



# 精通

# Oracle10g PL/SQL

王海亮 林立新  
于三禄 郑建茹 等编著

# 编程



中国水利水电出版社  
[www.waterpub.com.cn](http://www.waterpub.com.cn)

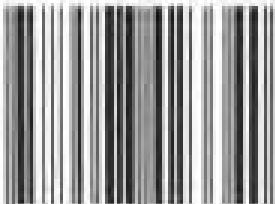
责任编辑：李立清 加工编辑：刘永道 封面设计：俞兆君

# 精通

## Oracle 10g PL/SQL 编程

- ◆ Oracle9i Pro\*C/C++ 编程指南
- ◆ Oracle9i 快速入门
- ◆ Oracle9i PL/SQL 从入门到精通
- ◆ Oracle9i JDeveloper 开发指南
- ◆ 精通 Oracle 10g PL/SQL 编程

ISBN 7-5084-2334-8



9 787508 423340 >



北京万水电信息有限公司

Beijing Multi-Channel Electronic Information Co., Ltd.

地址：北京市海淀区长春桥路5号新起点家园4号楼1706室

邮编：100089

电话：(010)8256-2819（总机）

传真：(010)8256-4371

E-mail: rochannel@263.net

ISBN 7-5084-2334-8

定价：48.00 元

万水 Oracle 技术丛书

# 精通 Oracle 10g PL/SQL 编程

王海亮 林立新 于三禄 郑建茹 等编著

中国水利水电出版社

## 内 容 提 要

PL/SQL 是 Oracle 特有的编程语言，它不仅具有过程编程语言的基本特征（循环、条件分支等），而且还具有对象编程语言的高级特征（重载、继承等）。

本书是专门为 Oracle 应用开发人员提供的编程指南。通过学习本书，读者不仅可以掌握 PL/SQL 的基础编程知识（嵌入 SQL 语句和编写子程序），而且还可以掌握 PL/SQL 的所有高级开发特征（使用记录类型、集合类型、对象类型和大对象类型）。另外，本书还为应用开发人员提供了大量 Oracle 9i 和 Oracle 10g 新增加的 PL/SQL 特征。

本书不仅适合于 PL/SQL 初学者，而且也适合于有经验的 PL/SQL 编程人员，本书还可以作为 Oracle 培训班的教材或者辅助材料。

### 图书在版编目（CIP）数据

精通 Oracle 10g PL/SQL 编程 / 王海亮等编著. —北京：中国水利水电出版社，2004

（万水 Oracle 技术丛书）

ISBN 7-5084-2334-8

I . 精… II . 王… III . 关系数据库—数据库管理系统， Oracle IV . TP311.138

中国版本图书馆 CIP 数据核字（2004）第 087353 号

书 名	精通 Oracle 10g PL/SQL 编程
作 者	王海亮 等编著
出版 发行	中国水利水电出版社（北京市三里河路 6 号 100044） 网址： <a href="http://www.waterpub.com.cn">www.waterpub.com.cn</a> E-mail：mchannel@263.net（万水） <a href="mailto:sales@waterpub.com.cn">sales@waterpub.com.cn</a> 电话：(010) 63202266（总机） 68331835（营销中心） 82562819（万水） 全国各地新华书店和相关出版物销售网点
经 销	北京万水电子信息有限公司 北京蓝空印刷厂 787mm×1092mm 16 开本 27.75 印张 680 千字 2004 年 9 月第 1 版 2004 年 9 月第 1 次印刷 0001—5000 册 48.00 元
排 版	北京万水电子信息有限公司
印 刷	北京蓝空印刷厂
规 格	787mm×1092mm 16 开本 27.75 印张 680 千字
版 次	2004 年 9 月第 1 版 2004 年 9 月第 1 次印刷
印 数	0001—5000 册
定 价	48.00 元

凡购买我社图书，如有缺页、倒页、脱页的，本社营销中心负责调换

版权所有·侵权必究

# 前　　言

本书是专门为 Oracle 应用开发人员所提供的编程指南。它不仅为应用开发人员提供了在 PL/SQL 块中嵌入 SQL 语句和编写子程序（过程、函数和包）的方法，而且还介绍了在 PL/SQL 块中使用记录变量（类似于 C/C++ 的结构）、集合类型（索引表、嵌套表、VARRAY —— 类似于 C/C++ 的数组）、对象类型（类似于 C++/Java 的类）和处理 LOB 对象的方法，为应用开发人员提供了大量 Oracle9i 和 Oracle 10g 新增加的 PL/SQL 特征。

## 读者对象

无论是对初学者，还是有经验的 PL/SQL 开发人员，本书都将成为不可缺少的一本有用的参考书。对于 PL/SQL 初学者来说，该书不仅为你介绍了 PL/SQL 开发工具和具体的开发方法，而且还提供了相应的习题和答案。对于 PL/SQL 开发人员，可能已经非常熟悉 PL/SQL 的基本开发方法，但可能对 PL/SQL 高级内容（集合类型、对象类型、LOB 对象处理）和 Oracle9i/Oracle 10g 的 PL/SQL 新特性知之较少，本书对于这些高级知识和新特性有非常详尽的介绍。另外本书还为 PL/SQL 开发人员提供了二十多个常用的 Oracle 系统包。

## 目标

在学习了本书之后，读者应该达到以下目标：

- 掌握 PL/SQL 开发工具包括 SQL\*Plus/iSQL\*Plus、PL/SQL Developer 以及 Procedure Builder 的使用方法。
- 掌握编写 SELECT 语句、DML 语句（INSERT, UPDATE, DELETE）和事务控制语句（COMMIT, ROLLBACK, SAVEPOINT）的方法，以及 SQL 函数的使用方法。
- 掌握在 PL/SQL 块中嵌入 SELECT 语句、DML 语句和事务控制语句的方法，以及在 PL/SQL 块中使用记录类型、集合类型和对象类型的方法。
- 掌握编写 PL/SQL 过程、函数、包和触发器的方法。

## 本书组织及特点

- 第 1 章：PL/SQL 综述。介绍 SQL 和 PL/SQL 的作用，PL/SQL 的优点，以及在 Oracle 10g, Oracle9i 中 PL/SQL 的新特征。
- 第 2 章：PL/SQL 开发工具。介绍 PL/SQL 的常用开发工具，如 SQL\*Plus/iSQL\*Plus、

PL/SQL Developer, Procedure Builder 的使用方法。

- 第 3 章：PL/SQL 基础。介绍 PL/SQL 应用程序的基础知识，包括 PL/SQL 块的结构分类、定义和使用变量、编写 PL/SQL 代码的规则，另外还介绍 Oracle 10g 的新特征——新数据类型 BINARY\_FLOAT 和 BINARY\_DOUBLE，以及指定字符串文本的新方法。
- 第 4 章：使用 SQL 语句。介绍使用简单 SELECT 语句、复杂 SELECT 语句（连接查询、子查询、数据分组、合并查询、层次查询等）、DML 语句（INSERT, UPDATE, DELETE）以及事务控制语句（COMMIT, ROLLBACK, SAVEPOINT）的方法。
- 第 5 章：SQL 函数。介绍所有的 SQL 函数，包括 Oracle9i 新增加的日期时间函数和 XML 函数，以及 Oracle 10g 新增加的 SCN\_TO\_TIMESTAMP, TIMESTAMP\_TO\_SCN 和集合函数等。
- 第 6 章：访问 Oracle。介绍在 PL/SQL 块中如何访问 Oracle 数据库，具体地介绍在 PL/SQL 块中嵌入 SELECT、INSERT、UPDATE 和 DELETE 语句，以及嵌入事务控制语句的方法。
- 第 7 章：编写控制结构。介绍如何使用 PL/SQL 控制结构，包括 IF 语句、LOOP 语句、GOTO 语句和 NULL 语句。另外还介绍 Oracle9i 的 PL/SQL 新特征——CASE 语句的作用和使用方法。
- 第 8 章：使用复合数据类型。介绍如何定义和使用 PL/SQL 记录变量、PL/SQL 索引表、嵌套表和 VARRAY 等复合数据类型。另外还介绍了 Oracle9i 在记录变量上的新特征——在 INSERT 语句和 UPDATE 语句中使用记录变量、FORALL 语句，以及 Oracle 10g 的新特征——集合嵌套、嵌套表的集合操作符和比较操作符，在 FORALL 语句中使用新子句 INDICES OF 和 VALUES OF 的方法。
- 第 9 章：使用游标。介绍如何使用显式游标、参数游标、游标 FOR 循环和游标变量。另外还介绍 Oracle9i 新增加的 PL/SQL 特征——使用嵌套游标。
- 第 10 章：处理例外。介绍预定义例外、非预定义例外和自定义例外的作用和使用方法，以及使用例外处理函数 RAISE\_APPLICATION\_ERROR, SQLCODE, SQLERRM 的方法。另外还介绍 Oracle 10g 的新特征——使用 PL/SQL 编译警告检测死代码和影响 PL/SQL 性能的代码。
- 第 11 章：开发子程序。介绍开发 PL/SQL 过程和函数的方法，另外还介绍使用输入参数、输出参数和输入输出参数的方法。
- 第 12 章：开发包。介绍开发 PL/SQL 包的方法，以及如何使用重载、构造过程和纯度级别等特征。
- 第 13 章：开发触发器。介绍 DML 触发器、INSTEAD-OF 触发器、系统事件触发器的作用及开发方法。
- 第 14 章：开发动态 SQL。介绍使用动态 SQL 处理非查询语句和查询语句的方法。还介绍 Oracle9i 的 PL/SQL 新特征——在动态 SQL 中使用 BULK 子句的方法。

- 第 15 章：使用对象类型。介绍对象类型的基本概念和作用，以及定义和使用简单对象类型、复杂对象类型的方法。还介绍 Oracle9i 的新特征——对象类型继承和自定义对象类型的构造方法。
- 第 16 章：使用 LOB 对象。介绍 CLOB、BLOB 和 BFILE 的作用以及使用方法。
- 第 17 章：使用 PL/SQL 系统包。介绍常用的 PL/SQL 系统包。

## 编著者相关书籍

《Oracle9i 快速入门》——王海亮、王海凤、张立民等编著，水利水电出版社。

《Oracle9i Pro\*C/C++编程指南》——王海亮、王海凤、张立民等编著，水利水电出版社

《Oracle9i 系统管理培训教程》——王海亮编著，机械工业出版社。

本书主要由王海亮、林立新、丁三禄、郑建茹等编著，另外，刘喜泉、宋和文、蒲建军、李新国、郝连奎、冯国庆、王乐天、武长毅、王宏斌、孙刚、王海凤、张立民、王宇欣等人也为编写本书提供了大量的资料和技术帮助。由于时间紧迫和编者水平有限，书中难免出现错误，敬请读者批评指正。本工作室人员都具有丰富的 Oracle 应用开发、培训和技术支持经验，曾经为电信、移动、联通、油田、银行、社保、证券期货、海关、教育等行业进行了 Oracle 技术支持和培训，并且获得用户的一致好评。如果您有 Oracle 应用开发、培训和技术支持需求，欢迎您来电来函与我们进行联系。

### 编著者

2004 年 3 月于呼和浩特

联系电话：0471-2210753

电子邮箱：whl88321@21cn.com

whl88321@163.com

# 目 录

## 前言

<b>第1章 PL/SQL 综述</b>	1
1.1 SQL 简介	1
1.2 PL/SQL 简介	3
1.3 Oracle 10g PL/SQL 新特征	5
<b>第2章 PL/SQL 开发工具</b>	8
2.1 SQL*Plus	8
2.2 PL/SQL Developer	12
2.3 Procedure Builder	14
2.4 习题	17
<b>第3章 PL/SQL 基础</b>	20
3.1 PL/SQL 块简介	20
3.1.1 PL/SQL 块结构	20
3.1.2 PL/SQL 块分类	22
3.2 定义并使用变量	25
3.2.1 标量变量	26
3.2.2 复合变量	29
3.2.3 参照变量	31
3.2.4 LOB 变量	32
3.2.5 非 PL/SQL 变量	33
3.3 编写 PL/SQL 代码	34
3.3.1 PL/SQL 词汇单元	34
3.3.2 PL/SQL 代码编写规则	37
3.4 习题	39
<b>第4章 使用 SQL 语句</b>	41
4.1 使用基本查询	41
4.1.1 简单查询语句	41
4.1.2 使用 WHERE 子句	48
4.1.3 使用 ORDER BY 子句	52
4.2 使用 DML 语句	55
4.2.1 插入数据	55
4.2.2 更新数据	58
4.2.3 删除数据	60
4.3 使用事务控制语句	61
4.3.1 事务和锁	61
4.3.2 提交事务	61
4.3.3 回退事务	62
4.3.4 只读事务	63
4.3.5 顺序事务	63
4.4 数据分组	64
4.4.1 分组函数	64
4.4.2 GROUP BY 和 HAVING	66
4.4.3 ROLLUP 和 CUBE	68
4.4.4 GROUPING SETS	70
4.5 连接查询	71
4.5.1 相等连接	72
4.5.2 不等连接	73
4.5.3 自连接	74
4.5.4 内连接和外连接	75
4.6 子查询	79
4.6.1 单行子查询	79
4.6.2 多行子查询	79
4.6.3 多列子查询	81
4.6.4 其他子查询	82
4.7 合并查询	85
4.8 其他复杂查询	86
4.9 习题	89
<b>第5章 SQL 函数</b>	92
5.1 数字函数	92
5.2 字符函数	96
5.3 日期时间函数	101
5.4 转换函数	105
5.5 集合函数	111

5.6 其他单行函数.....	112	9.1 显式游标.....	179
5.7 分组函数.....	121	9.2 参数游标.....	184
5.8 对象函数.....	126	9.3 使用游标更新或删除数据.....	184
5.9 习题 .....	127	9.4 游标 FOR 循环.....	187
<b>第 6 章 访问 Oracle .....</b>	<b>129</b>	9.5 使用游标变量.....	189
6.1 检索单行数据.....	129	9.6 使用 CURSOR 表达式.....	191
6.2 操纵数据.....	132	9.7 习题 .....	192
6.2.1 插入数据 .....	132		
6.2.2 更新数据 .....	133		
6.2.3 删除数据 .....	134		
6.2.4 SQL 游标.....	135		
6.3 事务控制语句.....	136		
6.4 习题 .....	137		
<b>第 7 章 编写控制结构 .....</b>	<b>139</b>		
7.1 条件分支语句.....	139		
7.2 CASE 语句 .....	142		
7.3 循环语句.....	144		
7.4 顺序控制语句.....	146		
7.5 习题 .....	147		
<b>第 8 章 使用复合数据类型 .....</b>	<b>149</b>		
8.1 PL/SQL 记录 .....	149		
8.1.1 定义 PL/SQL 记录 .....	149		
8.1.2 使用 PL/SQL 记录 .....	150		
8.2 PL/SQL 集合 .....	153		
8.2.1 索引表 .....	153		
8.2.2 嵌套表 .....	155		
8.2.3 变长数组 (VARRAY) .....	157		
8.2.4 PL/SQL 记录表 .....	158		
8.2.5 多级集合 .....	159		
8.2.6 集合方法 .....	161		
8.2.7 集合赋值 .....	164		
8.2.8 比较集合 .....	168		
8.3 批量绑定 .....	171		
8.3.1 FORALL 语句.....	172		
8.3.2 BULK COLLECT 子句.....	176		
8.4 习题 .....	177		
<b>第 9 章 使用游标 .....</b>	<b>179</b>		
9.1 显式游标.....	179		
9.2 参数游标.....	184		
9.3 使用游标更新或删除数据.....	184		
9.4 游标 FOR 循环.....	187		
9.5 使用游标变量.....	189		
9.6 使用 CURSOR 表达式.....	191		
9.7 习题 .....	192		
<b>第 10 章 处理例外 .....</b>	<b>194</b>		
10.1 例外简介.....	194		
10.2 处理预定义例外.....	196		
10.3 处理非预定义例外.....	202		
10.4 处理自定义例外.....	203		
10.5 使用例外函数.....	204		
10.6 PL/SQL 编译警告 .....	205		
10.7 习题 .....	208		
<b>第 11 章 开发子程序 .....</b>	<b>209</b>		
11.1 开发过程.....	209		
11.2 开发函数.....	214		
11.3 管理子程序.....	218		
11.4 习题 .....	222		
<b>第 12 章 开发包 .....</b>	<b>224</b>		
12.1 建立包.....	224		
12.2 使用包重载.....	229		
12.3 使用包构造过程.....	231		
12.4 使用纯度级别.....	233		
12.5 习题 .....	235		
<b>第 13 章 开发触发器 .....</b>	<b>237</b>		
13.1 触发器简介.....	237		
13.2 建立 DML 触发器.....	238		
13.2.1 语句触发器 .....	239		
13.2.2 行触发器 .....	242		
13.2.3 使用 DML 触发器.....	245		
13.3 建立 INSTEAD OF 触发器.....	248		
13.4 建立系统事件触发器.....	250		
13.5 管理触发器.....	253		
13.6 习题 .....	254		
<b>第 14 章 开发动态 SQL .....</b>	<b>256</b>		

14.1 动态 SQL 简介 .....	256	17.2 DBMS_JOB .....	317
14.2 处理非查询语句 .....	257	17.3 DBMS_PIPE .....	319
14.3 处理多行查询语句 .....	260	17.4 DBMS_ALERT .....	324
14.4 在动态 SQL 中使用 BULK 子句 ...	262	17.5 DBMS_TRANSACTION .....	327
14.5 习题 .....	265	17.6 DBMS_SESSION .....	329
<b>第 15 章 使用对象类型 .....</b>	<b>267</b>	17.7 DBMS_ROWID .....	332
15.1 对象类型简介 .....	267	17.8 DBMS_RLS .....	335
15.2 建立和使用简单对象类型 .....	271	17.9 DBMS_DDL .....	341
15.3 建立和使用复杂对象类型 .....	282	17.10 DBMS_SHARED_POOL .....	342
15.3.1 对象类型嵌套 .....	283	17.11 DBMS_RANDOM .....	343
15.3.2 参照对象类型 .....	286	17.12 DBMS_LOGMNR .....	344
15.3.3 对象类型继承 .....	288	17.13 DBMS_FLASHBACK .....	347
15.4 维护对象类型 .....	290	17.14 DBMS_OBFUSCATION _TOOLKIT .....	349
15.5 习题 .....	291	17.15 DBMS_SPACE .....	353
<b>第 16 章 使用 LOB 对象 .....</b>	<b>294</b>	17.16 DBMS_SPACE_ADMIN .....	356
16.1 LOB 简介 .....	294	17.17 DBMS_TTS .....	358
16.2 DBMS_LOB 包 .....	295	17.18 DBMS_REPAIR .....	359
16.3 访问 LOB .....	307	17.19 DBMS_RESOURCE_MANAGER	361
16.3.1 访问 CLOB .....	307	17.20 DBMS_STATS .....	369
16.3.2 访问 BLOB .....	310	17.21 UTL_FILE .....	378
16.3.3 访问 BFILE .....	312	17.22 UTL_INADDR .....	385
16.4 习题 .....	313		
<b>第 17 章 使用 Oracle 系统包 .....</b>	<b>315</b>	<b>附录 A 习题参考答案 .....</b>	<b>386</b>
17.1 DBMS_OUTPUT .....	315	<b>附录 B 使用 SQL*Plus .....</b>	<b>424</b>

# 第 1 章 PL/SQL 综述

PL/SQL (Procedural Language/SQL) 是 Oracle 在标准 SQL 语言上的过程性扩展。PL/SQL 不仅允许嵌入 SQL 语句，而且允许定义变量和常量，允许使用条件语句和循环语句，允许使用例外处理各种错误，从而提供了更加强大的功能。在允许运行 Oracle 的任何操作系统平台，都可以使用 PL/SQL。本章将介绍 PL/SQL 的特征、优点，以及 PL/SQL 在 Oracle 10g 和 Oracle9i 中的新特性，在学习了本章之后，读者应该了解：

- PL/SQL 的功能和作用；
- PL/SQL 的优点和特征；
- Oracle 10g、Oracle9i 的 PL/SQL 新特征。

## 1.1 SQL 简介

20 世纪 80 年代初，ANSI (American National Standards Institute) 数据库标准委员会开始制订有关关系语言的标准，但是直到 1986 年，数据库标准委员会才推出第一个 SQL 语言标准 SQL-86。随着数据库技术的发展，SQL 标准也在不断进行扩展和修正，并且数据库标准委员会先后又推出和制订了 SQL-89，SQL-92 以及 SQL-99 标准。1979 年，Relational Software 公司（Oracle 前身）首先向市场推出了 SQL 执行工具，Oracle 完全遵从 ANSI 的 SQL 标准，并且将最新的 SQL-99 标准集成到了 Oracle9i/Oracle 10g 数据库中。

SQL (Structured Query Language) 是关系数据库的基本操作语言，它是应用程序与数据库进行交互操作的接口。它将数据查询 (Data Query)、数据操纵 (Data Manipulation)、数据定义 (Data Definition) 和数据控制 (Data Control) 功能集于一体，从而使得应用开发人员、数据库管理员、最终用户都可以通过 SQL 语言访问数据库，并执行相应操作。

### 1. SQL 语言特点

- SQL 语言采用集合操作方式，对数据的处理是成组进行的，而不是一条一条处理。通过使用集合操作方式，可以加快数据的处理速度。
- 执行 SQL 语句时，每次只能发送并处理一条语句。如果要降低语句发送和处理次数，可以使用 PL/SQL。
- 执行 SQL 语句时，用户只需要知道其逻辑含义，而不需要关心 SQL 语句的具体执行步骤。例如，使用 WHERE 子句检索数据时，用户可以取得所需要的记录，而这些记录如何存储、如何检索不需要用户干预。Oracle 会自动优化 SQL 语句，确定最佳访问途径，执行 SQL 语句，最终返回实际数据。
- 使用 SQL 语句时，既可以采用交互方式执行（例如 SQL\*Plus），也可以将 SQL 语句嵌入到高级语言中执行（例如 PRO\*C/C++，PRO\*COBOL，SQLJ）。

### 2. SQL 语言分类

- 数据查询语言 (SELECT 语句)：用于检索数据库数据。在 SQL 所有语句中，SELECT

语句的功能和语法最复杂、最灵活。

- 数据操纵语言 (Data Manipulation Language, DML): 用于改变数据库数据, 包括 INSERT, UPDATE 和 DELETE 三条语句。其中 INSERT 语句用于将数据插入数据库中, UPDATE 语句用于更新已经存在的数据库数据, 而 DELETE 语句则用于删除已经存在的数据库数据。
- 事务控制语言 (Transactional Control Language, TCL): 用于维护数据的一致性, 包括 COMMIT, ROLLBACK 和 SAVEPOINT 三条语句。其中 COMMIT 语句用于确认已经进行的数据库改变, ROLLBACK 语句用于取消已经进行的数据库改变, 而 SAVEPOINT 语句则用于设置保存点, 以取消部分数据库改变。
- 数据定义语言 (Data Definition Language, DDL): 用于建立、修改和删除数据库对象。例如使用 CREATE TABLE 可以建表; 使用 ALTER TABLE 可以修改表结构; 使用 DROP TABLE 可以删除表。但是要注意, DDL 语句会自动提交事务。
- 数据控制语言 (Data Control Language, DCL): 用于执行权限授予和收回操作, 包括 GRANT 和 REVOKE 两条命令, 其中 GRANT 命令用于给用户或角色授予权限, 而 REVOKE 命令则用于收回用户或角色所具有的权限。但是要注意, DCL 语句会自动提交事务。

### 3. SQL 语句编写规则

- SQL 关键字不区分大小写, 既可以使用大写格式, 也可以使用小写格式, 或者混用大小写格式。示例如下:

```
SQL> SELECT ename,sal,job,deptno FROM emp;
SQL> select ename,sal,job,deptno from emp;
```

大小写均可

- 对象名和列名不区分大小写, 它们既可以使用大写格式, 也可以使用小写格式, 或者混用大小写格式。示例如下:

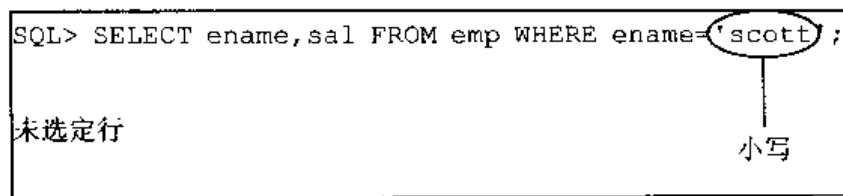
```
SQL> SELECT ename,sal,job,deptno FROM emp;
SQL> SELECT ENAME,SAL,JOB,DEPTNO FROM EMP;
```

大小写均可

- 字符值和日期值区分大小写。当在 SQL 语句中引用字符值和日期值时, 必须要给出正确的大小写数据, 否则不能返回正确信息。示例如下:

```
SQL> SELECT ename,sal FROM emp WHERE ename='SCOTT';
ENAME      SAL
-----  -----
SCOTT      3000
```

大写



- 在应用程序中编写 SQL 语句时，如果 SQL 语句文本很短，可以将语句文本放在一行上；如果 SQL 语句文本很长，可以将语句文本分布到多行上，并且可以通过使用跳格和缩进提高可读性。另外，在 SQL\*Plus 中 SQL 语句要以分号结束。示例如下：

**示例一：单行语句文本**

```
SELECT ename,sal FROM emp;
```

**示例二：多行语句文本**

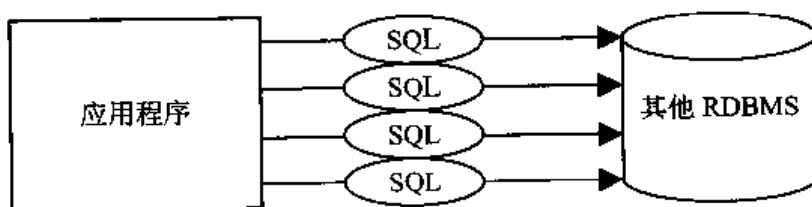
```
SELECT a.dname,b.ename,b.sal,b.comm,b.job
  FROM dept a RIGHT JOIN emp b
    ON a.deptno=b.deptno AND a.deptno=10;
```

## 1.2 PL/SQL简介

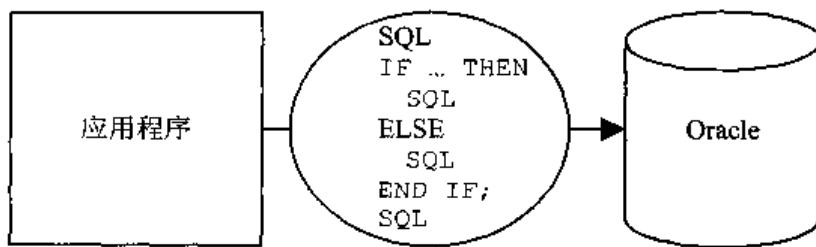
PL/SQL 是 Oracle 在标准 SQL 语言上的过程性扩展，它不仅允许嵌入 SQL 语句，而且允许定义变量和常量，允许过程语言结构（条件分支语句和循环语句），允许使用例外处理 Oracle 错误等，在运行 Oracle 的任何平台上应用开发人员都可以使用 PL/SQL。通过使用 PL/SQL，可以在一个 PL/SQL 块中包含多条 SQL 语句和 PL/SQL 语句。PL/SQL 具有以下一些优点和特征：

### 1. 提高应用程序的运行性能

在编写 Oracle 数据库应用程序时，开发人员可以直接将 PL/SQL 块内嵌到应用程序中，其最大优点是可以降低网络开销、提高应用程序性能。对于其他异质数据库（例如 SQL Server, Sybase, DB2 等），当应用程序访问 RDBMS 时，每次只能发送单条 SQL 语句。如下图所示：



可以看出，执行四条 SQL 语句需要在网络上发送四次。而对于 Oracle 数据库来说，通过使用 PL/SQL 块，可以将多条 SQL 语句组织到同一个 PL/SQL 块中，从而降低了网络开销，提高了应用程序的性能。如下图所示：



如上所示，通过使用 PL/SQL 块，在网络上只需要发送一次 PL/SQL 块，就可以完成所有 SQL 语句的发送和数据处理工作。

## 2. 提供模块化的程序设计功能

当开发数据库应用程序时，为了简化客户端应用程序的开发和维护工作，可以首先将企业规则或商业逻辑集成到 PL/SQL 子程序（过程、函数和包）中，然后在应用程序中调用子程序实现相应的程序功能。例如，如果应用程序需要取得雇员工资，那么可以建立函数实现该项功能。示例如下：

```

CREATE FUNCTION get_sal(no NUMBER)
RETURN NUMBER IS
    salary NUMBER(6,2);
BEGIN
    SELECT sal INTO salary FROM emp WHERE empno=no;
    RETURN salary;
END;
/

```

如上所示，当开发了该函数之后，应用程序就可以直接调用该函数返回特定雇员的工资，而不需要专门编写其他程序代码。如果商业逻辑或企业规则发生改变，那么只需要重新建立子程序就可以了，而不需要修改客户端的应用程序代码。

## 3. 允许定义标识符

当使用 PL/SQL 开发应用模块时，为了使得应用模块与应用环境实现数据交互，需要定义变量、常量、游标和例外等各种标识符。例如函数 get\_sal 中的 no 为输入参数，用于接收雇员编号的输入值，而 salary 变量则用于临时存储雇员工资。

## 4. 具有过程语言控制结构

PL/SQL 是 Oracle 在标准 SQL 上的过程性扩展，它不仅允许在 PL/SQL 块内嵌入 SQL 语句，而且允许在 PL/SQL 块中使用各种类型的条件分支语句和循环语句。示例如下：

```

DECLARE
    CURSOR emp_cursor IS SELECT ename,sal FROM emp FOR UPDATE;
    emp_record emp_cursor%ROWTYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF emp_record.sal<2000 THEN
            UPDATE emp SET sal=sal*1.1 WHERE CURRENT OF emp_cursor;
        END IF;
    END LOOP;
END;
/

```

```
    END LOOP;
END;
/
```

该PL/SQL块使用了循环（LOOP语句）取得所有雇员的姓名和工资，并且使用了条件控制语句（IF）判断雇员工资，如果工资低于2000，则给该雇员增加10%的工资。

### 5. 具有良好的兼容性

PL/SQL是Oracle所提供的用于实现应用模块的语言，在允许运行Oracle的任何平台上都可以使用PL/SQL。例如，不仅可以在Oracle数据库中使用PL/SQL开发数据库端的过程、函数和触发器，也可以在Oracle所提供的应用开发工具Developer中使用PL/SQL开发客户端的过程、函数和触发器。

### 6. 处理运行错误

当设计并开发应用程序时，为了提高应用程序的健壮性，避免应用程序运行的异常问题，开发人员应该想办法处理应用程序可能出现的各种运行错误。通过使用PL/SQL所提供的例外（EXCEPTION），开发人员可以集中处理各种Oracle错误和PL/SQL错误，从而简化了错误处理。示例如下：

```
DECLARE
    name VARCHAR2(10);
BEGIN
    SELECT ename INTO name FROM emp WHERE empno=&no;
    dbms_output.put_line(name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('该雇员不存在');
END;
/
```

## 1.3 Oracle 10g PL/SQL新特征

### 1. Oracle 10.1新特征

- 通过在Oracle 10g数据库中引入新的PL/SQL编译器，Oracle极大地加强了PL/SQL的执行性能，并且PL/SQL的性能增强是由系统自动完成的，而不需要开发人员进行任何处理。
- 在Oracle9i中，当使用FORALL语句执行批量SQL操作时，只能在PL/SQL集合类型（嵌套表、VARRAY）上使用连续下标；而从Oracle 10g开始，通过使用INDICES OF子句和VALUES OF子句，就可允许在PL/SQL集合类型（嵌套表、VARRAY）上使用非连续下标。
- 从Oracle 10.1开始，在编写PL/SQL块时可允许开发人员使用IEEE754所提供的新数据类型BINARY\_FLOAT和BINARY\_DOUBLE。
- 从Oracle 10.1开始，Oracle为嵌套表提供了更多新的操作，包括比较嵌套表是否相等、检测某元素是否为嵌套表成员、检测一个嵌套表是否是另一个嵌套表的子集、在嵌套表上执行集合操作等等。

- 在 Oracle 10g 之前，当编译 PL/SQL 子程序时，只要子程序符合 PL/SQL 的语法规则，就可以成功地编译子程序。例如，当编写了包含死代码（从未执行的代码）或不可预料结果的子程序时，Oracle 不会为开发人员提供任何附加信息。而从 Oracle 10.1 开始，通过使用新的初始化参数 PLSQL\_WARNINGS 和 PL/SQL 包 DBMS\_WARNING，在编写了包含死代码（从未执行的代码）或不可预料结果的子程序时，Oracle 会生成警告信息，从而使得开发人员可以编写更加高效、健壮的 PL/SQL 代码。
- 在 Oracle 10.1 之前，当为字符串变量赋值时，必须要用两个单引号，如果在字符串数值中需要包含单引号，那么必须要使用两个单引号来表示一个单引号值；而从 Oracle 10.1 开始，如果在字符串数值中需要包含单引号，那么开发人员可以在字符串中使用其他分隔符（{}、[]、<>）包围字符串，使得开发人员可以在字符串中直接将单引号作为字符值引用。
- 从 Oracle 10.1 开始，Oracle 可以隐含地将 CLOB 数据转换为 NCLOB 数据，也可以将 NCLOB 数据隐含地转换为 CLOB 数据，但为了加快数据转换速度，用户可以继续使用 TO\_CLOB 和 TO\_NCLOB 函数进行类型转换。
- 从 Oracle 10.1 开始，Oracle 增加了新的倒叙查询函数 SCN\_TO\_TIMESTAMP 和 TIMESTAMP\_TO\_SCN。通过使用函数 SCN\_TO\_TIMESTAMP，可以根据 SCN 值取得相应的日期时间值；通过使用函数 TIMESTAMP\_TO\_SCN，可以根据日期时间值取得相应的 SCN 值。

### 2. Oracle 9.2 新特征

- 在 Oracle 9.2 之前，当使用 PL/SQL 记录执行 INSERT 或 UPDATE 操作时，必须要使用记录属性为相应列提供数据；而从 Oracle 9.2 开始，当使用 PL/SQL 记录执行 INSERT 或 UPDATE 操作时，开发人员就可以直接引用记录变量插入或更新数据。
- 在 Oracle 9.2 中，当在 PL/SQL 块内执行 SELECT 操作时，可允许直接使用 PL/SQL 记录表接收多行多列的数据，而不需要分别为每个列指定相应的 PL/SQL 表。
- 在 Oracle 9.2 中，当定义集合类型（PL/SQL 表、VARRAY、嵌套表）时，可允许使用 VARCHAR2 值作为下标，提供了类似于 Perl 语言散列表（hash table）的功能。
- 在 Oracle 9.2 之前，不允许开发人员定义对象类型的构造方法，当初始化对象实例时，开发人员只能使用系统默认的对象构造方法；而从 Oracle 9.2 开始，Oracle 不仅允许开发人员定义对象类型的构造方法，而且还允许重载对象类型的构造方法，这样在初始化对象实例时，就可以引用需要的构造方法。
- 在 Oracle 9.2 中，Oracle 为包 UTL\_FILE 增加了一些新的函数，使得应用开发人员可以在 PL/SQL 中执行各种通用的文件操作。
- 在 Oracle 9.2 中，Oracle 为对象类型增加了 TREAT 函数，使得在调用对象方法时可以动态选择要使用的继承对象类型。

### 3. Oracle 9.0.1 新特征

- 在 Oracle 9.0.1 中，Oracle 集成了 SQL 和 PL/SQL 解析器，使得 PL/SQL 可以支持 SQL 语句的完整语法。
- 在 Oracle 9.0.1 中，Oracle 提供了 CASE 语句和表达式，简化了多重分支语句的处理

(类似于 C 语言的 SWITCH 语句)。

- 在 Oracle 9.0.1 中, 通过提供对象类型继承, Oracle 增强了面向对象程序设计 (OOP) 的支持, 并使得应用开发人员可以定义父类型/子类型。
- 在 Oracle 9.0.1 中, Oracle 允许在定义了对象类型之后增加、删除对象类型的属性和方法。
- 在 Oracle 9.0.1 中, Oracle 增加了新的日期时间类型, 包括 TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH 等。
- 在 Oracle 9.0.1 中, Oracle 提供了 TABLE 函数, 简化了对集合数据 (嵌套表和 VARRAY) 的处理。
- 在 Oracle 9.0.1 中, Oracle 提供了 CURSOR 表达式, 使得在 PL/SQL 块中可以嵌套游标。
- 在 Oracle 9.0.1 中, Oracle 允许嵌套集合类型, 使得应用开发人员通过使用集合嵌套提供对多维数据的支持。
- 在 Oracle 9.0.1 中, 增强了对 LOB 类型数据的支持, 使得开发人员可以在字符函数中引用 CLOB 和 NCLOB 数据, 将 LONG 类型数据升级为 LOB 类型。
- 在 Oracle 9.0.1 中, 允许执行批量 SQL 操作, 包括批量提取、批量插入和批量更新操作, 而且批量 SQL 操作是使用 FORALL 语句来实现的。在 Oracle 9.0 之前, 如果在 PL/SQL 块使用 SELECT INTO, 每次只能返回一行数据; 而从 Oracle 9.0 开始, 通过使用 SELECT BULK COLLECT INTO 操作, 一次就可以查询多行数据到集合变量中。

## 第2章 PL/SQL 开发工具

在学习使用 PL/SQL 语言开发 PL/SQL 应用程序之前，应用开发人员应该首先了解各种常用的 PL/SQL 开发工具，并学会使用这些工具编辑、编译、调试和运行 PL/SQL 应用程序。本章将介绍各种常用的 PL/SQL 开发工具，在学习了本章之后，大家应该能够完成以下任务：

- 学会使用 SQL\*Plus；
- 学会使用 PL/SQL Developer；
- 学会使用 Procedure Builder。

### 2.1 SQL\*Plus

SQL\*Plus 是 Oracle 公司提供的一个工具程序，它用于运行 SQL 语句和 PL/SQL 块，并且用于跟踪调试 SQL 语句和 PL/SQL 块。该工具程序不仅可以在命令行运行，也可以在 Windows 窗口环境中运行。从 Oracle9i 开始，Oracle 公司还提供了在 Web 页面中运行 SQL\*Plus 的工具——iSQL\*Plus。下面详细介绍使用这些工具程序的方法。

#### 1. 在命令行运行 SQL\*Plus

在命令行运行 SQL\*Plus 是使用 sqlplus 命令来完成的，该命令的语法如下：

```
sqlplus [username]/[password]@[server]
```

其中，username 用于指定数据库用户名，password 用于指定用户口令，而 server 则用于指定主机字符串（网络服务名）。当连接到本地数据库时，不需要提供网络服务名。示例如下：

```
D:\>sqlplus scott/tiger
SQL*Plus: Release 10.1.0.2.0 - Production on 星期日 3月 28 15:20:31 2004
Copyright (c) 1982, 2004, Oracle. All rights reserved.
连接到:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

当在客户端使用 SQL\*Plus 连接到远程数据库时，必须使用 Net Manager 配置网络服务名，并且使用该网络服务名连接到远程数据库。示例如下：

```
D:\>sqlplus scott/tiger@orc1 —————— 网络服务名
SQL*Plus: Release 10.1.0.2.0 - Production on 星期日 3月 28 15:20:31 2004
Copyright (c) 1982, 2004, Oracle. All rights reserved.
连接到:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

当连接到数据库之后，就可以开发、测试并执行 PL/SQL 块了。下面是在 SQL\*Plus 命令行中编写并执行过程的示例：

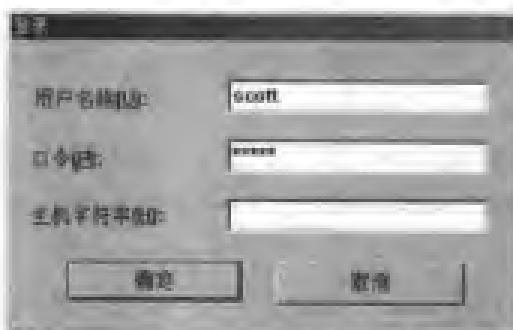
```

SQL> CREATE PROCEDURE insert_dept (no NUMBER, name VARCHAR2)
  2 IS
  3 BEGIN
  4   INSERT INTO dept (deptno,dname) VALUES (no, name);
  5 END;
  6 /
过程已创建。
SQL> exec insert_dept(50, 'SALES')
PL/SQL 过程已成功完成。

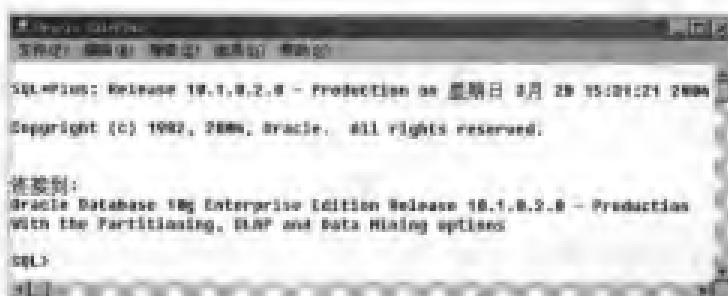
```

## 2. 在 Windows 环境中运行 SQL\*Plus

如果在 Windows 95/98/NT/2000/XP 平台上安装了 Oracle 数据库产品，那么可以在窗口环境中运行 SQL\*Plus。具体方法为“开始→程序→Oracle-OraHome10→Application Development→SQL Plus”，或者在命令行执行 sqlplusw 命令。当运行了窗口环境的 SQL\*Plus 之后，会弹出如下窗口：



如图所示，在输入了用户名和口令之后，单击“确定”按钮就可以连接到本地数据库了。如果要连接到远程数据库，还必须在“主机字符串”处输入网络服务名。当连接到 SQL\*Plus 之后，会显示如下窗口界面：



在成功地连接到数据库之后，就可以编辑、编译并执行 PL/SQL 块了。下面是在 SQL\*Plus 窗口界面中建立并执行函数的示例：

```

SQL> CREATE OR REPLACE FUNCTION get_sal(name VARCHAR2)
  2 RETURN NUMBER IS
  3   v_sal NUMBER(6,2);
  4 BEGIN
  5   SELECT sal INTO v_sal FROM emp WHERE upper(ename)=upper(name);
  6   RETURN v_sal;
  7 END;

```

```

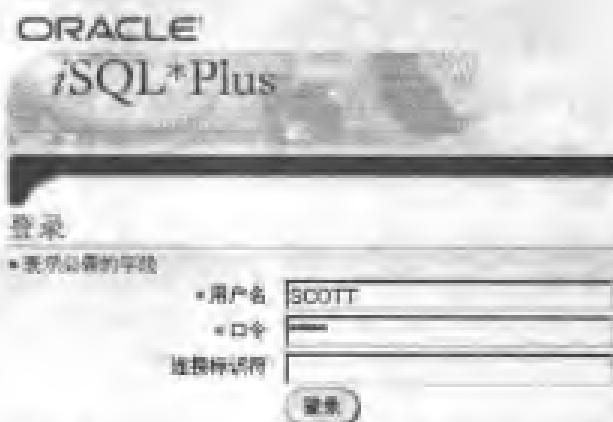
8 /
函数已创建。
SQL> VAR salary NUMBER
SQL> exec :salary:=get_sal('scott')
PL/SQL 过程已成功完成。
SQL> PRINT salary
      SALARY
-----
      3000

```

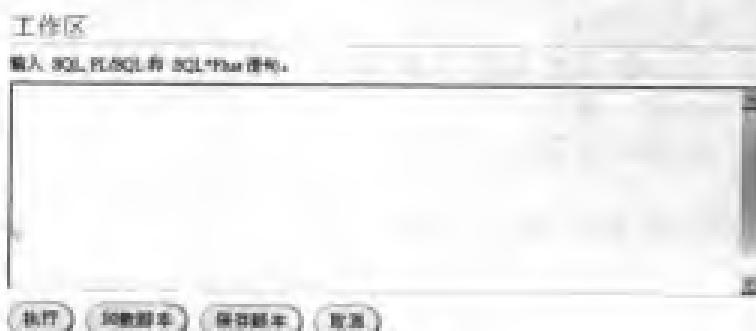
### 3. iSQL\*Plus

iSQL\*Plus 是 Oracle9i 新增加的特征，它是 SQL\*Plus 在浏览器中的实现方式。在 Oracle9i 中，为了在浏览器中运行 iSQL\*Plus，必须首先在 Oracle Server 端启动 HTTP Server，然后才能在客户端使用浏览器连接到 iSQL\*Plus；而在 Oracle 10g 之中，为了在浏览器中运行 iSQL\*Plus，必须首先在 Oracle Server 端使用 isqlplusctl start 命令启动 iSQL\*Plus 应用服务器，然后才能在客户端使用浏览器连接到 iSQL\*Plus。

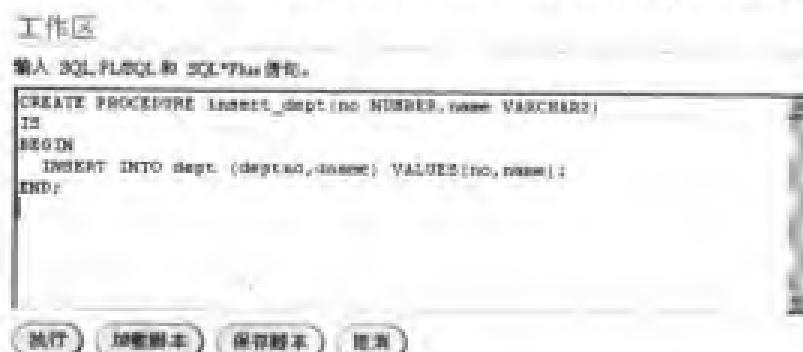
当运行 iSQL\*Plus 时，首先启动浏览器，然后在地址栏中输入 URL 地址（URL 地址格式：<http://主机名:端口号/isqlplus>），其中主机名用于指定 Oracle Server 所在的机器名或 IP 地址，而端口号则用于指定 iSQL\*Plus 的监听端口号（Oracle 10g 的默认 iSQL\*Plus 端口号为 5560）。当输入了正确地址（例如，<http://wanghailiang:5560/isqlplus>）之后，会启动 iSQL\*Plus，如下图所示：



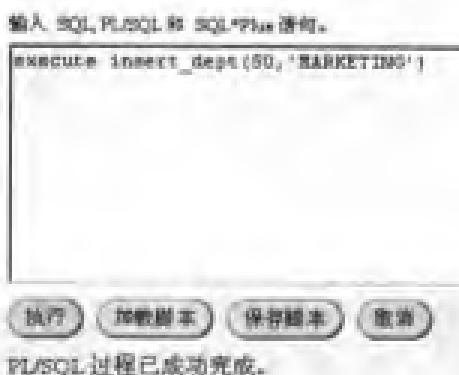
如图所示，当启动了 iSQL\*Plus 之后，输入数据库用户名、口令和连接标识符（网络服务器名），然后单击“登录”就可以连接到相应的数据库。在连接到数据库之后，会显示如下工作区图形界面：



如图所示，当在iSQL\*Plus工作区中输入了SQL、PL/SQL和SQL\*Plus语句之后，单击“执行”按钮，就会执行相应的SQL、PL/SQL和SQL\*Plus语句。使用iSQL\*Plus建立过程的示例如下：



在iSQL\*Plus中执行该过程，只要在“输入语句”部分输入 execute 命令，然后单击“执行”按钮即可。示例如下：



PL/SQL 过程已成功完成。

#### 4. 在SQL\*Plus中检测PL/SQL错误

通过在SQL\*Plus中执行SHOW ERRORS命令，可以检测PL/SQL错误所在行以及错误的原因。示例如下：

```
SQL> CREATE PROCEDURE insert_dept (no NUMBER, name VARCHAR2)
  2 IS
  3 BEGIN
  4   INSERT INTO dept (deptno, dname) VALUES (no, name)
  5 END;
  6 /
```

警告：创建的过程带有编译错误。

```
SQL> show errors
PROCEDURE INSERT_DEPT 出现错误:
LINE/COL  ERROR
-----  

4/3      PL/SQL: SQL Statement ignored
4/51     PL/SQL: ORA-00933: SQL命令未正确结束
5/4      PLS-00103: 出现符号 "end-of-file" 在需要下列之一时:
begin case declare
end exception exit for goto if loop mod null pragma raise
return select update while with <an identifier>
```

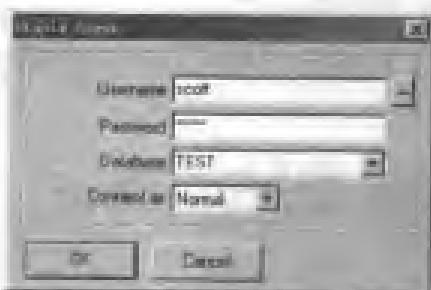
```

<a double-quoted delimited-identifier> <a bind variable> <<
close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge
<a single-quoted SQL string> pipe

```

## 2.2 PL/SQL Developer

PL/SQL Developer 是用于开发 PL/SQL 子程序的集成开发环境 (IDE)，它是一个独立产品，而不是 Oracle 10g 的附带产品。当运行 PL/SQL Developer 时，首先会弹出登录窗口。如下图所示：



在输入了用户名、口令和网络服务名之后，单击“OK”按钮就可以连接到数据库，并且会显示如下图形界面：



PL/SQL Developer 不仅实现了 SQL\*Plus 的所有功能，而且还可以用于跟踪和调试 PL/SQL 程序、监视和调整 SQL 语句的性能。该工具程序提供了多种窗口工具界面，以实现不同功能，在这里只介绍与开发 PL/SQL 子程序相关的窗口界面。

### 1. Command Window

Command Window 实现了 SQL\*Plus 的所有功能，它不仅允许运行 SQL\*Plus 命令和 SQL 命令，而且允许编辑、编译并运行 PL/SQL 块。当运行该窗口界面时，首先选择“File—>New—>Command Window”启动窗口界面 Command Window，然后就可以在该命令窗口中编辑、编译并运行 PL/SQL 块了，编辑、编译、执行 PL/SQL 块的方法与在 SQL\*Plus 中的方法完全相同。示例如下：

```

SQL> CREATE PROCEDURE insert_dept (no NUMBER, name VARCHAR2)
  2 IS
  3 BEGIN
  4   INSERT INTO dept (deptno,dname) VALUES (no,name);
  5 END;
  6 /
Procedure created.

SQL> exec insert_dept (50,'SALES');

PL/SQL procedure successfully completed.

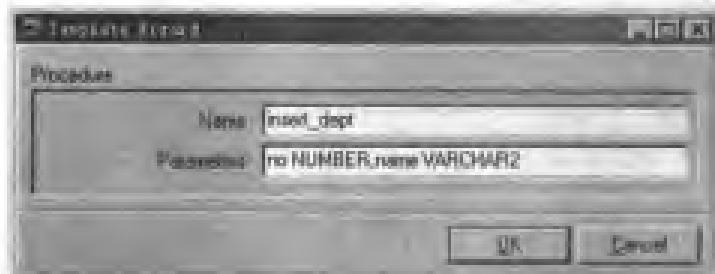
SQL>

```

The status bar at the bottom right shows: PL/SQL procedure successfully completed.

## 2. Program Window

Program Window 允许建立和修改各种子程序，包括过程、函数、包、触发器以及对象类型。例如，如果要建立过程，则依次选择“File—>New—>Program Window—>Procedure”选项，此时会显示如下对话框：



如上所示，在输入了过程名及其参数之后，单击“OK”按钮，就会进入“Program Window”界面。在程序窗口界面中，输入相应的 PL/SQL 代码，然后按“F8”键即可编译该过程。如下图所示：

```

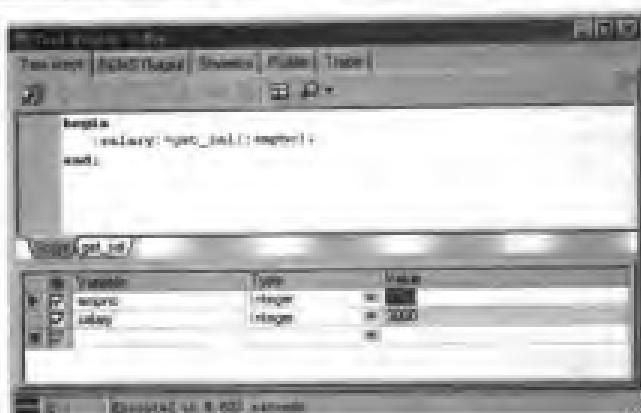
create or replace procedure insert_dept (no NUMBER, name VARCHAR2) is
begin
  INSERT INTO dept (deptno,dname) VALUES (no,name);
end insert_dept;

```

The status bar at the bottom right shows: Compiled successfully.

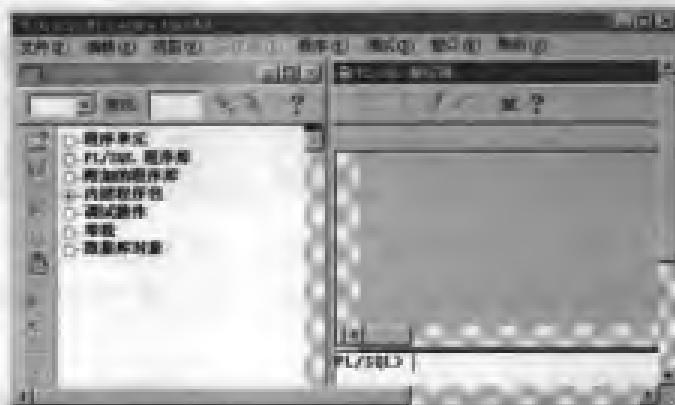
## 3. Test Window

Test Window 用于测试并跟踪 PL/SQL 子程序的运行结果。当使用该窗口时，首先依次选择“File—>New—>Test Window”选项，启动该窗口界面，然后在 Test Script 中编写要执行的 PL/SQL 代码。在编写了 PL/SQL 代码之后，在变量栏可以指定输入变量和输出变量，然后按“F9”键即可跟踪 PL/SQL 代码的运行过程及输出结果了。如下图所示：



## 2.3 Procedure Builder

Procedure Builder 是 Oracle 应用开发工具 Developer 所提供的一个产品，它可以用于建立、执行和调试 PL/SQL 子程序。使用该工具不仅可以在服务器端的 PL/SQL 子程序，而且可以用于开发客户端的 PL/SQL 子程序。当运行该 GUI（Graphics User Interface）工具时，会显示出如下图形界面：



Procedure Builder 工具主要由对象导航器（Object Navigator）、PL/SQL 解析器（PL/SQL Interpreter）、程序单元编辑器（Program Unit Editor）、存储程序单元编辑器（Stored Program Unit Editor）、数据库触发器编辑器（Database Trigger Editor）等五个部分组成。下面介绍如何在 Procedure Builder 工具程序中连接到数据库，并说明该工具每个组成部分的作用。

### 1. 连接到数据库

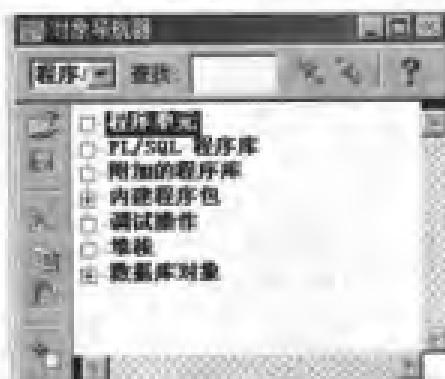
当使用 Procedure Builder 开发存储程序或数据库触发器时，必须首先连接到数据库。选择“文件→连接”，会弹出连接对话框，如下图所示：



此时在输入了用户名、口令和数据库（网络服务名）之后，单击“连接”按钮就可以连接到数据库了。

## 2. 对象导航器

对象导航器提供了访问开发环境对象的方法。通过使用对象导航器，不仅可以用于建立并操纵程序单元、PL/SQL 程序库，而且还可以用于浏览 PL/SQL 程序单元以及数据库对象。对象导航器以树型结构列出可以访问的所有对象，如下图所示：



- 程序单元：用于建立客户端的 PL/SQL 子程序。
- PL/SQL 程序库：用于将客户端的 PL/SQL 子程序组织到 OS 文件或数据库中。
- 附加的程序库：用于将 PL/SQL 程序库添加到开发环境中，从而使得其他程序单元可以引用该程序库中的 PL/SQL 程序。
- 内建程序包：Developer 所提供的 PL/SQL 程序包，客户端的 PL/SQL 程序单元可以直接引用这些包中的过程和函数。
- 调试操作：用于监视 PL/SQL 程序单元的执行。
- 堆栈：用于显示执行调试操作的子程序调用入口。
- 数据库对象：用于浏览服务器端的 PL/SQL 子程序、PL/SQL 程序库、表和视图等数据库对象。

## 3. PL/SQL 解析器

PL/SQL 解析器用于定义、显示、运行 SQL 语句和 PL/SQL 程序单元。运行 PL/SQL 解析器，只需选择“程序—>PL/SQL 解析器”，就可以运行 SQL 语句和 PL/SQL 程序单元了。如下图所示：

The screenshot shows the PL/SQL Parser window with the following content:

```
PL/SQL> CREATE TABLE test(id INT);
PL/SQL> PROCEDURE insert_test(id INT)
  IS
  BEGIN
    INSERT INTO test VALUES(id);
  END;
PL/SQL> insert_test(1);
PL/SQL> commit;
PL/SQL>
```

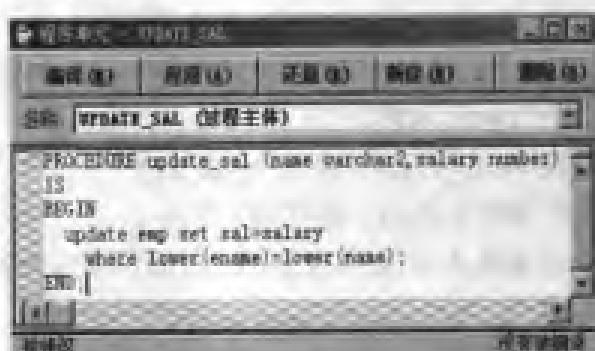
图中第一条语句用于建立 TEST 表 (DDL); 第二条语句建立客户端的过程 insert\_test; 第三条语句运行过程 insert\_test, 最后一条语句提交了事务。需要注意, 因为上述语句访问了数据库, 所以必须首先连接到数据库, 然后才能执行如上操作。

#### 4. 程序单元编辑器

程序单元编辑器用于建立、编辑、编译客户端的 PL/SQL 程序单元。运行程序单元编辑器, 只需选择“程序—>程序单元编辑器”。在进入程序单元编辑器之后, 即可建立新的 PL/SQL 程序单元, 单击“新建”按钮会显示如下对话框:



在给出了程序单元的名称之后, 单击“确定”按钮, 即可以在程序单元编辑器中编辑并编译 PL/SQL 程序单元。如下图所示:



#### 5. 存储程序单元编辑器

存储程序单元编辑器用于建立、编辑、编译服务器端的 PL/SQL 程序单元。需要注意, 当使用存储程序单元编辑器时, 必须首先要连接到数据库。运行存储程序单元编辑器, 只需选择“程序—>存储程序单元编辑器”。在进入存储程序单元编辑器之后, 可以建立新的 PL/SQL 程序单元, 单击“新建”按钮会显示如下对话框:



在给出了程序单元的名称之后，单击“确定”按钮，即可在程序单元编辑器中编辑并编译 PL/SQL 程序单元。如下图所示：



## 6. 数据库触发器编辑器

数据库触发器编辑器用于建立和编辑数据库触发器。但大家需要注意，当使用数据库触发器编辑器时，必须首先要连接到数据库。运行数据库触发器编辑器，只需选择“程序→数据库触发器编辑器”，并在对话框中指定触发器名称、触发操作以及触发条件等即可。如下图所示：



## 2.4 习题

- 建立练习表 CUSTOMER, PRODUCT, ORD 和 ITEM，并分别插入数据。
- 使用 SQL\*Plus 建立过程 add\_ord 并调用该过程。

\* 建立过程 add\_ord

```

CREATE OR REPLACE PROCEDURE add_ord(
    v_ordid NUMBER,v_orddate DATE,v_custid NUMBER,
    v_shipdate DATE,v_total NUMBER)
IS
    e_integrity EXCEPTION;
    e_shipdate EXCEPTION;
  
```

```

PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
    IF v_shipdate>=v_orddate THEN
        INSERT INTO ord VALUES(v_ordid,v_orddate,
                               v_custid,v_shipdate,v_total);
    ELSE
        RAISE e_shipdate;
    END IF;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        raise_application_error(-20001,'该订单已存在');
    WHEN e_integrity THEN
        raise_application_error(-20002,'该客户不存在');
    WHEN e_shipdate THEN
        raise_application_error(-20003,'交付日期不能早于预订日期');
END;
/

```

- 调用过程 add\_ord

```

exec add_ord(600,'02-1月-04',215,'10-1月-04',100)
exec add_ord(652,'02-1月-04',266,'10-1月-04',100)
exec add_ord(652,'10-1月-04',210,'02-1月-04',100)
exec add_ord(652,'02-1月-04',221,'10-1月-04',100)
COMMIT;

```

### 3. 使用 PL/SQL Developer 建立过程 upd\_shipdate 并调用该过程。

- 使用 Program Window 窗口建立过程 upd\_shipdate

```

create or replace procedure upd_shipdate
    (v_ordid NUMBER,v_shipdate DATE)
is
    e_no_row EXCEPTION;
begin
    UPDATE ord SET ship_date=v_shipdate WHERE ord_id=v_ordid;
    IF SQL%NOTFOUND THEN
        RAISE e_no_row;
    END IF;
EXCEPTION
    WHEN e_no_row THEN
        raise_application_error(-20004,'该订单不存在');
end upd_shipdate;

```

- 使用 Command Window 调用过程 upd\_shipdate

```

exec upd_shipdate(653,'12-1月-94')
exec upd_shipdate(652,'12-1月-94')
COMMIT;

```

### 4. 使用 Procedure Builder 建立过程 del\_ord，并调用该过程。

- 使用“存储程序单元编辑器”建立过程 del\_ord

```
PROCEDURE del_ord(v_ordid NUMBER) IS
```

```
e_integrity EXCEPTION;
PRAGMA EXCEPTION_INIT(e_integrity,-2292);
BEGIN
    DELETE FROM ord WHERE ord_id=v_ordid;
    IF SQL%NOTFOUND THEN
        raise_application_error(-20005,'该订单不存在');
    END IF;
EXCEPTION
    WHEN e_integrity THEN
        raise_application_error(-20006,'该订单存在相应条款');
END;
```

- 使用“PL/SQL 解析器”调用过程 del\_ord

```
del_ord(610);
del_ord(788);
del_ord(652);
COMMIT;
```

# 第3章 PL/SQL 基础

开发人员使用 PL/SQL 编写应用模块时，不仅需要掌握 SQL 语句的编写方法，而且必须要掌握 PL/SQL 语句及其语法规则。本章将介绍 PL/SQL 的基础知识，在学习了本章之后，读者应该完成以下任务：

- 了解 PL/SQL 块的基本结构以及 PL/SQL 块的分类；
- 学会在 PL/SQL 块中定义和使用变量；
- 学会在 PL/SQL 块中编写可执行语句；
- 了解编写 PL/SQL 代码的指导方针；
- 了解 Oracle 10g 的新特征——新数据类型 BINARY\_FLOAT 和 BINARY\_DOUBLE，以及指定字符串文本的新方法。

## 3.1 PL/SQL 块简介

块（Block）是 PL/SQL 的基本程序单元，编写 PL/SQL 程序实际就是编写 PL/SQL 块。要完成相对简单的应用功能，可能只需要编写一个 PL/SQL 块；而如果要实现复杂的应用功能，那么可能需要在一个 PL/SQL 块中嵌套其他 PL/SQL 块。编写 PL/SQL 应用模块，块的嵌套层次没有限制。

### 3.1.1 PL/SQL 块结构

PL/SQL 块由三个部分组成：定义部分、执行部分、例外处理部分。其中，定义部分用于定义常量、变量、游标、例外、复杂数据类型等；执行部分用于实现应用模块功能，该部分包含了要执行的 PL/SQL 语句和 SQL 语句；例外处理部分用于处理执行部分可能出现的运行错误。PL/SQL 块的基本结构如下所示：

```
DECLARE
/*
 * 定义部分——定义常量、变量、复杂数据类型、游标、例解
*/
BEGIN
/*
 * 执行部分——PL/SQL 语句和 SQL 语句
*/
EXCEPTION
/*
 * 例外处理部分——处理运行错误
*/
END; /* 块结束标记 */
```

其中，定义部分以 DECLARE 开始，该部分是可选的；执行部分以 BEGIN 开始，该部分

是必须的；例外处理部分以 EXCEPTION 开始，该部分是可选的；而 END 则是 PL/SQL 块的结束标记。需要注意，DECLARE，BEGIN，EXCEPTION 后面没有分号（;），而 END 后则必须要带有分号（;）。下面举例说明各部分的作用。

### 示例一：只包含执行部分的 PL/SQL 块

当编写 PL/SQL 程序时，执行部分是必须的，而其他两个部分则是可选的。执行部分可以编写要执行的 PL/SQL 语句和 SQL 语句。示例如下：

```
set serveroutput on
BEGIN
    dbms_output.put_line('Hello,everyone!');
END;
/
Hello,everyone!
```

当执行该 PL/SQL 块时，会输出消息“Hello,everyone!”。其中 dbms\_output 是 Oracle 所提供的系统包，put\_line 是该包所包含的过程，用于输出字符串信息。关于 dbms\_output 包的详细介绍，请参见第 17 章的内容。注意，当使用 dbms\_output 包输出数据或消息时，必须要将 SQL\*Plus 的环境变量 serveroutput 设置为 on。

### 示例二：包含定义部分和执行部分的 PL/SQL 块

为了在 PL/SQL 块中使用变量、常量、例外和显式游标，开发人员必须要在定义部分定义变量、常量、例外和显式游标。示例如下：

```
DECLARE
    v_ename VARCHAR2(5);
BEGIN
    SELECT ename INTO v_ename FROM emp
    WHERE empno=&no;
    dbms_output.put_line('雇员名:'||v_ename);
END;
/
输入 no 的值: 7788
雇员名:SCOTT
```

当执行该 PL/SQL 块时，会根据输入的雇员号显示雇员名。为了临时存放雇员名，就必须要定义变量。在上例中，&no 为 SQL\*Plus 的替代变量，关于替代变量的作用请读者参见附录 B。

### 示例三：包含定义部分、执行部分和例外处理部分的 PL/SQL 块

为了避免 PL/SQL 程序的运行错误，提高 PL/SQL 程序的健壮性，应该合理地处理 PL/SQL 程序的运行错误。读者可能会注意到，当运行示例二时，如果输入了不存在的雇员号，则会提示“ORA-01403:未找到数据”错误。要避免该错误，可以在 PL/SQL 块中加入例外处理部分，示例如下：

```
DECLARE
    v_ename VARCHAR2(5);
BEGIN
    SELECT ename INTO v_ename FROM emp
    WHERE empno=&no;
    dbms_output.put_line('雇员名:'||v_ename);
```

```

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('请输入正确的雇员号！');
END;
/
输入 no 的值: 1234
请输入正确的雇员号!

```

如例所示，输入了不正确的雇员号，会提示你输入正确的雇员编号，从而避免了程序运行错误。其中 NO\_DATA\_FOUND 是 PL/SQL 的预定义例外。

### 3.1.2 PL/SQL 块分类

当使用 PL/SQL 开发应用模块时，根据需要实现的应用模块功能，可以将 PL/SQL 块划分为匿名块、命名块、子程序和触发器等四种类型。下面分别介绍这四种类型的 PL/SQL 块。

#### 1. 匿名块

匿名块是指没有名称的 PL/SQL 块，匿名块既可以内嵌到应用程序（例如 Pro\*C/C++）中，也可以在交互式环境（例如 SQL\*Plus）中直接使用。示例如下：

```

DECLARE
  v_avgsal NUMBER(6,2);
BEGIN
  SELECT avg(sal) INTO v_avgsal FROM emp
  WHERE deptno=&no;
  dbms_output.put_line('平均工资:'||v_avgsal);
END;
/
输入 no 的值: 10
平均工资:2916.67

```

如例所示，当在 SQL\*Plus 中执行以上 PL/SQL 块时，会根据输入的部门号显示部门平均工资。但因为该 PL/SQL 块直接以 DECLARE 开始，没有给出任何名称，所以该 PL/SQL 块属于匿名块。

#### 2. 命名块

命名块是指具有特定名称标识的 PL/SQL 块，命名块与匿名块非常类似，只不过在 PL/SQL 块前使用<>>加以标记。当使用嵌套块时，为了区分多级嵌套层次关系，可以使用命名块加以区分。示例如下：

```

<<outer>>
DECLARE
  v_deptno NUMBER(2);
  v_dname  VARCHAR2(10);
BEGIN
  <<inner>>
  BEGIN
    SELECT deptno INTO v_deptno FROM emp
    WHERE lower(ename)=lower('&name');
  END;--<<inner>>

```

```

SELECT dname INTO v_dname FROM dept
WHERE deptno=v_deptno;
dbms_output.put_line('部门名:'||v_dname);
END; -- <<outer>>
/
输入 name 的值: scott
部门名:RESEARCH

```

如例所示，<<outer>>和<<inner>>分别是主块（外层块）和子块（内层块）的标记，这种PL/SQL 块被称为命名块。

### 3. 子程序

子程序包括过程、函数和包。当开发 PL/SQL 子程序时，既可以开发客户端的子程序，也可以开发服务器端的子程序。客户端子程序主要用在 Developer 中，而服务器端子程序可以用在任何应用程序中。通过将商业逻辑和企业规则集成到 PL/SQL 子程序中，可以简化客户端程序的开发和维护，并且提高应用程序的性能。下面简单介绍一下各种子程序。

#### (1) 过程

过程用于执行特定操作。当建立过程时，既可以指定输入参数 (IN)，也可以指定输出参数 (OUT)。通过在过程中使用输入参数，可以将应用环境的数据传递到执行部分；通过使用输出参数，可以将执行部分的数据传递到应用环境。在 SQL\*Plus 中可以使用 CREATE PROCEDURE 命令建立过程。示例如下：

```

CREATE PROCEDURE update_sal(name VARCHAR2,newsal NUMBER)
IS
BEGIN
    UPDATE emp SET sal=newsal
    WHERE lower(ename)=lower(name);
END;
/

```

如例所示，过程 update\_sal 用于更新雇员工资。当在 SQL\*Plus 中调用该过程时，可以使用 execute 命令或 call 命令。示例如下：

```

示例一 SQL> exec update_sal('scott',2000)
示例二 SQL> call update_sal('scott',2000);

```

#### (2) 函数

函数用于返回特定数据。当建立函数时，在函数头部必须包含 RETURN 子句，而在函数体内必须要包含 RETURN 语句返回数据。在 SQL\*Plus 中可以使用 CREATE FUNCTION 命令建立函数，示例如下：

```

CREATE FUNCTION annual_income(name VARCHAR2)
RETURN NUMBER IS
    annual_salary NUMBER(7,2);
BEGIN
    SELECT sal*12+nvl(comm,0) INTO annual_salary
    FROM emp WHERE lower(ename)=lower(name);
    RETURN annual_salary;
END;
/

```

如上例所示，函数 `annual_income` 用于返回雇员的全年收入（包括工资和奖金）。当调用该函数时，可以使用多种方法。在这里将使用 SQL\*Plus 绑定变量存放输出结果，示例如下：

```
SQL> VAR income NUMBER
SQL> CALL annual_income('scott') INTO :income;
SQL> PRINT income
INCOME
-----
24000
```

### (3) 包

包用于逻辑组合相关的过程和函数，它由包规范和包体两部分组成。包规范用于定义公用的常量、变量、过程和函数，在 SQL\*Plus 中建立包规范可以使用 `CREATE PACKAGE` 命令。示例如下：

```
CREATE PACKAGE emp_pkg IS
    PROCEDURE update_sal(name VARCHAR2, newsal NUMBER);
    FUNCTION annual_income(name VARCHAR2) RETURN NUMBER;
END;
/
```

包规范只包含了过程和函数的说明，而没有过程和函数的实现代码。包体用于实现包规范中的过程和函数，在 SQL\*Plus 中建立包体可以使用 `CREATE PACKAGE BODY` 命令。示例如下：

```
CREATE PACKAGE BODY emp_pkg IS
    PROCEDURE update_sal(name VARCHAR2, newsal NUMBER)
    IS
    BEGIN
        UPDATE emp SET sal=newsal
        WHERE lower(ename)=lower(name);
    END;
    FUNCTION annual_income(name VARCHAR2) RETURN NUMBER
    IS
        annual_salary NUMBER(7,2);
    BEGIN
        SELECT sal*12+nvl(comm,0) INTO annual_salary
        FROM emp WHERE lower(ename)=lower(name);
        RETURN annual_salary;
    END;
END;
/
```

当调用包的过程和函数时，在过程和函数名之前必须要带有包名作为前缀（包名.子程序名），而如果要访问其他方案的包，还必须要加方案名作为前缀（方案名.包名.子程序名）。示例如下：

```
示例：SQL> call emp_pkg.update_sal('scott',1500);
示例：SQL> VAR income NUMBER
SQL> CALL emp_pkg.annual_income('scott') INTO :income;
```

```
SQL> PRINT income
INCOME
-----
18000
```

#### 4. 触发器

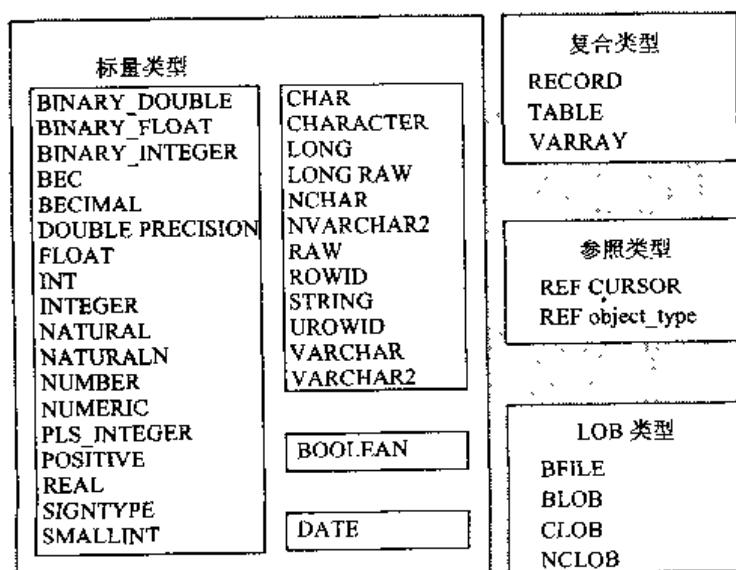
触发器是指隐含执行的存储过程。当定义触发器时，必须要指定触发事件以及触发操作，常用的触发事件包括 INSERT, UPDATE 和 DELETE 语句，而触发操作实际是一个 PL/SQL 块。在 SQL\*Plus 中建立触发器是使用 CREATE TRIGGER 命令来完成的，示例如下：

```
CREATE TRIGGER update_cascade
AFTER UPDATE OF deptno ON dept
FOR EACH ROW
BEGIN
  UPDATE emp SET deptno=:new.deptno
  WHERE deptno=:old.deptno;
END;
/
```

如上例所示，触发器 update\_cascade 用于实现级联更新，如果不建立该触发器，那么当更新 dept 表的 deptno 列数据时就会显示错误“ORA-02292：违反完整约束条件 (SCOTT.FK\_DEPTNO) - 已找到子记录日志”；而在建立了该触发器之后，当更新 deptno 列时，就会级联更新 emp 表的 deptno 列的相关数据。

## 3.2 定义并使用变量

编写 PL/SQL 程序时，若临时存储数值，必须要定义变量和常量；若在应用环境和子程序之间传递数据，必须要为子程序指定参数。而在 PL/SQL 程序中定义变量、常量和参数时，则必须要为它们指定 PL/SQL 数据类型。在编写 PL/SQL 程序时，可以使用标量（Scalar）类型、复合（Composite）类型、参照（Reference）类型和 LOB（Large Object）类型等四种类型。如下图所示：



图中列出了在编写 PL/SQL 程序时可以引用的各种数据类型，其中 BINARY\_DOUBLE 和 BINARY\_FLOAT 是 Oracle 10g 新增加的数据类型。另外，从 Oracle9i 开始，Oracle 还增加了一些日期时间类型，包括 TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL ZONE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH 等，这些数据类型都可以在 PL/SQL 块中引用。

### 3.2.1 标量变量

标量变量是指只能存放单个数值的变量。当编写 PL/SQL 程序时，最常用的变量就是标量变量。当定义标量变量时，必须要指定标量数据类型。标量数据类型包括数字类型、字符类型、日期类型和布尔类型，每种类型又包含有相应的子类型，例如 NUMBER 类型包含有 INTEGER, POSITIVE 等子类型。

#### 1. 常用标量类型

##### (1) VARCHAR2(n)

该数据类型用于定义可变长度的字符串，其中 n 用于指定字符串的最大长度，其最大值为 32767 字节。当使用该数据类型定义变量时，必须要指定长度。需要注意，当在 PL/SQL 块中使用该数据类型操纵 VARCHAR2 表列时，其数值的长度不应该超过 4000 字节。

##### (2) CHAR(n)

该数据类型用于定义固定长度的字符串，其中 n 用于指定字符串的最大长度，其最大值为 32767 字节。当使用该数据类型定义变量时，如果没有指定 n，则其默认值为 1。需要注意，当在 PL/SQL 块中使用该数据类型操纵 CHAR 表列时，其数值的长度不应该超过 2000 字节。

##### (3) NUMBER(p,s)

该数据类型用于定义固定长度的整数和浮点数，其中 p 表示精度，用于指定数字的总位数；s 表示标度，用于指定小数点后的数字位数。例如，假定定义了变量 NUMBER(6,2)，那么整数位数最大应该是几位呢？答案是 4 位。

##### (4) DATE

该数据类型用于定义日期和时间数据，其数据长度为固定长度（7 字节）。但需注意，当给 DATE 变量赋值时，数据必须要与日期格式和日期语言匹配。

##### (5) TIMESTAMP

该数据类型是 Oracle9i 新增加的数据类型，它也用于定义日期和时间数据。给 TIMESTAMP 变量赋值的方法与给 DATE 变量赋值的方法完全相同。但当显示 TIMESTAMP 变量数据时，不仅会显示日期，而且还会显示时间和上下午标记。

##### (6) LONG 和 LONG RAW

LONG 数据类型用于定义变长字符串，类似于 VARCHAR2 数据类型，但其字符串的最大长度为 32760 字节；LONG RAW 数据类型用于定义变长的二进制数据，其数据最大长度为 32760 字节。

##### (7) BOOLEAN

该数据类型用于定义布尔变量，其变量的值为 TRUE、FALSE 或 NULL。需要注意，该数据类型是 PL/SQL 数据类型，表列不能采用该数据类型。

##### (8) BINARY\_INTEGER

该数据类型用于定义整数，其数值范围在-2147483647 和 2147483647 之间。在 Oracle9i 之前，当在 PL/SQL 块中定义 PL/SQL 表时，必须使用该数据类型作为下标的数据类型。需要注意，该数据类型是 PL/SQL 数据类型，表列不能采用该数据类型。

### (9) BINARY\_FLOAT 和 BINARY\_DOUBLE

BINARY\_FLOAT 和 BINARY\_DOUBLE 是 Oracle 10g 新增加的数据类型，分别用于定义单精度的浮点数和双精度的浮点数。这两种数据类型主要用于高速的科学计算，当为 BINARY\_FLOAT 类型的变量赋值时，应该带有后缀 f（例如 1.5f）；当为 BINARY\_DOUBLE 类型的变量赋值时，应该带有后缀 d（例如 3.00095d）。

## 2. 定义标量变量

当编写 PL/SQL 程序时，如果要引用变量变量，必须首先在定义部分定义变量变量，然后才能在执行部分或例外处理部分中使用这些标量变量。

### (1) 语法

在 PL/SQL 块中定义变量和常量的语法如下：

```
identifier [CONSTANT] datatype [NOT NULL] {:= | DEFAULT expr]
```

- **identifier**: 用于指定变量或常量的名称。
- **CONSTANT**: 用于指定常量。当定义常量时，必须指定它的初始值，并且其数值不能改变。
- **datatype**: 用于指定变量或常量的数据类型。
- **NOT NULL**: 用于强制初始化变量（不能为 NULL）。当指定 NOT NULL 选项时，必须要为变量提供数值。
- **:=**: 用于为变量和常量指定初始值。
- **DEFAULT**: 用于为变量和常量指定初始值。
- **expr**: 用于指定初始值的 PL/SQL 表达式，可以是文本值、其他变量、函数等。

### (2) 定义标量变量示例

当定义标量变量时，必须要使用标量数据类型。示例如下：

```
v_ename      VARCHAR2(10);
v_sal       NUMBER(6,2);
v_balance   BINARY_FLOAT; --Oracle 10g 新数据类型
c_tax_rate  CONSTANT NUMBER(3,2):=5.5;
v_hiredate  DATE;
v_valid     BOOLEAN  NOT NULL DEFAULT FALSE;
```

如例所示，以上语句定义了五个变量（v\_ename, v\_sal, v\_balance, v\_hiredate, v\_valid）和一个常量（c\_tax\_rate），并且为变量 v\_valid 提供了初始值。需要注意，如果在定义变量时没有指定初始值，那么变量初始值为 NULL。

## 3. 使用标量变量

当在定义部分定义了标量变量之后，在执行部分和例外处理部分可以引用这些标量变量。需要注意，在 PL/SQL 块中为变量赋值不同于其他编程语言，必须要在等号前加冒号（:=）。下面以输入雇员号显示雇员姓名、工资、个人所得税为例，说明在 PL/SQL 块中使用标量变量的方法。示例如下：

```
DECLARE
```

```

v_ename VARCHAR2(5);
v_sal NUMBER(6,2);
c_tax_rate CONSTANT NUMBER(3,2):=0.03;
v_tax_sal NUMBER(6,2);

BEGIN
    SELECT ename,sal INTO v_ename,v_sal
    FROM emp WHERE empno=&eno;
    v_tax_sal:=v_sal*c_tax_rate;
    dbms_output.put_line('雇员名:'||v_ename);
    dbms_output.put_line('雇员工资:'||v_sal);
    dbms_output.put_line('所得税:'||v_tax_sal);
END;
/
输入 eno 的值: 7788
雇员名:SCOTT
雇员工资:1500
所得税:45

```

读者可能会注意到，在执行该 PL/SQL 块时，如果雇员名长度超过 5 字节，则会显示错误信息“ORA-06502: PL/SQL: 数字或值错误：字符串缓冲区太小”，原因是 `v_ename` 变量的最大长度为 5 字节。因为 `ename` 列的最大长度为 10 字节，所以只需要将 `v_ename` 变量的最大长度设置为 10 字节就可以了。在应用模块设计完成之后，假定将来需要输入名称超过 10 字节的雇员，那么就需要修改 `EMP` 表的 `ENAME` 列数据长度，很显然还需要修改应用模块。为了提高程序的可用性，降低 PL/SQL 程序的维护工作量，可以使用`%TYPE` 属性定义变量。

#### 4. 使用`%TYPE` 属性

当定义 PL/SQL 变量存放列值时，必须确保变量使用合适的数据类型和长度，否则在运行过程中可能会出现 PL/SQL 运行错误。为了避免这种不必要的错误，可以使用`%TYPE` 属性定义变量。当使用`%TYPE` 属性定义变量时，它会按照数据库列或其他变量来确定新变量的类型和长度。示例如下：

```

DECLARE
    v_ename emp.ename%TYPE;
    v_sal emp.sal%TYPE;
    c_tax_rate CONSTANT NUMBER(3,2):=0.03;
    v_tax_sal v_sal%TYPE;

BEGIN
    SELECT ename,sal INTO v_ename,v_sal
    FROM emp WHERE empno=&eno;
    v_tax_sal:=v_sal*c_tax_rate;
    dbms_output.put_line('雇员名:'||v_ename);
    dbms_output.put_line('雇员工资:'||v_sal);
    dbms_output.put_line('所得税:'||v_tax_sal);
END;
/
输入 eno 的值: 7788
雇员名:SCOTT

```

雇员工资:1500

所得税:45

如例所示，变量 v\_ename, v\_sal 与 EMP 表的 ename 列、sal 列的数据类型和长度完全一致，而变量 v\_tax\_sal 与变量 v\_sal 的数据类型和长度完全一致。这样，当 ename 列和 sal 列的类型和长度发生改变时，该 PL/SQL 块将不需要进行任何修改。

### 3.2.2 复合变量

复合变量是指用于存放多个值的变量。当定义复合变量时，必须要使用 PL/SQL 的复合数据类型。PL/SQL 包括 PL/SQL 记录、PL/SQL 表、嵌套表以及 VARRAY 等四种复合数据类型。

#### 1. PL/SQL 记录

PL/SQL 记录类似于高级语言中的结构，每个 PL/SQL 记录一般都包含多个成员。当使用 PL/SQL 记录时，首先需要在定义部分定义记录类型和记录变量，然后在执行部分引用该记录变量。需要注意，当引用记录成员时，必须要加记录变量作为前缀（记录变量.记录成员）。示例如下：

```

DECLARE
    TYPE emp_record_type IS RECORD (
        name    emp.ename%TYPE,
        salary  emp.sal%TYPE,
        title   emp.job%TYPE);
    emp_record emp_record_type;
BEGIN
    SELECT ename,sal,job INTO emp_record
    FROM emp WHERE empno=7788;
    dbms_output.put_line('雇员名:'||emp_record.name);
END;
/
雇员名:SCOTT

```

如例所示，emp\_record\_type 是 PL/SQL 记录类型，并且该 PL/SQL 记录类型包含了三个成员 (name, salary, title)；emp\_record 是记录变量；emp\_record.name 则表示引用记录变量 emp\_record 的成员 name。

#### 2. PL/SQL 表

PL/SQL 表类似于高级语言中的数组。需要注意，PL/SQL 表与高级语言的数组有所区别，高级语言数组的下标不能为负，但 PL/SQL 表的下标可以为负值；高级语言数组的元素个数有限制，而 PL/SQL 表的元素个数没有限制，并且其下标没有上下限。当使用 PL/SQL 表时，必须首先在定义部分定义 PL/SQL 表类型和 PL/SQL 表变量，然后在执行部分中引用该 PL/SQL 表变量。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
BEGIN
    SELECT ename INTO ename_table(-1) FROM emp

```

```

    WHERE empno=7788;
    dbms_output.put_line('雇员名:'||ename_table(-1));
END;
/
雇员名:SCOTT

```

如例所示, `ename_table_type` 为 PL/SQL 表类型; `emp.ename%TYPE` 指定了 PL/SQL 表元素的数据类型和长度; `ename_table` 为 PL/SQL 表变量, `ename_table(-1)` 则表示下标为-1 的元素。

### 3. 嵌套表

嵌套表 (Nested Table) 类似于高级语言中的数组。需要注意, 高级语言数组和嵌套表的下标都不能为负值; 高级语言的元素个数是有限制的, 而嵌套表的元素个数是没有限制的。嵌套表和 PL/SQL 表非常类似, 但嵌套表可以作为表列的数据类型, 而 PL/SQL 表不能作为表列的数据类型。当在表列中使用嵌套表时, 必须首先使用 CREATE TYPE 语句建立嵌套表类型。示例如下:

```

CREATE OR REPLACE TYPE emp_type AS OBJECT(
    name VARCHAR2(10), salary NUMBER(6,2),
    hiredate DATE);
/
CREATE OR REPLACE TYPE emp_array IS TABLE OF emp_type;
/

```

如例所示, 对象类型 `emp_type` 用于存储雇员信息, 而 `emp_array` 是基于 `emp_type` 的嵌套表类型, 它可以用于存储多个雇员的信息。当建立了嵌套表类型之后, 就可以在表列或对象属性中将其作为用户自定义数据类型来引用。但需要注意, 当使用嵌套表类型作为表列时, 必须要为其指定专门的存储表。示例如下:

```

CREATE TABLE department(
    deptno NUMBER(2), dname VARCHAR2(10),
    employee emp_array
) NESTED TABLE employee STORE AS employee;

```

### 4. VARRAY

VARRAY (变长数组) 类似于嵌套表, 它可以作为表列和对象类型属性的数据类型。但需要注意, 嵌套表的元素个数没有限制, 而 VARRAY 的元素个数是有限制的。当使用 VARRAY 时, 必须首先建立 VARRAY 类型。示例如下:

```

CREATE TYPE article_type AS OBJECT (
    title VARCHAR2(30), pubdate DATE
);
/
CREATE TYPE article_array IS VARRAY(20) OF article_type;
/

```

如例所示, 对象类型 `article_type` 用于存储文章信息, 而 `article_array` 则用于存储多篇文章的信息, 并且最多可以存储 20 篇文章。当建立了 VARRAY 类型之后, 可以在表列或对象属性中将其作为用户自定义数据类型来引用。示例如下:

```

CREATE TABLE author(
    id NUMBER(6), name VARCHAR2(10), article article_array
);

```

注意，嵌套表列数据需要存储在专门的存储表中，而 VARRAY 数据则与其他列数据一起存放在表段中。

### 3.2.3 参照变量

参照变量是指用于存放数值指针的变量。通过使用参照变量，可以使得应用程序共享相同对象，从而降低占用空间。在编写 PL/SQL 程序时，可以使用游标变量（REF CURSOR）和对象类型变量 REF obj\_type 等两种参照变量类型。

#### 1. REF CURSOR

当使用显式游标时，需要在定义显式游标时指定相应的 SELECT 语句，这种显式游标称为静态游标。当使用游标变量时，在定义游标变量时不需要指定 SELECT 语句，而是在打开游标时指定 SELECT 语句，从而实现动态的游标操作。示例如下：

```

DECLARE
    TYPE c1 IS REF CURSOR;
    emp_cursor c1;
    v_ename emp.ename%TYPE;
    v_sal emp.sal%TYPE;
BEGIN
    OPEN emp_cursor FOR
        SELECT ename,sal FROM emp WHERE deptno=10;
    LOOP
        FETCH emp_cursor INTO v_ename,v_sal;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line(v_ename);
    END LOOP;
    CLOSE emp_cursor;
END;
/
CLARK
KING
MILLER

```

如例所示，c1 为 REF CURSOR 类型，而 emp\_cursor 为游标变量，并且在打开游标变量时指定了其所对应的 SELECT 语句。

#### 2. REF obj\_type

当编写对象类型应用时，为了共享相同对象，可以使用 REF 引用对象类型，REF 实际是指向对象实例的指针。下面通过示例说明如何使用 REF。首先建立对象类型 home 和对象表 homes，然后插入数据。示例如下：

```

CREATE OR REPLACE TYPE home_type AS OBJECT(
    street VARCHAR2(50),city VARCHAR2(20),
    state VARCHAR2(20),zipcode VARCHAR2(6),
    owner VARCHAR2(10)
);
/
CREATE TABLE homes OF home_type;

```

```

INSERT INTO homes VALUES ('呼伦北路 12 号', '呼和浩特',
    '内蒙', '010010', '马鸣');
INSERT INTO homes VALUES ('呼伦北路 13 号', '呼和浩特',
    '内蒙', '010010', '秦斌');
COMMIT;

```

如例所示，对象表 homes 存放着家庭所在地以及户主姓名。假定每个家庭有四口人，当进行人口统计时，为了使得同一家庭的每个家庭成员可以共享家庭地址，可以使用 REF 引用 home\_type 对象类型，从而降低占用空间。示例如下：

```

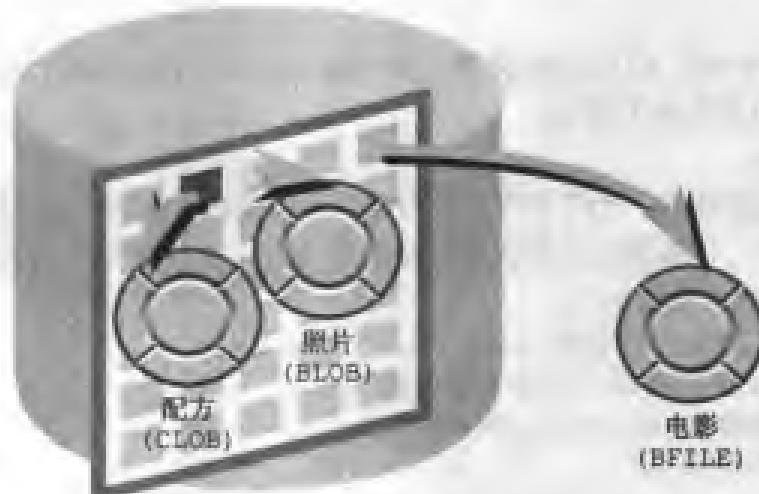
CREATE TABLE person (
    id NUMBER(6) PRIMARY KEY,
    name VARCHAR2(10), addr REF home_type
);
INSERT INTO person SELECT 1, '马鸣', ref(p)
FROM homes p WHERE p.owner='马鸣';
INSERT INTO person SELECT 2, '马武', ref(p)
FROM homes p WHERE p.owner='马鸣';
INSERT INTO person SELECT 3, '王敏', ref(p)
FROM homes p WHERE p.owner='马鸣';
COMMIT;

```

如例所示，当为 person 表插入数据时，addr 列将会存放指向 homes 表相应数据的地址指针。

### 3.2.4 LOB 变量

LOB 变量是指用于存储大批量数据的变量。Oracle 将 LOB 分为两种：内部 LOB 和外部 LOB。内部 LOB 包括 CLOB、BLOB 和 NCLOB 三种类型，它们的数据被存储在数据库中，并且支持事务操作（提交、回退、保存点）。外部 LOB 只有 BFILE 一种类型，该类型的数据被存储在 OS 文件中，并且不支持事务操作。其中，CLOB/NCLOB 用于存储大批量字符数据，BLOB 用于存储大批量二进制数据，而 BFILE 则存储指向 OS 文件的指针。如下图所示：



因为医药的配方需要用大量中文说明，所以可以将其所对应的数据库列定义为 CLOB 类

型；因为照片对应于图形/图象（二进制数据），所以可以将其所对应的数据库列定义为 BLOB 类型；为了通过数据库访问 OS 电影文件，可以在数据库中定义 BFILE 类型的数据库列。

### 3.2.5 非 PL/SQL 变量

当在 SQL\*Plus 或应用程序（例如 Pro\*C/C++）中与 PL/SQL 块之间进行数据交互时，需要使用 SQL\*Plus 变量或应用程序变量来完成。当在 PL/SQL 块中引用非 PL/SQL 变量时，必须要在非 PL/SQL 变量前加冒号（“:”）。

#### 1. 使用 SQL\*Plus 变量

在 PL/SQL 块中引用 SQL\*Plus 变量时，必须首先使用 VARIABLE 命令定义变量；而如果要在 SQL\*Plus 中输出变量内容，则需要使用 PRINT 命令。示例如下：

```
var name varchar2(10)
BEGIN
    SELECT ename INTO :name FROM emp
    WHERE empno=7788;
END;
/
PRINT name
NAME
-----
SCOTT
```

#### 2. 使用 Procedure Builder 变量

当在 PL/SQL 块中引用 Procedure Builder 变量时，必须首先使用 CREATE 命令定义变量，而如果在 Procedure Builder 中输出变量内容，则可以使用包 TEXT\_IO。示例如下：

```
PL/SQL> .CREATE CHAR name LENGTH 10
PL/SQL> BEGIN
    +> SELECT ename INTO :name FROM emp
    +> WHERE empno=7788;
    +> END;
PL/SQL> TEXT_IO.PUT_LINE(:name);
SCOTT
```

#### 3. 使用 Pro\*C/C++变量

当在 PL/SQL 块中引用 Pro\*C/C++程序的宿主变量时，必须首先定义宿主变量，而如果要输出变量内容，则可以使用 printf()语句。示例如下：

```
char name[10];
EXEC SQL EXECUTE
BEGIN
    SELECT ename INTO :name FROM emp
    WHERE empno=7788;
END;
END-EXEC;
printf("雇员名:%s\n",name);
```

### 3.3 编写 PL/SQL 代码

在编写 PL/SQL 应用程序时，为了开发正确、高效的 PL/SQL 块，必须要遵从 PL/SQL 代码编写的规则，否则会导致编译错误或运行错误。而每行 PL/SQL 代码又是由特定的 PL/SQL 词汇（Lexical）单元组成的，所以 PL/SQL 应用开发人员除了要遵从代码编写规则之外，还必须要掌握 PL/SQL 各种词汇单元的作用。

#### 3.3.1 PL/SQL 词汇单元

当编写 PL/SQL 块时，每个 PL/SQL 块都包含多行代码，而每行代码又是由多个合法单元组成的，这些合法单元被称为词汇。请看如下示例：

```
DECLARE
    v_sal NUMBER(6,2);
BEGIN
    v_sal:=1000;
END;
/
```

如上所示，第四行 `v_sal`、`:=`、`1000` 是三个词汇单元，它们之间可以用空格隔开，例如语句“`v_sal := 1000`”也是合法的。PL/SQL 有分隔符（Delimiter）、标识符（Identifier）、文字串（Literal）和注释（Comment）等四种词汇单元。

##### 1. 分隔符

分隔符是指具有特定含义的单个符号或组合符号。例如，用户可以使用单符号分隔符执行算术运算（`+`，`-`，`*`、`/`），可以使用组合分隔符执行赋值操作（`:=`）和比较操作（`>=`）等。

###### (1) 单符号分隔符

单符号分隔符是指只包含单个符号的 PL/SQL 分隔符，下表列出了 PL/SQL 所有的单符号分隔符：

符号	含义
<code>+</code>	加法操作符
<code>%</code>	属性提示符
<code>,</code>	字符串分隔符
<code>.</code>	组件分隔符
<code>/</code>	除法操作符
<code>(</code>	表达式或列表分隔符
<code>)</code>	表达式或列表分隔符
<code>:</code>	非 PL/SQL 变量提示符
<code>,</code>	项分隔符（表名、列名等分隔符）
<code>*</code>	乘法操作符
<code>"</code>	双引号变量分隔符

续表

符号	含义
=	相等操作符
<	小于操作符
>	大于操作符
@	远程数据库访问操作符
;	语句终止符
-	减法操作符或负数操作符

## (2) 组合分隔符

组合分隔符是指由多个符号组成的 PL/SQL 分隔符，下表列出了 PL/SQL 中所有的组合分隔符：

符号	含义
:=	赋值操作符
→	关联操作符
	连接操作符
**	幂操作符
<<	标号开始分隔符
>>	标号结束分隔符
/*	多行注释开始分隔符
*/	多行注释结束分隔符
..	范围操作符
<>	不等操作符
!=	不等操作符
^=	不等操作符
<=	小于等于操作符
>=	大于等于操作符
--	单行注释提示符

## 2. 标识符

标识符用于指定 PL/SQL 程序单元和程序项的名称。通过使用标识符，可以定义常量、变量、例外、显式游标、游标变量、参数、子程序以及包的名称。当使用标识符定义 PL/SQL 程序项或程序单元时，必须要满足以下规则：

- 当使用标识符定义变量、常量时，每行只能定义一个标识符。
- 当使用标识符定义变量、常量时，标识符名称必须要以阿拉伯字符（A~Z, a~z）开始，并且最大长度为 30 个字符。如果以其他字符开始，那么必须要使用双引号引住。
- 当使用标识符定义变量、常量时，标识符名称只能使用符号 A~Z, a~z, 0~9, \_, \$和#。如果要使用其他字符，那么必须要使用双引号引住。

- 当使用标识符定义变量、常量时，标识符名称不能使用 Oracle 的关键字。例如不能使用 SELECT, UPDATE 等作为变量名。如果要使用 Oracle 关键字定义变量、常量，那么必须要使用双引号引住。

### 示例一：合法的标识符定义

```
v_ename      VARCHAR2(10);
v$sal        NUMBER(6,2);  v#sal  NUMBER(6,2);
v#error      EXCEPTION;
"1234"       VARCHAR2(20); ----以数字开始，带有双引号
"变量 A"     NUMBER(10,2); ----包含汉字，带有双引号
```

### 示例二：非法的标识符定义

```
v%ename      VARCHAR2(10);   — 非法符号 (%)
2sal         NUMBER(6,2);    — 以数字开始非法
#v1          EXCEPTION;      — 以#号开始非法
v1,v2        VARCHAR2(20);   — 每行只能定义一个变量
变量 A       NUMBER(10,2);   — 不能以汉字开始
select        NUMBER(10,2);   — 不能使用关键字作为变量名
```

## 3. 文本

文本是指数字、字符、字符串、日期值或布尔值，而不是标识符。文本包括数字文本、字符文本、字符串文本、布尔文本、日期时间文本。

### (1) 数字文本

数字文本是指整数或浮点数，它们可以直接在算术表达式中引用。当编写 PL/SQL 代码时，用户可以使用科学计数法和幂操作符 (\*\*）。注意，科学计数法和幂操作符只适用于 PL/SQL 语句，而不适用于 SQL 语句。数字文本的示例如下：

100    2.45    3e3    5E6    6\*10\*\*3

### (2) 字符文本

字符文本是指用单引号引住的单个字符，这些字符可以是 PL/SQL 支持的所有可打印字符，包括阿拉伯字符（A~Z, a~z）、数字字符（0~9）以及其他符号（<, >等）。字符文本的示例如下：

'A'    '9'    '<'    ''    '%'

### (3) 字符串文本

字符串文本是指由两个或两个以上字符组成的字符值。当指定字符串文本时，必须要用单引号将字符串文本引住。字符串文本的示例如下：

'Hello World'    '\$9600'    '10-NOV-91'

在 Oracle 10g 之前，如果字符串文本包含单引号，那么必须使用两个单引号表示。例如，如果要为某个变量赋值 “I'm a string,you're a string.”，那么字符串文本必须要采用以下格式：

string\_var:='I''m a string, you''re a string.';

在 Oracle 10g 之中，如果字符串文本包含单引号，那么既可以使用原有格式赋值，也可以使用其他分隔符（[]、{}、<>等）赋值。注意，如果要使用分隔符[]、{}、<>为字符串赋值，那么不仅需要在分隔符前后加单引号，而且需要带有前缀 q。示例如下：

string\_var:=q'[I'm a string, you're a string.]';

### (4) 布尔文本

布尔文本是指 BOOLEAN 值，它主要用在条件表达式中，布尔文本有三种值：TRUE、FALSE 和 NULL。

### (5) 日期时间文本

日期时间文本是指日期时间值。与字符串类似，日期文本也必须要用单引号引住，并且日期值必须要与日期格式和日期语言匹配。示例如下：

```
'10-NOV-91'    '1997-10-22 13:01:01'    '09-10月-03'
```

## 4. 注释

注释用于解释单行代码或多行代码的作用，从而提高了 PL/SQL 程序的可读性。当编译并执行 PL/SQL 代码时，PL/SQL 编译器会忽略注释。注释又包括单行注释和多行注释。

### (1) 单行注释

单行注释是指放置在一行上的注释文本，并且单行注释主要用于说明单行代码的作用。在 PL/SQL 中使用--符号编写单行注释，示例如下：

```
SELECT sal INTO salary FROM emp -- 取得雇员工资  
WHERE empno = emp_id;
```

### (2) 多行注释

多行注释是指分布到多行上的注释文本，并且其主要作用是说明一段代码的作用。在 PL/SQL 中使用/\* ... \*/来编写多行注释，示例如下：

```
DECLARE  
    v_sal emp.sal%TYPE;  
BEGIN  
    /*  
     * 以下代码用于取得雇员工资  
     */  
    SELECT sal INTO v_sal FROM emp  
    WHERE empno=&no;  
    dbms_output.put_line(v_sal);  
END;  
/  
输入 no 的值： 7788  
3000
```

## 3.3.2 PL/SQL 代码编写规则

为了编写正确、高效的 PL/SQL 块，PL/SQL 应用开发人员必须遵从特定的 PL/SQL 代码编写规则，否则会导致编译错误或运行错误。在编写 PL/SQL 代码时，应该遵从以下一些规则：

### 1. 标识符命名规则

当在 PL/SQL 中使用标识符定义变量、常量时，标识符名称必须以字母开始，并且长度不能超过 30 个字符。另外，为了提高程序的可读性，Oracle 建议用户按照以下规则定义各种标识符：

- 当定义变量时，建议使用 v\_ 作为前缀，例如 v\_sal, v\_job 等。
- 当定义常量时，建议使用 c\_ 作为前缀，例如 c\_rate。
- 当定义游标时，建议使用 \_cursor 作为后缀，例如 emp\_cursor。
- 当定义例外时，建议使用 e\_ 作为前缀，例如 e\_integrity\_error。
- 当定义 PL/SQL 表类型时，建议使用 \_table\_type 作为后缀，例如 sal\_table\_type。

- 当定义 PL/SQL 表变量时，建议使用\_table 作为后缀，例如 sal\_table。
- 当定义 PL/SQL 记录类型时，建议使用\_record\_type 作为后缀，例如 emp\_record\_type。
- 当定义 PL/SQL 记录变量时，建议使用\_record 作为后缀，例如 emp\_record。

## 2. 大小写规则

当在 PL/SQL 块中编写 SQL 语句和 PL/SQL 语句时，语句既可以使用大写格式，也可以使用小写格式。但是，为了提高程序的可读性和性能，Oracle 建议用户按照以下大小写规则编写代码：

- SQL 关键字采用大写格式，例如 SELECT, UPDATE, SET, WHERE 等。
- PL/SQL 关键字采用大写格式，例如 DECLARE, BEGIN, END 等。
- 数据类型采用大写格式，例如 INT, VARCHAR2, DATE 等。
- 标识符和参数采用小写格式，例如 v\_sal, c\_rate 等。
- 数据库对象和列采用小写格式，例如 emp, sal, ename 等。

## 3. 代码缩进

类似于其他编程语言，当编写 PL/SQL 块时，为了提高程序可读性，同级代码应该对齐，而下一级代码则应该缩进。示例如下：

```

DECLARE
    v_detpno      NUMBER(2);
    v_location    VARCHAR2(13);
BEGIN
    SELECT deptno,location
    INTO v_deptno,v_location
    FROM dept
    WHERE dname = 'SALES';
    ...
END;

```

```

BEGIN
    IF x=0 then
        y=1;
    END IF;
END;

```

## 4. 嵌套块和变量范围

嵌套块是指嵌入在一个 PL/SQL 块中的另一个 PL/SQL 块，其中被嵌入的块被称为子块，而包含子块的 PL/SQL 块则称为主块。当使用嵌套块时，注意，子块可以引用主块所定义的任何标识符，但主块却不能引用子块的任何标识符。也就是说，子块定义的标识符是局部标识符（局部变量），而主块定义的标识符是全局标识符（全局变量）。如下图所示：

```

x BINARY_INTEGER;
BEGIN
    ...
    DECLARE
        y NUMBER;
    BEGIN
        ...
        END;
    ...
END;

```

### 5. 在 PL/SQL 块中可以使用的 SQL 函数

当编写 PL/SQL 代码时，可以直接引用大多数的单行 SQL 函数。这些 SQL 函数包括单行数字函数（例如 ROUND）、单行字符函数（例如 UPPER）、转换函数（例如 TO\_CHAR）以及各种日期函数（例如 MONTHS\_BETWEEN）。但是，用户需要注意，某些 SQL 函数只能在 SQL 语句中引用，而不能直接在 PL/SQL 语句中引用，这些 SQL 函数包括 GREATEST, LEAST, DECODE 以及所有分组函数（例如 SUM）。

## 3.4 习题

1. 以下哪种 PL/SQL 块用于返回数据？

- A. 匿名块
- B. 命名块
- C. 过程
- D. 函数
- E. 触发器

2. 以下定义变量和常量的方法中，哪些是正确的？

- A. v\_ename VARCHAR2(10);
- B. v\_sal,v\_comm NUMBER(6,2);
- C. v\_sal NUMBER(6,2) NOT NULL;
- D. c\_tax CONSTANT NUMBER(6,2) DEFAULT 0.17;
- E. %sal NUMBER(6,2);
- F. v\_comm emp.comm%TYPE;

3. 请根据结果确定以下变量的数据类型

- (1) v1:=100;
- (2) v2:=null;
- (3) v3:=(v\_comm IS NULL);
- (4) v4:=SYSDATE;
- (5) v5:='hello world';

4. 请首先阅读以下嵌套块内容：

```
DECLARE
  v1 NUMBER(6);
  v2 NUMBER(6);
BEGIN
  v1:=100;
  v2:=200;
  DECLARE
    v1 NUMBER(6);
    v3 NUMBER(6);
  BEGIN
    v1:=110;
    v2:=210;
    v3:=300;
  END;
  v3:=400;
```

```
END;
```

请确定在执行了以上 PL/SQL 块之后，子块和主块的变量结果：

- 子块中 v1, v2, v3 的值；
  - 主块中 v1, v2, v3 的值。
5. 在 SQL\*Plus 中编写一个简单的 PL/SQL 块，并显示输出信息“黄河是中国的母亲河”。
  6. 在 SQL\*Plus 中编写一个简单的 PL/SQL 块，使用 SQL\*Plus 替代变量，输入两个数字值，在 PL/SQL 块中定义变量存储两数相除结果，并显示结果。
  7. 在 SQL\*Plus 中使用 VARIABLE 命令定义字符串类型的绑定变量，并编写 PL/SQL 块为该绑定变量赋值“中国”，然后使用 PRINT 命令显示其结果。

## 第4章 使用SQL语句

SQL是关系数据库的基本操作语言，它是应用程序与数据库进行交互操作的接口，SQL语言包括数据查询语言（SELECT）、数据操纵语言（INSERT, UPDATE, DELETE）、事务控制语言（COMMIT, ROLLBACK, SAVEPOINT）、数据定义语言（CREATE, ALTER, DROP）、数据控制语言（GRANT、REVOKE）等五个部分。当编写PL/SQL应用程序时，只能直接嵌入SELECT语句、DML语句和事务控制语句。本章将详细介绍SELECT语句、DML语句、事务控制语句的作用，以及编写这些SQL语句的方法。在学习了本章之后，读者应该学会使用：

- SELECT语句去完成基本查询功能；
- INSERT, UPDATE 和 DELETE语句去操纵数据库数据；
- COMMIT, ROLLBACK 和 SAVEPOINT语句去控制事务；
- SELECT语句去实现各种复杂查询功能（数据分组、连接查询、子查询、层次查询、合并查询等）。

### 4.1 使用基本查询

SELECT语句用于检索数据库的数据，它是所有SQL语句中语法最复杂、功能也最强大的SQL语句。本小节只介绍使用SELECT语句去实现基本查询操作的方法，包括执行简单查询，以及WHERE子句和ORDER BY子句的使用方法。

#### 4.1.1 简单查询语句

在实际应用中为了取得数据信息，用户经常需要从数据库中查询数据。例如，通过查询部门表（DEPT），可以检索企业的所有部门信息；通过查询雇员表（EMP），可以检索企业的所有雇员信息。在关系数据库中，查询数据是使用SELECT语句来完成的。其基本语法如下：

```
SELECT <*, column [alias], ...> FROM table;
```

如上所示，SELECT关键字用于指定要检索的列；其中星号(\*)表示检索所有列，column用于指定要检索的列或表达式；alias用于指定列或表达式的别名；FROM关键字用于指定要检索的表。下面以检索DEPT和EMP表的数据为例，说明使用简单查询语句的方法。

##### 1. 确定表结构

当检索表数据时，既可以检索所有列的数据，也可以检索特定列的数据。但如果要检索特定表列的数据，必须要清楚表的结构。通过使用SQL\*Plus的DESCRIBE命令（可以简写为DESC），可以显示表结构。下面以显示DEPT和EMP表的结构为例，说明使用DESCRIBE命令的方法。查看DEPT表结构的示例如下：

```
SQL> DESC DEPT
      名称          是否为空? 类型
```

```
-----  
DEPTNO          NOT NULL NUMBER(2)  
DNAME           VARCHAR2(14)  
LOC              VARCHAR2(13)
```

如上所示，DEPTNO 列用于标识部门号，DNAME 列用于标识部门名称，LOC 列用于标识部门所在位置。查看 EMP 表结构的示例如下：

```
SQL> DESC EMP  
名称          是否为空? 类型  
-----  
EMPNO         NOT NULL NUMBER(4)  
ENAME          VARCHAR2(10)  
JOB            VARCHAR2(9)  
MGR            NUMBER(4)  
HIREDATE       DATE  
SAL             NUMBER(7,2)  
COMM            NUMBER(7,2)  
DEPTNO         NUMBER(2)
```

如上所示，EMPNO 用于标识雇员编号；ENAME 用于标识雇员名称；JOB 用于标识雇员岗位；MGR 用于标识雇员的管理者编号；HIREDATE 用于标识雇员的雇佣日期；SAL 用于标识雇员的工资；COMM 用于标识雇员的补助；DEPTNO 用于标识雇员所在的部门号。

## 2. 检索所有列

为了检索表的所有列数据，可以在 SELECT 关键字后指定星号 (\*)。下面以检索 DEPT 表的所有行、所有列数据为例，说明查询所有列的方法。示例如下：

```
SQL> SELECT * FROM DEPT;  
DEPTNO DNAME      LOC  
-----  
10 ACCOUNTING    NEW YORK  
20 RESEARCH      DALLAS  
30 SALES         CHICAGO  
40 OPERATIONS    BOSTON
```

## 3. 检索特定列

当检索表的特定列时，可以在 SELECT 关键字后指定列名。如果要检索多个列的数据，那么列之间使用逗号 (,) 隔开。下面以检索所有雇员的姓名、岗位、工资以及所在部门号为例，说明查询特定列的方法。示例如下：

```
SQL> set pagesize 30  
SQL> SELECT ename,sal,job,deptno FROM emp;  
ENAME      SAL JOB      DEPTNO  
-----  
SMITH     800 CLERK    20  
ALLEN    1600 SALESMAN 30  
WARD     1250 SALESMAN 30  
JONES    2975 MANAGER  20  
MARTIN   1250 SALESMAN 30  
BLAKE    2850 MANAGER  30
```

```

CLARK      2450 MANAGER      10
SCOTT      3000 ANALYST      20
KING       5000 PRESIDENT    10
TURNER     1500 SALESMAN     30
ADAMS      1100 CLERK        20
JAMES      950 CLERK         30
FORD       3000 ANALYST     20
MILLER     1300 CLERK        10

```

已选择 14 行

#### 4. 检索日期列

检索日期列与检索其他列没有任何区别。注意，日期数据的默认显示格式为“DD-MON-YY”，如果希望使用其他显示格式（YYYY-MM-DD），那么必须使用 TO\_CHAR 函数进行转换。另外，不同语言、地区的日期显示结果会有所不同。例如，如果语言为“SIMPLIFIED CHINESE”，则月名显示为中文格式（例如 5 月）；如果语言为“AMERICAN”，则月名显示为英文简写格式（例如 MAY）。下面以示例说明默认的日期显示格式，以及改变日期显示格式的方法。

##### 示例一：使用默认日期显示格式显示雇员雇佣日期

```

SQL> SELECT ename,hiredate FROM emp;
ENAME      HIREDATE
-----
SMITH      17-12月-80
ALLEN      20-2月 -81
WARD       22-2月 -81
JONES      02-4月 -81
MARTIN     28-9月 -81
BLAKE      01-5月 -81
CLARK      09-6月 -81
SCOTT      03-12月-81
KING       17-11月-81
TURNER     08-9月 -81
ADAMS      03-12月-81
JAMES      03-12月-81
FORD       03-12月-81
MILLER     23-1月 -82

```

已选择 14 行

##### 示例二：使用 YYYY-MM-DD 显示格式显示雇员雇佣日期

```

SQL> SELECT ename,TO_CHAR(hiredate,'YYYY-MM-DD') FROM emp;
ENAME      TO_CHAR(HI
-----
SMITH      1980-12-17
ALLEN     1981-02-20
WARD      1981-02-22
JONES      1981-04-02
MARTIN     1981-09-28
BLAKE      1981-05-01

```

```

CLARK      1981-06-09
SCOTT      1981-12-03
KING       1981-11-17
TURNER     1981-09-08
ADAMS      1981-12-03
JAMES      1981-12-03
FORD       1981-12-03
MILLER     1982-01-23

```

已选择 14 行

### 5. 取消重复行

当执行查询操作时，某些情况下可能会显示完全相同的数据结果，而完全相同的显示结果可能没有任何实际意义。因此在实际应用环境中，可能需要取消完全重复的显示结果，完成这项任务可以使用 DISTINCT 关键字。下面通过示例说明不使用 DISTINCT 关键字和使用 DISTINCT 关键字的区别。

#### 示例一：显示所有部门号及岗位（保留重复行）

```
SQL> SELECT deptno, job FROM emp;
DEPTNO   JOB
```

```

----- -----
20 CLERK
30 SALESMAN
30 SALESMAN
20 MANAGER
30 SALESMAN
30 MANAGER
10 MANAGER
20 ANALYST
10 PRESIDENT
30 SALESMAN
20 CLERK
30 CLERK
20 ANALYST
10 CLERK

```

已选择 14 行

#### 示例二：显示所有部门号及岗位（取消重复行）

```
SQL> SELECT DISTINCT deptno, job FROM emp;
DEPTNO   JOB
```

```

----- -----
10 CLERK
10 MANAGER
10 PRESIDENT
20 ANALYST
20 CLERK
20 MANAGER
30 CLERK
30 MANAGER

```

```
30 SALESMAN  
已选择 9 行
```

## 6. 使用算术表达式

当执行查询操作时，可以在数字列上使用算术表达式（+，-，×，÷），其中乘、除的优先级要高于加、减。如果要改变优先级，那么可以使用括号。下面以显示雇员名及全年工资为例，说明在 SELECT 语句中使用算术表达式的方法。示例如下：

```
SQL> SELECT ename,sal*12 FROM emp;  
ENAME          SAL*12  
-----  
SMITH          9600  
ALLEN          19200  
WARD           15000  
JONES          35700  
MARTIN         15000  
BLAKE          34200  
CLARK          29400  
SCOTT          36000  
KING           60000  
TURNER         18000  
ADAMS          13200  
JAMES          11400  
FORD           36000  
MILLER         15600
```

已选择 14 行。

## 7. 使用列别名

当在 SQL\*Plus 中执行查询操作时，首先会显示列标题，然后才会显示数据。默认情况下，列标题是大写格式的列名或表达式。通过使用列别名，可以改变列标题的显示样式。如果要使用列别名，那么列别名应在列或者表达式之后，在二者之间可以加 AS 关键字。注意，如果列别名有大小写之分，并包含特殊字符或空格，那么这样的别名必须用双引号引住。示例如下：

```
SQL> SELECT ename AS "姓名",sal*12 AS "年收入" FROM emp;  
姓名      年收入  
-----  
SMITH      9600  
ALLEN      19200  
WARD       15000  
JONES      35700  
MARTIN     15000  
BLAKE      34200  
CLARK      29400  
SCOTT      36000  
KING       60000  
TURNER     18000  
ADAMS      13200
```

```

JAMES      11400
FORD       36000
MILLER     15600
已选择 14 行

```

## 8. 处理 NULL

NULL 表示未知值，它既不是空格也不是 0。当给表插入数据时，如果没有给某列提供数据，并且该列没有默认值，那么其数据为 NULL。注意，当算术表达式包含 NULL 时，其结果也是 NULL。示例如下：

```

SQL> SELECT ename,sal,comm,sal+comm FROM emp;
ENAME      SAL      COMM      SAL+COMM
-----
SMITH     800
ALLEN    1600     300     1900
WARD      1250     500     1750
JONES     2975
MARTIN    1250    1400     2650
BLAKE     2850
CLARK     2450
SCOTT     3000     50     3050
KING      5000
TURNER    1500      0     1500
ADAMS     1100
JAMES     950
FORD      3000
MILLER    1300
已选择 14 行

```

如上所示，当检索雇员实际工资（SAL+COMM）时，如果 COMM 不是 NULL，则会显示实发工资值（SAL+COMM）；如果 COMM 为 NULL，则因结果为 NULL，所以不会显示任何结果。为了正确地显示雇员实发工资，就必须要处理 NULL。在 Oracle9i 之前，处理 NULL 只能使用 NVL 函数；从 Oracle9i 开始，还可以使用 NVL2 函数处理 NULL 值。下面通过示例说明使用这两个函数处理 NULL 的方法。

### 示例一：使用 NVL 函数处理 NULL 值

NVL 函数用于将 NULL 转变为实际值，其语法格式为 NVL(expr1, expr2)。如果 expr1 是 null，则返回 expr2；如果 expr1 不是 null，则返回 expr1。参数 expr1 和 expr2 可以是任意数据类型，但二者的数据类型必须要匹配。使用 NVL 函数处理 NULL 的示例如下：

```
SQL> SELECT ename,sal,comm,sal+nvl(comm,0) AS "月收入" FROM emp;
```

ENAME	SAL	COMM	月收入
SMITH	800		800
ALLEN	1600	300	1900
WARD	1250	500	1750
JONES	2975		2975
MARTIN	1250	1400	2650
BLAKE	2850		2850

CLARK	2450		2450
SCOTT	3000	50	3050
KING	5000		5000
TURNER	1500	0	1500
ADAMS	1100		1100
JAMES	950		950
FORD	3000		3000
MILLER	1300		1300

已选择 14 行。

### 示例二：使用 NVL2 函数处理 NULL 值

NVL2 是 Oracle9i 新增加的函数，该函数也用于处理 NULL，其语法格式为 NVL2(expr1,expr2,expr3)。如果 expr1 不是 null，则返回 expr2；如果 expr1 是 null，则返回 expr3。参数 expr1 可以是任意数据类型，而 expr2 和 expr3 可以是除 LONG 之外的任何数据类型。但注意，expr2, expr3 的数据类型必须要与 expr1 的数据类型匹配。使用 NVL2 函数处理 NULL 的示例如下：

```
SQL> SELECT ename,nvl2(comm,sal+comm,sal) FROM emp;
ENAME      NVL2(COMM, SAL+COMM, SAL)
-----
SMITH          800
ALLEN         1900
WARD          1750
JONES         2975
MARTIN        2650
BLAKE         2850
CLARK         2450
SCOTT         3050
KING          5000
TURNER        1500
ADAMS         1100
JAMES          950
FORD          3000
MILLER        1300
已选择 14 行。
```

### 9. 连接字符串

为了显示更有意义的查询结果，有时需要将多个字符串连接起来，连接字符串是使用“||”操作符来完成的。当连接字符串时，如果在字符串中要加入数字值，那么在“||”后可以直接指定数字；如果在字符串中要加入字符和日期值，则必须用单引号引住。下面以连接雇员及其岗位为例，说明连接字符串的方法。示例如下：

```
SQL> SELECT ename||' is a '||job AS "Employee Detail" FROM emp;
Employee Detail
-----
SMITH is a CLERK
ALLEN is a SALESMAN
WARD is a SALESMAN
```

```

JONES is a MANAGER
MARTIN is a SALESMAN
BLAKE is a MANAGER
CLARK is a MANAGER
SCOTT is a ANALYST
KING is a PRESIDENT
TURNER is a SALESMAN
ADAMS is a CLERK
JAMES is a CLERK
FORD is a ANALYST
MILLER is a CLERK
已选择 14 行。

```

#### 4.1.2 使用 WHERE 子句

当执行简单查询语句时，因为没有指定任何限制条件，所以会检索表的所有行。但在实际应用环境中，用户往往只需要获得某些行的数据。例如，检索部门 10 的所有雇员信息，或检索岗位 CLERK 的所有雇员信息等等。在执行查询操作时，通过使用 WHERE 子句可以限制查询显示结果。语法如下：

```
SELECT <*, column [alias], ...> FROM table [WHERE condition(s)];
```

如上所示，WHERE 关键字用于指定条件子句；condition 用于指定具体的条件，如果条件子句返回值为 TRUE，则会检索相应行的数据，如果条件为 FALSE，则不会检索该行数据。当编写条件子句时，需要使用各种比较操作符，下表列出了所有的比较操作符：

比较操作符	含义
=	等于
<>、!=	不等于
>=	大于等于
<=	小于等于
>	大于
<	小于
BETWEEN ... AND ...	在两值之间
IN(list)	匹配于列表值
LIKE	匹配于字符样式
IS NULL	测试 NULL

##### 1. 在 WHERE 条件中使用数字值

当在 WHERE 条件中使用数字值时，既可以用单引号引住数字值，也可以直接引用数值。下面以显示工资高于 2000 的雇员为例，说明在 WHERE 子句中使用数字值的方法。示例如下：

```

SQL> SELECT ename,sal FROM emp WHERE sal>2000;
ENAME      SAL

```

```
-----  
JONES      2975  
BLAKE      2850  
CLARK      2450  
SCOTT      3000  
KING       5000  
FORD       3000
```

已选择 6 行。

### 2. 在 WHERE 条件中使用字符值

当在 WHERE 条件中使用字符值时，必须要用单引号引住。下面以显示雇员 SCOTT 的岗位和工资为例，说明在 WHERE 子句中使用字符值的方法。示例如下：

```
SQL> SELECT job,sal FROM emp WHERE ename='SCOTT';  
JOB          SAL  
-----  
ANALYST     3000
```

注意，因为字符值区分大小写，所以在引用字符值时必须要指定正确的大小写格式，否则不能正确显示输出信息。为了避免字符值的大小写问题，可以使用函数 UPPER 或 LOWER 转换大小写。示例如下：

```
SQL> SELECT job,sal FROM emp WHERE ename='scott';  
未选定行  
SQL> SELECT job,sal FROM emp WHERE lower(ename)='scott';  
JOB          SAL  
-----  
ANALYST     3000
```

### 3. 在 WHERE 条件中使用日期值

当在 WHERE 条件中使用日期值时，必须要用单引号引住，并且日期值必须要符合日期显示格式。如果日期值不符合默认日期显示格式，那么必须使用 TO\_DATE 函数进行转换。下面以显示在 1982 年 1 月 1 日之后雇佣的雇员为例，说明在 WHERE 子句中使用不同格式日期值的方法。

#### 示例一：符合默认日期格式

```
SQL> SELECT ename,sal,hiredate FROM emp  
2 WHERE hiredate>'01-1月 -82';  
ENAME        SAL HIREDATE  
-----  
MILLER      1300 23-1月 -82
```

#### 示例二：不符合默认日期格式

```
SQL> SELECT ename,sal,hiredate FROM emp  
2 WHERE hiredate>TO_DATE('1982-01-01','YYYY-MM-DD');  
ENAME        SAL HIREDATE  
-----  
MILLER      1300 23-1月 -82
```

### 4. 在 WHERE 条件中使用 BETWEEN ... AND 操作符

BETWEEN...AND...操作符用于指定特定范围条件，在 BETWEEN 操作符后指定较小的

一个值，在 AND 操作符后则指定较大的一个值。下面以显示工资在 1000~2000 之间的雇员信息为例，说明使用 BETWEEN ... AND ... 操作符的方法。示例如下：

```
SQL> SELECT ename,sal,hiredate,job FROM emp
  2 WHERE sal between 1000 AND 2000;
    ENAME      SAL HIREDATE   JOB
    -----
    ALLEN      1600 20-2月 -81 SALESMAN
    WARD       1250 22-2月 -81 SALESMAN
    MARTIN     1250 28-9月 -81 SALESMAN
    TURNER     1500 08-9月 -81 SALESMAN
    ADAMS      1100 03-12月 -81 CLERK
    MILLER     1300 23-1月 -82 CLERK
已选择 6 行。
```

## 5. 在 WHERE 条件中使用 LIKE 操作符

LIKE 操作符用于执行模糊查询。当执行查询操作时，如果不能完全确定某些信息的查询条件，但这些信息又具有某些特征，那么可以使用模糊查询。当执行模糊查询时，需要使用通配符“%”和“\_”，其中“%”（百分号）用于表示 0 个或多个字符，而“\_”（下划线）则用于表示单个字符。如果要将通配符“%”和“\_”作为字符值使用，那么需要在 ESCAPE 之后使用转义符。下面以示例说明使用 LIKE 操作符的方法。

### 示例一：显示首字符为 S 的所有雇员名及其工资

```
SQL> SELECT ename,sal FROM emp WHERE ename LIKE 'S%';
    ENAME      SAL
    -----
    SMITH      800
    SCOTT     3000
```

### 示例二：显示第三个字符为大写 A 的所有雇员名及其工资

```
SQL> SELECT ename,sal FROM emp WHERE ename LIKE '__A%';
    ENAME      SAL
    -----
    BLAKE     2850
    CLARK     2450
    ADAMS     1100
```

### 示例三：显示雇员名包含“\_”的雇员信息（其中 ESCAPE 后的字符 a 为转义符）

```
SQL> SELECT ename,sal FROM emp
  2 WHERE ename LIKE '%a_%' ESCAPE 'a';
    ENAME      SAL
    -----
    FL_MARY    3000
```

## 6. 在 WHERE 条件中使用 IN 操作符

IN 操作符用于执行列表匹配操作。当列或表达式结果匹配于列表中的任一个值时，条件子句返回 TRUE。下面以显示工资为 800 和 1250 的雇员信息为例，说明使用 IN 操作符的方法。示例如下：

```
SQL> SELECT ename,sal FROM emp WHERE sal IN (800,1250);
```

```

ENAME      SAL
-----
SMITH      800
WARD       1250
MARTIN    1250

```

### 7. 在 WHERE 条件中使用 IS NULL 操作符

IS NULL 操作符用于检测列或表达式的结果是否为 NULL。如果结果为 NULL，则返回 TRUE；否则返回 FALSE。下面以显示公司总裁为例，说明使用 IS NULL 的方法（公司总裁的管理者编号为 NULL）。示例如下：

```

SQL> SELECT ename,sal FROM emp WHERE mgr IS NULL;
ENAME      SAL
-----
KING      5000

```

注意，当与 NULL 进行比较时，千万不要使用等于 (=)、不等于 (<>) 操作符。尽管使用它们不会有语法错误，但条件子句返回总是 FALSE。示例如下：

```

SQL> SELECT ename,sal FROM emp WHERE mgr=NULL;
未选定行

```

### 8. 在 WHERE 子句中使用逻辑操作符

当执行查询操作时，许多情况下需要指定多个查询条件。当使用多个查询条件时，必须要使用逻辑操作符 AND, OR 和 NOT，它们的含义如下：

操作符	含义
AND	如果条件都是 TRUE，则返回 TRUE，否则返回 FALSE
OR	如果任一个条件是 TRUE，则返回 TRUE，否则返回 FALSE
NOT	如果条件是 FALSE，则返回 TRUE；如果条件是 TRUE，则返回 FALSE

逻辑操作符 AND、OR、NOT 的优先级低于任一种比较操作符，在这三个操作符中，NOT 优先级最高，AND 其次，OR 最低。如果要改变优先级，则需要使用括号。注意，NOT 操作符主要与 BETWEEN ... AND, LIKE, IN 以及 IS NULL 结合使用。下面以示例说明在 WHERE 子句中使用这三种逻辑操作符的方法。

#### 示例一：显示在部门 20 中岗位 CLERK 的所有雇员信息

```

SQL> SELECT ename,sal,job,deptno FROM emp
  2 WHERE deptno=20 AND job='CLERK';
ENAME      SAL JOB      DEPTNO
-----
SMITH      800 CLERK      20
ADAMS     1100 CLERK      20

```

#### 示例二：显示工资高于 2500 或岗位为 MANAGER 的所有雇员信息

```

SQL> SELECT ename,sal,job,deptno FROM emp
  2 WHERE sal>2500 OR job='MANAGER';
ENAME      SAL JOB      DEPTNO
-----
JONES     2975 MANAGER      20

```

```

BLAKE        2850 MANAGER      30
CLARK        2450 MANAGER      10
SCOTT        3000 ANALYST     20
KING         5000 PRESIDENT    10
FORD         3000 ANALYST     20

```

已选择 6 行。

### 示例三：显示补助非空的雇员信息

```

SQL> SELECT ename,sal,comm FROM emp
  2 WHERE comm IS NOT NULL;
ENAME      SAL      COMM
-----
ALLEN      1600    300
WARD       1250    500
MARTIN     1250    1400
SCOTT      3000    50
TURNER     1500    0

```

### 4.1.3 使用 ORDER BY 子句

在执行查询操作时，默认情况下会按照行数据插入的先后顺序来显示行数据。但在实际应用中经常需要对数据进行排序，以显示更直观的数据，数据排序是使用 ORDER BY 子句来完成的，其语法如下：

```
SELECT <*,column [alias], ...> FROM table[WHERE condition(s)] [ORDER BY expr {ASC|DESC}];
```

如上所示，expr 用于指定要排序的列或表达式，ASC 用于指定进行升序排序（默认），DESC 用于指定进行降序排序。注意，当在 SELECT 语句中同时包含多个子句（WHERE, GROUP BY, HAVING, ORDER BY 等）时，ORDER BY 必须是最后一条子句。

#### 1. 升序排序

默认情况下，当使用 ORDER BY 执行排序操作时，数据以升序方式排列。在执行升序排序时，也可以在排序列后指定 ASC 关键字。下面以工资升序显示雇员信息为例，说明使用升序排序的方法。示例如下：

```

SQL> SELECT ename,sal FROM emp WHERE deptno=30
  2 ORDER BY sal;
ENAME      SAL
-----
JAMES      950
WARD       1250
MARTIN     1250
TURNER     1500
ALLEN      1600
BLAKE      2850

```

已选择 6 行。

注意，当执行升序排序时，如果被排序列包含 NULL 值，那么 NULL 会显示在最后面。示例如下：

```
SQL> SELECT ename,sal,comm FROM emp WHERE deptno=30
  2 ORDER BY comm;
ENAME          SAL      COMM
-----
TURNER        1500      0
ALLEN         1600     300
WARD          1250     500
MARTIN        1250    1400
BLAKE         2850
JAMES         950
```

## 2. 降序排序

当使用 ORDER BY 子句执行排序操作时，默认情况下会执行升序排序。为了执行降序排序，必须要指定 DESC 关键字。下面以降序显示雇员工资为例，说明使用降序排序的方法。示例如下：

```
SQL> SELECT ename,sal,comm FROM emp WHERE deptno=30
  2 ORDER BY sal DESC;
ENAME          SAL      COMM
-----
BLAKE         2850
ALLEN         1600     300
TURNER        1500      0
WARD          1250     500
MARTIN        1250    1400
JAMES         950
```

已选择 6 行。

注意，当执行降序排序时，如果排序列存在 NULL 值，那么 NULL 会显示在最前面。示例如下：

```
SQL> SELECT ename,sal,comm FROM emp WHERE deptno=30
  2 ORDER BY comm DESC;
ENAME          SAL      COMM
-----
BLAKE         2850
JAMES         950
MARTIN        1250    1400
WARD          1250     500
ALLEN         1600     300
TURNER        1500      0
```

已选择 6 行。

## 3. 使用多列排序

当使用 ORDER BY 子句执行排序操作时，不仅可以基于单个列或单个表达式进行排序，也可以基于多个列或多个表达式进行排序。当以多个列或多个表达式进行排序时，首先按照第一个列或表达式进行排序，当第一个列或表达式存在相同数据时，然后以第二个列或表达式进行排序。下面以工资升序、补助降序显示雇员信息为例，说明使用多列排序的方法。示例如下：

```
SQL> SELECT ename,sal,comm FROM emp WHERE deptno=30
  2 ORDER BY sal ASC,comm DESC;
ENAME          SAL      COMM
-----
JAMES           950
MARTIN         1250     1400
WARD            1250     500
TURNER          1500      0
ALLEN           1600     300
BLAKE           2850
已选择 6 行。
```

#### 4. 使用非选择列表列进行排序

当使用 ORDER BY 子句执行排序操作时，多数情况下选择列表都会包含被排序列。但在实际情况下，选择列表可以不包含排序列。下面以工资降序显示雇员名为例，说明使用非选择列表列进行排序的方法。示例如下：

```
SQL> SELECT ename FROM emp ORDER BY sal DESC;
ENAME
-----
KING
SCOTT
FORD
JONES
BLAKE
CLARK
ALLEN
TURNER
MILLER
WARD
MARTIN
ADAMS
JAMES
SMITH
已选择 14 行。
```

#### 5. 使用列别名排序

如果在 WHERE 子句中为列或表达式定义了别名，那么当执行排序操作时，既可以使用列或表达式进行排序，也可以使用列别名进行排序。下面以降序显示雇员全年工资为例，说明使用列别名进行排序的方法。示例如下：

```
SQL> SELECT ename,sal*12 AS "全年工资" FROM emp
  2 WHERE deptno=30
  3 ORDER BY "全年工资" DESC;
ENAME      全年工资
-----
BLAKE      34200
ALLEN      19200
TURNER     18000
```

```

WARD          15000
MARTIN        15000
JAMES         11400
已选择 6 行。

```

### 6. 使用列位置编号排序

当执行排序操作时，不仅可以指定列名、列别名进行排序，也可以按照列或表达式在选择列表中的位置进行排序。如果列名或表达式名称很长，那么使用列位置排序可以缩短排序语句的长度。另外当使用 UNION, UNION ALL, INTERSECT, MINUS 等集合操作符合并查询结果时，如果选择列表中的列名不同，并且希望进行排序，那么必须使用列位置。示例如下：

```

SQL> SELECT ename,sal*12 "全年工资" FROM emp
      2 WHERE deptno=20 ORDER BY 2 DESC;
    ENAME      全年工资
    -----
SCOTT          36000
FORD           36000
JONES          35700
ADAMS          13200
SMITH          9600

```

## 4.2 使用 DML 语句

DML 语句 (INSERT, UPDATE, DELETE) 用于操纵表和视图的数据。通过执行 INSERT 语句，可以给表增加数据；通过执行 UPDATE 语句，可以更新表的数据；通过执行 DELETE 语句，可以删除表的数据。

### 4.2.1 插入数据

当要给表增加数据时，可以使用 INSERT 语句。使用 INSERT 语句既可以为表插入单行数据，也可以通过子查询将一张表的多行数据插入到另一张表中。从 Oracle9i 开始，Oracle 还提供了多表插入功能，即使用一条 INSERT 语句同时为多张表插入数据。但使用 INSERT 语句插入数据时，应注意：

- 如果为数字列插入数据，则可以直接提供数字值；如果为字符列或日期列插入数据，则必须使用单引号引住。
- 当插入数据时，数据必须要满足约束规则，并且必须要为主键列和 NOT NULL 列提供数据。
- 当插入数据时，数据必须要与列的个数和顺序保持一致。

#### 1. 插入单行数据

插入单行数据是使用 INSERT ... VALUES 语句来完成的，其语法如下：

```

INSERT INTO <table> [(column[,column,...])]
VALUES (value[,value,...])

```

如上所示，table 用于指定表名或视图名；column 用于指定列名，如果要指定多个列，那么列之间要用逗号分开；value 用于提供列数据。当使用 INSERT 语句插入数据时，既可以指

定列列表，也可以不指定列列表。如果不指定列列表，那么在 VALUES 子句中必须要为每个列提供数据，并且数据顺序要与表列顺序完全一致。如果指定列的列表，则只需要为相应列提供数据。下面通过示例说明插入单行数据的方法。

### 示例一：不使用列的列表插入单行数据

当使用 INSERT 插入数据时，可以不指定列的列表。注意，如果不指定列的列表，那么必须要为所有列提供数据，并且数据的顺序必须与列的顺序保持一致。示例如下：

```
SQL> INSERT INTO dept VALUES(50, 'TRAIN', 'BOSTON');
已创建 1 行。
```

### 示例二：使用列的列表插入单行数据

当使用列的列表插入数据时，只需为相应列提供数据。而对于那些未出现在列的列表中的列，其数据为 NULL。示例如下：

```
SQL> INSERT INTO emp (empno,ename,job,hiredate)
  2  VALUES(1234,'JOHN','CLERK','01-3月-86');
已创建 1 行。
```

### 示例三：使用特定格式插入日期值

在使用 INSERT 语句为表插入数据时，默认情况下日期值必须要与于日期格式、日期语言匹配，否则在插入数据时会显示错误信息。如果用户希望使用习惯方式插入日期数据，那么必须要使用 TO\_DATE 函数进行转换。示例如下：

```
SQL> INSERT INTO emp (empno,ename,job,hiredate)
  2  VALUES(1356,'MARY','CLERK',
  3  to_date('1983-10-20','YYYY-MM-DD'));
已创建 1 行。
```

### 示例四：使用 DEFAULT 提供数据

从 Oracle9i 开始，当使用 INSERT 语句插入数据时，可以使用 DEFAULT 关键字提供数值。当指定 DEFAULT 时，如果列存在默认值，则会使用其默认值；如果列不存在默认值，则自动使用 NULL。示例如下：

```
SQL> INSERT INTO dept VALUES(60, 'MARKET', DEFAULT);
已创建 1 行。
SQL> SELECT * FROM dept WHERE deptno=60;
DEPTNO DNAME          LOC
----- -----
      60 MARKET
```

## 2. 使用子查询插入数据

当使用 VALUES 子句插入数据时，一次只能插入一行数据；当使用子查询插入数据时，可以将一张表的数据复制到另一张表中。当处理行迁移、复制表数据或者装载外部表数据到数据库时，可以使用子查询插入数据。其语法如下：

```
INSERT INTO <table> [(column[,column,...])] subQuery
```

如上所示，table 用于指定表名或视图名；column 用于指定列名，如果要指定多个列，那么列之间要用逗号分开；subquery 用于指定为目标表提供数据的子查询。当使用子查询插入数据时，INSERT 列的数据类型和个数必须要与子查询列的数据类型和个数完全匹配。下面通过示例说明使用子查询插入数据的方法。

### 示例一：使用子查询插入数据

```
SQL> INSERT INTO employee (empno,ename,sal,deptno)
  2  SELECT empno,ename,sal,deptno FROM emp
  3  WHERE deptno=20;
已创建 6 行。
```

### 示例二：使用子查询执行直接装载

```
SQL> INSERT /*+APPEND */ INTO employee (empno,ename,sal,deptno)
  2  SELECT empno,ename,sal,deptno FROM emp
  3  WHERE deptno=20;
已创建 6 行。
```

注意，尽管以上两条语句的执行结果一样，但第二条语句使用`/*+APPEND */`来表示采用直接装载方式。当要装载大批量数据时，采用第二种方法装载数据的速度要远远优于第一种方法。

### 3. 使用多表插入数据

在 Oracle9i 之前，当执行 `INSERT` 语句插入数据时，只能为单个表插入数据。从 Oracle9i 开始，使用 `INSERT` 语句可以将某张表的数据同时插入到多张表中。语法如下：

```
INSERT ALL insert_into_clause [value_clause] subquery;
  INSERT conditional_insert_clause subquery;
```

如上所示，`insert_into_clause` 用于指定 `INSERT` 子句；`value_clause` 用于指定值子句；`subquery` 用于指定提供数据的子查询；`conditional_insert_clause` 用于指定 `INSERT` 条件子句。下面通过示例说明使用 `INSERT` 语句执行多表插入的方法。

### 示例一：使用 ALL 操作符执行多表插入

当使用 `ALL` 操作符执行多表插入时，在每个条件子句上都要执行 `INTO` 子句后的子查询。示例如下：

```
SQL> INSERT ALL
  2  WHEN deptno=10 THEN INTO dept10
  3  WHEN deptno=20 THEN INTO dept20
  4  WHEN deptno=30 THEN INTO dept30
  5  WHEN job='CLERK' THEN INTO clerk
  6  ELSE INTO other
  7  SELECT * FROM emp;
已创建 18 行。
```

如上所示，在执行了以上 `INSERT` 语句之后，会将部门 10 的雇员信息插入到 `DEPT10` 表，将部门 20 的雇员信息插入到 `DEPT20` 表，将部门 30 的雇员信息插入到 `DEPT30`，将岗位 CLERK 的所有雇员插入到 `CLERK`，将其他行插入到 `OTHER` 表。

### 示例二：使用 FIRST 操作符执行多表插入

当使用 `FIRST` 操作符执行多表插入时，如果数据已经满足了先前条件，并且已经被插入到某表，那么该行数据在后续插入中将不会被再次使用。示例如下：

```
SQL> INSERT FIRST
  2  WHEN deptno=10 THEN INTO dept10
  3  WHEN deptno=20 THEN INTO dept20
  4  WHEN deptno=30 THEN INTO dept30
  5  WHEN job='CLERK' THEN INTO clerk
  6  ELSE INTO other
```

```
7  SELECT * FROM emp; 已创建 14 行。
```

### 4.2.2 更新数据

当要更新表中行的数据时，可以使用 UPDATE 语句。使用 UPDATE 语句时，既可以使用表达式更新列值，也可以使用子查询更新一列或多列的数据。使用 UPDATE 语句应注意以下事项：

- 如果要更新数字列，则可以直接提供数字值；如果要更新字符列或日期列，则数据必须用单引号引住。
- 当更新数据时，数据必须要满足约束规则。
- 当更新数据时，数据必须要与列的数据类型匹配。

#### 1. 使用表达式更新数据

当要直接修改某行或某几行的数据时，可以在 SET 子句中直接指定列的新值或者指定表达式。注意，当执行 UPDATE 语句时如果未指定 WHERE 子句，则会修改表中所有行的数据。UPDATE 语句的语法如下：

```
UPDATE <table|view>
SET <column> = <value> [, <column> = <value>]
[WHERE <condition>];
```

如上所示，table 用于指定要更新的表；view 用于指定要更新的视图；column 用于指定要更新的列；value 用于指定更新后的列值；condition 用于指定条件子句。下面以示例说明使用表达式更新数据的方法。

#### 示例一：更新单列数据

```
SQL> UPDATE emp SET sal=2460 WHERE ename='SCOTT';
已更新 1 行。
```

#### 示例二：更新多列数据

当使用 UPDATE 语句修改表中行的数据时，既可以修改一列，也可以修改多列。当修改多列时，列之间用逗号分开。示例如下：

```
SQL> UPDATE emp SET sal=sal*1.1,comm=sal*0.1
    2 WHERE deptno=20;
已更新 5 行。
```

#### 示例三：更新日期列数据

当更新日期列中的数据时，数据格式一定要与默认日期格式、日期语言相匹配，否则会显示错误信息。如果使用习惯方式指定日期值，就需要使用 TO\_DATE 函数进行转换。示例如下：

```
SQL> UPDATE emp SET hiredate=TO_DATE('1984/01/01','YYYY/MM/DD')
    2 WHERE empno=7788;
已更新 1 行。
```

#### 示例四：使用 DEFAULT 选项更新数据

从 Oracle9i 开始，当执行 UPDATE 语句更新列数据时，可以使用 DEFAULT 选项提供数据值。如果列存在默认值，则会使用默认值更新数据；如果列不存在默认值，则使用 NULL。示例如下：

```
SQL> SELECT job FROM emp WHERE ename='SCOTT';
JOB
-----

```

```

ANALYST
SQL> UPDATE emp SET job=DEFAULT WHERE ename='SCOTT';
已更新 1 行。
SQL> SELECT job FROM emp WHERE ename='SCOTT';
JOB
-----

```

#### 示例五：更新违反约束规则的数据：

当使用 UPDATE 语句更新数据时，新数据必须满足约束规则，否则会显示错误信息。示例如下：

```

SQL> UPDATE emp SET deptno=55 WHERE empno=7788;
UPDATE emp SET deptno=55 WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-02291: 违反完整约束条件 (SCOTT.FK_DEPTNO) - 未找到父项关键字

```

如上所示，因为在 DEPT 表中不存在部门 55，所以当修改 DEPTNO 列为 55 时会违反参照完整性约束。

#### 2. 使用子查询更新数据

当使用 UPDATE 语句更新数据时，不仅可以使用表达式或数值直接更新数据，也可以使用子查询更新数据。某些情况下，使用子查询执行效率更好。另外，当使用触发器复制表之间的数据时，使用子查询可以更新相关表的数据。下面通过示例说明使用子查询更新数据的方法。

##### (1) 更新关联数据

当更新关联数据时，使用子查询可以降低网络开销。假设你希望使雇员 SCOTT 的岗位、工资、补助与雇员 SMITH 完全相同，如果使用表达式或列值进行修改，那么首先需要取得雇员 SMITH 的岗位、工资和补助，然后执行 UPDATE 语句进行修改。示例如下：

```

SQL> SELECT job,sal,comm FROM emp WHERE ename='SMITH';
JOB          SAL      COMM
-----
CLERK        2200     200
SQL> UPDATE emp SET job='CLERK',sal=2200,comm=200
2 WHERE ename='SCOTT';
已更新 1 行。

```

在存在可以确定的条件时，通过使用子查询只需要编写一条语句就可以完成这项任务，从而降低了网络开销。示例如下：

```

SQL> UPDATE emp SET (job,sal,comm)=(
2 SELECT job,sal,comm FROM emp WHERE ename='SMITH')
3 WHERE ename='SCOTT';
已更新 1 行。

```

##### (2) 复制表数据

当使用触发器复制表数据时，如果表 A 数据被修改，那么表 B 数据也应该修改。通过使用子查询，可以基于一张表修改另一张表的数据。示例如下：

```

SQL> UPDATE employee SET deptno=
2 (SELECT deptno FROM emp WHERE empno=7788)

```

```
3 WHERE job=(SELECT job FROM emp WHERE empno=7788);
已更新 5 行。
```

### 4.2.3 删 除 数据

当要删除表中某行的数据时，可以使用 DELETE 语句。使用该语句既可以删除一行数据，也可以删除多行数据。其语法如下：

```
DELETE FROM <table|view> [WHERE <condition>];
```

如上所示，table 用于指定表名；view 用于指定视图名；condition 用于指定条件子句。当使用 DELETE 语句删除数据时，如果不指定 WHERE 条件子句，那么会删除表或视图的所有行。下面以示例说明使用 DELETE 语句删除数据的方法。

#### 示例一：删除满足条件的数据

当使用 DELETE 语句删除数据时，通过指定 WHERE 子句可以删除满足条件的数据。假定要解雇员 SMITH，那么在删除时指定 WHERE 子句。示例如下：

```
SQL> DELETE FROM emp WHERE ename='SMITH';
已删除 1 行。
```

#### 示例二：删除表的所有数据

当使用 DELETE 语句删除表中的数据时，如果不指定 WHERE 子句，那么会删除表中的所有数据。示例如下：

```
SQL> DELETE FROM emp;
已删除 15 行。
```

#### 示例三：使用 TRUNCATE TABLE 截断表

当使用 DELETE 语句删除表的所有数据时，不会释放表所占用的空间。如果用户确定要删除表的所有数据，那么使用“TRUNCATE TABLE”语句速度更快。示例如下：

```
SQL> TRUNCATE TABLE emp;
表已截掉。
```

如上所示，使用 TRUNCATE TABLE 语句不仅会删除表的所有数据，而且还会释放表段所占用的空间。注意，DELETE 语句的操作可以回退，但 TRUNCATE TABLE 语句的操作不能回退。

#### 示例四：使用子查询删除数据

当使用 DELETE 语句删除数据时，可以直接在 WHERE 子句中指定值，并根据条件来删除数据。另外，也可以在 WHERE 子句中使用子查询作为条件。假定要解雇 SALES 部门的所有雇员，那么在 WHERE 子句中需要使用子查询。示例如下：

```
SQL> DELETE FROM emp WHERE deptno=
      2 (SELECT deptno FROM dept WHERE dname='SALES');
已删除 6 行。
```

#### 示例五：删除主表数据的注意事项

当使用 DELETE 语句删除数据时应该注意：当删除主表数据时，必须要确保从表不存在相关记录，否则会显示错误信息。示例如下：

```
SQL> DELETE FROM dept WHERE deptno=10;
DELETE FROM dept WHERE deptno=10
*
```

ERROR 位于第 1 行:

ORA-02292: 违反完整约束条件 (SCOTT.FK\_DEPTNO) - 已找到了记录日志

如上所示，因为在 DEPT 表和 EMP 表之间有主从关系，所以当删除主表 (DEPT) 数据时，必须要确保从表 (EMP) 不存在相关子记录。因为部门 10 存在雇员，所以删除失败。

### 4.3 使用事务控制语句

事务用于确保数据库数据的一致性，它由一组相关的 DML 语句组成。该组 DML 语句所执行的操作要么全部成功，要么全部取消。假定某客户要将储蓄帐户的现金转移到支票帐户，此时需要执行两个操作：减少储蓄帐户的现金；同时，增加支票帐户的金额。为了确保数据库数据一致性，这两个操作必须全部成功，或者全部取消。

数据库事务主要由 INSERT、UPDATE、DELETE 和 SELECT ... FOR UPDATE 语句组成。当在应用程序中执行第一条 SQL 语句时，开始事务；当执行 COMMIT 或 ROLLBACK 语句时结束事务。

#### 4.3.1 事务和锁

当执行事务操作 (DML 语句) 时，Oracle 会在被作用表上加表锁，以防止其他用户改变表结构；同时会在被作用行上加行锁，以防止其他事务在相应行上执行 DML 操作。假定会话 A 更新 EMP 表行的数据，那么会在表 EMP 上加表锁；此时如果其他会话修改表结构，则会显示错误信息。示例如下：

会话 A	会话 B
<pre>SQL&gt; UPDATE emp SET sal=sal*1.1   2 WHERE ename='SCOTT'; 已更新 1 行。</pre>	<pre>SQL&gt; ALTER TABLE emp ADD remark VARCHAR2(100); ERROR 位于第 1 行: ORA-00054: 资源忙, 要求指定 NOWAIT</pre>

在 Oracle 数据库中，为了确保数据库数据的读一致性，不允许其他用户读取脏数据（未提交事务）。假定会话 A 将雇员 SCOTT 工资修改为 2000（未提交），那么其他会话将只能查询到原来的工资；只有在会话 A 提交了事务之后，其他会话才能查询到新工资。示例如下：

会话 A	会话 B
<pre>SQL&gt; UPDATE emp SET sal=2000   2 WHERE ename='SCOTT'; 已更新 1 行。</pre>	<pre>SQL&gt; SELECT sal FROM emp WHERE ename='SCOTT';           SAL -----       3000</pre>

#### 4.3.2 提交事务

使用 COMMIT 语句可以提交事务。当执行了 COMMIT 语句之后，会确认事务变化、结束事务、删除保存点、释放锁。当使用 COMMIT 语句结束事务之后，其他会话将可以看到

事务变化后的新数据。示例如下：

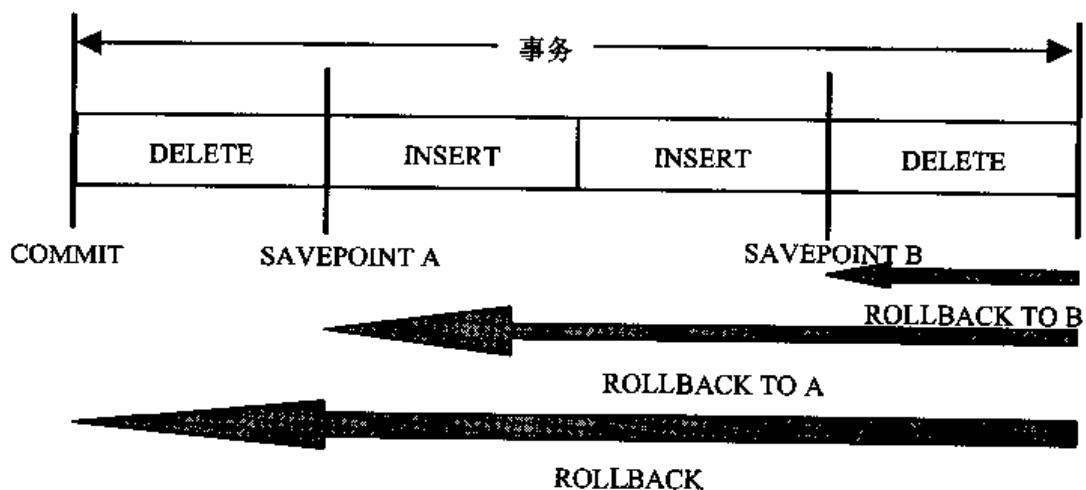
会话 A	会话 B
<pre>SQL&gt; UPDATE emp SET sal=2000   2 WHERE ename='SCOTT'; SQL&gt; COMMIT; 提交完成。</pre>	<pre>SQL&gt; SELECT sal FROM emp WHERE ename='SCOTT';           SAL -----         2000</pre>

如上所示，当会话 A 更新了 SCOTT 雇员的工资，并且提交了事务之后，其他会话（会话 B）将可以取得更新后的新数据。注意，当出现以下情况时会自动提交事务：

- 当执行 DDL 语句时会自动提交事务，例如 CREATE TABLE, ALTER TABLE, DROP TABLE 等语句。
- 当执行 DCL 语句（GRANT、REVOKE）时。
- 当退出 SQL\*Plus 时。

#### 4.3.3 回退事务

在介绍回退事务之前，先给大家介绍一下保存点（savepoint）的概念和作用。保存点是事务中的一点，它用于取消部分事务。当结束事务时，会自动删除该事务所定义的所有保存点。在执行 ROLLBACK 命令时，通过指定保存点可以取消部分事务。如下图所示：



如上所示，当设置了保存点 A 和 B 之后，如果执行“ROLLBACK TO B”，则取消保存点 B 后面的操作（DELETE）；如果执行“ROLLBACK TO A”，则会取消保存点 A 后的操作（两个 INSERT、一个 DELETE）；如果执行“ROLLBACK”，则会取消所有事务操作，并结束事务。

##### 1. 设置保存点

设置保存点是使用 SQL 命令 SAVEPOINT 来完成的。另外，开发人员在编写应用程序时，也可以使用包 DBMS\_TRANSACTION 的过程 SAVEPOINT 来设置保存点。示例如下：

SQL> savepoint a;

或

SQL> exec dbms\_transaction.savepoint('a')

## 2. 取消部分事务

为了取消部分事务，用户可以回退到保存点。回退到保存点既可以使用 ROLLBACK 命令，也可以使用包 DBMS\_TRANSACTION 的过程 ROLLBACK\_SAVEPOINT。示例如下：

```
SQL> rollback to a;
```

或

```
SQL> exec dbms_transaction.rollback_savepoint('a')
```

## 3. 取消全部事务

使用 ROLLBACK 命令可以取消全部事务，开发人员在编写应用程序时也可以使用包 DBMS\_TRANSACTION 的过程 ROLLBACK 取消全部事务。示例如下：

```
SQL> rollback;
```

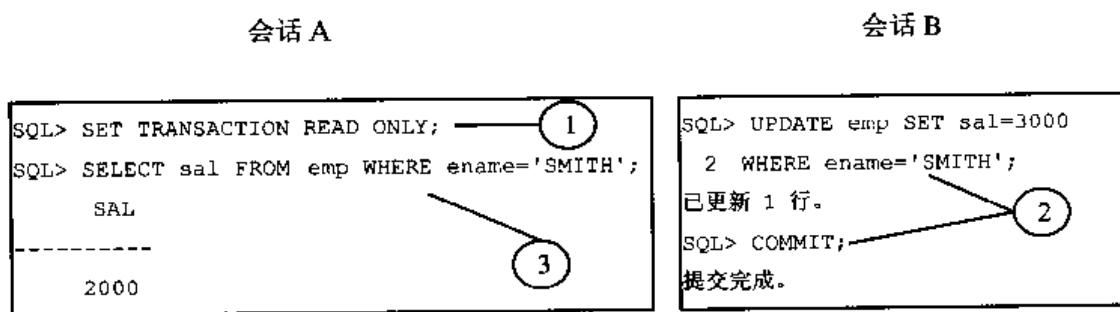
或

```
SQL> exec dbms_transaction.rollback
```

当使用 ROLLBACK 取消全部事务时，会取消所有事务变化、结束事务、删除所有保存点并释放锁。当出现系统灾难或应用程序地址例外时，会自动回退其事务变化。

### 4.3.4 只读事务

只读事务是指只允许执行查询操作，而不允许执行任何 DML 操作的事务。当使用只读事务时，可以确保用户取得特定时间点的数据。假定企业需要在每天 16 点统计最近 24 小时的销售信息，而不统计当天 16 点之后的销售信息，那么用户可以使用只读事务。在设置了只读事务之后，尽管其他会话可能会提交新事务，但只读事务将不会取得新的数据变化，从而确保取得特定时间点的数据信息。如下图所示：



如上所示，假定会话 A 在时间点 1 设置了只读事务，会话 B 在时间点 2 更新了 SMITH 的工资并执行了提交操作，会话 A 在时间点 3 查询 SMITH 工资时将会取得时间点 1 的工资值，而不会取得时间点 2 的新工资值。注意，当设置只读事务时，该语句必须是事务开始的第一条语句。另外在应用程序中，使用过程 READ\_ONLY 也可以设置只读事务。示例如下：

```
SQL> SET TRANSACTION READ ONLY;
```

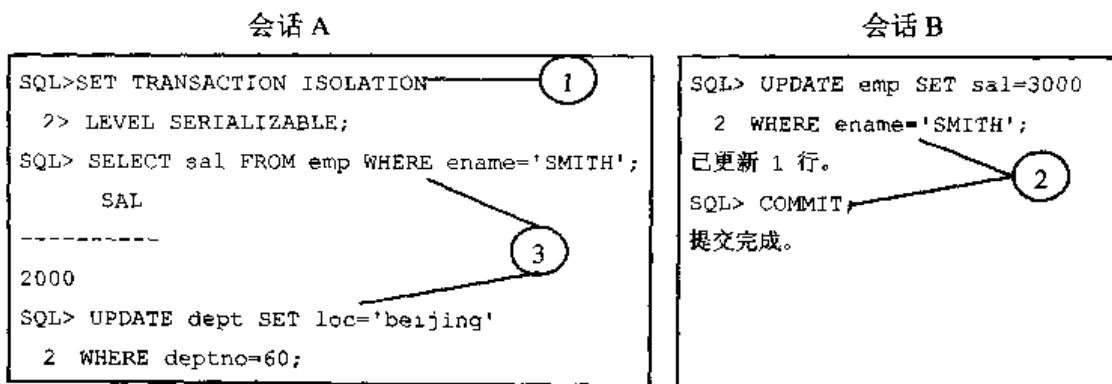
或

```
SQL> exec dbms_transaction.read_only
```

### 4.3.5 顺序事务

只读事务可以使得用户取得特定时间点的数据信息，但当设置了只读事务时，会话将不能执行 INSERT/UPDATE/DELETE 等 DML 操作。为了使得用户可以取得特定时间点的数据，

并且允许执行 DML 操作，可以使用顺序事务。如下图所示：



如图中示，假定会话 A 在时间点 1 设置了顺序事务，会话 B 在时间点 2 更新了 SMITH 的工资并执行了提交操作，会话 A 在时间点 3 查询 SMITH 工资时将会取得时间点 1 的工资信息，而不会取得时间点 2 的新工资值。注意，当设置顺序事务时，该语句必须是事务开始的第一条语句。示例如下：

```
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## 4.4 数据分组

在开发数据库应用程序时，经常需要统计数据库中的数据。当执行数据统计时，需要将表中的数据划分成几个组，最终统计每个组的数据结果。假设用户经常需要统计不同部门的雇员总数、雇员的平均工资、雇员的工资总计，并且希望最终生成如下统计报表：

部门号	雇员总数	平均工资	工资总计
10	3	2916.6	8750
20	6	2815	14075
30	6	1566.6	9400

在关系数据库中，数据分组是通过使用 GROUP BY 子句、分组函数以及 HAVING 子句共同实现的。其中 GROUP BY 子句用于指定要分组的列（例如 DEPTNO），而分组函数则用于显示统计结果（如 COUNT, AVG, SUM 等），而 HAVING 子句则用于限制分组显示结果。

### 4.4.1 分组函数

分组函数用于统计表的数据。与单行函数不同，分组函数作用于多行，并返回一个结果，所以有时也被称为多行函数。一般情况下，分组函数要与 GROUP BY 子句结合使用。在使用分组函数时，如果忽略了 GROUP BY 子句，那么会汇总所有行，并产生一个结果。Oracle 数据库提供了大量的分组函数，在这里给大家介绍最常用的七个分组函数：

- MAX：该函数用于取得列或表达式的最大值，它适用于任何数据类型。
- MIN：该函数用于取得列或表达式的最小值，它适用于任何数据类型。
- AVG：该函数用于取得列或表达式的平均值，它只适用于数字类型。
- SUM：该函数用于取得列或表达式的总和，它只适用于数字类型。

- COUNT：该函数用于取得总计行数。
- VARIANCE：该函数用于取得列或表达式的方差，并且该函数只适用于数字类型。当只有一行数据时，其返回值为0；当存在多行数据时，方差是按照如下公式计算取得： $(\sum(expr)^2 - \sum(expr)^2 / COUNT(expr)) / (COUNT(expr) - 1)$
- STDDEV：该函数用于取得列或表达式的标准偏差，并且该函数只适用于数字类型。当只有一行数据时，其返回值为0；当存在多行数据时，Oracle按照方差的平方根来计算标准偏差。

当使用分组函数时，分组函数只能出现在选择列表、ORDER BY 和 HAVING 子句中，而不能出现在 WHERE 和 GROUP BY 子句中。另外，使用分组函数还有以下一些注意事项：

- 当使用分组函数时，除了函数 COUNT(\*)之外，其他分组函数都会忽略 NULL 行。
- 当执行 SELECT 语句时，如果选择列表同时包含列、表达式和分组函数，那么这些列和表达式必须出现在 GROUP BY 子句中。
- 当使用分组函数时，在分组函数中可以指定 ALL 和 DISTINCT 选项。其中 ALL 是默认选项，该选项表示统计所有行数据（包括重复行）；如果指定 DISTINCT，则只会统计不同行值。

### 示例一：取得最大值和最小值

当执行 SELECT 语句时，分组函数 MAX 和 MIN 可以用于取得最大值和最小值。下面以取得雇员最高工资和最低工资为例，说明使用这两个函数的方法。示例如下：

```
SQL> SELECT max(sal),min(sal) FROM emp;
      MAX(SAL)    MIN(SAL)
----- -----
      5000        800
```

### 示例二：取得平均值和总和

当执行 SELECT 语句时，分组函数 AVG 和 SUM 可以用于取得平均值和总和。下面以取得所有雇员的平均工资和工资总和为例，说明使用这两个函数的方法。示例如下：

```
SQL> SELECT avg(sal),sum(sal) FROM emp;
      AVG(SAL)    SUM(SAL)
----- -----
     2073.21429    29025
```

### 示例三：取得总计行数

当执行 SELECT 语句时，使用函数 COUNT(\*)可以取得总计行数。下面以显示 EMP 表的总计行数为例，说明使用该函数的方法。示例如下：

```
SQL> SELECT count(*) FROM emp;
      COUNT(*)
-----
      14
```

另外，在 count 函数中还可以引用表达式。因为分组函数会忽略 NULL 行，所以使用 count(表达式)会显示 NOT NULL 的总计行数。下面以显示补助非空的所有雇员总数为例，说明使用 count(表达式)的方法。示例如下：

```
SQL> SELECT count(comm) FROM emp;
      COUNT(COMM)
```

-----  
5

#### 示例四：取得方差和标准偏差

当执行 SELECT 语句时，函数 VARIANCE 和 STDDEV 可以用于取得方差和标准偏差。下面以计算 SAL 列的方差和标准偏差为例，说明使用函数 VARIANCE 和 STDDEV 的方法。示例如下：

```
SQL> SELECT variance(sal),stddev(sal) FROM emp;
VARIANCE(SAL) STDDEV(SAL)
```

-----  
1398313.87 1182.50322

#### 示例五：取消重复值

当在 SELECT 语句中使用分组函数时，默认情况下会使用 ALL 选项显示所有数据的统计值（包括重复值）。为了在显示统计时取消重复值，需要使用 DISTINCT 选项。下面以显示总计部门数为例，说明使用该选项的方法。示例如下：

```
SQL> SELECT count(distinct deptno) AS distinct_dept FROM emp;
DISTINCT_DEPT
```

-----  
3

### 4.4.2 GROUP BY 和 HAVING

GROUP BY 子句用于对查询结果进行分组统计，而 HAVING 子句则用于限制分组显示结果。注意，如果在选择列表中同时包含有列、表达式和分组函数，那么这些列和表达式必须出现在 GROUP BY 子句中。使用 GROUP BY 和 HAVING 子句的语法如下：

```
SELECT column, group_function FROM table
[WHERE condition] [GROUP BY group_by_expression]
[HAVING group_condition];
```

如上所示，column 用于指定选择列表中的列或表达式；group\_function 用于指定分组函数；condition 用于指定条件子句；group\_by\_expression 用于指定分组表达式；group\_condition 用于指定排除分组结果的条件。下面以示例说明使用 GROUP BY 子句和 HAVING 子句的方法。

#### 示例一：使用 GROUP BY 进行单列分组

单列分组是指在 GROUP BY 子句中使用单个列生成分组统计结果。当进行单列分组时，会基于列的每个不同值生成一个数据统计结果。下面以显示每个部门的平均工资和最高工资为例，说明使用 GROUP BY 进行单列分组的方法。示例如下：

```
SQL> SELECT deptno,avg(sal),max(sal) FROM emp
2 GROUP BY deptno;
DEPTNO AVG(SAL) MAX(SAL)
```

-----  
10 2916.66667 5000  
20 2175 3000  
30 1566.66667 2850

#### 示例二：使用 GROUP BY 进行多列分组

多列分组是指在 GROUP BY 子句中使用两个或两个以上的列生成分组统计结果。当进行

多列分组时，会基于多个列的不同值生成数据统计结果。下面以显示每个部门每种岗位的平均工资和最高工资为例，说明使用 GROUP BY 进行多列分组的方法。示例如下：

```
SQL> SELECT deptno, job, avg(sal), max(sal) FROM emp
  2 GROUP BY deptno, job;
    DEPTNO JOB          AVG(SAL)   MAX(SAL)
    -----
      10 CLERK        1300       1300
      10 MANAGER      2450       2450
      10 PRESIDENT    5000       5000
      20 CLERK        950        1100
      20 ANALYST      3000       3000
      20 MANAGER      2975       2975
      30 CLERK        950        950
      30 MANAGER      2850       2850
      30 SALESMAN     1400       1600
```

已选择 9 行。

### 示例三：使用 HAVING 子句限制分组显示结果

HAVING 子句用于限制分组统计结果，并且 HAVING 子句必须跟在 GROUP BY 子句后面。下面以显示平均工资低于 2000 的部门号、平均工资及最高工资为例，说明使用 HAVING 子句的方法。示例如下：

```
SQL> SELECT deptno, avg(sal), max(sal) FROM emp
  2 GROUP BY deptno
  3 HAVING avg(sal)<2000;
    DEPTNO  AVG(SAL)   MAX(SAL)
    -----
      30  1566.66667    2850
```

当执行数据统计时，读者一定要正确的使用 GROUP BY 子句、WHERE 子句和分组函数，以避免不必要的错误。使用 GROUP BY 子句、WHERE 子句和分组函数有以下一些注意事项：

- 分组函数只能出现在选择列表、HAVING 子句和 ORDER BY 子句中。
- 如果在 SELECT 语句中同时包含有 GROUP BY、HAVING 以及 ORDER BY 子句，则必须将 ORDER BY 子句放在最后。默认情况下，当使用 GROUP BY 子句统计数据时，会自动按照分组列的升序方式显示统计结果。通过使用 ORDER BY 子句，可以改变数据分组的排序方式，示例如下：

```
SQL> SELECT deptno, avg(sal), max(sal) FROM emp
  2 GROUP BY deptno ORDER BY avg(sal);
    DEPTNO  AVG(SAL)   MAX(SAL)
    -----
      30  1566.66667    2850
      20      2175       3000
      10  2916.66667    5000
```

- 如果选择列表包含有列、表达式和分组函数，那么这些列和表达式必须出现在 GROUP BY 子句中，否则会显示错误信息。如下所示：

```
SQL> SELECT deptno, job, avg(sal) FROM emp
  2 GROUP BY deptno;
SELECT deptno, job, avg(sal) FROM emp
*
*
```

ERROR 位于第 1 行:

ORA-00979: 不是 GROUP BY 表达式

- 当限制分组显示结果时, 必须要使用 HAVING 子句, 而不能在 WHERE 子句中使用分组函数限制分组显示结果, 否则会显示错误信息。如下所示:

```
SQL> SELECT DEPTNO, AVG(SAL) FROM EMP
  2 WHERE SUM(SAL)>1000 GROUP BY DEPTNO;
  WHERE SUM(SAL)>1000 GROUP BY DEPTNO
  *
```

ERROR 位于第 2 行:

ORA-00934: 此处不允许使用分组函数

#### 4.4.3 ROLLUP 和 CUBE

当直接使用 GROUP BY 执行数据统计时, 只会生成列的相应数据统计。例如, 如果要统计不同部门不同岗位的平均工资, 那么直接使用 GROUP BY 子句。其统计结果见如下表格:

岗位 部门号	CLERK	ANALYST	MANAGER	PRESIDENT	SALESMAN
10	1300		2450	5000	
20	2050	3500	2975		
30	950		2850		1400

在实际应用中, 如果以上统计结果还不能满足需求, 还希望产生横向、纵向的统计结果, 此时可以使用 ROLLUP 和 CUBE 操作符。当使用 ROLLUP 操作符时, 在生成原有统计结果的基础上, 还会生成横向小计结果(部门平均工资、所有雇员平均工资)。如下图所示:

岗位 部门号	CLERK	ANALYST	MANAGER	PRESIDENT	SALESMAN	小计
10	1300		2450	5000		2916
20	2050	3500	2975			2815
30	950		2850		1400	1566
合计						2301

当使用 CUBE 操作符时, 在原有 ROLLUP 统计结果的基础上, 还会生成纵向小计结果(岗位平均工资)。如下图所示:

岗位 部门号	CLERK	ANALYST	MANAGER	PRESIDENT	SALESMAN	小计
10	1300		2450	5000		2916
20	2050	3500	2975			2815
30	950		2850		1400	1566
	1587	3500	2758	5000	1400	
合计						2301

### 示例一：使用 ROLLUP 操作符

当直接使用 GROUP BY 子句进行多列分组时，只能生成简单的数据统计结果。为了生成数据统计以及横向小计统计，可以在 GROUP BY 子句中使用 ROLLUP 操作符。下面以显示每部门每岗位的平均工资、每部门的平均工资、所有雇员平均工资为例，说明使用 ROLLUP 操作符的方法。示例如下：

```
SQL> set pagesize 30
SQL> SELECT deptno,job,avg(sal) FROM emp
  2 GROUP BY ROLLUP (deptno,job);
DEPTNO JOB          AVG(SAL)
-----
10  CLERK        1300
10  MANAGER      2450
10  PRESIDENT    5000
10              2916.66667
20  CLERK        950
20  ANALYST     3000
20  MANAGER      2975
20              2175
30  CLERK        950
30  MANAGER      2850
30  SALESMAN    1400
30              1566.66667
                           2073.21429
```

已选择 13 行。

### 示例二：使用 CUBE 操作符

当直接使用 GROUP BY 子句进行多列分组时，只能生成简单的数据统计结果。为了生成数据统计以及横向小计统计，可以在 GROUP BY 子句中使用 ROLLUP 操作符。为了生成数据统计、横向小计、纵向小计结果，可以使用 CUBE 操作符。下面以显示每部门每岗位平均工资、部门平均工资、岗位平均工资、所有雇员平均工资为例，说明使用 CUBE 操作符的方法。示例如下：

```
SQL> SELECT deptno,job,avg(sal) FROM emp
  2 GROUP BY CUBE(deptno,job);
DEPTNO JOB          AVG(SAL)
-----
                           2073.21429
CLERK        1037.5
ANALYST      3000
MANAGER     2758.33333
SALESMAN    1400
PRESIDENT    5000
10          2916.66667
10  CLERK        1300
10  MANAGER      2450
10  PRESIDENT    5000
```

20	2175
20 CLERK	950
20 ANALYST	3000
20 MANAGER	2975
30	1566.66667
30 CLERK	950
30 MANAGER	2850
30 SALESMAN	1400

已选择 18 行。

### 示例三：使用 GROUPING 函数

GROUPING 函数用于确定统计结果是否用到了特定列。如果函数返回 0，则表示统计结果使用了该列；如果函数返回 1，则表示统计结果未使用该列。下面以使用 CUBE 操作符并确定统计结果所使用的列为例，说明使用 GROUPING 函数的方法。示例如下：

```
SQL> SELECT deptno, job, avg(sal), grouping(deptno), grouping(job)
  2 FROM emp GROUP BY cube (deptno, job);
    DEPTNO   JOB      AVG(SAL) GROUPING(DEPTNO) GROUPING(JOB)
```

		2073.21429	1	1
CLERK	1037.5	1	0	
ANALYST	3000	1	0	
MANAGER	2758.33333	1	0	
SALESMAN	1400	1	0	
PRESIDENT	5000	1	0	
10	2916.66667	0	1	
10 CLERK	1300	0	0	
10 MANAGER	2450	0	0	
10 PRESIDENT	5000	0	0	
20	2175	0	1	
20 CLERK	950	0	0	
20 ANALYST	3000	0	0	
20 MANAGER	2975	0	0	
30	1566.66667	0	1	
30 CLERK	950	0	0	
30 MANAGER	2850	0	0	
30 SALESMAN	1400	0	0	

已选择 18 行。

#### 4.4.4 GROUPING SETS

当使用 GROUP BY 子句执行数据分组统计时，默认情况下只会显示相应列的分组统计结果。在编写应用程序时，有些情况可能需要生成多种分组数据结果。在早期 Oracle 版本中，显示多个分组统计的结果，必须要编写多条数据分组语句来实现；从 Oracle9i 开始，使用 GROUPING SETS 操作符可以合并多个分组的结果。下面以示例说明 GROUPING SETS 操作符的作用及使用方法。

### 示例一：显示部门平均工资

当要显示每个部门的平均工资时，需要使用部门号（DEPTNO）执行分组统计操作。示例如下：

```
SQL> SELECT deptno,avg(sal) FROM emp GROUP BY deptno;
      DEPTNO    AVG(SAL)
----- -----
      10  2916.66667
      20      2175
      30  1566.66667
```

### 示例二：显示岗位平均工资

当显示每个岗位的平均工资时，需要使用岗位（JOB）执行分组统计操作。示例如下：

```
SQL> SELECT job,avg(sal) FROM emp GROUP BY job;
      JOB        AVG(SAL)
----- -----
ANALYST          3000
CLERK            1037.5
MANAGER          2758.33333
PRESIDENT        5000
SALESMAN         1400
```

### 示例三：显示部门平均工资和岗位平均工资

为了显示多个分组的统计结果，可以使用 GROUPING SETS 操作符合并分组统计结果。例如，如果既要显示部门的平均工资，也要显示岗位的平均工资，那么可以使用 GROUPING SETS 操作符合并分组结果。示例如下：

```
SQL> SELECT deptno,job,avg(sal) FROM emp
  2 GROUP BY GROUPING SETS(deptno,job);
      DEPTNO JOB        AVG(SAL)
----- -----
      10      2916.66667
      20      2175
      30      1566.66667
ANALYST          3000
CLERK            1037.5
MANAGER          2758.33333
PRESIDENT        5000
SALESMAN         1400
```

已选择 8 行。

## 4.5 连接查询

连接查询是指基于两个或两个以上表或视图的查询。在实际应用中，查询单个表可能无法满足应用程序的实际需求（例如显示 SALES 部门位置以及雇员名），在这种情况下就需要进行连接查询（DEPT 和 EMP 表）。在使用连接查询时，应注意以下事项：

- 当使用连接查询时，必须在 FROM 子句后指定两个或两个以上的表。

- 当使用连接查询时，应该在列名前加表名作为前缀。但是，如果不同表之间的列名不同，那么不需要在列名前加表名作为前缀；如果在不同表之间存在同名列，那么在列名之前必须要加表名作为前缀。否则会因为列的二义性而报错，如下所示：

```
SQL> SELECT deptno,dname,ename FROM dept,emp
   2 WHERE dept.deptno=emp.deptno;
SELECT deptno,dname,ename FROM dept,emp
   *
```

ERROR 位于第 1 行：

ORA-00918：未明确定义列

- 当使用连接查询时，必须在 WHERE 子句中指定有效的连接条件（在不同表的列之间进行连接）。如果不指定连接条件，或者指定了无效的连接条件，那么会导致生成笛卡儿集 ( $X \times Y$ )。如下所示：

```
SQL> SELECT dept.dname,emp.ename FROM dept,emp
   2 WHERE dept.dname='SALES';
```

DNAME	ENAME
SALES	SMITH
SALES	ALLEN
SALES	WARD
SALES	JONES
SALES	MARTIN
SALES	BLAKE
SALES	CLARK
SALES	SCOTT
SALES	KING
SALES	TURNER
SALES	ADAMS
SALES	JAMES
SALES	FORD
SALES	MILLER

已选择 14 行。

当进行连接查询时，使用表别名可以简化连接查询语句。当指定表别名时，别名应该跟在表名后面。不使用表别名和使用表别名的示例如下：

#### 示例一：不使用表别名

```
SQL> SELECT dept.dname,emp.ename FROM dept,emp
   2 WHERE dept.deptno=emp.deptno;
```

#### 示例二：使用表别名 (d、e 为别名)

```
SQL> SELECT d.dname,e.ename FROM dept d,emp e
   2 WHERE d.deptno=e.deptno;
```

### 4.5.1 相等连接

相等连接是指使用相等比较符 (=) 指定连接条件的连接查询，该种连接查询主要用于

检索主从表之间的相关数据。使用相等连接的语法如下：

```
SELECT table1.column, table2.column FROM table1,table2 WHERE
table1.column1 = table2.column2;
```

如上所示，当使用相等连接时，必须要使用等值比较符（=）指定连接条件。下面以示例说明使用相等连接的方法。

#### 示例一：使用相等连接执行主从查询

下面以显示所有雇员的名称、工资及其所在的部门名称为例，说明使用相等连接的方法。示例如下：

```
SQL> SELECT e.ename,e.sal,d.dname FROM emp e,dept d
  2 WHERE e.deptno=d.deptno;
    ENAME          SAL  DNAME
----- -----
SMITH           800  RESEARCH
ALLEN          1600  SALES
WARD            1250  SALES
JONES           2975  RESEARCH
MARTIN          1250  SALES
BLAKE           2850  SALES
CLARK           2450  ACCOUNTING
SCOTT           3000  RESEARCH
KING             5000  ACCOUNTING
TURNER          1500  SALES
ADAMS           1100  RESEARCH
JAMES            950  SALES
FORD             3000  RESEARCH
MILLER          1300  ACCOUNTING
```

已选择 14 行。

#### 示例二：使用 AND 指定其他条件

当使用相等连接时，会显示满足连接条件的所有数据。为了在执行相等连接的同时指定其他连接条件，可以在 WHERE 子句中使用 AND 操作符。下面以显示部门 10 的部门名、雇员名及其工资为例，说明使用 AND 操作符指定其他条件的方法。示例如下：

```
SQL> SELECT d.dname,e.ename,e.sal FROM emp e,dept d
  2 WHERE e.deptno=d.deptno AND d.deptno=10;
    DNAME        ENAME          SAL
----- -----
ACCOUNTING    CLARK           2450
ACCOUNTING    KING            5000
ACCOUNTING    MILLER          1300
```

### 4.5.2 不等连接

不等连接是指在连接条件下使用除相等比较符外的其他比较操作符的连接查询，并且不等连接主要用于在不同表之间显示特定范围的信息。SALGRADE 表存放着工资级别信息（例如 5 级工资范围为 3001~9999，4 级工资范围为 2001~3000 等等）。下面以显示所有雇员的

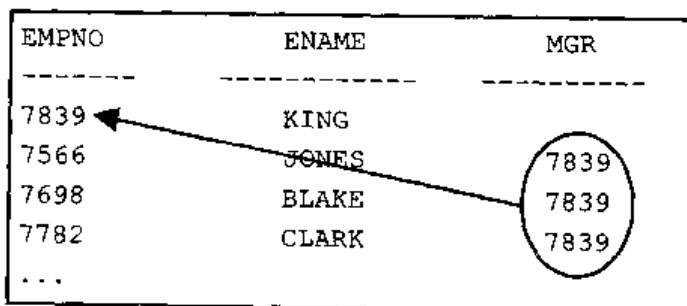
名称、工资及其工资级别为例，说明使用不等连接的方法。示例如下：

```
SQL> SELECT a.ename,a.sal,b.grade FROM emp a,salgrade b
  2 WHERE a.sal BETWEEN b.losal AND b.hisal;
    ENAME      SAL      GRADE
    -----  -----
SMITH        800       1
JAMES        950       1
ADAMS       1100       1
WARD         1250       2
MARTIN       1250       2
MILLER       1300       2
TURNER       1500       3
ALLEN        1600       3
CLARK        2450       4
BLAKE        2850       4
JONES        2975       4
SCOTT        3000       4
FORD         3000       4
KING         5000       5
```

已选择 14 行。

#### 4.5.3 自连接

自连接是指在同一张表之间的连接查询，它主要用在自参照表上显示上下级关系或者层次关系。自参照表是指在不同列之间具有参照关系或主从关系的表。例如，EMP 表包含有 EMPNO（雇员号）和 MGR（管理者号）列，二者之间就具有参照关系。如下所示：



根据 EMPNO 列和 MGR 列的对应关系，可以确定雇员 JONES, BLAKE 和 CLARK 的管理者为 KING。为了显示雇员及其管理者之间的对应关系，可以使用自连接。因为自连接是在同一张表之间的连接，所以必须要定义表别名。下面以显示 BLAKE 雇员的上级领导为例，说明使用自连接的方法。示例如下：

```
SQL> SELECT manager.ename FROM emp manager,emp worker
  2 WHERE manager.empno=worker.mgr
  3 AND worker.ename='BLAKE';
    ENAME
    -----
KING
```

#### 4.5.4 内连接和外连接

内连接用于返回满足连接条件的记录；而外连接则是内连接的扩展，它不仅会返回满足连接条件的所有记录，而且还会返回不满足连接条件的记录。在 Oracle9i 之前，连接语法都是在 WHERE 子句中指定的；从 Oracle9i 开始，还可以在 FROM 子句中指定连接语法。语法如下：

```
SELECT table1.column, table2.column
  FROM table1 [INNER | LEFT | RIGHT | FULL ] JOIN table2ON table1.column1
= table2.column2;
```

INNER JOIN 表示内连接；LEFT JOIN 表示左外连接；RIGHT JOIN 表示右外连接；FULL JOIN 表示完全外连接；ON 子句用于指定连接条件。注意，如果使用 FROM 子句指定内、外连接，则必须要使用 ON 子句指定连接条件；如果使用 (+) 操作符指定外连接，则必须使用 WHERE 子句指定连接条件。

##### 1. 内连接

内连接用于返回满足连接条件的所有记录。默认情况下，在执行连接查询时如果没有指定任何连接操作符，那么这些连接查询都属于内连接。下面以显示部门 10 的部门名及其雇员名为例，说明使用内连接的方法。示例如下：

```
SQL> SELECT a.dname,b.ename FROM dept a,emp b
  2 WHERE a.deptno=b.deptno AND a.deptno=10;
    DNAME      ENAME
  -----
ACCOUNTING    CLARK
ACCOUNTING    KING
ACCOUNTING    MILLER
```

另外，当执行连接查询时，通过在 FROM 子句中指定 INNER JOIN 选项，也可以指定内连接。示例如下：

```
SQL> SELECT a.dname,b.ename FROM dept a INNER JOIN emp b
  2 ON a.deptno=b.deptno AND a.deptno=10;
    DNAME      ENAME
  -----
ACCOUNTING    CLARK
ACCOUNTING    KING
ACCOUNTING    MILLER
```

从 Oracle9i 开始，如果主表的主键列和从表的外部键列名称相同，那么还可以使用 NATURAL JOIN 关键字自动执行内连接操作。示例如下：

```
SQL> SELECT dname,ename FROM dept NATURAL JOIN emp;
    DNAME      ENAME
  -----
RESEARCH     SMITH
SALES       ALLEN
SALES       WARD
RESEARCH     JONES
SALES       MARTIN
SALES       BLAKE
```

```

ACCOUNTING    CLARK
ACCOUNTING    KING
SALES         TURNER
SALES         JAMES
RESEARCH       FORD
ACCOUNTING    MILLER
RESEARCH       SCOTT
RESEARCH       ADAMS
已选择 14 行。

```

### 2. 左外连接

左外连接是通过指定 LEFT [OUTER] JOIN 选项来实现的。当使用左外连接时，不仅会返回满足连接条件的所有记录，而且还会返回不满足连接条件的连接操作符左别表的其他行。下面以显示部门 10 部门名、雇员名，以及其他部门名为例，说明使用左外连接的方法。示例如下：

```

SQL> SELECT a.dname,b.ename FROM dept a LEFT JOIN emp b
  2 ON a.deptno=b.deptno AND a.deptno=10;
DNAME        ENAME
-----
ACCOUNTING   CLARK
ACCOUNTING   KING
ACCOUNTING   MILLER
RESEARCH
SALES
OPERATIONS
已选择 6 行。

```

### 3. 右外连接

右外连接是通过指定 RIGHT [OUTER] JOIN 选项来实现的。当使用右外连接时，不仅会返回满足连接条件的所有行，而且还会返回不满足连接条件的连接操作符右别表的其他行。下面以显示部门 10 的部门名、雇员名以及其他雇员名为例，说明使用右外连接的方法。示例如下：

```

SQL> SELECT a.dname,b.ename FROM dept a RIGHT JOIN emp b
  2 ON a.deptno=b.deptno AND a.deptno=10;
DNAME        ENAME
-----
ACCOUNTING   MILLER
ACCOUNTING   KING
ACCOUNTING   CLARK
          JAMES
          TURNER
          BLAKE
          MARTIN
          WARD
          ALLEN
          FORD
          ADAMS

```

```
SCOTT
JONES
SMITH
```

已选择 14 行。

#### 4. 完全外连接

完全外连接是通过指定 FULL [OUTER] JOIN 选项来实现的。当使用完全外连接时，不仅会返回满足连接条件的所有行，而且还会返回不满足连接条件的所有其他行。下面以显示部门 10 的部门名、雇员名以及其他部门名和雇员名为例，说明使用完全外连接的方法。示例如下：

```
SQL> SELECT a.dname,b.ename FROM dept a FULL JOIN emp b
  2 ON a.deptno=b.deptno AND a.deptno=10;
    DNAME      ENAME
----- -----
ACCOUNTING    CLARK
ACCOUNTING    KING
ACCOUNTING    MILLER
RESEARCH
SALES
OPERATIONS
          SMITH
          ALLEN
          WARD
          JONES
          MARTIN
          BLAKE
          SCOTT
          TURNER
          ADAMS
          JAMES
          FORD
```

已选择 17 行。

#### 5. 使用 (+) 操作符

在 Oracle9i 之前，当执行外连接时，都是使用连接操作符 (+) 来完成的。尽管可以使用操作符 (+) 执行外连接操作，但 Oracle9i 开始 Oracle 建议使用 OUTER JOIN 执行外连接。使用 (+) 操作符执行外连接的语法如下：

```
SELECT table1.column, table2.column FROM table1, table2
  WHERE table1.column1(+) = table2.column2;
```

当使用 (+) 操作符执行外连接时，应该将该操作符放在显示较少行（完全满足连接条件行）的一端。当使用 (+) 操作符时，必须要注意以下事项：

- (+) 操作符只能出现在 WHERE 子句中，并且不能与 OUTER JOIN 语法同时使用。
- 当使用 (+) 操作符执行外连接时，如果在 WHERE 子句中包含有多个条件，则必须在所有条件中都包含 (+) 操作符。
- (+) 操作符只适用于列，而不能用在表达式上。
- (+) 操作符不能与 OR 和 IN 操作符一起使用。

- (+) 操作符只能用于实现左外连接和右外连接，而不能用于实现完全外连接。

### (1) 使用 (+) 操作符执行左外连接

当使用左外连接时，不仅会返回满足连接条件的所有行，而且还会返回不满足连接条件的左别表的其他行。因为 (+) 操作符要放在行数较少的一端，所以在 WHERE 子句中应该将该操作符放在右别表的一端。下面以显示部门 10 的部门名、雇员名以及其他部门名为例，说明使用 (+) 操作符执行左外连接的方法。示例如下：

```
SQL> SELECT a.dname,b.ename FROM dept a ,emp b
  2 WHERE a.deptno=b.deptno(+) AND b.deptno(+) =10;
    DNAME      ENAME
    -----
ACCOUNTING    CLARK
ACCOUNTING    KING
ACCOUNTING    MILLER
SALES
OPERATIONS
RESEARCH
已选择 6 行。
```

### (2) 使用 (+) 操作符执行右外连接

当使用右外连接时，不仅会返回满足连接条件的所有行，而且还会返回不满足连接条件的右别表的其他行。因为 (+) 操作符要放在行数较少的一端，所以在 WHERE 子句中应该将该操作符放在左别表的一端。下面以显示部门 10 的部门名、雇员名以及其他雇员名为例，说明使用 (+) 操作符执行右外连接的方法。示例如下：

```
SQL> SELECT a.dname,b.ename FROM dept a ,emp b
  2 WHERE a.deptno(+) =b.deptno AND a.deptno(+) =10
  3 ORDER BY a.dname;
    DNAME      ENAME
    -----
ACCOUNTING    MILLER
ACCOUNTING    KING
ACCOUNTING    CLARK
          JAMES
          TURNER
          BLAKE
          MARTIN
          WARD
          ALLEN
          FORD
          ADAMS
          SCOTT
          JONES
          SMITH
已选择 14 行。
```

## 4.6 子查询

子查询是指嵌入在其他 SQL 语句中的 SELECT 语句，也称为嵌套查询。注意，当在 DDL 语句中引用子查询时，可以带有 ORDER BY 子句；但是当在 WHERE 子句、SET 子句中引用子查询时，不能带有 ORDER BY 子句。子查询具有以下一些作用：

- 通过在 INSERT 或 CREATE TABLE 语句中使用子查询，可以将源表数据插入到目标表中。
- 通过在 CREATE VIEW 或 CREATE MATERIALIZED VIEW 中使用子查询，可以定义视图或实体化视图所对应的 SELECT 语句。
- 通过在 UPDATE 语句中使用子查询可以修改一列或多列数据。
- 通过在 WHERE, HAVING, START WITH 子句中使用子查询，可以提供条件值。

根据子查询返回结果的不同，子查询又被分为单行子查询、多行子查询和多列子查询。如下图所示：



### 4.6.1 单行子查询

单行子查询是指只返回一行数据的子查询语句。当在 WHERE 子句中引用单行子查询时，可以使用单行比较符 (=, >, <, >=, <=, <>)。下面以显示 SCOTT 同事的姓名、工资和部门号为例，说明单行子查询的使用方法。示例如下：

```
SQL> SELECT ename,sal,deptno FROM emp WHERE deptno=
  2 (SELECT deptno FROM emp WHERE ename='SCOTT');
    ENAME      SAL   DEPTNO
-----  -----
  SMITH       800     20
  JONES      2975     20
  SCOTT      3000     20
  ADAMS      1100     20
  FORD       3000     20
```

### 4.6.2 多行子查询

多行子查询是指返回多行数据的子查询语句。当在 WHERE 子句中使用多行子查询时，

必须要使用多行比较符 (IN, ALL, ANY)。它们的作用如下：

运算符	含义
IN	匹配于子查询结果的任一个值即可
ALL	必须要符合子查询结果的所有值
ANY	只要符合子查询结果的任一个值即可

注意，ALL 和 ANY 操作符不能单独使用，而只能与单行比较符 (=, >, <, >=, <=, <>) 结合使用。

### 1. 在多行子查询中使用 IN 操作符

当在多行子查询中使用 IN 操作符时，会处理匹配于子查询任一个值的行。下面以显示匹配于部门 10 岗位的雇员名、岗位、工资、部门号为例，说明在多行子查询中使用 IN 操作符的方法。示例如下：

```
SQL> SELECT ename,job,sal,deptno FROM emp WHERE job IN
  2 (SELECT distinct job FROM emp WHERE deptno=10);
ENAME      JOB          SAL      DEPTNO
-----  -----
CLARK     MANAGER      2450      10
BLAKE     MANAGER      2850      30
JONES     MANAGER      2975      20
KING      PRESIDENT    5000      10
MILLER    CLERK        1300      10
JAMES     CLERK        950       30
ADAMS     CLERK        1100      20
SMITH     CLERK        800       20
已选择 8 行。
```

### 2. 在多行子查询中使用 ALL 操作符

ALL 操作符必须与单行操作符结合使用，并且返回行必须要匹配于所有子查询结果。下面以显示高于部门 30 的所有雇员工资的雇员名、工资和部门号为例，说明在多行子查询中使用 ALL 操作符的方法。示例如下：

```
SQL> SELECT ename,sal,deptno FROM emp WHERE sal>all
  2 (SELECT sal FROM emp WHERE deptno=30);
ENAME      SAL      DEPTNO
-----  -----
JONES     2975      20
SCOTT     3000      20
KING      5000      10
FORD      3000      20
```

### 3. 在多行子查询中使用 ANY 操作符

ANY 操作符必须与单行操作符结合使用，并且返回行只需匹配于子查询的任一个结果即可。下面以显示高于部门 30 的任意雇员工资的雇员名、工资和部门号为例，说明在多行子查询中使用 ANY 操作符的方法。示例如下：

```
SQL> SELECT ename,sal,deptno FROM emp WHERE sal>ANY
```

```

2 (SELECT sal FROM emp WHERE deptno=30);
      ENAME      SAL     DEPTNO
      -----
      KING       5000      10
      SCOTT      3000      20
      FORD        3000      20
      JONES      2975      20
      BLAKE      2850      30
      CLARK      2450      10
      ALLEN      1600      30
      TURNER     1500      30
      MILLER     1300      10
      WARD        1250      30
      MARTIN     1250      30
      ADAMS       1100      20
      已选择 12 行。

```

### 4.6.3 多列子查询

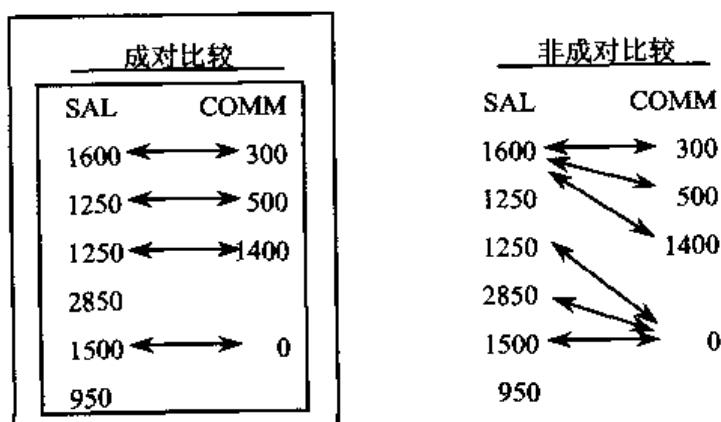
单行子查询是指子查询只返回单列单行数据，多行子查询是指子查询返回单列多行数据，二者都是针对单列而言的。而多列子查询则是指返回多列数据的子查询语句。当多列子查询返回单行数据时，在 WHERE 子句中可以使用单行比较符；当多列子查询返回多行数据时，在 WHERE 子句中必须使用多行比较符（IN, ANY, ALL）。下面以显示与 SMITH 部门和岗位完全相同的所有雇员为例，说明使用多列子查询的方法。示例如下：

```

SQL> SELECT ename,job,sal,deptno FROM emp WHERE (deptno,job)=
      2 (SELECT deptno,job FROM emp WHERE ename='SMITH');
      ENAME      JOB          SAL     DEPTNO
      -----
      SMITH      CLERK        800      20
      ADAMS      CLERK        1100     20

```

在使用子查询比较多个列的数据时，既可以使用成对比较，也可以使用非成对比较。其中，成对比较要求多个列的数据必须同时匹配，而非成对比较则不要求多个列的数据同时匹配。如下图所示：



如上图所示，当进行成对比较时，要求工资和补助必须同时匹配；而当执行非成对比较时，只要工资匹配于工资列表中的某一个、补助匹配于补助列表中的某一个就可以了。下面通过示例说明成对比较和非成对比较的区别。在执行子查询之前，请大家先执行以下语句更新雇员 CLARK 的工资和补助：

```
UPDATE emp SET sal=1500,comm=300 WHERE ename='CLARK';
```

### 1. 成对比较示例

当执行成对比较时，因为要求多个列的数据必须同时匹配，所以必须要使用多列子查询实现。下面以显示工资和补助与部门 30 雇员的工资和补助完全匹配的所有雇员为例，说明进行成对比较的方法。示例如下：

```
SQL> SELECT ename,sal,comm,deptno FROM emp
  2 WHERE (sal,nvl(comm,-1)) IN (SELECT sal,nvl(comm,-1)
  3   FROM emp WHERE deptno=30);
```

ENAME	SAL	COMM	DEPTNO
ALLEN	1600	300	30
WARD	1250	500	30
MARTIN	1250	1400	30
BLAKE	2850		30
TURNER	1500	0	30
JAMES	950		30

已选择 6 行。

### 2. 非成对比较示例

执行非成对比较时，应该要使用多个多行子查询来实现。下面以显示工资匹配于部门 30 工资列表、补助匹配于部门 30 补助列表的所有雇员为例，说明进行非成对比较的方法。示例如下：

```
SQL> SELECT ename,sal,comm,deptno FROM emp
  2 WHERE sal IN (SELECT sal FROM emp WHERE deptno=30)
  3   AND nvl(comm,-1) IN
  4     (SELECT nvl(comm,-1) FROM emp WHERE deptno=30);
```

ENAME	SAL	COMM	DEPTNO
ALLEN	1600	300	30
MARTIN	1250	1400	30
WARD	1250	500	30
BLAKE	2850		30
TURNER	1500	0	30
CLARK	1500	300	10
JAMES	950		30

已选择 7 行。

## 4.6.4 其他子查询

在 WHERE 子句中除了可以使用单行子查询、多行子查询以及多列子查询外，还可以使用相关子查询。另外在 FROM 子句、DML 语句、DDL 语句中也可以使用子查询。本小节介

绍如何在这些语句中使用子查询。

### 1. 相关子查询

相关子查询是指需要引用主查询表列的子查询语句，相关子查询是通过 EXISTS 谓词来实现的。下面以显示工作在“NEW YORK”的所有雇员为例，说明相关子查询的使用方法，示例如下：

```
SQL> SELECT ename, job, sal, deptno FROM emp WHERE EXISTS
  2 (SELECT 1 FROM dept WHERE dept.deptno=emp.deptno
  3 AND dept.loc='NEW YORK');
    ENAME      JOB          SAL      DEPTNO
  -----  -----
  MILLER    CLERK        1300       10
  KING      PRESIDENT    5000       10
  CLARK     MANAGER     2450       10
```

如上所示，当使用 EXISTS 谓词时，如果子查询存在返回结果，则条件为 TRUE；如果子查询没有返回结果，则条件为 FALSE。

### 2. 在 FROM 子句中使用子查询

当在 FROM 子句中使用子查询时，该子查询会被作为视图对待，因此也被称为内嵌视图。注意，当在 FROM 子句中使用子查询时，必须要给子查询指定别名。下面以显示高于部门平均工资的雇员信息为例，说明在 FROM 子句中使用子查询的方法。示例如下：

```
SQL> SELECT ename, job, sal FROM emp,
  2 (SELECT deptno, avg(sal) avgsal FROM emp
  3 GROUP BY deptno) dept
  4 WHERE emp.deptno=dept.deptno AND sal>dept.avgsal;
    ENAME      JOB          SAL
  -----  -----
  ALLEN    SALESMAN     1600
  JONES    MANAGER      2975
  BLAKE    MANAGER      2850
  SCOTT    ANALYST      3000
  KING     PRESIDENT    5000
  FORD     ANALYST      3000
已选择 6 行。
```

### 3. 在 DML 语句中使用子查询

子查询不仅适用于 SELECT 语句，也适用于任何 DML 语句。下面举例说明在 DML 语句中使用子查询的方法。

#### (1) 在 INSERT 语句中使用子查询

通过在 INSERT 语句中引用子查询，可以将一张表的数据装载到另一张表中。下面以将 EMP 表的数据装载到 EMPLOYEE 表中为例，说明在 INSERT 语句中使用子查询的方法。示例如下：

```
SQL> INSERT INTO employee (id, name, title, salary)
  2 SELECT empno, ename, job, sal FROM emp;
已创建 14 行。
```

#### (2) 在 UPDATE 语句中使用子查询

当在 UPDATE 语句中使用子查询时，既可以在 WHERE 子句中引用子查询（返回未知条件值），也可以在 SET 子句中使用子查询（修改列数据）。下面以将 SMITH 同岗位的雇员工资和补助更新为与 SMITH 的工资和补助完全相同为例，说明在 UPDATE 语句中使用子查询的方法。示例如下：

```
SQL> UPDATE emp SET (sal,comm)=
  2 (SELECT sal,comm FROM emp WHERE ename='SMITH')
  3 WHERE job=(SELECT job FROM emp WHERE ename='SMITH');
已更新 4 行。
```

### (3) 在 DELETE 语句中使用子查询

在 DELETE 语句中使用子查询时，可以在 WHERE 子句中引用子查询返回未知条件值。下面以删除 SALES 部门的所有雇员为例，说明在 DELETE 的 WHERE 子句中使用子查询的方法。示例如下：

```
SQL> DELETE FROM emp WHERE deptno=
  2 (SELECT deptno FROM dept WHERE dname='SALES');
已删除 6 行。
```

## 4. 在 DDL 语句中使用子查询

除了可以在 SELECT, INSERT, UPDATE、DELETE 语句中使用子查询外，也可以在 DDL 语句中使用子查询。注意，当在 SELECT 和 DML 语句中使用子查询时，WHERE 子句和 SET 子句的子查询语句不能包含 ORDER BY 子句；但在 DDL 语句中使用子查询时，子查询可以包含 ORDER BY 子句。下面通过示例说明在 DDL 语句中使用子查询的方法。

### (1) 在 CREATE TABLE 语句中使用子查询

通过在 CREATE TABLE 中使用子查询，可以在建立新表的同时复制表中的数据。下面以建立 new\_emp 表，并将 EMP 表的数据复制到该表为例，说明在 CREATE TABLE 语句中使用子查询的方法。示例如下：

```
SQL> CREATE TABLE new_emp(id,name,sal,job,deptno) AS
  2 SELECT empno,ename,sal,job,deptno FROM emp;
```

### (2) 在 CREATE VIEW 语句中使用子查询

建立视图时，必须指定视图所对应的子查询语句。下面以建立视图 DEPT\_10，说明在 CREATE VIEW 语句中使用子查询的方法。示例如下：

```
SQL> CREATE OR REPLACE VIEW dept_10 AS
  2 SELECT empno,ename,job,sal,deptno FROM emp
  3 WHERE deptno=10 ORDER BY empno;
```

### (3) 在 CREATE MATERIALIZED VIEW 语句中使用子查询

建立实体化视图时，必须要指定实体化视图所对应的 SQL 语句，并且该 SQL 语句将来可以用于查询重写。下面以建立实体化视图 summary\_emp 为例，说明在 CREATE MATERIALIZED VIEW 语句中使用子查询的方法。示例如下：

```
SQL> CREATE MATERIALIZED VIEW summary_emp AS
  2 SELECT deptno,job,avg(sal) avgsal,sum(sal) sumsal
  3 FROM emp GROUP BY cube(deptno,job);
```

## 4.7 合并查询

为了合并多个 SELECT 语句的结果，可以使用集合操作符 UNION, UNION ALL, INTERSECT 和 MINUS。语法如下：

```
SELECT 语句 1 [UNION | UNION ALL | INTERSECT | MINUS] SELECT 语句 2
```

...

这些集合操作符具有相同的优先级，当同时使用多个操作符时，会按照从左至右的方式引用这些集合操作符。当使用集合操作符时，必须确保不同查询的列个数和数据类型都要匹配。另外，使用集合操作符有以下一些限制：

- 对于 LOB, VARRAY 和嵌套表列来说，集合操作符是无效的。
- 对于 LONG 列来说，UNION, INTERSECT, MINUS 操作符是无效的。
- 如果选择列表包含了表达式，则必须要为其指定列别名。

### 1. UNION

UNION 操作符用于获取两个结果集的并集。当使用该操作符时，会自动去掉结果集中的重复行，并且会以第一列的结果进行排序。下面以显示工资高于 2500 的雇员和岗位为“MANAGER”的雇员为例，说明使用该操作符的方法。示例如下：

```
SQL> SELECT ename,sal,job FROM emp WHERE sal>2500
      2 UNION
      3 SELECT ename,sal,job FROM emp WHERE job='MANAGER';
    ENAME          SAL   JOB
    -----  -----
    BLAKE        2850  MANAGER
    CLARK        2450  MANAGER
    FORD         3000  ANALYST
    JONES        2975  MANAGER
    KING         5000  PRESIDENT
    SCOTT        4000  ANALYST
    SMITH        3000  CLERK
已选择 7 行。
```

### 2. UNION ALL

UNION ALL 操作符用于获取两个结果集的并集。但与 UNION 操作符不同，该操作符不会取消重复值，而且也不会以任何列进行排序。下面以显示工资高于 2500 的雇员和岗位为“MANAGER”的雇员为例，说明使用该操作符的方法。示例如下：

```
SQL> SELECT ename,sal,job FROM emp WHERE sal>2500
      2 UNION ALL
      3 SELECT ename,sal,job FROM emp WHERE job='MANAGER';
    ENAME          SAL   JOB
    -----  -----
    SMITH        3000  CLERK
    JONES        2975  MANAGER
    BLAKE        2850  MANAGER
    SCOTT        4000  ANALYST
```

```

KING           5000 PRESIDENT
FORD          3000 ANALYST
JONES         2975  MANAGER
BLAKE         2850  MANAGER
CLARK         2450  MANAGER
已选择 9 行。

```

### 3. INTERSECT

INTERSECT 操作符用于获取两个结果集的交集。当使用该操作符时，只会显示同时存在于两个结果集中的数据，并且会以第一列进行排序。下面以显示工资高于 2500 并且岗位为“MANAGER”的雇员为例，说明使用该操作符的方法。示例如下：

```

SQL> SELECT ename,sal,job FROM emp WHERE sal>2500
2 INTERSECT
3 SELECT ename,sal,job FROM emp WHERE job='MANAGER';
ENAME      SAL JOB
-----
BLAKE     2850 MANAGER
JONES     2975 MANAGER

```

### 4. MINUS

MINUS 操作符用于获取两个结果集的差集。当使用该操作符时，只会显示在第一个结果集中存在，在第二个结果集中不存在的数据，并且会以第一列进行排序。下面以显示工资高于 2500 但岗位不是“MANAGER”的雇员为例，说明使用该操作符的方法。示例如下：

```

SQL> SELECT ename,sal,job FROM emp WHERE sal>2500
2 MINUS
3 SELECT ename,sal,job FROM emp WHERE job='MANAGER';
ENAME      SAL JOB
-----
FORD     3000 ANALYST
KING    5000 PRESIDENT
SCOTT   4000 ANALYST
SMITH   3000 CLERK

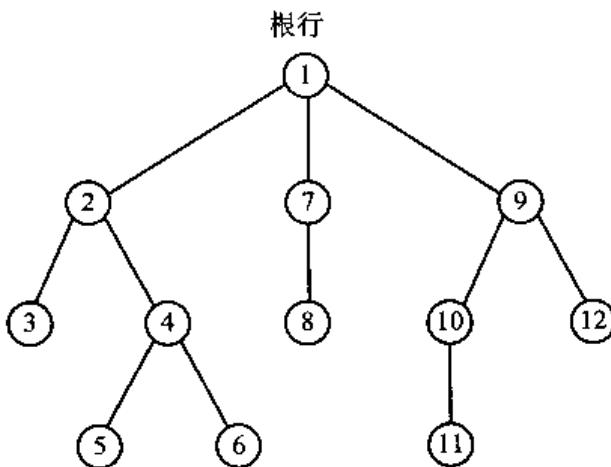
```

## 4.8 其他复杂查询

在实际应用中，除了经常会用到数据分组、连接查询、子查询以及合并查询之外，在特定情况下还可以使用其他复杂查询语句，包括层次查询、倒叙查询等等。本节介绍这些复杂查询的作用及使用方法。

### 1. 层次查询

当表具有层次结构数据时，通过使用层次查询可以更直观的显示数据结果，并显示其数据之间的层次关系。如下图所示：



如上图所示，假定某层次查询返回 12 行，则根行（第 1 行）应该是层次最高的行；而第 2, 7, 9 行是根行的下一级行，它们具有相同层次；第 3、4 行（相同层次）为第 2 行的下一级行，以此类推。在层次查询中，伪列 LEVEL 可以用于返回层次，根行层次为 1，第二级行层次为 2，以此类推。层次查询子句的语法如下：



- **START WITH:** 用于指定层次查询的根行。
- **CONNECT BY:** 用于指定父行和子行之间的关系。在 condition 表达式中，必须使用 PRIOR 引用父行。语法如下：

... PRIOR expr = expr 或 ... expr = PRIOR expr

EMP 表是具有层次结构数据的表，其中 EMPNO 列对应于雇员号，而 MGR 列对应于管理者编号。通过使用层次查询，可以显示雇员之间的上下级关系。下面以显示除“CLERK”外所有其他雇员的上下级关系为例，说明使用层次查询的方法。示例如下：

```

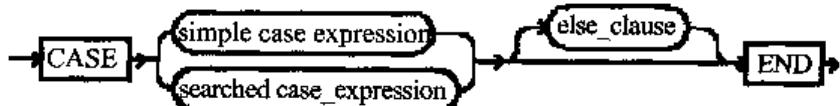
SQL> col ename format a15
SQL> col job format a15
SQL> SELECT LPAD(' ',3*(LEVEL-1))||ename ename,
2   LPAD(' ',3*(LEVEL-1))||job job FROM emp
3 WHERE job<>'CLERK' START WITH mgr IS NULL
4 CONNECT BY mgr=PRIOR empno;
ENAME          JOB
-----
KING          PRESIDENT
JONES         MANAGER
SCOTT        ANALYST
FORD          ANALYST
BLAKE        MANAGER
ALLEN        SALESMAN
WARD          SALESMAN
MARTIN       SALESMAN
TURNER       SALESMAN

```

```
CLARK          MANAGER
已选择 10 行。
```

## 2. 使用 CASE 表达式

为了在 SQL 语句中使用 IF...THEN...ELSE 语法，可以使用 CASE 表达式。通过 CASE 表达式，可以避免调用过程来完成条件分支操作。CASE 表达式的语法如下：



当使用 CASE 表达式时，使用 WHEN 子句可以指定条件语句。下面以显示雇员名、工资及工资级别为例，说明使用 CASE 表达式的方法。示例如下：

```
SQL> SELECT ename,sal,CASE WHEN sal>3000 THEN 3
  2   WHEN sal>2000 THEN 2 ELSE 1 END grade
  3 FROM emp WHERE deptno=10;
ENAME          SAL      GRADE
-----
CLARK          2450     2
KING           5000     3
MILLER         1300     1
```

## 3. 倒叙查询

默认情况下，当执行查询操作时，只能看到最近提交的数据。从 Oracle9i 开始，通过使用倒叙查询（Flashback Query）特征，可以查看到过去某时间点所提交的数据。注意，如果使用倒叙查询，那么要求数据库必须采用 UNDO 管理方式，并且初始化参数 undo\_retention 限制了 UNDO 数据的保留时间。下面举例说明查询当前数据，以及历史数据的方法。

### (1) 查看当前数据

默认情况下，执行查询操作只能查看到最近所提交的数据。示例如下：

```
SQL> SELECT ename,sal FROM emp WHERE ename='CLARK';
ENAME          SAL
-----
CLARK          2695
```

### (2) 查看历史数据

执行倒叙查询时，通过在 FROM 子句后指定 AS OF 子句可以查看过去的历史数据。在 AS OF 子句中既可以指定时间，也可以指定 SCN。注意，使用倒叙查询只能看到 5 分钟之前变化的数据，而不能查看到 5 分钟之内变化的数据。示例如下：

```
SQL> SELECT ename,sal FROM emp AS OF TIMESTAMP to_timestamp(
  2   '2003-05-18 19:59:00','YYYY-MM-DD HH24:MI:SS')
  3 WHERE ename='CLARK';
ENAME          SAL
-----
CLARK          2450
```

### (3) 使用 DBMS\_FLASHBACK 包获取特定 SCN 的数据

```
SQL> exec dbms_flashback.enable_at_system_change_number(717402)
PL/SQL 过程已成功完成。
```

```

SQL> SELECT sal FROM emp WHERE ename='SCOTT';
      SAL
-----
      3630
SQL> exec dbms_flashback.disable
PL/SQL 过程已成功完成。
SQL> SELECT sal FROM emp WHERE ename='SCOTT';
      SAL
-----
      3000

```

#### 4. 使用 WITH 子句重用子查询

对于多次使用相同子查询的复杂查询语句来说，用户可能会将查询语句分成两条语句执行。第一条语句将子查询结果存放到临时表，第二条查询语句使用临时表处理数据。从 Oracle9i 开始，通过 WITH 子句可以给子查询指定一个名称，并且使得在一条语句中可以完成所有任务，从而避免了使用临时表。

**示例一：显示部门工资总和高于雇员工资总和三分之一的部门名及工资总和（两次使用相同子查询）**

```

SQL> SELECT dname, SUM(sal) AS dept_total FROM emp, dept
  2 WHERE emp.deptno = dept.deptno GROUP BY dname
  3 HAVING SUM(sal) >
  4 (SELECT SUM(sal) * 1/3 FROM emp, dept
  5 WHERE emp.deptno = dept.deptno
  6 );
DNAMEN      DEPT_TOTAL
-----
RESEARCH      14075

```

**示例二：显示部门工资总和高于雇员工资总和三分之一的部门名及工资总和（使用 WITH 子句重用子查询）**

```

SQL> WITH summary AS (
  2   SELECT dname, SUM(sal) AS dept_total FROM emp, dept
  3   WHERE emp.deptno = dept.deptno GROUP BY dname
  4 )
  5 SELECT dname, dept_total FROM summary WHERE dept_total >
  6   ( SELECT SUM(dept_total) * 1/3 FROM summary);
DNAMEN      DEPT_TOTAL
-----
RESEARCH      10875

```

## 4.9 习题

1. 使用简单查询语句显示：

- (1) 所有部门名称；
- (2) 所有雇员名及其全年收入（工资+补助），并指定列别名“年收入”；

(3) 存在雇员的所有部门号;

2. 限制查询数据显示:

(1) 工资超过 2850 的雇员姓名和工资;

(2) 工资不在 1500 到 2850 之间的所有雇员名及工资;

(3) 代码为 7566 的雇员姓名及所在部门代码;

(4) 部门 10 和 30 中工资超过 1500 的雇员名及工资;

(5) 无管理者的雇佣名及岗位。

3. 使用排序数据的方法显示

(1) 在 1981 年 2 月 1 日~1981 年 5 月 1 日之间雇佣的雇员名、岗位及雇佣日期，并以雇佣日期的先后进行排序。

(2) 获得补助的所有雇员名、工资及补助额，并以工资和补助的降序排序。

4. 为 DEPT 表插入一条数据，包括:

● 部门号: 50

● 部门名称: ADMINISTRATOR

● 部门位置: BOSTON

5. 为 EMP 表插入一条数据，包括:

● 雇员号: 1587

● 雇员名: JOHN

● 工资: 1000

● 雇佣日期: 1987-03-05

● 部门号: 30

6. 给部门 10 的每个雇员增加 10% 的工资。

7. 删除部门 50。

8. 提交事务。

9. 使用分组函数和数据分组子句显示

(1) 所有雇员的平均工资、总计工资、最高工资、最低工资。

(2) 每种岗位的雇员总数、平均工资。

(3) 雇员总数，以及获得补助的雇员数。

(4) 管理者的总人数。

(5) 雇员工资的最大差额。

(6) 每个部门每个岗位的平均工资、每个部门的平均工资、每个岗位的平均工资。

10. 使用连接查询方法显示

(1) 部门 20 的部门名，以及该部门的所有雇员名、雇员工资及岗位。

(2) 获得补助的所有雇员名、补助额以及所在部门名。

(3) 在“DALLAS”工作的所有雇员名、雇员工资及所在部门名。

(4) 雇员 SCOTT 的管理者名。

(5) 查询 EMP 表和 SALGRADE 表，显示部门 20 的雇员名、工资及其工资级别。

(6) 部门 10 的所有雇员名、部门名，以及其他部门名。

(7) 部门 10 的所有雇员名、部门名，以及其他雇员名。

(8) 部门 10 的所有雇员名、部门名，以及其他部门名和雇员名。

11. 使用子查询，显示：

- (1) BLAKE 同部门的所有雇员，但不显示 BLAKE。
- (2) 超过平均工资的所有雇员名、工资及部门号。
- (3) 超过部门平均工资的所有雇员名、工资及部门号。
- (4) 高于 CLERK 岗位所有雇员工资的所有雇员名、工资及岗位。
- (5) 工资、补助额与 SCOTT 完全一致的所有雇员名、工资及补助额。

12. 使用集合操作符

(1) 执行如下语句建立视图：

```
CREATE VIEW dept_20 AS SELECT * FROM emp WHERE deptno=20;
CREATE VIEW job_clerk AS SELECT * FROM emp WHERE job='CLERK';
```

(2) 使用视图 dept\_20 和 job\_clerk 取得部门 20 或岗位为 CLERK 的所有雇员名、工资（不显示重复值）。

(3) 使用视图 dept\_20 和 job\_clerk 取得部门 20 或岗位为 CLERK 的所有雇员名、工资（显示重复值）。

(4) 使用视图 dept\_20 和 job\_clerk 取得部门 20 并且岗位为 CLERK 的所有雇员名和工资。

(5) 使用视图 dept\_20 和 job\_clerk 取得部门 20 但岗位不是 CLERK 的所有雇员名和工资。

13. 使用层次查询取得职位最高的雇员及其下一级雇员的雇员名和岗位。

## 第5章 SQL 函数

SQL 函数包括单行函数和多行函数，其中单行函数是指输入一行输出也是一行的函数；多行函数也被称为分组函数，它会根据输入的多行数据输出一个结果。SQL 函数不仅可以在 SQL 语句中引用，也可以在 PL/SQL 块内引用。大多数单行函数都可以直接在 PL/SQL 块内引用，但多行函数不能由 PL/SQL 块直接引用，而只能在 PL/SQL 块的内嵌 SQL 语句中引用。本章将详细介绍 Oracle 所提供的 SQL 函数，这些函数包含了许多 Oracle 10g 和 Oracle9i 新增加的函数。在学习了本章之后，读者应该：

- 在 SQL 语句和 PL/SQL 块中使用数字函数；
- 在 SQL 语句和 PL/SQL 语句中使用字符函数；
- 在 SQL 语句和 PL/SQL 语句中使用日期函数；
- 在 SQL 语句和 PL/SQL 语句中使用转换函数；
- 在 SQL 语句和 PL/SQL 语句中使用 Oracle 10g 新增加的集合函数；
- 在 SQL 语句中使用分组函数；
- 了解并掌握 Oracle 10g、Oracle9i 新增加的 SQL 函数。

### 5.1 数字函数

数字函数的输入参数和返回值都是数字型，并且多数函数精确到 38 位。函数 COS、COSH、EXP、LN、LOG、SIN、SINH、SQRT、TAN 和 TANH 精确到 36 位，而函数 ACOS、ASIN、ATAN 和 ATAN2 则精确到 30 位。这些数字函数不仅可以在 SQL 语句中引用，也可以直接在 PL/SQL 块中引用。下面详细介绍 Oracle 所提供的各种数字函数。

- ABS(n)：该函数用于返回数字 n 的绝对值。示例如下：

```
SQL> DECLARE
  2    v_abs NUMBER(6,2);
  3  BEGIN
  4    v_abs:=abs(&no);
  5    dbms_output.put_line('绝对值:'||v_abs);
  6  END;
  7 /
```

输入 no 的值： -12.3

绝对值:12.3

- ACOS(n)：该函数用于返回数字 n 的反余弦值，输入值的范围是-1~1，输出值的单位为弧度。示例如下：

```
SQL> SELECT acos(.3),acos(-.3) FROM dual;
      ACOS (.3)  ACOS (-.3)
----- -----
      1.26610367 1.87548898
```

- ASIN(n): 该函数用于返回数字 n 的反正弦值，输入值的范围是-1~1，输出值的单位为弧度。示例如下：

```
SQL> DECLARE
  2    v_asin NUMBER(6,2);
  3  BEGIN
  4    v_asin:=asin(0.8);
  5    dbms_output.put_line('0.8 的反正弦值:'||v_asin);
  6  END;
  7 /
0.8 的反正弦值:.93
```

- ATAN(n): 该函数用于返回数字 n 的反正切值，输入值可以是任何数字，输出值的单位为弧度。示例如下：

```
SQL> SELECT atan(10.3),atan(-20.3) FROM dual;
ATAN(10.3) ATAN(-20.3)
-----
1.47401228 -1.521575
```

- ATAN2(n,m): 该函数用于返回数字 n 除以数字 m 的反正切值。输入值除了 m 不能为 0 以外，可以是任意数字（m 不能为 0），输出值的单位为弧度。示例如下：

```
SQL> DECLARE
  2    v_atan2 NUMBER(6,2);
  3  BEGIN
  4    v_atan2:=atan2(19,3);
  5    dbms_output.put_line('19/3 的反正切值:'||v_atan2);
  6  END;
  7 /
19/3 的反正切值:1.41
```

- CEIL(n): 该函数用于返回大于等于数字 n 的最小整数。示例如下：

```
SQL> SELECT ceil(15),ceil(15.1) FROM dual;
CEIL(15) CEIL(15.1)
-----
15      16
```

- COS(n): 该函数用于返回数字 n（以弧度表示的角度值）的余弦值。

```
SQL> DECLARE
  2    v_cos NUMBER(6,2);
  3  BEGIN
  4    v_cos:=cos(0.5);
  5    dbms_output.put_line('0.5 的余弦值:'||v_cos);
  6  END;
  7 /
0.5 的余弦值:.88
```

- COSH(n): 该函数用于返回数字 n 的双曲余弦值。示例如下：

```
SQL> SELECT COSH(0) "0 的双曲余弦值" FROM DUAL;
0 的双曲余弦值
-----
```

- EXP(n): 该函数用于返回 e 的 n 次幂 ( $e = 2.71828183 \dots$ )。示例如下:

```
SQL> DECLARE
  2   v_exp NUMBER(6,2);
  3 BEGIN
  4   v_exp:=exp(4);
  5   dbms_output.put_line('e 的 4 次幂:'||v_exp);
  6 END;
  7 /
e 的 4 次幂:54.6
```

- FLOOR(n): 该函数用于返回小于等于数字 n 的最大整数。示例如下:

```
SQL> SELECT floor(15),floor(15.1) FROM dual;
FLOOR(15) FLOOR(15.1)
-----
          15           15
```

- LN(n): 该函数用于返回数字 n 的自然对数，其中数字 n 必须大于 0。示例如下:

```
SQL> DECLARE
  2   v_ln NUMBER(6,2);
  3 BEGIN
  4   v_ln:=ln(4);
  5   dbms_output.put_line('4 的自然对数:'||v_ln);
  6 END;
  7 /
4 的自然对数:1.39
```

- LOG(m,n): 该函数用于返回以数字 m 为底的数字 n 的对数，数字 m 可以是除 0 和 1 以外的任何正整数，数字 n 可以是任何正整数。示例如下:

```
SQL> SELECT log(2,8),log(10,100) FROM dual;
LOG(2,8) LOG(10,100)
-----
            3             2
```

- MOD(m,n): 该函数用于取得两个数字相除后的余数。如果数字 n 为 0，则返回结果为 m。示例如下:

```
SQL> DECLARE
  2   v_mod NUMBER(6,2);
  3 BEGIN
  4   v_mod:=mod(10,3);
  5   dbms_output.put_line('10 除 3 的余数:'||v_mod);
  6 END;
  7 /
10 除 3 的余数:1
```

- POWER(m,n): 该函数用于返回数字 m 的 n 次幂，底数 m 和指数 n 可以是任意数字。但如果数字 m 为负数，则数字 n 必须是正数。示例如下:

```
SQL> SELECT power(-2,3),power(2,-1) FROM dual;
POWER(-2,3) POWER(2,-1)
-----
```

- ROUND(n,[m]): 该函数用于执行四舍五入运算; 如果省略 m, 则四舍五入至整数位; 如果 m 是负数, 则四舍五入到小数点前 m 位; 如果 m 是正数, 则四舍五入至小数点后 m 位。示例如下:

```
SQL> DECLARE
  2   v_round NUMBER(6,2);
  3   BEGIN
  4     v_round:=round(&no,1);
  5     dbms_output.put_line('四舍五入到小数点后一位:'||v_round);
  6   END;
  7 /
```

输入 no 的值: 65.698

四舍五入到小数点后一位:65.7

- SIGN(n): 该函数用于检测数字的正负。如果数字 n 小于 0, 则函数的返回值为 -1; 如果数字 n 等于 0, 则函数的返回值为 0; 如果数字 n 大于 0, 则函数的返回值为 1。示例如下:

```
SQL> SELECT sign(-10),sign(0),sign(20) FROM dual;
      SIGN(-10)    SIGN(0)    SIGN(20)
----- ----- -----
          -1          0          1
```

- SIN(n): 该函数用于返回数字 n (以弧度表示的角) 的正弦值。示例如下:

```
SQL> DECLARE
  2   v_sin NUMBER(6,2);
  3   BEGIN
  4     v_sin:=sin(0.3);
  5     dbms_output.put_line('0.3 的正弦值:'||v_sin);
  6   END;
  7 /
```

0.3 的正弦值:.3

- SINH(n): 该函数用于返回数字 n 的双曲正弦值。示例如下:

```
SQL> SELECT SINH(.5) FROM dual;
      SINH(.5)
-----
      .521095305
```

- SQRT(n): 该函数用于返回数字 n 的平方根, 并且数字 n 必须大于等于 0。示例如下:

```
SQL> DECLARE
  2   v_sqrt NUMBER(6,2);
  3   BEGIN
  4     v_sqrt:=sqrt(10);
  5     dbms_output.put_line('10 的平方根:'||v_sqrt);
  6   END;
  7 /
```

10 的平方根:3.16

- TAN(n): 该函数用于返回数字 n (以弧度表示的角) 的正切值。示例如下:

```
SQL> SELECT TAN(45 * 3.14159265359/180) FROM dual;
```

```
TAN(45*3.14159265359/180)
-----
```

```
1
```

- **TANH(n):** 该函数用于返回数字 n 的双曲正切值。示例如下：

```
SQL> DECLARE
  2   v_tanh NUMBER(6,2);
  3   BEGIN
  4     v_tanh:=tanh(10);
  5     dbms_output.put_line('10 的双曲正切值:'||v_tanh);
  6   END;
  7 /
10 的双曲正切值:1
```

- **TRUNC(n,[m]):** 该函数用于截取数字。如果省略数字 m，则将数字 n 的小数部分截去；如果数字 m 是正数，则将数字 n 截取至小数点后的第 m 位；如果数字 m 是负数，则将数字 n 截取至小数点的前 m 位。示例如下：

```
SQL> SELECT trunc(45.926),trunc(45.926,1),trunc(45.926,-1)
  2   FROM dual;
TRUNC(45.926) TRUNC(45.926,1) TRUNC(45.926,-1)
-----
45           45.9          40
```

## 5.2 字符函数

字符函数的输入参数为字符类型，其返回值是字符类型或数字类型。字符函数既可以在 SQL 语句中使用，也可以直接在 PL/SQL 块中引用。下面详细介绍 Oracle 所提供的字符函数，以及在 SQL 语句和 PL/SQL 块中使用这些字符函数的方法。

- **ASCII(char):** 该函数用于返回字符串首字符的 ASCII 码值，示例如下：

```
SQL> SELECT ascii('a') "a",ascii('A') "A" FROM dual;
      a        A
-----
      97       65
```

- **CHR(n):** 该函数用于将 ASCII 码值转变为字符。示例如下：

```
SQL> DECLARE
  2   v_chr VARCHAR2(10);
  3   BEGIN
  4     v_chr:=chr(56);
  5     dbms_output.put_line('ASCII 码为 56 的字符:'||v_chr);
  6   END;
  7 /
ASCII 码为 56 的字符:8
```

- **CONCAT:** 该函数用于连接字符串，其作用与连接操作符 (||) 完全相同。示例如下：

```
SQL> SELECT concat('Good',' Morning') FROM dual;
CONCAT('GOOD
```

Good Morning

- INITCAP(char): 该函数用于将字符串中每个单词的首字符大写，其他字符小写，单词之间用空格和非字母字符分隔。示例如下：

```
SQL> DECLARE
  2    v_initcap VARCHAR2(10);
  3  BEGIN
  4    v_initcap:=initcap('my word');
  5    dbms_output.put_line('首字符大写:'||v_initcap);
  6  END;
  7 /
首字符大写:My Word
```

- INSTR(char1,char2 [,n[,m]]): 该函数用于取得子串在字符串中的位置，其中数字 n 为起始搜索位置，数字 m 为子串出现次数。如果数字 n 为负数，则从尾部开始搜索；数字 m 必须为正整数，并且 n 和 m 的默认值为 1。示例如下：

```
SQL> SELECT instr('morning','n') FROM dual;
INSTR('MORNING','N')
-----
4
```

- LENGTH(char): 该函数用于返回字符串的长度。如果字符串的类型为 CHAR，则其长度包括所有的后缀空格；如果 char 是 null，则返回 null。示例如下：

```
SQL> DECLARE
  2    v_len INT;
  3  BEGIN
  4    v_len:=length('my word');
  5    dbms_output.put_line('字符串长度:'||v_len);
  6  END;
  7 /
字符串长度:7
```

- LOWER(char): 该函数用于将字符串转换为小写格式。示例如下：

```
SQL> SELECT lower('SQL introduction') FROM dual;
LOWER('SQLINTROD
-----
sql introduction
```

- LPAD(char1,n,char2): 该函数用于在字符串 char1 的左端填充字符串 char2，直至字符串总长度为 n，char2 的默认值为空格。如果 char1 长度大于 n，则该函数返回 char1 左端的 n 个字符。示例如下：

```
SQL> DECLARE
  2    v_lpad VARCHAR2(10);
  3  BEGIN
  4    v_lpad:=lpad('aaaa',10,'*');
  5    dbms_output.put_line('在字符串左端添加字符*:'||v_lpad);
  6  END;
  7 /
在字符串左端添加字符*:*****aaaa
```

- **LTRIM(char1[,set]):** 该函数用于去掉字符串 char1 左端所包含的 set 中的任何字符。Oracle 从左端第一个字符开始扫描，逐一去掉在 set 中出现的字符，当遇到不是 set 中的字符时终止，然后返回剩余结果。示例如下：

```
SQL> SELECT ltrim('morning','m'),ltrim('morning','or')
  2  FROM dual;
LTrim( LTrim(
-----
orning morning
```

- **NLS\_INITCAP(char,'nls\_param'):** 该函数用于将字符串 char 的首字符大写，其他字符小写，其中 char 用于指定 NCHAR 或 NVARCHAR2 类型字符串，其前面加上 n，用单引号括起来，nls\_param 的格式为“nls\_sort=sort”，用于指定特定语言特征。示例如下：

```
SQL> DECLARE
  2   v_nls_initcap NCHAR(10);
  3  BEGIN
  4    v_nls_initcap:=nls_initcap(n'my word');
  5    dbms_output.put_line('首字符大写:'||v_nls_initcap);
  6  END;
  7 /
首字符大写:My Word
```

- **NLS\_LOWER(char,'nls\_param'):** 该函数用于将字符串转变为小写，其中 nls\_param 的格式为“nls\_sort=sort”，用于指定特定语言特征。示例如下：

```
SQL> SELECT nls_lower(n'SQL') FROM dual;
NLS_LO
-----
sql
```

- **NLS\_SORT(char,'nls\_param'):** 该函数用于按照特定语言的要求进行排序，其中 nls\_param 的格式为“nls\_sort=sort”，用于指定特定语言特征。示例如下：

```
SQL> SELECT * FROM test
  2 ORDER BY NLSSORT(name, 'NLS_SORT = XDanish');
NAME
-----
Gaardiner
Gaberd
```

- **NLS\_UPPER(char,'nls\_param'):** 该函数用于将字符串转变为大写，其中 nls\_param 的格式为“nls\_sort=sort”，用于指定特定语言特征。示例如下：

```
SQL> DECLARE
  2   v_upper VARCHAR2(10);
  3  BEGIN
  4    v_upper:=nls_upper('my word','nls_sort=XGERMAN');
  5    dbms_output.put_line('字符串大写:'||v_upper);
  6  END;
  7 /
字符串大写:MY WORD
```

- **REGEXP\_REPLACE(source\_string,pattern[,replace\_string[,position[,occurrence[,match\_parameter]]]])**: 该函数是 Oracle 10g 新增加的函数, 它扩展了函数 REPLACE 的功能, 并且该函数用于按照特定表达式的规则替换字符串。其中, 参数 source\_string 用于指定源字符表达式, pattern 用于指定规则表达式, replace\_string 用于指定替换字符串, position 用于指定起始搜索位置, occurrence 用于指定替换出现的第 n 个字符串, match\_parameter 用于指定默认匹配操作的文本串。示例如下:

```
SQL> SELECT REGEXP_REPLACE(country_name, '(.)', '\1') "REGEXP_REPLACE"
  2> FROM countries;
REGEXP_REPLACE
-----
Argentina
Australia
Belgium
Brazil
Canada
.
.
```

- **REGEXP\_SUBSTR(source\_string,pattern[,position[,occurrence[,match\_parameter]]])**: 该函数是 Oracle 10g 新增加的函数, 它扩展了函数 SUBSTR 的功能, 并且用于按照特定表达式的规则返回字符串的子串。其中, source\_string 用于指定源字符串表达式, pattern 用于指定规则表达式, position 用于指定起始搜索位置, occurrence 用于指定第 n 次出现的字符串, match\_parameter 用于指定默认匹配操作的文本串。示例如下:

```
SQL> SELECT REGEXP_SUBSTR('http://www.oracle.com/products',
  2> 'http://([[:alnum:]]+\.(?)(3,4)/?)' "REGEXP_SUBSTR" FROM DUAL;
REGEXP_SUBSTR
-----
http://www.oracle.com/
```

- **REPLACE(char,search\_string[,replacement\_string])**: 该函数用于将字符串的子串替换为其他子串。如果 replacement\_string 为 null, 则会去掉指定子串; 如果 search\_string 为 null, 则返回原有字符串。示例如下:

```
SQL> SELECT replace('缺省值为 10','缺省','默认') FROM dual;
REPLACE(
-----
默认值为 10
```

- **RPAD(char1,n,char2)**: 该函数用于在字符串 char1 的右端填充字符串 char2, 直至字符串的总长度为 n, char2 的默认值为空格。如果 char1 长度大于 n, 则该函数返回 char1 左端的 n 个字符。示例如下:

```
SQL> DECLARE
  2    v_rpad VARCHAR2(10);
  3  BEGIN
  4    v_rpad:=rpad('aaaa',10,'*');
  5    dbms_output.put_line('在右端添加字符:'||v_rpad);
  6  END;
  7  /
```

在右端添加字符:aaaa\*\*\*\*\*

- **RTRIM(char [,set]):** 该函数用于去掉字符串 char 右端所包含的、set 中的任何字符。Oracle 从右端第一个字符开始扫描，逐一去掉在 set 中出现的字符，当遇到不是 set 中的字符时终止，然后返回剩余结果。示例如下：

```
SQL> SELECT rtrim('morning','ing') FROM dual;
RTR
---
mor
```

- **SOUNDEX(char):** 该函数用于返回字符串的语音表示，使用该函数可以比较发音相同的字符串。示例如下：

```
SQL> SELECT soundex('ship'),soundex('sheep') FROM dual;
SOUN SOUN
-----
S100 S100
```

- **SUBSTR(char,m[,n]):** 该函数用于取得字符串的子串，其中数字 m 是字符开始位置，数字 n 是子串的长度。如果 m 为 0，则从首字符开始；如果 m 是负数，则从尾部开始。示例如下：

```
SQL> DECLARE
 2   v_subs VARCHAR2(10);
 3 BEGIN
 4   v_subs:=substr('morning',1,3);
 5   dbms_output.put_line('字符串的子串:'||v_subs);
 6 END;
 7 /
```

字符串的子串:mor

- **TRANSLATE(char,from\_string,to\_string):** 该函数用于将字符串 char 的字符按照 from\_string 和 to\_string 的对应关系进行转换。示例如下：

```
SQL> SELECT TRANSLATE ('2KRW229',
 2   '0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ',
 3   '9999999999XXXXXX9999XXXXXX9999XXXXXX') "TRANS"
 4  FROM DUAL;
TRANS
-----
9XXX999
```

- **TRIM(char FROM string):** 该函数用于从字符串的头部、尾部或两端截断特定字符，参数 char 为要截去的字符，string 是源的字符串。示例如下：

```
SQL> DECLARE
 2   v_source VARCHAR2(20):='ABCDGHJHJAB';
 3   v_trim VARCHAR2(20);
 4 BEGIN
 5   v_trim:=trim('A' FROM v_source);
 6   dbms_output.put_line(v_trim);
 7 END;
 8 /
```

BCDGHIJHJAB

- **UPPER(char)**: 该函数用于将字符串转换为大写格式。示例如下:

```
SQL> SELECT upper('sql') FROM dual;
UPP
---
SQL
```

### 5.3 日期时间函数

日期时间函数用于处理 DATE 和 TIMESTAMP 类型的数据。除了函数 MONTHS\_BETWEEN 返回数字值外，其他日期函数均返回 DATE 类型的数据。Oracle 是以 7 位数字格式来存放日期数据的，包括世纪、年、月、日、小时、分钟、秒，并且默认日期显示格式为“DD-MON-YY”。下面详细介绍 Oracle 所提供的日期时间函数，以及在 SQL 语句和 PL/SQL 块中使用这些函数的方法。

- **ADD\_MONTHS(d,n)**: 该函数用于返回特定日期时间 d 之后（或之前）的 n 个月所对应的日期时间（n 为正整数表示之后；n 为负整数表示之前）。示例如下：

```
SQL> DECLARE
  2   v_date DATE;
  3 BEGIN
  4   v_date:=add_months(sysdate,-14);
  5   dbms_output.put_line('当前日期前 14 个月对应的日期：'
  6   ||v_date);
  7 END;
  8 /
```

当前日期前 14 个月对应的日期:27-10 月-02

- **CURRENT\_DATE**: 该函数是 Oracle9i 新增加的函数，用于返回当前会话时区所对应的日期时间。示例如下：

```
SQL> ALTER SESSION SET TIME_ZONE = '+5:0';
SQL> ALTER SESSION SET nls_date_format='YYYY-MM-DD HH24:MI';
SQL> SELECT current_date FROM dual;
CURRENT_DATE
```

-----  
2003-12-27 06:44

- **CURRENT\_TIMESTAMP**: 该函数是 Oracle9i 新增加的函数，用于返回当前会话时区的日期时间。示例如下：

```
SQL> SELECT current_timestamp FROM dual;
CURRENT_TIMESTAMP
```

-----  
27-12 月-03 07.45.22.146000 下午 +08:00

- **DBTIMEZONE**: 该函数是 Oracle9i 新增加的函数，用于返回数据库所在时区。示例如下：

```
SQL> DECLARE
  2   v_zone VARCHAR2(10);
```

```

3  BEGIN
4      v_zone:=dbtimezone;
5      dbms_output.put_line('当前数据库时区:'||v_zone);
6  END;
7 /

```

当前数据库时区:+08:00

- EXTRACT: 该函数是 Oracle9i 新增加的函数, 用于从日期时间值中取得所需要的特定数据(例如取得年份、月份等)。示例如下:

```

SQL> SELECT extract(YEAR FROM sysdate) year FROM dual;
      YEAR
-----
      2003

```

- FROM\_TZ: 该函数是 Oracle9i 新增加的函数, 用于将特定时区的 TIMESTAMP 值转变为 TIMESTAMP WITH TIME ZONE 值。示例如下:

```

SQL> DECLARE
2      v_tzv VARCHAR2(100);
3  BEGIN
4      v_tzv:=from_tz(TIMESTAMP '2003-03-28 08:00:00','3:00');
5      dbms_output.put_line(v_tzv);
6  END;
7 /

```

28-3月 -03 08.00.00.000000000 上午 +03:00

- LAST\_DAY(d): 该函数用于返回特定日期所在月份的最后一天。示例如下:

```

SQL> SELECT last_day(sysdate) FROM dual;
LAST_DAY(S
-----
31-12月-03

```

- LOCALTIMESTAMP: 该函数是 Oracle9i 新增加的函数, 用于返回当前会话时区的日期时间。示例如下:

```

SQL> DECLARE
2      v_ts VARCHAR2(100);
3  BEGIN
4      v_ts:=localtimestamp;
5      dbms_output.put_line('当前日期时间:'||v_ts);
6  END;
7 /

```

当前日期时间:27-12月-03 08.25.08.607000000 下午

- MONTHS\_BETWEEN(d1, d2): 该函数用于返回日期 d1 和 d2 之间相差的月数。如果 d1 小于 d2, 则返回负数。如果日期 d1 和 d2 的天数相同或都是月底, 则返回整数; 否则 Oracle 以每月 31 天为准来计算结果的小数部分。示例如下:

```

SQL> SELECT months_between(sysdate,'31-8月-1998')
2  FROM dual;
MONTHS_BETWEEN(SYSDATE,'31-8月-1998')
-----
```

63.8984229

- NEW\_TIME(date,zone1,zone2): 该函数用于返回时区一的日期时间所对应的时区二的日期时间。示例如下：

```
SQL> DECLARE
  2   v_time DATE;
  3 BEGIN
  4   dbms_session.set_nls('nls_date_format',
  5   'YYYY-MM-DD HH24:MI:SS');
  6   v_time:=new_time(to_date('2003-11-10 12:10:00',
  7   'YYYY-MM-DD HH24:MI:SS'),'BST','EST');
  8   dbms_output.put_line('当前日期时间:'||v_time);
  9 END;
10 /
当前日期时间:2003-11-10 18:10:00
```

- NEXT\_DAY(d, char): 该函数用于返回指定日期后的第一个工作日（由 char 指定）所对应的日期。示例如下：

```
SQL> SELECT next_day(sysdate,'星期一') FROM dual;
NEXT_DAY(S
-----
29-12月-03
```

- NUMTODSINTERNAL(n,char\_expr): 该函数用于将数字 n 转换为 INTERVAL DAY TO SECOND 格式，其中 char\_expr 可以是 DAY, HOUR, MINUTE 或 SECOND。示例如下：

```
SQL> DECLARE
  2   v_date VARCHAR2(100);
  3 BEGIN
  4   v_date:=numtodsinterval(10000,'MINUTE');
  5   dbms_output.put_line('10000分钟对应的时间:'||v_date);
  6 END;
  7 /
10000分钟对应的时间: +000000006 22:40:00.000000000
```

- NUMTOYMINTERNAL(n,char\_expr): 该函数用于将数字 n 转换为 INTERVAL YEAR TO MONTH 格式，其中 char\_expr 可以是 YEAR 或 MONTH。示例如下：

```
SQL> SELECT numtoyminterval(100,'MONTH') AS year_month FROM dual;
YEAR_MONTH
-----
+000000008-04
```

- ROUND(d,[fmt]): 该函数用于返回日期时间的四舍五入结果。如果 fmt 指定年度，则 7 月 1 日为分界线；如果 fmt 指定月，则 16 日为分界线；如果指定天，则中午 12:00 时为分界线。示例如下：

```
SQL> DECLARE
  2   v_date DATE;
  3 BEGIN
  4   v_date:=ROUND(SYSDATE,'MONTH');
```

```

5   dbms_output.put_line(SYSDATE||'四舍五入结果:'||v_date);
6 END;
7 /
28-12月-03 四舍五入结果:01-1月 -04

```

- **SESSIONTIMEZONE**: 该函数是 Oracle9i 新增加的函数, 用于返回当前会话所在时区。示例如下:

```

SQL> SELECT sessiontimezone FROM dual;
SESSIONTIMEZONE
-----
+08:00

```

- **SYS\_EXTRACT\_UTC(datetime\_with\_timezone)**: 该函数用于返回特定时区时间所对应的格林威治时间。示例如下:

```

SQL> DECLARE
2   v_timestamp TIMESTAMP;
3 BEGIN
4   v_timestamp:=SYS_EXTRACT_UTC(SYSTIMESTAMP);
5   dbms_output.put_line('格林威治时间:'||v_timestamp);
6 END;
7 /

```

格林威治时间:27-12月-03 11.43.54.285000 下午

- **SYSDATE**: 该函数用于返回当前系统的日期时间, 示例如下:

```

SQL> SELECT sysdate FROM dual;
SYSDATE
-----
28-12月-03

```

- **SYSTIMESTAMP**: 该函数是 Oracle9i 新增加的函数, 用于返回当前系统的日期时间及时区。示例如下:

```

SQL> DECLARE
2   v_timestamp VARCHAR2(100);
3 BEGIN
4   v_timestamp:=SYSTIMESTAMP;
5   dbms_output.put_line('当前系统时间及时区:'||v_timestamp);
6 END;
7 /

```

当前系统时间及时区:28-12月-03 07.46.47.745000000 上午 +08:00

- **TO\_DSINTERNAL(char['nls\_param'])**: 该函数是 Oracle9i 新增加的函数, 用于将符合特定日期和时间格式的字符串转变为 INTERVAL DAY TO SECOND 类型。示例如下:

```

SQL> SELECT to_dsinterval('58:10:10') FROM dual;
TO_DSINTERVAL('58:10:10')
-----
+000000005 08:10:10.000000000

```

- **TO\_TIMESTAMP(char[fmt,'nls\_param'])**: 该函数是 Oracle9i 新增加的函数, 用于将符合特定日期和时间格式的字符串转变为 TIMESTAMP 类型。示例如下:

```
SQL> DECLARE
```

```

2   v_timestamp TIMESTAMP;
3 BEGIN
4   v_timestamp:=TO_TIMESTAMP('01-1月-03');
5   dbms_output.put_line('日期时间值:'||v_timestamp);
6 END;
7 /

```

日期时间值:01-1月 -03 12.00.00.000000 上午

- **TO\_TIMESTAMP\_TZ(char[fmt,['nls\_param']])**: 该函数是 Oracle9i 新增加的函数, 用于将符合特定日期和时间格式的字符串转变为 TIMESTAMP WITH TIME ZONE 类型。示例如下:

```

SQL> SELECT to_timestamp_tz('2003-01-01','YYYY-MM-DD')
2 FROM dual;
TO_TIMESTAMP_TZ('2003-01-01','YYYY-MM-DD')
-----
01-1月 -03 12.00.00.000000000 上午 +08:00

```

- **TO\_YMINTERNAL(char)**: 该函数是 Oracle9i 新增加的函数, 用于将字符串转变为 INTERVAL YEAR TO MONTH 类型。示例如下:

```

SQL> DECLARE
2   v_date DATE;
3 BEGIN
4   v_date:=SYSDATE+TO_YMINTERVAL('01-01');
5   dbms_output.put_line('当前日期后的1年1个月:'||v_date);
6 END;
7 /

```

当前日期后的1年1个月:28-1月 -05

- **TRUNC(d,[fmt])**: 该函数用于截断日期时间数据。如果 fmt 指定年度, 则结果为本年度的1月1日; 如果 fmt 指定月, 则结果为本月1日。示例如下:

```

SQL> SELECT trunc(sysdate,'MONTH') FROM dual;
TRUNC(SYSD
-----
01-12月-03

```

- **TZ\_OFFSET(time\_zone\_name||SESSIONTIMEZONE||DBTIMEZONE)**: 该函数是 Oracle9i 新增加的函数, 用于返回特定时区与 UTC (格林威治) 相比的时区偏移。示例如下:

```

SQL> select tz_offset('EST') FROM dual;
TZ_OFFSET
-----
-05:00

```

## 5.4 转换函数

转换函数用于将数值从一种数据类型转换为另一种数据类型。在某些情况下, Oracle Server 会隐含地转换数据类型。但在编写应用程序时, 为了防止出现编译错误, 如果数据类型

不同，那么应该使用转换函数进行类型转换。下面详细介绍 Oracle 所提供的各种转换函数，以及在 SQL 语句和 PL/SQL 块中使用这些转换函数的方法。

- **ASCIISTR(string)**: 该函数是 Oracle9i 新增加的函数，用于将任意字符集的字符串转变为数据库字符集的 ASCII 字符串。示例如下：

```
SQL> SELECT ASCIISTR('中国') FROM DUAL;
ASCIISTR('
-----
\4E2D\56FD
```

- **BIN\_TO\_NUM(expr[,expr][,expr]...)**: 该函数是 Oracle9i 新增加的函数，用于将位向量值转变为实际的数字值。示例如下：

```
SQL> select bin_to_num(1,0,1,1,1) from dual;
BIN_TO_NUM(1,0,1,1,1)
-----
23
```

- **CAST(expr AS type\_name)**: 该函数用于将一个内置数据类型或集合类型转变为另一个内置数据类型或集合类型。示例如下：

```
SQL> DECLARE
 2   v_cast VARCHAR2(20);
 3   BEGIN
 4     v_cast:=cast(SYSDATE AS VARCHAR2);
 5     dbms_output.put_line('转换结果:'||v_cast);
 6   END;
 7 /
转换结果:28-12月-03
```

- **CHARTOROWID(char)**: 该函数用于将字符串值转变为 ROWID 数据类型，但字符串值必须符合 ROWID 格式。示例如下：

```
SQL> SELECT CHARTOROWID('AAAFd1AAFAAAABSAA/') FROM dual;
CHARTOROWID('AAAFD
-----
AAAFd1AAFAAAABSAA/
```

- **COMPOSE(string)**: 该函数是 Oracle9i 新增加的函数，用于将输入字符串转变为 UNICODE 字符串值。示例如下：

```
SQL> SELECT COMPOSE ('o' || UNISTR('\0308')) FROM DUAL;
CO
--
?
```

- **CONVERT(char,dest\_char\_set,source\_char\_set)**: 该函数用于将字符串从一个字符集转变为另一个字符集。示例如下：

```
SQL> DECLARE
 2   v_convert VARCHAR2(20);
 3   BEGIN
 4     v_convert:=CONVERT('中国','US7ASCII','WE8ISO8859P1');
 5     dbms_output.put_line('转换结果:'||v_convert);
 6   END;
```

```
7 /
转换结果:O??u
```

- DECOMPOSE(string): 该函数是 Oracle9i 新增加的函数, 用于分解字符串并返回相应的 UNICODE 字符串。示例如下:

```
SQL> SELECT DECOMPOSE ('Châteaux') FROM DUAL;
DECOMPOS
-----
Chateaux
```

- HEXTORAW(char): 该函数用于将十六进制字符串转变为 RAW 数据类型。示例如下:

```
SQL> DECLARE
 2   v_raw RAW(20);
 3   BEGIN
 4     v_raw:=HEXTORAW('AB56FA2C');
 5     dbms_output.put_line('转换结果:'||v_raw);
 6   END;
 7 /
转换结果:AB56FA2C
```

- NUMTODSINTERNAL(n,char\_expr): 上一节已详细说明。
- NUMTOYMINTERNAL(n,char\_expr): 上一节已详细说明
- RAWTOHEX(raw): 该函数是 Oracle9i 新增加的函数, 用于将 RAW 数值转变为十六进制字符串。示例如下:

```
SQL> DECLARE
 2   v_char VARCHAR2(20);
 3   BEGIN
 4     v_char:=RAWTOHEX('AB56FA2C');
 5     dbms_output.put_line('转换结果:'||v_char);
 6   END;
 7 /
转换结果:AB56FA2C
```

- RAWTONHEX(raw): 该函数是 Oracle9i 新增加的函数, 用于将 RAW 数值转变为 NVARCHAR2 的十六进制字符串。示例如下:

```
SQL> SELECT rawtonhex('7D') FROM dual;
RAWTONHE
-----
3744
```

- ROWIDTOCHAR(rowid): 该函数是 Oracle9i 新增加的函数, 用于将 ROWID 值转变为 VARCHAR2 数据类型。示例如下:

```
SQL> DECLARE
 2   v_char VARCHAR2(20);
 3   BEGIN
 4     v_char:=ROWIDTOCHAR('AAAFFIAAFAAAAABSAAb');
 5     dbms_output.put_line('转换结果:'||v_char);
 6   END;
 7 /
```

转换结果: AAAFFIAAFAAAABSAAb

- ROWIDTONCHAR(rowid): 该函数是 Oracle9i 新增加的函数, 用于将 ROWID 值转变为 NVARCHAR2 数据类型。示例如下:

```
SQL> SELECT rowidtonchar('AAAFFIAAFAAAABSAAb') FROM dual;
ROWIDTONCHAR('AAAFFIAAFAAAABSAAb')
```

```
-----  
AAAFFIAAFAAAABSAAb
```

- SCN\_TO\_TIMESTAMP(number): 该函数是 Oracle 10g 新增加的函数, 用于根据输入的 SCN 值返回所对应的大致日期时间, 其中 number 用于指定 SCN 值。示例如下:

```
SQL> SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM emp
  2 WHERE empno=7788;
SCN_TO_TIMESTAMP(ORA_ROWSCN)
```

```
-----  
28-3月 -04 03.17.08.000000000 下午
```

- TIMESTAMP\_TO\_SCN(timestamp): 该函数是 Oracle 10g 新增加的函数, 用于根据输入的 TIMESTAMP 返回所对应的 SCN 值, 其中 timestamp 用于指定日期时间。示例如下:

```
SQL> SELECT TIMESTAMP_TO_SCN(order_date) FROM orders
  2> WHERE order_id = 5000;
TIMESTAMP_TO_SCN(ORDER_DATE)
```

```
-----  
574100
```

- TO\_CHAR(character): 该函数用于将 NCHAR, NVARCHAR2, CLOB 和 NCLOB 数据转变为数据库字符集数据当用于 NCHAR, NVARCHAR2 和 NCLOB 类型时, 字符串用单引号括起来, 前面加上 n。示例如下:

```
SQL> DECLARE
  2   v_char VARCHAR2(20);
  3 BEGIN
  4   v_char:=TO_CHAR(n'中华人民共和国');
  5   dbms_output.put_line('转换结果:'||v_char);
  6 END;
  7 /
```

转换结果: 中华人民共和国

- TO\_CHAR(date[,fmt[,nls\_param]]): 该函数用于将日期值转变为字符串, 其中 fmt 用于指定日期格式, nls\_param 用于指定 NLS 参数。示例如下:

```
SQL> SELECT to_char(sysdate,'YYYY-MM-DD') FROM dual;
TO_CHAR(SY
```

```
-----  
2003-12-28
```

- TO\_CHAR(n[,fmt[,nls\_param]]): 该函数用于将数字值转变为 VARCHAR2 数据类型。示例如下:

```
SQL> DECLARE
  2   v_char VARCHAR2(20);
  3 BEGIN
```

```

4   v_char:=TO_CHAR(-10000,'L99G999D99MI');
5   dbms_output.put_line('转换结果:'||v_char);
6 END;
7 /

```

转换结果: RMB10,000.00-

- **TO\_CLOB(char):** 该函数是 Oracle9i 新增加的函数, 用于将字符串转变为 CLOB 类型。char 参数使用 NCHAR, NVARCHAR2 和 NCLOB 类型时, 字符串需用单引号括起来, 且在前面加上 n。示例如下:

```

SQL> SELECT to_clob(n'中华人民共和国') FROM dual;
TO_CLOB(N'中华人民共和国')
-----
```

中华人民共和国

- **TO\_DATE(char,[fmt[,nls\_param]]):** 该函数用于将符合特定日期格式的字符串转变为 DATE 类型的值。示例如下:

```

SQL> DECLARE
2   v_date DATE;
3 BEGIN
4   v_date:=TO_DATE('01-01-2001','DD-MM-YYYY');
5   dbms_output.put_line('转换结果:'||v_date);
6 END;
7 /

```

转换结果:01-1月 -01

- **TO\_DSINTERNAL(char,['nls\_param']):** 上一节已详细说明。
- **TO\_LOB(long\_column):** 该函数是 Oracle9i 新增加的函数, 用于将 LONG 或 LONG RAW 列的数据转变为相应的 LOB 类型。示例如下:

```

SQL> INSERT INTO a (clob_column)
2> SELECT TO_LOB(long_column) FROM b;
```

- **TO\_MULTI\_BYTE(char):** 该函数用于将单字节字符串转变为多字节字符串。示例如下:

```

SQL> DECLARE
2   v_multi VARCHAR2(10);
3 BEGIN
4   v_multi:=TO_MULTI_BYTE('abcd');
5   dbms_output.put_line('转换结果:'||v_multi);
6 END;
7 /

```

转换结果:a b c d

- **TO\_NCHAR(character):** 该函数用于将字符串由数据库字符集转变为民族字符集。示例如下:

```

SQL> SELECT to_nchar('伟大的中国') FROM dual;
TO_NCHAR('
-----
```

伟大的中国

- **TO\_NCHAR(datetime,[fmt[,nls\_param]]):** 该函数用于将日期时间值转变为民族字符集

的字符串。示例如下：

```
SQL> DECLARE
 2   v_nchar NVARCHAR2(30);
 3 BEGIN
 4   v_nchar:=to_nchar(SYSDATE);
 5   dbms_output.put_line('转换结果:'||v_nchar);
 6 END;
 7 /
转换结果:28-12月-03
```

- **TO\_NCHAR(number)**: 该函数用于将数字值转变为民族字符集的字符串。示例如下：

```
SQL> SELECT to_nchar(10) FROM dual;
TO_N
-----
10
```

- **TO\_NCLOB(clob\_column|char)**: 该函数用于将 CLOB 列或字符串转变为 NCLOB 类型。示例如下：

```
SQL> DECLARE
 2   v_nclob NCLOB;
 3 BEGIN
 4   v_nclob:=to_nclob('伟大的祖国');
 5 END;
 6 /

```

- **TO\_NUMBER(char,[fmt[,nls\_param]])**: 该函数用于将符合特定数字格式的字符串值转变为数字值。示例如下：

```
SQL> SELECT to_number('RMB1000.00','L99999D99') FROM dual;
TO_NUMBER('RMB1000.00','L99999D99')
-----
1000
```

- **TO\_SINGLE\_BYTE(char)**: 该函数用于将多字节字符集数据转变为单字节字符集。示例如下：

```
SQL> DECLARE
 2   v_single VARCHAR2(10);
 3 BEGIN
 4   v_single:=to_single_byte('a b c d');
 5   dbms_output.put_line('转换结果:'||v_single);
 6 END;
 7 /

```

转换结果:abcd

- **TO\_YMINTERVAL(char)**: 上一节已详细说明。

- **TRANSLATE ... USING**: 该函数用于将字符串转变为数据库字符集 (CHAR\_CS) 或者民族字符集 (NCHAR\_CS)。示例如下：

```
SQL> SELECT translate('中国' USING NCHAR_CS) FROM dual;
TRAN
-----
```

中国

- **UNISTR(string):** 该函数是 Oracle9i 新增加的函数，用于输入字符串并返回相应的 UNICODE 字符。示例如下：

```
SQL> DECLARE
 2   v_unicode VARCHAR2(10);
 3   BEGIN
 4     v_unicode:=UNISTR('\00D6');
 5     dbms_output.put_line('UNICODE 字符:'||v_unicode);
 6   END;
 7 /
UNICODE 字符:?
```

## 5.5 集合函数

从 Oracle 10g 开始，为了扩展集合类型（嵌套表和 VARRAY）的功能，Oracle 新增加了几个新的集合函数。下面给介绍这些集合函数。

- **CARDINALITY(nested\_table):** 该函数是 Oracle 10g 新增加的函数，用于返回嵌套表的实际元素个数。示例如下：

```
SQL> SELECT product_id, CARDINALITY(ad_textdocs_ntab)
 2> FROM print_media;
PRODUCT_ID CARDINALITY(AD_TEXTDOCS_NTAB)
-----
3060 3
2056 3
3106 3
2268 3
```

- **COLLECT(column):** 该函数是 Oracle 10g 新增加的函数，用于根据输入列和被选择行建立嵌套表结果。示例如下：

```
SQL> CREATE TYPE phone_book_t AS TABLE OF phone_list_typ;
 2> /
SQL> SELECT CAST(COLLECT(phone_numbers) AS phone_book_t)
 2> FROM customers;
```

- **POWERMULTISET(expr):** 该函数是 Oracle 10g 新增加的函数，用于生成嵌套表的超集（包含所有非空的嵌套表）。示例如下：

```
SQL> CREATE TYPE cust_address_tab_tab_typ
 2> AS TABLE OF cust_address_tab_typ;
 3> /
SQL> SELECT CAST(POWERMULTISET(cust_address_ntab)
 2> AS cust_address_tab_tab_typ) FROM customers_demo;
CAST(POWERMULTISET(CUST_ADDRESS_NTAB) AS CUST_ADDRESS_TAB_TAB_TYP)
(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
-----
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
```

```
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US')))

. . .

```

- **POWERMULTISET\_BY\_CARDINALITY(expr,cardinality)**: 该函数是 Oracle 10g 新增加的函数，用于根据嵌套表和元素个数，生成嵌套表的超集（包含所有非空的嵌套表）。示例如下：

```
SQL> UPDATE customers_demo SET cust_address_ntab =
  2> cust_address_ntab MULTISET UNION cust_address_ntab;
SQL> SELECT CAST(POWERMULTISET_BY_CARDINALITY(cust_address_ntab, 2)
  2> AS cust_address_tab_tab_typ) FROM customers_demo;
CAST(POWERMULTISET_BY_CARDINALITY(CUST_ADDRESS_NTAB, 2)
AS
CUST_ADDRESS_TAB_TAB_TYP)
(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
-----
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'),
CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'),
CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'),
CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))

. . .

```

- **SET(nested\_table)**: 该函数用于取消嵌套表中的重复结果，并生成新的嵌套表。示例如下：

```
SQL> SELECT SET(cust_address_ntab) address
  2> FROM customers_demo;
ADDRESS(STREET_ADDRESS, POSTAL_CODE, CITY)
-----
CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901',
'Kokomo'))
CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218',
'Indianapolis'))
CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404',
'Bloomington'))
```

## 5.6 其他单行函数

除了单行数字、字符、日期和转换函数之外，Oracle 还提供了一些其他的单行函数。下面详细介绍如何在 SQL 语句和 PL/SQL 块中使用这些单行函数。

- **BFILENAME('directory','filename')**: 该函数用于初始化 BFILE 定位符, 其中 directory 是与 OS 路径相关的 DIRECTORY 对象, filename 是 OS 文件的名称。示例如下:

```
SQL> DECLARE
  2   v_content VARCHAR2(100);
  3   v_bfile BFILE;
  4   amount INT;
  5   offset INT:=1;
  6 BEGIN
  7   v_bfile:=bfilename('USER_DIR','a.txt');
  8   amount:=dbms_lob.getlength(v_bfile);
  9   dbms_lob.fileopen(v_bfile);
 10  dbms_lob.read(v_bfile,amount,offset,v_content);
 11  dbms_lob.fileclose(v_bfile);
 12  dbms_output.put_line(v_content);
 13 END;
 14 /
D6D0B9FAA3ACD6D0B9FAA3ACCEB0B4F3B5C4D6D0B9FA
```

- **COALESCE(expr1[,expr2][,expr3]...)**: 该函数是 Oracle9i 新增加的函数, 用于返回表达式列表中第一个 NOT NULL 表达式的结果。示例如下:

```
SQL> DECLARE
  2   v_expr1 INT;
  3   v_expr2 INT:=100;
  4   v_expr3 INT:=1000;
  5   v_nn INT;
  6 BEGIN
  7   v_nn:=COALESCE(v_expr1,v_expr2,v_expr3);
  8   dbms_output.put_line(v_nn);
  9 END;
 10 /
100
```

- **DECODE(expr,search1,result[,search2,result2,...][,default])**: 该函数用于返回匹配于特定表达式的结果。如果 search1 匹配于 expr, 则返回 result1; 如果 search2 匹配于 expr, 则返回 result2, 依此类推; 如果没有任何匹配关系, 则返回 default。示例如下:

```
SQL> SELECT deptno,ename,sal,
  2 decode(deptno,10,sal*1.2,20,sal*1.1,sal) "New Salary"
  3 FROM emp ORDER BY deptno;
      DEPTNO ENAME          SAL New Salary
----- -----
      10 CLARK           2450    2940
      10 KING            5000    6000
      10 MILLER          1300    1560
      20 SMITH            800     880
      20 ADAMS           1100    1210
```

- **DEPTH(n)**: 该函数是 Oracle9i 新增加的函数, 用于返回 XML 方案中 UNDER\_PATH 路径所对应的相对层数。其中参数 n 用于指定相对层数。示例如下:

```
SQL> SELECT PATH(1), DEPTH(2) FROM resource_view
  2 WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
  3 AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;
PATH(1)                      DEPTH(2)
-----
/www.oracle.com                1
/www.oracle.com/xwarehouses.xsd    2
```

- **DUMP(expr,return\_fmt)**: 该函数用于返回表达式所对应的数据类型代码、长度以及内部表示格式，其中 return\_fmt 用于指定返回格式（8: 八进制符号，10: 十进制符号，16: 十六进制符号，17: 单字符）。注意，该函数只能在 SQL 语句中使用。示例如下：

```
SQL> SELECT dump('hello',1016) FROM dual;
DUMP('HELLO',1016)

-----
```

Type=96 Len=5 CharacterSet=ZHS16GBK: 68,65,6c,6c,6f

- **EMPTY\_BLOB()**: 该函数用于初始化 BLOB 变量，示例如下：

```
SQL> DECLARE
  2   v_lob BLOB;
  3   BEGIN
  4     v_lob:=empty_blob();
  5   END;
  6 /
```

- **EMPTY\_CLOB()**: 该函数用于初始化 CLOB 变量，示例如下：

```
SQL> UPDATE person SET resume=EMPTY_CLOB() WHERE id=1;
```

- **EXISTSNODE(XMLType\_instance,Xpath\_string)**: 该函数是 Oracle9i 新增加的函数，用于确定 XML 节点路径是否存在，返回 0 表示节点不存在，返回 1 表示节点存在。示例如下：

```
SQL> SELECT existsnode(VALUE(p), '/PurchaseOrder/User')
  2 FROM xmstable p;
EXISTSNODE(VALUE(P), '/PURCHASEORDER/USER')

-----
```

1

- **EXTRACT(XMLType\_instance,Xpath\_string)**: 该函数是 Oracle9i 新增加的函数，用于返回 XML 节点路径下的相应内容。示例如下：

```
SQL> SELECT extract(value(p), '/PurchaseOrder/User')
  2 FROM xmstable p;
EXTRACT(VALUE(P), '/PURCHASEORDER/USER')

-----
```

<User>ADAMS</User>

- **EXTRACTVALUE(XMLType\_instance,Xpath\_string)**: 该函数是 Oracle9i 新增加的函数，用于返回 XML 节点路径下的值。示例如下：

```
SQL> SELECT extractvalue(value(p), '/PurchaseOrder/User')
  2 FROM xmstable p;
EXTRACTVALUE(VALUE(P), '/PURCHASEORDER/USER')

-----
```

ADAMS

- GREATEST(expr1 [,expr2] ...): 该函数用于返回列表表达式 expr1, expr2, ... 中值最大的一个。在比较之前, expr2 等项会被隐含地转换为 expr1 的数据类型。示例如下:

```
SQL> SELECT GREATEST ('BLACK', 'BLANK', 'BACK') FROM dual;
```

GREAT

----

BLANK

- LEAST(expr [,expr] ...): 该函数用于返回列表表达式 expr1, expr2, ... 中值最小的一个。在比较之前, expr2 等项会被隐含地转换为 expr1 的数据类型。示例如下:

```
SQL> SELECT LEAST('BLACK', 'BLANK', 'BACK') FROM dual;
```

LEAS

----

BACK

- NLS\_CHARSET\_DECL\_LEN(byte\_count,charset\_id): 该函数用于返回字节数在特定字符集中占用的字符个数。示例如下:

```
SQL> SELECT NLS_CHARSET_DECL_LEN
  2    (200, nls_charset_id('ZHS16GBKFIXED'))
  3  FROM DUAL;
NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('ZHS16GBKFIXED'))
```

-----  
100

- NLS\_CHARSET\_ID(text): 该函数用于返回字符集的 ID 号。示例如下:

```
SQL> SELECT nls_charset_id('ZHS16GBKFIXED') FROM dual;
NLS_CHARSET_ID('ZHS16GBKFIXED')
```

-----  
1852

- NLS\_CHARSET\_NAME(number): 该函数用于返回特定 ID 号所对应的字符集名。示例如下:

```
SQL> SELECT nls_charset_name(852) FROM dual;
NLS_CHAR
-----
ZHS16GBK
```

- NULLIF(expr1,expr2): 该函数是 Oracle9i 新增加的函数, 用于比较表达式 expr1 和 expr2。如果二者相等, 则返回 NULL, 否则返回 expr1。示例如下:

```
SQL> DECLARE
  2    v_expr1 INT:=100;
  3    v_expr2 INT:=100;
  4  BEGIN
  5    IF NULLIF(v_expr1,v_expr2) IS NULL THEN
  6      dbms_output.put_line('二者相等');
  7    ELSE
  8      dbms_output.put_line('二者不等');
  9    END IF;
10  END;
```

11 /  
二者相等

- **NVL(expr1,expr2):** 该函数用于将 NULL 转变为实际值。如果 expr1 是 null, 则返回 expr2; 如果 expr1 不是 null, 则返回 expr1。参数 expr1 和 expr2 可以是任意数据类型, 但二者数据类型必须要匹配。示例如下:

```
SQL> SELECT ename,sal,comm,sal+nvl(comm,0) salary
  2  FROM emp WHERE deptno=30;
    ENAME      SAL      COMM      SALARY
-----  -----  -----  -----
  ALLEN      1760      300      2060
  WARD       1375      500      1875
  MARTIN     1375     1400      2775
  BLAKE      3135          3135
  TURNER     1650          0      1650
  JAMES      1045          1045
```

- **NVL2(expr1,expr2,expr3):** 该函数是 Oracle9i 新增加的函数, 它也用于处理 NULL。如果 expr1 不是 null, 则返回 expr2; 如果 expr1 是 null, 则返回 expr3。参数 expr1 可以是任意数据类型, 而 expr2 和 expr3 可以是除 LONG 之外的任何数据类型。示例如下:

```
SQL> SELECT ename,sal,comm,nvl2(comm,sal+comm,sal) salary
  2  FROM emp WHERE deptno=30;
    ENAME      SAL      COMM      SALARY
-----  -----  -----  -----
  ALLEN      1760      300      2060
  WARD       1375      500      1875
  MARTIN     1375     1400      2775
  BLAKE      3135          3135
  TURNER     1650          0      1650
  JAMES      1045          1045
```

- **PATH(correction\_integer):** 该函数是 Oracle9i 新增加的函数, 用于返回特定 XML 资源所对应的相对路径。示例如下:

```
SQL> SELECT PATH(1), DEPTH(2) FROM resource_view
  2  WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
  3  AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;
    PATH(1)                  DEPTH(2)
-----  -----
/www.oracle.com                      1
/www.oracle.com/xwarehouses.xsd      2
```

- **SYS\_CONNECT\_BY\_PATH(column,char):** 该函数是 Oracle9i 新增加的函数, (只适用于层次查询), 用于返回从根到节点的列值路径。示例如下:

```
SQL> SELECT LPAD(' ', 2*level-1)||_
  2  SYS_CONNECT_BY_PATH(ename,'/') "Path"
  3  FROM emp START WITH ename = 'SCOTT'
  4  CONNECT BY PRIOR empno = mgr;
```

```
Path
```

```
-----  
/SCOTT  
/SCOTT/ADAMS
```

- **SYS\_CONTEXT('context', 'attribute')**: 该函数用于返回应用上下文的特定属性值，其中 context 为应用上下文名，而 attribute 则用于指定属性名。示例如下：

```
SQL> SELECT sys_context('userenv', 'session_user') "数据库用户"  
2   sys_context('userenv', 'os_user') "OS 用户"  
3  FROM dual;  
数据库用户          OS 用户
```

```
-----  
SCOTT           whl
```

- **SYS\_DBURIGEN**: 该函数是 Oracle9i 新增加的函数，根据列或属性生成类型为 DBUriType 的 URL。示例如下：

```
SQL> SELECT sys_dburigen(ename) FROM emp  
2 WHERE empno=7788;  
SYS_DBURIGEN(ENAME) (URL, SPARE)
```

```
-----  
DBURITYPE ('/PUBLIC/EMP/ROW[ENAME='SCOTT']/ENAME', NULL)
```

- **SYS\_EXTRACT\_UTC(datetime\_with\_timezone)**: 在日期时间函数一节中已说明。
- **SYS\_GUID**: 该函数用于生成类型为 RAW 的 16 字节的惟一标识符，每次调用该函数都会生成不同的 RAW 数据。示例如下：

```
SQL> SELECT sys_guid() FROM dual;  
SYS_GUID()
```

```
-----  
6EF8AA169D884A4B8FBA88A4F07844E0
```

- **SYS\_TYPEID(object\_type\_value)**: 该函数用于返回惟一的类型 ID 值。示例如下：

```
SELECT name, SYS_TYPEID(VALUE(p)) "Type_id" FROM persons p;  
NAME Type_id
```

```
-----  
Bob 01
```

```
Joe 02
```

```
Tim 03
```

- **SYS\_XMLAGG(expr[fmt])**: 该函数是 Oracle9i 新增加的函数，用于汇总所有 XML 文档，并生成一个 XML 文档。示例如下：

```
SQL> SELECT SYS_XMLAGG(SYS_XMLGEN(ename))  
2  FROM emp WHERE deptno=10;  
SYS_XMLAGG(SYS_XMLGEN(ENAME))
```

```
-----  
<ROWSET>  
  <ENAME>CLARK</ENAME>  
  <ENAME>KING</ENAME>  
  <ENAME>MILLER</ENAME>  
</ROWSET>
```

- **SYS\_XMLGEN(expr[fmt])**: 该函数是 Oracle9i 新增加的函数，根据数据库表的行和列生成一个 XMLType 实例。示例如下：

```
SQL> SELECT sys_xmlgen(ename) FROM emp WHERE deptno=10;
SYS_XMLGEN(ENAME)
-----
<ENAME>CLARK</ENAME>
<ENAME>KING</ENAME>
<ENAME>MILLER</ENAME>
```

- **UID**: 该函数用于返回当前会话所对应的用户 ID 号。示例如下：

```
SQL> DECLARE
 2   v_uid INT;
 3 BEGIN
 4   v_uid:=UID;
 5   dbms_output.put_line('会话用户 ID 号:'||v_uid);
 6 END;
 7 /
会话用户 ID 号:51
```

- **UPDATEXML(XMLType\_instance,Xpath\_string,value\_expr)**: 该函数是 Oracle9i 新增加的函数，用于更新特定 XMLType 实例相应节点路径的内容。示例如下：

```
SQL> UPDATE xmltable p SET p=UPDATEXML(value(p),
 2   '/PurchaseOrder/User/text()', 'SCOTT');
```

- **USER**: 该函数用于返回当前会话所对应的数据库用户名。示例如下：

```
SQL> DECLARE
 2   v_user VARCHAR2(10);
 3 BEGIN
 4   v_user:=USER;
 5   dbms_output.put_line('会话用户:'||v_user);
 6 END;
 7 /
会话用户:SCOTT
```

- **USERENV(parameter)**: 该函数用于返回当前会话上下文的属性信息，其中 parameter 有以下取值：

**ISDBA**: 如果用户具有 DBA 权限，则返回 TRUE；否则返回 FALSE。

**LANGUAGE**: 返回当前会话的语言、地区和字符集。

**TERMINAL**: 返回当前会话所在终端的 OS 标识符。

**CLIENT\_INFO**: 返回由包 DBMS\_APPLICATION\_INFO 所存储的用户会话信息（最长 64 字节）。

使用该函数的示例如下：

```
SQL> SELECT USERENV('LANGUAGE') FROM DUAL;
USERENV('LANGUAGE')
-----
SIMPLIFIED CHINESE_CHINA.ZHS16GBK
```

- **VSIZE(expr)**: 该函数用于返回 Oracle 内部存储 expr 的实际字节数。如果 expr 是 null，则该函数返回 null。注意，该函数只能在 SQL 语句中使用。示例如下：

```
SQL> SELECT ename,vsize(ename) FROM emp WHERE deptno=10;
ENAME      VSIZE(ENAME)
-----
CLARK          5
KING           4
MILLER         6
```

- XMLAGG(XMLType\_instance [ORDER BY sort\_list]): 该函数是 Oracle9i 新增加的函数，用于汇总多个 XML 块，并生成 XML 文档。示例如下：

```
SQL> SELECT xmagg(xmlelement("employee",ename||' '||sal))
  2  FROM emp WHERE deptno=10;
XMLAGG(XMLELEMENT ("EMPLOYEE",ENAME||' '||SAL))
-----
<employee>CLARK 2450</employee>
<employee>KING 5000</employee>
<employee>MILLER 1300</employee>
```

- XMLCOLATTVAL(value\_expr1[,value\_expr2],...): 该函数是 Oracle9i 新增加的函数，用于生成 XML 块，并增加“column”作为属性名。示例如下：

```
SQL> SELECT xmlelement("emp",xmlcolattval(ename,sal))
  2  FROM emp WHERE empno=7788;
XMLELEMENT ("EMP", XMLCOLATTVAL (ENAME, SAL))
-----
<emp>
  <column name="ENAME">SCOTT</column>
  <column name="SAL">3000</column>
</emp>
```

- XMLCONCAT(XMLType\_instance1[,XMLType\_instance2],...): 该函数是 Oracle9i 新增加的函数，用于连接多个 XMLType 实例，并生成一个新的 XMLType 实例。示例如下：

```
SQL> SELECT xmlconcat(xmlelement("ename",ename),
  2   xmlelement("sal",sal))
  3  FROM emp WHERE deptno=10;
XMLCONCAT (XMLELEMENT ("ENAME", ENAME), XMLELEMENT ("SAL", SAL))
-----
<ename>CLARK</ename>
<sal>2450</sal>
<ename>KING</ename>
<sal>5000</sal>
<ename>MILLER</ename>
<sal>1300</sal>
```

- XMLELEMENT(identifier[,xml\_attribute\_clause][,value\_expr]): 该函数是 Oracle9i 新增加的函数，用于返回 XMLType 的实例，其中参数 identifier (必须) 用于指定元素名，参数 xml\_attribute\_clause (可选) 用于指定元素属性子句，参数 value\_expr (可选) 用于指定元素值。示例如下：

```
SQL> select xmlelement("DATE",sysdate) from dual;
```

```

XMLEMENT ("DATE", SYSDATE)
-----
<DATE>28-12-03</DATE>
SQL> SELECT xmlement ("Emp",
 2   xmlattributes(empno AS "ID", ename)) Employee
 3  FROM emp WHERE deptno = 10;
EMPLOYEE

-----
<Emp ID="7782" ENAME="CLARK"/>
<Emp ID="7839" ENAME="KING"/>
<Emp ID="7934" ENAME="MILLER"/>

```

- **XMLFOREST(value\_expr1[,value\_expr2],...):** 该函数是 Oracle9i 新增加的函数，用于返回 XML 块。示例如下：

```

SQL> SELECT xmlement ("Employee",xmlforest(ename,sal))
 2  FROM emp WHERE empno=7788;
XMLEMENT ("EMPLOYEE", XMLFOREST (ENAME, SAL))

-----
<Employee>
  <ENAME>SCOTT</ENAME>
  <SAL>3000</SAL>
</Employee>

```

- **XMLSEQUENCE(xmltype\_instance):** 该函数是 Oracle9i 新增加的函数，用于返回 XMLType 实例中顶级节点以下的 VARRAY 元素。示例如下：

```

SQL> SELECT xmlsequence(extract(value(x),
 2   '/PurchaseOrder/LineItems/*')) varray FROM xmitable x;
VARRAY

-----
XMLSEQUENCETYPE (XMLTYPE (<LineItem ItemNumber="1">
  <Description>The Ruling Class</Description>
  <Part Id="715515012423" UnitPrice="39.95" Quantity="2"/>
</LineItem>
), XMLTYPE (<LineItem ItemNumber="2">
  <Description>Diabolique</Description>
  <Part Id="037429135020" UnitPrice="29.95" Quantity="3"/>
</LineItem>
), XMLTYPE (<LineItem ItemNumber="3">
  <Description>8 1/2</Description>
  <Part Id="037429135624" UnitPrice="39.95" Quantity="4"/>
</LineItem>
))

```

- **XMLTRANSFORM(xmltype\_instance,xsl\_ss):** 该函数是 Oracle9i 新增加的函数，用于将 XMLType 实例按照 XSL 样式进行转换，并生成新的 XMLType 实例。示例如下：

```

SQL> SELECT XMLTRANSFORM(w.warehouse_spec, x.col1).GetClobVal()
 2  FROM warehouses w, xsl_tab x
 3  WHERE w.warehouse_name = 'San Francisco';

```

## 5.7 分组函数

分组函数也被称为多行函数，它会根据输入的多行数据返回一个结果。分组函数主要用于执行数据统计或数据汇总操作，并且分组函数只能出现在 SELECT 语句的选择列表、ORDER BY 子句和 HAVING 子句中。注意，分组函数不能直接在 PL/SQL 语句中引用，而只能在内嵌 SELECT 语句中使用。下面详细介绍 Oracle 所提供的各种分组函数，以及在 SQL 语句中引用这些分组函数的方法。

- **AVG([ALL|DISTINCT]expr):** 该函数用于计算平均值。示例如下：

```
SQL> DECLARE
  2   v_avg NUMBER(6,2);
  3   BEGIN
  4     SELECT avg(sal) INTO v_avg FROM emp;
  5     dbms_output.put_line('雇员平均工资:'||v_avg);
  6   END;
  7 /
雇员平均工资:2140.36
```

- **CORR(expr1,expr2):** 该函数用于返回成对数值的相关系数，其数值使用表达式 “ $\text{COVAR\_POP(expr1,expr2)} / (\text{STDDEV\_POP(expr1)} * \text{STDDEV\_POP(expr2)})$ ” 获得。示例如下：

```
SQL> SELECT weight_class, CORR(list_price, min_price)
  2   FROM product_information
  3   GROUP BY weight_class;
WEIGHT_CLASS    CORR(LIST_PRICE,MIN_PRICE)
-----
1              .99914795
2              .999022941
3              .998484472
4              .999359909
5              .999536087
```

- **COUNT([ALL|DISTINCT]expr):** 该函数用于返回总计行数。示例如下：

```
SQL> SELECT count(distinct sal) FROM emp;
COUNT(DISTINCTSAL)
-----
12
```

- **COVAR\_POP(expr1,expr2):** 该函数用于返回成对数字的协方差（Covariance），其数值使用表达式 “ $(\sum(\text{expr1} * \text{expr2}) - \sum(\text{expr1}) * \sum(\text{expr2}) / n) / n$ ” 取得。示例如下：

```
SELECT t.calendar_month_number,
       COVAR_POP(s.amount_sold, s.quantity_sold) AS covar_pop,
       COVAR_SAMP(s.amount_sold, s.quantity_sold) AS covar_samp
  FROM sales s, times t
 WHERE s.time_id = t.time_id AND t.calendar_year = 1998
 GROUP BY t.calendar_month_number;
```

CALENDAR_MONTH_NUMBER	COVAR_POP	COVAR_SAMP
1	5437.68586	5437.88704
2	5923.72544	5923.99139
3	6040.11777	6040.38623
4	5946.67897	5946.92754

- COVAR\_SAMP(expr1,expr2): 该函数用于返回成对数字的协方差，其数值使用表达式“(sum(expr1\*expr2)-sum(expr1)\*sum(expr2)/n)/(n-1)”取得。
- CUME\_DIST(expr1,expr2,...) WITHIN GROUP (ORDER BY expr1,expr2,...): 该函数用于返回特定数值在一组行数据中的累积分布比例。示例如下：

```
SQL> SELECT cume_dist(2000) WITHIN GROUP
  2  (ORDER BY sal) "CUME-DIST" FROM emp;
CUME-DIST
```

- DENSE\_RANK(expr1,expr2,...) WITHIN GROUP (ORDER BY expr1,expr2,...): 该函数用于返回特定数据在一组行数据中的等级。示例如下：

```
SQL> SELECT dense_rank(5000) WITHIN GROUP
  2  (ORDER BY sal) rank FROM emp;
RANK
```

- FIRST: 该函数是 Oracle9i 新增加的函数，不能单独使用，必须与其他分组函数结合使用。通过使用该函数，可以取得排序等级的第一级，然后使用分组函数汇总该等级的数据。示例如下：

```
SQL> SELECT MIN(sal) KEEP (DENSE_RANK FIRST ORDER BY comm DESC)
  2  "补助最高级别雇员的最低工资",
  3  MAX(sal) KEEP (DENSE_RANK FIRST ORDER BY comm DESC)
  4  "补助最高级别雇员的最高工资"
  5  FROM emp;
```

补助最高级别雇员的最低工资 补助最高级别雇员的最高工资

-----  
800 5000

- GROUP\_ID(): 该函数是 Oracle9i 新增加的函数，用于区分分组结果中的重复行。示例如下：

```
SQL> SELECT deptno, job, avg(sal), group_id()
  2  FROM emp GROUP BY deptno, ROLLUP(deptno, job);
DEPTNO JOB      AVG(SAL) GROUP_ID()
```

```
-----  
10 CLERK      1300      0  
10 MANAGER    2450      0  
10 PRESIDENT  5000      0  
20 CLERK      950       0  
20 ANALYST   3000      0
```

```

20 MANAGER      2975      0
30 CLERK        950       0
30 MANAGER      2850      0
30 SALESMAN     1400      0
10             2916.66667    0
20             2175       0
DEPTNO JOB      AVG(SAL) GROUP_ID()
-----
30           1566.66667    0
10           2916.66667    1
20           2175       1
30           1566.66667    1

```

- **GROUPING(expr)**: 该函数用于确定分组结果是否用到了特定的表达式，返回值为 0，表示用到了该表达式，返回值为 1 表示未用该表达式。示例如下：

```

SQL> SELECT deptno,job,sum(sal),grouping(job)
  2 FROM emp GROUP BY rollup(deptno,job);
DEPTNO JOB      SUM(SAL) GROUPING(JOB)
-----
10 CLERK      1300      0
10 MANAGER    2450      0
10 PRESIDENT   5000      0
10             8750      1
...
```

- **GROUPING\_ID(expr1[,expr2],...)**: 该函数是 Oracle9i 新增加的函数，用于返回对应于特定行的 GROUPING 位向量的值。示例如下：

```

SQL> SELECT deptno,job,sum(sal),grouping_id(job,deptno)
  2 FROM emp GROUP BY rollup(deptno,job);
DEPTNO JOB      SUM(SAL) GROUPING_ID(JOB,DEPTNO)
-----
10 CLERK      1300      0
...
10             8750      2
...
29025          3

```

- **LAST**: 该函数是 Oracle9i 新增加的函数，不能单独使用，必须与其他分组函数结合使用。通过使用该函数，可以取得排序等级的最后一级，然后使用分组函数汇总该等级的数据。示例如下：

```

SQL> SELECT MIN(sal) KEEP (DENSE_RANK LAST ORDER BY comm)
  2   "补助最高级别雇员的最低工资",
  3   MAX(sal) KEEP (DENSE_RANK LAST ORDER BY comm)
  4   "补助最高级别雇员的最高工资"
  5 FROM emp;
补助最高级别雇员的最低工资 补助最高级别雇员的最高工资

```

```

-----
```

```

800          5000

```

- **MAX([ALL|DISTINCT]expr):** 该函数用于取得列或表达式的最大值。示例如下：

```
SQL> SELECT deptno,max(sal) FROM emp GROUP BY deptno;
      DEPTNO    MAX(SAL)
----- -----
      10        5000
      20        3000
      30        2850
```

- **MIN([ALL|DISTINCT]expr):** 该函数用于取得列或表达式的最小值。示例如下：

```
SQL> SELECT deptno,min(sal) FROM emp GROUP BY deptno;
      DEPTNO    MIN(SAL)
----- -----
      10        1300
      20        800
      30        950
```

- **PERCENT\_RANK(expr1,expr2,...) WITHIN GROUP (ORDER BY expr1,expr2,...):** 该函数用于返回特定数值在统计级别中所占的比例。示例如下：

```
SQL> SELECT percent_rank(3000) WITHIN GROUP(ORDER BY sal)
      2    percent FROM emp;
      PERCENT
-----
.785714286
```

- **PERCENTILE\_CONT(percent\_expr) WITHIN GROUP (ORDER BY expr):** 该函数是 Oracle9i 新增加的函数，用于返回在统计级别中处于某个百分点的特定数值（按照连续分布模型确定）。示例如下：

```
SQL> SELECT percentile_cont(.6) WITHIN GROUP(ORDER BY sal)
      2    value FROM emp;
      VALUE
-----
2280
```

- **PERCENTILE\_DISC(percent\_expr) WITHIN GROUP (GRDER BY expr):** 该函数是 Oracle9i 新增加的函数，用于返回在统计级别中处于某个百分点的特定数值（按照离散分布模型确定）。示例如下：

```
SQL> SELECT percentile_disc(.6) WITHIN GROUP(ORDER BY sal)
      2    value FROM emp;
      VALUE
-----
2450
```

- **RANK(expr1,expr2,...) WITHIN GROUP (GRDER BY expr1,expr2,...):** 该函数用于返回特定数值在统计数值中所占据的等级。示例如下：

```
SQL> SELECT rank(3000) WITHIN GROUP(ORDER BY sal)
      2    rank FROM emp;
      RANK
-----
12
```

- **STDDEV([ALL|DISTINCT]expr)**: 该函数用于取得标准偏差，其数值是按照方差 VARIANCE 的平方根取得的。示例如下：

```
SQL> SELECT stddev(sal) FROM emp;
STDDEV(SAL)
```

```
-----  
1182.50322
```

- **STDDEV\_POP(expr)**: 该函数用于返回统计标准偏差，并返回统计方差的平方根。示例如下：

```
SQL> SELECT stddev_pop(sal) FROM emp;
STDDEV_POP(SAL)
```

```
-----  
1139.48862
```

- **STDDEV\_SAMP(expr)**: 该函数用于返回采样标准偏差，并返回采样方差的平方根。示例如下：

```
SQL> SELECT stddev_samp(sal) FROM emp;
STDDEV_SAMP(SAL)
```

```
-----  
1182.50322
```

- **SUM([ALL|DISTINCT]expr)**: 该函数用于计算列或表达式的总和。示例如下：

```
SQL> SELECT deptno,sum(sal) FROM emp GROUP BY deptno;
DEPTNO    SUM(SAL)
```

```
-----  
10        8750  
20        10875  
30        9400
```

- **VAR\_POP(expr)**: 该函数用于返回统计方差，其数值使用公式“ $\text{sum(expr}^*\text{expr)} - \text{sum(expr})*\text{sum(expr)}/\text{COUNT(expr)}$ ”取得。示例如下：

```
SQL> SELECT var_pop(sal) FROM emp;
VAR_POP(SAL)
```

```
-----  
1298434.31
```

- **VAR\_SAMP(expr)**: 该函数用于返回采样方差，其数值使用公式“ $\text{sum(expr}^*\text{expr)} - \text{sum(expr})*\text{sum(expr)}/\text{COUNT(expr)}$ ”取得。示例如下：

```
SQL> SELECT var_samp(sal) FROM emp;
VAR_SAMP(SAL)
```

```
-----  
1398313.87
```

- **VARIANCE([ALL|DISTINCT]expr)**: 该函数用于返回列或表达式的方差，其数值与 VAR\_SAMP 完全相同。示例如下：

```
SQL> SELECT variance(sal) FROM emp;
VARIANCE(SAL)
```

```
-----  
1398313.87
```

## 5.8 对象函数

对象函数用于操纵 REF 对象。REF 对象实际是指向对象类型数据的指针，为了操纵 REF 对象，必须要使用对象函数。在介绍如何使用这些对象函数之前，首先建立示例对象表，并为它们插入数据。示例如下：

```

SQL> CREATE TYPE cust_address_typ_new AS OBJECT
  2  ( street_address VARCHAR2(40)
  3 , postal_code VARCHAR2(10)
  4 , city VARCHAR2(30)
  5 , state_province VARCHAR2(10)
  6 , country_id CHAR(2)
  7 );
  8 /
SQL> CREATE TABLE address_table OF cust_address_typ_new;
SQL> CREATE TABLE customer_addresses (
  2 add_id NUMBER,
  3 address REF cust_address_typ_new
  4 SCOPE IS address_table);
SQL> INSERT INTO address_table VALUES
  2 ('1 First','G45 EU8','Paris','CA','US');
SQL> INSERT INTO customer_addresses
  2 SELECT 999, REF(a) FROM address_table a;
SQL> commit;

```

- **DEREF(expr)**: 该函数用于返回参照对象 expr 所引用的对象实例。示例如下:

```
SQL> SELECT deref(address).city FROM customer_addresses;  
DREF(ADDRESS).CITY
```

Part 5

- **MAKE\_REF(object\_table|object\_view,key)**: 该函数可以基于对象视图的一行数据或基于对象表（存在基于主键的对象标识符）的一行数据建立 REF。示例如下：

```
SELECT MAKE_REF (oc_inventories, 3003) FROM DUAL;  
MAKE_REF(OC_INVENTORIES, 3003)
```

- REF(expr): 该函数用于返回对象行所对应的 REF 值。示例如下:

```
SQL> SELECT ref(e) FROM address_table e;
REF(E)
```

0000280209722332B88E884C2F9FCE456DF888DA0823F6D12F97784CD8BBCCA61E9E68  
0140003C0000

- REFTOHEX(expr): 该函数用于将 REF 值转变为十六进制字符串。示例如下

```
SQL> SELECT reftohex(ref(e)) FROM address_table e;
```

```
REFTOHEX(REF(E))
```

```
-----  
0000280209722332B88E884C2F9FCE456DF888DA0823F6D12F97784CD8BBCCA61E9E68  
7D0E0140003C0000
```

- **VALUE(expr):** 该函数用于返回行对象所对应的对象实例数据，其中 expr 用于指定行对象的别名。示例如下：

```
SQL> SELECT value(e).city FROM address_table e;
```

```
VALUE (E).CITY
```

```
-----  
Paris
```

## 5.9 习题

1. 某开发人员希望返回大于等于数字 F 的整数，应该使用以下哪个函数？  
 A. CEIL(F)      B. FLOOR(F)      C. ROUND(F)      D. TRUNC(F+1)
2. 函数 ROUND(45.92)+TRUNC(45.92)的输出结果为  
 A. 89      B. 90      C. 91      D. 92
3. 某开发人员希望字符串返回格式为“Good Morning”，应该使用哪个字符函数？  
 A. UPPER      B. LOWER      C. CONCAT      D. INITCAP
4. 某开发人员希望返回日期 D1 和 D2 之间相差的天数，应该使用以下哪个表达式？  
 A. MONTHS\_BETWEEN(D1,D2)      B. D1-D2
5. 某开发人员希望显示下周二的日期，应该使用哪几个函数？  
 A. NEXT\_DAY      B. SYSDATE      C. LAST\_DAY      D. MONTHS\_BETWEEN
6. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入数字值，然后使用 DBMS\_OUTPUT 显示它的四舍五入结果及其整数值。格式如下：  
 输入 no 的值: 56.67  
 四舍五入结果:57  
 整数值:56
7. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入字符串，然后使用 DBMS\_OUTPUT 输出其大写格式、小写格式，以及首字符大写其他字符小写的格式。格式如下：  
 输入 string 的值: my china  
 大写格式:MY CHINA  
 小写格式:my china  
 首字符大写:My China
8. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入日期数据，然后使用 DBMS\_OUTPUT 输出 10 个月后对应的日期，以及 5 年 10 个月之前所对应的日期。格式如下：  
 输入 date 的值: 2003-10-10  
 5 年 10 个月之前的日期:10-12 月-97  
 10 个月之后的日期:10-8 月 -04
9. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入数字数据，然后使用 DBMS\_OUTPUT

以本地货币格式显示结果。格式如下：

输入 number 的值： 1500

本地货币格式： RMB1,500

10. 编写 PL/SQL 块，使用 DBMS\_OUTPUT 显示 ITEM 表中所有产品的平均单价，以及  
订单条款所有产品的总计价格。格式如下：

产品平均单价:34.42

所有产品价格总和:3903.8

# 第6章 访问 Oracle

PL/SQL 是 Oracle 在标准 SQL 语言上的过程性扩展，通过在 PL/SQL 块中嵌入 SQL 语句，可以访问 Oracle 数据库。注意，在 PL/SQL 块中只能直接嵌入 SELECT、DML 语句（INSERT, UPDATE, DELETE）以及事务控制语句（COMMIT, ROLLBACK, SAVEPOINT），而不能直接嵌入 DDL 语句（CREATE, ALTER, DROP）和 DCL 语句（GRANT, REVOKE）。本章将给大家介绍如何在 PL/SQL 块中嵌入 SELECT、DML 语句和事务控制语句，在学习了本章之后，读者应该学会：

- 在 PL/SQL 块中检索单行数据；
- 在 PL/SQL 块中嵌入 DML 语句；
- 使用 SQL 游标属性；
- 在 PL/SQL 块中嵌入事务控制语句。

## 6.1 检索单行数据

通过在 PL/SQL 块中嵌入 SELECT 语句，可以将数据库数据检索到变量中，然后可以输出或处理该变量的数据。注意，当在 PL/SQL 块中嵌入 SELECT 语句时，必须要带有 INTO 子句，语法如下：

```
SELECT select_list
      INTO {variable_name[,variable_name]... | record_name}
     FROM table
    WHERE condition;
```

如上所示，`select_list` 用于指定选择列表（列或表达式）；`variable_name` 用于指定接收结果的标量变量名；而 `record_name` 则用于指定接收结果的记录变量名，`table` 用于指定表或视图名；而 `condition` 则用于指定条件子句。注意，当在 PL/SQL 块中直接使用 `SELECT INTO` 语句时，该语句必须返回一条数据，并且只能返回一条数据。如果 `SELECT` 语句返回了多行数据，或者没有返回数据，那么会触发例外并显示错误信息。

### 1. 使用标量变量接收数据

使用标量变量接收 `SELECT` 语句的输出结果时，变量个数必须要与 `SELECT` 语句后的选择列表个数完全一致，而变量类型和长度必须要与相应选择列表项的数据类型和长度匹配。示例如下：

```
DECLARE
  v_ename emp.ename%TYPE;
  v_sal   emp.sal%TYPE;
BEGIN
  SELECT ename,sal INTO v_ename,v_sal
  FROM emp WHERE empno=&no;
  dbms_output.put_line('雇员名:'||v_ename);
  dbms_output.put_line('雇员薪水:'||v_sal);
```

```

END;
/
输入 no 的值: 7788
雇员名:SCOTT
雇员薪水:1500

```

如上例所示，因为选择列表包含两个列（ename 和 sal），所以 INTO 子句后必须要使用两个变量接收结果（v\_ename 和 v\_sal），并且它们的类型和长度与 ename 列、sal 列的类型和长度完全一致。

## 2. 使用记录变量接收数据

使用记录变量接收数据时，记录成员的个数必须要与选择列表项的个数完全一致，并且记录成员的类型和长度一定要与相应选择列表项的数据类型和长度相匹配。示例如下：

```

DECLARE
    TYPE emp_record_type IS RECORD (
        ename emp.ename%TYPE, sal emp.sal%TYPE);
    emp_record emp_record_type;
BEGIN
    SELECT ename,sal INTO emp_record
    FROM emp WHERE empno=&no;
    dbms_output.put_line('雇员名:'||emp_record.ename);
    dbms_output.put_line('雇员薪水:'||emp_record.sal);
END;
/
输入 no 的值: 7788
雇员名:SCOTT
雇员薪水:1500

```

如上所示，因为选择列表项包含两个列 ename 和 sal，所以记录类型 emp\_record\_type 应该包含两个记录成员 ename 和 sal，并且它们的类型与 ename 列和 sal 列完全一致。

## 3. 嵌入 SELECT 语句注意事项

当在 PL/SQL 块中直接使用 SELECT INTO 语句时，该语句必须要返回一条数据，并且只能返回一条数据，否则会触发 PL/SQL 例外，或显示错误信息。

### (1) NO\_DATA\_FOUND 例外

SELECT INTO 语句没有返回任何数据时，会触发 NO\_DATA\_FOUND 例外。如果没有在外处理部分捕捉并处理该例外，则会将错误消息传递到调用环境。示例如下：

```

DECLARE
    v_ename emp.ename%TYPE;
    v_sal   emp.sal%TYPE;
BEGIN
    SELECT ename,sal INTO v_ename,v_sal
    FROM emp WHERE empno=&no;
    dbms_output.put_line('雇员名:'||v_ename);
    dbms_output.put_line('雇员薪水:'||v_sal);
END;
/
输入 no 的值: 1111

```

```

DECLARE
*
ERROR 位于第 1 行:
ORA-01403: 未找到数据
ORA-06512: 在 line 5

```

在执行了以上 PL/SQL 块之后，因为不存在雇员号为 1111 的雇员，并且没有捕捉 NO\_DATA\_FOUND 例外，所以会在调用环境中显示错误信息。关于如何捕捉并处理例外错误，请参见后面章节的内容。

### (2) TOO\_MANY\_ROWS 例外

当 SELECT INTO 语句返回多条数据时，会触发 TOO\_MANY\_ROWS 例外。如果没有在例外处理部分捕捉并处理该例外，则会将错误消息传递到调用环境。示例如下：

```

DECLARE
  v_ename emp.ename%TYPE;
  v_sal   emp.sal%TYPE;
BEGIN
  SELECT ename,sal INTO v_ename,v_sal
  FROM emp WHERE deptno=&no;
END;
/
输入 no 的值: 10
DECLARE
*
ERROR 位于第 1 行:
ORA-01422: 实际返回的行数超出请求的行数
ORA-06512: 在 line 5

```

如上所示，因为部门 10 存在多个雇员，所以当执行 SELECT INTO 语句时会触发 TOO\_MANY\_ROWS 例外。但因为没有捕捉并处理该例外，所以会在调用环境中显示错误消息。

### (3) WHERE 子句使用注意事项

在 WHERE 子句中使用变量时注意，变量名不能与列名相同，否则会触发 TOO\_MANY\_ROWS 例外。如果没有在例外处理部分捕捉并处理该例外，则会将错误消息传递到调用环境。示例如下：

```

DECLARE
  empno NUMBER(6):=7788;
  v_ename VARCHAR2(10);
BEGIN
  SELECT ename INTO v_ename FROM emp
  WHERE empno=empno;
END;
/
DECLARE
*
ERROR 位于第 1 行:
ORA-01422: 实际返回的行数超出请求的行数
ORA-06512: 在 line 5

```

当执行以上 PL/SQL 块时，会在调用环境中显示错误消息，原因是 WHERE 子句使用了同名的列和变量（empno）。因此，在编写 PL/SQL 程序时一定要注意，变量名不应该与列名相同。

## 6.2 操纵数据

在 PL/SQL 块中不仅可以嵌入 SELECT 语句，也可以嵌入 DML 语句。通过嵌入 INSERT 语句，可以将数据插入到 Oracle 数据库中；通过嵌入 UPDATE 语句，可以更新数据库数据；通过嵌入 DELETE 语句，可以删除数据库数据。另外，通过使用 SQL 游标属性，还可以取得 DML 语句所作用的行数。下面将介绍如何在 PL/SQL 块中插入数据、更新数据和删除数据。

### 6.2.1 插入数据

在 PL/SQL 块中插入数据是通过嵌入 INSERT 语句来完成的，插入数据的语法与在 SQL\*Plus 中直接执行 INSERT 语句没有任何区别，只不过在提供数值时可以使用 PL/SQL 变量。当使用 INSERT 语句插入数据时，既可以使用 VALUES 子句，也可以使用子查询。插入数据的语法如下：

```
INSERT INTO <table> [(column[, column,...])]
VALUES (value[, value,...])

INSERT INTO <table> [(column[, column,...])]
SubQuery
```

#### 示例一：在 PL/SQL 块中使用 VALUES 子句插入数据

在 PL/SQL 块中使用 VALUES 子句插入数据时，每次只能插入一条数据。另外大家需注意，在使用 INSERT 语句插入数据时，必须要为表的主键列和 NOT NULL 列提供数据。示例如下：

```
DECLARE
    v_deptno dept.deptno%TYPE;
    v_dname  dept.dname%TYPE;
BEGIN
    v_deptno:=&no;
    v_dname:='&name';
    INSERT INTO dept (deptno,dname)
    VALUES(v_deptno,v_dname);
END;
/
输入 no 的值: 50
输入 name 的值: SALES
```

在执行了以上 PL/SQL 块之后，会根据提示输入的部门号和部门名称为 DEPT 表插入相应数据。例如，上例插入了部门号为 50、部门名称为 SALES 的新部门。

#### 示例二：在 PL/SQL 块中使用子查询插入数据

使用子查询插入数据实际是将一张表的数据复制到另一张表中，当在 PL/SQL 块中使用子查询插入数据时，INSERT 语句后列的数据类型和个数必须要与子查询列的数据类型和个数完全匹配。示例如下：

```

DECLARE
    v_deptno emp.deptno%TYPE:=&no;
BEGIN
    INSERT INTO employee
    SELECT * FROM emp WHERE deptno=v_deptno;
END;
/
输入 no 的值: 10

```

当执行了以上 PL/SQL 块之后，会根据输入的部门号将相应部门的雇员数据复制到 EMPLOYEE 表中。例如，上例将部门 10 的雇员信息插入到了表 EMPLOYEE 中。

### 6.2.2 更新数据

在 PL/SQL 块中更新数据是使用 UPDATE 语句来完成的，更新数据的语法与在 SQL\*Plus 中直接执行 UPDATE 语句没有任何区别，只不过在提供数值时可以使用 PL/SQL 变量。当使用 UPDATE 语句时，既可以使用表达式更新列值，也可以使用子查询更新一列或多列的数据。更新数据的语法如下：

```

UPDATE <table|view>
SET <column> = <value> [, <column> = <value>]
[WHERE <condition>];

```

#### 示例一：使用表达式更新列值

在使用 UPDATE 语句更新数据时，列值必须要满足数据类型、长度以及各种约束规则。示例如下：

```

DECLARE
    v_deptno dept.deptno%TYPE:=&no;
    v_loc dept.loc%TYPE:='&loc';
BEGIN
    UPDATE dept SET loc=v_loc
    WHERE deptno=v_deptno;
END;
/
输入 no 的值: 50
输入 loc 的值: BEIJING

```

在执行了以上的 PL/SQL 块之后，会根据输入的部门号和部门位置更新特定部门的部门位置。例如，上例将部门 50 的部门位置修改为 BEIJING。

#### 示例二：使用子查询更新列值

通过在 UPDATE 语句中使用子查询，可以更新关联数据，从而降低网络开销，提高数据操纵性能。示例如下：

```

DECLARE
    v_ename emp.ename%TYPE:='&name';
BEGIN
    UPDATE emp SET (sal,comm)=
    (SELECT sal,comm FROM emp WHERE ename=v_ename)
    WHERE job=(SELECT job FROM emp WHERE ename=v_ename);

```

```

END;
/
输入 name 的值: SCOTT

```

在执行了以上的 PL/SQL 块之后，会根据输入的雇员名更新与该雇员岗位完全相同的所有雇员的岗位和补助。例如，上例将与 SCOTT 岗位相同的所有雇员的工资和补助更新为与 SCOTT 完全相同。

### 6.2.3 删除数据

在 PL/SQL 块中删除数据是使用 DELETE 语句来完成的，删除数据的语法与在 SQL\*Plus 中直接执行 DELETE 语句没有任何区别，只不过在提供数值时可以使用 PL/SQL 变量。删除数据的语法如下：

```
DELETE FROM <table|view> [WHERE <condition>];
```

在使用 DELETE 语句删除数据时，在 WHERE 子句中既可以直接使用变量或数值，也可以使用子查询。注意，如果在从表中存在关联数据，那么在删除主表相应数据行时会显示错误信息。例如如果部门 10 存在雇员，那么在删除部门 10 时会显示错误信息，并且该部门的数据不会被删除。

#### 示例一：使用变量删除数据

当使用 DELETE 语句删除数据时，如果不指定 WHERE 子句，那么会删除表中的所有数据。在 WHERE 子句中使用变量删除数据的示例如下：

```

DECLARE
    v_deptno dept.deptno%TYPE:=&no;
BEGIN
    DELETE FROM dept WHERE deptno=v_deptno;
END;
/
输入 no 的值: 50

```

当执行了以上的 PL/SQL 块之后，会根据输入的部门号删除特定部门。例如，上例的运行结果会删除部门 50。

#### 示例二：使用子查询删除数据

在 WHERE 子句中不仅可以直接使用变量或数值，也可以使用子查询来删除数据。示例如下：

```

DECLARE
    v_ename emp.ename%TYPE:='&name';
BEGIN
    DELETE FROM emp
    WHERE deptno=(SELECT deptno FROM emp
                  WHERE ename=v_ename);
END;
/
输入 name 的值: SCOTT

```

当执行了以上的 PL/SQL 块之后，会根据输入的雇员名解雇雇员所在部门的所有雇员。例如，上例会删除 SCOTT 所在部门的所有雇员。

#### 6.2.4 SQL 游标

当执行 SELECT, INSERT, UPDATE 以及 DELETE 语句时, Oracle Server 会为这些 SQL 语句分配相应的上下文区 (Context Area)。并且 Oracle 使用上下文区解析并执行相应的 SQL 语句, 而游标是指向上下文区的指针。在 Oracle 数据库中, 游标包括隐含游标和显式游标两种类型。其中隐含游标又被称为 SQL 游标, 它专门用于处理 SELECT INTO, INSERT, UPDATE 以及 DELETE 语句; 而显式游标则用于处理多行的 SELECT 语句。当在 PL/SQL 块中执行 INSERT、UPDATE 以及 DELETE 语句时, 为了取得 DML 语句作用的结果, 必须要使用 SQL 游标属性, SQL 游标包括 SQL%FOUND, SQL%NOTFOUND, SQL%ROWCOUNT, SQL%ISOPEN 等四种属性。

##### 1. SQL%ISOPEN

游标属性 SQL%ISOPEN 用于确定 SQL 游标是否已经打开。当在 PL/SQL 块中执行 SELECT INTO, INSERT, UPDATE 以及 DELETE 语句时, Oracle 会隐含地打开游标, 并且在语句执行完成之后会隐含地关闭游标, 所以对于开发人员来说该属性的值永远都是 FALSE, 并且在开发 PL/SQL 应用时不需要使用该游标属性。

##### 2. SQL%FOUND

游标属性 SQL%FOUND 用于确定 SQL 语句执行是否成功。注意, SQL 语句执行是否成功是根据是否有作用行来判断, 当 SQL 语句有作用行时, 其属性值为 TRUE; 当 SQL 语句没有作用行时, 其属性值为 FALSE。使用该属性的示例如下:

```

DECLARE
    v_deptno emp.deptno%TYPE:=&no;
BEGIN
    UPDATE emp SET sal=sal*1.1
    WHERE deptno=v_deptno;
    IF SQL%FOUND THEN
        dbms_output.put_line('语句执行成功');
    ELSE
        dbms_output.put_line('该部门不存在雇员');
    END IF;
END;
/
输入 no 的值: 30
语句执行成功

```

当执行了以上 PL/SQL 块之后, 会根据输入的部门号更新相应部门的雇员工资。例如, 上例会更新部门 30 的所有雇员的工资; 而如果输入了不存在的部门号, 则 UPDATE 语句不会修改任何行, 并显示“该部门不存在雇员”。

##### 3. SQL%NOTFOUND

游标属性 SQL%NOTFOUND 用于确定 SQL 语句执行是否成功。注意, SQL 语句执行是否成功是根据是否有作用行来判断, 当 SQL 语句有作用行时, 其属性值为 FALSE; 当 SQL 语句没有作用行时, 其属性值为 TRUE。使用该属性的示例如下:

```
DECLARE
```

```

    v_deptno emp.deptno%TYPE:=&no;
BEGIN
    UPDATE emp SET sal=sal*1.1
    WHERE deptno=v_deptno;
    IF SQL%NOTFOUND THEN
        dbms_output.put_line('该部门不存在雇员');
    ELSE
        dbms_output.put_line('语句执行成功');
    END IF;
END;
/
输入 no 的值: 50
该部门不存在雇员

```

当执行了以上 PL/SQL 块之后，会根据输入的部门号更新相应部门的雇员工资。例如，因为部门 50 不存在，所以会显示“该部门不存在雇员”；而如果输入了存在的部门号，则 UPDATE 语句会成功执行，并显示“语句执行成功”。

#### 4. SQL%ROWCOUNT

游标属性 SQL%ROWCOUNT 用于返回 SQL 语句所作用的总计行数，示例如下：

```

DECLARE
    v_deptno emp.deptno%TYPE:=&no;
BEGIN
    UPDATE emp SET sal=sal*1.1
    WHERE deptno=v_deptno;
    dbms_output.put_line('修改了'||SQL%ROWCOUNT||'行');
END;
/
输入 no 的值: 30
修改了 6 行

```

当执行了以上的 PL/SQL 块之后，会根据输入的部门号更新特定部门的雇员工资，并显示被修改的行数信息。例如，上例更新了部门 30 的所有雇员工资，并且输出了被更新的行数。如果输入不存在的部门号，则会显示“修改了 0 行”。

### 6.3 事务控制语句

当编写 PL/SQL 程序时，不仅可以直接嵌入 SELECT 和 DML 语句，也可以直接嵌入事务控制语句。在 Oracle 数据库中，事务控制语句包括 COMMIT、ROLLBACK 以及 SAVEPOINT 等三种语句，在 PL/SQL 块中使用事务控制语句与在 SQL\*Plus 中直接使用事务控制语句没有任何区别。

#### 示例一：在 PL/SQL 块中使用 COMMIT 和 ROLLBACK 语句

COMMIT 语句用于提交事务，从而确认事务变化；ROLLBACK 语句用于回退事务，从而取消事务变化。在 PL/SQL 块中使用 COMMIT 和 ROLLBACK 语句的示例如下：

```

DECLARE
    v_sal emp.sal%TYPE:=-salary;

```

```

    v_ename emp.ename%TYPE:='&name';
BEGIN
    UPDATE emp SET sal=v_sal WHERE ename=v_ename;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;
/
输入 salary 的值: 1200
输入 name 的值: SCOTT

```

在执行了以上 PL/SQL 块之后，会根据输入的雇员名和雇员工资更新特定雇员的工资，并且在更新了工资之后提交事务。而如果执行该 PL/SQL 块出现例外，则会取消事务变化。

#### 示例二：在 PL/SQL 块中使用 ROLLBACK 和 SAVEPOINT 语句

SAVEPOINT 语句用于设置保存点，通过与 ROLLBACK 语句结合使用可以取消部分事务。在 PL/SQL 块中使用 ROLLBACK 和 SAVEPOINT 语句的示例如下：

```

BEGIN
    INSERT INTO temp VALUES(1);
    SAVEPOINT a1;
    INSERT INTO temp VALUES(2);
    SAVEPOINT a2;
    INSERT INTO temp VALUES(3);
    SAVEPOINT a3;
    ROLLBACK TO a2;
    COMMIT;
END;
/

```

当执行了以上 PL/SQL 块之后，应该为 TEMP 表实际插入了几条数据呢？

## 6.4 习题

1. 在 PL/SQL 块中不能直接嵌入以下哪些语句？
  - A. SELECT
  - B. INSERT
  - C. CREATE TABLE
  - D. GRANT
  - E. COMMIT
2. 当 SELECT INTO 语句没有返回行时，会触发以下哪种例外？
  - A. TOO\_MANY\_ROWS
  - B. VALUE\_ERROR
  - C. NO\_DATA\_FOUND
3. 当执行 UPDATE 语句时没有更新任何行，会触发以下哪种例外？
  - A. VALUE\_ERROR
  - B. NO\_DATA\_FOUND
  - C. 不会触发任何例外
4. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户编号、客户名称，为 CUSTOMER 表插入一条数据，并提交事务。
  - 客户编号：100
  - 客户名称：SCOTT

5. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户名称（不区分大小写）、城市，并更新该客户所在城市

- 客户名称: scott
- 城市: NEW YORK

6. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户编号，并使用 DBMS\_OUTPUT 包显示客户名称及所在城市

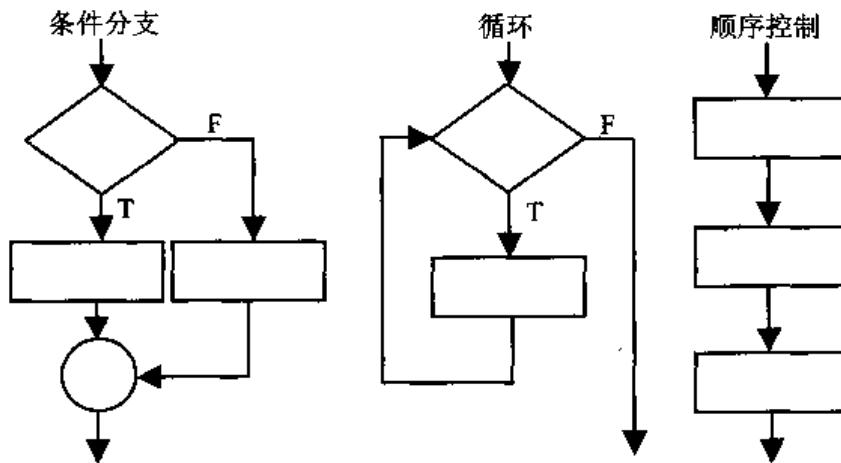
- 客户编号: 100

7. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户名称（不区分大小写），删除该客户的信息，并使用 SQL 游标属性确定删除了几行数据

- 客户名称: scott

## 第7章 编写控制结构

编写计算机应用程序时，任何计算机语言（例如 C, JAVA, COBOL 等）都能处理各种基本的控制结构：条件分支结构、循环结构和顺序控制结构等，PL/SQL 也不例外。它不仅可以嵌入 SQL 语句，而且还支持条件分支语句（IF, CASE）、循环语句（LOOP）和顺序控制语句（GOTO, NULL）等。如下图所示：



本章将详细介绍如何在 PL/SQL 块中编写各种控制结构，在学习了本章之后，读者应该学会：

- 使用各种 IF 语句；
- 使用 Oracle9i 的新特征——CASE 语句；
- 使用各种类型的循环语句；
- 使用顺序控制语句——GOTO 和 NULL。

### 7.1 条件分支语句

条件分支语句用于依据特定情况选择要执行的操作。在 Oracle9i 之前执行条件分支操作都需要使用 IF 语句来完成，并且 PL/SQL 提供了三种条件分支语句：IF-THEN, IF-THEN-ELSE, IF-THEN-ELSIF。语法如下：

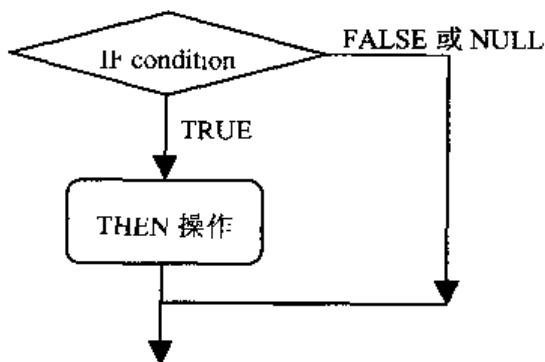
```
IF condition THEN
    statements;
[ELSIF condition THEN
    statements;]
[ELSE
    statements;]
END IF;
```

如上所示，当使用条件分支语句时，不仅可以使用 IF 语句进行简单条件判断，而且还可以使用 IF 语句进行二重分支和多重分支判断。注意，ELSIF 和 END IF 可能与其他语言的语言

法有所不同，ELSIF 为一个单词，而 END IF 则是两个单词。

### 1. 简单条件判断

简单条件判断用于执行单一条件判断。如果满足特定条件，则会执行相应操作；如果不满足条件，则退出条件分支语句。简单条件判断是使用 IF-THEN 语句来完成的，其执行流程图如下图所示：



如图中所示，当使用简单条件判断时，如果 condition 为 TRUE，那么 PL/SQL 执行器会执行 THEN 后的操作；如果 condition 为 FALSE 或 NULL，那么 PL/SQL 执行器会直接退出条件分支语句。示例如下：

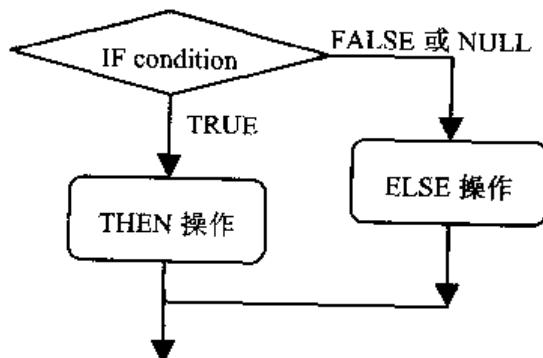
```

DECLARE
  v_sal NUMBER(6,2);
BEGIN
  SELECT sal INTO v_sal FROM emp
  WHERE lower(ename)=lower('&&name');
  IF v_sal<2000 THEN
    UPDATE emp SET sal=v_sal+200
    WHERE lower(ename)=lower('&name');
  END IF;
END;
/
输入 name 的值: miller
  
```

如上所示，在执行以上 PL/SQL 块时，首先会提示输入雇员名，然后取得雇员工资；如果雇员工资低于 2000，则为其增加 200 元的工资。

### 2. 二重条件分支

二重条件分支是指根据条件来选择两种可能性。当使用二重条件分支时，如果满足条件，则执行一组操作；如果不满足条件，则执行另外一组操作。二重条件分支是使用 IF ... THEN ... ELSE 来完成的，其执行流程图如下所示：



如上所示，在执行 IF ... THEN ... ELSE 语句时，如果 condition 为 TRUE，那么 PL/SQL 执行器会执行 THEN 后的操作；如果 condition 为 FALSE 或 NULL，那么 PL/SQL 执行器会执行 ELSE 后的操作。示例如下：

```

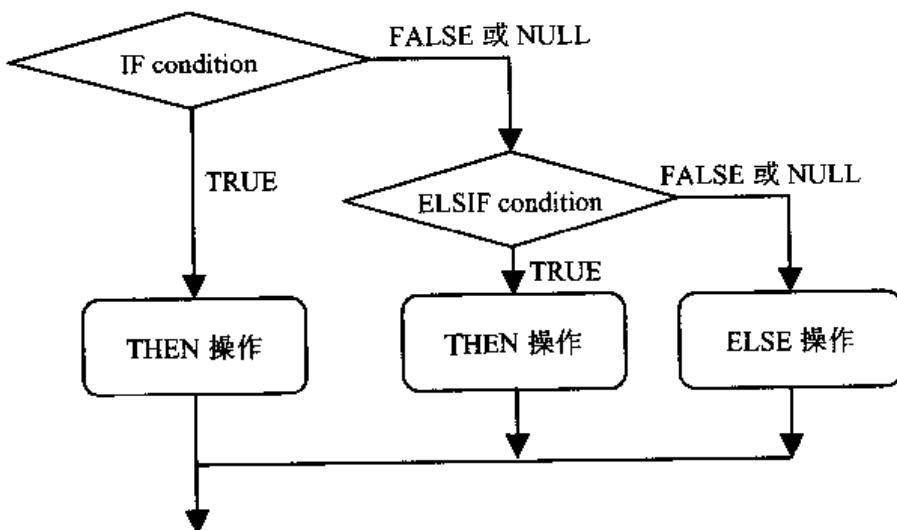
DECLARE
    v_comm NUMBER(6,2);
BEGIN
    SELECT comm INTO v_comm FROM emp
    WHERE empno=&&no;
    IF v_comm<>0 THEN
        UPDATE emp SET comm=v_comm+100
        WHERE empno=&no;
    ELSE
        UPDATE emp SET comm=200
        WHERE empno=&no;
    END IF;
END;
/
输入 no 的值: 7934

```

如上所示，在执行了以上 PL/SQL 块之后，如果雇员补助不是 0，则在原来的基础上增加 100 元；如果补助为 0 或 NULL 时，则设置其补助为 200 元。

### 3. 多重条件分支

多重条件分支用于执行最复杂的条件分支操作。当使用多重条件分支时，如果满足第一个条件，则执行第一种操作；如果不满足第一个条件，则检查是否满足第二个条件，如果满足第二个条件，则执行第二种操作；如果不满足第二个条件，则检查是否满足第三个条件，依此类推。多重条件分支是使用 IF ... THEN ... ELSIF 语句来完成的，其执行流程图如下图所示：



如图中所示，当执行 IF ... THEN ... ELSIF 语句时，如果第一个条件为 TRUE，那么 PL/SQL 执行器会执行第一个 THEN 后的操作；如果第一个条件为 FALSE 或 NULL，那么 PL/SQL 执行器会判断第二个条件 (ELSIF)；如果第二个条件为 TRUE，那么 PL/SQL 执行器会执行第二个 THEN 后的操作，依此类推。如果所有条件都为 FALSE 或 NULL，那么 PL/SQL 执行器会

执行 ELSE 后的操作。示例如下：

```

  undefined no
  DECLARE
    v_job VARCHAR2(10);
    v_sal NUMBER(6,2);
  BEGIN
    SELECT job,sal INTO v_job,v_sal
    FROM emp WHERE empno=&no;
    IF v_job='PRESIDENT' THEN
      UPDATE emp SET sal=v_sal+1000 WHERE empno=&no;
    ELSIF v_job='MANAGER' THEN
      UPDATE emp SET sal=v_sal+500 WHERE empno=&no;
    ELSE
      UPDATE emp SET sal=v_sal+200 WHERE empno=&no;
    END IF;
  END;
/
输入 no 的值: 7788

```

如上所示，在执行以上 PL/SQL 块时，如果雇员岗位为“PRESIDENT”，则为其增加 1000 元的工资；如果雇员岗位为“MANAGER”，则为其增加 500 元的工资；而对于其他岗位的雇员，则为其增加 200 元的工资。

## 7.2 CASE 语句

在 Oracle9i 之前，执行多重条件分支操作是使用 IF 语句来完成的；从 Oracle9i 开始，不仅可以用 IF 语句执行多重条件分支操作，也可以使用 CASE 语句来执行多重条件分支操作。当处理多重条件分支时，使用 CASE 语句更加简捷，而且执行效率也更好，所以建议大家使用 CASE 语句。使用 CASE 语句处理多重条件分支有两种方法，第一种方法是使用单一选择符进行等值比较；第二种方法是使用多种条件进行非等值比较。下面分别介绍如何使用这两种方法。

### 1. 在 CASE 语句中使用单一选择符进行等值比较

当使用 CASE 语句执行多重条件分支时，如果条件选择符完全相同，并且条件表达式为相等条件选择，那么可以选择使用单一条件选择符进行等值比较。语法如下：

```

CASE selector
  WHEN expression1 THEN sequence_of_statements1;
  WHEN expression2 THEN sequence_of_statements2;
  ...
  WHEN expressionN THEN sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE;

```

如上所示，selector 用于指定条件选择符，expression 用于指定条件值的表达式，sequence\_of\_statement 用于指定要执行的条件操作。如果设置的所有条件都不满足，则会执行 ELSE 后的语句。注意，为了避免 CASE\_NOT\_FOUND 例外，在编写 CASE 语句时应该带有

ELSE 子句。示例如下：

```

DECLARE
    v_deptno emp.deptno%TYPE;
BEGIN
    v_deptno:=&no;
    CASE v_deptno
        WHEN 10 THEN
            UPDATE emp SET comm=100 WHERE deptno=v_deptno;
        WHEN 20 THEN
            UPDATE emp SET comm=80 WHERE deptno=v_deptno;
        WHEN 30 THEN
            UPDATE emp SET comm=50 WHERE deptno=v_deptno;
        ELSE
            dbms_output.put_line('不存在该部门');
    END CASE;
END;
/
输入 no 的值: 10

```

执行以上 PL/SQL 块时，会提示用户输入部门号，并更新相应部门的雇员补助。如果输入了除 10、20、30 之外的其他部门，则会显示信息“不存在该部门”。

## 2. 在 CASE 语句中使用多种条件比较

当使用单一条件选择符进行等值比较时，可以使用 CASE selector 语法来实现。如果包含有多种条件进行不等比较，那么必须在 WHEN 子句中指定比较条件。语法如下：

```

CASE
    WHEN search_condition1 THEN sequence_of_statements1;
    WHEN search_condition2 THEN sequence_of_statements2;
    ...
    WHEN search_conditionN THEN sequence_of_statementsN;
    [ELSE sequence_of_statementsN+1;]
END CASE;

```

如上所示，search\_condition 用于指定不同比较条件，sequence\_of\_statement 用于指定当满足特定条件时要执行的操作。如果设置的所有条件都得不到满足，则会执行 ELSE 后的语句。大家需要注意，为了避免 CASE\_NOT\_FOUND 例外，在编写 CASE 语句时应该带有 ELSE 子句。使用多种条件比较的示例如下：

```

DECLARE
    v_sal emp.sal%TYPE;
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename,sal INTO v_ename,v_sal
    FROM emp WHERE empno=&no;
    CASE
        WHEN v_sal<1000 THEN
            UPDATE emp SET comm=100 WHERE ename=v_ename;
        WHEN v_sal<2000 THEN

```

```

        UPDATE emp SET comm=80 WHERE ename=v_ename;
        WHEN v_sal<6000 THEN
            UPDATE emp SET comm=50 WHERE ename=v_ename;
        END CASE;
    END;
/
输入 no 的值: 7788

```

执行以上 PL/SQL 块时，会根据输入的雇员号将雇员名、工资分别存放到变量 `v_ename`、`v_sal` 中，然后会根据 CASE 条件更新其补助。

### 7.3 循环语句

为了在编写的 PL/SQL 块中重复执行一条语句或者一组语句，可以使用循环控制结构。编写循环控制结构时，用户可以使用基本循环、WHILE 循环和 FOR 循环等三种类型的循环语句，下面分别介绍使用这三种循环语句的方法。

#### 1. 基本循环

在 PL/SQL 中最简单格式的循环语句是基本循环语句，这种循环语句以 LOOP 开始，以 END LOOP 结束，其语法如下：

```

LOOP
    statement1;
    .
    .
    EXIT [WHEN condition];
END LOOP;

```

如上所示，当使用基本循环时，无论是否满足条件，语句至少会被执行一次。当 `condition` 为 TRUE 时，会退出循环，并执行 END LOOP 后的相应操作。注意，当编写基本循环时，一定要包含 EXIT 语句，否则 PL/SQL 块会陷入死循环；另外当使用基本循环时，大家还应该定义循环控制变量，并且在循环体内修改循环控制变量的值。下面是使用基本循环的示例：

```

SQL> CREATE TABLE temp(cola INT);
表已创建。
DECLARE
    i INT:=1;
BEGIN
    LOOP
        INSERT INTO temp VALUES(i);
        EXIT WHEN i=10;
        i:=i+1;
    END LOOP;
END;
/

```

如上所示，在执行以上 PL/SQL 块时，会使用基本循环为 TEMP 表插入 10 条数据(1, 2, ..., 10)，并且当 `i=10` 时会退出循环。

## 2. WHILE 循环

基本循环至少要执行一次循环体内的语句，而对于 WHILE 循环来说，只有条件为 TRUE 时，才会执行循环体内的语句。WHILE 循环以 WHILE ... LOOP 开始，以 END LOOP 结束，其语法如下：

```
WHILE condition LOOP
    statement1;
    statement2;
    .
    .
    .
    END LOOP;
```

如上所示，当 condition 为 TRUE 时，PL/SQL 执行器会执行循环体内的语句；而当 condition 为 FALSE 或 NULL 时，会退出循环，并执行 END LOOP 后的语句。注意，当使用 WHILE 循环时，应该定义循环控制变量，并在循环体内改变循环控制变量的值。使用 WHILE 循环的示例如下：

```
DECLARE
    i INT :=1;
BEGIN
    WHILE i<=10 LOOP
        INSERT INTO temp VALUES(i);
        i:=i+1;
    END LOOP;
END;
/
```

如上所示，当执行以上 PL/SQL 块时，会使用 WHILE 循环为 TEMP 表插入 10 条记录（1, 2, ..., 10），当 i=11 时会退出循环体。

## 3. FOR 循环

当使用基本循环或 WHILE 循环时，需要定义循环控制变量，并且循环控制变量不仅可以使用 NUMBER 类型，也可以使用其他数据类型；而当使用 FOR 循环时，Oracle 会隐含定义循环控制变量。FOR 循环的语法如下：

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
        statement1;
        statement2;
        .
        .
        .
    END LOOP;
```

如上所示，counter 是循环控制变量，并且该变量由 Oracle 隐含定义，不需要显式定义；lower\_bound 和 upper\_bound 分别对应于循环控制变量的下界值和上界值。默认情况下，当使用 FOR 循环时，每次循环时循环控制变量会自动增一；如果指定 REVERSE 选项，那么每次循环时循环控制变量会自动减一。使用 FOR 循环的示例如下：

```
BEGIN
    FOR i IN REVERSE 1..10 LOOP
        INSERT INTO temp VALUES(i);
    END LOOP;
END;
/
```

如上所示，当执行以上 PL/SQL 块时，会为 TEMP 表插入 10 条记录。因为指定了 REVERSE 选项，所以被插入数据的顺序为 10, 9..., 1。

#### 4. 嵌套循环和标号

嵌套循环是指在一个循环语句之中嵌入另一个循环语句，而标号（Label）则用于标记嵌套块或嵌套循环。通过在嵌套循环中使用标号，可以区分内层循环和外层循环，并且可以在内层循环中直接退出外层循环。在编写 PL/SQL 块时，可以使用<<label\_name>>定义标号。示例如下：

```

DECLARE
    result INT;
BEGIN
    <<outer>>
    FOR i IN 1..100 LOOP
        <<inner>>
        FOR j IN 1..100 LOOP
            result:=i*j;
            EXIT outer WHEN result=1000;
            EXIT WHEN result=500;
        END LOOP inner;
        dbms_output.put_line(result);
    END LOOP outer;
    dbms_output.put_line(result);
END;
/

```

如上所示，当执行以上 PL/SQL 块时，如果 result=1000，那么会直接退出外层循环，而当 result=500 时只会退出内层循环。

## 7.4 顺序控制语句

PL/SQL 不仅提供了条件分支语句和循环控制语句，而且还提供了顺序控制语句 GOTO 和 NULL。但与 IF, CASE 和 LOOP 语句不同，GOTO 语句和 NULL 语句，一般情况下不要使用。下面简单介绍如何在 PL/SQL 程序中使用 GOTO 语句和 NULL 语句。

### 1. GOTO

GOTO 语句用于跳转到特定标号处去执行语句。注意，因为使用 GOTO 语句会增加程序的复杂性，并且使得应用程序可读性变差，所以开发应用程序时，一般都建议用户不要使用 GOTO 语句。其语法如下：

```
GOTO label_name;
```

其中，label\_name 是已经定义的标号名。注意，当使用 GOTO 跳转到特定标号时，标号后至少要包含一条可执行语句。使用 GOTO 语句的示例如下：

```

DECLARE
    i INT:=1;
BEGIN
    LOOP

```

```

    INSERT INTO temp VALUES(i);
    IF i=10 THEN
        GOTO end_loop;
    END IF;
    i:=i+1;
    END LOOP;
<<end_loop>>
    dbms_output.put_line('循环结束');
END;
/
循环结束

```

如上所示，在执行以上 PL/SQL 块时，如果 i=10，则会跳转到标号 end\_loop，并执行随后的语句。

## 2. NULL

NULL 语句不会执行任何操作，并且会直接将控制传递到下一条语句。使用 NULL 语句的主要好处是可以提高 PL/SQL 程序的可读性，示例如下：

```

DECLARE
    v_sal emp.sal%TYPE;
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename,sal INTO v_ename,v_sal
    FROM emp WHERE empno=&no;
    IF v_sal<3000 THEN
        UPDATE emp SET comm=sal*0.1 WHERE ename=v_ename;
    ELSE
        NULL;
    END IF;
END;
/
输入 no 的值： 7369

```

当执行以上 PL/SQL 块时，会根据输入的雇员号确定雇员名及其工资；如果雇员工资低于 3000，则将其补助设置为工资的 10%；如果雇员工资高于 3000，则不会执行任何操作(NULL)。

## 7.5 习题

### 1. 请查看以下 IF 语句：

```

DECLARE
    Sal NUMBER:=500;
    Comm NUMBER;
Begin
    IF Sal<100 then
        Comm:=0;
    ELSIF sal<600 then
        Comm:=sal*0.1;
    END IF;

```

```

ELSIF sal<1000 then
    Comm:=sal*0.15;
ELSE
    Comm:=sal*0.2;
ENDIF;
END;

```

在执行了以上语句之后，变量 COMM 的结果应是：

- A. 0      B. 50      C. 75      D. 100

2. 请查看以下 CASE 语句：

```

DECLARE
    v_sal NUMBER:=1000;
    v_tax NUMBER;
Begin
    CASE
        WHEN v_sal<1500 THEN
            v_tax:=v_sal*0.03;
        WHEN v_sal<2500 THEN
            v_tax:=v_sal*0.04;
        WHEN v_sal<3500 THEN
            v_tax:=v_sal*0.05;
        WHEN v_sal<8000 THEN
            v_tax:=v_sal*0.08;
    END CASE;
END;
/

```

当执行了以上 PL/SQL 块之后，变量 v\_tax 的值应为

- A. 30      B. 40      C. 50      D. 80

3. 首先执行 SQL 语句“CREATE TABLE temp (cola INT)”建立 TEMP 表，然后编写 PL/SQL 块为该表插入 8 条数据（从 1~10，但排除 5~7）。

4. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户名（不区分大小写）和所在城市，并修改客户所在城市。如果客户不存在，则显示消息“该客户不存在”。格式如下：

输入 customer\_name 的值： clark

输入 city 的值： Boston

5. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入部门号，然后使用 CASE 语句判断条件并更新雇员工资

如果部门号为 10，则为其雇员增加 10% 的工资；

如果部门号为 20，则为其雇员增加 8% 的工资；

如果部门号为 30，则为其雇员增加 5% 的工资；

如果输入其他数字，则显示“该部门不存在”。

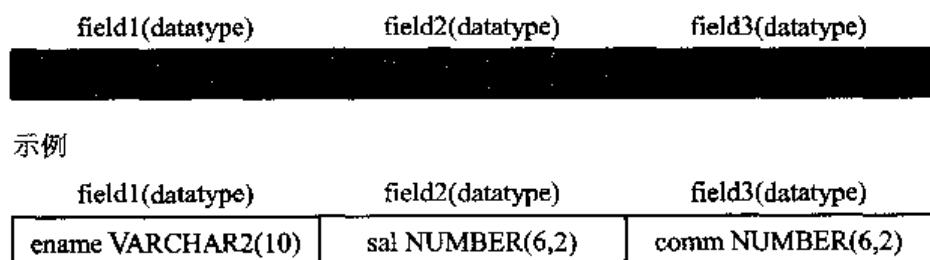
# 第8章 使用复合数据类型

当使用标量变量处理 Oracle 数据时，每个标量变量只能存放单个值，也就是说只能处理单行单列的数据。如果要使用标量变量处理单行多列数据，那么必须要定义多个变量接收不同列的数据；为了简化单行多列数据的处理，可以使用 PL/SQL 记录；为了保留并处理多行单列的数据，可以使用索引表、嵌套表和 VARRAY；为了处理多行多列的数据，应该使用 PL/SQL 记录表。本章将介绍如何使用这些复合数据类型，在学习了本章之后，大家应学会：

- 使用 PL/SQL 记录和%ROWTYPE 属性；
- 使用索引表、嵌套表和变长数组（VARRAY）；
- 使用 PL/SQL 记录表；
- 使用 FORALL 语句和 BULK COLLECT 子句。
- 在嵌套表上使用集合操作符、在 FORALL 语句中使用 INDICES OF 子句和 VALUES OF 子句。

## 8.1 PL/SQL 记录

PL/SQL 记录（Record）类似于高级语言中的结构，它有益于处理单行数据。例如，当要检索某雇员的姓名、工资和补助时，如果使用标量变量接收数据信息，那么需要定义三个变量。为了简化单行多列的数据处理，开发人员可以使用 PL/SQL 记录。PL/SQL 记录由一组相关的记录成员（Field）组成，其结构如下图所示：



### 8.1.1 定义 PL/SQL 记录

当使用 PL/SQL 记录时，应用开发人员既可以自定义记录类型和记录变量，也可以使用 %ROWTYPE 属性直接定义记录变量。

#### 1. 自定义 PL/SQL 记录

当使用自定义的 PL/SQL 记录时，需要分别定义 PL/SQL 记录类型和记录变量。定义记录类型和记录变量的语法如下：

```
TYPE type_name IS RECORD (
```

```

    field_declaration,
    field_declaration]...
);
    identifier type_name;

```

如上所示, `type_name` 用于指定自定义记录类型的名称 (`IS RECORD` 表示记录类型); `field_declaration` 用于指定记录成员的定义; `identifier` 用于指定记录变量名。当为记录类型指定多个成员时, 记录成员之间用逗号隔开。示例如下:

```

DECLARE
    TYPE emp_record_type IS RECORD(
        name      emp.ename%TYPE,
        salary    emp.sal%TYPE,
        dno       emp.deptno%TYPE
    );
    emp_record emp_record_type;
    ...

```

如例所示, 记录类型 `emp_record_type` 包含三个记录成员 `name`, `salary` 和 `dno`; `emp_record` 是基于记录类型 `emp_record_type` 所定义的记录变量。

## 2. 使用%ROWTYPE 属性定义记录变量

`%ROWTYPE` 属性可以基于表或视图定义记录变量。当使用该属性定义记录变量时, 记录成员的名称和类型与表或视图列的名称和类型完全相同。为了简化表或视图所有列数据的处理, 应该使用该属性定义记录变量。而如果只是处理某几列数据, 那么应该使用自定义记录类型和记录变量, 这样可以节省内存空间。使用`%ROWTYPE` 属性定义记录变量的语法如下:

```

identifier table_name%ROWTYPE;
或
identifier view_name%ROWTYPE;

```

当使用`%ROWTYPE` 属性定义记录变量时, 记录成员个数、名称、类型与表或视图列的个数、名称、类型完全相同。示例如下:

```

dept_record dept%ROWTYPE;
emp_record emp%ROWTYPE;

```

如例所示, 记录变量 `dept_record` 的成员名为表 `dept` 的列名 (`deptno`, `dname`, `loc`); 而记录变量 `emp_record` 的成员名为表 `emp` 的列名 (`empno`, `ename`, ... )。

### 8.1.2 使用 PL/SQL 记录

在 Oracle9i 之前, 如果在内嵌 SQL 语句中使用 PL/SQL 记录变量, 那么只有 `SELECT INTO` 语句可以直接引用记录变量, 而 `INSERT`, `UPDATE` 和 `DELETE` 语句则只能引用记录成员, 不能直接引用记录变量。从 Oracle9i 开始, 开发人员不仅可以在 `SELECT` 语句中直接引用记录变量, 也可以在 `INSERT` 和 `UPDATE` 语句中直接引用记录变量。下面通过示例介绍如何使用 PL/SQL 记录。

#### 1. 在 `SELECT INTO` 语句中使用 PL/SQL 记录

`SELECT INTO` 语句用于检索单行数据。如果选择列表包含的多个列和表达式, 并且使用标量变量接收数据, 那么需要定义多个标量变量; 如果使用 PL/SQL 记录变量接收数据, 就只需要一个记录变量就可以接收数据, 从而简化了数据处理。当在 `SELECT INTO` 语句中使用

PL/SQL 记录时，既可以直接使用记录变量，也可以使用记录成员。

### 示例一：在 SELECT INTO 语句中使用记录变量

当在 SELECT INTO 语句中直接使用记录变量时，选择列表中的列和表达式的顺序、个数、类型必须要与记录成员的顺序、个数、类型完全匹配。示例如下：

```
set serveroutput on
DECLARE
    TYPE emp_record_type IS RECORD(
        name emp.ename%TYPE,
        salary emp.sal%TYPE,
        dno emp.deptno%TYPE);
    emp_record emp_record_type;
BEGIN
    SELECT ename,sal,deptno INTO emp_record
    FROM emp WHERE empno=&no;
    dbms_output.put_line(emp_record.name);
END;
/
输入 no 的值: 7788
SCOTT
```

当执行以上 PL/SQL 块时，会根据输入的雇员号显示其名称。注意，当引用记录成员时，必须在成员名之前加记录变量名作为前缀。

### 示例二：在 SELECT INTO 语句中使用记录成员

当在 SELECT INTO 语句中直接使用记录成员时，选择列表后列和表达式的顺序可以任意指定，但记录成员需要与之匹配。示例如下：

```
DECLARE
    TYPE emp_record_type IS RECORD(
        name emp.ename%TYPE,
        salary emp.sal%TYPE,
        dno emp.deptno%TYPE);
    emp_record emp_record_type;
BEGIN
    SELECT ename,sal INTO emp_record.name,emp_record.salary
    FROM emp WHERE empno=&no;
    dbms_output.put_line(emp_record.name);
END;
/
输入 no 的值: 7369
SMITH
```

## 2. 在 INSERT 语句中使用 PL/SQL 记录

在 Oracle9i 之前，如果使用 PL/SQL 记录插入数据，那么在 VALUES 子句中只能使用记录成员；从 Oracle9i 开始，不仅可以在 VALUES 子句中使用记录成员来插入数据，而且也可以直接使用记录变量插入数据。

### 示例一：在 VALUES 子句中使用记录变量

在 VALUES 子句中使用记录变量插入数据是 Oracle9i 新增加的特征，当在 VALUES 子句中使用记录变量插入数据时，列的顺序、个数、类型必须要与记录成员的顺序、个数、类型完全匹配。示例如下：

```
DECLARE
    dept_record dept%ROWTYPE;
BEGIN
    dept_record.deptno:=50;
    dept_record.dname:='ADMINISTRATOR';
    dept_record.loc:='BEIJING';
    INSERT INTO dept VALUES dept_record;
END;
/
```

如上例所示，当执行了以上 PL/SQL 块之后，会为 DEPT 表插入一行数据（部门号：50，部门名：ADMINISTRATOR，部门位置：BEIJING）。

### 示例二：在 VALUES 子句中使用记录成员

当在 VALUES 子句中使用记录成员插入数据时，列的顺序可以任意指定，但记录成员需要与之匹配，示例如下：

```
DECLARE
    dept_record dept%ROWTYPE;
BEGIN
    dept_record.deptno:=60;
    dept_record.dname:='SALES';
    INSERT INTO dept (deptno,dname) VALUES
        (dept_record.deptno,dept_record.dname);
END;
/
```

如例所示，在执行了以上 PL/SQL 块之后，会为 DEPT 表插入一行数据（部门号：60，部门名：SALES）。

### 3. 在 UPDATE 语句中使用 PL/SQL 记录

在 Oracle9i 之前，如果使用 PL/SQL 记录更新数据，那么在 SET 子句中只能使用记录成员；从 Oracle9i 开始，在 SET 子句中不仅可以使用记录成员，而且也可以直接使用记录变量。

#### (1) 在 SET 子句中使用记录变量

在 SET 子句中使用记录变量是 Oracle9i 新增加的特征，当在 SET 子句中使用记录变量更新数据时，列的顺序、个数、类型必须要与记录成员的顺序、个数、类型完全匹配。下面以修改部门 30 的部门位置为例，说明在 UPDATE 语句的 SET 子句中使用记录变量更新数据的方法。示例如下：

```
DECLARE
    dept_record dept%ROWTYPE;
BEGIN
    dept_record.deptno:=30;
```

```

dept_record.dname:='SALES';
dept_record.loc:='SHANGHAI';
UPDATE dept SET ROW=dept_record WHERE deptno=30;
END;
/

```

### (2) 在 SET 子句中使用记录成员

当在 SET 子句中使用记录成员更新数据时，列的顺序可以任意指定，但记录成员需要与之匹配。下面以修改部门 10 的部门位置为例，说明在 UPDATE 语句的 SET 子句中使用记录成员更新数据的方法。示例如下：

```

DECLARE
    dept_record dept%ROWTYPE;
BEGIN
    dept_record.loc:='GUANGZHOU';
    UPDATE dept SET loc=dept_record.loc WHERE deptno=10;
END;
/

```

### 4. 在 DELETE 语句中使用 PL/SQL 记录

当使用 PL/SQL 记录删除数据时，只能在 DELETE 语句的 WHERE 子句中使用记录成员。下面以删除部门 50 为例，说明在 DELETE 语句中使用 PL/SQL 记录的方法。示例如下：

```

DECLARE
    dept_record dept%ROWTYPE;
BEGIN
    dept_record.deptno:=50;
    DELETE FROM dept WHERE deptno=dept_record.deptno;
END;
/

```

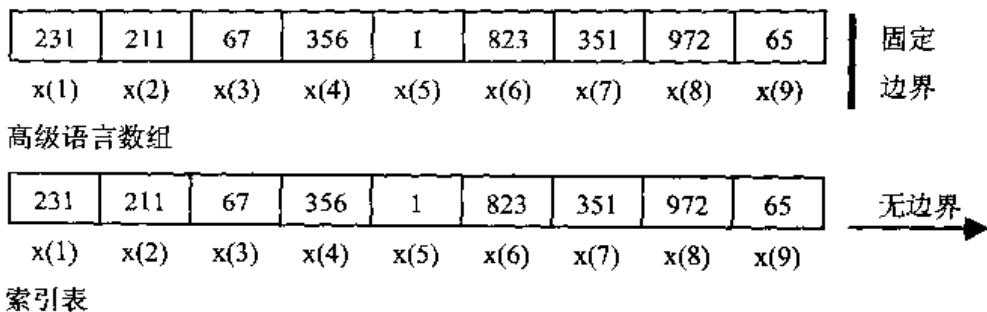
## 8.2 PL/SQL 集合

为了处理单行单列的数据，开发人员可以使用标量变量；为了处理单行多列的数据，开发人员可以使用 PL/SQL 记录；而为了处理单列多行数据，开发人员可以使用 PL/SQL 集合。例如，为了存放单个雇员的姓名，开发人员可以使用标量变量；而为了存放多个雇员的姓名，开发人员应该使用 PL/SQL 集合变量。

PL/SQL 集合类型是类似于高级语言数组的一种复合数据类型，集合类型包括索引表（PL/SQL 表）、嵌套表（Nested Table）和变长数组（VARRAY）等三种类型。当使用这些集合类型时，必须要注意三者之间的区别，以便选择最合适的数据类型。

### 8.2.1 索引表

索引表也称为 PL/SQL 表，它是 Oracle 早期版本用于处理 PL/SQL 数组的数据类型。注意，高级语言数组的元素个数是有限制的，并且下标不能为负值；而索引表的元素个数没有限制，并且下标可以为负值。如下图所示：



如上所示，索引表的下标不仅可以为负值，而且其元素个数没有限制。注意，索引表只能作为 PL/SQL 复合数据类型使用，而不能作为表列的数据类型使用。定义索引表的语法如下：

```
TYPE type_name IS TABLE OF element_type
[NOT NULL] INDEX BY key_type;
identifier type_name;
```

如上所示，`type_name` 用于指定用户自定义数据类型的名称（`IS TABLE` .. `INDEX` 表示索引表）；`element_type` 用于指定索引表元素的数据类型；`NOT NULL` 表示不允许引用 `NULL` 元素，`key_type` 用于指定索引表元素下标的数据类型（`BINARY_INTEGER`、`PLS_INTEGER` 或 `VARCHAR2`）；`identifier` 用于定义索引表变量。注意，在 Oracle9i 之前，索引表下标只允许使用数据类型 `BINARY_INTEGER` 和 `PLS_INTEGER`；而从 Oracle9i 开始，索引表下标不仅允许使用数据类型 `BINARY_INTEGER` 和 `PLS_INTEGER`，而且允许使用数据类型 `VARCHAR2`。

下面以示例说明在编写 PL/SQL 程序时使用索引表的方法。

#### 示例一：在索引表中使用 `BINARY_INTEGER` 和 `PLS_INTEGER`

在 Oracle9i 之前，索引表的元素下标只能使用数据类型 `BINARY_INTEGER` 和 `PLS_INTEGER`，示例如下：

```
set serveroutput on
DECLARE
  TYPE ename_table_type IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
  ename_table ename_table_type;
BEGIN
  SELECT ename INTO ename_table(-1) FROM emp
    WHERE empno=&no;
  dbms_output.put_line('雇员名:'||ename_table(-1));
END;
/
输入 no 的值: 7788
雇员名:SCOTT
```

如例所示，在执行了以上 PL/SQL 块之后，会根据输入的雇员号返回雇员名，其中 `ename_table(-1)` 表示下标为 -1 的元素。

#### 示例二：在索引表中使用 `VARCHAR2`

从 Oracle9i 开始，当定义索引表时，不仅允许使用 `BINARY_INTEGER` 和 `PLS_INTEGER` 作为元素下标的数据类型，而且也允许使用 `VARCHAR2` 作为元素下标的数据类型。通过使用 `VARCHAR2` 下标，可以在元素下标和元素值之间建立关联。示例如下：

```

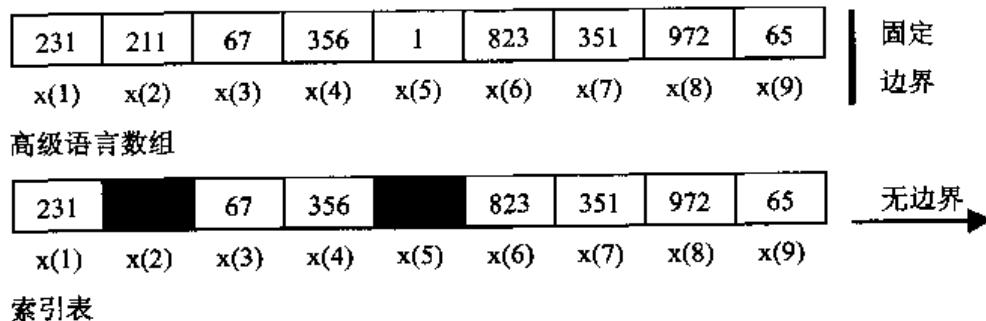
SET SERVEROUTPUT ON
DECLARE
    TYPE area_table_type IS TABLE OF NUMBER
        INDEX BY VARCHAR2(10);
    area_table area_table_type;
BEGIN
    area_table('北京'):=1;
    area_table('上海'):=2;
    area_table('广州'):=3;
    dbms_output.put_line('第一个元素:'||area_table.first);
    dbms_output.put_line('最后一个元素:'||area_table.last);
END;
/
第一个元素:北京
最后一个元素:上海

```

如上所示，在执行了以上 PL/SQL 块之后，会返回第一个元素的下标和最后一个元素的下标，因为元素下标的数据类型为字符串（数值为汉字），所以确定第一个元素和最后一个元素会以汉语拼音格式进行排序。

### 8.2.2 嵌套表

嵌套表也是一种用于处理 PL/SQL 数组的数据类型。注意，高级语言数组的元素下标从 0 或 1 开始，并且元素个数是有限制的；而嵌套表的元素下标从 1 开始，并且元素个数没有限制。另外，高级语言的数组元素值是顺序的，而嵌套表元素的数组元素值可以是稀疏的。如下图所示：



如图中所示，嵌套表元素的下标从 1 开始，并且元素个数没有限制。注意，索引表类型不能作为表列的数据类型使用，但嵌套表类型可以作为表列的数据类型使用。定义嵌套表的语法如下：

```

TYPE type_name IS TABLE OF element_type;
Identifier type_name;

```

如上所示，`type_name` 用于指定嵌套表的类型名；`element_type` 用于指定嵌套表元素的数据类型，`identifier` 用于定义嵌套表变量。注意，当使用嵌套表元素时，必须首先使用其构造方法初始化嵌套表。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    ename_table ename_table_type:=ename_table_type('A', 'A');

```

下面说明使用嵌套表的方法：

### 1. 在 PL/SQL 块中使用嵌套表

当在 PL/SQL 块中使用嵌套表变量时，必须首先使用构造方法初始化嵌套表变量，然后才能在 PL/SQL 块内引用嵌套表元素。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    ename_table ename_table_type;
BEGIN
    ename_table:=ename_table_type('MARY','MARY','MARY');
    SELECT ename INTO ename_table(2) FROM emp
        WHERE empno=&no;
    dbms_output.put_line('雇员名:'||ename_table(2));
END;
/
输入 no 的值: 7788
雇员名:SCOTT

```

如上所示，当执行了以上 PL/SQL 块之后，会根据输入的雇员号返回雇员姓名。其中，ename\_table\_type 为嵌套表类型；而 ename\_table\_type() 是其构造方法。

### 2. 在表列中使用嵌套表

嵌套表类型不仅可以在 PL/SQL 块中直接引用，也可以作为表列的数据类型使用。但如果在表列中使用嵌套表类型，必须首先使用 CREATE TYPE 命令建立嵌套表类型。另外注意，当使用嵌套表类型作为表列的数据类型时，必须要为嵌套表列指定专门的存储表。在表列中使用嵌套表类型的示例如下：

```

CREATE TYPE phone_type IS TABLE OF VARCHAR2(20);
/
CREATE TABLE employee (
    id NUMBER(4), name VARCHAR2(10), sal NUMBER(6,2),
    phone phone_type
) NESTED TABLE phone STORE AS phone_table;

```

如上所示，在使用 CREATE TYPE 命令建立了嵌套表类型 phone\_type 之后，就可在建立表 employee 时使用该嵌套表类型。

下面通过示例说明在 PL/SQL 块中操纵嵌套表列的方法。

#### 示例一：在 PL/SQL 块中为嵌套表列插入数据

当定义嵌套表类型时，Oracle 自动为该类型生成相应的构造方法。当为嵌套表列插入数据时，需要使用嵌套表的构造方法。示例如下：

```

BEGIN
    INSERT INTO employee VALUES(1,'SCOTT',800,
        phone_type('0471-3456788','13804711111'));
END;
/

```

#### 示例二：在 PL/SQL 块中检索嵌套表列的数据

当在 PL/SQL 块中检索嵌套表列的数据时，需要定义嵌套表类型的变量接收其数据。示例如下：

```

set serveroutput on
DECLARE
    phone_table phone_type;
BEGIN
    SELECT phone INTO phone_table
        FROM employee WHERE id=1;
    FOR i IN 1..phone_table.COUNT LOOP
        dbms_output.put_line('电话号码:'||phone_table(i));
    END LOOP;
END;
/
电话号码:0471-3456788
电话号码:13804711111

```

### 示例三：在 PL/SQL 块中更新嵌套表列的数据

当在 PL/SQL 块中更新嵌套表列的数据时，首先需要定义嵌套表变量，并使用构造方法初始化该变量，然后才可在执行部分使用 UPDATE 语句更新其数据。示例如下：

```

DECLARE
    phone_table phone_type:=phone_type('0471-3456788',
                                         '13804711111','0471-2233066','13056278568');
BEGIN
    UPDATE employee SET phone=phone_table
        WHERE id=1;
END;
/

```

### 8.2.3 变长数组 (VARRAY)

VARRAY 也是一种用于处理 PL/SQL 数组的数据类型，它也可以作为表列的数据类型使用。该数据类型与高级语言数组非常类似，其元素下标以 1 开始，并且元素的最大个数是有限制的。如下图所示：

231	211	67	356	1	823	351			最大：9
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)			

如上图所示，VARRAY 元素的下标从 1 开始，当前数组的上界为 7，将来可以扩展到 8 或 9，但最多 9 个元素。定义 VARRAY 的语法如下：

```

TYPE type_name IS VARRAY(size_limit) OF element_type [NOT NULL];
Identifier type_name;

```

如上所示，type\_name 用于指定 VARRAY 类型名，size\_limit 用于指定 VARRAY 元素的最大个数；element\_type 用于指定元素的数据类型；identifier 用于定义 VARRAY 变量。注意，当使用 VARRAY 元素时，必须要使用其构造方法初始化 VARRAY 元素。示例如下：

```

DECLARE
    TYPE ename_table_type IS VARRAY(20) OF emp.ename%TYPE;
    ename_table ename_table_type:=ename_table_type('A','A');

```

下面举例说明使用 VARRAY 的方法：

### 示例一：在 PL/SQL 块中使用 VARRAY

当在 PL/SQL 块中使用 VARRAY 变量时，必须首先使用其构造方法来初始化 VARRAY 变量，然后才能在 PL/SQL 块内引用 VARRAY 元素。示例如下：

```

DECLARE
    TYPE ename_table_type IS VARRAY(20) OF emp.ename%TYPE;
    ename_table ename_table_type:=ename_table_type('mary');
BEGIN
    SELECT ename INTO ename_table(1) FROM emp
    WHERE empno=&no;
    dbms_output.put_line('雇员名:'||ename_table(1));
END;
/
输入 no 的值: 7788
雇员名:SCOTT

```

如上所示，当执行了以上 PL/SQL 块之后，会根据输入的雇员号返回雇员姓名。其中，ename\_table\_type 为 VARRAY 类型，而 ename\_table\_type() 是其构造方法。

### 示例二：在表列中使用 VARRAY

VARRAY 类型不仅可以在 PL/SQL 块中直接引用，也可以作为表列的数据类型使用。但如果要在表列中引用该数据类型，则必须使用 CREATE TYPE 命令建立 VARRAY 类型。在表列中使用 VARRAY 类型的示例如下：

```

CREATE TYPE phone_type IS VARRAY(20) OF VARCHAR2(20);
/
CREATE TABLE employee (
    id NUMBER(4), name VARCHAR2(10),
    sal NUMBER(6,2), phone phone_type);

```

如上所示，在使用 CREATE TYPE 命令建立了 VARRAY 类型 phone\_type 之后，就可在建立表 employee 时引用该 VARRAY 类型。在 PL/SQL 块中操纵 VARRAY 列的方法与操纵嵌套表列的方法完全相同，但注意，嵌套表列的元素个数没有限制，而 VARRAY 列的元素个数是有限制的。

## 8.2.4 PL/SQL 记录表

PL/SQL 变量用于处理单行单列数据，PL/SQL 记录用于处理单行多列数据，PL/SQL 集合用于处理多行单列数据。为了在 PL/SQL 块中处理多行多列数据，开发人员可以使用 PL/SQL 记录表。PL/SQL 记录表结合了 PL/SQL 记录和 PL/SQL 集合的优点，从而可以有效地处理多行多列的数据。示例如下：

```

DECLARE
    TYPE emp_table_type IS TABLE OF emp%ROWTYPE
        INDEX BY BINARY_INTEGER;
    emp_table emp_table_type;
BEGIN
    SELECT * INTO emp_table(1) FROM emp
    WHERE empno=&no;
    dbms_output.put_line('雇员姓名:'||emp_table(1).ename);

```

```

dbms_output.put_line('雇员工资:'||emp_table(1).sal);
END;
/
输入 no 的值: 7788
雇员姓名:SCOTT
雇员工资:1200

```

在执行了以上 PL/SQL 块之后，会将雇员 7788 所对应的行的数据检索到 PL/SQL 记录表元素 emp\_table(1)中，并最终显示其雇员名及其工资。

### 8.2.5 多级集合

多级集合是指嵌套了集合类型的集合类型。通过使用多级集合，可以在 PL/SQL 中实现类似于多维数组的功能。在 Oracle9i 之前，当定义集合类型（索引表、嵌套表、VARRAY）时，其元素的数据类型可以是标量类型、PL/SQL 记录类型以及对象类型，但不能是集合类型。也就是说，Oracle9i 之前只能采用类似于一维数组的集合类型；而从 Oracle9i 开始，允许集合类型嵌套另一种集合类型，从而在 PL/SQL 中实现了多维数组。下面举例说明如何在 PL/SQL 块中使用多级集合。

#### 示例一：在 PL/SQL 块中使用多级 VARRAY

当在 PL/SQL 块中实现类似于多维数组的功能时，如果多维数组的元素个数是有限制的，那么可以在 VARRAY 类型中嵌套另一个 VARRAY 类型。下面以定义二维 VARRAY(10,10)为例，说明使用多级 VARRAY 的方法。示例如下：

```

DECLARE
    -- 定义一维 VARRAY
    TYPE a1_varray_type IS VARRAY(10) OF INT;
    -- 定义二维 VARRAY 集合
    TYPE n1_varray_type IS VARRAY(10) OF a1_varray_type;
    -- 初始化二维集合变量
    n1 n1_varray_type:=n1_varray_type(
        a1_varray_type(58,100,102),
        a1_varray_type(55,6,73),
        a1_varray_type(2,4));
BEGIN
    dbms_output.put_line('显示二维数组所有元素');
    FOR i IN 1..n1.COUNT LOOP
        FOR j IN 1..n1(i).COUNT LOOP
            dbms_output.put_line('n1'||i||','||j||')='
            ||n1(i)(j));
        END LOOP;
    END LOOP;
END;
/
显示二维数组所有元素
n1(1,1)=58
n1(1,2)=100
n1(1,3)=102
n1(2,1)=55

```

```

nv1(2,2)=6
nv1(2,3)=73
nv1(3,1)=2
nv1(3,2)=4

```

### 示例二：在 PL/SQL 块中使用多级嵌套表

在 PL/SQL 块中实现类似于多维数组的功能时，如果多维数组的元素个数没有限制，那么可以在嵌套表类型中嵌套另一个嵌套表类型。下面以定义二维嵌套表为例，说明使用多级嵌套表的方法。示例如下：

```

DECLARE
    -- 定义一维嵌套表
    TYPE a1_table_type IS TABLE OF INT;
    -- 定义二维嵌套表集合
    TYPE na1_table_type IS TABLE OF a1_table_type;
    -- 初始化二维集合变量
    nv1 na1_table_type:=na1_table_type(
        a1_table_type(2,4),
        a1_table_type(5,73));
BEGIN
    dbms_output.put_line('显示二维数组所有元素');
    FOR i IN 1..nv1.COUNT LOOP
        FOR j IN 1..nv1(i).COUNT LOOP
            dbms_output.put_line('nv1'||i||','||j||')='
                ||nv1(i)(j));
        END LOOP;
    END LOOP;
    END;
/
显示二维数组所有元素
nv1(1,1)=2
nv1(1,2)=4
nv1(2,1)=5
nv1(2,2)=73

```

### 示例三：在 PL/SQL 块中使用多级索引表

在 PL/SQL 块中实现类似于多维数组的功能时，如果多维数组的元素个数没有限制，那么不仅可以使用多级嵌套表实现，也可以使用多级索引表实现。下面以定义二维索引表为例，说明使用多级索引表的方法。示例如下：

```

DECLARE
    -- 定义一维 table
    TYPE a1_table_type IS TABLE OF INT
        INDEX BY BINARY_INTEGER;
    -- 定义二维 table 集合
    TYPE na1_table_type IS TABLE OF a1_table_type
        INDEX BY BINARY_INTEGER;
    nv1 na1_table_type;
BEGIN

```

```

nv1(1)(1):=10;
nv1(1)(2):=5;
nv1(2)(1):=100;
nv1(2)(2):=50;
dbms_output.put_line('显示二维数组所有元素');
FOR i IN 1..nv1.COUNT LOOP
    FOR j IN 1..nv1(i).COUNT LOOP
        dbms_output.put_line('nv1'||i||','||j||')='
        ||nv1(i)(j));
    END LOOP;
END LOOP;
END;
/
显示二维数组所有元素
nv1(1,1)=10
nv1(1,2)=5
nv1(2,1)=100
nv1(2,2)=50

```

### 8.2.6 集合方法

集合方法是 Oracle 提供的用于操纵集合变量的内置函数或过程，其中 EXISTS, COUNT, LIMIT, FIRST, NEXT, PRIOR 和 NEXT 是函数，而 EXTEND, TRIM 和 DELETE 则是过程。集合方法的调用语法如下：

```
collection_name.method_name[(parameters)]
```

注意，集合方法只能在 PL/SQL 语句中使用，而不能在 SQL 语句中调用。另外集合方法 EXTEND 和 TRIM 只适用于嵌套表和 VARRAY，而不适用于索引表。

#### 1. EXISTS

该方法用于确定集合元素是否存在，如果集合元素存在，则返回 TRUE；如果集合元素不存在，则返回 FALSE。使用该集合方法的示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    ename_table ename_table_type;
BEGIN
    IF ename_table.EXISTS(1) THEN
        ename_table(1):='SCOTT';
    ELSE
        dbms_output.put_line('必须初始化集合元素');
    END IF;
END;
/
必须初始化集合元素

```

如上所示，在执行以上 PL/SQL 块时，会显示消息“必须初始化集合元素”。大家需要注意，在引用嵌套表或 VARRAY 元素之前，必须首先初始化相应元素，而索引表元素在 SELECT 语句中可以直接引用。

## 2. COUNT

该集合方法用于返回当前集合变量中的元素总个数。如果集合元素存在数值，则统计结果会包含该元素；如果集合元素为 NULL，则统计结果不会包含该元素。使用该方法的示例如下：

```
DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
BEGIN
    ename_table(-5):='SCOTT';
    ename_table(1):='SMITH';
    ename_table(5):='MARY';
    ename_table(10):='BLAKE';
    dbms_output.put_line('集合元素总个数:'||ename_table.count);
END;
/
集合元素总个数:4
```

## 3. LIMIT

该方法用于返回集合元素的最大个数。因为嵌套表和索引表的元素个数没有限制，所以调用该方法会返回 NULL；而对于 VARRAY 来说，该方法会返回 VARRAY 所允许的最大元素个数。示例如下：

```
DECLARE
    TYPE ename_table_type IS VARRAY(20) OF emp.ename%TYPE;
    ename_table ename_table_type:=ename_table_type('mary');
BEGIN
    dbms_output.put_line('集合元素的最大个数:'|
        ||ename_table.limit);
END;
/
集合元素的最大个数:20
```

## 4. FIRST 和 LAST

FIRST 方法用于返回集合变量第一个元素的下标，而 LAST 方法则用于返回集合变量最后一个元素的下标。示例如下：

```
DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
BEGIN
    ename_table(-5):='SCOTT';
    ename_table(1):='SMITH';
    ename_table(5):='MARY';
    ename_table(10):='BLAKE';
    dbms_output.put_line('第一个元素:'||ename_table.first);
    dbms_output.put_line('最后一个元素:'||ename_table.last);
END;
```

```

/
第一个元素:-5
最后一个元素:10

```

### 5. PRIOR 和 NEXT

PRIOR 方法用于返回当前集合元素的前一个元素的下标，而 NEXT 方法则用于返回当前集合元素的后一个元素的下标。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
BEGIN
    ename_table(-5):='SCOTT';
    ename_table(1):='SMITH';
    ename_table(5):='MARY';
    ename_table(10):='BLAKE';
    dbms_output.put_line('元素 5 的前一个元素:' || 
        ename_table.prior(5));
    dbms_output.put_line('元素 5 的后一个元素:' || 
        ename_table.next(5));
END;
/
元素 5 的前一个元素:1
元素 5 的后一个元素:10

```

### 6. EXTEND

该方法用于扩展集合变量的尺寸，并为它们增加元素。注意，该方法只适用于嵌套表和 VARRAY。该方法有 EXTEND, EXTEND(n), EXTEND(n,i)等三种调用格式，其中 EXTEND 用于为集合变量添加一个 null 元素，EXTEND(n)用于为集合变量添加 n 个 null 元素，而 EXTEND(n,i)则用于为集合变量添加 n 个元素（元素值与第 i 个元素相同）。示例如下：

```

DECLARE
    TYPE ename_table_type IS VARRAY(20) OF VARCHAR2(10);
    ename_table ename_table_type;
BEGIN
    ename_table:=ename_table_type('MARY');
    ename_table.EXTEND(5,1);
    dbms_output.put_line('元素总个数: ' || ename_table.COUNT);
END;
/
元素总个数: 6

```

### 7. TRIM

该方法用于从集合尾部删除元素，它有 TRIM 和 TRIM(n)两种调用格式。其中 TRIM 用于从集合尾部删除一个元素；而 TRIM(n)则用于从集合尾部删除 n 个元素。注意，该方法只适用于嵌套表和 VARRAY。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF VARCHAR2(10);

```

```

    ename_table ename_table_type;
BEGIN
    ename_table:=ename_table_type('A', 'A', 'A', 'A', 'A');
    ename_table.TRIM(2);
    dbms_output.put_line('元素总个数: '||ename_table.COUNT);
END;
/
元素总个数: 3

```

## 8. DELETE

该方法用于删除集合元素，但该方法只适用于嵌套表和索引表，而不适用于 VARRAY。该方法有 DELETE, DELETE(n), DELETE(m,n) 等三种调用格式。其中 DELETE 用于删除集合变量的所有元素；DELETE(n) 用于删除集合变量的第 n 个元素；而 DELETE(m,n) 则用于删除集合变量中从 m 到 n 之间的所有元素。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
BEGIN
    ename_table(-5):='SCOTT';
    ename_table(1):='SMITH';
    ename_table(5):='MARY';
    ename_table(10):='BLAKE';
    ename_table.DELETE(5);
    dbms_output.put_line('元素总个数:'||ename_table.COUNT);
END;
/
元素总个数:3

```

### 8.2.7 集合赋值

当使用嵌套表和 VARRAY 时，通过执行 INSERT, UPDATE, FETCH, SELECT, 赋值语句，用户可以将一个集合的数据赋值给另一个集合。从 Oracle 10g 开始，当给嵌套表赋值时，还可以使用 SET, MULTISET UNION, MULTISET INTERSECT, MULTISET EXCEPT 等集合操作符。其中，SET 操作符用于取消嵌套表中的重复值，MULTISET UNION 用于取得两个嵌套表的并集（带有 DISTINCT 操作符可以取消重复结果）；MULTISET INTERSECT 用于取得两个嵌套表的交集；MULTISET EXCEPT 用于取得两个嵌套表的差集。下面介绍如何进行集合赋值。

#### 1. 将一个集合的数据赋值给另一个集合

当使用赋值语句 (:=) 或 SQL 语句将源集合中的数据赋值给目标集合时，会自动清除目标集合原有的数据，并将源集合中的数据赋值给该目标集合。示例如下：

```

DECLARE
    TYPE name_varray_type IS VARRAY(4) OF VARCHAR2(10);
    name_array1 name_varray_type;
    name_array2 name_varray_type;

```

```

BEGIN
    name_array1:=name_varray_type('SCOTT','SMITH');
    name_array2:=name_varray_type('a','a','a','a');
    dbms_output.put('name_array2 的原数据:');
    FOR i IN 1..name_array2.COUNT LOOP
        dbms_output.put(' '||name_array2(i));
    END LOOP;
    dbms_output.new_line;
    name_array2:=name_array1;
    dbms_output.put('name_array2 的新数据:');
    FOR i IN 1..name_array2.COUNT LOOP
        dbms_output.put(' '||name_array2(i));
    END LOOP;
    dbms_output.new_line;
END;
/
name_array2 的原数据: a a a a
name_array2 的新数据: SCOTT SMITH

```

注意，当进行集合赋值时，源集合和目标集合的数据类型必须完全一致。如果集合元素数据类型一致，但集合类型不一致，那也不能进行赋值。下面是赋值错误的一个示例：

```

DECLARE
    TYPE name_varray1_type IS VARRAY(4) OF VARCHAR2(10);
    TYPE name_varray2_type IS VARRAY(4) OF VARCHAR2(10);
    name_array1 name_varray1_type;
    name_array2 name_varray2_type;
BEGIN
    name_array1:=name_varray1_type('SCOTT','SMITH');
    name_array2:=name_array1;
END;
/
name_array2:=name_array1;
*
ERROR 位于第 8 行:
ORA-06550: 第 8 行, 第 16 列:
PLS-00382: 表达式类型错误
ORA-06550: 第 8 行, 第 3 列:
PL/SQL: Statement ignored

```

## 2. 给集合赋 NULL 值

编写 PL/SQL 程序时，某些情况可能需要清空集合变量的所有数据。在清空集合变量的所有数据时，既可以使用集合方法 DELETE 和 TRIM，也可以将一个 NULL 集合变量赋值给目标集合变量。下面以给集合赋 NULL 值为例，说明清空集合数据的方法、示例如下：

```

DECLARE
    TYPE name_varray_type IS VARRAY(4) OF VARCHAR2(10);
    name_array name_varray_type;
    name_empty name_varray_type;

```

```

BEGIN
    name_array:=name_varray_type('SCOTT','SMITH');
    dbms_output.put_line('name_array 的原有元素个数:' ||
        ||name_array.count);
    name_array:=name_empty;
    IF name_array IS NULL THEN
        dbms_output.put_line('name_array 的现有元素个数:0');
    END IF;
END;
/
name_array 的原有元素个数:2
name_array 的现有元素个数:0

```

### 3. 使用集合操作符给嵌套表赋值

从 Oracle 10g 开始,在编写 PL/SQL 程序时允许将多个嵌套表的结果组合到某个嵌套表中,这项任务是使用 ANSI 集合操作符 (SET, MULTISET UNION, MULTISET INTERSECT, MULTISET EXCEPT) 来完成的。下面以示例说明使用这些集合操作符的方法。

#### (1) 使用 SET 操作符

SET 操作符用于取消特定嵌套表中的重复值。下面以去掉嵌套表 nt\_table 的重复元素为例,说明使用 SET 操作符的方法。示例如下:

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt_table nt_table_type:=nt_table_type(2,4,3,1,2);
    result nt_table_type;
BEGIN
    result:=SET(nt_table);
    dbms_output.put('result:');
    FOR i IN 1..result.count LOOP
        dbms_output.put(' '||result(i));
    END LOOP;
    dbms_output.new_line;
END;
/
result: 2 4 3 1

```

#### (2) 使用 MULTISET UNION 操作符

MULTISET UNION 用于取得两个嵌套表的并集。当使用该操作符合并嵌套表结果时,结果集中会包含重复值。下面以合并嵌套表 nt1 和 nt2 的结果为例,说明使用 MULTISET UNION 操作符的方法。示例如下:

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3);
    nt2 nt_table_type:=nt_table_type(3,4,5);
    result nt_table_type;
BEGIN
    result:=nt1 MULTISET UNION nt2;

```

```

    dbms_output.put('result:');
    FOR i IN 1..result.count LOOP
        dbms_output.put(' '||result(i));
    END LOOP;
    dbms_output.new_line;
END;
/
result: 1 2 3 3 4 5

```

### (3) 使用 MULTISET UNION DISTINCT 操作符

MULTISET UNION DISTINCT 操作符用于取得两个嵌套表的并集，并取消重复结果。当使用 MULTISET UNION 操作符合并嵌套表时，结果集中会包含重复值。为了取消结果集中的重复值，应该使用 MULTISET UNION DISTINCT 操作符。下面以合并嵌套表 nt1 和 nt2 的结果为例，说明使用 MULTISET UNION DISTINCT 操作符的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3);
    nt2 nt_table_type:=nt_table_type(3,4,5);
    result nt_table_type;
BEGIN
    result:=nt1 MULTISET UNION DISTINCT nt2;
    dbms_output.put('result:');
    FOR i IN 1..result.count LOOP
        dbms_output.put(' '||result(i));
    END LOOP;
    dbms_output.new_line;
END;
/
result: 1 2 3 4 5

```

### (4) 使用 MULTISET INTERSECT 操作符

MULTISET INTERSECT 操作符用于取得两个嵌套表的交集。下面以取得嵌套表 nt1 和 nt2 的交集为例，说明使用 MULTISET INTERSECT 操作符的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3);
    nt2 nt_table_type:=nt_table_type(3,4,5);
    result nt_table_type;
BEGIN
    result:=nt1 MULTISET INTERSECT nt2;
    dbms_output.put('result:');
    FOR i IN 1..result.count LOOP
        dbms_output.put(' '||result(i));
    END LOOP;
    dbms_output.new_line;
END;
/

```

```
result: 3
```

### (5) 使用 MULTISET EXCEPT 操作符

MULTISET EXCEPT 用于取得两个嵌套表的差集。下面以取得在嵌套表 nt1 中存在，但是在嵌套表 nt2 中不存在的元素为例，说明使用 MULTISET EXCEPT 操作符的方法。示例如下：

```
DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3);
    nt2 nt_table_type:=nt_table_type(3,4,5);
    result nt_table_type;
BEGIN
    result:=nt1 MULTISET EXCEPT nt2;
    dbms_output.put('result:');
    FOR i IN 1..result.count LOOP
        dbms_output.put(' '||result(i));
    END LOOP;
    dbms_output.new_line;
END;
/
result: 1 2
```

## 8.2.8 比较集合

在 Oracle 10g 之前，当使用嵌套表类型和 VARRAY 类型的集合变量时，开发人员可以检测集合变量是否为 NULL。从 Oracle 10g 开始，开发人员还可以比较两个集合变量是否相同，另外还可以在嵌套表上使用 CARDINALITY, SUBMULTISET OF, MEMBER OF, IS A SET, IS EMPTY 等集合操作符。其中，函数 CARDINALITY 用于返回嵌套表变量的元素个数，操作符 SUBMULTISET OF 用于确定一个嵌套表是否为另一个嵌套表的子集，操作符 MEMBER OF 用于检测特定数据是否为嵌套表元素，操作符 IS A SET 用于检测嵌套表是否包含重复的元素值，操作符 IS EMPTY 用于检测嵌套表是否为 NULL。下面介绍比较集合元素，以及使用各种操作符的方法。

### 1. 检测集合是否为 NULL

当编写复杂的 PL/SQL 应用程序时，经常需要检测集合变量是否为 NULL，在 Oracle 10g 之前，使用 IS NULL 操作符可以检测嵌套表或 VARRAY 变量是否为 NULL。示例如下：

```
DECLARE
    TYPE name_array_type IS VARRAY(3) OF VARCHAR2(10);
    name_array name_array_type;
BEGIN
    IF name_array IS NULL THEN
        dbms_output.put_line('name_array 未初始化');
    END IF;
END;
/
name_array 未初始化
```

但 Oracle 10g 开始，当检测嵌套表是否为 NULL 时，不仅可以使用 IS NULL 操作符，也

可以使用 IS EMPTY 操作符。注意，IS EMPTY 操作符只适用于嵌套表，而不适用于 VARRAY。示例如下：

```

DECLARE
    TYPE name_table_type IS TABLE OF VARCHAR2(10);
    name_table name_table_type;
BEGIN
    IF name_table IS EMPTY THEN
        dbms_output.put_line('name_table 未初始化');
    END IF;
END;
/
name_table 未初始化

```

## 2. 比较嵌套表是否相同

在 Oracle 10g 之前，不能直接比较两个嵌套表是否相同。但从 Oracle 10g 开始，允许使用比较操作符=和!=检测两个嵌套表变量是否相同。注意，使用这两个比较符只能比较嵌套表，而不能比较 VARRAY 和索引表。示例如下：

```

DECLARE
    TYPE name_table_type IS TABLE OF VARCHAR2(10);
    name_table1 name_table_type;
    name_table2 name_table_type;
BEGIN
    name_table1:=name_table_type('SCOTT');
    name_table2:=name_table_type('SMITH');
    IF name_table1=name_table2 THEN
        dbms_output.put_line('两个嵌套表完全相同');
    ELSE
        dbms_output.put_line('两个嵌套表数值不同');
    END IF;
END;
/
两个嵌套表数值不同

```

## 3. 在嵌套表上使用集合操作符

从 Oracle 10g 开始，开发人员可以在嵌套表上使用 ANSI 集合操作符 CARDINALITY, MEMBER OF, IS A SET。注意，这些操作符只适用于嵌套表，而不适用于 VARRAY 和索引表。

### (1) 使用函数 CARDINALITY

函数 CARDINALITY 用于返回嵌套表变量的元素个数。下面以输出嵌套表 nt1 的元素个数为例，说明使用函数 CARDINALITY 的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3,1);
BEGIN
    dbms_output.put_line('元素个数：'
    ||cardinality(nt1));
END;

```

```

/
元素个数:4

```

#### (2) 使用操作符 SUBMULTISET OF

操作符 SUBMULTISET OF 用于确定一个嵌套表是否为另一个嵌套表的子集。下面以检测嵌套表 nt1 是否为嵌套表 nt2 的子集为例，说明使用操作符 SUBMULTISET OF 的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3);
    nt2 nt_table_type:=nt_table_type(1,2,3,4);
BEGIN
    IF nt1 SUBMULTISET OF nt2 THEN
        dbms_output.put_line('nt1 是 nt2 的子集');
    END IF;
END;
/
nt1 是 nt2 的子集

```

#### (3) 使用操作符 MEMBER OF

操作符 MEMBER OF 用于检测特定数据是否为嵌套表的元素。下面以检测变量 v1 中的值是否为嵌套表 nt1 的元素为例，说明使用操作符 MEMBER OF 的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3,5);
    v1 NUMBER:=&v1;
BEGIN
    IF v1 MEMBER OF nt1 THEN
        dbms_output.put_line('v1 是 nt1 的元素');
    END IF;
END;
/
输入 v1 的值: 1
v1 是 nt1 的元素

```

#### (4) 使用操作符 IS A SET

操作符 IS A SET 用于检测嵌套表是否包含重复的元素值。下面以检测嵌套表 nt1 是否包含重复元素为例，说明使用操作符 IS A SET 的方法。示例如下：

```

DECLARE
    TYPE nt_table_type IS TABLE OF NUMBER;
    nt1 nt_table_type:=nt_table_type(1,2,3,5);
BEGIN
    IF nt1 IS A SET THEN
        dbms_output.put_line('嵌套表 nt1 无重复值');
    END IF;
END;
/

```

嵌套表 nt1 无重复值

### 8.3 批量绑定

批量绑定是 Oracle9i 新增加的特征，是指执行单次 SQL 操作能传递所有集合元素的数据。当在 SELECT, INSERT, UPDATE, DELETE 语句上处理批量数据时，通过批量绑定，可以极大地加快数据处理速度，提高应用程序的性能。在对不使用批量绑定和使用批量绑定进行比较之前，应首先建立示例表 DEMO：

```
CREATE TABLE demo (
    id NUMBER(6) PRIMARY KEY, name VARCHAR2(10)
);
```

下面以不使用批量绑定和使用批量绑定为 DEMO 表分别插入 5000 行数据为例，比较使用这两种方法插入数据的速度。

#### 1. 不使用批量绑定

在 Oracle9i 之前，当使用 VALUES 子句将数据插入到数据库表时，每次只能插入一条数据。如果要插入 5000 行数据，那么就需要调用 5000 次 INSERT 语句。因此，为了将多个集合元素的数据插入到数据库表，就必须要使用循环方式来完成。下面以将素引表的 5000 个元素插入到 DEMO 表为例，说明在 Oracle9i 之前将集合元素插入到数据库表中的方法。示例如下：

```
DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    TYPE name_table_type IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
    name_table name_table_type;
    start_time NUMBER(10);
    end_time NUMBER(10);
BEGIN
    FOR i IN 1..5000 LOOP
        id_table(i):=i;
        name_table(i):='Name'||to_char(i);
    END LOOP;
    start_time:=dbms_utility.get_time;
    FOR i IN 1..id_table.COUNT LOOP
        INSERT INTO demo VALUES(id_table(i),name_table(i));
    END LOOP;
    end_time:=dbms_utility.get_time;
    dbms_output.put_line('总计时间(秒):'||to_char((end_time-start_time)/100));
END;
/
总计时间(秒):6.41
```

## 2. 使用批量绑定

在 Oracle9i 之中, 当使用 VALUES 子句为数据库表插入数据时, 通过使用批量绑定特征, 只需要执行一条 INSERT 语句就可以插入 5000 行数据。示例如下:

```

DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    TYPE name_table_type IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
    name_table name_table_type;
    start_time NUMBER(10);
    end_time NUMBER(10);
BEGIN
    FOR i IN 1..5000 LOOP
        id_table(i):=i;
        name_table(i):='Name'||to_char(i);
    END LOOP;
    start_time:=dbms_utility.get_time;
    FORALL i IN 1..id_table.COUNT
        INSERT INTO demo VALUES(id_table(i),name_table(i));
    end_time:=dbms_utility.get_time;
    dbms_output.put_line('总计时间(秒)：'||to_char((end_time-start_time)/100));
END;
/
总计时间(秒)：.33

```

## 3. 结论

当不使用批量绑定时, 插入 5000 行数据需要 6.41 秒, 而使用批量绑定需要的时间仅仅是 0.33 秒, 显然使用批量绑定的速度要远远优于不使用批量绑定。批量绑定是使用 BULK COLLECT 子句和 FORALL 语句来完成的, 其中 BULK COLLECT 子句用于取得批量数据, 该子句只能用于 SELECT 语句、FETCH 语句和 DML 返回子句中; 而 FORALL 语句只适用于执行批量的 DML 操作。

### 8.3.1 FORALL 语句

当要在 PL/SQL 应用程序中执行批量 INSERT、UPDATE 和 DELETE 操作时, 可以使用 FORALL 语句。在 Oracle9i 之中, 当使用 FORALL 语句时, 必须具有连续的元素; 而从 Oracle 10g 开始, 通过使用新增加的 INDICES OF 子句和 VALUES OF 子句, 可以使用不连续的集合元素。注意, FOR 语句是循环语句, 但 FORALL 语句却不是循环语句。从 Oracle 10g 开始, FORALL 语句有三种执行语法, 如下所示:

- 语法一:

```
FORALL index IN lower_bound.. upper_bound
    sql_statement;
```

如上所示, index 是隐含定义的整数变量 (将作为集合元素下标被引用); lower\_bound 和

`upper_bound` 分别是集合元素的上界和下界。

- 语法二：

```
FORALL index IN INDICES OF collection
    [BETWEEN lower_bound..AND. upper_bound]
    sql_statement;
```

如上所示，`INDICES OF` 子句用于指定只取得对应于 `collection` 集合元素下标的 `index` 值。

- 语法三：

```
FORALL index IN VALUES OF index_collection
    sql_statement;
```

如上所示，`VALUES OF` 子句用于指定 `index` 值从集合变量 `index_collection` 中取得。注意，在 Oracle9i 中只能使用第一种语法。下面通过示例说明使用 `FORALL` 语句执行批量 DML 操作的方法。

### 1. 在 `INSERT` 语句上使用批量绑定

当使用批量绑定为数据库表插入数据时，首先需要给集合元素赋值，然后使用 `FORALL` 语句执行批量绑定插入操作。下面以将索引表 `id_table` 的前 10 条数据批量插入到 `DEMO` 表为例，说明在 `INSERT` 语句上使用批量绑定的方法，示例如下：

```
DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    TYPE name_table_type IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
    name_table name_table_type;
BEGIN
    FOR i IN 1..10 LOOP
        id_table(i):=i;
        name_table(i):='Name'||to_char(i);
    END LOOP;
    FORALL i IN 1..id_table.COUNT
        INSERT INTO demo VALUES(id_table(i),name_table(i));
END;
/
```

### 2. 在 `UPDATE` 语句上使用批量绑定

当使用批量绑定更新数据库数据时，首先需要给集合元素赋值，然后使用 `FORALL` 语句执行批量绑定更新操作。下面以使用索引表 `id_table` 和 `name_table` 更新前 5 条数据为例，说明在 `UPDATE` 语句上使用批量绑定的方法。示例如下：

```
DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    TYPE name_table_type IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
    name_table name_table_type;
```

```

BEGIN
    FOR i IN 1..5 LOOP
        id_table(i):=i;
        name_table(i):='N'||to_char(i);
    END LOOP;
    FORALL i IN 1..id_table.COUNT
        UPDATE demo SET name=name_table(i)
        WHERE id=id_table(i);
    END;
    /

```

### 3. 在 DELETE 语句上使用批量绑定

当使用批量绑定删除数据库表的数据时，首先需要为集合元素赋值，然后才使用 FORALL 语句执行批量绑定删除相应数据。下面以使用索引表 id\_table 删除 DEMO 表的前 3 行数据为例，说明在 DELETE 语句上使用批量绑定的方法。示例如下：

```

DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
BEGIN
    FOR i IN 1..3 LOOP
        id_table(i):=i;
    END LOOP;
    FORALL i IN 1..id_table.COUNT
        DELETE FROM demo WHERE id=id_table(i);
    END;
    /

```

### 4. 在 FORALL 语句中使用部分集合元素

使用 FORALL 语句执行批量绑定时，既可以使用集合的所有元素，也可以使用集合的部分元素。下面以将索引表 id\_table 的元素 8、9、10 插入到 DEMO 表中为例，说明在 FORALL 语句中使用部分集合元素的方法。示例如下：

```

DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6)
        INDEX BY BINARY_INTEGER;
    id_table id_table_type;
BEGIN
    FOR i IN 1..10 LOOP
        id_table(i):=11-i;
    END LOOP;
    FORALL i IN 8..10
        INSERT INTO demo (id) VALUES(id_table(i));
    END;
    /
SQL> SELECT id FROM demo;
      ID
-----

```

1  
2  
3

### 5. 在 FORALL 语句上使用 INDICES OF 子句

INDICES OF 子句是 Oracle 10g 新增加的特征，该子句用于跳过 NULL 集合元素。下面以使用嵌套表 id\_table 去删除 DEMO 表的 id 值为 1、3、5 的数据为例，说明在 FORALL 语句中使用 INDICES OF 子句的方法。示例如下：

```
DECLARE
    TYPE id_table_type IS TABLE OF NUMBER(6);
    id_table id_table_type;
BEGIN
    id_table:=id_table_type(1,null,3,null,5);
    FORALL i IN INDICES OF id_table
        DELETE FROM demo WHERE id=id_table(i);
END;
/
SQL> select id from demo;
      ID
-----
      2
```

### 6. 在 FORALL 语句上使用 VALUES OF 子句

VALUES OF 子句是 Oracle 10g 新增加的特征，该子句用于从其他集合变量中取得集合下标 (index) 的值。在介绍如何使用 VALUES OF 子句之前，应首先执行以下语句建立 new\_demo 表：

```
CREATE TABLE new_demo AS SELECT * FROM demo WHERE 1=0;
```

下面以将 DEMO 表的 id=6、8、10 等三行数据复制到 NEW\_DEMO 表中为例，说明在 FORALL 语句中使用 VALUES OF 子句的方法。示例如下：

```
DECLARE
    TYPE id_table_type IS TABLE OF demo.id%TYPE;
    TYPE name_table_type IS TABLE OF demo.name%TYPE;
    id_table id_table_type;
    name_table name_table_type;
    TYPE index_pointer_type IS TABLE OF PLS_INTEGER;
    index_pointer index_pointer_type;
BEGIN
    SELECT * BULK COLLECT INTO id_table,name_table
    FROM demo;
    index_pointer:=index_pointer_type(6,8,10);
    FORALL i IN VALUES OF index_pointer
        INSERT INTO new_demo VALUES(id_table(i),name_table(i));
END;
/
SQL> SELECT id FROM new_demo;
      ID
```

```
-----
6
8
10
```

### 7. 使用 SQL%BULK\_ROWCOUNT 属性

属性 SQL%BULK\_ROWCOUNT 是专门为 FORALL 语句提供的，用于取得在执行批量绑定操作时第 i 个元素所作用的行数。示例如下：

```
DECLARE
  TYPE dno_table_type IS TABLE OF NUMBER(3);
  dno_table dno_table_type:=dno_table_type(10,20);
BEGIN
  FORALL i IN 1..dno_table.COUNT
    UPDATE emp SET sal=sal*1.1 WHERE deptno=dno_table(i);
    dbms_output.put_line('第 2 个元素更新的行数:' || sql%bulk_rowcount(2));
  END;
/
第 2 个元素更新的行数:5
```

### 8.3.2 BULK COLLECT 子句

BULK COLLECT 子句用于取得批量数据，它只适用于 SELECT INTO 语句，FETCH INTO 语句和 DML 返回子句。通过使用该子句，可以将批量数据存放到 PL/SQL 集合变量中。使用 BULK COLLECT 语句的语法如下：

```
... BULK COLLECT INTO collection_name [, collection_name] ...
```

如上所示，collection\_name 用于指定集合变量名。下面通过示例说明如何在 SELECT INTO 和 DML 返回子句中使用 BULK COLLECT 子句，关于如何在 FETCH INTO 语句中使用 BULK COLLECT 子句，请参见第 9 章相关内容。

#### 1. 在 SELECT INTO 语句中使用 BULK COLLECT 子句

在 Oracle9i 之前，当编写 SELECT INTO 语句时，该语句必须返回一行数据，并且只能返回一行数据，否则会触发 PL/SQL 例外。从 Oracle9i 开始，通过在 SELECT INTO 语句中使用 BULK COLLECT 子句，可以一次将 SELECT 语句的多行结果检索到集合变量中。下面以检索特定部门的所有雇员为例，说明在 SELECT INTO 语句中使用 BULK COLLECT 子句的方法。示例如下：

```
DECLARE
  TYPE emp_table_type IS TABLE OF emp%ROWTYPE
  INDEX BY BINARY_INTEGER;
  emp_table emp_table_type;
BEGIN
  .
  .
  .
  SELECT * BULK COLLECT INTO emp_table
  FROM emp WHERE deptno=&no;
  FOR i IN 1..emp_table.COUNT LOOP
    dbms_output.put_line('雇员姓名:' || emp_table(i).ename);
  END LOOP;
```

```

END;
/
输入 no 的值: 10
雇员姓名:CLARK
雇员姓名:KING
雇员姓名:MILLER

```

## 2. 在 DML 的返回子句中使用 BULK COLLECT 子句

执行 DML 操作时会改变数据库数据。为了取得 DML 操作所改变的数据，可以使用 RETURNING 子句。为了取得 DML 所作用的多行数据，需要使用 BULK COLLECT 子句。下面以删除特定部门所有雇员，并显示雇员名称为例，说明在 DML 返回子句中使用 BULK COLLECT 子句的方法。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    ename_table ename_table_type;
BEGIN
    DELETE FROM emp WHERE deptno=&no
        RETURNING ename BULK COLLECT INTO ename_table;
    dbms_output.put('雇员名:');
    FOR i IN 1..ename_table.COUNT LOOP
        dbms_output.put(ename_table(i)||' ');
    END LOOP;
    dbms_output.new_line;
END;
/
输入 no 的值: 10
雇员名:CLARK KING MILLER

```

## 8.4 习题

1. 以下哪几种复合数据类型可以作为表列？
  - A. 记录类型
  - B. 嵌套表
  - C. 索引表
  - D. VARRAY
2. 当初始化哪种类型的集合元素时，可以直接给元素赋值？
  - A. 嵌套表
  - B. 索引表
  - C. VARRAY
3. 当定义索引表时，其下标可以使用以下哪些数据类型？
  - A. CHAR
  - B. VARCHAR2
  - C. INT
  - D. INTEGER
  - E. BINARY\_INTEGER
  - F. PLS\_INTEGER
4. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入订单号，使用 PL/SQL 记录变量接收客户名和订单价格总额，并显示订单所对应的客户名和价格总额。格式如下：

输入 id 的值: 610

客户名:BOB

订单总额:101.4

5. 编写 PL/SQL 块，使用%ROWTYPE 属性基于 PRODUCT 表来定义记录变量，并使用

SQL\*Plus 替代变量为记录成员提供数据，然后为 PRODUCT 表插入数据。格式如下：

输入 id 的值： 100900

输入 description 的值： 高尔夫球

6. 编写 PL/SQL 块，分别定义 VARRAY 变量 name\_array（存放客户名）、city\_array（存放客户所在城市），然后使用 BULK COLLECT 子句检索所有客户名称和所在城市，并显示客户名和所在城市。格式如下：

客户名:BOB,所在城市:SPRING

客户名:MARY,所在城市:HOUSTON

7. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入订单号，使用 PL/SQL 记录表接收所有条款信息，并显示条款编号及每个条款的价格总额。格式如下：

输入 id 的值： 600

条款编号:1,总价:42

条款编号:2,总价:58

## 第9章 使用游标

当在 PL/SQL 块中执行查询语句（SELECT）和数据操纵语句（DML）时，Oracle 会为其分配上下文区（Context Area），游标是指向上下文区的指针。对于数据操纵语句和单行 SELECT INTO 语句来说，Oracle 会为它们分配隐含游标。在 Oracle9i 之前，为了处理 SELECT 语句返回的多行数据，开发人员必须要使用显式游标；从 Oracle9i 开始，开发人员既可以使用显式游标处理多行数据，也可以使用 SELECT .. BULK COLLECT INTO 语句处理多行数据。本章给大家介绍如何在 PL/SQL 块中使用显式游标，在学习了本章之后，读者应该学会：

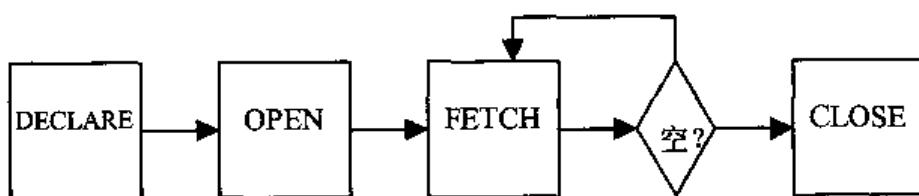
- 使用显式游标以及游标属性；
- 使用参数游标；
- 使用显式游标更新或删除数据；
- 使用游标 FOR 循环；
- 使用游标变量；
- 使用 FETCH .. BULK COLLECT INTO 语句和 CURSOR 表达式。

### 9.1 显式游标

PL/SQL 包含隐含游标和显式游标等两种游标类型，其中隐含游标用于处理 SELECT INTO 和 DML 语句，而显式游标则专门用于处理 SELECT 语句返回的多行数据。

#### 1. 使用显式游标

为了处理 SELECT 语句返回的多行数据，开发人员可以使用显式游标，使用显式游标包括定义游标、打开游标、提取数据和关闭游标四个阶段，如下图所示：



#### (1) 定义游标

在使用显式游标之前，必须首先在定义部分定义游标。定义游标用于指定游标所对应的 SELECT 语句，语法如下：

```
CURSOR cursor_name IS select_statement;
```

如上所示，cursor\_name 用于指定的游标名称；select\_statement 用于指定游标所对应的 SELECT 语句。

#### (2) 打开游标

当打开游标时，Oracle 会执行游标所对应的 SELECT 语句，并且将 SELECT 语句的结果

暂时存放到结果集中。语法如下：

```
OPEN cursor_name;
```

该游标名必须是在定义部分已经被定义的游标。

### (3) 提取数据

在打开游标之后，SELECT 语句的结果被临时存放到游标结果集中。为了处理结果集中的数据，需要使用 FETCH 语句提取游标数据。在 Oracle9i 之前，使用 FETCH 语句每次只能提取一行数据；从 Oracle9i 开始，通过使用 FETCH..BULK COLLECT INTO 语句，每次可以提取多行数据。语法如下：

```
语法一：FETCH cursor_name INTO variable1,variable2,...;
```

```
语法二：FETCH cursor_name
```

```
      BULK COLLECT INTO collect1, collect2,...[LIMIT rows];
```

如上所示，variable 用于指定接收游标数据的变量；collect 用于指定接收游标结果的集合变量。注意，当使用语法一时，必须要使用循环语句处理结果集的所有数据。

### (4) 关闭游标

在提取并处理了结果集的所有数据之后，就可以关闭游标并释放其结果集了。语法如下：

```
CLOSE cursor_name;
```

## 2. 显式游标属性

显式游标属性用于返回显式游标的执行信息，这些属性包括%ISOPEN，%FOUND，%NOTFOUND 和%ROWCOUNT。当使用显式游标属性时，必须要在显式游标属性之前带有显式游标名作为前缀（游标名.属性名）。

### (1) %ISOPEN

该属性用于确定游标是否已经打开。如果游标已经打开，则返回值为 TRUE；如果游标没有打开，则返回值为 FALSE。示例如下：

```
IF c1%ISOPEN THEN      ——如果游标打开，则执行相应操作
  ...
ELSE                   ——如果游标未打开，则打开游标
  OPEN c1;
END IF;
```

### (2) %FOUND

该属性用于检查是否从结果集中提取到了数据。如果提取到数据，则返回值为 TRUE；如果未提取到数据，则返回值为 FALSE。示例如下：

```
LOOP
  FETCH c1 INTO var1,var2;    ——提取数据到变量中
  IF c1%FOUND THEN          ——如果提取到数据，则进行处理
    ...
  ELSE                      ——如果未提取到数据，则退出循环
    EXIT;
  END IF;
END LOOP;
```

### (3) %NOTFOUND

该属性与%FOUND 属性恰好相反。如果提取到数据，则返回值为 FALSE；如果没有提取到数据，则返回值为 TRUE。示例如下：

```

LOOP
    FETCH c1 INTO var1,var2;      --提取数据到变量中
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;

```

#### (4) %ROWCOUNT

该属性用于返回到当前行为止已经提取到的实际行数。示例如下：

```

LOOP
    FETCH c1 INTO my_ename, my_deptno;
    IF c1%ROWCOUNT > 10 THEN
        ...
    END IF;
    ...
END LOOP;

```

### 3. 显式游标使用示例

#### 示例一：在显式游标中使用 FETCH ... INTO 语句

在 Oracle9i 之前，使用 FETCH ... INTO 语句每次只能处理一行数据。为了处理结果集中的多行数据，必须要使用循环语句进行处理。下面以在 PL/SQL 块中显示部门 10 的所有雇员名及其工资为例，说明在显式游标中使用 FETCH .. INTO 语句的方法。示例如下：

```

DECLARE
    CURSOR emp_cursor IS
        SELECT ename,sal FROM emp WHERE deptno=10;
        v_ename emp.ename%TYPE;
        v_sal emp.sal%TYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_ename,v_sal;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line(v_ename||': '||v_sal);
    END LOOP;
    CLOSE emp_cursor;
END;
/
CLARK: 2450
KING: 5000
MILLER: 1300

```

#### 示例二：在显式游标中，使用 FETCH ... BULK COLLECT INTO 语句提取所有数据

从 Oracle9i 开始，通过使用 FETCH ... BULK COLLECT INTO 语句，一次就可以提取结果集的所有数据。下面以显示部门 10 的所有雇员名为例，说明使用 FETCH ... BULK COLLECT INTO 语句提取所有数据的方法。示例如下：

```

DECLARE
    CURSOR emp_cursor IS
        SELECT ename FROM emp WHERE deptno=10;

```

```

TYPE ename_table_type IS TABLE OF VARCHAR2(10);
ename_table ename_table_type;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor BULK COLLECT INTO ename_table;
  FOR i IN 1.. ename_table.COUNT LOOP
    dbms_output.put_line(ename_table(i));
  END LOOP;
  CLOSE emp_cursor;
END;
/
CLARK
KING
MILLER

```

### 示例三：在显式游标中使用 FETCH ... BULK COLLECT INTO .. LIMIT 语句提取部分数据

当使用 `FETCH .. BULK COLLECT INTO` 语句提取数据时，默认情况下会提取结果集的所有数据。如果结果集含有大量数据，并且使用 `VARRAY` 集合变量接收数据，那么可能需要限制每次提取的行数。下面以每次提取 5 行数据为例，说明使用 `LIMIT` 子句限制提取行的方法。示例如下：

```

DECLARE
  TYPE name_array_type IS VARRAY(5) OF VARCHAR2(10);
  name_array name_array_type;
  CURSOR emp_cursor IS SELECT ename FROM emp;
  rows INT:=5;
  v_count INT:=0;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor BULK COLLECT INTO name_array
      LIMIT rows;
    dbms_output.put('雇员名:');
    FOR i IN 1..(emp_cursor%ROWCOUNT-v_count) LOOP
      dbms_output.put(name_array(i)||' ');
    END LOOP;
    dbms_output.new_line;
    v_count:=emp_cursor%ROWCOUNT;
    EXIT WHEN emp_cursor%NOTFOUND;
  END LOOP;
  CLOSE emp_cursor;
END;
/
雇员名:SMITH ALLEN WARD JONES MARTIN
雇员名:BLAKE CLARK SCOTT KING TURNER
雇员名:ADAMS JAMES FORD MILLER

```

#### 示例四：使用游标属性

当使用显式游标时，为了取得显式游标的执行信息，需要使用显式游标属性。下面以使用显式游标属性%ISOPEN 和%ROWCOUNT 为例，说明在 PL/SQL 块中使用显式游标属性的方法。示例如下：

```

DECLARE
    CURSOR emp_cursor IS
        SELECT ename FROM emp WHERE deptno=10;
    TYPE ename_table_type IS TABLE OF VARCHAR2(10);
    ename_table ename_table_type;
BEGIN
    IF NOT emp_cursor%ISOPEN THEN --如果游标未打开，则打开游标
        OPEN emp_cursor;
    END IF;
    FETCH emp_cursor BULK COLLECT INTO ename_table;
    dbms_output.put_line('提取的总计行数:'|| emp_cursor%ROWCOUNT); --显示总计行数
    CLOSE emp_cursor;
END;
/
提取的总计行数:3

```

#### 示例五：基于游标定义记录变量

使用%ROWTYPE 属性不仅可以基于表和视图定义记录变量，也可以基于游标定义记录变量。当基于游标定义记录变量时，记录成员名实际就是 SELECT 语句的列名或列别名。为了简化显式游标的数据处理，建议开发人员使用记录变量存放游标数据。下面以显示所有雇员名及其工资为例，说明在处理显式游标数据时使用记录变量的方法。示例如下：

```

DECLARE
    CURSOR emp_cursor IS SELECT ename,sal FROM emp;
    emp_record emp_cursor%ROWTYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line('雇员名:'|| emp_record.ename||',雇员工资:'||emp_record.sal);
    END LOOP;
    CLOSE emp_cursor;
END;
/
雇员名:SMITH,雇员工资: 880
雇员名:ALLEN,雇员工资: 1936
雇员名:WARD,雇员工资: 1512.5
雇员名:JONES,雇员工资: 3272.5
雇员名:MARTIN,雇员工资: 1512.5
雇员名:BLAKE,雇员工资: 2300

```

```

雇员名:CLARK,雇员工资: 2695
雇员名:SCOTT,雇员工资: 1000
雇员名:KING,雇员工资: 5500
雇员名:TURNER,雇员工资: 1815
雇员名:ADAMS,雇员工资: 1210
雇员名:JAMES,雇员工资: 1149.5
雇员名:FORD,雇员工资: 3300
雇员名:MILLER,雇员工资: 1430

```

## 9.2 参数游标

参数游标是指带有参数的游标。在定义了参数游标之后，当使用不同参数值多次打开游标时，可以生成不同的结果集。定义参数游标的语法如下：

```
CURSOR cursor_name(parameter_name datatype) IS select_statement;
```

如上所示，当定义参数游标时，需要指定参数名及其数据类型。下面以显示特定部门所有雇员名为例，说明定义和使用参数游标的方法。示例如下：

```

DECLARE
    CURSOR emp_cursor(no NUMBER) IS
        SELECT ename FROM emp WHERE deptno=no;
        v_ename emp.ename%TYPE;
BEGIN
    OPEN emp_cursor(10);
    LOOP
        FETCH emp_cursor INTO v_ename;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line(v_ename);
    END LOOP;
    CLOSE emp_cursor;
END;
/
CLARK
KING
MILLER

```

注意，定义参数游标时，游标参数只能指定数据类型，而不能指定长度。另外，定义参数游标时，一定要在游标子查询的 WHERE 子句中引用该参数，否则失去了定义参数游标的意义。

## 9.3 使用游标更新或删除数据

通过使用显式游标，不仅可以一行一行地处理 SELECT 语句的结果，而且也可以更新或删除当前游标行的数据。注意，如果要通过游标更新或删除数据，在定义游标时必须要带有 FOR UPDATE 子句，语法如下：

```
CURSOR cursor_name(parameter_name datatype)
```

```

IS select_statement
FOR UPDATE [OF column_reference] [NOWAIT];

```

如上所示，FOR UPDATE 子句用于在游标结果集数据上加行共享锁，以防止其他用户在相应行上执行 DML 操作；当 SELECT 语句引用到多张表时，使用 OF 子句可以确定哪些表要加锁，如果没有 OF 子句，则会在 SELECT 语句所引用的全部表上加锁；NOWAIT 子句用于指定不等待锁。在提取了游标数据之后，为了更新或删除当前游标行数据，必须在 UPDATE 或 DELETE 语句中引用 WHERE CURRENT OF 子句。语法如下：

```

UPDATE table_name SET column=.. WHERE CURRENT OF cursor_name;
DELETE table_name WHERE CURRENT OF cursor_name;

```

### 1. 使用游标更新数据

下面以给工资低于 2000 的雇员增加 100 元工资为例，说明使用显式游标更新数据的方法。示例如下：

```

DECLARE
  CURSOR emp_cursor IS
    SELECT ename,sal FROM emp FOR UPDATE;
    v_ename emp.ename%TYPE;
    v_oldsal emp.sal%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_ename,v_oldsal;
    EXIT WHEN emp_cursor%NOTFOUND;
    IF v_oldsal<2000 THEN
      UPDATE emp SET sal=sal+100 WHERE CURRENT OF emp_cursor;
    END IF;
  END LOOP;
  CLOSE emp_cursor;
END;
/

```

### 2. 使用游标删除数据

下面以解雇部门 30 的所有雇员为例，说明使用显式游标删除数据的方法。示例如下：

```

DECLARE
  CURSOR emp_cursor IS
    SELECT ename,sal,deptno FROM emp FOR UPDATE;
    v_ename emp.ename%TYPE;
    v_oldsal emp.sal%TYPE;
    v_deptno emp.deptno%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_ename,v_oldsal,v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    IF v_deptno=30 THEN
      DELETE FROM emp WHERE CURRENT OF emp_cursor;
    END IF;
  END LOOP;
  CLOSE emp_cursor;
END;
/

```

```

    END IF;
END LOOP;
CLOSE emp_cursor;
END;
/

```

### 3. 使用 OF 子句在特定表上加行共享锁

如果游标子查询涉及到多张表，那么在默认情况下会在所有修改表行上加行共享锁。为了只在特定表上加行共享锁，需要在 FOR UPDATE 子句后带有 OF 子句。下面以显示所有雇员姓名、工资、部门名，并更新部门 30 的所有雇员工资为例，说明在显式游标中使用 OF 子句给特定表加锁的方法。示例如下：

```

DECLARE
  CURSOR emp_cursor IS
    SELECT ename,sal,dname,emp.deptno FROM emp,dept
    WHERE emp.deptno=dept.deptno
    FOR UPDATE OF emp.deptno;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    IF emp_record.deptno=30 THEN
      UPDATE emp SET sal=sal+100 WHERE CURRENT OF emp_cursor;
    END IF;
    dbms_output.put_line('雇员名:'||emp_record.ename
      ||',工资:'||emp_record.sal
      ||',部门名:'||emp_record.dname);
  END LOOP;
  CLOSE emp_cursor;
END;
/
雇员名:SMITH,工资:800,部门名:RESEARCH
雇员名:ALLEN,工资:1600,部门名:SALES
雇员名:WARD,工资:1250,部门名:SALES
雇员名:JONES,工资:2975,部门名:RESEARCH
雇员名:MARTIN,工资:1250,部门名:SALES
雇员名:BLAKE,工资:2850,部门名:SALES
雇员名:CLARK,工资:2450,部门名:ACCOUNTING
雇员名:KING,工资:5000,部门名:ACCOUNTING
雇员名:TURNER,工资:1500,部门名:SALES
雇员名:JAMES,工资:950,部门名:SALES
雇员名:FORD,工资:3000,部门名:RESEARCH
雇员名:MILLER,工资:1300,部门名:ACCOUNTING
雇员名:SCOTT,工资:3000,部门名:RESEARCH
雇员名:ADAMS,工资:1100,部门名:RESEARCH

```

#### 4. 使用 NOWAIT 子句

使用 FOR UPDATE 语句对被作用行加锁，如果其他会话已经在被作用行上加锁，那么在默认情况下当前会话会一直等待对方释放锁。通过在 FOR UPDATE 子句中指定 NOWAIT 语句，可以避免等待锁。当指定了 NOWAIT 子句之后，如果其他会话已经在被作用行加锁，那么当前会话会显示错误提示信息，并退出 PL/SQL 块。示例如下：

```

DECLARE
    CURSOR emp_cursor IS
        SELECT ename,sal FROM emp FOR UPDATE NOWAIT;
        v_ename emp.ename%TYPE;
        v_oldsal emp.sal%TYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_ename,v_oldsal;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_oldsal<2000 THEN
            UPDATE emp SET sal=sal+100 WHERE CURRENT OF emp_cursor;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
/
DECLARE
*
第 1 行出现错误:
ORA-00054: 资源忙, 但指定以 NOWAIT 方式获取资源
ORA-06512: 在 line 3
ORA-06512: 在 line 7

```

## 9.4 游标 FOR 循环

游标 FOR 循环是在 PL/SQL 块中使用游标最简单的方式，简化了对游标的处理。当使用游标 FOR 循环时，Oracle 会隐含地打开游标、提取游标数据并关闭游标。使用游标 FOR 循环的语法如下：

```

FOR record_name IN cursor_name LOOP
    statement1;
    statement2;
    ...
END LOOP;

```

如上所示，cursor\_name 是已经定义的游标名；record\_name 是 Oracle 隐含定义的记录变量名。当使用游标 FOR 循环时，在执行循环体内容之前，Oracle 会隐含地打开游标，并且每循环一次提取一次数据，在提取了所有数据之后，会自动退出循环并隐含地关闭游标。

### 1. 使用游标 FOR 循环

当使用游标开发 PL/SQL 应用程序时,为了简化程序代码,建议大家使用游标 FOR 循环。下面以顺序显示 EMP 表的所有雇员为例,说明使用游标 FOR 循环的方法。示例如下:

```
DECLARE
  CURSOR emp_cursor IS SELECT ename,sal FROM emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    dbms_output.put_line('第'||emp_cursor%ROWCOUNT|||
      '个雇员: '||emp_record.ename);
  END LOOP;
END;
/
第 1 个雇员: SMITH
第 2 个雇员: ALLEN
第 3 个雇员: WARD
第 4 个雇员: JONES
第 5 个雇员: MARTIN
第 6 个雇员: BLAKE
第 7 个雇员: CLARK
第 8 个雇员: SCOTT
第 9 个雇员: KING
第 10 个雇员: TURNER
第 11 个雇员: ADAMS
第 12 个雇员: JAMES
第 13 个雇员: FORD
第 14 个雇员: MILLER
```

### 2. 在游标 FOR 循环中直接使用子查询

当使用游标 FOR 循环时,习惯作法是首先在定义部分定义游标,然后在游标 FOR 循环中引用该游标。如果在使用游标 FOR 循环时不需要使用任何游标属性,那么可以直接在游标 FOR 循环中使用子查询。下面以显示 EMP 表的所有雇员名为例,说明在游标 FOR 循环中直接使用子查询的方法。示例如下:

```
BEGIN
  FOR emp_record IN
    (SELECT ename,sal FROM emp)  LOOP
    dbms_output.put_line(emp_record.ename);
  END LOOP;
END;
/
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
```

```

SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER

```

## 9.5 使用游标变量

类似于 C 语言的指针变量，PL/SQL 的游标变量中存放着指向内存地址的指针。当使用显式游标时，需要在定义部分指定其所对应的静态 SELECT 语句；而当使用游标变量时，开发人员可以在打开游标变量时指定其所对应的 SELECT 语句。开发人员可以在 OCI, Pro\*C/C++、FORMS 和 REPORTS 等应用中直接使用 PL/SQL 游标变量。

### 1. 游标变量使用步骤

在 PL/SQL 块中使用游标变量包括定义游标变量、打开游标、提取游标数据、关闭游标等四个阶段。具体步骤如下：

#### (1) 定义 REF CURSOR 类型和游标变量

为了在 PL/SQL 块中定义游标变量，必须首先定义 REF CURSOR 类型，然后才能定义游标变量。定义 REF CURSOR 类型和游标变量的语法如下：

```

TYPE ref_type_name IS REF CURSOR [RETURN return_type];
cursor_variable ref_type_name;

```

如上所示，`ref_type_name` 用于指定自定义类型名；`RETURN` 子句用于指定 REF CURSOR 返回结果的数据类型；`cursor_variable` 用于指定游标变量名。注意，当指定 `RETURN` 子句时，其数据类型必须是记录类型；另外，不能在包内定义游标变量。

#### (2) 打开游标

在定义了游标变量之后，为了引用该游标变量，在打开游标时需要指定其所对应的 SELECT 语句。当打开游标变量时，会执行游标变量所对应的 SELECT 语句，并将 SELECT 语句结果存放到游标结果集中。语法如下：

```
OPEN cursor_variable FOR select_statement;
```

如上所示，`select_statement` 用于指定游标所对应的 SELECT 语句。

#### (3) 提取游标数据

在打开游标之后，SELECT 语句的结果被临时存放到游标结果集中。为了处理结果集中的数据，需要使用 FETCH 语句提取游标数据。在 Oracle9i 之前，使用 FETCH 语句每次只能提取一行数据；从 Oracle9i 开始，通过使用 `FETCH .. BULK COLLECT INTO` 语句，每次可以提取多行数据。语法如下：

语法一：`FETCH cursor_variable INTO variable1,variable2,...;`

语法二：`FETCH cursor_variable
 BULK COLLECT INTO collect1, collect2,...[LIMIT rows];`

如上所示，`variable` 用于指定接收游标数据的变量；`collect` 用于指定接收游标结果的集合变量。注意，当使用语法一时，必须使用循环语句处理结果集的所有数据。

#### (4) 关闭游标变量

在提取并处理了所有游标数据之后，就可以关闭游标变量并释放其结果集了。语法如下：

```
CLOSE cursor_variable;
```

#### 2. 游标变量使用示例

##### 示例一：在定义 REF CURSOR 类型时不指定 RETURN 子句

如果在定义 REF CURSOR 类型时不指定 RETURN 子句，那么在打开游标时可以指定任何的 SELECT 语句。下面以顺序地显示部门 10 的所有雇员名称为例，说明使用游标变量（不包含 RETURN 子句）的方法。示例如下：

```
DECLARE
    TYPE emp_cursor_type IS REF CURSOR;
    emp_cursor emp_cursor_type;
    emp_record emp%ROWTYPE;
BEGIN
    OPEN emp_cursor FOR SELECT * FROM emp WHERE deptno=10;
    LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line('第'||emp_cursor%ROWCOUNT
            ||'个雇员:'||emp_record.ename);
    END LOOP;
    CLOSE emp_cursor;
END;
/
第 1 个雇员:CLARK
第 2 个雇员:KING
第 3 个雇员:MILLER
```

##### 示例二：在定义 REF CURSOR 类型时指定 RETURN 子句

如果在定义 REF CURSOR 类型时指定了 RETURN 子句，在打开游标时 SELECT 语句的返回结果必须与 RETURN 子句所指定的记录类型相匹配。下面以顺序地显示部门 20 的所有雇员名称为例，说明使用游标变量（包含 RETURN 子句）的方法。示例如下：

```
DECLARE
    TYPE emp_record_type IS RECORD(
        name VARCHAR2(10), salary NUMBER(6,2));
    TYPE emp_cursor_type IS REF CURSOR
        RETURN emp_record_type;
    emp_cursor emp_cursor_type;
    emp_record emp_record_type;
BEGIN
    OPEN emp_cursor FOR SELECT ename,sal FROM emp
        WHERE deptno=20;
    LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        dbms_output.put_line('第'||emp_cursor%ROWCOUNT
```

```

    ('个雇员:'||emp_record.name);
END LOOP;
CLOSE emp_cursor;
END;
/
第 1 个雇员:SMITH
第 2 个雇员:JONES
第 3 个雇员:SCOTT
第 4 个雇员:ADAMS
第 5 个雇员:FORD

```

如上所示，因为当定义 REF CURSOR 类型时指定了 RETURN 子句，所以游标子查询语句的返回结果必须与记录类型 emp\_record\_type 匹配。例如，如果在打开游标变量时指定“SELECT ename,sal,deptno FROM emp WHERE deptno=20”，那么在执行 PL/SQL 块时会显示如下错误信息：

```

OPEN emp_cursor FOR SELECT ename,sal,deptno FROM emp
*
ERROR 位于第 9 行:
ORA-06550: 第 9 行, 第 23 列:
PLS-00382: 表达式类型错误
ORA-06550: 第 9 行, 第 3 列:
PL/SQL: SQL Statement ignored

```

## 9.6 使用 CURSOR 表达式

CURSOR 表达式是 Oracle9i 新增加的特征，用于返回嵌套游标。在 Oracle9i 之前，使用显式游标时，结果集只能包含普通数据；从 Oracle9i 开始，结果集不仅可以包含普通数据，而且允许包含嵌套游标的数据。使用 CUSROR 表达式的语法如下：

```
CURSOR(subquery)
```

通过使用 CURSOR 表达式，开发人员可以在 PL/SQL 块中处理更加复杂的基于多张表的关联数据。为了在 PL/SQL 块中取得嵌套游标的数据，需要使用嵌套循环。下面以显示部门名、雇员名和工资为例，说明在 PL/SQL 块中使用 CURSOR 表达式的方法。示例如下：

```

DECLARE
  TYPE refcursor IS REF CURSOR;
  CURSOR dept_cursor(no NUMBER) IS
    SELECT a.dname,CURSOR(SELECT ename,sal FROM emp
      WHERE deptno=a.deptno)
      FROM dept a WHERE a.deptno=no;
  empcur refcursor;
  v_dname dept.dname%TYPE;
  v_ename emp.ename%TYPE;
  v_sal emp.sal%TYPE;
BEGIN
  OPEN dept_cursor(&no);
  LOOP

```

```

      FETCH dept_cursor INTO v_dname,empcur;
      EXIT WHEN dept_cursor%NOTFOUND;
      dbms_output.put_line('部门名:'||v_dname);
      LOOP
      FETCH empcur INTO v_ename,v_sal;
      EXIT WHEN empcur%NOTFOUND;
      dbms_output.put_line('雇员名:'||v_ename||
      ',工资:'||v_sal);
      END LOOP;
      END LOOP;
      CLOSE dept_cursor;
END;
/
输入 no 的值: 10
部门名:ACCOUNTING
雇员名:CLARK, 工资:2450
雇员名:KING, 工资:5000
雇员名:MILLER, 工资:1300

```

## 9.7 习题

1. 当使用显式游标时，在执行了哪条语句后应该检查游标是否包含行？
  - A. OPEN
  - B. FETCH
  - C. CLOSE
  - D. CURSOR
2. 在以下哪些语句中可以包含 WHERE CURRENT OF 子句？
  - A. OPEN
  - B. FETCH
  - C. DELETE
  - D. SELECT
  - E. UPDATE
  - F. CURSOR
3. 查看以下游标定义语句，哪行会引起错误？
  1. DECLARE
  2. CURSOR cust\_cursor (p\_cust\_id, p\_last\_name)
  3. IS
  4.     SELECT cust\_id, first\_name, last\_name, credit\_limit
  5.     FROM customer
  6.     WHERE cust\_id = p\_cust\_id
  7.     AND last\_name = p\_last\_name;
  - A. 2
  - B. 3
  - C. 4
  - D. 5
  - E. 6
4. 编写 PL/SQL 块，定义显式游标显示订单号、客户名以及订单总价，并以订单总价的降序排序。格式如下：
 

订单号:612,客户名:BLAKE,      总价:5860  
  订单号:610,客户名:BOB,      总价:101.4  
  订单号:601,客户名:CLARK,      总价:60.8  
  ...
5. 编写 PL/SQL 块，使用游标 FOR 循环更新订单交付日期。如果交付日期与预订日期相

差在 15 天之内，则不需要更新交付日期；如果交付日期与预订日期相差超过 15 天，则显示订单号、预订日期和交付日期，然后将交付日期更新为预订日期之后的第 15 天。格式如下：

订单号:601,预订日期:01-5 月 -90,交付日期:30-5 月 -90

订单号:600,预订日期:01-5 月 -90,交付日期:29-5 月 -90

6. 编写 PL/SQL 块，使用参数游标输入订单号，然后显示该订单的每个条款号、条款总价和产品名（使用 CURSOR 表达式）。格式如下：

输入 id 的值： 600

条款号:1,总价:42,产品名:ACE TENNIS RACKET II

条款号:2,总价:58,产品名:ACE TENNIS NET

# 第 10 章 处理例外

当开发 PL/SQL 应用程序时，为了提高应用程序的健壮性，开发人员必须考虑 PL/SQL 程序可能出现的各种错误，并进行相应的错误处理。如果不进行错误处理，在出现运行错误时，会终止 PL/SQL 程序的运行，并显示错误信息。在编写 PL/SQL 程序时，通过使用例外（Exception）可以处理运行错误。本章将详细介绍使用例外处理 PL/SQL 运行错误的方法，在学习了本章之后，读者应该学会：

- 使用预定义例外；
- 使用非预定义例外；
- 使用自定义例外；
- 使用 Oracle 提供的例外处理函数；
- 使用初始化参数 PLSQL\_WARNINGS 和 PL/SQL 包 DBMS\_WARNING 生成警告信息。

## 10.1 例外简介

例外（Exception）是一种 PL/SQL 标识符。如果运行 PL/SQL 块时出现错误或警告，则会触发例外。当触发例外时，默认情况下会终止 PL/SQL 块的执行。通过在 PL/SQL 块中引入例外处理部分，可以捕捉各种例外，并根据例外出现的情况进行相应的处理。

### 1. 例外分类

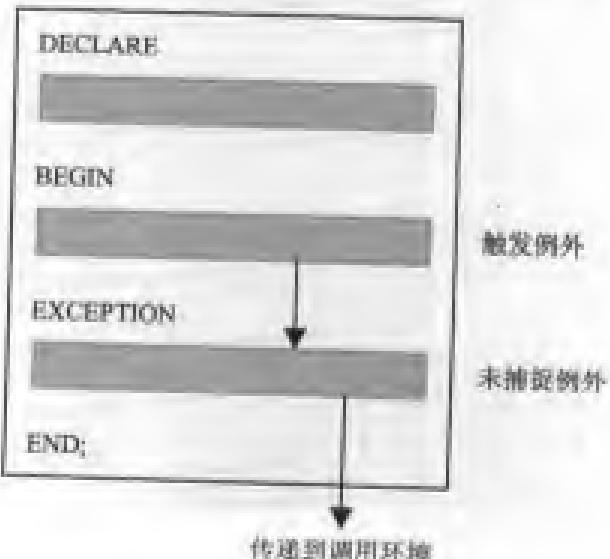
为了处理 PL/SQL 应用程序的各种错误，开发人员可以使用各种类型的例外。Oracle 提供了预定义例外、非预定义例外和自定义例外等三种例外类型。其中，预定义例外用于处理常见的 Oracle 错误；非预定义例外则用于处理预定义例外所不能处理的 Oracle 错误；自定义例外则用于处理与 Oracle 错误无关的其他情况。

### 2. 处理例外

编写 PL/SQL 应用程序时，为了提高程序的健壮性，开发人员应该捕捉可能出现的各种例外，并进行合适的处理。如果不捕捉例外，Oracle 会在出现错误时将错误传递到调用环境；如果捕捉到例外，Oracle 会在 PL/SQL 块内解决运行错误。处理例外可以采用以下两种方法：

#### （1）传递例外

如果在运行 PL/SQL 应用程序时出现例外，并且在例外处理部分没有捕捉到该例外，Oracle 会将该例外传递到调用块或 PL/SQL 运行环境。如右图所示：



如图中所示，如果在例外处理部分（EXCEPTION）没有捕捉例外，Oracle 会将例外传递到调用环境。以下示例的 PL/SQL 块用于根据输入的雇员号显示雇员名，如果雇员号在 EMP 表中不存在，则会触发 NO\_DATA\_FOUND 例外。因为没有捕捉到该例外，所以会在 SQL\*Plus 调用环境中显示相应的错误信息。示例如下：

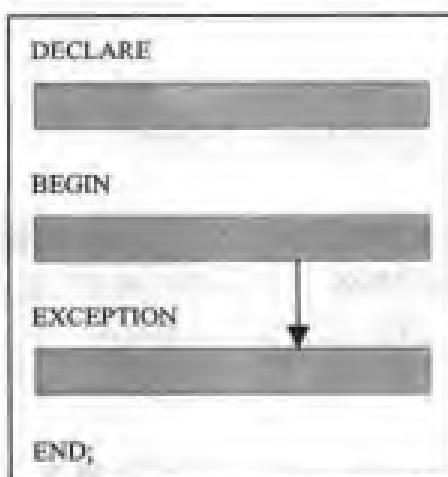
```

DECLARE
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp
        WHERE empno=&no;
    dbms_output.put_line('雇员名:'||v_ename);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('查询只能返回单行');
END;
/
输入 no 的值: 1111
DECLARE
    *
ERROR 位于第 1 行:
ORA-01403: 未找到数据
ORA-06512: 在 line 4

```

## (2) 捕捉并处理例外

为了提高 PL/SQL 应用程序的健壮性，开发人员应该预计到应用程序可能出现的各种错误，并使用例外处理部分有效地解决各种可能错误，从而为最终用户提供更加合理的帮助信息。PL/SQL 程序处理例外的另一种方法就是捕捉并处理例外，如下图所示：



在 PL/SQL 块中捕捉并处理例外需要使用例外处理部分来完成，例外处理部分是以关键字 EXCEPTION 开始的，语法如下所示：

```

EXCEPTION
    WHEN exception1 [OR exception2 . . .] THEN
        statement1;

```

```

statement2;
. . .
[WHEN exception3 [OR exception4 . . .] THEN
statement1;
statement2;
. . .
[WHEN OTHERS THEN
statement1;
statement2;
. . .

```

如上所示，例外处理部分以关键字 EXCEPTION 开始，在例外处理部分中可以使用 WHEN 子句捕捉各种例外，如果还有其他未预计到的例外，可以使用 WHEN OTHERS 子句进行捕捉和处理。注意，WHEN OTHERS 必须是例外处理部分的最后一条子句。以下示例的 PL/SQL 块用于根据输入的雇员号显示雇员名，如果雇员号在 EMP 表中不存在，则会触发 NO\_DATA\_FOUND 例外。为了合理地处理这种异常情况，在编写 PL/SQL 程序时应该捕捉 NO\_DATA\_FOUND 例外，为用户提供更有意义的信息。示例如下：

```

DECLARE
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE empno=&no;
    dbms_output.put_line('雇员名:'||v_ename);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('雇员号不正确，请核实雇员号！');
END;
/
输入 no 的值： 1111
雇员号不正确，请核实雇员号！

```

## 10.2 处理预定义例外

预定义例外是指由 PL/SQL 所提供的系统例外。当 PL/SQL 应用程序违反了 Oracle 规则或系统限制时，则会隐含地触发一个内部例外。为了处理各种常见的 Oracle 错误，PL/SQL 为开发人员提供了二十多个预定义例外，每个预定义例外都对应一个 Oracle 系统错误。下面将介绍这些预定义例外。

### 1. 常用预定义例外

当使用预定义例外时，应该了解 PL/SQL 块的最常见运行错误，并掌握与之相关的预定义例外。下面介绍常见的预定义例外，例外所对应的 Oracle 错误，以及捕捉并处理这些例外的方法。

#### (1) ACCESS INTO NULL

该例外对应于 ORA-06530 错误。当开发对象类型应用时，在引用对象属性之前，必须首先初始化对象。如果没有初始化对象，直接为对象属性赋值，就会隐含地触发 PL/SQL 例外

**ACCESS\_INTO\_NULL**。假定使用如下语法建立了对象类型:

```
CREATE TYPE emp_type AS OBJECT
  (name VARCHAR2(10), sal NUMBER(6,2));
  /
```

当在 PL/SQL 块中使用对象类型 emp\_type 的对象属性 name 和 sal 时, 必须首先初始化对象, 否则会隐含地触发例外 ACCESS\_INTO\_NULL。捕捉并处理该例外的示例如下:

```
DECLARE
  emp emp_type;
BEGIN
  emp.name:='SCOTT';
EXCEPTION
  WHEN ACCESS_INTO_NULL THEN
    dbms_output.put_line('首先初始化对象 emp');
END;
/
首先初始化对象 emp
```

### (2) CASE\_NOT\_FOUND

该例外对应于 ORA-06592 错误。当在 PL/SQL 块中编写 CASE 语句时, 如果在 WHEN 子句中没有包含必须的条件分支, 并且没有包含 ELSE 子句, 就会隐含地触发 CASE\_NOT\_FOUND 例外。捕捉并处理该例外的示例如下:

```
undef no
DECLARE
  v_sal emp.sal%TYPE;
BEGIN
  SELECT sal INTO v_sal FROM emp WHERE empno=&no;
CASE
  WHEN v_sal<1000 THEN
    UPDATE emp SET sal=sal+100 WHERE empno=&no;
  WHEN v_sal<2000 THEN
    UPDATE emp SET sal=sal+150 WHERE empno=&no;
  WHEN v_sal<3000 THEN
    UPDATE emp SET sal=sal+200 WHERE empno=&no;
END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    dbms_output.put_line('在 CASE 语句中缺少与'||v_sal||'相关的条件');
END;
/
输入 no 的值: 7839
在 CASE 语句中缺少与 5000 相关的条件
```

### (3) COLLECTION\_IS\_NULL

该例外对应于 ORA-06531 错误。在给集合元素 (嵌套表或 VARRAY 类型) 赋值前, 必须首先初始化集合元素。如果没有初始化集合元素, 则会隐含地触发 COLLECTION\_IS\_NULL

例外。捕捉并处理该例外的示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    ename_table ename_table_type;
BEGIN
    SELECT ename INTO ename_table(2) FROM emp
        WHERE empno=&no;
    dbms_output.put_line('雇员名:'||ename_table(2));
EXCEPTION
    WHEN COLLECTION_IS_NULL THEN
        dbms_output.put_line('必须使用构造方法初始化集合元素');
END;
/
输入 no 的值: 7788
必须使用构造方法初始化集合元素

```

#### (4) CURSOR\_ALREADY\_OPEN

该例外对应于 ORA-06511 错误。当重新打开已经打开的游标时，会隐含地触发例外 CURSOR\_ALREADY\_OPEN。例如，如果用户已经使用 OPEN 命令打开了显式游标，并执行游标 FOR 循环，就会隐含地触发该例外（游标 FOR 循环会隐含地打开游标）。捕捉并处理该例外的示例如下：

```

DECLARE
    CURSOR emp_cursor IS SELECT ename,sal FROM emp;
BEGIN
    OPEN emp_cursor;
    FOR emp_record IN emp_cursor LOOP
        dbms_output.put_line(emp_record.ename);
    END LOOP;
EXCEPTION
    WHEN CURSOR_ALREADY_OPEN THEN
        dbms_output.put_line('游标已经打开');
END;
/
游标已经打开

```

#### (5) DUP\_VAL\_ON\_INDEX

该例外对应于 ORA-00001 错误。当在惟一索引所对应的列上键入重复值时，会隐含地触发例外 DUP\_VAL\_ON\_INDEX。捕捉并处理该例外的示例如下：

```

BEGIN
    UPDATE dept SET deptno=&new_no WHERE deptno=&old_no;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('在 deptno 列上不能出现重复值');
END;
/
输入 new_no 的值: 10
输入 old_no 的值: 40

```

在deptno列上不能出现重复值

#### (6) INVALID\_CURSOR

该例外对应于ORA-01001错误。当试图在不合法的游标上执行操作时，会隐含地触发例外INVALID\_CURSOR。例如，如果要从未打开的游标提取数据，或者关闭未打开的游标，则会触发该例外。捕捉并处理该例外的示例如下：

```
DECLARE
    CURSOR emp_cursor IS SELECT ename,sal FROM emp;
    emp_record emp_cursor%ROWTYPE;
BEGIN
    FETCH emp_cursor INTO emp_record;
    CLOSE emp_cursor;
EXCEPTION
    WHEN INVALID_CURSOR THEN
        dbms_output.put_line('请检查游标是否已经打开');
END;
/
请检查游标是否已经打开
```

#### (7) INVALID\_NUMBER

该例外对应于ORA-01722错误。当内嵌SQL语句不能有效地将字符转变成数字时，会隐含地触发例外INVALID\_NUMBER，例如数字值“100”被写成“1OO”。捕捉并处理该例外的示例如下：

```
BEGIN
    UPDATE emp SET sal=sal+'100';
EXCEPTION
    WHEN INVALID_NUMBER THEN
        dbms_output.put_line('输入的数字值不正确');
END;
/
输入的数字值不正确
```

#### (8) NO\_DATA\_FOUND

该例外对应于ORA-01403错误。当执行SELECT INTO未返回行，或者引用了索引表未初始化的元素时，会隐含地触发例外NO\_DATA\_FOUND。捕捉并处理该例外的示例如下：

```
DECLARE
    v_sal emp.sal%TYPE;
BEGIN
    SELECT sal INTO v_sal FROM emp
        WHERE lower(ename)=lower('&name');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('不存在该雇员');
END;
/
输入 name 的值： MARY
不存在该雇员
```

### (9) TOO\_MANY\_ROWS

该例外对应于 ORA-01422 错误。当执行 SELECT INTO 语句时，如果返回超过一行，则会触发该例外。捕捉并处理该例外的示例如下：

```
DECLARE
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE sal=&sal;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        dbms_output.put_line('返回多行');
END;
/
输入 sal 的值： 3000
返回多行
```

### (10) ZERO\_DIVIDE

该例外对应于 ORA-01476 错误。当运行 PL/SQL 块时，如果使用数字值除 0，则会隐含地触发该例外。捕捉并处理该例外的示例如下：

```
DECLARE
    num1 INT:=100;
    num2 INT:=0;
    num3 NUMBER(6,2);
BEGIN
    num3:=num1/num2;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        dbms_output.put_line('分母不能为 0');
END;
/
分母不能为 0
```

### (11) SUBSCRIPT\_BEYOND\_COUNT

该例外对应于 ORA-06533 错误。当使用嵌套表或 VARRAY 元素时，如果元素下标超出了嵌套表或 VARRAY 元素的范围，则会隐含地触发 SUBSCRIPT\_BEYOND\_COUNT 例外。捕捉并处理该例外的示例如下：

```
DECLARE
    TYPE emp_array_type IS VARRAY(20) OF VARCHAR2(10);
    emp_array emp_array_type;
BEGIN
    emp_array:=emp_array_type('SCOTT','MARY');
    dbms_output.put_line(emp_array(3));
EXCEPTION
    WHEN SUBSCRIPT_BEYOND_COUNT THEN
        dbms_output.put_line('超出下标范围');
END;
/
```

超出下标范围

### (12) SUBSCRIPT\_OUTSIDE\_LIMIT

该例外对应于 ORA-06532 错误。当使用嵌套表或 VARRAY 元素时，如果元素下标为负值，则会隐含地触发 SUBSCRIPT\_BEYOND\_COUNT 例外。捕捉并处理该例外的示例如下：

```
DECLARE
    TYPE emp_array_type IS VARRAY(20) OF VARCHAR2(10);
    emp_array emp_array_type;
BEGIN
    emp_array:=emp_array_type('SCOTT','MARY');
    dbms_output.put_line(emp_array(-1));
EXCEPTION
    WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN
        dbms_output.put_line('嵌套表和 VARRAY 下标不能为负');
END;
/
嵌套表和 VARRAY 下标不能为负
```

### (13) VALUE\_ERROR

该例外对应于 ORA-06502 错误。当在 PL/SQL 块中执行赋值操作时，如果变量长度不足以容纳实际数据，则会隐含地触发例外 VALUE\_ERROR。捕捉并处理该例外的示例如下：

```
DECLARE
    v_ename VARCHAR2(5);
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE empno=&no;
    dbms_output.put_line(v_ename);
EXCEPTION
    WHEN VALUE_ERROR THEN
        dbms_output.put_line('变量尺寸不足');
END;
/
输入 no 的值： 7934
变量尺寸不足
```

## 2. 其他预定义例外

除了在 PL/SQL 块中经常需要处理常见的预定义例外之外，某些情况下你可能还需要使用其他预定义例外。下面简单介绍这些例外的作用。

### (1) LOGIN\_DENIED

该例外对应于 ORA-01017 错误。当 PL/SQL 应用程序需要连接到 Oracle 数据库时，如果提供了不正确的用户名或口令，则会隐含地触发例外 LOGIN\_DENIED。

### (2) NOT\_LOGGED\_ON

该例外对应于 ORA-01012 错误。如果应用程序没有连接到 Oracle 数据库，那么在执行 PL/SQL 块中访问数据库时会隐含地触发例外 NOT\_LOGGED\_ON。

### (3) PROGRAM\_ERROR

该例外对应于 ORA-06501 错误。如果出现该错误，则表示存在 PL/SQL 内部问题，用户此时可能需要重新安装数据字典和 PL/SQL 系统包。

#### (4) ROWTYPE MISMATCH

该例外对应于 ORA-06504 错误。当执行赋值操作时，如果宿主游标变量和 PL/SQL 游标变量的返回类型不兼容，那么会隐含地触发例外 ROWTYPE\_MISMATCH。

#### (5) SELF\_IS\_NULL

该例外对应于 ORA-30625 错误。当使用对象类型时，如果在 null 实例上调用成员方法，则会隐含地触发例外 SELF\_IS\_NULL。

#### (6) STORAGE\_ERROR

该例外对应于 ORA-06500 错误。PL/SQL 块运行时，如果超出内存空间或者内存被损坏，则会隐含地触发例外 STORAGE\_ERROR。

#### (7) SYS\_INVALID\_ROWID

该例外对应于 ORA-01410 错误。当将字符串转变为 ROWID 时，必须使用有效的字符串。如果使用了无效的字符串，会隐含地触发例外 SYS\_INVALID\_ROWID。

#### (8) TIMEOUT\_ON\_RESOURCE

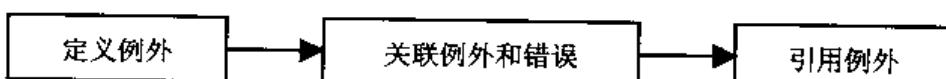
该例外对应于 ORA-00051 错误。当如果 Oracle 在等待资源时出现超时错误，则会隐含地触发例外 TIMEOUT\_ON\_RESOURCE。

### 10.3 处理非预定义例外

非预定义例外用于处理与预定义例外无关的 Oracle 错误。使用预定义例外，只能处理 21 个 Oracle 错误。而当使用 PL/SQL 开发应用程序时，可能还会遇到其他的一些 Oracle 错误。例如，如果在 PL/SQL 块中执行 DML 语句，并且数据违反了约束规则，则会直接将 Oracle 错误传递到调用环境。示例如下：

```
BEGIN
    UPDATE emp SET deptno=&dno WHERE empno=&eno;
END;
/
输入 dno 的值: 11
输入 eno 的值: 7788
BEGIN
*
ERROR 位于第 1 行:
ORA-02291: 违反完整约束条件 (SCOTT.FK_DEPTNO) - 未找到父项关键字
ORA-06512: 在 line 2
```

为了提高 PL/SQL 程序的健壮性，应该在 PL/SQL 应用程序中合理地处理这些 Oracle 错误，此时就需要使用非预定义例外。使用非预定义例外的步骤如下图所示：



如图所示，使用非预定义例外包括三步：首先在定义部分定义例外名，然后在例外和 Oracle 错误之间建立关联，最终在例外处理部分捕捉并处理例外。当定义 Oracle 错误和例外之间的关联关系时，需要使用伪过程 EXCEPTION\_INIT。下面以在 PL/SQL 块中处理 ORA-02291 错

误为例，说明使用非预定义例外的方法。示例如下：

```

DECLARE
    e_integrity EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
    UPDATE emp SET deptno=&dno WHERE empno=&eno;
EXCEPTION
    WHEN e_integrity THEN
        dbms_output.put_line('该部门不存在');
END;
/
输入 dno 的值： 11
输入 eno 的值： 7788
该部门不存在

```

如例所示，因为在 DEPT 表和 EMP 表之间具有主外键关系，所以当修改雇员的部门号时，部门号必须在 DEPT 表中存在。如果该部门号在表中不存在，则会隐含触发 ORA-02291 对应的例外 e\_integrity，并显示合理的输出信息。

## 10.4 处理自定义例外

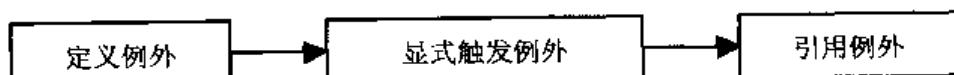
自定义例外是指由 PL/SQL 开发人员所定义的例外。预定义例外和非预定义例外都与 Oracle 错误有关，并且出现 Oracle 错误时会隐含触发相应例外；而自定义例外与 Oracle 错误没有任何关联，它是由开发人员为特定情况所定义的例外。看一看以下 PL/SQL 块的运行：

```

DECLARE
    e_integrity EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
    UPDATE emp SET deptno=&dno WHERE empno=&eno;
EXCEPTION
    WHEN e_integrity THEN
        dbms_output.put_line('该部门不存在');
END;
/
输入 dno 的值： 10
输入 eno 的值： 1111

```

如例所示，尽管雇员 1111 不存在，但在执行 UPDATE 操作时不会显示任何错误信息。为了给用户提供更加有用、更有意义的信息，在这种情况下应该提示“该雇员不存在”的信息。为了完成这项任务，在 PL/SQL 块中就应该使用自定义例外。与预定义例外和非预定义例外不同，自定义例外必须显式触发。使用自定义例外的步骤如下图所示：



如图所示，当使用自定义例外时，首先需要在定义部分（DECLARE）定义例外，然后在

执行部分 (BEGIN) 触发例外 (使用 RAISE 语句), 最后在例外处理部分 (EXCEPTION) 捕捉并处理例外。下面以处理前例情况为例, 说明使用自定义例外的方法。示例如下:

```

DECLARE
    e_integrity EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_integrity,-2291);
    e_no_employee EXCEPTION;
BEGIN
    UPDATE emp SET deptno=&dno WHERE empno=&eno;
    IF SQL%NOTFOUND THEN
        RAISE e_no_employee;
    END IF;
EXCEPTION
    WHEN e_integrity THEN
        dbms_output.put_line('该部门不存在');
    WHEN e_no_employee THEN
        dbms_output.put_line('该雇员不存在');
END;
/
输入 dno 的值: 10
输入 eno 的值: 1111
该雇员不存在

```

## 10.5 使用例外函数

当在 PL/SQL 块中出现 Oracle 错误时, 通过使用例外函数可以取得错误号以及相关的错误消息, 其中函数 SQLCODE 用于取得 Oracle 错误号, 而 SQLERRM 则用于取得与之相关的错误消息。另外, 通过在存储过程、函数和包中使用 RAISE\_APPLICATION\_ERROR 可以自定义错误号和错误消息。

### 1. SQLCODE 和 SQLERRM

SQLCODE 用于返回 Oracle 错误号, 而 SQLERRM 则用于返回该错误号所对应的错误消息。为了在 PL/SQL 应用程序中处理其他未预料到的 Oracle 错误, 用户可以在例外处理部分的 WHEN OTHERS 子句后引用这两个函数, 以取得相关的 Oracle 错误。示例如下:

```

undef v_sal
DECLARE
    v_ename emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp
    WHERE sal=&v_sal;
    dbms_output.put_line('雇员名:'||v_ename);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('不存在工资为'||&v_sal||'的雇员');
    WHEN OTHERS THEN
        dbms_output.put_line('错误号:'||SQLCODE);

```

```

    dbms_output.put_line(SQLERRM);
END;
/
输入 v_sal 的值: 3000
错误号:-1422
ORA-01422: 实际返回的行数超出请求的行数

```

## 2. RAISE\_APPLICATION\_ERROR

该过程用于在 PL/SQL 应用程序中自定义错误消息。注意，该过程只能在数据库端的子程序（过程、函数、包、触发器）中使用，而不能在匿名块和客户端的子程序中使用。使用该过程的语法如下：

```
raise_application_error(error_number, message[, {TRUE | FALSE}]);
```

其中，`error_number` 用于定义错误号，该错误号必须是在 -20000 到 -20999 之间的负整数；`message` 用于指定错误消息，并且该消息的长度不能超过 2048 字节；第三个参数为可选参数，如果设置为 `TRUE`，则该错误会被放在先前错误堆栈中；如果设置为 `FALSE`（默认值），则会替换先前所有错误。使用该过程的示例如下：

```

CREATE OR REPLACE PROCEDURE raise_comm
(eno NUMBER, commission NUMBER)
IS
    v_comm emp.comm%TYPE;
BEGIN
    SELECT comm INTO v_comm FROM emp WHERE empno=eno;
    IF v_comm IS NULL THEN
        RAISE_APPLICATION_ERROR(-20001, '该雇员无补助');
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('该雇员不存在');
END;
/

```

如上所示，在建立了过程 `raise_comm` 之后，就可以在应用程序或 SQL\*Plus 中调用该过程，如果雇员补助为 `NULL`，则会显示自定义的错误消息。示例如下：

```

SQL> exec raise_comm(7788,100)
BEGIN raise_comm(7788,100); END;
*
ERROR 位于第 1 行:
ORA-20001: 该雇员无补助
ORA-06512: 在"SCOTT.RAISE_COMM", line 9
ORA-06512: 在 line 1

```

## 10.6 PL/SQL 编译警告

在 Oracle 10g 之前，编写 PL/SQL 子程序时，只要子程序符合 SQL 和 PL/SQL 的语法及语义规则，Oracle 就会成功地编译子程序，并且不会提示任何其他信息。例如下面的子程序

## DEAD CODE:

```

CREATE OR REPLACE PROCEDURE dead_code AS
    x number := 10;
BEGIN
    if x = 10 then
        x := 20;
    else
        x := 100; -- 死代码 (永远不会执行)
    end if;
END dead_code;
/

```

如例所示, else 子句后的相关代码行从来都不执行, 该段代码实际就是死代码。在编写 PL/SQL 程序时, 应该避免出现类似的死代码。从 Oracle 10g 开始, 为了提高 PL/SQL 子程序的健壮性并避免运行错误, 在编写 PL/SQL 子程序之前开发人员可以激活警告检查。在激活了警告检查之后, 当编写 PL/SQL 子程序时就可以检查子程序的未定义结果以及可能带来的性能问题。

### 1. PL/SQL 警告分类

PL/SQL 警告可分成三类警告消息, 通过使用不同的警告类型, 可以禁止或显示特定警告消息。PL/SQL 警告分类如下:

- SEVERE: 该种警告用于检查可能出现的不可预料结果或错误结果, 例如参数的别名问题。
- PERFORMANCE: 该类警告用于检查可能引起的性能问题, 例如在执行 INSERT 操作时为 NUMBER 列提供了 VARCHAR2 类型的数据。
- INFORMATIONAL: 该类警告用于检查子程序中的死代码。
- ALL: 该关键字用于检查所有警告 (SEVERE, PERFORMANCE, INFORMATIONAL)。

### 2. 控制 PL/SQL 警告消息

为了使得数据库可以在编译 PL/SQL 子程序时发出警告消息, 需要设置初始化参数 PLSQL\_WARNINGS。初始化参数 PLSQL\_WARNINGS 不仅可以在系统级或会话级设置, 也可以在 ALTER PROCEDURE 命令中进行设置。当设置初始化参数 PLSQL\_WARNINGS 时, 既可以激活或禁止所有警告类型 (ALL, SEVERE, PERFORMANCE, INFORMATIONAL), 也可以激活或禁止特定消息号。例如:

```

SQL> ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL';
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
SQL> ALTER PROCEDURE hello COMPILE
  2> PLSQL_WARNINGS='ENABLE:PERFORMANCE';
SQL> ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE',
  2> 'DISABLE:PERFORMANCE', 'ERROR:06002';

```

当激活或禁止 PL/SQL 编译警告时, 不仅可以使用 ALTER SYSTEM, ALTER SESSION 和 ALTER PROCEDURE 命令, 还可以使用 PL/SQL 系统包 DBMS\_WARNINGS。示例如下:

```

SQL> CALL DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:ALL',
'SESSION');

```

### 3. 使用 PL/SQL 编译警告

#### (1) 检测死代码

死代码是指在 PL/SQL 子程序中从来不会被执行的代码。当子程序中带有死代码时，不仅会占用内存空间，而且会降低程序执行效率，所以编写 PL/SQL 子程序时应该避免死代码。从 Oracle 10g 开始，使用 PL/SQL 编译警告可以检测死代码。在介绍如何检测死代码之前，应首先建立包含死代码的过程 DEAD\_CODE。示例如下：

```
CREATE OR REPLACE PROCEDURE dead_code AS
  x number := 10;
BEGIN
  if x = 10 then
    x := 20;
  else
    x := 100; -- 死代码（永远不会执行）
  end if;
END dead_code;
/
```

为了检测该子程序是否包含死代码，必须首先激活警告检查，然后重新编译子程序，最后使用 SHOW ERRORS 命令显示警告错误。示例如下：

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:INFORMATIONAL';
SQL> ALTER PROCEDURE dead_code COMPILE;
SP2-0805: 过程已变更，但带有编译警告
SQL> show errors
PROCEDURE DEAD_CODE 出现错误:
LINE/COL ERROR
-----
7/5      PLW-06002: Message 6002 not found; No message file for
                  product=plsql, facility=PLW
```

#### (2) 检测引起性能问题的代码

编写 PL/SQL 子程序时，如果数值与变量的数据类型不符合，Oracle 会隐含地转换数据类型。但因为数据类型转换会影响子程序性能，所以在编写 PL/SQL 子程序时应该尽可能避免性能问题。在介绍如何检测引起性能问题的代码之前，应首先建立包含隐含数据类型转换的过程 UPDATE\_SAL。示例如下：

```
CREATE OR REPLACE PROCEDURE update_sal
(name VARCHAR2,salary VARCHAR2)
IS
BEGIN
  UPDATE emp SET sal=salary WHERE ename=name;
END;
/
```

为了检测该子程序是否会引起性能问题，应首先激活警告检查，然后重新编译子程序，最后再使用 SHOW ERRORS 命令显示警告错误。示例如下：

```
SQL> ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
SQL> ALTER PROCEDURE update_sal COMPILE;
```

SP2-0805: 过程已变更，但带有编译警告

SQL> show errors

PROCEDURE UPDATE\_SAL 出现错误:

LINE/COL ERROR

-----  
5/3 PLW-07202: Message 7202 not found; No message file for  
product=plsql, facility=PLW

## 10.7 习题

1. RAISE 语句应该放在 PL/SQL 块的哪个部分?
  - A. 子程序头部
  - B. 定义部分
  - C. 执行部分
  - D. 例外处理部分
2. 在例外和 Oracle 错误之间建立关联时，应该在哪个部分完成?
  - A. 定义部分
  - B. 执行部分
  - C. 例外处理部分
3. 假定在 EMP 表上定义了 CHECK 约束，要求雇员工资不能高于 6000。为了处理工资超过 6000 可能出现的错误，应该使用哪种例外?
  - A. 预定义例外
  - B. 非预定义例外
  - C. 自定义例外
4. 哪个是使用过程 raise\_application\_error 的原因?
  - A. 捕捉自定义例外
  - B. 捕捉预定义例外
  - C. 捕捉非预定义例外
  - D. 发出用户自定义的错误消息
5. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入订单号，显示交付日期和总价，并处理 NO\_DATA\_FOUND 例外。如果不存在该订单，则显示消息“请检查订单号，并输入正确的订单号”。
  6. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入订单号和客户号，然后更新订单所对应的客户号，并处理可能出现的错误。
    - 如果成功地修改了订单客户，则显示消息“订单..的新客户号为..”。
    - 如果不存在该订单，则显示消息“请检查订单号，然后输入正确的订单号”。
    - 如果不存在该客户，则显示消息“请检查客户号，然后输入正确的客户号”
  7. 编写 PL/SQL 块，使用 SQL\*Plus 替代变量输入客户号，删除该客户的信息，并处理可能出现的错误。
    - 如果成功地删除了客户，则显示消息“删除了客户...”。
    - 如果客户不存在，则显示消息“客户不存在”。
    - 如果违反了完整性约束，则显示消息“有订单的客户不能被删除”。

# 第 11 章 开发子程序

子程序是指被命名的 PL/SQL 块，这种块可以带有参数，可以在不同应用程序中多次调用。PL/SQL 有两种类型的子程序：过程和函数，其中过程用于执行特定操作，而函数则用于返回特定数据。当开发 PL/SQL 子程序时，既可以开发客户端的子程序，也可以开发服务器端的子程序。客户端子程序主要用于 Developer 应用（Forms, Reports 和 Graphics）中，而服务器端子程序可以用于任何应用程序中。通过将商业逻辑和企业规则集成到 PL/SQL 子程序中，可以简化客户端应用的开发和维护，并提高应用程序的性能。本章将详细介绍如何开发和调用服务器端的 PL/SQL 子程序，在学习了本章之后，读者应该学会：

- 建立和调用过程；
- 建立和调用函数；
- 管理 PL/SQL 子程序。

## 11.1 开发过程

过程用于执行特定操作。如果在应用程序中经常需要执行特定的操作，可以基于这些操作建立一个特定的过程。通过使用过程，不仅可以简化客户端应用程序的开发和维护，而且还可以提高应用程序的运行性能。建立过程的语法如下所示：

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  (argument1 [mode1] datatype1, argument2 [mode2] datatype2,...)
  IS [AS]
    PL/SQL Block;
```

如上所示，`procedure_name` 用于指定过程名称；`argument1`, `argument2` 等则用于指定过程的参数；`IS` 或 `AS` 用于开始一个 PL/SQL 块。注意，当指定参数数据类型时，不能指定其长度。另外，当建立过程时，既可以指定输入参数（IN），也可以指定输出参数（OUT）及输入输出参数（IN OUT）。通过在过程中使用输入参数，可以将应用环境的数据传递到执行部分；通过使用输出参数，可以将执行部分的数据传递到应用环境。定义子程序参数时，如果不指定参数模式，则默认为输入参数；如果要定义输出参数，那么需要指定 `OUT` 关键字；如果要定义输入输出参数，则需要指定 `IN OUT` 关键字。下面通过示例说明建立过程和使用各种参数模式的方法。

### 1. 建立过程：不带任何参数

建立过程时，过程既可以带有参数，也可以不带任何参数。下面以建立用于输出当前系统日期和时间的过程为例，说明建立该种过程的方法。示例如下：

```
CREATE OR REPLACE PROCEDURE out_time
IS
BEGIN
  dbms_output.put_line(systimestamp);
```

```
END;
/

```

建立了过程 out\_time 之后，就可以调用该过程了。在 SQL\*Plus 环境中调用过程有两种方法，一种是使用 execute（简写为 exec）命令，另一种是使用 call 命令。

#### 示例一：使用 execute 命令调用过程

```
SQL> set serveroutput on
SQL> exec out_time
11-12月-03 02.40.29.886000000 下午 +08:00
```

#### 示例二：使用 call 命令调用过程

```
SQL> set serveroutput on
SQL> call out_time();
31-12月-03 08.26.12.425000000 下午 +08:00
```

### 2. 建立过程：带有 IN 参数

建立过程时，可以通过使用输入参数，将应用程序的数据传递到过程中。当为过程定义参数时，如果不指定参数模式，那么默认就是输入参数，另外也可以使用 IN 关键字显式地定义输入参数。下面以建立为雇员插入数据的过程 ADD\_EMPLOYEE 为例，说明建立带有输入参数的过程的方法。示例如下：

```
CREATE OR REPLACE PROCEDURE add_employee
(eno NUMBER, name VARCHAR2, sal NUMBER,
 job VARCHAR2 DEFAULT 'CLERK', dno NUMBER)
IS
    e_integrity EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_integrity,-2291);
BEGIN
    INSERT INTO emp(empno,ename,sal,job,deptno)
        VALUES(eno,name,sal,job,dno);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        RAISE_APPLICATION_ERROR(-20000,'雇员号不能重复');
    WHEN e_integrity THEN
        RAISE_APPLICATION_ERROR(-20001,'部门号不存在');
END;
/
```

如上所示，因为在建立过程 ADD\_EMPLOYEE 时所有参数都没有指定参数模式，所以这些参数全部都是输入参数。当调用该过程时，除了具有默认值的参数之外，其他参数必须要提供数值。调用示例如下：

#### 示例一：数据满足约束规则

```
SQL> exec add_employee(1111,'MARY',2000,'MANAGER',10)
```

#### 示例二：输入重复的雇员号

```
SQL> exec add_employee(1111,'CLARK',2000,'MANAGER',10)
BEGIN add_employee(1111,'CLARK',2000,'MANAGER',10); END;
*
```

ERROR 位于第 1 行：

ORA-20000: 雇员号不能重复

ORA-06512: 在"SCOTT.ADD\_EMPLOYEE", line 12

ORA-06512: 在 line 1

### 示例三：输入不存在的部门号

```
SQL> exec add_employee(1112,'CLARK',2000,'MANAGER',15)
BEGIN add_employee(1112,'CLARK',2000,'MANAGER',15); END;
*
ERROR 位于第 1 行:
ORA-20001: 部门号不存在
ORA-06512: 在"SCOTT.ADD_EMPLOYEE", line 14
ORA-06512: 在 line 1
```

### 3. 建立过程：带有 OUT 参数

过程不仅可以用于执行特定操作，而且也可以用于输出数据，在过程中输出数据是使用 OUT 或 IN OUT 参数来完成的。当定义输出参数时，必须要提供 OUT 关键字。下面以建立用于输出雇员名及其工资的过程为例，说明建立带有 OUT 参数的过程的方法。示例如下：

```
CREATE OR REPLACE PROCEDURE query_employee
(eno NUMBER, name OUT VARCHAR2, salary OUT NUMBER)
IS
BEGIN
    SELECT ename,sal INTO name,salary FROM emp
    WHERE empno=eno;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20000,'该雇员不存在');
END;
/
```

如上所示，因在建立过程 query\_employee 时，没有为参数 eno 没有指定参数模式，所以该参数是输入参数；因为参数 name 和 salary 指定了 OUT 关键字，所以这两个参数是输出参数。当在应用程序中调用该过程时，必须要定义变量接收输出参数的数据。下面是在 SQL\*Plus 中调用该过程的示例：

```
SQL> var name VARCHAR2(10)
SQL> var salary NUMBER
SQL> exec query_employee(7788,:name,:salary)
PL/SQL 过程已成功完成。
SQL> PRINT name salary
NAME
-----
SCOTT
SALARY
-----
3000
SQL> exec query_employee(7799,:name,:salary)
BEGIN query_employee(7799,:name,:salary); END;
*
ERROR 位于第 1 行:
ORA-20000: 该雇员不存在
ORA-06512: 在"SCOTT.QUERY_EMPLOYEE", line 9
```

ORA-06512: 在 line 1

#### 4. 建立过程: 带有 IN OUT 参数

定义过程时, 不仅可以指定 IN 和 OUT 参数, 也可以指定 IN OUT 参数。IN OUT 参数也称为输入输出参数, 当使用这种参数时, 在调用过程之前需要通过变量给该种参数传递数据, 在调用结束之后, Oracle 会通过该变量将过程结果传递给应用程序。下面以计算两个数值相除结果的过程 compute 为例, 说明在过程中使用 IN OUT 参数的方法。示例如下:

```
CREATE OR REPLACE PROCEDURE compute
(num1 IN OUT NUMBER, num2 IN OUT NUMBER)
IS
    v1 NUMBER;
    v2 NUMBER;
BEGIN
    v1:=num1/num2;
    v2:=MOD(num1,num2);
    num1:=v1;
    num2:=v2;
END;
/
```

如上所示, 在过程 compute 中, num1, num2 为输入输出参数。当在应用程序中调用该过程时, 必须要提供两个变量临时存放数值, 在运算结束之后会将两数相除的商和余数分别存放到这两个变量中。下面是在 SQL\*Plus 中调用该过程的示例:

```
SQL> var n1 NUMBER
SQL> var n2 NUMBER
SQL> exec :n1:=100
SQL> exec :n2:=30
SQL> exec compute(:n1,:n2)
SQL> PRINT n1 n2
      N1
-----
      3.33333333
      N2
-----
      10
```

#### 5. 为参数传递变量和数据

当调用带有参数的子程序时, 需要将数值或变量传递给参数。为参数传递变量和数据可以采用位置传递、名称传递和组合传递等三种方法。注意, 如果在定义参数时带有默认值, 那么在调用子程序时可以不给该参数提供数值。在介绍如何使用各种方法为参数传递数值之前, 应首先建立用于为 DEPT 表增加数据的过程。示例如下:

```
CREATE OR REPLACE PROCEDURE add_dept
(dno NUMBER, dname VARCHAR2 DEFAULT NULL,
 loc VARCHAR2 DEFAULT NULL)
IS
BEGIN
    INSERT INTO dept VALUES(dno,dname,loc);
```

```

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    RAISE_APPLICATION_ERROR(-20000,'部门号不能重复');
END;
/

```

### 示例一：位置传递

位置传递是指在调用子程序时按照参数定义的顺序依次为参数指定相应变量或者数值。示例如下：

```

SQL> exec add_dept(50,'SALES','NEW YORK');
SQL> exec add_dept(60);
SQL> exec add_dept(70,'ADMIN');

```

如上所示，第二次调用时为参数 dno 提供了数据，但没有给参数 dname 和 loc 提供数据；第三次调用为参数 dno 和 dname 提供了数据，但没有给参数 loc 提供数据。原因是参数 dname 和 loc 都具有默认值 (NULL)。

### 示例二：名称传递

名称传递是指在调用子程序时指定参数名，并使用关联符号 “=>” 为其提供相应的数值或变量。示例如下：

```

SQL> exec add_dept(dname=>'SALES',dno=>50);
SQL> exec add_dept(dno=>60);

```

### 示例三：组合传递

组合传递是指在调用子程序时同时使用位置传递和名称传递。示例如下：

```

SQL> exec add_dept(50,loc=>'NEW YORK');
SQL> exec add_dept(60,dname=>'SALES',loc=>'NEW YORK');

```

## 6. 查看过程源代码

当建立了过程之后，Oracle 会将过程名、源代码及其执行代码存放到数据字典中。当调用过程时，应用程序会按照其执行代码直接执行，而不需要重新解析过程代码，所以使用子程序的性能要优于直接执行 SQL 语句。通过查询数据字典 USER\_SOURCE，可以显示当前用户的所有子程序及其源代码。示例如下：

```

SQL> SELECT text FROM user_source WHERE name='ADD_DEPT';
TEXT
-----
PROCEDURE add_dept
(dno NUMBER,dname VARCHAR2 DEFAULT NULL,
loc VARCHAR2 DEFAULT NULL)
IS
BEGIN
  INSERT INTO dept VALUES(dno,dname,loc);
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    RAISE_APPLICATION_ERROR(-20000,'部门号不能重复');
END;

```

## 7. 删除过程

当过程不再需要时，用户可以使用 DROP PROCEDURE 命令来删除该过程。示例如下：

```
SQL> DROP PROCEDURE add_dept;
```

## 11.2 开发函数

函数用于返回特定数据。如果在应用程序中经常需要通过执行 SQL 语句来返回特定数据，那么可以基于这些操作建立特定的函数。通过使用函数，不仅可以简化客户端应用程序的开发和维护，而且还可以提高应用程序的执行性能。建立函数的语法如下所示：

```
CREATE [OR REPLACE] FUNCTION function_name
    (argument1 [mode1] datatype1,
     argument2 [mode2] datatype2,
     . . .)
RETURN datatype
IS|AS
    PL/SQL Block;
```

如上所示，`function_name` 用于指定函数名称；`argument1`、`argument2` 等则用于指定函数的参数，注意，当指定参数数据类型时，不能指定其长度；`RETURN` 子句用于指定函数返回值的数据类型；`IS` 或 `AS` 用于开始一个 PL/SQL 块。注意，当建立函数时，在函数头部必须要带有 `RETURN` 子句，在函数体内至少要包含一条 `RETURN` 语句。另外，当建立函数时，既可以指定输入参数（IN），也可以指定输出参数（OUT）及输入输出参数（IN OUT）。下面通过示例说明建立函数，以及使用各种参数模式的方法。

### 1. 建立函数：不带任何参数

当建立函数时，函数既可以带有参数，也可以不带参数。下面以建立用于显示当前数据库用户名的函数为例，说明建立该种函数的方法。示例如下：

```
CREATE OR REPLACE FUNCTION get_user
RETURN VARCHAR2
IS
    v_user VARCHAR2(100);
BEGIN
    SELECT username INTO v_user FROM user_users;
    RETURN v_user;
END;
/
```

建立了函数 `get_user` 之后，就可以在应用程序中调用该函数了。因为函数有返回值，所以它只能作为表达式的一部分来调用。调用示例如下：

#### 示例一：使用变量接收函数返回值

```
SQL> var v1 VARCHAR2(100)
SQL> exec :v1:=get_user
SQL> PRINT v1
V1
-----
SCOTT
```

#### 示例二：在 SQL 语句中直接调用函数

```
SQL> SELECT get_user FROM dual;
GET_USER
```

```
-----  
SCOTT
```

### 示例三：使用包 DBMS\_OUTPUT 调用函数

```
SQL> set serveroutput on
SQL> exec dbms_output.put_line('当前数据库用户: '||get_user)
当前数据库用户: SCOTT
```

### 2. 建立函数：带有 IN 参数

当建立函数时，通过使用输入参数，可以将应用程序的数据传递到函数中，最终通过执行函数可以将结果返回到应用程序中。当定义参数时，如果不指定参数模式，则默认为输入参数，所以 IN 关键字既可以指定，也可以不指定。下面以建立用于返回雇员工资的函数为例，说明建立带有输入参数函数的方法。示例如下：

```
CREATE OR REPLACE FUNCTION get_sal(name IN VARCHAR2)
RETURN NUMBER
AS
  v_sal emp.sal%TYPE;
BEGIN
  SELECT sal INTO v_sal FROM emp
  WHERE upper(ename)=upper(name);
  RETURN v_sal;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    raise_application_error(-20000,'该雇员不存在');
END;
/
```

在建立了函数 get\_sal 之后，就可以在应用程序中调用该函数了。调用该函数的示例如下：

### 示例一：输入存在的雇员：

```
SQL> var sal NUMBER
SQL> exec :sal:=get_sal('scott')
SQL> print sal
      SAL
-----
      3000
```

### 示例二：输入不存在的雇员

```
SQL> exec :sal:=get_sal('mary')
BEGIN :sal:=get_sal('mary'); END;
*
ERROR 位于第 1 行:
ORA-20000: 该雇员不存在
ORA-06512: 在"SCOTT.GET_SAL", line 12
ORA-06512: 在 line 1
```

### 3. 建立函数：带有 OUT 参数

一般情况下，函数只需要返回单个数据。如果希望使用函数同时返回多个数据，例如同时返回雇员名和工资，那么就需要使用输出参数了。为了在函数中使用输出参数，必须要指定 OUT 参数模式。下面以建立用于返回雇员所在部门名和岗位的函数为例，说明建立带有

OUT 参数函数的方法。示例如下：

```

CREATE OR REPLACE FUNCTION get_info
(name VARCHAR2,title OUT VARCHAR2)
RETURN VARCHAR2
AS
    deptname dept.dname%TYPE;
BEGIN
    SELECT a.job,b.dname INTO title,deptname
    FROM emp a,dept b
    WHERE a.deptno=b.deptno
    AND upper(a.ename)=upper(name);
    RETURN deptname;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        raise_application_error(-20000,'该雇员不存在');
END;
/

```

在建立了函数 get\_info 之后，就可以在应用程序中调用该函数了。注意，因为该函数带有 OUT 参数，所以不能在 SQL 语句中调用该函数，而必须要定义变量接收 OUT 参数和函数的返回值。在 SQL\*Plus 中调用函数 get\_info 的示例如下：

```

SQL> var job varchar2(20)
SQL> var dname varchar2(20)
SQL> exec :dname:=get_info('scott',:job)
SQL> print dname job
DNAME
-----
RESEARCH
JOB
-----
ANALYST

```

#### 4. 建立函数：带有 IN OUT 参数

建立函数时，不仅可以指定 IN 和 OUT 参数，也可以指定 IN OUT 参数。IN OUT 参数也被为输入输出参数。使用这种参数时，在调用函数之前需要通过变量给该种参数传递数据，在调用结束之后 Oracle 会将函数的部分结果通过该变量传递给应用程序。下面以计算两个数值相除的结果的函数 result 为例，说明在函数中使用 IN OUT 参数的方法。示例如下：

```

CREATE OR REPLACE FUNCTION result
(num1 NUMBER,num2 IN OUT NUMBER)
RETURN NUMBER
AS
    v_result NUMBER(6);
    v_remainder NUMBER;
BEGIN
    v_result:=num1/num2;
    v_remainder:=MOD(num1,num2);
    num2:=v_remainder;

```

```

    RETURN v_result;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        raise_application_error(-20000,'不能除 0');
END;
/

```

注意，因为该函数带有 IN OUT 参数，所以不能在 SQL 语句中调用该函数，而必须使用变量为 IN OUT 参数传递数值并接收数据，另外还需要定义变量接收函数返回值。调用该函数的示例如下：

```

SQL> var result1 NUMBER
SQL> var result2 NUMBER
SQL> exec :result2:=30
SQL> exec :result1:=result(100,:result2)
SQL> print result1 result2
      RESULT1
-----
      3
      RESULT2
-----
      10

```

## 5. 函数调用限制

因为函数必须要返回数据，所以只能作为表达式的一部分调用。另外，函数也可以在 SQL 语句的以下部分调用：

- SELECT 命令的选择列表；
- WHERE 和 HAVING 子句中；
- CONNECT BY, START WITH, ORDER BY 以及 GROUP BY 子句中；
- INSERT 命令的 VALUES 子句中；
- UPDATE 命令的 SET 子句中。

注意，并不是所有函数都可以在 SQL 语句中调用，在 SQL 语句中调用函数有一些限制：

- 在 SQL 语句中只能调用存储函数（服务器端），而不能调用客户端的函数；
- 在 SQL 语句中调用的函数只能带有输入参数（IN），而不能带有输出参数（OUT）和输入输出参数（IN OUT）；
- 在 SQL 语句中调用的函数只能使用 SQL 所支持的标准数据类型，而不能使用 PL/SQL 的特有数据类型（例如 BOOLEAN, TABLE 和 RECORD 等）；
- 在 SQL 语句中调用的函数不能包含 INSERT, UPDATE 和 DELETE 语句。

## 6. 查看函数源代码

当建立了函数之后，Oracle 会将函数名及其源代码信息存放到数据字典中。通过查询数据字典 USER\_SOURCE，可以显示当前用户的所有子程序及其源代码。示例如下：

```

SQL> set pagesize 40
SQL> SELECT text FROM user_source WHERE name='RESULT';
TEXT

```

```

-----
FUNCTION result
(num1 NUMBER, num2 IN OUT NUMBER)
RETURN NUMBER
AS
  v_result NUMBER(6);
  v_remainder NUMBER;
BEGIN
  v_result:=num1/num2;
  v_remainder:=MOD(num1,num2);
  num2:=v_remainder;
  RETURN v_result;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    raise_application_error(-20000,'不能除 0');
END;

```

## 7. 删除函数

当某函数不再需要时，你可以使用 DROP FUNCTION 命令删除该函数。示例如下：

```
SQL> DROP FUNCTION result;
```

## 11.3 管理子程序

### 1. 列出当前用户的子程序

数据字典视图 USER\_OBJECTS 用于显示当前用户所包含的所有对象。它不仅可以用于列出用户的表、视图、索引等，也可以用于列出用户的过程、函数和包。列出当前用户所包含的过程和函数的示例如下：

```

SQL> col object_name format a20
SQL> SELECT object_name,created,status FROM user_objects
  2 WHERE object_type IN ('PROCEDURE','FUNCTION');
OBJECT_NAME          CREATED      STATUS
-----              -----
ADD_EMPLOYEE        11-12月-03  VALID
COMPUTE             11-12月-03  VALID
GET_INFO            15-12月-03  VALID
GET_SAL             16-11月-03  VALID
GET_USER            15-12月-03  VALID
OUT_TIME            11-12月-03  VALID
QUERY_EMPLOYEE     11-12月-03  VALID
RAISE_COMM          10-12月-03  VALID
RAISE_SALARY        16-11月-03  INVALID

```

如上所示，object\_name 用于标识对象名称；object\_type 用于标识对象类型；created 用于标识对象建立时间；status 用于标识对象当前状态，VALID 表示对象有效，而 INVALID 则表示对象无效（不能引用）。

## 2. 列出子程序源代码

数据字典视图 USER\_SOURCE 用于列出子程序的源代码，示例如下：

```
SQL> SELECT text FROM user_source WHERE name='RAISE_SALARY';
TEXT
-----
PROCEDURE raise_salary
  (no number,increase number) IS
BEGIN
  UPDATE emp SET sal=sal+increase WHERE empno=no;
END;
```

## 3. 列出子程序编译错误

当编写子程序时，如果对象编译成功，则会显示消息“过程（函数）已建立”；如果对象编译不成功，则会显示“警告：创建的过程（函数）带有编译错误”。那么，如何确定错误出现在哪行，以及错误原因呢？用户可以使用两种方法，第一种方法是使用 SHOW ERRORS 命令，第二种方法是使用数据字典视图 USER\_ERRORS。请看如下错误示例：

```
CREATE OR REPLACE PROCEDURE raise_salary
  (no number,increase number) IS
BEGIN
  UPDATE emp SET sal=sal+increase WHERE empno=no
END;
/
警告：创建的过程带有编译错误。
```

### (1) 使用 SHOW ERRORS 命令确定错误原因和位置

```
SQL> SHOW ERRORS PROCEDURE raise_salary
PROCEDURE RAISE_SALARY 出现错误：
LINE/COL ERROR
-----
4/4      PL/SQL: SQL Statement ignored
5/2      PL/SQL: ORA-00933: SQL 命令未正确结束
5/5      PLS-00103: 出现符号 "end-of-file" 在需要下列之一时:
          begin case declare
          end exception exit for goto if loop mod null pragma raise
          return select update while with <an identifier>
          <a double-quoted delimited-identifier> <a bind variable> <<
          close current delete fetch lock insert open rollback
          savepoint set sql execute commit forall merge
          <a single-quoted SQL string> pipe
```

### (2) 使用数据字典视图 USER\_ERRORS 确定错误原因和位置

```
SQL> col text format a50
SQL> SELECT line||'/'||position AS "LINE/COL",text error
  2  FROM user_errors WHERE name='RAISE_SALARY';
LINE/COL ERROR
-----
4/4      PL/SQL: SQL Statement ignored
5/2      PL/SQL: ORA-00933: SQL 命令未正确结束
```

5/5

PLS-00103: 出现符号 "end-of-file" 在需要下列之一时:  
begin case declare  
end exception exit for goto if loop mod null pragma raise  
return select update while with <an identifier>  
<a double-quoted delimited-identifier> <a bind variable> <<  
close current delete fetch lock insert open rollback  
savepoint set sql execute commit forall merge  
<a single-quoted SQL string> pipe

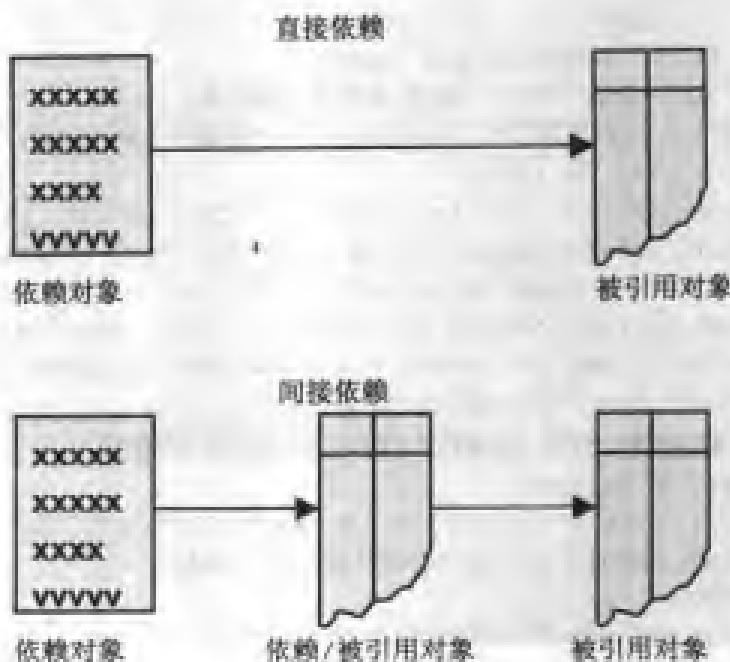
### (3) 重新建立对象 raise\_salary

根据错误提示信息，可以确定错误原因发生在第 4 行，并且可以确定错误原因是 SQL 语句结束时缺少分号。建立子程序的正确命令如下：

```
CREATE OR REPLACE PROCEDURE raise_salary
    (no number, increase number) IS
BEGIN
    UPDATE emp SET sal=sal+increase WHERE empno=no;
END;
/
过程已创建。
```

### 4. 列出对象依赖关系

建立存储对象（过程、函数、包、视图、触发器）时，它们往往需要引用其他对象。例如，当建立过程 raise\_salary 时，在 UPDATE 语句中引用了 EMP 表。出于该原因，将存储过程 raise\_salary 称为对象依赖（dependent object），而将表 EMP 称为被引用对象（referenced object）。对象依赖包括直接依赖和间接依赖两种情况，其中直接依赖是指存储对象直接依赖于被引用对象（例如过程 raise\_salary 和表 EMP 之间存在直接依赖关系），而间接依赖是指对象间接依赖于被引用对象。如下图所示：



注意，无论是直接依赖，还是间接依赖，当修改了被引用对象的结构时，都会使得相关依赖对象转变为 INVALID 状态。那么如何确定对象之间的依赖关系呢？确定对象依赖关系有两种方法，第一种方法是使用数据字典视图 USER\_DEPENDENCIES 确定直接依赖关系，另一种方法是使用工具视图 DEPTREE 和 IDEPTREE 确定直接依赖和间接依赖关系。

### (1) 使用 USER\_DEPENDENCIES 确定直接依赖关系

下面以显示直接依赖于表 EMP 的所有对象以及对象类型为例，说明使用数据字典视图 USER\_DEPENDENCIES 的方法。示例如下：

```
SQL> SELECT name,type FROM user_dependencies
  2 WHERE referenced_name='EMP';
NAME                      TYPE
-----
RAISE_SALARY                PROCEDURE
RAISE_COMM                  PROCEDURE
DEPT10                     VIEW
ADD_EMPLOYEE                PROCEDURE
QUERY_EMPLOYEE               PROCEDURE
GET_SAL                      FUNCTION
GET_INFO                     FUNCTION
```

### (2) 使用工具视图 DEPTREE 和 IDEPTREE 确定直接依赖和间接依赖关系

当使用工具视图 DEPTREE 和 IDEPTREE 确定依赖关系时，必须首先运行 SQL 脚本 utldtree.sql 来建立这两个视图和过程 DEPTREE\_FILL，然后调用过程 DEPTREE\_FILL 填充这两个视图。示例如下：

```
SQL> @%oracle_home%\rdbms\admin\utldtree
SQL> exec deptree_fill('TABLE','SCOTT','EMP')
```

如上所示，在执行了过程 deptree\_fill 之后，会将直接或间接依赖于 SCOTT.EMP 表的所有对象填充到视图 DEPTREE 和 IDEPTREE 中，通过查询它们可以非常直观地显示直接或间接依赖于 SCOTT.EMP 表的对象。示例如下：

```
SQL> SELECT nested_level,name,type FROM deptree;
NESTED_LEVEL    NAME                      TYPE
-----
1              RAISE_SALARY                PROCEDURE
1              GET_SAL                   FUNCTION
0              EMP                      TABLE
1              RAISE_COMM                PROCEDURE
1              ADD_EMPLOYEE               PROCEDURE
1              QUERY_EMPLOYEE            PROCEDURE
1              GET_INFO                 FUNCTION
1              DEPT10                  VIEW
SQL> SELECT * FROM ideptree;
DEPENDENCIES
-----
TABLE SCOTT.EMP
PROCEDURE SCOTT.RAISE_SALARY
PROCEDURE SCOTT.RAISE_COMM
```

```

VIEW SCOTT.DEPT10
PROCEDURE SCOTT.ADD_EMPLOYEE
PROCEDURE SCOTT.QUERY_EMPLOYEE
FUNCTION SCOTT.GET_SAL
FUNCTION SCOTT.GET_INFO

```

### 5. 重新编译子程序

当修改了被引用对象的结构时，就会将相关依赖对象转变为无效（INVALID）状态。例如，当为 EMP 表增加了列之后，会使所有依赖对象转变为 INVALID 状态。示例如下：

```

SQL> ALTER TABLE emp ADD remark VARCHAR2(100);
SQL> SELECT object_name,object_type FROM user_objects
  2 WHERE status='INVALID';
OBJECT_NAME          OBJECT_TYPE
-----
ADD_EMPLOYEE        PROCEDURE
DEPT10              VIEW
GET_INFO             FUNCTION
GET_SAL              FUNCTION
PROCEDURE            PROCEDURE
QUERY_EMPLOYEE      PROCEDURE
RAISE_COMM          PROCEDURE
RAISE_SALARY         PROCEDURE

```

当对象状态为 INVALID 时，为了避免子程序的运行错误，应该重新编译这些存储对象。示例如下：

```

SQL> ALTER PROCEDURE add_employee COMPILE;
SQL> ALTER VIEW dept10 COMPILE;
SQL> ALTER FUNCTION get_info COMPILE;

```

## 11.4 习题

1. 以下哪种程序单元必须返回数据？
  - A. 触发器
  - B. 函数
  - C. 过程
  - D. 包
2. 当建立过程时，使用以下哪些参数可以输出数据？
  - A. IN 参数
  - B. OUT 参数
  - C. IN OUT 参数
  - D. 任何参数都不能输出数据
3. 建立函数 valid\_cust，根据输入的客户号，检查客户是否存在。如果客户存在，则返回 TRUE，否则返回 FALSE。
4. 建立过程 ADD\_ORD，根据输入的订单号、预订日期、客户号、交付日期和订单总价，为 ORD 表插入数据，然后调用该过程。当建立过程 ADD\_ORD 时，实现以下商业规则：
  - 使用函数 valid\_cust 检查客户号是否正确；如果正确，则插入数据，否则显示自定义错误消息“ORA-20001：检查并输入正确的客户号”；
  - 如果交付日期小于预订日期，则显示自定义错误消息“ORA-20002：交付日期必须在预订日期之后”；

- 如果输入了已经存在的订单号，则显示自定义错误消息“ORA-20003：该订单已经存在”。

5. 建立过程 UPD\_SHIPDATE，根据输入的订单号和交付日期，更新 ORD 表特定订单的交付日期，然后调用该过程。当建立过程 UPD\_SHIPDATE 时，实现以下商业规则：

- 如果交付日期小于预订日期，则显示自定义错误消息“ORA-20001：交付日期必须在预订日期之后”；
- 如果订单不存在，则显示自定义错误消息“ORA-20002：请检查并输入正确的订单号”。

6. 建立函数 GET\_TOTAL，根据输入的订单号返回订单总价，然后调用该函数。当建立函数 GET\_TOTAL 时，实现以下商业规则：

- 如果订单不存在，则显示自定义错误消息“ORA-20001：请检查并输入正确的订单号”。

7. 建立过程 DELETE\_ORD，根据输入的订单号取消特定订单，然后调用该过程。当建立过程 DELETE\_ORD 时，实现以下商业规则：

- 如果订单不存在，则显示自定义错误消息“ORA-20001：请检查并输入正确的订单号”。

## 第 12 章 开发包

包（Package）用于逻辑组合相关的 PL/SQL 类型（例如 TABLE 类型和 RECORD 类型）、PL/SQL 项（例如游标和游标变量）和 PL/SQL 子程序（例如过程和函数）。通过使用 PL/SQL 包，不仅简化了应用设计，提高了应用性能，而且还可以实现信息隐藏、子程序重载等功能。本章将详细介绍开发 PL/SQL 包的方法，在学习了本章之后，读者应该能学会：

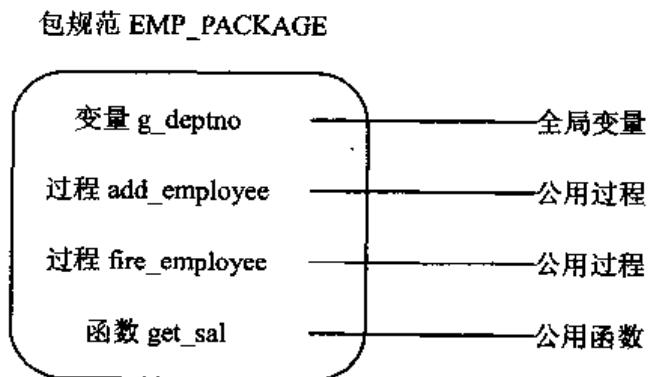
- 建立包规范和包体；
- 在包内定义公用组件和私有组件；
- 使用重载特征；
- 建立构造过程；
- 使用纯度级别。

### 12.1 建立包

包用于逻辑组合相关的 PL/SQL 类型、项和子程序，它由包规范（Package Specification）和包体（Package Body）两部分组成。当建立包时，需要首先建立包规范，然后再建立包体。

#### 1. 建立包规范

包规范实际是包与应用程序之间的接口，它用于定义包的公用组件，包括常量、变量、游标、过程和函数等。在包规范中所定义的公用组件不仅可以在包内引用，而且也可以由其他的子程序引用。假定在定义包规范 EMP\_PACKAGE 时，定义了公用变量 g\_deptno、公用过程 add\_employee 和 fire\_employee，以及公用函数 get\_sal，那么它们不仅可以在包 EMP\_PACKAGE 内引用，而且也可以由其他子程序引用。如下图所示：



建立包规范时，需要注意，为了实现信息隐藏，不应该将所有组件全部放在包规范处定义，而应该只定义公用组件。在 SQL\*Plus 中建立包规范是使用 CREATE PACKAGE 命令来完成的，语法如下：

```
CREATE [OR REPLACE] PACKAGE package_name
```

```

IS | AS
public type and item declarations
subprogram specificationsEND package_name;

```

如上所示，`package_name` 用于指定包名，而以 IS 或 AS 开始的部分用于定义公用组件。下面以建立用于维护 EMP 表的包 `EMP_PACKAGE` 为例，说明建立包规范的方法。当定义该包规范时，需要定义公用变量 `g_deptno`、公用过程 `add_employee` 和 `fire_employee`，以及公用函数 `get_sal`。示例如下：

```

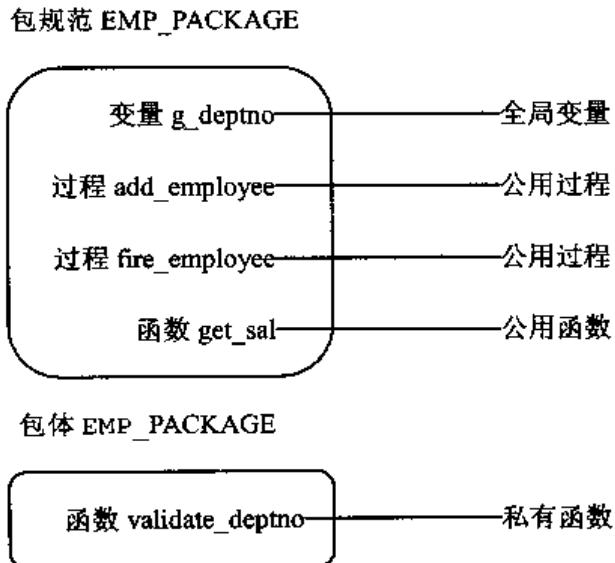
CREATE OR REPLACE PACKAGE emp_package IS
    g_deptno NUMBER(3):=30;
    PROCEDURE add_employee(eno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER DEFAULT g_deptno);
    PROCEDURE fire_employee(eno NUMBER);
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER;
END emp_package;
/

```

当执行了以上命令之后，会建立包规范 `emp_package`，并且定义了所有公用组件。但因为只定义了过程和函数的头部，没有编写过程和函数的执行代码，所以公用的过程和函数只有在建立了包体之后才能调用。

## 2. 建立包体

包体用于实现包规范所定义的过程和函数。当建立包体时，读者也可以单独定义私有组件，包括变量、常量、过程和函数等，但在包体中所定义的私有组件只能在包内使用，而不能由其他子程序引用。假定在建立包体 `EMP_PACKAGE` 时定义了函数 `validate_deptno`，那么该函数只能在包 `EMP_PACKAGE` 内使用，而不能由其他子程序调用。如下图所示：



当建立包时，为了实现信息隐藏，应该在包体内定义私有组件；为了实现包规范中所定义的公用过程和函数，必须建立包体。建立包体是使用命令 `CREATE PACKAGE BODY` 来完成的。语法如下：

```
CREATE [OR REPLACE] PACKAGE BODY package_name
```

```

IS | AS
    private type and item declarations
    subprogram bodies
END package_name;

```

如上所示，`package_name` 用于指定包名，而用 `IS` 或 `AS` 开始的部分定义私有组件，并实现包规范中所定义的公用过程和函数。注意，包体名称与包规范名称必须相同。下面以实现包规范 `emp_package` 的公用组件，以及私有组件 `validate_deptno` 为例，说明建立包体的方法。示例如下：

```

CREATE OR REPLACE PACKAGE BODY emp_package IS
    FUNCTION validate_deptno(v_deptno NUMBER)
        RETURN BOOLEAN
    IS
        v_temp INT;
    BEGIN
        SELECT 1 INTO v_temp FROM dept WHERE deptno=v_deptno;
        RETURN TRUE;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END;
    PROCEDURE add_employee(eno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER DEFAULT g_deptno)
    IS
    BEGIN
        IF validate_deptno(dno) THEN
            INSERT INTO emp (empno,ename,sal,deptno)
            VALUES (eno,name,salary,dno);
        ELSE
            raise_application_error(-20010,'不存在该部门');
        END IF;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            raise_application_error(-20011,'该雇员已存在');
    END;
    PROCEDURE fire_employee(eno NUMBER) IS
    BEGIN
        DELETE FROM emp WHERE empno=eno;
        IF SQL%NOTFOUND THEN
            raise_application_error(-20012,'该雇员不存在');
        END IF;
    END;
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER
    IS
        v_sal emp.sal%TYPE;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno=eno;

```

```

        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            raise_application_error(-20012,'该雇员不存在');
    END;
END emp_package;
/

```

执行了以上命令之后，会建立包体 EMP\_PACKAGE，应用程序只能直接调用该包内的所有公用组件，而私有函数 VALIDATE\_DEPTNO 则不能被应用程序调用。

### 3. 调用包组件

对于包的私有组件，只能在包内调用，并且可以直接调用；而对于包的公用组件，既可以在包内调用，也可以在其他应用程序中调用。但需要注意，当在其他应用程序中调用包的组件时，必须要加包名作为前缀（包名.组件名）。下面举例说明调用包组件的方法。

#### 示例一：在同一个包内调用包组件

当调用同一包内的其他组件时，可以直接调用，不需要加包名作为前缀。下面以包 emp\_package 的过程 add\_employee 调用包私有组件 validate\_deptno 为例，说明调用同一个包内其他组件的方法。示例如下：

```

CREATE OR REPLACE PACKAGE BODY emp_package IS

    PROCEDURE add_employee(eno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER DEFAULT g_deptno)
    IS
    BEGIN
        IF validate_deptno(dno) THEN
            INSERT INTO emp (empno, ename, sal, deptno)
            VALUES (eno, name, salary, dno);
        ELSE
            raise_application_error(-20010, '不存在该部门');
        END IF;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            raise_application_error(-20011, '该雇员已存在');
    END;
    ...

```

#### 示例二：调用包公用变量

当在其他应用程序中调用包的公用变量时，必须要在公用变量名前加包名作为前缀，并且注意其数值在当前会话内一直生效。示例如下：

```
SQL> exec emp_package.g_deptno:=20
```

#### 示例三：调用包公用过程

当在其他应用程序中调用包的公用过程时，必须要在公用过程名前加包名作为前缀。示例如下：

```
SQL> exec emp_package.add_employee(1111, 'MARY', 2000)
SQL> exec emp_package.add_employee(1112, 'CLARK', 2000, 10)
```

当执行了以上两条命令之后，会为 EMP 表增加两条记录（部门号分别为 20 和 10）。而如果输入的部门号或雇员号（有误），则会显示错误信息。如下所示：

```
SQL> exec emp_package.add_employee(1113,'BLAKE',2000,50)
BEGIN emp_package.add_employee(1113,'BLAKE',2000,50); END;
*
ERROR 位于第 1 行:
ORA-20010: 不存在该部门
ORA-06512: 在"SCOTT.EMP_PACKAGE", line 22
ORA-06512: 在 line 1
SQL> exec emp_package.fire_employee(1113)
BEGIN emp_package.fire_employee(1113); END;
*
ERROR 位于第 1 行:
ORA-20012: 该雇员不存在
ORA-06512: 在"SCOTT.EMP_PACKAGE", line 32
ORA-06512: 在 line 1
```

#### 示例四：调用包公用函数

当在其他应用程序中调用包的公用函数时，需要在函数名之前加包名作为前缀。因为函数只能作为表达式的一部分来调用，所以应该定义变量接收函数的返回值。在 SQL\*Plus 中定义变量并输出数据的示例如下：

```
SQL> VAR salary NUMBER
SQL> exec :salary:=emp_package.get_sal(7788)
SQL> print salary
      SALARY
-----
      3000
```

#### 示例五：以其他用户身份调用包公用组件

当以其他用户身份调用包的公用组件时，必须在组件名前加用户名和包名作为前缀（用户名.包名.组件名）。示例如下：

```
SQL> conn system/manager
SQL> exec scott.emp_package.add_employee(1115,'SCOTT',1200)
SQL> exec scott.emp_package.fire_employee(1115)
```

#### 示例六：调用远程数据库包的公用组件

当调用远程数据库包的公用组件时，在组件名之前加包名作为前缀，在组件名之后需要带有数据库链名作为后缀（包名.组件名@数据库链名）。示例如下：

```
SQL> exec emp_package.add_employee@orasrv(1116,'SCOTT',1200)
```

#### 4. 查看包源代码

当建立了包之后，Oracle 会将包名及其源代码信息存放到数据字典中。通过查询数据字典 USER\_SOURCE，可以显示当前用户的包及其源代码。示例如下：

```
SQL> SELECT text FROM user_source
  2 WHERE name='EMP_PACKAGE' AND type='PACKAGE';
TEXT
-----
-----
```

```

PACKAGE emp_package IS
    g_deptno NUMBER(3) :=30;
    PROCEDURE add_employee(eno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER DEFAULT g_deptno);
    PROCEDURE fire_employee(eno NUMBER);
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER;
END emp_package;

```

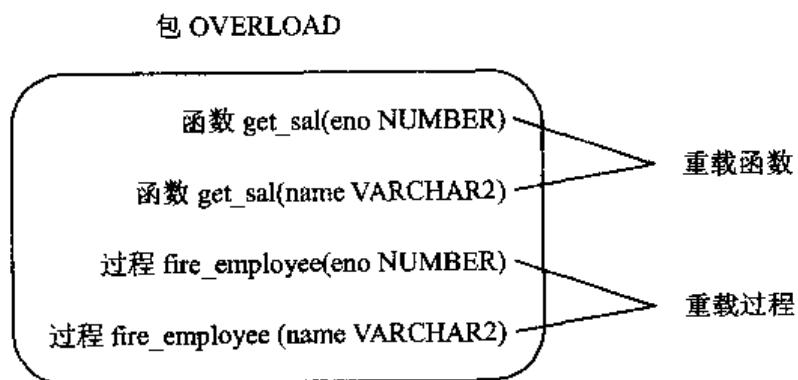
### 5. 删除包

当包不再需要时，可以删除包。如果只删除包体，那么可以使用命令 DROP PACKAGE BODY；如果同时删除包规范和包体，那么可以使用命令 DROP PACKAGE。示例如下：

```
SQL> DROP PACKAGE emp_package;
```

## 12.2 使用包重载

重载(overload)是指多个具有相同名称的子程序。定义包时，使用重载特性，可以使用户在调用同名组件时使用不同参数传递数据，从而方便用户使用。例如，当取得雇员工资或解雇雇员时，可能希望既可以输入雇员号，也可以输入雇员名，此时就需要使用包的重载特征。如下图所示：



### 1. 建立包规范

使用重载特性时，同名的过程和函数必须具有不同的输入参数。但需注意，同名函数返回值的数据类型必须完全相同。下面以建立使用雇员号和雇员名取得雇员工资、解雇雇员的包规范为例，说明定义重载过程和重载函数的方法。示例如下：

```

CREATE OR REPLACE PACKAGE overload IS
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER;
    FUNCTION get_sal(name VARCHAR2) RETURN NUMBER;
    PROCEDURE fire_employee(eno NUMBER);
    PROCEDURE fire_employee(name VARCHAR2);
END;
/

```

### 2. 建立包体

当建立包体时，必须要给不同的重载过程和重载函数提供不同的实现代码。下面以建立

包体 overload 为例，说明实现重载过程和重载函数的方法。示例如下：

```

CREATE OR REPLACE PACKAGE BODY overload IS
    FUNCTION get_sal(eno NUMBER) RETURN NUMBER
    IS
        v_sal emp.sal%TYPE;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno=eno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            raise_application_error(-20020,'该雇员不存在');
    END;
    FUNCTION get_sal(name VARCHAR2) RETURN NUMBER
    IS
        v_sal emp.sal%TYPE;
    BEGIN
        SELECT sal INTO v_sal FROM emp
        WHERE upper(ename)=upper(name);
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            raise_application_error(-20020,'该雇员不存在');
    END;
    PROCEDURE fire_employee(eno NUMBER) IS
    BEGIN
        DELETE FROM emp WHERE empno=eno;
        IF SQL%NOTFOUND THEN
            raise_application_error(-20020,'该雇员不存在');
        END IF;
    END;
    PROCEDURE fire_employee(name VARCHAR2) IS
    BEGIN
        DELETE FROM emp WHERE upper(ename)=upper(name);
        IF SQL%NOTFOUND THEN
            raise_application_error(-20020,'该雇员不存在');
        END IF;
    END;
END;
/

```

### 3. 调用重载过程和重载函数

建立了包规范和包体之后，就可以调用包的公用组件了。在调用重载过程和重载函数时，PL/SQL 执行器会自动根据输入参数值的数据类型确定要调用的过程和函数。示例如下：

```

SQL> VAR sal1 NUMBER
SQL> VAR sal2 NUMBER
SQL> exec :sal1:=overload.get_sal('scott')
SQL> exec :sal2:=overload.get_sal(7369)

```

```

SQL> PRINT sal1 sal2
      SAL1
-----
      3000
      SAL2
-----
      800
SQL> exec overload.fire_employee(7369)
SQL> exec overload.fire_employee('scott')

```

### 12.3 使用包构造过程

在包中定义了全局变量之后，有些情况下，会话中可能还需要初始化全局变量，此时可以使用包的构造过程，这种过程类似于高级语言中的构造函数（例如 C++ 语言）和构造方法（例如 Java 语言）。当在会话内第一次调用包的公用组件时，会自动执行其构造过程，并且该构造过程在同一会话内只会执行一次。下面以限制新老员工工资不能低于雇员的最低工资，并且不能超过雇员的最高工资为例，说明使用包构造过程的方法。

#### 1. 建立包规范

因为包的构造过程用于初始化包的全局变量，所以在定义包规范时应该定义相关的全局变量。因为要限制员工工资在最低工资和最高工资之间，所以应该定义两个全局变量 `minsal` 和 `maxsal` 分别存放雇员的最低工资和最高工资。建立包规范的示例如下：

```

CREATE OR REPLACE PACKAGE emp_package IS
    minsal NUMBER(6,2);
    maxsal NUMBER(6,2);
    PROCEDURE add_employee(eno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER);
    PROCEDURE upd_sal(eno NUMBER, salary NUMBER);
    PROCEDURE upd_sal(name VARCHAR2, salary NUMBER);
END;
/

```

执行了以上 PL/SQL 块之后，会建立包规范 `emp_package`，并定义两个全局变量 `minsal`、`maxsal` 和三个公用过程 `add_employee`、`upd_sal`（重载过程）。

#### 2. 建立包体

为了在运行包组件时将雇员的最低工资和最高工资分别赋值给全局变量 `minsal` 和 `maxsal`，需要在包体内编写构造过程。包的构造过程没有任何名称，它是在实现了包的其他过程之后，以 `BEGIN` 开始、以 `END` 结束的部分。示例如下：

```

CREATE OR REPLACE PACKAGE BODY emp_package IS
    PROCEDURE add_employee(cno NUMBER, name VARCHAR2,
                           salary NUMBER, dno NUMBER)
    IS
    BEGIN
        IF salary BETWEEN minsal AND maxsal THEN
            INSERT INTO emp (empno, ename, sal, deptno)

```

```

        VALUES(eno, name, salary, dno);
    ELSE
        raise_application_error(-20001, '工资不在范围内');
    END IF;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        raise_application_error(-20002, '该雇员已经存在');
END;
PROCEDURE upd_sal(eno NUMBER, salary NUMBER) IS
BEGIN
    IF salary BETWEEN minsal AND maxsal THEN
        UPDATE emp SET sal=salary WHERE empno=eno;
        IF SQL%NOTFOUND THEN
            raise_application_error(-20003, '不存在该雇员号');
        END IF;
    ELSE
        raise_application_error(-20001, '工资不在范围内');
    END IF;
END;
PROCEDURE upd_sal(name VARCHAR2, salary NUMBER) IS
BEGIN
    IF salary BETWEEN minsal AND maxsal THEN
        UPDATE emp SET sal=salary
        WHERE upper(ename)=upper(name);
        IF SQL%NOTFOUND THEN
            raise_application_error(-20004, '不存在该雇员名');
        END IF;
    ELSE
        raise_application_error(-20001, '工资不在范围内');
    END IF;
END;
BEGIN
    SELECT min(sal),max(sal) INTO minsal,maxsal FROM emp;
END;
/

```

当执行了以上语句之后，会建立包体 EMP\_PACKAGE，其构造过程没有任何名称，它位于程序尾部，并以 BEGIN 开始（第 39 行），以 END 结束。

### 3. 调用公用组件

在建立了包规范和包体之后，就可以在应用程序中引用包的公用组件了。当在同一会话中第一次调用包的公用组件时，会自动执行其构造过程，而将来调用其他组件时则不会再调用其构造过程，所以构造过程也称为“只调用一次”的过程。当 salary 值在工资范围内时，会成功地执行相应过程。示例如下：

```

SQL> exec emp_package.add_employee(1111,'MARY',3000,20)
PL/SQL 过程已成功完成。
SQL> exec emp_package.upd_sal('mary',2000)

```

PL/SQL 过程已成功完成。

而当工资不在最低工资和最高工资之间时，则会提示错误信息。示例如下：

```
SQL> exec emp_package.upd_sal('mary',5500)
BEGIN emp_package.upd_sal('mary',5500); END;
*
ERROR 位于第 1 行:
ORA-20001: 工资不在范围内
ORA-06512: 在"SCOTT.EMP_PACKAGE", line 36
ORA-06512: 在 line 1
```

## 12.4 使用纯度级别

当使用包的公用函数时，它既可以作为表达式的一部分使用，也可以在 SQL 语句中使用。但如果要在 SQL 语句中引用包的公用函数，那么该公用函数不能包含 DML 语句（INSERT、UPDATE 和 DELETE），也不能读写远程包的变量。为了对包的公用函数加以限制，在定义包规范时可以使用纯度级别（purity level）限制公用函数。定义纯度级别的语法如下：

```
PRAGMA RESTRICT_REFERENCES (function_name,
    WNDS [,WNPS] [,RNDS] [,RNPS] );
```

如上所示，`function_name` 用于指定已经定义的函数名；`WNDS` 用于限制函数不能修改数据库数据（也即禁止执行 DML 操作）；`WNPS` 用于限制函数不能修改包变量（也即不能给包变量赋值）；`RNDS` 用于限制函数不能读取数据库数据（也即禁止执行 SELECT 操作）；`RNPS` 用于限制函数不能读取包变量（也即不能将包变量赋值给其他变量）。下面以限制函数不能修改包变量为例，说明使用纯度级别的方法。

### 1. 建立包规范

当使用纯度级别限制包的公用函数时，必须首先在包规范中定义函数，然后指定该函数的纯度级别。示例如下：

```
CREATE OR REPLACE PACKAGE purity IS
    minsal NUMBER(6,2);
    maxsal NUMBER(6,2);
    FUNCTION max_sal RETURN NUMBER;
    FUNCTION min_sal RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES(max_sal,WNPS);
    PRAGMA RESTRICT_REFERENCES(min_sal,WNPS);
END;
/
```

执行以上语句后，将建立包规范 `purity`，并指定函数 `max_sal` 和 `min_sal` 的纯度级别为 `WNPS`。

### 2. 建立包体

因为在定义包规范时为函数 `max_sal` 和 `min_sal` 指定了纯度级别 `WNPS`，所以在这两个函数内不能给变量 `minsal` 和 `maxsal` 赋值。错误示例如下：

```
CREATE OR REPLACE PACKAGE BODY purity IS
    FUNCTION max_sal RETURN NUMBER
```

```

IS
BEGIN
  SELECT max(sal) INTO maxsal FROM emp;
  RETURN maxsal;
END;
FUNCTION min_sal RETURN NUMBER
IS
BEGIN
  SELECT min(sal) INTO minsal FROM emp;
  RETURN minsal;
END;
END;
/

```

如上所示，因为函数体的 SELECT 语句为 minsal 和 maxsal 分别进行了赋值，也即修改了包变量的数据，所以会显示编译错误，在 SQL\*Plus 中通过执行 show errors 命令可以查看到如下编译错误：

```

PACKAGE BODY PURITY 出现错误:
LINE/COL ERROR
-----
2/3      PLS-00452: 子程序 'MAX_SAL' 违反了它的相关编译指示
8/3      PLS-00452: 子程序 'MIN_SAL' 违反了它的相关编译指示

```

尽管在函数体内不能为全局变量 minsal 和 maxsal 赋值，但却可以读取它们的数据。在函数体内正确引用这两个变量的包体如下：

```

CREATE OR REPLACE PACKAGE BODY purity IS
  FUNCTION max_sal RETURN NUMBER
  IS
  BEGIN
    RETURN maxsal;
  END;
  FUNCTION min_sal RETURN NUMBER
  IS
  BEGIN
    RETURN minsal;
  END;
BEGIN
  SELECT min(sal),max(sal) INTO minsal,maxsal FROM emp;
END;
/

```

如上所示，尽管在函数体内不能修改包变量 minsal 和 maxsal，但却可以读取它们的数据（RETURN 语句）。

### 3. 调用包的公用函数

当建立了包规范和包体之后，就可以在应用程序中引用包的公用组件了。为了返回雇员的最高工资和最低工资，既可以使用包的全局变量，也可以使用包的公用函数。在 SQL\*Plus 中引用包的全局变量和包的公用函数的示例如下：

```
SQL> VAR minsal NUMBER
SQL> VAR maxsal NUMBER
SQL> exec :minsal:=purity.minsal
SQL> exec :maxsal:=purity.max_sal()
SQL> PRINT minsal maxsal
      MINSAL
-----
      800
      MAXSAL
-----
      5000
```

## 12.5 习题

1. 为了在 SQL 语句中引用包函数，应该为该函数指定哪个纯度级别？  
A. WNDS      B. WNPS      C. RNDS      D. RNPS
2. 为了在其他应用程序中引用包的组件，应该在包的哪个部分定义该组件？  
A. 包规范      B. 包体
3. 为了实现信息隐藏，应该在包的哪个部分定义组件？  
A. 包规范      B. 包体
4. 建立用于操纵 ORD 表的包 ORD\_PACKAGE，并调用该包的公用过程和函数。当建立包 ORD\_PACKAGE 时，应该实现以下商业规则：
  - 定义私有函数 VALID\_CUST，检查客户号是否在 CUSTOMER 表中存在；如果客户号存在，则返回 TRUE，否则返回 FALSE；
  - 定义公用过程 ADD\_ORD，根据输入的订单号、预订日期、客户号、交付日期、订单总价为 ORD 表增加订单。如果订单已经存在，则显示自定义错误消息“ORA-20001：该订单已经存在，请核实订单号”；如果客户号不存在，则显示自定义错误消息“ORA-20002：该客户不存在，请核实客户号”；如果交付日期小于预订日期，则显示自定义错误消息“ORA-20003：交付日期不能小于预订日期，请核实交付日期”。
  - 定义公用过程 UPD\_SHIPDATE，根据输入的订单号和交付日期，更新特定订单的交付日期。如果订单不存在，则显示自定义错误消息“ORA-20004：请检查并输入正确的订单号”；如果交付日期小于预订日期，则显示自定义错误消息“ORA-20003：交付日期不能小于预订日期，请核实交付日期”。
  - 定义公用函数 GET\_INFO，根据输入的订单号返回客户名和订单总价。如果订单不存在，则显示自定义错误消息“ORA-20004：请检查并输入正确的订单号”。
  - 定义公用过程 DEL\_ORD，根据输入的订单号取消特定订单。如果订单不存在，则显示自定义错误消息“ORA-20004：请检查并输入正确的订单号”。
5. 建立用于操纵 CUSTOMER 表的包 CUST\_PACKAGE，然后调用其公用的过程和函数。当建立包 CUST\_PACKAGE，实现以下商业规则：
  - 定义重载过程 UPD\_CITY，分别根据输入的客户号或客户名更新客户所在城市。如

果客户号或客户名不存在，则显示自定义错误消息“ORA-20001：该客户不存在，请核实客户号或客户名”；

- 定义重载函数 GET\_CITY，分别根据输入的客户号或客户名返回客户所在城市。如果客户号或客户名不存在，则显示自定义错误消息“ORA-20001：该客户不存在，请核实客户号或客户名”。

# 第 13 章 开发触发器

触发器是指存放在数据库中，并被隐含执行的存储过程。在 Oracle8i 之前，只允许基于表或视图的 DML 操作（INSERT, UPDATE 和 DELETE）建立触发器；而从 Oracle8i 开始，不仅支持 DML 触发器，也允许基于系统事件（启动数据库、关闭数据库、登录）和 DDL 操作建立触发器。本章将详细介绍如何建立各种类型的触发器，在学习了本章之后，读者应学会：

- 建立各种 DML 触发器；
- 建立 INSTEAD OF 触发器；
- 建立系统事件触发器。

## 13.1 触发器简介

触发器是指被隐含执行的存储过程，它可以使用 PL/SQL, Java 和 C 进行开发。当发生特定事件（例如修改表、建立对象、登录到数据库）时，Oracle 会自动执行触发器的相应代码。触发器由触发事件、触发条件和触发操作三部分组成。

### 1. 触发事件

触发事件是指引起触发器被触发的 SQL 语句、数据库事件或用户事件。在 Oracle8i 之前，触发事件只能是 DML 操作；而从 Oracle8i 开始，不仅支持原有的 DML 事件，而且还增加了其他触发事件。具体的触发事件如下：

- 启动和关闭例程；
- Oracle 错误消息；
- 用户登录和断开会话；
- 特定表或视图的 DML 操作；
- 在任何方案上的 DDL 语句。

### 2. 触发条件（可选）

触发条件是指使用 WHEN 子句指定一个 BOOLEAN 表达式，当布尔表达式返回值为 TRUE 时，会自动执行触发器相应代码；当布尔表达式返回值为 FALSE 或 UNKNOWN 时，不会执行触发操作。

### 3. 触发操作

触发操作是指包含 SQL 语句和其他执行代码的 PL/SQL 块，不仅可以使用 PL/SQL 进行开发，也可以使用 Java 语言和 C 语言进行开发。当触发条件为 TRUE 时，会自动执行触发操作的相应代码。但编写触发器执行代码时，需要注意以下限制：

- 触发器代码的大小不能超过 32K。如果确实需要使用大量代码建立触发器，应该首先建立存储过程，然后在触发器中使用 CALL 语句调用存储过程。
- 触发器代码只能包含 SELECT, INSERT, UPDATE 和 DELETE 语句，而不能包含 DDL 语句（CREATE, ALTER, DROP）和事务控制语句（COMMIT, ROLLBACK）

和 SAVEPOINT)。

## 13.2 建立 DML 触发器

在 Oracle8i 之前，只能基于 DML 事件建立触发器。在建立了 DML 触发器之后，如果发生了相应的 DML 操作，就会自动执行触发器的相应代码。当建立 DML 触发器时，需要指定触发时机 (BEFORE 或 AFTER)、触发事件 (INSERT, UPDATE, DELETE)、表名、触发类型、触发条件以及触发操作。

### 1. 触发时机

触发时机用于指定触发器的触发时间。当指定 BEFORE 关键字时，表示在执行 DML 操作之前触发触发器；当指定 AFTER 关键字时，表示在执行了 DML 操作之后触发触发器。

### 2. 触发事件

触发事件用于指定导致触发器执行的 DML 操作，也即 INSERT, UPDATE 和 DELETE 操作。既可以使用单个触发事件，也可以组合多个触发事件。

### 3. 表名

因为 DML 触发器是针对特定表进行的，所以必须指定 DML 操作所对应的表。

### 4. 触发类型

触发类型用于指定当触发事件发生之后，需要执行几次触发操作。如果指定语句触发类型（默认），那么只会执行一次触发器代码；如果指定行触发类型，则会在每个被作用行上执行一次触发器代码。

### 5. 触发条件

触发条件用于指定执行触发器代码的条件，只有条件为 TRUE 时才会执行触发器代码。注意，当编写 DML 触发器时，只允许在行触发器上指定触发条件。

### 6. 触发操作

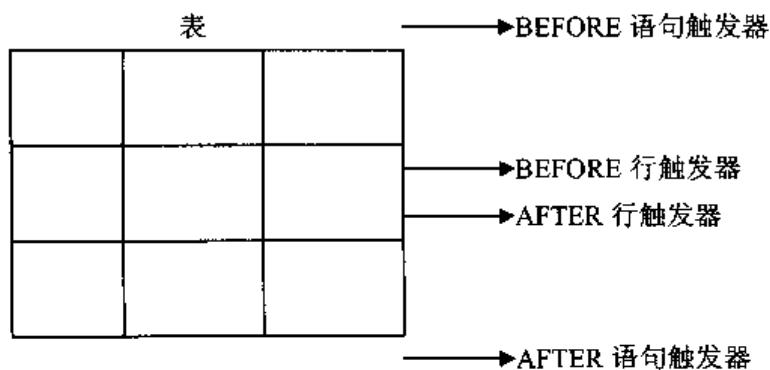
触发操作用于指定触发器执行代码。如果使用 PL/SQL 存储过程、Java 存储过程或外部存储过程来实现触发器代码，那么在触发操作部分可直接使用 CALL 语句调用相应过程。如果使用 PL/SQL 匿名块编写触发操作，则应该按照以下格式进行编写：

```
[DECLARE]
    ——定义变量、常量等
BEGIN
    ——编写 SQL 语句和 PL/SQL 语句
EXCEPTION
    ——编写例外处理语句
END;
```

### 7. DML 触发器触发顺序

#### (1) DML 触发器在单行数据上的触发顺序

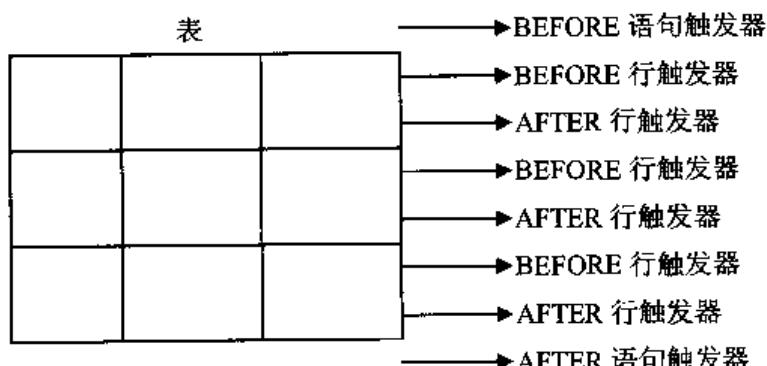
当针对某一表的相同 DML 操作而建立了多个 DML 触发器 (BEFORE/AFTER 语句触发器、BEFORE/AFTER 行触发器) 时，如果在单行数据上执行了该种 DML 操作，则触发器会按照以下顺序执行：



如图所示，对于单行数据而言，无论是语句触发器，还是行触发器，触发器代码实际只被执行一次，并且执行顺序为 BEFORE 语句触发器、BEFORE 行触发器、DML 操作、AFTER 行触发器、AFTER 语句触发器。

### (2) DML 触发器在多行数据上的触发顺序

当针对某一表的相同 DML 操作而建立了多个 DML 触发器（BEFORE/AFTER 语句触发器、BEFORE/AFTER 行触发器）时，如果在多行数据上执行了该种 DML 操作，则触发器会按照以下顺序执行：



如图所示，对于多行数据而言，语句触发器只被执行一次，而行触发器在每个作用行上都执行一次。

#### 13.2.1 语句触发器

语句触发器是指当执行 DML 语句时被隐含执行的触发器。如果在表上针对某种 DML 操作建立了语句触发器，那么当执行 DML 操作时会自动执行触发器的相应代码。当审计 DML 操作，或者确保 DML 操作安全执行时，可以使用语句触发器。注意，使用语句触发器时，不能记录列数据的变化。建立语句触发器的语法如下：

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing event1 [OR event2 OR event3]
  ON table_name
  PL/SQL block;
```

如上所示，trigger\_name 用于指定触发器名；timing 用于指定触发时机（BEFORE 或 AFTER）；event 用于指定触发事件（INSERT、UPDATE 和 DELETE）；table\_name 用于指定

DML 操作所对应的表名。下面通过示例说明如何建立语句触发器。

### 1. 建立 BEFORE 语句触发器

为了确保 DML 操作在正常情况下执行，可以基于 DML 操作建立 BEFORE 语句触发器。例如，为了禁止工作人员在休息日改变雇员信息，开发人员可以建立 BEFORE 语句触发器，以实现数据的安全保护。示例如下：

```
CREATE OR REPLACE TRIGGER tr_sec_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
  IF to_char(sysdate,'DY','nls_date_language=AMERICAN')
    IN ('SAT','SUN') THEN
    raise_application_error(-20001,
      '不能在休息日改变雇员信息');
  END IF;
END;
/
```

在建立了触发器 tr\_sec\_emp 之后，如果星期六、星期日在 EMP 表上执行 DML 操作，则会显示错误信息。示例如下：

```
SQL> UPDATE emp SET sal=sal*1.1 WHERE deptno=10;
UPDATE emp SET sal=sal*1.1 WHERE deptno=10
*
ERROR 位于第 1 行:
ORA-20001: 不能在休息日改变雇员信息
ORA-06512: 在 "SCOTT.TR_SEC_EMP", line 4
ORA-04088: 触发器 'SCOTT.TR_SEC_EMP' 执行过程中出错
```

### 2. 使用条件谓词

当在触发器中同时包含多个触发事件（INSERT, UPDATE, DELETE）时，为了在触发器代码中区分具体的触发事件，可以使用以下三个条件谓词：

- **INSERTING**: 当触发事件是 INSERT 操作时，该条件谓词返回值为 TRUE，否则为 FALSE。
- **UPDATING**: 当触发事件是 UPDATE 操作时，该条件谓词返回值为 TRUE，否则为 FALSE。
- **DELETING**: 当触发事件是 DELETE 操作时，该条件谓词返回值为 TRUE，否则为 FALSE。

下面举例说明在触发器中使用这三个条件谓词的方法，示例如下：

```
CREATE OR REPLACE TRIGGER tr_sec_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
  IF to_char(sysdate,'DY','nls_date_language=AMERICAN')
    IN ('SAT','SUN') THEN
    CASE
      WHEN INSERTING THEN
        raise_application_error(-20001,
          '不能在休息日增加雇员');
```

```

    WHEN UPDATING THEN
        raise_application_error(-20002,
            '不能在休息日更新雇员');
    WHEN DELETING THEN
        raise_application_error(-20003,
            '不能在休息日解雇雇员');
    END CASE;
END IF;
END;
/

```

当建立了触发器 tr\_sec\_emp 之后，如果星期六、星期日在 EMP 表上执行 DML 操作，则会根据不同操作显示不同的错误号和错误消息。示例如下：

```

SQL> DELETE FROM emp WHERE empno=7788;
DELETE FROM emp WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-20003: 不能在休息日解雇雇员
ORA-06512: 在"SCOTT.TR_SEC_EMP", line 12
ORA-04088: 触发器 'SCOTT.TR_SEC_EMP' 执行过程中出错

```

### 3. 建立 AFTER 语句触发器

为了审计 DML 操作，或者在 DML 操作之后执行汇总运算，可以使用 AFTER 语句触发器。例如，为了审计在 EMP 表上 INSERT, UPDATE 和 DELETE 的操作次数，可以建立 AFTER 触发器。在建立 AFTER 触发器之前，首先建立审计表 audit\_table。示例如下：

```

CREATE TABLE audit_table(
    name VARCHAR2(20),ins INT,upd INT,del INT,
    starttime DATE,endtime DATE);

```

为了审计在 EMP 表上 DML 操作执行的次数、最早执行时间和最近执行时间，需要建立 AFTER 语句触发器。示例如下：

```

CREATE OR REPLACE TRIGGER tr_audit_emp
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_temp INT;
BEGIN
    SELECT count(*) INTO v_temp FROM audit_table
    WHERE name='EMP';
    IF v_temp=0 THEN
        INSERT INTO audit_table VALUES
        ('EMP',0,0,0,SYSDATE,NULL);
    END IF;
    CASE
        WHEN INSERTING THEN
            UPDATE audit_table SET ins=ins+1,endtime=SYSDATE
            WHERE name='EMP';
        WHEN UPDATING THEN
            UPDATE audit_table SET upd=upd+1,endtime=SYSDATE

```

```

        WHERE name='EMP';
        WHEN DELETING THEN
            UPDATE audit_table SET del=del+1,endtime=SYSDATE
            WHERE name='EMP';
        END CASE;
    END;
/

```

在建立了触发器 `tr_audit_emp` 之后，在 `EMP` 表上执行 DML 操作时，都会将 DML 操作次数以及时间段记录在审计表 `audit_table` 中。示例如下：

```

SQL> UPDATE emp SET sal=2000 WHERE empno=7788;
SQL> UPDATE emp SET sal=2000 WHERE empno=7369;
SQL> SELECT upd,starttime,endtime FROM audit_table
  2 WHERE name='EMP';
          UPD STARTTIME ENDTIME
-----
2 17-12月-03 17-12月-03

```

### 13.2.2 行触发器

行触发器是指执行 DML 操作时，每作用一行就触发一次的触发器。审计数据变化时，可以使用行触发器。建立行触发器的语法如下：

```

CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
[WHEN condition]
PL/SQL block;

```

如上所示，`trigger_name` 用于指定触发器名；`timing` 用于指定触发时机（`BEFORE` 或 `AFTER`）；`event` 用于指定触发事件（`INSERT`, `UPDATE`, `DELETE`）；`REFERENCING` 子句用于指定引用新、旧数据的方式，默认情况下使用 `old` 修饰符引用旧数据，使用 `new` 修饰符引用新数据；`table_name` 用于指定 DML 操作所对应的表；`FOR EACH ROW` 表示建立行触发器；`WHEN` 子句（可选）用于指定触发条件。下面通过示例说明如何建立行触发器。

#### 1. 建立 BEFORE 行触发器

在开发数据库应用时，为了确保数据符合商业逻辑或企业规则，应该使用约束对输入数据加以限制，但某些情况下使用约束可能无法实现复杂的商业逻辑或企业规则，此时可以考虑使用 `BEFORE` 行触发器。下面以确保雇员工资不能低于其原有工资为例，说明建立 `BEFORE` 行触发器的方法。示例如下：

```

CREATE OR REPLACE TRIGGER tr_emp_sal
BEFORE UPDATE OF sal ON emp
FOR EACH ROW
BEGIN
    IF :new.sal < :old.sal THEN
        raise_application_error(-20010,'工资只涨不降');
    END IF;
END;

```

```

END IF;
END;
/

```

在建立触发器 `tr_emp_sal` 之后，如果雇员新工资低于其原有工资，则会提示错误信息。示例如下：

```

SQL> UPDATE emp SET sal=800 WHERE empno=7788;
UPDATE emp SET sal=800 WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-20010: 工资只涨不降
ORA-06512: 在"SCOTT.TR_EMP_SAL", line 3
ORA-04088: 触发器 'SCOTT.TR_EMP_SAL' 执行过程中出错

```

## 2. 建立 AFTER 行触发器

为了审计 DML 操作，可以使用语句触发器或 Oracle 系统提供的审计功能；而为了审计数据变化，则应该使用 AFTER 行触发器。下面以审计雇员工资变化为例，说明使用 AFTER 行触发器的方法。在建立触发器之前，首先应建立存放审计数据的表 `audit_emp_change`，示例如下：

```

CREATE TABLE audit_emp_change (
    name VARCHAR2(10), oldsal NUMBER(6,2),
    newsal NUMBER(6,2), time DATE);

```

为了审计所有雇员的工资变化和雇员工资的更新日期，必须要建立 AFTER 行触发器。示例如下：

```

CREATE OR REPLACE TRIGGER tr_sal_change
AFTER UPDATE OF sal ON emp
FOR EACH ROW
DECLARE
    v_temp INT;
BEGIN
    SELECT count(*) INTO v_temp FROM audit_emp_change
        WHERE name=:old.ename;
    IF v_temp=0 THEN
        INSERT INTO audit_emp_change
            VALUES(:old.ename,:old.sal,:new.sal,SYSDATE);
    ELSE
        UPDATE audit_emp_change
            SET oldsal=:old.sal,newsal=:new.sal,time=SYSDATE
            WHERE name=:old.ename;
    END IF;
END;
/

```

在建立触发器 `tr_sal_change` 之后，当修改雇员工资时，会将每个雇员的工资变化全部写入到审计表 `audit_emp_change` 中。示例如下：

```

SQL> UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
已更新 6 行。
SQL> SELECT * FROM audit_emp_change;

```

NAME	OLDSAL	NEWSAL	TIME
ALLEN	1760	1936	17-12 月-03
WARD	1375	1512.5	17-12 月-03
MARTIN	1375	1512.5	17-12 月-03
BLAKE	3135	3448.5	17-12 月-03
TURNER	1650	1815	17-12 月-03
JAMES	1045	1149.5	17-12 月-03

### 3. 限制行触发器

当使用行触发器时，默认情况下会在每个被作用行上执行一次触发器代码。为了使得在特定条件下执行行触发器代码，就需要使用 WHEN 子句对触发条件加以限制。下面以审计岗位为“SALESMAN”的雇员工资变化为例，说明限制行触发器的方法。示例如下：

```

CREATE OR REPLACE TRIGGER tr_sal_change
AFTER UPDATE OF sal ON emp
FOR EACH ROW
WHEN (old.job='SALESMAN')
DECLARE
    v_temp INT;
BEGIN
    SELECT count(*) INTO v_temp FROM audit_emp_change
    WHERE name=:old.ename;
    IF v_temp=0 THEN
        INSERT INTO audit_emp_change
        VALUES(:old.ename,:old.sal,:new.sal,SYSDATE);
    ELSE
        UPDATE audit_emp_change
        SET oldsal=:old.sal,newsal=:new.sal,time=SYSDATE
        WHERE name=:old.ename;
    END IF;
END;
/

```

当建立触发器 tr\_sal\_change 时，因为使用 WHEN 子句指定了触发条件，所以只有在满足触发条件时才会执行触发器代码。这样，当修改部门 30 的雇员工资时，只有部分雇员会被审计。示例如下：

```

SQL> UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
已更新 6 行。
SQL> SELECT * FROM audit_emp_change;
NAME      OLDSAL     NEWSAL TIME
-----  -----  -----  -----
ALLEN      1600      1760  17-12 月-03
WARD       1250      1375  17-12 月-03
MARTIN     1250      1375  17-12 月-03
TURNER     1500      1650  17-12 月-03

```

### 4. DML 触发器使用注意事项

当编写 DML 触发器时，触发器代码不能从触发器所对应的基表中读取数据。例如，如果

要基于 EMP 表建立触发器，那么该触发器的执行代码不能包含对 EMP 表的查询操作。尽管在建立触发器时不会出现任何错误，但在执行相应触发操作时会显示错误信息。假定希望雇员工资不能超过当前的最高工资，并使用触发器实现该规则。示例如下：

```
CREATE OR REPLACE TRIGGER tr_emp_sal
BEFORE UPDATE OF sal ON emp
FOR EACH ROW
DECLARE
    maxsal NUMBER(6,2);
BEGIN
    SELECT max(sal) INTO maxsal FROM emp;
    IF :new.sal>maxsal THEN
        raise_application_error(-20010,'超出工资上限');
    END IF;
END;
/
```

如上所示，当建立触发器 tr\_emp\_sal 时，不会显示任何错误。但因为触发器代码引用了基表 emp，所以在执行 UPDATE 操作时会显示如下错误消息：

```
SQL> UPDATE emp SET sal=6000 WHERE empno=7788;
UPDATE emp SET sal=6000 WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-04091: 表 SCOTT.EMP 发生了变化, 触发器/函数不能读
ORA-06512: 在"SCOTT.TR_EMP_SAL", line 4
ORA-04088: 触发器 'SCOTT.TR_EMP_SAL' 执行过程中出错
```

### 13.2.3 使用 DML 触发器

为了确保数据库数据满足特定的商业规则或企业逻辑，可以使用约束、触发器和子程序实现。因为约束性能最好，实现最简单，所以首选约束；如果使用约束不能实现特定规则，那么应该选择触发器；如果触发器仍然不能实现特定规则，那么应该选择子程序（过程和函数）。DML 触发器可以用于实现数据安全保护、数据审计、数据完整性、参照完整性、数据复制等功能，下面通过示例给大家说明如何实现这些功能。

#### 1. 控制数据安全

在服务器级控制数据安全是通过授予和收回对象权限来完成的，例如为了使得 SMITH 用户可以在 SCOTT.EMP 表上执行 DML 操作和 SELECT 操作，必须要为 SMITH 用户授予相应的对象权限。如下所示：

```
SQL> CONN SCOTT/TIGER
SQL> GRANT SELECT,INSERT,UPDATE,DELETE ON emp TO SMITH;
```

当用户具有了以上对象权限之后，就可以随时在 EMP 表上执行相应的 SQL 操作。为了实现更复杂的安全模型（例如限制要修改的数据、修改时间等），就需要使用 DML 触发器了。下面以限制用户在正常工作时间（9:00~17:00）改变 EMP 表数据为例，说明使用 DML 触发器控制数据安全的方法。示例如下：

```
CREATE OR REPLACE TRIGGER tr_emp_time
```

```

BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
    IF to_char(SYSDATE, 'HH24') NOT BETWEEN
        '9' AND '17' THEN
        raise_application_error(-20101, '非工作时间');
    END IF;
END;
/

```

建立了触发器 `tr_emp_time` 之后，只能在 9:00~17:00 之间在 EMP 表上执行 DML 操作。如果不在该时间段，则会显示如下错误信息：

```

SQL> UPDATE emp SET sal=3200 WHERE empno=7788;
UPDATE emp SET sal=3200 WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-20101: 非工作时间
ORA-06512: 在"SCOTT.TR_EMP_TIME", line 4
ORA-04088: 触发器 'SCOTT.TR_EMP_TIME' 执行过程中出错

```

## 2. 实现数据审计

审计可以用于监视非法和可疑的数据库活动。Oracle 数据库本身提供了审计功能，例如，如果要对 EMP 表上的 DML 操作进行审计，可以执行如下命令：

```

SQL> AUDIT INSERT,UPDATE,DELETE ON emp
  2 BY ACCESS;

```

如上所示，在设置了审计选项之后，如果在 EMP 表上执行 INSERT、UPDATE 和 DELETE 操作，Oracle 会将关于 SQL 操作的信息（用户、时间等）写入到数据字典中。注意，使用数据库审计只能审计 SQL 操作，而不会记载数据变化。为了审计 SQL 操作所引起的数据变化，必须要使用 DML 触发器。示例如下：

```

CREATE OR REPLACE TRIGGER tr_sal_change
AFTER UPDATE OF sal ON emp
FOR EACH ROW
DECLARE
    v_temp INT;
BEGIN
    SELECT count(*) INTO v_temp FROM audit_emp_change
    WHERE name=:old.ename;
    IF v_temp=0 THEN
        INSERT INTO audit_emp_change
        VALUES(:old.ename,:old.sal,:new.sal,SYSDATE);
    ELSE
        UPDATE audit_emp_change
        SET oldsal=:old.sal,newsal=:new.sal,time=SYSDATE
        WHERE name=:old.ename;
    END IF;
END;
/

```

在建立了触发器 `tr_sal_change` 之后，当修改雇员工资时，会将每个雇员的工资变化全部写入到审计表 `audit_emp_change` 中。示例如下：

```
SQL> UPDATE emp SET sal=sal*1.1 WHERE deptno=30;
SQL> SELECT * FROM audit_emp_change;
NAME          OLDSAL      NEWSAL        TIME
-----        -----       -----        -----
ALLEN         1760        1936        17-12月-03
WARD          1375        1512.5      17-12月-03
MARTIN        1375        1512.5      17-12月-03
BLAKE          3135        3448.5      17-12月-03
TURNER        1650        1815        17-12月-03
JAMES          1045        1149.5      17-12月-03
```

### 3. 实现数据完整性

数据完整性用于确保数据库数据满足特定的商业逻辑或企业规则。数据完整性可以使用约束、触发器和子程序实现。因为约束的实现最简单，性能也最好，所以实现数据完整性首选约束。例如，为了限制雇员工资不能低于 800 元，可以选用 CHECK 约束。示例如下：

```
SQL> ALTER TABLE emp ADD CONSTRAINT ck_sal
  2 CHECK (sal>=800);
```

但某些情况下使用约束无法实现特定的商业规则，此时可以使用触发器来实现数据完整性。例如，假定希望雇员的新工资不能低于其原工资，但也不能高出原工资的 20%，使用约束显然无法实现该规则，但通过触发器却可以实现该项规则。示例如下：

```
CREATE OR REPLACE TRIGGER tr_check_sal
BEFORE UPDATE OF sal ON emp
FOR EACH ROW
WHEN (new.sal<old.sal OR new.sal>1.2*old.sal)
BEGIN
  raise_application_error(-20931,
    '工资只升不降，并且升幅不能超过 20%');
END;
/
```

在建立了触发器 `tr_check_sal` 之后，如果雇员新工资不符合相应规则，则会提示错误信息。示例如下：

```
SQL> UPDATE emp SET sal=sal*1.25 WHERE empno=7788;
UPDATE emp SET sal=sal*1.25 WHERE empno=7788
*
ERROR 位于第 1 行:
ORA-20931: 工资只升不降，并且升幅不能超过 20%
ORA-06512: 在"SCOTT.TR_CHECK_SAL", line 2
ORA-04088: 触发器 'SCOTT.TR_CHECK_SAL' 执行过程中出错
```

### 4. 实现参照完整性

参照完整性是指若两个表之间具有主从关系（也即主外键关系），当删除主表数据时，必须确保相关的从表数据已经被删除；当修改主表的主键列数据时，必须确保相关从表数据已经被修改。为了实现级联删除，可以在定义外部键约束时指定 `ON DELETE CASCADE` 关键字。

示例如下：

```
SQL> ALTER TABLE emp ADD CONSTRAINT fk_deptno
  2 FOREIGN KEY (deptno) REFERENCES dept(deptno)
  3 ON DELETE CASCADE;
```

当用如上方式建立了外部键约束 fk\_deptno 之后，在删除主表 DEPT 的数据时，会同时删除从表 EMP 的所有相关数据。但使用约束却不能实现级联更新，如果要更新 DEPT 表的部门号，则会显示如下错误信息：

```
SQL> UPDATE dept SET deptno=50 WHERE deptno=10;
UPDATE dept SET deptno=50 WHERE deptno=10
*
ERROR 位于第 1 行:
ORA-02292: 违反完整约束条件 (SCOTT.FK_DEPTNO) - 已找到子记录日志
```

如上所示，错误原因是 EMP 表包含有该部门的相应雇员。为了实现级联更新，可以使用触发器。示例如下：

```
CREATE OR REPLACE TRIGGER tr_update_cascade
AFTER UPDATE OF deptno ON dept
FOR EACH ROW
BEGIN
  UPDATE emp SET deptno=:new.deptno
    WHERE deptno=:old.deptno;
END;
/
```

在建立了触发器 tr\_update\_cascade 之后，当更新 DEPT 表的部门号时，会级联更新 EMP 表的相应雇员的部门号。示例如下：

```
SQL> UPDATE dept SET deptno=50 WHERE deptno=10;
SQL> SELECT ename FROM emp WHERE deptno=50;
ENAME
-----
CLARK
KING
MILLER
```

### 13.3 建立 INSTEAD OF 触发器

对于简单视图，可以直接执行 INSERT, UPDATE 和 DELETE 操作。但是对于复杂视图，不允许直接执行 INSERT, UPDATE 和 DELETE 操作。当视图符合以下任何一种情况时，都不允许直接执行 DML 操作。具体情况如下：

- 具有集合操作符 (UNION, UNION ALL, INTERSECT, MINUS);
- 具有分组函数 (MIN, MAX, SUM, AVG, COUNT 等);
- 具有 GROUP BY, CONNECT BY 或 START WITH 等子句;
- 具有 DISTINCT 关键字;
- 具有连接查询。

为了在具有以上情况的复杂视图上执行 DML 操作，必须要基于视图建立 INSTEAD-OF 触发器。在建立了 INSTEAD-OF 触发器之后，就可以基于复杂视图执行 INSERT, UPDATE 和 DELETE 语句。但建立 INSTEAD-OF 触发器有以下注意事项：

- INSTEAD OF 选项只适用于视图；
- 当基于视图建立触发器时，不能指定 BEFORE 和 AFTER 选项；
- 在建立视图时没有指定 WITH CHECK OPTION 选项；
- 当建立 INSTEAD OF 触发器时，必须指定 FOR EACH ROW 选项。

下面举例说明基于复杂视图建立 INSTEAD-OF 触发器的方法。

### 1. 建立复杂视图 dept\_emp

视图是逻辑表，本身没有任何数据。视图只是对应于一条 SELECT 语句，当查询视图时，其数据实际是从视图基表上取得。为了简化部门及其雇员信息的查询，应建立复杂视图 dept\_emp。示例如下：

```
CREATE OR REPLACE VIEW dept_emp AS
  SELECT a.deptno, a.dname, b.empno, b.ename
    FROM dept a, emp b
   WHERE a.deptno=b.deptno;
```

当执行以上语句建立了复杂视图 dept\_emp 之后，直接查询视图 dept\_emp 会显示部门及其雇员信息，但不允许执行 DML 操作。示例如下：

```
SQL> SELECT * FROM dept_emp WHERE deptno=10;
      DEPTNO  DNAME          EMPNO  ENAME
-----  -----
        10 ACCOUNTING      7782  CLARK
        10 ACCOUNTING      7839  KING
        10 ACCOUNTING      7934  MILLER
SQL> INSERT INTO dept_emp VALUES(50, 'ADMIN', '1223', 'MARY');
      INSERT INTO dept_emp VALUES(50, 'ADMIN', '1223', 'MARY')
      *
      ERROR 位于第 1 行:
      ORA-01779: 无法修改与非键值保存表对应的列
```

### 2. 建立 INSTEAD-OF 触发器

为了在复杂视图上执行 DML 操作，必须要基于复杂视图来建立 INSTEAD-OF 触发器。下面以在复杂视图 dept\_emp 上执行 INSERT 操作为例，说明建立 INSTEAD-OF 触发器的方法。示例如下：

```
CREATE OR REPLACE TRIGGER tr_instead_of_dept_emp
INSTEAD OF INSERT ON dept_emp
FOR EACH ROW
DECLARE
  v_temp INT;
BEGIN
  SELECT count(*) INTO v_temp FROM dept
    WHERE deptno=:new.deptno;
  IF v_temp=0 THEN
    INSERT INTO dept (deptno, dname)
```

```

        VALUES (:new.deptno, :new.dname);
END IF;
SELECT count(*) INTO v_temp FROM emp
WHERE empno=:new.empno;
IF v_temp=0 THEN
    INSERT INTO emp (empno,ename,deptno)
        VALUES (:new.empno,:new.ename,:new.deptno);
END IF;
END;
/

```

当建立了 INSTEAD-OF 触发器 tr\_instead\_of\_dept\_emp 之后，就可以在复杂视图 dept\_emp 上执行 INSERT 操作了。示例如下：

```

SQL> INSERT INTO dept_emp VALUES(50,'ADMIN','1223','MARY');
SQL> INSERT INTO dept_emp VALUES(10,'ADMIN','1224','BAKE');

```

执行了以上两条 INSERT 语句之后，就为 DEPT 表插入了一条数据，为 EMP 表插入了两条数据，想想其原因。

## 13.4 建立系统事件触发器

系统事件触发器是指基于 Oracle 系统事件（例如 LOGON 和 STARTUP）所建立的触发器。通过使用系统事件触发器，提供了跟踪系统或数据库变化的机制。下面介绍一些常用的系统事件属性函数，以及建立各种事件触发器的方法。

### 1. 常用事件属性函数

建立系统事件触发器时，应用开发人员经常需要使用事件属性函数。常用的事件属性函数如下：

- ora\_client\_ip\_address：用于返回客户端的 IP 地址。
- ora\_database\_name：用于返回当前数据库名。
- ora\_des\_encrypted\_password：用于返回 DES 加密后的用户口令。
- ora\_dict\_obj\_name：用于返回 DDL 操作所对应的数据库对象名。
- ora\_dict\_obj\_name\_list(name\_list OUT ora\_name\_list\_t)：用于返回在事件中被修改的对象名列表。
- ora\_dict\_obj\_owner：用于返回 DDL 操作所对应的对象的所有者名。
- ora\_dict\_obj\_owner\_list(owner\_list OUT ora\_name\_list\_t)：用于返回在事件中被修改对象的所有者列表。
- ora\_dict\_obj\_type：用于返回 DDL 操作所对应的数据库对象的类型。
- ora\_grantee(user\_list OUT ora\_name\_list\_t)：用于返回授权事件的授权者。
- ora\_instance\_num：用于返回例程号。
- ora\_is\_alter\_column(column\_name IN VARCHAR2)：用于检测特定列是否被修改。
- ora\_is\_creating\_nested\_table：用于检测是否正在建立嵌套表。
- ora\_is\_drop\_column(column\_name IN VARCHAR2)：用于检测特定列是否被删除。
- ora\_is\_servererror(error\_number)：用于检测是否返回了特定 Oracle 错误。

- ora\_login\_user: 用于返回登录用户名。
- ora\_sysevent: 用于返回触发触发器的系统事件名。

## 2. 建立例程启动和关闭触发器

为了跟踪例程启动和关闭事件，可以分别建立例程启动触发器和例程关闭触发器。为了记载例程启动和关闭的事件和时间，首先建立事件表 event\_table。示例如下：

```
SQL> conn sys/oracle as sysdba
SQL> create table event_table(event varchar2(30),time date);
```

在建立了事件表 event\_table 之后，就可以在触发器中引用该表了。注意，例程启动触发器和例程关闭触发器只有特权用户才能建立，并且例程启动触发器只能使用 AFTER 关键字，而例程关闭触发器只能使用 BEFORE 关键字。示例如下：

```
CREATE OR REPLACE TRIGGER tr_startup
AFTER STARTUP ON DATABASE
BEGIN
    INSERT INTO event_table VALUES(ora_sysevent,SYSDATE);
END;
/
CREATE OR REPLACE TRIGGER tr_shutdown
BEFORE SHUTDOWN ON DATABASE
BEGIN
    INSERT INTO event_table VALUES(ora_sysevent,SYSDATE);
END;
/
```

在建立了触发器 tr\_startup 之后，当打开数据库之后，会执行该触发器的相应代码；在建立了触发器 tr\_shutdown 之后，在关闭例程之前，会执行该触发器的相应代码，但 SHUTDOWN ABORT 命令不会触发该触发器。示例如下：

```
SQL> SHUTDOWN
SQL> STARTUP
SQL> SELECT event,to_char(time,'YYYY/MM/DD HH24:MI') time
  2 FROM event_table;
EVENT                      TIME
-----
SHUTDOWN                   2003/12/18 09:25
STARTUP                    2003/12/18 09:26
```

## 3. 建立登录和退出触发器

为了记载用户登录和退出事件，可以分别建立登录和退出触发器。为了记载登录用户和退出用户的名称、时间和 IP 地址，应该首先建立专门存放登录和退出的信息表 LOG\_TABLE。示例如下：

```
SQL> conn sys/oracle as sysdba
SQL> CREATE TABLE log_table(
  2   username VARCHAR2(20),logon_time DATE,
  3   logoff_time DATE,address VARCHAR2(20));
```

在建立了 LOG\_TABLE 表之后，就可以在触发器中引用该表了。注意，登录触发器和退出触发器一定要以特权用户身份建立，并且登录触发器只能使用 AFTER 关键字，而退出触发器

器只能使用 BEFORE 关键字。示例如下：

```

CREATE OR REPLACE TRIGGER tr_logon
AFTER LOGON ON DATABASE
BEGIN
    INSERT INTO log_table (username,logon_time,address)
        VALUES(ora_login_user,SYSDATE,ora_client_ip_address);
END;
/
CREATE OR REPLACE TRIGGER tr_logoff
BEFORE LOGOFF ON DATABASE
BEGIN
    INSERT INTO log_table (username,logoff_time,address)
        VALUES(ora_login_user,SYSDATE,ora_client_ip_address);
END;
/

```

在建立了触发器 tr\_logon 之后，当用户登录到数据库之后，会执行其触发器代码；在建立了触发器 tr\_logoff 之后，当用户断开数据库连接之前，会执行其触发器代码。示例如下：

```

SQL> conn scott/tiger@orcl
SQL> conn system/manager@orcl
SQL> conn sys/oracle@orcl as sysdba
SQL> select * from log_table;

```

USERNAME	LOGON_TIME	LOGOFF_TIM	ADDRESS
SCOTT		18-12月-03	
SYSTEM	18-12月-03		127.0.0.1
SYSTEM		18-12月-03	
SYS	18-12月-03		127.0.0.1
SYS		18-12月-03	
SCOTT	18-12月-03		127.0.0.1

#### 4. 建立 DDL 触发器

为了记载系统所发生的 DDL 事件（CREATE, ALTER, DROP 等），可以建立 DDL 触发器。为了记载 DDL 事件信息，应该建立专门的表，以便存放 DDL 事件信息。示例如下：

```

SQL> conn sys/oracle as sysdba
SQL> CREATE TABLE event_ddl(
  2   event VARCHAR2(20),username VARCHAR2(10),
  3   owner VARCHAR2(10),objname VARCHAR2(20),
  4   objtype VARCHAR2(10),time DATE);

```

在建立了表 event\_ddl 之后，就可以在触发器中引用该表了。为了记载 DDL 事件，应该建立 DDL 触发器。注意，当建立 DDL 触发器时，必须要使用 AFTER 关键字。示例如下：

```

CREATE OR REPLACE TRIGGER tr_ddl
AFTER DDL ON scott.schema
BEGIN
    INSERT INTO event_ddl VALUES(
        ora_sysevent,ora_login_user,ora_dict_obj_owner,
        ora_dict_obj_name,ora_dict_obj_type,SYSDATE);

```

```
END;
```

```
/
```

在建立了触发器 tr\_ddl 之后，如果在 SCOTT 方案对象上执行了 DDL 操作，则会将该信息记载到表 event\_ddl 中。示例如下：

```
SQL> conn scott/tiger
SQL> CREATE TABLE temp(cola int);
SQL> DROP TABLE temp;
SQL> select event,owner,objname from event_ddl;
EVENT          OWNER      OBJNAME
-----
CREATE        SCOTT      TEMP
DROP          SCOTT      TEMP
```

## 13.5 管理触发器

### 1. 显示触发器信息

建立触发器时，Oracle 会将触发器信息写入到数据字典中，通过查询数据字典视图 USER\_TRIGGERS，可以显示当前用户所包含的所有触发器信息。示例如下：

```
SQL> conn scott/tiger
SQL> SELECT trigger_name,status FROM user_triggers
  2 WHERE table_name='EMP';
TRIGGER_NAME          STATUS
-----
TR_AUDIT_EMP          ENABLED
TR_CHECK_SAL          ENABLED
TR_EMP_SAL            ENABLED
TR_SAL_CHANGE          ENABLED
TR_SEC_EMP            ENABLED
```

### 2. 禁止触发器

禁止触发器是指使触发器临时失效。当触发器处于 ENABLE 状态时，如果在表上执行 DML 操作，则就会触发相应的触发器。如果基于 INSERT 操作建立了触发器，当使用 SQL\*Loader 装载大批量数据时会触发触发器。为了加快数据装载速度，应该在装载数据之前禁止触发器。方法如下：

```
SQL> ALTER TRIGGER tr_check_sal DISABLE;
```

### 3. 激活触发器

激活触发器是指使触发器重新生效。当使用 SQL\*Loader 装载了数据之后，为了使被禁止的触发器生效，应该激活触发器。方法如下：

```
SQL> ALTER TRIGGER tr_check_sal ENABLE;
```

### 4. 禁止或激活表的所有触发器

如果在表上同时存在多个触发器，那么使用 ALTER TABLE 命令可以一次禁止或激活所有触发器，示例如下：

```
SQL> ALTER TABLE emp DISABLE ALL TRIGGERS;
SQL> ALTER TABLE emp ENABLE ALL TRIGGERS;
```

### 5. 重新编译触发器

当使用 ALTER TABLE 命令修改表结构（例如增加列、删除列）时，会使得其触发器转变为 INVALID 状态。在这种情况下，为了使得触发器继续生效，需要重新编译触发器。示例如下：

```
SQL> ALTER TRIGGER tr_check_sal COMPILE;
```

### 6. 删除触发器

当触发器不再需要时，可以使用 DROP TRIGGER 命令删除触发器。注意，在表上的触发器越多，对于 DML 操作的性能影响也越大，所以一定要适度使用触发器。删除触发器的示例如下：

```
SQL> DROP TRIGGER tr_check_sal;
```

## 13.6 习题

1. 在触发器执行代码中可以包含以下哪些 SQL 语句？
  - A. SELECT 语句
  - B. DML 语句
  - C. DDL 语句
  - D. 事务控制语句
2. 当建立 DML 触发器时，为了审计数据库数据的变化，应该建立哪种类型的触发器？
  - A. BEFORE 语句触发器
  - B. AFTER 语句触发器
  - C. BEFORE 行触发器
  - D. AFTER 行触发器
3. 当在复杂视图上执行 UPDATE 操作时，应该建立以下哪种触发器？
  - A. BEFORE 语句触发器
  - B. AFTER 语句触发器
  - C. BEFORE 行触发器
  - D. BEFORE 语句触发器
  - E. INSTEAD-OF 触发器
4. SCOTT 用户可以建立以下哪些类型的触发器？
  - A. DML 触发器
  - B. INSTEAD-OF 触发器
  - C. 例程启动触发器
  - D. 登录触发器
5. 在 ORD 表上针对 INSERT 操作建立触发器 tr\_add\_ord，并实现以下商业规则：
  - 如果在星期六、星期天增加订单，那么显示自定义错误消息“ORA-20001：只能在工作日增加订单”；
  - 如果在 9:00—17:00 之外的其他时间增加订单，则显示自定义错误消息“ORA-20002：只能在工作时间增加订单”；
6. 在 ORD 表上针对 UPDATE 操作建立级联更新触发器 tr\_upd\_ordid，并实现以下商业规则：
  - 首先使用如下语句建立审计表 AUDIT\_ORDID\_CHANGE：
 

```
CREATE TABLE audit_ordid_change(
            old_ordid NUMBER(4), new_ordid NUMBER(4), changetime DATE
        );
```
  - 更新 ord\_id 列时，为审计表 audit\_ordid\_change 插入数据；
  - 级联更新 ITEM 表的相关 ord\_id 列数据。
7. 在视图 ORD\_ITEM 上基于 UPDATE 操作建立 INSTEAD-OF 触发器 tr\_ord\_item，并实现以下商业规则：

- 首先使用如下语句建立复杂视图 ord\_item:

```
CREATE OR REPLACE VIEW ord_item AS
    SELECT a.ord_id,b.item_id,b.actual_price,
           b.quantity FROM ord a,item b
    WHERE a.ord_id=b.ord_id;
```

- 根据订单号和条款号更新单价和数量，并更新条款总价和订单总价。

## 第 14 章 开发动态 SQL

当编写 PL/SQL 块时，直接嵌入 SQL 语句和 PL/SQL 语句只能完成一些相对固定的任务。假定已建立存储过程 `update_sal`，该过程接收两个输入参数 `eno` 和 `salary`，并在该过程体内使用 SQL 语句“`UPDATE emp SET sal=salary WHERE empno=eno`”来根据雇员号修改雇员工资。这种 SQL 语句在建立过程时已经完成了编译工作，这种 SQL 语句被称为静态 SQL。但是，一些 PL/SQL 程序要求必须在运行时建立和处理 SQL 语句（例如根据不同 `SELECT` 语句输出不同报表），这种 SQL 语句只有在执行时才能确定，所以被称为动态 SQL 语句。本章将介绍如何在 PL/SQL 中开发动态 SQL，在学习了本章之后，读者应该能够：

- 区分静态 SQL 和动态 SQL；
- 学会使用动态 SQL 处理非查询语句；
- 学会使用动态 SQL 处理多行查询语句；
- 学会使用 Oracle9i 动态 SQL 新特征——使用集合处理动态 SQL 数据。

### 14.1 动态 SQL 简介

当编写 PL/SQL 程序时，如果 PL/SQL 块只是用于完成某些特定的功能，那么可以使用静态 SQL；如果 PL/SQL 块需要灵活地处理各种数据操作，那么应该使用动态 SQL。下面介绍静态 SQL 和动态 SQL 的区别及特征。

#### 1. 静态 SQL

静态 SQL 是指直接嵌入在 PL/SQL 块中的 SQL 语句。在编写 PL/SQL 块时，静态 SQL 用于完成特定的或固定的任务。示例如下：

```
SELECT ename,sal INTO v_ename,v_sal FROM emp WHERE empno=v_empno;
INSERT INTO emp (empno,ename,sal) VALUES(v_empno,v_ename,v_sal);
UPDATE emp SET sal=v_sal WHERE ename=v_ename;
DELETE FROM emp WHERE empno=v_empno;
```

上例所示的四条语句都属于静态 SQL 语句，它们用于完成特定的任务。其中，第一条 SQL 语句（`SELECT`）用于检索特定雇员的姓名和工资，第二条 SQL 语句（`INSERT`）用于增加雇员，第三条 SQL 语句（`UPDATE`）用于更新雇员工资，第四条 SQL 语句（`DELETE`）用于删除雇员记录。

#### 2. 动态 SQL

动态 SQL 是指在运行 PL/SQL 块时动态输入的 SQL 语句。如果在 PL/SQL 块中需要执行 DDL 语句（例如 `CREATE`, `ALTER`, `DROP` 语句）、DCL 语句（`GRANT`, `REVOKE`），或者在 PL/SQL 块中需要执行更加灵活的 SQL 语句（例如在 `SELECT` 语句中使用不同的 `WHERE` 条件），那么就必须使用动态 SQL。在 PL/SQL 块中编写动态 SQL 语句时，需要将 SQL 语句存放到字符串变量中，而且 SQL 语句可以包含占位符（以冒号开始）。在 PL/SQL 块中的动态

SQL语句示例如下：

```
CREATE TABLE temp(cola INT,colb VARCHAR2(10))
GRANT SELECT ON emp TO smith
DELETE FROM emp WHERE sal>:a
SELECT ename,sal FROM emp WHERE empno=:l
```

如上所示，第一条语句是DDL语句，第二条语句为DCL语句，在PL/SQL块中只能使用动态SQL执行这两种语句；第三条语句和第四条语句都带有占位符，在PL/SQL块中用于动态接收输入数据。

### 3. 静态SQL和动态SQL的比较

- 静态SQL是在编写PL/SQL块时直接嵌入的SQL语句；而动态SQL是在运行PL/SQL块时动态输入的SQL语句。
- 静态SQL性能要优于动态SQL，因此当编写PL/SQL块时，如果功能完全确定，则使用静态SQL；如果不能确定要执行的SQL语句，则使用动态SQL。

### 4. 动态SQL的处理方法

当编写动态SQL程序时，根据要处理SQL语句的不同，可以使用三种不同类型的动态SQL方法。

#### (1) 使用EXECUTE IMMEDIATE语句

EXECUTE IMMEDIATE语句可以处理多数动态SQL操作，包括DDL语句(CREATE, ALTER, DROP)、DCL语句(GRANT, REVOKE)、DML语句(INSERT, UPDATE, DELETE)，以及单行SELECT语句。注意，EXECUTE IMMEDIATE语句不能用于处理多行查询语句。

#### (2) 使用OPEN-FOR, FETCH和CLOSE语句

为了处理动态的多行查询操作，必须要使用OPEN-FOR语句打开游标，使用FETCH语句循环提取数据，最终使用CLOSE语句关闭游标。

#### (3) 使用批量动态SQL

批量动态SQL是Oracle9i新增加的特征。通过使用批量动态SQL，可以加快SQL语句处理，进而提高PL/SQL程序的性能。

## 14.2 处理非查询语句

为了动态地处理非查询语句(例如DDL, DCL和DML语句)，可以在PL/SQL块中使用EXECUTE IMMEDIATE语句。注意，除了可以处理动态DDL, DCL和DML语句之外，它还可以用于处理单行查询语句。使用该语句的语法如下：

```
EXECUTE IMMEDIATE dynamic_string
[ INTO {define_variable[, define_variable]... | record}]
[ USING [IN | OUT | IN OUT] bind_argument
[, [IN | OUT | IN OUT] bind_argument]...]
[ {RETURNING | RETURN} INTO bind_argument[, bind_argument]...];
```

如上所示，dynamic\_string用于指定存放SQL语句或PL/SQL块的字符串变量；define\_variable用于指定存放单行查询结果的变量；输入bind\_argument(IN)用于指定存放被传递给动态SQL值的变量；输出bind\_argument(OUT)用于指定存放动态SQL返回值的变

量。下面通过示例说明如何在 PL/SQL 块中使用 EXECUTE IMMEDIATE 语句执行动态 SQL 操作。

### 1. 使用 EXECUTE IMMEDIATE 处理 DDL 操作

当在 PL/SQL 块中处理 DDL 操作时, EXECUTE IMMEDIATE 后面只需要带有 DDL 语句文本即可, 而不需要 INTO 和 USING 子句。下面以建立用于删除表的过程为例, 说明处理 DDL 操作的方法。示例如下:

```
CREATE OR REPLACE PROCEDURE drop_table(table_name VARCHAR2)
IS
    sql_statement VARCHAR2(100);
BEGIN
    sql_statement:='DROP TABLE '||table_name;
    EXECUTE IMMEDIATE sql_statement;
END;
/

```

当执行以上命令建立了过程 drop\_table 之后, 就可以调用该过程动态地删除表了。示例如下:

```
SQL> exec drop_table('worker')
```

### 2. 使用 EXECUTE IMMEDIATE 处理 DCL 操作

当在 PL/SQL 块中处理 DCL 操作时, EXECUTE IMMEDIATE 后面只需要带有 DCL 语句文本即可, 而不需要 INTO 和 USING 子句。下面以建立为用户授予系统权限的过程为例, 说明处理 DCL 操作的方法。示例如下:

```
SQL> conn system/manager
CREATE OR REPLACE PROCEDURE grant_sys_priv
(priv VARCHAR2,username VARCHAR2)
IS
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='GRANT ''||priv||' TO '||username;
    EXECUTE IMMEDIATE sql_stat;
END;
/

```

当建立了过程 grant\_sys\_priv 之后, 就可以调用该过程为其他用户或角色授予系统权限了。示例如下:

```
SQL> exec grant_sys_priv('CREATE SESSION','SCOTT')
```

### 3. 使用 EXECUTE IMMEDIATE 处理 DML 操作

当使用 EXECUTE IMMEDIATE 处理 DML 语句时, 如果 DML 语句既没有占位符, 也没有 RETURNING 子句, 那么在 EXECUTE IMMEDIATE 语句之后不需要带有 USING 和 RETURNING INTO 子句; 如果 DML 语句包含有占位符, 那么在 EXECUTE IMMEDIATE 语句之后必须要带有 USING 子句; 如果 DML 语句带有 RETURNING 子句, 那么在 EXECUTE IMMEDIATE 语句之后需要带有 RETURNING INTO 子句。下面分别说明在这三种情况下的处理方法。

#### (1) 处理无占位符和 RETURNING 子句的 DML 语句

当使用 EXECUTE IMMEDIATE 处理无占位符和 RETURNING 子句的 DML 语句时，不需要带有 USING 和 INTO 子句。下面以给部门 30 的所有雇员增加 10% 的工资为例，说明这种处理方法。示例如下：

```
DECLARE
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='UPDATE emp SET sal=sal*1.1 WHERE deptno=30';
    EXECUTE IMMEDIATE sql_stat;
END;
/
```

### (2) 处理包含占位符的 DML 语句

当使用 EXECUTE IMMEDIATE 处理包含占位符的 DML 语句时，需要使用 USING 子句为占位符提供输入数据。下面以给不同部门增加工资为例，说明这种处理方法。示例如下：

```
DECLARE
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='UPDATE emp SET sal=sal*(1+:percent/100)'
        ||' WHERE deptno=:dno';
    EXECUTE IMMEDIATE sql_stat USING &1,&2;
END;
/
输入 1 的值： 15
输入 2 的值： 20
```

在执行了以上 PL/SQL 块之后，会为部门 20 的每个雇员增加 15% 的工资，其中 USING 后的第一个值是为第一个占位符提供的，第二个值是为第二个占位符提供的。注意，占位符可以使用任意的名称。

### (3) 处理包含 RETURNING 子句的 DML 语句

当使用 EXECUTE IMMEDIATE 处理包含 RETURNING 子句的 DML 语句时，必须要使用 RETURNING INTO 子句接收返回数据。注意，当直接使用 EXECUTE IMMEDIATE 处理带有 RETURNING 子句的 DML 语句时，只能处理作用在单行上的 DML 语句。如果 DML 语句作用在多行上，则必须要使用 BULK 子句。下面以给特定雇员增加工资，并输出新工资为例，说明处理带有 RETURNING 子句的动态 DML 语句的方法。示例如下：

```
DECLARE
    salary NUMBER(6,2);
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='UPDATE emp SET sal=sal*(1+:percent/100)'
        ||' WHERE empno=:eno RETURNING sal INTO :salary';
    EXECUTE IMMEDIATE sql_stat USING &1,&2
        RETURNING INTO salary;
    dbms_output.put_line('新工资:'||salary);
END;
/
```

```
输入 1 的值: 15
输入 2 的值: 7788
新工资:3450
```

当执行了以上 PL/SQL 块之后，会根据输入的工资百分比和雇员号修改特定雇员的工资，并显示修改后的工资结果。

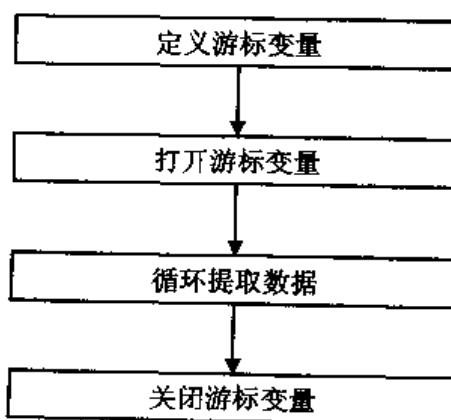
#### 4. 使用 EXECUTE IMMEDIATE 处理单行查询

使用 EXECUTE IMMEDIATE 不仅可以处理 DDL, DCL 和 DML 语句，而且还可以用于处理单行查询语句。但在使用 EXECUTE IMMEDIATE 语句处理单行查询语句时，需要使用 INTO 子句接收返回数据。下面以输出特定雇员的雇员姓名和工资为例，说明使用 EXECUTE IMMEDIATE 处理单行查询语句的方法。示例如下：

```
DECLARE
    sql_stat VARCHAR2(100);
    emp_record emp%ROWTYPE;
BEGIN
    sql_stat:='SELECT * FROM emp WHERE empno=:eno';
    EXECUTE IMMEDIATE sql_stat INTO emp_record USING &1;
    dbms_output.put_line('雇员'||emp_record.ename||
        '的工资为'||emp_record.sal);
END;
/
输入 1 的值: 7369
原值    7:      INTO emp_record USING &1;
新值    7:      INTO emp_record USING 7369;
雇员 SMITH 的工资为 800
```

### 14.3 处理多行查询语句

使用 EXECUTE IMMEDIATE 只能处理单行查询语句，为了动态地处理 SELECT 语句所返回的多行数据，需要使用 OPEN-FOR, FETCH 和 CLOSE 语句。动态地处理 SELECT 语句返回的多行数据步骤如下图所示。



#### 1. 定义游标变量

因为动态地处理多行查询需要使用游标变量来完成，所以首先需要在定义部分定义游标

变量。定义游标变量的语法如下：

```
TYPE cursortype IS REF CURSOR;
cursor_variable cursortype;
```

如上所示，cursortype 用于指定 REF CURSOR 类型，而 cursor\_variable 则用于指定游标变量名。

## 2. 打开游标变量

在定义了游标变量之后，就可以在执行部分使用 OPEN-FOR 语句打开游标变量。当打开游标变量时，会执行游标变量所对应的 SELECT 语句，并将查询结果存放到游标结果集中。打开游标变量的语法如下：

```
OPEN cursor_variable FOR dynamic_string
[USING bind_argument[, bind_argument]...];
```

如上所示，dynamic\_string 是动态的 SELECT 语句，bind\_argument 用于指定存放传递给动态 SELECT 语句值的变量。

## 3. 循环提取数据

在打开游标变量之后，查询结果已经被存放到游标结果集中。为了取得游标结果集中的所有数据，需要使用 FETCH 语句提取数据。默认情况下，每条 FETCH 语句只能提取一行数据，为了提取结果集的所有数据，需要使用循环语句。语法如下：

```
FETCH cursor_variable INTO {var1[,var2]... | record_var};
```

如上所示，var 是用于接收提取结果的变量；record\_var 是用于接收提取结果的记录变量。

## 4. 关闭游标变量

在循环提取了游标结果集的所有数据之后，就可以使用 CLOSE 语句关闭游标变量。关闭游标变量之后，将释放游标结果集的所有数据。关闭游标变量的语法如下：

```
CLOSE cursor_variable;
```

## 5. 多行查询示例

下面以动态输入部门号，显示特定部门的所有雇员姓名和工资为例，说明使用动态 SQL 处理多行查询的方法。示例如下：

```
DECLARE
    TYPE empcurtyp IS REF CURSOR;
    emp_cv empcurtyp;
    emp_record emp%ROWTYPE;
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='SELECT * FROM emp WHERE deptno=:dno';
    OPEN emp_cv FOR sql_stat USING &dno;
    LOOP
        FETCH emp_cv INTO emp_record;
        EXIT WHEN emp_cv%NOTFOUND;
        dbms_output.put_line('雇员名:'||emp_record.ename
            ||',工资:'||emp_record.sal);
    END LOOP;
    CLOSE emp_cv;
END;
```

```

/
输入 dno 的值: 10
雇员名:CLARK, 工资:2450
雇员名:KING, 工资:5000
雇员名:MILLER, 工资:1300

```

在执行了以上 PL/SQL 块之后，会根据输入的部门号 10 返回该部门的所有雇员名及其工资。

## 14.4 在动态 SQL 中使用 BULK 子句

在动态 SQL 中使用 BULK 子句是 Oracle9i 新增加的特征。通过使用 BULK 子句，可以加快批量数据的处理速度，从而提高应用程序的性能。当使用 BULK 子句时，实际是动态 SQL 语句将变量绑定为集合元素。注意，当使用 BULK 子句时，集合类型可以是 PL/SQL 所支持的索引表、嵌套表和 VARRAY，但集合元素必须使用 SQL 数据类型（例如 NUMBER、CHAR 等），而不能使用 PL/SQL 数据类型（例如 BINARY\_INTEGER、BOOLEAN 等）。在 Oracle9i 中，有三种语句支持 BULK 子句：EXECUTE IMMEDIATE，FETCH 和 FORALL，下面分别介绍在这三种语句中使用 BULK 子句的方法。

### 1. 在 EXECUTE IMMEDIATE 语句中使用动态 BULK 子句

EXECUTE IMMEDIATE 语句用于处理动态 DDL，DCL，DML 语句以及单行查询语句。但是我们注意到，当直接使用 EXECUTE IMMEDIATE 语句时，不能处理作用在多行上的动态 DML 返回子句。而通过引入 BULK 子句，不仅可以处理作用在多行上的动态 DML 返回子句，而且还可以处理多行查询语句。在 EXECUTE IMMEDIATE 语句中使用动态 BULK 子句的语法如下：

```

EXECUTE IMMEDIATE dynamic_string
[BULK COLLECT INTO define_variable[, define_variable ...]]
[USING bind_argument[, bind_argument ...]]
[(RETURNING | RETURN)
BULK COLLECT INTO return_variable[, return_variable ...]);

```

如上所示，dynamic\_string 用于指定存放动态 SQL 语句的字符串变量；define\_variable 用于指定存放查询结果的集合变量；bind\_argument 用于指定绑定变量（存放传递给动态 SQL 的数据）；return\_variable 用于指定接收 RETURNING 子句返回结果的集合变量。下面通过示例说明在 EXECUTE IMMEDIATE 语句中使用 BULK 子句的方法。

#### (1) 使用 BULK 子句处理 DML 语句返回子句

当处理作用在多行上的动态 DML 返回子句时，必须在 EXECUTE IMMEDIATE 语句后带有 BULK 子句。下面以动态修改特定部门的工资，并返回修改后的雇员名及其工资为例，说明使用 BULK 子句处理动态 DML 语句的方法。示例如下：

```

DECLARE
  TYPE ename_table_type IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE sal_table_type IS TABLE OF emp.sal%TYPE
    INDEX BY BINARY_INTEGER;

```

```

ename_table ename_table_type;
sal_table sal_table_type;
sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='UPDATE emp SET sal=sal*(1+:percent/100)'
    ||' WHERE deptno=:dno'
    ||' RETURNING ename,sal INTO :name,:salary';
EXECUTE IMMEDIATE sql_stat USING &percent,&dno
    RETURNING BULK COLLECT INTO ename_table,sal_table;
FOR i IN 1..ename_table.COUNT LOOP
    dbms_output.put_line('雇员'||ename_table(i)
    ||'的新工资为'||sal_table(i));
END LOOP;
END;
/
输入 percent 的值: 15
输入 dno 的值: 20
雇员 SMITH 的新工资为 920
雇员 JONES 的新工资为 3421.25
雇员 SCOTT 的新工资为 3450
雇员 ADAMS 的新工资为 1265
雇员 FORD 的新工资为 3450

```

在执行了以上 PL/SQL 块之后，会为部门 20 的所有雇员增加 15% 的工资，并输出其新工资值。

## (2) 使用 BULK 子句处理多行查询

通过在 EXECUTE IMMEDIATE 语句中引入 BULK 子句，不仅使得该语句可以处理单行查询，而且也可以处理多行查询语句。下面以动态显示特定部门的所有雇员名为例，说明在 EXECUTE IMMEDIATE 语句中使用 BULK 子句处理多行查询的方法。示例如下：

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='SELECT ename FROM emp WHERE deptno=:dno';
    EXECUTE IMMEDIATE sql_stat
        BULK COLLECT INTO ename_table USING &dno;
    FOR i IN 1..ename_table.COUNT LOOP
        dbms_output.put_line(ename_table(i));
    END LOOP;
END;
/
输入 dno 的值: 10
CLARK
KING

```

MILLER

### 2. 在 FETCH 语句中使用 BULK 子句

当使用 OPEN-FOR, FETCH, CLOSE 语句处理动态的多行查询语句时, 因为在默认情况下 FETCH 语句每次只能提取单行数据, 所以为了处理所有数据, 需要使用循环语句。通过在 FETCH 语句中引入 BULK 子句, 一次就可以提取所有数据。在 FETCH 语句中使用 BULK 子句的语法如下:

```
FETCH dynamic_cursor
  BULK COLLECT INTO define_variable[, define_variable ...];
```

如上所示, dynamic\_cursor 是已经打开的游标变量。下面以动态返回特定岗位的雇员名为例, 说明在 FETCH 语句中使用 BULK 子句的方法。示例如下:

```
DECLARE
    TYPE empcurtyp IS REF CURSOR;
    emp_cv empcurtyp;
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    ename_table ename_table_type;
    sql_stat VARCHAR2(100);
BEGIN
    sql_stat:='SELECT ename FROM emp WHERE job=:title';
    OPEN emp_cv FOR sql_stat USING '&job';
    FETCH emp_cv BULK COLLECT INTO ename_table;
    FOR i IN 1..ename_table.COUNT LOOP
        dbms_output.put_line(ename_table(i));
    END LOOP;
    CLOSE emp_cv;
END;
/
输入 job 的值: CLERK
SMITH
ADAMS
JAMES
MILLER
```

### 3. 在 FORALL 语句中使用 BULK 子句

使用 FORALL 语句, 可允许在动态 SQL 语句中为输入变量同时提供多个数据, 但 FORALL 语句只适用于动态的 INSERT, UPDATE 和 DELETE 语句, 而不适用于动态的 SELECT 语句, 并且 FORALL 语句和 EXECUTE IMMEDIATE 语句是结合使用的。语法如下:

```
FORALL index IN lower bound..upper bound
    EXECUTE IMMEDIATE dynamic_string
    USING bind_argument | bind_argument(index)
        [, bind_argument | bind_argument(index)] ...
    [{RETURNING | RETURN} BULK COLLECT
        INTO bind_argument[, bind_argument ... ]];
```

下面以修改多个雇员工资, 并返回雇员新工资为例, 说明结合使用 FORALL 语句和 EXECUTE IMMEDIATE 语句的方法。示例如下:

```

DECLARE
    TYPE ename_table_type IS TABLE OF emp.ename%TYPE;
    TYPE sal_table_type IS TABLE OF emp.sal%TYPE;
    ename_table ename_table_type;
    sal_table sal_table_type;
    sql_stat VARCHAR2(100);
BEGIN
    ename_table:=ename_table_type('SCOTT','SMITH','CLARK');
    sql_stat:='UPDATE emp SET sal=sal*1.1 WHERE ename=:1'
        ||' RETURNING sal INTO :2';
    FORALL i IN 1..ename_table.COUNT
        EXECUTE IMMEDIATE sql_stat USING ename_table(i)
            RETURNING BULK COLLECT INTO sal_table;
    FOR j IN 1..ename_table.COUNT LOOP
        dbms_output.put_line('雇员'||ename_table(j)
            ||'的新工资为'||sal_table(j));
    END LOOP;
END;
/
雇员 SCOTT 的新工资为 3795
雇员 SMITH 的新工资为 1012
雇员 CLARK 的新工资为 2695

```

当执行了以上 PL/SQL 块之后，会为雇员 SCOTT, SMITH 和 CLARK 增加 10% 的工资，并输出它们的新工资。

## 14.5 习题

1. 当在 PL/SQL 块中编写静态 SQL 语句时，PL/SQL 块可以处理以下哪些 SQL 语句？
  - A. DDL 语句
  - B. DCL 语句
  - C. SELECT 语句
  - D. DML 语句
  - E. 事务控制语句
2. 以下哪些 SQL 语句必须要使用动态 SQL 进行处理？
  - A. DDL 语句
  - B. DCL 语句
  - C. SELECT 语句
  - D. DML 语句
  - E. 事务控制语句
3. 以 SYS 用户登录，建立用于执行 DDL 操作的过程 exec\_ddl，并执行该过程。
4. 编写 PL/SQL 块，使用动态 SQL 根据订单号和条款号更新产品单价、产品数量。格式如下：
 

输入 price 的值: 40  
   输入 quantity 的值: 10  
   输入 ordid 的值: 600  
   输入 itemid 的值: 1  
   PL/SQL 过程已成功完成。
5. 编写 PL/SQL 块，使用动态 SQL 根据订单号返回条款号以及条款总价。格式如下：

输入 ordid 的值: 600

条款号:1,条款总价:42

条款号:2,条款总价:58

PL/SQL 过程已成功完成。

6. 编写 PL/SQL 块, 在动态 SQL 中使用 BULK 子句根据输入的多个订单号更新交付日期为当前日期, 并返回每个订单对应的客户号。格式如下:

输入 1 的值: 600

输入 2 的值: 601

输入 3 的值: 602

订单:600,客户:221

订单:601,客户:217

订单:602,客户:218

PL/SQL 过程已成功完成。

# 第 15 章 使用对象类型

随着计算机软件技术的不断发展，面向对象的程序设计思想正在逐步取代原有的面向过程的程序设计思想。面向对象程序设计（OOP）可以大大降低建立复杂应用的开销时间，所以已经被应用开发人员广泛接受并付诸实践。在 PL/SQL 中，面向对象的程序设计是基于对象类型来完成的。本章将详细介绍如何建立和使用对象类型，在学习了本章之后，读者应该能够：

- 了解对象类型及其组成部分的作用；
- 学会建立简单对象类型，并掌握使用简单对象类型的方法；
- 学会建立复杂对象类型，并掌握使用复杂对象类型的方法；
- 学会建立参照对象类型（REF），并掌握使用参照对象类型的方法；
- 学会使用 Oracle9i 的新特征——类型继承和自定义构造方法。

## 15.1 对象类型简介

对象类型是用户自定义的一种复合数据类型，它封装了数据结构和用于操纵这些数据结构的过程和函数。在建立复杂应用程序时，通过使用对象类型可以降低应用开发难度，进而提高应用开发的效率和速度。

### 1. 对象类型组件

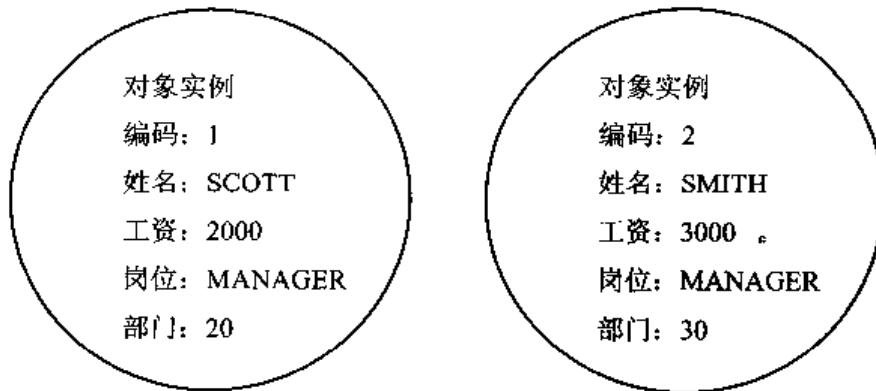
对象类型包括属性和方法，其中，属性（Attribute）用于描述对象所具有的特征，而方法（Method）则用于实现对象所执行的操作。例如，雇员对象类型 `employee_type` 应该具有编码、姓名、工资、岗位以及部门号等属性，还应该具有用于调整岗位、调整工资和调整部门的方法。如下图所示：

对象类型 <code>employee_type</code>			
属性:		方法:	
<code>eno</code>	编码	<code>change_job</code>	调整岗位
<code>name</code>	姓名	<code>change_sal</code>	调整工资
<code>salary</code>	工资	<code>change_dept</code>	调整部门
<code>job</code>	岗位		
<code>dno</code>	部门号		

### 2. 对象类型和对象实例

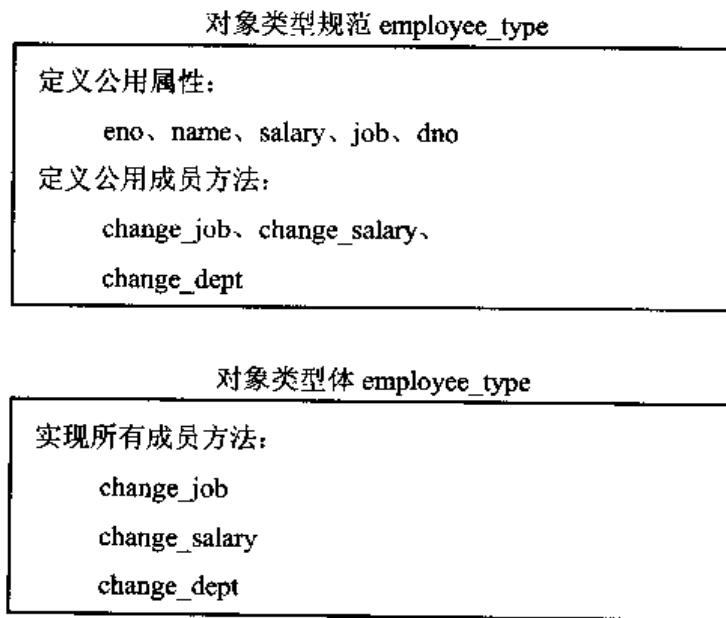
对象类型（Object Type）是为了描述现实世界对象所抽象出来的具体特征，应该涵盖对象所具有的公共特性。例如，每个雇员都有编码、姓名、工资、岗位和部门号等特征，所以对象类型 `employee_type` 应该包含这些特征的描述。对象实例（Object Instance）是对对象类型的具

体实现，对应于现实世界的具体对象。例如，雇员 SCOTT 和 SMITH 是对象类型 `employee_type` 的两个对象实例。如下图所示：



### 3. 构造对象类型

对象类型包括对象类型规范 (Object Type Specification) 和对象类型体 (Object Type Body) 两个部分。其中，对象类型规范是对象与应用的接口，它用于定义对象的公用属性和方法；而对象类型体则用于实现对象类型规范所定义的公用方法。例如，当建立对象类型 `employee_type` 时，首先需要建立对象类型规范，其次需要通过对象类型体实现公用的对象方法。如下图所示：



### 4. 对象类型属性

对象类型属性用于描述对象所具有的特征，例如对象类型 `employee_type` 具有 `eno`, `name`, `salary` 等属性，这些属性分别用于描述雇员编码、姓名和工资。对象类型最少要包含一个属性，最多可以包含 1000 个属性。当定义对象类型属性时，必须要提供属性名和数据类型，对象类属性可以使用多数 Oracle 数据类型，但不能使用以下数据类型：

- LONG 和 LONG RAW;
- ROWID 和 UROWID;
- PL/SQL 特有类型（例如 BINARY\_INTEGER, BOOLEAN, %TYPE, %ROWTYPE,

REF CURSOR, RECORD, PLS\_INTEGER 等)。

注意，在定义对象类型属性时，既不能指定对象属性的默认值，也不能指定 NOT NULL 选项。

### 5. 对象类型方法

对象类型方法用于描述对象所要执行的操作。当定义对象类型时，既可以定义方法，也可以不定义方法。在定义方法时，可以定义构造方法、 MEMBER 方法、 STATIC 方法、 MAP 方法以及 ORDER 方法。

#### (1) 构造方法 (Construct Method)

构造方法用于初始化对象并返回对象实例，其作用类似于 Java 语言的构造方法。构造方法是与对象类型同名的函数，其默认参数是对象类型的所有属性，例如对象类型 employee\_type 的构造方法为 employee\_type()。在 Oracle9i 之前，开发人员只能使用系统默认的构造方法；而从 Oracle9i 开始，允许开发人员自定义构造方法。

#### (2) MEMBER 方法

MEMBER 方法用于访问对象实例的数据。如果在对象类型中需要访问特定对象实例的数据，则需要定义 MEMBER 方法。当使用 MEMBER 方法时，可以使用内置参数 SELF 访问当前对象实例。当定义 MEMBER 方法时，无论是否定义 SELF 参数，它都会被作为第一个参数传递给 MEMBER 方法。但如果要定义参数 SELF，那么其类型必须要使用当前对象类型。注意， MEMBER 方法只能由对象实例调用，而不能由对象类型调用。调用 MEMBER 方法的示例如下：

```
object_type_instance.method()
```

#### (3) STATIC 方法

STATIC 方法用于访问对象类型，它用于在对象类型上执行全局操作，而不需要访问特定对象实例的数据，因此 STATIC 方法不能引用 SELF 参数。注意， STATIC 方法只能由对象类型调用，而不能由对象实例调用。调用 STATIC 方法的示例如下：

```
Object_type.method()
```

#### (4) MAP 方法

对于标量变量来说，其数据可以直接进行比较，例如 NUMBER 类型数据可以按照大小进行排序，而 VARCHAR2 类型数据可以按照字符的 ASCII 码值进行排序；但对象类型的数据不能直接进行比较，例如对象类型 employee\_type 具有多个属性，每个属性又具有特定的数据类型，所以无法直接进行比较。为了按照特定规则排序对象实例的数据，可以在对象类型上定义 MAP 方法。MAP 方法是对对象类型的一种可选方法，它可以将对象实例映射为标量类型数据 (DATE, NUMBER 或 VARCHAR2)，然后根据该标量类型数据可以排序对象实例。但注意，对象类型最多只能定义一个 MAP 方法。

#### (5) ORDER 方法

MAP 方法可以在多个对象实例之间进行排序，而 ORDER 方法只能比较两个对象实例的大小。注意，定义对象类型时，最多只能定义一个 ORDER 方法，而且 MAP 方法和 ORDER 方法不能同时定义。使用 MAP 方法和 ORDER 方法的原则如下：

- 如果应用程序不需要比较对象实例，则不需要定义 MAP 和 ORDER 方法。
- 如果应用程序需要比较多个对象实例，则选择 MAP 方法。

- 如果应用程序只需要比较两个对象实例，则选择 ORDER 方法。

## 6. 对象表 (Object Table)

对象表是指包含对象类型列的表。对于普通表而言，其列全部使用标量数据类型（例如 NUMBER, VARCHAR2, DATE, LONG, LOB 等）；而对象表至少会包含一个对象类型列（例如 employee\_type 类型）。Oracle 又包含行对象和列对象两种对象表，其中行对象是指直接基于对象类型所建立的表，而列对象则是指包含多个列的对象表：

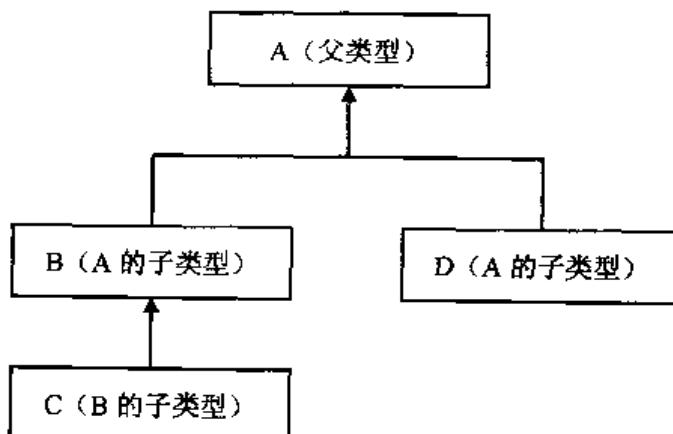
行对象：CREATE TABLE employee OF employee\_type;

列对象：

```
CREATE TABLE department(
    dno NUMBER, dname VARCHAR2(10),
    employee employee_type
);
;
```

## 7. 对象类型继承 (Type Inheritance)

对象类型继承是 Oracle9i 新增加的特征，是指一个对象类型继承另一个对象类型。对象类型继承由父类型 (Supertype) 和子类型 (Subtype) 组成，其中父类型用于定义不同对象类型的公用属性和方法，而子类型不仅继承了父类型的公用属性和方法，而且还可具有自己的私有属性和方法。如下图所示：



上图中示意性地给出了不同对象类型之间的关系，类型 B 和 D 是类型 A 的子类型，它们具有类型 A 的所有属性和方法，而且还具有自己的私有属性和方法；类型 C 是类型 B 的子类型，它具有类型 B 的所有属性和方法，并且还具有自己的私有属性和方法。

## 8. REF 数据类型

REF 是指向行对象的逻辑指针，是 Oracle 的一种内置数据类型。建表时通过使用 REF 引用行对象，可以使不同表共享相同对象，从而降低内存占用，并提高了应用性能。假定基于对象类型 employee\_type 建立了行对象 employee，那么在建立 department 表时为了引用行对象 employee 中的数据，可以使用 REF 数据类型。示例如下：

```
CREATE TABLE department(
    dno NUMBER(2), dname VARCHAR2(10), emp REF employee_type);
```

## 15.2 建立和使用简单对象类型

对象类型包括对象类型规范和对象类型体两部分。建立对象类型时，需要首先建立对象类型规范，然后建立对象类型体。但是，如果在建立对象类型规范时没有定义方法，则不需要建立对象类型体。建立对象类型规范的语法如下：

```
CREATE OR REPLACE TYPE type_name AS OBJECT (
    attribute1 datatype[,attribute2 datatype,...],
    [MEMBER|STATIC method1 spec, MEMBER|STATIC method2 spec,...]);
```

如上所示，`type_name` 用于指定对象类型的名称；`attribute` 用于指定对象类型属性的名称，`datatype` 用于指定对象类型属性的数据类型，`method` 用于指定对象类型方法的名称。当建立对象类型时如果定义了对象方法，那么为了使用该对象类型的方法，必须要建立对象类型体，建立对象类型体的语法如下：

```
CREATE OR REPLACE TYPE BODY type_name AS
    MEMBER|STATIC method1 body;
    MEMBER|STATIC method2 body;
    . . .]
```

如上所示，`type_name` 是对象类型的名称；`method` 是对象类型方法的名称；而 `body` 则是用 PL/SQL 所编写的方法实现代码。下面说明如何建立对象规范和对象类型体，以及使用对象类型的方法。

### 1. 建立和使用不包含任何方法的对象类型

建立对象类型时，至少要为对象类型定义一个属性，并且对象类型最多可以定义 1000 个属性，但对象类型可以不包含任何方法。下面以建立并使用对象类型 `person_typ1` 为例，说明如何建立和使用不包含对象方法的对象类型。对象类型 `person_typ1` 用于描述人员的基本特征，该对象类型会包含姓名、性别和出生日期等三个属性。如下所示：

对象类型 person_typ1		
属性:		方法:
<code>name</code>	姓名	无任何方法
<code>gender</code>	性别	
<code>birthdate</code>	出生日期	

其中对象类型 `person_typ1` 包含 `name`、`gender`、`birthdate` 等三个对象属性，并且不包含任何方法。建立该对象类型的示例如下：

```
CREATE OR REPLACE TYPE person_typ1 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2), birthdate DATE
);
```

执行了以上命令之后，会建立名称为 `person_typ1` 的对象类型，并且该对象类型包含三个属性：`name`、`gender`、`birthdate`。建立了对象类型之后，就可以使用该对象类型了，下面举例

说明使用该对象类型的方法。

### (1) 建立行对象(基于对象类型 person\_typ1)

行对象是指直接基于对象类型所建立的表，下面以建立并使用对象表 person\_tab1 为例，说明建立并使用行对象的方法。建立对象表 person\_tab1 的示例如下：

```
SQL> CREATE TABLE person_tab1 OF person_typ1;
```

当执行了上述命令之后，会建立行对象 person\_tab1。建立了行对象之后，就可以在 PL/SQL 块内使用该行对象了。下面举例说明在 PL/SQL 块中使用行对象的方法。

#### 示例一：为行对象插入数据

当使用 DESC 命令查看行对象结构时，会发现，其结构与普通表没有任何区别。当为行对象插入数据时，既可以使用对象类型的构造方法，也可以直接使用给普通表插入数据的方法。在 PL/SQL 块中为行对象插入数据的示例如下：

```
BEGIN
    INSERT INTO person_tab1 VALUES ('马丽', '女', '11-1月-76');
    INSERT INTO person_tab1 VALUES
        (person_typ1('王鸣', '男', '12-4月-76'));
END;
/
```

如例所示，第一条 INSERT 语句与为普通表插入数据没有任何区别，而第二条 INSERT 语句则使用了对象类型的构造方法来插入数据。

#### 示例二：检索行对象数据

当在 PL/SQL 块中检索行对象数据时，如果要将数据检索到对象类型变量中，则必须要使用函数 VALUE 取得行对象数据，并检索到对象类型变量中。下面以将 person\_tab1 表的数据检索到对象类型变量中，说明检索行对象数据的方法。示例如下：

```
DECLARE
    person person_typ1;
BEGIN
    SELECT VALUE(p) INTO person FROM person_tab1 p
        WHERE p.name='&name';
    dbms_output.put_line('性别:' || person.gender);
    dbms_output.put_line('出生日期:' || person.birthdate);
END;
/
输入 name 的值： 马丽
性别：女
出生日期：11-1月-76
```

#### 示例三：更新行对象数据

更新行对象数据时如果按照对象属性更新数据，则必须要为行对象定义别名。下面以更新“马丽”的出生日期为例，说明更新行对象数据的方法。示例如下：

```
BEGIN
    UPDATE person_tab1 p SET p.birthdate='11-2月-76'
        WHERE p.name='马丽';
END;
/
```

#### 示例四：删除行对象数据

删除行对象数据时如果按照对象属性删除数据，则必须要为行对象定义别名。下面以删除“马丽”为例，说明删除行对象数据的方法。示例如下：

```
BEGIN
    DELETE FROM person_tab1 p WHERE p.name='马丽';
END;
/
```

#### (2) 建立列对象（包含对象类型 person\_typ1）

列对象是指在建表时指定了对象类型列的对象表。下面以建立并使用对象表 employee\_tab1 为例，说明建立并使用列对象的方法。建立对象表 employee\_tab1 的示例如下：

```
CREATE TABLE employee_tab1(
    eno NUMBER(6), person person_typ1,
    sal NUMBER(6,2), job VARCHAR2(10)
);
```

执行了以上命令之后，会建立对象表 employee\_tab1。因为该表不仅包含了标量类型列 eno，sal 和 job，而且还包含了对象类型列 person，所以被称为列对象。当建立了列对象之后，就可以在 PL/SQL 块中使用该列对象了。下面举例说明在 PL/SQL 块中使用列对象的方法。

#### 示例一：为列对象 employee\_tab1 插入数据

为列对象插入数据时必须要使用对象类型的构造方法为对象列提供数据。下面通过给 employee\_tab1 表插入数据为例，说明给列对象插入数据的方法。示例如下：

```
BEGIN
    INSERT INTO employee_tab1(eno,sal,job,person)
        VALUES(1,2000,'高级钳工',
               person_typ1('王鸣','男','01-8月-76'));
END;
/
```

#### 示例二：检索对象类型列的数据

检索行对象数据到对象类型变量时必须要使用 VALUE 函数；而检索列对象的对象类型列数据时可以直检将对象实例数据检索到对象类型变量中。下面以检索人员信息及其工资为例，说明在列对象中检索对象类型列数据的方法。示例如下：

```
DECLARE
    employee person_typ1;
    salary NUMBER(6,2);
BEGIN
    SELECT person,sal INTO employee,salary
        FROM employee_tab1 WHERE eno=&no;
    dbms_output.put_line('雇员名:'||employee.name);
    dbms_output.put_line('雇员工资:'||salary);
END;
/
输入 no 的值： 1
雇员名：王鸣
雇员工资：2000
```

### 示例三：更新对象列数据

更新列对象的对象列数据时必须要为列对象定义别名，并且引用对象属性（列对象别名.对象类型列名.对象属性名）。下面以更新特定人员的出生日期为例，说明更新对象列数据的方法。示例如下：

```
BEGIN
    UPDATE employee_tabl p SET p.person.birthdate='&newdate'
        WHERE p.person.name='&name';
END;
/
输入 newdate 的值: 12-2月 -75
输入 name 的值: 王鸣
```

### 示例四：依据对象属性删除数据

依据对象属性删除列对象数据时必须要为对象表定义别名，并且引用对象属性（列对象别名.对象类型列名.对象属性名）。下面以删除特定人员为例，说明依据对象属性删除列对象数据的方法。示例如下：

```
BEGIN
    DELETE FROM employee_tabl p WHERE p.person.name='王鸣';
END;
/
```

## 2. 建立和使用包含 MEMBER 方法的对象类型

MEMBER 方法用于访问对象实例的数据，如果在对象类型中需要访问特定对象实例的数据，则必须要定义 MEMBER 方法。注意，MEMBER 方法只能由对象实例调用，而不能由对象类型调用。下面以建立和使用对象类型 person\_typ2 为例，说明如何在对象类型中定义和使用 MEMBER 方法。对象类型 person\_typ2 将用于描述人员特征，该对象类型包含有姓名、性别、出生日期、地址等四个属性，另外还包含有用于改变地址和取得对象信息的两个方法。如下所示：

对象类型 person_typ2	
属性:	方法:
name	姓名
gender	性别
birthdate	出生日期
address	地址
	change_address: 改变地址
	get_info: 取得对象信息

其中，对象类型 person\_typ2 包含 name, gender, birthdate 和 address 等四个属性，以及一个 MEMBER 过程 change\_address 和一个 MEMBER 函数 get\_info。建立对象类型规范 person\_typ2 的示例如下：

```
CREATE OR REPLACE TYPE person_typ2 AS OBJECT (
    name VARCHAR2(10), gender VARCHAR2(2),
    birthdate DATE, address VARCHAR2(100),
    MEMBER PROCEDURE change_address(new_addr VARCHAR2),
```

```

    MEMBER FUNCTION get_info RETURN VARCHAR2
);
/

```

如上所示，因为在建立对象类型规范 person\_typ2 时定义了 MEMBER 方法，所以必须要通过对象类型体实现这些方法。建立对象类型体 person\_typ2 的示例如下：

```

CREATE OR REPLACE TYPE BODY person_typ2 IS
    MEMBER PROCEDURE change_address(new_addr VARCHAR2)
    IS
    BEGIN
        address:=new_addr;
    END;
    MEMBER FUNCTION get_info RETURN VARCHAR2
    IS
        v_info VARCHAR2(100);
    BEGIN
        v_info:='姓名:'||name||', 出生日期:'||birthdate;
        RETURN v_info;
    END;
END;
/

```

在完成了对象类型 person\_typ2 的创建工作之后，就可以使用该对象类型了。基于 person\_typ2 建立对象表 employee\_tab2，并为其插入数据的示例如下：

```

CREATE TABLE employee_tab2 (
    eno NUMBER(6), person person_typ2,
    sal NUMBER(6,2), job VARCHAR2(10)
);
INSERT INTO employee_tab2(eno,sal,job,person)
    VALUES(1,2000,'高级焊工',
           person_typ2('王鸣','男','11-1月-75','呼和浩特兴安北路 55 号')
);
INSERT INTO employee_tab2(eno,sal,job,person)
    VALUES(2,1500,'质量检查员',
           person_typ2('马丽','女','11-5月-75','呼和浩特兴安南路 20 号')
);

```

在执行了以上语句之后，就会建立对象表 employee\_tab2，并插入两条数据。因为在定义对象类型 person\_tab2 时定义了对象方法，所以可以在 PL/SQL 块中使用其对象方法。下面以调用对象方法 change\_address 改变人员地址为例，说明如何在 PL/SQL 块中使用对象方法。示例如下：

```

DECLARE
    v_person person_typ2;
BEGIN
    SELECT person INTO v_person FROM employee_tab2
    WHERE eno=&eno;
    v_person.change_address('呼和浩特北垣东街 12 号');
    UPDATE employee_tab2 SET person=v_person WHERE eno=&eno;

```

```

    dbms_output.put_line(v_person.get_info);
END;
/
输入 no 的值: 1
姓名:王鸣, 出生日期:11-1月 -75

```

### 3. 建立和使用包含 STATIC 方法的对象类型

STATIC 方法用于访问对象类型。如果需要在对象类型上执行全局操作，则应该定义 STATIC 方法。注意，STATIC 方法只能由对象类型访问，而不能由对象实例访问。下面以建立和使用对象类型 person\_typ3 为例，说明如何在对象类型中定义和使用 STATIC 方法。对象类型 person\_typ3 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期、注册日期等四个属性，另外还包含有用于取得当前日期和取得对象信息的两个方法。如下所示：

对象类型 person_typ3	
属性:	方法:
name 姓名	getdate: 取得当前日期
gender 性别	get_info: 取得对象信息
birthdate 出生日期	
regdate 注册日期	

其中，对象类型 person\_typ3 包含 name, gender, birthdate 和 regdate 等四个属性，以及一个 STATIC 方法 getdate 和一个 MEMBER 方法 get\_info。建立对象类型规范 person\_typ3 的示例如下：

```

CREATE OR REPLACE TYPE person_typ3 AS OBJECT (
    name VARCHAR2(10), gender VARCHAR2(2),
    birthdate DATE, regdate DATE,
    STATIC FUNCTION getdate RETURN DATE,
    MEMBER FUNCTION get_info RETURN VARCHAR2
);
/

```

在执行了上述命令之后，就会建立对象类型规范 person\_typ3。因为该对象类型规范包含了 STATIC 方法和 MEMBER 方法，所以为了使用这两个方法，必须建立对象类型体来实现这两个方法。建立对象类型体的示例如下：

```

CREATE OR REPLACE TYPE BODY person_typ3 IS
    STATIC FUNCTION getdate RETURN DATE IS
        BEGIN
            RETURN SYSDATE;
        END;
    MEMBER FUNCTION get_info RETURN VARCHAR2
        IS
        BEGIN
            RETURN '姓名:' || name || ',注册日期:' || regdate;
        END;

```

```

    END;
END;
/

```

在建立了对象类型规范和对象类型体之后，就可以使用该对象类型及其 STATIC 方法和 MEMBER 方法。基于对象类型 person\_typ3 建立对象表 employee\_tab3，并为其插入数据的示例如下：

```

CREATE TABLE employee_tab3 (
    eno NUMBER(6), person person_typ3,
    sal NUMBER(6,2), job VARCHAR2(10)
);

```

执行了以上命令之后，就会建立对象表 employee\_tab3。因为建立对象类型时定义了 STATIC 方法和 MEMBER 方法，所以在 PL/SQL 块中可以引用这些方法。下面通过给对象表 employee\_tab3 插入数据为例，说明如何在对象类型上使用 STATIC 方法。示例如下：

```

BEGIN
    INSERT INTO employee_tab3 (eno,sal,job,person)
        VALUES(&no,&salary,'&title',person_typ3(
            '&name','&sex','&birthdate',person_typ3.getdate()
        ));
END;
/
输入 no 的值： 1
输入 salary 的值： 2000
输入 title 的值： 高级钳工
输入 name 的值： 王鸣
输入 sex 的值： 男
输入 birthdate 的值： 11-1 月 -75

```

执行了以上 PL/SQL 块之后，就会根据输入信息为对象表 employee\_tab3 插入一条数据，并且注册日期 regdate 通过使用 STATIC 方法 getdate() 提供。为了取得雇员信息，可以使用 MEMBER 方法 get\_info。示例如下：

```

DECLARE
    v_person person_typ3;
BEGIN
    SELECT person INTO v_person FROM employee_tab3
        WHERE eno=&no;
    dbms_output.put_line(v_person.get_info());
END;
/
输入 no 的值： 1
姓名:王鸣,注册日期:22-12 月 -03

```

#### 4. 建立和使用包含 MAP 方法的对象类型

MAP 方法用于将对象实例映射为标量数值 (NUMBER, DATE, VARCHAR2 等)。对于相同对象类型的不同对象实例来说，因为它们的数据类型是复合数据类型，所以对象实例之间不能直接进行比较。为了排序多个对象实例的数据，可以在建立对象类型时定义 MAP 方法。但是，需要注意，一个对象类型最多只能定义一个 MAP 方法，并且 MAP 方法和 ORDER 方

法不能同时使用。下面以建立和使用对象类型 person\_typ4 为例，说明如何在对象类型中使用 MAP 方法。对象类型 person\_typ4 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期等三个属性，另外还包含有用于取得人员年龄的一个 MAP 方法。如下所示：

对象类型 person_typ4		
属性:		方法:
name	姓名	getage: 取得人员年龄
gender	性别	
birthdate	出生日期	

如上所示，对象类型 person\_typ4 包含 name, gender 和 birthdate 等三个属性，以及一个 MAP 方法 getage。建立对象类型规范 person\_typ4 的示例如下：

```
CREATE OR REPLACE TYPE person_typ4 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2), birthdate DATE,
    MAP MEMBER FUNCTION getage RETURN VARCHAR2
);
/

```

因为对象类型规范 person\_typ4 定义了 MAP 函数 getage，所以必须要通过对象类型体实现该函数。建立对象类型体的示例如下：

```
CREATE OR REPLACE TYPE BODY person_typ4 IS
    MAP MEMBER FUNCTION getage RETURN VARCHAR2
    IS
    BEGIN
        RETURN TRUNC((SYSDATE-birthdate)/365);
    END;
END;
/

```

在建立了对象类型规范和对象类型体之后，就可以使用该对象类型及其方法。基于对象类型 person\_typ4 建立对象表 employee\_tab4 并为其插入数据的示例如下：

```
CREATE TABLE employee_tab4(
    eno NUMBER(6), person person_typ4,
    sal NUMBER(6,2), job VARCHAR2(10)
);
INSERT INTO employee_tab4(eno, sal, job, person)
VALUES(1, 1500, '图书管理员', person_typ4(
    '马丽', '女', '11-1月-76'));
INSERT INTO employee_tab4(eno, sal, job, person)
VALUES(2, 2000, '高级焊工', person_typ4(
    '王鸣', '男', '11-5月-75'));
INSERT INTO employee_tab4(eno, sal, job, person)
VALUES(3, 2500, '高级工程师', person_typ4(
    '李奇', '男', '11-5月-70'));

```

在执行了以上语句之后，就会建立对象表 employee\_tab4，并为其插入三条数据。因为在

建立对象类型时定义了 MAP 方法，所以可以在 PL/SQL 块中使用 MAP 方法排序对象实例的数据。下面以比较对象表 employee\_tab4 的前两条数据为例，说明使用 MAP 方法 getage 比较对象实例的方法。示例如下：

```

DECLARE
    TYPE person_table_type IS TABLE OF person_typ4;
    person_table person_table_type;
    v_temp VARCHAR2(100);
BEGIN
    SELECT person BULK COLLECT INTO person_table
        FROM employee_tab4;
    IF person_table(1).getage() > person_table(2).getage() THEN
        v_temp:=person_table(1).name||'比'|| 
            person_table(2).name||'大';
    ELSE
        v_temp:=person_table(1).name||'比'|| 
            person_table(2).name||'小';
    END IF;
    dbms_output.put_line(v_temp);
END;
/
马丽比王鸣小

```

### 5. 建立和使用包含 ORDER 方法的对象类型

ORDER 方法用于比较两个对象实例的大小。注意，一个对象类型最多只能包含一个 ORDER 方法，并且 ORDER 方法不能与 MAP 方法同时使用。下面以建立和使用对象类型 person\_typ5 为例，说明如何在对象类型中使用 ORDER 方法。对象类型 person\_typ5 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期等三个属性，另外还包含有用于比较人员年龄大小的一个 ORDER 方法。如下所示：

对象类型 person_typ5		
属性:		方法:
name	姓名	compare: 比较人员年龄大小
gender	性别	
birthdate	出生日期	

其中，对象类型 person\_typ5 包含 name, gender 和 birthdate 等三个属性，以及一个 ORDER 方法 compare。建立对象类型规范 person\_typ5 的示例如下：

```

CREATE OR REPLACE TYPE person_typ5 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2), birthdate DATE,
    ORDER MEMBER FUNCTION compare(p person_typ5)
    RETURN INT);
/

```

因为对象类型规范 person\_typ5 定义了 ORDER 函数 compare，所以必须要使用对象类型

体实现该函数。建立对象类型体的示例如下：

```
CREATE OR REPLACE TYPE BODY person_typ5 IS
  ORDER MEMBER FUNCTION compare(p person_typ5) RETURN INT
  IS
  BEGIN
    CASE
      WHEN birthdate>p.birthdate THEN RETURN 1;
      WHEN birthdate=p.birthdate THEN RETURN 0;
      WHEN birthdate<p.birthdate THEN RETURN -1;
    END CASE;
  END;
END;
/
```

在建立了对象类型规范和对象类型体之后，就可以使用该对象类型及其方法了。基于对象类型 person\_typ5 建立对象表 employee\_tab5，并为其插入两条数据。示例如下：

```
CREATE TABLE employee_tab5(
  eno NUMBER(6),person person_typ5,
  sal NUMBER(6,2),job VARCHAR2(10)
);
INSERT INTO employee_tab5(eno,sal,job,person)
VALUES(1,1500,'图书管理员',person_typ5(
  '马丽','女','11-1月-76'));
INSERT INTO employee_tab5(eno,sal,job,person)
VALUES(2,2000,'高级焊工',person_typ5(
  '王鸣','男','11-5月-75'));
```

在执行了以上命令之后，就会建立对象表 employee\_tab5，并为其插入两条数据。因为建立对象类型时定义了 ORDER 方法，所以在 PL/SQL 块中可以使用 ORDER 方法比较不同对象实例的数据。下面以比较对象表 employee\_tab5 的前两条数据为例，说明使用 MAP 方法 compare 比较对象实例的方法。示例如下：

```
DECLARE
  TYPE person_table_type IS TABLE OF person_typ5;
  person_table person_table_type;
  v_temp VARCHAR2(100);
BEGIN
  SELECT person BULK COLLECT INTO person_table
    FROM employee_tab5;
  IF person_table(1).compare(person_table(2))=1 THEN
    v_temp:=person_table(1).name||'比'|||
    person_table(2).name||'大';
  ELSE
    v_temp:=person_table(1).name||'比'|||
    person_table(2).name||'小';
  END IF;
  dbms_output.put_line(v_temp);
END;
```

马丽比王鸣大

### 6. 建立和使用包含自定义构造方法的对象类型

建立对象类型时，Oracle会自动为对象类型生成相应的构造方法，并且构造方法用于初始化对象实例。在Oracle9i之前，用户只能使用系统所提供的默认构造方法初始化对象实例；而从Oracle9i开始，Oracle允许开发人员自定义构造方法，从而使得初始化对象实例更加灵活，更加符合面向对象程序设计（OOP）的特征。注意，当自定义构造方法时，构造方法的名称必须要与对象类型的名称完全相同，并且必须要使用CONSTRUCTOR FUNCTION关键字定义构造方法。下面以建立和使用对象类型person\_typ6为例，说明如何在对象类型中自定义构造方法。对象类型person\_typ6将用于描述人员信息，该对象类型包含有姓名、性别、出生日期等三个属性，另外还包含有用于初始化对象实例的三个自定义构造方法。如下所示：

对象类型 person_typ6		
属性：		构造方法：
name	姓名	person_typ6(name)
gender	性别	person_typ6(name,gender)
birthdate	出生日期	person_typ6(name,gender,birthdate)

其中，对象类型person\_typ6包含name，gender和birthdate等三个对象属性，以及三个带有不同输入参数的构造方法。建立对象类型person\_typ6的示例如下：

```
CREATE OR REPLACE TYPE person_typ6 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2), birthdate DATE,
    CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2)
        RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2,
        gender VARCHAR2) RETURN SELF AS RESULT,
    CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2,
        gender VARCHAR2, birthdate DATE) RETURN SELF AS RESULT
);
/

```

在执行了以上PL/SQL块之后，就会建立对象类型规范person\_typ6，并且定义三个构造方法。因为在建立对象类型规范时定义了构造方法，所以必须要建立对象类型体实现其构造方法。示例如下：

```
CREATE OR REPLACE TYPE BODY person_typ6 IS
    CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2)
        RETURN SELF AS RESULT
    IS
    BEGIN
        self.name:=name;
        self.gender:='女';
        self.birthdate:=SYSDATE;
```

```

        return;
    END;
CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2,
    gender VARCHAR2) RETURN SELF AS RESULT
IS
BEGIN
    self.name:=name;
    self.gender:=gender;
    self.birthdate:=SYSDATE;
    return;
END;
CONSTRUCTOR FUNCTION person_typ6(name VARCHAR2,
    gender VARCHAR2,birthdate DATE) RETURN SELF AS RESULT
IS
BEGIN
    self.name:=name;
    self.gender:=gender;
    self.birthdate:=birthdate;
    return;
END;
END;
/

```

在执行以上命令建立了对象类型体之后，就可以在应用程序中使用各种构造方法初始化对象实例。下面以基于对象类型 person\_typ5 建立对象表，并使用各种构造方法为其插入数据为例，说明如何使用各种自定义构造方法。示例如下：

```

CREATE TABLE employee_tab6(
    eno NUMBER(6),person person_typ6,
    sal NUMBER(6,2),job VARCHAR2(10)
);
INSERT INTO employee_tab6 (eno,sal,job,person)
    VALUES(1,1500,'图书管理员',person_typ6('马丽'));
INSERT INTO employee_tab6 (eno,sal,job,person)
    VALUES(2,2000,'高级钳工',person_typ6('王鸣','男'));
INSERT INTO employee_tab6 (eno,sal,job,person)
    VALUES(3,2500,'高级工程师',person_typ6('李奇','男','01-1月-70'));

```

在执行了以上命令之后，就会建立对象表 employee\_tab6，并使用不同构造方法分别插入一条数据。

### 15.3 建立和使用复杂对象类型

简单对象类型是指独立的并且与其他对象类型无关的对象类型，而复杂对象类型是指与其他对象类型具有关联关系的对象类型。上节详细介绍了如何建立和使用简单对象类型、行对象、列对象，并且介绍了如何使用 MEMBER 方法、STATIC 方法、MAP 方法、ORDER 方法和自定义构造方法，本节将介绍如何使用复杂对象类型。

### 15.3.1 对象类型嵌套

对象类型嵌套是指在一个对象类型中嵌套另一个对象类型。定义对象类型时，必须要为对象类型提供相应的属性。指定对象属性时，除了要给出属性名称之外，还需要给出对象属性所对应的数据类型。指定对象属性的数据类型时，不仅可以使用标量数据类型（例如 NUMBER、DATE、VARCHAR2 等），而且还可以使用自定义的对象类型（嵌套对象类型）。下面以建立和使用对象类型 addr\_typ7 和 person\_typ7 为例，说明如何建立和使用嵌套对象类型。

#### 1. 建立对象类型 addr\_typ7

对象类型 addr\_typ7 将用于描述地址信息，该对象类型包含有省会、城市、街道、邮政编码等四个属性，另外还包含一个用于取得地址信息的方法。如下所示：

对象类型 addr_typ7		
属性：		方法：
state	省会	get_addr：取得地址信息
city	城市	
street	街道	
zipcode	邮政编码	

其中，对象类型 addr\_typ7 包括 state、city、street 和 zipcode 等四个对象属性，以及一个 MEMBER 函数 get\_addr。建立该对象类型的示例如下：

```
CREATE OR REPLACE TYPE addr_typ7 AS OBJECT(
    state VARCHAR2(20), city VARCHAR2(20),
    street VARCHAR2(50), zipcode VARCHAR2(6),
    MEMBER FUNCTION get_addr RETURN VARCHAR2
);
/

```

因为在建立对象类型规范 addr\_typ7 时定义了方法 get\_addr，所以必须要建立对象类型体实现该方法。示例如下：

```
CREATE OR REPLACE TYPE BODY addr_typ7 AS
    MEMBER FUNCTION get_addr RETURN VARCHAR2 IS
        BEGIN
            RETURN state||city||street;
        END;
    END;
/

```

在建立了对象类型规范和对象类型体之后，就可以引用对象类型 addr\_typ7 及其对象方法。

#### 2. 建立对象类型 person\_typ7

对象类型 person\_typ7 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期、地址等四个属性，另外还包含有用于取得人员信息的一个方法。如下所示：

对象类型 person_typ7	
属性:	方法:
name 姓名	get_info: 取得人员信息
gender 性别	
birthdate 出生日期	
address 地址	

其中，对象类型 person\_typ7 包括 name, gender, birthdate, address 等四个对象属性，以及一个 MEMBER 函数 get\_info。建立该对象类型规范的示例如下：

```
CREATE OR REPLACE TYPE person_typ7 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2),
    birthdate DATE, address addr_typ7,
    MEMBER FUNCTION get_info RETURN VARCHAR2
);
/

```

因为建立对象类型 person\_typ7 时定义了对象方法 get\_info，所以必须要建立对象类型体实现该方法。建立对象类型体的示例如下：

```
CREATE OR REPLACE TYPE BODY person_typ7 AS
    MEMBER FUNCTION get_info RETURN VARCHAR2
    IS
    BEGIN
        RETURN '姓名:' || name || ',家庭住址:' || address.get_addr();
    END;
END;
/

```

在建立了对象类型规范和对象类型体之后，就可以使用对象类型 person\_typ7 及其对象方法。

### 3. 建立并操纵对象表 employee\_tab7

对象表 employee\_tab7 用于存放雇员信息，包括雇员号、工资、岗位以及人员详细信息（姓名、住址等）。建立对象表 employee\_tab7 的示例如下：

```
CREATE TABLE employee_tab7(
    eno NUMBER(6), person person_typ7,
    sal NUMBER(6,2), job VARCHAR2(10)
);

```

在建立了对象表 employee\_tab7 之后，就可以执行 SQL 操作操纵该对象表了。

### 4. 操纵对象表 employee\_tab7

#### 示例一：在 PL/SQL 块中为对象表插入数据

当为对象表插入数据时，需要使用对象类型的构造方法。下面通过给对象表 employee\_tab7 增加两行数据为例，说明为嵌套对象类型列插入数据的方法。示例如下：

```
BEGIN
    INSERT INTO employee_tab7(eno,sal,job,person)
    VALUES(1,1500,'图书管理员',person_typ7(

```

```

    '马丽', '女', '01-11月-76', addr_typ7(
    '内蒙古自治区', '呼和浩特市', '呼伦北路 22 号', '010010')
  );
  INSERT INTO employee_tab7(eno,sal,job,person)
  VALUES(2,2000,'高级钳工',person_typ7(
    '王鸣', '男', '11-12月-75', addr_typ7(
    '内蒙古自治区', '呼和浩特市', '呼伦北路 50 号', '010010'
  )));
END;
/

```

**示例二：在 PL/SQL 块中更新对象列数据**

当更新对象列数据时，首先应该将对象列数据检索到对象类型变量，然后改变对象类型变量的相应属性，最后执行 UPDATE 语句更新对象列数据。示例如下：

```

DECLARE
  v_person person_typ7;
BEGIN
  SELECT person INTO v_person FROM employee_tab7
  WHERE eno=&no;
  v_person.address.street:='&street';
  UPDATE employee_tab7 SET person=v_person
  WHERE eno=&no;
END;
/
输入 no 的值： 1
输入 street 的值： 北垣东街 11 号

```

**示例三：在 PL/SQL 块中检索对象列数据**

当在 PL/SQL 块中输出对象列数据时，应该首先将对象列数据检索到对象类型变量中，然后使用对象方法显示对象数据，或者直接使用对象属性显示数据。示例如下：

```

DECLARE
  v_person person_typ7;
BEGIN
  SELECT person INTO v_person FROM employee_tab7
  WHERE eno=&eno;
  dbms_output.put_line(v_person.get_info);
END;
/
输入 eno 的值： 1
姓名：马丽，家庭住址：内蒙古自治区呼和浩特市北垣东街 11 号

```

**示例四：在 PL/SQL 块中删除对象表数据**

在 PL/SQL 块中删除对象表数据，与删除普通表数据没有区别。示例如下：

```

BEGIN
  DELETE FROM employee_tab7 WHERE eno=&no;
END;
/
输入 no 的值： 1

```

### 15.3.2 参照对象类型

参照对象类型是指在建立对象表时使用 REF 定义表列，REF 实际是指向行对象数据的逻辑指针。通过使用 REF 定义表列，可以使得一个对象表引用另一个对象表（行对象）的数据，从而节省磁盘空间和内存空间。下面以建立对象类型 person\_typ8、行对象 person\_tab8 和对象表 employee\_tab8 为例，说明如何使用参照对象类型。

#### 1. 建立对象类型 person\_typ8

对象类型 person\_typ8 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期、地址等四个属性，另外还包含有用于取得人员信息的方法。如下所示：

对象类型 person_typ8		
属性:		方法:
name	姓名	get_info: 取得人员信息
gender	性别	
birthdate	出生日期	
address	地址	

其中，对象类型 person\_typ8 包含 name, gender, birthdate, address 等四个对象属性，以及一个 MEMBER 函数 get\_info。建立该对象类型的示例如下：

```
CREATE OR REPLACE TYPE person_typ8 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2),
    birthdate DATE, address VARCHAR2(50),
    MEMBER FUNCTION get_info RETURN VARCHAR2
);
/

```

因为在定义对象类型规范时定义了方法 get\_info，所以必须建立对象类型体实现该方法。示例如下：

```
CREATE OR REPLACE TYPE BODY person_typ8 AS
    MEMBER FUNCTION get_info RETURN VARCHAR2
    IS
    BEGIN
        RETURN '姓名:' || name || ',家庭住址:' || address;
    END;
END;
/

```

#### 2. 建立行对象 person\_tab8 并追加数据

行对象 person\_tab8 用于存放人员信息，在建立了对象类型 person\_typ8 之后，可以基于该对象类型建立行对象 person\_tab8，以存放人员信息。建立行对象 person\_tab8 并为其追加数据的示例如下：

```
CREATE TABLE person_tab8 OF person_typ8;
INSERT INTO person_tab8 VALUES('马丽', '女', '11-1月-76',
```

```
'呼和浩特市北垣东街 11 号');
INSERT INTO person_tab8 VALUES ('王鸣', '男', '11-2 月-76',
    '呼和浩特市呼伦南路 21 号');
```

### 3. 建立对象表 employee\_tab8

对象表 employee\_tab8 用于存放雇员信息。因为 person\_tab8 表已经包含了人员的部分信息，所以为了降低占用空间，employee\_tab8 表应该直接引用表 person\_tab8 的数据。为了引用行对象的数据，在建表时应该使用 REF 定义表列。建立对象表 employee\_tab8 的示例如下：

```
CREATE TABLE employee_tab8 (
    eno NUMBER(6), person REF person_typ8,
    sal NUMBER(6,2), job VARCHAR2(10)
);
```

### 4. 操纵对象表 employee\_tab8

#### 示例一：为对象表插入数据

因为在建立对象表 employee\_tab8 时使用 REF 定义了表列 person，所以给该表列插入数据时必须引用其他表的数据。引用行对象时，需要使用函数 REF()，其返回值实际是指向相应数据行的指针。示例如下：

```
BEGIN
    INSERT INTO employee_tab8
        SELECT 1,REF(a),2000,'图书管理员' FROM person_tab8 a
        WHERE a.name='马丽';
    INSERT INTO employee_tab8
        SELECT 2,REF(a),2200,'高级钳工' FROM person_tab8 a
        WHERE a.name='王鸣';
END;
/
```

#### 示例二：检索 REF 对象列数据

检索 REF 对象列数据时，因为该列实际存放的是指向行对象数据的指针，所以为了取得行对象的相应数据，必须要使用 DEREF 函数取得 REF 列所对应的实际数据。示例如下：

```
DECLARE
    v_person person_typ8;
BEGIN
    SELECT DEREF(person) INTO v_person FROM employee_tab8
        WHERE eno=&no;
    dbms_output.put_line(v_person.get_info);
END;
/
输入 no 的值: 1
```

姓名:马丽,家庭住址:呼和浩特市北垣东街 11 号

### 示例三：更新 REF 对象列数据

因为 REF 列存放着指向行对象数据的指针，所以如果要修改其列所引用的数据，就必须修改相应的行对象。示例如下：

```
DECLARE
    v_person person_typ8;
```

```

BEGIN
    SELECT DEREF(person) INTO v_person FROM employee_tab8
        WHERE eno=&no;
    v_person.address:='&address';
    UPDATE person_tab8 SET address=v_person.address
        WHERE name=v_person.name;
END;
/
输入 no 的值: 1
输入 address 的值: 呼和浩特市呼伦北路 100 号

```

#### 示例四：删除对象表数据

在 PL/SQL 块中删除对象表数据时，与删除普通表数据没有区别。示例如下：

```

BEGIN
    DELETE FROM employee_tab8 WHERE eno=&no;
END;
/
输入 no 的值: 1

```

### 15.3.3 对象类型继承

对象类型继承是 Oracle9i 新增加的特征，是指一个对象类型继承另一个对象类型，并且对象类型继承由父类型和子类型组成。其中，父类型用于定义不同对象类型的公用属性和方法，而子类型不仅继承了父类型的公用属性和方法，而且还可具有自己的私有属性和方法。当使用对象类型继承时，在定义父类型时必须要指定 NOT FINAL 选项；如果不指定该选项，默认选项为 FINAL，表示该对象类型不能被继承。下面以建立父类型 person\_typ9、子类型 employee\_typ9、对象表 employee\_tab9 为例，说明使用对象类型继承的方法。

#### 1. 建立父对象类型 person\_typ9

对象类型 person\_typ9 将用于描述人员信息，该对象类型包含有姓名、性别、出生日期、地址等四个属性，另外还包含有用于取得人员信息的方法。如下所示：

对象类型 person_typ9		
属性:	方法:	
name	姓名	get_info: 取得人员信息
gender	性别	
birthdate	出生日期	
address	地址	

其中，对象类型 person\_typ9 包含 name, gender, birthdate 和 address 等四个对象属性，以及一个 MEMBER 函数 get\_info。因为该对象类型将被作为父类型使用，所以在建立时必须指定 NOT FINAL 选项。建立该对象类型规范的示例如下：

```

CREATE OR REPLACE TYPE person_typ9 AS OBJECT(
    name VARCHAR2(10), gender VARCHAR2(2),

```

```

birthdate DATE, address VARCHAR2(50),
MEMBER FUNCTION get_info RETURN VARCHAR2
) NOT FINAL;
/

```

因为在建立对象类型规范时定义了对象方法 `get_info`, 所以必须建立对象类型体来实现该方法。示例如下:

```

CREATE OR REPLACE TYPE BODY person_typ9 AS
  MEMBER FUNCTION get_info RETURN VARCHAR2
  IS
    BEGIN
      RETURN '姓名:' || name || ',家庭住址:' || address;
    END;
END;
/

```

## 2. 建立子对象类型 `employee_typ9`

对象类型 `employee_typ9` 被用于描述雇员信息, 它不仅会继承 `person_typ9` 的所有属性和方法, 而且还包括了编号、工资、岗位等私有属性, 以及用于取得雇员其他信息的私有方法。如下所示:

对象类型 <code>employee_typ9</code>	
父类型: <code>person_typ9</code>	私有属性和方法:
	eno 雇员号
	sal 工资
	job 岗位
	get_other: 取得雇员其它信息

其中, 对象类型 `employee_typ9` 将继承对象类型 `person_typ9` 的所有对象属性和对象方法, 并且还包含有 `eno`, `sal`, `job` 等三个私有属性, 以及一个 MEMBER 函数 `get_other`。因为在定义对象类型 `person_typ9` 时指定了 NOT FINAL 关键字, 所以为了引用该对象类型已有的属性和方法, 对象类型 `employee_typ9` 可以继承该数据类型。建立子对象类型 `employee_typ9` 的示例如下:

```

CREATE OR REPLACE TYPE employee_typ9 UNDER person_typ9(
  eno NUMBER(6), sal NUMBER(6,2), job VARCHAR2(10),
  MEMBER FUNCTION get_other RETURN VARCHAR2
);
/

```

建立了对象类型 `employee_typ9` 之后, 它就会继承对象类型 `person_typ9` 的所有属性(`name`, `gender`, `birthdate`, `address`) 和方法 (`get_info`), 并且还具有私有属性 (`eno`, `sal`, `job`) 以及私有方法 (`get_other`)。因为在建立该对象类型规范时定义了方法 `get_other`, 所以必须建立对象类型体, 以便实现该方法。示例如下:

```

CREATE OR REPLACE TYPE BODY employee_typ9 AS
  MEMBER FUNCTION GET_OTHER RETURN VARCHAR2

```

```

IS
BEGIN
    RETURN '雇员名称:'||name||', 工资:'||sal;
END;
END;
/

```

### 3. 建立并操纵对象表 employee\_tab9

#### 示例一：建立对象表并插入数据

```

SQL> CREATE TABLE employee_tab9 OF employee_typ9;
SQL> INSERT INTO employee_tab9 VALUES(
2  '马丽','女','01-11月-76','呼和浩特市呼伦北路11号',
3  1,1500,'图书管理员',
4 );
SQL> INSERT INTO employee_tab9 VALUES(
2  '王鸣','男','11-11月-76','呼和浩特市呼伦北路15号',
3  2,2000,'高级钳工',
4 );

```

#### 示例二：使用对象方法输出数据

```

DECLARE
    v_employee employee_typ9;
BEGIN
    SELECT VALUE(a) INTO v_employee FROM employee_tab9 a
        WHERE a.eno=&no;
    dbms_output.put_line(v_employee.get_info);
    dbms_output.put_line(v_employee.get_other);
END;
/
输入 no 的值: 1
姓名:马丽,家庭住址:呼和浩特市呼伦北路11号
雇员名称:马丽,工资:1500

```

## 15.4 维护对象类型

### 1. 显示对象类型信息

执行 CREATE TYPE 命令建立对象类型时，Oracle 会将对象类型的信息存放到数据字典中。通过查询数据字典视图 USER\_TYPES，可以显示当前用户所包含的所有对象类型的信息。示例如下：

```

SQL> SELECT type_name,attributes,final FROM user_types;
TYPE_NAME          ATTRIBUTES FIN
-----
PERSON_TYP1          3 YES
PERSON_TYP2          4 YES
PERSON_TYP3          4 YES
PERSON_TYP4          3 YES

```

```
PERSON TYPE9          4 NO
```

...

如上所示，`type_name` 用于标识对象类型的名称，`attributes` 用于标识对象类型所包含的属性个数，而 `final` 用于标识对象类型是否可以作为父类型使用（YES：不可，NO：可以）。另外，在 SQL\*Plus 中，通过执行 `DESC` 命令可以查看对象类型所包含的属性和方法。示例如下：

```
SQL> DESC person_typ1
      名称          是否为空? 类型
-----
      NAME          VARCHAR2(10)
      GENDER        VARCHAR2(2)
      BIRTHDATE     DATE
```

## 2. 增加和删除对象类型属性

在建立了对象类型之后，通过使用 `ALTER TYPE` 命令可以为对象类型增加新的属性，或者删除已存在属性。如果已经基于对象类型建立了对象类型或对象表，那么在为对象类型增加或删除属性时必须要带有 `CASCADE` 关键字。示例如下：

```
SQL> ALTER TYPE person_typ1 ADD ATTRIBUTE
      2 address VARCHAR2(50) CASCADE;
SQL> ALTER TYPE person_typ1 DROP ATTRIBUTE birthdate CASCADE;
```

## 3. 增加和删除对象类型方法

使用 `ALTER TYPE` 不仅可以增加和删除对象类型的属性，也可以增加和删除对象类型的方法。如果已经基于对象类型建立了对象类型或对象表，那么在增加和删除对象方法时必须要带有 `CASCADE` 关键字。示例如下：

```
SQL> ALTER TYPE person_typ1
      2 ADD MEMBER FUNCTION get_info RETURN VARCHAR2
      3 CASCADE;
SQL> CREATE OR REPLACE TYPE BODY person_typ1 AS
      2 MEMBER FUNCTION get_info RETURN VARCHAR2 IS
      3 BEGIN
      4     RETURN '雇员名:'||name||',家庭住址:'||address;
      5 END;
      6 END;
      7 /
```

## 15.5 习题

1. 定义对象类型时，为了比较对象实例的数据，可以使用哪些对象方法？
  - A. MEMBER 方法
  - B. STATIC 方法
  - C. ORDER 方法
  - D. MAP 方法
  - E. 不需要任何对象方法
2. 当定义父对象类型时，必须指定哪个关键字？
  - A. NOT FINAL
  - B. FINAL
  - C. 不需要指定任何关键字
3. 以下哪些对象方法可以使用对象实例调用？
  - A. MEMBER 方法
  - B. STATIC 方法
  - C. ORDER 方法
  - D. MAP 方法

- A. MEMBER 方法 B. STATIC 方法 C. ORDER 方法 D. MAP 方法
4. 以下哪种对象方法只能由对象类型调用?
- A. MEMBER 方法 B. STATIC 方法 C. ORDER 方法 D. MAP 方法
5. 建立对象类型 bank\_account, 其对象属性和对象方法如下所示:

对象类型 bank_account	
属性:	方法:
acct_no 账户号	deposit: 存款到账户
balance 账户金额	withdraw: 从账户中取款
status 账户状态	

如上所示, 账户号是 6 位的数字, status 可以是“OPEN”或“CLOSE”状态。存款时, 需要确保账户状态为“OPEN”; 取款时, 不仅要确保账户状态为“OPEN”, 而且还确保取款额度不能大于账户余额。

6. 基于对象类型 bank\_account 建立行对象 account, 然后执行以下操作:

- 为行对象增加一条数据: 123456 (账户号)、5586.87 (金额)、OPEN (状态);
- 使用对象方法 deposit 为账户号 123456 增加 2000 的存款;
- 使用对象方法 withdraw 从账户号 123456 取出 1500 元;

7. 建立对象类型 employee\_type, 其对象属性和对象方法如下所示:

对象类型 employee_type	
属性:	方法:
eno 雇员号	get_info: 输入雇员号取得雇员名和工资
name 雇员名	change_sal: 改变雇员工资
sal 雇员工资	change_job: 改变雇员岗位
job 雇员岗位	
hiredate 雇佣日期	

8. 建立嵌套表类型 employee\_table\_type, 其元素类型采用 employee\_type。

9. 建立对象表 department, 包含 dno (部门号)、dname (部门名)、loc (部门位置) 和 employee (使用 employee\_table\_type 对象类型), 然后在该表上执行以下 SQL 操作:

- 插入第一行数据: 部门号 (10)、部门名 (财务处)、部门位置 (北京)  
雇员一信息: 1 马丽、2000、处长、03-10月-98  
雇员二信息: 2 张华、1500、会计、13-10月-99  
雇员三信息: 3 明珠、1200、出纳、11-10月-01
- 插入第二行数据: 部门号 (20)、部门名 (物资处)、部门位置 (上海)  
雇员四信息: 4 秦琼、2000、处长、03-10月-98  
雇员五信息: 5 罗数、1500、采购、13-10月-99  
雇员六信息: 6 裴伟、1200、保管、11-10月-01

- 使用 SQL\*Plus 替代变量输入部门号，然后使用对象方法 get\_info 显示该部门的所有雇员信息，格式如下：

输入 deptno 的值： 20

雇员名:秦琼,工资:2000

雇员名:罗敏,工资:1500

雇员名:裴伟,工资:1200

- 使用 SQL\*Plus 替代变量输入部门号、雇员号和新工资，然后使用对象方法 change\_sal 更新该雇员的工资，格式如下：

输入 deptno 的值： 10

输入 empno 的值： 1

输入 salary 的值： 2500

PL/SQL 过程已成功完成。

## 第 16 章 使用 LOB 对象

LOB (Large Object) 是专门用于处理大对象的一种数据类型，其所存放的数据长度可以达到 4G 字节。在 Oracle 的早期版本中，存放大对象是使用 LONG、LONG RAW 来实现的，随着版本的升级，这两种数据类型将被逐渐淘汰。Oracle 建议使用 LOB 类型取代 LONG 和 LONG RAW 类型。本章将详细介绍如何在 PL/SQL 中使用 LOB 对象，在学习了本章之后，读者应该学会：

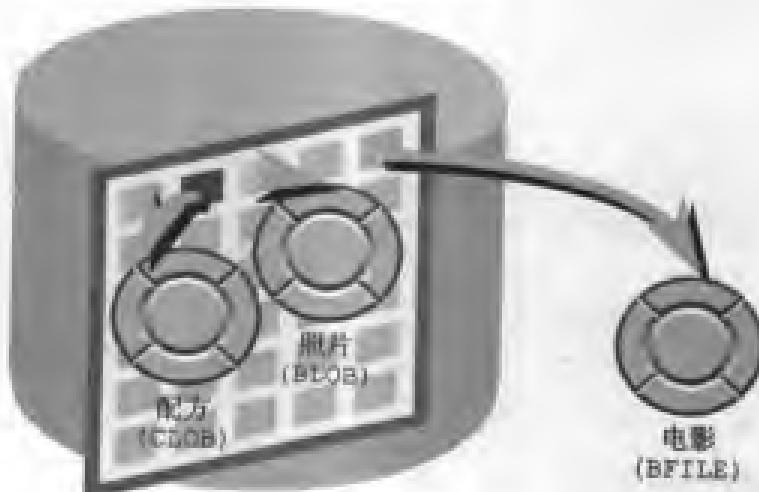
- 各种 LOB 类型的作用，以及 LOB 类型和 LONG、LONG RAW 类型的区别；
- 在 PL/SQL 块中使用 CLOB 类型；
- 在 PL/SQL 块中使用 BLOB 类型；
- 在 PL/SQL 块中使用 BFILE 类型。

### 16.1 LOB 简介

LOB 类型专门用于存储大对象的数据，包括大文本、图形/图像、视频剪切等大数据。尽管现在仍然可以使用 LONG 和 LONG RAW 类型存放类似数据，但因为这两种类型会被逐渐淘汰，所以建议使用 LOB 类型。

#### 1. LOB 类型

Oracle 将 LOB 分为两种：内部 LOB 和外部 LOB。内部 LOB 包括 CLOB、BLOB 和 NCLOB 三种类型，它们的数据被存储在数据库中，并且支持事务操作（提交、回退、保存点）；外部 LOB 只有 BFILE 一种类型，该类型的数据被存储在操作系统（OS）文件中，并且不支持事务操作。其中，CLOB/NCLOB 用于存储大批量字符数据，BLOB 用于存储大批量二进制数据，而 BFILE 则存储着指向 OS 文件的指针。如下图所示：



如图中所示，因为医药配方需要使用大量中文说明，所以可以将其所对应的数据库列定

义为 CLOB 类型；因为照片对应于图形/图像（二进制数据），所以可以将其所对应的数据库列定义为 BLOB 类型；为了通过数据库访问 OS 电影文件，可以在数据库中定义 BFILE 类型的数据库列。

## 2. 比较 LOB 和 LONG

LONO 和 LOB 都可以用于存储大批量字符数据和二进制数据。在 Oracle8 版本之前，存放大对象数据是使用 LONG 和 LONG RAW 类型来完成；从 Oracle8 版本开始，既可以使用 LONG 和 LONG RAW 存放大对象，也可以使用 LOB 存放大对象。下表列出了它们的区别：

LONG, LONG RAW	LOB
一个表只能有一个 LONG 或 LONG RAW 列	一个表可以有多个 LOB 列
数据最大长度：2G 字节	数据最大长度：4G 字节
SELECT：返回数据	SELECT：返回 LOB 定位符
数据存储：行内	数据存储：行内或行外
不支持对象类型	支持对象类型
数据访问：顺序访问	数据访问：随机访问

注意，使用 LONG/LONG RAW 列时，其数据与其他列的数据相邻存放；而使用 LOB 列时，如果数据小于 4000 字节，则与其他列相邻存放（行内）；如果数据大于 4000 字节，则数据被存放到专门的 LOB 段中（行外）。

## 3. 临时 LOB

编写 PL/SQL 应用程序时，可以使用临时 LOB。临时 LOB 相当于局部变量，与数据库表无关，并且只能由当前应用程序建立和使用。注意，当在 SQL 语句中使用临时 LOB 时，它们只能作为输入宿主变量使用，可在 WHERE 子句，VALUES 子句和 SET 子句中使用临时 LOB，而不能在 INTO 子句中使用临时 LOB。

## 16.2 DBMS\_LOB 包

DBMS\_LOB 是 Oracle 提供的、专门用于处理 LOB 类型数据的 PL/SQL 包，该包定义了一些常量、过程和函数，下面简述该包所包含的组件。

### 1. 常量

在 DBMS\_LOB 包中定义了一些常量，这些常量可以在 PL/SQL 应用程序中直接引用，引用方法为包名.常量名（例如 DBMS\_LOB.call）。DBMS\_LOB 包定义了以下六个常量：

```
file_READONLY CONSTANT BINARY_INTEGER := 0;
lob_READONLY CONSTANT BINARY_INTEGER := 0;
lob_READWRITE CONSTANT BINARY_INTEGER := 1;
lobmaxsize CONSTANT INTEGER := 4294967295;
call CONSTANT PLS_INTEGER := 12;
session CONSTANT PLS_INTEGER := 10;
```

### 2. 过程 APPEND

该过程用于将源 LOB 变量的内容添加到目标 LOB 变量的尾部。注意，该过程只适用于

内部 LOB 类型 (BLOB 和 CLOB)，而不适用于 BFILE 类型。语法如下：

```
DBMS_LOB.APPEND (
    dest_lob IN OUT NOCOPY BLOB,src_lob IN BLOB);
DBMS_LOB.APPEND (
    dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
    src_lob IN CLOB CHARACTER SET dest_lob%CHARSET);
```

如上所示，dest\_lob 用于指定目标 LOB 变量，src\_lob 用于指定源 LOB 变量。使用该过程的示例如下：

```
DECLARE
    dest_lob CLOB;
    src_lob CLOB;
BEGIN
    src_lob:='中国';
    dest_lob:='你好，';
    dbms_lob.append(dest_lob,src_lob);
    dbms_output.put_line(dest_lob);
END;
/
你好，中国
```

### 3. 过程 CLOSE

该过程用于关闭已经打开的 LOB，它不仅适用于 CLOB/NCLOB 和 BLOB 类型，也适用于 BFILE 类型。语法如下：

```
DBMS_LOB.CLOSE (lob_loc IN OUT NOCOPY BLOB/CLOB/BFILE);
```

### 4. 函数 COMPARE

该函数用于比较两个 LOB 的全部内容或部分内容。不仅适用于 CLOB/NCLOB 和 BLOB 类型，也适用于 BFILE 类型。注意，该函数只能用于比较同类型的 LOB 变量。语法如下：

```
DBMS_LOB.COMPARE (
    lob_1 IN BLOB/CLOB/BFILE,lob_2 IN BLOB/CLOB/BFILE,
    amount IN INTEGER := 4294967295,
    offset_1 IN INTEGER := 1,
    offset_2 IN INTEGER := 1)
RETURN INTEGER;
```

如上所示，lob\_1 用于指定第一个 LOB 变量；lob\_2 用于指定第二个 LOB 变量；amount 用于指定字符个数 (CLOB) 或字节个数 (BLOB)；offset\_1 用于指定第一个 LOB 的起始比较位置；offset\_2 用于指定第二个 LOB 的起始比较位置。如果比较结果相同，则返回 0；如果比较结果不同，则返回一个非 0 的整数。使用该函数的示例如下：

```
DECLARE
    dest_lob CLOB;
    src_lob CLOB;
BEGIN
    src_lob:='中国';
    dest_lob:='&content';
    IF dbms_lob.compare(src_lob,dest_lob)=0 THEN
        dbms_output.put_line('内容相同');
```

```

    ELSE
        dbms_output.put_line('内容不同');
    END IF;
END;
/
输入 content 的值: 中国
内容相同

```

### 5. 过程 COPY

该过程用于将源 LOB 变量的部分或全部内容复制到目标 LOB 变量中，它只适用于内部 LOB 类型（CLOB/NCLOB/BLOB），而不适用于 BFILE 类型。语法如下：

```
DBMS_LOB.COPY (
    dest_lob IN OUT NOCOPY BLOB/CLOB/NCLOB,
    src_lob IN BLOB/CLOB/NCLOB,
    amount IN INTEGER,
    dest_offset IN INTEGER := 1,
    src_offset IN INTEGER := 1);
```

如上所示，dest\_offset 用于指定要复制到目标 LOB 变量的起始位置，src\_offset 用于指定源 LOB 变量中开始复制的起始位置。使用该过程的示例如下：

```
DECLARE
    dest_lob CLOB;
    src_lob CLOB;
    amount INT;
BEGIN
    src_lob:='中国';
    dest_lob:='你好, ';
    amount:=dbms_lob.getlength(src_lob);
    dbms_lob.copy(dest_lob,src_lob,amount,3);
    dbms_output.put_line(dest_lob);
END;
/
你好中国
```

### 6. 过程 CREATETEMPORARY

该过程用于建立临时 LOB，只适用于内部 LOB 类型（BLOB/CLOB/NCLOB），但不适用于 BFILE 类型。当执行该过程建立临时 LOB 时，Oracle 会将该临时 LOB 建立在用户的临时表空间中。语法如下：

```
DBMS_LOB.CREATETEMPORARY (
    lob_loc IN OUT NOCOPY BLOB/CLOB/NCLOB,
    cache IN BOOLEAN,dur IN PLS_INTEGER := 10);
```

如上所示，lob\_loc 用于指定 LOB 定位符；cache 用于指定是否要将 LOB 读取到缓冲区；dur 用于指定何时清除临时 LOB（10：会话结束清除临时 LOB；12：调用结束清除临时 LOB）。使用该过程的示例如下：

```
DECLARE
    src_lob CLOB;
BEGIN
```

```

    dbms_lob.createtemporary(src_lob, TRUE);
END;
/

```

## 7. 过程 ERASE

该过程用于删除 LOB 变量的全部内容或部分内容，只适用于内部 LOB 类型 (BLOB/CLOB/NCLOB)，而不适用于 BFILE 类型。语法如下：

```

DBMS_LOB.ERASE (
    lob_loc IN OUT NOCOPY BLOB/CLOB/NCLOB,
    amount IN OUT NOCOPY INTEGER,
    offset IN INTEGER := 1);

```

如上所示，offset 用于指定开始删除内容的起始位置。使用该过程的示例如下：

```

DECLARE
    src_lob CLOB;
    offset INT;
    amount INT;
BEGIN
    src_lob:='欢迎使用 PL/SQL 编程指南';
    amount:=10;
    offset:=5;
    dbms_lob.erase(src_lob,amount,offset);
    dbms_output.put_line(src_lob);
END;
/

```

欢迎使用

## 8. 过程 FILECLOSE

该过程用于关闭已经打开的 BFILE 定位符所指向的 OS 文件。语法如下：

```
DBMS_LOB.FILECLOSE (file_loc IN OUT NOCOPY BFILE);
```

如上所示，file\_loc 用于指定 BFILE 定位符。

## 9. 过程 FILECLOSEALL

该过程用于关闭当前会话已经打开的所有 BFILE 文件。语法如下：

```
DBMS_LOB.FILECLOSEALL
```

## 10. 函数 FILEEXISTS

该函数用于确定 BFILE 定位符所指向的 OS 文件是否存在。语法如下：

```
DBMS_LOB.FILEEXISTS (file_loc IN BFILE) RETURN INTEGER;
```

如果文件存在，则返回 1；如果文件不存在，则返回 0。使用该函数的示例如下：

```

DECLARE
    file1 BFILE;
BEGIN
    file1:=bfilename('G','README.DOC');
    IF dbms_lob.fileexists(file1)=0 THEN
        dbms_output.put_line('文件不存在');
    ELSE
        dbms_output.put_line('文件存在');
    END IF;

```

```
END;
```

```
/
```

文件存在

### 11. 过程 FILEGETNAME

该过程用于取得 BFILE 定位符所对应的目录别名和文件名。语法如下：

```
DBMS_LOB.FILEGETNAME (
    file_loc IN BFILE,
    dir_alias OUT VARCHAR2, filename OUT VARCHAR2);
```

如上所示，dir\_alias 用于取得 BFILE 定位符所对应的目录别名；filename 用于取得 BFILE 定位符所对应的文件名。使用该过程的示例如下：

```
DECLARE
    dir_alias VARCHAR2(20);
    filename VARCHAR2(50);
    file_loc BFILE;
BEGIN
    SELECT filename INTO file_loc FROM bfile_table
        WHERE fno=&no;
    dbms_lob.filegetname(file_loc,dir_alias,filename);
    dbms_output.put_line('目录别名:'||dir_alias);
    dbms_output.put_line('文件名:'||filename);
END;
/
输入 no 的值： 1
目录别名:G
文件名:README1.DOC
```

### 12. 函数 FILEISOPEN

该函数用于确定 BFILE 所对应的 OS 文件是否已经打开。语法如下：

```
DBMS_LOB.FILEISOPEN (file_loc IN BFILE) RETURN INTEGER;
```

如果文件已经被打开，则返回 1；如果文件没有被打开，则返回 0。使用该过程的示例如下：

```
DECLARE
    file1 BFILE;
BEGIN
    file1:=bfilename('G','README1.DOC');
    IF dbms_lob.fileisopen(file1)=0 THEN
        dbms_output.put_line('文件未打开');
    ELSE
        dbms_output.put_line('文件已经打开');
    END IF;
END;
/
文件未打开
```

### 13. 过程 FILEOPEN

该过程用于打开 BFILE 所对应的 OS 文件。语法如下：

```
DBMS_LOB.FILEOPEN (
    file_loc IN OUT NOCOPY BFILE,
    open_mode IN BINARY_INTEGER := file_READONLY);
```

如上所示, `open_mode` 用于指定文件的打开模式。注意, OS 文件只能以只读方式打开。使用该过程的示例如下:

```
DECLARE
    file1 BFILE;
BEGIN
    file1:=bfilename('G','README.DOC');
    IF dbms_lob.fileexists(file1)=1 THEN
        dbms_lob.fileopen(file1);
        dbms_output.put_line('文件已经被打开');
    END IF;
    dbms_lob.fileclose(file1);
END;
/
文件已经被打开
```

#### 14. 过程 FREETEMPORARY

该过程用于释放放在默认临时表空间中的临时 LOB。语法如下:

```
DBMS_LOB.FREETEMPORARY (
    lob_loc IN OUT NOCOPY BLOB/CLOB/NCLOB);
```

如上所示, `lob_loc` 用于指定 LOB 定位符。使用该过程的示例如下:

```
DECLARE
    src_lob CLOB;
BEGIN
    dbms_lob.createtemporary(src_lob,TRUE);
    src_lob:='中华人民共和国';
    dbms_lob.freetemporary(src_lob);
END;
/
```

#### 15. 函数 GETCHUNKSIZE

当建立包含 CLOB 列或 BLOB 列的表时, 通过指定 CHUNK 参数可以指定操纵 LOB 需要分配的字节数(该值是数据块尺寸的整数倍), 如果不指定该参数, 其默认值为数据块的尺寸。通过使用函数 GETCHUNKSIZE, 可以取得 CHUNK 参数所对应的值。语法如下:

```
DBMS_LOB.GETCHUNKSIZE (lob_loc IN BLOB/CLOB/NCLOB)
RETURN INTEGER;
```

使用该函数的示例如下:

```
DECLARE
    src_lob CLOB;
    chunksizes INT;
BEGIN
    src_lob:='中华人民共和国';
    chunksizes:=dbms_lob.getchunksize(src_lob);
    dbms_output.put_line('CHUNK 尺寸:'||chunksizes);
```

```

END;
/
CHUNK 尺寸:8132

```

### 16. 函数 GETLENGTH

该函数用于取得 LOB 数据的实际长度，不仅适用于 CLOB 和 BLOB 类型，而且也适用于 BFILE 类型。语法如下：

```

DBMS_LOB.GETLENGTH (
    lob_loc IN BLOB/CLOB/NCLOB) RETURN INTEGER;

```

使用该函数的示例如下：

```

DECLARE
    file1 BFILE;
    length INT;
BEGIN
    file1:=bfilename('G','README.DOC');
    length:=dbms_lob.getlength(file1);
    dbms_output.put_line('文件长度:'||length);
END;
/
文件长度:111616

```

### 17. 函数 INSTR

该函数用于返回特定样式数据在 LOB 中从某偏移位置开始第 n 次出现时的具体位置，它不仅适用于 BLOB 和 CLOB 类型，也适用于 BFILE 类型。语法如下：

```

DBMS_LOB.INSTR (
    lob_loc IN BLOB/CLOB/NCLOB/BFILE, pattern IN RAW/VARCHAR2,
    offset IN INTEGER := 1, nth IN INTEGER := 1)
RETURN INTEGER;

```

如上所示，pattern 用于指定要搜索的字符串或二进制数据，offset 用于指定搜索的起始位置，nth 用于指定第 n 次的出现次数。使用该函数的示例如下：

```

DECLARE
    src_lob CLOB;
    location INT;
    offset INT;
    occurrence INT;
BEGIN
    src_lob:='中国，中国，伟大的中国';
    offset:=2;
    occurrence:=2;
    location:=dbms_lob.instr(src_lob,'中国',
        offset,occurrence);
    dbms_output.put_line('从第'||offset ||
        '字符开始,中国第'||occurrence||'次出现的具体位置:' ||
        location);
END;
/
从第 2 字符开始,中国第 2 次出现的具体位置:10

```

### 18. 函数 ISOPEN

该函数用于确定 LOB 是否已经被打开，既适用于 BLOB 和 CLOB 类型，也适用于 BFILE 类型。语法如下：

```
DBMS_LOB.ISOPEN (lob_loc IN BLOB/CLOB/BFILE) RETURN INTEGER;
```

如果 LOB 已经被打开，则返回 1；否则返回 0。使用该函数的示例如下：

```
DECLARE
    src_lob CLOB;
BEGIN
    src_lob:='中国，中国，伟大的中国';
    IF dbms_lob.isopen(src_lob)=0 THEN
        dbms_lob.open(src_lob,1);
    END IF;
    dbms_lob.close(src_lob);
END;
/
```

### 19. 函数 ISTEMPORARY

该函数用于确定 LOB 定位符是否为临时 LOB，语法如下：

```
DBMS_LOB.ISTEMPORARY (lob_loc IN BLOB/CLOB/NCLOB)
RETURN INTEGER;
```

如果是临时 LOB，则返回 1；否则返回 0。使用该函数的示例如下：

```
DECLARE
    src_lob CLOB;
BEGIN
    IF dbms_lob.istemporary(src_lob)=1 THEN
        dbms_output.put_line('已经是临时 LOB');
    ELSE
        dbms_output.put_line('临时 LOB 需要建立');
        dbms_lob.createtemporary(src_lob, TRUE);
    END IF;
    dbms_lob.freetemporary(src_lob);
END;
/
临时 LOB 需要建立
```

### 20. 过程 LOADFROMFILE

该过程用于将 BFILE 的部分或者全部内容复制到目标 LOB 变量（CLOB 或者 BLOB）中。语法如下：

```
DBMS_LOB.LOADFROMFILE (
    dest_lob IN OUT NOCOPY BLOB/CLOB,
    src_file IN BFILE,amount IN INTEGER,
    dest_offset IN INTEGER := 1,
    src_offset IN INTEGER := 1);
```

如上所示，src\_file 用于指定 BFILE 定位符，注意，当使用该过程将 BFILE 数据装载到 CLOB 中时，不会进行字符集转换，因此要确保 BFILE 数据与数据库具有相同字符集，否则装载后的数据为乱码。使用该过程的示例如下：

```

DECLARE
    src_lob BFILE;
    dest_lob CLOB;
    amount INT;
BEGIN
    src_lob:=bfilename('G','A.TXT');
    dbms_lob.createtemporary(dest_lob,TRUE);
    dbms_lob.fileopen(src_lob,0);
    amount:=dbms_lob.getlength(src_lob);
    dbms_lob.loadfromfile(dest_lob,src_lob,amount);
    dbms_lob.fileclose(src_lob);
    dbms_lob.freetemporary(dest_lob);
END;
/

```

## 21. 过程 LOADBLOBFROMFILE

该过程用于将 BFILE 数据装载到 BLOB 中，并且在装载后可以取得新的偏移位置。语法如下：

```

DBMS_LOB.LOADBLOBFROMFILE (
    dest_lob IN OUT NOCOPY BLOB,src_bfile IN BFILE,
    amount IN INTEGER,dest_offset IN OUT INTEGER,
    src_offset IN OUT INTEGER);

```

如上所示，`src_bfile` 用于指定 BFILE 定位符；`dest_offset (IN)` 用于指定目标 LOB 的起始位置；`dest_offset (OUT)` 用于取得装载完成后的偏移位置；`src_offset (IN)` 用于指定 BFILE 定位符的起始位置；`src_offset (OUT)` 用于取得 BFILE 读取完成后的偏移位置。使用该过程的示例如下：

```

DECLARE
    src_lob BFILE;
    dest_lob BLOB;
    amount INT;
    src_offset INT:=1;
    dest_offset INT:=1;
BEGIN
    src_lob:=bfilename('G','A.TXT');
    dbms_lob.createtemporary(dest_lob,TRUE);
    dbms_lob.fileopen(src_lob,0);
    amount:=dbms_lob.getlength(src_lob);
    dbms_lob.loadblobfromfile(dest_lob,src_lob,
        amount,dest_offset,src_offset);
    dbms_lob.fileclose(src_lob);
    dbms_lob.freetemporary(dest_lob);
    dbms_output.put_line('新的偏移位置:'||dest_offset);
END;
/
新的偏移位置:12

```

## 22. 过程 LOADCLOBFROMFILE

该过程用于将 BFILE 数据装载到 CLOB 中。当使用该过程装载数据到 CLOB 中时，可以指定字符集 ID 号，并进行字符集转换。因此，建议在装载 BFILE 数据到 CLOB 中时使用该过程。语法如下：

```
DBMS_LOB.LOADCLOBFROMFILE (
    dest_lob IN OUT NOCOPY CLOB,
    src_bfile IN BFILE, amount IN INTEGER,
    dest_offset IN OUT INTEGER,
    src_offset IN OUT INTEGER,
    src_csid IN NUMBER,
    lang_context IN OUT INTEGER,
    warning OUT INTEGER);
```

如上所示，src\_csid 用于指定源文件的字符集 ID 号；lang\_context (IN) 用于指定语言上下文；lang\_context (OUT) 用于取得先前装载的语言上下文；warning 用于取得警告消息。使用该过程的示例如下：

```
DECLARE
    src_lob BFILE;
    dest_lob CLOB;
    amount INT;
    src_offset INT:=1;
    dest_offset INT:=1;
    csid INT:=0;
    lc INT:=0;
    warning INT;
BEGIN
    src_lob:=bfilename('G','A.TXT');
    dbms_lob.createtemporary(dest_lob,TRUE);
    dbms_lob.fileopen(src_lob,0);
    amount:=dbms_lob.getlength(src_lob);
    dbms_lob.loadclobfromfile(dest_lob,src_lob,
        amount,dest_offset,src_offset,csid,lc,warning);
    dbms_lob.fileclose(src_lob);
    dbms_output.put_line(dest_lob);
    dbms_lob.freetemporary(dest_lob);
END;
/
中国，中国，伟大的中国
```

## 23. 过程 OPEN

该过程用于在打开 LOB 时指定 LOB 的读写模式：只读 (dbms\_lob.blob\_READONLY)、读写 (dbms\_lob.blob\_READWRITE)，不仅适用于 BLOB 和 CLOB，也适用于 BFILE。语法如下：

```
DBMS_LOB.OPEN (
    lob_loc IN OUT NOCOPY BLOB/CLOB/BFILE,
    open_mode IN BINARY_INTEGER);
```

如上所示，open\_mode 用于指定 LOB 的读写模式。使用该过程的示例如下：

```

DECLARE
    src_lob CLOB;
    v1 VARCHAR2(100):='中华人民共和国';
    amount INT;
BEGIN
    amount:=length(v1);
    dbms_lob.createtemporary(src_lob,TRUE);
    dbms_lob.open(src_lob,dbms_lob lob_readwrite);
    dbms_lob.write(src_lob,amount,1,v1);
    dbms_lob.close(src_lob);
    dbms_output.put_line(src_lob);
    dbms_lob.freetemporary(src_lob);
END;
/
中华人民共和国

```

#### 24. 过程 READ

该过程用于将 LOB 数据读取到缓冲区中，不仅适用于 BLOB 和 CLOB，也适用于 BFILE。语法如下：

```

DBMS_LOB.READ (
    lob_loc IN BLOB/CLOB/BFILE,amount IN OUT NOCOPY BINARY_INTEGER,
    offset IN INTEGER,buffer OUT RAW/VARCHAR2);

```

如上所示，amount(IN)用于指定要读取的字节个数(BLOB)或字符个数(CLOB)，amount(OUT)用于取得实际读取的字节个数或字符个数；使用该过程的示例如下：

```

DECLARE
    src_lob CLOB:='伟大的中国';
    amount INT;
    buffer VARCHAR2(200);
    offset INT:=1;
BEGIN
    amount:=dbms_lob.getlength(src_lob);
    dbms_lob.open(src_lob,DBMS_LOB.LOB_READONLY);
    dbms_lob.read(src_lob,amount,offset,buffer);
    dbms_output.put_line(buffer);
    dbms_lob.close(src_lob);
END;
/
伟大的中国

```

#### 25. 函数 SUBSTR

该函数用于返回 LOB 中从指定位置开始的部分内容，不仅适用于 BLOB 和 CLOB，也适用于 BFILE。语法如下：

```

DBMS_LOB.SUBSTR (
    lob_loc IN BLOB/CLOB/BFILE,amount IN INTEGER := 32767,
    offset IN INTEGER := 1)
RETURN RAW;

```

使用该函数的示例如下：

```

DECLARE
    src_lob CLOB:='中国，中国，伟大的中国';
    amount INT;
    v1 VARCHAR2(200);
    offset INT;
BEGIN
    amount:=10;
    offset:=4;
    v1:=dbms_lob.substr(src_lob,amount,offset);
    dbms_output.put_line(v1);
END;
/
中国，伟大的中国

```

## 26. 过程 TRIM

该过程用于截断 LOB 内容到指定长度，只适用于 BLOB 和 CLOB，而不适用于 BFILE。语法如下：

```

DBMS_LOB.TRIM (
    lob_loc IN OUT NOCOPY BLOB/CLOB/NCLOB,newlen IN INTEGER);

```

如上所示，newlen 用于指定截断后的 LOB 长度。使用该过程的示例如下：

```

DECLARE
    src_lob CLOB:='中国，中国，伟大的中国';
    amount INT;
BEGIN
    amount:=5;
    dbms_lob.trim(src_lob,amount);
    dbms_output.put_line(src_lob);
END;
/
中国，中国

```

## 27. 过程 WRITE

该过程用于将缓冲区数据写入到 LOB 中的特定位置，只适用于 BLOB 和 CLOB，而不适用于 BFILE。语法如下：

```

DBMS_LOB.WRITE (
    lob_loc IN OUT NOCOPY BLOB/CLOB, amount IN BINARY_INTEGER,
    offset IN INTEGER,buffer IN RAW/VARCHAR2);

```

如上所示，buffer 用于指定要写入的内容。使用该过程的示例如下：

```

DECLARE
    src_lob CLOB:='我的祖国';
    amount INT;
    offset INT;
    buffer VARCHAR2(100):='，伟大的中国';
BEGIN
    offset:=dbms_lob.getlength(src_lob)+1;
    amount:=length(buffer);
    dbms_lob.write(src_lob,amount,offset,buffer);

```

```

    dbms_output.put_line(src_lob);
END;
/
我的祖国，伟大的中国

```

## 28. 过程 WRITEAPPEND

该过程用于将缓冲区数据写入到LOB尾部,只适用于BLOB和CLOB,而不适用于BFILE。  
语法如下:

```

DBMS_LOB.WRITEAPPEND (
    lob loc IN OUT NOCOPY BLOB/CLOB/NCLOB,
    amount IN BINARY_INTEGER,buffer IN RAW);

```

使用该过程的示例如下:

```

DECLARE
    src_lob CLOB:='我的祖国';
    amount INT;
    buffer VARCHAR2(100):='，伟大的中国';
BEGIN
    amount:=length(buffer);
    dbms_lob.writeappend(src_lob,amount,buffer);
    dbms_output.put_line(src_lob);
END;
/
我的祖国，伟大的中国

```

## 16.3 访问 LOB

编写 LOB 应用程序时,需要访问 LOB 列的数据,LOB 列的数据是使用 LOB 定位符来访问的。当 BLOB 和 CLOB 列数据长度小于 4000 字节时,其数据被存放在行内;当它们的数据长度大于 4000 字节时,其数据被存放在专门的 LOB 段中。在编写 PL/SQL 块时,使用包 DBMS\_LOB 可以访问 LOB 数据。

### 16.3.1 访问 CLOB

CLOB 用于存放大批量的文本数据,所允许的最大数据长度为 4G 字节。在 Oracle8 版本之前,存放大批量文本数据使用 LONG 来完成,它所允许的数据最大长度为 2G 字节。在一个表上允许有多个 CLOB 类型的列,但只允许一个 LONG 类型的列。另外,无论是性能方面还是管理方面,CLOB 类型都比 LONG 类型更加优越,所以建议开发人员选择 CLOB 类型。下面通过一个完整示例说明使用 CLOB 类型的方法。

#### 1. 建立包含 CLOB 列的表

当在表中需要定义包含大批量文本数据的列时,应该选择 CLOB 类型。下面以建立表 lob\_example1 为例,说明定义 CLOB 列的方法。示例如下:

```

SQL> CREATE TABLE lob_example1(
  2      id NUMBER(6) PRIMARY KEY,
  3      name VARCHAR2(10),resume CLOB

```

```
4 );
```

## 2. 初始化 CLOB 列

在建立了表 lob\_example1 之后，就可以为其插入数据了。当为 CLOB 列插入数据时，可以使用函数 EMPTY\_CLOB() 初始化 CLOB 列。注意，函数 EMPTY\_CLOB() 与 NULL 不同，尽管该函数没有提供任何数据，但却分配了 LOB 定位符。为 lob\_example1 表插入数据并初始化 CLOB 列的示例如下：

```
SQL> INSERT INTO lob_example1 VALUES(1,'王鸣',EMPTY_CLOB());
SQL> INSERT INTO lob_example1 VALUES(2,'马丽',EMPTY_CLOB());
SQL> COMMIT;
```

## 3. 更新 CLOB 列的数据

因为 CLOB 列用于存放大批量文本数据，所以不可能一次写入所有文本数据，而通常的作法是在已有数据的基础上追加新数据，为 CLOB 列追加数据可以使用包 DBMS\_LOB 中的过程 WRITE 和 WRITEAPPEND 来完成。注意，如果要更新 CLOB 列的数据，那么在检索 CLOB 列时就必须带有 FOR UPDATE 子句。下面以给 resume 列添加数据为例，说明更新 CLOB 列的数据的方法。示例如下：

```
DECLARE
    lob_loc CLOB;
    text VARCHAR2(200);
    amount INT;
    offset INT;
BEGIN
    SELECT resume INTO lob_loc FROM lob_example1
    WHERE id=&id FOR UPDATE;
    offset:=DBMS_LOB.GETLENGTH(lob_loc)+1;
    text:='&resume';
    amount:=LENGTH(text);
    DBMS_LOB.WRITE(lob_loc,amount,offset,text);
    COMMIT;
END;
/
输入 id 的值： 1
输入 resume 的值： 1998 年毕业于哈尔滨工业大学
```

在执行了以上 PL/SQL 块之后，会根据输入的 ID 号为人员添加简历，并且新内容会追加到原有内容之后。

## 4. 读取 CLOB 列的内容

因为 CLOB 列包含了大批量文本数据（最大可达到 4G），所以在实际应用环境中单次读取其所有数据可能会有一些问题（缓冲区尺寸不足）。因此，为了读取 CLOB 列的所有数据，应该使用循环方式进行处理。当读取 CLOB 列的数据时，应该使用包 DBMS\_LOB 的过程 READ 来完成。示例如下：

```
DECLARE
    lob_loc CLOB;
    buffer VARCHAR2(200);
    amount INT;
```

```

offset INT;
BEGIN
    SELECT resume INTO lob_loc FROM lob_example1
        WHERE id=&id;
    offset:=6;
    amount:=DBMS_LOB.GETLENGTH(lob_loc);
    DBMS_LOB.READ(lob_loc,amount,offset,buffer);
    dbms_output.put_line(buffer);
END;
/
输入 id 的值: 1
毕业于哈尔滨工业大学

```

在执行了以上 PL/SQL 块之后，就会读取 CLOB 列的部分数据（第 6 个字符开始的所有字符），并显示这些数据。

### 5. 将文本文件内容写入到 CLOB 列

在实际应用中使用 CLOB 列时，可能经常需要将文本文件内容写入到 CLOB 列中。在开发 PL/SQL 应用程序时，使用过程 LOADFROMFILE 或 LOADCLOBFROMFILE 可以完成这项任务。为了避免字符集问题，建议使用 LOADCLOBFROMFILE。下面以将文件“马丽.txt”的内容写入到 RESUME 列为例，说明将文本文件内容写入到 CLOB 列的方法。示例如下：

```

DECLARE
    lobloc CLOB;
    fileloc BFILE;
    amount INT;
    src_offset INT:=1;
    dest_offset INT:=1;
    csid INT:=0;
    lc INT:=0;
    warning INT;
BEGIN
    fileloc:=bfilename('G','马丽.TXT');
    DBMS_LOB.FILEOPEN(fileloc,0);
    amount:=DBMS_LOB.GETLENGTH(fileloc);
    SELECT resume INTO lobloc FROM lob_example1
        WHERE id=2 FOR UPDATE;
    DBMS_LOB.LOADCLOBFROMFILE(lobloc,fileloc,amount,
        dest_offset,src_offset,csid,lc,warning);
    DBMS_LOB.FILECLOSE(fileloc);
    COMMIT;
END;
/

```

### 6. 将 CLOB 列内容写入到文本文件

当要将 CLOB 列的内容写入到文本文件时，不仅需要使用 DBMS\_LOB 包读取 CLOB 列的内容，而且需要使用 UTL\_FILE 包建立文本文件并写入内容。下面以将 RESUME 列的内容写入到文本文件 g:\file\aa.txt 中为例，说明将 CLOB 列的内容写入到文本文件的方法。示例如下：

```

DECLARE
    lobloc CLOB;
    amount INT;
    offset INT:=1;
    buffer VARCHAR2(2000);
    handle UTL_FILE.FILE_TYPE;
BEGIN
    SELECT resume INTO lobloc FROM lob_example1
        WHERE id=&id;
    amount:=dbms_lob.getlength(lobloc);
    dbms_lob.read(lobloc,amount,offset,buffer);
    handle:=utl_file.fopen('USER_DIR','a.txt','w',2000);
    utl_file.put_line(handle,buffer);
    utl_filefclose(handle);
END;
/
输入 id 的值: 1
PL/SQL 过程已成功完成。

```

### 16.3.2 访问 BLOB

BLOB 用于存放大批量的二进制数据，它所允许的数据最大长度为 4G 字节。在 Oracle8 版本之前，存放大批量的二进制数据是使用 LONG RAW 类型来完成的，它所允许的数据最大长度为 2G 字节。在一个表上允许有多个 BLOB 类型的列，但只允许一个 LONG RAW 类型的列；另外，无论是性能方面还是管理方面，BLOB 类型都比 LONG RAW 类型更加优越，所以建议开发人员选择 BLOB 类型。下面通过一个完整示例说明使用 BLOB 类型的方法。

#### 1. 建立包含 BLOB 列的表

当在表中定义存放大批量二进制数据（例如图形/图象数据）的列时，应该选择 BLOB 类型。下面以建立 lob\_example2 表为例，说明定义 BLOB 列的方法。示例如下：

```

SQL> CREATE TABLE lob_example2 (
  2   id NUMBER(6) PRIMARY KEY,
  3   name VARCHAR2(10),photo BLOB
  4 );

```

#### 2. 初始化 BLOB 列

在建立了表 lob\_example2 之后，就可以为其插入数据了。当在 BLOB 列中插入数据时，可以使用函数 EMPTY\_BLOB() 初始化 BLOB 列。注意，函数 EMPTY\_BLOB() 与 NULL 不同，尽管该函数没有提供任何数据，但却分配了 LOB 定位符。在 lob\_example2 表中插入数据并初始化 BLOB 列的示例如下：

```

SQL> INSERT INTO lob_example2 VALUES(1,'王鸣',EMPTY_BLOB());
SQL> INSERT INTO lob_example2 VALUES(2,'马丽',EMPTY_BLOB());
SQL> COMMIT;

```

#### 3. 将二进制文件内容写入 BLOB 列

因为 BLOB 列用于存放二进制数据，所以其数据的写入往往是通过写入文件内容来实现。为了完成这项任务，大家可以使包 DBMS\_LOB 的过程 LGADBLOBFROMFILE 来完成。下

面以将图象文件“马丽.bmp”内容写入到 PHOTO 列为例，说明将 BLOB 列的数据写入到二进制文件中的方法。示例如下：

```

DECLARE
    lobloc BLOB;
    fileloc BFILE;
    amount INT;
    src_offset INT:=1;
    dest_offset INT:=1;
BEGIN
    SELECT photo INTO lobloc FROM lob_example2
        WHERE id=&id FOR UPDATE;
    fileloc:=bfilename('G','&filename');
    DBMS_LOB.FILEOPEN(fileloc,0);
    amount:=DBMS_LOB.GETLENGTH(fileloc);
    DBMS_LOBLOADBLOBFROMFILE(lobloc,fileloc,amount,
        dest_offset,src_offset);
    DBMS_LOB.FILECLOSE(fileloc);
    COMMIT;
END;
/
输入 id 的值： 1
输入 filename 的值： 马丽.BMP

```

#### 4. 读取 BLOB 列数据

因为 BLOB 列中存放着二进制数据，所以当读取其数据时应该使用 RAW 变量接收其数据。

读取 BLOB 列的数据可以使用包 DBMS\_LOB 的过程 READ 来完成。示例如下：

```

DECLARE
    lobloc BLOB;
    buffer RAW(2000);
    amount INT;
    offset INT:=1;
BEGIN
    SELECT photo INTO lobloc FROM lob_example2
        WHERE id=&id;
    amount:=DBMS_LOB.GETLENGTH(lobloc);
    DBMS_LOB.READ(lobloc,amount,offset,buffer);
END;
/
输入 id 的值： 1

```

在执行了以上 PL/SQL 块之后，就会将 BLOB 列的内容读取到变量 buffer 中，但要求 PHOTO 列的数据必须小于 2000 字节。如果大于 2000 字节，那么需要使用循环方式读取数据。

#### 5. 将 BLOB 列的内容写入到二进制文件

当要将 BLOB 列的内容写入到二进制文件时，不仅需要使用 DBMS\_LOB 包读取 BLOB 列的内容，而且需要使用 UTL\_FILE 包建立二进制文件并写入内容。下面以将 PHOTO 列的内容写入到二进制文件 g:\file\abmp 中为例，说明将 BLOB 列的内容写入到二进制文件中的方法。

示例如下：

```

DECLARE
    lobloc BLOB;
    amount INT;
    offset INT:=1;
    buffer RAW(1000);
    handle UTL_FILE.FILE_TYPE;
BEGIN
    SELECT photo INTO lobloc FROM lob_example2
        WHERE id=&id;
    amount:=dbms_lob.getlength(lobloc);
    dbms_lob.read(lobloc,amount,offset,buffer);
    handle:=utl_file.fopen('USER_DIR','a.bmp','w',1000);
    utl_file.put_raw(handle,buffer);
    utl_filefclose(handle);
END;
/
输入 id 的值： 1
PL/SQL 过程已成功完成。

```

### 16.3.3 访问 BFILE

BFILE 是外部 LOB 类型，存放着指向 OS 文件的指针，并且其所对应的文件长度不能超过 4G。为了在数据库中访问 OS 文件的数据，建议使用 BFILE 类型。但是注意，BFILE 所对应的 OS 文件内容只能读取，而不能修改。当在 Oracle 数据库中使用 BFILE 类型访问 OS 文件时，必须首先建立 DIRECTORY 对象，而建立 DIRECTORY 对象则要求用户必须具有 CREATE ANY DIRECTORY 权限。为用户授予 CREATE ANY DIRECTORY 权限，并建立 DIRECTORY 对象的示例如下：

```

SQL> conn system/manager
SQL> GRANT CREATE ANY DIRECTORY TO scott;
SQL> conn scott/tiger
SQL> CREATE DIRECTORY bfile_dir AS 'G:\BFILE_EXAMPLE';

```

注意，当建立 DIRECTORY 对象时，一定要确保 OS 目录已经存在。如果 OS 目录不存在，那么在建立 DIRECTORY 对象时不会显示错误，但当依据该 DIRECTORY 对象访问 OS 文件时会提示错误信息。在建立了 DIRECTORY 对象之后，就可以使用 BFILE 类型访问 OS 文件。下面通过一个完整示例说明使用 BFILE 类型的方法。

#### 1. 建立包含 BFILE 列的表

当要在表中访问 OS 文件中的内容时，需要使用 BFILE 类型。下面以建立表 lob\_example3 为例，说明定义 BFILE 列的方法。示例如下：

```

SQL> CREATE TABLE lob_example3(
  2   id NUMBER(6) PRIMARY KEY,
  3   name VARCHAR2(10),resume BFILE
  4 );

```

## 2. 初始化 BFILE 列

在建立了表 lob\_example3 之后，就可以为其插入数据。当为 BFILE 列插入数据时，可以使用函数 BFILENAME() 来初始化 BFILE 列。注意，当使用 BFILENAME() 函数时，DIRECTORY 对象必须使用大写格式。为 lob\_example3 表插入数据并初始化 BFILE 列的示例如下：

```
SQL> INSERT INTO lob_example3
  2  VALUES(1,'王鸣',bfilename('BFILE_DIR','王鸣.TXT'));
SQL> INSERT INTO lob_example3
  2  VALUES(2,'马丽',bfilename('BFILE_DIR','马丽.TXT'));
SQL> COMMIT;
```

## 3. 读取 BFILE 列的内容

为了读取 BFILE 列中的内容，可以使用 DBMS\_LOB 包的 READ 过程。当读取 BFILE 列中的数据时，应该使用 RAW 变量接收其读出的数据。示例如下：

```
DECLARE
    buffer RAW(2000);
    amount INT;
    offset INT;
    lobloc BFILE;
BEGIN
    SELECT resume INTO lobloc FROM lob_example3
    WHERE id=&id;
    DBMS_LOB.FILEOPEN(lobloc,0);
    amount:=DBMS_LOB.GETLENGTH(lobloc);
    offset:=1;
    DBMS_LOB.READ(lobloc,amount,offset,buffer);
    DBMS_LOB.FILECLOSE(lobloc);
END;
/
输入 id 的值： 2
```

## 16.4 习题

1. 在表上可以有一个 LONG 列、一个 LONG RAW 列吗？
  - A. 可以
  - B. 不可以
2. 在表上可以有多个 LOB 列吗？
  - A. 可以
  - B. 不可以
3. 以下哪种 LOB 列的内容不能修改？
  - A. CLOB
  - B. BLOB
  - C. BFILE
  - D. NCLOB
4. 建立表 employee，其列定义如下所示：

列名	数据类型	约束	作用
ID	NUMBER(6)	PRIMARY KEY	雇员编号
NAME	VARCHAR2(10)		雇员姓名

续表

列名	数据类型	约束	作用
SAL	NUMBER(6,2)		雇员工资
DNO	NUMBER(2)		雇员所在部门号
RESUME	CLOB		雇员简历
PHOTO	BLOB		雇员照片

5. 为 EMPLOYEE 表插入以下两条数据，并分别初始化 CLOB 列和 BLOB 列：

- 第一条数据：1（雇员号）、秦明（姓名）、2000（工资）、部门号（10）；
- 第二条数据：2（雇员号）、林冲（姓名）、3000（工资）、部门号（20）；

6. 以 SYSTEM 用户登录并建立 DIRECTORY 对象 INFO (OS 目录: g:\info)，然后将读写该 DIRECTORY 对象的权限授予 SCOTT 用户。

7. 将 G 盘 info 目录下文本文件 qinming.txt 和 linchong.txt 的内容灌入到 RESUME 列中。文件内容如下：

- g:\info\qinming.txt:

秦明，男，1975 年出生于内蒙古自治区巴彦淖尔盟乌拉特前旗，1993 年考入内蒙古大学计算机科学与工程系，1997 年大学本科毕业，并于同年分配到航天工业总公司第六研究院工作。

- g:\info\linchong.txt:

林冲，男，1972 年出生于内蒙古自治区巴彦淖尔盟乌拉特前旗，1990 年考入内蒙古工业大学自动控制系，1994 年大学本科毕业，并于同年分配到航天工业总公司第六研究院工作。

8. 读取 RESUME 列中的内容，并将其分别写入到文本文件 qm.txt 和 lch.txt 中。

9. 将 G 盘 info 目录下的照片文件 qinming.bmp 和 linchong.bmp 的内容写入到 PHOTO 列中，这两个文件包含了秦明和林冲的照片

10. 读取 PHOTO 列中的内容，并将其分别写入到 qm.bmp 和 lch.bmp 文件中。

# 第 17 章 使用 Oracle 系统包

为了扩展数据库的功能，Oracle 提供了大量的 PL/SQL 系统包。建立应用系统时，可以直接使用 Oracle 系统包所提供的过程和函数。在 Oracle 10g 中，Oracle 提供的系统包多达几百个，每个系统包都用于完成特定的功能，本章将给读者介绍一些常用的 Oracle 系统包。

## 17.1 DBMS\_OUTPUT

DBMS\_OUTPUT 包用于输入和输出信息，使用过程 PUT 和 PUT\_LINES 可以将信息发送到缓冲区，使用过程 GET\_LINE 和 GET\_LINES 可以显示缓冲区信息。下面将详细介绍该包所包含的过程和函数。

### 1. ENABLE

该过程用于激活对过程 PUT, PUT\_LINE, NEW\_LINE, GET\_LINE 和 GET\_LINES 的调用。如果 DBMS\_OUTPUT 包没有被激活，那么将会忽略对这些过程的调用。注意，如果在 SQL\*Plus 中使用 SERVEROUTPUT 选项，则没有必要使用该过程。当调用该过程时，缓冲区最大尺寸为 1000000 字节，最小尺寸为 2000 字节，并且默认尺寸为 20000 字节。使用该过程的语法如下：

```
DBMS_OUTPUT.ENABLE (buffer_size IN INTEGER DEFAULT 20000);
```

### 2. DISABLE

该过程用于禁止对过程 PUT, PUT\_LINE, NEW\_LINE, GET\_LINE 和 GET\_LINES 的调用。注意，如果在 SQL\*Plus 中使用 SERVEROUTPUT 选项，则没有必要使用该过程。语法如下：

```
DBMS_OUTPUT.DISABLE;
```

### 3. PUT 和 PUT\_LINE

过程 PUT\_LINE 用于将一个完整行的信息写入到缓冲区中，过程 PUT 则用于分块建立行信息。当使用过程 PUT\_LINE 时，会自动在行的尾部追加行结束符；当使用过程 PUT 时，需要使用过程 NEW\_LINE 追加行结束符。语法如下：

```
DBMS_OUTPUT.PUT (item IN NUMBER);
DBMS_OUTPUT.PUT (item IN VARCHAR2);
DBMS_OUTPUT.PUT (item IN DATE);
DBMS_OUTPUT.PUT_LINE (item IN NUMBER);
DBMS_OUTPUT.PUT_LINE (item IN VARCHAR2);
DBMS_OUTPUT.PUT_LINE (item IN DATE);
```

如上所示，item 用于指定要输出的内容。当在 SQL\*Plus 中使用过程 PUT 和 PUT\_LINE 时，需要设置 SERVEROUTPUT 选项。示例如下：

```
SET SERVEROUTPUT ON
BEGIN
  DBMS_OUTPUT.PUT_LINE('伟大的中华民族');
  DBMS_OUTPUT.PUT('中国');
```

```

DBMS_OUTPUT.PUT('伟大的祖国');
DBMS_OUTPUT.NEW_LINE;
END;
/
伟大的中华民族
中国,伟大的祖国

```

#### 4. NEW\_LINE

该过程用于在行的尾部追加行结束符。当使用过程 PUT 时，必须调用 NEW\_LINE 过程来结束行。语法如下：

```
DBMS_OUTPUT.NEW_LINE;
```

#### 5. GET\_LINE 和 GET\_LINES

过程 GET\_LINE 用于取得缓冲区的单行信息，过程 GET\_LINES 用于取得缓冲区的多行信息。语法如下：

```

DBMS_OUTPUT.GET_LINE(line OUT VARCHAR2,status OUT INTEGER);
DBMS_OUTPUT.GET_LINES(lines OUT CHARARR,numlines IN OUT INTEGER);

```

如上所示，line 用于取得缓冲区的单行信息（最大 255 字节），status 用于确定过程执行是否成功，返回 0 表示执行成功，返回 1 则表示没有行；lines 用于取得缓冲区的多行信息，numlines 用于指定要检索的行数，并返回实际检索的行数。当在 SQL\*Plus 中输出信息时，可以直接使用 SERVEROUTPUT 选项；而在其他应用程序中使用 DBMS\_OUTPUT 输出信息时，需要使用过程 GET\_LINE 和 GET\_LINES。

#### 示例一：使用 GET\_LINE

```

VAR line VARCHAR2(100)
VAR status NUMBER
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('伟大的中华民族');
    DBMS_OUTPUT.PUT('中国');
    DBMS_OUTPUT.PUT('伟大的祖国');
    DBMS_OUTPUT.NEW_LINE;
    DBMS_OUTPUT.GET_LINE(:line,:status);
END;
/
中国,伟大的祖国

```

#### 示例二：使用 GET\_LINES

因为 GET\_LINE 只能输出单行信息，所以只需要定义一个输出变量就可以接收其数据。但因为 GET\_LINES 需要输出多行信息，所以必须要定义集合类型（索引表、嵌套表或 VARRAY）来接收其多行数据。示例如下：

```

DECLARE
    TYPE line_table_type IS TABLE OF VARCHAR2(255)
    INDEX BY BINARY_INTEGER;
    line_table line_table_type;
    lines NUMBER(38):=3;
BEGIN

```

```

DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.PUT_LINE('伟大的中华民族');
DBMS_OUTPUT.PUT('中国');
DBMS_OUTPUT.PUT('，伟大的祖国');
DBMS_OUTPUT.NEW_LINE;
DBMS_OUTPUT.GET_LINES(line_table,lines);
END;
/

```

## 17.2 DBMS\_JOB

DBMS\_JOB 包用于安排和管理作业队列。通过使用作业，可以使 Oracle 数据库定期执行特定的任务。注意，当使用 DBMS\_JOB 管理作业时，必须确保设置了初始化参数 JOB\_QUEUE\_PROCESSES（不能为 0）。下面详细介绍包 DBMS\_JOB 所包含的过程和函数。

### 1. SUBMIT

该过程用于建立一个新作业。当建立作业时，需要给出作业要执行的操作、作业的下次运行日期以及运行时间间隔。语法如下：

```

DBMS_JOB.SUBMIT (
    job OUT BINARY_INTEGER, what IN VARCHAR2,
    next_date IN DATE DEFAULT sysdate,
    interval IN VARCHAR2 DEFAULT 'null',
    no_parse IN BOOLEAN DEFAULT FALSE,
    instance IN BINARY_INTEGER DEFAULT any_instance,
    force IN BOOLEAN DEFAULT FALSE);

```

如上所示，job 用于指定作业编号；what 用于指定作业要执行的操作；next\_date 用于指定作业的下次运行日期；interval 用于指定运行作业的时间间隔；no\_parse 用于指定是否解析与作业相关的过程；instance 用于指定哪个例程可以运行作业；force 用于指定是否强制运行与作业相关的例程。下面以建立用于分析 SCOTT.EMP 表的作业为例，说明建立作业的方法。示例如下：

```

VAR jobno NUMBER
BEGIN
    DBMS_JOB.SUBMIT(:jobno,
        'dbms_ddl.analyze_object(''TABLE'',
        ''SCOTT'', ''EMP'', ''COMPUTE'')',
        SYSDATE, 'SYSDATE + 1');
    commit;
END;
/
PRINT jobno
    JOBNO
-----

```

## 2. REMOVE

该过程用于删除作业队列中的特定作业。语法如下：

```
DBMS_JOB.REMOVE (job IN BINARY_INTEGER);
```

下面以删除作业 1 为例，说明使用该过程的方法。示例如下：

```
SQL> exec dbms_job.remove(1)
```

## 3. CHANGE

该过程用于改变与作业相关的所有信息，包括作业操作、作业运行日期以及运行时间间隔等。语法如下：

```
DBMS_JOB.CHANGE (
    job IN BINARY_INTEGER, what IN VARCHAR2,
    next_date IN DATE, interval IN VARCHAR2,
    instance IN BINARY_INTEGER DEFAULT NULL,
    force IN BOOLEAN DEFAULT FALSE);
```

下面以改变作业的运行时间间隔为例，说明使用该过程的方法。示例如下：

```
SQL> exec dbms_job.change(2,null,null,'SYSDATE+2')
```

## 4. WHAT

该过程用于改变作业要执行的操作。语法如下：

```
DBMS_JOB.WHAT (job IN BINARY_INTEGER, what IN VARCHAR2);
```

下面以改变作业 2 的运行操作为例，说明使用 WHAT 过程的方法。示例如下：

```
SQL> exec dbms_job.what(2,'dbms_stats.gather_table_stats-
> (''SCOTT'', ''EMP'');')
```

## 5. NEXT\_DATE

该过程用于改变作业的下次运行日期。语法如下：

```
DBMS_JOB.NEXT_DATE (job IN BINARY_INTEGER, next_date IN DATE);
```

下面以改变作业 2 的下次运行日期为例，说明使用过程 NEXT\_DATE 的方法。示例如下：

```
SQL> exec dbms_job.next_date('2','SYSDATE+1')
```

## 6. INSTANCE

该过程用于改变运行作业的例程。语法如下：

```
DBMS_JOB.INSTANCE (
    job IN BINARY_INTEGER, instance IN BINARY_INTEGER,
    force IN BOOLEAN DEFAULT FALSE);
```

下面以改变运行作业 2 的例程为例，说明使用过程 INSTANCE 的方法。示例如下：

```
SQL> exec dbms_job.instance(2,1)
```

## 7. INTERVAL

该过程用于改变作业的运行时间间隔。语法如下：

```
DBMS_JOB.INTERVAL (job IN BINARY_INTEGER, interval IN VARCHAR2);
```

下面以改变作业 2 的运行时间间隔为例，说明使用过程 INTERVAL 的方法。示例如下：

```
SQL> exec dbms_job.interval(2,'SYSDATE+1/24/60')
```

## 8. BROKEN

该过程用于设置作业的中断标记。当中断了作业之后，作业将不会被运行。语法如下：

```
DBMS_JOB.BROKEN (
    job IN BINARY_INTEGER, broken IN BOOLEAN,
```

```
next_date IN DATE DEFAULT SYSDATE);
```

如上所示, `broken` 用于指定中断标记 (TRUE 表示中断); 下面以改变作业 2 的中断标记为例, 说明使用过程 `BROKEN` 的方法。示例如下:

```
SQL> exec dbms_job.broken(2,TRUE,'SYSDATE+1')
```

## 9. RUN

该过程用于运行已存在的作业。语法如下:

```
DBMS_JOB.RUN (
    job IN BINARY_INTEGER, force IN BOOLEAN DEFAULT FALSE);
```

下面以运行作业 2 为例, 说明使用过程 `RUN` 的方法。示例如下:

```
SQL> exec dbms_job.run(2)
```

## 10. 作业使用示例

当在 Oracle 数据库中使用作业时, 应该首先使用过程 `SUBMIT` 来建立作业, 然后使用过程 `RUN` 来运行作业。下面以每天搜集 SCOTT 方案的所有对象统计为例, 说明在 Oracle 数据库中使用作业的方法。

### (1) 建立作业

```
VAR jobno NUMBER
BEGIN
    DBMS_JOB.SUBMIT(:jobno,
        'dbms_stats.gather_schema_stats('''SCOTT''');',
        SYSDATE, 'SYSDATE + 1');
    COMMIT;
END;
/
PL/SQL 过程已成功完成。
SQL> PRINT jobno
      JOBNO
-----
      3
```

### (2) 运行作业

```
SQL> exec dbms_job.run(3)
PL/SQL 过程已成功完成。
```

## 17.3 DBMS\_PIPE

包 `DBMS_PIPE` 用于在同一例程的不同会话之间进行管道通信。Oracle 管道 (Pipe) 类似于 UNIX 系统的管道, 但它不是采用操作系统机制实现的, 其管道信息被缓存在 SGA 中, 当关闭例程时会丢失管道信息。在建立管道时, 既可以建立公用管道, 也可以建立私有管道。其中, 公用管道是指所有数据库用户都可以访问的管道, 而私有管道只能由建立管道的数据用户访问。注意, 如果用户要执行包 `DBMS_PIPE` 中的过程和函数, 则必须要为用户授权。示例如下:

```
SQL> conn sys/oracle as sysdba
SQL> GRANT EXECUTE ON dbms_pipe TO scott;
```

### 1. CREATE\_PIPE

该函数用于建立公用管道或私有管道。如果将参数 `private` 设置为 TRUE，则建立私有管道；如果设置为 FALSE，则建立公用管道。语法如下：

```
DBMS_PIPE.CREATE_PIPE (
    pipename IN VARCHAR2,
    maxpipesize IN INTEGER DEFAULT 8192,
    private IN BOOLEAN DEFAULT TRUE)
RETURN INTEGER;
```

如上所示，`pipename` 用于指定管道的名称，`maxpipesize` 用于指定管道消息的最大尺寸，`private` 用于指定管道类型。如果该函数返回 0，则表示建立管道成功；否则表示建立管道失败。下面以建立公用管道 `public_pipe` 为例，说明使用该函数的方法。示例如下：

```
DECLARE
    flag INT;
BEGIN
    flag:=dbms_pipe.create_pipe('public_pipe',8192,FALSE);
    IF flag=0 THEN
        dbms_output.put_line('建立公用管道成功');
    END IF;
END;
/
建立公用管道成功
```

### 2. PACK\_MESSAGE

该过程用于将消息写入到本地消息缓冲区。为了给管道发送消息，首先需要使用过程 `PACK_MESSAGE` 将消息写入到本地消息缓冲区，然后使用过程 `SEND_MESSAGE` 将本地消息缓冲区中的消息发送到管道。语法如下：

```
DBMS_PIPE.PACK_MESSAGE (item IN VARCHAR2);
DBMS_PIPE.PACK_MESSAGE (item IN NCHAR);
DBMS_PIPE.PACK_MESSAGE (item IN NUMBER);
DBMS_PIPE.PACK_MESSAGE (item IN DATE);
DBMS_PIPE.PACK_MESSAGE_RAW (item IN RAW);
DBMS_PIPE.PACK_MESSAGE_ROWID (item IN ROWID);
```

如上所示，`item` 用于指定管道消息，其输入值可以是字符、数字、日期等多种数据类型。下面以将雇员信息写入本地消息缓冲区为例，说明使用该过程的方法。示例如下：

```
DECLARE
    v_ename emp.ename%TYPE;
    v_sal emp.sal%TYPE;
    v_rowid ROWID;
BEGIN
    SELECT ename,sal,rowid INTO v_ename,v_sal,v_rowid
    FROM emp WHERE empno=&no;
    dbms_pipe.pack_message('雇员名:'||v_ename);
    dbms_pipe.pack_message('工资:'||v_sal);
    dbms_pipe.pack_message('ROWID:'||v_rowid);
END;
```

```
/  
输入 no 的值: 7788
```

### 3. SEND\_MESSAGE

该函数用于将本地消息缓冲区中的内容发送到管道。使用该函数的语法如下：

```
DBMS_PIPE.SEND_MESSAGE (
    pipename IN VARCHAR2,
    timeout IN INTEGER DEFAULT MAXWAIT,
    maxpipesize IN INTEGER DEFAULT 8192)
RETURN INTEGER;
```

如上所示，timeout 用于指定发送消息的超时时间，如果函数返回 0，则表示消息发送成功；如果函数返回 1，则表示发送消息超时；如果函数返回 3，则表示出现中断。下面以将本地缓冲区消息发送到 PUBLIC\_PIPE 为例，说明使用函数 SEND\_MESSAGE 的方法。示例如下：

```
DECLARE
    flag INT;
BEGIN
    flag:=dbms_pipe.send_message('PUBLIC_PIPE');
    IF flag=0 THEN
        dbms_output.put_line('消息成功发送到管道');
    END IF;
END;
/  
消息成功发送到管道
```

### 4. RECEIVE\_MESSAGE

该函数用于接收管道消息，并将接收到的消息写入到本地消息缓冲区。当接收到管道消息之后，会删除管道消息。注意，管道消息只能被接收一次。使用该函数的语法如下：

```
DBMS_PIPE.RECEIVE_MESSAGE (
    pipename IN VARCHAR2,
    timeout IN INTEGER DEFAULT maxwait)
RETURN INTEGER;
```

如果函数返回 0，则表示接收消息成功；如果函数返回 1，则表示出现超时；如果函数返回 2，则表示本地缓冲区不能容纳管道消息；如果函数返回 3，则表示发生中断。下面以接收管道 public\_pipe 的消息为例，说明接收管道消息的方法。示例如下：

```
DECLARE
    flag INT;
BEGIN
    flag:=dbms_pipe.receive_message('PUBLIC_PIPE');
    IF flag=0 THEN
        dbms_output.put_line('接收公用管道消息成功');
    END IF;
END;
/  
接收公用管道消息成功
```

### 5. NEXT\_ITEM\_TYPE

该函数用于确定本地消息缓冲区下一项的数据类型，在调用了 RECEIVE\_MESSAGE 之后

调用该函数。语法如下：

```
DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER;
```

如果该函数返回 0，则表示管道没有任何消息；如果返回 6，则表示下一项的数据类型为 NUMBER；如果返回 9，则表示下一项的数据类型为 VARCHAR2；如果返回 11，则表示下一项的数据类型为 ROWID；如果返回 12，则表示下一项的数据类型为 DATE；如果返回 23，则表示下一项的数据类型为 RAW。使用该函数的示例如下：

```
DECLARE
    item_no INT;
BEGIN
    item_no:=dbms_pipe.next_item_type;
    dbms_output.put_line('项编号：'||item_no);
END;
/
项编号：9
```

## 6. UNPACK\_MESSAGE

该过程用于将消息缓冲区中的内容写入到变量中。在使用函数 RECEIVE\_MESSAGE 接收到管道消息之后，应该使用过程 UNPACK\_MESSAGE 取得消息缓冲区的消息。语法如下：

```
DBMS_PIPE.UNPACK_MESSAGE (item OUT VARCHAR2);
DBMS_PIPE.UNPACK_MESSAGE (item OUT NCHAR);
DBMS_PIPE.UNPACK_MESSAGE (item OUT NUMBER);
DBMS_PIPE.UNPACK_MESSAGE (item OUT DATE);
DBMS_PIPE.UNPACK_MESSAGE_RAW (item OUT RAW);
DBMS_PIPE.UNPACK_MESSAGE_ROWID (item OUT ROWID);
```

当使用过程 UNPACK\_MESSAGE 取出消息缓冲区的消息时，每次只能取出一条消息。如果要取出多条消息，则需要多次调用过程 UNPACK\_MESSAGE。示例如下：

```
DECLARE
    message VARCHAR2(100);
BEGIN
    dbms_pipe.unpack_message(message);
    dbms_output.put_line(message);
END;
/
雇员名:SCOTT
```

## 7. REMOVE\_PIPE

该函数用于删除已经建立的管道。语法如下：

```
DBMS_PIPE.REMOVE_PIPE (pipename IN VARCHAR2) RETURN INTEGER;
```

如果该函数返回 0，则表示删除管道成功，否则会显示错误信息。下面以删除公用管道 PUBLIC\_PIPE 为例，说明使用该函数的方法。示例如下：

```
DECLARE
    flag INT;
BEGIN
    flag:=dbms_pipe.remove_pipe('PUBLIC_PIPE');
    IF flag=0 THEN
```

```

    dbms_output.put_line('删除公用管道成功');
END IF;
END;
/
删除公用管道成功

```

### 8. PURGE

该过程用于清除管道中的内容。语法如下：

```
DBMS_PIPE.PURGE (pipename IN VARCHAR2);
```

### 9. RESET\_BUFFER

该过程用于复位管道缓冲区：因为所有管道都共享单个管道缓冲区，所以在使用新管道之前应该复位管道缓冲区。语法如下：

```
DBMS_PIPE.RESET_BUFFER;
```

### 10. UNIQUE\_SESSION\_NAME

该函数用于为特定会话返回唯一的名称，并且名称的最大长度为 30 字节。对于同一会话来说，其值不会改变。示例如下：

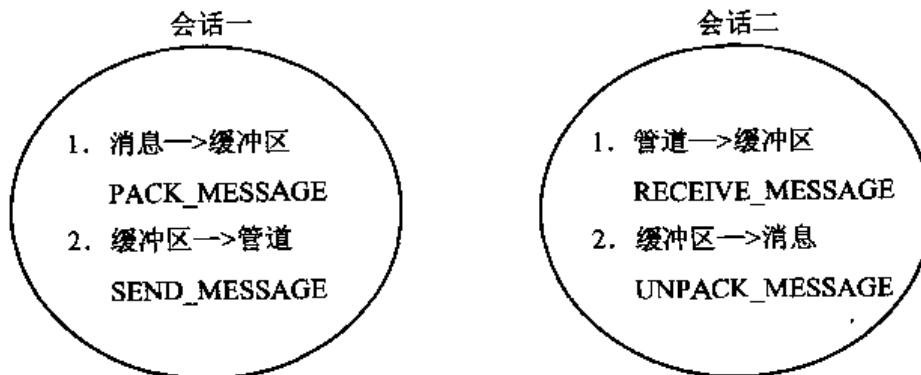
```

DECLARE
  v_session VARCHAR2(200);
BEGIN
  v_session:=dbms_pipe.unique_session_name;
  dbms_output.put_line(v_session);
END;
/
ORA$PIPE$000800130001

```

### 11. 管道使用示例

当使用管道（PIPE）时，一个会话需要将消息发送到管道中，而另一个会话则需要接收管道消息。当发送消息到管道时，需要首先将消息写入本地消息缓冲区，然后将本地消息缓冲区内容发送到管道；当接收管道消息时，需要首先使用本地消息缓冲区接收管道消息，然后从消息缓冲区中取得具体消息。如下图所示：



下面以建立过程 `send_message` 和 `receive_message`，并使用这两个过程发送和接收消息为例，说明使用管道的方法。

#### 示例一：建立过程 `send_message`

过程 `send_message` 将用于建立管道并发送消息。示例如下：

```

CREATE OR REPLACE PROCEDURE send_message(
    pipename VARCHAR2,message VARCHAR2)
IS
    flag INT;
BEGIN
    flag:=dbms_pipe.create_pipe(pipename);
    IF flag=0 THEN
        dbms_pipe.pack_message(message);
        flag:=dbms_pipe.send_message(pipename);
    END IF;
END;
/

```

### 示例二：建立过程 receive\_message

过程 receive\_message 将用于接收并输出管道消息。示例如下：

```

CREATE OR REPLACE PROCEDURE receive_message(
    pipename VARCHAR2,message OUT VARCHAR2)
IS
    flag INT;
BEGIN
    flag:=dbms_pipe.receive_message(pipename);
    IF flag=0 THEN
        dbms_pipe.unpack_message(message);
        flag:=dbms_pipe.remove_pipe(pipename);
    END IF;
END;
/

```

### 示例三：使用过程 send\_message

会话一：SQL> exec send\_message('pipe1','你好吗')

会话二：

```

SQL> VAR message VARCHAR2(100)
SQL> exec scott.receive_message('pipe1',:message)
SQL> PRINT message
MESSAGE
-----
```

你好吗

## 17.4 DBMS\_ALERT

包 DBMS\_ALERT 用于生成并传递数据库预警信息。合理地使用包和数据库触发器，可以使得在发生特定数据库事件时将信息传递给应用程序。但是，如果某个数据库用户要使用包 DBMS\_ALERT，则必须要以 SYS 登录，为该用户授予执行权限。示例如下：

```

SQL> conn sys/oracle as sysdba
SQL> GRANT EXECUTE ON dbms_alert TO scott;
```

## 1. REGISTER

该过程用于注册预警事件。语法如下：

```
DBMS_ALERT.REGISTER (name IN VARCHAR2);
```

如上所示，`name` 用于指定预警事件名称，其值不能超过 30 字节。下面以注册预警事件 `ALERT1` 为例，说明注册预警事件的方法。示例如下：

```
SQL> exec dbms_alert.register('alert1')
```

## 2. REMOVE

该过程用于删除会话不需要的预警事件。语法如下：

```
DBMS_ALERT.REMOVE (name IN VARCHAR2);
```

下面以删除预警事件 `ALERT1` 为例，说明删除预警事件的方法。示例如下：

```
SQL> exec dbms_alert.remove('alert1')
```

## 3. REMOVEALL

该过程用于删除当前会话所有已经注册的预警事件。语法如下：

```
DBMS_ALERT.REMOVEALL
```

## 4. SET\_DEFAULTS

该过程用于设置检测预警事件的时间间隔，默认时间间隔为 5 秒。语法如下：

```
DBMS_ALERT.SET_DEFAULTS (sensitivity IN NUMBER);
```

如上所示，`sensitivity` 用于指定检测预警事件的时间间隔。下面以设置时间间隔为 20 秒为例，说明使用该过程的方法。示例如下：

```
SQL> exec dbms_alert.set_defaults(20)
```

## 5. SIGNAL

该过程用于指定预警事件所对应的预警消息。只有在提交事务时才会发出预警信号，而当回退事务时不会发出预警信号。语法如下：

```
DBMS_ALERT.SIGNAL (name IN VARCHAR2,message IN VARCHAR2);
```

如上所示，`message` 用于指定预警事件的消息，并且消息长度不能超过 1800 字节。示例如下：

```
SQL> exec dbms_alert.signal('alert1','hello')
```

## 6. WAITANY

该过程用于等待当前会话的任何预警事件，并且在预警事件发生时输出相应信息。在执行该过程之前，会隐含地发出 COMMIT。使用该过程的语法如下：

```
DBMS_ALERT.WAITANY (
    name OUT VARCHAR2,message OUT VARCHAR2,
    status OUT INTEGER,timeout IN NUMBER DEFAULT MAXWAIT);
```

如上所示，`status` 用于返回状态值，返回 0 表示发生了预警事件，返回 1 表示超时；`timeout` 用于设置等待预警事件的超时时间。

## 7. WAITONE

该过程用于等待当前会话的特定预警事件，并且在发生预警事件时输出预警消息。在执行该过程之前，会隐含地发出 COMMIT。使用该过程的语法如下：

```
DBMS_ALERT.WAITONE (
    name IN VARCHAR2,message OUT VARCHAR2,
    status OUT INTEGER,timeout IN NUMBER DEFAULT MAXWAIT);
```

## 8. 预警事件使用示例

当在应用程序中使用预警事件时，需要结合使用 DBMS\_ALERT 包和触发器。下面以修改雇员工资发出预警事件为例，说明如何在应用程序中使用预警事件。

### 示例一：建立触发器 tr\_update\_sal

触发器 tr\_update\_sal 用于给预警事件 sal\_upd\_alert 发出信号。建立该触发器的示例如下：

```
CREATE OR REPLACE TRIGGER tr_upd_sal
AFTER UPDATE OF sal ON emp
BEGIN
    dbms_alert.signal('sal_upd_alert','修改了雇员工资');
END;
/
```

在建立了触发器 tr\_upd\_sal 之后，如果用户修改了任何雇员的工资，都会触发预警事件 sal\_upd\_alert，并生成相应预警消息。

### 示例二：建立过程 wait\_event

过程 wait\_event 用于注册并等待预警事件 sal\_upd\_alert。为了在应用程序中使用预警事件，必须首先注册预警事件，然后等待预警事件的发生，在预警事件不需要时可删除该预警事件。建立过程 wait\_event 的示例如下：

```
CREATE OR REPLACE PROCEDURE wait_event(name VARCHAR2)
IS
    message VARCHAR2(200);
    status INT;
BEGIN
    dbms_alert.register(name);
    dbms_alert.waitone(name,message,status);
    IF status=0 THEN
        dbms_output.put_line('预警消息:'||message);
    END IF;
    dbms_alert.remove(name);
END;
/
```

### 示例三：使用预警事件

```
set serveroutput on
BEGIN
    FOR i IN 1..5 LOOP
        wait_event('sal_upd_alert');
    END LOOP;
END;
/
预警消息：修改了雇员工资
预警消息：修改了雇员工资
预警消息：修改了雇员工资
预警消息：修改了雇员工资
预警消息：修改了雇员工资
```

如上所示，当执行了以上 PL/SQL 块之后，会话就会处于等待状态。当其他会话修改了雇

员工资之后，就会显示预警消息。在显示了 5 次预警消息之后，会退出循环。

## 17.5 DBMS\_TRANSACTION

该包用于在过程、函数和包中执行 SQL 事务处理语句，下面给大家详细介绍该包所包含的过程和函数。

### 1. READ\_ONLY

该过程用于开始只读事务，其作用与 SQL 语句 SET TRANSACTION READ ONLY 完全相同。注意，该过程必须是事务开始的第一条语句。语法如下：

```
DBMS_TRANSACTION.READ_ONLY;
```

### 2. READ\_WRITE

该过程用于开始读写事务，其作用与 SQL 语句 SET TRANSACTION READ WRITE 完全相同。注意，该过程必须是事务开始的第一条语句。语法如下：

```
DBMS_TRANSACTION.READ_WRITE;
```

### 3. ADVISE\_ROLLBACK

该过程用于建议回退远程数据库的分布式事务，其作用与 SQL 语句 ALTER SESSION ADVISE ROLLBACK 完全相同。语法如下：

```
DBMS_TRANSACTION.ADVISE_ROLLBACK;
```

### 4. ADVISE\_NOTHING

该过程用于建议远程数据库的分布式事务不进行任何处理，其作用与 SQL 语句 ALTER SESSION ADVISE NOTHING 完全相同。语法如下：

```
DBMS_TRANSACTION.ADVISE_NOTHING;
```

### 5. ADVISE\_COMMIT

该过程用于建议提交远程数据库的分布式事务，其作用与 SQL 语句 ALTER SESSION ADVISE COMMIT 完全相同。语法如下：

```
DBMS_TRANSACTION.ADVISE_COMMIT;
```

### 6. USE\_ROLLBACK\_SEGMENT

该过程用于指定事务所要使用的回滚段，其作用与 SQL 语句 SET TRANSACTION USE ROLLBACK SEGMENT 回滚段名作用完全相同。语法如下：

```
DBMS_TRANSACTION.USE_ROLLBACK_SEGMENT(rb_name VARCHAR2);
```

如上所示，rb\_name 用于指定事务所要使用的回滚段名称。

### 7. COMMIT\_COMMENT

该过程用于在提交事务时指定注释，其作用与 SQL 语句 COMMIT COMMENT <text> 完全相同。语法如下：

```
DBMS_TRANSACTION.COMMIT_COMMENT(cmnt VARCHAR2);
```

如上所示，cmnt 用于指定与事务相关的注释信息。

### 8. COMMIT\_FORCE

该过程用于强制提交分布式事务，其作用与 SQL 语句 COMMIT FORCE text,number 完全相同。语法如下：

```
DBMS_TRANSACTION.COMMIT_FORCE {
```

```
xid VARCHAR2, scn VARCHAR2 DEFAULT NULL);
```

如上所示，`xid` 用于指定事务 ID 号，`scn` 用于指定系统改变号。

#### 9. COMMIT

该过程用于提交当前事务，其作用与 SQL 语句 COMMIT 完全相同。语法如下：

```
DBMS_TRANSACTION.COMMIT;
```

#### 10. SAVEPOINT

该过程用于设置保存点，其作用与 SQL 语句 SAVEPOINT `savepoint` 完全相同。语法如下：

```
DBMS_TRANSACTION.SAVEPOINT(savept VARCHAR2);
```

如上所示，`savept` 用于指定保存点名称。

#### 11. ROLLBACK

该过程用于回退当前事务，其作用与 SQL 语句 ROLLBACK 完全相同。语法如下：

```
DBMS_TRANSACTION.ROLLBACK;
```

#### 12. ROLLBACK\_SAVEPOINT

该过程用于回退到保存点，并取消部分事务，其作用与 SQL 语句 ROLLBACK TO SAVEPOINT <savepoint\_name> 完全相同。语法如下：

```
DBMS_TRANSACTION.ROLLBACK_SAVEPOINT(savept VARCHAR2);
```

如上所示，`savept` 用于指定要回退到的保存点名称。

#### 13. ROLLBACK\_FORCE

该过程用于强制回退分布式事务，其作用与 SQL 语句 ROLLBACK FORCE <text> 完全相同。语法如下：

```
DBMS_TRANSACTION.ROLLBACK_FORCE(xid VARCHAR2);
```

如上所示，`xid` 用于指定事务 ID 号。

#### 14. BEGIN\_DISCRETE\_TRANSACTION

该过程用于开始独立事务模式，语法如下：

```
DBMS_TRANSACTION.BEGIN_DISCRETE_TRANSACTION;
```

#### 15. PURGE\_MIXED

该过程用于清除分布式事务的混合事务结果，语法如下：

```
DBMS_TRANSACTION.PURGE_MIXED(xid VARCHAR2);
```

如上所示，`xid` 用于指定事务 ID 号。

#### 16. PURGE\_LOST\_DB\_ENTRY

该过程用于清除本地数据库所记载的远程事务入口，该事务入口操作因为远程数据库问题未能在远程数据库完成。语法如下：

```
DBMS_TRANSACTION.PURGE_LOST_DB_ENTRY (xid VARCHAR2);
```

如上所示，`xid` 用于指定事务入口号。

#### 17. LOCAL\_TRANSACTION\_ID

该函数用于返回当前事务的事务标识号，其语法如下：

```
DBMS_TRANSACTION.LOCAL_TRANSACTION_ID (
    create_transaction BOOLEAN := FALSE)
RETURN VARCHAR2;
```

如上所示，`create_transaction` 用于指定是否要启动新事务，如果设置为 TRUE，则会启动新事务。

### 18. STEP\_ID

该函数用于返回排序 DML 事务的惟一正整数。语法如下：

```
DBMS_TRANSACTION.STEP_ID RETURN NUMBER;
```

## 17.6 DBMS\_SESSION

该包提供了使用 PL/SQL 实现 ALTER SESSION 命令, SET ROLE 命令和其他会话信息的方法。下面详细介绍该包所包含的过程和函数。

### 1. SET\_IDENTIFIER

该过程用于设置会话的客户 ID 号，语法如下：

```
DBMS_SESSION.SET_IDENTIFIER (client_id VARCHAR2);
```

如上所示, client\_id 用于指定当前会话的应用标识符。

### 2. SET\_CONTEXT

该过程用于设置应用上下文属性。语法如下：

```
DBMS_SESSION.SET_CONTEXT (
    namespace VARCHAR2, attribute VARCHAR2, value VARCHAR2);
DBMS_SESSION.SET_CONTEXT (
    namespace VARCHAR2, attribute VARCHAR2, value VARCHAR2,
    username VARCHAR2;client_id VARCHAR2 );
```

如上所示, namespace 用于指定应用上下文的命名空间; attribute 用于指定应用上下文的属性; value 用于指定属性值; username 用于指定应用上下文的用户名属性。

### 3. CLEAR\_CONTEXT

该过程用于清除应用上下文的属性设置，语法如下：

```
DBMS_SESSION.CLEAR_CONTEXT(
    namespace VARCHAR2,client_identifier VARCHAR2,
    attribute VARCHAR2);
```

如上所示, client\_identifier 只适用于全局上下文。

### 4. CLEAR\_IDENTIFIER

该过程用于删除会话的 set\_client\_id。语法如下：

```
DBMS_SESSION.CLEAR_IDENTIFIER();
```

### 5. SET\_ROLE

该过程用于激活或禁止会话角色，与 SQL 语句 SET ROLE 作用完全相同。语法如下：

```
DBMS_SESSION.SET_ROLE (role_cmd VARCHAR2);
```

如上所示, role\_cmd 会追加到 set role 命令之后，在 SQL\*Plus 中使用该过程激活或禁止角色的示例如下：

```
SQL> exec dbms_session.set_role('DBA')
SQL> exec dbms_session.set_role('none')
```

### 6. SET\_SQL\_TRACE

该过程用于激活或禁止当前会话的 SQL 跟踪，其作用与 SQL 语句 ALTER SESSION SET SQL\_TRACE=... 完全相同。语法如下：

```
DBMS_SESSION.SET_SQL_TRACE (sql_trace boolean);
```

如上所示，`sql_trace` 用于指定布尔值。当设置为 TRUE 时，表示激活 SQL 跟踪；当设置为 FALSE 时，表示禁止 SQL 跟踪。在 SQL\*Plus 中使用该过程激活和禁止 SQL 跟踪的示例如下：

```
SQL> exec dbms_session.set_sql_trace(TRUE)
SQL> exec dbms_session.set_sql_trace(FALSE)
```

### 7. SET\_NLS

该过程用于设置 NLS 特征，其作用与 SQL 语句 `ALTER SESSION SET <nls_param>=<value>` 完全相同。语法如下：

```
DBMS_SESSION.SET_NLS (param VARCHAR2, value VARCHAR2);
```

如上所示，`param` 用于指定 NLS 参数；`value` 用于指定 NLS 参数的值。在 SQL\*Plus 中使用该过程设置 NLS 参数的示例如下：

```
SQL> exec dbms_session.set_nls('nls_date_format',''YYYY-MM-DD')
SQL> SELECT sysdate FROM dual;
SYSDATE
-----
2004-01-31
```

### 8. CLOSE\_DATABASE\_LINK

该过程用于关闭已经打开的数据库链，其作用与 SQL 语句 `ALTER SESSION CLOSE DATABASE LINK <name>` 完全相同。语法如下：

```
DBMS_SESSION.CLOSE_DATABASE_LINK (dblink VARCHAR2);
```

如上所示，`dblink` 用于指定要关闭的数据库链名。

### 9. RESET\_PACKAGE

该过程用于复位当前会话的所有包，并且会释放包状态。语法如下：

```
DBMS_SESSION.RESET_PACKAGE;
```

### 10. MODIFY\_PACKAGE\_STATE

该过程用于修改当前会话的 PL/SQL 程序单元的状态，语法如下：

```
DBMS_SESSION.MODIFY_PACKAGE_STATE(
    action_flags IN PLS_INTEGER);
```

如上所示，`action_flags` 用于指定 PL/SQL 程序单元位标记，当设置为 1 时，会释放 PL/SQL 程序单元所占用的内存；当设置为 2 时，会重新初始化 PL/SQL 包。

### 11. UNIQUE\_SESSION\_ID

该函数用于返回当前会话的惟一 ID 标识符。在 SQL\*Plus 中使用该函数的示例如下：

```
SQL> SELECT dbms_session.unique_session_id FROM dual;
UNIQUE_SESSION_ID
-----
000B02990001
```

### 12. IS\_ROLE\_ENABLED

该函数用于确定当前会话是否激活了特定角色，语法如下：

```
DBMS_SESSION.IS_ROLE_ENABLED (rolename VARCHAR2)
    RETURN BOOLEAN;
```

如上所示，`rolename` 用于指定角色名。如果返回 TRUE，则表示角色被激活；如果返回 FALSE，则表示角色未被激活。示例如下：

```
set serveroutput on
```

```

BEGIN
  IF dbms_session.is_role_enabled('CONNECT') THEN
    dbms_output.put_line('CONNECT 角色被激活');
  END IF;
END;
/
CONNECT 角色被激活

```

### 13. IS\_SESSION\_ALIVE

该函数用于确定特定会话是否处于活动状态，语法如下：

```
DBMS_SESSION.IS_SESSION_ALIVE (uniqueid VARCHAR2)
  RETURN BOOLEAN;
```

如上所示，uniqueid 用于指定会话 ID 号。如果会话处于活动状态，则返回 TRUE，否则返回 FALSE。

### 14. SET\_CLOSE\_CACHED\_OPEN\_CURSORS

该过程用于打开或关闭 close\_cached\_open\_cursors，其作用与 ALTER SESSION SET CLOSE\_CACHED\_OPEN\_CURSORS .. 完全相同。语法如下：

```
DBMS_SESSION.SET_CLOSE_CACHED_OPEN_CURSORS (
  close_cursors BOOLEAN);
```

如上所示，close\_cursors 用于指定布尔值。当设置为 TRUE 时，会打开 close\_cached\_open\_cursors；当设置为 FALSE 时，会关闭 close\_cached\_open\_cursors。

### 15. FREE\_UNUSED\_USER\_MEMORY

该过程用于在执行了大内存操作（超过 100K）之后回收未用内存，语法如下：

```
DBMS_SESSION.FREE_UNUSED_USER_MEMORY;
```

### 16. SET\_CONTEXT

该过程用于设置应用上下文属性的值，语法如下：

```
DBMS_SESSION.SET_CONTEXT (
  namespace VARCHAR2,attribute VARCHAR2,value VARCHAR2,
  username VARCHAR2,client_id VARCHAR2);
```

### 17. LIST\_CONTEXT

该过程用于返回当前会话的命名空间和上下文列表，语法如下：

```

TYPE AppCtxRecTyp IS RECORD (
  namespace VARCHAR2(30),attribute VARCHAR2(30),
  value VARCHAR2(256));
TYPE AppCtxTabTyp IS TABLE OF AppCtxRecTyp INDEX BY BINARY_INTEGER;
DBMS_SESSION.LIST_CONTEXT (list OUT AppCtxTabTyp,size OUT NUMBER);
```

如上所示，list 用于取得当前会话的列表集；size 用于返回列表个数。

### 18. SWITCH\_CURRENT\_CONSUMER\_GROUP

该过程用于改变当前会话的资源使用组，语法如下：

```
DBMS_SESSION.switch_current_consumer_group (
  new_consumer_group IN VARCHAR2,
  old_consumer_group OUT VARCHAR2,
  initial_group_on_error IN BOOLEAN);
```

如上所示，new\_consumer\_group 用于指定新资源使用组；old\_consumer\_group 用于取得原

有资源使用组; `initial_group_on_error` 用于指定布尔值; 当设置为 TRUE 时, 出错后会使用原有资源使用组。

## 17.7 DBMS\_ROWID

包 `DBMS_ROWID` 用于在 PL/SQL 程序和 SQL 语句中取得行标识符 (ROWID) 的信息并建立 ROWID。通过使用包 `DBMS_ROWID`, 可以取得行所在的文件号、行所在文件的数据块号、行所在数据块的行号, 以及数据库对象号等信息。

### 1. ROWID\_CREATE

该函数用于建立 ROWID, 语法如下:

```
DBMS_ROWID.ROWID_CREATE (
    rowid_type IN NUMBER, object_number IN NUMBER,
    relative_fno IN NUMBER, block_number IN NUMBER,
    row_number IN NUMBER)
RETURN ROWID;
```

如上所示, `rowid_type` 用于指定 ROWID 类型 (0: 受限 ROWID, 1: 扩展 ROWID); `object_number` 用于指定数据对象号; `relative_fno` 用于指定相对文件号; `block_number` 用于指定在文件中的数据块号; `row_number` 用于指定在数据块中的行号。使用该函数的示例如下:

```
set serveroutput on
DECLARE
    my_rowid ROWID;
BEGIN
    my_rowid:=dbms_rowid.rowid_create(1,12197,3,100,1);
    dbms_output.put_line(my_rowid);
END;
/
AAAC+1AADAAAABKAAB
```

### 2. ROWID\_INFO

该过程用于取得特定 ROWID 的详细信息, 语法如下:

```
DBMS_ROWID.ROWID_INFO (
    rowid_in IN ROWID, rowid_type OUT NUMBER,
    object_number OUT NUMBER, relative_fno OUT NUMBER,
    block_number OUT NUMBER, row_number OUT NUMBER);
```

如上所示, `rowid_in` 用于指定 ROWID 示例如下:

```
DECLARE
    rowid_type NUMBER;
    object_number NUMBER;
    relative_fno NUMBER;
    block_number NUMBER;
    row_number NUMBER;
BEGIN
    dbms_rowid.rowid_info('AAAC90AAFAAAAAcAAK',rowid_type,
        object_number,relative_fno,block_number,row_number);
```

```

    dbms_output.put_line('数据对象号:'||object_number);
END;
/
数据对象号:12148

```

### 3. ROWID\_TYPE

该函数用于返回特定 ROWID 的类型，语法如下：

```
DBMS_ROWID.ROWID_TYPE (row_id IN ROWID) RETURN NUMBER;
```

当返回值为 0 时，表示受限 ROWID；当返回值为 1 时，表示扩展 ROWID。示例如下：

```

SQL> SELECT dbms_rowid.rowid_type('AAAC90AAFAAAAAcaAK')
  2  FROM dual;
DBMS_ROWID.ROWID_TYPE('AAAC90AAFAAAAAcaAK')

-----
1

```

### 4. ROWID\_OBJECT

该函数用于取得特定 ROWID 所对应的数据对象号语法如下：

```
DBMS_ROWID.ROWID_OBJECT (row_id IN ROWID) RETURN NUMBER;
```

示例如下：

```

SQL> SELECT dbms_rowid.rowid_object(ROWID)
  2  FROM dept WHERE deptno=10;
DBMS_ROWID.ROWID_OBJECT(ROWID)

-----
12146

```

### 5. ROWID\_RELATIVE\_FNO

该函数用于取得特定 ROWID 所对应的相对文件号语法如下：

```
DBMS_ROWID.ROWID_RELATIVE_FNO (row_id IN ROWID)
RETURN NUMBER;
```

示例如下：

```

SQL> SELECT dbms_rowid.rowid_relative_fno(ROWID)
  2  FROM dept WHERE deptno=10;
DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID)

-----
5

```

### 6. ROWID\_BLOCK\_NUMBER

该函数用于返回特定 ROWID 在数据文件中所对应的数据块号，语法如下：

```
DBMS_ROWID.ROWID_BLOCK_NUMBER (row_id IN ROWID)
RETURN NUMBER;
```

如上所示，row\_id 用于指定要解析的 ROWID。示例如下：

```

SQL> SELECT dbms_rowid.rowid_block_number(ROWID)
  2  FROM dept WHERE deptno=10;
DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)

-----
12

```

### 7. ROWID\_ROW\_NUMBER

该函数用于返回特定 ROWID 在数据块中所对应的行号，语法如下：

```
DBMS_ROWID.ROWID_ROW_NUMBER (row_id IN ROWID)
RETURN NUMBER;
```

示例如下：

```
SQL> SELECT dbms_rowid.rowid_row_number(ROWID)
  2  FROM dept WHERE deptno=10;
DBMS_ROWID.ROWID_ROW_NUMBER(ROWID)
-----
0
```

## 8. ROWID\_TO\_ABSOLUTE\_FNO

该函数用于返回特定 ROWID 所对应的绝对文件号，语法如下：

```
DBMS_ROWID.ROWID_TO_ABSOLUTE_FNO (
    row_id IN ROWID, schema_name IN VARCHAR2,
    object_name IN VARCHAR2)
RETURN NUMBER;
```

如上所示，schema\_name 用于指定包含表的方案名；object\_name 用于指定表名。示例如下：

```
DECLARE
    abs_fno INT;
    rowid_val CHAR(18);
BEGIN
    SELECT ROWID INTO rowid_val FROM emp WHERE empno = 7788;
    abs_fno := dbms_rowid.rowid_to_absolute_fno(
        rowid_val, 'SCOTT', 'EMP');
    dbms_output.put_line('7788 所对应的绝对文件号:' || abs_fno);
END;
/
7788 所对应的绝对文件号:5
```

## 9. ROWID\_TO\_EXTENDED

该函数用于将受限 ROWID 转变为扩展 ROWID，语法如下：

```
DBMS_ROWID.ROWID_TO_EXTENDED (
    old_rowid IN ROWID, schema_name IN VARCHAR2,
    object_name IN VARCHAR2, conversion_type IN INTEGER)
RETURN ROWID;
```

如上所示，conversion\_type 用于指定转换类型（rowid\_convert\_internal/external\_convert\_external）。示例如下：

```
SQL> SELECT empno,ename FROM emp
  2  WHERE rowid=dbms_rowid.rowid_to_extended
  3  ('0000001C.0007.0005', 'SCOTT', 'EMP', 0);
  EMPNO ENAME
-----
7788 SCOTT
```

## 10. ROWID\_TO\_RESTRICTED

该函数用于将扩展 ROWID 转换为受限 ROWID，语法如下：

```
DBMS_ROWID.ROWID_TO_RESTRICTED (
    old_rowid IN ROWID, conversion_type IN INTEGER)
```

```
RETURN ROWID;
```

示例如下：

```
SQL> SELECT dbms_rowid.rowid_to_restricted(rowid,0)
  2  FROM emp WHERE empno=7788;
DBMS_ROWID.ROWID_T
-----
00000001C.0007.0005
```

### 11. ROWID\_VERIFY

该函数用于检查是否可以将受限 ROWID 转变为扩展 ROWID，语法如下：

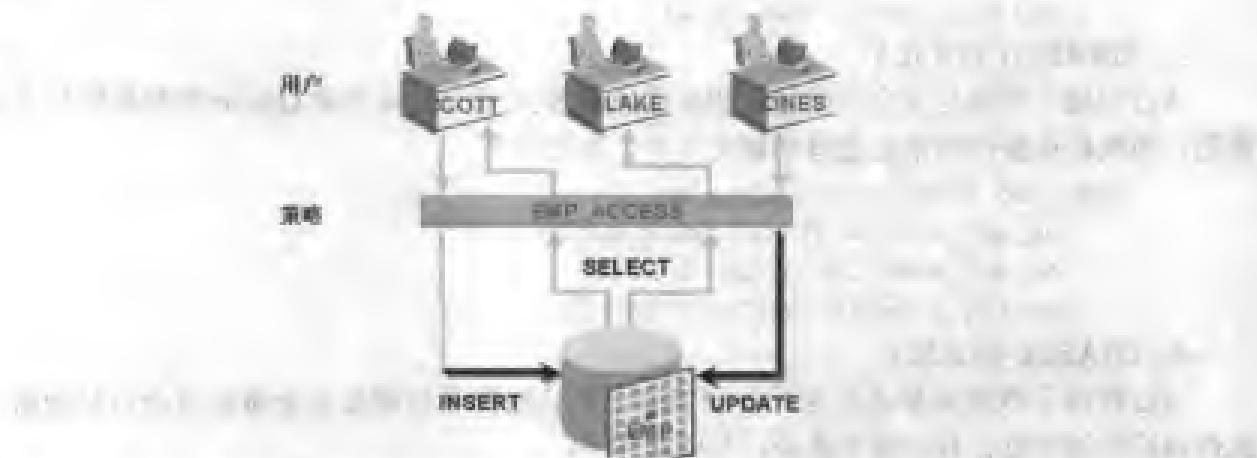
```
DBMS_ROWID.ROWID_VERIFY (
    rowid_in IN ROWID,schema_name IN VARCHAR2,
    object_name IN VARCHAR2,conversion_type IN INTEGER
    RETURN NUMBER;
```

如果函数返回 0，则表示可以转换；如果函数返回 1，则表示不能转换。示例如下：

```
SQL> SELECT dbms_rowid.rowid_verify
  2  ('00000001C.0008.0005','SCOTT','EMP',0) FROM dual;
DBMS_ROWID.ROWID_VERIFY('00000001C.0008.0005','SCOTT','EMP',0)
-----
```

## 17.8 DBMS\_RLS

包 DBMS\_RLS 只适用于 Oracle Enterprise Edition，它用于实现精细访问控制，并且精细访问控制是通过在 SQL 语句中动态增加谓词（WHERE 子句）来实现的。通过使用 Oracle 的精细访问控制特征，可以使不同数据库用户在执行相同 SQL 语句时操作同一张表上的不同数据。如下图所示：



如图中所示，通过使用策略 EMP\_ACCESS，用户 SCOTT, BLAKE, JONES 在执行相同 SQL 语句时，可以返回不同结果。例如，当他们分别执行“SELECT ename FROM emp”时，可以使用户 SCOTT 查询所有雇员，使用户 BLAKE 只能检索部门 30 的雇员，而用户 JONES 只能检索部门 20 的雇员。下面给大家介绍包 DBMS\_RLS 所包含的子程序。

### 1. ADD\_POLICY

该过程用于为表、视图或同义词增加一个安全策略，当执行该操作结束时会自动提交事务。语法如下：

```
DBMS_RLS.ADD_POLICY (
    object_schema IN VARCHAR2 NULL,
    object_name IN VARCHAR2,
    policy_name IN VARCHAR2,
    function_schema IN VARCHAR2 NULL,
    policy_function IN VARCHAR2,
    statement_types IN VARCHAR2 NULL,
    update_check IN BOOLEAN FALSE,
    enable IN BOOLEAN TRUE,
    static_policy IN BOOLEAN FALSE);
```

如上所示，`object_schema` 用于指定包含表、视图或同义词的方案（默认值 `NULL` 表示当前方案）；`object_name` 用于指定要增加安全策略的表、视图或同义词；`policy_name` 用于指定要增加的安全策略名称；`function_schema` 用于指定策略函数的所在方案（默认值 `NULL` 表示当前方案）；`policy_function` 用于指定生成安全策略谓词的函数名；`statement_types` 用于指定使用安全策略的 SQL 语句（默认值 `NULL` 表示适用于 `SELECT, INSERT, UPDATE` 以及 `DELETE` 语句）；`update_check` 用于指定在执行 `INSERT` 或 `UPDATE` 时是否检查安全策略；`enable` 用于指定是否要激活安全策略；`static_policy` 用于指定是否要生成静态的安全策略。

### 2. DROP\_POLICY

该过程用于删除定义在特定表、视图或同义词上的安全策略，当执行该操作结束时会自动提交事务。语法如下：

```
DBMS_RLS.DROP_POLICY (
    object_schema IN VARCHAR2 NULL,
    object_name IN VARCHAR2,
    policy_name IN VARCHAR2);
```

### 3. REFRESH\_POLICY

该过程用于刷新与安全策略修改相关的所有 SQL 语句，并使得 Oracle 重新解析相关 SQL 语句，当执行该操作结束时会自动提交事务。语法如下：

```
DBMS_RLS.REFRESH_POLICY (
    object_schema IN VARCHAR2 NULL,
    object_name IN VARCHAR2 NULL,
    policy_name IN VARCHAR2 NULL);
```

### 4. ENABLE\_POLICY

该过程用于激活或禁止特定的安全策略，默认情况下当增加安全策略时会自动激活。当执行该操作结束时会自动提交事务，其语法如下：

```
DBMS_RLS.ENABLE_POLICY (
    object_schema IN VARCHAR2 NULL,
    object_name IN VARCHAR2,
    policy_name IN VARCHAR2,
    enable IN BOOLEAN);
```

## 5. CREATE\_POLICY\_GROUP

该过程用于建立安全策略组，语法如下：

```
DBMS_RLS.CREATE_POLICY_GROUP (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    policy_group VARCHAR2 );
```

如上所示，`policy_group` 用于指定安全策略组的名称。

## 6. ADD\_GROUPED\_POLICY

该过程用于增加与特定策略组相关的安全策略，语法如下：

```
DBMS_RLS.ADD_GROUPED_POLICY(
    object_schema VARCHAR2,
    object_name VARCHAR2,
    policy_group VARCHAR2,
    policy_name VARCHAR2,
    function_schema VARCHAR2,
    policy_function VARCHAR2,
    statement_types VARCHAR2,
    update_check BOOLEAN,
    enabled BOOLEAN,
    static_policy BOOLEAN FALSE );
```

## 7. ADD\_POLICY\_CONTEXT

该过程用于为应用增加上下文，语法如下：

```
DBMS_RLS.ADD_POLICY_CONTEXT (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    namespace VARCHAR2,
    attribute VARCHAR2 );
```

如上所示，`namespace` 用于指定命名空间；`attribute` 用于指定上下文属性。

## 8. DELETE\_POLICY\_GROUP

该过程用于删除安全策略组，语法如下：

```
DBMS_RLS.DELETE_POLICY_GROUP (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    policy_group VARCHAR2 );
```

## 9. DROP\_GROUPED\_POLICY

该过程用于删除特定策略组的安全策略，语法如下：

```
DBMS_RLS.DROP_GROUPED_POLICY (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    policy_group VARCHAR2,
    policy_name VARCHAR2 );
```

## 10. DROP\_POLICY\_CONTEXT

该过程用于删除对象的上下文，语法如下：

```
DBMS_RLS.DROP_POLICY_CONTEXT (
```

```

object_schema VARCHAR2,
object_name VARCHAR2,
namespace VARCHAR2,
attribute VARCHAR2 );

```

### 11. ENABLE\_GROUPED\_POLICY

该过程用于激活或禁止特定策略组的安全策略，语法如下：

```

DBMS_RLS.ENABLE_GROUPED_POLICY (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    group_name VARCHAR2,
    policy_name VARCHAR2,
    enable BOOLEAN );

```

### 12. REFRESH\_GROUPED\_POLICY

该过程用于刷新与特定安全策略组的安全策略相关的 SQL 语句（重新解析 SQL 语句），语法如下：

```

DBMS_RLS.refresh_grouped_policy (
    object_schema VARCHAR2,
    object_name VARCHAR2,
    group_name VARCHAR2,
    policy_name VARCHAR2 );

```

## 13. 使用 DBMS\_RLS 实现精细访问控制

假定应用开发人员希望 SYS, SYSTEM, SCOTT 用户可以访问 EMP 表的所有雇员，BLAKE 用户只能访问部门 30 的雇员，JONES 用户只能访问部门 20 的雇员，而其他用户只能访问部门 10 的雇员。下面以实现该目标为例，说明使用精细访问控制的方法。具体步骤如下：

### (1) 建立应用上下文

为了实现精细访问控制，必须要建立应用上下文。一般情况下建立应用上下文是由 DBA 来完成的；如果要以其他用户身份建立应用上下文，则要求该用户必须具有 CREATE ANY CONTEXT 系统权限。示例如下：

```

SQL> conn system/manager
SQL> CREATE OR REPLACE CONTEXT empenv USING scott.ctx;

```

当执行了上述命令之后，就会建立名称为 EMPENV 的应用上下文，并且其属性由 SCOTT 方案的包 CTX 包进行设置。

### (2) 建立包过程设置应用上下文属性

在建立了应用上下文之后，就应该建立用于设置上下文属性的包了。如果会话用户为 JONES，则设置属性 DEPTNO 为 20；如果会话用户为 BLAKE，则设置属性 DEPTNO 为 30；而如果是其他会话用户，则设置属性 DEPTNO 为 10。示例如下：

```

CREATE OR REPLACE PACKAGE scott.ctx AS
    PROCEDURE set_deptno;
END;
/
CREATE OR REPLACE PACKAGE BODY scott.ctx AS
    PROCEDURE set_deptno IS
        id NUMBER;

```

```

BEGIN
  IF sys_context('userenv','session_user')='JONES' THEN
    DBMS_SESSION.SET_CONTEXT('empenv', 'deptno', 20);
  ELSIF sys_context('userenv','session_user')='BLAKE' THEN
    DBMS_SESSION.SET_CONTEXT('empenv', 'deptno', 30);
  ELSE
    DBMS_SESSION.SET_CONTEXT('empenv', 'deptno', 10);
  END IF;
END;
/

```

### (3) 建立登录触发器

用户登录到数据库之后，会自动触发登录触发器。建立登录触发器的目的是要隐含调用过程 ctx.set\_deptno，从而设置上下文属性。注意，必须要以 SYS 用户身份建立登录触发器。示例如下：

```

SQL> conn sys/oracle as sysdba
SQL> CREATE OR REPLACE TRIGGER login_trig
  2 AFTER LOGON ON DATABASE CALL scott.ctx.set_deptno
  3 /

```

如上所示，当用户登录到数据库时，会隐含调用过程 SCOTT.CTX.SET\_DEPTNO，并设置应用上下文 EMPENV 的属性 DEPTNO。

### (4) 建立策略函数

在增加策略之前，必须首先建立策略函数，并且策略函数必须带有两个参数，第一个参数对应于方案名，而第二个参数则对应于表名、视图名或同义词名。示例如下：

```

CREATE OR REPLACE PACKAGE scott.emp_security AS
  FUNCTION emp_sec(p1 VARCHAR2,p2 VARCHAR2) RETURN VARCHAR2;
END;
/
CREATE OR REPLACE PACKAGE BODY scott.emp_security AS
  FUNCTION emp_sec(p1 VARCHAR2,p2 VARCHAR2) RETURN VARCHAR2
  IS
    d_predicate VARCHAR2(2000);
  BEGIN
    IF user NOT IN ('SYS','SYSTEM','SCOTT') THEN
      d_predicate := 'deptno = SYS_CONTEXT(''empenv'', ''deptno'')';
      RETURN d_predicate;
    END IF;
    RETURN '1=1';
  END;
END;
/

```

如上所示，当以用户 JONES 登录时，谓词为“deptno=20”；当以用户 BLAKE 登录时，谓词为“deptno=30”；当以 SYS, SYSTEM, SCOTT 用户登录时，谓词为“1=1”；而当以其他用户身份登录时，谓词为“deptno=10”。

### (5) 增加策略

在建立了策略函数之后，就可以增加策略，并定义对象、策略、策略函数以及 SQL 语句之间的对应关系。增加策略是使用包 DBMS\_RLS 来完成的，示例如下：

```
SQL> EXECUTE DBMS_RLS.ADD_POLICY ('scott', 'emp', 'emp_policy', -
2> 'scott', 'emp_security.emp_sec', 'select');
```

在执行了过程 ADD\_POLICY 之后，会在系统默认的策略组 SYS\_DEFAULT 中增加策略 emp\_policy，并且在表 SCOTT.EMP 上的 SELECT 语句会使用该策略。其中，第一个参数为对象所在方案名，第二个参数为对象名，第三个参数为策略名，第四个参数为策略函数所在方案名，第五个参数为策略函数，第六个参数为使用该策略的 SQL 语句（如果不指定，则 SELECT, INSERT, UPDATE 和 DELETE 语句都会使用该策略）。当完成了以上步骤之后，就实现了精细访问控制。示例如下：

- SYS, SYSTEM, SCOTT 登录：当查询 SCOTT.EMP 表时，因为谓词为“1=1”，所以会将示例查询语句转变为“SELECT ename,deptno FROM scott.emp WHERE 1=1”，也即会返回 EMP 表的所有数据。示例如下：

SQL> conn scott/tiger
SQL> SELECT ename,deptno FROM scott.emp;
ENAME DEPTNO
-----
SMITH 20
ALLEN 30
CLARK 10
...

- JONES 登录：当查询 SCOTT.EMP 表时，因为谓词为“deptno=20”，所以会将示例查询语句转变为“SELECT ename,deptno FROM scott.emp WHERE deptno=20”，也即只会显示部门 20 的雇员信息。示例如下：

SQL> conn jones/jones
SQL> SELECT ename,deptno FROM scott.emp;
ENAME DEPTNO
-----
SMITH 20
JONES 20
SCOTT 20
ADAMS 20
FORD 20

- BLAKE 登录：当查询 SCOTT.EMP 表时，因为谓词为“deptno=30”，所以会将示例查询语句转变为“SELECT ename,deptno FROM scott.emp WHERE deptno=30”，也即只会显示部门 30 的雇员信息。示例如下：

```

SQL> conn blake/blake
SQL> SELECT ename,deptno FROM scott.emp;
ENAME          DEPTNO
-----
ALLEN          30
WARD           30
MARTIN         30
BLAKE          30
TURNER         30
JAMES          30

```

- 其他用户登录：当查询 SCOTT.EMP 表时，因为谓词为“deptno=10”，所以会将示例查询语句转变为“SELECT ename,deptno FROM scott.emp WHERE deptno=10”，也即只会显示部门 10 的雇员信息。示例如下：

```

SQL> conn /
SQL> SELECT ename,deptno FROM scott.emp;
ENAME          DEPTNO
-----
CLARK          10
KING           10
MILLER         10

```

## 17.9 DBMS\_DDL

该包提供了在 PL/SQL 块中执行 DDL 语句的方法，并且该包也提供了一些 DDL 的特殊管理方法。下面详细介绍该包所包含的过程和函数。

### 1. ALTER\_COMPILE

该过程用于重新编译过程、函数和包，语法如下：

```
DBMS_DDL.ALTER_COMPILE (
    type VARCHAR2, schema VARCHAR2, name VARCHAR2);
```

如上所示，type 用于指定对象类型（PROCEDURE, FUNCTION, PACKAGE, TRIGGER）；schema 用于指定对象所在方案；name 用于指定对象名。示例如下：

```
exec dbms_ddl.alter_compile('PROCEDURE',NULL,'ADD_EMP');
```

### 2. ANALYZE\_OBJECT

该过程用于分析表、索引和簇并生成统计数据，语法如下：

```
DBMS_DDL.ANALYZE_OBJECT (
    type VARCHAR2, schema VARCHAR2, name VARCHAR2,
    method VARCHAR2,
    estimate_rows NUMBER DEFAULT NULL,
    estimate_percent NUMBER DEFAULT NULL,
```

```
method_opt VARCHAR2 DEFAULT NULL,
partname VARCHAR2 DEFAULT NULL);
```

如上所示, type 用于指定对象类型 (TABLE, INDEX 或 CLUSTER); method 用于指定分析方法 (COMPUTE、ESTIMATE、DELETE); estimate\_rows 用于指定要估计的行数; estimate\_percent 用于指定要估计的百分比; method\_opt 用于指定分析方法选项 (FOR TABLE、FOR ALL COLUMNS 等); partname 用于指定要分析的分区。示例如下:

```
SQL> exec dbms_ddl.analyze_object('TABLE',NULL,-
> 'EMP','COMPUTE')
IS_TRIGGER_FIRE_ONCE;
```

该函数用于检测特定的 DML 或 DDL 触发器是否只触发一次。语法如下:

```
DBMS_DDL.IS_TRIGGER_FIRE_ONCE(
    trig_owner IN VARCHAR2,trig_name IN VARCHAR2)
RETURN BOOLEAN;
```

如上所示, trig\_owner 用于指定触发器所有者; trig\_name 用于指定触发器名。如果函数返回 TRUE, 则表示触发器只被触发一次。

```
SET_TRIGGER_FIRING_PROPERTY;
```

该过程用于设置 DML 或 DDL 触发器的触发属性, 语法如下:

```
DBMS_DDL.SET_TRIGGER_FIRING_PROPERTY(
    trig_owner IN VARCHAR2,trig_name IN VARCHAR2,
    fire_once IN BOOLEAN);
```

如上所示, fire\_once 用于指定触发器属性, 当设置为 TRUE 时只触发一次, 当设置为 FALSE 时总是被触发。

## 17.10 DBMS\_SHARED\_POOL

该包提供了对共享池的一些过程和函数访问, 它使用户可以显示共享池中的对象尺寸、绑定对象到共享池、清除绑定到共享池的对象。为了使用该包, 必须运行 dbmspool.sql 脚本来建立该包。下面介绍该包所包含的过程和函数。

### 1. SIZES

该过程用于显示在共享池中大于指定尺寸的对象, 语法如下:

```
DBMS_SHARED_POOL.SIZES (minsize NUMBER);
```

如上所示, minsize 用于指定要显示对象的最小尺寸 (单位: KB)。示例如下:

```
SQL> set serveroutput on
SQL> exec dbms_shared_pool.sizes(100);
SIZE(K) KEPT   NAME
-----
377 YES     SYS.STANDARD          (PACKAGE)
```

### 2. KEEP

该过程用于将特定对象绑定到共享池中, 语法如下:

```
DBMS_SHARED_POOL.KEEP (name VARCHAR2,flag CHAR DEFAULT 'P');
```

如上所示, name 用于指定要绑定的对象名; flag 用于指定对象类型 (P: 过程、函数和包; T: 对象类型; R: 触发器; Q: 序列)。示例如下:

```
SQL> exec dbms_shared_pool.keep('standard');
```

### 3. UNKEEP

该过程用于清除被绑定到共享池中的对象，语法如下：

```
DBMS_SHARED_POOL.UNKEEP (name VARCHAR2, flag CHAR DEFAULT 'P');
```

示例如下：

```
SQL> exec dbms_shared_pool.unkeep('standard');
```

### 4. ABORTED\_REQUEST\_THRESHOLD

该过程用于设置共享池终止请求的阈值，语法如下：

```
DBMS_SHARED_POOL.ABORTED_REQUEST_THRESHOLD (
    threshold_size NUMBER);
```

如上所示，`threshold_size` 用于指定共享池阈值尺寸（5000~20 字节）。示例如下：

```
SQL> exec dbms_shared_pool.aborted_request_threshold(10000)
```

## 17.11 DBMS\_RANDOM

该包提供了内置的随机数生成器，可以用于快速生成随机数。下面介绍该包所包含的过程和函数。

### 1. INITIALIZE

该过程用于初始化 DBMS\_RANDOM 包。在初始化 DBMS\_RANDOM 包时，必须要提供随机数种子。语法如下：

```
DBMS_RANDOM.INITIALIZE (seed IN BINARY_INTEGER);
```

如上所示，`seed` 用于指定随机数种子。

### 2. SEED

该过程用于复位随机数种子，语法如下：

```
DBMS_RANDOM.SEED (seed IN BINARY_INTEGER);
```

### 3. RANDOM

该函数用于生成随机数，语法如下：

```
DBMS_RANDOM.RANDOM RETURN BINARY_INTEGER;
```

### 4. TERMINATE

该过程用于关闭 DBMS\_RANDOM 包，语法如下：

```
DBMS_RANDOM.TERMINATE;
```

### 5. 随机数使用示例

下面以生成 10000 以内的 10 个随机数为例，说明使用 DBMS\_RANDOM 包生成随机数的方法。示例如下：

```
DECLARE
    num INT;
    seed NUMBER:=10000000;
BEGIN
    dbms_random.initialize(seed);
    FOR i IN 1..10 LOOP
        num:=abs(dbms_random.random()/seed);
        dbms_output.put_line(num);
    END LOOP;
END;
```

```

    END LOOP;
    dbms_random.terminate;
END;
/
18
144
124
137
199
130
36
44
137
138

```

## 17.12 DBMS\_LOGMNR

通过使用包 DBMS\_LOGMNR 和 DBMS\_LOGMNR\_D，可以分析重做日志和归档日志所记载的事务变化，最终确定误操作（例如 DROP TABLE）的时间、跟踪用户事务操作、跟踪并还原表的 DML 操作。下面介绍包 DBMS\_LOGMNR 和 DBMS\_LOGMNR\_D 所包含的过程和函数。

### 1. DBMS\_LOGMNR.ADD\_LOGFILE

该过程用于为日志分析列表增加或删除日志文件，或者建立日志分析列表。语法如下：

```

DBMS_LOGMNR.ADD_LOGFILE(
    LogFileName IN VARCHAR2,
    Options IN BINARY_INTEGER default ADDFILE );

```

如上所示，`logfilename` 用于指定要增加或删除的日志文件名称；`options` 用于指定选项（其中，DBMS\_LOGMNR.NEW：建立日志分析列表；DBMS\_LOGMNR.ADDFILE：增加日志文件；DBMS\_LOGMNR.REMOVEFILE：删除日志文件）。

### 2. DBMS\_LOGMNR.START\_LOGMNR

该过程用于启动 LogMiner 会话，语法如下：

```

DBMS_LOGMNR.START_LOGMNR(
    startScn IN NUMBER default 0, endScn IN NUMBER default 0,
    startTime IN DATE default '01-jan-1988',
    endTime IN DATE default '01-jan-2988',
    DictFileName IN VARCHAR2 default '',
    Options IN BINARY_INTEGER default 0 );

```

如上所示，`startscn` 用于指定日志分析的起始 SCN 值；`endscn` 用于指定日志分析的结束 SCN 值；`starttime` 用于指定日志分析的起始时间，`endtime` 用于指定日志分析的结束时间；`dictfilename` 用于指定日志分析要使用的字典文件名；`options` 用于指定 LogMiner 分析选项。

### 3. DBMS\_LOGMNR.END\_LOGMNR

该过程用于结束 LogMiner 会话，语法如下：

```
DBMS_LOGMNR.END_LOGMNR;
```

#### 4. DBMS\_LOGMNR.MINE\_VALUE

该函数用于返回要摘取的列信息，该函数在启动 LogMiner 之后调用。语法如下：

```
dbms_logmnr.mine_value(
    sql_redo_undo IN RAW,
    column_name IN VARCHAR2 default '') RETURN VARCHAR2;
```

如上所示，`sql_redo_undo` 用于指定要摘取的数据（`REDO_VALUE` 或 `UNDO_VALUE`）；`column_name` 用于指定要摘取的列（格式：`schema.table.column`）。

#### 5. DBMS\_LOGMNR.COLUMN\_PRESENT

该函数用于确定列是否出现在数据的 REDO 部分或 UNDO 部分，语法如下：

```
dbms_logmnr.column_present(
    sql_redo_undo IN RAW,
    column_name IN VARCHAR2 default '') RETURN NUMBER;
```

如果列在 REDO 或 UNDO 部分存在，则返回 1，否则返回 0。

#### 6. DBMS\_LOGMNR\_D.BUILD

该过程用于建立字典文件，语法如下：

```
DBMS_LOGMNR_D.BUILD (
    dictionary_filename IN VARCHAR2,
    dictionary_location IN VARCHAR2, options IN NUMBER);
```

如上所示，`dictionary_filename` 用于指定字典文件名；`dictionary_location` 用于指定文件所在位置；`options` 用于指定字典要写入位置（`STORE_IN_FLAT_FILE`：文本文件，`STORE_IN_REDO_LOGS`：重做日志）。

#### 7. DBMS\_LOGMNR\_D.SET\_TABLESPACE

该过程用于改变 LogMiner 表所在表空间，语法如下：

```
DBMS_LOGMNR_D.SET_TABLESPACE(
    new_tablespace IN DEFAULT VARCHAR2,
    dict_tablespace IN DEFAULT VARCHAR2,
    spill_tablespace IN DEFAULT VARCHAR2);
```

如上所示，`new_tablespace` 用于指定 LogMiner 表所在表空间；`dict_tablespace` 用于指定字典表所在表空间；`spill_tablespace` 用于指定溢出表所在表空间。

#### 8. LogMiner 使用示例

下面以分析 TEMP 表的 DDL 和 DML 操作为例，介绍使用 LogMiner 分析重做日志和归档日志的方法。在使用 LogMiner 之前，首先建立表 TEMP，然后执行 DML 操作和日志切换操作，生成归档日志。如下所示：

```
sqlplus /nolog
connect system/manager@test
CREATE TABLE temp
  (cola NUMBER, colb VARCHAR2(10));
ALTER SYSTEM SWITCH LOGFILE;
INSERT INTO temp VALUES(9, 'A');
UPDATE temp SET cola=10;
COMMIT;
ALTER SYSTEM SWITCH LOGFILE;
```

```
DELETE FROM temp;
ALTER SYSTEM SWITCH LOGFILE;
```

### (1) 建立字典文件

字典文件用于存放表及对象 ID 号之间的对应关系。从 Oracle9i 开始，字典信息既可被摘取到字典文件中，也可被摘取到重做日志中。摘取字典信息到字典文件的方法如下：

- 设置字典文件所在目录

```
SQL> ALTER SYSTEM SET utl_file_dir="g:\test"
      2 SCOPE=SPFILE;
```

- 重新启动 Oracle Server

```
sqlplus /nolog
conn sys/test@test as sysdba
SHUTDOWN IMMEDIATE
STARTUP
```

- 摘取字典信息

```
BEGIN
    dbms_logmnr_d.build (
        dictionary_filename=>'dict.ora',
        dictionary_location=>'g:\test\logminer');
END;
/
```

### (2) 建立日志分析列表

- 停止 Oracle Server 并装载数据库

```
sqlplus /nolog
CONN sys/test@test AS SYSDBA
SHUTDOWN IMMEDIATE
STARTUP MOUNT
```

- 建立日志分析列表

```
BEGIN
    dbms_logmnr.add_logfile(
        options=>dbms_logmnr.NEW,
        logfilename=>'g:\test\arc1\test11.arc');
END;
/
```

- 增加其他日志文件（可选）

```
BEGIN
    dbms_logmnr.add_logfile(
        options=>dbms_logmnr.ADDFILE,
        logfilename=>'g:\test\arc1\test12.arc');
END;
/
```

### (3) 启动 LogMiner 执行分析

```
BEGIN
    dbms_logmnr.start_logmnr(
        DictFileName=>'g:\test\logminer\dict.ora',
        STARTTIME=>TO_DATE('2004-04-03:10:10:00',
```

```

        'YYYY-MM-DD:HH24:MI:SS'),
ENDTIME=>TO_DATE('2004-04-03:15:30:00',
        'YYYY-MM-DD:HH24:MI:SS')
);

```

#### (4) 查看日志分析结果

在启动 LogMiner 执行了日志分析之后，就可以查看日志分析结果了。但是要注意，日志分析结果只能在当前会话查看，而其他会话将不能查看到日志分析结果。

- 显示 DML 结果

SELECT operation,sql_redo,sql_undo FROM v\$logmnr_contents WHERE seg_name='TEMP';		
OPERATION	SQL_REDO	SQL_UNDO
INSERT	insert into SYSTEM.TEMP ...	delete from SYSTEM.TEMP ...
UPDATE	update SYSTEM.TEMP.....	update SYSTEM.TEMP ...
DELETE	delete from SYSTEM.TEMP ...	insert into SYSTEM.TEMP ...

- 显示 DDL 结果

SELECT to_char(timestamp,'YYYY-MM-DD:HH24:MI:SS') time,sql_redo FROM v\$logmnr_contents WHERE sql_redo LIKE '%create%' OR sql_redo LIKE '%CREATE%';	
TIME	SQL_RED0
2002-04-03:15:43:32	CREATE TABLE temp (cola NUMBER,colb VARCHAR2(10));

- 显示在用字典文件

SELECT db_name,filename FROM v\$logmnr_dictionary;	
DB_NAME	FILENAME
TEST	g:\test\logminer\dict.ora

#### (5) 结束 LogMiner

```
SQL> execute dbms_logmnr.end_logmnr;
```

### 17.13 DBMS\_FLASHBACK

该包用于激活或禁止会话的 FLASHBACK 特征，为了使得普通用户可以使用该包，必须

要将执行该包的权限授予这些用户。示例如下：

```
SQL> conn sys/admin as sysdba
已连接。
SQL> GRANT EXECUTE ON dbms_flashback TO SCOTT;
```

### 1. ENABLE\_AT\_TIME

该过程用于以时间方式激活会话的 FLASHBACK，语法如下：

```
DBMS_FLASHBACK.ENABLE_AT_TIME (query_time IN TIMESTAMP);
```

如上所示，query\_time 用于指定 FLASHBACK 对应的时间点。

### 2. ENABLE\_AT\_SYSTEM\_CHANGE\_NUMBER

该过程用于以系统改变号 (SCN) 方式激活会话的 FLASHBACK，语法如下：

```
DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER (
    query_scn IN NUMBER);
```

如上所示，query\_scn 用于指定 FLASHBACK 对应的 SCN 值。

### 3. GET\_SYSTEM\_CHANGE\_NUMBER

该函数用于取得系统的当前 SCN 值，语法如下：

```
DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER RETURN NUMBER;
```

### 4. DISABLE

该过程用于禁止会话的 FLASHBACK 模式，语法如下：

```
DBMS_FLASHBACK.DISABLE;
```

### 5. FLASHBACK 使用示例

#### (1) 取得 SCOTT 雇员的工资及系统的 SCN 值

```
SQL> conn scott/tiger
SQL> SELECT sal FROM emp WHERE ename='SCOTT';
      SAL
-----
      3630
SQL> SELECT dbms_flashback.get_system_change_number()
      2  FROM dual;
DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER()
-----
717402
```

#### (2) 更新 SCOTT 工资，并休眠 5 分钟

```
SQL> UPDATE emp SET sal=3000 WHERE ename='SCOTT';
已更新 1 行。
SQL> commit;
SQL> exec dbms_lock.sleep(300)
```

#### (3) 使用 DBMS\_FLASHBACK 取得特定 SCN 时间点对应的数据

```
SQL> exec dbms_flashback.enable_at_system_change_number(717402)
PL/SQL 过程已成功完成。
SQL> SELECT sal FROM emp WHERE ename='SCOTT';
      SAL
-----
      3630
```

```

SQL> exec dbms_flashback.disable
PL/SQL 过程已成功完成。
SQL> SELECT sal FROM emp WHERE ename='SCOTT';
      SAL
-----
      3000

```

## 17.14 DBMS\_OBFUSCATION\_TOOLKIT

该包用于加密和解密应用数据，另外还可以生成密码校验和。通过加密输入数据，可以防止黑客或其他用户窃取私有数据；而通过结合使用加密和密码校验和，可以防止黑客破坏被加密的数据。当使用该包加密数据时，要求被加密数据的长度必须为8字节的整数倍。当使用DES算法加密数据时，密钥长度不能低于8字节；当使用DES3算法加密数据时，密钥长度不能低于16字节。下面给大家介绍该包所包含的过程和函数。

### 1. DESEncrypt

该过程用于使用DES算法对输入数据进行加密，并生成加密格式的数据。当使用该过程加密数据时，密钥不能少于8个字符，并且输入数据必须是8字节的整数倍。语法如下：

```

DBMS_OBFUSCATION_TOOLKIT.DESENCRYPT(
    .input RAW, key RAW, encrypted_data OUT RAW);
DBMS_OBFUSCATION_TOOLKIT.DESENCRYPT(
    input_string VARCHAR2, key_string VARCHAR2,
    encrypted_string OUT VARCHAR2);

```

如上所示，`input_string`(`input`)用于指定输入字符串或输入的二进制数据；`key_string`(`key`)用于指定加密密钥；`encrypted_string`(`encrypted_data`)用于指定存放加密结果的字符串。示例如下：

```

DECLARE
    encrypted_string VARCHAR2(100);
BEGIN
    dbms_obfuscation_toolkit.desencrypt(
        input_string=>'SCOTTsco', key_string=>'abcd1234',
        encrypted_string=>encrypted_string);
    dbms_output.put_line(encrypted_string);
END;
/
□D 谎诺？

```

### 2. DESDecrypt

该过程用于对使用DES算法所生成的加密数据进行解密。当对数据进行解密时，解密密钥必须要与加密密钥完全一致。语法如下：

```

DBMS_OBFUSCATION_TOOLKIT.DESDECRYPT(
    input RAW, key RAW, decrypted_data OUT RAW);
DBMS_OBFUSCATION_TOOLKIT.DESDECRYPT (
    input_string VARCHAR2, key_string VARCHAR2,
    decrypted_string OUT VARCHAR2);

```

如上所示, decrypted\_string ( decrypted\_data ) 用于指定存放解密结果的字符串。示例如下:

```

DECLARE
    encrypted_string VARCHAR2(100);
    decrypted_string VARCHAR2(100);
BEGIN
    dbms_obfuscation_toolkit.desencrypt(
        input_string=>'SCOTTsco',key_string=>'abcd1234',
        encrypted_string=>encrypted_string);
    dbms_obfuscation_toolkit.desdecrypt(
        input_string=>encrypted_string,key_string=>'abcd1234',
        decrypted_string=>decrypted_string);
    dbms_output.put_line(decrypted_string);
END;
/
SCOTTsco

```

### 3. DES3Encrypt

该过程用于使用 DES3 算法对输入数据进行加密，并生成加密格式的数据。当使用该过程加密数据时，密钥不能少于 16 个字符，并且输入数据必须是 8 字节的整数倍。语法如下：

```

DBMS_OBFUSCATION_TOOLKIT.DES3ENCRYPT(
    input RAW,key RAW,encrypted_data OUT RAW);
DBMS_OBFUSCATION_TOOLKIT.DES3ENCRYPT(
    input_string VARCHAR2,key_string VARCHAR2,
    encrypted_string OUT VARCHAR2);

```

示例如下：

```

DECLARE
    str1 VARCHAR2(8):='中国你好';
    key VARCHAR2(16):='ABCFDSDSADDS1234';
    str2 VARCHAR2(100);
BEGIN
    dbms_obfuscation_toolkit.des3encrypt(
        input_string=>str1,key_string=>key,
        encrypted_string=>str2);
    dbms_output.put_line(str2);
END;
/
桐埃?蝶

```

### 4. DES3Decrypt

该过程用于对使用 DES3 算法所生成的加密数据进行解密。在对数据进行解密时，解密密钥必须要与加密密钥完全一致。语法如下：

```

DBMS_OBFUSCATION_TOOLKIT.DES3DECRYPT(
    input RAW,key RAW,decrypted_data OUT RAW);
DBMS_OBFUSCATION_TOOLKIT.DES3DECRYPT (
    input_string VARCHAR2,key_string VARCHAR2,
    decrypted_string OUT VARCHAR2);

```

示例如下：

```

DECLARE
    str1 VARCHAR2(8):='中国你好';
    key VARCHAR2(16):='ABCDEFSDSADDS1234';
    str2 VARCHAR2(100);
    str3 VARCHAR2(100);
BEGIN
    dbms_obfuscation_toolkit.des3encrypt(
        input_string=>str1,key_string=>key,
        encrypted_string=>str2);
    dbms_obfuscation_toolkit.des3decrypt(
        input_string=>str2,key_string=>key,
        decrypted_string=>str3);
    dbms_output.put_line(str3);
END;
/
中国你好

```

## 5. MD5

该过程用于使用 MD5 算法生成密码校验和。通过使用密码校验和，可以防止其他用户破坏被传输的加密数据。语法如下：

```

DBMS_OBFUSCATION_TOOLKIT.MD5(
    input RAW,checksum OUT RAW);
DBMS_OBFUSCATION_TOOLKIT.DES3DECRYPT (
    input_string VARCHAR2,checksum_string OUT VARCHAR2);

```

如上所示，`checksum_string` (`checksum`) 用于指定存放密码校验和的字符串。示例如下：

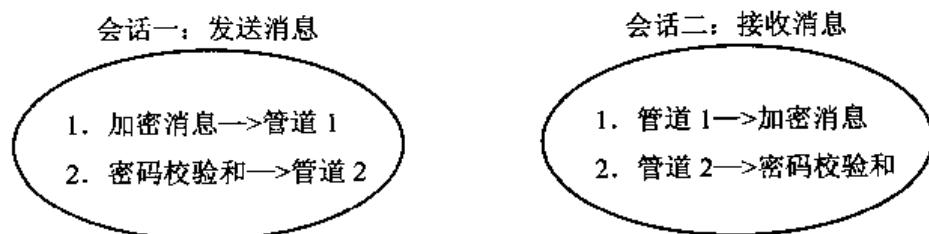
```

DECLARE
    str1 VARCHAR2(8):='中国你好';
    str2 VARCHAR2(100);
BEGIN
    dbms_obfuscation_toolkit.md5(
        input_string=>str1,checksum_string=>str2);
    dbms_output.put_line(str2);
END;
/
枕 I 潛螢鄭?~麓雉

```

## 6. DBMS\_OBFUSCATION\_TOOLKIT 使用示例

为了防止黑客窃取数据，应该对数据进行加密；而为了防止黑客篡改数据，则应该使用密码校验来确保数据的正确性。下面以使用管道发送加密消息，并确保消息的正确性为例，说明包 DBMS\_OBFUSCATION\_TOOLKIT 的使用方法。如下图所示：



### 示例一：建立过程 send\_message

过程 send\_message 将用于生成消息的密码校验和、加密消息，并且会将密码校验和发送到管道 CHECKSUM，而将加密消息发送到管道 ENCRYPT。注意，当使用该过程为管道发送消息时，消息长度必须为 8 字节的整数倍。示例如下：

```
CREATE OR REPLACE PROCEDURE send_message(
    message VARCHAR2)
IS
    flag INT;
    checksum VARCHAR2(100);
    key VARCHAR2(100) := '123456778SAD';
    encry_str VARCHAR2(100);
BEGIN
    /* 使用 MD5 算法为消息生成密码校验和 */
    dbms_obfuscation_toolkit.md5(
        input_string=>message,checksum_string=>checksum);
    /* 建立管道 checksum，并将密码校验和发送到该管道 */
    flag:=dbms_pipe.create_pipe('checksum');
    IF flag=0 THEN
        dbms_pipe.pack_message(checksum);
        flag:=dbms_pipe.send_message('checksum');
    END IF;
    /* 加密要发送的消息 */
    dbms_obfuscation_toolkit.desencrypt(
        input_string=>message,key_string=>key,
        encrypted_string=>encry_str);
    /* 建立管道 encrypt，并将加密消息发送到该管道 */
    flag:=dbms_pipe.create_pipe('encrypt');
    IF flag=0 THEN
        dbms_pipe.pack_message(encry_str);
        flag:=dbms_pipe.send_message('encrypt');
    END IF;
END;
/
```

### 示例二：建立过程 receive\_message

过程 receive\_message 用于接收管道 checksum 的密码校验和、管道 encrypt 的加密消息，然后对加密消息进行解密，并且生成解密消息的密码校验和，然后比较两种密码校验和，以确定消息是否被篡改。如果消息没有被篡改，则输出消息；如果消息被篡改，则显示“消息被篡改”。示例如下：

```
CREATE OR REPLACE PROCEDURE receive_message
IS
    flag INT;
    source_checksum VARCHAR2(100);
    dest_checksum VARCHAR2(100);
    key VARCHAR2(100) := '123456778SAD';
    encry_str VARCHAR2(100);
```

```

decry_str VARCHAR2(100);
BEGIN
    /* 从管道 encrypt 中接收加密消息 */
    flag:=dbms_pipe.receive_message('encrypt');
    IF flag=0 THEN
        dbms_pipe.unpack_message(encry_str);
        flag:=dbms_pipe.remove_pipe('encrypt');
    END IF;
    /* 从管道 checksum 中接收密码校验和 */
    flag:=dbms_pipe.receive_message('checksum');
    IF flag=0 THEN
        dbms_pipe.unpack_message(source_checksum);
        flag:=dbms_pipe.remove_pipe('checksum');
    END IF;
    /* 使用密钥解密消息，并生成密码校验和 */
    dbms_obfuscation_toolkit.desdecrypt(
        input_string=>encry_str,key_string=>key,
        decrypted_string=>decry_str);
    dbms_obfuscation_toolkit.md5(
        input_string=>decry_str,checksum_string=>dest_checksum);
    /* 比较密码校验和，如果相同，则显示消息；否则显示消息被篡改 */
    IF trim(source_checksum)=trim(dest_checksum) THEN
        dbms_output.put_line(decry_str);
    ELSE
        dbms_output.put_line('消息被篡改');
    END IF;
END;
/

```

### 示例三：使用过程 send\_message

会话一：SQL> exec send\_message('中国你好')  
 会话二：  
 SQL> set serveroutput on  
 SQL> exec soott.receive\_message  
 中国你好

## 17.15 DBMS\_SPACE

包DBMS\_SPACE用于分析段增长和空间的需求，下面介绍该包所包含的过程和函数。

### 1. UNUSED\_SPACE

该过程用于返回对象（表、索引、簇）的未用空间，语法如下：

```

DBMS_SPACE.UNUSED_SPACE (
    segment_owner IN VARCHAR2,segment_name IN VARCHAR2,
    segment_type IN VARCHAR2,total_blocks OUT NUMBER,
    total_bytes OUT NUMBER,unused_blocks OUT NUMBER,
    unused_bytes OUT NUMBER,last_used_extent_file_id OUT NUMBER,

```

```

    last_used_extent_block_id OUT NUMBER,
    last_used_block OUT NUMBER,
    partition_name IN VARCHAR2 DEFAULT NULL);

```

如上所示, segment\_owner 用于指定段所有者; segment\_name 用于指定段名; segment\_type 用于指定段类型; total\_blocks 用于返回段的总计块个数; total\_bytes 用于返回段的总计字节数; unused\_blocks 用于返回段的未用块个数; unused\_bytes 用于返回段的未用字节数; last\_used\_extent\_file\_id 用于返回包含数据的最后一个区所在文件的编号; last\_used\_block 用于返回包含数据的最后一个区的块编号; last\_used\_block 用于返回包含数据的最后一个区的最后一个块; partition\_name 用于指定要分析的段分区名。示例如下:

```

DECLARE
    total_blocks NUMBER;
    total_bytes NUMBER;
    unused_blocks NUMBER;
    unused_bytes NUMBER;
    last_used_extent_file_id NUMBER;
    last_used_extent_block_id NUMBER;
    last_used_block NUMBER;
BEGIN
    dbms_space.unused_space('SYSTEM','T1','TABLE',
        total_blocks,total_bytes,unused_blocks,
        unused_bytes,last_used_extent_file_id,
        last_used_extent_block_id,last_used_block);
    dbms_output.put_line('HWM=' ||
        TO_CHAR(total_blocks-unused_blocks-1));
END;
/
HWM=2

```

## 2. FREE\_BLOCKS

该过程用于返回对象(表、索引、簇)的空闲块信息, 语法如下:

```

DBMS_SPACE.FREE_BLOCKS (
    segment_owner IN VARCHAR2,segment_name IN VARCHAR2,
    segment_type IN VARCHAR2,freelist_group_id IN NUMBER,
    free_blks OUT NUMBER,scan_limit IN NUMBER DEFAULT NULL,
    partition_name IN VARCHAR2 DEFAULT NULL);

```

如上所示, freelist\_group\_id 用于指定段的空闲列表组号; free\_blks 用于返回空闲列表组所对应的空闲列表个数; scan\_limit 用于指定要读取的空闲列表块的最大个数; 示例如下:

```

DECLARE
    free_blocks NUMBER;
BEGIN
    dbms_space.free_blocks('SYSTEM','T1','TABLE'
        ,0,free_blocks);
    dbms_output.put_line('组 0 的空闲列表个数:' ||
        ||free_blocks);
END;

```

```
/  
组0的空闲列表个数:1
```

### 3. SPACE\_USAGE

该过程用于显示段 HWM (High Water Mark) 以下数据块的空间使用情况，并且该过程只适用于自动段空间管理的表空间。语法如下：

```
DBMS_SPACE.SPACE_USAGE(  
    segment_owner IN varchar2, segment_name IN varchar2,  
    segment_type IN varchar2, unformatted_blocks OUT number,  
    unformatted_bytes OUT number,  
    fs1_blocks OUT number, fs1_bytes OUT number,  
    fs2_blocks OUT number, fs2_bytes OUT number,  
    fs3_blocks OUT number, fs3_bytes OUT number,  
    fs4_blocks OUT number, fs4_bytes OUT number,  
    full_blocks OUT number, full_bytes OUT number,  
    partition_name IN varchar2 DEFAULT NULL);
```

如上所示，`unformatted_blocks` 用于返回未格式化块的个数；`unformatted_bytes` 用于返回未格式化的字节数；`fs1_blocks` 用于返回空闲空间在 0~25% 之间的块个数；`fs1_bytes` 用于返回空闲空间在 0~25% 之间的字节数；`fs2_blocks` 用于返回空闲空间在 25%~50% 之间的块个数；`fs2_bytes` 用于返回空闲空间在 25%~50% 之间的字节数；`fs3_blocks` 用于返回空闲空间在 50%~75% 之间的块个数；`fs3_bytes` 用于返回空闲空间在 50%~75% 之间的字节数；`fs4_blocks` 用于返回空闲空间在 75%~100% 之间的块个数；`fs4_bytes` 用于返回空闲空间在 75%~100% 之间的字节数；`full_blocks` 用于返回段的总计块个数；`full_bytes` 用于返回段的总计字节数；示例如下：

```
variable unf number  
variable unfb number  
variable fs1 number  
variable fs1b number  
variable fs2 number  
variable fs2b number  
variable fs3 number  
variable fs3b number  
variable fs4 number  
variable fs4b number  
variable full number  
variable fullb number  
begin  
    dbms_space.space_usage('U1','T','TABLE',  
        :unf, :unfb,:fs1, :fs1b,:fs2, :fs2b,  
        :fs3, :fs3b,:fs4, :fs4b,:full, :fullb);  
end;  
/  
print unf  
print unfb  
print fs4
```

```

print fs4b
print fs3
print fs3b
print fs2
print fs2b
print fs1
print fs1b
print full
print fullb

```

## 17.16 DBMS\_SPACE\_ADMIN

包 DBMS\_SPACE\_ADMIN 提供了局部管理表空间的功能，下面给大家介绍该包所包含的过程和函数。

### 1. SEGMENT\_VERIFY

该过程用于检查段的区映像是否与位图一致，语法如下：

```

DBMS_SPACE_ADMIN.SEGMENT_VERIFY (
    tablespace_name IN VARCHAR2,
    header_relative_file IN POSITIVE,
    header_block IN POSITIVE,
    verify_option IN POSITIVE DEFAULT SEGMENT_VERIFY_EXTENTS);

```

如上所示，tablespace\_name 用于指定段所在表空间；header\_relative\_file 用于指定段头所在的相对文件号；header\_block 用于指定段头所在的块号；verify\_option 用于指定检查方式。示例如下：

```
SQL> exec dbms_space_admin.segment_verify('USERS3', 9, 68)
```

### 2. SEGMENT\_CORRUPT

该过程用于将段标记为损坏或有效，语法如下：

```

DBMS_SPACE_ADMIN.SEGMENT_CORRUPT (
    tablespace_name IN VARCHAR2, header_relative_file IN POSITIVE,
    header_block IN POSITIVE,
    corrupt_option IN POSITIVE DEFAULT SEGMENT_MARK_CORRUPT);

```

如上所示，corrupt\_option 用于指定损坏（SEGMENT\_MARK\_CORRUPT）或有效（SEGMENT\_MARK\_VALID）选项。示例如下：

```
SQL> exec dbms_space_admin.segment_corrupt('USERS3', 9, 68)
```

### 3. SEGMENT\_DROP\_CORRUPT

该过程用于删除被标记为损坏的段，语法如下：

```

DBMS_SPACE_ADMIN.SEGMENT_DROP_CORRUPT (
    tablespace_name IN VARCHAR2,
    header_relative_file IN POSITIVE, header_block IN POSITIVE);

```

示例如下：

```
SQL> exec dbms_space_admin.segment_drop_corrupt('USERS3', 9, 68)
```

### 4. SEGMENT\_DUMP

该过程用于转储特定段的头块和区映像块，语法如下：

```
DBMS_SPACE_ADMIN.SEGMENT_DUMP (
    tablespace_name IN VARCHAR2,
    header_relative_file IN POSITIVE,
    header_block IN POSITIVE,
    dump_option IN POSITIVE DEFAULT SEGMENT_DUMP_EXTENT_MAP);
```

如上所示，dump\_option 用于指定转储选项。示例如下：

```
SQL> exec dbms_space_admin.segment_dump('USERS3', 9, 68)
```

## 5. TABLESPACE\_VERIFY

该过程用于检查表空间所有段的位图和区映像，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_VERIFY (
    tablespace_name IN VARCHAR2,
    verify_option IN POSITIVE DEFAULT TABLESPACE_VERIFY_BITMAP);
```

示例如下：

```
SQL> exec dbms_space_admin.tablespace_verify('USERS3')
```

## 6. TABLESPACE\_FIX\_BITMAPS

该过程用于将特定范围的空间标记为空闲或已用，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_FIX_BITMAPS (
    tablespace_name IN VARCHAR2,
    dbarange_relative_file IN POSITIVE,
    dbarange_begin_block IN POSITIVE,
    dbarange_end_block IN POSITIVE,
    fix_option IN POSITIVE);
```

如上所示，dbarange\_relative\_file 用于指定 DBA 范围内的相对文件号；dbarange\_begin\_block 用于指定数据文件区的起始块编号；dbarange\_end\_block 用于指定数据文件区的结束块编号；fix\_option 用于指定选项（TABLESPACE\_EXTENT\_MAKE\_FREE 或 TABLESPACE\_EXTENT\_MAKE\_USED）。示例如下：

```
EXEC DBMS_SPACE_ADMIN.TABLESPACE_FIX_BITMAPS('USERS', 4, 33, 83, 7);
```

## 7. TABLESPACE\_REBUILD\_BITMAPS

该过程用于重新建立合适的位图。如果没有指定位图块，则将重建特定表空间的所有位图块。语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_REBUILD_BITMAPS (
    tablespace_name IN VARCHAR2,
    bitmap_relative_file IN POSITIVE DEFAULT NULL,
    bitmap_block IN POSITIVE DEFAULT NULL);
```

如上所示，bitmap\_relative\_file 用于指定位图块的相对文件号；bitmap\_block 用于指定位图块的块号。示例如下：

```
exec dbms_space_admin.tablespace_rebuild_bitmaps('USERS3')
```

## 8. TABLESPACE\_REBUILD\_QUOTAS

该过程用于重建表空间配额，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_REBUILD_QUOTAS (
    tablespace_name IN VARCHAR2);
```

示例如下：

```
EXEC DBMS_SPACE_ADMIN.TABLESPACE_REBUILD_QUOTAS('USERS3')
```

## 9. TABLESPACE\_MIGRATE\_FROM\_LOCAL

该过程用于将局部管理表空间转变为字典管理表空间，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_FROM_LOCAL (
    tablespace_name IN VARCHAR2);
```

示例如下：

```
EXEC DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_FROM_LOCAL ('USERS1')
```

## 10. TABLESPACE\_MIGRATE\_TO\_LOCAL

该过程用于将字典管理表空间转变为局部管理表空间，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_TO_LOCAL (
    tablespace_name IN VARCHAR2);
```

如上所示，`tablespace_name` 用于指定字典管理表空间名。示例如下：

```
EXEC DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_TO_LOCAL ('USERS1')
```

## 11. TABLESPACE\_RELOCATE\_BITMAPS

该过程用于移动位图到指定位置，语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_RELOCATE_BITMAPS (
    tablespace_name IN VARCHAR2,
    relative_fno IN BINARY_INTEGER,
    block_number IN BINARY_INTEGER)
```

如上所示，`relative_fno` 用于指定相对文件号；`block_number` 用于指定数据块编号。示例如下：

```
SQL> exec DBMS_SPACE_ADMIN.TABLESPACE_RELOCATE_BITMAPS-
> ('USERS3', 9, 8)
```

## 12. TABLESPACE\_FIX\_SEGMENT\_STATES

该过程用于修正表空间的段状态。当升级表空间时，如果出现例程终止，那么为了重新升级表空间，必须要修正该表空间中相应段的状态。语法如下：

```
DBMS_SPACE_ADMIN.TABLESPACE_FIX_SEGMENT_STATES (
    tablespace_name);
```

示例如下：

```
exec DBMS_SPACE_ADMIN.TABLESPACE_FIX_SEGMENT_STATES ('USERS3')
```

## 17.17 DBMS\_TTS

包 `DBMS_TTS` 用于检查表空间集合是否是自包含的，并在执行了检查之后，将违反自包含规则的信息写入到临时表 `TRANSPORT_SET_VIOLATIONS` 中。下面介绍该包所包含的过程和函数。

### 1. TRANSPORT\_SET\_CHECK

该过程用于检查表空间集合是否是自包含的，语法如下：

```
DBMS_TTS.TRANSPORT_SET_CHECK (
    ts_list IN VARCHAR2, incl_constraints IN BOOLEAN DEFAULT,
    full_closure IN BOOLEAN DEFAULT FALSE);
```

如上所示，`ts_list` 用于指定表空间列表，如果要指定多个表空间，则表空间之间使用逗号隔开；`incl_constraints` 用于指定是否要检查完整性约束；`full_closure` 用于指定是否要进行完全

或部分相关性检查。示例如下：

```
SQL> exec sys.dbms_tts.transport_set_check('users1,users2')
PL/SQL 过程已成功完成。
```

```
SQL> SELECT * FROM sys.transport_setViolations;
VIOLATIONS
```

```
-----  
Sys owned object A3 in tablespace USERS2 not allowed in pluggable set  
Sys owned object A4 in tablespace USERS2 not allowed in pluggable set
```

如上所示，在查询临时表 `transport_setViolations` 时，如果有返回信息，则会显示违反自包含表空间规则的原因；如果没有返回行，则表示表空间是自包含的。

## 2. DOWNGRADE

该过程用于降级搬移表空间的相关数据，语法如下：

```
DBMS_TTS.DOWNGRADE;
```

## 17.18 DBMS\_REPAIR

包 `DBMS_REPAIR` 用于检测、修复在表和索引上的损坏数据块，下面介绍该包所包含的过程和函数。

### 1. ADMIN\_TABLES

该过程提供了管理修复表和孤表的功能，语法如下：

```
DBMS_REPAIR.ADMIN_TABLES (
    table_name IN VARCHAR2, table_type IN BINARY_INTEGER,
    action IN BINARY_INTEGER, tablespace IN VARCHAR2 DEFAULT NULL);
```

如上所示，`table_name` 用于指定要处理的表名，必须要指定前缀 `ORPHAN` 或 `REPAIR`；`table_type` 用于指定表类型（`ORPHAN_TABLE` 或 `REPAIR_TABLE`）；`action` 用于指定要执行的管理操作（`CREATE_ACTION`：建立表；`PURGE_ACTION`：删除所有行；`DROP_ACTION`：删除表）；`tablespace` 用于指定表所在表空间。示例如下：

```
SQL> exec dbms_repair.admin_tables('REPAIR_TABLE',-
> DBMS_REPAIR.REPAIR_TABLE, DBMS_REPAIR.CREATE_ACTION, 'SYSTEM')
SQL> exec dbms_repair.admin_tables('ORPHAN_TABLE',-
> DBMS_REPAIR.ORPHAN_TABLE, DBMS_REPAIR.CREATE_ACTION, 'SYSTEM')
```

如上所示，在执行了第一条命令之后，就会建立修复表 `REPAIR_TABLE`，并且该修复表被用于存放损坏数据块的信息；在执行了第二条语句之后，会建立孤表 `ORPHAN_TABLE`，并且将该表用于存放指向损坏数据块的索引入口信息。

### 2. CHECK\_OBJECT

该过程用于检查特定对象，并将损坏信息填写到修复表中。语法如下：

```
DBMS_REPAIR.CHECK_OBJECT (
    schema_name IN VARCHAR2, object_name IN VARCHAR2,
    partition_name IN VARCHAR2 DEFAULT NULL,
    object_type IN BINARY_INTEGER DEFAULT TABLE_OBJECT,
    repair_table_name IN VARCHAR2 DEFAULT 'REPAIR_TABLE',
    flags IN BINARY_INTEGER DEFAULT NULL,
```

```

relative_fno IN BINARY_INTEGER DEFAULT NULL,
block_start IN BINARY_INTEGER DEFAULT NULL,
block_end IN BINARY_INTEGER DEFAULT NULL,
corrupt_count OUT BINARY_INTEGER);

```

如上所示，schema\_name 用于指定要检查对象的方案名；object\_name 用于指定要检查的对象名；partition\_name 用于指定要检查的分区名；object\_type 用于指定要检查对象的类型（TABLE\_OBJECT 或 INDEX\_OBJECT）；repair\_table\_name 用于指定要被填写的修复表名；flags 为将来使用而保留；relative\_fno 用于指定相对文件号；block\_start 用于指定要检查的起始块号；block\_end 用于指定要检查的结束块号；corrupt\_count 用于返回损坏的块个数。示例如下：

```

SQL> VAR corr_count NUMBER
SQL> exec dbms_repair.check_object('SCOTT','EMP',-
> corrupt_count=>:corr_count)
PL/SQL 过程已成功完成。
SQL> print corr_count

```

### 3. DUMP\_ORPHAN\_KEYS

该过程用于报告指向损坏数据块行的索引入口，并且会将相应索引入口的信息插入到孤表中。语法如下：

```

DBMS_REPAIR.DUMP_ORPHAN_KEYS (
    schema_name IN VARCHAR2, object_name IN VARCHAR2,
    partition_name IN VARCHAR2 DEFAULT NULL,
    object_type IN BINARY_INTEGER DEFAULT INDEX_OBJECT,
    repair_table_name IN VARCHAR2 DEFAULT 'REPAIR_TABLE',
    orphan_table_name IN VARCHAR2 DEFAULT 'ORPHAN_KEYS_TABLE',
    flags IN BINARY_INTEGER DEFAULT NULL,
    key_count OUT BINARY_INTEGER);

```

如上所示，object\_type 用于指定对象类型（INDEX\_OBJECT）；repair\_table\_name 用于指定修复表名；orphan\_table\_name 用于指定孤表名；key\_count 用于返回索引入口个数。示例如下：

```

SQL> VAR key_count NUMBER
SQL> exec dbms_repair.dump_orphan_keys('SCOTT',-
> 'PK_EMP',orphan_table_name=>'ORPHAN_TABLE',-
> key_count=>:key_count)
PL/SQL 过程已成功完成。
SQL> PRINT key_count

```

### 4. FIX\_CORRUPT\_BLOCKS

该过程用于修复被损坏的数据块，这些被损坏的数据块是在执行了 CHECK\_OBJECT 之后生成的。语法如下：

```

DBMS_REPAIR.FIX_CORRUPT_BLOCKS (
    schema_name IN VARCHAR2, object_name IN VARCHAR2,
    partition_name IN VARCHAR2 DEFAULT NULL,
    object_type IN BINARY_INTEGER DEFAULT TABLE_OBJECT,
    repair_table_name IN VARCHAR2 DEFAULT 'REPAIR_TABLE',
    flags IN BINARY_INTEGER DEFAULT NULL,
    fix_count OUT BINARY_INTEGER);

```

如上所示，`object_type` 用于指定对象类型（`TABLE_OBJECT`）；`fix_count` 用于返回修复的数据块个数。示例如下：

```
SQL> VAR fix_count NUMBER
SQL> exec dbms_repair.fix_corrupt_blocks('SCOTT',-
> 'EMP',fix_count=>:fix_count)
PL/SQL 过程已成功完成。
SQL> PRINT fix_count
```

## 5. REBUILD\_FREELISTS

该过程用于重建指定对象的空闲列表，语法如下：

```
DBMS_REPAIR.REBUILD_FREELISTS (
    schema_name IN VARCHAR2, object_name IN VARCHAR2,
    partition_name IN VARCHAR2 DEFAULT NULL,
    object_type IN BINARY_INTEGER DEFAULT TABLE_OBJECT);
```

如上所示，`object_type` 用于指定对象类型（`TABLE_OBJECT`）。示例如下：

```
SQL> exec dbms_repair.rebuild_freelists('SCOTT','EMP')
```

## 6. SKIP\_CORRUPT\_BLOCKS

该过程用于指定在扫描对象（表或索引）时跳过损坏块，语法如下：

```
DBMS_REPAIR.SKIP_CORRUPT_BLOCKS (
    schema_name IN VARCHAR2, object_name IN VARCHAR2,
    object_type IN BINARY_INTEGER DEFAULT TABLE_OBJECT,
    flags IN BINARY_INTEGER DEFAULT SKIP_FLAG);
```

如上所示，`object_type` 用于指定对象类型（`TABLE_OBJECT`）；`flags` 用于指定是否要跳过损坏块（`SKIP_FLAG`: 跳过损坏块；`NO_SKIP_FLAG`: 不跳过损坏块）。示例如下：

```
SQL> exec dbms_repair.skip_corrupt_blocks('SCOTT','EMP')
```

## 7. SEGMENT\_FIX\_STATUS

该过程用于修复位图入口的损坏，语法如下：

```
DBMS_REPAIR.SEGMENT_FIX_STATUS (
    segment_owner IN VARCHAR2, segment_name IN VARCHAR2,
    segment_type IN BINARY_INTEGER DEFAULT TABLE_OBJECT,
    file_number IN BINARY_INTEGER DEFAULT NULL,
    block_number IN BINARY_INTEGER DEFAULT NULL,
    status_value IN BINARY_INTEGER DEFAULT NULL,
    partition_name IN VARCHAR2 DEFAULT NULL);
```

如上所示，`segment_owner` 用于指定段所有者；`segment_name` 用于指定段名；`segment_type` 用于指定段类型；`file_number` 用于指定数据块所在的相对文件号；`block_number` 用于指定数据块号；`status_value` 用于指定块状态值（1: 全块；2: 0~25%；3: 25%~50%；4: 50%~75%；5: 75%~100%）；`partition_name` 用于指定分区名。示例如下：

```
SQL> execute dbms_repair.segment_fix_status('SYS', 'MYTAB')
```

## 17.19 DBMS\_RESOURCE\_MANAGER

包 `DBMS_RESOURCE_MANAGER` 用于维护资源计划、资源使用组和资源计划指令；包 `DBMS_RESOURCE_MANAGER_PRIVS` 用于维护与资源管理相关的权限，下面介绍它们所包

含的过程和函数。

### 1. DBMS\_RESOURCE\_MANAGER.CREATE\_PLAN

该过程用于建立资源计划，语法如下：

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN (
    plan IN VARCHAR2, comment IN VARCHAR2,
    cpu_mth IN VARCHAR2 DEFAULT 'EMPHASIS',
    active_sess_pool_mth IN VARCHAR2
        DEFAULT 'ACTIVE_SESS_POOL_ABSOLUTE',
    parallel_degree_limit_mth IN VARCHAR2
        DEFAULT 'PARALLEL_DEGREE_LIMIT_ABSOLUTE',
    queueing_mth IN VARCHAR2 DEFAULT 'FIFO_TIMEOUT');
```

如上所示，plan 用于指定资源计划名；comment 用于指定用户注释信息；cpu\_mth 用于指定 CPU 资源的分配方法；active\_sess\_pool\_mth 用于指定最大活动会话的分配方法；parallel\_degree\_limit\_mth 用于指定并行度的分配方法；queueing\_mth 用于指定活动会话池的队列策略类型。

### 2. DBMS\_RESOURCE\_MANAGER.CREATE\_SIMPLE\_PLAN

该过程用于建立简单资源计划，该资源计划最多包含 8 个资源使用组。语法如下：

```
DBMS_RESOURCE_MANAGER.CREATE_SIMPLE_PLAN (
    SIMPLE_PLAN IN VARCHAR2 DEFAULT,
    CONSUMER_GROUP1 IN VARCHAR2 DEFAULT,
    GROUP1_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP2 IN VARCHAR2 DEFAULT,
    GROUP2_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP3 IN VARCHAR2 DEFAULT,
    GROUP3_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP4 IN VARCHAR2 DEFAULT,
    GROUP4_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP5 IN VARCHAR2 DEFAULT,
    GROUP5_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP6 IN VARCHAR2 DEFAULT,
    GROUP6_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP7 IN VARCHAR2 DEFAULT,
    GROUP7_CPU IN NUMBER DEFAULT,
    CONSUMER_GROUP8 IN VARCHAR2 DEFAULT,
    GROUP8_CPU IN NUMBER DEFAULT);
```

### 3. DBMS\_RESOURCE\_MANAGER.UPDATE\_PLAN

该过程用于更新资源计划的定义，语法如下：

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN (
    plan IN VARCHAR2, new_comment IN VARCHAR2 DEFAULT NULL,
    new_cpu_mth IN VARCHAR2 DEFAULT NULL,
    new_active_sess_pool_mth IN VARCHAR2 DEFAULT NULL,
    new_parallel_degree_limit_mth IN VARCHAR2 DEFAULT NULL,
    new_queueing_mth IN VARCHAR2 DEFAULT NULL);
```

如上所示，new\_comment 用于指定用户的新的注释信息；new\_cpu\_mth 用于指定 CPU 资

源的新的分配方法; new\_active\_sess\_pool\_mth 用于指定最大活动会话的新的分配方法; new\_parallel\_degree\_limit\_mth 用于指定并行度的新的分配方法; new\_queueing\_mth 用于指定活动会话池的新的队列策略类型。

#### 4. DBMS\_RESOURCE\_MANAGER.DELETE\_PLAN

该过程用于删除资源计划, 语法如下:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN (plan IN VARCHAR2);
```

#### 5. DBMS\_RESOURCE\_MANAGER.DELETE\_PLAN CASCADE

该过程用于删除资源计划及其所有后代(资源计划指令、子计划和资源使用组), 语法如下:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_CASCADE (plan IN VARCHAR2);
```

#### 6. DBMS\_RESOURCE\_MANAGER.CREATE\_CONSUMER\_GROUP

该过程用于建立资源使用组, 语法如下:

```
DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP (
    consumer_group IN VARCHAR2, comment IN VARCHAR2,
    cpu_mth IN VARCHAR2 DEFAULT 'ROUND-ROBIN');
```

如上所示, consumer\_group 用于指定资源使用组名。

#### 7. DBMS\_RESOURCE\_MANAGER.UPDATE\_CONSUMER\_GROUP

该过程用于更新资源使用组信息, 语法如下:

```
DBMS_RESOURCE_MANAGER.UPDATE_CONSUMER_GROUP (
    consumer_group IN VARCHAR2,
    new_comment IN VARCHAR2 DEFAULT NULL,
    new_cpu_mth IN VARCHAR2 DEFAULT NULL);
```

#### 8. DBMS\_RESOURCE\_MANAGER.DELETE\_CONSUMER\_GROUP

该过程用于删除资源使用组, 语法如下:

```
DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP (
    consumer_group IN VARCHAR2);
```

#### 9. DBMS\_RESOURCE\_MANAGER.CREATE\_PLAN\_DIRECTIVE

该过程用于建立资源计划指令, 语法如下:

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
    plan IN VARCHAR2, group_or_subplan IN VARCHAR2,
    comment IN VARCHAR2, cpu_p1 IN NUMBER DEFAULT NULL,
    cpu_p2 IN NUMBER DEFAULT NULL, cpu_p3 IN NUMBER DEFAULT NULL,
    cpu_p4 IN NUMBER DEFAULT NULL, cpu_p5 IN NUMBER DEFAULT NULL,
    cpu_p6 IN NUMBER DEFAULT NULL, cpu_p7 IN NUMBER DEFAULT NULL,
    cpu_p8 IN NUMBER DEFAULT NULL,
    active_sess_pool_p1 IN NUMBER DEFAULT UNLIMITED,
    queueing_p1 IN NUMBER DEFAULT UNLIMITED,
    switch_group IN VARCHAR2 DEFAULT NULL,
    switch_time IN NUMBER DEFAULT UNLIMITED,
    switch_estimate IN BOOLEAN DEFAULT FALSE,
    max_est_exec_time IN NUMBER DEFAULT UNLIMITED,
    undo_pool IN NUMBER DEFAULT UNLIMITED,
    parallel_degree_limit_p1 IN NUMBER DEFAULT UNLIMITED);
```

如上所示，group\_or\_subplan 用于指定资源使用组或者子计划的名称；cpu\_p1 用于指定 CPU 资源分配方法的第一个参数；cpu\_p2 用于指定 CPU 资源分配方法的第二个参数；cpu\_p3 用于指定 CPU 资源分配方法的第三个参数；cpu\_p4 用于指定 CPU 资源分配方法的第四个参数；cpu\_p5 用于指定 CPU 资源分配方法的第五个参数；cpu\_p6 用于指定 CPU 资源分配方法的第六个参数；cpu\_p7 用于指定 CPU 资源分配方法的第七个参数；cpu\_p8 用于指定 CPU 资源分配方法的第八个参数；active\_sess\_pool\_p1 用于指定最大活动会话分配方法的第一个参数；queueing\_p1 用于指定队列超时时间；switch\_group 用于指定到达切换时间时要切换到的资源使用组；switch\_time 用于指定切换时间；当设置 switch\_estimate 为 true 时，通知 Oracle 使用执行时间估计自动切换资源使用组，默认值为 FALSE；undo\_pool 用于指定资源使用组的 UNDO 池尺寸；parallel\_degree\_limit\_p1 用于指定并行度分配方法的第一个参数。

## 10. DBMS\_RESOURCE\_MANAGER.UPDATE\_PLAN\_DIRECTIVE

该过程用于更新资源计划指令，语法如下：

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE (
    plan IN VARCHAR2, group_or_subplan IN VARCHAR2,
    new_comment IN VARCHAR2 DEFAULT NULL,
    new_cpu_p1 IN NUMBER DEFAULT NULL,
    new_cpu_p2 IN NUMBER DEFAULT NULL,
    new_cpu_p3 IN NUMBER DEFAULT NULL,
    new_cpu_p4 IN NUMBER DEFAULT NULL,
    new_cpu_p5 IN NUMBER DEFAULT NULL,
    new_cpu_p6 IN NUMBER DEFAULT NULL,
    new_cpu_p7 IN NUMBER DEFAULT NULL,
    new_cpu_p8 IN NUMBER DEFAULT NULL,
    new_active_sess_pool_p1 IN NUMBER DEFAULT NULL,
    new_queueing_p1 IN NUMBER DEFAULT NULL,
    new_parallel_degree_limit_p1 IN NUMBER DEFAULT NULL,
    new_switch_group IN VARCHAR2 DEFAULT NULL,
    new_switch_time IN NUMBER DEFAULT NULL,
    new_switch_estimate IN BOOLEAN DEFAULT FALSE,
    new_max_est_exec_time IN NUMBER DEFAULT NULL,
    new_undo_pool IN NUMBER DEFAULT UNLIMITED);
```

如上所示，new\_cpu\_p1 用于指定 CPU 资源分配方法的第一个参数；new\_cpu\_p2 用于指定 CPU 资源分配方法的第二个参数；new\_cpu\_p3 用于指定 CPU 资源分配方法的第三个参数；new\_cpu\_p4 用于指定 CPU 资源分配方法的第四个参数；new\_cpu\_p5 用于指定 CPU 资源分配方法的第五个参数；new\_cpu\_p6 用于指定 CPU 资源分配方法的第六个参数；new\_cpu\_p7 用于指定 CPU 资源分配方法的第七个参数；new\_cpu\_p8 用于指定 CPU 资源分配方法的第八个参数；new\_active\_sess\_pool\_p1 用于指定最大活动会话分配方法的第一个参数；new\_queueing\_p1 用于指定队列超时时间；new\_switch\_group 用于指定到达切换时间时要切换到的资源使用组；new\_switch\_time 用于指定切换时间；当设置 new\_switch\_estimate 为 true 时，通知 Oracle 使用执行时间估计自动切换资源使用组，默认值为 FALSE；new\_undo\_pool 用于指定资源使用组的 UNDO 池尺寸；new\_parallel\_degree\_limit\_p1 用于指定并行度分配方法的第一个参数。

**11. DBMS\_RESOURCE\_MANAGER.DELETE\_PLAN\_DIRECTIVE**

该过程用于删除资源计划指令，语法如下：

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE (
    plan IN VARCHAR2,group_or_subplan IN VARCHAR2);
```

**12. DBMS\_RESOURCE\_MANAGER.CREATE\_PENDING\_AREA**

该过程用于建立 pending 内存区，并且该内存区将用于改变资源管理对象。语法如下：

```
DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA;
```

**13. DBMS\_RESOURCE\_MANAGER.VALIDATE\_PENDING\_AREA**

该过程用于校验资源管理器的改变，语法如下：

```
DBMS_RESOURCE_MANAGER.VALIDATE_PENDING_AREA;
```

**14. DBMS\_RESOURCE\_MANAGER.CLEAR\_PENDING\_AREA**

该过程用于清除资源管理器的改变，语法如下：

```
DBMS_RESOURCE_MANAGER.CLEAR_PENDING_AREA;
```

**15. DBMS\_RESOURCE\_MANAGER.SUBMIT\_PENDING\_AREA**

该过程用于提交资源管理器的改变，语法如下：

```
DBMS_RESOURCE_MANAGER.SUBMIT_PENDING_AREA;
```

**16. DBMS\_RESOURCE\_MANAGER.SET\_INITIAL\_CONSUMER\_GROUP**

该过程用于指定用户的初始资源使用组，语法如下：

```
DBMS_RESOURCE_MANAGER.SET_INITIAL_CONSUMER_GROUP (
    user IN VARCHAR2,consumer_group IN VARCHAR2);
```

如上所示，user 用于指定用户名；consumer\_group 用于指定用户的初始资源使用组名。

**17. DBMS\_RESOURCE\_MANAGER.SWITCH\_CONSUMER\_GROUP\_FOR\_SESS**

该过程用于改变特定会话的资源使用组，语法如下：

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_SESS (
    session_id IN NUMBER,session_serial IN NUMBER,
    consumer_group IN VARCHAR2);
```

如上所示，session\_id 用于指定会话 ID 号；session\_serial 用于指定会话序列号。

**18. DBMS\_RESOURCE\_MANAGER.SWITCH\_CONSUMER\_GROUP\_FOR\_USER**

该过程用于改变特定用户所有会话的资源使用组，语法如下：

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_USER (
    user IN VARCHAR2,consumer_group IN VARCHAR2);
```

**19. DBMS\_RESOURCE\_MANAGER\_PRIVS.GRANT\_SYSTEM\_PRIVILEGE**

该过程用于将资源管理权限授予用户或角色，语法如下：

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SYSTEM_PRIVILEGE (
    grantee_name IN VARCHAR2,
    privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER',
    admin_option IN BOOLEAN);
```

如上所示，grantee\_name 用于指定被授权的用户或角色；privilege\_name 用于指定要授予的资源管理权限；admin\_option 用于指定是否可以转授资源管理权限（TRUE：转授权限；FALSE：不能转授权限）。示例如下：

```
SQL> conn system/manager
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SYSTEM_PRIVILEGE-
```

```
> ('SCOTT', 'ADMINISTER_RESOURCE_MANAGER', TRUE)
```

如上所示，在执行了以上命令之后，就会将资源管理权限授予 SCOTT 用户，该用户可以将该权限再授予其他用户。

## 20. DBMS\_RESOURCE\_MANAGER\_PRIVS.REVOKE\_SYSTEM\_PRIVILEGE

该过程用于收回资源管理权限，语法如下：

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SYSTEM_PRIVILEGE (
    revokee_name IN VARCHAR2,
    privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER');
```

如上所示，revokee\_name 用于指定被收回权限的用户或角色；privilege\_name 用于指定要收回的资源管理权限。示例如下：

```
SQL> conn system/manager
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SYSTEM_PRIVILEGE-
> ('SCOTT', 'ADMINISTER_RESOURCE_MANAGER')
```

如上所示，在执行了以上命令之后，用户 SCOTT 所具有的资源管理权限被收回。

## 21. DBMS\_RESOURCE\_MANAGER\_PRIVS.GRANT\_SWITCH\_CONSUMER\_GROUP

该过程用于将用户或角色分配给特定的资源使用组，语法如下：

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP (
    grantee_name IN VARCHAR2, consumer_group IN VARCHAR2,
    grant_option IN BOOLEAN);
```

如上所示，grant\_option 用于指定资源使用组转授选项。示例如下：

```
SQL> conn system/manager
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP-
> ('SCOTT', 'SYS_GROUP', TRUE)
```

如上所示，在执行了以上命令之后，就会将 SCOTT 用户分配给资源使用组 SYS\_GROUP，并且该用户可以再将其他用户分配给该资源使用组。

## 22. DBMS\_RESOURCE\_MANAGER\_PRIVS.REVOKE\_SWITCH\_CONSUMER\_GROUP

该过程用于收回分配给用户或角色的资源使用组，语法如下：

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SWITCH_CONSUMER_GROUP (
    revokee_name IN VARCHAR2, consumer_group IN VARCHAR2);
```

示例如下：

```
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SWITCH_CONSUMER_GROUP-
> ('SCOTT', 'SYS_GROUP')
```

如上所示，在执行了以上命令之后，SCOTT 用户使用资源使用组 SYS\_GROUP 的权限被收回。

## 23. 资源管理使用示例

默认情况下，只有特权用户 SYS，DBA 用户 SYSTEM 可以进行资源管理。为了使其他用户也可以进行资源管理，必须授予他们相应的资源管理权限。在建立与资源管理相关的对象时，必须要在 PENDING 内存区中建立资源对象，并且需要校验资源对象的正确性，最终提交 PENDING 内存区。下面通过示例说明使用资源管理的方法。

### (1) 为用户授予资源管理权限

默认情况下，只有特权用户 SYS，DBA 用户 SYSTEM 可以进行资源管理。为了使普通用

户可以进行资源管理，必须要为其授予相应的权限。下面以将资源管理权限授予 SCOTT 为例，说明为用户授予资源管理权限的方法。示例如下：

```
SQL> conn system/manager@test
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SYSTEM_PRIVILEGE-
> ('SCOTT', 'ADMINISTER_RESOURCE_MANAGER', FALSE)
```

## (2) 建立各种资源对象

当用户具有资源管理权限时，就可以建立资源使用组、资源计划和资源计划指令。但是在建立资源对象之前，必须首先分配 PENDING 内存区；在建立资源对象完成之后，必须检查并提交 PENDING 内存区。示例如下：

- 建立 PENDING 内存区：为了临时存放各种资源对象，必须首先建立 PENDING 内存区。示例如下：

```
SQL> conn SCOTT/TIGER@TEST
已连接。
SQL> EXEC DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA
```

- 建立资源使用组：在定义资源使用组时，必须提供资源使用组名称和相应注释信息。下面以建立资源使用组 OLTP 和 DSS 为例，说明建立资源使用组的方法。示例如下：

```
SQL> EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP-
> ('OLTP', '联机事务处理组')
SQL> EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP-
> ('DSS', '决策支持组')
```

- 建立资源计划：在建立资源计划时，必须提供资源计划名称和相应注释信息。下面以建立资源计划 DAY 和 NIGHT 为例，说明建立资源计划的方法。示例如下：

```
SQL> EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN-
> ('DAY', '该资源计划用于联机事务处理')
SQL> EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN-
> ('NIGHT', '该资源计划用于决策支持')
```

- 建立资源计划指令：通过建立资源计划指令，可以指定资源使用组和资源计划的关联关系。注意，当建立资源计划指令时，必须要在资源计划和 OTHER\_GROUPS 组之间定义关联关系。下面以在资源使用组 SYS\_GROUP, OLTP, OTHER\_GROUPS 和资源计划 DAY 之间建立关联，在资源使用组 SYS\_GROUP, DSS, OTHER\_GROUPS 和资源计划 NIGHT 之间建立关联为例，说明建立资源计划指令的方法。示例如下：

```
BEGIN
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'DAY', group_or_subplan=>'SYS_GROUP',
        comment=>'最高级别组', cpu_p1=>100,
        parallel_degree_limit_p1=>3);
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'DAY', group_or_subplan=>'OLTP',
        comment=>'中间级别组', cpu_p2=>80,
        parallel_degree_limit_p1=>1);
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'DAY', group_or_subplan=>'OTHER_GROUPS',
        comment=>'最低级别组', cpu_p3=>80,
```

```

    parallel_degree_limit_p1=>1);
END;
/
PL/SQL 过程已成功完成。
BEGIN
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'NIGHT',group_or_subplan=>'SYS_GROUP',
        comment=>'最高级别组',cpu_p1=>100,
        parallel_degree_limit_p1=>20);
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'NIGHT',group_or_subplan=>'DSS',
        comment=>'中间级别组',cpu_p2=>80,
        parallel_degree_limit_p1=>20);
    DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
        plan=>'NIGHT',group_or_subplan=>'OTHER_GROUPS',
        comment=>'最低级别组',cpu_p3=>80,
        parallel_degree_limit_p1=>20);
END;
/
PL/SQL 过程已成功完成

```

- 验证 PENDING 内存区：在建立了各种相关的资源对象之后，必须首先验证 PENDING 内存区。如果验证通过，那么可以提交 PENDING 内存区，并建立资源对象；如果验证未能通过，需要清除 PENDING 内存区，重新建立资源对象。示例如下：  
 SQL> exec DBMS\_RESOURCE\_MANAGER.VALIDATE\_PENDING\_AREA  
 PL/SQL 过程已成功完成。
- 提交 PENDING 内存区：在 PENDING 内存区验证成功之后，可以提交 PENDING 内存区，最终建立永久的资源管理对象。示例如下：  
 SQL> exec DBMS\_RESOURCE\_MANAGER.SUBMIT\_PENDING\_AREA  
 PL/SQL 过程已成功完成。

### (3) 分配用户到资源使用组

为了使得用户可以使用特定资源使用组的相关资源，需要分配用户到特定资源使用组。

下面以将 SCOTT 用户分配到资源使用组 OLTP 和 DSS 为例，说明分配用户到资源使用组的方法。示例如下：

```

SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.-
> GRANT_SWITCH_CONSUMER_GROUP('SCOTT','OLTP',FALSE)
SQL> exec DBMS_RESOURCE_MANAGER_PRIVS.-
> GRANT_SWITCH_CONSUMER_GROUP('SCOTT','DSS',FALSE)

```

### (4) 设置用户的默认资源使用组

数据库用户可以属于多个资源使用组，但在特定会话特定时刻只能使用某个资源使用组的相应资源。通过设置用户的默认资源使用组，可以使得在用户登录时自动使用相应资源使用组的资源。下面以将 SCOTT 用户的默认资源使用组设置为 OLTP 为例，说明设置用户默认资源使用组的方法。示例如下：

```

SQL> exec DBMS_RESOURCE_MANAGER.SET_INITIAL_CONSUMER_GROUP-
> ('SCOTT','OLTP')

```

### (5) 激活资源计划

为了通过数据库资源管理器限制数据库用户的资源使用，必须要激活资源计划。下面以在日间激活资源计划 DAY 为例，说明激活资源计划的方法。示例如下：

```
SQL> ALTER SYSTEM SET RESOURCE_MANAGER_PLAN=DAY
2 SCOPE=MEMORY;
```

### (6) 改变会话或用户的资源使用组

如果数据库用户属于多个资源使用组，那么在初始登录时会使用默认资源使用组，为了改变特定会话的资源使用组，可以执行以下语句：

```
SQL> exec DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_SESS-
> (7,8,'DSS')
```

为了改变特定用户所有会话的资源使用组，可以执行以下语句：

```
SQL> exec DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_USER-
> ('SCOTT','DSS')
```

## 17.20 DBMS\_STATS

包 DBMS\_STATS 用于搜集、查看、修改数据库对象的优化统计信息，下面介绍该包所包含的过程和函数。

### I. GET\_COLUMN\_STATS

该过程用于取得列的统计信息，语法如下：

```
DBMS_STATS.GET_COLUMN_STATS (
    ownname VARCHAR2, tablename VARCHAR2, colname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    distcnt OUT NUMBER, density OUT NUMBER,
    nullcnt OUT NUMBER, srec OUT StatRec,
    avgclen OUT NUMBER, statown VARCHAR2 DEFAULT NULL);
```

如上所示，`ownname` 用于指定方案名；`tablename` 用于指定表名；`colname` 用于指定列名；`partname` 用于指定分区名；`stattab` 用于指定用户统计表名；`statid` 用于指定与统计相关的标识符；`distcnt` 用于返回不同值的个数；`density` 用于返回列的密度；`nullcnt` 用于返回列的 NULL 个数；`srec` 用于返回列的最大、最小和直方图值；`avgclen` 用于返回列的平均长度；`statown` 用于指定包含 STATTAB 的方案名。示例如下：

```
DECLARE
    dist_count NUMBER;
    density    NUMBER;
    null_count NUMBER;
    srec DBMS_STATS.STATREC;
    avg_col_len NUMBER;
BEGIN
    dbms_stats.get_column_stats('SCOTT','EMP','JOB',
        distcnt=>dist_count,density=>density,
        nullcnt=>null_count,srec=>srec,
```

```

    avgclen=>avg_col_len);
    dbms_output.put_line('不同列值个数: '||dist_count);
    dbms_output.put_line('列平均长度: '||avg_col_len);
END;
/
不同列值个数: 5
列平均长度: 7

```

## 2. GET\_INDEX\_STATS

该过程用于取得索引的统计信息，语法如下：

```

DBMS_STATS.GET_INDEX_STATS (
    ownname VARCHAR2, indname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    numrows OUT NUMBER, numlblks OUT NUMBER, numdist OUT NUMBER,
    avgblk OUT NUMBER, avgdblk OUT NUMBER, clstfct OUT NUMBER,
    indlevel OUT NUMBER, statown VARCHAR2 DEFAULT NULL);

```

如上所示，ownname 用于指定索引所有者名；indname 用于指定索引名；partname 用于指定索引分区名；stattab 用于指定统计表名；statid 用于指定统计表相关的标识符；numrows 用于返回索引行数；numlblks 用于返回索引块个数；numdist 用于返回索引不同键值个数；avgblk 用于返回每个键值占用的平均叶块个数；avgdblk 用于返回每个键值对应表行所占用的平均数据块个数；clstfct 用于返回索引的聚簇因子；indlevel 用于返回索引层数；statown 用于指定统计表所有者。示例如下：

```

DECLARE
    numrows NUMBER;
    numlblks NUMBER;
    numdist NUMBER;
    avgblk NUMBER;
    avgdblk NUMBER;
    clstfct NUMBER;
    indlevel NUMBER;
BEGIN
    dbms_stats.get_index_stats('SCOTT', 'PK_EMP',
        numrows=>numrows, numlblks=>numlblks,
        numdist=>numdist, avgblk=>avgblk,
        avgdblk=>avgdblk, clstfct=>clstfct,
        indlevel=>indlevel);
    dbms_output.put_line('叶块个数:'||numlblks);
    dbms_output.put_line('索引层次:'||indlevel);
END;
/
叶块个数:1
索引层次:0

```

## 3. GET\_SYSTEM\_STATS

该过程用于从统计表或数据字典中取得系统统计信息，语法如下：

```
DBMS_STATS.GET_SYSTEM_STATS (
    status OUT VARCHAR2, dstart OUT DATE, dstop OUT DATE,
    pname VARCHAR2, pvalue OUT NUMBER,
    stattab IN VARCHAR2 DEFAULT NULL,
    statid IN VARCHAR2 DEFAULT NULL,
    statown IN VARCHAR2 DEFAULT NULL);
```

如上所示，status 用于返回状态信息（COMPLETED，AUTOGATHERING，MANUALGATHERING）；dstat 用于返回起始搜集日期；dstop 用于返回结束搜集日期；pname 用于指定要取得的参数名（sreadtim，mreadtim，cpuspeed，mbrc，maxthr，slavethr）；pvalue 用于返回参数值。

#### 4. GET\_TABLE\_STATS

该过程用于取得表的统计信息，语法如下：

```
DBMS_STATS.GET_TABLE_STATS (
    ownname VARCHAR2, tabname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    numrows OUT NUMBER, numblkls OUT NUMBER, avgrlen OUT NUMBER,
    statown VARCHAR2 DEFAULT NULL);
```

如上所示，avgrlen 用于返回表行的平均长度；示例如下：

```
DECLARE
    numrows NUMBER;
    numblkls NUMBER;
    avgrlen NUMBER;
BEGIN
    dbms_stats.get_table_stats('SCOTT', 'EMP',
        numrows=>numrows, numblkls=>numblkls,
        avgrlen=>avgrlen);
    dbms_output.put_line('表的总计行数:' || numrows);
    dbms_output.put_line('表所占用的块个数:' || numblkls);
    dbms_output.put_line('表行的平均长度:' || avgrlen);
END;
/
表的总计行数:14
表所占用的块个数:1
表行的平均长度:40
```

#### 5. DELETE\_COLUMN\_STATS

该过程用于删除列的统计信息，语法如下：

```
DBMS_STATS.DELETE_COLUMN_STATS (
    ownname VARCHAR2, tabname VARCHAR2, colname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    cascade_parts BOOLEAN DEFAULT TRUE,
    statown VARCHAR2 DEFAULT NULL,
```

```
    no_invalidate BOOLEAN DEFAULT FALSE);
```

如上所示，`cascade_parts` 用于指定是否要级联删除分区统计；`no_invalidate` 用于指定是否要使相关游标无效。示例如下：

```
exec dbms_stats.delete_column_stats('SCOTT','EMP','ENAME')
```

## 6. DELETE\_INDEX\_STATS

该过程用于删除索引统计信息，语法如下：

```
DBMS_STATS.DELETE_INDEX_STATS (
    ownname VARCHAR2, indname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL, stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    cascade_parts BOOLEAN DEFAULT TRUE,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
exec dbms_stats.delete_index_stats('SCOTT','PK_EMP')
```

## 7. DELETE\_SYSTEM\_STATS

该过程用于删除系统统计信息，语法如下：

```
DBMS_STATS.DELETE_SYSTEM_STATS (
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

## 8. DELETE\_TABLE\_STATS

该过程用于删除表的统计信息，语法如下：

```
DBMS_STATS.DELETE_TABLE_STATS (
    ownname VARCHAR2, tabname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    cascade_parts BOOLEAN DEFAULT TRUE,
    cascade_columns BOOLEAN DEFAULT TRUE,
    cascade_indexes BOOLEAN DEFAULT TRUE,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

如上所示，`cascade_columns` 用于指定是否级联删除列统计；`cascade_indexes` 用于指定是否级联删除索引统计；示例如下：

```
exec dbms_stats.delete_table_stats('SCOTT','EMP')
```

## 9. DELETE\_SCHEMA\_STATS

该过程用于删除特定方案的统计信息，语法如下：

```
DBMS_STATS.DELETE_SCHEMA_STATS (
    ownname VARCHAR2, stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL, statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
exec dbms_stats.delete_schema_stats('SCOTT')
```

## 10. DELETE\_DATABASE\_STATS

该过程用于删除整个数据库的统计信息，语法如下：

```
DBMS_STATS.DELETE_DATABASE_STATS (
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
exec dbms_stats.delete_database_stats
```

## 11. CREATE\_STAT\_TABLE

该过程用于在特定方案中建立统计表，语法如下：

```
DBMS_STATS.CREATE_STAT_TABLE (
    ownname VARCHAR2,stattab VARCHAR2,
    tblspace VARCHAR2 DEFAULT NULL);
```

如上所示，tblspace 用于指定统计表所在表空间。示例如下：

```
exec dbms_stats.create_stat_table('scott','stattab')
```

## 12. DROP\_STAT\_TABLE

该过程用于删除特定方案的统计表，语法如下：

```
DBMS_STATS.DROP_STAT_TABLE (
    ownname VARCHAR2,stattab VARCHAR2);
```

示例如下：

```
exec dbms_stats.drop_stat_table('scott','stattab')
```

## 13. EXPORT\_COLUMN\_STATS

该过程用于导出列统计并存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_COLUMN_STATS (
    ownname VARCHAR2,tablename VARCHAR2,colname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

示例如下：

```
SQL> exec dbms_stats.export_column_stats-
> ('scott','emp','ename',stattab=>'stattab')
```

## 14. EXPORT\_INDEX\_STATS

该过程用于导出索引统计信息，并存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_INDEX_STATS (
    ownname VARCHAR2,indname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

示例如下：

```
SQL> exec dbms_stats.export_index_stats-
> ('scott','pk_emp',stattab=>'stattab')
```

### 15. EXPORT\_SYSTEM\_STATS

该过程用于导出系统统计信息，并存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_SYSTEM_STATS (
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

### 16. EXPORT\_TABLE\_STATS

该过程用于导出表的统计信息，并将其存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_TABLE_STATS (
    ownname VARCHAR2, tabname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL, stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL, cascade BOOLEAN DEFAULT TRUE,
    statown VARCHAR2 DEFAULT NULL);
```

示例如下：

```
SQL> exec dbms_stats.export_table_stats-
> ('scott','emp',stattab=>'stattab')
```

### 17. EXPORT\_SCHEMA\_STATS

该过程用于导出方案的统计信息，并将其存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_SCHEMA_STATS (
    ownname VARCHAR2, stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL, statown VARCHAR2 DEFAULT NULL);
```

示例如下：

```
SQL> exec dbms_stats.export_schema_stats-
> ('scott',stattab=>'stattab')
```

### 18. EXPORT\_DATABASE\_STATS

该过程用于导出数据库的所有统计信息，并存储到统计表中，语法如下：

```
DBMS_STATS.EXPORT_DATABASE_STATS (
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

示例如下：

```
SQL> exec dbms_stats.export_database_stats-
> (stattab=>'stattab',statown=>'SCOTT')
```

### 19. IMPORT\_COLUMN\_STATS

该过程用于从统计表中取得列统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_COLUMN_STATS (
    ownname VARCHAR2, tabname VARCHAR2, colname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL, stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
SQL> exec dbms_stats.import_column_stats-
> ('SCOTT','EMP','ENAME',stattab=>'STATTAB',statown=>'SCOTT')
```

## 20. IMPORT\_INDEX\_STATS

该过程用于从统计表中取得索引统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_INDEX_STATS (
    ownname VARCHAR2, indname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
SQL> exec dbms_stats.import_index_stats-
> ('SCOTT','PK_EMP',stattab=>'STATTAB',statown=>'SCOTT')
```

## 21. IMPORT\_SYSTEM\_STATS

该过程用于从统计表中取得系统统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_SYSTEM_STATS (
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL);
```

## 22. IMPORT\_TABLE\_STATS

该过程用于从统计表中取得表统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_TABLE_STATS (
    ownname VARCHAR2, tabname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    cascade BOOLEAN DEFAULT TRUE,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
SQL> exec dbms_stats.import_table_stats-
> ('SCOTT','EMP',stattab=>'STATTAB',statown=>'SCOTT')
```

## 23. IMPORT\_SCHEMA\_STATS

该过程用于从统计表中取得方案统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_SCHEMA_STATS (
    ownname VARCHAR2, stattab VARCHAR2,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
SQL> exec dbms_stats.import_schema_stats-
> ('SCOTT',stattab=>'STATTAB',statown=>'SCOTT')
```

## 24. IMPORT\_DATABASE\_STATS

该过程用于从统计表中取得数据库所有对象的统计，并将其存储到数据字典中。语法如下：

```
DBMS_STATS.IMPORT_DATABASE_STATS (
    stattab VARCHAR2, statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

示例如下：

```
SQL> exec dbms_stats.import_database_stats-
> (stattab=>'STATTAB',statown=>'SCOTT')
```

## 25. GATHER\_INDEX\_STATS

该过程用于搜集索引统计，语法如下：

```
DBMS_STATS.GATHER_INDEX_STATS (
    ownname VARCHAR2, indname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    estimate_percent NUMBER DEFAULT NULL,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    degree NUMBER DEFAULT NULL,
    granularity VARCHAR2 DEFAULT 'DEFAULT',
    no_invalidate BOOLEAN DEFAULT FALSE);
```

如上所示，`ownname` 用于指定方案名；`indname` 用于指定索引名；`partname` 用于指定分区名；`estimate_percent` 用于指定要预估的行百分比；`stattab` 用于指定用户统计表名；`statid` 用于指定与统计相关的标识符；`statown` 用于指定包含 STATTAB 的方案名；`degree` 用于指定并行度；`granularity` 用于指定要搜集索引的粒度（`DEFAULT`, `SUBPARTITION`, `PARTITION`, `GLOBAL`, `ALL`）；`no_invalidate` 用于指定是否要使相关游标无效。示例如下：

```
SQL> exec dbms_stats.gather_index_stats('SCOTT','PK_EMP')
```

## 26. GATHER\_TABLE\_STATS

该过程用于搜集表统计，语法如下：

```
DBMS_STATS.GATHER_TABLE_STATS (
    ownname VARCHAR2, tabname VARCHAR2,
    partname VARCHAR2 DEFAULT NULL,
    estimate_percent NUMBER DEFAULT NULL,
    block_sample BOOLEAN DEFAULT FALSE,
    method_opt VARCHAR2 DEFAULT 'FOR ALL COLUMNS SIZE 1',
    degree NUMBER DEFAULT NULL,
    granularity VARCHAR2 DEFAULT 'DEFAULT',
    cascade BOOLEAN DEFAULT FALSE,
    stattab VARCHAR2 DEFAULT NULL,
    statid VARCHAR2 DEFAULT NULL,
    statown VARCHAR2 DEFAULT NULL,
    no_invalidate BOOLEAN DEFAULT FALSE);
```

如上所示，`block_sample` 用于指定是否使用随机采样的块个数取代行百分比；`method_opt` 用于指定列统计的搜集方法；`cascade` 用于指定是否要级联搜集索引统计；示例如下：

```
SQL> exec dbms_stats.gather_table_stats('SCOTT','EMP')
```

## 27. GATHER\_SCHEMA\_STATS

该过程用于搜集特定方案所有对象的统计，语法如下：

```
DBMS_STATS.GATHER_SCHEMA_STATS (
    ownname VARCHAR2, estimate_percent NUMBER DEFAULT NULL,
    block_sample BOOLEAN DEFAULT FALSE,
```

```

method_opt VARCHAR2 DEFAULT 'FOR ALL COLUMNS SIZE 1',
degree NUMBER DEFAULT NULL,
granularity VARCHAR2 DEFAULT 'DEFAULT',
cascade BOOLEAN DEFAULT FALSE,
stattab VARCHAR2 DEFAULT NULL,statid VARCHAR2 DEFAULT NULL,
options VARCHAR2 DEFAULT 'GATHER',objlist OUT ObjectTab,
statown VARCHAR2 DEFAULT NULL,
no_invalidate BOOLEAN DEFAULT FALSE,
gather_temp BOOLEAN DEFAULT FALSE);

```

如上所示, options 用于指定统计搜集选项 (GATHER, GATHER AUTO, GATHER STALE, GATHER EMPTY, LIST AUTO, LIST STALE、LIST EMPTY); objlist 用于返回旧对象或空对象; gather\_temp 用于指定是否要搜集临时表统计。示例如下:

```
SQL> exec dbms_stats.gather_schema_stats('SCOTT')
```

## 28. GATHER\_DATABASE\_STATS

该过程用于搜集数据库所有对象的统计, 语法如下:

```

DBMS_STATS.GATHER_DATABASE_STATS (
estimate_percent NUMBER DEFAULT NULL,
block_sample BOOLEAN DEFAULT FALSE,
method_opt VARCHAR2 DEFAULT 'FOR ALL COLUMNS SIZE 1',
degree NUMBER DEFAULT NULL,
granularity VARCHAR2 DEFAULT 'DEFAULT',
cascade BOOLEAN DEFAULT FALSE,
stattab VARCHAR2 DEFAULT NULL,
statid VARCHAR2 DEFAULT NULL,
options VARCHAR2 DEFAULT 'GATHER',
objlist OUT ObjectTab,
statown VARCHAR2 DEFAULT NULL,
gather_sys BOOLEAN DEFAULT FALSE,
no_invalidate BOOLEAN DEFAULT FALSE,
gather_temp BOOLEAN DEFAULT FALSE);

```

如上所示, estimate\_percent 用于指定要预估的行百分比; block\_sample 用于指定是否使用随机采样的块个数取代行百分比; method\_opt 用于指定列统计的搜集方法; degree 用于指定并行度; granularity 用于指定表粒度 (DEFAULT, SUBPARTITION, PARTITION, GLOBAL, ALL); cascade 用于指定是否要级联搜集索引统计; stattab 用于指定用户统计表名; statid 用于指定与统计相关的标识符; options 用于指定统计搜集选项 (GATHER, GATHER AUTO, GATHER STALE, GATHER EMPTY, LIST AUTO, LIST STALE, LIST EMPTY); objlist 用于返回旧对象或空对象; statown 用于指定包含 STATTAB 的方案名; gather\_sys 用于指定是否要搜集 SYS 用户对象的统计; no\_invalidate 用于指定是否要使相关游标无效; gather\_temp 用于指定是否要搜集临时表统计。示例如下:

```
SQL> exec dbms_stats.gather_database_stats
```

## 29. GATHER\_SYSTEM\_STATS

该过程用于搜集系统统计, 语法如下:

```

DBMS_STATS.GATHER_SYSTEM_STATS (
gathering_mode VARCHAR2 DEFAULT 'NOWORKLOAD',

```

```
interval INTEGER DEFAULT NULL,stattab VARCHAR2 DEFAULT NULL,
statid VARCHAR2 DEFAULT NULL,statown VARCHAR2 DEFAULT NULL);
```

如上所示, `gathering_mode` 用于指定搜集模式值 (NOWORKLOAD, INTERVAL, START|STOP); `interval` 用于指定搜集统计的时间间隔 (只适用于 INTERVAL 模式); 示例如下:

```
SQL> exec dbms_stats.gather_system_stats
```

## 17.21 UTL\_FILE

包 `UTL_FILE` 用于读写 OS 文件。使用该包访问 OS 文件时, 必须要为 OS 目录建立相应的 DIRECTORY 对象。当用户要访问特定目录下的文件时, 必须要具有读写 DIRECTORY 对象的权限。在介绍如何使用 `UTL_FILE` 包之前, 应首先建立 DIRECTORY 对象, 并要为用户授权。示例如下:

```
SQL> conn system/manager
SQL> CREATE DIRECTORY user_dir AS 'G:\FILE';
SQL> GRANT READ,WRITE ON DIRECTORY user_dir TO scott;
```

下面介绍该包所提供的类型、过程和函数。

### 1. FILE\_TYPE

该类型是 `UTL_FILE` 包中所定义的记录类型, 其成员是私有的, 不能被直接引用。该类型的定义如下:

```
TYPE file_type IS RECORD (
    id BINARY_INTEGER,datatype BINARY_INTEGER
);
```

### 2. FOPEN

该函数用于打开 OS 文件。注意, 使用该函数最多可以同时打开 50 个文件。语法如下:

```
UTL_FILE.FOPEN (
    location IN VARCHAR2,filename IN VARCHAR2,
    open_mode IN VARCHAR2,max_linesize IN BINARY_INTEGER)
RETURN file_type;
```

当成功地执行了该函数之后, 就会返回文件句柄, 访问该文件可以直接使用文件句柄。如果执行失败, 则会触发例外或显示错误。注意, 当指定文件位置时, 必须要使用 DIRECTORY 对象, 并且其名称必须大写。示例如下:

```
DECLARE
    handle UTL_FILE.FILE_TYPE;
BEGIN
    handle:=utl_file.fopen('USER_DIR','readme.txt','r',1000);
    dbms_output.put_line('打开文件成功');
END;
/
打开文件成功
```

### 3. FOPEN\_NCHAR

该函数用于以 UNICODE 方式打开文件。当使用该函数打开文件之后, 读写文件会使用 UNICODE 取代数据库字符集。语法如下:

```
UTL_FILE.FOPEN_NCHAR(
    location IN VARCHAR2,
    filename IN VARCHAR2,
    open_mode IN VARCHAR2,
    max_linesize IN BINARY_INTEGER)
RETURN file_type;
```

#### 4. IS\_OPEN

该函数用于确定文件是否已经打开，语法如下：

```
UTL_FILE.IS_OPEN (file IN FILE_TYPE) RETURN BOOLEAN;
```

如上所示，file 用于指定文件句柄。如果文件已经被打开，则返回 TRUE，否则返回 FALSE。

使用该函数的示例如下：

```
DECLARE
    handle UTL_FILE.FILE_TYPE;
BEGIN
    IF NOT utl_file.is_open(handle) THEN
        handle:=utl_file.fopen('USER_DIR','&filename','r',1000);
    END IF;
    dbms_output.put_line('打开文件成功');
END;
/
输入 filename 的值： a.txt
打开文件成功
```

#### 5. FCLOSE

该过程用于关闭已经打开的文件。语法如下：

```
UTL_FILE.FCLOSE(file IN OUT file_type);
```

#### 6. FCLOSE\_ALL

该过程用于关闭当前会话打开的所有文件。语法如下：

```
UTL_FILE.FCLOSE_ALL;
```

#### 7. GET\_LINE

该过程用于从已打开文件中读取行内容，行内容会被读取到输出缓冲区。语法如下：

```
UTL_FILE.GET_LINE (
    file IN FILE_TYPE,buffer OUT VARCHAR2,
    linesize IN NUMBER,len IN PLS_INTEGER DEFAULT NULL);
```

如上所示，buffer 用于存储读取信息；linesize 用于指定要读取的最大字节数；len 用于指定实际读取的长度。使用该过程的示例如下：

```
DECLARE
    handle UTL_FILE.FILE_TYPE;
    buffer VARCHAR2(100);
BEGIN
    handle:=utl_file.fopen('USER_DIR','a.txt','r',1000);
    utl_file.get_line(handle,buffer,100);
    dbms_output.put_line(buffer);
    utl_filefclose(handle);
END;
```

马丽，1997 年毕业于哈尔滨工业大学计算机科学与工程系，同年分配到航天工业总公司第六研究院工作。

### 8. GET\_LINE\_NCHAR

该过程用于以 UNICODE 方式读取已打开文件的行内容，并且将行内容读取到输出缓冲区中。使用该过程的语法如下：

```
UTL_FILE.GET_LINE_NCHAR(
    file IN FILE_TYPE,buffer OUT VARCHAR2,
    len IN PLS_INTEGER DEFAULT NULL);
```

使用该过程的示例如下：

```
DECLARE
    handle UTL_FILE.FILE_TYPE;
    buffer VARCHAR2(1000);
BEGIN
    handle:=utl_file.fopen_nchar(
        'USER_DIR','a.txt','r',1000);
    utl_file.get_line_nchar(handle,buffer);
    dbms_output.put_line(buffer);
    utl_filefclose(handle);
END;
/
```

### 9. GET\_RAW

该过程用于从文件中读取 RAW 字符串，并调节文件指针到读取位置。语法如下：

```
UTL_FILE.GET_RAW (
    fid IN utl_file.file_type,r OUT NOCOPY RAW,
    len IN PLS_INTEGER DEFAULT NULL);
```

如上所示，fid 用于指定文件句柄；r 用于取得读取信息。使用该函数的示例如下：

```
DECLARE
    handle UTL_FILE.FILE_TYPE;
    buffer RAW(100);
BEGIN
    handle:=utl_file.fopen('USER_DIR','a.txt','r',1000);
    utl_file.get_raw(handle,buffer,100);
    dbms_output.put_line(buffer);
    utl_filefclose(handle);
END;
/
C2EDC0F6A3AC31393937C4EAB1CFD2B5D3DAB9FEB6FBB1F
```

### 10. PUT

该过程用于将缓冲区内容写入到文件中。当使用 PUT 过程时，文件必须要以写方式打开。在写入缓冲区内容之后，如果要结束行，那么可以使用 NEW\_LINE 过程。语法如下：

```
UTL_FILE.PUT (file IN FILE_TYPE,buffer IN VARCHAR2);
```

使用该过程的示例如下：

```
DECLARE
```

```

handle UTL_FILE.FILE_TYPE;
buffer VARCHAR2(100);
BEGIN
    handle:=utl_file.fopen('USER_DIR','b.txt','w',1000);
    buffer:='&content1';
    utl_file.put(handle,buffer);
    utl_file.new_line(handle);
    buffer:='&content2';
    utl_file.put_line(handle,buffer);
    utl_file.fclose(handle);
END;
/
输入 content1 的值： 中华民族具有 5000 年的历史
输入 content2 的值： 中国，中国，伟大的中国

```

## 11. PUT\_NCHAR

该过程用于将缓冲区内容以 UNICODE 方式写入到文件。语法如下：

```
UTL_FILE.PUT_NCHAR(file IN FILE_TYPE,buffer IN VARCHAR2);
```

使用该过程的示例如下：

```

DECLARE
    handle UTL_FILE.FILE_TYPE;
    buffer VARCHAR2(100);
BEGIN
    handle:=utl_file.open_nchar('USER_DIR','b.txt','w',1000);
    buffer:='&content1';
    utl_file.put_nchar(handle,buffer);
    utl_file.new_line(handle);
    buffer:='&content2';
    utl_file.put_line_nchar(handle,buffer);
    utl_file.close(handle);
END;
/
输入 content1 的值： 中华民族具有 5000 年的历史
输入 content2 的值： 中国，中国，伟大的中国

```

## 12. PUT\_RAW

该过程用于将 RAW 缓冲区中的数据写入到 OS 文件。语法如下：

```
UTL_FILE.PUT_RAW (
    fid IN utl_file.file_type,r IN RAW,
    autoflush IN BOOLEAN DEFAULT FALSE);
```

如上所示，fid 用于指定文件句柄，r 用于指定存放 RAW 数据的缓冲区，autoflush 用于指定是否要自动刷新缓冲区数据。使用该过程的示例如下：

```

DECLARE
    handle UTL_FILE.FILE_TYPE;
    buffer RAW(100);
BEGIN
    handle:=utl_file.fopen('USER_DIR','b.txt','w',1000);

```

```

    buffer:='&content';
    utl_file.put_raw(handle,buffer);
    utl_file.new_line(handle);
    utl_file.fclose(handle);
END;
/
输入 content 的值: 01f3d5a4c7d8

```

### 13. NEW\_LINE

该过程用于为文件增加行终止符。语法如下：

```
UTL_FILE.NEW_LINE (file IN FILE_TYPE,lines IN NATURAL := 1);
```

如上所示，`lines` 用于指定要增加的行终止符个数。

### 14. PUT\_LINE

该过程用于将文本缓冲区内容写入到文件中。当使用该过程为文件追加内容时，会自动在内容的尾部追加行终止符。语法如下：

```
UTL_FILE.PUT_LINE (
    file IN FILE_TYPE,buffer IN VARCHAR2,
    autoflush IN BOOLEAN DEFAULT FALSE);
```

### 15. PUT\_LINE\_NCHAR

该过程用于将文本缓冲区内容以 UNICODE 方式写入文件。当使用该过程为文件写入内容时，会自动在尾部追加行终止符。语法如下：

```
UTL_FILE.PUT_LINE_NCHAR(
    file IN FILE_TYPE,buffer IN VARCHAR2);
```

### 16. PUTF

该过程用于以特定格式将文本内容写入到 OS 文件，其中格式符`%s` 表示字符串，格式符`\n` 表示行终止符。语法如下：

```
UTL_FILE.PUTF (
    file IN FILE_TYPE,format IN VARCHAR2,
    [arg1 IN VARCHAR2 DEFAULT NULL,
     . . .
     arg5 IN VARCHAR2 DEFAULT NULL]);
```

如上所示，`format` 用于指定格式符（最多 5 个`%s`），`arg1`, ..., `arg5` 用于指定对应于格式符的字符串。使用该过程的示例如下：

```

DECLARE
    handle UTL_FILE.FILE_TYPE;
BEGIN
    handle:=utl_file.fopen('USER_DIR','b.txt','w',1000);
    utl_file.putf(handle,'%s\n%s\n%s\n',
                  '&line1','&line2','&line3');
    utl_file.fclose(handle);
END;
/
输入 line1 的值: 中国拥有 5000 年的历史
输入 line2 的值: 中国是四大文明古国
输入 line3 的值: 中华人民共和国万岁

```

### 17. PUTF\_NCHAR

该过程用于以特定格式将文本内容以 UNICODE 方式写入到 OS 文件中，其中格式符%*s* 表示字符串；格式符\n 表示行终止符。语法如下：

```
UTL_FILE.PUTF_NCHAR (
    file IN FILE_TYPE, format IN VARCHAR2,
    [arg1 IN VARCHAR2 DEFAULT NULL,
     .
     .
     .
    arg5 IN VARCHAR2 DEFAULT NULL]);
```

如上所示，*farg1*, ..., *arg5* 用于指定为%*s* 格式符所提供的字符串。

### 18. FFLUSH

该过程用于将数据强制性写入到 OS 文件。正常情况下，当给文件写入数据时，数据会被暂时存放在缓存中，过程 FFLUSH 用于强制将数据写入到 OS 文件。语法如下：

```
UTL_FILE.FFLUSH (file IN FILE_TYPE);
```

### 19. FSEEK

该过程用于移动文件指针到特定位置。当使用该过程移动文件指针时，既可以指定文件指针的绝对位置，也可以指定文件指针的相对位置。语法如下：

```
UTL_FILE.FSEEK (
    fid IN utl_file.file_type,
    absolute_offset IN PL_INTEGER DEFAULT NULL,
    relative_offset IN PLS_INTEGER DEFAULT NULL);
```

如上所示，*absolute\_offset* 用于指定文件指针的绝对位置（单位：字节）；*relative\_offset* 用于指定文件指针的相对位置（单位：字节）。使用该过程的示例如下：

```
DECLARE
    handle utl_file.file_type;
BEGIN
    handle:=utl_file.fopen('USER_DIR','a.txt','r');
    dbms_output.put_line('文件指针起始位置:'|| 
        utl_file.fgetpos(handle));
    utl_file.fseek(handle,20);
    dbms_output.put_line('文件指针当前位置:'|| 
        utl_file.fgetpos(handle));
    utl_filefclose(handle);
END;
/
文件指针起始位置:0
文件指针当前位置:20
```

### 20. FREMOVE

该过程用于删除磁盘文件。语法如下：

```
UTL_FILE.FREMOVE (location IN VARCHAR2,filename IN VARCHAR2);
```

如上所示，*location* 用于指定 DIRECTORY 对象（必须大写）；*filename* 用于指定要删除的 OS 文件名。示例如下：

```
SQL> exec utl_file.fremove('USER_DIR','b.txt');
```

## 21. FCOPY

该过程用于将源文件的全部或部分内容复制到目标文件中。当使用该过程时，如果不设置起始行和结束行，则将复制文件的所有内容。语法如下：

```
UTL_FILE.FCOPY (
    location IN VARCHAR2,
    filename IN VARCHAR2,
    dest_dir IN VARCHAR2,
    dest_file IN VARCHAR2,
    start_line IN PLS_INTEGER DEFAULT 1,
    end_line IN PLS_INTEGER DEFAULT NULL);
```

如上所示，location 用于指定源文件所在目录对应的 DIRECTORY 对象；filename 用于指定源文件名；dest\_dir 用于指定目标文件所在目录对应的 DIRECTORY 对象；dest\_file 用于指定目标文件的名称；start\_line 用于指定起始行号；end\_line 用于指定结束行号。使用 FCOPY 过程的示例如下：

```
SQL> exec utl_file.fcopy('USER_DIR','a.txt','USER_DIR','c.txt')
```

## 22. FGETPOS

该函数用于返回文件指针所在的偏移位置。语法如下：

```
UTL_FILE.FGETPOS (fileid IN file_type) RETURN PLS_INTEGER;
```

## 23. FGETATTR

该过程用于读取磁盘文件，并返回文件属性。语法如下：

```
UTL_FILE.FGETATTR(
    location IN VARCHAR2, filename IN VARCHAR2,
    exists OUT BOOLEAN, file_length OUT NUMBER,
    blocksize OUT NUMBER);
```

如上所示，location 用于指定 OS 目录所对应的 DIRECTORY 对象；filename 用于指定 OS 文件名；exists 用于确定文件是否存在；file\_length 用于取得文件长度；blocksize 用于取得 OS 块的尺寸。示例如下：

```
DECLARE
    fileexist BOOLEAN;
    filelen INT;
    os_block INT;
BEGIN
    utl_file.fgetattr('USER_DIR','readme.txt',fileexist,
        filelen,os_block);
    IF fileexist THEN
        dbms_output.put_line('文件尺寸:'||filelen);
        dbms_output.put_line('OS 块尺寸:'||os_block);
    END IF;
END;
/
文件尺寸:111616
OS 块尺寸:0
```

#### 24. FRENAME

该过程用于修改已存在的 OS 文件名，其作用与 UNIX 的 mv 命令完全相同。在修改文件名时，通过指定 `overwrite` 参数，可以覆盖已存在的文件。语法如下：

```
UTL_FILE.FRENAME (
    location IN VARCHAR2,
    filename IN VARCHAR2,
    dest_dir IN VARCHAR2,
    dest_file IN VARCHAR2,
    overwrite IN BOOLEAN DEFAULT FALSE);
```

如上所示，`overwrite` 用于指定是否要覆盖已存在文件（`FALSE`：不能覆盖，`TRUE`：覆盖）。使用该过程的示例如下：

```
exec utl_file.frename('USER_DIR','d.txt','USER_DIR','c.txt')
```

### 17.22 UTL\_INADDR

该包用于取得局域网或 Internet 环境中的主机名和 IP 地址，下面介绍该包所包含的过程和函数。

#### 1. GET\_HOST\_NAME

该函数用于取得指定 IP 地址所对应的主机名，语法如下：

```
UTL_INADDR.GET_HOST_NAME (ip IN VARCHAR2 DEFAULT NULL)
    RETURN VARCHAR2;
```

如上所示，`ip` 用于指定 TCP/IP 地址。使用该函数的示例如下：

```
SQL> select utl_inaddr.get_host_name('127.0.0.1') hostname
  2  FROM dual;
HOSTNAME
-----
wanghailiang
```

#### 2. GET\_HOST\_ADDRESS

该函数用于取得指定主机所对应的 IP 地址，语法如下：

```
UTL_INADDR.GET_HOST_ADDRESS (host IN VARCHAR2 DEFAULT NULL)
    RETURN VARCHAR2;
```

如上所示，`host` 用于指定主机名。使用该函数的示例如下：

```
SQL> select utl_inaddr.get_host_address('wanghailiang') ip
  2  FROM dual;
IP
-----
127.0.0.1
```

## 附录 A 习题参考答案

### 第 2 章

#### 1. 建立练习表并插入数据。

- 建立 CUSTOMER 表并插入数据

```
CREATE TABLE CUSTOMER (
    CUSTOMER_ID      NUMBER (6) CONSTRAINT pk_cust PRIMARY KEY,
    NAME             VARCHAR2 (45),
    ADDRESS          VARCHAR2 (40),
    CITY             VARCHAR2 (30),
    STATE            VARCHAR2 (2),
    ZIP_CODE         VARCHAR2 (9),
    AREA_CODE        NUMBER (3),
    PHONE_NUMBER     NUMBER (7));
INSERT INTO CUSTOMER VALUES (
    214 , 'BOB', '45 SPRUCE ST.',
    'SPRING', 'TX', '77388', '713', '5555172');
INSERT INTO CUSTOMER VALUES (
    215 , 'MARY', '400 E. 23RD',
    'HOUSTON', 'TX', '77026', '713', '5558015');
INSERT INTO CUSTOMER VALUES (
    216 , 'BLAKE', '547 PRENTICE RD.',
    'CHELSEA', 'MA', '02150', '617', '5553047');
INSERT INTO CUSTOMER VALUES (
    217 , 'CLARK', '333 WOOD COURT',
    'GRAPEVINE', 'TX', '76051', '817', '5552352');
INSERT INTO CUSTOMER VALUES (
    218 , 'SMITH', '346 GARDEN BLVD.',
    'FLUSHING', 'NY', '11355', '718', '5552131');
INSERT INTO CUSTOMER VALUES (
    221 , 'KING', '2 MEMORIAL DRIVE',
    'HOUSTON', 'TX', '77007', '713', '5554139');
INSERT INTO CUSTOMER VALUES (
    222 , 'STEVEN', '4000 PARKRIDGE BLVD.',
    'DALLAS', 'TX', '75205', '214', '5558735');
INSERT INTO CUSTOMER VALUES (
    223 , 'KLINTON', '23 WHITE ST.',
    'MALDEN', 'MA', '02148', '617', '5554983');
COMMIT;
```

- 建立 PRODUCT 表并插入数据

```

CREATE TABLE PRODUCT (
    PRODUCT_ID      NUMBER (6) CONSTRAINT pk_product PRIMARY KEY,
    DESCRIPTION     VARCHAR2 (30));
INSERT INTO PRODUCT VALUES ('100860', 'ACE TENNIS RACKET I');
INSERT INTO PRODUCT VALUES ('100861', 'ACE TENNIS RACKET II');
INSERT INTO PRODUCT VALUES ('100870', 'ACE TENNIS BALLS-3 PACK');
INSERT INTO PRODUCT VALUES ('100871', 'ACE TENNIS BALLS-6 PACK');
INSERT INTO PRODUCT VALUES ('100890', 'ACE TENNIS NET');
INSERT INTO PRODUCT VALUES ('101860', 'SP TENNIS RACKET');
COMMIT;

```

- 建立 ORD 表并插入数据

```

CREATE TABLE ORD (
    ORD_ID          NUMBER (4) CONSTRAINT pk_ord PRIMARY KEY,
    ORD_DATE        DATE,
    CUSTOMER_ID    NUMBER (6),
    SHIP_DATE       DATE,
    TOTAL           NUMBER (8,2),
CONSTRAINT fk_customer_id FOREIGN KEY(customer_id)
    REFERENCES customer(customer_id);
INSERT INTO ORD VALUES
    (610, TO_DATE(2448264, 'J'), 214, TO_DATE(2448265, 'J'), 101.4);
INSERT INTO ORD VALUES
    (611, TO_DATE(2448268, 'J'), 215, TO_DATE(2448268, 'J'), 45);
INSERT INTO ORD VALUES
    (612, TO_DATE(2448272, 'J'), 216, TO_DATE(2448277, 'J'), 5860);
INSERT INTO ORD VALUES
    (601, TO_DATE(2448013, 'J'), 217, TO_DATE(2448042, 'J'), 60.8);
INSERT INTO ORD VALUES
    (602, TO_DATE(2448048, 'J'), 218, TO_DATE(2448063, 'J'), 56);
INSERT INTO ORD VALUES
    (600, TO_DATE(2448013, 'J'), 221, TO_DATE(2448041, 'J'), 42);
COMMIT;

```

- 建立 ITEM 表并插入数据

```

CREATE TABLE ITEM (
    ORD_ID          NUMBER (4),
    ITEM_ID         NUMBER (4),
    PRODUCT_ID     NUMBER (6),
    ACTUAL_PRICE   NUMBER (8,2),
    QUANTITY        NUMBER (8),
    TOTAL           NUMBER (8,2),
CONSTRAINT pk_ord_item PRIMARY KEY(ord_id, item_id),
CONSTRAINT fk_ord_id FOREIGN KEY (ord_id) REFERENCES ord(ord_id),
CONSTRAINT fk_product_id FOREIGN KEY (product_id)
    REFERENCES product(product_id)
);

```

```

INSERT INTO ITEM VALUES (600,1,100861,42,1,42);
INSERT INTO ITEM VALUES (600,2,100890,58,1,58);
INSERT INTO ITEM VALUES (611,1,100861,45,1,45);
INSERT INTO ITEM VALUES (612,1,100860,30,100,3000);
INSERT INTO ITEM VALUES (601,1,101860,2.4,12,28.8);
INSERT INTO ITEM VALUES (601,2,100860,32,1,32);
INSERT INTO ITEM VALUES (602,1,100870,2.8,20,56);
INSERT INTO ITEM VALUES (610,1,100890,58,3,174);
INSERT INTO ITEM VALUES (610,2,100861,42,2,84);
INSERT INTO ITEM VALUES (610,3,100860,32,12,384);
COMMIT;

```

## 第 3 章

1. D
2. A、D、F
3. 请根据结果确定以下变量的数据类型
  - (1) 数字类型
  - (2) 任意类型
  - (3) 布尔类型
  - (4) 日期时间类型
  - (5) 字符串类型
4. 因为子块可以引用主块的变量，而主块不能引用子块的变量，所以子块和主块的变量值分别为
  - 子块：v1, v2, v3 的结果分别为 110, 210, 300
  - 主块：v1, v2 分别为 100, 210，而 v3 则不能引用
5.
 

```

SQL> set serveroutput on
SQL> BEGIN
 2   dbms_output.put_line('黄河是中国的母亲河');
 3 END;
 4 /
黄河是中国的母亲河
      
```
6.
 

```

SQL> DECLARE
 2   v_result NUMBER;
 3 BEGIN
 4   v_result:=&nol/&no2;
 5   dbms_output.put_line('相除结果:'||v_result);
 6 END;
 7 /
输入 nol 的值: 10
输入 no2 的值: 4
      
```

相除结果:2.5

7.

```
SQL> VAR v_result VARCHAR2(10)
SQL> BEGIN
2   :v_result:='中国';
3 END;
4 /
SQL> PRINT v_result
V_RESULT
-----
中国
```

## 第 4 章

1. 当检索数据库数据时，需要使用 SELECT 语句。

(1) 因为只需要显示部门名，所以在 SELECT 语句中只需要指定 DNAME 列。

```
SQL> SELECT dname FROM dept;
DNAME
-----
ACCOUNTING
RESEARCH
SALES
OPERATIONS
```

(2) 因为在 COMM 列上存在 NULL，而包含 NULL 的算术表达式结果为 NULL，所以必须要使用函数 NVL 处理 NULL。

```
SQL> set pagesize 30
SQL> SELECT ename,sal*12+nvl(comm,0) "年收入" FROM emp;
ENAME      年收入
-----
SMITH      9600
ALLEN     19500
WARD       15500
JONES      35700
MARTIN    16400
BLAKE      34200
CLARK      29400
SCOTT      36050
KING       60000
TURNER    18000
ADAMS      13200
JAMES      11400
FORD       36000
MILLER    15600
已选择 14 行。
```

(3) 因为部门与雇员之间具有一对多的关系，所以如果直接显示部门号，就会有许多重

复值。为了避免显示重复值，应该使用 DISTINCT 取消重复值。

```
SQL> SELECT DISTINCT deptno FROM emp;
      DEPTNO
-----
      10
      20
      30
```

## 2. 当要限制查询结果时，在 SELECT 语句中使用 WHERE 子句。

### (1) 当要显示超过某值的行时，应该在 WHERE 子句中使用 > 操作符。

```
SQL> SELECT ename,sal FROM emp WHERE sal>2850;
      ENAME        SAL
-----
      JONES        2975
      SCOTT        3000
      KING         5000
      FORD         3000
```

### (2) 当要显示不在范围内的行时，应该在 WHERE 子句中使用 NOT BETWEEN ... AND 操作符。

```
SQL> SELECT ename,sal FROM emp
  2 WHERE sal NOT BETWEEN 1500 AND 2850;
      ENAME        SAL
-----
      SMITH        800
      WARD         1250
      JONES        2975
      MARTIN       1250
      SCOTT        3000
      KING         5000
      ADAMS        1100
      JAMES         950
      FORD         3000
      MILLER       1300
```

已选择 10 行。

### (3) 当要显示等于某值的行时，应该在 WHERE 子句中使用 = 操作符。

```
SQL> SELECT ename,sal FROM emp WHERE empno=7566;
      ENAME        SAL
-----
      JONES        2975
```

### (4) 当要指定多个条件时，应该在 WHERE 子句中使用 AND 或 OR 操作符。

```
SQL> SELECT ename,sal FROM emp
  2 WHERE deptno IN (10,30) AND sal>1500;
      ENAME        SAL
-----
      ALLEN       1600
      BLAKE       2850
```

```
CLARK      2450
KING       5000
```

- (5) 当检测 NULL 时, 在 WHERE 子句中必须要使用 IS NULL 操作符, 而不能使用 = NULL.

```
SQL> SELECT ename, job FROM emp WHERE mgr IS NULL;
ENAME      JOB
-----
KING      PRESIDENT
```

3. 通过在 SELECT 语句中使用 ORDER BY 子句可以对数据进行排序。

- (1) 当要在 WHERE 子句中引用日期值时, 必须要与默认日期语言和显示格式匹配。

```
SQL> SELECT ename, job, hiredate FROM emp
  2 WHERE hiredate BETWEEN '01-2月 -81' AND '01-5月 -81'
  3 ORDER BY hiredate;
ENAME      JOB      HIREDATE
-----
ALLEN     SALESMAN  20-2月 -81
WARD      SALESMAN  22-2月 -81
JONES     MANAGER   02-4月 -81
BLAKE     MANAGER   01-5月 -81
```

- (2) 当执行多列排序时, 首先会以第一列进行排序; 当第一列的值相同时, 会以第二列进行排序。

```
SQL> SELECT ename, sal, comm FROM emp WHERE comm IS NOT NULL
  2 ORDER BY sal DESC, comm DESC;
ENAME      SAL      COMM
-----
SCOTT     3000      50
ALLEN     1600      300
TURNER    1500      0
MARTIN    1250      1400
WARD      1250      500
```

4. 当使用 INSERT 语句为表插入数据时, 如果要给所有列全部插入数据, 那么, 不需要指定列, 列表, 但数据顺序必须要与列的顺序保持一致。

```
SQL> INSERT INTO dept VALUES(50, 'ADMINISTRATOR', 'BOSTON');
```

5. 当使用 INSERT 语句为表插入数据时, 如果只给某些列插入数据, 那么, 必须要指定列的列表, 并且必须要为 NOT NULL 列提供数据。

```
SQL> INSERT INTO emp (empno, ename, sal, deptno, hiredate)
  2 VALUES(1587, 'JOHN', 1000, 30,
  3 to_date('1987-03-05', 'YYYY-MM-DD'));
```

6. 如果要更新表行数据, 那么, 必须要使用 UPDATE 语句。因为要更新部门 10 的雇员工资, 所以必须要指定 WHERE 子句。

```
SQL> UPDATE emp SET sal=sal*1.1 WHERE deptno=10;
```

7. 如果要删除表行数据, 那么, 必须使用 DELETE 语句。因为要删除部门 50, 所以必须要指定 WHERE 子句。

```
SQL> DELETE FROM dept WHERE deptno=50;
```

8. 当要确认事务变化时，使用 COMMIT 语句提交事务：COMMIT；
9. 当要对数据进行分组统计时，可以在 SELECT 语句中使用 GROUP BY、HAVING 子句、分组函数，以及 CUBE 和 ROLLUP 操作符。

(1) 使用分组函数 AVG, SUM, MAX, MIN 可以取得平均值、总和、最大值和最小值。

```
SQL> SELECT avg(sal),sum(sal),max(sal),min(sal) FROM emp;
      AVG(SAL)    SUM(SAL)    MAX(SAL)    MIN(SAL)
----- ----- ----- -----
      2361.07143     33055      5500       950
```

(2) 因为要以岗位执行分组统计，所以在 GROUP BY 子句中应该指定 JOB 为分组列。

```
SQL> SELECT job,count(*),avg(sal) FROM emp
  2 GROUP BY job;
      JOB      COUNT(*)    AVG(SAL)
----- -----
ANALYST        2          3500
CLERK          4          1620
MANAGER         3          2825
PRESIDENT       1          5500
SALESMAN        4          1400
```

(3) 当统计行数时，可以使用 COUNT 函数。而如果要统计 NOT NULL 行，就需要使用 COUNT(expr)。

```
SQL> SELECT count(*),count(comm) FROM emp;
      COUNT(*) COUNT(COMM)
----- -----
           14          4
```

(4) MGR 列对应于管理者代码，但因为该列具有重复值，所以如果要显示管理者总数，应该使用 DISTINCT 选项取消重复值。

```
SQL> SELECT count(distinct mgr) FROM emp;
      COUNT(DISTINCT MGR)
----- -----
               6
```

(5) 工资最大差额=雇员最高工资-雇员最低工资。

```
SQL> SELECT max(sal)-min(sal) FROM emp;
      MAX(SAL)-MIN(SAL)
----- -----
            4550
```

(6) 当执行分组统计时，如果在统计数据中要显示横向和纵向小计值，必须要在 GROUP BY 子句中使用 CUBE 操作符。

```
SQL> SELECT deptno,job,avg(sal) FROM emp
  2 GROUP BY CUBE(deptno,job);
      DEPTNO   JOB      AVG(SAL)
```

```
----- -----
          2073.21429
      CLERK      1037.5
      ANALYST     3000
```

```

MANAGER    2758.33333
SALESMAN     1400
PRESIDENT    5000
10          2916.66667
10 CLERK      1300
10 MANAGER    2450
10 PRESIDENT   5000
20          2175
20 CLERK      950
20 ANALYST    3000
20 MANAGER    2975
30          1566.66667
30 CLERK      950
30 MANAGER    2850
30 SALESMAN    1400

```

已选择 18 行。

10. 当显示多表数据时，必须使用连接查询。而当执行连接查询时，必须要指定连接条件，否则会产生笛卡儿集。

- (1) 因为要显示部门 20 的部门名和雇员信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（部门号=20）。

```

SQL> SELECT a.dname,b.ename,b.sal,b.job FROM dept a,emp b
  2 WHERE a.deptno=b.deptno AND a.deptno=20;

```

ENAME	DNAME	SAL	JOB
SMITH	RESEARCH	3000	CLERK
JONES	RESEARCH	2975	MANAGER
SCOTT	RESEARCH	4000	ANALYST
ADAMS	RESEARCH	1100	CLERK
FORD	RESEARCH	3000	ANALYST

- (2) 因为要显示获得补助的雇员及其部门信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（补助非空）。

```

SQL> SELECT a.ename,a.comm,b.dname FROM emp a,dept b
  2 WHERE a.deptno=b.deptno AND a.comm IS NOT NULL;

```

ENAME	COMM	DNAME
ALLEN	300	SALES
WARD	500	SALES
MARTIN	1400	SALES
TURNER	0	SALES

- (3) 因为要显示工作在 DALLAS 的雇员及部门信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（部门位置=DALLAS）。

```

SQL> SELECT a.ename,a.sal,b.dname FROM emp a,dept b
  2 WHERE a.deptno=b.deptno AND b.loc='DALLAS';

```

ENAME	SAL	DNAME
-------	-----	-------

SMITH	3000	RESEARCH
JONES	2975	RESEARCH
SCOTT	4000	RESEARCH
ADAMS	1100	RESEARCH
FORD	3000	RESEARCH

(4) 在 EMP 表的 EMPNO 列和 MGR 列之间具有参照关系，在这两个列之间使用自连接可以显示雇员之间的上下级关系。

```
SQL> SELECT m.ename FROM emp w,emp m
  2 WHERE w.mgr=m.empno AND w.ename='SCOTT';
ENAME
-----
JONES
```

(5) 因为工资级别对应于一定的工资范围，所以当要显示雇员的工资级别时，可以在 EMP 表和 SALGRADE 表之间执行不等连接。

```
SQL> SELECT a.ename,a.sal,b.grade FROM emp a,salgrade b
  2 WHERE a.sal BETWEEN b.losal AND b.hisal AND a.deptno=20;
ENAME      SAL      GRADE
-----
ADAMS      1100      1
SMITH      3000      4
JONES      2975      4
FORD       3000      4
SCOTT      4000      5
```

(6) 因为要显示部门 10 的雇员及其部门信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（部门号=10）。另外，因为还要显示所有其它部门的名称，所以必须要使用外连接。从 Oracle9i 开始，Oracle 建议在 FROM 子句中指定外连接语句。

```
SQL> SELECT a.ename,b.dname FROM emp a RIGHT JOIN dept b
  2 ON a.deptno=b.deptno AND a.deptno=10;
ENAME      DNAME
-----
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER    ACCOUNTING
          SALES
          OPERATIONS
          RESEARCH
```

(7) 因为要显示部门 10 的雇员及其部门信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（部门号=10）。另外，因为还要显示所有其它雇员的名称，所以必须要使用外连接。

```
SQL> SELECT a.ename,b.dname FROM emp a LEFT JOIN dept b
  2 ON a.deptno=b.deptno AND a.deptno=10 ORDER BY b.dname;
ENAME      DNAME
-----
CLARK      ACCOUNTING
```

```

KING      ACCOUNTING
MILLER    ACCOUNTING
SMITH
ALLEN
WARD
TURNER
JAMES
FORD
ADAMS
SCOTT
JONES
MARTIN
BLAKE
已选择 14 行。

```

(8) 因为要显示部门 10 的雇员及其部门信息，所以不仅需要指定连接条件，而且还需要使用 AND 指定其他条件（部门号=10）。另外因为还要显示所有其它部门的名称和雇员的名称，所以必须要使用完全外连接。

```

SQL> SELECT a.ename,b.dname FROM emp a FULL JOIN dept b
  2 ON a.deptno=b.deptno AND a.deptno=10 ORDER BY b.dname;
        ENAME      DNAME
        -----
CLARK      ACCOUNTING
KING      ACCOUNTING
MILLER    ACCOUNTING
          OPERATIONS
          RESEARCH
          SALES
SMITH
ALLEN
WARD
TURNER
JAMES
FORD
ADAMS
SCOTT
JONES
MARTIN
BLAKE
已选择 17 行。

```

#### 11. 当要取得未知数据时，可能需要用到子查询

(1) 当显示 BLAKE 的同事时，首先需要知道 BLAKE 所在部门号，然后根据该部门号就可以确定其同事信息了。因为部门号是未知值，所以可以用子查询取得其数值。另外因为不希望显示 BLAKE 的信息，所以还要用 AND 谓词排除该雇员。

```

SQL> SELECT ename,sal FROM emp WHERE deptno-
  2 (SELECT deptno FROM emp WHERE ename='BLAKE')

```

```

3 AND ename<>'BLAKE';
ENAME      SAL
-----
ALLEN      1600
WARD       1250
MARTIN    1250
TURNER     1500
JAMES      950

```

(2) 当要显示超过平均工资的雇员信息时，首先需要取得平均工资，然后指定查询条件显示所需信息。因为平均工资为未知值，所以可以使用子查询取得其数据。

```

SQL> SELECT ename,sal,deptno FROM emp WHERE sal>
2 (SELECT avg(sal) FROM emp);
ENAME      SAL      DEPTNO
-----
SMITH     3000      20
JONES     2975      20
BLAKE     2850      30
CLARK     2650      10
SCOTT     4000      20
KING      5500      10

```

(3) 因为要显示超过部门平均工资的所有雇员信息，所以必须首先确定每个部门的平均工资，然后才将每个雇员的工资与其部门平均工资进行比较。要完成该项任务，在 FROM 子句中使用内嵌视图取得部门号及其平均工资，然后按照部门号进行连接查询，并使用 AND 谓词指定其它比较条件。

```

SQL> SELECT a.ename,a.sal,a.deptno FROM emp a,
2 (SELECT deptno,avg(sal) avgsal FROM emp GROUP BY deptno) b
3 WHERE a.deptno=b.deptno AND a.sal>b.avgsal;
ENAME      SAL      DEPTNO
-----
KING      5500      10
SMITH     3000      20
FORD      3000      20
SCOTT     4000      20
JONES     2975      20
ALLEN     1600      30
BLAKE     2850      30

```

(4) 因为要显示高于 CLERK 岗位所有雇员工资的雇员，所以需要使用多行子查询，并且需要使用多行比较符 ALL。

```

SQL> SELECT ename,sal,job FROM emp WHERE sal>ALL
2 (SELECT sal FROM emp WHERE job='CLERK');
ENAME      SAL JOB
-----
SCOTT     4000 ANALYST
KING      5500 PRESIDENT

```

(5) 因为要显示工资和补助与 SCOTT 完全匹配的雇员，所以需要使用多列子查询。另

外因为 COMM 列存在 NULL，所以必须要使用 NVL 函数处理 NULL。

```
SQL> SELECT ename,sal,comm FROM emp WHERE (sal,nvl(comm,-1))=
  2 (SELECT sal,nvl(comm,-1) FROM emp WHERE ename='SCOTT');
    ENAME      SAL      COMM
    -----
    SCOTT      3000
    FORD       3000
```

12. 使用集合操作符 UNION、UNION ALL、INTERSECT、MINUS 可以合并查询结果。

(1) 执行如下语句建立视图。

```
SQL> CREATE VIEW dept_20 AS SELECT * FROM emp WHERE deptno=20;
SQL> CREATE VIEW job_clerk AS SELECT * FROM emp WHERE job='CLERK';
```

(2) 当要显示查询结果的并集时，既可以使用 UNION 操作符，也可以使用 UNION ALL 操作符。但是如果要求取消重复值，则必须使用 UNION 操作符。当使用 UNION 操作符时，会自动按照第一列进行升序排序。

```
SQL> SELECT ename,sal FROM dept_20 UNION
  2 SELECT ename,sal FROM job_clerk;
    ENAME      SAL
    -----
    ADAMS     1100
    FORD      3000
    JAMES     950
    JONES     2975
    MILLER    1300
    SCOTT     3000
    SMITH     800
已选择 7 行。
```

(3) 当要显示查询结果的并集时，既可以使用 UNION 操作符，也可以使用 UNION ALL 操作符。但是如果要求保留重复值，则必须使用 UNION ALL 操作符。当使用该操作符时，不会进行排序。

```
SQL> SELECT ename,sal FROM dept_20 UNION ALL
  2 SELECT ename,sal FROM job_clerk;
    ENAME      SAL
    -----
    SMITH     800
    JONES     2975
    SCOTT     3000
    ADAMS     1100
    FORD      3000
    SMITH     800
    ADAMS     1100
    JAMES     950
    MILLER    1300
已选择 9 行。
```

(4) 当要显示查询结果的交集时，可以使用 INTERSECT 操作符。当使用该操作符时，

会自动按照第一列进行升序排序。

```
SQL> SELECT ename,sal FROM dept_20 INTERSECT
  2  SELECT ename,sal FROM job_clerk;
ENAME      SAL
-----
ADAMS      1100
SMITH      800
```

(5) 当要显示查询结果的差集时, 可以使用 MINUS 操作符。当使用该操作符时, 会自动按照第一列进行升序排序。

```
SQL> SELECT ename,sal FROM dept_20 MINUS
  2  SELECT ename,sal FROM job_clerk;
ENAME      SAL
-----
FORD      3000
JONES     2975
SCOTT     3000
```

13. 当在自参照表上显示上下级以及层次关系时, 既可以使用自连接, 也可以使用层次查询。但是为了更直观地显示层次关系, 应该使用层次查询。因为只需要显示两层雇员的信息, 所以在 WHERE 子句中应该使用伪列 LEVEL 作为条件。

```
SQL> SELECT LPAD(' ',3*(LEVEL-1))||ename ename,
  2  LPAD(' ',3*(LEVEL-1))||job job FROM emp
  3  WHERE level<3 START WITH mgr IS NULL
  4  CONNECT BY mgr=PRIOR empno;
ENAME      JOB
-----
KING      PRESIDENT
JONES     MANAGER
BLAKE     MANAGER
CLARK     MANAGER
```

## 第 5 章

1. A
  2. C
  3. D
  4. B
  5. A
  - 6.
- ```
SQL> set serveroutput on
SQL> DECLARE
  2  v1 NUMBER;
  3  v2 NUMBER;
  4  BEGIN
```

```

5   v1:=round(&&no);
6   v2:=trunc(&no);
7   dbms_output.put_line('四舍五入结果:'||v1);
8   dbms_output.put_line('整数值:'||v2);
9 END;
10 /
输入 no 的值: 56.67
四舍五入结果:57
整数值:56

```

7.

```

SQL> DECLARE
2   v_str1 VARCHAR2(100);
3   v_str2 VARCHAR2(100);
4   v_str3 VARCHAR2(100);
5 BEGIN
6   v_str1:=UPPER('&&string');
7   v_str2:=LOWER('&string');
8   v_str3:=INITCAP('&string');
9   dbms_output.put_line('大写格式:'||v_str1);
10  dbms_output.put_line('小写格式:'||v_str2);
11  dbms_output.put_line('首字符大写:'||v_str3);
12 END;
13 /
输入 string 的值: my china
大写格式:MY CHINA
小写格式:my china
首字符大写:My China

```

8.

```

SQL> DECLARE
2   v_temp DATE;
3   v_before DATE;
4   v_after DATE;
5 BEGIN
6   v_temp:=TO_DATE('&date','YYYY-MM-DD');
7   v_before:=v_temp-TO_YMINTERVAL('05-10');
8   v_after:=ADD_MONTHS(v_temp,10);
9   dbms_output.put_line('5年10个月之前的日期:'||v_before);
10  dbms_output.put_line('10个月之后的日期:'||v_after);
11 END;
12 /
输入 date 的值: 2003-10-10
5年10个月之前的日期:10-12月-97
10个月之后的日期:10-8月-04

```

9.

```

SQL> DECLARE
2   v_char VARCHAR2(20);

```

```

3 BEGIN
4   v_char:=TO_CHAR(&number,'L99,999');
5   dbms_output.put_line('本地货币格式:'||v_char);
6 END;
7 /
输入 number 的值: 1500
本地货币格式:      RMB1,500

```

10.

```

SQL> DECLARE
2   v_avg NUMBER(6,2);
3   v_total NUMBER(10,2);
4 BEGIN
5   SELECT avg(actual_price),sum(total)
6   INTO v_avg,v_total FROM item;
7   dbms_output.put_line('产品平均单价:'||v_avg);
8   dbms_output.put_line('所有产品价格总和:'||v_total);
9 END;
10 /
产品平均单价:34.42
所有产品价格总和:3903.8

```

## 第 6 章

1. C、D

2. C

3. C

4.

```

SQL> BEGIN
2   INSERT INTO customer (customer_id,name)
3   VALUES(&id,'&name');
4   COMMIT;
5 END;
6 /
输入 id 的值: 100
输入 name 的值: SCOTT

```

5.

```

SQL> BEGIN
2   UPDATE customer SET city='&city'
3   WHERE upper(name)=upper('&name');
4   COMMIT;
5 END;
6 /
输入 city 的值: NEW YORK
输入 name 的值: scott

```

6.

```

SQL> DECLARE
  2      v_city customer.city%TYPE;
  3      v_name customer.name%TYPE;
  4  BEGIN
  5      SELECT name,city INTO v_name,v_city
  6      FROM customer WHERE customer_id=&id;
  7      dbms_output.put_line('客户名称:'||v_name);
  8      dbms_output.put_line('客户所在城市:'||v_city);
  9  END;
10 /
输入 id 的值: 100
客户名称:SCOTT
客户所在城市:NEW YORK

```

7.

```

SQL> DECLARE
  2      v_rowcount INT;
  3  BEGIN
  4      DELETE FROM customer WHERE upper(name)=upper('&name');
  5      v_rowcount:=SQL%ROWCOUNT;
  6      dbms_output.put_line('删除了'||v_rowcount||'行');
  7  END;
  8 /
输入 name 的值: scott
删除了 1 行

```

## 第 7 章

1. B

2. A

3.

```

SQL> BEGIN
  2      FOR i IN 1..10 LOOP
  3          IF i<>5 AND i<>7 THEN
  4              INSERT INTO temp VALUES(i);
  5          END IF;
  6      END LOOP;
  7  END;
  8 /

```

4.

```

SQL> DECLARE
  2      v_name VARCHAR2(20);
  3      v_city VARCHAR2(20);
  4  BEGIN
  5      v_name:='&customer_name';

```

```

6   v_city:='&city';
7   UPDATE customer SET city=v_city
8   WHERE upper(name)=upper(v_name);
9   IF SQL%NOTFOUND THEN
10      dbms_output.put_line('不存在该客户');
11   END IF;
12 END;
13 /

```

输入 customer\_name 的值: clark  
输入 city 的值: Boston

5.

```

SQL> DECLARE
2   v_deptno NUMBER(2);
3   BEGIN
4   v_deptno:=&deptno;
5   CASE v_deptno
6   WHEN 10 THEN
7     UPDATE emp set sal=sal*1.1 WHERE deptno=v_deptno;
8   WHEN 20 THEN
9     UPDATE emp set sal=sal*1.08 WHERE deptno=v_deptno;
10  WHEN 30 THEN
11    UPDATE emp set sal=sal*1.05 WHERE deptno=v_deptno;
12  ELSE
13    dbms_output.put_line('不存在该部门');
14  END CASE;
15 END;
16 /

```

输入 deptno 的值: 30

## 第 8 章

1. B、D

2. B

3. B、E、F

4.

```

SQL> set serveroutput on
SQL> DECLARE
2   TYPE cust_record_type IS RECORD(
3     name customer.name%TYPE, total ord.total%TYPE
4   );
5   cust_record cust_record_type;
6   BEGIN
7   SELECT a.name,b.total INTO cust_record
8   FROM customer a,ord b
9   WHERE a.customer_id=b.customer_id AND b.ord_id=&id;

```

```
10    dbms_output.put_line('客户名:'||cust_record.name);
11    dbms_output.put_line('订单总额:'||cust_record.total);
12 END;
13 /
```

输入 id 的值: 610

客户名:BOB

订单总额:101.4

5.

```
SQL> DECLARE
  2    product_record product%ROWTYPE;
  3  BEGIN
  4    product_record.product_id:=&id;
  5    product_record.description:='&description';
  6    INSERT INTO product VALUES product_record;
  7  END;
  8 /
```

输入 id 的值: 100900

输入 description 的值: 高尔夫球

6.

```
SQL> DECLARE
  2    TYPE name_array_type IS VARRAY(20) OF VARCHAR2(30);
  3    TYPE city_array_type IS VARRAY(20) OF VARCHAR2(30);
  4    name_array name_array_type;
  5    city_array city_array_type;
  6  BEGIN
  7    SELECT name,city BULK COLLECT
  8      INTO name_array,city_array FROM customer;
  9    FOR i IN 1..name_array.COUNT LOOP
10      dbms_output.put_line('客户名:'||name_array(i)
11        ||',所在城市:'||city_array(i));
12    END LOOP;
13  END;
14 /
```

客户名:BOB, 所在城市:SPRING

客户名:MARY, 所在城市:HOUSTON

客户名:BLAKE, 所在城市:CHELSEA

客户名:CLARK, 所在城市:GRAPEVINE

客户名:SMITH, 所在城市:FLUSHING

客户名:KING, 所在城市:HOUSTON

客户名:STEVEN, 所在城市:DALLAS

客户名:KLINTON, 所在城市:MALDEN

7.

```
SQL> DECLARE
  2    TYPE item_table_type IS TABLE OF item%ROWTYPE
  3    INDEX BY PLS_INTEGER;
  4    item_table item_table_type;
```

```

5  BEGIN
6    SELECT * BULK COLLECT INTO item_table
7    FROM item WHERE ord_id=&id;
8    FOR i IN 1..item_table.COUNT LOOP
9      dbms_output.put_line('条款编号:'|||
10        item_table(i).item_id||',总价:'|||
11        item_table(i).total);
12    END LOOP;
13  END;
14 /
输入 id 的值: 600
条款编号:1,总价:42
条款编号:2,总价:58

```

## 第 9 章

1. B
2. C、E
3. A
- 4.

```

SQL> DECLARE
2   CURSOR ord_cursor IS SELECT a.ord_id,b.name,a.total
3     FROM ord a,customer b
4     WHERE a.customer_id=b.customer_id
5     ORDER BY a.total DESC;
6   ord_record ord_cursor%ROWTYPE;
7   BEGIN
8     OPEN ord_cursor;
9     LOOP
10       FETCH ord_cursor INTO ord_record;
11       EXIT WHEN ord_cursor%NOTFOUND;
12       dbms_output.put_line('订单号:'||ord_record.ord_id||
13         ',客户名:'||ord_record.name|||
14         ',总价:'||ord_record.total);
15     END LOOP;
16     CLOSE ord_cursor;
17   END;
18 /
订单号:612,客户名:BLAKE,总价:5860
订单号:610,客户名:BOB,总价:101.4
订单号:601,客户名:CLARK,总价:60.8
订单号:602,客户名:SMITH,总价:56
订单号:611,客户名:MARY,总价:45
订单号:600,客户名:KING,总价:42

```

- 5.

```

SQL> DECLARE
  2   CURSOR ord_cursor IS
  3     SELECT ord_id,ord_date,ship_date FROM ord
  4     FOR UPDATE;
  5 BEGIN
  6   FOR ord_rec IN ord_cursor LOOP
  7     IF ord_rec.ship_date-ord_rec.ord_date>15 THEN
  8       dbms_output.put_line('订单号:'||ord_rec.ord_id
  9           ||',预订日期:'||ord_rec.ord_date
 10          ||',交付日期:'||ord_rec.ship_date);
 11     UPDATE ord SET ship_date=ord_date+15
 12     WHERE CURRENT OF ord_cursor;
 13   END IF;
 14   END LOOP;
 15 END;
 16 /

```

订单号:601,预订日期:01-5月 -90,交付日期:30-5月 -90

订单号:600,预订日期:01-5月 -90,交付日期:29-5月 -90

6.

```

SQL> DECLARE
  2   CURSOR item_cursor(id NUMBER) IS
  3     SELECT item_id,total,CURSOR(SELECT description
  4       FROM product WHERE product_id=item.product_id)
  5     FROM item WHERE ord_id=id;
  6   TYPE ref_cur_type IS REF CURSOR;
  7   prod_ref ref_cur_type;
  8   v_itemid item.item_id%TYPE;
  9   v_total item.total%TYPE;
 10   v_desc product.description%TYPE;
 11 BEGIN
 12   OPEN item_cursor(&id);
 13   LOOP
 14     FETCH item_cursor INTO v_itemid,v_total,prod_ref;
 15     EXIT WHEN item_cursor%NOTFOUND;
 16     dbms_output.put('条款号:'||v_itemid||
 17           ',总价:'||v_total);
 18   LOOP
 19     FETCH prod_ref INTO v_desc;
 20     EXIT WHEN prod_ref%NOTFOUND;
 21     dbms_output.put(',产品名:'||v_desc);
 22   END LOOP;
 23   dbms_output.new_line;
 24   END LOOP;
 25   CLOSE item_cursor;
 26 END;
 27 /

```

```
输入 id 的值: 600
条款号:1, 总价:42, 产品名:ACE TENNIS RACKET II
条款号:2, 总价:58, 产品名:ACE TENNIS NET
```

## 第 10 章

1. C
2. A
3. C
4. D
- 5.

```
SQL> set serveroutput on
SQL> DECLARE
 2   v_shipdate ord.ship_date%TYPE;
 3   v_total ord.total%TYPE;
 4 BEGIN
 5   SELECT ship_date,total INTO v_shipdate,v_total
 6   FROM ord WHERE ord_id=&id;
 7   dbms_output.put_line('订单总价:'||v_total
 8   ||',交付日期:'||v_shipdate);
 9 EXCEPTION
10   WHEN NO_DATA_FOUND THEN
11     dbms_output.put_line('该订单不存在');
12 END;
13 /
```

输入 id 的值: 667  
该订单不存在

- 6.

```
SQL> DECLARE
 2   v_cid ord.customer_id%TYPE;
 3   v_oid ord.ord_id%TYPE;
 4   e_integrity EXCEPTION;
 5   e_no_rows EXCEPTION;
 6   PRAGMA EXCEPTION_INIT(e_integrity,-2291);
 7 BEGIN
 8   v_cid:=&customer_id;
 9   v_oid:=&ord_id;
10   UPDATE ord SET customer_id=v_cid
11   WHERE ord_id=v_oid;
12   IF SQL%FOUND THEN
13     dbms_output.put_line('订单'||v_oid||
14     '的新客户号:'||v_cid);
15   ELSE
16     RAISE e_no_rows;
17 END IF;
```

```

18 EXCEPTION
19 WHEN e_integrity THEN
20   dbms_output.put_line('请检查客户号，然后输入正确的客户号');
21 WHEN e_no_rows THEN
22   dbms_output.put_line('请检查订单号，然后输入正确的订单号');
23 END;
24 /
输入 customer_id 的值: 215
输入 ord_id 的值: 1111
请检查订单号，然后输入正确的订单号

```

7.

```

SQL> DECLARE
2   v_name customer.name%TYPE;
3   e_integrity EXCEPTION;
4   e_no_rows EXCEPTION;
5   PRAGMA EXCEPTION_INIT(e_integrity,-2292);
6 BEGIN
7   DELETE FROM customer WHERE customer_id=&id
8   RETURNING name INTO v_name;
9   IF SQL%FOUND THEN
10    dbms_output.put_line('删除了客户'||v_name);
11 ELSE
12   RAISE e_no_rows;
13 END IF;
14 EXCEPTION
15 WHEN e_integrity THEN
16   dbms_output.put_line('有订单的客户不能被删除');
17 WHEN e_no_rows THEN
18   dbms_output.put_line('该客户不存在');
19 END;
20 /
输入 id 的值: 215
有订单的客户不能被删除

```

## 第 11 章

1. B
2. B、C
- 3.

```

SQL> CREATE OR REPLACE FUNCTION valid_cust(id NUMBER)
2  RETURN BOOLEAN IS
3  v_temp INT;
4 BEGIN
5   SELECT 1 INTO v_temp FROM customer
6   WHERE customer_id=id;

```

```
7    RETURN TRUE;
8  EXCEPTION
9    WHEN NO_DATA_FOUND THEN
10      RETURN FALSE;
11  END;
12 /  
  
4.  
SQL> CREATE OR REPLACE PROCEDURE add_ord(
2    id NUMBER,orddate DATE,cid NUMBER,
3    shipdate DATE,total NUMBER)
4  IS
5  BEGIN
6    IF NOT valid_cust(cid) THEN
7      raise_application_error(-20001,
8          '检查并输入正确的客户号');
9    END IF;
10   IF shipdate<orddate THEN
11      raise_application_error(-20002,
12          '交付日期必须在预订日期之后');
13    END IF;
14   INSERT INTO ord VALUES(id,orddate,cid,shipdate,total);
15 EXCEPTION
16   WHEN DUP_VAL_ON_INDEX THEN
17      raise_application_error(-20003,
18          '该订单已经存在');
19 END;
20 /  
SQL> exec add_ord(605,SYSDATE,223,SYSDATE+15,1000)
PL/SQL 过程已成功完成。  
  
5.  
SQL> CREATE OR REPLACE PROCEDURE upd_shipdate(
2    id NUMBER,shipdate DATE)
3  IS
4    orddate DATE;
5  BEGIN
6    UPDATE ord SET ship_date=shipdate
7    WHERE ord_id=id RETURNING ord_date INTO orddate;
8    IF SQL%NOTFOUND THEN
9      raise_application_error(-20002,
10          '该订单不存在');
11    END IF;
12    IF shipdate<orddate THEN
13      ROLLBACK;
14      raise_application_error(-20001,
15          '交付日期不能小于预订日期');
16    END IF;
```

```
17 END;
18 /
SQL> exec upd_shipdate(605,SYSDATE+10)
PL/SQL 过程已成功完成。
6.
SQL> CREATE OR REPLACE FUNCTION get_total(
 2  id NUMBER) RETURN NUMBER IS
 3  v_total ord.total%TYPE;
 4 BEGIN
 5  SELECT total INTO v_total FROM ord
 6  WHERE ord_id=id;
 7  RETURN v_total;
 8 EXCEPTION
 9  WHEN NO_DATA_FOUND THEN
10    raise_application_error(-20001,
11      '请检查并输入正确的订单号');
12 END;
13 /
SQL> VAR total NUMBER
SQL> exec :total:=get_total(605)
PL/SQL 过程已成功完成。
SQL> PRINT total
      TOTAL
-----
      1000
7.
SQL> CREATE OR REPLACE PROCEDURE del_ord(
 2  id NUMBER) IS
 3 BEGIN
 4  DELETE FROM ord WHERE ord_id=id;
 5  IF SQL%NOTFOUND THEN
 6    raise_application_error(-20001,
 7      '请检查并输入正确的订单号');
 8  END IF;
 9 END;
10 /
SQL> exec del_ord(605)
PL/SQL 过程已成功完成。
```

## 第 12 章

1. A
2. A
3. B
- 4.

```
SQL> CREATE OR REPLACE PACKAGE ord_package IS
  2   PROCEDURE add_ord(id NUMBER, orddate DATE,
  3         cid NUMBER, shipdate DATE, total NUMBER);
  4   PROCEDURE upd_shipdate(id NUMBER, shipdate DATE);
  5   FUNCTION get_info(id NUMBER) RETURN VARCHAR2;
  6   PROCEDURE del_ord(id NUMBER);
  7 END;
  8 /
```

程序包已创建。

```
SQL> CREATE OR REPLACE PACKAGE BODY ord_package IS
  2   FUNCTION valid_cust(cid NUMBER) RETURN BOOLEAN
  3   IS
  4     v_temp INT;
  5   BEGIN
  6     SELECT 1 INTO v_temp FROM customer
  7     WHERE customer_id=cid;
  8     RETURN TRUE;
  9   EXCEPTION
 10     WHEN NO_DATA_FOUND THEN
 11       RETURN FALSE;
 12   END;
 13   PROCEDURE add_ord(id NUMBER, orddate DATE,
 14         cid NUMBER, shipdate DATE, total NUMBER) IS
 15   BEGIN
 16     IF NOT valid_cust(cid) THEN
 17       raise_application_error(-20002,
 18           '该客户不存在, 请核实客户号');
 19     END IF;
 20     IF shipdate<orddate THEN
 21       raise_application_error(-20003,
 22           '交付日期不能小于预订日期, 请核实交付日期');
 23     END IF;
 24     INSERT INTO ord VALUES(id, orddate,
 25         cid, shipdate, total);
 26   EXCEPTION
 27     WHEN DUP_VAL_ON_INDEX THEN
 28       raise_application_error(-20001,
 29           '该订单已存在, 请核实订单号');
 30   END;
 31   PROCEDURE upd_shipdate(id NUMBER, shipdate DATE)
 32   IS
 33     orddate DATE;
 34   BEGIN
 35     UPDATE ord SET ship_date=shipdate
 36     WHERE ord_id=id RETURNING ord_date INTO orddate;
 37     IF SQL%NOTFOUND THEN
```

```
38      raise_application_error(-20004,
39          '请检查并输入正确的订单号');
40  END IF;
41  IF shipdate<orddate THEN
42      ROLLBACK;
43      raise_application_error(-20003,
44          '交付日期不能小于预订日期, 请核实交付日期');
45  END IF;
46 END;
47 FUNCTION get_info(id NUMBER) RETURN VARCHAR2
48 IS
49     v_result VARCHAR2(100);
50 BEGIN
51     SELECT '客户名:'||b.name||', 总金额:'||a.total
52         INTO v_result FROM ord a,customer b
53         WHERE a.customer_id=b.customer_id
54         AND a.ord_id=id;
55     RETURN v_result;
56     EXCEPTION WHEN NO_DATA_FOUND THEN
57         raise_application_error(-20004,
58             '请检查并输入正确的订单号');
59 END;
60 PROCEDURE del_ord(id NUMBER) IS
61 BEGIN
62     DELETE FROM ord WHERE ord_id=id;
63     IF SQL%NOTFOUND THEN
64         raise_application_error(-20004,
65             '请检查并输入正确的订单号');
66     END IF;
67 END;
68 END;
69 /
```

程序包主体已创建。

```
SQL> exec ord_package.add_ord(606,SYSDATE,215,SYSDATE+10,2000)
PL/SQL 过程已成功完成。
SQL> exec ord_package.upd_shipdate(606,SYSDATE+9)
PL/SQL 过程已成功完成。
SQL> VAR result VARCHAR2(100)
SQL> exec :result:=ord_package.get_info(606)
PL/SQL 过程已成功完成。
SQL> PRINT result
RESULT
-----
客户名:MARY, 总金额:2000
SQL> exec ord_package.del_ord(606)
PL/SQL 过程已成功完成。
```

5.

```
SQL> CREATE OR REPLACE PACKAGE cust_package IS
  2   PROCEDURE upd_city(p_id NUMBER,p_city VARCHAR2);
  3   PROCEDURE upd_name(p_name VARCHAR2,p_city VARCHAR2);
  4   FUNCTION get_city(p_id NUMBER) RETURN VARCHAR2;
  5   FUNCTION get_name(p_name VARCHAR2) RETURN VARCHAR2;
  6 END;
  7 /
```

程序包已创建。

```
SQL> CREATE OR REPLACE PACKAGE BODY cust_package IS
  2   PROCEDURE upd_city(p_id NUMBER,p_city VARCHAR2)
  3   IS
  4   BEGIN
  5     UPDATE customer SET city=p_city
  6     WHERE customer_id=p_id;
  7     IF SQL%NOTFOUND THEN
  8       raise_application_error(-20001,
  9         '该客户不存在, 请核实客户号或客户名');
 10    END IF;
 11  END;
 12  PROCEDURE upd_name(p_name VARCHAR2,p_city VARCHAR2)
 13  IS
 14  BEGIN
 15    UPDATE customer SET city=p_city
 16    WHERE name=p_name;
 17    IF SQL%NOTFOUND THEN
 18      raise_application_error(-20001,
 19        '该客户不存在, 请核实客户号或客户名');
 20    END IF;
 21  END;
 22  FUNCTION get_city(p_id NUMBER) RETURN VARCHAR2
 23  IS
 24    v_city customer.city%TYPE;
 25  BEGIN
 26    SELECT city INTO v_city FROM customer
 27    WHERE customer_id=p_id;
 28    RETURN v_city;
 29  EXCEPTION
 30    WHEN NO_DATA_FOUND THEN
 31      raise_application_error(-20001,
 32        '该客户不存在, 请核实客户号或客户名');
 33  END;
 34  FUNCTION get_name(p_name VARCHAR2) RETURN VARCHAR2
 35  IS
 36    v_city customer.city%TYPE;
 37  BEGIN
```

```

38   SELECT city INTO v_city FROM customer
39   WHERE name=p_name;
40   RETURN v_city;
41 EXCEPTION
42   WHEN NO_DATA_FOUND THEN
43       raise_application_error(-20001,
44           '该客户不存在, 请核实客户号或客户名');
45 END;
46 END;
47 /

```

程序包主体已创建。

```

SQL> exec cust_package.upd_city(214,'NEW YORK')
PL/SQL 过程已成功完成。
SQL> exec cust_package.upd_city('MARY','HOUSTON')
PL/SQL 过程已成功完成。
SQL> VAR city VARCHAR2(20)
SQL> exec :city:=cust_package.get_city(214)
PL/SQL 过程已成功完成。
SQL> PRINT city
CITY
-----
NEW YORK
SQL> exec :city:=cust_package.get_city('MARY')
PL/SQL 过程已成功完成。
SQL> PRINT city
CITY
-----
HOUSTON

```

## 第 13 章

1. A、B
2. D
3. E
4. A、B
- 5.

```

SQL> CREATE OR REPLACE TRIGGER tr_add_ord
  2 BEFORE INSERT ON ord
  3 BEGIN
  4   IF to_char(SYSDATE,'DY') IN ('星期日','星期六')
  5   THEN
  6       raise_application_error(-20001,
  7           '只能在工作日增加订单');
  8   END IF;
  9   IF to_char(SYSDATE,'HH24') NOT BETWEEN '9' AND '17'

```

```

10      THEN
11      raise_application_error(-20002,
12          '只能在工作时间增加订单');
13  END IF;
14 END;
15 /
SQL> INSERT INTO ord VALUES(666,SYSDATE,215,SYSDATE+1,2000);
INSERT INTO ord VALUES(666,SYSDATE,215,SYSDATE+1,2000)
*
ERROR 位于第 1 行:
ORA-20002: 只能在工作时间增加订单
ORA-06512: 在"SCOTT.TR_ADD_ORD", line 9
ORA-04088: 触发器 'SCOTT.TR_ADD_ORD' 执行过程中出错

```

6.

```

SQL> CREATE OR REPLACE TRIGGER tr_upd_ordid
2 AFTER UPDATE OF ord_id ON ord
3 FOR EACH ROW
4 BEGIN
5     INSERT INTO audit_ordid_change
6         VALUES(:old.ord_id,:new.ord_id,SYSDATE);
7     UPDATE item SET ord_id=:new.ord_id
8     WHERE ord_id=:old.ord_id;
9 END;
10 /
SQL> UPDATE ord SET ord_id=666 WHERE ord_id=610;
SQL> SELECT * FROM audit_ordid_change;
OLD_ORDID NEW_ORDID CHANGETIME
-----
610      666 01-1 月 -04
SQL> SELECT item_id,total FROM item WHERE ord_id=666;
ITEM_ID      TOTAL
-----
1          174
2           84
3          384

```

7.

```

SQL> CREATE OR REPLACE TRIGGER tr_ord_item
2 INSTEAD OF UPDATE ON ord_item
3 FOR EACH ROW
4 DECLARE
5     v_ord_total ord.total%TYPE;
6     v_item_total item.total%TYPE;
7 BEGIN
8     v_item_total:=:new.actual_price*:new.quantity;
9     UPDATE item SET actual_price=:new.actual_price,
10        quantity=:new.quantity,total=v_item_total

```

```

11 WHERE ord_id=:new.ord_id AND item_id=:new.item_id;
12 SELECT sum(total) INTO v_ord_total
13 FROM item WHERE ord_id=:new.ord_id;
14 UPDATE ord SET total=v_ord_total
15 WHERE ord_id=:new.ord_id;
16 END;
17 /
SQL> UPDATE ord_item SET actual_price=40,quantity=10
2 WHERE ord_id=666 AND item_id=2;
SQL> SELECT sum(total) FROM item WHERE ord_id=666;
SUM(TOTAL)
-----
958
SQL> SELECT total FROM ord WHERE ord_id=666;
TOTAL
-----
958

```

## 第 14 章

1. C、D、E

2. A、B

3.

```

SQL> conn sys/oracle as sysdba
已连接。
SQL> CREATE OR REPLACE PROCEDURE exec_ddl(
2   ddl_stat VARCHAR2)
3 IS
4 BEGIN
5   EXECUTE IMMEDIATE ddl_stat;
6 END;
7 /
过程已创建。
SQL> exec exec_ddl('CREATE TABLE temp(cola INT)')
PL/SQL 过程已成功完成。
SQL> exec exec_ddl('DROP TABLE temp')
PL/SQL 过程已成功完成。

```

4.

```

SQL> DECLARE
2   sql_stat VARCHAR2(100);
3 BEGIN
4   sql_stat:='UPDATE item SET actual_price=:1 ||
5   ',quantity=:2 ||
6   ' WHERE ord_id=:3 AND item_id=:4';
7   EXECUTE IMMEDIATE sql_stat

```

```

8      USING &price, &quantity, &ordid, &itemid;
9  END;
10 /
输入 price 的值: 40
输入 quantity 的值: 10
输入 ordid 的值: 600
输入 itemid 的值: 1
5.
SQL> DECLARE
2   TYPE refcur IS REF CURSOR;
3   item_cv refcur;
4   item_record item%ROWTYPE;
5   sql_stat VARCHAR2(100);
6 BEGIN
7   sql_stat:='SELECT * FROM item WHERE ord_id=:1';
8   OPEN item_cv FOR sql_stat USING &ordid;
9   LOOP
10    FETCH item_cv INTO item_record;
11    EXIT WHEN item_cv%NOTFOUND;
12    dbms_output.put_line('条款号:' ||
13      ||item_record.item_id||',条款总价:' ||
14      ||item_record.total||);
15  END LOOP;
16  CLOSE item_cv;
17 END;
18 /
输入 ordid 的值: 600
条款号:1,条款总价:42
条款号:2,条款总价:58
PL/SQL 过程已成功完成。

```

6. 编写 PL/SQL 块，在动态 SQL 中使用 BULK 子句根据输入的多个订单号将交付日期更新为当前日期，并返回每个订单对应的客户号。格式如下：

```

SQL> DECLARE
2   sql_stat VARCHAR2(100);
3   TYPE ordid_table_type IS TABLE OF ord.ord_id%TYPE;
4   ordid_table ordid_table_type;
5   TYPE cid_table_type IS TABLE OF ord.customer_id%TYPE;
6   cid_table cid_table_type;
7 BEGIN
8   sql_stat:='UPDATE ord SET ship_date=SYSDATE ' ||
9   'WHERE ord_id=:b ' ||
10  'RETURNING customer_id INTO :c';
11  ordid_table:=ordid_table_type(&1,&2,&3);
12  FORALL i IN 1..ordid_table.COUNT
13  EXECUTE IMMEDIATE sql_stat USING ordid_table(i)
14  RETURNING BULK COLLECT INTO cid_table;

```

```

15   FOR i IN 1..cid_table.COUNT LOOP
16     dbms_output.put_line('订单:'||ordid_table(i)
17       ||',客户:'||cid_table(i));
18   END LOOP;
19 END;
20 /
输入 1 的值: 600
输入 2 的值: 601
输入 3 的值: 602
订单:600,客户:221
订单:601,客户:217
订单:602,客户:218
PL/SQL 过程已成功完成。

```

## 第 15 章

1. C、D
2. A
3. A、C、D
4. B
- 5.

```

SQL> CREATE OR REPLACE TYPE bank_account AS OBJECT(
2   acct_no NUMBER(6),balance NUMBER,status VARCHAR2(6),
3   MEMBER PROCEDURE deposit(amount NUMBER),
4   MEMBER PROCEDURE withdraw(amount NUMBER)
5 );
6 /
类型已创建。
SQL> CREATE OR REPLACE TYPE BODY bank_account AS
2   MEMBER PROCEDURE deposit(amount NUMBER)
3   IS
4   BEGIN
5     IF upper(status)='OPEN' THEN
6       balance:=balance+amount;
7     ELSE
8       raise_application_error(-20001,
9         '该账户已停用');
10    END IF;
11  END;
12  MEMBER PROCEDURE withdraw(amount NUMBER)
13  IS
14  BEGIN
15    IF upper(status)='OPEN' THEN
16      IF balance>=amount THEN
17        balance:=balance-amount;

```

```

18      ELSE
19          raise_application_error(-20002,
20              '账户余额不足');
21      END IF;
22  ELSE
23      raise_application_error(-20001,
24          '该账户已停用');
25  END IF;
26 END;
27 END;
28 /
类型主体已创建。

```

6.

```

SQL> CREATE TABLE account OF bank_account;
SQL> INSERT INTO account VALUES(bank_account(
2 123456,5586.87,'OPEN'));
SQL> DECLARE
2  acct bank_account;
3 BEGIN
4  SELECT VALUE(b) INTO acct FROM account b
5  WHERE b.acct_no=123456;
6  acct.deposit(2000);
7  UPDATE account b SET b=acct
8  WHERE b.acct_no=123456;
9 END;
10 /
PL/SQL 过程已成功完成。

```

```

SQL> DECLARE
2  acct bank_account;
3 BEGIN
4  SELECT VALUE(b) INTO acct FROM account b
5  WHERE b.acct_no=123456;
6  acct.withdraw(1000);
7  UPDATE account b SET b=acct
8  WHERE b.acct_no=123456;
9 END;
10 /
PL/SQL 过程已成功完成。

```

7.

```

SQL> CREATE OR REPLACE TYPE employee_type AS OBJECT(
2  eno NUMBER(6),name VARCHAR2(10),sal NUMBER(6,2),
3  job VARCHAR2(10),hiredate DATE,
4  MEMBER FUNCTION get_info RETURN VARCHAR2,
5  MEMBER PROCEDURE change_sal(salary NUMBER),
6  MEMBER PROCEDURE change_job(title NUMBER)
7 );

```

```

8 /
类型已创建。
SQL> CREATE OR REPLACE TYPE BODY employee_type IS
  2 MEMBER FUNCTION get_info RETURN VARCHAR2
  3 IS
  4   result VARCHAR2(100);
  5 BEGIN
  6   result:='雇员名:'||name||',工资:'||sal;
  7   RETURN result;
  8 END;
  9 MEMBER PROCEDURE change_sal(salary NUMBER)
10 IS
11 BEGIN
12   sal:=salary;
13 END;
14 MEMBER PROCEDURE change_job(title NUMBER)
15 IS
16 BEGIN
17   job:=title;
18 END;
19 END;
20 /

```

类型主体已创建。

8.

```

SQL> CREATE OR REPLACE TYPE employee_table_type
  2 IS TABLE OF employee_type;
  3 /

```

类型已创建。

9.

- 建立 DEPARTMENT 表

```

SQL> CREATE TABLE department(
  2 dno NUMBER(2),dname VARCHAR2(20),loc VARCHAR2(20),
  3 employee employee_table_type
  4 )NESTED TABLE employee STORE AS employee_table;

```

- 插入数据

```

SQL> INSERT INTO department VALUES(10,'财务处','北京',
  2 employee_table_type(
  3   employee_type(1,'马丽',2000,'处长','03-10月-98'),
  4   employee_type(2,'张华',1500,'会计','13-10月-99'),
  5   employee_type(3,'明珠',1200,'出纳','11-10月-01')
  6 ));

```

已创建 1 行。

```

SQL> INSERT INTO department VALUES(20,'物资处','上海',
  2 employee_table_type(
  3   employee_type(4,'秦琼',2000,'处长','03-10月-98'),
  4   employee_type(5,'罗敏',1500,'采购','13-10月-99'),

```

```

5      employee_type(6,'裴伟',1200,'保管','11-10月-01')
6  ));

```

已创建 1 行。

- 检索雇员信息

```

SQL> DECLARE
2   employee_array employee_table_type;
3 BEGIN
4   SELECT employee INTO employee_array
5   FROM department WHERE dno=&deptno;
6   FOR i IN 1..employee_array.COUNT LOOP
7     dbms_output.put_line(employee_array(i).get_info());
8   END LOOP;
9 END;
10 /

```

输入 deptno 的值: 20

雇员名:秦琼,工资:2000

雇员名:罗敏,工资:1500

雇员名:裴伟,工资:1200

- 更新雇员工资

```

SQL> DECLARE
2   employee_array employee_table_type;
3   v_dno department.dno%TYPE;
4   v_eno NUMBER(6);
5   v_sal NUMBER(6,2);
6 BEGIN
7   v_dno:=&deptno;
8   v_eno:=&empno;
9   v_sal:=&salary;
10  SELECT employee INTO employee_array
11  FROM department WHERE dno=v_dno;
12  FOR i IN 1..employee_array.COUNT LOOP
13    IF employee_array(i).eno=v_eno THEN
14      employee_array(i).sal:=v_sal;
15    END IF;
16  END LOOP;
17  UPDATE department SET employee=employee_array
18  WHERE dno=v_dno;
19 END;
20 /

```

输入 deptno 的值: 10

输入 empno 的值: 1

输入 salary 的值: 2500

PL/SQL 过程已成功完成。

## 第 16 章

1. B

2. A

3. C

4.

```
SQL> CREATE TABLE employee(
 2   id NUMBER(6),name VARCHAR2(10),sal NUMBER(6,2),
 3   dno NUMBER(2),resume CLOB,photo BLOB
 4 );
```

5.

```
SQL> INSERT INTO employee VALUES(1,'秦明',2000,10,
 2   empty_clob(),empty_blob());
```

已创建 1 行。

```
SQL> INSERT INTO employee VALUES(2,'林冲',3000,20,
 2   empty_clob(),empty_blob());
```

已创建 1 行。

```
SQL> COMMIT;
```

6.

```
SQL> conn system/manager
已连接。
```

```
SQL> CREATE DIRECTORY info AS 'g:\info';
目录已创建。
```

```
SQL> GRANT READ,WRITE ON DIRECTORY info TO scott;
授权成功。
```

7.

```
SQL> DECLARE
 2   lobloc CLOB;
 3   fileloc BFILE;
 4   amount INT;
 5   src_offset INT:=1;
 6   dest_offset INT:=1;
 7   csid INT:=0;
 8   lc INT:=0;
 9   warning INT;
10 BEGIN
11   SELECT resume INTO lobloc FROM employee
12   WHERE id=&id FOR UPDATE;
13   fileloc:=bfilename('INFO','&filename');
14   DBMS_LOB.FILEOPEN(fileloc,0);
15   amount:=DBMS_LOB.GETLENGTH(fileloc);
16   DBMS_LOB.LOADCLOBFROMFILE(lobloc,fileloc,amount,
17     dest_offset,src_offset,csid,lc,warning);
```

```
18   DBMS_LOB.FILECLOSE(fileloc);
19   COMMIT;
20 END;
21 /
输入 id 的值: 1
输入 filename 的值: qinming.txt
PL/SQL 过程已成功完成。
```

8.

```
SQL> DECLARE
  2   lobloc CLOB;
  3   amount INT;
  4   offset INT:=1;
  5   buffer VARCHAR2(2000);
  6   handle UTL_FILE.FILE_TYPE;
  7 BEGIN
  8   SELECT resume INTO lobloc FROM employee
  9   WHERE id=&id;
 10   amount:=dbms_lob.getlength(lobloc);
 11   dbms_lob.read(lobloc,amount,offset,buffer);
 12   handle:=utl_file.fopen('INFO','&filename','w',2000);
 13   utl_file.put_line(handle,buffer);
 14   utl_filefclose(handle);
 15 END;
 16 /
输入 id 的值: 1
输入 filename 的值: qm.txt
PL/SQL 过程已成功完成。
```

9.

```
SQL> DECLARE
  2   lobloc BLOB;
  3   fileloc BFILE;
  4   amount INT;
  5   src_offset INT:=1;
  6   dest_offset INT:=1;
  7 BEGIN
  8   SELECT photo INTO lobloc FROM employee
  9   WHERE id=&id FOR UPDATE;
 10   fileloc:=bfilename('INFO','&filename');
 11   DBMS_LOB.FILEOPEN(fileloc,0);
 12   amount:=DBMS_LOB.GETLENGTH(fileloc);
 13   DBMS_LOB.LOADBLOBFROMFILE(lobloc,fileloc,amount,
 14     dest_offset,src_offset);
 15   DBMS_LOB.FILECLOSE(fileloc);
 16   COMMIT;
 17 END;
 18 /
```

输入 id 的值: 1  
输入 filename 的值: qinming.bmp  
PL/SQL 过程已成功完成。

10.

```
SQL> DECLARE
  2    lobloc BLOB;
  3    len    INT;
  4    amount INT;
  5    offset INT:=1;
  6    buffer RAW(20000);
  7    handle UTL_FILE.FILE_TYPE;
  8 BEGIN
  9   SELECT photo INTO lobloc FROM employee
10   WHERE id=&id;
11   len:=dbms_lob.getlength(lobloc);
12   FOR i IN 1..CEIL(len/10000) LOOP
13     offset:=(i-1)*10000+1;
14     IF i=CEIL(len/10000) AND MOD(len,10000)<>0 THEN
15       amount:=MOD(len,10000);
16     ELSE
17       amount:=10000;
18     END IF;
19     dbms_lob.read(lobloc,amount,offset,buffer);
20     handle:=utl_file.fopen('INFO','&filename',
21                           'a',amount*2);
22     utl_file.put_raw(handle,buffer);
23     utl_file.new_line(handle);
24     utl_filefclose(handle);
25   END LOOP;
26 END;
27 /
```

输入 id 的值: 1  
输入 filename 的值: qm.bmp  
PL/SQL 过程已成功完成。

## 附录 B 使用 SQL\*Plus

SQL\*Plus 是 Oracle 提供的一个工具程序, 它不仅可以用于测试、运行 SQL 语句和 PL/SQL 块; 而且还可以用于管理 Oracle 数据库。从 Oracle9i 开始, 你不仅可以用原有的 SQL\*Plus, 而且还可以在 Web 浏览器中实现 SQL\*Plus 的功能——iSQL\*Plus。本附录会给大家介绍使用 SQL\*Plus 的方法。

### 1. 启动 SQL\*Plus

为了使用 SQL\*Plus, 必须首先要启动 SQL\*Plus。Oracle 不仅提供了命令行和图形界面的 SQL\*Plus, 而且还可以在 Web 浏览器中运行。

#### (1) 在命令行运行 SQL\*Plus

在命令行运行 SQL\*Plus 是使用 sqlplus 命令来完成的, 该命令适用于任何操作系统平台, 语法如下:

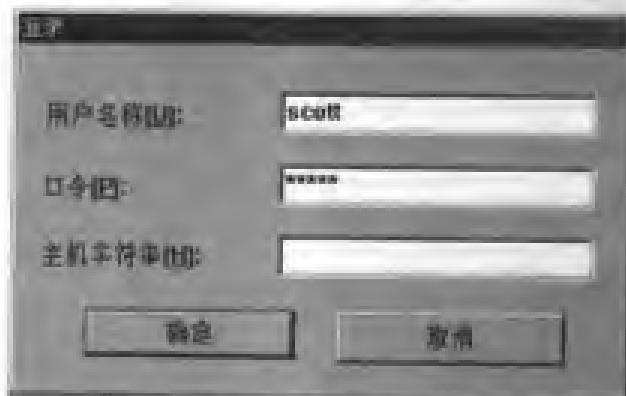
```
sqlplus [username]/{password}@server
```

如上所示, username 用于指定数据库用户名; password 用于指定用户口令; server 则用于指定主机字符串(网络服务名)。当连接到本地数据库时, 不需要提供网络服务名; 如果要连接到远程数据库, 则必须要使用网络服务名。示例如下:

```
D:\>sqlplus scott/tiger
SQL*Plus: Release 10.1.0.2.0 - Production on 星期日 3月 29 15:20:31 2004
Copyright (c) 1982, 2004, Oracle. All rights reserved.
连接到:
Oracle Database 10g Enterprise Edition Release 10.1.0.2.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

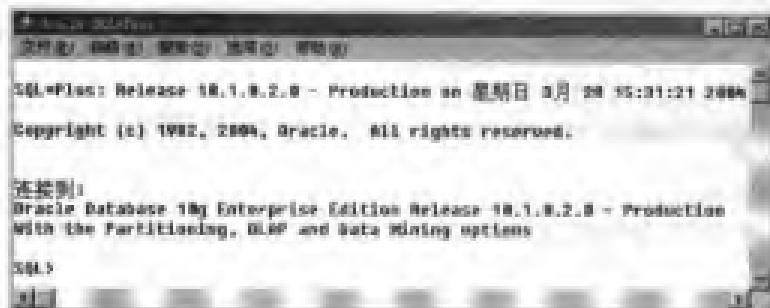
#### (2) 在 Windows 环境中运行 SQL\*Plus

如果在 Windows 环境中安装了 Oracle 数据库产品, 那么可以在窗口环境中运行 SQL\*Plus。具体方法为“开始→程序→Oracle-OraHome10→Application Development→SQL Plus”, 此时会弹出如下窗口:



如图中所示, 在输入用户名和口令之后, 单击“确定”按钮就可以连接到本地数据库。

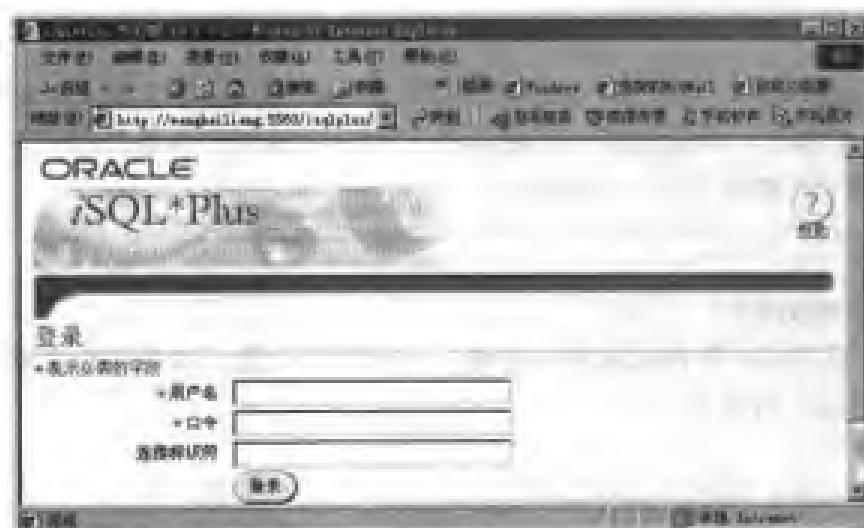
如果要连接到远程数据库，还必须在“主机字符串”处输入网络服务名。当连接到 SQL\*Plus 之后，会显示如下窗口界面：



### (3) iSQL\*Plus

iSQL\*Plus 是 Oracle9i 的新特征，它是 SQL\*Plus 在浏览器中的实现方式。在 Oracle9i 之中，为了在浏览器中运行 iSQL\*Plus，必须首先在 Oracle Server 端启动 HTTP Server；而在 Oracle10g 之中，为了在浏览器中运行 iSQL\*Plus，必须首先在 Oracle Server 端使用“isqlplusctl start”命令启动 iSQL\*Plus 应用服务器，然后才能在客户端使用浏览器连接到 iSQL\*Plus。

当使用 iSQL\*Plus 时，必须首先连接到数据库，然后才能执行 SQL\*Plus 命令、SQL 命令和 PL/SQL 语句。当使用 SQL\*Plus 时，既可以以普通数据库用户身份登录，也可以以特权用户登录。在这里只介绍以普通用户身份连接到 iSQL\*Plus 的方法。在打开浏览器之后，如果要以普通用户身份登录到数据库，必须在地址栏中输入“<http://主机名:端口号/isqlplus>”。要注意，Oracle10.1 版本默认的端口号为 5560。当输入正确地址（例如，<http://wanghailiang:5560/isqlplus>）之后，会显示如下界面：



如图中所示，在输入用户名和口令之后，单击“登录”按钮就可以连接到本地数据库；而如果要连接到远程数据库，还必须要在“连接标识符”处输入网络服务名。在连接到数据库之后，会显示如下窗口界面：



## 2. 连接命令

启动 SQL\*Plus 时，需要输入用户名、口令和网络服务名，然后才能连接到数据库。在连接到数据库之后，使用 CONNECT 命令可以切换连接；使用 DISCONNECT 可以断开连接。下面介绍与连接相关的各种命令。

### (1) CONN[NECT]

该命令用于连接到数据库。注意，使用该命令建立新会话时，会自动断开先前会话。示例如下：

```
SQL> conn scott/tiger@test
已连接。
```

当执行上述连接命令之后，会以 SCOTT 用户建立新会话，其中 test 为网络服务名。注意，当连接到远程数据库时，网络服务名是必须的。当以特权用户身份连接时，必须要带有 AS SYSDBA 或 AS SYSOPER 选项。示例如下：

```
SQL> conn sys/oracle as sysdba
已连接。
```

### (2) DISC[ONNECT]

该命令用于断开已经存在的数据库连接。大家需要注意，该命令只是断开连接会话，而不会退出 SQL\*Plus。示例如下：

```
SQL> DISC
从 Oracle10g Enterprise Edition Release 10.1.0.2.0
...
```

### (3) PASSW[ORD]

该命令用于修改用户的口令。注意，任何用户都可以使用该命令修改其自身口令，但如何修改其他用户的口令，则必须以 DBA 身份 (SYS 和 SYSTEM) 登录。在 SQL\*Plus 中，当修改用户口令时，可以使用该命令取代 SQL 命令 ALTER USER。下面以修改 SCOTT 用户的口令为例，说明使用该命令的方法。示例如下：

```
SQL> conn system/manager
已连接。
```

```
SQL> passw scott
更改 scott 的口令
新口令:
重新键入新口令:
口令已更改
```

#### (4) EXIT

该命令用于退出 SQL\*Plus，另外你也可以使用 QUIT 命令退出 SQL\*Plus。使用该命令不仅会断开连接，而且也会退出 SQL\*Plus。注意，默认情况下，当执行该命令时会自动提交事务。

### 3. 编辑命令

当在 SQL\*Plus 中执行 SQL 语句时，Oracle 会将 SQL 语句暂时存放到 SQL 缓冲区中。例如当执行语句“SELECT \* FROM dept”时，该语句会被暂时存放到 SQL 缓冲区。当执行新的 SQL 语句时，会自动清除先前 SQL 缓冲区的内容，并将新语句放到 SQL 缓冲区中。例如当执行语句“SELECT empno,ename,sal,hiredate,comm,deptno FROM emp WHERE deptno=10”时，会清除先前 SQL 语句，并将该语句放到 SQL 缓冲区中。使用 SQL\*Plus 所提供的编辑命令可以显示、编辑和修改 SQL 缓冲区的内容。例如，使用 LIST 可以列出 SQL 缓冲区内容，使用 CHANGE 可以修改 SQL 缓冲区内容，下面将详细介绍这些编辑命令。

#### (1) L[IST]

该命令用于列出 SQL 缓冲区的内容，使用该命令可以列出 SQL 缓冲区某行、某几行或所有行的内容。在显示结果中，数字为具体的行号，而“\*”则表示当前行。

##### 示例一：列出 SQL 缓冲区所有内容

```
SQL> l
1  SELECT empno,ename,sal,hiredate,comm,deptno
2  FROM emp
3* WHERE deptno=10
```

##### 示例二：列出 SQL 缓冲区首行内容：

```
SQL> l1
1* SELECT empno,ename,sal,hiredate,comm,deptno
```

#### (2) A[PPEND]

该命令用于在 SQL 缓冲区的当前行尾部添加内容。大家一定要注意，该命令将内容追加到标记为“\*”的行的尾部。示例如下：

```
SQL> a AND job='CLERK'
3* WHERE deptno=10 AND job='CLERK'
```

#### (3) C[HANGE]

该命令用于修改 SQL 缓冲区的内容。如果在编写 SQL 语句时写错了某个词，那么使用该命令可以进行修改。示例如下：

```
SQL> SELECT ename FROM temp WHERE deptno=10;
ORA-00942: 表或视图不存在
SQL> c/temp/emp
1* SELECT ename FROM emp WHERE deptno=10
```

#### (4) DEL

该命令用于删除 SQL 缓冲区中内容，使用它可以删除某行、某几行或所有行。默认情况

下，当直接执行 DEL 时，只删除当前行的内容。示例如下：

```
SQL> 1
 1 select ename,sal,hiredate,deptno from emp
 2* where deptno=10
SQL> del
SQL> 1
 1* select ename,sal,hiredate,deptno from emp
```

如果一次要删除多行，则指定起始行号和终止行号，例如“DEL 3 5”。

#### (5) I[INPUT]

该命令用于在 SQL 缓冲区的当前行后新增加一行。示例如下：

```
SQL> 1
 1* select ename,sal,hiredate,deptno from emp
SQL> i where deptno=10
SQL> 1
 1 select ename,sal,hiredate,deptno from emp
 2* where deptno=10
```

如果要在首行前增加内容，则使用“0 文本”。

```
SQL>0 create table temp as
SQL>L
 1 create table temp as
 2* select deptno from dept where dname='SALES'
```

#### (6) n

该数值用于定位 SQL 缓冲区的当前行。示例如下：

```
SQL> 1
 1 select ename,sal,hiredate,deptno from emp
 2* where deptno=10
SQL> 1
 1* select ename,sal,hiredate,deptno from emp
```

#### (7) ED[IT]

该命令用于编辑 SQL 缓冲区的内容。当运行该命令时，在 Windows 平台中会自动启动“记事本”，以编辑 SQL 缓冲区的内容。

#### (8) RUN 和/

RUN 和/命令都可以用于运行 SQL 缓冲区中的 SQL 语句。但注意，当使用 RUN 命令时，还会列出 SQL 缓冲区内容。示例如下：

```
SQL> run
 1 select ename,sal,hiredate,deptno from emp
 2* where deptno=10
ENAME          SAL HIREDATE      DEPTNO
-----  -----  -----
CLARK          2450 09-6月 -81      10
KING           5000 17-11月 -81     10
MILLER         1300 23-1月 -82     10
```

### 4. 文件操纵命令

通常情况下，当在 SQL\*Plus 中执行 SQL 命令、PL/SQL 块或 SQL\*Plus 命令时，需要使用

键盘输入命令。如果经常需要执行某些命令，可以将这些命令保存到 SQL 脚本文件中。通过使用脚本，一方面可以降低命令输入量，另一方面可以避免输入错误。下面介绍用于操纵 SQL 脚本的命令。

### (1) SAVE

该命令用于将当前 SQL 缓冲区的内容保存到 SQL 脚本中。当执行该命令时，默认选项为 CREATE，即建立新文件。示例如下：

```
SQL> SELECT * FROM dept WHERE deptno=10;
DEPTNO DNAME          LOC
-----
10 ACCOUNTING      NEW YORK
上述 SQL> SAVE c:\a.sql CREATE
已创建文件 c:\a.sql
```

当执行命令之后，就会建立新脚本文件 a.sql，并将 SQL 缓冲区内容存放到该文件中。如果 SQL 脚本已经存在，使用 REPLACE 选项可以替换已存在的 SQL 脚本；如果要给已存在的 SQL 脚本追加内容，可以使用 APPEND 选项。

### (2) GET

该命令与 SAVE 命令作用恰好相反，用于将 SQL 脚本中的所有内容装载到 SQL 缓冲区中。示例如下：

```
SQL> get c:\a.sql
1* SELECT * FROM dept WHERE deptno=10
```

### (3) START 和@

START 和@命令用于运行 SQL 脚本文件。注意，当运行 SQL 脚本文件时，应该指定文件路径。示例如下：

### (3) SQL> @c:\a

```
DEPTNO DNAME          LOC
-----
10 ACCOUNTING      NEW YORK
```

### (4) @@

该命令与@命令类似，也可以运行脚本文件，但主要作用是在脚本文件中嵌套调用其它的脚本文件。当使用该命令嵌套脚本文件时，可在调用文件所在目录下查找相应文件名。

### (5) EDIT

该命令不仅可用于编辑 SQL 缓冲区内容，也可以用于编辑 SQL 脚本文件。当运行该命令时，会启动默认的系统编辑器来编辑 SQL 脚本。运行方法为“SQL> ed c:\a.sql”。

### (6) SPOOL

该命令用于将 SQL\*Plus 屏幕内容存放到文本文件中。执行该命令时，应首先建立假脱机文件，并将随后 SQL\*Plus 屏幕的所有内容全部存放到该文件中，最后使用 SPOOL OFF 命令关闭假脱机文件。示例如下：

```
SQL> spool c:\a.txt
SQL> SELECT * FROM dept WHERE deptno=10;
DEPTNO DNAME          LOC
-----
```

```
10      ACCOUNTING    NEW YORK
SQL> spool off
```

## 5. 格式命令

SQL\*Plus 不仅可以用于执行 SQL 语句、PL/SQL 块，而且还可以根据 SELECT 结果生成报表。使用 SQL\*Plus 的格式命令可以控制报表的显示格式，例如使用 COLUMN 命令可以控制列的显示格式；使用 TTITLE 命令可以指定页标题；使用 BTITLE 命令可以指定页脚注。本节将介绍这些格式命令。

### (1) COL[UMN]

该命令用于控制列的显示格式。COLUMN 命令包含有四个选项，其中 CLEAR 选项用于清除已定义列的显示格式；HEADING 选项用于指定列的显示标题；JUSTIFY 选项用于指定列标题的对齐格式（LEFT, CENTER, RIGHT）；FORMAT 选项用于指定列的显示格式。其中格式模型包含以下一些元素：

- An: 设置 CHAR、VARCHAR2 类型列的显示宽度；
- 9: 在 NUMBER 类型列上禁止显示前导 0；
- 0: 在 NUMBER 类型列上强制显示前导 0；
- \$: 在 NUMBER 类型列前显示美元符号；
- L: 在 NUMBER 类型列前显示本地货币符号；
- .: 指定 NUMBER 类型列的小数点位置；
- ,: 指定 NUMBER 类型列的千分隔符；

#### 示例一：使用 COLUMN 设置列显示格式

```
SQL> col ename heading 'name' format a10
SQL> col sal heading 'sal' format L99999.99
SQL> SELECT ename,sal,hiredate FROM emp WHERE empno=7788;
      name          sal        HIREDATE
      -----        -----
SCOTT           RMB3000.00     03-12 月-81
```

#### 示例二：使用 COLUMN 清除列显示格式：

```
SQL> col ename clear
SQL> col sal clear
SQL> SELECT ename,sal,hiredate FROM emp WHERE empno=7788;
      ENAME         SAL HIREDATE
      -----
SCOTT           3000 03-12 月-81
```

### (2) TTITLE

该命令用于指定页标题，页标题会自动显示在页的中央。如果页标题由多个词组成，则用单引号引住。如果要将页标题分布在多行显示，则用“|”分开不同单词。如果不希望显示页标题，则使用“TTITLE OFF”命令，禁止显示。示例如下：

```
SQL> set linesize 40
SQL> tttitle 'employee report'
SQL> SELECT ename,sal,hiredate FROM emp WHERE empno=7788;
星期四 5月 15          第 1
                           employee report
```

| ENAME | SAL  | HIREDATE   |
|-------|------|------------|
| SCOTT | 3000 | 03-12 月-81 |

### (3) BTITLE

该命令用于指定页脚注，页脚注会自动显示在页的中央。如果页脚注由多个词组成，则用单引号引住。如果要将页脚注分布在多行显示，则用“|”分开不同单词。如果不希望显示页脚注，则使用“BTITLE OFF”命令，禁止显示。示例如下：

```
SQL> bttitle 'page end'
SQL> SELECT ename,sal,hiredate FROM emp WHERE empno=7788;
ENAME      SAL HIREDATE
-----  -----
SCOTT      3000    03-12 月~81
...
          page end
```

### (4) BREAK

该命令用于禁止显示重复行，并将显示结果分隔为几个部分，以表现更友好的显示结果，通常应该在 ORDER BY 的排序列上使用该命令。示例如下：

```
SQL> break on deptno skip 1
SQL> set pagesize 40
SQL> SELECT deptno,ename,sal FROM emp ORDER BY deptno;
DEPTNO ENAME           SAL
-----  -----
10  CLARK            4200
     KING             7500
     MILLER           3430
20  SMITH            968
     ADAMS            1331
     FORD              3630
     SCOTT            3630
     JONES            3599.75
```

## 6. 交互式命令

如果经常要执行某些 SQL 语句和 SQL\*Plus 命令，可以将这些语句和命令存放到 SQL 脚本中。通过使用 SQL 脚本，一方面可以降低命令输入量，另一方面可以避免用户的输入错误。为了使得 SQL 脚本可以根据不同输入获得不同结果，需要在 SQL 脚本中包含交互式命令。通过使用交互式命令，可以在 SQL\*Plus 中定义变量，并且在运行 SQL 脚本时可以为这些变量动态输入数据。下面介绍 SQL\*Plus 的交互式命令，以及引用变量所使用的标号。

### (1) &

引用替代变量（Substitution Variable）时，必须要带有该标号。如果替代变量已经定义，则会直接使用其数据；如果替代变量没有定义，则会临时定义替代变量（该替代变量只在当前语句中起作用），并需要为其输入数据。注意，如果替代变量为数字列提供数据，则可以直接引用；如果替代变量为字符类型列或日期类型列提供数据，则必须要用单引号引住。示例如下：

```

SQL> SELECT ename,sal FROM emp WHERE deptno=&no AND job='&job';
输入 no 的值: 20
输入 job 的值: CLERK
原值 1: SELECT ename,sal FROM emp WHERE deptno=&no AND job='&job'
新值 1: SELECT ename,sal FROM emp WHERE deptno=20 AND job='CLERK'
ENAME          SAL
-----
SMITH        2000
ADAMS        1100

```

### (2) &&

该标号类似于单个&标号。但需要注意，&标号所定义的替代变量只在当前语句中起作用；而&&标号所定义的变量会在当前 SQL\*Plus 环境中一直生效。示例如下：

|                                                                                                                                                                                                                                                                                            |                                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| <pre> SQL&gt; SELECT ename,sal FROM emp WHERE deptno=&amp;no AND job='&amp;&amp;job'; 输入 no 的值: 20 输入 job 的值: CLERK ... SQL&gt; SELECT ename,sal FROM emp WHERE deptno=&amp;no 原值 1: SELECT ename,sal FROM emp WHERE deptno=&amp;no 新值 1: SELECT ename,sal FROM emp WHERE deptno=20 </pre> | <p>定义了 no 变量<br/>直接引用 no 变量</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|

如例所示，当第一次引用 no 变量时，使用&&标号需要为其输入数据；当第二次引用 no 变量时，使用&标号直接引用其原有值，而不需要输入数据。

### (3) DEFINE

该命令用于定义类型为 CHAR 的替代变量，而且该命令所定义的替代变量只在当前 SQL\*Plus 环境中起作用。当使用该命令定义变量时，如果变量值包含空格或区分大小写，则用引号引住。另外，使用“DEFINE 变量名”可以检查变量是否已经定义。示例如下：

```

SQL> SET VERIFY OFF
SQL> DEFINE title=CLERK
SQL> SELECT ename,sal FROM emp WHERE job='&title';
ENAME          SAL
-----
SMITH        968
ADAMS        1331
JAMES        1045
MILLER       3430

```

### (4) ACCEPT

该命令可以用于定义 CHAR, NUMBER 和 DATE 类型的替代变量。与 DEFINE 命令相比，ACCEPT 命令更加灵活。当使用该命令定义替代变量时，还可以指定变量输入提示、变量输入格式、隐藏输入内容。

#### 示例一：指定变量输入提示

```

SQL> ACCEPT title PROMPT '请输入岗位：'
请输入岗位: CLERK

```

```
SQL> SELECT ename,sal FROM emp WHERE job='&title';
ENAME          SAL
-----
SMITH          968
ADAMS          1331
JAMES          1045
MILLER         3430
```

### 示例二：隐藏用户输入

```
SQL> accept pwd hide
```

```
SQL> conn scott/&pwd@test
已连接。
```

### (5) UNDEFINE

该命令用于清除替代变量的定义。示例如下：

```
SQL> undefine pwd
SQL> conn scott/&pwd@test
输入 pwd 的值: tiger
已连接。
```

### (6) PROMPT 和 PAUSE

PROMPT 命令用于输出提示信息，而 PAUSE 命令则用于暂停脚本执行。在 SQL 脚本中结合使用这两条命令，可以控制 SQL 脚本的暂停和执行。假定在 a.sql 脚本中包含以下命令：

```
PROMPT '按<Return>键继续'
PAUSE
```

当运行该 SQL 脚本时，会暂停执行，示例如下：

```
SQL> @c:\a
'按<Return>键继续'
```

...

### (7) VARIABLE

该命令用于在 SQL\*Plus 中定义绑定变量。当在 SQL 语句或 PL/SQL 块中引用绑定变量时，必须要在绑定变量前加冒号（:）；当直接给绑定变量赋值时，需要使用 EXECUTE 命令（类似于调用存储过程）。示例如下：

```
SQL> VAR no NUMBER
SQL> exec :no:=7788
PL/SQL 过程已成功完成。
SQL> SELECT ename FROM emp WHERE empno=:no;
ENAME
-----
SCOTT
```

### (8) PRINT

该命令用于输出绑定变量结果，示例如下：

```
SQL> PRINT no
```

```
NO
```

```
-----
```

```
7788
```

## 7. 显示和设置环境变量

使用 SQL\*Plus 的环境变量可以控制其运行环境，例如设置行显示宽度、设置每页显示的行数、设置自动提交标记、设置自动跟踪等等。使用 SHOW 命令可以显示当前 SQL\*Plus 的环境变量设置；使用 SET 命令可以修改当前 SQL\*Plus 的环境变量设置。下面介绍常用的 SQL\*Plus 环境变量。

### (1) 显示所有环境变量

为了显示 SQL\*Plus 的所有环境变量，必须要使用 SHOW ALL 命令。示例如下：

```
SQL> show all
appinfo 为 OFF 并且已设置为"SQL*Plus"
arraysize 15
autocommit OFF
autoprint OFF
autorecovery OFF
...
```

### (2) ARRAYSIZE

该环境变量用于指定数组提取尺寸，其默认值为 15。该值越大，网络开销将会越低，但占用内存会增加。假定使用默认值，如果查询返回行数为 50 行，则需要通过网络传送 4 次数据；如果设置为 25，则网络传送次数只有两次。示例如下：

```
SQL> show arraysize
arraysize 15
SQL> set arraysize 25
```

### (3) AUTOCOMMIT

该环境变量用于设置是否自动提交 DML 语句，其默认值为 OFF（表示禁止自动提交）。当设置为 ON 时，每次执行 DML 语句都会自动提交。示例如下：

```
SQL> show autocommit
autocommit OFF
SQL> set autocommit on
```

### (4) COLSEP

该环境变量用于设置列之间的分隔符，默认分隔符为空格。如果要使用其它分隔符，则使用 SET 命令进行设置。示例如下：

```
SQL> set colsep |
SQL> SELECT ename,sal FROM emp WHERE empno=7788;
ENAME	SAL
SCOTT     |      3000
```

### (5) FEEDBACK

该环境变量用于指定显示反馈行数信息的最低行数，其默认值为 6。如果要禁止显示行数反馈信息，则将 FEEDBACK 设置为 OFF。假设只要有查询结果就希望返回行数，那么可以将该环境变量设置为 1。示例如下：

```
SQL> set feedback 1
SQL> select ename,sal from emp where empno=7788;
ENAME      SAL
```

```
-----  
SCOTT          3000  
已选择 1 行。
```

**(6) HEADING**

该环境变量用于设置是否显示列标题，其默认值为 ON。如果不显示列标题，则设置为 OFF。示例如下：

```
SQL> set heading off  
SQL> select ename,sal from emp where empno=7788;  
SCOTT          3000  
已选择 1 行。
```

**(7) LINESIZE**

该环境变量用于设置行宽度，默认值为 80。在默认情况下，如果数据长度超过 80 个字符，那么在 SQL\*Plus 中会折行显示数据结果。要在一行中显示全部数据，应该设置更大的值。示例如下：

```
SQL> set linesize 120  
SQL> select * from emp where empno=7788;  
EMPNO    ENAME      JOB         MGR      HIREDATE ...  
----- ...  
7788     SCOTT      ANALYST    7566    03-12 月-81 ...
```

**(8) LONG**

该环境变量用于设置 LONG 和 LOB 类型列的显示长度。其默认值为 80，也就是说当查询 LONG 或 LOB 列时，只会显示该列的前 80 个字符。要显示更多字符，应该设置更大的值。示例如下：

```
SQL> show long  
long 80  
SQL> set long 300
```

**(9) PAGESIZE**

该环境变量用于设置每页所显示的行数，其默认值为 14。要显示更多行，则应该设置更大的值。示例如下：

```
SQL> show pagesize  
pagesize 14  
SQL> set pagesize 40
```

**(10) SERVEROUTPUT**

该环境变量用于控制服务器输出，其默认值为 OFF，表示禁止服务器输出。在默认情况下，当调用 DBMS\_OUTPUT 包时，不会在 SQL\*Plus 屏幕上显示输出结果。在调用 DBMS\_OUTPUT 包时，为了在屏幕上输出结果，必须要将 SERVEROUTPUT 设置为 ON。示例如下：

```
SQL> set serveroutput on  
SQL> exec dbms_output.put_line('hello')  
hello  
PL/SQL 过程已成功完成。
```

**(11) TERMOUT**

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off
SQL> @c:\a
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on
SQL> SELECT count(*) FROM sales;
COUNT(*)
-----
1016271
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```

该环境变量用于控制 SQL 脚本的输出，其默认值为 ON。当使用默认值时，如果 SQL 脚本有输出结果，则会在屏幕上输出显示结果；如果设置为 OFF，则不会在屏幕上输出 SQL 脚本的任何结果。示例如下：

```
SQL> set termout off  
SQL> @c:\a  
SQL>
```

#### (12) TIME

该环境变量用于设置在 SQL 提示符前是否显示系统时间，默认值为 OFF，表示禁止显示系统时间。如果设置为 ON，则在 SQL 提示符前会显示系统时间。示例如下：

```
SQL> set time on  
10:31:18 SQL>
```

#### (13) TIMING

该环境变量用于设置是否要显示 SQL 语句执行时间，默认值为 OFF，表示不会显示 SQL 语句执行时间。如果设置为 ON，则会显示 SQL 语句执行时间。示例如下：

```
SQL> set timing on  
SQL> SELECT count(*) FROM sales;  
COUNT(*)  
-----  
1016271  
已用时间： 00: 00: 04.09
```