每秒有 6 300 次物理读，而只有 100 次物理写)。查询本身似乎被很好地调整(通过总的占有率判断；有可能存在个别查询没有正确调整)。读和等待报告了同样的问题——它

们不能足够快地访问磁盘以避免等待。因此，首先应该关注于稳定 I/O 环境。它是否执行得很好？如果在 I/O 系统中没有发现问题且不能修改应用程序 SQL，则需要更新或增强环境以支持应用程序产生的负载。

应该评估普通 I/O 环境的各个方面，包括控制器、网络接口卡、磁盘阵列和网络(共享的网络存储器)以便了解它们对性能的影响。在很多系统中，I/O 分布在很多磁盘上，用来访问磁盘的这些常见元素对性能的影响比磁盘本身的影响要大。

在大多数现代企业级配置中，缓冲区域对于磁盘环境是可用的。因此，大多数物理读请求不到达磁盘，缓冲区为读请求提供服务。在本分析中，大约 10%的逻辑读产生物理读请求。I/O 环境本打算通过其缓冲区处理 90%的物理读请求。由于没有正确配置该环境的缓冲区以支持该活动量(由存储管理人员确定)，因此大多数物理读请求到达磁盘。提高缓冲区的使用，就能提高环境中的所有 I/O，并避免很多磁盘访问。

随着 I/O 的改善，应用程序对 CPU 的需求也降低。由于系统不需要花大量时间管理 I/O 队列，所以 CPU 需求变低——这些资源现在对于用户进程是可用的。系统上其他某些进程可能现在可以自由获取更多的可用资源，说明即使完成调整工作之后，也需要继续监控系统。

为了进一步降低系统负载，把以批处理形式执行的操作从应用程序的在线处理部分中删除。为了进行比较，表 48-3 显示了 I/O 环境调整和应用程序修改之后的统计信息：

表 48-3 I/O 环境调整和应用程序修改之后的统计信息

统 计 信 息	值(每秒)
逻辑读次数	*67 000*
物理读次数	3 830(调整前是 6 300)
缓冲区忙等待次数	25(调整前是 55)
物理读等待次数：Db 文件顺序读(读 1 块)	466(调整前是 3 028)
Db 文件离散读(读多块)	173(调整前是 416)
事务处理	10
物理写	115

在修改后的环境中，物理读的次数降低了 40%，而等待的次数降低了 80%以上。数据库能够支持同样次数的逻辑读、事务处理以及物理写(在写时需要较少的等待)——现在最终用户性能令人满意。批处理仍然需要关注，虽然重新调度编写糟糕的代码可能减少在线用户的争用，但稍后仍然需要完成该工作。

事情还没有结束——仍然有等待发生，数据库随时间不断变大，用户将开发新的查询，应用于程序模块随着时间不断将添加。在这些修改中，应该随着时间的流逝，应该跟踪关键的统计信息，这样才能不受环境变化的影响、识别问题区域，并预测未来的性能。

48.2 示例分析 2：破坏应用程序的查询

差不多每个数据库最终都将支持消耗更多资源(与它应该消耗的资源相比)的查询。该查询可能执行多个全表扫描(如果索引丢失)。该查询可能产生笛卡尔(Cartesian)乘积(当在 from 子

句中列出多个表，但没有指定它们的连接语法时)。查询执行上百万行的多个嵌套排序。

通常，当调整 SQL 时，应该使用尽量少的行。如果仅仅需要一个千万行的表中的 10 行，那么设法使用索引返回符合 where 子句条件的尽量少的行，然后从这些行中过滤，得到想要的 10 行。

虽然该方法好像很简单，但是却经常反其道而行之。开发应用程序时，应当考虑编写的每条 SQL 语句如何执行——它的频率、每次执行的性能要求、对其他命令的依赖性，以及将使用的索引等等。目标是尽量更有效地支持业务处理。如果没有考虑好如何使用这些代码，则应用程序可能受到损害，应用程序的速度可能使得业务处理的速度无法接受。

在大型在线处理的应用程序中，下面格式的查询是执行频率最高的命令(基于 V$SQL 的 Execution 列中的统计信息)。因为该查询执行了全表扫描，所以 V$SQL 项上物理读(Disk_Reads 列)的累计次数接近逻辑读(Buffer_Gets 列)的累计次数：

```
select T35.C1
  from T35
 where (lower(T35.C4) like '%tom clark%');
```

如果用在产品化的应用程序上，则该查询至少存在 3 个错误。如果运行该查询一次，则性能可能令人满意——将执行表 T35 的全表扫描(原因列在下一段中)。但是当很多用户并发执行相似的查询，且表 T35 包含上百万行时，会发生什么呢？如果通过 Web 接口使这种类型的查询对所有用户，那么怎么样呢？应用程序性能受到损害，并对应用程序应该支持的进程造成不利的影响。

该查询的问题包括下面几方面：

- 没有使用绑定变量　除非数据库管理员将 CURSOR_SHARING 初始化参数设置为 FORCE 或 SIMILAR(首选的设置)，否则将分别解析查询的每次执行。如果 CURSOR_SHARING 设置为 EXACT，则共享的 SQL 区域将充满上千条几乎一样的命令，这些命令被分别解析，仅仅在作为搜索字符串传递的字面值上不同。
- 在 C4 列上没有定义基于函数的索引，但是查询执行该列上的 LOWER 函数。如果仅标准索引可用，那么 Oracle 不能使用该索引支持这个查询。
- 搜索字符串以%通配符开始。这样的结果是，为了查找符合条件的值，要求 Oracle 执行全表扫描。

注意：
应当生成用于调整的每个查询的说明计划。

考虑第 3 个问题可能的解决方案——搜索的字符串的内容。在该应用程序中，用户输入姓名(在此示例中是“tom clark”)，应用程序代码把通配符放在名字的前面和后面，试图使能找到的潜在匹配数量最大化。应用程序开发人员把所有可能与搜索的字符精确串匹配的潜在值都呈现给用户。

从数据库管理角度看，可以确保正确配置 CURSOR_SHARING，可以在 C4 列上创建基于函数的索引，这样可以选择索引扫描代替表扫描。但是核心问题仍然保持不变——如何重

写查询，才能选择尽量少的行(为了生成其结果集)?

答案在于 Oracle 文本的使用。如第 27 章所描述，可以在列上创建文本索引来支持字符串、通配符、文本扩展等的搜索。在此示例中，把名称作为搜索的字符串传递，文本索引将搜索其匹配的索引的值——考虑通配符。应用程序开发人员将修改查询，为了使用 Oracle 文本语法；同时数据库管理员必须创建和维护文本索引对象。但是，无论如何，性能优点是显著的——消除了大型表上的全文扫描。在此示例中，对于频繁执行的查询的好处与维护文本索引的系统开销相抵消。

注意：
如果使用索引去索引姓名，就必须创建空的禁用列表(stoplist)。否则，Oracle 将忽略通用词(如“AN”)，而它可能是真实的姓名。

在这个特殊的示例中，对此查询使用文本索引极大地减少了基于 Web 的用户的搜索时间。搜索表执行的物理读和逻辑读的次数显著减少，且当应用程序增大时，继续保持很少。获得性能好处的代价是，数据库管理员和应用程序维护小组需要管理更多文本索引对象。

文本索引不能解决每个问题。在大多数情况下，包含多列的标准索引才是解决方案。重要步骤是识别资源最密集的查询(通过查询 V$SQL 或使用 Statspack/AWR)，从应用程序体系结构的视角评价它们。对每个查询，询问其用于什么目的、为什么这样做，以及正在做什么。考虑下面这个简单的查询：

```
select COUNT(*) from COUNTRIES;
```

个别情况下，每次执行查词很快完成——表中仅有几百个国家，表可以缓存到数据块缓冲区缓存中。但是如果每天执行数百万次这样的查询，它将比数据库中其他任意查询执行更多的逻辑读操作。这些逻辑读转化为分配给静态数据重复查询的 CPU 资源(这些 CPU 资源可以用于更有用的动态数据的查询)。查找出现问题的查询时，一定要检查执行很多次的查询，以及每次执行消耗很多资源的查询。

使用 10053 跟踪事件

一个查询在每个环境中可能不使用相同的说明计划。当解析查询时，Oracle 检查查询可能的执行方式，并指定每一种可选方案的成本。成本不是随意指定的，Oracle 考虑表的统计信息(例如，行数)、索引(包括其选择性)和数据库参数值。

Oracle 提供一个跟踪事件，它可用来查看在评价可选路径时优化程序执行的计算。该跟踪事件的输出是写到数据库服务器上 USER_DUMP_DEST 目录中(在启动参数文件中定义)的一个跟踪文件。

在会话中，可以通过下面的命令启用跟踪功能：

```
alter session set
events '10053 trace name context forever, level 1';
```

打开会话的跟踪功能后，现在应该运行想要分析的 SQL 了。当 SQL 命令完成时，可以通过下面的命令关闭跟踪功能：

```
alter session set
    events '10053 trace name context off';
```

跟踪信息只在第一次解析命令时生成。如果命令已经在实例中解析，则跟踪文件将显示执行的命令，但不显示优化程序执行的计算。

对于不同版本的 Oracle，跟踪文件输出的各部分的顺序有所不同，但一般会包括下面这些内容：

- 数据库的初始化参数
- 对象级统计信息
- 考虑的表访问路径
- 考虑的连接顺序和操作选项
- 选择的路径

参数设置和对象级统计信息对于解释何时评估查询的执行路径是很重要的。运行在不同数据库的同一个查询可能遵循不同的执行路径，这会导致应用程序性能有很大的差别。此性能差别常常是由于数据库统计信息的差异而导致的。用户可以使用这一跟踪事件来查看统计信息如何影响选择的路径。

对于查询所涉及的每个表，跟踪文件将显示表的统计信息和索引的统计信息。统计信息和初始化参数的调整是一个数学问题。每个表执行全表扫描的成本是什么？每种连接(嵌套循环、排序合并和散列连接)的成本是什么？排序区域的大小是多大？查询需要使用临时段来存储的可能性有多大？对于每种连接，如果按照表的访问顺序来变换连接的顺序，结果会怎么样呢？

每一种排列都会被评估，并会为每种方法生成一个成本。然后对这些成本进行比较，并选择出成本最低的那种方法。

在数据库级或会话级改变参数值会影响成本的计算方式，从而影响到哪个连接路径被选中。例如，增加 DB_FILE_MULTIBLOCK_READ_COUNT 的设置会使全表扫描的成本减少，而减少 OPTIMIZER_INDEX_COST_ADJ 参数的设置会使索引访问的成本减少。

另外，改变对象的统计信息将影响优化程序计算期间使用的成本。如果您关心在未来的产品版本中优化程序的选择，则可以从产品数据库导出统计，并将它们导入到测试数据库。下一次解析命令时，将使用新的统计信息，并生成反映计算结果的说明计划。

在数据库生命周期中发生巨大变化时，可以使用 10053 跟踪事件来验证优化程序对关键查询的计算。例如，可以使用跟踪事件来确定下面这些变化是如何影响优化程序的选择的：

- 数据库版本中的变化
- 操作平台中的变化
- 初始化参数中的变化
- 对象统计信息中的变化
- 提供给查询的提示中的变化

在每个示例中，跟踪输出为优化程序正在执行的计算提供了基础，从而消除了查询调整中的猜测工作。

48.3 示例分析3：长期运行的批处理作业

某公司计划在重复的基础上，把大量数据从事务处理系统移到报表系统。由于事务处理系统不受其控制，因此没有机会使用数据库链接、物化视图或其他复制方法直接与源系统交互。相反，数据源是每天从事务处理系统发送过来的单个大型文件。

试图重复地把大型文件加载到报表系统中，将导致可用性问题——通过传统的插入操作加载数据所需要的时间超过了窗口操作的可用时间。加载过程中的任何故障都会导致更大的问题，因为部分加载的数据被删除并被重新加载。结果，加载作业的性能影响整个报表应用程序的可用性。根据当前和预期的数据量，在12个月内每天加载的数据量将增至3倍。

假设数据库配置正确，没有I/O或内存问题需要解决，下面几个方法可用来解决这类问题：

- 虽然源系统把数据作为一个集合提供，但不必一次性全部插入。不用每天把数据加载到报表系统一次，而是考虑整天发送增量式加载。如果加载过程不影响查询(例如，通过在加载中使索引无效或引起已报告的数据的不规则性)，这些小的加载就能够与用户查询一起运行。
- 不要加载数据；而是使用外部表。在少量定义良好的查询报告数据次数很少的情况下，此方法很理想。
- 并行化大型操作。使用服务器上所有可用的资源。尽管这可能有助于当前性能问题，但它可能影响应用程序随数据量增长的扩展能力。
- 把数据加载到临时工作表中，然后把它们视为现有表的部分。

每个选择都至少包含一个微小的设计修改，如果不知道最终用户如何使用数据，就不可能选择合适的解决方案进行修改。数据加载到单个表还是多个表中？最终用户是否直接访问数据所加载到的表？表中的数据如何失效？

首先考虑最坏的情况：用户一次性需要所有可用的数据，因此不能使用增量式加载。加载的数据插入多个相关表中。此外，加载过程应该对数据库中其余数据的可用性影响最小。最后，最终用户直接查询加载的表。

在这种情况下，应当考虑对加载的表进行分区。对每天的数据加载，创建新表(新表的结构与用户将要查询的表相同)。把数据加载到表中，对它进行索引、分析，并执行任何必要的转换(这些转换为用户提供格式正确的数据)。当数据准备显示给用户时，执行alter table exchange partition命令，告诉Oracle处理该表(该表作为用户将要查询的表的分区而创建)。因为当添加新分区时仅仅涉及应用程序的表，所以对应用程序的停机时间最小——特别是如果能够只使用分区表上的本地索引。

exchange partition方法有效地划分在该表上执行的两个操作——您对它的加载和用户对它的查询。一旦能够分离这两个操作，设计和维护能力将极大提高。

考虑稍微好一点的情况：用户一次性需要所有可用的数据，但是不直接查询加载的表。用户查询基于该表的物化视图。应当调整物化视图，使物化视图能够更快地更新，允许对所加载表的修改增量式地应用到用户将要查询的对象上。同样，把批量加载操作从相同对象的

查询中分离将显著增强维护选项。关于物化视图刷新选项配置的详细信息，请参阅第 26 章。

如果用户一直查询聚集数据的物化视图，则根本不应该考虑加载数据。取而代之的是，在 insert as select 操作时使用外部表(请参阅第 28 章)在汇总表中引用数据文件。外部表不能被索引，因此将导致全文件扫描——但由于不是加载原始数据，汇总表的 insert as select 可以更快地完成(如果加载数据然后对其进行聚集)。其他好处是，仅需一次性存储原始数据(在文件系统级)而不需要两次。

不论选择哪些选项，都应该调查开发可用系统资源的选项的用法— 包括并行查询、并行 DML、并行加载、分区，以及批处理。有很多功能可以用在提高加载过程自身的性能方面：在加载过程中使用 nologging，使用 SQL*Loader Direct Path 加载，启动并行数据泵(Data Pump)作业。注意：这些选项很多都可以应用于并列的分区级。

调整批处理加载需要基于经验对将要使用的集合处理的程度做出判断。可以配置应用程序使其一次处理一行。例如，可以向主表中插入一行，然后在向主表中插入第 2 行之前，把其所有子行插入其他表中，不断重复上述过程。随着行数的增加，基于行的方法的性能将降低。虽然基于行的方法保证了不同加载表之间的读一致性，但是如果在空闲时间执行作业(或使用 exchange partition 方法)，用户可能就觉察不到其益处。

比较而言，可以创建一次性处理大量行的集合操作。可能在某一时刻，集合中处理的行的数量将影响加载的性能，特别是在包含复杂的排序时。在这种情况下，需要把大型批处理划分成更小的步骤，处理更小的分组。例如，操作流如下：

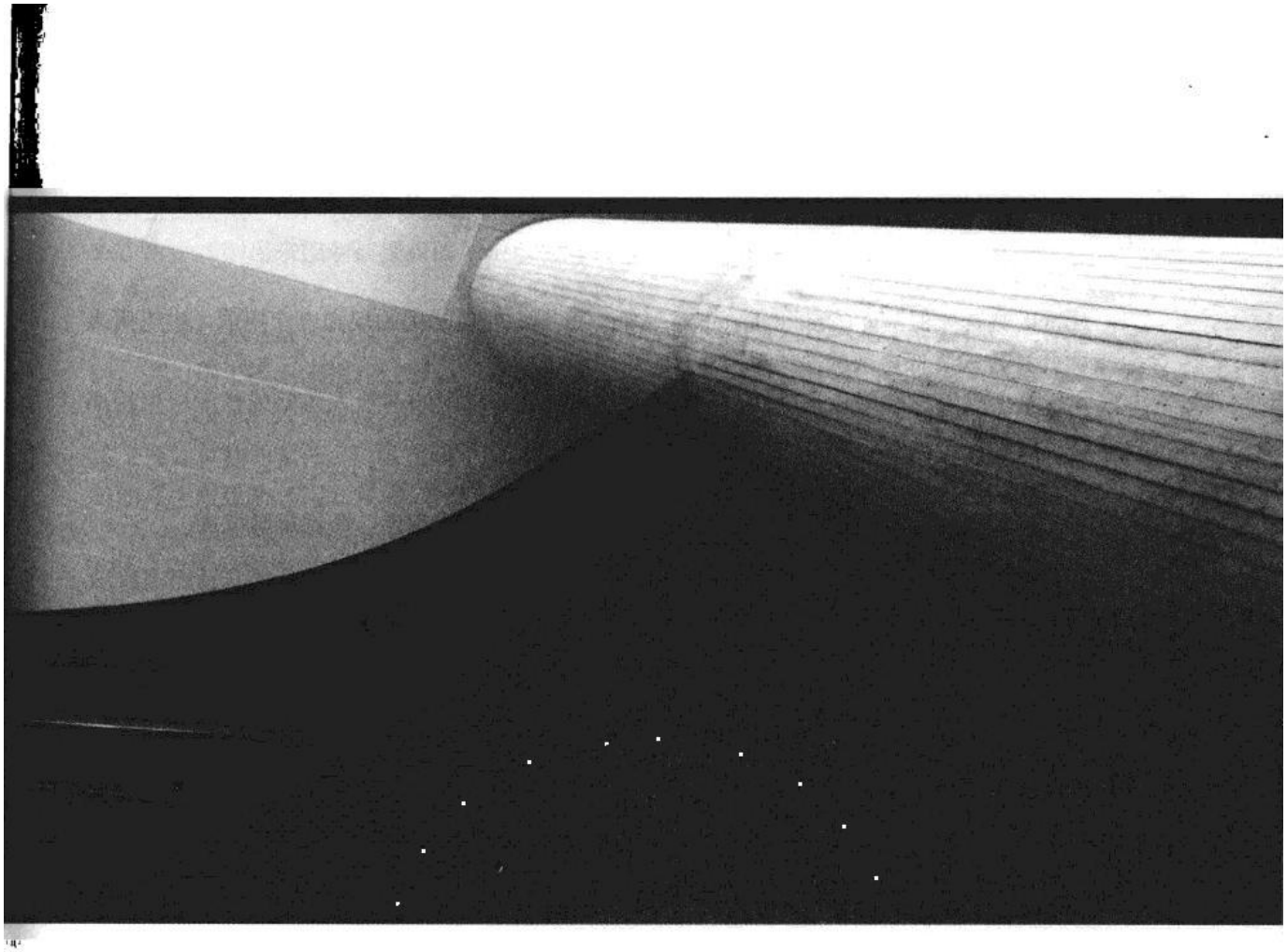
(1) 从输入表中选择 100 000 行。
(2) 在这些行上执行聚合和变换。
(3) 把聚合和变换的结果存储到一个临时工作表中。
(4) 从输入表中选择下 100 000 行。
(5) 重复上述过程直到所有行处理完并插入临时工作表中。
(6) 对临时工作表正确地索引和分析。
(7) 执行临时工作表中数据的其他操作(例如，生成总计数或新列，或执行与其他表的连接)。
(8) 从临时工作表中把行插入产品表中。
(9) 删除或截短临时工作表。

这样的操作序列试图获得两种方法的优势——作用于行集合的过程方法。每个行集合应该是多大？这取决于操作环境、所执行的操作，以及数据本身。应当适当地测试该处理方法，以确保多个过程方法比单个批处理操作完成得更快。应该考虑过程方法中大量潜在的故障点，从而相应地处理异常。

在本示例中，原来每天的加载作业需要 12 个小时完成。把它分解成更小的部分，全部时间减少到 3 个小时——如果需要，时间足够可以在加载窗口中多次运行该作业。

设计决策需要用户知道数据及其使用方式，以及如何影响业务处理。不管体系结构如何，必须能够在可重复的条件下模拟现实工作负荷测试应用程序。在测试过程中和产品使用过程中，随着数据量的增长，必须能够监控环境和应用程序的性能问题。

在调整过程中，必须做出权衡。例如，在批处理调整工作中，为了提高整体性能，可以给数据库分配更多磁盘空间(作为临时工作表)。应用程序的每一部分都应当作为整个应用程序的重要部分，相应地开发和计划。当应用程序的性能影响业务处理的性能时，应当识别出现问题的区域(如环境、特定的查询或数据流体系结构)，从而更好地满足应用程序的需求的。



第49章

高级体系结构选项——DB保险库、内容DB和记录DB

Oracle数据库的常见安全问题之一是：授权用户SYS和SYSTEM以及具有SYSDBA权限的其他账户对数据库中的所有数据具有完全的访问权。在过去，可以审计所有数据库用户的活动，但不能阻止授权用户查看和/或修改应用程序数据。Oracle已经发布了新产品来处理这一问题，此产品称为Oracle Database Vault(Oracle数据库保险库)。

49.1 Oracle 数据库保险库

Oracle 数据库保险库(Oracle Database Vault)可用来处理常用的规则遵守要求，也可用来以下面几种方式降低内部威胁的风险：

- 阻止权限高的用户、DBA 和具有 SYSADMIN 权限的其他人访问或修改应用程序数据。
- 通过将安全管理与数据库管理分开来实施职责分离。
- 控制何时、何地、以何种方式以及由何人来访问应用程序、数据和数据库。
- 提供了几种即装即用(out-of-the-box)的安全报告，展示了在数据库中访问的内容和访问时间

萨班斯法案(SOX)、欧盟隐私保护法和医疗保险信息交换和保密性法案(HIPAA)等政府法规都要求对敏感信息的访问、泄露和修改有很强的内部控制机制，因为对这些敏感信息的访问、泄露和修改会导致欺诈、身份盗窃、财务违规和财务犯罪。Oracle 数据库保险库的域、命令规则以及要素和职责分离功能都可以添加到已有的应用程序环境中，以满足各种政府法规，而无需更改已有的应用程序代码。

Oracle 数据库保险库可用于 Oracle Database 11g 版本 1、Oracle Database 10g 版本 2 和 Oracle Database 9i 版本 2。经过认证的 Oracle 数据库保险库可用于 Oracle PeopleSoft 应用程序和 Oracle 电子商务套件，其他的应用程序的认证正在进行中。

49.1.1 Oracle 数据库保险库的新概念

Oracle 数据库保险库引入了几个新概念：

- 域(Realm) 域是在安装了 Oracle 数据库保险库的数据库中，起“保护区”作用的容器。Oracle 数据库保险库的管理员可以创建域，并在域中定义内容。域可以由单个表、多个表、整个应用程序或多个应用程序组成。
- 命令规则(Command rule) 命令规则是安全管理员可以创建用来控制用户如何执行 SQL 语句的规则集合，这些 SQL 语句几乎涵盖了所有的 SQL 语句，包括 SELECT、ALTER SYSTEM、数据库定义语言(DDL)和数据库操纵语言(DML)语句。命令规则可以作用于规则集，以确定是否允许执行哪条语句。规则集使用一天中的时间、IP 地址、主机名或与用户相关联的任何几个可识别属性等因素。例如，如果用户满足命令规则的要求：对应用程序的访问只限于工作时间，且只能从内部 IP 地址访问(同时还包括其他任何配置参数)，那么用户只被授权访问某些数据。这些限制条件可应用于所有系统用户，包括权力最强大的 DBA 或 SYSDBA 角色。
- 多因素授权(Multifactor authorization) 多因素授权是指在决策过程中利用多个因素的规则集。安全管理员可以根据特定的遵守要求或安全要求来定义规则，例如，连接只限于某些特定的 IP 地址或某一范围的 IP 地址，且只限于工作时间等。

要控制权限很高的用户需要使用 Oracle 数据库保险库的域和命令规则。Oracle 数据库保险库在安装时创建了多个“即装即用”的安全域。Oracle 数据库保险库的域可以在很大程度

上阻止高权限的用户使用“select any”等权限来访问他们在其他地方无权访问的应用程序对象。命令规则用来控制被授权的用户以何种方式、在何时、在何地连接到数据库并执行 SQL 命令。

当评估命令规则的执行时，可以使用数据库保险库因素。这阻止了对数据库的即席式访问，并启用了多因素授权过程。通过将安全管理和数据库账户管理的权限与常规的 DBA 责任分开，Oracle 数据库保险库实施了职责分离。例如，数据库账户管理在 Oracle 数据库保险库中视为单独的责任。一旦安装了 Oracle 数据库保险库，就会自动阻止具有“create user”权限的高权限用户使用此权限，除非以后由Oracle数据库保险库将账户管理责任授予这些用户。

Oracle 和第三方应用程序应该在 Oracle 数据库保险库之前安装，因为 Oracle 数据库保险库对账户管理和安全管理实施职责分离。如果 Oracle 数据库保险库已启用，则 Oracle 和第三方应用程序使用的某些安装过程和打补丁过程会失败。相应的解决方案是，可以先禁用 Oracle 数据库保险库，等安装过程或打补丁过程完成之后，再重新启用 Oracle 数据库保险库。

49.1.2　禁用 Oracle 数据库保险库

执行以下操作步骤可以禁用 Oracle 数据库保险库：

(1) 关闭数据库。实时应用群集(RAC)数据库的所有实例必须关闭。如果 dbconsole 进程正在运行，则停止此进程。对于单实例安装和 Oracle 实时应用群集安装，使用如下命令：

```
$ emctl stop dbconsole
```

对于单实例安装，关闭数据库实例，如下所示：

```
$ sqlplus "sys / as sysoper"
Enter password: password
SQL> SHUTDOWN NORMAL
SQL> EXIT
```

对于 RAC 安装，关闭每个数据库实例，如下所示：

```
$ srvctl stop database -d db_name
```

(2) 重新链接 Oracle 可执行程序，关闭 Oracle 数据库保险库的选项：

```
$ cd $ORACLE_HOME/rdbms/lib
$ make -f ins_rdbms.mk dv_off
$ cd $ORACLE_HOME/bin
$ relink oracle
```

对于 RAC 安装，在所有节点上运行以上命令。

(3) 在 SQL*Plus 中，启动数据库。

对于单实例数据库安装，使用如下命令：

```
$ sqlplus "sys / as sysoper"
Enter password: password
SQL> STARTUP
SQL> EXIT
```

对于 RAC 安装，使用如下命令：

```
$ srvctl start database -d db_name
```

(4) 在命令提示符下，使用 dvca -action disable 选项来运行 Oracle 数据库保险库配置助手(Database Vault Configuration Assistant，DVCA)：

```
dvca -action disable
  -oh $ORACLE_HOME
  -service myservicename
  -instance myinstance
  -dbname mydbname
  -owner_account myownername
  -logfile dvcalog.txt
```

- -oh 是 Oracle 主目录
- -service 是数据库服务名
- -instance 是数据库实例名
- -dbname 是数据库名
- owner_account 是 Oracle 数据库保险库所有者的账户名

关于如何使用 DVCA 的更多信息，可参阅 *Oracle Database Vault Administrators Guide*。

现在 Oracle 数据库保险库已禁用，接下来可以重新启动数据库，或执行要求的任务，如打补丁或安装软件。

49.1.3 启用 Oracle 数据库保险库

执行如下操作步骤可以在 UNIX 操作系统上启用 Oracle 数据库保险库：

(1) 在命令提示符下，使用 DVCA 重新启用 Oracle 数据库保险库，如下所示：

```
dvca -action enable
  -oh $ORACLE_HOME
  -service myservicename
  -instance myinstance
  -dbname mydbname
  -owner_account myownername
  -logfile dvcalog.txt
```

(2) 关闭软件过程。确保环境变量 ORACLE_HOME、ORACLE_SID 和 PATH 正确设置。

(3) 如果 dbconsole 进程正在运行，则停止此进程。对于单实例安装和 RAC 安装，使用如下命令：

```
$ emctl stop dbconsole
```

(4) 关闭数据库实例。

对于单实例安装，使用如下命令：

```
$ sqlplus "sys / as sysoper"
Enter password: password
SQL> SHUTDOWN NORMAL
SQL> EXIT
```

对于 RAC 安装，使用如下命令：

```
$ srvctl stop database -d db_name
```

(5) 重新链接 Oracle 可执行程序，打开 Oracle 数据库保险库的选项：

```
$ cd $ORACLE_HOME/rdbms/lib
$ make -f ins_rdbms.mk dv_on
$ cd $ORACLE_HOME/bin
$ relink oracle
```

对于 RAC 安装，在所有节点上运行这些命令。

(6) 在 SQL*Plus 中，启动数据库。

对于单实例数据库安装，使用如下命令：

```
$ sqlplus "sys / as sysoper"
Enter password: password
SQL> STARTUP
SQL> EXIT
```

对于 RAC 安装，使用如下命令：

```
$ srvctl start database -d db_name
```

关于如何启用或禁用 Oracle 数据库保险库的更多信息，可参阅 *Oracle Database Vault Administration Guide*。

49.1.4　数据库保险库安装的注意事项

在 Oracle Database 9.2.0.8、10.2.0.3 和 10.2.0.4 环境中安装 Oracle 数据库保险库的客户在开始安装过程之前应该验证称为 TEMP 的临时表空间存在。这一安装依赖性在 Oracle Database 11g 中已删除。

在 Oracle 数据库保险库安装过程中，安装人员提供用来创建账户管理责任的选项。Oracle 建议创建账户管理责任，以便在数据库保险库安全管理员(所有者)、账户管理和其他 DBA 的责任之间实现更加明确的职责分离。

1. Oracle SYSDBA 访问

Oracle 建议严格限制使用 SYSDBA 角色进行连接。只有在绝对必要时，和只有对那些仍要求用 SYSDBA 连接的应用程序(如 Oracle RMAN 和强制打补丁过程)，才使用 SYSDBA 角色连接到数据库。对于所有其他情况，创建指定的数据库账户来完成日常数据库管理操作。将来在 Oracle 中，任何活动都不再需要以 SYSDBA 连接。

2. Oracle 数据库管理

Oracle 建议将 Oracle SYSTEM 账户用作普通 DBA 的客户应该为他们的数据库管理员创建指定的 DBA 账户。这么做会提高数据库中管理行为的责任意识。

3. Oracle SYSTEM 用户

过去开发的很多应用程序已使用 Oracle 用户账户 SYSTEM 来存储某些应用程序表。对

于某些应用程序，可能有必要将 SYSTEM 账户添加到域授权中，程序才能继续正常工作。需要注意的是，可以在 SYSTEM 账户上建立约束，以增加这些情况下的安全性。例如，Oracle 数据库保险库规则集可用来限制 SYSTEM 用户到特定 IP 地址的连接。

4. 根和其他操作系统访问

Oracle 数据库保险库并不阻止权限高的操作系统用户直接访问数据库文件。为实现这种保护，建议采用 Oracle 的透明数据加密(TDE)。Oracle 建议通过创建个性化账户在操作系统级别上小心地评价和限制直接访问。

在 Linux 和 UNIX 环境中，这些个性化账户应该在需要的时候使用 sudo 来访问 Oracle 软件所有者。使用 sudo 可以控制每个个性化用户能够执行哪条特定命令。允许的命令不应该包括 make 和 relink。但是，在打补丁或紧急情况下，可以启用这两个命令，不过持续时间很有限。

5. 职责分离

通过提供对数据库操作(例如，创建账户、授予权限大的角色、改变表结构、将与安全相关的程序包用于虚拟专用数据库和标签安全)的细粒度控制，Oracle 数据库保险库可以帮助管理员更安全地管理操作。Oracle 数据库保险库的默认职责分离可以分为 3 种类型：

- 数据库账户管理　数据库保险库阻止即席式创建数据库账户，除非管理员被明确地分配了数据库保险库账户管理的管理员角色。
- 数据库管理　数据库的日常管理工作(例如，与创建和管理表空间以及调整优化参数相关的任务)保持不变。Oracle 数据库保险库阻止 DBA 角色的即席式授权以及访问如 DBMS_RLS 等功能强大的程序包。
- 电子商务套件数据库保险库安全管理　只有数据库保险库管理员才能改变与域和命令规则相关的数据库保险库电子商务套件安全设置。

安全管理员可以扩展 Oracle 为 Oracle 电子商务套件提供的安全设置，以适应他们特定的安全需求。定制的域和命令规则可与很多内置的数据库保险库因素联合使用，从而进一步加强安全性。例如，客户可以设置 CONNECT 命令规则将数据库连接限制到特定范围的 IP 地址，从而创建到 Oracle 数据库的可信路径，并有助于阻止应用程序绕开安全问题。关于如何扩展 Oracle 提供的安全设置的更多信息，可参考 Oracle Metalink 站点。

6. 命令规则

Oracle 数据库保险库使用命令规则的概念来控制任何用户对大多数 SQL 命令的执行。命令规则可用来控制 DDL 命令和 DML 命令。使用命令规则可以保护数据库及其应用程序，避免任何用户(包括应用程序所有者)未经授权改变它们或恶意改变它们。

命令规则可以使用数据库保险库管理 Web 接口(DVA)或数据库保险库管理应用程序编程接口(API)来创建。

Oracle 将在安装时创建和显示 DVA URL。URL 的格式是 http://hostname:portnumber/dva。以数据库保险库管理员(所有者)的身份登录。单击“命令规则”，然后单击“创建”。填充属

性，对想要限制的命令施加必要的限制。

数据库保险库 API 由 DVSYS.DBMS_MACADM 程序包中的过程和函数提供。此 PL/SQL 程序包允许用户编写应用程序来配置域、因素、规则集、命令规则、安全应用角色和 Oracle 标签安全策略，而这些东西通常是在 Oracle 数据库保险库中配置的。

只有具有 DV_ADMIN 或 DV_OWNER 角色的用户才能使用 DVSYS.DBMS_MACADM 程序包。

下面的过程使用 DVSYS.DBMS_MACADM 程序包来创建名为 PERS 的域，保护个人模式中的数据不被 DBA 访问。需要注意的是，将 Audit Option 设置为 1 可以在违反域时——例如，当未授权用户试图修改被 drop table 命令的域规则所保护的对象时——创建审计记录。

```
BEGIN
  DVSYS.DBMS_MACADM.CREATE_REALM
  ( REALM_NAME => 'PERS REALM',
    DESCRIPTION => 'This realm protects personnel data from
dba access', ENABLED => 'Y',
  AUDIT_OPTIONS => 1);
END;
/
COMMIT;
```

下面的代码保护 PERS 模式中的所有对象：

```
BEGIN
DVSYS.DBMS_MACADM.ADD_OBJECT_TO_REALM(REALM_NAME => 'PERS REALM',
  OBJECT_OWNER => 'PERS',
  OBJECT_NAME => '%',
  OBJECT_TYPE => '%');
END;
/
COMMIT;
```

下面的代码将 PERS 用户授权为 PERS 域的所有者：

```
BEGIN
DVSYS.DBMS_MACADM.ADD_AUTH_TO_REALM
  (REALM_NAME => 'PERS REALM',
   GRANTEE => 'PERS');
END;
/
COMMIT;
```

Oracle 的最佳实践表明，应该在 Oracle 数据库保险库中为如下 SQL 语句创建命令规则：

- CREATE CLUSTER
- DROP CLUSTER
- CREATE DATABASE LINK
- DROP DATABASE LINK
- DROP INDEX
- DROP SEQUENCE
- CREATE SYNONYM

- CROP SYNONYM
- ALTER TABLE
- CREATE TABLE
- DROP TABLE
- TRUNCATE TABLE
- DROP TABLESPACE
- CREATE VIEW
- DROP VIEW

关于使用 DVSYS.DBMS_MACADM 程序包的更多信息，请参阅 *Oracle Database Vault Administrator's Guide*。Oracle 数据库保险库最佳实践的清单可从 http://www.oracle.com/technology/deploy/security/database-security/database-vault/index.html 获得。

49.2 Oracle 内容数据库套件

Oracle 内容数据库套件(Oracle Content Database Suite)是可用于 Oracle 数据库企业版软件的一个选项。Oracle 内容数据库套件的主要用途是提供了一组服务来管理组织机构内的所有文档。

Oracle 内容数据库套件包含几个关键特性，讨论如下。

49.2.1 存储库

Oracle 内容数据库套件的存储库组件包括 Oracle 内容服务器(Oracle Content Server)和 Oracle 内容数据库(Oracle Content Database)。Oracle 内容服务器由访问 Oracle 内容数据库的程序和可执行代码组成：

- **Oracle 内容服务器** Oracle 内容服务器运行在所有标准的操作系统上，例如 Microsoft Windows Server、Red Hat、SuSe Linux、AIX、Solaris 和 HP-UX 等操作系统。Oracle 内容服务器要求 Web 服务器软件，并支持 Apache、Sun Java System Web Server、IBM HTTP server 和 Microsoft 的 Internet Information Service。Oracle 内容服务器也要求 Java 虚拟机(Java Virtual Machine，JVM)在 Web 服务器上执行服务器端 Java 程序。数据库软件也需要安装在相同的服务器上，除 Oracle 数据库以外，还支持几种不同的数据库。
- **Oracle 内容数据库** Oracle 提供默认数据库安装，可用来存储由 Oracle 内容服务器管理的内容的所有修订的元数据。将 Oracle 数据库用作 Oracle 内容数据库可提供使用内置的 Oracle Database 细粒度访问控制的功能，这允许在文件夹或文档级别上指定不同的安全等级。根据用户访问文件夹或文档中数据的业务需要创建数据库角色并把这些角色分配给用户。Oracle 内容数据库提供了审计机制，使组织机构能够审计内容数据库中的任何事件，并在需要的时候生成审计跟踪和报告。

49.2.2　文档管理

Oracle 内容数据库套件中的文档管理使用多个工具，包括 portlet 套件、集成转换器和脚本：

- Oracle Content Database Portlet Suite——Oracle Content Portlet Suite 能够将 Oracle 内容服务器的功能集成到各种门户服务器中，例如 WebLogic、WebSphere、Plum Logis 和 Sun One。
- Oracle Dynamic Converter——Oracle Dynamic Converter 用来将文档转换成期望的格式，以便存储和显示存储的内容。内容可以以 Microsoft Word、Adobe Acrobat 和其他几种格式在 Web 浏览器中或在无线设备上显示。
- Oracle Content Desktop——Oracle Content Desktop 软件安装在用户桌面上，提供客户端应用程序和内容服务器实例之间的接口。它也提供与 Microsoft Outlook 的集成，这可用来将内容直接传输到 Microsoft Outlook，或从 Microsoft Outlook 传输过来，包括电子邮件消息和附件。
- Oracle Content Integration Suite——Oracle Content Integration Suite(CIS)能够与内容服务器通信。除工作在非 J2EE 环境中以外，它还可以部署在很多不同的 J2EE 应用服务器上。
- Oracle Drive——Oracle Drive 是一个 WebDAV 客户。它将 Oracle Portal Repository 映射为一个驱动器，并允许直接从 Windows 桌面执行桌面授权和发布以及门户特有的元数据属性。
- Kofax Release Script——Kofax Release Script 是用来将纸质文档转换为电子文档的一组工具。

49.2.3　用户安全性

用户使用用户 ID 和口令登录到 Oracle Content Server，就像是登录到 Oracle 数据库一样。用户名和口令存储在 Context Server 中，也可以选择存储在 Oracle Context Database 中。有 4 种不同的用户：

- **消费者(consumer)**　消费者是存储在 Oracle Context Database 中数据的用户，他们使用 Web 浏览器搜索、查找和查看内容。消费者不能向 Context Database 添加内容，他们的登录凭证用来确定他们对哪些内容有访问权。
- **贡献者(contributor)**　贡献者也是存储在 Oracle Context Database 库中数据的消费者，但他们也可以向内 Context Database 添加新内容或修订 Context Database 中已有的内容。
- **管理员(administrator)**　管理员负责设置和维护 Oracle Context Server 和 Oracle Context Database 以及用户群体。管理员也可以指定子管理员来协助管理系统。数据库保险库管理员无法看到域所保护的数据。数据库保险库管理员也不能将他自己添加到他创建的任何域的授权用户列表中。这是 Oracle 数据库保险库的职责分离的一部分。

- **子管理员(sub-administrator)** 子管理员维护内容管理系统及其用户登录的一个子集。
- **预定义用户(predefined user)** Oracle Content Server 在安装时创建两个用户。第一个用户称为 sysadmin，他是默认的系统管理员。此用户 ID 被分配了管理员角色，且不能从系统中删除。应立即更改其默认口令。第二个用户称为 user1，它被分配了贡献者角色。根据需要，此用户 ID 可以从系统中删除。但如果保留此用户 ID，则应该修改其默认口令。

1. 安全组

登记在内容管理系统的所有文件都必须分配到一个安全组。Oracle 提供了两个默认的安全组："公有"安全组和"安全"安全组。任何消费者都可以查看"公有"安全组中的所有文档，而存储在"安全"安全组中的内容系统文件只能由系统管理员查看。使用用户管理员(User Admin)界面可以定义和管理更多的安全组。为用户分配权限，以便他们有权读、写、删除或管理安全组中的文档。也可以设置角色，提供一组权限给用户。

2. 外部用户安全性

可以设置Oracle Content Server安全性来使用Microsoft的活动目录(Active Directory)用户库或符合 LDAP 的用户库，如 iPlanet。当用户登录到 Content Server 时，系统会根据他们的属性为他们分配角色和账户，他们的属性定义在外部的安全系统中。

3. 网络访问

必须配置网络访问，以便通过 Oracle Content Server 应用程序只允许访问 Oracle 目录结构。由于 Oracle Content Server 中的数据目录和配置目录包含用户名和口令，因此不应该在网络上共享。

49.3 Oracle Records Database

Oracle 10.1 提供了 Oracle Records Database 新产品，这种产品可用来管理电子记录，这些电子记录记录了规则遵守情况和合法发现信息的情况。Oracle Records Database 为电子记录的声明、分类、存储和检索提供了基础。

同时使用 Oracle Records DB 与 Oracle Context Database 可以提供这样的功能，即指定将某个文档或其他内容实体保留一段特定的时间，防止或者控制在此保留期间对该文档进行更改，并在该保留期满后以预先确定的方式处理该文档。在 Oracle Records DB 中创建的策略是通过 Oracle Content DB 应用程序根据 Oracle Content DB 存储库中的信息来实现的。

Oracle Records DB 以 Web 服务的形式提供了记录管理的管理过程，因此，Oracle Records DB 可以与组织机构内的应用程序和业务流程集成在一起。管理员可以使用 Oracle BPEL 工作流捕获和声明存储在 Records DB 中的记录。同时也提供了相应的集成，可以捕获 Oracle 电子商务套件、PeopleSoft、Siebel 和 SAP 中的记录。

Oracle Records DB 管理员可以通过建立各种分层结构(包括 Records DB 中的记录系列、分类和文件夹)来建模其组织机构的策略，然后通过角色将不同级别的访问应用于数据。提供

的可扩展的审计功能可以收集全局审计历史记录和对象审计历史记录以及定制审计。

Oracle Records DB 为 Network Appliance SnapLock 和 EMC Centera 存储设备提供了硬件集成，以便使用 Oracle Records Database 中定义的策略来存储、维护和提供对电子文档的访问。配置是从 Oracle Content DB 管理控制台提供的。

2007 年，Oracle 通用记录管理器(Universal Records Manager，URM)取代了 Oracle Records DB。使用 Oracle URM 可以对数据库记录、电子邮件附件和文件服务器中存储的电子文档应用保留策略及合法发现信息。Oracle URM 也可以用来定义和管理通过了美国国防部 5015.2 标准认证的记录管理服务器上的记录和保留策略。

Oracle URM 就像 Oracle Records DB 一样使用 Web 接口，并使用 HTTP、SOAP 和 JavaBeans 进行通信。Oracle URM 包括以下 4 个主要部分：

- **记录管理器(Records Manager，RM)**　记录管理器用来控制记录的创建、分类、保留时间和销毁。分类和保留策略可定义并自动应用到新近创建的记录。可以定义保留策略来归档到期的记录，并把它们移动到辅助存储器，或者销毁它们。
- **物理记录管理器(Physical Records Manager，PRM)**　物理记录管理器用来管理以电子形式或物理形式存储的记录和非记录内容的保留和处置。
- **通知服务(Notification Services)**　通知服务与 Oracle 工作流服务联合使用以便为记录创建保留策略和通知。
- **适配器服务(Adapter Services)**　Oracle URM 提供了一组服务，这些服务可用来监控存储库、执行各种任务以支持 URM、管理通信。Oracle URM 以 Web 服务的形式提供这些服务，并把这些服务作为 Java 和.NET 软件开发工具箱的一部分。与 Oracle Records DB 一样，因为 Oracle URM 使用 Oracle Content Server，所以需要安装在与 Oracle Content Server 相同的服务器上。支持的平台包括 Windows Server 和 Linux。基本的 Oracle 数据库必须是 Oracle 10.2 或更高版本才能使用 Oracle URM，且应该与 Oracle Content DB、Oracle Records DB 和 Oracle URM 使用的安装平台无关。

Oracle Content DB 在配置时安装了很多数据库对象和模式。Oracle Content DB 创建了如下模式：

- content(内容)、content$dr 和 content$cm 用来管理内容和相关元数据
- 关系、索引等
- content$id 用来从 Oracle Internet Directory 启用用户资源供给
- 内置在工作流中的 Oracle Workflow 的 Owf_mgr

Oracle Content DB 也创建如下队列：

- ifs_bpel_in 和 ifs_bpel_out 用来与 Oracle BPEL Process Manager 通信
- ifs_in 和 ifs_out 用来与 Oracle Workflow 通信

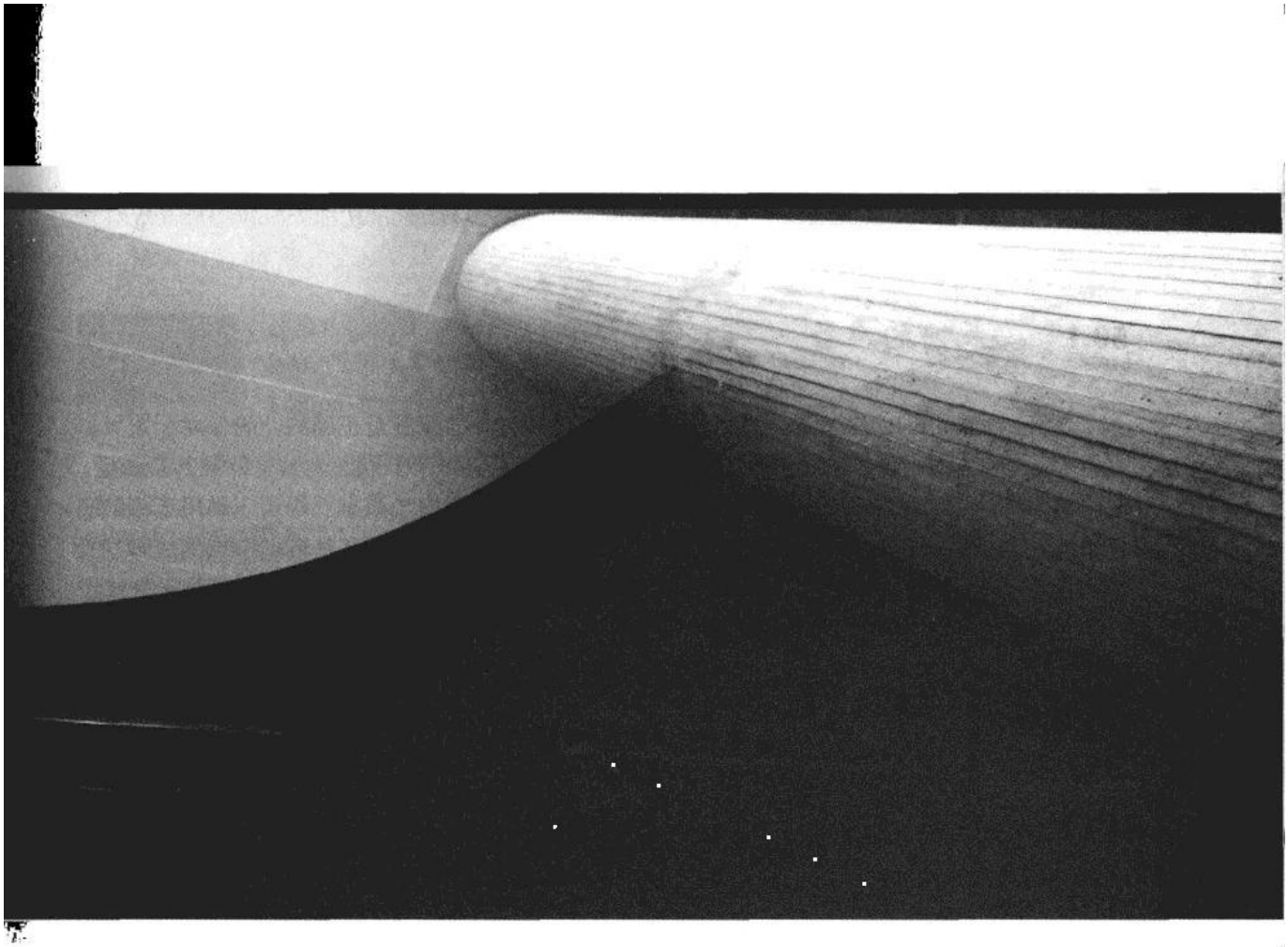
在数据库正常操作期间，一组定义良好的进程需要访问以前列出的数据库对象。Oracle 数据库保险库使组织机构可以创建一组域和命令规则，支持的规则和规则集可以限制对这一最小进程集的访问。

Oracle 在 otn.oracle.com 上提供了一组 PL/SQL 脚本，这些脚本启用正确的 Oracle 数据库保险库配置来保护数据库中的 Oracle Content DB 和 Oracle Records DB 数据。此配置由一组

域和规则组成，这些域和规则限制对 Oracle Content DB 对象的访问，另外也限制连接到知名机器——Oracle Content DB 中间层、Oracle BPEL Process Manager 中间层和 Oracle Internet Directory。这些规则有望与 IT 安全最佳实践联合使用，以保护和控制对这些相关机器的访问。

每个组织机构都需要调整这些规则以适应诊断和维护应用程序所需要的更多访问。DBA 可能需要直接访问 Content DB，生成 AWR 和 ADDM 或其他报告，以优化调整耗时太长的查询。建议在更多的域上启用这样的访问，包括只在必要时才启用的域和审计所有访问的域。

关于 Oracle Database Vault、Oracle Content DB、Oracle Records DB 和 Oracle URM 的更多信息、文档和下载请参见 otn.oracle.com。



第 50 章

Oracle 实时应用群集

在 Oracle 实时应用群集(Real Application Cluster, RAC)环境中，多个实例访问一个单独的数据库。这些实例通常在不同服务器上创建，这些服务器互相连接在一起，可以快速地访问共享磁盘区域。数据库文件驻留在共享磁盘集，并且每个实例具有自己的控制文件和在线重做日志文件，这些文件也存储在共享磁盘上。

每个实例共享相同的数据文件和相同的控制文件。联机重做日志文件被分配给使用 THREAD 参数的特定实例，只有具有此线程号的实例才写入联机重做日志文件中。每个实例必须能够读所有联机重做日志文件，以便拥有的实例关闭时，任何实例都能够归档日志文件。用户可以通过连接的其中一个实例连接到数据库。如果某个实例失败，就可以动态地通过群集中的另外一个实例重新连接到数据库。因此 RAC 提供高可用性的解决方案(解决服务器和实例故障)和可伸缩的高性能解决方案(通过增加可用于连接进程的内存和处理器)。

本章对 RAC 的安装和管理进行概述。很多细节是特定于操作系统的，但是配置进程流对不同平台是相同的。

50.1 安装前的准备

在开始 Oracle RAC 安装之前，必须验证环境和群集的配置是正确的，确保此配置可以支持 Oracle RAC。存放实例的服务器在群集中称为节点。每个节点应当具有外部共享磁盘。每个节点应当具有冗余转换器支持故障切换需求。每个节点应当具有一个用于私用连接的私用 Internet 协议(Internet Protocol, IP)地址以及一个用于客户端连接和从其他节点的故障切换连接的公有 IP 地址。

每个节点必须支持 TCP/IP(传输控制协议/Internet 协议)和支持互连软件协议。

注意：
必须选择 Oracle 支持的互连软件协议。关于最新的认证源，请参阅 Metalink 网站(http://metalink.oracle.com)。

Oracle 提供群集软件(clusterware)，简化安装和配置过程。对于共享磁盘区域，Oracle 推荐使用其自动存储管理(Automatic Storage Management)功能(参阅第 51 章)。

注意：
在开始安装过程之前，确保安装操作系统所需要的所有补丁。作为标准 Oracle 软件安装过程的一部分，必须检查所有需要的操作系统补丁级别。

DBA 和应用程序开发人员一般不参与网络系统设计，但对于 RAC 安装，您需要积极地参与网络配置。例如，MTU 网络设置确定数据在服务器之间发送时每个包是多大。在 RAC 安装中，很多数据库块可以在节点之间发送，尽可能高效率地通信是至关重要的。如果 MTU 设置为 1 500 字节(这是它的历史值，也是某些安装中的默认值)，那么每个 32KB 的数据库块将需要 20 多个独立的网络包。即使这些包在节点之间的传输需要 1 毫秒，读一个块的时间就等于 20 毫秒，您读取磁盘数据的速度也可能会比这快些。如果硬件支持增加 MTU，则可以提高块在互连群集间的传输速率。如果您正在移动到 RAC 环境，则需要理解网络和硬件配置如何影响应用程序的性能和可扩展性。系统管理员应该确保配置正确，使此配置支持块以尽可能少的系统开销跨 RAC 节点传输。

50.2 安装 RAC

Oracle Database 11g 中 RAC 的安装分为两个步骤。第 1 步，使用 Oracle 通用安装程序(Oracle Universal Installer，OUI)安装群集就绪服务(Cluster Ready Services，CRS)。CRS 为群集管理提供完整的解决方案集合。可以使用 CRS 管理群集成员、组服务、组资源管理和高可用性的配置。可以使用 CRS 定义支持在服务器上分配工作负载的服务。自动工作负载信息

库(Automatic Workload Repository，AWR)收集服务程序上的统计信息；关于 AWR 的详细信息，请参阅第 46 章。

注意：
RAC 可用于 Oracle Database 10g 和 Oracle Database 11g 的标准版(Standard Edition)和企业版(Enterprise Edition)。

安装过程的两个步骤都使用 OUI；关于 OUI 的详细信息，请参阅第 2 章。在完成第 1 步(安装 CRS)后，可以开始第 2 步的安装：在与第 1 步中使用的不同的 Oracle 主目录下安装带 RAC 的 Oracle Database 11g 软件。也可以使用安装程序为 RAC 环境创建并配置服务程序。当软件安装完成后，OUI 将启动数据库配置助手(Database Configuration Assistant)完成环境配置并创建 RAC 数据库。

注意：
如果已经安装了先前的 Oracle 群集数据库版本，则数据库更新助手(Database Upgrade Assistant, DBUA)将升级数据库到 RAC 的 11g 版本。

50.2.1　存储

可以使用自动存储管理(Automatic Storage Management，ASM)功能创建磁盘组，并简化其管理。如第 51 章所述，数据库管理员可以把新的磁盘添加到磁盘组中；Oracle 将管理所涉及磁盘中文件的内容和发布。

ASM 一般在数据库软件之前安装。在 Oracle 10.2.0 中，Oracle 建议将 ASM 安装在它自己的$ORACLE_HOME 目录中。在 Oracle 11g 中，Oracle 建议将 ASM 安装在与 CRS 相同的 $ORACLE_HOME 目录中。Oracle 11g 建议的方法有望成为未来版本的标准。

如果没有可用的硬件冗余，则可以使用 ASM 在 Oracle 内提供冗余。定义磁盘组时，可以指定故障切换组；Oracle 将使用该故障切换组作为磁盘组中主磁盘组的冗余形式。Oracle 的安装程序把基于软件的冗余称为“标准”冗余。可以指定多个故障切换组获得更多的冗余保护(以更多空间为代价)。如果硬件镜像可用，则可以使用外部(基于文件系统)冗余。

很多 RAC 环境同时使用原始设备和逻辑卷管理软件。逻辑卷管理(Logical volume management, LVM)为原始设备提供文件系统管理接口。

可以使用 ASM 磁盘组进一步降低管理 RAC 数据库的复杂性——Oracle 管理文件和文件系统。可以手动或通过 Oracle 企业管理器(Oracle Enterprise Manager, OEM)的网格控件(Grid Control)选项来管理文件和磁盘组。

50.2.2　初始化参数

作为配置实例的一部分，必须为保存在服务器参数文件中的数据库初始化参数设置值。RAC 安装应当使用服务器参数文件管理实例共享的参数集。

对于 RAC 群集中的所有实例，下面这些初始化参数必须具有相同的值：

- ACTIVE_INSTANCE_COUNT
- ARCHIVE_LAG_TARGET
- COMPATIBLE
- CLUSTER_DATABASE
- CLUSTER_DATABASE_INSTANCES
- CONTROL_FILES
- DB_BLOCK_SIZE
- DB_DOMAIN
- DB_FILES
- DB_NAME
- DB_RECOVERY_FILE_DEST
- DB_RECOVERY_FILE_DEST_SIZE
- DB_UNIQUE_NAME
- INSTANCE_TYPE (RDBMS or ASM)
- PARALLEL_MAX_SERVERS
- REMOTE_LOGIN_PASSWORD_FILE
- UNDO_MANAGEMENT

注意：
如果 DML_LOCKS 和 RESULT_CACHE_MAX_SIZE 设置为 0，则它们的值在所有实例中必须相同。

下面这些初始化参数对于 RAC 群集中的每个实例必须具有独一无二的值：

- ASM_PREFERRED_READ_FAILURE_GROUPS (只用于 ASM 实例)
- INSTANCE_NAME
- INSTANCE_NUMBER
- UNDO_TABLESPACE

注意：
在一个实例中，设置 INSTANCE_NUMBER 和 THREAD 为相同的值。

除了这些参数外，还应该将各实例的如下初始化参数指定为相同的值，具体参数如表 50-1 所示。

表 50-1 其他具有相同初始化值的参数

参 数	说 明
ARCHIVE_LAG_TARGET	Oracle RAC 数据库中实例具有不同的值可能会增加系统开销，因为数据库处理过程执行其他的自动同步。 当 Oracle RAC 数据库使用流时，值应该大于 0

(续表)

参　数	说　明
LICENSE_MAX_USERS	因为此参数确定在数据库中定义的对用户数量的限制，所以数据库的所有实例具有相同的值就很有用，这样可以查看当前值，而与正在使用的实例无关。设置不同的值可能会使 Oracle 在实例启动期间生成其他告警消息，或者导致与数据库用户管理相关的命令在某些实例上失败
LOG_ARCHIVE_FORMAT	如果不对所有实例使用相同的值，则不必使介质恢复复杂化。恢复的实例期望所需的归档日志文件名具有它自己的 LOG_ARCHIVE_FORMAT 值所定义的格式，而与哪个实例创建了归档日志文件无关。支持 Data Guard 的数据库必须对所有实例使用相同的 LOG_ARCHIVE_FORMAT 值，无论是发送还是接收归档重做日志文件
SPFILE	如果此参数不对所有实例识别相同的文件，则每个实例在故障切换、负载均衡和正常操作期间的行为可能不同，且无法预测。另外，使用 ALTER SYSTEM SET 或 ALTER SYSTEM RESET 命令使 SPFILE 所做的改变只保存在运行此命令的实例所使用的 SPFILE 中。您所做的改变不会反映在使用不同 SPFILE 的实例中。如果 SPFILE 值在由服务器设置这些值的实例中不同，则应该重新启动不使用默认 SPFILE 的实例
TRACE_ENABLED	如果想要使诊断跟踪信息始终对 Oracle RAC 数据库可用，则必须在所有数据库实例中将 TRACE_ENABLED 设置为 TRUE。如果只跟踪某些实例，则当需要的诊断信息只有在实例的 TRACE_ENABLED 设置为 FALSE 才可访问，诊断信息可能不可用
UNDO_RETENTION	通过在每个实例中设置不同的 UNDO_RETENTION 值，可能降低可伸缩性，并在故障切换之后遭遇不可预料的行为。因此，在将不同的 UNDO_RETENTION 参数值赋予 Oracle RAC 数据库中的实例之前，应该仔细考虑是否会从中获益

避免在 RAC 实例中使用动态内存管理功能。一般来说，动态内存管理产生的网络流量对数据库整体性能有负面影响，对数据库支持的应用程序也没有什么好处。

50.3　启动和停止 RAC 实例

可以使用网格控件管理与 RAC 数据库相关的所有实例。可以通过网格控件停止、启动和监控数据库、实例及其侦听器。对于所有非 RAC 数据库，可以使用 SQL*Plus 发出 shutdown 和 startup 命令。对于 RAC 数据库，如本节示例所示，可以使用 SRVCTL 实用程序。

不论使用什么工具来关闭或启动实例，下面的规则都适用于 RAC 群集：

- 关闭实例不影响其他正在运行的实例的操作。
- 关闭某个设置为共享模式的数据库需要关闭 RAC 群集中的所有实例。

- 在 shutdown normal 或 shutdown immediate 之后，不需要恢复实例。如果在 RAC 实例上使用 shutdown abort，则仍在运行的实例将为关闭的实例执行实例恢复操作。如果被终止的是最后运行的实例，则下一个启动的实例将执行恢复操作。
- 如果使用具有 local 选项的 shutdown transactional，则实例中的所有事务都将在关闭前提交或回滚。其他实例中的事务处理不影响本地的 shutdown transactional。如果不指定 local 子句，则 shutdown transactional 将等待在 shutdown 命令之前启动的所有实例中的所有事务处理完成。

可以使用 SQL*Plus 命令，每次关闭和启动一个 RAC 数据库，直到整个环境已经启动。关于 startup 和 shutdown 命令的详细信息，请参阅第 51 章。可以通过 startup mount 命令在本地节点启动实例。

可以使用 SRVCTL 实用程序，用一条命令管理多个实例。启动实例的语法是：

```
srvctl start instance -d <db_name> -i <inst_name_list>
                [-o <start_options>] [-c <connect_str> | -q]
```

srvctl start 之后的关键字指定将被启动的服务——实例、数据库、服务、节点应用程序(nodeapp)或自动存储管理(asm)。注意可以将多个实例作为 inst_name_list 参数值的一部分传递(通过逗号隔开的列表)。start_options 为 open、mount 和 nomount。当连接到数据库时，connect_str 参数传递连接字符串。如果指定 q 参数，则运行 SRVCTL 时告诉 SRVCTL 来提示用户连接信息。默认情况下，SRVCTL 将以 SYSDBA 连接到本地数据库。

以下命令将启动和打开本地数据库 MYDB，并将打开 MYDB 数据库的所有实例：

```
srvctl start -database -d mydb -o open
```

以下命令将为数据库 MYDB 启动两个实例：

```
srvctl start instance -d mydb -i mydb1,mydb2
```

对于 ASM 实例，指定节点名(-n)，实例名(-i)和启动选项(-o)，如下面的示例所示。实例名是可选的，因为在一个节点上只可以运行一个 ASM 实例。

```
srvctl start asm -n mynode1 -i asm1 -o open
```

注意：

可以使用 OEM 或 SQL*Plus 查看作为群集一部分的实例。在 SQL*Plus 中，查询 V$ACTIVE_INSTANCE 视图。该视图针对每个实例显示相应的一行——显示实例编号、主机名和实例名。

可以使用下面的命令停止数据库、实例和其他服务：

```
srvctl stop {database | instance | service | nodeapps | asm }
```

表 50-2 列出了其他可用的 SRVCTL 命令。

表 50-2　SRVCTL 命令选项

SRVCTL　命　令	说　明
srvctl add {database | instance | service | nodeapps | asm}	把 1 个配置添加到群集数据库配置中。srvctl add service 将服务添加到数据库中，并把它们分配给实例。可以使用 srvctl add service 命令为某个服务配置透明应用程序故障切换(Transparent Application Failover, TAF)策略
srvctl config {database | service | nodeapps | asm }	显示存储在 Oracle 群集注册表中的配置信息
srvctl disable {database | instance | service | asm }	禁用指定的对象，阻止它自动启动
srvctl enable {database | instance | service | asm }	使指定的对象可以自动启动
srvctl getenv {database | instance | service | nodeapps }	显示环境状态
srvctl modify {database | instance | service | nodeapps }	在不删除和添加 CRS 资源的情况下，修改对象配置
srvctl relocate service	把指定的服务名从一个指定的实例重新分配给另一个指定的实例
srvctl remove {database | instance | service | nodeapps | asm}	从群集环境中删除指定的对象
srvctl setenv {database | instance | service | nodeapps }	设置配置文件中的环境值
srvctl start {database | instance | service | nodeapps | asm}	为数据库、所有或指定的实例、所有或指定的服务名或者节点级应用程序启动 CRS 启用的非运行应用程序
srvctl status {database | instance | service | nodeapps | asm}	显示指定对象的当前状态
srvctl stop {database | instance | service | nodeapps | asm}	为数据库、所有或指定的实例、所有或指定的服务名或者节点级应用程序停止 CRS 应用程序
srvctl unsetenv {database | instance | service | nodeapps }	清除配置文件中的环境值

50.4　透明应用程序故障切换

发生实例故障时，RAC 提供透明应用程序故障切换(TAF)支持。为防止硬件故障，数据库文件存储器必须是容错的(通常采用 RAID 阵列、镜像或强制 ASM 软件镜像)。当用户通过实例连接数据库时，用户执行的任何事务处理记录在该实例的重做日志线程中。

如果实例出现故障(或通过 shutdown abort 关闭)，则继续存在的某个实例将执行所必需的恢复。使用 TAF，用户的连接将通过同一群集中的另一个实例重新建立，连接到同一数据库。

当安装和配置 RAC 环境时，Oracle 配置网络服务。例如，如果创建服务 db.us.acme.com，则 tnsnames.ora 文件中的服务名记录项可能类似于：

```
db.us.acme.com=
  (description= (load_balance=on) (failover=on)
  (address_list=
    (address=(protocol=tcp)(host=db1-server)(port=1521))
    (address=(protocol=tcp)(host=db2-server)(port=1521)))
  (connect_data= (service_name=db.us.acme.com)))
```

用户可以通过 db.us.acme.com 服务名(例如，通过远程 SQL*Plus 会话)连接实例并执行查询。当用户连接后，可以关闭 db1-server 主机。如果用户(仍然在同一会话中)试图执行另一个查询，则会话将通过 db2-server 主机透明地重新建立，并执行该查询。

数据库的 listener.ora 文件记录项将包含与下面程序清单相似的记录项：

```
listeners_db.us.acme.com=
  (address=(protocol=tcp)(host=db1-server)(port=1521))
  (address=(protocol=tcp)(host=db2-server)(port=1521))
```

如果用户已经修改了会话中的环境变量(通过 alter session)，或者已经设置了特定于会话的变量，则那些值在新连接的会话中可能必须重新建立。

如本章前面所述，您可以创建和管理 RAC 群集中的服务。可以指定服务运行在一个或多个实例上；服务结合群集数据库资源来简化群集的可管理性。

可以使用 SVRCTL 创建和添加服务。可以把服务分配给实例，用于首选的(标准)和可用的(恢复)处理。可以标识其他可用的实例，以支持服务级修改或用于预先计划的运行中断。

注意：
可以使用 DBCA 管理服务分配和服务优先级。

把服务分配给节点是 RAC 设计过程的一个重要组成部分。其目标是优化配置，而不是阻碍配置。为了达到这一目标，应该避免创建将执行多个会话的 RAC 配置，这些会话长时间在群集的不同节点上同时更新相同的块。如果能够将用户隔离开，使他们能够共享数据而不是争用数据，就可以大大增加成功的可能性。可能需要将特定用户强制在群集的特定节点，而不依赖用户的循环分配。您越能够控制对表使用锁，就越能够控制群集的性能。

50.5 为群集添加节点和实例

可以给已有的群集添加节点，并给已有的 RAC 环境添加实例。节点必须符合本章前面所提出的条件——它们必须配置合适的网络协议，而且必须支持与群集内其他节点之间的高速数据传输。它们通常最简单，以使在新节点上的操作系统成为群集中的已有的某个节点的副本。节点必须能通过等价文件执行远程命令。

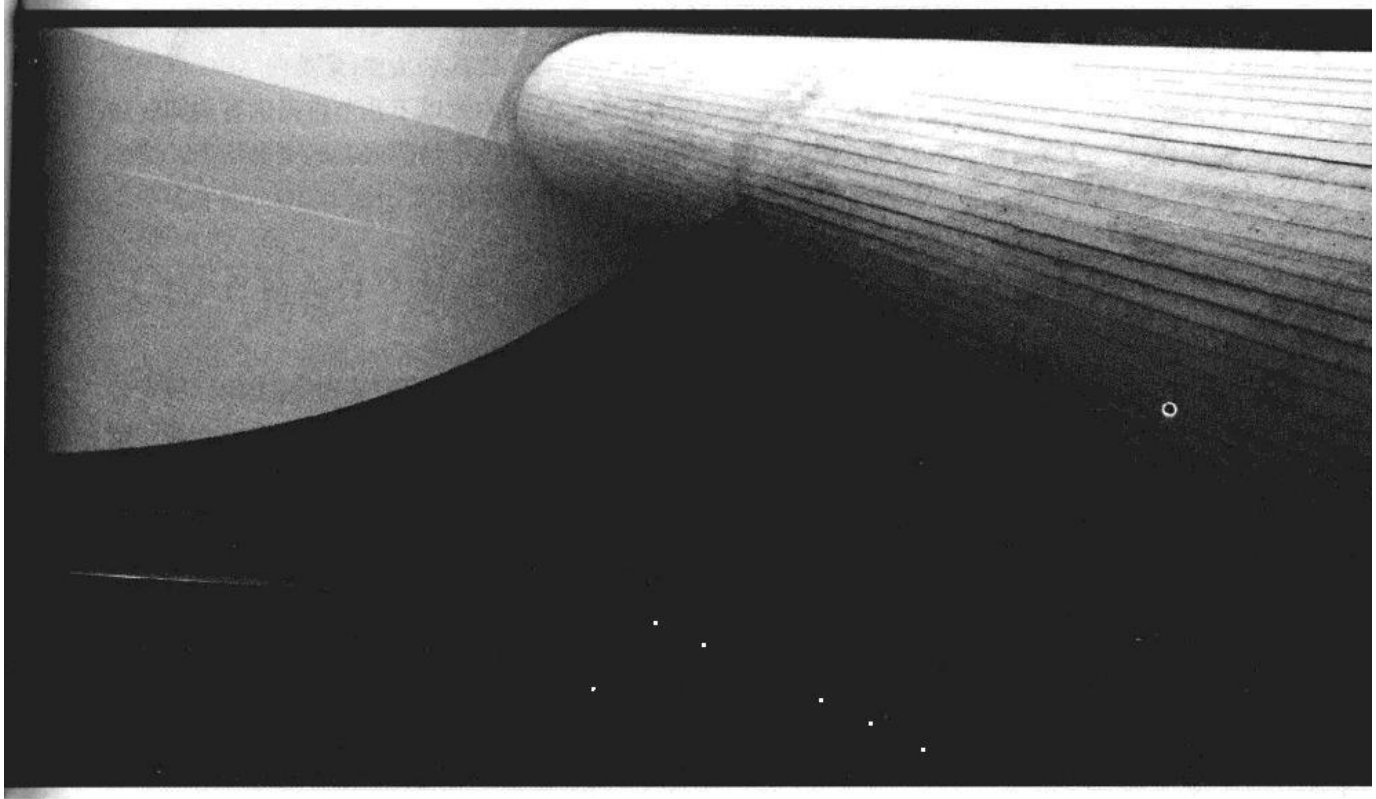
新节点具体的安装步骤因操作系统而异。通常，UNIX 群集管理员将把新节点添加到群集软件配置中(基于其供应商的说明)。然后可以启动 Oracle 通用安装程序(OUI)的添加节点(addNode)模式(在 UNIX 中，通过<CRS_HOME>/OUI/bin/addNode.sh；在 Windows 中，通过

<CRS_HOME>\oui\bin\addNode.bat)。当验证配置时，OUI 将提供一系列提示，以把新节点添加到群集中。

接着，OUI 将开始安装过程，根据需要，实例化根脚本，并把 CRS 文件复制到新的节点，然后提示在操作系统级运行几个脚本。然后 OUI 保存群集目录。然后可以执行 CRSSETUP 实用程序，为新的节点添加 CRS 信息。然后可以运行最后一个实用程序 RACGONS，为新的服务器配置 Oracle 通知服务端口。关于 CRSSETUP 和 RACGONS 实用程序的语法请参阅 Oracle 文档，它们包含节点名、节点编号和端口的详细配置。

一旦节点添加到群集中，就可以添加实例。启动数据库配置助手(参阅第 2 章)，选择“Real Application Cluster database”(实时应用群集数据库)。然后选择“Add instance(添加实例)”和实例将要访问的数据库名。在提示的地方输入实例参数，并接受概要信息启动实例的创建过程。Oracle 将创建实例及其网络服务。

为了从 RAC 群集中删除已有的实例，在不包含将要删除的实例的节点上启动 DBCA。DBCA 将在配置显示当前实例的一个列表；从列表中选择要删除的实例。DBCA 将显示分配给该实例的服务的一个列表，以便将它们重新分配给群集中的其他实例。选择的实例将从 RAC 群集中撤消注册。



第51章

数据库管理指南

本章将介绍管理 Oracle Database 11g 数据库所涉及的基本步骤。数据库管理员 (DBA)的工作由很多部分组成，并且还有许多专门为 DBA 所撰写的书籍。本章将概括介绍 DBA 的任务，并给出标准 DBA 工具的使用指南。但是本章并不面面俱到地介绍每个 DBA 任务。

前几章已经介绍了许多 DBA 的功能，如创建表、索引、视图、用户、同义词、数据库链接和程序包。因此，本章介绍的 DBA 主题将集中于 DBA 角色的产品控制功能：

- 创建数据库
- 启动和停止数据库
- 设置和管理数据库内存区域的大小
- 分配和管理对象的空间

- 创建和管理撤消
- 执行备份

对于以上每个主题，开发人员与 DBA 都有许多选择。本章以下几节将提供有关如何开始工作，以及如何向 DBA 和系统管理员询问的大量信息。关于这些内容的更多信息，请参考清华大学出版社引进并出版的《Oracle Database 11g DBA 手册》。

51.1 创建数据库

生成 create database 脚本的最简单的方法是借助于 Oracle 通用安装程序(OUI)。在安装 Oracle 时，OUI 提供了通过数据库配置助手(DBCA)创建数据库的选项。如果使用该选项，则作为软件安装过程的一部分，DBCA 将创建一个数据库。以后您可以随时调用 DBCA 来创建其他数据库。完整的命令语法，请参阅附录 A 中的“CREATE DATABASE”条目。

create database 命令是从 SQL*Plus 中，利用具有 SYSDBA 系统权限的账户发布的：

```
connect system/manager as sysdba
startup nomount
create database...
```

为了成功地发布 connect as SYSDBA 命令，须在操作系统级和数据库级有相应的权限。关于所需的操作系统权限的有关信息，请参阅操作系统专用的 Oracle 文档。

使用 Oracle 企业管理器

Oracle 企业管理器(Oracle Enterprise Manager，OEM)是一个图形用户界面(GUI)工具，是标准 Oracle 工具集的一部分。该工具允许 DBA 从个人电脑上管理数据库。OEM 工具集提供了管理远程数据库的强大界面。所有 DBA 都可以使用相同的中心 OEM 存储库(即在数据库中创建的一组表)来完成他们的工作。除了这些变化外，OEM 还包括任务调度和分配功能，以使数据库不停地覆盖。

在安装和配置 OEM 之前，必须作出几个关键性的决定。应当确定 OEM 的存储库创建在何处，以及打算怎样和何时执行备份来保护该信息库。由于可以将 OEM 用作 Oracle 恢复管理器(Oracle Recovery Manager，RMAN)的一个界面，因此恢复信息可以存储在 OEM 存储库中。可以创建一个小的、单独的数据库来存储 OEM 存储库。应当确保经常备份该数据库，以保证存储库本身的恢复。

如果您是唯一使用 OEM 工具集的 DBA，则不必考虑谁来管理环境中的特定数据库。如果站点中有几个 DBA，则应当确定任务定义、数据库职责和调度安排。使用 OEM，可以依据各个任务来给组中的每个 DBA 授予访问级别和权限。可以配置 OEM，使您能够向其他 DBA 发送电子邮件请求和任务安排，或对问题进行控制，以加快解决问题的速度。

如果已经安装了以前的 OEM 版本，则可将信息库移动到最新的版本中，以便使用新的功能。如果系统中有多个存储库，则需要采取措施，确保移动每个存储库时不会破坏当前已存储的信息。

虽然不一定要使用 OEM，但它提供了管理数据库的一个公用 GUI。当企业规模(以及数据库和 DBA 的数量)增大时，DBA 界面的一致性将支持更改控制过程和生产控制过程的一致性实现。

OEM 可用于管理实时应用群集(RAC)环境和 Grid 环境。通过给管理人员或者数据库提供图形界面，OEM 简化了 DBA 新手的学习过程，并使很复杂的功能可以简单地完成。

51.2　启动和停止数据库

要启动数据库，应当在 SQL*Plus 中发布 startup 命令，如以下程序清单所示。在本章的示例中，数据库名为 MYDB。

```
connect system/manager as sysdba;
startup open MYDB;
```

另外，可以安装该数据库：

```
connect system/manager as sysdba;
startup mount MYDB;
```

当已安装数据库但还未打开时，可以管理其文件。例如，如果在数据库关闭时移动了数据库的某些文件，则应当在重新启动之前让 Oracle 知道能从哪里找到这些文件。为了给出这些文件的新位置，可以安装该数据库(如上面的程序清单所示)，然后使用 alter database 命令在新的位置重命名旧文件。一旦已经告诉了 Oracle 这些文件的新位置，就可以通过 alter database open 命令打开数据库了。

关闭数据库时主要有 4 种选择。在正常关闭(默认)时，Oracle 在关闭之前等待所有用户退出数据库。在事务型关闭中，Oracle 在关闭前等待活动的事务处理完成。在立即关闭中，Oracle 回滚所有尚未提交的事务，并使当前已登录的用户退出。在异常关闭中，数据库立即关闭，所有未提交的事务都将丢失。

注意：
数据库在关机或启动过程中不允许新的登录。

为执行正常的关闭，可以使用 shutdown 命令。事务型关闭使用 shutdown transactional。立即关闭使用 shutdown 命令的 shutdown immediate 版本，如下面的程序清单所示。要异常中止数据库，应使用 shutdown abort。

```
connect system/manager as sysdba;
shutdown immediate
```

如果使用 shutdown abort，则在下一次数据库启动期间，Oracle 将自动执行恢复操作。shutdown abort 在某些情况下也是有用的，例如，在即将发生电源故障时需要在最短的时间内停止数据库。在后续的启动中，Oracle 会执行任何必要的恢复操作。

还可以使用 OEM，通过 Instance Manager/Configuration 对话框上的 General 选项卡来启动和关闭实例。

51.3 设置和管理内存区域大小

在启动数据库时，Oracle 分配一个供所有数据库用户共享的内存区域(系统全局区，SGA)。SGA 的两个最大的区域通常为数据库缓冲区缓存(database buffer cache)和共享池(shared pool)，它们的大小将直接影响数据库的内存需求和数据库操作的性能。它们的大小受数据库初始化文件中的参数控制。

数据库缓冲区缓存是 SGA 中的一个区域，用来保存从数据库的数据段(如表和索引)中读取的数据块。数据库缓冲区缓存的大小由数据库的初始化参数文件中的 DB_CACHE_SIZE 参数(以字节数表示)确定。数据库块的默认大小在数据库创建过程中，通过参数文件中指定的 DB_BLOCK_SIZE 参数设置。管理数据库缓冲区缓存的大小是管理和优化数据库的重要组成部分。

虽然数据库具有默认的块大小，但可以为不同的数据库块大小创建缓存区，然后创建使用这些缓存的表空间。例如，可以创建数据库块为 4KB 的数据库，而某些表空间设置为 8KB。8KB 的缓存大小通过 DB_8K_CACHE_SIZE 参数来设置。为了创建使用此缓存的表空间，在 create tablespace 命令中指定 blocksize 8K 即可。如果数据库的默认块大小为 4KB，则不设置 DB_4K_CACHE_SIZE 的值。为 DB_CACHE_SIZE 指定的大小将用于 4KB 的缓存。

注意：
在创建使用指定块大小的表空间之前，指定该块大小的缓存必须存在。

可以在数据库运行期间重新设置不同的高存区。如果未分配的内存可用，则缓存只能增大。这些缓存必须按粒度(granule)增加或减少。对于一个 SGA 小于 128M 的数据库来说，粒度大小为 4MB。因此，DB_8K_CACHE_ SIZE 可以为 4M、8M、12MB 等。如果试图使用任何其他的设置，Oracle 就会舍入为下一个接近的粒度大小。下面的程序清单显示了参数 DB_8k_CACHE_SIIE 的设置。为了查看内存是否可用，查询 V$SGA_DYNAMIC_FREE_MEMORY 视图。

```
alter system set DB_8K_CACHE_SIZE = 8m;
```

如果创建一个使用非默认数据库块大小的表空间，则必须保证能在数据库参数文件中更新相应的缓存大小参数(如 DB_8K_CACHE_SIZE)。如果使用 init.ora 文件，则必须用新值更新它。如果使用系统参数文件(首选的方法)，那么在执行包含 scope=both 子句的 alter system 命令时，它会自动更新系统参数文件。

注意：
不能在数据库打开时，更改 SGA_MAX_SIZE 参数或 JAVA_POOL_SIZE 参数的值。

Oracle 将使用一种在内存中尽量长期地保持最常用的块的算法，来管理缓冲区缓存中的空间。当缓存中需要可用空间时，新的数据块将使用经常访问的块占用的空间，或者使用修改过的块(一旦它已经写入磁盘中)占用的空间。

如果 SGA 的大小不足以容纳最常用的数据，则不同的对象将争用数据块缓冲区缓存中的空间。当多个应用程序使用同一个数据库从而共享相同的 SGA 时，尤其可能发生争用。在这种情况下，每个应用程序中最近使用的表和索引与其他应用程序中最近使用的对象相互争用 SGA 中的空间。结果，对数据块缓冲区缓存的数据的请求将使“占用”与“丢失”的比率更低。数据块缓冲区缓存的丢失将导致数据读取的物理 I/O 操作，从而使性能下降。

共享池存储数据字典缓存(关于数据库结构的信息)和库缓存(关于对数据库执行的语句的信息)。当数据块缓冲区和字典缓存允许数据库中的用户共享结构信息和数据信息时，库缓存允许共享常用的 SQL 语句。

共享池包含对数据库运行的 SQL 语句的执行计划和分析树。当一个相同的 SQL 语句第二次运行时(由任何用户)，Oracle 能够利用共享池中的分析信息来加速该语句的执行。与数据块缓冲区缓存一样，共享池通过 LRU 算法进行管理。当共享池装满时，最近最少使用的执行路径和分析树将从库缓存中删除，以便为新的项腾出空间。如果共享池太小，则语句将被不断地重新加载到库缓存中，这将影响性能。

共享池的大小(以字节为单位)通过 SHARED_POOL_SIZE 初始化参数设置。和其他缓存一样，共享池的大小可以在数据库打开时进行改动。

可以使用共享内存自动管理(Automatic Shared Memory Management，ASMM)动态地管理内存区域的大小。要启动 ASMM，应设置 SGA_TARGET 数据库初始参数为非零值。在设置 SGA_TARGET 为想要的 SGA 大小之后(即，所有的缓存加在一起的和)，将与缓存相关的其他参数(DB_CACHE_SIZE、SHARED_POOL_SIZE、JAVA_POOL_SIZE 和 LARGE_POOL_SIZE)都设置为零。要使这些更改生效，关闭并重启数据库；然后数据库将开始主动管理不同缓存的大小。可以通过 V$SGASTAT 动态性能视图来监控任意时刻缓存的大小。当数据库运行时，可以改变 SGA_TARGET 参数值，而且 Oracle 会自动改变相关缓存的大小。

注意：
为了使实例间流量最小，RAC 实例应该使用固定的缓存大小，而不是动态的缓存管理。

当数据库的工作负载改变时，数据库将改变缓存大小，用来反映应用程序的需要。例如，如果在夜间有重量级的批处理负载，而在白天需要更多的在线事务负载，则当负载更改时，数据库可改变缓存大小。这些变更是自动的，不需要 DBA 的干预。如果指定初始化参数文件中池的值，则 Oracle 将此值用作该池的最小值。

注意：
DBA 可以在缓冲区缓存中高速 KEEP 池和 RECYCLE 池。KEEP 池和 RECYCLE 池不受动态缓存大小更改的影响。

从 OEM 内部，通过单击 Memory Parameters 选项，可以查看动态内存管理是否启用；Automatic Shared Memory Management 按钮可设置为启用或者禁用。

初始化参数文件

数据库实例的特征(如 SGA 的大小和后台进程数量)是在启动过程中设置的。这些参数存储在称为系统参数文件的文件中。为了为打开的数据库生成系统参数文件，可使用 create spfile 命令。可以通过 create pfile from spfile 命令，生成系统参数文件的一个可读版本。这些命令的语法可参见附录 A。此参数文件的“pfile”版本(以前称为 init.ora 文件)一般包含数据库名作为文件名的一部分。例如，MYDB 数据库可以具有一个名为 initmydb.ora 的 pfile 文件。通常，应当使用服务器参数文件选项来管理参数设置，因为这样能支持数据库运行时参数的动态更改。

51.4 分配和管理对象的空间

创建数据库时，它被划分为称为表空间的多个逻辑部分。SYSTEM 表空间是创建的第一个表空间，而 SYSAUX 表空间是作为每一个 Oracle Database 11g 数据库的一部分而创建的。然后，可以创建其他的表空间来保存不同类型的数据(如表、索引和撤消段)。可以通过 alter tablespace 命令的 rename to 子句，重命名表空间。

注意：
从 Oracle 10g 开始，创建的 SYSAUX 作为每个数据库的一部分。如果使用 Oracle 提供的数据库升级助手，则自动创建 SYSAUX。

创建表空间时，就创建了保存其数据的数据文件(datafile)。这些文件在其创建过程中立即分配指定的空间。每个数据文件只能支持一个表空间。数据文件可设置为用完空间时自动扩展，可以设置它们扩展的增量及最大大小。每个用户的模式是一个逻辑数据库对象的集合，如表和索引，它们引用存储在表空间中的物理数据结构。用户模式中的对象可以存储在多个表空间中，而一个表空间可以包含多个模式中的对象。

创建数据库对象(如表或索引)时，通过用户默认或给出的指令，将它分配到一个表空间。在该表空间中创建一个段，用来保存与该对象有关的数据。

段由称为区的多个部分组成，区是一组连续的 Oracle 块。一旦已有的区不能再容纳新的数据，该段就获得另一个区。如果一个段由多个区组成，则不保证这些区是连续的。

表空间可以创建为本地管理(默认)或字典管理。数据库有一个或多个表空间，而表空间由一个或多个数据文件组成。在这些表空间中，Oracle 存储数据库对象的段。每个段可以有

多个区。特定对象的区信息存储在数据字典和段头中。如果段的增长超过特定数量的区(这取决于块大小和平台)，Oracle 就将创建区映射块，以跟踪更多的区。

管理表空间、数据文件、段和数据库对象使用的空间是 DBA 的基本职责之一。

当创建表空间时，指定表空间的默认存储参数。在此表空间中创建的每个对象将使用这些存储参数，除非显式地重写它们。通过建立一组连续的存储参数值，可以简化表空间中的空间分配和管理，如 51.4.1 节所示。

51.4.1　存储子句的含义

段使用的空间数量由其存储参数确定。这些参数在创建段时由数据库确定。如果在 create table、create index 或 create cluster 命令中没有特定存储参数，则该数据库将使用存储该段的表空间的默认存储参数。

注意：
可以通过 create user、alter user 和 grant 命令给用户分配默认表空间，并在这些表空间中分配空间限额。相应的命令语法，请参阅附录 A。

在创建表、索引或其他段时，可以使用本地管理表空间(推荐选项)的默认值，或作为 create 命令的组成部分指定 storage 子句。还可以指定 tablespace 子句，它可以使 Oracle 将数据存储在特定的表空间中。例如，在字典管理的表空间中，create table 命令可以包含以下子句：

```
tablespace USERS
storage (initial 1M next 1M pctincrease 0
minextents 1 maxextents 200)
```

如果没有指定 storage 子句，则使用该表空间的默认存储参数。对于本地管理的 USERS 表空间，create table 命令只需要包含以下命令即可：

```
tablespace USERS
```

存储参数指定了 initial 区的大小、next 区的大小、pctincrease(每个后继区将几何增长的因子)、 maxextents(区的最大数目)和 minextents(区的最小数目)。在创建了段之后，除非将对象进行重新组织，否则不能更改 initial 值和 minextents 值。每个表空间的存储参数的默认值可以在 DBA_TABLESPACES 视图和 USER_TABLESPACES 视图中找到。

在创建表空间时，可以为其指定默认存储参数。下面的命令创建一个本地管理的表空间并指定其默认存储参数。关于 create tablespace 的语法，请参阅附录 A。

```
create tablespace CODES_TABLES
datafile '/u01/oracle/VLDB/codes_tables.dbf' size 10M
extent management local uniform size 256K;
```

创建段之后，它至少将获得一个区。初始区将用来存储数据，直到它不再拥有可用空间为止(pctfree 子句可用来在段的每个块内保留一定比例的空间，以便为已有数据行的更新留下空间)。如果创建字典管理的表空间，则其段的大小可不统一；可以通过获得第二个区来扩充

段，该区的大小由 next 参数指定。默认情况下，使用本地区管理来创建表空间。

在此示例中，假定创建该表空间的数据库的块大小为 4KB，那么创建表空间时带有 extent management 语句，该语句声明为 local 且带有 uniform size 256K 子句。位图中的每一位描述 64 块(256/4)。如果省略 uniform size 子句，则默认为 autoallocate。uniform 的默认大小为 1MB。

注意：
如果在 create tablespace 命令中指定 local，则不能指定 default storage 子句、minextents 或 temporary。

本地管理的表空间可以接管字典管理的表空间中 DBA 所执行的某些空间管理任务。由于其体系结构，表空间不太可能成为碎片，它们的对象也不太可能出现与空间有关的问题。

可以使用少量的存储参数创建一组本地管理的表空间，并解决数据库中大多数空间请求问题。例如，可如下所示创建 3 个 DATA 表空间：

```
create tablespace DATA_SMALL
datafile '/u01/oracle/VLDB/data_small.dbf'
size 100M
extent management local uniform size 1M;

create tablespace DATA_MEDIUM
datafile '/u01/oracle/VLDB/data_medium.dbf'
size 1000M
extent management local uniform size 4M;

create tablespace DATA_LARGE
datafile '/u01/oracle/VLDB/data_large.dbf'
size 10000M
extent management local uniform size 16M;
```

在此示例中，DATA_SMALL 表空间用每个大小为 1MB 的区创建对象。DATA_MEDIUM 使用 4MB 的区大小，DATA_LARGE 使用 16MB 的区大小。如果有一个较小的表，则可把它放入 DATA_SMALL 表空间。如果它的大小增长较快，则可将其移入 DATA_MEDIUM 或 DATA_LARGE 表空间。

在每个示例表空间中，所有对象都具有相同的存储参数，这能简化以后执行的任何维护工作。使用大小一致的区还能改善从表空间中删除或移走对象时的空间的重用性。

以下几节描述各种主要对象类型的空间管理。请注意，如果使用的是本地管理的表空间，则有许多 storage 子句不适用，要依赖于表空间来管理基于在表空间级定义的参数的空间。虽然这会导致某些空间的过度分配，但极大地简化了数据库中的空间管理工作。

51.4.2 表段

表段(table segment)也称数据段(data segment)，存储与表或群集相关的数据行。每个数据段包含一个头块，这些头块可作为该段的空间目录。

一旦数据段获得了一个区，它将就保持该区，直到该段被删除、截断或压缩。使用 delete 命令从表中删除行，不影响已经分配给该表的空间数量。区的数量将不断增加，直到达到用户的表空间限额，或者表空间中的空间用完(如果数据文件不能自动扩展)为止。

要使数据段中浪费的空间数量最小化，可以调整 pctfree 参数。该参数指定每个数据块中保持可用的空间数量。然后，在值为 NULL 的列更新为非空值时，或者在更新为该行的其他值从而使该行长度增加的情况下该可用空间可使用。pctfree 的正确设置要针对特定的应用程序，因为它依赖于将要执行的更新操作的特性。

可以用 alter table 命令更改已有表的大部分存储参数。可使用 alter table 命令的 move 选项更改表的表空间分配。

51.4.3　索引段

和表段一样，索引段(index segment)保存已经分配给它们的空间，直到删除它们为止。但是，如果它们索引的表或群集被删除，则它们也将被间接地删除。为了使争用最小化，索引应当存储在与保存其相关表的表空间不同的表空间中。

可以使用 alter index 命令的 rebuild 选项更改索引的 storage 设置和 tablespace 设置。在下面的程序清单中，JOB_PK 索引在 INDEXES 表空间中被重新构建。

```
alter index JOB_PK rebuild
    tablespace INDEXES;
```

在索引的重建(rebuild)过程中，旧索引和新索引同时临时性地存储在数据库中。因此，在执行 alter index rebuild 命令之前，必须有足够的空间可用来存储两个索引。

在创建索引时，优化程序默认收集有关索引内容的统计信息。

51.4.4　系统管理的撤消

可以使用自动撤消管理功能将所有撤消数据放置在一个表空间中。在创建了撤消表空间时，Oracle 用系统管理的撤消(SMU)来管理回滚数据的存储、保留和空间利用情况。

为开始使用 SMU，必须首先使用下面的命令创建一个撤消表空间：

```
create undo tablespace undo_tablespace datafile 'filespec' size n;
```

接着，设置系统参数文件中或 init.ora 文件中的 UNDO_MANAGEMENT 初始化参数为 AUTO。可以通过 UNDO_TABLESPACE 初始化参数指定撤消表空间。不需要在撤消表空间中创建或管理回滚段。

1. 设置撤消保留

可以通过 UNDO_RETENTION 初始化参数，明确地指定 Oracle 在撤消表空间中保留撤消数据多长时间。保留撤消数据有两个目的：它不仅支持保留长时间运行的查询的非活动、在用数据，而且支持显示当前数据以前状态的查询(参考第 29 章)。

例如，如果设置：

```
UNDO_RETENTION=600
```

则 Oracle 将尽最大努力在数据库中将所有已提交的撤消数据保留 600 秒钟。通过此设置，所花时间小于 10 分钟的任何查询都将不会导致 ORA-1555(“snapshot too old”)错误。在数据库运行时，可用 alter system 命令更改 UNDO_RETENTION 参数值。可以使用 create tablespace 和 alter tablespace 命令迫使 Oracle 保证，甚至以牺牲当前事务为代价来维护请求的撤消保留。

2. 创建撤消表空间

在数据库创建过程中，可以创建一个撤消表空间，如下面的程序清单所示：

```
create database UNDODB
undo tablespace UNDO_TBS
...
```

在现存的数据库中，可以使用 create tablespace 命令创建撤消表空间：

```
create undo tablespace UNDO_TBS
datafile '/u01/oracle/undodb/undo_tbs_1.dbf'
size 100m;
```

关于 create tablespace 命令的完整语法，请参阅附录 A。

一旦创建了撤消表空间，就可以使用 alter tablespace 命令管理它。例如，可以给撤消表空间添加数据文件或重新命名已有的数据文件，就像对任何其他表空间一样。为了停止将某个表空间作为撤消表空间，需要用 alter system 命令给 UNDO_TABLESPACE 初始化参数设置一个新值，或者在关闭/启动数据库时更改该参数。

51.4.5 临时段

临时段(temporary segment)存储分类操作过程(如大型查询、索引创建以及 union 操作)中的临时数据。每个用户都有一个临时表空间，该表空间是在通过 create user 创建账户或通过 alter user 更改账户时指定的。用户的临时表空间必须是已被指定为临时表空间的某个表空间。

在创建数据库时，可以为所有用户指定一个默认的临时表空间。对于已有的数据库，可以用 alter database 命令更改默认的表空间，如下面的程序清单所示：

```
alter database default temporary tablespace TEMP;
```

在更改数据库的默认临时表空间分配时，已经分配旧的默认临时表空间的用户将被重定向到新默认的临时表空间。为查看当前默认的临时表空间，可执行下列查询：

```
select Property_Value
  from DATABASE_PROPERTIES
where Property_Name = 'DEFAULT_TEMP_TABLESPACE';
```

可以使用 create temporary tablespace 命令指定一个表空间作为“临时”表空间。临时表空间不能用来保存任何永久段，它只能保存在操作过程中创建的临时段。使用临时表空间的第一次排序在临时表空间中分配一个临时段；当查询结束时，不删除该临时段使用的空间。而是由其他的查询接着使用，这样可使排序操作节省分配和释放临时段的空间的系统开销。如果应用程序经常使用临时段进行排序操作，则可以使用一个专用的临时表空间，这样，排

序过程可以执行得更快。

可以创建多个临时表空间并将它们组合。使用 create temporary tablespace 或者 alter tablespace 命令的 table group 子句将这些临时表空间分配给表空间组。然后指定该组作为用户默认的表空间。表空间组有助于支持涉及排序的并行操作。为了将一个已有的表空间专门用于临时段，应当指定 create tablespace 或 alter tablespace 命令的 temporary 子句，如下面的程序清单所示：

```
alter tablespace TEMP temporary;
```

注意：
如果 TEMP 中存储了任何永久段(如表或索引)，则上述命令将失败。

为启用 TEMP 表空间来存储永久(非临时的)对象，可以使用 create tablespace 或 alter tablespace 命令的 permanent 子句，如下面的程序清单所示：

```
alter tablespace TEMP permanent;
```

DBA_TABLESPACES 数据字典视图中的 Contents 列显示表空间的状态，此状态为“TEMPORARY”或“PERMANENT”。

51.4.6　可用空间

表空间的可用区(Free Extent)是该表空间中相邻可用块的一个集合。表空间可以包含多个数据区和可用区。删除一个段时，它的区默认情况下未被释放，也没有被标记为可用。在 Oracle Database 10g 中，该默认行为是当该对象移动到用户的回收站时用于保留该空间。可以使用 flashback table to before drop 命令从回收站恢复该对象(参考第 30 章)。使用 purge 命令可以从回收站释放空间(参考附录 A)。

为了强制表空间合并其可用空间，可使用 alter tablespace 子句的 coalesce 子句，如下面的程序清单所示：

```
alter tablespace DATA coalesce;
```

此命令将强制 DATA 表空间中相邻的可用区合并为更大的可用区。此命令将像 SMON 那样，最多合并 8 个不同的区域。

注意：
alter tablespace 命令不合并数据区所分隔的可用区。

本地管理的表空间不需要相同程度的可用空间管理。由于本地管理的表空间的所有段可配置为具有统一的区大小，因此很容易重用已删除的区。

在分配新区时，除非没有别的办法，否则 Oracle 将不合并相邻的可用区。在字典管理的表中，这可能会导致使用表空间中后面的较大可用区，而不使用表空间中前面的较小可用区。

表空间中较小的可用区变成了“影响速度的垃圾”，因为它们没有足够大的空间可供使用。随着这种使用模式的延续，表空间中浪费的空间数量会不断增加。

在理想的数据库中，所有对象都以适当的大小创建，所有可用的空间总是存储在一起，形成一个等待使用的资源池。如果能在应用程序使用中避免动态空间分配，就既能排除对性能的影响，又能消除应用程序出故障的隐患。

51.4.7 设置数据库对象的大小

为数据库对象选择正确的空间分配参数很重要。开发人员在创建第一个数据库对象之前就应当开始估算空间的需求。然后，可以根据实际使用情况的统计信息，将空间需求细化。以下各小节将讨论表、索引和群集的空间估算方法。

1. 为什么要设置对象的大小

设置数据库对象的大小有 3 个原因：

- 预先分配数据库中的空间，使将来管理对象的空间需求的工作量最小化。
- 减少由于空间的过度分配所浪费的空间量。
- 增加已删除的可用空间被其他段重用的可能性。

可以通过以下几节提供的设置大小方法来实现这些目标。此方法基于 Oracle 给数据库对象分配空间的内部方法。此方法依赖于近似计算而不是精确计算，近似计算将明显地简化定义大小的过程并简化数据库的长期维护性。

2. 空间计算的黄金规则

应当保持数据库上的空间计算简单、通用和一致。相对于 Oracle 可能会忽略的极为详细的空间计算，有多种提高工作效率的方法。即使遵循最为严格的计算方法，仍然不能确定 Oracle 怎样把数据加载到表或索引中。

考虑具有 100 000 行的表上的索引。在测试过程中，排好顺序后将数据加载到表中，索引需要 920 个块。然后截断这个表和索引，用一种非排序的顺序重新加载数据，现在索引需要 1 370 个块。索引所需的空间增加了 49%，这是由 Oracle 的内部索引管理进程导致的。如果不相信这个 49%的可靠性，那么为什么还要花这么多时间来考虑大小的设置呢?

下一节将介绍怎样简化空间估计过程，使用户能有更多的精力完成更有用的 DBA 工作。不管是生成字典管理的表空间的默认储存值，还是生成本地管理的表空间的区大小，都应当遵循下面的过程。

3. 空间计算的基本规则

Oracle 在分配空间时，将遵循一组内部规则：

- Oracle 只分配整块，不分配部分块。
- Oracle 分配块组。
- Oracle 可以根据表空间中的可用空间分配较大的或较小的块组。

对于本地管理的表空间，区始终具有统一的大小，这一大小是在创建或生成自动分配算

法时指定的。我们的目的应当是使用 Oracle 的空间分配方法，而不是违反这一规则。如果使用统一的区大小，就可以在很大程度上将空间分配工作委托给 Oracle 来完成。

4. 区大小对性能的影响

减少表中区的数量不会直接提高性能。在某些情况下(如并行查询环境)，表中有多个区会减少 I/O 争用并提高性能。为便于管理，这些区需要正确地设置大小。

Oracle 以两种方式从表中读取数据，即通过 RowID(通常在后面紧跟一个索引访问)和通过全表扫描。如果数据通过 RowID 进行读取，则表中区的数量不是读性能的一个因素。Oracle 将从物理位置读取每个数据行(在 RowID 中指定)，并检索该数据。

为了减少区大小所造成的性能损失，必须选择以下策略之一：

(1) 创建明显大于操作系统的 I/O 缓冲区大小的区。如果这些区很大，那么即使区的大小不是 I/O 缓冲区大小的倍数，也几乎不需要其他的读操作。

(2) 设置 DB_FILE_MULTIBLOCK_READ_COUNT，以完全利用操作系统的 I/O 缓冲区大小。请注意，该值设置太高可能会使优化程序认为实际的全表扫描更有效，导致改变现有的执行计划。

(3) 如果必须创建较小的区，则应当选择区大小为操作系统的 I/O 缓冲区大小的倍数。

为了最大限度地重用字典管理的表空间中已删除的可用区，使用满足下面这一条件的一组区大小，即每个区大小将保持为每个较小区大小的整倍数。这条规则的最简单实现是创建双倍增加的区大小：1MB、2MB、4MB、8MB、16MB、32MB 等。为了减少需要管理的区大小的数量，可以不双倍增加区大小，而是 4 倍增加区大小：1MB、4MB、16MB 等。用这些值作为区大小，将消除 I/O 问题，增加重用已删除区的可能性。

5. 设置对象的大小

为有效地管理空间，需要做的是设置满足前几节所述条件的一组空间值。一旦空间分配完毕，就可以按表空间分隔它们了。例如：

```
create tablespace DATA_1M
datafile '/u01/oracle/VLDB/data_1m.dbf'
size 100M
extent management local uniform size 1M;

create tablespace DATA_4M
datafile '/u02/oracle/VLDB/data_4m.dbf'
size 400M
extent management local uniform size 4M;

create tablespace DATA_16M
datafile '/u03/oracle/VLDB/data_16m.dbf'
size 16000M
extent management local uniform size 16M;
```

在该示例中，创建了 3 个独立的 DATA 表空间，区大小分别为 1MB、4MB、16MB。如果需要创建一个 3MB 的表，都可以用 DATA_1M 中 3 个 1MB 的区来创建，或者用 DATA_4M 中一个 4MB 的区来创建。将增大为 10MB 的表可放置于 DATA_16M 中。

随着表大小的增大，默认的存储子句也应按照空间规则和区大小的标准相应增大。随后是 DATA_64M 后面，再接着是 DATA_256M 和 DATA_1G。应当在数据库上使用相同的区大小，以减轻整个数据库环境的空间管理工作。

随着区大小的增大，表空间上的区大小的分布一般会导致表类型的分离，较小的静态表将孤立在具有较小区大小的表空间中。较大的事务处理的表(或其分区)将隔离为较大区大小的表，这简化了以后的管理和调整活动。

注意：
数据库的块大小可以在表空间级改动。应当保证所选择的区大小考虑到数据库中使用的最大块大小。限制数据库中非标准大小块的使用将会简化跨数据库维护和大小设置过程。

51.5 监控撤消表空间

如第 29 章所述，可以使用系统管理的撤消(SMU)功能来简化事务管理。指定 UNDO_TABLESPACE、UNDO_RETENTION 以及 UNDO_MANAGEMENT 初始参数的值。

注意：
如果使用回滚段而不是 SMU，则参考附录 A 中的条目 create rollback segment、alter rollback segment 和 set transaction。

收集关于撤消表空间使用情况的统计信息的主要视图是 V$UNDOSTAT。V$UNDOSTAT 视图提供了保留数据的使用情况的直方图。Oracle 以 10 分钟为间隔，在 V$UNDOSTAT 视图的 UndoBlks 列中记录使用了的撤消块的总数，并在 TxnCount 列中记录所支持的事务的数量。可以使用来自 V$UNDOSTAT 视图的数据确定撤消表空间的大小设置对应用程序的事务的数量是否合适。Oracle 在 V$UNDOSTAT 视图中维护 144 行，以反映过去的统计信息。

V$UNDOSTAT 视图中其他令人感兴趣的列还有 MaxQuerylen，它以秒为单位记录执行时间最长的查询，撤消保留时间必须大于该值。如果 V$UNDOSTAT 视图中 NoSpaceErrCnt 列的值非零，则应该给撤消表空间增加空间，以避免以后出现与空间有关的错误。另一列 SsOlderRcnt 记录 ORA-01555("SnapShot too old")错误发生的次数。增加 UNDO_RETENTION 的值可以减少该错误发生的次数。

51.6 自动存储管理

可以使用自动存储管理(ASM)功能增强并简化存储管理工作。ASM 有许多显著的优点，包括：

- 可以轻松地将新磁盘增加到数据库可用的磁盘组；Oracle 将决定如何最好地使用这些磁盘，以分配 I/O 工作负载。

- 该 I/O 工作负载可动态分配到这些可用磁盘上，而无需 DBA 的干涉。
- 对于不同类型的文件，可以在不同层次维护该 I/O 带宽，通过数据库维护所有的带宽。
- 若硬件镜像未启用，则可启用软件镜像。

配置 ASM

为使用 ASM，必须配置 ASM 实例。使用 Oracle 提供的用于安装和配置 ASM 和数据库软件的工具，可参见第 2 章和第 3 章。

然后可以使用具有可用磁盘的 create diskgroup 命令。例如，可能有 4 个可用磁盘：

```
create diskgroup DISKGROUP1
    external redundancy
    disk '/u01', '/u02', '/u03', '/u04';
```

注意：
可以在磁盘名中使用通配符。

external redundancy 子句意味着磁盘组依靠硬件镜像。为使用软件镜像，使用 normal redundancy 子句，并指定通过 failgroup 子句构成镜像组的磁盘。在软件镜像中，Oracle 使用该镜像组作为一组磁盘，而不是作为该磁盘组磁盘的一对一的镜像。为了查看组成磁盘组的这些磁盘，可查询 V$ASM_DISK 动态视图。

既然磁盘组可用，就可以创建使用该磁盘组的表空间，如下面的示例所示：

```
create tablespace USERS2 datafile '+diskgroup1/users2_01'
    size 500m;
```

Oracle 将在指定的磁盘组中创建用于 USERS2 表空间的数据文件，并通过这些可用的磁盘来管理其 I/O 需求。如果需要向磁盘组中添加更多磁盘，则使用 alter diskgroup 命令：

```
alter diskgroup DISKGROUP1 add disk '/u05';
```

可以用 drop diskgroup 命令删除磁盘组。

ASM 实例被安装但未打开。一旦在服务器上创建了 ASM 实例，数据库实例就可以使用 ASM 实例来管理数据库相关文件的逻辑卷。

一般而言，应该为不同类型的文件(数据文件、重做日志文件、归档日志文件、Data Pump 导出文件，等等)创建不同的磁盘组。然后可以管理每种文件的物理布局，允许使用能最好地支持写操作的物理配置来存储写操作频繁日志文件，而数据文件存储在全力支持读操作的布局中。

51.7　段空间管理

除了在磁盘组级自动管理存储之外，Oracle 还能够管理段内空间。可以使用这些选项来回收段内未使用的空间、估算未来需要的空间，并在线重新组织表。

如果应用程序频繁从表中删除行，则表的块仅可以部分地填充行。可以使用 alter table 命令手工压缩该表内的空间。首先启用该表内行的移动功能：

```
alter table BOOKSHELF_CHECKOUT enable row movement;
```

然后让 Oracle 移动表内的行，以压缩块内行分配的空间。

```
alter table BOOKSHELF_CHECKOUT shrink space compact;
```

shrink space compact 操作会重新组织该表的行，使这些行存储得更紧凑。但是，不影响表的高水位标志(已经写入的行的最大块号)。因为所有全表扫描都将读取所有的块，直到高水位标志，所以应该调整该值。可以使用 alter table 命令的 shrink space 子句调整高水位标志。例如：

```
alter table BOOKSHELF_CHECKOUT shrink space;
```

可以通过 shrink space cascade 选项级联本更改对表的索引的影响。

要预测空间的使用情况，可以使用 DBMS_SPACE 程序包的 OBJECT_GROWTH_TREND 函数。传递的参数有模式名、对象名和对象类型，如下面的程序清单所示。应当作为具有 SYSDBA 权限的用户，来运行本程序。

```
select * from
table(DBMS_SPACE.OBJECT_GROWTH_TREND
('PRACTICE','BOOKSHELF_CHECKOUT','TABLE'));
```

该输出将显示对象过去的空间使用情况，以及对象的未来空间需求。结果是基于自动工作负载存储库(Automated Workload Repository，AWR)收集的统计信息。使用新的后台进程 MMON，AWR 收集并存储支持动态管理和优化的数据库统计信息。通过 AWR 收集的这些信息，Oracle 可以计划未来空间的使用情况并识别优化机会。

51.8 移动表空间

可移动表空间是可以从一个数据库“拔出”，并“插入”到另一个数据库的表空间。要成为可移动表空间，表空间或表空间集必须是独立的。表空间集不能包含任何引用其他表空间中对象的对象。因此，如果移动包含索引的表空间，就必须作为相同可移动表空间集的一部分而移动包含该索引的基表的表空间。表空间之间的对象组织和分布得越好，就越容易生成用于移动的表空间的独立集。

要移动表空间，需要生成表空间集，把该表空间集复制或移动到新的数据库中，并将该集插入新的数据库中。数据库必须位于相同的操作系统，具有相同的 Oracle 版本和字符集。

51.8.1 生成可移动表空间集

可移动表空间集包含所有用于被移动表空间的数据文件，以及为这些表空间导出的元数据。可视情况选择是否将参考完整性约束包括为可移动表空间集的一部分。如果选择使用参

考完整性约束，则可移动表空间集将增大，用来包括维护主键关系所需要的表。因为多个数据库中可以具有相同的代码表，所以参考完整性约束为可选项。例如，可以计划从测试数据库中将表空间移动到产品数据库中。如果在测试数据库中具有一个 COUNTRY 表，则在产品数据库中可能已有一个相同的 COUNTRY 表。既然两个数据库中的代码表相同，就不必移动参考完整性约束了。可以移动包含非代码的表空间；表空间一旦移动，然后就重新启用目标数据库中的参考完整性约束，以简化可移动表空间集的创建。

要决定表空间集是否独立，可以执行 DBMS_TTS 包的 TRANSPORT_SET_CHECK 过程。本过程接受两个输入参数：表空间集和一个布尔标志(如果要考虑参考完整性约束，则设置为 TRUE)。在以下示例中，对于 AGG_DATA 和 AGG_INDEXES 表空间的组合，不考虑参考完整性约束：

```
execute DBMS_TTS.TRANSPORT_SET_CHECK('AGG_DATA,AGG_INDEXES',FALSE);
```

注意：
不能移动 SYSTEM、SYSAUX 或临时表空间。

如果在指定的集中存在任何违反独立性的情况，则 Oracle 将填充 TRANSPORT_SET_VIOLATIONS 数据字典视图。否则该视图为空。

一旦选择了独立的表空间集，就要使表空间为只读，如下所示：

```
alter tablespace AGG_DATA read only;
alter tablespace AGG_INDEXES read only;
```

接下来，使用 TRANSPORT_TABLESPACES 和 TABLESPACES 导出参数导出表空间的元数据。

```
expdp TRANSPORT_TABLESPACES=Y TABLESPACES=(AGG_DATA,AGG_INDEXES)
CONSTRAINTS=N GRANTS=Y TRIGGERS=N
```

如上面的示例所示，可以指定是否同表空间的元数据一起导出触发器、约束和授权。还应当注意在可移动表空间集中拥有对象的账户名。现在，可以将表空间的数据文件复制到不同的区域。如果需要，可以在当前数据库中使表空间恢复为读写模式。在生成可移动表空间集之后，可以将其文件(包括导出的内容)移动到目标数据库可访问的区域。

51.8.2　插入可移动表空间集

一旦将可移动表空间集移动至目标数据库可访问的区域，就可以将该集插入目标数据库中。首先，使用 Import 导入已经导出的元数据。

```
impdp TRANSPORT_TABLESPACES=Y DATAFILES=(agg_data.dbf, agg_indexes.dbf)
```

在导入过程中，可以指定作为可移动表空间集一部分的数据文件。可以选择性地指定表空间(通过 TABLESPACE 参数)和对象所有者(通过 OWNERS 参数)。

在导入完成之后，可移动表空间集中所有的表空间都为只读模式。可以在目标数据库中使用 alter tablespace read write 命令，以将新的表空间设置为读写模式。

```
alter tablespace AGG_DATA read write;
alter tablespace AGG_INDEXES read write;
```

注意不能更改正正在移动的对象的所有权关系。

可移动表空间支持大型数据集的极快速移动。在数据仓库中，可以使用可移动表空间把聚合从核心仓库发布到数据集市，或者把聚合从数据集市发布到全局数据仓库。任何只读数据都可以快速分布到多个数据库，可以发送数据文件和导出的元数据，而不发送 SQL 脚本。这个已修改数据的移动过程可大大简化管理远程数据库、远程数据集市和大型数据移动操作的过程。

在 Oracle 11g 中，可以移动分区表的一个分区。当在目标一侧导入分区的元数据时，使用 Data Pump Import 的 DEPARTITION 选项导入分区，并作为新环境中一个单独的表。关于 Data Pump 的详细信息请参见第 24 章。

注意：
源操作系统和目标操作系统可以不相同，在本过程中支持末端转换。数据文件必须通过 RMAN 的 convert tablespace …to platform 命令进行转换。

51.9 进行备份

与本章其他部分一样，本节只是一个概述，而不进行深入讨论。备份和恢复比较复杂，所有备份和恢复的方法在某个生产环境中实现之前，都应该进行全面的测试和试验。本节的目的是帮助开发人员理解 Oracle 的功能，以及备份决策对恢复工作和可用性的影响。在某个生产环境中使用备份和恢复方法之前，应当参考有关该内容的专门书籍，其中包括 Oracle 的文档。

Oracle 提供的备份方法可分为以下几类：

- 使用 Data Pump Export 和 Data Pump Import 的逻辑备份
- 物理文件系统备份：脱机备份和联机备份
- 通过恢复管理器(Recovery Manager，RMAN)进行的增量物理文件系统备份

以下各节将介绍每种方法及其功能。

51.9.1 Data Pump Export 和 Data Pump Import

Data Pump 是一个基于服务器的数据抽取和导入实用程序。它对原来的 Import 和 Export 实用程序在结构上和功能上都有很大的增强。Data Pump 允许停止并重新启动作业，查看正在运行的作业的状态，以及限制导出和导入的数据。在 Oracle 11g 中，在写输出数据时可以对输出数据进行压缩、加密和更改(通过 REMAP_DATA 选项)，还可以将分区导入为单独的表。

注意：
虽然 Data Pump 可以使用由原始的 Export 实用程序生成的文件，但原始的 Import 实用程序不能使用从 Data Pump Export 生成的文件。

因为 Data Pump 作为服务器进程运行，所以给用户带来的好处是多方面的。启动作业的客户端进程可以断开与作业的连接并在以后重新连接。与 Export/Imprt 相比，Data Pump 的性能得到增强，这是因为数据不再必须由客户端程序来处理。Data Pump 的抽取和加载可以并行进行，从而进一步增强了性能。

Data Pump 执行数据的逻辑备份，它从正在运行的数据库的表中查询数据。在恢复期间，使用 Data Pump 可以将数据库(或选择的表)还原到导出启动之前的状态。通过 Data Pump 还原的数据以后不能前途到之后的状态——它是执行备份时那一刻数据的固定快照。

关于 Data Pump 及其功能的详细描述参见第 24 章。

51.9.2　脱机备份

脱机备份(有时称为冷备份)是数据库文件的物理备份，是在使用 shutdown normal、shutdown immediate 或 shutdown transactional 关闭数据库后进行的。在数据库关闭时，数据库经常使用的每个文件都进行了备份。这些文件提供了数据库在关闭那一刻的完整映像。

在冷备份中，应当备份下列文件：

- 所有数据文件
- 所有控制文件
- 所有联机重做日志

注意：
也应该备份数据库初始化参数文件，特别是在备份将用于灾难恢复过程时。

在数据库关闭时对这些文件进行备份，将提供该数据库在关闭时刻的一个完整映像。这些文件的完整集合可在以后从备份中检索到，并且该数据库可以运行。除非执行联机备份(本章后面将讨论)，否则在数据库打开时，不允许执行数据库的文件系统备份。

注意：
对联机重做日志文件的备份要小心。在恢复过程中，需要确保不意外地重写恢复所需要的任何已有的联机重做日志文件。

在理想情况下，所有数据文件都位于每个设备的同级目录中。例如，所有数据库文件都存储在每个设备的/oracle 目录下一个专门的示例子目录中(如/db01/oracle/MYDB)。这样的目录应当包含数据库所有的数据文件、重做日志文件和控制文件。唯一不在此位置中的可选择进行脱机备份的文件是产品的初始化参数文件，它应该在 Oracle 软件基本目录下的/app/oracle/admin/INSTANCE_NAME/pfile 子目录中，或者在 Oracle 软件主目录下的/dbs 目录中。

如果使用上例中的目录结构，则备份命令将大大简化，因为能够在文件名中使用通配符。在关闭数据库之后，可将文件备份到备份目的区域(磁带或一个独立的磁盘区域)。

由于脱机备份涉及修改数据库的可用性，因此脱机备份通常安排在夜间进行。

脱机备份非常可靠。但为了减少它们对数据库可用性的影响，应使用联机备份。正如51.9.3节所描述的那样，联机备份使用Oracle的ARCHIVELOG模式，可以在数据库使用过程中进行一致性的文件系统备份。

51.9.3 联机备份

可以对正以ARCHIVELOG模式运行的任何数据库进行联机备份。在此模式中，将归档联机重做日志，创建数据库内所有事务的一个完整日志。

Oracle以循环方式写入联机重做日志文件。在写满第一个日志文件后，它开始写第二个日志文件，直到写满为止，然后开始写第3个日志文件。一旦最后一个联机重做日志文件也写满，LGWR(Log Writer)后台进程将开始重写第一个重做日志文件的内容。

当Oracle以ARCHIVELOG模式运行时，ARC0-ARCt(Archiver)后台进程在开始重写每个重做日志文件前，对其进行复制。这些归档的重做日志文件通常被写入一个磁盘设备中，也可以直接写入磁带设备，但这要涉及很多的操作。

可以在数据库打开时，执行数据库的文件系统备份，只要该数据库正在以ARCHIVELOG模式运行即可。联机备份要将每个表空间(或者整个数据库)设置为备份状态，备份其相关的数据文件，然后将表空间还原到其正常状态。

注意：
当使用Oracle提供的RMAN实用程序时，不必将每个表空间都置为备份状态。

数据库可以从联机备份中完全恢复，并且可以通过归档重做日志，回滚到任意一个时刻。然后当打开数据库时，数据库中在该时刻已提交的任何事务都将还原，并且任何未提交的事务都被回滚。

当打开数据库时，将会备份以下文件，具体取决于使用的RMAN命令：

- 所有数据文件
- 所有归档重做日志文件
- 一个控制文件(通过alter database命令)

联机备份过程非常有用，其原因有两个。首先，它们提供完整的时间点的恢复。不以ARCHIVELOG模式运行的数据库只能恢复到备份发生的时刻。其次，它们允许数据库在文件系统备份过程中保持打开状态。因此，即使由于用户的需要不能关闭的数据库也仍然能够进行文件系统备份。

1. 开始

为了利用ARCHIVELOG的功能，数据库必须首先处于ARCHIVELOG模式。以下程序清单说明了以ARCHIVELOG方式设置数据库的步骤。运行SQL*Plus并安装数据库(假设在

这些示例中其名称为“mydb”），然后进行如下更改：

```
connect system/manager as sysdba
startup mount mydb;
alter database archivelog;
alter database open;
```

以下命令将从 SQL*Plus 中显示数据库的当前 ARCHIVELOG 状态：

```
archive log list
```

注意：

为查看当前活动的联机重做日志及其序列号，可以查询 V$LOG 动态视图。

为将数据库改回到 NOARCHIVELOG 方式，可以在关闭数据库后使用下面的这组命令：

```
connect system/manager as sysdba
startup mount mydb;
alter database noarchivelog;
alter database open;
```

处于 ARCHIVELOG 模式的数据库将一直保持此模式，直到将它设置为 NOARCHIVELOG 模式为止。归档重做日志文件的位置由数据库参数文件的设置确定。需要留意的参数如下(这里使用的是样本值)：

```
log_archive_dest_1 = /db01/oracle/arch/CC1/arch
log_archive_dest_state_1 = ENABLE
log_archive_format = arch%s.arc
```

在本例中，归档重做日志文件被写入目录/db01/oracle/arch/CCl 中。归档重做日志文件都用字母“arch”开始，后跟一串数字。例如，归档重做日志文件的目录可包含以下文件：

```
arch_170.arc
arch_171.arc
arch_172.arc
```

每个文件都包含来自单个联机重做日志的数据。它们按创建的顺序编号。归档重做日志文件的大小可以变化，但不能超过联机重做日志文件的大小。在 LOG_ARCHIVE_FORMAT 设置中，可用的变量如表 51-1 所示。

表 51-1　在 LOG_ARCHIVE_FORMAT 设置中可用的变量

变　量	说　明
%s	日志序列号
%S	日志序列号，零填充
%t	线程号
%T	线程号，零填充
%a	激活 ID
%d	数据库 ID
%r	构建确保名称唯一的重设日志 ID，用于跨越数据库的多个示例的归档日志文件

如果归档重做日志文件的目的目录的空间耗尽，则 ARCH 将停止处理联机重做日志的数据并且暂停数据库。可以通过把更多的空间添加到归档重做日志文件的目的盘中，或者通过备份归档重做日志文件，然后从该目录中删除它们来处理这种情况。也可以通过 LOG_ARCHIVE_DEST_n 参数指定另一个位置，只有在初始位置满时才使用此位置。

注意：
除非已经备份了归档重做日志文件并验证可以成功地还原它们，否则绝不能删除它们。

除非执行了本节所示的 alter database archivelog 命令，否则数据库不会处于 ARCHIVELOG 模式。一旦数据库处于 ARCHIVELOG 模式，它就在后面的数据库关闭和启动中保持此模式，直到使用 alter database noarchivelog 命令显式地将它设置为 NOARCHIVELOG 模式为止。对于要启动的归档，LOG_ARCHIVE_MAX_PROCESSES 参数必须设置为大于 0 的值，默认值是 2。

注意：
归档日志目的区域不同于闪回恢复区域。闪回恢复区域支持 flashback database 命令。关于该命令及其相关参数的详细介绍，请参阅第 30 章。

2. 使用 O/S 实用程序执行联机数据库备份

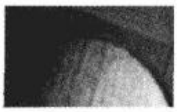
一旦数据库以 ARCHIVELOG 模式运行，就可以在它打开并对用户可用时，进行备份。该功能在保证要归档数据库的持续可用性的同时，也保证了数据库的可恢复性。

尽管联机备份可以在正常的工作时间进行，但它们还是应当尽量安排在用户活动最少的时候进行，这样做有几个原因。首先，联机备份要使用操作系统命令来备份物理文件，并且这些命令将使用系统中可用的 I/O 资源(这将影响交互式用户的系统性能)。其次，当备份表空间时，向归档重做日志文件写入事务的方式将发生改变。在使一个表空间进入“备份”模式时，DBWR 进程把属于表空间的任何文件的缓冲区缓存中的所有块写回磁盘。在把这些块读回内存，并进行更改时，它们将被复制到第一次对其进行更改的日志缓冲区。只要它们保存在缓冲区缓存中，就不用把它们重新复制到联机重做日志文件中。这将会使用归档重做日志文件目的目录中的大量空间。

热备份的命令文件有 3 部分：

(1) 数据文件的备份，依次由以下部分组成：
   A. 设置表空间为备份状态
   B. 备份表空间的数据文件
   C. 还原表空间为正常状态
(2) 归档重做日志文件的备份，它包括以下部分：
   A. 记录哪些文件在归档重做日志目的目录中
   B. 备份归档重做日志文件，然后(选择性地)删除或压缩这些文件

(3) 通过 alter database backup controlfile 命令，备份控制文件。



注意：
此联机备份过程通过 RMAN 实用程序自动进行。

可以创建一个执行备份的脚本文件。但推荐使用 RMAN 实用程序。

当备份数据文件时，可以直接将其备份到磁带或磁盘上。如果有足够可用的磁盘空间，则选择磁盘，因为这样将大大减少完成备份过程所需要的时间。

51.9.4 Recovery Manager

Oracle database 提供了名为 RMAN 的 Recovery Manager 实用程序，从而使用户能够用命令行模式，或者用 Oracle Enterprise Manager 内的 Recovery Manager，自动地备份和恢复数据库。可以采用这两种方法中的任意一种来备份、还原和恢复数据库文件。RMAN 最大的好处之一是不将表空间置为热备份模式，因此，改变了的数据块不被复制到联机重做日志文件中，复制到联机重做日志文件会影响系统 I/O 资源。

Recovery Manager 通过 Recovery Catalog 或通过将所需信息放置在要备份的数据库的控制文件中，来跟踪备份。Recovery Manager 添加了在其他 Oracle 备份实用程序中所没有的新备份功能。Recovery Manager 中有 4 个组件，分别是 RMAN 可执行程序、一个或多个目标数据库、恢复目录数据库和介质管理软件。必须具有的组件为 RMAN 可执行程序和目标数据库。

Recovery Manager 允许执行数据文件的增量物理备份。在完全(称为 0 级)数据文件备份过程中，曾经用于数据文件的所有块都将进行备份。在累积(1 级)数据文件备份中，自最后一次完全数据文件备份以来使用的所有块都进行备份。而增量(2 级)数据文件备份只备份自最近一次累积备份或完全备份以来更改过的那些块。可以定义增量备份所使用的级别。

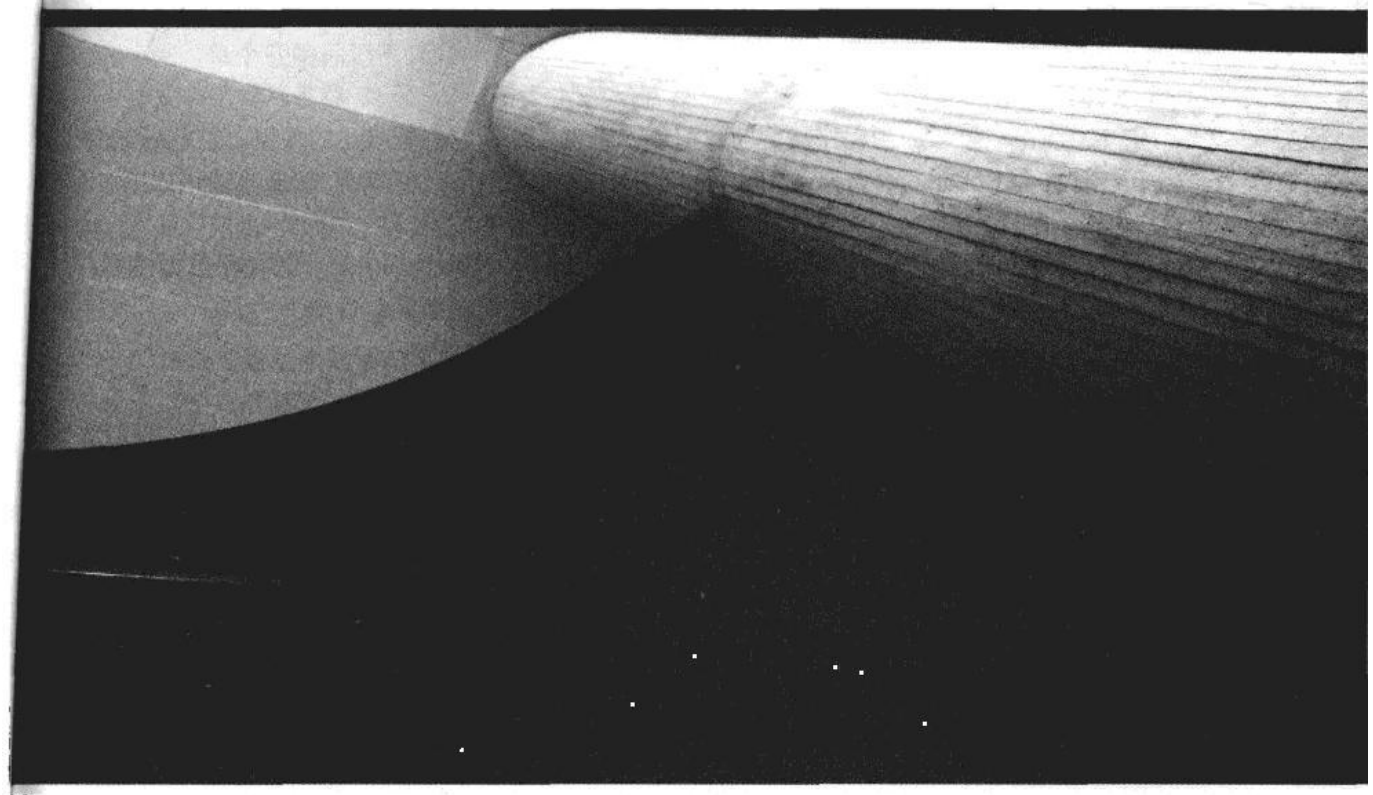
能够执行数据文件增量备份和累积备份大大改善了备份的性能。最大的性能改进体现在非常庞大的数据库中(一个大的表空间中只有一个小的子集发生了变化)。使用传统的备份方法，需要备份该表空间中的所有数据文件。而使用 Recovery Manager，只要备份自上次备份以来变化了的块即可。

在使用 Recovery Manager 进行数据库恢复时，需要知道哪些文件是当前的，哪些文件是还原的，以及打算使用的备份方法。如果使用恢复目录，Recovery Manager 就在一个 Oracle 数据库中存储其信息目录，而且需要备份该数据库，否则会丢失信息的整个备份和恢复目录。

Recovery Manager 只由 DBA 使用，DBA 要针对环境确定 Recovery Manager 的体系结构(例如，确定恢复目录的位置)。应当与 DBA 协作，以理解所使用的恢复选项以及数据库可用性和恢复时间的含义。

51.10 展望

本章给出了DBA每天要处理的任务的高级概述。除了这里所介绍的内容外，DBA还要监控数据库、优化数据库、安装软件、维护数据库的连接等很多其他任务。如果对这些任务感兴趣，就请参阅Oracle文档集或*Oracle DBA Handbook*。开发人员越了解数据库管理，就越能更好地利用数据库内在的功能构建应用程序。



第 52 章

Oracle 中的 XML 指南

XML(eXtensible Markup Language，可扩展标记语言)是描述分层数据的一种标准化语法。XML 是结构化文档和数据的一种通用格式，而不是程序设计语言。熟悉 HTML(Hypertext Markup Language，超文本标记语言)的人应该熟悉基于标记的语法，这种语法提供了处理复杂数据的灵活性。HTML 告诉 Web 浏览器如何表示数据，而 XML 告诉应用程序数据的含义。

XML 非常适合于解决异构系统之间的数据交换问题。使用它，可以很容易地访问、转换和存储在数据库中驻留的数据。本章将概述 Oracle 中可用的 XML 访问方法。XML 及其工具集还在不断发展，最新信息请登录 http://www.w3.org。

52.1 文档类型定义、元素及属性

在 XML 中，使用应用程序特定的标记或“元素”括住它们所描述的数据。这些标记的语法在一种称为 Document Type Definition(DTD，文档类型定义)的特殊 XML 文档中定义。DTD 定义有效的 XML 文档的结构。这种结构严格分层。

每个 XML 文档都是实例化的信息集(infoset)——组成文档及其信息项的抽象数据模型。信息项主要包括元素、属性和内容。例如，一个 XML 信息集可能包含如下内容：

```
<book>
  <title>MY LEDGER</title>
  <chapter num="1">
    <title>Beginning</title>
    <text>&chapter1;</text>
  </chapter>
</book>
```

在本示例中，标记<book>和</book>定义了信息集的起点和终点。此 XML 文档描述了一本名为“MY LEDGER”的书。这本书有很多章，第一章的标题为“Beginning”。该章的正文未存储在此 XML 文档内，而是创建了一个指向外部文件的指针。

“title”不仅是文件中的一个标记，还是一个元素。它有内容，在此例中为“MY LEDGER”。如本例所示，可通过在标记(用于标题)之间或在标记内(如章节号设置)提供值，来设置元素的内容。元素可包含内容、子元素，或两者都包含，也可以使两者内容都为空。对于数据交换，应当避免创建既包含内容又包含子元素的元素。为创建空元素，可使用如下所示的标记格式：

```
<name/>
```

注意：
元素名区分大小写。

每个元素只有一个父元素，而一个父元素可以有任意数目的子元素。XML 的严格分层特性意味着：如果一个元素在另一元素内开始(如“chapter”在“book”内)，则它的结束标记也必须位于父元素的结束标记之前。

正如 book 示例所示，元素具有属性。例如，chapter 元素有一个名为 num 的属性。属性值括在双引号内：

```
<chapter num="1">
```

注意：
与元素名一样，属性名也区分大小写。

一般来说，应用程序使用特定项的元素属性(如章节号)和用于批处理数据的元素内容(本例中为章的正文)。

如果 XML 文档满足下列条件，它就是格式良好的文档：

- 每个开始标记都具有一个相应的结束标记
- 元素不重叠
- 有一个根元素
- 属性值始终位于引号内
- 元素中不能有两个属性同名
- 注释和处理指令不出现在标记内
- 在元素或属性的字符数据中没有未转义的“<”或“&”符号

这些都是基本的语法规则。如果一个 XML 文档符合这些规则，它可能仍然无效。为了使 XML 文档有效，它必须符合文档类型定义(DTD)的规则。DTD 是描述什么样的元素和实体可出现在 XML 文档内，以及每个元素的内容和属性是什么的正规方法。

例如，考虑下面的 XML 文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CustomerList SYSTEM "Example1.dtd">
<CustomerList>
  <Customer preferredCustomer="">
    <ID>12345</ID>
    <Address>
      <Name>TUSC</Name>
      <Street>377 E BUTTERFIELD RD</Street>
      <City>LOMBARD</City>
      <State>IL</State>
      <Country>USA</Country>
      <ZIP_CODE>60148</ZIP_CODE>
      <PHONE>630-960-2909</PHONE>
    </Address>
    <CreditRating>EXCELLENT</CreditRating>
    <SalesRepID></SalesRepID>
    <RegionID>5</RegionID>
    <Comments>THIS IS A TEST XML DCCUMENT</Comments>
    <ShippingMethod>M</ShippingMethod>
  </Customer>
</CustomerList>
```

第 1 行为一个 XML 声明，以“<?”开始，以“?>”结束：

```
<?xml version="1.0" encoding="UTF-8"?>
```

version 属性现在应当设置为 1.0。encoding 属性是可选的，其默认值为 UTF-8。还可以设置第 3 个属性 standalone。在本例中，XML 文档不是独立的文档，它有一个相关的 DTD，如第 2 行所示：

```
<!DOCTYPE CustomerList SYSTEM "Example1.dtd">
```

该示例的相关 DTD 文件 Example1.dtd 如下面的程序清单所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT CustomerList (Customer*)>
```

```
<!ELEMENT Customer (ID, Address, CreditRating*, SalesRepID*,
                    RegionID, Comments*, ShippingMethod)>
<!ATTLIST Customer preferredCustomer CDATA #IMPLIED>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT Address (Name, Street*, City, County,
                   State, Country, ZIP_CODE, PHONE)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT County (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
<!ELEMENT ZIP_CODE (#PCDATA)>
<!ELEMENT PHONE (#PCDATA)>
<!ELEMENT CreditRating (#PCDATA)>
<!ELEMENT SalesRepID (#PCDATA)>
<!ELEMENT RegionID (#PCDATA)>
<!ELEMENT Comments (#PCDATA)>
<!ELEMENT ShippingMethod (#PCDATA)>
```

DTD 类似于外部表的控制文件规范。它告诉应用程序在 XML 文档中可以找到的内容(元素名和元素层次)。一个 XML 文档只能有一个 DTD。

元素声明的语法为：

```
<!ELEMENT element_name (content_model)>
```

其中，content_model 指定元素可以或必须具有的子元素及其顺序。最简单的内容模型为 #PCDATA，它表示相应的元素只能包含分析过的字符数据。例如：

```
<!ELEMENT Name (#PCDATA)>
```

在此示例中，CustomerList 元素包含零个或多个“Customer”元素，如*后缀所示：

```
<!ELEMENT CustomerList (Customer*)>
```

允许使用的后缀如下：

- * 允许出现零次或多次。
- + 允许出现一次或多次。
- ? 允许出现零次或一次。

正如 Address 元素规范所示，可以指定用逗号分隔的多个元素的序列：

```
<!ELEMENT Address (Name, Street*, City, County,
                   State, Country, ZIP_CODE, PHONE)>
```

可在规范内提供多个选项，用竖杠(|)字符分隔。例如，二维图形上的一个点可利用其横坐标和纵坐标或距原点的距离和角度，来进行定位：

```
<!ELEMENT Point ((x,y) | (distance, angle))>
```

对于实际的应用程序，DTD 可能会很快变得非常复杂。DTD 内可能会有属性列表、外部引用、实体属性、ID、条件包含以及其他结构。关于 DTD 结构及选项的详细评述，请参

阅 XML 文档。

注意：
在创建一个标准业务操作的 DTD 前，应当查看一下是否已经有符合要求的 DTD。http://www.xml.org/xml/registry.jsp 等 Web 站点提供了 DTD 的许多示例。它们为行业标准 DTD 及其组件的文档提供有价值的资源。

52.2　XML 模式

XML 模式(XML Schema)规范是 DTD 的替代物。它包括数据输入，这在使用关系数据库时特别有用。尽管 DTD 不可扩展(它们仅仅是控制文件)，但 XML 模式在下列方式中是可扩展的：

- 部分可在其他模式中重用。
- 复杂结构可在其他模式中重用。
- 可基于已有数据类型创建派生数据类型。
- 一个文档实例可引用多种模式。

在 XML 模式内，元素用 xsd:element 来声明，如下面的程序清单所示：

```
<xsd:element name="Date">
```

元素的数据类型可以通过 type 属性来声明：

```
<xsd:element name="Date" type="date">
```

可以通过 minOccurs 和 maxOccurs 属性，来指定元素出现的最少和最多次数：

```
<xsd:element name="Date" type="date" minOccurs="0" maxOccurs="1">
```

内置的 XML 模式类型包括基本类型(如串、布尔值、数值、浮点数、日期、时间和日期时间)以及派生类型(如 normalizedString、整型、令牌和语言等)。类型的完整列表可在 http://www.w3.org/TR/xmlschema-2/中找到。

可以创建和命名复杂类型，就像在 Oracle 中可以创建抽象数据类型一样。下面的代码可以创建“Address”类型：

```
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="Street1"/>
    <xsd:element name="Street2"/>
    <xsd:element name="City"/>
    <xsd:element name="State"/>
    <xsd:element name="Zip"/>
  </xsd:sequence>
</xsd:complexType>
```

之后，可以在定义元素时，引用“Address”复杂类型：

```
<xsd:element name="mailingAddress" type="Address"/>
<xsd:element name="billingAddress" type="Address"/>
```

如果为单个元素声明内容模型，就可以使用匿名类型，如下面的程序清单所示：

```
<xsd:element name="Name">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="First"/>
      <xsd:element name="MiddleInitial"/>
      <xsd:element name="Last"/>
    </xsd:sequence>
  </xsd:complexType
</xsd:element>
```

属性用 xsd:attribute 声明，如下例所示：

```
<xsd:attribute name="paid" type="boolean"/>
```

可以在复杂类型内创建属性：

```
<xsd:element name="Invoice">
  <xsd:complexType>
    <xsd:attribute name="paid" use="optional" default="True"/>
  </xsd:complexType>
</xsd:element>
```

除了创建属性外，还可以创建属性组和模型组。模型组定义是组或元素定义，为重用而命名。“组”元素必须为顶层模式组件(“模式”元素的子元素)。例如，下面的程序清单显示 MealOptions 组(由“FlightDinner”元素引用)的创建过程。

```
<xsd:element name="FlightDinner">
  <xsd:sequence>
    <xsd:element ref="Name"/>
    <xsd:group ref="MealOptions"/>
  </xsd:sequence>
</xsd:element>

<xsd:group name="MealOptions">
  <xsd:choice>
    <xsd:element name="Fish" type="Meal"/>
    <xsd:element name="Meat" type="Meal"/>
    <xsd:element name="Vegetarian" type="Meal"/>
  </xsd:choice>
</xsd:group>
```

如果元素常常使用一组相同的属性，就可以将这些属性组织在一起。这类似于第 39 章所描述的方法，即将常用的属性组合，形成新的对象类型。例如，下面“Product”元素的定义使用了一个名为 productDetails 的属性组。其他元素可重用这个相同的属性组，以提高实现数据的标准化表示的能力。

```
<xsd:element name="Product">
  <xsd:complexType>
    <xsd:attributeGroup ref="productDetails"/>
```

```
  </xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="productDetails">
  <xsd:attribute name="productID" use="required" type="xsd:ID"/>
  <xsd:attribute name="name" use="required" type="xsd:string"/>
  <xsd:attribute name="description" use="required" type="xsd:string"/>
  <xsd:attribute name="unit" type="xsd:positiveInteger"/>
  <xsd:attribute name="price" use="required" type="xsd:decimal"/>
  <xsd:attribute name="stock" type="xsd:integer"/>
</xsd:attributeGroup>
```

可以使用 xsd:unque，在一个文档实例内实现唯一性：

```
<xsd:unique name="uniqueProductID">
  <xsd:selector xpath="tc:Product/"/>
  <xsd:field xpath="tc:@productID"/>
</xsd:unique>
```

声明唯一性的层次与“Product”元素的复杂类型定义的层次相同。“tc:”名称空间用于正在创建的模式，并在 xsd:schema 元素内声明。xsd:selector 的值指定约束的元素(“Product”元素)，xsd:field 指定约束的属性(productID)。

可以使用 xsd:key 和 xsd:keyref 创建元素之间的引用。例如，可以创建一个“Customer”元素，然后在多个“Invoice”元素中引用它：

```
<xsd:key name="customerID">
  <xsd:selector xpath="tc:SalesData/"/>
  <xsd:field xpath="tc:Customer/@CustID"/>
</xsd:key>

<xsd:keyref name="item" refer="customerID">
  <xsd:selector xpath="tc:SalesData/"/>
  <xsd:field xpath="tc:Invoice/@CustID"/>
</xsd:keyref>
```

为允许一个元素为 NULL，可以使用 nillable 属性：

```
<xsd:element name="FlightDinner" nillable="true"/>
```

52.3　使用 XSU 选择、插入、更新和删除 XML 值

可以使用 Oracle 的 XML-SQL 实用程序(XSU)将 SQL 查询的结果转换为 XML。XSU 支持 DTD 的动态生成，并将 XML 插入数据库表中。还可以使用 XSU 更新或删除数据库对象中的行。

可以通过 Java 和 PL/SQL API(应用程序接口)访问 XSU。XSU Java 类是 Oracle 核心软件的组成部分。PL/SQL API 是一种包装器，它把 Java 类发布到 PL/SQL。因为 XSU 类存储在数据库中，所以可以编写新的存储过程来直接访问 XSU 的 Java 类。由于 PL/SQL API 可用，因此可直接通过 SQL 访问 XSU 的功能。

例如，可以使用 XSU PL/SQL 过程创建 RATING 表的 XML 版本。下面的程序清单创建

一个名为 PRINTCLOBOUT 的过程，它将作为数据输出过程的一部分而被调用。相应的数据将作为 CLOB 读入。

```
create or replace procedure PRINTCLOBOUT(result IN OUT NOCOPY CLOB)
  is
xmlstr varchar2(32767);
line varchar2(2000);
begin
  xmlstr := dbms_lob.SUBSTR(result,32767);
  loop
    exit when xmlstr is null;
    line := substr(xmlstr,1,instr(xmlstr,chr(10))-1);
    dbms_output.put_line('| '||line);
    xmlstr := substr(xmlstr,instr(xmlstr,chr(10))+1);
  end loop;
end;
/
```

既然 PRINTCLOBOUT 过程已经存在，就执行下面的匿名 PL/SQL 块。由于该块查询 RATING 表，因此该表必须可从用户的模式中进行访问。

注意：
执行本 PL/SQL 块之前，必须启动服务器输出(set serveroutput on)。

```
declare
  queryCtx DBMS_XMLquery.ctxType;
  result CLOB;
begin
  -- set up the query context...
  queryCtx := DBMS_XMLQuery.newContext('select * from rating');
  -- get the result...
  result := DBMS_XMLQuery.getXML(queryCtx);
  -- Now you can use the result to put it in tables/send as messages...
  printClobOut(result);
  DBMS_XMLQuery.closeContext(queryCtx); -- you must close the query...
end;
/
```

结果得到 RATING 表，使用 XML 标记进行了格式化：

```
| <?xml version = '1.0'?>
| <ROWSET>
|   <ROW num="1">
|     <RATING>1</RATING>
|     <RATINGDESCRIPTION>ENTERTAINMENT</RATINGDESCRIPTION>
|   </ROW>
|   <ROW num="2">
|     <RATING>2</RATING>
|     <RATINGDESCRIPTION>BACKGROUND INFORMATION</RATINGDESCRIPTION>
|   </ROW>
|   <ROW num="3">
|     <RATING>3</RATING>
|     <RATINGDESCRIPTION>RECOMMENDED</RATINGDESCRIPTION>
|   </ROW>
```

```
|   <ROW num="4">
|     <RATING>4</RATING>
|     <RATINGDESCRIPTION>STRONGLY RECOMMENDED</RATINGDESCRIPTION>
|   </ROW>
|   <ROW num="5">
|     <RATING>5</RATING>
|     <RATINGDESCRIPTION>REQUIRED READING</RATINGDESCRIPTION>
|   </ROW>
| </ROWSET>

PL/SQL procedure successfully completed.
```

52.3.1　使用 XSU 进行插入、更新和删除

为了将文档插入表或视图中，可以使用 DBMS_XMLSave 程序包。XSU 将分析该文档，并创建一条将值插入表或视图的所有列中的 insert 语句。缺少的元素可视为 NULL 值。

```
create or replace procedure INSPROC(xmlDoc IN CLOB,
tableName IN VARCHAR2) is
    insCtx DBMS_XMLSave.ctxType;
    rows number;
begin
  insCtx := DBMS_XMLSave.newContext(tableName); -- get the context
  rows := DBMS_XMLSave.insertXML(insCtx,xmlDoc); -- insert the doc
  DBMS_XMLSave.closeContext(insCtx); -- close the handle
end;
/
```

现在可以将 XML 文档作为 CLOB 的输入值传递给 INSPROC 过程，并传递表名。INSPROC 将接受 XML 文档并将其值插入表中。可以调用一个独立的过程 UPDATEPROC(如下面的程序清单所示)，来执行更新。

```
create or replace procedure UPDATEPROC ( xmlDoc IN clob) is
  updCtx DBMS_XMLSave.ctxType;
  rows number;
begin
    updCtx := DBMS_XMLSave.newContext('RATING'); -- get context
    DBMS_XMLSave.clearUpdateColumnList(updCtx); -- clear settings
    DBMS_XMLSave.setKeyColumn(updCtx,'RATING'); -- set RATING as key
    rows := DBMS_XMLSave.updateXML(updCtx,xmlDoc); -- update
    DBMS_XMLSave.closeContext(updCtx); -- close context
end;
/
```

在本示例中，Rating 列是用于更新的 where 子句的键列：

```
DBMS_XMLSave.setKeyColumn(updCtx,'RATING'); -- set RATING as key
```

新值作为 XML 文档，以 CLOB 格式传入，并且 RATING 表的值依据 XML 文档的内容进行更新。

delete 操作也以相同的方式进行工作。下面的过程以一个 XML 文档作为其输入，设定 Rating 列作为 RATING 表的键，并基于输入文档中的键值执行 delete 操作。请注意，newContext 以表名 RATING 作为其输入值，而 setKeyColumn 以键列名(这里恰巧也是 Rating)作为其输

入值。

```
create or replace procedure DELETEPROC(xmlDoc IN clob) is
  delCtx DBMS_XMLSave.ctxType;
  rows number;
begin
   delCtx := DBMS_XMLSave.newContext('RATING');
   DBMS_XMLSave.setKeyColumn(delCtx,'RATING');
   rows := DBMS_XMLSave.deleteXML(delCtx,xmlDoc);
   DBMS_XMLSave.closeContext(delCtx);
end;
/
```

52.3.2 XSU 和 Java

如果愿意，则可以通过调用 Java 类而不是 PL/SQL 过程，来执行 XSU DML 操作。为了取代 DBMS_XMLQuery 程序包，应当执行 oracle.xml.sql.query.OracleXMLQuery 类。可以使用 oracle.xml.sql.dml.OracleXMLSave 取代 DBMS_XMLSave 程序包。

例如，下面的程序清单稍微定制了一下 Oracle 提供的用于 XML 查询的样本 Java 程序。它依赖于连接管理的方式与 JDBC 示例中所示的方式非常相似。此程序导入 oracle.xml.sql.query.OracleXMLQuery 类，允许它获取 XML 串，然后，调用 Java 的 Systerm.Out.println 方法显示输出。

```
import oracle.xml.sql.query.OracleXMLQuery;
import java.lang.*;
import java.sql.*;

// class to test the String generation!
class testXMLSQL {

   public static void main(String[] argv)
   {

      try{
        // Create the connection.
        Connection conn = getConnection("practice","practice");

        // Create the query class.
        OracleXMLQuery qry = new OracleXMLQuery(conn, "select * from rating");

        // Get the XML string.
        String str = qry.getXMLString();

        // Print the XML output.
        System.out.println(" The XML output is:\n"+str);
        // Always close the query to get rid of any resources...
      qry.close();
      }catch(SQLException e){
       System.out.println(e.toString());
      }
    }

   // Get the connection given the user name and password...!
```

```
    private static Connection getConnection(String username,
  String password)
    throws SQLException
  {
     // Register the JDBC driver...
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

     // Create the connection using the OCI8 driver.
      Connection conn =
DriverManager.getConnection("jdbc:oracle:oci8:@or90",username,password);
     return conn;
  }
}
```

关于连接管理和 Java 的详细内容，请参阅第 VI 部分。

52.3.3　定制查询过程

用户可以创建自己的过程，以简化与 DBMS_XMLQuery 程序包和 DBMS_XMLSave 程序包的交互工作。例如，下面的程序清单创建一个名为 GET_XML 的函数，GET_XML 将从表中选择行，并以 XML 格式显示它们。

```
create function GET_XML (in_SQL varchar2)
    return CLOB
is queryCtx DBMS_XMLQuery.ctxType;
    result CLOB;
begin queryCtx := DBMS_XMLQuery.newContext(in_SQL);
    result := DBMS_XMLQuery.getXML(queryCtx);
    DBMS_XMLQuery.closeContext(queryCtx);
    return result;
end;
/
```

由于数据将以 CLOB 格式返回，因此可以使用 set long 命令设置最大显示长度(默认为 80 个字符)。可以从 SQL 查询中调用 GET_XML，如下面的程序清单所示：

```
set pagesize 100 long 2000

select GET_XML('select * from category') from DUAL;
GET_XML('SELECT*FROMCATEGORY')

--------------------------------------------------
<?xml version = '1.0'?>
<ROWSET>
   <ROW num="1">
      <CATEGORYNAME>ADULTREF</CATEGORYNAME>
      <PARENTCATEGORY>ADULT</PARENTCATEGORY>
      <SUBCATEGORY>REFERENCE</SUBCATEGORY>
   </ROW>
   <ROW num="2">
      <CATEGORYNAME>ADULTFIC</CATEGORYNAME>
      <PARENTCATEGORY>ADULT</PARENTCATEGORY>
      <SUBCATEGORY>FICTION</SUBCATEGORY>
   </ROW>
   <ROW num="3">
```

```
        <CATEGORYNAME>ADULTNF</CATEGORYNAME>
        <PARENTCATEGORY>ADULT</PARENTCATEGORY>
        <SUBCATEGORY>NONFICTION</SUBCATEGORY>
    </ROW>
    <ROW num="4">
        <CATEGORYNAME>CHILDRENPIC</CATEGORYNAME>
        <PARENTCATEGORY>CHILDREN</PARENTCATEGORY>
        <SUBCATEGORY>PICTURE BOOK</SUBCATEGORY>
    </ROW>
    <ROW num="5">
        <CATEGORYNAME>CHILDRENFIC</CATEGORYNAME>
        <PARENTCATEGORY>CHILDREN</PARENTCATEGORY>
        <SUBCATEGORY>FICTION</SUBCATEGORY>
    </ROW>
    <ROW num="6">
        <CATEGORYNAME>CHILDRENNF</CATEGORYNAME>
        <PARENTCATEGORY>CHILDREN</PARENTCATEGORY>
        <SUBCATEGORY>NONFICTION</SUBCATEGORY>
    </ROW>
</ROWSET>
```

52.4 使用 XMLType

可以使用 XMLType 作为表中的一种数据类型。XMLType 可用来存储和查询数据库中的 XML 数据。作为一种类型，XMLType 拥有(使用称为 Xpath 表达式的一类操作)用来访问、提取和查询 XML 数据的成员函数。SYS_XMLGEN、SYS_XMLAGG 和 DBMS_XMLGEN 程序包从已有的对象-关系数据中创建 XMLType 值。在指定某一列为使用 XMLType 数据类型的列时，Oracle 将在内部以 CLOB 数据类型存储相应的数据。

下面的程序清单创建一个使用 XMLType 数据类型的表：

```
create table MY_XML_TABLE
(Key1       NUMBER,
Xml_Column  SYS.XMLTYPE);
```

如果描述该表，就可以看到下面的列：

```
desc MY_XML_TABLE

Name                                      Null?    Type
----------------------------------------- -------- ------------
KEY1                                               NUMBER
XML_COLUMN                                         SYS.XMLTYPE
```

如果增加描述的详细程度，则还能看到 XMLType 列的成员函数。以下程序清单显示了众多可用函数和方法中的前面几个：

```
set describe depth 5
desc MY_XML_TABLE
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
 KEY1                                               NUMBER
```

```
  XML_COLUMN                                        SYS.XMLTYPE

METHOD
------
  STATIC FUNCTION CREATEXML RETURNS XMLTYPE
  Argument Name                  Type                    In/Out  Default?
  ------------------------------ ----------------------- ------ --------
  XMLDATA                        CLOB                    IN

METHOD
------
  MEMBER FUNCTION EXTRACT RETURNS XMLTYPE
  Argument Name                  Type                    In/Out  Default?
  ------------------------------ ----------------------- ------ --------
  XPATH                          VARCHAR2                IN

METHOD
------
  MEMBER FUNCTION EXISTSNODE RETURNS NUMBER
  Argument Name                  Type                    In/Out Default?
  ------------------------------ ----------------------- ------ --------
  XPATH                          VARCHAR2                IN

METHOD
------
  MEMBER FUNCTION ISFRAGMENT RETURNS NUMBER

METHOD
------
  MEMBER FUNCTION GETCLOBVAL RETURNS CLOB

METHOD
------
  MEMBER FUNCTION GETSTRINGVAL RETURNS VARCHAR2

METHOD
------
  MEMBER FUNCTION GETNUMBERVAL RETURNS NUMBER
```

现在，可以调用 XMLType 的 CREATEXML 方法将值插入 MY_XML_TABLE 中：

```
insert into MY_XML_TABLE (Key1, Xml_Column)
values (1, SYS.XMLTYPE.CREATEXML
  ('<book>
    <title>MY LEDGER</title>
    <chapter num= "1 ">
      <title>Beginning</title>
      <text>This is the ledger of G.B. Talbot</text>
  </chapter>
</book>'));
```

可以调用 GETCLOBVAL 方法来查询数据。例如，如下查询：

```
select M.Xml_Column.GETCLOBVAL() as XML_Data
    from MY_XML_TABLE M
  where Key1 = 1;
```

返回以下数据：

```
XML_DATA
--------------------------------------------------------
<book>
  <title>MY LEDGER</title>
  <chapter num="1">
    <title>Beginning</title>
    <text>This is the ledger of G.B. Talbot</text>
  </chapter>
</book>
```

注意：
可以在用 XMLType 数据类型创建的列上创建函数索引，以改善查询的性能。

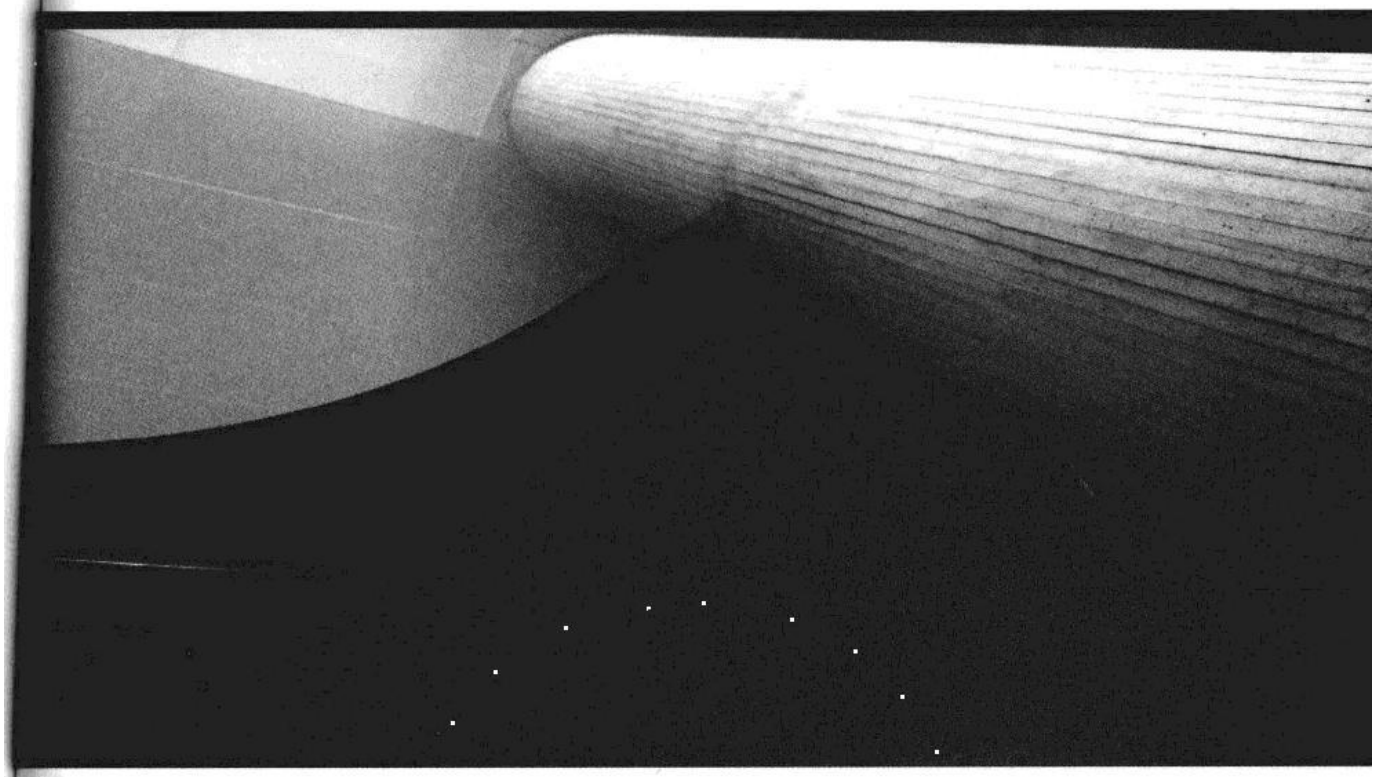
52.5 其他功能

本章已经介绍了 XML 数据与 Oracle 数据库交互的高级概念。此外，还有许多其他的可能性，包括使用用于数据格式化的样式表和创建 XSQL 页面。甚至可以在 XMLType 列上创建文本索引，只要安装了 Oracle Context 选项：

```
create index XML_INDEX on MY_XML_TABLE (Xml_Column)
  indextype is ctxsys.context;
```

Oracle 提供了很多与 XML 数据交互的程序包，包括 DBMS_XMLDOM、DBMS_XMLGEN、DBMS_XMLINDEX、DBMS_XMLPARSER、DBMS_XMLQUERY、DBMS_XMLSAVE、DBMS_XMLSCHEMA、DBMS_XMLSTORE 和 DBMS_XMLTRANSLATIONS。

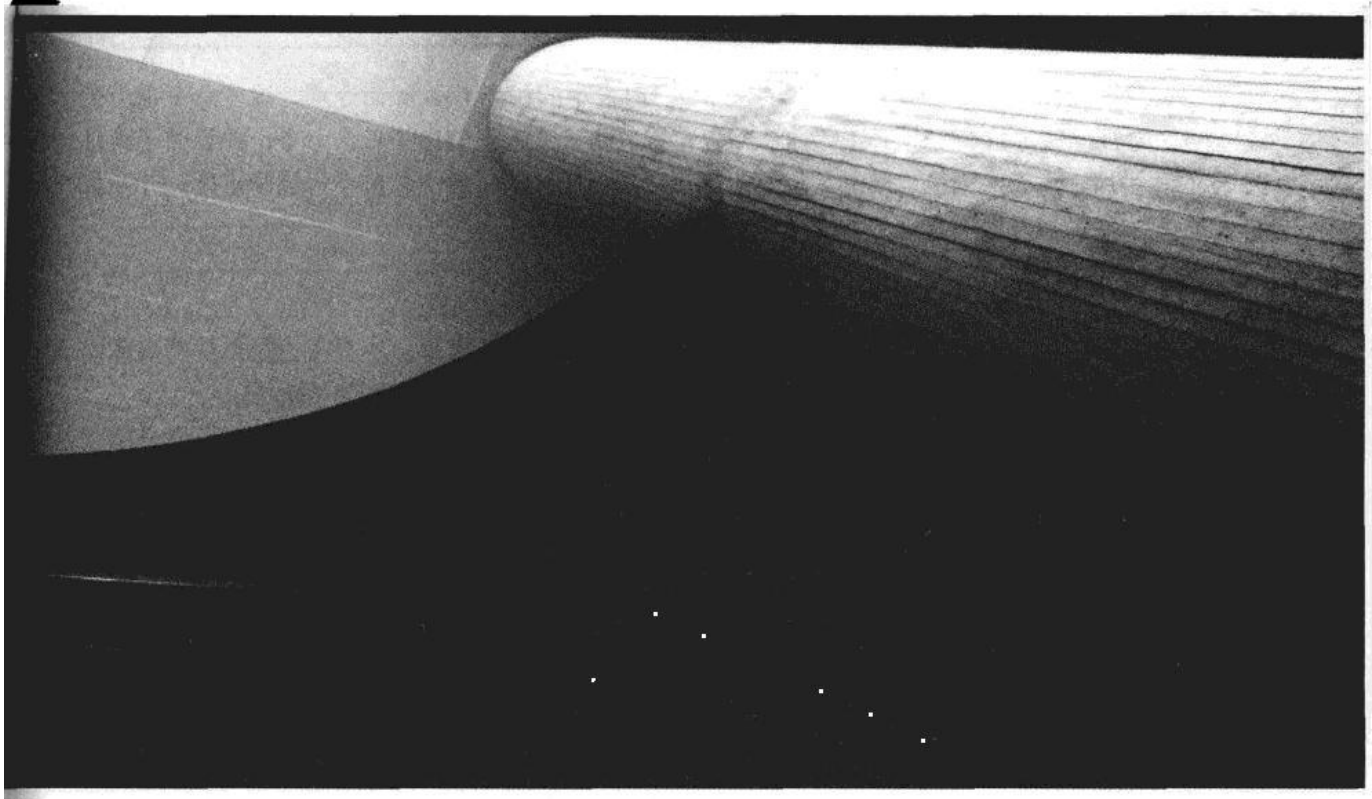
Oracle 还提供支持更多用法的实用程序，如 oraxml(分析 XML 文档的一个命令行界面)和 oraxsl(用来在多个 XML 文档上应用样式表的命令行界面)等。每个实用程序和数据访问方法的完整介绍超出了本书范围；关于其他选项和样本程序，请参阅 Oracle Database 11g 提供的 XML 文档。关于作用于 XMLType 实例的函数的描述，包括 EXTRACT、EXTRACTVALUE、SYS_XMLAGG、SYS_XMLGEN、UPDATEXML、XMLAGG、XMLCONCAT 以及 XMLELEMENT，请参考附录 A。



第Ⅷ部分

附　　录

附录 A　命令和术语参考



附 录 A

命令和术语参考

本附录版权(有效期为 2008 年)归 Oracle 公司所有。本附录不仅包含了经 Oracle 公司许可的 Oracle Corporation User Documentation 部分修订信息，而且简要总结了大多数常用 SQL 命令、函数和数据类型。这些命令按字母顺序排列。这些材料摘自 *Oracle Database SQL Reference Release 11g*、*SQL*Plus User's Guide and Reference* 以及 *Oracle Database Utilities 11g Release 1*。

本部分包含最主要的 Oracle 命令、函数和关键字的参考以及不同主题的交叉引用。本参考可供 Oracle 开发人员和用户使用，但要求用户对 Oracle 的各种产品有一定的了解。本参考的说明部分介绍了如何最有效地使用本参考中的各项内容。

1. 本参考包含的内容

本参考几乎包含 SQL、PL/ SQL 和 SQL*Plus 中的每条 Oracle 命令项，这些命令也用在 Oracle Database 11g 和 SQL 中。本参考以正确的格式或语法列出了每条命令，并说明了其目的和用途、重要的细节、限制、相关的提示以及使用示例。各主题按字母顺序排列，并经常会在本参考以及本书前面的各个章节中交叉引用。

2. 本参考不包含的内容

本参考不是教程，不解释面向界面的开发和管理工具。此外，本参考不包括那些功能比较特定和较少用到的命令。关于这些内容，可以参阅 Oracle 手册中的相应信息。

3. 各项的常规格式

本参考中的各项通常是术语的定义或者是函数、命令和关键字的描述。每项通常由 6 部分组成，即关键字、类型、参阅(SEE ALSO)交叉引用、关键字出现的格式、其组成部分的描述以及使用示例。下面是一个典型的格式：

RPAD

参阅：CHARACTER FUNCTIONS、LPAD、LTRIM、RTRIM、TRIM 和第 7 章。

格式：RPAD(string，length[，'set'])

描述：Right PAD。RPAD 通过在字符串右边添加一组特定字符，使字符串达到一定的长度。如果未指定字符集，则默认将空格用作填充字符。

示例：

```
select RPAD('HELLO ',24,'WORLD') from DUAL;
```

结果如下：

```
HELLO WORLDWORLDWORLDWOR
```

4. 每项的组成部分

关键字(KEYWORD)通常单独占一行。在某些情况下，后跟一个简短的定义。如果某个关键字在不同的 Oracle 工具或不同的 Oracle 版本中有多个定义，则此关键字后跟一个简短的限定词，以表示此关键字会多次列出。

参阅部分给出与该关键字密切相关的其他主题，或给出本书中详细介绍在实际中怎样使用该关键字的章节。有时，也会引用含有超出本参考范围的 Oracle 手册或参考书籍。

格式部分一般使用 Oracle 手册的表示法，所有 SQL 和其他关键字均为大写。在实际使用中，必须严格按给出的形式输入(不要求区分大小写的除外)。变量和可变参数用小写表示，实际使用时应当替换为适当的值。注意别漏掉圆括号。

5. 变量的标准用法

变量的一些标准用法如表 A-1 所示。

表 A-1　变量的一些标准用法

变　量	表　示
column	列名
database	数据库名
link	在 Oracle Net 中的链接名
password	口令
segment	段名
table	表名
tablespace	表空间名
user	用户名或所有者名
view	视图名

6. 其他格式化指导原则

其他格式化的指导原则如下：

- character 表示此值必须是单个字符。
- string 一般代表一个字符列，或经过数据自动转换后可作为一个 CHAR 列或 VARCHAR2 列处理的表达式或列。
- value 一般代表一个 NUMBER 列，或经过数据自动转换后可作为 NUMBER 列处理的表达式或列。
- date 一般代表一个 DATE 列，或经过数据自动转换后可作为 DATE 列处理的表达式或列。
- datetime 一般代表一个 TIMESTAMP 列，或经过数据自动转换后可作为 TIMESTAMP 列处理的表达式或列。
- integer 必须是一个整数，如 -3、0 或 12。
- expression 表示列的任何形式。它可以是一个字面量、变量、算术计算、函数或者函数和列的任何组合，其最终结果是一个值，如一个串、数字或日期。
- 有时也使用其他符号，如 condition 或 query。这会在上下文中解释，或本身就很明显，不需要进行解释。
- 可选项括在方括号中，如[user.]，表示 user 可以使用，也可以不使用。
- 程序清单中的可替代项由一条竖线分隔，如 OFF | ON(读作 OFF 或者 ON)。在一些系统中，这条竖线显示为一条短竖线。
- 必选项指项目列表中必需的项，用大括号括起来，如{OFF | ON}。
- 列表中的默认项(如果有)将列在第一位。
- 3 个句点(或省略号)表示前面的表达式可以重复任意多次，如 column[, column]...，这表示 column 可以为由逗号分隔的任意数目的其他列。
- 在少数情况下，对于要显示的内容，普通的表示法不能用或不适合。在这样的情况下，描述部分将更加清楚地说明格式部分中想要处理的内容。

7. 程序清单中的其他元素

一些命令有返回类型，表示函数返回值的数据类型。

描述是命令及其组成部分的解释。描述中的英文通常指格式部分给出的命令或变量。

示例或者显示函数的结果，或解释在一个实际命令中怎样使用关键字。示例的风格与格式部分不一样。它遵循本书第 I 部分的风格(第 5 章描述过)，因为这种风格在实际编码实践中更为典型。

8. 关键字的顺序

此参考是按字母顺序排列的，所有以符号开头的项都位于以字母 A 开头的项之前。

有连字符的单词或其中有下划线字符的单词按字母顺序排列，将连字符或下划线字符作为空格对待。

9. 符号

表 A-2 按出现的先后顺序列出各个符号，同时给出符号的一个简短定义或名称。

表 A-2 附录中出现的符号

符 号	说 明
_	下划线
!	感叹号
"	双引号
#	镑符号
$	美元符号
?	问号
%	百分号
&	与符号
&&	双与符号
'	单引号或撇号
()	圆括号
*	星号或乘号
**	PL/SQL 中的幂符号
+	加号
--	减号或连字符
-	双连字符、SQL 注释减号或连字符
.	句点或点、名称或变量分隔符
..	到……地方
/	除号或斜杠
/*	斜杠星号、SQL 注释的开始
:	冒号
:=	PL/SQL 中的赋值符号
;	分号
<<>>	PL/SQL 中的标签名限定符
<	小于符号
<=	小于等于符号
<>	不等于符号

(续表)

符　号	说　明
!=	不等于符号
=	等于符号
>	大于符号
>=	大于等于符号
@	@符号
@@	双@符号
[]	方括号
^	插入符号
^=	不等于符号
{}	大括号
|	分隔竖杠符号
||	连接符号

_(下划线)

下划线表示 LIKE 运算符的一个位置。请参阅 LIKE。

_EDITOR

请参阅 EDIT。

!(感叹号)

参阅：$、=、CONTAINS、SOUNDEX，第 27 章。

格式：在 SQL 中，“!”可作为“不等于”表达式“!=”的一部分。例如，可以选择不在亚洲的所有城市：

```
select City
  from LOCATION
 where Continent != 'ASIA';
```

对于 CONTEXT 文本索引，“!”通知文本引擎执行 SOUNDEX 搜索。要搜索的项将会扩展，以包含发音与搜索项相似的那些项，使用此文本的 SOUNDEX 值确定可能的匹配：

```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, '!grate')>0;
```

"(双引号)

参阅：ALIAS、TO_CHAR、第 7 章和第 10 章。

描述：双引号把包含特殊字符或空格的表或列的别名括起来，也可以把 TO_CHAR 的日期格式子句中的字面量文本括起来。

示例：双引号可用于别名，如下例所示：

```
select NEXT_DAY(CycleDate,'FRIDAY') "Pay Day!"
  from PAYDAY;
```

将其用于 TO_CHAR 的格式部分的示例如下所示：

```
select FirstName, Birthdate, TO_CHAR(BirthDate,
       '"Baby Girl on" fmMonth ddth, YYYY, "at" HH:MI "in the Morning"')
       "Formatted"
  from BIRTHDAY
 where FirstName = 'VICTORIA';

FIRSTNAME       BIRTHDATE   Formatted
--------------- ---------   ------------------------------
VICTORIA        20-MAY-49   Baby Girl on May 20th, 1949,
                            at 3:27 in the Morning
```

#(镑符号)

参阅： REMARK、/**/和第 6 章。

描述： #结束 SQL*Plus 启动文件中的一个文档块(块以 DOCUMENT 开头)。SQL*Plus 忽略从单词 DOCUMENT 到#之间的所有行。

$(美元符号)

参阅： @、@@、HOST、START、CONTAINS 和第 27 章。

格式： 用于 SQL*Plus 的示例，如下所示：

```
$ host_command
```

在 UNIX 环境下，执行主机命令时$符号可以用“!”替换。

用于 CONTEXT 文本索引的示例如下所示：

```
select column
  from table
 where CONTAINS( text_column, '$search_term') >0;
```

描述： $将任何主机命令传回给操作系统执行，而不用退出 SQL*Plus。$为 HOST 的缩写。但它并非在所有操作系统或硬件中都起作用。

对于 CONTEXT 文本索引，$指示搜索引擎对搜索项进行词根扩展。词根扩展包含了具有相同词根的搜索词。例如，单词“sing”的词根扩展将包含“singing”、“sang”和“sung”。

?(问号)

参阅： CONTAINS 和第 27 章。

格式：

```
select column
  from table
 where CONTAINS( text_column, '?term')>0;
```

描述： 对于 CONTEXT 文本索引，“?”指示文本引擎进行模糊匹配搜索。要搜索的项将扩展为包含拼写类似于搜索项的项，用文本索引作为可能的匹配源。

%(百分号)

%是一个表示 LIKE 运算符的位置和字符数目的通配符。请参阅 LIKE。

%FOUND

参阅： %ISOPEN、%NOTFOUND、%ROWCOUNT、CURSOR、SQL CURSOR 和第 32 章。

格式：

```
cursor%FOUND
```

或

```
SQL%FOUND
```

描述： %FOUND 为 select、insert、update 和 delete 操作成功的标志。cursor 是一个在 PL/SQL 块中声明的显式游标名或名为 SQL 的隐式游标。%FOUND 可以作为后缀，附加到游标名上。这二者都是 PL/SQL 块中 SELECT、INSERT、UPDATE 和 DELETE 语句执行成功的标志。

PL/SQL 临时预留出一部分内存作为执行 SQL 语句的临时存储器，或者用于存储一些关于执行状态的信息(或属性)。如果 SQL 语句是 select，则这个区域将包含一行数据。

%FOUND 为其中一个属性。通常在游标中执行了显式的 fetch 后，才测试%FOUND(通过 IF/THEN 逻辑)。如果 fetch 检索到一行，则%FOUND 为 TRUE；否则为 FALSE。它总是得出 TRUE、FALSE 或 NULL 值。%NOTFOUND 在逻辑上正好相反。当%FOUND 为 TRUE 时它为 FALSE，在%FOUND 为 FALSE 时它为 TRUE，而在%FOUND 为 NULL 时它为 NULL。

在一个显式游标打开前测试其%FOUND 的结果，将触发一个 EXCEPTION(错误代码为 ORA-01001，INVALID CURSOR)。

%ISOPEN

参阅： SQL CURSOR 和第 32 章。

格式：

```
cursor%ISOPEN
```

描述： cursor 必须是一个显式声明的游标名或者名为 SQL 的隐式游标。如果指定的游标是打开的，则判定%ISOPEN 的值为 TRUE；否则为 FALSE。SQL%ISOPEN 的值总是为 FALSE，因为 SQL 游标总是在执行未显式声明的 SQL 语句时自动打开并关闭(请参阅 SQL CURSOR)。%ISOPEN 用于 PL/SQL 逻辑，它不能作为 SQL 语句的一部分。

%NOTFOUND

请参阅%FOUND。

%ROWCOUNT

参阅： CLOSE、DECLARE CURSOR、DELETE、FETCH、INSERT、OPEN、SELECT、

UPDATE 和第 32 章。

格式：

```
cursor%ROWCOUNT
```

描述： cursor 必须是一个显式声明的游标名或者名为 SQL 的隐式游标。cursor%ROWCOUNT 包含从这个游标的有效集合中获取的行的累积总数。%ROWCOUNT 可特意用来处理固定数目的行，但更为常见的是用只打算返回一行的 select 语句的异常处理程序(例如 select…into)。在这些情况下，如果没有行返回，则%ROWCOUNT 设置为 0(%NOTFOUND 也可以进行此测试)。如果返回多行，则不管返回的实际行数是多少，它都设置为 2。

可以在 PL/SQL 逻辑中使用%ROWCOUNT，它不能作为 SQL 语句的组成部分。如果使用 SQL%ROWCOUNT，则只能引用最近打开的隐式游标。如果隐式游标没有打开，则 SQL%ROWCOUNT 将返回 NULL。

%ROWTYPE

参阅： FETCH 和第 32 章。

格式：

```
{[user.]table | cursor}%ROWTYPE
```

描述： %ROWTYPE 声明一个记录变量，它具有与表(或视图)中整个行相同的结构，或者与指定的游标检索到的行具有相同的结构。%ROWTYPE 用作变量声明的一部分并确保此变量包含适当的字段和数据类型，以便处理所有取出的列。如果使用[user.]table，则该表(或视图)必须存在于在数据库中。

为了创建包含相应字段的变量，可使用 declare、一个变量名和带有表名的%ROWTYPE，然后选择一行给该记录。由于该变量与该表有相同的结构，因此可以以 variable.field 形式引用字段，即使用类似于 SQL 语句中使用的 table.column 的表示法。select…into 和 fetch…into 是加载整个记录变量的唯一方法。而记录中的个别字段可以使用如下所示的“:=”进行加载：

```
BOOKSHELF_RECORD.Publisher := 'PANDORAS';
```

如果将一个游标用作%ROWTYPE 的前缀，则它会包含一条 select 语句，需要多少列其中就仅包含多少列。然而，如果从一个指定游标中取出的列是一个表达式，而不是一个简单的列名，则必须在使用此方法引用该表达式前，在 select 语句中给出一个别名。

%TYPE

参阅： %ROWTYPE 和第 32 章。

格式：

```
{[user.]table. column | variable}%TYPE
```

描述： %TYPE 声明的变量与前面已经声明过的变量类型相同，它也可以声明已连接的

数据库中已有表的一个特殊列。

示例：在本例中，声明一个新变量 Writer，它具有与 AUTHOR 表中的 AuthorName 列相同的数据类型。由于 Writer 已经存在，因此可以用它来声明另一个新变量 Coauthor。

```
Writer    AUTHOR.AuthorName%TYPE;
Coauthor  Writer%TYPE;
```

&或&&(与符号或双与符号)

参阅：":"、ACCEPT、DEFINE、START、CONTAINS 和第 27 章。

格式：用于 SQL*Plus 的格式如下：

```
& integer
& variable
&& variable
```

用于 CONTEXT 索引的格式如下：

```
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property & harvests')>0;
```

描述："&"与"&&"有好几种用法("&&"仅用于下面的第二种定义)：

- 作为"SQL*Plus 启动文件中参数的前缀。值被"&1"和"&2"等替换，请参阅 START。
- 作为 SQL*Plus 的 SQL 命令中替换变量的前缀。如果发现一个未定义的"&"或"&&"变量，则 SQL*Plus 将提示一个值。"&&"将定义变量，从而保留其值。"&"将不定义也不保留该值，只替换一次输入的值。请参阅 ACCEPT 和 DEFINE。
- 在使用 CONTEXT 索引时，"&"用来指定涉及多个搜索项的文本搜索的 AND 条件。如果这些搜索项中任何一个没有找到，则搜索将不返回文本。

'(单引号)

参阅："(双引号)和第 7 章。

格式：

```
'string'
```

描述：单引号括住一个字面值，如括住一个字符串或日期常量。为了在串常量中使用一个单引号或一个撇号，应该用两个单引号(而不是双引号)。只要有可能，就应当避免在数据中(并且在别处)使用单引号。

示例：

```
select 'William' from DUAL;
```

将输出：

```
William
```

而：

```
select 'William''s brother' from DUAL;
```

输出结果为：

```
William's brother
```

( )(圆括号)

参阅： PRECEDENCE、SUBQUERY、UPDATE、第 13 章和第 15 章。

描述： 圆括号将子查询或列的列表括起来，或控制表达式内的优先级。

***(乘号)**

参阅： +、-、/、第 9 章和第 27 章。

格式：

```
value1 * value2
```

描述： value1*value2 表示 value1 乘以 value2。对于 CONTEXT 索引，“*”用来分别为每个不同的搜索项加权。

****(幂符号)**

参阅： POWER。

格式：

```
x ** y
```

描述：x 为底数，y 为指数，表示 x 的 y 次幂。x 和 y 可以是常量、变量、列或表达式。两者都必须是数值。

示例：

```
4**4 = 256
```

+(加号)

参阅： -、*、/和第 9 章。

格式：

```
value1 + value2
```

描述： value1+value2 表示 value1 加上 value2。

-(减号[形式 1])

参阅： +、*、/、第 9 章、CONTAINS、MINUS 和第 27 章。

格式：

```
value1 - value2
```

描述：value1-value2 表示 value1 减去 value2。对于 CONTEXT 索引，减号告诉两项的

文本搜索在比较该结果与阈值的分数前，从第一项的搜索分数中减去第二项的搜索分数。对于 CTXCAT 索引，减号告诉文本搜索引擎：如果发现某项后面紧跟一个减号，则从结果集中排除该行。

-(连字符[形式 2])

参阅：第 6 章。

格式：

```
command text -
        text -
        text
```

描述：SQL*Plus 命令的延续符，“-”使一条命令在下一行中继续执行。

示例：

```
ttitle left 'Current Portfolio' -
right 'November 1st, 1999' skip 1 -
center 'Industry Listings ' skip 4;
```

--(注释)

参阅：/**/、REMARK 和第 6 章。

格式：

```
-- any text
```

描述：--(注释)告诉 Oracle 一条注释已经开始。从此处到本行结尾的一切内容都作为注释处理。这些限定符只在 SQL 本身或 PL/SQL 中使用，且必须出现在 SQLTERMINATOR 之前。

示例：

```
select Feature, Section, Page
-- this is just a comment
  from NEWSPAPER -- this is another comment
 where Section = 'F';
```

.(句点或点[形式 1])(SQL*Plus)

格式：

```
&variable.suffix
```

描述：“.”是一个变量分隔符，在 SQL*Plus 中用来将变量名与后缀分隔开，以防止将后缀作为变量名的一部分。

示例：把后缀“st”将与变量“&Avenue”的内容进行连接：

```
define Avenue = 21
select '100 &Avenue.st Street' from DUAL;
```

结果如下：

```
100 21st Street
```

同样的方法还可以用于 WHERE 子句。

.(句点或点[形式 2])

参阅： SYNTAX OPERATORS 和第 38 章。

格式：

```
[user.][table.]column
```

描述： 这里的“.”是一个名称分隔符，用来指定列的全名，包含(可选)其表名或用户。句点还用来处理 select、update 和 delete 语句中的抽象数据类型中的列。

示例：

```
select Practice.BOOKSHELF.Title
  from Practice.BOOKSHELF, Practice.BOOKSHELF_AUTHOR
 where Practice.BOOKSHELF.Title = Practice.BOOKSHELF_AUTHOR.Title;
```

如果该表包含基于抽象数据类型的列，则数据类型的属性可以通过“.”符号访问。

```
select C.Person.Address.City
  from Company C
 where C.Person.Address.State = 'FL';
```

更多示例请参阅第 38 章。

..(到……地方)

请参阅 LOOP。

/(除号[形式 1])

参阅： +、-、*和第 9 章。

格式：

```
value1 / value2
```

描述： value1/value2 表示 value1 除以 value2。

/(除号[形式 2])(SQL*Plus)

参阅： “;”、BUFFER、EDIT、GET、RUN 以及 SET 下的 SQLTERMINATOR。

描述： /(除号)在 SQL 缓冲区中执行 SQL，但不显示 SQL。与 RUN 不同，RUN 首先显示 SQL。

//(注释)**

参阅： REMARK 和第 6 章。

结果如下：

```
100 21st Street
```

同样的方法还可以用于 WHERE 子句。

.(句点或点[形式 2])

格式：

```
/* any text */
```

描述：“/*”告诉 Oracle 一段注释已经开始。从这里开始往下的所有内容，即使有很多字和行，Oracle 也看作注释，直到遇到“*/”为止(“*/”表示注释结束)。注释不能嵌套于注释内，即一个“/*”由其后的第一个“*/”终止，即使中间再插入另一个“/*”，也是如此。

示例：

```
select Feature, Section, Page
    /* this is a multi-line comment
       used to extensively document the code */
  from NEWSPAPER
where Section = 'F';
```

:(冒号、宿主变量前缀)

参阅：INDICATOR VARIABLE

格式：

```
:name
```

描述：name 是一个宿主(host)变量名。当 PL/SQL 通过 Oracle 预编译器嵌入一个宿主语言中时，宿主变量可以从 PL/SQL 块中将一个逗号作为它们的宿主语言名的前缀来引用。事实上，这就是宿主变量。如果 PL/SQL 通过赋值(:=)修改了它的值，宿主变量的值就会被修改。它们可以用于任何可以使用 PL/SQL 变量的地方。一个例外的情况是将 NULL 值赋予宿主变量，它不支持直接这种赋值，而需要使用指示器变量。

示例：

```
int BookCount;
...
   select COUNT(*) into :BookCount from BOOKSHELF;
```

:=(赋值符号)

参阅：DECLARE CURSOR、FETCH、SELECT INTO 和第 32 章。

格式：

```
variable := expression
```

描述：把 expression 赋予 PL/SQL 的 variable，expression 可以是常量、NULL，或者是带有其他变量、字面量或 PL/SQL 函数的计算结果。

示例：

```
Extension := Quantity * Price;
Title := 'BARBERS WHO SHAVE THEMSELVES';
Name := NULL;
```

;(分号)

参阅：/(斜杠)、EDIT、GET、RUN、SET 下面的 SQLTERMINATOR、CONTAINS、NEAR 和第 27 章。

描述：分号执行 SQL 或它前面的命令。对于 CONTEXT 索引，分号表示应该对给定的文本字符串执行近似搜索。如果搜索项是“summer”和“lease”，那么这两项的近似搜索为：

```
select Text
  from SONNET
 where CONTAINS(Text,'summer;lease') >0;
```

在判定文本搜索结果时，与单词“summer”和“lease”彼此接近的文本将比与单词“summer”和“lease”差异较大的文本得分高。

<<>>(PL/SQL 标签名限定符)

参阅：BEGIN、BLOCK STRUCTURE、END、GOTO、LOOP 和第 32 章。

@(“At”符号[形式 1])(SQL*Plus)

参阅：@@和 START

格式：

```
@file
```

描述：@启动名为 file 的 SQL*Plus 启动文件。@类似于 START，但不允许有命令行参数。

@(“At”符号[形式 2])

参阅：CONNECT、COPY、CREATE DATABASE LINK 和第 25 章。

格式：

```
CON[NECT] user[/password] [@database];
COPY [FROM user/password@database]
     [TO user/password@database]
     {APPEND | CREATE | INSERT | REPLACE}
     table [ (column, [column]...) ]
     USING query;
SELECT... FROM [user.]table[link] [, [user.]table[@link] ]...
```

描述：@在 CONNECT 或者 COPY 命令中用作数据库名的前缀，或者在 FROM 子句中用作链接名的前缀。

@@(双“At”符号)(SQL*Plus)

参阅：@(形式 1)和 START。

格式：

```
@@file
```

描述：@@启动一个名为 file 的嵌套 SQL*Plus 启动文件。@@类似于@，但在一个命令文件中使用时，@@的不同之处在于，它在与调用它的命令文件相同的目录(而不是在执行该命令文件的目录)中搜索启动文件。

{}(花括号)

参阅：CONTAINS、第 27 章和第 42 章。

格式：对于文本搜索，{}表示被括起来的文本应该作为搜索字符串的一部分，即使它是一个保留字也是如此。例如，如果搜索项为“in and out”，并且想要搜索整个短语，则必须把“and”括在花括号内：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'transactions {and} finances')>0;
```

在 Java 中，{}表示一个块的开始和结束。

|(竖线)

参阅：BTITLE、SET HEADSEP、TTITLE、第 6 章和第 27 章。

格式：

```
text|text
```

描述：在用于 SQL*Plus 的 COLUMN、TTITLE 或 BTITLE 命令中时，“|”为默认的 HEADSEP 字符，并用来标志将一行拆分为二行(在用于本参考的程序清单中时，“|”用来分开选项。“variable | literal”可理解为“变量或字面值”。在某些机器上显示为一条短竖线)。在文本查询中，“|”表示涉及两个搜索项的 OR 条件。如果找到其中一个搜索项且其搜索得分超过指定的阈值，则该搜索返回文本。

示例：下面说明了怎样将“|”用做 HEADSEP 字符：

```
TTITLE 'This is the First Line|and This is the Second'
```

输出结果是：

```
This is the First Line
and This is the Second
```

下面说明了如何将“|”用于文本搜索查询中的 OR 子句：

```
REM CONTAINS method for CONTEXT indexes:
select Title
  from BOOK_REVIEW_CONTEXT
 where CONTAINS(Review_Text, 'property | harvests')>0;
REM CATSEARCH method for CTXCAT indexes:
select Title
  from BOOK_REVIEW_CTXCAT
 where CATSEARCH(Review_Text, 'property | harvests', NULL)>0;
```

||(连接符号)

参阅：.(句点或点[形式 1])、CHARACTER FUNCTIONS、SUBSTR 和第 7 章。

格式：

```
expression1 || expression2
```

描述：“||”将两个串连接到一起。

示例：使用“||”显示一列城市，后跟一个逗号、一个空格和相应的国家：

```
select City||', '||Country from LOCATION;
```

ABS(绝对值)

参阅：NUMBER FUNCTIONS 和第 9 章。

格式：

```
ABS(value)
```

value 必须是一个数值，可以是字面量数值、数值列名、包含有效数值的字面值字符串或只包含一个有效数值的字符列。

描述：绝对值用来衡量某些值的幅度大小，它总是一个正数。

示例：

```
ABS(146) = 146
ABS(-30) = 30
ABS('-27.88') = 27.88
```

ABSTRACT DATATYPES

抽象数据类型(Abstract datatypes)是由一个或多个属性组成的数据类型。抽象数据类型不限于标准的 Oracle 数据类型 NUMBER、DATE 和 VARCHAR2，它可以由多个属性和(或)其他已经定义过的抽象数据类型组成。抽象数据类型还称为对象类型(object type)，这一叫法更为常见。有关示例可参阅第 38 章，它的创建语法可以参阅 CREATE TYPE。

ACCEPT

参阅：&、&&和 DEFINE。

格式：

```
ACC[EPT] variable [NUM[BER]|CHAR|DATE|BINARY_FLOAT|BINARY_DOUBLE] [FOR[MAT]
format] [DEF[AULT] default] [PROMPT text | NOPR[OMPT]] [HIDE]
```

描述：ACCEPT 接受用户键盘输入的值，并把它按指定的数据类型放入指定的变量中。如果该变量以前没有定义(DEFINE)，就创建该变量。FORMAT 指定响应的输入格式(如 A20)。如果不指定响应，则依据 DEFAULT 设置默认值。PROMPT 在接受输入前向用户显示文本。NOPROMRT 跳过一行并等待输入，不显示任何提示。如果既不使用 PROMPT 也不使用 NOPROMPT 子句，就会导致 ACCEPT 创建一个要求输入变量值的提示。HIDE 隐藏用户的

输入，它对于口令等需要保护的内容很有用。

示例：下面用“How hot？” 提示用户，并把用户的输入放在名为 Temperature 的数值变量中：

```
ACCEPT Temperature NUMBER PROMPT 'How hot? '
```

ACCUMULATE

请参阅 CONTAINS。

ACOS

参阅：ASIN、ATAN、ATAN2、COS、COSH、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

```
ACOS(value)
```

描述：ACOS 函数返回一个余弦值对应的弧度。输入值的范围为 - 1~1，输出值用弧度表示。

ADD_MONTHS

参阅：DATE FUNCTIONS 和第 10 章。

格式：

```
ADD_MONTHS(date,integer)
```

描述：ADD_MONTHS 将多个月份添加到 date 上，并返回未来多个月的日期。date 必须是一个合法的 Oracle 日期。integer 必须是一个整数，非整数值将被截断为下一个最小的整数。若 integer 为负值，则返回一个过去的日期。

示例：ADD_MONTHS 向 April 22，2001 添加一个月和 0.3 个月：

```
select ADD_MONTHS(TO_DATE('22-APR-01'),1),
       ADD_MONTHS(TO_DATE('22-APR-01'),.3) from DUAL;

ADD_MONTH ADD_MONTH
--------- ---------
22-MAY-01 22-APR-01
```

AGGREGATE FUNCTIONS

参阅：第 9 章。

描述：聚集函数用一组值中的不同数值计算出一个单个总计数值(如求和与求均值)。下面是 Oracle 版本的 SQL 中当前所有的组函数按字母顺序排列的列表。每个函数都在本参考中的其他地方用它们自己的名称(包含适当的格式和用法)列出。聚集函数在选择列表、ORDER BY 子句和 HAVING 子句中很有用。

函数名和用途

AVG 函数给出一组记录的平均值。

COLLECT 函数接受输入的 column 值，并在选中的行外创建一个输入类型的嵌套表。

CORR 函数返回一组数对的相关系数。

CORR_*函数支持非参数或秩相关(CORR_K 和 CORR_S)。
COUNT 函数统计一列或一个表(用*)中行的数目。
COVAR_POP 函数返回一组数对的总体协方差。
COVAR_SAMP 函数返回一组数对的样本协方差。
CUME_DIST 函数计算一组值中某个值的累积分布。
DENSE_RANK 函数计算一组已排好序的行中某一行的排序值。
FIRST 函数作用于来自行集的一组值，得到按照给定排序规范排列在最前面的一组行。
GROUP_ID 函数区分源于 GROUP BY 规范的复制组。
GROUPING 函数从常规已分组的行中区分超聚集行。
GROUPING_ID 函数返回与行关联的 GROUPING 位向量对应的一个数值。
LAST 函数作用于来自行集的一组值，得出按给定排序规范排列在最后的一组行。
MAX 函数给出一组行中所有值的最大值。
MEDIAN 函数给出一组行的中间值(不考虑 NULL)。
MIN 函数给出一组行中所有值的最小值。
PERCENT_RANK 函数计算一组行的排序值，用百分数表示。
PERCENTILE_CONT 函数是对一个假定的连续分布模型进行百分比计算的函数。
PERCENTILE_DISC 函数是对一个假定的离散分布模型进行百分比计算的函数。
RANK 函数计算一组值中某个值的排序值。
REGR 函数对一组值进行线形回归分析。
STATS_BINOMIAL_TEST 函数测试一个给定的比例和样本比例之间的差别。
STATS_CROSSTAB 函数分析两个标称值。
STATS_F_TEST 函数测试两个变量是否有显著差别。
STATS_KS_TEST 函数测试两个变量是否源于同一个总体。
STATS_MODE 函数返回一组值中出现频率最高的值。
STATS_MW_TEST 函数测试虚假设的两个独立样本。
STATS_ONE_WAY_ANOVA 函数是变量的单向分析。
STATS_T_TEST_*函数测量方式差别的重要性。
STATS_WSR_TEST 函数决定样本之间的差别是否和 0 显著不同。
STDDEV 函数给出一组行中所有值的标准差。
STDDEV_POP 函数计算总体标准差，并返回总体方差的平方根。
STDDEV_SAMP 函数计算累积样本标准差，并返回样本方差的平方根。
SUM 函数返回一组行中所有值的和。
VAR_POP 函数返回一组数(除去组中的 NULL 值后)的总体方差。
VAR_SAMP 函数返回一组数(除去组中的 NULL 值后)的样本方差。
VARIANCE 函数给出一组行中所有值的方差。

ALIAS

别名是在 SQL 语句内分配给表或列的临时名称，可用此别名在相同的语句(如果是表)内

或者在 SQL*Plus 命令内(如果是列)的其他地方引用它。可用 AS 关键字把列定义和它的别名分隔开。请参阅"(双引号)、AS 和 SELECT。当别名用于表时，可称为相关名(correlation name)。

ALL

参阅： ANY、BETWEEN、EXISTS、IN、LOGICAL OPERATORS 和第 13 章。

```
operator ALL list
```

描述： “!=ALL” 等价于 NOT IN。operator 可以是 “=”、“>”、“>=”、“<”、“<=” 或 “!=” 之一，list 可以是一系列字面量字符串(如 ‘Talbot’、‘Jones’、‘Hild’ 等)或者为一系列字面量数值(如 2、43、76、32.06、444 等)，也可以是来自一个子查询的列，其中子查询的每行都是列表的一个成员，如下所示：

```
LOCATION.City != ALL (select City from WEATHER)
```

list 值也可以是主查询的 WHERE 子句中的一组列，如下所示：

```
Prospect != ALL (Vendor, Client)
```

限定条件： list 不能是子查询中的一组列，例如：

```
Prospect != ALL (select Vendor, Client from . . .)
```

许多人发现这个运算符和 ANY 运算符非常难记，因为在某些情况下它们的逻辑不太直观。结果，某些形式的 EXISTS 通常被替换。带有 ALL 的运算符与一个列表的组合可用下面的解释来说明：

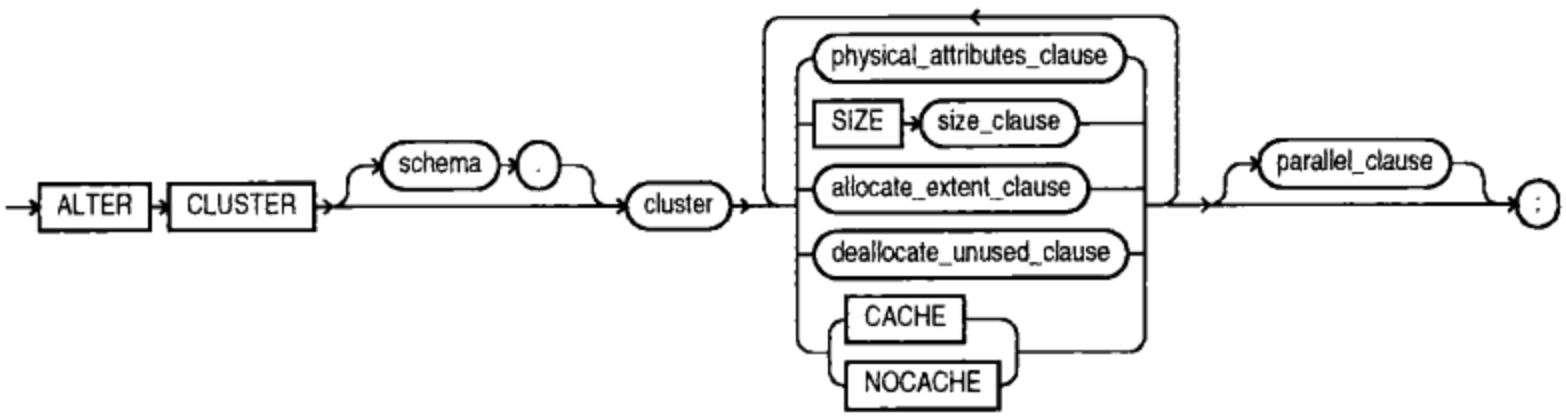
Page = ALL(4,2,7)	Page 等于列表(4,2,7)中的每一项——其中项没有数目限定。子查询可返回一个列表，其中所有项都是相同的，并且 Page 的值可等于它们中的每一项，但这种情况很罕见。
Page > ALL(4,2,7)	Page 大于列表(4,2,7)中的最大项——限定为大于 7 的任何项
Page > = ALL(4,2,7)	Page 大于等于列表(4,2,7)中的最大项——限定为大于等于 7 的任何项
Page < ALL(4,2,7)	Page 小于列表(4,2,7)中的最小项——限定为小于 2 的任何项
Page <= ALL(4,2,7)	Page 小于等于列表(4,2,7)中的最小项——限定为小于等于 2 的任何项
Page != ALL(4,2,7)	Page 不等于列表(4,2,7)中的任意项——限定为除 4、2、7 外的任何项

ALTER CLUSTER

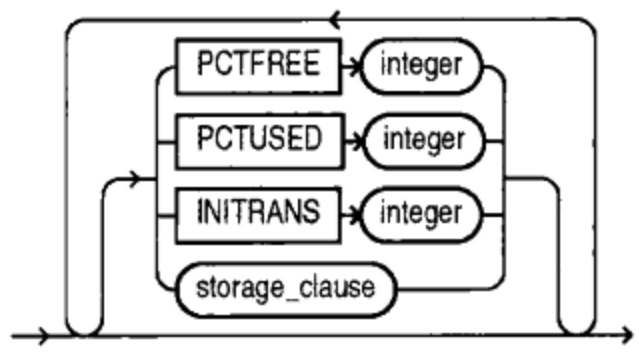
参阅： CREATE CLUSTER、CREATE TABLE、STORAGE 和第 17 章。

格式：

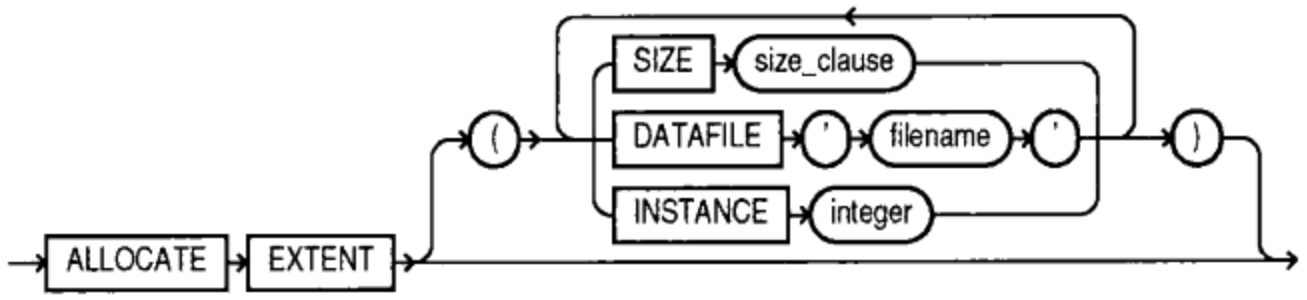
alter_cluster::=



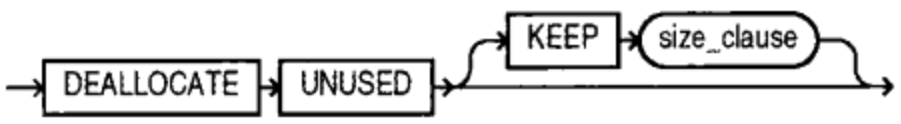
physical_attributes_clause::=



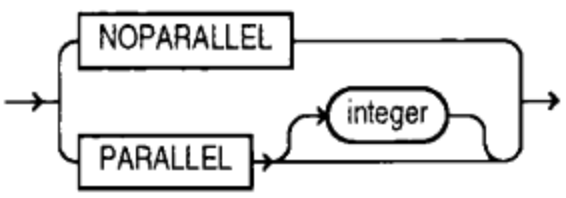
allocate_extent_clause::=



deallocate_unused_clause::=



parallel_clause::=



描述：相应参数的描述可以在 CREATE TABLE 下找到。这些参数除了在这里用于更改群集外，其目的和作用相同。大小在 CREATE CLUSTER 中描述。要更改一个群集，必须拥有该群集，或者拥有 ALTER ANY CLUSTER 系统权限。

群集中所有的表都使用由 CREATE CLUSTER 设置的 STORAGE 值。ALTER CLUSTER 可更改这些值，以用于未来的群集块，但并不影响那些已经存在的值。ALTER CLUSTER 不允许更改 STORAGE 中的 extent 参数。DEALLOCATE UNUSED 子句允许群集缩小大小，释放不用的空间。

示例：

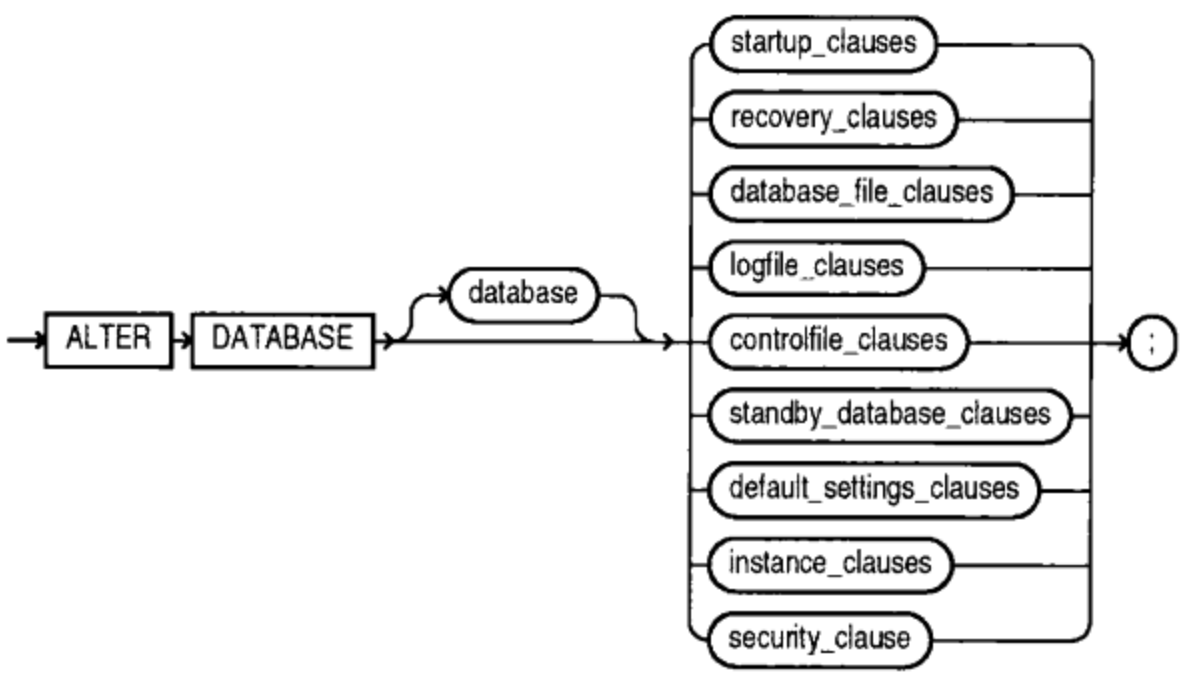
```
alter cluster BOOKandAUTHOR size 1024;
```

ALTER DATEBASE

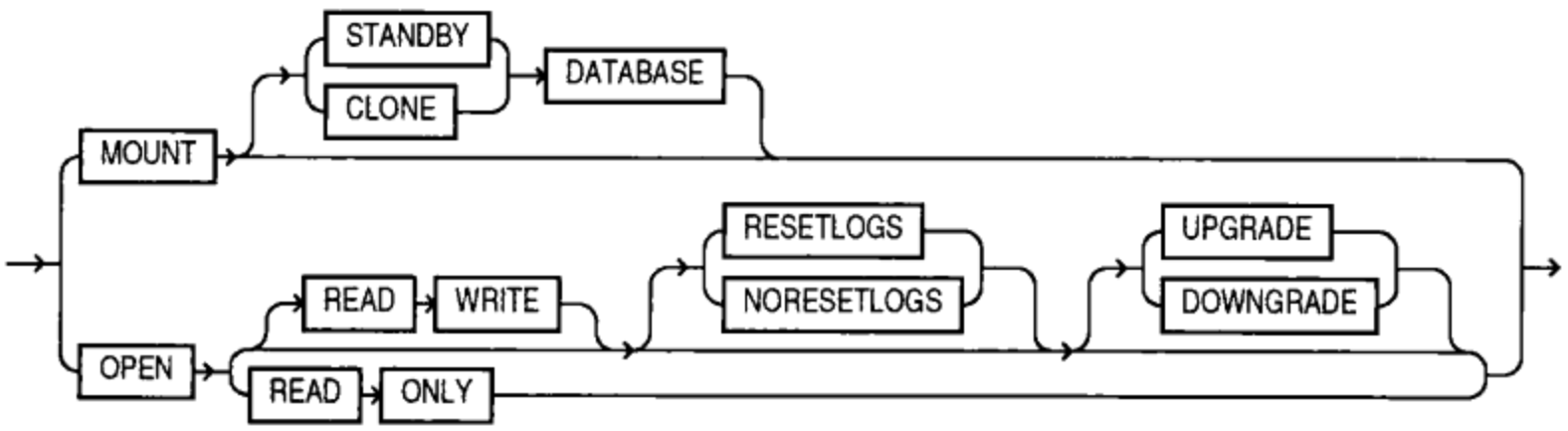
参阅：ALTER ROLLBACK SEGMENT、CREATE DATABASE、RECOVER、STARTUP、SHUTDOWN 和第 51 章。

格式：

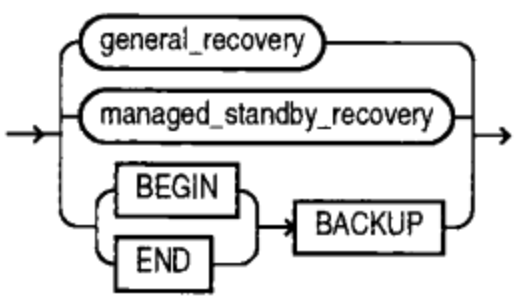
alter_database::=



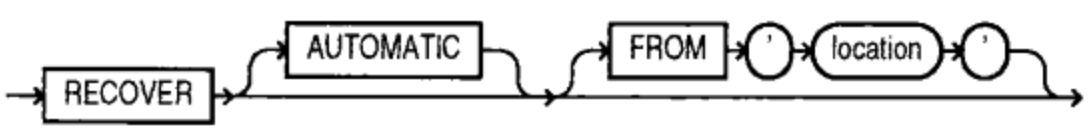
startup_clauses::=

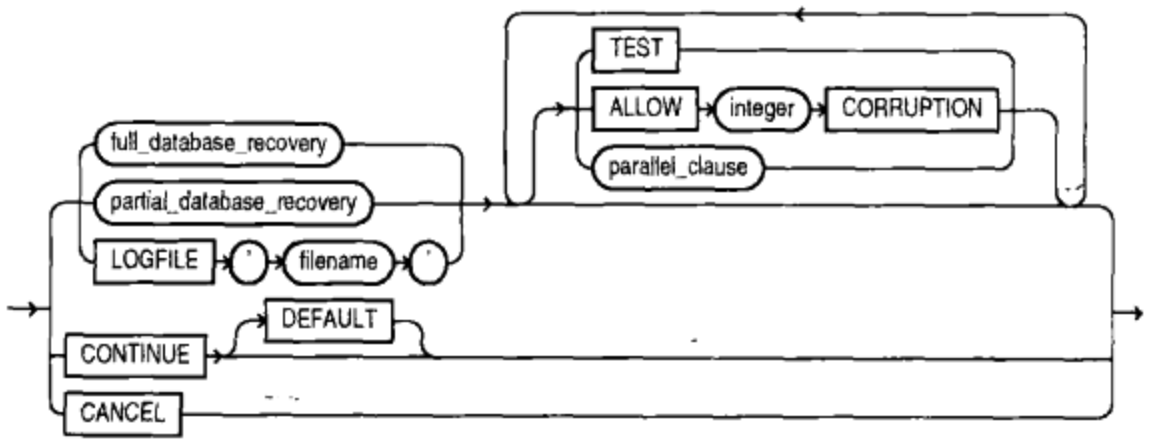


recovery_clauses::=

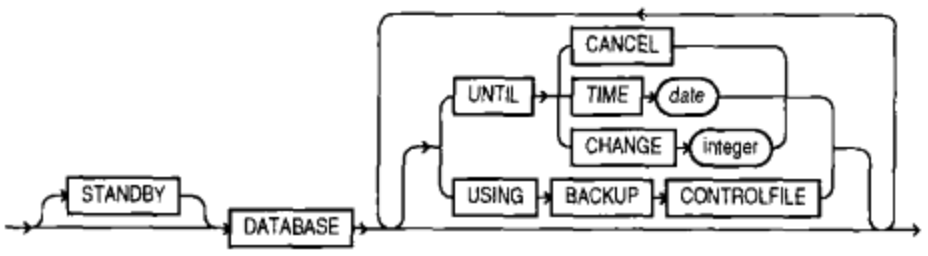


general_recovery::=

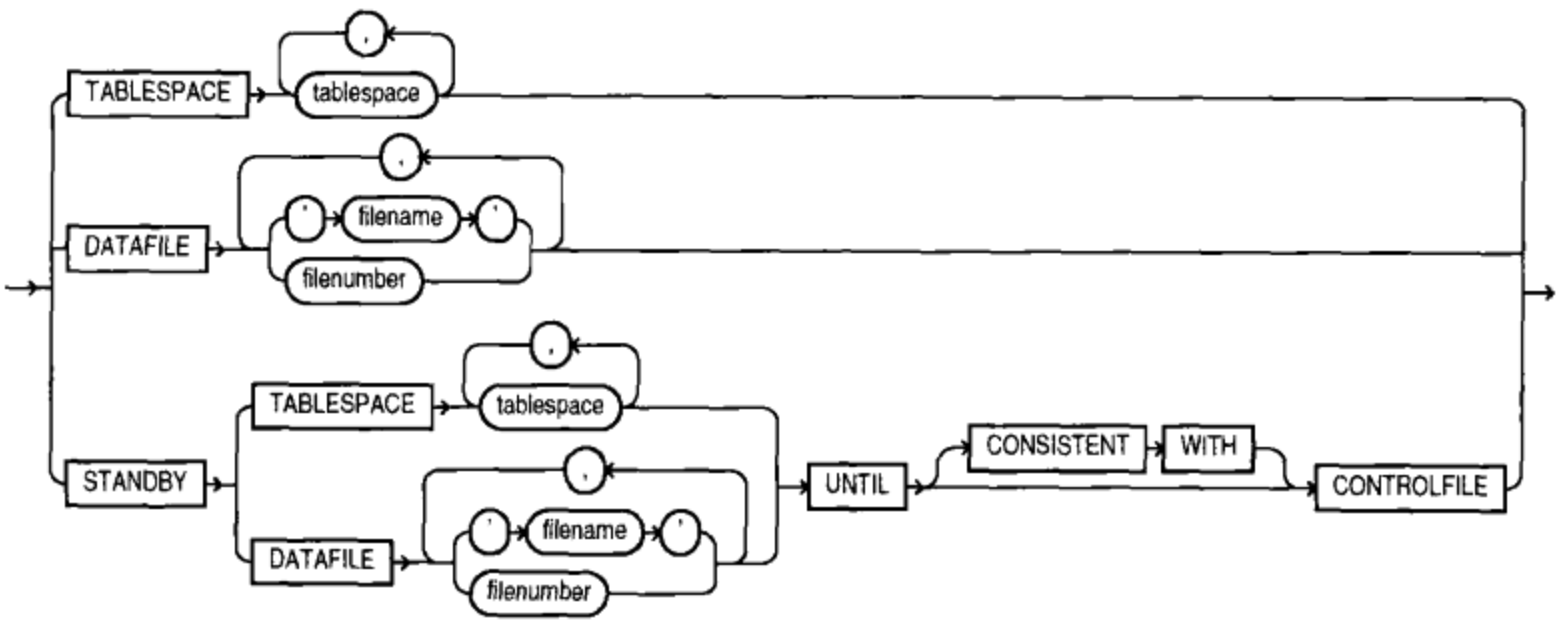




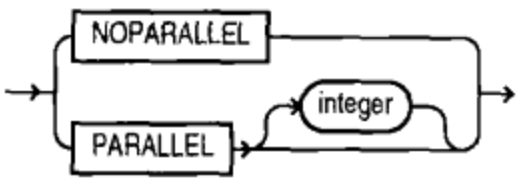
full_database_recovery::=



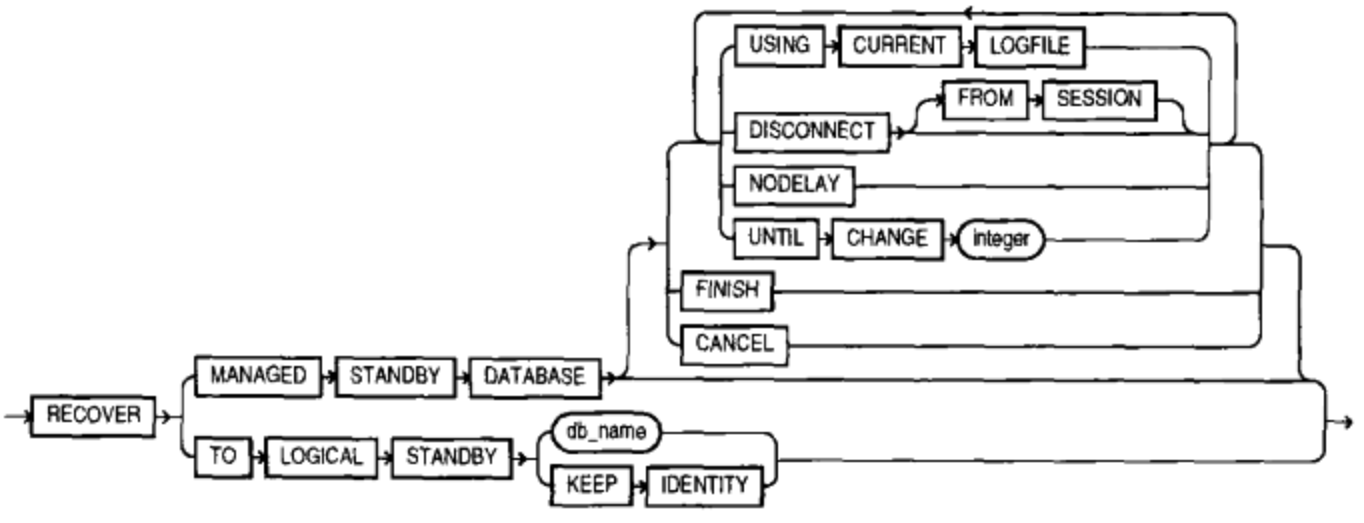
partial_database_recovery::=



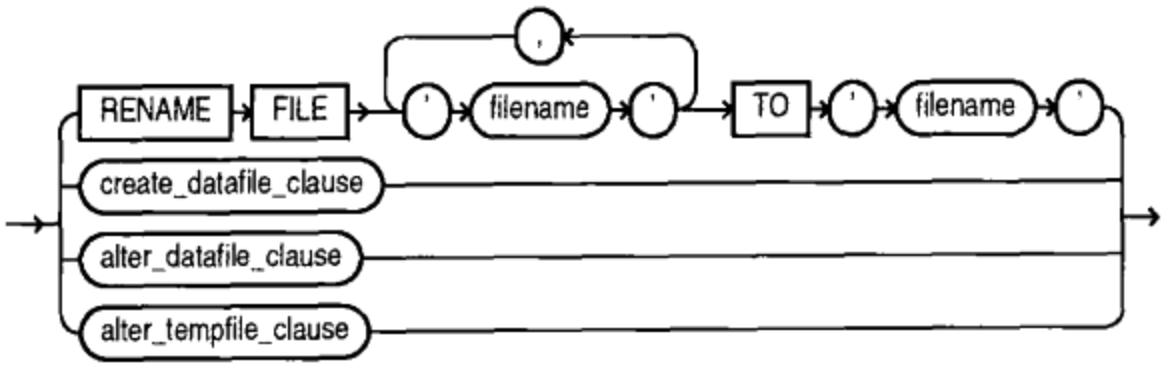
parallel_clause::=



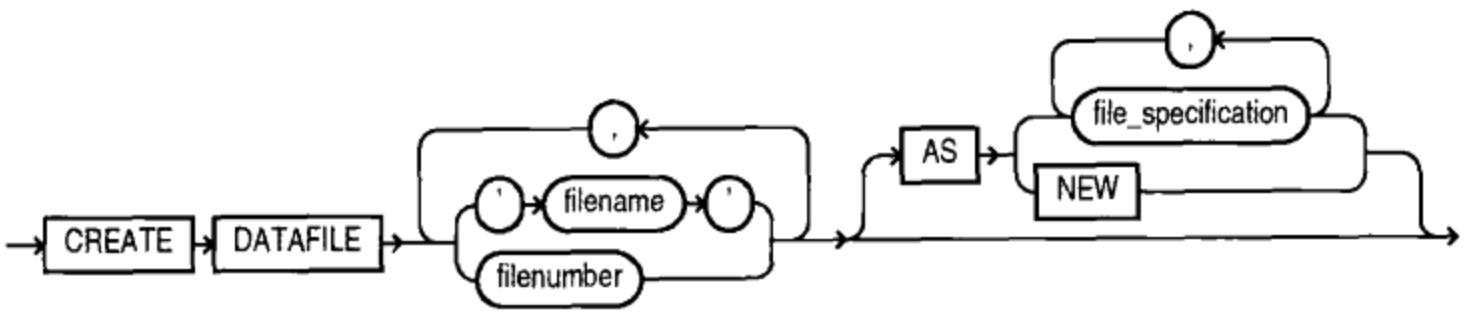
managed_standby_recovery::=



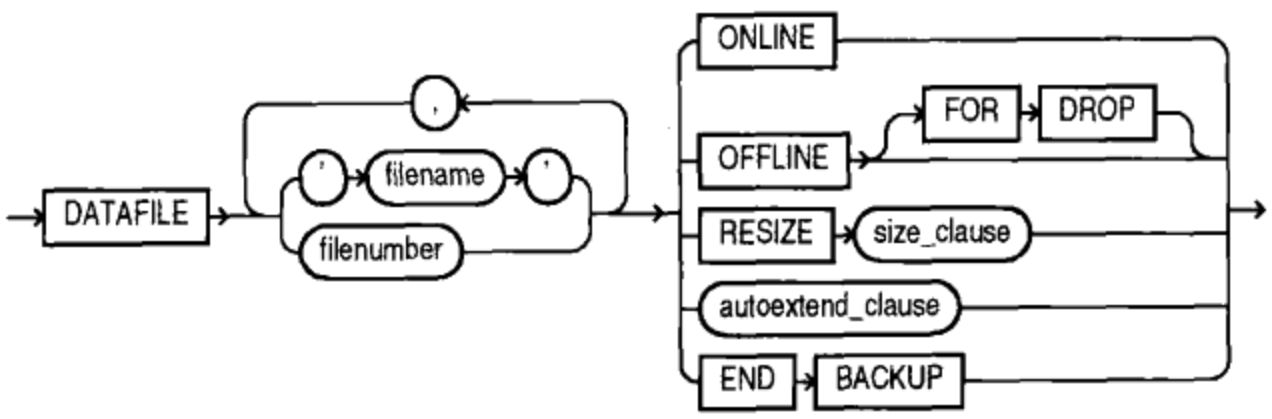
database_file_clauses::=



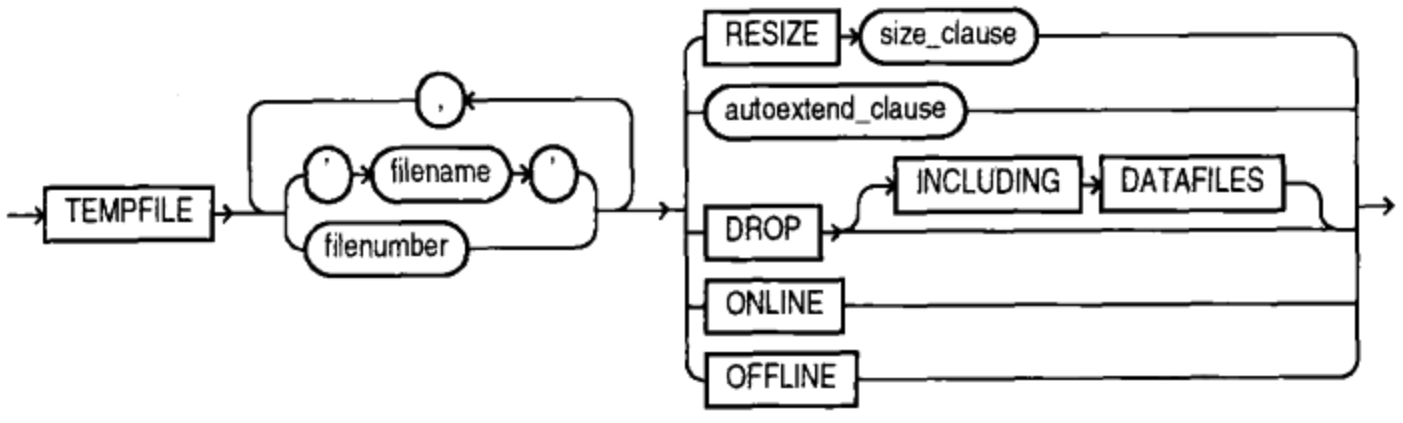
create_datafile_clause::=



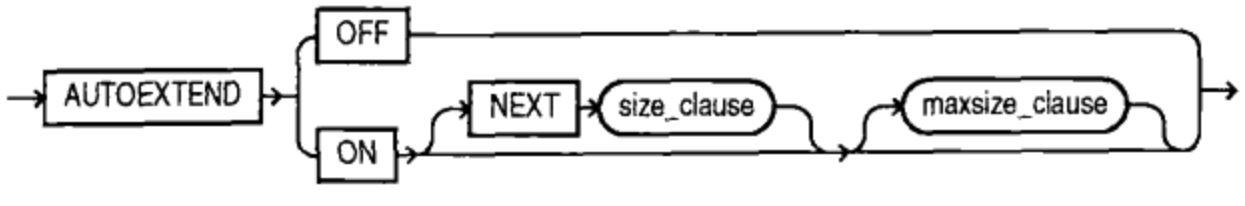
alter_datafile_clause::=



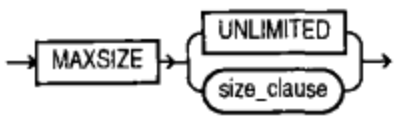
alter_tempfile_clause::=



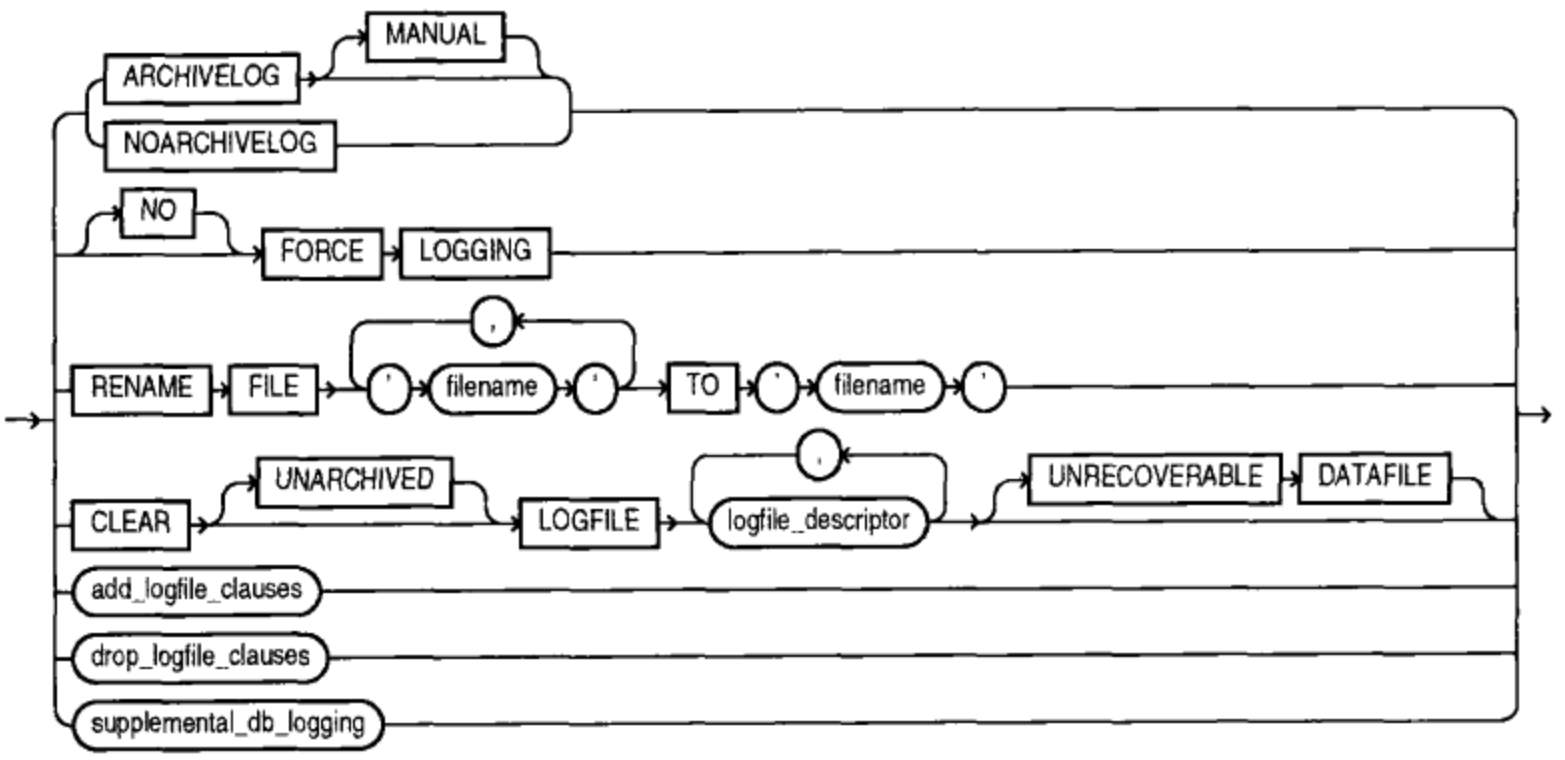
autoextend_clause::=



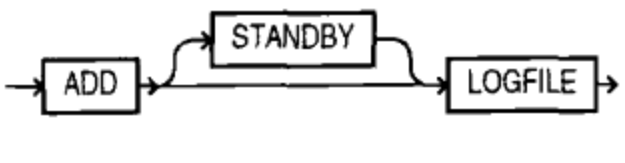
maxsize_clause::=

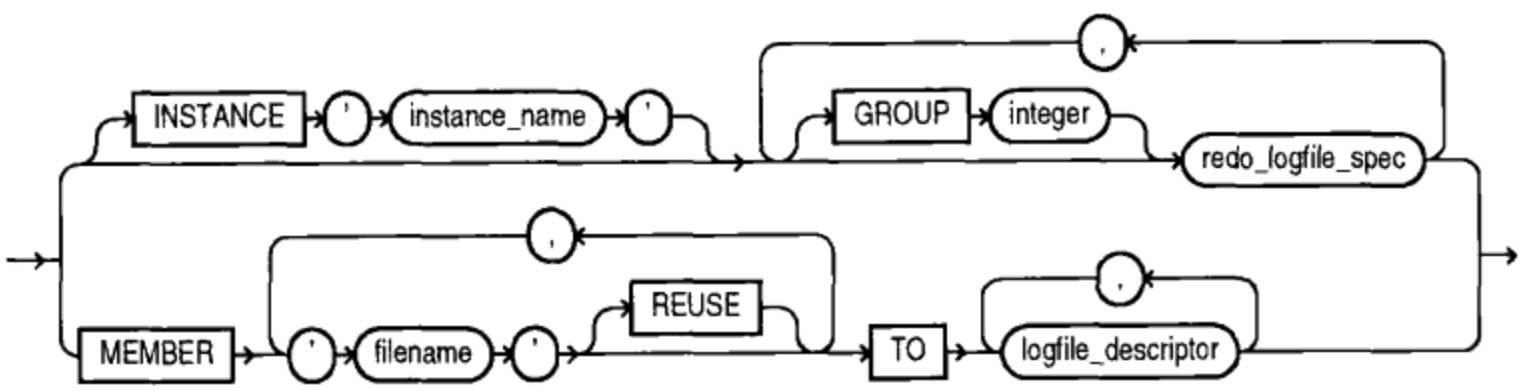


logfile_clauses::=

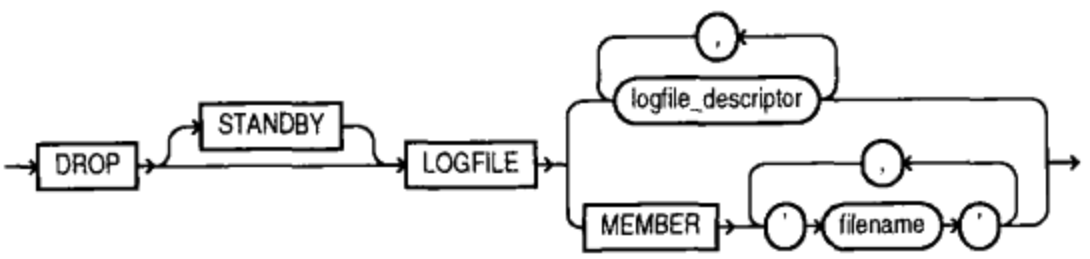


add_logfile_clauses::=

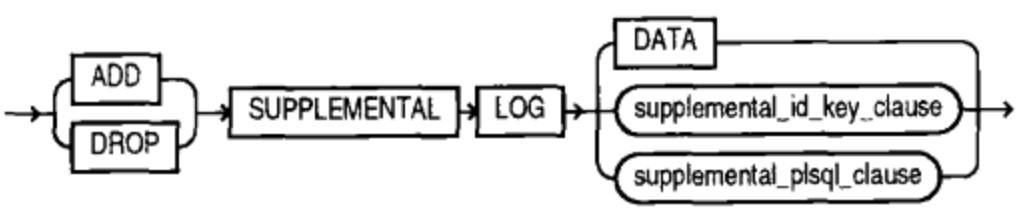




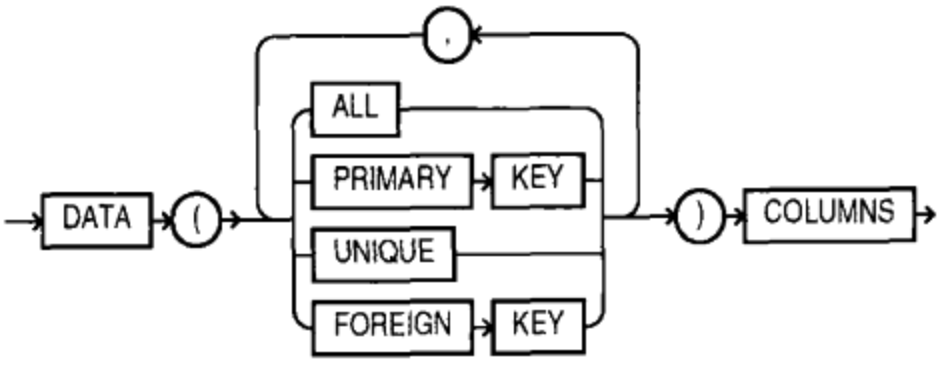
drop_logfile_clauses::=



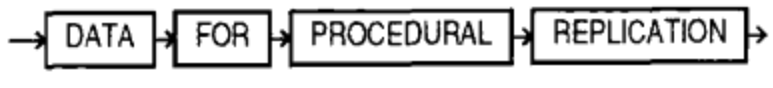
supplemental_db_logging::=



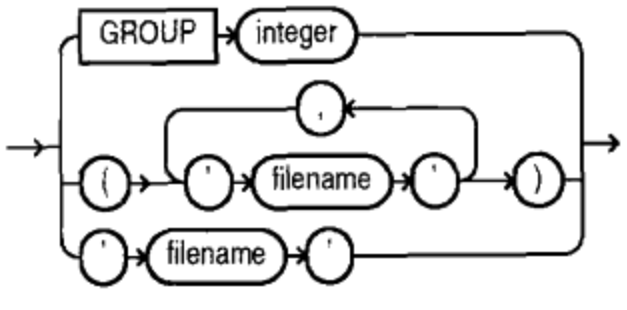
supplemental_id_key_clause::=



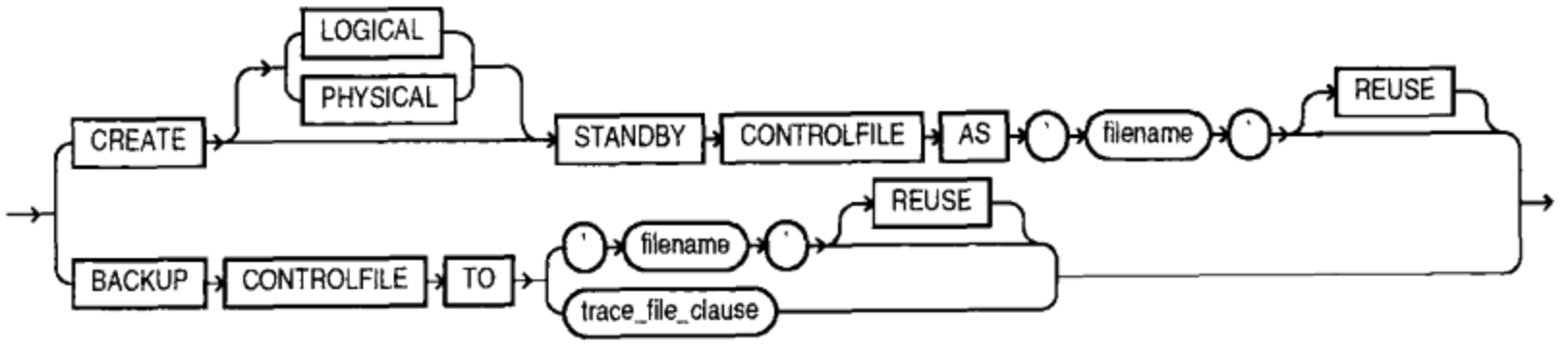
supplemental_plsql_clause::=



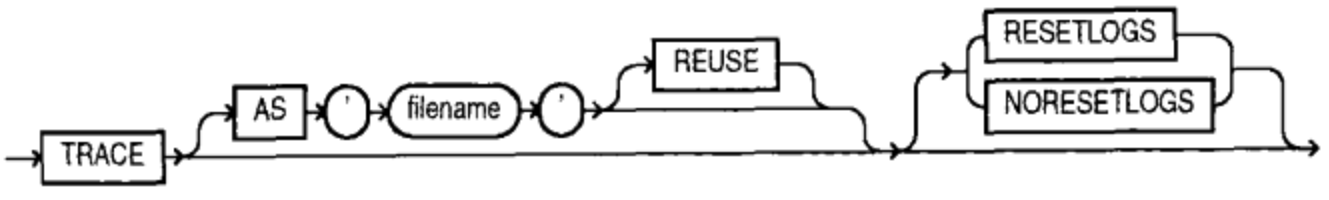
logfile_descriptor::=



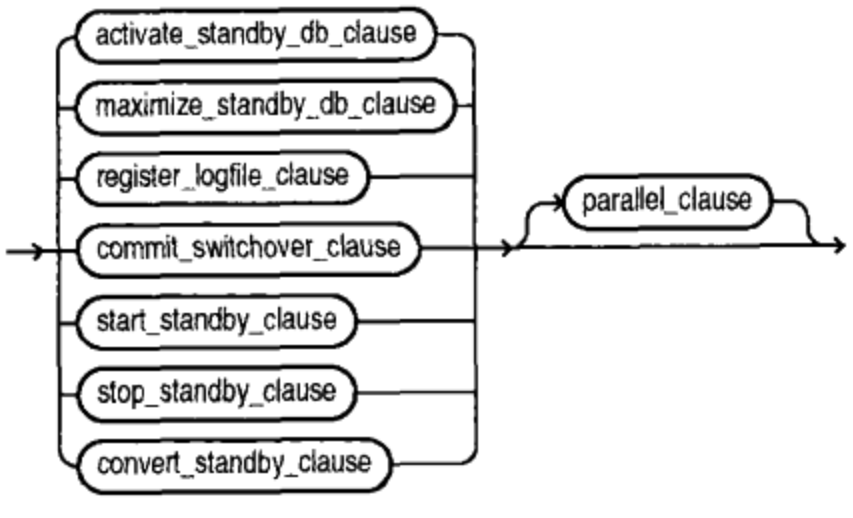
controlfile_clauses::=



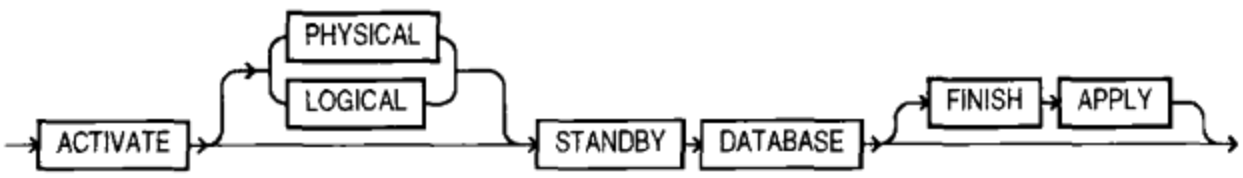
trace_file_clause::=



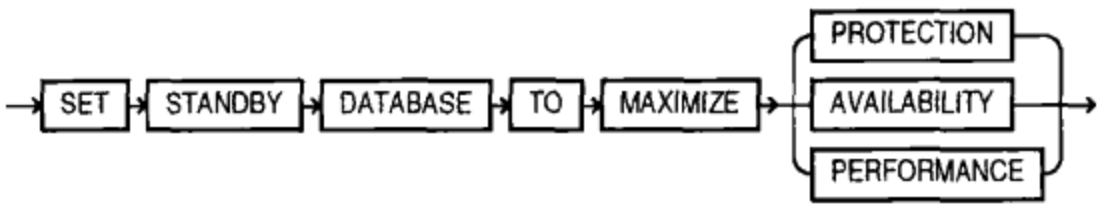
standby_database_clauses::=



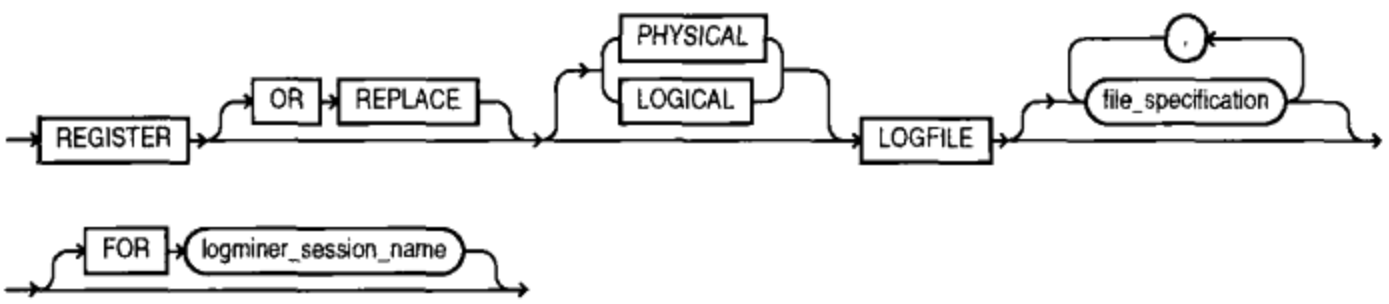
activate_standy_db_clause::=



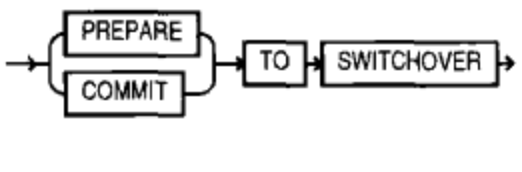
maximize_standby_db_clause::=

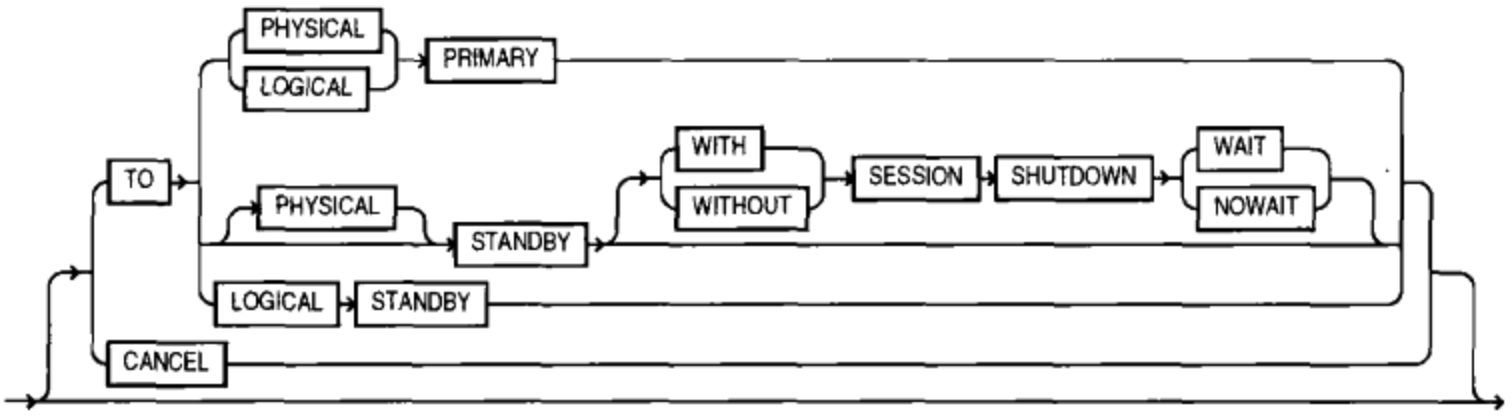


register_logfile_clause::=

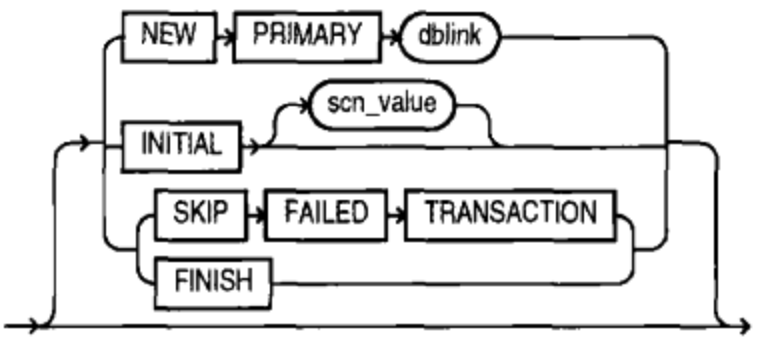


commit_switchover_clause::=



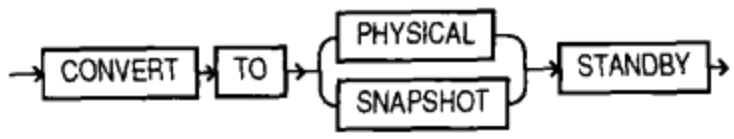


start_standby_clause::=

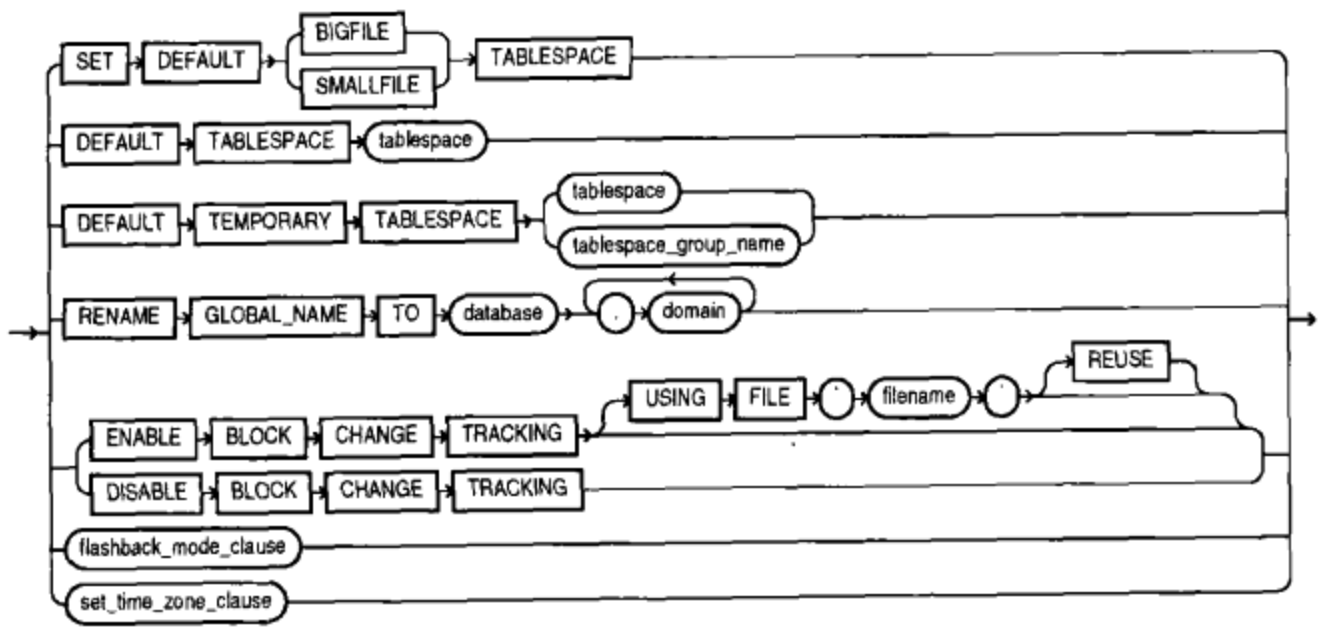


stop_standby_clause::=

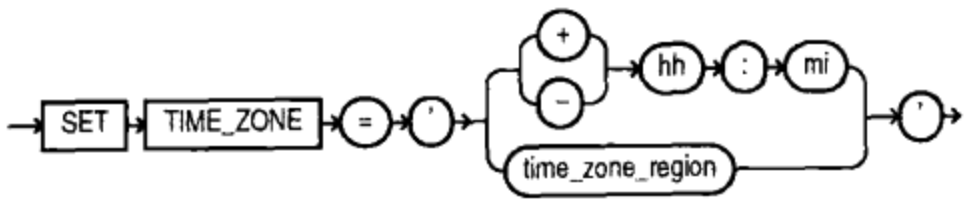
convert_standby_clause::=



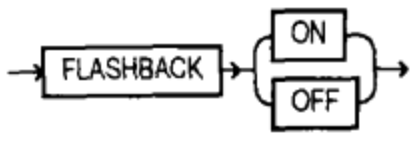
default_settings_clauses::=



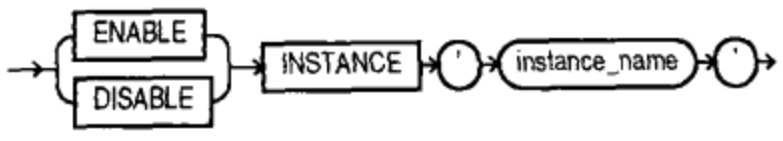
set_time_zone_clause::=



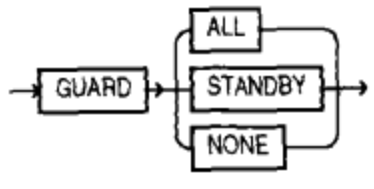
flashback_mode_clause::=



instance_clauses::=



security_clause::=



描述：要更改数据库，必须拥有 ALTER DATABASE 系统权限。

database 为数据库名，不能多于 8 个字符。

在首次创建一个数据库时，它是以 EXCLUSIVE 模式 MOUNT(安装)的，表示只有创建者可以访问它。如果已经安装了 Oracle Real Application Clusters 选项，为了允许多个实例访问数据库，可以使用 MOUNT PARALLEL。

在安装了数据库之后，可以对其执行 OPEN(打开)操作。RESETLOGS 重置重做日志，清除所有重做项。在从介质故障中进行部分恢复后，可使用该选项来还原数据库。NORESETLOGS 选项保留打开数据库时的所有设置。

可以用 RECOVER 子句恢复数据库。该命令为丢失的数据库执行介质恢复。可使用多个恢复进程将重做项应用到每个实例的数据文件中。请参阅 RECOVER。可以备份(BACKUP)一个 CONTROLFILE 到指定的文件或跟踪文件中。

STANDBY 数据库在数据库发生故障时提供自动故障转移的机制。关于备用数据库的创建和管理的详细信息，请参阅 *Oracle Database Administrator's Guide*。

RENAME 修改已有的数据库名或重做日志文件名。DBA(数据库管理员)还可以进行 CREATE DATAFILE 和 CREATE TEMPFILE 操作，并管理数据文件自动扩展后的增量。AUTOEXTEND 子句可根据需要，以 NEXT 大小的增量动态地对数据文件进行扩展，最大可扩展到 MAXSIZE(或 UNLIMITED)。可使用 RESIZE 子句增加或减少已有数据文件的大小。可以用 DATAFILE 子句使一个文件 ONLINE(联机)或 OFFLINE(脱机)。可以创建(CREATE)一个新的 DATAFILE(数据文件)来替换旧的数据文件，而不是重新创建一个由于没有备份而丢失的数据文件。

ARCHIVELOG 和 NOARCHIVELOG 定义使用重做日志文件的方式。NOARCHIVELOG 是默认方式，使用它表示将重用重做日志文件，而不在其他地方保留其内容。此方式提供介质故障(如磁盘崩溃)以外的实例恢复。ARCHIVELOG 强制对重做日志文件进行归档(一般归档到另外的磁盘或磁带)，因此可对介质故障进行恢复。该模式也支持实例恢复。

ADD LOGFILE 把重做日志文件添加到数据库中。redo_logfile_spec 指定重做日志文件的名称和大小。SIZE 是为该文件预留的字节数，如果单位为 KB，则预留的字节数为该值乘以 1024；如果单位为 MB，则乘以 1 048 576。REUSE(没有 SIZE)表示销毁具有该文件名的文件中的任何内容，并且把这个名称提供给数据库。如果文件不存在，则用具有 REUSE 的 SIZE 创建它，如果存在，则检查它的大小。如果文件不存在，则只用 SIZE 创建它；但如果存在，就返回一个错误。重做日志文件被分配给一个线程，要么用显式的 THREAD(线程)子句分配，要么分配给当前 Oracle 实例分配的线程。GROUP 是重做日志文件的一个集合。可以通过列出重做日志文件，添加一组(GROUP)重做日志文件，并且可以用一个整数为该组命名。

ADD LOGFILE MEMBER 通过指定 GROUP 整数，或者列出重做日志文件组中的所有重做日志文件，向一个已有的重做日志文件组添加新日志文件。

DROP LOGFILE 删除一个已有的重做日志文件组。DROP LOGFILE MEMBER 删除重做日志文件组中的一个或多个日志文件。

可以通过 RENAME GLOBAL_NAME 来修改数据库名。如果指定一个域，则告诉 Oracle 该数据库在网络的什么位置上。还必须从远程数据库修改对数据库的引用。

可以使用 RESET COMPATIBILITY 选项，以及初始化参数文件中的 COMPATIBLE 参数，把与 Oracle 兼容的版本恢复到先前的版本。

可以启用(ENABLE)或禁用(DISABLE)一个线程(THREAD)。PUBLIC 使该线程对于任何实例可用，而不需要特殊的线程。

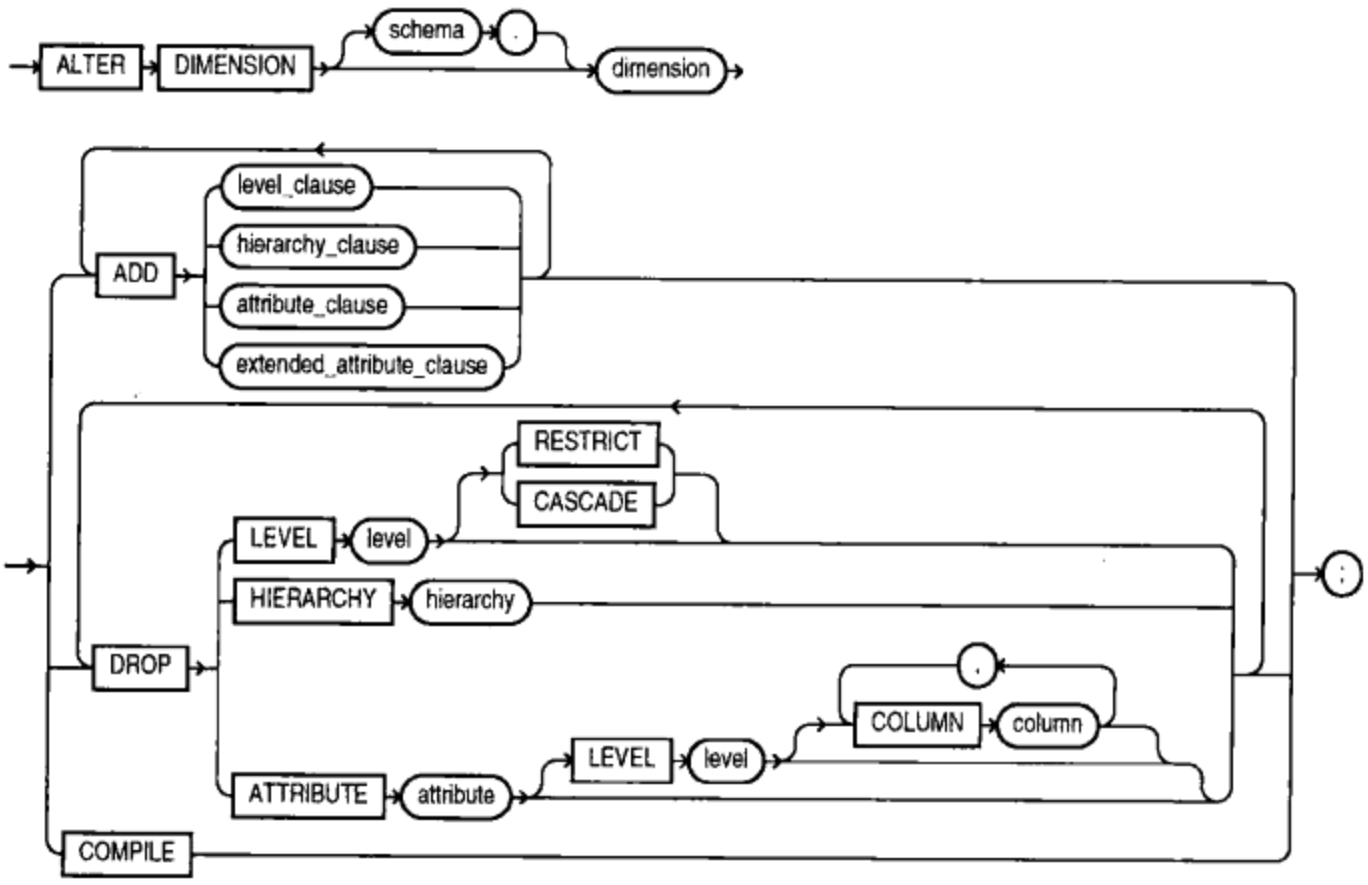
在 Oracle Database 11g 中，ALTER DATABASE 命令得到增强，简化和扩展了 managed_standby_recovery 和 standby_database_clauses 选项。supplemental_db_logging 子句包含了新语法，用来补充 PL/SQL 调用的登录。

ALTER DIMENSION

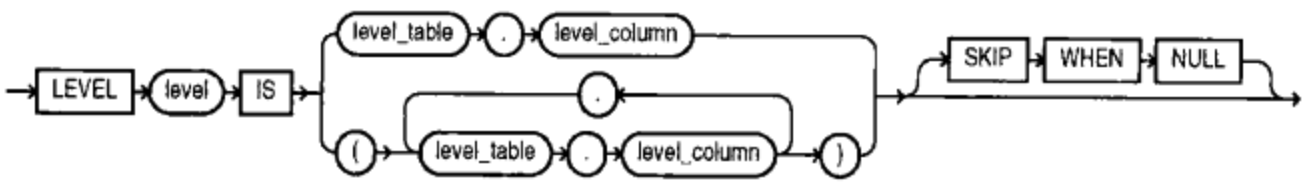
参阅：CREATE DIMENSION、DROP DIMENSION 和第 26 章。

格式：

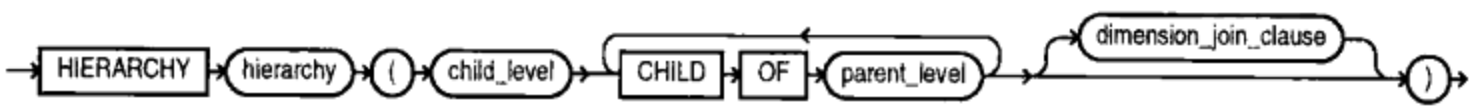
alter_dimension::=



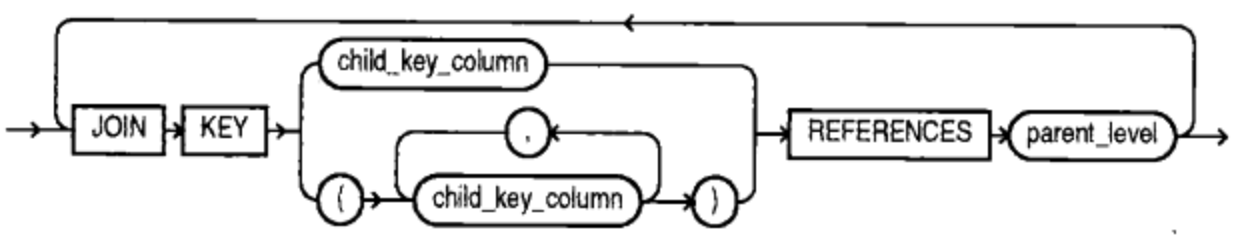
level_clause::=



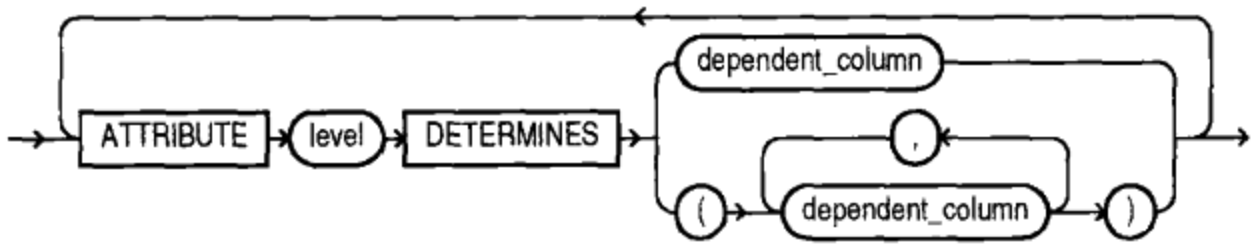
hierarchy_clause::=



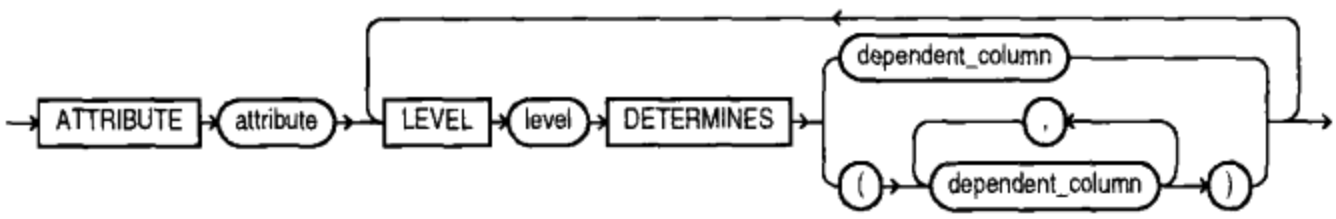
dimension_join_clause::=



attribute_clause::=



extended_attribute_clause::=



描述：ALTER DIMENSION 更改现有维度的属性、层次结构和级别。请参阅 CREATE DIMENSION。

示例：

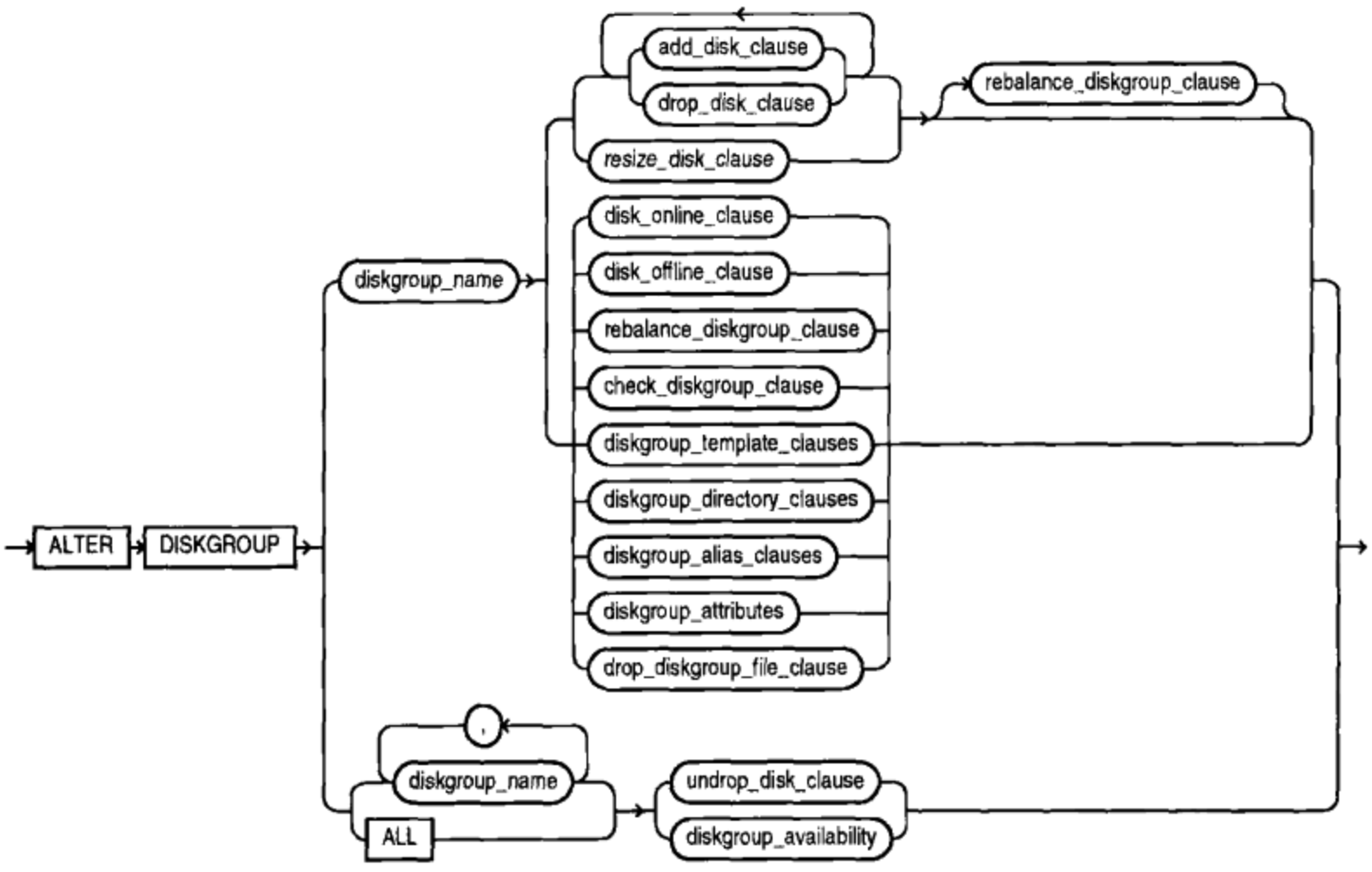
```
alter dimension TIME drop hierarchy MONTH_TO_DATE;
```

ALTER DISKGROUP

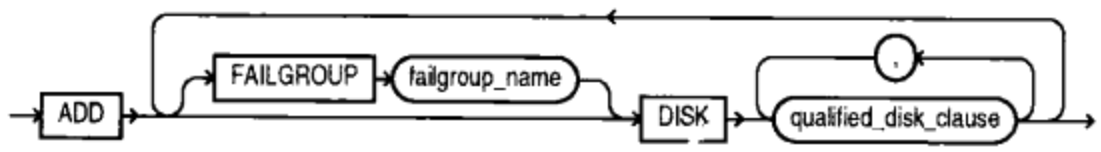
参阅：CREATE DISKGROUP、DROP DISKGROUP 和第 51 章。

格式：

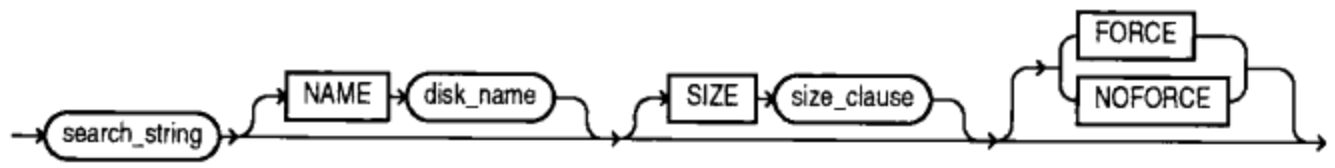
alter_diskgroup::=



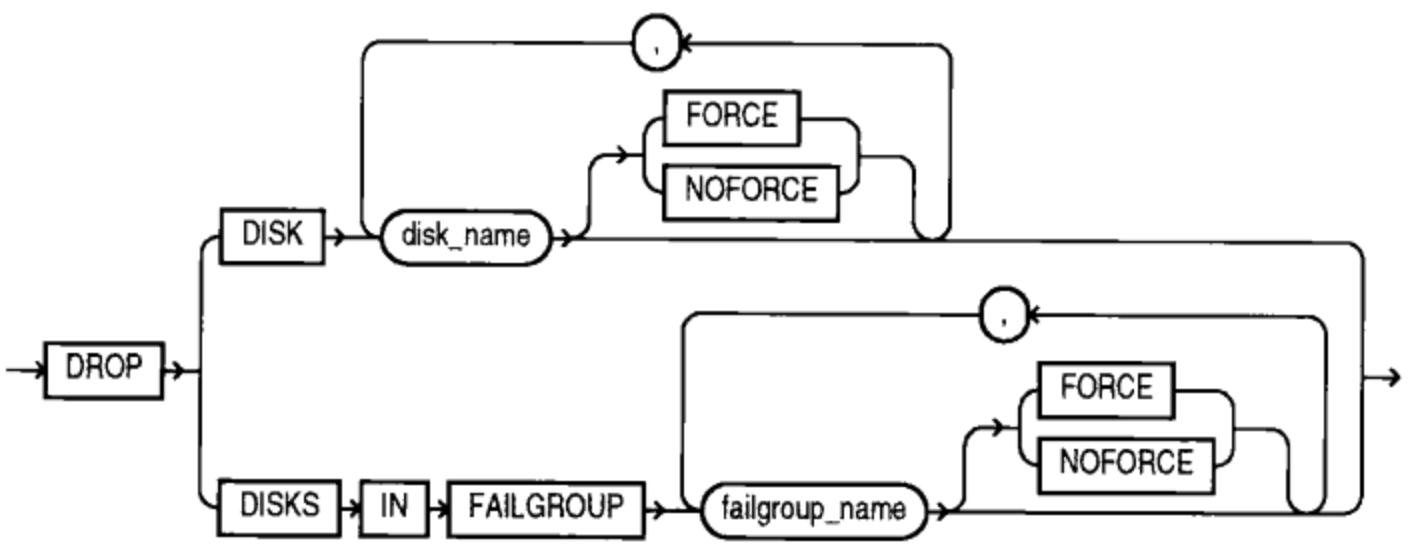
add_disk_clause::=



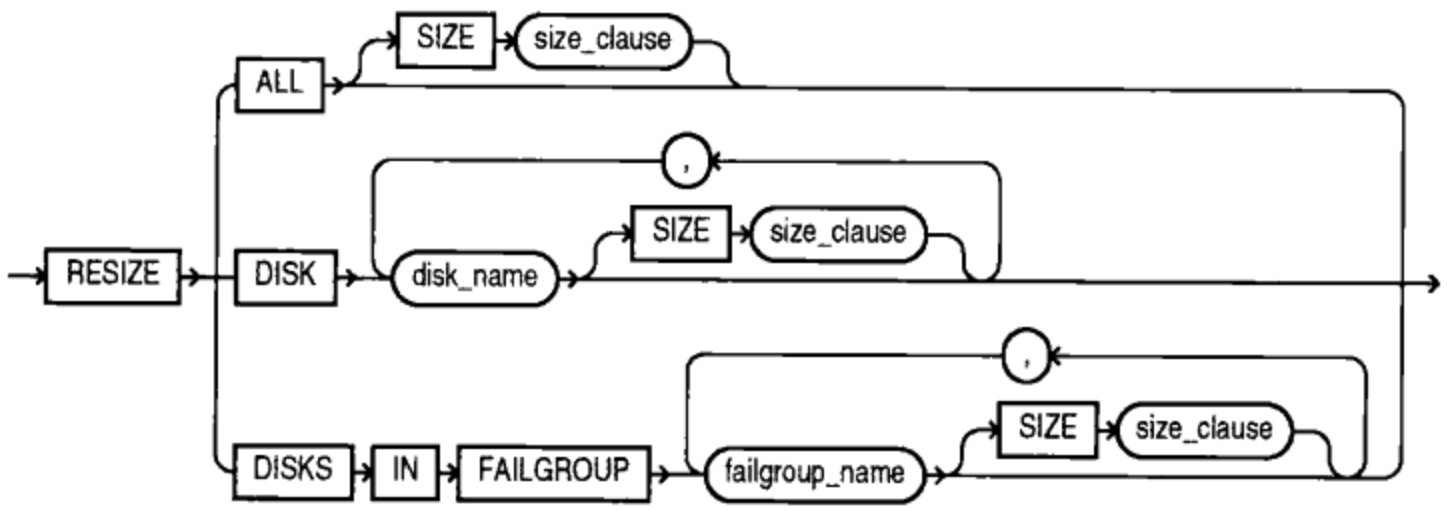
qualified_disk_clause::=



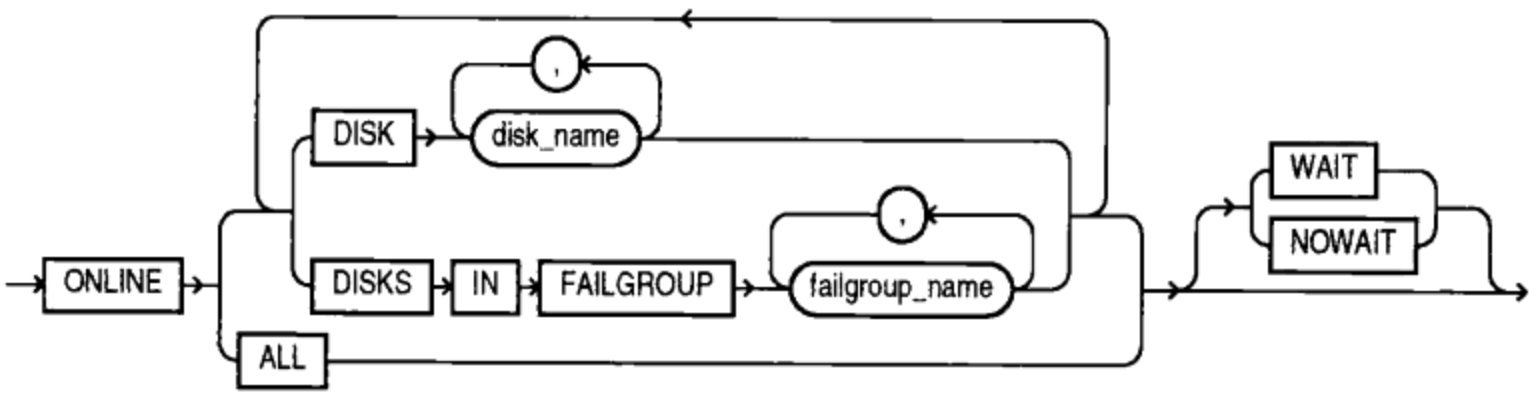
drop_disk_clauses::=



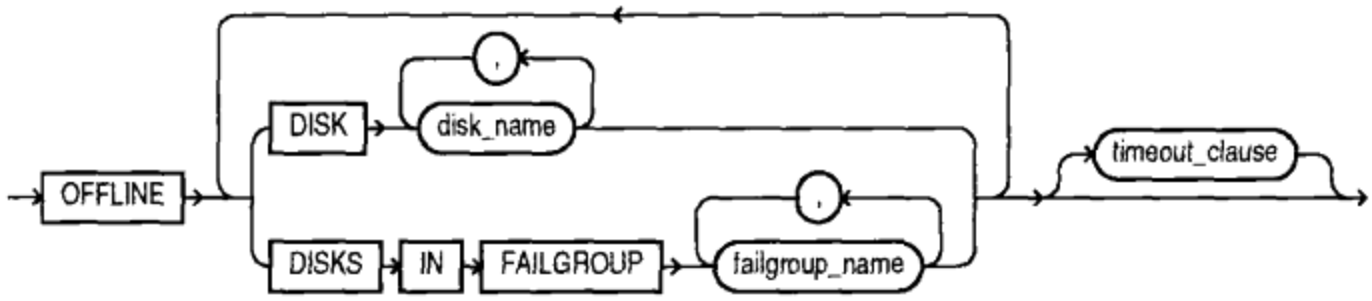
resize_disk_clauses::=



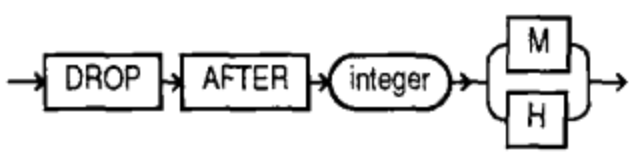
disk_online_clause::=



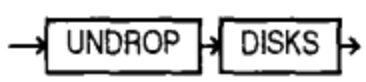
disk_offline_clause::=



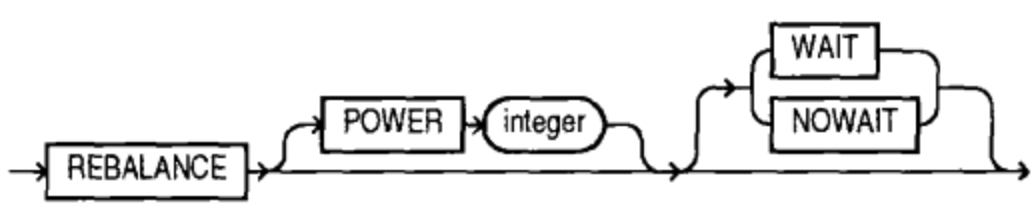
timeout_clause::=



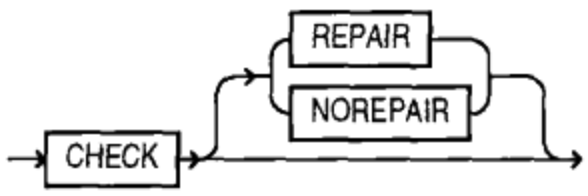
undrop_disk_clause::=



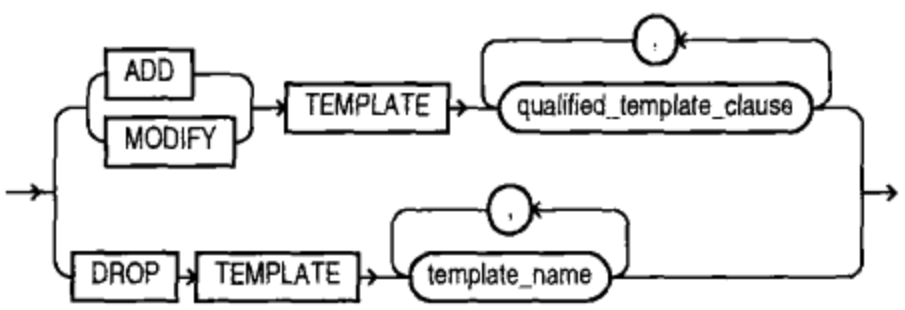
rebalance_diskgroup_clause::=



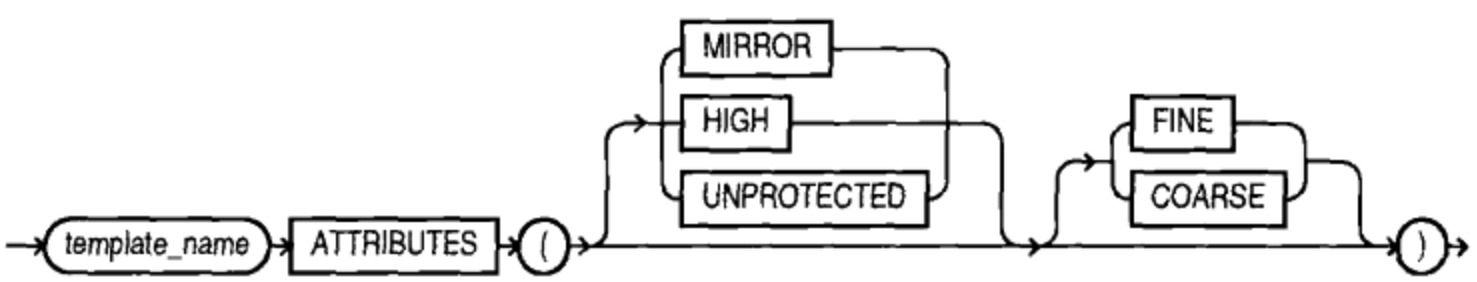
check_diskgroup_clause::=



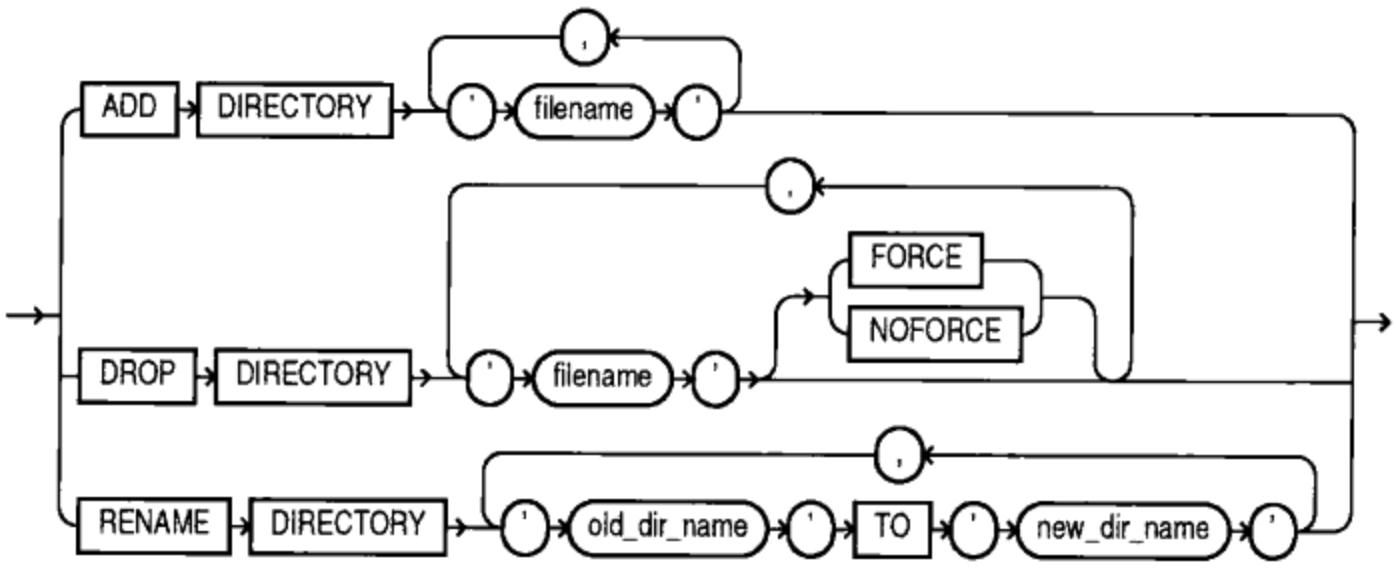
diskgroup_template_clauses::=



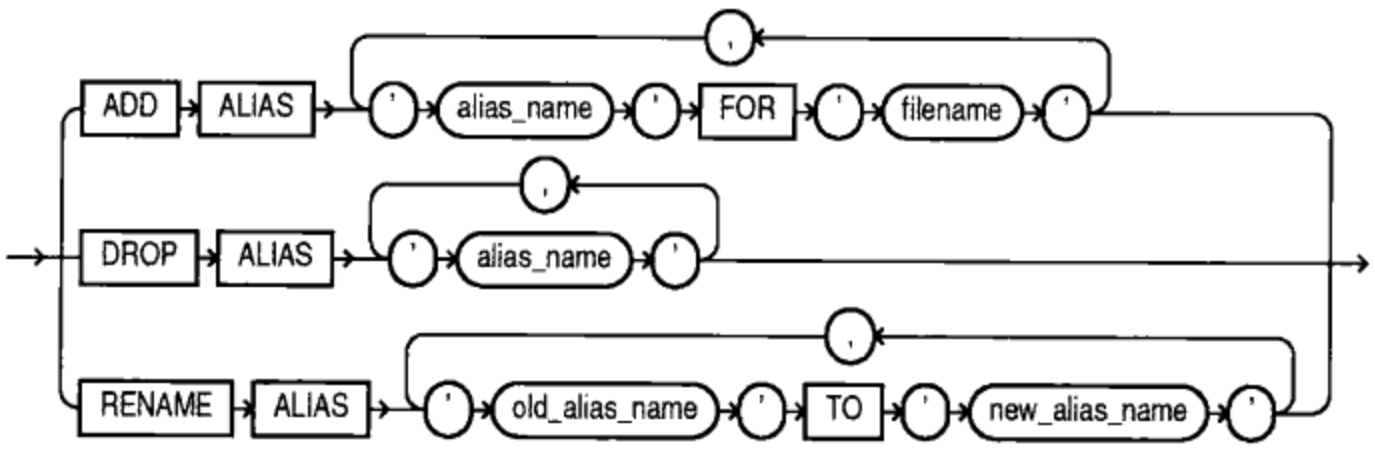
qualified_template_clause::=



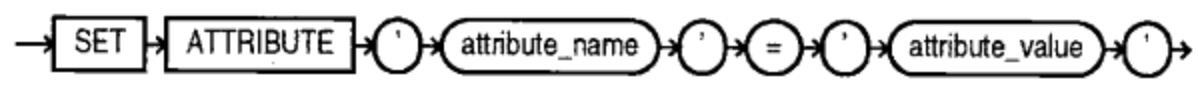
diskgroup_directory_clauses::=



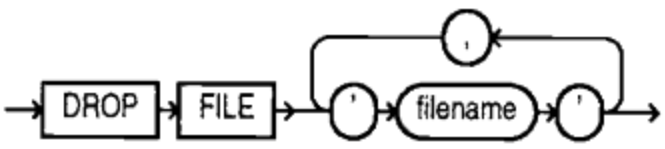
diskgroup_alias_clauses::=



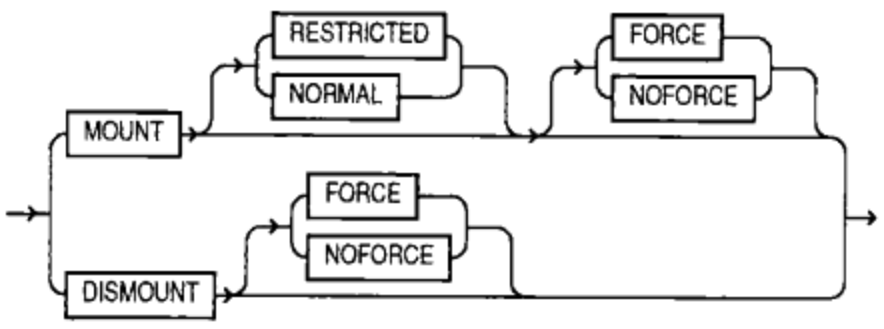
diskgroup_attributes::=



drop_diskgroup_file_clause::=



diskgroup_availability::=



描述：ALTER DISKGROUP 允许在一组磁盘或一组磁盘的某个磁盘上执行大量操作。

DISK_CLAUSES 选项包含添加、删除组中的磁盘和调整磁盘的大小。在 DISKGROUP_CLAUSES 选项中，可以重新平衡和检查磁盘组。

ALTER DISKGROUP 命令只有在安装了自动存储管理(ASM)并启动了一个 ASM 实例后才有效。

例如，从一个已存在的磁盘组中删除一个磁盘：

```
alter diskgroup DGROUP_01 drop disk DGROUP_01_0001;
```

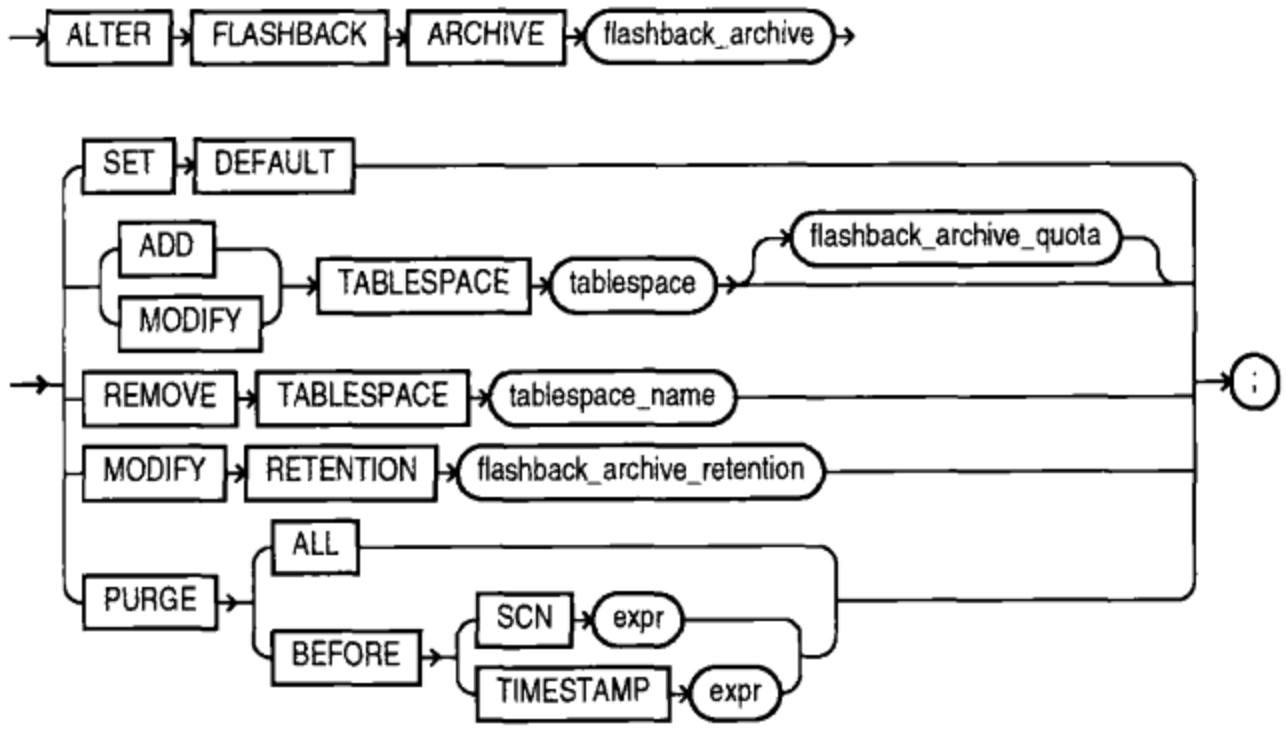
在 Oracle Database 11g 中，ALTER DISKGROUP 命令包含增强和简化版本的 check_diskgroup_clause 选项和 diskgroup_availability 选项，并添加了 disk_offline_clause 选项和 disk_online_ clauses 选项。

ALTER FLASHBACK ARCHIVE

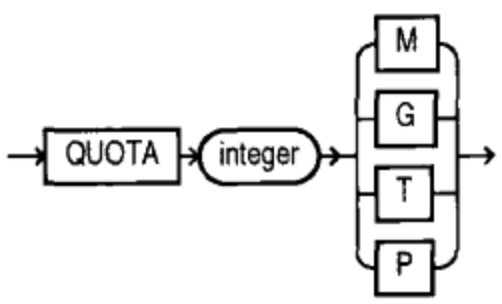
参阅：CREATE FLASHBACK ARCHIVE 和 DROP FLASHBACK ARCHIVE。

格式：

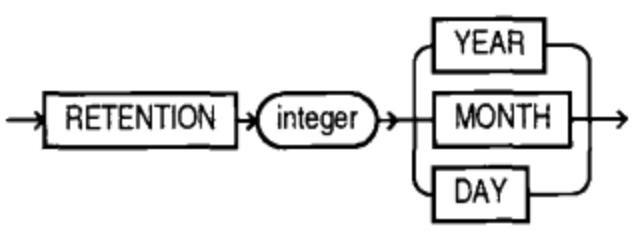
alter_flashback_archive::=



flashback_archive_quota::=



flashback_archive_retention::=



描述：使用 ALTER FLASHBACK ARCHIVE 执行下面的任务。

- 指定一个闪回数据归档作为系统默认的闪回数据归档
- 添加闪回数据归档使用的表空间
- 更改闪回数据归档使用的表空间的配额
- 删除闪回数据归档使用的表空间
- 更改闪回数据归档的保留时间
- 清除不再需要的旧数据的闪回数据归档

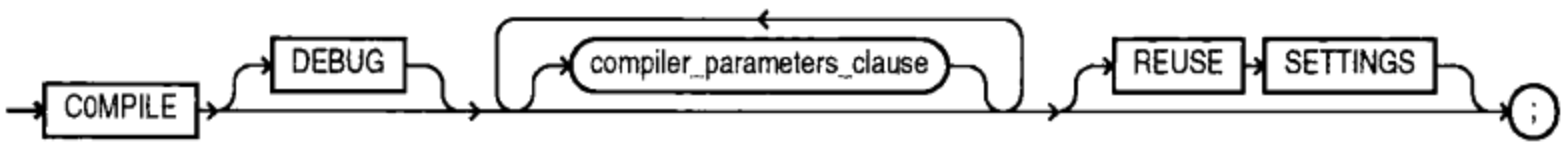
必须拥有 FLASHBACK ARCHIVE ADMINISTER 系统权限才能以任何方式改变闪回数据归档。也必须在受影响的表空间上具有适当的权限才能添加、修改或删除闪回数据归档表空间。

ALTER FUNCTION

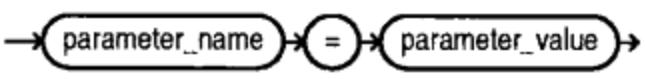
参阅：CREATE FUNCTION、DROP FUNCTION 和第 35 章。

格式：

alter_function::=



compiler_parameters_clause::=



描述：使用 ALTER FUNCTION 命令重新编译一个无效的独立的存储函数。显式重编译可以消除运行时隐式重编译的需求。使用 DEBUG 选项生成并存储 PL/SQL 调试器使用的代码。

示例：

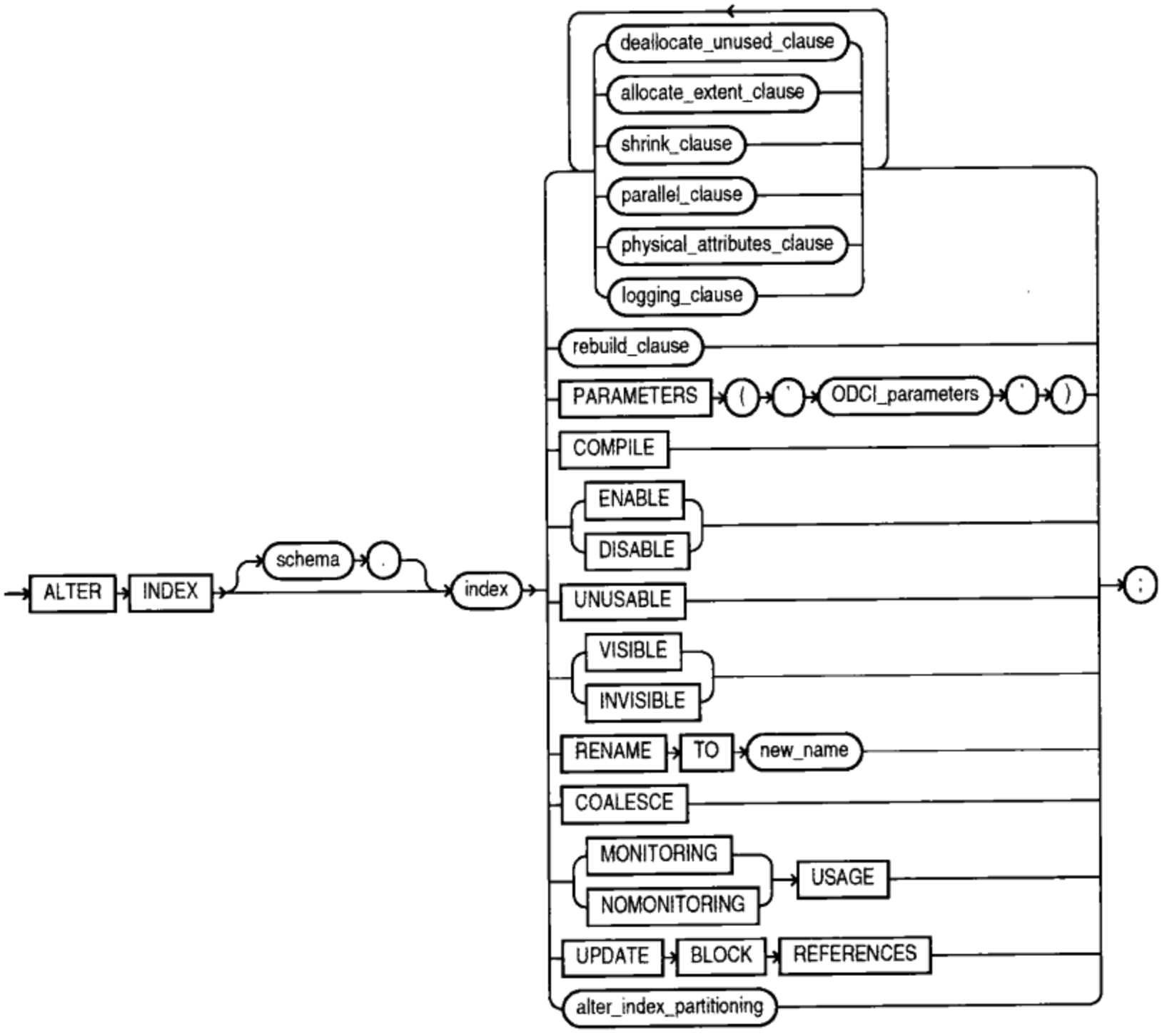
```
alter function OVERDUE_CHARGES compile;
```

ALTER INDEX

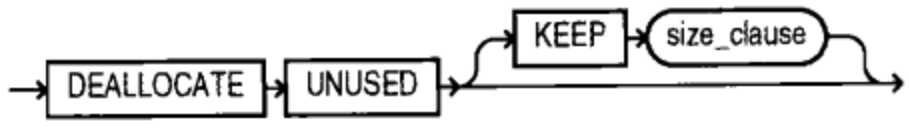
参阅：CREATE INDEX、DROP INDEX、STORAGE、第 17 章和第 46 章。

格式：

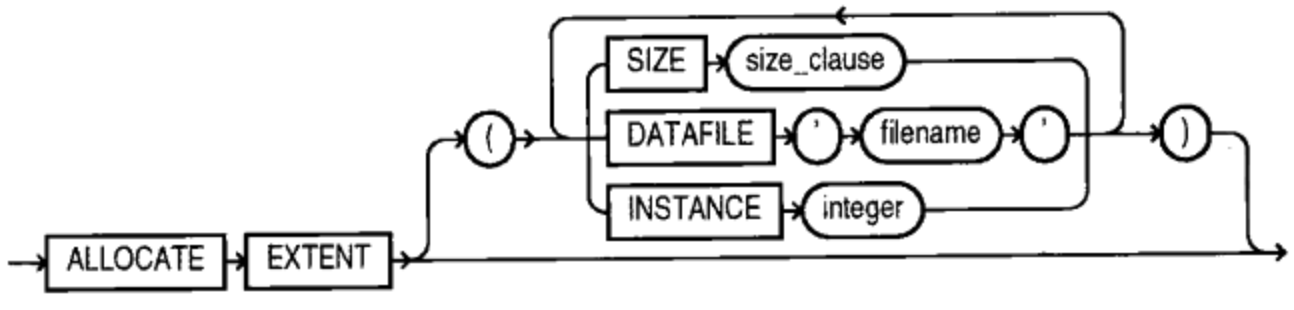
alter_index



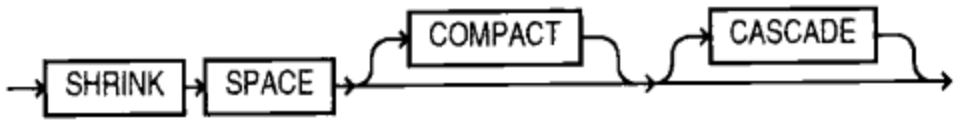
deallocate_unused_clause::=



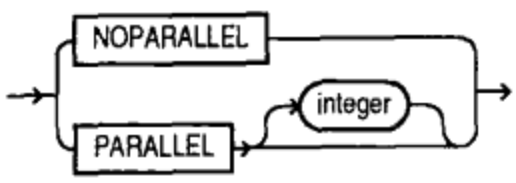
allocate_extent_clause::=



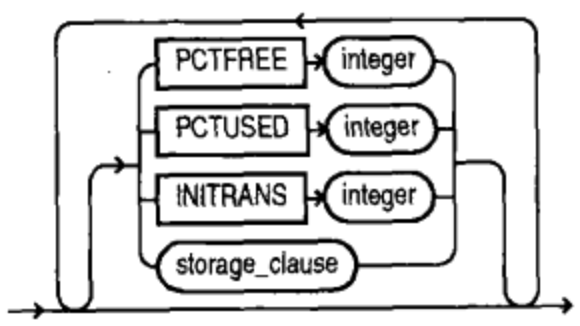
shrink_clause::=



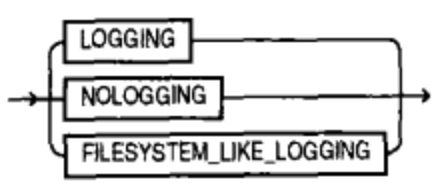
parallel_clause::=



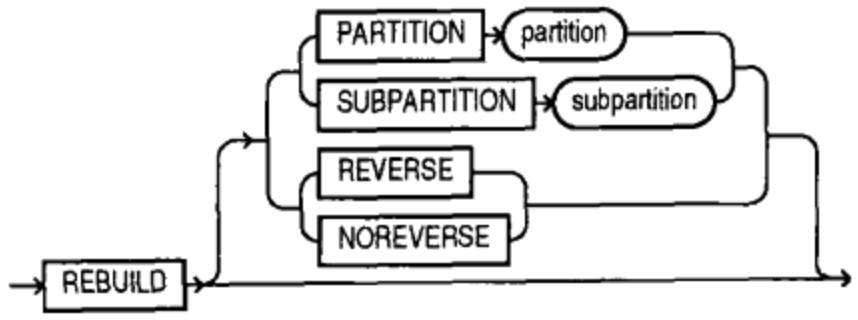
physical_attributes_clause::=

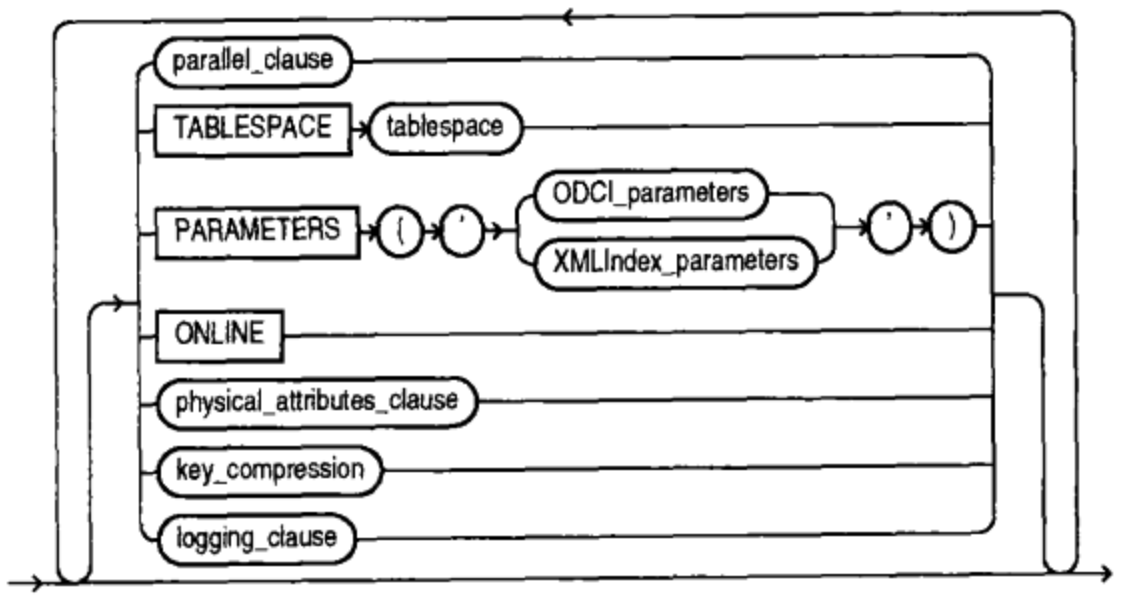


logging_clause::=

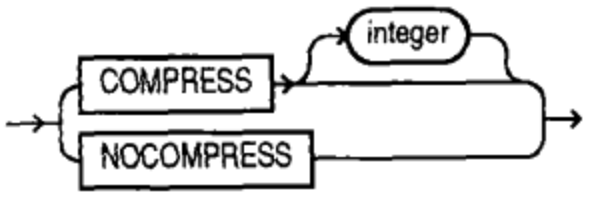


rebuild_clause::=

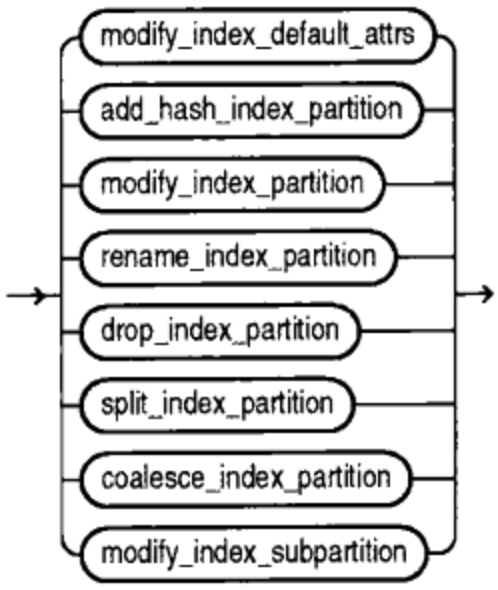




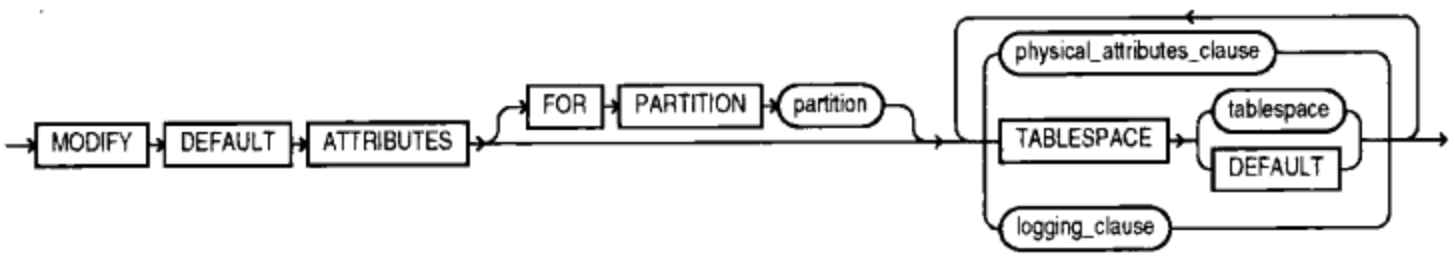
key_compression::=



alter_index_partitioning::=

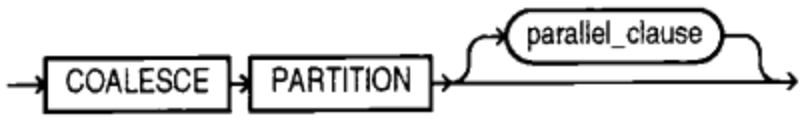


modify_index_default_attrs::=

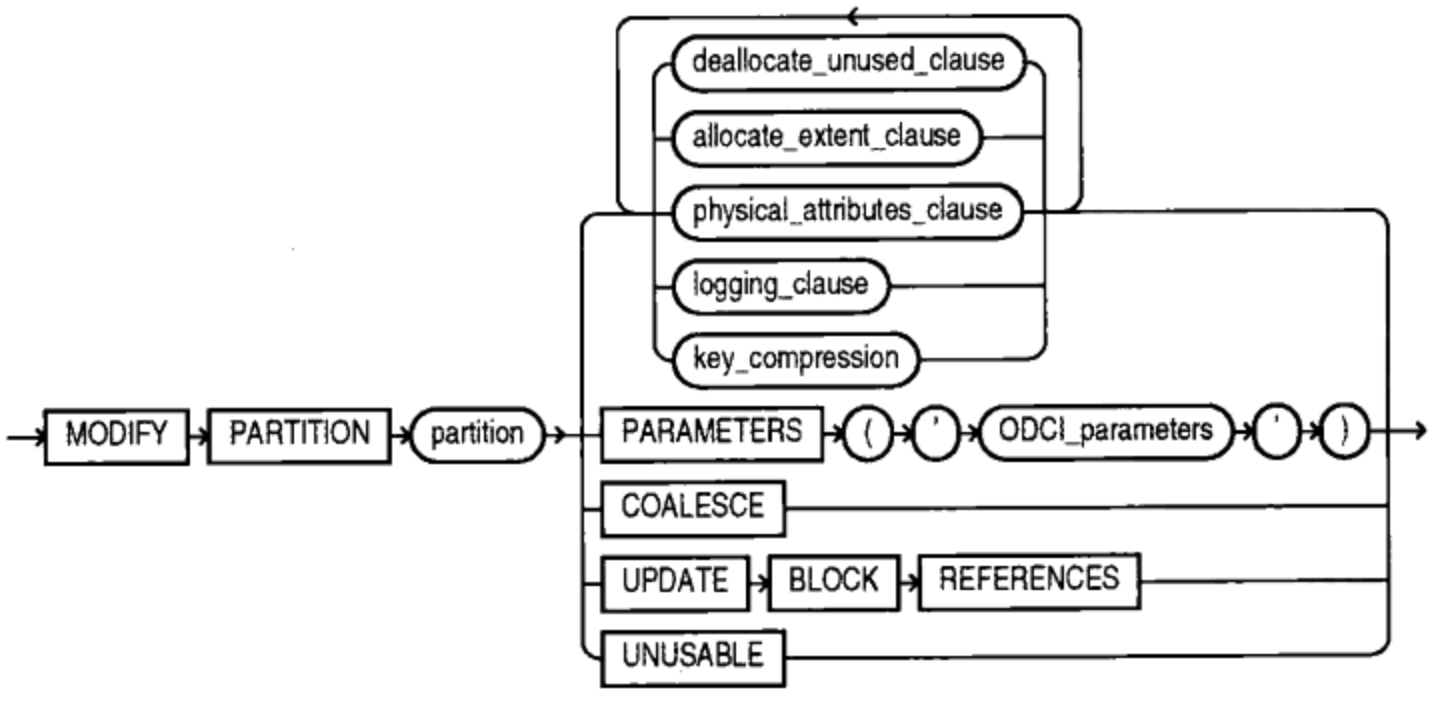


add_hash_index_partition::=

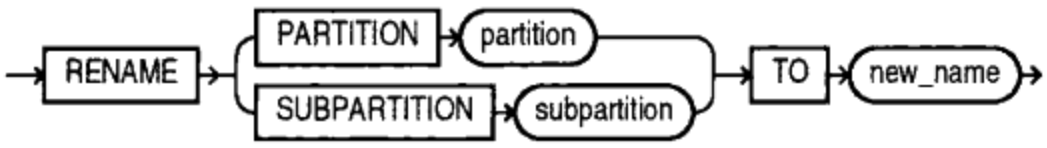
coalesce_index_partition::=



modify_index_partition::=

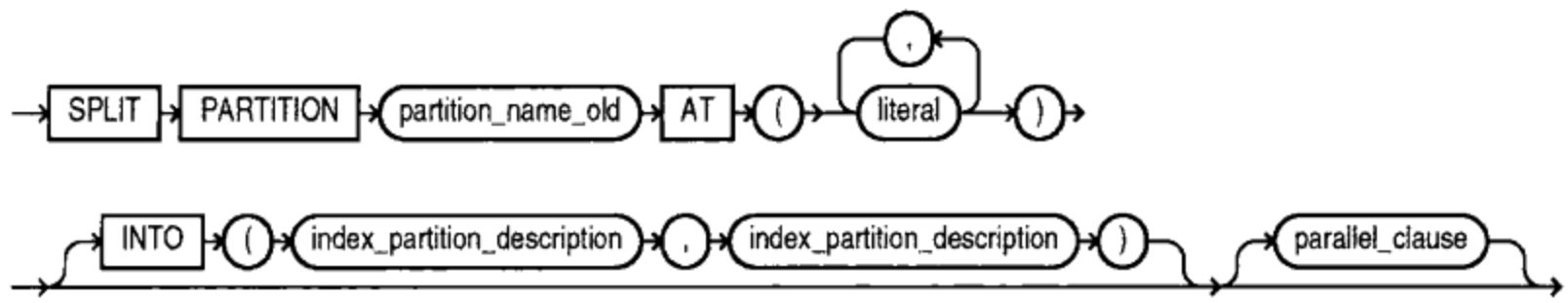


rename_index_partition::=

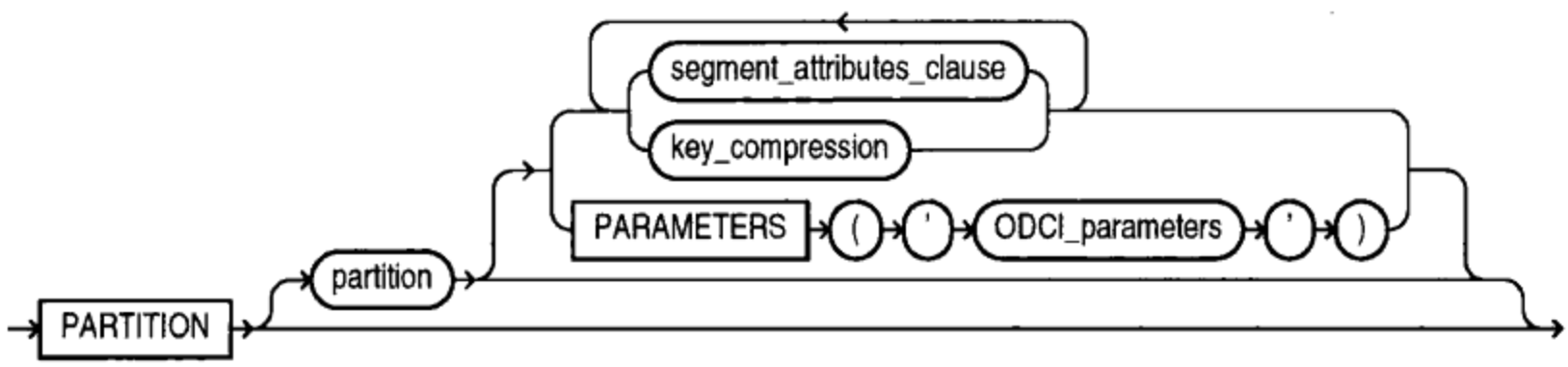


drop_index_partition::=

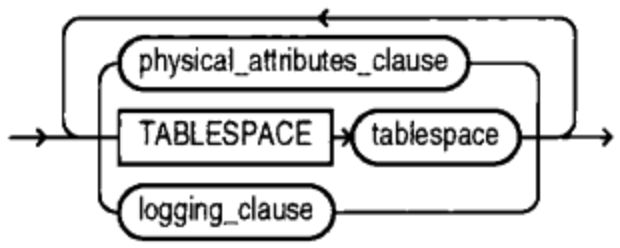
split_index_partition::=



index_partition_description::=



segment_attributes_clause::=



modify_index_subpartition::=

描述：user 是拥有希望更改的索引的用户(如果您不是该用户，则必须拥有 DBA 权限)。index 是一个已有索引的名称。关于 INITRANS 的描述，请参阅 CREATE TABLE。索引的 INITRANS 的默认值是 2。STORAGE 包含在 STORAGE 下描述的子句。要创建一个索引，必须拥有表的 INDEX 权限，而且还必须拥有这个索引或拥有 ALTER ANY INDEX 系统权限。

可以用 ALTER INDEX 命令的 REBUILD 子句修改一个已有索引的存储特性。REBUILD 使用现有索引作为新索引的基础。它支持所有的索引存储命令，如 STORAGE(用于分配盘区)、

TABLESPACE(用于将索引移动到一个新的表空间)和 INITRANS(修改初始条目数)。

可以修改、重命名、删除、拆分或重建索引分区。

COMPRESS 启动键压缩，消除非唯一索引的键列值的重复出现。integer 指定前缀的长度(要压缩的前缀列的个数)。默认时，不压缩索引。不能压缩位图索引。

COMPUTE STATISTICS 在重建索引时，生成基于优化程序的成本的统计信息。

在 Oracle 11g 中，ALTER INDEX 包含下面的选项：

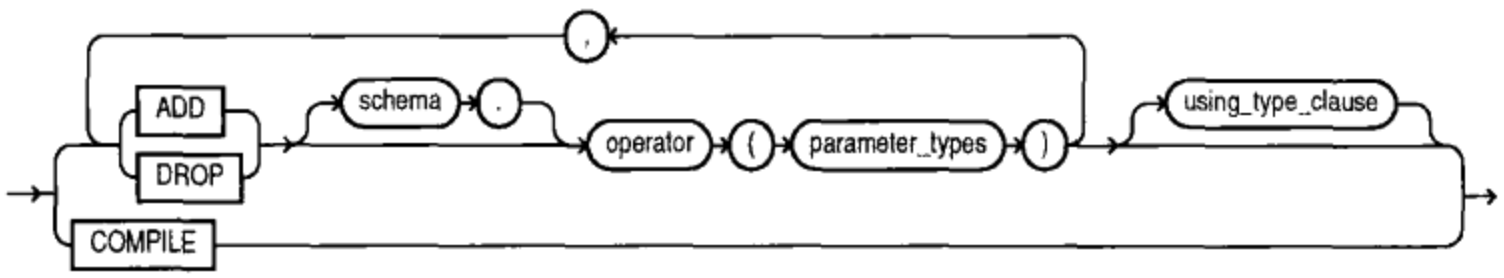
- MIGRATE 参数，它允许域索引从用户管理的存储表移动到系统管理的存储表
- INVISIBLE 参数，它允许用户修改索引，使得索引对优化程序不可见
- 参数子句，它允许用户重新构建 XMLIndex 以及域索引

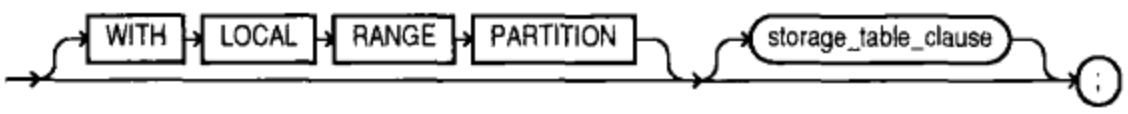
ALTER INDEXTYPE

参阅：CREATE INDEXTYPE 和 DROP INDEXTYPE。

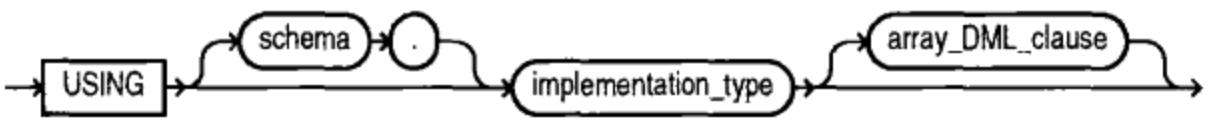
格式：

alter_indextype::=

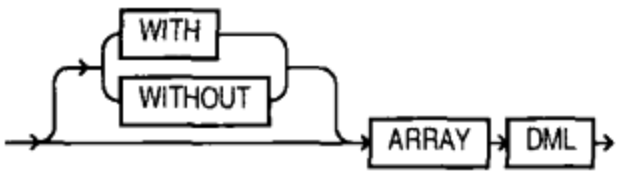


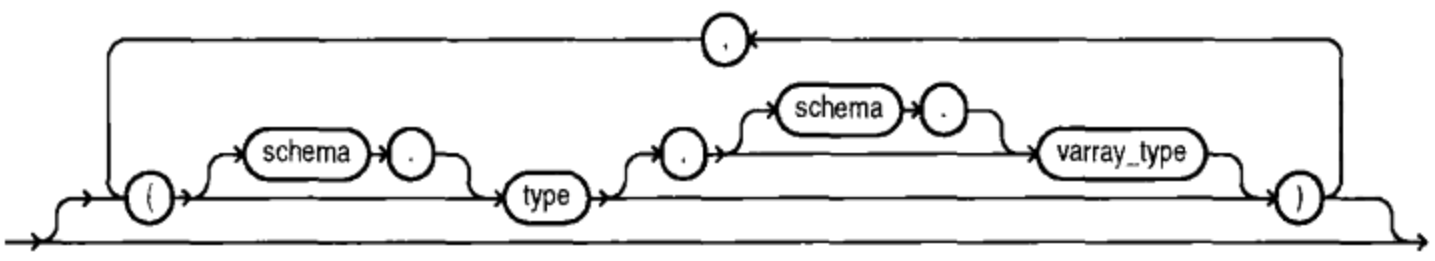


using_type_clause::=



array_DML_clause::=





storage_table_clause::=

描述：使用 ALTER INDEXTYPE 添加或删除索引类型的运算符，修改实现类型，或修改索引类型的属性。

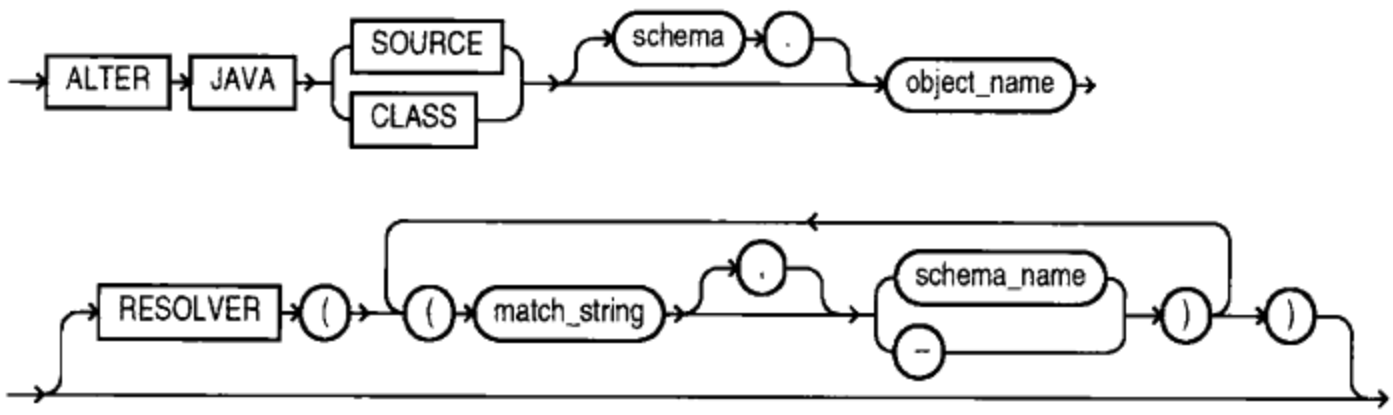
在 Oracle Database 11g 中，可以指定域索引是范围-分区索引，且由数据库管理它们的存储表和分区维护操作。

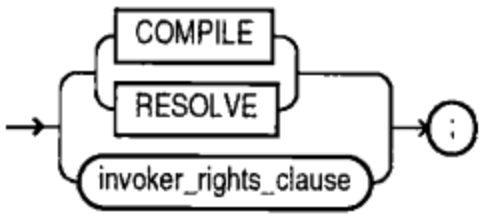
ALTER JAVA

参阅：CREATE JAVA、DROP JAVA 和第 42 章。

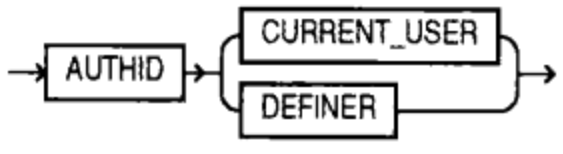
格式：

alter_java::=





invoker_rights_clause::=



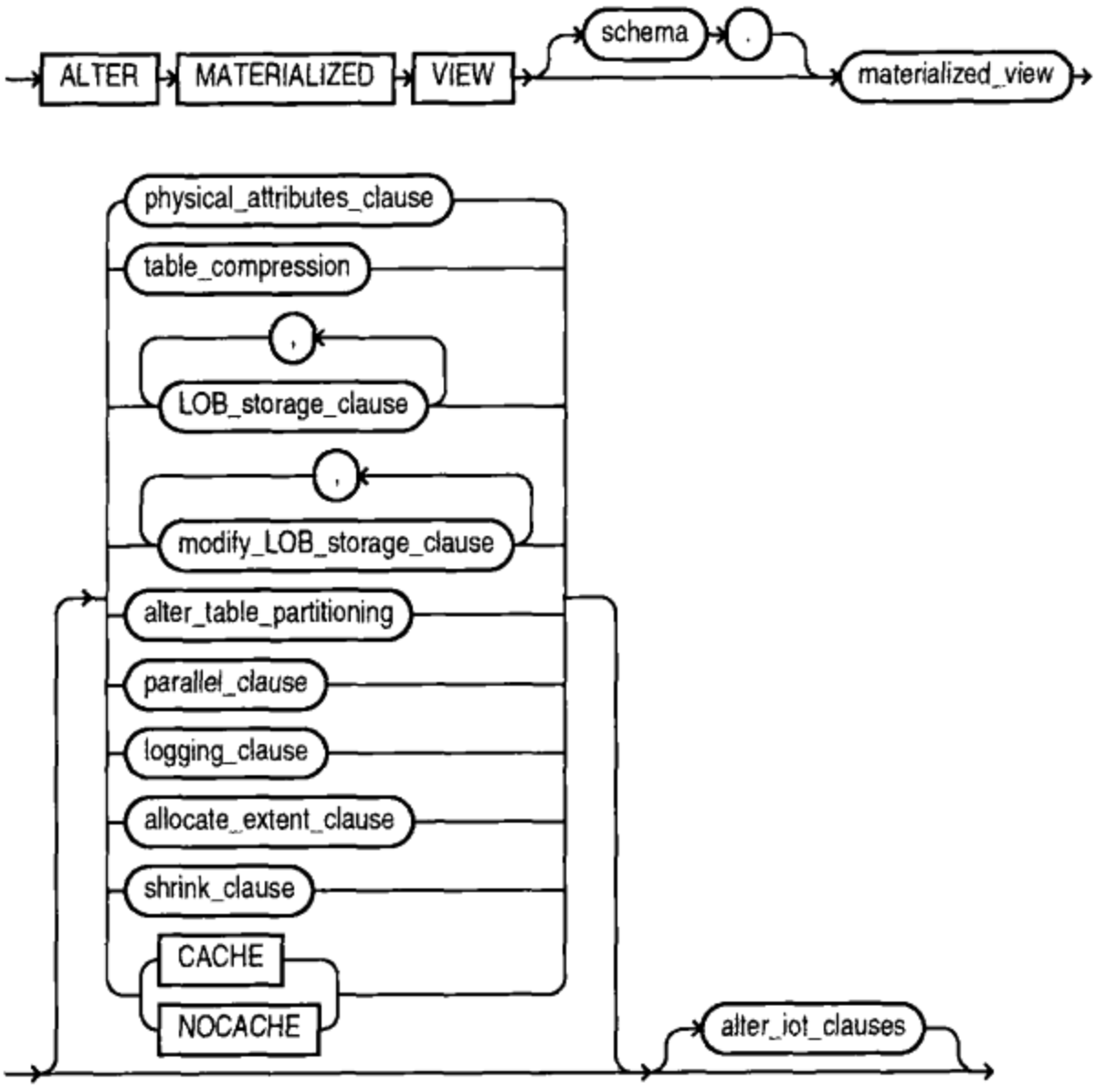
描述：ALTER JAVA 重新编译一个 Java 源模式对象。还可以用这条命令修改强制的授权(定义者或当前用户)。要更改 Java 源模式对象，必须拥有相应的源或类，或者拥有 ALTER ANY PROCEDURE 系统权限。

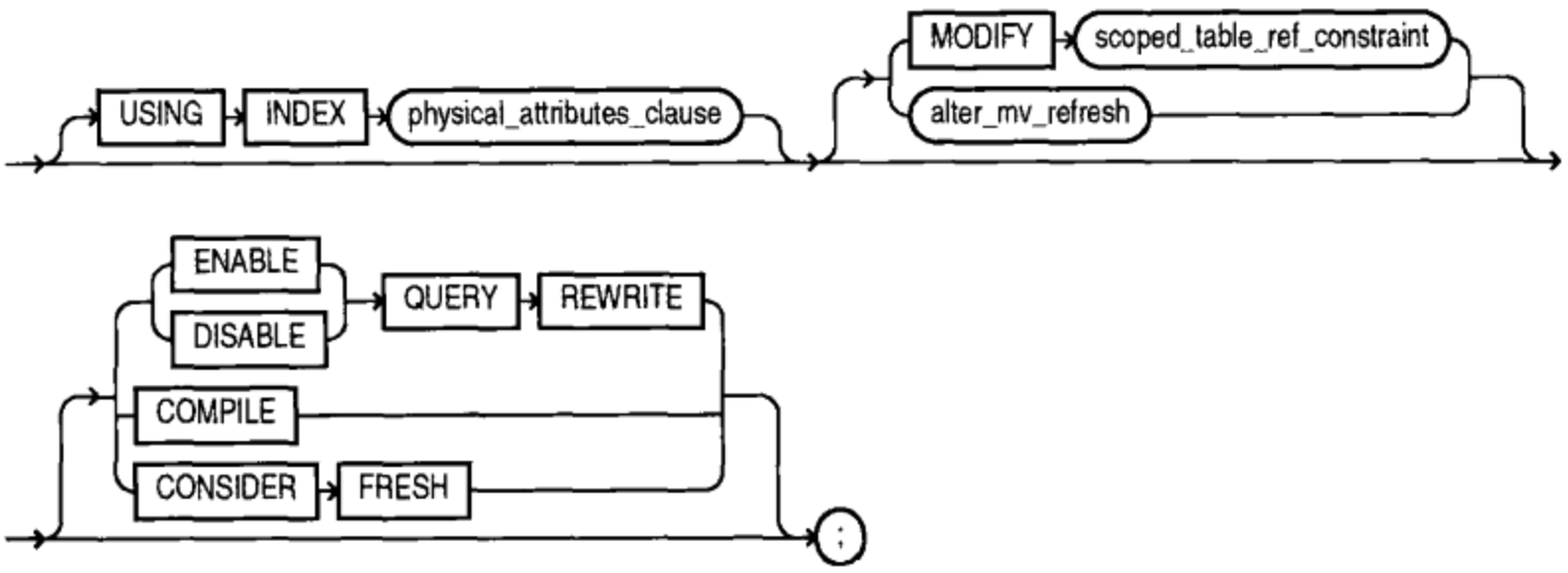
ALTER MATERIALIZED VIEW

参阅：CREATE MATERIALIZED VIEW、DROP MATERIALIZED VIEW 和第 26 章。

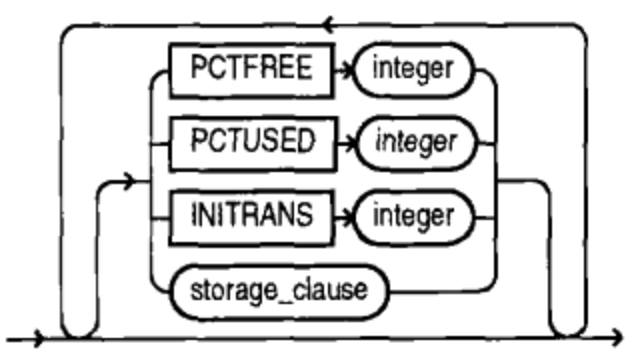
格式：

alter_materialized_view::=

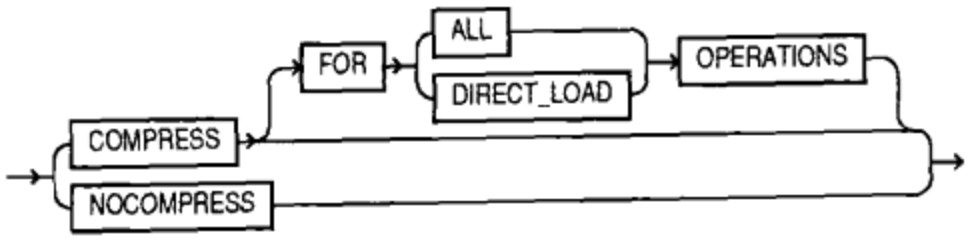




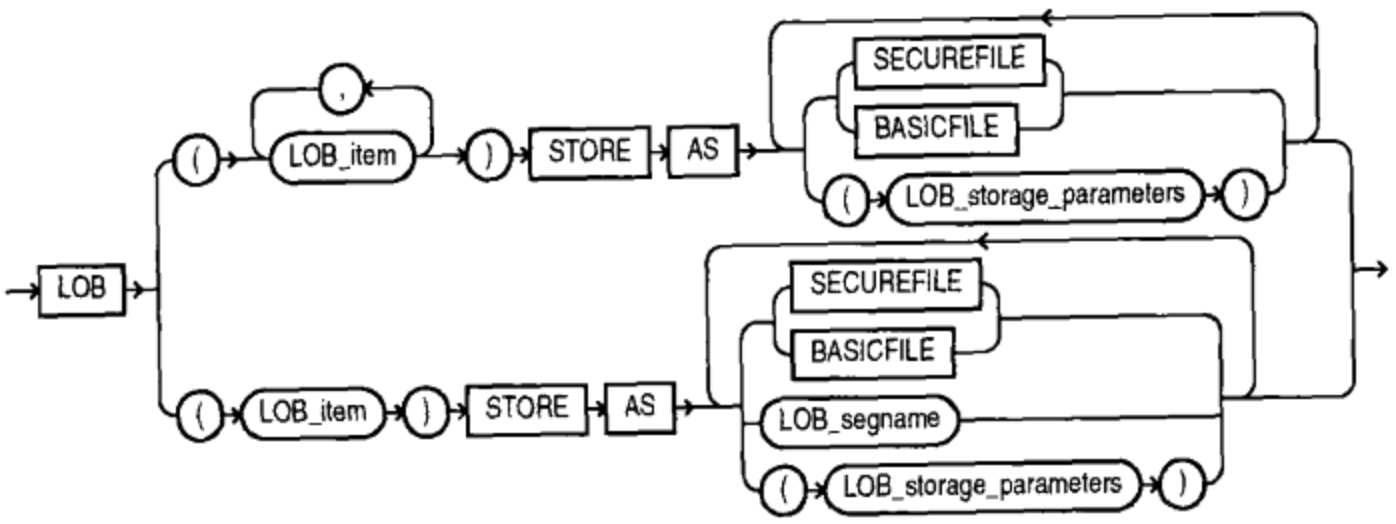
physical_attributes_clause::=



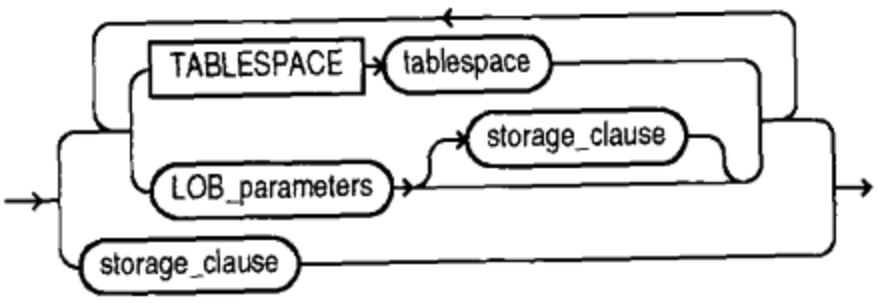
table_compression::=



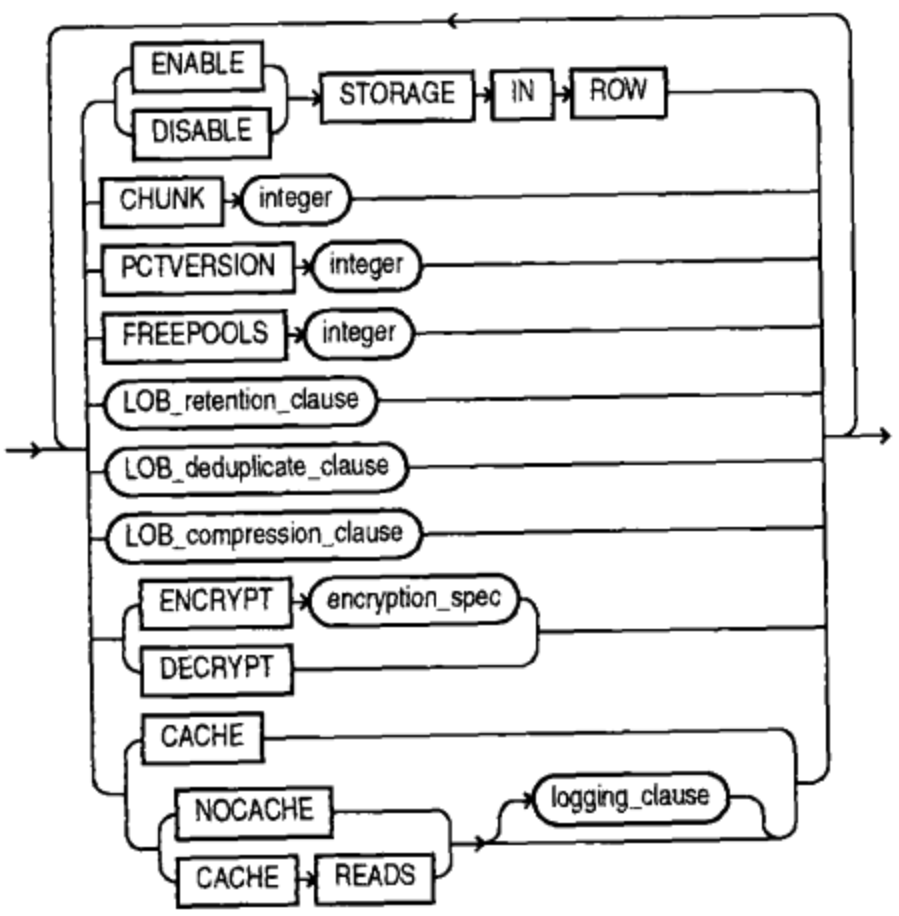
LOB_storage_clause::=



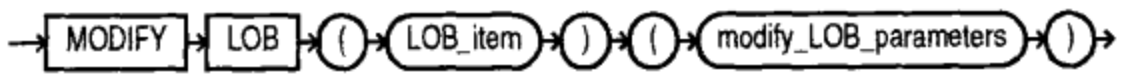
LOB_storage_parameters::=



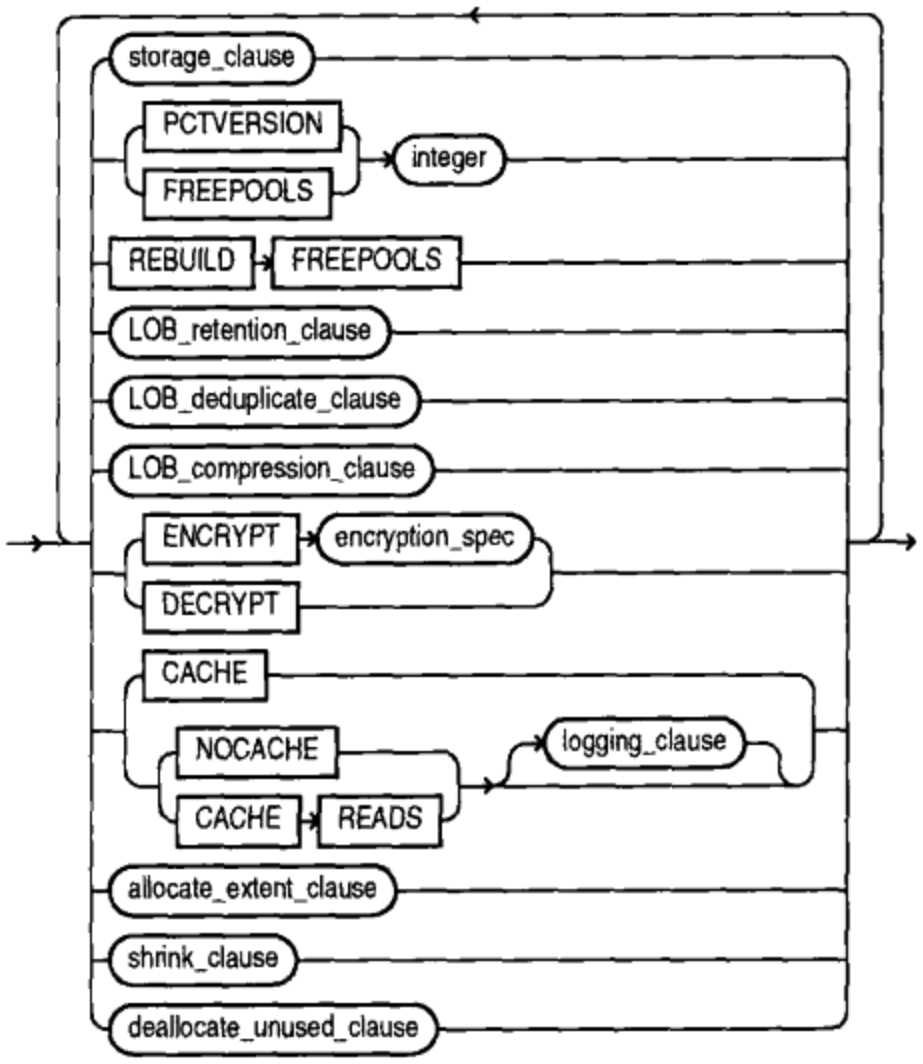
LOB_parameters::=



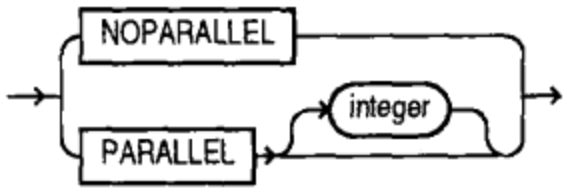
modify_LOB_storage_clause::=



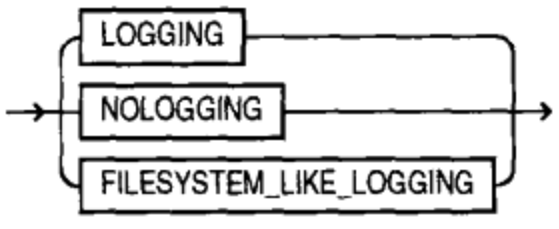
modify_LOB_parameters::=



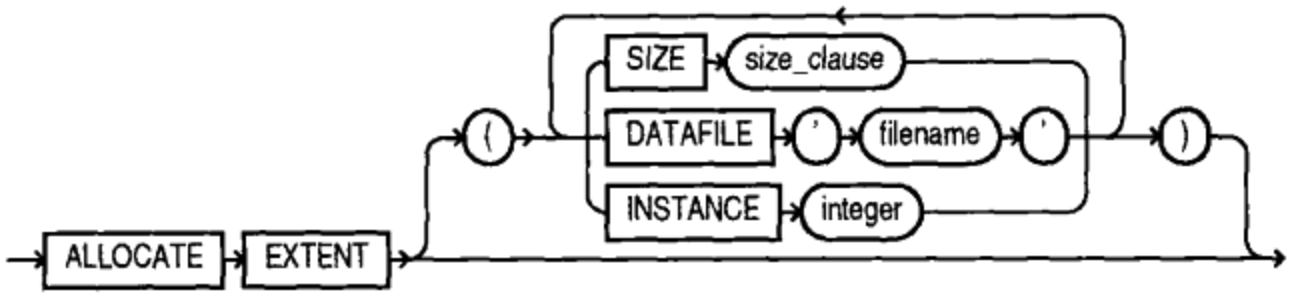
parallel_clause::=



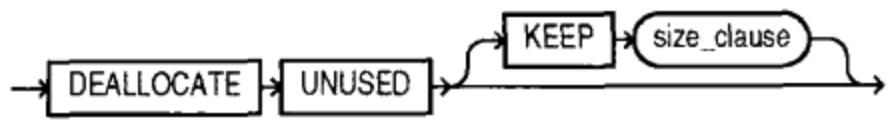
logging_clause::=



allocate_extent_clause::=

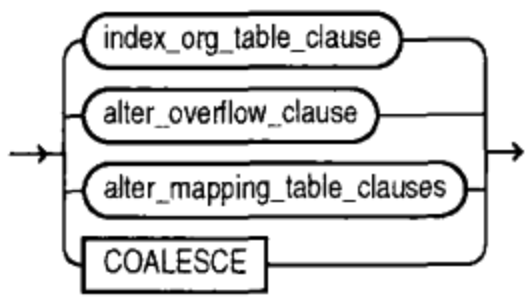


deallocate_unused_clause::=

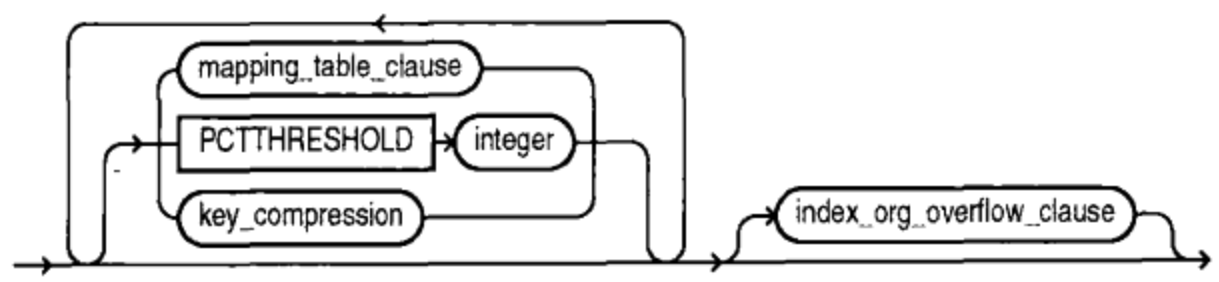


shrink_clause::=

alter_iot_clauses::=

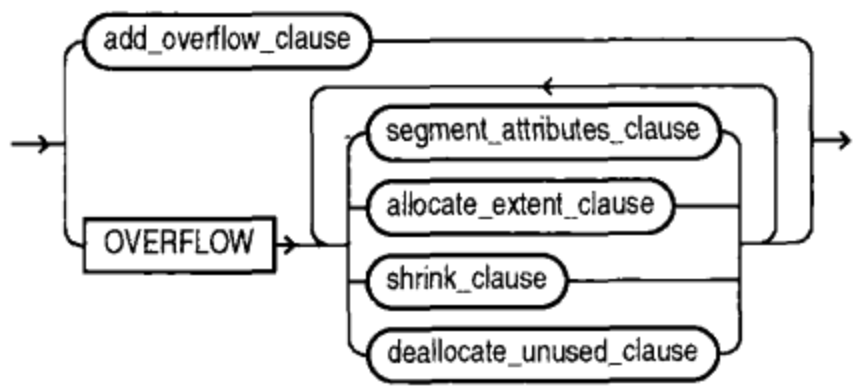


index_org_table_clause::=

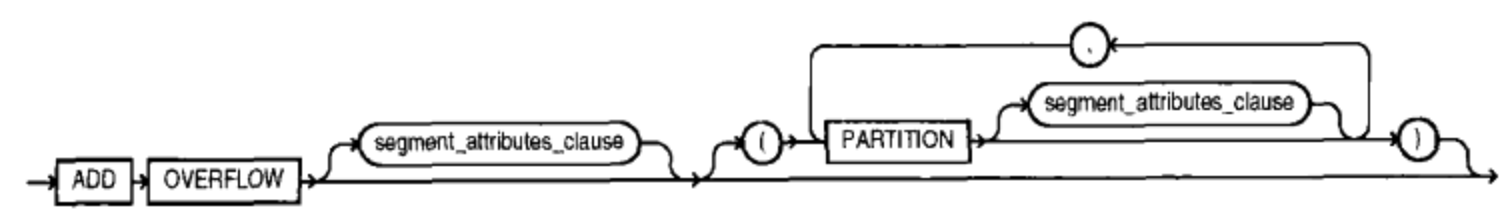


index_org_overflow_clause::=

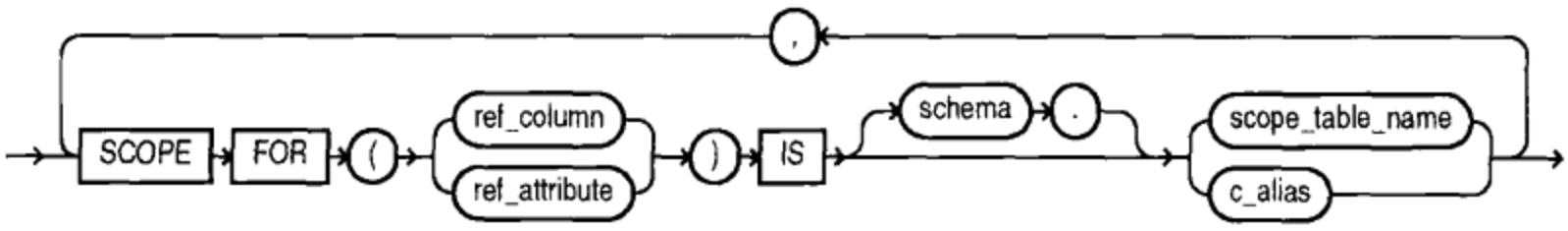
alter_overflow_clause::=



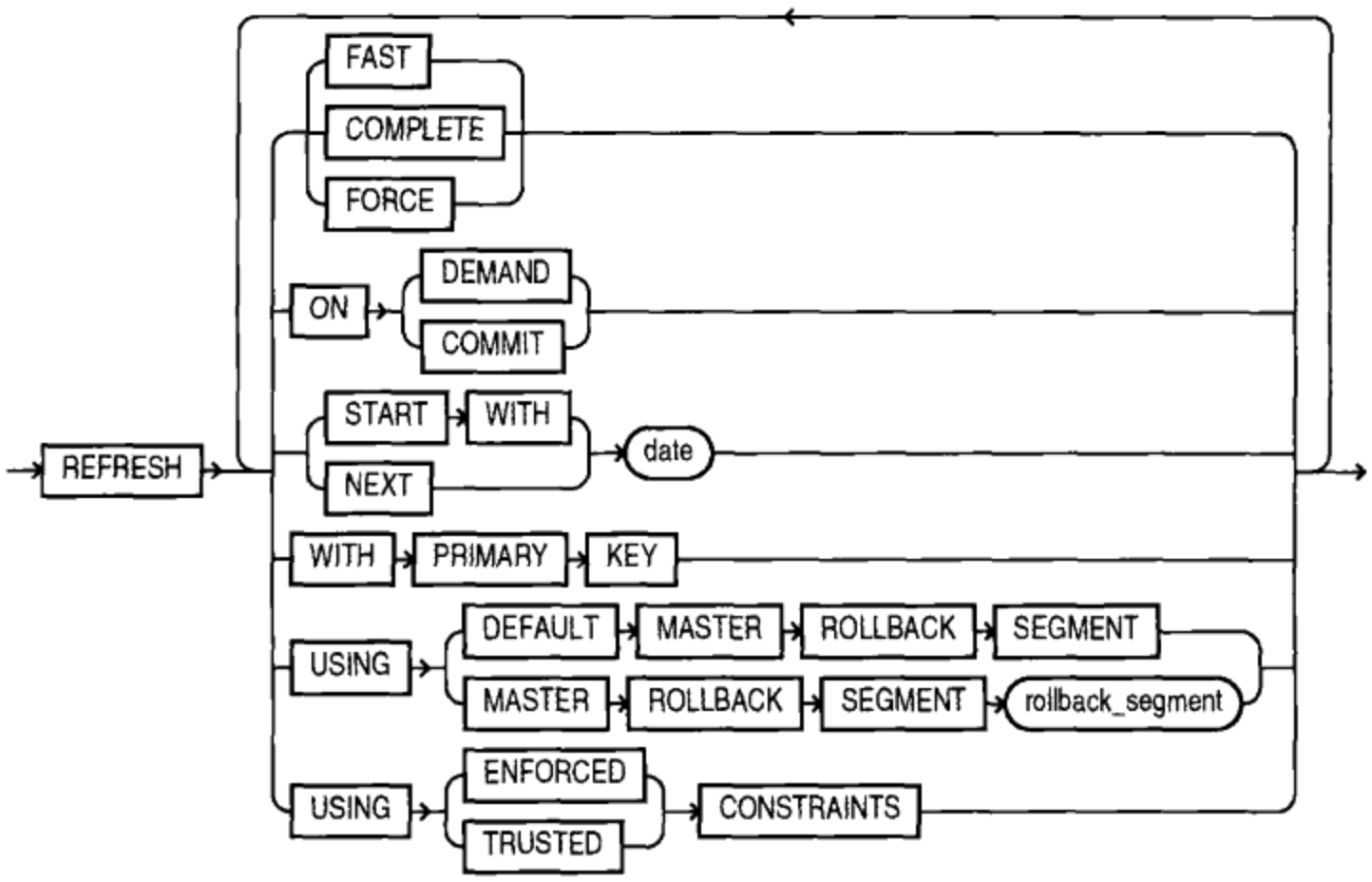
add_overflow_clause::=



scoped_table_ref_constraint::=



alter_mv_refresh::=



描述：为了向后兼容，关键字 SNAPSHOT 应该替换 MATERIALIZED VIEW。

ALTER MATERIALIZED VIEW 允许修改物化视图的存储特性或刷新模式及次数。还可以为物化视图启用或禁用查询重写。要更改一个物化视图，必须拥有该物化视图或拥有 ALTER ANY SNAPSHOT 或 ALTER ANY MATERIALIZED VIEW 系统权限。详细内容请参阅 CREATE MATERIALIZED VIEW 和第 26 章。

示例：

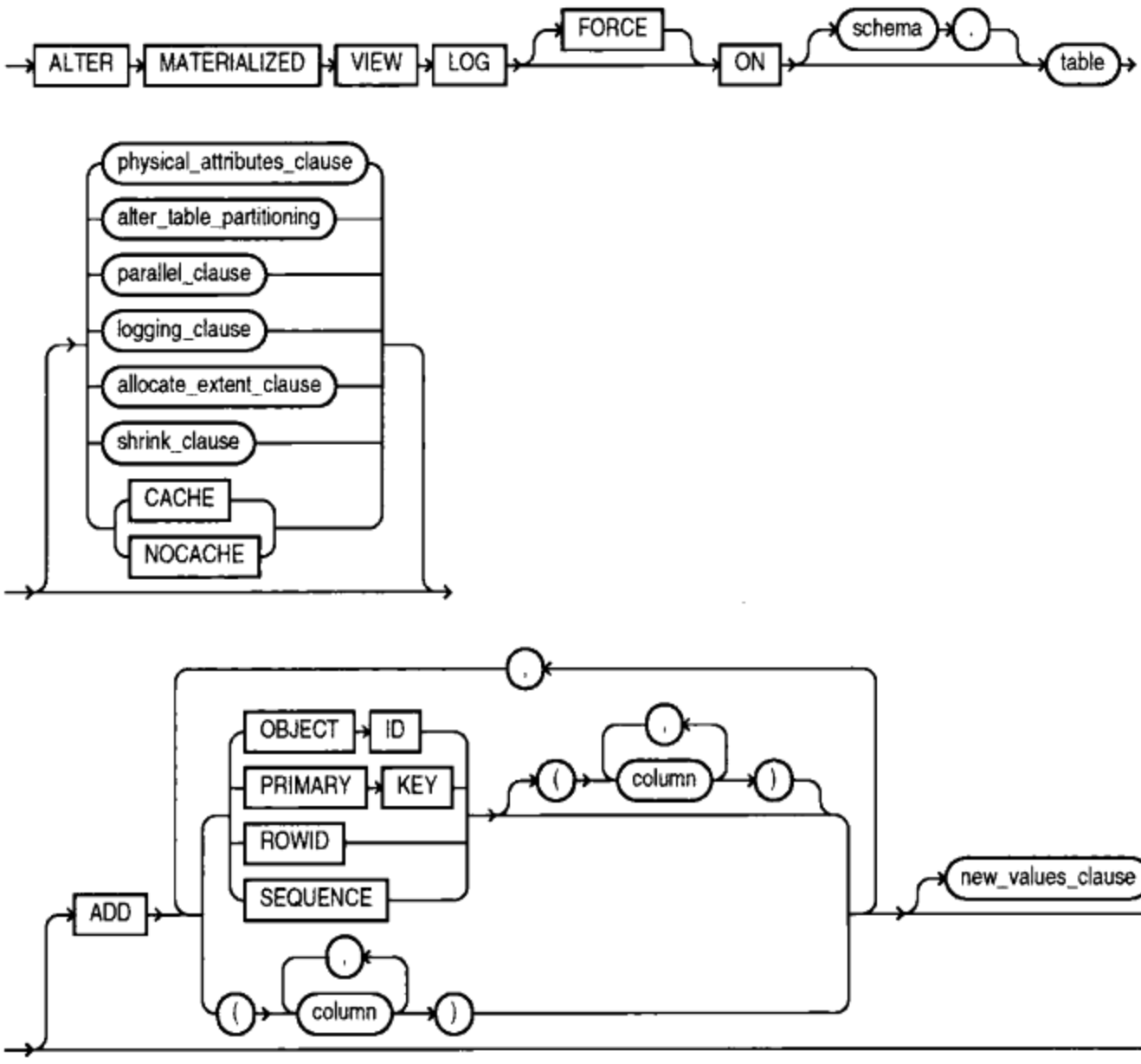
```
alter materialized view MONTHLY_SALES
enable query rewrite;
```

ALTER MATERIALIZED VIEW LOG

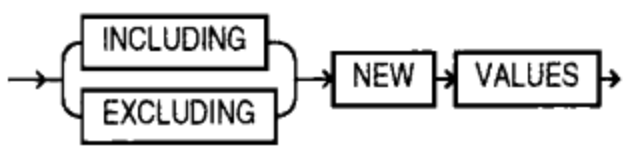
参阅：CREATE MATERIALIZED VIEW、CREATE MATERIALIZED VIEW LOG、DROP MATERIALIZED VIEW LOG 和第 26 章。

格式：

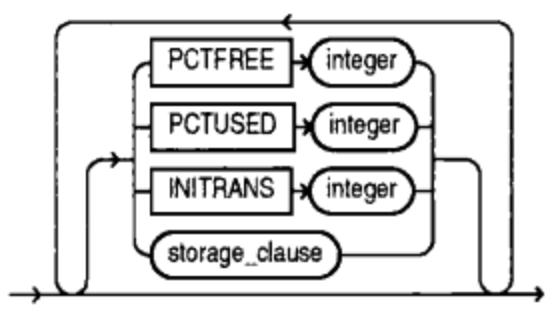
alter_materialized_view_log::=



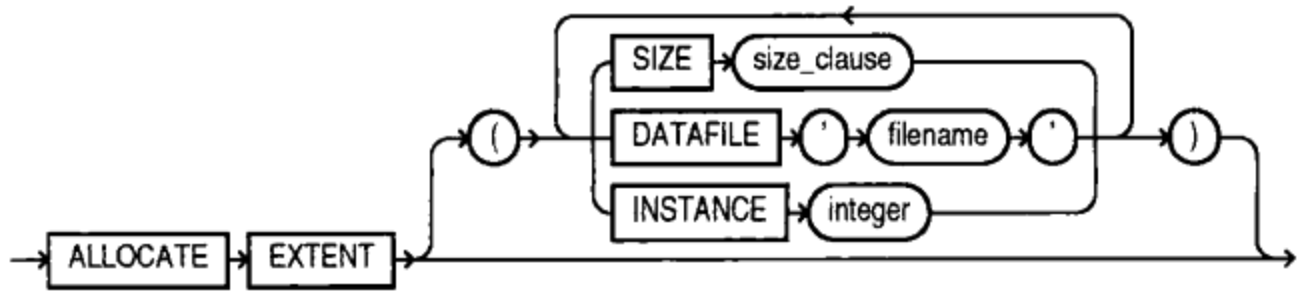
new_values_clause::=



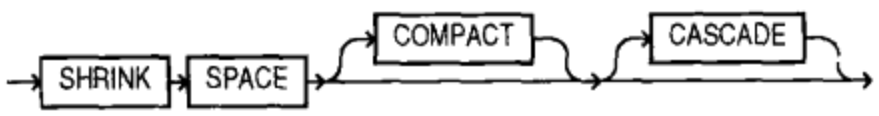
physical_attributes_clause::=



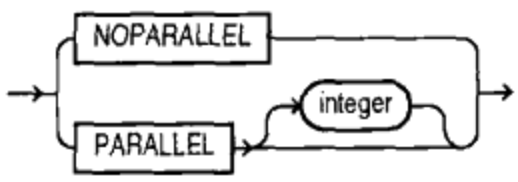
allocate_extent_clause::=



shrink_clause::=



parallel_clause::=



描述：为了向后兼容，关键字 SNAPSHOT 应该替换 MATERIALIZED VIEW。

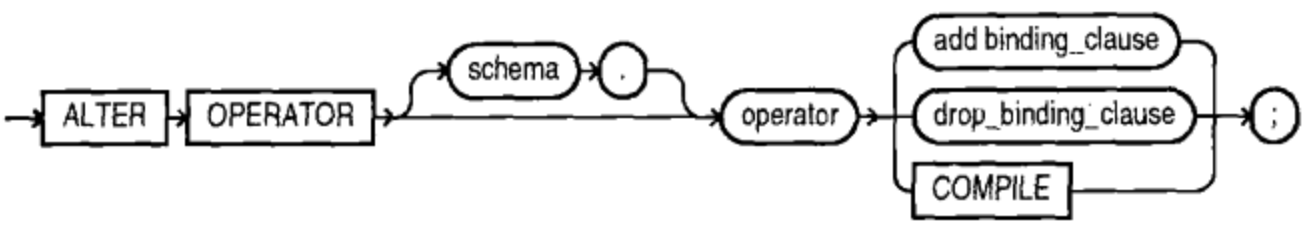
ALTER MATERIALIZED VIEW LOG 允许修改物化视图日志的存储特性。要更改日志，必须拥有日志表的 ALTER 权限或者拥有 ALTER ANY TABLE 系统权限。详细内容请参阅 CREATE MATERIALIZED VIEW LOG 和第 26 章。

ALTER OPERATOR

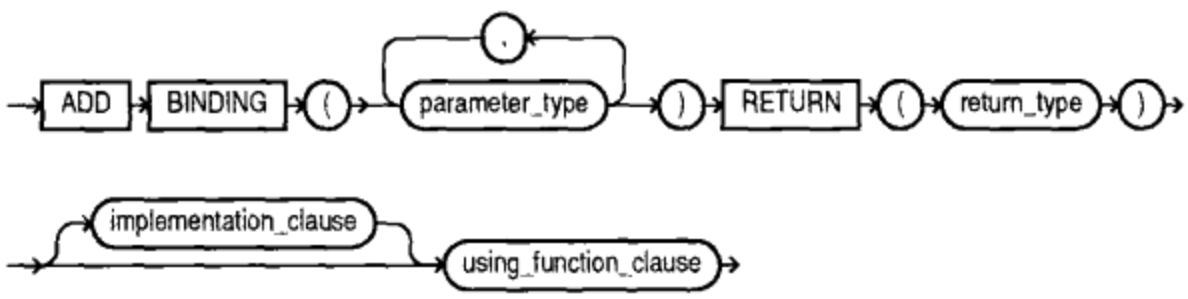
参阅：CREATE OPERATOR 和 DROP OPERATOR。

格式：

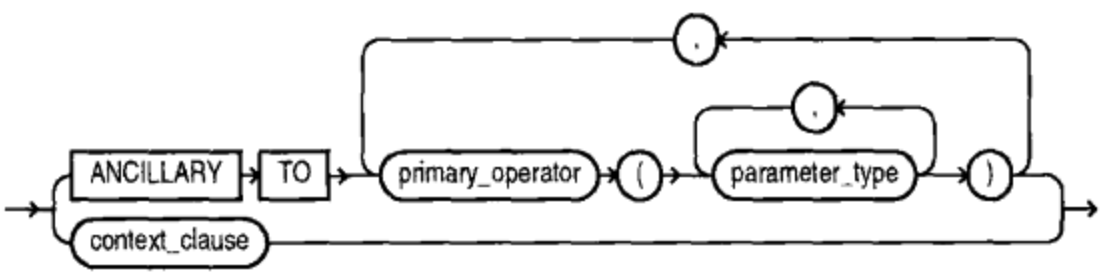
alter_operator::=



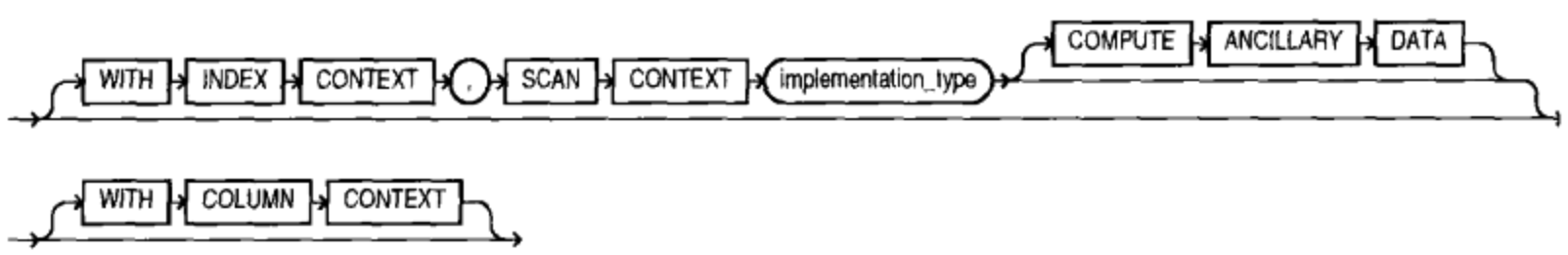
add_binding_clause::=



implementation_clause::=

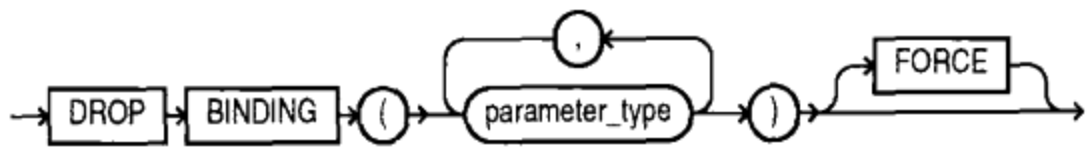


context_clause::=



using_function_clause::=

drop_binding_clause::=



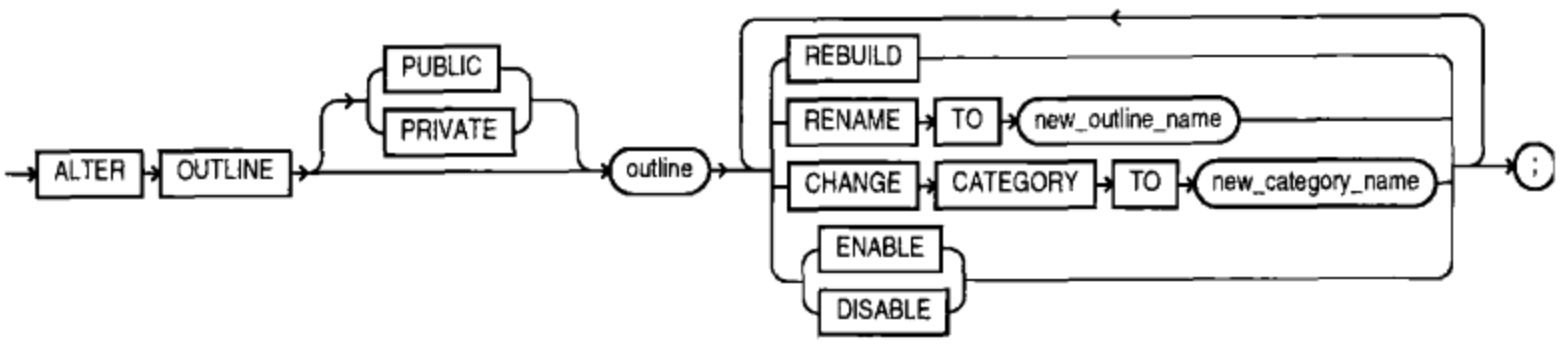
描述：可使用 ALTER OPERATOR 添加绑定，删除绑定或编译一个已存在的运算符。

ALTER OUTLINE

参阅：CREATE OUTLINE、DROP OUTLINE 和第 46 章。

格式：

alter_outline::=



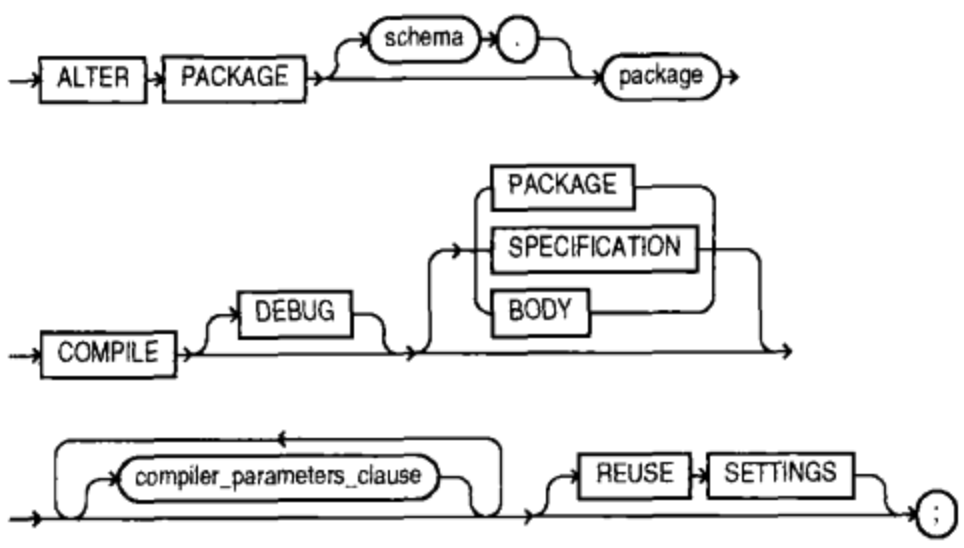
描述：可使用 ALTER OUTLINE 重命名一个存储概要，或者把它分配到另一个类别。存储概要维护以前执行的各查询的提示集合。

ALTER PACKAGE

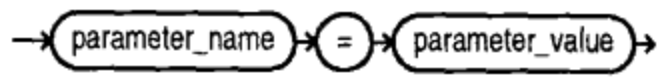
参阅：CREATE PACKAGE、DROP PACKAGE 和第 35 章。

格式：

alter_package::=



compiler_parameters_clause::=



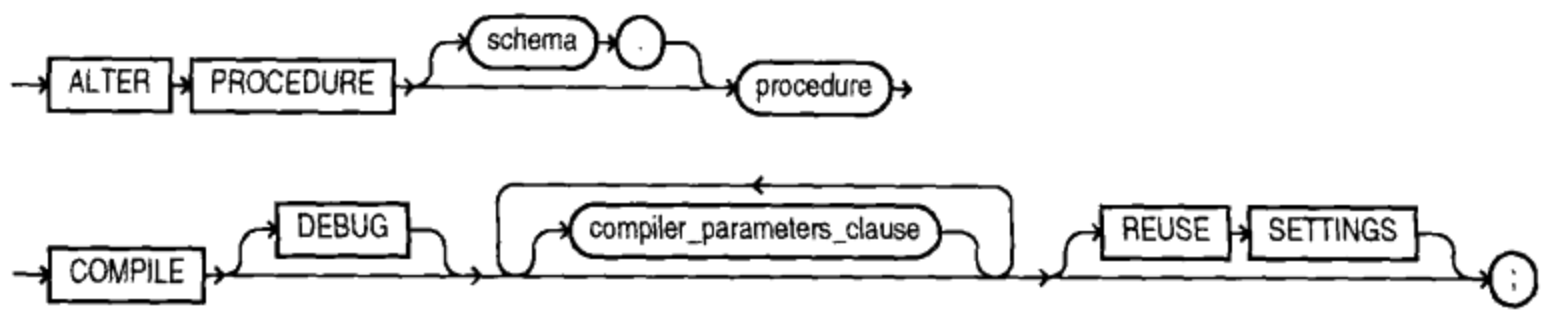
描述：ALTER PACKAGE 重新编译程序包规范和(或)程序包体。如果用 PACKAGE 重新编译包规范，就要同时重新编译包规范和程序包体。重新编译包规范将会导致引用过程被重新编译。可以重新编译 BODY 而不影响程序包规范。要更改程序包，必须拥有该程序包，或者拥有 ALTER ANY PROCEDURE 系统权限。

ALTER PROCEDURE

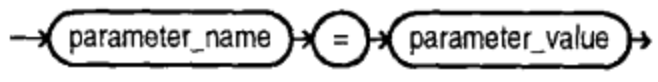
参阅：CREATE PROCEDURE、DROP PROCEDURE 和第 35 章。

格式：

alter_procedure::=



compiler_parameters_clause::=



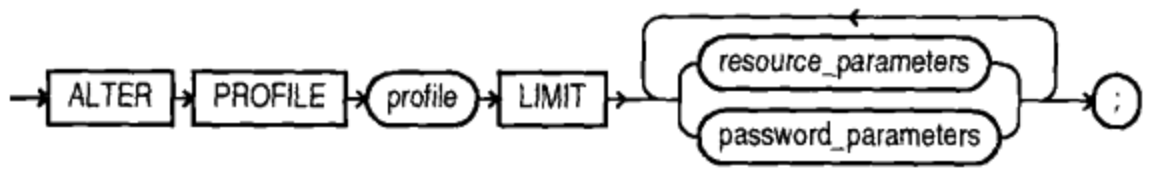
描述：ALTER PROCEDURE 重新编译 PL/SQL 过程。通过预先显式地编译一个过程，可以避免运行时的系统开销和错误消息。要更改一个过程，必须拥有该过程或者拥有 ALTER ANY PROCEDURE 系统权限。

ALTER PROFILE

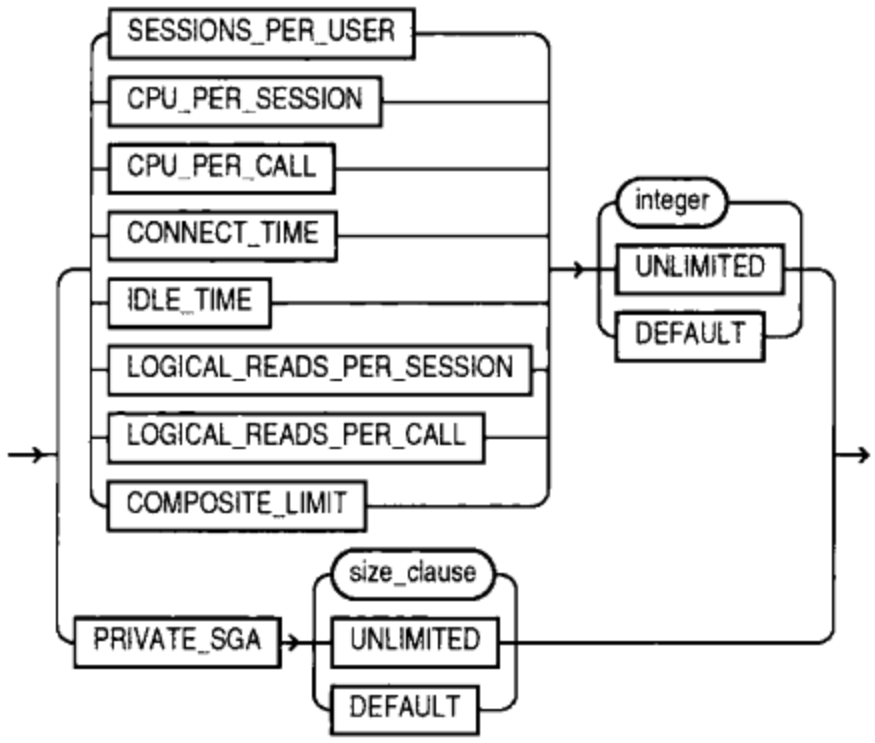
参阅：CREATE PROFILE、DROP PROFILE 和第 19 章。

格式：

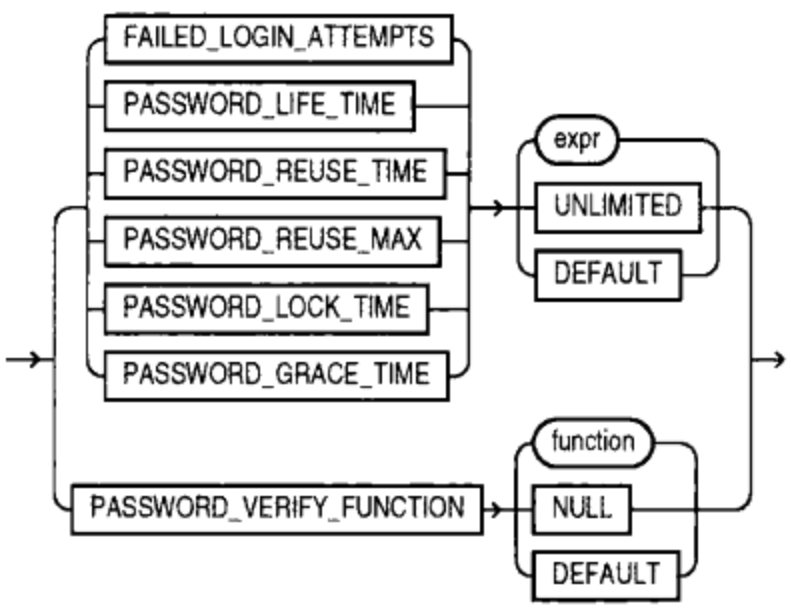
alter_profile::=



resource_parameters::=



password_parameters::=



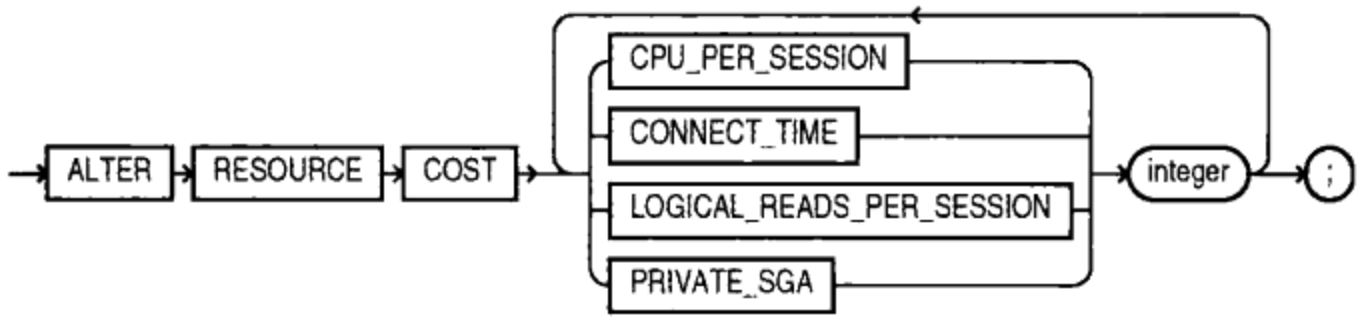
描述： ALTER PROFILE 可以修改特定的配置文件设置。详细内容请参阅 CREATE PROFILE。

ALTER RESOURCE COST

参阅： CREATE PROFILE。

格式：

alter_resource_cost::=



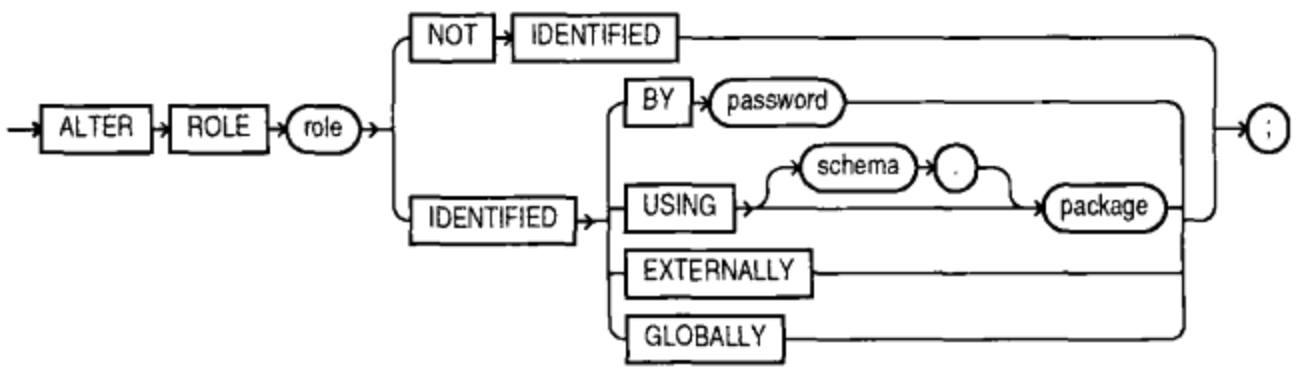
描述：资源成本公式计算在 Oracle 会话中使用的资源的总成本。ALTER RESOURCE COST 允许给几个资源分配权重。CPU_PER_SESSION 是在会话中使用的 CPU 量，以百分之一秒为单位。CONNECT_TIME 是会话时间，以分钟为单位。LOGICAL_READS_PER_SESSION 是一次会话期间从内存和磁盘读取的数据块数。PRIVATE_SGA 是专用系统全局区(SGA)中的字节数，它只与使用共享服务器体系结构有关。通过分配权重，可以修改用来计算总资源成本的公式。

ALTER ROLE

参阅：CREATE ROLE、DROP ROLE 和第 19 章。

格式：

alte_role::=



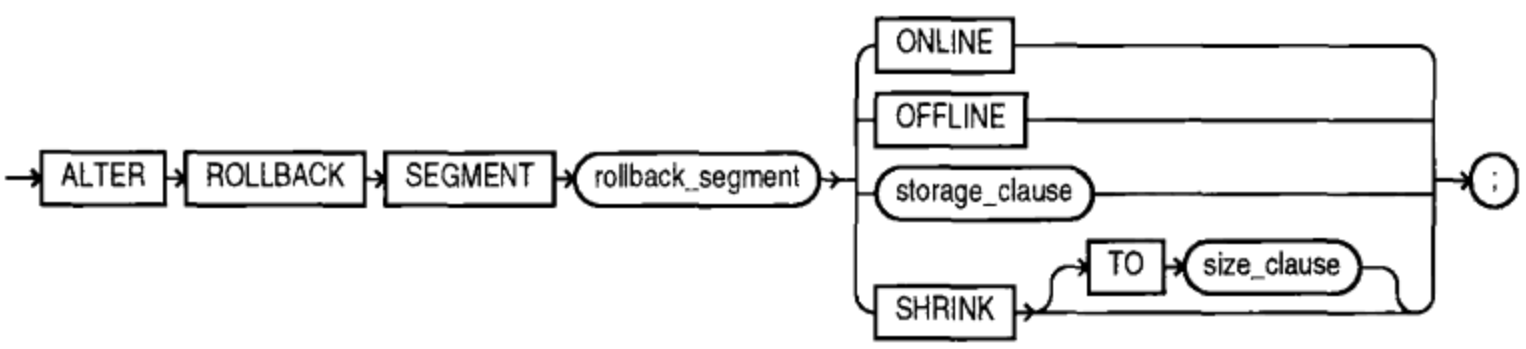
描述：ALTER ROLE 可以用来修改角色。

ALTER ROLLBACK SEGMENT

参阅：CREATE DATABASE、CREATE ROLLBACK SEGMENGT、CREATE TABLESPACE、DROP ROLLBACK SEGMENT、STORAGE 和第 51 章。

格式：

alter_rollback_segment::=



描述：rollback_segment 是分配给回滚段的名称。STORAGE 包含在 STORAGE 下描述的子句。INITIAL 选项和 MINEXTENTS 选项不适用。在数据库处于打开状态时，可以使回滚段联机(ONLINE)或脱机(OFFLINE)。可以把回滚段缩小(SHRINK)为指定大小(默认为其 OPTIMAL 大小)。

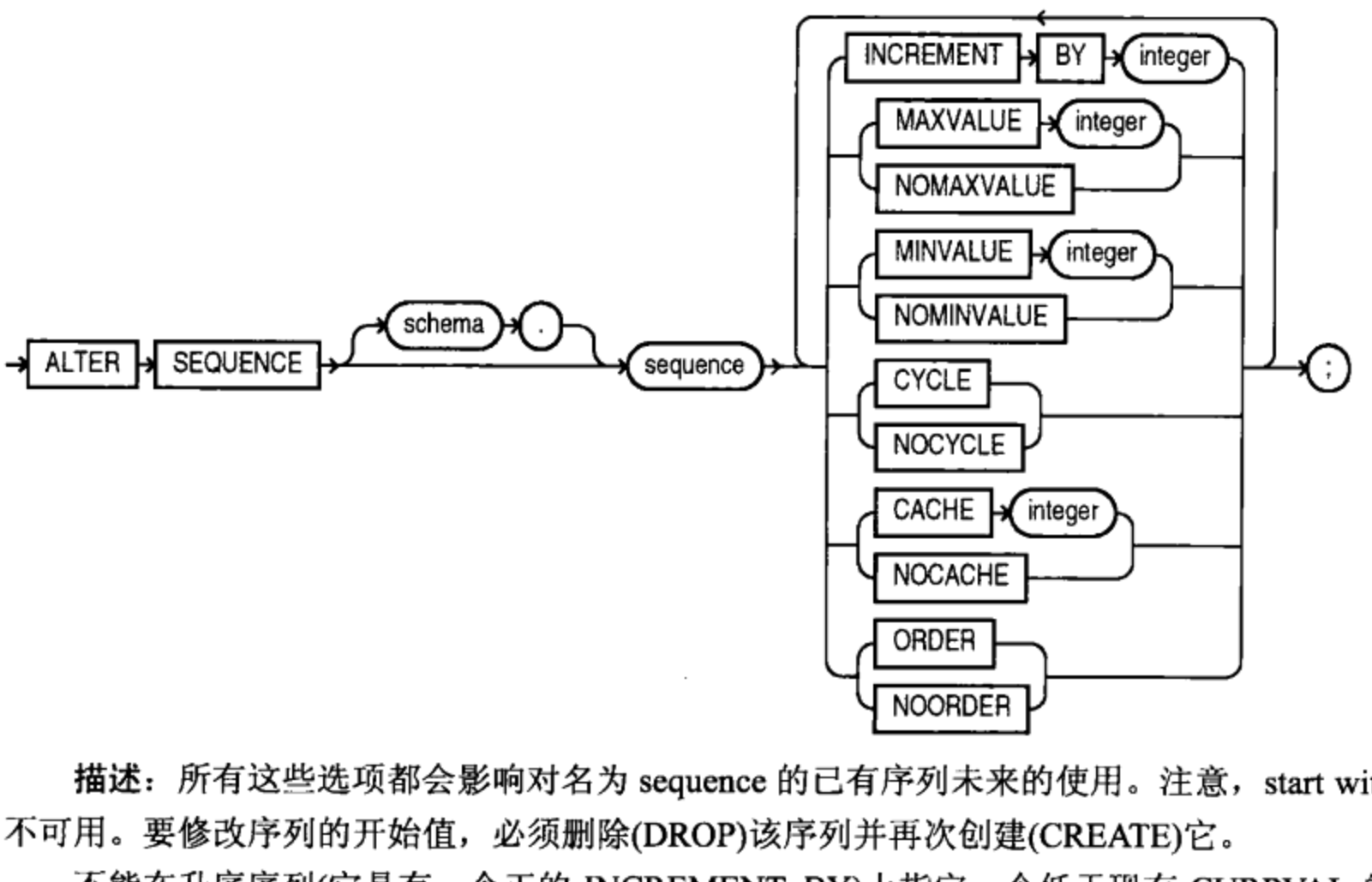
应当使用系统管理撤消，以便数据库能够在需要的时候自动创建和删除回滚段。

ALTER SEQUENCE

参阅：AUDIT、CREATE SEQUENCE、DROP SEQUENCE、GRANT、REVOKE、PSEUDO_COLUMNS 下的 NEXTVAL 与 CURRVAL 和第 17 章。

格式：

alter_sequence::=



描述：所有这些选项都会影响对名为 sequence 的已有序列未来的使用。注意，start with 不可用。要修改序列的开始值，必须删除(DROP)该序列并再次创建(CREATE)它。

不能在升序序列(它具有一个正的 INCREMENT BY)上指定一个低于现有 CURRVAL 的新 MAXVALUE。在降序序列(它具有一个负的 INCREMENT BY)上对 MINVALUE 也有类似规则。

要更改一个序列，必须在该序列上拥有 ALTER 权限，或者拥有 ALTER ANY SEQUENCE 系统权限。

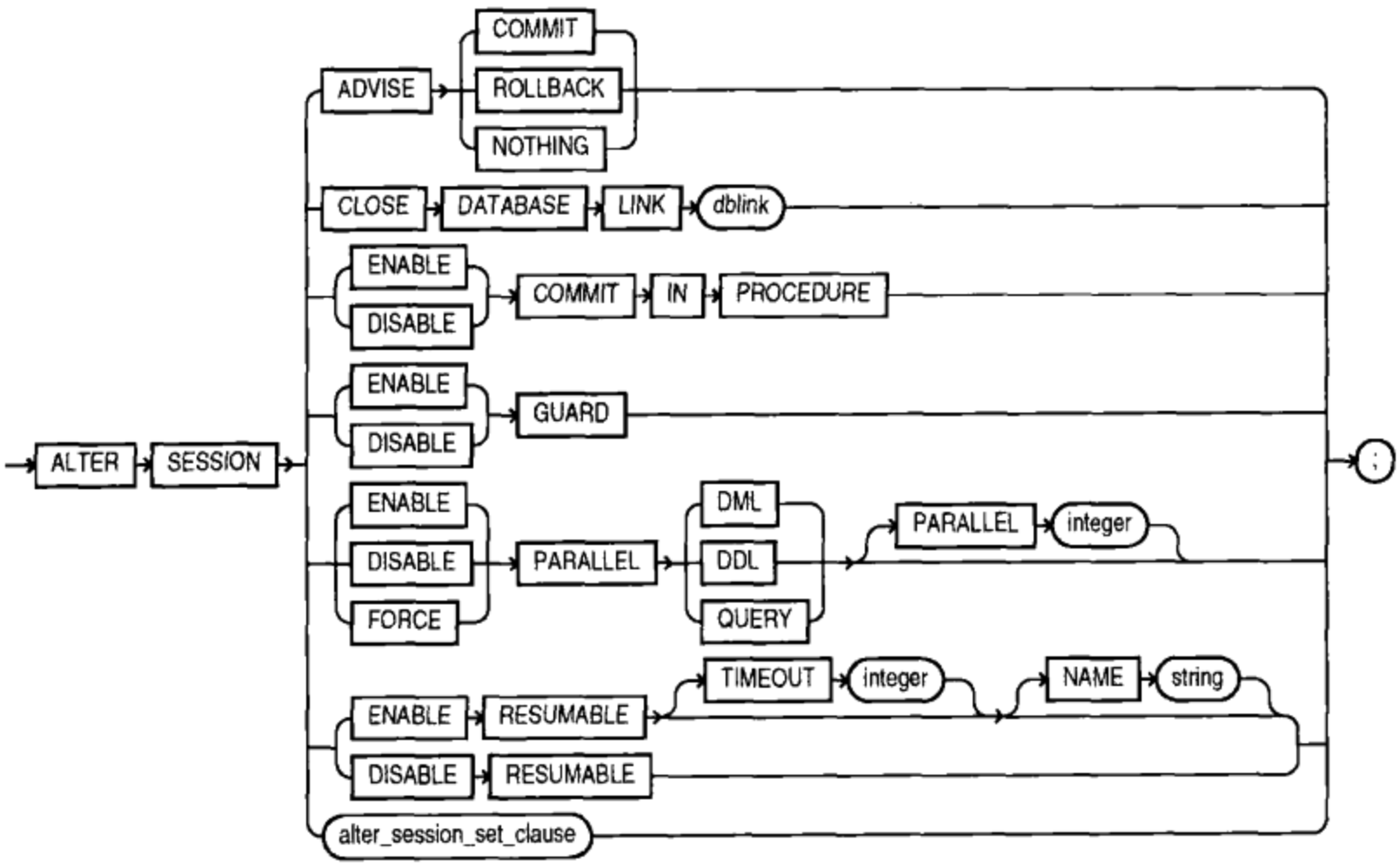
ALTER SEQUENCE 的其他所有功能除了应用于已有序列外，其作用与在 CREATE SEQUENCE 中的相同。详细内容请参阅 CTEATE SEQUENCE。

ALTER SESSION

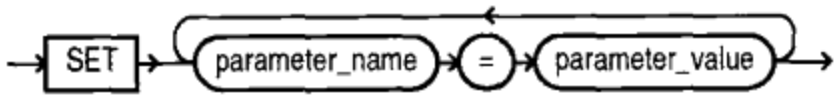
参阅：ALTER SYSTEM。

格式：

alter_session::=



alter_session_set_clause::=



描述：ALTER SESSION 命令用来更改当前用户会话的实际设置。

可以指定过程是否可以提交(COMMIT)，DML 操作的并行程度，以及是否启用可恢复操作。如果允许为会话使用并行 DML 或 DDL，则仍然必须为并行执行的操作指定 PARALLEL 提示。

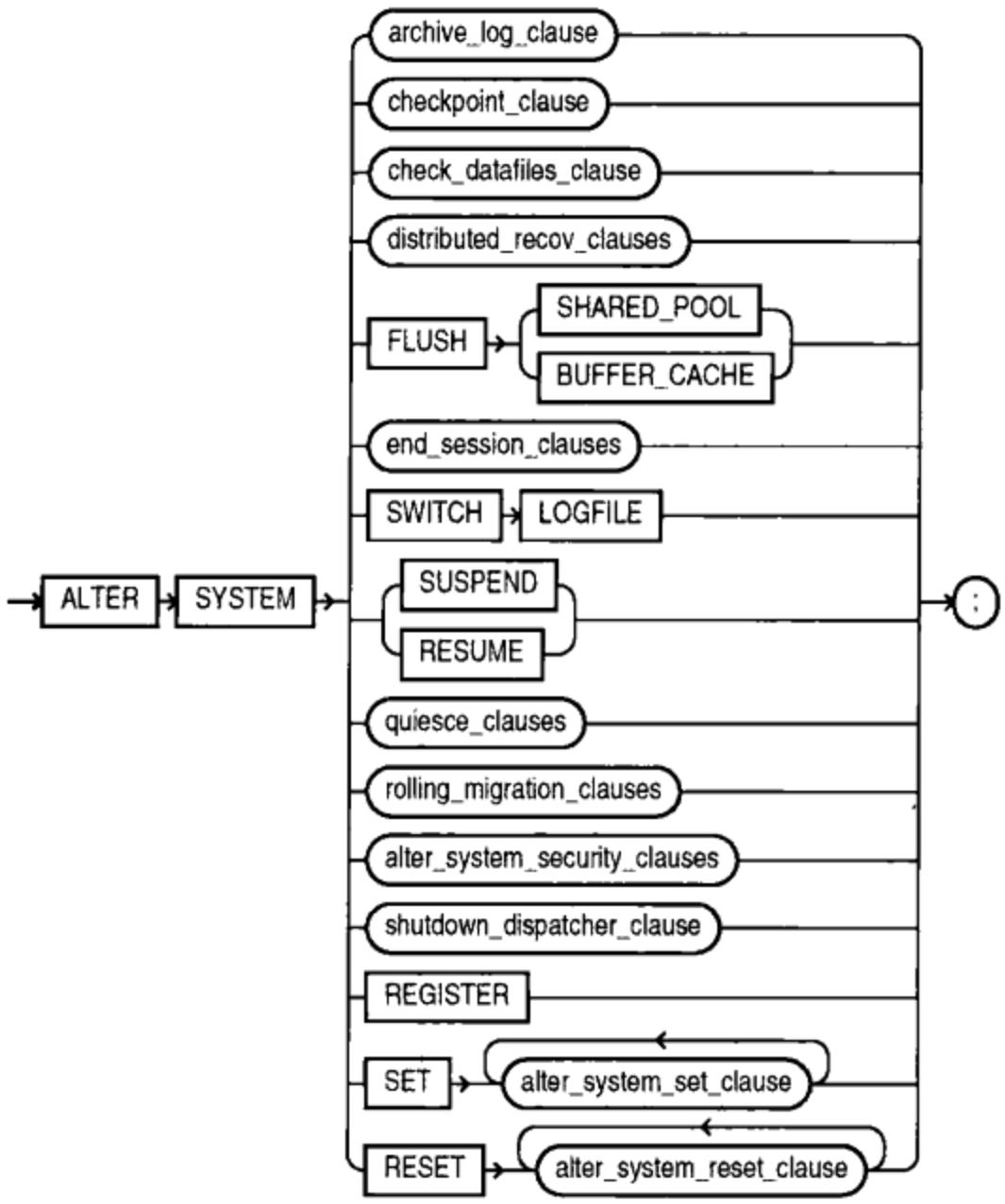
要启用或禁用可恢复空间分配，必须拥有 RESUMABLE 系统权限。

ALTER SYSTEM

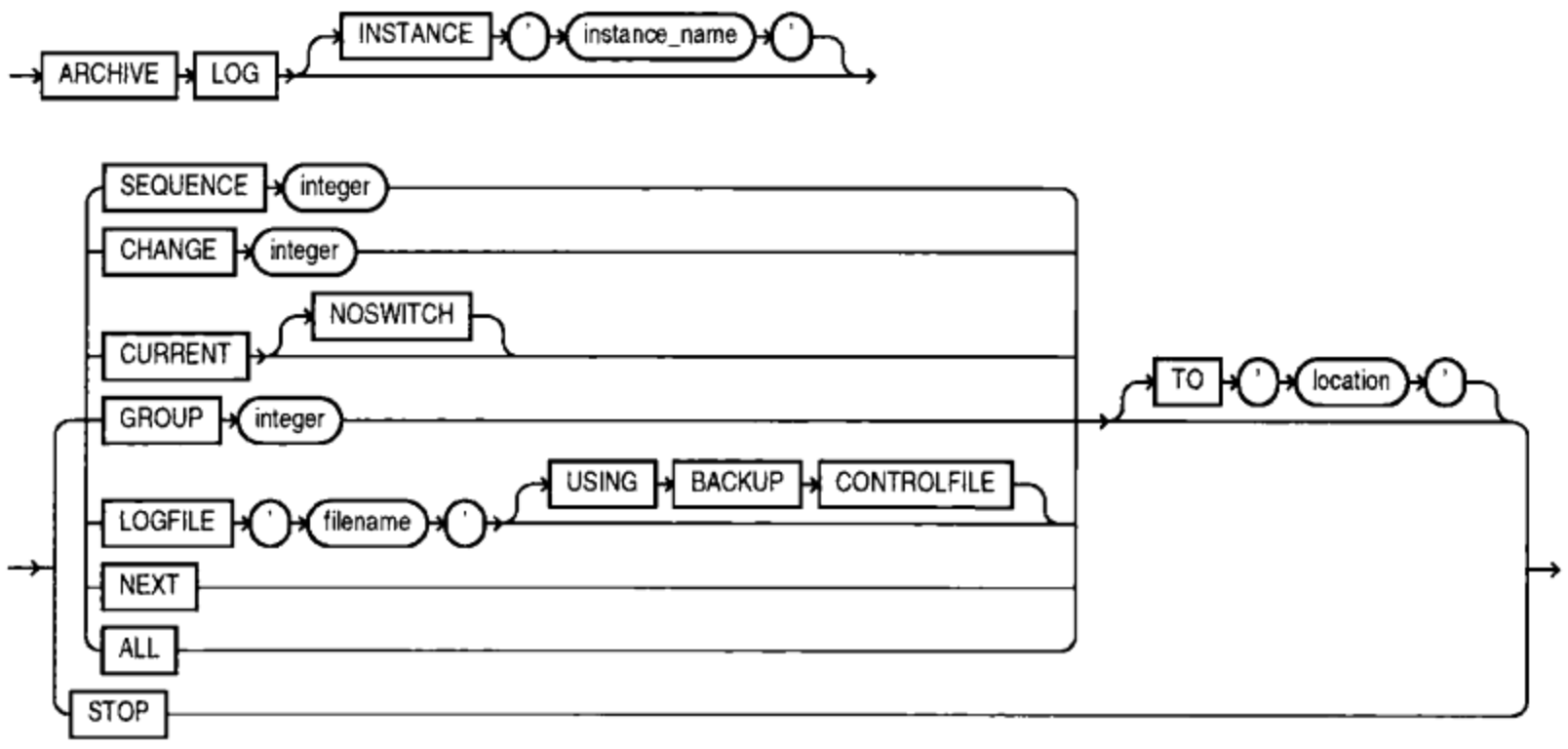
参阅：ALTER SESSION 和 CREATE PROFILE。

格式：

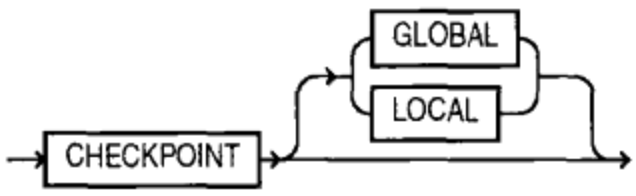
alter_system::=



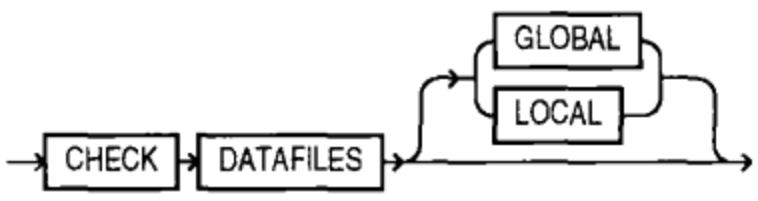
archive_log_clause::=



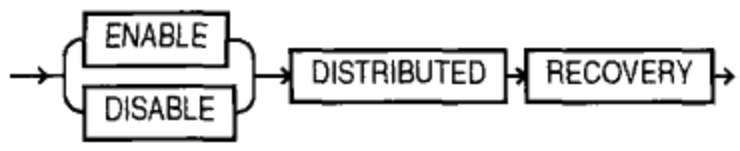
checkpoint_clause::=



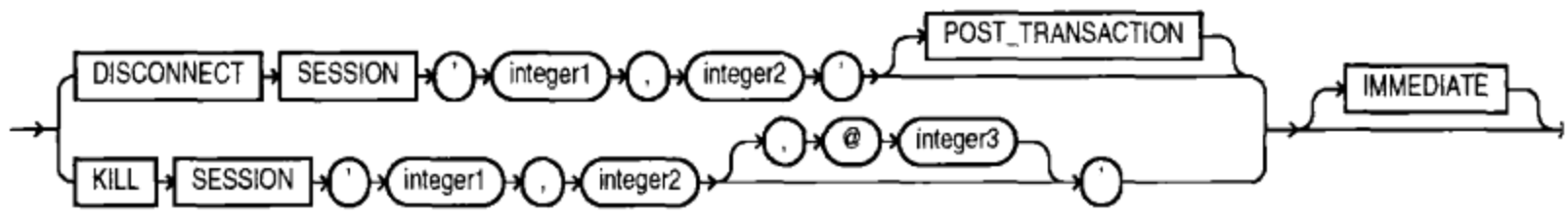
check_datafiles_clause::=



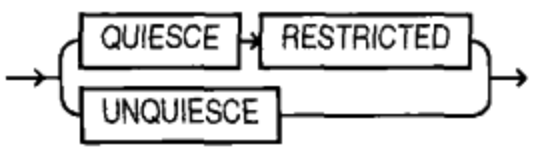
distributed_recov_clauses::=



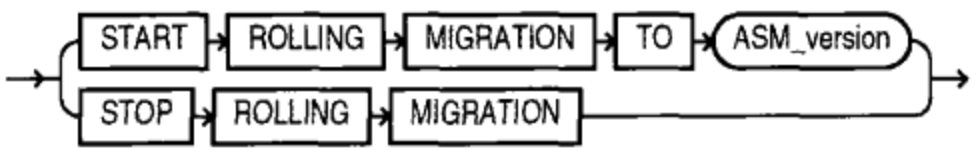
end_session_clauses::=



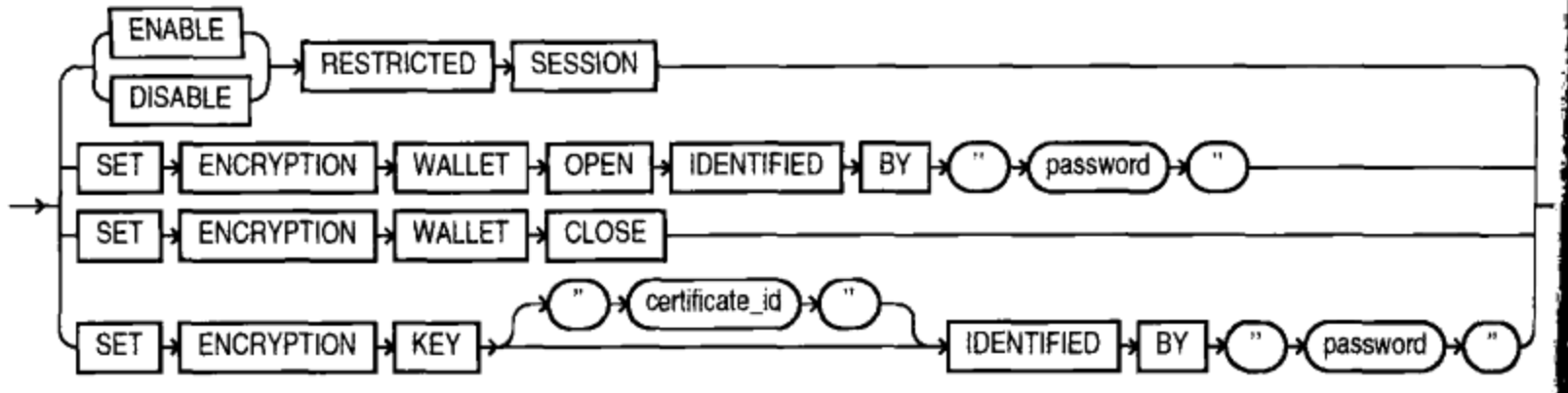
quiesce_clauses::=



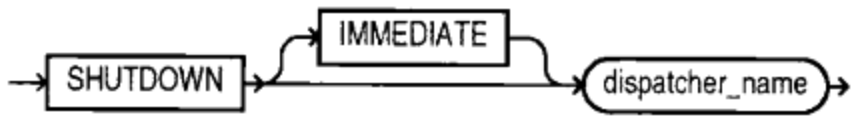
rolling_migration_clauses::=



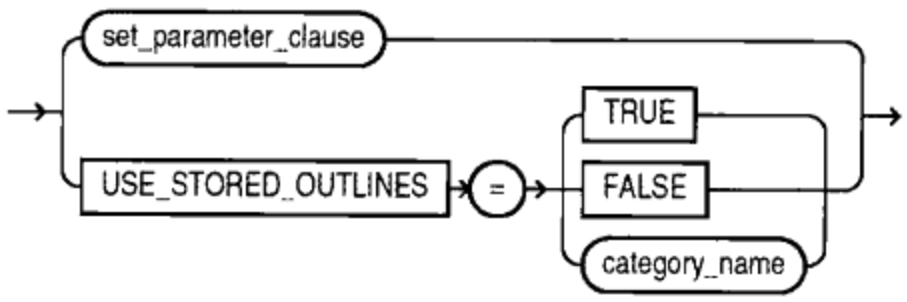
alter_system_security_clauses::=



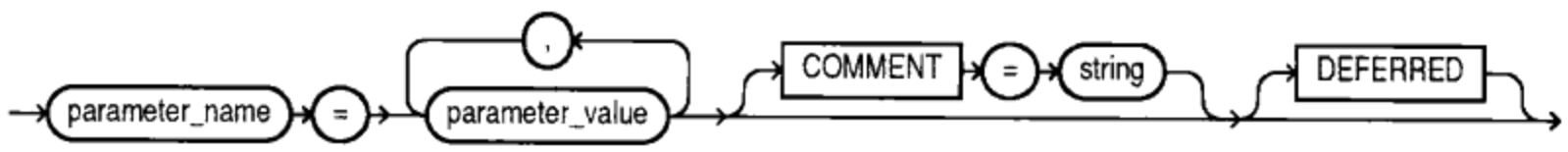
shutdown_dispatcher_clause::=

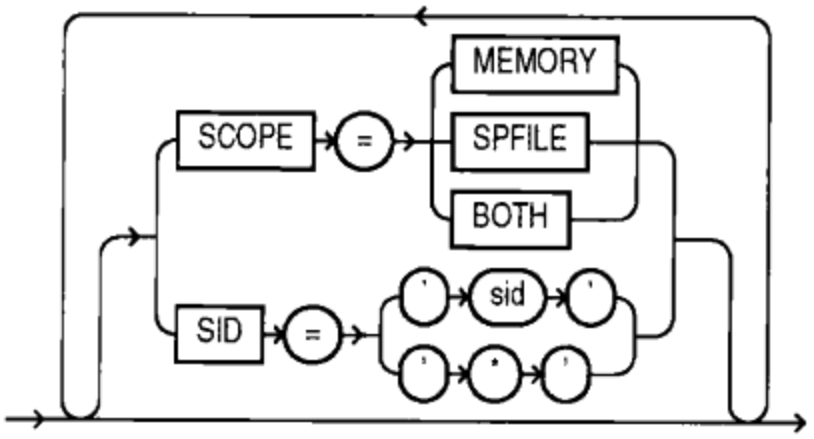


alter_system_set_clause::=

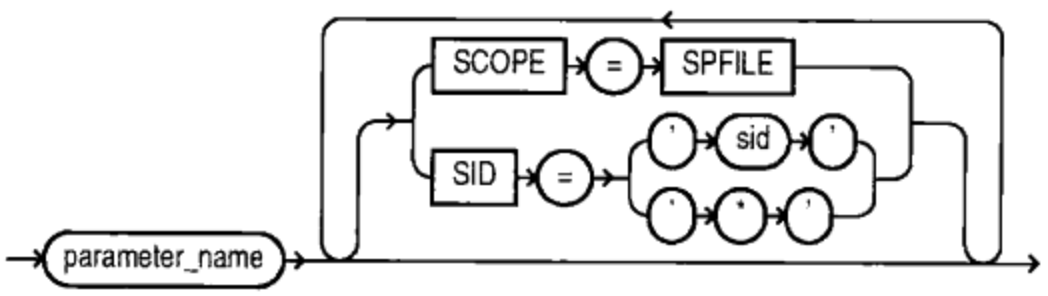


set_parameter_clause::=





alter_system_reset_clause::=



描述：ALTER SYSTEM 允许用户以多种方式配置 Oracle 实例。通过 SET 选项，可以在数据库运行时，修改许多初始化参数值。如果正在使用系统参数文件，则修改将被保存，并将在下次启动数据库时生效。

还可以使用 ALTER SYSTEM 命令进行 SWITCH LOGFILE(切换日志文件)操作，强制 Oracle 更改重做日志文件组。

CHECKPOINT 为 Oracle 的所有实例(GLOBAL)或用户实例(LOCAL)执行检查点。CHECK DATAFILES 操作验证 Oracle 的每个实例(GLOBAL)或用户实例(LOCAL)是否可以访问所有联机数据文件。

ENABLE 或 DISABLE RESTRICTED SESSION 打开或者关闭一个受限的会话，只有拥有该系统权限的用户可以访问该会话。

ENABLE 或 DISABLE DISTRIBUTED RECOVERY 分别打开或者关闭这个选项。

FLUSH SHARED_POOL 清除 SGA 共享池中的所有数据。

FLUSH BUFFER_CACHE(从 Oracle Database 10g 开始可用)清除 SGA 的缓冲区缓存。

SUSPEND 挂起数据库的活动，RESUME 重新启用该数据库。

QUIESCE RESTRICTED 把数据库置为一个一致的状态，在此状态中，只有以 SYSDAB 连接的 SYS 和 SYSTEM 可以执行操作。UNQUIESCE 使数据库返回其正常的模式。

ARCHIVE LOG 人工归档重做日志文件，或启用自动归档。

KILL SESSION 使用 SID 列和 V$SESSION 表的标识会话的 SERIAL#列结束一个会话。DISCONNECT SESSION 与 KILL SESSION 不同，它提供在会话终止前强制提交会话的未完成事务的选项。

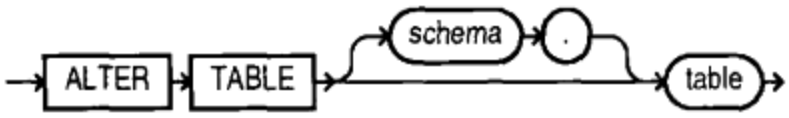
在 Oracle Database 11g 中，可以使用 ALTER SYSTEM 终止 RAC 环境中另一个实例的会话。另外，可以使用新的 rolling_migration_clauses 准备 ASM 群集，以便在所有节点已经迁移到相同的软件版本之后将 ASM 群集迁移并返回到正常的操作。

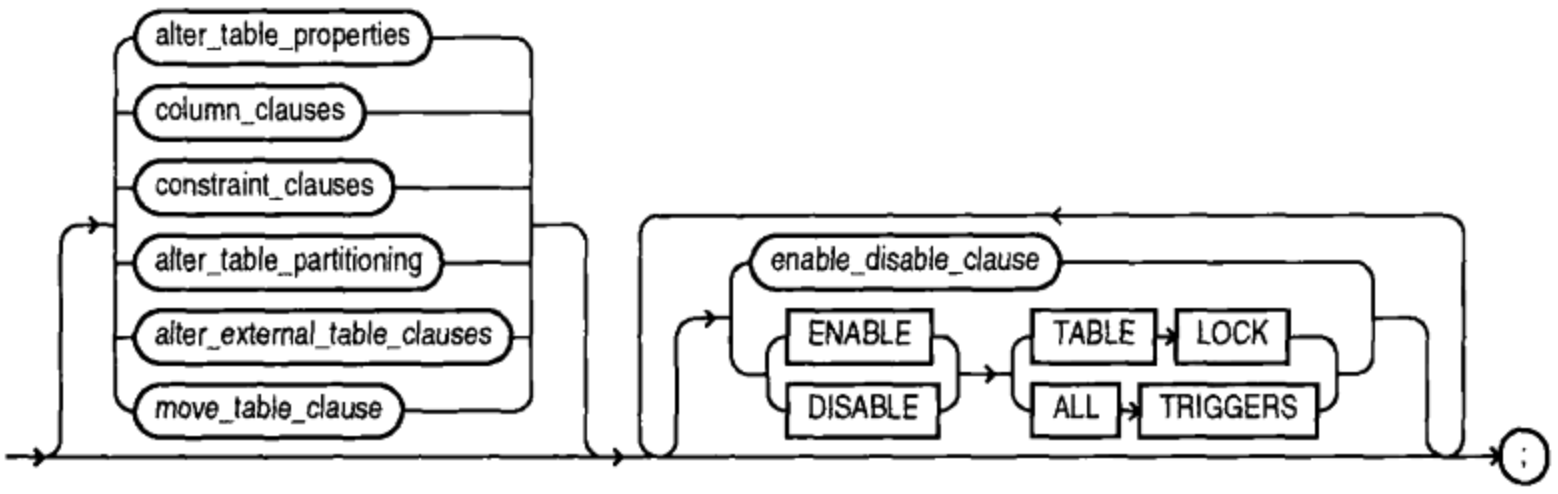
ALTER TABLE

参阅： ALTER TRIGGER、CREATE TABLE、DROP TABLE、ENABLE 和第 17 章。

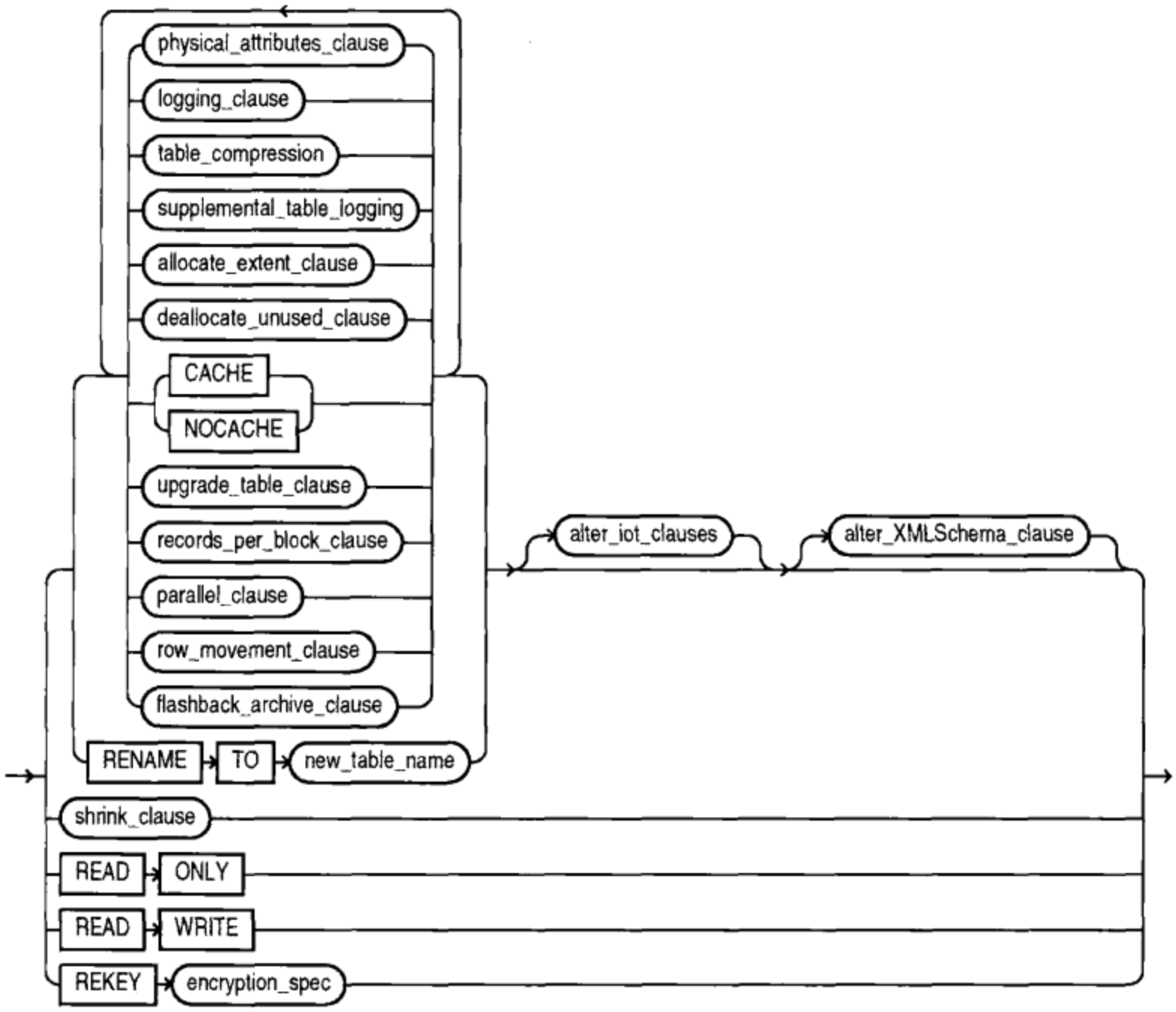
格式：

alter_table::=

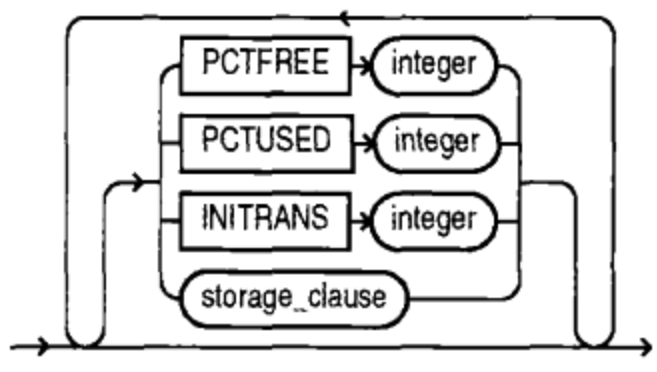




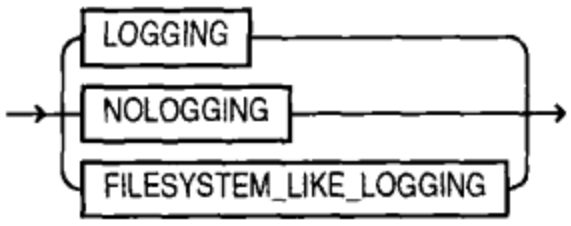
alter_table_properties::=



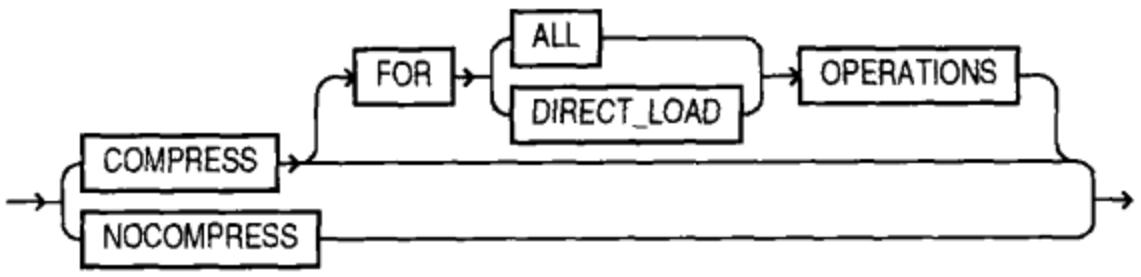
physical_attributes_clause::=



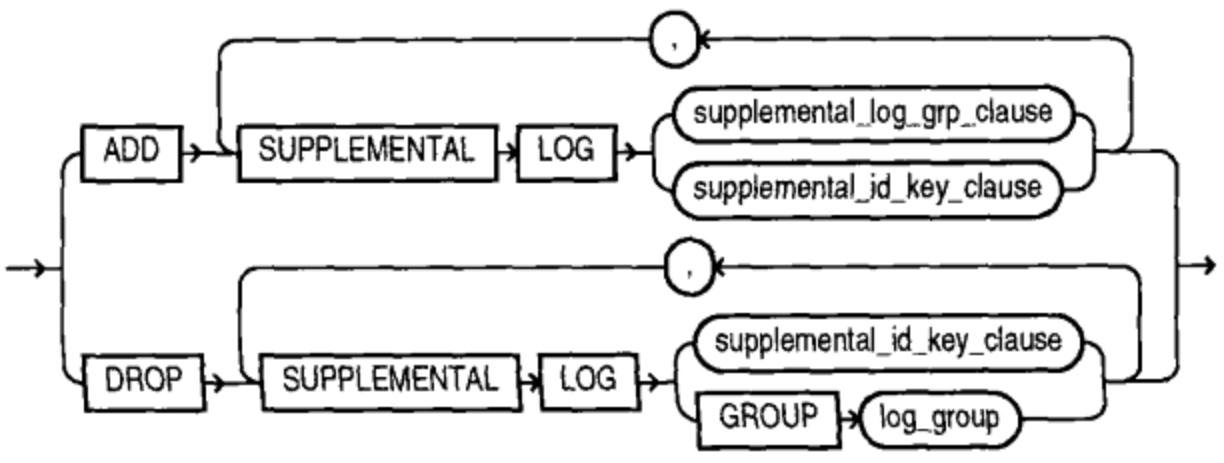
logging_clause::=



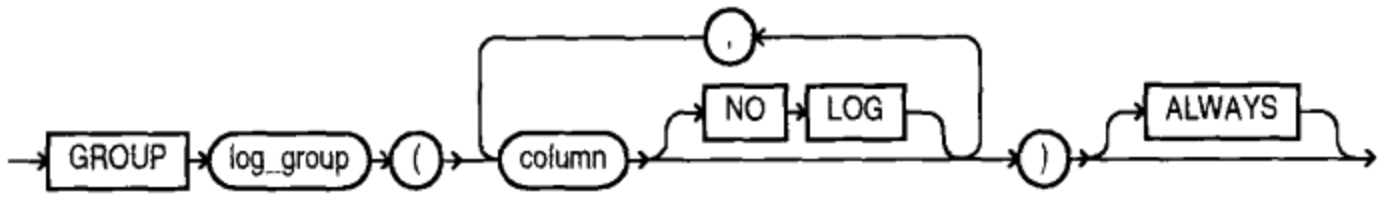
table_compression::=



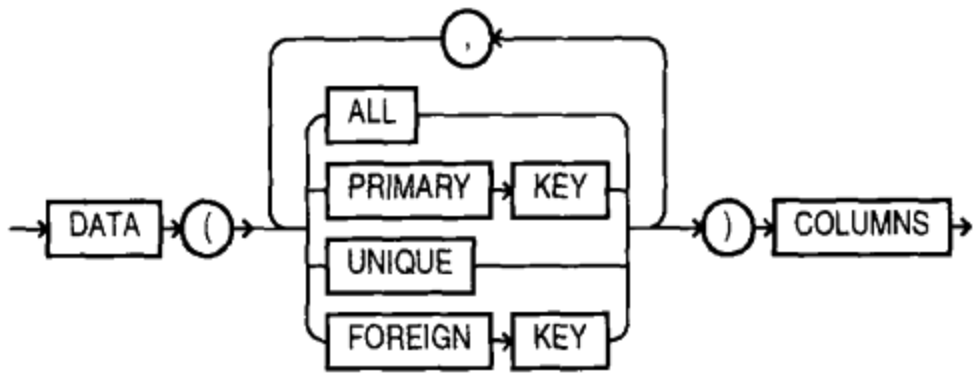
supplemental_table_logging::=



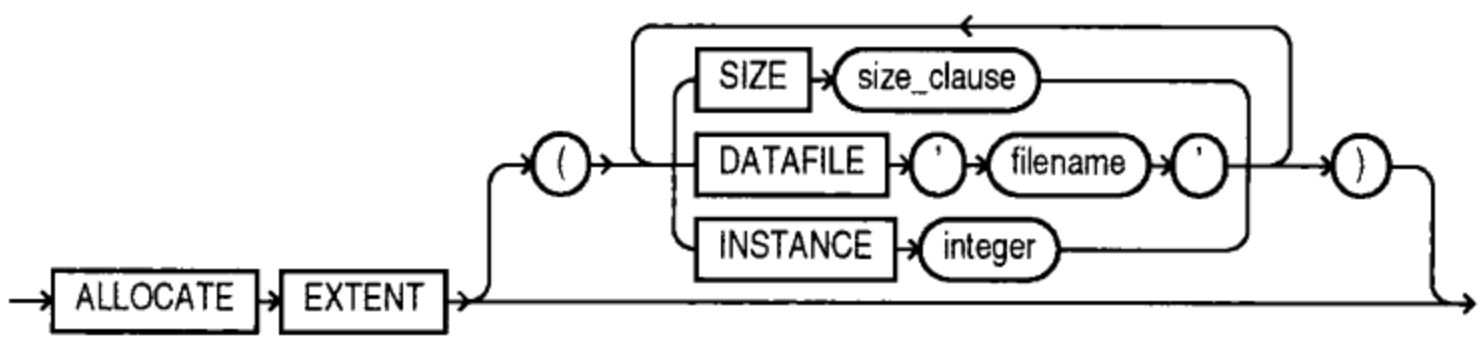
supplemental_log_grp_clause::=



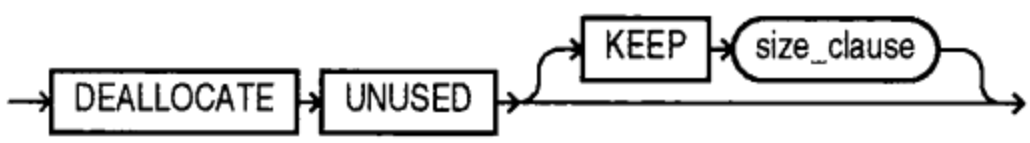
supplemental_id_key_clause::=



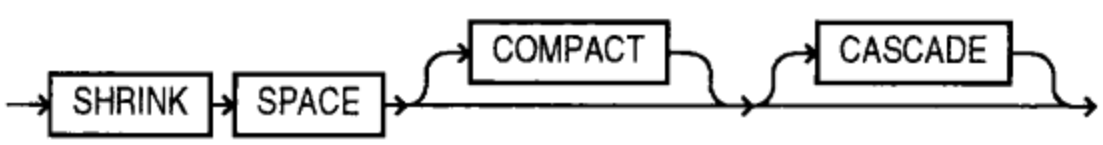
allocate_extent_clause::=



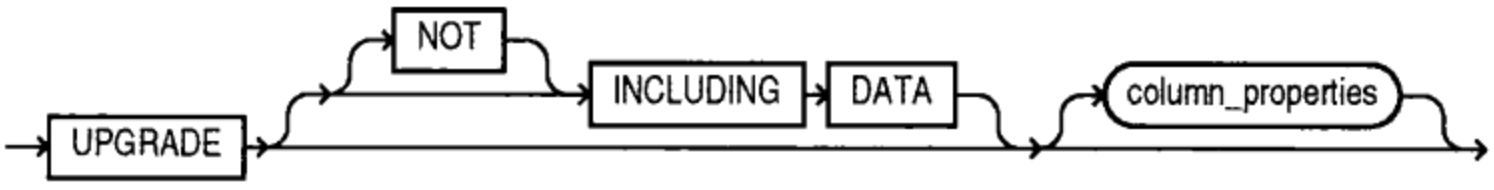
deallocate_unused_clause::=



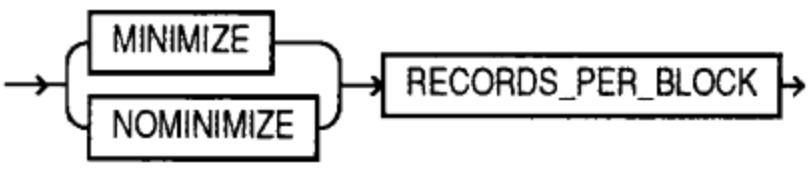
shrink_clause::=



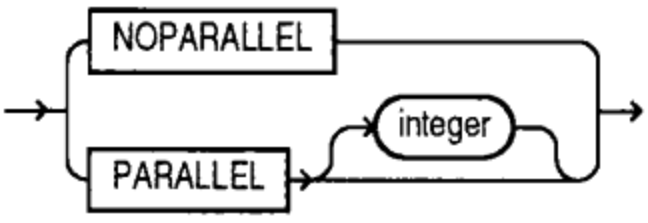
upgrade_table_clause::=



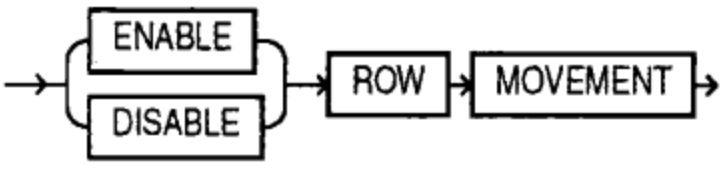
records_per_block_clause::=



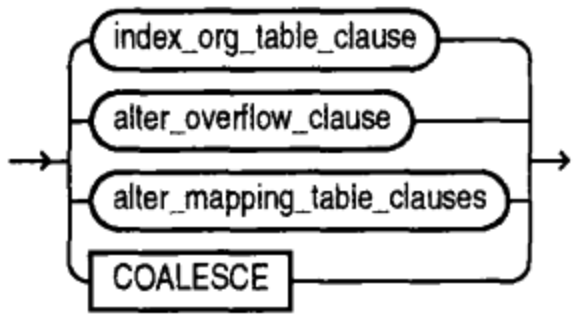
parallel_clause::=



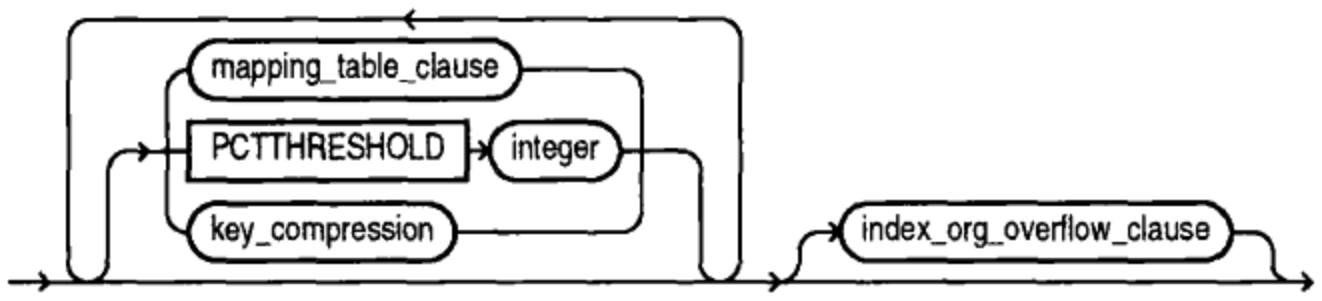
row_movement_clause::=



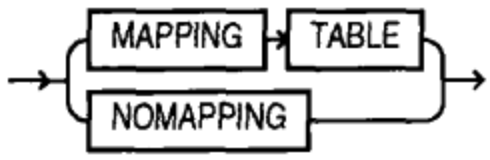
alter_iot_clauses::=



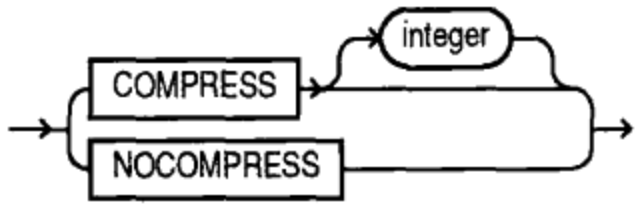
index_org_table_clauses::=



mapping_table_clauses::=

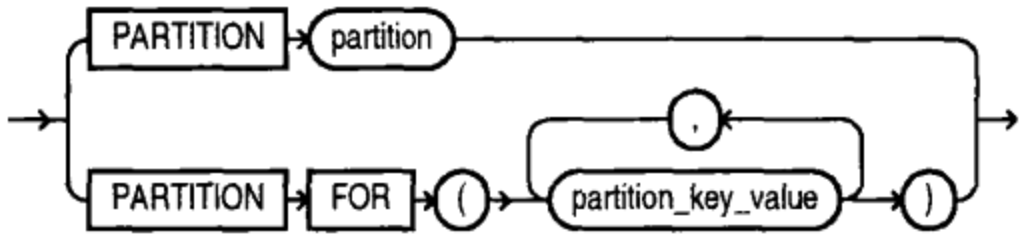


key_compression::=

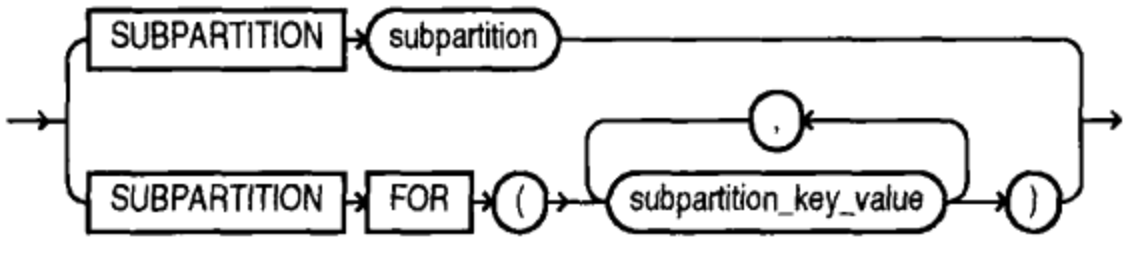


index_org_overflow_clause::=

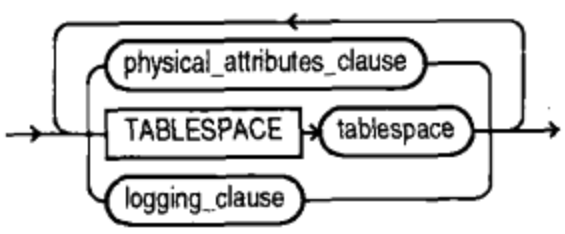
partition_extended_name::=



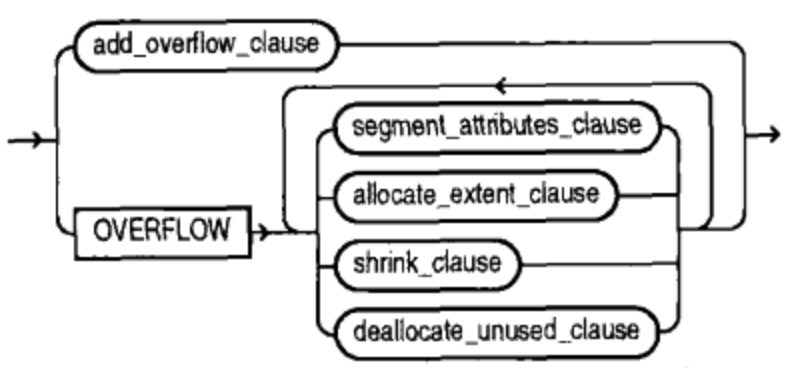
subpartition_extended_name::=



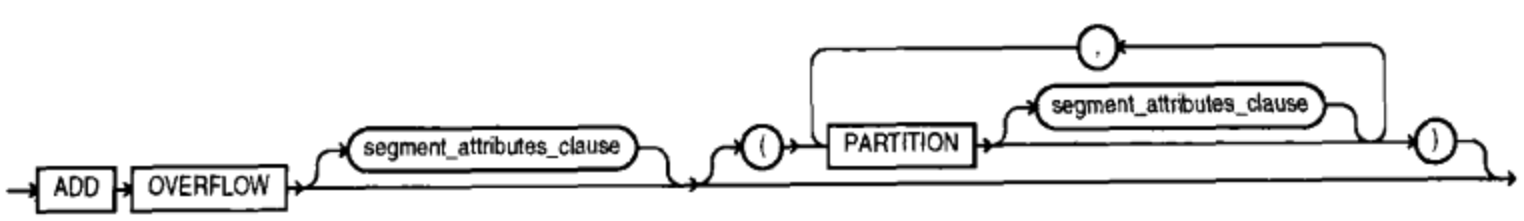
segment_attributes_clause::=



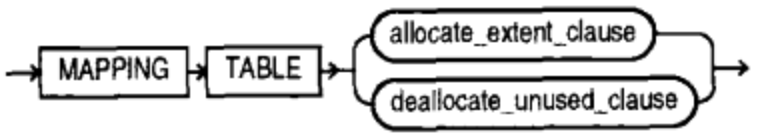
alter_overflow_clause::=



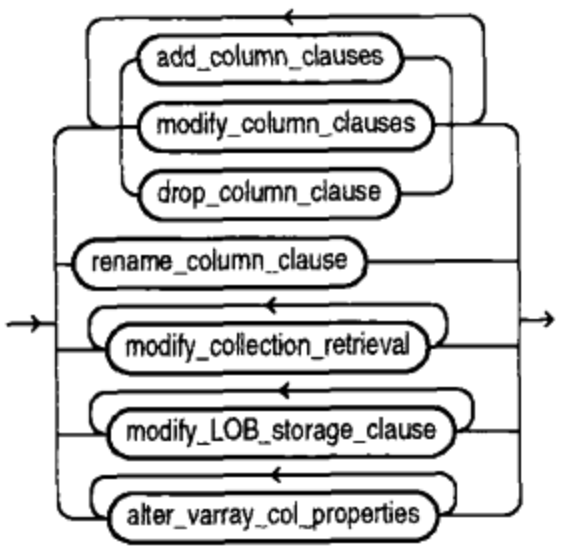
add_overflow_clause::=



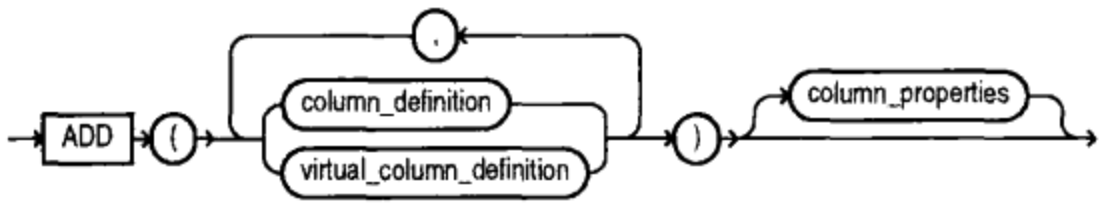
alter_mapping_table_clause::=



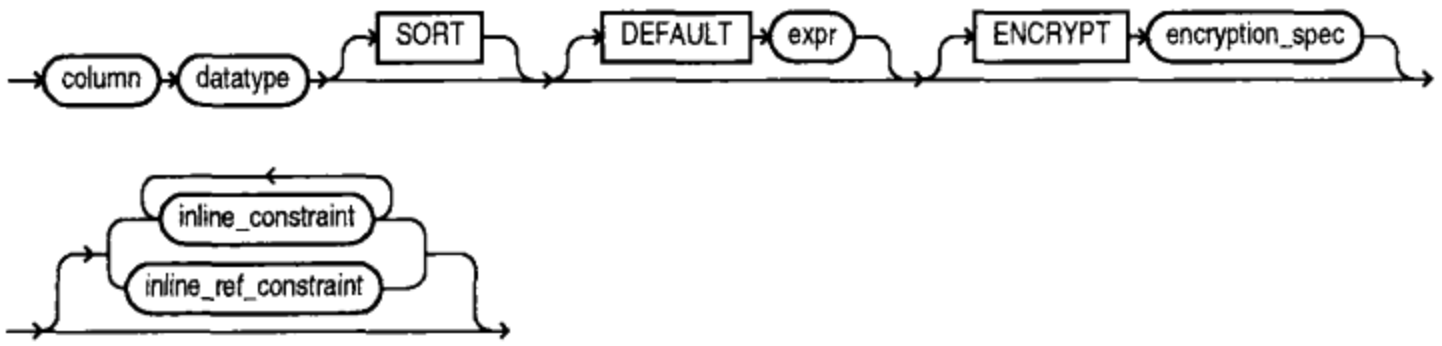
column_clause::=



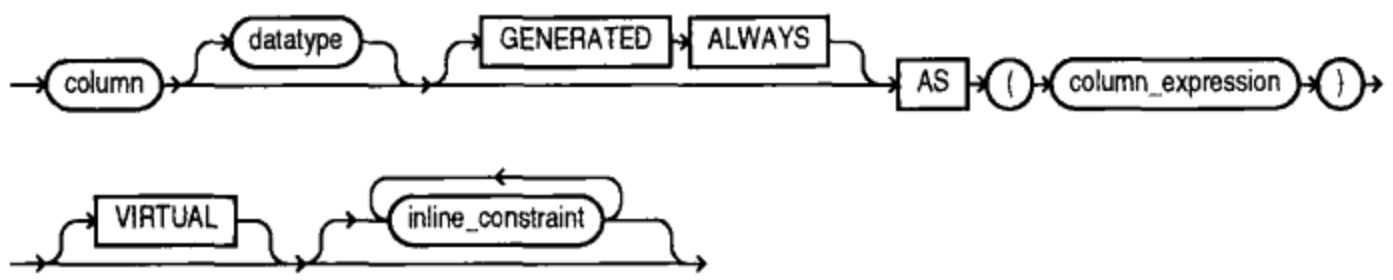
add_column_clause::=



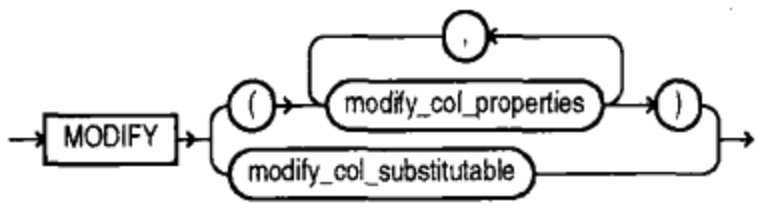
column_definition::=



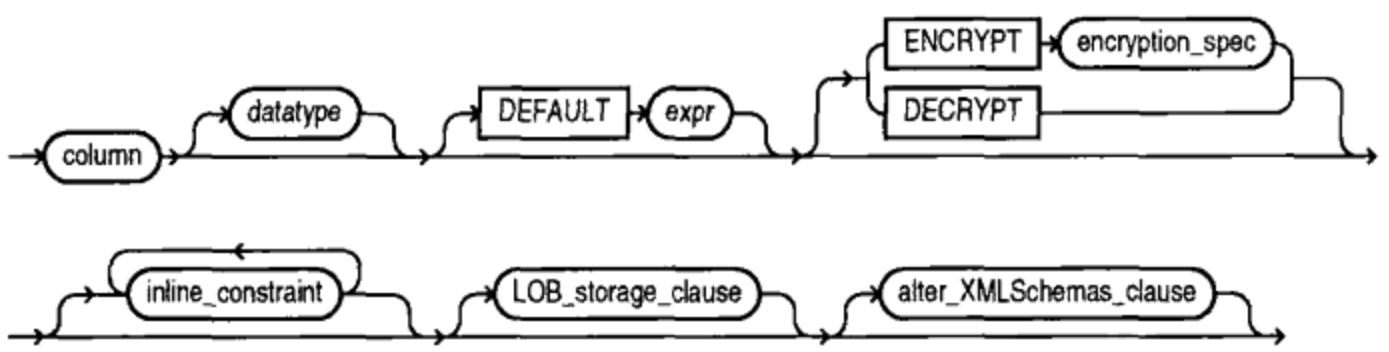
virtual_column_definition::=



modify_column_clauses::=



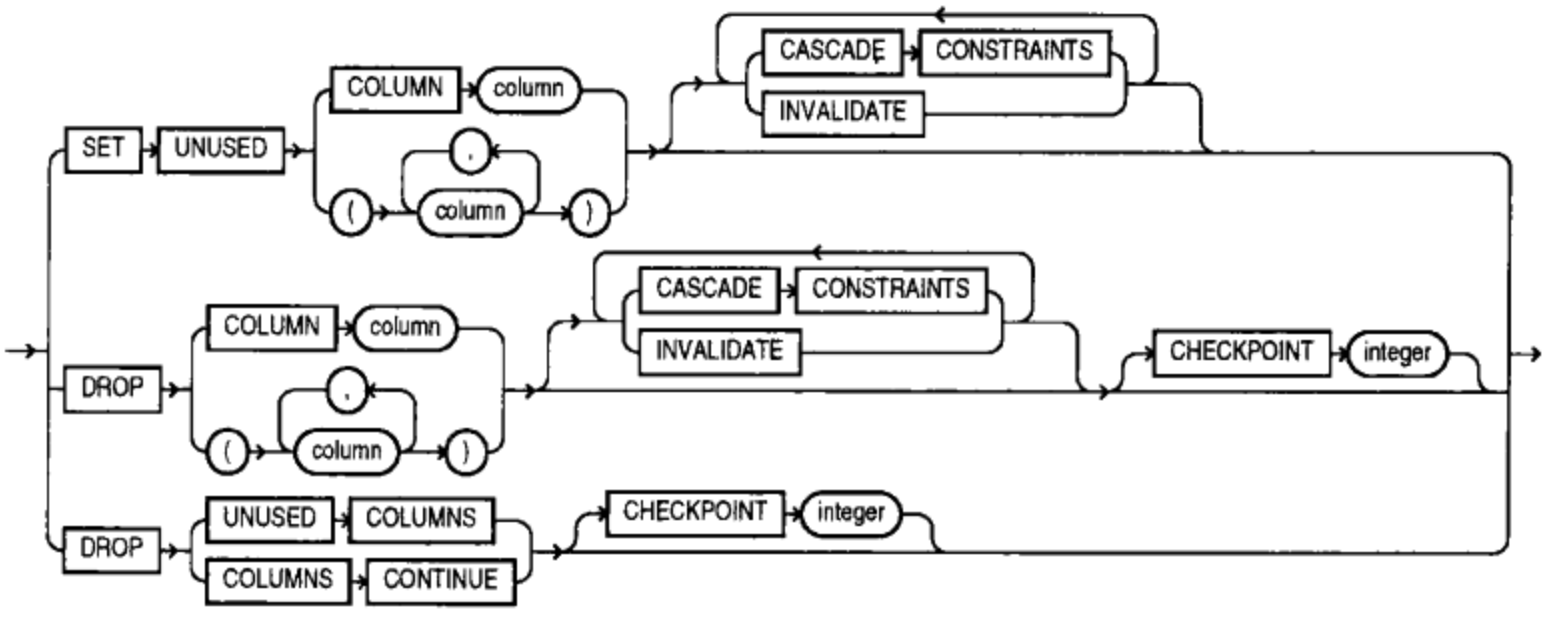
modify_col_properties::=



encryption_spec::=

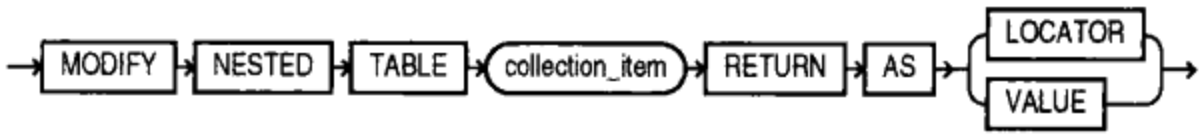
modify_col_substitutable::=

drop_column_clause::=

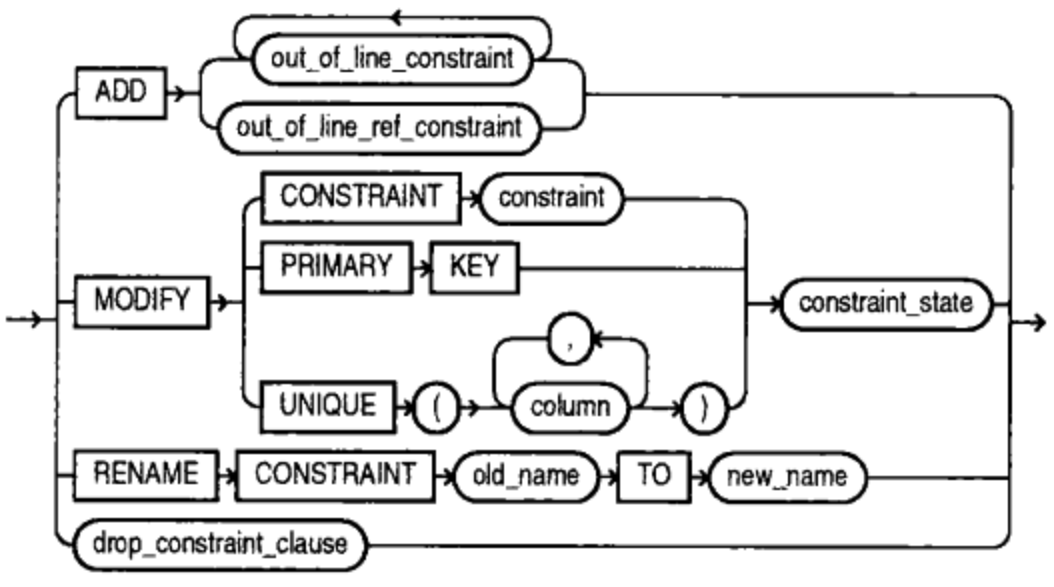


rename_column_clause::=

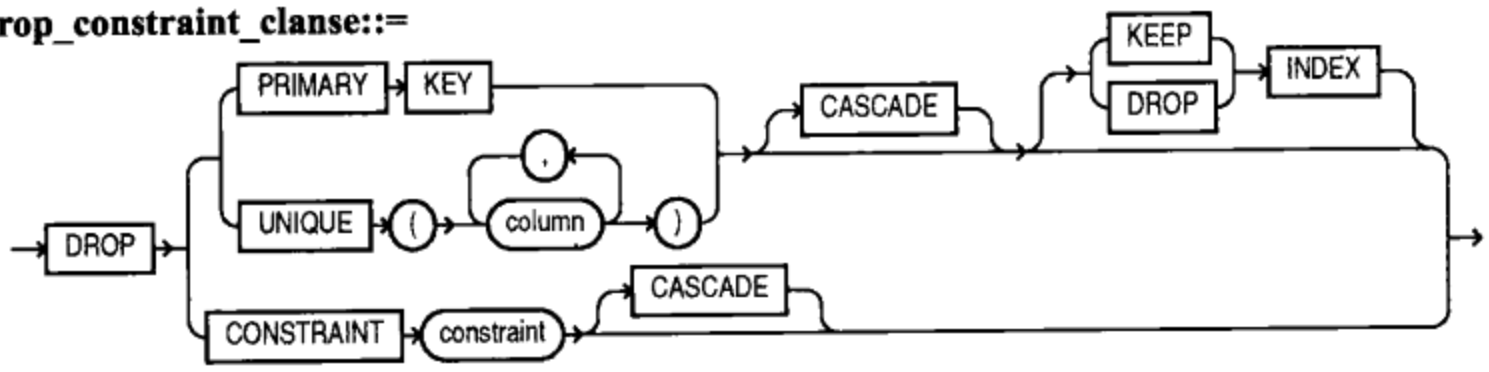
modify_collection_retrieval::=



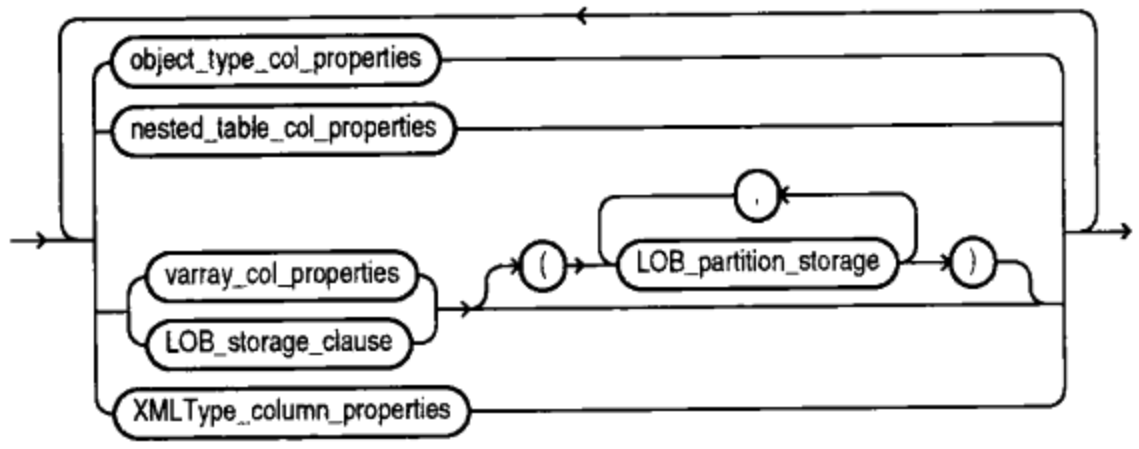
constraint_clauses::=



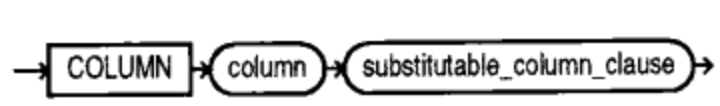
drop_constraint_clanse::=



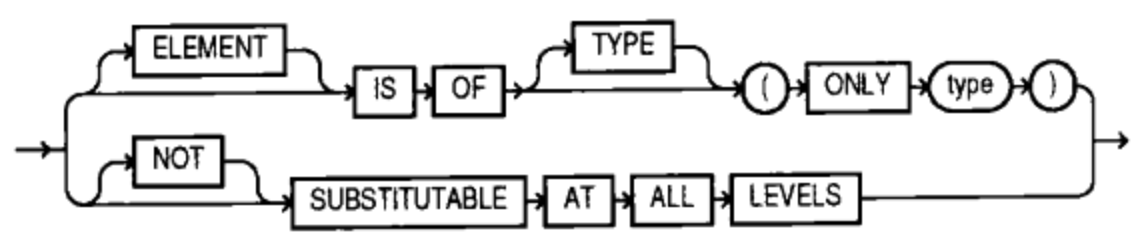
column_properties::=



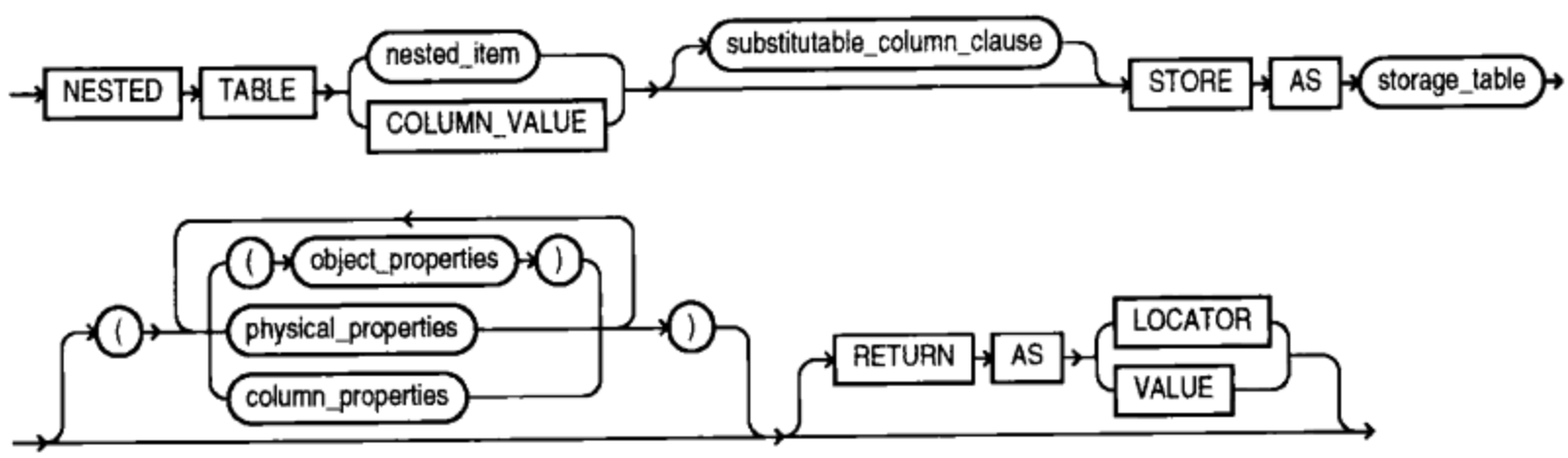
object_type_col_properties::=



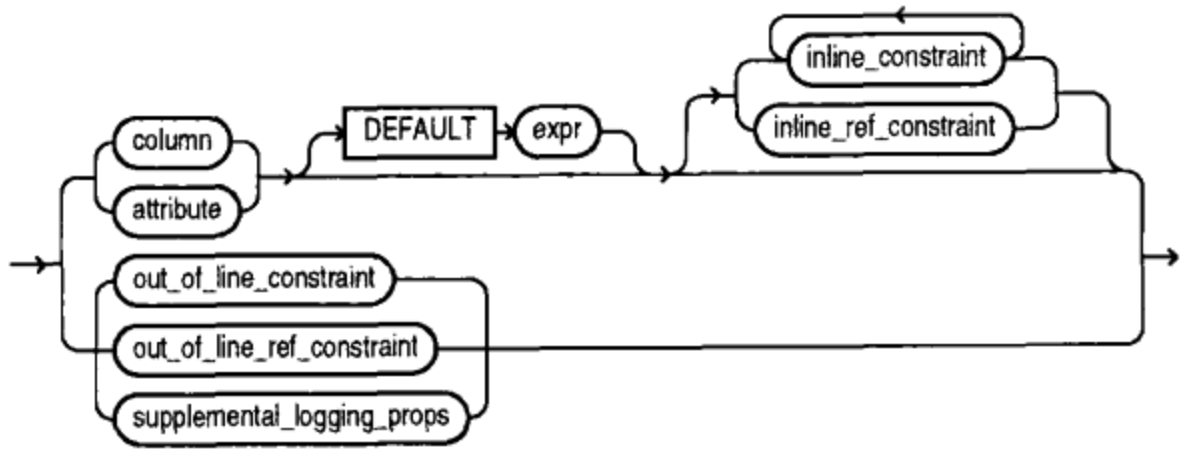
substitutable_column_clause::=



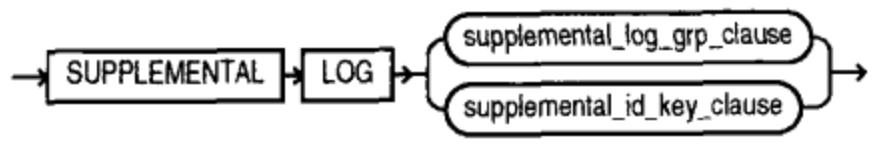
nested_table_col_properties::=



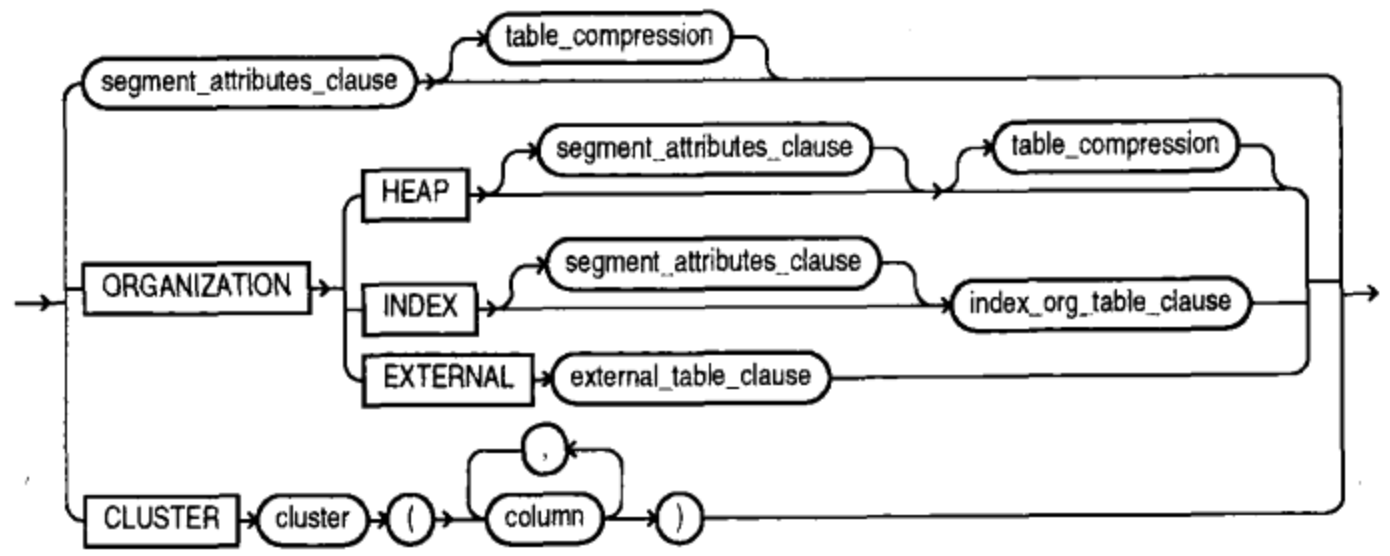
object_properties::=



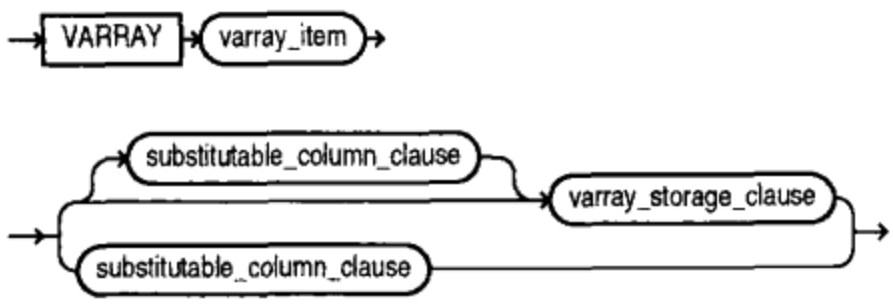
supplemental_logging_props::=



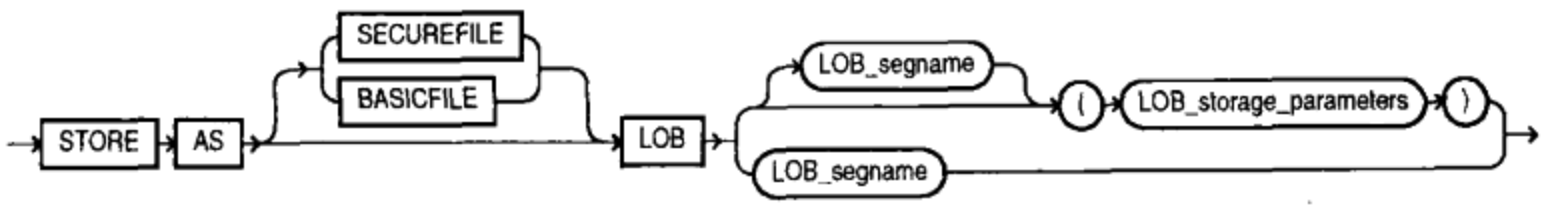
physical_properties::=



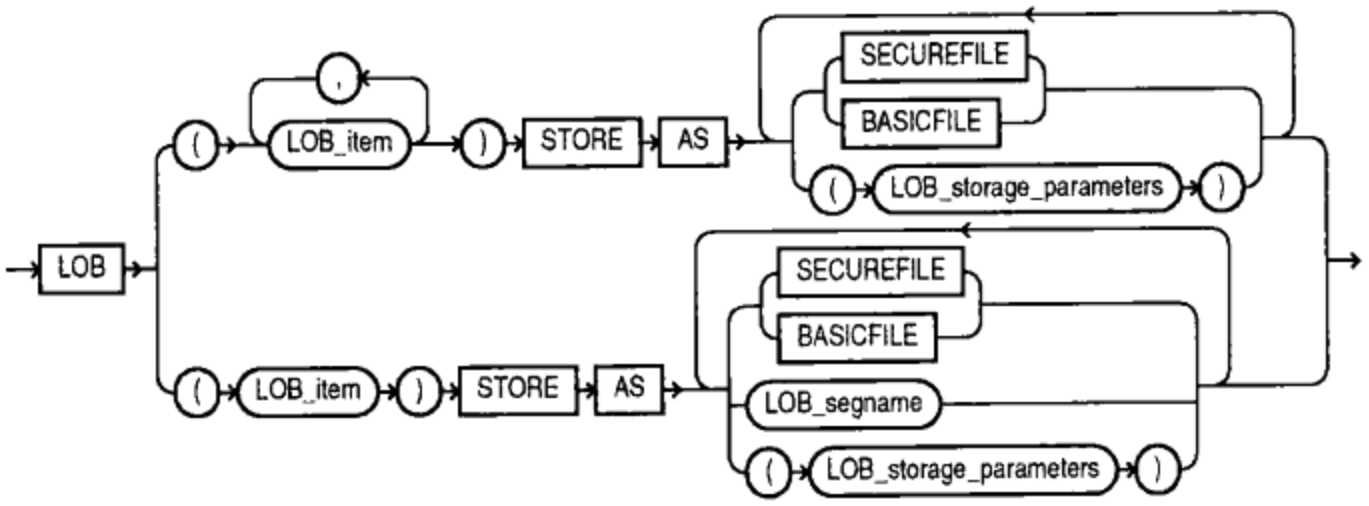
varray_col_properties::=



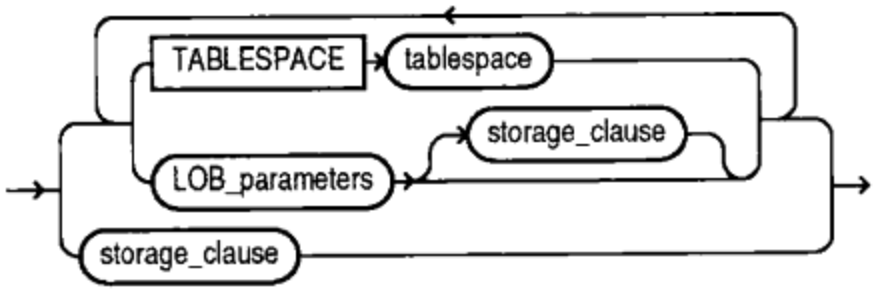
varray_storage_clause::=



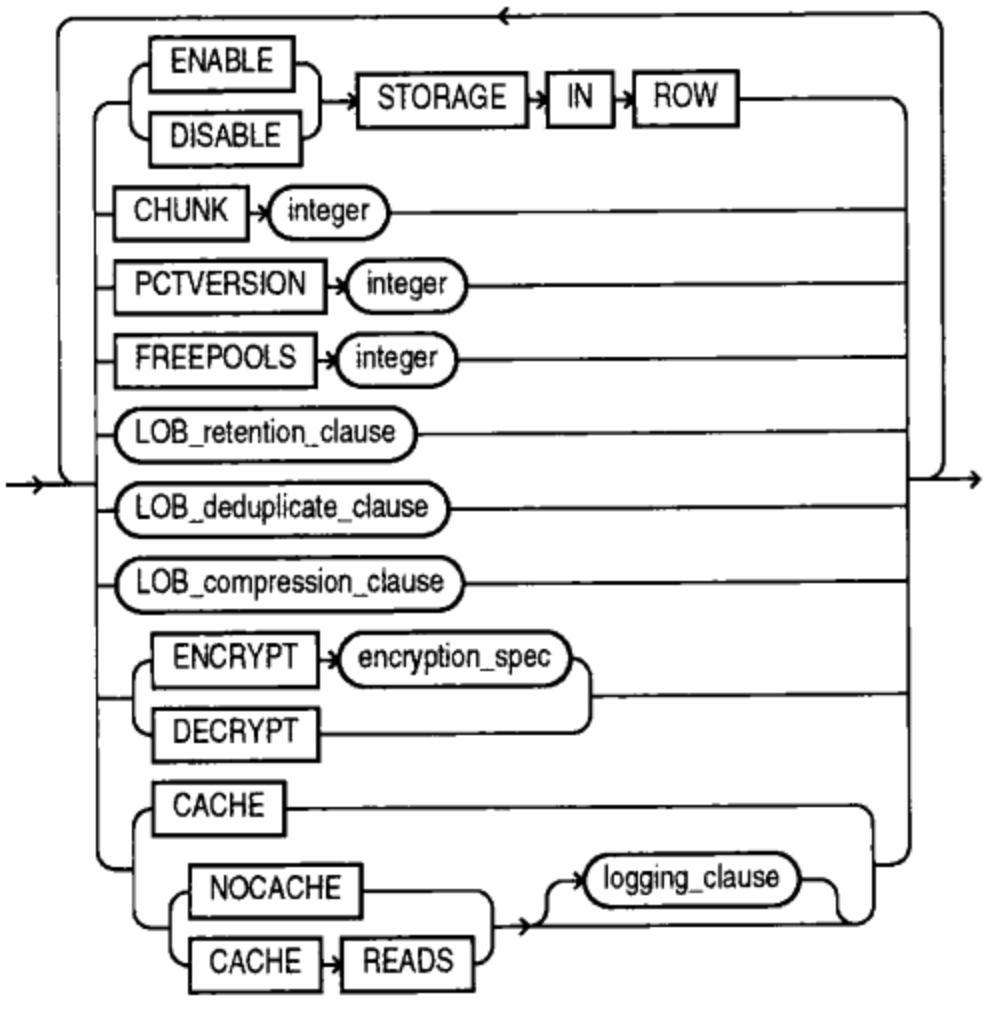
LOB_storage_clause::=



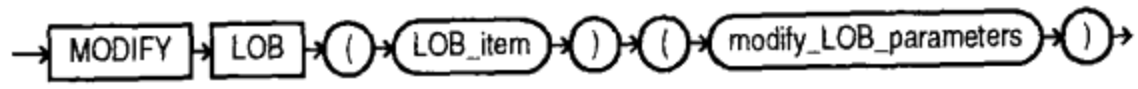
LOB_storage_parameters::=



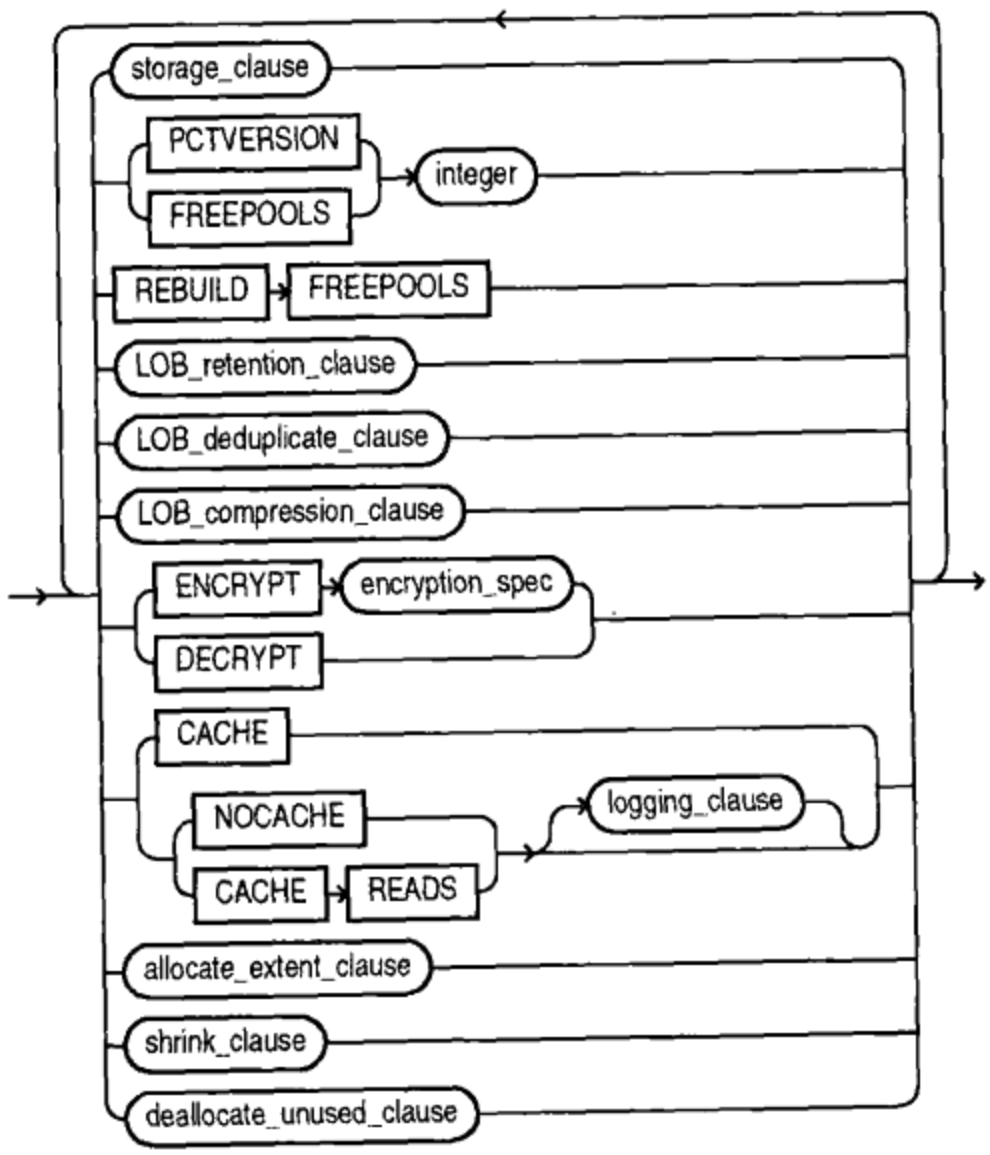
LOB_parameters::=



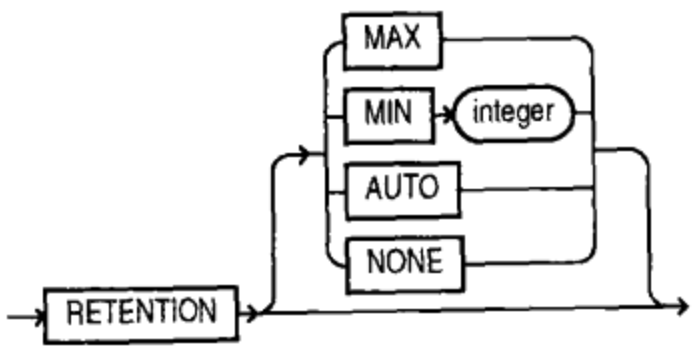
modify_LOB_storage_clause::=



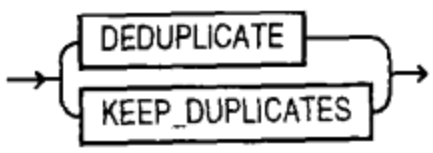
modify_LOB_parameters::=



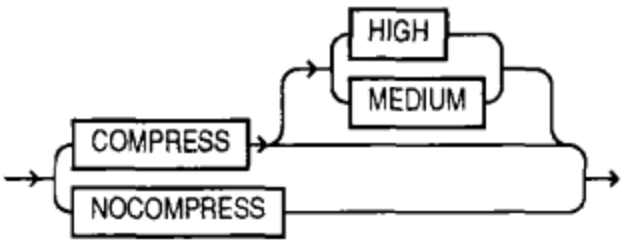
LOB_retention_clause::=



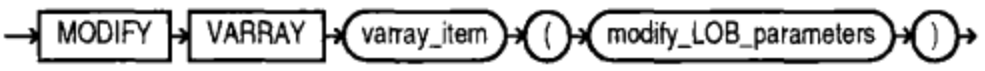
LOB_deduplicate_clause::=



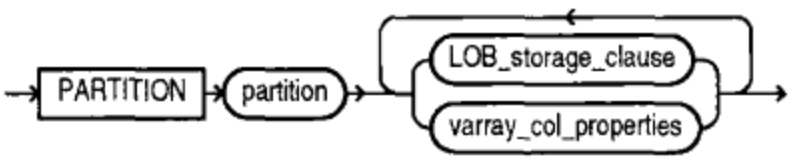
LOB_compression_clause::=

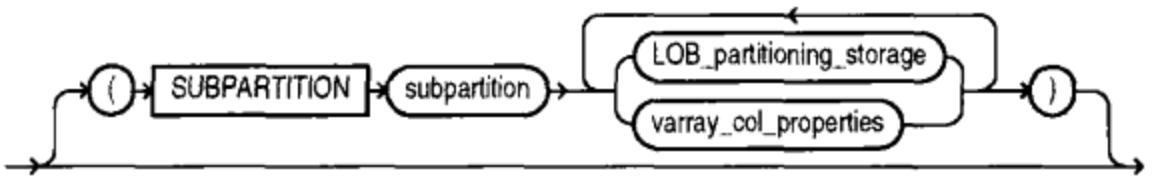


alter_varray_col_properties::=

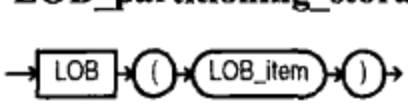


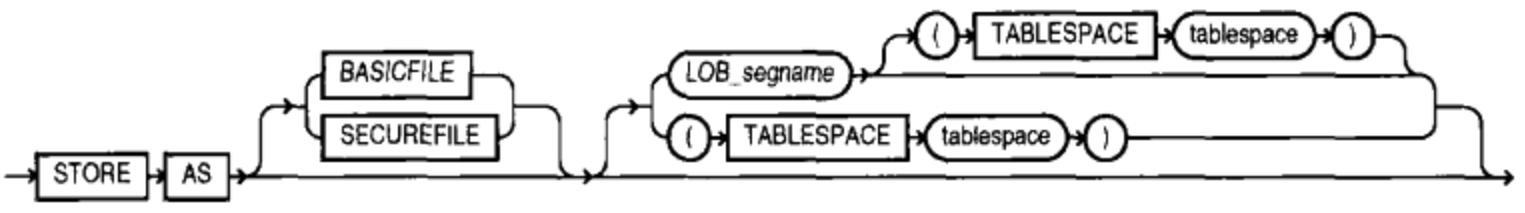
LOB_partition_storage::=





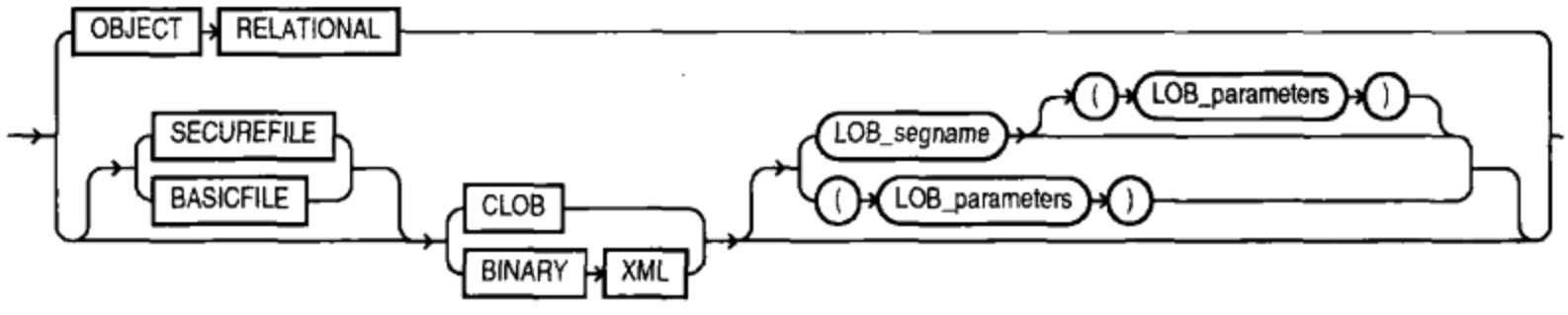
LOB_partitioning_storage::=



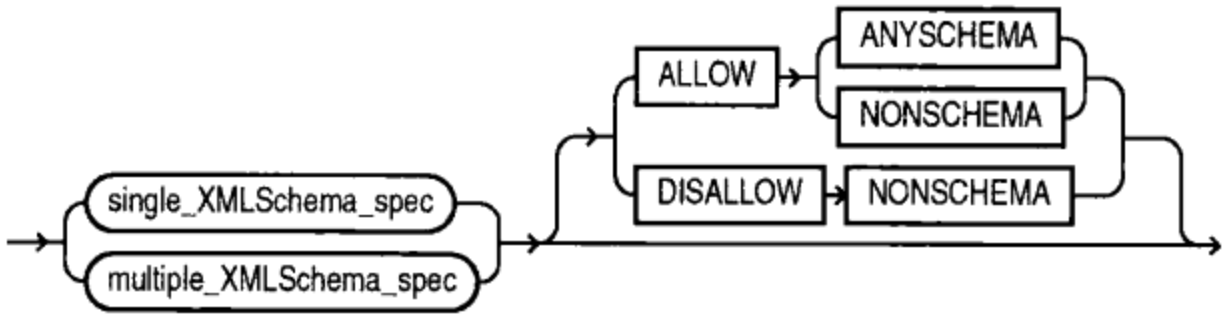


XMLType_column_properties::=

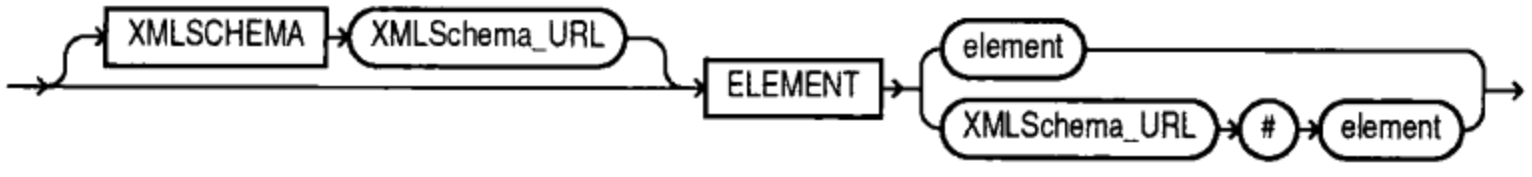
XMLType_storage::=



XMLSchema_spec::=

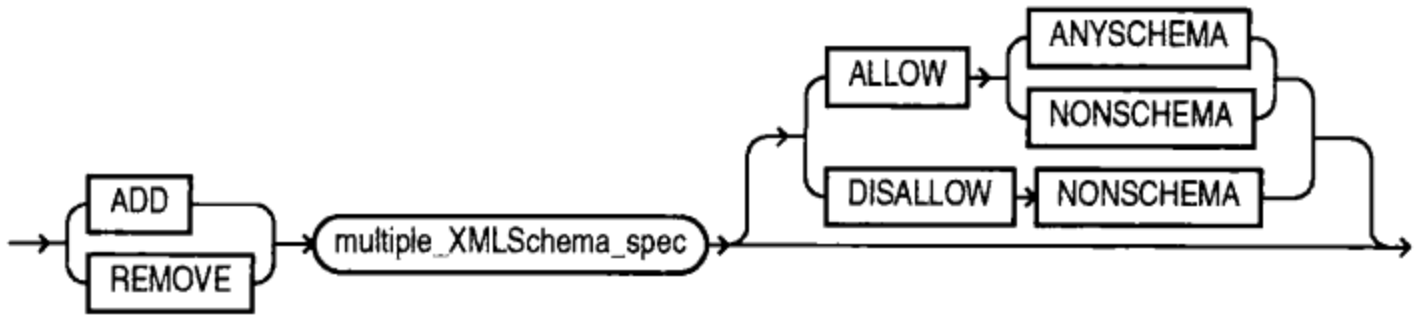


single_XMLSchema_spec::=

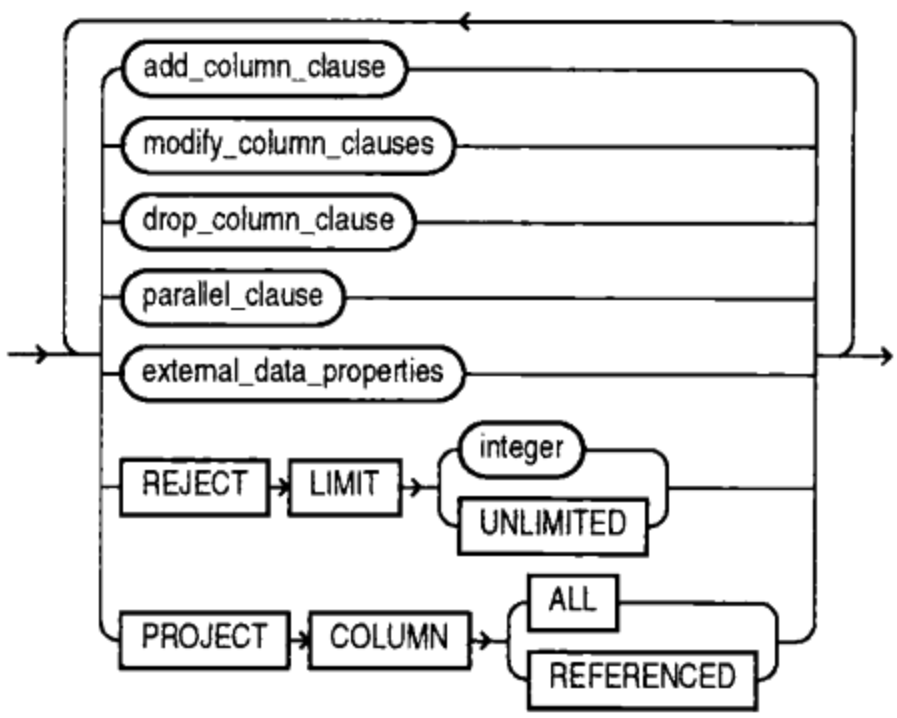


multiple_XMLSchema_spec::=

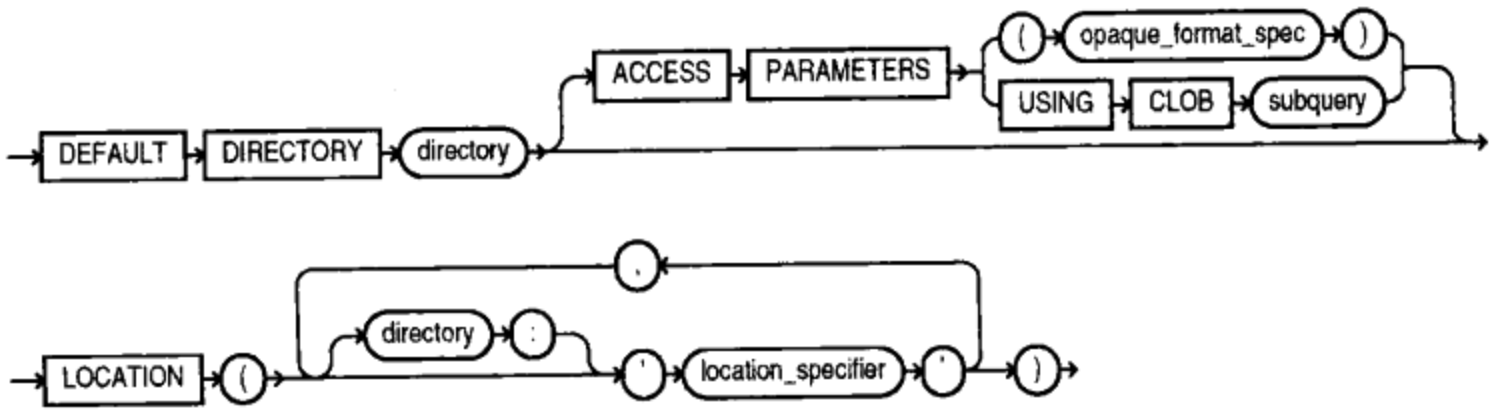
alter_XMLSchemas_clause::=



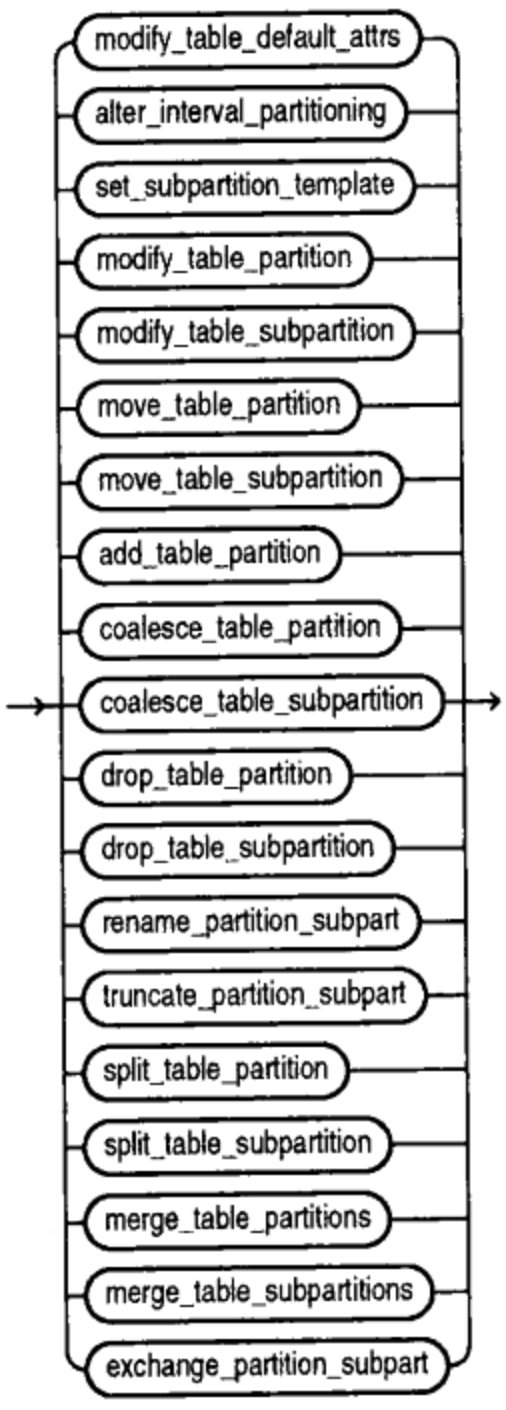
alter_external_table::=



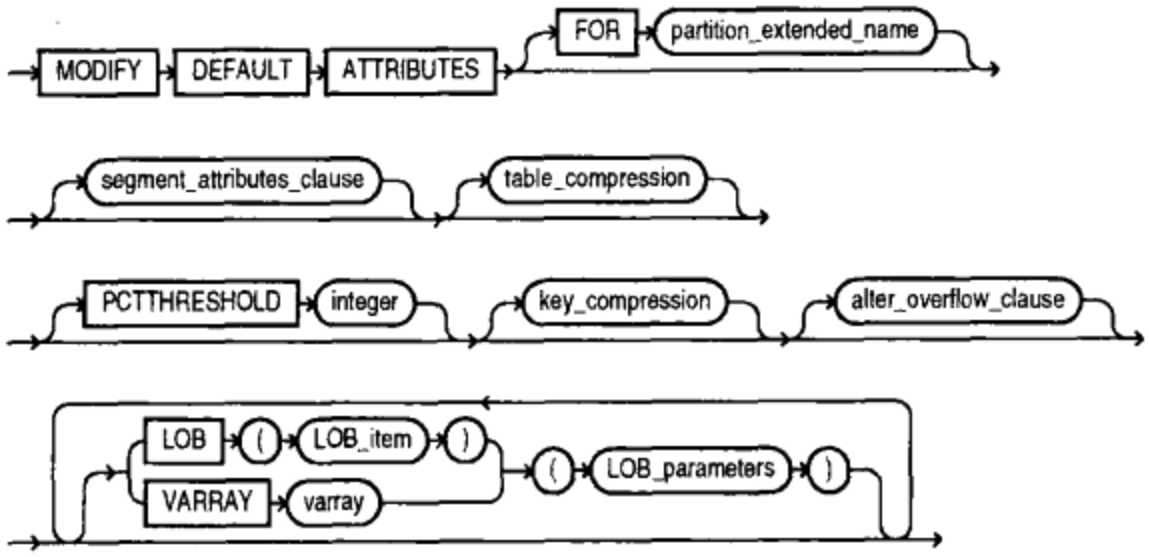
external_data_properties::=



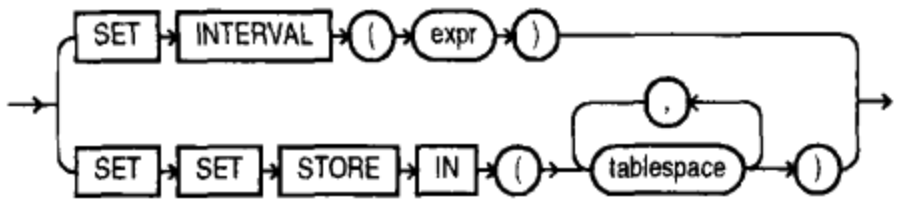
alter_table_partitioning::=



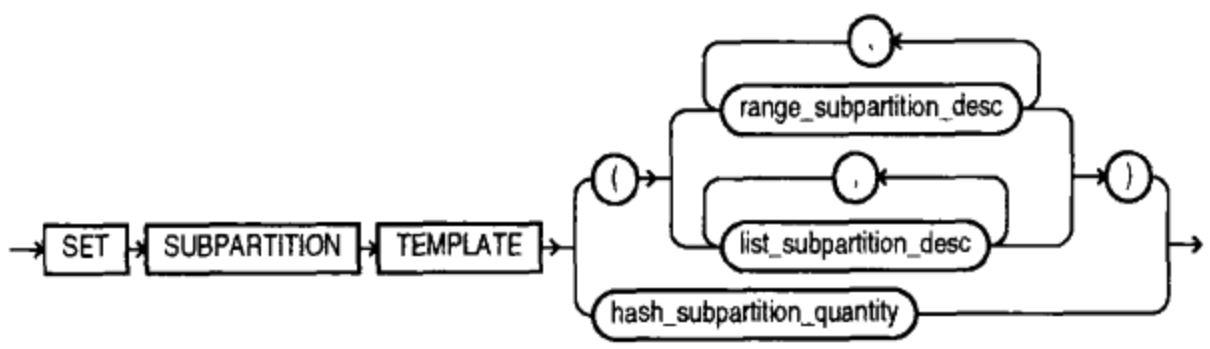
modify_table_default_attrs::=



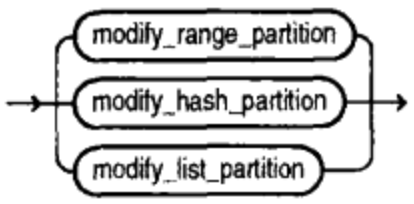
alter_interval_partitioning::=



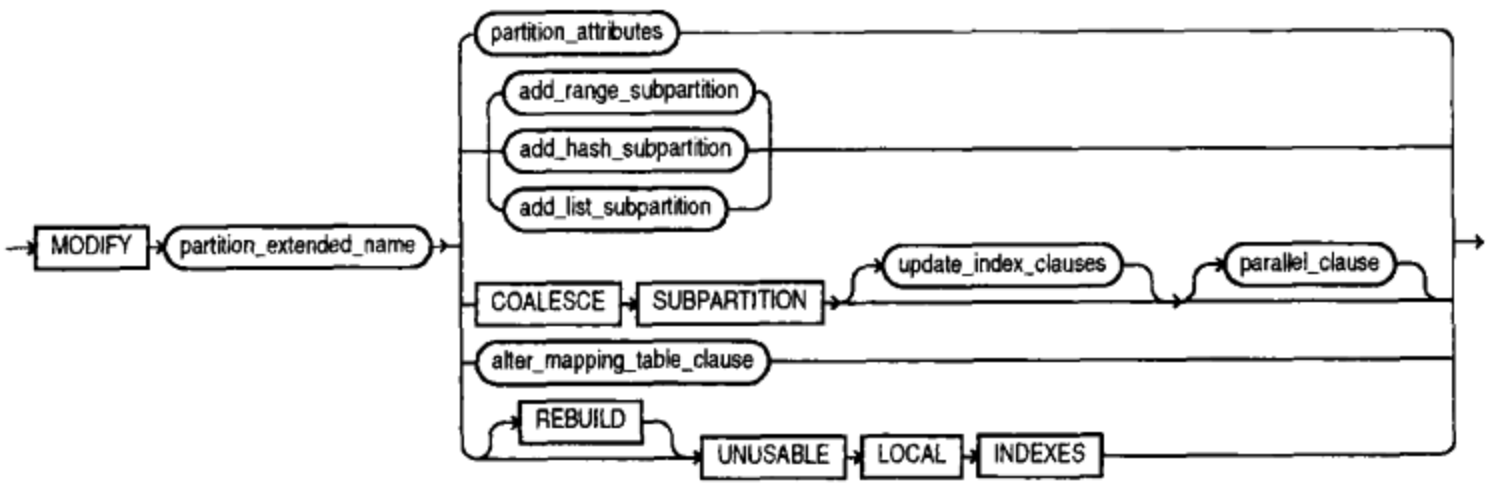
set_subpartition_template::=



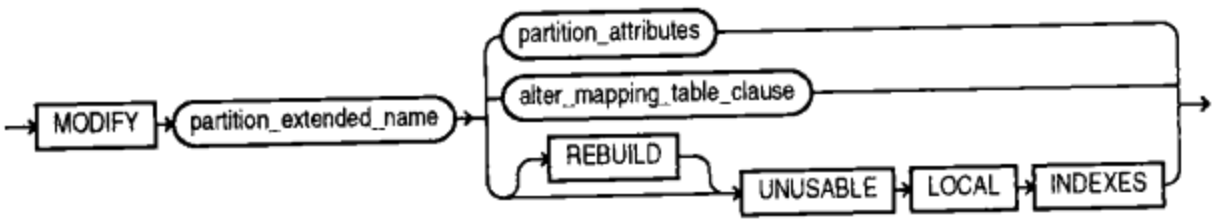
modify_table_partition::=



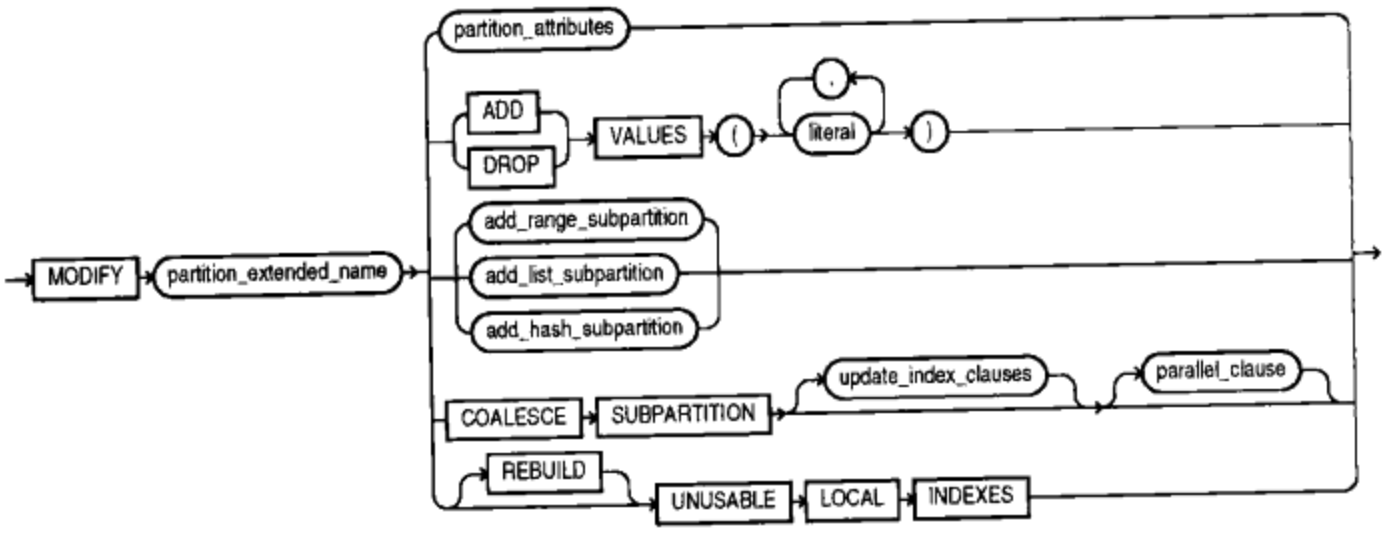
modify_range_partition::=



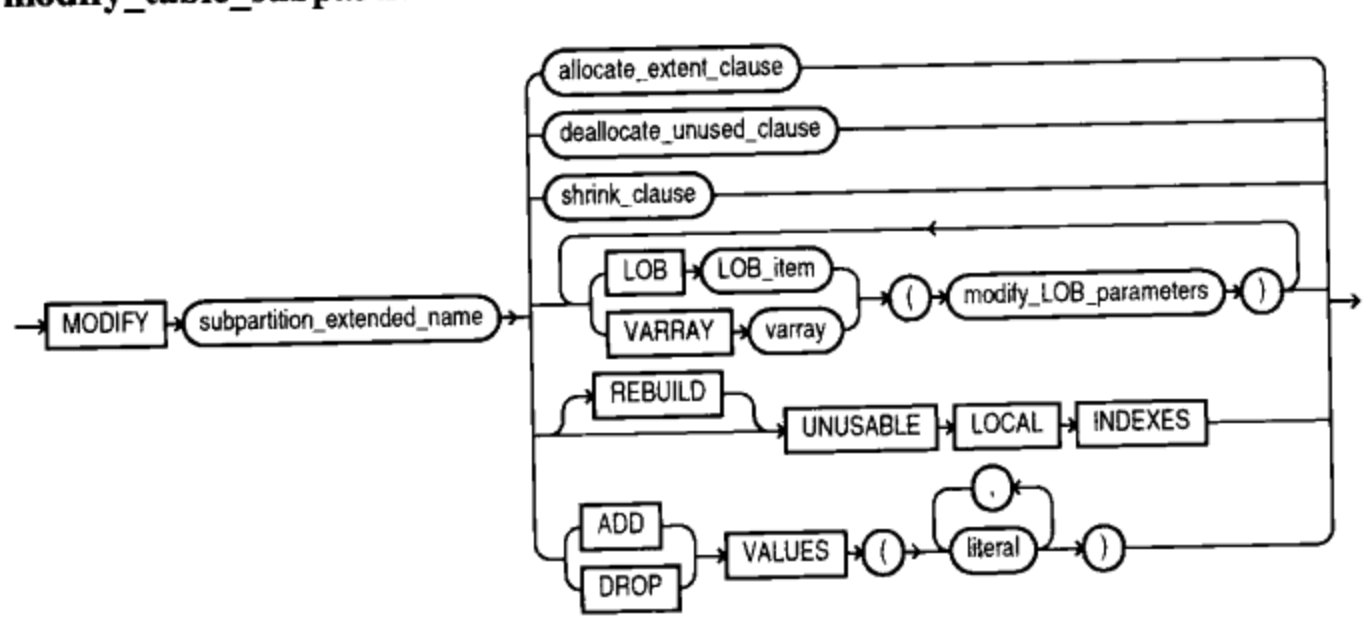
modify_hash_partition::=



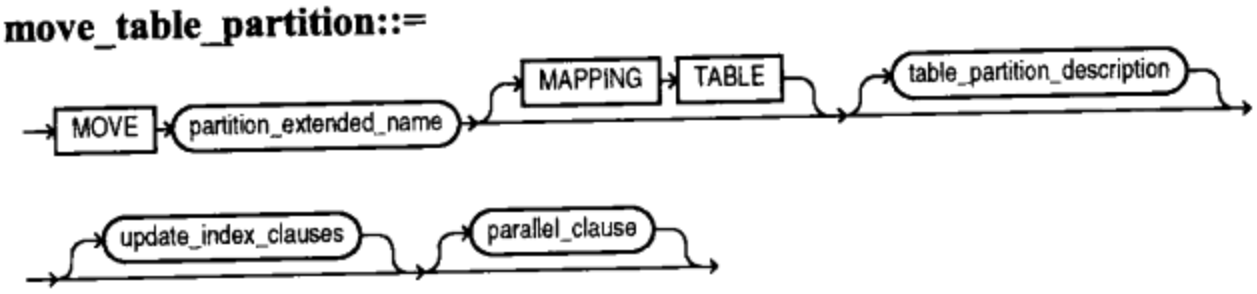
modify_list_partition::=



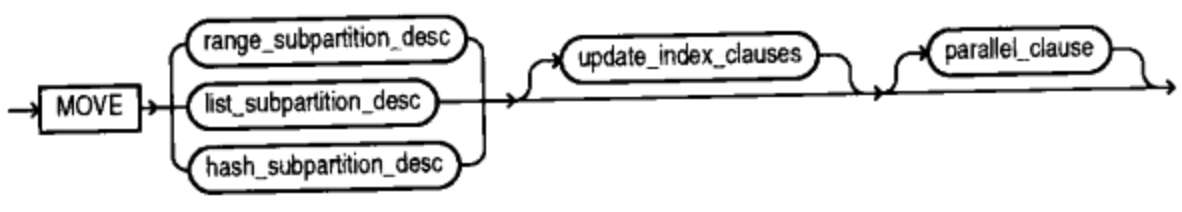
modify_table_subpartition::=



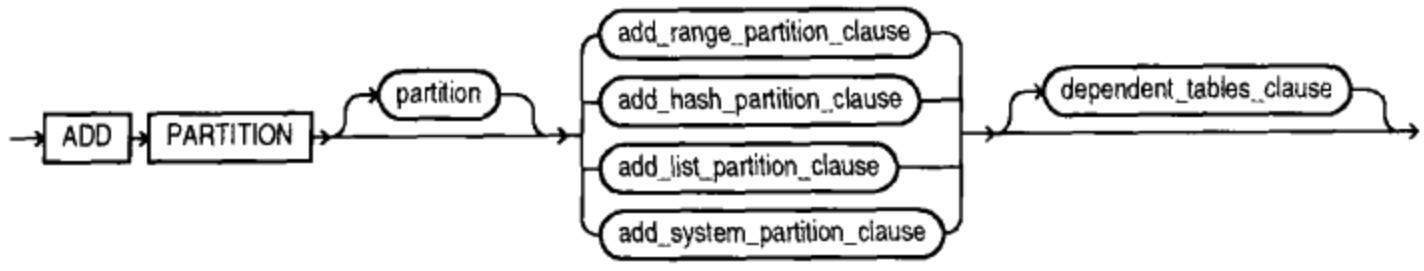
move_table_partition::=



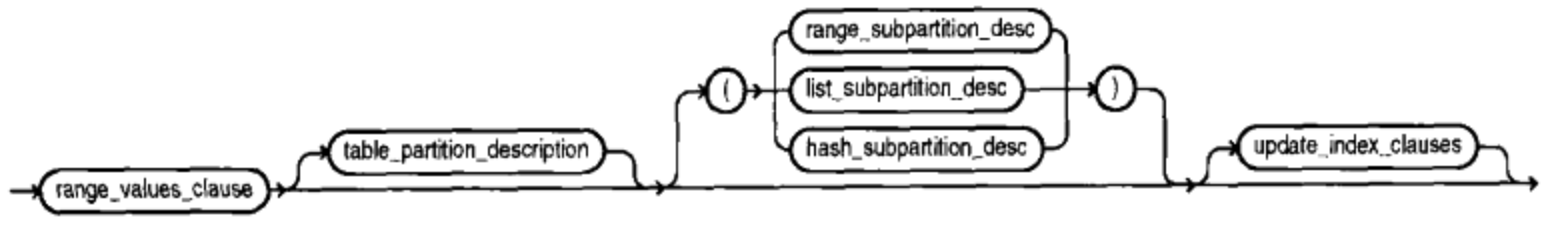
move_table_subpartition::=



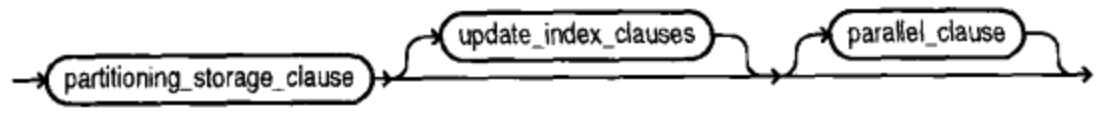
add_table_partition::=



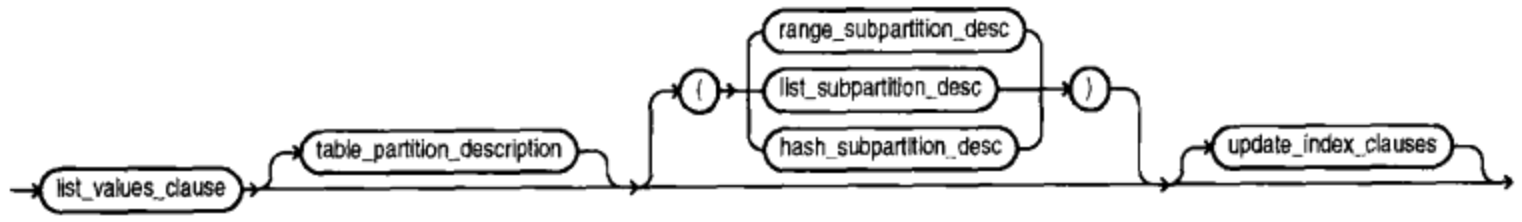
add_range_partition_clause::=



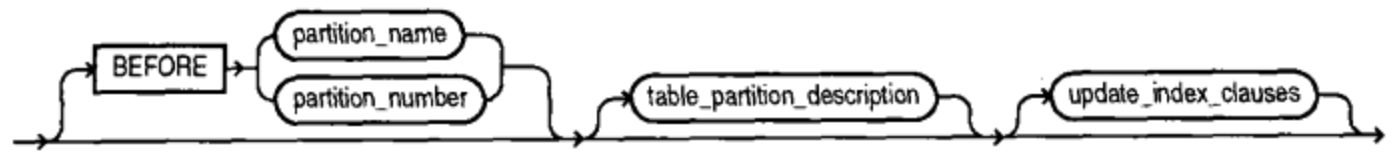
add_hash_partition_clause::=



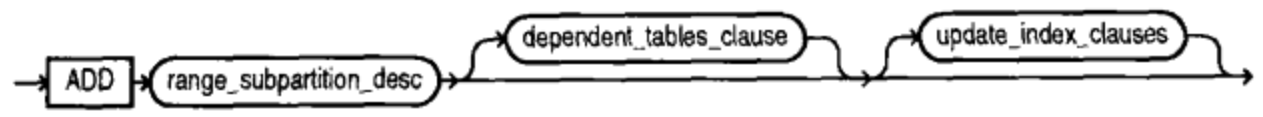
add_list_partition_clause::=



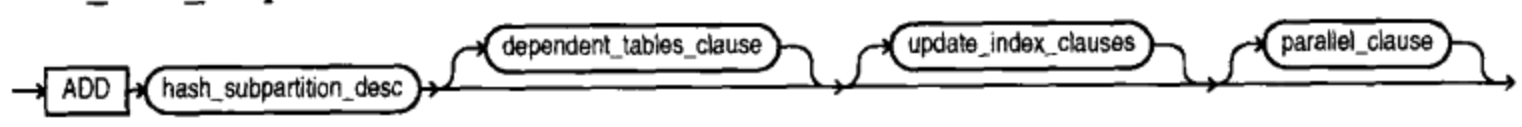
add_system_partition_clause::=



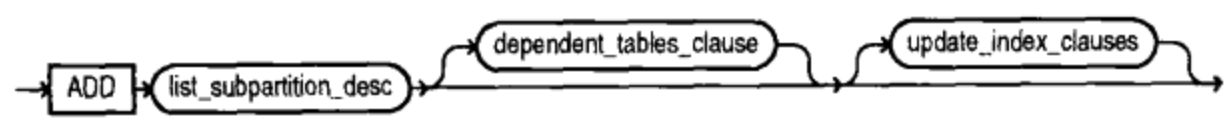
add_range_subpartition::=



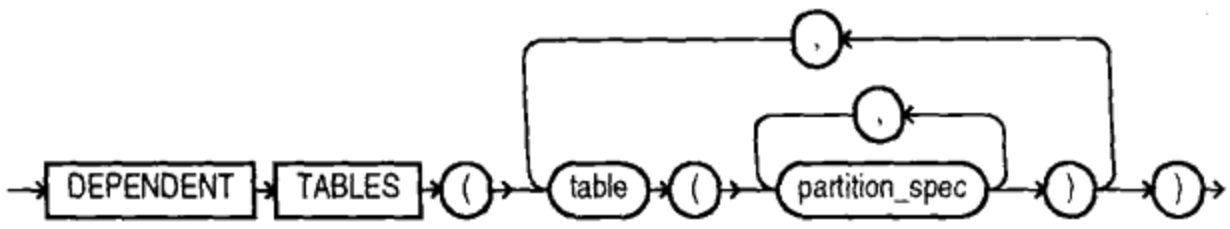
add_hash_subpartition::=



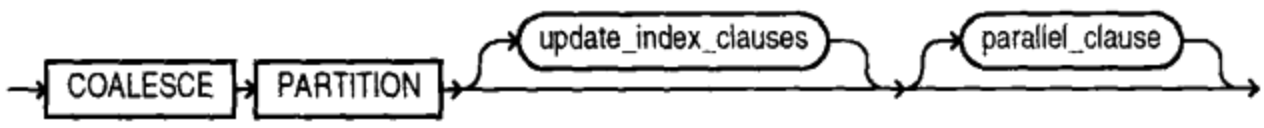
add_list_subpartition::=



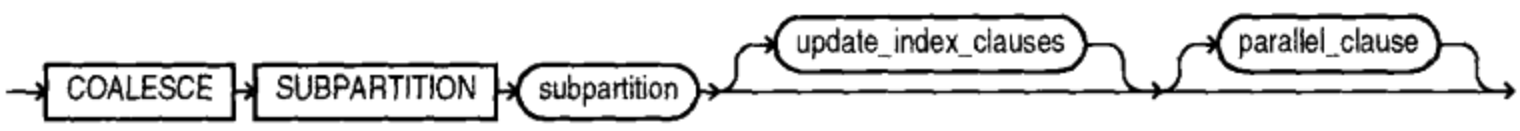
dependent_tables_clause:=



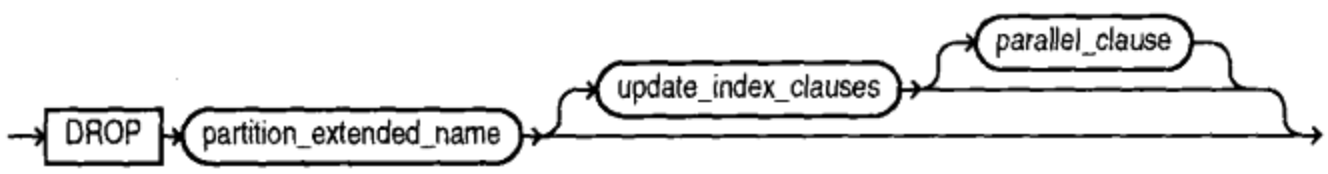
coalesce_table_partition::=



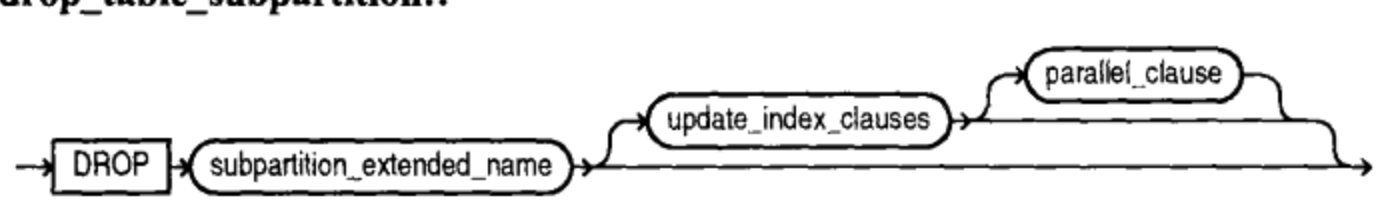
coalesce_table_subpartition::=



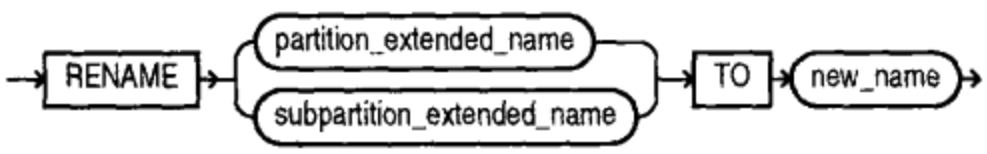
drop_table_partition::=



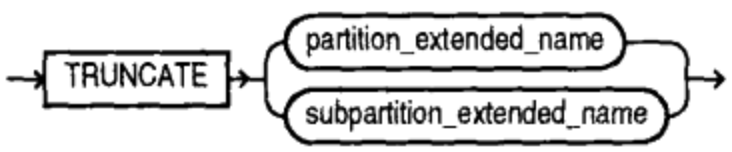
drop_table_subpartition::=



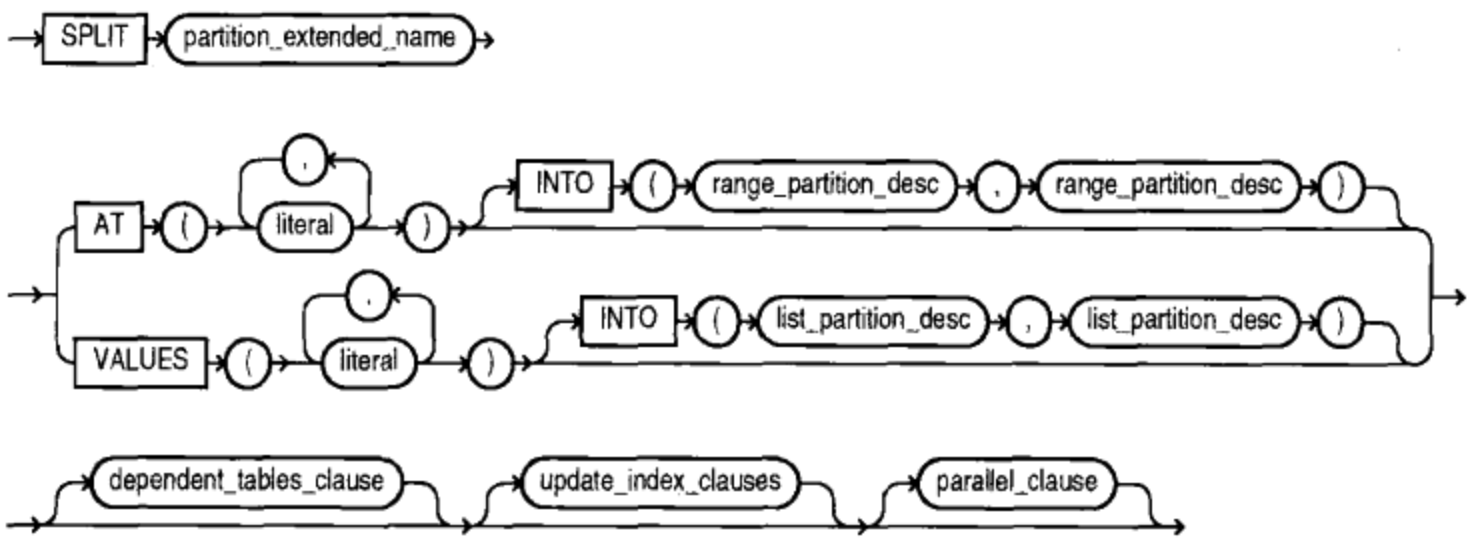
rename_partition_subpart::=



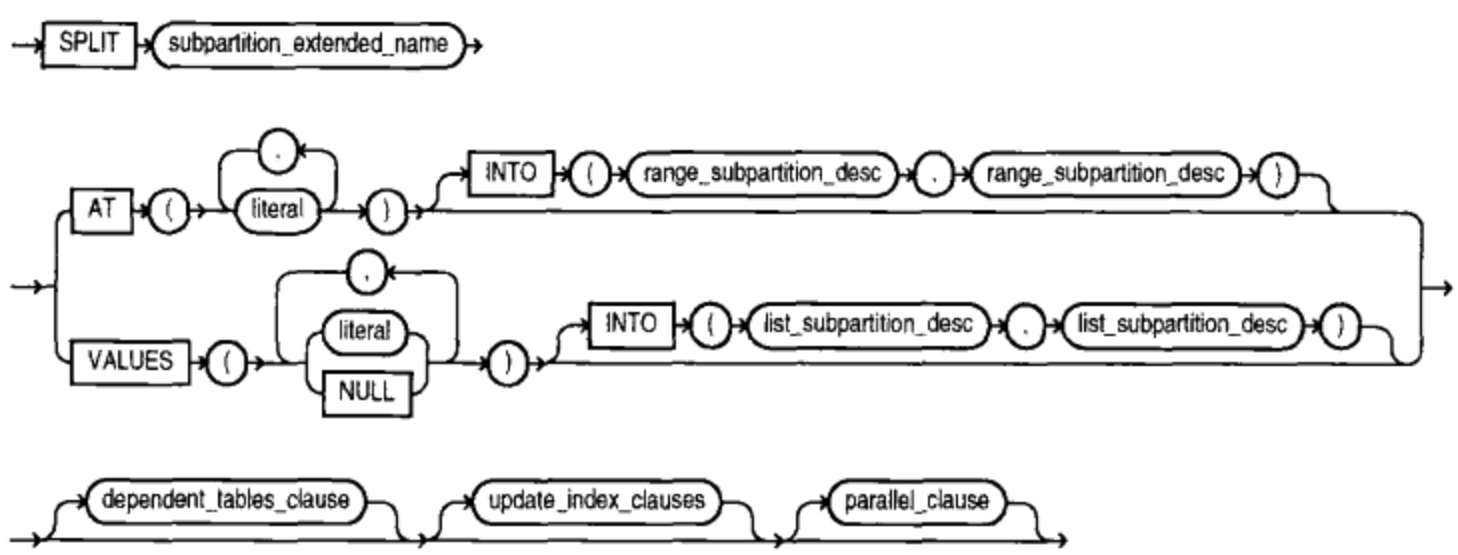
truncate_partition_subpart::=



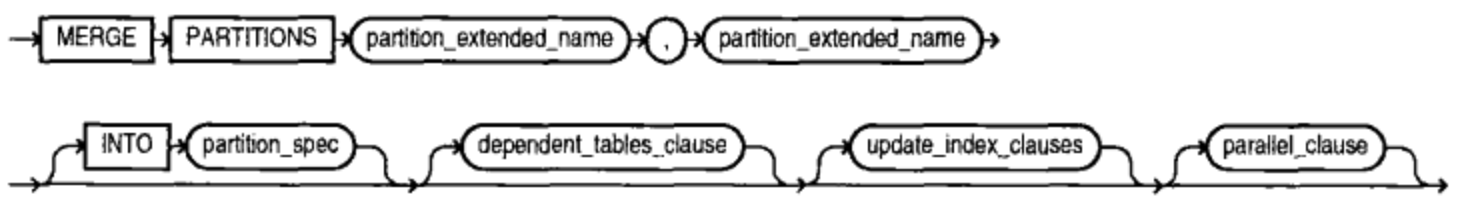
split_table_partition::=



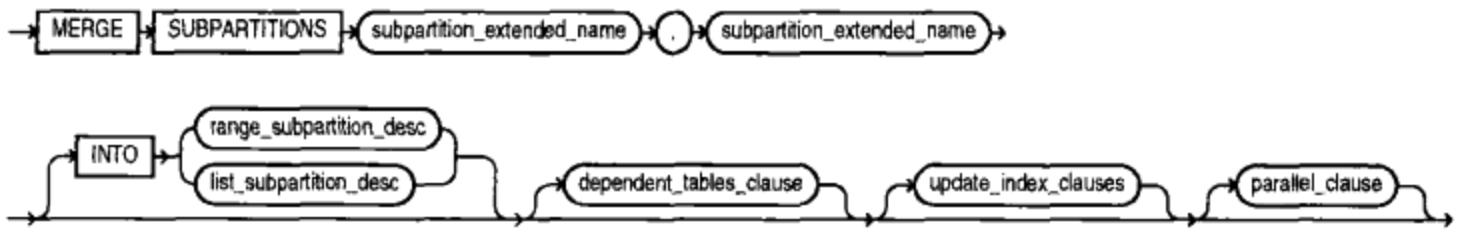
split_table_subpartition::=



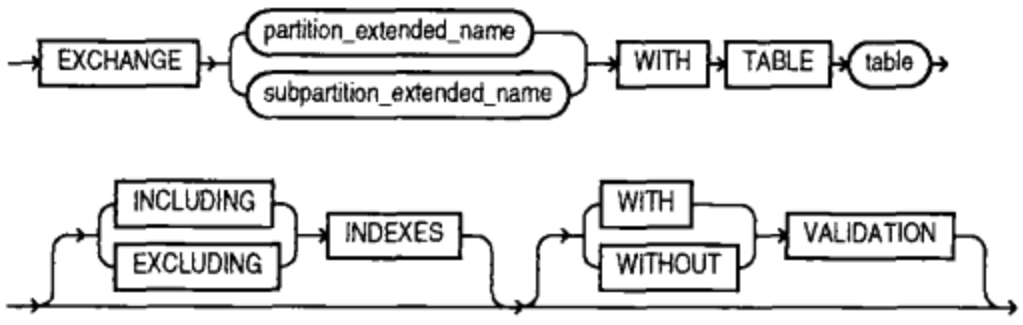
merge_table_partitions::=

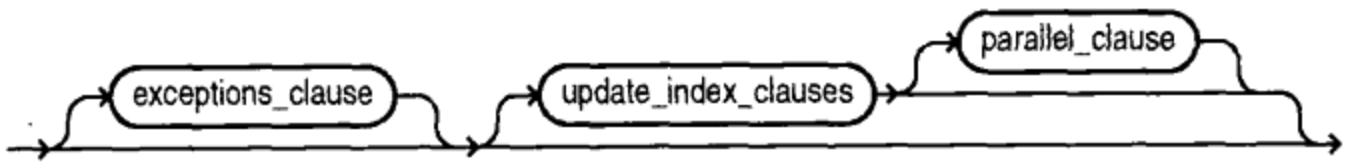


merge_table_subpartitions::=



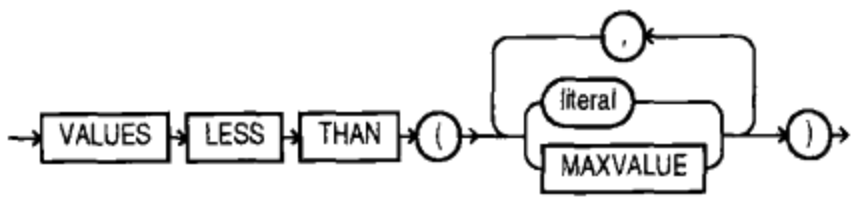
exchange_partition_subpart::=



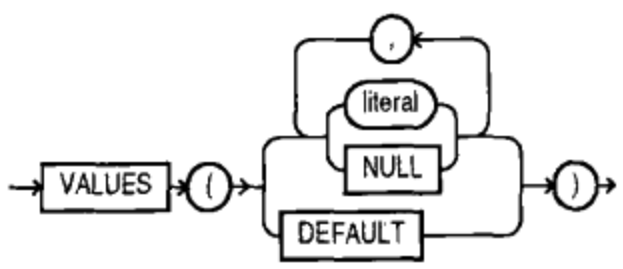


exceptions_clause::=

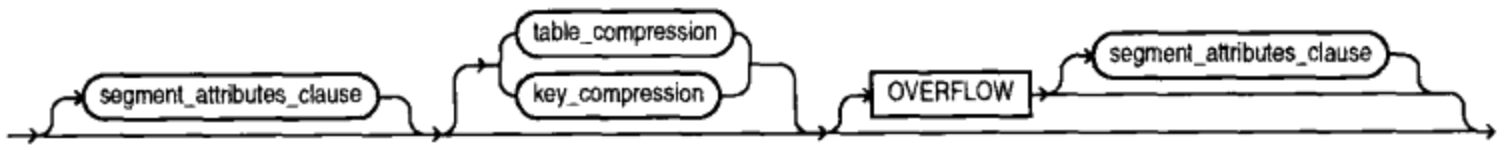
range_values_clause::=

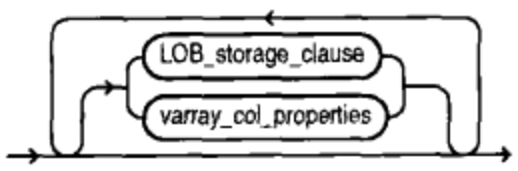


list_values_clause::=

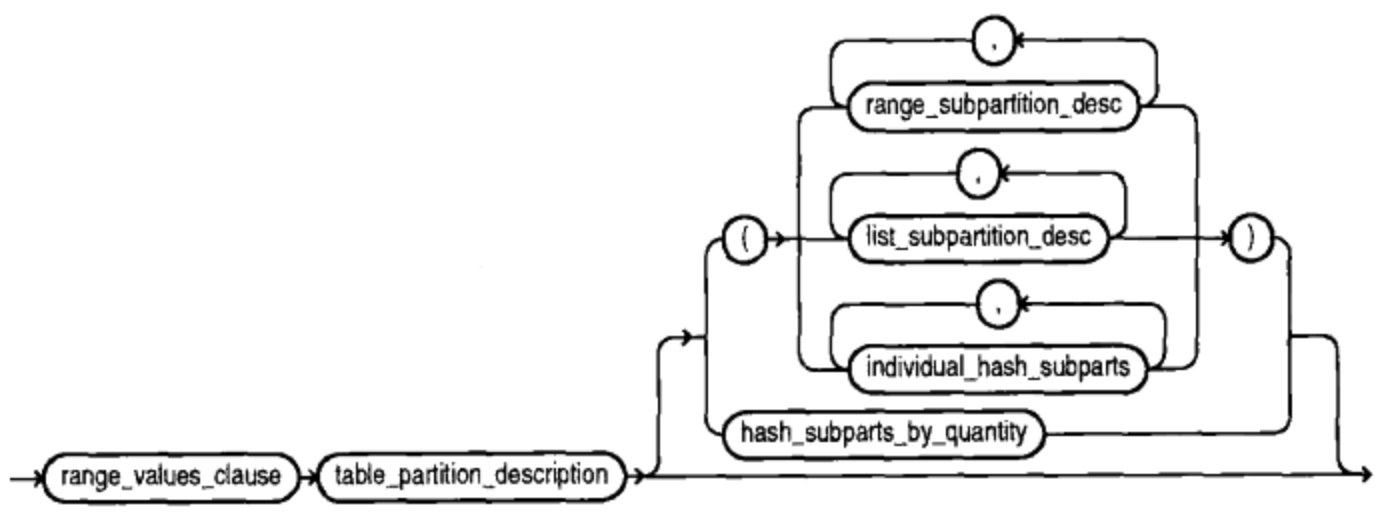


table_partition_description::=

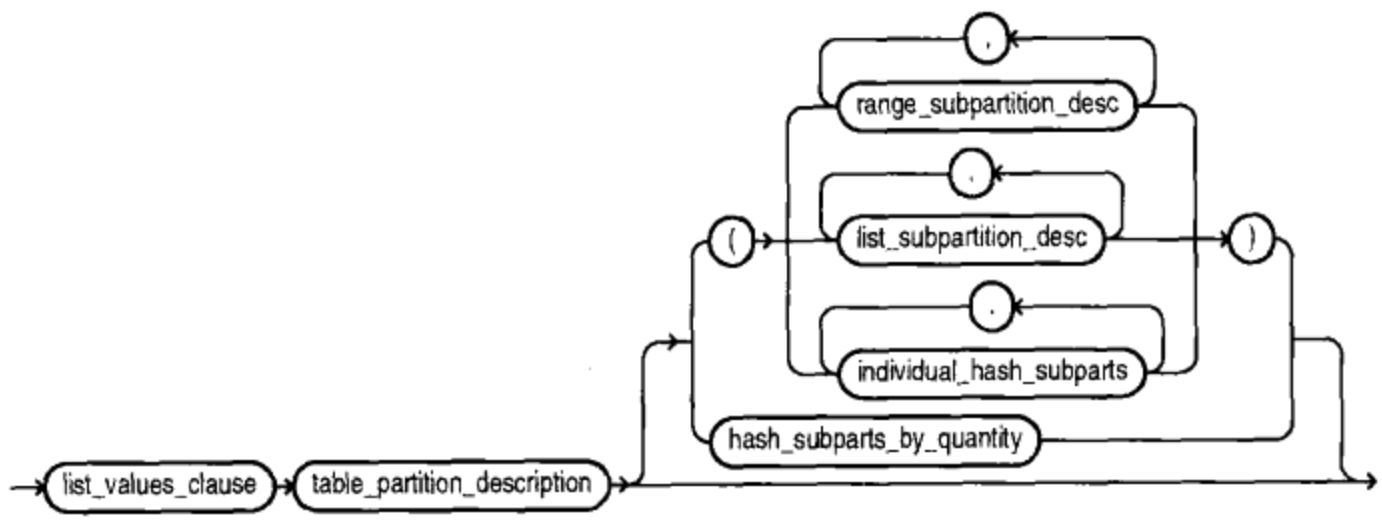




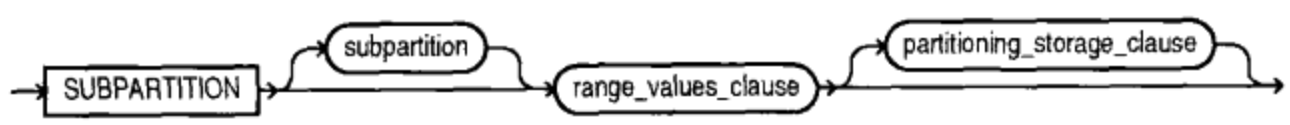
range_partition_desc::=



list_partition_desc::=

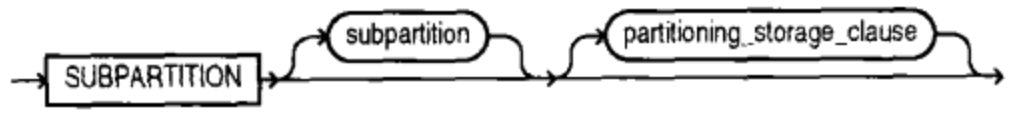


range_subpartition_desc::=

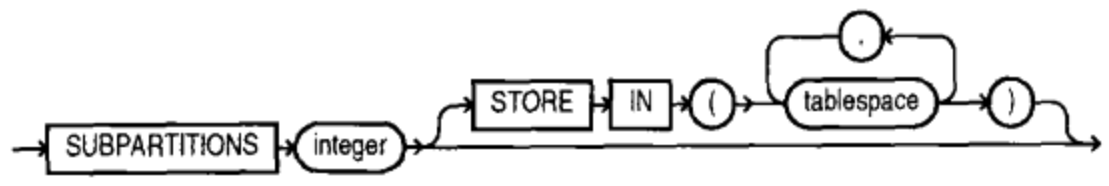


list_subpartition_desc::=

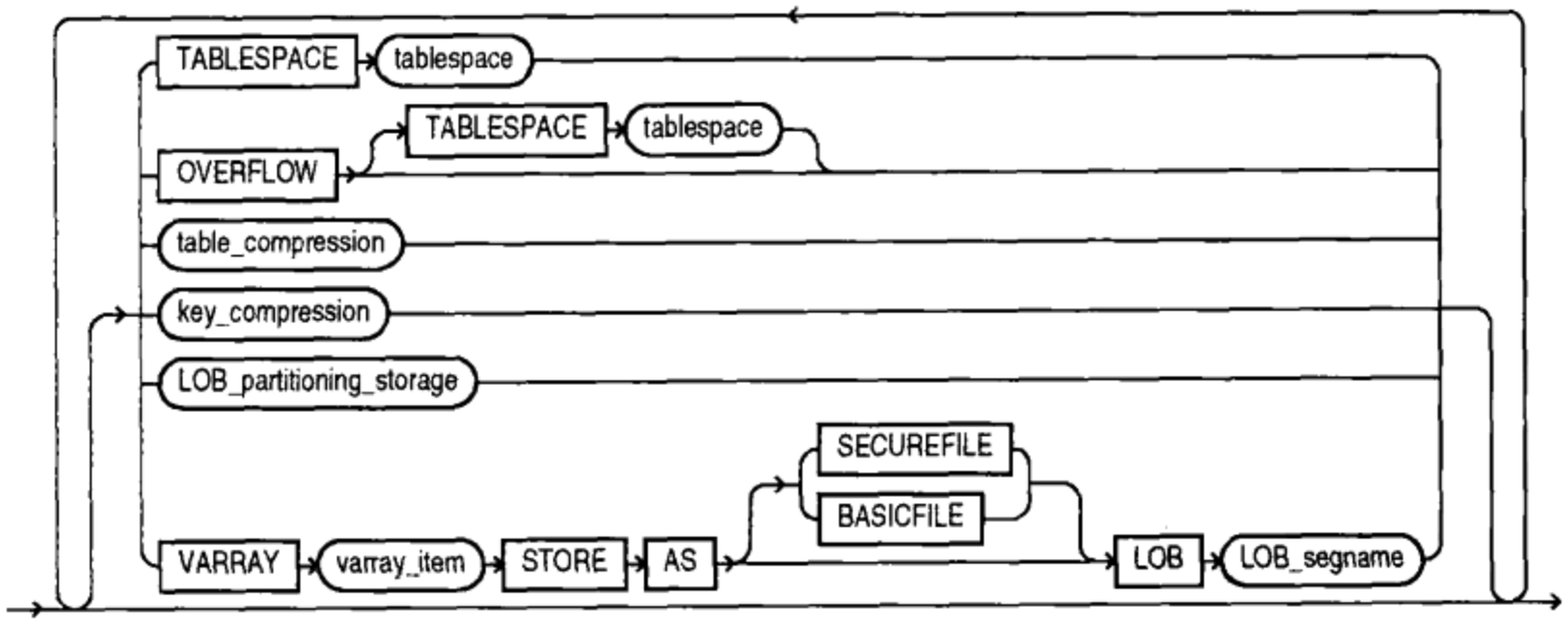
individual_hash_subparts::=



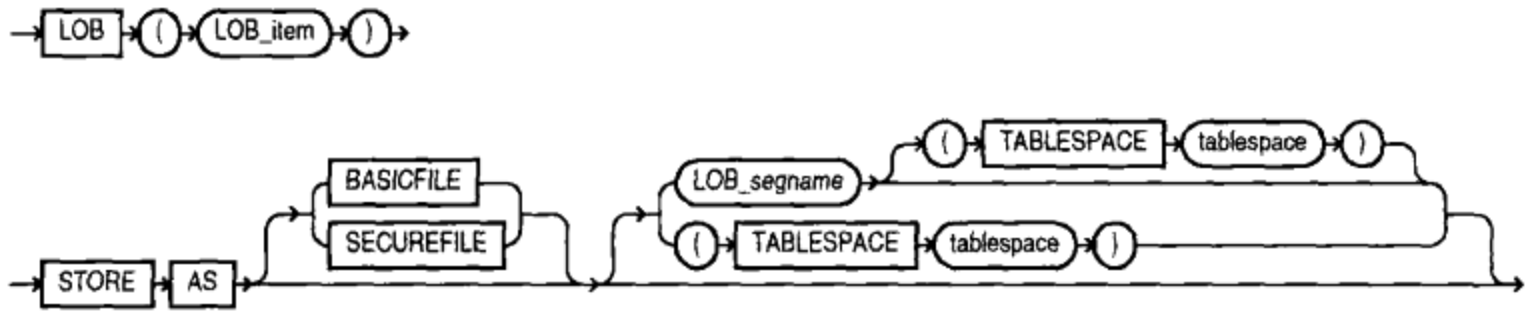
hash_subparts_by_quantity::=



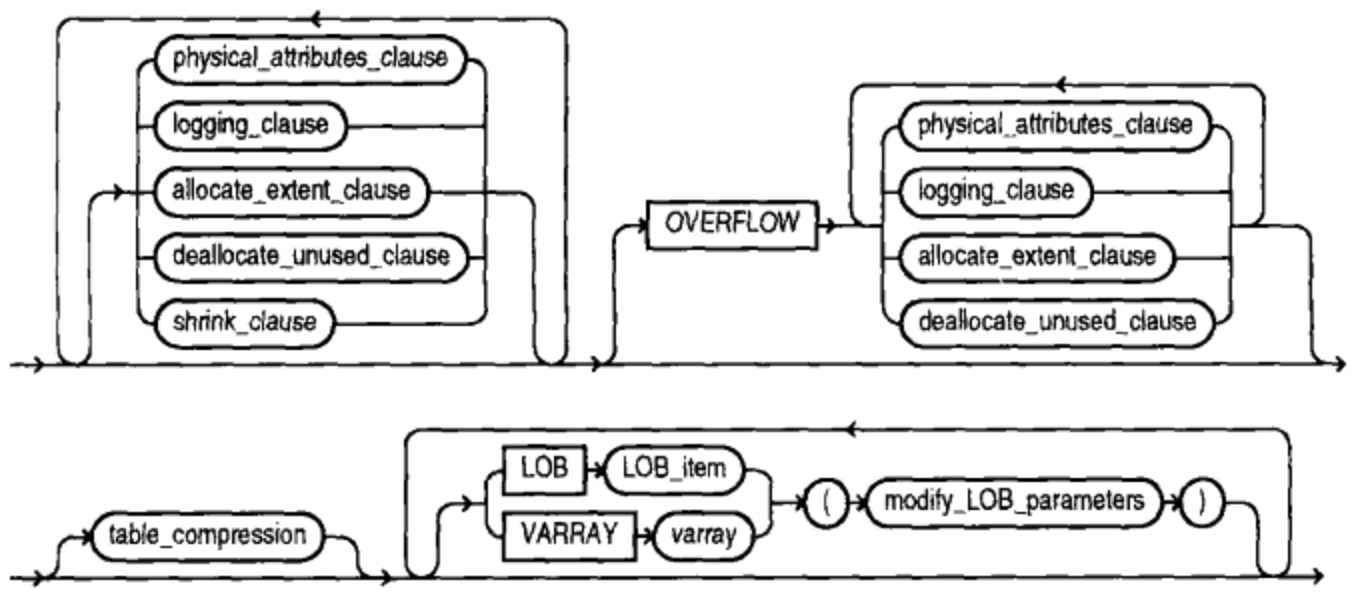
partitioning_storage_clause::=



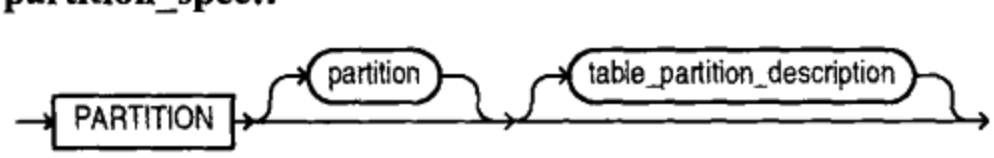
LOB_partitioning_storage::=



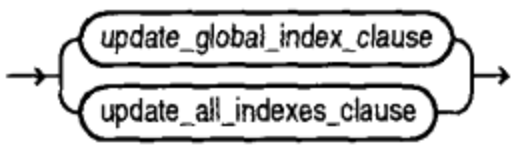
partition_attributes::=



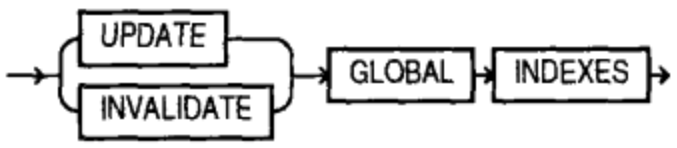
partition_spec::=



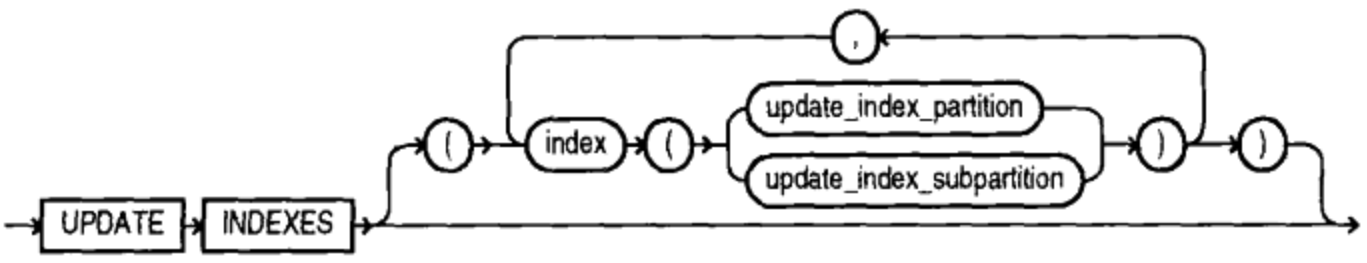
update_index_clauses::=



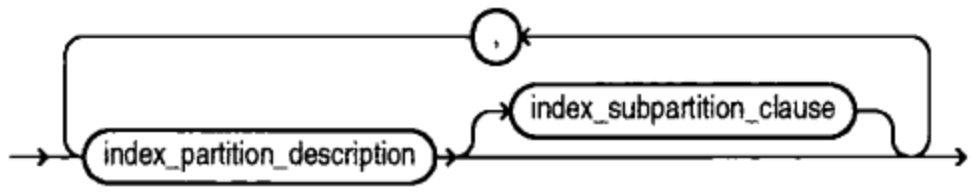
update_global_index_clause::=



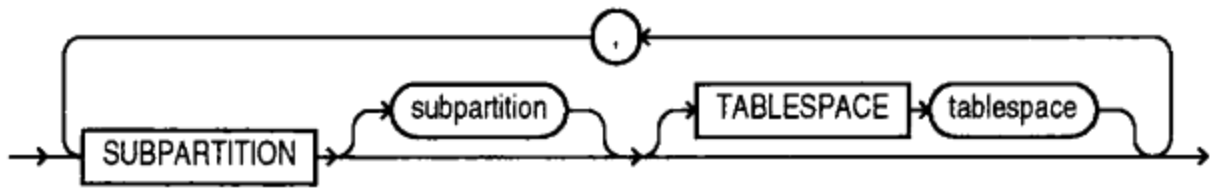
update_all_indexes_clause::=



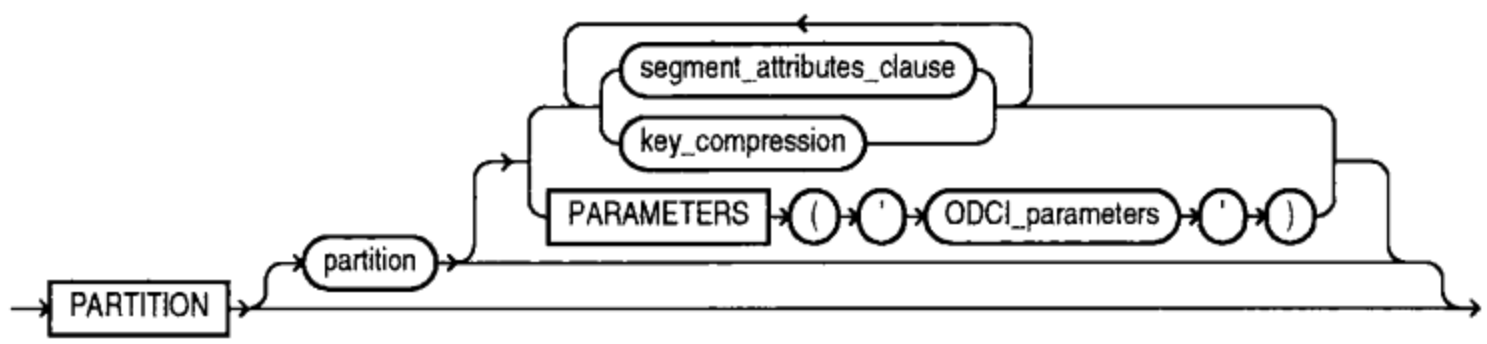
update_index_partition::=



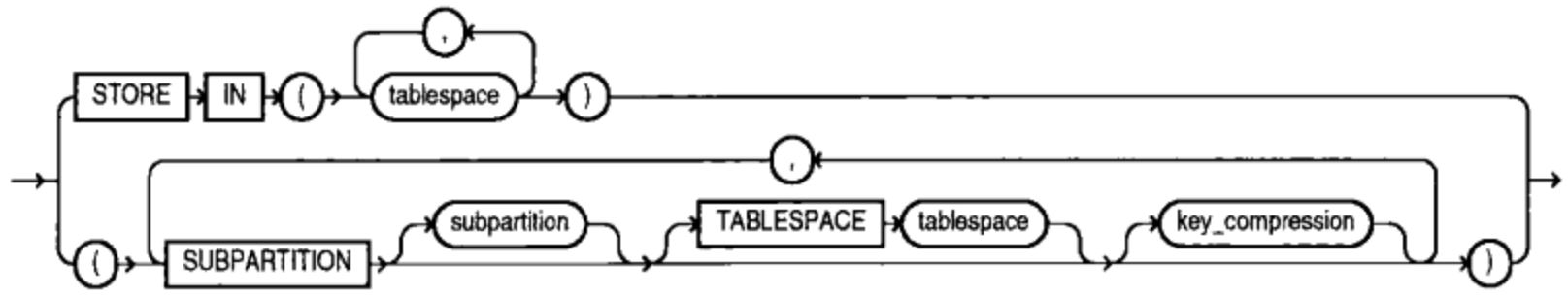
update_index_subpartition::=



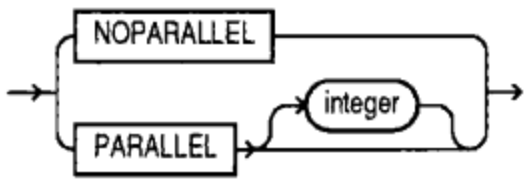
index_partition_description::=



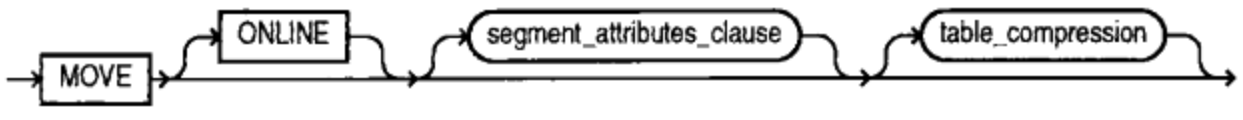
index_subpartition_clause::=

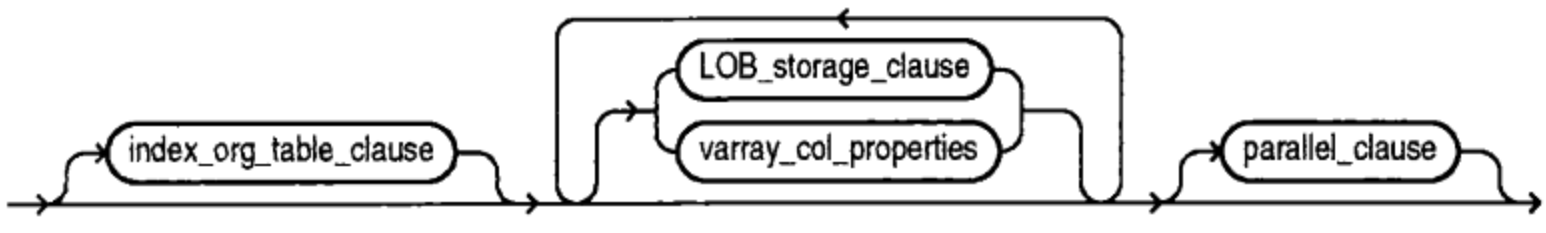


parallel_clause::=

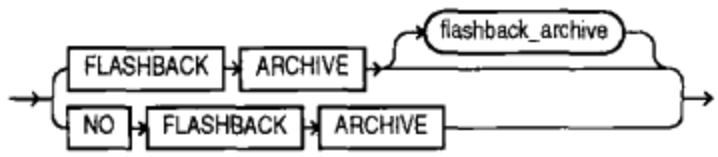


move_table_clause::=

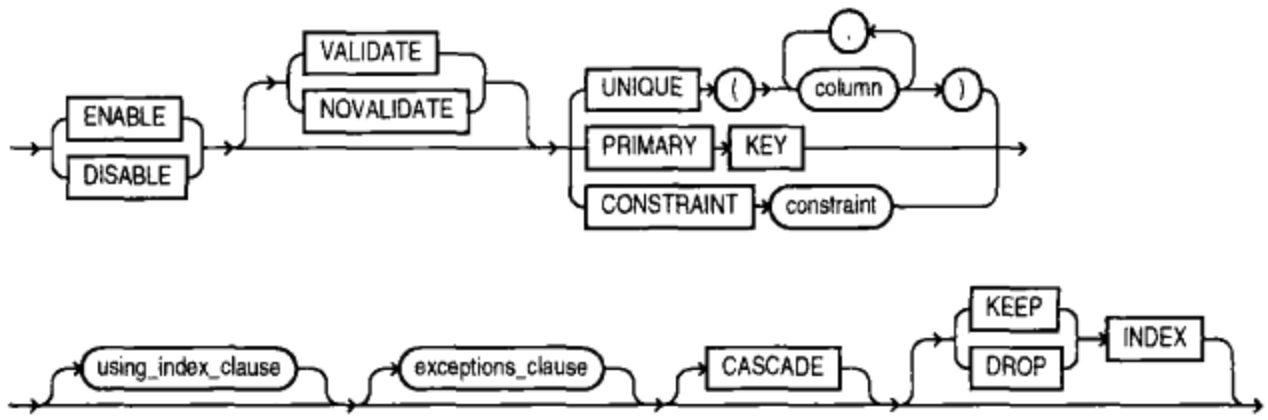




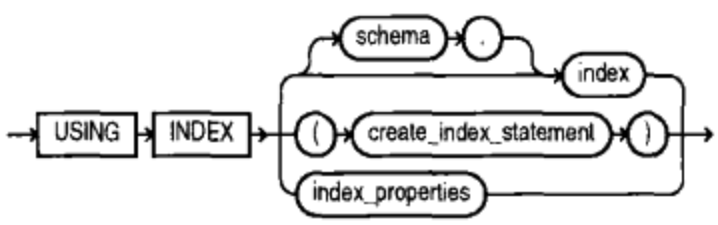
flashback_archive_clause::=



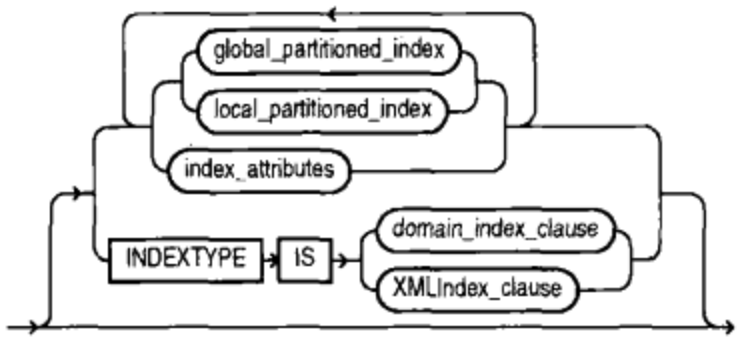
enable_disable_clause::=



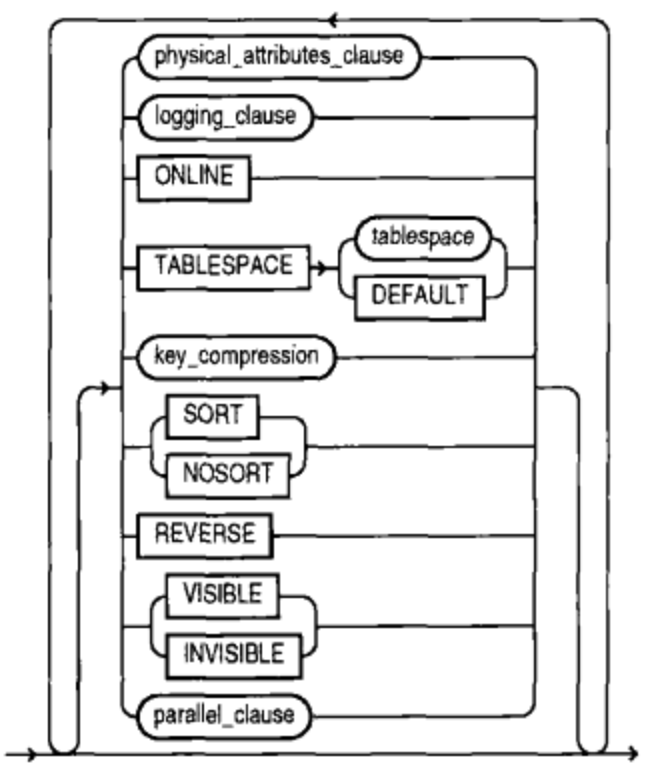
using_index_clause::=



index_properties::=



index_attributes::=



描述：ALTER TABLE 修改已有表的定义。ADD 允许在已有表的结尾新添加一列，或者给表的定义添加一个约束。它们都遵循与在 CREATE TABLE 中所用的相同格式。

MODIFY 修改已有的一列，但有如下的限制：

- 仅当表的每一行中的列都为 NULL 时，才能修改该列的类型或减小其大小。
- NOT NULL 列只能添加到没有行的表中。
- 仅当已有列在每行中都具有非 NULL 值或在更改时被指定为默认值时，才能将其修改为 NOT NULL。
- 增加没有指定 NULL 的 NOT NULL 列的长度，将会使它保持 NOT NULL。

可以用 SET UNUSED 子句标记列为未使用。在使用 DROP UNUSED COLUMNS 子句时，会重新组织表，且所有未使用的列都会被删除。

ALLOCATE EXTENT 允许显式地新分配一个区。ENABLE 和 DISABLE 分别用于启用和禁用约束。除了应用于已有的表外，ALTER TABLE 其他功能的作用方式与 CREATE TABLE 中的一样。详细内容请参阅 CREATE TABLE。

要更改表，必须拥有该表的 ALTER 权限或者拥有 ALTER ANY TABLE 系统权限。

CACHE 指定：在读取时，将该表的块标记为“最近使用”，并尽可能久地保留在数据块缓冲区缓存中。如果表较小且相对静止，则该选项很有用。NOCACHE(默认)使表块恢复为正常的最近最少使用(LRU)列表行为。

PARALLEL 以及 DEGREE 和 INSTANCES 指定表的并行特性(用于使用 Real Application Clusters 选项的数据库)。DEGREE 指定所用的查询服务器的个数。INSTANCES 指定为了进行并行查询处理，怎样将表在 Real Application Clusters 实例间分开。整数 n 指定表在指定数目的可用实例间分开。

可以使用 ALTER TABLE 进行 ADD、DROP、EXCHANGE、MODIFY、MOVE、SPLIT 或 TRUNCATE 分区。详细内容请参阅第 17 章和 CREATE TABLE。

LOB 存储子句指定本身不存储在表中的 LOB 数据(外部行)所用的存储区域。

MOVE ONLINE 选项允许在联机表重组期间为 DML 操作访问表。MOVE 和 MOVE ONLINE 选项只对非分区索引组织表可用。在 MOVE ONLINE 操作过程中，不支持对表的并行 DML 操作。

可以使用 COMPRESS 关键字指示 Oracle 压缩数据段，以便减少用直接路径 INSERT 加载的表对磁盘和内存的使用。

在 Oracle Database 11g 中，ALTER TABLE 在以下几方面得到增强：

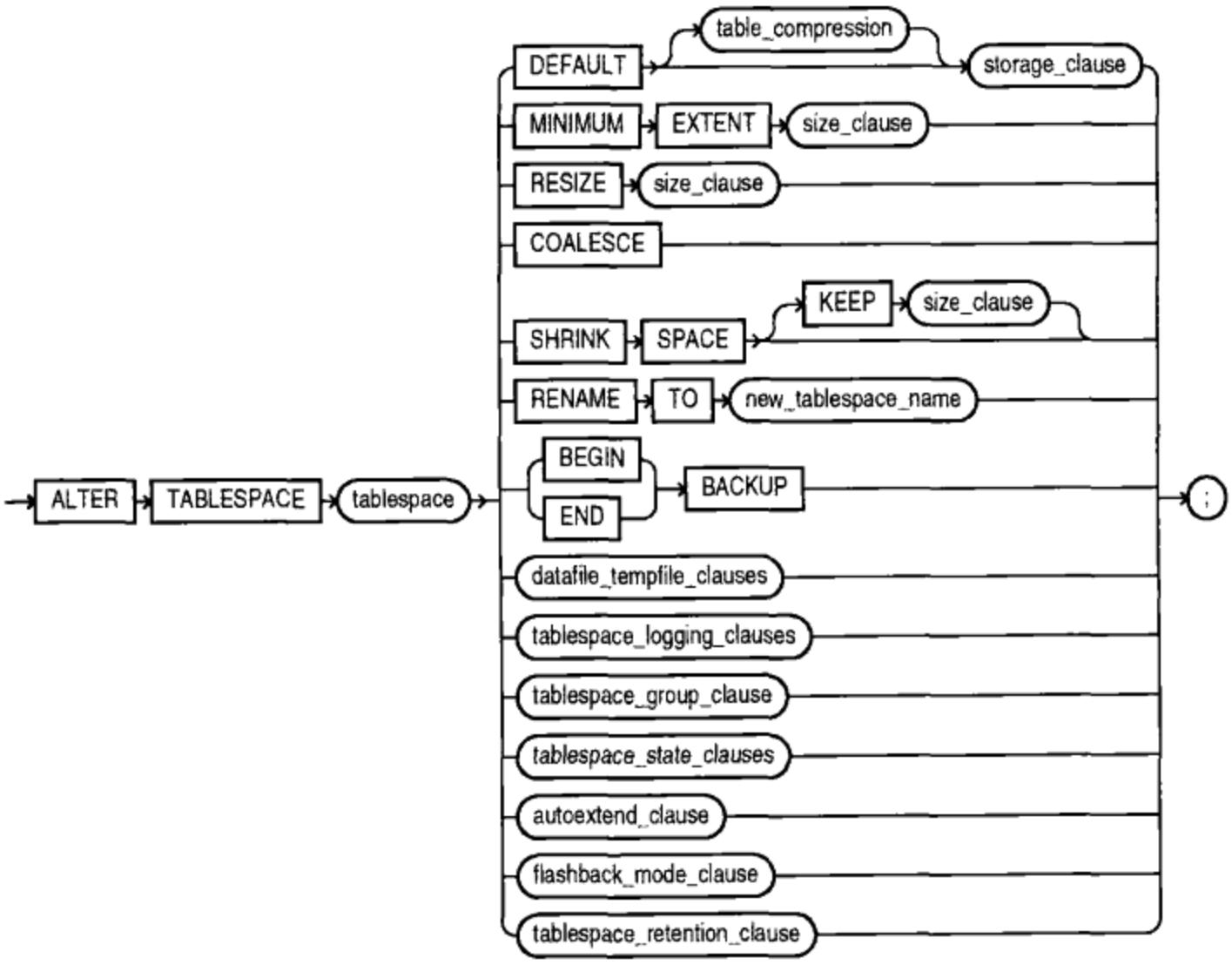
- add_column_clause 的性能得到提高。
- 由于 READ ONLY | READ WRITE 子句允许将表置于只读模式，因此阻止了 DDL 或 DML 改变表，直到将表恢复为读写模式。
- add_table_partition 子句的语法得到扩展，支持添加系统分区。
- flashback_archive_clause 允许启用或禁用表的历史跟踪。
- add_column_clause 支持向表添加虚拟列。
- 可以修改 XMLType 表，以添加或删除多个 XMLSchema 中的一个。
- alter_interval_partitioning 子句允许将范围-分区的表转换为间隔-分区的表。
- dependent_tables_clause 允许将表上的各种分区维护操作级联到引用-分区的子表上。

ALTER TABLESPACE

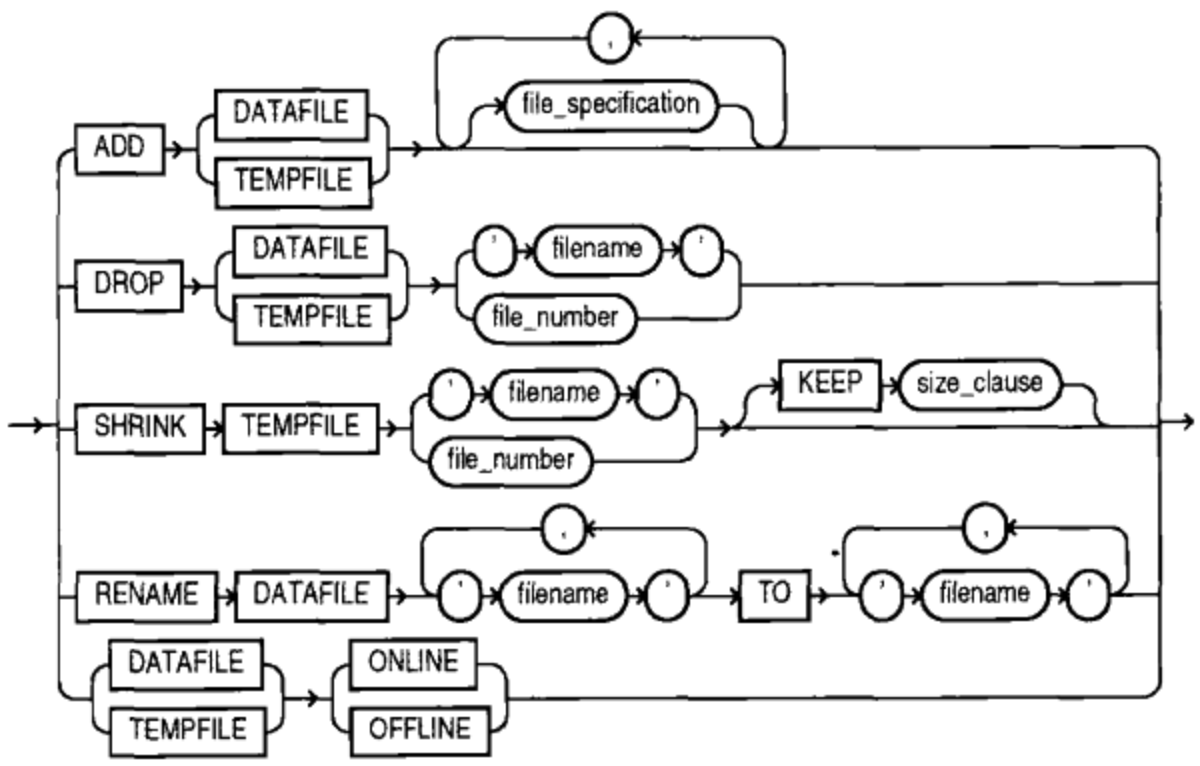
参阅：CREATE TABLESPACE、DROP TABLESPACE、STORAGE、第 22 章和第 51 章。

格式：

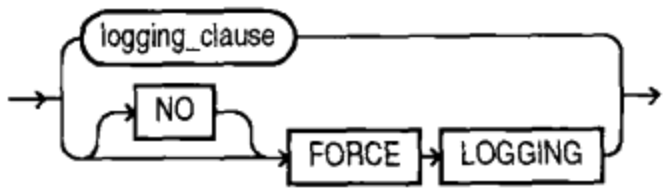
alter_tablespace::=



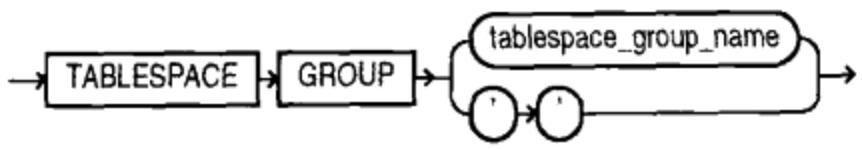
datafile_tempfile_clauses::=



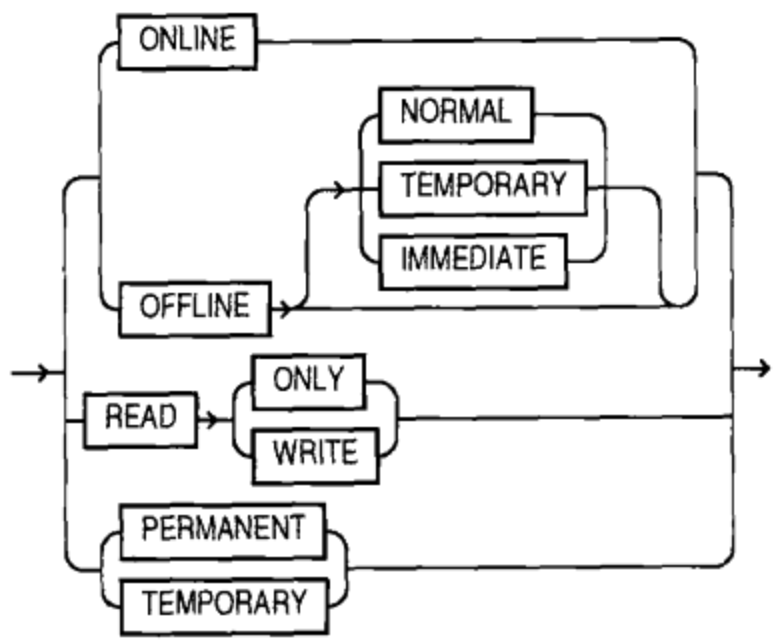
tablespace_logging_clauses::=



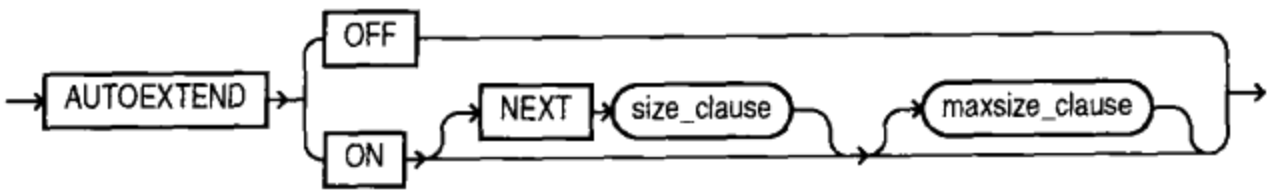
tablespace_group_clause::=



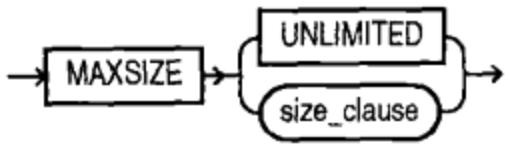
tablespace_state_clauses::=



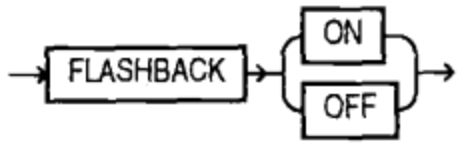
autoextend_clause::=



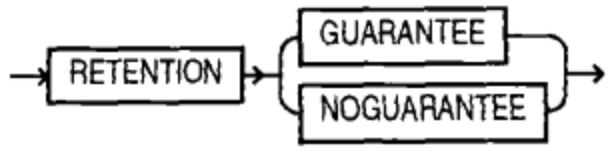
maxsize_clause::=



flashback_mode_clause::=



tablespace_retention_clause::=



描述：tablespace 是一个已有表空间的名称。ADD DATAFILE 根据 file_specification 的描述，给表空间添加一个文件或一系列文件，file_specification 定义了数据库的文件名和大小。

文件名的格式是操作系统特有的。SIZE 是为这个文件预留的字节数。如果后缀为 K，则预留的字节数为该值乘以 1 024；如果后缀为 M，则乘以 1 048 576。REUSE(不带 SIZE)表示销毁具有该文件名的任何文件中的内容，并把该名称分配给表空间使用。当文件不存在时，带 SIZE 的 REUSE 创建此文件。如果文件存在，则检查其大小。如果文件不存在时，只用 SIZE 就可以创建该文件；但如果存在，则返回一个错误。

根据需要，可以使用 AUTOEXTEND 子句，以 NEXT 大小递增，动态地将数据文件的大小调整为最大值 MAXSIZE(或 UNLIMITED)。

NOLOGGING 为表空间中某些特定类型的事务指定写入重做日志项的默认操作。可以在对象级别重写这个设置。RENAME DATAFILE 修改已有表空间文件的名称。在为表空间重新命名时，表空间应当脱机。请注意，实际上 RENAME 并不是对文件重命名，它只是把新名称与该表空间关联起来。事实上，重命名操作系统文件必须由操作系统自身完成。为了恰当地给文件重命名，首先要执行 ALTER TABLESPACE OFFLINE 操作，然后在操作系统中给文件重命名，接着用 ALTER TABLESPACE 来 RENAME 它们，最后执行 ALTER TABLESPACE ONLINE。

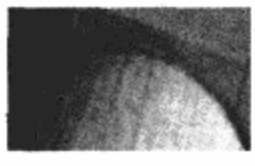
DEFAULT STORAGE 为表空间中将来创建的对象定义默认存储器，除非这些默认值被重写，如被 CREATE TABLE 重写。ONLINE 使表空间回到联机状态。OFFLINE 使表空间脱机，可以不等待用户退出(IMMIDIATE)就脱机，也可以在用户全部停止使用表空间之后(NORMAL)才脱机。

MINIMUM EXTENT 设置在表空间中所创建区的最小存储空间。COALESCE 将表空间中相邻的可用区组合为几个较大的可用区。

READ ONLY 表空间永不会被 Oracle 更新，它可存放在只读介质上。只读表空间不需要重复备份。创建的所有表空间都可读写。为了将一个只读表空间修改为读写状态，可使用 READ WRITE 子句。

BEGIN BACKUP 可在任何时候执行。它将表空间置于某种状态，以便保证用户访问表空间时能进行联机备份。END BACKUP 指出系统备份结束。如果表空间联机，则任何系统备份还必须备份归档重做日志。如果表空间脱机，则不需要备份归档重做日志。

TEMPORARY 表空间不能包含任何永久对象(如表或索引)，它们只能包含 Oracle 在处理分类操作或创建索引时使用的临时段。PERMANENT 表空间可保存任何类型的 Oracle 段。



注意：

对于大文件表空间，可以对只作用在小文件表空间的数据文件级别的一些命令(如 RESIZE)使用 ALTER TABLESPACE。

可以重命名表空间(通过 RENAME 子句)。还可以保证保存未过期的撤消数据，甚至以正在进行的需要使用撤消空间的事务为代价(通过 RETENTION GUARANTEE)。

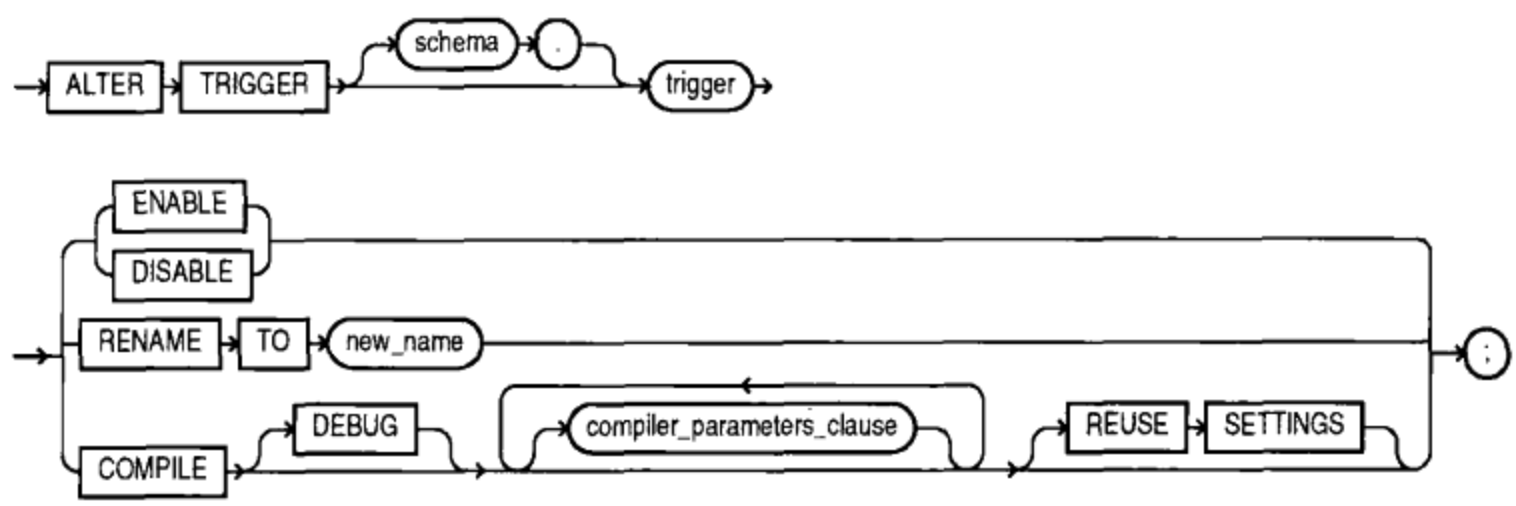
在 Oracle Database 11g 中，可以缩小临时表空间和单个临时文件所使用的空间。

ALTER TRIGGER

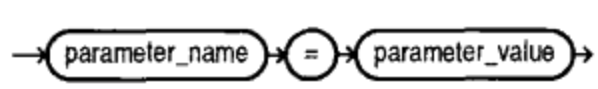
参阅：CREATE TRIGGER、DROP TRIGGER 和第 34 章。

格式：

alter_trigger::=



compiler_parameters_clause::=



描述：ALTER TRIGGER 启用、禁用、重命名或重新编译 PL/SQL 触发器。要使用 ALTER TRIGGER 命令，必须拥有相应的触发器或拥有 ALTER ANY TRIGGER 系统权限。

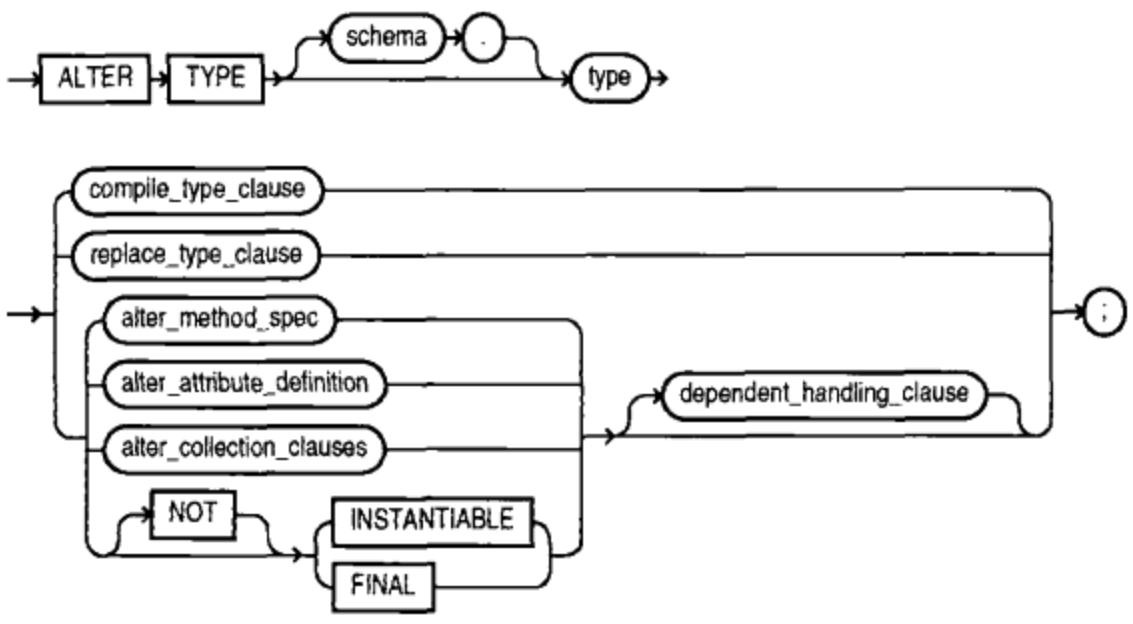
可以用 COMPILE 子句手动重新编译一个无效的触发器对象。因为触发器具有依赖性，所以如果触发器所依赖的某个对象被修改了，它们就会变得无效。DEBUG 子句允许在触发器重新编译过程中生成 PL/SQL 信息。

ALTER TYPE

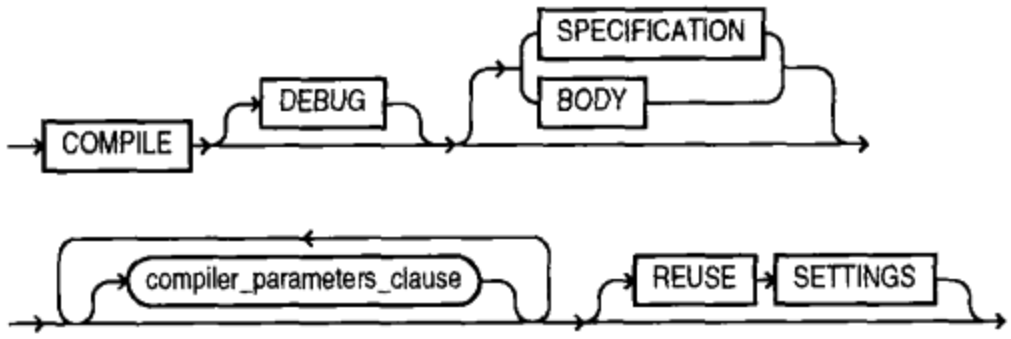
参阅：CREATE TYPE、CREATE TYPE BODY 和第 38 章。

格式：

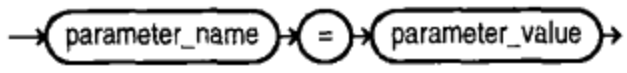
alter_type::=



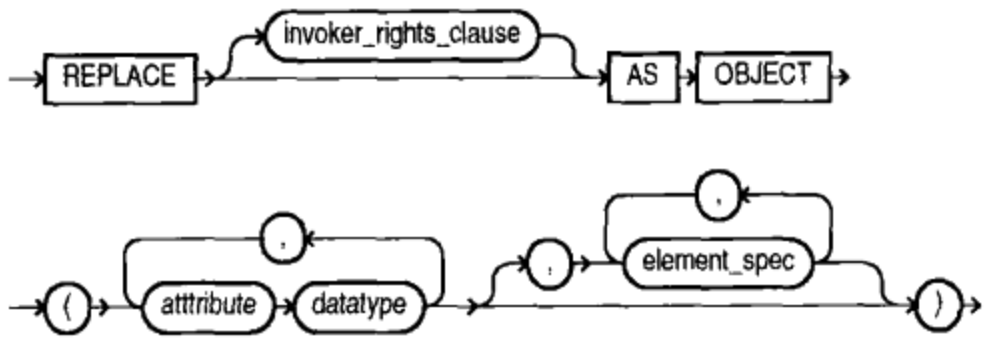
compile_type_clause::=



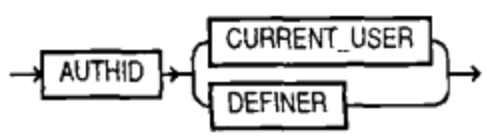
compiler_parameters_clause::=



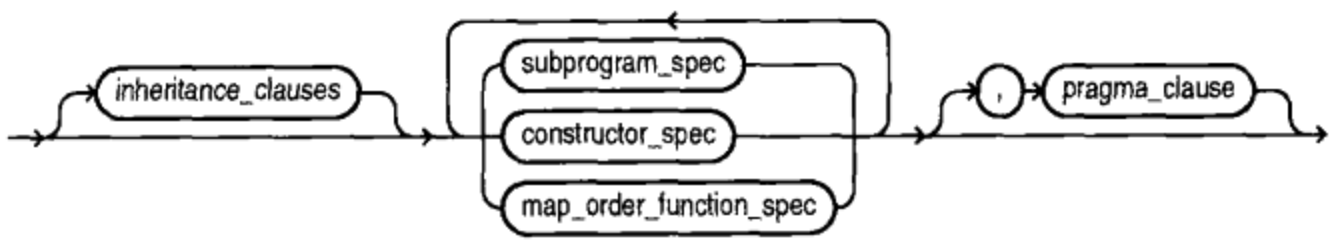
replace_type_clause::=



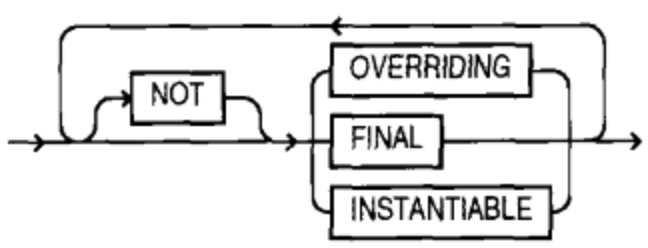
invoker_rights_clause::=



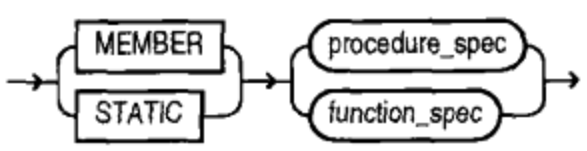
element_spec::=



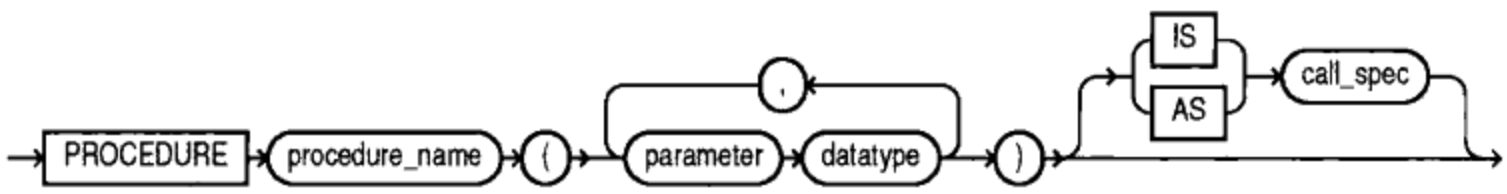
inheritance_clauses::=



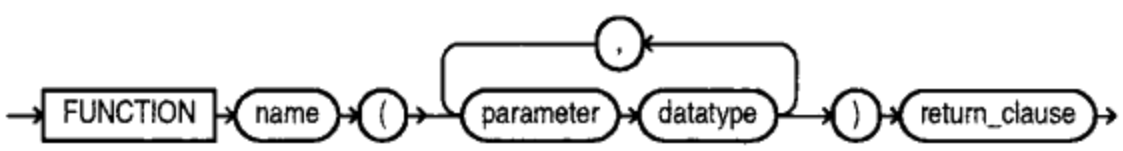
subprogram_spec::=



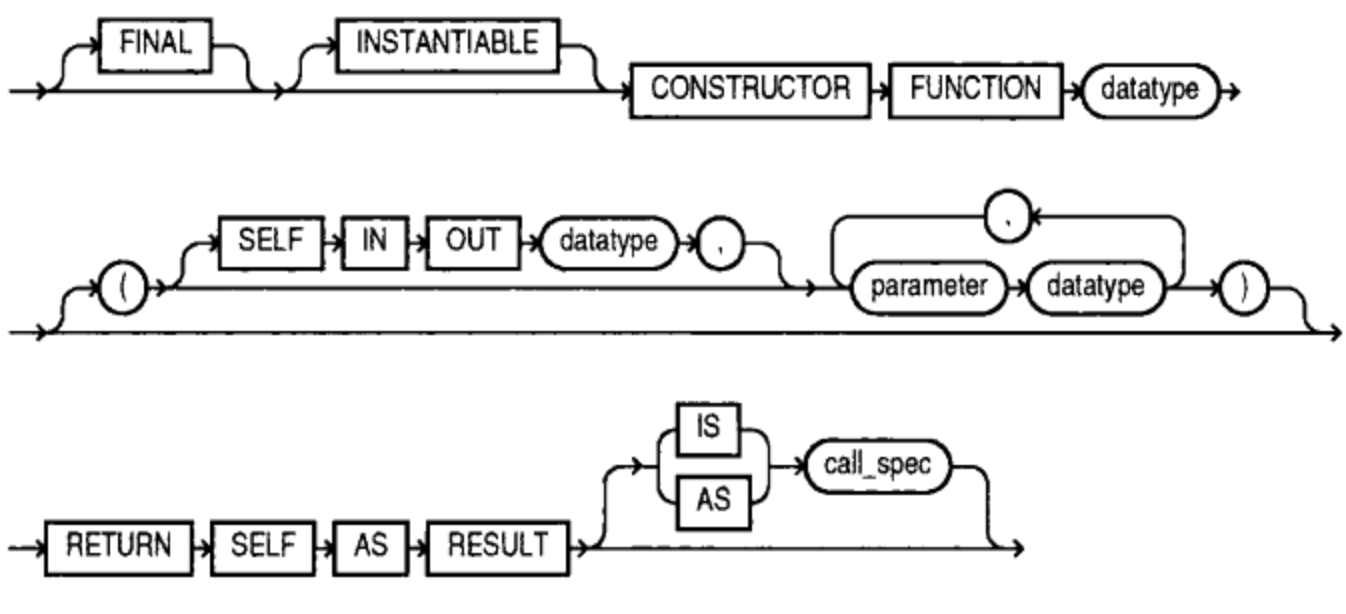
procedure_spec::=



function_spec::=

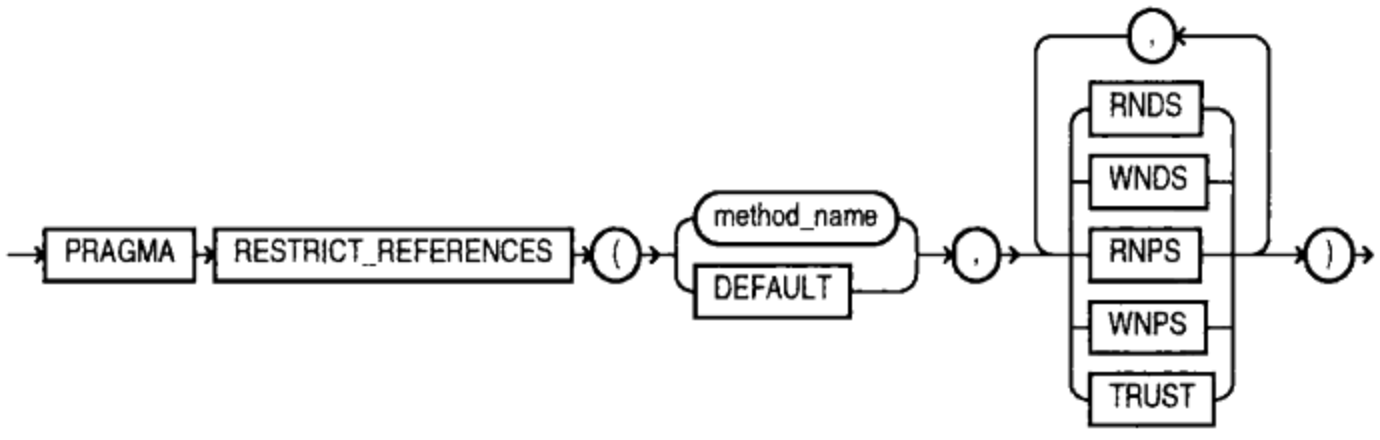


constructor_spec::=

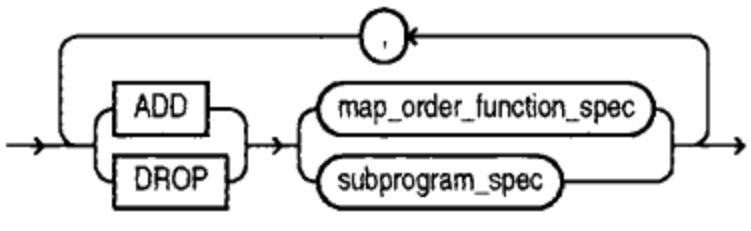


map_order_function_spec::=

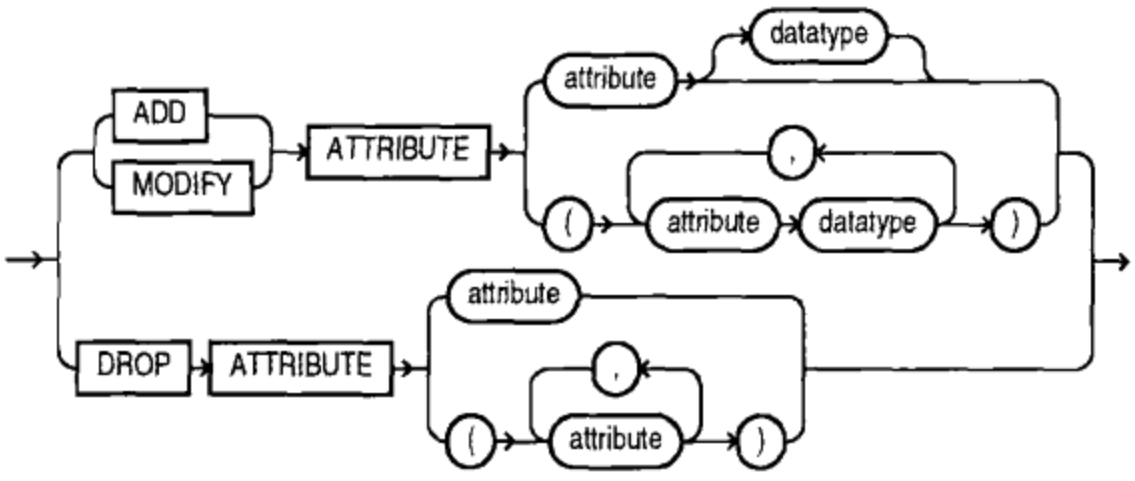
pragma_clause::=



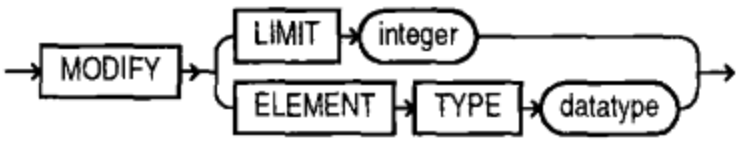
alter_method_spec::=



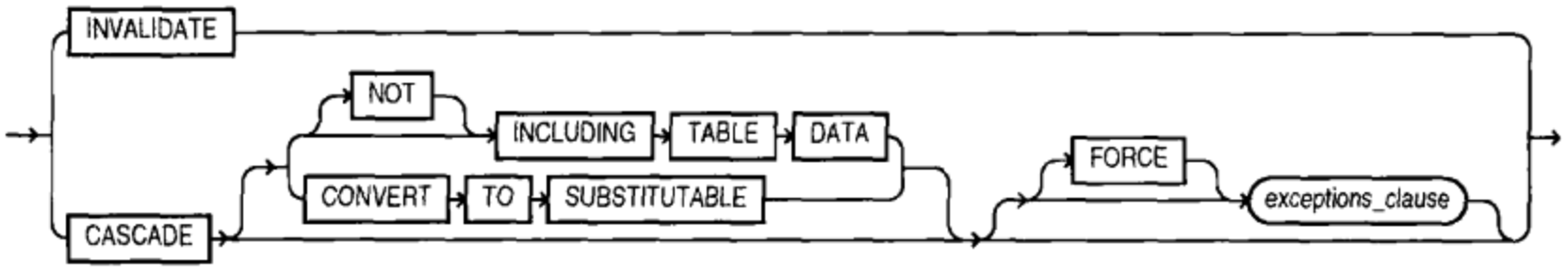
alter_attribute_definition::=



alter_collection_clauses::=



dependent_handling_clause::=



exceptions_clause::=

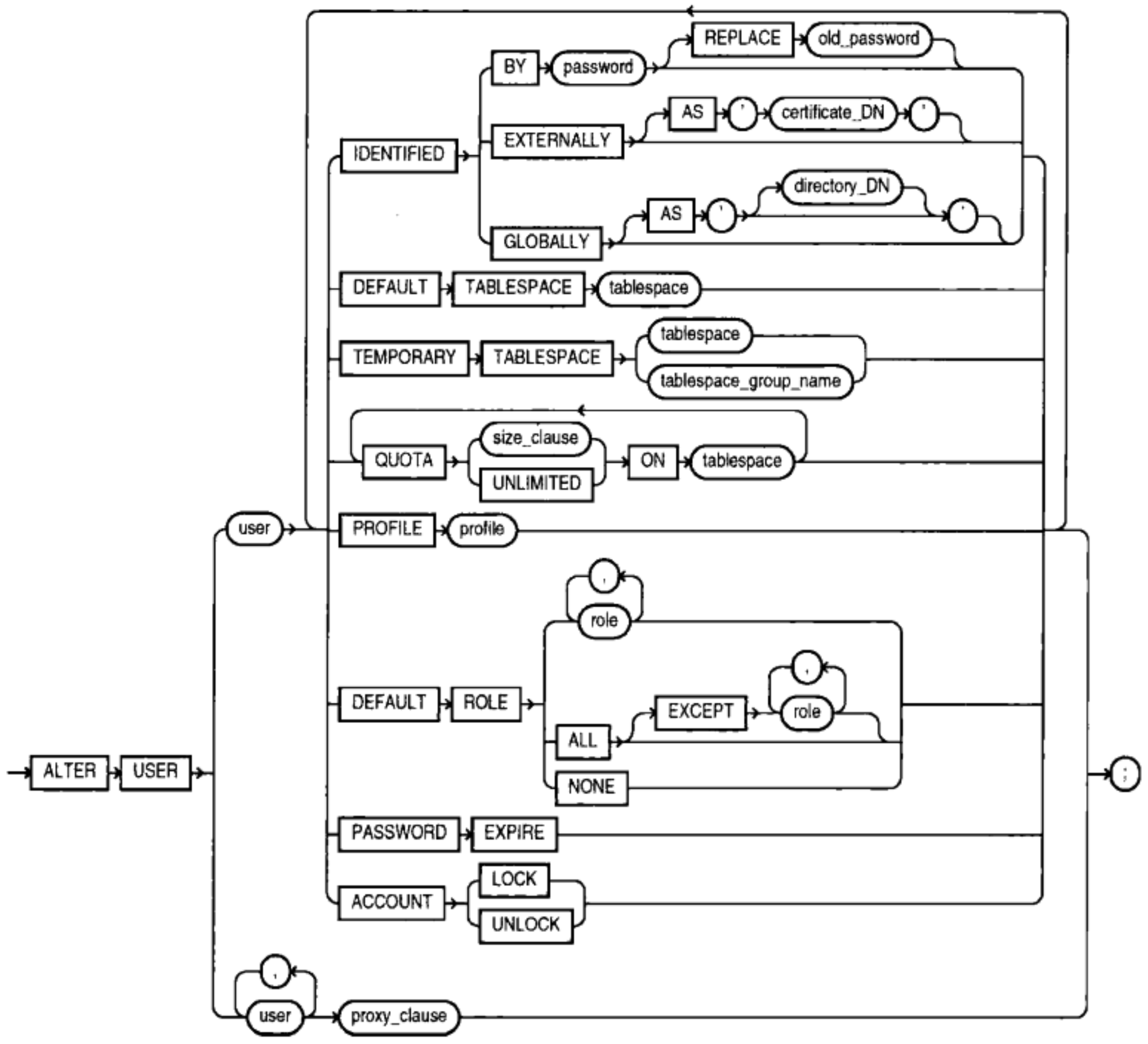
描述：ALTER TYPE 允许修改已有的抽象数据类型。在创建一个作用于抽象数据类型的方法时，可使用 CREATE TYPE BODY 命令(参阅本附录的条目)。类型体中定义的每个方法都必须首先在类型中列出。

ALTER USER

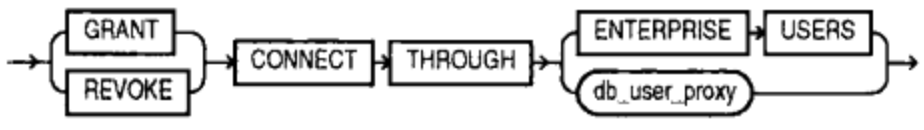
参阅：CREATE TABLESPACE、GRANT、CREATE PROFILE、CREATE USER 和第 19 章。

格式：

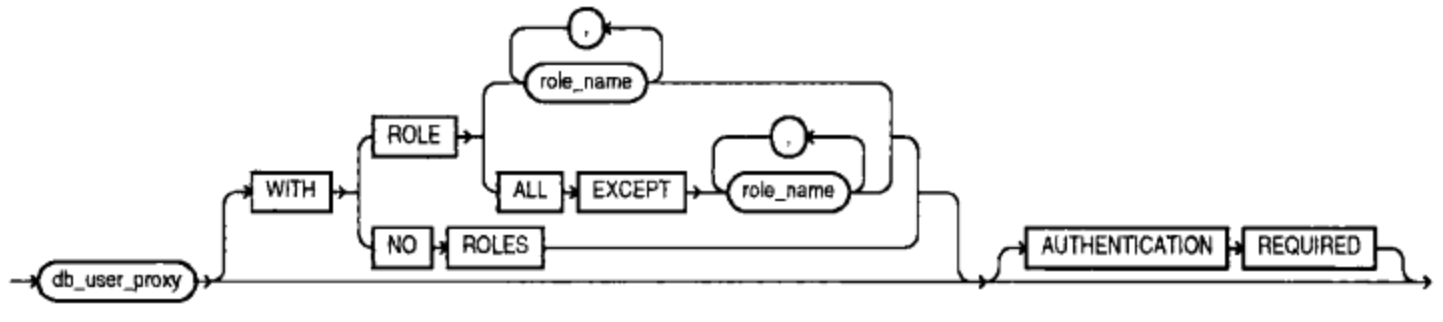
alter_user::=



proxy_clause::=



db_user_proxy::=



描述：可以使用 ALTER USER 修改用户的口令、DEFAULT 表空间(对于用户拥有的对象)或 TEMPORARY 表空间(对于用户所用的临时段)。如果不用 ALTER USER，则二者的默认值可以设置为授予(GRANT)用户资源和在数据库级别定义的默认临时表空间的第一个表空间(对象和临时段的表空间)的默认值。ALTER USER 还可以修改限额、资源配置文件或默认角色(参阅 CREATE USER)。可使用 PASSWORD EXPIRE 子句强制用户的口令到期。这样，在下一次登录时，用户就必须修改其口令。ACCOUNT UNLOCK 子句允许为连续登录失败次数超过最大值的账户解锁。关于口令控制的详细内容请参阅 CREATE PROFILE。要更改一个用户，必须拥有 ALTER USER 系统权限。

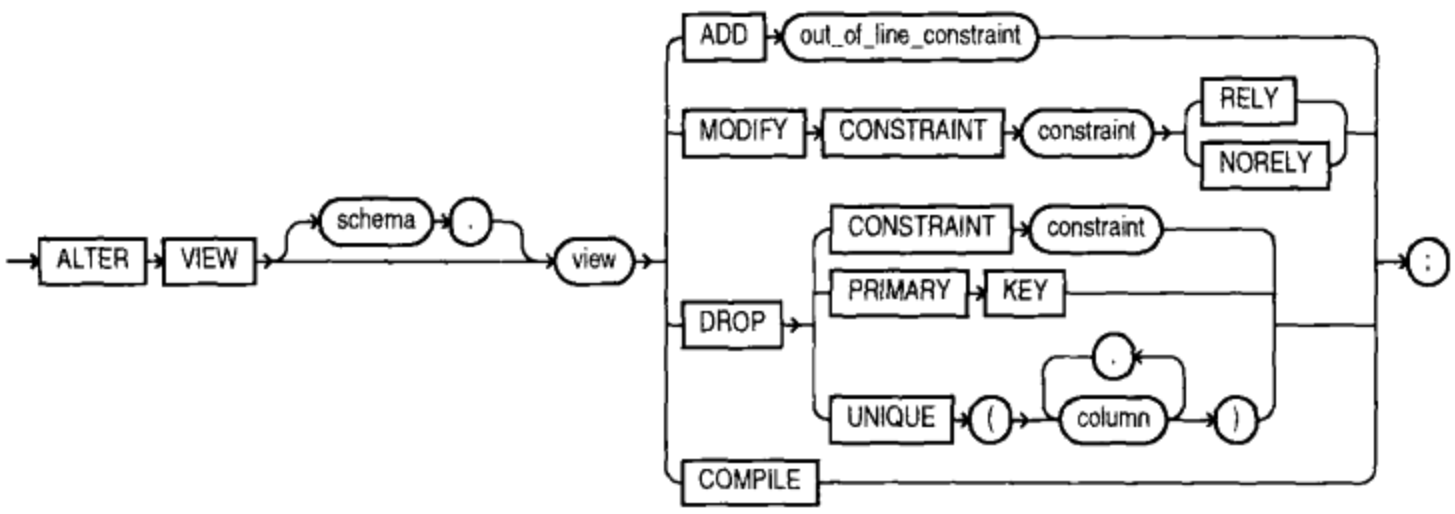
可以使用 PROXY 子句作为特定用户进行连接，并激活此用户的全部或部分角色，或者全部都不激活。通常 PROXY 用于涉及应用服务器的 3 层应用程序。

ALTER VIEW

参阅：CREATE VIEW、DROP VIEW 和第 17 章。

格式：

alter_view::=



描述：ALTER VIEW 重新编译一个视图，或更改其约束。要更改一个视图，必须拥有该视图，或者拥有 ALTER ANY TABLE 系统权限。

ANALYTIC FUNCTIONS

分析函数计算基于一组行(每组返回多行)的一个聚集值。许多分析函数在 AGGREGATE FUNCTIONS 条目下列出。当作为分析函数来使用时，它们只能在 ORDER BY 子句中使用。

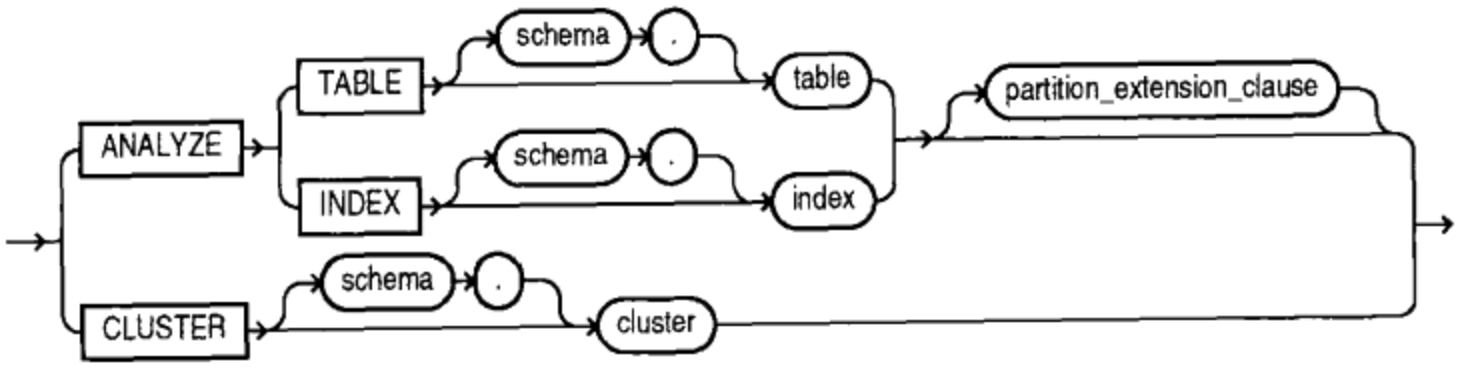
在下面的以字母顺序排列的分析函数(参阅语法描述的个别条目)中，后面带星号的函数允许使用开窗子句：AVG*、CORR*、COUNT、COVAR_POP*、COVAR_SAMP*、CUME_DIST、DENSE_RANK、FIRST、FIRST_VALUE*、LAG、LAST、LAST_VALUE*、LEAD、MAX*、MIN*、NTILE、PERCENT_RANK、PERCENTILE_CONT、PERCENTILE _DISC、RANK、RATIO_TO_REPORT、REGR_*、ROW_NUMBER、STDDEV*、STDDEV_ POP*、STDDEV_SAMP*、SUM*、VAR_POP*、VAR_SAMP*和 VARIANCE*。

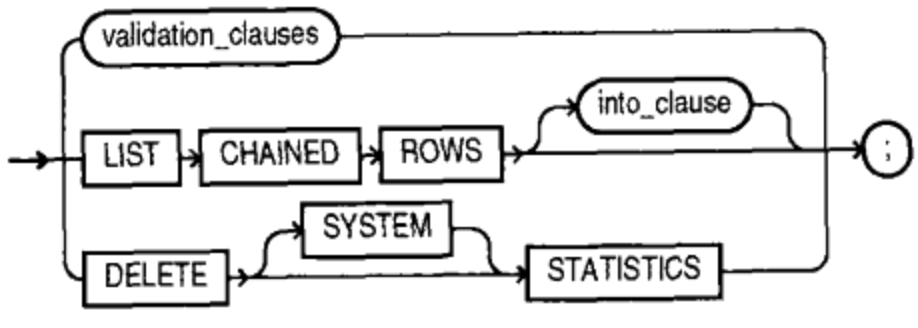
ANALYZE

参阅：第 46 章。

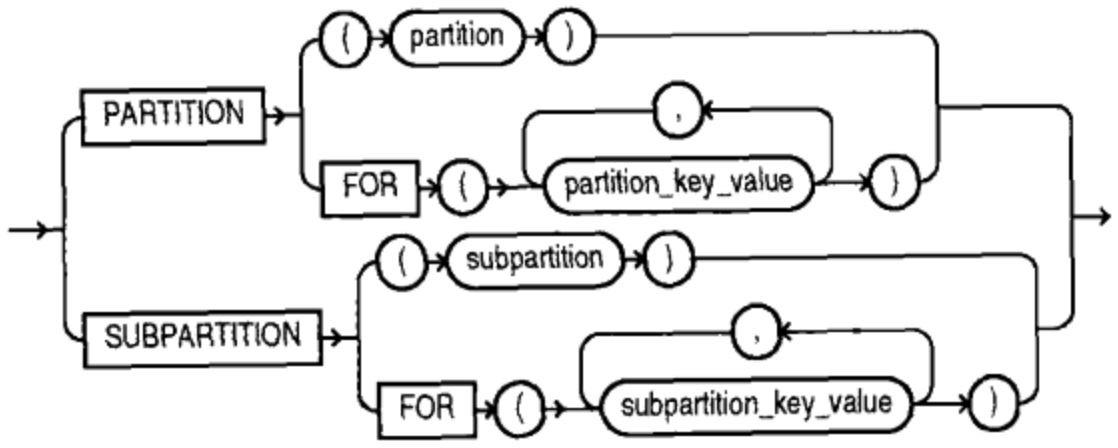
格式：

analyze::=

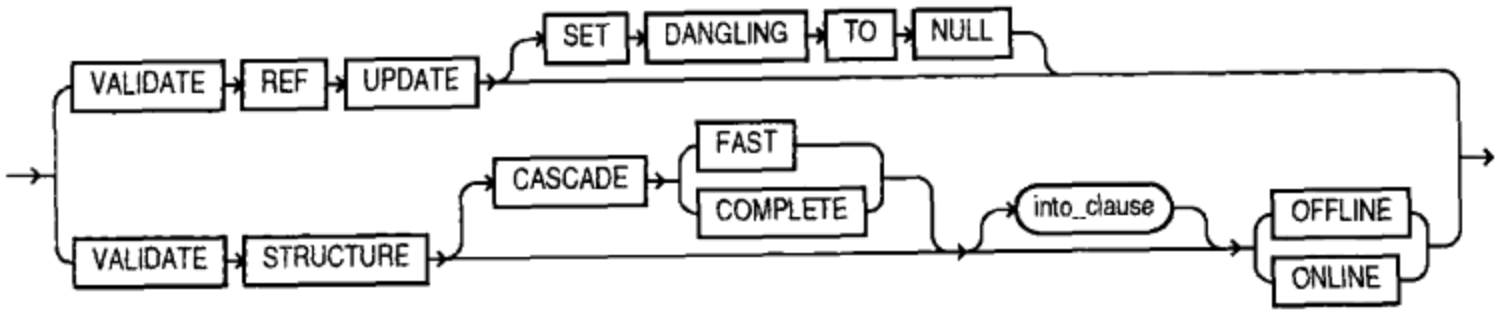




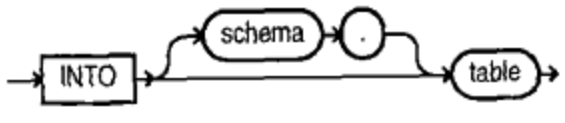
partition_extension_clause::=



validation_clauses::=



into_clause::=



描述：Oracle 建议使用 DBMS_STATS 程序包来替换 ANALYZE 命令。ANALYZE 允许用户为优化程序收集关于表、群集、分区或索引的相关统计信息，并且将它们存储在数据字典中。ANALYZE 还允许删除这些统计信息；它使对象结构生效，并且在本地表中识别包含列表的表或群集的迁移行和链接行。可以使用 VALIDATE STRUCTURE 子句测试可能有数据损坏的对象。要分析一个表，必须拥有该表，或者拥有 ANALYZE ANY 系统权限。

LIST CHAINED ROW 子句记录与用户在表中指定的行链接有关的信息。

AND

请参阅 LOGICAL OPERATORS、PRECEDENCE 和 CONTAINS。

ANY

参阅：ALL、BETWEEN、EXSISTS、IN、*LOGICAL OPERATORS* 和第 13 章。

格式：

```
operator ANY list
```

描述："=ANY" 等同于 IN。operator 可以为 "="、">"、">="、"<"、"<=" 或 "!="，而 list 可以为一系列字面量字串串(如 'Talbot'、'Jones' 或 'Hild')，或一系列字面量数值(如 2、43、76、32.06 或 444)，也可以是某个子查询的列，其中子查询的每行都是该列表的一个成员，如下所示：

```
LOCATION.City = ANY (select City from WEATHER)
```

list 变量也可以是主查询的 where 子句中的一系列的列，如下所示：

```
Prospect = ANY (Vendor, Client)
```

限定条件：list 不能是子查询中一系列的列，如下所示：

```
Prospect = ANY (select Vendor, Client from . . .)
```

许多人发现这个运算符和 ALL 运算符非常难记，因为在某些情况下它们的逻辑不太直观。结果，它通常可以替换某些形式的 EXISTS。

包含 ANY 的运算符与和一个列表的组合可用下面的解释来说明：

Page = ANY(4,2,7)	Page 在列表(4,2,7)中——限定为 2、4 和 7 都可以
Page > ANY(4,2,7)	Page 大于列表(4,2,7)中任何一项——即使限定为 3 也行，因为它大于 2
Page > = ANY(4,2,7)	Page 大于等于列表(4,2,7)中任何一项——即使限定为 2 也行，因为它等于 2
Page < ANY(4,2,7)	Page 小于列表(4,2,7)中任何一项——即使限定为 6 也行，因为它小于 7
Page < = ANY(4,2,7)	Page 小于等于列表(4,2,7)中任何一项——即使限定为 7 也行
Page != ANY(4,2,7)	Page 不等于列表(4,2,7)中任何一项——只要此列表中不止有一个值，限定为任何数都行。在只有一个值的情况下，"!=ANY" 等同于 "!="

APPEND(SQL*Plus)

参阅：CHANGE、DEL、EDIT、LIST 和第 6 章。

格式：

```
A[PPEND] text
```

描述：APPEND 是 SQL*Plus 命令行编辑器的一个功能。APPEND 将文本置于当前缓冲区中当前行的末尾，不插入空格。如果想在当前行的末尾和 text 之间添加一个空格，就应当在 APPEND 和 text 之间放置两个空格。如果想要在行的末尾追加一个分号，则同时放置两个(其中一个分号将作为命令结束符并被丢弃)。APPEND 在 iSQL*Plus 中不可用。

示例：

```
APPEND ;;
```

APPENDCHILDXML

格式：

```
APPENDCHILDXML
   ( XMLType_instance, XPath_string, value_expr [, namespace_string ] )
```

描述：APPENDCHILDXML 将用户提供的值追加到目标 XML 中，作为 XPath 表达式指定节点的子节点。

ARCHIVE LOG

参阅：ALTER DATABASE、RECOVER 和第 51 章。

格式：

```
ARCHIVE LOG {LIST|STOP}|{START|NEXT|ALL| integer} [TO destination]
```

描述：ARCHIVE LOG 启动或停止联机重做日志文件的自动归档，手动归档指定的重做日志文件，或显示有关重做日志文件的信息。START 启用自动归档，STOP 禁用自动归档。LIST 显示联机重做日志的归档状态。NEXT 手动归档已经填充但还未归档的下一组联机重做日志文件。ALL 手动归档所有已经填充但未归档的重做日志文件组。可以指定联机重做日志组号和归档文件的目标区域。

示例：

```
ARCHIVE LOG ALL
ARCHIVE LOG LIST
```

AS

参阅：ALIAS、TO_CHAR、第 7 章和第 10 章。

描述：AS 用来分隔列公式与列别名。

示例：下面用 AS 分隔列别名 PayDay 与它的列公式：

```
select NEXT_DAY(CycleDate,'FRIDAY') AS PayDay
from PAYDAY;
```

ASCII

参阅： CHARACTER FUNCTIONS、ASCIISTR 和 CHR。

格式：

```
ASCII(string)
```

描述： ASCII 是美国信息交换标准码的缩写。它是用数字数据表示可打印字符的一种约定。

ASCII 函数将返回串中第一个(最左边)字符的 ASCII 值。一个字符的 ASCII 值为 0~255 之间的一个整数。0~127 之间的那些值已经作了很好的定义。但对于大于 127 的值(“扩展的 ASCII 字符集”)，不同的国家、不同的应用程序和不同的计算机厂商的处理方式各不相同。例如，字母 A 的 ASCII 码为 65，B 为 66，C 为 67 等。小数点的 ASCII 码为 46。减号的 ASCII 码为 45。数字 0 的 ASCII 码对应 48，1 对应 49，2 对应 50 等。

示例：

```
select ASCII('.'), ASCII(.5),
       ASCII('E'), ASCII('EMILY')
  from DUAL;

ASCII('.')  ASCII(.5) ASCII('E') ASCII('EMILY')
----------  --------- ---------- --------------
        46         46         69             69
```

ASCIISTR

参阅： CHARACTER FUNCTIONS、ASCII 和 CHR。

格式：

```
ASCIISTR(string)
```

描述： ASCII 函数接受一个串并返回其在数据库字符集中相应的 ASCII 码。

示例：

```
select ASCIISTR('E'), ASCIISTR('EMILY')
  from DUAL;

A ASCII
- -----
E EMILY
```

ASIN

参阅： ACOS、ATAN、ATAN2、COS、COSH、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

格式：

```
ASIN(value)
```

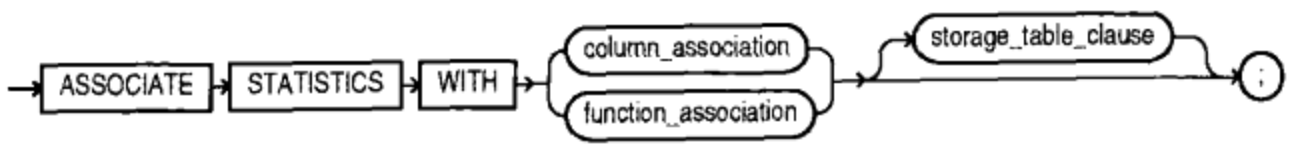
描述：ASIN 返回一个值的反正弦。输入值的范围为 - 1~1，输出值用弧度表示。

ASSOCIATE STATISTICS

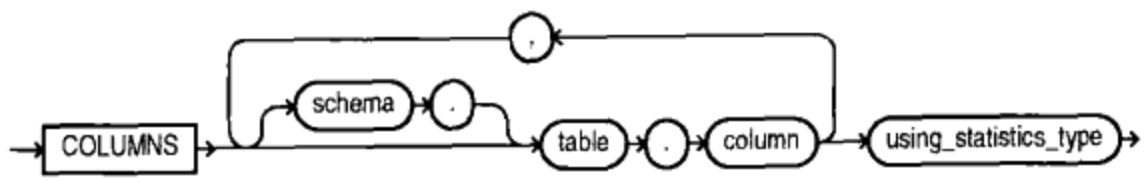
参阅：ANALYZE。

格式：

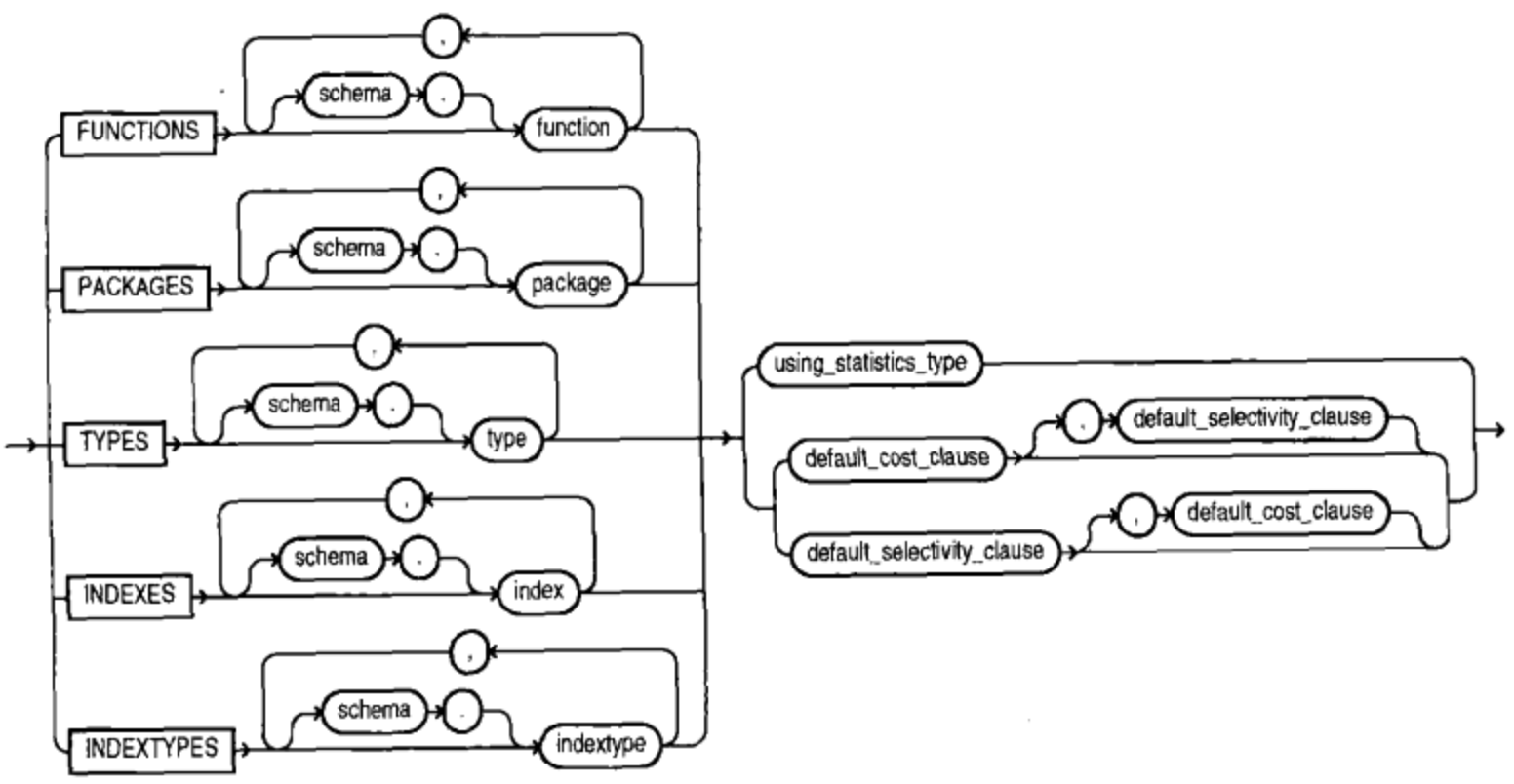
associate_statistics::=



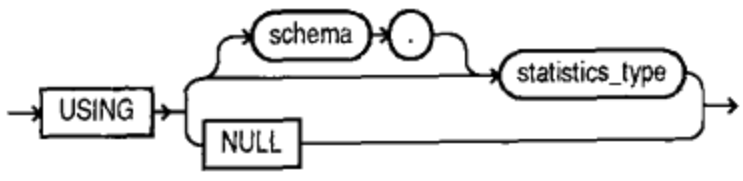
column_association::=



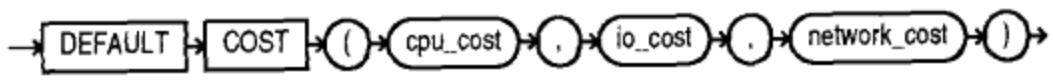
function_association::=



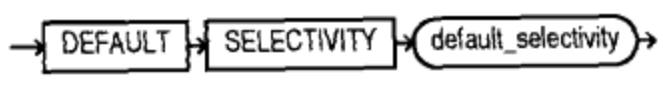
using_statistics_type::=



default_cost_clause::=



default_selectivity_clause::=



storage_table_clause::=

描述：ASSOCIATE STATISTICS 将一组统计函数与一列或多列、独立函数、程序包、类型或索引关联起来。用户必须拥有更改(ALTER)正被修改的对象所要求的权限。

在分析了与统计函数关联的对象之后，可以在 USER_USTATS 中查看各个关联。

在 Oracle Database 11g 中，可以指定数据库应该管理在系统托管的域索引上收集的统计信息的存储。

ATAN

参阅：ACOS、ASIN、ATAN2、COS、COSH、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

格式：

```
ATAN(value)
```

描述：ATAN 返回一个值的反正切。输入值不受限制，输出值用弧度表示。

ATAN2

参阅：ACOS、ASIN、ATAN、COS、COSH、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

格式：

```
ATAN2(value1, value2)
```

描述：ATAN2 返回两个值的反正切。输入值不受限制，输出值用弧度表示。

ATTRIBUTE(SQL*Plus)

参阅：COLUMN 和第 38 章。

格式：

```
ATTRIBUTE [type_name. attribute_name [option ...]]
```

option 可以是以下某个选项：

```
ALI[AS] alias
CLE[AR]
FOR[MAT] format
LIKE { type_name.attribute_name| alias}
ON|OFF
```

描述：ATTRIBUTE 设置在 SQL*Plus 中用户定义的类型列属性的显示属性。

示例：

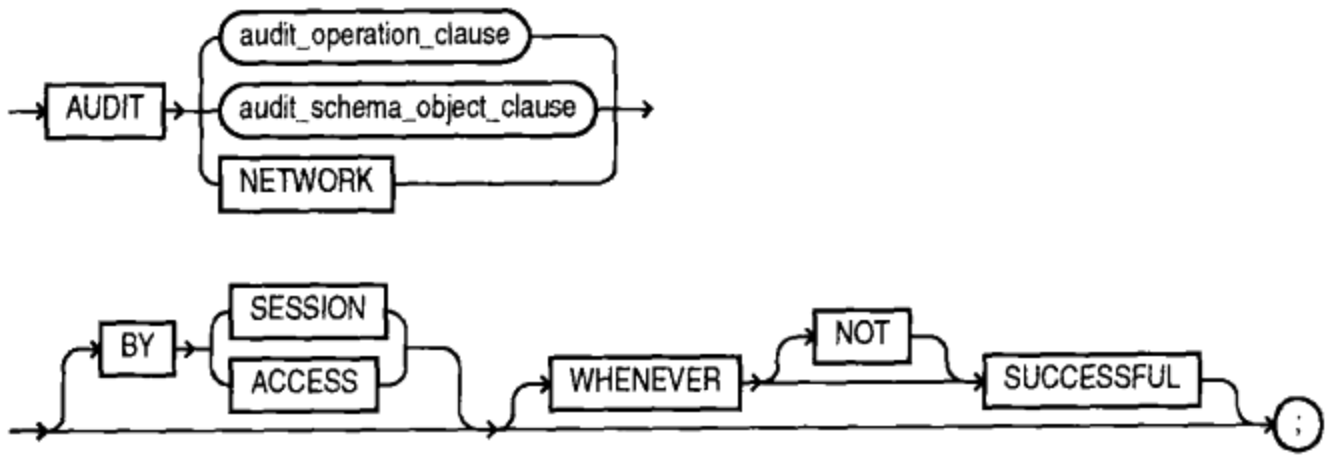
```
ATTRIBUTE ADDRESS_TY.Street FORMAT A50
```

AUDIT

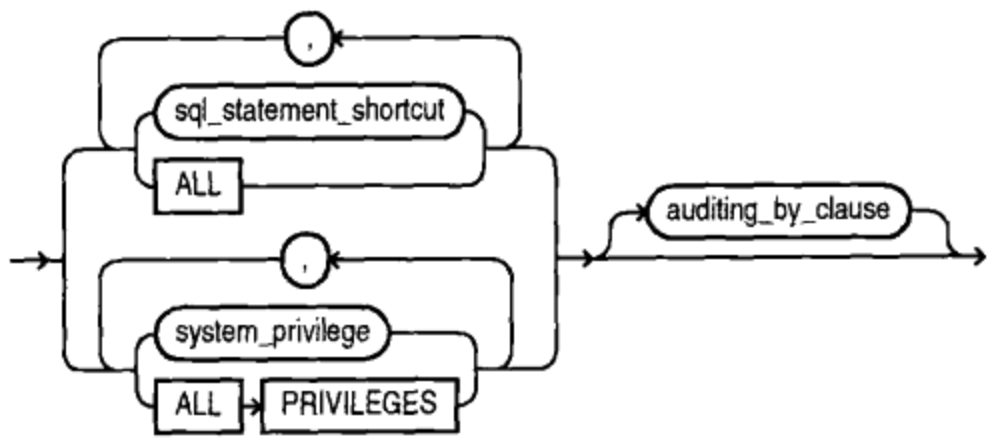
参阅： CREATE DATABASE LINK、NOAUDIT 和 PRIVILEGE。

格式：

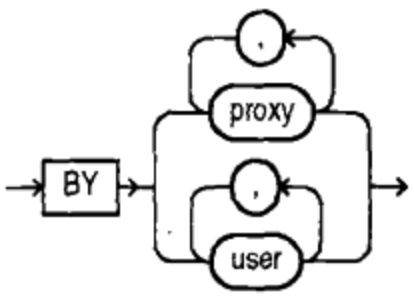
audit::=



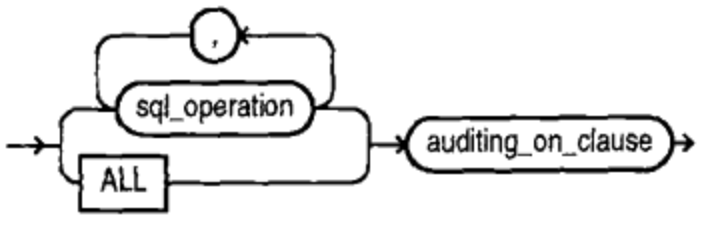
audit_operation_clause::=



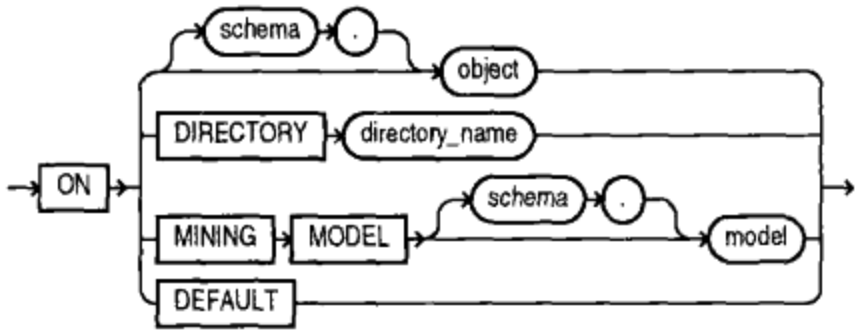
auditing_by_clause::=



audit_schema_object_clause::=



auditing_on_clause::=



描述：可以审计(AUDIT)用户执行的语句和访问指定对象的语句。

statement option 包含的内容和含义如表 A-3 所示。

表 A-3 statement option 包含的内容

选 项	审 计
ALTER SYSTEM	更改系统
CLUSTER	创建、更改、删除或截断群集
CONTEXT	创建或删除上下文
DATABASE LINK	创建或删除数据库链接
DIMENSION	创建、更改或删除维度
DIRECTORY	创建或删除目录
INDEX	创建、更改、分析或删除索引
MATERIALIZED VIEW	创建、更改或删除物化视图
NOT EXISTS	SQL 语句由于对象不存在而失败
PROCEDURE	创建或删除过程、函数、库或程序包；创建程序包体
PROFILE	创建、更改或删除配置文件
PUBLIC DATABASE LINK	创建或删除公有的数据库链接
PUBLIC SYNONYM	创建或删除公有的同义词
ROLE	创建、更改、删除或设置角色
ROLLBACK SEGMENT	创建、更改或删除回滚段
SEQUENCE	创建或删除序列
SESSION	登录尝试
SYNONYM	创建或删除同义词
SYSTEM AUDIT	审计或不审计 SQL 语句
SYSTEM GRANT	审计或取消系统权限和角色
TABLE	创建、删除或截断表
TABLESPACE	创建、更改或删除表空间
TRIGGER	创建、更改或删除触发器，用 ENABLE | DISABLE ALL TRIGGERS 更改表
TYPE	创建、更改或删除类型或类型体
USER	创建、更改或删除用户
VIEW	创建或删除视图

以下是其他可用的语句级选项，如表 A-4 所示。

表 A-4　其他可用的语句级选项

选　项	审计的命令
ALTER SEQUENCE	ALTER SEQUENCE
ALTER TABLE	ALTER TABLE
COMMENT TABLE	COMMENT ON TABLE table(表)、view(视图)、materialized view(物化视图)，COMMENT ON COLUMN table.column(表的列)、view.column (视图的列)、materialized view.column(物化视图的列)
DELETE TABLE	DELETE FROM table(表)、view(视图)
EXECUTE PROCEDURE	CALL 执行任何过程或函数，或访问程序包中的任何变量、库或游标
GRANT DIRECTORY	GRANT 权限 ON 目录。 REVOKE 权限 ON 目录
GRANT PROCEDURE	GRANT 权限 ON 过程、函数和程序包 REVOKE 权限 ON 过程、函数和程序包
GRANT SEQUENCE	GRANT 权限 ON 序列 REVOKE 权限 ON 序列
GRANT TABLE	GRANT 权限 ON 表、视图和物化视图 REVOKE 权限 ON 表、视图和物化视图
GRANT TYPE	GRANT 权限 ON TYPE REVOKE 权限 ON TYPE
INSERT TABLE	INSERT INTO 表、视图
LOCK TABLE	LOCK TABLE 表、视图
SELECT SEQUENCE	任何包含序列.CURRVAL 或序列.NEXTVAL 的语句
SELECT TABLE	SELECT FROM 表、视图和物化视图
UPDATE TABLE	UPDATE 表、视图

BY user 仅审计由特定用户发出的 SQL 语句。默认情况下，审计所有用户。

WHENEVER [NOT] SUCCESSFUL 仅在试图访问审计表或系统工具成功(或不成功)时，向审计表中写入一行。省略这个可选的子句会导致：无论访问是否成功都写入一行。

审计信息写入一个名为 SYS.AUD$的表并通过数据字典视图访问。

示例：该示例由名为 CLERK_USER 的用户审计所有更新内容：

```
audit UPDATE TABLE by clerk_user;
```

AVG

参阅：COMPUTE、AGGREGATE FUNCTIONS 和第 9 章。

格式：

```
AVG ( [ DISTINCT | ALL ] expr ) [OVER ( analytic_clause )]
```

描述：AVG 是一组行的 expr 的平均值。DISTINCT 仅对非重复值求平均值。此函数忽略 NULL 行，这可能会影响结果。

BEGIN

参阅：BLOCK STRUCTURE、DECLARE CURSOR、END、EXCEPTION、SET 下的 TERMINATOR 和第 32 章。

格式：

```
[<<block label>>]
[DECLARE]
BEGIN
     ... block logic ...
END [block label];
```

描述：BEGIN 是一个 PL/SQL 块可执行部分的开始语句。在它后边可以是任何合法的 PL/SQL 逻辑和一个异常处理程序，并以 END 语句结束。在 BEGIN 和 END 之间至少要有一条可执行语句。详细内容请参阅 BLOCK STRUCTURE。

block label 是以单词 BEGIN 开始，以单词 END 结束的 PL/SQL 块的名称。单词 END 后跟一个结束符，通常是分号(例外的情况，请参阅 SET 下的 TERMINATOR)。block label 遵循普通的对象命名约定，并且必须用“<<”和“>>”括起来。这些符号告诉 PL/SQL 这是一个标签。BEGIN 前可选放一个称为 DECLARE(跟在 block label 后)的部分，也可以选择包含一个称为 EXCEPTION 的部分。

BETWEEN

请参阅 LOGICAL OPERATORS。

BFILE

BFILE 是一种用于存储在数据库外部的二进制 LOB 数据的数据类型(详细内容请参阅 DATATYPES)。Oracle 不维护外部存储的数据的读一致性和数据完整性。在数据库中，存储的 LOB 定位器值指向外部文件。在创建 BFILE 项之前，首先要创建目录。详细内容请参阅 CREATE DIRECTORY。

BFILENAME

参阅：BFILE、CREATE DIRECTORY 和第 40 章。

格式：

```
BFILENAME ( 'directory' , 'filename' )
```

描述：BFILENAME 返回与指定文件名和目录对象相关联的 BFILE 定位器。

BINARY_DOUBLE

请参阅 DATATYPES。

BINARY_FLOAT

请参阅 DATATYPES。

BIN_TO_NUM

参阅： NUMBER FUNCTIONS 和第 9 章。

格式：

```
BIN_TO_NUM ( expr [, expr]... )
```

描述： BIN_TO_NUM 将一个位向量转换成与其等价的数字。

示例：

```
select BIN_TO_NUM(1,0,0,1) from DUAL;
BIN_TO_NUM(1,0,0,1)
-------------------
9
```

BITAND

参阅： DECODE 和第 16 章。

格式：

```
BITAND ( argument1, argument2 )
```

描述： BITAND 对 argument1 和 argument2 按位执行 AND 操作，这两个参数必须定义为非负整数，并返回一个整数。此函数通常与 DECODE 函数一起使用。

BLOB

BLOB 是一个二进制大对象，存储在数据库内部。详细内容请参阅第 41 章。

BLOCK STRUCTURE

块结构是 PL/SQL 中块部分的结构。

参阅： BEGIN、DECLARE CURSOR、END、EXCEPTION、GOTO 和第 32 章。

描述： 通过使用 Oracle 的预编译程序，可以将 PL/SQL 块嵌入 SQL*Plus 或其他几种程序设计语言中。PL/SQL 程序块的结构如下：

```
[<<block label>>]
[DECLARE
.. declarations (CURSOR, VARIABLE and EXCEPTION)...]

BEGIN

... block logic ( executable code)...

[EXCEPTION

... exception handling logic ( for fatal errors)...]

END [block label];
```

正如方括号所示，DECLARE 部分和 EXCEPTION 部分都是可选的。可以选择用块名标记一个块，块名必须用“<<”和“>>”括起来。

虽然在块逻辑或异常处理逻辑部分中，块可以嵌套在其他块中，但块的各部分必须按照这种顺序排列。可以使用 GOTO 将 blocks 用于分支，或作为另一个块中引用变量的前缀(详细内容请参阅 DECLARE)。如果要在游标声明部分中引用一个变量，则它必须在声明游标之前进行声明。

示例：

```
<<calc_areas>>
DECLARE
   pi constant NUMBER(9,7) := 3.14159;
   area NUMBER(14,2);
   cursor rad_cursor is
          select * from RADIUS_VALS;
   rad_val rad_cursor%ROWTYPE;
BEGIN
   open rad_cursor;
     loop
       fetch rad_cursor into rad_val;
         exit when rad_cursor%NOTFOUND;
         area := pi*power(rad_val.radius,2);
         insert into AREAS values (rad_val.radius, area);
     end loop;
   close rad_cursor;
END calc_areas;
```

BREAK(SQL*Plus)

参阅：CLEAR、COMPUTE 和第 6 章。

格式：

```
BRE[AK] [ON report_element [action [action]]] ...
```

其中，report_element 是：

```
{column| expr|ROW|REPORT}
```

action 是：

```
 [SKI[P] n|[SKI[P]] PAGE][NODUP[LICATES]|DUP[LICATES]]
```

描述：当 SQL*Plus 检测到一个指定的修改(诸如某个页面结束或某个表达式的值被修改)时，就会发生中断。中断将导致 SQL*Plus 执行在 BREAK 命令中指定的动作(如 SKIP)，并输出来自 COMPUTE 命令的结果，如一列的平均值或总计值。一次只有一个 BREAK 命令有效。新的 BREAK 命令可以指定将会导致中断的修改和相关联的动作。

BREAK ON REPORT 在报表或查询结束时导致中断。

由于表达式值的修改随时都可能在 REPORT 中断之前发生，因此在单个 BREAK 语句中可能有几个 ON 表达式子句。但是，它们在 BREAK 命令中的顺序应当与在 select 语句的 order by 子句中相同。也就是说，出现在 BREAK 命令中的每个表达式也应当以相同的序列放在

select 的 order by 子句中，否则结果没有任何意义。此外，它们的顺序应当是从最大的分组到最小的分组(如 ON Corporation、ON Division、ON Project)。

ON ROW 导致每个选择的行中断。

ON PAGE 在每页的结束处引起中断，并且与表达式、ON ROW 或 ON REPORT 导致的中断无关。

在输出中断关联的 COMPUTE 的结果前，SKIP 跳过许多行，同时 PAGE 或 SKIP PAGE 跳到一个新的页面。

除了发生 BREAK 后的第一个值，NODUPLICATES 禁止输出每一行的 BREAK 中表达式或列的值。

BREAK 自身将显示当前的中断设置。

CLEAR BREAKS 删除任何已有的 BREAK。

BTITLE(SQL*Plus)

参阅：ACCEPT、DEFINE、PARAMETERS、REPFOOTER、REPHEADER、SET HEADSEP 和 TTITLE。

格式：

```
BTI[TLE] [printspec [text|variable]... | OFF | ON]
```

描述：BTITLE 在每页报表的底部放置 text(可能有多行)。OFF 和 ON 禁止或还原 text 的显示，而不修改其内容。BTITLE 自己显示当前的 btitle 选项和文本或变量。text 是希望给予此报表的页脚，variable 为一个用户定义的变量或系统维护的变量，包括 SQL.LNO(当前行号)、SQL.PNO(当前页码)、SQL.RELEASE(当前 Oracle 版本)、SQL.SQLCODE(当前错误代码)和 SQL.USER(用户名)。

printspec 选项的描述如下：

- COL n 直接跳到当前行(从左边界开始的)第 n 个位置。
- S[KIP] n 表示输出 n 个空行。如果不指定 n，则输出一个空行。如果 n 为 0，则不输出空行，且当前输出位置成为当前行的第一列(从页的最左边开始计数)。
- TAB n 表示向前跳 n 个位置(如果 n 为负值，则向后跳)。
- BOLD 表示用粗体输出输出数据。
- LE[FT]、CE[NTER]和 R[IGHT]表示在当前行上分别使数据左对齐、居中和右对齐。跟在这些命令之后的任何文本或变量都作为一个组被对齐，直到命令结束，或者遇到一个 LEFT、CENTER、RIGHT 或 COL。CENTER 和 RIGHT 使用 SET LINESIZE 命令设置的值确定在何处放置文本或变量。
- FORMAT string(串)指定后面的文本或变量格式的控制模型，并遵循与 COLUMN 命令中的 FORMAT 相同的语法，如 FORMAT A12 或 FORMAT $999 999.99。每次出现 FORMAT 时，它都会取代前一个 FORMAT 的作用。如果不指定 FORMAT 模型，则使用由 SET NUMFORMAT 设置的模型。如果没有设置 NUMFORMAT，就使用 SQL* Plus 的默认设置。

除非被加载的变量具有用 TO_CHAR 重新格式化的日期，否则将按照默认格式输出数

据值。

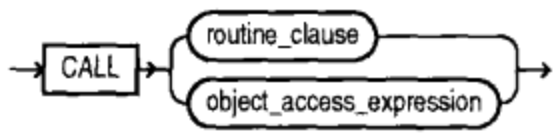
在单个 BTITLE 中可以使用任何数目的选项、文本和变量。它们的；输出按指定的顺序进行，且按其前面子句指定的位置放置和格式化。

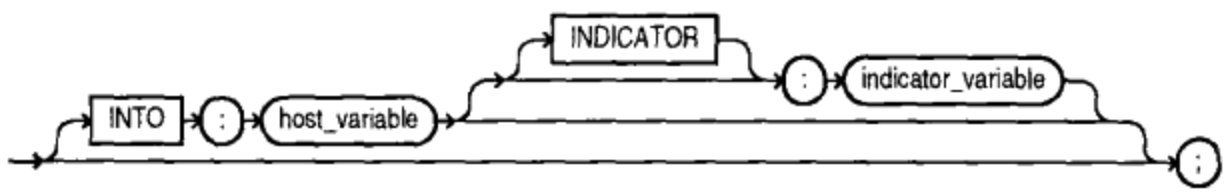
CALL

参阅： EXECUTE 和第 44 章。

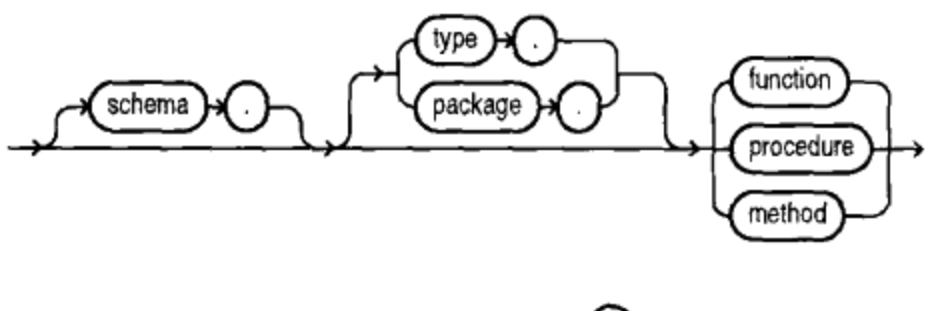
格式：

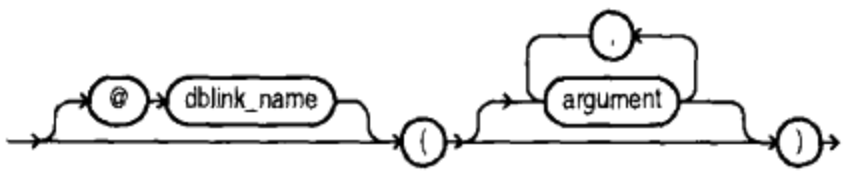
call::=



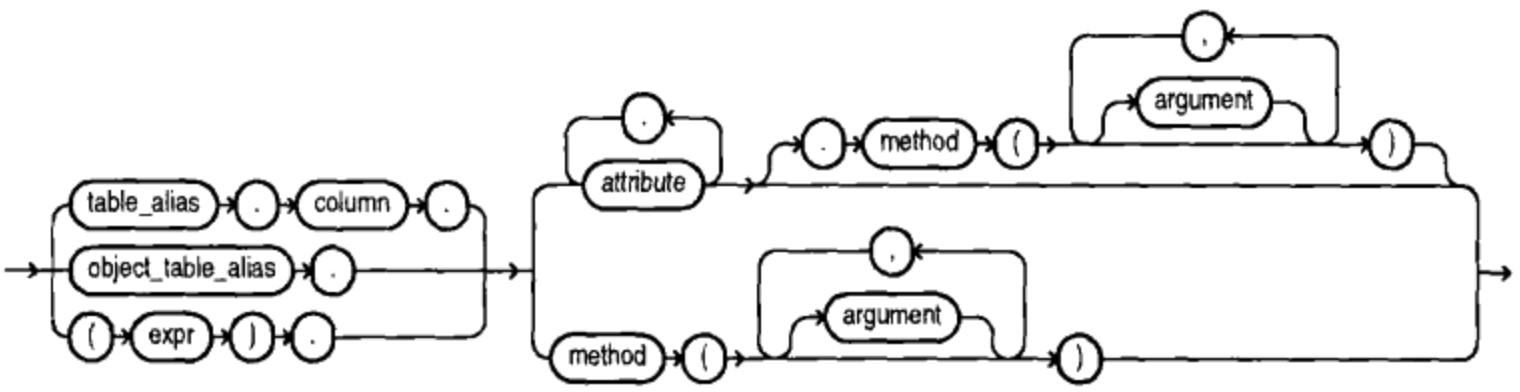


routine_clause::=





object_access_expression::=



描述： 可以从 SQL 内，用 CALL 执行一个存储过程或函数。必须拥有该过程或函数的 EXECUTE 权限，或者拥有该过程或函数所在程序包的 EXECUTE 权限。

在 Oracle Database 11g 中，如果例程接受任何参数，则可以在被调用例程的参数中使用位置、命名和混合标记。

CARDINALITY

参阅：COLLECTION FUNCTIONS。

格式：

```
CARDINALITY ( nested_table )
```

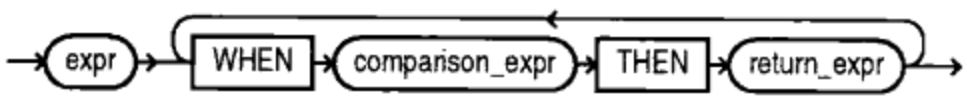
描述：CARDINALITY 返回嵌套表中元素的数目。如果嵌套表为空，或是一个空的集合，则 CARDINALITY 将返回 NULL。

CASE

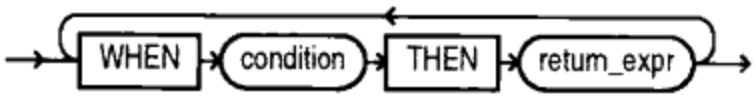
参阅：OTHER FUNCTIONS、DECODE、TRANSLATE 和第 16 章。

格式：

simple_case_expression::=



searched_case_expression::=



else_clause::=

描述：CASE 支持 SQL 语句中的 if-then-else 逻辑。可以使用 WHEN/THEN 子句告诉数据库为每个条件执行什么操作。可以使用 ELSE 子句处理 WHEN 子句以外的异常情况。

示例：

```
select distinct
       CASE CategoryName
         when 'ADULTFIC' then 'Adult Fiction'
         when 'ADULTNF' then 'Adult Nonfiction'
         when 'ADULTREF' then 'Adult Reference'
         when 'CHILDRENFIC' then 'Children Fiction'
         when 'CHILDRENNF' then 'Children Nonfiction'
         when 'CHILDRENPIC' then 'Children Picturebook'
         else CategoryName
       end
from BOOKSHELF;
```

CAST

参阅：第 39 章。

格式：

```
CAST ( { expr | ( subquery ) | MULTISET ( subquery ) } AS type_name )
```

描述：可以使用 CAST 把一种数据类型转换成另一种数据类型。在使用像嵌套表这样的收集器时，最常使用 CAST。

CATSEARCH

CATSEARCH 用于在 Oracle Text 中创建的 CTXCAT 文本索引中进行文本搜索。关于索引创建和文本搜索示例的内容，请参阅第 27 章。支持文本搜索的运算符如表 A-5 所示。

表 A-5 支持文本搜索的运算符

运 算 符	说 明
|	如果其中一个搜索项得分超过阈值，就返回一条记录
AND	如果两个搜索项得分同时都超过阈值，就返回一条记录。这是默认操作
-	返回包含“-”前面内容的行，而不包含“-”后面内容的行
NOT	同“-”
()	指定判定搜索条件的顺序
" "	用来括住短语

CEIL

参阅：FLOOR 和 NUMBER FUNCTIONS。

格式：

```
CEIL(value)
```

描述：CEIL 是大于或等于 value 的最小整数。

示例：

```
CEIL(1.3) = 2
CEIL(-2.3) = -2
```

CHANGE(SQL*Plus)

参阅：APPEND、DEL、EDIT、LIST 和第 6 章。

格式：

```
C[HANGE] / old text/ new text/
```

描述：CHANGE 是 SQL*Plus 命令行编辑器的一个功能。它将当前缓冲区中当前行的 old text 更改为 new text(在 LIST 中该行用一个“*”标记)。CHANGE 在 iSQL*Plus 中不可用。

CHANGE 在旧文本搜索中忽略大小写。3 个句点为一个通配符。如果在 old text 前面放置“...”，则第一次出现的 old text 以及 old text 之前的任何字符都被 new text 替换。如果在 old text 后面放置了“...”，则第一次出现的 old text 以及之后的任何字符都被 new text 替换。如果在 old text 中嵌入了“...”，则从 3 个点前的 old text 部分开始，直到 3 个点后面的 old text 的所有字符都被 new text 所替换。

CHANGE 和第一个/(斜杠)之间的空格可以省略。如果不需要插入句尾空格，则不需要最后的分隔符。可以使用除了/(斜杠)以外的分隔符。跟在 CHANGE 一词后面的任何字符(除空格外)都可以视为分隔符。

示例：如果当前缓冲区中的当前行为：

```
where CategoryName = 'ADULTFN'
```

则

```
C /ADULTFN/ADULTNF
```

将该行修改为：

```
where CategoryName = 'ADULTNF'
```

CHAR DATATYPE

请参阅 DATATYPES。

CHARACTER FUNCTIONS

参阅：CONVERSION FUNCTIONS、NUMBER FUNCTIONS、OTHER FUNCTIONS 和第 7 章。

描述：这是 Oracle 版本的 SQL 中所有当前字符函数按字母顺序排列的列表。每个字符函数都在本参考中其他地方在其相应名称下列出，并给出了适当的格式和用法。

函数名和用法

```
string || string
```

“||”(连接符号)连接两个串。|(分隔竖杠符号)符号称为分隔竖线，但在某些计算机上，它是一条实践。

```
ASCII(string)
```

ASCII 给出串的第一个字符的 ASCII 值。

```
CHR(integer)
```

CHR 给出其 ASCII 值等于给出的正整数的字符。

```
CONCAT(string1, string2)
```

CONCAT 连接两个串，等同于“||”(连接符号)。

```
INITCAP(string)
```

INITCAP 表示 INITial CAPital。它将一个单词或一系列单词的第一个字母修改为大写。

```
INSTR(string, set [, start [ , occurrence ] ])
```

INSTR 找到一组字符 IN a STRing(在一个串中)的开始位置。

```
LENGTH(string)
```

它给出串的 LENGTH(长度)。

```
LOWER(string)
```

LOWER 将串中每个字母转换为小写。

```
LPAD(string, length [,' set'])
```

LPAD 表示 Left PAD。它通过在串的左边添加一组指定的字符，使串达到指定的长度。

```
LTRIM(string [,' set'])
```

LTRIM 表示 Left TRIM。它去掉所有出现在串左边的属于一组字符的任何一个字符。

```
NLS_INITCAP(string[,'NLS_SORT= sort'])
```

NLS_INITCAP 表示 National Language Support INITial CAPital。此版本的 INITCAP 使用排序序列 sort 进行大小写转换。

```
NLS_LOWER(string[,'NLS_SORT= sort'])
```

NLS_LOWER 表示 National Language Support LOWERcase。此版本的 LOWER 使用排序序列 sort 进行大小写转换。

```
NLS_UPPER(string[,'NLS_SORT= sort'])
```

NLS_UPPER 表示 National Language Support UPPERcase。这个版本的 UPPER 使用排序序列 sort 进行大小写转换。

```
NLSSORT( string[,'NLS_SORT= sort'])
```

Oracle 使用 National Language Support SORT。它给出基于排序序列 sort 的给定字符串的排序序列值，如果省略，则给出此站点选择的 National Language Support 选项。

```
REGEXP_COUNT(source_string, pattern,
[,position, match_parameter])
```

REGEXP_COUNT 返回指定的正则表达式模式在源串中出现的次数。

```
REGEXP_INSTR(source_string, pattern
[, position, occurrence, return_option, match_parameter])
```

REGEXP_INSTR 返回一个指示搜索模式开始的整数。REGEXP_INSTR 通过允许搜索正则表达式模式，来扩展 INSTR 的功能。

```
REGEXP_REPLACE(source_string, pattern
[,replace_string, position, occurrence, match_parameter])
```

REGEXP_REPLACE 在每次用 replace_string(0 个或多个字符)替换 pattern 时，返回 source_string。REGEXP_REPLACE 通过允许搜索正则表达式模式，来扩展 REPLACE 的功能。

```
REGEXP_SUBSTR(source_string, pattern
```

```
[,position,occurrence, match_parameter])
```

REGEXP_SUBSTR 返回一个基于正则表达式搜索结果的 source_string 的子集。REGEXP_SUBSTR 通过允许搜索正则表达式模式来扩展 SUBSTR 的功能。

```
REPLACE(string, if [,then])
```

REPLACE 返回每次用 then 替换 if(0 个或多个字符)的 string。如果不指定 then 串，则所有出现的 if 串都将被删除。请参阅 TRANSLATE。

```
RPAD(string, length [ ,'set'])
```

RPAD 表示 Right PAD。它通过在右边添加一组指定的字符，来使串达到特定长度。

```
RTRIM(string [,'set'])
```

RTRIM 表示 Right TRIM。它去掉所有出现在串右边的属于一组字符的任何一个字符。

```
SOUNDEX(string)
```

SOUNDEX 把一个串转换为代码值。发音类似的名称很可能具有相同的代码值。可使用 SOUNDEX 来比较可能有微小拼写差异但发音仍然相同的名称。

```
SUBSTR(string, start [,count])
```

SUBSTRing 提取出一段字符串，它是从位置 start 开始并从 start 开始计数 count 个字符的一个子串。

```
TRANSLATE(string,if,then)
```

TRANSLATE 基于 then 串中字符与 if 串中字符的位置匹配，逐个字符地 TRANSLATE(转换)串。请参阅 REPLACE。

```
TREAT(expr AS [REF][user.] type)
```

TREAT 修改表达式的已声明类型。

```
TRIM
    ( [{ { LEADING | TRAILING | BOTH } [trim_character]}
       | trim_character
       } FROM ] trim_source )
```

TRIM 删除所有出现在一个串的右边、左边或两边的一组字符中的任何一个字符。

```
UPPER(string)
```

UPPER 将串中的每个字母转换为大写。

```
USERENV(option)
```

USERENV 返回有关 USER ENVironment(用户环境)的信息，通常用于审计跟踪。其选项包含“ENTRYID”、“SESSIONID”和“TERMINAL”。它仍支持 USERENV，但是已经由 SYS_CONTEXT 的 UserEnv 名称空间替换。

```
VSIZE(string)
```

VSIZE 指定在 Oracle 中 string 的存储空间大小。

CHARTOROWID

参阅： CONVERSION FUNCTIONS 和 ROWIDTOCHAR。

格式：

```
CHARTOROWID(string)
```

描述： CHARTOROWID 表示 CHARacter TO ROW Identifier(字符到行标识符)。它将一个字符串修改为一个内部的 Oracle 行标识符或 ROWID。

CHR

参阅： ASCII 和 CHARACTER FUNCTIONS。

格式：

```
CHR(integer)
```

描述： CHR 将返回 ASCII 值为 integer 的字符(integer 表示一个 0~255 之间的整数值，因为一个字符的 ASCII 值在整数 0~255 之间)。0~127 之间的那些值已经作了很好的定义。但对于大于 127 的值(称为扩展 ASCII 字符集)，不同的国家、应用程序和计算机厂商的处理方式各不相同。例如，字母 A 的 ASCII 码为 65，B 为 66，C 为 67 等。小数点的 ASCII 码为 46。减号的 ASCII 码为 45。数字 0 的 ASCII 码对应 48，1 对应 49，2 对应 50 等。

示例：

```
select CHR(77), CHR(46), CHR(56) from DUAL;

C C C
- - -
M . 8
```

CLEAR(SQL*Plus)

参阅： BREAK、COLUMN 和 COMPUTE。

格式：

```
CL[EAR] option
```

描述： CLEAR 清除选项。

BRE[AKS]清除由 BREAK 命令设置的中断。

BUFF[ER]清除当前缓冲区。

COL[UMNS]清除由 COLUMN 命令设置的选项。

COMP[UTES]清除由 COMPUTE 命令设置的选项。

SCR[EEN]清屏。

SQL 清除 SQL 缓冲区。

TIMI[NG]删除 TIMING 命令创建的所有计时区域。

示例：为清除计算，可以使用

```
clear computes
```

为了清除列定义，可以使用

```
clear columns
```

CLOB

CLOB 是一种支持字符类大对象的数据类型。请参阅第 40 章。

CLOSE

参阅：DECLARE CURSOR、FETCH、FOR、OPEN 和第 32 章。

格式：

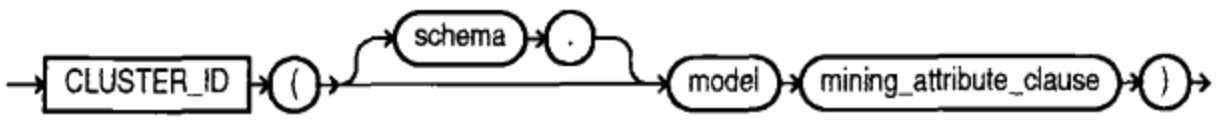
```
CLOSE cursor;
```

描述：CLOSE 关闭指定的游标，并释放游标所占用的资源，供 Oracle 在其他地方使用。cursor 必须是一个当前打开的游标名。

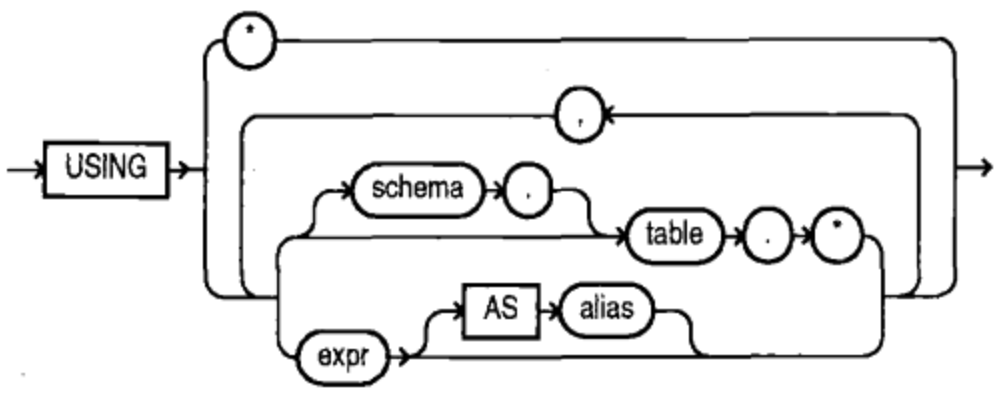
即使一个游标已被关闭，它的定义也还没有丢失。只要显式地声明了游标，就可以再次发出 OPEN cursor 命令。FOR 循环也将隐式地打开(OPEN)一个声明过的游标。请参阅 CURSOR FOR LOOP。

CLUSTER_ID

格式：



mining_attribute_clause::=

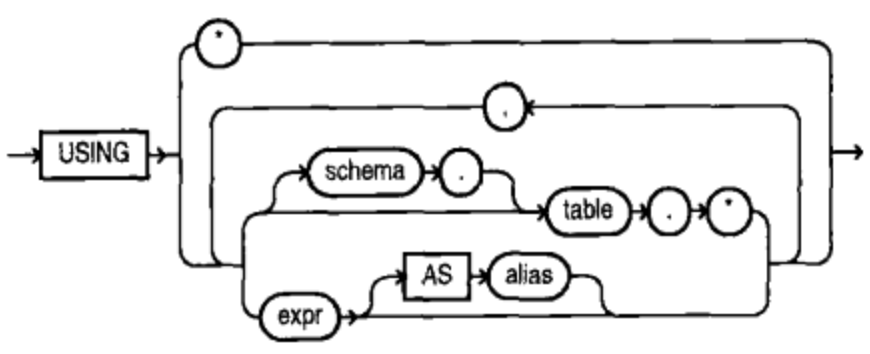


描述：此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的群集模型。它返回最可能具有 mining_attribute_clause 中指定的预测器的那个群集的标识符。返回值是 Oracle NUMBER。

CLUSTER_PROBABILITY

格式：

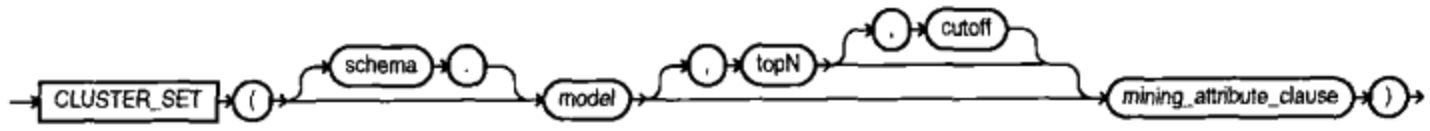
mining_attribute_clause::=



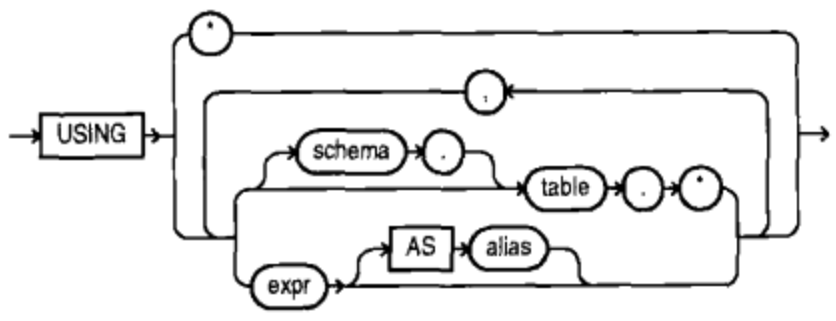
描述：此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的群集模型。它返回与指定模型相关联的群集中的输入行的成员的可信度。

CLUSTER_SET

格式：



mining_attribute_clause::=



描述：此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的群集模型。它返回对象的可变数组，对象包含指定行可能属于的所有群集。可变数组中的每个对象都是一对标量值，包含群集 ID 和群集可能性。对象字段被命名为 CLUSTER_ID 和 PROBABILITY，二者都是 Oracle NUMBER。

COALESCE

参阅：DECODE 和第 16 章。

格式：

```
COALESCE(value1, value2, ...)
```

描述：COALESCE 将返回在所提供的列表值中遇到的第一个非 NULL 值。

COLLECT

参阅：COLLECTION FUNCTIONS 和 CAST。

格式：

```
COLLECT(column)
```

描述：COLLECT 接受输入的 column 值，并在被选择的行外创建一个输入类型的嵌套表。要得到函数的结果，必须在 CAST 函数中使用它。

COLLECTION FUNCTIONS

收集函数作用于嵌套表和可变数组，详细内容请参阅第 39 章。可用的收集函数包括 CARDINALITY、COLLECT、POWERMULTISET、POWERMULTISET_BY_CARDINALITY 和 SET。

COLUMN

参阅：ALIAS 和第 6 章。

格式：

```
COL[UMN] {column | expression}
    [ ALI[AS] alias ]
    [ CLE[AR] | DEF[AULT] ]
    [ ENTMAP {ON|OFF}
    [ FOLD_A[FTER]
    [ FOLD_B[EFORE]
    [ FOR[MAT] format ]
    [ HEA[DING] text
       [ JUS[TIFY] {L[EFT]|C[ENTER]| R[IGHT]} ] ]
    [ LIKE {expression | alias} ]
    [ NEWL[INE] ]
    [ NEW_V[ALUE] variable ]
    [ NOPRI[NT]|PRI[NT] ]
    [ NUL[L] text ]
    [ OLD_V[ALUE] variable ]
    [ ON | OFF ]
    [ WRA[PPED]|WOR[D_WRAPPED]|TRU[NCATED] ]...
```

描述：COLUMN 控制列和列标题的格式化。各选项全都是累积的，可以同时在单独一行上输入，也可以随时在不同的行上输入。唯一的要求是单词 COLUMN 和列或表达式必须出现在每个单独的行上。如果其中的一个选项被重复，则最近指定的选效有效。COLUMN 本身将显示所有列当前的所有定义。只有一列或一个表达式的 COLUMN 将显示该列当前的定义。

column 或 expression 指在 select 中使用的列或表达式。如果使用表达式，则必须用与 select 语句中完全相同的方式输入表达式。如果这个表达式在 select 中是 Amount*Rate，那么在 COLUMN 命令中输入 Rate*Amount 将不起作用。如果在 select 语句中为列或者表达式指定一个别名，这里就必须使用该别名。

如果选择(select)不同表中名称相同的列(按顺序选择)，则该列名的 COLUMN 命令将会应用到两个表上。通过在 select 中给列赋予不同的别名(而不是用 COLUMN 命令的 ALIAS 子句)，并为每个列别名输入一个 COLUMN 命令，即可避免这种情况。

ALIAS 为列指定一个新名称，这个新名称可用来在 BREAK 和 COLUMN 命令中引用该列。

CLEAR 删除列定义。

DEFAULT 保留所定义的列和 ON，但删除任何其他选项。

ENTMAP 允许为 HTML 输出中选择的列打开或关闭实体映射。

FOLD_A[FTER]和 FOLD_B[EFORE]指示 Oracle 在输出时，将单行折成多行输出。可选择在列前或列后进行换行。

FORMAT 指定列的显示格式。此格式必须是一个字面量，如 A25 或 990.99。在不指定 FORMAT(格式)的情况下，列宽度与在表中定义的宽度相同。

较长列的宽度默认为 SET LONG 的值。常规的 CHAR 字段和 LONG 字段具有 FORMAT An 格式设置的宽度，其中 n 为一个整数，它是列后来的宽度。

数值列的宽度默认为 SET NUMWIDTH 的值，但是可以用 FORMAT 子句的宽度修改它，如 FORMAT 999,999.99。下面的选项可用于 SET NUMFORMAT 和 COLUMN FORMAT 命令，如表 A-6 所示。

表 A-6 可用于 SET NUMFORMAT 和 COLUMN FORMAT 命令的选项

格 式	结 果
,	显示逗号
.	显示句点
$99999	在每个数的前面放置一个美元符号
999990	显示末尾的 0，如果值为 0，则显示一个 0
099999	显示前导 0
9999990	9 和 0 的数目决定可显示的最大位数
999,999,999.99	逗号和小数点将放置在所显示的模式中
B99999	如果值为 0，则显示空格
C999	在所示位置上显示 ISO 货币符号
99D99	在指定位置上显示十进制数
9.999EEEE	用科学计数法显示(要求 4 个 E)
9G999	在所示位置上显示组分隔符
L999	显示本地货币符号
99999MI	如果为负数，则负号跟在该数的后面。默认时负号在左边
99999PR	负数显示在“<”和“>”之间
RN 或 rn	显示大写或小写的罗马数字，值在 1~3999 之间
S9999 或 9999S	对于正值，返回前导或尾端的+(加号)；对于负值，返回前导或尾端的-(减号)
TM	显示可能的十进制字符中最小数字的数据
U999	在指定位置上显示欧元或其他双重货币符号
999V99	数乘以 n 个 10，其中 n 为 V 右边的位数
X	显示指定位数的四舍五入整数值的十六进制值

HEADING 重新标记一列的标题。默认值为列名或表达式。如果文本有空格或标点符号，则必须用单引号括起来。文本中的 HEADSEP 字符(一般为“|”)使 SQL*Plus 另起一行。在定义列时，COLUMN 命令将记住当前的 HEADSEP 字符，除非重新定义列，否则继续将它用于该列，即使修改 HEADSEP 字符，也是如此。

JUSTIFY 使该列的标题对齐。默认时，数值列为 RIGHT(右)对齐，而其他列为 LEFT(左)对齐。

LIKE 为当前列复制以前定义的列的列定义，其表达式或标签可用在其他列定义中。只复制在当前列命令中未显式定义的其他列的功能。

NEWLINE 在输出列值前另起一行。

NEW_VALUE 命名一个变量来保存在 TTITLE 命令中所使用的列值。详细内容请参阅第 6 章。

NOPRINT 和 PRINT 打开或关闭列的显示。

若某一列有一个 NULL 值时，则 NULL 设置文本为显示。默认时，显示与定义的列宽度相同的空字符串。

OFF 或 ON 打开或关闭某列的所有这些选项，而不影响该列的内容。

OLD_VALUE 命名一个变量，以保存在 BTITLE 命令使用的列值。

WRAPPED、WORD_WRAPPED 和 TRUNCATED 控制 SQL*Plus 怎样显示过宽的列标题或串值。WRAP 将换行显示。WORD_WRAP 同样换行，但会拆散单词。TRUNCATED 把值截断为列定义的宽度。

COMMENT

参阅：第 45 章。

格式：

```
COMMENT ON
{ TABLE [schema .] { table | view | materialized view }
| COLUMN [schema .] { table . | view . | materialized view . } column
| OPERATOR [schema .] operator
| INDEXTYPE [schema .] indextype
}
IS 'text';
```

描述：COMMENT 将关于对象或列的注释文本插入数据字典中。

要从数据库中删除一个注释，只需设置它为空串即可(设置文本为‘’)。

在 Oracle Database 11g 中，COMMENT 有一个新的 MINING MODEL 子句，它允许为数据挖掘模型提供描述性注释。

COMMIT

提交意味着永久保留对数据所做的更改(insert、update 和 delete)。在存储所做的更改前，旧数据和新数据都存在，以便更改数据或者还原到以前的状态(“回滚”)。在用户输入 Oracle 的 SQL 命令 COMMIT 时，该事务中所做的所有更改都成为永久性的。

COMMIT

参阅： ROLLBACK 和 SAVEPOINT。

格式：

```
COMMIT [WORK] [ COMMENT 'text' | FORCE 'text' [, integer] ];
```

描述： COMMIT 提交自上一个 COMMIT 命令隐式或显式地执行以来对数据库所做的修改。WORK 是可选的，并且在用法上没有影响。

COMMENT 将一个文本注释与事务相关联。在分布或事务未能完成的事件中，可使用数据字典视图 DBA_2PC_PENDING 查看相应的注释。FORCE 手动提交一个不确定的分布式事务。

COMPOSE

参阅： CONVERSION FUNCTIONS 和第 11 章。

格式：

```
COMPOSE(string)
```

描述： COMPOSE 可以接受任何数据类型的串作为其参数，并返回一个与输入相同的字符集中完全标准化格式的 Unicode 串。

示例： 为显示元音 O，可以使用以下程序：

```
select COMPOSE ( 'o' || UNISTR('\0308') ) from DUAL;

C
-
ö
```

COMPUTE(SQL*Plus)

参阅： BREAK 和 AGGREGATE FUNCTIONS。

格式：

```
COMP[UTE][AVG|COU[NT]|MAX[IMUM]|MIN[IMUM]|NUM[BER]|STD|SUM|VAR[IANCE]]...
    [function LABEL label_name
    OF {expression | column | alias} ...
    ON {expression | column | alias | REPORT | ROW}...]
```

描述： expression 是一列或一个表达式。COMPUTE 对从表中选中的列或表达式进行计算。它只与 BREAK 命令一起使用。

默认时，Oracle 将使用函数名(SUM 和 AVG 等)作为查询输出中结果的标签。LABEL 允许指定重写默认值的 label_name。

OF 命名要计算其值的列或表达式。这些列也必须在 select 子句中，否则将忽略 COMPUTE。

ON 协调 COMPUTE 与 BREAK 命令。在 ON expression 的值被修改或指定的 ROW 或 REPORT 发生中断时，COMPUTE 输出计算的值，并重新开始计算。关于协调的详细内容请

参阅 BREAK。

COMPUTE 自身实际上显示计算的结果。

AVG、MAXIMUM、MINIMUM、STD、SUM 和 VARIANCE 都可以用于数值表达式。MAXIMUM 和 MINIMUM 也可以作用于字符表达式，但不能作用于 DATA 表达式。COUNT 和 NUMBER 作用于任何表达式类型。

除 NUMBER 外，表 A-7 中的所有计算都忽略具有 NULL 值的行：

表 A-7　COMPUTE 中涉及的函数

函 数 名	说　明
AVG	求平均值
COUNT	统计非 NULL 值的个数
MAXIMUM	求最大值
MINIMUM	求最小值
NUMBER	统计所有返回行的数量
STD	求标准偏差
SUM	求非 NULL 值的和
VARIANCE	求方差

连续的计算简单地按顺序进行，不用逗号，如下所示：

```
compute sum avg max of Amount Rate on report
```

此命令计算整个报表的 Amount 与 Rate 的和、平均数和最大值。

示例：为了对每项的分类和整个报表进行计算，输入下列内容：

```
break on Report on Industry skip 1
compute sum of Volume on Industry Report
```

CONNECT

参阅：COMMIT、DISCONNECT 和第 25 章。

格式：

```
CON[NECT] [{ user[/ password] [@ connect_identifier] |/} [AS SYSOPER|SYSDBA]];
```

描述：必须在 SQL*Plus 中使用此命令，尽管不需要登录 Oracle(请参阅 DISCONNECT)。CONNECT 提交任何挂起的更改，退出 Oracle，并作为指定的 user 登录。如果不给出 password，系统就会提示输入。在响应提示而输入口令时，不显示口令。

@connect_identifier 连接到指定的数据库。该数据库可能在主机上，或者在通过 Oracle Net 连接的另一台计算机上。

CONNECT BY

参阅：第 14 章。

格式：

```
SELECT expression [,expression]...
    FROM [user.]table
  WHERE condition
CONNECT BY [PRIOR] expression = [PRIOR] expression
  START WITH expression = expression
ORDER BY expression
```

描述：CONNECT BY 是一个运算符，在 SELECT 语句中用来创建反映树形结构数据(如公司组织，家谱等)的继承性的报表。START WITH 告诉从树的何处开始。下面是几条规则：

- 与 CONNECT BY 表达式有关的 PRIOR 的位置确定哪个表达式标识根，哪个表达式标识树的分支。
- where 子句将从树中删除个别节点，但不删除其子节点(或父节点，这取决于 PRIOR 的位置)。
- CONNECT BY 中的限定条件(特别指不等于号而不是等于号)将删除单个节点及其所有子子点。
- CONNECT BY 不能与 where 子句中的表连接一起使用。

示例：

```
select Cow, Bull, LPAD(' ',6*(Level-1))||Offspring AS Offspring,
       Sex, Birthdate
  from BREEDING
  connect by Offspring = PRIOR Cow
    start with Offspring = 'DELLA'
    order by Birthdate;
```

在本示例中，下面的子句：

```
connect by Offspring = PRIOR Cow
```

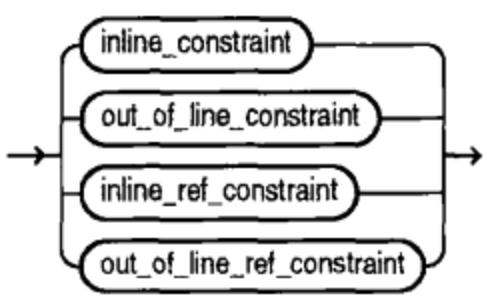
表示相应的子孙是 PRIOR(先于)这头奶牛的牛。

CONSTRAINTS

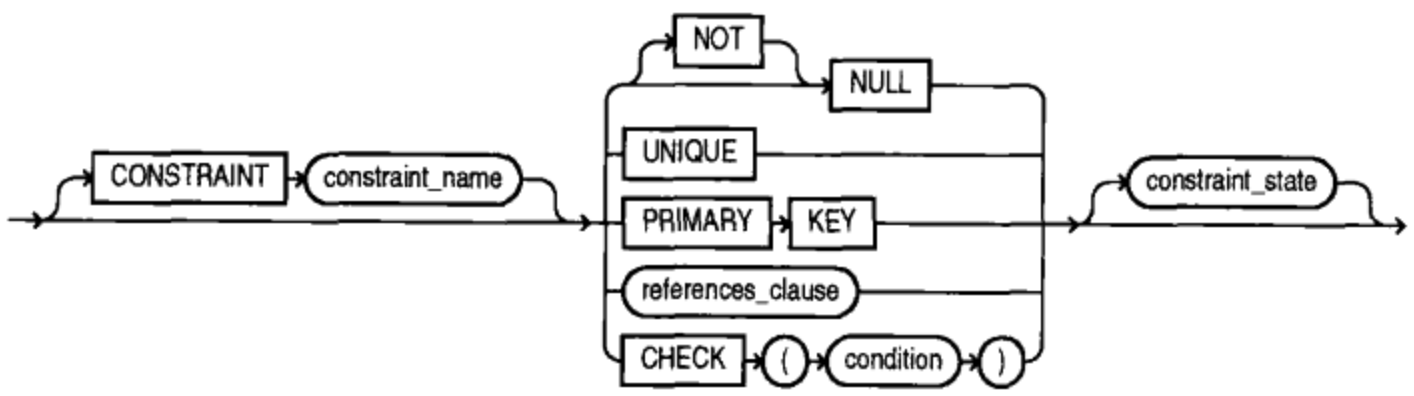
参阅：CREATE TABLE、INTEGRITY CONSTRAINT 和第 17 章。

格式：

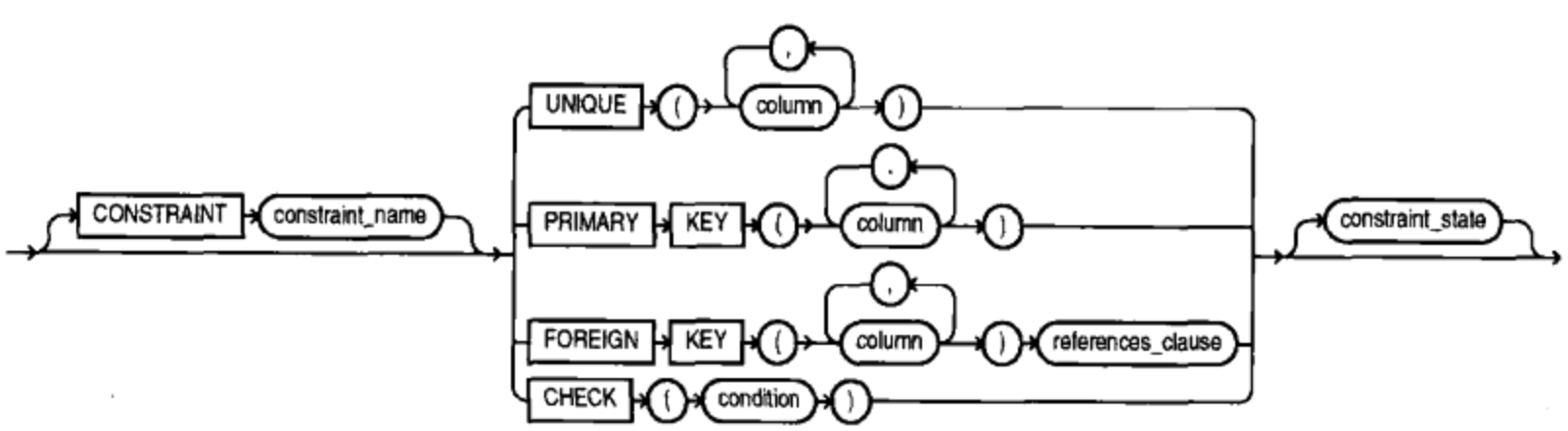
constraint::=



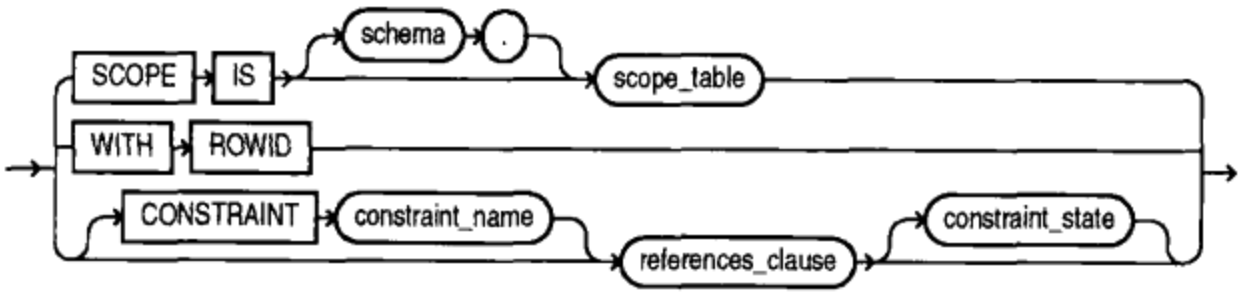
inline_constraint::=



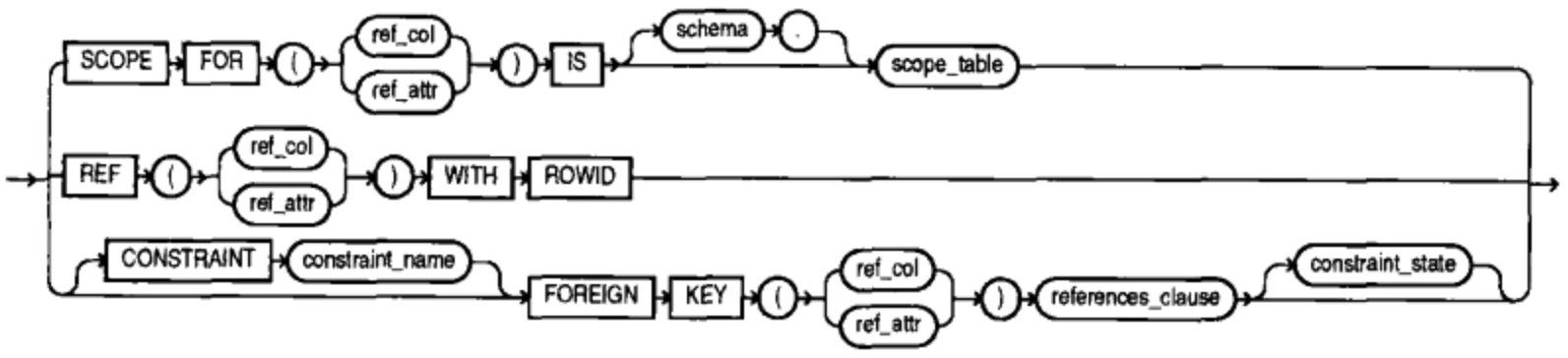
out_of_line_constraint::=



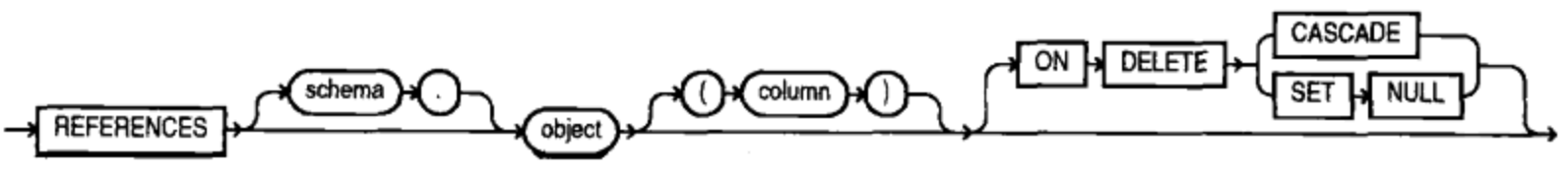
inline_ref_constraint::=



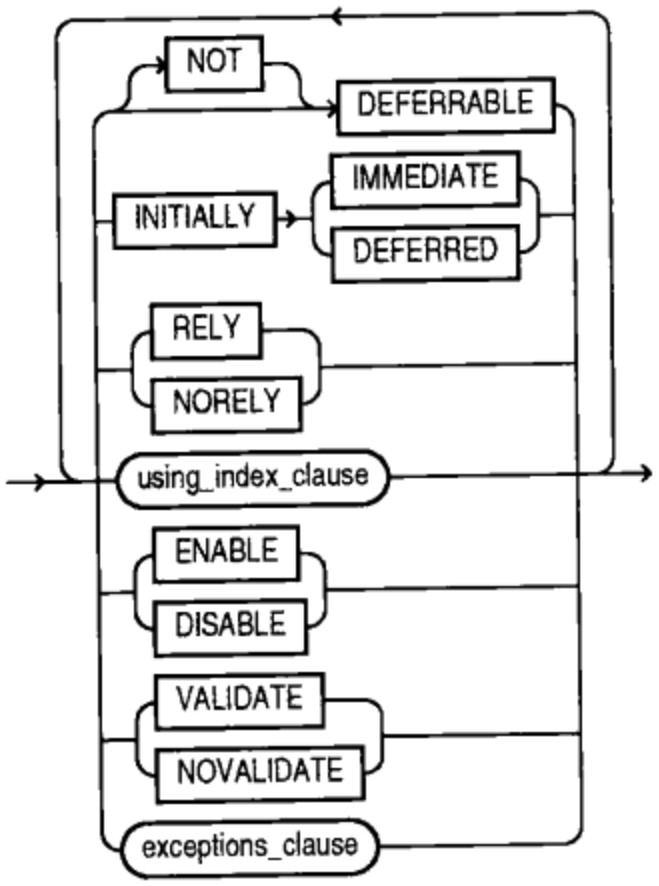
out_of_line_ref_constraint::=



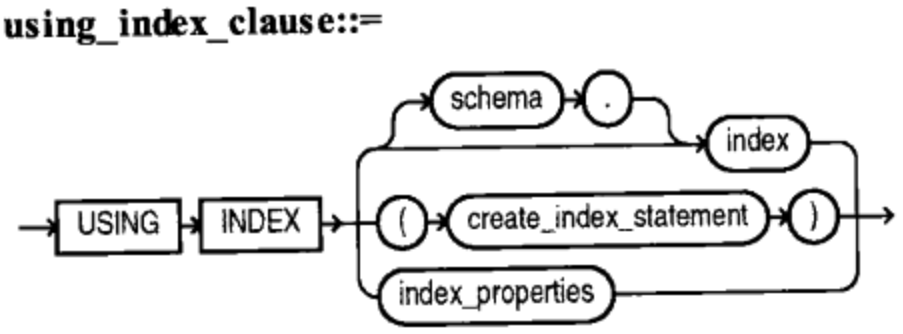
references_clause::=



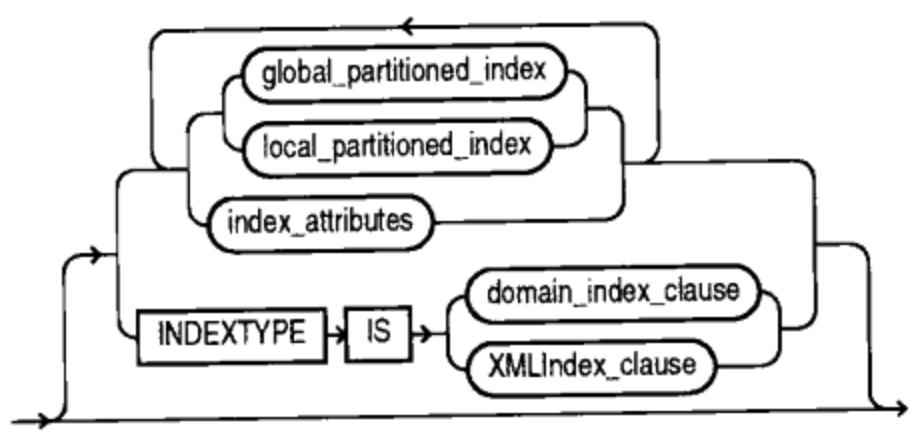
constraint_state::=



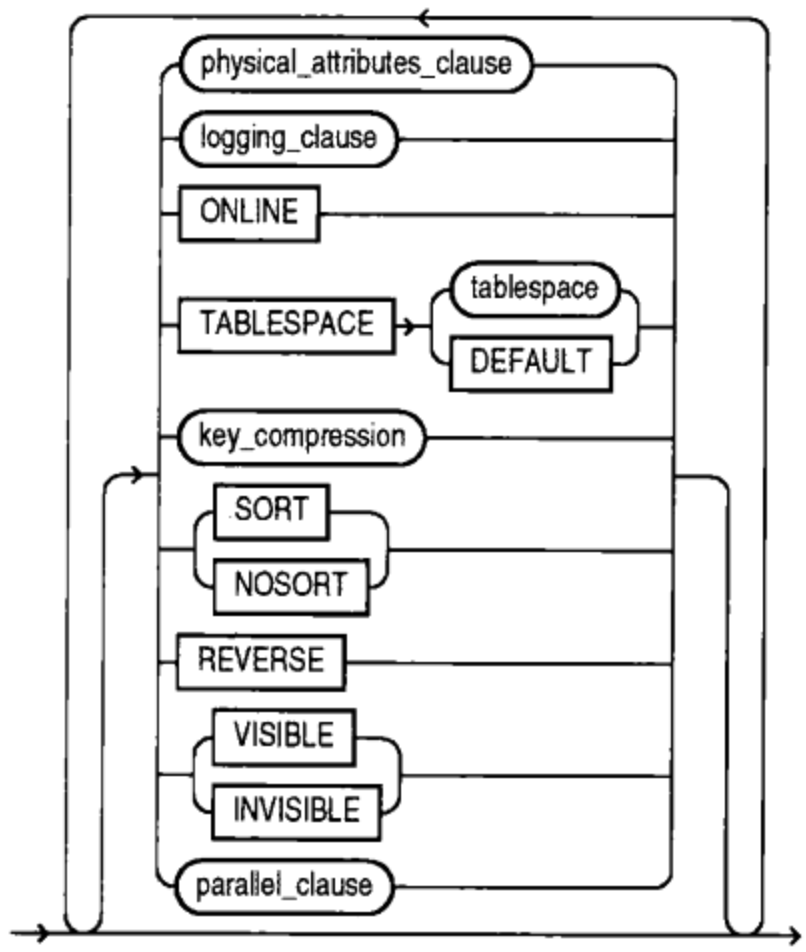
using_index_clause::=



index_properties::=



index_attributes::=



exceptions_clause::=

描述：约束用 CREATE TABLE 和 ALTER TABLE 命令的 RELATIONAL_PROPERTIES 子句定义。使用约束子句可以创建一个约束或更改一个已有的约束。可以启用或禁用约束。如果禁用某个约束然后又试图重新启用它，则 Oracle 将检查数据。如果该约束不能重新启用，则 Oracle 可以把相应的异常写到某个单独的表中，以便于检查。

对于 PRIMARY KEY 和 UNIQUE 约束来说，Oracle 将创建索引。作为这些约束的约束子句的一部分，可以使用 USING INDEX 子句指定表空间和相应索引的存储器。

CONTAINS

CONTAINS 使用 Oracle Text 中的 CONTEXT 索引判定文本搜索。CONTAINS 支持的文本搜索运算符如表 A-8 所示。

表 A-8　CONTAINS 支持的文本搜索运算符

运　算　符	描　　述
OR	如果任何一个搜索项得分超过阈值，则返回一条记录
|	同 OR
AND	如果两个搜索项的得分同时超过阈值，则返回一条记录
&	同 AND
ACCUM	如果搜索项得分的和超过阈值，则返回一条记录

(续表)

运 算 符	描 述
,	同 ACCUM
MINUS	如果第一个搜索项的得分减去第二个搜索项的得分超过阈值，则返回一条记录
-	同 MINUS
*	给搜索项的得分分配不同的权重
NEAR	得分基于搜索项在搜索文本中彼此的接近程度
;	同 NEAR
{}	如果保留字是搜索项的一部分，则括住保留字，如 AND
%	多字符通配符
_	单字符通配符
$	在执行搜索前执行搜索项的词根扩展
?	在执行搜索前执行搜索项的模糊匹配(即允许拼写错误)
!	执行 SOUNDEX(语音)搜索
()	指定判定搜索条件的顺序

CONVERSION FUNCTIONS

参阅： CHARACTER FUNCTIONS、NUMBER FUNCTIONS、AGGREGATE FUNCTIONS 和第 11 章。

描述： 表 A-9 是按字母顺序列出的 Oracle 版本的 SQL 中目前所有的转换函数和变换函数。

表 A-9 Oracle 版本的 SQL 中目前所有的转换函数和变换函数

函 数 名	定 义
ASCIISTR	转换任何字符集中的串，并返回数据库字符集中的 ASCII 串
BIN_TO_NUM	将二进制数值转换为其等价的数值
CAST	将一个内置类型或集合类型 CAST(强制转换)成另一个内置类型或集合类型。通常用于嵌套表和可变数组
CHARTOROWID	修改字符串，使其起 Oracle 内部行标识符(即 ROWID)的作用
COMPOSE	在与输入相同的字符集中，用完全标准化形式将任何数据类型的串转换为一个 Unicode 串
CONVERT	将字符串从一种国家语言字符集 CONVERT(转换)为另一种语言字符集
DECOMPOSE	在与输入相同的字符集中进行规范的分解后，将任意数据类型的串转换成 Unicode 串
HEXTORAW	将十六进制字符串修改为二进制串
NUMTODSINTERVAL	将数字转换成 INTERVAL DAY TO SECOND 字面值
NUMTOYMINTERVAL	将数字转换成 INTERVAL YEAR TO MONTH 字面值
RAWTOHEX	将二进制数的串修改为十六进制的字符串
RAWTONHEX	将 RAW 转换成包含其等价的十六进制值的 NVARCHAR2 字符值
ROWIDTOCHAR	将内部 Oracle 行标识符(ROWID)更改为字符串
ROWIDTONCHAR	将 ROWID 值转换为 NVARCHAR2 数据类型
SCN_TO_TIMESTAMP	将系统变更号转换成近似时间戳
TIMESTAMP_TO_SCN	将时间戳转换成近似系统变更号
TO_BINARY_DOUBLE	返回双精度的浮点数

(续表)

函 数 名	定　义
TO_BINARY_FLOAT	返回单精度的浮点数
TO_CHAR	将 NUMBER 或 DATE 转换成字符串
TO_CLOB	将 LOB 列的 NCLOB 值或其他字符串转换成 CLOB 值
TO_DATE	将 NUMBER、CHAR 或 VARCHAR2 转换成 DATE(一种 Oracle 类型数据)
TO_DSINTERVAL	将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换成 INTERVAL DAY TO SECOND 类型
TO_LOB	将 LONG 转换成 LOB，作为 INSERT AS SELECT 的一部分
TO_MULTI_BYTE	将字符串中的单字节字符转换成多字节字符
TO_NCHAR	将来自数据库字符集中的字符串、NUMBER 或 DATE 转换成国家字符集
TO_NCLOB	将 LOB 列中的 CLOB 值或其他字符串转换成 NCLOB 值
TO_NUMBER	将一个 CHAR 或 VARCHAR2 转换成一个数字
TO_SINGLE_BYTE	将 CHAR 或 VARCHAR2 中的多字节字符转换成单字节
TO_TIMESTAMP	将一个字符串转换为一个 TIMESTAMP 数据类型的值
TO_TIMESTAMP_TZ	将一个字符串转换为一个 TIMESTAMP WITH TIME ZONE 数据类型的值
TO_YMINTERVAL	将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换成 INTERVAL YEAR TO MONTH 类型
TRANSLATE…USING	将一个串中的字符 TRANSLATE(转换)成不同的字符
UNISTR	将一个串转换成数据库中 Unicode 字符集中的 Unicode

CONVERT

参阅：CONVERSION FUNCTIONS。

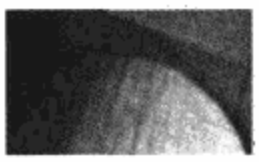
格式：

```
CONVERT(string,[destination_set,[source_set]])
```

描述：CONVERT 用于将 string 中的字符从一种标准的位表示转换为另一种表示，如从 US7ASCII(如果未输入任何一个字符集，则其为默认字符集)转换为 WE8DEC。当在某台计算机上输入的列数据包含在另一台计算机上不能正确显示或输出的字符时，通常使用此函数。在大多数情况下，CONVERT 允许将一种合理的标准转换为另一种标准。最常见的字符集有：

- US7ASCII：美国 7 位 ASCII 码字符集
- WE8ISO8859P1：ISO8859-1 西欧 8 位字符集
- EE8MSWIN1250：Microsoft Windows 东欧 Code Page 1250
- WE8MSWIN1252：Microsoft Windows 西欧 Code Page 1252
- WE8EBCDIC1047：IBM 西欧 EBCDIC Code Page 1047
- JA16SJISTILDE：日本 Shift-JIS 字符集，与 MS Code Page 932 兼容
- ZHT16MSWIN950：Microsoft Windows 繁体汉字 Code Page 950
- UTF8：Unicode 3.0 通用字符集 CESU-8 编码格式
- AL32UTF8：Unicode 5.0 通用字符集 UTF-8 编码格式

COPY(SQL*Plus)



注意：
在 Oracle8i 或其后的版本中，SQL*Plus 的 COPY 命令没有增强，以处理各版本中引入的数据类型或功能。在将来的版本中，COPY 命令可能会作废。

参阅：CREATE DATABASE LINK 和第 25 章。

格式：

```
COPY [FROM user/password@database]
     [TO user/password@database]
     {APPEND | CREATE | INSERT | REPLACE}
     table [ (column [,column]...) ]
     USING query
```

描述：COPY 使用 Oracle Net 将副本 FROM(从)一个表复制 TO(到)另一台计算机的表中。FROM 是源表的用户名、口令和数据库，而 TO 是目标表。FROM 或 TO 可以省略，不论省略的是哪个，缺少的子句都将使用用户的默认数据库。由于源数据库和目标数据库一定不能相同，因此 FROM 子句和 TO 子句只能省略一个。

APPEND 添加到目标表。如果该表不存在，就创建该表。CREATE 要求创建目标表。如果该表已经存在，就会出现“table already exists”错误。INSERT 添加到目标表。如果目标表不存在，就会出现“table does not exist”错误。REPLACE 删除目标表中的数据，并用源表的数据替换它。如果目标表不存在，就创建该表。

table 是目标表名。column 是目标表中的列名。如果指定列名，则列的个数必须与查询中的相同。如果没有指定列，则目标表中的复制列将具有与源表中相同的名称。query 标识源表，并确定从其中复制哪些行和列。

SET LONG(参阅 SET)确定可以复制的长字段的长度。数据长度大于 LONG 值的长列将被截断。SET COPYCOMMIT 确定在提交之前要复制多少组行。SET ARRAYSIZE 确定一组中有多少行。

示例：本示例将图书馆借阅信息(表 BOOKSHELF_CHECKOUT)从 EDMESTON 数据库复制到本地 SQL*Plus 用户连接的数据库。表 LOCAL_CHECKOUT 是由该副本创建的。列在目标表中重命名。请注意每行结束时所使用的短划线(-)。这是必需的。此命令没有用分号结束(因为它是一个 SQL*Plus 命令，而不是 SQL 命令)。关于 COPY 命令的相关选项，请参阅 SET 命令。

```
copy from PRACTICE/PRACTICE@EDMESTON -
    create LOCAL_CHECKOUT (Borrower, Title) -
      using select Name, Title -
              from BOOKSHELF_CHECKOUT
```

CORR

参阅： AGGREGATE FUNCTIONS。

格式：

```
CORR ( expr1 , expr2 ) [OVER ( analytic_clause )]
```

描述： CORR 返回一组数字对的相关系数。expr1 和 expr2 都是数字表达式。在去除 expr1 或 expr2 为 NULL 的数字对后，Oracle 将该函数应用到(expr1，expr2)组上。然后 Oracle 进行以下计算：

```
COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))
```

此函数返回 NUMBER 类型的一个值。如果此函数应用到一个空集上，则返回 NULL。

CORR_K 和 CORR_S

参阅： CORR。

格式：

```
{CORR_K | CORR_S }
 ( expr1, expr2 [, { COEFFICIENT |ONE_SIDED_SIG|TWO_SIDED_SIG}])
```

描述： CORR_K 和 CORR_S 函数支持非参数或排序相关。相关系数接受范围为 - 1~1 之间的一个值。1 表示完全正相关， - 1 表示完全负相关。参数必须是数值或隐式转换成数值数据类型。ONE_SIDED_SIG 用于相关性的单侧显著性。TWO_SIDED_SFG 是用于双侧显著性。

COS

参阅： ACOS、ASIN、ATAN、ATAN2、COSH、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

格式：

```
COS(value)
```

描述： COS 返回一个值的余弦，角度用弧度表示。将角度乘以 π/180，可以将一个角度值转换为弧度。

COSH

参阅： ACOS、ASIN、ATAN、ATAN2、COS、EXP、LN、LOG、SIN、SINH、TAN 和 TANH。

格式：

```
COSH(value)
```

描述： COSH 返回一个值的双曲余弦。

COUNT

参阅： AGGREGATE FUNCTIONS、第 9 章和第 12 章。

格式：

```
COUNT ( ( * | [ DISTINCT | ALL ] expr ) ) [OVER ( analytic_clause )]
```

描述： COUNT 统计 expression 为非 NULL 的行数，然后通过查询返回该计数值。使用 DISTINCT，COUNT 只统计具有不同值的非 NULL 行。如果使用“*”，COUNT 则对所有的行进行统计，而不论它是否为 NULL。

COVAR_POP

参阅： AGGREGATE FUNCTIONS。

格式：

```
COVAR_POP ( expr1 , expr2 ) [OVER ( analytic_clause )]
```

描述： COVAR_POP 返回一组数字对的总体协方差。可以将它作为一个聚集函数或分析函数。

expr1 和 expr2 都是数字表达式。在去除 expr1 或 expr2 为 NULL 的所有数字对后，Oracle 将该函数应用到(expr1，expr2)组上。之后 Oracle 进行以下计算：

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

这里的 n 是指 expr1 和 expr2 都非 NULL 的(expr1，expr2)对的个数。

COVAR_SAMP

参阅： AGGREGATE FUNCTIONS。

格式：

```
COVAR_POP ( expr1 , expr2 ) [OVER ( analytic_clause )]
```

描述： COVAR_SAMP 返回一组数字对的样本协方差。可以将它作为一个聚集函数或分析函数。

expr1 和 expr2 都是数字表达式。在去除 expr1 或 expr2 为 NULL 的所有数字对后，Oracle 将该函数应用到(expr1，expr2)组上。之后 Oracle 进行以下计算：

```
(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)
```

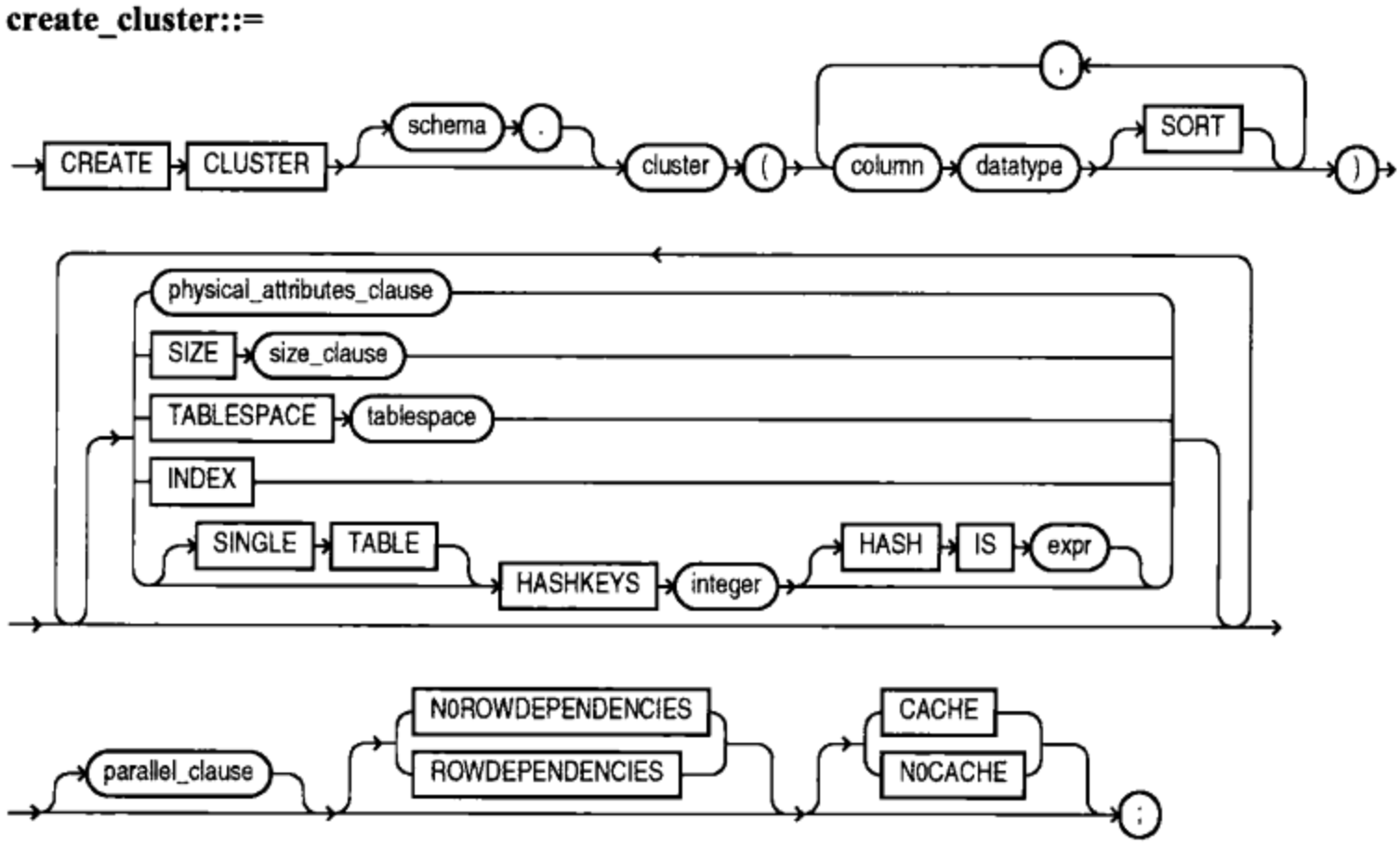
这里的 n 是指 expr1 和 expr2 都非 NULL 的(expr1，expr2)对的个数。

CREATE CLUSTER

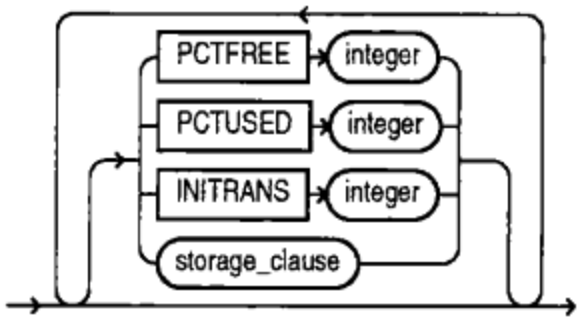
参阅： CREATE INDEX、CREATE TABLE 和第 17 章。

格式：

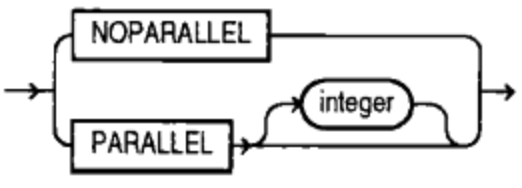
create_cluster::=



physical_attributes_clause::=



parallel_clause::=



描述：CREATE CLUSTER 为一个或多个表创建一个群集。使用具有 CLUSTER 子句的 CREATE TABLE 将表添加到群集中。CREATE CLUSTER 至少需要来自每个表的一个群集列。它们必须有相同的数据类型和大小，但不需要有相同的名称。对于一个群集中的表，具有相同群集列值的行保存在磁盘的相同区域(相同的逻辑块)中。当群集列为连接表通常所用的列时，这能够改善性能。

每个群集列中的每个不同值只能存储一次，无论它在表中或行中出现一次或多次。这样通常可以减少存储表所需的磁盘空间，但每个表看起来仍然像包含自己的所有数据一样。具有 LONG 列的表不能聚集成群集。

cluster 是为群集创建的名称。column 和 datatype 除不能指定为 NULL 和 NOT NULL 外，与 CREATE TABLE 的方法相同。但是，在实际的 CREATE TABLE 语句中，群集中至少有一个群集列必须为 NOT NULL(非空)。SIZE 以字节为单位，用来设置一个逻辑块(不是物理块)的大小。SPACE 是群集的初始磁盘分配，与 CREATE TABLE 中所用的一样。

SIZE 是存储与单个群集键相关联的所有群集表的所有行所需空间的总平均量。较小的 SIZE 值可能会增加访问群集中的表所需要的时间，但可以减少所用的磁盘空间。SIZE 应当是物理块大小的一个恰当的因子(divisor)。否则，Oracle 将使用下一个更大的因子。如果 SIZE 超过了物理块大小，Oracle 就使用物理块大小来替换。

默认情况下，为群集创建了索引。在将任何数据放入群集之前，必须在群集键上创建索引。然而，如果指定散列群集形式，就不必(不能够)在群集键上创建索引。相反，Oracle 使用一个散列函数来存储表的行。

可以创建自己的散列值来作为表的一列，并使用 HASH IS 子句告诉 Oracle 将该列作为散列值。否则，Oracle 将使用一个基于群集键列的内部散列函数。HASHKEYS 子句实际创建散列群集并指定散列值的个数，四舍五入为最接近的素数。最小值为 2。

关于常用存储子句参数的详细内容请参阅 STORAGE。

CREATE CONTEXT

格式：

```
CREATE [OR REPLACE] CONTEXT namespace USING [schema .] package
[ INITIALIZED { EXTERNALLY | GLOBALLY }
| ACCESSED GLOBALLY ] ;
```

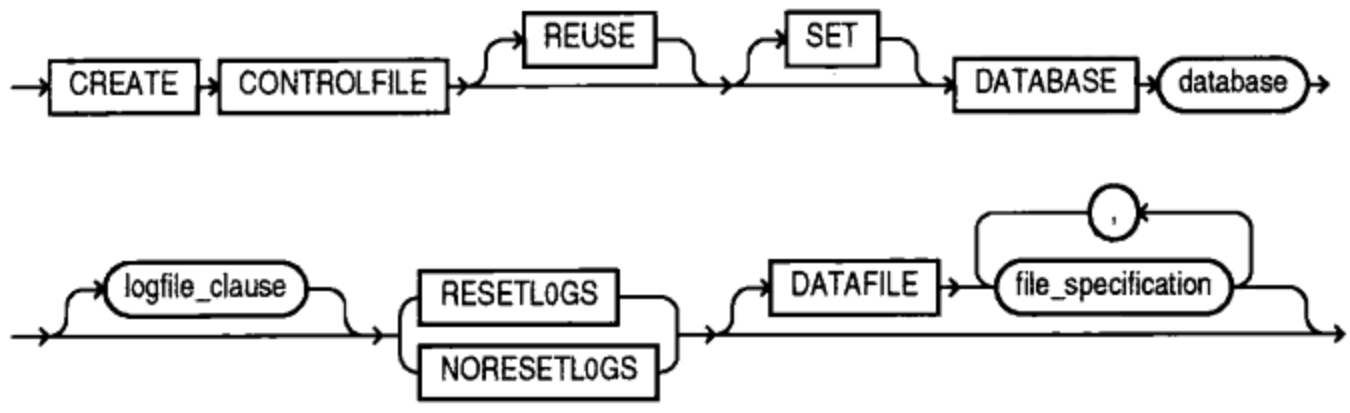
描述：上下文(context)是一组用来保护应用程序的属性。CREATE CONTEXT 为 context 创建一个名称空间，并把此名称空间与外部创建的设置上下文的程序包相关联。要创建上下文名称空间，必须拥有 CREATE ANY CONTEXT 系统权限。

CREATE CONTROLFILE

参阅：ALTER DATABASE 和 CREATE DATABASE。

格式：

create_controlfile::=

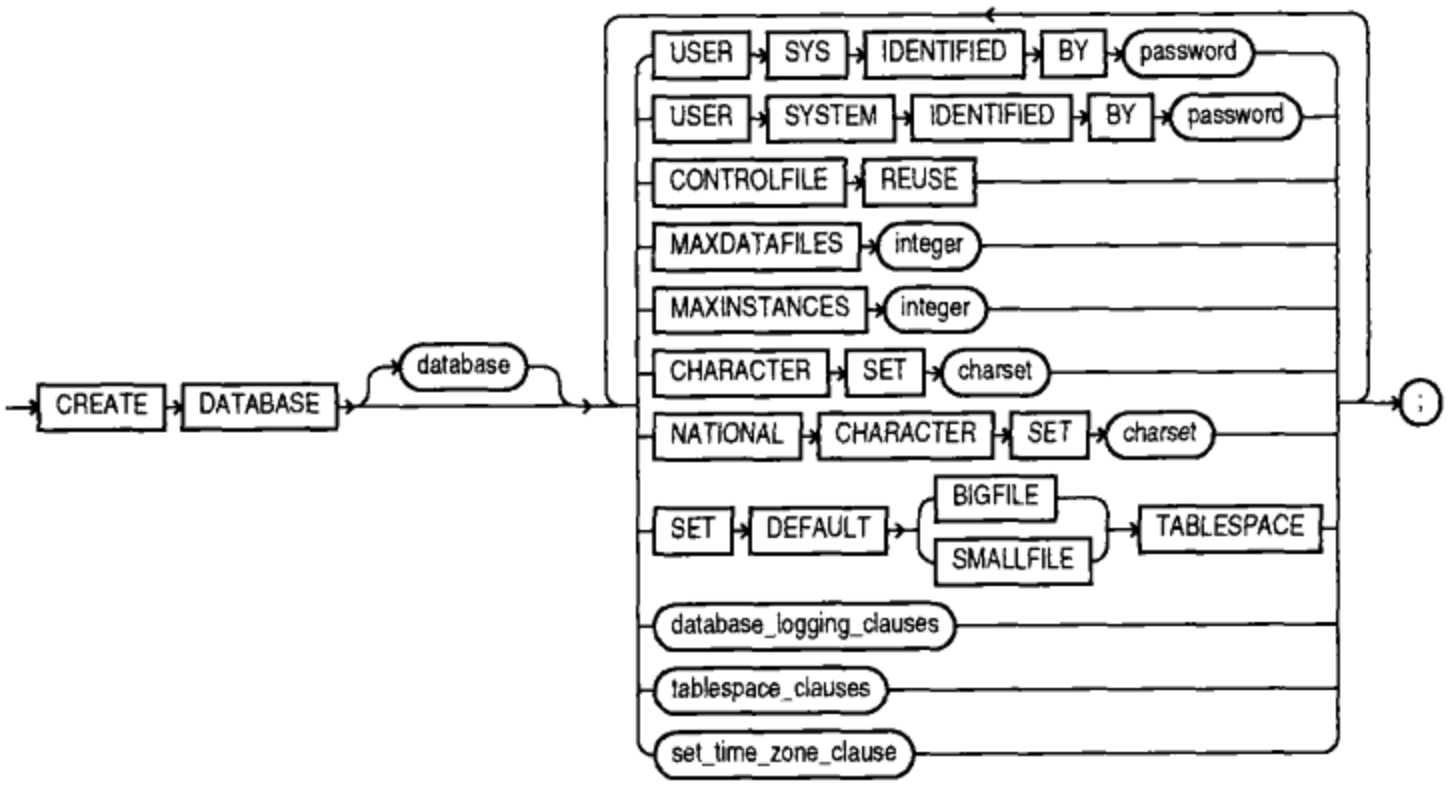


CREATE DATABASE

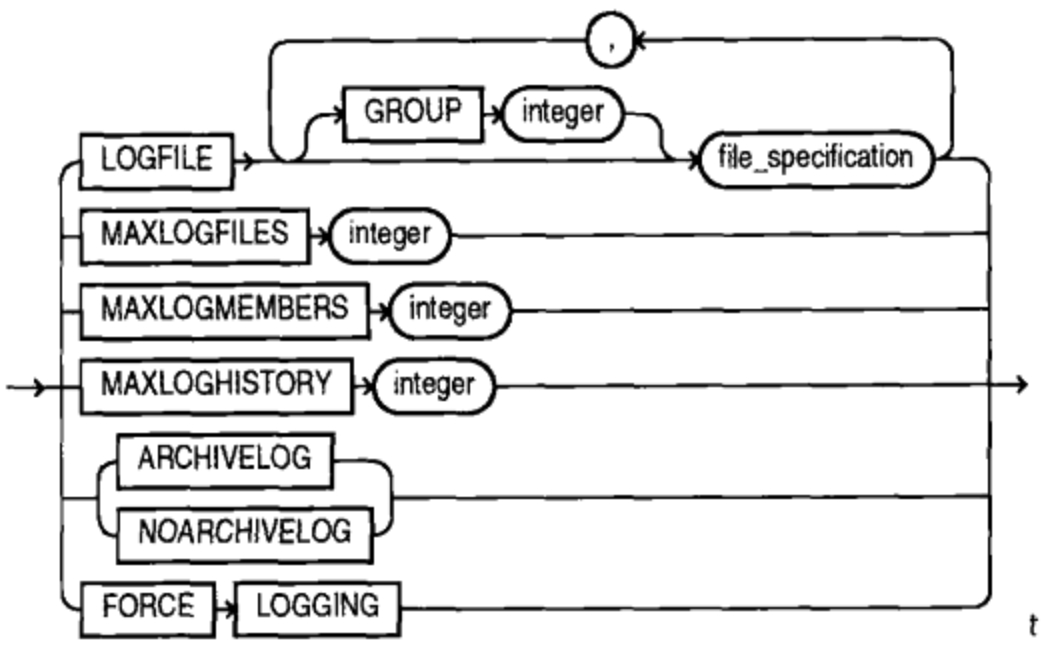
参阅：ALTER DATABASE、CREATE CONTROLFILE、CREATE ROLLBACK SEGMENT、CREATE TABLESPACE、SHUTDOWN、STARTUP、第 2 章和第 51 章。

格式：

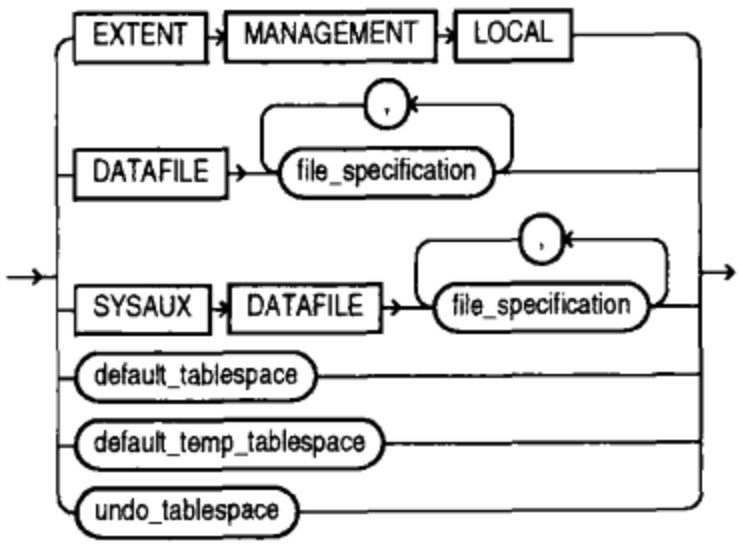
create_database::=



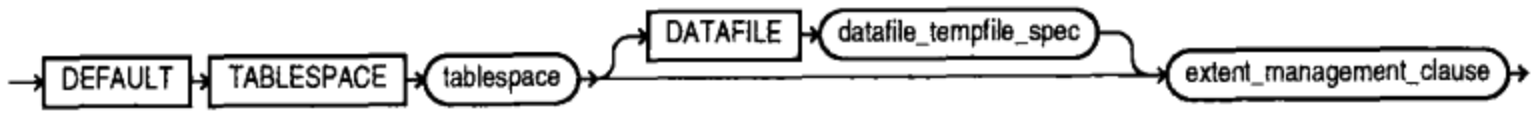
database_logging_clauses::=



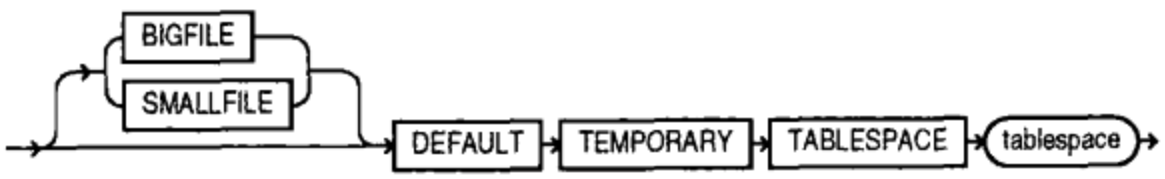
tablespace_clauses::=



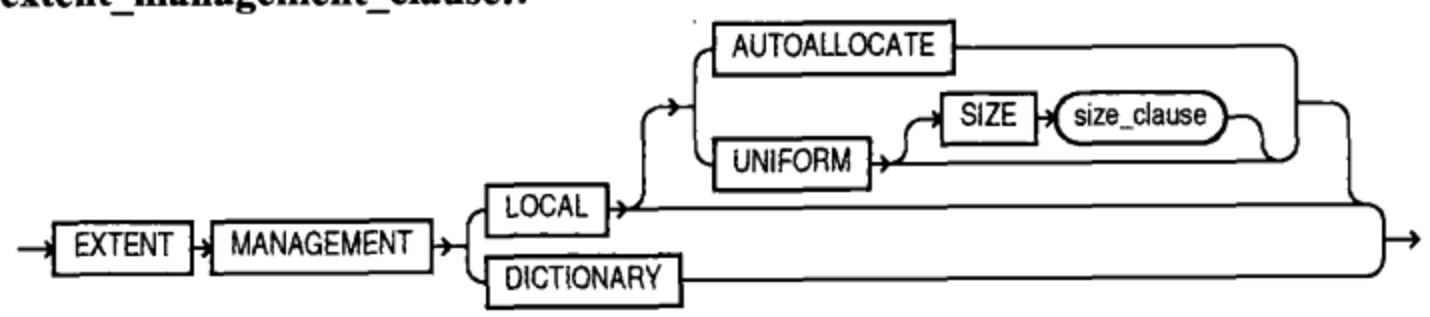
default_tablespace::=



default_temp_tablespace::=

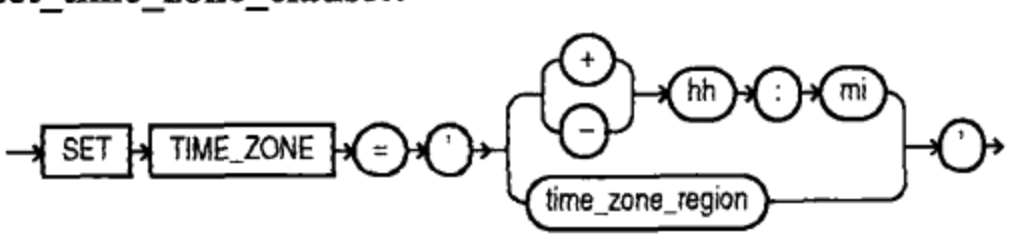


extent_management_clause::=

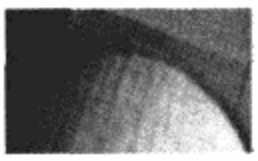


undo_tablespace::=

set_time_zone_clause::=



描述：database 是数据库名，不能多于 8 个字符。init.ora 中的 DB_NAME 包含默认数据库名。一般来说，这条命令仅应由经验丰富的数据库管理员使用。



注意：
在一个已有的数据库中使用这条命令将会删除指定的数据文件。

file_definition 定义重做日志文件名和数据文件名与大小。

```
'file' [SIZE integer [K | M] [REUSE]
```

SIZE 是为该文件预留的字节数。若后缀为 K，则预留的字节数为该值乘以 1 024；若后缀为 M，则乘以 1 048 576(也支持 G 和 T)。REUSE(不带 SIZE)表示销毁具有此文件名的任何文件中的内容，并且把这个名称与数据库相关联。带 SIZE 的 REUSE 在文件不存在时创建文件，在文件存在就检查其大小。CONTROLFILE REUSE 重写初始化参数文件中 CONTROL_FILES 参数定义的已有的控制文件。

LOGFILE 命名用作重做日志文件的文件。如果不使用此参数，则 Oracle 将默认创建两个文件。MAXLOGFILES 重写 LOG_FILES 初始化参数，并定义这个数据库任何时候可以创建的重做日志文件的最大数目。除非重新创建此控制文件，否则这个数目以后不能增加。文件的最小数目为 2。使用较大的数仅仅会生成稍微大点的控制文件。

DATAFILE 命名数据库自身使用的文件。MAXDATAFILES 设置该数据库可以创建的文件的绝对上限，并重写 DB_FILES 初始化参数。使用较大的数仅仅会生成稍微大点的控制文件。

可以指定自动存储管理(Automatic Storage Management)文件和文件系统文件。在为一个数据文件打开(ON)AUTOEXTEND 选项时，该数据文件将根据需要以 NEXT 大小的增量动态地扩展，最大可达到 MAXSIZE(或 UNLIMITED)。

MAXINSTANCES 重写 init.ora 中的 INSTANCES 参数，并设置同时可安装和打开这个数据库的实例的最大数目。

ACHIVELOG 和 NOARCHIVELOG 在第一次创建数据库时定义使用重做日志文件的方式。NOARCHIVELOG 是默认的，它表示重用重做日志文件而不在别处保留其内容。这样便提供了实例恢复，但不对介质故障(如磁盘崩溃)进行恢复。ACHIVELOG 强制归档重做日志文件(通常归档到另外的磁盘或磁带)，这样就可对介质故障进行恢复。这种模式也支持实例恢复。此参数可以被 ALTER DATABASE 重新设置。

MAXLOGMEMBERS 选项指定重做日志文件组的副本的最大数目。MAXLOGHISTORY 选项指定归档重做日志文件的最大数目，仅在归档重做日志文件用于实时应用群集时。CHARACTER SET 选项指定用来存储数据的字符集，此字符集取决于具体的操作系统。

使用撤消(回滚)段的自动化程度更高的处理，可以指定 UNDO TABLESPACE 子句来分配一个专门用来保存已撤消数据的表空间。数据库必须以自动撤消管理(AUM)模式启动。

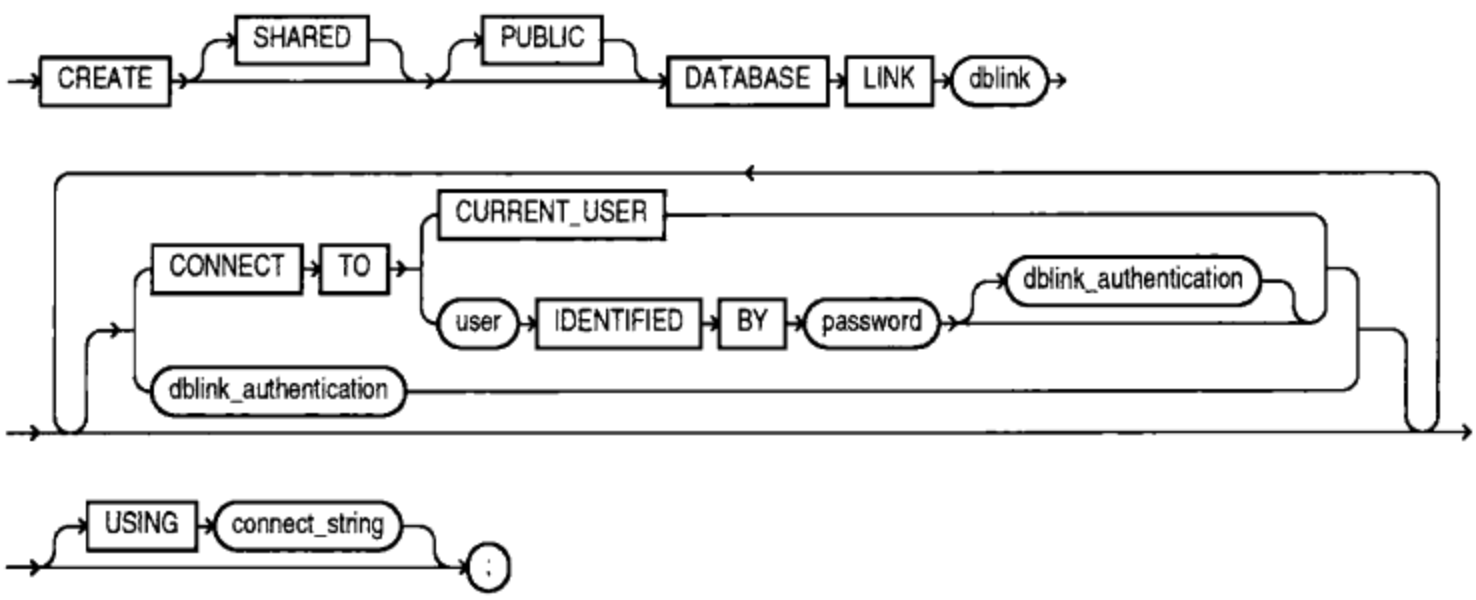
可以使用 DEFAULT TEMPORARY TABLESPACE 子句指定一个非 SYSTEM 的表空间，作为在数据库中创建的所有新用户的默认临时表空间。

CREATE DATABASE LINK

参阅：CREATE SYNONYM、SELECT 和第 25 章。

格式：

create_database_link::=



dblink_authentication::=

描述：dblink 是给链接提供的名称。connect_string 是可以通过 Oracle Net 访问的远程数据库的定义，它定义本地数据库与远程数据库上的一个用户名之间的链接。PUBLIC 链接只能由拥有 CREATE PUBLIC DATABASE LINK 系统权限的用户创建，但之后除那些已经用相同的名称创建了私有链接的用户外，所有用户都可以使用这些 PUBLIC 链接。如果不指定 PUBLIC，则此链接只有那些执行 CREATE DATABASE LINK 语句的用户可以使用。connect_string 是远程数据库的 Oracle Net 服务名。

可以像访问本地表一样访问远程表，只不过在 SELECT 语句中的 FROM 子句中，远程表名必须以@link 作为后缀。大多数系统设置同时链接的最大数目为 4。DBA 可以利用 init.ora 中的 OPEN_LINKS 参数增加这个数。

树状结构查询具有一定的限制。除在 CONNECT BY 子句中以外，它们不能使用 PRIOR 运算符。START WITH 不能包含子查询。CONNECT BY 和 START WITH 不能使用函数 USERENV(‘ENTRYID’)或伪列 RowNum。

要创建数据库链接，必须拥有本地数据库的 CREATE DATABASE LINK 权限和远程数据库的 CREATE SESSION 权限。要创建一个公有数据库链接，必须拥有 CREATE PUBLIC DATABASE LINK 系统权限。

如果使用 CONNECT TO CURRENT_USER 子句，则此链接将试图使用当前的用户名和口令打开远程数据库中的一个连接。因此必须协调在本地数据库和远程数据库之间所做的口令修改，否则数据库链接可能停止工作。

如果使用共享服务器体系结构，就可以创建 SHARED 数据库链接，从而不必创建借助于它们的许多不同的专用连接。在创建 SHARED 链接时，必须用远程数据库中有效的用户名和口令作为相应连接的身份验证。

示例：以下代码定义一个名为 EDMESTON_BOOKS 的链接，该链接连接数据库 EDMESTON 上的 Practice 用户名。

```
create database link EDMESTON_BOOKS
  connect to Practice identified by Practice
    using 'EDMESTON';
```

现在可以查询 Practice 的表：

```
select Title, Publisher
  from BOOKSHELF@EDMESTON_BOOKS;
```

也可以创建同义词来隐藏表的远程性：

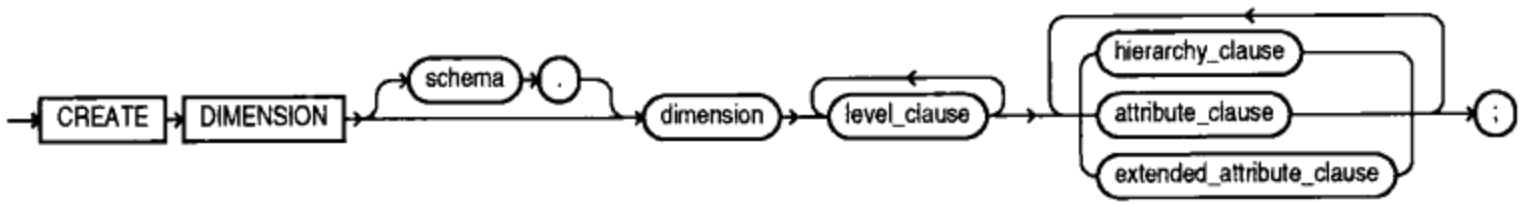
```
create synonym BOOKSHELF for BOOKSHELF@EDMESTON_BOOKS;
```

CREATE DIMENSION

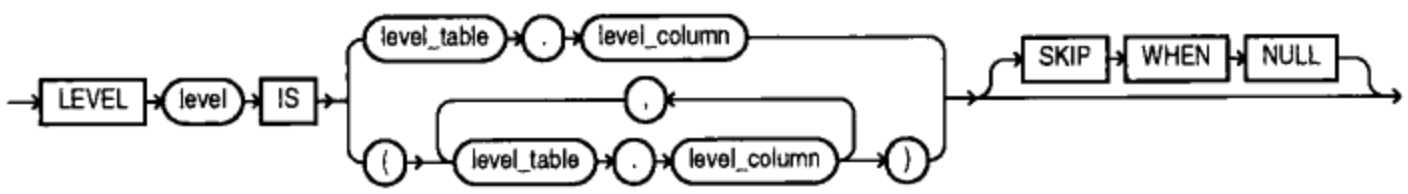
参阅：ALTER DIMENSION、DROP DIMENSION 和第 26 章。

格式：

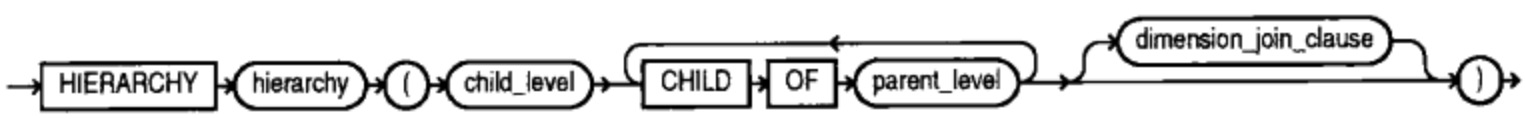
create_dimension::=



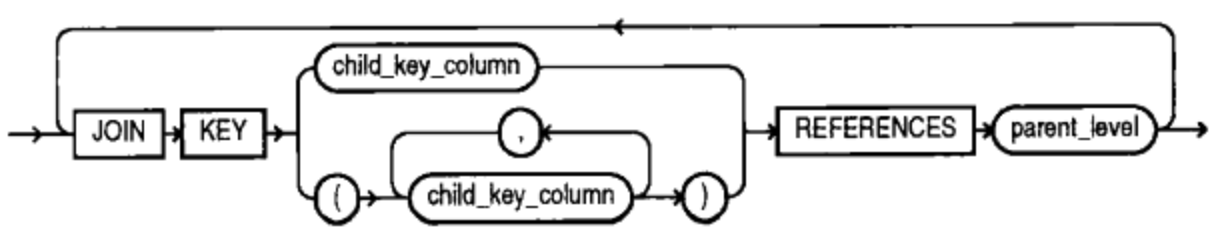
level_clause::=



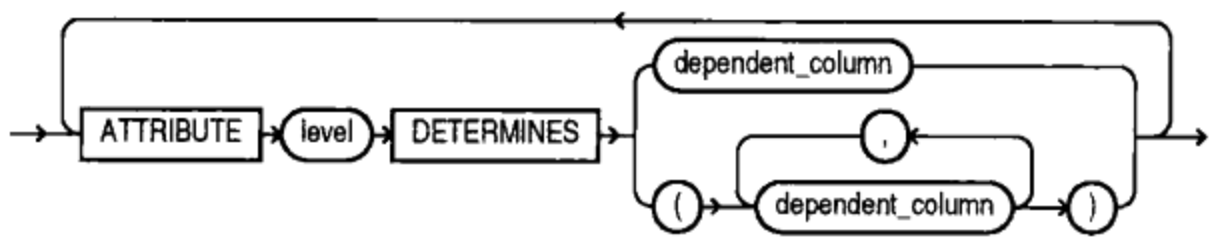
hierarchy_clause::=



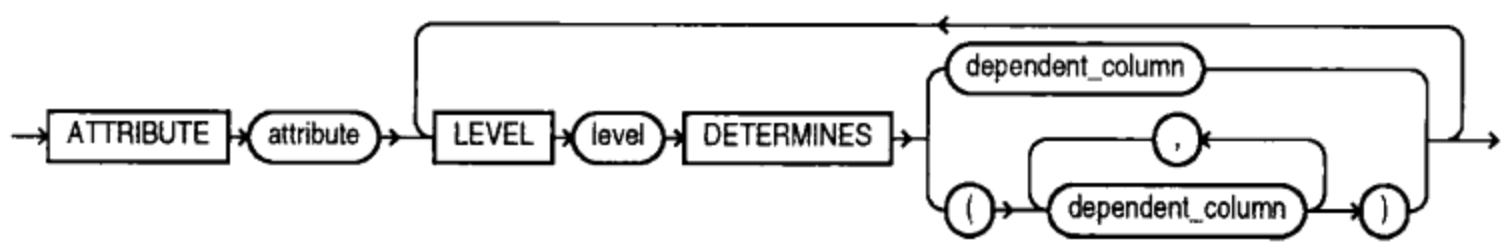
dimension_join_clause::=



attribute_clause::=



extended_attribute_clause::=



描述：CREATE DIMENSION 创建表中相关列之间的层次结构，以供优化程序使用。优化程序在确定物化视图是否返回与其基表相同的数据时，将要使用维度值。要创建维度，必须拥有 CREATE DIMENSION 权限。要在其他用户的模式中创建维度，必须拥有 CREATE ANY DIMENSION 权限。

LEVEL 定义维度中的级别。HIERARCHY 定义级别之间的关系。ATTRIBUTE 将指定的属性分配给维度中的级别。JOIN_KEY 定义级别之间的连接子句。

示例：考虑地理维度，由于大洲被分成不同的国家，因此国家数据和洲数据之间有层次结构。对于名为 COUNTRY 的表(具有 Country 列和 Continent 列)和名为 CONTINENT 的第二个表(具有名为 Continent 的列)来说，可以使用以下程序：

```
create table CONTINENT (
  Continent VARCHAR2(30));

create table COUNTRY (
Country VARCHAR2(30) not null,
Continent VARCHAR2(30));

create dimension GEOGRAPHY
level COUNTRY_ID     is COUNTRY.Country
level CONTINENT_ID   is CONTINENT.Continent
hierarchy COUNTRY_ROLLUP (
   COUNTRY_ID child of
   CONTINENT_ID
join key COUNTRY.Continent references CONTINENT_ID);
```

CREATE DIRECTORY

参阅：BFILE、第 28 和第 40 章。

格式：

```
CREATE [OR REPLACE] DIRECTORY directory AS ' path_ name';
```

描述：在 Oracle 中，“directory”是操作系统目录的别名。在访问 BFILE 数据类型值或

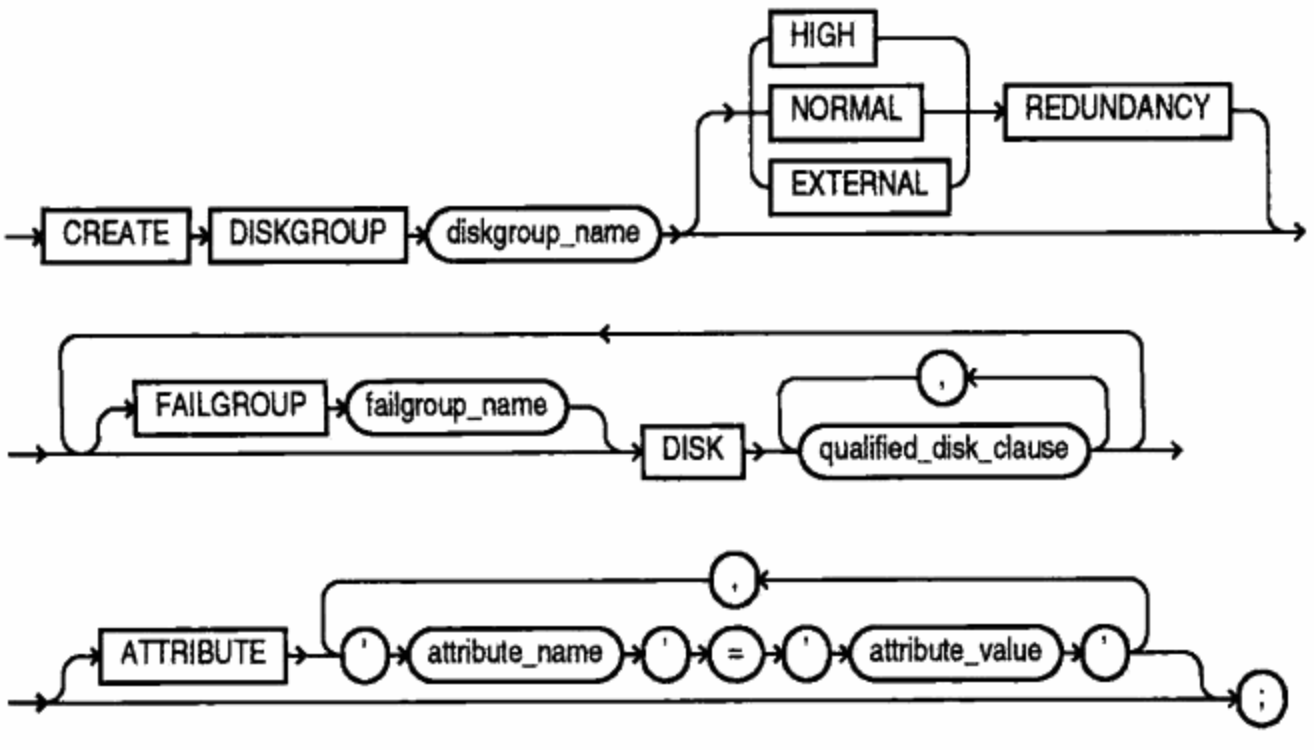
外部表之前，必须先创建一个目录。

CREATE DISKGROUP

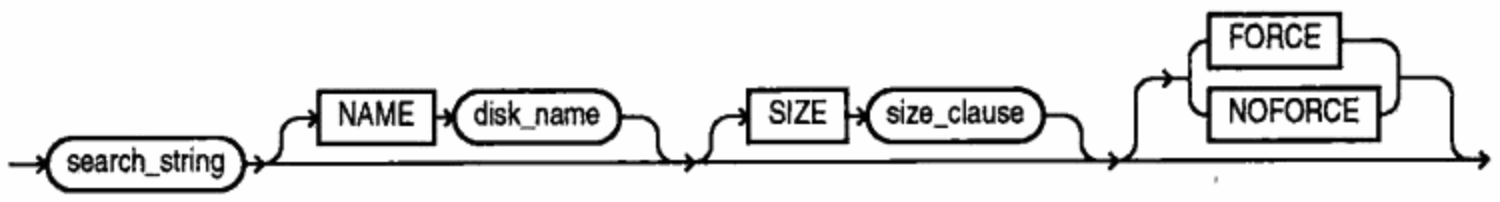
参阅： ALTER DISKGROUP、DROP DISKGROUP 和第 51 章。

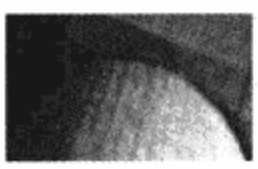
格式：

create_diskgroup::=



qualified_disk_clause::=





注意：

CREATE DISKGROUP 仅在使用自动存储管理(ASM)且已经启动了一个 ASM 实例时才有效。

描述：

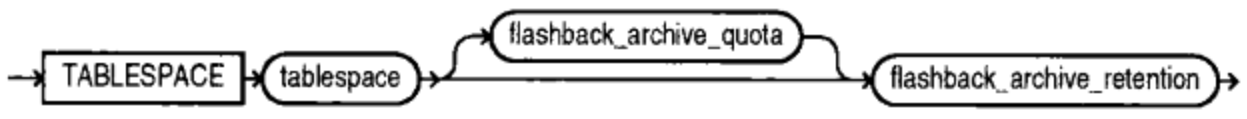
CREATE DISKGROUP 创建一个磁盘集合。Oracle 作为一个逻辑单元来管理磁盘组，并在磁盘上均匀地分布每个文件，以平衡 I/O。Oracle 在磁盘组的可用磁盘上自动分布数据库文件，且当存储配置修改时自动重新平衡存储空间。CREATE DISKGROUP 创建磁盘组、把磁盘分配给磁盘组并首次安装磁盘组。

CREATE FLASHBACK ARCHIVE

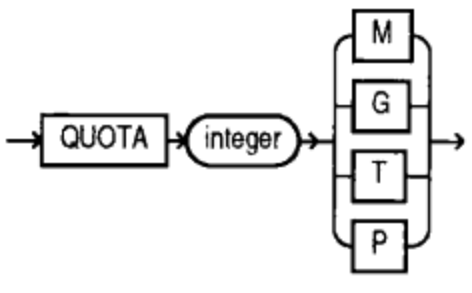
参阅： ALTER FLASHBACK ARCHIVE 和 DROP FLASHBACK ARCHIVE。

格式：

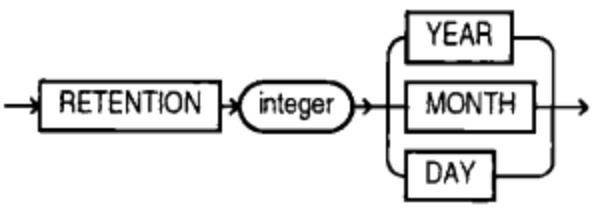
create_flashback_archive::=



flashback_archive_quota::=



flashback_archive_retention::=



描述：使用 CREATE FLASHBACK ARCHIVE 创建闪回数据归档，这提供了自动跟踪事务数据的变更，并将这些变更归档到特定数据库对象中的功能。闪回数据归档由多个表空间组成，并存储来自跟踪表的所有事务的历史数据。

闪回数据归档保留历史数据，历史数据的持续时间使用 RETENTION 参数来指定。历史数据可以使用闪回查询 AS OF 子句来查询。超过指定保留期的已归档历史数据自动被清除。

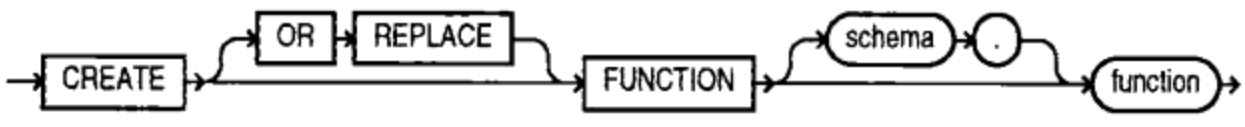
必须拥有 FLASHBACK ARCHIVE ADMINISTER 系统权限才能创建闪回数据归档。此外，必须拥有 CREATE TABLESPACE 系统权限才能创建闪回数据归档，同时在历史信息将要驻留的表空间上必须拥有充足的配额。为了将闪回数据归档指定为系统默认的闪回数据归档，必须以 SYSDBA 登录。

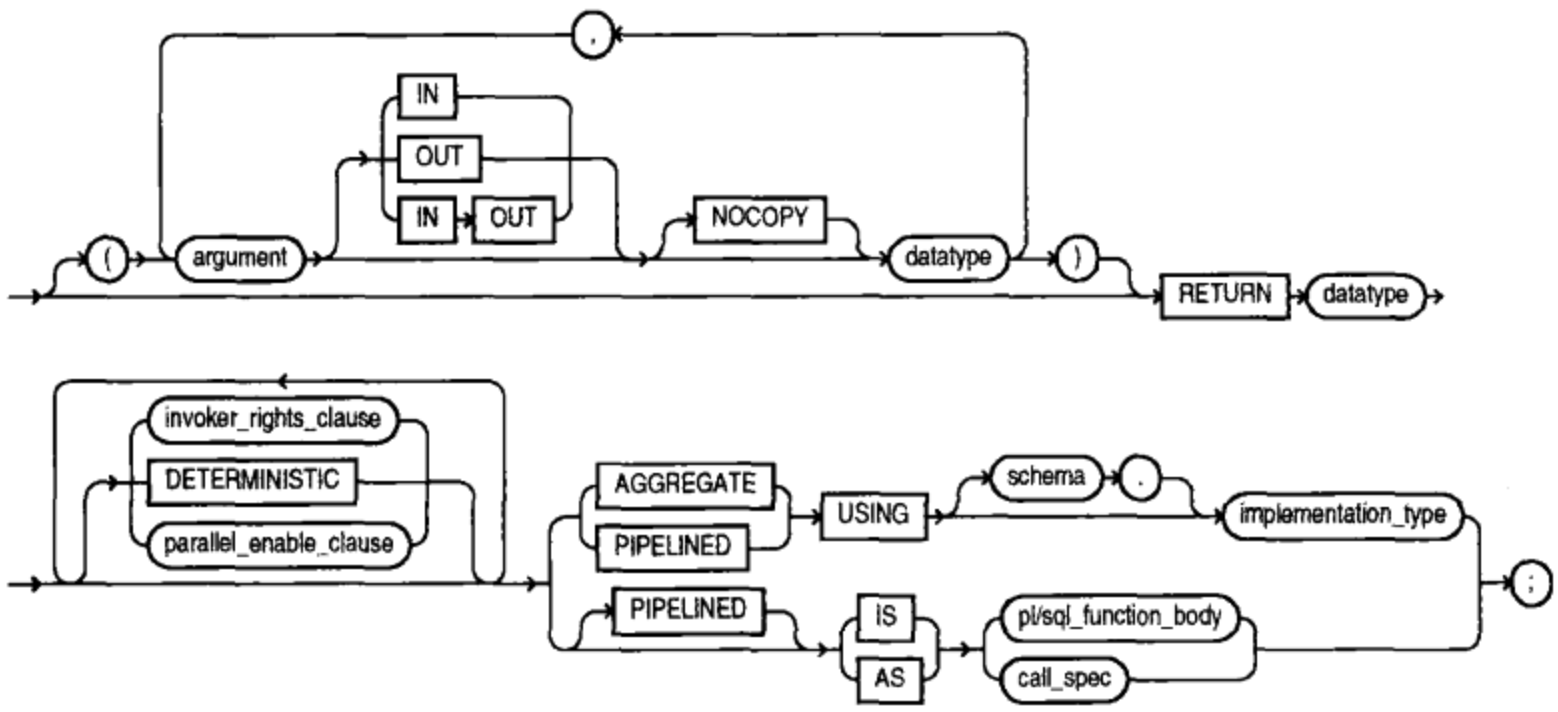
CREATE FUNCTION

参阅：ALTER FUNCTION、BLOCK STRUCTURE、CREATE LIBRARY、CREATE PACKAGE、CREATE PROCEDURE、DATATYPES、DROP FUNCTION 和第 35 章。

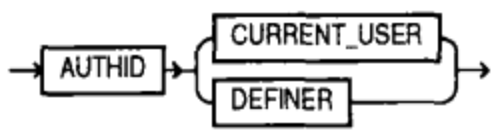
格式：

create_function::=

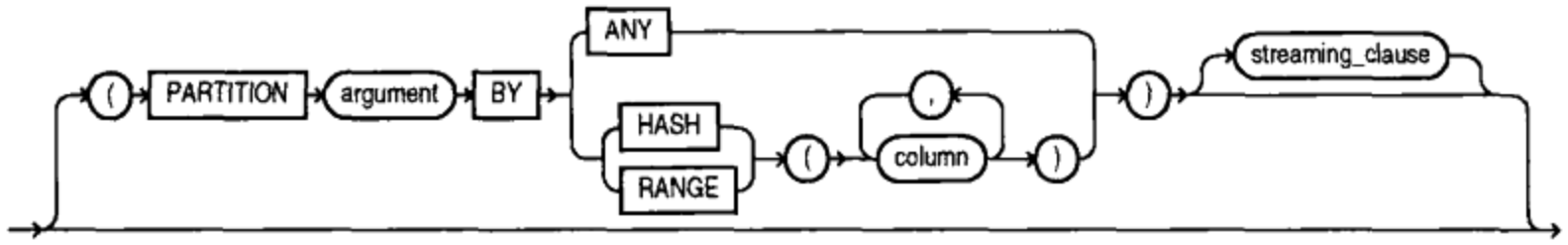




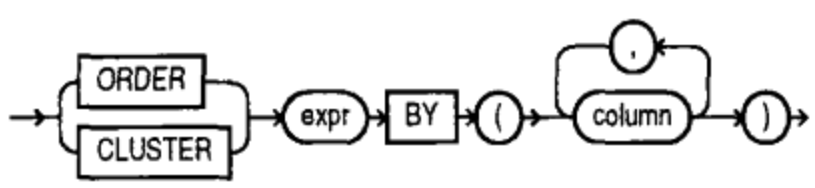
invoker_rights_clause::=



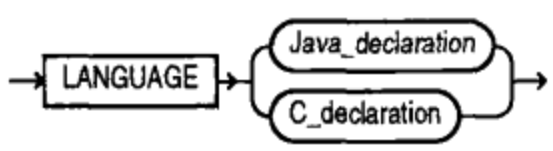
parallel_enable_clause::=



streaming_clause::=

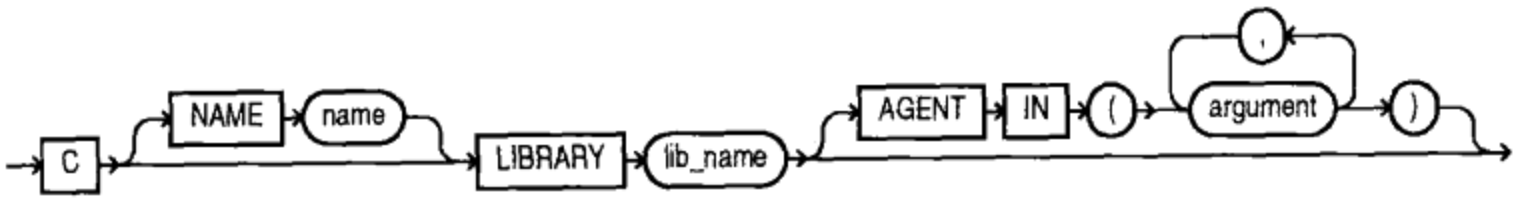


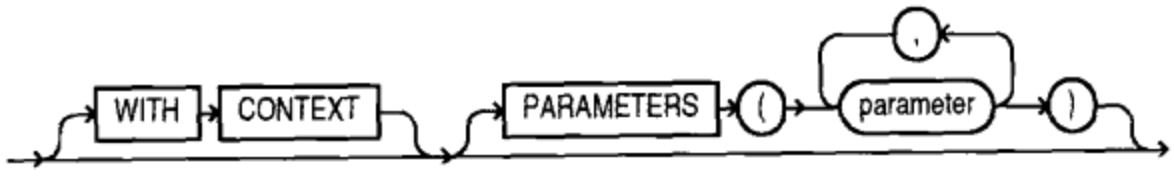
call_spec::=



Java_declaration::=

C_declaration::=





描述：function 是正在定义的函数名。函数可以有参数、某种数据类型的命名参数，每个函数都要按 RETURN(返回)子句指定的数据类型返回一个值。PL/SQL 块将函数的行为定义为一系列声明、PL/SQL 程序语句和异常。

IN 限定符表示在调用函数时，必须给参数指定一个值。但因为对于一个函数来说，这是必须做的工作，所以语法是可选的。在过程中，可以有其他类型的参数。函数和过程之间的差别在于函数返回一个值给主调环境。

要创建一个函数，必须拥有 CREATE PROCEDURE 系统权限。为了在另一个用户的账户中创建函数，必须拥有 CREATE ANY PROCEDURE 系统权限。

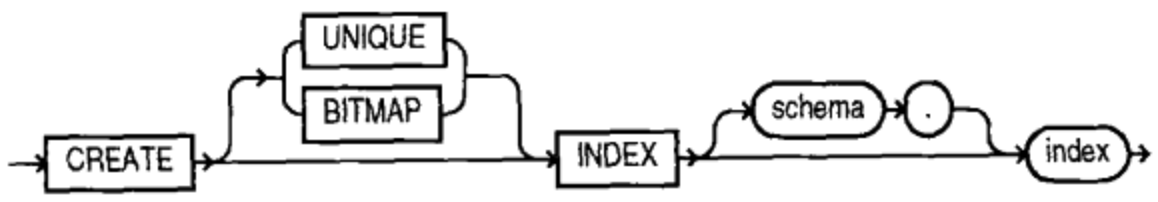
函数可以使用存储在数据库外部的 C 库(请参阅 CREATE LIBRARY)。如果在函数内使用了 Java，则可以在 LANGUAGE 子句内提供一个 Java 声明。INVOKER_RIGHTS 子句可以用来指定是否以此函数的所有者(定义者)权限或以当前用户(调用者)的权限执行函数。

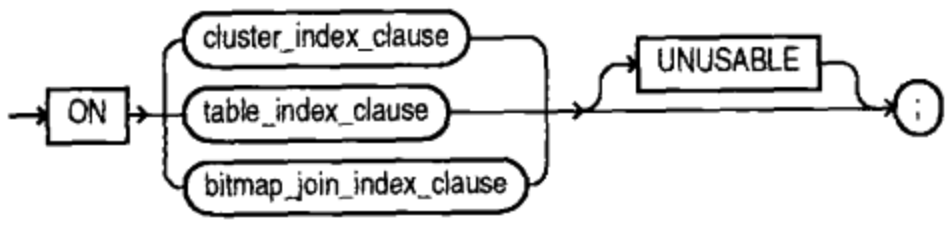
CREATE INDEX

参阅：ANALYZE、ALTER INDEX、DROP INDEX、INTEGRITY CONSTRAINT、STORAGE、第 17 章和第 46 章。

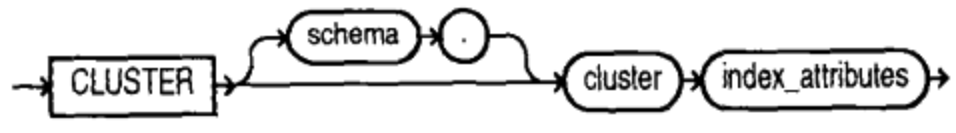
格式：

create_index::=

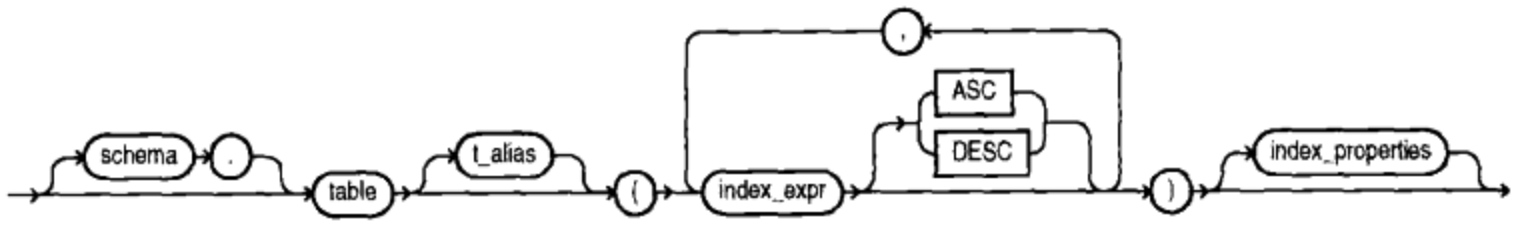




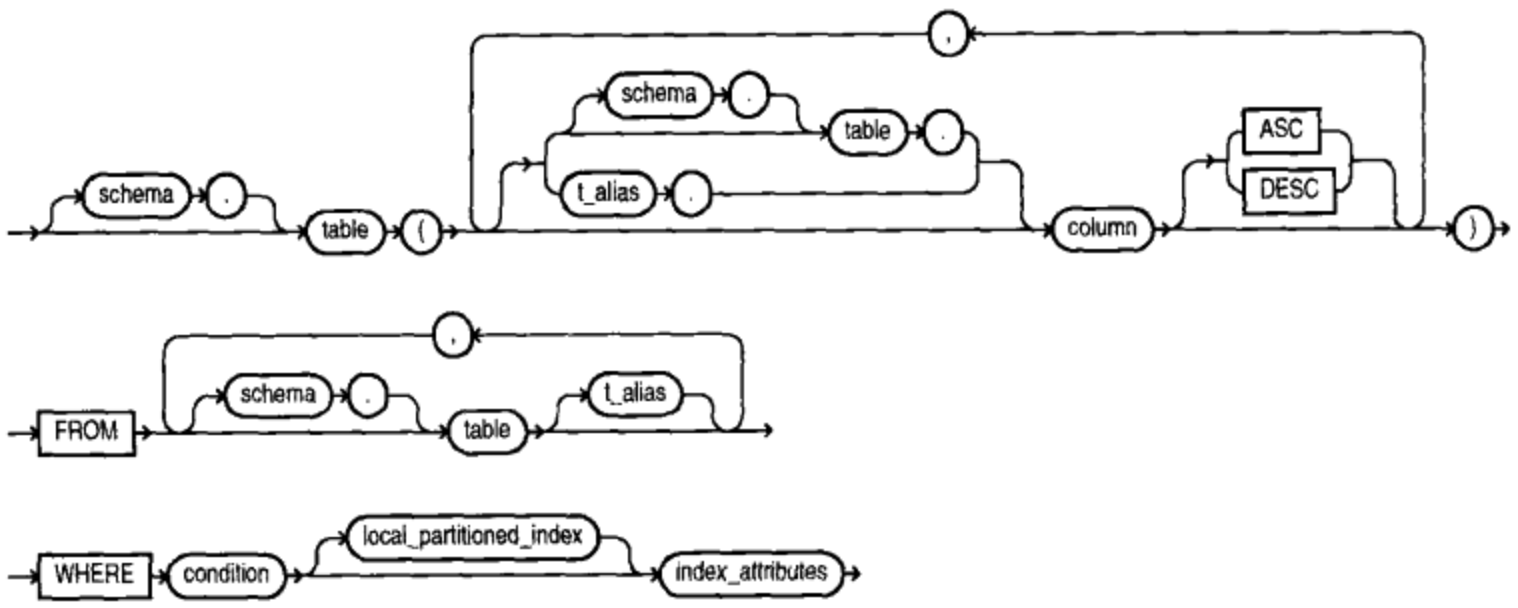
cluster_index_clause::=



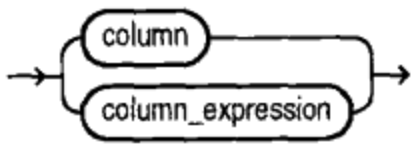
table_index_clause::=



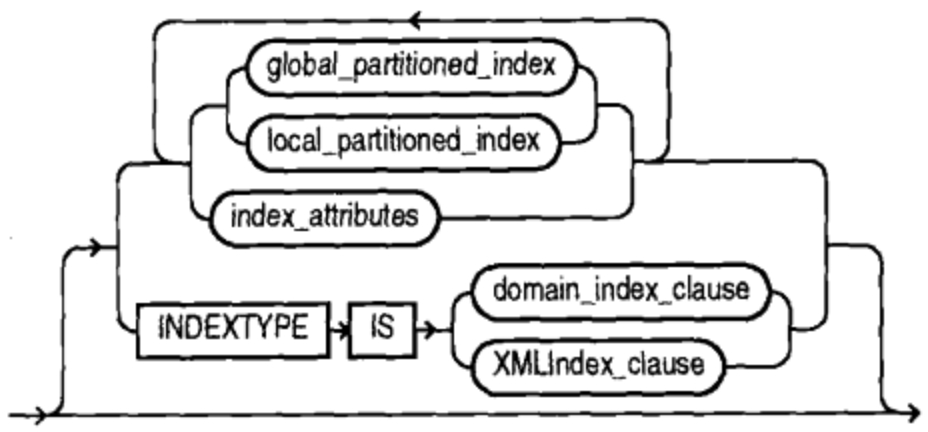
bitmap_join_index_clause::=



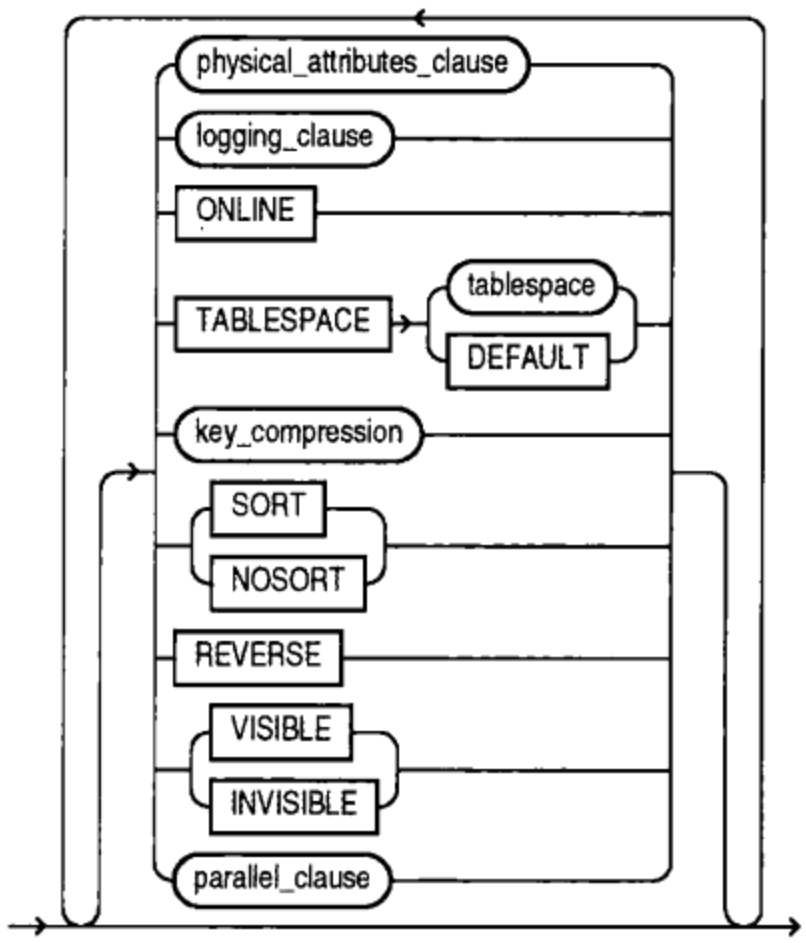
index_expr::=



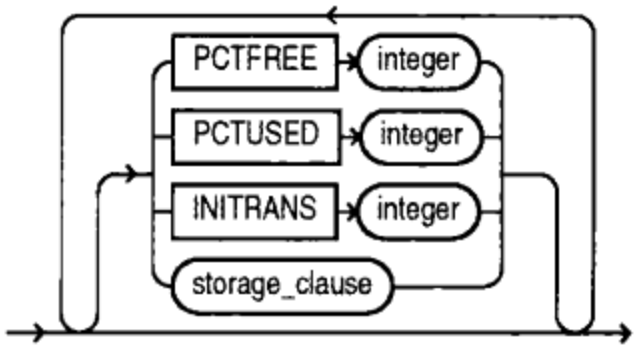
index_properties::=



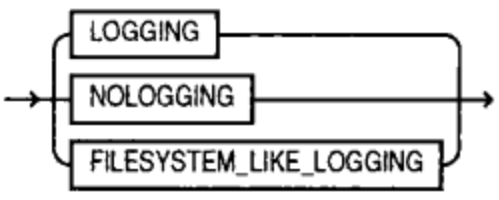
index_attributes::=



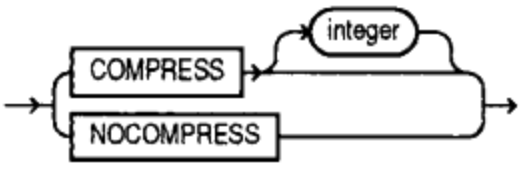
physical_attributes_clause::=



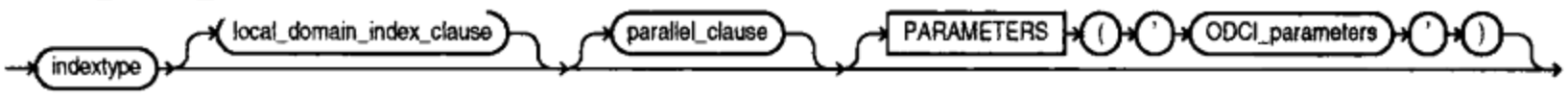
logging_clause::=



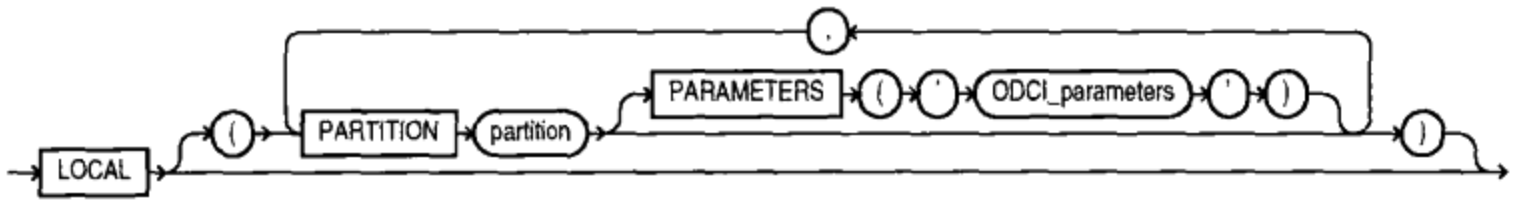
key_compression::=



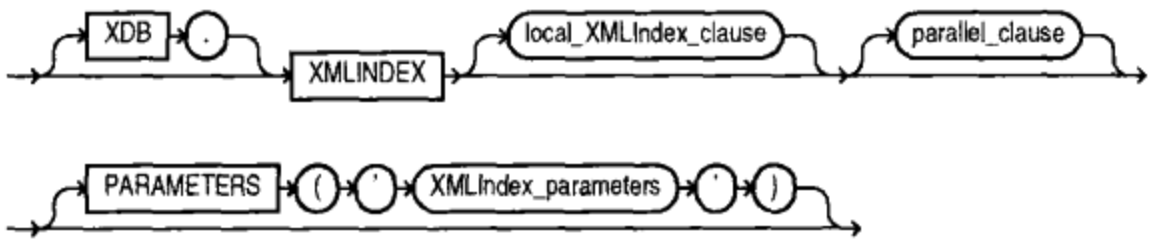
domain_index_clause::=



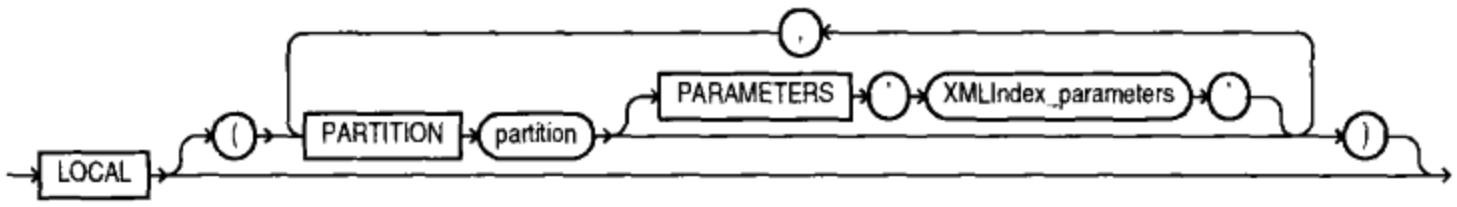
local_domain_index_clause::=



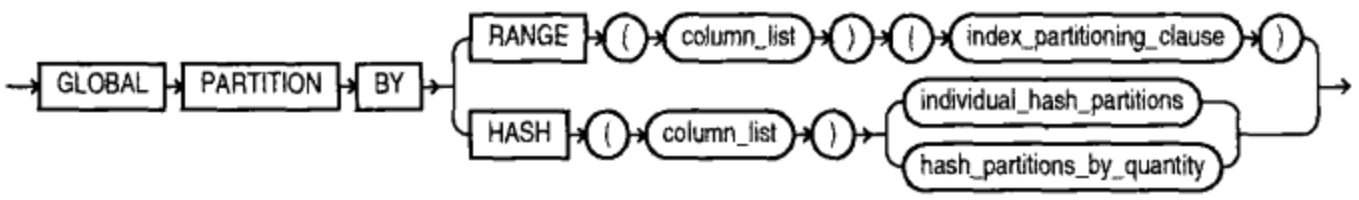
XMLIndex_clause::=



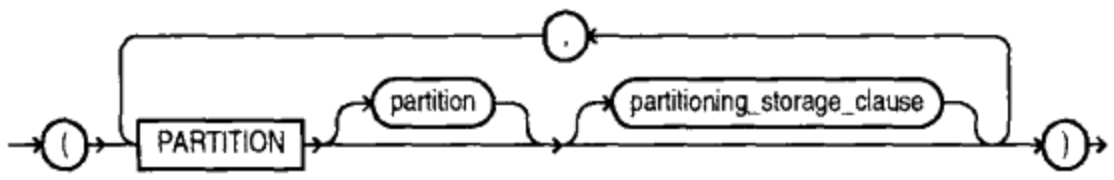
local_XMLIndex_clause::=



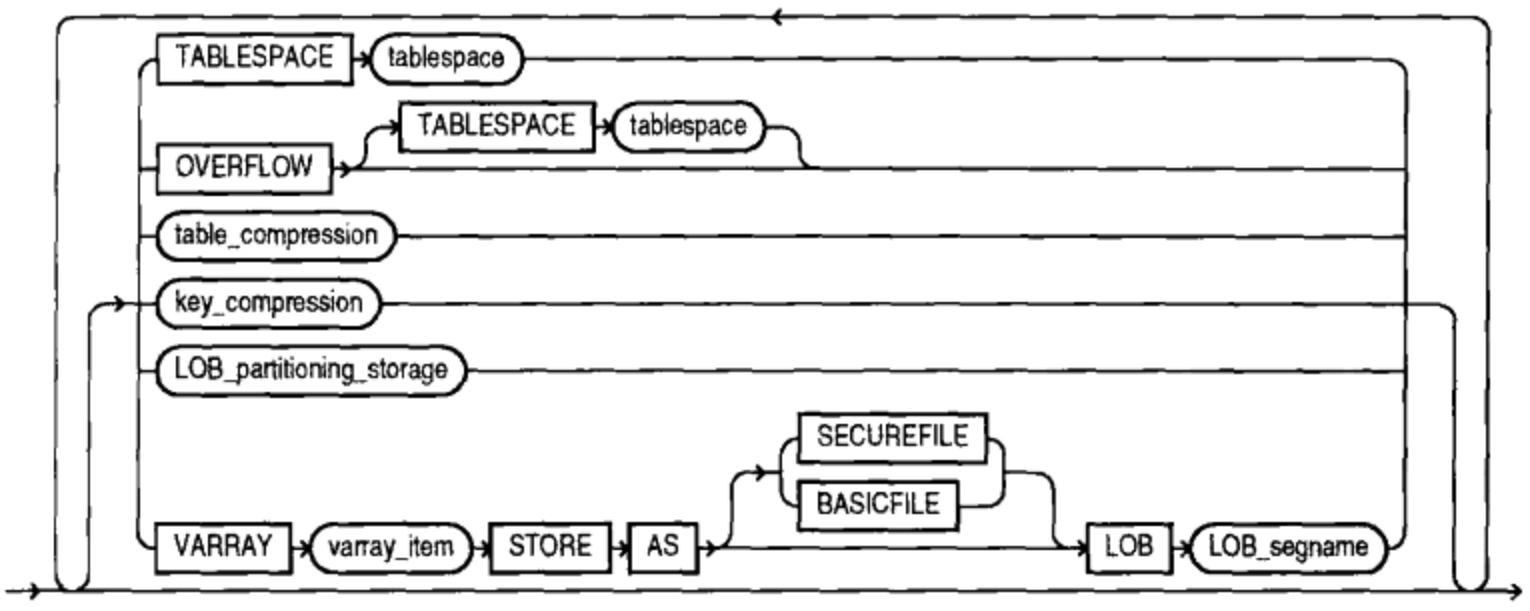
global_partitioned_index::=



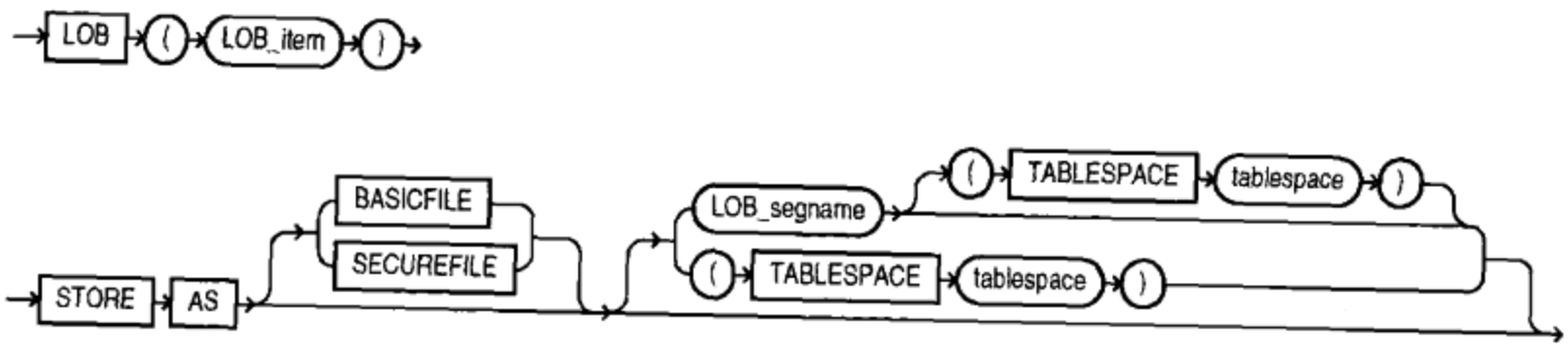
individual_hash_partitions::=



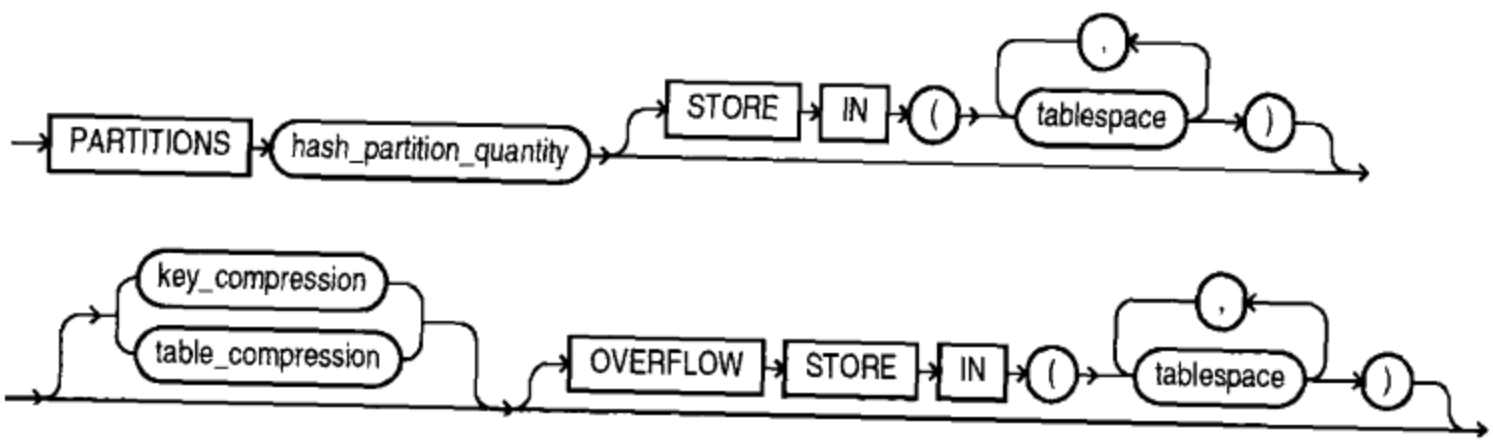
partitioning_storage_clause::=



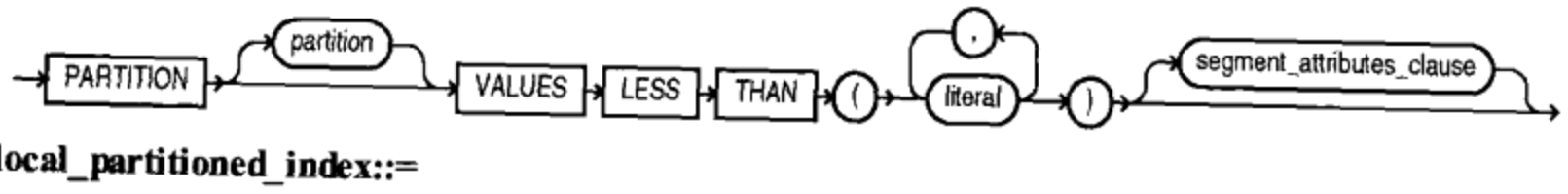
LOB_partitioning_storage::=



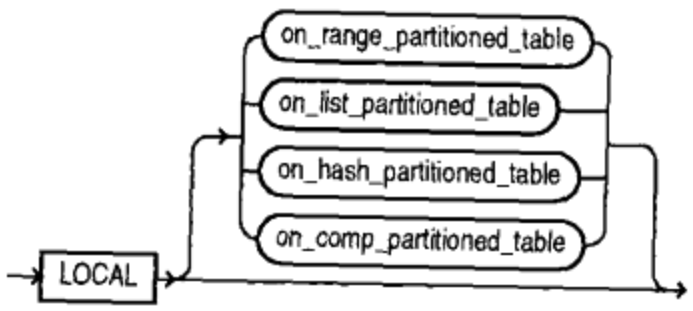
hash_partitions_by_quantity::=



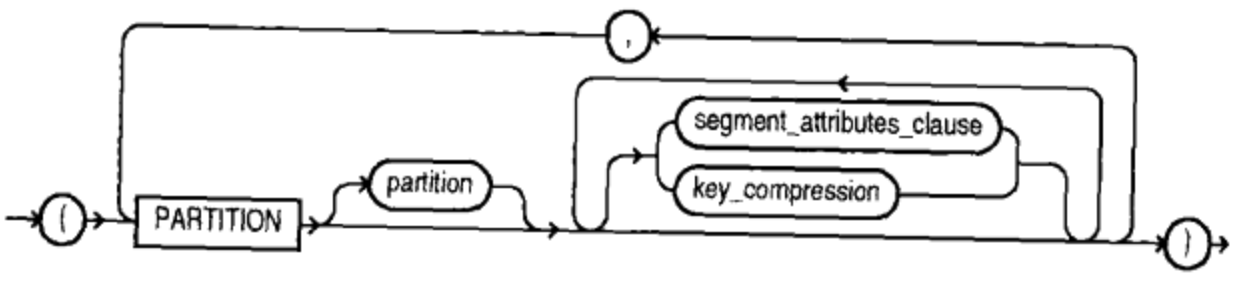
index_partitioning_clause::=



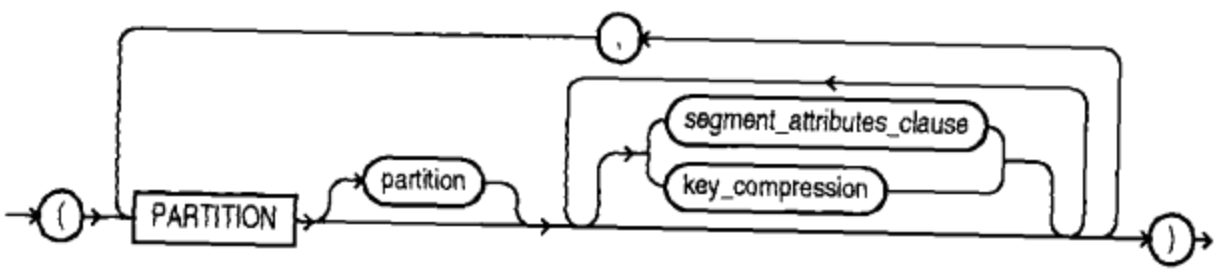
local_partitioned_index::=



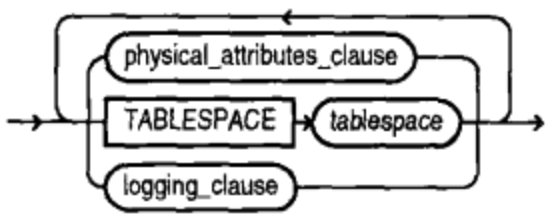
on_range_partitioned_table::=



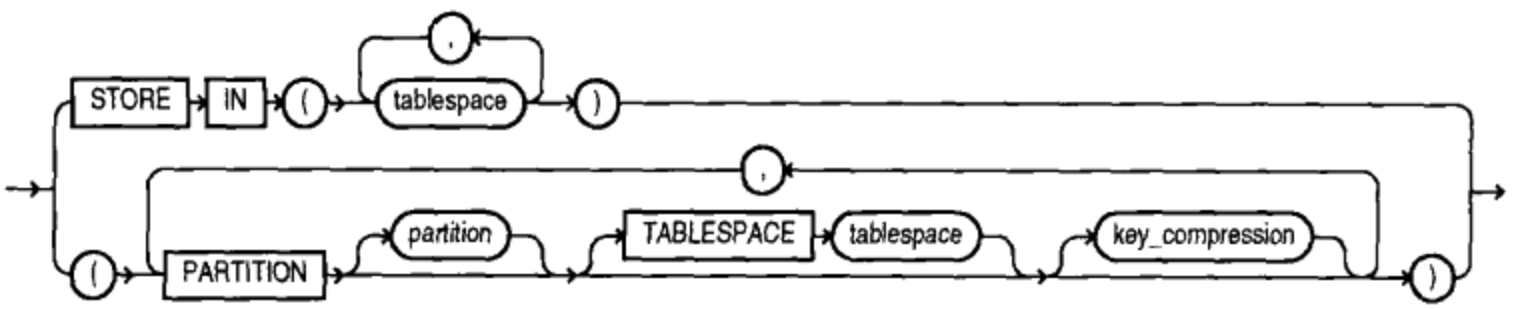
on_list_partitioned_table::=



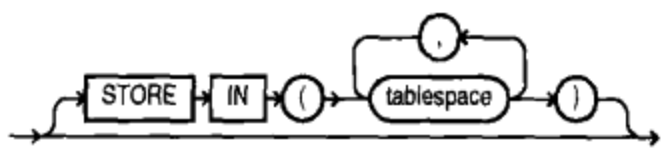
segment_attributes_clause::=

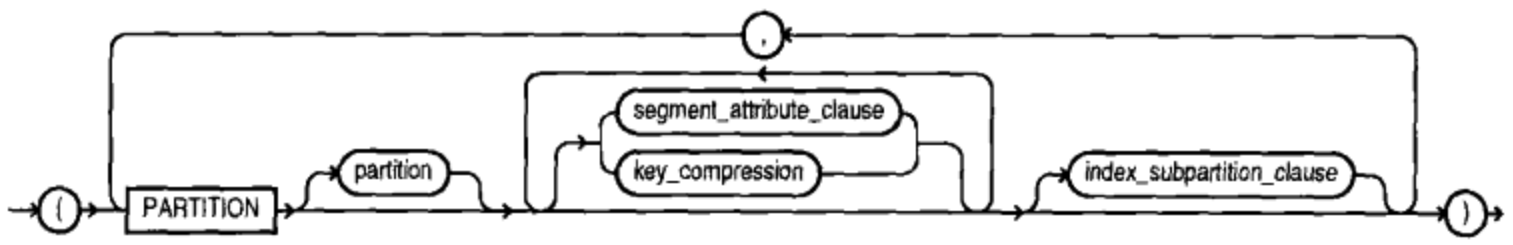


on_hash_partitioned_table::=

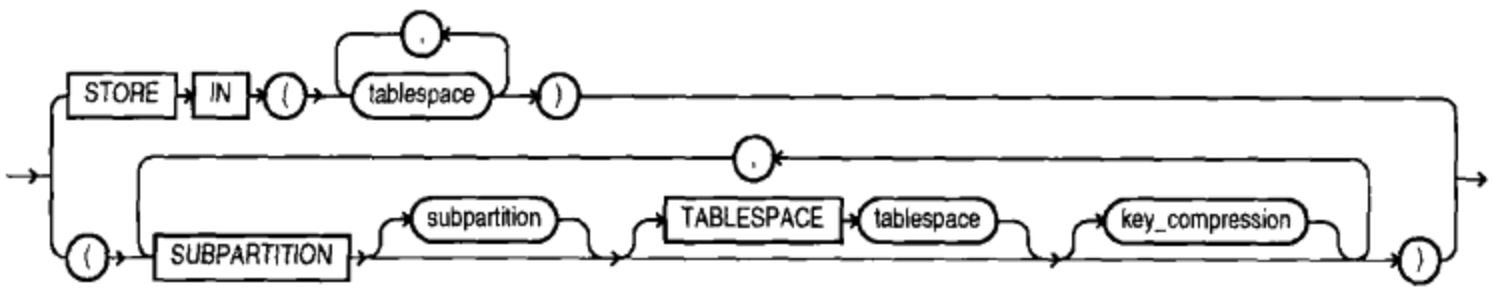


on_comp_partitioned_table::=

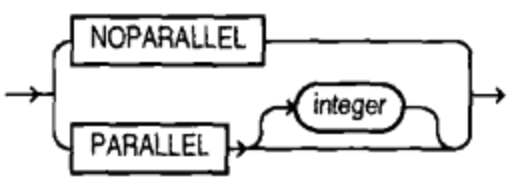




index_subpartition_clause::=



parallel_clause::=



描述：index 是分配给此索引的名称。一般最好让它反映被检索的表和列。table 和 column 是为其创建索引的列和表。唯一索引保证每一个索引行的索引列中的值为唯一的。可在列上使用 UNIQUE 约束自动创建唯一索引。指定多列将创建一个复合索引。ASC 和 DESC 表示升序和降序。CLUSTER 是为群集创建索引的群集键名。群集的键必须在被访问的关联表上创建了索引。PCTFREE 是索引中为新项和更新的项保留的空间百分比，最小为 0。

TABLESPACE 是给其分配索引的表空间名。物理属性部分包含在 STORAGE 下描述的子句。NOSORT 是一个选项，它的主要作用是在当且仅当被索引的列中的值已经为升序时，减少创建索引的时间。即使以后这些值不按升序排列，该选项也不会有坏作用，NOSORT 仅在按照顺序创建索引时，才起作用。如果行是无序的，则 CREATE INDEX 将返回错误消息，但不会破坏任何内容，而且允许不用 NOSORT 选项重新运行。

PARALLEL 与 DEGREE 和 INSTANCES 一同指定索引的并行特性。DEGREE 指定用来创建索引的查询服务器的数目。INSTANCES 指定怎样在并行查询处理的实时应用群集的实例之间拆分索引。整数 n 指定拆分索引的可用实例的数目。

要创建索引，必须拥有要索引的表，或者拥有该表的 INDEX 权限，或者拥有 CREATE ANY INDEX 系统权限。要创建基于函数的索引，必须拥有 QUERY REWRITE 权限。

BITMAP 创建位图索引，位图索引对于不同值很少的列很有用。PARTITION 子句在分区表上创建索引。位图连接索引在单个索引中为多个表存储索引键值(通常用于数据仓库应用程序)。位图连接索引使用 FROM 子句和 WHERE 子句指定相关的表和连接条件。

REVERSE 以倒序方式存储索引值的字节。不能颠倒位图索引。

COMPRESS 通过压缩非唯一、非分区索引来节省存储空间。在数据检索中，显示出的数据就像未曾压缩过一样。可以关闭索引压缩，也可以使用 NOCOMPRESS 子句重新创建索引。

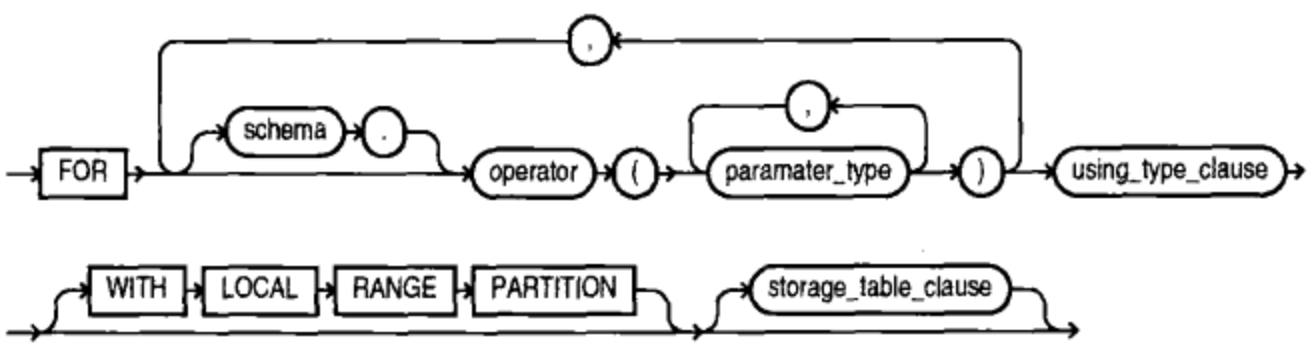
COMPUTE STATISTICS 在创建索引时分析索引数据。

在 Oracle Database 11g 中，可以创建本地分区域索引，可以为 XML 数据创建 XMLIndex 索引。可以使用 index_attributes 子句创建优化程序不可见的索引。

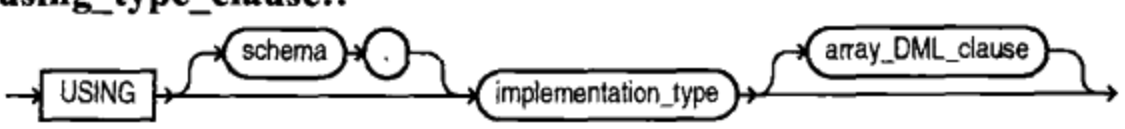
CREATE INDEXTYPE

格式：

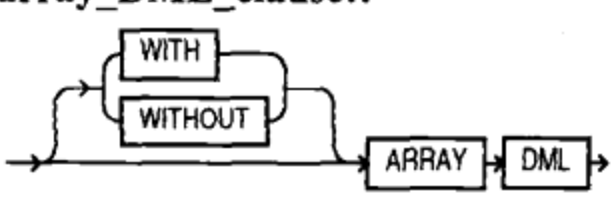
create_indextype::=

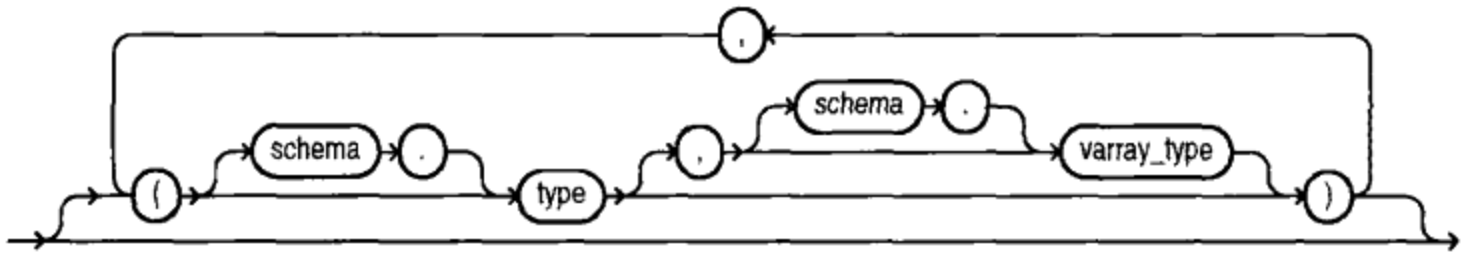


using_type_clause::=



array_DML_clause::=





storage_table_clause::=

描述：CREATE INDEXTYPE 指定域索引所使用的例程。要创建索引类型，必须拥有 CREATE INDEXTYPE 系统权限。要在另一个用户的模式中创建索引类型，必须拥有 CREATE ANY INDEXTYPE 系统权限。

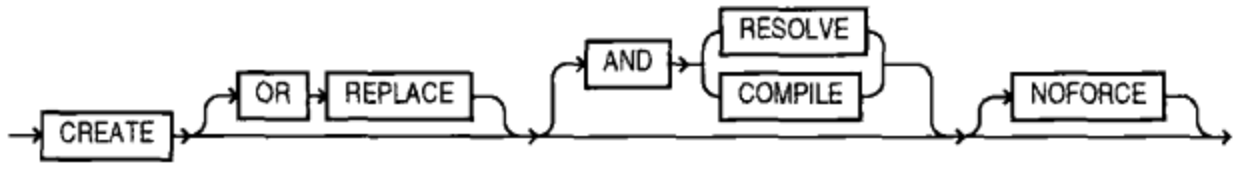
在 Oracle Database 11g 中，可以指定域索引为范围-分区索引，也可以指定这些索引的存储表和分区维护操作由数据库管理。

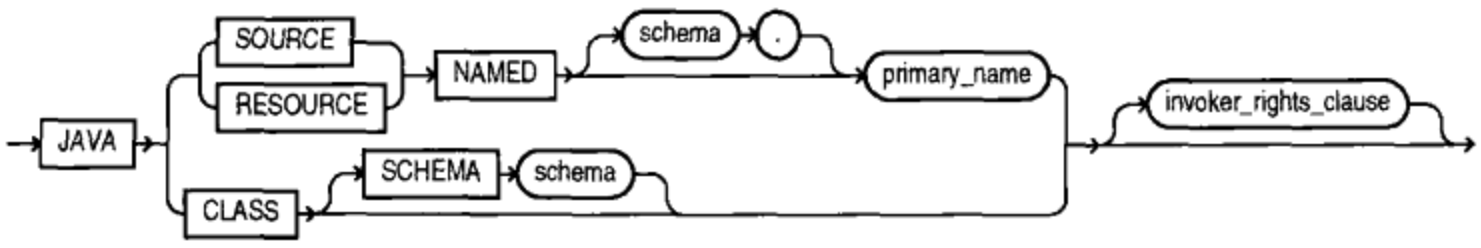
CREATE JAVA

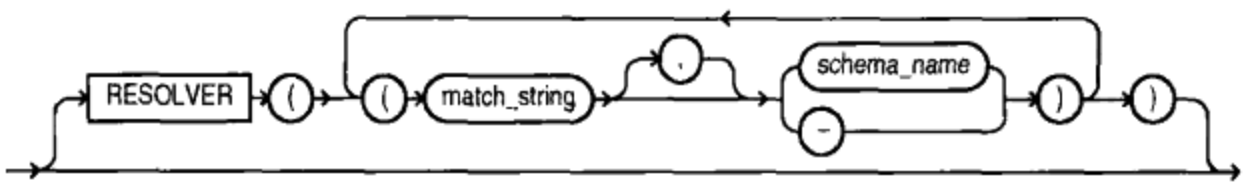
参阅：ALTER JAVA、DROP JAVA、第 43 章和第 44 章。

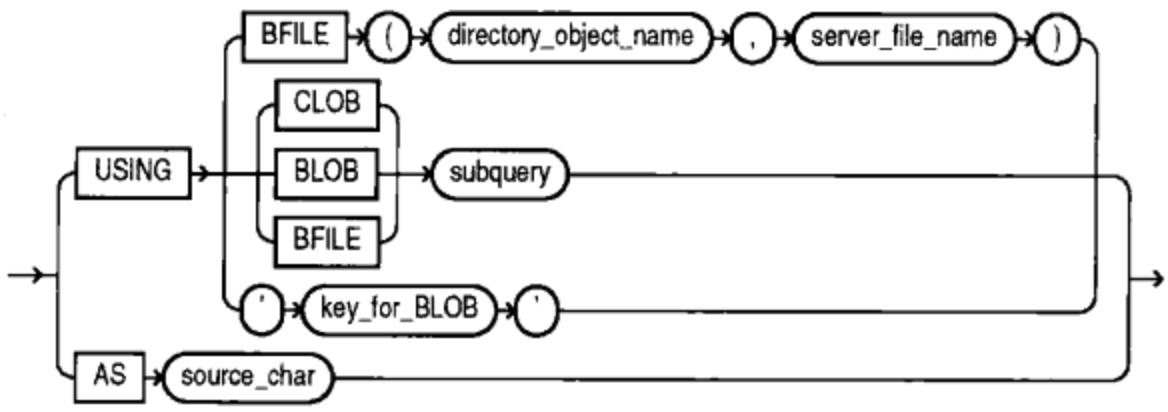
格式：

create_java::=

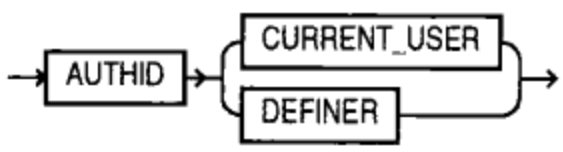








invoker_rights_clause::=



描述：CREATE JAVA 创建 JAVA 源、类或资源。必须拥有 CREATE PROCEDURE 系统权限或(为了在另一个用户的模式中创建对象)CREATE ANY PROCEDURE 系统权限。要在另一个用户的模式中替换这样一个模式对象，必须拥有 ALTER ANY PROCEDURE 系统权限。

JAVA SOURCE、JAVA CLASS 和 JAVA RESOURCE 子句分别加载源、类和资源。

AUTHID 子句指定此函数是使用函数所有者的权限还是当前用户的权限来执行。

示例：以下命令创建一个 Java 源：

```
create java source named "Hello" as
    public class Hello (
       public static String hello() (
          return "Hello World"; ) );
```

CREATE LIBRARY

参阅：CREATE FUNCTION、CREATE PACKAGE BODY、CREATE PROCEDURE 和第 35 章。

格式：

```
CREATE [OR REPLACE] LIBRARY [ schema .] libname
{ IS | AS } 'filespec' [AGENT 'agent_ dblink'];
```

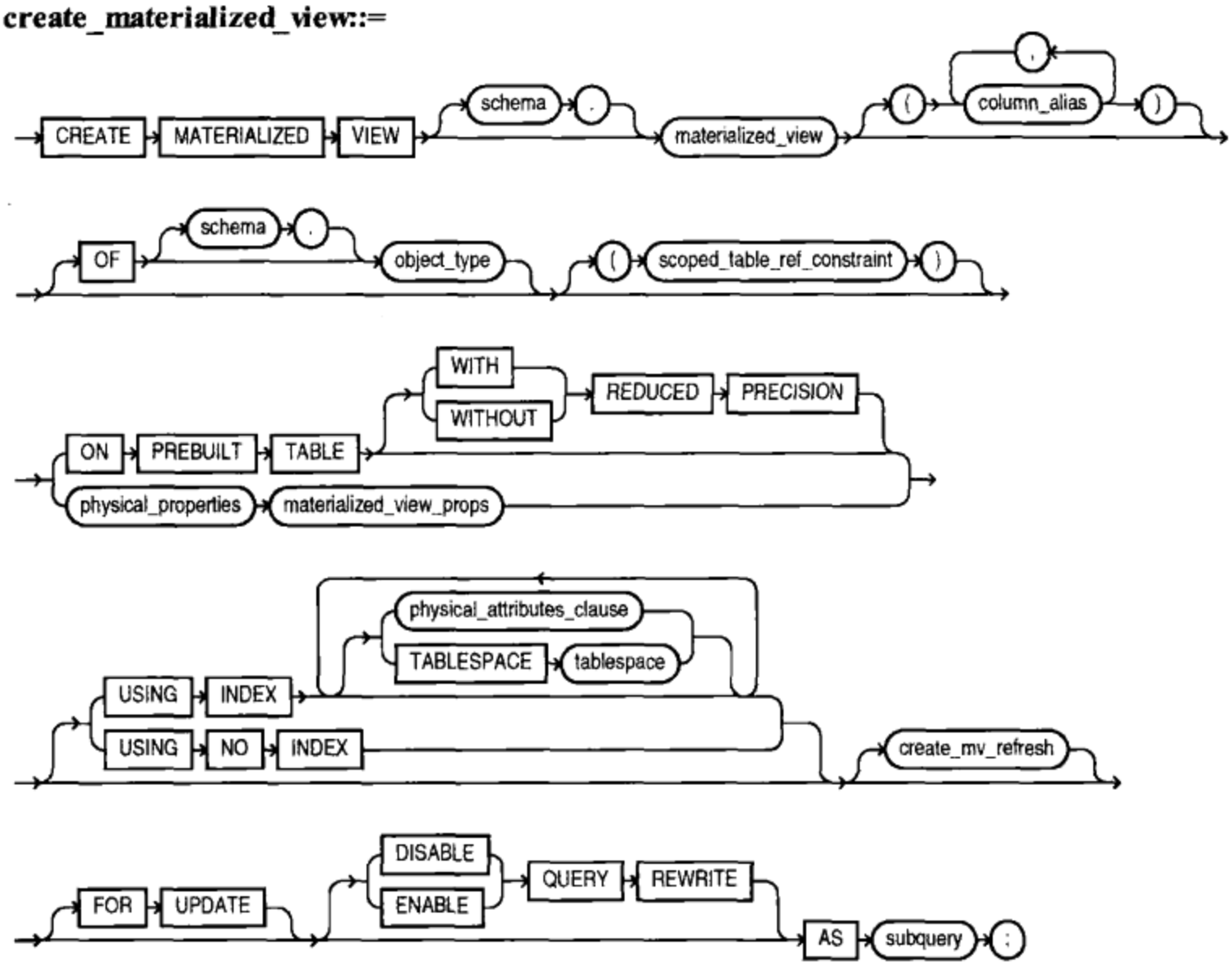
描述：CREATE LIBRARY 创建一个库对象，允许引用一个操作系统共享的库，SQL 和 PL/SQL 可以从该库中调用外部的 3GL 函数和过程。要使用存储在库中的过程和函数，必须拥有该库的 EXECUTE 权限。如果想从数据库链接(而不是服务器)运行外部过程，则应指定 AGENT 子句。Oracle 将使用 agent_dblink 指定的数据库链接运行外部过程。如果省略此子句，则默认的服务器代理(extproc)将运行外部过程。

CREATE MATERIALIZED VIEW

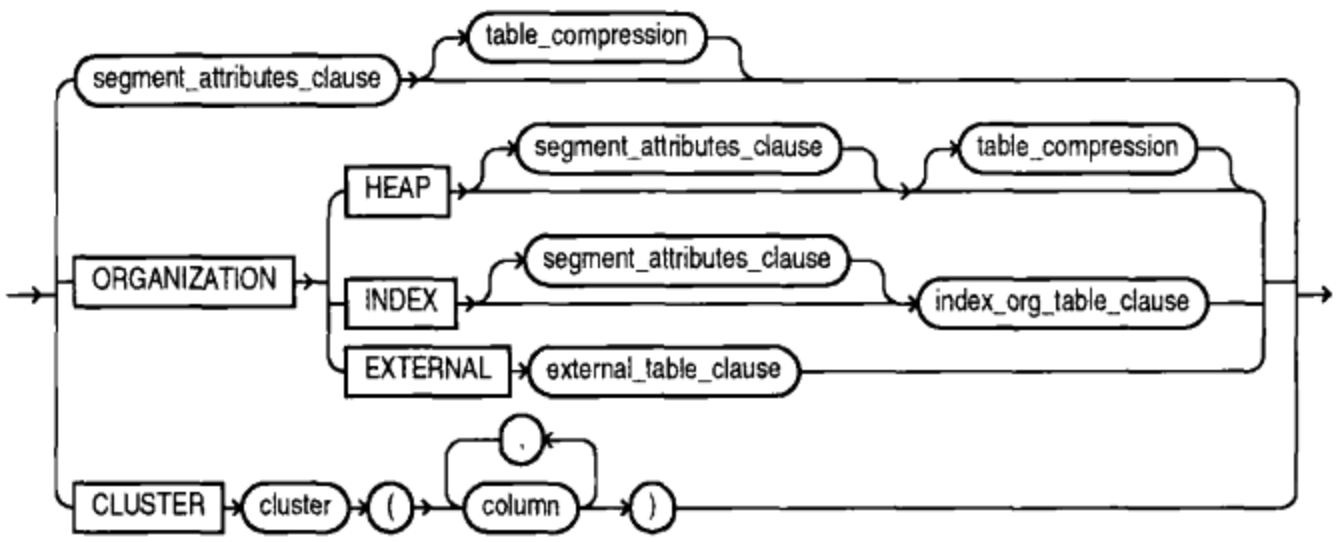
参阅：ALTER MATERIALIZED VIEW、CREATE MATERIALIZED VIEW LOG、DROP MATERIALIZED VIEW、STORAGE 和第 26 章。

格式：

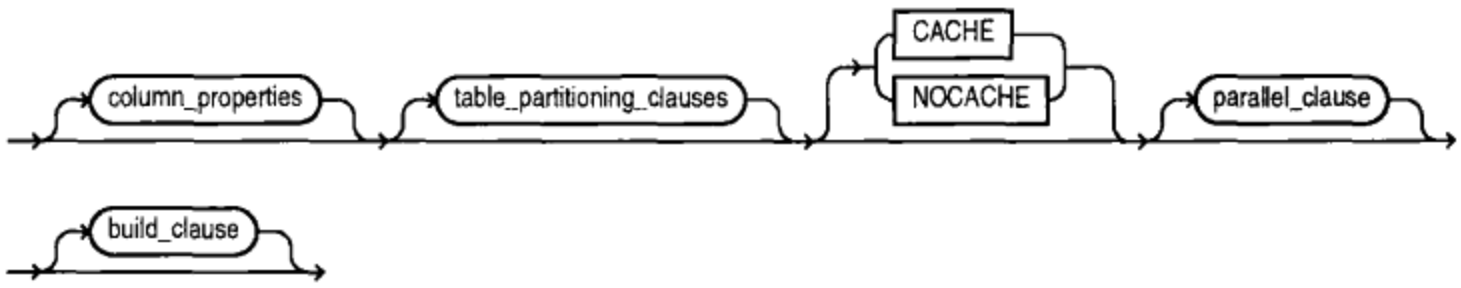
create_materialized_view::=



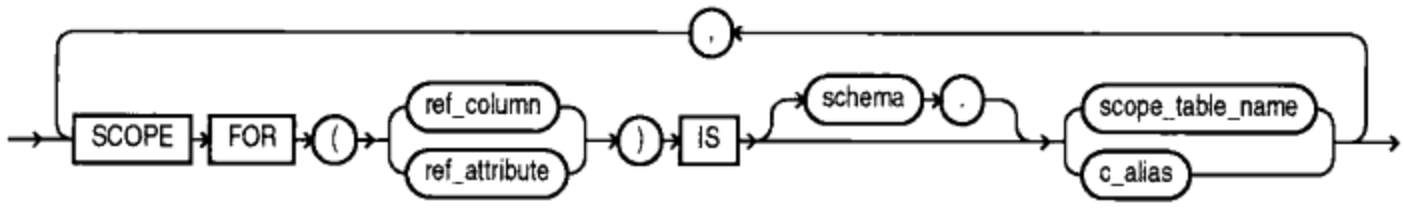
physical_properties::=



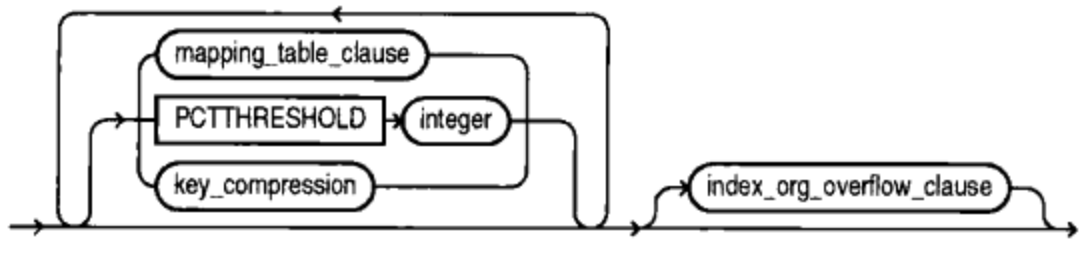
materialized_view_props::=



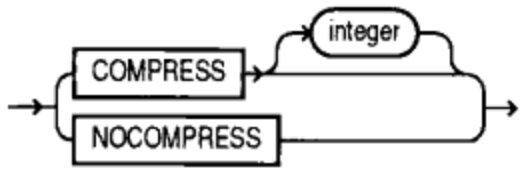
scoped_table_ref_constraint::=



index_org_table_clause::=

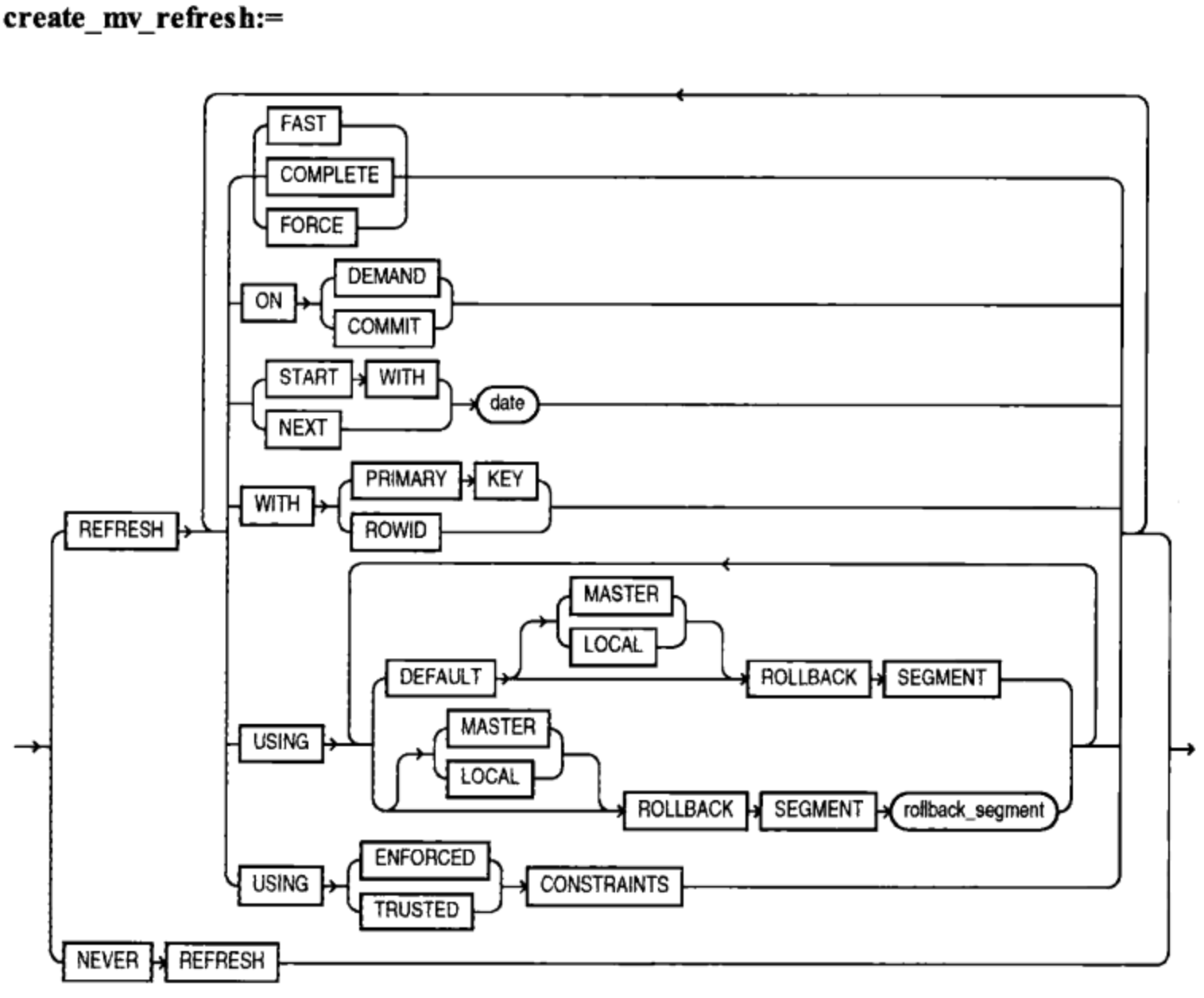


key_compression::=

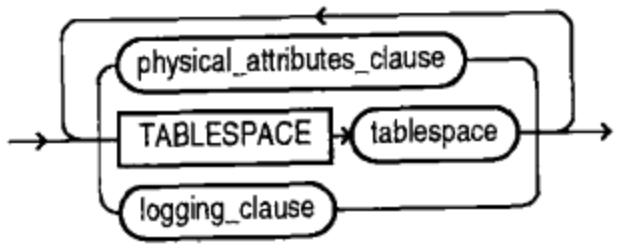


index_org_overflow_clause::=

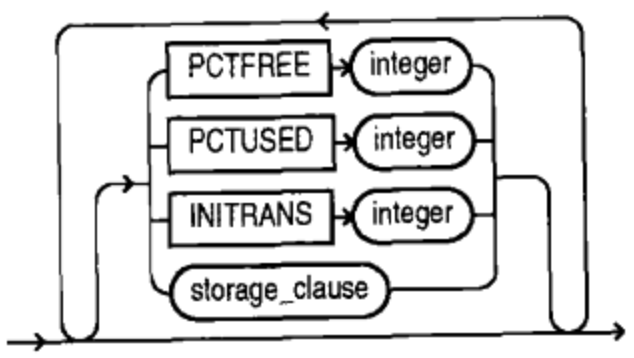
create_mv_refresh::=



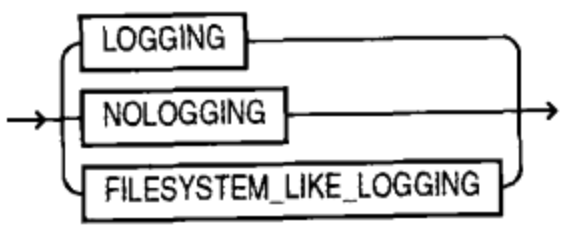
segment_attributes_clause::=



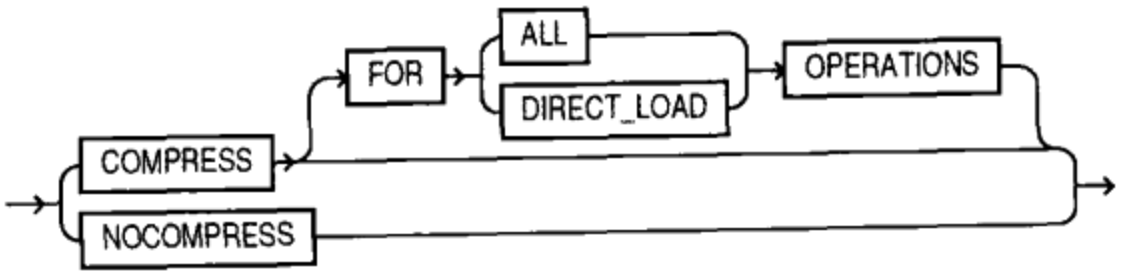
physical_attributes_clause::=



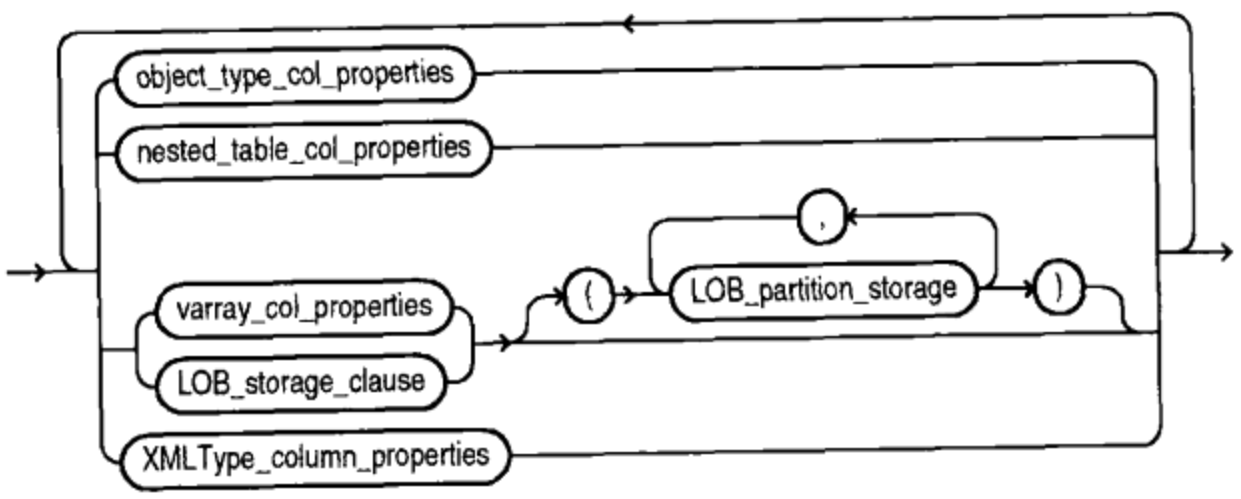
logging_clause::=



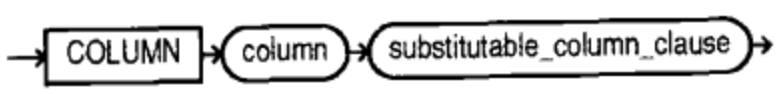
table_compression::=



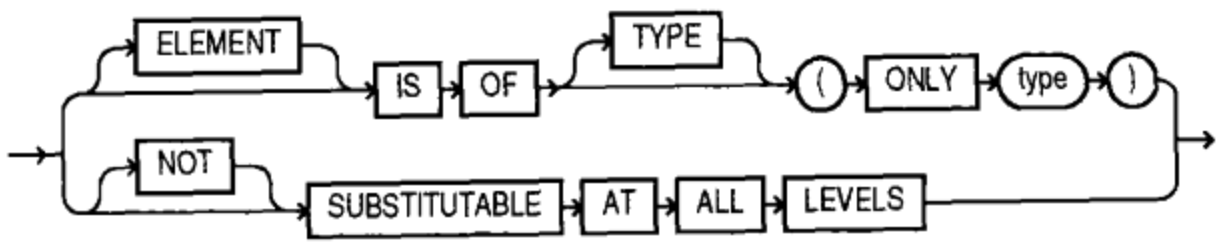
column_properties::=



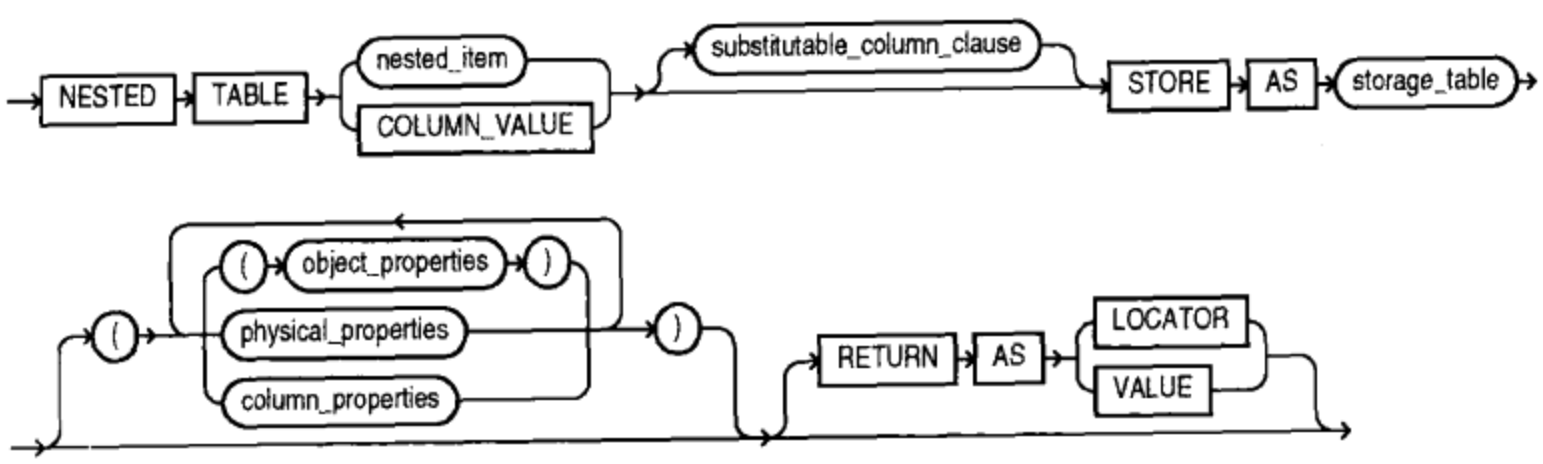
object_type_col_properties::=



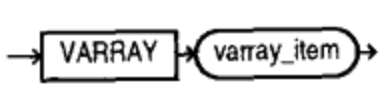
substitutable_column_clause::=

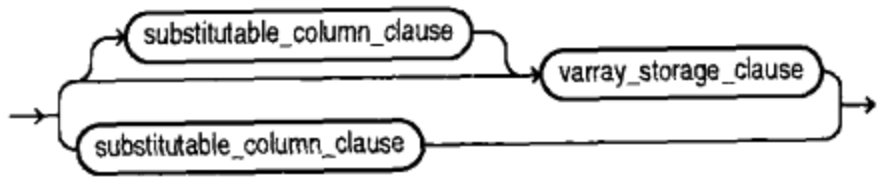


nested_table_col_properties::=

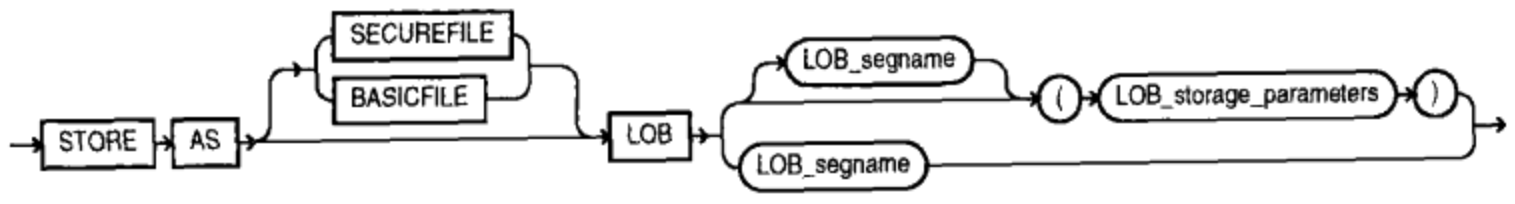


varray_col_properties::=

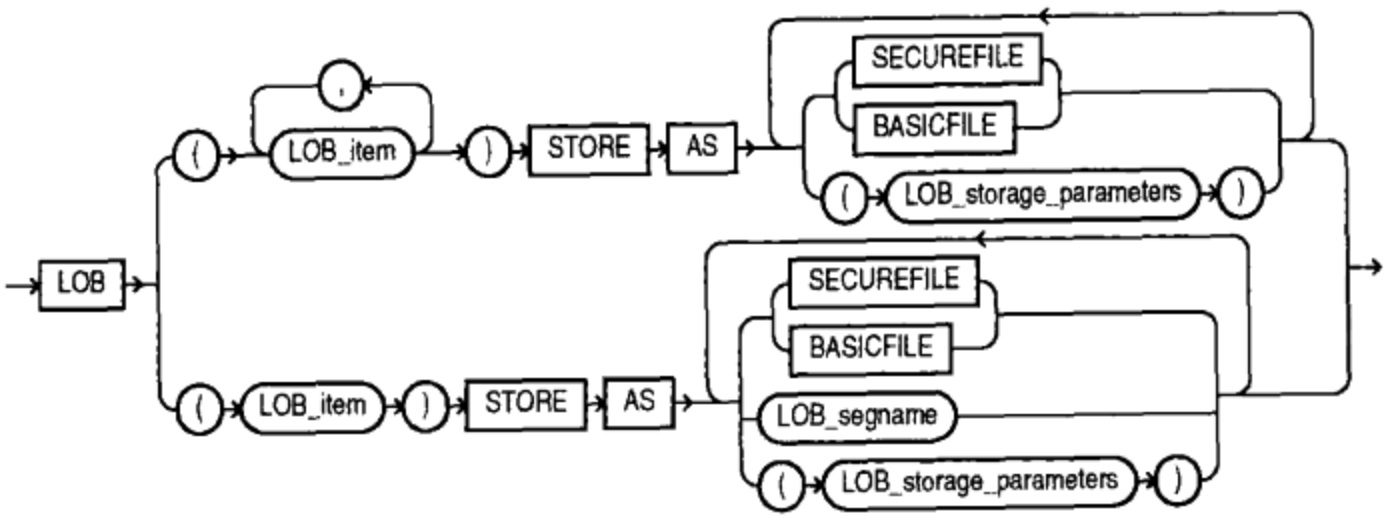




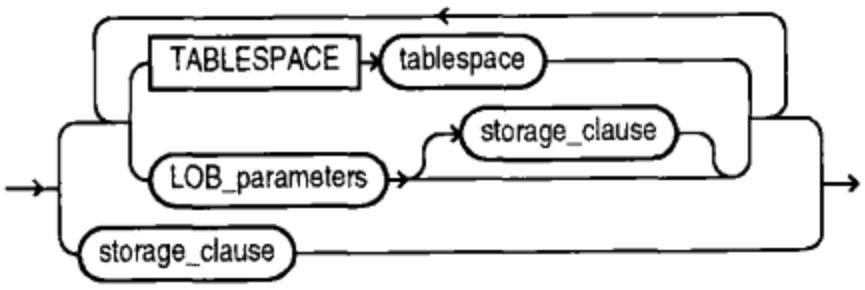
varray_storage_clause::=



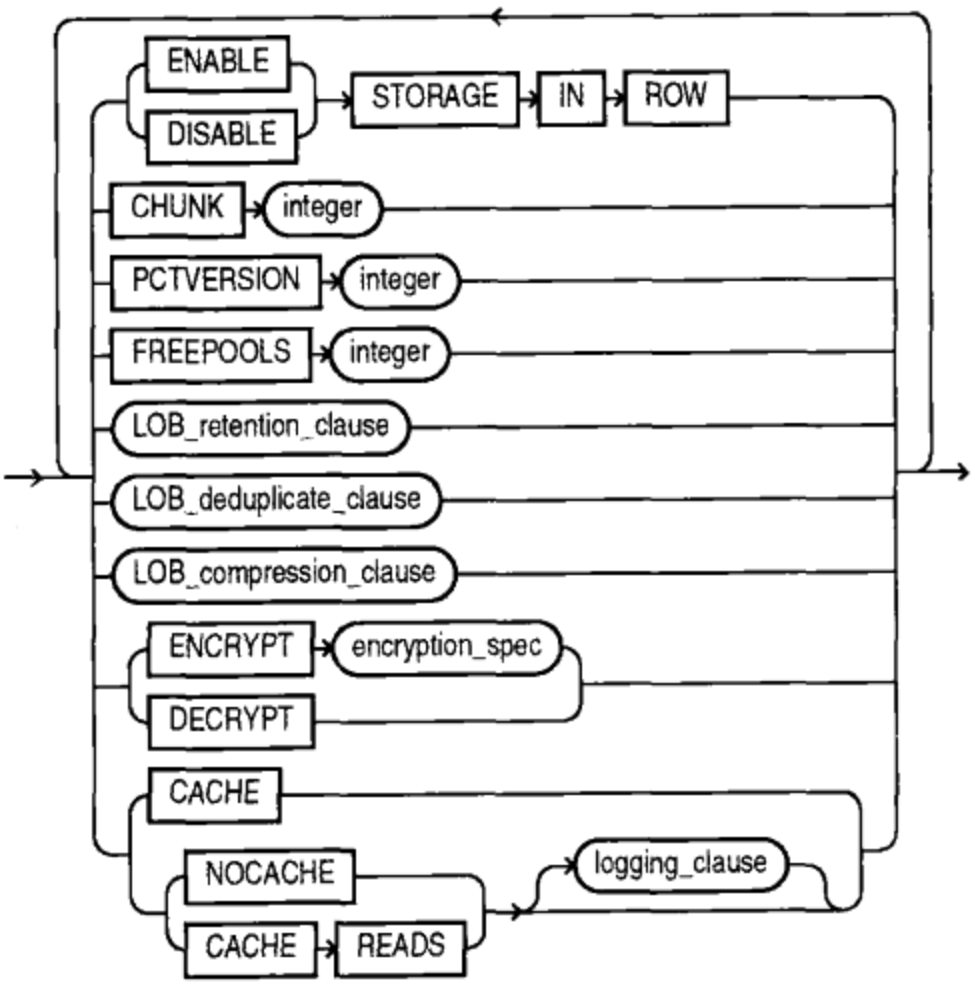
LOB_storage_clause::=



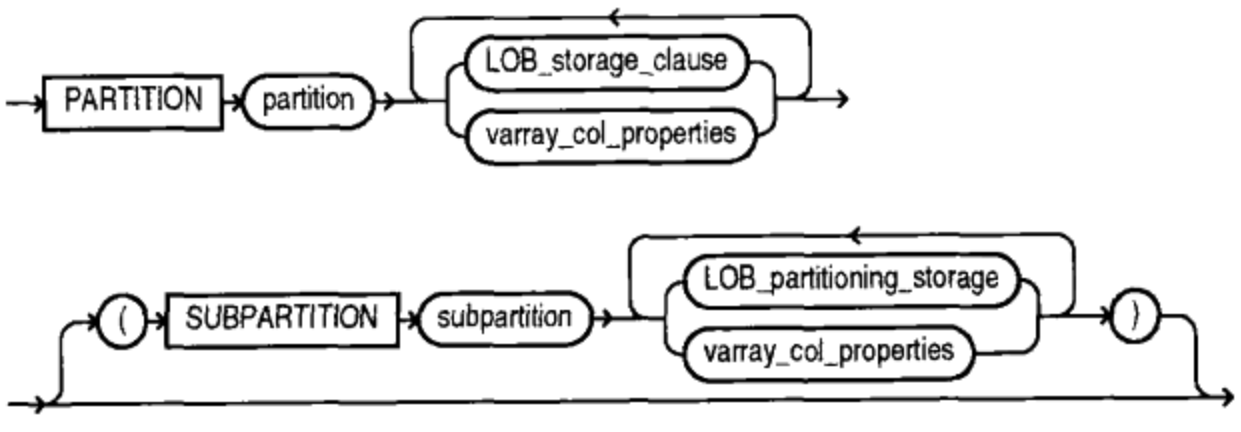
LOB_storage_parameters::=



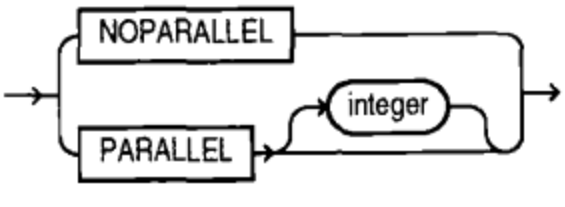
LOB_parameters::=



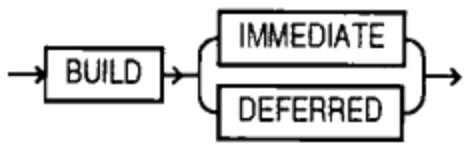
LOB_partition_storage::=



parallel_clause::=



build_clause::=



描述：为向后兼容，支持关键字 SNAPSHOT，以替换 MATERIALIZED VIEW。

CREATE MATERIALIZED VIEW 创建一个物化视图，这是保存查询结果的表(通常在一个或多个表上，这些表称为主表)。主表可以位于本地数据库，或者位于远程数据库中。可使用 REFRESH 子句使数据在一定的时间间隔内进行刷新。

要给查询重写启用一个物化视图，必须拥有 QUERY REWRITE 系统权限。如果物化视图基于其他模式中的对象，则必须拥有 GLOBAL QUERY REWRITE 权限。

物化视图不能包含 LONG 列或引用 SYS 拥有的任何对象。

FAST 刷新使用与主表关联的物化视图日志刷新此物化视图。COMPLETE 刷新重新执行的查询。FORCE 刷新让 Oracle 在 FAST 刷新或 COMPLETE 刷新之间进行选择。Oracle 首先在 START WITH date 上刷新物化视图。如果给出一个 NEXT date，则 Oracle 将以 START WITH date 和 NEXT date 之间的日期差指定的时间间隔刷新物化视图。

简单物化视图使用简单的查询从单个主表中选择数据。复杂的物化视图在查询中使用 GROUP BY、CONNECT BY、子查询、连接或集合运算等选择数据。Oracle 只能对具有物化视图日志的简单物化视图进行 FAST 刷新。

要在自己的模式中创建一个物化视图，必须拥有 CREATE MATERIALIZED VIEW 系统权限。要在另一个用户的模式中创建一个物化视图，必须拥有 CREATE ANY MATERIALIZED VIEW 系统权限。

示例：

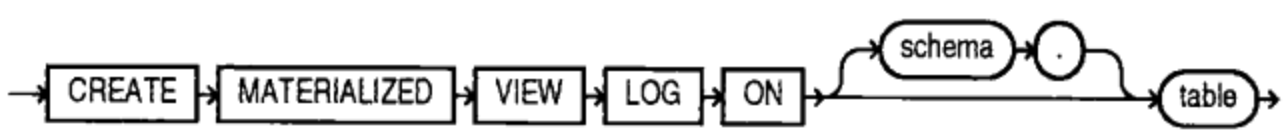
```
create materialized view LOCAL_BOOKSHELF
storage (initial 100K next 100K pctincrease 0)
tablespace USERS
refresh force
start with SysDate next SysDate+7
with primary key
as
select * from BOOKSHELF@REMOTE_CONNECT;
```

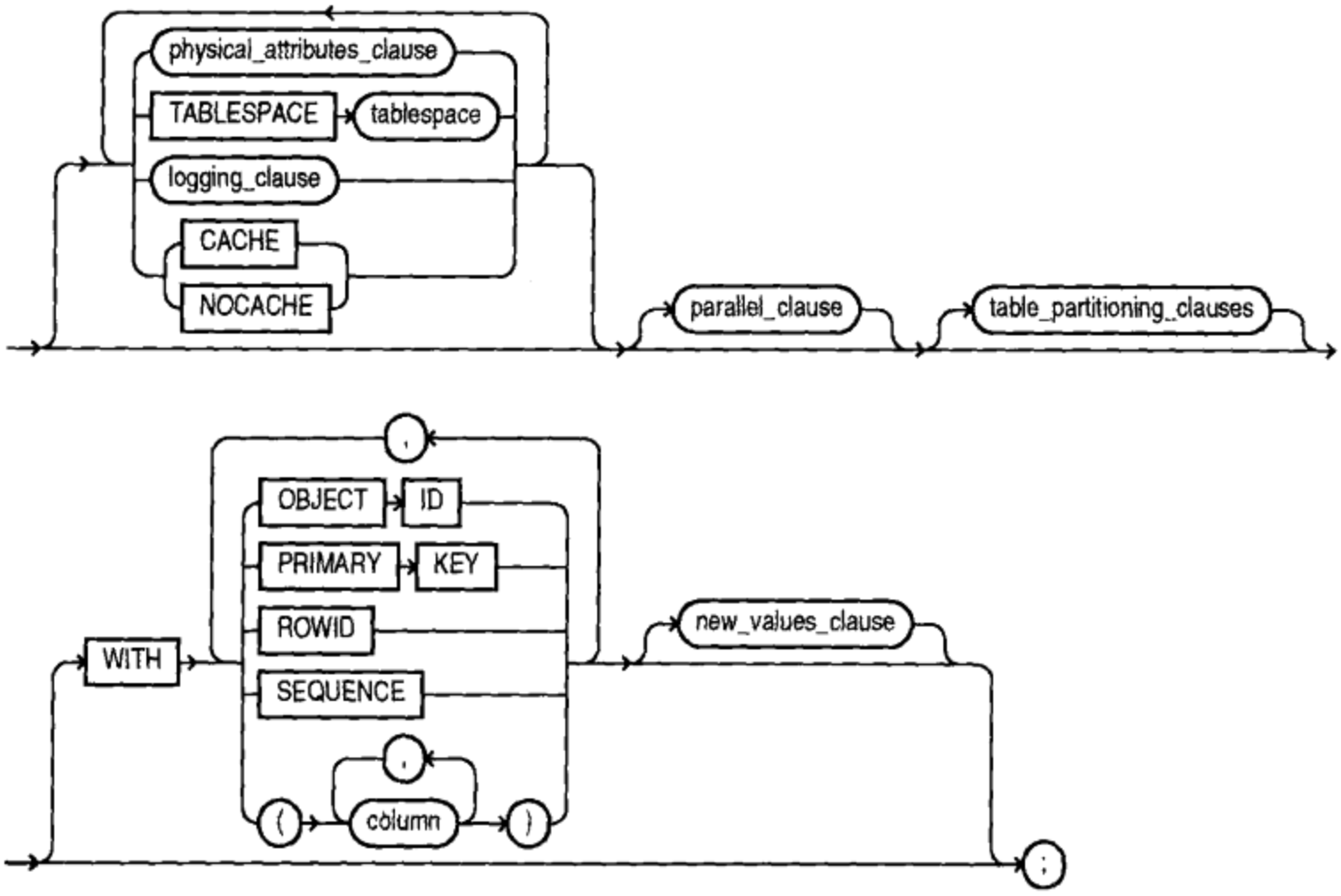
CREATE MATERIALIZED VIEW LOG

参阅：ALTER MATERIALIZED VIEW LOG、CREATE MATERIALIZED VIEW、DROP MATERIALIZED VIEW LOG、STORAGE 和第 26 章。

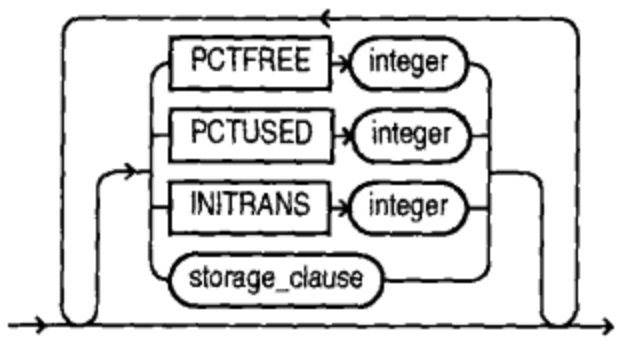
格式：

create_materialized_vw_log::=

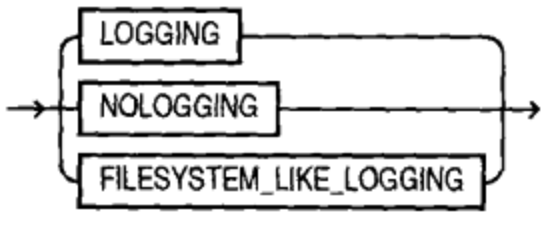




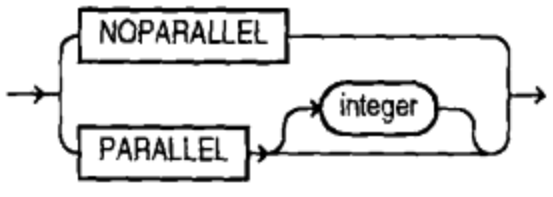
physical_attributes_clause::=



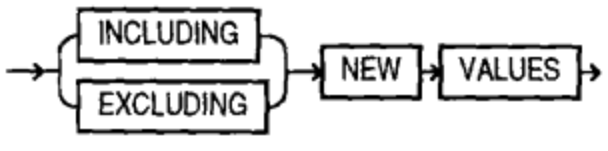
logging_clause::=



parallel_clause::=



new_values_clause::=



描述：为向后兼容，支持关键字 SNAPSHOT 替换 MATERIALIZED VIEW。

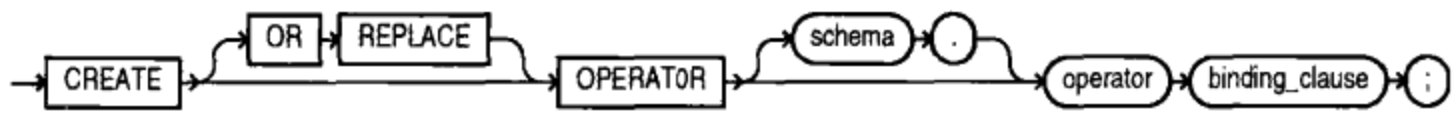
CREATE MATERIALIZED VIEW LOG 创建与物化视图的主表相关联的表，以跟踪主表的数据的修改。Oracle 使用物化视图日志快速(FAST)刷新主表的物化视图。存储选项指定表的存储。只有当有一个基于主表的简单物化视图时，Oracle 才记录对数据库的修改。

一个给定的主表可以只有一个日志，并且存储在与主表相同的模式中。要创建位于自己的主表上的物化视图日志，必须拥有 CREATE TABLE 系统权限。如果主表在另一个用户的模式中，则必须拥有 CREATE ANY TABLE 和 COMMENT ANY TABLE 系统权限以及主表的 SELECT 权限。

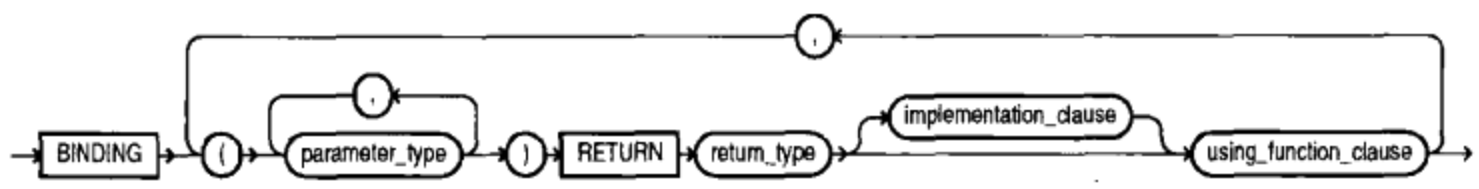
CREATE OPERATOR

格式：

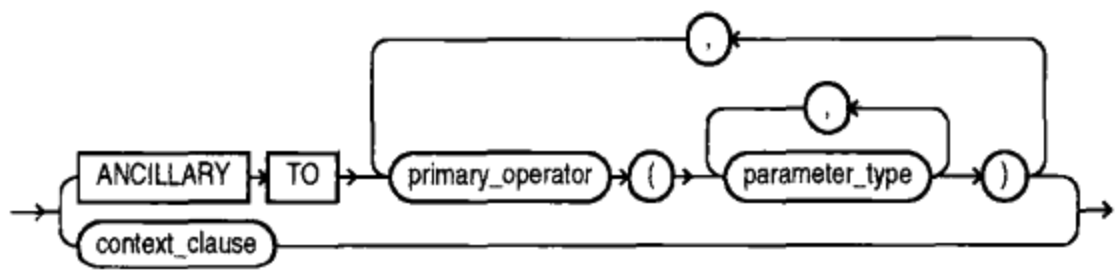
create_operator::=



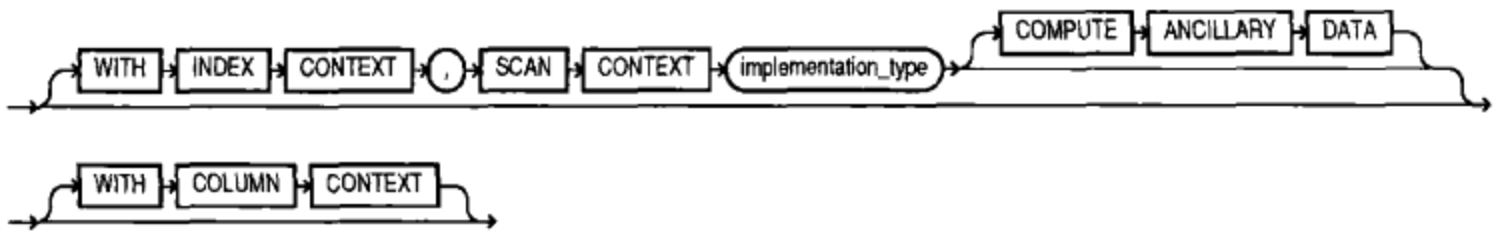
binding_clause::=



implementation_clause::=



context_clause::=



using_function_clause::=

描述：CREATE OPERATOR 创建一个新的运算符并定义其绑定的内容。可以在索引类型和 SQL 语句中引用该运算符。运算符可依次引用函数、程序包、类型以及用户定义的其他对象。

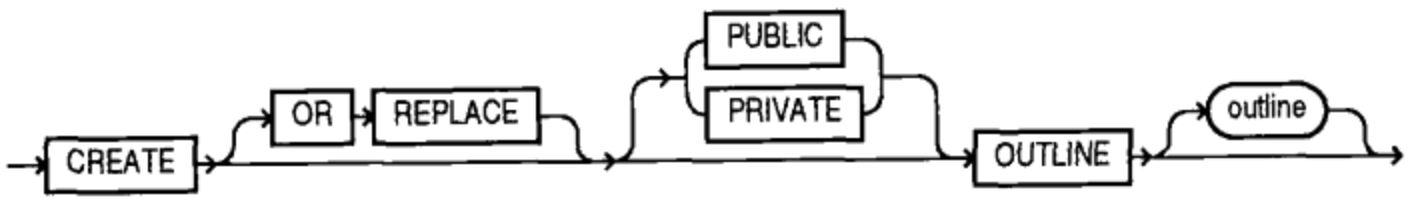
要创建一个运算符，必须拥有该运算符所引用的函数和运算符的 EXECUTE 权限，也必须拥有 CREATE OPERATOR 系统权限。如果在另一个用户的模式中创建该运算符，则必须拥有 CREATE ANY OPERATOR 系统权限。

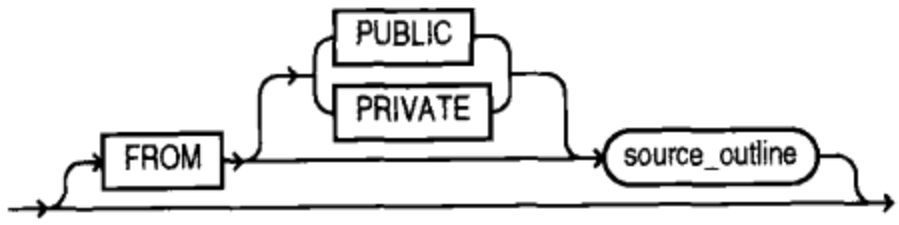
CREATE OUTLINE

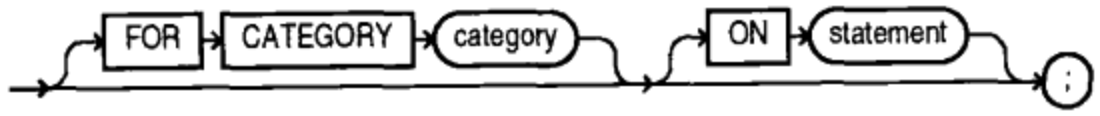
参阅：第 46 章。

格式：

create_outline::=







描述：CREATE OUTLINE 创建一个存储概要，它是一组用来创建关联查询的执行计划的提示。该查询后面的执行将使用这组相同的提示。可以将存储概要按类别分组。

要创建运算符，必须拥有 CREATE ANY OUTLINE 系统权限。

示例：

```
create outline TEST
    for category DEVELOPMENT
     on select AuthorName, COUNT(AuthorName)
  from BOOKSHELF_AUTHOR
group by AuthorName;
```

CREATE PACKAGE

参阅：ALTER PACKAGE、CREATE FUNCTION、CREATE PACKAGE BODY、CREATE PROCEDURE、CURSOR、DROP PACKAGE、EXCEPTION、RECORD、TABLE、VARIABLE、DECLARATION 和第 35 章。

格式：

```
CREATE [OR REPLACE] PACKAGE [ schema .] package
```

```
[AUTHID { CURRENT_USER | DEFINER }]
{ IS | AS } pl/sql_package_spec;
```

描述：CREATE PACKAGE 创建 PL/SQL 程序包的规范，即一组公有过程、函数、异常、变量、常量和游标的规范。添加 OR REPLACE 可以替换程序包规范(如果程序包规范存在的话)，这将使该程序包无效，并开始重新编译程序包体和依赖于程序包规范的任何对象。

将过程和函数打包可以使它们通过变量、常量和游标共享数据。它使用户可以立即具有作为角色的一部分授予整个集合的权限。Oracle 访问程序包中的所有元素比单独访问它们更有效。如果修改了程序包体，由于 Oracle 分别存储程序包体和程序包规范，因此不必重新编译使用该程序包的任何内容。

要创建一个程序包，必须拥有 CREATE PROCEDURE 系统权限。要在另一个用户的账户下创建程序包，必须拥有 CREATE ANY PROCEDURE 系统权限。

AUTHID 子句可以指定是用该程序包所有者的权限还是用当前运行程序包代码的用户的权限执行程序包代码。

CREATE PACKAGE BODY

参阅：ALTER PACKAGE、CREATE FUNCTION、CREATE LIBRARY、CREATE PACKAGE、CREATE PROCEDURE、CURSOR、DROP PACKAGE、EXCEPTION、RECORD、TABLE、VARIABLE DECLARATION 和第 35 章。

格式：

```
CREATE [OR REPLACE] PACKAGE BODY [ schema .] package
{ IS | AS } pl/sql_package_body;
```

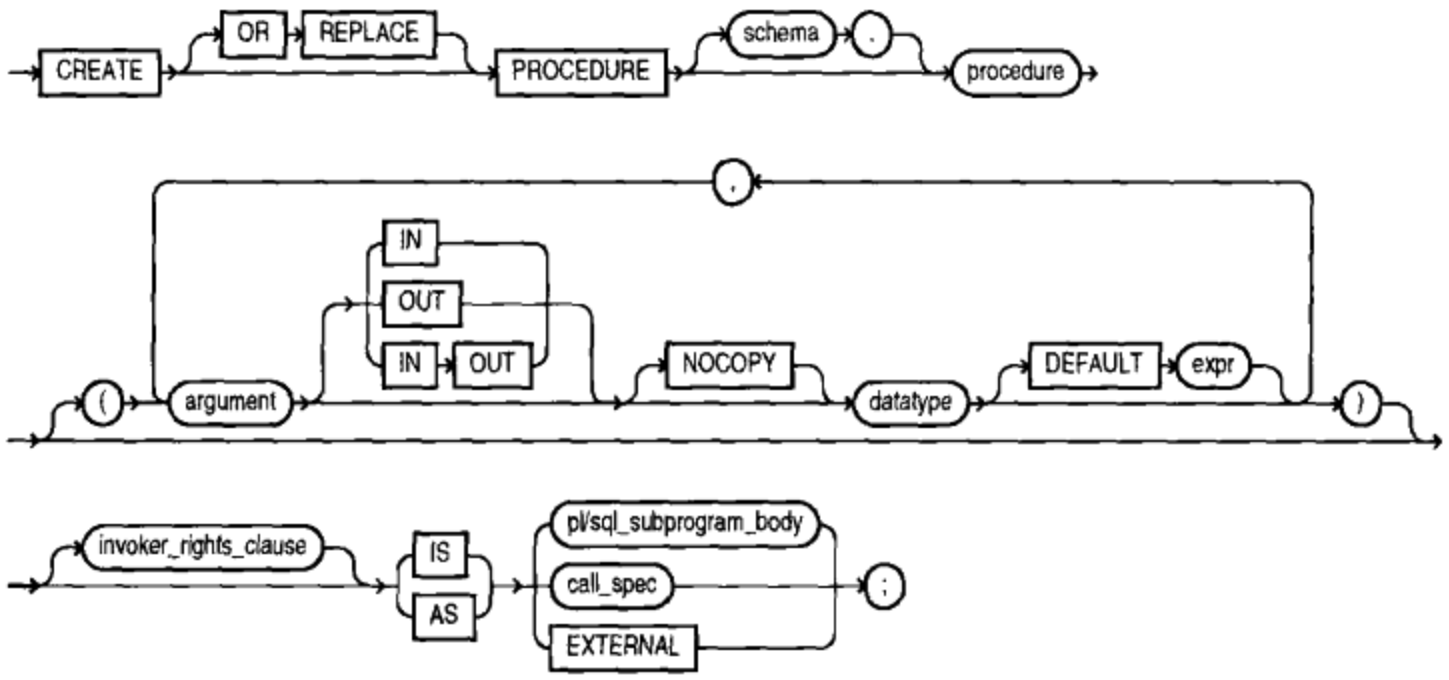
描述：CREATE PACKAGE BODY 构建用 CREATE PACKAGE 创建的指定程序包的程序包体。添加 OR REPLACE 可以替换程序包体(如果程序包体存在的话)。必须拥有 CREATE PROCEDURE 系统权限，才能创建程序包体。要在另一个用户的账户下创建程序包体

CREATE PROCEDURE

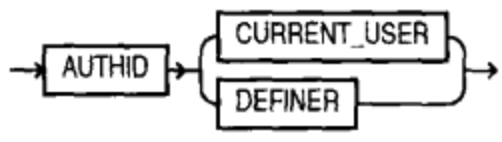
参阅：ALTER PROCEDURE、BLOCK STRUCTURE、CREATE LIBRARY、CREATE FUNCTION、CREATE PACKAGE、DATATYPES、DROP PROCEDURE、第 35 章和第 44 章。

格式：

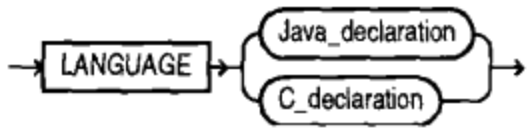
create_procedure::=



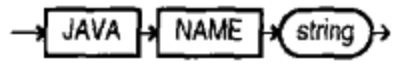
invoker_rights_clause::=



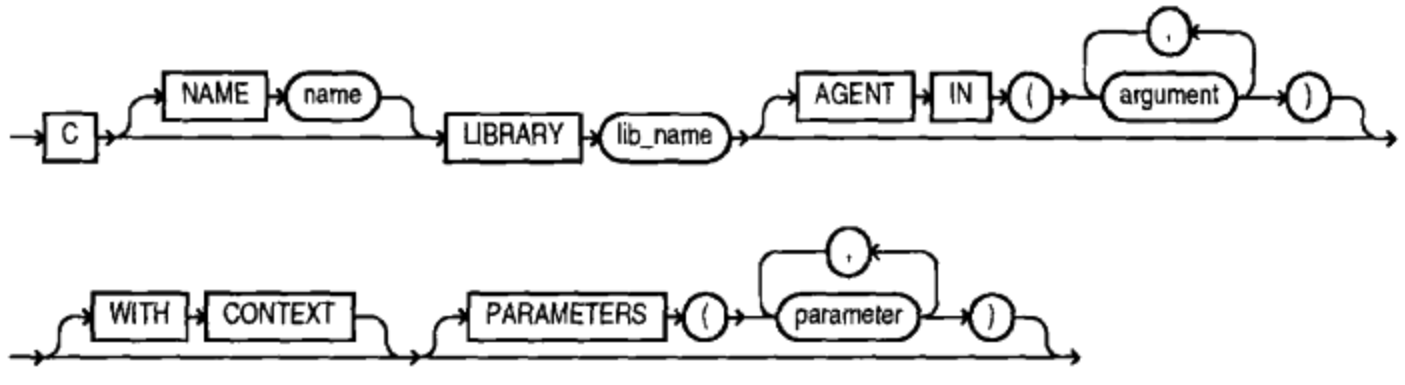
call_spec::=



Java_declaration::=



C_declaration::=



描述：CREATE PROCEDURE 创建过程的规范和过程体。过程可能有参数，即特定数据类型的指定参数。PL/SQL 块将该过程的行为定义为一系列声明、PL/SQL 程序语句和异常。

IN 限定符表示在调用该过程时，必须指定参数的值。OUT 限定符表示该过程通过此参

数将一个值传回给调用者。IN OUT 限定词结合 IN 和 OUT 的含义，即指定一个值，该过程将用另一个值替换它。如果没有任何限定符，则该参数默认为 IN。函数和过程之间的区别是函数将返回一个值给调用环境(除了给任何 OUT 参数之外)。

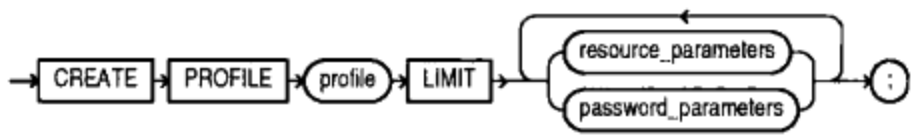
PL/SQL 块可以引用一个外部 C 库中的程序，或引用 Java 调用规范。invoker_rights (AUTHID)子句可以指定该过程是使用函数所有者的权限还是当前调用此函数的用户的权限执行。

CREATE PROFILE

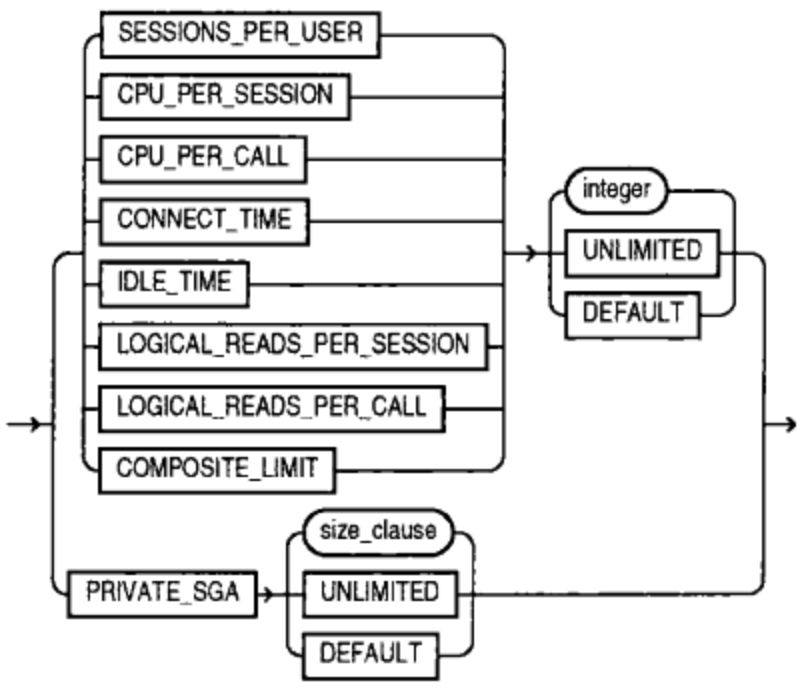
参阅：ALTER PROFILE、ALTER RESOURCE COST、ALTER SYSTEM、ALTER USER、CREATE USER、DROP PROFILE 和第 19 章。

格式：

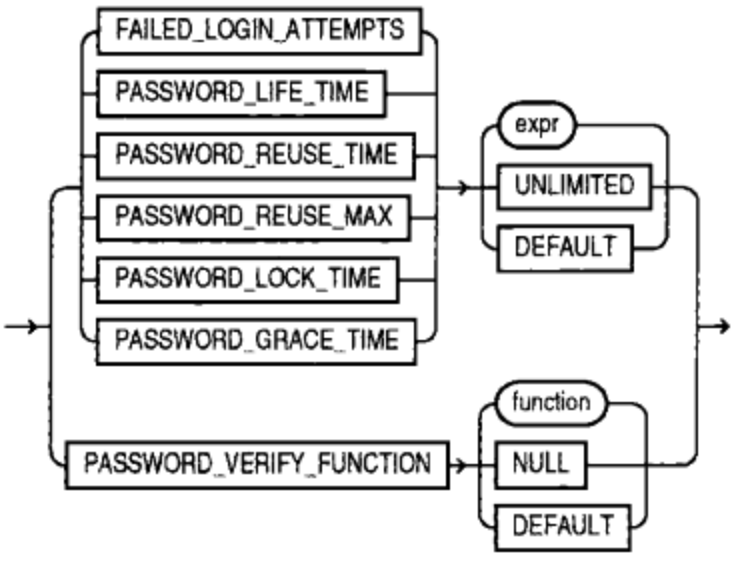
create_profile::=



resource_parameters::=



password_parameters::=



描述：CREATE PROFILE 创建一组关于数据库资源的使用的限制。当使用 CREATE USER 或 ALTER USER 将配置文件与用户相关联时，可以通过那些限制来控制用户所执行的操作。要使用 CREATE PROFILE，必须通过初始化参数 RESOURCE_LIMIT 或通过 ALTER SYSTEM 命令启用资源限制。

SESSIONS_PER_USER 把用户限定到几个(integer)并发的 SQL 会话。CPU_PER_SESSION 限制 CPU 的时间，以百分之一秒为单位。CPU_PER_CALL 限制分析、执行或获取调用的 CPU 时间，以百分之一秒为单位。CONNECT_TIME 限制会话耗费的时间，以分钟为单位。IDLE_ TIME 限制在指定分钟后断开与用户的连接，但该限制在查询正在运行时不能应用。LOGICAL_READS_PER_SESSION 限制每个会话读取的块数，LOGICAL_READS_PER_CALL 为分析、执行或获取调用限制每个会话读取的块数。PRIVATE_SGA 限制在 SGA 中为私有空间所分配的空间量，K 和 M 选项只应用于此限制。COMPOSITE_LIMIT 基于 CPU、连接时间、逻辑读和私有 SGA 资源的加权和，来限制服务单元中一个会话的总资源成本。

UNLIMITED 表示在特殊的资源上没有限制。DEFAULT 从 DEFAULT 配置文件中查找限制条件，该值可以通过 ALTER PROFILE 命令进行修改。

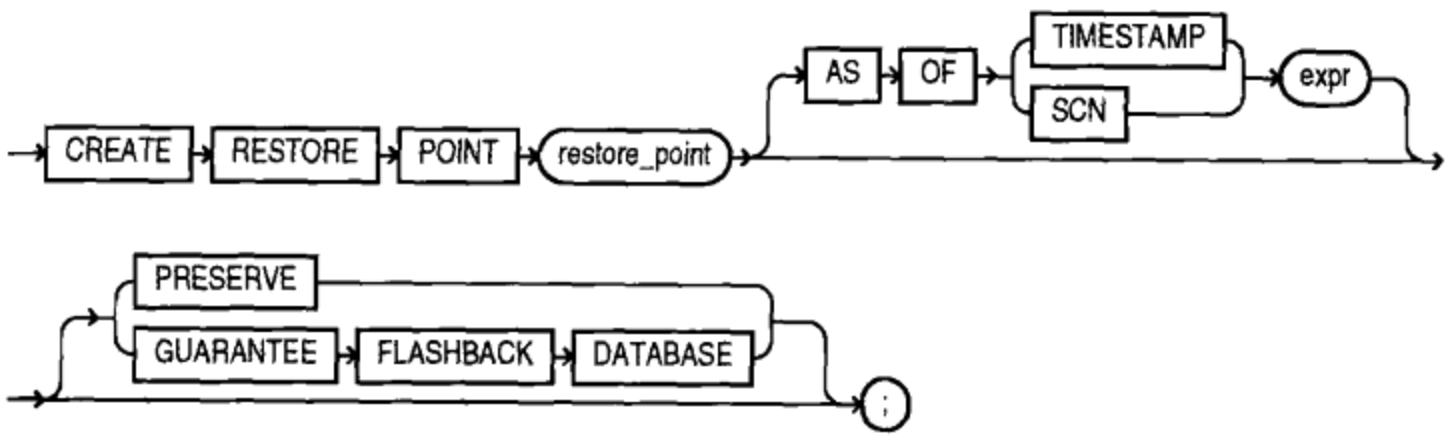
如果用户超越了某个限制，则 Oracle 将终止并回滚该事务，然后结束会话。要创建配置文件，用户必须拥有 CREATE PROFILE 系统权限。可以使用 ALTER USER 命令将配置文件与用户关联在一起。

CREATE RESTORE POINT

参阅：FLASHBACK DATABASE、FLASHBACK TABLE 和 DROP RESTORE POINT。

格式：

create_restore_point::=



描述：使用 CREATE RESTORE POINT 来创建还原点，它是与时间戳或数据库的 SCN 相关联的名称。还原点可用来把表或数据库到闪回还原点指定的时间，而无需确定 SCN 或时间戳。还原点也可用于各种 RMAN 操作，包括备份和数据库复制。可以使用 RMAN 在实现归档备份的过程中创建还原点。

CREATE ROLE

参阅：ALTER ROLE、ALTER USER、CREATE USER、DROP ROLE、GRANT、REVOKE、SET ROLE 和第 19 章。

格式：

```
CREATE ROLE role
    [ NOT IDENTIFIED
    | IDENTIFIED { BY password |
    USING [ schema .] package | EXTERNALLY | GLOBALLY }];
```

描述：使用 CREATE ROLE，可以创建一个指定的角色或一组权限。在向用户授予该角色时，就向其授予了该角色的所有权限。首先用 CREATE ROLE 创建角色，然后使用 GRANT 命令向该角色授予权限。在用户要访问该角色允许使用的内容时，可以用 SET ROLE 启用该角色。另外，该角色可以通过 ALTER USER 或 CREATE USER 命令设置为用户的 DEFAULT 角色。

如果在角色上没置口令保护，则想要使用权限的用户必须在 SET ROLE 命令中提供口令，以便启用该角色。USING package 子句可以创建一个应用程序角色，它是只能由使用授权程序包的应用程序而启用的角色。如果不指定 schema，则 Oracle 假定该程序包在您自己的模式中。

CREATE ROLLBACK SEGMENT

参阅：ALTER ROLLBACK SEGMENT、CREATE DATABASE、CREATE TABLESPACE、DROP ROLLBACK SEGMENT、STORAGE 和第 51 章。

格式：

```
CREATE [PUBLIC] ROLLBACK SEGMENT rollback_segment
    [{ TABLESPACE tablespace | storage_clause }
    [ TABLESPACE tablespace | storage_clause ]...];
```

描述：rollback_segment 是赋予该回滚段的名称。tablespace 是赋予该回滚段的表空间名。一个表空间可以有多个回滚段。

storage_clause 包含可在 STORAGE 下描述的子句。如果使用 PUBLIC，则该回滚段可以由请求它的任何实例使用；否则只能用于在初始化参数文件中命名它的实例。

CREATE SCHEMA

参阅：CREATE TABLE、CREATE VIEW 和 GRANT。

格式：

```
CREATE SCHEMA AUTHORIZATION schema
    { create_table_statement | create_view_statement | grant_statement }
    [ create_table_statement | create_view_statement | grant_statement ]...;
```

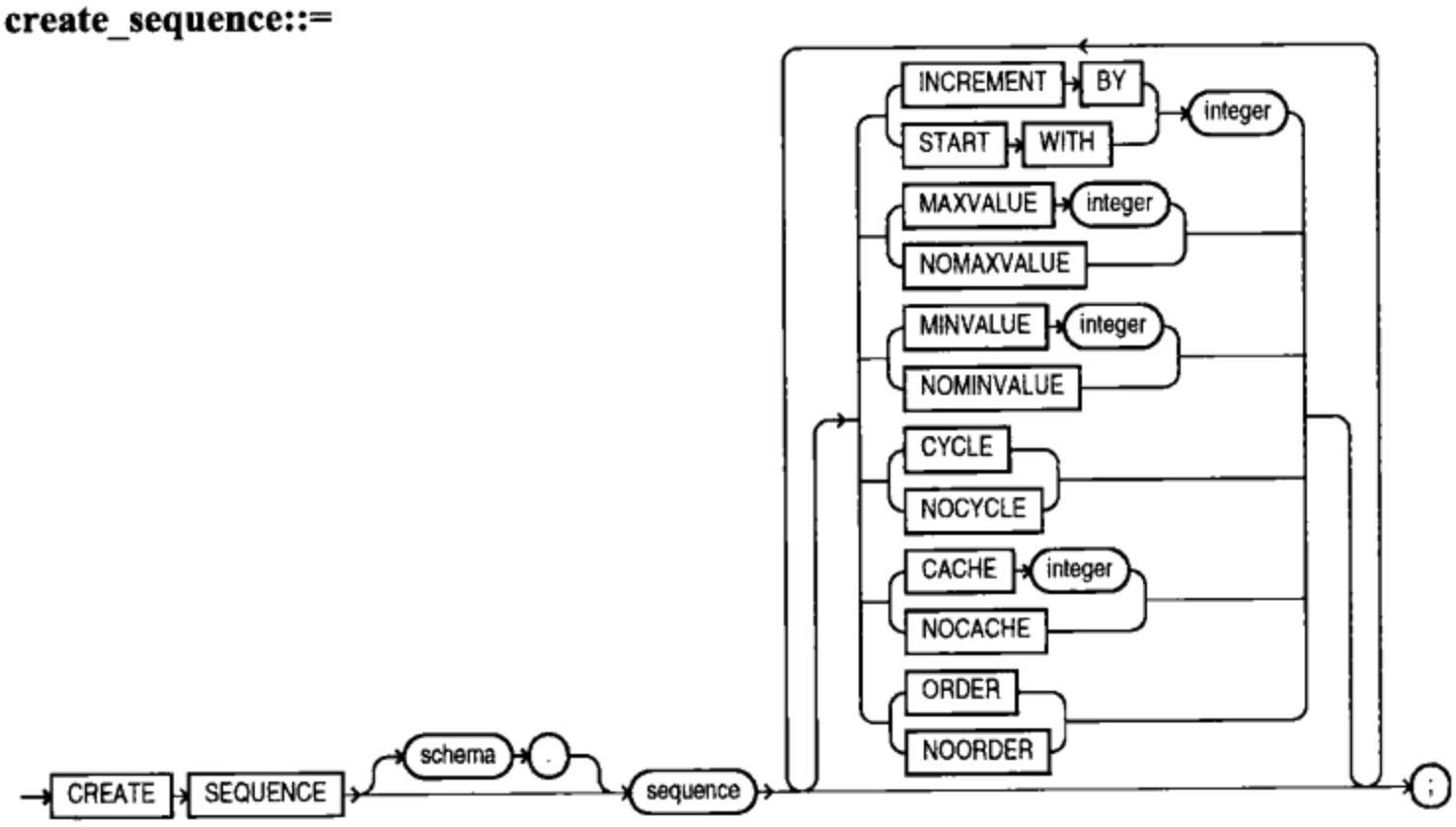
描述：CREATE SCHEMA 命令像创建单个事务一样创建表、视图和授权的一个集合。模式名与 Oracle 用户名相同。CREATE TABLE、CREATE VIEW 和 GRANT 命令都是标准的命令，而且命令出现的顺序不重要，即使存在内部依赖性，也是如此。

CREATE SEQUENCE

参阅：ALTER SEQUENCE、AUDIT、DROP SEQUENCE、GRANT、REVOKE、NEXTVAL、PSEUDO-COLUMNS 下的 CURRVAL 和第 17 章。

格式：

create_sequence::=



描述： sequence 是赋予序列的名称。默认的 INCREMENT BY 为 1。正数表示序列号按该整数等间隔递增。负数表示序列以相同的方式递减(序列号减少的值)。START WITH 是该序列将开始的号码。对于递增序列，默认的 START WITH 用 MAXVALUE 表示；对于递减序列，默认的 STARTWITH 用 MINVALUE 表示。可使用 START WITH 重写该默认值。

MINVALUE 是该序列将生成的最小号码。对于递增序列其默认值为 1。MAXVALUE 是序列将生成的最大号码，对于递减序列，该默认值为 - 1。为了允许使序列无限制地递增，只需为递增序列指定 MINVALUE、为递减序列指定 MAXVALUE 即可。为停止创建序列号并在试图生成新序列号时强制一个错误，可以为递增序列指定 MAXVALUE，或为递减序列指定 MINVALUE，再加上 NOCYCLE。为了重新启动以 MAXVALUE 或 MINVALUE 开始的序列，可指定 CYCLE。

CACHE 允许将一组预先分配的序列号保存在内存中。默认值为 20。该值集必须小于 MAXVALUE 减 MINVALUE 的差。

ORDER 保证序列号将按照接收请求的顺序分配给请求它们的实例。这只在使用实时应用群集的应用程序中需要。

要在自己的模式中创建序列，必须拥有 CREATE SEQUENCE 系统权限。要在另一个用户的模式中创建序列，必须拥有 CREATE ANY SEQUENCE 系统权限。

CREATE SPFILE

参阅： CREATE PFILE 和 init.ora。

格式：

```
CREATE SPFILE [= 'spfile_ name'] FROM PFILE [= 'pfile_ name'];
```

描述： CREATE SPFILE 从客户端基于文本的初始化参数文件中创建服务器参数文件。

服务器参数文件是只存在于服务器上的二进制文件，从客户机位置调用以启动数据库。服务器参数文件允许对个别参数做永久修改。通过 ALTER SYSTEM 命令所做的参数修改将在数据库关闭和启动过程中保留，即使不手动更新外部参数文件也是如此。

在 Oracle Database 11g 中，可以从系统范围的当前参数设置来创建系统参数文件。

CREATE SYNONYM

参阅：CREATE DATABASE LINK、CREATE TABLE、CREATE VIEW、第 17 章和第 25 章。

格式：

```
CREATE [PUBLIC] SYNONYM [ schema .] synonym
    FOR [ shema .] object [@ dblink];
```

描述：CREATE SYNONYM 创建表、视图、序列、存储过程、函数、程序包、物化视图、Java 类对象或同义词等对象的同义词，其中也包含在远程数据库中的那些对象。PUBLIC 使同义词对于所有的用户可用，但同义词只能由 DBA 或拥有 CREATE PUBLIC DATEBASE LINK 系统权限的用户来创建。如果没有 PUBLIC，则用户必须用其用户名作为同义词的前缀。synonym 是同义词的名称。schema.object 是系统引用的表、视图或同义词的名称。同义词也可以有同义词。@database_link 是到远程数据库的一个数据库链接。同义词引用由该数据库链接指定的远程数据库中的表。

示例：

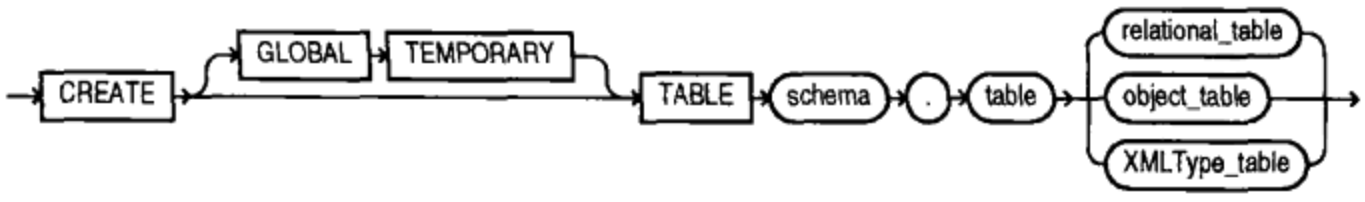
```
create synonym LOCAL_BOOKSHELF for BOOKSHELF@REMOTE_CONNECT;
```

CREATE TABLE

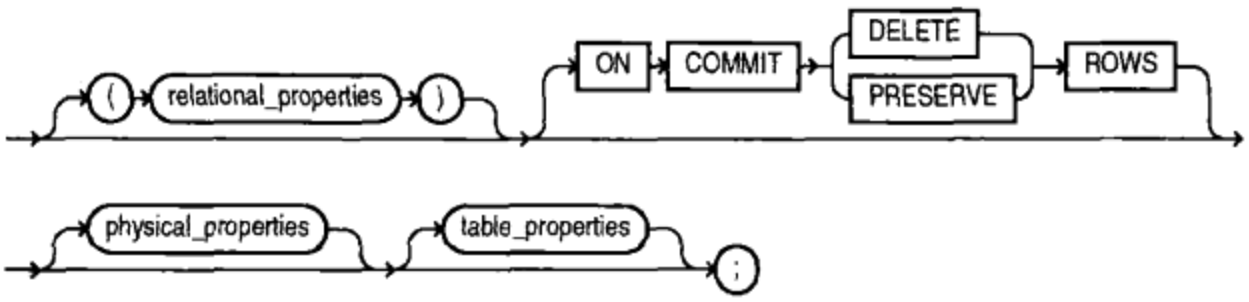
参阅：ALTER TABLE、CREATE CLUSTER、CREATE INDEX、CREATE TABLESPACE、CREATE TYPE、DATATYPES、DROP TABLE、INTEGRITY CONSTRAINT、OBJECT NAMES、STORAGE、第 4 章、第 17 章、第 28 章、第 38 章、第 39 章和第 41 章。

格式：

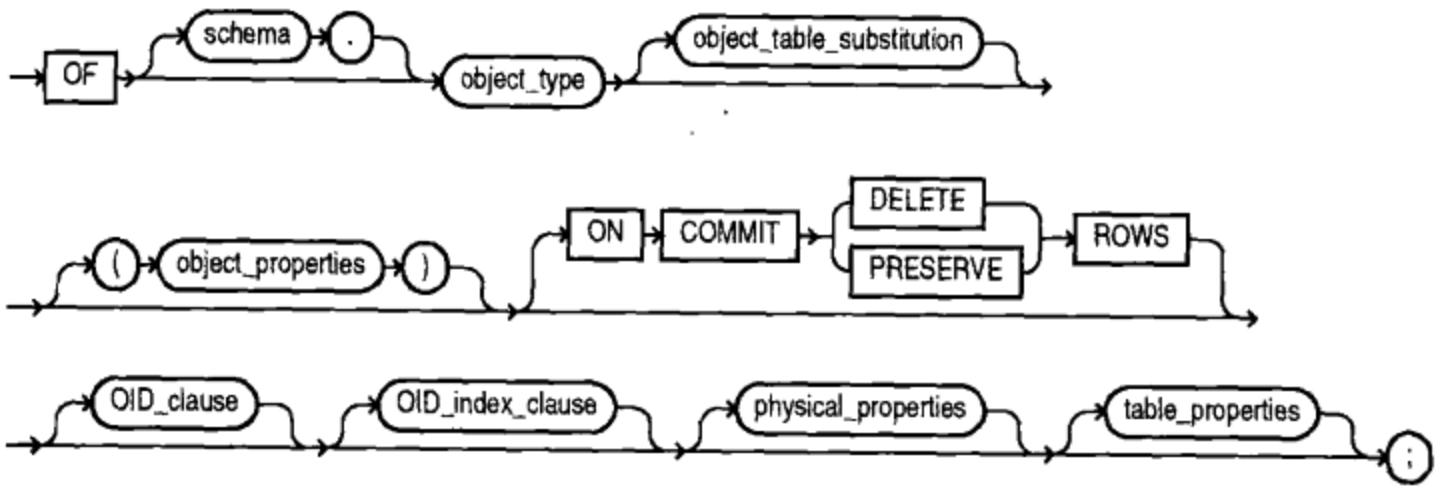
create table::=



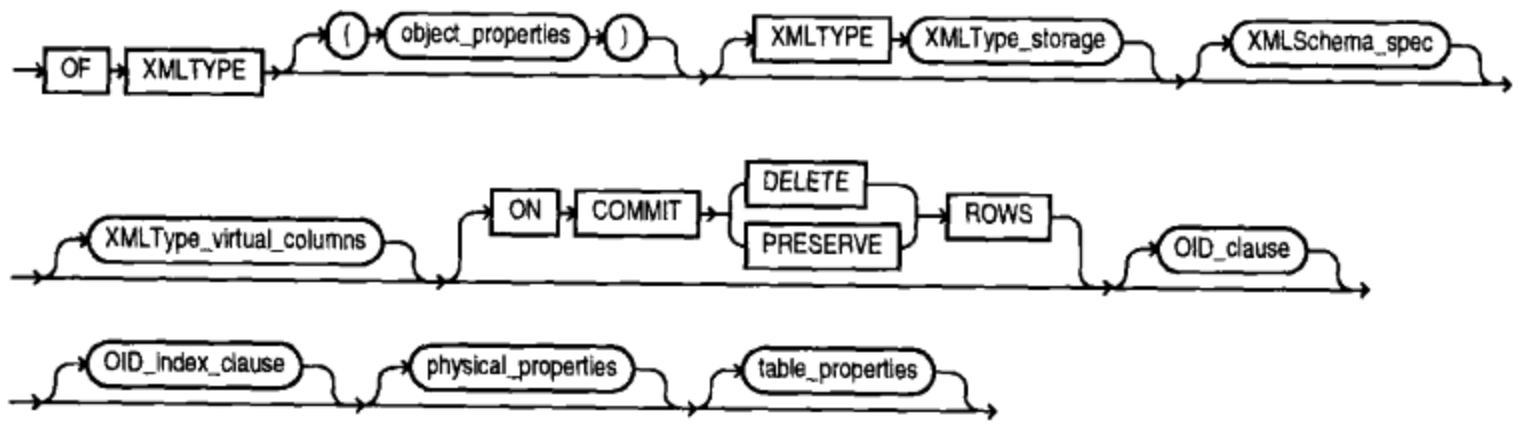
relational_table::=



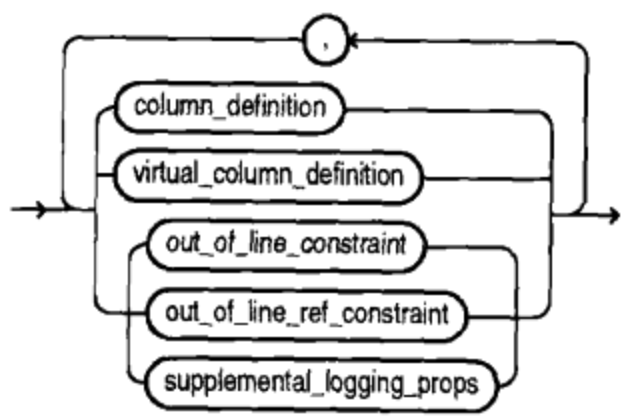
object_table::=



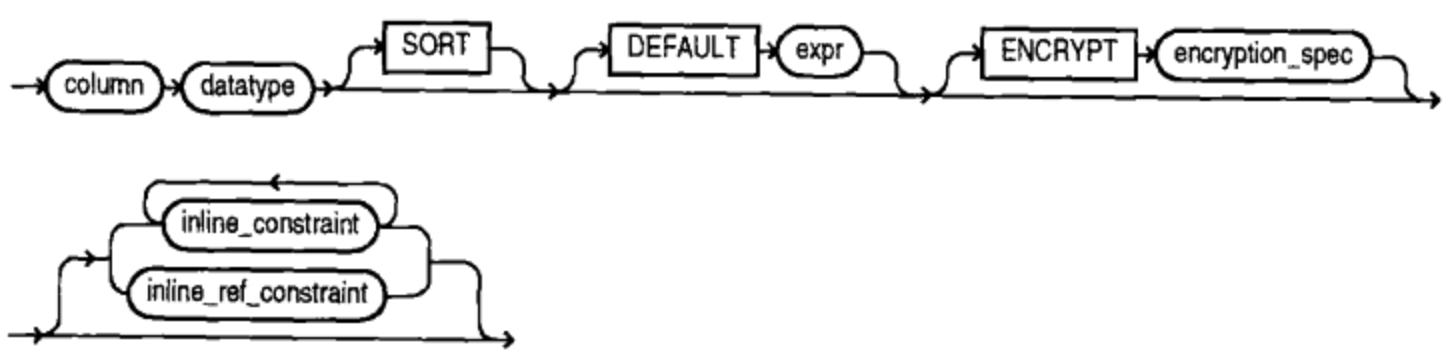
XMLType_table::=



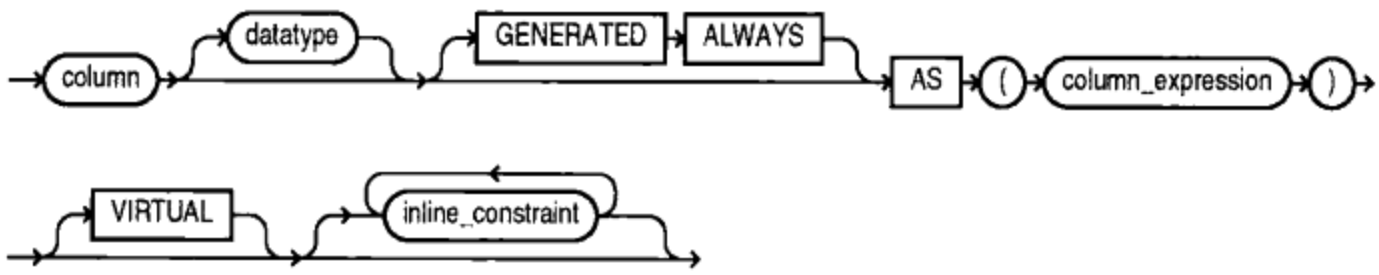
relational_properties::=



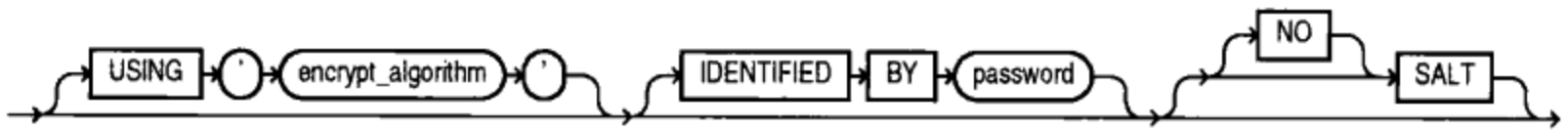
column_definition::=



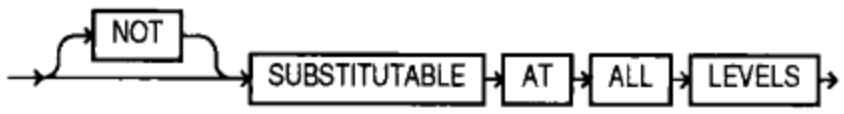
virtual_column_definition::=



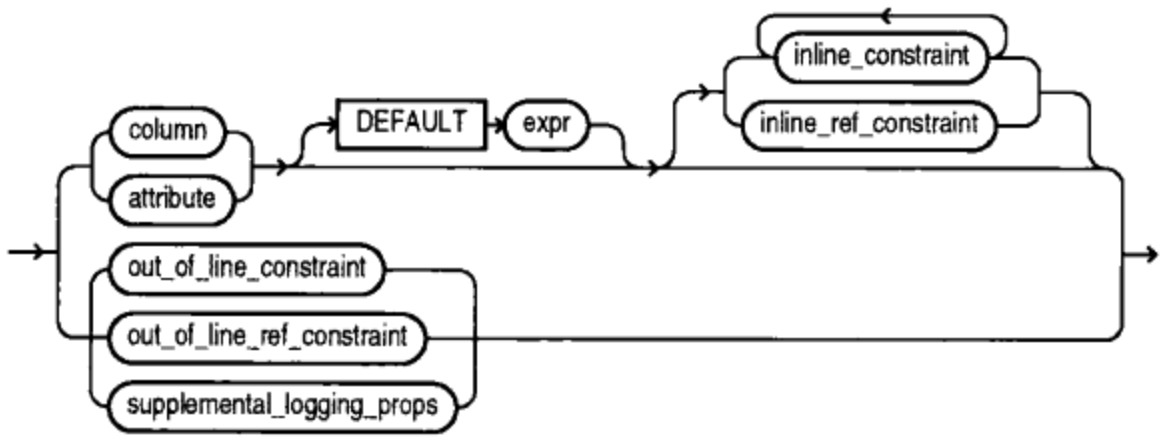
encryption_spec::=



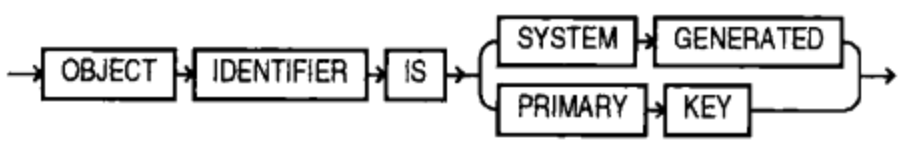
object_table_substitution::=



object_properties::=

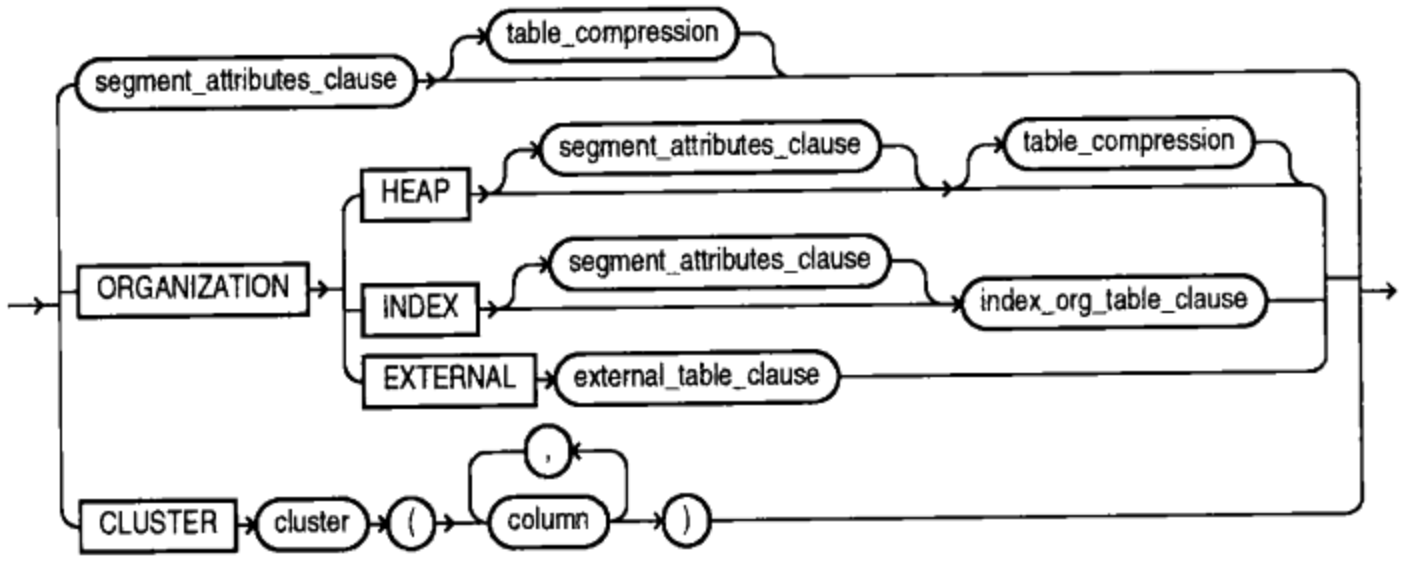


oid_clause::=

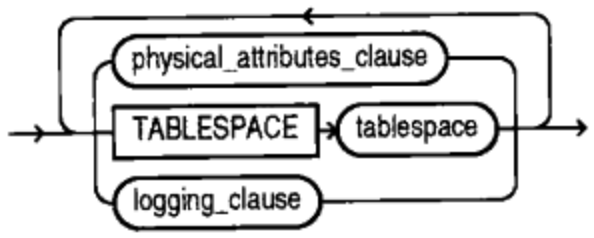


oid_index_clause::=

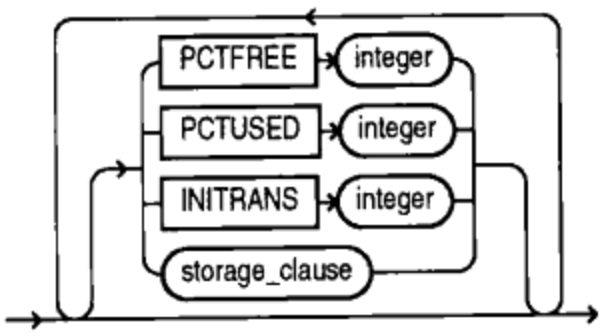
physical_properties::=



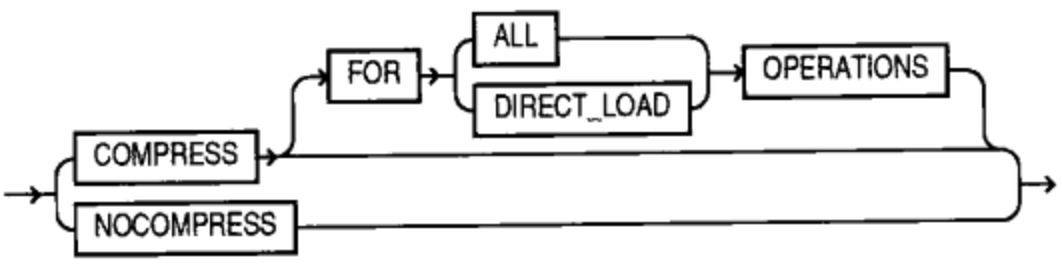
segment_attributes_clause::=



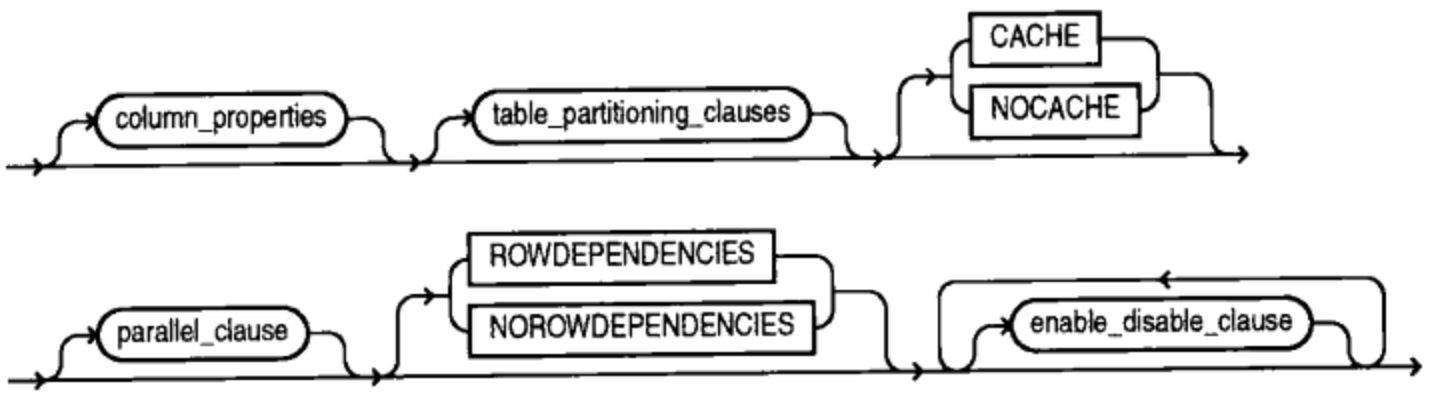
physical_attributes_clause::=

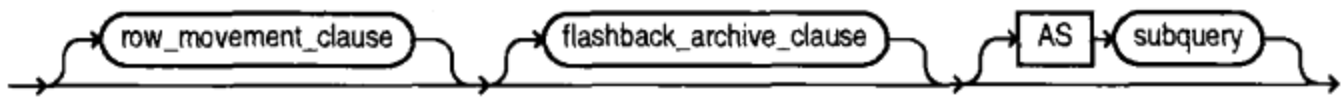


table_compression::=

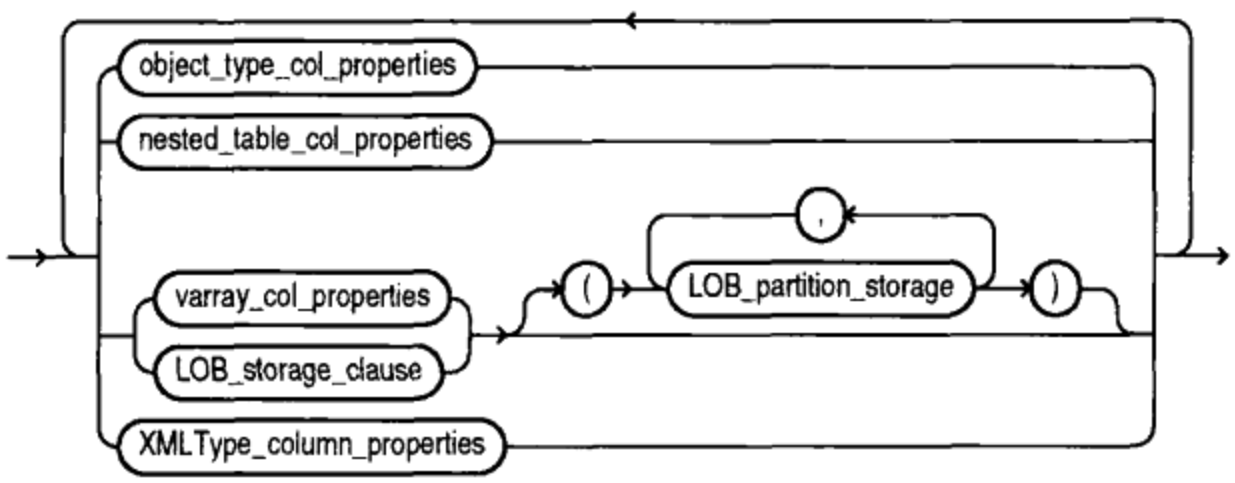


table_properties::=

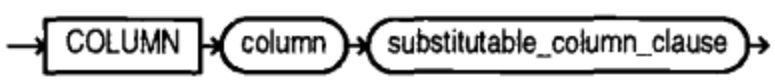




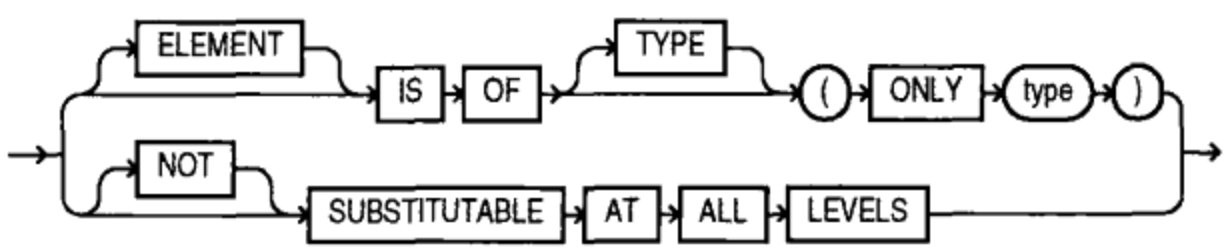
column_properties::=



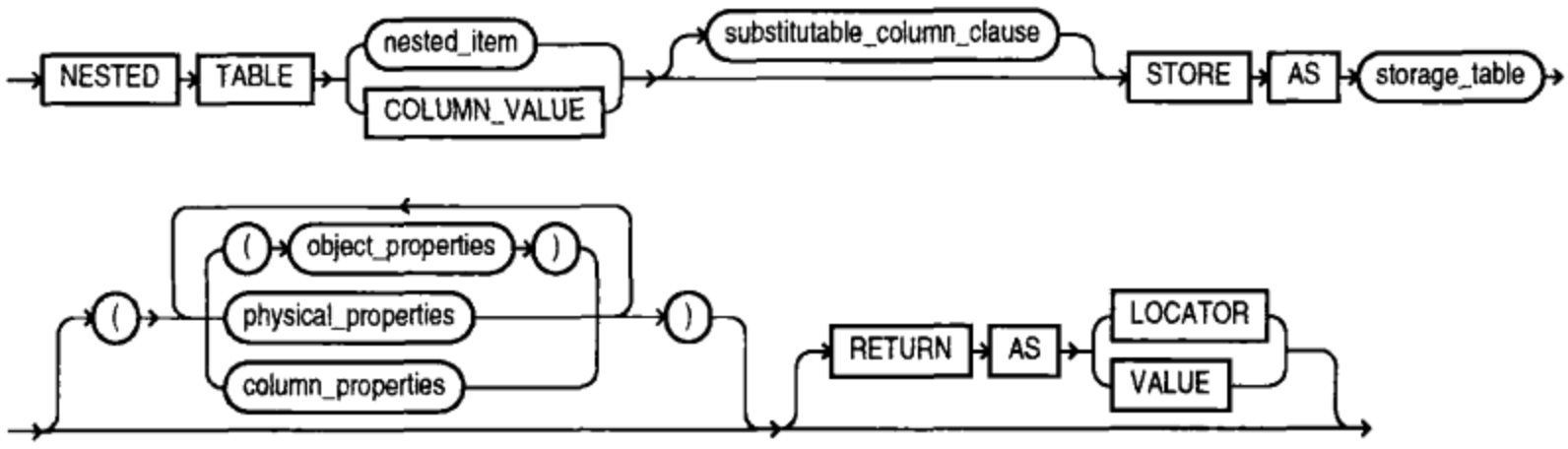
object_type_col_properties::=



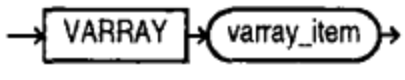
substitutable_column_clause::=

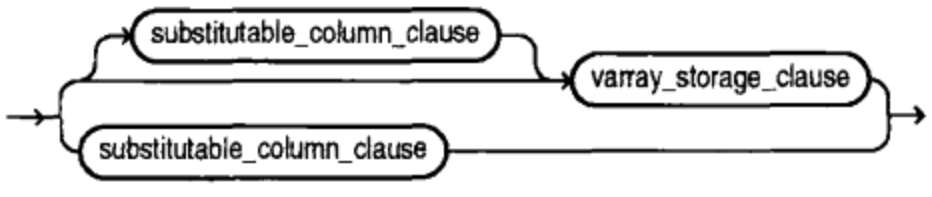


nested_table_col_properties::=

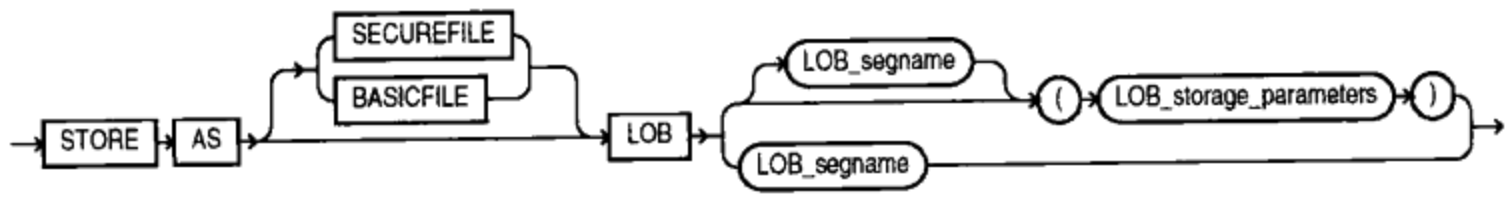


varray_col_properties::=

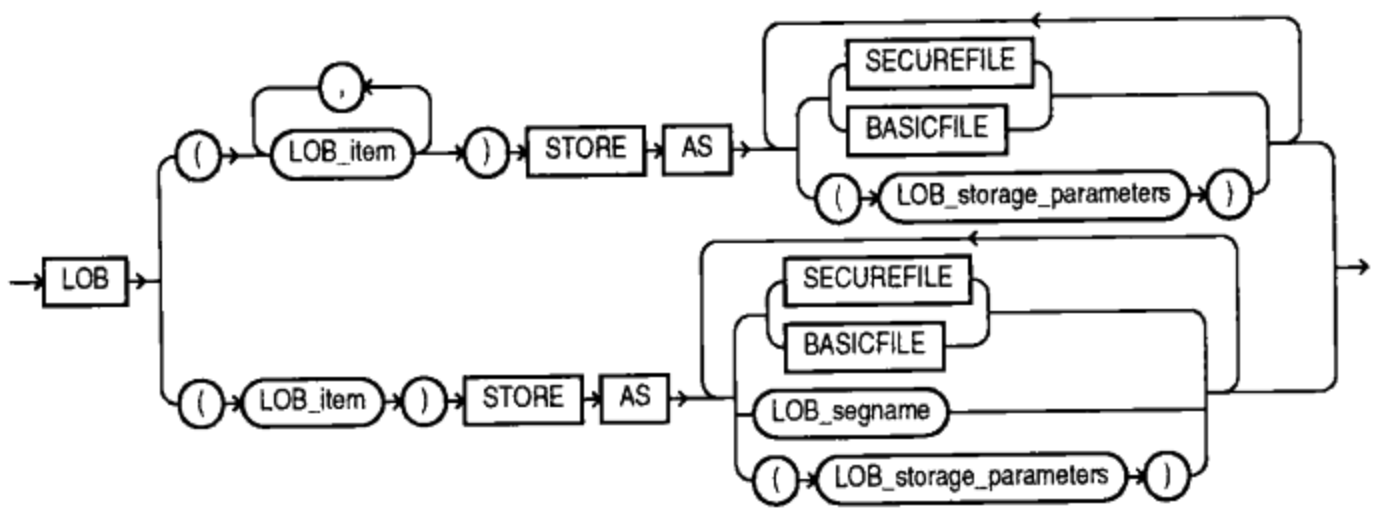




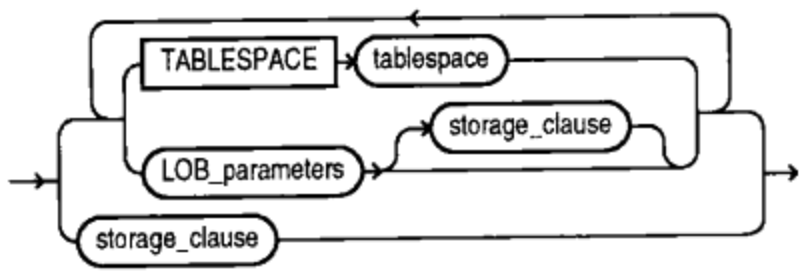
varray_storage_clause::=



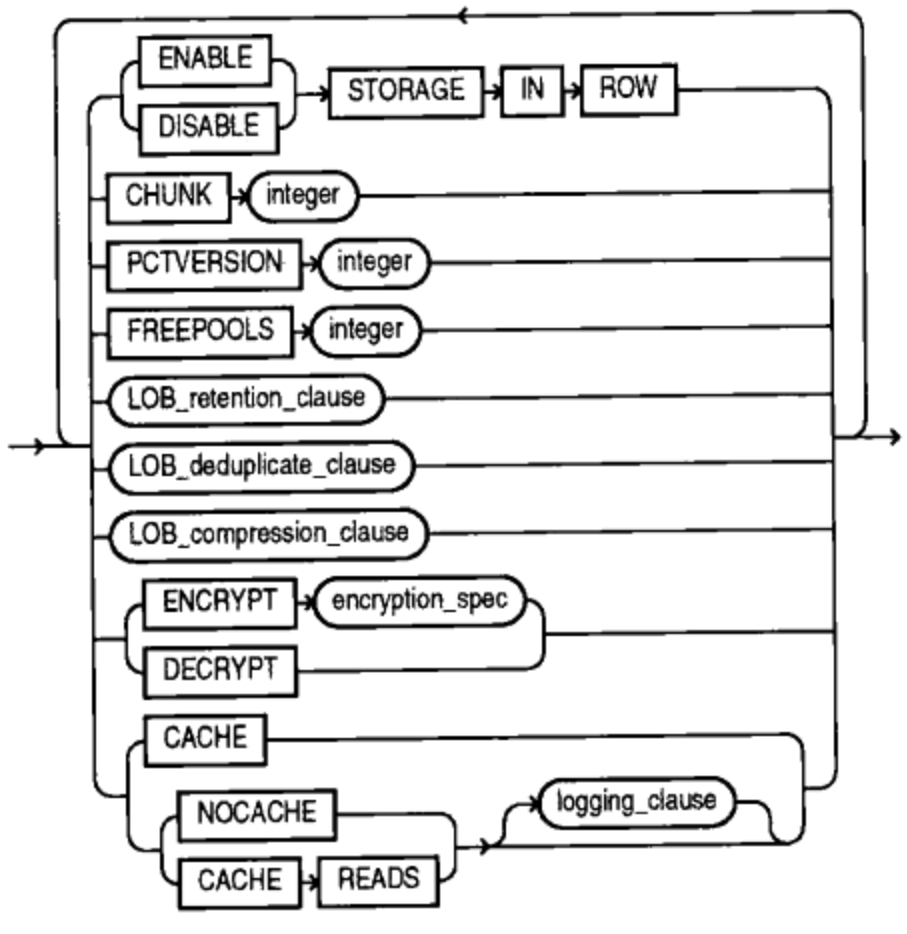
LOB_storage_clause::=



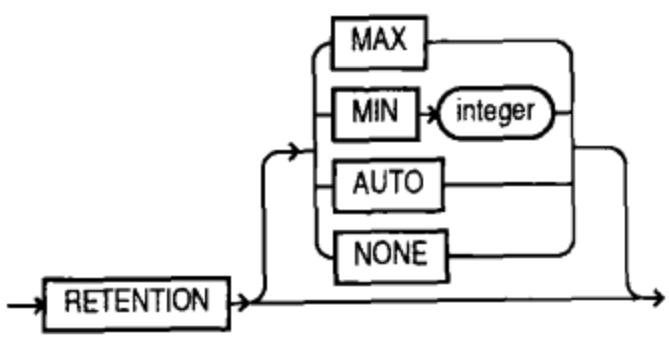
LOB_storage_parameters::=



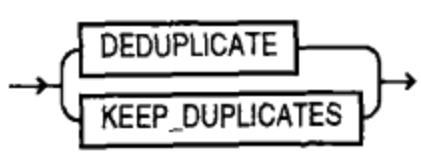
LOB_parameters::=



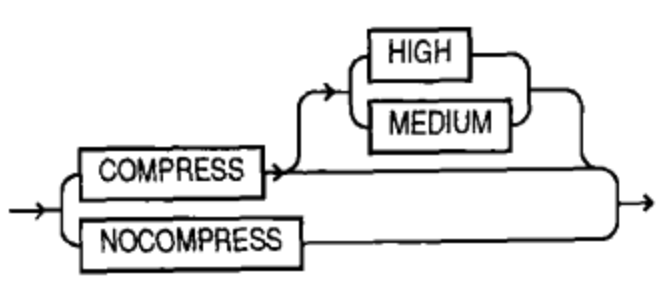
LOB_retention_clause::=



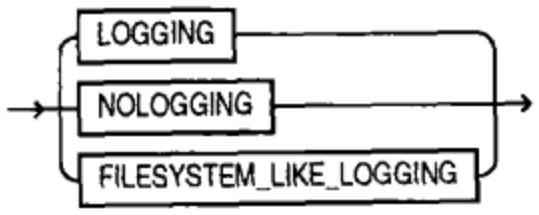
LOB_deduplicate_clause::=



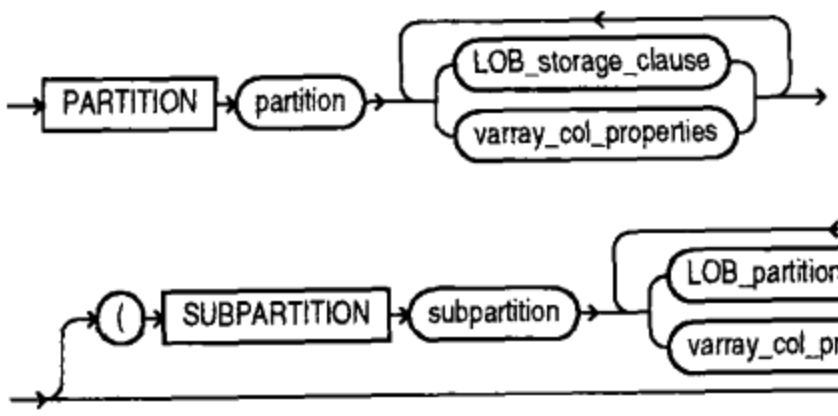
LOB_compression_clause::=



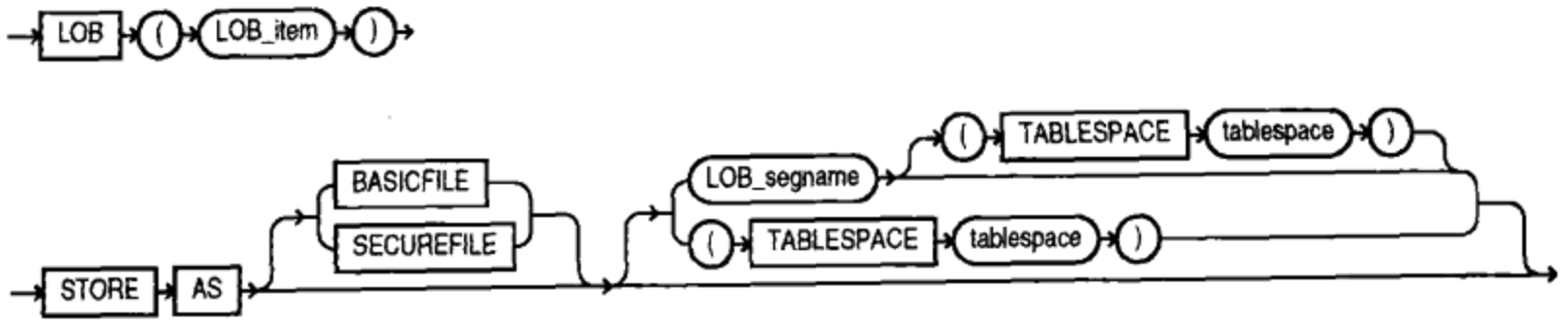
logging_clause::=



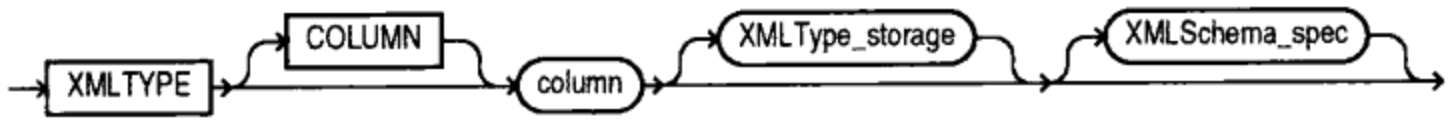
LOB_partition_storage::=



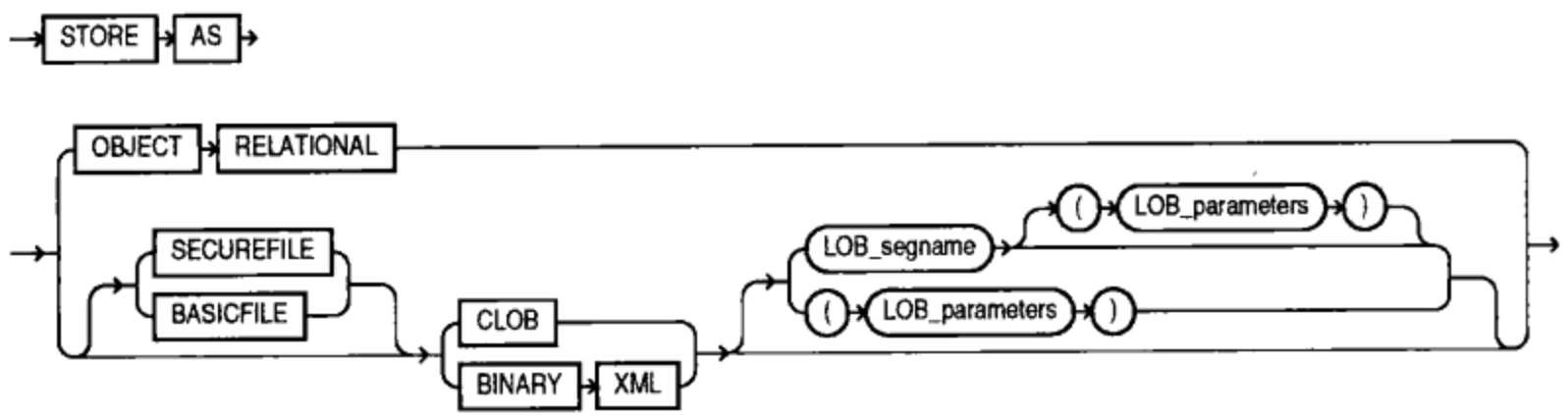
LOB_partitioning_storage::=



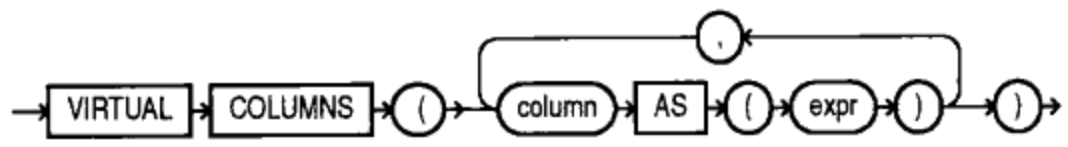
XMLType_column_properties::=



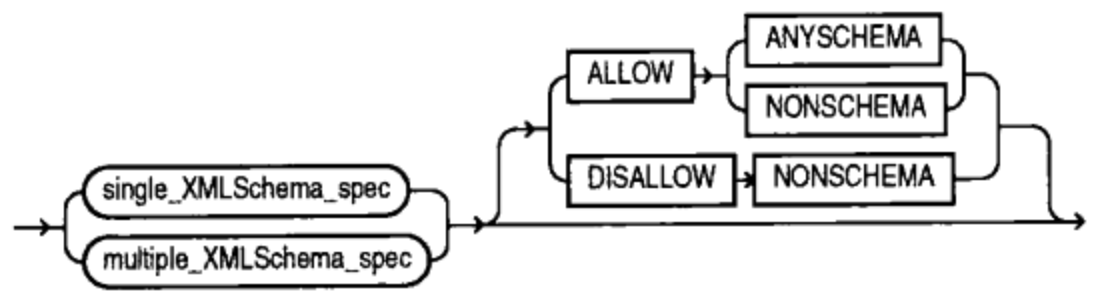
XMLType_storage::=



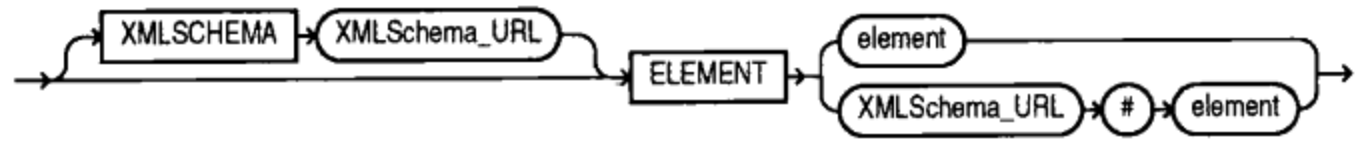
XMLType_virtual_columns::=



XMLSchema_spec::=

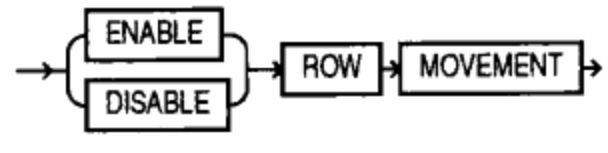


single_XMLSchema_spec::=

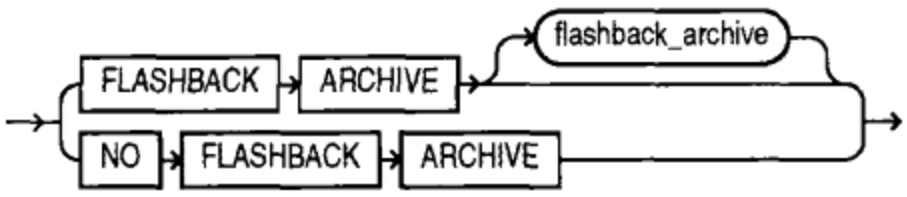


multiple_XMLSchema_spec::=

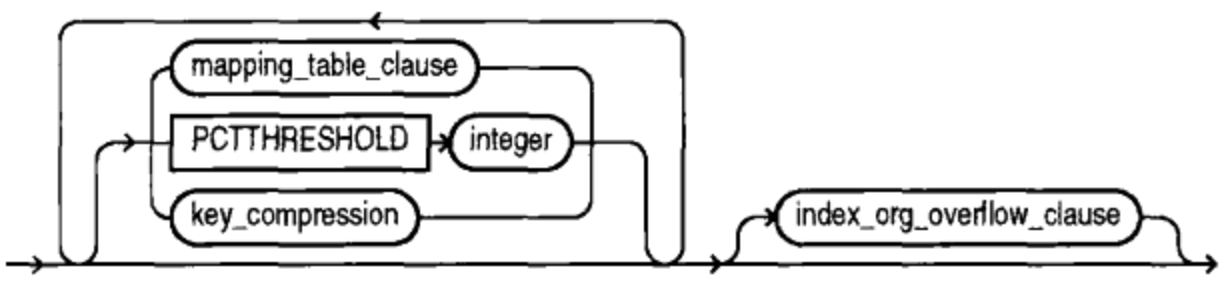
row_movement_clause::=



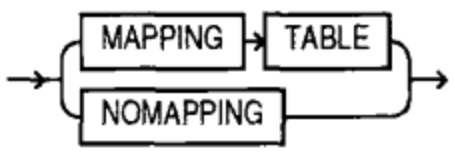
flashback_archive_clause::=



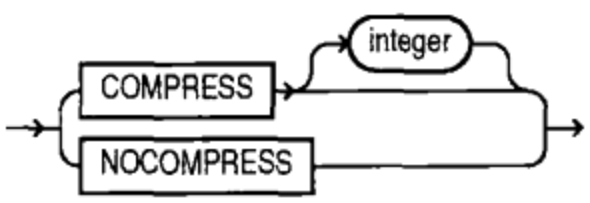
index_org_table_clause::=



mapping_table_clauses::=

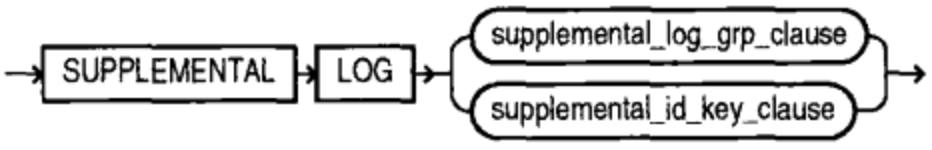


key_compression::=

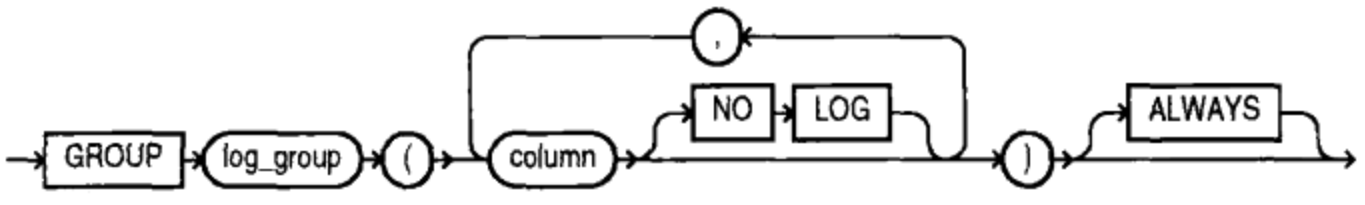


index_org_overflow_clause::=

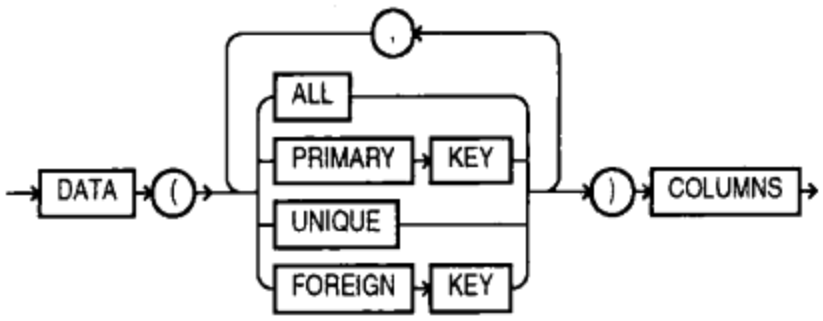
supplemental_logging_props::=



supplemental_log_grp_clause::=

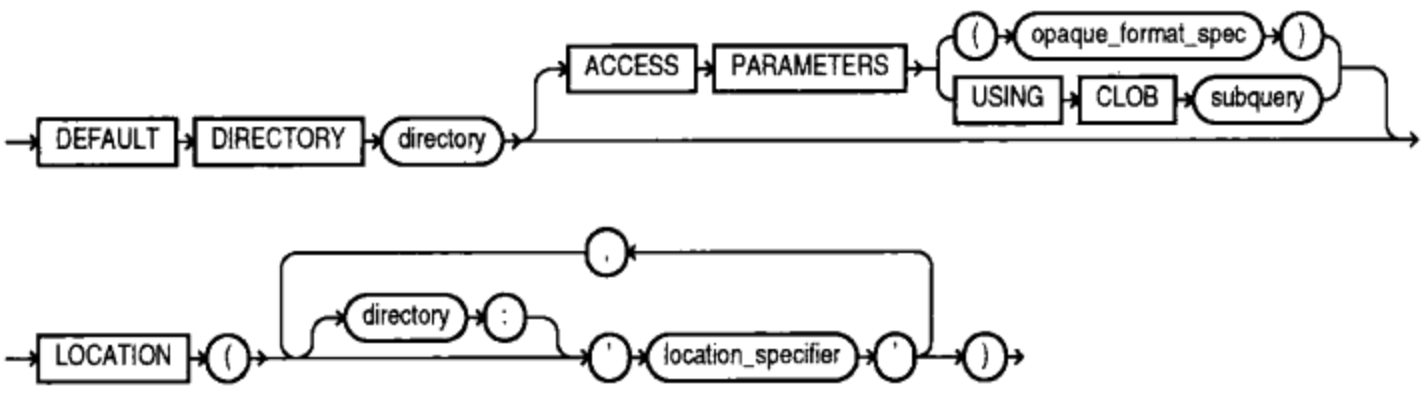


supplemental_id_key_clause::=

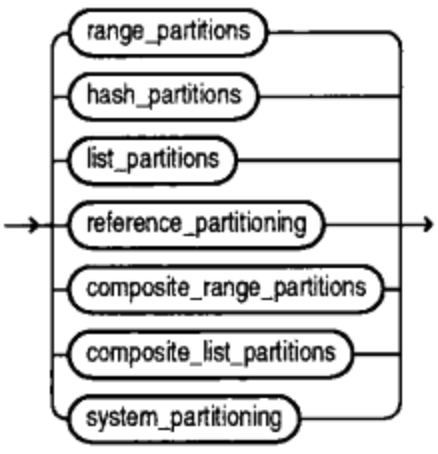


external_table_clause::=

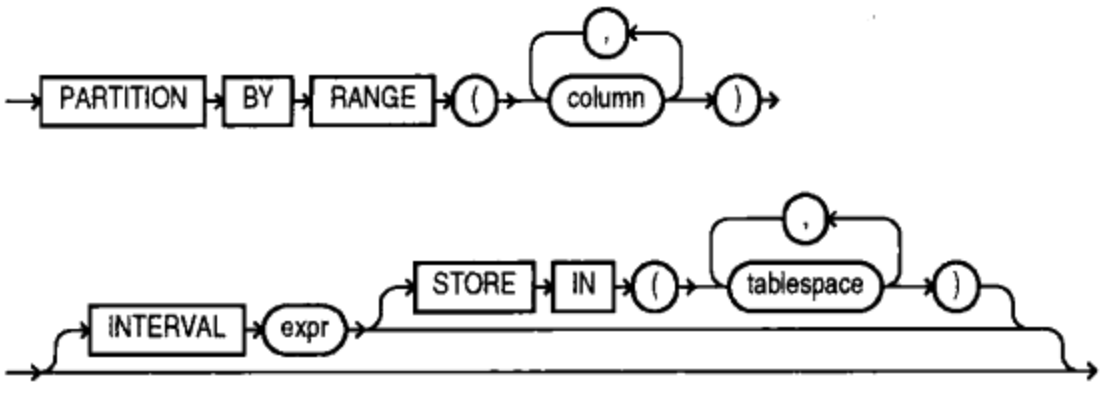
external_data_properties::=

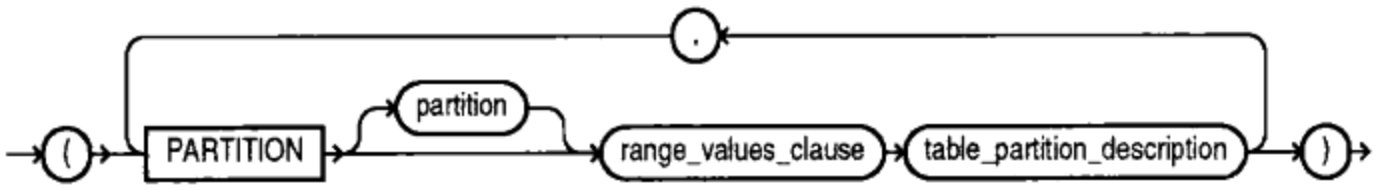


table_partitioning_clauses::=

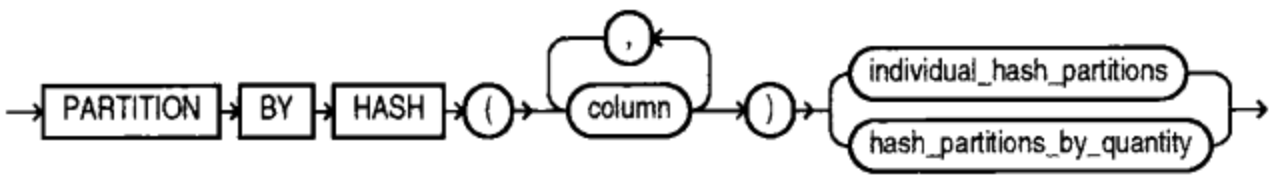


range_partitions::=

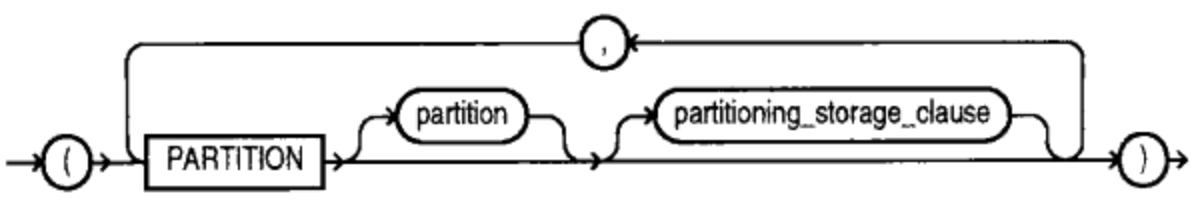




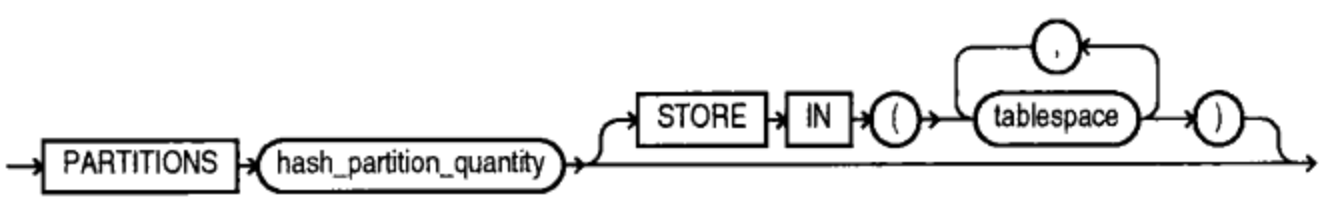
hash_partitions::=

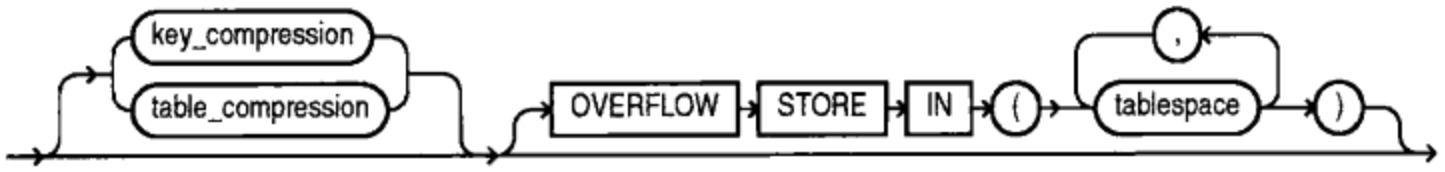


individual_hash_partitions::=

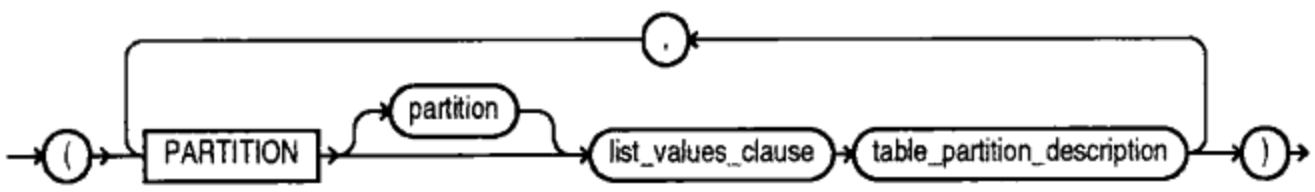


hash_partitions_by_quantity::=

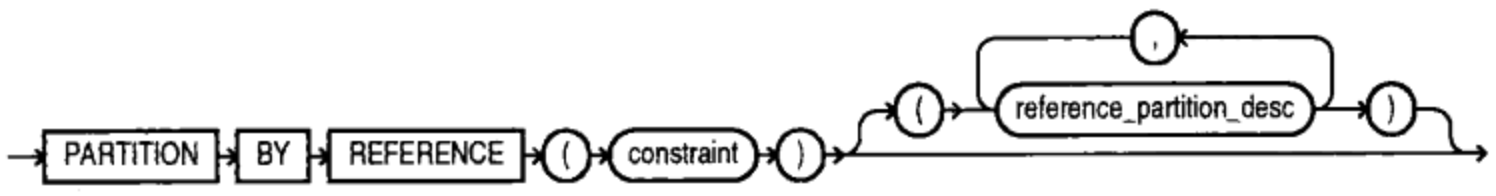




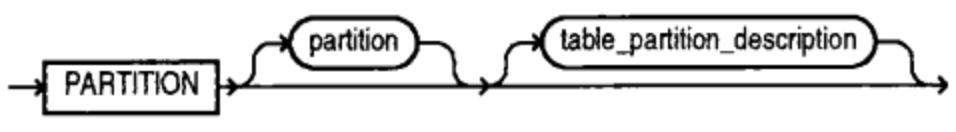
list_partitions::=



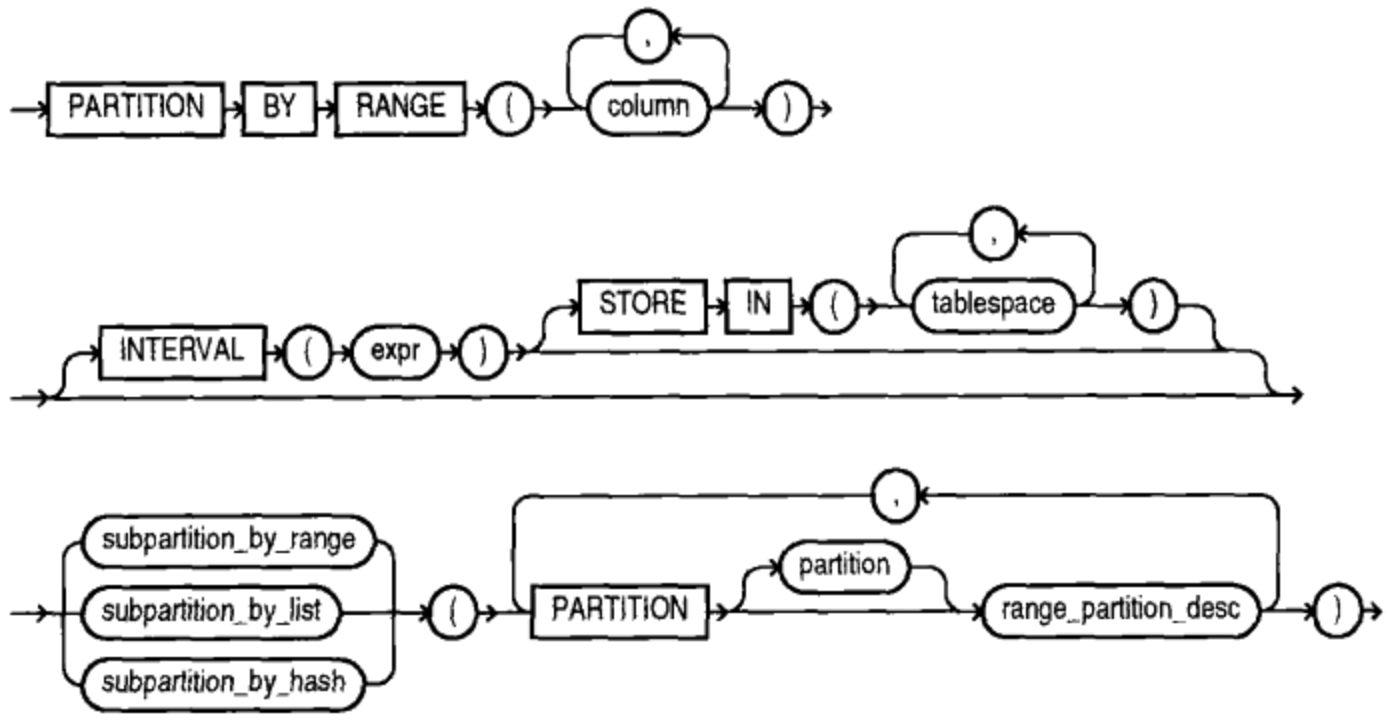
reference_partitioning::=



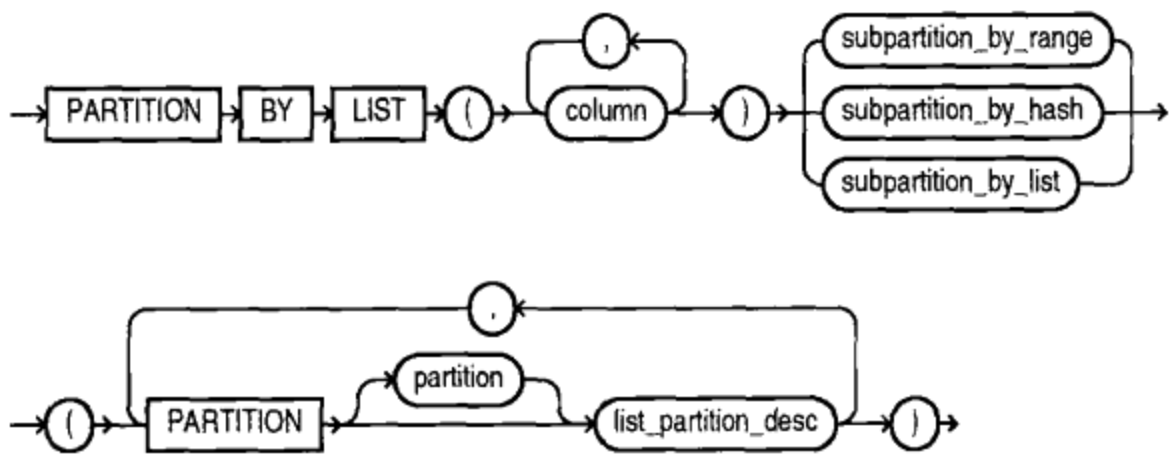
reference_partition_desc::=



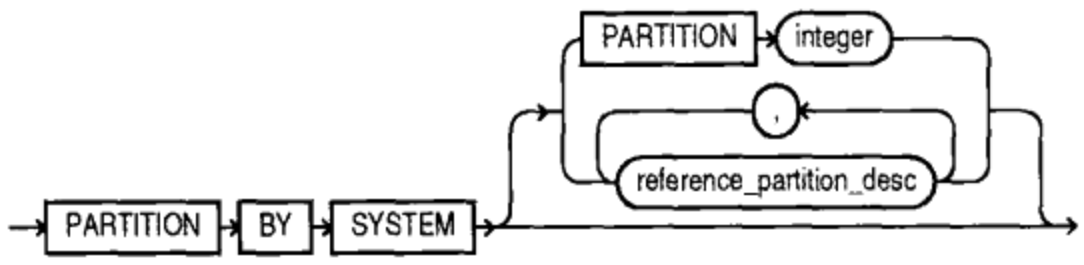
composite_range_partitions::=



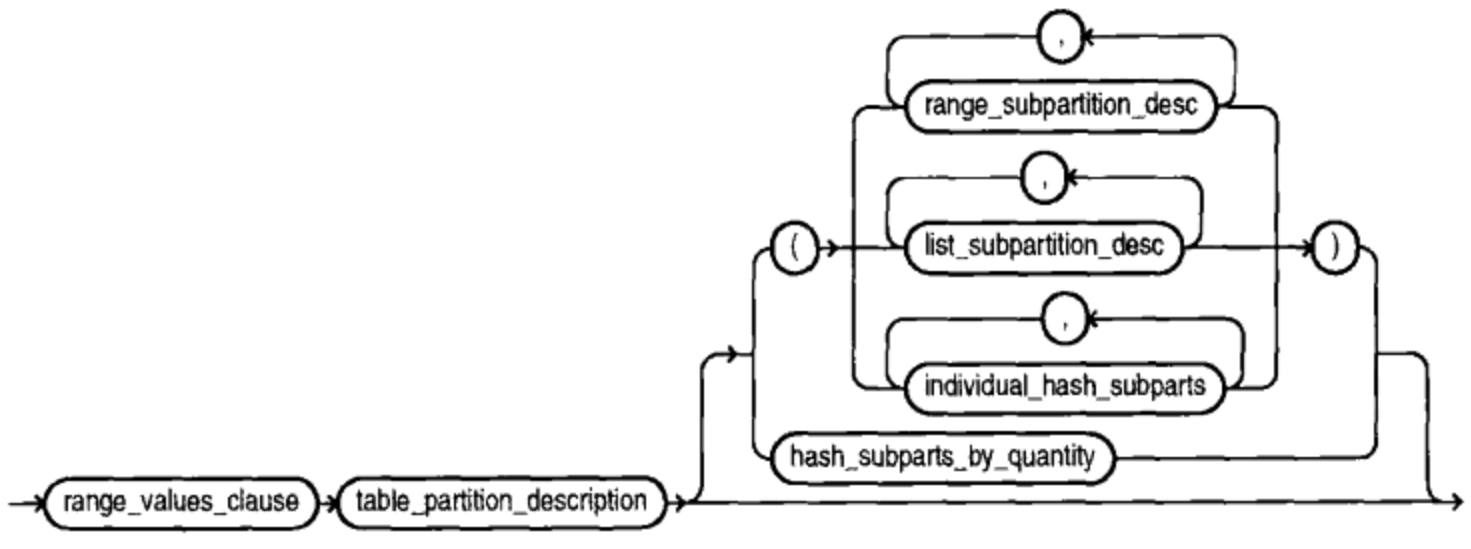
composite_list_partitions::=



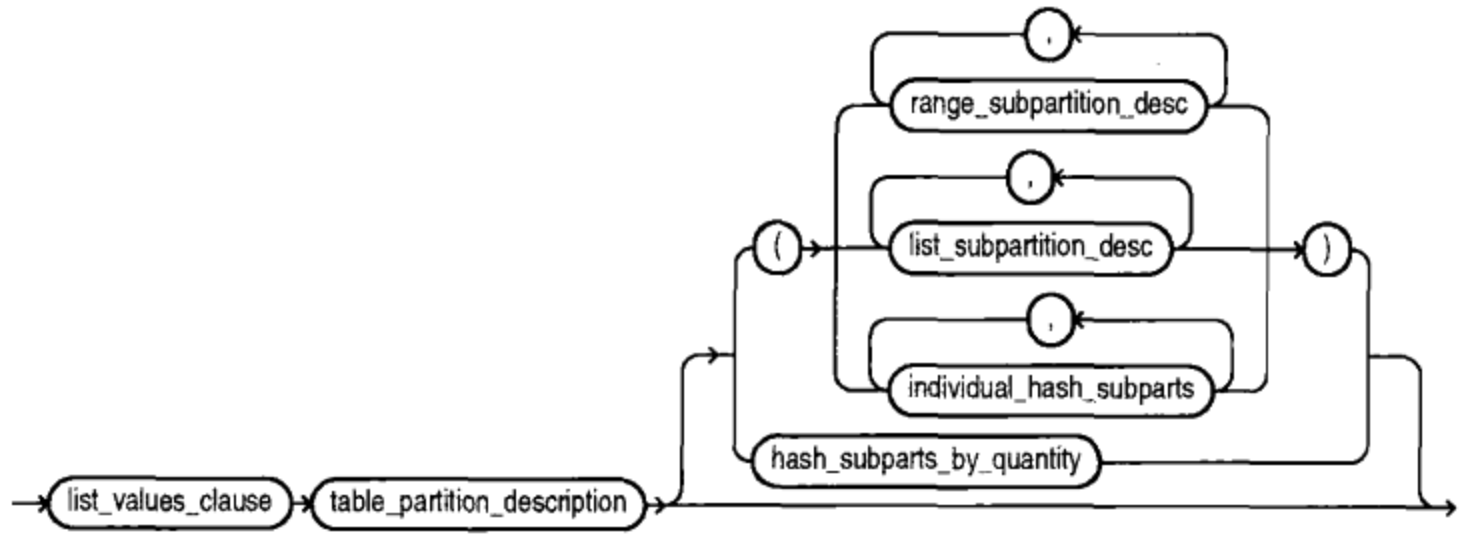
system_partitioning::=



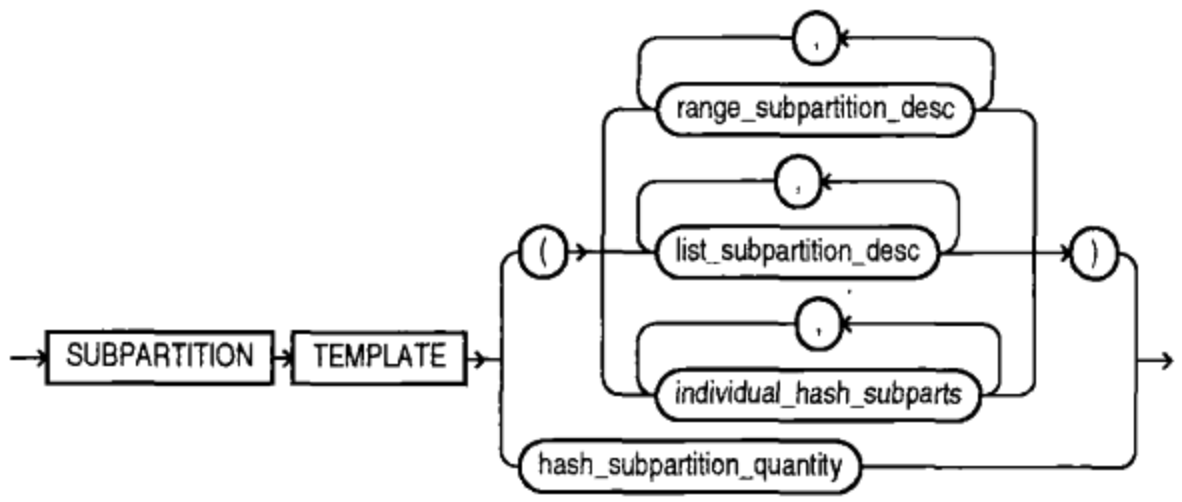
range_partition_desc::=



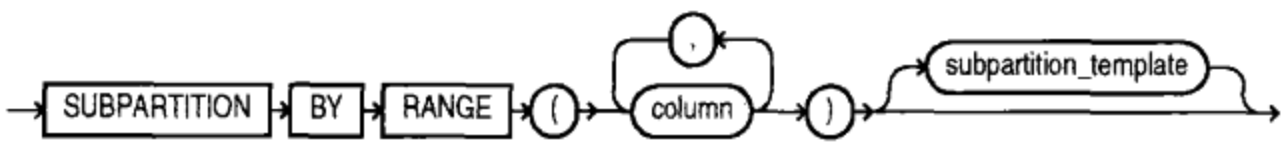
list_partition_desc::=



subpartition_template::=

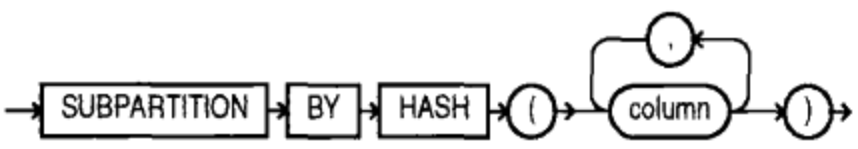


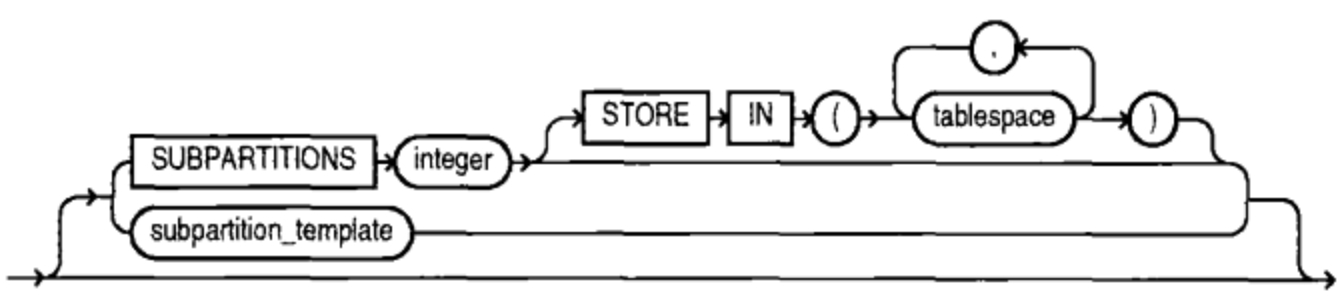
subpartition_by_range::=



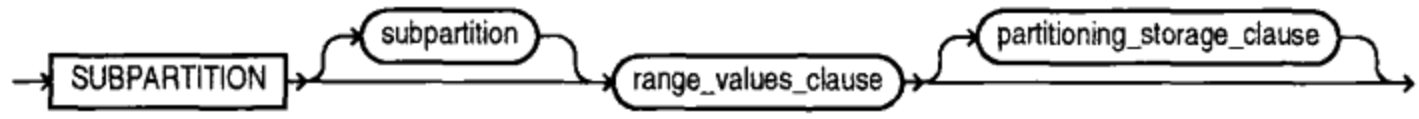
subpartition_by_list::=

subpartition_by_hash::=

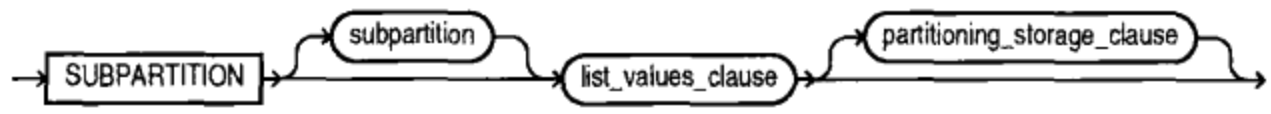




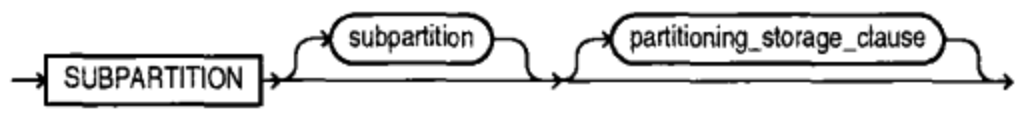
range_subpartition_desc::=



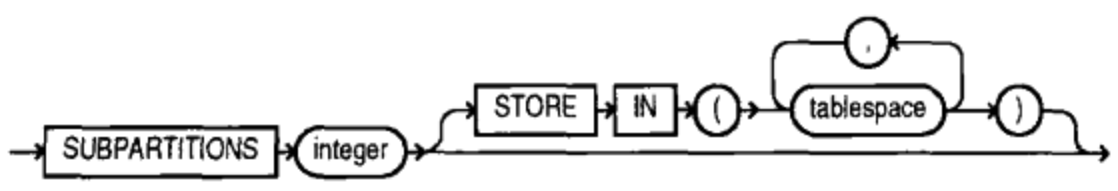
list_subpartition_desc::=



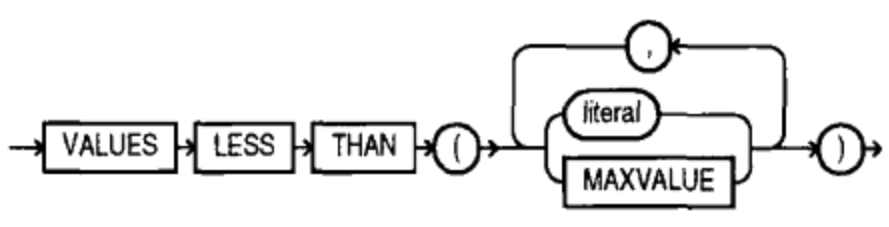
individual_hash_subparts::=



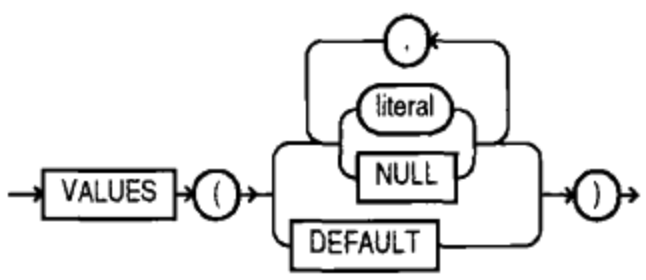
hash_subparts_by_quantity::=



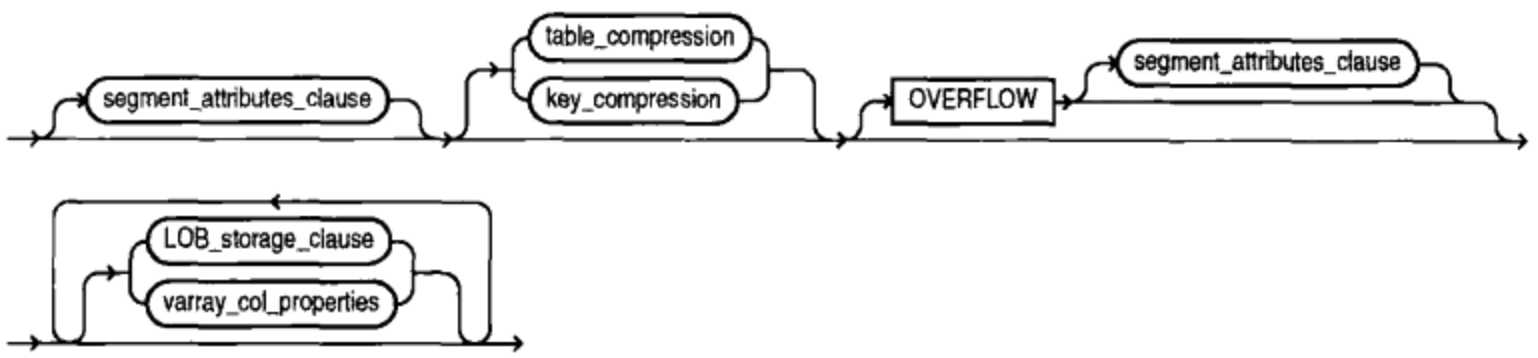
range_values_clause::=



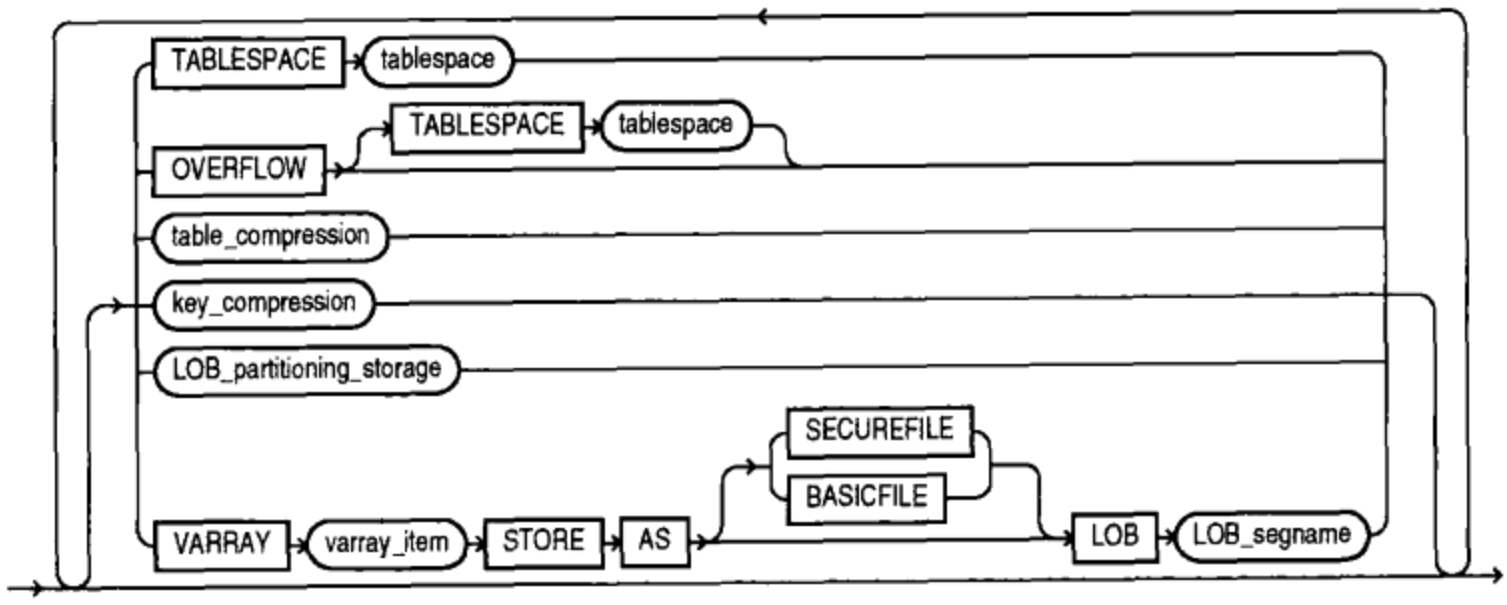
list_values_clause::=



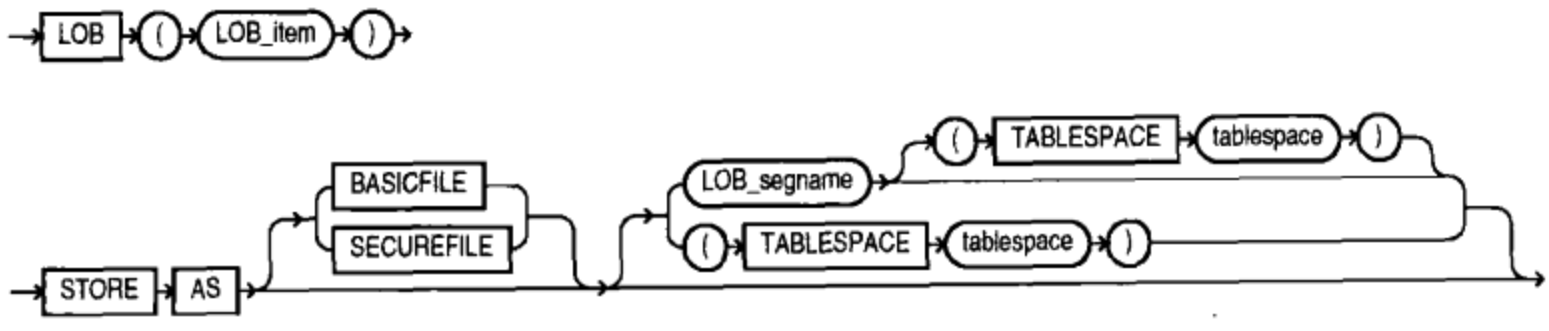
table_partition_description::=



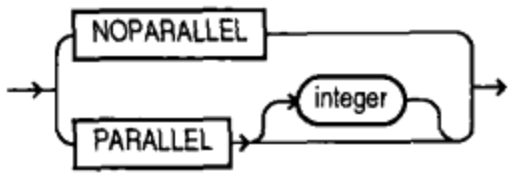
partitioning_storage_clause::=



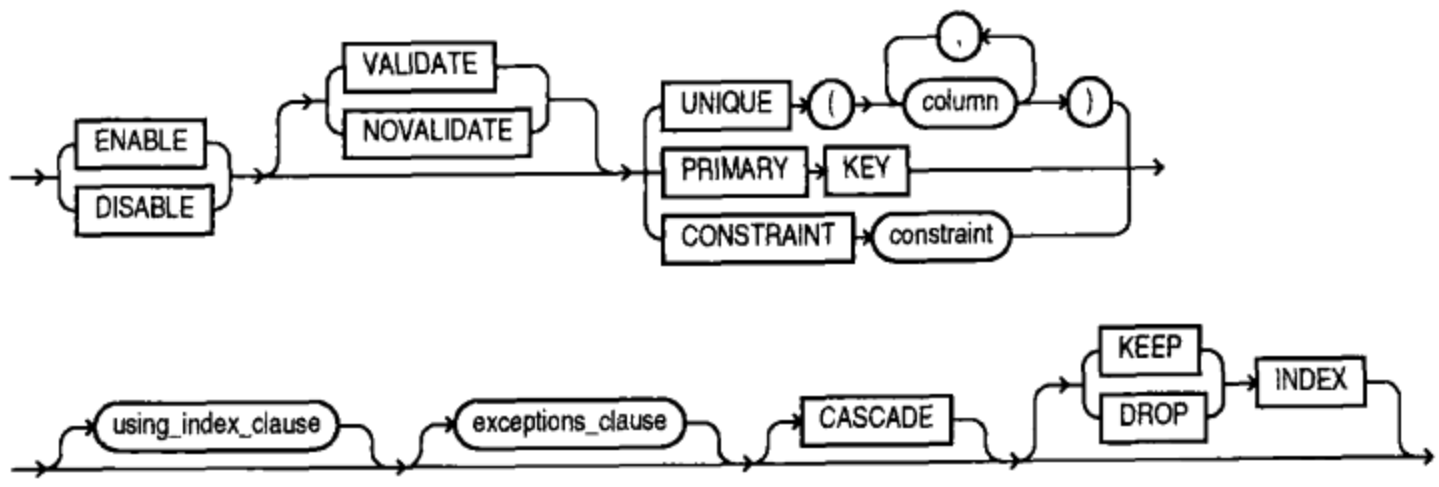
LOB_partitioning_storage::=



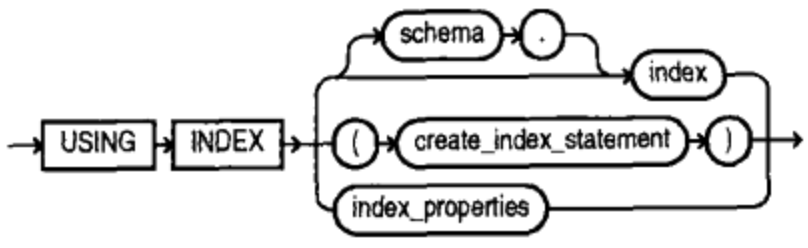
parallel_clause::=



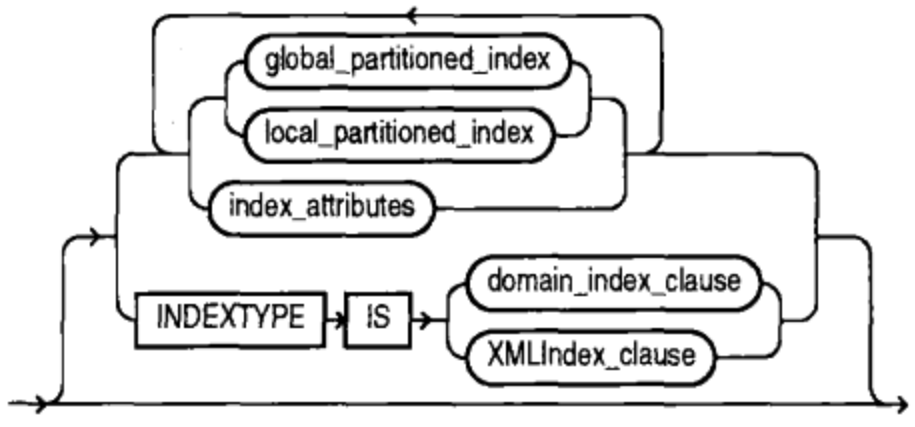
enable_disable_clause::=



using_index_clause::=



index_properties::=



index_attributes::=

INITRANS 指定可以同时更新一个数据块的初始事务的个数(select 不包含在内)。INITRANS 的范围为 1~255。1 为默认值。每个事务都在该数据块本身中占用一定的空间，直到事务完成为止。当为该块创建的事务的个数大于 INITRANS 的值时，将自动为它们分配空间。排队等候执行的事务将在该块中占用越来越多的可用空间，尽管通常可用空间比所有并发事务所需要的空间多。

当 Oracle 将一行插入表中时，它首先检查当前块中有多少可用空间(块的大小依赖于操作系统，请参阅系统的 Oracle Installation and User's Guide(Oracle 安装说明和用户指南))。如果行的大小使剩下的可用空间小于该块的 PCTFREE 百分比，则 Oracle 将该行放在一个新分配的块中。默认值为 10，0 为最小值。

PCTUSED 的默认值为 40。这是一个块中可用空间的最小百分数，该值使它作为在这个块中插入新行的候选空间。Oracle 通过删除操作跟踪块中有多少空间可用。如果它低于 PCTUSED，则 Oracle 将使该块可用于插入操作。

STORAGE 包含 STORAGE 下描述的子句。TABLESPACE 是分配给该表的表空间名。

ENABLE 和 DISABLE 子句分别启用和禁用约束。

PARALLEL 和 DEGREE 与 INSTANCES 一起指定表的并行特征。DEGREE 指定要使用的查询服务器的数目，INSTANCES 指定在表与一个用于并行查询处理的实时应用群集的实例之间如何分离。整数 n 指定该表将在指定数目的可用实例间分割。

PARTITION 子句控制表数据的分区方式，可以按范围、按散列或按多个分区方法的组合方式进行分区。关于分区的详细内容请参阅第 18 章。

临时表包含只对于创建它的会话可视的数据。如果一个临时表定义为 GLOBAL，则其定义可以被所有用户看到。临时表不能分区，而且不支持许多标准的表功能(如可变数组数据类型、LOB 数据类型和索引组织)。例如，以下命令创建了一个表，表中的行将在当前会话结束时被删除：

```
create global temporary table WORK_SCHEDULE (
       StartDate  DATE,
       EndDate    DATE,
       PayRate    NUMBER)
       on commit  preserve rows;
```

AS 子句通过返回的查询记录创建新表的行。该查询的列和类型必须与 CREATE TABLE 语句中定义的那些列和类型相匹配。在 AS 子句中使用的查询不能包含任何 LONG 数据类型的列。可以在 create table…as select 操作过程中使用 NOLOGGING 关键字关闭日志记录。该选项禁用重做日志项(这些项通常在填充新表期间写入)，因而在改进性能的同时影响了恢复数据的能力(如果一个实例故障发生在下一次备份以前的话)。

LOB 子句指定用于内部存储 LOB(BLOB、CLOB 和 NCLOB)数据的外部数据存储。对象表的定义部分可应用于对象表，在对象表中，每一行都有一个对象 ID(OID)。

在 Oracle Database 11g 中，CREATE TABLE 有如下几方面的增强：

- flashback_archive_clause 子句允许创建启用了跟踪历史变更的表
- system_partitioning 子句允许使用 BY SYSTEM 对表进行分区
- 可以创建虚拟列

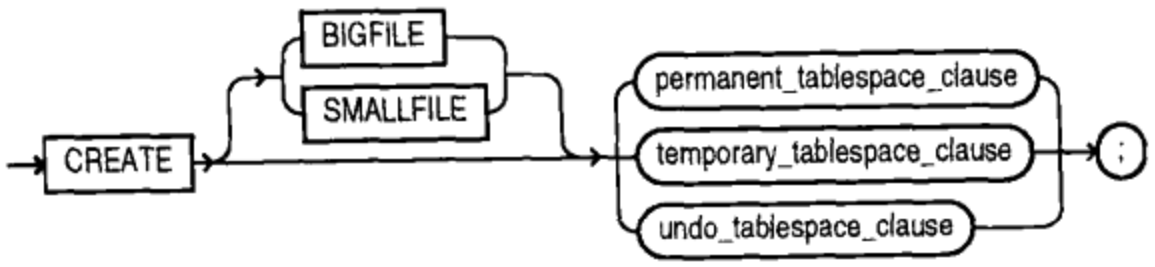
- 可以以二进制 XML 格式存储 XML 数据
- 可以通过引用一个分区表对另一个表进行分区
- LOB 存储现在包括 SECUREFILE 参数、压缩选项、重复数据删除选项和加密选项

CREATE TABLESPACE

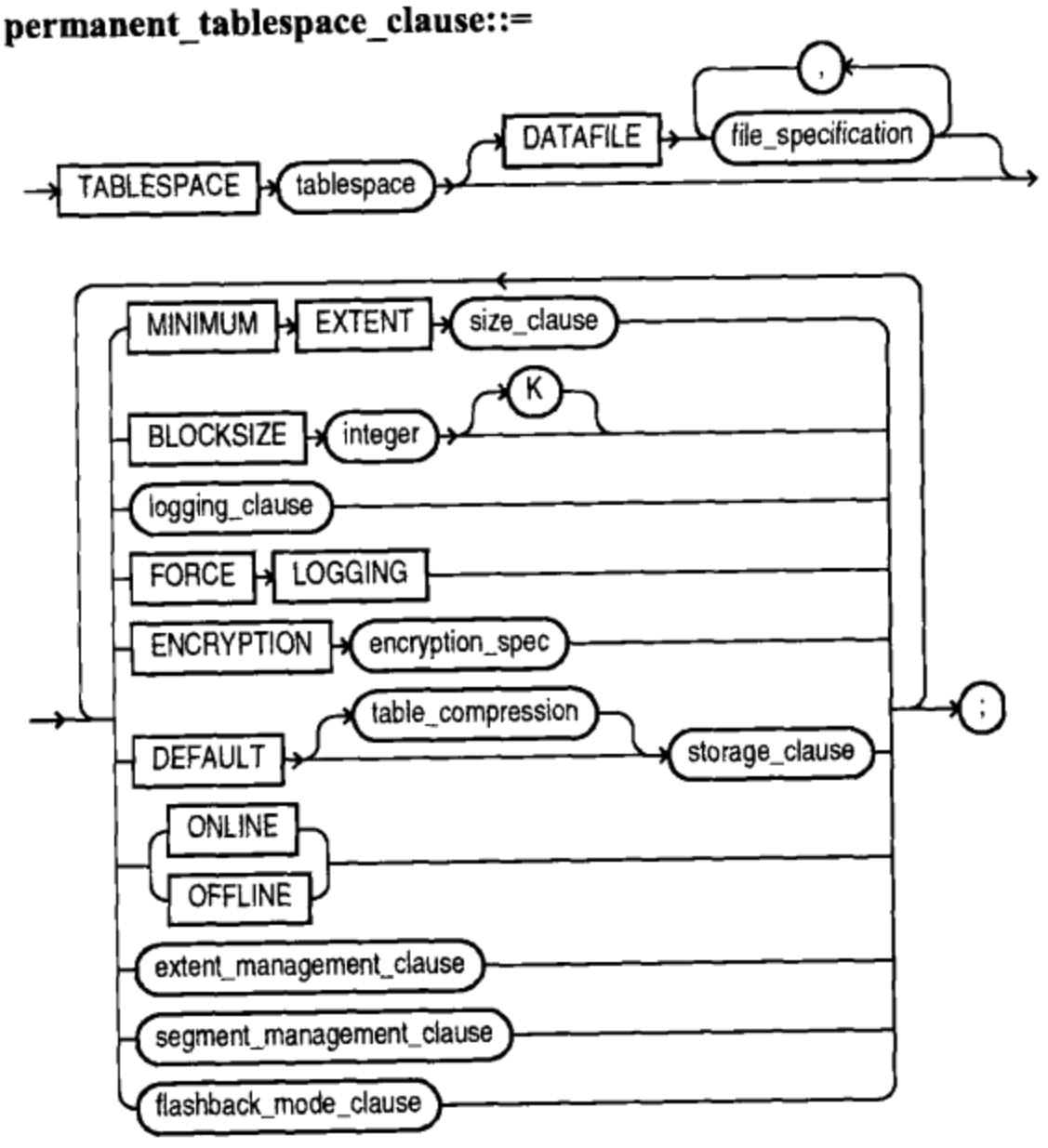
参阅： ALTER TABLESPACE、DROP TABLESPACE、第 22 章和第 51 章。

格式：

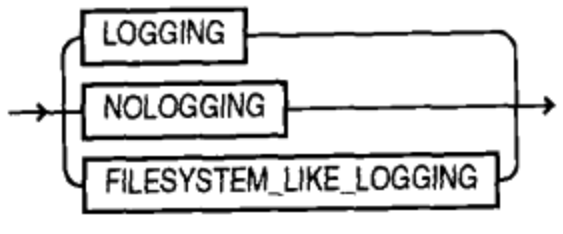
create_tablespace::=



permanent_tablespace_clause::=

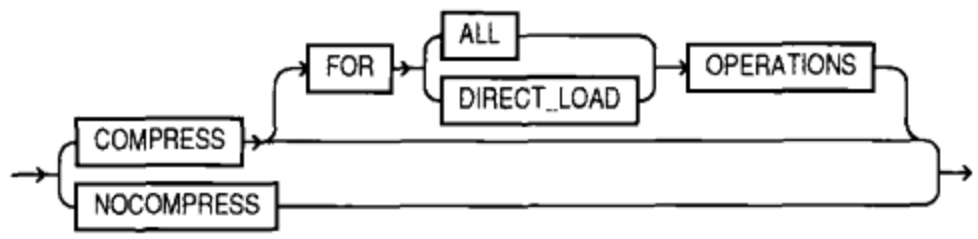


logging_clause::=

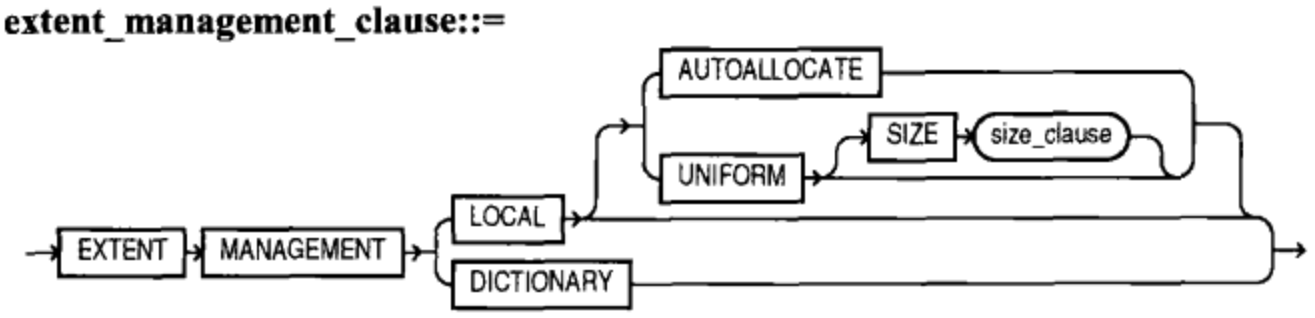


encryption_spec::=

table_compression::=

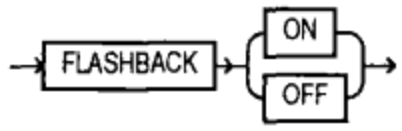


extent_management_clause::=

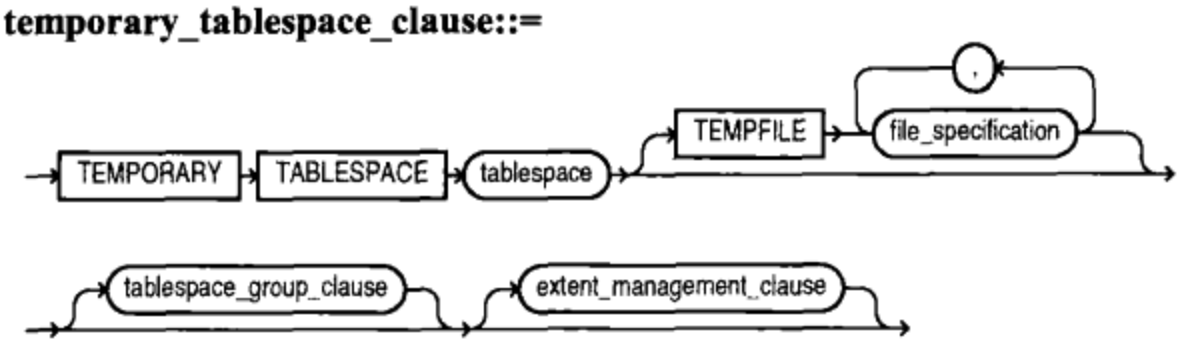


segment_management_clause::=

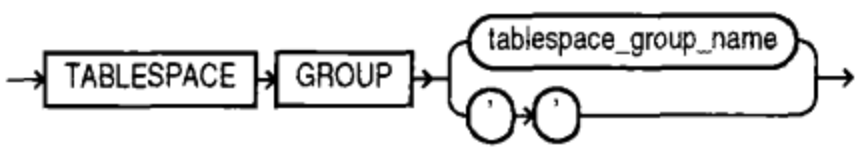
flashback_mode_clause::=



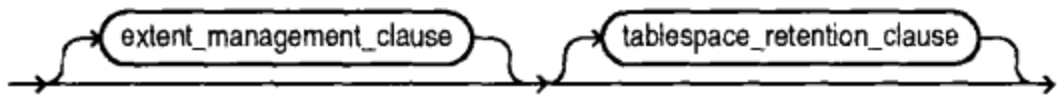
temporary_tablespace_clause::=



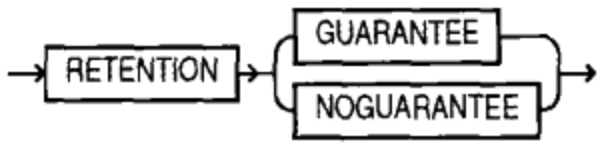
tablespace_group_clause::=



undo_tablespace_clause::=



tablespace_retention_clause::=



描述：tablespace 是表空间名，遵循 Oracle 的命名约定。DATAFILE 是按照 file_definition 描述的一个文件或一系列文件，它定义文件名和文件大小：

```
'file' [SIZE integer [K | M | G | T] [REUSE]
```

文件格式由操作系统指定。SIZE 是为该文件预留的字节数，以千字节(K)、兆字节(M)、千兆字节(G)和百万兆字节(T)为单位。DEFAULT STORAGE 定义所有在该表空间中创建的对象的默认存储，除非这些默认值在 CREATE TABLE 命令自身中被重写。默认值 ONLINE 表明该表空间一创建完成就对用户可用。OFFLINE 则禁止访问该表空间，直到 ALTER TABLESPACE 将它修改为 ONLINE 为止。DBA_TABLESPACES 给出所有表空间的状态。

SIZE 和 REUSE 一起告诉 Oracle，如果文件已经存在，就重用该文件(这时该文件包含的任何内容将被删除)；如果不存在则创建该文件。如果不使用 REUSE，SIZE 就创建一个不存在的文件，但如果该文件存在则将返回一个错误。如果不使用 SIZE，则该文件必须已经存在。

MINIMUM EXTENT 大小参数指定表空间中最小的区。默认值不仅特定于操作系统而且特定于数据库块的大小。

当设置为状态 ON 时，AUTOEXTEND 选项将根据需要按 NEXT 大小递增，动态地扩展一个数据文件，最大值为 MAXSIZE(或 UNLIMITED)。如果表空间只用于在查询处理中所创建的临时段，则可以以 TEMPORARY 表空间创建它。如果该表空间要包含永久对象(如表)，则应当使用默认设置 PERMANENT 选项。

EXTENT MANAGEMENT 控制记录表空间的区管理数据的方法。默认情况下，区的使用为 LOCAL——区的位置信息存储在表空间的数据文件的位图中。

可以使用 BIGFILE 子句创建一个大文件表空间。同样也可以把一个临时表空间分配给临时表空间组(通过 TABLESPACE GROUP 子句)。对撤消表空间，可以保证未过期的数据即使以正在进行的需要撤消段空间事务为代价，也将被保存(通过 RETENTION GUARANTEE)。

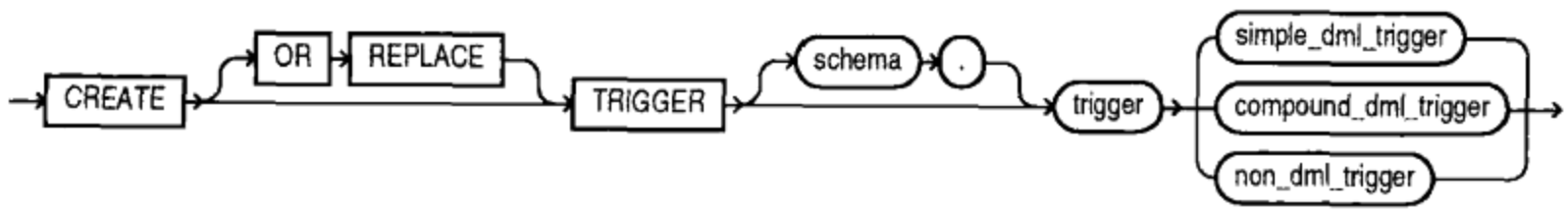
在 Oracle Database 11g 中，可以使用 ENCRYPT 关键字加密整个表空间。

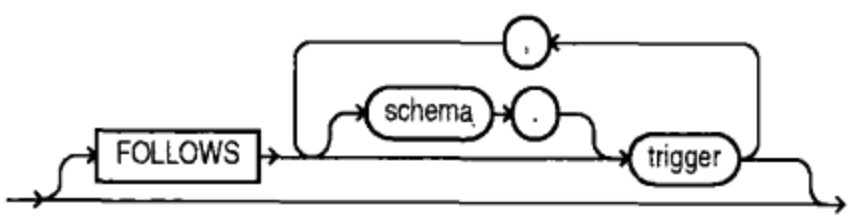
CREATE TRIGGER

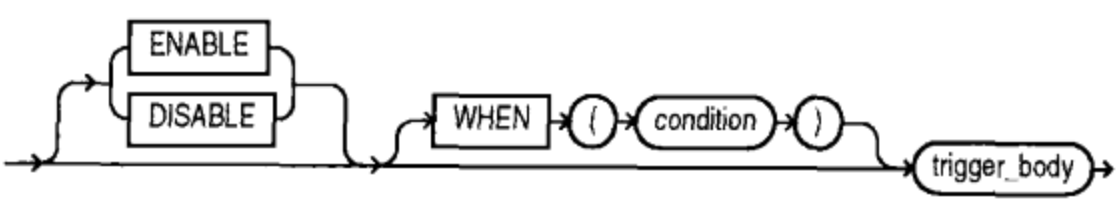
参阅：ALTER TABLE、ALTER TRIGGER、BLOCK STRUCTURE、DROP TRIGGER、第 34 章和第 38 章。

格式：

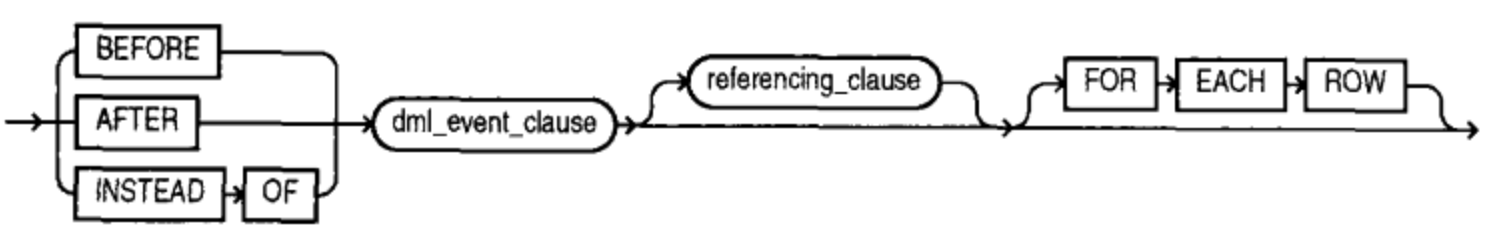
create_trigger::=



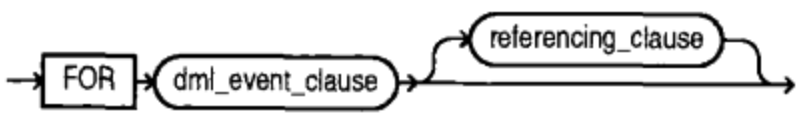




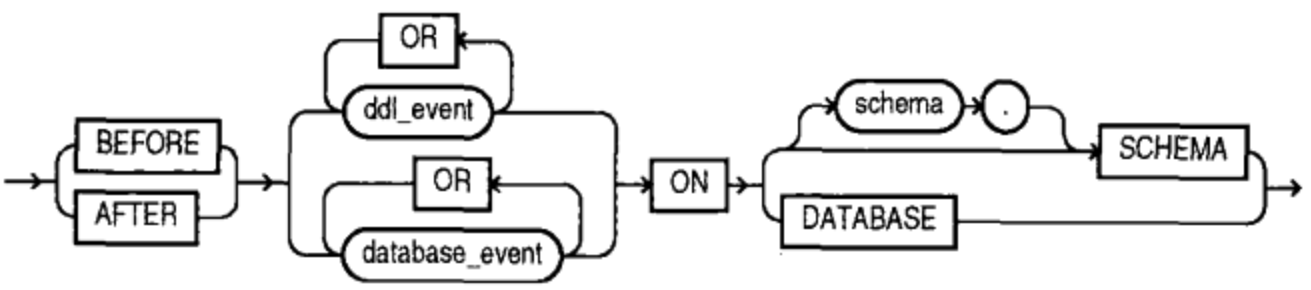
simple_dml_trigger::=



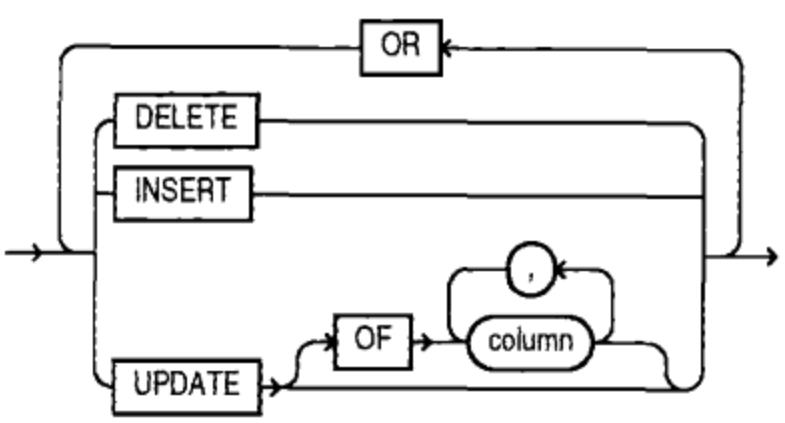
compound_dml_trigger::=

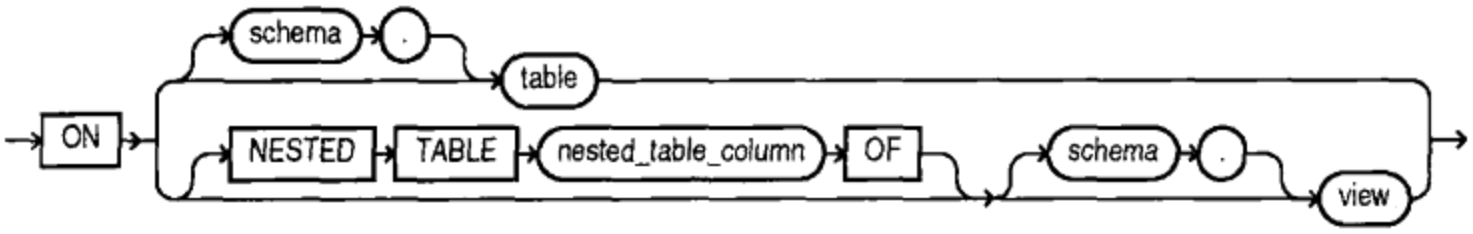


non_dml_trigger::=

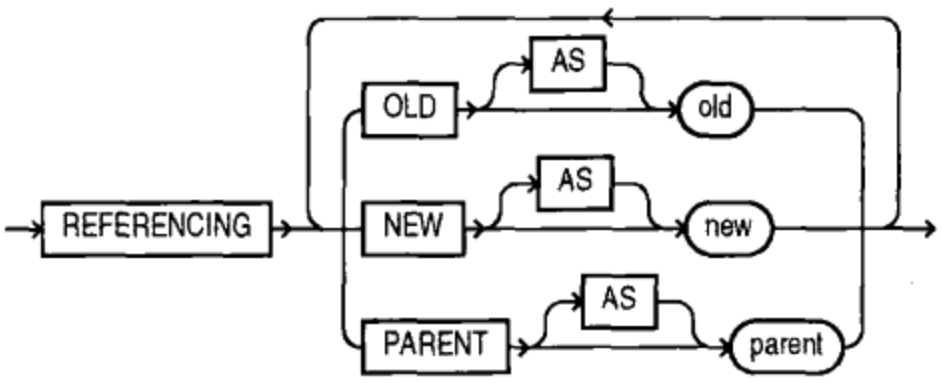


dml_event_clause::=





referencing_clause::=



描述：CREATE TRIGGER 创建并启用一个数据库触发器，即与表关联的一个存储过程块，它在 ON 子句中指定，当在该表上执行指定的 SQL 语句时，Oracle 将自动执行该触发器。可以使用触发器施加复杂的约束，也可以使用触发器在数据库中传播更改，而不是在应用程序中进行更改。以此种方式，触发器代码只需要实现一次，而不必在每个应用程序中都实现。

CREATE TRIGGER 命令的主子句指定触发该块的 SQL 操作(如 DELETE、INSERT 或 UPDATE)以及何时触发激活器(BEFORE、AFTER 或 INSTEAD OF 执行触发器操作)。如果在 UPDATE 触发器中指定一个 OF 子句，则触发器只在更新一个或多个指定的列时触发。请注意，还可以创建在 DDL 事件发生时激活的触发器(包括 CREATE 、ALTER 和 DROP)以及在指定数据库事件发生时激活的触发器(登录、注销、数据库启动、关闭和服务器错误)。

REFERENCING 子句指定表的 OLD 和 NEW 版本的相关表名。可以在引用列时使用该名字，以免混淆，尤其是在表名为 OLD 或 NEW 时。默认的名称为 OLD 和 NEW。

FOR EACH ROW 子句指定触发器为行触发器，即对受触发器操作影响的每一行都激活一次的触发器。WHEN 子句限制触发器的执行，仅当满足 condition 时，才执行该触发器。该条件是一个 SQL 条件，而不是 PL/SQL 条件。

可以通过 ALTER TRIGGER 和 ALTER TABLE 命令禁用和启用触发器。如果禁用一个触发器，则当潜在的触发操作发生时，Oracle 不激活该触发器。CREATE TRIGGER 命令自动启用触发器。

要在自己拥有的表上创建触发器，必须拥有 CREATE TRIGGER 系统权限。要在另一个用户的表中创建触发器，必须拥有 CREATE ANY TRIGGER 系统权限。必须直接授予用户在执行存储过程时所需的所有权限。

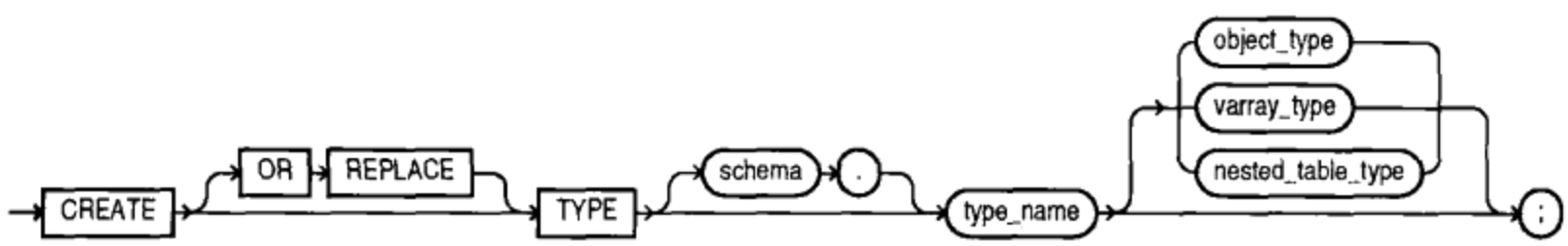
在 Oracle Database 11g 中，CREATE TRIGGER 允许通过指定一个多部分的 PL/SQL 块来创建复合触发器。新的 FOLLOWS 子句允许对多个触发器进行排序。可以使用 ENABLE 和 DISABLE 子句创建启用或禁用形式的触发器。

CREATE TYPE

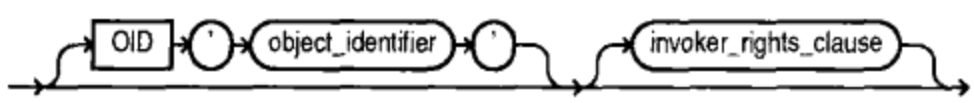
参阅：CREATE TABLE、CREATE TYPE BODY、第 38 章和第 30 章。

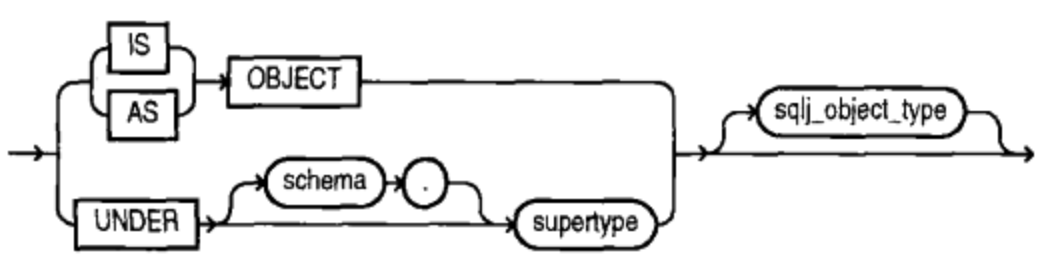
格式：

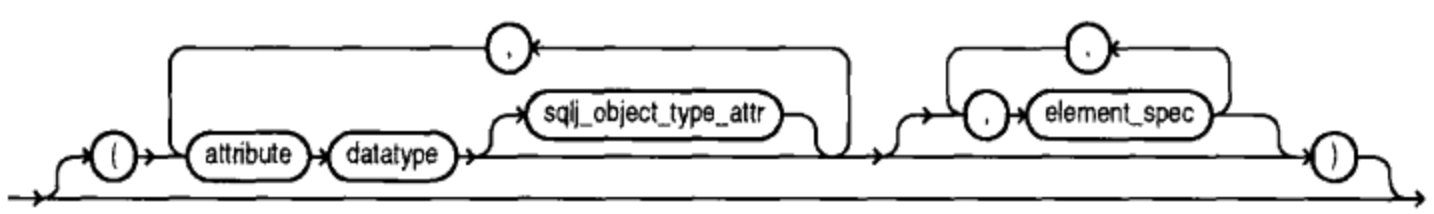
create_type::=

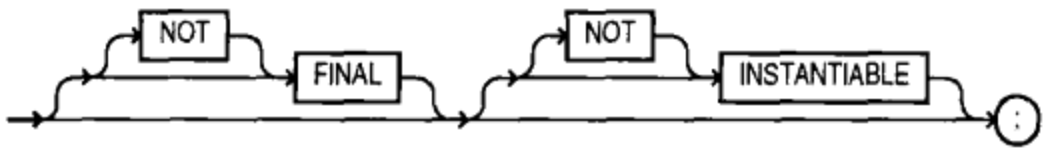


object_type::=

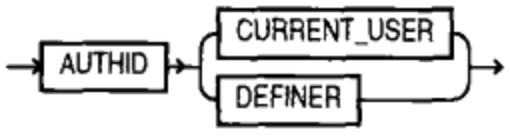




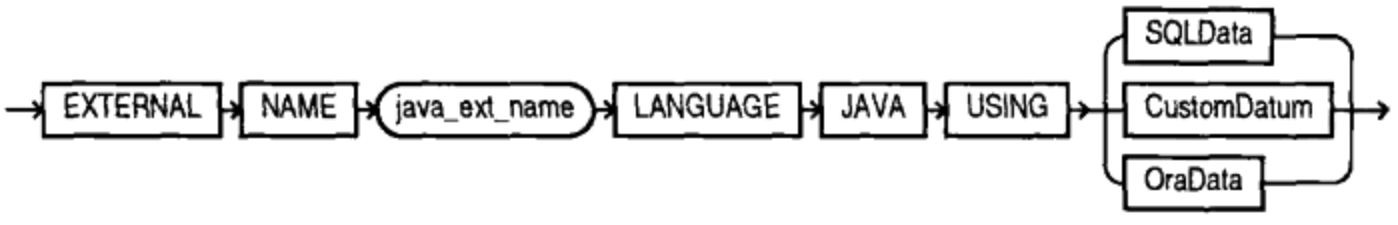




invoker_rights_clause::=

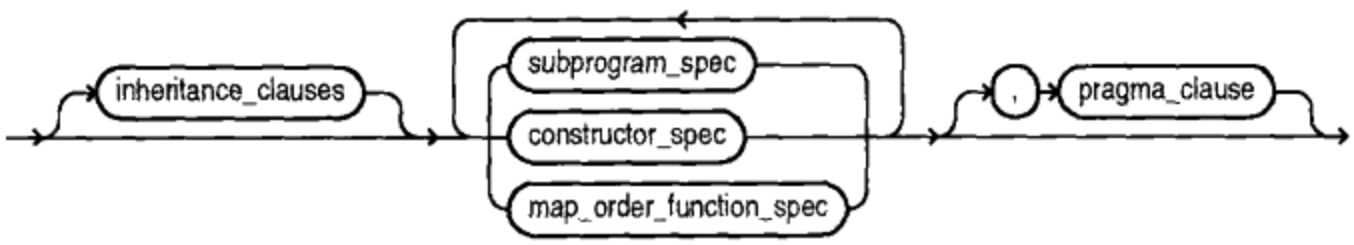


sqlj_object_type::=

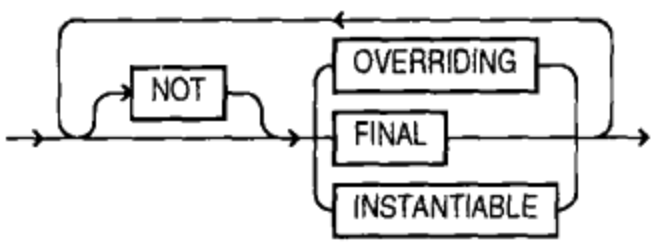


sqlj_object_type_attr::=

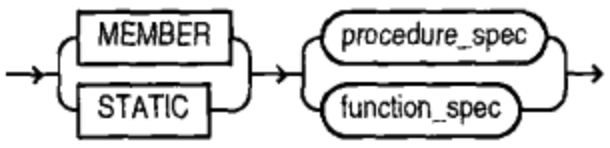
element_spec::=



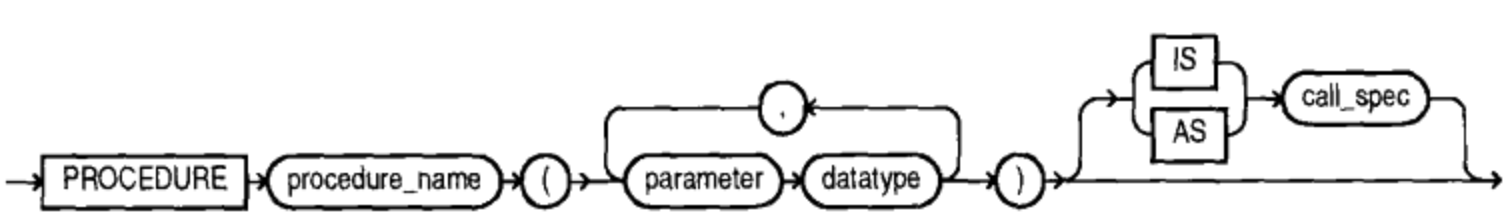
inheritance_clauses::=



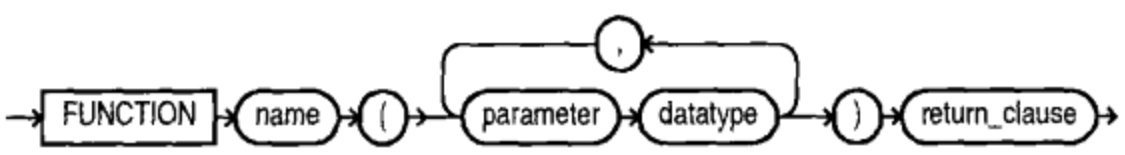
subprogram_spec::=



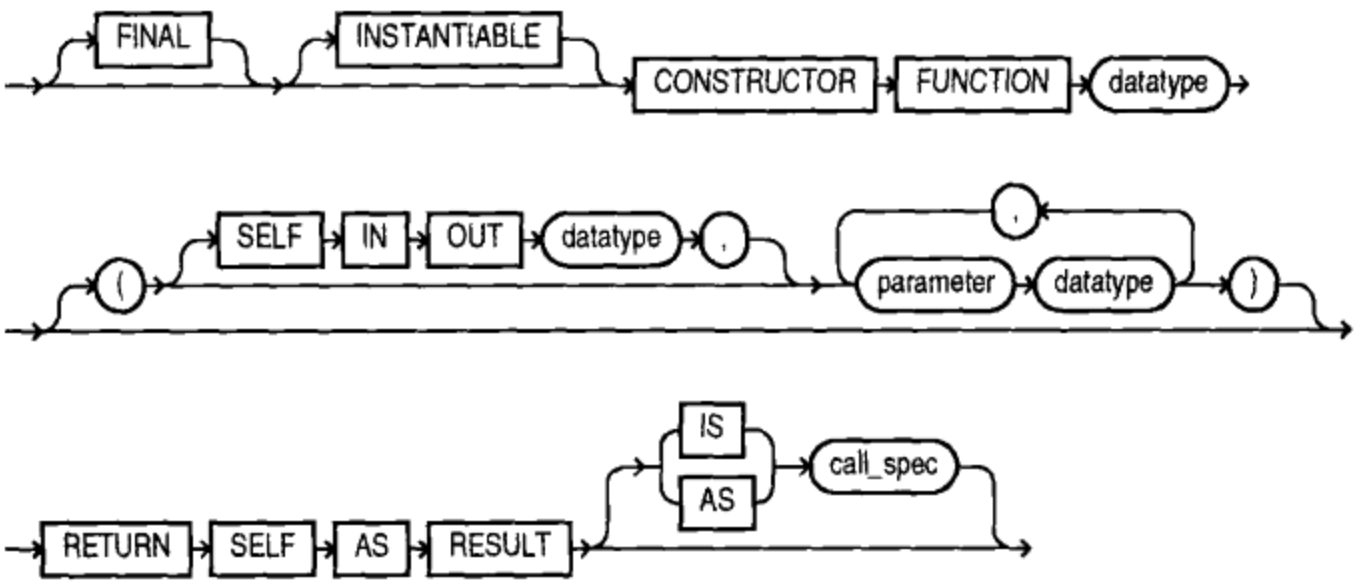
procedure_spec::=



function_spec::=

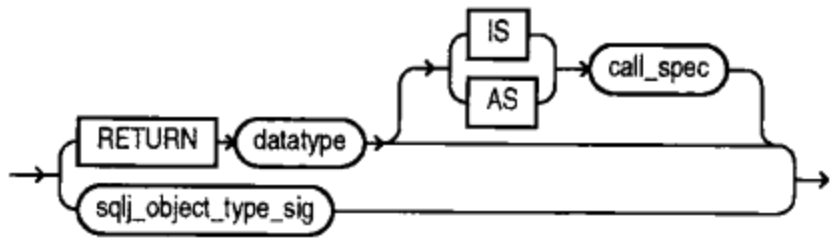


constructor_spec::=

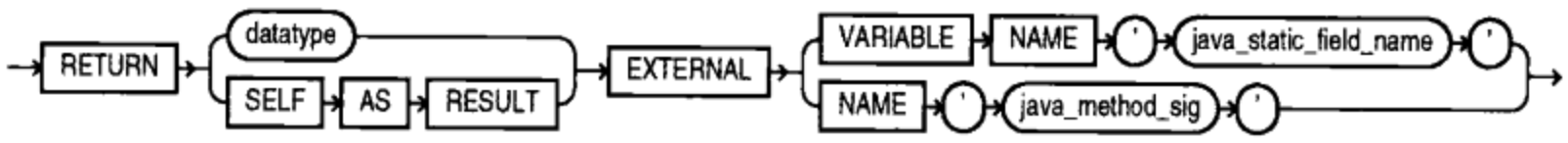


map_order_function_spec::=

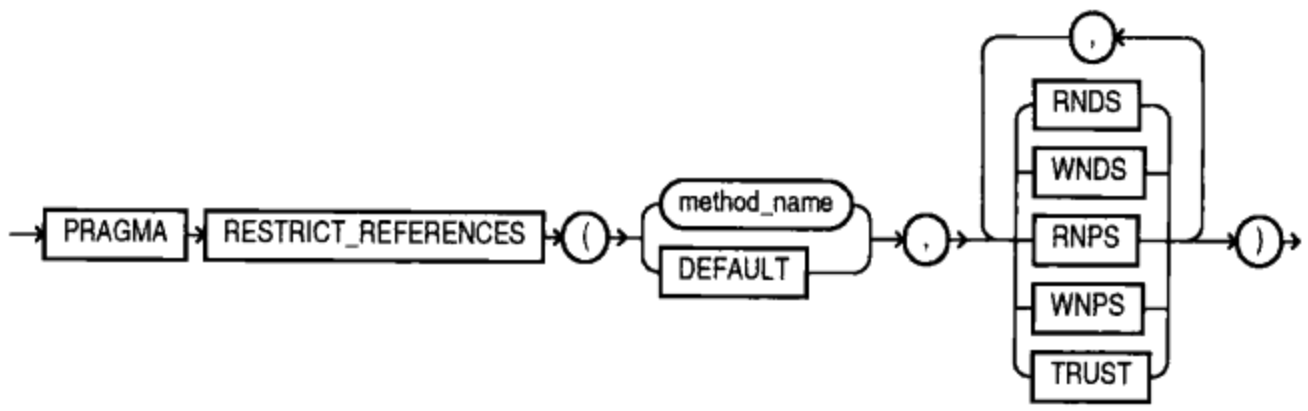
return_clause::=



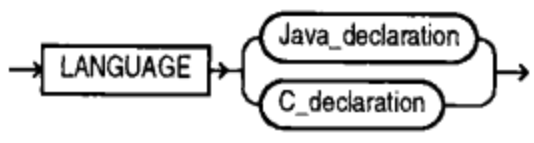
sqlj_object_type_sig::=



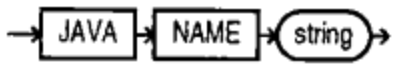
pragma_clause::=



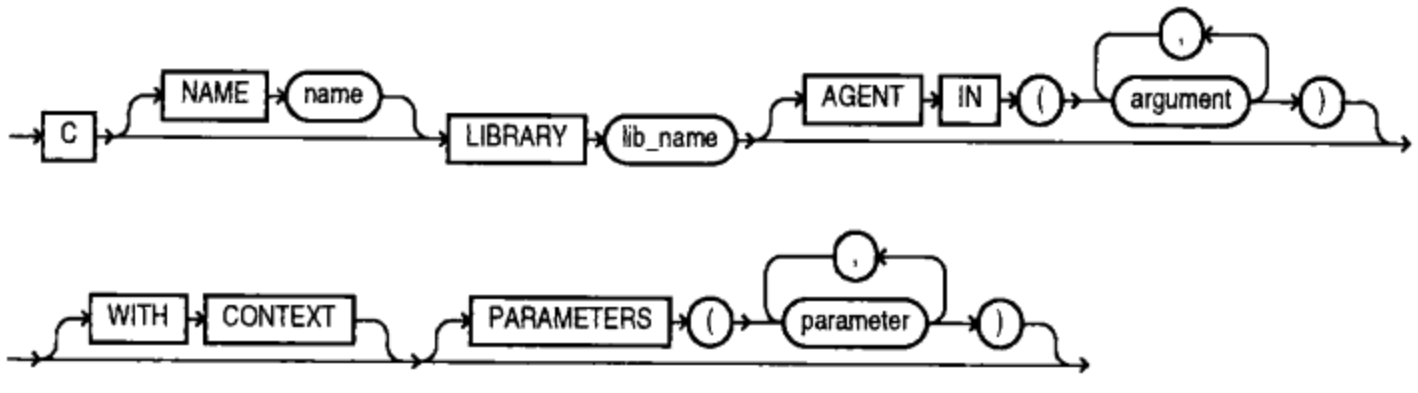
call_spec::=



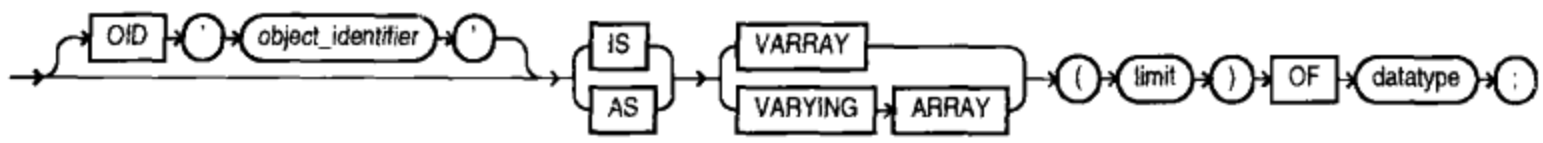
Java_declaration::=



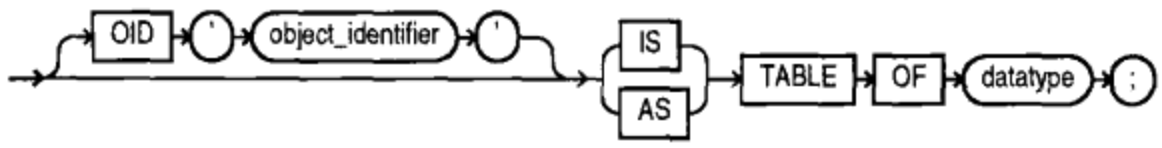
C_declaration::=



varray_type::=



nested_table_type::=



描述：CREATE TYPE 创建一个抽象数据类型、指定的可变数组(VARRAY)、嵌套表类型或不完整的对象类型。要创建或替换一个类型，必须拥有 CREATE TYPE 权限。要在另一个用户的模式中创建类型，必须拥有 CREATE ANY TYPE 系统权限。

在创建抽象数据类型时，可以以 Oracle 提供的数据类型(如 DATE 和 NUMBER)和以前定义的抽象数据类型为基础。

一个“不完整”的类型有名称但没有属性或方法。然而，它可以由其他对象类型引用，并可以用来定义相互引用的对象类型。如果有两个相互引用的类型，则必须将其中一个创建成不完整的类型，然后创建第二个类型，最后用其正确的定义重新创建第一个类型。

如果打算为类型创建方法，则需要在该类型规范中声明方法的名字。

示例：创建一个有相关方法的类型：

```
create or replace type ANIMAL_TY as object
(Breed      VARCHAR2(25),
 Name       VARCHAR2(25),
 BirthDate  DATE,
member function AGE (BirthDate IN DATE) return NUMBER,
PRAGMA RESTRICT_REFERENCES(AGE, WNDS));
```

本示例创建一个可变数组：

```
create or replace type TOOLS_VA as varray(5) of VARCHAR2(25);
```

本示例创建没有任何方法的类型：

```
create type ADDRESS_TY as object
(Street  VARCHAR2(50),
 City    VARCHAR2(25),
 State   CHAR(2),
 Zip     NUMBER);
```

本示例创建使用另一个类型的类型：

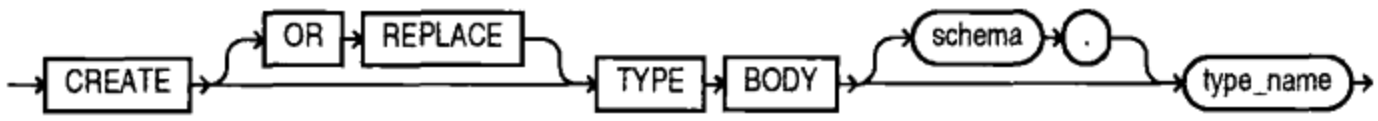
```
create type PERSON_TY as object
(Name     VARCHAR2(25),
 Address  ADDRESS_TY);
```

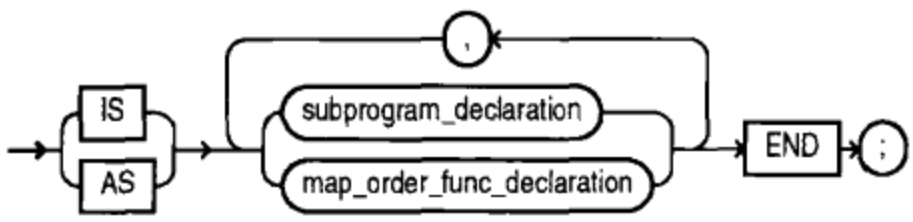
CREATE TYPE BODY

参阅：CREATE FUNCTION、CREATE PROCEDURE、CREATE TYPE、第 38 章、第 39 章和第 41 章。

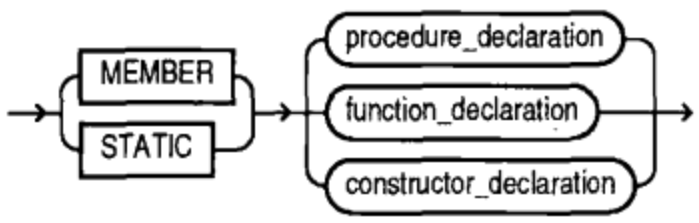
格式：

create_type_body::=

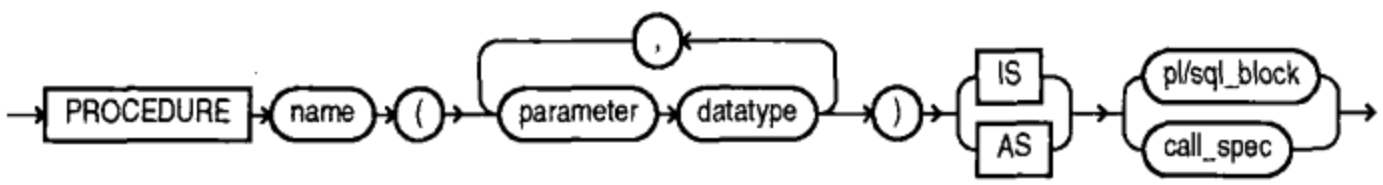




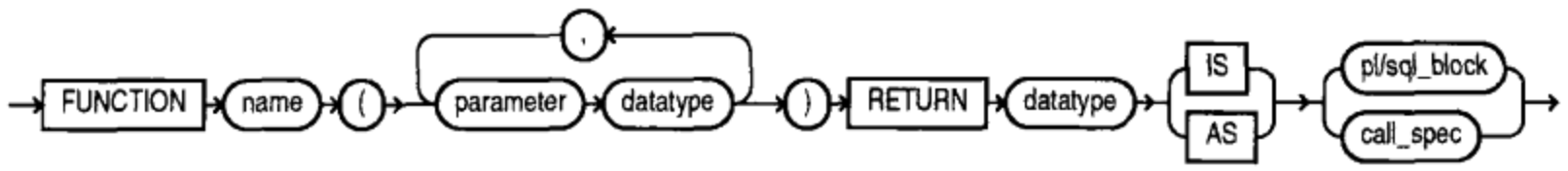
subprogram_declaration::=



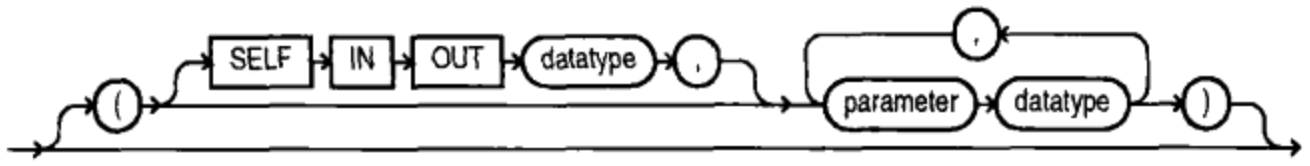
procedure_declaration::=

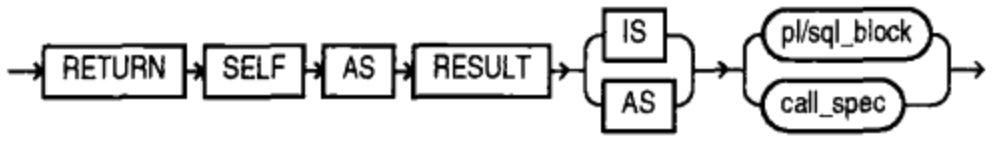


function_declaration::=



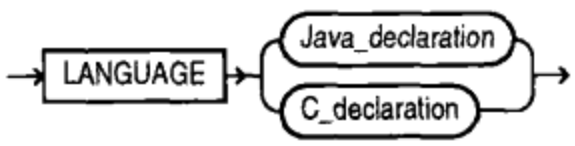
constructor_declaration::=





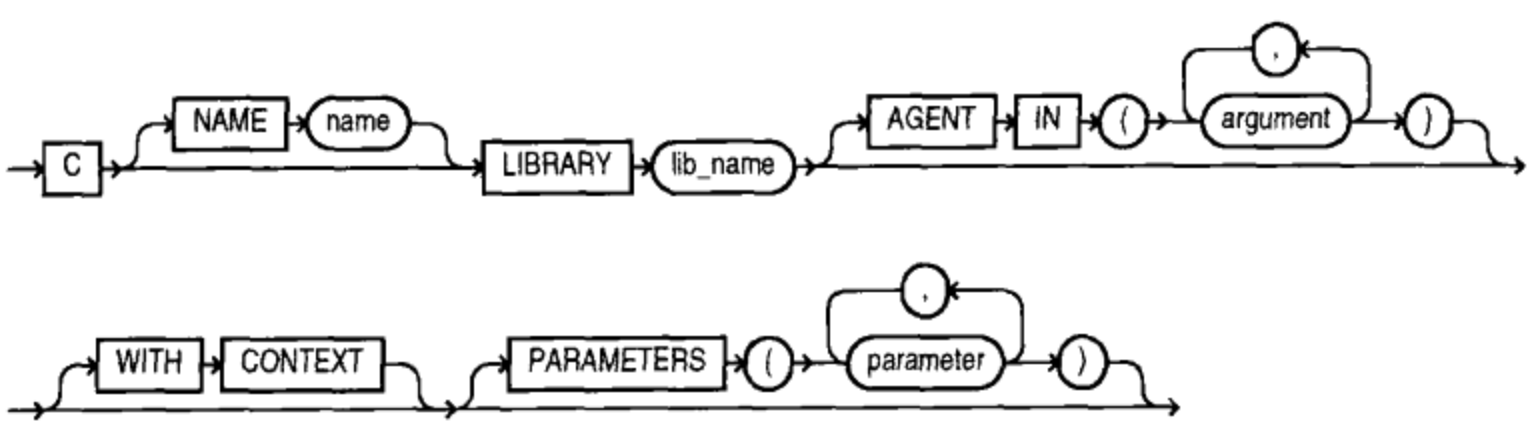
map_order_func_declaration::=

call_spec::=



Java_declaration::=

C_declaration::=



描述：CREATE TYPE BODY 指定通过 CREATE TYPE 创建的类型所应用的方法。要创建类型体，必须拥有 CREATE TYPE 权限。要在另一个用户的模式中创建类型体，必须拥有 CREATE ANY TYPE 权限。

示例：

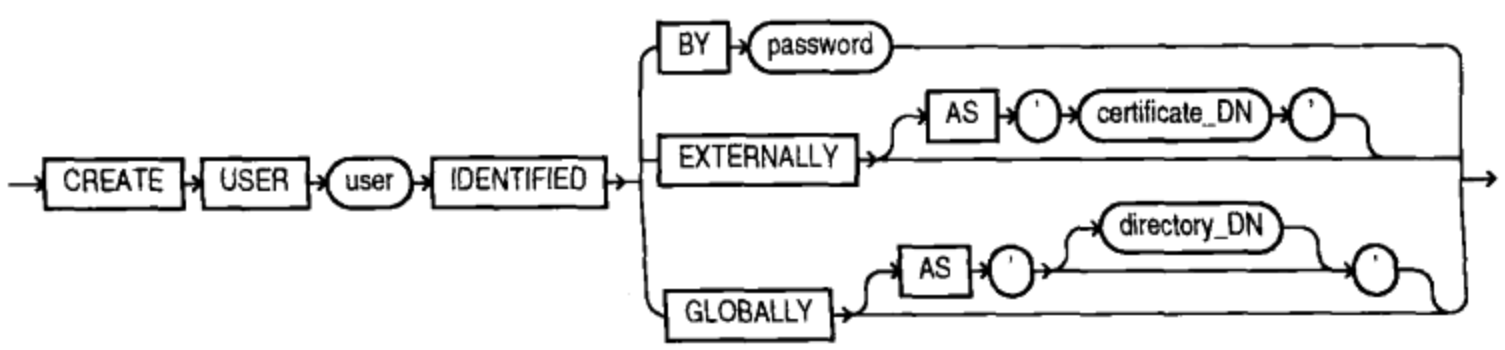
```
create or replace type body ANIMAL_TY as
  member function Age (BirthDate DATE) return NUMBER is
    begin
      RETURN ROUND(SysDate - BirthDate);
    end;
  end;
```

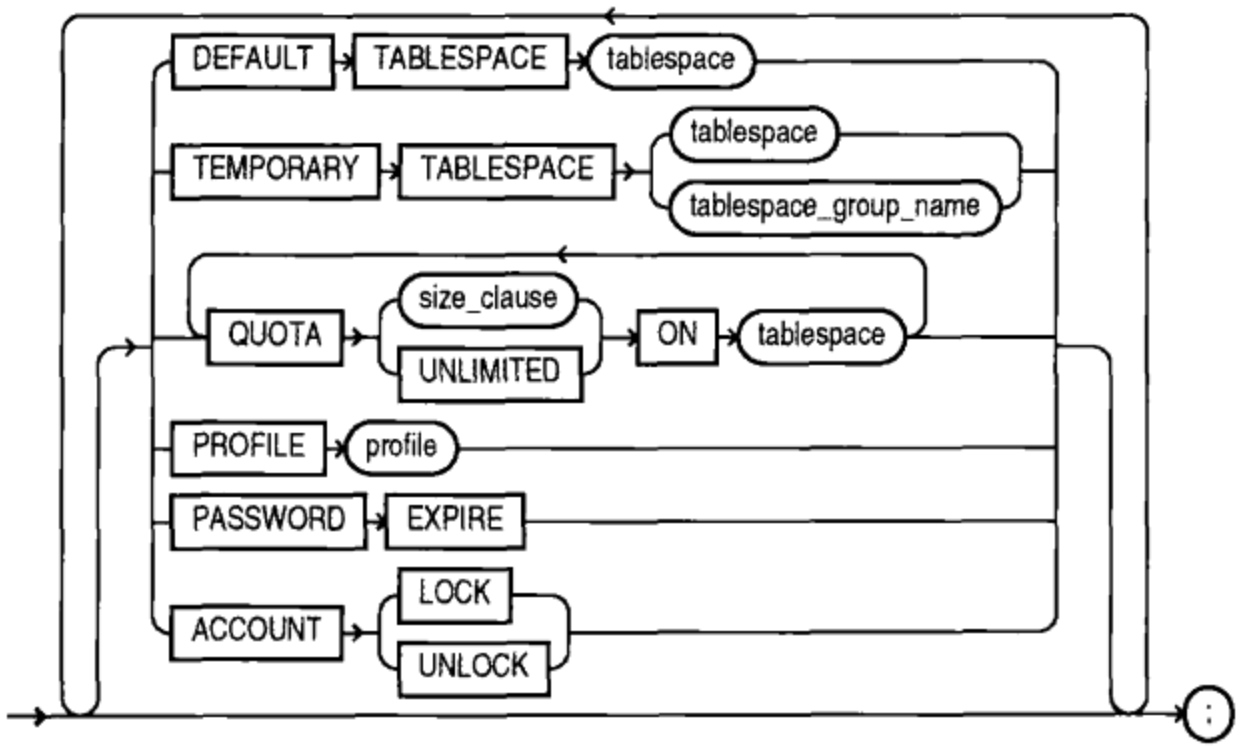
CREATE USER

参阅：ALTER USER、CREATE PROFILE、CREATE ROLE、CREATE TABLESPACE、GRANT 和第 19 章。

格式：

create_user::=





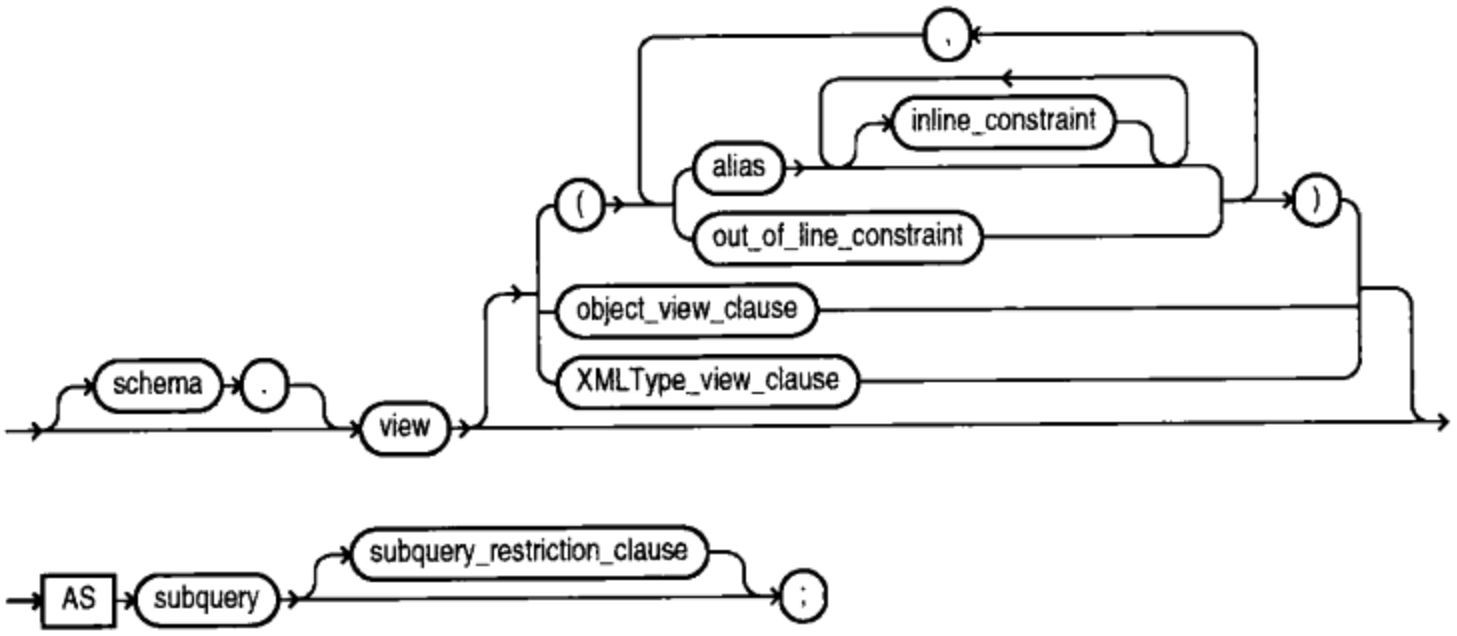
描述：CREATE USER 创建一个用户账户，允许使用某组权限和存储设置登录到数据库中。如果指定了一个口令，则必须将该口令用于登录。如果指定了 EXTERNALLY 选项，则访问操作要通过操作系统安全性的验证。外部验证使用 OS_AUTHENT_PREEIX 初始化参数在操作系统用户 ID 前面加前缀，因此在 CREATE USER 中指定的用户名应当包含前缀(通常为 OPS$)。

DEFAULT TABLESPACE 是用户在其中创建对象的表空间。TEMPORARY TABLESPACE 是为用户的操作创建临时对象的表空间。

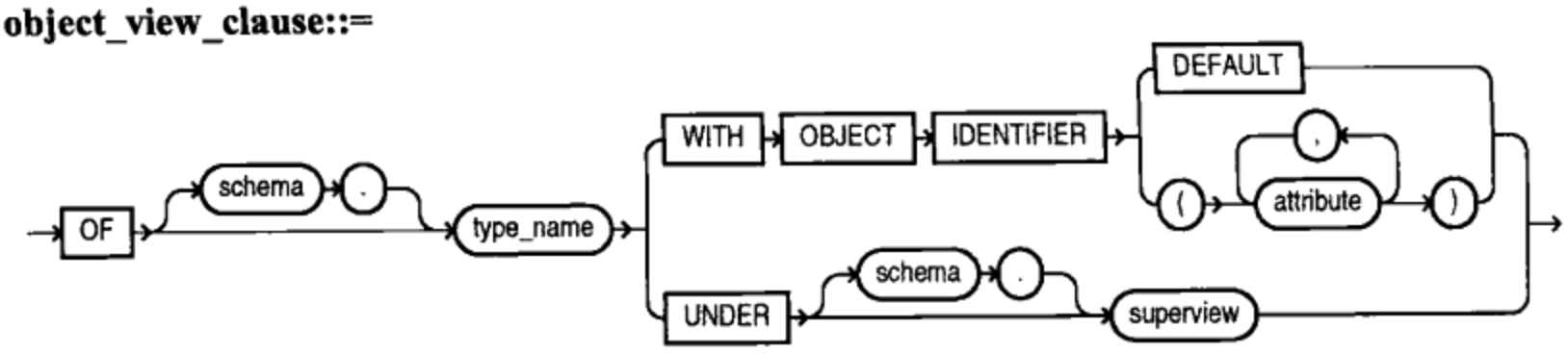
可以在这两个表空间的任意一个中放置一个 QUOTA，来限制空间的数量，用户可以以字节为单位(分别选择 K 或 M 选项，分别表示千字节和兆字节)进行分配。PROFILE 子句将一个指定的配置文件分配给用户，以限制数据库资源的使用。如果不指定配置文件，则 Oracle 将 DEFAULT 配置文件分配给用户。

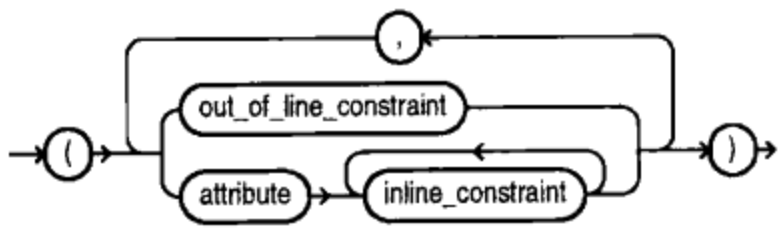
初次创建一个用户时，该用户没有权限。必须使用 GRANT 命令给该用户授予角色和权限。通常最少应该授予 CREATE SESSION 权限。

关于口令过期和账户锁定的信息，请参阅 CREATE PROFILE 和第 10 章

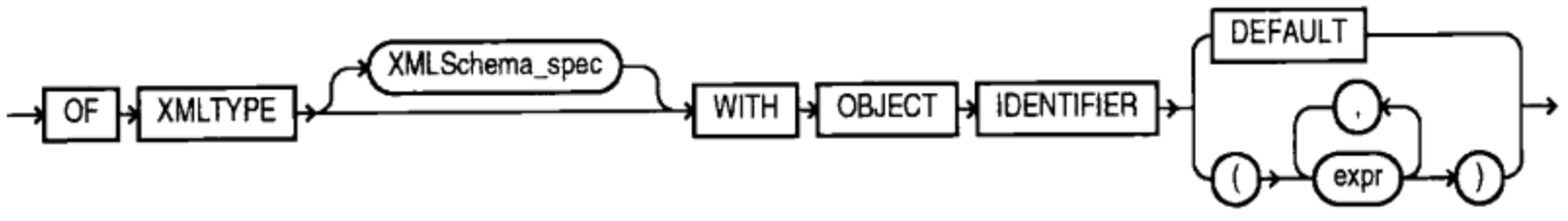


object_view_clause::=

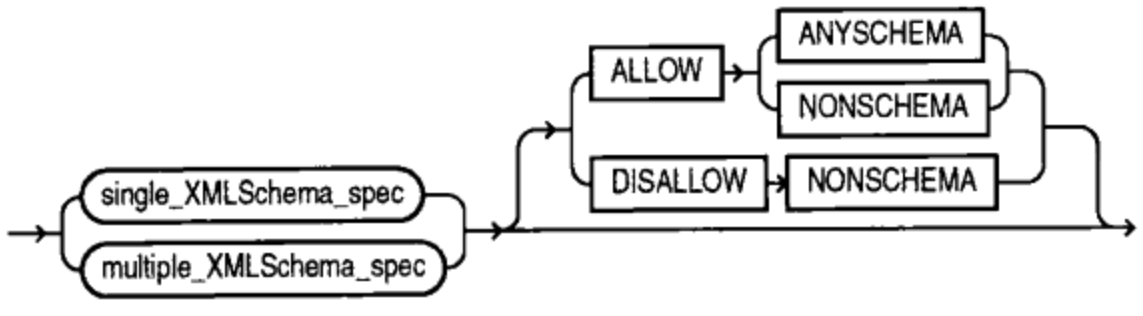




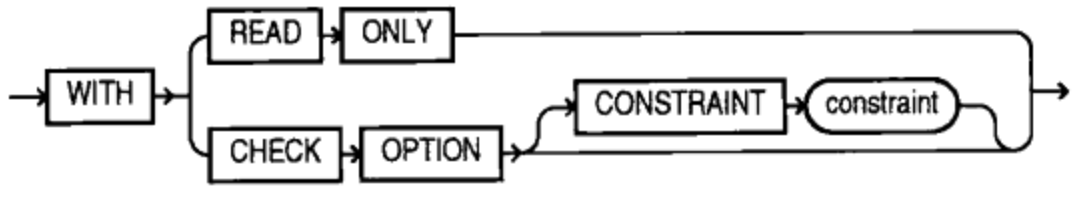
XMLType_view_clause::=



XMLSchema_spec::=



subquery_restriction_clause::=



描述：CREATE VIEW 定义名为 view 的视图。user 是创建视图的用户名。OR REPLACE 选项重新创建该视图(如果它已经存在的话)。FORCE 选项用来创建视图，而无论该视图所引用的表是否存在或者用户是否拥有相应的权限。用户仍然不能执行该视图，但可以创建该视图。NO FORCE 选项只在基表存在且用户拥有相应的权限的情况下，才创建该视图。

如果指定一个别名，则视图将使用该别名作为查询中相应列的名称。如果没有指定别名，则视图将继承查询中的列名。在这种情况下，查询中的每列必须拥有唯一的名称，每个名称都遵循常规的 Oracle 命名约定。它不能为一个表达式。查询中的别名本身还可用于对列进行重命名。

AS query 标识表中的列和此视图中出现的其他视图。它的 WHERE 子句将确定要检索哪些行。

WITH CHECK OPTION 限制视图中所执行的 INSERT 和 UPDATE 操作，以避免它们创建视图本身不能选择的行，该选项基于 CREATE VIEW 语句的 WHERE 子句。WITH CHECK OPTION 可用在基于另一个视图的视图中。但是，如果基础视图也有 WITH CHECK OPTION 选项，则将忽略它。

constraint 是给 CHECK OPTION 指定的一个可选名称。如果没有它，Oracle 就以 SYS_Cn 的形式分配一个名称，其中 n 为一个整数。Oracle 分配的名称通常将在导入过程中被更改，而用户指定的名称将不会被更改。

如果一个视图基于单个表且它的查询不包含 GROUP BY 子句、DISTINCT 子句、分组函数或对伪列 RowNum 的引用，则 UPDATE 和 DELETE 将作用于该视图中的行。可以更新(UPDATE)包含其他伪列或表达式的视图，只要它们不在此 UPDATE 操作中引用即可。

如果视图基于单个表且它的查询不包含 GROUP BY 子句、DISTINCT 子句、分组函数、对任何伪列的引用或任何表达式，则可以在视图中插入(INSERT)行。使用 INSTEAD OF 触发器，还可以通过拥有多个表的视图来插入行。如果该视图用 WITH READ ONLY 创建，则只允许 SELECT 该视图，不允许在该视图上进行数据操纵。

可以创建对象视图，以便在已有关系表上叠加抽象数据类型。要创建一个视图，必须拥有 CREATE VIEW 系统权限。要在另一个用户的模式中创建视图，必须拥有 CREATE ANY VIEW 系统权限。

CUBE

参阅：GROUPING、ROLLUP 和第 12 章。

格式：

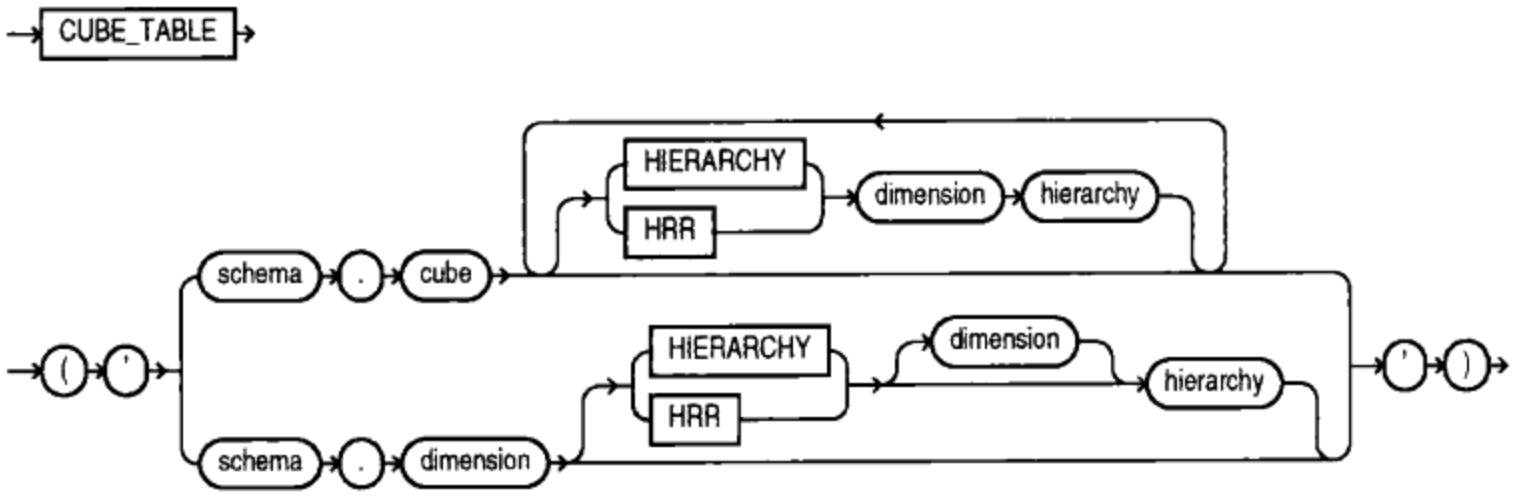
```
GROUP BY CUBE (column1, column2)
```

描述：CUBE 基于每行的表达式的所有可能组合的值对行进行分组，并为每组返回一行汇总信息。可以使用 CUBE 操作产生交叉表报表。请参阅第 12 章。

CUBE_TABLE

参阅：AGGREGATE FUNCTIONS。

格式：



描述：CUBE_TABLE 从立方体或维度中抽取数据，并以二维格式的关系表返回，它可以用于基于 SQL 的应用程序中。

此函数接受一个 VARCHAR2 参数。可选的分层子句允许指定维度的层次结构。立方体可以有多个分层子句，每个维度一个子句。

可以生成以下不同类型的表：

- 立方表　在立方体中针对每个维度包含一个键列，针对每个度量和计算出来的度量包含一列。要创建一个立方表，在指定立方体时可以用立方体分层子句，也可以不用立方分层子句。对于具有多个分层的维度，此子句限制返回值为指定分层中的维度成员和级别。如果不使用分层子句，则包含所有维度成员和所有级别。
- 维度表　包含一个键列，也针对每个级别和每个属性包含一列。所有维度成员和所有级别都包含在表中。要创建一个维度表，指定维度时不用维度分层子句。
- 分层表　包含维度表的所有列以及一个父成员列和每个源级别的列。不是指定分层一部分的任何维度成员和级别都从表中排除。要创建一个分层表，指定维度时使用维度分层子句。

CUBE_TABLE 是一个表函数，它始终用在 SELECT 语句的上下文中，语法如下：

```
select ... from table(CUBE_TABLE('arg'));
```

CUME_DIST

参阅：AGGREGATE FUNCTIONS。

格式：对于聚集的格式为：

```
CUME_DIST ( expr [, expr]... ) WITHIN GROUP
( ORDER BY
  expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]]...)
```

对于分析的格式为：

```
CUME_DIST ( ) OVER ( [query_partition_clause] order_by_clause )
```

描述：CUME_DIST 计算一组值中每一个值的累积分布。CUME_DIST 的返回值的范围

大于 0 且小于等于 1。连接的值总是估算出相同的累积分布值。

CURRENT_DATE

参阅： DATE FUNCTIONS。

格式：

```
CURRENT_DATE
```

描述： CURRENT_DATE 返回会话时区的当前日期，其值为数据类型 DATE 的公历值。

示例：

```
SELECT CURRENT_DATE FROM DUAL;
```

CURRENT_TIMESTAMP

参阅： DATE FUNCTIONS。

格式：

```
CURRENT_TIMESTAMP [ ( precision ) ]
```

描述： CURRENT_TIMESTAMP 返回会话时区的当前日期和时间，它为 TIMESTAMP WITH TIME ZONE 数据类型的值。时区的位移量反映 SQL 会话的当前本地时间。如果省略精度，则默认值为 6。该函数和 LOCALTIMESTAMP 之间的区别是：当 LOCALTIMESTAMP 返回一个 TIMESTAMP 值时，CURRENT_TIMESTAMP 返回一个 TIMESTAMP WITH TIME ZONE 值。precision 指定返回的时间值的小数部分的精度为秒。

示例：

```
select CURRENT_TIMESTAMP from DUAL;
```

CURSOR(PL/SQL)

参阅： CREATE PACKAGE、CREATE PACKGAE BODY 和第 32 章。

格式：

```
CURSOR cursor [(parameter datatype[,parameter datatype]...)
  [IS query]
```

描述： 可以在 PL/SQL 程序包中指定一个游标并声明其游标体。该规范只包含带有相应数据类型的参数的列表，不包含 IS 子句，而游标体则包含二者。参数可出现在查询中可以出现常量的任何地方。可以指定游标作为程序包规范的公有声明的一部分，而游标体作为隐藏的程序包体的一部分。

CURSOR FOR LOOP

在 FOR 循环中，循环执行指定的次数。在游标 FOR 循环中，使用查询结果动态地确定循环执行的次数。在游标 FOR 循环中，打开、获取和关闭游标都是隐式执行的，不需要显式

地对这些动作进行编码。

以下程序清单显示了查询 RADIUS_VALS 表并将记录插入 AREAS 表中的游标 FOR 循环：

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  area NUMBER(14,2);
  cursor rad_cursor is
     select * from RADIUS_VALS;
begin
  for rad_val in rad_cursor
    loop
      area := pi*power(rad_val.radius,2);
      insert into AREAS values (rad_val.radius, area);
    end loop;
end;
```

在游标 FOR 循环中，没有 OPEN 或 FETCH 命令。

```
for rad_val in rad_cursor
```

该命令隐式打开 rad_cursor 游标，并将值引入 rad_val 变量。注意，rad_val 变量没有在使用游标 FOR 循环时显式声明。当游标中再也没有记录时，循环退出并关闭游标。在游标 FOR 循环中，不需要 CLOSE 命令。关于在 PL/SQL 中游标管理的详细信息请参阅第 32 章。

CV

格式：

```
CV([dimension_column])
```

描述：CV 只能用于 SELECT 语句的 MODEL 子句中。它返回一个从规则左侧传送到右侧的维度列当前值，以提供关于维度列的相关索引。

DATA MINING FUNCTIONS

数据挖掘函数作用于使用 DBMS_DATA_MINING 程序包或 Oracle Data Mining Java API 构建的模型。SQL 数据挖掘函数有 CLUSTER_ID、CLUSTER_PROBABILITY、CLUSTER_SET、FEATURE_ID、FEATURE_SET、FEATURE_VALUE、PREDICTION、PREDICTION_BOUNDS、PREDICTION_COST、PREDICTION_DETAILS、PREDICTION_PROBABILITY 和 PREDICTION_SET。

DATATYPES

参阅：CHARACTER FUNCTIONS、CONVERSION FUNCTIONS、DATE FUNCTIONS、LIST FUNCTIONS、NUMBER FUNCTIONS 和 OTHER FUNCTIONS。

描述：在创建一个表并定义其中的列时，这些列必须具有各自特定的数据类型。Oracle 的基本数据类型有 VARCHAR2、CHAR、DATE、LONG、LONG RAW、NUMBER、RAW 以及 ROWID，但是为了与其他 SQL 数据库兼容，它的 CREATE TABLE 语句还接受这些数据类型的多种版本。表 A-10 汇总了 Oracle 内置的数据类型。

表 A-10　Oracle 内置的数据类型

数 据 类 型	定　　义
VARCHAR2(size)	变长字符串，最多具有 size 个字节(最多为 4 000)
NVARCHAR2(size)	变长字符串，最多具有 size 个字节(最多为 4 000)或字符，根据所选择的语言字符集而定
NUMBER(precision，scale)	用于指定了 precision 且在小数点后有 scale 位数字的 NUMBER 列。precision 的取值范围从 1~38；scale 的取值范围从 - 84～127
LONG	长度最大为 2GB 的可变长字符数据
DATE	有效的日期范围从公元前 4712 年的 1 月 1 日到公元 9999 年的 12 月 31 日
BINARY_FLOAT	32 位浮点数
BINARY_DOUBLE	64 位浮点数
TIMESTAMP(precision)	日期加时间，其中 precision 是 Seconds 字段的小数部分中数字的位数(在 0~9 之间取值，默认为 6)
TIMESTAMP(precision) WITH TIME ZONE	TIMESTAMP 加时区转换值。其中 precision 是 Seconds 字段的小数部分中数字的位数(在 0~9 之间取值，默认为 6)
TIMESTAMP(precision) WITH LOCAL TIME ZONE	TIMESTAMP，标准化为本地时区
INTERVAL YEAR(precision) TO MONTH	以年和月表示的时间段，其中 precision 为 YEAR 日期时间字的段中数字位数(在 0~9 之间取值，默认为 2)
INTERVAL DAY(day_precision)TO SECOND(second_precision)	以日、小时、分钟、秒表示的时间段，其中 day_precision 为 DAY 日期时间字段中数字的位数(在 0~9 之间取值，默认为 2)，second_precision 为 Second 字段的小数部分中数字的位数(在 0~9 之间取值，默认为 6)
RAW(size)	原始二进制数据，size 个字节长。最大长度为 2 000 个字节
LONG RAW	原始二进制数据；其他的与 LONG 相同
ROWID	64 位的基字符串，表示表中行的唯一地址
UROWID(size)	用于索引编排表中一行的逻辑地址的 64 位基字符串；最大的 size 为 4 000 字节
CHAR(size)	定长字符数据，size 为字符长度。最大长度为 2 000。默认为 1 字节。右边用空格填充以达到 size 指定的长度
NCHAR(size)	CHAR 的多字节字符集版本
CLOB	字符大对象，最大长度为 4GB
NCLOB	与 CLOB 相同，但是包含 Unicode 字符
BLOB	二进制大对象，最大长度为 4GB
BFILE	指向二进制操作系统文件的指针

DATE

DATE 是存储日期和时间数据的一个标准的 Oracle 数据类型。标准的日期格式如 22-APR-04。DATE 列可包含公元前 4712 年 1 月 1 日到公元 9999 年 12 月 31 日之间的一个日期和时间。

DATE FORMATS

参阅： DATE FUNCTIONS 和第 10 章。

描述： 日期格式模型用于转换函数。表 A-11 列出了日期模型组件：

表 A-11 日期模型组件

日期模型组件	说 明
/,-: .;	在 TO_CHAR 的显示中结合使用，在 TO_DATE 的格式中忽略的标点符号
A.D.或 AD	AD 指示器，有没有句点都可以
A.M.或 AM	显示 A.M.或 P.M.，根据一天中的具体时间而定。有没有句点都可以
B.C.或 BC	与 A.D.或 AD 相同
CC	世纪：(如 2004 年为 21 世纪)
SCC	世纪，公元前的日期加前缀 '-'
D	一周中的天数：1～7
DAY	DAY 的完整拼写，填充到 9 个字符
DD	一个月中的天数：1～31
DDD	一年中的天数(从 1 月 1 日开始)：1～366
DL	用本地长格式表示的日期；在 U.S.格式中，等价于 fmDay,Month dd,yyyy
DS	用本地短格式表示的日期；在 U.S.格式中，等价于 MM/DD/RRRR
DY	星期几的 3 个字母的缩写：FRI
E	缩写的纪元名(用于日本皇室、ROC 官方以及泰国佛教的历法)
EE	E 格式的完整纪元名版本
FF[1..9]	秒的小数部分；FF 后的数字指定了在秒的小数部分中显示的数字位数
FM	去掉首尾的空格；没有 FM 的话，所有的月和日都以相同的宽度显示
FX	格式要求：指定准确的格式，匹配字符参数和日期格式模型
HH	一天中的小时：始终是 1～12
HH12	与 HH 相同
HH24	一天以 24 小时计算的小时：0～23
I	ISO 标准的一位数字的年份
IW	ISO 标准的一年中的周：1～53
IY	ISO 标准的两位数字的年份
IYY	ISO 标准的 3 位数字的年份
IYYY	ISO 标准的 4 位数字的年份
J	儒略历；自公元前 4712 年的 12 月 31 日以来的天数
MI	一小时中的分：0～59
MM	一年中的月：01～12
MON	月的 3 个字母的缩写：例如，AUG
MONTH	月的完整拼写：例如，AUGUST

(续表)

日期模型组件	说　明
P.M.	与 A.M.作用相同
Q	一年中的季度：1～4
RM	罗马数字的月份
RR	与当前日期对应的年份的最后两位数字
RRRR	完整的年份，支持两位或 4 位数字的输入
SS	一分钟内的秒：1～60
SSSSS	从午夜开始计算的秒：始终是 0～86 399
TS	短时格式，与 DL 或 DS 一起使用
TZD	省去时间信息的白昼
TZH	时区小时
TZM	时区分钟
TZR	时区区域
W	月中的周数，其中第一周从当月的第一天开始
WW	年中的周数，其中第一周从该年的第一天开始
X	本地根字符
YEAR 或 SYEAR	拼写出的年份；S 标识以负号开头的公元前的日期
YYYY 或 SYYYY	完整的 4 位数字表示的年份；S 标识以负号开头的公元前的日期
Y,YYY	有逗号的年份
Y	年份的最后一位数字
YY	年份的最后两位数字
YYY	年份的最后 3 位数字

下面的日期格式只适用于 TO_CHAR，对 TO_DATE 不起作用：

“string”	在 TO_CHAR 的显示中结合使用的 string。
TH	数字的后缀：ddTH 或 DDTH 得到 24th 或 24TH。大写是由数字 dd 或 DD 而不是由 TH 决定的。可以使用日期中的任何数字：YYYY、DD、MM、HH、MI、SS 等。
SP	数字的后缀，将数字按相应的要求拼写出来：DDSP、DdSP、ddSP 得出 THREE、Three 或 three。大写是由数字 dd 或 DD 而不是由 SP 决定的。可以使用日期中的任何数字：YYYY、DD、MM、HH、MI、SS 等。
SPTH	TH 和 SP 的后缀组合，强制按规定拼写出来数字并给出序数后缀：Ddspth 得到 Third。大写是由数字 dd 或 DD 而不是由 SP 决定的。可以使用日期中的任何数字：YYYY、DD、MM、HH、MI、SS 等。
THSP	与 SPTH 相同。

DATE FUNCTIONS

参阅：DATE FORMATS 和第 10 章。

描述：下面是 Oracle 版本的 SQL 中的所有当前日期函数按字母顺序列出的一个清单。其中每个函数按照各自的名字在本参考的其他地方列出，同时给出正确的格式和用法。

- **ADD_MONTHS(date，count)** 将 count 个月份加到 date 上。
- **CURRENT_DATE** 返回会话时区的当前日期。
- **CURRENT_TIMESTAMP** 返回带有效时区信息的当前时间戳。
- **DBTIMEZONE** 以 UTC 格式返回当前数据库时区。
- **EXTRACT(datetime)** 从一个日期值中提取日期的某个部分，如从日期列的值中提取月份值。
- **FROM_TZ** 将一个时间戳值转换为一个带时区值的时间戳。
- **LAST_DAY(date)** 给出 date 所属月的最后一天的日期。
- **LOCALTIMESTAMP** 返回有效时区中的本地时间戳，不显示时区信息。
- **MONTHS_BETWEEN(date2，date1)** 以月份为单位进行 date2－date1 减法运算(月份数可以为小数)。
- **NEW_TIME(date，'this'，'other')** 给出 this 时区中的 date(和时间)。this 用当前时区的 3 个字符的缩写替换。other 用想要知道其时间和日期的其他时区的 3 个字符的缩写替换。时区如表 A-12 所示。

表 A-12 不同的时区

时区缩写	说 明
AST/ADT	大西洋标准/白昼时间
BST/BDT	白令海标准/白昼时间
CST/CDT	中央标准/白昼时间
EST/EDT	东部标准/白昼时间
GMT	格林尼治标准时间
HST/HDT	阿拉斯加-夏威夷标准/白昼时间
MST/MDT	山地标准/白昼时间
NST	纽芬兰岛标准时间
PST/PDT	太平洋标准/白昼时间
YST/YDT	育空地区标准/白昼时间

- **NEXT_DAY(date,'day')** 给出 date 之后下一天的日期，其中'day'为'Monday'、'Tuesday'等。
- **NUMTODSINTERVAL(n,'value')** 把数字或表达式 n 转换为一个 INTERVAL DAY TO SECOND 字面量，其中 value 为 n 的单位，如'DAY'、'HOUR'、'MINUTE'、'SECOND'。
- **NUMTOYMINTERVAL(n,'value')** 把数字或表达式转换 n 为一个 INTERVAL YEAR TO MONTH 字面量，其中 value 为 n 的单位，如'YEAR'、'MONTH'。
- **ROUND(date,'format')** 如果不指定 format，那么若 date 的时间在中午以前，则将 date 舍入为 12A.M.(午夜，即该天的开始)；否则舍入到下一天。关于舍入的 format 的用法，请参阅 ROUND。
- **SESSIONTIMEZONE** 返回当前会话时区的值。
- **SYS_EXTRACT_UTC(datetime_with_timezone)** 从一个带时区位移量的日期时间字段中提取 Coordinated Universal Time(UTC，世界标准时间)。

- **SYSDATE**　返回当前日期和时间。
- **SYSTIMESTAMP**　返回系统日期，包括数据库的小数秒和时区。
- **TO_CHAR(date,'format')**　根据 format 重新格式化 date。
- **TO_DATE(string,'format')**　将 string 以给定的'format'转换为 Oracle 日期。除了 string，也可以接受 number，但是有一定的限制。'format'也是有限制的。
- **TO_DSINTERVAL('value')**　把 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的 value 转换为 INTERVAL DAY TO SECOND 类型。
- **TO_TIMESTAMP('value')**　把 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的 value 转换为 TIMESTAMP 数据类型的值。
- **TO_TIMESTAMP_TZ('value')**　把 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的 value 转换为 TIMESTAMP WITH TIMEZONE 数据类型的值。
- **TO_YMINTERVAL('value')**　把 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的 value 转换为 INTERVAL YEAR TO MONTH 数据类型。
- **TRUNC(date,'format')**　如果不指定 format，则设置 date 为 12A.M.(午夜，即该天的开始)。关于截断的 format 的用法，请参阅 TRUNC。
- **TZ_OFFSET('value')**　返回与基于语句执行的日期输入的 value 相对应的时区偏移量。

DBTIMEZONE

参阅： DATE FUNCTIONS。

格式：

```
DBTIMEZONE
```

描述： DBTIMEZONE 返回数据库时区的值。

示例：

```
select DBTIMEZONE from DUAL;
```

DECLARE CURSOR(格式 1——嵌入式 SQL)

参阅： CLOSE、DECLARE DATABASE、DECLARE STATEMENT、FETCH、OPEN、PREPARE、SELECT、SELECT(嵌入 SQL)和 *Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [AT {database | :host_variable}]
    DECLARE cursor CURSOR
       FOR {SELECT command | statement}
```

描述： DECLARE CURSOR 语句必须出现在引用游标的任何 SQL 之前，而且必须在与引用它的过程相同的源中编译。它的名称必须对源是唯一的。

database 是已经在 DECLARE DATABASE 中声明过且在一个 CONNECT 中使用的一个数据库名；host_variable 是将此名称作为值的一个变量。cursor 是赋予该游标的名称。SELECT

command 是一个没有 INTO 子句的查询。可供选择的另一个方法是，可以使用在 SQL DECLARE STATEMENT 语句中声明过的一条 SQL 语句或一个 PL/SQL 块。

当在 SELECT 语句中包含 FOR UPDATE OF 子句时，UPDATE 可以利用 WHERE CURRENT OF cursor 子句引用游标，尽管游标仍然必须打开，并且从 FETCH 中显示一行。

DECLARE CURSOR(格式 2——PL/SQL)

参阅： CLOSE、FETCH、OPEN、SELECT…INTO 和第 32 章。

格式：

```
DECLARE
CURSOR cursor (parameter datatype [,parameter datatype]...)
    IS select_statement
    [FOR UPDATE OF column [,column]...];
```

描述： 游标是 Oracle 用来控制当前处理的行的一个工作区。DECLARE CURSOR 命名一个游标并声明它是(IS)某个 select_statement 语句(的结果)。select_statement 语句是必需的，并且可能不包含 INTO 子句。必须声明游标。然后可以打开(OPEN)它，之后才可以把行 FETCH 到游标之中。最后可以关闭(CLOSE)游标。

除非首先在 select 之前的列表中作为参数标识变量，否则不能在 select_statement 的 WHERE 子句中直接使用变量。请注意，DECLARE CURSOR 并不执行 SELECT 语句，而且在一个 OPEN 语句中给各参数赋值前它们没有值。这些参数具有标准的对象名和数据类型，数据类型包括 VARCHAR2、CHAR、NUMBER、DATE 和 BOOLEAN，但全都没有大小和小数位限制。

如果希望用带 CURRENT OF 子句的 UPDATE 或 DELETE 命令更改游标中的当前行，则必须使用 FOR UPDATE OF。

DECLARE DATABASE(嵌入式 SQL)

参阅： CONNECT(嵌入式 SQL)、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL DECLARE database DATABASE
```

描述： DECLARE DATABASE 声明在 SQL 语句的 AT 子句中使用的远程数据库名，这些 SQL 语句包括 COMMIT、DECLARE CURSOR、DELETE、INSERT、ROLLBACK、SELECT 和 UPDATE。

DECLARE STATEMENT(嵌入式 SQL)

参阅： CLOSE、FETCH、OPEN、PREPARE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [AT { database | : host_variable}]
    DECLARE STATEMENT { statement | block_name} STATEMENT
```

描述：statement 为 DECLARE CURSOR 语句中的语句名，而且必须与这里的语句名相同。database 是 DECLARE DATABASE 声明过的一个数据库名，host_variable 可能包含与其值一样的名称。这条命令仅在 DECLARE CURSOR 出现在 PREPARE 命令之前时才需要。使用这条命令时，它应该位于 DECLARE、DESCRIBE、OPEN 或 PREPARE 之前，而且必须在与引用它的过程相同的源中编译。

DECLARE TABLE

参阅：CREATE TABLE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL DECLARE table TABLE
 (column datatype [NULL|NOT NULL],
        ...);
```

描述：table 为要声明的表名。column 为列名，datatype 为其数据类型。此语法结构与 CREATE TABLE 的语法结构非常类似，包括 NULL 和 NOT NULL 的用法也很类似。

可以使用 DECLARE TABLE 来告诉预编译程序，在用 SQLCHECK=FULL 运行时，忽略实际的 Oracle 数据库表定义。预编译程序将认为这里的表描述与程序有关，从而忽略数据库中的表定义。在改变表定义或迄今为止尚未创建表时，可以使用这条命令。如果 SQLCHECK 不等于 FULL(这表示不再为数据库检查表和列)，预编译程序将忽略这条命令，这条命令将变为文档。

示例：

```
select DISTINCT City,
       DECODE(City, 'SAN FRANCISCO', 'CITY BY THE BAY', City)
  from COMFORT;

CITY           DECODE(CITY,'SA
-------------- ---------------
KEENE          KEENE
SAN FRANCISCO  CITY BY THE BAY
```

DECOMPOSE

参阅： *CONVERSION FUNCTIONS*。

格式：

```
DECOMPOSE ('string')
```

描述： DECOMPOSE 在与输入相同的字符集中进行规范的分解后，将任意数据类型的串转换为 Unicode 串。

DEFINE(SQL*Plus)

参阅： SET、EDIT。

格式：

```
DEF[INE] [variable] | [ variable = text]
```

描述： variable 是一个用户或者是希望赋值或列出其值的预定义变量。text 是变量的新值；它必须是 CHAR 值，并且，如果它含有标点或空格，就应该用单引号括起来。没有变量的 DEFINE 将列出所有的变量。

表 A-13 列出的是 Oracle 为每个会话预定义的变量：

表 A-13 Oracle 为每个会话预定义的变量

变 量	描 述
_CONNECT_IDENTIFIER	用于建立当前连接的连接标识符
_DATE	当前日期，或者一个用户定义的固定的串
_EDITOR	指定通过 EDIT 命令使用的编辑器。
_O_RELEASE	数据库的完整的版本号
_O_VERSION	数据库的当前版本
_PRIVILEGE	当前连接的权限级别(例如，AS SYSDBA)
_SQLPLUS_RELEASE	所用 SQL*Plus 版本的完整的版本号
_USER	用于当前连接的用户名

DEL(SQL*Plus)

参阅： APPEND、CHANGE、EDIT、INPUT、LIST 和第 6 章。

格式：

```
DEL [ n | n m | n* | n LAST | * | * n | * LAST | LAST]
```

描述：DEL 删除当前缓冲区中的当前行。可利用单条 DEL 命令删除多行。DEL 在 iSQL*Plus 中不可用。

示例：

```
del
```

此命令删除 SQL*Plus 缓冲区中的当前行。

要用一条命令删除某个范围中的行，在 DEL 命令中指定要删除的行的范围的开始和结束的行号即可。下面的命令将删除当前缓冲区中的第 2~8 行。

```
del 2 8
```

要删除从第 4 行到结束的所有行，使用字符“*”。

```
del 4*
```

DELETE(格式 1——PL/SQL)

参阅：DECLARE CURSOR、UPDATE 和第 32 章。

描述：在 PL/SQL 中，DELETE 遵循标准的 SQL 命令规则，同时增加了下面的特性：

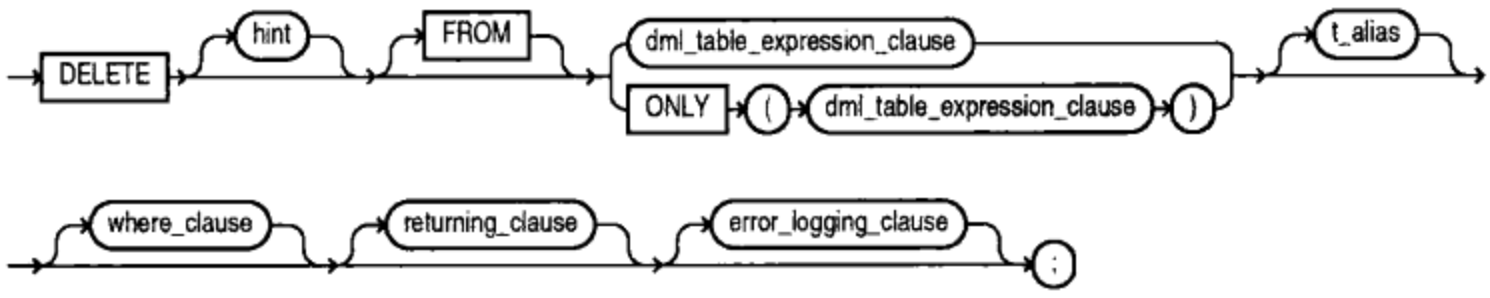
- PL/SQL 函数和/或变量可以像一个字面量那样用在 WHERE 子句中。
- DELETE…WHERE CURRENT OF cursor 可与 SELECT FOR UPDATE 联合使用以删除最后 FETCH 的行。FETCH 在 FOR 循环中可以是显式的也可以是隐式的。
- 与 UPDATE 和 INSERT 一样，DELETE 命令仅在 SQL 游标中执行。可检查 SQL 游标的属性看 DELETE 是否成功。SQL%ROWCOUNT 包含删除的行数。如果为 0，就表示没有删除(DELETE)行(而且，如果没有 DELETE 行，则 SQL%FOUND 将为 FALSE)。

DELETE(格式 2——SQL 命令)

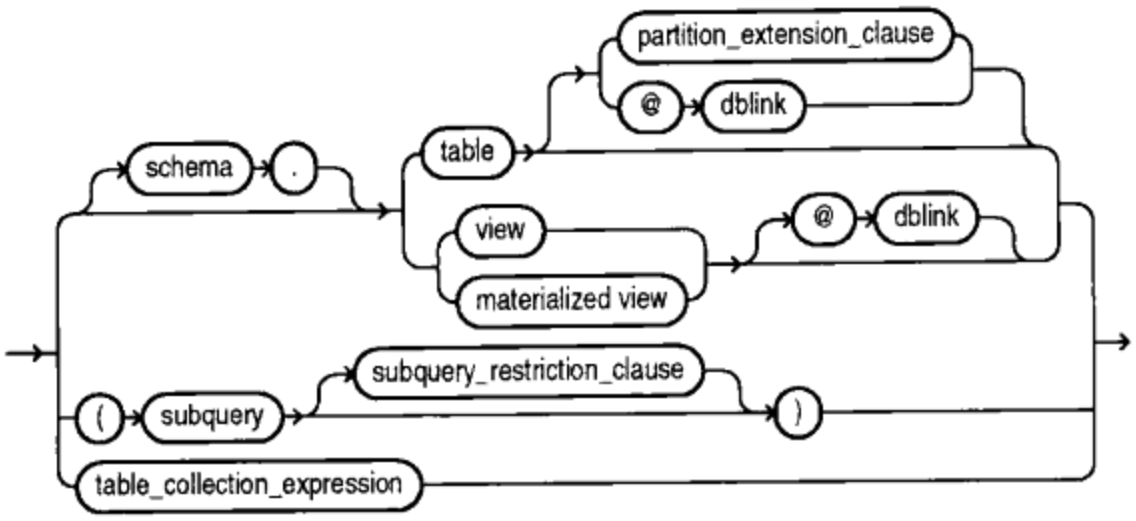
参阅：DROP TABLE、FROM、INSERT、SELECT、UPDATE、WHERE 和第 15 章。

格式：

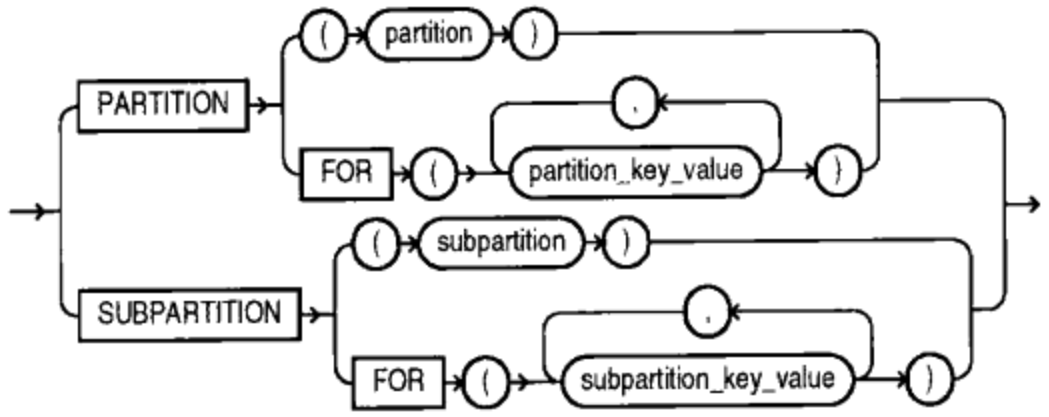
delete::=



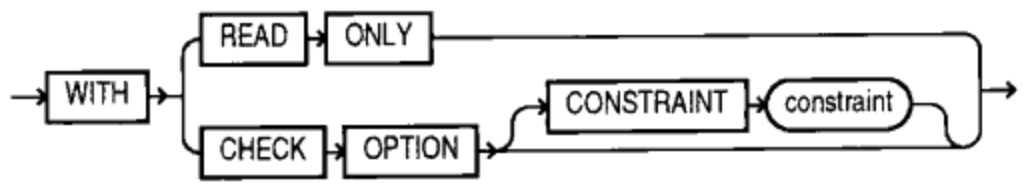
DML_table_expression_clause::=



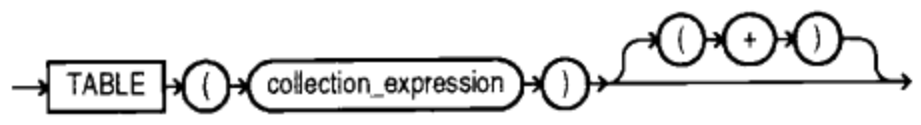
partition_extension_clause::=



subquery_restriction_clause::=



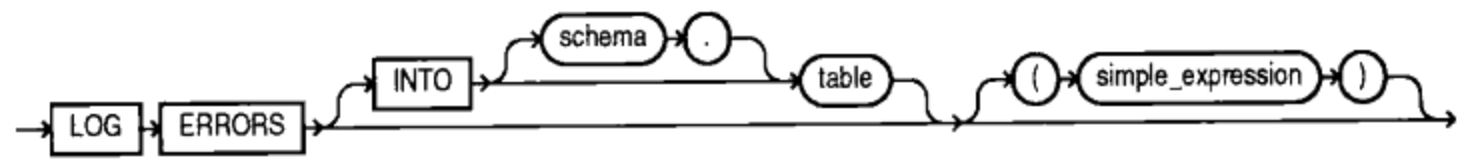
table_collection_expression::=

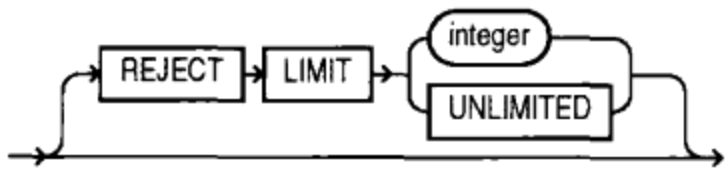


where_clause::=

returning_clause::=

error_logging_clause::=





描述：DELETE 删除 table 中所有满足 condition 的行。condition 可以包含相关的查询并使用相关的别名。如果表是远程的，则必须定义一个数据库链接。@link 指定链接。如果输入 link，但不输入 user，那么查询将在远程数据库上寻找此用户所拥有的表。

在使用 DELETE 命令时，可以指定分区或子分区。为了从一个表中删除行，必须具有该表的 DELETE 权限。如果 SQL92_SECURITY 初始化参数设置为 TRUE，且 DELETE 引用表本身的列，则用户还必须具有 SELECT 权限。

示例：该例从 COMFORT 表中删除城市 Keene 的所有行：

```
delete from COMFORT
 where City = 'KEENE';
```

DELETE(格式 3——嵌入式 SQL)

格式：

```
EXEC SQL [AT {db_name | :host_variable} ] [FOR :host_integer | integer]
  DELETE [FROM]
  [ (subquery)
  | [user.] {table | view } [@dblink |PARTITION (partition_name)] ]
  [alias] [WHERE {condition | CURRENT OF cursor} ]
  [{ RETURN | RETURNING } expr [, expr]... INTO
    :host_variable [[INDICATOR] :ind_variable]
    [, :host_variable [[INDICATOR] :ind_variable]]...]
```

描述：这种格式的 DELETE 命令允许在一个预编译程序内删除行(从表、视图或仅仅是索引的表)。

DELETEXML

格式：

```
DELETEXML
  ( XMLType_instance, XPath_string
    [, namespace_string ] )
```

描述：DELETEXML 删除目标 XML 中与 XPath 表达式匹配的一个或多个节点。

输入变量是：

- XMLType_instance，它是 XMLType 的一个实例。
- XPath_string，它是一个 XPath 表达式，指定要删除的一个或多个节点。XPath_string 可以指定为绝对的(开头使用斜杠)，也可以指定为相对的(开头省略斜杠)。如果开头省略斜杠，则相对路径的上下文默认为根节点。XPath_string 指定的节点的任何子节点也会被删除。

- 可选的namespace_string，它为XPath_string提供命名空间信息。此参数必须是VARCHAR2类型。

DENSE_RANK

参阅：AGGREGATE FUNCTIONS、第 12 章。

格式：

```
DENSE_RANK ( expr [, expr]... ) WITHIN GROUP
( ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...)
```

用于分析的格式如下：

```
DENSE_RANK ( ) OVER ( [ query_partition_clause] order_by_clause )
```

描述：DENSE_RANK 计算一个有序行组中行的排序。相关的示例请参阅第 12 章。

DEPTH

格式：

```
DEPTH(integer)
```

描述：DEPTH 是一个只与 UNDER_PATH 和 EQUALS_PATH 条件一起使用的辅助函数。它返回由具有相同相关变量的 UNDER_PATH 条件指定的路径中的级数。这里，integer 可以是任何 NUMBER 整数。

DEREF

DEREF 运算符返回引用对象的值。请参阅第 41 章。

DESCRIBE(格式 1——SQL*Plus)

参阅：CREATE TABLE。

格式：

```
DESC[RIBE] {[user.] object[@connect_identifier]}
```

描述：DESCRIBE 显示指定对象的定义。如果省略 user，则 SQL*Plus 将显示当前用户拥有的表。定义包括表和它的列，每个列的列名、NULL 或 NOT NULL 状态、数据类型以及宽度或精度。DESCRIBE 也可以用在视图、同义词、函数和过程的说明上。

示例：

```
describe COMFORT

Name                              Null?     Type
--------------------------------- --------  ------------
CITY                              NOT NULL  VARCHAR2(13)
```

```
SAMPLEDATE                              NOT NULL  DATE
NOON                                              NUMBER(3,1)
MIDNIGHT                                          NUMBER(3,1)
PRECIPITATION                                     NUMBER
```

DESCRIBE(格式 2——嵌入式 SQL)

参阅： PREPARE。

格式：

```
EXEC SQL DESCRIBE  [ BIND VARIABLES FOR
                        | SELECT LIST FOR]
                 statement_name INTO descriptor
```

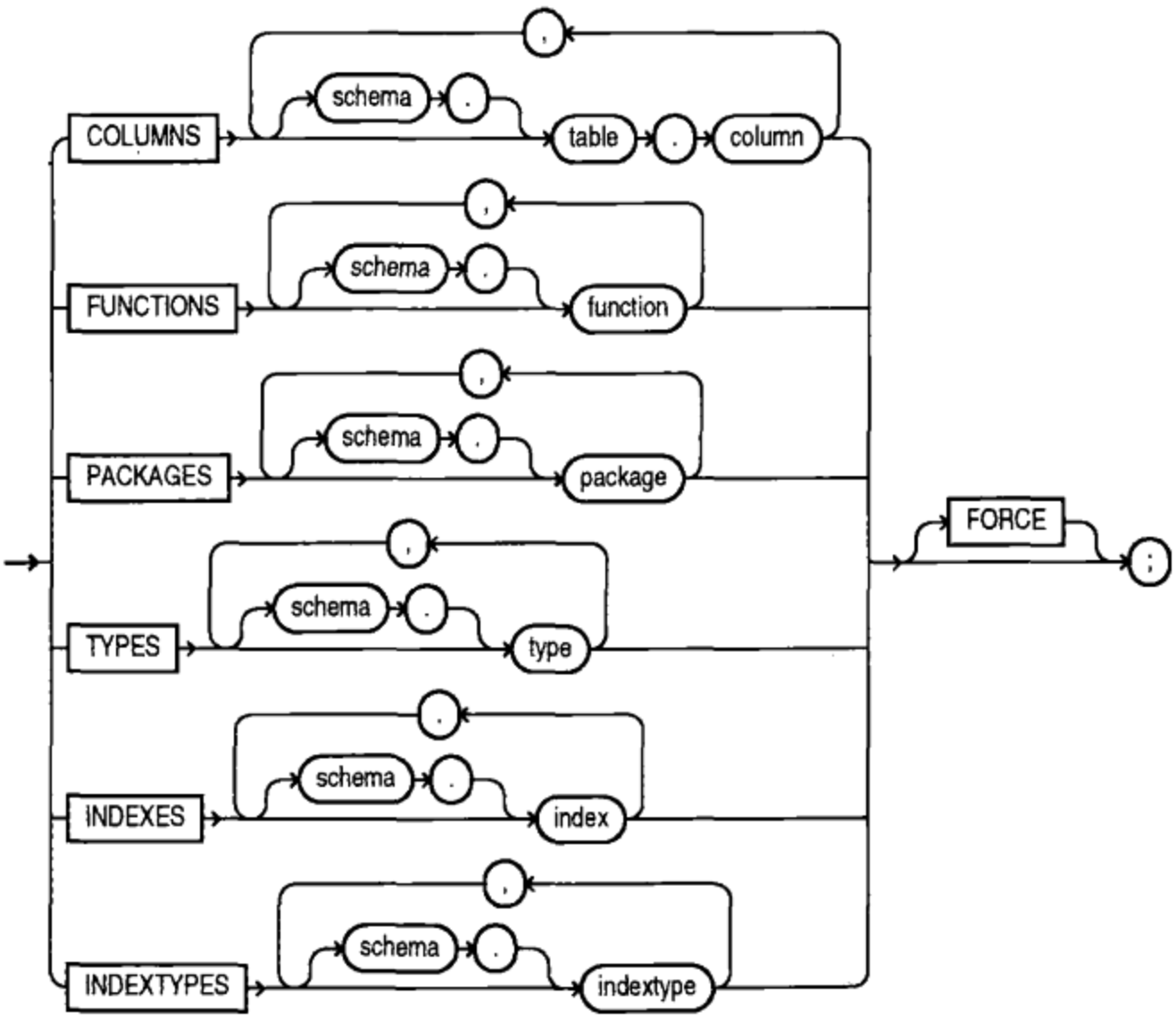
描述： DESCRIBE 命令初始化一个描述符以保存动态 SQL 语句或 PL/SQL 块的宿主变量的描述。DESCRIBE 命令遵循 PREPARE 命令的规定。

DISASSOCIATE STATISTICS

参阅： ASSOCIATE STATISTICS。

格式：

disassociate_statistics::=



描述：ASSOCIATE STATISTICS 把一组统计函数与一个或多个列、独立函数、程序包、类型或索引关联在一起。DISASSOCIATE STATISTICS 取消这种关联。用户必须在基对象上具有 ALTER 权限。

DISCONNECT(SQL*Plus)

参阅：CONNECT、EXIT 和 QUIT。

格式：

```
DISC[ONNECT]
```

描述：DISCONNECT 把挂起的更改提交给数据库并使用户从 Oracle 中退出。这时用户的 SQL*Plus 会话仍然有效，并且 SQL*Plus 的功能继续起作用，只是断开了与数据库的连接。用户可以编辑缓冲区，使用自己的编辑器，假脱机或停止假脱机，或者再次连接到 Oracle。但是在建立连接之前，不能执行 SQL。

EXIT 或 QUIT 将使用户返回到主机的操作系统。如果在 EXIT 或 QUIT 时仍然与数据库连接着，则会自动断开连接并退出 Oracle。

DISTINCT

DISTINCT 表示唯一。它是 SELECT 语句和分组函数的组成部分。请参阅 AGGREGATE FUNCTIONS 和 SELECT。

DROP CLUSTER

参阅：CREATE CLUSTER、DROP TABLE、第 17 章。

格式：

```
DROP CLUSTER [schema .] cluster
  [INCLUDING TABLES [CASCADE CONSTRAINTS]];
```

描述：DROP CLUSTER 从数据库中删除一个群集。如果该群集不在用户的模式中，则用户必须拥有 DROP ANY CLUSTER 权限。不能删除一个含有表的群集，必须首先删除表。INCLUDING TABLES 子句可以自动删除所有群集表。CASCADE CONSTRAINTS 选项从群集外的表中删除所有参照完整性约束，这些表引用群集表中的键。

个别的表不能从群集中删除。为了达到相同的效果，用一个新的名称复制表(使用带 AS SELECT 的 CREATE TABLE)，然后删除旧表(将它们从群集中删除)，再将复制的表重命名(RENAME)为删除了的表名，发出适当的 GRANT 命令，并创建所需的索引。

DROP CONTEXT

参阅：CREATE CONTEXT。

格式：

```
DROP CONTEXT namespace;
```

描述：DROP CONTEXT 从数据库中删除上下文名称空间。用户必须具有 DROP ANY CONTEXT 系统权限。

DROP DATABASE

参阅：CREATE DATABASE、第 51 章。

格式：

```
DROP DATABASE;
```

描述：DROP DATABASE 删除当前数据库(数据库必须已经安装并关闭)，以及在控制文件中列出的所有控制文件和数据文件。任何相关的 spfile 文件也要删除。但是已经存档的日志和备份不可以删除。如果数据库在原始磁盘上，则 DROP DATABASE 不删除原始磁盘的专有文件。

DROP DATABASE LINK

参阅：CREATE DATABASE LINK、第 25 章。

格式：

```
DROP [PUBLIC] DATABASE LINK link;
```

描述：DROP DATABASE LINK 删除用户所拥有的一个数据库链接。对于公有链接，必须使用可选的 PUBLIC 关键字，而且用户必须是使用它的 DBA。在删除私有链接时不能使用 PUBLIC。link 是要删除的链接名。为了删除一个公有数据库链接，用户必须有 DROP PUBLIC DATABASE LINK 系统权限。用户可以在自己的账户内删除私有数据库链接。

示例：下例将删除一个名为 ADAH_AT_HOME 的数据库链接：

```
drop database link ADAH_AT_HOME;
```

DROP DIMENSION

参阅：CREATE DIMENTION、第 26 章。

格式：

```
DROP DIMENSION [schema.] dimension;
```

描述：DROP DIMENSION 删除用户所拥有的一个维度。用户必须具有 DROP ANY DIMENSION 系统权限。

示例：下例将删除 GEOGRAPHY 维度：

```
drop dimension GEOGRAPHY;
```

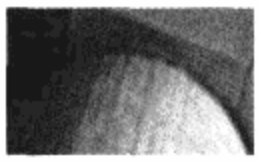
DROP DIRECTORY

参阅：CREATE DIRECTORY、第 24 章、第 28 章、第 40 章。

格式：

```
DROP DIRECTORY directory_name;
```

描述：DROP DIRECTORY 删除一个已有的目录。要删除一个目录，用户必须有 DROP ANY DIRECTORY 权限。



注意：
不要删除其文件正在使用的目录。

示例：下例将删除一个名为 REVIEWS 的目录。

```
drop directory REVIEWS;
```

DROP DISKGROUP

参阅：ALTER DISKGROUP、CREATE DISKGROUP、第 51 章。

格式：

```
DROP DISKGROUP diskgroup_name [ {INCLUDING | EXCLUDING} CONTENTS ];
```

描述：对于使用 Automated Storage Management(ASM，自动存储管理)的数据库，DROP DISKGROUP 删除一个 ASM 磁盘组及磁盘组内的所有文件。ASM 首先确认磁盘组内的所有文件都已关闭；如果有一个文件打开，则磁盘组不能删除。

在 Oracle Database 11g 中，可以使用 FORCE 子句删除不能再由 ASM 实例安装的磁盘组。

DROP FLASHBACK ARCHIVE

参阅：ALTER FLASHBACK ARCHIVE、CREATE FLASHBACK ARCHIVE。

格式：

```
DROP FLASHBACK ARCHIVE flashback_archive;
```

描述：使用 DROP FLASHBACK ARCHIVE 从系统删除闪回数据归档。此语句删除闪回数据归档和其中的所有历史数据，但不删除闪回数据归档使用的表空间。

必须具有 FLASHBACK ARCHIVE ADMINISTER 系统权限才能删除闪回数据归档。

DROP FUNCTION

参阅：ALTER FUNCTION、CREATE FUNCTION、第 35 章。

格式：

```
DROP FUNCTION [schema.] function_name;
```

描述：DROP FUNCTION 删除指定的函数。Oracle 会使依赖或调用此函数的任意对象无效。为了删除一个不属于自己的函数，用户必须具有 DROP ANY PROCEDURE 系统权限。

DROP INDEX

参阅： ALTER INDEX、CREATE INDEX、CREATE TABLE、第 17 章和第 51 章。

格式：

```
DROP INDEX [schema.] index [FORCE];
```

描述： DROP INDEX 删除指定的索引。用户必须拥有此索引或者具有 DROP ANY INDEX 系统权限。FORCE 只应用于域索引。即使索引类型的例程调用返回一个错误或索引标记为 IN PROGRESS，也可以使用这个子句删除域索引。

DROP INDEXTYPE

参阅： CREATE INDEXTYPE。

格式：

```
DROP INDEXTYPE [schema.] indextype [FORCE];
```

描述： DROP INDEXTYPE 删除指定的索引类型及与统计类型关联的任意索引类型。必须具有 DROP ANY INDEXTYPE 系统权限。

DROP JAVA

参阅： CREATE JAVA、第 42 章。

格式：

```
DROP JAVA { SOURCE | CLASS | RESOURCE } [schema.] object_name;
```

描述： DROP JAVA 删除指定的 Java 类、源或资源。用户必须具有 DROP PROCEDURE 系统权限。如果相应的对象位于其他用户的模式内，则还必须具有 DROP ANY PROCEDURE 系统权限。

DROP LIBRARY

参阅： CREATE LIBRARY。

格式：

```
DROP LIBRARY library_name;
```

描述： DROP LIBRARY 删除指定的库。用户必须具有 DROP ANY LIBRARY 系统权限。

DROP MATERIALIZED VIEW

参阅： ALTER MATERIALIZED VIEW、CREATE MATERIALIZED VIEW、第 26 章。

格式：

```
DROP MATERIALIZED VIEW [schema.] materialized_view [PRESERVE TABLE];
```

描述：DROP MATERIALIZED VIEW 删除指定的物化视图。用户必须拥有该物化视图或者具有 DROP ANY SNAPSHOT 或者 DROP ANY MATERIALIZED VIEW 系统权限。

SNAPSHOT 关键字可以代替 MATERIALIZED VIEW 以支持向后兼容。

PRESERVE TABLE 允许用户在删除物化视图对象后，保留物化视图容器表及其内容。所得到的表与删除的物化视图相同。虽然与物化视图相关联的元数据被删除了，但是索引、嵌套表列、这些列的存储表及其元数据都保存了下来。

如果要删除一个在预制表上创建的物化视图，那么数据库删除该物化视图并且预制表恢复为表的身份。

DROP MATERIALIZED VIEW LOG

参阅：ALTER MATERIALIZED VIEW LOG、CREATE MATERIALIZED VIEW LOG、第 26 章。

格式：

```
DROP MATERIALIZED VIEW LOG ON [schema.] table;
```

描述：DROP MATERIALIZED VIEW LOG 删除指定的日志表。为了删除一个物化视图日志，用户必须具有删除表的权限。在删除日志后，主表上的任何物化视图都将被完全(COMPLETE)刷新，而不是快速(FAST)刷新。

关键字 SNAPSHOT 可以代替 MATERIALIZED VIEW 以支持向后兼容。

DROP OPERATOR

参阅：ALTER OPERATOR。

格式：

```
DROP OPERATOR [schema.] operator [FORCE];
```

描述：DROP OPERATOR 删除一个用户定义的运算符。用户必须拥有该运算符或具有 DROP ANY OPERATOR 系统权限。指定 FORCE 删除运算符，即使它当前正被一个或多个模式对象(索引类型、程序包、函数、过程等)引用也可以进行此操作，并且标记这些对象为 INVALID。

DROP OUTLINE

参阅：CREATE OUTLINE。

格式：

```
DROP OUTLINE outline;
```

描述：DROP OUTLINE 从数据库中删除一个存储概要。用户必须具有 DROP ANY OUTLINE 系统权限。使用此概要的 SQL 语句的后续执行将在运行时判定自己的执行路径。

DROP PACKAGE

参阅： ALTER PACKAGE、CREATE PACKAGE、第 35 章。

格式：

```
DROP PACKAGE [BODY] [schema.] package;
```

描述： DROP PACKAGE 删除指定的程序包。如果使用可选的 BODY 子句则只删除程序包体而不删除程序包说明。如果删除程序包说明，则 Oracle 将使任何依赖该程序包的对象无效。但是如果只删除程序包体，则不会如此。用户必须拥有该程序包或者具有 DROP ANY PROCEDURE 系统权限。

DROP PROCEDURE

参阅： ALTER PROCEDURE、CREATE PROCEDURE、第 35 章。

格式：

```
DROP PROCEDURE [schema.] procedure;
```

描述： DROP PROCEDURE 删除指定的过程。Oracle 使任何依赖或调用该过程的对象无效。用户必须拥有该过程或者具有 DROP ANY PROCEDURE 系统权限。

DROP PROFILE

参阅： ALTER PROFILE、CREATE PROFILE、第 19 章。

格式：

```
DROP PROFILE profile [CASCADE];
```

描述： DROP PROFILE 删除指定的配置文件。用户必须具有 DROP PROFILE 系统权限。CASCADE 将从已经分配了该配置文件的任何用户处重新分配该配置文件；然后在相应的位置自动分配 DEFAULT 配置文件。如果要把正在删除的配置文件分配给当前的用户，则必须使用 CASCADE。

DROP RESTORE POINT

参阅： FLASHBACK DATABASE、FLASHBACK TABLE、CREATE RESTORE POINT。

格式：

```
DROP RESTORE POINT restore_point;
```

描述： 使用 DROP RESTORE POINT 语句从数据库删除常规还原点或担保式恢复点。

不需要删除常规还原点。数据库会在需要的时候，按照 restore_point 语法的描述，自动删除最旧的还原点。但是，如果想重用相同的名称，则可以删除常规还原点。

担保式恢复点不会自动删除。因此，如果想要从数据库删除担保式还原点，则必须使用 DROP RESTORE POINT 语句显式地删除。

要删除常规还原点，必须具有 SELECT ANY DICTIONARY 或 FLASHBACK ANY TABLE 系统权限。要删除担保式还原点，必须具有 SYSDBA 系统权限。

DROP ROLE

参阅：ALTER ROLE、CREATE ROLE、第 19 章。

格式：

```
DROP ROLE role;
```

描述：DROP ROLE 删除指定的角色。必须已经用 WITH ADMIN OPTION 给用户授予了该角色或者用户具有 DROP ANY ROLE 系统权限。

DROP ROLLBACK SEGMENT

参阅：ALTER ROLLBACKE SEGMENT、CREATE ROLLBACK SEGMENT、CREATE TABLESPACE、SHUTDOWN、STARTUP 和第 51 章。

格式：

```
DROP ROLLBACK SEGMENT rollback_segment;
```

描述：rollback_segment 是要删除的已有回滚段的名称。当执行该语句时，一定不能正在使用该段。PUBLIC 用于删除公有回滚段。

数据字典视图 DBA_ROLLBACK_SEGS 的 Status 列可以显示哪些回滚段正在使用。如果回滚段正在使用，则可以等待直到它不再使用为止，或者使用 IMMEDIATE 来 SHUTDOWN(关闭)数据库，然后使用 STARTUP 进入 EXCLUSIVE 模式。要删除回滚段，用户必须具有 DROP ROLLBACK SEGMENT 系统权限。

DROP SEQUENCE

参阅：ALTER SEQUENCE、CREATE SEQUENCE、第 17 章。

格式：

```
DROP SEQUENCE [schema.] sequence_name;
```

描述：sequence_name 是要删除的序列名。要删除一个序列，用户必须拥有该序列或者拥有 DROP ANY SEQUENCE 系统权限。

DROP SYNONYM

参阅：CREATE SYNONYM、第 25 章。

格式：

```
DROP [PUBLIC] SYNONYM [schema.] synonym [FORCE];
```

描述：DROP SYNONYM 删除指定的同义词。要删除一个公有同义词，用户必须具有 DROP ANY PUBLIC SYNONYM 系统权限。要删除一个私有同义词，用户必须拥有该同义词或者具有 DROP ANY SYNONYM 系统权限。如果同义词依赖于表或用户定义的类型，那么用户必须指定 FORCE。

DROP TABLE

参阅：ALTER TABLE、CREATE INDEX、CREATE TABLE、DROP CLUSTER 和第 17 章。

格式：

```
DROP TABLE [schema.] table [CASCADE CONSTRAINTS] [PURGE] ;
```

描述：DROP TABLE 删除指定的表。要删除一个表，用户必须拥有该表或者具有 DROP ANY TABLE 系统权限。删除表的同时也删除了索引以及与之相关联的权限。在已删除的表上构建的对象将被标记为无效并且终止工作。

CASCADE CONSTRAINTS 选项删除所有引用已删除的表中键的参照完整性约束。

通过在 DROP CLUSTER 中使用 INCLUDING TABLES 子句可以删除一个群集及其所有的表。

使用 DROP TABLE 语句可以将表或对象表放入回收站，或者将表及其所有数据从数据库中全部删除。除非指定 PURGE 子句，否则 DROP TABLE 语句不会把释放的空间返还给表空间以备其他对象使用，并且此空间还继续算在用户的空间配额内。如果表空间用完了为新对象预留的空间，那么对象可以自动从回收站内清除。

DROP TABLESPACE

参阅：ALTER TABLESPACE、CREATE DATABASE、CREATE TABLESPACE 和第 51 章。

格式：

```
DROP TABLESPACE tablespace
  [INCLUDING CONTENTS [AND DATAFILES] [CASCADE CONSTRAINTS]];
```

描述：tablespace 是要删除的表空间的名字。INCLUDING CONTENTS 选项允许删除包含数据的表空间。如果没有 INCLUDING CONTENTS，则只能删除空的表空间。表空间在删除前应该脱机(请参阅 ALTER TABLESPACE)，或者说，如果任何用户正在访问表空间中的数据、索引、回滚段或临时段，那么将阻止删除操作。用户必须具有 DROP TABLESPACE 系统权限才能使用该命令。

DROP TRIGGER

参阅：ALTER TRIGGER、CREATE TRIGGER、第 34 章和第 38 章。

格式：

```
DROP TRIGGER [schema.] trigger;
```

描述：DROP TRIGGER 删除指定的触发器。用户必须拥有该触发器或者具有 DROP ANY TRIGGER 系统权限。

DROP TYPE

参阅： CREATE TYPE、第 38 章。

格式：

```
DROP TYPE [schema.] type_name [ FORCE | VALIDATE ];
```

描述： DROP TYPE 删除一个抽象数据类型的说明和类型体。用户必须拥有该类型或者具有 DROP ANY TYPE 系统权限。除非使用 FORCE，否则不能删除一个被表或其他对象引用的类型。

DROP TYPE BODY

参阅： CREATE TYPE、CREATE TYPE BODY、DROP TYPE、第 38 章。

格式：

```
DROP TYPE BODY [schema.] type_name;
```

描述： DROP TYPE BODY 删除指定类型的类型体。用户必须拥有该类型或者具有 DROP ANY TYPE 系统权限。

DROP USER

参阅： ALTER USER、CREATE USER、第 19 章。

格式：

```
DROP USER user [CASCADE];
```

描述： DROP USER 删除指定的用户。用户必须具有 DROP USER 系统权限。CASCADE 选项在删除该用户前先删除该用户的模式中的所有对象。而且，如果用户在该模式中有对象，就必须指定 CASCADE。

DROP VIEW

参阅： CREATE SYNONYM、CREATE TABLE、CREATE VIEW、第 17 章。

格式：

```
DROP VIEW [schema.] view [CASCADE CONSTRAINTS];
```

描述： DROP VIEW 删除指定的视图。要删除一个视图，要么该视图必须在用户的模式中，要么用户必须具有 DROP ANY VIEW 系统权限。

DROPJAVA

DROPJAVA 是一个实用程序，它从数据库中删除 Java 类。请参阅 LOADJAVA。

DUAL

参阅： FROM、SELECT、第 10 章。

描述： DUAL 是一个只有一行和一列的表。由于 Oracle 的许多函数都可以作用于列和字

面量，因此可以使用字面量或伪列来说明某些功能。当这样做时，SELECT 语句不会关心哪些列是属于该表的，而且一行就足以说明某个要点了。

示例：下例给出了当前 User 和 SysDate：

```
select User, SysDate from DUAL;
```

DUMP

参阅：RAWTOHEX。

格式：

```
DUMP( string [, format [, start [, length] ] ] )
```

描述：DUMP 用内部数据格式(即 ASCII、八进制、十进制、十六进制或字符格式)显示 string 的值。format 默认为 ASCII 或 EBCDIC，这取决于具体的机器；8 为八进制，10 为十进制，16 为十六进制，17 为字符(与 ASCII 或 EBCDIC 相同)。start 是该串的开始位置，length 是要显示的字符数。string 可以为字面量或表达式。

示例：下例说明了 COMFORT 表的第一行中 City 的 1~8 个字符是如何用十六进制表示：

```
select City, DUMP(City,16,1,8) from COMFORT where rownum < 2;

CITY           DUMP(CITY,16,1,8)
-------------  ------------------------------------
SAN FRANCISCO  Typ=1 Len= 13: 53,41,4e,20,46,52,41,4e
```

EDIT(SQL*Plus)

参阅：DEFINE、SET、第 6 章。

格式：

```
ED[IT] [file[.ext]]
```

描述：EDIT 调用一个外部的标准文本编辑器并将文件名传递给该编辑器。如果省略.ext，则追加扩展名.sql。如果 file 和.ext 都被省略，则调用编辑器并将一个包含当前缓冲区内容的文件(由 SQL*Plus 构建的)的名字传递给它。本地用户变量_EDITOR 决定 EDIT 使用哪个文本编辑器。_EDITOR 可以用 DEFINE 更改，通常该操作最好在 login.sql 文件中进行，无论何时在调用 SQL*Plus 时都会读取该文件。

如果当前缓冲区为空并且调用 EDIT 时不带文件名，则 EDIT 将失败。可以使用 SET EDITFILE 命令设置由 EDIT 命令创建的文件的默认文件名。

EMPTY_BLOB

参阅：第 40 章。

格式：

```
EMPTY_BLOB()
```

描述：EMPTY_BLOB 返回一个空的 LOB 定位器，该定位器可用来初始化 LOB 变量，或者在 INSERT 或 UPDATE 语句中把一个 LOB 列或属性初始化为 EMPTY。EMPTY 表示初始化 LOB，但不填充数据。

示例：

```
UPDATE BOOK_TEXT set BlobCol = EMPTY_BLOB();
```

EMPTY_CLOB

参阅：第 40 章。

格式：

```
EMPTY_CLOB()
```

描述：EMPTY_CLOB 返回一个空的 LOB 定位器，该定位器可用来初始化 LOB 变量，或者在 INSERT 或 UPDATE 语句中把一个 LOB 列或属性初始化为 EMPTY。EMPTY 表示初始化 LOB，但不填充数据。

示例：

```
UPDATE BOOK_TEXT set ClobCol = EMPTY_CLOB();
```

END

参阅：BEGIN、BLOCK STRUCTURE、第 32 章。

格式：

```
END [block label] ;
```

描述：END 与 END IF 和 END LOOP 语句无关。关于这两个语句的详细内容请参阅 IF 和 LOOP。

END 是 PL/SQL 块的可执行部分(以及整个块)的结束语句。如果该块在 BEGIN 部分命名，那么在单词 END 后面也必须给出其名称。在 BEGIN 和 END 之间至少需要一个可执行语句。

EXCEPTION

参阅：BLOCK STRUCTURE、RAISE、第 32 章。

格式：

```
EXCEPTION
  {WHEN {OTHERS | exception [OR exception]...}
        THEN statement; [statement;]...}
  [ WHEN {OTHER | exception [OR exception]...}
        THEN statement; [statement;]...]...
```

描述：PL/SQL 块的 EXCEPTION 部分在出现异常标志时用来传递程序控制权。异常标志可以是用户定义的，也可以是 PL/SQL 自动引发的系统异常。关于用户定义的异常标志的详细内容请参阅 RAISE。系统异常标志实际上都是 BOOLEAN 变量，其值为 TRUE 或 FALSE。

WHEN 子句可用来测试这些变量。例如，如果用户在没有登录的情况下试图向 Oracle 发布一个命令，则会引发 NOT_LOGGED_ON 异常标志。用户不需要自己声明(DECLARE)一个异常或引发(RAISE)该标志，PL/SQL 将完成这些工作。

当引发任何异常标志时，程序立刻暂停它所进行的任何工作，并跳转到当前块的 EXCEPTION 部分。但是，该部分既不能自动知道引发的是哪一个异常标志，也不知道应该执行什么操作。因此，必须首先对 EXCEPTION 部分进行编码，以检查所有可能出现的异常标志，然后为每个异常标志给出操作指令。这就是 WHEN…THEN 逻辑所做的工作。可以使用 WHEN OTHERS 来捕捉和处理未声明的异常的所有没有预料到的错误。

示例：这里，在 EXCEPTION 部分测试一个异常标志。当计算试图除以零时引发异常 ZERO_ DIVIDE 标志。

```
declare
  pi constant NUMBER(9,7) := 3.1415927;
  radius INTEGER(5);
  area NUMBER(14,2);
  some_variable NUMBER(14,2);
begin
  radius := 3;
  loop
    some_variable := 1/(radius-4);
    area := pi*power(radius,2);
      insert into AREAS values (radius, area);
    radius := radius+1;
    exit when area \> 100;
  end loop;
exception
  when ZERO_DIVIDE
    then insert into AREAS values (0,0);
end;
```

系统引发的异常标志如表 A-14 所示。

表 A-14　系统引发的异常标志

异 常 标 志	说　明
ACCESS_INTO_NULL	程序试图给未初始化的(自动为空)对象的属性赋值
CASE_NOT_FOUND	没有选择 CASE 语句的 WHEN 子句中的任何选项，而且没有 ELSE 子句
COLLECTION_IS_NULL	程序试图对未初始化的(自动为空)的嵌套表或可变数组使用不同于 EXISTS 的集合方法，或者程序试图给未初始化的嵌套表或可变数组的元素赋值
CURSOR_ALREADY_OPEN	程序试图打开一个已经打开的游标。游标在重新打开前必须关闭。CURSOR FOR LOOP 自动打开它所引用的一个游标。因此，程序不能在循环内部打开相应的游标
DUP_VAL_ON_INDEX	程序试图在由唯一索引约束的数据库列中存储重复的值
INVALID_CURSOR	程序试图执行非法的游标操作，如关闭一个未打开的游标
INVALID_NUMBER	在 SQL 语句中，由于字符串不表示有效的数字而导致从字符串转换为数字的操作失败(在过程语句中，将引发 VALUE_ERROR 错误)。当批处理 FETCH 语句中的 LIMIT 子句表达式的值为非正数时，也会引发该异常

(续表)

异常标志	说 明
LOGIN_DENIED	程序试图用一个无效的用户名和/或口令登录到 Oracle
NO_DATA_FOUND	SELECT INTO 语句不返回任何行，或者程序引用嵌套表中已经删除的元素或索引表中未初始化的元素。SQL 聚集函数，如 AVG 和 SUM 总是返回一个值或一个空值。因此，调用聚集函数的 SELECT INTO 语句绝不会引发 NO_DATA_FOUND 异常标记。因为期望 FETCH 语句最终不返回任何行，所以在出现这种情况时，不引发任何异常
NOT_LOGGED_ON	程序发布一个没有连接到 Oracle 的数据库调用
PROGRAM_ERROR	PL/SQL 有内部问题
ROWTYPE_MISMATCH	分配中所涉及的宿主游标变量和 PL/SQL 游标变量具有不兼容的返回类型。例如，在将一个打开的宿主游标变量传递给一个存储子程序时，实参和形参的返回类型必须兼容
SELF_IS_NULL	程序试图在空的实例中调用 MEMBER 方法。即内置参数 SELF(它总是传递给 MEMBER 方法的第一个参数)为空
STORAGE_ERROR	PL/SQL 用完内存或内存损坏
SUBSCRIPT_BEYOND_COUNT	程序使用大于集合中元素个数的索引号引用嵌套表或可变的数组元素
SUBSCRIPT_OUTSIDE_LIMIT	程序使用超出合法范围的(如－1)引用嵌套表或可变的数组元素
• SYS_INVALID_ROWID	从字符串到通用 ROWID(行标识符)的转换由于字符串不表示一个有效的 ROWID 而失败
TIMEOUT_ON_RESOURCE	在 Oracle 等待某个资源时超时
TOO_MANY_ROWS	SELECT INTO 语句返回多行
VALUE_ERROR	出现算术、转换、截断或大小约束错误。例如，在程序选择一个列值放入一个字符变量时，如果该值比变量所声明的长度长，则 PL/SQL 终止赋值，并引发 VALUE_ERROR 异常。在过程语句中，如果字符串到数字的转换失败，就也引发 VALUE_ERROR 异常(在 SQL 语句中，引发 INVALID_NUMBER 异常)
ZERO_DIVIDE	程序试图用零作为除数

相关联的 Oracle 错误号和 SQLCode 值如表 A-15 表示。

表 A-15 相关联的 Oracle 错误号和 SQLCocle 值

异 常	Oracle 错误号	SQLCode
ACCESS_INTO_NULL	ORA-06530	–6530
CASE_NOT_FOUND	ORA-06592	–6592
COLLECTION_IS_NULL	ORA-06531	–6531
CURSOR_ALREADY_OPEN	ORA-06511	–6511
DUP_VAL_ON_INDEX	ORA-00001	–1
INVALID_CURSOR	ORA-01001	–1001
INVALID_NUMBER	ORA-01722	–1722
LOGIN_DENIED	ORA-01017	–1017

(续表)

异　常	Oracle 错误号	SQLCode
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

OTHERS 用于捕捉 EXCEPTION 部分中所有未检查到的任何异常标志。它总是最后一条 WHEN 语句，并且必须是独立的。它不能包含在其他任何异常中。

异常处理(尤其是当它与您声明的异常相关时)应该是为致命性错误(即导致正常处理中断的错误)而准备的。

如果引发了一个没有在当前块中检测到的异常标志，则程序将跳转到该封闭块的 EXCEPTION 块中，等等，直到发现引发的异常，或者控制跳转到宿主程序为止。

EXCEPTION 部分可以用与执行块相同的方式引用变量。即它们可以直接引用局部变量，或者通过以其他块的名字作为前缀来引用其他块的变量。

应该在某些异常测试中使用警告。例如，如果 EXCEPTION 部分接受一个错误消息并将其插入到数据库表中，则 NOT_LOGGED_ON 异常将使 EXCEPTION 部分进入无限循环。应设计 THEN 后面的语句，从而使程序不在第一个地方出现重复性的错误。还可以使用不带异常名的 RAISE，这将把控制权自动传递给下一个外部 EXCEPTION 块，或者主程序。更多的信息请参阅 RAISE。

EXCEPTION_INIT

参阅：DECLARE CURSOR、EXCEPTION、SQLCODE 和 *Programmer's Guide to the Oracle Precompilers*。

格式：

```
PRAGMA EXCEPTION_INIT(exception, integer);
```

描述：标准的系统异常处理，如 ZERO_DIVIDE，是通过名字引用。这些异常实际上只是与内部的 Oracle 错误号相关联的名字。有几百个这样的错误号，但只有最常用的那些错误号有名字。未命名的那些错误仍然可以引发异常标志并将控制权传递给 EXCEPTION 块，但它们都将由 OTHERS 捕获，而不是由名字捕获。

可以通过将自己的名字分配给其他的 Oracle 错误号来改变这种情况，EXCEPTION_INIT 可以做到这一点。exception 是分配给整数错误号的一个名字(完整的列表请参阅 Oracle 的 Error Messages and Codes Manual)。如果错误代码为负数(对于致命性错误为真)，则 integer 应当为负数，并且 exception 必须遵照常规的对象命名规则。

请注意，该命令的格式需要在 EXCEPTION_INIT 之前加 PRAGMA。编译指示(pragma)是用于 PL/SQL 编译器的指令，而不是可执行代码。编译指示必须在 PL/SQL 块的 DECLARE 部分中，并且必须放在异常声明的前面。

示例：

```
DECLARE
    some_bad_error        exception;
    pragma                exception_init(some_bad_error, -660);
BEGIN
  ...
  EXCEPTION
    when some_bad_error
       then ...
END;
```

EXECUTE

参阅：CALL、CREATE FUNCTION、CREATE PACKAGE、CREATE PACKAGE BODY、CREATE PROCEDURE、GRANT 和第 35 章。

格式：

```
EXECUTE procedural_object_name [arguments];
```

描述：

EXECUTE 执行一个过程、程序包或函数。为了执行程序包中的过程，应在 EXECUTE 命令中指定程序包名和过程名，如下例所示。该示例执行一个名为 NEW_BOOK 的过程，此过程在一个名为 BOOKSHELF_PACKAGE 的程序包中。值‘TALE OF TWO CITIES’将作为输入传递给该过程。

```
execute BOOKSHELF_PACKAGE.NEW_BOOK('TALE OF TWO CITIES');
```

要执行一个过程对象，用户必须具有该对象的 EXECUTE 权限。详细内容请参阅 GRANT。

EXECUTE(动态嵌入的 SQL)

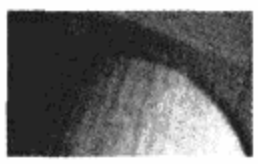
参阅：EXECUTE IMMEDIATE、PREPARE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [FOR { integer | :array_size }]
  EXECUTE statement_id
  [USING
    { DESCRIPTOR SQLDA_descriptor
    | :host_variable [[INDICATOR] :indicator_variable]
      [, :host_variable [[INDICATOR] :indicator_variable]]... } ]
```

描述：当 WHERE 子句使用数组时，integer 是用来限制迭代次数的宿主变量。statement_id 是要执行的 INSERT、DELETE 或 UPDATE 语句(不允许使用 SELECT 语句)的标识符。USING

将一列替换值引入先前准备好的语句的宿主变量中。



注意：
关于所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

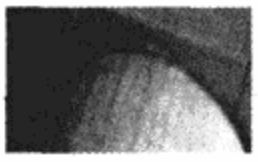
EXECUTE IMMEDIATE(动态嵌入的 SQL)

参阅：EXECUTE、PREPARE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [AT {database | :host_variable}]
EXECUTE IMMEDIATE {:string | literal}
```

描述：database 是不包含默认名在内的数据库连接名。host_variable 可能包含名字这样的内容作为其值。string 是包含 SQL 语句的宿主变量串。literal 是包含 SQL 语句的字符串。EXECUTE IMMEDIATE 语句不能包含 string 中除可执行的 SQL 语句之外的宿主变量引用。SQL 语句经过分析转换成可执行的格式并执行，然后销毁可执行的格式。它只用于执行一次的语句。需要多次执行的语句应该使用 PREPARE 和 EXECUTE，这消除了每次执行时分析语句的系统开销。



注意：
关于所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

EXISTS

参阅：ALL、ANY、IN、NOT EXISTS、第 13 章。

格式：

```
select ...
 where EXISTS (select...)
```

描述：如果跟在 WHERE 子句后面的子查询至少返回一行，则 EXISTS 返回 TRUE。子查询中的 SELECT 子句可以为一列、一个字面量或一个星号，这无关紧要(约定建议使用一个星号或一个“x”)。

与 ANY 和 ALL 相比，许多人更喜欢使用 EXISTS，因为它更容易记忆和理解，并且 ANY 和 ALL 的大多数格式都可以用 EXISTS 来重新构造。

示例：下例中的查询利用 EXISTS 从 BOOKSHELF 表中列出了已经有书籍借出的所有 Publisher：

```
select Publisher
  from BOOKSHELF B
 where EXISTS
       (select 'x' from BOOKSHELF_CHECKOUT
       where BOOKSHELF_CHECKOUT.Title = B.Title);
```

EXISTSNODE

参阅：第 47 章。

格式：

```
EXISTSNODE ( XMLType_instance , Xpath_string )
```

描述：EXISTSNODE 决定使用路径遍历文档是否会导致节点。它接受一个包含 XML 文档和指定路径的 VARCHAR2 串的 XMLType 实例作为参数。

返回值为 NUMBER：如果在文档上应用 XPath 遍历后不剩下任何节点，则返回值为 0；如果有节点剩下，则返回值大于 0。

EXIT(格式 1——PL/SQL)

参阅：END、LOOP、第 32 章。

格式：

```
EXIT [loop] [WHEN condition];
```

描述：如果没有任何选项，EXIT 就使用户退出当前的循环，并将控制权转移到该循环后面的下一个语句中。如果处于一个嵌套循环中，则可以通过指定循环的名字从任何闭合循环中退出。如果指定一个条件，则当该条件为 TRUE 时退出。循环中的任何游标都将在 EXIT 时自动关闭。

EXIT(格式 2——SQL*Plus)

参阅：COMMIT、DISCONNECT、QUIT、START。

格式：

```
{EXIT | QUIT} [SUCCESS|FAILURE|WARNING|integer|variable|:BindVariable]
[COMMIT|ROLLBACK]
```

描述：EXIT 结束 SQL*Plus 会话并使用户返回到操作系统、主调程序或菜单。除非指定 ROLLBACK，否则，EXIT 将挂起的更改提交给数据库。而且，还返回一个返回码。SUCCESS、FAILURE 和 WARNING 具有特定于操作系统的值，FAILURE 和 WARNING 的值在某些操作系统上可能相同。

integer 是可以作为返回码显式传回的值，variable 允许动态设置该值。variable 可以是用户定义的变量，或者系统变量，如 SQL.SQLCODE，它总是包含最后执行的 SQL 语句的 SQLCODE，该语句或者在 SQL*Plus 中，或者在嵌入式 PL/SQL 块中。

EXP

参阅：LN、NUMBER FUNCTIONS、第 9 章。

格式：

```
EXP(n)
```

描述：EXP 返回 e 的 n 次幂；e=2.718281828……

EXPLAIN PLAN

参阅：第 46 章。

格式：

```
EXPLAIN PLAN [SET STATEMENT_ID = 'name']
    [INTO [schema.] table [@ dblink]]
    FOR statement;
```

描述：name 标识 statement(当它出现在输出表中时)的说明计划，并且遵循常规的对象命名约定。如果不指定 name，则表中的 STATEMENT_ID 列将为 NULL。table 是说明计划将要进入的输出表的名字。它必须在该过程执行前创建。启动文件 utlxplan.sql 包含格式并且可以用来创建默认表 PLAN_TABLE。如果不指定 table，则 EXPLAIN PLAN 将使用 PLAN_TABLE 表。statement 是需要为 Oracle 的执行计划进行分析的任何 SELECT、INSERT、UPDATE 或 DELETE 语句的简单文本。

EXTRACT(datetime)

参阅：DATE FUNCTIONS、第 10 章。

格式：

```
EXTRACT
( { { YEAR
    | MONTH
    | DAY
    | HOUR
    | MINUTE
    | SECOND }
  | { TIMEZONE_HOUR
    | TIMEZONE_MINUTE }
  | { TIMEZONE_REGION
    | TIMEZONE_ABBR
} }
FROM { datetime_value_expression | interval_value_expression } )
```

描述：EXTRACT 从日期时间或时间间隔表达式中提取和返回指定日期时间字段的值。

示例：

```
select BirthDate,
       EXTRACT(Month from BirthDate) AS Month
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTHDATE        MONTH
---------  ----------
20-MAY-49     .      5
```

EXTRACT(XML)

参阅： 第 52 章。

格式：

```
EXTRACT (XMLType_instance, Xpath_string)
```

描述： EXTRACT 类似于 EXISTSNODE 函数。它应用 VARCHAR2 XPath 串并返回一个包含 XML 段的 XMLType 实例。

EXTRACTVALUE

参阅： XMLType。

格式：

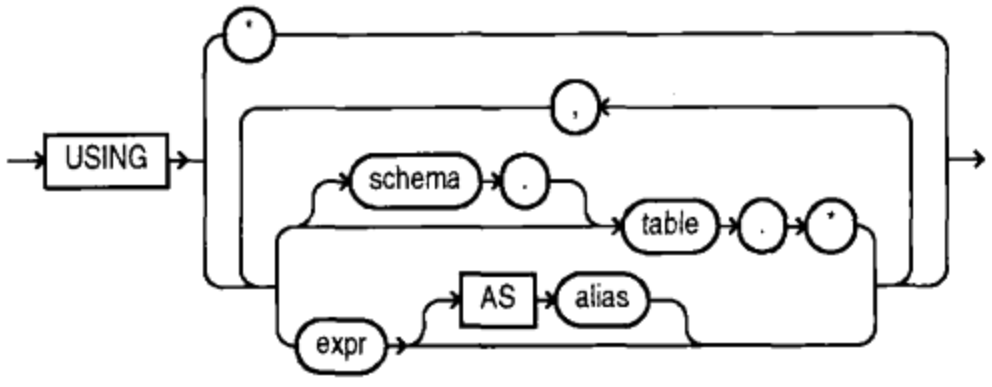
```
EXTRACTVALUE(XMLType_instance, Xpath_string [,namespace_string])
```

描述： EXTRACTVALUE 将一个 XMLType 实例和一个 XPath 表达式作为参数，并且返回结果节点的标量值。该结果必须是孤立节点，且必须是文本节点、属性或元素。通过比较可知，EXTRACT 函数返回一个 XML 段；而 EXTRACTVALUE 返回 XML 段的标量值。

FEATURE_ID

格式：

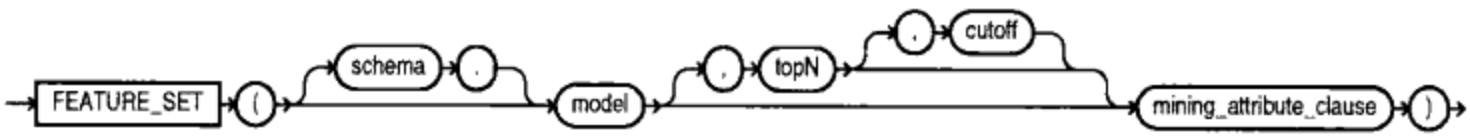
mining_attribute_clause:=



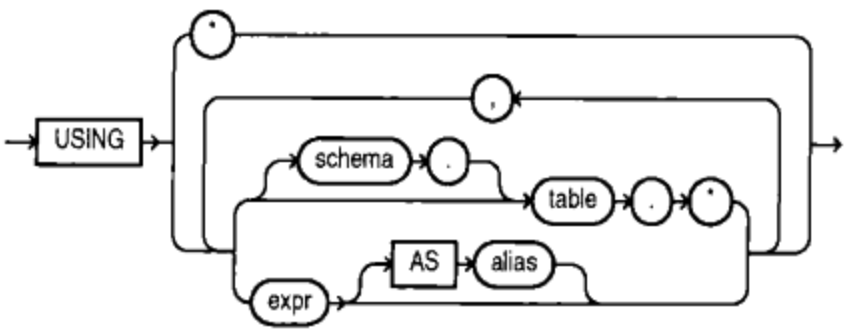
描述： 此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的特征抽取模型。它返回一个 Oracle NUMBER 类型的值，该值是行中最大值的特征标识符。

FEATURE_SET

格式：



mining_attribute_clause:=

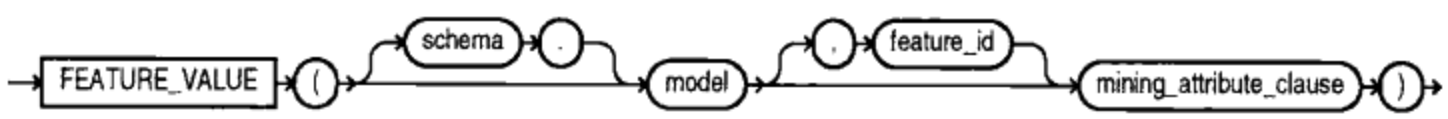


描述：此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的特征抽取模型。它返回一个对象可变数组，包含了所有可能的特征。可变数组中的每个对象都是一对标量值，包含特征 ID 和特征值。对象字段命名为 FEATURE_ID 和 VALUE，二者都是 Oracle NUMBER 类型。

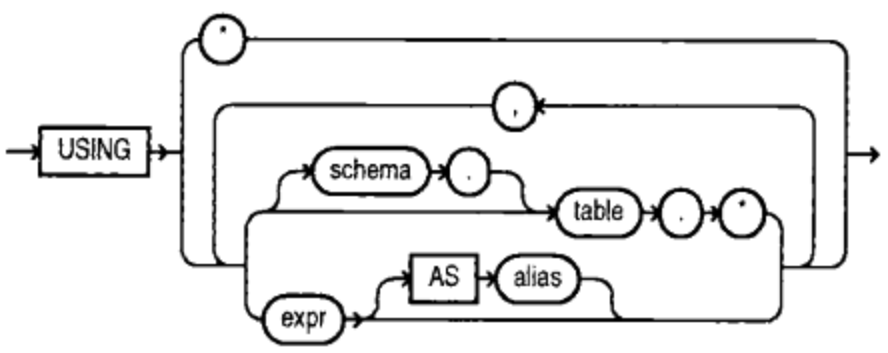
可选的 topN 参数是一个正整数，它把特征集限制为最大的 N 个值中的一个值的那些特征。如果与第 N 个值有关联，则数据库仍然只返回 N 个值。如果省略此参数，则函数返回所有特征。

FEATURE_VALUE

格式：



mining_attribute_clause:=



描述：此函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的特征抽取模型。它返回给定特征的值。如果省略 feature_id 参数，则函数返回最大的特征值。此形式可以与 FEATURE_ID 函数一同使用以获得最大的特征/值组合。

FETCH(格式 1——嵌入式 SQL)

参阅：CLOSE、DECLARE CURSOR、DESCRIBE、INDICATOR VARIABLE、OPEN、PREPARE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [FOR { integer | :array_size }]
  FETCH { cursor | :cursor_variable }
  { USING DESCRIPTOR SQLDA_descriptor
  | INTO :host_variable [[INDICATOR] :indicator_variable]
    [, :host_variable [[INDICATOR] :indicator_variable]]... }
```

描述：integer 是一个宿主变量，它设置要取出并放入输出变量中的数据行的最大值。cursor 是由 DECLARE CURSOR 预先设置的游标名。“:host_variable”是将要放置取出数据的一个或多个宿主变量。如果宿主变量中的任何一个是数组，则所有变量(在 INTO 列表中)必须为数组。INDICATOR 关键字是可选的。

注意：

关于所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

FETCH(格式 2——PL/SQL)

参阅：%FOUND、%ROWTYPE、CLOSE、DECLARE CURSOR、LOOP、OPEN、SELECT…INTO 和第 32 章。

格式：

```
FETCH cursor INTO {record | variable [,variable]...};
```

描述：FETCH 获得一行数据。给定游标的 SELECT 语句确定要检索哪些列，它的 WHERE 语句确定可以检索哪些行以及多少行，这称为“活动集”，但是，除非使用 FETCH 语句逐行地取出该活动集，否则不能处理它。FETCH 从该活动集中获得一行并将此行的值放入记录(或变量的串)中，该记录已经在 DECLARE 中定义。

如果使用变量列表方法，则该游标的选择列表中的每列都必须有一个相应的变量，并且每一列必须已经在 DECLARE 部分中声明。数据类型必须是相同的或兼容的。

如果使用记录方法，则该记录使用%ROWTYPE 属性进行声明，它声明了与该 SELECT 中列列表相同(具有相同的数据类型)的记录的结构。然后该记录中的每个变量分别使用记录名作为前缀进行访问，并且变量名与列名相同。

示例：

```
declare
  pi constant   NUMBER(9,7) := 3.1415927;
  area          NUMBER(14,2);
  cursor rad_cursor is
           select * from RADIUS_VALS;
  rad_val rad_cursor%ROWTYPE;
begin
  open rad_cursor;
    loop
      fetch rad_cursor into rad_val;
```

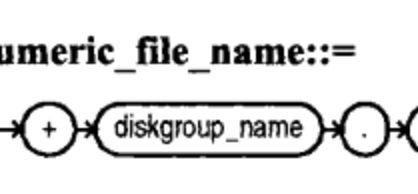
```
      exit when rad_cursor%NOTFOUND;
        area := pi*power(rad_val.radius,2);
        insert into AREAS values (rad_val.radius, area);
    end loop;
  close rad_cursor;
end;
```

file_specification

参阅：ALTER TABLESPACE、CREATE CONTROLFILE、CREATE DATABASE、CREATE LIBRATY 和 CREATE TABLESPACE。

格式：

file_specification::=



datafile_tempfile_spec::=

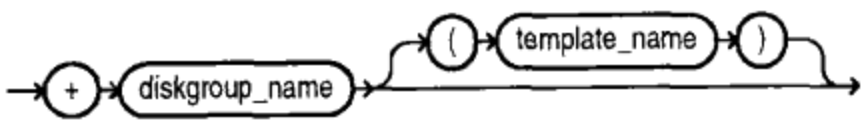
redo_log_file_spec::=

ASM_filename::=

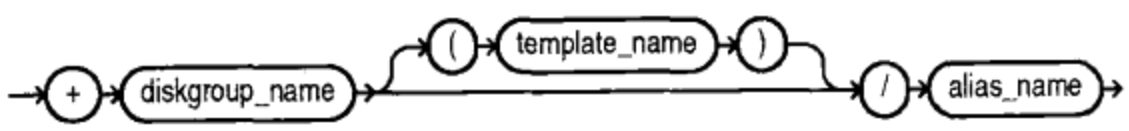
fully_qualified_file_name::=

numeric_file_name::=

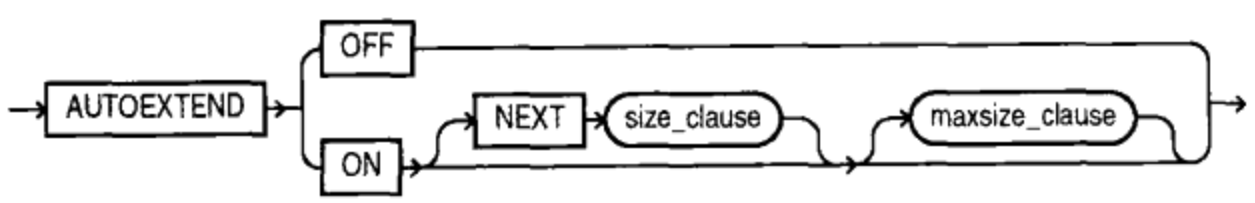
incomplete_file_name::=



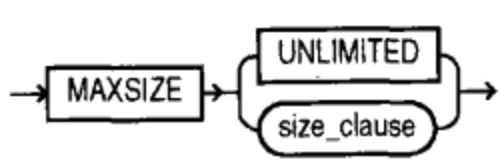
alias_file_name::=



autoextend_clause::=



maxsize_clause::=



描述：file_specification 定义数据库使用的文件的名字、位置和大小。例如，在创建一个表空间时，必须为表空间至少指定一个要使用的数据文件。file_specification 为 CREATE TABLESPACE 命令的数据文件部分提供了格式。

文件说明部分的子子句 datafile_tempfile_spec 和 redo_log_file_spec 除可以指定文件系统类型的文件外，还允许指定自动存储管理文件。

FIRST

参阅：DENSE_RANK、LAST。

格式：

```
aggregate_function KEEP
  ( DENSE_RANK FIRST ORDER BY
    expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]
    [, expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]]...
  ) [OVER query_partitioning_clause]
```

描述：FIRST 既是一个聚集函数又是一个分析函数，它作用于来自一组行的一组值上，这组行按给定的分类说明排在最前面。aggregate_function 为标准的数值分组函数之一(如 MIN、MAX、AVG 和 STDDEV)。

FIRST_VALUE

参阅：LAST_VALUE。

格式：

```
FIRST_VALUE (expr) [IGNORE NULLS] OVER (analytic_clause)
```

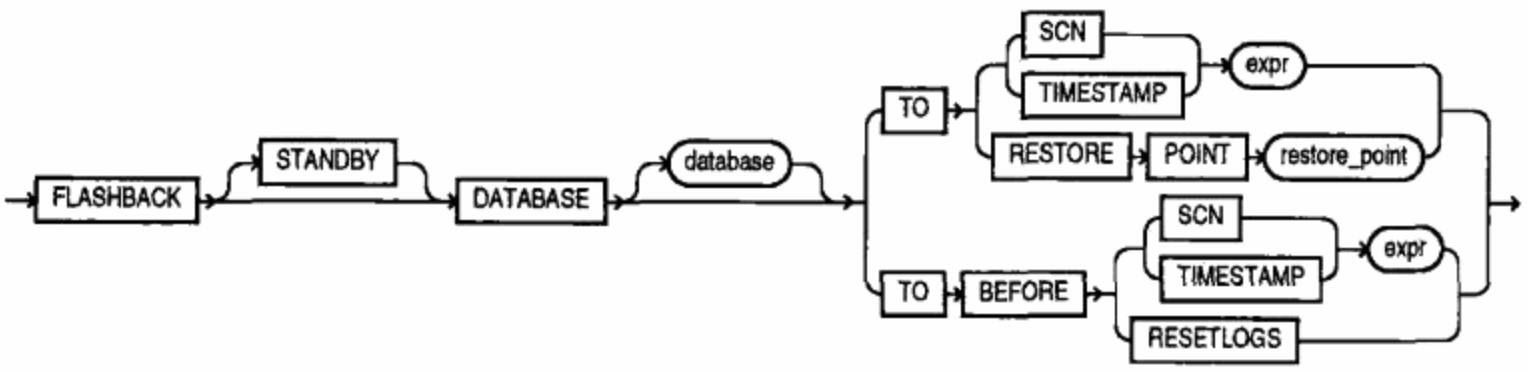
描述：FIRST_VALUE 是一个分析函数。它返回一组有序值中的第一个值。如果该组的第一个值为空，则 FIRST_VALUE 返回 NULL，除非指定 IGNORE NULLS。

FLASHBACK DATABASE

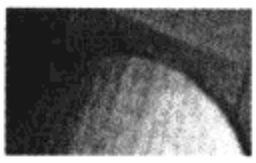
参阅：FLASHBACK TABLE、第 30 章。

格式：

flashback_database::=



描述：FLASHBACK DATABASE 将数据库返回到过去时间或系统更改号(SCN)。该命令为执行不完全数据库恢复提供了一种快速的选择。执行 FLASHBACK DATABASE 操作后，为了能对闪回后的数据库进行写访问，必须用 ALTER DATABASE OPEN RESETLOGS 命令重新打开该数据库。



注意：

必须用 ALTER DATABASE FLASHBACK ON 命令将数据库置于 FLASHBACK 模式。数据库必须以 EXCLUSIVE 模式安装，但是没有打开。

可以通过查询 V$FLASHBACK_DATABASE_LOG 视图来确定数据库闪回的程度。保留在数据库中的闪回数据的数量由 DB_FLASHBACK_RETENTION_TARGET 初始化参数和快速恢复区域的大小来控制。

示例：

```
STARTUP MOUNT EXCLUSIVE;
ALTER DATABASE FLASHBACK ON;
ALTER DATABASE OPEN;
```

如果数据库打开的时间已经超过一个小时，则将它闪回：

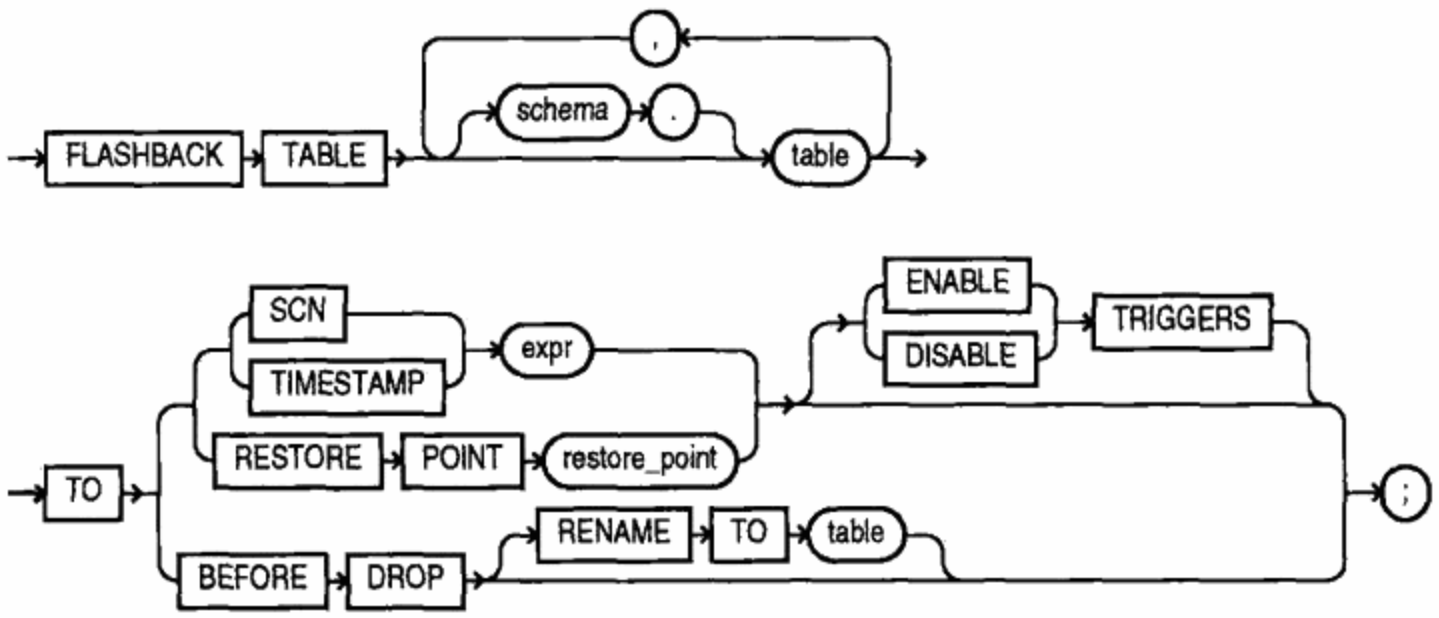
```
SHUTDOWN DATABASE;
STARTUP MOUNT EXCLUSIVE;
FLASHBACK DATABASE TO TIMESTAMP SYSDATE-1/24;
```

FLASHBACK TABLE

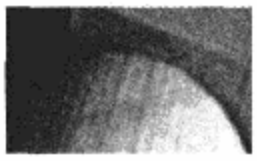
参阅：FLASHBACK DATABASE、第 30 章。

格式：

flashback_table::=



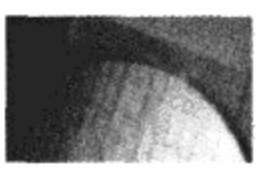
描述：FLASHBACK TABLE 在出现人为的或应用程序错误时把表还原到先前的状态。Oracle 不能在更改表结构的任何 DDL 操作中将表还原到先前的状态。



注意：

数据库必须正在使用 FLASHBACK TABLE 的 Automatic Undo Management (自动撤消管理)进行工作。闪回到旧数据的功能受到保留在撤消表空间中的撤消数量以及 UNDO_RETENTION 初始化参数设置的限制。

FLASHBACK TABLE 语句不能回滚。但是，可以发布另一个 FLASHBACK TABLE 语句并指定一个先于当前时间的时间。

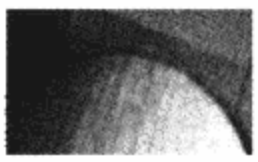


注意：

发布 FLASHBACK TABLE 命令前要记录当前的 SCN。

用户必须具有该表的 FLASHBACK 对象权限或者 FLASHBACK ANY TABLE 系统权限。用户还必须具有该表的 SELECT、INSERT、DELETE 和 ALTER 对象权限。闪回列表中的所有表都必须启用了行移动(row movement)。要将表闪回到一个DROP TABLE操作以前的状态，只需要具有删除表的权限。

在 FLASHBACK TABLE 操作过程中，Oracle 在所有指定表上获得排他的 DML 锁。该命令在单个事务中执行，该事务分布在所有的表上。如果它们中任何一个失败，则整个语句失败。



注意：
FLASHBACK TABLE TO SCN 或 FLASHBACK TABLE TO TIMESTAMP 不保存 ROWID。FLASHBACK TABLE TO BEFORE DROP 不恢复参照约束。

在闪回一个表时，表的统计信息没有闪回。存在于表中的索引被恢复并反映表在闪回点的状态。在闪回点删除的索引没有还原。在闪回点后面创建的索引继续存在并且将被更新以反映更旧的数据。

要查看回收站的内容，查询 USER_RECYCLEBIN。

示例：

```
FLASHBACK TABLE emp TO BEFORE DROP;
```

FLOOR

参阅：CEIL、NUMBER FUNCTIONS、第 9 章。

格式：

```
FLOOR(value)
```

描述：FLOOR 是小于或等于 value 的最大整数。

示例：

```
FLOOR(2) = 2
FLOOR(-2.3) = -3
```

FOR

请参阅 LOOP 和 CURSOR FOR LOOP。

FORMAT

请参阅 BTITLE、COLUMN、DATE FORMAT、TO_CHAR、TO_DATE 和 TTITLE。

FROM

参阅：DELETE、SELECT、第 5 章。

格式：

```
DELETE FROM [user.]table[@link] [alias]
  WHERE condition

SELECT... FROM [user.]table[@link] [, [user.]table[@link] ]...
```

描述：table 是 DELETE 或 SELECT 使用的表名。link 是到远程数据库的链接。DELETE 和 SELECT 命令都需要用 FROM 子句来定义要从中删除或选择行的那些表。如果该表由另一个用户所拥有，则它的名字必须用该所有者的用户名做前缀。

如果表是远程的，则必须定义数据库链接。@link 指定该链接。如果输入了 link 值但未

输入 user，则该查询将在远程数据库中寻找该用户所拥有的表。condition 可以包含相关的查询并使用相关的 alias。

FROM_TZ

参阅： TIMESTAMP WITH TIME ZONE。

格式：

```
FROM_TZ (timestamp_value , time_zone_value)
```

描述： FROM_TZ 将一个时区的时间戳值转换为一个 TIMESTAMP WITH TIME ZONE 值。time_zone_value 是一个格式为“TZH:TZM”的字符串，或用可选的 TZD 格式的 TZR 返回一个串的字符表达式。

GET(SQL*Plus)

参阅： EDIT、SAVE。

格式：

```
GET file[.ext] [LIS[T] | NOL[IST]]
```

描述： GET 将名为 file 的宿主系统文件加载到当前缓冲区(SQL 缓冲区或指定的缓冲区)中。如果不给出文件类型，则 GET 假定文件类型为 SQL。LIST 使 SQL*Plus 列出正在加载到缓冲区中的行，这是默认值。NOLIST 使文件不列出其内容。如果没有指定文件的扩展名，则默认值为.sql。要更改默认的文件扩展名，使用 SET SUFFIX 命令；详细内容请参阅 SET。

示例： 该示例得到一个名为 work.sql 的文件。

```
get work
```

GOTO

参阅： BLOCK STRUCTURE、第 34 章。

格式：

```
GOTO label;
```

描述： GOTO 将控制权转移到指定的 label 后面的代码部分。这样的标号可以放在一个执行块或者一个 EXCEPTION 部分中的任何合法的语句前面。其中，语句存在着某些限制：

- GOTO 不能将控制权转移到包含在当前块中、或 FOR 循环中、或 IF 语句中的块上。
- GOTO 不能将控制权转移到属于它自己的块以外的标签上，除非它是包含自己块中的一块。

示例： 以下是一个用 GOTO 构建的非正规循环，实际上事先并不知道循环的最大次数是多少，就可以将从 1~10 000(大约)的等比数列插入(INSERT)一个表中：

```
DECLARE
  ...
BEGIN
   x := 0;
```

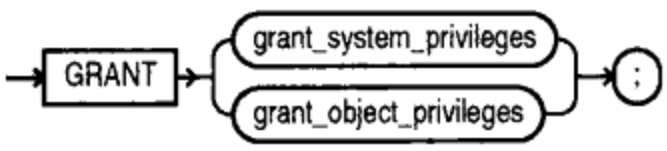
```
    y := 1;
 <<alpha>>
    x := x + 1;
    y := x*y;
    insert ...
    if y > 10000
       then goto beta;
     goto alpha;
 <<beta>>
     exit;
```

GRANT

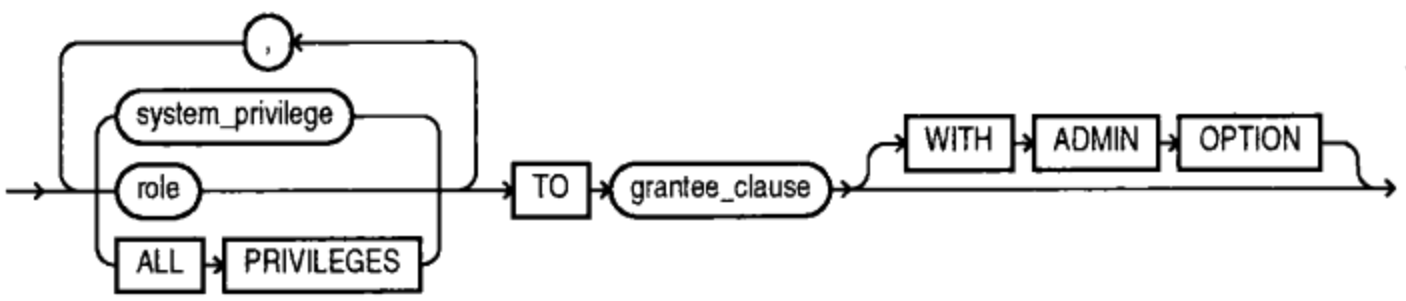
参阅： ALTER USER、CREATE USER、PRIVILEGE、REVOKE、ROLE 和第 19 章。

格式：

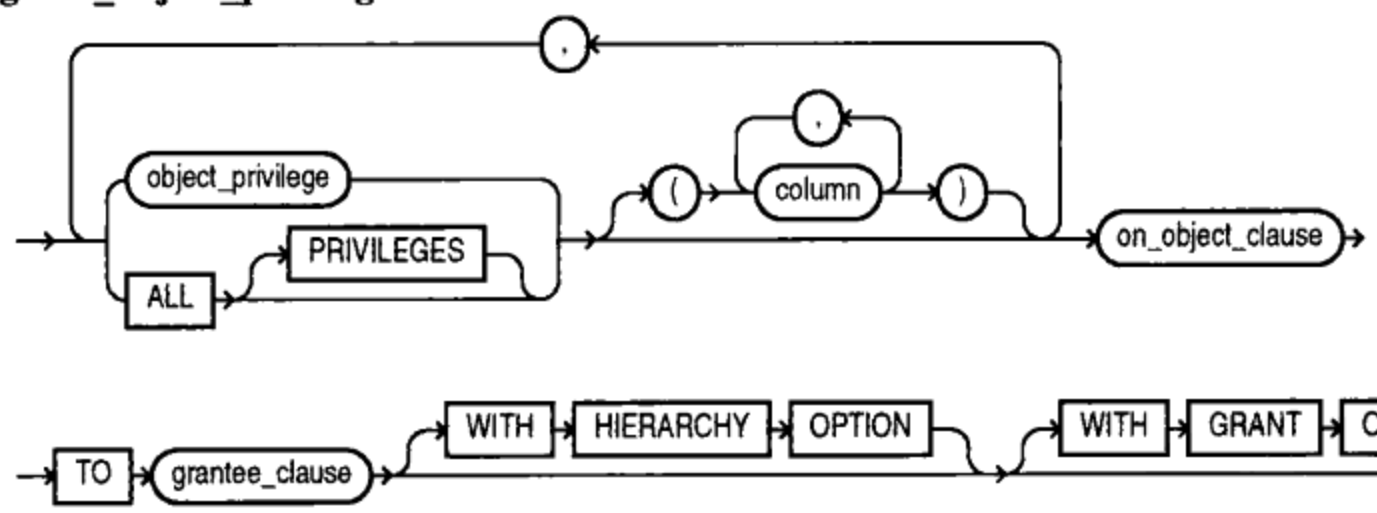
grant::=



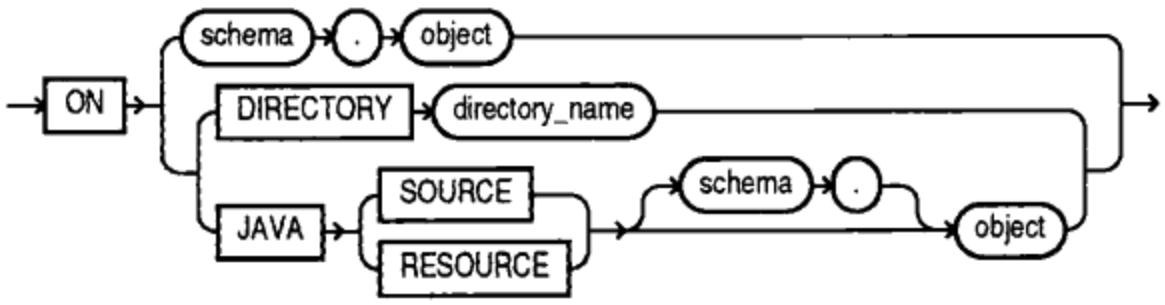
grant_system_privileges::=



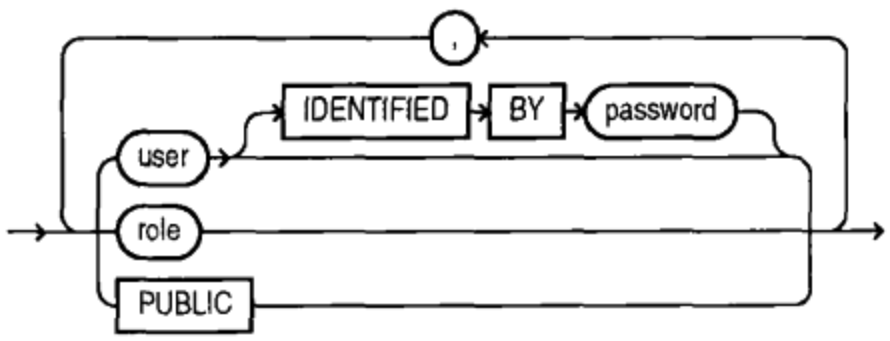
grant_object_privileges::=



on_object_clause::=



grantee_clause::=



描述：GRANT 的第一种格式把一个或多个系统权限扩充给用户和角色。系统权限是执行某种数据定义或控制命令(如 ATLER SYSTEM、CREATE ROLE 或 GRANT 本身)的一个授权。关于这些权限的详细内容请参阅 PRIVILEGE。通过 CREATE USER 创建一个用户，授予它系统权限后才允许登录到数据库和其他操作中。没有授权的用户根本不能在 Oracle 中进行任何操作。role 是由 CREATE ROLE 创建的权限的集合。和用户一样，一个角色在创建时并不拥有任何权限，但是可以通过一个或多个 GRANT 来获得权限。可以把一个角色授予另一个角色以创建角色的嵌套网络，这给数据库管理员管理系统的安全提供了很大的灵活性。

WITH ADMIN OPTION 选项允许被授权的用户或角色(被授予者)给其他的被授予者授予系统权限或角色。被授予者也可以更改或删除用 WITH ADMIN OPTION 所授予的角色。

GRANT 命令的第二种格式将给任何用户或角色授予 object privilege(对象权限)，该权限影响 Oracle 数据库中特定种类的对象。object 可以是表、视图、序列、过程、函数、程序包、物化视图或这些对象的同义词。对同义词的授权(GRANT)实际上是对该同义词所引用的基本对象的授权。关于对象权限的完整列表以及各种对象的授权的讨论，请参阅 PRIVILEGE。

如果正在给某个表或视图授予 INSERT、REFERENCES 或 UPDATE 权限，则 GRANT 可以给应用 GRANT 的表或视图指定一组列。GRANT 只应用于这些列。如果 GRANT 没有指定一组列，则 GRANT 将应用于表或视图中的所有列。

PUBLIC 将权限授予当前和未来的所有用户。

WITH GRANT OPTION 可以将被授予的权限授予另一个用户或角色。

GREATEST

参阅： LEAST、LIST FUNCTIONS、MAX、第 9 章和第 10 章。

格式：

```
GREATEST(value1, value2, ...)
```

描述：GREATEST 选择值列表中的最大值。这些值可以是列、字面量或表达式，以及 CHAR、VARCHAR2、NUMBER 或 DATE 数据类型。具有较大值的数字大于具有较小值的数字。所有负数小于所有正数。因此，－10 小于 10；－100 小于－10。较晚的日期大于较早的日期。

字符串按位置逐个进行比较，从串的最左边开始，直到遇到第一个不相同的字符为止。该位置上具有较大字符的串认为是较大的串。如果一个字符按计算机排序规则出现在另一个字符的后面，则该字符大于另一个字符。通常这意味着 B 大于 A，但是 A 与 a 或与数字 1 的

比较结果将会随计算机的不同而不同。

如果在比较两个串时，一直到较短的那个串的结尾都相等，则较长的那个串是较大的串。如果两个字符串相同且长度相同，则它们是相等的。在 SQL 中，输入字面量数字时不能用单引号引起来，这一点很重要。如‘10’小于‘6’，因为引号使得系统将它们视为字符串而不是数字，所以‘6’将大于‘10’的十位上的 1。

与许多其他的 Oracle 函数和逻辑运算符不同，GREATEST 和 LEAST 函数不能对日期格式的字面量字段串进行判定。为了使 LEAST 和 GREATEST 正常运行，必须对这样的字面量字符串使用 TO_DATE 函数。

GROUP BY

参阅：CUBE、HAVING、ORDER BY、ROLLUP、WHERE 和第 12 章。

格式：

```
SELECT expression [, expression]...
 GROUP BY expression [, expression]...
HAVING condition
    ...
```

描述：GROUP BY 使一个 SELECT 为所有选中的、在一个或多个指定的列或表达式中具有相同值的行生成一个汇总行。SELECT 子句中的每个表达式必须为以下内容之一：

- 一个常量
- 一个没有参数的函数(SysDate、User)
- 一个分组函数，如 SUM、AVG、MIN、MAX 或 COUNT
- 与 GROUP BY 子句中的一个表达式完全匹配

在 GROUP BY 子句中引用的列不必在 SELECT 子句中，尽管它们必须在该表中。

可以使用 HAVING 子句确定 GROUP BY 将要包含的组。另一方面，WHERE 子句确定哪些行将包含在组中。

GROUP BY 和 HAVING 跟在 WHERE、CONNECT BY 和 START WITH 之后。ORDER BY 子句在 WHERE、GROUP BY 和 HAVING 子句(它们按此顺序执行)之后执行。它可以使用分组函数，或者 GROUP BY 子句中的列，或者一个函数与列的组合。如果它使用一个分组函数，该函数将作用于组，则 ORDER BY 将该函数的结果按顺序分类。如果 ORDER BY 子句使用 GROUP BY 子句中的一列，则它把基于该列所返回的行进行分类。分组函数和单个列可以在 ORDER BY 子句中组合使用(只要该列在 GROUP BY 子句中即可)。

在 ORDER BY 子句中，可以指定一个分组函数和它所作用的列，即使它们与 SELECT、GROUP BY 或 HAVING 子句中的分组函数或列完全无关。另一方面，如果在 ORDER BY 子句中指定了不属于分组函数的一列，则它必须在 GROUP BY 子句中。

示例：

```
select Title, COUNT(*)
  from BOOKSHELF_AUTHOR
 group by Title
```

```
having COUNT(*) > 1;
```

GROUP_ID

参阅： AGGREGATE FUNCTIONS、GROUP BY、GROUPING_ID 和第 12 章。

格式：

```
GROUP_ID( )
```

描述： GROUP_ID 识别由 GROUP BY 说明产生的重复组。

GROUPING

GROUPING 是与 ROLLUP 和 CUBE 函数联合使用以检测 NULL 的一个函数。请参阅 ROLLUP。

GROUPING_ID

GROUPING_ID 返回对应于 GROUPING 位矢量(与一行相关联)的一个数值。请参阅 ROLLUP、CUBE 和第 12 章。

HAVING

参阅 GROUP BY。

HELP(SQL*Plus)

格式：

```
HELP | ? [topic]
```

描述： HELP 命令访问 SQL*Plus 帮助系统。HELP INDEX 显示主题的一个列表。

HEXTORAW

参阅： RAWTOHEX。

格式：

```
HEXTORAW(hex_ string)
```

描述： HEXTORAW(HEXadecimal(十六进制)TO RAW)将十六进制数字的字符串变为二进制。

HINT

在一个查询中，可以指定提示，该提示指示查询处理中基于成本的优化程序。要指定提示，可以使用以下语法。在 SELECT 关键字后紧接着输入以下串：

```
/*+
```

接着，添加提示，如

```
FULL(bookshelf)
```

可用下面的串结束该提示：

[redacted]*/

关于可用提示以及它们对查询处理的作用的描述，请参阅第 46 章。

HOST(SQL*Plus)
参阅： $、!、@、@@、START。

格式：

```
        then
          insert into AREAS values (rad_val.radius, area);
      end if;
    close rad_cursor;
end;
```

IN

参阅： ALL、ANY、LOGICAL OPERATORS、WHERE 和第 5 章。

格式：

```
WHERE expression IN ({'string' [,'string']... | select...})
```

描述： IN 等价于“=ANY”。在第一个选项中，IN 说明表达式等于后面的字面量 string 列表中的任何成员。在第二个选项中，它表示表达式等于来自子查询的任何行的任何值。这两个选项在逻辑上是相等的，第一个选项给出了由字面量字符串构成的一个列表，而第二个选项从查询中构建了一个列表。

INDEX

索引是 Oracle/SQL 特性的一个常见术语，主要用来提高执行速度和在特定数据上施加唯一性。一般说来，索引提供了一种比全表扫描更快的访问表数据的方法。对于在表的索引字段中找到的每个值(NULL 值除外)，索引都包含对应的一项，并包含指向该值所在行的指针。

INDEX-ORGANIZED TABLE

索引编排表的数据根据表的主键列值进行分类。索引编排表可以在一个索引中存储整个表的数据。普通的索引只在索引中存储索引列；而索引编排表在索引中存储所有表的列。

要创建索引编排表，可以使用 CREATE TABLE 命令的 ORGANIZATION INDEX 子句，如下例所示：

```
create table TROUBLE (
City           VARCHAR2(13),
SampleDate     DATE,
Noon           NUMBER(4,1),
Midnight       NUMBER(4,1),
Precipitation  NUMBER,
constraint TROUBLE_PK PRIMARY KEY (City, SampleDate))
organization index;
```

要将 TROUBLE 创建为一个索引编排表，必须在其上创建 PRIMARY KEY 约束。

通常，索引编排表适合于多对多关系中相关联的表。理想情况下，索引编排表几乎没有(或没有)不属于主键的列。

请参阅第 17 章和 CREATE TABLE。

INDICATOR VARIABLE

参阅： “:” (冒号，宿主变量的前缀)、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
:name [INDICATOR] :indicator
```

描述：指示器变量(在名字和指示器都是宿主语言变量名时使用)是一个数据字段，其值指明宿主变量是否应看作 NULL(使用 INDICATOR 关键字是为了增强可读性，没有别的作用)。

name 可以是任意合法的宿主变量数据名，它可以在宿主程序中使用并包含在预编译程序的 DECLARE 部分中。indicator 在预编译程序的 DECLARE 部分中定义为两个字节的整数。

几乎没有过程语言直接支持带有 NULL 或未知值的变量，尽管 Oracle 和 SQL 很容易做到。为了扩展 Oracle 已经开发了预编译程序的语言来支持 NULL 概念，可以将指示器变量与一个宿主变量相联系(这很像是一个标志)以便指明它是否为 NULL。虽然宿主变量及其指示器变量可以在宿主语言代码中分别引用和设置，但在 SQL 或 PL/SQL 中总是连接在一起。

示例：

```
BEGIN
    select Title, AuthorName into :Title, :AuthorName:AuthorNameInd
      from BOOKSHELF_AUTHOR;
  IF :AuthorName:AuthorNameInd IS NULL
    THEN :AuthorName:AuthorNameInd := 'No Auth'
  END IF;
END;
```

请注意，对变量是否为 NULL 的测试，以及(如果是的话)该变量的赋值是否为“No Auth”都是使用连接的变量名完成的。PL/SQL 知道检查第一种情形中的指示器，并在第二种情形中设置变量值。这两个变量一起处理，就像一个 Oracle 列一样。需要注意以下几个要点：

- 在单个 PL/SQL 块中，宿主变量必须始终保持独立或者始终与其指示器变量连接。
- 指示器变量不能在 PL/SQL 中单独引用，尽管它在宿主程序中可以。

当在宿主程序中(但在 PL/SQL 外)设置宿主变量时，要注意以下几点：

- 指示器变量设置为 -1 将迫使连接的变量在 PL/SQL 中被视为 NULL，在逻辑测试和 INSERT 或 UPDATE 中都是如此。
- 设置指示器变量大于或等于 0 将迫使连接的变量被视为等于宿主变量的值，并为 NOT NULL。
- PL/SQL 测试输入某块中的所有指示器变量的值，并在已有块中设置它们。

当在 PL/SQL 中通过带有 INTO 子句的 SQL 语句加载连接的宿主变量时，以下规则适用于检查宿主程序内(但在 PL/SQL 之外)的指示器变量：

- 如果从数据库中加装的值为 NULL，则指示器变量将等于 -1。宿主变量的值不确定，也应该这样处理。
- 如果数据库中的值完全地和正确地加载到了宿主变量中，则指示器变量等于 0。
- 如果只有一部分数据能够放入宿主变量中，则指示器变量将大于 0，且等于数据库列中数据的实际长度。宿主变量将包含该数据库列的一部分数据。

init.ora

init.ora 是一个数据库系统参数文件，它包含了使用 CREATE DATABASE、STARTUP 或 SHUTDOWN 命令启动一个系统时使用的设置和文件名。可以使用 init.ora 文件，或者通过 CREATE SPFILE 命令创建一个系统参数文件。请参阅 CREATE PFILE 和 CREATE SPFILE。

INITCAP

参阅： CHARACTER FUNCTIONS、LOWER、UPPER、第 7 章。

格式：

```
INITCAP(string)
```

描述： INITCAP(INITial CAPital)将一个单词或一系列单词的第一个字母改为大写。使用该命令时还要注意符号的存在，该命令将 INITCAP 空格或符号(如逗号、句号、冒号、分号、感叹号、@、#、$等)后面的任何字母。

示例：

```
INITCAP('this.is,an-example of!how@initcap#works')
```

将得到以下结果：

```
This.Is,An-Example Of!How@Initcap#Works
```

INPUT(SQL*Plus)

参阅： APPEND、CHANGE、DEL、EDIT、第 6 章。

格式：

```
I[NPUT] [text]
```

描述： INPUT 在当前缓冲区的当前行后面添加一行新文本。使用 INPUT 允许在当前行后面输入多行，并且系统在没有任何内容的行上遇到 ENTER 时终止。INPUT 和 text 之间的空格不添加到文本行中，但是其他空格将添加进去。关于当前行的讨论请参阅 DEL。INPUT 在 iSQL*Plus 中不可用。

INSERT(格式 1——嵌入式 SQL)

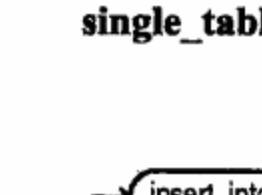
参阅： EXECUTE、CURSOR FOR LOOP、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [AT { db_name | :host_variable }]
  [FOR { :host_integer | integer }]
  INSERT INTO
  [ [schema.] { table | view }
    [ @db_link | PARTITION (part_name) ] ]
  [(column) [, (column)]...]
{ VALUES (expr [, expr]...) | subquery }
```

```
[{ RETURN | RETURNING } expr [, expr]... INTO
:host_variable [[INDICATOR] :ind_variable]
[, :host_variable [[INDICATOR] :ind_variable]]...]
```

描述：db_name 是用户的默认数据库以外的数据库，host_variable 包含该数据库的名字。":host_integer"是限制处理 INSERT 的次数的宿主值(请参阅 CURSOR FOR LOOP)。table 是已有的任何表、视图或同义词，db_link 是存储表的远程数据库的名字(关于 VALUES 子句、列和查询的讨论，请参阅 INSERT 格式 3 的定义)。expression 在这里可以是":variable[:indicator]"格式的表达式或宿主变量。



注意：
关于所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。该参考书不包含所有可能的预编译程序命令。

INSERT(格式 2——PL/SQL)

参阅：SQL CURSOR、第 32 章。

格式：

```
INSERT INTO [user.]table[@db_link] [(column [,column]...)]
    VALUES (expression [,expression]...) | query...);
```

描述：PL/SQL 的 INSERT 的用法与其通常的格式(下面的格式 3)是相同的，但是有以下例外：

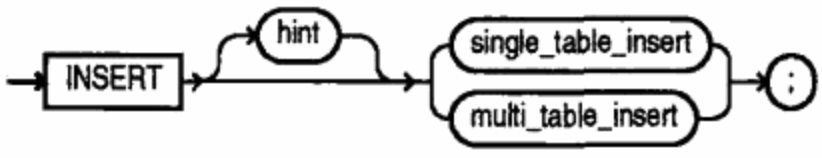
- 对于 VALUES 列表中的值，可以在表达式中使用 PL/SQL 变量。
- 每个变量将与处理常量(以及变量的值)相同的方法进行处理。
- 如果使用 INSERT 的查询版本，则不能使用 SELECT 的 INTO 子句。

INSERT(格式 3——SQL 命令)

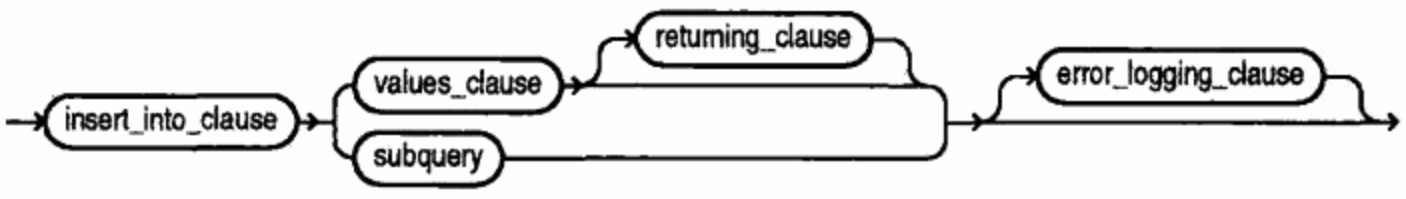
参阅：CREATE TABLE、第 5 章、第 15 章、第 38 章和第 39 章。

格式：

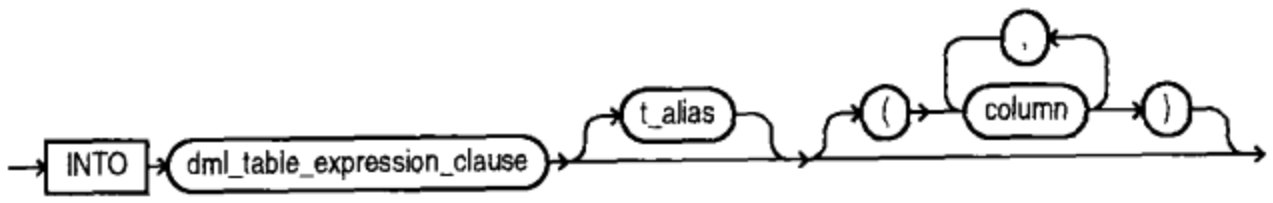
insert::=



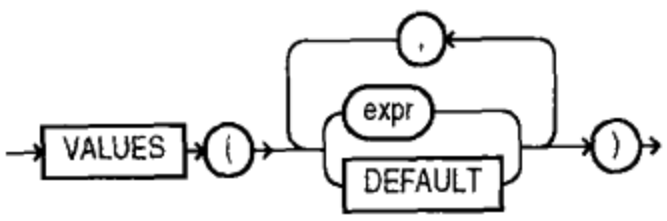
single_table_insert::=



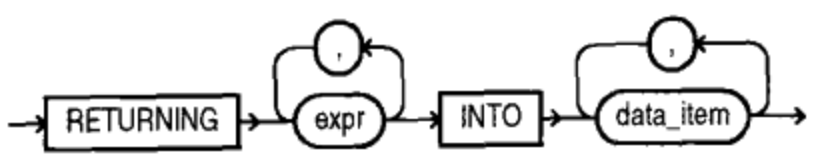
insert_into_clause::=



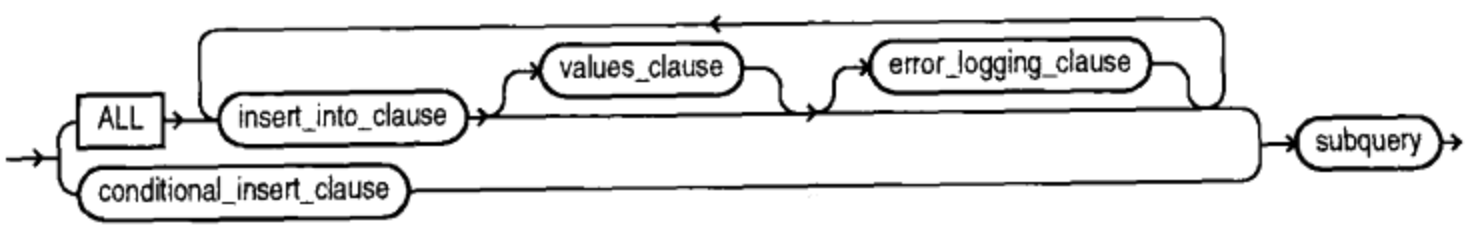
values_clause::=



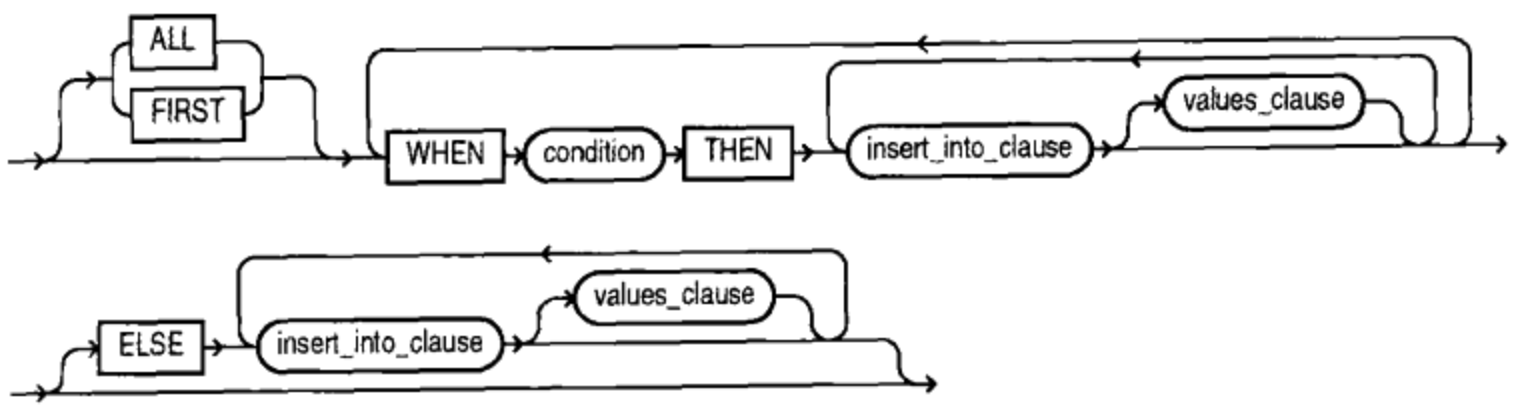
returning_clause::=



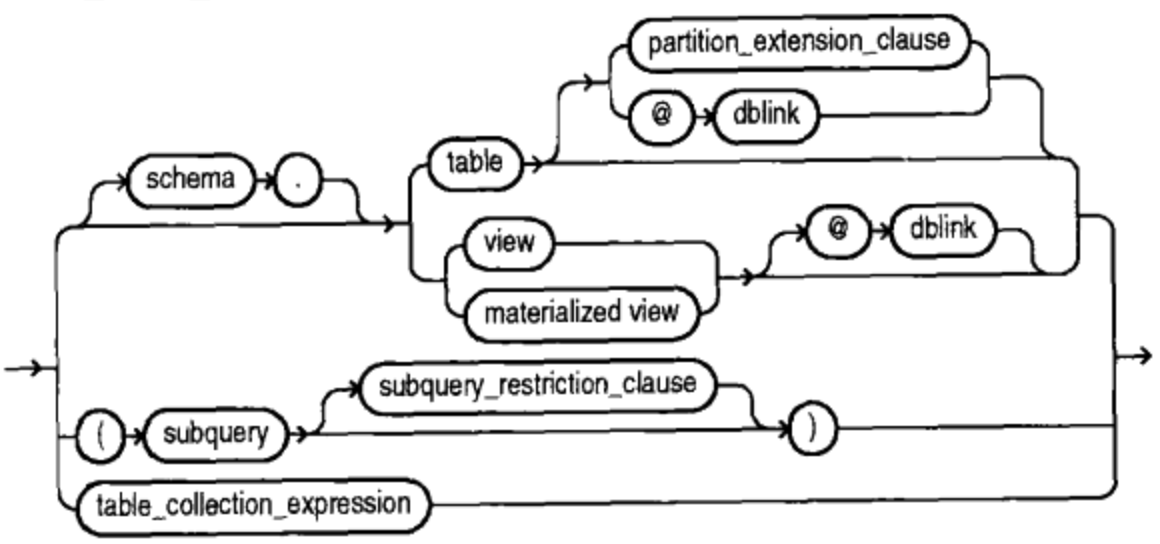
multi_table_insert::=



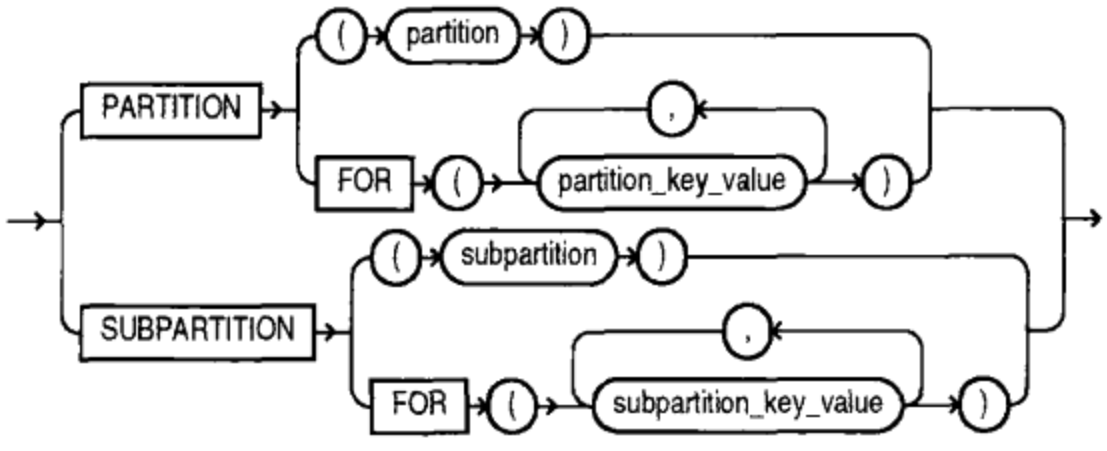
conditional_insert_clause::=



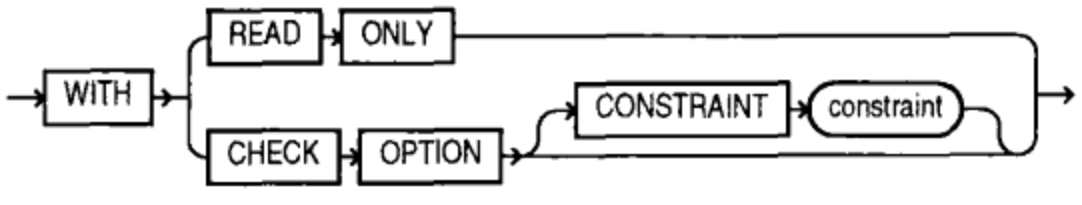
DML_table_expression_clause::=



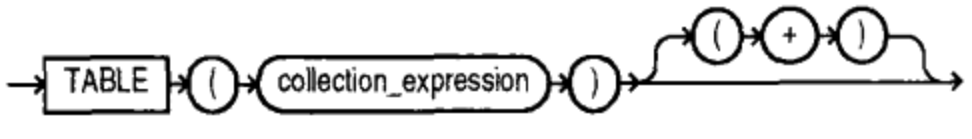
partition_extension_clause::=



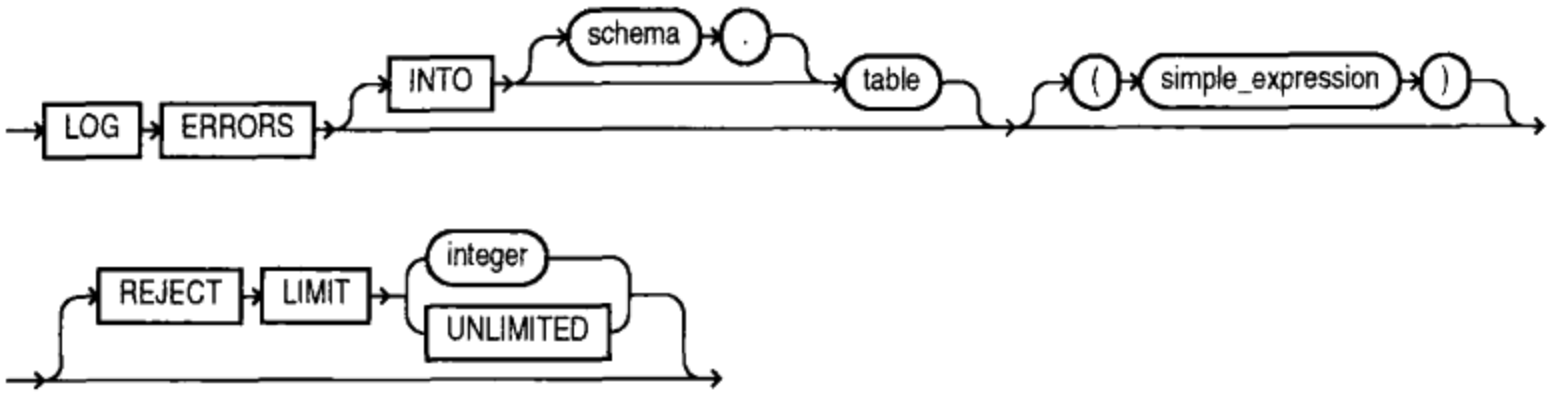
subquery_restriction_clause::=



table_collection_expression::=



error_logging_clause::=



描述：INSERT 将一个或多个新行添加到表或视图中。可选的 schema 必须是一个用户，并且对它的表具有插入权限。table 是将要插入数据行的表。如果给定一组列，则每一列必须与一个表达式(SQL 表达式)匹配。不在此列表中的列将得到 NULL 值，并且都不能定义为 NOT NULL 否则 INSERT 将失败。如果没有给定一组列，则必须给出表中所有列的值。

带有子查询的 INSERT 可以根据该子查询返回的行数添加多行，每个查询列与列列表中的列必须在位置上匹配。如果不指定列列表，则表必须有相同的列数量和列类型。USER_TAB_COLUMNS 数据字典视图将显示这些列，就像在 SQL*Plus 中所做的 DESCRIBE 操作一样。

示例：以下程序将一行插入 COMFORT 表中：

```
insert into COMFORT values ('KEENE','23-SEP-03',99.8,82.6,NULL);
```

下面将 City 列、SampleDate 列(两者都是 NOT NULL 列)和 Precipitation 列插入 COMFORT

表中：

```
insert into COMFORT (City, SampleDate, Precipitation) values
                    ('KEENE','22-DEC-03',3.9);
```

为了将城市 KEENE 的数据复制到名为 NEW_HAMPSHIRE 的新表中，可使用以下程序：

```
insert into NEW_HAMPSHIRE
select * from COMFORT
 where City = 'KEENE';
```

INSERTCHILDXML

格式：

```
INSERTCHILDXML
    ( XMLType_instance, XPath_string, child_expr,
        value_expr [, namespace_string ] )
```

描述：INSERTCHILDXML 将用户提供的值插入 XPath 表达式指定的节点的目标 XML 中。

INSERTXMLBEFORE

格式：

```
INSERTXMLBEFORE
    ( XMLType_instance, XPath_string,
       value_expr [, namespace_string ] )
```

描述：INSERTXMLBEFORE 将用户提供的值插入 XPath 表达式指定的节点之前的目标 XML 中。

INSTR

参阅：CHARACTER FUNCTIONS、SUBSTR、第 7 章。

格式：

```
INSTR(string, set[, start[, occurrence]])
```

描述：INSTR 在一个 string 中查找一组(set)字符的位置，从该串的 start 位置开始查找 set 的第一次出现、第二次出现、第三次出现，等等，直到查找到第 occurrence 次出现为止。该函数还可作用于 NUMBER 和 DATE 数据类型。start 也可以是负数，表示从串的末尾开始向前搜索。

示例：为了在串中找到“PI”第三次出现的位置，可以使用以下语句：

```
INSTR('PETER PIPER PICKED A PECK OF PICKLED PEPPERS','PI',1,3)
```

此函数的结果为 30，即 PI 第三次出现的位置。

INTEGRITY CONSTRAINT

完整性约束是限制一列的有效值范围的规则。它是创建表时添加在列上的。关于其语法，

请参阅 CONSTRAINTS。

如果没有命名一个约束，则 Oracle 将以 SYS_C*n* 格式分配一个名字，这里的 *n* 为一个整数。Oracle 分配的名字通常在导入过程中更改，而用户分配的名字则不会改变。

NULL 允许使用 NULL 值。NOT NULL 指定每行中的此列必须有非 NULL 值。

UNIQUE 使列值唯一。在一个表上只能有一个 PRIMARY KEY 约束。如果一列为 UNIQUE，则它不能声明为 PRIMARY KEY(PRIMARY KEY 也强制使用唯一性)。索引强制使用 UNIQUE 或 PRIMARY KEY，且 USING INDEX 子句及其选项指定索引的存储特征。关于这些选项的更多信息请参阅 CREATE INDEX。

REFERENCES 以外键形式[user.]table[(column)]标识列。如果省略 column，则表示 user.table 中的名字与表中的名字相同。请注意，当在 table_constraint(简略描述)中使用 REFERENCES 时，它的前面必须有 FOREIGN KEY。这里没有使用它是因为只有此列被引用。对于 FOREIGN KEY 来说，table_constraint 可以引用几列。如果删除了表的主键行的话，则 ON DELETE CASCADE 指示 Oracle 通过删除所依赖表的外键行来自动维护参照完整性。

CHECK 假定列的值符合以下条件：

```
Amount number(12,2) CHECK (Amount >= 0)
```

condition 可以是测试 TRUE 或 FALSE 的任何有效的表达式。它可以包含函数、表中的任何列以及字面量。

EXCEPTION INTO 子句指定一个表，此表用来放置有关违反已启用的完整性约束的行的信息。此表必须为本地的。

DISABLE 选项可以在创建完整性约束时禁用它。当约束被禁用时，Oracle 不能自动启用它。可以在 ALTER TABLE 中用 ENABLE 子句启用约束。

还可以创建表级约束。除单个约束可以引用多列以外(如 3 列同时声明为主键或外键)，表约束与列约束一样。

INTERSECT

参阅：MINUS、QUERY OPERATORS、UNION、第 13 章。

格式：

```
select...
INTERSECT
select...
```

描述：INTERSECT 将两个查询组合在一起，并且只返回第一个 SELECT 语句中至少与第二个 SELECT 语句中的一行相同的那些行。在这两个 SELECT 语句之间，虽然列的名字不需要相同但列的数目和数据类型必须一致。然而，INTERSECT 所产生的行的数据必须是相同的。

INTERVAL DAY TO SECOND

INTERVAL DAY(day_precision)TO SECOND(second_precision)数据类型存储以日、小时、分钟和秒为单位的时间段。可以指定 day_precision 为 DAY 日期时间字段中的数字位数(默认

为 2)，指定 second_precision 为 Seconds 字段的小数部分的数字位数(默认为 6)。

INTERVAL YEAR TO MONTH

INTERVAL YEAR(precision)TO MONTH 数据类型以年和月为单位来存储时间段，其中 precision 是 YEAR 日期时间字段中的数字位数(默认为 2)。

IS A SET

参阅： IS EMPTY、第 39 章。

格式：

```
nested_table IS [NOT] A SET
```

描述： 利用 IS A SET 条件来测试指定的嵌套表是否由唯一元素组成。如果嵌套表是一个集合，则即使其长度为零，该条件也返回 TRUE；否则返回 FALSE。如果嵌套表为空，则返回 NULL。

IS ANY

参阅： SELECT。

格式：

```
dimension column IS ANY
```

描述： IN ANY 子句只与行间计算有关，并且只能用于 SELECT 语句的 MODEL 子句中。利用该条件来限制一个维度列的所有值，包括 NULL 在内。该条件总是返回一个布尔值 TRUE 来限制列的所有值。

IS EMPTY

参阅： IS A SET、第 39 章。

格式：

```
nested table IS [NOT] EMPTY
```

描述： 利用 IS [NOT] EMPTY 条件来测试指定的嵌套表是否为空，而不管集合中是否有元素为 NULL。该条件返回一个布尔值：如果条件为空，则对 IS EMPTY 条件返回 TRUE；如果条件非空，则对 IS NOT EMPTY 条件返回 TRUE。如果指定嵌套表或数组为 NULL，则结果为 NULL。

IS NULL

参阅： LOGICAL OPERATORS、第 5 章和第 9 章。

格式：

```
WHERE column IS [NOT] NULL
```

描述： IS NULL 测试没有任何数据的列(或表达式)。NULL 的测试明显不同于相等性测

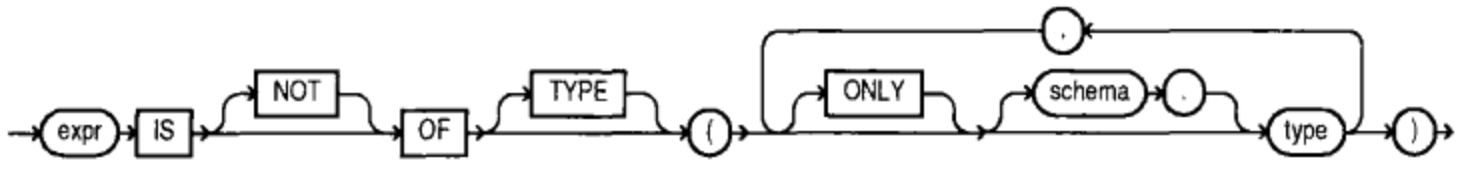
试，由于 NULL 表示值是未知的或不相关的，因此不能说它等于任何值，即使是另一个 NULL。

IS OF type

参阅： VALUE、第 38 章。

格式：

is of type conditions::=



描述： 利用 IS OF type 条件来测试基于其特定类型信息的对象实例。如果 expr 为 NULL，则该条件判定为 NULL。如 expr 非空，则该条件在以下任意一种情况下判定为 TRUE(如果指定 NOT 关键字，则判定为 FALSE)：

- expr 最特殊的类型是 type 列表中指定的某种类型的子类型，并且没有为该类型指定 ONLY。
- expr 最特殊的类型在 type 列表中显式指定。

expr 通常接受带相关变量的 VALUE 函数的格式。

IS PRESENT

参阅： SELECT。

格式：

```
cell_reference IS PRESENT
```

描述： IS PRESENT 条件只与行间计算有关，并且只能用于 SELECT 语句的 MODEL 子句中。利用该条件来测试引用的单元格是否出现在 MODEL 子句的执行之前。如果单元格存在于 MODEL 子句的执行之前，则该条件返回 TRUE；否则返回 FALSE。

ITERATION_NUMBER

参阅： SELECT。

格式：

```
ITERATION_NUMBER
```

描述： ITERATION_NUMBER 函数只与行间计算有关。它只能用在 SELECT 语句的 MODEL 子句中，并且只有在 MODEL RULES 子句中指定了 ITERATE(number)时才可以使用。它返回一个整数，表示模式规则中已经完成的迭代。第一次迭代时，ITERATION_NUMBER 函数返回 0。对接下来的各个迭代，ITERATION_NUMBER 函数的返回值每次加 1。

JOIN

参阅： SELECT、第 5 章。

格式：

```
WHERE TableA.column = TableB.column
```

或

```
FROM TableA INNER JOIN TableB
  on TableA.Column = TableB.Column
```

或

```
FROM TableA INNER JOIN TableB
using(Column)
```

或

```
FROM TableA NATURAL JOIN TableB
```

描述：连接用来组合两个或多个表中的列和数据(特殊情况下，也可以是一个表与其自身的组合)。所有的表都列在 SELECT 语句的 FROM 子句中，两个表之间的关系在 WHERE 子句中指定，通常为简单的等式，如下所示：

```
where BOOK_ORDER.Title = BOOKSHELF.Title
```

这通常称为等值连接(equi-join)，因为它在 WHERE 子句中使用了等号。虽然也可以使用其他形式的等式来连接表，如“\>=”、“\<”等等，但是其结果通常没有意义。还有另一种情况，即等号的一端或两端包含表达式，可能是 SUBSTR 或列的组合，这是很常见的用法。

将两个表连接在一起而不使用连接子句(既不在 FROM 子句中使用，也不在 WHERE 子句中使用)会产生一个笛卡尔积，它是一个表的每一行与另一个表的每一行的组合。80 行的表与 100 行的表的组合将产生 8 000 行(这通常无意义)。

外部连接是一种有意识地从一个表中检索选中的但与另一个表中的行不相匹配的行的方法。可使用 ANSI 标准的外部连接语法选项(CROSS JOIN、NATURAL JOIN、LEFT OUTER JOIN、RIGHT OUTER JOIN 和 FULL OURTER JOIN)。

LABEL(PL/SQL)

标签是与可执行语句相关联的一个单词，通常用于标识 GOTO 语句的目标。

LAG

参阅：LEAD。

格式：

```
LAG ( value_expr [, offset] [, default] )
    OVER ( [query_partition_clause] order_by_clause )
```

描述：LAG 可以同时访问一个表的多行而不用自连接。给定某个查询返回的一系列行和一个游标的位置，LAG 可以访问位于该位置前的一个给定物理偏移量处的行。

LAST

参阅：DENSE_RANK、FIRST。

格式：

```
aggregate_function KEEP
( DENSE_RANK LAST ORDER BY
  expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [ NULLS { FIRST | LAST }]]...
) [OVER query_partition_clause]
```

描述：LAST 既是一个聚集函数也是一个分析函数，它作用于来自一组行的一组值，这组行按照给定的分类说明排在最后。aggregate_function 是一个标准的数值分组函数(如 MIN、MAX、AVG 和 STDDEV)。

LAST_DAY

参阅：ADD_MONTHS、DATE FUNCTIONS、NEXT_DAY、第 10 章。

格式：

```
LAST_DAY (date)
```

描述：LAST_DAY 给出 date 所在月的最后一天的日期。

示例：

```
LAST_DAY ('05-NOV-04')
```

得到的结果为 30-NOV-04。

LAST_VALUE

参阅：FIRST_VALUE。

格式：

```
LAST_VALUE ( expr [IGNORE NULLS]) OVER ( analytic_clause )
```

描述：LAST_VALUE 是一个分析函数。它返回一组有序值中的最后一个值。如果这组值中的最后一个值为空，则该函数返回 NULL，除非指定 IGNORE NULLS。

LEAD

参阅：LAG。

格式：

```
LEAD ( value_expr [, offset] [, default] )
   OVER ( [query_partition_clause] order_by_clause )
```

描述：LEAD 可以同时访问一个表的多行而不用自连接。给定某个查询返回的一系列行和一个游标的位置，LEAD 可以访问位于该位置前的一个给定物理偏移量处的行。

LEAST

参阅： GREATEST、LIST FUNCTIONS、第 10 章。

格式：

```
LEAST(value1, value2, ...)
```

描述： LEAST 是一组列、表达式或值中的最小值。这些值可以是 VARCHAR2、CHAR、DATE 或 NUMBER 数据类型，尽管如果没有 TO_DATE 函数，LEAST 就不能正确估算字面量日期(如“20-MAY-04”)。关于判定相关值的讨论请参阅 GREATEST。

LENGTH

参阅： CHARACTER FUNCTIONS、VSIZE、第 7 章。

格式：

```
LENGTH(string)
```

描述： LENGTH 给出串、数值、日期或表达式的长度。string 可以是 CHAR(返回的长度将包含所有尾端空格)、VARCHAR2、NCHAR、NVARCHAR2、CLOB 或 NCLOB 类型。有以下几个 LENGTH 函数，它们都具有相同的格式：

- LENGTH 将字符作为输入字符集定义的字符来计算长度。
- LENGTHB 用字节来代替字符。
- LENGTHC 使用 Unicode 完全字符。
- LENGTH2 使用 UCS2 码点。
- LENGTH4 使用 UCS4 码点。

LEVEL

参阅： CONNECT BY、PSEUDO-COLUMNS、第 14 章。

格式：

```
LEVEL
```

描述： 层(level)是一个伪列，与 CONNECT BY 一起使用，根节点等于 1，根节点的子节点等于 2，根节点的子节点的子节点等于 3，依此类推。层主要是说明用户已经遍历到树的哪一层。

LIKE

参阅： LOGICAL OPERATORS、第 5 章。

格式：

```
WHERE string LIKE string
```

描述： LIKE 执行模式匹配。一条下划线表示一个空格。一个百分号表示任意数目(包括 0 在内)的空格或字符。如果 LIKE 在比较的第一个位置使用“_”或“%”(如下面的第二个

和第三个示例所示)，则列上的任何索引都被忽略。

示例：

```
Feature LIKE 'MO%' "Feature begins with the letters MO."
Feature LIKE '_ _I%' "Feature has an I in the third position."
Feature LIKE '%0%0%' "Feature has at least two 0's in it."
```

LIST(SQL*Plus)

参阅：APPEND、CHANGE、EDIT、INPUT、RUN、第 6 章。

格式：

```
L[IST] [ n |n m|n*|n LAST|*|*n|*LAST|LAST]
```

描述：LIST 列出当前缓冲区中从 n 开始到 m 结束的行，n 和 m 都是整数。最后一行成为缓冲区的当前行，并用星号标记。如果没有 n 和 m，则 LIST 将列出所有行。只有一个数字的 LIST 将只显示那一行。LIST 和 n 之间的空格不是必需的，但是有助于增加可读性。LIST n*列出从 n 开始到缓冲区结束的所有行。LIST *n 列出从开始到指定行的所有行。

示例：利用该例列出当前 SQL 缓冲区的内容：

```
LIST
```

星号将指出当前行。要列出第二行，可用以下代码：

```
LIST 2
```

这条语句使第 2 行成为当前缓冲区的当前行。

LIST FUNCTIONS

参阅：所有其他函数、第 10 章。

描述：以下是 SQL 的 Oracle 版本中的所有当前列表函数按字母顺序排列的列表。每个函数都会在本附录的其他地方以各自的名字列出，并给出正确的格式和用法。

下例给出集合中的第一个非 NULL 值：

```
COALESCE(value1, value2, ...)
```

下例给出一个列表中的最大值：

```
GREATEST(value1, value2, ...)
```

下例给出一个列表中的最小值：

```
LEAST(value1, value2, ...)
```

LN

参阅：NUMBER FUNCTIONS、第 9 章。

格式：

```
LN(number)
```

描述：LN 是 number 的自然对数，即以 e 为底的对数。如果 number 是 BINARY_FLOAT，则该函数返回 BINARY_DOUBLE。

LNNVL

参阅：SELECT。

格式：

```
LNNVL ( condition )
```

描述：当条件的一个或两个操作数可能为空时，LNNVL 提供了一种判定条件的方式。该函数只能用于查询的 WHERE 子句中。它接受条件的一个参数，并且如果条件为 FALSE 或 UNKNOWN，它就返回 TRUE；如果条件为 TRUE，则返回 FALSE。LNNVL 能够用于任何出现标量表达式的地方，即使是在 IS [NOT] NULL、AND 或 OR 条件无效但是需要用于解释潜在的空值的上下文中。当 Oracle 内部需要把 NOT IN 条件作为 NOT EXISTS 条件时，LNNVL 可能出现在说明计划中。

示例：如果想搜索某一列大于给定值的所有行，则标准格式为

```
where column > value
```

如果也要返回 column 为 NULL 的行，则使用 LNNVL。

```
where LNNVL( column > value)
```

LOB

LOB 是一个大对象。Oracle 支持几种大对象数据类型，包括 BLOB(二进制大对象)、CLOB(字符大对象)和 BFILE(二进制文件，存储在数据库外部)。请参阅第 40 章和 CREATE TABLE 的 LOB 子句。

LOCAL TIMESTAMP

参阅：DATE FUNCTIONS、第 10 章。

格式：

```
LOCALTIMESTAMP [( timestamp_precision )]
```

描述：LOCALTIMESTAMP 返回会话时区的当前日期和时间作为数据类型 TIMESTAMP 的一个值。

示例：

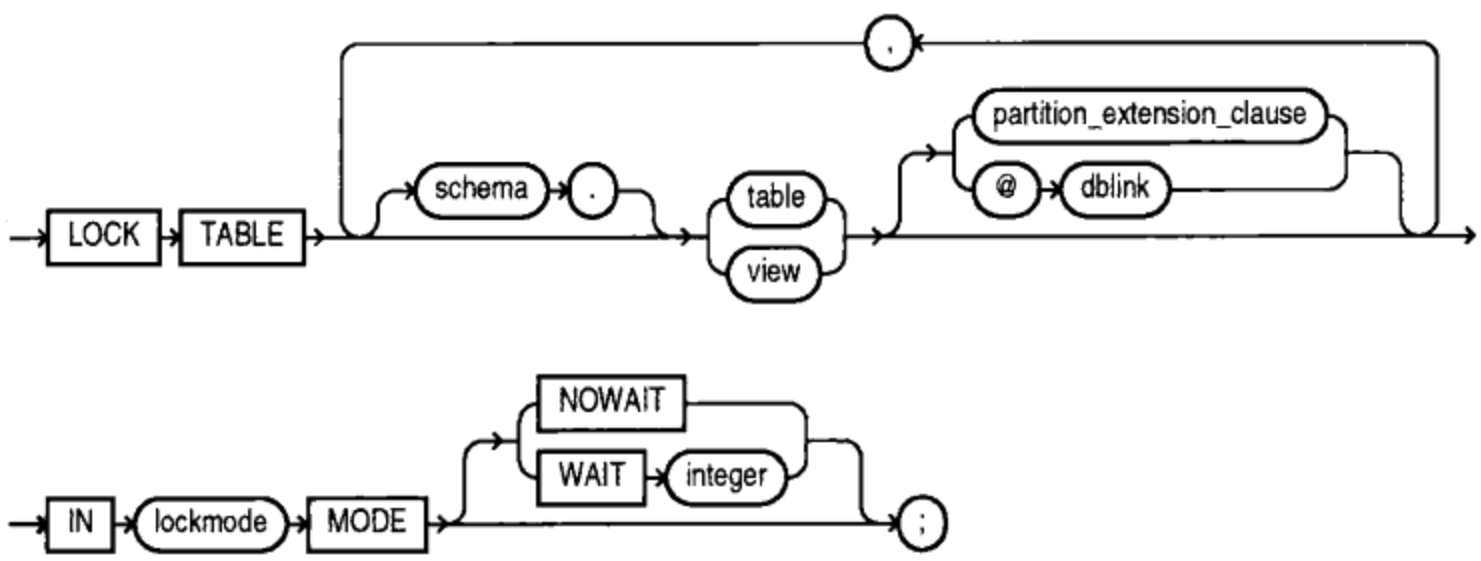
```
SELECT LOCALTIMESTAMP FROM DUAL;
```

LOCK TABLE

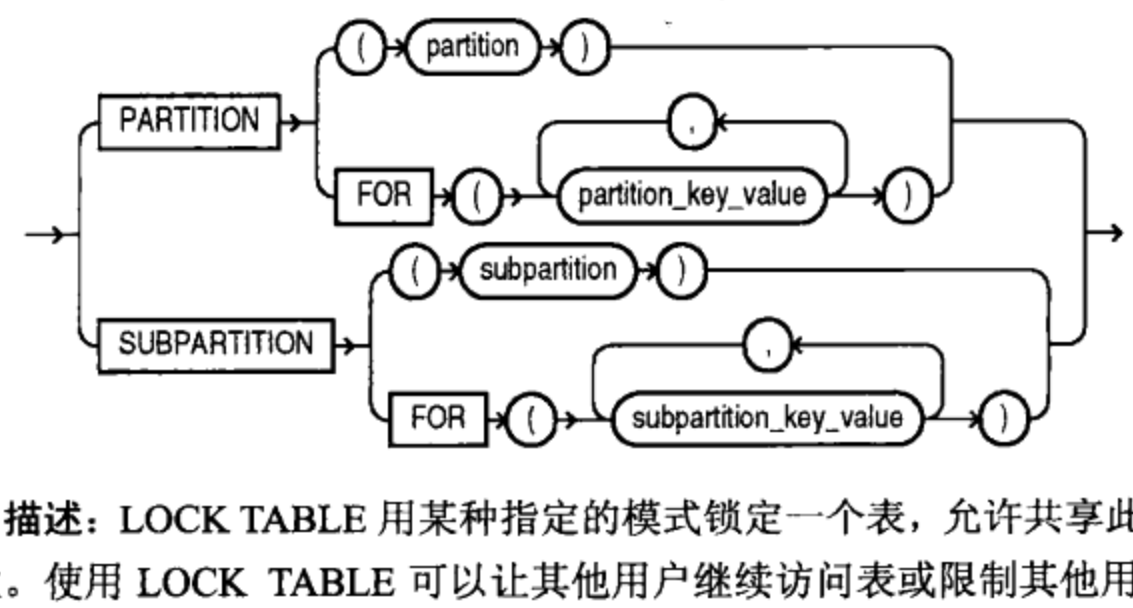
参阅：COMMIT、DELETE、INSERT、ROLLBACK、SAVEPOINT 和 UPDATE。

格式：

lock_table::=



partition_extension_clause::=



描述：LOCK TABLE 用某种指定的模式锁定一个表，允许共享此表，但是不破坏数据完整性。使用 LOCK TABLE 可以让其他用户继续访问表或限制其他用户对表的访问。无论选择哪个选项，表都仍然处于锁定模式，直到 COMMIT 或 ROLLBACK 事务为止。

锁的模式包括 ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE、SHARE、SHARE ROW EXCLUSIVE 和 EXCLUSIVE。

EXCLUSIVE 锁只允许用户查询锁定的表但是不能执行其他操作。其他用户不能锁定该表。SHARE 锁允许并行查询但不允许更新锁定的表。

如果使用 ROW SHARE 或 SHARE UPDATE 锁，则任何用户都不能为了拥有排他访问权而锁定整个表，它们允许所有用户同时访问此表。这两类锁是同义的，SHARE UPDATE 是为兼容 Oracle 以前的版本而保留下来的。

ROW EXCLUSIVE 锁类似于 ROW SHARE，但是它禁止使用共享锁，因此每次只有一个用户可以访问此表。

如果不能完成 LOCK TABLE 命令(通常由于其他人已经执行了以前存在竞争的某种类型的 LOCK TABLE 操作)，则 LOCK TABLE 将一直等待，直到它可以完成为止。如果希望避免这种情况，并简单地把控制权返回给用户，可以使用 NOWAIT 选项。请注意，可以锁定特定的分区和子分区。

在 Oracle Database 11g 中，可以指定为获得表上的 DML 锁，语句应该等待的最长秒数。

LOG

参阅： NUMBER FUNCTIONS、第 9 章。

格式：

```
LOG(base, number)
```

描述： LOG 给出 number 以指定的 base 为底的对数。

示例：

```
LOG(EXP(1),3) = 1.098612   -- log(e) of 3
LOG(10,100) = 2            -- log(10) of 100
```

LOGICAL OPERATORS

参阅： PRECEDENCE。

格式： 以下列表列出了 SQL 的 Oracle 版本中的所有当前逻辑运算符。其中大多数运算符都在本附录的其他地方按照各自的名字列出，并给出正确的格式和用法。所有运算符都作用于列或字面量。

测试单个值的逻辑运算符

```
=           expression      等于 expression
>           expression      大于 expression
>=          expression      大于等于 expression
<           expression      小于 expression
<=          expression      小于等于 expression
!=          expression      不等于 expression
^=          expression      不等于 expression
<>          expression      不等于 expression

EXISTS (query)
NOT EXISTS (query)

LIKE expression
NOT LIKE expression

expression IS NULL
expression IS NOT NULL
```

测试多个值的逻辑运算符

```
ANY (expression [,expression]... | query)
ALL (expression [,expression]... | query)
```

ANY 和 ALL 需要一个等式运算符作为前缀，如“\>ANY”、“=ALL”等。

```
IN (expression [,expression]... | query)
NOT IN (expression [,expression]... | query)

BETWEEN expression AND expression
NOT BETWEEN expression AND expression
```

其他逻辑运算符

```
+           加
-           减
*           乘
/           除
**          幂
||          连接
()          重写通常的优先规则，或将子查询括起来
NOT         反转逻辑表达式
AND         组合逻辑表达式
OR          组合逻辑表达式
UNION       组合查询结果
UNION ALL   组合查询结果，包含重复值
INTERSECT   组合查询结果
MINUS       组合查询结果
```

LONG DATATYPE

请参阅 DATATYPES。

LONG RAW DATATYPE

LONG RAW 列包含原始二进制数据，其他方面与 LONG 列相同。输入到 LONG RAW 列中的值必须用十六进制表示法。

LOOP

可以在一个 PL/SQL 块中使用循环来处理多条记录。PL/SQL 支持三种类型的循环，如表 A-16 所示。

表 A-16　PL/SQL 支持的 3 种循环

循 环 类 型	描　　述
简单循环	重复执行直到在该循环中遇到 EXIT 或 EXIT WHEN 语句为止的循环
FOR 循环	重复指定次数的循环
WHILE 循环	重复执行直到满足条件为止的循环

可以使用循环来处理游标中的多条记录。最常见的游标循环是游标 FOR 循环。关于简单的和基于游标处理的逻辑使用循环的详细内容请参阅 CURSOR FOR LOOP 和第 32 章。

LOWER

参阅：CHARACTER FUNCTIONS、INITCAP、UPPER、第 7 章。

格式：

```
LOWER(string)
```

描述：LOWER 将串中的每个字母转换成小写。

示例：

```
LOWER('PeninSula') = peninsula
```

LPAD

参阅： CHARACTER FUNCTIONS、LTRIM、RPAD、RTRIM、第 7 章。

格式：

```
LPAD(string, length [,'set'])
```

描述： Left PAD 通过将一组(set)字符添加到串的左边使 string 达到指定的长度(length)。如果没有指定 set，则默认的填充字符为空格。

示例：

```
LPAD('>',11,'-')
```

将得到

```
- - - - - >
```

LTRIM

参阅： CHARACTER FUNCTIONS、LPAD、RPAD、RTRIM、第 7 章。

格式：

```
LTRIM(string [,'set'])
```

描述： Left TRIM 去掉所有在 string 左边出现的一组(set)字符中的任何一个字符。

示例：

```
LTRIM('NANCY','AN')
```

将得到

```
CY
```

MAKE_REF

参阅： 第 41 章。

格式：

```
MAKE_REF ( table | view , key [, key] )
```

描述： MAKE_REF 函数利用引用某个对象视图的基表的表的外键构造一个 REF(引用)。MAKE_REF 允许在已有的外键关系上构造引用。相关示例请参阅第 41 章。

MAX

参阅： AGGREGATE FUNCTIONS、COMPUTE、MIN、第 9 章。

格式：

```
MAX([DISTINCT | ALL] value) [OVER analytic_clause]
```

描述：MAX 是一组行的所有 value 的最大值。MAX 忽略 NULL 值。DISTINCT 选项在这里无意义，因为所有值的最大值与不同值的最大值相同。

MEDIAN

参阅：AVG、COUNT。

格式：

```
MEDIAN ( expr ) [OVER (query_partition_clause)]
```

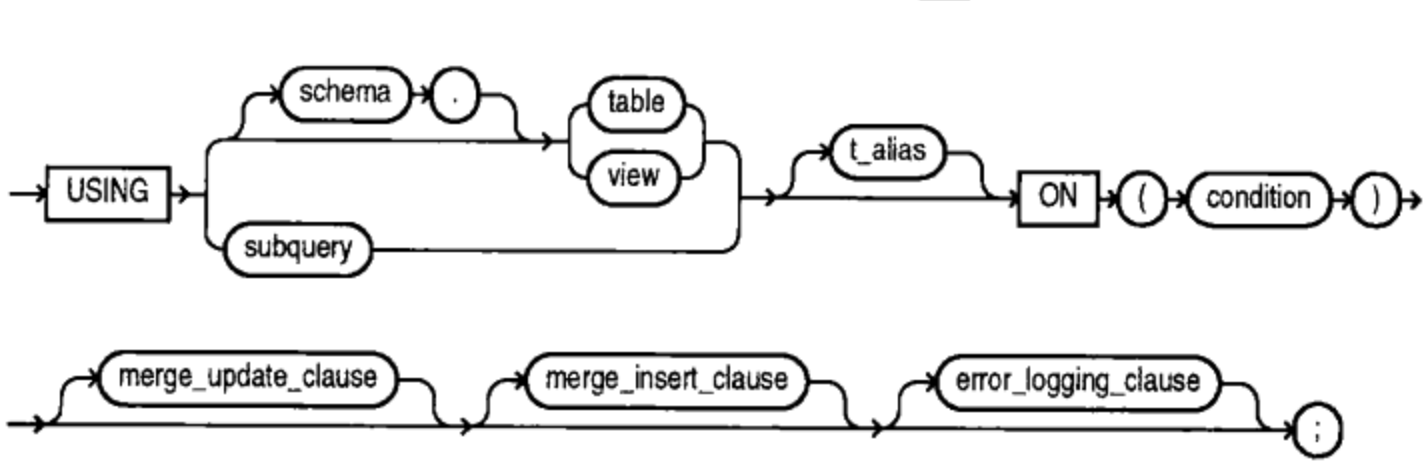
描述：MEDIAN 接受一个数值或日期时间值。一旦将值进行排序后，它就返回中间值或者可能是中间值的内插值。NULL 被忽略。

MERGE

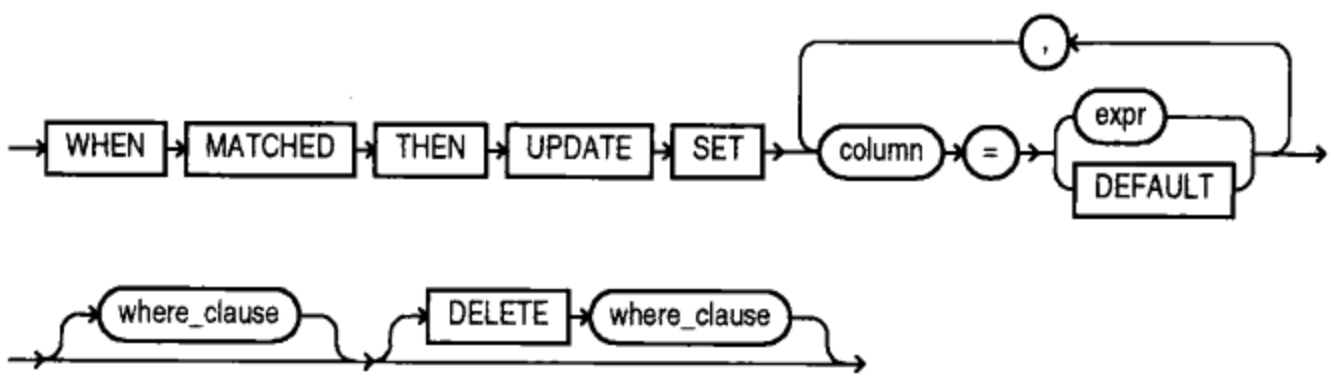
参阅：INSERT、UPDATE、第 15 章。

格式：

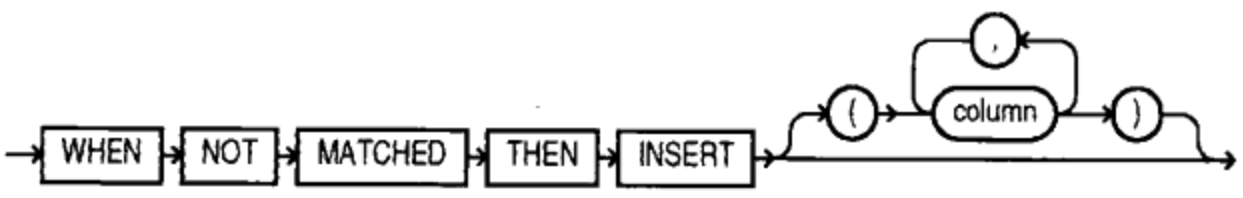
merge::=

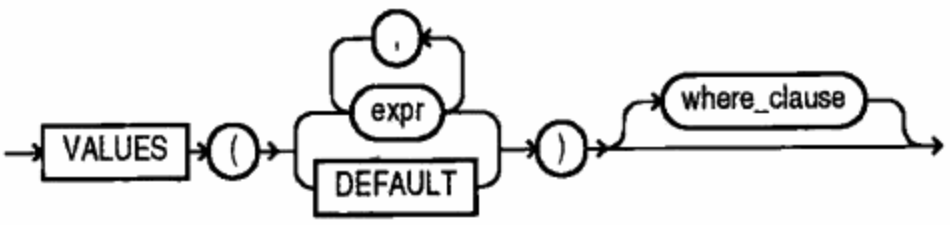


merge_update_clause::=



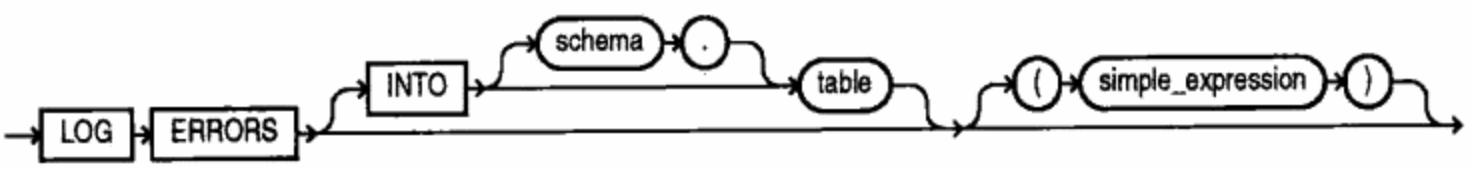
merge_insert_clause::=

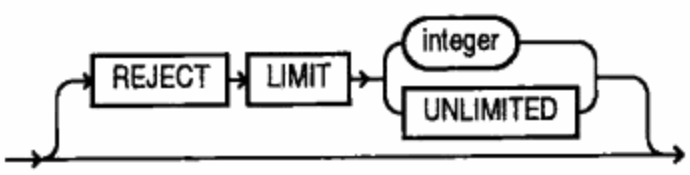




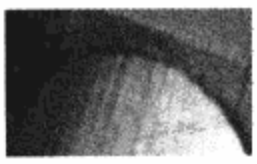
where_clause::=

error_logging_clause::=





描述：MERGE 基于已选的一组行，在单个命令中执行目标表的更新和插入。可以指定条件来决定插入或更新操作是否逐行执行。



注意：
由于 MERGE 执行查询，因此可以使用提示。

用户可以指定 UPDATE 或 INSERT 操作，或者指定两者。而且，也可以在 UPDATE 操作过程中从目标表中删除行。

在 Oracle Database 11g 中，用户可以在具有域索引的表上使用 MERGE 命令。

MIN

参阅：AGGREGATE FUNCTIONS、COMPUTE、MAX、第 9 章。

格式：

```
MIN([DISTINCT | ALL] value) [OVER (analytic_clause)]
```

描述：MIN 是一组行中所有 value 的最小值。MIN 忽略 NULL 值。DISTINCT 选项是没有意义的，因为所有值的最小值等于不同值的最小值。

MINUS

参阅：INTERSECT、QUERY OPERATORS、UNION、第 13 章和第 27 章。

格式：

```
select
MINUS
select
```

在针对 CONTEXT 索引的 Oracle Text 查询中：

```
select column
  from TABLE
where CONTAINS(Text,'text MINUS text') >0;
```

描述：MINUS 组合两个查询。它只返回来自第一个 SELECT 语句且不出现在第二个 SELECT 语句中的行(第一个 SELECT 减去(MINUS)第二个 SELECT)。两个 SELECT 语句之间列的数目及数据类型必须相同，尽管列名不需要相同。MINUS 排除的行中的数据必须相同。关于 INTERSECT、MINUS 和 UNION 的重要差异和作用的讨论请参阅第 13 章。

在针对 CONTEXT 索引的文本搜索中，MINUS 告诉两项的文本搜索，在比较结果和阈得分之前，从第一个项的搜索得分中减去第二个项的搜索得分。

MOD

参阅：NUMBER FUNCTIONS、第 9 章。

格式：

```
MOD(value, divisor)
```

描述：MOD 用 value 除以 divisor，并给出余数。MOD(23，6) = 5 表示 23 除以 6，商为 3 余数为 5，因此取模的结果为 5。value 和 divisor 都可以为任何实数，但是 divisor 不能为 0。

示例：

```
MOD(100,10)    = 0
MOD(22,23)     = 22
MOD(10,3)      = 1
MOD(-30.23,7)  = -2.23
MOD(4.1,.3)    =   .2
```

第二个示例说明了当除数比被除数大时 MOD 所做的操作。它将被除数作为余数。请注意下面这种重要情况：

```
MOD(value,1) = 0
```

如果 value 为一个整数。这是测试一个数是否为整数的很好的方法。

MODEL 函数

模型函数只与行间计算有关，并且只能用于 SELECT 语句的 MODEL 子句中。它们是非递归的。模型函数有 CV、ITERATION_NUMBER、PRESENTNNV、PRESENTV 和 PREVIOUS。

MONTHS_BETWEEN

参阅：ADD_MONTHS、DATE FUNCTIONS、第 10 章。

格式：

```
MONTHS_BETWEEN(date2, date1)
```

描述：MONTHS_BETWEEN 给出以月表示的 date2 减去 date1 的值。结果一般不是整数。

NANVL

参阅： LNNVL。

格式：

```
NANVL ( m, n )
```

描述： NANVL 只对 BINARY_FLOAT 或 BINARY_DOUBLE 类型的浮点数是有用的。如果输入值 m 不是一个数字，则 Oracle 返回另外一个值 n；否则它返回 m。

NCHAR

NCHAR 是 CHAR 数据类型的多字节版本。它存储最大长度为 2000 个字节的定长字符数据。请参阅 DATATYPES。

NCHR

参阅： CHR。

格式：

```
NCHR(number)
```

描述： NCHR 以国家语言字符集返回 number 的二进制等价字符。

NCLOB

NCLOB 数据类型是 CLOB 数据类型的多字节版本。NCLOB 存储包含 Unicode 字符的字符大对象，其长度最大为 4GB。请参阅 DATATYPES。

NEAR

参阅： CONTAINS、第 27 章。

描述： 在 CONTEXT 索引中，NEAR 指出应该对给定文本串执行接近搜索。如果搜索项为“summer”和“lease”，则这两个词的接近搜索为：

```
select Text
  from SONNET
 where CONTAINS(Text,'summer NEAR lease') >0;
```

在判定文本搜索结果时，单词“summer”和“lease”相互较为接近的文本的得分比“summer”和“lease”相互分开较远的文本的得分要高。

NEW_TIME

请参阅 DATE FUNCTIONS。

NEXT_DAY

请参阅 DATE FUNCTIONS。

NEXTVAL

请参阅 PSEUDO-COLUMNS。

NLS_CHARSET_DECL_LEN

参阅：NCHAR。

格式：

```
NLS_CHARSET_DECL_LEN ( byte_count , char_set_id )
```

描述：NLS_CHARSET_DECL_LEN 返回 NCHAR 列的声明宽度(用字符数表示)。

NLS_CHARSET_ID

参阅：NLS_CHARSET_NAME。

格式：

```
NLS_CHARSET_ID ( text )
```

描述：NLS_CHARSET_ID 返回与字符集名 text 相对应的字符集 ID 号。

NLS_CHARSET_NAME

参阅：NLS_CHARSET_ID。

格式：

```
NLS_CHARSET_NAME ( number )
```

描述：NLS_CHARSET_NAME 返回与 ID 号 number 相对应的字符集名称。

NLS_INITCAP

参阅：CHARACTER FUNCTIONS、INITCAP、第 7 章。

格式：

```
NLS_INITCAP(number[, nls_parameters])
```

描述：NLS_INITCAP 除了增加一个参数串外，与 INITCAP 函数相同。nls_parameters 串括在单引号中，给出大写的特定语言序列的排序序列。一般来说，虽然以会话的默认排序序列来使用 INITCAP，但是这个函数允许指定要使用的精确的排序序列。

NLS_LOWER

参阅：CHARACTER FUNCTIONS、LOWER、第 7 章。

格式：

```
NLS_LOWER(number[, nls_parameters])
```

描述：NLS_LOWER 除了增加一个参数串外，与 LOWER 函数相同。nls_parameters 串括在单引号中，给出小写的特定语言序列的排序序列。一般来说，虽然以会话的默认分类序列来使用 LOWER，但是这个函数允许指定要使用的精确的排序序列。

NLS_UPPER

参阅：CHARACTER FUNCTIONS、UPPER、第 7 章。

格式：

```
NLS_UPPER(number[, nls_parameters])
```

描述：NLS_UPPER 除了增加一个参数串外，与 UPPER 函数相同。nls_parameters 串括在单引号中，给出大写的特定语言序列的排序序列。一般来说，虽然以会话的默认排序序列来使用 UPPER，但是这个函数允许指定要使用的精确的排序序列。

NLSSORT

参阅：CREATE DATABASE。

格式：

```
NLSSORT(character[, nls_parameters])
```

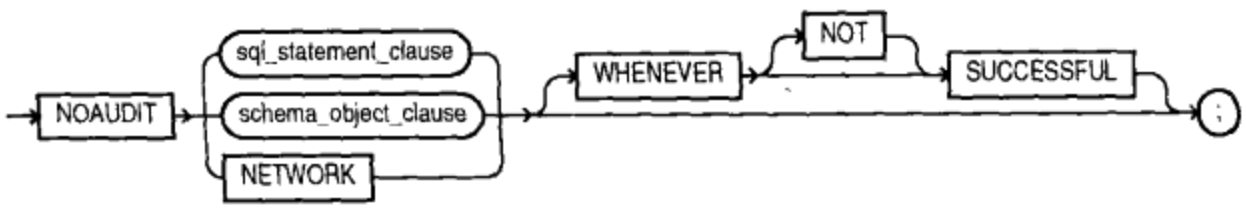
描述：NLSSORT(National Language Support SORT，国家语言支持排序)基于为本站点选择的 National Language Support 选项，给出给定 character 的排序序列值(一个整数)。如果不想使用会话或数据库的默认排序序列，则可用括在单引号内的 nls_parameters 串指定特定语言序列的排序序列。

NOAUDIT

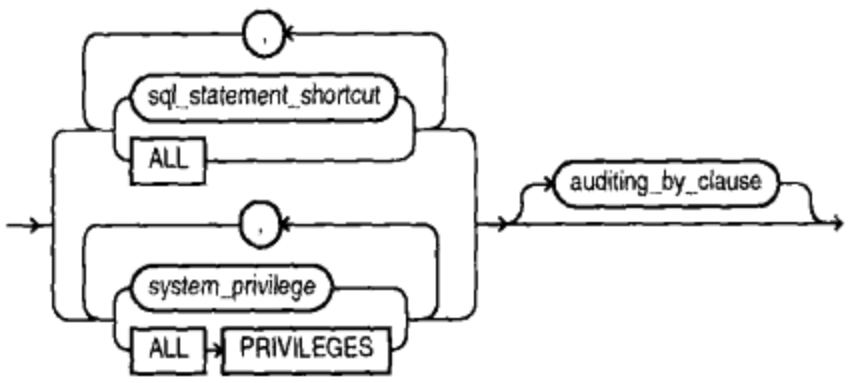
参阅：AUDIT、PRIVILEGE。

格式：

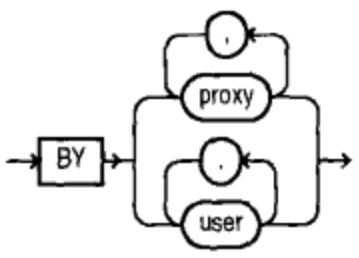
noaudit::=



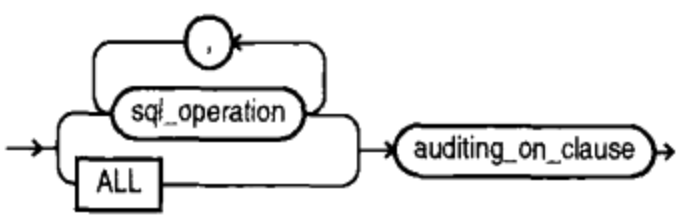
audit_operation_clause::=



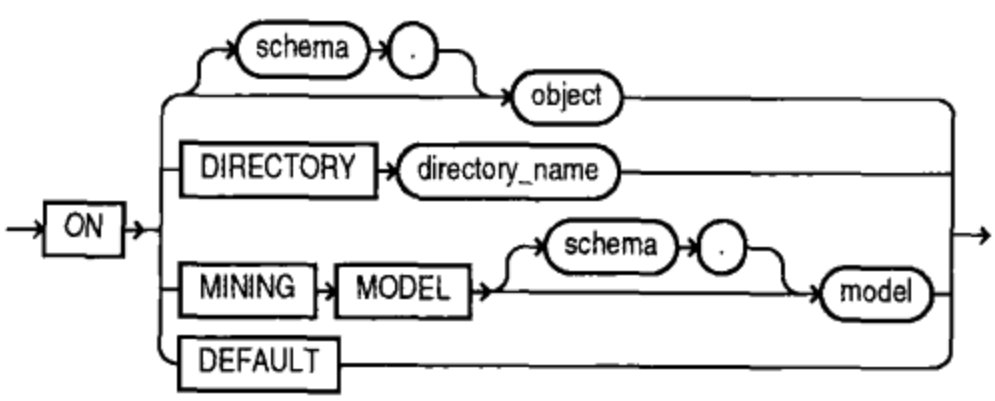
auditing_by_clause::=



audit_schema_object_clause::=



auditing_on_clause::=



描述：NOAUDIT 停止对作为 AUDIT 命令的结果进行审计的 SQL 语句的审计。它停止审计一条语句(参阅 AUDIT)或者停止审计由系统权限授权(参阅 PRIVILEGE)的一条语句。如果有一条包含用户列表的 BY 子句，则此命令停止这些用户发布的语句的审计。如果没有 BY 子句，则 Oracle 停止审计所有用户的语句。WHENEVER SUCCESSFUL 选项只停止审计那些成功完成的语句，WHENEVER NOT SUCCESSFUL 只停止审计那些导致错误的语句。

NOAUDIT 还停止使用表、视图或同义词的选项的审计。要停止任何表、视图或同义词的审计，用户必须拥有它们，或者具有 DBA 权限。option 指简要描述的选项。user 是对象所有者的用户名。object 是一个表、视图或同义词。option 指定将对什么命令停止审计。ON object 指定正在被审计的对象，包括表、视图或者表、视图或序列的同义词。ALL 停止对象上所有命令的审计。

示例：下例停止对 BOOKSHELF 表上所有 UPDATE 或 DELETE 的审计：

```
noaudit update, delete on BOOKSHELF;
```

下面的命令停止对 BOOKSHELF 表的所有不成功的访问的审计：

```
noaudit all on BOOKSHELF whenever not successful;
```

NOT

参阅：LOGICAL OPERATORS、第 5 章。

描述：NOT 出现在逻辑运算符 BETWEEN、IN、LIKE 和 EXISTS 的前面，使其作用相反。NOT 也可以出现在 NULL 的前面，如 IS NOT NULL。

NOT EXISTS

参阅：ALL、ANY、EXISTS、IN、第 13 章。

格式：

```
select ...
where NOT EXISTS (select...);
```

描述：如果跟在 WHERE 子句后面的子查询返回一行或多行，则 WHERE 子句中的 NOT EXISTS 返回 false。子查询中的 SELECT 子句可以是一列、一个字面量或一个星号，这无关紧要。关键的是子查询中的 WHERE 子句是否返回一行。

示例：NOT EXISTS 常常用来确定一个表中的哪些记录在另一个表中没有匹配的记录。关于使用 NOT EXISTS 的示例请参阅第 13 章。

NTILE

参阅：WIDTH_BUCKET。

格式：

```
NTILE ( expr ) OVER ( [ query_partition_clause] order_by_clause )
```

描述：NTILE 是一个分析函数。它把一个有序的数据集划分成许多由表达式指示的数据块(bucket)，并给每一行分配合适的块号。对这些数据块进行编号，并且对于每个分区表达式必须解析为一个正的常数。

NULL(格式 1——PL/SQL)

参阅：BLOCK STRUCTURE。

格式：

```
NULL;
```

描述：NULL 语句与 NULL 值无关。其主要目的是通过说“什么也不做”，使代码可读性更强。它还提供了一种得到空块的方式(因为 PL/SQL 在 BEGIN 和 END 之间至少需要一条可执行语句)。它通常用做一系列条件测试中的某一个测试(一般是最后一个)后面的语句。

示例：

```
IF Age > 65 THEN
    ...
ELSEIF AGE BETWEEN 21 and 65 THEN
    ...
ELSE
 NULL;
ENDIF;
```

NULL(格式 2——SQL 列值)

参阅：AGGREGATE FUNCTIONS、CREATE TABLE、INDICATOR VARIABLE、NVL 和第 5 章。

描述：NULL 值是一个未知的、无关的或没有意义的值。任何 Oracle 数据类型都可以是 NULL。也就是说，一个给定行中的任何 Oracle 列都可以没有值(除非在创建表时指定该列为 NOT NULL)。NUMBER 数据类型中的 NULL 与零不一样。

几乎没有过程语言直接支持具有 NULL 或未知值的变量的思想，尽管 Oracle 和 SQL 很容易就做到这一点。为了扩展 Oracle 已经为其开发了预编译程序的语言以支持 NULL 概念，将 INDICATOR VARIABLE 关联到其宿主变量，就像一个标志，用来指出该变量是否为

NULL。宿主变量及其指示器变量在宿主语言代码中可以分别引用和设置，但是在 SQL 或 PL/SQL 中总是连接在一起的。

可以利用 NVL 函数来检测一列是否有值，并将 NULL 值转换为该列的数据类型的一个具体的值。例如，NVL(Name，'NOT KNOWN')将 Name 列中的 NULL 值转换为单词 NOT KNOWN。对于非 NULL 值(即存在一个名字)，NVL 函数只返回相应的名字。NVL 函数的作用与 NUMBER 和 DATE 函数的作用类似。

除 COUNT(*)和 COMPUTE NUMBER 外，聚集函数忽略空值。而其他的函数，在其判定的值中存在 NULL 值时，它们返回一个空值：

```
NULL + 1066 is NULL.
LEAST(NULL,'A','Z') is NULL.
```

因为 NULL 表示一个未知的值，所以值为 NULL 的两列互不相等。因此，等号和其他逻辑运算符(除 IS NULL 和 IS NOT NULL 外)不能使用 NULL。例如：

```
where Name = NULL
```

不是一条有效的 WHERE 子句。NULL 需要使用单词 IS：

```
where Name IS NULL
```

在 ORDER BY 的排序过程中，如果是升序，则 NULL 值总是第一个出现，而降序时 NULL 值总是最后一个出现。

在数据库中存储 NULL 值时，如果它们位于具有实际值的两列之间，则用单个字节表示它们。如果位于行的最后(CREATE TABLE 中列定义的最后)，则不用字节表示它们。如果表中有的列可能经常要包含 NULL 值，则一般将这些列安排在 CREATE TABLE 列列表的末尾；这样能节省少量的磁盘空间。

NULL 值不出现在索引中，只有一种情况例外，那就是当群集键中所有值都是 NULL 的情况。

NULLIF

参阅：COALESCE、DECODE、NULL、第 16 章。

格式：

```
NULLIF ( expr1 , expr2 )
```

描述：NULLIF 比较 expr1 和 expr2。如果它们相等，则函数返回 NULL。如果它们不相等，则函数返回 expr1。不能为 expr1 指定字面量 NULL。

NULLIF 函数在逻辑上等价于下面的 CASE 表达式：

```
CASE WHEN expr1 = expr2 THEN NULL
ELSE expr1 END
```

NUMBER DATATYPE

NUMBER 数据类型是一种可以包含数值的标准的 Oracle 数据类型，所包含的数值可以有也可以没有小数点和符号。有效的值为 0、正数和负数。NUMBER 数据类型定义了精度和

数值范围。精度可以从 1~38；数值范围可以从 - 84~127。

NUMBER FORMATS

参阅： COLUMN、第 9 章。

描述： 表 A-17 中的选项对 SET NUMFORMAT 和 COLUMN FORMAT 命令都有效：

表 A-17 对 SET NUMFORMAT 和 COLUMN FORMAT 命令都有效的选项

选 项	描 述
9999990	9 和 0 的数目决定了能够显示的最大数字位数
999,999,999.99	逗号和小数点将以所示的模式放置
999990	显示尾部的 0
099999	显示前置的 0
$99999	在每个数字前面加上$符号
B99999	如果定点数的整数部分为 0，则将整数部分显示为空
99999MI	如果数是负数，则减号跟在数的后面。默认情况下负号位于左边
S9999	如果是正数，则在数的前面放置“+”；如果是负数，则数的前面放置“—”
9999S	如果是正数，则在数的后面放置“+”；如果是负数，则数的后面放置“—”
99999PR	负数显示在“<”和“>”内
99D99	在指定的位置显示小数点
9G999	在所示的位置显示组分隔符
C9999	在此位置显示 ISO 货币符号
9G999	在指定位置返回组分隔符(NLS_NUMERIC_CHARACTER 参数的当前值)。可以在数字格式模型中指定多个组分隔符
L999	在指定的位置显示本地货币符号
,	显示逗号
.	显示句号
9.999EEEE	以科学计数法显示(需要 4 个 E)
999V99	数值乘以 10n，n 是 V 右边的数字位数。999V99 把 1234 转换成 123400
9999RN	对 1~3999 之间的整数，返回其对应的罗马数字的值。当使用小写的“rn”时，以小写形式显示罗马数字
TM	尽可能返回的字符的最小值(文本最小值)
U999	在指定的位置返回欧元或其他双重货币符号
X	返回指定数字位数的十六进制值

NUMBER FUNCTIONS

下面是 SQL 的 Oracle 版本中所有当前单值数值函数的一个有序列表，如表 A-18 所示。每个函数都会在本附录的其他地方按照各自的名字列出，并给出正确的格式和用法。这些函数都可以用作常规的 SQL 函数以及 PL/SQL 函数。请参阅 AGGREGATE FUNCTIONS 和 LIST FUNCTIONS。

表 A-18　SQL 的 Oracle 版本中的所有单值数值函数

函　数	定　义
Value1＋value2	加法运算
value1－value2	减法运算
value1 * value2	乘法运算
value1 / value2	除法运算
ABS(value)	value 的绝对值(ABSolute)
ACOS(value)	value 的反余弦(Arc COSine)，以弧度表示
ASIN(value)	value 的反正弦(Arc SINe)，以弧度表示
ATAN(value)	value 的反正切(Arc TANgent)，以弧度表示
ATAN2(value1, value2)	value1 和 value2 的反正切(Arc TANgent)，以弧度表示
BITAND(value1, value2)	value1 和 value2 的按位与(BITwise AND)，value1 和 value2 都必须解析为非负整数，并返回一个整数
CEIL(value)	数值限度(CEILing)：大于或等于 value 的最小整数
COS(value)	value 的余弦(COSine)
COSH(value)	value 的双曲余弦(Hyperbolic COSine)
EXP(value)	指数(EXPonent)运算(e 的 value 次方)
FLOOR(value)	小于或等于 value 的最大整数
LN(value)	value 的自然对数(Natural Logarithm)
LOG(value)	以 10 为底的 value 的对数(LOGarithm)
MOD(value，divisor)	取模(MODulus)运算
NANVL(value1, value2)	对 BINARY_FLOAT 和 BINARY_DOUBLE 类型的数，如果 value1 不是一个数，则返回 value2
POWER(value, exponent)	value 的 exponent 次幂(POWER)
REMAINDER(value1, value2)	value1 除以 value2 的余数
ROUND(value, precision)	按 precision 将 value 舍入(ROUND)
SIGN(value)	如果 value 为正则为 1；如果为负则为－1；如果为零则为 0
SIN(value)	value 的正弦(SINe)
SINH(value)	value 的双曲正弦(Hyperbolic SINe)
SQRT(value)	value 的平方根(SQuare Root)
TAN(value)	value 的正切(TANgent)
TANH(value)	value 的双曲正切(Hyperbolic TANgent)
TRUNC(value, precision)	按 precision 截取(TRUNCate)value
WIDTH_BUCKET(expr, min, max, num)	构造等宽直方图

NUMTODSINTERVAL

参阅：DATATYPES、第 11 章。

格式：

```
NUMTODSINTERVAL ( n , 'char_ expr' )
```

描述：NUMTODSINTERVAL 将 n 转换为一个 INTERVAL DAY TO SECOND 字面量。n 可以是一个数或者是解析为一个数的表达式。char_expr 可以是 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型。char_expr 的值指定 n 的单位，并且必须解析为下列串值中的一个：‘DAY’、‘HOUR’，‘MINUTE’ 或 ‘SECOND’。

NUMTOYMINTERVAL

参阅：DATATYPES、第 11 章。

格式：

```
NUMTOYMINTERVAL ( n , 'char_ expr' )
```

描述：NUMTOYMINTERVAL 将 n 转换为一个 INTERVAL YEAR TO MONTH 字面量。n 可以是一个数或者是解析为一个数的表达式。char_expr 可以是 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型。char_expr 的值指定 n 的单位，并且必须解析为 ‘YEAR’ 或 ‘MONTH’。

NVARCHAR2

NVARCHAR2 为 VARCHAR2 数据类型的多字节等价形式。它的最大长度为 4 000 字节。请参阅 DATATYPES。

NVL

参阅：AGGREGATE FUNCTIONS、NULL、OTHER FUNCTIONS、第 9 章。

格式：

```
NVL(value, substitute)
```

描述：如果 value 为 NULL，则函数返回 substitute。如果 value 非 NULL，则函数返回 value。value 可以是任何 Oracle 数据类型。substitute 可以是一个字面量、另一列或一个表达式，但是必须与 value 的数据类型相同。

NVL2

参阅：AGGREGATE FUNCTIONS、NULL、OTHER FUNCTIONS、第 9 章。

格式：

```
NVL2 ( expr1 , expr2 , expr3 )
```

描述：NVL2 为 NVL 的扩展形式。在 NVL2 中，不能返回 expr1，而是返回 expr2 或 expr3。如果 expr1 非 NULL，则 NVL2 返回 expr2。如果 expr1 为 NULL，则 NVL2 返回 expr3。参数 expr1 可以具有任何数据类型。参数 expr2 和 expr3 可以是除 LONG 以外的任意数据类型。

OBJECT NAMES

可以给数据库对象命名，如给表、视图、同义词、别名、列、索引、用户、序列、表空间等等命名。给对象命名的规则如下：

- 对象名的长度为 1~30 个字符，但是数据库名和宿主文件名除外，数据库名最多为 8 个字符，宿主文件名的长度则依赖于操作系统。
- 名字不能包含引号。
- 一个名字必须满足以下条件：
  - 以字母开头
  - 只包含字符 A~Z、0~9、$、#、_
  - 不能是 Oracle 的保留字(请参阅 RESERVED WORDS)
  - 不能与同一个用户所拥有的其他数据库对象的名字重复

对象名不区分大小写。对象名应该遵循有意义的命名约定。

OBJECT REFERENCE FUNCTIONS

对象引用函数操纵 REF，其中 REF 是对指定对象类型的对象的引用。对象引用函数在该引用中的各条目下进行描述。这类函数有 DEREF、MAKE_REF、REF、REFTOHEX 和 VALUE。

OID

OID 是一个对象标识符，它是 Oracle 为数据库中每个对象分配的。例如，对象表中的每个行对象都具有分配给它的一个 OID 值；此 OID 用来解决对行对象的引用。Oracle 在删除一个对象后，不重用其 OID 值。请参阅第 41 章。

OPEN

参阅：CLOSE、DECLARE CURSOR、FETCH、LOOP、第 32 章。

格式：

```
OPEN cursor [( parameter[, parameter]...)
```

描述：OPEN 与 DECLARE CURSOR 和 FETCH 联合使用。DECLARE CURSOR 创建一条要执行的 SELECT 语句，并创建在其 WHERE 子句中使用的参数(PL/SQL 变量)的一个列表，但是它并不执行查询。

实际上，OPEN cursor 在指定的游标中执行查询并将结果保存在一个临时区域(staging area)中。在这个区域中可以利用 FETCH 每次一行地将结果调入，并用 FETCH 的 INTO 子句将它们的列值放入局部变量中。如果游标 SELECT 语句使用参数，则参数的实际值被传递给 OPEN 语句的参数列表中的 SELECT 语句。它们必须在数量和位置上匹配，并且具有兼容的数据类型。

还有另一种方法可以把 OPEN 语句中的值与 SELECT 列表中的值关联起来：

```
DECLARE
    cursor mycur(Title, Publisher) is select ...
```

```
BEGIN
   open my_cur(new_book => Title, 'PANDORAS' => Publisher);
```

这里，new_book 是一个 PL/SQL 变量(它或许是从一个数据输入对话框中加载的)，它指向 Title，因此将变量 new_book 中当前的内容加载到 Title 中。Publisher 加载值'PANDORAS'，这样它们就成了游标 my_cu 的参数(关于游标中参数的更详细的内容请参阅 DECLARE CURSOR)。

虽然也可以将指向的关联和位置关联结合起来，但位置关联必须首先出现在 OPEN 语句的列表中。

尽管可以先关闭游标然后再重新打开它，但是不能重新打开一个已经打开的游标。也不能使用在 CURSOR FOR LOOP 中具有当前 OPEN 语句的游标。

OPEN CURSOR(嵌入式 SQL)

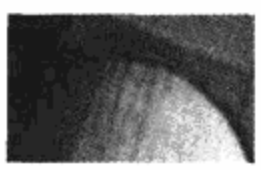
参阅：CLOSE、DECLARE CURSOR、FETCH、PREPARE、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [FOR { integer | :array_size }]
    OPEN cursor
      [USING
       {DESCRIPTOR SQLDBA_descriptor
       | :variable[[ INDICATOR ]:indicator_variable
       [,:variable[ INDICATOR ]:indicator_variable]]... }]
```

描述：cursor 是先前在 DECLARE CURSOR 语句中命名的一个游标的名称。可选的 USING 可以基于位置(变量的个数和数据类型必须相同)引用在 DECLARE CURSOR 的语句中替换的变量的宿主变量列表；也可以引用描述符名，此描述符名引用先前的 DESCRIBE 的结果。

OPEN cursor 分配一个游标，定义行的活动集合(在打开游标时替代宿主变量)，并且将游标恰好定位在集合的第一行前。在执行 FETCH 之前不检索任何行。宿主变量一旦替换后就不再改变。要改变它们，必须重新打开游标(不必先关闭它)。



注意：
所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

OR

参阅：AND、CONTAINS、第 5 章和第 27 章。

描述：OR 将两个逻辑表达式组合在一起，如果两个逻辑表达式任何一个为真，则结果为 TRUE。

示例：下面的查询将得到用于城市 KEENE 和 SAN FRANCISCO 的数据：

```
select * from COMFORT
   where City = 'KEENE' OR City = 'SAN FRANCISCO'
```

在 CONTEXT 查询中，OR 运算符可用于涉及多个搜索项的搜索。如果找到两个搜索项中的任意一个，且它的搜索得分超过指定的阈值，则搜索将返回相应的文本。

ORA_HASH

格式：

```
ORA_HASH ( expr [, max_bucket [,seed_value] ])
```

描述：ORA_HASH 计算给定表达式的散列值。该函数对于分析数据的子集以及生成随机样本这类的操作是有用的。expr 是 Oracle 要计算其散列值的数据(通常是一个列名)。max_bucket 决定最大数据块(bucket)的值(0~4 294 967 295)。参数 seed_value(0~4 294 967 295)使得 Oracle 能够对相同的 expr 得到不同的结果。

ORDER BY

参阅：FROM、GROUP BY、HAVING、SELECT、WHERE、第 5 章。

格式：

```
ORDER BY {expression [,expression]... |
          position [,position]... }
          alias [,alias]... }
[ ASC | DESC
```

描述：ORDER BY 子句使 Oracle 在显示查询结果前对这些结果进行分类。这可以通过 expression(expression 可以是一个简单的列名或者一组复杂的函数)、alias(请参阅下面的注意)或 SELECT 子句中的列 position(请参阅下面的注意)来完成。行根据主机的排序序列首先按第一个表达式或位置排序，然后按第二个表达式或位置排序，依此类推。如果没有指定 ORDER BY，那么从表中选择行的顺序是不确定的，而且从一个查询到下一个查询有可能发生变化。

注意：
虽然 Oracle 仍然支持在 ORDER BY 子句中使用列位置，但是这个功能不再是 SQL 标准的组成部分了，而且也不保证在将来的版本中支持这个功能。应该使用列别名。

ASC 或 DESC 指定升序还是降序，并且可以放在 ORDER BY 子句的每个表达式、位置或别名之后。在 Oracle 中，NULL 值位于升序行之前，跟在降序行之后。

ORDER BY 可以跟在除 FOR UPDATE OF 以外的任何子句之后。

如果同时指定 ORDER BY 和 DISTINCT，则 ORDER BY 子句可能只引用 SELECT 子句中的列或表达式。

当使用 UNION、INTERSECT 或 MINUS 运算符时，第一个 SELECT 中的列名可能与后面的 SELECT 中的名字不同。ORDER BY 必须使用第一个 SELECT 中的列名。

只有 CHAR、VARCHAR2、NUMBER 和 DATE 数据类型(以及特殊的数据类型 ROWID)可以出现在 ORDRE BY 子句中。

示例：

```
select Title, Publisher from BOOKSHELF
order by Title;
```

其他函数

下面是 SQL 的 Oracle 版本中不容易归入其他函数种类的、所有当前函数的一个按字母顺序的列表。每个函数都在本附录的其他地方按照各自的名字列出，并给出正确的格式和用法。

BFILENAME，COALESCE，CV，DECODE，DEPTH，DUMP，EMPTY_BLOB，EMPTY_CLOB，EXISTSNODE，EXTRACT，EXTRACTVALUE，GREATEST，LEAST，LNNVL，NLS_CHARSET_DECL_LEN，NLS_CHARSET_ID，NLS_CHARSET_NAME，NULLIF，NVL，NVL2，ORA_HASH，PATH，PRESENTNNV，PRESENTV，PREVIOUS，SYS_CONNECT_BY_PATH，SYS_CONTEXT，SYS_DBURIGEN，SYS_EXTRACT_UTC，SYS_GUID，SYS_TYPEID，SYS_XMLAGG，SYS_XMLGEN，UID，UPDATEXML，USER，USERENV，VSIZE，XMLAGG，XMLCOLATTVAL，XMLCONCAT，XMLFOREST，XMLSEQUENCE，XMLTRANSFORM

PARAMETERS

参阅：&、&&、ACCEPT、DEFINE。

描述：参数允许使用在 SQL*Plus 命令行上传递的值执行启动文件。这些参数跟在启动文件名之后，用空格隔开。在文件内，它们按照在命令行上出现的顺序依次引用。&1 为第一个，&2 为第二个，依此类推。除此规则之外，使用的规则与使用 DEFINE 或 ACCEPT 加载变量时的规则相同，也可以按相同的方式在 SQL 语句中使用它们。

不过，参数也有一个限制，那就是不能将多个单词组成的参数传递给单个变量。每个变量只能接受一个单词、日期或数值。如果通过在命令行把参数放在引号中来解决该问题，则将导致多个词被连接起来。

示例：假如有一个名为 fred.sql 的启动文件，它包含下面的 SQL 语句：

```
select Title, Publisher
  from BOOKSHELF
 where Title > &1;
```

用下面的命令启动它：

```
start fred.sql M
```

得到以“M”后面的字符开头的所有标题的报表。

PASSWORD(SQL*Plus)

参阅：ALTER USER、第 19 章。

格式：

```
PASSW[ORD] [username]
```

描述：可以在 SQL*Plus 中使用 PASSWORD 命令来修改您的账户或其他账户的口令(需具有 ALTER ANY USER 权限)。如果使用 PASSWORD 命令，则在输入新口令时将不在屏幕上显示它。具有 DBA 权限的用户可以利用 PASSWORD 命令修改任何用户的口令，而其他用户只能修改自己的口令。

在输入 PASSWORD 命令时，系统将会提示用户输入旧口令和新口令。

示例：

```
password
Changing password for dora
Old password:
New password:
Retype new password:
```

在成功地修改了口令后，用户将会收到以下反馈信息：

```
Password changed
```

PATH

参阅：DEPTH。

格式：

```
PATH ( correlation_integer )
```

描述：PATH 是一个只与 UNDER_PATH 和 EQUALS_PATH 条件一起使用的辅助函数。它返回通向父条件中指定资源的相关路径。correlation_integer 值可以是任何 NUMBER 类型的整数。小于 1 的值看作 1。

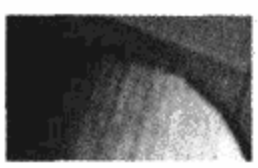
PAUSE(SQL*Plus)

参阅：ACCEPT、PROMPT。

格式：

```
PAU[SE] [text];
```

描述：PAUSE 与 PROMPT 相似，所不同的只是 PAUSE 首先显示空行，然后显示包含 text 的行，最后等待用户按 RETURN 键。如果不输入 text，则 PAUSE 显示两个空行，然后等待用户按 RETURN 键。



注意：
即使命令的输入源重定向到来自一个文件，PAUSE 也将等待来自终端的 RETURN，这表示包含 SET TERMOUT OFF 的启动文件可以挂起，等待 RETURN，而不会有为什么它在等待(或它在等待什么)这样的消息。使用 PAUSE 应该谨慎。

示例：

```
prompt Report Complete.
```

```
pause Press RETURN to continue.
```

PERCENT_RANK

参阅： AGGREGATE FUNCTIONS。

格式：

对于聚集，可以使用：

```
PERCENT_RANK ( expr [, expr]... ) WITHIN GROUP
( ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...)
```

对于分析，可以使用：

```
PERCENT_RANK ( ) OVER ( [ query_partition_clause] order_by_clause )
```

描述： 作为一个聚集函数，对于由函数的参数和对应的分类说明标识的一个假想的行 R，PERCENT_RANK 计算行 R 的等级减 1 再除以聚集组中的行数。进行这个计算，就像假想的行 R 已经被插入 Oracle 在其上进行聚集的行组中一样。此函数的参数在每个聚集组内标识一个假想的行。因此，它们必须全都要对每个聚集组中的固定的表达式进行判定。固定参数表达式和聚集的 ORDER BY 子句中的表达式按位置匹配。因此，参数的个数必须相同且它们的类型也必须兼容。

PERCENT_RANK 返回值的范围为 0~1，包括 0 和 1。任何集合中的第一行的 PERCENT_RANK 为 0。

对于行 R，作为分析函数，PERCENT_RANK 计算 R 的等级减 1，除以 1，小于被判定的行数(整个查询结果集或分区)。

PERCENTILE_CONT

参阅： AGGREGATE FUNCTIONS、PERCENTILE_DISC。

格式：

```
PERCENTILE_CONT ( expr ) WITHIN GROUP ( ORDER BY expr [ DESC | ASC ] )
[OVER ( query_partition_clause )]
```

描述： PERCENTILE_CONT 是一个假想的连续分布模型的反分布函数。它接受一个百分值和一个分类说明，返回一个内插值，这个内插值的范围是分类说明的百分值。在计算中忽略 NULL 值。

第一个 expr 必须等于 0~1 之间的一个数值，因为它是一个百分值。这个 expr 在每个聚集组中必须相同。ORDER BY 子句接受一个必须为数值或日期时间值的表达式，因为它们是 Oracle 能够在其上完成内插的类型。

PERCENTILE_DISC

参阅： AGGREGATE FUNCTIONS、PERCENTILE_CONT。

格式：

```
PERCENTILE_DISC ( expr ) WITHIN GROUP ( ORDER BY expr [ DESC | ASC ] )
[OVER ( query_partition_clause )]
```

描述：PERCENTILE_DISC 是一个假想的离散分布模型的反分布函数。它接受一个百分值和一个分类说明，并从集合中返回一个元素。在计算中忽略 NULL 值。

第一个 expr 必须等于 0 和 1 之间的一个数值，因为它是一个百分值。这个表达式在每个聚集组中必须相同。ORDER BY 子句接受一个表达式，该表达式可以是任何可排序的类型。

对于给定的百分值 P，PERCENTILE_DISC 函数在 ORDER BY 子句中对表达式的值进行排序，并返回大于等于 P 的具有最小 CUME_DIST 值的表达式(相对于相同的排序说明)。

POWER

参阅：SQRT、第 9 章。

格式：

```
POWER(value, exponent)
```

描述：POWER 是 value 的 exponent 次幂。

POWERMULTISET

参阅：第 39 章。

格式：

```
POWERMULTISET ( expr )
```

描述：POWERMULTISET 在输入嵌套表时使用，并返回包含输入嵌套表的所有非空子集(称为“多重集”，submultiset)的嵌套表的一个嵌套表。expr 是可以为一个嵌套表的任何表达式；如果嵌套表为空，则返回一个错误。

POWERMULTISET_BY_CARDINALITY

参阅：POWERMULTISET。

格式：

```
POWERMULTISET ( expr, cardinality )
```

描述：POWERMULTISET_BY_CARDINALITY 在输入嵌套表时使用，并返回包含指定基数的输入嵌套表的所有非空子集(称为“多重集”，submultiset)的嵌套表的一个嵌套表。expr 是可以为一个嵌套表的任何表达式，cardinality 可以是任意正整数；如果嵌套表为空，则返回一个错误。

PRECEDENCE

参阅：LOGICAL OPERATORS、QUERY OPERATORS、第 13 章。

描述：表 A-19 是按优先级的降序列出的运算符。优先级相同的运算符列在同一行上。

优先级相同的运算符按从左到右的顺序进行判定。所有 AND 都在任何 OR 之前执行。这些运算符在本附录中按各自的符号或名字分别列出和描述。

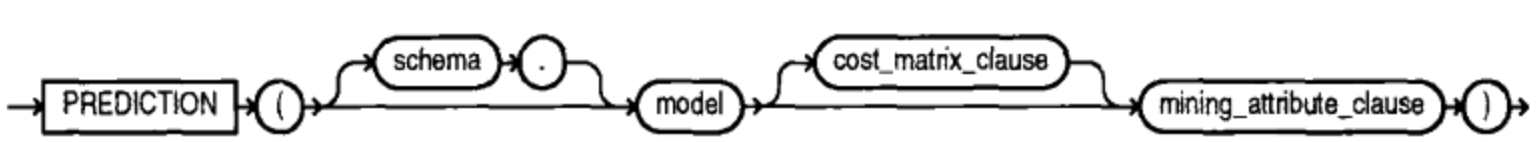
表 A-19 按优先级顺序列出的运算符

运 算 符	功 能
-	继续 SQL*Plus 命令。在下一行上继续一条命令
&	SQL*Plus 启动文件中参数的前缀。将&1、&2 等替换为单词。请参阅 START
& &&	SQL*Plus 的 SQL 命令中替换的前缀。如果找到未定义的&或“&&”变量，则 SQL*Plus 将提示输入一个值。“&&”还定义变量并保存其值；“&”不定义变量也不保存值。请参阅“&”和“&&”、DEFINE 和 ACCEPT
:	PL/SQL 中宿主变量的前缀
.	变量分隔符，在 SQL*Plus 中用来分隔变量名与后缀，从而不将后缀视为变量名的组成部分
()	括住子查询或者列列表
'	括住一个字面量，如一个字符串或日期常量。为了在串常量中使用单引号，可以使用两个单引号(不是双引号)
"	括住包含特殊字符或空格的表或列别名
"	括住 TO_CHAR 的日期格式子句中的字面量文本
@	位于 COPY 中的数据库名之前，或者位于 FROM 子句的链接名之前
()	重写正常的操作符优先级
+-	数或数值表达式的前缀符号(正号或符号)
*/	乘和除
+-	加和减
||	字符值连接
NOT	颠倒表达式的结果
AND	如果两个条件都为真，则返回真
OR	如果两个任中任意一个条件为真，则返回真
UNION	从两个查询中返回所有不同的行
INTERSECT	从两个查询中返回所有匹配的不同的行
MINUS	返回出现在第一个查询中而不出现在第二个查询中的所有不同行

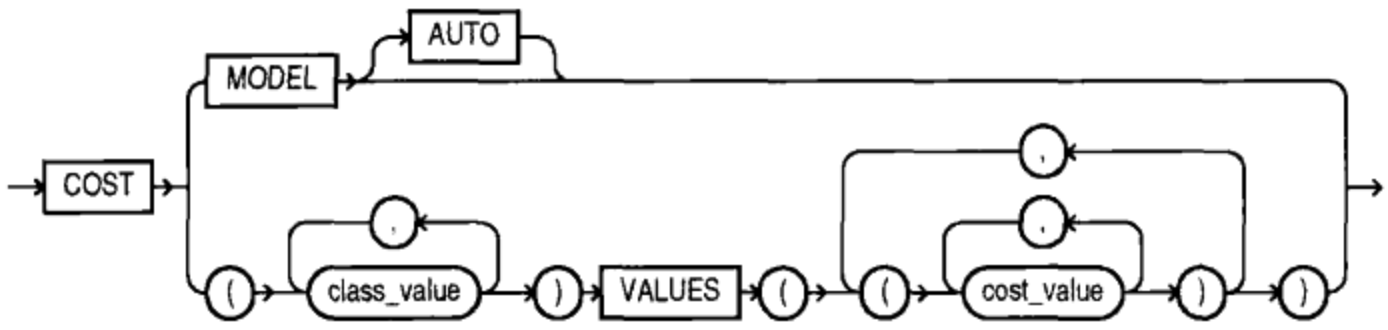
PREDICTION

参阅：PREDICTION_COST、PREDICTION_BOUNDS 和 PREDICTION_SET。

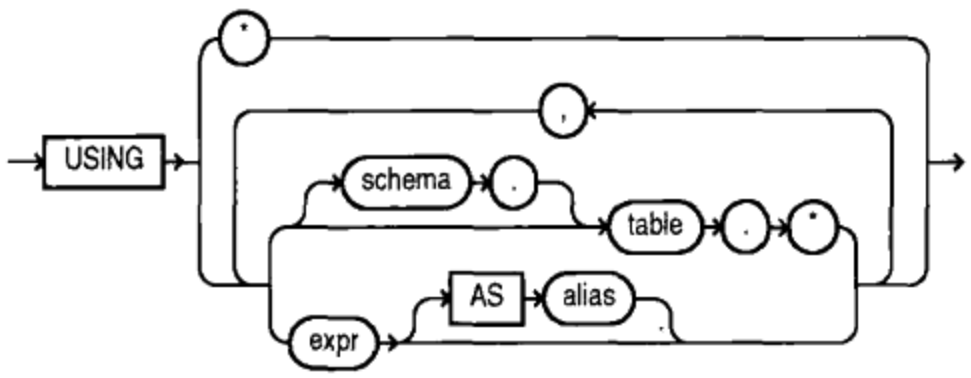
格式：



cost_matrix_clause::=



mining_attribute_clause::=

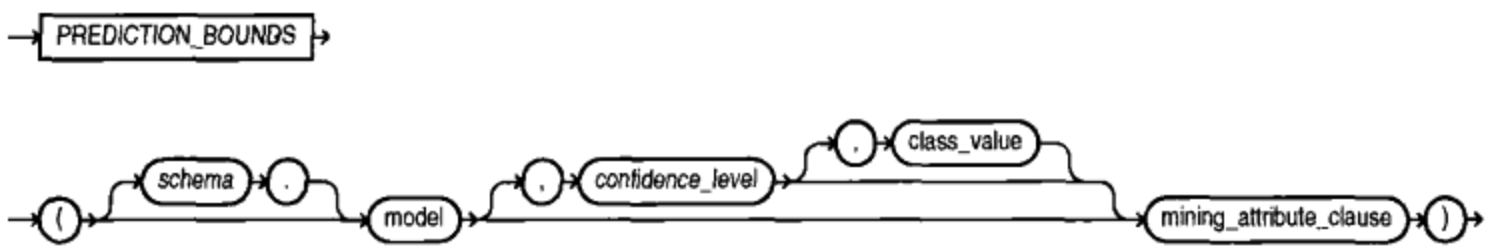


描述：PREDICTION 函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的模型。它返回模型的最佳预测。返回的数据类型取决于模型构建期间所使用的目标值类型。对于回归模型，此函数返回预期值。

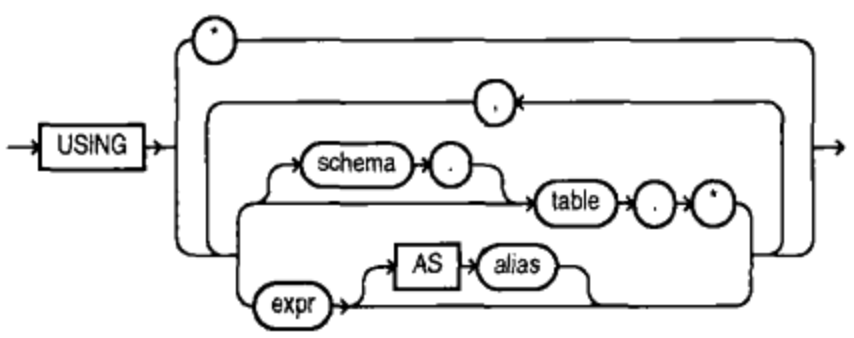
PREDICTION_BOUNDS

参阅：PREDICTION、PREDICTION_COST 和 PREDICTION_SET。

格式：



mining_attribute_clause::=

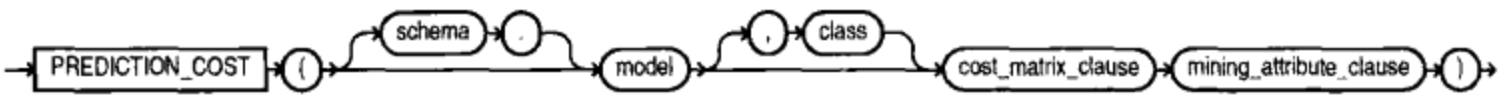


描述：PREDICTION_BOUNDS 函数只用于广义线性模型。它返回一个对象，该对象包含 LOWER 和 UPPER 两个 NUMBER 字段。对于回归挖掘函数，边界适用于预测值。对于分类挖掘函数，边界适用于概率值。如果使用岭回归构建 GLM，或者如果在构建期间发现此协方差矩阵是奇异矩阵，则此函数的 LOWER 和 UPPER 两个字段返回 NULL。

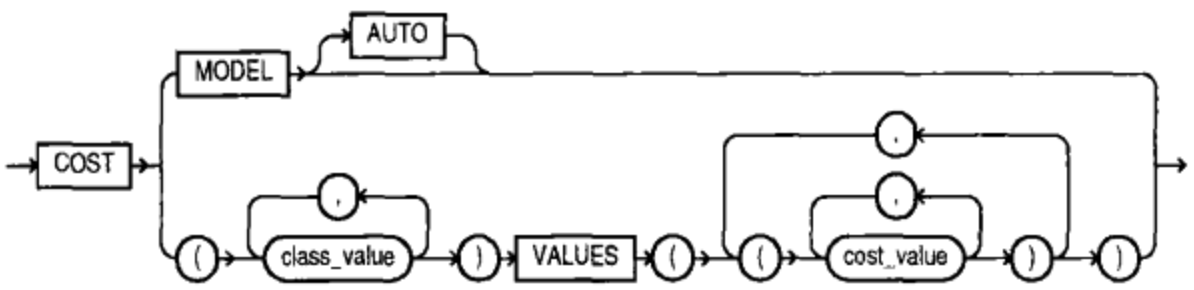
PREDICTION_COST

参阅： PREDICTION。

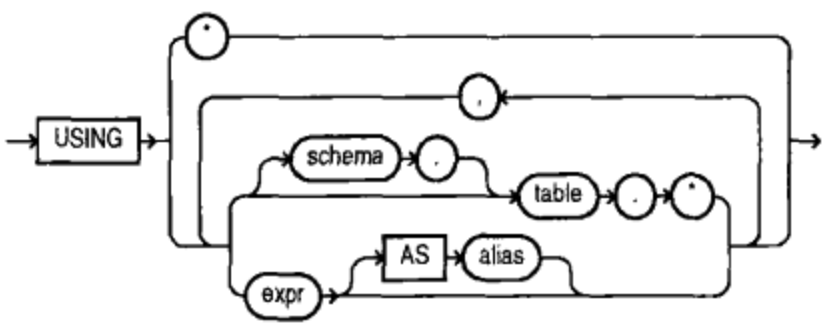
格式：



cost_matrix_clause::=



mining_attribute_clause::=

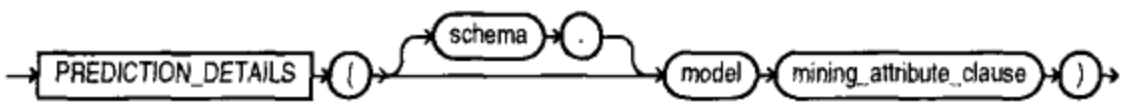


描述： PREDICTION_COST 函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的任何分类模型。它返回给定预测的成本核算，其类型是 Oracle 的 NUMBER。

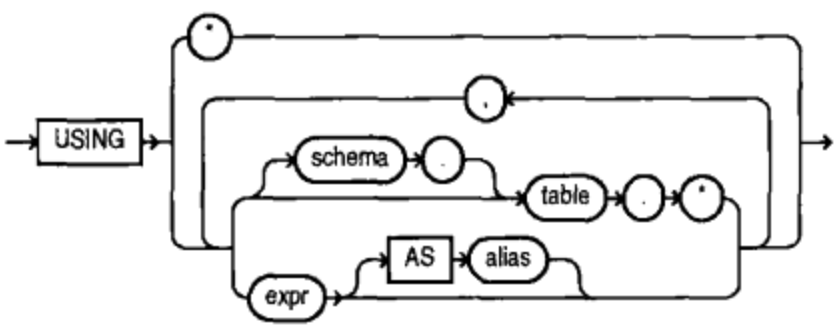
PREDICTION_DETAILS

参阅： PREDICTION、PREDICTION_BOUNDS 和 PREDICTION_COST。

格式：



mining_attribute_clause::=



描述： PREDICTION_DETAILS 函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的决策树模型和奇异特性的自适应贝叶斯网络(ABN)模型。

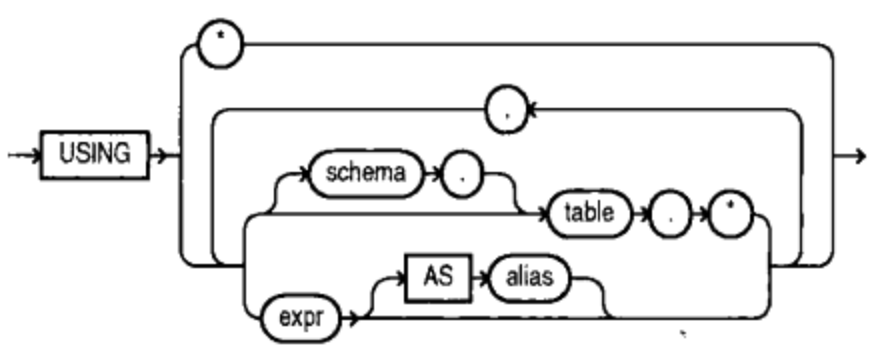
此函数返回一个 XML 字符串，字符串包含与输入行得分相关的、特定于模型的信息。

PREDICTION_PROBABILITY

参阅： PREDICTION、PREDICTION_BOUNDS 和 PREDICTION_COST。

格式：

mining_attribute_clause::=

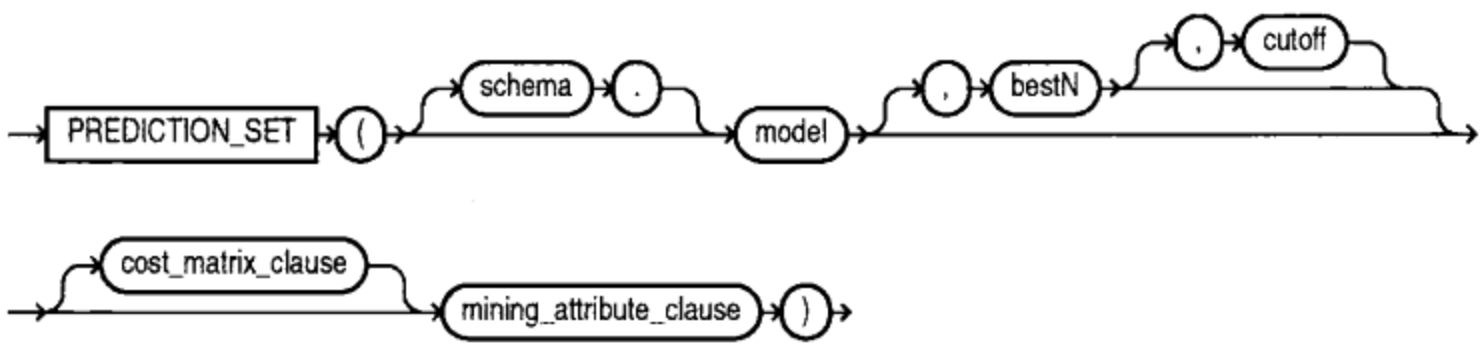


描述： PREDICTION_PROBABILITY 函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的分类模型。此函数对其他类型的模型无效。它返回给定预测的概率，其类型是 Oracle 的 NUMBER。

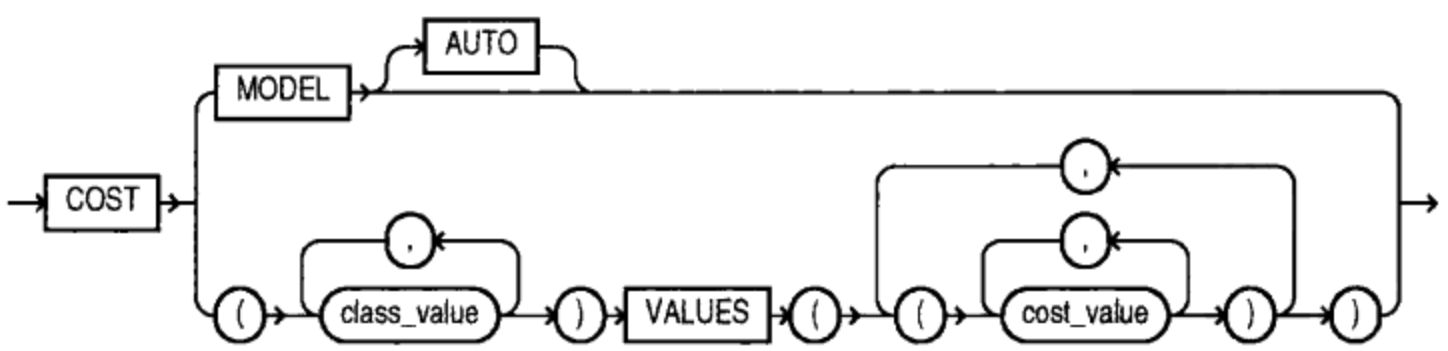
PREDICTION_SET

参阅： PREDICTION、PREDICTION_BOUNDS 和 PREDICTION_COST。

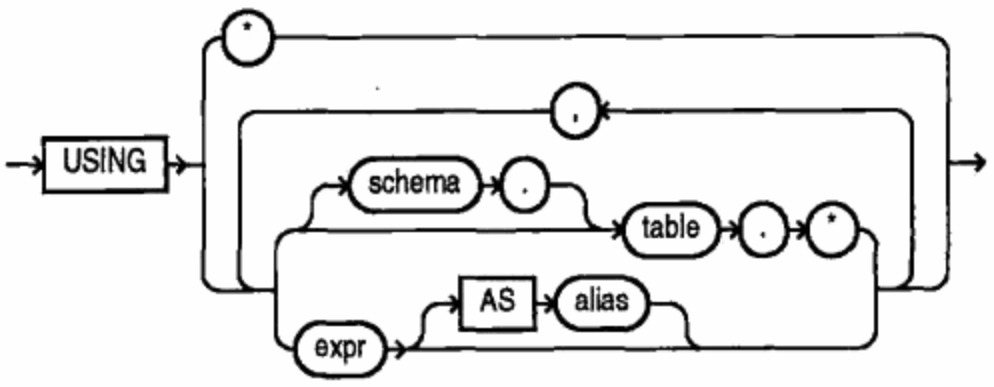
格式：



cost_matrix_clause::=



mining_attribute_clause::=



描述：PREDICTION_SET 函数用于使用 DBMS_DATA_MINING 程序包或使用 Oracle Data Mining Java API 创建的分类模型。此函数对其他类型的模型无效。它返回对象的一个可变数组，对象数组包含多级分类场景中的所有类。这些对象字段命名为 PREDICTION、PROBABILITY 和 COST。PREDICTION 字段的数据类型取决于模型构建期间所使用的目标值类型。其他两个字段的类型都是 Oracle 的 NUMBER。返回的元素是有顺序的，最佳预测排在最前面，最差的预测排在最后面。

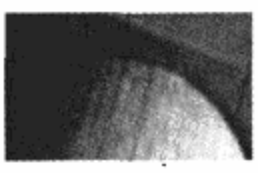
PREPARE(嵌入式 SQL)

参阅：CLOSE、CURSOR、DECLARE CURSOR、FETCH、OPEN、*Programmer's Guide to the Oracle Precompilers*。

格式：

```
EXEC SQL [AT { db_name | :host_variable }] PREPARE statement_id
    FROM { :host_string | 'text' | select_command }
```

描述：PREPARE 分析宿主变量:host_string 或字面量 text 中的 SQL。它分配一个 statement_id 作为对 SQL 的引用。如果以前使用过 statement_id，则此引用替代它。此 SQL 是一条 SELECT 语句，且可能包括 FOR UPDATE OF 子句。:host_string 不是所用的宿主变量的实际名称，而只是一个占位符。OPEN CURSOR 在其 USING 子句中指定输入宿主变量，FETCH 基于位置在其 INTO 子句中指定输出宿主变量。一条语句仅需要分析一次。它然后可以多次执行。



注意：

所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

示例：

```
query_string : string(1..100)
get(query_string);
EXEC SQL prepare FRED from :query_string;
EXEC SQL execute FRED;
```

PRESENTNNV

参阅： RPESENTV。

格式：

```
PRESENTNNV ( cell_reference, expr1, expr2 )
```

描述： PRESENTNNV 只与行间计算有关。它只能用在 SELECT 语句的 MODEL 子句中，然后放在模型规则的右边。MODEL 子句执行前，如果 cell_reference 存在且非空，则返回 expr1；否则返回 expr2。

PRESENTV

参阅： PRESENTNNV。

格式：

```
PRESENTV ( cell_reference, expr1, expr2 )
```

描述： PRESENTV 只与行间计算有关。它只能用在 SELECT 语句的 MODEL 子句中，然后放在模型规则的右边。MODEL 子句执行前，如果 cell_reference 存在(但是可能为空)，则返回 expr1；否则返回 expr2。

PREVIOUS

参阅： ITERATION_NUMBER。

格式：

```
PREVIOUS ( cell_reference )
```

描述： PREVIOUS 只与行间计算有关。它只能用在 SELECT 语句的 MODEL 子句中，然后放在 MODEL RULES 子句的 ITERATE…[UNTIL]子句中。它在每次迭代的开始处返回 cell_reference 的值。

PRINT

参阅： VARIABLE、第 32 章。

格式：

```
PRI[NT] [variable…]
```

描述： PRINT 显示指定变量(通过 VARIABLE 命令创建)的当前值。可以在一条命令中输出多个变量的当前值。

PRIOR

请参阅 CONNECT BY。

PRIVILEGE

权限是赋予某个 Oracle 用户执行某些动作的许可。在 Oracle 中，如果没有相应的权限，

用户就不能执行任何动作。

有两种权限，即系统权限和对象权限。系统权限把许可扩展为可执行各种数据定义和数据控制命令，如 CREATE TABLE 和 ALTER USER，或者甚至登录数据库。对象权限则把许可扩展为可在特定的数据库对象上进行操作。

Oracle 中的系统权限包括用于各种 CREATE、ALTER 和 DROP 命令的 CREATE、ALTER 和 DROP 权限。其中具有关键字 ANY 的权限表示用户可以在任何已经授予相应权限的模式(不仅限于自己的模式)上使用这些权限。表 A-20 中的标准权限给出了可以执行的命令，不需要进一步解释。对于一些不太直观的权限，这里做了解释。

系统权限如表 A-20 所示。

表 A-20 系统权限

权 限	允许进行的活动
顾问架构(ADVISOR FRAMEWORK)	
ADVISOR	通过包含 DBMS_ADVISOR 和 DBMS_SQLTUNE 的 PL/SQL 程序包来访问顾问架构
ADMINISTER SQL TUNING SET	通过 DBMS_SQLTUNE 程序包来创建、选择、加载和删除被授权者所拥有的 SQL 调整集合
ADMINISTER ANY SQL TUNING SET	通过 DBMS_SQLTUNE 程序包来创建、选择、加载和删除任何用户所拥有的 SQL 调整集合
CREATE ANY SQL PROFILE	接受调整顾问推荐的 SQL 配置文件
DROP ANY SQL PROFILE	删除已有的 SQL 配置文件
ALTER ANY SQL PROFILE	更改已有的 SQL 配置文件的属性
群集(CLUSTER)	
CREATE CLUSTER	在被授权者的模式中创建群集
CREATE ANY CLUSTER	在任意模式中创建群集。其作用类似于 CREATE ANY TABLE
ALTER ANY CLUSTER	更改任意模式中的群集
DROP ANY CLUSTER	删除任意模式中的群集
语境(CONTEXT)	
CREATE ANY CONTEXT	创建任意语境名称空间
DROP ANY CONTEXT	撤消任意语境名称空间
数据库(DATABASE)	
ALTER DATABASE	更改数据库
ALTER SYSTEM	发布 ALTER SYSTEM 语句
AUDIT SYSTEM	发布 AUDIT 语句
数据库链接(DATABASE LINK)	
CREATE DATABASE LINK	在被授权者的模式中创建私有数据库链接
CREATE PUBLIC DATABASE LINK	创建公有数据库链接
DROP PUBLIC DATABASE LINK	删除公有数据库链接

(续表)

权　　限	允许进行的活动
调试(DEBUGGING)	
DEBUG CONNECT SESSION	将当前会话连接到调试器中
DEBUG ANY PROCEDURE	调试任意数据库对象中的所有 PL/SQL 和 Java 代码。显示应用程序执行的所有 SQL 语句的信息
字典(DICTIONARY)	
ANALYZE ANY DICTIONARY	分析任何数据字典对象
维度(DIMENSION)	
CREATE DIMENSION	在被授权者的模式中创建维度
CREATE ANY DIMENSION	在任意的模式中创建维度
ALTER ANY DIMENSION	更改任意模式中的维度
DROP ANY DIMENSION	删除任意模式中的维度
目录(DIRECTORY)	
CREATE ANY DIRECTORY	创建目录数据库对象
闪回数据归档(FLASHBACK DATA ARCHIVE)	
FLASHBACK ARCHIVE ADMINISTER	创建、更改或删除任意闪回数据归档
索引类型(INDEXTYPE)	
CREATE INDEXTYPE	在被授权者的模式中创建索引类型
CREATE ANY INDEXTYPE	在任意的模式中创建索引类型
ALTER ANY INDEXTYPE	修改任意模式中的索引类型
DROP ANY INDEXTYPE	删除任意模式中的索引类型
EXECUTE ANY INDEXTYPE	引用任意模式中的索引类型
索引(INDEX)	
CREATE ANY INDEX	在任意模式中创建域索引或者在任意模式中的任何表上创建索引
ALTER ANY INDEX	更改任意模式中的索引
DROP ANY INDEX	删除任意模式中的索引
作业调度程序对象(JOB SCHEDULER OBJECT)	
CREATE JOB	在被授权者的模式中创建作业、调度程序或程序(利用 DBMS_SCHEDULER)
CREATE ANY JOB	在任意的模式中创建、更改或删除作业、调度程序或程序
EXECUTE ANY PROGRAM	使用被授权者的模式中的作业中的任何程序
EXECUTE ANY CLASS	指定被授权者的模式中的作业中的任何作业类
MANAGE SCHEDULER	创建、更改或删除任何作业类、窗口或窗口组

(续表)

权 限	允许进行的活动
库(LIBRARY)	
CREATE LIBRARY	在被授权者的模式中创建外部过程/函数库
CREATE ANY LIBRARY	在任意模式中创建外部过程/函数库
DROP ANY LIBRARY	删除任意模式中的外部过程/函数库
物化视图(MATERIALIZED VIEW)	
CREATE MATERIALIZED VIEW	在被授权者的模式中创建物化视图
CREATE ANY MATERIALIZED VIEW	在任意模式中创建物化视图
ALTER ANY MATERIALIZED VIEW	更改任意模式中的物化视图
DROP ANY MATERIALIZED VIEW	删除任意模式中的物化视图
QUERY REWRITE	该权限已经废除
GLOBAL QUERY REWRITE	当物化视图引用任意模式中的表或视图时，允许利用物化视图进行重写
ON COMMIT REFRESH	在数据库中任意表上创建一个提交时刷新的物化视图 将数据库中任意表上的在请求时刷新的物化视图更改为提交时刷新的物化视图
FLASHBACK ANY TABLE	在任意模式中的任何表、视图或物化视图上发布 FALSHBACK QUERY。
挖掘模型(MINING MODEL)	
CREATE MINING MODEL	创建挖掘模型
CREATE ANY MINING MODEL	在任意模式中创建挖掘模型
ALTER ANY MINING MODEL	更改任意挖掘模型
DROP ANY MINING MODEL	删除任意模式中的任何挖掘模型
SELECT ANY MINING MODEL	计分或查看任意模式中的任何模型
COMMENT ANY MINING MODEL	在任意模式中的任何模型上创建注释
OLAP 立方体(OLAP CUBE)	
CREATE CUBE	创建 OLAP 立方体
CREATE ANY CUBE	在任意模式中创建 OLAP 立方体
ALTER ANY CUBE	更改任意模式中的 OLAP 立方体
DROP ANY CUBE	删除任何 OLAP 立方体
SELECT ANY CUBE	查询或查看任意模式中的任何 OLAP 立方体
UPDATE ANY CUBE	更新任意模式中的任何立方体
OLAP 立方体度量文件夹(OLAP CUBE MEASURE FOLDER)	
CREATE MEASURE FOLDER	创建 OLAP 度量文件夹
CREATE ANY MEASURE FOLDER	在任意模式中创建 OLAP 度量文件夹
DELETE ANY MEASURE FOLDER	从任意模式中的 OLAP 度量文件夹中删除
DROP ANY MEASURE FOLDER	删除任意模式中的任何度量文件夹

(续表)

权　　限	允许进行的活动
INSERT ANY MEASURE FOLDER	将一个度量插入任意模式中的任何度量文件夹中
OLAP 立方体维度(OLAP CUBE MEASURE FOLDER)	
CREATE CUBE DIMENSION	创建 OLAP 立方体维度
CREATE ANY CUBE DIMENSION	在任意模式中创建 OLAP 立方体维度
ALTER ANY CUBE DIMENSION	更改任意模式中的 OLAP 立方体维度
DELETE ANY CUBE DIMENSION	从任意模式中的 OLAP 立方体维度中删除
DROP ANY CUBE DIMENSION	删除任意模式中的 OLAP 立方体维度
INSERT ANY CUBE DIMENSION	插入到任意模式中的 OLAP 立方体维度
SELECT ANY CUBE DIMENSION	查看或查询 OLAP 立方体维度
UPDATE ANY CUBE DIMENSION	更新任意模式中的 OLAP 立方体维度
OLAP 立方体构建过程(OLAP CUBE BUILD PROCESSE)	
CREATE CUBE BUILD PROCESS	创建 OLAP 立方体构建过程
CREATE ANY CUBE BUILD PROCESS	在任意模式中创建 OLAP 立方体构建过程
DROP ANY CUBE BUILD PROCESS	删除任意模式中的 OLAP 立方体构建过程
UPDATE ANY CUBE BUILD PROCESS	更新任意模式中的 OLAP 立方体构建过程
运算符(OPERATOR)	
CREATE OPERATOR	在被授权者的模式中创建运算符及其绑定
CREATE ANY OPERATOR	在任意的模式中创建运算符及其绑定
ALTER ANY OPERATOR	修改任意模式中的运算符
DROP ANY OPERATOR	删除任意模式中的运算符
概要(OUTLINE)	
CREATE ANY OUTLINE	创建公有概要，它可以用在使用该概要的任意模式中
ALTER ANY OUTLINE	修改大纲
DROP ANY OUTLINE	删除概要
过程(PROCEDURE)	
CREATE PROCEDURE	在被授权者的模式中创建存储过程、函数和程序包
CREATE ANY PROCEDURE	在任意的模式中创建存储过程、函数和程序包
ALTER ANY PROCEDURE	更改任意模式中的存储过程、函数和程序包
DROP ANY PROCEDURE	删除任意模式中的存储过程、函数和程序包
EXECUTE ANY PROCEDURE	执行过程或函数(独立的或打包的)。引用任意模式中的公有程序包变量
配置文件(PROFILE)	
CREATE PROFILE	创建配置文件
ALTER PROFILE	更改配置文件
DROP PROFILE	撤消配置文件

(续表)

权 限	允许进行的活动
角色(ROLE)	
CREATE ROLE	创建角色
ALTER ANY ROLE	更改数据库中的任意角色
DROP ANY ROLE	删除角色
GRANT ANY ROLE	授权数据库中的任意角色
回滚段(ROLLBACK SEGMENT)	
CREATE ROLLBACK SEGMENT	创建回滚段
ALTER ROLLBACK SEGMENT	更改回滚段
DROP ROLLBACK SEGMENT	删除回滚段
序列(SEQUENCE)	
CREATE SEQUENCE	在被授权者的模式中创建序列
CREATE ANY SEQUENCE	在任意的模式中创建序列
ALTER ANY SEQUENCE	更改数据库中的任意序列
DROP ANY SEQUENCE	删除任意模式中的序列
SELECT ANY SEQUENCE	引用任意模式中的序列
会话(SESSION)	
CREATE SESSION	连接到数据库
ALTER RESOURCE COST	设置会话源的成本
ALTER SESSION	发布 ALTER SESSION 语句
RESTRICTED SESSION	在利用 SQL*Plus 的 STARTUP RESTRICT 语句启动实例后登录
同义词(SYNONYM)	
CREATE SYNONYM	在被授权者的模式中创建同义词
CREATE ANY SYNONYM	在任意的模式中创建私有同义词
CREATE PUBLIC SYNONYM	创建公有同义词
DROP ANY SYNONYM	删除任意模式中的私有同义词
DROP PUBLIC SYNONYM	删除公有同义词
表(TABLE)	
CREATE TABLE	在被授权者的模式中创建表
CREATE ANY TABLE	在任意的模式中创建表。包含表的模式的所有者必须在表空间上具有空间限额来包含该表
ALTER ANY TABLE	更改任意模式中的任意表或视图
BACKUP ANY TABLE	利用 Export 实用程序从其他用户的模式中递增地导出对象
DELETE ANY TABLE	从任意模式中的表、表分区或视图中删除行
DROP ANY TABLE	删除或截断任意模式中的表或表分区
INSERT ANY TABLE	向任意模式中的表和视图中插入行
LOCK ANY TABLE	锁定任意模式中的表和视图
SELECT ANY TABLE	查询任意模式中的表、视图或物化视图

(续表)

权　　限	允许进行的活动
FLASHBACK ANY TABLE	在任意模式中的任意表、视图或物化视图上发布一个闪回查询
UPDATE ANY TABLE	更新任意模式中表和视图中的行
表空间(TABLESPACE)	
CREATE TABLESPACE	创建表空间
ALTER TABLESPACE	更改表空间
DROP TABLESPACE	删除表空间
MANAGE TABLESPACE	使表空间脱机和联机，开始和结束表空间的备份
UNLIMITED TABLESPACE	使用无限制的表空间量。此权限将重写所分配的任何限额。如果从某个用户处收回此权限，则该用户的模式对象仍然保留，但是除非授予该用户特定的表空间限额，否则拒绝进一步的表空间分配。此系统权限不能授予角色
触发器(TRIGGER)	
CREATE TRIGGER	在被授权者的模式中创建数据库触发器
CREATE ANY TRIGGER	在任意模式中创建数据库触发器
ALTER ANY TRIGGER	启用、禁用或编译任意模式中的数据库触发器
DROP ANY TRIGGER	删除任意模式中的数据库触发器
ADMINISTER DATABASE TRIGGER	在 DATABASE 上创建触发器(也必须具有 CREATE TRIGGER 或 CREATE ANY TRIGGER 权限)
类型(TYPE)	
CREATE TYPE	在被授权者的模式中创建对象类型或对象类型体
CREATE ANY TYPE	在任意模式中创建对象类型或对象类型体
ALTER ANY TYPE	更改任意模式中的对象类型
DROP ANY TYPE	删除任意模式中的对象类型和对象类型体
EXECUTE ANY TYPE	使用和引用任意模式中的对象类型和集合类型，如果对特定用户进行了授权，则该用户可以调用任意模式中的对象类型的方法。如果将 EXECUTE ANY TYPE 授予某个角色，则拥有该启动角色的用户还是不能调用任意模式中的对象类型方法
UNDER ANY TYPE	在非最终对象类型之下创建子类型
用户(USER)	
CREATE USER	创建用户。该权限还允许创建者完成以下操作： 分配任意表空间上的限额。 设置默认和临时表空间。 指定一个配置文件作为 CREATE USER 语句的一部分
ALTER USER	更改任意用户。该权限还允许被授权者完成以下操作： 更改其他用户的口令或身份验证方法 分派任意表空间上的限额 设置默认和临时表空间 指定配置文件和默认角色

(续表)

权 限	允许进行的活动
DROP USER	删除用户
视图(VIEW)	
CREATE VIEW	在被授权者的模式中创建视图
CREATE ANY VIEW	在任意模式中创建视图
DROP ANY VIEW	删除任意模式中的视图
UNDER ANY VIEW	在任意对象视图之下创建子视图
FLASHBACK ANY TABLE	在任意模式中的任何表、视图或物化视图上发布一个 SQL 闪回查询。执行 DBMS_FLASHBACK 过程不需要此权限
MERGE ANY VIEW	如果用户已被授予 MERGE ANY VIEW 权限，则对于此用户发布的任何查询，优化程序使用视图合并就可以在不执行检查的情况下提高查询性能，否则要执行检查才能确保视图合并未违反视图

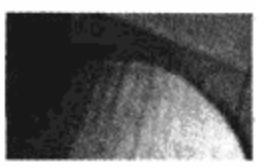
(续表)

权　限	允许进行的活动
SYSDBA	执行 STARTUP 和 SHUTDOWN 操作 ALTER DATABASE：打开、安装、备份或更改字符集 CREATE DATABASE ARCHIVELOG 和 RECOVER CREATE SPFILE 包括 RESTRICTED SESSION 权限
SYSOPER	执行 STARTUP 和 SHUTDOWN 操作 ALTER DATABASE OPEN | MOUNT | BACKUP ARCHIVELOG 和 RECOVER CREATE SPFILE 包括 RESTRICTED SESSION 权限

系统角色如表 A-21 所示。

表 A-21　系统角色

系 统 角 色	用　途
CONNECT，RESOURCE 和 DBA	为与 Oracle 数据库软件以前的版本兼容而准备
DELETE_CATALOG_ROLE EXECUTE_CATALOG_ROLE SELECT_CATALOG_ROLE	这些角色为访问数据字典视图和程序包而准备
EXP_FULL_DATABASE IMP_FULL_DATABASE	这些角色为方便使用导入和导出实用程序而准备
AQ_USER_ROLE AQ_ADMINISTRATOR_ROLE	Oracle Advanced Queueing(Oracle 高级队列)需要用到这些角色
SNMPAGENT	该角色由 Enterprise Manager Intelligent Agent (企业管理人员智能代理)使用
RECOVERY_CATALOG_OWNER	恢复目录模式所有者的创建过程需要该角色
HS_ADMIN_ROLE	支持异类服务(Heterogeneous Service)需要该角色
SCHEDULER_ADMIN	该角色允许被授权者执行 DBMS_SCHEDULER 程序包的过程



注意：
对于外部表，有效的权限只有 CREATE ANY TABLE、ALTER ANY TABLE、DROP ANY TABLE 和 SELECT ANY TABLE。

对象权限只适用于某些类型的对象。表 A-22 给出了相应的关系和可授予的对象权限。

表 A-22 相应的关系和可授予的对象权限

对象权限	表	视图	序列	过程、函数、程序包	物化视图	目录	库	用户定义类型	运算符	索引类型
ALTER	X		X							
DELETE	X	X			X[b]					
EXECUTE				X[c]			X[c]	X[c]	X[c]	X[c]
DEBUG	X	X		X				X		
FLASHBACK	X	X			X					
INDEX	X									
INSERT	X	X			X[b]					
ON COMMIT REFRESH					X					
QUERY REWRITE	X									
READ						X				
REFERENCES	X	X								
SELECT	X	X	X		X					
UNDER		X						X		
UPDATE	X	X			X[b]					
WRITE						X				

注：[a] 为了授予对象权限，Oracle 把 Java 类、源或资源看作过程。

[b] DELETE、INSERT 和 UPDATE 权限仅能授予可更新的物化视图。

[c] 利用 DBMS_SCHEDULER 程序包创建作业调度程序对象。这些对象一经创建，就可以在作业调度程序类和程序上授予 EXECUTE 对象权限。可以在作业调度程序作业、程序和调度程序上授予 ALTER 权限。

对于 FLASHBACK ARCHIVE、MINING MODEL 和 OLAP 选项，可用的权限如表 A-23 所示。

表 A-23

FLASHBACK DATA ARCHIVE PRIVILEGE	下面的闪回数据归档权限授权闪回数据归档上的操作
FLASHBACK ARCHIVE	用来启用或禁用表的历史跟踪
MINING MODEL PRIVILEGES	下面的挖掘模型权限授权挖掘模型上的操作。对于用户自己模式中的模型，不需要这些权限
ALTER	用来使用可应用的 DBMS_DATA_MINING 过程来更改挖掘模型名或相关的成本矩阵
SELECT	用来计分或查看挖掘模型。计分使用 SQL 函数的 PREDICTION 系列或者使用 DBMS_DATA_MINING.APPLY 过程。查看模型使用 DBMS_DATA_MINING.GET_MODEL_DETAILS_* 过程

(续表)

OLAP PRIVILEGES	如果正在使用带 OLAP 选项的 Oracle Database，则下面的对象权限有效
INSERT	用来将成员插入OLAP立方体维度中或者将度量插入度量文件夹中
ALTER	用来更改 OLAP 立方体维度或立方体的定义
DELETE	用来从OLAP立方体维度中删除成员或者从度量文件夹中删除度量
SELECT	用来查看或查询 OLAP 立方体或立方体维度
UPDATE	用来更新 OLAP 立方体的测量值或立方体维度的属性值

PROMPT(SQL*Plus)

参阅： ACCEPT。

格式：

```
PRO[MPT] [text]
```

描述： PROMPT 在用户的屏幕上显示 text。如果没有指定 text，则显示一个空行。要显示一个变量，请参阅 PRINT。

PSEUDO-COLUMNS

描述： 伪列是被选择时产生一个值的“列”，但它不是表的一个实际的列。如表 A-24 所示，下面是 Oracle 中当前的伪列：

表 A-24　Oracle 中当前的伪列

伪　列	返 回 的 值
COLUMN_VALUE	当引用不带 COLUMNS 子句的一个 XMLTable 结构时，或者当使用 TABLE 函数引用标量嵌套表类型时，数据库返回只包含一列的一个虚拟表。此伪列的名称是 COLUMN_VALUE
CONNECT_BY_ISCYCLE	如果当前行有一个同时也是父行先的子集，则等于 1。否则等于 0
CONNECT_BY_ISLEAF	如果当前行是由 CONNECT BY 条件定义的树的一个叶子，则等于 1。否则等于 0
Level	根节点等于 1，根节点的子节点等于 2，依此类推。根据它能知道遍历的树的深度
sequence.CURRVAL	此序列名的当前值
sequence.NEXTVAL	此序列名的下一个值，而且对序列进行增量
VERSIONS_STARTTIME	对于闪回版本查询，返回查询所返回的行的第一个版本的时间戳
VERSIONS_STARTSCN	对于闪回版本查询，返回查询所返回的行的第一个版本的 SCN
VERSIONS_ENDTIME	对于闪回版本查询，返回查询所返回的行的最后一个版本的时间戳
VERSIONS_ENDSCN	对于闪回版本查询，返回查询所返回的行的最后一个版本的 SCN

(续表)

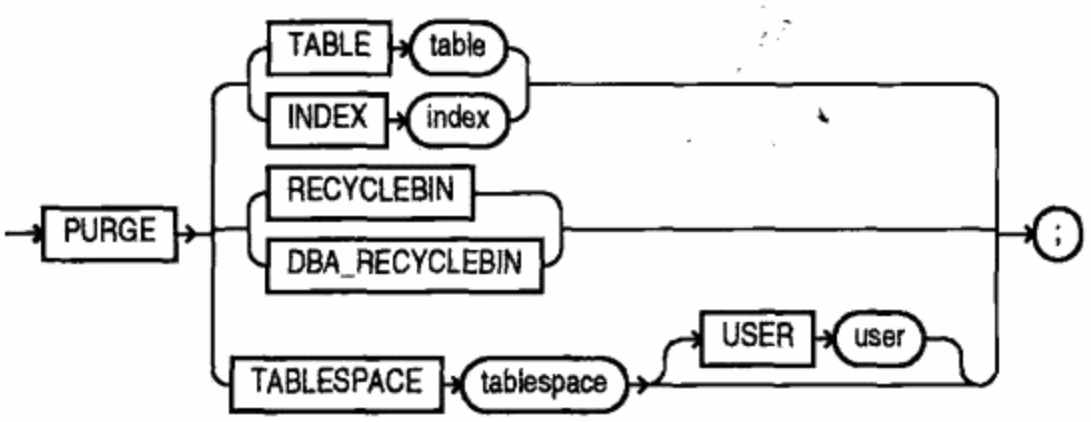
伪 列	返 回 的 值
VERSIONS_XID	对于闪回版本查询，对每一行的每个版本，返回创建该行版本的事务的事务ID
VERSIONS_OPERATION	对于闪回版本查询，对每一行的每个版本，对引起行版本的操作返回一个字符：I(插入)、U(更新)或 D(删除)
OBJECT_ID	返回对象表或视图的列的对象标识符
OBJECT_VALUE	为对象表、XMLType 表、对象视图或 XMLType 视图的列返回其系统产生的名称
ORA_ROWSCN	返回对行最近一次修改的保守上限 SCN，它可以近似说明该行最后被更新的时间(应该为更细粒度的审计创建行级依赖)关系
ROWID	返回一行的行标识符。在 UPDATE…WHERE 和 SELECT…FOR UPDATE 中使用 ROWID。该被授权者只有某一行被更新
ROWNUM	返回从表中选择时返回的行的序列号。第一行的 ROWNUM 为 1，第二行为 2 等
XMLDATA	允许访问基本的 XMLType LOB 数据或对象关系列

PURGE

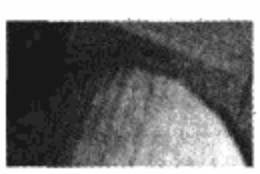
参阅：第 51 章。

格式：

purge::=



描述：PURGE 从回收站中删除一个表或索引并释放与该对象关联的所有空间，或者删除整个回收站，或者从回收站中删除一个已经删除的表空间的部分或全部。



注意：
PURGE 命令不能回滚。对象被清除(purge)后不能恢复。

要查看回收站的内容，可以查询 RECYCLEBIN 数据字典视图。

可以利用 TABLE 或 INDEX 关键字来指定希望清除的对象名。指定 RECYCLEBIN 将清除用户回收站中的所有对象。DBA_RECYCLEBIN 清空所有用户的回收站。TABLESPACE 从回收站中清除指定表空间内驻留的所有对象。USER *user* 为特定用户清空表空间中的空间。

示例：下面的命令可以清除一个表。

```
PURGE TABLE bookshelf;
```

下面的命令可以清空当前用户的回收站。

```
PURGE RECYCLEBIN;
```

QUERY OPERATORS

表 A-25 是 SQL 的 Oracle 版本中所有当前查询运算符的按字母顺序的列表。其中每个运算符都在本附录的其他地方按照各自的名字列出，并给出正确的格式和用法。请参阅第 13 章。

表 A-25　SQL 的 Oracle 版本中当前的查询运算符

运 算 符	用　　途
INTERSECT	从两个查询中返回所有匹配的不同的行
MINUS	返回出现在第一个查询而没有出现在第二个查询中的所有不同的行
UNION	从两个查询中返回所有不同的行
UNION ALL	从两个查询中返回所有行

QUIT

参阅：COMMIT、DISCONNECT、EXIT、SET AUTOCOMMIT、START。

格式：

```
QUIT
```

描述：QUIT 结束 SQL*Plus 会话并使用户返回操作系统、调用程序或菜单。

RAISE

参阅：EXCEPTION 和 EXCEPTION_INIT。

格式：

```
RAISE [exception]
```

描述：RAISE 基于正在测试的条件命名希望触发的异常标志。异常必须是一个已经显式 DECLARE(声明)的异常，或者是一个内部系统异常，如 DIVIDE_BY_ZERO(完整的列表请参阅 EXCEPTION)。

RAISE 语句使控制转向当前块的 EXCEPTION 部分，在这个部分中必须进行测试以确定触发了哪个异常。如果没有给出 EXCEPTION 部分，则控制转向封闭块的最近的 EXCEPTION 部分(主要是通过块的嵌套进行返回)。如果找不到处理此异常的逻辑，则控制从 PL/SQL 传递给主调程序或环境，同时附带一个未处理的异常错误(使用 OTHERS 可以避免这种情况。请参阅 EXCEPTION)。

不包含显式命名的异常的RAISE只能用在一种环境中，即用在一个EXCEPTION部分中，以便强制由封闭块的 EXCEPTION 部分，而不是当前 EXCEPTION 部分来处理当前异常。例如，如果出现 NOT_LOGGED_ON 错误，并且本地 EXCEPTION 显示如下内容：

```
when not_logged_on
    then raise;
```

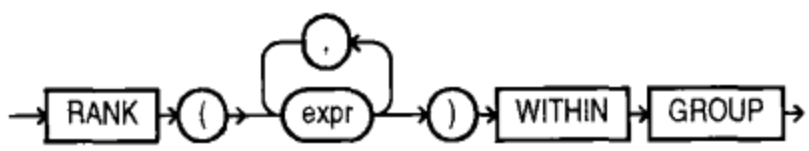
则它将把控制返回给发生 NOT_LOGGED_ON 错误的下一个封闭块的 EXCEPTION 部分；如果找不到这样的封闭块，就返回到相应的程序。该 EXCEPTION 块也可以测试 NOT_LOGGED_ON。这样的好处是对于某种类别的错误，特别是步骤很多的恢复、跟踪或错误记录等，可以设置每个嵌套块的 EXCEPTION 部分以便简单地将 EXCEPTION 返回到用于部署的某个点。

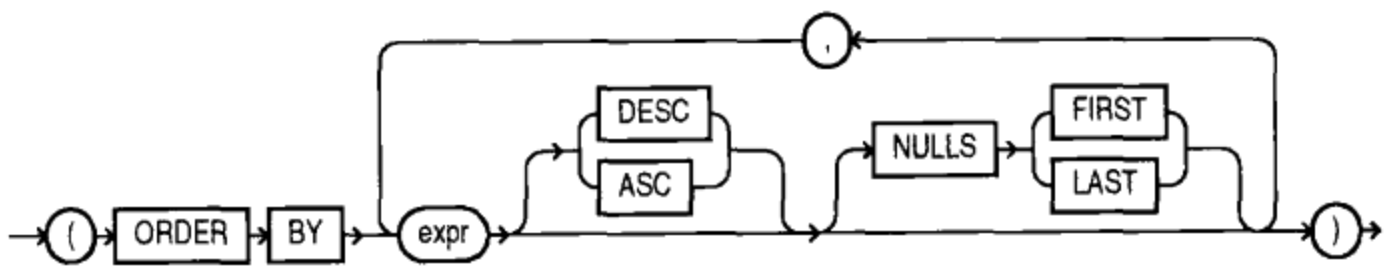
RANK

参阅：ANALYTIC FUNCTIONS。

聚集语法格式：

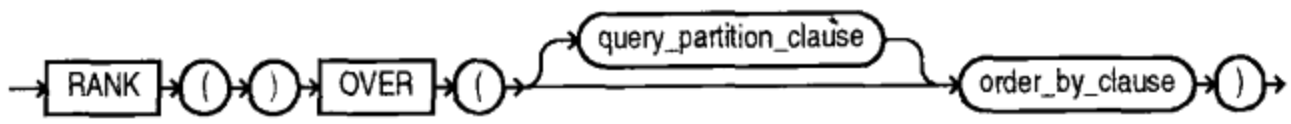
rank_aggregate::=





分析语法格式：

rank_analytic::=



描述：RANK 计算一组值中某个值的级别。返回类型为 NUMBER。级别标准值相等的行得到相同的级别。

RATIO_TO_REPORT

参阅：AGGREGATE FUNCTIONS。

格式：

```
RATIO_TO_REPORT ( expr ) OVER ( [ query_partition_clause] )
```

描述：RATIO_TO_REPORT 是一个分析函数。它计算一个值相对于一组值之和的比率。如果判定 *expr* 为 NULL，则比率报告(ratio-to-report)值也判定为 NULL。

RAW DATATYPE

RAW 列包含主机以任何格式存储的二进制数据。RAW 列对存储二进制(非字符)数据很

有用。

RAWTOHEX

参阅：HEXTORAW。

格式：

```
RAWTOHEX(binary_string)
```

描述：RAWTOHEX 将一个二进制数字串转化为一个十六进制数值字符串。

RAWTONHEX

参阅：RAWTOHEX。

格式：

```
RAWTONHEX ( raw )
```

描述：RAWTONHEX 将 *raw* 转换为一个等价的十六进制的 NVARCHAR2 字符值。

RECORD(PL/SQL)

参阅：TABLE(PL/SQL)、DATATYPES。

格式：

```
TYPE new_type IS RECORD
  (field {type | table.column%TYPE}[NOT NULL]
  [,field {type | table.column%TYPE}[NOT NULL]...]);
```

描述：RECORD 声明一种以后可以用来声明此种类型变量的新类型。记录的各个组成部分为字段，每个字段具有自己的数据类型。此数据类型可以是一种标准的 PL/SQL 数据类型(包括另一个 RECORD 但是不包括 TABLE)，或者是对指定表中特定列的类型的引用。每个字段还可能具有一个 NOT NULL 限定符，以指定该字段必须总是具有一个非 NULL 值。

可以利用点表示法来引用记录中的各个字段。

RECOVER

RECOVER 是 ALTER DATABASE 命令中的一个子句。

参阅：ALTER DATABASE、第 51 章。

格式：

```
RECOVER {general | managed | BEGIN BACKUP | END BACKUP}
```

其中的 GENERAL 子句语法如下：

```
 [AUTOMATIC] [FROM location]
{ {full_database_recovery | partial_database_recovery |LOGFILE filename}
[ {TEST | ALLOW integer CORRUPTION | parallel_clause }
[TEST | ALLOW integer CORRUPTION | parallel_clause]...]
```

```
|CONTINUE [DEFAULT]|CANCEL}
```

其中的 FULL DATABASE RECOVERY 子句语法如下：

```
[STANDBY] DATABASE
[ {UNTIL {CANCEL | TIME date | CHANGE integer} | USING BACKUP CONTROLFILE}
[UNTIL {CANCEL | TIME date | CHANGE integer} | USING BACKUP CONTROLFILE]...]
```

其中的 PARTIAL DATABASE RECOVERY 子句语法如下：

```
{TABLESPACE tablespace [, tablespace]... | DATAFILE datafilename [,
datafilenumber]...| STANDBY
{TABLESPACE tablespace [, tablespace]... | DATAFILE datafilename [,
datafilenumber]...}
UNTIL [CONSISTENT WITH] CONTROLFILE }
```

PARALLEL 子句语法如下：

```
{NOPARALLEL | PARALLEL [ integer]}
```

MANAGED 子句语法如下：

```
MANAGED STANDBY DATABASE recover_clause | cancel_clause | finish_clause
```

RECOVER 子句语法如下：

```
{{DISCONNECT [FROM SESSION]|{TIMEOUT integer | NOTIMEOUT}}
|{NODELAY | DEFAULT DELAY | DELAY integer} | NEXT integer
|{EXPIRE integer | NO EXPIRE} | parallel_clause
| USING CURRENT LOGFILE | UNTIL CHANGE integer
| THROUGH {[THREAD integer] SEQUENCE integer
    | ALL ARCHIVELOG | {ALL|LAST|NEXT} SWITCHOVER}}
[DISCONNECT [FROM SESSION] | {TIMEOUT integer | NOTIMEOUT}
 | {NODELAY|DEFAULT DELAY|DELAY integer}| NEXT integer
 | {EXPIRE integer | NO EXPIRE}| parallel_clause
 | USING CURRENT LOGFILE | UNTIL CHANGE integer
 | THROUGH {[THREAD integer]SEQUENCE integer
    | ALL ARCHIVELOG | {ALL|LAST|NEXT} SWITCHOVER}}
```

CANCEL 子句语法如下：

```
CANCEL [IMMEDIATE] [WAIT | NOWAIT]
```

FINISH 子句语法如下：

```
[DISCONNECT [FROM SESSION]][parallel_clause]
FINISH [SKIP [STANDBY LOGFILE]][WAIT|NOWAIT]
```

描述：与 ALTER DATABASE 命令的 RECOVER 子句一样，RECOVER 命令利用各种选项恢复一个数据库。AUTOMATIC 恢复在恢复过程中自动生成重做日志文件名。FROM 子句指定归档重做日志文件组的位置，而且必须是一个完全限定的文件名。默认值由初始化参数 LOG_ARCHIVE_DEST(或 LOG_ARCHIVE_DEST_1)设置。

DATABASE 选项恢复整个数据库，它是默认值。可以选择 UNTIL CANCEL 恢复数据库直到用 CANCEL 选项取消恢复为止。选择 UNTIL TIME 可以恢复到指定的日期和时间(以

YYYY-MM-DD:HH24:MI:SS 格式)。选择 UNTIL CHANGE 可以恢复到由一个整数指定的系统更改号(*system change number*，SCN，分配给每个事务的唯一编号)。选择 USING BACKUP CONTROLFILE 应用备份控制文件中的重做日志进行恢复。

可以通过 TEST 选项执行试验恢复，并且 CONTINUE 一个多实例恢复，即使该实例已经遭到破坏。DELAY 参数涉及在恢复过程中将日志应用到备份数据库；默认时，若通过 LOG_ARCHIVE_DEST_*n* 参数值来定义，它们将被延迟。

RECYCLE BIN

被删除的对象默认放入回收站，而不是立即被删除。回收站允许还原已删除的对象而不需要恢复整个数据库。可以通过 PURGE 命令从回收站中清除指定的对象。请参阅 PURGE。

要查看目前回收站中的对象，可以查询 RECYCLEBIN 数据字典视图或利用 SHOW RECYCLEBIN 命令。

REF

参阅：第 41 章。

格式：

```
REF ( correlation_variable )
```

描述：REF 是一个与对象表一起使用的引用函数；输入的相关变量是一个表别名。REF 返回该变量的对象实例。详细的示例请参阅第 41 章。

REFTOHEX

参阅：第 41 章。

格式：

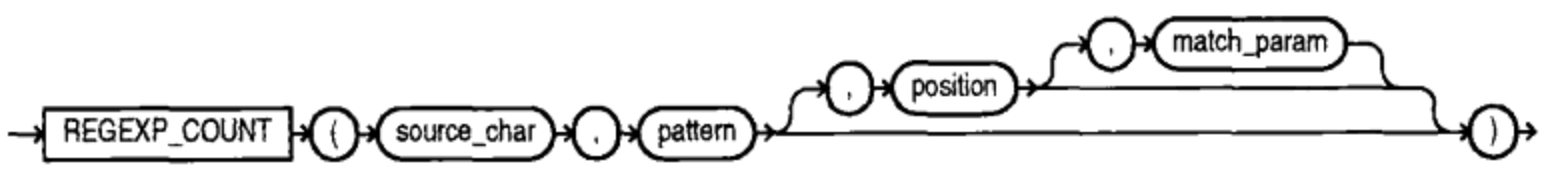
```
REFTOHEX ( expr )
```

描述：REFTOHEX 将参数 *expr* 转换为一个等价的十六进制的字符值。*expr* 必须返回一个 REF。

REGEXP_COUNT

参阅：REGEXP_INSTR、第 8 章。

格式：



描述：通过返回模式在源串中出现的次数，REGEXP_COUNT 对 REGEXP_INSTR 函数的功能进行了补充。该函数利用由输入字符集定义的字符来判定串。此函数返回一个整数，用来表示模式出现的次数。如果未找到匹配的串，则函数返回 0。

- source_char 是一个起搜索值作用的字符表达式。它通常是一个字符列，可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2、CLOB 和 NCLOB 等数据类型。
- pattern 是正则表达式。它通常是一个文本字面量，可以是 CHAR、VARCHAR2、NCHAR 和 NVARCHAR2 等数据类型。它最多可包含 512 个字节。如果 pattern 的数据类型与 source_char 的数据类型不相同，则 Oracle Database 将 pattern 转换为 source_char 的数据类型。
- REGEXP_COUNT 忽略模式中的子表达式圆括号。例如，模式“(123(45))”等价于“12345”。
- position 是一个正整数，表示 source_char 中 Oracle 应该开始搜索的那个字符。默认值是 1，意思是 Oracle 从 source_char 的第一个字符开始搜索。找到 pattern 的第一次出现的位置后，数据库从第一次出现的位置之后的第一个字符开始搜索第二次出现的位置。
- match_param 是一个文本字面量，允许更改此函数的默认匹配行为。可以为 match_param 指定下面的一个或多个值：
  - ‘i’指定区分大小写的匹配。
  - ‘c’指定不区分大小写的匹配。
  - ‘n’允许句点(.)匹配换行符，这里句点是匹配任何字符的字符。
  - ‘m’将源串视为多行。Oracle 将“^”和“$”分别视为源串中任何地方的任何行的开始和结束，而不仅仅是整个源串的开始和结束。如果忽略此参数，则 Oracle 将源串视为一行。
  - ‘x’忽略空格字符。默认情况下，空格字符与它们本身相匹配。

如果指定了多个互相矛盾的值，则 Oracle 使用最后那个值。例如，如果指定了‘ic’，则 Oracle 使用区分大小写的匹配。如果指定的字符超出前面所列的范围，则 Oracle 返回一个错误。

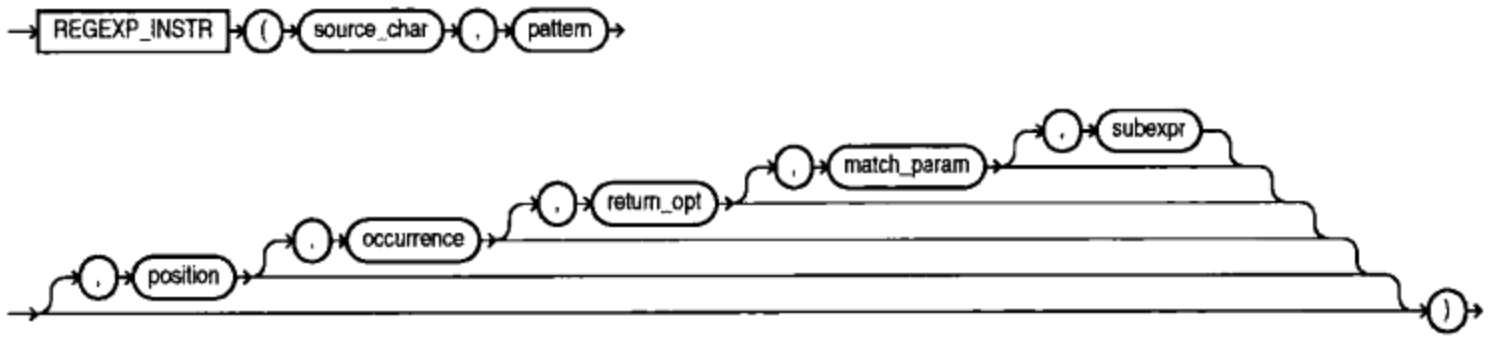
如果省略 match_param，那么：

- 默认的区分大小写由 NLS_SORT 参数的值来决定。
- 句点(.)不匹配换行符。
- 源串被视为一行。

REGEXP_INSTR

参阅： INSTR、第 8 章。

格式：



描述： 通过允许在串中搜索正则表达式模式，REGEXP_INSTR 扩展了 INSTR 函数的功

能。该函数利用输入字符集定义的字符来判定串。它根据 return_option 参数的值返回一个整数，指出匹配子串开始或结束的位置。如果没有发现匹配的子串，则函数返回 0。

REGEXP 函数的示例请参阅第 8 章。

在 Oracle Database 11g 中，可以将被判定的正则表达式的特定子串作为目标。

REGEXP_LIKE

参阅： LIKE、第 8 章。

格式：

```
REGEXP_LIKE(source_string, pattern [match_parameter])
```

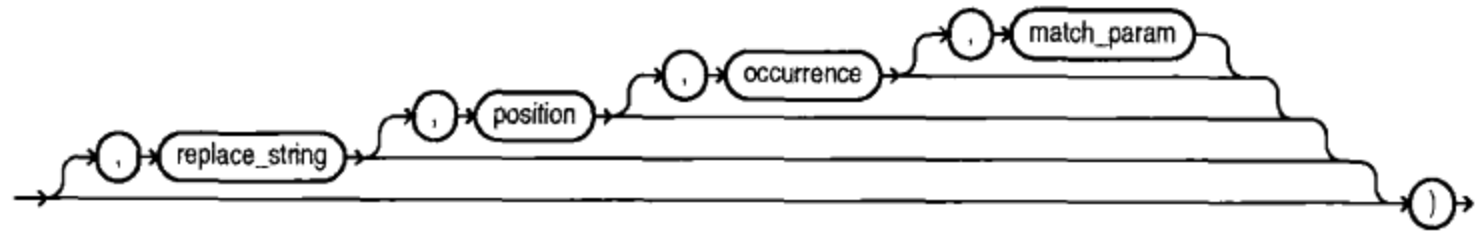
描述： 通过允许在串中搜索正则表达式模式，REGEXP_LIKE 扩展了 LIKE 运算符的功能。该函数利用输入字符集定义的字符来判定串。

REGEXP 函数的示例请参阅第 8 章。

REGEXP_REPLACE

参阅： REPLACE、第 8 章。

格式：



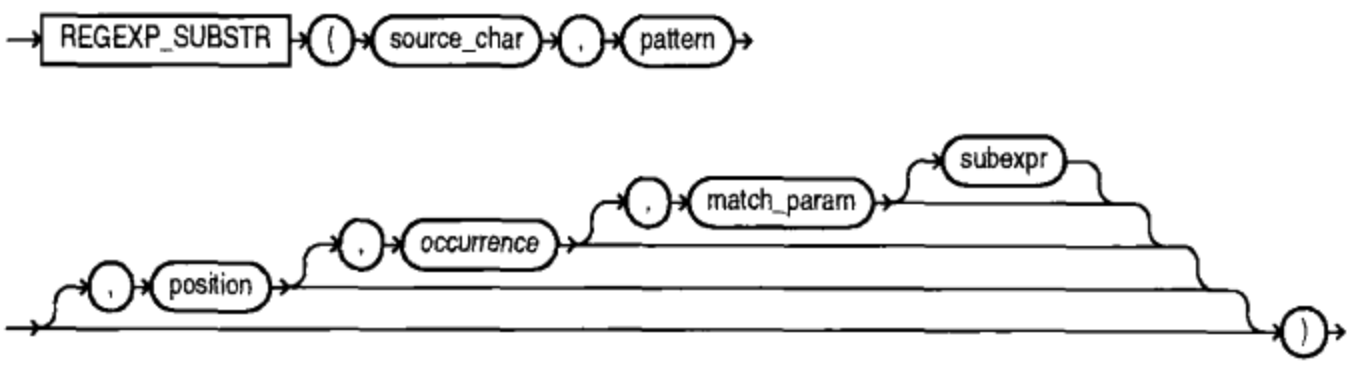
描述： 通过允许在串中搜索正则表达式模式，REGEXP_REPLACE 扩展了 REPLACE 函数的功能。默认时，函数返回 source_string，其中每一次出现的正则表达式模式都用 replace_string 代替。返回的串与 source_string 在相同的字符集中。如果第一个参数不是一个 LOB，则函数返回 VARCHAR2；如果第一个参数是 LOB，则返回 CLOB。

REGEXP 函数的示例请参阅第 8 章。

REGEXP_SUBSTR

参阅： SUBSTR、第 8 章。

格式：



描述：通过允许在串中搜索正则表达式模式，REGEXP_SUBSTR 扩展了 SUBSTR 函数的功能。它与 REGEXP_INSTR 类似，所不同的是它返回子串本身而不是返回子串的位置。该函数返回的串与 source_string 在相同的字符集中，作为 VARCHAR2 或 CLOB 数据。

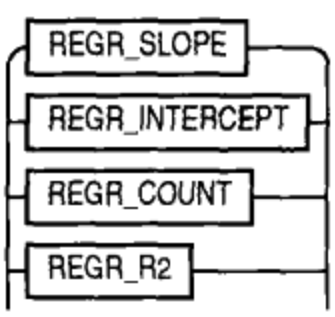
REGEXP 函数的示例请参阅第 8 章。

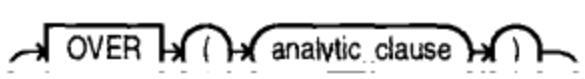
在 Oracle Database 11g 中，可以将被判定的正则表达式的特定子串作为目标。

REGR_(LINEAR REGRESSION) FUNCTIONS

参阅：AGGREGATE FUNCTIONS。

格式：





描述：REMARK 在一个启动文件中开始一个单行注释，通常是文档。它不作为命令解释。REMARK 不能出现在 SQL 语句中。

示例：

```
REM Reduce the default column width for this data field:
column ActionDate format a6 trunc
```

RENAME

参阅：COPY、CREATE SEQUENCE、CREATE SYNONYM、CREATE TABLE、CREATE VIEW。

格式：

```
RENAME old TO new;
```

描述：RENAME 把表、视图、序列或同义词的旧名字改为新名字。名字不用引号括起来，新名字既不能是保留字也不能是用户已有的对象名。

示例：下例将 BOOKSHELF 表更改为 BOOK 表：

```
rename BOOKSHELF to BOOK;
```

REPFOOTER(SQL*Plus)

参阅：ACCEPT、BTITLE、DEFINE、PARAMETERS、REPHEADER。

格式：

```
REPF[OOTER] [PAGE][ option [ text| variable]... | OFF|ON]
```

描述：REPFOOTER(REPort FOOTER)在报表的最后一页放置一个页脚(可能有多行)。OFF 和 ON 分别取消或还原文本的显示而不改变其内容。REPFOOTER 显示当前 REPFOOTER 选项及 text 或 variable。PAGE 在输出指定的报表页脚以前开始新的一页。

text 是希望给予此报表的一个页脚，variable 是一个用户定义的变量或一个系统维护的变量，其中包括 SQL.LNO(当前行号)、SQL.PNO(当前页码)、SQL.RELEASE(当前 Oracle 版本号)、SQL.SQLCODE(当前错误代码)和 SQL.USER(用户名)。

option 的有效值为：

- COL n 从当前行的左边空白直接跳到位置 n。
- S[KIP] n 输出 n 个空行。如果不指定 n，则输出一个空行。如果 n 为 0，则不输出空行且要输出的当前位置变为当前行的位置 1(页的最左边)。
- TAB n 向前跳到第 n 个位置(如果 n 为负，则向后跳)。
- LE[FT]、CE[NTER]和 R[IGHT]分别用于左对齐、居中和右对齐当前行上的数据。跟在这些命令后的任意文本或变量作为一组对齐，直到命令结束或者遇到 LEFT、CENTER、RIGHT 或 COL 为止。CENTER 和 RIGHT 使用 SET LINESIZE 命令设置的值来确定在何处放置文本或变量。

- BOLD 以粗体输出数据。SQL*Plus 通过在连续的 3 行上重复数据来表示在终端上粗体输出。
- FORMAT string 指定用来控制后面的文本或变量格式的格式模型，它遵循与 COLUMN 命令中的 FORMAT 相同的语法，如 FORMAT A12 或 FORMAT $999,990.99。每次出现 FORMAT，它将取代前一个正起作用的 FORMAT。如果没有指定格式模型，则使用 SET NUMFORMAT 设置的模型。如果没有设置 NUMFORMAT，则使用 SQL*Plus 的默认设置。

除非将 TO_CHAR 重新格式化的一个日期加载到一个变量中，否则将根据默认格式输出日期值。

REPHEADER(SQL*Plus)

参阅： ACCEPT、BTITLE、DEFINE、PARAMETERS、REPFOOTER。

格式：

```
REPH[EADER] [PAGE] [ option [ text| variable]... | OFF|ON]
```

描述： REPHEADER(REPort HEADER)把一个标题(可能有多行)放在报表的第一页上。OFF 和 ON 分别取消或还原文本的显示而不改变其内容。REPHEADER 显示当前 REPHEADER 选项及 text 或 variable。PAGE 在输出完报表标题后开始新的一页。

text 是希望给予此报表的标题，variable 是一个用户定义的变量或一个系统维护的变量，其中包括 SQL.LNO(当前行号)、SQL.PNO(当前页码)、SQL.RELEASE(当前 Oracle 版本号)、SQL.SQLCODE(当前错误代码)和 SQL.USER(用户名)。

option 的有效值为：

- COL n 从当前行的左边空白直接跳到位置 n。
- S[KIP] n 输出 n 个空行。如果不指定 n，则输出一个空行。如果 n 为 0，则不输出空行且要输出的当前位置变为当前行的位置 1(页的最左边)。
- TAB n 向前跳到第 n 个位置(如果 n 为负，则向后跳)。
- LE[FT]、CE[NTER]和 R[IGHT]分别用于左对齐、居中和右对齐当前行上的数据。跟在这些命令后的任意文本或变量作为一组对齐，直到命令结束或者遇到 LEFT、CENTER、RIGHT 或 COL 为止。CENTER 和 RIGHT 使用 SET LINESIZE 命令设置的值来确定在何处放置文本或变量。
- BOLD 以粗体输出数据。SQL*Plus 通过在连续的 3 行上重复数据来表示在终端上粗体输出。
- FORMAT string 指定用来控制后面的文本或变量格式的 FORMAT MODEL，它遵循与 COLUMN 命令中的 FORMAT 相同的语法，如 FORMAT A12 或 FORMAT $999,990.99。每次出现 FORMAT，它将取代前一个正起作用的 FORMAT。如果没有指定 FORMAT MODEL，则使用 SET NUMFORMAT 设置的模型。如果没有设置 NUMFORMAT，则使用 SQL*Plus 的默认设置。

除非将由 TO_CHAR 重新格式化的一个日期加载到一个变量中，否则将根据默认格式输

出日期值。

REPLACE

参阅：CHARACTER FUNCTIONS、TRANSLATE、第 7 章。

描述：

```
REPLACE(string, if, then)
```

描述：REPLACE 用 0 个或多个字符替换串中的一个或多个字符。if 为一个或多个字符。string 中出现的每个 if 都用 then 的内容来替换。

示例：

```
REPLACE('BOB','BO','TA') = TAB
```

保留字

参阅：OBJECT NAMES。

描述：保留字是对 SQL 具有特殊意义的词，因此不能用作对象名。将它与关键字作比较，关键字可以用作对象名，但是关键字将来也有可能成为保留字。并不是每个词在所有产品中都是保留字，下面的清单列出了何种产品保留哪些词。在下面的保留字中，具有*的项是 SQL 和 ANSI 中的保留字。

ACCESS	ADD *	ALL *	ALTER *	AND *
ANY *	AS *	ASC *	AUDIT	BETWEEN *
BY *	CHAR *	CHECK *	CLUSTER	COLUMN
COMMENT	COMPRESS	CONNECT *	CREATE *	CURRENT *
DATE *	DECIMAL *	DEFAULT *	DELETE *	DESC *
DISTINCT *	DROP *	ELSE *	EXCLUSIVE	EXISTS
FILE	FLOAT *	FOR *	FROM *	GRANT *
GROUP *	HAVING *	IDENTIFIED	IMMEDIATE *	IN *
INCREMENT	INDEX	INITIAL	INSERT *	INTEGER *
INTERSECT *	INTO *	IS *	LEVEL *	LIKE *
LOCK	LONG	MAXEXTENTS	MINUS	MLSLABEL
MODE	MODIFY	NOAUDIT	NOCOMPRESS	NOT *
NOWAIT	NULL *	NUMBER	OF *	OFFLINE
ON *	ONLINE	OPTION *	OR *	ORDER *
PCTFREE	PRIOR *	PRIVILEGES *	PUBLIC *	RAW
RENAME	RESOURCE	REVOKE *	ROW	ROWID
ROWNUM	ROWS *	SELECT *	SESSION *	SET *
SHARE	SIZE *	SMALLINT *	START	SUCCESSFUL
SYNONYM	SYSDATE	TABLE *	THEN *	TO *
TRIGGER	UID	UNION *	UNIQUE *	UPDATE *
USER *	VALIDATE	VALUES *	VARCHAR *	VARCHAR2
VIEW *	WHENEVER *	WHERE	WITH *	

表 A-26 列出了 PL/SQL 中的保留字：

表 A-26 PL/SQL 中的保留字

开始的字符	保 留 字
A	ALL, ALTER, AND, ANY, ARRAY, ARROW, AS, ASC, AT
B	BEGIN, BETWEEN, BY
C	CASE, CHECK, CLUSTERS, CLUSTER, COLAUTH, COLUMNS, COMPRESS,CONNECT, CRASH, CREATE, CURRENT
D	DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DROP
E	ELSE, END, EXCEPTION, EXCLUSIVE, EXISTS
F	FETCH, FORM, FOR, FROM
G	GOTO, GRANT, GROUP
H	HAVING
I	IDENTIFIED, IF, IN, INDEXES, INDEX, INSERT, INTERSECT, INTO, IS
L	LIKE, LOCK
M	MINUS, MODE
N	NOCOMPRESS, NOT, NOWAIT, NULL
O	OF, ON, OPTION, OR, ORDER, OVERLAPS
P	PRIOR, PROCEDURE, PUBLIC
R	RANGE, RECORD, RESOURCE, REVOKE
S	SELECT, SHARE, SIZE, SQL, START, SUBTYPE
T	TABAUTH, TABLE, THEN, TO, TYPE
U	UNION, UNIQUE, UPDATE, USE
V	VALUES, VIEW, VIEWS
W	WHEN, WHERE, WITH

表 A-27 列出了 PL/SQL 中的关键字：

表 A-27 PL/SQL 中的关键字

开始的字符	关 键 字
A	A, ADD, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG
B	BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE
C	C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARSETFORM,CHARSETID, CHARSET, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONVERT, COUNT, CURSOR, CUSTOMDATUM
D	DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DETERMINISTIC, DOUBLE, DURATION
E	ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXIT, EXTERNAL

(续表)

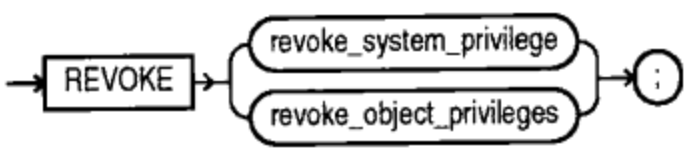
开始的字符	关 键 字
F	FINAL, FIXED, FLOAT, FORALL, FORCE, FUNCTION
G	GENERAL
H	HASH, HEAP, HIDDEN, HOUR
I	IMMEDIATE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT,INTERFACE, INTERVAL, INVALIDATE, ISOLATION
J	JAVA
L	LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP
M	MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISET
N	NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE
O	OBJECT, OCICOLL, OCIDATETIME, OCIDATE, OCIDURATION, OCIINTERVAL, OCILOBLOCATOR, OCINUMBER, OCIRAW, OCIREFCURSOR, OCIREF, OCIROWID, OCISTRING, OCITYPE, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVARY, OTHERS, OUT, OVERRIDING
P	PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIVATE
R	RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, RELIES_ON, REM, REMAINDER, RENAME, RESULT, RESULT_CACHE, RETURN, RETURNING, REVERSE, ROLLBACK, ROW
S	SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQLCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTABLE, SUBTYPE, SUM, SYNONYM
T	TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED, TYPE
U	UB1, UB2, UB4, UNDER, UNSIGNED, UNTRUSTED, USE, USING
V	VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID
W	WHILE, WORK, WRAPPED, WRITE
Y	YEAR
Z	ZONE

REVOKE

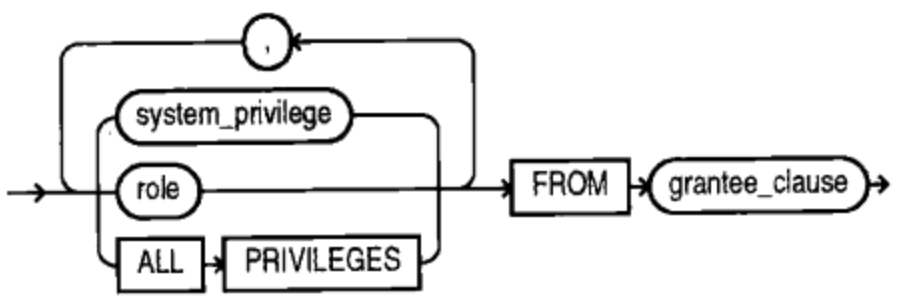
参阅：GRANT、PRIVILEGE、第 19 章。

格式：

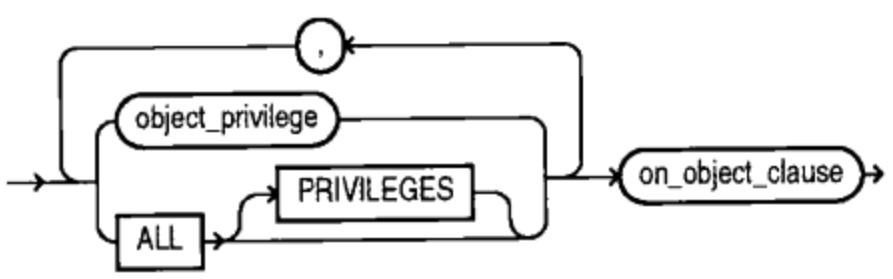
revoke::=

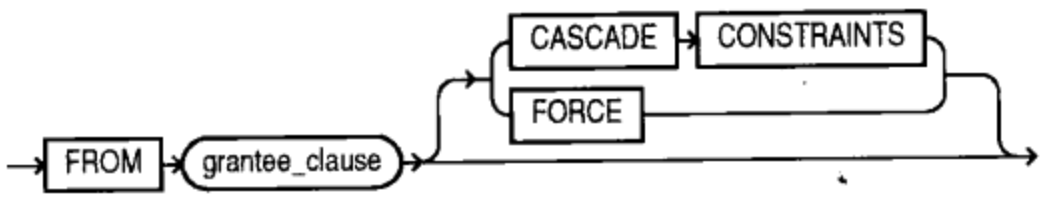


revoke_system_privileges::=

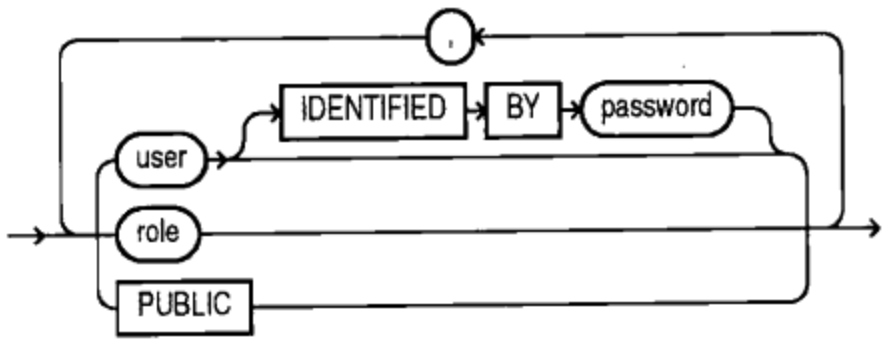


revoke_object_privileges::=

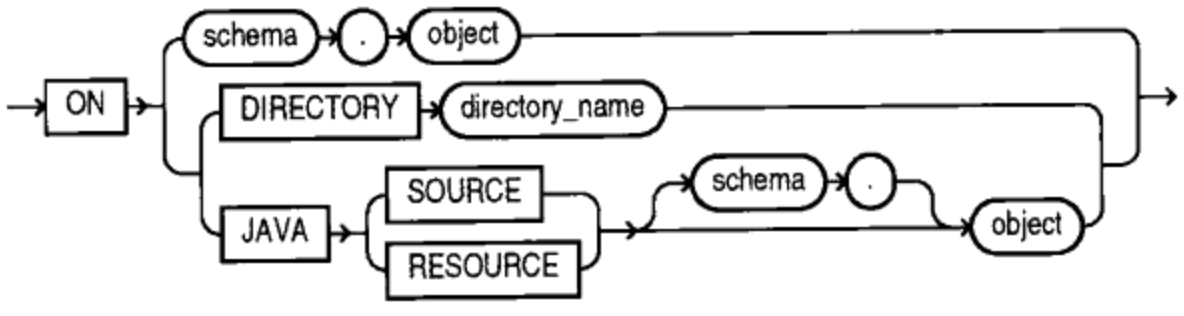




grantee_clause::=



on_object_clause::=



描述：REVOKE 命令取消用户的权限或角色，或者取消角色的权限。任何系统权限都可以取消。请参阅 PRIVILEGE。

如果取消一个用户的某个权限，此用户就不能再执行这个权限所允许的操作了。如果取消了一个角色的某个权限，那么除非其他角色给用户授权或直接给用户授权，否则授予此角色的用户将不能执行该权限所允许的操作。如果取消了 PUBLIC 一个权限，通过 PUBLIC 授权的用户就不能执行该权限所允许的操作。

如果取消某个用户的一个角色，该用户就不能再启用该角色(请参阅 ENABLE)，但是可以继续在当前会话中使用相应的权限。如果取消一个角色的了角色，授予该角色的用户就不能再启用这个角色，但是可以在当前会话中继续使用相应的权限。如果取消了 PUBLIC 的一个角色，通过 PUBLIC 授予此角色的用户就不能再启用此角色了。

对于对象访问来说，REVOKE 命令从某个用户或角色处取消特定对象上的对象权限。如果取消一个用户的权限，该用户就不能执行对象上的权限所允许的操作。如果取消一个角色的权限，则除非通过其他角色授予用户这些权限，或者直接授予用户这些权限，否则授予该角色的用户不能执行这些权限所允许的操作。如果取消 PUBLIC 的权限，则通过 PUBLIC 授予这些权限的用户不能执行相应的操作。

CASCADE CONSTRAINTS 子句删除由用户或授予相应角色的用户定义的任何参照完整性约束。这也适用于 REFERENCES 权限。

ROLLBACK(格式 1——SQL)

参阅：COMMIT、SET AUTOCOMMIT、第 15 章。

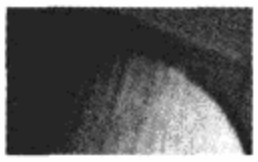
格式：

```
ROLLBACK [WORK] [ TO [SAVEPOINT] savepoint | FORCE 'text' ];
```

描述：ROLLBACK 回退(reverse)自最后一次提交或回滚以来对数据库中的表所做的修改，并且释放表上的任何锁。只要一个事务处理被中断，如由于执行错误、电源故障等原因而中断，就会发生自动回滚。ROLLBACK 不仅影响最后的 INSERT、UPDATE 或 DELETE 语句，而且还影响自最后的 COMMIT 以来出现的任何语句。这样就允许将工作块作为一个整体处理，只有在希望进行的所有修改都完成时才进行 COMMIT 操作。

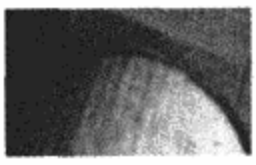
利用 SAVEPOINT、ROLLBACK 跳转到正在处理的事务序列中一个特定的已命名的位置——SAVEPOINT。它清除任何临时的 SAVEPOINT 并释放产生该 SAVEPOINT 之后出现的所有表锁或行锁。

利用 FORCE、ROLLBACK 可以手动地回滚一个由字面量文本标识的不确定事务，这个字面量是来自数据字典视图 DBA_2PC_PENDING 的一个局部的或全局的事务 ID。



注意：

如果 SET AUTOCOMMIT 为 ON，则每个 INSERT、UPDATE 或 DELETE 将立即和自动地向数据库提交修改。输入单词 ROLLBACK 将产生消息“ROLLBACK COMPLETE”，但是这并不代表什么，因为 ROLLBACK 仅回滚到最后一个 COMMIT。



注意：
所有 DDL 命令都引起 COMMIT。

ROLLBACK(格式 2——嵌入 SQL)

参阅：COMMIT、SAVEPOINT、SET TRANSACTION、*Programmer's Guide to the Oracle Precompilers*。

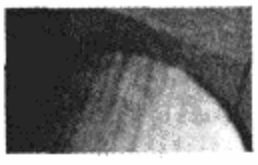
格式：

```
EXEC SQL [AT {database | :variable }]
ROLLBACK [WORK]
[TO [SAVEPOINT] savepoint] | [FORCE text] | [RELEASE] ]
```

描述：ROLLBACK 结束当前的事务处理，回退当前事务处理导致的任何修改并释放所持有的任意锁，但是并不影响宿主变量或程序流。database 指出 ROLLBACK 应该作用的数据库名。不给出 database 表示作用于用户的默认数据库。SAVEPOINT 允许回滚到以前声明的一个指定的 savepoint(请参阅 SAVEPOINT)。

FORCE 人工地回滚一个由字面量文本指定的不确定事务，这个字面量文本包含来自数据字典视图 DBA_2PC_PENDING 的事务 ID。WORK 完全是可选的，目的是增强可读性。

ROLLBACK 和 COMMIT 都具有 RELEASE 选项。应该在最后一个事务处理之后由它们中的一个来指定 RELEASE，否则在程序执行过程中设置的锁将阻碍其他用户。如果程序异常终止，则自动发生 ROLLBACK(具有 RELEASE)。



注意：
关于所有命令的完整列表请参阅 *Programmer's Guide to the Oracle Precompilers*。本附录不包含所有可能的预编译程序命令。

ROLLUP

参阅：GROUPING、CUBE、第 12 章。

格式：

```
GROUP BY ROLLUP(column1, column2)
```

描述：ROLLUP 生成一个查询内的行组的小计。请参阅第 12 章。

ROUND(格式 1——针对日期)

参阅：DATE FUNCTIONS、ROUND(NUMBER)、TRUNC、第 10 章。

格式：

```
ROUND (date, 'format')
```

描述：ROUND 根据 format 进行 date 的舍入。如果不给出 format，那么当时间为 12:00:00

P.M.(正午)或更迟时，date 舍入到下一天；如果时间在正午以前，则舍入为当前的日期。换句话说，就是舍入到最近的一天。所得到日期的时间设置为 12 A.M.，即当天一开始的时间。

舍入可用的格式如表 A-28 所示。

表 A-28　舍入可用的格式

格　式	含　义
CC、SCC	世纪(从 1950、2050 等的 1 月 1 号凌晨的午夜开始，向上舍入到下个世纪的 1 月 1 号)
SYEAR、SYYY、Y、YY、YYY、YYYY 和 YEAR	年(自 7 月 1 号凌晨的午夜开始向上舍入到下一年的 1 月 1 号)
IYYY、IYY、IY、I	ISO 年
Q	季度(自当季度第二个月的 16 号凌晨的午夜开始向上舍入，不管该月中的天数)
MONTH、MON、MM、RM	月(自 16 号凌晨的午夜开始向上舍入，不管该月的天数)
WW	周中与年的第一天相同的天
IW	周中与 ISO 年的第一天相同的天
W	周中与月的第一天相同的天
DDD、DD、J	自正午起向上舍入到下一天；与没有格式的 ROUND 相同
DAY、DY、D	周的第一天
HH、HH12、HH24	小时
MI	分钟

在进行 WW 和 W 舍入时，将被舍入的日期的时间与一个设置为 12 A.M.的日期(一天的开始)进行比较。ROUND 的结果为一个新日期，其时间也设置为 12 A.M.。

ROUND(格式 2——针对数字)

参阅：CEIL、FLOOR、NUMBER FUNCTIONS、ROUND(DATE)、TRUNC、第 9 章。

格式：

```
ROUND(value, precision)
```

描述：ROUND 将 value 按照 precision 进行舍入，其中 precision 是一个整数，可以为正数、负数或零。负整数表示将小数点左边的给定位数进行舍入，正整数表示将小数点右边的给定位数进行舍入。

示例：

```
ROUND(123.456,0)  = 123
ROUND(123.456,-2) = 100
ROUND(-123.456,2) = -123.46
```

ROW_NUMBER

参阅：ROWNUM。

格式：

```
ROW_NUMBER ( ) OVER ( [ query_partition_clause] order_by_clause )
```

描述：ROW_NUMBER 是一个分析函数。它用 ORDER BY 子句指定行的有序序列(从 1 开始)，把唯一的编号分配给它所应用的每行(分区中的每行或查询返回的每行)。

ROWID

ROWID 是行的逻辑地址，它在数据库中是唯一的。

ROWID 可以用来 SELECT 或用于 WHERE 子句中，但不能用 INSERT、UPDATE 或 DELETE 修改。它不是实际的数据列，而仅仅是构成数据库地址信息的逻辑地址。ROWID 在 WHERE 子句中很有用，可以进行行的快速 UPDATE 或 DELETE。但当它所在的表进行导入和导出时它可能会变化。

ROWID 的数据类型是十六进制串，它表示表中行的唯一地址。这个数据类型主要用于 ROWID 伪列返回的值。

ROWIDTOCHAR

参阅：CHARTOROWID、CONVERSION FUNCTIONS。

格式：

```
ROWIDTOCHAR(RowId)
```

描述：ROWIDTOCHAR(ROW IDentifier TO CHARacter)修改 Oracle 内部的行标识符或 ROWID，使其起字符串的作用。但因为 Oracle 将自动进行这种转换，所以这个函数实际并不需要。它似乎可作为一般的调试工具使用。

ROWIDTONCHAR

参阅：CHARTOROWID、ROWIDTOCHAR、CONVERSION FUNCTIONS

格式：

```
ROWIDTONCHAR(RowId)
```

描述：ROWIDTONCHAR(ROW IDentifier TO NCHARacter)把 Oracle 内部行标识符或 ROWID 修改为 NVARCHAR2 数据类型。

ROWNUM

参阅：PSEUDO-COLUMNS。

格式：

```
ROWNUM
```

描述：ROWNUM 返回第 1 次从表中选择时返回的行的序列号。第 1 行的 ROWNUM 为

1，第 2 行的 ROWNUM 为 2，依此类推。但要注意，即使 SELECT 语句中简单的 ORDER BY 子句也可能会打乱(disorder)ROWNUM 的值(在进行排序前分配给各行的)。

RPAD

参阅：CHARACTER FUCTIONS、LPAD、LTRIM、RTRIM、TRIM、第 7 章。

格式：

```
RPAD(string,length [,'set'])
```

描述：RPAD(Right PAD)通过在右边添加一组(*set*)字符使字符串达到一定长度。如果不指定 *set*，则默认添加的字符为空格。

示例：

```
select RPAD('HELLO ',24,'WORLD') from DUAL;
```

输出结果为：

```
HELLO WORLDWORLDWORLDWOR
```

RTRIM

参阅：CHARACTER FUCTIONS、LPAD、LTRIM、RPAD、TRIM、第 7 章。

格式：

```
RTRIM(string [,'set'])
```

描述：RTRIM(Right TRIM)去掉字符串右边出现的一组字符中的任何字符。

示例：

```
RTRIM('GEORGE','OGRE')
```

输出结果什么也没有，是零长度的空字符串。另一方面：

```
RTRIM('EDYTHE','HET')
```

输出结果为：

```
EDY
```

RUN

参阅：/、@、@@、EDIT、START。

格式：

```
R[UN]
```

描述：RUN 显示 SQL 缓冲区中的 SQL 命令，然后执行它。除了“/”不首先显示 SQL 外，RUN 类似于“/”命令。

SAVE(SQL*Plus)

参阅： EDIT、GET、第 6 章。

格式：

```
SAV[E] file[.ext] [ CRE[ATE] | REP[LACE] | APP[END] ]
```

描述： SAVE 把当前缓冲区中的内容保存到宿主文件中，宿主文件名是 *file*。如果不指定文件类型(扩展名)，则在文件名上加上文件类型.sql。默认扩展名可用 SET SUFFIX 修改。SAVE 将挂起的工作提交给数据库。

如果文件已经存在，CREATE 选项就强行中止 SAVE 操作并产生错误消息。REPLACE 选项替换任何已存在文件，如果不存在任何文件就创建新文件。

APPEND 选项将此文件添加到任意已存在文件的末尾，如果不存在文件则创建新文件。

示例：

下面示例将当前缓冲区中的内容保存到 lowcost.sql 文件中。

```
save lowcost
```

SAVEPOINT

参阅： COMMIT、ROLLBACK、SET TRANSACTION。

格式：

```
SAVEPOINT savepoint;
```

描述： 保存点(savepoint)是事务处理中可以回滚的点。保存点允许回滚(ROLLBACK)到当前事务处理的部分。参阅 ROLLBACK。

下面显示的第一个版本是典型的 SQL*Plus。SAVEPOINT 的用途是能够在一组 SQL*Plus 语句的开始指定一个名字，然后在需要的时候回滚到这个名字，从而撤消该组语句的结果。Savepoint 可以嵌套也可以排序，可以很方便地控制从错误点的恢复。其基本思想是把构成逻辑事务的一系列 SQL 语句集中到一起，直到其中的所有步骤都已经成功完成后才提交(COMMIT)。这样如果某个步骤失败，就能够再次执行该步骤而不必撤消它前面所做的工作。下面是一个示例：

```
insert into COMFORT
    values ('WALPOLE', '22-APR-04',50.1, 24.8, 0);

    savepoint A;

    insert into COMFORT
    values ('WALPOLE', '27-MAY-04',63.7, 33.8, 0);

    savepoint B;

    insert into COMFORT
    values ('WALPOLE', '07-AUG-04',83.0, 43.8, 0);
```

可以通过返回到保存点 B，只回滚最后一个 INSERT 操作。

```
rollback to savepoint B;
```

此示例说明，可以将 SQL 语句或 SQL 语句组集中在一起，成组或逐条地撤消。在宿主程序中或者使用 PL/SQL 或 SQL*Plus 时，可能想要根据最近执行的 SQL 语句或条件测试的结果创建能够回滚的保存点。ROLLBACK 允许返回到指定保存点指定的步骤。

如果函数、逻辑部分或 SQL 语句失败，就可以将数据库中的数据返回到局部步骤开始前的状态。然后再次执行它。

SAVEPOINT 名对事物处理(其定义以 COMMIT 或无条件的 ROLLBACK 结束)来说必须是唯一的，但不必对所有数据库对象都唯一。例如 SAVEPOINT 可以与某个表同名(如果 SAVEPOINT 与要更新的表同名，则甚至还可能会使代码更容易理解)。SAVEPOINT 的名字遵循对象命名约定。

如果新 SAVEPOINT 的名字与前面的 SAVEPOINT 相同，则新的 SAVEPOINT 将取代前面的那个 SAVEPOINT。前面那个 SAVEPOINT 丢失，再也不能回滚到它了。

一旦发布 COMMIT 或无条件的 ROLLBACK(不是回滚到某个保存点)，则前面所有的 SAVEPOINT 都被删除。回忆一下，所有 DDL 语句(如 DISCONNECT、CREATE TABLE、CREATE INDEX 等等)都自动发布隐含的 COMMIT，任何严重的故障(如果程序异常中止或计算机死机)将导致自动执行 ROLLBACK。

SCN_TO_TIMESTAMP

参阅： PSEUDO_COLUMNS 下的 ORA_ROWSCN

格式：

```
SCN_TO_TIMESTAMP ( number )
```

描述：SCN_TO_TIMESTAMP 接收参数 number，该 number 等于系统更改号(system change number，SCN)，并返回与该 SCN 相关联的近似的时间戳。返回值是 TIMESTAMP 数据类型。这个函数可以在任何时候获得相关 SCN 的时间戳。例如，可以与 ORA_ROWSCN 伪列联合使用将行的最近修改与时间戳关联起来。

SCORE

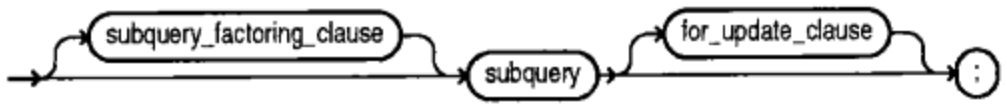
SCORE 函数被 Oracle Text 中的 CONTEXT 索引用来判定文本字符串对指定的搜索条件的满足程度。可以显示单个搜索的得分，通常将此得分与某个阈值进行比较。

SELECT(格式 1——SQL)

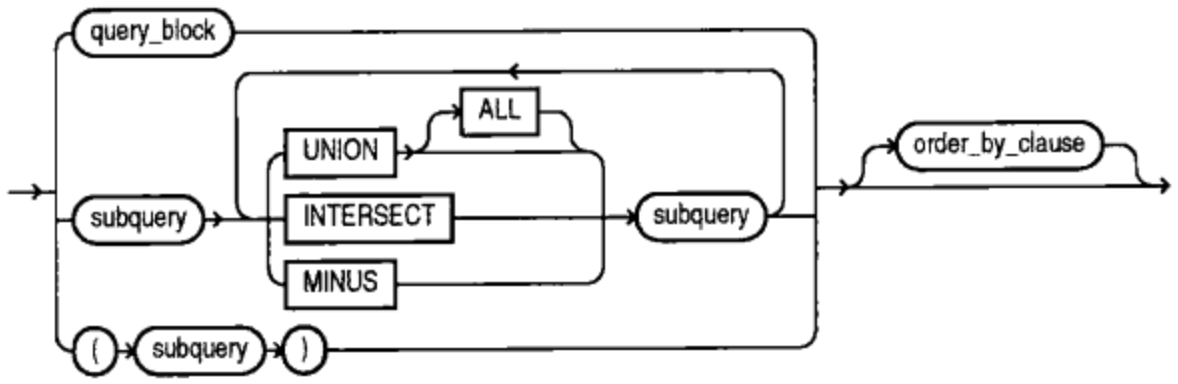
参阅： CONNECT BY、DELETE、DUAL、FROM、GROUP BY、HAVING、INSERT、JOIN、LOGICAL OPERATORS、ORDER BY、QUERY OPERATORS、SUBQUERY、SYNTAX OPERATORS、UPDATE、WHERE

格式：

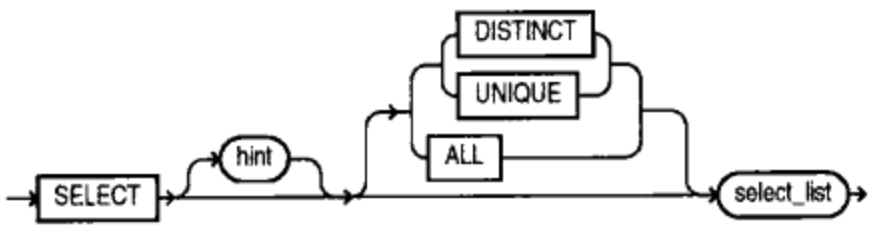
select::=

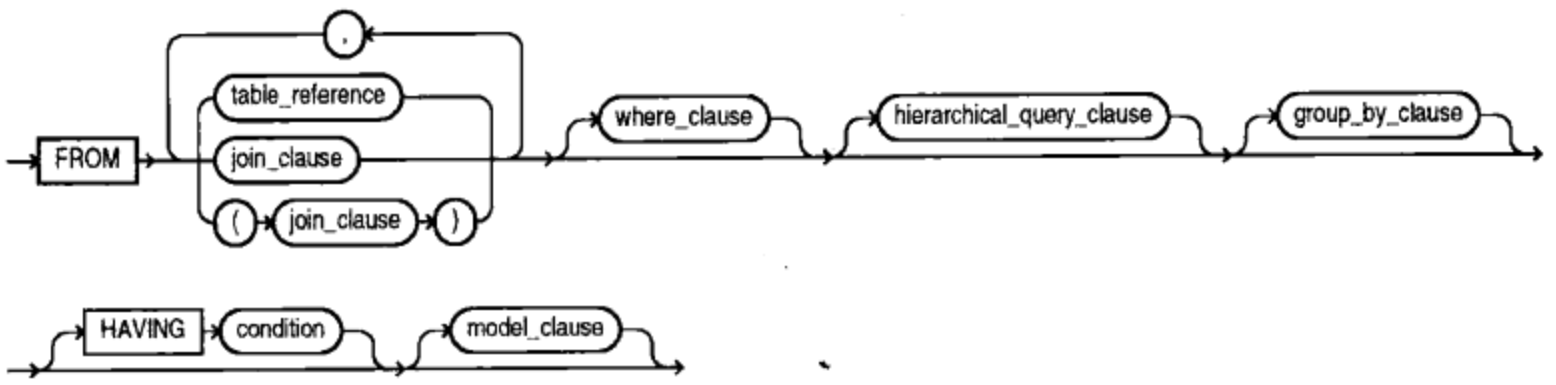


subquery::=

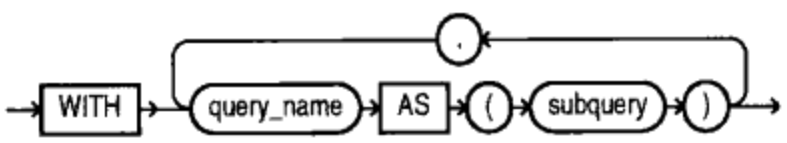


query_block::=

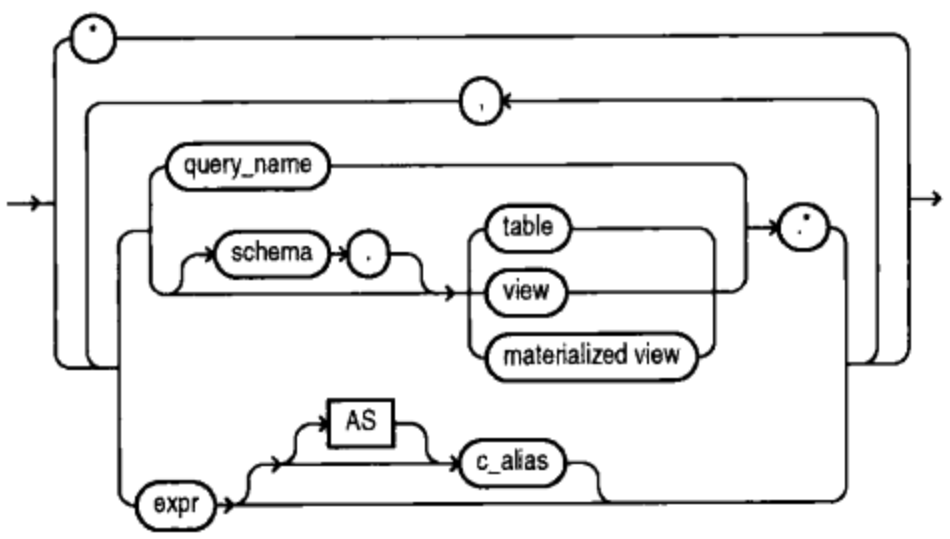




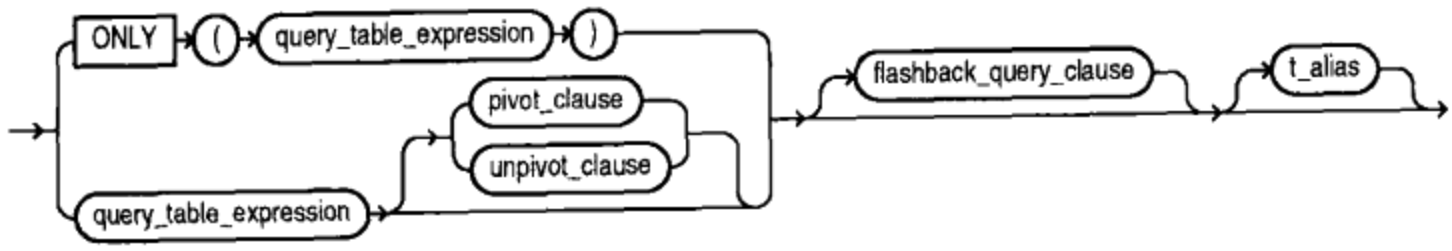
subquery_factoring_clause::=



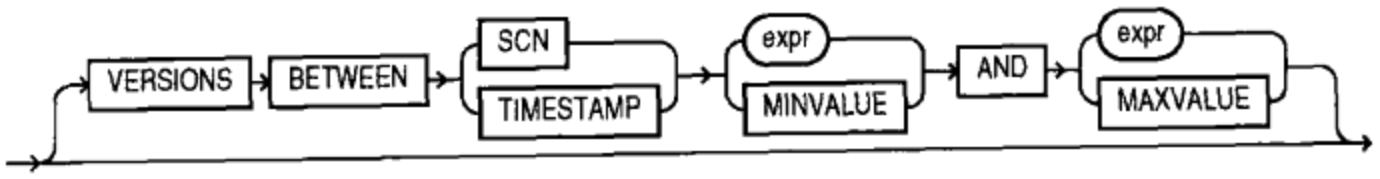
select_list::=

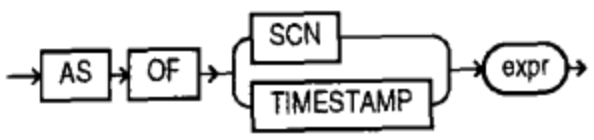


table_reference::=

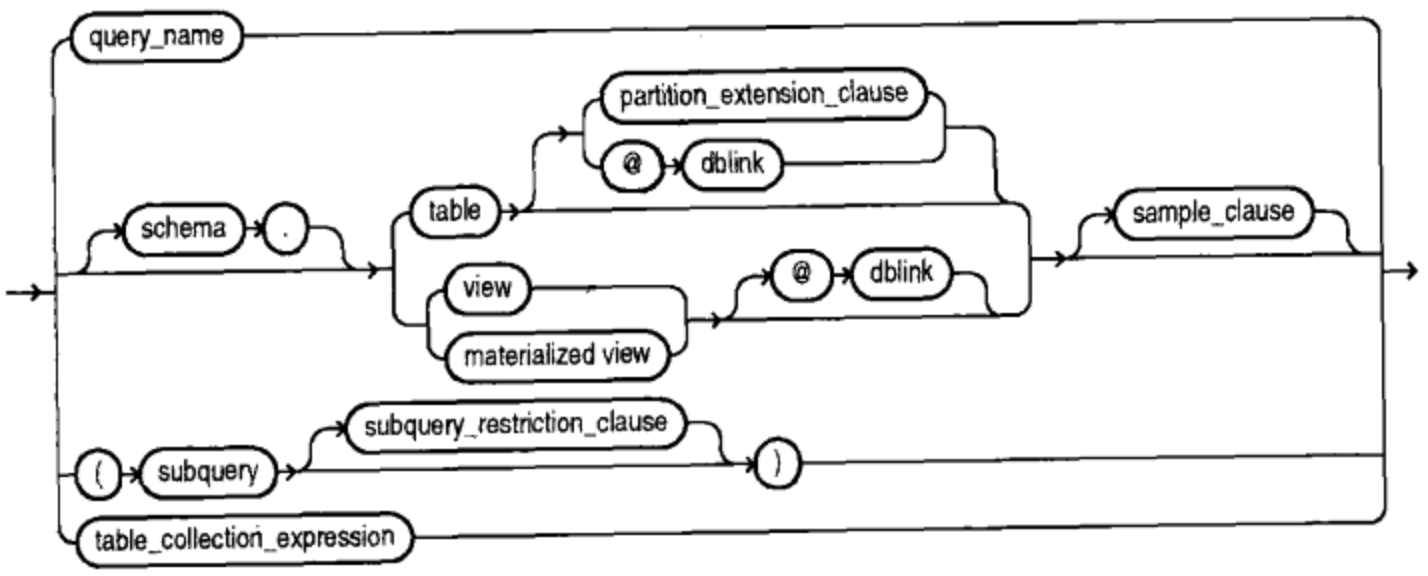


flashback_query_clause::=

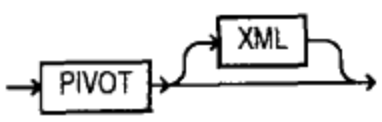


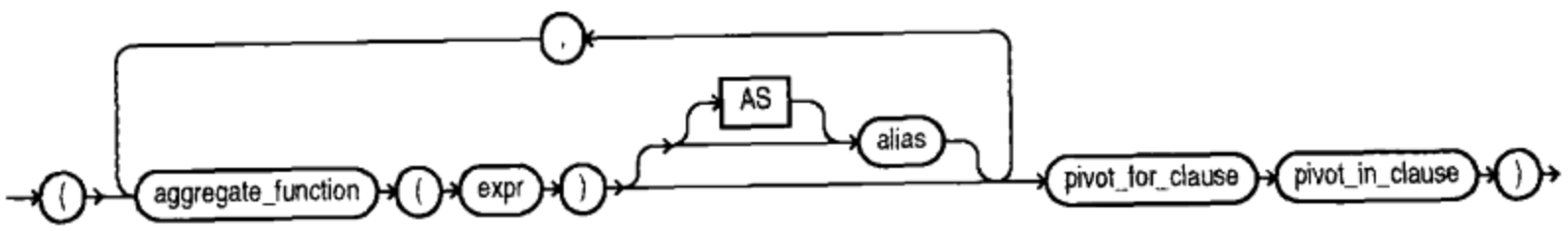


query_table_expression::=

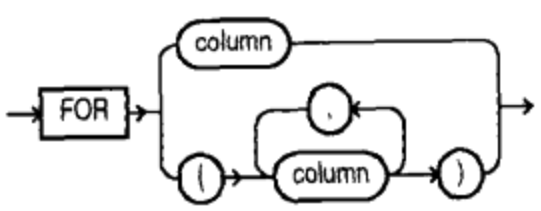


pivot_clause::=

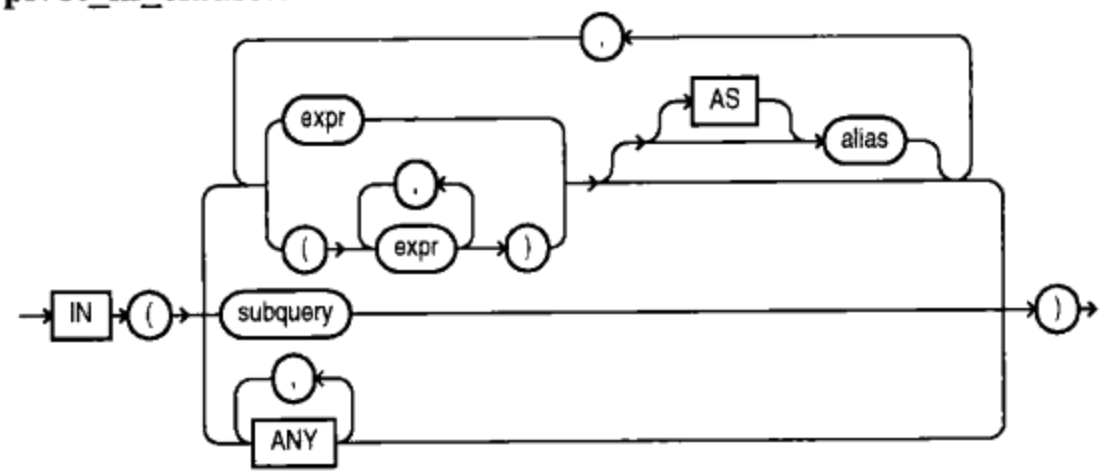




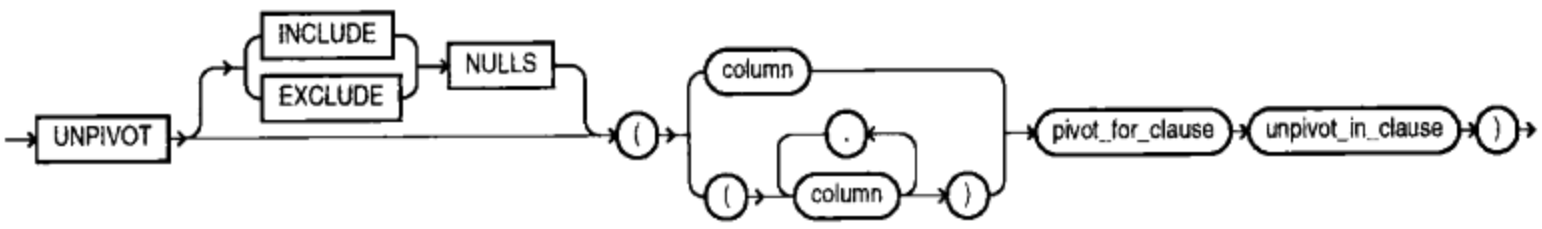
pivot_for_clause::=



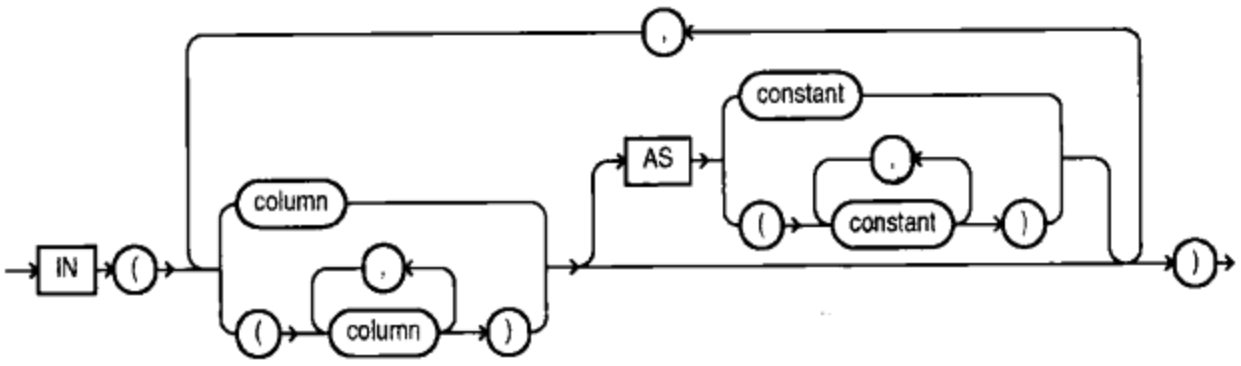
pivot_in_clause::=



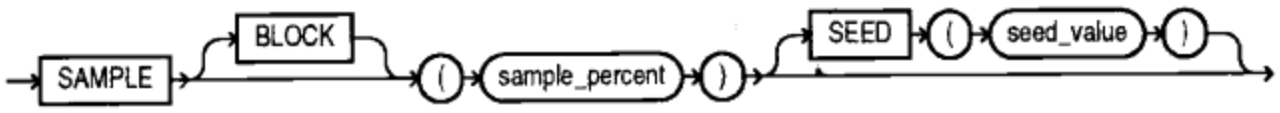
unpivot_clause::=



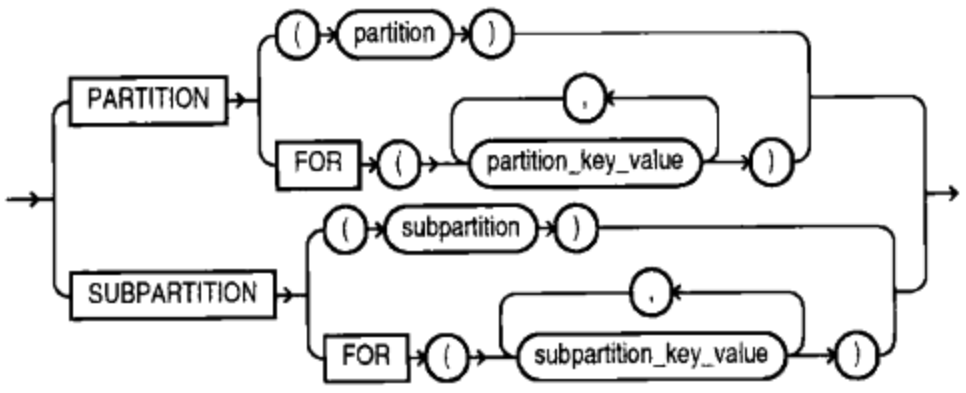
unpivot_in_clause::=



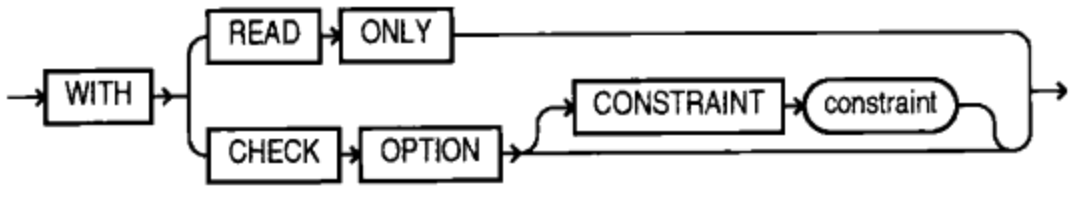
sample_clause::=



partition_extension_clause::=

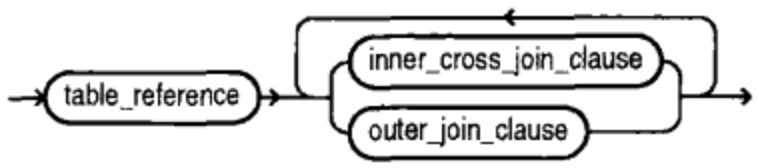


subquery_restriction_clause::=

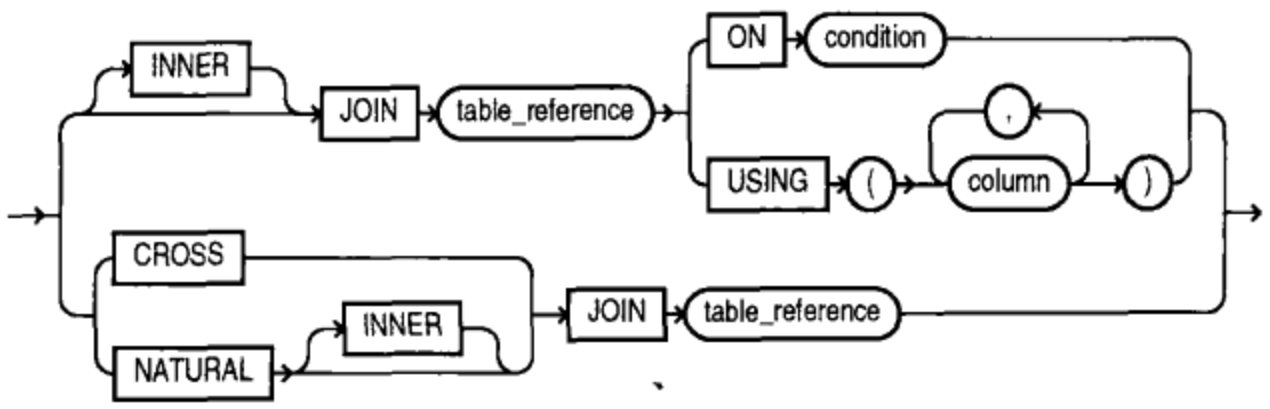


table_collection_expression::=

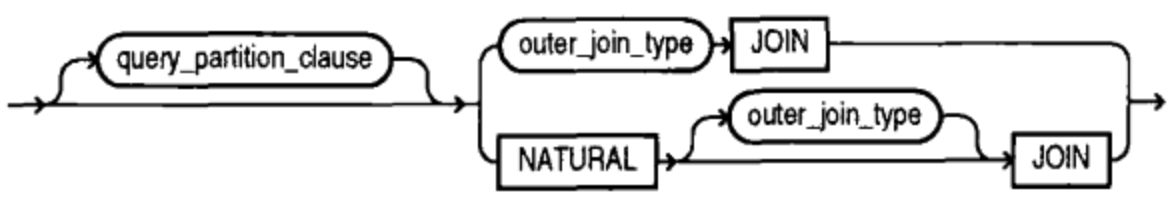
join_clause::=



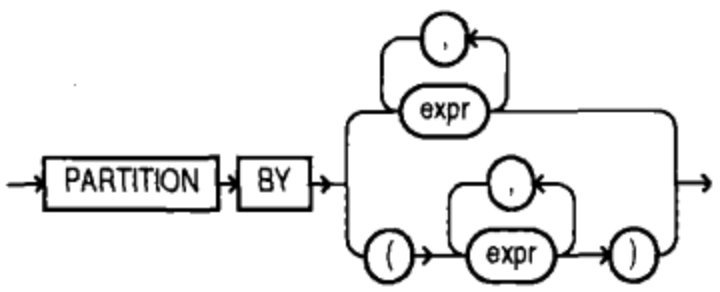
inner_cross_join_clause::=



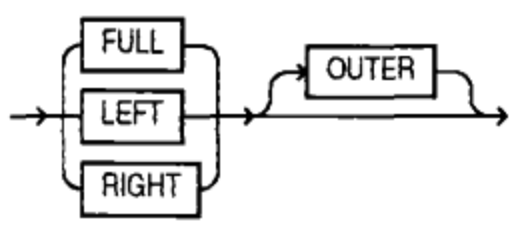
outer_join_clause::=



query_partition_clause::=

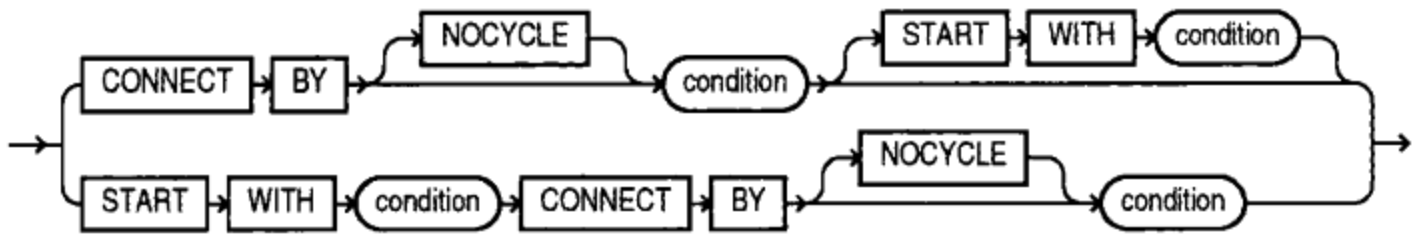


outer_join_type::=

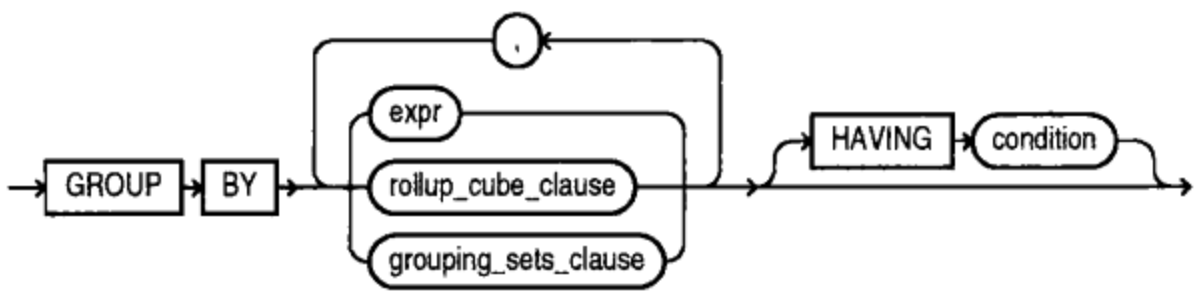


where_clause::=

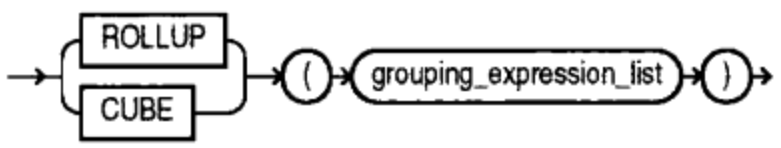
hierarchical_query_clause::=



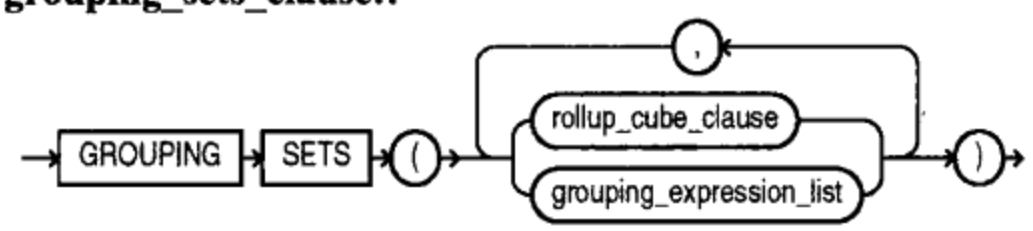
group_by_clause::=



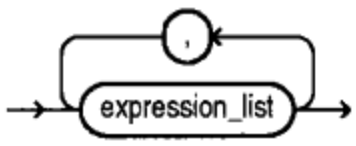
rollup_cube_clause::=



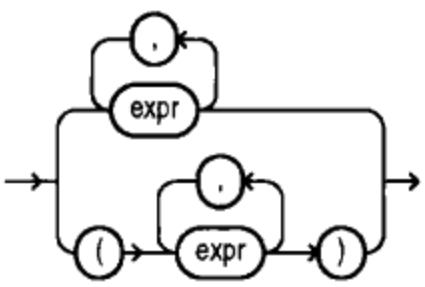
grouping_sets_clause::=



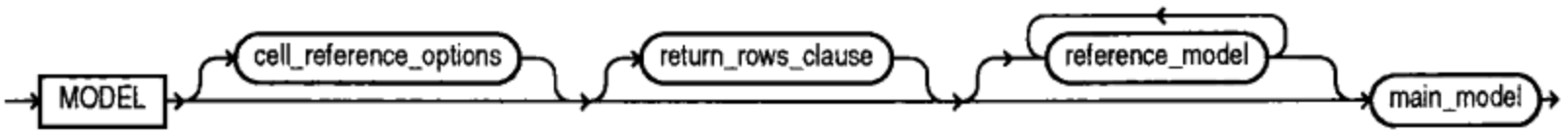
grouping_expression_list::=



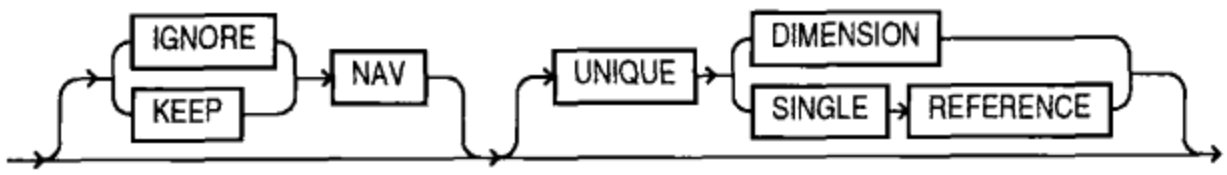
expression_list::=



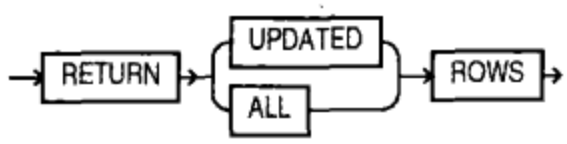
model_clause::=



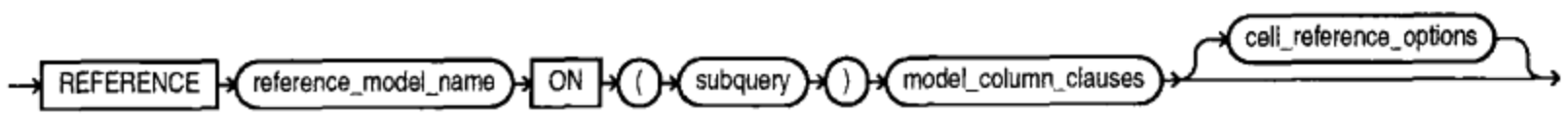
cell_reference_options::=



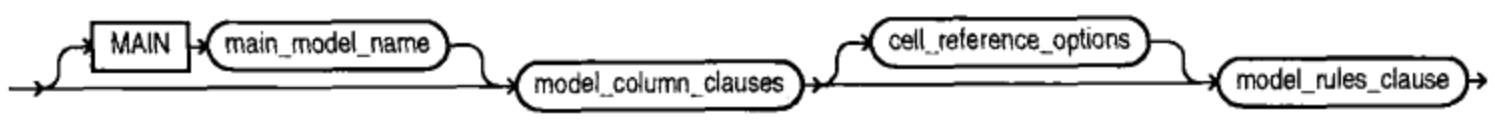
return_rows_clause::=



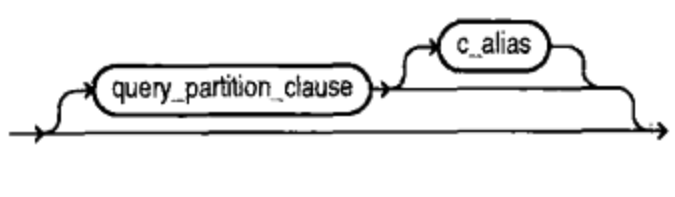
reference_model::=

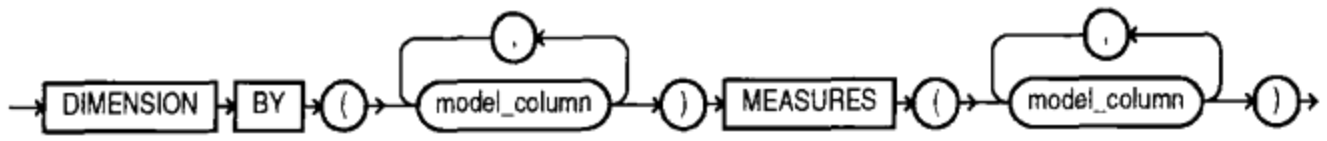


main_model::=

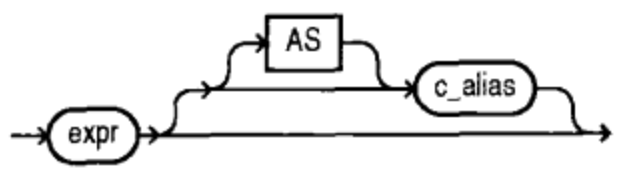


model_column_clauses::=

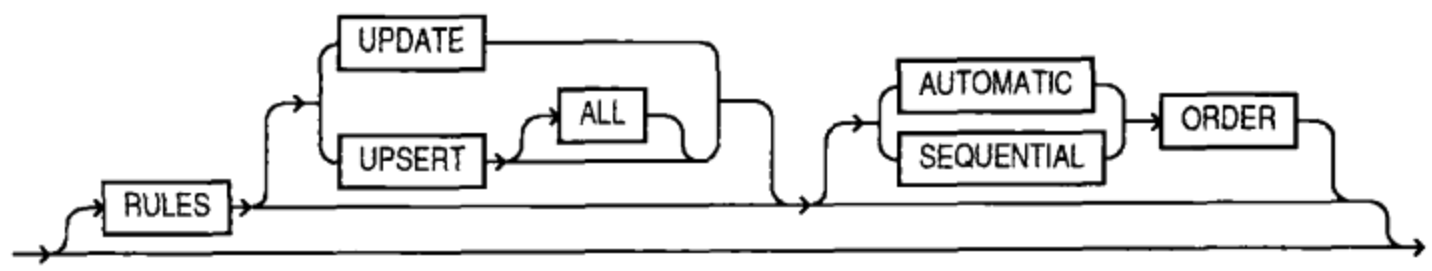


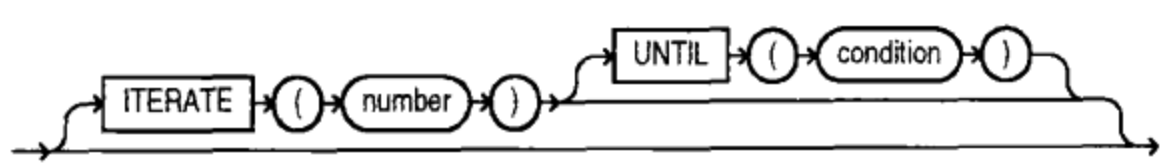


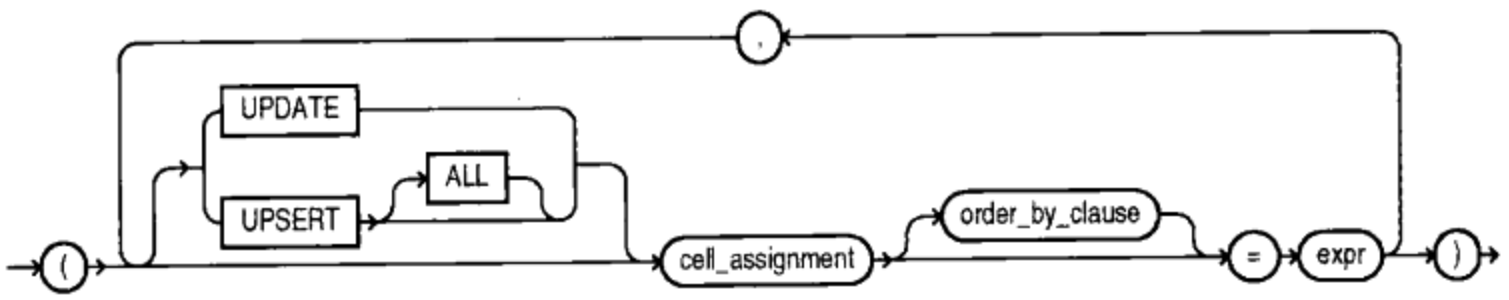
model_column::=



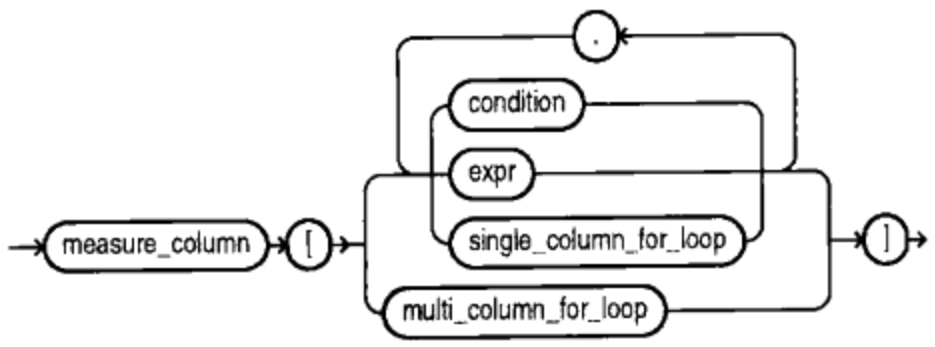
model_rules_clause::=



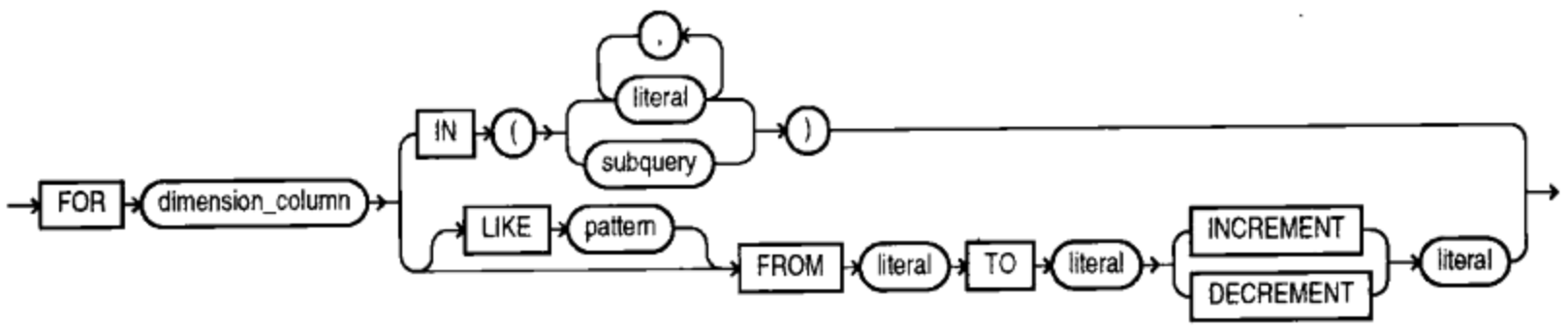




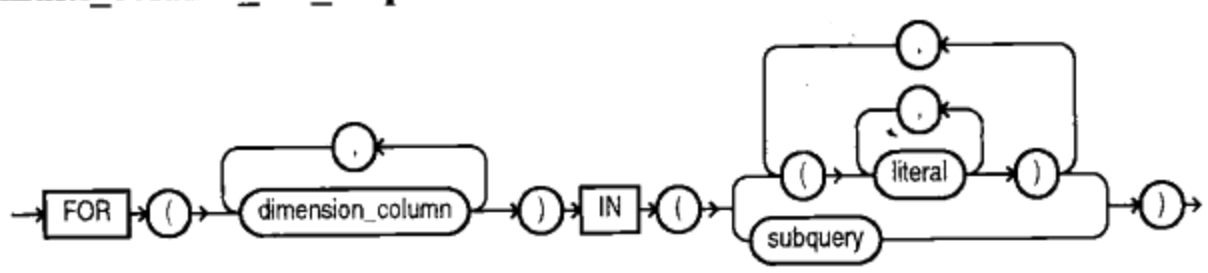
cell_assignment::=



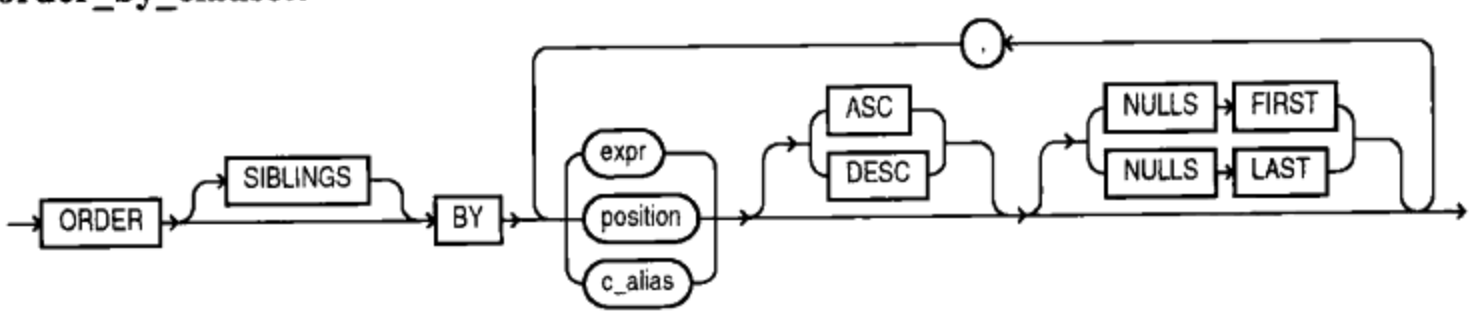
single_column_for_loop::=



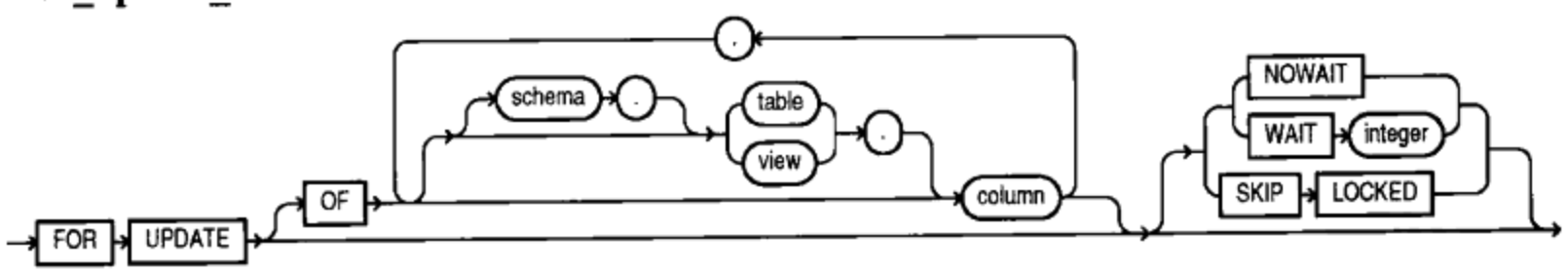
multi_column_for_loop::=



order_by_clause::=



for_update_clause::=



描述：SELECT 从一个或多个表或视图中检索行，它可以作为命令或其他 SQL 命令中的子查询(有限制)，包括 SELECT、INSERT、UPDATE 和 DELETE。ALL 表示返回所有满足条件的行(这是默认的)。DISTINCT 表示只返回那些唯一的行，任何重复的行将首先被剔除。

*(星号)本身显示 FROM 子句中所有的表的所有的列。table.*表示显示该表的所有列。星号本身不能与其他列组合，但 table.*后面可以跟列名和表达式，包括 ROWID。

select_list 可以包括列的任何形式。它可以是列名、字面量、数学计算、函数或几个函数的组合。最常用的形式是简单的列名或在列名前放置表名或表别名。c_alias 是列或表达式的重命名。列别名前可放置单词 AS。此列别名在显示时将成为列标题，可以被 SQL*Plus 中的 COLUMN、TTITLE 和 BTITLE 命令引用。列别名可用于 ORDER BY 子句中。如果列别名具有多个词，或者包含既非数字也非字母的字符，则必须用双引号括起来。

schema、table 和 dblink 表示从其中提取行的表或视图。schema 和 dblink 是可选的，如果不给出它们则 Oracle 默认使用当前用户。但是在查询中指定用户可以减少 Oracle 的系统开销并使查询运行得更快。可以在 FROM 子句中指定子查询。Oracle 将执行此子查询，对待其结果行就像它们来自于视图一样。

这里的表 t_alias 将重命名这个查询的表。这个别名不同于表达式别名，它可在 SELECT 语句中其他地方引用，如作为该表的任何列名的前缀。事实上，如果表有别名，则这个别名必须在该表的列具有表名前缀时使用。当在查询中连接两个长名字的表，而且这两个表的许多列具有相同的名字，并需要在 SELECT 子句中明确地指定它们时，一般要给这两个表起别名。别名一般也用在相关子查询中和将一个表连接到自身时。

condition 可以是任何测试真或假的有效表达式。它可以包含函数、列(来自表)和字面量。position 允许 ORDER BY 子句根据表达式在 SELECT 子句的相对位置而不是根据名字来标识表达式。这对于复杂表达式是很有价值的。不应该继续在 ORDER BY 子句中使用顺序位置；可以在这些位置使用列别名。当前的 SQL 标准不支持在 ORDER BY 子句中使用顺序位置，以后 Oracle 也不可能支持它们。ASC 和 DESC 指定 ORDER BY 子句中每个表达式是升序还是降序。

column 是在 FROM 子句中列出的表中的列。这不能是表达式，但必须是实际的列名。NOWAIT 表示遇到锁定行的 SELECT…FOR UPDATE 企图时立即终止并返回给用户，而不是等待片刻后再次更新该行。

FOR UPDATE OF 子句给已经选择的行加锁。SELECT…FOR UPDATE OF 后紧跟一条 UPDATE…WHERE 命令，或者，如果决定不更新任何东西，则跟一条 COMMIT 或 ROLLBACK 命令。FOR UPDATE OF 子句也包括 INSERT 和 DELETE 操作。一旦锁定了一行，其他用户不能更新它，直到这一行被 COMMIT 命令(或 AUTOCOMMIT)或 ROLLBACK 释放。SELECT…FOR UPDATE OF 不能包含 DISTINCT、GROUP BY、UNION、INTERSECT、MINUS 或任何分组函数，如 MIN、MAX、AVG 或 COUNT。所指出的列没有意义，给出它只是为了与其他 SQL 版本兼容。Oracle 只锁定出现在 FOR UPDATE 子句中的表。如果 OF 子句没有列出任何表，则 Oracle 将锁定 FROM 子句中的所有表，以便进行更新。所有这些表必须位于同一数据库中，而且如果在 SELECT 中有对 LONG 列或序列的引用，则这些表必须位于与 LONG 列或序列相同的数据库中。

如果对分区表进行查询，就可以列出作为查询的 FROM 子句的组成部分的分区名。详细内容请参阅第 18 章。

SELECT 语句中其他个别子句都在本附录中相应的条目下进行描述。在 Oracle 9i Release 2 中引入了 AS OF{TIMESTAMP | SCN}子句进行闪回查询。在 Oracle Datebase 10g 中引入了 VERSIONS BETWEEN 子句进行闪回版本查询(flashback version query)。

在 Oracle Database 11g 中，可以使用新增的 PIVOT 和 UNPIVOT 操作将行旋转成列。

其他说明：子句必须按照给出的顺序放置，以下情况除外：

- CONNECT BY、START WITH、GROUP BY 和 HAVING，它们的相互顺序可以是任意的。
- ORDER BY 和 FOR UPDATE OF，它们的相互次序可以是任意的。

SELCET(格式 2——嵌入 SQL)

参阅：CONNECT 、DECLARE CURSOR、DECLARE DATABASE、EXECUTE、FETCH、FOR、PREPARE、UPDATE(格式 1)、WHENEVER、*Programmer's Guide to the Oracle Precompliers*。

格式：

```
EXEC SQL [AT { db_name | :host_variable }]
  SELECT select_list INTO
  :host_variable [[INDICATOR] :indicator_variable]
  [, :host_variable [[INDICATOR] :indicator_variable]]...
  FROM table_list [WHERE condition]
  { [START WITH condition] CONNECT BY condition
  | CONNECT BY condition [START WITH condition]
  | GROUP BY expr [, expr]... [HAVING condition] }
  [ [START WITH condition] CONNECT BY condition
  | CONNECT BY condition [START WITH condition]
  | GROUP BY expr [, expr]... [HAVING condition] ]...
  [ WITH READ ONLY | WITH CHECK OPTION ]
  [{ UNION | UNION ALL | INTERSECT | MINUS } SELECT command]
[ ORDER BY
    { expr | position } [ ASC | DESC ]
    [, { expr | position } [ ASC | DESC ]]...
| FOR UPDATE
    [OF [[schema.] { table. | view. | materialized view. }] column
      [, OF [[schema.] { table. | view. | materialized view. }] column]... ]
        [NOWAIT] ]
[ ORDER BY
  { expr | position } [ ASC | DESC ]
  [, { expr | position } [ ASC | DESC ]]...
| FOR UPDATE
  [OF [[schema.] { table. | view. | materialized view. }] column
    [, OF [[schema.] { table. | view. | materialized view. }] column]... ]
   [NOWAIT] ]...
```

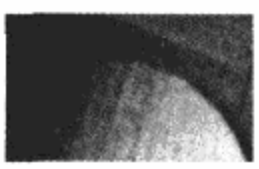
描述：请参阅 SELECT(格式 1)中各种子句的描述。下面是嵌入式 SQL 特有的元素：

- AT 数据库可选择性地指定来自前面的 CONNECT 语句的数据库，数据库名来自前面的 DECLARE DATABASE 语句。
- INTO:host_variable[,:host_variable]是宿主变量列表，SELECT 语句的结果将加载到其中。如果这个列表中的任何变量是数组，则列表中的所有变量都必须是数组，尽管不必是大小相同的数组。
- WHERE 子句可以引用非数组的宿主变量。
- SELECT 子句可以在使用常量的任何地方引用宿主变量。

可利用带变量数组的嵌入式 SQL SELECT，用单个 FETCH 返回多行。还可以把它用于与后面的 FETCH 一起使用的 DECLARE CURSOR 或 PREPARE 语句，并且它可以包含 FOR UPDATE OF 子句。以后的 UPDATE 语句可以使用它自己的 CURRENT OF 子句引用 FOR UPDATE OF 子句中指定的列。详细内容请参阅 DECLARE CURSOR 和 UPDATE(格式 2)。

如果嵌入的 SELECT 没有返回行，则设置 SQLCODE 为＋100，且宿主变量保持不变。在这种情况下，WHENEVER 允许修改程序流。

满足 SELECT 条件的所有行都在打开时锁定。COMMIT 释放这些锁，不允许进行进一步的 FETCH 操作。这表示必须在发布 COMMIT 命令之前 FETCH 和处理希望更新的所有行。



注意：
所有命令的完整列表请参阅 Programmer's Guide to the Oracle Precompliers。
本附录不包含所有可能的预编译程序命令。

SELECT…INTO

参阅：%ROWTYPE、%TYPE、DECLARE CURSOR、FETCH。

格式：

```
SELECT expression [,expression]...
  INTO {variable [,variable]... | record}
  FROM [user.]table [,[user.]table]...
 [where...][group by... [having...]] [order by...];
```

描述：这不是在 DECLARE CURSOR 中使用的 SELECT 语句的格式。这种格式利用命名为 SQL 的隐式游标，在块的执行部分(在 BEGIN 和 END 之间)执行，并将正返回的一行的值复制到一个指定变量的字符串中，或者复制到其结构由%ROWTYPE(就像所选择的列那样)声明的记录中。如果采用这种使用变量的格式，则各变量必须进行 DECLARE 且与检索的那些列具有兼容的数据类型。变量在 INTO 子句中出现的顺序必须对应于 SELECT 子句中相应列的顺序。

这种 SELECT 可用在如下循环中：PL/SQL 变量出现在 WHERE 子句中的循环中(或者甚至在 SELECT 子句中起常量作用)，但每次执行 SELECT 时，它必须只返回一行。如果返回多行，则将会触发 TOO_MANY_ROWS 异常，并将 SQL%ROWCOUNT 设置为 2(详细内容请参阅 EXCEPTION)。SQL%FOUND 将为真。

如果不返回任何行，则将触发 NO_DATA_FOUND 异常，并将 SQL%ROWCOUNT 设置

为 0。SQL%FOUND 将为假。

SESSIONTIMEZONE

参阅： 第 10 章。

格式：

```
SESSIONTIMEZONE
```

描述： SESSIONTIMEZONE 返回当前会话的时区值。

示例：

```
select SESSIONTIMEZONE from DUAL;
```

SET(SQL*Plus)

参阅： SET TRANSACTION、SHOW、第 6 章。

格式：

```
SET feature value
```

其中 feature 和 value 如下面的清单所示。默认值加下划线。

- APPI[NFO] {ON | OFF | text}通过 DBMS_APPLICATION_INFO 程序包启用命令文件的注册。注册名出现在 V$SESSION 动态性能表的 Module 列中。
- ARRAY[SIZE]{15 | n}设置 SQL*Plus 可以一次取出的批处理的行的大小。范围是 1~5000。较大的值可以改善取出很多行的查询和子查询的效率，但会占用更多内存。使用 100 以上的值一般不会产生太大改善。ARRAYSIZE 对 SQL*Plus 的工作没有其他作用。
- AUTO[COMMIT] {ON | OFF | IMM[EDIATE] | n}使 SQL 在每条 SQL 命令完成后立即向数据库提交挂起的更改。n 指定完成 n 条命令后进行提交。OFF 停止自动提交，这样必须专门用 COMMIT 命令提交更改。很多命令，如 QUIT、EXIT、CONNECT 和所有 DDL 命令将导致任何挂起的更改的 COMMIT。
- AUTOP[RINT] {ON | OFF}设置 SQL*Plus 是否自动显示 PL/SQL 块或被执行的过程中使用的绑定变量。
- AUTORECOVERY [ON | OFF] 要求 RECOVER 命令自动应用恢复期间所需的默认归档重做日志文件的文件名。
- AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]允许在查询完成后查看每个查询的执行路径。为了查看这个执行路径，首先必须在自己的模式中创建 PLAN_TABLE 表(通过运行 Oracle 软件目录下的/rdbms/admin 目录中的 utlxplan.sql 文件创建它)。
- BLO[CKTERMINATOR] {. | c | ON | OFF}设置用来表示 PL/SQL 块结束的符号。这个符号不能是字母或数字。使用 RUN 或“/”(斜杠)执行这个块。

- CMDS[EP] {; | c | ON | OFF}设置分隔一行上的多条 SQL*Plus 命令的分隔符号。ON 和 OFF 控制在一行上是否能输入多个命令。
- COLSEP {_ | text}设置列之间输出的值。
- COM[PATIBILITY]{V7 | V8 | NATIVE}指定当前连接的 Oracle 版本。
- CON[CAT] {. | c | ON | OFF}设置一个符号，该符号可用来终止或限定后面跟一个符号、字符或单词的用户变量，否则这个符号、字符或单词将被认为是用户变量名的组成部分。设置 COMCAT ON 将重置其值为句点(.)。
- COPYC[OMMIT] {0 | n}将 n 个批处理的行周期性地提交给目标数据库(一个批处理中行的数目由 ARRAYSIZE 设置)。有效的值是 0~5000。如果值为 0，则仅在 COPY 操作结束时进行提交。
- COPYTYPECHECK {ON | OFF}允许在使用 COPY 命令时禁止进行数据类型检查。
- DEF[INE] {& | c | ON | OFF}确定 SQL*Plus 是否要查找和代替要替换变量的命令并用它们的 DEFINE 的值加载它们。
- DESCRIBE [DEPTH {1 | n | ALL}][LINENUM{ON | OFF}][INDENT{ON | OFF}]设置层次深度，可就这个深度递归地描述对象。DEPTH 子句的有效范围是 1~50。
- ECHO {ON | OFF}在执行启动文件时将 SQL*Plus 命令回显(显示)到屏幕上。OFF 使 SQL*Plus 执行它们而不显示它们。另一方面，命令的结果受 TERMOUT 控制。
- EDITF[ILE] file_name[.ext]设置 EDIT 命令创建的默认文件名。
- EMB[EDDED] {ON | OFF}允许一系列报表中的新报表在前一个报表结束后从页面上任何地方开始。OFF 强制新报表在新页面的顶部开始。
- ERRORL[OGGING] {ON | OFF} [TABLE [schema.]tablename] [TRUNCATE] [IDENTIFIER identifier]启用将 SQL、PL/SQL 和 SQL*Plus 的错误记录到错误日志中。查询错误被写入默认表 SPERRORLOG 中，此表默认在当前模式中。
- ESC[APE]{\ | c | ON | OFF}定义所输入的字符为转义字符。OFF 不定义转义字符。ON 启用转义字符。ON 把 c 的值变回默认的“\”。可在替换字符(通过 SET DEFINE 设置)前使用转义字符指示 SQL*Plus 把替换字符作为普通字符而不是变量替换的请求来对待。
- ESCCHAR [@ | ? | % | $ | OFF]使文件名中的特定字符能够进行转义。
- FEED[BACK] {6 | n | ON | OFF}使 SQL*Plus 在查询之后显示“所选择的记录”(如果至少选择了 n 个记录)。ON 或 OFF 分别表示打开或关闭此显示。SET FEEDBACK 为 0 与 OFF 的作用相同。
- FLAGGER {OFF | ENTRY | INTERMED[IATE] | FULL}设置 FIPS 级别与 SQL92 兼容。
- FLU[SH] {ON | OFF}在可以运行启动文件且在完成之前不需要显示或交互时使用。OFF 使操作系统不显示输出。ON 还原输出的显示。OFF 可以改善性能。
- HEA[DING] {ON | OFF}禁止通常出现在列上面的标题(文本和下划线)。 ON 允许显示它们。

- HEADS[EP]{⊥| c | ON | OFF}定义输入的字符为标题分隔字符。ON 或 OFF 分别表示打开或关闭标题分隔。
- INSTANCE [instance_path | LOCAL]更改会话的默认实例为指定的实例路径。使用 SET INSTANCE 命令不连接到数据库。在不指定实例时，命令使用默认实例。
- LIN[ESIZE] {80 | n}设置行宽，即转到下一行之前行上可以显示的所有字符数。这个数字在 SQL*Plus 为标题居中或右对齐而计算合适的位置时也可以使用。n 的最大值为 999。在 iSQL*Plus 中，默认值是 150。
- LOBOF[FSET] {n | 1}设置开始位置，从这个位置开始检索和显示 CLOB 和 NCLOB 数据。
- LOGSOURCE [pathname]指定恢复期间检索归档日志的起始位置。默认值由 Oracle 初始化文件中的初始化参数 LOG_ARCHIVE_DEST 设置。使用不给出路径名的 SET LOGSOURCE 命令将还原默认位置。
- LONG {80 | n}是显示或复制(假脱机)LONG、CLOB 和 NCLOB 值的最大宽度。n 的值在 1~2GB 之间。
- LONGC[HUNKSIZE] {80 | n}设置 SQL*Plus 检索 LONG、CLOB 和 NCLOB 值的递增大小，以字符为单位。
- MARK[UP] HTML [ON|OFF][HEAD text][text][TABLE text][ENTMAP{ON | OFF}] [SPOOL {ON | OFF}] [PRE[FORMAT] {ON | OFF}]输出 HTML 标记文本。SET MARKUP 仅指定 SQL*Plus 输出为编码的 HTML。必须使用 SET MARKUP HTML ON SPOOL ON 和 SQL*Plus 的 SPOOL 命令创建和命名假脱机文件，并开始把 HTML 输出写入它。SET MARKUP 与 SQLPLUS_MARKUP 具有相同的选项和行为。
- NEWP[AGE] {1 | n | NONE}设置在页的底部和下一页的顶部标题之间输出的空行数。0 在每页的顶部发送一个换页。如果正在显示其输出，则一般会清除屏幕。
- NULL text 设置 SQL*Plus 发现空值时替换的文本。无文本的 NULL 显示空白。
- NUMF[ORMAT] format 设置显示数据项时的默认数字格式。详细内容请参阅 NUMBER FORMATS。
- NUM[WIDTH] {10 | n}设置数值显示的默认宽度。
- PAGES[IZE] {14 | n}设置每页的行数。
- PAU[SE]{ON | OFF | text}使 SQL*Plus 在显示每个输出页后等待用户按回车键。OFF 表示在显示页期间不暂停。text 是 SQL*Plus 在等待用户按回车键时显示在屏幕底部的消息。PAUSE 在显示第 1 页之前将等待，用户必须按回车键才能看到该页。
- RECSEP {WR[APPED] | EA[CH] | OFF}指定查询结果的换行规则。SQL*Plus 返回的每一行可用一个字符与其他行分开，这个字符由 RECSEPCHAR 定义。默认情况下，这个字符为空格，并且只用于换行的记录(从而也用于列)。
- RECSEPCHAR 与 RECSEP 一起使用。默认为空格，但如果愿意，就也可设置它为别的符号。
- SERVEROUT[PUT] {ON | OFF} [SIZE n] [FOR[MAT]{WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}] 能够显示 PL/SQL 过程(来自 DBMS_OUTPUT 程序包——请参阅

Application Developer's Guide)的输出。WRAPPED、WORD_WRAPPED、TRUNCATED 和 FORMAT 用来确定输出文本的格式。

- SHIFT[INOUT] {VIS[IBLE] | INV[ISIBLE]}允许纠正显示转义字符的终端的对齐方式。应该对将转义字符显示为可见字符(如空格或冒号)的终端使用 VISIBLE。INVISIBLE 正好相反，它不显示换档字符。
- SHOW[MODE] {ON | OFF}使 SQL*Plus 显示 SET 特性的新旧设置，以及在它更改时显示其值。OFF 停止显示它们。
- SQLBL[ANKLINES] {ON | OFF}控制 SQL*Plus 在 SQL 命令或脚本中是否允许使用空行。ON 将空行和新行解释为 SQL 命令或脚本的组成部分。OFF 为默认设置，它不允许在 SQL 命令或脚本中有空行或新行。
- SQLC[ASE] {MIX[ED] | LO[WER] | UP[PER]}在 Oracle 执行 SQL 或 PL/SQL 块前转换其中的所有的文本，包括字面量和标识符。MIXED 使字符保持输入时的大小写。LOWER 将所有字符转换为小写，UPPER 把所有字符转换为大写。
- SQLCO[NTINUE]{> | text}为必须在下一行继续的长行设置为提示符显示的字符序列。如果在 SQL*Plus 命令行的最后输入“-” (连字符)，则此 SQLCONTINUE 符号或文本从屏幕下一行的左边开始提示用户。
- SQLN[UMBER] {ON | OFF}告诉 SQL*Plus 超出所输入的第 1 行的 SQL 行将具有与提示相同的行号。如果它为 OFF，则 SQLPROMPT 将只在另外的行上显示。
- SQLPLUSCOMPAT[IBILITY] {x.y[.z]}设置 VARIABLE 的行为或输出格式为 x.y[.z]指定的修订版本或版本。其中 x 为版本号，y 为修订版本号，z 为更新号。在后面的发修订本中，SQLPLUSCOMPATIBILITY 会影响特性而不是 VARIABLE。设置 SQLPLUSCOMPATIBILITY 的值为小于 9.0.0 的版本将导致 NCHAR 或 NVARCHAR2 数据类型的 VARIABLE 定义恢复到 Oracle8i 的行为，而变量的大小以字节或字符为单位依赖于所选的国家语言字符集。
- SQLPRE[FIX] {#|c}设置 SQL*Plus 前缀字符。在输入 SQL 命令或 PL/SQL 块时，可在单独的行上输入一条 SQL*Plus 命令，以此 SQL*Plus 前缀字符作为前缀。SQL*Plus 将立即执行此命令而不影响所输入的 SQL 命令或 PL/SQL 块。此前缀字符必须不能是一个字母数字字符。
- SQLP[ROMPT] {SQL >| text}设置 SQL*Plus 命令提示符。
- SQLT[ERMINATOR] {; | c | ON | OFF}设置用于结束 SQL 命令并立即开始执行 SQL 命令的字符为 c。
- SUF[FIX] {SQL | text}设置 SQL*Plus 在引用命令文件的命令中使用的默认文件扩展名。SUFFIX 不控制假脱机文件的扩展名。
- TAB {ON | OFF}确定 SQL*Plus 在终端的输出中怎样格式化空格。OFF 表示用空格格式化输出中的空格。ON 表示使用制表符格式化输出中的空格。Tab 设置为每 8 个字符一组。
- TERM[OUT] {ON | OFF}启用或禁止显示，因此可从命令文件中假脱机输出而在屏幕上看不到该输出。ON 显示该输出。

- TI[ME] {ON | OFF}在每个命令提示符前显示当前时间。OFF 禁止显示时间。
- TIMI[NG] {ON | OFF}显示在每条 SQL 命令或 PL/SQL 块上运行的计时统计信息。OFF 禁止每条命令的计时。
- TRIM[OUT] {ON | OFF}删除每行末尾的空格，在利用较慢的通信设备访问 SQL*Plus 时，可以极大地改善性能。OFF 表示允许 SQL*Plus 显示尾部空格。
- TRIMS[POOL] {ON | OFF}删除每行末尾的空格。OFF 允许 SQL*Plus 在每行末尾包含空格。
- UND[ERLINE] {- | c | ON | OFF}设置在 SQL*Plus 报表中用来给列标题添加下划线的字符。
- VER[IFY] {ON | OFF}控制 SQL*Plus 在用值代替替换变量前后是否列出 SQL 语句或 PL/SQL 命令的文本。ON 表示列出文本，OFF 表示不列出文本。
- WRA[P] {ON | OFF}控制如果 SELECT 的行超过了当前行的宽度，SQL*Plus 是否截断 SELECT 行的显示。OFF 表示截断 SELECT 的行，ON 表示允许换到下一行。

描述：SET 打开(ON)或关闭(OFF)某个 SQL*Plus 特性或将其设置为某个值。所有这些特性都由 SET 命令更改并用 SHOW 命令显示。如果 SHOW 单词后面跟着某个命令名，则 SHOW 将显示这个命令；如果只输入单词 SHOW，则显示所有命令。

SET(函数)

参阅：IS A SET、第 39 章。

格式：

```
SET (nested_table)
```

描述：SET 通过消除相同部分将嵌套表转换为集合。函数返回彼此没有相同元素的嵌套表。返回的嵌套表和输入的嵌套表具有相同的类型。

SET CONSTRAINTS

参阅：INTEGRITY CONSTRAINT。

格式：

```
SET { CONSTRAINT | CONSTRAINTS }
    { constraint [, constraint]... | ALL }
    { IMMEDIATE | DEFERRED };
```

描述：SET CONSTRAINTS DEFERRED 告诉 Oracle 在提交事务前不检查约束条件的有效性。这样便能够暂时地推迟数据的完整性检查。在不能保证事务处理的顺序而又需要对多个相关表执行 DML，通常采用延迟约束。用户必须拥有约束所修改的表或在表上具有 SELECT 权限。

默认设置是 SET CONSTRAINTS IMMEDIATE，在这种情况下，一旦在表中插入记录就进行约束检查。

SET ROLE

参阅： ALTER USER、CREATE ROLE、第 20 章。

格式：

```
SET ROLE
    { role [IDENTIFIED BY password] [, role [IDENTIFIED BY password]]...
    | ALL [EXCEPT role [, role]...]
    | NONE };
```

描述： SET ROLE 启用或禁用当前 SQL 会话中授予用户的角色。第 1 个选项使用户启用特定的角色，如果此角色有口令的话，就可以选择性地给定口令(参阅 CREATE ROLE)。第 2 个选项使用户启用除(EXCEPT)特定角色外的所有(ALL)角色。这些角色必须直接授予用户，它们不是通过其他角色授予用户的角色。ALL 选项不允许启用具有口令的角色。第 3 个选项 NONE 禁用当前会话的所有角色。

SET TRANSACTION

参阅： COMMIT、ROLLBACK、SAVEPOINT、TRANSACTION、第 51 章。

格式：

```
SET TRANSACTION
    { { READ { ONLY | WRITE }
      | ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }
      | USE ROLLBACK SEGMENT rollback_segment }
      [NAME 'test']
    | NAME 'test' } ;
```

描述： SET TRANSACTION 启动一个事务。标准的 SQL 事务保证语句级的读一致性，保证来自一个查询的数据在语句执行时是一致的。但某些事务需要更有力的保证，即要求一系列访问一个或多个表的 SELECT 语句同时立即看到所有数据的一致快照。SET TRANSACTION READ ONLY 指定了这种更严格的读一致性。其他用户对查询数据的更改将不会影响该数据的事务视图。在只读事务中，只能使用 SELECT、LOCK TABLE、SET ROLE、ALTER SESSION 和 ALTER SYSTEM 命令。

对于包括 INSERT、UPDATE 或 DELETE 命令的事务，Oracle 给此事务分配回滚段。SET TRANSACTION USE ROLLBACK SEGMENT 指定当前事务将使用特定的回滚段。

SET TRANSACTION 必须是事务处理中的第 1 条 SQL 语句。在 SET TRANSACTION 之前紧挨着放置 COMMIT 可保证 SET TRANSACTION 为事务处理中的第 1 条 SQL 语句。末尾的 COMMIT 释放数据的快照。这很重要，因为数据库需要释放它所保留的资源以维护一致性。

SHOW(SQL*Plus)

参阅： SET。

格式：

```
SHO[W] { option }
```

option 可以是通过 SET 命令设置的任何参数或以下参数：

```
system_variable
ALL
BTI[TLE]
ERR[ORS] [ { FUNCTION | PROCEDURE | PACKAGE | PACKAGE BODY |
  TRIGGER| VIEW | TYPE | TYPE BODY | DIMENSION | JAVA CLASS } [schema.]name]
LNO
PARAMETERS [parameter_name]
PNO
RECYC[LEBIN] [original_name]
REL[EASE]
REPF[OOTER]
REPH[EADER]
SGA
SPOO[L]
SQLCODE
TTI[TLE]
USER
```

描述：SHOW 显示 SET 的某个特性的值、或者 SET 的 ALL 特性的值或其他 SQL*Plus 项的值。这些特性(包括多个 SET 特性)中有多个可跟在单词 SHOW 之后，每个特性都可以单独显示在一行上。返回的行将按字母顺序排列。可以显示的特性是：

- system_variable 是用 SET 命令设置的任意变量集合。
- BTI[TLE]显示当前的 BTITLE 定义。
- ERR[ORS]显示编译存储对象最近遇到的错误。
- LNO 显示当前行号(被显示的当前页中的行)。
- PARAMETERS 显示特定参数的设置，如果不指定任何参数则显示所有参数的设置。
- PNO 显示页码。
- RECYC[LEBIN]显示在回收站中的可以用 FLASHBACK TABLE BEFORE DROP 命令还原的对象的名称。
- REL[EASE]给出这个版本的 Oracle 的修订版本号。
- REPF[OOTER]显示报表的页脚。
- REPH[EADER]显示报表标题。
- SGA 显示 SGA 的当前内存分配。
- SPOO[L]说明输出是否假脱机(ON 或 OFF)。请参阅 SPOOL。
- SQLCODE 显示最近的 Oracle 错误的错误消息编号。
- TTI[TLE]显示当前的 TITLE 定义。
- USER 显示用户的 ID。

SHUTDOWN

参阅：STARTUP、RECOVER、第 51 章。

格式：

```
SHUTDOWN [ABORT|IMMEDIATE|NORMAL|TRANSACTIONAL [LOCAL]]
```

描述：SHUTDOWN 关闭实例，可以选择性地关闭或卸载它。如果使用 ABORT，则没有提交的事务不能回滚。NORMAL(默认)禁止进一步的连接，但等待所有当前用户断开与数据库的连接。IMMEDIATE 不等待当前调用完成或用户断开与数据库的连接。TRANSACTIONAL 允许活动事务完成时执行关闭操作。TRANSACTIONAL LOCAL 仅在本地实例中指定事物的关闭操作。

用户必须通过 AS SYSDBA 或 AS SYSOPER 进行连接才能使用 SHUTDOWN。

SIGN

参阅：+、PRECEDENCE、第 9 章。

格式：

```
SIGN(value)
```

描述：如果 value 为正，则 SIGN 等于 1；如果 value 为负，则 SIGN 等于 -1；如果 value 为零，则 SIGN 等于 0。

示例：

```
SIGN(33)   = 1
SIGN(-.6)  = -1
SIGN(0)    = 0
```

SIN

参阅：ACOS、ASIN、ATAN、ATAN2、COS、COSH、NUMBER FUNCTIONS、SINH、TAN 和第 9 章。

格式：

```
SIN(value)
```

描述：SIN 返回以弧度表示的角度 value 的正弦值。

示例：

```
select SIN(30*3.141593/180) Sine -- sine of 30 degrees in radians
from DUAL;
```

SINH

参阅：ACOS、ASIN、ATAN、ATAN2、COS、NUMBER FUNCTIONS、SIN、TAN 和第 9 章。

格式：

```
SINH(value)
```

描述：SINH 返回角度 value 的双曲正弦。

SOUNDEX

参阅：LIKE、第 7 章和第 27 章。

格式：

```
SOUNDEX(string)
```

描述：SOUNDEX 查找发音(SOUND)与示例字符串(Example string)类似的词。SOUNDEX 对字母及字母组合的通常发音作出某种假设。被比较的两个词必须以相同的字母开头。可对 CONTEXT 索引中的文本字符串内的个别词执行 SOUNDEX 搜索。

示例：

```
select LastName, FirstName, Phone
  from ADDRESS
 where SOUNDEX(LastName)=SOUNDEX('SEPP)

LASTNAME                  FIRSTNAME                 PHONE
------------------------- ------------------------- ------------
SZEP                      FELICIA                   214-555-8383
SEP                       FELICIA                   214-555-8383
```

SPOOL(SQL*Plus)

参阅：SET、第 6 章。

格式：

```
SPO[OL] [file[.ext][CRE[ATE]|REP[LACE]|APP[END]]|OFF|OUT]
```

描述：SPOOL 开始或停止 SQL*Plus 输出到宿主系统文件或系统打印机的假脱机(复制)。SPOOL file 使 SQL*Plus 把所有输出假脱机到指定的文件中。如果不指定文件类型，则 SPOOL 添加默认文件类型，和 SAVE 操作类似，文件类型通常是.lst，但在某些主机上有变化。OFF 表示停止假脱机。OUT 表示停止假脱机并把文件发送到打印机。REPLACE 替换具有相同名字的已存在的文件的内容。为了把输出假脱机到文件中而不显示它，应该在启动文件中的 SQL 语句之前执行 SET TERMOUT OFF，而且在 SPOOL 命令之前。SPOOL 本身显示当前(或最近)的假脱机文件的名称。

SQL CURSOR

参阅：%FOUND、%ISOPEN、%NOTFOUND、%ROWCOUNT、第 32 章。

描述：每当处理的 SQL 语句不是显式指定或打开的游标的一部分时，都要隐式地打开名为 SQL 的游标(参阅 DECLARE)。任何时候只有这样的一个游标。在 INSERT、UPDATE、DELETE(它们与显式游标毫无关系)或者单行 SELECT(碰巧被执行而无显式游标)之前或之后，可通过检查 SQL%FOUND 和 SQL%NOTFOUND 来测试此游标的属性%FOUND 和 %NOTFOUND。

关于这些测试的详细内容请参阅%FOUND。关于这个属性在不同条件下的值的详细内容

请参阅%ROWCOUNT。SQL%ISOPEN 总是为 FALSE，因为 Oracle 在执行 SQL 语句之后自动关闭了 SQL 游标。

SQLCODE

参阅：EXCEPTION、SQLERRM。

格式：

```
variable := SQLCODE
```

描述：SQLCODE 是返回最近的错误号的错误函数，但在异常处理程序(参阅 EXCEPTION)之外总是返回 0。这是因为期望异常的发生(即 SQLCODE 非零时)立即将控制权传递给 PL/SQL 块的 EXCEPTION 部分。一旦控制权到达该处，就可测试 SQLCODE 的值，或用它给某个变量赋值。

SQLCODE 不能用作 SQL 语句的一部分(如将错误代码的值插入错误表中)。但可以设置某个变量等于 SQLCODE，然后在 SQL 语句中使用此变量：

```
sql_error := sqlcode;
insert into PROBLEMLOG values (sql_error);
```

SQLERRM

参阅：EXCEPTION、EXCEPTION_INIT、SQLCODE。

格式：

```
SQLERRM[(integer)]
```

描述：如果不给出 integer，SQLERRM 就返回与当前 SQLCODE 相关的错误消息。如果给出 integer，则返回与该整数值相关的错误消息。与 SQLCODE 一样，这个函数不能直接用在 SQL 语句中，但可以在赋值中使用：

```
error_message := sqlerrm;
```

或者用于检索与错误代码 1403 相关的错误消息：

```
error_message := sqlerrm(1403);
```

在异常处理程序之外(参阅 SQLCODE)，未给出整数参数的这个函数将总是返回“正常，成功完成(normal，successful completion)”。在异常处理程序中，将会得到下面的某个消息：

- 与 Oracle 错误有关的消息。
- 对于显式引发的用户异常，如果未指定消息文本，则得到“用户定义的异常(User-defined exception)”。
- 用 PRAGMA EXCEPTION-INIT 加载的用户定义的消息文本。

SQLLDR

参阅：第 24 章。

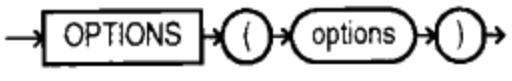
格式：表 A-29 是 SQLLDR 命令的命令行参数：

表 A-29 SQLLDR 命令的命令行参数

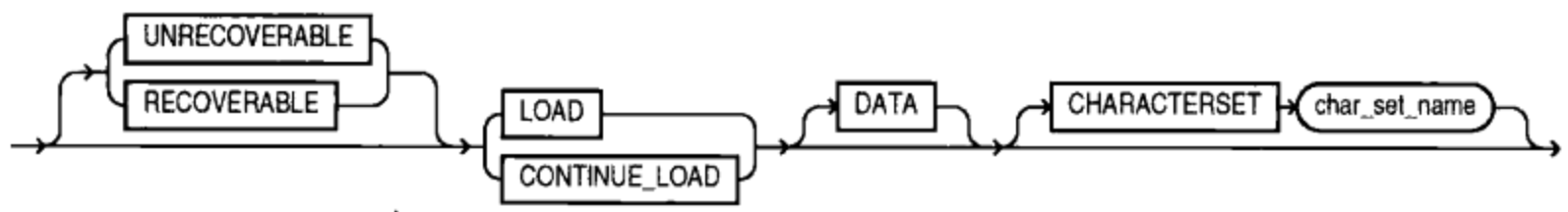
参 数	说 明
Userid	加载操作的用户名和口令，用斜杠分隔
Control	控制文件名
Log	重做日志文件名
Bad	坏文件名
Data	数据文件名
Discard	丢弃文件名
Discardmax	在停止加载前丢弃的最大行数。默认值为允许丢弃所有的行
Skip	在开始加载数据前要跳过的输入文件中的逻辑行数。通常在同一个输入文件部分加载后重新加载期间使用。默认值为 0
Load	要加载的逻辑行数。默认值为全部行
Errors	允许的错误数。默认值为 50
Rows	一次性提交的行数。使用此参数在加载期间分解事务的大小。传统路径加载的默认值是 64，直接路径加载的默认值为全部行
Bindsize	传统路径的绑定数组的大小，以字节为单位。默认值取决于操作系统
Silent	用 Y/N 标号在加载期间禁止消息
Direct	使用直接路径加载。默认值为 FALSE
Parfile	包含其他加载参数说明的参数文件名
Parallel	执行并行加载。默认值为 FALSE
File	从中分配区的文件(针对并行加载)
Skip_Unusable_Indexes	允许加载到在不可用状态下有索引的表中。默认值为 FALSE
Skip_Index_Maintenance	停止直接路径加载的索引维护，使它们处于不可用状态。默认值为 FALSE
Commit_Discontinued	当加载中断后提交已加载的行。默认值为 FALSE
Readsize	读缓冲区的大小，以字节为单位，默认值为 1 048 576
External_Table	用于加载的外部表，默认值为 NOT_USED；其他有效的值为 GENERATE_ONLY 和 EXECUTE
ColumnArrayRows	直接路径列数组的行数，默认值为 5 000
StreamSize	直接路径流缓冲区的大小，以字节为单位，默认值为 256 000
Multithreading	在直接路径中使用多线程
Resumable	启用或禁用当前会话的可恢复操作，默认值为 FALSE
Resumable_Name	标识可恢复语句的文本字符串
Resumable_Timeout	可恢复操作的等待时间，以秒为单位。默认为 7 200 秒
Date_Cache	数据转换缓存的大小，以数据项为单位。默认为 1 000

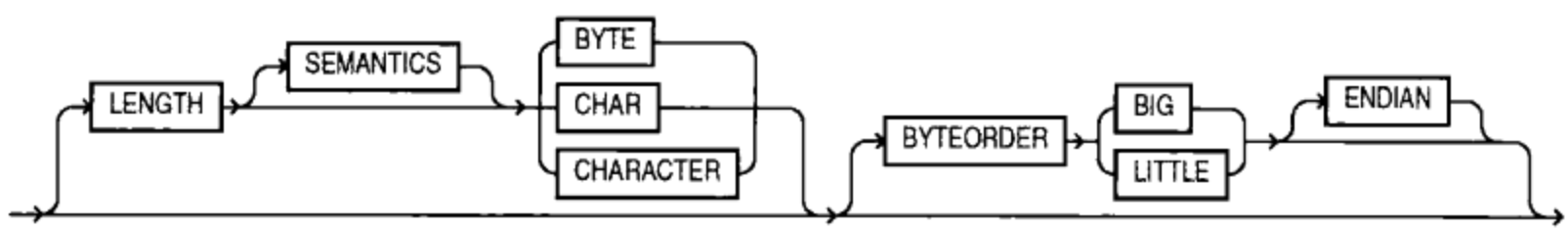
描述：SQLLDR 将数据从外部文件加载到 Oracle 数据库的表中。相应的示例请参阅第 23 章。其控制文件的语法如下：

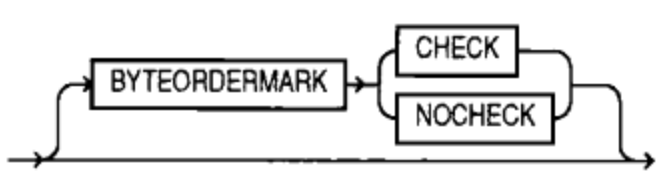
Options clause

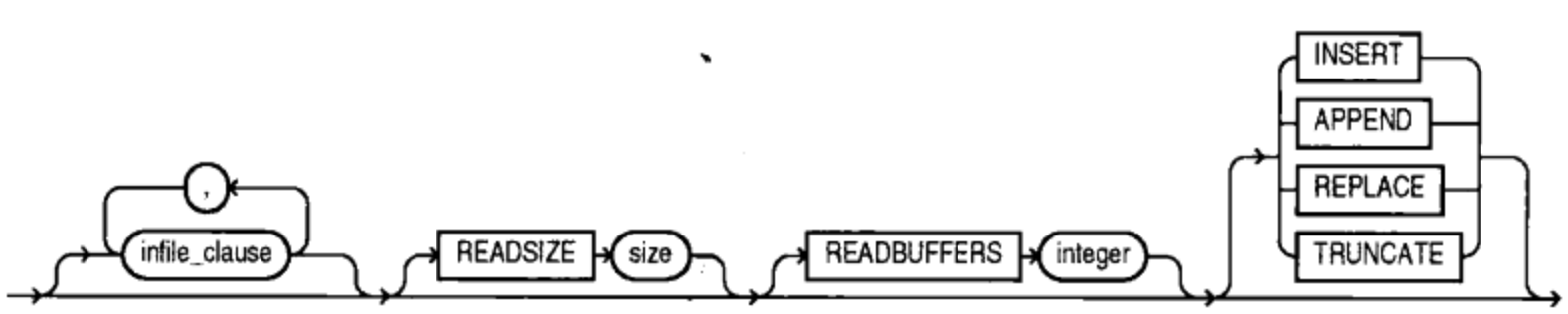


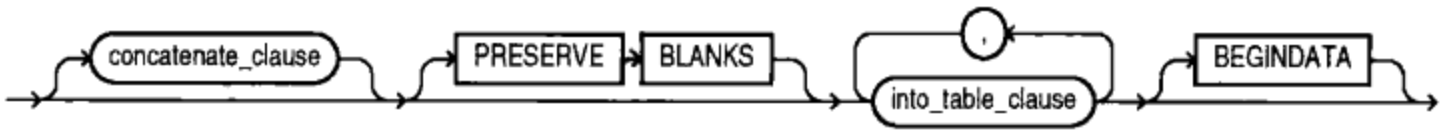
Load statement



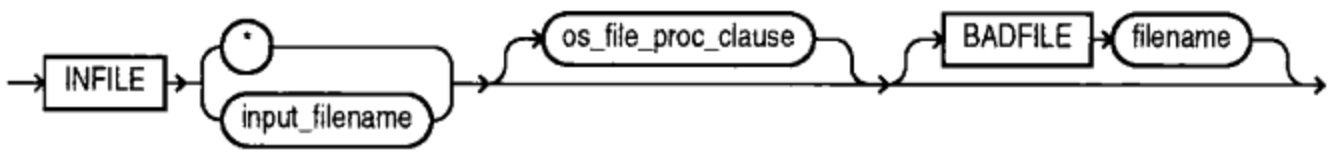


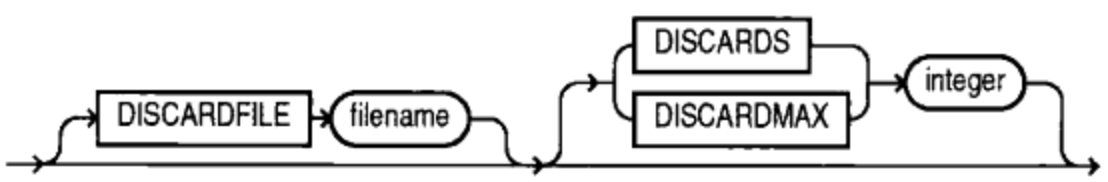


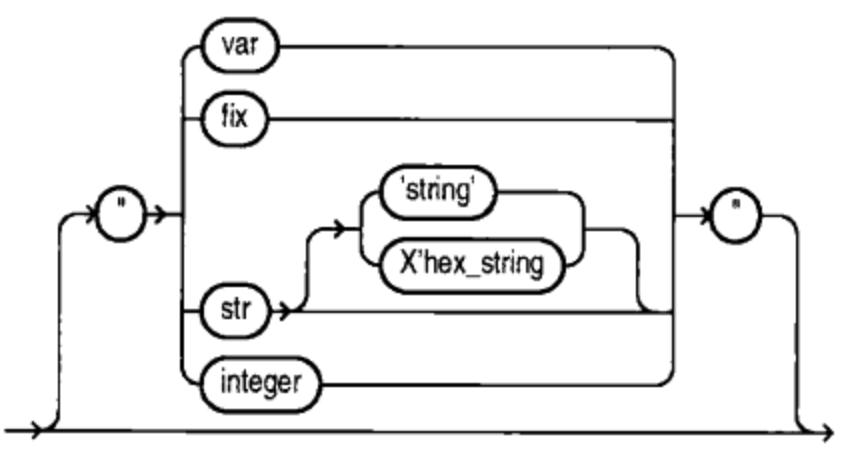




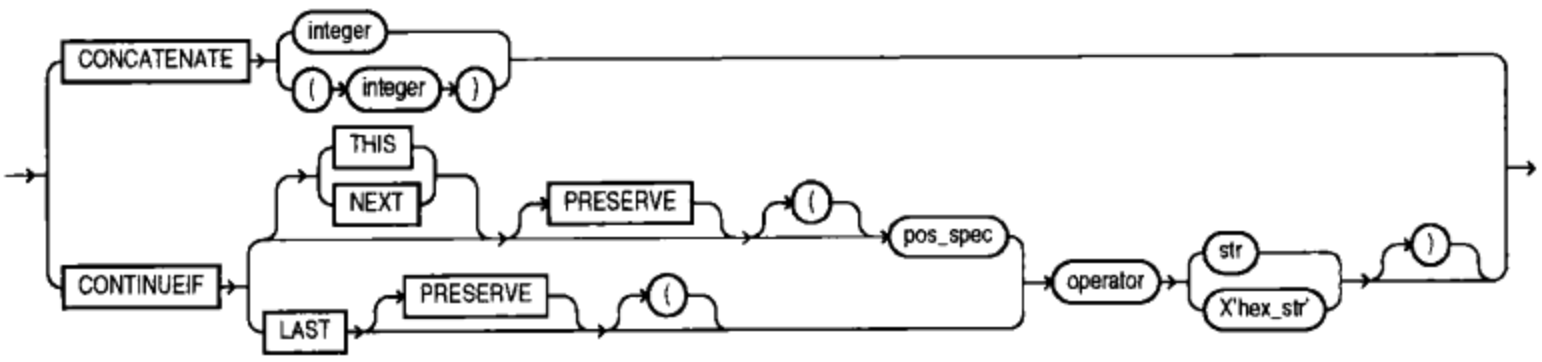
infile_clause



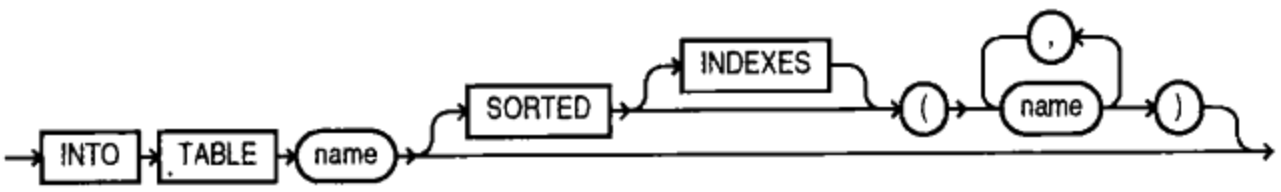


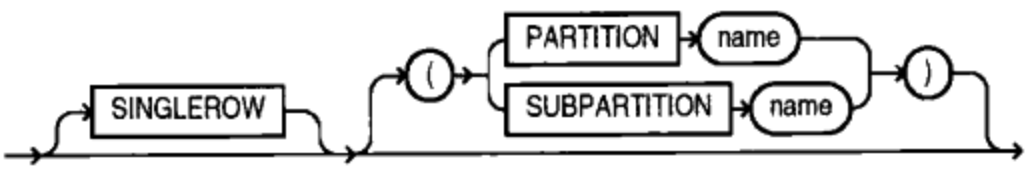


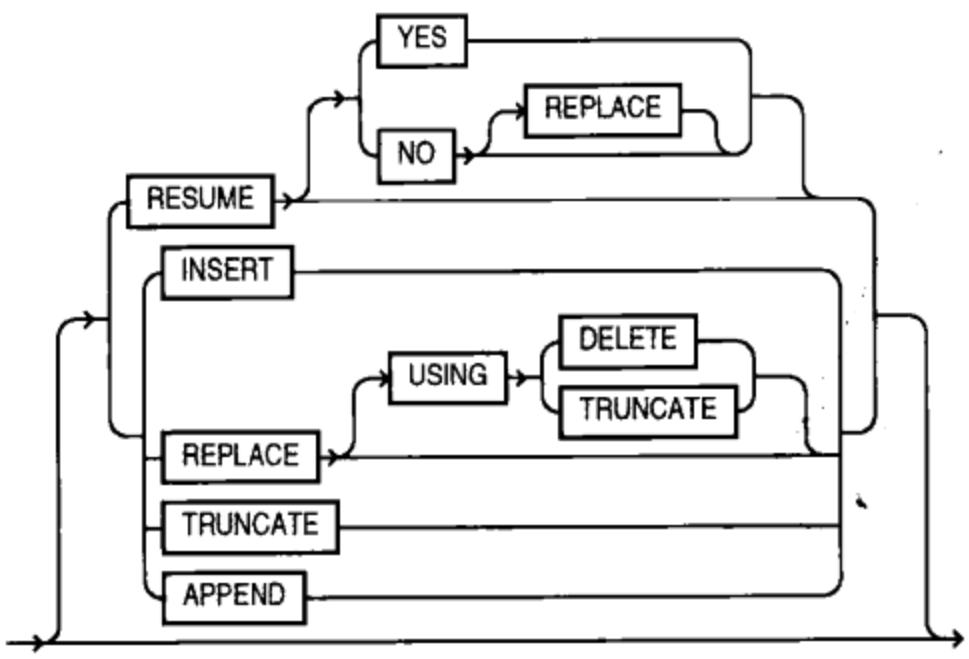
concatenate_clause

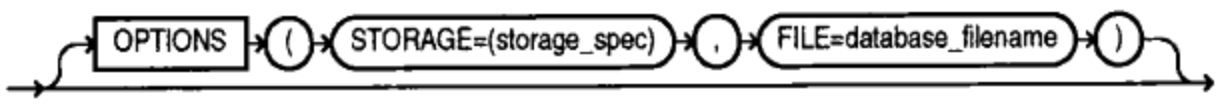


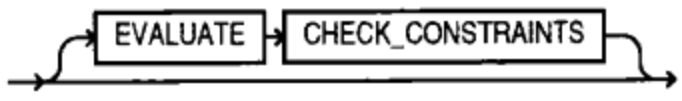
into_table_clause

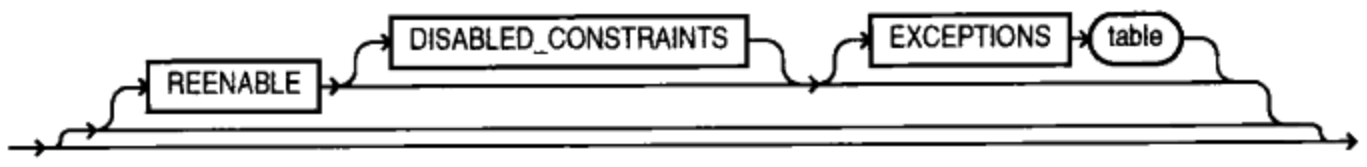


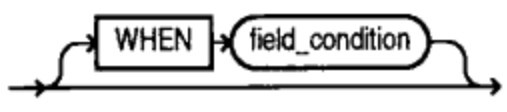


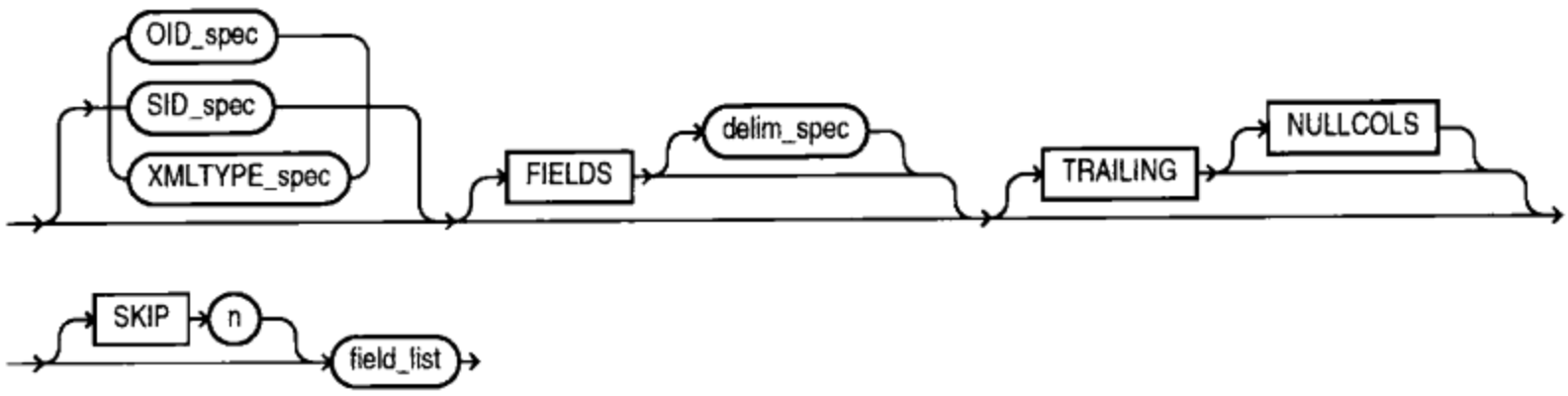




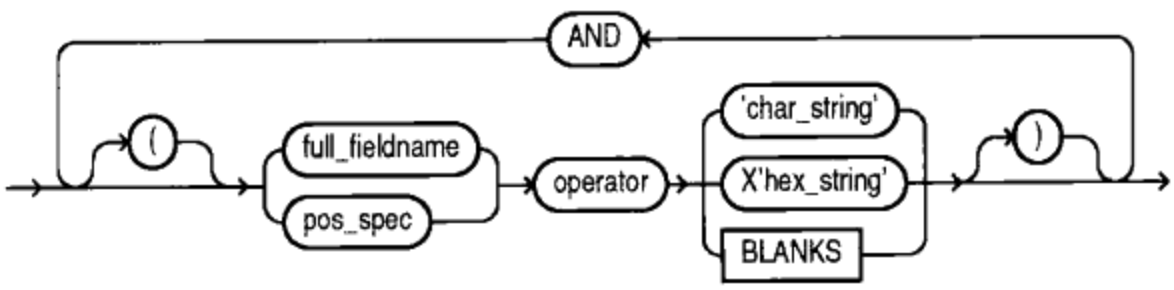




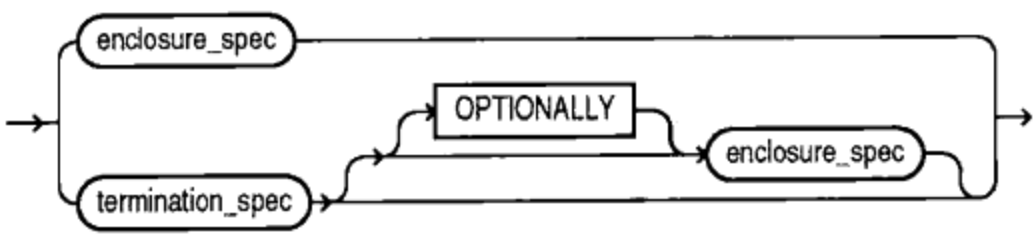




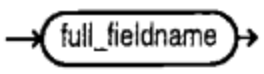
field_condition



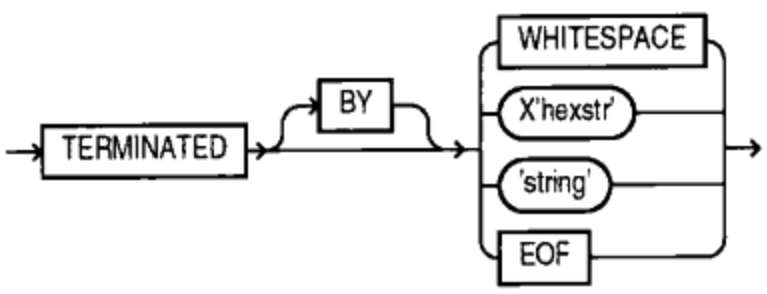
delim_spec



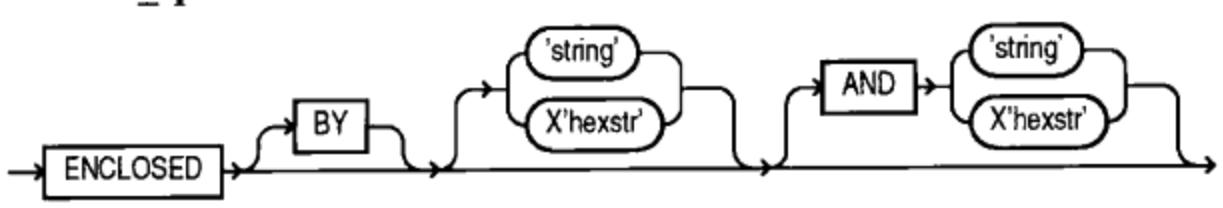
full_fieldname



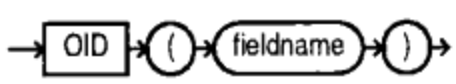
termination_spec



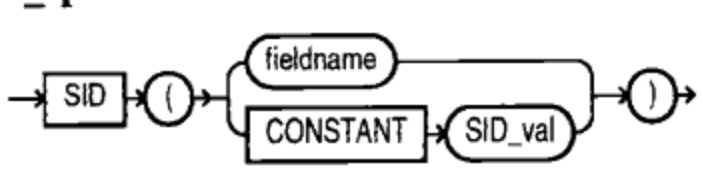
enclosure_spec



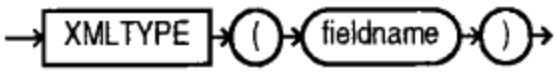
oid_spec



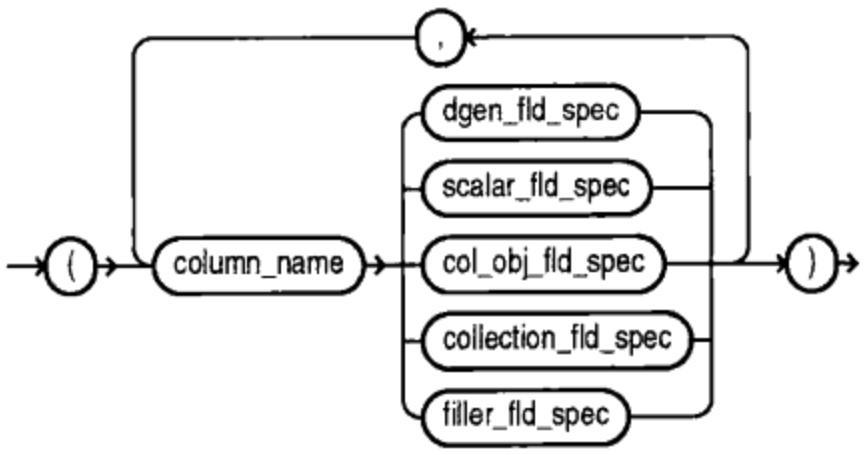
sid_spec



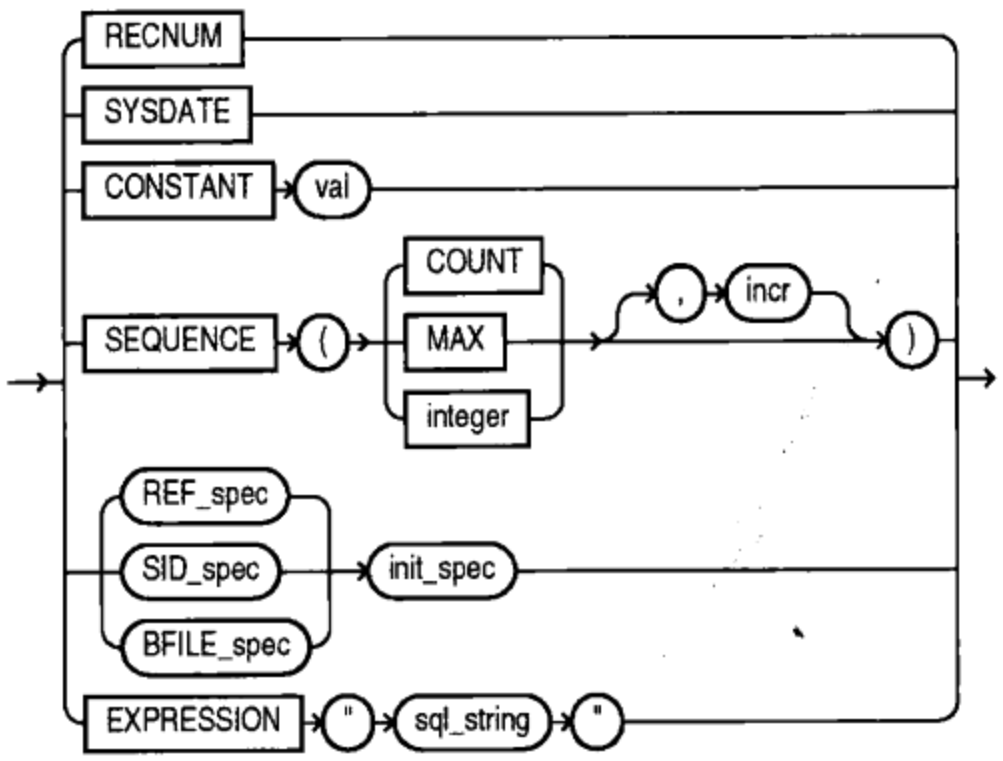
xmltype_spec



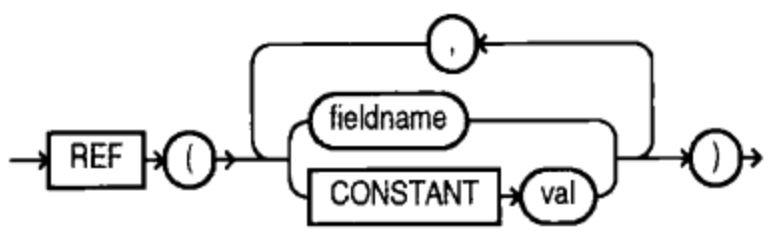
field_list



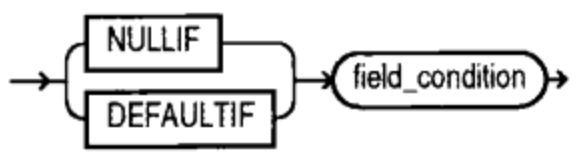
dgen_fld_spec



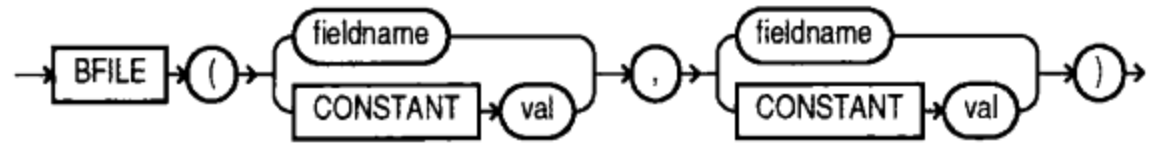
ref_spec



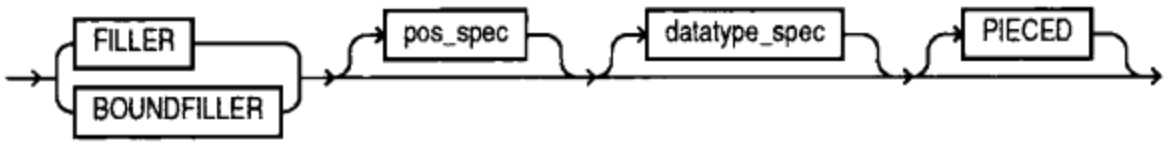
init_spec



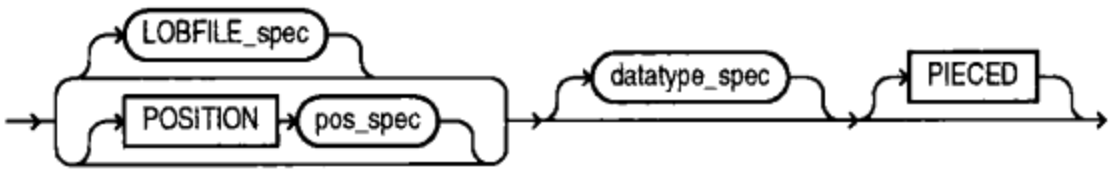
bfile_spec



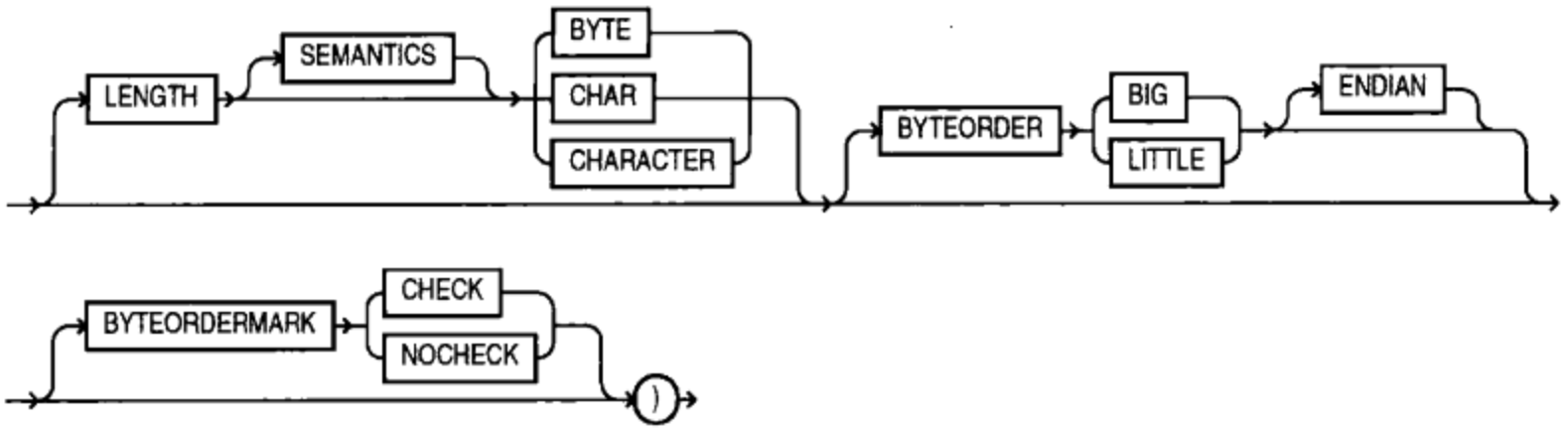
filler_fld_spec



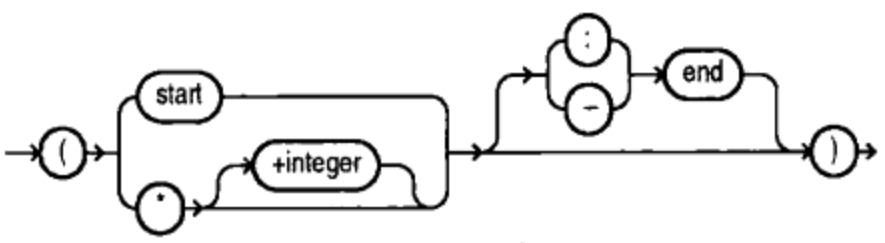
scalar_fld_spec



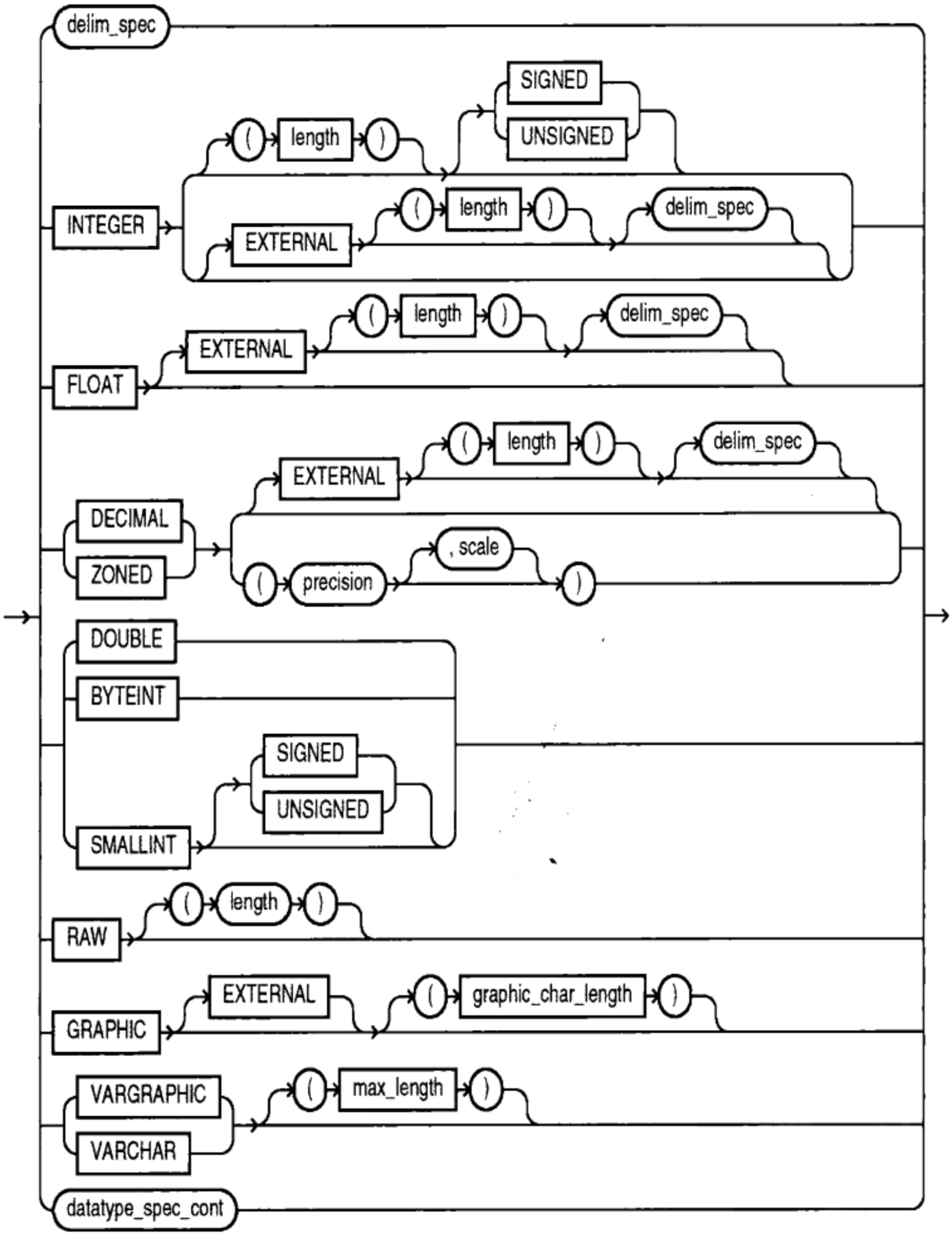
lobfile_spec



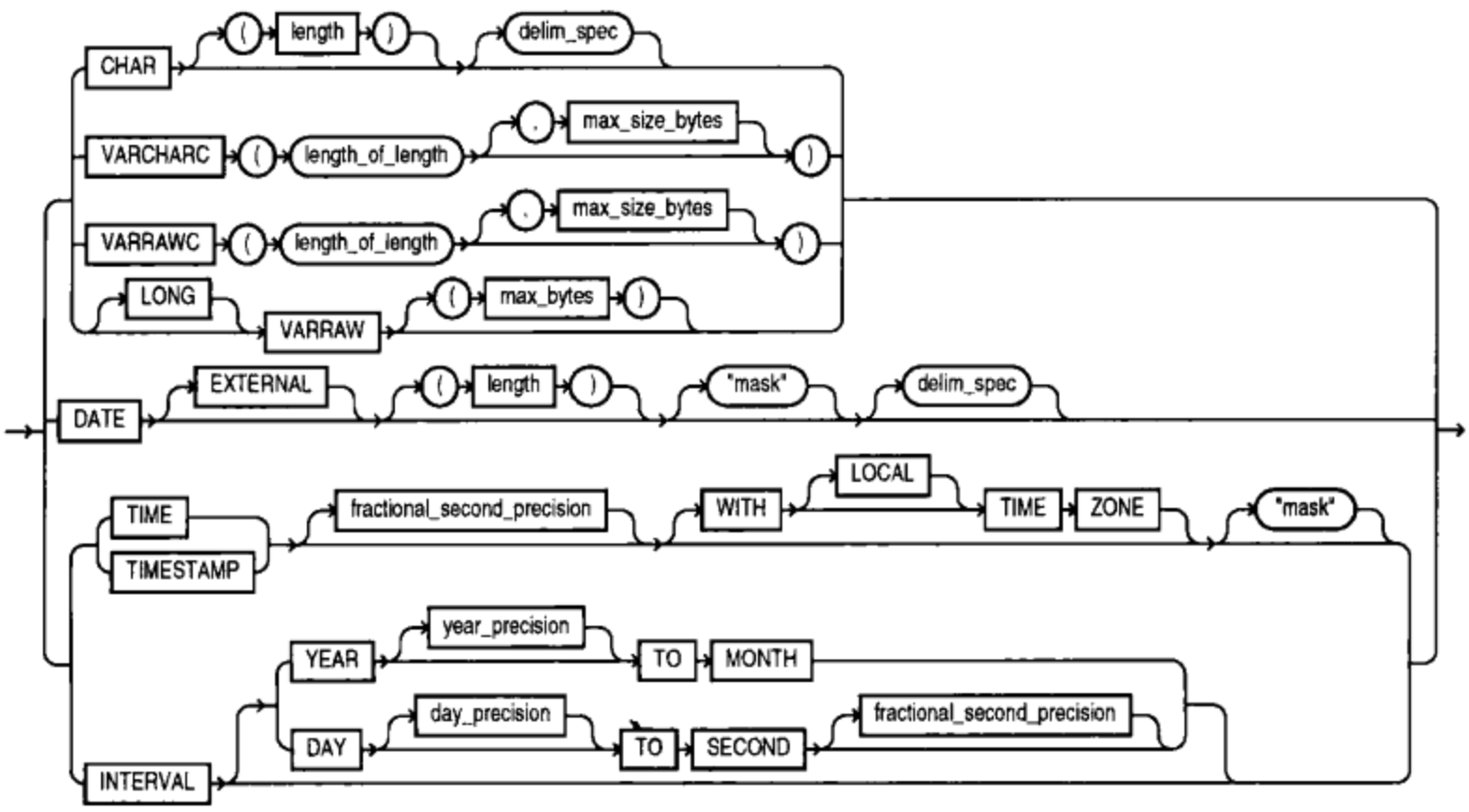
pos_spec



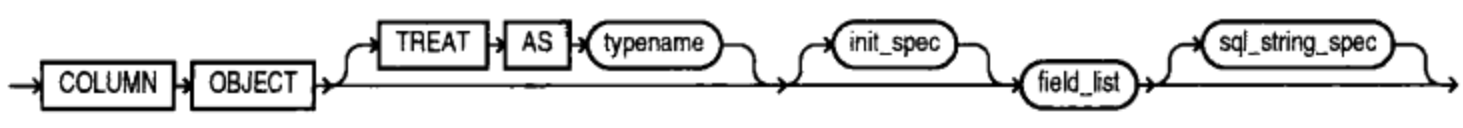
datatype_spec



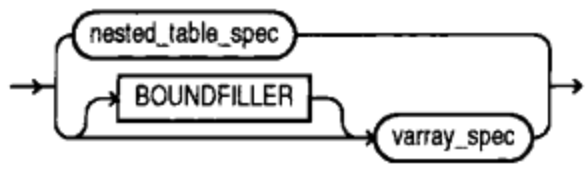
datatype_spec_cont



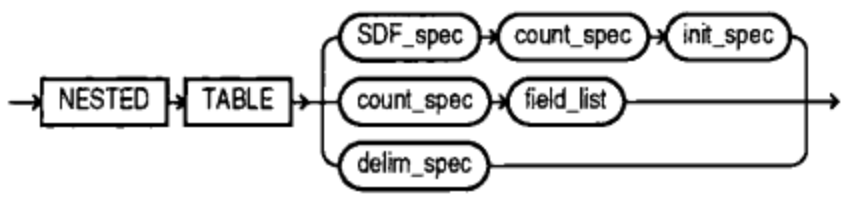
col_obj_fld_spec



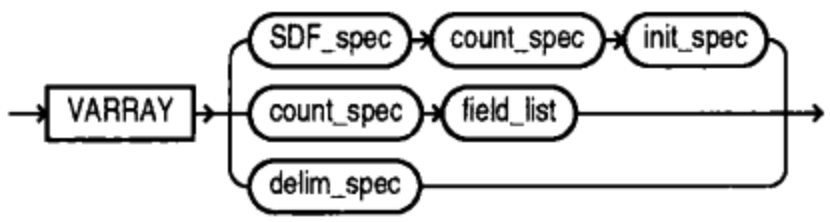
collection_fld_spec



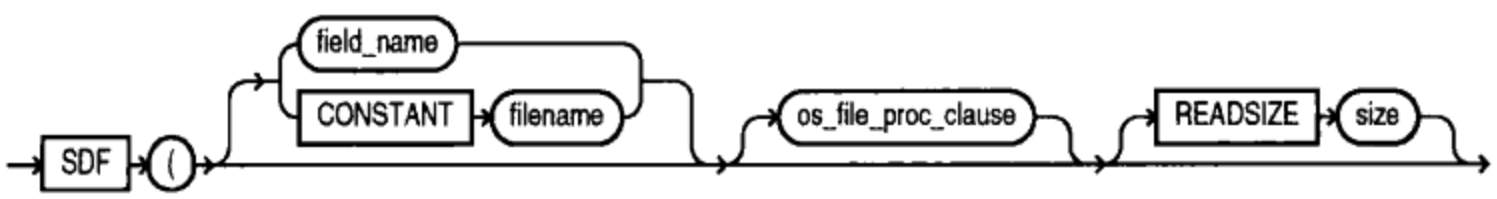
nested_table_spec

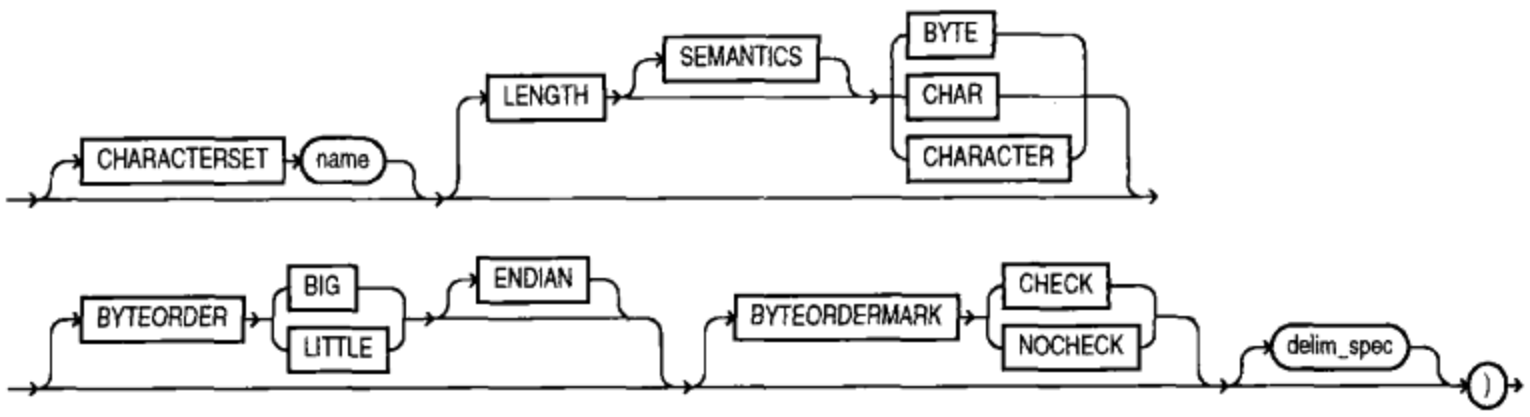


varray_spec

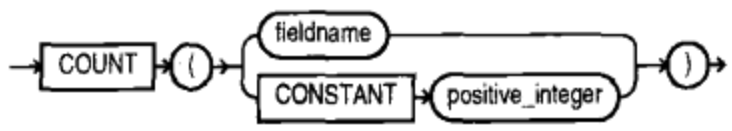


sdf_spec





count_spec



SQLPLUS

参阅：第 6 章。

格式：

```
SQLPLUS [user[/password][@database] [@file] ] [-SILENT] |
  [/NOLOG] [-SILENT] | [-?]
```

描述：SQLPLUS 启动 SQL*Plus。用户输入用户名和口令将登录到默认数据库。只输入用户名，SQL*Plus 会提示输入口令，在输入口令时不显示所输入的内容。输入@database 将连接到指定的数据库而不是默认数据库。@database 可位于任何位置，只要您所登录的计算机通过 Oracle Net 连接到它即可。在口令和@database 之间一定不能有空格。关于 Oracle Net 的更多信息，请参阅第 23 章。@file 将在加载 SQL*Plus 后立即运行启动文件。在@file 之前必须有一个空格。如果没有与 SQLPLUS 一起在命令行上输入用户名和口令，则它们必须是此文件的第 1 行。如果它们在文件中，而在命令行上又输入了它们，则将出现一条错误消息，但启动文件仍然正常运行。要把用户名和口令放在文件的顶部，应该用斜杠(/)分隔它们：

```
GEORGE/MISTY
```

/NOLOG 使 SQL*Plus 开始运行，但不使用户登录到 Oracle 数据库。必须在之后使用 CONNECT 连接到 Oracle(请参阅 CONNECT)。

-SILENT 禁止所有 SQL*Plus 的屏幕显示，包括禁止命令提示以及 SQL*Plus 登录和版权信息。它使其他程序在使用 SQL*Plus 时，程序的用户看不到这种使用。

“-?”显示 SQL*Plus 的当前版本和级别号而实际上不用启动它。

示例：SQL*Plus 的常规启动如下：

```
sqlplus george/misty
```

为在数据库 EDMESTON 上启动，可使用如下命令：

```
sqlplus george/misty@EDMESTON
```

为启动报表文件 REPORT6(它在第 1 行上包含了用户名和口令)，可使用以下命令：

```
sqlplus @report6
```

为在数据库 EDMESTON 上启动报表文件 REPORT6(它在第 1 行上包含了用户名和口令)，可使用以下命令：

```
sqlplus george/misty@EDMESTON @report6
```

SQRT

参阅： POWER。

格式：

```
SQRT(value)
```

描述： SQRT 计算 value 的平方根。

示例：

```
SQRT(64) = 8
```

在 Oracle 中，不能求负数的平方根。

START(SQL*Plus)

参阅： @、@@、ACCEPT、DEFINE、SPOOL、第 6 章。

格式：

```
STA[RT] {url|file[.ext]} [parameter] [parameter]...
```

描述： START 执行指定的启动文件的内容(因为这种文件是用 START 命令执行的，所以称它们为启动文件)。启动文件可包含任意 SQL*Plus 命令。如果不指定文件类型，则 START 假定文件类型为.sql。跟在文件名后的参数被替换到启动文件内的变量中。这些变量必须命名为&1、&2、&3 等，并按顺序从左到右地接收参数。启动文件中每次出现“&1”都会得到第 1 个参数。由多个单词组成的参数应该括在单引号(‘’)中；否则参数限定为 1 个词或数字。

示例：

```
start checkout INNUMERACY
```

其中，文件 checkout.sql 包含变量“&1”，字符串“INNUMERACY”将被替换到其中。

STARTUP

参阅： SHUTDOWN、RECOVER、第 50 章。

格式：

```
STARTUP options | upgrade_options
```

options 的语法如下：

```
[FORCE][RESTRICT][PFILE=filename][QUIET][MOUNT[dbname]|
[OPEN [open_options][dbname]]|NOMOUNT]
```

open_options 的语法如下：

```
READ {ONLY | WRITE [RECOVER]} | RECOVER
```

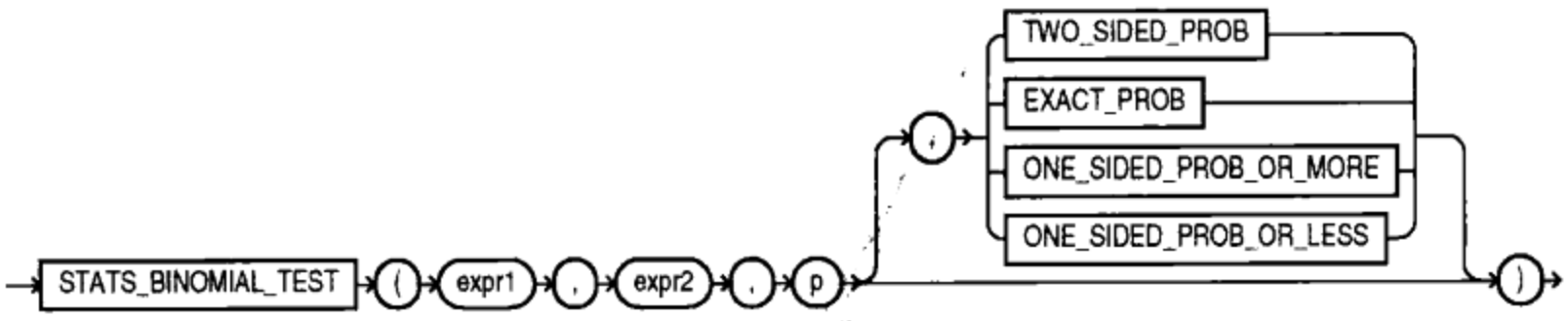
update_options 的语法如下：

```
[PFILE=filename]{UPGRADE|DOWNGRADE} [QUIET]
```

描述：STARTUP 启动一个实例。FORCE 在启动前用 SHUTDOWN ABORT 关闭正在运行的实例。RESTRICT 只允许授权的用户登录。PFILE 选项允许指定在启动时使用的 init.ora 文件(代替标准的 spfile)。STARTUP RECOVER 与执行 RECOVER DATABASE 并启动实例具有同样的作用。UPGRADE 在 OPEN UPGRADE 模式下启动数据库，仅当数据库在新的 Oracle 软件版本中第 1 次启动时使用。

STATS_BINOMIAL_TEST

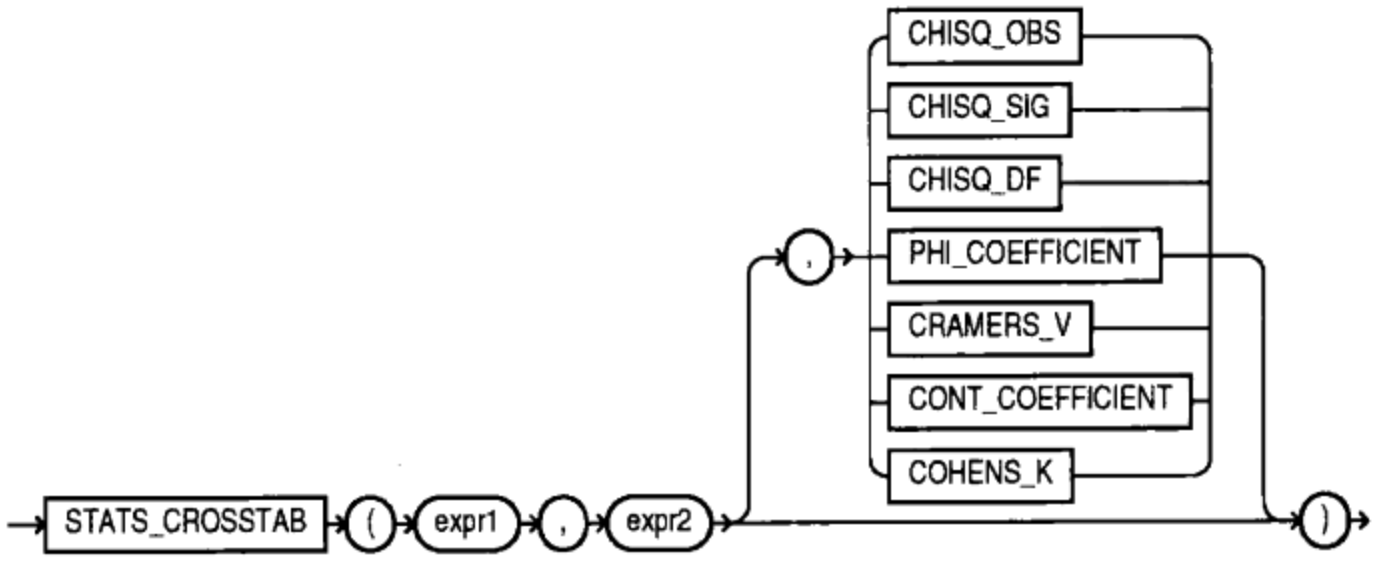
格式：



描述：STATS_BINOMIAL_TEST 用来对二分变量(只存在两个可能的值)进行精确的概率检验。它检验样本部分和给定部分的不同之处。

STATS_CROSSTAB

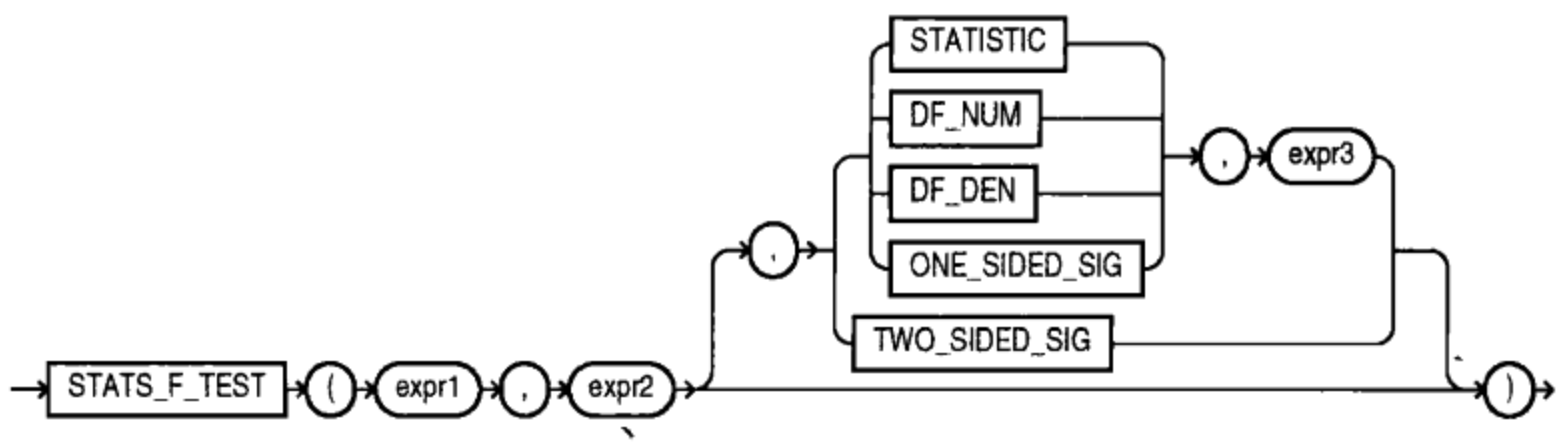
格式：



描述：STATS_CROSSTAB 接受 3 个参数：两个表达式和 1 个 VARCHAR2 类型的返回值。exp1 和 exp2 是被分析的两个名义变量。函数返回一个数字，由第 3 个参数的值确定。第 3 个参数的值可为 CHISQ_OBS、CHISQ_SIG、CHISQ_DF、PHI_COEFFICIENT、CRAMERS_V、CONT_COEFFICIENT 和 COHENS_K。

STATS_F_TEST

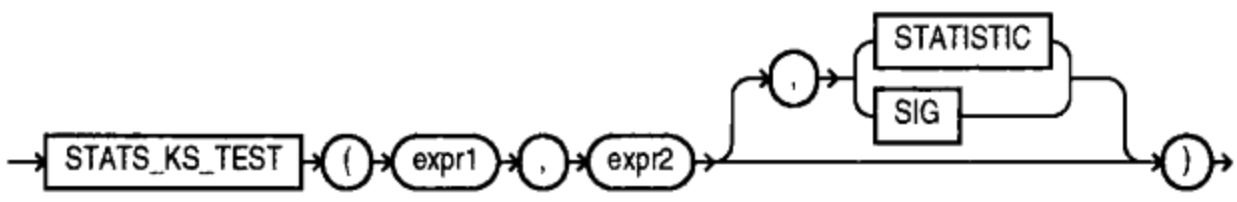
格式：



描述：STATS_F_TEST 检验两个方差是否显著不同。由于 f 观察值是一个方差和另一个方差的比率，因此与 1 相差比较大的值通常暗示存在显著不同。

STATS_KS_TEST

格式：



描述：STATS_KS_TEST 是柯尔莫诺夫-斯米尔诺夫(Kolmogorov-Smirnov)函数，它比较两个样本，检验它们是否来自同一个总体或来自具有相同分布的总体。它不假设获取样本的总体是正态分布的。

STATS_MODE

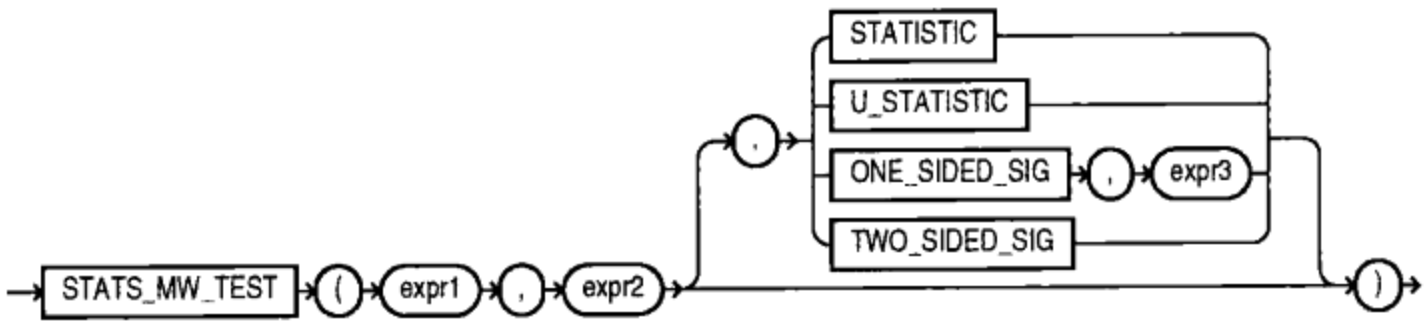
格式：

```
STATS_MODE (expr)
```

描述：STATS_MODE 把一组值当做它的参数并返回出现频率最高的值。如果存在多个模式，则 Oracle 选择其中之一并只返回那个值。

STATS_MW_TEST

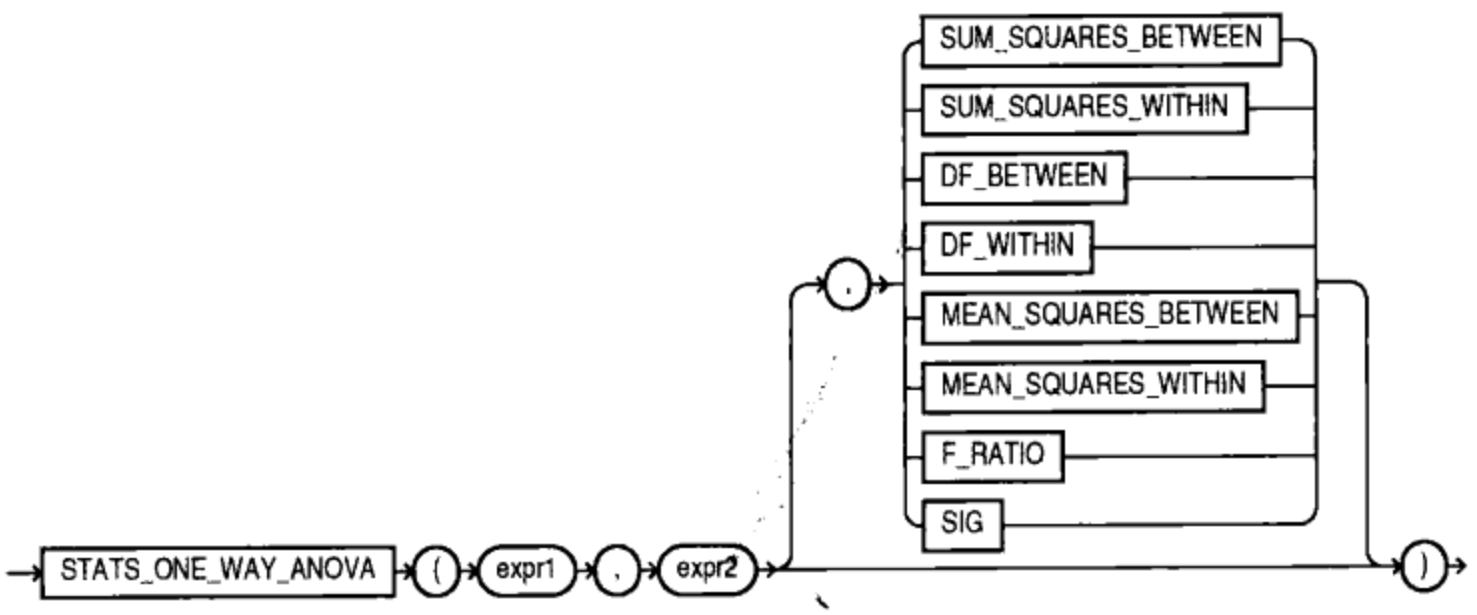
格式：



描述：曼恩-惠特尼(Mann-Whitney)检验比较两个独立样本，检验零假设(这两个总体具有相同的分布函数)而不是备择假设(这两个总体具有两个不同的分布函数)。

STATS_ONE_WAY_ANOVA

格式：

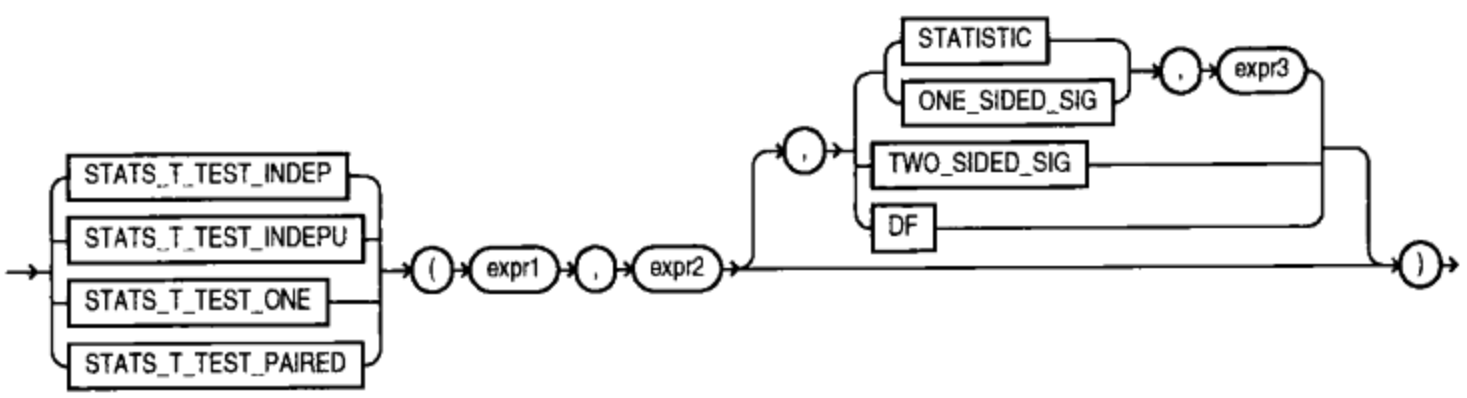


描述：STATS_ONE_WAY_ANOVA 是方差函数的单边分析。它通过比较两个不同的方差估计(均方误差和均方差)检验不同均值的统计显著性。该函数返回一个数字，由第 3 个参数的值确定(默认是 SIG，表示显著性)。

STATS_T_TEST_*函数

格式：

stats_t_test::=



描述：t 检验(t-test)测定不同均值的显著性。可以用它比较两组的均值或具有常数的某一组的均值。STATS_T_TEST_*函数接受 3 个参数：两个表达式和 1 个 VARCHAR2 类型的返回值。该函数返回一个数字，由第 3 个参数的值确定(默认是 TWO_SIDED_SIG)。

检验如下：

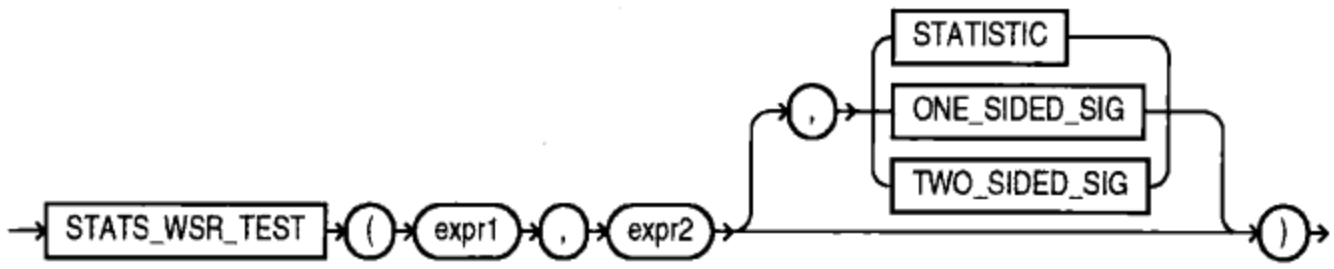
- STATS_T_TEST_ONE　1 个样本的 t 检验
- STATS_T_TEST_PAIRED　两个样本，1 对 t 检验
- STATS_T_TEST_INDEP　具有相同方差的两个独立组的 t 检验
- STATS_T_TEST_INDEPU　具有不同方差的两个独立组的 t 检验

第 3 个参数的值如下：

- STATISTIC　t 的观察值
- DF　自由度
- ONE_SIDED_SIG　t 的单边显著性
- TWO_SIDED_SIG　t 的双边显著性

STATS_WSR_TEST

格式：



描述：STATS_WSR_TEST 是双样本的威尔科克森带符号秩(Wilcoxon Signed Ranks)检验，确定样本之间的不同中值是否与 0 显著不同。如果不指定第 3 个参数，则它的默认值是 TWO_SIDED_SIG。

STDDEV

参阅：GROUP FUNCTIONS、VARIANCE、第 9 章。

格式：

```
STDDEV( [ DISTINCT | ALL ] value) [OVER ( analytic_clause )]
```

描述：STDDEV 给出一组行中额定值的标准偏差。它在计算中忽略 NULL 值。

STDDEV_POP

参阅：AGGREGATE FUNCTIONS、STTDEV、STDDEV_SAMP。

格式：

```
STDDEV_POP ( expr ) [OVER ( analytic_clause )]
```

描述：STDDEV_POP 计算总体标准偏差并返回总体方差的平方根。

STDDEV_SAMP

参阅： AGGREGATE FUNCTIONS、STTDEV、STDDEV_POP。

格式：

```
STDDEV_SAMP ( expr ) [OVER ( analytic_clause )]
```

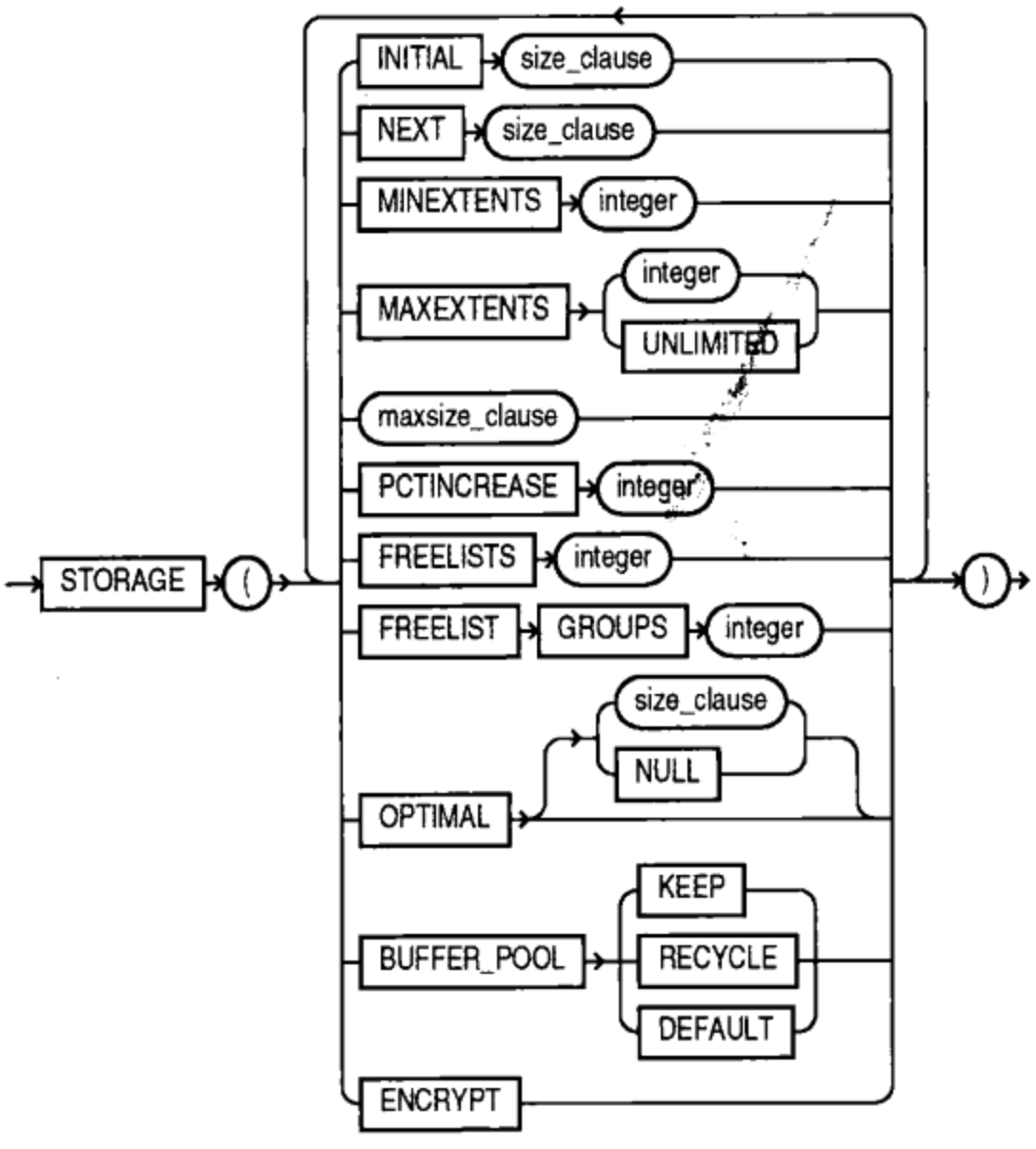
描述： STDDEV_SAMP 计算累积的样本标准偏差并返回样本方差的平方根。

STORAGE

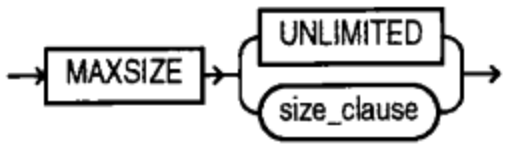
参阅： CREATE CLUSTER、CREATE INDEX、CREATE ROLLBACK SEGMENT、CREATE MATERIALIZED VIEW、CREATE MATERIALIZED VIEW LOG、CREATE TABLE、CREATE TABLESPACE 以及它们相应的 ALTER 语句。

格式：

storage_clause::=



maxsize_clause::=



描述：STORAGE 子句在前面的“参阅”部分列出的任何一个 CREATE 和 ALTER 语句中都是可选的。它不是一条 SQL 语句，不能独立使用。在下面的段落中，块指大小依赖于操作系统及数据库初始化参数的数据库块(参见 BLOCK)。

INITIAL 把空间的第 1 个区分配给对象。如果不指定 INITIAL，则默认使用 5 个数据块。可分配的最小初始区为两个数据块，而最大的区依赖于具体的操作系统：可用一个整数表示它们，也可以用一个整数后跟 K 或 M 来分别表示千字节或兆字节。

NEXT 是初始化区填满后分配的区大小。如果不指定 NEXT，则默认使用 5 个数据块。可分配的下一个最小区是 1 个数据块，最大值依赖于具体的操作系统。可使用 K 和 M 分别表示千字节和兆字节。

PCTINCREASE 控制超过第 2 个区的增长率。如果设置为 0，则每个额外的区将与第 2 个区(由 NEXT 指定)的大小相同。如果 PCTINCREASE 是正整数，则每个后继的区都按这个百分数进行递增。例如，如果 PCTINCREASE 为 50(在不指定时的默认值)，则每个后继的区将比前一个区大 50%。PCTINCREASE 不能为负数。PCTINCREASE 最小为 0，最大值有赖于具体的操作系统。Oracle 四舍五入区的大小为操作系统块大小的下一个倍数。

如果不指定 MINEXTENTS，则 MINEXTENTS 默认值为 1(回滚段为 2)，这表示在创建对象时，只分配初始区。大于 1 的数目将创建许多个区的对象(与每个区本身一样，这些区不需要在磁盘上相邻)，每个区的大小由 INITIAL、NEXT 和 PCTINCREASE 设置的值确定。所有这些区都在创建对象时分配。

MAXEXTENTS 设置可分配的区的总数限制。最小值为 1，这是默认值，最大值依赖于具体的操作系统。可指定 UNLIMITED 的 MAXEXTENTS 值。

OPTIMAL 为回滚段设置以字节为单位的最优大小。Oracle 将动态地释放回滚段中的区以保持最优大小。NULL 表示 Oracle 不能释放回滚段的区，这是默认行为。必须提供大于或等于 MINEXTENTS、INITIAL、NEXT 和 PCTINCREASE 参数分配给回滚段的初始空间的大小。

FREELIST GROUPS 给出可用列表组的数目，默认值为 1。这个设置适用于表、群集或索引的 Oracle 实时应用群集选项。

FREELISTS 设置每个可用列表组的可用列表的。

BUFFER_POOL 设置读入表块的缓冲池。默认时，所有块读入 DEFAULT 池中。可为希望长期在内存中保存的块创建独立的缓冲池(KEEP 池)或为希望快速退出内存的块创建独立的缓冲池(RECYCLE 池)。

对于大多数应用程序，可以使用 Oracle 的自动空间管理功能，包括本地管理的表空间。在本地管理的表空间中，只定义一次存储参数，Oracle 为在表空间中创建的对象管理空间。

STORE(SQL*Plus)

参阅：SAVE、START、第 6 章。

格式：

```
STORE SET file[.ext] [ CRE[ATE] | REP[LACE] | APP[END] ]
```

描述：STORE 将所有 SQL*Plus 环境设置保存到一个文件中。

示例：下面的命令把当前的 SQL*Plus 环境设置保存到文件 settings.sql 中。

```
store settings.sql
```

SUBQUERY

可将查询(即一条 select 语句)作为另一个 SQL 语句(称为“父语句”或“外部语句”)的一部分，包括 CREATE TABLE、DELETE、INSERT、SELECT 和 UPDATE 等，或者在 SQL*Plus 的 COPY 命令中使用，以便定义父语句在其执行时使用的行或列。子查询的结果本身不能显示，而是传递给父 SQL 语句使用。有下面几条规则：

- 在 UPDATE 或 CREATE TABLE 命令中，子查询必须对于插入或更新的每个列返回一个值。然后，父 SQL 语句用这个值或这些值来 INSERT 或 UPDATE 相应的行。
- 子查询不能包含 ORDER BY 和 FOR UPDATE FOR 子句。
- “相关”子查询用在 SELECT 语句的 WHERE 子句中，并引用父 SELECT 命令使用的表的别名。父 SELECT 语句只对为了选择而判定的每行测试一次(标准的子查询仅对父查询判定一次)。更详细的信息请参阅第 13 章。

除了这些限制外，还应遵循 SELECT 语句的常规规则。

SUBSTITUTION

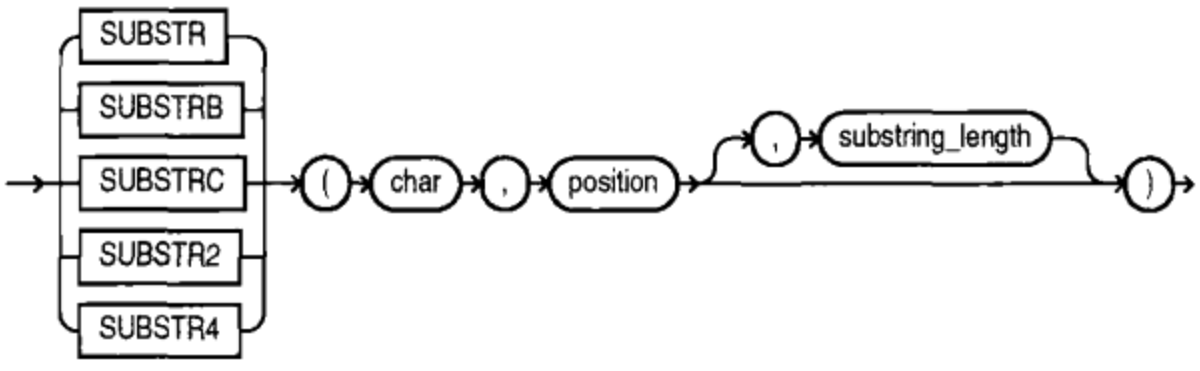
参阅&、&&、ACCEPT、DEFINE。

SUBSTR

参阅：||、CHARACTER FUNCTIONS、INSTR、第 7 章。

格式：

substr::=



描述：SUBSTR 截取从 start 开始包含 count 个字符一个串。如果不指定 count，则截取的忆子串从 start 开始直到串的结束。

示例：

```
SUBSTR('NEWSPAPER',5)
```

结果为：

```
PAPER
```

SUM

参阅： COMPUTE、GROUP FUNCTIONS、第 9 章。

格式：

```
SUM([DISTINCT | ALL] value) [OVER (analytic_clause)]
```

描述： SUM 求一组行的所有 value 的和。DISTINCT 使 SUM 只将唯一的值加到总数中一次，这一般没有多大意义。

SYNTAX OPERATORS

参阅： LOGICAL OPERATORS、PRECEDENCE。

描述： 语法运算符在所有运算符中具有最高的优先级，它们可以出现在 SQL 语句中的任何地方。表 A－30 按优先级的顺序(降序)列出它们。优先级相等的语法运算符从左到右排列。大多数语法运算符都在本附录相应的条目中进行介绍。

表 A-30　语法运算符

运　算　符	功　能
-	继续 SQL*Plus 命令。在下一行继续一条命令
&	SQL*Plus 启动文件中参数的前缀。可以用单词替代&1、&2 等等。详细内容请参阅 START
& &&	SQL*Plus 的 SQL 命令中替换变量的前缀。如果找到未定义的“&”或“&&”变量，则 SQL*Plus 将提示输入一个值。“&&”也定义变量并保存其值，而“&”不这样。详细内容请参阅“&”和“&&”、DEFINE 和 ACCEPT
:	SQL*FORMS 中的变量的前缀并用于 PL/SQL 中的宿主变量
.	变量分隔符，在 SQL*Plus 中用来分隔变量名与后缀，以便不会将后缀视为变量名的组成部分，在 SQL 中用于用户、表和列名之间
()	括住子查询和列列表或控制优先级。
'	括住一个字面量，如一个字符串或日期常量。为了在串常量中使用单引号，应该使用两个单引号(不是双引号)
"	括住包含特殊字符或空格的表别名或列别名
"	在 TO_CHAR 的日期格式子句中括住字面量文本
@	放置于 COPY 子句中的数据库名之前或 FROM 子句中的链接名之前

SYS_CONNECT_BY_PATH

参阅： CONNECT BY、第 14 章。

格式：

```
SYS_CONNECT_BY_PATH (column, char)
```

描述： SYS_CONNECT_BY_PATH 只在分层查询中有效。它返回从根到节点的列值的路径，以及按 CONNECT BY 条件返回的每一行由 char 分隔的列值。

SYS_CONTEXT

参阅： CREATE CONTEXT。

格式：

```
SYS_CONTEXT ( namespace , parameter [, length] )
```

描述： SYS_CONTEXT 返回与上下文 namespace 相关联的 parameter 值。

SYS_DBURIGEN

参阅： 第 52 章。

格式：

```
SYS_DBURIGEN
    ( { column | attribute } [rowid]
      [, { column | attribute } [rowid]]...
      [, 'text( )'] )
```

描述： SYS _DBURIGEN 以一个或多个列或属性以及一个可选的 ROWID 作为它的参数，生成对应于特定列或行对象的 DBUriType 数据类型的 URL。然后可利用这个 URL 从数据库中检索 XML 文档。

引用的所有列或属性必须驻留在相同的表中。它们必须执行一个主键的函数。也就是说，它们实际上不需要与表的主键匹配，但它们必须引用一个唯一值。如果指定多列，则除最后一列外的所有列都标识数据库中的行，而指定的最后一列标识行中的列。

默认情况下，此 URL 指向格式化的 XML 文档。如果希望此 URL 仅指向文档的文本，则应该指定可选的‘text ()’(在这个 XML 上下文中，小写的‘text’是关键字而不是语法占位符)。

SYS_EXTRACT_UTC

参阅： 第 10 章。

格式：

```
SYS_EXTRACT_UTC ( datetime_with_timezone )
```

描述： SYS_EXTRACT_UTC 利用时区差从一个日期时间中提取 UTC(世界标准时间)。

SYS_GUID

格式：

```
SYS_GUID ( )
```

描述： SYS_GUID 生成并返回由 16 字节组成的全局唯一标识符(RAW 值)。在大多数平台上，所生成的这个标识符由一个主标识符和调用此函数的进程或线程的进程标识符或线程标识符，及该进程或线程的不重复的值(字节序列)组成。

SYS_TYPEID

参阅：第 38 章。

格式：

```
SYS_TYPEID ( object_type_value )
```

描述：SYS_TYPEID 返回最特殊的操作数类型的类型标识符(type ID)。这个值主要用来识别一个可替换列下面的类型判别列。例如，可利用 SYS_TYPEID 返回的值在类型判别列上构建一个索引。

SYS_XMLAGG

参阅：SYS_XMLGEN、第 45 章。

格式：

```
SYS_XMLAGG ( expr [fmt] )
```

描述：SYS_XMLAGG 聚集所有由 **expr** 表示的 XML 文档或片段，并生成一个 XML 文档。它添加具有默认名 ROWSET 的封闭元素。如果希望格式化此 XML 文档为其他格式，就可指定 fmt，它是 SYS.XMLGenFormatType 对象的一个实例。

SYS_XMLGEN

参阅：SYS_XMLAGG、第 52 章。

格式：

```
SYS_XMLGEN ( expr [fmt] )
```

描述：SYS_XMLGEN 以一个表达式(值为数据库的特定的列或行)作为参数，并返回包含一个 XML 文档的 SYS.XMLType 类型的实例。expr 可以是标量值、用户定义的类型或 XMLType 实例。

如果 expr 是标量值，则此函数返回包含此标量值的一个 XML 元素。如果 expr 为一个类型，则此函数将用户定义的类型属性映射到 XML 元素。如果 expr 是一个 XMLType 实例，则此函数将文档括在一个默认标记名为 ROW 的 XML 元素之中。

默认情况下，XML 文档的元素与 expr 的元素相匹配。例如，如果 expr 解析为一个列名，则括住的 XML 元素将与列名相同。如果希望将 XML 文档格式化为不同的格式，则可指定 fmt，它是 SYS.XMLGenFormatType 对象的一个实例。

SYSDATE

参阅：第 10 章。

格式：

```
SYSDATE
```

描述：SYSDATE 返回当前的日期和时间。

示例：

```
select SYSDATE from DUAL;
```

SYSTIMESTAMP

参阅：第 10 章。

格式：

```
SYSTIMESTAMP
```

描述：SYSTIMESTAMP 返回当前的日期和时间。

示例：

```
select SYSTIMESTAMP from DUAL;
```

TABLE(PL/SQL)

参阅：DATA TYPES、RECORD(PL/SQL)。

格式：

```
TYPE new_type IS TABLE OF
{type | table.column%TYPE}[NOT NULL]
INDEX BY BINARY_INTEGER;
```

描述：TABLE 声明一种以后可用来声明此种类型变量的新类型。PL/SQL 表有一列和一个整数键，并且可具有任意数目的行。该列的数据类型可以是其中一个标准的 PL/SQL 数据类型(包括其他 RECORD 但不是一个 TABLE)，也可以是指定数据库表中特定列的一个类型引用。每个字段还可以有一个 NOT NULL 限定符，它指定该字段必须总是有一个非空值。

INDEX BY BINARY_INTEGER 子句是必需的，且提示此索引是一个整数。可以在圆括号中使用这个索引来引用该表的任意行。

如果引用没有赋予任何数据的索引，则 PL/SQL 将引发 NO_DATA_FOUND 异常。

TAN

参阅：ACOS、ASIN、ATAN、ATAN2、COS、COSH、NUMBER FUNCTIONS、SIN、TANH、第 9 章。

格式：

```
TAN(value)
```

描述：TAN 返回用弧度表示的角度值 value 的正切。

示例：

```
select TAN(135*3.141593/180) Tan -- tangent of 135 degrees in radians
```

```
from DUAL;
```

TANH

参阅：ACOS、ASIN、ATAN、ATAN2、COS、COSH、NUMBER FUNCTIONS、SIN、TAN、第 9 章。

格式：

```
TANH(value)
```

描述：TANH 返回角度 value 的双曲正切。

TIMESTAMP

TIMESTAMP(fractional_second_precision)数据类型记录日期的年、月和日的值，以及时间的时、分、秒的值，其中 fractional_second_precision 是 SECOND 日期时间字段的小数部分的数字位数。fractional_second_precision 所接受的值为 0~9。默认值是 6。详细内容请参阅 DATATYPES 和第 10 章。

TIMESTAMP WITH TIME ZONE

TIMESTAMP(fractional_second_precision)WITH TIME ZONE 数据类型记录日期的年、月和日的值，以及时间的时、分、秒的值，其中 fractional_second_precision 为 SECOND 日期时间字段的小数部分的数字位数。fractional_second_precision 所接受的值为 0~9。默认值为 6。

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP(fractional_second_precision)WITH LOCAL TIME ZONE 数据类型记录 TIMESTAMP WITH TIME ZONE 的值，在存储数据时规范化为数据库的时区。在检索数据时，用户看到的是会话时区中的数据。

TIMING(SQL*Plus)

参阅：CLEAR、SET。

格式：

```
TIMI[NG] [ START area | STOP | SHOW ];
```

描述：TIMING 根据区域(area)名记录从 START 到 STOP 之间的时间。START 打开一个计时区域并将 area 作为其标题。area 必须是一个单字。可以同时存在几个区域。最近创建的区域在删除前为当前计时区域，删除它之后，以前创建的计时区域成为当前计时区域。实际上计时区域可以嵌套。最近创建的区域显示的时间量总是最少，因为根据定义，在它以前创建的计时区域经历的时间较长。任意计时区域的总时间为它的净时间加上它后面的所有计时区域的时间。

SHOW 给出当前计时区域的名称和经历的时间。

STOP 给出当前计时区域的名称和经历的时间、删除此计时区域并使它前面的计时区域(如果有的话)成为当前计时区域。关于具体机器上时间的精确含义的详细内容，请参阅相应的主机操作系统的 SQL*Plus User's Guide and Reference。

示例：为了创建一个名为1的计时区域，可以输入：

```
timing start 1
```

为了查看当前计时区域的名称和时间，但允许它继续运行，可以输入：

```
timing show
```

为了查看当前计时区的名称和累计时间，停止它，并使前一个计时区成为当前计时区，可用以下代码：

```
timing stop
```

TO_BINARY_DOUBLE

参阅：TO_BINARY_FLOAT。

格式：

```
TO_BINARY_DOUBLE (expr [,fmt [,'nlsparam']] )
```

描述：TO_BINARY_DOUBLE 返回双精度浮点数。expr 可以是字符串或 NUMBER 类型的数值。

TO_BINARY_FLOAT

参阅：TO_BINARY_DOUBLE。

格式：

```
TO_BINARY_FLOAT (expr [,fmt [,'nlsparam']] )
```

描述：TO_BINARY_FLOAT 返回单精度浮点数。expr 可以是字符串或 NUMBER 类型的数值。

TO_CHAR(字符格式)

参阅：TO_CHAR(日期和数值格式)。

格式：

```
TO_CHAR ( nchar | clob | nclob )
```

描述：TO_CHAR(character)函数将 NCHAR、NVARCHAR2、CLOB 或 NCLOB 数据转换为数据库字符集。

示例：

```
SELECT TO_CHAR(Title) FROM BOOKSHELF;
```

TO_CHAR(日期和数值格式)

参阅：DATE FORMATS、DATE FUNCTIONS、NUMBER FORMATS、To_DATE、第 10 章

格式：

```
TO_CHAR(date| interval,'format'[,'nlsparam'])
TO_CHAR(number,'format'[,'nlsparam'])
```

描述：TO_CHAR 根据 format 重新格式化数值或日期。date 必须是定义为 Oracle 的 DATE 数据类型的一列。它不能为字符串，即使是默认日期格式 DD_MON_YY 也不行。在 TO_CHAR 函数中使用字符串(date 出现在此字符串中)的唯一方法是把它括在 TO_DATE 函数中。此函数的可能的格式有很多种，这些格式列在 NUMBER FORMATS 和 DATE FORMATS 下面。

示例：请注意下面示例中的格式。在 SQL*Plus 中，为取消 TO_CHAR 生成的当前默认日期宽度(大约为 100 个字符宽)，这是必需的：

```
column Formatted format a30 word_wrapped heading 'Formatted'

select BirthDate, TO_CHAR(BirthDate,'MM/DD/YY') Formatted
  from BIRTHDAY
 where FirstName = 'VICTORIA';

BIRTHDATE   Formatted
---------   ------------------------------
20-MAY-49   05/20/49
```

TO_CLOB

参阅：第 40 章。

格式：

```
TO_CLOB ( lob_column | char )
```

描述：TO_CLOB 将一个 LOB 列或其他字符串中的 NCLOB 值转换为 CLOB 值。char 可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2、CLOB 或 NCLOB 中任何一种数据类型。Oracle 通过把基本 LOB 数据从国家语言字符集转换为数据库字符集来执行这个函数。请注意，LONG 值可通过 ALTER TABLE 直接转换为 CLOB 列。

TO_DATE

参阅：DATE FORMATS、DATE FUNCTIONS、TO_CHAR、第 10 章。

格式：

```
TO_DATE( string,'format'[,'nlsparam'])
```

描述：TO_DATE 以给定的格式将字符串(string)转换为 Oracle 日期。它也接受字符串之外的数值，但有一定的限制。string 是字面量字符串、字面量数值或包含字符串或数值的数据库列。除一种情况外，在所有情况下的格式必须对应于 format 所描述的格式。只有在字符串是 'DD-MON-YY' 格式时，才可以省略格式。请注意，format 是受限制的。关于 TO_DATE 可接收的格式请参阅 DATE FORMATS。

示例：

```
select TO_DATE('02/22/04','MM/DD/YY') from DUAL;

TO_DATE('
---------
22-FEB-04
```

TO_DSINTERVAL

参阅： DATATYPES。

格式：

```
TO_DSINTERVAL ( char ['nlsparam'] )
```

描述： TO_DSINTERVAL 把 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 INTERVAL DAY TO SECOND 类型。char 为要转换的字符串。可在这个函数中指定的唯一合法的 NLS 参数是 NLS_NUMERIC_CHARACTERS。这个参数可具有：NLS_NUMERIC_CHARACTERS = "dg" 格式，其中 d 和 g 分别表示十进制字符和组分隔符。

TO_LOB

参阅： INSERT、第 40 章。

格式：

```
TO_LOB(long_column)
```

描述： TO_LOB 将 long_column 中的 LONG 值转换为 LOB 值。这个函数只能应用于 LONG 列，并且只在 INSERT 命令的子查询的 SELECT 列表中使用。

TO_MULTI_BYTE

参阅： CONVERSION FUNCTIONS、第 11 章。

格式：

```
TO_MULTI_BYTE(string)
```

描述： TO_MULTI_BYTE 将字符串 string 中的单字节字符转换为等价的多字节字符。如果某个字符没有等价的多字节字符，则函数返回未经转换的字符。

TO_NCHAR(字符)

参阅： CONVERSION FUNCTIONS、TRANSLATE…USING。

格式：

```
TO_NCHAR ( { char | clob | nclob} [, fmt [, 'nlsparam']] )
```

描述： TO_NCHAR(character)将字符串、CLOB 或 NCLOB 从数据库字符集转换为国家语言字符集。这个函数等价于国家语言字符集中带有 USING 子句的 TRANSLATE…USING 函数。

TO_NCHAR(日期时间)

参阅：CONVERSION FUNCTIONS。

格式：

```
TO_NCHAR ( { datetime | interval } [, fmt [, 'nlsparam']] )
```

描述：TO_NCHAR(datatime)将 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL MONTH TO YEAR 或 INTERVAL DAY TO SECOND 数据类型的字符串从数据库字符集转换为国家语言字符集。

TO_NCHAR(数值)

参阅：CONVERSION FUNCTIONS。

格式：

```
TO_NCHAR ( n [, fmt [, 'nlsparam']] )
```

描述：TO_NCHAR(number)将数值转换为 NVARCHAR2 字符集中的字符串。对应于 n 的可选的 fmt 和 nlsparam 可以是 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL MONTH TO YEAR 或 INTERVAL DAY TO SECOND 数据类型。

TO_NCLOB

参阅：CONVERSION FUNCTIONS、第 40 章。

格式：

```
TO_NCLOB ( lob_column | char )
```

描述：TO_NCLOB 将 LOB 列中的 CLOB 值或者其他字符串转换为 NCLOB 值。char 可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2、CLOB 或 NCLOB 数据类型。Oracle 通过将 LOB 列中的字符集从数据库字符集转换为国家语言字符集来实现这个函数。

TO_NUMBER

参阅：CONVERSION FUNCTIONS、第 11 章。

格式：

```
TO_NUMBER( string [, fmt [, ' nlsparam']] )
```

描述：TO_NUMBER 将字符串 string 转换为 NUMBER 数据类型。它要求串是只包含字符 0~9、一、十和“.”的正确格式化的数值。由于 Oracle 自动进行数据转换，因此除了在 ORDER BY 或比较中用于包含数字的字符列外，基本上不需要这个函数。

示例：

```
TO_NUMBER('333.46')
```

TO_SINGLE_BYTE

参阅：CONVERSION FUNCTIONS、第 11 章。

格式：

```
TO_SINGLE_BYTE(string)
```

描述：TO_SINGLE_BYTE 将字符串 string 中的多字节字符转换为等价的单字节字符。如果某个多字节字符没有等价的单字节字符，则此函数将返回未经转换的字符。

TO_TIMESTAMP

参阅：CONVERSION FUNCTIONS、TO_CHAR、TIMESTAMP、TO_TIMESTAMP_TZ。

格式：

```
TO_TIMESTAMP ( char [ , fmt ['nlsparam']] )
```

描述：如果字符(char)不是 TIMESTAMP 数据类型的默认格式，则 TO_TIMESTAMP 使用 fmt 格式将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符转换为 TIMESTAMP 数据类型的值。可选的 nlsparam 在此函数中的用途与用于日期转换的 TO_CHAR 函数的用途相同。

TO_TIMESTAMP_TZ

参阅：CONVERSION FUNCTIONS、TO_TIMESTAMP、TIMESTAMP WITH TIME ZONE。

格式：

```
TO_TIMESTAMP_TZ ( char [ , fmt ['nlsparam']] )
```

描述：TO_TIMESTAMP_TZ 将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符(char)转换为 TIMESTAMP WITH TIME ZONE 数据类型的值。

TO_YMINTERVAL

参阅：CONVERSION FUNCTIONS、INTERVAL YEAR TO MONTH。

格式：

```
TO_YMINTERVAL ( char )
```

描述：TO_YMINTERVAL 将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL YEAR TO MONTH 数据类型，其中 char 为要转换的字符串。

TRANSLATE

参阅：REPLACE、第 11 章。

格式：

```
TRANSLATE(string, from, to)
```

描述：TRANSLATE 查看 string 中的每个字符，然后检查 from，以确定该字符是否存在。如果存在，则 TRANSLATE 记录在 from 中的什么位置能找到这个字符，然后查看 to 中相同的位置。不管在该处找到任何字符，都用它替换 string 中的字符。

示例：

```
select TRANSLATE('I LOVE MY OLD THESAURUS','AEIOUY','123456')
from DUAL;

TRANSLATE('ILOVEMYOLDTH
-----------------------
3 L4V2 M6 4LD TH2S15R5S
```

TRANSLATE…USING

参阅：CONVERSION FUNCTIONAS、CONVERT、TO_NCHAR、UNISTR。

格式：

```
TRANSLATE ( text USING { CHAR_CS | NCHAR_CS } )
```

描述：TRANSIATE…USING 将 text 转换成指定的字符集，此字符集是为了在数据库字符集和国家语言字符集之间进行转换而指定的。text 参数是要转换的表达式。

指定 USING CHAR_CS 参数可以将 text 转换为数据库字符集。输出数据类型为 VARCHAR2。

指定 USING NCHAR_CS 参数可以将 text 转换为国家语言字符集。输出数据类型为 NVARCHAR2。

此函数类似于 Oracle 的 CONVERT 函数，但如果输入或输出数据类型为 NCHAR 或 NVARCHAR2，则必须使用此函数而不是 CONVERT。如果输入包含 UCS2 码点或反斜杠(\)，则应该使用 UNISTR 函数。

TREAT

参阅：DEREF、REF、第 41 章。

格式：

```
TREAT ( expr AS [REF] [schema.] type )
```

描述：TREAT 更改表达式的声明类型。

如果 expr 的声明类型为 source_type，则类型(type)必须是 source_type 的某个超类型或子类型。如果 expr 最特定的类型是 type(或 type 的某个子类型)，则 TREAT 返回 expr。如果 expr 最特定的类型不是 type(或 type 的某个子类型)，则 TREAT 返回 NULL。

如果 expr 的声明类型为 REF source_type，则 type 必须是 source_type 的某个子类型或超

类型。如果 DEREF(expr)最特定的类型为 type(或 type 的某个子类型)，则 TREAT 返回 expr。如果 DEREF(expr)最特定的类型不为 type(或 type 的某个子类型)，则 TREAT 返回 NULL。

TRIM

参阅：LTRIM、RTRIM、第 7 章。

格式：

```
TRIM
( [{ { LEADING | TRAILING | BOTH } [trim_character]}
    | trim_character
    }   FROM ] trim_source )
```

描述：TRIM 用来去掉一个字符串的前导或尾随字符(或两者都去掉)。如果 trim_character 或 trim_source 是字符字面量，则必须把它们括在单引号中。

如果指定 LEADING，则 Oracle 删除等于 trim_character 的前导字符。如果指定 TRAILING，Oracle 删除等于 trim_character 的尾随字符。如果指定 BOTH 或三者都不指定，则 Oracle 删除等于 trim_character 的头字符和尾随字符。

如果只指定 trim_source，则 Oracle 删除前导空格和尾随空格。如果 trim_source 或 trim_character 中任意一个为 NULL，则 TRIM 函数将返回一个 NULL 值。

示例：

```
select TRIM('"' from Title) from MAGAZINE;
select TRIM(leading '"' from Title) from MAGAZINE;
select TRIM(trailing '"' from Title) from MAGAZINE;
```

TRUNC (格式 1——针对日期)

参阅：COLUMN、DATE FUNCTIONS、TRUNC(格式 2)、第 10 章。

格式：

```
TRUNC(date,'format')
```

描述：TRUNC 是根据格式(format)截取日期(date)的函数。如果没有 format，则包括 11:59:59 P.M.(午夜前)的 date 将被截取为 12 A.M. (午夜)，即新的一天的开始。关于有效的日期格式值请参阅 ROUND(格式 1)。

TRUNC 的结果是始终带有时间的一个日期，且该时间为 12 A.M.，即当天的开始时刻。

TRUNC (格式 2——针对数值)

参阅：COLUMN、NUMBER FUNCTIONS、TRUNC(格式 1)、第 9 章。

格式：

```
TRUNC(value, precision)
```

描述：TRUNC 的结果为按精度截取的值。

示例：

```
TRUNC(123.45,0)  = 123
TRUNC(123.45,-1) = 120
TRUNC(123.45,-2) = 100
```

TRUNCATE CLUSTER

参阅：CREATE CLUSTER、DELETE、DROP TABLE、第 17 章。

格式：

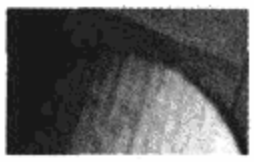
```
TRUNCATE CLUSTER [schema.] cluster
    [ {DROP | REUSE} STORAGE ] ;
```

描述：使用 TRUNCATE CLUSTER 语句可以删除群集中的所有行。默认情况下，Oracle Database 也执行如下任务：

- 释放已删除行使用的所有空间，MINEXTENTS 存储参数指定的行除外。
- 将 NEXT 存储参数设置为截取过程最后从段中删除的区的大小。

用 TRUNCATE 语句删除行比删除并重新创建群集的效率高。删除并重新创建群集使群集的依赖对象无效，这就需要在群集上重新授予对象权限，同时需要在表上重新创建索引和群集，并重新指定表的存储参数。而截取则不需要考虑这些事情。

用 TRUNCATE CLUSTER 语句删除行的速度比用 DELETE 语句删除所有行的速度快，尤其是如果群集有很多索引和其他依赖关系时更是如此。



注意：
不能回滚 TRUNCATE 语句。

要截取群集，群集必须在用户自己的模式中，或用户必须具有 DROP ANY TABLE 系统权限。

TRUNCATE TABLE

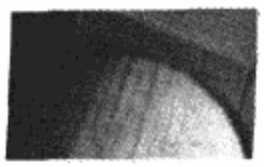
参阅：CREATE TABLE、DROP TABLE 和第 17 章。

格式：

```
TRUNCATE TABLE [schema.] table
  [ {PRESERVE | PURGE} MATERIALIZED VIEW LOG ]
  [ {DROP | REUSE} STORAGE ] ;
```

描述：TRUNCATE 从表中删除所有行。如果添加 DROP STORAGE 选项，则 TRUNCATE 将释放已删除行的空间。如果添加 REUSE STORAGE 选项，则 TRUNCATE 将在表中为新行预留空间。DROP STORAGE 为默认选项。

TRUNCATE 命令比 DELETE 命令的执行速度快，因为它不生成回滚信息、不激活任何 DELETE 触发器(因此必须小心使用)，并且不在物化视图日志中记录任何信息。另外，使用 TRUNCATE 不会使依赖于表中已删除的行或权限的对象无效。



注意：
不能回滚 TRUNCATE 语句。

TTITLE(SQL*Plus)

参阅： ACCEPT、BTITLE、DEFINE、PARAMETERS、REPFOOTER、REPHEADER。

格式：

```
TTI[TLE] [option [text|variable]... | OFF | ON]
```

描述： TTITLE(Top TITLE，顶部标题)在每页报表的顶部放置标题(可能有多行)。OFF 和 ON 禁止和还原文本的显示而不改变其内容。只给出 TTITLE 将显示当前的 TTITLE 选项和 text 或 variable。

text 是给予此报表的标题，variable 是用户定义的变量或系统维护的变量，其中包括当前行号 SQL.LNO、当前页码 SQL.PNO、当前 Oracle 版本号 SQL.RELEASE、当前错误代码 SQL.SQLCODE，以及用户名 SQL.USER。

如果 TTITLE 后面的第 1 个单词是有效的选项，则 SQL*Plus 按新格式使用 TTITLE。有效的选项为：

- COL n 直接从当前行的左边界跳到位置 n。
- S[KIP] n 输出 n 个空行。如果不指定 n，则输出一个空行。如果 n 为 0，则不输出空行，且输出的当前位置为当前行的位置 1(该页的最左边)。
- TAB n 向前跳 n 个位置(如果 n 为负数则向后跳)。
- BOLD 以粗体显示输出数据。
- LE[FT]、CE[NTER]和 R[IGHT]将当前行的数据左对齐、居中和右对齐。跟随在这些命令后面的任何文本或变量都作为同一个组的内容对齐，直到该命令结束，或者遇到LEFT、CENTER、RIGHT 或COL 命令为止。CENTER 和RIGHT 使用 SET LINESIZE 命令设置的值来确定将文本或变量放置在什么位置。
- FORMAT string 指定了格式模型(FORMAT MODEL)，该模型将控制后面的文本或变量的格式，并遵循与 COLUMN 命令中 FORMAT 同样的语法，如 FORMAT A12 或 FORMAT $999,990.99。每当 FORMAT 出现时，它都会替代前一个格式。如果没有指定 FORMAT 模型，则使用 SET NUMFORMAT 设置的格式。如果没有设置 NUMFORMAT，则使用 SQL*Plus 的默认值。

除非某个变量已经被加载了一个由 TO_CHAR 重新格式化了的日期，否则日期值将根据默认格式输出。

在单个的 TTITLE 中可以使用任何数量的选项、文本块和变量。它们都按照指定的顺序输出，并且都按照在它之前的子句所指定的格式进行定位和格式化。

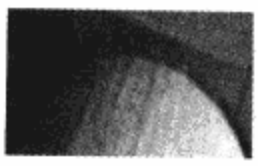
TYPE(嵌入式 SQL)

参阅： *Programmer's Guide to Oracle Precompiler*。

格式：

```
EXEC SQL TYPE type IS datatype
```

描述：PL/SQL 允许将 Oracle 外部数据类型分配给用户定义的数据类型。此数据类型可以包含长度、精度或小数位数。外部数据类型等价于用户定义的类型，并分配给所有宿主变量(这些宿主变量被分配给用户定义的类型)。关于外部数据类型的清单请参阅 *Programmer's Guide to Oracle Precompiler*。



注意：
所有命令的详细清单请参阅 *Programmer's Guide to Oracle Precompiler*。本附录不包含所有可能的预编译命令。

TZ_OFFSET

参阅：SESSIONTIMEZONE、DBTIMEZONE。

格式：

```
TZ_OFFSET
( { 'time_zone_name'
  | '{ + | - } hh : mi'
  | SESSIONTIMEZONE
  | DBTMEZONE
  })
```

描述：TZ_OFFSET 返回对应于基于执行语句的日期输入的值的时区偏移量。可以输入有效的时区名、UTC 的时区偏移量(它返回本身)或关键字 SESSIONTIMEZONE 或 DBTIMEZONE。关于有效值的清单，可查询 V$TIMEZONE_NAMES 动态性能视图的 TZNAME 列。

UID

格式

```
UID
```

描述：UID 返回一个整数，此整数唯一标识会话用户(登录的用户)。

UNDEFINE(SQL*Plus)

参阅：ACCEPT、DEFINE、PARAMETERS。

格式：

```
UNDEF[INE] variable
```

描述：UNDEFINE 删除用户变量的定义，该用户变量是由 ACCEPT、DEFINE 或作为 START 命令的参数定义的。可以在一个单独的命令中 UNDEFINE 多个变量的当前值。以下

程序清单示了相应的格式：

```
undefine variable1 variable2 ...
```

示例：为了 UNDEFINE 变量 Total，可使用以下命令：

```
undefine Total
```

UNION

参阅：INTERSECT、MINUS、QUERY OPERATORS、第 13 章。

格式：

```
select...
 UNION [ALL]
select...
```

描述：UNION 将两个查询组合起来。它返回两个 SELECT 语句中所有不同的行，或者指定 ALL 时返回所有的行而不管这些行是否重复。SELECT 语句之间的列的数量和数据类型必须一致，尽管列名不需要相同。关于 INTERSECT、UNION 和 MINUS 的重要的区别和作用以及在结果中优先顺序所起作用的讨论，请参阅第 13 章。

UNISTR

参阅：CONVERSION FUNCTIONS、CONVERT、TRANSLATE … USING。

格式：

```
UNISTR ( 'string' )
```

描述：UNISTR 以任意字符集的一个字符串作为它的参数，并以数据库 Unicode 字符集中的 Uniclde 字符返回它。为了在字符串中包括 UCS2 码点字符，可使用转义反斜杠(\\)后跟下一个数字来完成。为了包含反斜杠自身，可在其前面放置另一个反斜杠(\\\\)。

这个函数除提供 UCS2 码点和反斜杠字符的转义字符外，其功能类似于 TRANSLATE … USING 函数。

UPDATE (格式 1——嵌入式 SQL)

参阅：EXECUTE IMMEDIATE、FOR、PREPARE、SELECT(格式 2)、*Programmer's Guide to Oracle Precompiler*。

格式：

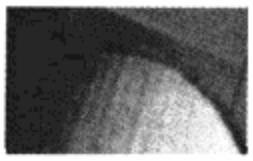
```
EXEC SQL [AT { dbname | :host_variable }]
    [FOR { :host_integer | integer }]
    UPDATE
    [ (subquery)
    | [schema.] { table | view } [ @db_link | PARTITION (part_name) ] ]
    SET
    { column = { expr | (subquery_2) }
    | (column [, column]...) = (subquery_1) }
```

```
[, { column = { expr | (subquery_2) }
   | (column [, column]...) = (subquery_1) } ]...
[WHERE { condition | CURRENT OF cursor }]
[{ RETURN | RETURNING } expr [, expr]... INTO
:host_variable [[INDICATOR] :ind_variable]
[, :host_variable [[INDICATOR] :ind_variable]]...]
```

描述：请参阅 UPDATE(格式 3)中的各种子句的描述。对嵌入式 SQL 来说唯一的元素如下：

- AT dbname 可选择性地指定来自前面的 CONNECT 语句的数据库，数据库名来自前面的 DECLARE DATABASE 语句。
- FOR:host_interger 设置可以取出的行数的最大值。integer 是命名的宿主变量。
- expr 可包含一个宿主:variable [:indicator]。
- WHERE 子句可以包含宿主变量或数组。
- CURRENT OF 更新指定的 cursor 取出的最后一行。但是，该游标必须打开并在该行中定位。如果它没有打开，则 CURRENT OF 子句(是 WHERE 子句的一部分)将导致 WHERE 找不到任何行，因此也不能更新任何内容。游标必须在 DECLARE CURSOR 语句中用 SELECT…FOR UPDATE OF 预先命名。

如果 SET 或 WHERE 中的任何宿主变量是数组，则其中的所有宿主变量都必须为数组，尽管它们不必是大小相同的数组。如果它们为数组，则 UPDATE 对数组中的每组组件执行一次，并且可以更新零行或多行。最大的数量依赖于最小数组的大小或 FOR 子句中的整数值(如果指定的话)。其他详细信息请参阅 CURSOR FOR LOOP。



注意：
所有命令的完整列表请参阅 *Programmer's Guide to Oracle Precompiler*。本附录不包含所有可能的预编译命令。

UPDATE (格式 2——PL/SQL)

参阅：DECLARE CURSOR、SUBQUERY。

格式：

```
UPDATE [user.]table[@dblink]
SET { column = expression | column = (select expression...)
  [ [,column = expression]... |
  [,column = (select
expression...)]... ] |
  (column [,column]...) = (subquery)}
  [WHERE {condition | CURRENT OF cursor}];
```

描述：PL/SQL 中的 UPDATE 与普通的 SQL UPDATE 语句的使用方式相同(除了可选择 WHERE 子句、WHERE CURRENT OF 游标外)。在本实例中，UPDATE 语句只影响游标中最后一个 FETCH 的结果，即当前一行。游标的 SELECT 语句必须包含关键字 FOR UPDATE。

与 INSERT、DELETE 和 SELECT…INTO 一样，UPDATE 语句总是使用名为 SQL 的隐式游标。此游标从来不需要声明(但为了使用 WHERE CURRENT OF 游标，必须在游标中声

明 SELECT…FOR UPDATE 语句)。其属性设置如下：

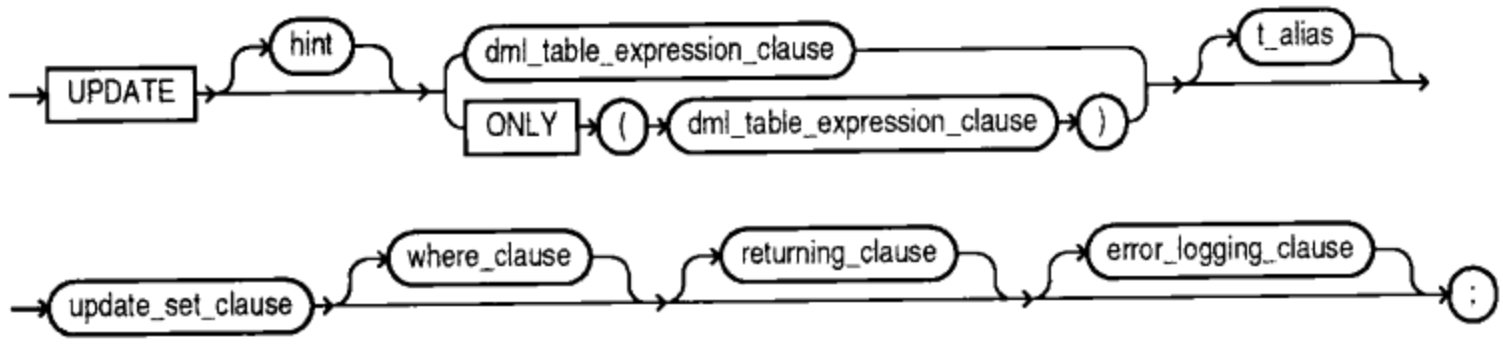
- SQL%ISOPEN 是不相关的。
- 如果更新了一行或多行，则 SQL%FOUND 为 TRUE；如果未更新任何行，则它为 FALSE。SQL%FOUND 的相反的属性是 SQL%NOTFOUND。
- SQL%ROWCOUNT 为已更新的行数。

UPDATE (格式 3——SQL 命令)

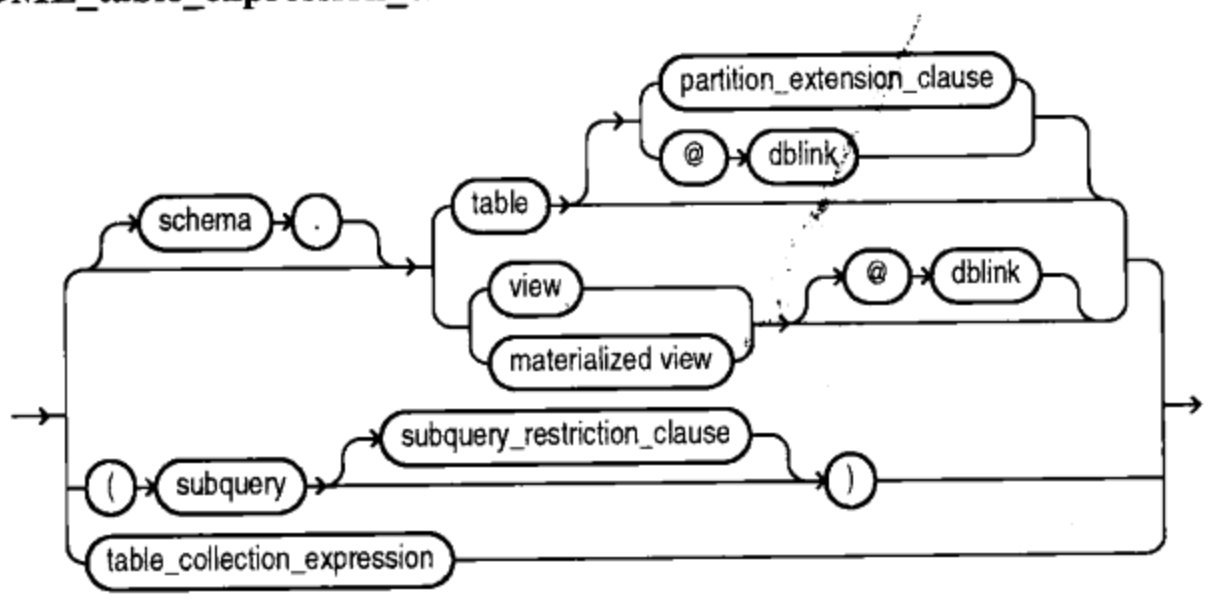
参阅： DELETE、INSERT、SELECT、SUBQUERY、WHERE、第 15 章。

格式：

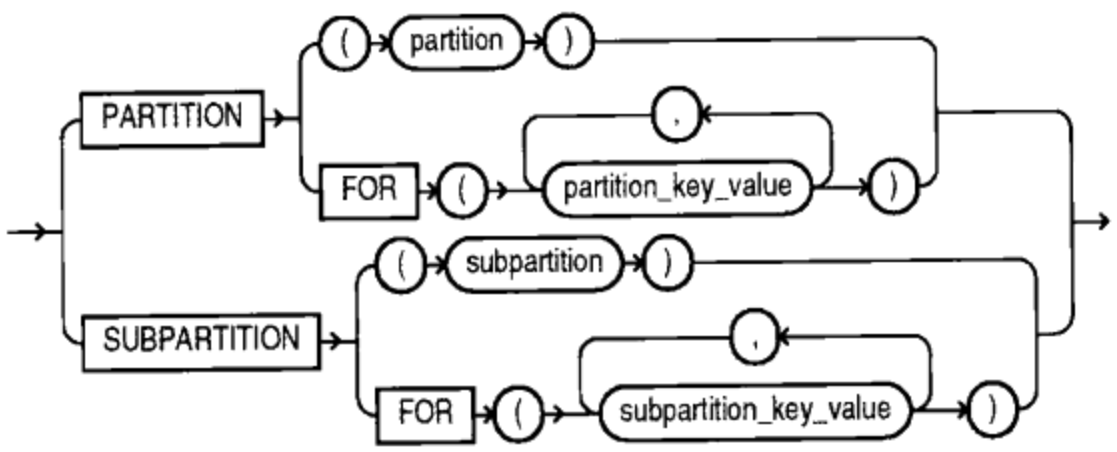
update::=



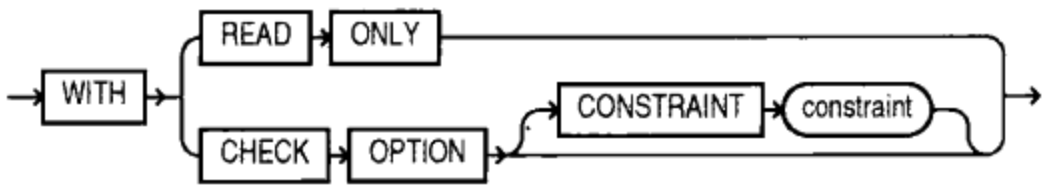
DML_table_expression_clause::=



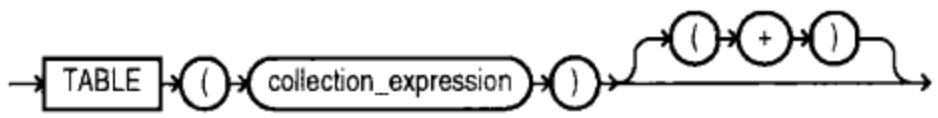
partition_extension_clause::=



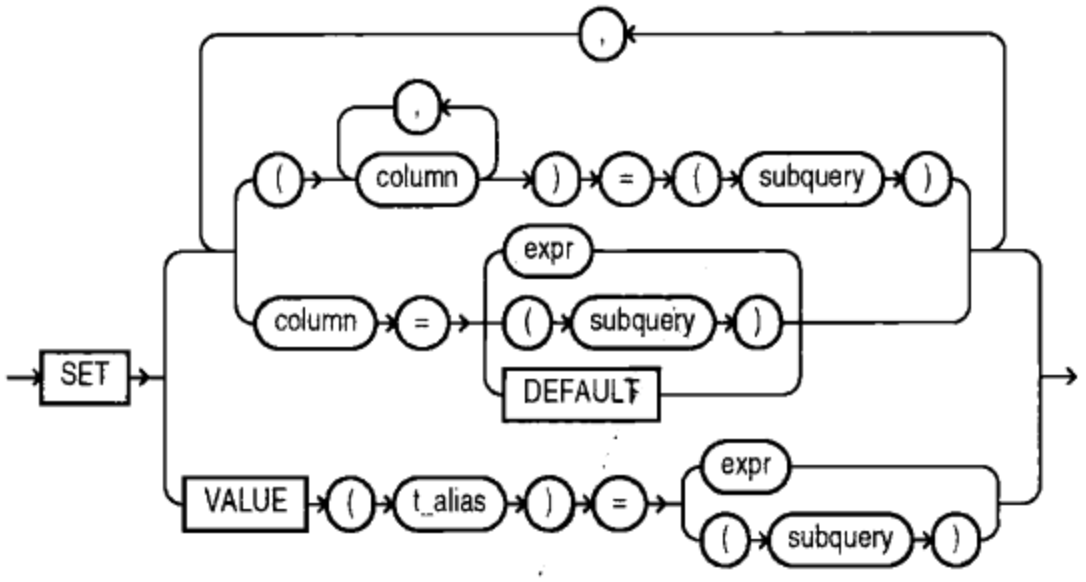
subquery_restriction_clause::=



table_collection_expression::=

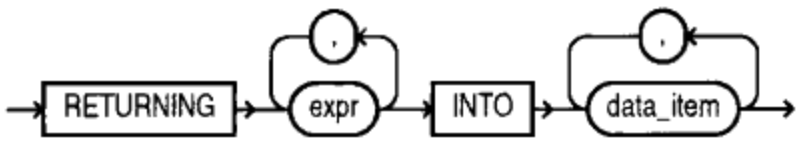


update_set_clause::=

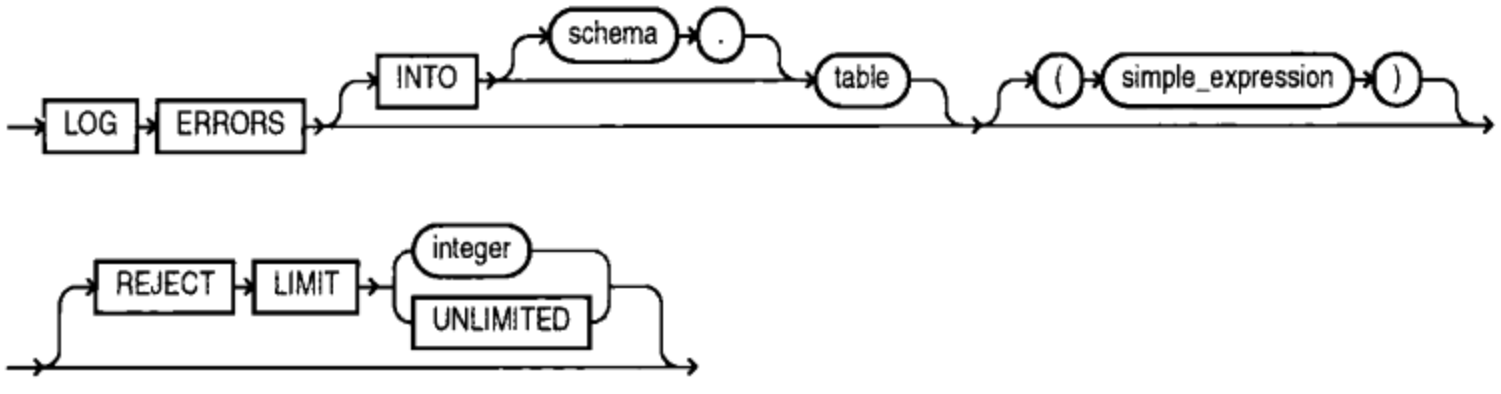


where_clause::=

returning_clause::=



error_logging_clause::=



描述：UPDATE 更改在指定的表中所列出的列值。WHERE 子句可以包含相关的子查询。子查询可以从正在更新的表中选择，尽管它必须只返回一行。如果没有 WHERE 子句，则更新所有行。如果使用 WHERE 子句，则只更新它选中的那些行。表达式在该命令执行时进行

判定，并且其结果将代替数据行中的当前列值。

子查询必须选择与 SET 子句左边圆括号中同样数量的列(以及相兼容的数据类型)。等于表达式的列优先于在等于子查询的圆括号中设置的列，它们都在一个单独的 UPDATE 语句中。

示例：为了将所有 Publisher 值设置为 NULL，可使用以下代码：

```
update BOOKSHELF set Publisher = NULL ;
```

以下程序将更新 COMFORT 表中的城市 Walpole，将 Walpole 的所有行的降雨量都设置为 NULL，中午的温度等于 Manchester 的温度，午夜的温度等于中午的温度减去 10℃。

```
update COMFORT set Precipitation = NULL,
                   (Noon, Midnight) =
          (select Temperature, Temperature - 10
            from WEATHER
           where City = 'MANCHESTER')
    where City = 'WALPOLE';
```

UPDATEXML

参阅：第 52 章。

格式：

```
UPDATEXML (XMLType_instance,
    [Xpath_string , value_expr [,Xpath_string , value_expr ...]
    [, namespace_string ] )
```

描述：UPDATEXML 把 XMLType 实例和 Xpath 值作为一对参数并返回包含更新值的 XMLType 实例。如果 Xpath_string 是 XML 元素，则相应的 value_expr 必须是 XMLType 实例。如果 Xpath_string 是属性或文本节点，则相应的 value_expr 可以是任意标量数据类型。每个 Xpath_string 的目标数据类型和它相应的 value_expr 必须匹配。可选的 namespace_string 必须解析成 VARCHAR2 值，该值为前缀指定默认映射或名称空间映射(Oracle 在判定 XPath 表达式值时使用)。

UPPER

参阅：LOWER、第 7 章。

格式：

```
UPPER(string)
```

描述：UPPER 将 string 中的每个字母转换成大写。

示例：

```
upper('Look what you''ve done!')
```

将产生以下结果：

```
LOOK WHAT YOU'VE DONE!
```

UROWID

UROWID(Universal RowID)是索引编排表的主键的数据类型。最大值和默认大小都是4 000字节。详细内容请参阅 DATATYPES 和 CREATE TABLE。

USER

参阅： UID。

格式：

```
USER
```

描述： Oracle 通过 USER 这一名称可以知道当前用户。由于 USER 是一个函数，因此可在任何 SELECT 语句中查询它。

示例：

```
select USER from DUAL;
```

USERENV

参阅： CHARACTER FUNCTIONS。

格式：

```
USERENV(option)
```

描述： USERENV 是一个遗留函数，它被 SYS_CONTEXT 的 UserEnv 名称空间代替。其选项及其返回值如表 A-31 所示。

表 A-31　USERENV 函数的选项和返回值

'CLIENT_INFO'	CLIENT_INFO 返回多达 64 字节的用户会话信息。可以通过使用 DBMS_APPLICATION_INFO 程序包的应用程序来存储该信息
'ENTRYID'	ENTRYID 返回可用的审计项标识符。在分布式 SQL 语句中不能使用该属性。为了在 USERENV 中使用该关键字，必须设置初始化参数 AUDIT_ TRAIL 为真
'ISDBA'	若用户通过操作系统或口令文件拥有 DBA 的权限，并用 DBA 进行了身份验证，则 ISDBA 返回 TRUE
'LANG'	LANG 返回语言名称的 ISO 缩写，即已有‘LANGUAGE’参数的简写形式
'LANGUAGE'	LANGUAGE 返回当前由会话使用的语言和地域，以及以下格式的数据库字符集：language _territory.characterset
'SESSIONID'	SESSIONID 返回审计会话标识符。在分布式 SQL 语句中不能使用该属性
'TERMINAL'	返回当前会话终端的操作系统标识符。在分布式 SQL 语句中，该属性返回本地会话的标识符。在分布式环境中，该选项只支持远程 SELECT 语句，而不支持远程 INSERT、UPDATE 或 DELETE 操作

VALUE

参阅： DEREF、REF、第 41 章。

格式：

```
VALUE ( correlation_variable )
```

描述： 在 SQL 语句中，VALUE 以一个与对象表的某行相关的关联变量(表别名)作为其参数，并返回在该对象表中存储的对象实例。对象实例的类型与对象表的类型相同。详细内容请参阅第 41 章。

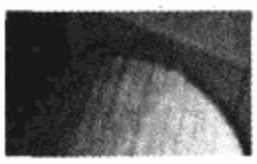
VAR(嵌入式 SQL)

参阅： SELECT(格式 2)、VARIABLE DECLARATION、*Programmer's Guide to Oracle Precompilers*。

格式：

```
EXEC SQL VAR host_variable
    { IS dtyp [({ length | precision, scale })]
      [CONVBUFSZ [IS] (size)]
    | CONVBUFSZ [IS] (size) }
```

描述： PL/SQL 可以通过 VAR 命令重写一个变量的默认数据类型的分配。一旦声明了变量，此变量就使用在 DECLARE 命令中为它分配的数据类型。VAR 用来在 PL/SQL 块中改变已声明变量的数据类型。



注意：
所有命令的完整列表请参阅 *Programmer's Guide to Oracle Precompiler*。本附录不包含所有可能的预编译命令。

VAR_POP

参阅： AGGREGATE FUNCTIONS、VAR_SAMP、VARIANCE。

格式：

```
VAR_POP ( expr ) [OVER ( analytic_clause )]
```

描述： VAR_POP 返回一组数的总体方差(在丢弃了该组数中的 NULL 值后)。

VAR_SAMP

参阅： AGGREGATE FUNCTIONS、VAR_POP、VARIANCE。

格式：

```
VAR_SAMP ( expr ) [OVER ( analytic_clause )]
```

描述： VAR_SAMP 返回一组数的样本方差(在丢弃了该组数中的 NULL 值后)。

VARCHAR2

参阅：DATATYPES。

VARIABLE

参阅：PRINT

格式：

```
VAR[IABLE] [variable_name [NUMBER|CHAR|CHAR (n{CHAR|BYTE])|NCHAR|NCHAR(n)
|VARCHAR2 (n[CHAR|BYTE]) | NVARCHAR2(n) | CLOB | NCLOB | REFCURSOR
|BINARY_FLOAT | BINARY_DOUBLE
```

描述：VARIABLE 声明一个可在 PL/SQL 中引用的绑定变量。每个变量都被分配一个 variable_name 和一个类型(NUMBER 或 CHAR)。对于 CHAR 变量可指定最大长度(n)。后面的变量名 VARIABLE 列出了此变量。

示例：在以下示例中，创建变量 bal，并使它等于一个函数的结果：

```
variable bal NUMBER
begin
  :bal := LATE_FEE('DORAH TALBOT');
end;
```

变量声明(PL/SQL)

格式：

```
variable [CONSTANT]
  {type | identifier%TYPE | [user.]table%ROWTYPE}
    [NOT NULL]
    [{DEFAULT | :=} expression];
```

描述：PL/SQL 允许在 PL/SQL 块中声明变量。如果声明的变量为 CONSTANT，则必须在该声明中用一个值初始化该变量，而且不能给该变量重新赋值。

变量的类型可以是 PL/SQL 类型(请参阅 DATATYPES)，另一个 PL/SQL 变量的类型或标识符给定的数据库列的类型，或者是引用一个与数据库表的记录相对应的 ROWTYPE(请参阅 %ROWTYPE)。

如果在声明中添加 NOT NULL，则不能再分配一个 NULL 值给该变量，而且必须初始化该变量。初始化(跟随在 DEFAULT 或赋值符号“:=”后面)表达式是产生被声明的类型值的任何有效的 PL/SQL 表达式。

VARIANCE

参阅：ACCEPT、AGGREGATE FUNCTIONS、COMPUTE、DEFINE、PARAMETERS、STDDEV、VAR_POP、 VAR_SAMP 和第 9 章。

格式：

```
VARIANCE ( [ DISTINCT | ALL ] expr ) [OVER ( analytic_clause )]
```

描述：VARIANCE 给出一组行的所有值(value)的方差。与其他组函数一样，VARIANCE 忽略 NULL 值。

VARRAY

VARRAY 是 CREATE TYPE 命令内部的子句，它说明数据库对可变数组内项数的限制。关于语法信息请参阅 CREATE TYPE。

VSIZE

参阅：CHARACTER FUNCTIONS、LENGTH、NUMBER FUNCTIONS、第 9 章。

格式：

```
VSIZE(value)
```

描述：VSIZE 是 Oracle 中值(value)的存储大小。对于数字，它通常小于显示的长度，因为在数据库中存储数字需要较少的空间。

WHENEVER

参阅：EXECUTE、FETCH。

格式：

```
EXEC SQL WHENEVER {NOT FOUND | SQLERROR | SQL WARNING}
    {CONTINUE |
    GOTO label |
    STOP |
    DO routine |
    DO BREAK |
    DO CONTINUE}
```

描述：WHENEVER 不是可执行的 SQL 语句，而是指示 Oracle 语言处理器在每个 SQL 语句之后嵌入“IF condition THEN GOTO label”语句的指令。当每个 SQL 语句执行时，测试其结果是否满足 condition。如果满足，程序就跳转转移到 label 处。

对于 3 种可能的条件，WHENEVER 可以与 CONTINUE 或者与一个不同的标签一起使用。每个条件对于所有随后的 SQL 语句都有效。具有这些条件之一的新的 WHENEVER 将完全代替其前面的 WHENEVER 对所有随后的 SQL 语句起作用。

NOT FOUND 条件在 SQLCODE 为 100 时引发，例如，这表示 FETCH 未能返回任何行(包括 INSERT 语句中的子查询)。

SQLERROR 在 SQLCODE 小于 0 时发生。这些是典型的致命性错误，并且需要处理重大的错误。

SQLWARNING 在出现非致命性“错误”时发生。这些错误包括加载到宿主变量的字符串被截断、与宿主变量的 INTO 列表不匹配的选择的一个列列表，以及没有 WHERE 子句的 DELETE 或 UPDATE 语句。

通常，最初的分支、继续、停止或例程执行都在每个程序的开始处进行设置，即在任何可执行的 SQL 语句的前面。但是，在程序体中的特定逻辑块中，任何或所有语句都可能根据

本地的需要而被代替。请注意，WHENEVER 不知道任何有关宿主语言的作用域规则、调用或分支的内容。从声明 WHENEVER 的那一刻起，直到另一个 WHENEVER(使用相同的条件)代替它为止，其规则都有效。另一方面，程序必须遵循 GOTO 和标签的语言作用域规则。

STOP 选项停止程序的执行，DO 选项调用某种宿主语言例程，该例程的语法依赖于宿主语言。DO BREAK 从循环中中断，DO CONTINUE 在条件满足时执行循环中的一条继续语句。

最后，在一个错误处理例程中，尤其是在包含 SQL 语句的例程中，WHENEVER SQLERROR 很可能应该包含 CONTINUE 或 STOP，或者它应该调用退出例程，以避免无限循环回到相同的错误处理例程中。

WHENEVER OSERROR(SQL*Plus)

参阅： WHENEVER SQLERROR。

格式：

```
WHENEVER OSERROR
{EXIT [SUCCESS|FAILURE|n|variable|:BindVariable] [COMMIT|ROLLBACK]
|CONTINUE [COMMIT|ROLLBACK|NONE]}
```

描述： WHENEVER OSERROR 处理 SQL*Plus 中的处理逻辑(如果发生操作系统错误的话)。EXIT 命令 SQL*Plus 退出，CONTINUE 关闭 EXIT。COMMIT 告诉 SQL*Plus 在退出前发布 COMMIT，ROLLBACK 告诉 SQL*Plus 在退出前发布 ROLLBACK。默认行为是 NONE。

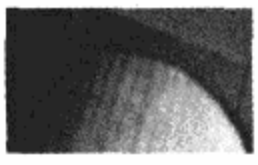
WHENEVER SQLERROR(SQL*Plus)

参阅： WHENEVER OSERROR。

格式：

```
WHENEVER SQLERROR
{EXIT [SUCCESS|FAILURE|WARNING|n|variable|:BindVariable]
[COMMIT|ROLLBACK]|CONTINUE [COMMIT|ROLLBACK|NONE]}
```

描述： 如果 SQL 命令或 PL/SQL 块遇到错误，则 WHENEVER SQLERROR 处理 SQL*Plus 中的处理逻辑。EXIT 命令 SQL*Plus 退出，CONTINUE 关闭 EXIT。COMMIT 告诉 SQL*Plus 在退出前发布 COMMIT 命令，ROLLBACK 告诉 SQL*Plus 在退出前发布 ROLLBACK 命令。默认行为是 NONE。



注意：
SQL 和 PL/SQL 错误启动 WHENEVER SQLERROR 处理。SQL*Plus 错误则不启动。

可以将 WHENEVER SQLERROR 根据需要放在启动文件中的若干个 SQL 语句之前。每个 WHENEVER SQLERROR 都将替代前一个语句，并保留其作用直到它被取代为止。

示例： 当下面示例中的 CREATE TABLE 命令由于字面量 KEENE 没有被括在单引号中而失败时，后面的 UPDATE 和 SELECT 将不能执行。WHENEVER SQLERROR 将立即退出 SQL*Plus，并将 SQL.SQLCODE 传递给主机：

```
WHENEVER SQLERROR EXIT SQL.SQLCODE;

create table KEENE as
select * from COMFORT
  where City = KEENE;

update KEENE set Noon = 75;

select * from KEENE;
```

WHERE

参阅： DELETE、LOGICAL OPERATORS、PRECEDENCE、SELECT、SYNTAX OPERATORS、UPDATE。

格式：

```
DELETE FROM [user.]table ...
  [ WHERE condition ]
SELECT ...
  [ WHERE condition ]
UPDATE [user.]table [alias] ...
  [WHERE condition ]
```

描述： WHERE 定义了控制 SELECT 语句将返回哪些行、DELETE 语句将删除哪些行或者 UPDATE 语句将更新哪些行的逻辑条件。

条件可以用以下一种或几种方法定义：

- 表达式之间的比较(=、\>、!=，等等)
- 一个表达式和一个查询之间的比较
- 一列表达式和查询中的一列表达式之间的比较
- 一个表达式和一个列表的任意一个(ANY)或所有(ALL)成员之间的比较，或者是一个表达式和从查询中得到的值之间的比较
- 测试一个表达式在(IN)或不在(NOT IN)一个列表中，或者在(IN)或不在(NOT IN)查询的结果中
- 测试是否在(BETWEEN)或不在(NOT BETWEEN)一个值和另一个值之间
- 测试表达式是否为空(IN NULL)或不为空(IN NOT NULL)
- 测试一个查询是否存在(EXIST)或不存在(NOT EXIST)结果
- 以上任何条件使用 AND 和 OR 的组合

示例：

```
where Section =ANY ('A','C','D');
where City !=ALL (select City from LOCATION);
where Section IN ('A','C','D');
where City NOT IN (select City from LOCATION);
where Page BETWEEN 4 and 7;
where Skill IS NULL;
where EXISTS (select * from WORKER where Age \> 90);
where Section = 'A' or Section = 'B' and Page = 1;
```

WHILE (PL/SQL)

参阅： LOOP、第 34 章。

WIDTH_BUCKET

参阅： NTILE。

格式：

```
WIDTH_BUCKET ( expr , min_value , max_value , num_buckets )
```

描述： WIDTH_BUCKET 构造等宽的直方图，即将直方图的区间划分为等宽的间隔(可以将该函数与 NTILE 比较，NTILE 创建等高直方图)。在理想情况下，每个存储桶都是实数线的“闭-开”间隔。对于给定的表达式，WIDTH_BUCKET 把存储桶号返回给经过判断后表达式的值落在的存储桶中。expr 是创建该直方图的表达式(一个数值或日期时间值)。如果 expr 为 NULL，则该表达式返回 NULL。min_value 和 max_value 是两个表达式，它们解析为 expr 可接受的范围区间的两个端点。这两个表达式也必须是数值或日期时间值，而不能是 NULL。num_buckets 是指存储桶数量的表达式，它是一个常数。该表达式必须为正整数。

XMLAGG

参阅： SYS_XMLAGG。

格式：

```
XMLAGG ( XMLType_instance [ order_by_clause] )
```

描述： XMLAGG 是一个聚集函数。它接受 XML 段的集合并返回聚集的 XML 文档。任何返回 NULL 的参数都从结果中丢弃。

除了 XMLAGG 返回节点的集合，XMLAGG 类似于 SYS_XMLAGG。XMLAGG 不接受使用 XMLFORMAT 对象的格式。同样，XMLAGG 不像 SYS_XMLAGG 那样将输出结果括在一个元素标签中。

XMLCAST

参阅： 第 52 章。

格式：

```
XMLCAST ( value_expression AS datatype )
```

描述： XMLCAST 将 value_expression 转换为由 datatype 指定的标量 SQL 数据类型。value_expression 参数是一个 SQL 表达式。datatype 参数可以是 NUMBER、VARCHAR2 或任何日期时间数据类型。

XMLCDATA

参阅： 第 52 章。

格式：

```
XMLCDATA ( value_expr )
```

描述：XMLCData 通过计算 value_expr 的值而生成 CDATA 部分。value_expr 必须解析为串。

XMLCOLATTVAL

参阅：第 52 章。

格式：

```
XMLCOLATTVAL ( value_expr [AS c_alias] [, value_expr [AS c_alias]] )
```

描述：XMLCOLATTVAL 创建一个 XML 段并扩展结果 XML，使得每个 XML 段都拥有包含 name 属性的名字列。可以使用 AS c_alias 子句更改 name 属性的值而不更改列名。

必须指定 value_expr 的值。如果 value_expr 的值为 NULL，则没有元素返回。

XMLCONCAT

参阅：XMLSEQUENCE

格式：

```
XMLCONCAT ( XMLType_instance [, XMLType_instance ] )
```

描述：XMLCONCAT 以一系列 XMLType 实例作为输入，连接每行的一系列元素，并返回已连接的系列。XMLCONCAT 与 XMLSEQUENCE 相反。

XMLDIFF

参阅：第 52 章。

格式：

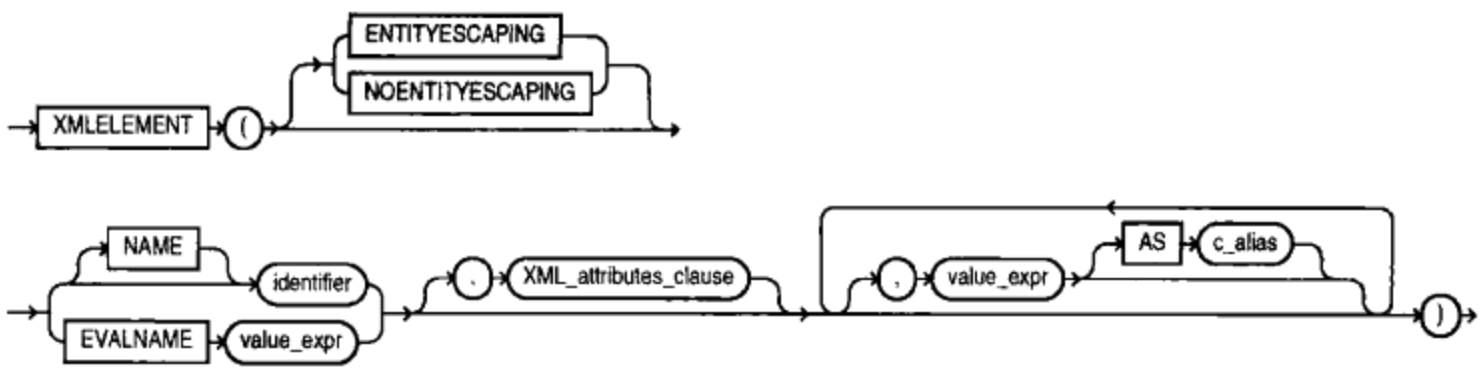
```
XMLDiff ( XMLType_document, XMLType_document [ , integer, string ] )
```

描述：XMLDiff 函数是 XmlDiff C API 的 SQL 接口。此函数比较两个 XML 文档，并在符合 Xdiff 模式的 XML 中捕获二者的差异。diff 文档作为 XMLType 文档被返回。

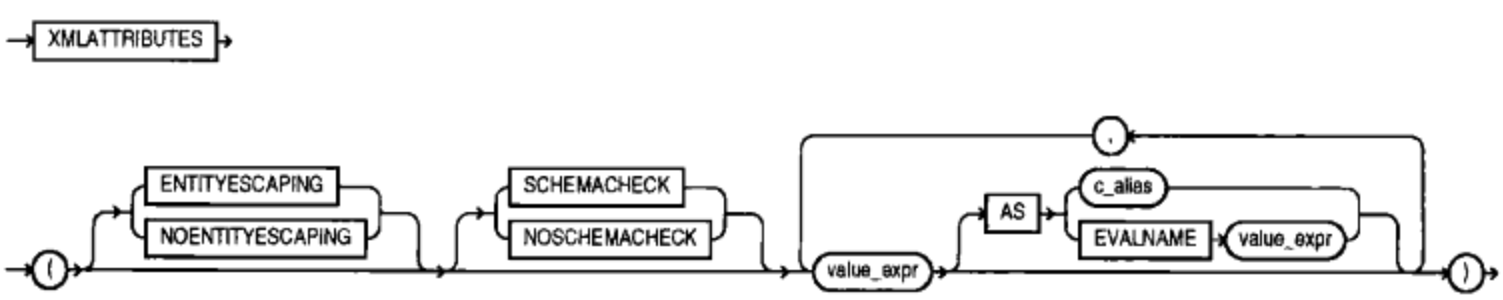
XMLELEMENT

参阅：SYS_XMLGEN。

格式：



XML_attributes_clause::=

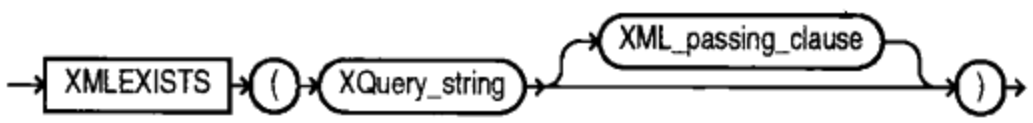


描述：XMLELEMENT 的参数有标识符(identifier)的元素名、此元素的属性集(可选)以及组成元素内容的参数。它返回 XML 类型的实例。XMLELEMENT 类似于 SYS_XMLGEN，除了 XMLELEMENT 可以包含返回的 XML 中的属性。XMLELEMENT 不接受使用 XMLFORMAT 对象的格式。

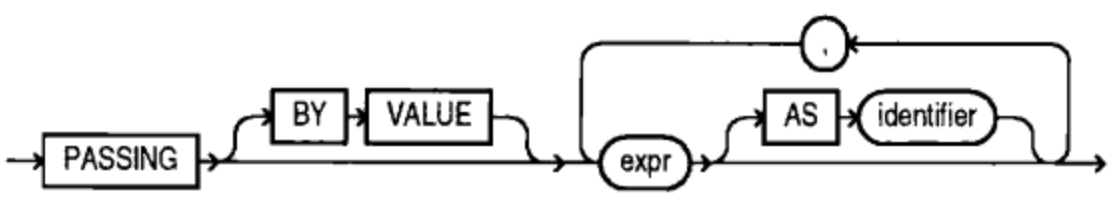
XMLEXISTS

参阅：第 52 章。

格式：



XML_passing_clause::=



描述：XMLExists 检查给定的 XQuery 表达式是否返回非空的 XQuery 序列。如果是，则此函数返回 TRUE；否则返回 FALSE。虽然参数 XQuery_string 是字面量字符串，但它能包含使用 XML_passing_clause 绑定的 XQuery 变量。

XML_passing_clause 中 expr 的是一个返回 XMLType 类型的表达式；或者是一个 SQL 标量数据类型的实例，它用作判断 XQery 表达式的上下文。可以在 PASSING 子句中只指定一个 expr，这不使用标识符。每个 expr 的判断结果都应该与 XQuery_string 中的标识符相对应。如果任何 expr 的后面未跟 AS 子句，则此表达式的判断结果用作判断 XQuery_string 的上下

文条目。

XMLFOREST

参阅： XMLCONCAT。

格式：

```
XMLFOREST ( value_expr [AS c_alias] [, value_expr [AS c_alias]] )
```

描述： XMLFOREST 首先将它的每个变量参数转换成 XML，然后返回与被转换的变量连接的 XML 段。

XMLPARSE

参阅： 第 52 章。

格式：

```
XMLPARSE
  ({ DOCUMENT | CONTENT } value_expr [ WELLFORMED ] )
```

描述： XMLParse 函数从 value_expr 的判断结果解析并生成 XML 实例。value_expr 必须解析为串。如果 value_expr 解析为空，则此函数返回空值。

XMLPATCH

参阅： 第 52 章。

格式：

```
XMLPatch ( XMLType_document, XMLType_document )
```

描述： XMLPatch 函数是 XmlPatch C API 的 SQL 接口。此函数用指定的变更来修补 XML 文档，并返回修补过的 XMLType 文档。

XMLPI

参阅： 第 52 章。

格式：

```
XMLPI
  ( { [ NAME ] identifier
    | EVALNAME value_expr } [, value_expr ] )
```

描述： XMLPI 函数通过 identifier、或者通过 identifier 和 value_expr 的结果生成 XML 处理指导。处理指导常常用来向应用程序提供与部分或全部 XML 文档相关的任何信息。应用程序使用处理指导来确定 XML 文档处理得有多好。

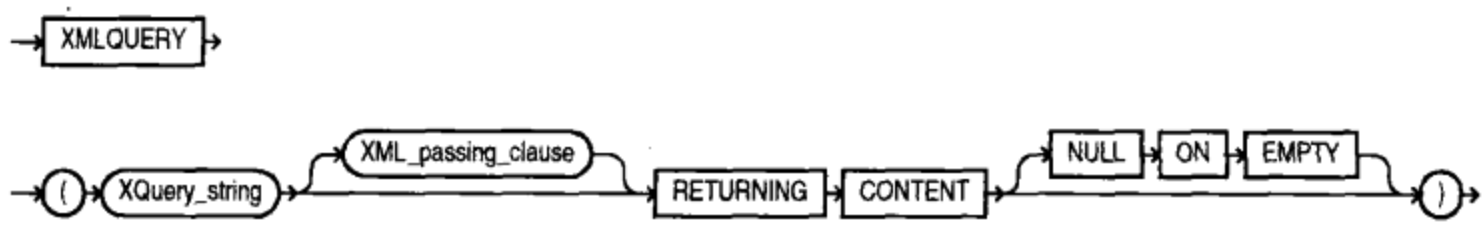
必须为 Oracle Database 指定一个值才能使用封闭标记。这可以通过指定串字面量 identifier 或通过指定 EVALNAME value_expr 来完成。如果通过指定 EVALNAME value_expr 来完成，则计算此值表达式，并将结果用作标识符，结果必须是一个串字面量。标识符最多

可以有 4 000 个字符，且不必是一个列名或列引用。标识符不能是表达式或空值。

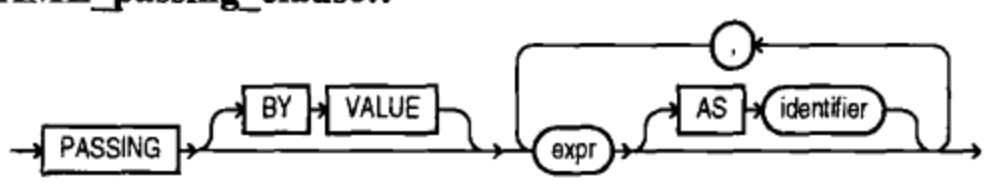
XMLQUERY

参阅：第 52 章。

格式：



XML_passing_clause::=



描述：XMLQUERY 函数允许查询 SQL 语句中的 XML 数据。它以 XQuery 表达式作为串字面量、可选的上下文条目和其他绑定变量，并返回使用这些输入值的 XQuery 表达式的结果。

XMLROOT

参阅：第 52 章。

格式：

```
XMLROOT
  ( value_expr, VERSION
  { value_expr | NO VALUE }
  [, STANDALONE { YES | NO | NO VALUE } ] )
```

描述：通过在已有 XML 值的 XML 根信息(前言)中提供版本和备份属性，XMLROOT 函数允许创建一个新的 XML 值。如果 value_expr 已经有前言，则数据库返回一个错误。如果输入是空，则函数返回值也为空。

XMLSEQUENCE

参阅：XMLCONCAT。

格式：

```
XMLSEQUENCE ( { XMLType_instance | sys_refcursor_instance [ , fmt] } )
```

描述：XMLSEQUENCE 具有两种格式：

- 第 1 种格式把 XMLType 实例作为输入，并返回 XMLType 中的最高级节点的可变数组。

- 第 2 种格式把 REFCURSOR 实例(和 XMLFORMAT 对象的可选实例)作为输入，并把游标的每行作为 XMLSEQUENCE 类型的 XML 文档返回。

XMLSERIALIZE

参阅： 第 52 章。

格式：

```
XMLSERIALIZE
  ( { DOCUMENT | CONTENT } value_expr [ AS datatype ]
    [ ENCODING xml_encoding_spec]
    { VERSION string_literal ]
    [ NO INDENT | { INDENT [SIZE = number] } ]
    [ { HIDE | SHOW } DEFAULTS ]
)
```

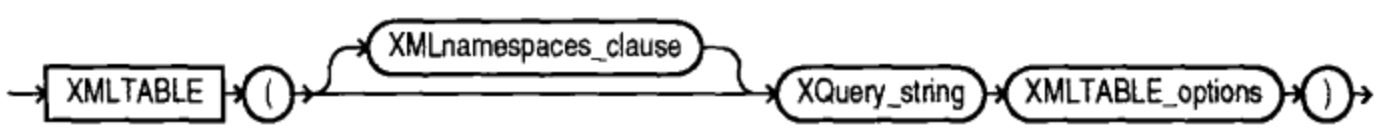
描述： XMLSERIALIZE 函数创建一个包含 value_expr 的内容的字符串或 LOB。

- 如果指定 document，则 value_expr 必须是一个有效的 XML 文档。
- 如果指定 content，则 value_expr 不必是只有单一根的 XML 文档。
- 指定的 datatype 可以是字符串类型(VARCHAR2 或 VARCHAR，但不能是 NVARCHAR 或 NVARCHAR2)、BLOB 或 CLOB。默认类型是 CLOB。
- 如果 datatype 是 BLOB，则可以指定 encoding 子句，以便在前言中使用指定的编码。
- 指定 version 子句，以便将提供的版本用作 XML 声明中(\<?xml version="..."...?\>)的 string_literal。
- 指定 no indent 将输出中所有无意义的空白字符取消掉。指定 indent size = N(这里 N 是一个总数)表示相对缩进 N 个空格可以使输出美观。如果 N 是 0，则美化式输出会在每个元素后面插入一个换行符，将每个元素单独放置在一行，但在输出中省去其他所有无意义的空格。如果使用 indent，却未指定 size，则缩进两个空格。如果省略此子句，则其行为是不确定的(无论是美化式输出还是非美化式输出)。
- hide defaults 和 show defaults 只适用于基于 XML 模式的数据。如果指定 show defaults，且输入数据中没有 XML 模式为其定义了默认值的任何可选元素或属性，那么这些元素或属性在输出中的值是其默认值。如果指定 hide defaults，则输出中不包含这样的元素或属性。hide defaults 是默认行为。

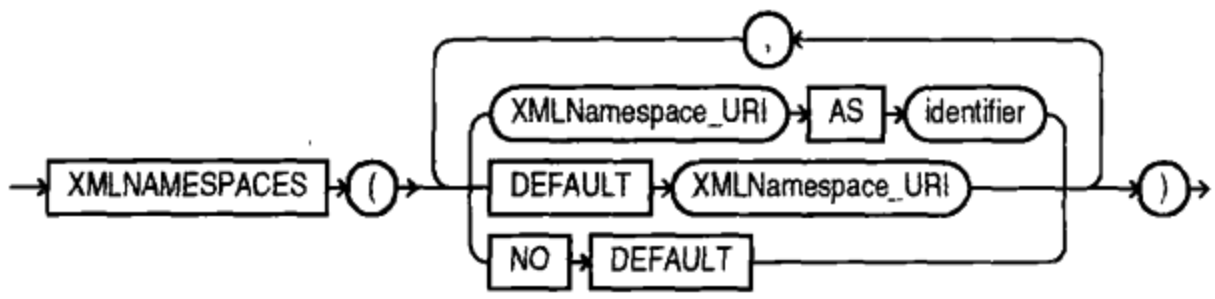
XMLTABLE

参阅： 第 52 章。

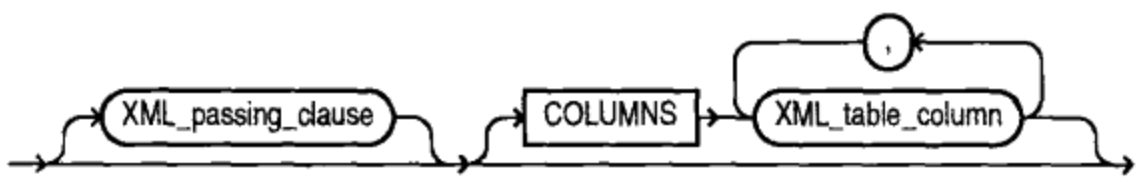
格式：



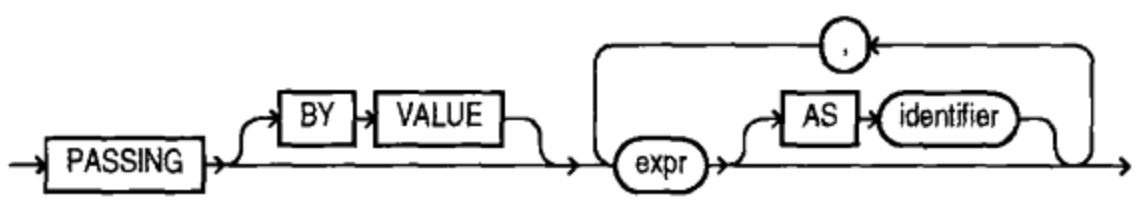
XMLnamespaces_clause::=



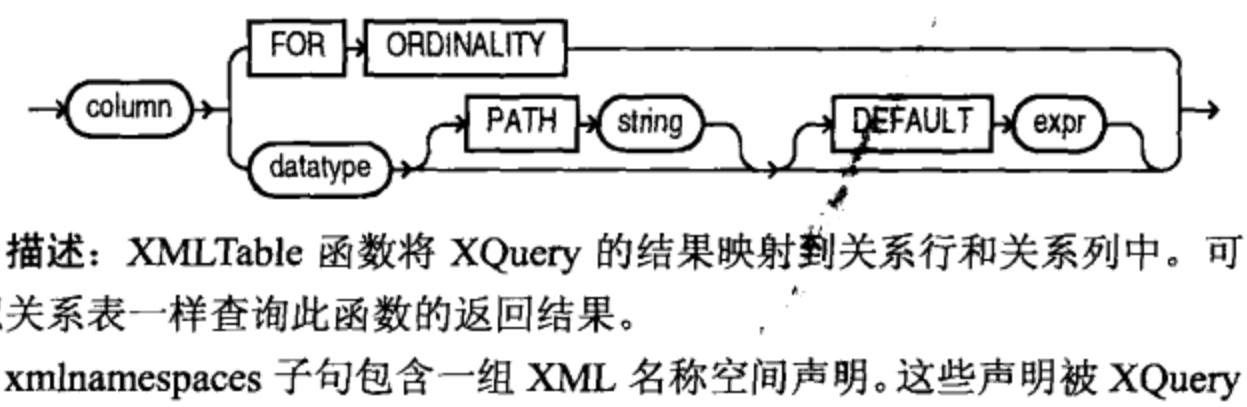
XMLTABLE_options::=



XML_passing_clause::=



XML_table_column::=



描述：XMLTable 函数将 XQuery 的结果映射到关系行和关系列中。可以像使用 SQL 的虚拟关系表一样查询此函数的返回结果。

xmlnamespaces 子句包含一组 XML 名称空间声明。这些声明被 XQuery 表达式(经过计算的 XQuery_string)和 XML_table_column 的 PATH 子句中的 XPath 表达式所引用，XQuery 表达式计算行，XPath 表达式为整个 XMLTable 函数计算列。如果想要在 columns 子句的 path 表达式中使用限定的名称，则需要指定 xmlnamespaces 子句。

XQuery_string 是一个完整的 XQuery 表达式，可以包含前言声明。

XML_passing_clause 中的 expr 是一个表达式，此表达式返回一个 XMLType 或返回 SQL 标量数据类型的一个实例，用作判断 XQuery 表达式的上下文。可以在 passing 子句中只指定一个 expr，而不指定标识符。每个 expr 的判断结果绑定到 XQuery_string 中相对应的标识符。如果任何 expr 的后面没有 AS 子句，则此表达式的判断结果用作判断 XQuery_string 的上下文条目。

可选的 columns 子句定义判断 XMLTable 将要创建的虚拟表的列。

XMLTRANSFORM

参阅：第 52 章。

格式：

```
XMLTRANSFORM ( XMLType_instance , XMLType_instance )
```

描述：XMLTRANSFORM 以一个 XMLType 实例和一个 XSL 样式表(它本身就是 XMLType 实例的一种格式)作为参数。它将样式表应用到实例上并返回一个 XMLType。

XMLType

XMLType 是在数据库中可用来存储和查询 XML 数据的数据类型。作为一种类型，XMLType 有成员函数，可使用称为 XPath 表达式的一类操作访问、提取和查询 XML 数据。SYS_XMLGEN、SYS_XMLAGG 和 DBMS_XMLGEN 程序包从已有的对象关系数据中创建 XMLType 值。当指定使用 XMLType 数据类型的一列时，Oracle 将以 CLOB 数据类型在内部存储该数据。

以下程序清单显示了如何创建了使用 XMLType 数据类型的表：

```
create table MY_XML_TABLE
(Key1 NUMBER,
Xml_Column SYS.XMLTYPE);

insert into MY_XML_TABLE (Key1, Xml_Column)
values (1, SYS.XMLTYPE.CREATEXML
  ('<book>
    <title>Complete Reference</title>
    <chapter num= "48">
      <title>Ending</title>
      <text>This is the end of the book.</text>
  </chapter>
</book>'));

select M.Xml_Column.GETCLOBVAL() as XML_Data
  from MY_XML_TABLE M
 where Key1 = 1;
```

关于 XMLType 及其相关方法的详细内容请参阅第 52 章。