第一篇　PRO＊C 程序设计

第一章　PRO＊C 程序概述

§1.1　什么叫 PRO＊C 程序

在 ORACLE 数据库管理系统中，有三种访问数据库的方法：

(1)用 SQL＊PLUS，它用 SQL 命令以交互的方式访问数据库。

(2)用第四代语言应用开发工具开发的应用程序访问数据库，这些工具有 SQL＊Forms、SQL＊ReportWriter、SQL＊Menu 和 ORACLE＊FORMS 等。

(3)利用在第三代高级语言内嵌入的 SQL 语句，或 ORACLE 库函数调用来访问数据库。

ORACLE 支持在六种高级语言中嵌入 SQL 语句，它们是 C，FORTRAN，PASCAL，COBOL，PL/I 和 Ada。我们把这些语言统称为宿主语言，用它们开发的应用程序叫 PRO 程序。如果宿主语言是 C，则相应的程序叫 PRO＊C，如果宿主语言是 FORTRAN，则相应的程序叫 PRO＊FORTRAN 程序，.....。本篇重点介绍 PRO＊C 程序，但它的基本思想和方法也适于 PRO＊FORTRAN、PRO＊COBOL、PRO＊PASCAL、和 PRO＊PL/I 等。而 PRO＊Ada 有些特殊，应参考 PRO＊Ada 的有关手册。

在 PRO＊C 程序中可以嵌入 SQL 语句，利用这些 SQL 语句可以完成动态地建立、修改和删除数据库中的表，也可以查询、插入、修改和删除数据库表中的行，还可以实现事务的提交和回滚。

在 PRO＊C 程序中还可以嵌入 PL/SQL 块，以改进应用程序的性能，特别是在网络环境下，可以减少网络传输和处理的总开销。

利用第三代高级语言内嵌入 SQL 语句来开发应用程序有以下三点好处：

(1) 它把过程化语言和非过程化语言相结合，形成一种更强有力的开发工具。利用它可以开发出满足各种复杂要求的应用程序，还可以引用窗口技术和鼠标技术等。

(2) 可以使开发的应用程序具有管理系统资源使用(如内存分配)、SQL 语句执行和指示器等能力。

(3) 提高了应用程序的执行速度，因为它把 SQL 语句翻译成相应的 ORACLE 库函数调用。

§1.2　ORACLE 预编译程序

ORACLE 预编译程序是一种强有力的应用程序开发工具，它输入 PRO 源程序，经处理

输出相应高级语言的源程序。例如，ORACLE 的 PRO * C 预编辑程序，它输入 PRO * C 源程序，输出 C 的源程序。

ORACLE 提供了六种预编译程序(统称 PRO 系列)，它们分别支持以下六种高级语言：

C

Pascal

FORTRAN

COBOL

PL/I

Ada

源程序中嵌入的 SQL 语句经预编译程序处理后，被翻译成相应 ORACLE 库函数的调用。

ORACLE 预编译程序的功能和特点如下：

1. 能用六种通用的高级程序设计语言中的任何一种来编写应用程序。
2. 嵌入的 SQL 语句完全遵循 ANSI 标准。
3. 可采用动态 SQL 技术。
4. 能开发出满足各种需求的应用程序。
5. 自动实现内部和外部数据类型转换。
6. 可嵌入 PL/SQL 块。
7. 能在命令行和程序行上指定预编译可选项。
8. 能用数据类型等价来控制 ORACLE 解释输入数据和格式化输出数据的方式。
9. 能分别预编译。
10. 能检查嵌入的 SQL 语句或 PL/SQL 块的语法和语义。
11. 可利用 SQL * Net 并行存取多个结点上的 ORACLE 数据。
12. 可使用数组 SQL 变量。
13. 能进行条件预编译。
14. 可用高级语言编写 SQL * Forms 的用户出口。
15. 能用 SQLCA 和 ORACA 进行错误诊断。

§1.3 PRO * C 程序的组成及举例

1.3.1 PRO * C 程序举例

为了对 PRO * C 程序有一个感性认识，我们首先来浏览一个 PRO * C 程序实例：

例 1.1 简单查询

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *    功能：
 *    该程序先与 ORACLE 相连接，然后提示用户输入一个职员号，
 *    最后根据该职员号查询数据库，查询出该职员的名字、工资、附加工资，
 *    并显示结果。
```

```
*   当输入职员号为0时，程序结束。
*********************************************************/
/******** 外部说明部 ********/
    #include <stdio.h>
    /*        说明段,说明SQL变量        */
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR username[20];
        VARCHAR password[20];
    EXEC SQL END DECLARE SECTION;
    /*        SQL通讯区说明        */
    EXEC SQL INCLUDE sqlca;
    /*        C函数说明,错误处理        */
    void sqlerror();
/********** 程序体 **********/
    main()
    {
      /******内部说明部 ********/
      /* 内部说明段,说明局部SQL变量 */
      EXEC SQL BEGIN DECLARE SECTION;
        int    emp_number;
        VARCHAR emp_name[15];
        float salary;
        float commission;
      EXEC SQL END DECLARE SECTION;

      /*        C变量的说明        */
      int total_number;

      /* 登录到ORACLE */
      strcpy(username.arr, "SCOTT");          /* 用户名 */
      username.len=strlen(username.arr);
      strcpy(password.arr, "TIGER");          /* 口令 */
      password.len=strlen(password.arr);
      EXEC SQL WHENEVER SQLERROR DO sqlerror();
                        /*错误处理说明语句*/
      EXEC SQL CONNECT :username IDENTIFIED BY :password;
                        /* 连接到ORACLE数据库管理系统 */
      printf("\nConnected to ORACLE as user: %s\n",username.arr);
```

```
    /* 循环查询职员信息 */
    total_number = 0;
    while (1)
{   emp_number = 0;
    printf("\nEnter employee number (0 to quit): ");
    scanf("%d", &emp_number);
    if(emp_number == 0) break;
    /* 查询语句 */
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    EXEC SQL SELECT ENAME, SAL, COMM
            INTO :emp_name, :salary, :commission
            FROM EMP
            WHERE EMPNO = :emp_number;
    /* 输出查询结果 */
    printf("\n\nEmployee\tSalary\tCommission\n");
    printf("--------\t------\t----------\n");
    emp_name.arr[emp_name.len] = '\0';
    printf("%-8s\t%6.2f\t%6.2f\n",
            emp_name.arr, salary, commission);
    total_number = total_number + 1;
    continue;
notfound:
    printf("\nNot a valid employee number - try again.\n");
}
printf("\n\nTotal number queried was: %d\n", total_number);
printf("\nHave a good day.\n");
/* 结束处理,退出 ORACLE 系统 /*
EXEC SQL COMMIT WORK RELEASE;
exit(0);
}
  /* 错误处理 */
  void sqlerror()
  {
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      printf("\nORALCE error detected:\n");
      /* 打印错误文本 */
      printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
      /* 回滚事务 */
      EXEC SQL ROLLBACK RELEASE;
```

```
        exit(1);
    }
```

从上例我们可以看到：该实例程序是由外部说明部和程序体两部分所组成。

外部说明部主要说明程序中所引用的外部变量及函数等，它由说明段、通讯区说明和C的有关说明等所组成。

说明段是由SQL语句

EXEC SQL BEGIN DECLARE SECTION;

开始，由

EXEC SQL END DECLARE SECTION;

结束。其间包含若干类型说明语句，用以说明SQL变量的类型。

语句

EXEC SQL INCLUDE Sqlca;

说明SQL通讯区，用以记录执行每一个嵌入SQL语句的状态信息。

程序体是由若干函数(简单的可以仅由一个主函数)所组成，其中有一个主函数。一般来说，每一个函数又包含如下成分：

· 内部说明部：说明函数内所用的局部变量，它的组成类似外部说明部，也是由说明段、局部通讯区说明及C的局部变量说明。

· C语言的可执行语句。

· 嵌入式SQL语句或PL/SQL块：用以实现对数据库中数据的操作。

在主函数内应包含一个登录语句，如

EXEC SQL CONNECT : username IDENTIFIED BY : Password; 以实现与数据库系统的连接。

在函数内除包含一般的SQL语句(SELECT, INSERT. . . . .)外，还应包含错误处理的说明语句，如

EXEC SQL WHENEVER. . . . . ;以便指出在执行SQL语句时，如果发生错误，应如何处理。

上例展示了具有一个源文件的程序之组成，对于具有多个源文件的程序，其每个源文件的组成与上述类似。

1.3.2 PRO * C程序的一般组成

从上述的例子我们可以得出这样一个结论：PRO * C程序实际是内嵌有SQL语句或PL/SQL块的C程序，因此它的组很类似C程序。但因为它内嵌有SQL语句或PL/SQL块，所以它还含有与之不同的成分。二者构成的差别可用图1-1和图1-2表示。

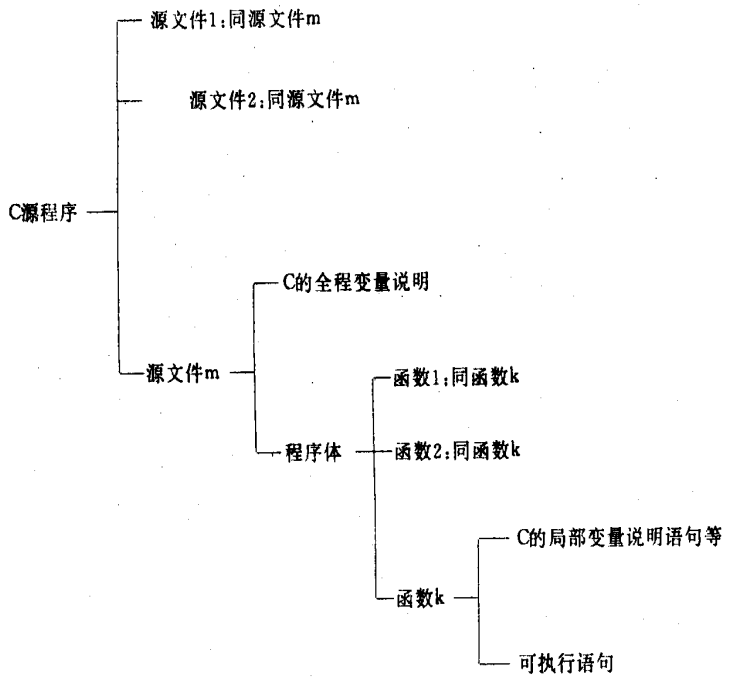


图1-1　C源程序的组成

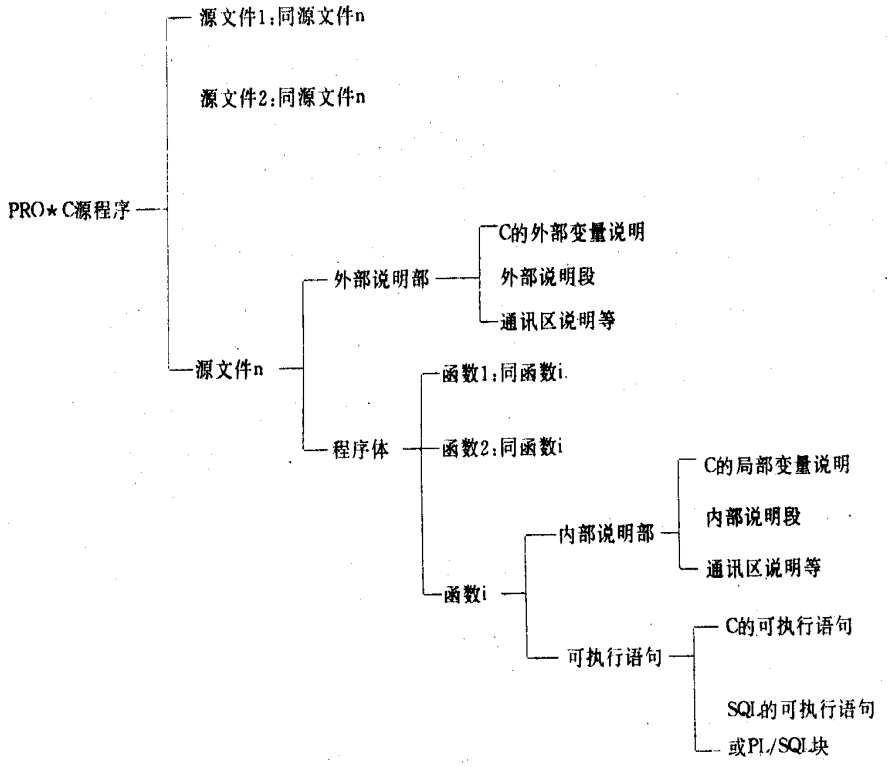


图1-2　PRO*C源程序的组成

注:C或PRO*C的一个源文件也可以只包含说明部分,而不包含程序体。

§1.4 开发和运行一个 PRO＊C 应用程序的基本步骤

PRO＊C 应用程序的开发和运行的基本步骤类似于 C 程序，唯一不同之处是在进行 C 编译之前需先进行预编译。图 1-3 说明应用程序的开发步骤。图 1—4 表示 PRO＊C 程序的处理过程。

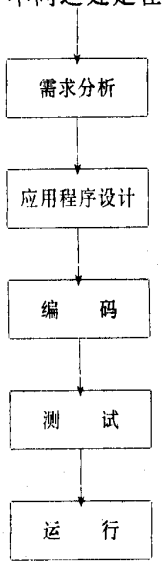


图1-3 应用程序开发步骤

§1.5 PRO＊C 程序书写格式的几点说明

为了使书写的程序可读性强，在编码时应遵循软件工程中所规定的编码规则。前面 1.3 节的实例程序也同时展示出一个 PRO＊C 程序的典范书写格式，供读者参考。读者应特别注意以下几点：

1. 要采用锯齿形的书写格式，

2. 要正确使用注释，

在 SQL 语句中，凡能放置空格的地方(除 EXEC SQL 关键字之间外)都可放置 C 语言风格的注释(/＊…＊/)。除此之外，还可在 SQL 语句行的末尾放置 ANSI 风格的注释，如下例所示：

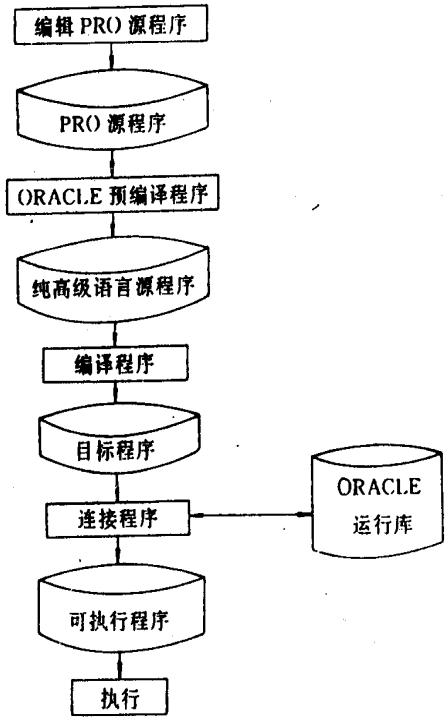


图 1-4 PRO＊C 程序的处理过程

```
EXEC SQL SELECT ENAME, SAL
    INTO :emp_name, :salary      -- output host variable
    FROM EMP
    WHERE DEPTNO=:dept_number;
```

但注释不能嵌套。

3. 嵌入式 SQL 语句或 PL/SQL 块中的关键词应大写。如 SELECT,……。

§1.6 参考资料

本篇参考了如下一些资料：

1. Programmer's Guide to the ORACLE precompilers, V1.5, part NO. 5315－15－1292

2. Pro＊C Supplement to the ORACLE precompilers Guide, V1.5, part NO. 5452－15－1292

3. ORACLE Server SQL Language Reference Manual, Part NO. 778－70

4. <<ORACLE 应用系统开发工具>> 孙宏昌、刘金亭、何毅华著

第二章　PRO＊C 程序设计的基础知识

§2.1　说明段

2.1.1　SQL 变量

在 PRO＊C 程序中含有 C 和 SQL 两种语言的语句，为了解决它们之间的信息传递问题，特引入了一种变量，称为 SQL 变量。

SQL 变量可在 SQL 语句中引用，也可在 C 的语句中引用，而 C 的变量只能在 C 的语句中引用。同 C 语言中的变量一样，SQL 变量也必需是先说明后引用，否则就会出现如下的错误信息：

Undeclared host variable "X" at line N in file "F"

其中 X 是 SQL 变量名，N 是 X 所在源程序行号，F 是该编译单位的源文件名。

2.1.2　说明段

SQL 变量必须在说明段说明。说明段是由语句

EXEC SQL BEGIN DECLARE SECTION;

开始，由语句

EXEC SQL END DECLARE SECTION;

结束。在这两个语句之间只允许包含下列语句：

(1) SQL 变量的类型说明语句

(2) EXEC SQL INCLUDE 语句

(3) EXEC SQL VAR 语句

(4) EXEC SQL TYPE 语句

但不允许包含说明为结构类型的 SQL 变量。

在 PRO＊C 程序中允许定义局部的和全程的说明段，在每个预编译单位内允许有多个说明段，但必须至少有一个全程说明段。说明段的任务是说明变量的数据类型。

例如：

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR name[15];
    VARCHAR password[10];
    int emp_no;
    float salary;
    char dept_name[50];
```

```
    short ind_sal;
EXEC SQL END DECLARE SECTION;
```

§2.2 数据类型和转换

ORACLE 支持内部和外部两类数据类型，内部数据类型描述 ORACLE 按什麼格式把数据存储在数据库的表列中，也用它来描述数据库的伪列。外部数据类型描述如何把数据存储在 SQL 变量中。

在预编译时，说明段中的每一个 SQL 变量都与一个外部类型码相关联。在运行时，把每一个 SQL 变量的外部类型码传递给 ORACLE，以实现内部和外部数据类型之间的自动转换，这种转换称之为缺省数据类型转换。也可使用动态方法或数据类型等价来替代这种缺省数据类型转换。

2.2.1 内部数据类型

表 2-1 中列出了 ORACLE 的所有内部数据类型。数据库中的表列或伪列使用这些数据类型。

表 2-1 ORACLE 内部数据类型

名 字	代 码	描 述
VARCHAR2	1	变长串，(=2000 字节。
NUMBER	2	定点或浮点数
LONG	8	变长串，2147483647 字节
ROWID	11	16 进制数串
DATE	12	定长日期/时间值，7 字节
RAW	23	定长二进制数据，255 字节
LONGRAW	24	变长二进制数据，2147483647 字节
CHAR	96	定长串，255 字节
MLSLABEL	105	定长二进制标号，5 字节

2.2.2 SQL 伪列和函数

伪列不是一个表中的实际列，但其处理类似于实际列。例如，它们的值也必须从一个表中选择。在有些应用中，使用从伪表中选择伪列的值是方便的。

SQL 识别表 2-2 所列出的伪列：

表 2-2 SQL 所识别的伪列

伪 列	内部数据类型	代 码	描 述
NEXTVAL	NUMBE	2	它为指定列返回一个序号，用它来产生事务处理的唯一序号。
CURRVAL	NUMBER	2	用它返回指定序列的当前序号。引用前应先引用 NEXTVAL，以产生一个序号。
ROWNUM	NUMBER	2	用它返回指定序列的序号，用于表中选择行。
LEVEL	NUMBER	2	用该伪列返回树结构中一节点的层次号。
ROWID	ROWID	11	返回 16 进制的行地址。

SQL 识别表 2-3 所列出的函数：

表 2-3　SQL 所识别的函数

函　数	对应的内部数据类型	代　码	描　述
USER	VARCHAR2	1	返回 ORACLE 当前用户名
UID	NUMBER	2	返回赋给 ORACLE 用户的唯一 ID 号
SYSDATE	DATE	12	返回当前日期和时间

可以在 SELECT、INSERT、DELETE 和 UPDATE 中引用伪列和函数。例如下面的语句是使用 SYSDATE 计算某职员被雇用的月数：

```
EXEC SQL SELECT MONTHS-BETWEEN(SYSDATE, HIREDATE)
    INTO :months_of_service
    FROM EMP
    WHERE EMPNO=:emp_number;
```

2.2.3　外部数据类型

外部数据类型包括全部的内部数据类型和普通宿主语言中所提供的几个数据类型。表 2-4 列出了全部的外部数据类型。

表 2-4　外部数据类型

名　字	代　码	描　述
VARCHAR2	1	用以存储变长字符串，最大长度 2000 字节。在输入时，ORACLE 删去尾部空格后再存入表列中。
NUMBER	2	二进制数，用来存储定点数或浮点数。可指定精度(1～38)和定标(－84～127)。
INTGER	3	有符号整数，用 2 或 4 字节的二进制数表示。
FLOAT	4	浮点数，通常要求 4 或 8 字节存储。
STRING	5	是以 Null 结尾的字符串，其它类似 VARCHAR2。
VARNUM	6	变长二进制数，它类似于 NUMBER，唯一区别是第一字节存储该值长度 。
LONG	8	变长字符串，最大长度 2G 字节，其他类似 VARCHAR2。
VARCHAR	9	变长字符串，它含 2 字节的长度字段和小于 65533 字节的串字段，对于 VARCHAR 数组元素，串字段最大长度为 65530。
ROWID	11	二进制值，标识表中的行。占用 13 字节。
DATE	12	定长日期/时间值，占用 7 字节，自左至右存放世纪、年、月、日、小时、分和秒。注
VARRAW	15	变长二进制数据，存储二进制数据或字符串。它由 2 字节长度字段和 65533 字节的串字段组成。
RAW	23	定长二进制数据，存储二进制串，最大长度 255 字节。
LONGRAW	24	变长二进制数据，最大长度 2G 字节，其它同 RAW。
UNSIGNED	68	无符号整数，是二或四字节的二进制数。必须指定长度。
LONGVARCHAR	94	变长字符串，也是由长度和串字段所组成，长度占 4 字，串字段(2G－5)字节。
LONGVARRAW	95	变长二进制数据，其它类似于 LONGVARCHAR。
CHAR	96	定长字符串，最长 255 字节。
CHARZ	97	C 中定长以 null 结尾的字符串，最长 255 字节。
MLSLABEL	106	变长二进制数据。

注：日期的表示格式如表 2-5。

表 2-5 DATE 格式

字 节	1	2	3	4	5	6	7
含 义	世纪	年	月	日	小时	分	秒
例 子							
06-DEC-1990	119	190	12	6	1	1	1

2.2.4 SQL 变量的数据类型

在说明段能为 SQL 变量指定的数据类型如表 2-6 所示。

表 2-6 SQL 变量的数据类型

C 的数据类型	描 述
char	单字符
char[n]	n 个字符数组
int	整数
short	短整数
long	长整数
float	单精度浮点数
double	双精度浮点数
VARCHAR[n]	变长字符串

这些数据类型实际上就是 C 语言的数据类型，其中 VARCHAR 可视为 C 数据类型的扩充。

2.2.5 数据类型转换

在预编译时，对于每一个 SQL 变量都分配一个外部数据类型。例如，把 VARCHAR2 外部数据类型分配给字符型 SQL 变量。这样预编译后，每一个变量都与一个外部数据类型码相关联。在运行时，SQL 语句中的每一个 SQL 变量的外部类型码被传递给 ORACLE。ORACLE 使用这些代码实现内外数据类型之间的转换。

在查询时，在把选择的列(或伪列)赋给输出 SQL 变量之前，如果表列和变量的类型不一致时，ORACLE 就把表列的数据类型转换成 SQL 变量的外部数据类型。同样，在把 SQL 变量的值存入数据库表列(或二者进行比较)之前，如果二者数据类型不一致时，ORACLE 就把 SQL 变量的外部数据类型转换成目标列的内部数据类型。

但在进行转换时，内部数据类型与外部数据类型必须相兼容，否则就无法实现。因此，程序的设计者要确保内外数据类型之间是可转换的。图 2-1 表示支持的数据类型转换。

EXTERNAL \ INTERNAL	1 VARCHAR2	2 NUMBER	8 LONG	11 ROWID	12 DATE	23 RAW	24 LONG RAW	96 CHAR
1 VARCHAR2	↵	↵	↵	↵ 1	↵ 2	↵ 3	↑ 3	↵
2 NUMBER	↵ 4	↵	↑					↵ 4
3 INTEGER	↵ 4	↵	↑					↵ 4
4 FLOAT	↵ 4	↵	↑					↵ 4
5 STRING	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3.5	↵
6 VARNUM	↵ 4	↵	↑					↵ 4
7 DECIMAL	↵ 4	↵	↑					↵ 4
8 LONG	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3.5	↵
9 VARCHAR	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3.5	↵
11 ROWID	↵		↵	↵				↵
12 DATE	↵		↑		↵			↵
15 VARRAW	↵ 6		↑ 5 ↵ 6			↵	↵	↵ 6
23 RAW	↵ 6		↑ 5 ↵ 6			↵	↵	↵ 6
24 LONG RAW	↵ 6		↑ 5 ↵ 6			↵	↵	↵ 6
68 UNSIGNED	↵ 4	↵	↑					↵ 4
91 DISPLAY	↵ 4	↵	↑					↵ 4
94 LONGVARCHAR	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3.5	↵
95 LONG VARRAW	↵ 6		↑ 5 ↵ 6			↵	↵	↵ 6
96 CHAR	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3	↵
97 CHARZ	↵	↵	↵	↵ 1	↵ 2	↵ 3	↵ 3	↵

注意：1. 对输入，宿主串必须是'BBBBBBBB. RRRR. FFFF'格式。
对输出，以相同格式返回值。↵ 表示只输入。

2. 对输入，宿主串必须是缺少 DATE 字符格式。对输出，以相同格式返回列值。↑ 表示只输出。

3. 对输入，宿主串必须是 16 进制格式，输出时以相同格式返回。

4. 对输出列值必须表示一个有效的数。↵ 表示输入或输出。

5. 长度必须小于或等于 2000。

6. 对输入，列值必须以 16 进制格式存放。对输出，列值必须是 16 进制格式。

图 2-1　支持的数据类型转换

§2.3 数据类型等价

数据类型等价向用户提供一种控制 ORACLE 解释输入数据和格式化输出数据的方法。可把 C 语言的数据类型与 ORACLE 的外部类型等价，也可把用户定义的类型与 ORACLE 外部类型等价。对这两种情况，可指定除 NUMBER(不需要，可用 VARNUM 替代)外的任何外部数据类型。

2.3.1 SQL 变量等价

按照缺省规定，PRO＊C 预编译程序把指定的外部数据类型分配给每一个 SQL 变量。如表 2-7 所示：

表 2-7　　外部数据类型与 SQL 变量类型的缺省对应

SQL 变量的数	外部数据类型	数据类型代码据类型
char,char[n],char＊	VARCHAR2	1
char,char[n],char＊	CHARZ	97(当 MODE＝ANSI 时)
int, int＊	INTEGER	3
short, short＊	INTEGER	3
long, long＊	INTEGER	3
float, float＊	FLOAT	4
double, double＊	FLOAT	4
VARCHAR[n]	VARCHAR	9

在说明段，可通过使用 VAR 语句来把 SQL 变量与外部数据类型等价，以替代这种缺省分配。VAR 语句的格式如下：

EXEC SQL VAR host_variable IS type_name[(legth)];

其中：

host_variable 是 SQL 变量，type_name 是外部数据类型，legth 是该外部数据类型的长度。

SQL 变量等价在以下几方面是有用的：

(1)在 C 程序中，字符串必须以 Null 终结，利用 SQL 变量等价，可保证从数据库表列 SELECT 或 FETCH 出的串以 Null 终结。例如，想把职员名从 EMP 表列中选择到字符数组中，然后传递给要求以 Null 终结串的函数。这时，只需把该宿主数组等价于外部数据类型 STRING 即可，不需再用 Null 显式地结束该串。如下例所示：

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    int emp_number;
    char emp_name[11];
    EXEC SQL VAR emp_name IS STRING(11);
EXEC SQL END DECLARE SECTION;
...
```

```
main()
{ …
    EXEC SQL SELECT ENAME INTO :emp_name
        FROM EMP
        WHERE EMPNO=:emp_number;
    printf("\n %s", emp_name);
    …
}
```

在此例中，EMP 表中 ENAME 的长度是 10 个字符，因此，应分配 emp_name 11 个字符，以便存放 Null 终结符。当把 ENAME 列的值 SELECT 到 emp_name 中时，程序接口就 Null 终结该值，而不需再编码给 emp_name 加 Null 字符的语句。

(2)当想让 ORACLE 的列只存储不解释的数据时，可用数据类型等价。例如，想在 LONG RAW 数据库表列存储浮点数数组时，可把数组与该外部数据类型等价。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
    …
    float salary[100];
    EXEC SQL VAR salary IS LONRAW(400);
EXEC SQL END DECLARE SECTION;
```

(3)可用数据类型等价来替代缺省分配。例如，如果把 DATE 列 SELECT 到字符型 SQL 变量中时，ORACLE 返回如下 9 字节长的标准格式串：

DD-MON-YY

但是，如果把 SQL 变量与 DATE 外部数据类型等价，则 ORACLE 返回如表 2－5 所示的 7 字节值。

2.3.2 用户定义类型等价

能把用户定义的数据类型与 ORACLE 外部数据类型等价。其方法是，先定义一个适合需要的新数据类型；然后，在说明段，用 TYPE 语句把新数据类型与外部数据类型等价。

用 TYPE 语句能把一个外部数据类型与用户定义的类型等价。其语句文法如下：

```
EXEC SQL TYPE user_type IS type_name[(length)][(REFERENCE)]
```

例如，假定用户需要一个保留图形字符串的变长串数据类型。这时可首先定义一个结构，该结构具有一个类型为 short 的长度字段和一个长度≤65533 字节的数据字段。然后用 typedef 定义一个基于该结构的新数据类型。最后再把新数据类型与 VARRAW 外部数据类型等价。如：

```
struct screen{
    short len;
    char buff[4000];
};
typedef struct screen graphics;
EXEC SQL BEGIN DECLARE SECTION;
```

```
        EXEC SQL TYPE graphics IS VARRAW(4000);
        graphics crt; /* 定义类型为 graphics 的宿主变量 */
        ...
    EXEC SQL END DECLARE SECTION;
```

预编译程序为长度字段分配 sizeof(short)个字节，为数据字段分配 4000 或更多的字节(取决于系统进行边界调整的需要)。

能显式地或隐式地把一个用户定义的类型说明为一个指针，然后在 EXEC SQL TYPE 中使用新类型。在这种情况下，必须在该语句的尾部使用 REFERENCE 子句。如下例所示：

```
    typedef unsigned char * my_raw;
    typedef char my_char[10];
    EXEC SQL BEGIN DECLARE SECTION;
        EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
        EXEC SQL TYPE my_char IS VARRAW(10) REFERENCE;
        my_raw graphics_buffer;
        my_char name
    EXEC SQL END DECLARE SECTION;
    ...
    graphics_buffer=(my_raw) malloc(4004);
```

这里为 graphics_buffer 分配了 4004 字节的内存单元，比该类型的长度多 4 字节。这是因为预编译程序要返回 2 字节(即 short 类型)长度和系统所要求的字边界调整

§2.4 SQL 变量的说明和引用

从数据结构角度来分，SQL 变量有简单变量，数组和指针。从功能来分，又有输入，输出和指示器变量。而输入和输出变量又叫宿主变量。

在 SELECT 或 FETCH 语句的 INTO 子句中所引用的变量叫输出 SQL 变量，因为 ORACLE 把从数据库表列中选择的值存放到这类变量中。而在其它 SQL 语句(INSERT，UPDATE……)中所包含的变量都叫输入 SQL 变量，ORACLE 把输入 SQL 变量中的数据输入给 ORACLE 存入数据库表列中，或与数据库表列中的数据进行比较。输入 SQL 变量也被用在 WHERE、HAVING 和 FOR 子句中。在 SQL 语句中，凡是允许出现值和表达式的地方，都允许出现输入 SQL 变量。

SQL 变量的名字可任意长，但只有开始 31 个字符有效。为了与 ANSI 和 ISO 标准兼容，要求名字以字母开始，且不包含连线和下划线字符，其长度必须≤18 个字符。SQL 变量的名字不允许与 ORACLE 保留字同名。

2.4.1 SQL 变量的说明和引用

(1)SQL 变量的说明

SQL 变量在引用之前必须先说明。所谓说明，就是在说明段，使用 C 的类型说明语句为每一个 SQL 变量指定一个 ORACLE 能支持的 C 数据类型。C 的类型必须与数据库列的数

据类型相兼容。在说明段能为SQL变量指定的数据类型如表2－6所示。

下面是SQL变量说明的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
        char name[15];                          (1)
        float salary=2500.00;                   (2)
        int var1,var2,var3;                     (3)
        float comm[50],var4[30];                (4)
        char username[50][15];                  (5)
        char work_date[10]="25－FEB－94";        (6)
EXEC SQL END DECLARE SECTION;
```

对简单C数据类型只能说明一维数组(如(1)和(4))，不允许说明二维或更高维数的数组。ORACLE把username[50][15]视为一维字符串数组。

对标准C数据类型，能使用一个说明语句同时说明几个SQL变量，如语句(3)和(4)等价于

```
EXEC SQL BEGIN DECLARE SECTION;
        int var1;
        int var2;
        int var3;
        float comm[50];
        float var4[30];
EXEC SQL END DECLARE SECTION;
```

可在说明段初始化SQL变量，如(2)和(6)，但不允许初始化数组，例如下面的说明是无效的：

```
EXEC SQL BEGIN DECLARE SECTION;
        int dept_number[3]={10,20,30};
EXEC SQL END DECLARE SECTION;
```

PRO＊C允许在说明段使用下列存储类型说明符来定义SQL变量：

- auto
- extern
- static

但不允许使用register存储类说明符。

SQL变量的存储类型决定了它被访问的场所和怎样被存储。下面的例子说明怎样在说明段编码存储类说明符。

```
main( )
{
    EXEC SQL BEGIN DECLARE SECTION;
        auto int m,n;           /* auto is the default */
        extern int sum;
```

```
        static char flag='Y';
    EXEC SQL END DECLARE SECTION;
    ...
}
```

允许在说明语句使用类型限制符 const 和 Volatile。一个 const 宿主变量必须有一个常数初始值；在程序中不能改变它的初始值。一个 Volatile 宿主变量有一个可用其程序不知道的方法(例如，通过设备)来改变的值。在说明段，也可同时使用存储类说明符和类型限制说明符。下例说明怎样在说明段编码类型限制说明符：

```
EXEC SQL BEGIN DECLARE SECTION;
    const int total=200;
    volatile short num;
    static const short sum=50;
EXEC SQL END DECLARE SECTION;
```

为了与 ANSI C 标准一致，PRO * C 预编译程序允许说明带或不带最大长度的 extern char[n] SQL 变量，例如

```
EXEC SQL BEGIN DECLARE SECTION;
    extern char dept_name[50];
    extern char name[ ];
    ...
EXEC SQL END DECLARE SECTION;
```

预编译程序在处理 name[]时产生警告信息，并假定其长度为 255 字节。因此，如果要存储长度超过 255 字节的 LONG 或 VARCHAR2 的列值时，必须指定最大长度。

在使用说明语句说明 SQL 变量的数据类型时，为变量指定的 C 数据类型必须与 ORACLE 的数据类型相兼容，否则就会产生数据类型转换错误。表 2-8 给出了与 ORACLE 的内部数据类型相兼容的 C 数据类型。

表 2-8　与 C 相兼容的 ORACLE 内部数据类型

内部类型	C 类型	描　　述
CHAR(X)(注[1])	char	单字符
	char[n]	n 字节的字符数组
VARCHAR2(Y)(注[1])	VARCHAR[n]	n 字节的变长字符数组
	int	整数
	short	短整数
	long	长整数
	float	单精度浮点数
	double	双精度浮点数
NUMBER	int	整数
	short	短整数
NUMBER(P,S)(注[2])	long	长整数
	float	单精度浮点数
	double	双精度浮点数
	char	单字符(注 3)
	char[n]	n 字节字符数组(注 4)
	VARCHAR[n]	n 字节变长字符数组
DATE(注[5])	char[n]	n 字节字符数组
LONG(注[6])	VARCHAR[n]	n 字节变长字符数组

RAW(X)（注[1]	unsigned char[n]	n字节无符号字符数组
LONG RAW（注[6]）	VARCHAR[n]	n字节变长字符数组
ROWID（注[7]）		
MLSLABEL(注[8])		

注意：

注1. X取值范围1至255，Y取值范围1至2000，缺省值为1

注2. P是精度，范围为1至38；S是定标，取值范围为－84至127

注3. 转换是字符，不是二进制

注4. 如果它们只包含可转换字符('0'至'9','.','+',' ','E','e'),串才能被转换为NUMBER。对于NLS设置可把十进制小数点转换为','。

注5. 当作为串类型转换时，DATE的长度取决于NLS设置。当作为二进制转换时，长度是7个字节。

注6. 对于VARCHAR，n不超过65533。

注7. 当作为串类型转换时，ROWID要求18至256字节。当作为二进制类型转换时，长度依赖于系统。

注8. 仅对Trusted ORACLE。

(2) SQL变量的引用

当在SQL语句中使用SQL变量时，必须在其之前加一个冒号(:)，但在C语句中引用时，不需加冒号。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
        float salary;
        float commission;
EXEC SQL END DECLARE SECTION;
        ...
printf("\nEnter employee number");
scanf("%d",&emp_number);
EXEC SQL SELECT SAL,COMM
        INTO :salary,:commission
        FROM EMP
        WHERE EMPNO= :emp_number;
pay=salary + commission;
......
```

2.4.2 指示器变量的说明和引用

(1) 什么是指示器变量

指示器变量是与宿主变量相关联的一类SQL变量，它被用来监督和管理与其相关联的宿主变量。每一个宿主变量都可定义一个指示器变量。指示器变量的具体作用如下：

- 向数据库表列输入Null值。
- 检查从数据库表列中选取的数据是否是Null值，或是否发生截断问题。

ORACLE如何实现指示器变量的上述作用呢？其办法是赋予它不同的值：

- 对于输入宿主变量，程序赋给指示器变量的值有如下含义：

−1:ORACLE 把 Null 值存入数据库的表列中，而不考虑与该指示器变量相关联的输入宿主变量的值。

≥0：ORACLE 把与该指示器变量相关联的输入宿主变量中的数据存入数据库表列中。

· 对于输出宿主变量，ORACLE 赋给指示器变量的值有如下含义：

−1：数据库表列的值是 Null，此时，与该指示器变量相关联的输出宿主变量中的值为不确定状态。

0：ORACLE 把数据库表列中的值原封不动地赋给与该指示器变量相关联的输出宿主变量中。

>0：ORACLE 把数据库表列中的值截断后赋给与该指示器变量相关联的输出宿主变量中。指示器变量返回一个整数值，指示该列值的原始长度，而且 SQLCA 的 SQLCODE 被置为 0。

(2) 指示器变量的说明

指示器变量在说明段被说明为短整型变量(2 字节)，其说明语句的位置是任意的，可放置在其相关宿主变量说明语句的前面或后面。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
        short ind_deptno; /* dept_number 的指示器变量 */
        int dept_number;
        char emp_name;
        short ind_name; /* emp_name 的指示器变量 */
EXEC SQL END DECLARE SECTION;
```

(3) 指示器变量的引用

在 SQL 语句中，指示器变量名字前应加冒号，而且必须附在其相关联的宿主变量之后。在 C 语句中，指示器变量如同 C 变量一样独立使用，不须前缀冒号，也不须跟在相关宿主变量之后。例如：

```
EXEC SQL SELECT EMPNO
        INTO :emp_number :ind_num
        FROM EMP
        WHERE ENAME = :emp_name;
if(ind_num = −1)
        printf("\n Employlee Number is Null!");
```

为了增进可读性，可以在宿主变量及其指示器变量之间加 INDICATOR 关键词。如：

```
EXEC SQL SELECT EMPNO
        INTO :emp_number INDICATOR :ind_num
        FROM EMP
        WHERE ENAME = :emp_name;
```

(4) 指示器变量的应用举例

例 1：向数据库表列中插入 Null 值

```
...
```

```
printf("\n" Enter Departmenrt number and name");
scanf("%d %s", &dept_number,dept_name );
if(dept_number=0)
        ind_num=-1;
else
        ind_num=0;
EXEC SQL INSERT INTO DEPT(DEPTNO, DNAME)
        VALUES(:dept_number:ind_num,:dept_name);
```

其中 ind_num 是 dept_number 的指示器变量。当输入的 dept_number 值是 0 时，则向 DEPT 表的 DEPTNO 列插入 Null 值。

例 2：检查从数据库表列中检索的值是否为 Null

```
EXEC SQL SELECT EMPNO, ENAME, SAL, COMM
        INTO :emp_number, :emp_name, :salary, : commission :ind_comm
        FROM EMP
        WHERE EMPNO=:emp_number;
if(ind_comm = -1)
        pay=salary
else
  pay=salary+commission;
```

2.4.3 指针宿主变量的说明和引用

(1) 指针宿主变量的说明

同 C 语言一样，PRO*C 也支持指针宿主变量。指针宿主变量在引用前也必须在说明段先说明。其说明格式同 C 语言。例如，下面的语句说明了两个指针宿主变量：

```
EXEC SQL BEGIN DECLARE SECTION:
    int *ptr;
    char *pname;
EXEC SQL END DECLARE SECTION;
```

(2) 指针宿主变量的引用

在 SQL 语句中引用指针变量时，指针名字前要前缀冒号(:)，而不加星号(*)。在 C 语句中的用法如同 C 语言的指针变量。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
    flat salary;
    char *pname;
    int emp_number;
EXEC SQL END DECLARE SECTION;
. . . . . .
EXEC SQL SELECT ENAME, SAL
    INTO :pname, :salary;
```

```
        FROM EMP
        WHERE EMPNO= :emp_number;
    printf("%s,%f",pname,salary);
```

指针所指对象的长度，对字符串来说，假设它以 Null 终结，其长度在运行时通过引用 strlen()函数来求出；其它对象则由其数据类型决定。

PRO*C 在说明段不能说明结构类型的变量，但它能用指针来引用结构成员。例如：

```
struct{
      int student_number;
      char name[20];
      char sex;
      int old;
      }student;
EXEC SQL BEGIN DECLARE SECTION;
    int *pnumber;
    char *pname;
    ......
EXEC SQL END DECLARE SECTION;
main()
{......
    pnumber = &student.student_number;
    pname = student.name;
    ......
}
```

2.4.4 数组 SQL 变量的说明和引用

(1)数组 SQL 变量的说明

PRO*C 支持一维数组，引用之前也必须先在说明段由说明语句说明。说明语句的格式同 C 语言。数组可说明为宿主数组，也可说明为指示器数组。一个指示器数组是与一个宿主数组相关联的。下面是数组说明的例子。

```
EXEC SQL BEGIN DECLARE SECTION;
    int emp_number[100];
    char emp_name[100][15];
    float salary[100];
    float commission[100];
EXEC SQL END DECLARE SECTION;
```

注意：

- PRO*C 不支持指针数组
- PRO*C 只支持一维数组，而 emp_name[100][15]视为一维字符串。
- 数组最大维数为 32767，如果超出此限制，则出现

"parameter out of range"错误。

(2) 数组 SQL 变量的引用

在 SQL 语句中引用数组时，只需写数组名，(注意：名字要前缀冒号)，不需写下标。在 C 语句中的用法同 C 语言中数组的用法。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
    int emp_number[100];
    char emp_name[100][15];
    float salary[100],commission[100];
    int dept_number;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT EMPNO ,ENAME,SAL,COMM
    INTO:emp_number, :emp_name, :salary, :commission
    FROM EMP
    WHERE DEPTNO = :dept_number;
```

而如下的写法是错误的

```
for(i=1; i<=100; i++)
EXEC SQL SELECT EMPNO, ENAME, SAL, COMM
    INTO :emp_number[i], :emp_name[i], :salary[i],commission[i]
    FROM EMP
    WHERE DEPTNO = :dept_number;
```

下面是使用指示器数组的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
    int emp_number[100];
    char emp_name[100][15];
    float salary[100]
    float commission[100];
    short ind_comm[100];      /* 指示器数组 */
EXEC SQL END DECLARE SECTION;
    ......
EXEC SQL INSERT EMP(EMPNO,ENAME,SAL,COMM)
        VALUES(:emp_number, :emp_name, :salary,commission:ind_comm);
```

由上所述可知，在 PRO * C 中引用数组有如下两点好处：

· 大大简化程序设计：省去大量不必要的变量命名及引用。

· 改进程序性能。如在 SOL 语句中，如果没有数组，上例中插入 100 行就要重复 100 次，而引入数组后则只须执行一次 INSERT 语句即可。这大大降低网络传输开销。

在使用宿主数组时，应注意如下几点：

· 当在 SOL 语句中引用多个数组时，这些数组的维数应当相同，否则 PRO * C 使用最小的数组维数，并发出警告。

• 在 VALUES、SET、INTO 或 WHERE 子句中，不允许把简单 SQL 变量与数组 SQL 变量混用。

• 在 UPDATE 或 DELETE 语句中，不允许把数组与 CURRENT OF 子句一起使用。

2.4.5 VARCHAR 变量的说明和引用

VARCHAR 数据类型用于说明一个变长字符串。它是 C 数据类型的扩充。C 语言中并没有该数据类型，为了便于字符串的处理，特引入了该类型。所以也把它叫做伪类型。

(1) VARCHAR 变量的说明

VARCHAR 变量在引用之前也必须在说明段说明，其说明的方法如下例所示：

```
EXEC SQL BEGIN DECLARE SECTIN;
    VARCHAR vstring[20];
EXEC SQL END DECLARE SECTION;
```

在预编译时，该变量被翻译成 C 语言中的一个结构变量：

```
struct{
    unsigned short len;
    unsigned char arr[20];
}vstring;
```

该结构变量包含两个成员：长度成员和数组成员。数组成员保存字符串，长度成员保存字符串的长度。结构变量的名字即 VARCHAR 变量的名字。

在说明 VARCHAR 变量时，必须指出串的最大长度，其长度界于 1 至 65533。下面的说明是无效的：

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR Vstring[ ];
EXEC SQL END DECLARE SECTION;
```

(2) VARCHAR 变量的引用

在 SQL 语句中引用时，应引用以冒号为前缀的结构名，而不写名字后的方括号及其下标，在 C 语句中引用时应引用结构的成员。如下例所示：

```
EXEC SQL BEGIN DECLARE SECTION;
    ......
    VARCHAR book_desc[100];
EXEC SQL BEGIN DECLARE SECTION;
    ......
main( )
{   ......
    EXEC SQL SELECT BDESC
      INTO :book_desc;
      FROM BOOK
      WHERE BNO = :book_number;
    book_desc.arr[book_desc.len]='\0';
```

```
    printf ("\n%s",book_desc.arr);
    ......
}
```

在执行上例时,ORACLE 把检索的字符串存放在 book_desc.arr 中,把串长存放在 book_desc.len 中。于是可使用长度成员来为字符串加 Null 终结符了。

VARCHAR 变量在作为输入变量引用时,程序应该先把字符串存入数组成员中,其长度存入长度成员中,然后再在 SQL 语句中引用。如下例:

```
EXEC SQL BEGIN DECLARE SECTION;
    ......
    int book_number;
    VARCHAR book_name[50];
    VARCHAR book_desc[100];
EXEC SQL END DECLARE SECTION;
main ( )
{ ......
  printf ("\n Enter book name、number and description:\n");
  scanf ("%s, %d, %s",book_name.arr,&book_number,book_desc.arr);
  book_name.len = strlen (book_name.arr);
  book_desc.len = strlen (book_desc.arr);
  EXEC SQL UPDATE BOOK
      SET BNAME = :book_name,
          BDESC = :book_desc
      WHERE BNO = :book_number;
  ......
}
```

§2.5 通讯区的说明

为了记录每个 SQL 语句的执行状态,以便进行错误诊断,ORACLE 数据库管理系统特提供了两个通讯区:SQL 通讯区(SQLCA)和 ORACLE 通讯区(ORACA)。

2.5.1 SQLCA 的说明

(1)么是 SQLCA

SQLCA 是一个结构类型的变量,它是 ORACLE 和应用程序的一个接口。该结构变量是为诊断错误和事件处理而设置的。

在执行 PRO * C 程序时,ORACLE 把每一个嵌入 SQL 语句执行的状态信息存入 SQLCA 中,这些信息包括错误代码、警告标志设置、诊断文本和处理行数等。如图 2.2 所示:

因此,在每一个可执行 SQL 语句执行后,就可根据 SQLCA 中的状态信息来判断该

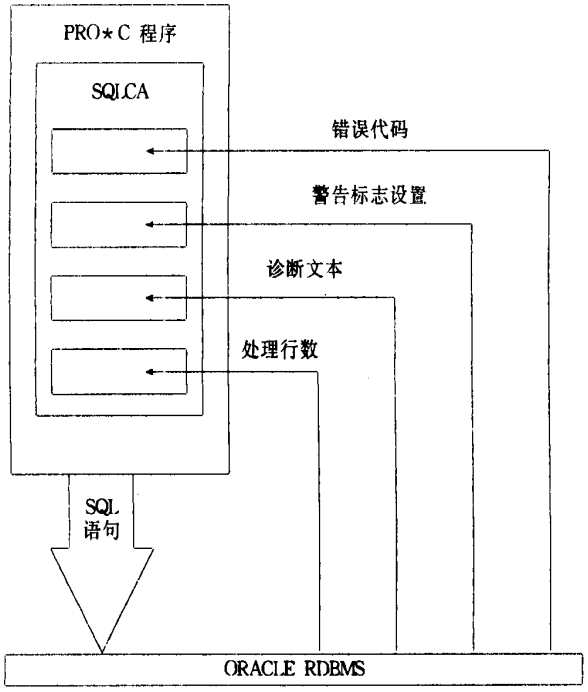


图2-2 SQCA 中保留的状态信息

SQL 语句的执行是否成功？出现什么错误和例外？处理了多少行？……。

(2) SQLCA 的组成

SQLCA 是 C 语言中的一个结构变量，其组成如表 2-9 所示。

表 2-9 SQLCA 中的变量

```
struct sqlca
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        unsigned short sqlerrml;
        char     sqlerrmc[70];
    }.sqlerrm;
    char     sqlerrp[8];
    long     sqlerrd[6];
    char     sqlwarn[8];
    char     sqlext[8];
};
struct sqlca sqlca;
```

sqlcaid：是字符串变量，它被初始化为 sqlca。它被用于标识一个 SQL 通讯区。

sqlabc：是一个整型变量，用于保留 SQL 通讯区的长度(以字节为单位)。

sqlcode：是一个整型变量，用于保留最近执行的 SQL 语句的状态码。这些状态码指明 SQL 语句的操作结果，可能是下列值之一：

0：表示该 SQL 语句被正确执行，没有发生错误和例外。

>0：意味着 ORACLE 执行了该语句，但遇到一个例外。当 ORACLE 未找到满足 WHERE 子句检索条件的行时，或者 SELECT 或 FETCH 未有行返回时，就出现例外。

<0：表示由于数据库、系统、网络或应用程序的错误，ORACLE 未执行该 SQL 语句。当出现这类错误时，当前事务一般应回滚。

sqlerrm：是个子结构，它包含如下两个元素：

sqlerrml：保留存储在 sqlerrmc 中的信息文本长度。

sqlerrmc：保留与 sqlcode 中的错误代码相对应的错误信息文本。信息文本的最大长度不超过 70 个字符。要取得长于 70 个字符的完整信息文本，必须用 SQLGLM()函数。只有 sqlcode 中的代码为负时，才引用 sqlerrmc 中的信息。如果当 sqlcode 中的代码为 0 时引用了 sqlerrmc，则取出的是与前面的 SQL 语句有关的信息。

sqlerrp：该字段目前尚未使用，保留将来使用。

sqlerrd：它是一个数组，有六个元素：

sqlerrd[0]：留待将来使用

sqlerrd[1]：留待将来使用

sqlerrd[2]：保存当前 SQL 语句处理的行数。但是，如果当前的 SQL 语句失败，则 sqlerrd[2]中的值无定义。如果在处理中间出现错误，则 sqlerrd[2]给出成功处理的行数。在 OPEN 语句执行后，sqlerrd[2]被置成 0，在 FETCH 后增值。对于 EXECUTE、INSERT、UPDATE、DELETE 和 SELECT INTO 语句，该计数反映成功处理的行数。

sqlerrd[3]：留待将来使用

sqlerrd[4]：保存相对位移，它指出在那个字符位置开始出现(语法)分析错误。其中第一个字符的相对位移是 0。

sqlerrd[5]：留待将来用

sqlwarn：是字符型数组，有 8 个元素，被用作警告标志。如果在执行 SQL 语句时有例外发生，则相应的元素置“W”标志。各元素描述如下：

sqlwarn[0]：如果其它警告标志被设置的话，该元素就被设置。

sqlwarn[1]：如果把一个截短的列值赋给一个输出宿主变量的话，则设置该标志。仅对字符数据截短时，才设置该标志；对于数字数据的截短并不设置该标志，也不返回负的 sqlcode 码。为了发现是否有截短发生，以及按多长截短，可检查输出宿主变量所对应的指示器变量，该指示器变量中的正值是列的原始长度。

sqlwarn[2]：该标志不再使用

sqlwarn[3]：如果查询选择表中的列数不等于 SELECT 或 FETCH 语句的 INTO 子句中的宿主变量个数时，设置该标志。返回的项数是二者中较小的项数。

sqlwarn[4]：如果表中的每一行都被未有 WHERE 子句的 DELETE 或 UPDATE 语句处理，则设置该标志。

sqlwarn[5]：当 EXEC SQL CREATE{PROCEDURE | FUNCTION | PACKAGE | PACKAGEBODY}语句由于 PL/SQL 编译错误而失败时，该标志被设置。

sqlwarn[6]、sqlwarn[7]：不再使用

sqlext：该串字段保留将来使用。

(3) SQLCA 的说明

当 MODE={ANSI13|ORACLE}时，要求说明 SQLCA，如果不说明的话，将导致编译或连接错误。当 MODE={ANSI|ANSI14}时，SQLCA 的说明是可选的，但必须说明一个名为 SQLCODE 的状态变量。

SQLCA 的说明方式有两种：

·用如下形式的 INCLUDE 语句：

EXEC SQL INCLUDE sqlca;

·把表 2-9 中的代码直接编写到程序中。

一个 PRO*C 程序可说明多个 SQLCA，例如可以有一个全程 SQLCA 和几个局部 SQLCA。ORACLE 仅对活动的 SQLCA 返回信息。

当 PRO*C 程序是由几个源文件组成时，全局 SQLCA 可以在一个源文件中定义，而在另一个源文件中说明为 extern 存储类，其说明方式是：

#define SQLCA_STORAGE_CLASS extern

可通过定义符号 SQLCA_INIT 来初始化 SQLCA。初始化 SQLCA 是好的程序设计风格。但是，如果 SQLCA 被说明为自动变量，可能不允许用 SQLCA_INIT 初始化它，因为某些操作系统和 C 编译程序不允许初始化自动变量。

SQLCA 主要用于错误诊断和事件处理，关于这方面的内容请参考 3.6 节错误处理。SQLCA 能提供至多 70 个字符长度的错误信息。要取得更长的完整错误信息文本，需要使用函数 sqlglm()。调用该函数的格式如下：

sqlglm(message_buffer, &buffer_size, &message_length);其中：

message_buffer：是文本缓冲区，ORACLE 把错误信息文本存放在这里，并用空格填充该缓冲区尾部。

buffer_size：是整型变量，存放缓冲区的长度(以字节为单位)。

message_length：是整型变量，ORACLE 把错误信息的实际长度存放在这里。

ORACLE 错误信息的最大长度是 512 个字符，其中包括错误代码、嵌套信息等。该函数返回的错误信息的最大长度取决于对 buffer_size 指定的值。在下例中，用 sqlglm()取得至多 100 个字符的错误信息：

```
main()
{
    char msg_buf[100];
    int buf_size=100;
    int msg_len;
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror
    ...
    sqlerror;
      sqlglm(msg_buf, &buf_size, &msg_len);
      msg_buf[msg_len]='\0';
      printf("\n%s \n", msg_buf);
      ...
}
```

2.5.2 ORACA 的说明

(1)么是 ORACA

ORACA 是一个类似于 SQLCA 的数据结构，可把它视为 SQLCA 的辅助通讯区。当需要的错误及状态信息比 SQLCA 提供的还要多时，就用 ORACA。ORACA 的使用是可选的，因为它增加了运行开销。

ORACA 帮助你诊断问题和监控 ORACLE 资源(如 SQL 语句执行和光标高速缓冲寄存器等)使用。

(2) ORACA 的组成

ORACA 是 C 中的一个结构变量，其组成如表 2-10 所示。

表 2-10 ORACA 中的变量

```
ORACA
  struct oraca
  {
      char          oracaid[8];
      long          oracabc;
      long          oracchf;
      long          oradbgf;
      long          orahchf;
      long          orastxtf;
      struct
          {
            unsigned short orastxtl;
            char     orastxtc[70];
          }orastxt;
      struct
          {
            unsigned short orasfnml;
            char     orasfnmc[70];
          }orasfnm;
      long          oraslnr;
      long          orahoc;
      long          oramoc;
      long          oracoc;
      long          oranor;
      long          oranpr;
      long          oranex;
  };
  struct oraca oraca;
```

ORACA 中的变量描述如下：

oracaid：

它被初始化为“ORACA”，用于标识一个 ORACLE 通讯区。

oracabc：

用于保存 ORACA 数据结构的长度(以字节为单位)。

oracchf：

如果主 DEBUG 标志(oradbgf)被设置，则该标志能在每个光标操作之前，搜集缓冲存储器的统计，并检查光标缓冲存储器的一致性。ORACLE 运行库做该一致性检查，并把发出的错误信息返回给 SQLCA。这个标志有下列设置：

0：禁止光标缓冲存储器一致性检查(缺省)。

1：能进行光标缓冲存储器一致性检查。

oradbgf：

这是 DEBUG 的主标志字段，根据它决定是否选择 DEBUG 操作。它有以下两个设置：

0：禁止所有的 DEBUG 操作(缺省)

1：允许进行所有的 DEBUG 操作。

orahchf：

它也是一个标志字段。如果主 DEBUG 标志(oradbgf)被设置，则每当预编译程序动态分配或释放内存单元时，它通知 ORACLE 运行库检查堆一致性。该标志必须在 CONNECT 命令发出之前设置，并且一旦设置，就不能被清除，它不考虑后续的变更要求。该标志对于发现破坏内存的程序错误是有用的。它有如下两个设置：

0：禁止堆一致性检查(缺省)。

1：允许堆一致性检查。

orastxtf：

该标志字段指定当前的 SQL 语句文本是否被保存，以及如何保存。它有如下设置：

0：不保存 SQL 语句文本(缺省)。

1：仅对 SQLERROR 保存 SQL 语句文本。

2：仅对 SQLERROR 或 SQLWARNING 保存 SQL 语句文本。

3：总是保存 SQL 语句文本。

SQL 语句文本被保存在命名为 orastxt 的子结构中。

以下的变量有助于迅速定位错误。

orastxt：

这是一个子结构，它有两个成员。用它来帮助发现错误的 SQL 语句。该结构保存 ORACLE 最近所分析的 SQL 语句文本。该结构的两个成员的描述如下：

- orastxtl：保存当前 SQL 语句的长度。
- orastxtc：保存当前 SQL 语句的文本。至多保存文本的开始 70 个字符。

由预编译程序所分析的语句不被保存在 ORACA 中。

orasfnm：

这是一个子结构，它用来标识包含当前 SQL 语句的源文件名。当一个应用由多个源文件组成时，用这个子结构帮助发现当前出错的 SQL 语句是处于那个源文件内。该结构包含如下两个字段：

- orasfnml：标识存放在 orasfnmc 中的文件名的长度。
- orasfnmc：保存源文件名，至多 70 个字符。

oraslnr：

标识当前 SQL 语句所在行的行号

如果主 DEBUG 标志(oradbgf)和光标缓冲存储器标志被设置，下边所叙述的变量能使你搜集光标缓冲存储器统计。这些变量由程序中所发的每一个 COMMIT 或 ROLLBACK 命令自动设置。在内部，对于每个连接的数据库都有一组这样的变量，ORACA 中的当前值从属于该数据库，最近的 COMMIT 或 ROLLBACK 相对于这些值被执行。

orahoc

它记录程序运行期间 MAXOPENCURSORS 被设置的最高值。

oramoc

它记录程序所需要打开的 ORACLE 光标的最大数。如果 MAXOPENCURSORS 被设置得太低，该最大数可能比 orahoc 大，则强制预编译程序扩展光标缓冲存储器。

oracoc

该字段记录程序当前打开的 ORACLE 光标数。

oranor

该字段记录程序所需要的光标缓冲存储器的再赋值次数。该数说明在光标缓冲存储器中“重复做”的程度，应使它尽可能低。

oranpr

该整型字段记录程序所需要的 SQL 语句分析次数。

oranex

该整型字段记录程序所需要的 SQL 语句执行次数，该数与 oranpr 数的比率应尽可能高。换句话说，就是要避免不必要的再分析。

(3) ORACA 的说明

ORACA 必须在说明段外边说明，说明的方式有两种：

· 用如下形式的 INCLUDE 语句：

```
EXEC SQL INCLUDE ORACA;
```

· 把表 2-10 中的代码直接编写到程序中。

在一个 PRO * C 程序中可使用多个 ORACA，例如一个全程的 ORACA 和几个局部的 ORACA。ORACLE 仅对“活动”的 ORACA 返回信息。

为了能够使用 ORACA，必须把 ORACA 预编译可选项指定为“YES”。指定方式有两种：

· 在命令行上指定 ORACA=YES

· 在程序行上编写如下语句：

```
EXEC ORACLE OPTION(ORACA=YES);
```

ORACA 是可选的，因为它增加了运行开销。其缺省设置为 ORACA=NO。

关于 ORACA 的应用请参考 3.6 错误处理。

2.5.3 SQLCODE 说明

当 MODE={ANSI|ANSI14}时，必须在说明段内部或外部说明一个 4 字节的名为 SQLCODE 整型变量，名字必须大写，如下例：

```
/* 说明宿主变量和指示器变量 */
EXEC SQL BEGIN DECLARE SECTION;
    ...
EXEC SQL END DECLARE SECTION;
/*说明状态变量 */
long SQLCODE;
```

允许说明局部或全程 SQLCODE 变量。一个 PRO*C 程序可说明一个全程 SQLCODE 和多个局部 SQLCODE。对局部 SQLCODE 的存取受其作用域限制。

在每一个 SQL 语句操作后，ORACLE 把一个状态码返回给当前有效作用域的 SQLCODE。然后程序可显式地或隐式地(用 WHENEVER 语句)访问该 SQLCODE，以决定当前 SQL 语句的执行状态。

状态码指出 SQL 语句操作的结果。状态码的取值及含义参考 SQLCA 中的 SQLCODE 成员。

如果在一个编译单位内只说明了一个 SQLCODE 变量，而未说明 SQLCA，则预编译程序为该编译单位分配一个内部 SQLCA，程序不能存取该内部 SQLCA。如果说明了一个 SQLCA 和 SQLCODE，则在每一个 SQL 语句操作后，ORACLE 把相同的状态码返回给二者。

当 MODE=｛NASI13｜ORACLE｝时，不必说明 SQLCODE 变量。

2.5.4 INCLUDE 语句

在前面，我们已提到用 INCLUDE 语句来说明通讯区，本段进一步说明它的使用方法。

INCLUDE 语句被用于把一个文件拷贝到程序内。它类似于 C 语言中的 #include 命令。其一般格式为：

EXEC SQL INCLUDE filename；当预编译时，该语句被 filename 文件的副本所替代。能 INCLUDE 任何文件。

如果系统使用文件扩展名，而又未指定它时，对于头文件，PRO*C 预编译程序假定为缺省扩展名(一般为 h)。如果未找到时，对源文件，PRO*C 预编译程序假定为缺省扩展名(一般为 C)。缺省扩展名取决于系统，系统不同扩展名也可能不同。

如果系统使用目录，可通过设置预编译可选项

INCLUDE=path 来为 INCLUDE 文件设置目录路径。其中 path 的缺省值是当前目录。预编译程序首先检索当前目录，然后检索由 INCLUDE 指定的目录，最后检索标准 INCLUDE 文件目录。因此，对 SQLCA 和 ORACA 这样的标准文件不必指定目录路径。但对非标准文件(除非它们被存放在当前目录内)必须用 INCLUDE 指定目录路径。可以像下面那样在命令行上指定多个路径：

… INCLUDE=path1 INCLUDE=path2 …在此情况下，预编译程序首先检索当前目录，然后顺序检索由 path1、path2…命名的目录，最后检索标准 INCLUDE 文件目录。

注意：由于预编译程序首先检索当前目录中的文件。因此，如果要 INCLUDE 的文件存放在别的目录中时，就要确保当前目录中没有与其同名的文件。另外，如果所用的操作

系统(如 UNIX)对大小写字母敏感时，就要保证指定的文件名与目录中存放的名字在大小写上完全一样。不要把 INCLUDE 语句与 C 中的 #include 命令混淆。对于含有嵌入式 SQL 语句的文件，应当用 INCLUDE 语句拷贝，因为它要被预编译。对于不包含嵌入式 SQL 语句的文件，可使用二者中的任何一个。例如，可用 #include 拷贝 SQLCA 和 ORACA。

第三章　应用程序的设计方法及举例

§3.1　PRO＊C 程序中嵌入的 SQL 语句

3.1.1　PRO＊C 程序中能嵌入的 SQL 语句

在应用程序中嵌入的 SQL 语句叫嵌入式 SQL 语句。允许在 PRO＊C 程序中嵌入的 SQL 语句如表 3-1 所示。它们的书写语法列在附录 A 中。

表 3-1　能在应用程序中嵌入的 SQL 语句

语句类型	语　句　名
数据定义	ALTER　ANALYZE　AUDIT COMMENT　CREATE　DROP GRANT　NOAUDIT　RENAME REVOKE　TRUNCATE
数据操纵	DELETE　EXPLAINPLAN　INSERT LOCK　TABLE　SELECT　UPDATE
会话期控制	ALTER　SESSION　SET　ROLE
系统控制	ALTER　SYSTEM
事务控制	COMMIT　ROLLBACK　SAVEPOINT SET　TRANSACTION

3.1.2　嵌入式 SQL 语句的书写文法

在 PRO＊C 程序中，能把 SQL 语句和 C 语言语句自由地混合书写，并能在 SQL 语句中使用 SQL 变量。嵌入式 SQL 语句的书写文法是：

- 以关键字 EXEC SQL 开始，
- 以 C 语言的语句终结符(分号)终结，
- 大多数嵌入式 SQL 语句与交互式 SQL 语句的区别仅仅是增加了一些子句或使用了一些 SQL 变量。

下面的语句表示嵌入式 SQL 语句与交互式 SQL 语句的文法区别。例如交互式 SELECT 和 ROLLBACK 的写法是：

```
SELECT EMPNO,ENAME,SAL, DEPTNO
    FROM EMP
    WHERE ENAME='ALLEN';
ROLLBACK WORK;
```

而嵌入式的写法是：

```
EXEC SQL SELECT EMPNO,ENAME,SAL, DEPTNO
```

```
        INTO :emp_number,:emp_name,:salary, :deptno
        FROM EMP
        WHERE ENAME='ALLEN';
    EXEC SQL ROLLBACK WORK;
```

3.1.3 可执行SQL语句和说明性SQL语句

PRO*C程序中嵌入的SQL语句分为可执行SQL语句和说明性SQL语句两类。

可执行语句导致对运行库SQLLIB的调用。执行它可实现应用程序与ORACLE的连接,可定义、查询、操纵ORACLE数据,也能控制对ORACLE数据的存取和对事务的处理等。它们能放置在宿主语言的可执行语句所能放置的任何位置。

说明性SQL语句不会导致对运行库SQLLIB的调用和返回代码,也不可能操纵ORACLE数据库。使用它们来说明ORACLE对象、通讯区和变量等。它们能被放置在宿主语言的说明语句所能放置的任何位置。表3-2给出了SQL语句的分类。

表3-2 嵌入式SQL语句分类

分类	语句	说明
说明性语句	ARRAYLEN*	PL/SQL情况下使用宿主数组
	BEGIN DECLARE SECTION*	说明宿主变量段的开始语句
	END DECLARE SECTION*	说明宿主变量段的结束语句
	DECLARE*	命名ORACLE对象
	INCLUDE*	文件拷贝
	TYPE*	用户定义数据类型等价
	VAR*	宿主变量等价
	WHENEVER	错误处理说明
可执行语句	ALTER	定义和控制对ORACLE数据的存取
	ANALYZE	
	AUDIT	
	COMMENT	
	CONNECT*	
	CREATE	
	DROP	
	GRANT	
	NOAUDIT	
	RENAME	
	REVOKE	
	TRUNCATE	
	CLOSE*	检索和操纵ORACLE数据
	DELETE	
	EXPLAINPLAN	
	FETCH*	
	INSERT	
	OPEN*	
	SELECT	
	UPDATE	

COMMIT ROLLBACK SAVEPOINT SET TRANSACTION	处理事务
DESCRIBE * EXECUTE * PREPARE *	使用动态 SQL
ALTER SESSION SET ROLE	控制会话期

注：表中有'*'的语句表示没有交互的对应语句

§3.2 应用程序的登录

为了对数据库中的数据进行查询和操纵，PRO * C 程序必须在进行这些操作之前，把程序与 ORACLE 数据库连接起来，即登录到 ORACLE 上。

3.2.1 有关登录的几个概念

(1)节点

网络中的通讯点叫节点。SQL * Net 把信息(SQL 语句、数据和状态码等) 从一个节点传递到另一个节点。

(2)协议

协议是在网络上存取的一组规则。这些规则确立了失败后恢复的过程、传送数据和检查错误的格式等。本节中的所有例子都是用 DECnet 的网络存取协议。

(3)SQL * Net 的连接语法

SQL * Net 通过 DECnet 对远程节点上的缺省数据库进行连接的语法是

d:node

其中 d 指出网络(DECnet)，而 node 指出远程节点。

通过 DECnet 对远程节点上的非缺省数据库进行连接的语法是

d:node—database 其中 node—database 指出非缺省数据库。

(4)缺省数据库和连接

每一个节点都有一个缺省数据库。如果在 CONNECT 语句中，指定了节点而未指定数据库，则连接到当地或远程节点的缺省数据库上。如果既未指定数据库又未指定节点时，则连接到当前节点的缺省数据库上。

缺省连接通过没有 AT 子句的 CONNECT 语句实现。连接可能是对在任何一个当地或远程节点上的任意一个缺省或非缺省数据库。没有 AT 子句的 SQL 语句是相对于缺省连接执行的。相反，非缺省连接是通过具有 AT 子句的 CONNECT 语句实现。有 AT 子句的 SQL 语句是相对于非缺省连接执行的。本书的绝大多数例子都是相对于缺省连接的。

所有的数据库名必须唯一，但两个或多个数据库名能指定相同连接。即能对任何节点上的任一数据库有多个连接。

3.2.2 登录到当前节点的缺省数据库上

向当前节点的缺省数据库上登录使用的 CONNECT 语句有如下三种方式：

(1)有 IDENTIFIED BY 子句，其格式如下：

EXEC SQL CONNECT :username IDENTIFIED BY :password；其中 username 和 password 是 char 或 VARCHAR 类型的宿主变量，分别包含用户的名字和口令。

(2)没有 IDENTIFIED BY 子句，其格式为：

EXEC SQL CONNECT :usr_pwd；其中宿主变量 usr_pwd 内包含由字符"/"分隔的用户名和口令(即 username/password)。

在使用以上两种格式进行登录时，应当首先在说明段定义包含用户名和口令的宿主变量；并在执行 CONNECT 语句之前设置它们，否则会造成登录失败。下面是实现登录的例子：

```
/* 说明 username 和 password */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username[20];
    VARCHAR password[20];
    ......
EXEC SQL END DECLARE SECTION;
...
/* 初始化 username 和 password */
strcpy(username.arr,"SCOTT");
username.len=strlen(username.arr);
strcpy(password.arr,"TIGER");
password.len=strlen(password.arr);
/* 登录到 ORACLE 上 */
EXEC SQL WHENEVER SQLERROR GOTO sqlerr;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

但是，不能把用户名和口令直接编写到 CONNECT 语句中，或者把用引号(')括起来的字母串写在 CONNECT 语句中。如下面的语句是无效的：

```
EXEC SQL CONNECT SCOTT IDENTIFIED BY TIGER;
EXEC SQL CONNECT 'SCOTT' IDENTIFIED BY 'TIGER';
```

(3)自动登录

用用户名 OPS$username 进行的登录叫自动登录。其中 username 是当前登录到操作系统上的用户的名字或任务名，而 OPS$username 是有效的 ORACLE 用户名。登录时，只需传递给预编译程序一个斜杠(/)字符。

例如

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    ...
    char oracleid='/';
EXEC SQL END DECLARE SECTION;
    ...
EXEC SQL CONNECT :oracleid;
```

这就把用户名为 OPS$username 的用户登录到 ORACLE 上了。如果用户登录到操作系统上的名字是 WANG,则该用户就以名字 OPS$WANG 登录到 ORACLE 上。

也可把一个串传递给预编译程序,但串中不能包含尾部空格。例如下例中的 CONNECT 语句将失败:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    char oracleid[10];
EXEC SQL END DECLARE SECTION;
...
strncpy(oracleid, "/       ", 10);
EXEC SQL CONNECT :oracleid;
```

3.2.3 单显式登录

上节我们描述了缺省连接,即对当地节点的缺省数据库的连接。如果想连接到其它数据库上,就必须显式地标识那个数据库。

在显式登录时,应该命名一个连接名,该连接名将在 SQL 语句的 AT 子句中引用。

下面的例子是对一个远程节点上的单个非缺省数据库进行登录:

```
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR username[10];
        VARCHAR password[10];
        VARCHAR db_name[20];
EXEC SQL END DECLARE SECTION;
...
/* 初始化 username 和 password */
strcpy(username.arr, "SCOTT");
username.len=strlen(username.arr);
strcpy(password.arr, "TIGER");
password.len=strlen(password.arr);

/* 把数据库名赋予 db_name */
strcpy(db_name.arr, "d:newyork_nondef");
db_name.len=strlen(db_name.arr);
```

```
/* 命名一个链接名 db_link_name */
EXEC SQL DECLARE db_link_name DATABASE;
/* 登录到远程节点的一个非缺省数据库上 */
EXEC SQL CONNECT :username IDENTIFIED BY:password
    AT db_link_name USING :db_name;
```

在上例中，宿主变量 username 和 password 标识一个用户。db_name 包含要登录到其上的远程节点的数据库名。db_link_name 命名一个非缺省连接，它是 ORACLE 使用的标识符，而不是宿主或程序变量，因此不需在说明段说明。

USING 子句指出与 db_link_name 相联系的网络、机器和数据库。其后就可使用含有 AT 子句(具有 db_link_name)的 SQL 语句在 db_name 指定的数据库上进行操作。

对上例，可在 AT 子句中使用字符宿主变量，因此该例可重写如下：

```
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username[10];
    VARCHAR password[10];
    char db_link_name[10];
    VARCHAR db_name[20];
EXEC SQL END DECLARE SECTION;
...
/* 初始化 username 和 password */
strcpy(username, "SCOTT");
strcpy(password, "TIGER");
username.len=strlen(username.arr);
password.len=strlen(password.arr);
/* 把数据库名赋予 db_name */
strcpy(db_name.arr, "d:newyork_nondef");
db_name.len=strlen(db_name.arr);
/* 初始化 db_link_name */
strcpy(db_link_name, "oracle1");

/* 登录到远程节点的一个非缺省数据库上 */
EXEC SQL CONNECT :username IDENDIFIED BY :password
            AT :db_link_name USING :db_name;
...
```

如果 db_link_name 是宿主变量时，则不需要 DECLARE DATABASE 语句。如果 db_link_name 是没被说明的标识符时，需在执行 CONNECT 语句之前有 DECLARE DATABASE 语句。

(1)在非缺省数据库上的 SQL 操作

如果授权的话，可在非缺省连接上执行任意 SQL 语句，如：

```
EXEC SQL AT db_link_name SELECT …
EXEC SQL AT db_link_name INSERT …
EXEC SQL AT db_link_name UPDATE …
```

下边语句中的 db_link_name 是宿主变量：

```
EXEC SQL AT :db_link_name DELETE …
```

如果 db_link_name 是宿主变量时，则 SQL 语句中引用的所有表必须在 DECLARE TABLE 语句中定义。否则预编译程序发出警告。

(2)光标控制

不能在光标控制语句(如 OPEN、FETCH、CLOSE)中使用 AT 子句。如果想把一个光标与一个显示标识的数据库相关联，就要在 DECLARE CURSOR 语句中使用 AT 子句。如：

```
EXEC SQL AT :db_link_name DECLARE emp_cursor
            CURSOR FOR …
EXEC SQL OPEN emp_cursor …
EXEC SQL FETCH emp_cursor …
EXEC SQL CLOSE emp_cursor;
```

如果 db_link_name 是宿主变量，那么它的说明必须在所有引用该光标的 SQL 语句范围内。例如，如果在一个函数中 OPEN 光标，而在另一个函数内由该光标做 FETCH，则必须把 db_link_name 说明为全程宿主变量。

在使用光标时，SQL 语句是对 DECLARE CURSOR 语句的 AT 子句中所命名的数据库进行操作。如果在光标说明语句中未使用 AT 子句，则是对缺省数据库进行操作。

AT：host_variable 子句允许变更与光标的联系；但是在光标还在打开时，不能变更该联系。请看下面的例子：

```
EXEC SQL AT :db_link_name DECLARE emp_cursor
            CURSOR FOR …
strcpy(db_link_name, "oracle1");
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO …
strcpy(db_link_name, "oracle2");
EXEC SQL OPEN emp_cursor; /* 非法打开,因 原光标仍在打开 */
EXEC SQL FETCH emp_cursor INTO …
```

这是非法的，因为在试图执行第二个 OPEN 语句时 emp_cursor 仍在打开。ORACLE 不支持一个光标同时与几个数据库相联系。因此，当一个光标为某连接打开时，必须先关闭它，然后才能使它为另一个连接再打开。改正上例错误的方法是先关闭，然后再打开。如下例所示。

```
…
EXEC SQL CLOSE emp_cursor; /* 关闭第一个光标 */
strcpy(db_ link_name, "oracle2");
EXEC SQL OPEN emp_cursor;
```

EXEC SQL FETCH emp_cursor INTO …

(3)动态 SQL

在动态 SQL 语句中使用 AT 子句的方法如下：

对动态 SQL 方法 1，如果想对非缺省连接使用动态 SQL 语句，就必须用 AT 子句。如：

EXEC SQL AT :db_link_name EXECUTE IMMEDIATE :sql_stmt;

对动态方法 2、3 和 4，如果想对非缺省连接执行相应的动态 SQL 语句，就只能在 DECLARE STATEMENT 语句中使用 AT 子句，所有其它动态 SQL 语句（如 PREPARE、DESCRIBE、OPEN、FETCH 和 CLOSE 等）决不能使用 AT 子句。下面的例子是对动态方法 2 的：

EXEC SQL AT :db_link_name DECLARE sql_stmt STATMENT;

EXEC SQL PREPARE sql_stmt FROM :sql_string;

EXEC SQL EXECUTE sql_stmt;

下面是动态方法 3 的例子

EXEC SQL :db_link_name DECLARE sql_stmt STATEMENT;

EXEC SQL PREPARE sql_stmt FROM :sql_string;

EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;

EXEC SQL OPEN emp_cursor …

EXEC SQL FETCH emp_cursor INTO …

EXEC SQL CLOSE emp_cursor;

由关动态 SQL 方法，请参考动态 SQL 技术章节。

3.2.4 并行登录

ORACLE 系统能通过 SQL * Net 支持分布处理。应用程序能对当地数据库、远程数据库、或二者的任意组合进行并行存取，也能对同一个数据库进行多个连接。在图 3—1 中，应用程序与一个当地的和三个远程的 ORACLE 数据库进行通讯。其中 ORA2、ORA3 和 ORA4 是在 CONNECT 语句中使用的简单逻辑名。

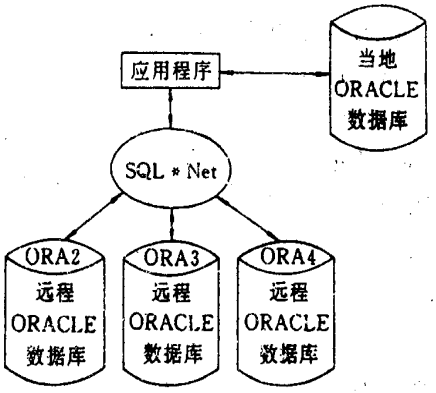


图 3-1 通过 SQL * Net 的连接

1. 多显式登录

类似单显式登录，也能把 AT 子句用于多显示登录。下面的例子是并行连接到两个非缺省数据库上：

```
/* SQL 变量说明 */
EXEC SQL BEGIN DECLARE SECTION;
        char username[10];
        char password[10];
        char db_name1[20];
        char db_name2[20];
EXEC SQL END DECLARE SECTION;
...
strcpy(username, "scott");
strcpy(password, "tiger");
strcpy(db_name1, "d:newyork_nondef1");
strcpy(db_name2, "d:chicago_nondef2");
/* 给每个数据库命名一个链接名 */
EXEC SQL DECLARE db_link_name1 DATABASE;
EXEC SQL DECLARE db_link_name2 DATABASE;
/* 连接到这两个数据库上 */
EXEC SQL CONNECT :username IDENDIFIED BY :password
            AT db_link_name1 USING :db_name1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
            AT db_link_name2 USING :db_name2;
```

未加说明的标识符 db_link_name1 和 db_link_name2 用于命名两个非缺省节点上的缺省数据库，这样此后的 SQL 语句就能通过该名字引用数据库了。

也能在 AT 子句中使用宿主变量，下例是连接到 N 个数据库上的例子：

```
/* SQL 变量说明 */
EXEC SQL BEGIN DECLARE SECTION;
        char username[10];
        char password[10];
        char db_name[20];
        char db_link_name[10];
EXEC SQL END DECLARE SECTION;
...
/* 初始化 username 和 password */
strcpy(username, "scott");
strcpy(password, "tiger");
for(i=1; i<=N; i++)
{
```

```
    /* 给下一个数据库一个链接名 */
    printf("Database link-Name? \n");
    scanf("%s", db_link_name);
    /* 给出下一个数据库的名字 */
    printf("SQL*Net string? \n");
    scanf("%s", db_name);
    /* 连接到这个数据库上 */
    EXEC SQL CONNECT :username IDENTIFIED BY :password
                AT :db_link_name USING :db_name;
}
...
```

其中 N 是非缺省数据库个数。

也能使用这个方法对同一个数据库做多个连接，如下例所示：

```
strcpy(username, "scott");
strcpy(password, "tiger");
strcpy(db_name, "d:newyork_nondef");
for(i=1; i<=N; i++)
{
    /* 给出数据库的下一个链接名 */
    printf("Database Name? \n");
    scanf("%s", db_link_name);
    /* 再次连接到该数据库上 */
    EXEC SQL CONNECT :username IDENDIFIED BY :password
            AT :db_link_name USING :db_name;
}
```

对上述的连接，虽然使用了相同的 SQL*Net 串，但也必须使用不同的数据库链接名。

当事务操纵两个或多个远程数据库上的数据时，应用程序必须保证这些事务的完整性。因此程序必须提交或回滚事务中的所有 SQL 语句。例如，假设一个应用程序是在两个账目数据库上操作，把借方账记入一个数据库，而把贷方账记入另一个数据库。然后对每一个数据库发一个 COMMIT。保证提交或回滚这两个事务是该应用程序的责任。

2. 隐式登录

ORACLE 的分布查询机构支持 隐式登录，因为分布查询只支持 SELECT 语句，故不需要显式登录。分布查询允许单个 SELECT 语句存取一个或多个非缺省数据库上的数据。

分布查询机构依赖于数据库链，数据库链把一个名字分配给一个 CONNECT 语句而不是链本身。在运行时，嵌入的 SELECT 语句通过指定的 ORACLE Server 执行，ORACLE Server 隐式地连接到该非缺省数据库，以取得所需要的数据。

(1)单隐式登录

下边的例子是连接单个非缺省数据库。程序首先执行下面的句子，以定义一个数据库链(数据库链通常是通过 DBA 或用户交互建立的)：

```
EXEC SQL CREATE DATABASE LINK db_link
        USING 'd:newyork_nondef';
```

然后，程序就能用该数据库链来查询非缺省 EMP 表。

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
        FROM emp@db_link
        WHERE DEPTNO=:dept_number;
```

数据库链与嵌入 SQL 语句的 AT 子句中所用的数据库名字无关。它只是告诉 ORACLE 非缺省数据库位于什么地方、访问它的路径、使用的 ORACLE 用户名和口令等。数据库链在显式地删除之前，一直保存在数据词典中。

为了便于引用数据库链，可建立一个同义词(这一般可用交互方式做)，如：

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link;
```

然后，程序就能查询该非缺省 EMP 表。

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
        FROM emp
        WHERE DEPTNO= :dept_number;
```

这对 emp 提供了位置透明。

(2)多隐式登录

下面的例子并行连接到两个非缺省数据库上。它首先执行下列语句，以定义两个数据库链和建立两个同义词：

```
EXEC SQL CREATE DATABASE LINK db_link1
        CONNECT TO username1 IDENTIFIED BY password1
            USING 'd:newyork_nondef';
EXEC SQL CREATE DATABASE LINK db_link2
        CONNECT TO username2 IDENTIFIED BY password2
            USING 'd:chicago_nondef';
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1;
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

然后，程序就能查询该非缺省 EMP 和 DEPT 表：

```
EXEC SQL SELECT ENAME, JOB, SAL, LOC
    FROM emp, dept
    WHERE emp.DEPTNO=dept.DEPTNO AND dept. DEPTNO= :dept_number;
```

ORACLE 通过完成 db_link1 上的非缺省 EMP 表与 db_link2 上的非缺省 DEPT 之间的连接来执行该查询。

§3.3 插入、更新和删除

本节和下节将重点说明嵌入 SQL 程序设计的基本概念、特征和方法，并给出几个程序实例。通过这两节学习，将使读者学会使用基本嵌入式 SQL 语句(如 INSERT、UPDATE、

DELETE 和 SELECT …)来编写 PRO＊C 程序的技术。有关 SQL 语句的文法请参考附录，在此不再赘述。

3.3.1 数据插入应用程序

数据插入是经常要进行的数据库操作，完成数据插入所用的嵌入 SQL 语句是 INSERT 语句。利用 INSERT 语句一次可以插入一行，也可一次插入多行。下面是两个 INSERT 语句的书写例子。

一次插入一行：

```
EXEC SQL INSERT
        INTO EMP (EMPNO,ENAME, JOB,SAL)
        VALUES(:emp_number, :emp_name, :job, :salary);
```

一次插入多行：

```
EXEC SQL BEGIN DECLARE SECTION;
    char emp_name[100][20];
    float salary[100];
    int insert_rows;
EXEC SQL END DECLARE SECTION;
...
insert_rows=50;     /* 希望插入 50 行 */
EXEC SQL FOR: insert_rows
    INSERT INTO EMP(ENAME, SAL)
        VALUES(:emp_name, :salary);
```

下面是两个数据插入应用实例：

例 3.1 利用循环向 EMP 表插入若干行。

```
/*************************************
*  说明：
*       程序登录后，首先提示用户输入插入行的各列值，
*       然后插入。
*       如果输入的职员号(emp_number)是零，则程序结束。
*************************************/
#include <stdio.h>
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userid[20];
    VARCHAR password[20];
    int emp_number;
    VARCHAR emp_name[15];
    VARCHAR job[10];
```

```
    float salary;
    int dept_number;
EXEC SQL END DECLARE SECTION;
/* 说明 SQLCA */ ,
EXEC SQL INCLUDE SQLCA;
/* 程序     */
main()
{
    int retcode;                        /* scanf 函数返回码 */
    /* 登录 ORACLE */
    strcpy(userid.arr, "SCOTT");              /* 初始化用户名 */
    userid.len=strlen(userid.arr);
    strcpy(password.arr, "TIGER");             /* 初始化口令 */
    pwd.len=strlen(password.arr);
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT: userid IDENTIFIED BY: password;
    printf("Connected to ORACLE as user: %s \n\n\n", userid.arr);
    /* 插入处理     */
    while(1)
    {
        /* 输入 职员号、名字、职务和工资 */
        printf("Enter employee number(or 0 to end):");
        retcode=scanf("%d", &emp_number);
        if(retcode==EOF || retcode==0 || emp_number==0)
            break;                  /* 退出循环 */
        printf("Enter employee name:");
        scanf("%s", emp_name.arr);
        emp_name.len=strlen(emp_name.arr) /* 设置长度 */
        printf("Enter employee's job:");
        scanf("%s", job.arr);
        job.len=strlen(job.arr);     /* 设置长度 */
        printf("Enter employee salary:");
        scanf("%f", &salary);
        printf("Enter employee dept_number:");
        scanf("%d", &dept_number);
        /* 插入一行      */
        EXEC SQL INSERT INTO EMP
            (EMPNO, ENAME, JOB, SAL, DEPTNO)
            VALUES(:emp_number, :emp_name, :job, :salary, :dept_number);
```

```
        /* 提交插入行 */
        EXEC SQL COMMIT WORK;
        printf("Employee %s added. \n\n", emp_name.arr);
    }
    /* 退出数据库 */
    EXEC SQL COMMIT WORK RELEASE;
    exit();
}
```

在执行该程序时,如果 SQL 语句发生错误,则结束插入处理。

例 3.2 使用数组一次向 EMP 表插入多行

```
/*************************************
*  说明:
*       程序登录后,先从 EMP1 表中检索多行然后插入 EMP 表中。
*       如果执行 SQL 语句时发生错误,则进行错误处理。
*************************************/
#include <stdio.h>
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
            VARCHAR userid[20];
            VARCHAR password[20];
            int dept_number;
            int insert_rows;
            int emp_number[100];
            VARCHAR emp_name[100][20];
            VARCHAR job[100][50];
            float salary[100];
EXEC SQL END DECLARE SECTION;
/* 说明 SQLCA */
EXEC SQL INCLUDE SQLCA;
/* 程序 */
main()
{
    /* 登录 */
    strcpy(userid.arr, "SCOTT");
    userid.len=strlen(userid.arr);
    strcpy(password.arr, "TIGER");
    password.len=strlen(password.arr);
```

```
    EXEC SQL WHENEVER SQLERROR DO sqlerror(); /* 错误处理说明 */
    EXEC SQL CONNECT :userid IDENTIFIED BY :password;
    printf("Connected to ORACLE as user: %s\n", userid.arr);

    printf("Enter DEPTNO? \n");
    scanf("%d", &dept_number);
    EXEC SQL WHENEVER NOT FOUND CONTINUE; /* 错误处理说明 */
    /* 从 EMP1 表查询 */
    EXEC SQL SELECT EMPNO, ENAME, JOB, SAL
          INTO :emp_number, :emp_name, :job, :salary
          FROM EMP1
          WHERE DEPTNO=:dept_number;
    insert_rows=sqlca.errd[2];     /* 处理行数，至多 100 行 */ ……(1)
    /* 向 EMP 表插入若干行，至多 100 行 */
    EXEC SQL FOR :insert_rows INSERT
          INTO EMP(EMPNO, ENAME, JOB, SAL)
          VALUES(:emp_number, :emp_name, :job, :salary);
    /* 结束处理,退出 ORACLE */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0)
}

/* 错误处理 */
sqlerror()
{
    /* 为了避免错误处理 时发生死循环,应给出此说明 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n ORACLE error detected:");
    printf("\n%s", sqlca.sqlerrm.sqlerrmc); /* 打印错误信息文本 */
    /* 使数据库恢复到插入之前状态 */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

几点说明：

(1) 在向一个表插入时，用户必须有向该表插入的特权。如果没有此特权，则必须赋予他对该表插入的特权。否则不能插入。

(2) FOR 子句

该子句设置要处理的数组元素个数。它适于 DELETE、EXECUTE、FETCH、INSERT、

OPEN 和 UPDATE 等语句，特别是对 INSERT、DELETE、UPDATE 语句非常有用。当不想对整个数组进行操作时，可用 FOR 子句限制要处理的元素数，使之正好满足所需要的元素个数。例如在上例中，如果只想把查询出的开始 30 行插入 EMP 表，可把(1)语句改为：

```
insert_rows=30;
```

FOR 子句中的 :insert_rows 一定是宿主变量，不允许是常量，如下例是非法的：

```
EXEC SQL FOR 30
    INSERT INTO(ENAME, SAL)
    VALUES(:emp_name, :salary)
```

而且宿主变量中设置的值一定小于或等于最小数组的维数，并大于 0。否则将产生错误信息，且没有行被处理。

(3) 使用数组注意事项

出现在 WHERE 子句中的宿主变量要不全是标量，要不全是数组，二者不能混用。如果它们全是标量，ORACLE 执行 INSERT 语句一次。如果它们全是数组，则 ORACLE 对于每一组数组元素执行 INSERT 语句一次。

WHERE 子句中的数组大小(维数)可以不同，在这种情况下，ORACLE 执行语句的次数由下面两种方法之一决定：

- 没有 FOR 子句时，由较小的数组维数决定。
- 有 FOR 子句时，由其后的宿主变量决定。

3.3.2 数据更新应用程序

更新数据库表中的数据也是常用的操作，用 UPDATE 语句实现更新操作。例如语句

```
EXEC SQL UPDATE EMP
        SET SAL=:salary:ind_sal, COMM=:commission:ind_comm
        WHERE EMPNO=:emp_number;
```

更新 EMP 表中一条记录。而

```
EXEC SQL BEGIN DECLARE SECTION;
        char job_title[10][20];
        float commission;
EXEC SQL END DECLARE SECTION;
        ...
EXEC SQL UPDATE EMP
        SET COMM=:commission
        WHERE JOB=:job_title;
```

更新 EMP 表中一组记录。下面是进行更新的应用实例。

例 3.3 修改职员的附加工资

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明:
 *        它首先登录,然后提示用户输入职员号、附加工资,
```

```
*         最后更新该职员的附加工资。
*         在本例中使用了指示器变量处理 Null 值。
**********************************/
#include <stdio.h>
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userid[20];
    VARCHAR password[20];
    VARCHAR emp_name[15];
    int emp_number;
    float commission;
    short ind_comm;          /* 定义指示器变量 */
EXEC SQL END DECLARE SECTION;
/* 说明 SQL 通讯区 */
EXEC SQL INCLUDE SQLCA;
main()
{
    int retcode;          /* scanf 返回码 */
    /* 登录 ORACLE */
    strcpy(userid.arr, "SCOTT"); /* 初始化用户名 */
    userid.len=strlen(userid.arr);
    strcpy(password.arr, "TIGER"); /* 初始化口令 */
    password.len=strlen(password.arr);
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL WHENEVER NOT FOUND STOP;
    EXEC SQL CONNECT :userid IDENTIFIED BY :password;
    printf("Connected to ORACLE user:%s\n", userid.arr);
    /* 输入职员号 */
    printf("Enter employee number to update:");
    scanf("%d", &emp_number);
    /* 查询并打印该职员的名字、附加工资等 */
    EXEC SQL SELECT ENAME, COMM INTO :emp_name, :commission
        FROM EMP
        WHERE EMPNO=:emp_number;
    printf("Employee:%s commission: %6.2f\n",emp_name.arr,
            commission);
    /* 输入新的附加工资 */
    printf("Enter new commission:");
    retcode=scanf("%f", &commission);
```

```
    ind_comm=0;                    /* 设置指示变量 */
    if (retcode==EOF || retcode==0)
        ind_comm=-1;               /* 设置指示器变量为 -1 */
    /* 更新该职员的附加工资 */
    EXEC SQL UPDATE EMP
        SET COMM=:commission:ind_comm
        WHERE EMPNO=:emp_number;
    printf("Employee %s updated. \n", emp_name.arr);
    EXEC SQL COMMIT WORK RELEASE;
    exit()
}
```

编写更新程序时应注意的几点：

(1) 用户在进行更新时，必须有更新的特权。

(2) 在 WHERE 子句和 SET 子句中所用的宿主变量必须或者都是标量，或者都是数组，不能二者混合使用。如果是标量，ORACLE 仅执行 UPDATE 语句一次。如果是数组，ORACLE 对于每一组数组元素执行一次，每次执行可更新 0 行、一行或多行。

数组宿主变量可以有不同的维数(大小)，此时 ORACLE 执行 UPDATE 语句的次数可由以下两种方式之一决定：

· 在没有 FOR 子句的情况下，由较小的数组维数决定。

· 在有 FOR 子句的情况下，由 FOR 子句中的宿主变量决定。

当 MODE=ANSI14 时，不允许使用数组。

(3) 累计修改的行数在 SQLCA 的 SQLERRD[2]中返回。

如果没有满足条件的行被更新，则在 SQLCA 的 SQLCODE 元素中返回错误信息。

如果省略 WHERE 子句，则更新表中所有行，且在 SQLCA 的 SQLWARN[4]中返回警告标志。

3.3.3 数据删除应用程序

删除数据库表中的行是经常要处理的操作，完成删除行操作的语句是 DELETE 语句。下面是 PRO*C 程序中经常使用的 DELETE 语句例子。

简单删除

```
EXEC SQL DELETE FROM EMP
    WHERE DEPTNO=:dept_number AND JOB=:job;
```

用光标删除

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT EMPNO, COMM
        FROM EMP
        WHERE DEPTNO=:dept_number
        FOR UPDATE OF COMM;
```

```
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_sursor INTO :emp_number, :commission;
EXEC SQL DELETE FROM EMP
    WHERE CURRENT OF emp_cursor;
```

使用数组进行删除：

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    int emp_number[10];
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DELETE FROM EMP
    WHERE EMPNO=:emp_number; /* 。对应于每个数组元素删除一行 */
```

使用数组删除：

```
EXEC SQL BEGIN DECLARE SECTION;
    int dept_number[10];
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DELETE
    FROM EMP
    WHERE DEPTNO=:dept_number; /* 对于每个数组元素删除多行 */
```

下面 是数据删除应用程序的例子：

例 3.4 从 EMP 表中删除一行

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明：
*        该例子是给定一个职员名，从 EMP 表中删除一个职员记录
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include<stdio.h>
/* 说明 SQL 变量和 SQL 通讯区 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userid[20];
    VARCHAR password[20];
    VARCHAR emp_name[15];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;

main()
{
```

```
    /* 登录 ORACLE */
    strcpy(userid.arr, "SCOTT");  /* 初始化用户名 */
    userid.len=strlen(userid.arr);
    strcpy(password.arr, "TIGER");  /* 初始化口令 */
    password.len=strlen(password.arr);
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT :userid IDENTIFIED BY :password;
    printf("Connected to ORACLE user: %s\n", userid.arr);
    /* 输入一个职员名 */
    printf("Enter employee name to delete:");
    scanf("%s", emp_name.arr);
    emp_name.len=strlen(emp_name.arr);
    /* 删除该职员记录 */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerr;
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    EXEC SQL DELETE FROM EMP WHERE ENAME=:emp_name;
    /* 提交变更,退出数据库 */
    EXEC SQL COMMIT WORK RELEASE;
    printf("Employee name %s dropped.\n", emp_name.arr);
    exit(0)
    /* 没找到要删除的记录 */
    notfound:
    printf("WARNING: employee name %s does not exist.\n",emp_name.arr);
    exit(1)
    /* 执行 SQL 语句时发生错误 */
    sqlerr:
        printf("%70S\n", sqlca.sqlerrm.sqlerrmc);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK;
        Exit(1);
}
```

进行删除时应注意的事项:

(1)当用户从一个表中删除行时,他应该有对此表进行删除的特权。

(2)WHERE 子句中的宿主变量必须全是标量、或者全是数组。如果全是标量,则 ORACLE 仅执行一次 DELETE 语句。如果全是数组,则 ORACLE 对每一组数组元素执行 DELETE 语句一次。每次执行可能删除 0 行、1 行或多行。

WHERE 子句中的数组宿主变量能有不同的大小,在这种情况下,ORACLE 执行 DELETE 语句的次数由下面两种方法之一决定:

· 在没有 FOR 子句的情况下，由较小的数组维数决定。

· 在有 FOR 子句的情况下，由 FOR 子句中的宿主变量决定。

(3) 如果没有满足该条件的行被删除，则返回“NOT FOUND”错误码。

在执行 DELETE 时，SQLCA 的 SQLERRD[2]中返回删除的累计行数。如果省略 WHERE 子句，则在 SQLCA 的 SQLWARN[4]中设置警告标志。

§ 3.4 查询应用程序

数据库查询分为单行数据查询和多行数据查询。前者叫简单查询，后者叫多行查询。无论哪一种查询，都要使用 SELECT 语句。

3.4.1 简单查询

语句

```
EXEC SQL SELECT EMPNO,ENAME, SEX, OLD
     INTO :emp_number, :emp_name, :sex, :old
     FROM EMP
     WHERE EMPNO=65147;
```

是返回单行的查询。简单查询的完整例子如第 1 章所示，此处不再另举。

查询时的注意事项：

(1)用户必须有查询有关信息的权限。

(2)如果没有检索到满足条件的行时，则 ORACLE 返回一个“no data found”错误码。

3.4.2 数据操作的综合例子

为了进一步说明 PRO * C 程序的编写方法，本节给出一个包含数据插入、更新、删除和查询的综合例子。在此我们可以进一步看到一个具有多个函数的 PRO * C 程序的结构。该例中包含有多个说明段，其中有一个全程说明段和四个局部说明段。

例 3.5 数据操作的综合例子

该程序的模块结构图如图 3-2 所示。

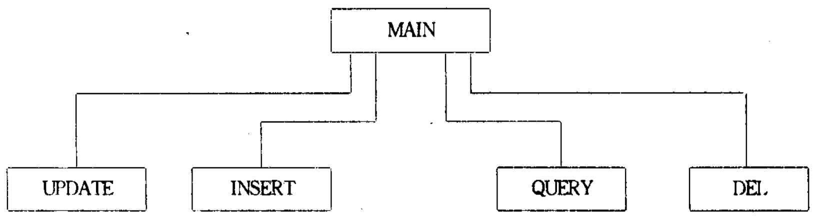


图3-2 模块结构图

```
/*************************************
*  说明：
*         本程序包含数据插入、更新、删除和查询等功能。
*         程序启动后，显示一个菜单，
*         选择需要的功能。并进行相应的处理。
```

```
*************************************/
#include <stdio.h>
/* 全程说明段 */
EXEX SQL BEGIN DECLARE SECTION;
    VARCHAR username[20];
    VARCHAR password[20];
EXEC SQL END DECLARE SECTION;
/* 说明通讯区 */
EXEC SQL INCLUDE sqlca;
/* 外部函数说明 */
void insert(), update(), del(), query();
/* 总控程序 */
main()
{
    char operate[5];
    /* 登录到 ORACLE 上 */
    strcpy(username.arr, "SCOTT");
    username.len=strlen(username.arr);
    strcpy(password.arr, "TIGER");
    password.len=strlen(password.arr);

    EXEC SQL WHENEVER SQLERROR GOTO logon_error;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n",username.arr);

    for(; ;)
    {
        /* 建立菜单 */
        print("\n****************[****");
        print("\n*                              *");
        print("\n*      1. Query                *");
        print("\n*                              *");
        print("\n*      2. Update               *");
        print("\n*                              *");
        print("\n*      3. Insert               *");
        print("\n*                              *");
        print("\n*      4. Delete               *");
        print("\n*                              *");
        print("\n*      5. Exit                 *");
```

```
        print("\n* * * * * * * * * * * * * * * * * * * * *");
        /* 输入操作员选择：选择查询时，输入 1；选择更新时，输入 2；…… */
        print("\n Enter selection: \n");
        gets(operate);
        /* 执行所选择的操作 */
        switch(operate[0])
        {
            case '1': query();
                      break;
            case '2': update();
                      break;
            case '3': insert();
                      break;
            case '4': del();
                      break;
            case '5': break;
            default:  printf("\n\n invalid selection\n");
                      break;
        }
        if(operate[0] =='5'
          break;
}
/* 结束处理 */
EXEC SQL COMMIT RELEASE;
printf("\n\n very good ! \n\n");
exit(0);

/* 登录错误处理 */
logon_error:
printf("\n invalid username/password\n");
printf("\n%.70s \n", sqlca.splerrm.sqlerrmc);
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK RELEASE;
exit(1);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明：
 *        该函数提示输入职员号，根据职员号查询该职员，然后更新它
```

```
 ************************************/
void update()
 {
        EXEC SQL BEGIN DECLARE SECTION;
            int emp_number;
            VARCHAR emp_name[20];
            VARCHAR job[50];
            short ind_job;
        EXEC SQL END DECLARE SECTION;

        char empnum[8];
    /* 输入职员号 */
    printf("\n\nEnter employee number (press RETURN to abort):");
    gets(empnum);
    if(! strcmp(empnum, ""))
    {
        printf("\n");
        return;
    }
    emp_number = atoi(empnum);
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    /* 查询该职员信息 */
    EXEC SQL SELECT ENAME, JOB
        INTO :emp_name, :job:ind_job
        FROM EMP
        WHERE EMPNO = :emp_number;
    emp_name.arr[emp_name.len] = '\0';
    switch (ind_job)
    {
        case -1: strcpy(job.arr, "NULL");
            job.len = strlen(job.arr);
            break;
        case 0: job.arr[job.len] = '\0';
            break;
        default: if(ind_job > 0)
            printf("\n\nWARNING: Job truncated.\n");
            break;
    }
```

```
    printf("\n\nNumber Employee Name Job\n");
    printf(" -------  ----------  --------\n");
    printf("%-9d%-15s%-8s\n", emp_number, emp_name.arr, job.arr);
    printf("\nEnter new employee name: ");
    /*输入新的职员信息 */
    gets(emp_name.arr);
    emp_name.len = strlen(emp_name.arr);
    printf("\n\nEnter new job: ");
    gets(job.arr);
    job.len = strlen(job.arr);
    ind_job = 0;
    if (! strcmp(job.arr, ""))
    ind_job = -1;
    /* 修改该职员信息 */
    EXEC SQL UPDATE EMP
        SET ENAME = :emp_name, JOB = :job:ind_job
        WHERE EMPNO = :emp_number;
    printf("\n\nEmployee %d updated. \n", emp_number);
    EXEC SQL COMMIT;
    return;

/* 没查到处理 */
notfound:
    printf ("\n\nWARNING: Employee %d does not exist. \n",emp_number);
    return;
/* 错误处理 */
sqlerror:
    printf("\n%.70s \n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(1);

} /* 更新结束 */

/**********************************
 *  说明:
 *       该函数提示输入职员号,根据职员号查询该职员,
 *       如果它存在,则停至插入; 如果它不存在,则输入职员信息;
```

```
*          然后插入它。
*********************************************/

void insert()
{
    /* 说明段 */
    EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
        VARCHAR emp_name[20];
        VARCHAR job[20];
        short ind_job;
    EXEC SQL END DECLARE SECTION;

    /* 输入职员号 */
    char empnum[8];
    printf("\n\nEnter employee number (press RETURN to abort):");
    gets(empnum);
    if(! strcmp(empnum, " "))
    {
        printf("\n");
        return;
    }
    emp_number = atoi(empnum);
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

    EXEC SQL WHENEVER NOT FOUND GOTO resume;

    /* 查询该职员信息 */
    EXEC SQL SELECT ENAME INTO :emp_name
        FROM EMP
        WHERE EMPNO = :emp_number;
    printf("\n\nWARNING: Employee %d already exists.\n",
        emp_number);
    return;

/*输入新的职员信息 */
resume:
    printf("\n\nEnter employee name: ");
    gets(emp_name.arr);
```

```
    emp_name.len = strlen(emp_name.arr);

    printf("\n\nEnter job: ");
    gets(job.arr);
    job.len = strlen(job.arr);
    ind_job = 0;
    if(! strcmp(job.arr, ""))
        ind_job = -1;

    /* 插入该职员信息 */
    EXEC SQL INSERT INTO EMP(EMPNO, ENAME, JOB)
        VALUES(:emp_number, :emp_name, :job:ind_job);

    /* 结束处理 */
    printf("\n\nEmployee %d inserted. \n", emp_number);
    EXEC SQL COMMIT;
    return;

/* 错误处理 */
sqlerror:
    printf("\n% .70s\n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(1);

} /* 插入结束 */

/*********************************
 *  说明:
 *       该函数提示输入职员号,根据职员号查询该职员,
 *       如果它不存在,则打印警告; 如果它存在,则输出该职员信息。
 *
 *********************************/
void query()
{
    /* 说明段 */
    EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
        VARCHAR emp_name[20];
```

```
        VARCHAR job[20];
        short ind_job;
    EXEC SQL END DECLARE SECTION;
    char empnum[8];

    /* 输入职员号 */
    printf("\n\nEnter Employee number (press RETURN to abort): ");
    gets(empnum);
    if(! strcmp(empnum, " "))
    {
        printf("\n");
        return'
    }

    /* 查询该职员信息 */
    emp_number = atoi(empnum);
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    EXEC SQL SELECT ENAME, JOB
        INTO :emp_name, :job:ind_job
        FROM EMP
        WHERE EMPNO = :emp_number;

    /* 打印该职员信息 */
    emp_name.arr[emp_name.len] = '\0';
    switch(ind_job)
    {
        case -1: strcpy(job.arr, "NULL");
            job.len = strlen(job.arr);
            break;
        case 0: job.arr[job.len] = '\0';
            break;
            default: if(ind_job > 0)
            printf("\n\nWARNING: Job truncated. \n");
            break;
    }
    printf("\n\nNumber Employee Name Job\n");
    printf("------ ---------- ---------\n");
    printf("%-9d%-15s%-8s\n",emp_number, emp_name.arr, job.arr);
```

```
    return;

notfound:
    printf(\n\nWARNING: Employee %d does not exist. \n",
        emp_number);
    return;

/* 错误处理 */
sqlerror:
        printf("\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK;
        exit(1);
} /* 查询结束 */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明:
 *        该函数提示输入职员号,根据职员号删除该职员,
 *        如果它不存在,则打印警告。
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void del()
{

    /* 说明段 */
    EXEC SQL BEGIN DECLARE SECTION;
        int emp_number;
    EXEC SQL END DECLARE SECTION;
    char empnum[8];

    /* 输入职员号 */
    printf("\n\nEnter employee number (press RETURN to abort):");
    gets(empnum);
    if(! strcmp(empnum, " "))
    {
        printf("\n");
        return;
    {
    emp_number = atoi(empnum);
```

```
    /* 删除该职员 */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    EXEC SQL DELETE
        FROM EMP
        WHERE EMPNO = :emp_number;
    printf("\n\nEmployee %d deleted.\n",
        emp_number);
    /* 结束处理 */
    EXEC SQL COMMIT;
    return;

/* 没找到 */
notfound:
    printf ("\n\n WARNING: Employee %d does not exit. \n",emp_number);
    return;

/* 错误处理 */
sqlerror:
    printf("\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(1);
} /* 删除结束 */
```

3.4.3 利用数组实现返回多行的查询

在实际应用中，经常迁到返回多行查询。实现返回多行查询有两种方法：数组和光标。如果一个查询操作返回的行数或行数的范围已知时，可通过在 SELECT 语句的 INTO 子句中使用数组 SQL 变量来实现返回多行的查询。如果它们都不知道时，就没法在说明段说明数组的维数。如果说明的数组维数小于实际返回的行数，则执行 SELECT 语句时就出现如下错误：

```
RTL-2112 SELECT..INTO returns too many rows
例如
    EXEC SQL BEGIN DECLARE SECTION;
        char emp_name[100][20];
        float salary[100];
        float commission[100];
    EXEC SQL END DECLARE SECTION;
```

```
...
EXEC SQL SELECT ENAME, SAL,COMM
    INTO :emp_name, :salary, :commission
    FROM EMP
    WHERE SAL> 500.00;
```

该 SELECT 至多只能返回 100 行。如果满足选择条件的行不超过 100 行，或者用户只希望检索开始 100 行时，可用这种方式处理。如果超过 100 行，例如 150 行，则不能用这种方式检索。因为它只能返回开始 100 行。如果再次执行该 SELECT 语句时，仍返回开始 100 行。对于这种情况，必须把数组维数说明得足够大。下面是使用数组进行查询的完整例子。

例 3.6 使用数组完成返回多行的查询

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明:
*        该程序提示输入部门号,根据部门号查询该部门的
*        职员信息,如果该部门的职员不存在,则打印警告;
*        否则输出该部门的职员信息。
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include stdio.h
/* 说明段 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR userid[20];
    VARCHAR password[20];
    char emp_name[100][15];
    char job[100][30];
    int emp_number[100];
    float salary[100];
    int dept_number;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
main()
{ int n, i;
    /* 登录到 ORACLE */
    strcpy(userid.arr,"SCOTT");
    userid.len=strlen(userid.arr);
    strcpy(password.arr,"TIGER");
    pwd.len=strlen(password.arr);
EXEC SQL WHENEVER SQLERROR goto_error;
EXEC SQL WHENEVER NOT FOUND GOTO notfound;
EXEC SQL CONNECT :userid IDENTIFIED BY :pasword;
```

```
    printf("Connected to ORACLE user:%s \n",userid.arr);
    /* 输入部门号 */
    print("\nEnter dept number:");
    scanf("%d",&dept_number);
    /* 查询     */
     EXEC SQL SELECT EMPNO,ENAME, JOB,SAL
        INTO :emp_number,:emp_name, :job,:salary
        FROM EMP
        WHERE DEPTNO= :dept_number;
    /* 打印 */
    n=SQLCA.SQLERRD[2];
    for(i=0; i<n; i++)
        printf("%d %-10s\t\t%-10s\t\t%6.2f\n",emp_number[i],
            emp_name[i],job[i],sal[i]);
    /* 结束处理 */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
/* 没找到 */
notfound:
    printf("\n\n WARNING: Employee does not exit. \n");
    return;

/* 错误处理 */
sqlerror:
    printf("\n% .70s \n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(1);
} /* 查询结束 */
```

假定上例选择的总行数不超过100。因为SQLCA.SQLERRD[2]中返回查询的总行数，所以可根据该数打印全部选择行。

使用注意：

(1) 仅当MODE={ANSI|ANSI13|ORACLE}时，SELECT语句才允许使用数组。只要INTO子句中的宿主变量其中有一个是数组，则其它必须都是数组。宿主数组可以有不同的维数，此时能够处理的数组元素个数由最小的数组元素个数决定。不允许在WHERE子句中使用数组。

(2)选择的累计行数在SQLCA的SQLERRD[2]中返回。

(3) SELECT语句中不允许使用FOR子句。

3.4.4 用光标实现返回多行的查询

从上面的讨论可知：进行多行查询时，仅当 MODE＝{ANSI|ANSI13|ORACLE}时，SELECT 语句才允许使用数组；而且，在使用数组查询时，如果返回的行数不能正确估计时，就不便使用数组。此时可使用光标语句。

用光标语句实现多行查询的基本思想是开辟一个缓冲区(或叫光标缓冲区)，先把满足查询条件的行(或记录)都检索到该缓冲区中，并使光标指向该缓冲区中的第一行；然后再从该缓冲区中一次一行或多行的取出进行处理。

用光标进行检索的步骤是：

- 说明一个光标，并使它与一个 SELECT 语句或 PL/SQL 块相关联。
- 打开该光标，把满足查询条件的行(或记录)都检索到该光标缓冲区中；
- 提取：从光标缓冲区中一次一行或多行的取出数据，进行处理，直至取完；
- 关闭光标，释放缓冲区等。

因此，使用光标查询要用到以下四条语句：

- DECLARE CURSOR：命名一个光标，并使它与一个 SELECT 语句或 PL/SQL 块相关联。
- OPEN：打开光标，把满足查询条件的行都检索到该光标缓冲区中，使光标指向该缓冲区中的第一行；
- FETCH：从光标缓冲区中提取行；
- CLOSE：关闭光标，释放空间

(1) 说明一个光标

用 DECLARE CURSOR 语句说明一个光标。例如语句

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ENAME, EMPNO, SAL
        FROM EMP
        WHERE DEPTNO=:dept_number;
```

说明了以个名为 emp_cursor 的光标。

DECLARE CURSOR 语句命名一个光标，并使它与一个 SELECT 语句或 PL/SQL 块相联系。在引用光标之前应先说明它，光标说明的作用范围在一个预编译单位内是全程的，而且名字在该范围内是唯一的。在一个编译单位内不能说明两个同名光标。

DECLARE CURSOR 语句是说明性语句，它必须物理地领先所有其它光标语句。

全部光标控制语句(DECLARE、OPEN、FETCH 和 CLODE)必须出现在同一个预编译单位内，不能在一个单位内说明，而在另一个单位引用。

可在一个 PRO＊C 程序内说明多个不同名的光标。

(2) 打开光标

用 OPEN 语句打开一个光标；例如

```
EXEC SQL OPEN emp_cursor;
```

打开一个名为 emp_cursor 的光标。

OPEN 语句具体完成以下几方面的工作：

·定义一个光标缓冲区，

·把满足查询条件的行都检索到该光标缓冲区中，

·使光标指向该缓冲区中的第一行；

·行处理计数器(SQLCA.SQLERRD[2])清0。

(3)提取

用FETCH语句从光标缓冲区中一次提取一行或多行记录。例如

EXEC SQL FETCH emp_cursor INTO :emp_number,emp_name, :salary;

该语句把光标(emp_cursor)所指向的行取出，并存入输出SQL变量(如emp_number,…)中。

在执行FETCH之前，必须首先用OPEN语句打开光标。允许在INTO子句中使用数组变量，也可用FOR子句来限制数组维数。如果FETCH语句从光标缓冲区中读取的列值是NULL，则相应的指示器变量置-1。在第一次执行FETCH语句之前，光标指向光标缓冲区中的第一行，每执行一次光标做相应的移动，使其指向下次被取的行。

如果INTO子句中的输出宿主变量是标量，则每执行一次FETCH，从光标缓冲区中提取一行。如果输出宿主变量是数组，则执行一次FETCH所提取的行数可由如下方法之一决定：

·如果有FOR子句，则由其后的变量的值决定。

·如果省略FOR子句，则由最小的数组维数决定。数组和标量不能混用。当MODE=ANSI14时，不允许使用数组。

当光标缓冲区中最后一行被提取时，下一个FETCH语句将导致错误，在SQLCA.SQLCODE中返回"NOT FOUND"错误码，且光标指向光标缓冲区中最后一行。

FETCH语句中不包含AT子句。因此，必须在DECLARE CURSOR语句中指明与光标相联系的数据库(除非是缺省数据库)。

光标只能在光标缓冲区中向前移动，如果要想使它指向后面已FETCH过的行，就必须重新打开光标，执行适当次数的FETCH才行。

(4)关闭光标

用CLOSE语句来关闭光标。例如

```
EXEC SQL CLOSE emp_cursor;
```

关闭emp_cursor光标。

例3.7 光标应用举例

```
/*************************************
*  说明：
*        该程序先登录到ORACLE上，然后说明和打开一个光标，
*        提取一个部门中的所有职员的职员号、名字、工作、工资和
*        附加工资，显示结果，最后关闭光标。
*************************************/
#include <stdio.h>
/* 用户定义数据类型 */
```

```
typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;
/* 定义 char 等价于以 Null 结尾的串类型 */
EXEC SQL TYPE asciz IS STRING(20);
    asciz username[20];
    asciz password[20];
    asciz emp_name[20];
    asciz job[20];
    float salary;
    float commission;
    int emp_number;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE sqlca;
void sqlerror(); /* 错误处理函数 */

main()
{
    /* 登录到 ORACLE. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");
    EXEC SQL WHENEVER SQLERROR DO sqlerror();
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n",username);
    /* 说明一个光标 emp_cursor */
    EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT EMPNO,ENAME,JOB, SAL, COMM
            FROM EMP
            WHERE DEPTNO=1010;
    /* 打开光标 emp_cursor */
    EXEC SQL OPEN emp_cursor;

    printf("\nNumber Name Job Salary Commission\n");
    printf("------ -------- -------- ------ -------\n");
    /* 循环提取一个部门中的所有职员信息 */

    for (;;)
    {
        EXEC SQL WHENEVER NOT FOUND DO break;
```

```
            EXEC SQL FETCH emp_cursor
                INTO :emp_number,:emp_name,:job, :salary, :commission;

            printf("%d%-11s%-11s%9.2f%13.2f\n",emp_number, emp_name,
                job,salary, commission);
        }
        /* 关闭光标 */
        EXEC SQL CLOSE emp_cursor;
        /* 结束处理 */
        printf("\nHave a good day.\n");

        EXEC SQL COMMIT WORK RELEASE;
        exit(0);
}

/* 错误处理 */
void sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\noracle error detected:\n");
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

在说明段使用了数据类型等价，使 char 和外部数据类型 STRING 等价。于是 asciz 类型的串均为以 Null 结尾的串。

例 3.8 批检索

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明:
*        程序定义和打开一个光标，然后用数组进行批检索，
*        并用函数 print_rows()打印结果。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#inciude <stdio.h>
/* 用户定义数据类型 */
typedef char asciz;
/* 说明段 */
EXEC SQL BEGIN DECLARE SECTION;
```

```
    /* 把 char 定义为与 STRING 等价的数据类型. */
    EXEC SQL TYPE asciz IS STRING(20);
    asciz username[20];
    asciz password[20];
    asciz emp_name[10][20];
    asciz job[10][20];
    int emp_number[10];
    float salary[10];
    float commission[10];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;

void print_rows(); /* 输出函数 */
void sqlerror(); /* 错误处理函数 */

main()
{
    int row_num; /* 已处理的总行数 */
    /* 登录到 ORACLE. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");
    EXEC SQL WHENEVER SQLERROR DO sqlerror();
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n",username);
    /* 说明一个光标 */
    EXEC SQL DECLARE c1 CURSOR FOR
        SELECT EMPNO,ENAME, JOB, SAL,COMM
            ROM EMP;
        /* 打开光标 */
        EXEC SQL OPEN c1;
        /* 初始化 row_num */
        row_num = 0;

        /* 使用数组循环提取,每次提取 10 行,提取完结束 */
        for (; ;)
        {
            EXEC SQL WHENEVER NOT FOUND DO break;
            EXEC SQL FETCH c1
```

```
            INTO :emp_number,:emp_name,:job :salqry,:commission;
        /* 打印 10 行 */
        print_rows(sqlca.sqlerrd[2] - row_num);
        /* 已处理的总行数 */
        row_num = sqlca.sqlerrd[2];
    }

    /* 打印余下的几行 */
    if((sqlca.sqlerrd[2] - row_num) > 0)
    print_rows(sqlca.sqlerrd[2] - row_num);
    /* 关闭光标 */
    EXEC SQL CLOSE c1;

    printf("\nHave a good day. \n");
    /* 结束处理 */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

/*输出函数*/
void print_rows(n)
int n;
{
    int i; /* 记录打印行数 */

    printf("\nNumber Employee Job Salary commission\n");
    printf("------- ------ ------ ----- -----\n");

    for(i = 0; i < n; i++)
        printf("%-9d%-8s%-11s%9.2f%9.2f\n",emp_number[i],emp_
            name[i],job[i],salary[i],commission[i]);
}

/* 错误处理函数*/
void sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\nORACLE error detected:\n");
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
```

```
        EXEC SQL ROLLBACK WORK RELEASE;
        exit(1);
    }
```

例 3.9 数据类型等价

本例使用了数据类型等价，使程序能使用 ORACLE 的外部数据类型 LONG RAW 来表示映象。

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明：程序的处理过程是：
 *        ·程序登录到 ORACLE 上，
 *        ·建立一个名为 IMAGE 的肖像表，
 *        ·把学生的二进制位图映象插入到 IMAGE 表中，
 *        ·输入一个学号，
 *        ·根据该学号，把他的二进制位图映象从 IMAGE 表中检索出来，
 *        ·在终端上显示该 二进制位图映象。
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdio.h>

#define NON_EXISTENT -942

typedef char asicz;
typedef char bitmap;

/* 说明段    */
EXEC SQL BEGIN DECLARE SECTION;
    /* 定义 asciz 等价于 STRING */
    EXEC SQL TYPE asciz IS STRING(20);
    asciz username[20];
    asciz password[20];
    int student_number;
    asciz student_name[20];
    asciz sex[5];
    int old;
    asciz location[20];
    /* 定义 bitmap 等价于 LONG RAW */
    EXEC SQL TYPE bitmap IS LONG RAW(8192);
    bitmap buffer;
EXEC SQL END DECLARE SECTION;
```

```
/*说明通讯区 */
EXEC SQL INCLUDE sqlca;

/* 说明外部函数 */
void getimage(); /* 取一个职员的肖像      */
void showimage(); /* 在屏幕上显示该肖像 */
void signoff(); /* 正常结束处理 */
void sqlerror(); /* 错误处理 */

main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        int studentno_sel;
    EXEC SQL END DECLARE SECTION;
    char reply[10];

    /* 登录到 ORACLE. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n", username);

    /* 是否删除原 IMAGE 表 ? */
    printf
    ("\nAbout to drop the IMAGE table -- OK [Y/N]?");
    gets(reply);
    printf("\n");
    if(reply[0] != 'Y' && reply[0] != 'y')
    {
        /* 不删除原 IMAGE 表, 结束处理 */
        printf("\nExiting ...\n");
        signoff();
    }
    /* 删除原 IMAGE 表 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
```

```
EXEC SQL DROP TABLE IMAGE;

if(sqlca.sqlcode == 0)
{
        printf("\nTable IMAGE has been dropped ");
        printf(" creating new table.\n");
}
else if (sqlca.sqlcode == NON_EXISTENT)
{
    printf("\nTable IMAGE does not exist");
    printf("- creating new table.\n");
}
else
        sqlerror();

/* 重新建立 IMAGE 表 */
EXEC SQL WHENEVER SQLERROR DO sqlerror();
EXEC SQL CREATE TABLE IMAGE
(STUDENTNO NUMBER(4) NOT NULL, BITMAP LONG RAW);

    /* 对于每一个学生,调用 getimage() 在 buffer 中产生其二进制位图映像 */
EXEC SQL DECLARE student_cursor CURSOR FOR
    SELECT STUDENTNO, SNAME FROM STUDENT;

EXEC SQL OPEN student_cursor;

printf
("\nINSERTing bitmaps into IMAGE for all students…\n\n");

for (;;)
{
    /* 在 STUDENT 表中,查询学生 */
    EXEC SQL WHENEVER NOT FOUND DO break;

    EXEC SQL FETCH student_cursor INTO :student_number, :student_name;

    printf("Student %-8s\n", student_name);

    /* 调用 getimage() 在 buffer 中产生其二进制位图映像 */
```

```
    getimage(student_number, buffer);

    /* 把产生的二进制位图映像插入 IMAGE 表中 */
    EXEC SQL INSERT INTO IMAGE VALUES (:student_number,:buffer);

    printf("is done! \n");
}

EXEC SQL CLOSE student_cursor ;
EXEC SQL COMMIT WORK;

/* 显示每个人的映像 */
printf
    ("\nDone INSERTing bitmaps, Next, display some.\n");

while(1)
{
    /* 输入一个学号 */
    studentno_sel = 0;
    printf("\nEnter student number (0 to quit):");
    scanf("%d", &studentno_sel);
    printf("\n");
    if(studentno_sel == 0)
      signoff();
    /* 根据该学号在 IMAGE 中查询其信息 */
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;

    EXEC SQL SELECT STUDENT.STUDENTNO, SNAME,
            SEX,OLD,LOC,BITMAP
        INTO :student_number,:student_name,:sex,:old,:location,:buffer
        FROM STUDENT,IMAGE
        WHERE STUDENT.STUDENTNO = :studentno_sel
        AND STUDENT.STUDENTNO = IMAGE.STUDENTNO;
    /* 显示该映像图 */
    showimage(buffer);

    printf("\nStudent %-.8s",student_name);
    printf("\nSex %s", sex);
    printf("Old%d\n", old);
```

```
        printf("\nLocation%-11s",location);

        continue;

notfound:
        printf("\nNot a valid student number - try again.\n")
    }
} /* ends main */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明:
 *        调用该函数,在 buf 中产生一个映像图。该函数是一个模仿
 *        函数,它产生的映像图是一个假想的映像图。
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void getimage (studentnum, buf)
int studentnum;
bitmap buf[8192];
{
    int i;
    /* 模仿产生一个映像图 */
    for (i = 0; i < 8192; i++)
    {
        buf[i] = (char) ((i * studentnum) % 256);
        if((i % 256) == 0) printf(".");
    }

}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *  说明:
 *        调用该函数,把 buf 中的映像图显示在屏幕上,
 *        该函数是一个模仿函数,它显示的映像图是一个假想的,
 *        并非 getimage()所产生的映像图。
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void showimage (buf)
bitmap buf[8192];
{
        int i;

        /* 模仿显示映像图 */
```

```
        for (i = 0; i < 10; i++)
            printf("\n\t\t* * * * * * * * * *");
        printf("\n");
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明:
*        调用该函数,结束处理
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void signoff()
{
        printf("\nHave a good day. \n");
        EXEC SQL COMMIT WORK RELEASE;
        exit(0);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明:
*        调用该函数,打印错误信息,退出 ORACLE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void sqlerror()
{

        EXEC SQL WHENEVER SQLERROR CONTINUE;

        printf("\nORACLE error detected:\n");
        printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);

        EXEC SQL ROLLBACK WORK RELEASE;
        exit(1);
}
```

§3.5 在 PRO*C 程序中嵌入 PL/SQL

3.5.1 嵌入 PL/SQL 块的优点

PL/SQL 语言是 SQL 语言的扩充，它具有过程性和结构性。在 pro*c 程序中可以嵌入 PL/SQL 块，嵌入 PL/SQL 块有如下优点：

(1) 改进性能

嵌入 PL/SQL 块能帮助改进性能。在使用嵌入 SQL 语句时，系统一次传送一个 SQL 语句给服务器。这样，对服务器的每次调用只能执行一个 SQL 语句。在使用嵌入 PL/SQL 块时，系统一次传送一个完整的块给服务器。这样，对服务器的每次调用就能执行一组 SQL 语句。这就使应用程序和 ORACLE 的通讯减至最少。

(2) 与 ORACLE 紧密集成

使用 PL/SQL 的另一个优点是与 ORACLE Server 紧密集成。例如，大多数 PL/SQL 数据类型对 ORACLE 数据字典来说都是其自身的数据类型。此外，还能在数据字典内的列定义上用%TYPE 属性建立变量说明。如下例所示：

```
job_title EMP.JOB%TYPE;
```

这样就不需要精确知道列的数据类型。而且，如果列的定义改变，则变量的说明也就跟着自动变化。这就提供了数据的独立性，降低了维护代价，并使程序适应数据库的变化。

(3) 光标 FOR 循环

如果使用 PL/SQL 的话，就不需要使用 DECLARE、OPEN、FETCH 和 CLOSE 语句来定义和操纵一个光标，而是使用一个光标 FOR 循环。例如：

```
DECLARE
    ...
BEGIN
    FOR emprec IN (SELECT empno, sal, comm FROM emp)LOOP
        IF emprec.comm/emprec.sal>0.25 THEN…
        ...
    END LOOP;
END;
```

例中的点(·)符号表示引用记录中的字段

(4) 过程和函数

PL/SQL 块有过程和函数两类子程序。一般来说，用一个过程来完成一个操作，而用一个函数计算一个值。过程和函数为应用开发提供了方便，它可让开发者自由裁剪 PL/SQL 语言，以适应其需要。下面是建立一个新部门的过程：

```
PROCEDURE create_dept
    (new_dname IN CHAR(14),
     new_loc IN CHAR(13),
     new_deptno OUT NUMBER(2)) IS
BEGIN
```

```
        SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
        INSERT INTO dept VALUES(new_deptno, new_dname, new_loc);
    END create_dept;
```

该过程接收一个新的部门名和地址,在部门号数据库序列中选择下一个值作为新的部门号。然后把新的部门号、名字和地址插入 dept 表中,并把新的部门号返回给调用者。

形式参数有三种状态:IN(缺省)、OUT 和 INOUT。IN 参数把值传递给正调用的子程序;OUT 参数把值返回给子程序的调用者;INOUT 参数传递初始值给正要调用的子程序,而把更新的值返回给调用者。

每一个实参的数据类型必须是可转换为相应形式参数的数据类型。

(5) 包装

PL/SQL 允许把逻辑相关的类型、程序对象和子程序封装成一个包装。可把包装编译并存放在 ORACLE 数据库中,以备多个应用共享,以提高开发效率,增强可维护性。

通常,包装包括两部分:说明和体。说明是应用的界面,它说明类型、常量、变量、例外、光标和可以应用的子程序。包体定义光标和子程序等。如下例:

```
/* 包装说明 */
PACKAGE emp_actions IS
    PROCEDURE hire_employee (empno NUMBER, emame CHAR,…);
    PROCEDURE fire_employee(emp_id NUMBER);
END emp_actions;
/* 包体 */
PACKAGE BODY emp_actions IS
    /* 过程定义 */
    PROCEDURE hire_employee(empno NUMBER, ename CHAR,…)IS
    BEGIN
        INSERT INTO emp VALUES (empno, ename,…);
    END hire_employee;
    PROCEDURE fire_employee(emp_id NUMBER) IS
    BEGIN
        DELETE FROM emp WHERE empno=emp_id;
    END fire_employee;
END emp_actions;
```

包装说明中的说明对应用是可见的和可访问的。包体中的实现细节是隐藏的和不可访问的。

(6) PL/SQL 表

PL/SQL 提供了一个命名为 TABLE 的组合数据类型。类型 TABLE 的对象叫 PL/SQL 表,这些表被用来模拟数据库表。可用类似数组的存取方法来存取 PL/SQL 表的行。

可在任何块、过程、函数或包装的说明部分说明 PL/SQL 表类型。下例说明一个叫 NumTabTyp TABLE 类型:

```
…
```

```
EXEC SQL EXECUTE
    DECLARE
        TYPE NumTabTyp IS TABLE OF NUMBER
            INDEX By BENARY_INTGER;
        ...
    BEGIN
      ...
    END
END_EXEC
  ...
```

一旦定义了类型 NumTabTyp，就能说明该类型的 PL/SQL 表，如下句所示：

```
num_tab NumTabTyp;
```

标识符 num_tab 表示一个 PL/SQL 表。

使用类似数组的方法来引用 PL/SQL 表中的行。例如，引用 PL/SQL 表 num_tab 的第 9 行的格式如下：

```
num_tab(9)…
```

(7) 能使用 RECORD 数据类型

可使用%ROWTYPE 属性来说明一个记录，以表示表中的一行。但是，用户不能为该记录中的字段指定数据类型或定义用户所拥有的字段。为此 PL/SQL 又提供了一个组合数据类型 RECORD。

类型 RECORD 的对象叫记录，一个记录是由若干字段所组成。例如，职员记录是由名字、工资、雇用日期等组成。这些字段虽然数据类型不同，但在逻辑上是相关的，因此可把它们作为一个逻辑单位来处理。

能在任何一块、过程、函数或包装的说明部分来说明记录类型或对象。下例说明了一个叫 DeptRecTyp 的 RECORD 类型：

```
DECLARE
    TYPE DeptRecTyp IS RECORD
        (deptno NUMBER(4) NOT NULL,
         dname CHAR(9),
         loc CHAR(14));
```

从上例中看出，字段说明类似变量说明。每一个字段都有唯一的名字和特定的数据类型。对任何一个字段说明都能加 NOT NULL 可选项，从而避免对该字段的 NULL 赋值。

一旦定义了一个 DeptRecTyp 类型，就能以如下形式说明该类型的记录：

```
dept_rec DeptRecTyp;
```

标识符 dept_rec 表示一个记录变量。

可用园点(·)符来引用记录变量中的个别字段。例如，引用 dept_rec 记录中的 dname 字段的格式如下：

```
dept_rec.dname…
```

3.5.2 嵌入 PL/SQL 块的方法

ORACLE 预编译程序处理 PL/SQL 块方法类似于处理单个 SQL 语句。因此，在 PRO *C 程序中凡是能放置 SQL 语句的地方，都能放置一个 PL/SQL 块。但是，PL/SQL 块必须用关键字 EXEC SQL EXECUTE 和 END-EXEC 括住。如下例所示：

```
EXEC SQL EXECUTE
    DECLARE -- PL/SQL 块
    ...
    BEGIN
    ...
    END;
END-EXEC;
```

其中 END-EXEC 之后应该有 C 语句的终结符(;)。

在预编译时，可通过指定 SQLCHECK 可选项来对 PL/SQL 块进行语法和语义检查。

3.5.3 使用宿主变量

同嵌入 SQL 语句一样，在嵌入式 PL/SQL 块中也可引用宿主变量和指示器变量。宿主变量是 C 语言与 PL/SQL 块通讯的关键，宿主变量能由 C 语言和 PL/SQL 共享。

例如，用户可从终端输入有关信息，程序用宿主变量接收该信息，并把它传递给 PL/SQL 块，然后 PL/SQL 块把该宿主变量中的信息存入数据库。另外，还可用宿主变量从数据库表中检索信息，并把它传递给宿主程序。

(1) 宿主变量的引用

在 PL/SQL 块内，把宿主变量作为整个块的全程量来处理，凡是能允许 PL/SQL 变量的地方都能使用宿主变量。与 SQL 语句中的宿主变量类似，PL/SQL 块中的宿主变量也需要用冒号开头。冒号使宿主量与 PL/SQL 变量和数据库对象区分开。

下面是两个包含嵌入式 PL/SQL 块的例子，其中都使用了宿主变量。

例 3.10 嵌入式 PL/SQL 的应用举例之一

该例说明在有 PL/SQL 块情况下宿主变量的用法。

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明：
*        程序提示用户输入部门号，然后显示该部门的名字、地址和
*        职员人数。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdio.h>
#include <string.h>
/* 说明段 */
typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;
    /* 定义 asciz 和 STRING 等价 */
```

```
    EXEC SQL TYPE asciz IS STRING(20);
    asiz username[20];
    asiz password[20];
    int dept_number;
    char dept_name[50];
    char location[50];
    char creat_date[9];
    int emp_sum_number;
EXEC SQL END DECLARE SECTION;
/* 说明通讯区 */
EXEC SQL INCLUDE SQLCA;
main()
{
    /* 登录到 ORACLE */
    strcpy (username,"SCOTT");
    strcpy (password, "TIGER");
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    EXEC SQL CONNECT :username IDENTIFIED By :password;
    printf("\n connected to ORACLE");

    while(1)
    {
        /* 输入部门号 */
        printf("\n Enter department Number (0 to end)?");
        Scanf("%d", &dept_number);
        if (dept_number==0 )
        {
            EXEC SQL COMMIT WORK RELEASE;
            printf("\n Exiting program");
            exit(0);
        {
        -------- PL/SQL 块开始 --------
        EXEC SQL EXECUTE
            BEGIN        --查询部门信息
              SELECT DNAME,LOC,CDATE,DNUMBER
                INTO :dept_name, :location, :creat_date, :emp_sum_number
                FROM DEPT
                WHERE DEPTNO=:dept_number;
            END;
```

```
    END_EXEC;
    -------- PL/SQL 块结束 --------
    /* 打印所查询的信息 */
    printf("\n Dept No. \t Dept Name\t Location \tCreate Date\tEmp Samary");
    printf("\n.....");
     printf ("\n %d %s %s%s %d", dept_number, dept_name,location,creat_
          date, emp_sum_number);
}
/* 错误处理 */
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    printf ("\n processing error!");
    printf ("\n Exiting program!");
    exit(1);
}
```

在此例中，在进入 PL/SQL 块之前，应设置宿主变量 dept_number；而宿主变量 dept_name，location，creat_date，emp_sum_number 是在 PL/SQL 块内设置的。

(2) VARCHAR 变量的引用

在嵌入式 PL/SQL 块中也可使用 VARCHAR 变长字符串，其方法类似嵌入 SQL 语句，但唯一不同点是：当 VARCHAR 变量是一个输出宿主变量时，在进入该块之前应初始化其长度字段，即把长度字段设置为 VARCHAR 变量的说明长度。如下例所示：

```
EXEC SQL BEGIN DECLARE SECTION;
    int dept_number;
    VARCHAR dept_name[50];
    ...
EXEC SQL END DECLARE SECTION;
    ...
    dept_name.len=50; /* 初始化长度字段 */
    -------- PL/SQL 块开始 --------
    EXEC SQL EXECUTE
        BEGIN
            SELECT DNAME
                INTO :dept_name
                FROM DEPT
                WHERE DEPTNO= :dept_number;
            ...
        END;
```

```
END-EXEC;
--------- PL/SQL 块结束 ---------
...
```

(3) 指示器变量的引用

尽管 PL/SQL 能够操纵 Null 值，可以不需要指示器变量。但是，因为宿主语言不能操纵 Null 值，所以还必须需要指示器变量。使用指示器变量可完成下列工作：

- 从宿主程序接收 Null 值存入数据库表列中
- 把 Null 值或截短值输出给宿主程序

在 PL/SQL 块中使用指示器变量时，应服从以下规则：

- 必须把指示器变量附在相关的宿主变量之后。
- 如果在 PL/SQL 块的某处把宿主变量与指示器变量一起引用，则在同一块内必须总是以这种方式来引用。

在下例中，指示器变量 ind_sal 与其宿主变量 salary 在 SELECT 语句中一起出现，因此，也必须以这种方式出现在 IF 语句中：

```
...
EXEC SQL EXECUTE
    BEGIN
        SELECT ENAME,SAL
            INTO :emp_name, :salary:ind_sal
            FROM EMP
            WHERE EMPNO= :emp_number;
        IF :salary:ind_sal IS NULL THEN…
        ...
    END;
END-EXEC;
```

注意：PL/SQL 处理 :salary:ind_sal 就象处理其它简单变量一样。当进入 PL/SQL 块时，如果指示器变量的值是－1，则 PL/SQL 自动把 Null 值赋给相应的宿主变量。当退出该块时，如果宿主变量是 Null 值，则 PL/SQL 自动把－1 赋给相应的指示器变量。在下例中，如果在进入 PL/SQL 块之前 ind_sal 的值是－1，则发生 Salary_missing 例外：

```
...
EXEC SQL EXECUTE
    BEGIN
        IF :Salary:ind_sal IS NULL THEN
            RAISE Salary_missing;
        END IF;
        ...
    END;
END-EXEC;
```

...

当把一个截短值赋给一个宿主变量时，PL/SQL 不发生例外。但是，如果使用指示器变量的话，PL/SQL 把它设置为原始串的长度。在下例中，宿主程序通过检查 ind_name 的值将能够知道是否把截短值赋给 emp_name 了：

```
...
EXEC SQL EXECUTE
    DECLARE
        ...
        new_name CHAR(10);
    BEGIN
        ...
        :emp_name: ind_name :=new_name;
        ...
    END;
END_EXEC;
```

(4) 使用宿主数组

能把输入宿主数组和指示器数组变量传递给 PL/SQL 块。在 PL/SQL 块中能用 BINARY_INTEGER 类型的 PL/SQL 变量或与其类型相兼容的宿主变量来检索它们。可把整个宿主数组传递给 PL/SQL 块(缺省规定)，也能用 ARRAYLEN 语句来指定较小的数组维数。

另外，还能通过调用一个过程来把宿主数组中的所有值赋给 PL/SQL 表的行。假定数组下标的范围是 m..n，则相应的 PL/SQL 表的索引范围总是 1..n−m+1。例如，如果数组下标的范围是 15..25，则相应 PL/SQL 表的索引范围是 1..11。

在下例中，把宿主数组 commission 传递给 PL/SQL 块。在 PL/SQL 块的函数调用中又引用该宿主数组。函数名为 median，它求出一系列数的中间值。其形式参数包括一个名为 num_tab 的 PL/SQL 表。函数调用把实参 commission 中的所有值赋给形参 num_tab 的行。

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    float commission[100];
EXEC SQL END DECLARE SECTION;
  ..........
/* 填充数组 commission */
  ..........

/* PL/SQL 块 */
EXEC SQL EXECUTE
    DECLARE
        /* 说明 PL/SQL 表 */
        TYPE NumTabTyp IS TABLE OF REAL
```

```
        INDEX BY BINARY_INTEGER;
    /* 说明 PL/SQL 变量 */
    median_comm REAL;
    n BINARY_INTEGER;
    ...
    /* 函数说明与定义 */
    FUNCTION median(num_tab NumTabTyp, n INTEGER)
      RETURN REAL IS
    BEGIN
    ..........          /*计算中间值*/
    END;
BEGIN
    n:=100;
    median_comm:=median(:commission,n); /* 函数调用 */
    ...
END;
END_EXEC;   /* PL/SQL 结束 */
...
```

也可用过程调用把 PL/SQL 表中的所有行值赋给一个数组的相应元素。

表 3-3 给出 PL/SQL 表中的行值和宿主数组元素之间的合法转换。例如，类型为 LONG 的宿主数组是与类型为 VARCHAR2、LONG、RAW 或 LONG RAW 的 PL/SQL 表是兼容的，但与类型为 CHAR 的 PL/SQL 表是不兼容的。

表 3-3 合法数据类型转换

PL/SQL Table	VARCHAR2	NUMBER	LONG	ROWID	DATE	RAW	LONG RAW	CHAR
VARCHAR2	↵		↵			↵	↵	
NUMBER		↵						
LONG	↵		↵			↵	↵	
ROWID				↵				
DATE					↵			
RAW	↵		↵			↵	↵	
LONG RAW	↵		↵			↵	↵	
CHAR								↵

当 PL/SQL 块只要求处理数组的一部分时，可用 ARRAYLEN 语句来指定较小的数组维数。ARRAYLEN 语句的格式如下：

EXEC SQL ARRAYLEN host_array (host_integer);

其中 host_integer 是 4 字节的整型宿主变量。ARRAYLEN 语句必须在说明段与要限定的数组说明同时出现，且在它的说明之后。下面是使用 ARRAYLEN 指定数组 salary 维数的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
    float salary[100];
    int m;
    EXEC SQL ARRAYLEN salary(m);
EXEC SQL END DECLARE SECTION;
    ...
/* 填充数组 salary */
    ...
m=30;
/* 指定数组维数 */
/* PL/SQL 块开始 */
EXEC SQL EXECUTE
    DECLARE
        /* 说明 PL/SQL 表 */
        TYPE NumTabTyp IS TABLE OF REAL
            INDEX BY BINARY_INTEGER;
        /* 说明 PL/SQL 变量 */
        median_salary REAL;
        /* 函数的说明与定义 */
        FUNCTION median(num_tab NumTabTyp, n INTEGER)
            RETURN REAL IS
        BEGIN
        ......          /*计算中间值 */
        END;
    BEGIN
        median_salary:=median (:salary,:m);
        ...
    END;
END_EXEC;
```

在此例中,因为 ARRAYLEN 把数组的元素个数从 100 降至 30,所以只有 30 个数组元素被传递给 PL/SQL 块。于是,当 PL/SQL 块被传递给 ORACLE 执行时,传送了一个更小的数组。这不仅节约时间,而且在网络环境下也降低了网络传输的信息量。

3.5.4 使用光标

对于嵌入 PL/SQL 块,管理光标缓冲存储器的处理类似于 SQL 语句。在运行时,一个称之为双亲的光标与整个 PL/SQL 块相联系。光标缓冲存储器上有一个相应的项,该项被连到 PGA 中的专用 SQL 区上。

PL/SQL 块内的每一个 SQL 语句也要求一个 PGA 中的专用 SQL 区。因此,对于这些 SQL 语句,PL/SQL 还要管理一个独立的高速缓冲存储器,叫子女光标高速缓冲存储器。其

光标称为子女光标。因为 PL/SQL 管理该子女光标高速缓冲存储器，所以不能用程序直接控制子女光标。

程序能同时使用的最大光标数由 ORACLE 初始化参数 OPEN_CURSORS 来设置。可用如下的方法计算光标的最大数：

最大光标数＝SQL 语句光标＋PL/SQL 双亲光标＋PL/SQL 儿子光标＋6 个开销光标。

最大光标数一定不能超过 OPEN_CURSORS 所设置的数，如果超出该限制，将出现如下错误：

```
ORA－01000: maximum open cursors exceeded
```

可通过指定 RELEASE_CURSOR＝YES 可选项来避免该错误。如果不想指定 RELEASE_CURSOR＝YES 可选项的话，则可在每个 PL/SQL 块后把该可选项再设置为 NO。例如

```
EXEC ORACLE OPTION (RELEASE_CURSOR＝YES);
    －－ 1st embedded PL/SQL block
EXEC ORACLE OPTION (RELEASE_CURSOR＝NO);
    －－ embedded SQL Statements
EXEC ORACLE OPTION (RELEASE_CURSOR＝YES);
    －－ second embedded PL/SQL block
EXEC ORACLE OPTION (RELEASE_CURSUR＝NO);
    －－ embedded SQL Statements
...
```

3.5.5 存储子程序

对于 PL/SQL 子程序(过程或函数)，能分别进行编译，并把生成的目标代码存放在 ORACLE 数据库中，供各个应用调用。由 ORACLE 工具(如 SQL＊PLUS)或 SQL＊DBA 显式建立的子程序叫存储子程序。一但把它们编译和存储在数据字典中，它们就成为数据库中的对象，以后就可直接执行。

当把一个 PL/SQL 块内的子程序或存储过程通过应用发送给 ORACLE 时，把它叫做联机子程序。ORACLE 编译联机子程序并把它储存在系统全局区(SGA)中，但不把源代码或目标代码存储在数据字典中。

在包装中定义的子程序被认为是包装的一部分，因而也叫包装子程序。不在包装内定义的存储子程序叫独立子程序。

(1)建立存储子程序

可通过嵌入 SQL 语句 CREATE FUNCTION、CREATE PROCEDURE 和 CREATE PACKAGE 来在宿主程序中建立独立子程序(过程或函数)和包装子程序。

下面是用 CREATE FUNCTION 建立一个存储函数的例子：

```
EXEC SQL CREATE
    FUNCTION old_ok (old INTEGER, departement CHAR)
        RETURN BOOLEAN AS
```

```
            min_old INTEGER;
            max_old INTEGER;
        BEGIN
            SELECT MINOLD, MAXOLD INTO min_old, max_old
                FROM OLDY
                WHERE DEPTNO=department;
            RETURN (old>=min_old) AND
                (old<=max_old);
        END old_ok;
    END_EXEC;
```

注意：

嵌入 CREATE{FUNCTION|PROCEDURE|PACKAGE}语句是以 EXEC SQL 开始(不是 EXEC SQL EXECUTE)，和以 PL/SQL 终结符 END_EXEC 结尾。

下面是建立包装子程序的例子，它用 CREATE PACKAGE 来建立存储包装的说明，用 CREATE PACKAGE BODY 建立存储包装体。

例 3.11 建立包装子程序

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明: *
*        包装名:emp_action
*        包装内包含的过程名:get_employees
*            过程的形式参数:包含三个说明为 OUT 的 PL/SQL 表
*        调用参数:与形参相匹配的实参是宿主数组
*        过程功能:从 emp 表中提取若干行,行数由调用者决定。
*                把一批职员数据检索到 PL/SQL 表中,
*                当过程完成时,自动把 PL/SQL 表中的行值
*                赋给宿主数组的相应元素。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 建立存储包装说明 */
EXEC SQL CREATE OR REPLACE PACKAGE emp_action AS
        /* PL/SQL 表说明 */
        TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
            INDEX BY BINARY_INTEGER;
        TYPE NumArrayTyp IS TABLE OF INTEGER
            INDEX BY BINARY_INTEGER;
        /* 过程说明 */
        PROCEDURE get_employees(
            dept_number IN INTEGER,
            batch_size IN INTEGER,
```

```
        found IN OUT INTEGER,
        done_fetch OUT INTEGER,
        emp_name OUT CharArrayTyp,
        address OUT CharArrayTyp,
        tex OUT NumArrayTyp);
    END emp_actions;
END_EXEC;

/* 定义存储包体 */
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS
    /* 说明光标     */
    CURSOR get_emp (dept_number IN INTEGER) IS
        SELECT ENAME, ADDRESS,TEX FROM EMP
            WHERE DEPTNO=dept_number;
    /* 过程定义 */
    PROCEDURE get_employees(
        dept_number IN INTEGER,
        batch_size IN INTEGER,
        found IN OUT INTEGER,
        done_fetch OUT INTEGER,
        emp_name OUT CharArrayTyp,
        address OUT CharArrayTyp,
        tex OUT NumArrayTyp) IS

    BEGIN     /*定义过程体 */
        /* 如果光标没有打开,则打开 */
        IF NOT get_emp % ISOPEN THEN
            OPEN get_emp(dept_number);
        END IF;
        done_fetch :=0;
        found :=0;
        /* 提取 batch_size 行 */
        FOR i IN 1..batch_size LOOP
            FETCH get_emp INTO emp_name(i),
                address(i), tex(i);
            IF get_emp % NOTFOUND THEN
                CLOSE get_emp;
                done_fetch :=1;
                EXIT;
```

```
            ELSE
                found:=found+1;
            END IF;
        END LOOP;
    END get_employees; /* 过程定义结束 */
END emp_actions; /* 定义存储包体结束 */
END_EXEC;
```

其中 OR REPLACE 子句用于重新定义和重新建立已存在的包装(不需事先删除它)、并再授与它特权。

如果嵌入 CREATE{FUNCTION |PROCEDURE| PACKAGE}语句失败,ORACLE 就产生一个警告(而不是错误)信息。

(2) 存储子程序的引用

必须在 PRO*C 程序的无名 PL/SQL 块内引用存储子程序,下面是调用一个独立过程的例子:

```
EXEC SQL EXECUTE
    BEGIN
        .....
        get_dept (:deptno);
    END;
END_EXEC;
```

存储子程序能使用参数,其中 deptno 是宿主变量。

在下面的例子中,若过程 get_dept 被存放在一个包装 dept_actions 中,这时必须用点(.)字符来限定过程调用:

```
EXEC SQL EXECUTE
    BEGIN
        ......
        dept_actions.get_dept (:deptno);
    END;
END_EXEC;
```

实 IN 参数可以是文字串、宿主变量、宿主数组、PL/SQL 常数或变量、PL/SQL 表、PL/SQL 用户定义记录、过程调用或表达式。但是,实 OUT 参数不能是文字串、过程调用或表达式。

例如 3.12 存储过程的应用

```
/*************************************
*  说明:
*        在该例中,三个形式参数是 PL/SQL 表,对应的实参是宿主
*        数组,该程序重复调用存储过程 get_employees,显示每一批
*        职员数据直至检索结束。
```

```
*************************************/
#include <stdio.h>
#include <string.h>

/* 说明SQL变量 */
typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;
    /* 定义等价数据类型 */
    EXEC SQL TYPE asciz IS STRING(20);
    asciz username[20]
    asciz password[20];
    int dept_number; /* 要查询的部门号 */
    char emp_name [10][21];
    char address[10][21];
    int tex[10];
    int done_flag; /* 循环结束标志 */
    int array_size;
    int ret_num; /* 返回的行数 */
    int SQLCODE;
EXEC SQL END DECLARE SECTION;
/* 说明通讯区 */
EXEC SQL INCLUDE sqlca;

/* 说明函数 */
int print_rows();    /* 输出函数 */
int sqlerror();     /* 错误处理函数 */

main()
{
    int i;
    /*登录到ORACLE. */
    strcpy (username, "SCOTT");
    strcpy (password, "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sqlerror();

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user:%s\n",username);
    /* 提示:输入部门号 */
```

```
    printf("enter department number:");
    scanf("%d",&dept_number);
    fflush(stdin);
    /*设置数组大小*/
    array_size=10;
    done_flag=0;      /* 循环结束标志:非0结束 */

    ret_num=0;
    /* 当 NOT FOUND 是真时,循环结束 */
    for (;;)
    {
        /* PL/SQL */
        EXEC SQL EXECUTE
            BEGIN
                /* 过程调用 */
                emp_actions.get_employees
                    (:dept_number, :array_size, :ret_num,
                    :done_flag, :emp_name, :address, :tex);
            END;
        END_EXEC;
        /* 输出 */
        print_rows(ret_num);
        if (done_flag)
                break;     /* 循环结束 */
    }
    /*退出 ORACLE */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

/*************************************
*  说明:输出n行.
*************************************/
print_rows(n)
int n;
{
    int i;
    if (n==0)
    {
```

```
        printf("No rows retrieved. \n");
        return;
    }
    /* 输出要输出的行数 */
    printf "\n\nGot %d row%c\n",n, n==1? '\0': 's');
    /* 输出标题 */
    printf("%-20.20s%-50.50s%s\n","Ename","Address","Tex");
    for (i=0;i<n;i++)
        printf("%-20.20s%-50.50s%d\n",
            emp_name[i], address[i], tex[i]);
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*   说明:错误处理
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf ("\nORACLE error detected:");
    printf("\n%.70s \n", sqlca. sqlerrm. sqlerrmc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1)
}
```

在子程序调用中,应注意实参的数据类型必须能转换为相应形参的数据类型。在退出存储过程之前,所有的 OUT 形参必须被赋值。否则,相应的实参值不确定。

PL/SQL 允许通过数据库链访问远程数据库。一般来说,数据库链是由 DBA 建立,并存放在数据字典中。数据库链告诉 ORACLE 远程数据库设置在什么地方、它的路径、以及所用的 ORACLE 用户名和口令。下面是使用数据库链 dallas 来调用过程 raise_salary 的例子:

```
EXEC SQL EXECUTE
    BEGIN
        raise_salary @ dallas(:emp_id, :increase);
    END;
END_EXEC;
```

可建立同义词,以提供远程子程序的位置透明度。如下例所示:

```
CREATE PUBLIC SYNONYM raize_salary
    FOR raise_salary @ dallas;
```

3.5.6 使用动态 SQL 方法

ORACLE 预编译程序对一个完整 PL/SQL 块的处理类似单个 SQL 语句。于是,能把一

个 PL/SQL 块存放在一个串宿主变量中。如果块中不包含宿主变量，就能用动态 SQL 方法 1 来执行 PL/SQL 串。如果块中包含已知个数的宿主变量，则能用动态方法 2 来分析执行 PL/SQL 串。如果块中包含未知个数的宿主变量，就必须用方法四。关于动态 SQL 方法的详细介绍，请参考动态 SQL 方法一章。

§3.6 错误处理

错误处理是应用程序的重要组成部分。应用程序在运行时出现的错误是由多种原因造成的，它可能是设计的错误、编码错误、硬件错误、无效的输入数据…。错误处理不可能把所有可能发生的错误都考虑到，只能考虑那些对程序影响大的错误。本节所讨论的错误处理是指执行 SQL 语句时所发生的错误。其内容是在执行 SQL 语句时如果所发生错误，应用程序如何才能及时发现和恢复它的执行错误或警告。

ORACLE 进行错误处理的重要依据是 SQLCA 和 ORACA。我们在第二章中已讨论过这两个通讯区。每一个 SQL 语句执行后，ORACLE 自动把其执行的结果码存入通讯区中。因此，在其执行后就可通过测试通讯区来确定该 SQL 语句的执行是否成功。在 ORACLE 数据库系统中，可采用如下三种方法测试错误：

- 用 WHENEVER 语句隐式测试 SQLCA
- 显式测试 SQLCA
- 用指示器变量测试 Null 值和“截短”现象。

关于用指示器变量测试 Null 和“截短”已在第 2 章中介绍过，本章不再叙述。

所谓显式测试 SQLCA，就是在程序中直接存取 SQLCA 变量，以确定 SQL 语句的执行是否发生错误。本节重点介绍用 WHENEVER 语句隐式测试 SQLCA 的方法。

3.6.1 使用 WHENEVER 语句进行错误处理

WHENEVER 语句是说明性语句。当执行 SQL 语句发生错误、警告或找不到等情况时，可用它来指出应采取的动作。使用该语句来诊断 SQL 错误是最简单、最常用和最有效的方法。

(1) WHENEVER 语句的格式

WHENEVER 语句的格式如下：

```
EXEC SQL WHENEVER { SQLERROR | NOT FOUND | SQLWARNING}
        {CONTINUE | DO function call | DO break | GOTO lable_name | STOP} ;
```

ORACLE 对以下三个条件自动检测 SQLCA：

- NOT FOUND

当执行 SQL 数据操纵语句时，如果没有一行被处理，SQLCODE 就被设置为 +1403 或 +100，此时“NOT FOUND”条件满足。

- SQLERROR

当 SQL 语句的执行发生错误时，SQLCODE 被设置为负值。此时“SQLERROR”条件满足。

- SQLWARNING

当把一个截短列值赋给一个输出宿主变量时，或有其它例外发生时，SQLWARN[]的相应元素被设置，此时“SQLWARNING”条件满足。

当 MODE={ANSI|ANSI14}时，SQLCA 是可选的。但是，当使用 WHENEVER SQLWARNING 时，必须说明 SQLCA。

当 ORACLE 发现上述三个条件之一时，就可采取下述五个动作中的任意一个：

· CONTINUE

程序继续运行下一个语句(如果可能的话)。这是一个缺省动作，它等价于不使用 WHENEVER 语句。通常使用它来切断条件检测。

· DO function call 或 DO break

当 SQL 语句的执行发生错误时，调用一个函数。但不允许给该函数传递参数，也不许它有返回值。

能使用 DO break 动作提早退出一个循环，或终结 switch 语句中的一个 case，它类似 C 中的 break 语句。下例说明 DO break 的用法：

```
main()
{ …
    EXEC SQL DECLARE dept_cursor CURSOR FOR
        SELECT DEPTNO, DNAME
            FROM DEPT;
    EXEC SQL OPEN dept_cursor;
    EXEC SQL WHENEVER NOT FOUND DO break;
    while(1)
    { EXEC SQL FETCH dept_cursor
            INTO :dept_number, :dept_name;
    …
    }
    EXEC SQL CLOSE dept_cursor;
    …
}
```

· GOTO lable_name

程序分支到 lable_name 所标的语句。

· STOP

停止程序运行，但不提交和 回滚事务。

(2)使用 WHENEVER 语句应注意的问题

· WHENEVER 语句的作用范围

WHENEVER 语句是一个说明性语句，它的作用范围是位置的，而不是逻辑的。它测试源文件中所有跟在其后的可执行 SQL 语句。因此，WHENEVER 语句应编写在第一个要测试的可执行 SQL 语句之前。

一个 WHENEVER 语句的有效性一直保持到另一个具有相同测试条件的 WHENEVER 语句为止。

在下例中，第一个 WHENEVER 语句被第二个替代。因此，第一个仅对 CONNECT 语句有效，第二个对 UPDATE 和 DROP 两句都有效，并不考虑从 step1 转到 step3 的控制流程：

```
step1:
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    ...
    GOTO step3;
step2:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL UPDATE EMP SET SAL=SAL * 1.10;
    ...
step3:
    EXEC SQL DROP INDEX EMP_INDEX;
    ...
```

在用光标提取行时，常用 NOT FOUND 条件来处理结束。如果 FETCH 语句没有返回数据时，程序应该分支到标号所标的程序段，以关闭光标…。例如：

```
...
main()
{ ...
    EXEC SQL WHENEVER NOT FOUND goto no_more;
    while(1)
    { EXEC SQL FETCH dept_cursor
            INTO :dept_name, :location;
    ...
    }
    ...
Dno_more:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CLOSE dept_cursor;
    EXEC SQL COMMIT RELEASE;
    ...
```

·在编错误处理的代码时，要注意避免出现无限循环

如下边的代码段，当 DELETE 语句由于没有满足查询条件的行而设置 NOT FOUND 时，就会出现无限循环。

```
...
main()
{ ...
    EXEC SQL WHENEVER NOT FOUND goto no_more;
```

```
    while(1)
    {
        EXEC SQL FETCH dept_cursor INTO :dept_name, :location;
        ...
    }
no_more:
    EXEC SQL DELETE FROM DEPT
            WHERE DEPTNO=:dept_number;
```

为了避免出现无限循环，上边的 no_more 开始的代码段可按如下格式写：

```
...
no_more:
    EXEC SQL WHENEVER NOT FOUND goto no_match;
    EXEC SQL DELETE FROM DEPT
        WHERE DEPTNO=:dept_number;
    ...
no_match:
    ...
```

· 对包含多个函数的源文件，要注意避免出现编译错误

因为，在一个源文件内，WHENEVER 语句管理所有跟在其后的 SQL 语句，如果这些 SQL 语句是分布在不同的函数内时，就有可能出现编译错误。在下例中，因标号 sqlerror 在 delete_row()函数内，而不在 INSERT 语句所处的函数 insert_row()内，因此出现编译错误：

```
delete_row()
{ ...
    EXEC SQL WHENEVER SQLERROR goto sqlerror;
    EXEC SQL DELETE
        FROM DEPT
        WHERE DEPTNO=:dept_number;
    ...
    sqlerror:
        puts("Error ocaurred!");
}
insert_row()
{ ...
    EXEC SQL INSERT INTO DEPT(DEPTNO, DNAME, LOC)
            VALUES(:dept_number, :dept_name, :location);
    ...
}
```

3.6.2 显式测试 SQLCA

显示测试SQLCA，就是可在每一个可执行SQL语句后编写一段代码，来测试SQLCA，以确定该SQL语句的执行是否发生错误，发生什么错误，并进行相应的处理。例如：

```
...
EXEC SQL INSERT……
if (sqlca.sqlcode<0)
{ /* handle error */
    printf("insert-error!");
    ...
}
```

3.6.3 错误处理举例

例 3.13 错误处理

```
/*************************************
 *  说明：
 *        该程序首先提示部门号，然后把职员的名字、
 *        工资插入表中，并诊断 SQLCA、ORACA 中的信息。
 *************************************/
EXEC SQL BEGIN DELCARE SECTION;
    char username[20];
    char password[20];
    char emp_name[15];
    int dept_number;
    float salary;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE ORACA;
...
printf("\n username?");
scanf("%s", username);
printf("\n password?");
scanf("%s", password);
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
EXEC SQL CONNECT :username INDETIFIED BY :password;
printf("\n connected to ORACLE");
/* 使 ORACA 能使用 */
EXEC ORACLE OPTION (ORACA=YES);
/* 在 ORACA 中设置标志 */
```

```
oraca.oradbgf=1;      /* 能进行 DEBUG 操作 */
oraca.oracchf=1;      /* 进行光标高速缓冲区统计 */
oraca.orastxtf=3;     /* 总保存 SQL 语句 */
printf("\n Department number?");
scanf("%d", &dept_number);
EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT ENAME, SAL * NVL(COMM, 0)
            FROM EMP
            WHERE DEPTNO= :dept_number;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
while(1)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    if (salary < 2500)
        EXEC SQL INSERT PAY1 VALUES(:emp_name, :salary);
    else
        EXEC SQL INSERT INTO PAY2 VALUES(:emp_name, :salary);
}
no_more:
    EXEC SQL CLOSE emp_cursor;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;
    printf("\nLast SQL statement:%s",oraca.orastxt.orastxtc);
    printf("\n··· at or near line number:%d",oraca.oraslnr);
    printf("\n");
    printf(\n"     cursor cache statistics");
    printf("\n ---------------------------- ");
    printf("\nMaximum value of MAXOPEN CURSORS%d",oraca.orahoc);
    printf("\nMaxmum open cursors requied:%d",oraca.oramoc);
    printf("\nCurrent number of open cursor:%d",oraca.oracoc);
    printf("\nNumber of cache reassignments:%d",oraca.oranor);
    printf("\nNumber of SQL statement parses:%d",oraca.oranpr);
    printf("\nNumber of SQL statement executions:%d",oraca.oranex);
    exit(0);

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
```

```
printf("\nLast SQL statement: %s",oraca.orastxt.orastxtc);
printf("\n··· at or near line number: %d",oraca.oraslnr);
printf("\n");
printf("\n Cursor cache statistics");
printf("\n ..................................");
printf("\nMaxmum value of MAXOPENCURSORS: %d",oraca.orahoc);
printf("\nMaxmum open cursors required: %d",oraca.oramoc);
printf("\nCurrent number of open cursors: %d",oraca.oracoc);
printf("Number of cache reassingnments: %d", oraca.oranor);
printf("\nNumber of SQL statement parses: %d",oraca.oranpr);
printf("\nNumber of SQL statement executions: %d",oraca.oranex);
exir(1);
.....
```

第四章　事务处理

§4.1　保护数据库中数据完整性和安全性的措施

ORACLE 管理的作业或任务叫会话期，运行一个应用程序或工具（如 SQL * Forms），就叫调用一个用户会话期。ORACLE 允许同时处理几个用户会话期和共享计算机资源。为此，它必须控制“并发性”和资源共享问题。否则就有可能破坏数据完整性，即可能按错误的规则改变数据或结构。因此，保护数据库中数据的完整性和安全是十分重要的问题。ORACLE 数据库管理系统提供了如下一些保护措施：

(1)ORACLE 使用锁来控制数据的并发存取。锁给用户一个数据库资源（如表或数据行）的临时所有权。当一个用户对数据库中的对象（如表或行）加排它锁时，在他使用锁来完成对数据的修改之前，别的用户就不能更改它。用户一般只需使用缺省封锁机制来保护 ORACLE 的数据和结构，但也可显示的使用行封锁或表封锁机制。

当两个或多个用户试图存取同一个数据库对象时，有可能出现死锁。例如，两个用户同时更新同一个表时，如果每一个人都试图更新由对方封锁的行的话，就会发生等待。因为每一个用户都在等待被另一方占用的资源，于是在 ORACLE 解除该死锁之前，二者都不能继续下去。为了解除死锁，ORACLE 向完成工作量最少的事务发错误信号，并在 SQLCA.SQLCODE 中返回错误码“deadlock detected while waiting for resource”。系统根据此信号撤销其封锁。

(2)当一个表在被一个用户查询同时又被另一个用户更新时，ORACLE 为该查询产生一个表数据的读一致视图。这样，在整个查询过程中，由该查询所要读的数据不会改变。在整个更新过程中，OARCLE 取表数据的快照，并在回滚段中记录变更。

(3)利用事务保护数据库

ORACLE 是面向事务的数据库，它用事务来保证数据库的一致性。

为了保证数据的完整性和一致性，在处理一个事务时，对数据的操作（如修改、删除和插入等）是在工作区中进行，而不是直接对数据库的表。即对数据库所做的变更先暂存放在工作区中，尚未写入数据库表中。当事务处理成功结束时，才把这些变更提交给数据库，即写入数据库表中。当事务中途失败时，可通过回滚整个事务、部分事务或行，来撤消对该事务所作的全部变更或部分变更。

§4.2　事务的定义和提交

(1)什么叫事务

一个事务是由一系列逻辑相关的 SQL 语句所组成，这些语句被用来完成某一特定任务。事务是 ORACLE 进行处理的单位，它所产生的变更或者同时被提交（永久化），或者被回滚（取消）。如果一个应用程序在一个事务中间失败，则数据库自动被恢复到该事务执行之前的状态。

(2)事务的组成

一个事务是由一系列逻辑相关的 SQL 语句所组成。

·事务的开始：

程序中的第一个可执行 SQL 语句(非 CONNECT 语句)开始一个事务。当一个事务结束时，下一个可执行 SQL 语句就自动开始另一个事务。

·事务的结束：

一个带或不带 RELEASE 参数的 COMMIT 或 ROLLBACK 语句结束一个事务。

一个数据定义语句(如 ALTER、CREATE 或 GRANT)在执行之前和执行之后都自动发一个 COMMIT 语句，隐式地永久化数据库的变更和结束事务。

当系统发生故障，或用户会话期因软件问题、硬件问题或被迫中断而意外停止时，也结束事务，并且 ORACLE 回滚事务。

(3)事务提交

在事务处理过程中，对数据库产生的变更是暂存放在工作区中，并未写入数据库表中。这时，其他用户还不能存取这些变更的数据，他们能看到的还是该事务开始之前的数据。因此，在事务结束时，应当把这些变更写入数据库中，这种写入工作叫提交。只有提交之后，其他用户才能访问这些数据。

事务提交是由 COMMIT 语句完成，该语句的功能描述如下：

·使当前事务对数据库所作的全部变更永久化。

·使这些变更对其他用户可见，

·取消所有的保留点(见下节)，

·释放所有的行和表封锁，但不分析锁，

·关闭 CURRENT OF 子句中引用的光标，或者当 MODE={ANSI|ANSI14}时，关闭所有显式光标。

·结束事务

COMMIT 语句常用的书写格式如下：

```
EXEC SQL COMMIT WORK RELEASE;
EXEC SQL AT :db_link_name COMMIT WORK;
```

其中：

AT 子句：指出所操作的数据库名，省略时是缺省数据库。

WORK：是为支持与标准 SQL 一致性而设置的，使用它可增强程序可读性。语句 COMMIT 和 COMMIT WORK 是等价的。

RELEASE：释放所有资源并切断与 ORACLE 的连接。

使用 COMMIT 语句的注意事项：

· COMMIT 语句不影响宿主变量的值和程序控制流程。

· 不需要在数据定义语句之后写 COMMIT 语句，因为数据定义语句执行前后都自动发出 COMMIT。因此，无论它们成功或失败，前面的事务都被提交。

程序中必须用 COMMIT(或 ROLLBACK)命令和 RELEASE 可选项来显式提交最后一个事务。

§4.3 事务回滚

4.3.1 事务保留点

事务保留点是事务内的标志点。当一个事务比较长时，可用保留点将它划分为几部分，以便对该复杂的事务给予更多的控制。例如，当一个事务包含多个功能时，可在每个功能的开始点标志一个保留点。于是，当一个功能中途失败时，就能很容易地把 ORACLE 数据恢复到该功能(而不是该事务)执行之前的状态。

事务保留点是由 SAVPOINT 语句建立。该语句的常用其格式有：

```
EXEC SQL AT :db_link_name SAVEPOINT savepoint_name;

EXEC SQL SAVEPOINT savepoint_name;
```

其中 savepoint_name 是保留点名。

例如，下面的事务包含插入、更新和删除，每个功能的开始都设立了一个保留点。当执行 DELETE(或 UPDATE，INSERT)发生错误时，就回滚到保留点 savepoint_delete(或 savepoint_update，savepoint_insert)，取消相应的变更。这种回滚称为部分回滚。

为了实现部分回滚，可用带 TO SAVEPOINT 子句的 ROLLBACK 语句把事务回滚到指定的保留点。

```
...
/* 插入一些行 */
EXEC SQL WHENEVER SQLERROR STOP;
n=15;
EXEC SQL SAVEPOINT savepoint_insert;
for (i=1; i<=n;i++)
{
    printf("\n employee number? =");
    scanf("%d", &emp_number);
    printf("\n employee name?");
    scanf("%s", emp_name.arr);
    emp_name.len=strlen(emp_name.arr);
    printf("\n employee salary? =");
    scanf("%f", &salary);
    EXEC SQL WHENEVER SQLERROR GOTO insert_error;
    EXEC SQL INSERT EMP (EMPNO, ENAME, SAL)
        VALUES (:emp_number, :emp_name, :salary);
}
/* 更新 */
n=20;
EXEC SQL WHENEVER SQLERROR STOP;
```

```
        EXEC SQL SAVEPOINT savepoint_update;
        for (i=1; i<=n;i++)
        {
            printf("\n employee number?");
            scanf ("%d", &emp_number);
            printf("\n new salary?");
            scanf("%f",&salary);
            EXEC SQL WHENEVER SQLERROR GOTO update_error;
            EXEC SQL UPDATE EMP SET SAL=:salary
                WHERE EMPNO= :emp_number;
        }

        /* 删除一些行 */
        EXEC SQL WHENEVER SQLERROR STOP;
        EXEC SQL SAVEPOINT savepoint_delete;
        printf("\n Department number?");
        scanf ("%d", &dept_number);
        EXEC SQL WHENEVER SQLERROR GOTO delete_error;
        EXEC SQL DELETE FROM EMP
            WHERE DEPTNO= :dept_number;
          ......

delete_error:
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXE SQL ROLLBACK TO SAVEPOINT savepoint_delete;
        ...
insert_error:
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXE SQL ROLLBACK TO SAVEPOINT savepoint_insert;
        ...
update_error:
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXE SQL ROLLBACK TO SAVEPOINT savepoint_update;
        ...
```

如果给两个保留点命名相同的名字时，则前面那个保留点将被取消。每个用户会话期的活动保留点个数最大缺省值是5。一个活动的保留点是指最后一次提交或回滚以来标志的保留点。数据库管理员能把这个数提高到255，这可通过在INIT.ORA文件中增加参数SAVEPOINTS的值来实现。

4.3.2 事务回滚

事务回滚是保证数据一致性的重要措施。ORACLE 使用 ROLLBACK 语句实现事务的回滚，使数据库恢复到变更之前的状态。例如，若在表删除过程中出现错误时，可用 ROLLBACK 语句恢复到没删除之前状态。ORACLE 提供了三个级别的回滚：

- 事务级回滚
- 部分事务回滚，即回滚到指定的保留点
- 语句级回滚

(1)事务级回滚

用不带 TO SAVEPOINT 子句的 ROLLBACK 语句可把当前事务回滚到其开始点，即使整个事务恢复到其处理之前的状态。这种回滚称之事务级回滚。回滚就保证不会使数据库停留在不一致状态。不带 TO SAVEPOINT 子句的 ROLLBACK 语句的常用格式有：

```
EXEC SQL ROLLBACK WORK RELEASE;
EXEC SQL AT :db_link_name ROLLBACK WORK RELEASE;
```

不带 TO SAVEPOINT 子句的 ROLLBACK 语句有如下功能：

- 取消当前事务对数据库所作的全部变更，
- 取消当前事务中的所有保留点，
- 结束该事务，
- 释放所有的表和行封锁，但不分析锁。
- 关闭 CURRENT OF 子句中所引用的光标，或者当 MODE={ANSI|ANSI14}时，关闭全部的显式光标。

(2)部分事务回滚

所谓部分事务回滚就是把当前事务回滚到其某一保留点(而不是开始点)。可用带 TO SAVEPOINT 子句的 ROLLBACK 语句把当前事务回滚到其中某一保留点。这样可避免取消当前事务的所有变更。带 TO SAVEPOINT 子句的 ROLLBACK 语句的常用格式有：

```
EXEC SQL ROLLBACK WORK TO SAVEPOINT savepoint_name ;
EXEC SQL AT :db_link_name ROLLBACK WORK
    TO SAVEPOINT savepoint_name ;
```

带 TO SAVEPOINT 子句的 ROLLBACK 语句的功能如下：

- 取消指定保留点被标志以来对数据库所作的全部变更。
- 取消指定保留点之后所标志的全部保留点，但回滚到的那个保留点不被取消。
- 取消指定保留点被标以来所获得的全部行或表封锁。

在使用 ROLLBACK 语句时，应注意以下几点：

- 在程序中，一定要用 RELEASE 可选项显式地提交或回滚最后一个事务，以切断与 ORACLE 的连接。
- ROLLBACK 语句是错误处理的一部分，因此，它应被放置在错误处理函数中。
- 不应该在 ROLLBACK TO SAVEPOINT 语句中指定 RELEASE 可选项。因为该可选项要释放程序中保持的所有资源并切断与 ORACLE 联系。

· 在执行 SQL 语句过程中遇到错误时，需要进行回滚；而在分析过程中遇到错误时，不需要进行回滚。

· ROLLBACK 语句不影响宿主变量的值或程序的控制流。

· 当 MODE={ANSI13|ORACLE}时，未在 CURRENT OF 子句中引用的光标在通过 ROLLBACK 时仍保持打开。

· 如果程序非正常结束，ORACLE 自动回滚事务。

(3)语句级回滚

在执行 SQL 语句之前，ORACLE 标志一个隐含的保留点(用户不能用)。如果 SQL 语句失败，ORACLE 就自动把它回滚，并在 sqlca. sqlcode 中返回一个错误码。例如，如果一个 INSERT 语句试图在一个唯一的索引中插入一个重复值时，就会产生一个错误，并回滚该语句。

ORACLE 也能回滚单个 SQL 语句以断开死锁。ORACLE 把一个错误信号发送给其中一个参加事务，并回滚那个事务中的当前 SQL 语句。

仅由失败的 SQL 语句开始的操作被回滚，在该 SQL 语句之 前所做的操作被保留。因此，如果一个数据定义语句失败的话，那么在它之前的变更被自动保留。

§4.4 只读事务

(1) 什么叫只读事务

满足以下条件的事务是只读事务：

· 以"SET TRANSACTION READ ONLY"语句开始，

· 以 COMMIT 或 ROLLBACK、数据定义语句结束，

· 其中只允许 SELECT 语句。不允许包括 INSERT、DELETE 或 SELECT FOR UPDATE OF 等语句，否则将引起错误。

在只读事务中，所有的查询都引用数据库的同一个快照，该快照提供多表、多查询、读一致视图，其他用户能 照例继续查询或更新数据。

SET TRANSACTION 语句的格式如下：

EXEC SQL SET TRANSACTION READ ONLY

其中 READ ONLY 参数是必须的。

(2) 只读事务的应用

当对一个或多个表进行多查询，而其他用户又要更新这些表时，只读事务是特别有用的。下面是一个只读事务的例子：

库存管理员通过使用一个只读事务来生成一天、上周和上月的销售活动汇总报表。该报表不受其他用户在该事务运行期间更新数据库的影响。

```
EXEC SQL SET TRANSACTION READ ONLY;
EXEC SQL SELECT SUM(SALEAMT) INTO :daily
    FROM SALES
    WHERE SALEDATE=SYSDATE;
EXEC SQL SELECT SUM(SALEAMT) INTO :weekly
    FROM SALES
```

在设计一个应用时，应该把逻辑上相关的活动划分在一个事务中，一个精心设计的事务应该正好包括完成给定任务所必须的步骤(不多也不少)。

对于所引用的表来说，其数据必须都处于一致状态，因此，事务中的所有 SQL 语句应该按一致的方法改变数据。例如，资金在两个银行帐目中的移动应包括一个借方和一个贷方帐目的更新。这两个帐目的更新应该同时成功或同时失败，一个无关的更新(如对另一个帐目的存款)不应包括在该事务中。

2. 封锁特权

如果用户要在应用程序中使用封锁语句，他就必须有使用该锁的特权。DBA 能封锁任何一个表。其他用户只能封锁他们自己所拥有的、或有 ALTER、SELECT、INSERT、UPDATE、或 DELETE 特权的表。

3. PL/SQL 块中的 COMMIT 和 ROLLBACK 语句

如果一个 PL/SQL 块是一个事务的一部分，则在块内的 COMMIT 和 ROLLBACK 将影响整个事务。在下例中，ROLLBACK 语句撤消由 UPDATE 和 INSERT 所产生的变更：

```
...
EXEC SQL INSERT INTO EMP...
EXEC SQL EXECUTE
    BEGIN
        UPDATE EMP
        ...
     EXECETPION
            WHEN DUP_VAL_ON_INDEX THEN
             ROLLBACK;
        ...
    END;
END_EXEC;
...
```

```
        printf("\nseat_number=%d,price=%f",seat_number,price);
        printf("\nEnter seat selection:Y or N");
        scanf("%c",selection);
        if (selection == 'Y')
        {
            EXEC SQL UPDATE EMP SET STATUS= 'Y'
                WHERE CURRENT OF seat_cursor;
            printf("\nseat-number selected: %d",seat_number);
        }
        printf("\n do you continue(y/n)?");
        c=getc( );
        if (c== 'n") break;
}
no_more:
        EXEC SQL CLOSE seat_cursor;
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL COMMIT WORK RELEASE;
        exit(0);
            ...
```

在上面的例子中,所有可能被更新的行在 OPEN 时被封锁,因此别的用户不能更新它们。当事务提交或回滚(除非回滚到保留点)时,行封锁被释放。

4.5.2 表封锁

可使用 LOCK TABLE 语句按指定的封锁方式显示地封锁一个或多个表。例如,下面的语句以行共享方式封锁 EMP 表:

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT;
```

行共享允许对一个表并行访问,并阻止其他用户为了排它而封锁整个表。

封锁方式决定其它什么样的封锁还能被放置在该表上。例如,许多用户能同时获取一个共享封锁;但一次只能有一个用户获取一个排它锁,其他用户不能对该表进行插入、更新或删除等。

对某些封锁方式,能同时在一个表上加几个;而对另一些封锁方式,则只允许一个表一个。例如,可以有多个用户同时都对一个表加 SHARE 封锁。但是一次只能有一个用户对一个表加 EXCLUSIVE 封锁。

当一个用户封锁一个表时,也就同时决定了其他用户怎样能访问它。一个封锁的表在 COMMIT 或 ROLLBACK 该事务之前仍保持封锁。

一个封锁决不会阻止其他用户查询该表,而一个查询也决不会对一个表加封锁。

§4.6 事务定义和控制中应注意的问题

1. 事务设计

在设计一个应用时，应该把逻辑上相关的活动划分在一个事务中，一个精心设计的事务应该正好包括完成给定任务所必须的步骤(不多也不少)。

对于所引用的表来说，其数据必须都处于一致状态，因此，事务中的所有 SQL 语句应该按一致的方法改变数据。例如，资金在两个银行帐目中的移动应包括一个借方和一个贷方帐目的更新。这两个帐目的更新应该同时成功或同时失败，一个无关的更新(如对另一个帐目的存款)不应包括在该事务中。

2. 封锁特权

如果用户要在应用程序中使用封锁语句，他就必须有使用该锁的特权。DBA 能封锁任何一个表。其他用户只能封锁他们自己所拥有的、或有 ALTER、SELECT、INSERT、UPDATE、或 DELETE 特权的表。

3. PL/SQL 块中的 COMMIT 和 ROLLBACK 语句

如果一个 PL/SQL 块是一个事务的一部分，则在块内的 COMMIT 和 ROLLBACK 将影响整个事务。在下例中，ROLLBACK 语句撤消由 UPDATE 和 INSERT 所产生的变更：

```
...
EXEC SQL INSERT INTO EMP...
EXEC SQL EXECUTE
    BEGIN
        UPDATE EMP
        ...
     EXECETPION
            WHEN DUP_VAL_ON_INDEX THEN
             ROLLBACK;
        ...
    END;
END_EXEC;
...
```

第五章　动态 SQL 技术

§5.1　动态 SQL 技术的基本思想和方法

5.1.1　什么是动态 SQL

动态 SQL 技术是一种先进的程序设计技术，它是相对于静态而言。在前面各章中所讲的嵌入 SQL 语句都是静态 SQL 语句。因为这些语句在编码时就能把它们完整的地写出来。但是，在有些情况下，在编码时 SQL 语句还不能完整写出来，而是在程序执行时才能构造出来。这种在程序执行过程中临时生成的 SQL 语句叫动态 SQL 语句。利用动态 SQL 语句来编写 PRO＊C 程序的方法叫动态 SQL 技术。

例如，一个通用的报表生成器，在编码时必须为生成各种报表提供各样的 SELECT 语句。在这种情况下，SELECT 语句的构成在运行之前是不知道的，而且这类语句每次执行都还可能有变化。如访问的表名、列名及查询条件等。象这类 SQL 语句叫动态 SQL 语句。

所以，不像静态 SQL 语句，动态 SQL 语句不是预先被嵌入在源程序中，而是在运行时通过程序临时输入或构造出来。

5.1.2　动态 SQL 的优缺点

使用动态 SQL 语句有如下一些优点：

·包含动态 SQL 语句的程序比一般的嵌入式 SQL 程序更加通用、功能更强。例如，一个通用的报表生成器能产生各种报表。

·包含动态 SQL 语句的程序比一般的嵌入式 SQL 程序的编码更加灵活。用户可在运行时通过交互地输入来构造 SQL 语句；例如，应用程序可以简单地提示用户在 SELECT、UPDATE 或 DELETE 语句的 WHERE 子句中所用的查询条件；或者向用户列出 SQL 操作、表和视图名、列名等的菜单，让他从中选择。

·包含动态 SQL 语句的程序比一般的嵌入式 SQL 程序更加精炼。

但是有些包含动态 SQL 语句的程序比一般的嵌入式 SQL 程序的编码更加复杂、使用的数据结构更特殊和需要更多的运行处理时间。虽然可以不去关心时间问题，但其编码比较复杂，它要求用户必须懂得动态 SQL 的概念和方法。

实际上，静态 SQL 几乎能满足所有程序设计需要。但有时程序可能比较繁索。如果要求应用非常灵活时，就使用动态 SQL。一般来说，如果下列情况之一不知道时，建议用动态 SQL 语句：

·SQL 语句的文本(命令、子句等等)

·宿主变量个数

·宿主变量的数据类型

·引用的数据库对象，如列、索引、序列、表、用户名和视图。

5.1.3 动态 SQL 语句的表示方法

一般来说，应该使用一个字符串变量(或包含一个有效的 SQL 语句文本的字符串)来表示一个动态 SQL 语句的文本，但该文本不包含 EXEC SQL 子句和语句终结符(;)。下列语句不能作动态 SQL 语句：

- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- INCLUDE
- OPEN
- PREPARE
- WHENEVER

在大多数情况下，动态 SQL 语句的文本中可能包含虚拟宿主变量，这些变量只为实宿主变量保留位置。因为虚拟宿主变量只是占有位置的变量，所以不需要说明它们，其名字可任意命名。例如，下列两个串之间没有差别：

'DELETE FROM EMP WHERE DEPTNO=:dept_number AND JOB=:job_title'

'DELETE FROM EMP WHERE DEPTNO=:v1 AND JOB=:v2'

5.1.4 动态 SQL 语句的处理过程

一个动态 SQL 语句的处理过程如下：

(1) 构造动态 SQL 语句：提示用户输入和构造一个动态 SQL 语句文本，

(2) 分析该语句：ORACLE 检查该语句的语法和语义以及引用的数据库对象是否有效，另外还检查数据库的存取权限，分配所需要的资源，并找出最佳存取路径。

(3) 虚实结合：提示用户输入实宿主变量的值，ORACLE 把实宿主变量与 SQL 语句相结合，获得实宿主变量的地址，以便存取它。

(4) ORACLE 执行该动态 SQL 语句，完成所要求做的事情(如删除行)。

(5) 动态 SQL 语句能用实宿主变量的新值重复执行。

5.1.5 动态 SQL 方法的选择

共有四种动态 SQL 方法：

1. 方法 1：

用方法 1 处理的 SQL 语句一定是非 SELECT 语句，且不包含任何虚拟输入宿主变量。例如下面的串是合法的：

'DELETE FROM EMP WHERE DEPTNO=17'

'GRANT SELECT ON EMP TO SCOTT"

2. 方法 2

用方法 2 处理的 SQL 语句也一定不是 SELECT 语句，它可能包含虚拟输入宿主变量，且该变量的类型和个数是已知的。如下例：

'INSERT INTO EMP (ENAME, JOB) VALUES (:v1, :v2)'

'DELETE FROM EMP WHERE EMPNO=:v'

3. 方法 3

用方法 3 处理的 SQL 语句一定是 SELECT 语句，而且它包含的选择表项(指紧跟在关键字 SELECT 之后的列名或表达式)或虚拟输入宿主变量的个数或类型在预编译时都已知。例如：

'SELECT MIN(SAL),MAX(SAL) FROM EMP WHERE DEPTNO=:dept_number'

'SELECT EMPNO,ENAME FROM EMP WHERE JOB=:job'

4. 方法 4

用方法 4 处理的 SQL 语句可以是 也可不是 SELECT 语句，但它包含的选择表项或虚拟输入宿主变量的个数和类型直到程序运行时还不知道。例如：

'INSERT INTO EMP (<未知>)VALUES(<未知>)'

'SELECT(<未知>) FROM EMP WHERE DEPTNO=20'

在这四种方法中，对每一个后面的方法所施加的限制都比前面的方法少，因而后者比前者更通用和更灵活。但是，后者的编码要比前者更困难。

对于一个实际应用如何选择正确的动态 SQL 方法呢？读者可根据图 5-1 作出决策。

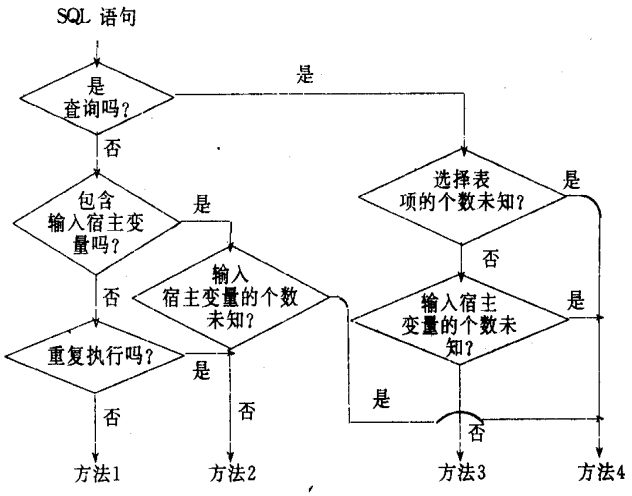


图5-1 选择正确的SQL方法

§5.2 动态 SQL 方法 1

方法 1 是最简单的一种动态 SQL 方法。它仅适用于非 SELECT 语句，且语句中不包含输入宿主变量。下面的每一个串都可作为一个合法的动态 SQL 语句：

'DELETE FROM EMP WHERE DNAME="COMPUTER"'

'CREATE TABLE table_name…'

'DROP INDEX index_name'

'UPDATE EMP SET EMPNO =591500'

'GRANT SELECT ON DEPT TO Mis. Mary'

'REVOKE RESOURCE FROM username'

方法 1 中的动态 SQL 语句是用 EXECUTE IMMEDIATE 语句来处理的。例如,语句

EXEC SQL EXECCUTE IMMEDIATE :host_string;分析和执行宿主串:host_string 中的 SQL 语句。而语句

EXEC SQL EXECCUTE IMMEDIATE 'UPDATE EMP SET EMPNO =591500';则直接分析和执行 'UPDATE SET ……' 语句。

EXECUTE IMMEDIATE 语句的功能是分析动态语句的文本,检查是否有错误。如果有错误则不执行它,并在 SQLCA. SQLCODE 中返回相应的错误码,如果未发现错误,则执行它。因此对于仅执行一次的动态语句,用方法 1 最合适。而对于重复多次执行的动态 SQL 语句,用方法 1 则会降低执行效率,应选用方法 2。

方法 1 的处理过程是:先构造一个动态 SQL 语句,然后用 EXECUTE IMMEDIATE 语句来处理它。

下面是方法 1 的应用例子。

例 5.1 动态 SQL 方法 1

```
/*************************************
*     说明:
*         该例使用动态 SQL 方法 1 来建立一个表、插入一行、提交插入、
*         然后删除该表。
*************************************/
# define USERNAME "SCOTT"
# define PASSWORD "TIGER"
# include <stdio.h>

/* 说明一个 SQL 通讯区和 ORACLE 通讯区 */
EXEC SQL INCLUDE sqlca;

EXEC SQL INCLUDE oraca;

/* 启动 ORACLE 通讯区 :ORACA=YES。*/
EXEC ORACLE OPTION (ORACA=YES);

/* 指定 RELEASE_CURSOR=YES 可选项,使 PRO*C 在执行后,释放与嵌入 SQL
语句有关的资源。这将保证在数据操纵之后,ORACLE 不保持表分析锁。
因此可使表的后续数据定义操作不会导致分析锁错误。*/
```

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
/* 说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
    char *username=USERNAME;
    char *password=PASSWORD;
    char *sqlstmt1;
    char sqlstmt2[10];
    VARCHAR sqlstmt3[80];
EXEC SQL END DECLARE SECTION;
main ( )
{
    /* 错误处理说明 */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    /* 如果发生错误,将当前的 SQL 语句文本保存到 ORACA 内。*/
    oraca.orastxtf=ORASTFERR;
    /* 登录到 ORACLE. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts ("\nConnected to ORACLE. \n");

    /* 执行一个文字串形式的动态 SQL 语句 CREATE 来建立一个表。这种用法实
    际上不是动态 SQL 方法,因为该动态 SQL 语句在预编译时就完全确定。*/
     puts ("CREATE TABLE STUDENT (SNO int,ENAME CHAR(15))");

    EXEC SQL EXECUTE IMMEDIATE "CREATE TABLE
          STUDENT(SNO int, ENAME CHAR(15))";

    /* 执行一个动态 SQL 语句 INSERT 来插入一个记录。
    一个字符指针指向该动态 SQL 语句,该语句串必须以 Null 结束。这种用法是动态
    SQL 方法,因为该动态 SQL 语句在执行时才能完全确定。*/
    sqlstmt1="INSERT INTO STUDENT VALUES (951785,'李萍')";
    puts(sqlstmt1);
    EXEC SQL EXECUTE IMMEDIATE :splstmt1;

    /* 执行一个动态 SQL 语句 COMMIT 来提交插入的记录。
    这里使用的是字符数组,这种用法也是动态 SQL 方法,因为该动态 SQL 语句在
    执行时才能完全确定。*/
    strncpy(splstmt2, "COMMIT   ",10);
    printf("%.10s\n", sqlstmt2);
    EXEC SQL EXECUTE IMMEDIATE :splstmt2;
```

```
    /* 执行一个动态 SQL 语句 DROP 来删除上面建立的表。
       这里使用的是 VARCHAR 变长字符串变量，这种用法也是动态 SQL 方法，因为
       该动态 SQL 语句在执行时才能完全确定。 */
  sqlstmt3.len=sprintf(sqlstmt3.arr, "DROP TABLE STUDENT");
    puts (sqlstmt3.arr);
    EXEC SQL EXECUTE IMMEDIATE :sqlstmt3;
    /*提交变更,切断与 ORACLE 连接 */
    EXEC SQL COMMIT RELEASE;
    puts("\nHave a good day! \n");
    exit(0);

  /* 错误处理：打印错误信息,发生错误的 SQL 语句和错误位置 */
  splerror:
    printf("\n%. *s\n",sqlca.sqlerrm.sqlerrml,
            sqlca.sqlerrm.sqlerrmc);
    printf("\n%. *s \n", oraca.orastxt.orastxtc);
    printf("on line %d \n\n",oraca.oraslnr);
    /* 为了避免死循环,特给出如下的错误处理说明 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    /* 回滚悬挂的变更,并退出 ORACLE. */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

§5.3 动态 SQL 方法 2

方法 2 与方法 1 类似，也是只适于非 SELECT 语句。动态方法 2 中的 SQL 语句可包含虚拟输入宿主变量和指示器变量，但它们的个数和数据类型在预编译时必须是已知的。在该方法中，动态 SQL 语句的处理分三步：

- 构造一个动态 SQL 语句，
- 用 PREPARE 语句来分析和命名该动态 SQL 语句，
- 用 EXECUTE 语句来执行它。

5.3.1 PREPARE 语句

该语句用于分析和命名一个动态 SQL 语句。其书写格式如下：

```
EXEC SQL PREPARE [statement_name][ block_name]
        FROM [:host_string][ string_literal];
```

其中：

statement_name：标识被分析的动态 SQL 语句。它是供预编译程序使用的标识符，而不

是宿主变量。

block_name：标识被分析的动态 PL/SQL 块，也是标识符。

:host_string：是一个宿主变量，它包含动态 SQL 语句或 PL/SQL 块的文本。

string_literal：是一个文字串，表示动态 SQL 语句或 PL/SQL 块的文本。

例如：

strcpy(sql_stmt, "DELETE FROM EMP WHERE JOB=:V");

EXEC SQL PREPARE stmt FROM :sql_stmt;

在该例中，stmt 标识被分析的 SQL 语句，V 表示一个虚拟输入宿主变量。

5.3.2 EXECUTE 语句

EXECUTE 语句执行一个预先分析过的动态 SQL 语句或 PL/SQL 块。方法 2 中使用的 EXECUTE 语句的书写格式如下：

```
EXEC SQL [ FOR :host_integer]
    EXECUTE [statement_name][block_name]
                [USING :host_variable[ indicator][,…] ];
```

indicator 的语法描述如下：

```
[INDICATOR] :host_variable
```

其中：

FOR 子句指出要处理的数组元素的最大个数。

statement_name 和 block_name 的说明同 5.3.1。

UNING 子句：指出一个实宿主变量表，用于替换虚拟宿主变量。

例如

EXEC SQL EXECUTE stmt USING :job;

其中 :job 是一个实输入宿主变量，它替换前面所述的动态 SQL 语句 stmt 中的 V。

使用方法 2 应注意如下两点：

• 方法 2 与方法 1 的一个重要区别是：方法 1 是每执行一次(动态 SQL 语句)就要分析一次；而方法 2 是分析一次，可用不同的实宿主变量值执行多次。因此，在方法 1 中，如果动态 SQL 语句要重复多次执行时，应选择方法 2，这样可避免对语句的重复分析。

• UNING 子句中的实宿主变量要与被分析的动态 SQL 语句中的虚拟宿主变在类型、次序上相对应，个数相匹配。如果 USING 子句中的实宿主变量其中有一个是数组时，则所有的必须都是数组。

5.3.3 动态 SQL 方法 2 的应用举例

例 5.2 动态 SQL 方法 2

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*     说明：
```

```
*          该例使用方法 2,循环向 EMP 表插入若干行。
*          当职员号为 0 时,插入结束。
*          然后删除若干行。
**********************************/
/* 定义符号常数 */
# define USERNAME "SCOTT"
# define PASSWORD "TIGER"
# include <stdio.h>

/* 说明 SQLCA 和 ORACA */
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE oraca;

/* 启动 ORACLE 通讯区 :ORACA=YES,使它能被使用。*/
EXEC ORACLE OPTION (ORACA=YES);
/*说明 SQL 变量 */
EXEC SQL BEGIN DECLARE SECTION;
    char *username=USERNAME;
    char *password =PASSWORD;
    VARCHAR sqlstmt[80];
    int emp_number;
    VARCHAR emp_name[15];
    VARCHAR job[50],job1[50],job2[50];
    float salary;
EXEC SQL END DECLARE SECTION;
main( )
{
    /* 错误处理说明. *
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    /* 发生错误时,保存 SQL 语句文本在 ORACA. */
    oraca.orastxtf=ORASTFERR;
    /* 登录到 ORACLE. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE. \n");
    /* 构造动态 INSERT 语句。其中包含四个虚拟变量 v1,v2,v3 和 v4. */
    sqlstmt.len=sprintf(sqlstmt.arr,
    "INSERT INTO EMP (EMPNO, ENAME,JOB,SAL) VALUES (:v1, :v2,v3,
            v4)");
    /*显示 SQL 语句 */
```

```
puts(sqlstmt.arr);
/* 用 PREPARE 语句分析当前的动态 INSERT 语句,语句名是 S。 */
EXEC SQL PREPARE S FROM :sqlstmt;
/*,循环向 EMP 表插入若干行 */
for(;;)
{
  /*提示输入职员号、职员名,工作和工资,然后执行 INSERT 语句。
  USING 子句指出实输入宿主变量,它被用来替换虚拟的输入宿主变量。*/
  printf("\nEnter employee number: ");
  scanf("%d",&emp_number);
  if (emp_number==0) break;
  printf("\nEnter employee name: ");
  scanf("%s",emp_name.arr);
  emp_name.len=strlen(emp_name.arr);
  printf("\nEnter employee job: ");
  scanf("%s",job.arr);
  job.len=strlen(job.arr);
  printf("\nEnter salary: ");
  scanf("%f",&salary);
  EXEC SQL EXECUTE S USING :emp_number,:emp_name,:job,:salary;
}
/*
再构造一个新的动态 SQL 语句:DELETE。其这包含两个虚拟变量 v1 和 v2.
*/
sqlstmt.len=sprintf(sqlstmt.arr,
  "DELETE FROM EMP WHERE JOB= :v1 OR JOB= :v2");
puts (sqlstmt.arr);
/*分析该语句 */
EXEC SQL PREPARE S FROM :sqlstmt;
/* 提示输入职员的工作,然后执行该语句。
USING 子句指出实输入宿主变量,它被用来替换虚拟的输入宿主变量。*/
printf("\nEnter employee-job1: ");
scanf("%s",job1.arr);
job1.len=strlen(job1.arr);
printf("\nEnter employee-job2: ");
scanf("%s",job2.arr);
job2.len=strlen(job2.arr);
EXEC SQL EXECUTE S USING :job1, :job2;
/*提交事务,退出 ORACLE. */
```

```
    EXEC SQL COMMIT RELEASE;
    printf("\nHave a good day! \n");
    exit(0);
    /* 错误处理 ,打印错误信息, */
    sqlerror:
        /* 打印错误信息文本 */
        printf("\n%. *s\n", sqlca. sqlerrm. splerrml,
            sqlca. sqlerrm. sqlerrmc);
        /* 打印当前 SQL 语句 */
        printf("\n \"%. *s. . . \"\n",oraca. orastxt. orastxtl,
            oraca. orastxt. orastxtc);
        /* 打印发生错误的 SQL 语句的行号和文件名 */
        printf ("on line %d of %. *s. \n\n", oraca. oraslnr,
            oraca. orasfnm. orasfnml, oraca. orasfnm. orasfnmc);
        /* 回滚事务,退出 ORACLE. */
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK RELEASE;
        exit(1);
}
```

§5.4 动态 SQL 方法 3

动态方法三只适于 SELECT 语句,在该语句中所包含的选择表项个数和虚拟输入宿主变量个数在预编译时都是已知的;但是,数据库对象(如表、列)的名字可以在运行时指定;WHERE、GROUP BY 和 ORDER BY 子句也可在运行时指定。

在方法三中,动态 SQL 语句的处理过程如下:

- 构造一个动态 SQL 语句,
- 用 PREPARE 语句来分析和命名该动态 SQL 语句,
- 用光标语句来执行它。

因此,在方法三中,要用到如下六个语句 :

(1) PREPARE

其描述同方法 2。

(2) DECLARE

在方法 3 中,它的常用写法如下:

```
EXEC SQL DECLARE cursor_name
        CURSOR FOR [statement_name][block_name];
```

其中 statement_name 和 block_name 标识与光标相联系的 SQL 语句或 PL/SQL 块的名字。该名字要在前面的 DECLARE STATEMENT 语句中说明。

(3) OPEN

在方法 3 中,它的常用写法如下:

EXEC SQL OPEN cursor

[USING :host_variable [[INDICATOR] :indicator_variable] [,……]];

其中：

cursor：是要打开的光标。

USING 子句：指出执行动态 SQL 语句所用的实宿主变量名，这些宿主变量替换动态 SQL 语句中的虚拟宿主变量。

:host_variable：是实际宿主变量，用于替换动态 SQL 语句中的虚拟宿主变量。

(4) FETCH 语句

在方法 3 中，它的常用写法如下：

EXEC SQL FETCH cursor

INTO :host_variable [[INDICATOR] :indicator_variable][,……];

其中：

cursor：是光标名，FETCH 语句从该光标所标识的缓冲区中读取一行或多行。

INTO 子句：指出一个宿主变量和可选的指示器变量表。所提取的数据被赋给这些变量。

(5) CLOSE 语句

其描述同第三章。

(6) DECLARE STATEMENT 语句

该说明语句说明一个 SQL 语句或 PL/SQL 块的标识符。在方法 3 中，它的常用写法如下：

EXEC SQL DECLARE [statement_name][block_name] STATEMENT;

下面是动态方法 3 的实例程序

例 3.3 动态 SQL 方法 3

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*     说明：
*          该例使用方法 3 来检索指定部门的全体职员信息。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 定义 符号常数 */
# define USERNAME "SCOTT"
# define PASSWORD "TIGER"
# include <stdio.h>
/* 说明 SQLCA 和 ORACA */
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE oraca;
# ifdef TRUE
# undef TRUE
# endif
# define TRUE 1
/* 启动 ORACA：ORACA=YES */
```

```
EXEC ORACLE OPTION (ORACA=YES);
/*说明SQL变量 */
EXEC SQL BEGIN DECLARE SECTION;
    char *username=USERNAME;
    char *password=PASSWORD;
    VARCHAR sqlstmt[80];
    int emp_number;
    VARCHAR emp_name[10];
    float salary;
    VARCHAR job[30];
    int dept_number;
EXEC SQL END DECLARE SECTION;
/* 程序部分 */
main( )
{
    /* 错误处理说明 */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    /* 发生错误时,保存SQL语句文本在ORACA. */
    oraca.orastxtf =ORASTFERR;
    /* 登录到ORACLE. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE. \n");
    /* 构造一个动态SQL语句:SELECT。其中包含一个虚拟输入宿主变量v1 。
       并显示该SQL语句 */
    sqlstmt.len=sprintf(sqlstmt.arr,
            "SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE
            DEPTNO=:v1");
    puts (sqlstmt. arr);
    /* 输出标题 */
    printf("\nNumber  Employee Name      Job          salary \n");
    printf("..........\n");
    /* 分析动态SQL语句 */
    EXEC SQL PREPARE s FROM :sqlstmt;
    /* 说明光标c */
    EXEC SQL DECLARE C CURSOR FOR S;
    /* 输入部门号,作为实参值,打开光标 */
    printf("\n Enter Department Number :");
    scanf("%d",&dept_number);
    EXEC SQL OPEN C USING :dept_number;
```

```
    /* 如果提取完 则分支到标号 notfound */
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    /* 循环提取,直到 NOT FOUND 成立结束 */
    while (TRUE)
    {
        EXEC SQL FETCH C INTO :emp_number,:emp_name,:job,salary;
        /* Null 结束字符串 */
        emp_name.arr(ename.len )= '\0';
        job.arr(job.len )= '\0';
        printf("\n%d\t%s\t%s\t%f",
                :emp_number,:emp_name,:job,:salary );
    }
notfound:
    /* 提取结束时,打印检索的行数 */
    printf("\nQuery returned %d row%s.\n\n", sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2]==1)? "":"s");
    /*关闭光标 */
    EXEC SQL CLOSE C;
    /* 结束事务,退出 ORACLE. */
    EXEC SQL COMMIT RELEASE;
    printf("Have a good day! \n");
    exit(0);
/* 错误处理 */
sqlerror:
    /* 打印有关信息 */
    printf ("\n%. *s\n",sqlca.sqlerrm.sqlerrml,
            sqlca.sqlerrm.sqlerrmc);
    printf("\n %. *s\n",oraca.orastxt.orastxtl,
            oraca.orastxt.orastxtc);
    printf("on line %d of %. *s \n\n",oraca.oraslnr,
            oraca.orasfnm.orasfnml, oraca.orasfnm.orasfnmc);
    /* 停止错误检查 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    /* 关闭光标 */
    EXEC SQL CLOSE C;
    /* 回滚事务,退出 ORACLE. */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

§5.5 动态SQL方法4

5.5.1 方法4的基本思想

1. 方法4的特点：

方法四是一个更加复杂的程序设计技术，它既适用于SELECT语句，也适用于非SELECT语句，但是，它与方法1,2,3相比有两个突出的不同：

(1) 方法4的动态SQL语句不但包含选择表项或虚拟输入宿主变量，而且它们的个数或数据类型在编译时还不知道。

(2) 在其它方法中，ORACLE和C之间的数据类型转换是自动实现的。而在方法4中，由于动态语句中的宿主变量个数和类型在编译时还不知道，因此不能实现自动转换，必须由程序来控制数据类型之间的转换。

2. 方法四的特殊要求

PRO*C预编译程序对于所有可执行的SQL语句(包括动态方法1,2,和3的SQL语句)都产生一个ORACLE调用。

但是，对于动态SQL方法4来说，其动态SQL语句中所包含的选择表项或虚拟输入宿主变量的个数、类型直到执行之前还不知道。这就不能在预编译时产生完整的ORACLE调用。也就是说在编译时没法实现对输入或输出宿主变的存储分配，以保存SQL语句中输入或输出宿 主变量的值；也没法实现数据类型的自动转换。为了执行这类动态SQL语句，就必须提供有关输入或输出宿主变量的信息。具体来说，对动态SQL方法4，ORACLE要求提供如下信息：

- 选择表项和实输入宿主变量(也叫结合变量—bind variable)的个数。
- 每一个选择表项和实输入宿主变量的长度。
- 每一个选择表项和实输入宿主变量的数据类型。
- 每一个输出宿主变量(存储选择表项的值)和实输入宿主变量的内存单元地址。

有了这些信息以后，相应的动态SQL语句才能被执行。

3. 提供信息的方法

为了保存执行动态SQL语句所需要的信息，系统特提供一个称之为SQL描述区(SQLDA)的程序数据结构。把ORACLE所需要的全部有关选择表项或虚拟输入宿主变量的信息，除了其值和名字外，都存储在SQLDA中。具体地说：

为选择表项和虚拟输入宿主变量各分配一个SQLDA，分别叫选择SQLDA和结合SQLDA。把选择表项的描述信息存储在选择SQLDA中，而把虚拟输入宿主变量的描述信息存储在结合SQLDA中。选择表项的值和名字被存储在输出缓冲区，实输入宿主变量的值和名字被存储在输入缓冲区。输出缓冲区的地址存储在选择SQLDA中，输入缓冲区的地址存储在结合SQLDA中。这样以来，ORACLE在处理动态SQL语句时，就知道把选择表项的值写在什么地方，和从什么地方取实输入宿主变量的值了。

怎样把值存储在缓冲区呢？一般来说，输出值是通过FETCH语句写入缓冲区，而输入值是通过程序填入。

4. 如何把信息填入SQLDA

SQLDA 中所包含的各种信息主要通过以下三种方法写入：

· sqlald()函数

该函数在分配描述区和缓冲区空间的同时，还把 SLI(选择表项)或 P(虚拟输入宿主变量)的名字在缓冲区中的地址和长度写入 SQLDA 中。

· 应用程序

通过程序把存放 SLI 或 BV(结合变量或实输入宿主变量)值的缓冲区地址、长度及数据类型写入 SQLDA。

· DESCRIBE 语句

DESCRIBE SELECT LIST 语句检查每一个选择表项，确定它的名字、数据类型、约束、长度、定标和精度，然后把这方面的信息存储在选择 SQLDA 和输出缓冲区中，以供用户使用。

DESCRIBE BIND VARIABLES 语句检查每一个虚拟输入宿 主变量，以确定它的名字和长度，然后把这些信息存储在输入缓冲区和结合 SQLDA 中，供用户使用。图 5-2 表示 SQLDA 中的变量哪些是由 sqlald()函数调用、DESCRIBE 命令、FETCH 命令或程序赋值来设置。例如下边的动态 SQL 语句：

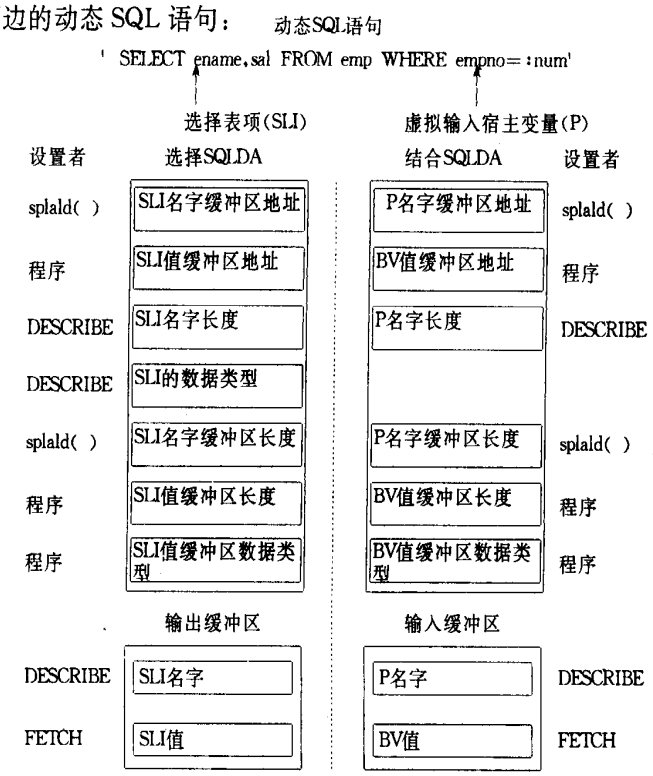


图5-2 怎样设置SQLDA中的变量

5.5.2 SQLDA 的说明和引用

1. SQLDA 的组成

为了用方法 4 处理动态 SQL 语句，程序必须显式地说明 SQLDA。SQLDA 是如表 5-1 所示的结构类型变量。

表 5-1　SQLDA 中的变量

SQLDA

```
struct sqlda
{
  long N;              /* SLI 或 P 的最大个数 */
  char **V;            /* 是指向 SLI 或 BV 值的地址数组的指针 */
  long *L;             /* 是指向 SLI 或 BV 值的长度数组的指针 */
  short *T;            /* 是指向 SLI 或 BV 值的数据类型数组的指针 */
  short **I;           /* 是指向 IV 值的地址数组指针 */
  long F;              /* SLI 或 P 的实际个数 */
  char **S;            /* 是指向 SLI 或 P 名字的地址数组的指针 */
  short *M;            /* 是指向 SLI 或 P 名字的最大长度数组的指针 */
  short *C;            /* 是指向 SLI 或 P 名字的当前长度数据组的指针 */
  char **X;            /* 是指向 IV 名字的地址数组的指针 */
  short *Y;            /* 是指向 IV 名字最大长度数组的指针 */
  short *Z;            /* 是指向 IV 名字当前长度数组的指针 */
};
struct sqlda sqlda;
```

注释：

SLI：表示选择表项
P：表示虚拟输入宿主变量
BV：表示实输入宿主变量
IV：表示指示器变量

2. SQLDA 变量

每一个 SQLDA 包含 12 个成员变量，它们的类型和用途如下：

(1) N 变量

N 指定要被描述的选择表项或虚拟输入宿主变量的最大个数。于是，N 确定了描述区中数组的元素个数。在发出 DESCRIBE 命令之前，必须用库函数 sqlald()把 N 设置为描述区中数组的维数。在 DESCRIBE 之后，必须再把 N 设置为所描述的选择表项或虚拟输入宿主变量的实际个数，该数据被存储在 F 变量中。

(2) V 变量

V 是一个数组指针，其元素分别指向缓冲区中存储的每一个选择表项或实输入宿主变量值。当对该描述区进行分配时，函数 sqlald()把数组元素 V[0]～V[N－1]设置为 0。

对于选择描述区，必须在发 FETCH 命令之前分配数据缓冲区和设置该数组。语句

EXEC SQL FETCH … USING DESCRIPTOR…把 FETCH(取到)的选择表项的值存储到由 V[0]～V[N－1]所指向的缓冲区内。例如，ORACLE 把第 i 个选择表项的值存储在 V[i]所指向的缓冲区中。

对于结合描述区，必须在发 OPEN 命令之前设置该数组。语句

EXEC SQL OPEN … USING DESCRIPTOR…指示 ORACLE 用由 V[0]～V[N－1]所指向的实输入宿主变量的值来执行动态 SQL 语句。例如，ORACLE 可用 V[i] 找到

输入数据缓冲区中的第 i 个实输入宿主变量的值。

(3) L 变量

L 是一个指针,它指向一个数组,该数组中包含存储在数据缓冲区内的选择表项或实输入宿主变量的长度。

对于选择描述区,DESCRIBE SELECT LIST 语句把长度数组设置为每一个选择表项所期望的最大数(例如 n)。FETCH 至多返回 n 个字符。

ORACLE 数据类型之间的长度格式有差别。对于 CHAR 或 VARCHAR2 选择表项,DESCRIBE SELECT LIST 把 L[i]设置为该选择表项的最大长度。对于 NUMBER 选择表项,在变量的低字节和下一个较高字节分别返回定标和精度,能使用库函数 sqlald()从 L[i]中分离出定标和精度值。

在 FETCH 之前必须把 L[i]重新设置为所需要的数据缓冲区的长度。例如,当把一个 NUMBER 强制为 C 的 char 串时,则把 L[i]设置为数的精度加符号和小数点 2 位。当把 NUMBER 强制为 C 的 float 时,则把 L[i]设置为系统的 float 长度。关于强制数据类型长度的更进一步内容,请看本章后面的数据转换一节。

对于结合描述区,在发 OPEN 命令之前,必须设置长度数组。例如,能用 strlen()来取得由用户输入的实输入宿主变量中的字符串长度,然后设置相应的数组元素。

如果想改变 ORACLE 用于第 i 个选择表项或实输入宿主变量值的长度,就需把 L[i]重新设置为所需长度。每一个输入或输出缓冲区能有不同的长度。

(4) T 变量

T 是一个指针,它指向一个数组,该数组存放选择表项或实输入宿主变量值的数据类型代码。当在数据缓冲区(由 V 数组元素进行寻址)中存储时,这些代码决定如何转换 ORACLE 数据。

对于选择描述区,DESCRIBE SELECT LIST 语句把数据类型代码数组设置为选择表项的内部数据类型(例如,CHAR、NUMBER 或 DATE)。

因为 ORACLE 数据类型的内部格式很难处理,所以在 FETCH 之前,可以设置某些数据类型。为了便于显示,通常把选择表项值的数据类型强制转换为 VARCHAR2。为了便于计算,有时需要把数据从 ORACLE 强制为 C 格式。

T[i]的高位指出第 i 个选择表项的 NULL/NOT NULL 状态。在发 OPEN 或 FETCH 命令之前,一定要清除该位。用库函数 sqlnul()来检索数据类型代码和清除 NULL/NOT NULL 位。

应当把 ORACLE 的 NUMBER 内部数据类型改变为与 V[i]所指向的数据缓冲区的 C 数据类型相一致的外部数据类型。

对于结合描述区,DESCRIBE BIND VARIABLES 把数据类型代码的数组设置为 0。在发 OPEN 之前,必须重新设置存储在每一个元素中的数据类型代码。该代码表示 V[i]所指向的数据缓冲区的外部(C)数据类型。往往实输入宿主变量的值以字符串形式存储,因此,数据类型的数组元素被设置为 1(即 VARCHAR2 数据类型代码)。

要改变第 i 个选择表项或实输入宿 主变量值的数据类型,就把 T[i]重新设置为所需要的数据类型。

(5) I 变量

I 是一个数组指针，其元素分别指向数据缓冲区中每一个指示器变量的值。必须设置地址数组中的元素 I[0]至 I[N－1]。

对于选择描述区，必须在发 FETCH 之前设置该地址数组。当 ORACLE 执行语句

EXEC SQL FETCH … USING DESCRIPTOR …

时，如果第 i 个返回的选择表项值是 NULL，那么 I[i]所指向的那个指示器变量的值被置为－1。否则，它被置为 0 或一个正整数(值被截短)。

对于结合描述区，必须在发 OPEN 命令之前，设置地址数组和相关的数据缓冲区。当 ORACLE 执行语句

EXEC SQL OPEN … USING DESCRIPTOR …

时，I[i]所指向的的那个指示器变量的值决定第 i 个实输入宿主变量是否是一个 NULL 值。如果指示器变量的值为－1，则其相关实输入宿主变量的值为 NULL。

(6) F 变量

F 是 DESCRIBE 所发现的选择表项或虚拟输入宿主变量的实际个数。因此，F 是由 DESCRIBE 来设置。如果 F＜0，则表示 DESCRIBE 发现了过多的选择表项或虚拟输入宿主变量。例如，如果先把 N 设置为 10，而后 DESCRIBE 发现了 11 个选择表项或虚拟输入宿入主变量，则 F 被设置为－11。

(7) S 变量

S 是一个数组指针，其元素分别指向数据缓冲区中所存储的每一个选择表项或虚拟输入宿主变量的名字。

使用 sqlald()来分配该数据缓冲区，并把其地址存放在 S 数组中。

DESCRIBE 指示 ORACLE 把第 i 个选择表项或虚拟输入宿主变量的名字存储在 S[i]所指向的数据缓冲区中。

(8) M 变量

M 是一个指针，它指向一个数组，该数组存放缓冲区中所包含的选择表项或虚拟输入宿主变量名字最大长度。这些缓冲区由 S 数组元素寻址。

当分配一个描述区时，sqlald()设置该数组元素 M[0]至 M[N－1]。当对 S[i]指向的数据缓冲区存储时，第 i 个名字被截成 M[i]中的长度(如果有必要的话)。

(9) C 变量

C 是一个指针，它指向一个数组，该数组元素包含选择表项或虚拟输入宿主变量名字的当前长度。

DESCRIBE 设置当前长度数组元素 C[0]至 C[N－1]。在 DESCRIBE 之后，该数组包含每个选择表项或虚拟输入宿 主变量名字中的字符个数。

(10) X 变量

X 是一个数组指针，其元素分别指向数据缓冲区中所存储的每一个指示器变量的名字。只能把指示器变量的名字与实输入宿主变量联系起来。因此，X 只适用于结合描述区。

用 sqlald( )来分配数据缓冲区并把它们的地址存储在 X 数组中。

DESCRIBE BIND VARIABLES 指示 ORACLE 把第 i 个指示器变量的名字存储在 X[i]所指向的数据缓冲区中

(11) Y 变量

Y 是一个指针，它指向一个数组，该数组的元素分别包含每一个指示器变量名字的最大长度。类似 X，Y 也只适用于结合描述区。

用 sqlald()来设置最大长度数组元素 Y[0]至 Y[N－1]。当在 X[i]指向的数据缓冲区中存储时，第 i 个名字被截成 Y[i]中的长度(如果需要时)。

(12) Z 变量

Z 是一个指针，它指向一个数组，该数组的元素分别包含当前每一个指示器变量名字的最大长度。类似 X，Z 也只适用于结合描述区。

DESCRIBE BIND VARIABLES 设置当前长度数组中的元素 Z[0]至 Z[N－1]。在 DESCRIBE 之后，该数组包含每一指示器变量名字中的字符个数。

3. SQLDA 的说明

使用方法 4 时，必须先说明一个 SQLDA。说明 SQLDA 的方式有如下三种：

(1)直接把表 5-1 中所示的代码编写到程序中。

(2)用 INCLUDE 语句

```
EXEC SQL INCLUDE sqlda;
```

(3)使用指针

若把表 5—1 中定义的结构作为一个文件(如叫 sqlda)，存于文件系统中，于是可用如下语句来说明 SQLDA：

```
EXEC SQL INCLUDE sqlda;
sqlda *bind_dp;
sqlda *select_dp;
bind_dp=sqlald(…)
select_dp=sqlald(…);
```

4. SQLDA 的引用

在方法 4 中，对于每一个活动的动态 SQL 语句都应说明一个 SQLDA。如果一个程序中有多个活动的动态 SQL 语句，则每一个语句都必须有它自己的 SQLDA。可为一个程序说明多个不同名的 SQLDA。例如，可说明如下三个选择 SQLDA：sel_desc1，sel_desc2，sel_desc3。因此，能通过三个并行打开的光标来同时进行 FETCH，但不能重复使用一个 SQLDA。

在说明 SQLDA 之后，还应该用 sqlald()函数为它分配存储空间。该函数的格式如下：

```
descriptor_name=sqlald(max_vars, max_name, max_ind_name);
```

其中：

max_vars：是 SQLDA 中要描述的选择表项或虚拟输入宿主变量的最大个数。

max_name：是选择表项或虚拟输入宿主变量名字的最大长度。

max_ind_name：是指示器变量名的最大长度。对于选择 SQLDA，该项设置为 0。

该函数除了分配 SQLDA 外，还分配相应的缓冲区，并把缓冲区的地址等填入 SQLDA 中。一般都使用指针来引用 SQLDA。

5.5.3 预备知识

为了实现动态方法 4，需要知道如下几个问题的知识：

· 数据类型转换

·强制数据类型

·处理 null/not null 数据类型

1. 数据类型转换

在既不使用数据类型等价又不使用动态 SQL 方法 4 的应用程序中，ORACLE 的内部和外部数据类型之间的转换是在预编译时解决的。

但是，在动态方法 4 中，需要通过程序来控制数据类型转换和格式化。即内部和外部数据类型之间的转换是在程序执行过程中，通过在 T 变量中设置数据类型代码来指定转换的。

当发一个 DESCRIBE SELECT LIST 命令时，ORACLE 为每一个选择表项返回一个内部数据类型代码到 T 变量中，例如，对第 i 个选择表项，数据类型在 T[i]中返回。

DESCRIBE BIND VARIABLES 命令把数据类型数组 T 设置为 0。因此，在发 OPEN 之前，必须重新设置该代码。以指出对实输入宿主变量所用的外部数据类型。

2. 强制数据类型

对于选择描述区，DESCRIBE SELECT LIST 返回 ORACLE 的内部数据类型。通常，该内部数据类型正好对应所要用的外部数据类型。但是，个别的内部数据类型映射的外部数据类型难以处理。因此，可以重新设置 T 变量的某些元素。例如，可把 NUMBER 值重新设置为 FLOAT 值(它对应于 C 中的 float 值)。ORACLE 在 FETCH 时做必要的内部和外部数据类型之间的转换。因此，在 DESCRIBE SELECT LIST 之后和 FETCH 之前一定要重新设置数据类型。

对于结合描述区，DESCRIBE BIND VARIABLES 并不返回实输入宿主变量的数据类型，而只返回它们的个数和名字。于是，必须显式地设置数据类型代码数组 T，以告诉 ORACLE 每一个实输入宿主变量的外部数据类型。ORACLE 在 OPEN 时做外部和内部数据类型之间的必要转换。

当再设置 T 数组中的数据类型代码时，就是进行“强制数据类型”转换。例如，要把第 i 个选择表项的值强制为 VARCHAR2 时，就用如下的语句：

```
select_des→T[i]=1;
```

当为了显示的目的把 NUMBER 选择表项的值强制为 VARCHAR2 时，还必须分离出该值的精度和定标字节，并用它们计算出最大的显示长度。然后，在 FETCH 之前必须重新设置 L 变量的相应元素，以告诉 ORACLE 所用的缓冲区长度。

例如，如果 DESCRIBE SELECT LIST 发现第 i 个选择表项是类型 NUMBER，而用户想以说明为 float 的 C 变量来存储该返回值，那就只需把 T[i]设置为 4，L[i]设置为系统的 float 长度。

在有些情况下，DESCRIBE SELECT LIST 返回的内部数据类型可能不适合用户的目的，例如 DATE 和 NUMBER。当 DESCRIBE 一个 DATE 选择表项时，ORACLE 就对 T 变量返回数据类型代码 12。如果不在 FETCH 之前重新设置该代码的话，则将以 7 字节的内部格式返回日期值。为了以它的缺省字符格式取得日期，可把数据类型代码从 12 改为 1(VARCHAR2)或 5(STRING)，而把长 L 的值从 7 增加到 9 或 10。

类似，当 DESCRIBE 一个 NUMBER 选择表项时，ORACLE 对 T 数组返回数据类型代码 2。除非在 FETCH 之前重新设置该代码，否则将返回该数值的内部格式（这可能不是你

所要的格式)。因此,可把该代码从 2 改变为 1(VARCHAR2)、3(INTEGER)、4(FLOAT)、5(STRING)或某些其它相应的数据类型。

库函数 sqlprc()分离精度和定标。通常,在 DESCRIBE SELECT LIST 之后使用它,并且它的第一个参数是 L[i]。该函数引用的格式如下:

```
sqlprc (long * length, int * precision, int * scale);
```

其中:

length:是指向一个长整型变量的指针,该长整型变量存储一个 ORACLE NUMBER 值的长度。长度被存放在 L[i]中,该值的定标和精度被存放在相应的低字节和下一个高字节中。

precision:是一个指向返回 NUMBER 值精度的整型变量指针,精度是有效位数。如果选择表项引用的是未指定大小的 NUMBER 值,则它被设置为 0。在这种情况下,因为未指定大小,所以可假定为最大精度。

scale:是指向一个返回 NUMBER 值定标的整型变量的指针。定标指出在什么地方舍入。例如,定标 2 意味着值在最接近百分之一位上舍入(3.456 变为 3.46);定标-3 意味着在最接近千位上舍入(3456 变为 3000)。

下面的例子表示怎样使用 sqlprc()函数来计算 NUMBER 值的最大显示长度。

```
/* 说明函数调用的变量 */
sqlda *select_des        /* 说明一个 SQLDA */
int prec;                /* 精度 */
int scal;                /* 定标 */
extern void sqlprc( );   /* 说明一个库函数 */
/* 分离精度和定标 */
sqlprc (&(select_des->L[i]), &prec, &scal);
/* 允许最大的 NUMBER 尺寸 */
if (prec==0)
    prec=40;
/* 允许小数点和符号 */
select_des->L[i]=prec+2;
```

对于某些 SQL 数据类型,函数 sqlprc()返回 0 作为精度和定标。函数 sqlpr2 和函数 sqlprc( )类似,除了表 5-2 所示的 SQL 数据类型外,它们有相同的参数表和返回值:

表 5-2 部分 SQL 数据类型的精度和定标

SQL 数据类型	二进制精度	二进制定标
FLOAT	126	-127
FLOAT(n)	n(范围 1…126)	-127
REAL	63	-127
DOVBLE PRECISION	126	-127

3. 处理 Null/Not Null 数据类型

对于每一个选择表列(不是表达式),DESCRIBE SELECT LIST 在选择描述区的 T 数据类型数组中返回一个 Null/Not Null 指示。如第 i 个选择表项被强制为 Not null,那么 T[i]的高位被清除,否则被设置。

在 OPEN 或 FETCH 语句中使用该数据类型之前,如果 Null 状态位被设置的话,则必须清除它。能使用库函数 sqlnul()来发现一个列是否允许 Null,并清除该数据类型的 Null 状态位。sqlnul( )函数的引用格式如下:

```
sqlnul (unsigned short * value_type,unsigned short * type_code,
                        int * null_status);
```

其中:

value_type:是一个指向无符号短整型变量的指针,该短整型变量存放一个选择表项的数据类型代码,该数据类型被存放在 T[i]中。

type_code:是指向无符号短整型变量的指针,该短整型变量返回该选择表列的数据类型代码,其高位被清除

null_status:是一个指向整型变量的指针,该整型变量返回选择表项的 Null 状态。1 意味着该列允许 Null;0 意味着该列不允许 Null。

下面是 sqlnul( )函数使用的例子。

```
/* 说明函数调用的变量 */
sqlda * select_des;        /* 说明一个 SQLDA */
unsigned short dtype;      /* Null 位被清除的数据类型代码 */
int nullok;                /* 1=null, 0=not null */
extern void sqlnul();      /* 说明库函数 */
/* 寻找是 Null 的列 */
sqlnul(&select_des→T[i], &dtype, &nullok);
if (nullok)
{    /* 允许 Nulls */
     ...
     /* 清除 null /not null 位 */
     sqlnul(&(select_des→T[i]), &(select_des→T[i]),&nullok);
}
```

5.5.4 动态方法 4 所用的 SQL 语句

在方法四中,处理动态 SQL 语句要用到如下几个嵌入 SQL 语句:

- PREPARE
- DECLARE CURSOR
- DESCRIBE
- OPEN
- FETCH
- CLOSE

下面主要描述 DESCRIBE 语句,其它语句已在前面各章中描述过,此处只说明其不同

点。

(1) OPEN 语句

OPEN 语句在方法 4 中的书写格式如下：

EXEC SQL OPEN cursor USING DESCRIPTOR descriptor_name；

其中 descriptor_name 是描述区名。

(2) FETCH 语句

FETCH 语句在方法 4 中的书写格式如下：

EXEC SQL FETCH cursor USING DESCRIPTOR descriptor_name；

其中 descriptor_name 是描述区名。

(3) DESCRIBE 语句

DESCRIBE 语句的主要功能是初始化一个描述区。即把动态 SQL 语句或 PL/SQL 块中的宿主变量描述信息保留到 SQLDA 中。

使用该语句之前必须先用 PREPARE 语句分析该动态 SQL 语句或 PL/SQL 块。该语句的书写文法如下：

EXEC SQL DESCRIBE [BIND VARIABLES FOR][SELECT LIST FOR]
[statement_name][block_name] INTO descriptor_name；

其中：

BIND VARIABLES FOR：初始化结合描述区，以保存 SQL 语句或 PL/SQL 块的输入宿主变量信息。

SELECT LIST FOR：初始化选择描述区，以保存 SELECT 语句的选择表项信息。

statement_name/block_name：标识预先用 PREPARE 语句分析的 SQL 语句或 PL/SQL 块。

descriptor_name：是要被初始化的描述区名。

使用该语句应注意如下几点：

· 在操作结合描述区或选择描述区之前，必须先发 DESCRIBE 语句。

· 不能把输入宿主变量和输出宿主变量二者描述到同一个描述区内。

· 由 DESCRIBE 语句所发现的变量数是分析的 SQL 语句或 PL/SQL 块中的虚拟宿主变量的总数，而不是唯一命名的虚拟变量的总数。

5.5.5 方法四的处理步骤

能用方法 4 来处理任何动态 SQL 语句，下面以查询为例来说明方法 4 的处理步骤。动态查询(SELECT)的处理一般采用如下 18 步：

1. 在说明段说明一个串型的宿主变量，以保存查询语句的文本。
2. 说明选择和结合 SQLDA。
3. 分配选择和结合描述区的空间
4. 设置能够被 DESCRIBE 的选择表项和虚拟输入宿主变量的最大个数。
5. 把查询文本存放在宿主串内。
6. PREPARE 宿主串中的查询语句
7. DECLARE 一个查询光标。

8. 把实输入宿主变量 DESCRIBE 进结合描述区中。

9. 把虚拟输入宿主变量的个数重新设置为 DESCRIBE 实际发现的个数。

10. 为 DESCRIE 发现的实输入宿主变量取值和分配存储空间。

11. 用 OPEN…USING 语句打开与结合描述区相对应的光标。

12. 用 DESCRIBE SELECT LIST 语句描述选择描述区。

13. 把选择表项的个数重新设置为 DESCRIBE 实际发现的个数。

14. 重新设置选择表项的长度和数据类型,以便显示它们。

15. 用 FETCH 把提取的行放入选择描述区所指向的数据缓冲区中。

16. 处理 FETCH 所返回的选择表值。

17. 释放选择表项、虚拟输入宿主变量、指示器变量和描述区所用的空间。

18. 关闭光标。

上述 18 步,并非任何时候都全要求,如果动态查询中的选择表项个数已知时,可以省略 DESCRIBE SELECT LIST 语句,而只需用方法三所用的 FETCH 语句即可:

EXEC SQL FETCH Cursor_name INTO host_Variable_list;

如果动态 SQL 语句中的虚拟输入宿主变量的个数已知时,可省略 DESCRIBE BIND VARIABLES 语句,而用方法三的 OPEN 语句:

EXEC SQL OPEN Cursor_name[USING host_Variables_List];

下面进一步详细描述每一步,为了避免混乱,对所引用的图作了如下限制:

- 限制描述区中的数组变量为 5 个元素
- 限制名字的最大长度为 5 个字符
- 限制值的最大长度为 10 个字符

(1) 说明一个串型宿主变量

程序需要说明一个串型宿主变量,以存放动态 SQL 语句的文本。宿主变量(在我们的例子中是 select_stmt)被说明为字符串。例如:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    int emp_number;
    VARCHAR emp_name[10]
    VARCHAR select_stmt[120]
    float bonus;
EXEC SQL END DECLARE SECTION;
```

(2) 说明 SQLDA

在我们的例子中,因为动态 SQL 语句(SELECT)中包含未知个数的选择表项和虚拟输入宿主变量,所以应该说明一个选择 SQLDA 和一个结合 SQLDA。这里是采用如下三个语句来说明它们的:

```
EXEC SQL INCLUDE sqlda;
sqlda * select_des;
sqlda * bind_des;
```

(3) 分配描述区的空间

用 sqlald()函数分配描述区的存贮空间，该函数的引用格式如下：

descriptor_name=sqlald(max_vars, max_name, max_ind_name);

sqlald()函数分配描述区所占用的存储空间(如图 5.3 所示)。

如果 max_name 是非 0，则分配由指针变量 S、M 和 C 寻址的数组。如果 max_ind_name 不是 0，则分配由指针变量 X、Y 和 Z 寻址的数组，如果 max_name 和 max_ind_name 是 0，则不分配相应的空间。

如果 sqlald()成功，则返回该描述区的地址。如果 sqlald()失败，则返回 0 指针。

在我们的例子中，用如下两个语句分配选择和结合描述区。

select_des=sqlald(3,5,0);

bind_des=sqlald(3,5,4);

对于选择描述区，总是把 max_ind_name 设置为 0，因此，不分配 X 寻址的空间。

(4) 设置 DESCRIBE 的最大个数

可用如下格式的赋值语句设置要描述的选择表项或虚拟输入宿主变量的最大个数：

select_des→N=3;

bind_des→N=3;

图 5-3 和 5-4 表示描述区的当前状态。其中图 5-3 中，指示器变量的各部分被划掉，表示不用它。

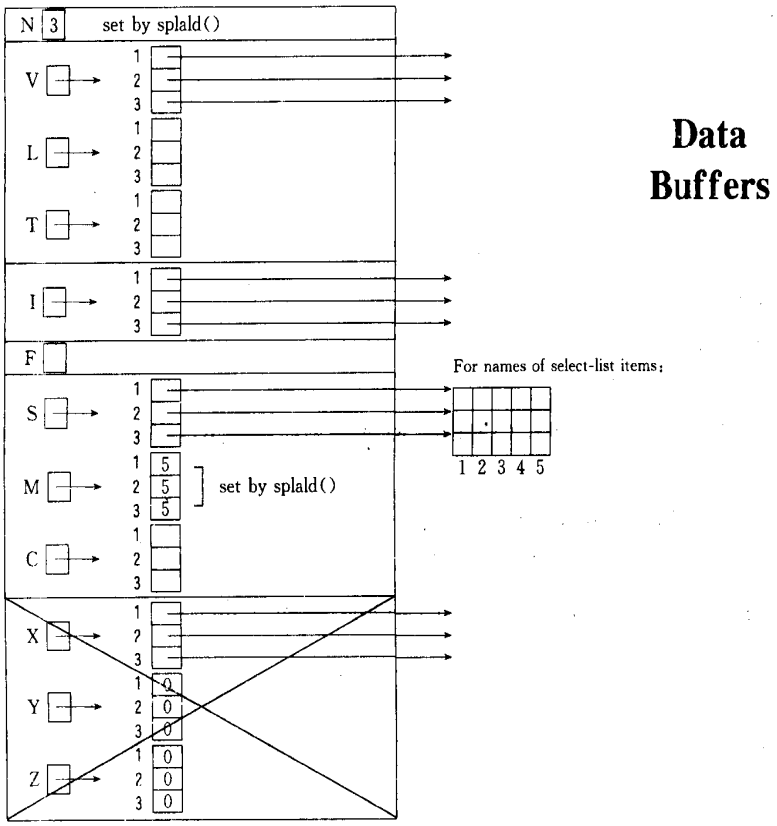


图 5-3 初始化选择描述区

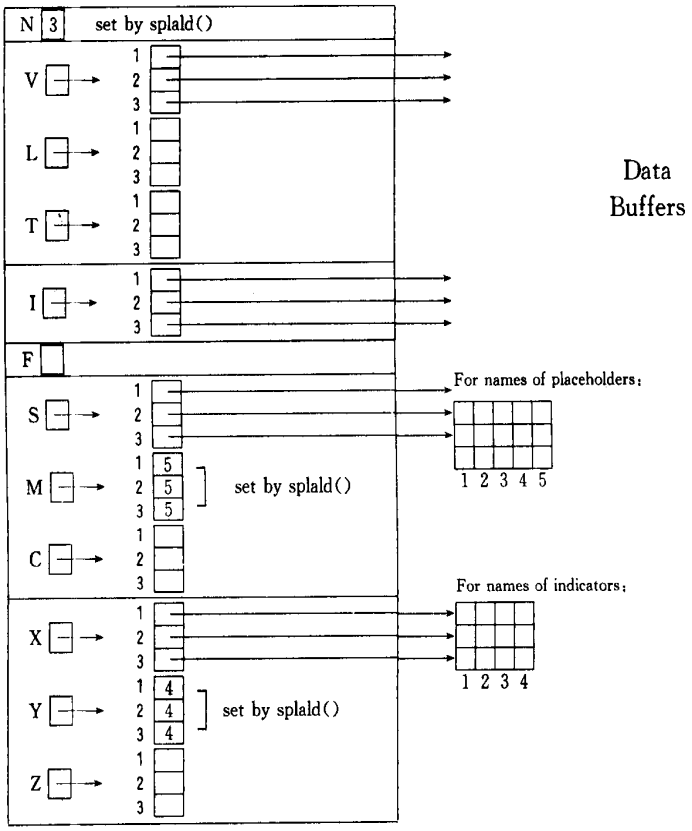


图 5-4　初始化结合描述区

(5) 把查询语句文本存入串型宿主变量内

程序提示用户输入和构造一个动态 SELECT 语句，然后把该 SQL 语句存放在 select_stmt 中。如下例：

```
printf("\n\nEnter SQL Statement:");
gets(select_stmt.arr);
select_stmt.len=strlen(select_stmt.arr);
```

假定用户输入如下的串：

"SELECT ENAME, EMPNO, COMM FROM EMP WHERE COMM<:bonus"

(6) 分析串型宿主变量中的查询语句

用 PREPARE 语句分析和命名该 SELECT 语句。在我们的例子中，PREPARE 分析串型宿主变量 select_stmt 中的 SQL 语句，并给它一个名字 sql_stmt. 如：

```
EXEC SQL PREPARE sql_stmt FROM: select_stmt;
```

(7) 说明一个光标

用 DECLARE CURSOR 语句说明一个光标，并把它与一个指定的 SELECT 语句相联系。在我们的例子中，DECLARE CURSOR 定义一个名为 emp_cursor 的光标，并把它与 sql_stmt 联系起来，如

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

其中 sql_stmt 是由 PREPARE 语句命名的语句名。

注意：在方法 4 中，对于所有的动态 SQL 语句(不只是查询语句)，都必须说明一个光

标。在非查询语句情况下，打开光标即执行该动态 SQL 语句。

(8) 描述(DESCRIBE)实输入宿主变量

DESCRIBE BIND VARIABLES 把虚拟输入宿主变量的描述信息放进结合描述区中。在我们的例子中，DESCRIBE 把 bonus 的描述信息放进 bind_des，如：

```
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
```

注意：

- 在 bind_des 前面一定不加冒号。
- DESCRIBE BIND VARIABLES 语句必须跟在 PREPARE 语句之后，和 OPEN 语句之前。

图 5.5 说明在 DESCRIBE 语句之后结合描述区的状态。此时 DESCRIBE 已把 F 设置为动态 SQL 语句中所包含的虚拟输入宿主变量的实际个数。

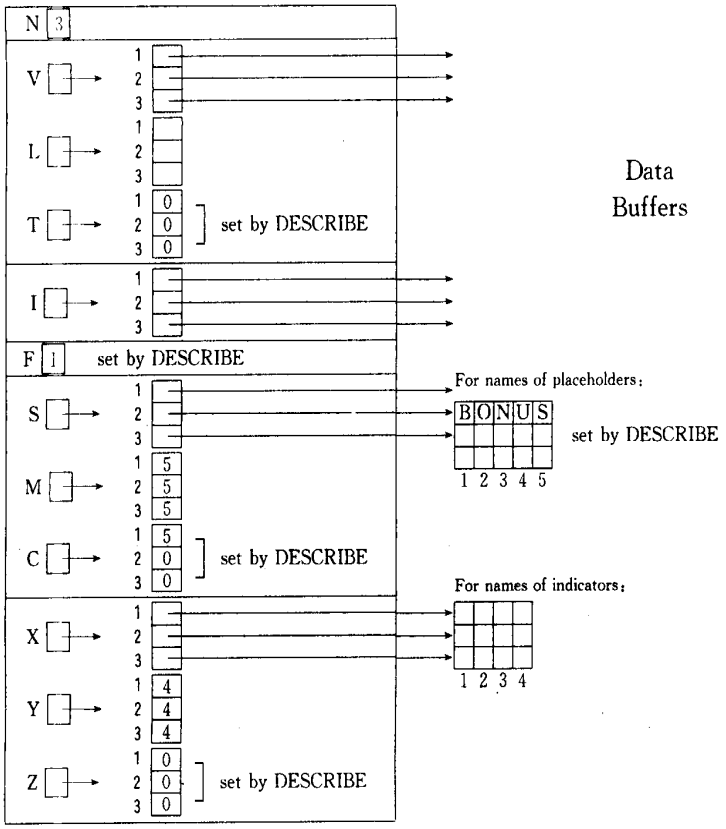


图 5-5 DESCRIBE 之后结合描述区的状态

(9) 重新设置虚拟输入宿主变量的个数

必须把虚拟输入宿主变量的最大个数重新设置为 DESCRIBE 实际发现的个数：

```
bind_des→N=bind_des→F;
```

(10) 为实输入宿主变量分配存储空间和取值

程序必须为实输入宿主变量分配存储空间和取值。如何取值可由程序的编码者决定。例如，可直接编写一段代码、或从一个文件读、或交互输入。

在我们的例子中，是用如下的代码段来实现的：它提示用户为实输入宿主变量输入值，然后进行空间分配：

```
for (i=0; i<bind_des->F; i++)
{
    /* 输入实输入宿主变量的值 */
    printf("\n Enter value of bind variable %. *s:\n?",
    (int) bind_des->C[i],bind_des->S[i]);
    gets (hostval);
    /* 设置值的长度 */
    bind_des->L[i]=strlen(hostval);
    /* 为实输入宿主变量分配存储空间 */
    bind_des->V[i]=malloc (bind_des->L[i]+1);
    /* 为指示器变量的值分配存储空间. */
    bind_des->I[i]=(unsigned short *) malloc(sizeof(short *));
    /*设置实输入宿主变量的值 */
    strcpy (bind_des->V[i], hostval);
    /* 设置指示器变量的值 */
    bind_des->I[i]=0;

    /* 数据类型设置为 VARCHAR2. */
    bind_des->T[i]=1;
}
```

假定用户为 bonus 提供一个 625 的值，图 5-6 表示结合描述区在赋值以后的状态。注意，值是以 Null 结束。

(11) 打开光标

OPEN 语句用于动态查询时，除了要把光标与结合描述区相联系外，其它与用于静态查询类似。在运行时它用缓冲区中的值来执行相应的 SQL 语句(如用与 bonus 相对应的值)来进行查询，并把光标设置在光标缓冲区的第一行。在我们的例子中，OPEN 把 emp_cursor 与 bind_des 相联系，如：

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR bind_des;
```

注意：不能在 bind_des 前面加冒号。

(12)描述(DESCRIBE)选择表

DESCRIBE SELECT LIST 语句用于把选择表项的描述信息(如每一个选择表项值的长度和数据类型)放入选择描述区中。因此，它必须被放置在 OPEN 语句之后和 FETCH 语句之前。在我们的例中用如下的语句：

```
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
```

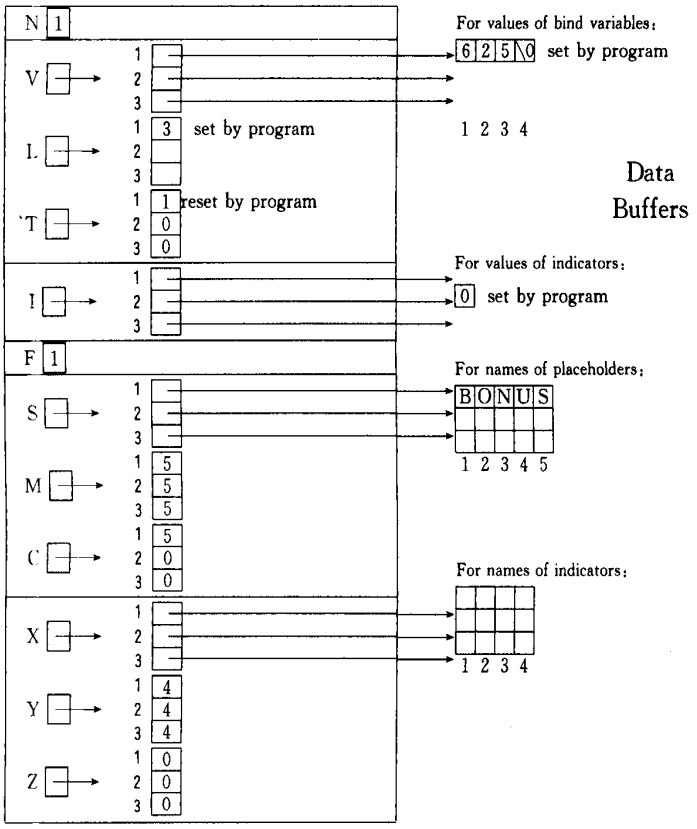


图 5-6　赋值后的结合描述区状态

请注意：

· DESCRIBE 把 F 设置为该查询选择表项中的实际项数，如果 SQL 语句不是查询，则被设置为 0。

· 此时 NUMBER 的长度仍然不能用，对于用 NUMBER 类型定义的列，还必须用库函数 sqlprc()来分离精度和定标。

(13) 重新设置选择表项的最大数

可用如下语句把选择表项的最大个数设置为 DESCRIBE 发现的实际个数：

```
select_des→N=select_des→F;
```

(14) 重新设置每一个选择表项的长度和数据类型

在我们的例子中，在用 FETCH 提取选择表项的值之前，要先用库函数 malloc()为它们分配存储空间。为了显示它们，还要重新设置长度和数据类型数组内的某些元素。其实现代码如下：

```
for (i=0; i<select_des→F; i++)
{
    /* 清除 NULL 位. */
    sqlnul (&(select_des→T[i]), &(select_des→T[i]), &nullok);
    /* 如果需要的话，再设置长度 */
    switch (select_des→T[i])
    {
```

```
        case 1:break;
        case 2:sqlprc (&select_des->L[i],&prec, &scal);
            if (prec==0) prec=40;
            select_des->L[i]=prec+2;
            if (scal<0) select_des->L[i]+=-scal;
            break;
        case 8:select_des->L[i]=240;
            break;
        case 11:select_des->L[i]=18;
            break;
        case 12:select_des->L[i]=9;
            break;
        case 23:break;
        case 24:select_des->L[i]=240;
            break;
    }
    /* 为选择表项的值分配存储空间 */
    select_des->V[i]=malloc(seelect_des->L[i]);
    /* 为指示器变量的值分配存储空间 */
    select_des->I[i]=(unsigned short *)malloc(sizeof(short *));
    /* 把所有的数据类型(除 LONG RAW 外)强制为 VARCHAR2. */
    if (select_des->T[i]! =24) select_des->T[i]=1;
}
```

图 5-8 表示 FETCH 之前的选择描述区。

注意:现在 NUMBER 的长度是可用的,而且所有的数据类型是 CHAR 。L[1]和 L[2]中的长度是 6 和 9,因为我们把描述的长度 4 和 7 增加了 2(考虑到正负号和小数点)。

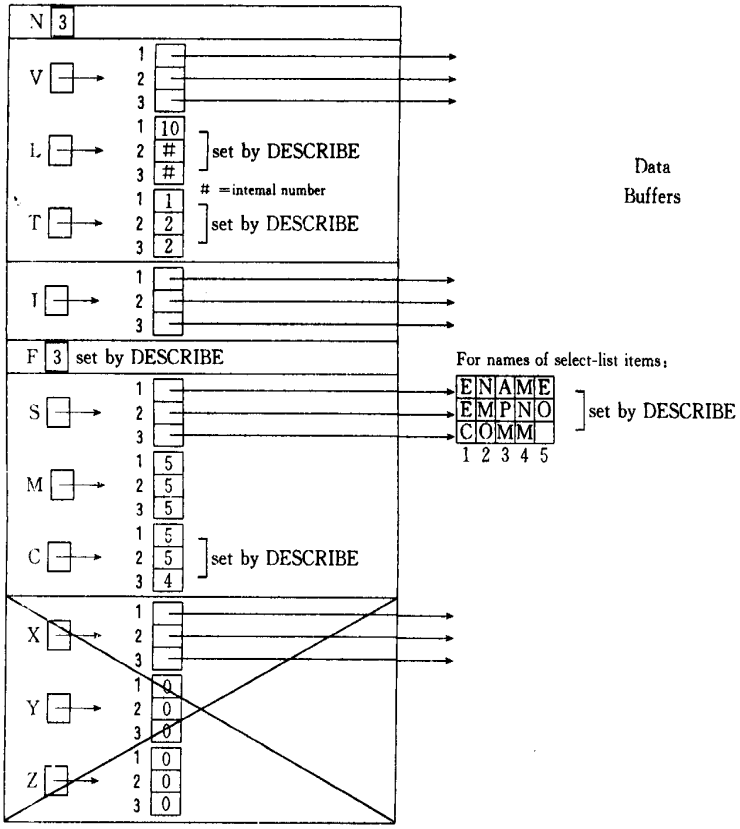


图 5-7　DESCRIBE 之后的选择描述区状态

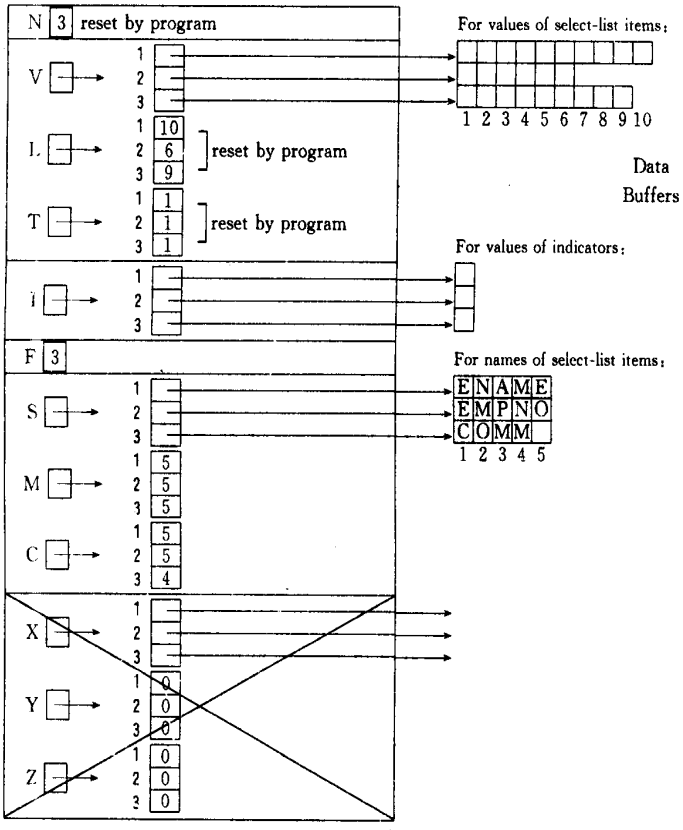


图 5-8　FETCH 之前选择描述区的状态

(15) 从光标缓冲区中提取行

用 FETCH 语句从从光标缓冲区中提取一行或多行。并把选择表项的值存入输出缓冲区，光标向前移动到下次提取的行。如果没有更多的行，FETCH 就把 sqlca. sqlcode 设置为“no data found”ORACLE 错误码。在我们的例子中，FETCH 把列 ENAME、EMPNO、和 COMM 的值返回给 select_des，如：

EXEC SQL FETCH emp_cursor USING DESCRIPTOR select_des;

图 5-9 表示 FETCH 之后选择描述区的状态。ORACLE 把选择表项和指示器的值存入由 V 和 I 的元素所寻址的输出数据缓冲区中。

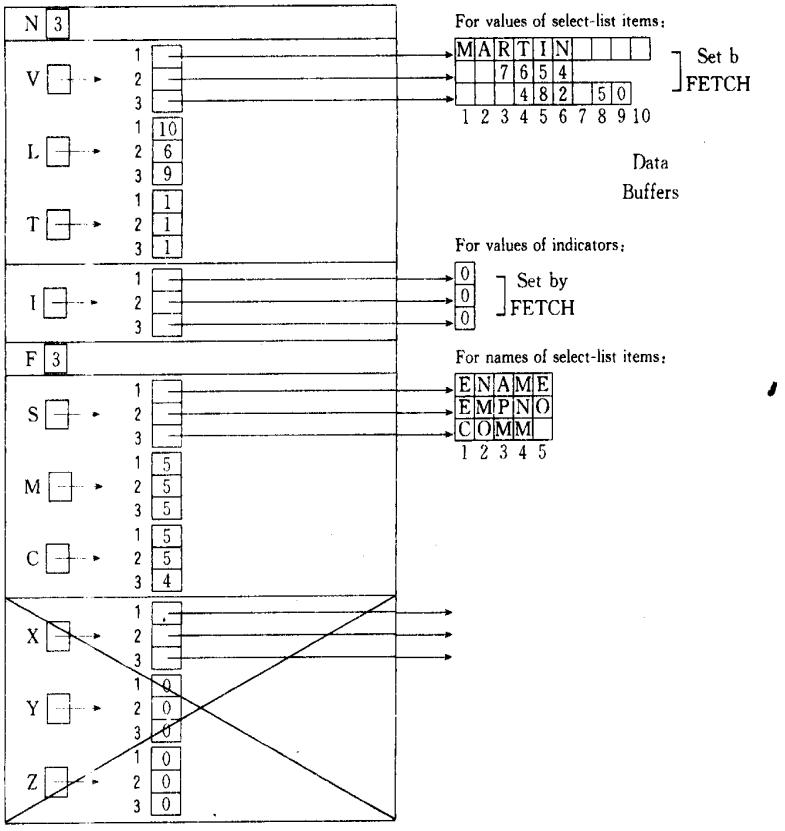


图 5-9 FETCH 之后的选择描述区

在输出缓冲区中，对于数据类型 1，ORACLE 使用存放在 L 数组中的长度，左对齐 CHAR 或 VARCHAR2 数据，右对齐 NUMBER 数据。

值‘MARTIN’从表 EMP 的 VARCHAR2(10)列中检索出来，使用 L[0]中的长度，ORACLE 左对齐 10 字节字段中的值，以填充缓冲区。

值 7654 从 NUMBER(4)列中检索出来，并强制为‘7654’。但是，L[1]中存放的长度(此处为 6)比它大 2，以考虑正负号和小数点。因此，ORACLE 右对齐 6 字节字段中的值。

值 482.50 从 NUMBER(7,2)列中检索出来，并强制为‘482.50’。同时 L[2]中保存的长度增 2，因此，ORACLE 右对齐 9 字节字段中的值。

(16) 取和处理选择表项的值

在 FETCH 之后，程序就能处理返回的值。在我们的例子中，要处理的是 ENAME、EM-

PNO 和 COMM 列的值。

(17) 撤消存储分配

使用 free()库函数撤消由函数 malloc()分配的存储空间,free 函数的调用格式如下:

```
free(pointer);
```

在我们的例子中,需要撤消为选择表项、实输入宿主变量和指示器变量的值所分配的存储空间。其语句如下:

```
/* 对于选择描述区 */
for (i=0; i<select_des->F; i++)
{
    free( select_des->V[i];
    free(select_des->I[i];
}
/* 对于结合描述区 */
for (i=0; i<bind_des->F; i++)
{
    free (bind_des->V[i]);
    free (bind_des->i[i]);
}
```

用 sqlclu()库函数撤消描述区本身的存储空间。该函数的格式如下:

```
sqlclu(descriptor_name);
```

描述区必须已经用 sqlald()分配过空间,否则结果将是不可预见的。在我们的例子中,用如下语句撤消选择和结合描述区的存储空间:

```
sqlclu(select_des);
sqlclu(bind_des);
```

(18) 关闭光标

CLOSE 使光标无效。在我们的例子中,CLOSE 使 emp_cursor 无效。如:

```
EXEC SQL CLOSE emp_cursor;
```

§5.5.6 方法4的应用举例

该例说明使用动态 SQL 方法4所需的基本步骤。用户能输入任何一个有效的 SQL 语句,它至多能包含40个实输入宿主变量和40个选择表项。程序至多能接收1023个字符的多行 SQL 语句。

例5.4 方法4例子

```
#  include <stdio.h>
#  include <string.h>
#  include <setjmp.h>
/* 定义选择表项和结合变量的最大个数 */
#  define MAX_ITEMS 40
```

```
    /* 定义选择表项和指示器变量名字的最大长度 */
    #define MAX_VNAME_LEN 30
    #define MAX_INAME_LEN 30
    #ifndef NULL
    #define NULL 0
    #endif
    char *dml_commands[]=("ESLECT","select",
                          "INSERT","insert",
                          "UPDATE","update",
                          "DELETE","delete");
    /* 说明串型宿主变量,以存放动态 SQL 语句 */
    EXEC SQL BEGIN DECLARE SECTION;
        char sql_statement(1024);
        EXEC SQL VAR sql_statement IS STRING(1024);
    EXEC SQL END DECLARE SECTION;
    /* 说明 SQLCA */
    EXEC SQL INCLUDE sqlca;
    /* 说明 SQL 描述区 */
    SQLDA *bind_dp;
    SQLDA *select_dp;
    /* 外部函数说明 */
    extern SQLDA *sqlald();
    extern void sqlnul();
    jmp_buf jmp_continue; /* buffer to hold longjmp info */
    int parse_flag=0; /* 错误处理标志 */
/*********************************
 *  函数名:主函数
 *  说明:程序的处理步骤如下:
 *        (1)登录到 ORACLE,
 *        (2)用 sqlald()为描述区分配存储空间,
 *        (3) 提示用户输入 SQL 语句,
 *        (4)分析该语句,说明一个光标,
 *        (5)用 DESCRIBE BIND 检查实输入宿主变量,
 *        (6)打开光标,
 *        (7)用 DESCRIBE 描述选择表项,
 *        (8)提取(FETCH)和显示每个数据行,
 *        (9)关闭光标。
 *********************************/
main()
```

```
{
  /* 函数说明 */
  int oracle_connect();
  int alloc_descriptors (/* int, int, int */);
  int get_sql_statement(/* void */);
  void set_bind_variables(void);
  void process_select_list(void);
  /* 自动量说明 */
  int i;
  /* 登录到 ORACLE. */
  if (oracle_connect()!=0)
      exit(1);     /* 登录失败 */
  /* 为选择和结合描述区分配空间 */
  if (alloc_descriptors (MAX_ITEMS, MAX_VNAME_LEN,
              MAX_INAME_LEN) !=0)
      exit(1); /* 分配失败 */
  /* 处理 SQL 语句 */
  for (;;)
  {
      i=setjmp(jmp_continue);
  /* 取语句,失败时结束 */
  if (get_sql_statement() !=0)
          break;
  /*分析该语句,并说明一个光标. */
  EXEC SQL WHENEVER SQLERROR DO sql_error();

  parse_flag=1; /*设置分析语句的错误标志*/
  EXEC SQL PREPARE S FROM :sql_statement;
  parse_flag=0; /* 设置分析语句的成功标志 */
  /* 说明光标 */
  EXEC SQL DECLARE C CURSOR FOR S;
  /* 为 SQL 语句中的虚拟输入宿主变量设置结合变量 */
  set_bind_variables();
  /* 打开光标和执行该语句,如果该语句不是 SELECT,
      则在 OPEN 之后处理结束 */
  EXEC SQL OPEN C USING DESCRIPTOR bind_dp;
  /* 调用 process_select_list 函数来处理选择表项,
    如果该语句不是 SELECT,则不作任何处理即返回。
  */
```

```
  process_select_list();
  /* 告诉用户处理了多少行 */
  for (i=0; i<8; i++)
  {
    if (strncmp(sql_statement, dml_commands[i], 6)==0)
    {
      printf("\n\n%d row%c processed. \n",
              sqlca.sqlerrd[2], sqlca.sqlerrd[2]==1? '\0' : 's');
      break;
      }
    }
  } /*语句处理循环结束 */
  /* 释放结合和选择描述区中为指针分配的存储空间 */
  for (i=0; i<MAX_ITEMS; i++)
  {
        if (bind_dp->V[i] !=(char *)NULL)
            free (bind_dp->V[i]);
        free(bind_dp->I[i]); /* MAX_ITEMS were allocated. */
        if (select_dp->V[i] !=(char *) NULL)
                free (select_dp->V[i]);
        free(selectdp_I[i]); /* MAX_ITEMS were allocated. */
  }
  /* 释放 SQLDA 所用的空间 */
  sqlclu (bind_dp);
  sqlclu (select_dp);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  /*关闭光标 */
  EXEC SQL CLOSE C;
  /* 结束处理,退出 ORACLE */
  EXEC SQL COMMIT WORK RELEASE;
  printf("\nHave a good day!\n");
  return;
  } /* 主函数结束 */
/*************************
 *  函数名: oracle_connect()
 *  说明:登录函数
 *************************
oracle_connect()
{
```

```
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR username(20);
        VARCHAR password(20);
    EXEC SQL END DECLARE SECTION;

    strcpy (username.arr"SCOTT");
    username.len=strlen(username.arr);
    strcpy (password.arr,"TIGER");
    password.len=strlen(password.arr);
    EXEC SQL WHENEVER SQLERROR GOTO connect_error;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user %s. \n",username.arr);
    return 0;
connect_error:
    fprintf (staderr, "Cannot connect to ORACLE as user %s\n",
            username.arr)
    return -1;
}
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*   函数名:alloc_descriptors()
*   说明:
*     用 sqlald()函数分配结合和选择描述区,
*     分配指向描述区中每个指示器变量的指针,
*     用 set_bind_variables() 或 process_select_list()函数
*     重新分配指向结合变量和选择表项的指针,
* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
alloc_descriptors(size, max_vname_len, max_iname_len)
int size; /* 描述区中数组元素的最大数 */
int max_vname_len; /* SQL 语句中选择表项名字的最大长度,最大30 */
int max_iname_len; /* 指示器变量名字的最大长度,最大30 */
{
    int i;
    /* 分配结合描述区别 */
    if ((bind_dp=
      sqlald(size, max_vname_len, max_iname_len))==(SQLDA *)
            NULL)
    {
      /* 分配失败 */
      fprintf(stderr,
```

```
            "Cannot allocate memory for bind descriptor.");
          return -1;
          }
        /* 分配选择描述区 */
        if ((select_dp =
              sqlald(size, max_vname_len, max_iname_len)) == (SQLDA *)
             NULL)
        {
          /* 分配失败 */
          fprintf(stderr,
            "Cannot allocate memory for select descriptor.");
            return -1;
        }
        select_dp->N=MAX_ITEMS;
        /* 分配指向缓冲区的指针 */
        for (i=0; i<MAX_ITEMS; i++)
             bind_dp->I[i] = (short *) malloc(sizeof(short *));
        for(i=0; i<MAX_ITEMS; i++)
             bind_dp->V[i] = (char *) malloc(max_vname_len);
        /* 分配选择变量指针 */
        for(i=0; i<MAX_ITEMS; i++)
             select_dp->I[i] = (short *) malloc(sizeof(short *));
        for (i=0; i<MAX_ITEMS; i++)
            select_dp->V[i]=(char *) malloc (max_vname_len);
        return 0;
    }
    /* * * * * * * * * * * * * * * * * * * * * * * *
    *  函数名: get_sql_statement ()
    *  说明:取 SQL 语句
    * * * * * * * * * * * * * * * * * * * * * * * * */
    get_sql_statement (/* void */)
    {
        char *cp, linebuf[256];
        int iter;
        /* ovid */ help (/* void */)
        for (iter =2; ;)
        {
          if (iter ==2)
          {
```

```
        printf ("\nSQL");
        sql_statement [0]='\0';
      }
      fgets(linebuf, sizeof (linebuf), stdin);
      cp=strrchr (linebuf, '\n');
      if (cp && cp != linebuf)
        *cp='';
      else if (cp==linebuf)
        continue;
      if (strncmp (linebuf, "exit", 4) ==0||
        strncmp (linebuf, "EXIT", 4) ==0)
      {
        return -1;
      }
      else if(linebuf[0] == '?' ||
        strncmp(linebuf, "help", 4) == 0||
        strncmp(linebuf, "HELP", 4) == 0)
      {
        help();
        iter=2;
        continue;
      }
      strcat (spl_statement, linebuf);
      if ((cp=strrchr(sql_statement, ';'))!=(char *)NULL)
      {
        *cp='\0';
        break;
      }
      else
      {
        printf("% 3d", iter++);
      }
    }
    return 0;
}
/*****************************************
*    函数名: set_bind_varlables (/* void */)
*    说明: 设置结合变量,即把结合变量的描述信息存入SQLDA
*****************************************/
```

```
/* void */
set_bind_varlables (/* void */)
{
  int i, n;
  char bind_var[64];

  /* 描述结合变量(即输入宿主变量) */
  EXEC SQL WHENEVER SQLERROR DO sql_error();
  /* 初始化数组元素个数 */
  bind_dp->N=MAX_ITEMS;
  EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_dp;

  /* 如果 F 是负,则结合变量过多 */
  if (bind_dp->F<0)
  {
    printf
      ("\nToo many bind variables (%d), maximum is %d\n,",
        --bind_dp->F, MAX_ITEMS);
    return;
  }
  /* 把 SQLDA 中的数组元素最大数设置为 DESCRIBE 发现的最大数 */
  bind_dp->N=bind_dp->F;
  /* 取每一个结合变量的值及有关信息:C[i],S[i], L[i], V[i],T[i], I[i] */
  for (i=0; i<bind_dp->F; i++)
  {
    printf ("\nEnter value for bind variable %. *s:",
            (int)bind_dp->C[i], bind_dp->S[i]);
    fgets (bind_var, sizeof( bind_var), stdin);
    /* 取长度,并把它放入 SQLDA 的 L[i] 中 */
    n=strlen(bind_var) -1;
    bind_dp->L[i]=n;
    /* 为值分配缓冲区,并把其值拷贝到缓冲区中 */
    bind_dp->V[i]=(char *) realloc(bind_dp->V[i],
                        (bind_dp->L[i] +1));
    strncpy (bind_dp->V[i], bind_var, n);
    /* 设置指示器变量的值 */
    if ((strncmp(bind_dp->V[i], "NULL", 4) ==0)||
      (strncmp (bind_dp->V[i], "null", 4)==0))
        * bind_dp->I[i] =-1;
```

```
    else
        *bind_dp->I[i]=0;
    /* 把结合数据类型设置为1(CHAR) */
    bind_dp->T[i] =1;
    }
}

/**********************************
*     函数名: process_select_list (/* void */)
*     说明:处理选择表项
**********************************/
/* void */
process_select_list (/* void */)
{
    int i, null_ok, precision, scale;
    /* 是否是 SELECT 语句? */
    if ((strncmp(sql_statement, "SELECT", 6) !=0) &&
    (strncmp(sql_statement, "select", 6) !=0))
    {
    /* 非 SELECT 语句 */
    select_dp->F=0;
    return;
    }
/*若是 SELECT,则描述选择表项。DESCRIBE 返回名字,数据类型,长度(包括精度
   和定标)和每一项的 Null/Not Null 位状态 */
select_dp->N=MAX_ITEMS;
EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_dp;
/* 如果 F 是负,则选择表项过多 */
if (selct_dp->F<0)
{
        printf
        ("\nToo many select_list items (%d), maximum is %d\n",
          -(select_dp->F), MAX_ITEMS);
        return;
    }
    /* 把 SQLDA 中的 F 变量设置为实际数 */
    select_dp->N=select_dp->F;
    /* 为每个选择表项分配存储存 */
    /* sqlprc():从 L[i]中分离精度和定标,
```

```
* sqlnul():再设置数据类型的高位,并检查该列是否是 NOT NULL,
*
* CHAR   该数据类型有长度,但没有0精度和定标,
*        长度是在 CREATE 时定义的。
* NUMBER   该数据类型只有在 CREATE 时定义,
*        才有精度和定标。若列定义就是 NUMBER,则精度和定标是0,
*        这时必须分配最大长度。
*
* DATE   用缺省格式时,返回长度7。
*        为了存储实际日期字符串,长度应该增至9。
*        若使用 TO_CHAR 函数,则最大长度应该是75。*/
* ROWID        如果强制为 CHAR,则总返回18长度,
*
* LONG 和
* LONG RAW 该数据类型返回0长度,因此必须设置一个最大长度。
*          本例是240个字符。
*/
printf ("\n");
for (i=0;i<select_dp->F; i++)
{
    /* 断开数据类型的高位(NOT NULL) */
    sqlnul (&(select_dp->T[i]), &(select_dp->T[i]), &null_ok);
    switch (select_dp->T[i])
    {
      case 1: /* CHAR 数据类型:不改变长度
                    对 TO_CHAR 转换可能例外(此处不处理) */
          break;
      case 2: /* NUMBER 数据类型:用 sqlprc()来分离精度和定标 */
          sqlprc (&(select_dp->L[i]), &precision, &scale);
            /* 允许 NUMBER 的最大尺寸 */
          if (precision==0)
              precision=40;
          /*考虑小数点和符号位 */
          select_dp->L[i]=precision +2;
          break;
      case 8: /* LONG 数据类型 */
          select_dp->L[i]=240;
          break;
      case 11: /* ROWID 数据类型 */
```

```
            select_dp->L[i]=18;
            break;
        case 12; /* DATE 数据类型 */
            select_dp->L[i]=9;
            break;
        case 23: /* RAW 数据类型 */
            break;
        case 24; /* LONG RAW 数据类型 */
            select_dp->L[i]=240;
            break;
        }
        /* 为选择表项值分配空间 */
        select_dp->V[i]=(chat *)realloc(select_dp->V[i],
                                        select_dp->L[i] +1);
        /*打印列标题,对 NUMBER 列右对齐 */
        if (select_dp->T[i]==2)
          printf ("%. *s",select_dp->L[i],select_dp->S[i]);
        else
          printf ("%-. *s", select_dp->L[i],select_dp->S[i]);
      /*把所有的数据类型强制为字符型,除 LONG RAW 外 */
      if (select_dp->T[i] !=24)
      select_dp->T[i]=1;
}
printf("\n\n");
/* FETCH 每一行,并打印其列值 */
EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;
for (; ;)
{
    EXEC SQL FETCH C USING DESCRIPTOR select_dp;
    /* 因为返回的值已被强制为字符串,因此这里的处理是必要的,
    该函数显示值。*/
    for (i =0; i<select_dq->F; i++)
    {
      if (*select_dp->I[i]== 0)
              printf ("%-*c", (int)select_dp->L[i], '');
      else
        printf ("%-*. *s", (int)select_dp->L[i],
                select_dp->V[i]);
    }
```

```
        printf("\n");
      }
end_select_loop:
    return;
}
/* * * * * * * * * * * * * * * * * * * * * * *
*     函数名:help()
*     说明:显示邦助信息
* * * * * * * * * * * * * * * * * * * * * * * * */
/* void */
help(/* void */)
{
    printf ("\nEnter a SQL statement at the SQL prompt. \n");
    printf ("Use standard (not embedded) SQL syntax. \n");
    printf ("Statements can continue over several lines,but \n");
    printf ("must be terminated with a semicolon. \n");
    printf ("Typing \"exit\"(no semicolon) exits the program.\n");
    printf ("Typing \"?\" or \"help\"gets this message. \n\n");
}
/* * * * * * * * * * * * * * * * * * * * * * * * *
*     函数名:sql_error()
*     说明:进行错误处理
* * * * * * * * * * * * * * * * * * * * * * * * * */
sql_error (/* "void" */)
{
    int i;
    /* ORACLE 错误处理程序 */
    printf ("\n\n%.70s\n",sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
        printf
        ("Parse error at character offset %d in SQL statement.\n",
          sqlca.sqlerrd[4]);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp (jmp_continue, 1);
}
```

§5.6 宿主数组在动态方法中的应用

在动态 SQL 方法中也可使用宿主数组,其用法类似于静态 SQL。例如,在动态方法2

中，可在 EXECUTE 语句使用输入宿主数组：

EXEC SQL EXECUTE Statement_name USING host_array_list;

其中 host_array_list 表示一个或多个数组。

在方法3中，可在 OPEN 语句中使用输入宿主数组，而在 FETCH 语句使用输出宿主数组：

EXEC SQL OPEN cursor_name USING host_array_list;

.....

EXEC SQL FETCH cursor_name INTO host_array_list;

为了在动态 SQL 方法4中使用输入或输出数组，必须在 EXECUTE 或 FETCH 语句中使用 FOR 子句来告诉 ORACLE 要处理的数组元素个数(这是必须的，因为 ORACLE 没有其它方法知道宿主数组的大小)，并在 V[i] 和 L[i]中用如下格式的语句来设置第 i 个输出数组(选择表项)或实输入数组的地址和元素数：

V[i]=array_address;

L[i]=element_size;

其中 arrary_address 是数组的地址、element_size 是该数组的元素个数。

下面是使用数组的例子，该例使用了三个输入数组。注意：对于方法4来说，EXECUTE 语句只能被用于非查询语句。

例5.5 数组在动态方法中的应用

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *     说明：该例向 EMP 表中插入若干行
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdio.h>
#include <string.h>
/* 说明段 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username[21];
    VARCHAR password[21];
    char * sql_stmt=
    "INSERT INTO EMP (EMPNO, ENAME, SAL) VALUES (:e, :n, :s)";
    int array_size=5;
EXEC SQL END DECLARE SECTION;
/* 说明 SQLCA 和 SQLDA */
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE sqlda;
SQLDA * binda;
说明外部量 */
char names[5][10];
int numbers[5];
float salary[5];
```

```
/* 说明函数 */
extern SQLDA *sqlald();
main()
{
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    /* 登录到 ORACLE. */
    strcpy (username.arr, "SCOTT");
    username.len =strlen (username.arr);
    strcpy (password.arr, "TIGER");
    password.len =strlen (password.arr);
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf ("Connected to ORACLE. \n");
    /* 分配描述区和设置 N 字段。
       必须在 DESCRIBE 之前做. */
    binda=sqlald (3, 5, 0);
    binda→N=3;
    /* 分析和描述 SQL 语句 */
    EXEC SQL PREPARE stmt FROM :sql_stmt;
    EXEC SQL DESCRIBE BIND VARIABLES ROF stmt INTO binda;
    /* 初始化 SQLDA */
    binda→V[0]=numbers;
    binda→L[0]=sizeof (int);
    binda→T[0]=3;
    binda→I[0]=0;
    binda→V[1]=&names[0][0];
    binda→L[1]=10;
    binda→T[1]=1;
    binda→I[1]=0;
    binda→V[2]=salary;
    binda→L[2]=sizeof (float);
    binda→T[2]=4;
    binda→I[2]=0;
    /* 初始化数据缓冲区 */
    strcpy (&names[0][0],"LI PING");
    numbers[0]=1014;
    salary[0]=1034.50;
    strcpy (&names[1][0],"ZHANG LIANG");
    numbers[1]=1015;
    salary[1]=693.15;
```

```
strcpy (&names[2][0],"WANG XIAOJIE");
numbers[2]=1016;
salary[2]=953.43;
strcpy (&names[3][0],"CUI YEYING");
numbers[3]=1017;
salary[3]=735.48;
strcpy (&names[4][0], "WESTON");
numbers[4]=1018;
salary[4]=815.63;
/* 进行 INSERT. */
printf ("Adding to the Sales force... \n");
EXEC SQL FOR :array_size
    EXECUTE stmt USING DESCRIPTOR binda;
/*输出处理的行数*/
printf ("%d rows inserted. \n\n",sqlca.sqlerrd[2]);
/* 结束处理*/
EXEC SQL COMMIT WORK RELEASE;
exit(0);
/*错误处理 */
sql_error:
    /*打印错误信息 */
    printf("\n%.70s",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit (1);
}
```

§5.7 在动态方法中使用 PL/SQL 块

ORACLE 预编译程序处理 PL/SQL 块的方法类似于处理单个 SQL 语句。一个 PL/SQL 块也能用一个串宿主变量或文字串来存储，但不应存储 EXEC SQL EXECUTE、END-EXEC 和语句终结符。预编译程序处理 PL/SQL 与处理 SQL 语句有两点差别：

·预编译程序把所有的 PL/SQL 宿主变量作输入宿主变量处理，不管它们在 PL/SQL 块内是用来代替输入或输出宿主变量(或二者)。

·不能用 PL/SQL 进行 FETCH，因为它可能包含任意数量的 SQL 语句。

如果 PL/SQL 块不包含宿主变量，则可使用方法1执行 PL/SQL 串。

如果 PL/SQL 块包含已知数量的输入和输出宿主变量，则可使用方法2来分析和执行该 PL/SQL 串。

必须把所有的宿主变量放在 USING 子句内。当执行 PL/SQL 串时，USING 子句中的宿主变量替换分析串中的相应虚拟宿主变量。尽管预编译程序把所有 PL/SQL 块中的宿主

变量作为输入宿主变量处理，但还是能正确赋值。输入的值被赋给输入宿主变量，而输出(列)的值被赋给输出宿主变量。

每一个分析的 PL/SQL 串中的虚拟宿主变量必须与 USING 子句中的实宿主变量相对应。

对于方法3，除了允许 FETCH 以外，其余与方法2相同。由于方法2不允许用 PL/SQL 来 FETCH，所以只好用方法3。

如果 PL/SQL 块包含一个未知数量的输入或输出宿主变量，则必须用方法4。为了使用方法4，对于所有的输入和输出宿主变量应建立一个结合描述区。执行 DESCRIBE BIND VARIABCES 就把所有的输入和输出变量的信息存放在结合描述区中。因为预编译程序把所有的 PL/SQL 宿主变量作为输入宿主变量来处理，所以不需执行 DESCRIBE SELECT LIST 语句。

第六章　编写 SQL * FORMS 的用户出口

§6.1　SQL * FORMS 用户出口的概念

6.1.1　什么是用户出口

用户出口是用宿主语言(如 C 语言)写的函数或子程序。它由应用开发者编写并由 SQL * FORMS 调用,用于完成某一特殊任务的处理。在用户出口中能嵌入 SQL 命令和 PL/SQL 块。可类似于 PRO 程序那样来预编译它。

当 SQL * Forms 的触发器调用一个用户出口时,该用户出口开始执行,执行后返回一个状态码给 SQL * Forms(参考图 6－1)。用户出口能在 SQL * Forms 的状态行上显示信息,也能取 Form 中的字段值和把一个值放置在 Form 的字段中;还能对 ORACLE 数据库中的数据进行操纵、计算和查询,甚至登录到不同的数据库上。

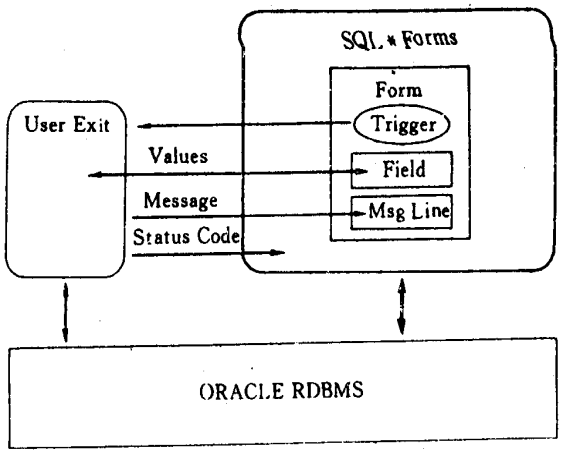


图 6-1　由 SQL * Forms 调用的用户出口

6.1.2　什么时候需写用户出口

由于 SQL * Forms 允许在触发器中使用 PL/SQL 块。因此,一般可不调用用户出口,而用 PL/SQL 过程。

因为编写用户出口要比编写 SQL、PL/SQL 或 SQL * Forms 命令更难。因此,只有 *SQL、PL/SQL 和 SQL * Forms 范围以外的处理才使用用户出口。一般在下述情况下才使用* 用户出口:

- 使用第三代语言(像 C 和 FORTRAN)能更快更容易做的操作(如数值积分),
- 控制实时设备或处理(例如,对打印机或图形设备发一系列命令),
- 需要扩展过程能力的数据操作(例如,递归分类),

·特殊文件的 I/O 操作。

6.1.3 开发用户出口的步骤

用户出口从编写到加入到 Form 内，需经如下七步：

1. 编写用户出口，

2. 预编译源代码，

3. 编译预编译产生的源代码，

4. 在表 AIPXTB 上加一项：用 GENXTB 实用程序把一项加到 IAPXIT 模块中的 IAP 程序表 IAPXTB 上(IAP 是 SQL * Forms 的一个成分。

5. 建立一个新的 IAP：通过连接标准的 IAP 模块、修改的 IAPXIT 模块和新的用户出口模块来建立。

6. 定义一个触发器：在 Form 中定义一个调用该用户出口的触发器。

7. 执行 Form：当运行该 Form 时，指示操作使用新的 IAP(如果不用新的 IAP 替换标准的 IAP 的话)。

§6.2 如何编写用户出口

6.2.1 编写用户出口所用的语句

可用如下类型的语句来编写 SQL * Forms 的用户出口：

·宿主语言(如 C)

·EXEC SQL：嵌入式 SQL 语句。

·EXEC ORACLE

·EXEC IAF GET

·EXEC IAF PUT

其中 EXEC IAF GET 和 PUT 语句是 Form 和用户出口的接口语句，在此作详细介绍，其它语句在前面各章中已有详述，在此就不再赘述。

6.2.2 变量

在 EXEC IAF GET 和 EXEC IAF PUT 语句中能使用宿主变量。但应注意以下几点：

·宿主变量必须与 Form 定义中所用的字段名相对应，

·宿主变量必须在用户出口的说明段说明，

·在 EXEC IAF 语句中必须以冒号(:)开头，

·不许在 EXEC IAF GET 和 PUT 语句中使用指示器变量。

6.2.3 IAF GET 语句

用户出口使用 IAF GET 语句来从 Form 字段上“取”值，并把这些值赋给宿主变量。而后用户出口就可使用这些宿主变量中的值来进行计算、数据操纵、更新等。GET 语句的文法如下：

EXEC IAF GET field_name1,field_name2,……

INTO :host_variable1,:host_variable2,……

其中 field_name 可能是下列 SQL * Forms 变量中的任何一种：

- 字段
- 块字段
- 系统变量
- 全程变量
- 包含一个字段、块字段、系统变量或全程变量值的宿主变量(用冒号开头)。

host_variable 是宿主变量。

例如下面的语句把 Form 字段 employee.name 中的值取出并赋给宿主变量 new_name：

EXEC IAF GET employee.name INTO :new_name;

所有的字段值都是字符串。如果字段和宿主变量的数据类型不一致，GET 把字段值转换为对应的宿主变量的数据类型。如果试图转换一个非法的或不支持的数据类型，就产生非法转换错误。

在上边的 IAF GET 语句中，是采用显式来指定块字段名（如 employee.name)。也能用一个串型宿主变量来指定块和字段名(叫隐式)，例如：

```
strcpy(blkfld,"employee.name");
EXEC IAF GET :blkfld INTO :new_name;
```

串型宿主变量中包含的必须是完整的块字段名(如 employee.name)。例如下面的用法是无效的：

```
strcpy(blk,"employee");
strcpy(fld,"name");
EXEC IAF GET :blk.:fld INTO :new_name;
```

在 GET 语句的字段表中，能混合使用显式和隐式两种方式，但不能以单个字段名引用。例如，下面的用法是非法的：

```
strcpy(fld,"name");
EXEC IAF GET employee.:fld INTO :new_name;
```

如果字段名的引用是模糊的(因为没有指定块名)，则 EXEC IAF 是无效的。对 Form 字段的无效或模糊引用就产生一个错误。

6.2.4 IAF PUT 语句

在用户出口中使用该语句能把常量和(或)宿主变量中的值放入 Form 的字段中。于是用户出口能在 SQL * Forms 的屏幕上显示任何一个值或信息。PUT 语句的文法如下：

```
EXEC IAF PUT field_name1,field_name2,……
    VALUES( :host_variable1, :host_variable2,……);
```

其中 field_name 和 host_variable 的含义同 GET 语句。

下面的语句把一个数值常量、串常量和宿主变量的值放入一个 Form 的字段内：

```
EXEC IAF PUT employee.number,employee.name,employee.job
    VALUES(7934,'MILLER',:new_job);
```

类似 GET，PUT 语句也允许使用一个串型宿主变量来指定块和字段名。例如：

strcpy(blkfld,"employee.name");

EXEC IAF PUT:blkfld VALUES(:new_name);

在字符方式的终端上,当用户出口返回(而不是赋值)时,显示放入字段中的值。在块方式终端上,当从设备上读入下一个字段时,显示读取的值。

如果用户出口几次改变一个字段的值,则仅最后一次改变有效。

另外,还能在用户出口中使用 WHENEVER 语句来发现无效数据类型转换(SQLERROR)、放入 Form 字段中的截短值(SQLWARNING)、和未有行返回的查询(NOT FOUND)。

§6.3 用户出口的引用

6.3.1 用户出口的引用方法

在 Form 中引用用户出口的方法是:定义一个触发器,在该触发器中,用一个命名为 USER_EXIT 的封装过程来引用一个用户出口。引用该封装过程的语法格式如下:

USER_EXIT(user_exit_string[,error_string]);

其中 user_exit_string 包含用户出口名和一些可选参数,error_string 包含用户出口失败时 SQL * Forms 所发出的错误信息。例如,下面的触发器命令调用一个命名为 LOOKUP 的用户出口:

USER_EXIT('LOOKUP');

注意:用户出口串用单引号(')括住,而不是双引号(")。

6.3.2 向用户出口传递参数

当调用一个用户出口时,SQL * Forms 把下列参数自动传递给它:

- Command line:是用户出口串
- Command line Length:是用户出口串的长度(以字符为单位)
- Error Message:是错误串(失败信息),如果它被定义的话
- Error Message Length:是错误串的长度
- In_Query:是一个布尔值,它指出该出口是以正常方式或是查询方式被调用。

但是,用户出口串允许在触发器命令中把更多的参数传递给它。例如,下面的触发器命令把 2 个参数(2025 和 A)和一个错误信息(LOOKUP faield)传递给用户出口 LOOKUP:

USER_EXIT('LOOKUP 2025 A','LOOKUP faield');

能用这个性能把字段名传递给用户出口,如下例所示:

USER_EXIT('CONCAT firstname,lastname, address');其中 firstname,lastname 和 address 是字段名。

但是,SQL * Forms 并不负责分析这些参数,用户出口要使用这些参数,必须自行负责分析识别它们。

6.3.3 把值返回给 Form

当 SQL * Forms 调用用户出口时,控制从 SQL * Forms 转向用户出口;当用户出口执

行完成时，控制又返回到 SQL * Forms。在把控制返回给 SQL * Forms 时，还返回一个状态码，以指出它是成功、失败或是遇到一个致命错误。状态码是 SQL * Forms 定义的整型常量。有三个状态码，其含义如下：

(1)IAPSUCC：用户出口成功结束。此时 SQL * Forms 继续执行成功的标号或下一步，除非调用触发器步设置反向返回代码开关。

(2)IAPFAIL：用户出口迂到一个错误，例如字段中有一个无效的值。此时，用户出口所传递的信息(是可选)出现在 SQL * Forms 屏幕底部的信息行上和显示错误的屏幕上。

(3)IAPFTL：用户出口迂到一个致命的错误，例如 SQL 语句的执行发生错误。此时，用户出口被迫中断，返回的错误信息出现在 SQL * Forms 显示错误的屏幕上。

如果一个用户出口改变了一个字段的值后，返回了一个失败或致命的错误码，则 SQL * Forms 并不废除这个改变。当反向返回码开关被设置和一个成功码被返回时，SQL * Forms 也不废除这个改变。

注意：对于其它宿主语言，状态码可能不是以 IAP 打头，而是以 SQL 打头。

用户出口能通过调用函数 SQLIEM 来指定一个错误信息。在控制返回 SQL * Forms 时，把该错误信息显示在信息行上或显示错误的屏幕上。指定的信息替换为该步定义的任何信息。SQLIEM 函数的调用格式如下：

SQLIEM(error_message，message_length)；

其中 error_message 是字符型变量，message_length 是整型变量。ORACLE 预编译程序生成相应的外部函数说明。在引用时，通过地址传递方式(不是值方式)来传递这两个参数。

§6.4 用户出口举例

下面是一个用户出口的例子，在 SQL * Forms 的触发器中用如下格式的命令行来调用用户出口：

user_exit ('CONCAT field1，field2，……，resuet_field')；其中 user_exit 是由 SQL * Forms 提供的一个封装过程，CONCAT 是用户出口名。

例 6.1 用户出口

```
/****************************************
 *    用户出口名：    CONCAT
 *    出口的调用格式：concat(cmd,cmdlen, msg, msglen, query)
 ****************************************/
#define min(a,b) ((a<b)? a:b)
#include <stdio.h>
/* 说明 SQLCA */
EXEC SQL INCLUDE sqlca;
/* 说明所有的宿主变量 */
EXEC SQL DEGIN DECLARE SECTION;
    VARCHAR   field[81];
    VARCHAR   value[81];
    VARCHAR   result[241];
```

```
EXEC SQL END BECLARE SECTION;
int concat(cmd,cmdlen, msg, msglen, query) /* 出口函数说明 */
char *cmd;/*触发器步中的命令行 ("CONCAT...")*/
int *cmdlen; /*命令行长度 */
char *msg;/*从 Form 中传来的触发器步错误信息 */
int *msglen; /*错误信息长度 */
int *query;/*如果由后台触发器查询,则为 TRUE,否则为
        FALSE */
    {char *cp=cmd+7; /*指向命令行串中字段名表的指针,
                    对"CONCAT"需要 7 个字节 */
    char *fp=(char *)& field.arr[0]; /*指向命令行串中字段名的指针 */
    char errmsg[81]; /*返回给 SQL*Forms
                        的错误信息 */
    int errlen;/* 返回给 SQL*Forms
                    的错误信息 长度*/
    /* 错误处理说明 */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    result.arr[0]='\0';
    /* 分析命令行串的字段名 */
    for (; *cp != '\0'; cp++)
    {
        if (*cp != ',')
                    /* 将字段名从命令行拷贝到 field.arr */
                    *fp++ = *cp;
        else
                    {/*现在已有全部的字段名 */
                     *fp='\0';
                     field.len=strlen(field.arr);
                     /*从 Form 中取字段值. */
                     EXEC IAF GET :field INTO :value;
                     value.arr[value.len]='\0';
                     strcat(result.arr,value.arr);
                     fp=(char *)& field.arr[0];/*再设置字段指针 */
        }
    }
    /*现在已有最后一个字段名*/
    field.len=strlen(field.arr);
    result.len=strlen(result.arr);
    /*把结果放置在 Form. */
```

```
    EXEC IAF PUT :field VALUES(:result);
    return(IAPSUCC);/* 触发器步成功 */
/* 错误处理 */
sqlerror:
    strcpy(errmsg, "CONCAT: ");
    strncat(errmsg,sqlca.sqlerrm,sqlerrmc,min(72,
        sqlca.sqlerrm.sqlerrm1));
    errlen=strlen(errmsg);
    /* 把错误信息传递给 SQL*Forms 的状态行 */
    sqliem(errmsg,&errlen);
    return(IAPFAIL); /* 触发器步失败 */
}
```

§6.5 用户出口的编译和连接

6.5.1 用户出口的预编译和编译

用户出口写好后，首先应当对它进行预编译和编译。预编译和编译的方法类似一般的PRO*C程序。请参考第7章运行ORACLE预编译程序。

6.5.2 使用GENXTB实用程序来在IAPXTB上加一项

模块IAPXIT中的IAP程序表IAPXTB包含连接到IAP中的每一个用户出口的项。IAPXTB告诉IAP每一个用户出口的名字、存储地址和宿主语言。当把一个新的用户出口加到IAP中时，必须把相应的项加到IAPXTB上。

IAPXTB是由数据库表(该表也命名为IAPXTB)产生的，能通过在操作系统的命令行上运行GENXTB Form来修改该数据库表。其格式如下：

RUNFORM GENXTB username/password

在发了此命令之后，显示该Form，此时就可输入所定义的每一个用户出口的如下信息：

- 出口名
- 宿主语言代码(C、COB、FOR、PAS、或PLI)
- 建立日期
- 最后修改日期
- 注释

在修改IAPXTB数据库表之后，用GENXTB实用程序来读该表和建立一个汇编(或C)源程序，该源程序定义模块IAPXIT和它包含的IAPXTB程序表。所用的源语言取决于操作系统。运行GENXTB实用程序的文法如下：

GENXTB username/password outfile

其中outfile给出由GENXTB建立的汇编程序或C源程序的名字

6.5.3 把用户出口连接到 SQL * Forms 中

在运行调用用户出口的一个 Form 之前，必须把用户出口连接到 IAP 和运行该 Form 的 SQL * Forms 成员上。

为了产生一个新的可执行的 IAP 拷贝，就要把用户出口的目标模块、标准 IAP 模块、IAPXIT 模块、和所需要的 ORACLE 库及宿主语言(如 C)连接库模块连接起来。

连接的细节依赖于所用的系统，请参考有关资料。

§6.6 开发用户出口的注意事项

在开发用户出口的过程中，应注意如下几点：

1. 出口名

用户出口名不能是 ORACLE 保留字。也要避免使用与 SQL * Forms 命令、函数代码和 SQL * Forms 所用的外部定义名相矛盾的名字。

SQL * Forms 在检索用户出口之前先把该出口名转换为大写。于是，如果所用的宿主语言对大小写敏感的话，则源代码中的用户出口名必须大写。

源码中用户出口的入口点名变为用户出口名本身。出口名必须是宿主语言和操作系统的一个有效文件名。

2. 与 ORACLE 的连接

用户出口是借助于 SQL * Forms 所产生的连接与 ORACLE 进行通讯。但是，用户出口也能通过 SQL * Net，与任何一个数据库建立辅助连接。

3. 使用宿主变量

在一般 PRO * C 程序中使用宿主变量的限制也适用于用户出口。宿主变量必须在用户出口的说明段命名，并在 EXEC SQL 和 EXEC IAF 语句中以冒号(:)开头。但是，不允许在 EXEC IAF 语句中使用宿主数组。

4. 更新表

一般来说，用户出口不应更新与 Form 相联系的数据库表。例如，假设一个操作员在 SQL * Forms 中更新一个与 Form 相联系的数据库表中的记录时，用户出口也更新该数据库表中的对应行。那么，当事务被提交时，由于用户出口变更的提交早于 Form 中变更的提交，于是，在 SQL * Forms 中更新的记录就会复盖用户出口中所作的相应更新。

5. 发命令

要避免由用户出口发 COMMIT 或 ROLLBACK 命令，应从 SQL * Forms 触发器中发 COMMIT 或 ROLLBACK。因为 ORACLE 提交或回滚由 SQL * Forms 操作符开始的操作，而不提交或回滚由用户出口所做的操作。数据定义命令(象 ALTER、CREATE 和 GRANT)的用法同 COMMIT 或 ROLLBACK，因为它们在执行前后都发一个隐含的 COMMIT。

第七章　运行 PRO＊C 预编译程序

§7.1　PRO＊C 预编译程序的操作命令及可选项

7.1.1　预编译程序的的操作命令

编译 PRO＊C 源程序所用的命令格式如下：

PROC INAME＝filename [OPtion_name1＝vatue1 OPtion_name2＝vatue2…]

其中 PROC 是预编译命令名；INAME＝filename 是预编译程序的输入文件，它指出 PRO＊C 源程序文件的路径及文件名，这是必须的。预编译程序输入该文件，进行预编译。PC 是标准输入文件的扩展名；如果输入文件的扩展名是标准名，则可缺省不写；如果不是标准的，则必须写。

option_namei(其中 i＝1,2,…)是预编译程序的可选项。在命令行中，命令名、输入文件及可选项相互之间用空格隔开。在指定可选项时，等号(＝)的左右两侧不能有空格。

预编译程序输入 PRO＊C 源程序，经预编译后产生 C 语言的代码序列(即 C 语言的源程序)，其中源文件中的嵌入 SQL 语句被翻译为相应的 ORACLE 库函数调用。在预编译过程中，如果发现语法错误，则发出错误信息。

例如，下面的命令行

PROC INAME＝infile.pc ONAME＝outfile.c 执行后，PRO＊C 源文件 infile.pc 被预编译生成输出文件 outfile.c。outfile.c 是 C 语言的代码序列，它又是 C 编译程序的输入文件。

7.1.2　预编译程序的可选项

1. 可选项清单

预编译程序提供了许多有用的可选项。这些可选项被用来控制如何使用资源、怎样格式化输入和输出、如何报告错误、以及如何管理光标等。表 7－1 列出了 PRO＊C 预编译程序提供的全部可选项。

表 7-1　PRO＊C 预编译程序的可选项

文　法	缺省值	说　明
ASACC＝YES|NO	NO	
CODE＝ANSI_C|KR_C	KR_C	如何生成 C 函数原型
DBMS＝NATIVE|V6|V7	NATIVE	
DEFINE＝symbol＊		定义条件预编译中的符号
ERRORS＝YES|NO	YES	错误是否被发送给终端
FIPS＝YES|NO	NO	是否标志 ANSI|ISO 扩展

文　法	缺省值	说　明
HOLD_CURSOR=YES|NO *	NO	高速缓冲存储器如何处理 SQL 语句
HOST=C|COBOL| COB741|FORTRAN| PASCAL|PL/I		输入文件的宿主语言
INAME=path and filename		输入文件名
INCLNDE=path *		INCLUDE 文件的目录路径
IRECLEN=inteher	80	记录长
LINES=YES|NO	NO	是否产生 C 的 #line 命令
LNAME=path and filename		清单文件名
LRECLEN=integer	132	清单文件记录长
LTYPE=LONG| SHORT|NONE	LONG	清单类型
NAXLITERAL=integer *		串的最大长度
MAXOPENCURSORS =integer *	10	光标高速缓冲存储器的最大数
MODE=ANSI|ISO| ANSO14|ANSI13| ISO13|ORACLE	ORA— CLE	与 ANSI|ISO 标准的一致
ONAME=path and filename		输出文件名
ORACA=YES|NO *	NO	是否使用 ORACA
ORACLEN=integer	80	输出文件记录长
PAGELEN=integer	66	清单中每页行数
RELEASE_CURSOR=	NO	光标高速缓冲器如何处理
YES|NO *		SQL 语句
SELECT_ERROR= YES|NO *	YES	如何处理 SELECT 错误
SQLCHECK=SENANTICS| FULL|SYNTAX| LIMITED|NONE *	SYNTAX	语法和语义检查的范围
USERID=username/password		有效的 ORACLE 用户名和口令
REF=YES|NO *	YES	清单中的义叉引用段

注：标有 * 的是可在程序行内输入的可选项。

2. 可选项的描述

为了便于参考，下面按字母顺序描述每一个预编译可选项。

(1)ASACC

功能：指出清单文件是否为了回车控制而遵循使用每一行第一列的 ASA 约定。

语法：ASACC=YES|NO

缺省值：NO

注意：只能在命令行上输入

(2)CODE

功能:指出 PRO * C 预编译程序所生成的 C 函数原型的格式(一个函数原型说明该函数及它的参量的数据类型)。CODE 可选项用于控制预编译程序所生成的 SQL 库函数的函数原型。

当 CODE=ANSI_C 时,PRO * C 预编译程序生成适应 ANSI C 标准的函数原型,

如:

```
extern void sqlora(long * ,void * )
```

当 CODE=KR_c 时,预编译程序生成与 Kernighan 和 Ritchie 所著 的《C 程序设计语言》(第一版)一致的代码。这时预编译程序注释出所生成的库函数的函数原型参数表。如:

```
extern void sqlora(/ * _long * , void * _ * /);
```

当 C 编译程序与 ANSI 不兼容时,应指定 CODE=KR_C。

语法 :CODE=ANSI_C|KR_C

缺省值:KR_C

注意:能在命令行或程序行上输入。

(3)DBMS

功能:指定所用的 ORACLE 数据库管理系统的版本,是 6 版、7 版或本国版。

语法:DBMS=NATIVE|V6|V7。

缺省值:NATIVE。

注意:只能在命令行上输入。

(4)DEFINE

功能:用于定义一个符号,以便进行条件预编译。

语法:DEFINE=symbol

缺省值:无

注意:能在命令行和程序行上输入。如果在程序行上输入时,用如下格式的语句:

```
EXEC ORACLE DEFINE symbol;
```

(5)ERRORS

功能:指出预编译程序的错误信息是否被发送到终端和清单文件上、或仅仅给清单文件。

当 ERRORS=YES 时,错误信息被发送给终端和清单文件。

当 ERRORS=NO 时,仅发送给清单文件。

语法:ERRORS=YES|NO

缺省值:YES

注意:能在命令行或程序行上输入。

(6)FIPS

功能:指出是否用 FIPS 标志符来标志 ANSI SQL 的扩展部分,除了特权强制规则外,任何一个扩展的 SQL 元素都是违背 ANSI 格式和语法规则的。

语法:FIPS=YES|NO

缺省值:NO

注意:能在命令行或程序行上输入。

当 FIPS=YES 时,如果使用一个 ANSI SQL 的 ORACLE 扩展,或以不一致的方式使用一个 ANSI SQL 性能,就发出警告(不是错误)信息。在预编译时,如果有下列ANSI SQL 扩展时,应指定 FIPS=YES 来标志它:

- 包括 FOR 子句的数组边界。
- SQLCA、ORACA 和 SQLDA 数据结构。
- 包括 DESCRIBE 语句的动态 SQL。
- 嵌入 PL/SQL 块。
- 自动数据类型转换。
- DATE、NUMBER、RAW、LONGRAW、VARRAW、ROWID、和 VARCHAR 数据类型。
- 指针宿主变量
- 指定运行可选项的 ORACLE OPTION 语句
- 用户出口中的 IAF 语句。
- CONNECT 语句。
- TYPE 和 VAR 数据类型等价语句。
- AT <db_name>子句。
- DECLARE…DATABASE、…STATEMENT 和…TABLE 语句。
- WHENEVER 语句中的 SQL WARNING 条件。
- WHENEVER 语句中的 DO、DO BREAK 和 STOP 动作。
- COMMIT 语句中的 COMMENT 和 FORCE TRANSACTION 子句。
- ROLLBACK 语句中的 FORCE TRANSACTION 和 TO SAVEPOINT 子句。
- COMMIT 和 ROLLBACK 语句中的 RELEASE 参数 。
- WHENEVER…GOTO 标号和 INTO 子句中的宿主变量的可选冒号前缀。

(7)HOLD_CURSOR

功能:该可选项用来控制光标和光标高速缓冲存储器之间的链。它指出 SQL 语句和 PL/SQL 块的光标如何使用。能用它来改进程序性能。

当执行 SQL 数据操纵语句时,其相关的光标被连到光标高速缓冲存储器中的一项上,该项又被依次连接到 ORACLE 专用的 SQL 区域上,该区域存储处理该语句所需的信息。

当 HOLD_CURSOR=NO 时,在 ORACLE 执行完 SQL 语句或关闭光标后,预编译程序直接撤去该链,释放分析块和分配给专用 SQL 区域的内存,并把该链标为可再使用。这时另一个 SQL 语句就又可用该链来指向光标高速缓冲存储器的项了。

当 HOLD_CURSOR=YES 时,该链被保留;预编译程序不能再使用它。这对于经常执行的 SQL 语句是有用的,因为它加速后续的执行,使其不需再分析语句或为

ORACLE 专用区的 SQL 区分配内存。

语法:HOLD_CURSOR=YES|NO

缺省值:NO

注意:能在命令行或程序行上输入。当在程序行上输入时,若与隐式光标一起使用的话,在执行 SQL 语句之前设置 HOLD_CURSOR;若与显式光标一起使用的话,应在打开光标前设置 HOLD_CURSOR。

另外还需注意:

RELEASE_CURSOR=YES 优先于 HOLD_CURSOR=YES;

HOLD_CURSOR=NO 优先于 RELEASE_CURSOR=NO。

(8)HOST

功能:指出输入文件的宿主语言

语法:HOST=C|COB74|COBOL|FORTRAN|PASCAL|PLI

缺省值:依赖于语言

注意:只能在命令行上输入。

如果使用非标准输入文件扩展名时,当指定 INAME 的值时,必须指定 HOST。

(9)INAME

功能:指定输入文件名

语法:INAME=Path and filename

缺省值:无

注意:只能在命令行上输入。如果指定 INAME 时,输入文件使用非标准扩展名,则必须指定 HOST。

(10)INCLUDE

功能:指定 EXEC SQL INCLUDE 文件的目录路径,它只适用于使 用目录的操作系统。

语法:INCLUDE=path

缺省值:当前目录

注意:能在命令行或程序行上输入。

(11)IRECLEN

功能:指定输入文件的记录长度。

语法:IRECLEN=integer

缺省值:80

注意:只能在命令行上输入。指定的值不应超过 ORECLEN 的值。

(12)LINES

功能:指出预编译程序是否要对其输出文件加#line 命令。

语法:LINES=YES|NO

缺省值:NO

注意:只能在命令行上输入。

当 LINES=YES 时,PRO * C 预编译程序对输出文件加 C 预编译程序命令#line。

当 LINES=NO 时,预编译程序不对其输出文件加#line 命令。

(13)LNAME

功能:为清单文件指定非缺省名

语法:LNAME=path and filename

缺省值:input.LIS,其中 input 是该输入文件的基本名。

注意:只能在命令行上输入。根据缺省规定,清单文件被写到当前目录上。

(14)LRECLEN

功能:指定清单文件的记录长度

语法:LRECLEN=integer

缺省值:132

注意:只能在命令行上输入。

LRECLEN 值的范围是 80 至 255。如果指定的值<80,则使用 80。如果指定的值>255,则使用 255。LRECLEN 应该至少比 IRECLEN 大 8,以考虑扦入行号。

(15)LTYPE

功能:指定清单类型

语法:LTYPE=LONG|SHORT|NONE

缺省值:LONG

注意:只能在命令行上输入。

当 LTYPE=LONG 时,则清单文件中包含输入行。

当 LTYPE=SHORT 时,则清单文件中不包含输入行。

当 LTYPE=NONE 时,不建立清单文件。

(16)MAXLITERAL

功能:指出预编译程序生成的串文字的最大长度,例如,如果编译程序不能处理超过 256 个字符的串文字的话,应指定 MAXLITERAL=256。

语法:MAXLITERAL=integer

缺省值:依赖于语言,对 C 语言为 1000

注意:能在程序行内和命令行上输入。在程序行上输入该可选项的方法如下:

```
EXEC ORACLE OPTION(MAXLITERAL=integer);
```

该语句必须编写在第一个 EXEC SQL 语句之前,而且对该可选项,程序只能设置一次。否则,预编译程序将发出警告信息,并忽略多加的或错放置的 EXEC ORACLE 语句。

(17)MAXOPENCURSORS

功能:指定同时打开的光标数。

语法:MAXOPENCURSORS=integer

缺省值:10

注意:能在程序行和命令行上输入。

使用该可选项能改进程序性能。

每个用户处理能打开的光标的最大数由 ORACLE 初始化参数 OPEN_CURSORS 设置,其范围为 5 至 255。使用该可选项可重新指定一个较低(而不是较高)的值,以替代初始化设置。MAXOPENCURSORS 的设置必须比初始设置

OPEN_CURSORS 至少低 6。

(18)MODE

功能:指定程序是遵守 ORACLE 的标准或是遵守 ANSI 标准。

语法:MODE=ANSI|ISO|ANSI14|ISO14|ANSI13|ISO13|ORACLE

缺省值:ORACLE

注意:只能在命令行上输入。下述几对值是等价的:

ANSI 和 ISO、ANSI14 和 ISO14、ANSI13 和 ISO13。

当 MODE=ORACLE 时,嵌入式 SQL 程序遵守 ORACLE 标准。

当 MODE={ANSI14|ANSI13}时,程序严格遵守 ANSI 标准。

当MODE=ANSI 时,程序完全遵守 ANSI 标准,而且下列变更有效:

- COMMIT 或 ROLLBACK 语句关闭所有的显式光标。而当 MODE={ANSI13|ORACLE}时仅关闭 CURRENT OF 子句中引用的光标。
- 不能打开已打开的光标,或关闭已关闭的光标。(而当 MODE=ORACLE 时,能再打开一个已打开的光标,以避免重新分析。)
- 不能把 Null 值 SELECT 或 FETCH 到没有相关指标器变量的宿主变量中。(当 MODE=ANSI13|ORACLE 时,不需要提供指标器变量。)
- 必须在说明段内或外说明一个 4 字节长的整型变量 SQLCODE
- 说明 SQLCA 是可选的,即不需要把 SQLCA 编写或拷贝到程序中。(当 MODE={ANSI13|ORACLE}时,则要求说明 SQLCA。)
- 如果 ORACLE 把一个截短列值赋给输出宿主变量时,不出现错误信息。
- 当发生"no data found"错误时,返回代码+100 给 SQLCODE(而不是+1403)。错误信息文本不变。
- 在 SQL 数据操纵语句中,每一个宿主变量必须用冒号(:)前缀。(当 MODE={ANSI13|ORACLE}时,在 SELECT 或 FETCH 语句的 INTO 子句中,冒号前缀是可选的。)
- CHAR 列值、USER 伪列值、字符宿主值、和文字串类似 ANSI 的定长字符串来处理。还有,当你赋值、比较、INSERT、UPDATE、SELECT、FETCH 这类值时,需使用空格填充语义。(当 MODE={ANSI13| ORACLE}时,这些值的处理类似 VARCHAR2 变长字符串,且不用空格填充语义。上述这些变更对不同版本的 PRO*C 预编译程序来说,有些可能是无效的,表 7-2 表示它们的有效性。

表 7-2 不同版本的 PRO*C 预编译程序的变更

变　更	V1.3	V1.4	V1.5
以类似于 ANSI 串的方法处理 CHAR 值	no	no	YES
允许数组操作	YES	no	YES
COMMIT,ROLLBACK 关闭显式光标	no	YES	YES
打开一个已打开的光标为非法	YES	YES	YES
当提取 Null 值时必须用指示器变量	no	YES	YES
必须说明 SQLCODE	no	YES	YES

变　更	V1.3	V1.4	V1.5
如果输出值被截短时没有错误信息	YES	YES	YES
"no dath faund"错误码是+100	YES	YES	YES
说明 SQLCA 是可选的	no	YES	YES
在 INTO 子句中要求冒号前缀	no	YES	YES

(19)ONAME

功能:指定输出文件名

语法:ONAME=path and filename

缺省值:输出文件被写到当前目录

注意:只能在命令行上输入

(20)ORACA

功能:指出程序是否能用 ORACLE 通讯区

语法:ORACA=YES|NO

缺省值:NO

注意:能在程序行或命令行上输入。

当 ORACA=YES 时,必须说明 ORACA。

(21)ORECLEN

功能:指定输出文件的记录长度

语法:ORECLEN=integer

缺省值:80

注意:只能在命令行上输入。指定的值应≥IRECLEN 的值。

(22) PAGELEN

功能:指定清单文件的每一物理页的行数。

语法:PAGELEN=integer

缺省值:66

注意:只能在命令行上输入。

(23)RELEASE_CURSOR

功能:该可选项用来控制光标和光标高速缓冲存储器之间的链。它指出 SQL 语句和 PL/SQL 块的光标如何使用。能用它来改进程序性能。

当执行 SQL 数据操纵语句时,其相关的光标被连到光标高速缓冲存储器中的一个项上,该项又被依次连接到 ORACLE 专用的 SQL 区域上,该区域存储处理该语句所需的信息。

当 RELEASE_CURSOR=YES 时,在 ORACLE 执行完 SQL 语句或关闭光标后,预编译程序直接撤去该链,释放分析块和分配给专用 SQL 区域的内存,并把该链标为可再使用。这时另一个 SQL 语句就又可用该链来指向光标高速缓冲存储器的项了。为了保证在关闭光标时使有关的资源被释放,必须指定

RELEASE_URSOR=YES。

当 RELEASE_CURSOR=NO 和 HOLD_CURSOR=YES 时,该链被保留。除非

打开的光标数超出 MAXOPENCURSORS 的值。这对于经常执行的 SQL 语句是有用的，因为它加速后续的执行，使其不需再分析语句或为 ORACLE 专用的 SQL 区分配内存。

语法：RELEASE_CURSOR＝YES|NO

缺省值：NO

注意：能在程序行或命令行上输入。当在程序行上输入时，若与隐式光标一起使用的话，应在执行 SQL 语句之前设置 RELEASE_CURSOR；若与显式光标 一起使用的话，应在关闭光标前设置 RELEASE_CURSOR。

另外需注意：

RELEASE_CURSOR＝YES 优先于 HOLD_CURSOR＝YES；

HOLD_CURSOR＝NO 优先于 RELEASE_CURSOR＝NO。

(24) SELECT_ERROR

功能：当 SELECT_ERROR＝YES 时，若单行 SELECT 语句返回多于一行，或多行 SELECT 语句返回的行数比宿主数组能容纳的还要多，则产生错误，且查询的结果是不确定的。当指定 SELECT_ERROR＝NO 时，则不产生错误。

语法：SELECT_ERROR＝YES|NO

缺省值：YES

注意：能在程序行或命令行上输入。

(25)SQLCHECK

功能 ：指定语法和语义检查的类型和范围。使用该可选项可使预编译程序通过检查嵌入 SQL 语句或 PL/SQL 的语法和语义来帮助调试程序。

语法：SQLCHECK＝SEMANTICS|FULL|SYNTAX|LIMITED|NONE

缺省值：SYNTAX

注意：能在程序行或命令行上输入，但在程序行指定的检查级别不能高于在命令行上指定的检查级别。例如，如果在命令行上指定了 SQLCHECK＝{SYATAX|LIMITED}那么就不能在程序行上指定 SQLCHECK＝{SEMANTICS|FULL}

当 SQLCHECK＝{SEMANTICS|FULL}时，预编译程序检查数据操纵语句(如 INSERT、UPDATE 等)、PL/SQL 块以及宿主变量的数据类型的语法和语义。对数据定义语句(如 CREATE、ALTER 等)做语法检查。但是，对具有 AT db_name 子句的数据操纵语句仅做语法检查。

当 SQLCHECK＝{SEMANTICS|FULL}时，预编译程序通过使用嵌入 DECLARE TABLE 语句或在命令行上使用 USERID 可选项与 ORACLE 相连接和访问数据词典来取得语义检查所需的信息。

如果在数据操纵语句或 PL/SQL 块中所引用的每一个表都在 DECLARE TABLE 语句中定义，则不需与 ORACLE 相连接。

如果与 ORACLE 相连接，而某些所需的信息在数据词典中又找不到，就必须用 DECLARE TABLE 语句提供丢失的信息。在 DECLARE TABLE 中定义的信息优先于在数据词典中定义的信息(如果它们冲突时)。

如果在程序中嵌入了 PL/SQL 块，就必须指定 SQLCHECK＝{SEMANTICS|

FULL}当 SQLCHECK＝{SYNTAX|LIMITED}时，预编译程序检查数据操纵语句、数据定义语句、宿主变量数据类型等的语法，而不做语义检查。也不考虑 DECLARE TABLE 语句，不许嵌入 PL/SQL 块。

请注意：值 LIMITED 和 SYNTAX 是等价的。

仅在下列情况下才指定 SQLCHECK＝NONE：

- 不需要做严格的数据类型检查
- 程序引用了仍未建立的表和漏掉了这些表的 DECLARE TABLE 语句。

(26)USERID

目的：指定 ORACLE 用户名和口令。

语法：USERID＝username/password

缺省值：无

注意：只能在命令行上输入

当使用自动登录特性时，不指定该可选项。当 SQLCHECK＝SEMANTICS 时，如果想使预编译程序通过与 ORACLE 相连和存取数据词典来取得所需的信息时，也必须指定 USERID。

(27)XREF

目的：指出是否在清单文件中列交叉索引表。

语法：XREF＝YES|NO

缺省值：YES

注意：该可选项可在程序行或命令行上输入。

当指定 XREF＝YES 时，在清单文件中将列出宿主变量、光标名和语句名的交叉索引表。

交叉索引表示每一个对象在程序中的什么地方定义和引用。

当 XREF＝NO 时，不列出交叉索引表。

3. 可选项的输入方法

所有的预编译可选项都能在命令行上输入，其中一部分可选项(表中标＊的)还能在程序行上输入。在命令行上输入的方法如前所述。在程序行上输入的方法是用 EXEC ORACLE 语句，其格式如下：

EXEC ORACLE OPTION(option_name＝value)；例如，可用如下语句指定 RELEASE_CURSOR 可选项：

EXEC ORACLE OPTION(RELEASE_CURSOR＝YES)；在程序行上输入的可选项优先于命令行上输入的同一个可选项。可在如下几种情况下在程序行上输入的可选项：

- 在预编译过程中改变可选项的值：如果在命令行或程序行上指定了一个可选项(如 HOLD_CURSOR＝YES)，在编译期间，希望在程序的某位置之后改变该可选项的值(如改为 HOLD_CURSOR＝NO)，则可在该位置放置一个如下的语：

EXEC ORACLE OPTION(HOLD_CURSOR＝NO)；

- 如果需指定的可选项较多，而操作系统又限定命令行上的字符个数时，可用 EXEC ORACLE 语句在程序行上指定命令行上尚未指定的可选项。甚至可以把程序行可选项作为文件存储到机器上，然后把它 INCLUDE 到程序的相应位置上。

4. 可选项的作用范围

对给定的预编译单位指定的可选项只影响该预编译单位，而不影响别的预编译单位。例如，如果对编译单位 A 指定了可选项 HOLD_CURSOR 和 RELEASE_CURSOR，而未有对单位 B 指定，则单位 A 中的 SQL 语句用指定的 HOLD_CURSOR 和 RELEASE_CURSOR 值运行，而单位 B 中的 SQL 语句用其缺省值运行。

当在程序行上输入可选项时，因为 EXEC ORACLE 语句的作用范围是位置的，而不是逻辑的。所以，一个 EXEC ORACLE 语句在没有用另一个 EXEC ORACLE 语句指定相同的可选项之前一直保持有效。在下例中，在未用 HOLD_CURSOR＝YES 替代之前，HOLD_CURSOR＝NO 一直保持有效：

```
EXEC SQL BEGIN DECLARE SECTION;
    char emp_name[20];
    int emp_number;
    ftoat salary;
    int dept_number;
  EXEC SQL END DECLARE SECTION
  ...
  EXEC SQL WHENEVER NOT FDUND GOTO no_more;
  EXEC ORACLE OPTION(HOLE_CURSOR=NO);
  EXEC SQL DECALRE c CURSOR FOR
       SELECT EMPNO,ENAME,SAL
            FROM EMP
            WHERE DEPTNO=:dept_number;
  printf("\n Enter Department Number:");
  scanf("%d",dept_number);
  EXEC SQL OPEN c ;
  printf("\n Number      Name          Salary");
  printf("\n ..........");
  while(1)
  {
     EXEC SQL FETCH c INTO :emp_number,:emp_name,:salary;
     printf("\n %d %s%f", emp_number,emp_name,salary);
  }
no_more:
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  while(1)
  {
     printf("\n Employee Number?");
     sanf("%d", &emp_number);
     if (emp_number==0)
```

```
        break;
    EXEC ORACLE OPTION(HOLD_CURSOR=YES);
    EXEC SQL SELECT ENAME,SAL
            INTO:emp_name,:salary
            FROM EMP
            WHERE EMPNO=:emp_number;
printf("\n salary for %s Is %d", emp_name,salary);
}
...
```

程序行上输入的可选项优先于命令行上输入的同一个可选项。

§7.2 条件预编译和分别预编译

7.2.1 条件预编译

条件预编译是根据指定的条件成立与否来决定 PRO*C 程序中的特定代码段是否被包括。例如,有这样一个 PRO*C 源程序,它包括 WINDOWS 系统环境的代码段和 MS-DOS 环境的代码段。如果要生成一个在 WINDOWS 系统环境下运行的应用程序时,在预编译时就要包括 WINDOWS 系统环境的代码段,而不包括 MS-DOS 环境的代码段。若要生成一个在 MS-DOS 环境下运行的应用程序时,在预编译时就要包括 MS-DOS 环境的代码段,而不包括 WINDOWS 系统环境的代码段。此时可使用条件预编译来实现。因此,可用条件预编译来编译在不同环境下运行的应用程序。这些被包括或排除的代码段叫条件代码段。通常条件代码段是由定义环境的语句和采取的动作所组成。可用下述语句来实现条件预编译:

```
EXEC ORACLE DEFINE symbol;   /* 定义一个符号 */
EXEC ORACLE IFDEF symbol;    /* 如果符号被定义 */
EXEC ORACLE IFNDEF symbol;   /* 如果符号未被定义 */
EXEC ORACLE ELSE;            /* 否则 */
EXEC ORACLE ENDIF;           /* 控制块结束 */
```

下面的例子说明进行条件预编译时程序代码的一般结构:

```
EXEC ORACLE DEFINE flag_code; /* 定义一个符号 */
.....
EXEC ORACLE IFDEF flag_code;  /* 如果符号被定义 */
```

代码段 1

```
EXEC ORACLE ELSE;             /* 否则 */
```

代码段 2

```
EXEC ORACLE ENDIF;            /* 控制块结束 */
```

要想使代码段 1 被预编译(即包括),只需定义 flag_code 即可;如果 flag_code 未被定义,则 代码段 2 被编译。

下面是对 SELECT 语句进行条件预编译的例子。当 site2 被定义时,该 SELECT 语句才被预编译:

```
EXEC ORACLE IFDEF site2;
        EXEC SQL SELECT DNAME
        INTO :dept_name
        FROM DEPT
        WHERE DEPTNO=:dept_number;
EXEC ORACLE ENDIF;
```

如果 site2 未被定义时,则该 SELECT 语句不被预编译(即被排除)。

条件块能嵌套,如下述结构:

```
EXEC ORACLE IFDEF outer;
    EXEC ORACLE IFDEF inner1;
        ...
    EXEC ORACLE ENDIF;
EXEC ORACLE ELSE;
        EXEC ORACLE IFNDEF inner2;
            ...
        EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

定义符号的方法有两种:

·在 PRO * C 程序中用 DEFINE 语句:

```
EXEC ORACLE DEFINE symbol;
```

·在命令行上用 DEFINE 可选项:

```
…INAME=filename…DEFINE=symbol
```

其中 symbol 大小写均可。

在 ORACLE 预编译程序被安装在系统上时,已为用户预先定义了一些特殊符号。预先定义的操作系统符号有 CMS、MVS、MSDOS、UNIX、和 VMS,硬件符号有 IBM 和 DEC。例如,在 VMS 系统上,预先定义了符号 DEC 和 VMS。用户在进行条件预编译时,可使用这些符号。

7.2.2 分别预编译

一个应用程序可由若干个源文件所组成。这时可先用 ORACLE 预编译程序分别预编译各个源文件,最后再把它们连接成一个可执行程序。

当对 PRO * C 程序实行分别预编译时,需注意如下几点:

·光标操作不能跨越预编译单位(文件)。也就是说,在一个文件中说明的光标,不能在另一个文件中打开或使用它(如 FETCH)。因此,在做分别预编译时,要确保对光标的定义和引用是在一个源文件中。

·如果在 PRO * C 程序的一个源文件中说明了一个全程的 SQLCA，则在所有其它源文件中只需把它说明为 extern 存储类即可。这可通过如下语句把 SQLCA 指派为 extern：

#define SQLCA_STORAGE_CLASS extern 如果不把 SQLCA 说明为外部的，则该源文件将使用其局部的 SQLCA。

·当预编译一个与 ORACLE 相连接的程序模块时，要指定一个对其它程序模块都足够大的 MAXOPENCURSORS 值。

§7.3 编译与连接

PRO * C 源程序经 PRO * C 预编译程序编译后，生成一个新的源文件(叫 C 源文件)。它是一个完全符合 C 文法要求的文件，其中已不再包括嵌入 SQL 语句和 PL/SQL 块等，这些成分已在预编译期间被转换为标准 ORACLE 运行库调用。

为了得到一个可执行的程序，在预编译后还应对生成的 C 源文件进行编译，然后再把编译所生成的目标模块与所需要的 ORACLE 运行库模块相连接。

连接程序解决目标模块中的符号引用。如果这些引用有冲突，则连接失败。

第二篇　ORACLE 调用接口

第一章　ORACLE 调用接口概述

§1.1　ORACLE 调用接口的有关概念

1.1.1　什么是 ORACLE 调用接口

ORACLE 调用接口(ORACLE CALL INTERFACE—简称 OCI)是一个应用程序的开发工具,它提供了一组应用程序设计的接口子例程(或函数)。利用这些接口子例程(或函数)可对 ORACLE 数据库中的数据和模式进行操纵。ORACLE 调用接口支持所有的 SQL 数据定义、数据操纵、查询和事务控制等。这样就可采用在第三代程序设计语言(如 C、COBOL 和 FORTRAN 等)中调用这些接口子例程(或函数)的方法来开发应用程序。

ORACLE 公司为三个最通用的高级语言(C、COBOL 和 FORTRAN)提供了接口子例程(或函数)库(又叫 OCI 子例程或函数库)。

1.1.2　什么是 OCI 程序

OCI 程序是 ORACLE 调用接口程序的简称。所谓 OCI 程序实质上就是用第三代高级程序设计语言写的程序,其特点是内部含有对 OCI 子例程(或函数)库的调用。例如,如果宿主语言是 C 语言的话,则 OCI 程序是其内含有对 OCI 函数(关于 C 的 OCI 函数)调用的 C 程序。本篇重点讲述关于 C 语言的调用接口。通过本篇学习,你将掌握如何利用在 C 语言内调用 OCI 函数来开发应用程序的方法。

1.1.3　利用 OCI 开发应用程序的优点

OCI 开发方法实质上是结构查询语言和第三代程序设计语言相结合的一种开发方法。我们知道,结构查询语言(SQL)是一个非过程语言,用这种语言写程序时只需指出做什么和对什么数据进行操作,而不需要确切指出如何实现这些操作。因此,SQL 语言易于学习,便于实现数据库事务的处理。

另一方面,第三代程序设计语言(如 C、COBOL 和 FORTRAN 等)是过程语言,它们比 SQL 语言更复杂,但更灵活,数据处理的能力更强。

在使用 OCI 开发应用程序时,对数据库的访问是通过调用 OCI 库函数来实现的,即通过 ORACLE 库调用来实现。如果宿主语言是 C 语言的话,ORACLE 库调用其实就是 C 程序中的函数调用,不过这些函数一般都是与数据库的访问有关的。因此,可以说,一个含 OCI 调用的 C 程序就是一个用纯 C 语言的语句编写的程序。这样的程序不仅具有 SQL 非过程

性能的优点，也具有第三代程序设计语言的过程性能，同时还可具有 PL/SQL 的优点。使开发的应用具有更强的数据处理能力和更大灵活性。

1.1.4 常用的一些特殊术语

在进一步讨论本篇的内容之前，为了便于叙述，在此特对 OCI 程序设计时常用到的一些特殊术语加以说明。

一个 SQL 语句，例如

```
SELECT empno,ename,sal
        FROM emp
        WHERE job='SOFTWARE'
              AND deptno=dept_number
```

包含如下几部分：

- SQL 命令：SELECT
- 选择表项：empno，ename 和 sal
- FROM 子句中的表名：emp
- WHERE 子句中的文字串输入值：'SOFTWARE'
- WHERE 子句的第二部分中的虚拟输入变量：dept_number

当开发 OCI 应用程序时，要调用一些例程，这些例程向 ORACLE7 Server 说明程序中的输入和输出变量的地址。在本书中，把说明虚拟输入变量的地址叫结合操作，说明输出变量的地址叫定义操作。

§1.2 OCI 程序的基本结构及举例

用 ORACLE 调用接口编写的应用程序一般应具有如下的结构：

(1) 分配数据结构，以便与 ORACLE 相连接和处理光标。

(2) 与一个或多个 ORACLE 数据库相连接。

(3) 当程序需要时，打开一个或多个光标。

(4) 处理 SQL 或 PL/SQL 语句，以完成对数据库的相应操作。

(5) 关闭光标(当使用时)。

(6) 切断与数据库的连接。

下面是一个用 C 语言写的 OCI 程序，读者从中可看出一个 OCI 程序的基本结构。

例 1.1 OCI 程序的基本结构

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *    说明：
 *        该例是返回多行的查询 。其处理步骤是：
 *            (a) 与 ORACLE 相连接
 *            (b) 打开光标
 *            (c) 处理 SQL 语句
 *                延迟分析 SELECT 语句
 *                输入部门号召
```

```
*                   延迟结合
*                   定义输出变量
*                   检索：执行 oexfet
*                   显示查询结果
*               (d) 关闭光标
*               (e) 结束处理,退出 ORACLE
******************************/
/* 定义 LDA ,CDA 和 HDA */
lda_def lda ;
cda_def cda;
ub1 hda[256];
/* 给出 SELECT 语句 */
text *sql_statement=
           "SELECT ename, sal FROM emp WHERE deptno=:1";
/* 定义数组 */
float salaries[12000];
text names[12000][20];
sword dept_number;

/* 程序体 */
main()
{
  int i, len;
  /* 与 ORACLE 相连接 */
  if (olon (&lda, "scott/tiger", -1, 0, -1, 0))
  {
        /* 连接失败 */
        printf ("cannot connect to ORACLE as scott/tiger\n");
        exit(1);
  }
  /* 打开光标 */
  if (oopen (&cda, &lda, 0, -1, -1, 0, -1))
  {
        /* 打开光标失败 */
        printf ("cannot open the cursor\n");
        ologof(&lda);
        exit(1);
  }
  /* 处理 SQL 语句 */
```

```
/* 延迟分析 */
oparse(&cda,sql_statement,-1,1,1)

/* 提示输入部门号 */
printf("Enter department number:");
scanf("%d,"&dept_number);

/* 延迟结合 */
obndrn(&cda,1,&dept_number,(int)sizeof(int),3,-1,
        0,0.-1,-1);
/* 定义输出变量 */
odefin(&cda,2,salaries,(int)sizeof(float),
        4,-1,sal_ind,0,-1,-1); /* FLOAT 的数据类型 4 */
odefin (&cda,1,names,20,1,-1,name_ind,0,-1,-1);
/*检索 12000 个人的工资*/
oexfet(&cda,12000,0,0); /* 不设置 cancel and exact */
/*显示查询结果 */
for(i=0;i<12000;i++)
{
  if(name_ind[i]==-1)
    names[i][0]='\0';
  else
    names[i][19]='\0';
  if(sal_ind[i]==-1)
    printf("\n%s",names[i]);
  else
    printf("\n%s\t%f",names[i],salaris[i]);
}
/* 关闭光标 */
if (oclose(&cda))
      exit(1);
/* 结束处理,退出 ORACLE */
if (ologof(&lda))
      exit(1);
exit(0);
```

§1.3 运行 OCI 程序的基本步骤

我们知道,OCI 程序是含有 OCI 函数调用的 C 程序,也就是说它实质上是一个 C 程序。因此,它的处理步骤应当同 C 类似,即包括如下三步:

(1)用 C 编译器进行编译

(2)用连接装配程序进行连接装配,生成一个可执行程序。

(3)执行该程序

图 1-1 表示运行 OCI 程序的基本步骤,它不需要预编译这一步。连接 OCI 程序的方法随系统的不同而异。有关编译和连接一个 OCI 应用程序的详细内容请看 ORACLE 安装和用户指南。

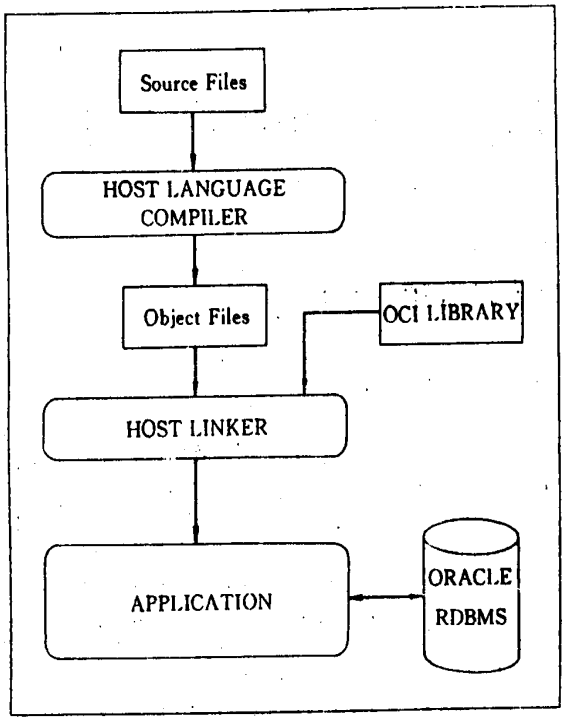


图 1-1 OCI 程序的处理步骤

§1.4 参考资料

(1) Programmer's Guide to the ORACLE Call Interfaces, part NO. 5411-70

(2) ORACLE Server SQL Language Reference Manual, Part NO. 778-70

(3) ORACLE 应用系统开发工具(孙宏昌,刘金亭,何毅华著)

第二章　OCI 程序设计的基础知识

本章主要介绍 OCI 程序设计的一些基础知识。

§2.1　OCI 程序中用到的数据结构

在 OCI 程序中要用到如下三个数据结构：

(1)登录数据区

(2)宿主数据区

(3)光标数据区

2.1.1　用户定义的数据类型名

为了便于描述，本篇采用的数据类型名不是系统提供的标准数据类型名，而是用户定义的数据类型名。下边列出这些定义，以便于读者阅读下文。

```
typedef int eword;              /* 有符号整数,不重要 */
typedef unsigned int uword;     /* 无符号整数,重要 */
typedef signed int sword;       /* 有符号整数,重要 */

typedef char eb1;               /* 有符号字符,不重要 */
typedef unsigned char ub1;      /* 无符号字符,重要 */
typedef signed char sb1;        /* 有符号字符,重要 */

typedef unsigned char text;     /* 无符号字符串 */
typedef short eb2;              /* 有符号短整型数,不重要 */
typedef unsigned short ub2;     /* 无符号短整型数,重要 */
typedef signed short sb2;       /* 有符号短整型数,重要 */
typedef long eb4;               /* 有符号长整型数,不重要 */
typedef unsigned long ub4;      /* 无符号长整型数,重要 */
typedef signed long sb4;        /* 有符号长整型数,重要 */
```

2.1.2　登录数据区

登录数据区(Logon Data Area 简称 LDA)是 C 程序中的一个结构变量，它用于建立程序和 ORACLE 数据库管理系统的通讯。其构成如下：

```
struct lda_def {
    sb2     v2_rc;      /* v2 返回码 */
    ub2     ft;         /* SQL 函数类型 */
    ub4     rpc;        /* 处理的行数 */
```

```
ub2         peo;            /*分析错误的相对位移 */
ub1         fc;             /* OCI 函数码 */
ub1         rcs1;           /* 填充区 */
ub2         rc              /* V7 返回码 */
ub1         wrn;            /* 警告标志 */
ub1         rcs2;           /* 保留没用 */
sword       rcs3;           /* 保留没用 */
struct {                    /* 行标识结构 */
        struct {
                ub4     rcs4;
                ub2     rcs5;
                ub1     rcs6;
        } rd;
        ub4     rcs7;
        ub2     rcs8;
} rid;
sword ose;                          /* OSD 依赖错误 */
dvoid *rcsp;                        /* 指向保留区的指针 */
ub1 rcs9[64-sizeof(struct cda_head)]; /* 填充区 */
};
```

最常使用的字段是返回码字段，在 OCI 的新版本中，检查错误信息时不使用 V2 返回码字段，该字段仅是为了与老的版本兼容而设置。

LDA 中字段的长度和相对位移依赖于系统。C 程序员应使用头文件 ocidfn.h 中的 LDA 定义(即上述结构变量)。

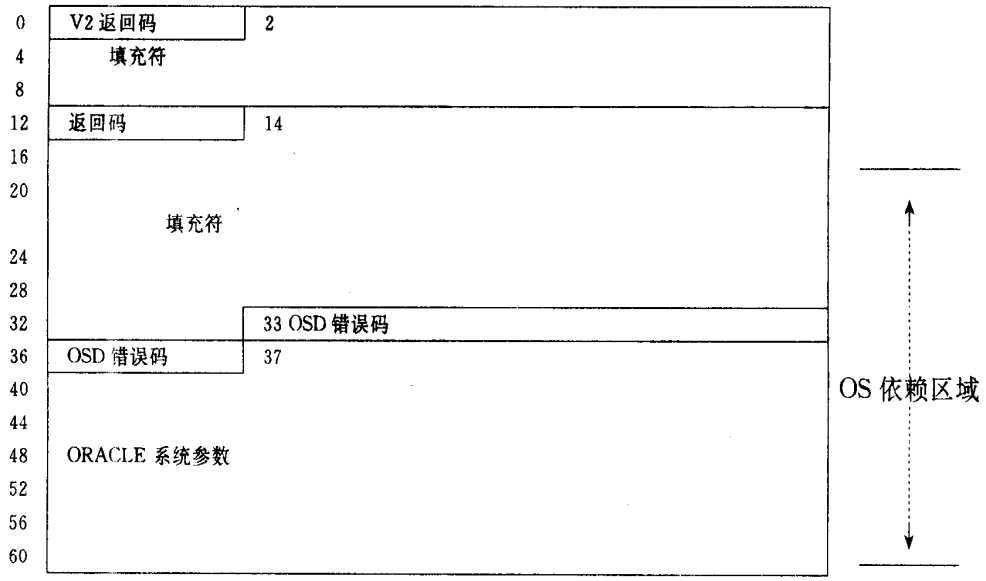


图 2-1 登录数据区

程序通过登录数据区建立与ORACLE数据库管理系统的连接。一个OCI应用程序在对数据库中的数据进行操作之前，必须先登录到ORACLE数据库上。OCI程序使用orlon或olon函数来完成该登录，建立与ORACLE的通讯。当一个应用程序并发连接到多个数据库上时，它必须分别为每一个并发连接说明一个LDA。当程序切断与一个ORACLE数据库的连接时，相应的LDA就能被重新使用。

注意：在建立了一个连接之后，就不能在处理过程中重新改变LDA的位置。因为在处理OCI的调用中，ORACLE使用该区域的地址。所以，该区域的地址在连接生存期过程中必须保持不变。

2.1.3 宿主数据区

宿主数据区(Host Data Area—简称HDA)是一个256字节的数据结构。它被用于与ORACLE数据库相连接。当程序要与ORACLE数据库相连接时，就必须为它分配该数据结构。当程序同时要与多个ORACLE数据库相连接时，则对每一个与ORACLE发生的并发连接，除了要为它分配一个LDA外，还应分配一个HDA。应用程序通过调用orlon或olon函数把这些数据区的地址传递给ORACLE。

注意：与LDA类似，在建立了一个连接之后，也不能在处理过程中又改变HDA的位置。因为在处理OCI的调用中，ORACLE使用这个区域的地址。所以，该区域的地址在连接生存期过程中必须保持不变。

2.1.4 光标数据区

1. 什么叫 光标数据区

同登录数据区类似，光标数据区(Cursor Data Area—简称CDA)也是C程序中的一个结构变量，它的构成同LDA，即如下所示：

```
struct  cda_def {
        sb2         v2_rc;          /* v2 返回码 */
        ub2         ft;             /* SQL 函数类型 */
        ub4         rpc;            /* 处理的行数 */
        ub2         peo;            /*分析错误的相对位移 */
        ub1         fc;             /* OCI 函数码 */
        ub1         rcs1;           /* 填充区 */
        ub2         rc              /* V7 返回码 */
        ub1         wrn;            /* 警告标志 */
        ub1         rcs2;           /* 保留没用 */
        sword       rcs3;           /* 保留没用 */
        struct {                    /* 行标识结构 */
          struct{
                ub4     rcs4;
                ub2     rcs5;
                ub1     rcs6;
```

```
        }rd;
        ub4     rcs7;
        ub2     rcs8;
    }rid;
    sword ose;                              /* OSD 依赖错误 */
    dvoid *rcsp;                            /* 指向保留区的指针 */
    ub1     rcs9[64-sizeof(struct cda_head)]; /* filler to 64 */
};
```

对于一个 32 位的机器系统来说,CDA 也是由 64 个字节所组成。图 2-2 表示一个典型的 CDA 结构。虽然 CDA 中的字段长度和相对位移依赖于系统。但是,对于所有的系统来说,全部字段都是按这个样子排列的。C 程序员应该使用上面所列出的 CDA 定义。

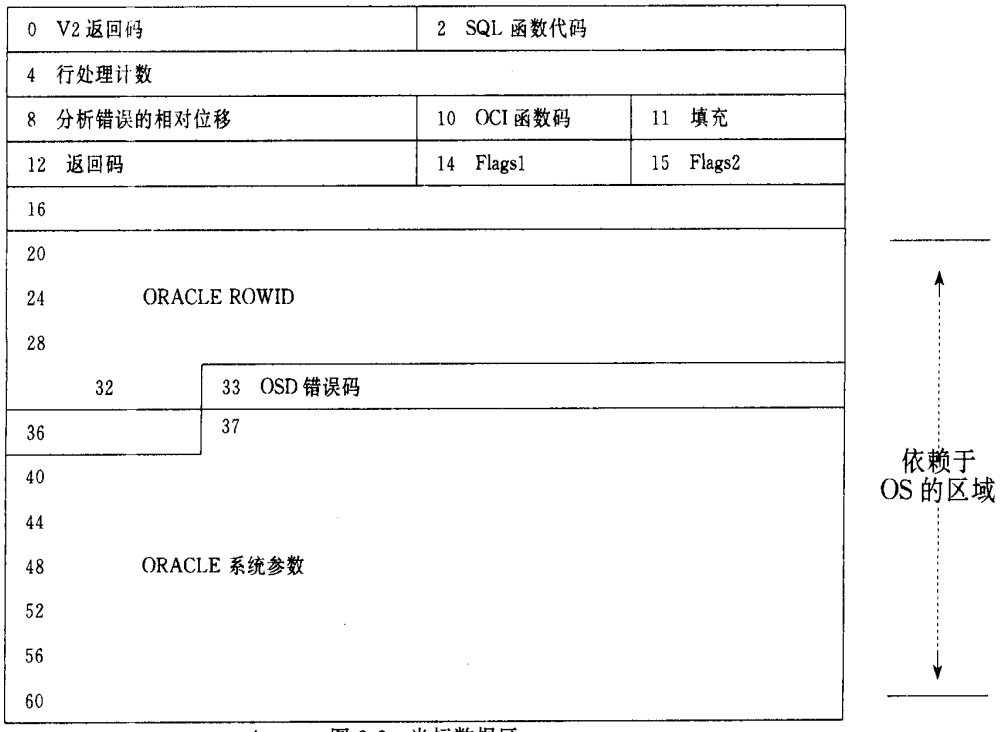


图 2-2 光标数据区

CDA 中每一个字段的描述如下:

(1)V2 返回码:是一个 2 字节的二进制整数字段,它保存(ORACLE 2 版的)OCI 调用的返回码。该字段只是为与老版本兼容而设置。在新的 OCI 版本中不使用它,而是使用返回码字段。

(2)SQL 函数代码:

是 2 字节的二进制整数字段,ORACLE 在内部使用它。每一个 SQL 命令都有一个函数代码。SQL 函数代码随 OCI 版本不同而改变。表 2-1 列出了它们的代码。

表 2-1 SQL 函数代码

代码	SQL 函数	代码	SQL 函数
01	CREATE TABLE	39	AUDIT
02	SET ROLE	40	NOAUDIT
03	INSERT	41	ALTER INDEX
04	SELECT	42	CREATE EXTERNAL DATABASE
05	UPDATE	43	DROP EXTERNAL DATABASE
06	DROP ROLE	44	CREATE DATABASE
07	DROP VIEW	45	ALTER DATABASE
08	DROP TABLE	46	CREATE ROLLBACK SEGMENT
09	DELETE	47	ALTER ROLLBACK SEGMENT
10	CREATE VIEW	48	DROP ROLLBACK SEGMENT
11	DROP USER	49	CREATE TABLESPACE
12	CREATE ROLE	50	ALTER TABLESPACE
13	CREATE SEQUENCE	51	DROP TABLESPACE
14	ALTER SEQUENCE	52	ALTER SESSION
15	(not used)	53	ALTER USER
16	DROP SEQUENCE	54	COMMIT
17	CREATE SCHEMA	55	ROLLBACK
18	CREATE CLUSTER	56	SAVEPOINT
19	CHRATE USER	57	CREATE CONTROL FILE
20	CREATE INDEX	58	ALTER TRACING
21	DROP INDEX	59	CREATE TRIGGER
22	DROP CLUSTER	60	ALTER TRIGGER
23	VALIDATE INDEX	61	DROP TRIGGER
24	CREATE PROCEDURE	62	ANALYZE TABLE
25	ALTER PROCEDURE	63	ANALYZE INDEX
26	ALTER TABLE	64	ANALYZE CLUSTER
27	EXPLAIN	65	CREATE PROFILE
28	GRANT	66	DROP PROFILE
29	REVOKE	67	ALTER PROFILE
30	CREATE SYNONYM	68	DROP PROCEDURE
31	DROP SYNONYM	69	(not used)
32	ALTER SYSTEM SWITCHLOG	70	ALTER RESOURCE COST
33	SET TRANSACTION	71	CREATE SNAPSHOTLOG
34	PL/SQL EXECUTE	72	ALTER SNAPSHOTLOG
35	LOCK TABLE	73	DROP SNAPSHOTLOG
36	(not used)	74	CREATE SNAPSHOT
37	RENAME	75	ALTER SNAPSHOT
38	COMMENT	76	DROP SNAPSHOT

(3) 行处理计数

该字段是 4 字节的二进制整数，它用于统计一个 SQL 语句处理的行数。例如，当执行插入、更新或删除语句时，它指出插入、更新或删除的行数；或者累计从查询结果集中提取的行数。

行处理计数字段仅在 oexec、oexn、oexfet、ofen 或 ofetch 调用之后有效。当调用 oexec 或

oexn 进行查询时，该字段被重新设置为 0，而在执行 ofetch 或 ofen 之后增量。在执行 oexfet 时，它也被重新设置为 0，而当提取完成时被设置。

(4) 分析错误的相对位移

该字段是 2 字节的二进制整数，在分析 SQL 语句时，它指出出现分析错误的开始字符位置。SQL 语句的第一个字符的相对位移为 0。分析错误有多种情况，例如可以是语法错误、违背安全、或不存在的表或列。该字段仅在 oparse 或 osql3 调用之后有效。对于 oparse 或 osql3 之外的 OCI 调用，虽然在该字段中可能会保留一个值，但是该值是没有意义的。

但要注意：如果 oparse 调用被延迟，则该字段仅在分析实际完成之后才有效。

(5) OCI 函数码

该字段是一字节的二进制整数，它指出当前完成的 OCI 函数。对于每一个使用 CDA 的 OCI 函数都有一个函数码。只有引用 LDA 的函数没有函数码。表 2-2 列出了每一个使用 CDA 的函数的 OCI 函数码。未列出的函数是没有 OCI 函数码。

表 2-2 OCI 函数码

OCI 例程码	OCI 例程	OCI 例程码	OCI 例程
04	OEXEC、OEXEN	30	OBNDRN
08	ODEFIN	34	OOPT
12	OFETCH、OFEN	52	OCAN
14	OOPEN	54	OPARSE
16	OCLOSE	56	OEXFET
22	ODSC	58	OFLNG
24	ONAME	60	ODESCR
26	OSQL3	62	OBNDRA
28	OBNDRV		

(6) 返回码

返回码字段是 2 字节的正二进制整数，它包含执行当前语句所产生的 ORACLE 错误码。错误码和信息在 ORACLE7 Server Messages and Codes Manual 中列出。可用 oerhms 例程来检索与该返回码相联系的错误信息文本。

(7) Flags1

该字段包含位警告标志，能设置多位。表 2-3 按位值列出这些标志。

表 2-3 警告标志

位值	十六进制值	含　义
1	1	有一个警告。当 Flags1 的任何一个其它位被设置时，该位就被设置。
2	2	如果在提取操作时，有一个数据项被截短，则该位被设置。
4	4	该位尚未使用。
8	8	该位尚未使用。
16	10	如果 UPDATE 或 DELETE 语句未包含 WHERE 子句的话，该位被设置。该位通过 osql3 来设置，当分析被执行时(可能被延迟)，通过 oparse 来设置。
32	20	PL/SQL 包装或过程被编译并送入数据库中，但是有编译错误。当 CREATE PROCEDURE、CREATE FUNCTION、CREATE PACKAGE 或 CREATE PACKAGE BODY 引起编译错误时，该位被设置。

64	40	当一个致命错误发生和一个事务被完全回滚时，该位被设置。在比7版低的版本中，不用此位。
128	80	该位尚未使用

(8) Flag2

该位目前尚未使用

(9)ORACLE ROWID

该字段以外部二进制格式(等价于外部数据类型11)存放ROWID，且在INSERT、UPDATE、DEIETE和SELECT FOR UPPATE操作之后有效。

ORACLE ROWID的尺寸依赖于系统，但可用以下几种方法来确定该尺寸：

·C程序员可使用sizeof(rid)来确定，其中rid是在头文件ocidfn.h中定义的CDA子结构。

·通过查询中描述的选择表项ROWID(例如，"SELECT ROWID FROM dual")来确定该字段的长度。使用odescr函数来描述选择表项，如果dbtype参数返回数据类型码11，则dbsize参数包含ROWID的正确二进制长度。(如果dbtype返回类型码1，则说明OCI程序被连到非ORACLE数据管理程序上。)

·参考所用系统的ORACLE安装手册和用户指南。

(10) OSD错误码

该字段包含一个与ORACLE相关的依赖于操作系统的错误码。例如，当试图执行磁盘I/O时，ORACLE接到了一个错误，则该字段被设置为依赖于操作系统的I/O系统失败码。

2. CDA的作用

它为程序中的用户光标和ORACLE中的SQL语句的分析表达式之间提供一个映象。有关SQL或PL/SQL语句的信息被保留在系统逻辑区(SGA)，和私用SQL区中。当ORACLE处理程序中的SQL语句时，它更新CDA中的字段，以表示语句处理的进度和状态。

2.2 SQL语句的处理

2.2.1 SQL语句的类型

OCI应用程序处理SQL语句的方法取决于语句类型。因此，当在程序中写处理SQL语句的代码时，必须考虑到要处理的语句类型。如果应用程序处理的是动态SQL语句，则在分析语句之后，可通过检查SQL函数码，以确定处理该语句所需要的步骤。实例中的第二个例子说明了此点。

在ORACLE7中有如下几类SQL语句：

(1)数据定义语言语句(DDL)

该类语句用于管理数据库中的实体，如建立新表、删去老表、和创立其它模式对象。例如：

```
CREATE TABLE emp
    (empno NUMBER(4),name CHAR(20), job CHAR(40))
DROP TABLE emp
```

GRANT UPDATE,INSERT,DELETE ON emp TO scott

REVOKE UPDATE ON emp FROM scott

(2) 控制语句

事务控制、会话控制和系统控制语句的处理类似于 OCI 应用程序中数据操纵语言语句。

(3) 数据操纵语言语句

这类语句用于改变数据库表中的数据。例如,把新行插入到表中,更新当前行中的列值,从表中删除行,封锁数据库中的表,说明 SQL 语句的执行策略。这类语句通常需要用输入变量把数据从程序输入到数据库。

查询语句用于从数据库检索数据。一个查询能返回 0、1 或多行数据。所有查询都由 SQL 关键字 SELECT 开始。因为查询要存取表中的数据,所以,常常也把它归入数据操纵语句一类。

(4)嵌入 SQL 语句

OCI 应用程序中不需要该类语句。

2.2.2 SQL 语句的处理步骤:

SQL 语句的处理步骤随语句的不同而不同,有些 SQL 语句只需一步,而另一些可能需四步或五步。OCI 程序员在编写处理 SQL 语句的代码时,应注意这点。一般来说,SQL 语句的处理步骤如下:

(1) 数据定义语句的处理步骤:

(a)用 oparse 分析该语句。

对不接收输入值或返回结果的数据定义语句,如果程序以非延迟方式连接,或以延迟方式连接且 oparse 的参数 defflg 是 0 的话,则能由 oparse 直接执行。不需要另外的处理步骤。

(2) 控制语句的处理步骤:

(a) 用 oparse 分析该语句

(b) 调用 oexn 来执行该语句

(3) 对数据操纵语句

(a) 用 oparse 分析该语句

(b) 调用 obndra、obndrn 或 obndrv 来把输入变量或数组的地址结合到 SQL 语句中的每个虚拟变量上。

(c)调用 oexn 来执行该语句。

(4)SELECT 语句的处理步骤:

(a) 用 oparse 分析该语句

(b) 调用 obndra、obndrn 或 obndrv 来把输入变量或数组的地址结合到 SQL 语句中的每个虚拟变量上。

(c) 使用 odescr 描述选择表项。该步是可选的;如果选择表项的个数和每一项的属性(例如长度和数据类型)在编译时已知,则不需要该步。

(d) 调用 odefin 来为 SQL 语句中每一个选择表项定义一个输出变量。

(e) 调用 oexfet 来执行该语句,然后提取满足该语句的行。如果结果集中的所有行不是

由 oexfet 来检索，就需要调用 ofen(也许多次)，来提取剩余的行。

以上各步的更详细描述请参考 OCI 程序的结构一节。

2.2.3 语句的延迟执行

1. 语句为什么要延迟执行

在 ORACLE 6 版和更早的版本中，对 OCI 例程的每一个调用都要求对 RDBMS 服务器做相应的调用。例如，当用 osql3 分析一个 SQL 语句时，就要把该 SQL 语句的文本传送给 RDBMS，并对它进行分析和把它存储在用户全局区(UGA)中。如果该语句中含有虚拟变量，则要通过把实输入变量的地址传递给 RDBMS 来实现变量的虚实结合。这就要求对每一个输入变量调用一次 RDBMS。对查询来说还需要对 RDBMS 作更多的调用，以便定义和保留选择表项的程序变量的地址；例如，当执行语句时，ORACLE 要求输入变量的值、执行结束时，还要返回结果。

当 OCI 应用和 ORACLE 数据库是在同一个机器上运行时，则对 ORACLE 的多次调用对应用程序性能的影响还不大。但是，在网络客户/服务器环境下，因为 OCI 应用程序是在客户机上运行，而与该程序所连接的 ORACLE 数据库可能是在不同的系统上(例如是远离数千公里之外的地方)运行，则对该数据库的多次反复调用就会大大降低应用性能。

为了提高性能，OCI 与 ORACLE7 数据库管理系统在处理 SQL 语句时，允许一步或多步的延迟执行。例如，分析 SQL 语句、结合输入变量以及定义输出变量这些步能延迟到该语句被实际执行时才处理。

2. 实现语句延迟执行的方法

语句的延迟执行是指分析、结合和定义步能延迟到与执行步一起执行。分析、结合和定义步的延迟执行是由如下两个因素控制：

- 连接 OCI 程序的方法
- 如何在 oparse 调用中设置 defflg 参数

所以可采用如下方法实现语句延迟执行：

(1) 用延迟方式连接：当用延迟方式把 OCI 程序连接到 ORACLE7 数据库上时，结合和定义步就一直被延迟到该语句被执行。这种延迟与应用程序使用的特殊 OCI 调用无关；只与所选的连接可选项有关。例如，如果一个 OCI 应用程序用延迟方式的连接可选项重新连接的话，那么结合和定义步总被延迟。

(2) 在 oparse 调用中使用设置为 0 的 defflg 参数：当用延迟方式连接程序时，如果 ORACLE7 oparse 调用使用设置为 0 的 defflg 参数来处理一个 SQL 语句，则该语句的分析步也被延迟。于是，在 SQL 语句被实际执行之前能延迟它的所有操作。对于版本 6 的 OCI 应用程序，必须用 oparse(替换 osql3)重新编码，以获得延迟分析。

对于 ORACLE7 的 OCI 应用程序，如果对它不作任何改变而采用非延迟方式连接，则它仍能运行，只是导致调用的状态类似于版本 6 的 OCI 应用情况。

如果希望使用语句延迟执行的优点而重新连接现存的版本 6 的 OCI 应用时，必须小心。应该仔细地检查现有的 OCI 应用，以确定延迟方式的执行在不改变应用逻辑的情况下能否执行。图 2-3 和图 2-4 说明：分析、结合、定义和执行/提取 OCI 调用怎样用延迟和非延迟分析与 ORACLE 通讯。

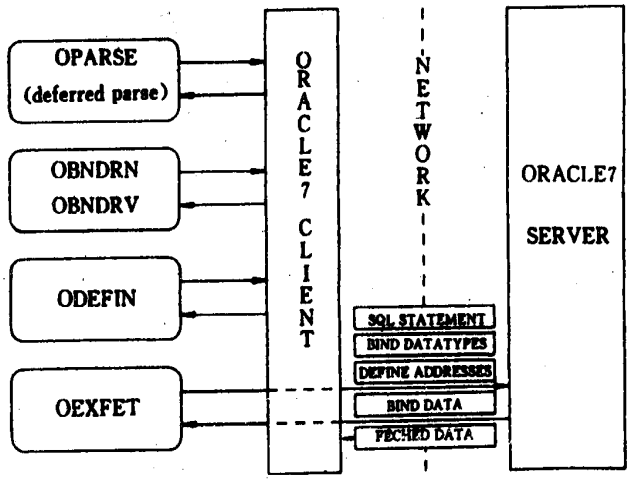


图 2-3　语句处理延迟分析

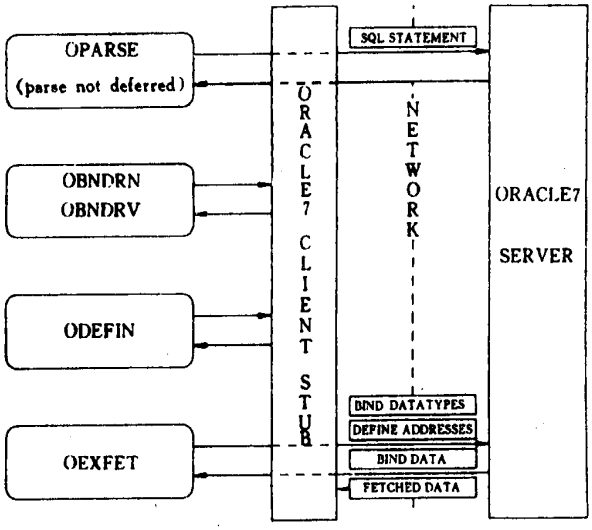


图 2-4　语句处理非延迟分析

3. 使用语句延迟执行的缺点

用延迟方式进行连接时，选择的是 ORACLE7 OCI 库，这些库用动态方法分配内存以延迟客户系统上变量的结合和定义信息，直到 ORACLE 为处理 SQL 语句需要这些信息。这个方法需要更多的客户系统上的内存。如果客户系统上的内存资源不足，则应使用非延迟连接。

§2.3　OCI 程序的编码步骤

在 1.2 节我们给出了一个 OCI 应用程序的基本结构，本节更详细的描述编写 OCI 应用程序的每一步。其中某些步是可选择的；例如，对于非查询语句不需要描述或定义选择表项。

(1)定义 OCI 数据结构

在连接到 ORACLE 之前，程序必须至少定义一个 LDA。如果程序需要多个并发连接，则对于每一个连接都需定义一个 LDA。在 C 中通常用单个 LDA 结构或一个这样结构的数

组来定义。如果用 orlon 调用来连接到 ORACLE 上的话，还必须对每一个 LDA 定义一宿主数据区(HDA)。

为了处理 SQL 语句，需定义一个或多个 CDA。对于每一个同时激活的 SQL 语句，都需定义一个 CDA。如果程序是串行处理一些 SQL 语句的话，则可只需定义一个 CDA。CDA 的定义如同 LDA。

(2)连接到 ORACLE 上

程序能通过调用 orlon 例程建立与一个或多个 ORACLE 数据库的通讯。为了连接到 ORACLE 上，首先应在程序中定义一个 LDA 和 HDA。在连接时所建立的与 ORACLE 的通讯要用所定义的 LDA 和 HDA 来实现。

如果 OCI 程序在某一时刻仅需要一个数据库连接时，可交互地使用 orlon 和 ologof 来接通和切断该连接。

但是，能用 orlon 进行任意个数的并发连接。

(3)打开光标

为了处理 SQL 语句，必须打开一个光标。可通过调用 oopen 例程来打开一个光标。光标打开是在与 ORACLE 相连接之后进行，因为 oopen 调用需要一个有效的 LDA 参数。该步必须在 CDA 能被用来分析一个 SQL 语句之前完成。

能使用光标来重复执行同一个 SQL 语句或执行一个新的 SQL 语句。当再次使用一个光标时，程序中相应 CDA 的内容在分析新的 SQL 语句时被重新设置。在再次使用该光标之前，不需要关闭和再打开它。

(4)分析语句

必须使用 oparse 例程来分析每一个 SQL 语句，使该语句与程序中的一个光标相联系。如果 oparse 的 defflg 参数是 0 的话，则数据定义语言语句在分析时执行，而不需要进一步的处理；如果 defflgG 参数是非 0，就必须调用 oexn 或 oexec 执行该 SQL 语句。所有的数据操纵语言语句和查询都需要进一步的处理。

(5)结合输入变量的地址

大多数数据操纵语言语句和一些查询语句都需要把程序中的数据输入给 ORACLE。这些数据可能是常数或文字数据，它们在编译时是已知的。例如，SQL 语句

```
INSERT INTO emp (empno,ename,job)
        VALUES( 65147,'Ficklin Vineyards','software');
```

包含几个文字串，如 'Ficklin Vineyards','software'。这类语句很受限制，它不能在运行时提供输入数据。

为了编写在运行时能提供输入数据的程序，则程序的 SQL 语句中应该使用虚拟变量。SQL 语句中的虚拟变量标志在什么地方提供数据。例如，SQL 语句

```
INSERT INTO emp(empno,ename,job)
        VALUES(:empno,:ename,:job);
```

包含了 3 个虚拟变量，它们由冒号(:)打头，表示程序必须通过它们提供输入数据。在 DELETE、INSERT 或 UPDATE 语句中能使用输入虚拟变量，对应于任何一个输入虚拟变量能使用一个表达式或文字串值。

对于 SQL 语句中的每一个虚拟变量，必须调用 OCI 例程来使程序中的一个实变量地

址结合到该虚拟变量上。这样，当程序被执行时，ORACLE 才能取得放置在该实输入(或结合)变量中的数据。

当执行结合步时，数据还不必被设置在结合变量中。结合步只告诉 ORACLE 结合变量的地址，以及数据类型和长度。但是，当执行 SQL 语句时，要确保该变量包含有效的值。

有三个 OCI 例程能用来把地址结合到虚拟变量上。它们是 obndrv、obndrn 和 obndra。

当使用 obndrv 例程时，必须指出虚拟变量的名字，例如上述语句中的":empno"等。OBNDRV 能在交互应用中使用，在交互方式下，用户可在运行时输入一个 SQL 语句。但在这种情况下，程序必须扫描 SQL 语句，以获得虚拟变量名。obndrv 例程的虚拟变量名不能是保留字，例如下面的 SQL 语句是非法的，因为 ROWID 是一个保留字：

```
SELECT ename FROM emp WHERE rowid=:ROWID
```

obndrn 是另一个把地址与虚拟变量相结合的例程。要使用这个例程，每一个虚拟变量的形式必须是:N，其中 N 是文字整数，范围在 1 和 255 之间。例如

```
SELECT ename,sal FROM emp
    WHERE (job=:1 AND sal>:2) OR
        (job ! =:1 AND sal<:2)
```

对于 obndrn 例程来说是一个有效的 SQL 语句。在这个语句中，有 4 个虚拟变量实例，但实际上只有 2 个虚拟变量。于是，只需要两个结合变量。对该语句，只需调用 2 次 obndrn。在一个 SQL 语句中，一个虚拟变量的所有出现只需通过一次调用就实现其结合。obndrn 允许使用一个索引变量来重复一组虚拟变量。不能用 obndrn 来结合 PL/SQL 块中的变量，必须用 obndra 或 obndrv。

obndra 例程用于把程序中的标量或数组的地址结合到 SQL 语句或 PL/SQL 块中的虚拟变量上。obndra 类似于 obndrv，但它提供辅助参数以指出数组的最大尺寸、返回数组的元素个数和长度、以及返回诸列错误的辅助参数。

(6) 描述选择表项

如果 SQL 语句是查询语句，就需要获得选择表项更多的信息，特别是对于动态查询语句更是如此。因为在动态情况下，程序预先还不知道选择表项的数据类型、列长以及显示尺寸等有关信息。

例如，用户可以输入这样一个查询

```
SELECT * FROM emp
```

其中缺少表 EMP 的列信息。

能用 odescr(描述)例程来获得这方面的信息。它返回需要确定怎样转换、显示或存储查询时所返回的数据信息。

为了处理动态选择表项，应在循环中调用 odescr 。在循环的开始，先把索引变量设置为 1，然后增加它，在每次重复时进行描述，直至"Variabhe not in list"错误在 CDA 的返回码字段返回。

(7) 定义选择表项

对于查询，应使用 odefin 例程来把程序中的输出变量(或数组)与查询中的每一个选择表项相联系。如果选择表项的个数预先不知道，如

```
SELECT * FROM emp
```

则可首先重复调用 odescr 来确定该个数。也可在同一个循环中先调用 odescr 而后调用 odefin，当 odescr 返回“Variable not in select_list”错误时退出循环。

但是，不使用 odefin 来定义 PL/SQL 块内的 SQL SELECT 语句中的选择表项。在这种情况中必须使用 obndra 或 obndrv。

能重新调用 odefin 来“再定义”输出变量，而不必再次分析或再次执行 SQL 语句。

(8) 执行语句

如果 SQL 语句是数据操纵语言语句，还必须执行该语句。该执行操作把所有结合变量中的值输入给 ORACLE。

有两种把数据输入给 ORACLE 的方法。其一是用 oexec 例程来重复执行 SQL 语句，并在每次交互时提供不同的输入值。其二是用 ORACLE 数组接口，通过使用 oexn 例程，借助单个语句输入许多值。(也能用 oexn 来执行仅处理单行数据的语句)。

当需要更新或插入大量数据时，使用数组接口将大大降低与 ORACLE 的通讯传输。这将大大改进性能，特别是在客户/服务器环境下。

(9) 提取查询行

在定义了输出变量的地址以后，就可通过调用下列三个例程之一来提取满足一个查询的行：

- oexfet
- ofen
- ofetch

如果打算用 oexfet 或 ofen 来提取多行的话，就必须确保为选择表项定义的输出变量是数组。如果使用数组接口，还必须有可选择的数组 OUT 参数。例如，指示器变量和列的输出变量必须也是数组。

(10) 关闭光标

在程序退出之前，或打开一组新光标之前，要用 oclose 例程来关闭每一个打开的光标。一旦光标被关闭，光标数据区就不再与 ORACLE 服务器相联系，并且该光标所用的服务器中的内存区域被释放。

(11) 切断与 ORACLE 的连接

在程序退出之前要调用 ologof 例程来终止与 ORACLE 的连接。对于每一个在 olon 或 orlon 调用中引用的 LDA，都要调用 ologof。

综上所述，一个 OCI 程序的结构如图 2-5 所示。

§2.4 OCI 程序的编码规则

在编写 OCI 应用程序时，应遵循如下一些规则：

(1)参数的数据类型

在 OCI 调用中使用如下三类参数：

- 地址
- 二进制整数
- 短二进制整数

地址参数把变量的地址传递给 ORACLE。当用 C 语言开发应用时，应保证给地址参数

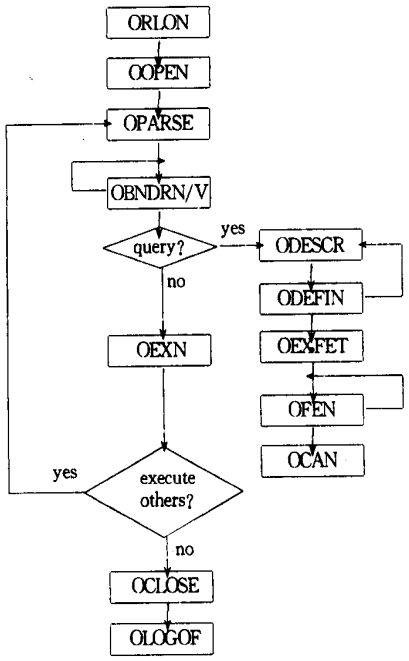


图2-5　OCI程序的逻辑流程

传递一个地址。例如，如果一个例程的参数被指定为短整型数的地址，并且所用的参数是rcode，则应以 &rcode 传递该参数。

二进制整型数的大小依赖于系统。短二进制整型数参数是较小的数，其大小也依赖于系统。

(2) 字符串

字符串是一个特殊类型的地址参数。每一个允许字符串的 OCI 例程也有一个串长参数，长度参数应被设置为串的精确长度。如果串以 Null 字符结尾，则能把长度参数设置为－1(不用 0)。

如果编译程序许可的话，能传递文字字符串。但是要注意，因为字符串是地址参数，所以程序实际传递的是地址。

(3) 指示器变量

对于数据操纵语言语句或查询语句，每一个结合和定义 OCI 调用(obndra、obndrv、obndrn 和 odefin)都有一个允许使用指示器变量的参数，或数组指示器变量(如果使用数组的话)。指示器变量的含义基本上同前面章节所述的一样。例如，对输入宿主变量，OCI 程序能赋给指示变量的值的含义如下：

－1：ORACLE 将把 Null 值赋给数据库的表列，而不考虑输入宿主变量的值。

≥0：ORACLE 将把输入变量的值赋给数据库的表列。

对于输出宿主变量，ORACLE 赋给指示器变量的值有如下含义：

－2：表示数据比能返回的最大数据长度还长。此时返回值被截短。

－1：列值是 Null，输出变量的值不变。

0：ORACLE 把列值原封不动地赋给宿主变量。

>0：数据库表列的长度大于输出变量的长度；项被截短。指示器变量中返回的正值是截短之前的实际长度。

(4) Null

在 OCI 程序中,把 Null 插入数据库表列中有如下几种方法:

· 在 INSERT 或 UPDATE 的语句文本中用串 Null,如 SQL 语句

```
INSERT INTO emp (ename,empno,deptno)
        VALUES(Null,8010,20)
```

使 ename 列为 Null。

· 在 OCI 结合调用中使用指示器变量。

在从数据库中提取数据时,为了发现 null,可在 obndra 或 odefin 例程中指定指示器参数,然后,在 oexfet、ofen 或 ofetch 之后检查返回的值。也能用列级别 rcode 参数来发现 Null。

(5) 撤消调用

在大多数平台上,用户能交互地撤消一个长时间运行或循环的 OCI 调用(如 oexn、oexec、oexfet、ofetch 或 ofen)。这可通过在键盘上输入操作系统的中断字符(通常是 CTRL—C)来实现。

当用操作系统的中断字符来撤消长时间运行或循环的调用时,则在 CDA 的返回码字段返回错误码 1013("user requested cancel of current operation")

如果 OCI 程序需要用一个机制(如时钟)来撤消一个长时间运行的调用时,可使用 O-BREAK 例程。

(6) 最大的数组

数组元素的最大个数是 32767。也就是说,oexn、ofen 或 oexfet 的 iters 或 nrows 参数不能被设置为大于该极限的值。

(7) 使用 ROWID

在 SELECT…FOR UPDATE OF…之后,CDA 中包含 ROWID。因此,在稍后的 UP-DATE 或 DELETE 语句中就可使用该二进制 ROWID。例如,对于如下的 SQL 语句:

```
SELECT ename
    FROM emp
    WHERE empno=7499 FOR UPDATE OF sal;
```

当 FETCH 被执行时,CDA 中的 ROWID 字段就包含被提取行的行标识符。可把该 ROWID 拷贝到程序内的一个缓冲区中,而后在 DELETE 或 UPDATE 语句中使用该保存的 ROWID。例如,如果 MY_OWID 是缓冲区,行标识被保存在该缓冲区,此后就能通过把新的 salary 结合到 :1 虚拟变量上和把 MY_ROWID 结合到 :2 虚拟变量上来处理如下的 SQL 语句:

```
UPDATE emp SET sal=:1 WHERE rowid=:2
```

当把 MY_ROWID 结合到 :2 上时,务必用数据类型码 11(ROWID)

(8) 优化编译程序

许多语言的编译器都优化所生成的代码,因此程序变量的地址并不精确反映任何时刻变量在内存中的实际存储单元。例如,优化器把共同使用的变量频繁地放置在机器的寄存器中,只是在子例程调用中引用它们时才把它们存放到内存单元中。

当在后边的调用中要用的变量地址作为参数被传递给 ORACLE 时,就必须保证所寻址的变量是在内存单元中,以确保后续的执行或提取调用。这适用于 odefin、obndrv 和 obn-

dra 调用。

保证变量地址当前值的最简方法是使编译器的优化器无效。然而许多编译器都提供了可选择的优化机制。例如，可以使局部的程序段或例程不优化。或者，对大多数 ANSI C 编译器来说，可把变量说明为 Volatile 使其优化无效。更详细内容请参考编译器参考手册。

§2.5 在 OCI 程序中使用 PL/SQL

2.5.1 OCI 程序中使用的 PL/SQL 块

PL/SQL 是 SQL 语言的扩展，它具有过程性，也可看作是 ORACLE 的过程语言。它用于处理更复杂的任务。PL/SQL 允许把若干个结构组合为一个块，并作为一个单位来执行它。这些结构可以是：

- 一个或多个 SQL 语句
- 变量说明
- 赋值语句
- 过程控制语句，如 IF—THEN—ELSE 语句和循环。
- 例外处理

在 OCI 程序中能使用 PL/SQL 块来做如下工作：

- 调用 ORACLE 存储过程和存储函数
- 把过程控制语句和几个 SQL 语句相组合，作为一个单元来执行。
- 使用特殊的 PL/SQL 功能，如记录、表、CURSOR FOR 循环和例外处理。

2.5.2 PL/SQL 块的处理过程

PL/SQL 块的处理过程是：先把 PL/SQL 块放置在一个串变量中，然后分析该串，结合变量，最后执行它。

在把程序变量结合到 PL/SQL 块中的虚拟变量上时，必须用 obndra(或 obndrv)例程来完成该结合。不能使用 obndrn。可用 obndra 来结合标量或数组宿主变量，用 Oobndrv 只能结合标量宿主变量。

例如，在如下的 PL/SQL 块中

```
BEGIN
    SELECT ename,sal,comm
            INTO :emp_name,:salary,:commission
            WHERE empno=:emp_number;
    END;
```

应该用 obndra 来结合 :emp_name,:salary,:commission 和 :emp_number 位置上的虚拟变量。

注意：不能用 odefin 来结合 PL/SQL 块中的宿主变量，而必须用 obndra 或 obndrv。对新的 ORACLE7 OCI 应用程序使用 obndra。

2.5.3 PL/SQL 的应用举例

在 OCI 程序中，常常用 PL/SQL 块来调用存储过程和存储函数。例如，假设有一个存储

在数据库中的过程 RAISE_SALAY，现在我们想在 OCI 程序中调用该过程。这时只需在无名 PL/SQL 块中嵌入一个对该过程的调用即可。

下面是一个用 C 写的程序段，它说明如何在 OCI 应用程序中调用一个存储过程。

例 2.1 如何调用存储过程

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*     说明：
*          该程序段提示用户输入职员 ID 号和工资。然后，通过调用 oexec
*          例程，来执行 PL/SQL 块，进而执行存储过程 RAISE_SALARY。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 定义一个串变量，并用 PL/SQL 块来初始化它 。 */
char plsql_statement[]= "BEGIN\
                          RAISE_SALARY(:emp_number,:new_sal);\
                        END;"
/* 说明变量 */
int empnum;
float salary;

/* 先连接到 ORACLE，然后打开光标。 */
.....
/* 分析该语句。因为 PL/SQL 块的文本是以 Null 结尾，
  所以长度参数使用 -1 */
oparse(&cda,plsql_statement, -1,1,2);

/* 把 C 中的变量与虚拟宿主变量相结合，对于 empnum 传递地址(即 &empnum)，
  3 和 4 分别是 INTEGER 和 FLOAT 数据类型代码. */
obndrv (&cda,":emp_number", -1,&empnum,sizeof(int),3,
              -1,0,0,-1,-1);
obndrv (&cda,":new_sal",-1,&salary, sizeof(float),4,
              -1,0,0,-1,-1);

/* 提示用户输入职员号和工资 */
printf ("Enter the employee number:");
scanf("%d",&empnum);
fflush(stdin);
printf("Enter the new salary:");
scanf("%f",&salary);

/* 执行 PL/SQL 块，进而执行调用存储过程 */
oexec(&cda);
```

```
/*提交事务*/
ocom(&lda);
.....
```

2.5.4 PL/SQL 错误号及错误信息

当执行 OCI 程序中的 PL/SQL 块发生错误时，则在 CDA 的返回代码段返回 ORACLE 错误号(6500 组)。可通过调用 oerhms 例程来得到 PL/SQL 的特定错误代码及信息。对 oerhms 的一次调用能返回多个 PL/SQL 错误信息。因此，应该为该错误信息分配一个大的缓冲区。一般有 1000 字节就足够了。

2.5.5 数组的限制

当使用 obndra 来把 OCI 程序中的数组结合到 PL/SQL 表上时，能结合的数据类型有一些限定。表 2-4 表示能实现的转换。

表 2-4 能支持的数数据类型转换

INTERNAL TYPES	PL/SQL TYPES							
	VARCHAR2	LONG	RAW	LONG RAW	NUMBER	DATE	CHAR	
VARCHAR2	√	√	√	√				
LONG	√	√	√	√				
RAW	√	√	√	√				
LONG RAW	√	√	√	√				
NUMBER					√			
ROWID						√		
DATE							√	
CHAR								√

§2.6 开发 X/Open DTP 应用

一个 X/Open 应用是在分布事务处理(DPT)环境下操作的应用。在一个抽象的模型中，应用请求资源管理程序提供许多不同类型的服务。数据库资源管理程序为应用提供对数据库中数据的存取。事务处理监控程序把标准 X/Open 接口用于资源管理程序，它本身包含有关事务协调方面的事务管理程序。

图 2-6 所示的 DPT 模型的各组成部分能相互作用，它提供了一个对数据库中数据一致和有效的存取。在 DTP 模型中指定 XA 接口，ORACLE 提供 XA 依从库，以连接到应用中。当地接口是 OCI API。

DTP 模式指出事务和资源管理程序怎样与应用代码相互作用。

能用 ORACLE 调用接口来开发遵守 X/Open 标准的应用。为了做到这点，应遵循下面所描述的一些限制。

应用程序并不建立和维护与数据库的连接。数据库的连接和断开要通过事务管理程序和 XA 接口来显示地处理。(ORACLE 提供 XA 接口。)这意味着一般在 X/Open 依从的应用中不用 ORLON 或 OLON 调用，也不调用 OLOGOF 来切断连接。

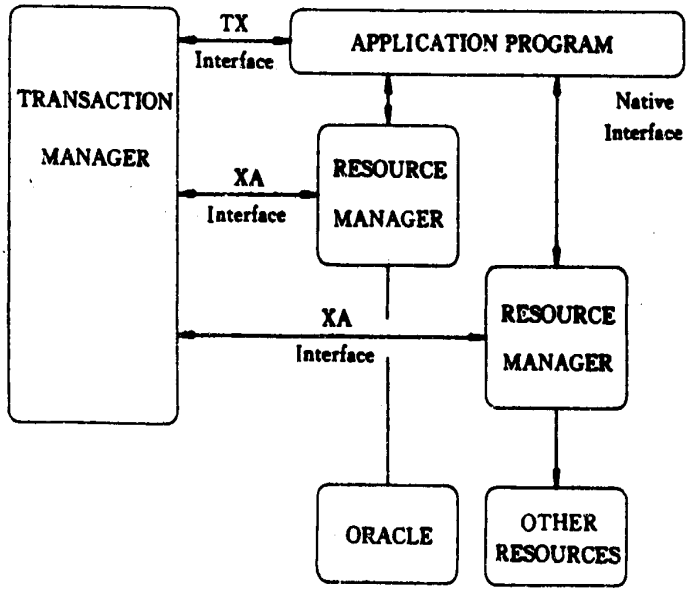


图 2-6　一个可能的 DTP 模型

用 OCI 写的应用要求一个有效的 LDA。可用 SQLLD2 例程来获得指定连接的有效 LDA。这里的连接是通过 XA 接口建立的。

应用代码一定不发 SQL 语句或进行影响全局事务状态的调用。对应用中的全局事务不能发 COMMIT，因为 COMMIT 必须由事务管理程序代码处理。也不能发 SQL 数据定义语言命令，因为数据定义语言命令执行隐式 COMMIT。当然对全局事务也不能用 OCI 调用 OCOM。

如果应用迁到一个妨碍进一步进行 SQL 操作的错误的话，它能执行一个内部的回滚。注意：在 XA 接口的以后版本中，这点有改变。

必须把 XA 库中的模块与应用的目标模块相连接，以获得 XA 接口的功能。因为 SQLLD2 是由 ORACLE 预编译程序库(SQLLIB)所提供，因此还必须把 SQLLIB 与应用程序的模块相连接。

在应用服务程序中，不能使用下面的 OCI 调用：

- OCOM、OCON、OCOF
- ORLON、OLON、OLOGOF

另外，不能发 SQL 定义语言命令，因为它们会引起隐式的提交，也不能发下面的事务控制 SQL 命令：

- COMMIT
- ROLLBACK
- SET TRANSACTION
- SAVEPOINT

第三章　OCI 程序的编码方法及所引用的 OCI 库函数

在本篇第 2 章描述了 OCI 程序的结构，本章进一步说明每一步的编码方法以及所引用的 OCI 库函数。本章所描述的库函数是关于 C 语言的 OCI 库函数，对于每一个函数都从如下五方面描述它：

- 功能
- 调用格式
- 参数说明
- 使用说明
- 相关函数

§3.1　调用 OCI 函数的几点说明

为了便于阅读，在描述每一步的编码方法之前，我们先就 OCI 函数调用的几个共同问题加一说明。

3.1.1　数据类型

本章例子中所用的数据类型是用户定义类型(请参考本篇 2.1.1 节和第 4 章的 oratypes.h 文件定义)。定义的类型(如 sb2 和 ub4)对不同的系统可能取不同的 C 类型。

3.1.2　数据结构

为了调用 OCI 函数，程序员必须定义一个或多个登录数据区(LDA)和光标数据区(CDA)。这两个数据区在结构和大小上相同，因此能使用同一个结构来说明二者。OCI 应用程序通常使用返回码字段。

在本章的实例代码中，使用结构 lda_def 和 cda_def 来定义 LDA 和 CDA。

3.1.3　参数

OCI 函数采用三类参数：

- 整型(sword,sb4,ub4)
- 短整型(sb2,ub2)
- 程序变量的地址(指针)

(1) 整型

当把整型文字串传递给 OCI 函数时，应当把文字串强制为参数的类型。例如，函数 oparse 有如下的原型：

```
oparse(struct cda_def *cursor, text *sqlstmt, sb4 sqllen, sword defflg, ub4 lngflg);
```

如果以下面的形式调用该函数：

oparse(&cda, "select sysdate from dual", -1,1,2);

C 编译程序将发出警告信息。因此,应该以下面的形式来调用:

oparse(&cda,(text)"seleat sysdate from dual",(sb4) -1,(sword) 1,(ub4) 2);

请注意:应当把有符号、无符号短整型数(sb2 和 ub2)与自然整数(sword)、有符号及无符号长整型数(sb4,ub4)区分开。

(2) 地址

所有指针参数都作为有效地址。当传递 NULL 指针(0)时,ORACLE 建议把它强制为相应的类型。

当参数是指针时,OCI 程序(决不是 OCI 函数)必须为指针所指的对象分配存储区。

串文字能被传递给参数类型为 text * 的 OCI 函数,如前面的 oparse 函数。

OCI 函数的参数有必须的参数、可选参数和不使用的参数等三种情况。

(1) 必须的参数

它是 ORACLE 所要求的参数,OCI 程序必须为它们提供有效的值。

(2) 可选参数

可选参数的使用取决于程序的需要,仅当需要时才指定有效的值。在每个函数的调用格式描述中,可选参数用方括号([])指出。当不需要指定有效的值时,如果它是一个整型数,则指定-1;如果它是地址参数,则指定 null 指针(0)。例如,对于结合调用,程序不需要提供指示器变量(因为程序中的所有输入值都不是 null)。结合函数 obndra,obndarv 和 obndrn 中的 indp 参数是可选的,它是指针;当不需要它时,就指定为 null 指针((sb2 *)0)。

(3) 不使用的参数

不使用的参数是 ORACLE 不需要的参数。在调用格式描述中用尖括号(< >))指出。在 C 语言中,对不使用的参数也必须指定,其方法是:如果是整型,则指定-1;如果是指针,则指定 0。

3.1.4 参数描述

对 OCI 函数的参数需进行类型和方式(mode)两方面的描述。

当参数是 LDA 或 CDA 时,类型为 struct lda_def * 或 struct cda_def *,即指向 stract lda_def 或 struct cda_def 的指针。OCI 程序不仅要分配指针,而且还要分配这些数据区。

当参数是一个广义指针(即指向任意变量,或数组变量的指针)时,则把它指定为 ubl 类型的指针。

参数的方式有三种可能值:

IN:是把数据传递给 ORACLE 的参数

OUT:是从 ORACLE 中检 索数据的参数

IN/OUT:是在调用时把数据传递给 ORACLE,返回时把检 索到的数据传递给 OCI 程序的参数。

3.1.5 函数返回值

当从 C 程序中调用 OCI 函数时,则返回一个整数值。如果函数正确完成,则返回 0。如果有错误,则返回非 0。当发生错误时,应检查光标数据区(CDA)中的返回码字段,以取得错

误号。

但函数 oerhms，sqlld2 和 sqllda 例外。其中 oerhms 返回信息长度。sqlld2 和 sqllda 是 void 函数，在 LDA 中返回错误说明。

3.1.6 变量的存储单元

当用函数 obndra、obndrv、obndrn 和 odefin 结合和定义程序变量时，ORACLE 是通过变量的地址来访问这些变量的。如果是结合变量，则在语句执行时，其地址必须保持不变。

如果把 LDA 和 CDA 结构传递给程序中的非 OCI 函数时，则必须把它们作为指针传递，而不是值传递。在执行 OCI 调用时，ORACLE 更新这些结构中的字段。如果程序使用这些结构的副本(若当前 OCI 调用没有更新它们)的话，则程序获得的可能不是当前信息。

§3.2 OCI 程序与 ORACLE 数据库的连接

OCI 程序要对数据库中的数据进行操作，首先必须连接到该数据库上(即登录)。其方法是通过调用 orlon 函数来建立该连接。一个程序能用 orlon 与一个或多个 ORACLE 数据库建立并发连接，以实现程序与这些数据库的通讯。

如果 OCI 程序在某一时刻仅需要与一个数据库连接时，可交互地使用 orlon 和 ologof 来接通和切断该连接。下面说明 orlon 函数的功能和用法：

(1) 功能

用该函数创立 OCI 程序与 ORACLE 数据库之间的连接和通讯。

(2) 调用格式

```
orlon (struct lda_def *lda, ubl *hda, text *uid,
       [sword uidl], [text *pswd], [sword pswdl],
       <sword audit>);
```

(3) 参数说明

lda：是指向 LDA 的指针

hda：是指向 256 字节的宿主数据区。

uid：是指向字符串的指针，该串包含一个用户名、可选择的口令、和可选择的宿主机标识符。如果包含口令的话，应把口令直接放在用户名之后，中间用“/”隔开。宿主机标识符放在用户名和口令之后，且用'@'字符领先。如果该参数中不包含口令，则口令必须在 pswd 参数中。下面是 uid 参数的有效例子：

name、name/password、name@d：nodename－dbname、
name/password@d：nodename－dbname.

下面的 uid 例子是不正确的：

name@d：inodename－dbname/password

uidl：是 uid 所指向的串的长度。如果该串以 Null 终结，则该参数应是－1。

pswd：是指向包含口令的串的指针。如果口令是 uid 所指串的一部分，则该参数应是 0。

pswdl：是口令的长度。如果 pswd 所指向的串是 Null 或以 Null 结尾，则该参数应是－1。

audit：该参数不再使用；只允许其值是 0 或－1。

(4) 使用说明

一个 OCI 程序能被多次连接到一个或多个 ORACLE 数据库上。对数据库的每一个并发连接都需要有一个 LDA 和 HDA。OCI 程序使用 LDA 和 HDA 与 ORACLE 进行通讯。

宿主数据区的内容对 ORACLE 完全是私有的,但 HDA 必须由 OCI 程序分配。

在 orlon 调用的执行过程中,对 HDA 和 LDA 所分配的内存地址应保持不变。

(5) 相关函数

ologof、olon、sqllda

下面的程序段展示了 OCI 程序与 ORACLE 进行连接的方法。

例 3.1 OCI 程序与 ORACLE 数据库的连接

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *     说明:
 *          该程序段实现与两个数据库的连接
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdlib.h>
/* 建立两个登录数据区 */
lda_def lda[2];
/* 建立两个宿主数据区 */
ubl hda[2][256];
/* 初始化识别名和口令 */
text *uid1="SCOTT/TIGER @ D: KERVMS_V7_R2";
text *uid2="SYSTEM @ D: KR2VMS_V7_R3";
text *pwd="MANAGER";
...
/* 第一个连接 */
if (orlon(&lda[0], &hda[0], uid1, -1, (text *) 0, -1, 0))
{     /* 连接失败 */
      error_handler (&lda[0]);
      exit (EXIT_FAILURE);
}
...
/* 第二个连接 */
if (orlon(&lda[1], &hda[1],uid2, -1. pwd, -1, 0))
{     /* 连接失败 */
      error_handler (&lda[1]);
      exit (EXIT_FAILURE);
}
......
```

当程序发出 orlon 调用之后,可使用 ologof 调用提交事务和切断连接。

LDA 的返回码字段指示 orlon 调用的结果,0 返回码指示成功的连接。

§3.3 打开光标

为了处理SQL语句，在连接后，首先必须打开光标。可通过调用oopen函数来打开一个光标。打开光标必须在分析一个SQL语句之前完成。

下面说明oopen函数的功能和用法：

(1) 功能

该函数打开一个指定的光标。

(2) 调用格式

oopen (struct cda_def *cursor, struct lda_def *lda,<text *dbn>,<sword dbnl>,
<sword arsize>,<text *uid>,<sword uidl>);

(3) 参数说明

cursor：是指向程序内的CDA的指针。

lda：是指向LDA的指针，该LDA是orlon调用所指定的那个LDA。

dbn ：此参数是为与ORACLE版本2兼容而设置的。在比它高的版本中，应指定为0。

dbnl：该参数是为与ORACLE版本2兼容而设置的，在比它高的版本中，应指定为-1。

arsize：该参数在ORACLE7中不使用。

uid：是指向字符串的指针，该串包含用户标识名和口令，用户口令与用户标识之间用斜杠"/"隔开。

uidl：是uid所指串的长度，如果该串以Null结尾，则该参数可省略。

注意：上述参数除cursor为OUT方式、lda为IN/OUT方式外，其余均为IN方式。

(4) 使用说明

oopen把光标数据区与ORACLE服务器中的数据区相联系。ORACLE使用这些数据区来保存SQL语句处理的状态信息。当ORACLE处理SQL语句时，有关错误和警告条件的状态及其它信息(如函数码)被存放到光标数据区内。一个程序同时能有多个光标在活动。

oparse分析一个SQL语句，并把它与光标相联系。在OCI函数中，总是通过使用光标来引用SQL语句。

CDA中的返回码字段表示oopen的执行状态。0表示oopen调用成功。

(5) 相关函数

olon、orlon、oparse。

§3.4 分析SQL语句

在执行任何有效SQL语句或PL/SQL无名块之前，必须先把SQL语句传递给ORACLE，和对它进行分析。分析的任务是进行语法、语义检查，选择最佳存取路径等，以备执行。对于数据定义语言语句，分析步还可完成其执行。OCI程序可通过调用oparse或osql3来完成上述工作。分析必须在连接和打开光标之后进行。下面说明oparse函数的功能和用法：

(1) 功能

该函数把SQL语句传给ORACLE，并对它进行分析和把分析的表示式存放在ORACLE的共享SQL高速缓存中；同时把已命名的光标与该语句相联系。这样后面的OCI调用就可使用光标名来引用该SQL语句。

(2) 调用格式

oparse (struct cda_def * cursor, text * sqlstmt,
[sb4 sqll, sword defflg, ub4 lngflg);

(3) 参数说明

cursor:是指向 CDA 的指针,该 CDA 是在 oopen 调用中所指定的那个 CDA。

sqlstmt:是指向包含 SQL 语句串的指针。

sqll:指出 SQL 语句的长度,如果由 sqlstmt 所指向的 SQL 语句串是 Null 结束,则该参数可省略。

defflg:如果是非 0,且该应用以延迟方式被连接,则 SQL 语句被延期,直至 odsc、odescr、oexec、oexn 或 oexfet 调用被执行。

注意:如果程序用延期方式连接可选项连接,则结合(obndra、obndrn 或 obndrv)和定义操作也被延期到执行或描述步。建议用户使用延期分析性能(只要有可能)。以改进程序性能,特别是在网络环境下。

lngflg:lngflg 参数决定 ORACLE 如何处理 SQL 语句或 PL/SQL 无名块。为了保障与 ANSI 的严格一致性,ORACLE7 定义了几个与 ORACLE6 有稍微差别的数据类型和操作。表 3--1 表示由参数 lngflg 所影响的 ORACLE6 与 ORACLE7 之间的差别。

表 3-1 lngflg 所影响的 ORACLE6 与 ORACLE7 之间的差别

情况	V6	V7
CHAR 列是定长(包括由 CREATETABLE 语句所建立的)	NO	YES
如果试图把一个 Null 值提取到一个未有相关指示器变量的输出变量中,则发出一个错误。	NO	YES
如果提取值被截断且未有相关的指示器变量,则发出错误。	YES	NO
对定长串,描述(odescr 或 odsc)返回内部数据类型 1。	YES	NO
对定长串,描述(odescr 或 odsc)返回数据类型 96。	n/a	YES

lngflg 参数有三种可能的设置:

0:指定版本 6 的情况。

1:程序相连接的数据库版本指定为正常情况(即版本 6 或 7)。

2:指定 ORACLE7 的情况。此时,如果你未被连接到
ORACLE7 数据库上,则 ORACLE 发出错误:
ORA—1011 cannot use this language type
when talking to V6 database

注意:上述参数除 cursor 为 IN/OUT 方式外,其余均为 IN 方式。

(4) 使用说明

oparse 调用使用其前面打开的光标,当程序需要操作多个活动的 SQL 语句时,能定义

多个并发光标。

语句能是任何有效的SQL语句或PL/SQL无名块。ORACLE分析该语句，并选择一个最佳的存取路径来完成其操作功能。

如果lngflg参数取非0，则该分析能被被延迟到该语句被执行或者到调用odescr来描述该语句。

如果oparse的defflg参数是0的话，则数据定义语言语句在分析时被执行，而不需要进一步的处理；如果defflg参数是非0，就必须调用oexn或oexec执行该SQL语句。所有的数据操纵语言语句和查询都需要进一步的处理。

(5) 相关函数

odescr、odsc、oexec、oexfet、oexn、oopen。

下面的例子是打开一个光标和分析SQL语句的程序段。oparse调用把该SQL语句与光标相联系。

```
...
/* 说明 LDA 和 CDA */
lda_def lda;
cda_def cursor;
/* 指定 SQL 语句 */
text *sql_stmt ="DELETE FROM emp
        WHERE empno=:Emp_number";
...
/* 打开光标 */
oopen (&lda, &cursor, 0, 0, 0, 0, 0);
/* 分析 SQL 语句 */
oparse (&cursor, sql_stmt, -1, 1, 2);
...
```

SQL语句的语法错误码在CDA的返回码字段中返回。如果语句不能被分析，则分析错误的相对位移字段指出错误在SQL语句文本中的位置。

§3.5 结合输入变量的地址

大多数数据操纵语言语句和查询语句在执行时都需要输入数据。为此程序的SQL语句中常含有虚拟变量。对于SQL语句中的每一个虚拟变量，必须调用OCI函数来把程序中的一个实变量地址结合到该虚拟变量上。这样，当执行程序时，ORACLE才能取到放置在该输入(或结合)变量中的数据。结合步只告诉ORACLE结合变量的地址，以及数据类型和长度。

有三个OCI函数能用来把地址结合到虚拟变量上。它们是obndrv、obndrn和obndra。它们必须在调用oparse之后，和调用oexn或oexec之前被调用。

3.5.1 OBNDRN、OBNDRV函数

(1) 功能

函数 obndrn 和 obndrv 把程序变量的地址与 SQL 语句中指定的虚拟变量相结合。

(2) 调用格式

```
obndrn (struct cda_def * cursor, sword sqlvn, ub1 * progv, sword progvl,
        sword ftype, <sword scale> [,ub2 * indp], <text * fmt>,
        <sword fmtl>, <sword fmtt>);
obndrv(struct cda_def * cursor, text * sqlvar [,sword sqlvl], ub1 * progv,
       sword progvl, sword ftype, <sword scale> [,sb2 * indp],
       <text * fmt>, <sword fmtl>, <sword fmtt>);
```

(3) 参数说明

cursor:是一个指向 CDA 的指针,通过 oparse 调用使 CDA 与 SQL 语句相联系。

sqlvar:该参数指向字符串,该字符串包含 SQL 语句中的虚拟变量的名字(包括前缀冒号)。它仅由 obndrv 使用。

sqlvl :它给出字符串 sqivar 的长度,包括前缀冒号。例如虚拟变量 :employee 的长度为 9。如果虚拟变量的名字以 hull 结尾,则该参数可省略(作为-1)。该参数仅供函数 obndrv 使用。

sqlvn:该参数仅由函数 obndrn 使用。它通过编号指出由光标所引用的 SQL 语句中的一个虚拟变量。例如,如果 sqlvn 是一个整型文字串 2,则它指的是 SQL 语句内由:2 标记的所有虚拟变量。

progv:是一个指向程序变量或数组变量的指针。当执行 oexec 或 oexn 时,其值被输入给 ORACLE。当执行 oexfet ,ofen 或 ofetch 时,保存检索的数据。

progvl:是程序变量或数组元素的长度(以字节为单位)。因为,对于有许多不同值的 progv 来说,在连续执行或批调用时,obndrn 或 obndrv 仅被调用一次;所以 progvl 必须包含 progv 的最大长度。

注意:对有些系统,sword 类型的长度是 2 个字节。当结合 LONG VARCHAR 和 LONG VARRAW 缓冲区时,把缓冲区的最大长度限制到 64K 字节。要结合更长的数据类型缓冲区时,则设置 progvl 为-1,并传递 progv 的开始 4 个字节;其内容为实际数据区长度(总缓冲区长-sizeof(sb4))。该值应在调用 obndrn 或 obndrv 之前设置。

ftype:是程序变量的 ORACLE 外部数据类型。当数据输入给 ORACLE 或从 ORACLE 检索时,ORACLE 进行数据内部和外部格式之间的转换。

scale:对于 C 语言,通常不使用。

indp:是一个指向短整型变量(或短整型数组)的指针,该短整型变量用作指示器变量。当作为输入时,如果指示器变量包含负值,则对应的列被设置为 NULL;否则被设置为由 progv 所指向的值。当作为输出时,如果指示器变量在 FETCH 后包含负值,则对应的列包含 NULL。

fmt :对于 C 语言,通常不使用。

fmtl:对于 C 语言,通常不使用。

fmtt:对于 C 语言,通常不使用。

注意：上述参数除 cursor 、progv 、indp 是 IN/OUT 方式外，其余均为 IN 方式。

(4) 使用说明

当程序员是在ORACLE7 环境下开发应用时，建议使用 obndra(而不要用 obndrn 或 obndrv)来把实变量地址与 SQL 语句中的虚拟变量相结合。

如果使用 obndrv 的话，则 SQL 语句中的虚拟变量是由一个冒号后跟一个 SQL 标识符所组成。

如下面的 SQL 语句

```
SELECT ename, sal, comm
    FROM emp
    WHERE deptno=:dept AND comm>:min_com
```

有两个虚拟变量，:dept 和 :min_com。

如果使用 obndrn ，则 SQL 语句中的虚拟变量是由一个冒号后跟一个范围界于 1 至 255 的整型数字串所组成。SQL 语句

```
SELECT ename, sal, comm
    FROM emp
    WHERE deptno=:2 AND comm>:1
```

有两个虚拟变量：1 和：2。

把上述第一个 SQL 语句中的 dept 虚拟变量与程序变量 dept_number 相结合的 obndrv 调用是：

```
#define INT 3 /* 整型数的外部数据类型码 */
cda_def cursor;
sword dept_number ,mininum_comm;
...
obndrv(&cursor,":dept", -1, (ub1 *) &dept_number,
    (sword) sizeof(sword), INT -1, (sb2 *) 0, (text *) 0, -1, -1);
```

因为文字串"detpt"是以 Null 结束的串，所以不需要 sqlvl 参数；可把它指定为－1。有些参数是可选的，例如指向指示器变量的指针 indp 等，在该例中不使用它们，因此把它们指定为 0，强制为 sb2 指针。另外也不使用 fmt，故也当作 null 指针。

如果使用 obndrn ，则参数 sqlvn 通过数字来标识虚拟变量。如果 sqlvn 被设置为 1，则程序变量被结合到虚拟变量：1 上。例如，调用 obndrn 把程序变量 mininum_com 结合到上述第二个 SQL 语句中的虚拟变量：2 上的方法如下：

```
obndrn(&cursor, 2, (ub1 *) &mininum_comm, (sword) sizeof(sword), INT, -1,
    (sb2 *) 0, (text *) 0, -1, -1);
```

其中虚拟变量：2 是通过在参数 sqlvn 中传递值 2 来表示的。sqlvn 可以是变量及文字串。不能在 PL/SQL 块中使用 obndrn 来把程序变量与虚拟变量相结合，因为 PL/SQL 不识别数字串形式的虚拟变量。在 PL/SQL 块内部总是使用 obndra (或 obndrv)，且用命名的虚拟变量。

obndrn 或 obndrv 函数必须在调用 oparse 分析 SQL 语句之后和调用 oexn，oexec 或 oexfet 执行它之前被调用。

一旦执行了程序变量的结合，就能通过改变程序变量的值而不用再结合就可再次执行该 SQL 语句。例如，如果已把 dept_number 的地址与虚拟变量:"dept"相结合，则当执行上述的 SQL 语句时，若想使用 new_dept_num(数据类型相同)，就必须再次调用 obndrv 来把新的程序变量与该虚拟变量相结合。

但是如果改变了变量的类型或长度时，就必须在再次执行之前重新分析和结合。

在调用 odescr 之后，不应使用 obndrv 和 obndrn，如果使用的话，就必须首先分析，而后再结合所有的变量。在结合的时候，ORACLE 存储程序变量的地址。如果同一个虚拟变量在 SQL 语句中出现多次，则对 obndrn 或 obndrv 的一次调用就把所有出现的虚拟变量与实(结合)变量相结合了。

在 CDA 的返回代码字段中返回结合的完成状态码。正确完成时返回 0。

(5) 相关函数

odescr、oexec、oexfet、oexn、oparse.

3.5.2 OBNDRA 函数

(1) 功能

obndra 把程序中的一个实际变量或数组的地址与 SQL 语句或 PL/SQL 块中的虚拟变量相结合。

(2) 调用格式

```
obndra(struct cda_def  * cursor, text * sqlvar [, sword sqlvl], ub1 * progv,
       sword progvl, sword ftype, <sword scale> [,sb2  * indp]
       [,ub2 *  alen] [,ub2 * arcode] [,ub4 * maxsiz] [,ub4  * cursiz],
       <text * fmt>, <sword fmtl>,<sword fmtt>);
```

(3) 参数说明

cursor:是指向 CDA 的指针，该 CDA 通过 oparse 调用建立与 SQL 语句相联系。

sqlvar:它是指向字符串的指针，该串包含 SQL 语句中的虚拟变量名(包括前面的冒号)。

sqlvl:是 sqlvar 所指向的字符串的长度，其中包括前面的冒号，如虚拟变量名:employee 的长度是 9。如果变量名是 Null 终结时可省略该参数。

progv:是指向一个或一组程序变量的指针，当执行 oexec，oexn 或 oexfet 时，从这些变量取得数据，或把检索的数据存入这些变量中。

progvl:是程序变量或数组元素的长度(以字节为单位)，因为在连续执行调用时，对于许多不同的 progv 值，只可能调用一次 obndra ，所以 progvl 必须包含 progv 最大长度。

ftype:是用户程序变量的外部数据类型。在把它与 SQL 语句中的虚拟变量相结合之前，ORACLE 把它的类型从外部转换为内部的格式。

scale:C 中一般不用该参数，通常把它设置为 -1。

indp :是指向一个指示器参数、或一组指示器变量 (如果 progv 是一个数组时) 的指针。当是数组时，必须包含与 progv 有相同元素个数。

alen:是指向数组元素的指针，该数组元素包含数据的长度。这是结合变量元素的有效

长度。例如，如果 progv 参数被说明为：text arr[5][20]；那么 alen 至少应该是有 5 个元素的数组。该数组的最大可使用尺寸由参数 maxsiz 确定。如果 arr 是 IN 参数，则 alen 中的每一个元素在执行调用前，应该被设置为数组 arr 的 对应元素中的数据长度(本例中≤20)。如果 arr 是 OUT 参数，则在 PL/SQL 块执行后，返回数据的长度出现在 alen 中。一旦用 obndra 执行了该结合，就可通过改变 alen 所指向的数组元素的值来改变结合变量的数据长度，而不需再结合；但长度不能大于 alen 中指定的长度。

arcode：该参数指向这样一个数组，该数组包含执行调用结合变量的错误码。在 arcode 中返回的错误码指出：progv 中的数据被截短，或者 SELECT（或 PL/SQL FETCH）时出现 Null。例如，ORA－01405 或 ORA－01406。如果 obndra 正被用于结合数组的元素(即 maxsiz 大于 1)，那么 arcode 必须也指向一个至少等尺寸的数组。

maxsiz：是被结合数组的最大维数，其值的范围为 1～32512，但数组的最大维数依赖于数据类型。最大数组维数是 32512。如果 obndra 被用于结合一个标量，则把该参数设置为 0，这意味着是一个元素的数组。

cursiz：是指向数组中元素实际个数的指针。如果 progv 是一个 IN 参数，则把 cursiz 设置为要被结合的数组的维数。如果 progv 是 OUT 参数，则 SQL 语句或 PL/SQL 块执行后，在 progv 数组中返回有效的数组元素数。如果 obndra 被用于结合一个标量，则把该参数设置为 null 指针((ub4 *)0)。

fmt：在 C 中通常不使用。

fmtl：在 C 中通常不使用。

fmtt：在 C 中通常不使用。

注意：上述参数 cursor、cursiz、alen、progv 和 indp 是 IN/OUT 方式；arcode 为 OUT 方式；其余为 IN 方式。

(4) 使用说明

能用 obndra 替换 obndrv 来把程序的标量变量与 SQL 语句或 PL/SQL 块中的虚拟变量相结合。能用 obndra 函数中的 alen 参数来改变被结合变量的大小，而不用再结合该变量。

若要把程序中的数组结合到 PL/SQL 表上时，则必须使用 obndra，因为该函数提供了一些辅助参数，使用户能够控制表的最大尺寸，和在块执行后能够取回当前表的尺寸。

obndra 函数必须在调用 oparse 之后，和在调用 oexn 或 oexec 之前被调用。

一旦结合了一个程序变量，就能通过改变该变量的值和长度，而不用再结合就可再次执行该 SQL 语句或 PL/SQL 块。

但是，如果要改变变量的类型，就必须在分析该语句或块之后，和执行之前，再次结合该变量。

(5) 相关函数

obndrv、oexec、oexn。

3.5.3 应用举例

下面是一个短小而完整的程序，此例说明怎样使用 obndra 才能使程序中的数组与 PL/

SQL 过程中的表相结合。

例 3.2 obndra 应用

```
/*************************************
 *    说明:
 *        该例说明怎样使用 obndra 使程序中的数组与 PL/SQL 过程中的表
 *        相结合。其操作过程是:
 *          连接 ORACLE
 *          打开光标
 *          删除原另件表和建新另件表
 *          建包装及包装体
 *          分析 PL/SQL 无名块(该块调用存储过程)
 *          把 C 数组结合到 PL/SQL 表上
 *          执行该 PL/SQL 块(执行存储过程),向另件表插入一组数据
 *          关闭光标
 *          结束事务,退出 ORACLE
 *************************************/
#include <stdio.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
/* 说明 CDA 和 LDA */
cda_def cda;
lda_def lda;
/* 初始化删表、建表变量 */
text *dt=(text *)"DROP TABLE part_nos";
text *ct=(text *)"CREATE TABLE part_nos(partno NUMBER,
                  description VARCHAR2(20))";
/*初始化建包装语句变量 */
text *cp=(text *)"\
    CREATE OR REPLACE PACKAGE update_parts AS\
        TYPE part_number IS TABLE OF part_nos.partno%TYPE\
            INDEX BY BINARY_INTEGER;\
        TYPE part_description IS TABLE OF part_nos.\
                description%TYPE\
            INDEX BY BINARY_INTEGER;\
        PROCEDURE add_parts (n        IN  INTEGER,\
                             descrip  IN part_description,\
                             partno   IN  part_number;\
    END update_parts;";
```

```
text *cb=(text *) "\
    CREATE OR REPLACE PACKAGE BODY update_parts AS\
        PROCEDURE add_parts (n       IN  TNTEGER,\
                             descrip IN  part_description,\
                             partno  IN  part_number) is\
        BEGIN\
            FOR i IN 1..n Loop\
              INSERT INTO part_nos\
                VALUES(partno(i),descrip(i));\
            END LOOP;\
          END add_parts;\
        END update_parts";

#define DESC_LEN                20
#define MAX_TABLE_SIZE          1200
/* PL/SQL 无名块 */
text *pl_sql_block=(text *) "\
      BEGIN\
          update_parts.add_parts(3,:description,:partno);\
      END;";
text descrip[3][20]={"Frammis","Widget","Thingie"};
sword numbers[]={12125,23169,12126};

ub2 descrip_alen[3]={DESC_LEN, DESC_LEN,DESC_LEN };
ub2 descrip_rc[3];
ub4 descrip_cs=(ub4) 3;
ub2 descrip_indp[3];

ub2 num_alen[3]={(ub2)sizeof(sword),
                 (ub2) sizeof(sword),
                 (ub2) sizeof(sword)};
ub2 num_rc[3];
ub4 num_cs=(ub4) 3;
ub2 num_indp[3];

dvoid oci_error(void);

main()
{
```

```
/* 连接 ORACLE */
printf("Connecting to ORACLE...");
if (olon(&lda,"scott/tiger",-1,NULL,-1,-1))
{ /* 连接失败 */
   printf("Cannot logon as scott/tinger. Exiting...\n");
   exit(1);
}
/* 打开光标 */
if (oopen(&cda, &lda, NULL, -1, -1, NULL, -1))
{ /* 打开光标失败 */
   printf("Cannot open cursor, exiting…\n");
   exit(1);
}
/* 删除表 */
printf("\nDropping table...");
/* 分析和执行 SQL 语句(DROP TABLE) */
if (oparse(&cda,dt,-1,0,2))
    if (cda.rc!=942)
        oci_error();
/* 分析和执行 SQL 语句(CREATE TABLE ) */
printf("\ncreating table…");
if(oparse(&cda, ct, -1, 0, 2))
    oci_error();
/* 分析和执行建包装语句( CREATE OR REPLACE PACKAGE) */
printf("\nCreating package...");
if (oparse(&cda,cp, -1, 0, 2))
    oci_error();
if(oexec(&cda))
    oci_error();

/* 分析和执行建包装体语句( CREATE OR REPLACE PACKAGE BODY)
 */
printf("\nCreating package body...");
if (oparse (&cda,cb,-1,0,2))
    oci_error();
if (oexec(&cda))
    oci_error();

/* 分析调用存储过程的 PL/SQL 无名块 */
```

```
printf("\nParsing PL/SQL block...");
if (oparse(&cda,pl_sql_block,-1,0,2))
    oci_error();
/* 把 C 数组结合到 PL/SQL 表上 */
printf("\nBinding arrays...");
if (obndra(&cda,
           (text *)":description",
           -1,
           (ub1 *) descrip,
           DESC_LEN,
           VARCHAR2_TYPE,
           -1,
           descrip_indp,
           descrip_alen,
           descrip_rc,
           (ub4)MAX_TABLE_SIZE,
           &descrip_cs,
           (text *)0,
           -1,
           -1))
    oci_error();
if (obndra(&cda,
           (text *)":partno",
           -1,
           (ub1 *) numbers,
           (sword) sizeof (sword),
           INT_TYPE,
           -1,
           num_indp,
           num_alen,
           num_rc,
           (ub4) MAX_TABLE_SIZE,
           &num_cs,
           (text *)0,
           -1,
           -1))
    oci_error();
/* 执行该块 */
```

```
        printf("\nExecuting block...");
        if (oexec(&cda))
            oci_error();
        printf("\n");
        /* 关闭光标 */
        if (oclose(&cda))
        {
            printf("Error closing cursor!\n");
            return -1;
        }

        /* 结束事务,退出 ORACLE */
        if(ologof(&lda))
        {
            printf("Error logging off!\n");
            return -1;
        }
        exit(1);
}

/* OCI 错误处理 */
dvoid
oci_error(void)
{
    text msg[600];
    sword rv;
    /* 取错误信息 */
    rv=oerhms(&lda,cda.rc,msg,600);
    /* 显示错误码和错误信息 */
    printf("\n\n%.*s",rv,msg);
    /* 显示发生错误的 OCI 函数 */
    printf("processing OCI function %s\n",oci_func_tab[cda.fc]);
    /* 关闭光标 */
    if (oclose(&cda))
        printf("Error closing cursor!\n");
    /* 结束事务,退出 ORACLE */
    if (ologof(&lda))
        printf("Error logging off!\n");
    exit(1);
```

}

§3.6 描述选择表项和 PL/SQL 过程参数

如果 SQL 语句是查询语句，则在执行该语句之前还必须知道有关选择表项的信息，特别是对于动态查询语句更是如此。因为在动态情况下，程序预先还不知道选择表项的数据类型、长度等有关信息。

例如，用户可以输入这样一个 SELECT 语句：

```
SELECT * FROM emp
```

其中缺少表 emp 的列信息。

能用 odescr(描述)函数来获得这方面的信息。

同样，为了执行存储在 ORACLE 库中的 PL/SQL 过程，也必须知道有关 PL/SQL 过程的信息。为此，应调用 odessp 函数来描述它，以获得存储过程(或函数)的信息。

3.6.1 ODESCR 函数

(1) 功能

该函数描述 SELECT 语句中的选择表项，并返回它的内部数据类型和长度等信息。

(2) 调用格式

odescr (struct cda_def * cursor, sword pos, sb4 * dbsize [,sb2 * dbtype] [,sb1 * cbuf] [,sb4 * cbuf1] [,sb4 * dsize] [,sb2 * prec] [,sb2 * scale] [,sb2 * nullok]);

(3) 参考说明

cursor：是一个指向 CDA 的指针。odescr 函数使用 cursor 来引用特定的 SQL 查询语句，该语句已由前面的 oparse(或 osql3)调用传递给 ORACLE。

pos：是 SQL 查询语句中选择表项的位置索引。每一项都用位置索引来引用，第一个选择表项(即最左一项)的位置索引从1开始。如果你指定的位置索引大于选择表中的项数，或小于1，则 odescr 在 CDA 的返回码字段返回“variable not in select_list”错误。

dbsize：是指向列的最大长度的指针。在 dbsize 中返回的值如表3-2所示：

表3-2 dbsize 中的返回值

oracle 列的类型	值
CHAR,VARCHAR2,RAW	表中列的长度
NUMBER	22(内部长度)
DATE	7(内部长度)
LONG LONGRAW	0
ROWID	(系统依赖)
返回数据类型1的函数	
(如 TO_CHAR())	和 dsize 参数相同

dbtype：存放选择表项的内部数据类型码。对于 CHAR 项（包括选择表中的文字串），返回的数据类型码可能依赖于你如何分析 SQL 语句。如果你使用 osql3或使用设置为0的 lngflg 参数的 oparse、或者在 ORACLE 版本6的环境下使用设置为1的 lngflg 参数的 oparse 的话，CHAR 项返回数据类型码1，否则 dbtype 返回96。选择表中的 USER 函数总是返回数据类型代码1。

cbuf：保存选择表项的名字，即列名或表达式的字符串。程序必须分配足够长的串缓冲区，以保存项名。

cbufl：指出 cbuf 的字节数(长度)。该参数必须在调用 odescr 之 前设置。如果 cbufl 没被指定(即作为0传递)，那么不返回选择表项名。如果选择表项名比 cbufl 长，则名字被截断。在从 odescr 返回时，cbufl 包含返回串的字节长度。

dsize：如果把选择表项作为字符串返回，则 dsize 保存选择表项的最大显示长度。当用函数(象 SUBSTR 或 TO_CHAR)修改列的表达式时，dsize 参数特别有用。

prec：返回数字选择表项的精度，精度是一个数的数字总位数。如果不需要精度值，就把该参数指定为0。

scale：是一个指向 short 的指针，它返回数字型选择表项的定标。如果不需要定标值，就把该参数作为0。对于版本6的 RDBMS，odescr 返回真正的定点数的定标和精度。对于浮点数，返回0精度和定标。如下所示：

SQL 数据类型	精度	定标
NUMBER(P)	P	0
NUMBER(P,S)	P	S
NUMBER	0	0
FLOAT (N)	0	0

对于 ORACLE7，SQL 类型 REAL，DOUBLE PRECISION ，FLOAT 和 FLOAT (N)返回实际的精度和－127的定标。

nullok：是指向 short 的指针，如果列不允许 Null 值，则返回0，如果允许 Null 值则返回非0。如果需要选择表项的 Null 状态，则把该参数指定为0。

注意：cursor 和 cbufl 为 IN/OUT 方式，pos 为 IN 方式，其它为 OUT 方式。

(4) 详细说明

odescr 函数替换较老的 odsc 函数。在分析(用 oparse 或 osql3)过 SQL 语句和结合所有的输入变量之后，才调用该函数。odescr 返回 SELECT 语句中选择表项的如下一些信息：

·最大尺寸(dbsize)
·内部数据类型代码(dbtype)
·列名(cbuf)
·列名的长度(cbufl)
·最大显示尺寸(dsize)
·数字项的精度(prec)
·数字的定标(scale)
·列中是否允许 Null 值(nullok)

因为描述操作(odescr)和结合操作(obndrn 或 obndrv)所返回的结果之间相互存在依赖性。因此,如果 SELECT 语句中有虚拟变量,而且又使用 odescr 来获得选择表项的尺寸或类型的话,应该在描述之前做结合操作。如果在完成描述之后需要重新结合输入变量,就必须在重新结合之前再分析 SQL 语句。注意,重新结合操作可能会改变选择表项的返回结果。

odescr 函数对于 SQL 查询特别有用,因为,其选择表项的个数、数据类型和尺寸,在运行之前可能还不知道。

CDA 的返回码字段指出 odescr 调用的成功(0)或失败(非0)。

odescr 使用位置索引来引用 SQL 查询语句中的选择表项。例如,SQL 语句

SELECT ename ,sal FROM emp WHERE sal>:min_sal

包含两个选择表项:ename 和 sal。它们的位置索引:sal 是2,ename 是1。

为了处理动态选择表项,应在循环中调用 odescr。在循环的开始,先把索引变量设置为1,然后增值它,在每次重复时进行描述,直至“Variabhe not in list”错误在 CDA 的返回码字段返回。

(5) 相关函数

odefin、oparse、osql3.

例3.3 ODESCR 函数的应用

```
/*********************************
 *    说明:
 *        该程序说明怎样描述选择表项。程序要求用户在运行时输入
 *        SELECT 语句,输出每一个选择项的名字、名字的长度和
 *        数据类型。
 *********************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>

#define NPOS 13

main()
{
    /* 说明 CDA 和 LDA */
    cda_def cda;
    lda_def lda;
    text sql_statement[256];
    sword i ,pos ;
    text cbuf[NPOS][20];
```

```
sword dbsize[NPOS] ,cbufl[NPOS],dsize[NPOS];
sb2 dbtype[NPOS] ,prec[NPOS],scale[NPOS],nullok[NPOS];
dvoid oci_error(lda_def * ,cda_def * );
/* 连接 ORACLE */
if (olon (&lda ,"scott/tiger",-1,0.-1,-1))
{    /* 连接失败 */
     printf("Cannot connect as scott. Exiting ...\n");
     exit(1);
}
/* 打开光标 */
if(oopen(&cda. &lda,0,-1,-1,0,-1))
{    /* 打开失败 */
     oci_error(&lda, &cda);
     exit(1);
}
for (;;)
{    /* 输入 SQL 语句 */
     printf("\nEnter SQL statement:\n");
     gets(sql_statement);
     if (strncmp(sql_statement,"exit",4)!=0) break;
     /* 分析该语句 */
     if (osql(&cda,sql_statement,-1,0,0,))
     {    /* 分析错误 */
          oci_error(&lda,&cda);
          continue;
     }
     for(pos=1;pos<=NPOS ;pos++
     {    /* 描述选择表项目 */
          cbufl[pos]=sizeof(cbuf[pos]);
          if (odescr(&cda,pos ,&dbsize[pos],&dbtype[pos],
               &cbuf[pos],&cbufi[pos],&dsize[pos],
               &prec[pos],&scale[pos],&nullok[pos]))
          {
               if(cda.rc==1007)
               break;
               oci_error(&lda,&cda);
               continue;
          }
     }
```

```
            /* 显示选择表项的总数和名字,列的长度和数据类型代码 */
            printf("\nThere were %d select_list items.\n"-pos);
            printf("Item name Length Datatype\n");
            printf("-----------------\n");
            for (i=1;i<=pos;i++)
            {
                printf("%*.*s"cbufl[i],cbuf[i]);
                printf("%*c"25 -cbufl[i],'');
                printf("%*6d %8d\n."cbufl[i] dbtype[i]);
            }
        }
        /* 关闭光标,结束事务,退出 ORACLE。*/
        oclose(&cda);
        ologof(&lda);
        exit(EXIT_SUCCESS);
}
/* 错误处理 */
dvoid
oci_error(Lda_Def *lda,Cda_Def *cda)
{
        text msg[512];

        printf("\n-ORACLE ERROR-\n");
        oerhms(lda,cda->rc,msg,(int)sizeof(msg));
        printf("%s",msg);
        if (cda->fcfc!=0)
            printf("processing OCI function %s\n",
                    oci-func-tab[cda->fc]);
}
```

3.6.2 ODESSP 函数

(1) 功能

该函数用于描述存储在 ORACLE 数据库中的 PL/SQL 过程或函数的参数。

(2) 调用格式

```
odessp (struct lda_def *lda, text *objnam, size_t onlen, ub1 *rsv1, size_t rsv1ln,
        ub1 *rsv2, size_t rsv2ln, ub2 *ovrld, ub2 *pos, ub2 * level, text **
        argnm, ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1 *mode, ub4 *dtsiz,
        ub2 *prec, sb2 *scale, ub1 *radix, ub4 *spare,ub4 *arrsiz);
```

(3) 参数说明

lda：是指向 LDA 的指针

objnam：是过程或函数名，包括可选模式和包装名。引号括住的名字被接收。也接收同义词，并被翻译。串能由 Null 终结。如果不是 Null 终结时，实际长度必须用 onlen 参数传递。

onlen：是 objnam 参数的字节长度。如果该参数以 Null 终结，则作－1传递。否则传递实际长度。

rsv1：留待 ORACLE 将来使用。

rsv1ln：留待 ORACLE 将来使用。

rsv2：留待 ORACLE 将来使用。

rsv2ln：留待 ORACLE 将来使用。

ovrld：是一个指明过程(或函数)是否过载的数组。如果不过载，则返回0。对于 n 个名字过载，过载过程返回1…n。

pos：指明返回参数在过程参数表中的位置。表中的第一个参数(最左一个参数)位置是1。当 POS 返回0时指明函数返回类型是所描述的。

level ：对于标量参数，level 返回0。对于记录参数，除对记录本身返回0外，还要指出其中的每一个参数在记录中的层次，层次从1开始。对数组参数，除数组本身返回0外，还要返回数组元素的层次号1，并描述数组元素的类型。

例如，有这样一个过程，它包含三个标量参数、一个数组(含10个元素)和一个记录(含有在同一层次上的3个标量参数)。对此，就需要用如下的最小数组维数来传递 odessp 数组：对标量参数数组维数为9；3，对数组参数为2，对记录参数为4。

argnam：是指向串数组的指针，该数组返回过程或函数中的每一个参数的名字，串不以 Null 终结。

arnlen：是 argnam 所指的每一个相应参数名的字节长度。

dtype：是每个参数的 ORACLE 数据类型码。对数字类型(如 FLOAT，INTEGER 和 REAL)返回代码2，VARCHAR2返回1，CHAR 返回96。

defsup：指出相应的参数是否有缺省值。返回0，表明没有缺省值。返回1时，表明在过程或函数说明中支持缺省值。

mode：该参数指明相应参数的方式。0表示 IN，1表示 OUT，而2表示 IN/OUT 。

dtsiz：是数据类型的字节长度。字符数据类型返回参数的长度。例如 emp 表包含一个 ename 列。如果过程描述中的一个参数是 emp. ename%TYPE 类型，则对于该参数返回其列的长度10。对 Number 类型返回22。

prec：该参数指出相应参数的精度(如果参数是数字时)。

scale：该参数指出相应参数的定标(如果该数是数字时)。

radix：如果参数是数字，则 radix 指出相应参数的基数。

spare：留待 ORACLE 将来使用。

arrsiz：当调用 odessp 时，传递 OUT 参数的数组元素个数。如果数组不是等长，则必须传递最短数组。当 odessp 返回时，arrsiz 返回数组元素个数 。

注意：上述参数除 lda、arrsiz 为 IN/OUT，objnam 、onlen 为 IN 外，其余为 OUT。

(4) 使用说明

调用该函数来获得存储过程(或函数)及其参数的特性。当调用 odessp 时,应向它传递:

·一个有效的 LDA,并且至少有执行该过程的特权。

·可选择地包括包装名的过程名。包装体不必存在,但要在包装中指定过程。

·过程名的总长度;如果以 Null 终结,则为-1。

如果过程存在,且 lda 参数中指定的连接许可执行该过程,则 odessp 就在一组数组参数中返回每个过程参数的信息。另外,如果它是一个函数,则还返回相应返回类型的信息。OCI 程序必须为 odessp 的所有参数分配数组,而且还必须传递一个参数(arrsiz)以指出数组的大小(如果它们不等,则指出最小数组的大小)。arrsiz 参数返回 odessp 函数返回的每一个数组元素个数。

如果发生错误,odessp 返回非0值,错误号在 LDA 的返回码字段中。可能返回如下的错误码:

-2000: 在 objnam 参数中命名的对象是一个包装,而不是过程或函数。

-2001: 在 objnam 中命名的函数过程在命名的包装中不存在。

-2002: 在 objnam 中指定一个数据库链。

ORA-0××××:ORACLE 代码,通常指出 objnam 内过程说明中的语法错误。

当 odessp 成功返回时,OUT 数组参数包含过程或函数参数的描述信息和函数的返回类型。

(5) 相关函数

odescr

例3.4 ODESSP 函数的应用

```
/***********************************
*    说明:
*         作为一个例子,我们考虑 SCOTT 模式中的包装 EMP_RECS。
*         该包装包含两个存储过程和一个存储函数,全命名为
*         GET_SAL_INFO。
***********************************/
/*包装说明*/
create or replace package EMP_RECS as

    procedure get_sal_info(
        name in emp.ename%type,
        salary out emp.sal%type);

    procedure get_sal_info(
        ID_num in emp.empno%type,
        salary out emp.sal%type);

    function get_sal_info(
        name emp.ename%type)return emp.sal%type;
```

```
end EMP_RECS
/* 提供一个足够大的返回数组,以保留值。*/
#define ASIZE 10
/*指出过程名,说明 obessp 的参数。*/
text *objnam="scott.emp_recs.get_sal_info";
ub2 ovrld[ASIZE];      /*返回过载信息 */
ub2 pos[ASIZE];        /* 返回表中的参数位置*/
ub2 level[ASIZE];      /*返回组合参数的层次*/
text *argnm[ASIZE];    /* 返回参数名 */
ub2 arnlen[ASIZE];     /* 返回参数名的长度 */
ub2 dtype[ASIZE];      /* 返回 ORACLE 数据类型码*/
ub1 defsup[ASIZE];     /* 返回"缺省提供"的指示器 */
ub1 mode[ASIZE];       /*返回方式 (IN OUT IN/OUT) */
ub4 dtsize[ASIZE];     /*返回长度 */
sb2 prec[ASIZE];       /* 返回一个数的精度 */
sb2 scale[ASIZE];      /* 返回一个数的定标 */
ub1 radix[ASIZE];      /* 返回一个数的基数 */
ub4 spare[ASIZE];      /*留待 oracle 将来使用。*/
ub4 arrsiz=(ub4) ASIZE;/* 指出 OUT 数组的大小 */
...
/* 调用 odessp 来描述函数 */
if (odessp (&lda,objnam ,-1,(ub1*)0,0,(ub1*)0,0,
    ovrld,pos ,level,argnm ,arnlen ,dtype ,def_sup ,mode,
    dtsize,prec,scale,radix ,spare ,&arrsiz))
{    /* odessp 调用发生错误 */
     handle_error(&lda);
}
/* 显示一些返回值*/
printf ("Overload\tLevel\tpos\tprocName \tDatatype\n");
for (i=0; i<arrsiz;i++)
{
  printf("%d\t%d\t%d\t%.*s\t%d\n"ovrld[i],level[i],pos[i],
        arnlen[i],argnm[i],dtype[i]);
}
...
```

当对 odessp 的调用完成时,返回参数数组按表3-3所示的形式填充。当总共有5个参数和一个函数返回类型时,arrsiz 中返回6。

表3-3 odessp 调用的返回值

	数组元素					
参数	0	1	2	3	4	5
ovrld	1	1	2	2	3	3
pos	1	2	1	2	0	1
level	1	1	1	1	1	1
argnm	name	salary	ID_num	salary	NULL	name
arnlen	4	6	6	6	0	4
dtype	1	2	2	2	2	1
defsup	0	0	0	0	0	0
mode	0	1	0	1	1	0
dtsiz	10	22	22	22	22	10
prec		7	4	7	7	
scale		2	0	2	2	
radix		10	10	10	10	
spare(注1)	n/a	n/a	n/a	n/a	n/a	n/a

注意1：oracle 留作将来使用

§3.7 定义选择表项

为了执行 SELECT 语句，在描述了选择表项以后，还应使用 odefin 例程来把程序中的输出变量(或数组)与 SELECT 语句中的每一个选择表项相联系。下面描述 odefin 函数：

(1) 功能

使用 odefin 函数为 SELECT 语句的选择表项定义输出变量。

(2) 调用格式

```
odefin(struct cda_def *cursor, sword pos, ub1 *buf, sword buf1,
  sword ftype, <sword scale> [,sb2 *indp], <text *fmt>, <sword fmtl>,
  <sword fmtt> [,ub2 *rlen] [,ub2 *rcode]);
```

(3) 参数说明

cursor：是指向 CDA 的指针，该 CDA 在相关的 oparse 或 osql3调用中被指定。

pos：是 SELECT 语句中选择表项的索引。其中第一个选择表项 (最左的一项)的索引为1，第二个为2...。odefin 使用该参数来把输出变量与给定的选择表项相联系。如果指定的索引大于选择表项中的项数或小于1，则 odefin 无效。如果不知道选择表项的个数时，可使用 odescr 例程来确定它。

buf ：是指向程序变量的指针，当 ofetch、ofen 或 oexfet 被执行时，使用它来保存数据。

bufl：是程序变量的字节长度。如果 buf 是一个数组，则是该数组元素的字节长度。注意，在有些系统上 sword 数据类型可能是2字节长。当定义 LONG VARCHAR 或

LONG VARRAW 类型的缓冲区时，因超出最大64K 字节的极限长。因此，应设置bufl 为－1，且把实际的数据区长度在调用 odefin 之前设置在 buf 的开始4字节中。

ftype：这是一个外部数据类型，在把选择表项的值传递给输出变量之前，首先把它转换为该数据类型。

scale：在 C 中一般不用它。

indp：在提取之后，indp 中的值表示所提取的选择表项是否是 Null，被截短，或返回原封不动的列值。如果在提取之后，indp 中的值是－2，则说明输出缓冲区太小，不能保留全部数据，于是输出被截短。能用表达式 *（ub2 *）&indp 来获得该列的长度。在使用 oparse 来分析 SQL 语句时，若未定义该列的指示器参数的话，则返回"fetched column value was truncated"错误。

如果调用 odefin 建立数组提取的话，则 indp 参数必须是数组的地址

fmt：在 C 中一般不用它。

fmtl：在 C 中一般不用它。

fmtt：在 C 中一般不用它。

rlen：是指向 ub2类型的指针，在提取操作完成后，ORACLE 把数据长度放置在所指的对象中(对变长数据类型应加长度字节)。如果 odefin 把数组与选择项相联系时，则 rlen 必须也是数组。在 ofetch，ofen 或 oexfet 操作之后，返回的长度是有效的。

rcode：是一个指向无符号短整型数的指针，它保留提取之后的列返回码。在 rcode 中返回的错误码指出该列的数据被截短或者遇到 Null。例如 ORA—01405或 ORA—01406。如果 odefin 被用于把一个数组与选择表项相联系，则 rcode 也必须是一个相同大小的 ub2数组

上述参数除 cursor 为 IN/OUT 方式外，其余均为 IN 方式。

(4) 使用说明

对于每一个 SELECT 语句中的选择表项，OCI 程序必须调用一次 odefin。对 odefin 的每次调用都把程序中的一个输出变量与查询中的一个选择表项相联系。输出变量必须是标量或串，并且必须与外部数据类型(ftype)兼容。为了与 oexfet 和 ofen 函数一起使用，输出变量能是标量、数组或串地址。

当程序调用 ofetch、ofen 或 oexfet 时，ORACLE 把数据放置在输出变量中。如果不知道 SQL 语句中选择表项的数目或项的内部数据类型长度时，能通过运行 odescr 函数获得它。

必须在调用 osql3或 oparse 来分析 SQL 语句之后和在提取数据之前来调用 odefin。

odefin 使用 SQL 语句中选择表项的位置索引来把输出变量与选择表项相联系。例如，在 SQL 话句

```
SELECT ename,empno,sal FROM emp WHERE sal>:min_sal
```

中，选择表项 sal 是在位置3，empno 是在位置2，而 ename 是在位置1。

调用 odefin 把输出缓冲区与上述语句中的选择表项相联系，其例子如下所示

```
#define INT 3
#define FLOAT 4 /* 定义数据类型代码 */
#define STR 5
#define ENAME_LEN 20
```

```
cda_def cursor; /* 分配一个光标 */
text employee_name[ENAME_LEN];
sword employee_number;
float salary;
sb2 ind_ename,ind_empno,ind_sal;
ub2 retc_ename,retc_empno,retc_sal;
ub2 retl_ename,retl_empno,retl_sal;
...
odefin(&cursor,1,employee_name,ENAME_LEN,STR,
    -1,&ind_ename,0,-1,-1,&retl_ename,&retc_ename);
odefin(&cursor,2,&employee_number,(int)sizeof(int),INT,
    -1 ,&indp_empno,0,-1,-1,&retl_empno,&retc_empno);
odefin(&cursor,3,&salary,(int)sizeof(float),FLOAT,
    -1,&indp_sal,0,-1,-1,&retl_sal,&retc_sal);
```

ORACLE 用 CDA 中的返回码字段提供行级返回码信息。如果需要列级分回码信息，就必须包括可选项参数 rcode(如上例)。在每次提取过程中，ORACLE 对处理的选择表项设置 rcode。该返回参数包括 ORACLE 错误码，并指出是成功完成(0)或是例外条件，如“null item fetched”，“item fetched was truncated”，或其它致命性错误。下面是 rcode 参数中能被返回的一些代码：

代 码	含 义
0	成功
1405	Null 被提取
1406	Ascii 或串缓冲区数据被截短。来自数据库的转换数据不填入缓冲区。检查 indp(如果指定的话)或 rlen 以确定数据原始长度。
1454	指定无效的转换：整数指定的不是1、2或4字节；实数指定的不是4或8字节。
1456	实数溢出。数据库列或表达式转换在机器上发生浮点溢出。
3115	不支持的数据类型

(5) 相关函数

odescr、oexfet、ofen、ofetch、oparse.

§3.8 执行 SQL 语句

如果 SQL 语句是控制语句或数据操纵语言语句(包括 SELECT 语句)，还必须执行该语句。实现执行操作的 OCI 函数有 oexn 和 oexec。

在执行步，往往需要把所有结合变量的值输入给 ORACLE，有两种输入数据的方法。其一是每次执行时提供不同的输入值，其二是使用数组。

3.8.1 OEXEC 函数

(1) 功能

该函数执行与光标相联系的 SQL 语句。

(2) 调用格式

oexec(struct cda_def *cursor);

(3) 参数说明

cursor 是一个指针,它指向在相关的 oparse 或 osql3 调用中指定的 CDA。

(4) 使用说明

在调用 oexec 之前,必须成功地调用 oparse 完成分析 SQL 语句。如果 SQL 语句包含输入虚拟变量的话,还必须调用 obndrv 或 obndrn 来把每一个虚拟变量与程序变量的地址相结合。

对于查询语句,在 oexec 调用之后,程序还必须用 ofen 或 ofetch 来提取查询结果中的每一行。

对于 UPDATE、DELETE 和 INSERT 语句,oexec 执行 SQL 语句,并在 CDA 中设置返回码字段和行处理统计字段。

对于数据定义语言语句不需要调用 oexec 来执行。因该语句在调用 osql3或 oparse(defflg 参数设置为0)时已被执行。

能用 oexn 通过结合标量变量(不是数组)和把 count 参数设置为1来替换 oexec。对于查询,可使用 oexfet 替换 oexec 和 ofen 的联用。

(5) 相关函数

obndrn、obndrv、oexfet、oexn、oparse。

3.8.2 OEXN 函数

(1) 功能

该函数执行一个 SQL 语句。能使用数组变量在一次调用中输入多行数据。

(2) 调用格式

oexn(struct cda_def *cursor ,sword iters ,sword rowoff);

(3) 参数说明

cursor:是 IN/OUT 方式的指针,它指向在相关的 oparse 或 osql3调用中指定的 CDA。

iters:是数组结合变量的总元素数。它不能大于32767,该变量是 IN 方式。

rowoff:是数组结合变量内基于0的相对位移。如果没错误发生,则 oexn 处理(iters－rowoff)个数组元素。该变量也是 IN 方式。

(4) 使用说明

oexn 函数类似于 oexec,但它允许使用数组结合变量进行操作。一般来说它比连续调用 oexec 更快,特别是在网络客户—服务器环境下。

使用 oexn 的模式是先用 obndrv 或 obndrn 把变量与 SQL 语句中的虚拟变量相结合。在调用 oexn 之前,数据必须在结合变量中。

oexn 的完成状态在 CDA 的返回代码字段中指出。如行处理统计不等于 iters,则对数组元素的失败操作行处理统计＋1。

甚至在其中一个数组元素上失败时,只要未发生回滚,就能继续处理该数组的其余元素。这可通过使用行相对位移来从一个数组元素(不是失败的那个元素)开始操作。

在下面的例子中，如果行处理统计在 oexn 完成时是5，则第6行被拒绝。在这种情况下，可用如下形式再次调用 oexn 来继续从第7行开始处理：

```
oexn(&cursor,10,6);
```

(5) 相关函数

oexec、oexfet

例3.5 oexn 函数的用法

```
/**********************************
 *    说明：
 *        该例说明了三个数组，其中第一个包含10个字符串（长度20），第二
 *        个包含10个整数，第三个含10个指示器，；还定义了一个向数据库
 *        插入多行的 SQL 语句。在结合数组之后，程序必须把第一个插入数
 *        据放置在 names[0]、emp_nos[0]中，把第二个插入数据放置
 *        name[1]和 emp_nos[1]中等等（该步在本例中省略）。然后调用
 *        oexn 把数组中的数据扦入到 emp 表中。
 **********************************/

/*定义结合变量的数据类型 */
#define CHRSTR 1
#define INT 3
/*定义光标 */
cda_def cursor;
/* 定义数组 */
text names[10][20]; /* 字符串数组 */
sword emp_nos[10]; /* 整数数组 */
sb2 ind_params[10]; /* 指示器参数数组 */
/* 要执行的 SQL 语句内含两个虚拟输入变量 n 和 e。*/
text *sql_stmt="INSERT INTO EMP (ENAME,EMPNO) VALUES\
            (:n,:e)";
...
/* 分析该语句（延迟方式）*/
oparse(&cursor ,sql_stmt,-1,1,1);
/* 把数组与虚拟输入变量相结合 */
obndrv(&cursor,":n",-1,names ,20,CHRSTR, -1,ind_params,0,-1,-1);
/* EMPNO 不是 NULL，所以不用指示器参数 */
obndrv(&cursor,":e",-1,emp_nos,(int) sizeof(int),
      INT,-1,0,0,-1,-1);
/*设置数据和指示器参数，然后执行该语句，插入数组的值 */
oexn(&cursor,10,0);
......
```

§3.9 提取查询行

对于 SELECT 语句，在定义了输出变量的地址以后，就可通过调用 oexfet 来执行和提取查询的行；或者在执行了 oexec 和 oexn 之后，通过调用 ofen 或 ofetch 函数来提取查询的行。

如果打算用 oexfet 或 ofen 来提取多行的话，就必须确保为选择表项定义的输出变量是数组。

3.9.1 OEXFET 函数

(1) 功能

Oexfet 执行与光标相联系的 SQL 语句。然后提取一行或多行。它也能执行光标删除(与 ocan 调用相同)

(2) 调用格式

oexfet(struct cda_def *cursor ,ub4 nrows, sword cancel ,sword exact);

(3) 参数说明

cursor：是一个指针，它指向在相关的 oparse 或 osql3调用中所指定的 CDA。

nrows：是提取的行数。如果 nrows 大于1，则必须定义一个数组及所需要的指示器变量来保存选择值。如果 nrows 大于指定查询的行数，则 CDA 中的行处理计数字段被设置为实际返回的行数，并返回错误 ORA_01403；no data found

cancel：如果该参数是非0，则光标在提取完成后被删除。这和 ocan 调用有相同的效果，但节省了调用开销。

exact：当 oexfet 被调用时，若该参数是非0，且查询返回的行数与 nrows 中指定的行数不完全相同时，则 oexfet 返回错误，但还返回行。如果查询返回的行数小于 nrows 参数中指定的行数，则 ORACLE 返回错误

ORA－01403：no data found

如果查询返回的行数大于 nrows 参数中指定的数，ORACLE 返回错误：

ORA－01422：Exact fetch returns more than requested number of rows

如果 exact 是非0，则不考虑 cancel 参数的设置，而总是执行光标删除。

注意：上述参数除 Cursor 为 IN/OUT 方式外，其余为 IN 方式。

(4) 使用说明

在调用 oexfet 之前，OCI 程序必须首先调用 oparse 分析 SQL 语句，调用 obndrv 或 obndrn(如果需要时)来结合输入变量，然后调用 odefin 来定义输出变量。

如果 OCI 程序使用延迟方式连接可选项来连接的话，则结合和定义步被延缓到 oexfet 被调用。如果 oparse 用延缓分析标志(defflg)参数为非0来调用，则分析步也被延迟到 oexfet 被调用。延迟执行使程序能用客户与服务器之间的最小信息旅程来完成处理。如果对非查询 DML 语句调用 oexfet 的话，ORACLE 会发出如下错误：

ORA－01002：fetch out of sequence 且该操作失败。

对 ORACLE7数据库系统，若使用设置为1或2的 lngflg 参数的 oparse 函数来分析 SQL 语句时，则比缓冲区大的字符串会被截断，列返回码(rcode)被设置为如下错误：

ORA－01406 fetched column value was truncated 并且指示器参数被设置为项的原始长度，但 oexfet 调用并不返回一个错误指示。如果遇到选择表项是 Null 的列，则与该列相关的列返回码(rcode)被设置为如下错误：

ORA－0145：fetched column value Is null 而指示器参数被设置为－1，oexfet 调用不返回错误。

但是，如果不定义指示器参数而运行程序的话，oexfet 返回1405错误；如果在未定义指示器参数时选择 Null，既使列返回码和返回长度被定义，它也总是发生错误。

oexfet 执行语句和提取行。如果你需要在 oexfet 完成之后来提取额外的行，应使用 ofen 函数。

(5) 相关函数

obndrn obndrv、odefin ofen oparse。

例3.6 OEXFET 函数的应用

```
/*********************************
 *    说明：
 *        该例是返回多行的查询，它调用 oexfet 函数，
 *        并使用了延迟分析、结合、和定义操作。
 *********************************/
/* 定义 LDA 和 CDA */
lda_def lda ;
cda_def cda
/* 给出 SELECT 语句 */
text *sql_statement=
    "SELECT ename, sal FROM emp WHERE deptno=:1";
/* 定义数组 */
float salaries[12000];
text names[12000][20];
sb2 sal_ind[12000],name_ind[12000]; /* 定义指示器变量 */
sword dept_number;
/* 连接到 ORACLE... */
...
/* 打开光标 */
oopen(&cda,&lda,0,-1,-1,0,-1,);
/* 延迟分析 */
oparse(&cda,sql_statement,-1,1,1);
/* 提示输入部门号召 */
printf("Enter department number:");
scanf("%d,"&dept_number);
...
/* 延迟结合 */
```

```
obndrn(&cda,1,&dept_number,(int)sizeof(int),3,-1,
       0,0,-1,-1);
/* 定义输出变量 */
odefin(&cda,2,salaries,(int)sizeof(float),
       4,-1,sal_ind,0,-1,-1); /* FLOAT 的数据类型 4 */
odefin (&cda,1,names,20,1,-1,name_ind,0,-1,-1);
/*检索不超过 12000 个人的工资*/
oexfet(&cda,12000,0,0); /* 不设置 cancel and exact */
...
```

3.9.2 OFEN 函数

(1) 功能

该函数把一行或多行提取到数组变量中。

(2) 调用格式

ofen (struct cda_def *cursor, sword nrows);

(3) 参数说明

cursor:是指向 CDA 的指针,该 CDA 与 SQL 语句相关联,可通过 oparse 或 osql3调用来分析该 SQL 语句。它是 INOUT 方式的参数。

nrows:定义数组的大小,它(=32767,如果是1,则 ofen 的作用类似于 ofetch.。它是 IN 方式的参数。

(4) 使用说明

ofen 与 oftch 相类似,但是 ofen 一次调用能把多行提取到数组变量中,指向数组的指针用 odefin 与 SQL 查询语句中的选择表项相结合。

当在 ORACLE7数据库上运行时,用 oparse 函数分析 SQL 语句(该 oparse 的 lngflg 参数应被设置为1或2)。比相关缓冲区大的字符串被截断,列返回码(rcode)被设置为如下错误:

ORA-01406:fetched column value was truncated 而指示器参数被设置为该项的原始长度。但是 ofen 调用并不返回错误指示。如果遇到一个选择表项是 Null,则与该列相关的列返回码(rcode)被设置为如下错误:

ORA-01405: felched column value is Null 且指示器参数被设置为-1,ofen 调用不返回错误。

但是如果没有定义指示器参数,则 ofen 返回1405错误。

既便是在提取单行时,ORACLE 公司也建议 ORACLE7 OCI 程序使用 oexfet(nrows 参数设置为1),而不是 oexec 和 ofen 混用。当预先不知道查询返回的精确行数时,在 oexfet 之后可使用 ofen 来提取额外的行。

ofen 的完成状态在 CDA 的返回码字段中指出。CDA 的行处理统计字段累计成功处理的行数。

(5) 相关函数

odefin、oexfet、ofetch、oparse。

例3.7 OFEN 函数的应用

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*    说明:
*         该例说明怎样使用 ofen 来提取多行 ,其处理步骤如下:
*         (a) 与 ORACLE 相连接
*         (b) 打开光标
*         (c) 分析 SELECT 语句
*         (d) 定义输出变量
*         (e) 执行该 SQL 语句(SELECT)
*         (f) 用 ofen 来提取行,每次10行,而后显示结果
*         (g) 关闭光标
*         (h) 结束处理,退出 ORACLE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#include <stdio.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ocidem.h>
/* 说明 LDA */
lda_def lda;
main()
{
    /* 说明 CDA */
    static cda_def cda;
    /* 说明数组 */
    static sb2 ind_a[10];
    ub2   rl[10], rc[10];
    static sword empno[10];
    static text names[10][MAX_NAME_LENGTH];
    sword i, n, rows_done;
    /* 与 ORACLE 相连接 */
    if (olon (&lda, "scott/tiger", -1, 0, -1, 0))
    {
        /* 连接失败 */
        printf ("cannot connect to ORACLE as scott/tiger\n");
        exit(1);
    }
    /*打开光标 */
    if (oopen (&cda, &lda, 0, -1, -1, 0, -1))
    {
```

```
        /* 打开光标失败 */
        printf ("cannot open the cursor\n");
        ologof(&lda);
        exit(1);
    }
    /* 分析 SELECT 语句 */
    if (osql3(&cda, "SELECT ENAME,EMPNO FROM EMP", -1))
    {
        分析步发生错误 */
        oci_error(&cda);
        exit(1);
    }
    /* 定义输出变量 */
    if (odefin(&cda, 1, names, MAX_NAME_LENGTH, 5, -1,
            ind_a, 0, -1, -1, rl, rc))
    {
        /* 定义输出变量发生错误 */
        oci_error(&cda);
        exit(1);
    }
    if (odefin(&cda, 2, empno, (int) sizeof(int), 3, -1,
            0, 0, -1, -1, 0, 0))
    {
        oci_error(&cda);
        exit (1);
    }
    /* 执行该 SQL 语句(SELECT) */
    if (oexec(&cda))
    {   /* 执行步发生错误 */
        oci_error(&cda);
        exit(1);
    }
    /* 用 ofen 来提取行,每次10行,而后显示结果 */
    for (rows_done=0 ; ;)
    {
        if (ofen(&cda, 10))
            if (cda.rc !=1403)
            {   /* 提取发生错误 */
                oci_error(&cda);
```

```
                exit (1);
            }
        /* 计算本次提取的行数,它总是(= 10 */
        n=cda.rpc-rows_done;
        /* 计算累计提取的总行数 */
        rows_done+=n;
        /* 显示提取的每一行 */
        for (i=0; i<n; i++)
        {
            if (ind_a[i])
                printf ("%s ", "(null)");
            else
              printf ("%s%*c", names[i],
                  MAX_NAME_LENGTH-r1[i], ' ');
            printf("%10d\n", empno[i]);
        }
        if (cda.rc ==1403) break; /* 没有更多的行 */
        }
        /* 显示查询的总行数量 */
        printf ("%d rows returned \n", cda.rpc);
        /* 关闭光标 */
        if (oclose(&cda))
            exit(1);
        /* 结束处理,退出 ORACLE */
        if (ologof(&lda))
            exit(1);
        exit(0);
}
/* 错误处理函数 */
oci_error(struct cda_def *cda)
{
    static text msg[512];
    sword len;

    len =oerhms(&lda, cda->rc, msg, (int) sizeof(msg));
    printf ("\n-ORACLE ERROR-\n");
    printf ("%.*s\n", len, msg );
    printf ("Processing OCI function %s\n",
        oci_func_tab[cda->rc]);
```

```
    return 0;
}
```

3.9.3 OFETCH 函数

(1) 功能

该函数把查询的行返回给程序，一次返回一行。

(2) 调用格式

```
ofetch(struct cda_def *cursor);
```

(3) 参数说明

该参数是一个指针，它指向与 SQL 语句相关联的 CDA。

(4) 使用说明

每一个查询选择表项的值被放置在由 odefin 调用所标识的缓冲区内。

当在 ORACLE7环境下运行时，使用 oparse 来分析该 SQL 语句，oparse 函数应具有设置为1或2的 lngflg 参数。如果取的字符串相对于存放它的缓冲区来说太长，则被截断，列返回码(rcode)被设置为如下错误：

ORA－01406：fetched column value was truncated 并且指示器参数被设置为提取行的原始长度。但 ofetch 调用并不返回错误指示。

如果对选择表项检索时遇到 Null，则该列的列返回码(rcode)被设置为如下错误：

ORA－01405：fetched column value is NuLL 并且其指示器参数被设置为－1。ofetch 调用并不返回错误。

但是，如果未定义指示器参数，则 ofetch 返回1405错误。

甚至当提取单行时，ORACLE 公司也建议 ORACLE7 OCI 程序使用 oexfet(具有设置为1的 nrows 参数)，而不用 oexec 和 ofetch 的组合。

每一次 ofetch 调用都从满足查询的行集中返回下一行，且 CDA 中的行处理计数增1。

不能再提取前面已提取过的行。如果想提取前面已提取过的行，必须通过再执行 oexec 调用。在最后一行提取后，下一个提取返回“no data found”返回码，此时行处理计数中包含查询返回的总行数。

(5) 相关函数

odefin、odescr、oexec、oexfet、ofen.

例3.8 OFETCH 函数的应用

```
/*************************************
 *    说明：
 *        该例说明如何能对查询语句使用 ofetch 来从查询集中提取数据。
 *        它是前面的 odefin 函数描述中所给定例子的继续。
 *************************************/
int rv;
.......
/* 执行 SQL 语句 */
oexec(&cursor);
```

```
/* 提取查询的每一行 */
for (; ;)
{
    /* 提取一行 */
    if (rv=ofetch(&cursor))
        break;      /* 行提取发生错误 */
    /* 输出提取的行 */
    if (retc_ename[0]==0)
        printf("%*.*s\t",retl_ename, retl_ename[0], employee_name)[0];
    else if (retc_ename[0]==1405)
        printf ("%*.*s\t", retl_ename[0], retl_ename[0], "Null");
    else
        break;

    if (retc_empno[1]==0)
        printf ("%d\t", employee_number);

    /*处理其余项 */
    ...
    printf("\n");
}
/* 提取结束 */
if (rv !=1403)
    errrpt (&cursor);
...
```

3.9.4 OFLNG 函数

(1) 功能

该函数提取 LONG 或 LONG RAW 列的一部分。

(2) 调用格式

```
ofing(struct cda_def *cursor, sword pos,
          ubl * buf, sb4 bufl, sword dtype,
          ub4 * retl, sb4 offset);
```

(3) 参数说明

cursor:是一个指针,它指向在相关的 oparse 或 osq13调用中所指定的 CDA。

pos: 是行中 LONG 类型列的索引位置。第一列位置是1。如果索引位置所对应的列不是 LONG 类型,则返回

Column does not have LONG datatype"

错误。如果不知道该位置,可用 odescr 通过选择表来索引。当在 dtype 参数中返回

LONG 数据类型码(8或24)时,循环索引变量的值(从1开始)就是 LONG 数据类型列的位置。

buf：是指向缓冲区的指针,该缓冲区保存 LONG 类型列的数据部分。

bufl：是 buf 的长度,以字节为单位。

dtype：是与 buf 的数据类型相对应的代码。

retl：返回的字节数。如果多于65535字节,则在该参数中仍返回65535。

offset：在 LONG 类型列中,被提取部分相对于第一字节的相对位移。

注意：除参数 buf 和 retl 为 OUT 方式,cursor 为 IN/OUT 方式外,其余均为 IN 方式。

(4) 使用说明

LONG 和 LONG RAW 列的长度不超过2千兆字节,oflng 函数允许你从当前行的 LONG、或 LONG RAW 表列的任意相对位移处开始提取任意字节数的数据。一个表只能有一个 LONG 或 LONG RAW 列,但是包括连接操作的查询能在它的选择表中包括几个 LONG 数据类型项。POS 参数指出 oflng 调用中引用的 LONG 类型表列的位置(即第几列)。

在调用 oflng 来检索 LONG 类型表列的部分字节之前,必须做一次或多次提取,使光标放置在所要求的行上。

oflng 函数对于非结构化 LONG 或 LONG RAW 列是有用的,因为这种列的数据不能作为一个整体块来操纵。

当调用 oflng 来从 LONG 类型的表列检索多段时,按从低到高的相对位移顺序来检索比从高到低或随机检索更有效。

(5) 相关函数

odescr、oexfet、ofen.

例3.9 OFLNG 的函数应用

```
/*************************************
*     说明：
*          该程序段是从 data_table 表的 data 列(LONG 列)的70000相对
*          位移处开始检索100K 字节。该表有2列,其中第二列是
*          LONG 数据。
*************************************/
#define DB_SIZE 102400
#define FALSE 0
#define TRUE 1

ub1 *data_area;
sb4 offset;
sb2 da_indp, id_no;
ub4 ret_len;
cda_def cda;
...
/* 分配缓冲区 */
```

```
data_area=(ub1 *) malloc (DB_SIZE);
...
/* 延迟分析 SELECT 语句 */
oparse(&cda,"SELECT id_no, data FROM data_table
            WHERE id_no=100", -1, TRUE, 1);
/* 为第一个选择表项, id_no ,定义一个输出变量 ,不用指示器参数 */
odefin(&cda, 1, &id_no, (int) sizeof (int), 3, -1, 0, 0, 0,
       -1, 0, 0);
/* 为第二个选择表项, data,定义一个输出变量 ,不用指示器参数 */
odefin(&cda, 2, data_area, DB_SIZE, 1, -1, &da_indp,
       0, 0, -1, 0, 0);
/* 执行和提取1行,光标现在是在第一行上 */
oexfet(&cda, 1, FALSE, FALSE);
/*从第一行第二列的指定位置(相对位移70000)提取100K 字节 */
oflng(&cda, 2, data_area, DB_SIZE, 1, &ret_len, (sb4) 70000);
......
```

§3.10 数据操纵和提取的控制

在实际应用中,往往要求人为终止某一操作。例如,停止某一 OCI 函数的执行等。为此,OCI 提供了如下两个函数:

·OBREAK

·OCAN

3.10.1 OBREAK 函数

(1) 功能

该函数使任何一个当前正执行的与指定的 LDA 相联系的 OCI 函数立即(非同步)中途夭折,它通常被用于停止一个未完成的长时间运行的操纵或提取。

(2) 调用格式

obreak(struct lda_def *lda);

(3) 参数说明

lda: 该参数是一个指针,它指向在 olon 或 orlon 调用中所指定的 LDA,其方式为 IN。

(4) 使用说明

如果在调用 obreak 时未有 OCI 函数运行,则该函数不起作用。如果在调用 obreak 之后的下一个 OCI 函数调用是提取,则该提取调用将被夭折。

obreak 是在其它 OCI 函数正在运行时能够调用的唯一一个 OCI 函数。当连接操作(olon 或 orlon)正在进行时,不应该使用 obreak,因为此时 LDA 是正处于不确定状态。

obreak 不是对所有的操作系统都适用。

(5) 相关函数

sqllda

例3.10 OBREAK 的应用

```
/*********************************
*    说明：
*         该例说明在 OCI 程序中怎样使用 obreak 来中断一个在
*         六秒钟内未完成的查询。
*         它只能在 DEC VAX/VMS 和 Unix 操作系统下操作。
*         该例必须连接两个任务以便正确操作。
*********************************/
#include <stdio.h>
#include <signal.h>
#include"ocidfn.h"

#define ANSI_C
#include" ocidem.h"
/* 定义 LDA 和 CDA */
lda_def lda;
cda_def cda;
/* 函数说明 */
dvoid sighandler(void);
dvoid (*old_sig)();
dvoid err(void);

main()
{
    /* 变量说明和初始化 */
    text *uid="scott@d:kervms_ora60";
    text *pwd="tiger";
    ub1 hda[256];
    text name[10];

    /* 与 ORACLE 相连接，程需必须连接两个任务，
       所以用 SQL*Net 连接。*/
    if (orlon(&lda,hda,uid,-1,pwd,-1,0))
    {
        /*连接失败 */
        printf("cannot connect as %s\n",uid);
        exit(1);
    }
    /* 打开光标 */
```

```
if(oopen(&cda,&lda ,0,-1,-1,0,-1))
{
    /* 不能打开 */
    printf("cannot open cursor data area\n");
    exit(1)
}
/* 保存指向当前信号处理函数的指针 */
old_sig=signal(SIGALRM,sighandler);
/*分析查询语句 */
if (oparse(&cda,"SELECT ename from emp",-1,0,2))
    err();
/* 为选择表项定义一个输出变量 */
if (odefin(&cda,1,name,sizeof(name),1,-1,(sb2 *)0,(text *)
        0,0,-1,(ub2 *)0,(ub2 *)0))
    err();
/*执行查询语句 */
if(oexec(&cda))
    err();
/* 设置时间,以判断是否超过6秒钟 */
alarm(6);

/* 开始查询 */
for(;;)
{
    /* 提取行 */
    if (ofetch(&cda))
    {
        /* 如果没有数据找到(几乎不会发生,除非 alarm 失败
           或者 emp 表小于6行),则 Break */
        f(cda.rc==1403) break;
        /* 当捕获到一个警报信号和完成 obreak 时,则在此点会
           遇到一个1013错误 */
           err();
    }
    printf("%10.10s\n",name);
    /* 放慢时间已过的查询 */
    sleep(1)
}
fprintf(stderr,"Unexpected termination.\n");
```

```
    err();
}

/* 定义一个新的警报函数来替换该标准的警报处理器 */
dvoid
sighandler(void)
{
    sword rv;

    fprintf(stderr,"Alarm signal has been caught\n");
    /* 调用 obreak()来中断处理中的 SQL 语句。*/
    if(rv=obreak(&lda))
        fprintf(stderr,"Error %d on obreak\n",rv);
    else
        fprintf(stderr,"obreak performed\n");
    }

    /*错误处理函数*/
    dvoid
    err(void)
    {
        text errmsg[512];
        sword n;

        n=oerhms(&lda,cda.rc,errmsg,sizeof(errmsg));
        fprintf(stderr,"\n-ORACLE error-\n%.*s\n",n,errmsg);
        fprintf(stderr,"processing OCI function %s\n",
                oci_func_tab[cda.fc]);
        oclose(&cda);
        ologof(&lda);
        exit(1)
    }
```

3.10.2 OCAN 函数

(1) 功能

该函数用于在提取所要求的行数之后撤消一个查询

(2) 调用格式

```
ocan(struct cda_def *cursor);
```

(3) 参数说明

cursor 是一个指针，方式为 IN/OUT。它指向在 oparse 或 osql3调用中指定的 CDA。

(4) 使用说明

该函数通知 ORACLE：指定光标进程中的操作结束。于是它释放任何与该光标相联系的资源，但保持与其分析共享 SQL 区中的表达式相联系的光标。

例如，如果只需要多行查询中的第一行，就可在第一次 ofetch 操作之后调用 ocan 来通知 ORACLE 程序将不再执行其它的提取操作。

如果使用 oexfet 函数来提取数据的话，则能通过指定 oexfet 的 cancel 参数为非0值来在提取完成之后执行一个有效的 ocan。

(5) 相关函数

oexfet、ofen、ofetch、oparse，osql3.

§3.11 关闭光标

在应用程序结束或打开一组新光标之前，要用 OCLOSE 函数来关闭每一个打开的光标。

(1) 功能

该函数把光标与 ORACLE 服务器中与其相联系的数据区切断，并且释放它所占用的的内存区域。

(2) 调用格式

oclose(struct cda_def *cursor);

(3) 参数说明

cursor 是指针，它指向相关的 oopen 调用中所指定的 CDA。

(4) 详细说明

该函数释放所有通过 oopen、oparse、执行和使用该光标的提取操作所获得的资源。如果该函数失败，则 CDA 的返回码字段包含相应的错误代码。

(5) 相关函数。

oopen oparse

§3.12 事务控制

ORACLE 允许并发处理和共享系统资源，为了保证数据的安全性，必须控制并发性和资源共享。ORACLE 是面向事务的数据库，它要靠事务管理来保证数据的安全性。OCI 提供如下一些函数来实现事务管理：

(1) 事务提交

·OCOM：提交当前事务。

·OCON：允许每一个 SQL 数据操纵语句的自动提交。

·OCOF：禁止每一个 SQL 数据操纵语句的自动提交。

另外，ologof 的成功调用，将自动发一个 COMMIT 语句。

(2) 回滚事务

·OROL：回滚当前事务。

·OOPT：用于设置非致命 ORACLE 错误的回滚可选项或设置等待可选项。

3.12.1 OCOM 函数

(1) 功能

该函数提交和结束当前事务

(2) 调用格式

ocom(struct lda_def *lda);

(3) 参数说明

lda 是一个指针,它指向 olon 或 orlon 连接调用中所指定的 LDA。

(4)使用说明

当前事务是从 olon 或 orlon 调用或者从最近一个 orol 或 ocom 调用后开始,一直延续到发出一个 ocom,orol 或 ologof 调用。如果 ocom 失败,则在 LDA 的返回码字段中指示失败原因。不要把 ocom 调用(提交)与 ocon 调用(接通自动提交)相混淆。

(5) 相关函数

ocon、ologof、olon、orlon、orol.

3.12.2 OCON 函数

(1) 功能

该函数允许每一个 SQL 数据操纵语句在执行后自动提交。

(2) 调用格式

ocon(struct lda_def *lda);

(3) 参数说明

与函数 ocom 相同。

(4) 使用说明

按照 ORACLE 缺省规定,在 OCI 程序的一开始就设置"禁止自动提交"。当自动提交接通时,在执行 SQL 语句之后,返回码字段中的0指示事务已被提交。自动提交比在每一个逻辑事务结束时放置 ocom 调用开销更大,灵活性更小。

如果 ocon 失败,则在 LDA 的返回码字段中指示失败原因。

(5) 相关函数

ocof、ocom、olon、orlon.

3.12.3 OCOF 函数

(1) 功能

该语句禁止每一个 SQL 数据操纵语句的自动提交。

(2) 调用格式

ocof(struct lda_def *lda)

(3) 参数说明

lda 说明同 ocom。

(4) 使用说明

按照 ORACLE 缺省规定,在 OCI 程序的一开始就设置"禁止自动提交"。由于自动提交

对性能有重大影响。因此，当某些特殊情况需要使用函数 ocon 接通自动提交时，就应该使用函数 ocof 来尽快禁止自动提交。如果 ocof 失败，则在 LDA 的返回代码字段中指示失败原因。

(5) 相关函数

ocom、ocon、olon、orlon.

3.12.4 OROL 函数

(1) 功能

该函数回滚当前事务。

(2) 调用格式

orol(struct lda_def *lda);

(3) 参数说明

lda 是指向 LDA 的指针。

(4) 使用说明

事务是一组相关的 SQL 语句。如果 orol 失败，其原因在 LDA 的返回码字段中指出。

(5) 相关函数

ocom、olon、orlon.

3.12.5 OOPT 函数

(1) 功能

当发生非致命 ORACLE 错误时，可用该函数设置回滚可选项。这些非致命错误主要指执行多行 INSERT 和 UPDATE SQL 语句发生的错误。它也被用于设置等待可选项(如所要求的资源不能使用的情况下)。

(2) 调用格式

```
oopt (struct cda_def *cursor, sword rbopt,
    sword waitopt);
```

(3) 参数说明

cursor：是一个 IN/OUT 方式的指针。它指向在相关的 oopen 调用中所使用的 CDA。

rbopt：ORACLE 6或以后的版本不支持该参数。

waitopt：该参数指出是否等待资源。如果这些资源当前不能使用时返回一个错误。如果该可选项被设置为0，则当资源不能用时程序无限期地等待。0是缺省值。如果该可选项被设置为4，则当所要求的资源不能用时，程序将接到一个错误返 回码。它是一个 IN 方式的参数。

(5) 相关函数

oopen

§3.13 切断与 ORACLE 的连接

在程序执行结束之前要调用 ologof 函数来切断与 ORACLE 的连接。对于每一个在 olon 或 orlon 调用中建立的连接，都要调用 ologof 来切段。下面是 ologof 的描述：

(1) 功能

该函数把LDA(登录数据区)与ORACLE 程序全程区断开,并释放ORACLE 用户进程占用的所有ORACLE 资源。

(2) 调用格式

ologof(struct lda_def * lda);

(3) 参数说明

lda 是指向 LDA 的指针,该 LDA 在 olon 或 orlon 调用中指定。

(4) 使用说明

对 ologof 的成功调用,系统自动发一个 COMMIT 语句来实现程序的注销,所有当前打开的光标被关闭。如果程序注销不成功或异常结束,则所有未完成的事务回滚。

如果程序有多个活动的连接,则对于每个活动的 LDA 必须执行一个独立的注销。

如果 ologof 失败,则在 LDA 的返回码字段中指出失败理由。

(5) 相关函数

orlon、olon.

§3.14 错误处理

错误处理是应用程序的重要组成部分,OCI 提供 oerhms 函数来为程序提供有关的错误信息。下面是该函数的描述:

(1) 功能:

该函数返回 ORACLE 的错误信息文本,并在参量 rcode 中给出错误代码。

(2) 调用格式

oerhms(struct_lda * lda, sb2 rcode, text * buf, sword bufsize);

(3) 参数说明

lda: 是指向 LDA 的指针。

rcode:该参数包含 ORACLE 错误号,即 LDA 或 CDA 的返回码。

buf:是指向缓冲区的指针,该缓冲区保存返回的错误信息文本。信息文本以 Null 终结。

bufsize:是缓冲区的字节长度。缓冲区的最大长度基本上未限制,但通常不需要大于1000字节。

(4) 使用说明

当调用 oerhms 函数时,第一个参数应传递活动的 LDA 地址,以便用 LDA 来检索错误信息。

当该调用成功完成时,它并不返回0,而是返回 buf 中的字符数。错误信息文本在 buf 中以 Null 结尾。

当用 oerhms 返回 PL/SQL 块中的错误信息时,(其中错误码在6550至6599之间),要确保分配一个大的 buf。因为可能返回几个信息。在大多数情况下,有1000字节就能满足。

下面的例子说明怎样从指定的 ORACLE 实例中获得错误信息。

```
lda_def lda[2];      /* 两个独立的连接 */
cda_def cda;
sword n_chars;       /* 保存错误信息长度 */
```

```
text msgbuf[512];     /* 缓冲区,保存错误信息 */
...
/* 当在第二个连接上发生错误时 */
n_chars =oerhms (&lda[1],cda.rc, msgbuf,
                (int)sizeof (msgbuf));
```

(5) 相关函数

oermsg orlon.

§3.15 在PRO*C程序中嵌入OCI函数调用

在PRO*C程序中允许嵌入OCI函数调用,即允许用C语言、OCI函数和嵌入SQL或PL/SQL块来混合编程。为此预编译系统提供了一个sqllda函数。下面描述该函数及其使用:

(1) 功能

sqllda函数是为在PRO*C程序中嵌入OCI函数调用而提供的。它是预编译程序库SQLLIB的一个成员。一个指向LDA的指针参数被传递给sqllda。返回结果填充在LDA的有关字段上。

(2) 调用格式

```
dvoid sqllda(struct lda_def *lda);
```

(3) 参数说明

lda是指向LDA的指针,在调用之前必须分配LDA数据区。

(4) 使用说明

如果程序内包含预编译程序的语句和OCI函数调用,则不能用orlon或olon来登录到ORACLE上。而必须用如下的嵌入SQL语句来登录:

```
EXEC SQL CONNECT…
```

但是,许多OCI函数都要求使用一个有效的LDA。sqllda函数在调用时获得一个LDA,并使用最近执行的CONNECT语句的连接信息来填充LDA。因此,在编码时,应该把sqllda函数逻辑地放在EXEC SQL CONNECT…语句之后调用。

sqllda不直接返回值,如果调用sqllda而没有有效的连接,则在lda参数的返回代码段中返回错误:

```
ORA-01012: not logged on
```

例3.11 sqllda函数的应用

```
/*********************************************
*    说明:
*        该例说明怎样在预编译程序—OCI调用的混合程序中实现
*        多个远程连接。
*********************************************/
/*  说明段 */
EXEC SQL BEGIN DECLARE SECTION;
    text user_id[20], passwd[20];
```

```
    text db_string1[20], bd_string2[20];
EXEC SQL END DECLARE SECTION;
...
/* 宿主变量说明 */
lda_def lda1;                          /* 说明两个 LDA */
lda_def lda2;
dvoid sqllda(lda_def *);               /* 说明 sqllda 函数 */
...
/* 初始化串变量 */
strcpy (user_id, "scott");             /* 用户名和口令 */
strcpy (passwd, "tiger");
strcpy (db_string1, "D:newyork");      /* 数据库名 */
strcpy (db_string2, "D:losangeles");
...
/* 连接第一个 ORACLE 数据库 */
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL CONNECT :user_id IDENTIFIED BY :passwd
      AT db_name1 USING :db_string1;
/* 给出第一个 LDA 供 OCI 使用 */
sqllda(&lda1);
/* 连接第二个 ORACLE 数据库 */
EXEC SQL DECLARE db_name2 DATABASE;
EXEC SQL CONNECT :user_id IDENTIFIED BY :passwd
      AT db_name2 USING :db_string2;
/* 给出第二个 LDA 供 OCI 使用 */
sqllda (&lda2);
.......
```

§3.16 分布事务处理

为了进行分布事务处理，特提供 sqlld2函数。下面简述该函数：

(1) 功能

该函数是为在 X/Open 分布事务处理环境中运行的 OCI 程序而提供的。sqlld2 填充 LDA 中的字段，如何填充取决于传递给它的连接信息。

(2) 调用格式

```
dvoid sqlld2(struct lda_def *lda, text *cname, sb4 *cnlen);
```

(3) 参数说明

lda：是一个 OUT 方式的指针，它指向 LDA。在调用 sqlld2之前必须分配该数据区。

cname：是一个 IN 方式的指针 ，它指向数据库连接名。如果名字串以 Null 终结，则把 cnlen 参数设置为－1L。如果名字串不以 Null 终结，则把 cnlen 设置为串的精确

长度。如果名字全由空格组成，则 sqlld2返回缺省连接的 LDA。

cnlen：是一个 IN 方式的指针，它指向 cname 参数的长度。如果 cname 以 Null 终结，则该参数设置为－1。如果 cnlen 设置为0，则 sqlld2返回缺省连接的 LDA，而不考虑 cname 的内容。

(4) 详细说明

在分布事务处理环境下操作的 OCI 程序并不管理它们自己的连接。但是，所有的 OCI 程序都需要一个有效的 LDA。使用 sqlld2来获得 LDA。

sqlld2 使用 cname 参数中所传递的连接名来填充 LDA。如果该参数是 Null 指针，或者如果 cnlen 参数设置为0，则返回缺省连接的 LDA。程序必须分配 LDA，然后用 lda 指针指向它。

sqlld2不直接返回一个值。如果调用 sqlld2，而没有有效的连接，则在 lda 参数的返回码字段返回如下错误：

ORA－01012：not logged on

sqlld2是 SQLLIB(ORACLE 预编译程序库)的组成部分。SQLLIB 必须被连接到所有调用 sqlld2的程序上。

例3.12 sqlld2 函数的应用

```
/***********************************
*     说明：
*       该例说明怎样使用 sqlld2来获得特定连接的 LDA
***********************************/
#include "ocidfn.h"
#include "ociapr.h"
/*定义两个 LDA */
lda_def lda1;
lda_def lda2;
sb4 clen1 =-1L;   /*名字串以 Null 终结 */
sb4 clen2 =-1L;
...
/* 给第一个 LDA ,供 OCI 使用 */
sqlld2(&lda1,"d:NewYork", &clen1);
if (lda1.rc !=0)
    handle_error();   /* 错误处理 */
/* 给第二个 LDA ,供 OCI 使用 */
sqllda(&lda2,"d:LosAngeles", &clen2);
if (lda2.rc !=0)
    handle_error();  /* 错误处理 */
......
```

§3.17 已过时和将要过时的 OCI 函数

在 ORACLE7 OCI 以前的版本中所使用的函数，有些已不再支持 ORACLE7 OCI。这些

函数如下：

过时的 OCI 函数	在 ORACLE7中推荐替换的函数
OBIND	OBNDRN 或 OBNDRV
OBINDN	OBNDRN 或 OBNDRV
ODFINN	ODEFIN
ODSRBN	ODESCR
OLOGON	ORLON
OSQL	OPARSE

如果应用程序是用早先的 ORACLE 版本写的，就必须重新编码，用表中所示的新函数替换那些过时的函数 。或者可写一个接口函数 来把老的调用映射到新的、推荐的调用上，使开发者不用修改原始代码只用再连接应用即可；或者写一个接口函数 ，来专门处理原始代码不能使用的地方。

还有一些 OCI 函数 ，它们虽然目前仍可在 ORACLE7 OCI 版本下使用，但其性能和功能都较差，目前已有功能较强、性能更好的新函数 替代它们。因此，为了改进应用的性能和功能，建议不要在新的程序中使用这些过时的老例程，而应使用新的函数 。

这些较老的函数 列表如下，ORACLE 在将来的 OCI 版本中将不支持这些调用。

老的 OCI 函数	为新的 ORACLE7程序推荐的函数
ODSC	ODESCR
OERMSG	OERHMS
OLON	ORLON
ONAME	ODESCR
OSQL3	OPARSE

第四章　OCI 程序实例

本节包含四个 OCI 程序实例和由这些实例程序所使用的头(·h)文件。读者通过阅读这些例子可进一步了解如何编写 OCI 程序。

§4.1　头文件

所有的实例程序和头文件都依赖于 oratypes.h 文件所定义的类型。如果读者试图编译和运行实例程序,就要确保包含 oratypes.h 文件,并且还要保证 OCI 的头文件目录路径是在编译程序的 include 路径上。下面描述各个头文件。

4.1.1　oratypes.h

下面是 oratypes.h 头文件清单,该头文件依赖于系统。它仅对 VAX/VMS C 编译程序有效。

```
/*
   oratypes.h 是 ORACLE 外部数据类型定义(VMS 版)文件,它依赖于系统。
   该头文件定义了本书各章节和下面的实例程序中所使用的 C 数据类型。
   当使用较老的 VMSC2.x 版编译器时,应在命令行上定义 'VMSC2X' 符号。
   对于不同的系统,用户必须对 oratypes.h 进行修改,以适应需要。
*/

#ifndef ORASTDDEF
# include <stddef.h>
# define ORASTDDEF
#endif

#ifndef ORALIMITS
# include <limits.h>
# define ORALIMITS
#endif

#ifndef SX_ORACLE
#define SX_ORACLE
#define SX
#define ORATYPES

#ifndef TRUE
```

```
# define TRUE 1
# define FALSE 0
#endif

# ifndef signed
# define signed
# endif /* signed */

typedef int eword;              /* 有符号整数,不重要 */
typedef unsigned int uword;     /*无符号整数,重要 */
typedef signed int sword;       /* 有符号整数,重要 */

#define EWORDMAXVAL ((eword) INT_MAX)
#define EWORDMINVAL ((eword)       0)
#define UWORDMAXVAL ((uword)UINT_MAX)
#define UWORDMINVAL ((uword)       0)
#define SWORDMAXVAL ((sword) INT_MAX)
#define SWORDMINVAL ((sword) INT_MIN)
#define MINEWORDMAXVAL ((eword) 32767)
#define MAXEWORDMINVAL ((eword)       0)
#define MINUWORDMAXVAL ((uword) 65535)
#define MAXEWORDMINVAL ((eword)       0)
#define MINUWORDMAXVAL ((sword) 32767)
#define MAXSWORDMINVAL ((sword) -32767)

typedef har eb1;                /* 有符号字符,不重要 */
typedef unsigned char ub1;      /* 无符号字符,重要 */
typedef signed char sb1;        /* 有符号字符,重要 */

#define EB1MAXVAL ((eb1) SCHAR_MAX)
#define EB1MINVAL ((eb1)       0)
#ifdef VMSC2X
# ifndef lint
# define UB1MAXVAL (UCHAR_MAX)
# endif
#endif

#ifndef UB1MAXVAL
# define UB1MAXVAL ((ub1) UCHAR_MAX)
```

```
#endif
#define UB1MINVAL ((ub1)       0)
#define SB1MAXVAL ((sb1)SCHAR_MAX)
#define SB1MINVAL ((sb1)SCHAR_MIN)
#define MINEB1MAXVAL ((eb1) 127)
#define MAXEB1MINVAL ((eb1)       0)
#define MINUB1MAXVAL ((ub1) 255)
#define MAXUB1MINVAL ((ub1)       0)
#define MINSB1MAXVAL ((sb1) 127)
#define MAXSB1MINVAL ((sb1) -127)
#define UB1BITS     CHAR_BIT
#define UB1MASK 0xff
typedef unsigned char text;       /* 无符号字符串 */
typedef     short    eb2;         /* 有符号短整型数,不重要 */
typedef unsigned short ub2;       /* 无符号短整型数,重要 */
typedef signed short sb2;         /* 有符号短整型数,重要 */
 #define EB2MAXVAL ((eb2) SHRT_MAX)
#define EB2MINVAL ((eb2)       0)
C#define UB2MAXVAL ((ub2)USHRT_MAX)
#define UB2MINVAL ((eb2)       0)
#define SB2MAXVAL ((sb2) SHRT_MAX)
#define SB2MINVAL ((sb2) SHRT_MIN)
#define MINEB2MAXVAL ((eb2) 32767)
#define MAXEB2MINVAL ((eb2)       0)
#define MINUB2MAXVAL ((ub2) 65535)
#define MAXUB2MINVAL ((ub2)       0)
#define MINSB2MAXVAL ((sb2) 32767)
#define MAXSB2MINVAL ((sb2) -32767)

typedef long eb4;                 /* 有符号长整型数,不重要 */
Ctypedef unsigned long ub4;       /* 无符号长整型数,重要 */
typedef signed long sb4;          /* 有符号长整型数,重要 */
#define EB4MAXVAL ((eb4) LONG_MAX)
#define EB4MINVAL ((eb4)       0)
#define UB4MAXVAL ((ub4)ULONG_MAX)
#define UB4MINVAL ((ub4)       0)
#define SB4MAXVAL ((sb4) LONG_MAX)
#define SB4MINVAL ((sb4) LONG_MIN)
#define MINEB4MAXVAL ((eb4) 2147483647)
```

```
#define MAXEB4MINVAL ((eb4)          0)
#define MINUB4MAXVAL ((ub4) 4294967295)
#define MAXUB4MINVAL ((ub4)          0)
#define MINSB4MAXVAL ((sb4) 2147483647)
#define MAXSB4MINVAL ((sb4) -2147483647)
# define CONST const

#ifndef VMSC2X
# define CONST const
#else
#endif

#ifdef VMSC2X
# define dvoid char
#else
# define dvoid void
#endif

typedef void ( * lgenfp_t)(/ * _void_ * /);
#define boolean int
#define SIZE_TMAXVAL UB4MAXVAL
#define MINSIZE_TMAXVAL (size_t)65535
#endif / * SX_ORACLE * /
```

4.1.2 Ocidfn.h

```
/ *
 * 该文件是 OCI 实例程序的公共头文件。
 * 该头文件说明 CDA 和 LDA 的结构。
 * 该文件中所用的数据类型已在 <oratypes.h>中定义
* /

#ifndef OCIDFN
#define OCIDFN
#include <oratypes.h>

/ * cda_head 结构是严格的 PRIVATE。它只在内部使用，
  在 OCI 程序中不使用它。
 * /
```

```
struct cda_head {
        sb2       v2_rc;
        ub2       ft;
        ub4       rpc;
        ub2       peo;
        ub1       fc;
        ub1       rcs1;
        ub2       rc
        ub1       wrn;
        ub1       rcs2;
        sword     rcs3;
        struct {
              struct {
                    ub4     rcs4;
                    ub2     rcs5;
                    ub1     rcs6;
              } rd;
              ub4     rcs7;
              ub2     rcs8;
        }rid;
        sword     ose;
        dvoid     *rcsp;
};

/* CDA 结构,占用64字节 */
  struct cda_def {

      sb2       v2_rc;          /* v2 返回码 */
      ub2       ft;             /* SQL 函数类型 */
      ub4       rpc;            /* 处理的行数 */
      ub2       peo;            /*分析错误的相对位移 */
      ub1       fc;             /* OCI 函数码 */
      ub1       rcs1;           /* 填充区 */
      ub2       rc              /* V7 返回码 */
      ub1       wrn;            /* 警告标志 */
      ub1       rcs2;           /* 保留没用 */
      sword     rcs3;           /* 保留没用 */
      struct {                  /* 行标识结构 */
          struct {
```

```
            ub4     rcs4;
            b2      rcs5;
            ub1     rcs6;
        } rd;
        b4      rcs7;
        ub2     rcs8;
    } rid;
    sword ose;                          /* OSD 依赖错误 */
  dvoid *rcsp;                          /* 指向保留区的指针 */
  ub1   rcs9[64-sizeof(struct cda_head)];   /* 填充到 64 */
};
typedef struct cda_def Cda_Def;

/*LDA 结构定义,它与 CDA 相同。*/
typedef struct cda_def Lda_Def;

/*输入数据类型 */
#define SQLT_CHR 1      /* (ORANET TYPE) 字符串 */
#define SQLT_NUM 2      /* (ORANET TYPE) oracle 数字 */
#define SQLT_INT 3      /* (ORANET TYPE) 整数 */
#define SQLT_FLT 4      /* (ORANET TYPE) 浮点数 */
#define SQLT_STR 5      /* 0结束的串 */
#define SQLT_VNU 6      /*在其前面具有长度字节的 NUM */
#define SQLT_PDN 7      /* (ORANET TYPE) Packed Decimal Numeric */
#define SQLT_LNG 8      /* long */
#define SQLT_VCS 9      /* 变长字符串 */
#define SQLT_NON 10     /* Null/empty PCC Descriptor entry */
#define SQLT_RID 11     /* 行标识 */
#define SQLT_DAT 12     /* ORACLE 格式的日期 */
#define SQLT_VBI 15     /* VCS 格式中的二进制 */
#define SQLT_BIN 23     /* 二进制数据(DTYBIN) */
#define SQLT_LBI 24     /* long 型 二进制 */
#define SQLT_UIN 68     /* 无符号整数 */
#define SQLT_SLS 91     /* 显示前面的分离符号*/
#define SQLT_LVC 94     /* 较长的长字符*/
#define SQLT_LVB 95     /* 较长的二进制 */
#define SQLT_AFC 96     /* Ansi 定长字符 */
#define SQLT_AVC 97     /* Ansi 变长字符 */
#define SQLT_LAB 105    /* 标号类型 */
```

```
#define SQLT_OSL 106      /* os 标号类型 */

#endif /* OCIDFN */
```

4.1.3 Ocidem.h

```
/*
 * 该文件说明在 OCI C 实例程序中使用的其它函数。
*/

#include <oratypes.h>

#ifndef OCIDEM
#define OCIDEM

/* 内部/外部数据类型码 */
#define VARCHAR2_TYPE        1
#define NUMBER_TYPE          2
#define INT_TYPE             3
#define FLOAT_TYPE           4
#define STRING_TYPE          5
#define ROWID_TYPE           11
#define DATE_TYPE            12
/* 在示范程序中用的 ORACLE 错误码 */
#define VAR_NOT_IN_LIST          1007
#define NO_DATA_FOUND            1403
#define NULL_VALUE_RETURNED 1405
/* 一些 SQL 和 OCI 函数码 */
#define FT_INSERT            3
#define FT_SELECT            4
#define FT_UPDATE            5
#define FT_DELETE            9

#define FC_OOPEN             14

/*
 * OCI 函数代码标号,
 * 对应于 CDA 中的 OCI 函数码
*/
CONST text *oci_func_tab[] = {(text *) "not used",
```

```
/* 1—2 */      (text *)"not used", (text *)"OSQL",
/* 3—4 */      (text *)"not used", (text *)"OEXEC,OEXN",
/* 5—6 */      (text *)"not used", (text *)"OBIND",
/* 7—8 */      (text *)"not used", (text *)"ODEFIN",
/* 9—10 */     (text *)"not used", (text *)"ODSRBN",
/* 11—12 */    (text *)"not used", (text *)"OFETCH,OFEN",
/* 13—14 */    (text *)"not used", (text *)"OOPEN",
/* 15—16 */    (text *)"not used", (text *)"OCLOSE",
/* 17—18 */    (text *)"not used", (text *)"not used",
/* 19—20 */    (text *)"not used", (text *)"not used",
/* 21—22 */    (text *)"not used", (text *)"ODSC",
/* 23—24 */    (text *)"not used", (text *)"ONAME",
/* 25—26 */    (text *)"not used", (text *)"OSQL3",
/* 27—28 */    (text *)"not used", (text *)"OBNDRV",
/* 29—30 */    (text *)"not used", (text *)"OBNDRN",
/* 31—32 */    (text *)"not used", (text *)"not used",
/* 33—34 */    (text *)"not used", (text *)"not used",
/* 35—36 */    (text *)"not used", (text *)"not used",
/* 37—38 */    (text *)"not used", (text *)"not used",
/* 39—40 */    (text *)"not used", (text *)"not used",
/* 41—42 */    (text *)"not used", (text *)"not used",
/* 43—44 */    (text *)"not used", (text *)"not used",
/* 45—46 */    (text *)"not used", (text *)"not used",
/* 47—48 */    (text *)"not used", (text *)"not used",
/* 49—50 */    (text *)"not used", (text *)"not used",
/* 51—52 */    (text *)"not used", (text *)"OCAN",
/* 53—54 */    (text *)"not used", (text *)"OPARSE",
/* 55—56 */    (text *)"not used", (text *)"OEXFET",
/* 57—58 */    (text *)"not used", (text *)"OFLNG",
/* 59—60 */    (text *)"not used", (text *)"ODESCR",
/* 61—62 */    (text *)"not used", (text *)"OBNDRA",
};

#endif    /* OCIDEM */
```

4.1.4 Ociapr.h

```
/*
 * 说明 OCI 函数,
 * 其中包括原型信息。
```

```
 *  ANSI C 编译器使用该头文件。
 */

#ifndef OCIAPR
#define OCIAPR

#include <oratypes.h>
#include <ocidfn.h>

sword obndra(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, ub2 *alen, ub2 *arcode, ub4 maxsiz,
             ub4 *cursiz, text *fmt, sword fmtl, sword fmtt);
sword obndrn(struct cda_def *cursor, sword sqlvn, ubl *progv,
             sword progvl, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt);
sword obndrv(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, text *fmt, sword fmtl, sword fmtt);
sword obreak(struct cda_def *lda);
sword ocan  (struct cda_def *cursor);
sword oclose(struct cda_def *cursor);
sword ocof  (struct cda_def *lda);
sword ocom  (struct cda_def *lda);
sword ocon  (struct cda_def *lda);
sword odefin(struct cda_def *cursor, sword pos, ub1 *buf,
             sword buf1, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt, ub2 *rlen,
             ub2 *rcode);
sword odessp(struct cda_def *cursor, text *objnam, size_t onlen,
             ub1 *rsvl, size_t rsulln, ub1 *rsv2, size_t rsv2ln,
             ub2 *ovrld, ub2 *pos, ub2 *level, text **argnam,
             ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1 *mode,
             ub4 *dtsiz, sb2 *prec, sb2 *scale, ub1 *radix,
             ub4 *spare, ub4 *arrsiz);
sword odsc  (struct cda_def *cursor, sword pos, sb2 *dbsize,
             sb2 *fsize, sb2 *rcode, sb2 *dtype, sb1 *buf,
             sb2 *buf1, sb2 *dsize);
sword odescr(struct cda_def *cursor, sword pos, sb4 *dbsize,
```

```
                    sb2 *dbtype, sb1 *cbuf, sb4 *cbufl, sb4 *dsize,
                    sb2 *prec, sb2 *scale, sb2 *nullok);
sword oerhms(struct cda_def *lda, sb2 rcode, text *buf,
                    sword bufsiz);
sword oermsg(sb2 rcode, text *buf);
sword oexec (struct cda_def *cursor);
sword oexfet(struct cda_def *cursor, ub4 nrows,
                    sword cancel, sword exact);
sword oexn  (struct cda_def *cursor, sword iters, sword rowoff);
sword ofen  (struct cda_def *cursor, sword nrows);
sword ofetch(struct cda_def *cursor);
sword oflng (struct cda_def *cursor, sword pos, ub1 *buf,
                    sb4 bufl, sword dtype, ub4 *retl, sb4 offset);
sword ologof(struct cda_def *lda);
sword olon  (struct cda_def *lda,text *uid, sword uidl,
                    text *pswd, sword pswdl, sword audit);
sword oopen (struct cda_def *cursor, struct cda_def *lda,
                    text *dbn, sword dbnl, sword arsize,
                    text *uid, sword uidl);
sword oopt  (struct cda_def *cursor, sword rbopt, sword waitopt);
sword oname (struct cda_def *sursor, sword pos, sb1 *tbuf,
                    sb2 *tbufl, sb1 *buf, sb2 *bufl);
sword oparse(struct cda_def *cursor, text *sqlstm, sb4 sqllen,
                    sword defflg, ub4 lngflg);
sword orlon (struct cda_def *lda, ub1 *hda, text *uid,
                    sword uidl, text *pswd, sword pswdl, sword audit);
sword orol  (struct cda_def *lda);
sword osql3 (struct cda_def *cda, text *sqlstm, sword sqllen);
sword sqlld2(struct cda_def *lda, text *cname, sb4 *cnlen);
sword sqllda(struct cda_def *lda);

#endif /* OCIAPR */
```

4.1.5 Ocikpr.h

```
/*
 * 说明 OCI 函数,
 * 其中注释出原型信息。
 * 非 ANSI C 编译器使用该头文件。
 */
```

```
#ifndef OCIKPR
#define OCIKPR

#include <oratypes.h>

sword obndra( /*_struct cda_def *cursor, text *sqlvar, sword sqlvl,
                ub1 *progv, sword progvl, sword ftype, sword scale,
                sb2 *indp, ub2 *alen, ub2 *arcode, ub4 maxsiz,
                ub4 *cursiz, text *fmt, sword fmtl,sword fmtt _*/ );
sword obndrn( /*_ struct cda_def *cursor, sword sqlvn, ub1 *progv,
                sword progvl, sword ftype, sword scale, sb2 *indp,
                text *fmt, sword fmtl, sword fmtt _*/ );
sword obndrv( /*_ struct cda_def *cursor, text *sqlvar, sword sqlvl,
                ub1 *progv, sword progvl, sword ftype, sword scale,
                sb2 *indp, text *fmt, sword fmtl, sword fmtt _*/ )
sword obreak( /*_ struct cda_def *lda _*/);
sword ocan   ( /*_ struct cda_def *cursor _*/);
sword oclose( /*_ struct cda_def *cursor _*/);
sword ocof   ( /*_ struct cda_def *lda _*/);
sword ocom   ( /*_ struct cda_def *lda _*/);
sword ocon   ( /*_ struct cda_def *lda _*/);
sword odefin( /*_ struct cda_def *cursor, sword pos, ub1 *buf,
                sword bufl, sword ftype, sword scale, sb2 *indp,
                text *fmt, sword fmtl, sword fmtt, ub2 *rlen,
                ub2 *rcode  _*/);

sword odsc
( /*_ struct cda_def *cursor, sword pos, sb2 *dbsize,
                sb2 *fsize, sb2 *rcode, sb2 *dtype, sb1 *buf,
                sb2 *bufl, sb2 *dsize  _*/ );
sword odescr ( /*_ struct cda_def *cursor, sword pos, sb4 *dbsize,
                sb2 *dbtype, sb1 *cbuf, sb4 *cbufl, sb4 *dsize,
                sb2 *prec, sb2 *scale, sb2 *nullok _*/ );
sword odessp ( /*   _ struct cda_def *cursor, text *objnam, size_t rsv2ln,
              ub1 *rsvl, size_t rsvlln, ub1 *rsv2, size_t rsv2ln,
              ub2 *ovrld, ub2 *pos, ub2 *level, text **argnam,
              ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1 * mode,
              ub4 *dtsiz, sb2 *prec, sb2 *scale, ub1 *radix,
```

```
                    ub4 *spare, ub4 *arrsiz */ );
sword oerhms ( /* _ struct cda_def *lda, sb2 rcode, text *buf,
                    sword bufsiz */ );
sword oermsg( /* _ sb2 rcode, text *buf */ );
sword oexec ( /* _ struct cda_def *cursor */ );
sword oexfet( /* _ struct cda_def *cursor, ub4 nrows,
                    sword cancel, sword exact */ );
sword oexn  ( /* _ struct cda_def *cursor, sword iters, sword rowoff */ );
sword ofen  ( /* _ struct cda_def *cursor, sword nrows */ );
sword ofetch( /* _ struct cda_def *cursor */ );
sword oflng ( /* _ struct cda_def *cursor, sword pos, ub1 *buf,
  sb4 buf1, swprd dtype, ub4 *ret1, sb4 offset */);
sword ologof( /* _ struct cda_def *lda */);
sword olon  ( /* _ struct cda_def *lda, text *uid, sword uidl,
                    text *pswd, sword pswdl, sword audit */ );
sword oopen ( /* _ struct cda_def *cursor, struct cda_def *lda,
                    text *dbn, sword dbnl, sword arsize,
                    text *uid, sword uidl */ );
sword oopt  ( /* _ struct cda_def *cursor, sword rbopt, sword waitopt */ );
sword oname ( /* _ struct cda_def *cursor, sword pos, sb1 *tbuf,
                    sb2 *tbuf1, sb1 *buf, sb2 *buf1 */ );
sword oparse( /* _ struct cda_def *cursor, text *sqlstm, sb4 sqllen,
                    sword defflg, ub4 lngflg */ );
sword orlon ( /* _ struct cda_def *lda, ub1 *hda, text *uid,
                    sword uidl, text *pswd, sword pswd1, sword audit */ );
sword osql3 ( /* _ struct cda_def *cda, text *sqlstm, sword sqllen */);
void sqlld2( /* _ struct cda_def *lda, text *cname, sb4 *cnlen */);
void sqllda( /* _ struct cda_def *lda */ );

#endif /* OCIKPR */
```

§4.2 实例1

```
/*************************************
 *    说明:
 *         该实例程序把新的职员记录加到职员数据库中,
 *         并为保证数据库的完整性进行必要的检查 。其处理步骤如下:
 *         (a) 与 ORACLE 相连接
 *         (b) 打开两个光标,
 *         (c) 切断自动提交
```

```
*          (d) 检索当前最大的职员号
*               分析 SQL 语句
*               定义选择表项的输出变量
*               执行和提取,得到最大的职员号
*          (e)确定职员名和职务的长度
*               分析"selemp"中的语句
*               描述"selemp"中语句的选择表项,得到职员名和职务的长度
*          (f)分析"insert" 中的 INSERT 语句
*               分配输出缓冲区
*               结合"insert"中的 语句的虚拟变量
*          (g) 分析"seldept"中的 SELECT 语句
*               结合"seldept" 中的 语句的虚拟变量
*               描述选择表项 "DNAME"
*               定义选择表项的输出变量
*          (h) 程序要求用户输入如下的数据:
*               职员名,职务,工资和部门
*               (通过查询 DEPT 表来确认输入的部门号有效)
*               当请求职员名时,如果输入'0',则程序结束。
*          (i) 自动选择职员号:开始时使用当前的最大职员号,以后每次增加10
*          (j) 执行插入,如果记录被成功地插入,
*               则显示职员名,部门名和职员号。
*          (k) 结束事务,提交变更。
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 包括头文件 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <oratypes.h>
/* LDA 和 CDA 结构说明 */
#include <ocidfn.h>
/* 使用 ANSI 函数风格说明 */
#include <ociapr.h>
/* 说明 demo 常数和结构 */
#include <ocidem.h>

/* 分析标志 */
#define DEFER_PARSE     1
#define NATIVE          1
```

```
#define VERSION_7          2
/* 用户名和口令 */
text *username = (text *) "SCOTT";
text *password = (text *) "TIGER";

/* 定义程序中使用的 SQL 语句 */
text *insert = (text *) "INSERT INTO emp (empno, ename, job, sal,deptno)\
    VALUES (:empno, :ename, :job, :sal, :deptno)";
text *seldept = (text *) "SELECT dname FROM dept WHEREdeptno = :1";
text *maxemp = (text *) "SELECT NVL (MAX(empno), 0) FROM emp";
text *selemp = (text *) "SELECT ename,job FORM emp";

/* 定义一个 LDA,一个 HDA, 和两个 CDA */
Lda_Def lda;
ub1     hda[256];
Cda_Def cda1;
Cda_Def cda2;

void err_report(Cda_Def *);

/* 程序体 */
main()
{
   /* 局部量说明 */
   sword empno, sal, deptno;
   sword len, len2, rv, dsize, dsize2;
   sb4 enamelen, joblen, deptlen;
   sb2 sal_ind, job_ind;
   sb2 db_type, db2_type;
   sb1 name_buf[20],name2_buf[20];
   text *cp, *ename, *job, *dept;
   /* 与 ORACLE 相连接,遇到错误时退出。*/
   if (orlon(&lda, hda, username, -1, password, -1, 0))
   {
      err_report(&lda);
      exit(EXIT_FAILURE);
   }
   printf("Connected to ORACLE as %s\n",username);
   /* 打开两个光标,当遇到错误时退出。*/
```

```
if (oopen(&cda1, &lda, (text *) 0,-1,-1,(text *) 0,-1))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}
if (oopen(&cda2, &lda, (text *) 0,-1,-1,(text *) 0,-1))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}

/* 切断自动提交,缺省是切断,当遇到错误时退出 */
if (ocof(&(lda))
{
    err_report (&lda);
    do_exit (EXIT_FAILURE);
}
/* 检索当前最大的职员号。*/
if (oparse (&cda1,maxemp, (sb4) -1, DEFER_PARSE,
          (ub4) VERSION_7)) /* 分析 SQL 语句 */
{
    err_report (&cda1);
    do_exit (EXIT_FAILURE);
}

if (odefin (&cda1, 1, (ub1 *) &empno, (sword) sizeof(sword),
             (sword) INT_TYPE,
             (sword) -1, (sb2 *) 0, (text *) 0, -1, -1,
             (ub2 *) 0, (ub2 *) 0)) /* 定义选择表项的输出变量 */

{
    err_report (&cda1);
    do_exit (EXIT_FAILURE);
}
if (oexfet (&cda1, (ub4) 1, FALSE, FALSE)) /* 执行和提取 */

{
    if (cda1.rc == NO_DATA_FOUND)
        empno = 10;
```

```
    else
    {
        err_report(&cda1);
        do_exit (EXIT_FAILURE);
    }
}
/*确定职员名和职务的长度 */
if (oparse (&cda1, selemp, (sb4) -1, FALSE, VERSTON_7))
                /*分析"selemp"中的语句*/
{
    err_report(&cda1);
    do_exit (EXIT_FAILURE);
}
len = sizeof (name_buf);
len2 = sizeof (name2_buf);
if (odescr (&cda1, 1, &enamelen,
            (sb2 *) &db_type, name_buf, (sb4 *) &len,
            (sb4 *) &dsize, (sb2 *) 0, (sb2 *) 0, (sb2 *) 0) ||
        odescr (&cda1, 2,&joblen,
            (sb2 *) &db_type, name2_buf, (sb4 *) &len2,
            (sb4 *) &dsize2, (sb2 *) 0, (sb2 *) 0, (sb2 *) 0))
              /* 描述"selemp"中语句的选择表项 */

{
    err_report (&cda1);
    do_exit (EXIT_FAILURE);
}
/*分析"insert"中的 语句 */
if (oparse (&cda1, insert, (sb4) -1, FALSE, (ub4) VERSION_7))
{
    err_report (&cda1);
    do_exit (EXIT_FAILURE);
}

/* 分析"seldept"中的语句 */
if (oparse (&cda2, seldept, (sb4) -1, FALSE, (ub4) VERSION_7))
{
    err_report(&cda2);
    do_exit (EXIT_FAILURE);
```

```
}
/* 分配输出缓冲区,考虑到 \n 和 '\0'. */
ename = (text *) malloc((int) enamelen + 2);
job = (text *) malloc ((int)joblen + 2);

/* 结合"insert"中的 语句的虚拟变量 */
if (obndrv (&cda1, (text *) ":ename", -1,
            (ub1 *) ename, enamelen+1,
            STRING_TYPE, -1, (sb2 *) 0, (text *) 0, -1, -1)||
    obndrv(&cda1, (text *) ":sal", -1,
        (ub1 *) job, joblen+1,
        STRING_TYPE, -1, &job_ind, (text *) 0, -1, -1) ||
    obndrv(&cda1, (text *) ":sal", -1, (ub1 *) &sal,
        (sword) sizeof (sal),
        INT_TYPE,-1, &sal_ind, (text *) 0, -1, -1)||
    obndrv(&cda1, (text *) ":deptno", -1,(ub1 *) &empno,
        (sword) sizeof (empno), INT_TYPE, -1,
        (sb2 *) 0, (text *) 0, -1, -1))
{
    err_report(&cda1);
    do_exit(EXIT_FAILURE);
}

/* 结合"seldept" 中的 语句的虚拟变量 */

if (obndrn(&cda2,
           1,
           (ub1 *) &deptno,
           (sword) sizeof(deptno),
           INT_TYPE,
           -1,
           (sb2 *) 0,
           (text *) 0,
           -1,
           -1))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}
```

```
/* 描述选择表项 "DNAME". */
len = sizeof (name_buf);
if (odescr(&cda2, 1,(sb4 *) &deptlen, &db_type,
            name_buf, &len, &dsize, (sb2 *) 0,
            (sb2 *) 0, (sb2 *) 0))
{
    err_report (&cda2);
    do_exit(EXIT_FAILURE);
}
/* 现在已知 dept 缓冲区长度,分配 dept 缓冲区. */
dept = (text *) malloc ((int) deptlen + 1);

/*定义选择表项的输出变量*/
if (odefin(&cda2,
            1,
            (ub1 *) dept,
            deptlen+1,
            STRING_TYPE,
            -1,
            (sb2 *) 0,
            (text *) 0,
            -1,
            -1,
            (ub2 *) 0,
            (ub2 *) 0))
{
    err_report(&cda2);
    do_exit(EXIT_FAILURE);
}
for (;;)
{
    /* 提示职员名,没名时退出循环 */
    printf ("\nEnter employee name (or CR to EXIT): ");
    fgets((char *) ename, (int) enamelen+1, stdin);
    fflush(stdin);
    cp = (text *) strchr((char *) ename, '\n');
    if (cp == ename)
    {
```

```
        printf ("Exiting... ");
        do_exit (EXIT_SUCCESS);
    }
    if (cp)
        *cp = '\0';
    else
        printf ("Employee name may be truncated.\n");
    /* 提示职员职务和工资 */
    printf ("Enter employee job: ");
    job_ind = 0;
    fgets((char *) job, (int) joblen + 1, stdin);
    fflush(stdin);
    cp = (text *) strchr ((char *) job, '\n');
    if (cp == job)
    {
        job_ind = -1;      /* 使它在表中为 NULL */
        printf ("Job is NULL.\n");/* 使用指示器变量 */
    }
    else if (cp == 0)
        printf ("Job description may be truncated. \n");
    else
        *cp = '\0';

    printf ("Enter employee salary: ");
    scanf ("%d", &sal);
    fflush(stdin);
    sal_ind=(sal<=0)? -2:0; /* 设置指示器变量 */
    /*
    *提示职员的部门号,
    *通过执行和提取来确认 输入的部门号是有效的。
    */
    do
    {
        printf("Enter employee dept:");
        scanf("%d", &deptno);
        fflush(stdin);
        if (oexec(&cda2)||
            (ofetch(&cda2) && (cda2.rc != NO_DATA_FOUND)))
        {
```

```
        err_report(&cda2);
        do_exit(EXIT_FAILURE);
      }
      if (cda2.rc == NO_DATA_FOUND)
        printf("The dept you entered doesn't exist. \n");
  } while (cda2.rc == NO_DATA_FOUND);

  /*
   * 职员号 empno 增 10,执行 INSERT 语句,
   * 如果返回码是 1 (索引号重复),则产生下一个职员号
   */
  empno += 10;
  if (oexec(&cda1) && cda1.rc != 1)
  {
     err_report (&cda1);
     do_exit(EXIT_FAILURE);
  }
  /* 执行插入 */
  while (cda1.rc == 1)
  {
     empno += 10;
     if (oexec(&cda1) && cda1.rc !=1)
     {
        err_report (&cda1);
        do_exit (EXIT_FAILURE);
     }
  }
  /* 提交变更,结束事务 */
  if(ocom(&lda))
  {
     err_report(&lda);
     do_exit(EXIT_FAILURE);
  }
  printf(
    "\n\n%s added to the %s departmetn as employee number %d \n",
        ename, dept, empno);
  } /* end for(;;) */
  do_exit (EXIT_SUCCESS);
}
```

```
/* 显示错误信息 */
void
err_report(Cda_Def *cursor)
{
    sword n;
    text msg[512];

    printf("\n-- ORACLE error-- \n");
    printf("\n");
    n = oerhms (&lda, cursor->rc, msg, (sword) sizeof(msg));
    fprintf (stderr, "%s\n", msg);
    if (cursor->fc > 0)
        fprintf (stderr, "Processing OCI function %s",
            oci_func_tab[cursor->fc]);
}

/*
* 结束处理:关闭光标,结束事务,切断与 ORACLE 连接。
*/
do_exit (sword exit_code)
{
    sword error = 0;
    if (oclose(&cda1))
    {
        fprintf(stderr,"Error closing cursor 1.\n");
        error++;
    }
    if (oclose (&cda2))
    {
        fprintf (stderr, "Error closing cursor 2.\n");
        error++;
    }
    if (ologof(&lda))
    {
        fprintf(stderr, "Error on disconnect.\n");
        error++;
    }
    if (error == 0 && exit_code == EXIT_SUCCESS)
```

```
                printf ("\nG'day\n");

            exit (exit_code);
    }
```

§4.3 实例2

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  说明:
*       该程序接收用户输入的任意 SQL 语句,然后处理该语句。
*       输入的语句可以是多行语句,并且必须以分号结束。
*       如果是查询语句,则显示结果 。其处理步骤如下:
*       (a) 与 ORACLE 连接
*       (b) 打开光标,遇到不可恢复的错误则退出
*       (c) 处理用户的 SQL 语句:
*         输入一个 SQL 语句,遇"exit"退出 ,
*         分析该语句,
*         结合输入变量,
*         如果语句是查询, 则执行前描述和定义所有选择表项。
*         执行该语句
*         提取和显示查询的行
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 包括 C 头文件 */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* 包括 OCI 指定的头文件 */
#include <oratypes.h>
#include <ocidfn.h>
#include <ociapr.h>
#include <ocidem.h>

/* 在该程序中使用的常数 */
#define MAX_BINDS        12
#define MAX_ITEM_BUFFER_SIZE 33
#define MAX_SELECT_LIST_SIZE 12
#define MAX_SQL_IDENTIFIER      31
#define PARSE_NO_DEFER       0
#define PARSE_V7_LNG         2
```

```
/*定义 LDA,CDA 和 HDA */
Lda_Def lda;
Cda_Def cda;
ub1 hda[256];

/*说明一个结合值的数组 */
text bind_values[MAX_BINDS][MAX_ITEM_BUFFER_SIZE];

/* 说明查询信息的结构 */
struct describe
{
    sb4     dbsize;
    sb2     dbtype;
    sb1     buf[MAX_ITEM_BUFFER_SIZE];
    sb4     buflen;
    sb4     dsize;
    sb2     precision;
    sb2     scale;
    sb2     nullok;
};

struct define
{
    ub1      buf[MAX_ITEM_BUFFER_SIZE];
    float    flt_fuf;
    sword    int_buf;
    sb2      indb;
    sb2      col_retlen, col_retcode;
};

/* 定义 deccribe 和 define 的结构变量 */
struct describe desc[MAX_SELECT_LIST_SIZE];
struct define def[MAX_SELECT_LIST_SIZE];

/* 说明该程序的函数 */
sword connect (void);                    /* 与 ORACLE 相连接 */
sword describe_define(Cda_Def *);        /*描述和定义所有选择表项*/
sword do_binds(Cda_Def *, text *);       /* 结合输入变量 */
```

```
void do_exit(sword);                    /* 结束处理 */
void oci_error(Cda_Def *);              /* 输出错误信息 */
sword get_sql_statement(void);          /* 输入一个 SQL 语句 */
void print_header(sword);               /* 输出标题 */
void print_rows(Cda_Def *, sword);      /* 提取和显示查询的行 */

/* 全程变量 */
static text sql_statement[2048];
static sword sql_function;
static sword numwidth = 8;
/* 程序体 */
main()
{
    sword col, errno, n, ncols;
    text *cp;

    /* 与 ORACLE 连接 */
    if (connect())
        exit (-1);

    /* 打开光标,遇到不可恢复的错误则退出 */
    if (oopen (&cda, &lda, (text *) 0, -1, -1, (text *) 0, -1))
    {
        printf("Error opening cursor. Exiting...\n");
        ologof(&lda);
        exit(-1);
    }
    /* 处理用户的 SQL 语句 */
    for (;;)
    {
        /* 输入一个 SQL 语句,遇"exit"退出 */
        if (get_sql_statement())
            do_exit(0);

        /* 分析该语句,非延迟分析。
           因此,如果有错误,则马上返回 */
        if (oparse (&cda, (text *) sql_statement, (sb4) -1,
                (sword) PARSE_NO_DEFER, (ub4) PARSE_V7_LNG))
        {
```

```
            oci_error(&cda);
            continue; /* 取下一句分析 */
        }
        /* 保存分析后立刻返回的 SQL 函数码 */
        sql_function = cda.ft;

        /* 结合输入变量 */
        if ((ncols = do_binds (&cda, sql_statement)) == -1)
            continue;

        /* 如果语句是查询，则执行前描述和定义所有选择表项。*/
        if (sql_function == FT_SELECT)
            if ((ncols = describe_define(&cda)) == -1)
              continue;

        /* 执行该语句 */
        if (oexec(&cda))
        {
            oci_error(&cda);
            continue;
        }
        /* 提取和显示查询的行 */
        if (sql_function == FT_SELECT)
        {
            print_header(ncols);
            print_rows(&cda, ncols);
        }
        /* 显示处理的行数 */
        if (sql_function == FT_SELECT ||
            sql_function == FT_UPDATE ||
            sql_function == FT_DELETE ||
            sql_function == FT_INSERT)
            printf("\n%d row%c processed. \n", cda.rpc,
                    cda.rpc ==1? '\0' : 's');
        else
            printf("\nStatement processed. \n");
        } /* end for (;;) */
} /* end main () */
```

```
/* 与ORACLE相连接 */
sword
connect()
{
    text username[132]
    text password[132]
    sword n;
    /* 连接失败时再次连接,至多三次 */
    for (n = 3; --n >= 0;)
    {
        /*输入用户名和口令人*/
        printf("Username: ");
        gets((char *) username);
        printf ("Password: ")
        gets ((char *) password);
        /*连接 */
        if (orlon (&lda, hda, username, -1,password,-1,-1))
        {  /*失败*/
           printf (" Cannot connect as %s.\n", username);
           printf("Try again .\n\n");
        }
        else
            return; /*成功 */
    }
    /*失败退出*/
    print f(" Connection failed. Exitng...\n");
    return -1;
}

/* 描述和定义选择表项 */
sword
describe_define (Cda_Def *cda)
{
    sword col, deflen, deftyp;
    static ubl *defptr;
    /* 描述选择表项*/
    for (col=0; col< MAX_SELECT_LIST_SIZE;col++)
    {
        desc[col].buflen = MAX_ITEM_BUFFER_SIZE;
```

```
if (odescr( cda, col + 1,& desc[col]. dbsiz,
            &desc[col],dbtype, &desc[col]. buf[0],
            &desc[col]. precision, &desc[col]. scale,
            &desc[col]. nullok))
{
    /* 描述结束,则 Break */

    if (cda->rc == VAR_NOT_IN_LIST)
        break;
    else
    {  /* 描述失败 */
       oi_error(cda);
       return -1;
    }
}
/* 调整显示尺寸和类型 */
switch (desc[col]. dbtype)
{
   case NUMBER_TYPE:
       desc[col]. dbsize = numwidth;
       /* 处理有定标的浮点 NUMBER */
       if (desc[col]. scale ! =0)
       {
           defptr =(ub1 * ) &def[col]. flt_buf;
           deflen = (sword) sizeof (float);
           deftyp = FLOAT_TYPE;
           desc[col]. dbtype = FLOAT_TYPE;
       }
       else
       {
          defptr = (ub1 * ) &def[col]. int_buf;
          deflen = (sword) sizeof(sword);
          deftyp = INT_TYPE;
          desc[col]. dbtype = INT_TYPE;
       }
       break;
       default:
        i f (desc[col]. dbtype == DATE_TYPE)
          desc[col]. dbsize =9;
```

```
            if (desc[col].dbtype == ROWID_TYPE)
              desc[col].dbsize =18;
            defptr = def[col].buf;
            deflen = desc[col].dbsize > MAX_ITEM_BUFFER_SIZE?
                MAX_ITEM_BUFFER_SIZE :desc[col].dbsize + 1;
            deftyp = STRING_TYPE;
            break;
        }
        /* 定义选择表项的输出变量 */
        if (odefin (cda, col + 1,
            defptr, deflen, deftyp,
            -1, &def[col].indp, (text *)0,-1,-1,
            &def[col].col_retlen,
            &def[col].col_retcode))
        {
            oci_error(cda);
            return -1;
        }
    }
return col;
}

/* 结合输入变量 */
sword
do_binds(Cda_Def *cda, text *stmt_buf)
{
    sword i, in_literal, n;
    text *cp, *ph;
    /* 寻找结合输入变量 */
    for (i = 0, in_literal = FALSE, cp = stm_buf;
            *cp && i< MAX_BINDS; cp++)
    {
        if (*cp =='\'')
            in_literal =~n_literal;
        if(*cp == ':' && ! in_literal)
        {
            for (ph= ++cp, n=0;
                *cp && (isalnum (*cp) || *cp =='_')
                    && n<MAX_SQL_IDENTIFIER;
```

```
            cp++,n++)
            ;
        *cp='\0';
        printf(" Enter value for %s:", ph);
        gets ((char *)&bind_values[i][0];
        /* 用 obndrv ()进行结合,
            注意:结合变量必须是静态的 */
        if (obndrv(cda, ph, -1, &bind_values [i][]0 ,-1,
                VARCHAR2_TYPE,
                -1,(sb2 *) 0,(text *) 0,-1,-1))
        {
            oci_error(cda);
            return -1;
        }
        i++;
    } /* end if ( * cp== ...) */
  }
    /* end for () */
  return i;
}

/* 退出处理, LDA 和 CDA 是是全程的 */
void
do_exit(sword rv)
{   /* 关闭光标 */
    if(oclose (&cda))
        fputs ("Error closing cursor !\n", stdout);
    /* 退出数据库 */
    if(ologof (&lda))
        fputs ("Error logging off!\n", stdout);
    exit (rv);
}

/* 打印错误信息,然后根据需要可选择继续,也可选择退出 */
void
oci_error(Cda_Def *cda)
{
    text msg[512];
    sword n;
```

```
    fputs("\n-- ORACLE ERROR --\n", stderr);
    n = oerhms(&lda, cda->rc,msg, (sword) sizeof (msg));
    fprintf(stderr, "%. *s",n,msg);
    fprintf(stderr, "Processing OCI function%s\n",
        oci_func_tab[cda->fc]);
    fprintf(stderr, "Do you want to continue? [yn];");
    fgets ((char *) msg, (int) sizeof(msg), stdin);
    if ( *msg !='\n' && *msg != 'y' && *msg !='Y')
        do_exit (1);
    fputc('\n',stdout);
}

/* 输入 SQL 语句,如果是 exit,则退出 */
sword
get_sql_statement ()
{
    text cbuf[1024];
    text *cp;
    sword stmt_level;

    for (stmt_level =1; ;)
        {
            if (stmt-level == 1)
            {
                /*初始化语句缓冲区,并打印提示. */
                *sql_statement = '\0';
                fputs ("\nOCISQL >", stdout);
            }

            esle
              printf("%3d
                ", stmt_level);
            /* 输入 SQL 语句 */
            gets ((char *) cbuf);
            if (*cbuf == '\0')
                continue;
            if (strncmp ((char *) cbuf, "exit", 4) ==0)
                return 1;
```

```
            /* 并置到语句缓充区中 */
            if (stmt_level >1)
                strcat ((char *) sql_statement, " ");
            strcat ((char *) sql_statement, (char *) cbuf);

            /* 检查结束符 */
            cp = &sql_statement[strlen ((char *)sql_statement)-1];

            while (isspace (* cp))
              cp--;
            if (*cp == ';')
            {
                *cp = '\0';
                break;
            }
            stmt_level++;
        }
        return 0;
}

/* 输出标题 */
void
print_header (sword ncols)
{
        sword col, n;

        fputc('\n', stdout);

        for (col =0; col < ncols; col ++)
        {
            n= desc[col].dbsize - desc[col].buflen;
            if (desc[col].dbtype == FLOAT_TYPE ||
                desc[col].dbtype == INT_TYPE)
            {
                printf ("%*c", n, ' ');
                printf ("%*.*s", desc [col].buflen,
                        desc[col].buflen, desc[col].buf);
            }
```

```
            else
            {
                printf("%*.*s", desc[col].buflen,
                        desc[col].buflen, desc[col].buf);
                printf("%*c", n, ' ');
            }
            fputc (' ',stdout);
        }
        fputc ('\n',stdout);

        for (col =0; col < ncols;col++)
        {
            for (n= desc[col].dbsize; --n >= 0; )
                fputc ('-', stdout);
            fputc (' ', stdout);
        }
        fputc('\n', stdout);
}

/*提取和输出行*/
void
print_rows(Cda_Def *cda, sword ncols)
{
        sword col, n;

        for (;;)
        {
                fputc('\n', stdout);
                /* Fetch a row. Break on end of fetch,
                        disregard null fetch "error". */
                if (ofetch(cda))
                {
                if (cda->rc== NO_DATA_FOUND)
                    break;
                if (cda -> rc != NULL_VALUE_RETURNED)
                    oci_error(cda);
                }
                for (col = 0;col < ncols ; col++)
                {
```

```
            /* Check col.return code for null. If
                null, pint n paces, else print value. */
            if (def [col].indp < 0)
                printf ("% *c",desc[col].dbsize, ' ');
            else
            {
                switch (desc[col].dbtype)
                {
                    case FLOAT_TYPE:
                      printf ("%*.* f", numwidth, 2, def[col].flt_
                    buf);
                      break;
                    case INT_TYPE:
                      printf ("%*d", numwidth, def[col].int_buf);
                      break;
                    default:
                      n = printf ("%s", def[col].buf);
                      n = desc[col].dbsize - n;
                      if (n > 0)
                        printf ("%*c", n, ' ');
                      break;
                }
            }
            fputc(' ', stdout);
        }
        } /* end for (;;) */
    }
```

§4.4 实例3

```
/*************************************
*    说明:
*        该例使用 oflng 函数来检索 LONG 列的一部分。
*        该例通过循环从表中提取64K 字节的信息块,并把它发送到
*        一个转换器缓冲区(例如,一个数字模拟转换器的 FIFO 缓冲
*        区)来玩数字语音信息。
*        为了更好地理解该例,程序建立了一个表,并把一些模拟数据
*        插入该表中。
*        本例中的转换器函数并非真实的,只是示意的。
*        该例的处理步骤如下.
```

```
*           (a) 与 ORACLE 相连接
*           (b) 打开光标
*           (C) 提示是否退出?(Y/N)
*           (d)删除 voice_mail 表:分析执行"DROP TABLE voice_mail" 语句
*           (e)建 voice_mail 表:分析执行"CREATE TABLE voice_mail" 语句
*           (f) 插入模拟信息 :
*             给出 INSERT 语句,分析它,结合虚拟变量,
*             设置结合变量的值,执行插入。
*           (g) 查询和提取信息块:
*             给出 SELECT 语句,分析它 ,定义输出变量
*             执行查询,提取 msg_id 和开始100字节的信息
*           (h) 玩这些信息,循环直到没有更多的数据输出
***********************************/
/*头文件*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MSG_SIZE 200000
#include <oratypes.h>
#include <ocidfn.h>
#include <ociapr.h>
#include <ocidem.h>

/*说明 CDA、LDA 和 HDA */
Cda_Def cda;
Lda_Def lda;
ub1     hda(256);

/*函数说明*/
dvoid do_exit (sword);
dvoid oci_error (Cda_Def * );
dvoid play_msg (ub1 * , sword);
/* 程序体 */
main ()
{  /*自动量说明*/
   text sql_statement [256];
   register sword i;
   sb2 indp;
```

```
ub2 retl, rcode;
sword msg_id;
sb4 msg_len,len, offset;
ub1 *ucp;
register ub1 *ucp1
b4 ret_len;

/* 与 ORACLE 相连接 */
if (orlon (&lda, hda, (text *) "scott/tiger", -1,
      (text *) 0,-1,0))
{
    fputs ("Cannot connect with username SCOTT. Exiting...\n",
        stderr);
    exit (EXIT_FAILURE);
}
fprintf("connected to oracle as user SCOTT. \n");

/* 打开光标 */
if(oopen(&cda, &lda,(text *)0, -1, -1,(text *)0, -1))
{ fputs("cannot open cursor. Exiting---\n", stderr);
  exit(EXIT_FAILURE);
}

/*提示是否退出?*/
fputs ("Program is about to drop the VOICE_MAIL table. \n", stdout );
fputs (" Is this OK (Y or N)? :", stdout);
fflush (stdout);
gets((char *) sql_statement);
if ( * sql_statement != 'y' && * sql-statement != 'Y')
    do_exit (EXIT_SUCCESS);

/*分析执行"DROP TABLE voice_mail" */
/* 非延迟分析,设置 defflg 参数,
    以便直接执行 DDL 语句 */
if (oparse (& cda, (text *) "DROP TABLE voice_mail", -1, FALSE, 2)))
{
    if (cda. rc== 942)
        fputs("Table did not exist. \n", stdout);
    else
```

```
        oci_error(&cda);
}
else
    fputs ("Dropped table \"Voice_mail\".\n", stdout);

/* 给出 SQL 语句:" CREATE TABLE ..." */
strcpy((char *) sql_statement, " CREATE TABLE voice_mail \
    (msg_id NUMBER(6),msg_len NUMBER (12), msg LONG RAW)");
/* 分析执行该语句 */
if (oparse (&cda, sql_statement, -1,FALSE, 2))
    oci_error (&cda);
fputs("created table\" voice_mail \".\n", stdout);

/* 插入模拟信息 */
strcpy ((char *)sql_statement,
    "INSERT INTO voice_mail (msg_id, msg_len,msg) \
                VALUES (:1,:2,:3)" );
if (oparse(&cda, sql_statement, -1,FALSE, 2)) /*分析*/
    oci_error(&cda);

if(obndrn(&cda, 1, (ub1 *) &msg_id,4,3,-1,
          (sb2 *) 0, (text *) 0, 0, -1))      /*结合虚拟变量*/

    oci_error(&cda);

ucp = (ub1 *)malloc (MSG_SIZE);    /* 在结合前,设置缓冲区地址 */

if (ucp == 0)
{
    fputs ("malloc error \n", stderr);
    do_exit(EXIT_FAILURE);
}
if (obndrn (&cad, 2, (ub1 *) &msg_len, 4,3,-1,
          (sb2 *) 0,(text *)0,0,-1))
/*结合虚拟变量*/

    oci_error (&cda);
if(obndrn(&cda, 3, ucp, MSG_SIZE, 24, -1, (sb2 *)0, (text *)0, 0, -1))
    oci_error(&cda);      /*结合虚拟变量*/
```

```
msg_id =100;          /* 在插人前设置结合变量的值 */
msg_len = MSG_SIZE;
for (i = 0, ucp1 = ucp; i < MSG_SIZE; i++)
    *ucp1++ = (ub1) i % 128;

if (oexn(&cda, 1,0))          /* 执行插人 */
    oci_error(&cda);
fputs ("Data inserted in table \"voice_mail\".\n",stdout);

/*在建立文本数据后,查询和提取信息块 */
strcpy ((char *) sql_statement, "select msg_id, msg_len, msg\
        from voice_mail where msg_id = 100"); /*给出 SELECT 语句 */
if (oparse (&cda, sql_statement, -1,0,2))          /*分析 */
    oci_error(&cda);
if (odefin (&cda,                          /*定义输出变量*/
                1,                  /* index */
                (ub1 *) &msg_id/* output variable */
                4,                  /* length */
                3,                  /* datatype */
                -1,                 /* scale */
                (sb2 *) 0,          /* indp */
                (text *) 0,         /* fmt */
                0,                  /* fmt1 */
                -1,                 /* fmtt */
                (ub2 *) 0,          /* retl */
                (ub2 *)0))          /* rcode */
    oci-error(&cda);
if (odefin (&cda,

                2,                  /* index */
                (ub1 *) &msg_len/* output variable */
                4,                  /* length */
                3,                  /* datatype */
                -1,                 /* scale */
                (sb2 *) 0,          /* indp */
                (text *) 0,         /* fmt */
                0,                  /* fmt1 */
                -1,                 /* fmtt */
```

```
                (ub2 *) 0,      /* retl */
                (ub2 *)0))      /* rcode */
    oci-error(&cda);
if (odefin (&cda,
                3,              /* index */
                ucp             /* output variable */
                100,            /* length */
                24,             /* LONG RAW datatype code */
                -1,             /* scale */
                &indp,          /* indp */
                (text *) 0,     /* fmt */
                0,              /* fmtl */
                -1,             /* fmtt */
                &retl,          /* retl */
                (&rcode))       /* rcode */
    oci-error(&cda);

/* 查询,得到 msg_id 和开始100字节的信息 */
if (oexfet (&cda,
                (ub4)1,         /* nrows */
                0,              /* cancel (FALSE) */
                0))             /* exact (FALSE ) */
{
    oci_error(&cda);
}
fprintf(stdout,
    "Message %d is available, length is %d. \n", msg_id, msg_len);
fprintf (stdout,
    "indp =%d, rcode = %d, retl = %d\n", indp, rcode,retl);

/* 玩这些信息,循环直到没有更多的数据输出 */

for (offset = (ub4) 0; ; offset += (ub4) 0x10000)
{
    len = msg_len < 0x10000 ? msg_len :0x10000;

    if (oflng(& cda,
                3,          /* position */
                ucp,        /* buf */
```

```
                    len,          /* buf1 */
                    24,           /* datatype */
                    &ret_len      /* ret1 */
                    offset))      /* offset */
            oci_error (&cda);

        /* 输出该信息块 */
        play msg(ucp, len);
        msg_len -= len;
        if (msg_len <=0)
          break;
    }
    do_exit (EXIT_SUCCESS);
} /* end of main */

/*玩信息 */
dvoid
play_msg (ub1 *buf, sword len)
{
    fprintf (stdout, "\"playing \" %d bytes. \n",len);
}

/* 错误处理 */
dvoid
oci_error (Cda-Def *cda)
{
     text msg[200];
     sword n;

     fputs("\n-- ORACLE ERROR --\n",stderr);
     n=oerhms (&lda, (sb2) cda-> rc,msg, 200);
     fprintf(stderr, "%.*s",n,msg);
     fprintf(stderr, "Processing OCI function %s\n.",
            oci_func_tab[(int) cda->fc]);
     do_exit(EXIT_FAILURE);
}

/* 退出处理 */
dvoid
```

```
do_exit(sword rv)
{
    fputs("Exiting...\n", stdout);
    if(oclose(&cda)) /* 关闭光标 */
    {
        fputs (" Error closing cursor. \n",stderr);
        rv = EXIT_FAILURE;
    }
    if (ologof(&lda)) /* 退出 ORACLE */
    {
        fputs(" Error logging off. \n", stderr);
        rv = EXIT_FAILURE;
    }
    exit(rv);
}
```

§4.5 实例4

```
/*********************************
*    说明:
*        该例根据光标把数据提取到 PL/QL 表内。
*        该表使用 obndra 函数被结合到 C 数组上。
*        其处理步骤如下:
*            (a) 与 ORACLE 相连接
*            (b) 打开 OCI 光标
*            (c) 提取和打印数据
*            (d) 关闭 OCI 光标
*            (e) 切断与 ORACLE 连接.
*
*        该程序中用到的包装如下:
* create or replace package calldemo as
* type char_arry is table of varchar2(20) index by binary integer;
* type num_array is table of float index by binary_integer;

*
* procedure get_employees(
*   dept_number in     integer,        --which department to query
*   batch_size in      integer,        --how many rows at a time
*   found         in out integer,      --n of rows actually returned
*   done_fetch out     integer,        --all done flag
```

```
*   emp_name   out   char_array,     --arrays of employee names,
*   job     out     char_array,       -- jobs,
*   sal     out     num_array);       -- salaries
*   end;
*
*
* create or replace package body calldemo as
*
* cursor get_emp( dept_number in integer ) is
*       select ename, job, sal from emp
*             where deptno = dept_number;
*
* -- 过程 get_employees 提取一批职员记录(记录数取决于过程
* -- 的调用者)。
* --该过程可以由其它过程或应用程序调用。其处理步过程如下:
*     --如果光标没有打开,则打开;
*     --提取一批记录行,并返回实际检索的行数。
*     --在结束提取时,过程关闭光标。
*
*     procedure get_emplyees(
*           dept_number in integer,
*           batch_size in integer,
*           found in out integer,
*           done_fetch out integer,
*           emp_name out char_array,
*           job out char_array,
*           sal out num_array) is
*
* begin
*     if NOT get_emp%ISOPEN then      --open the cursor if it is
*       open get_emp(dept_number);   -- not already open
*     end if;
*
* --提取 "batch_size" 行到 PL/SQL 表中,
* --当提取到结束行时,关闭光标并退出循环,
* --仅在迂到最后一组行时返回。
*
*     done_fetch: =FALSE;
*     found :=0;
```

```
*
* for i in 1..batch_size loop
*     fetch get_emp                    --get one emp table row
*       into emp_name (i), job(i), sal (i);
*
*     if get_emp%not found then        --if no row was found,then
*       close get_emp;                 --close the cursor
*       done_fetch := TRUE;            --indicate all done
*       exit;                          --exit the loop
*     else
*       found:= found +1;              --else count the row and continue
*     end if;
* end loop;
* end;
* end;
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* 头文件 */
#include <stdio.h>
#include <string.h>
#include <oratypes.h>
#include <ocidfn.h>
#include <ociapr.h>
#include <ocidem.h>

/* 定义数量 */
#define MAX_ARRAY_SIZE        5
#define NO_PARSE_DEFER        0
#define V7_LNGFLG           2
#define VC_LENGTH            20

/* 说明 CDA,LDA 和 HDA */
Cda_Def cda;
Lda_Def lda;
ub1    hda[256];

/* 说明该程序中的函数 */
dvoid do_fetch (void);
dvoid oci_error (void);
```

```
/* 程序开始 */
main(argc, argv)
sword argc;
text **argv;
{
   text username[128];
   /* 提供用户名和口令 */
   if (argc >1)
       strncpy((char *) username, (char *) argv[1],
           sizeof(username)-1);
   else
       strcpy ((char *) username, "SCOTT/TIGER" );
   /* 与 ORACLE 相连接 */
   if (orlon (&lda, hda, username, -1, (text *) 0,-1,-1))
   {
        printf(Cannot connect as %s. Exiting...\n", username);
        return -1;
   }
   else
       printf("Connected. \n");

   /* 打开 OCI 光标 */
   if (oopen (&cda, &lda, (text *) 0, -1,-1,(text *) 0,-1))
   {
        printf("Cannot open cursor data area, exiting...\n");
        return -1;
   }

   /* 提取和打印数据 */
   do_fetch ();

   /* 关闭 OCI 光标 */
   if (oclose(&cda))
   {
        printf("Error closing crusor!\n");
        return -1;
   }

   /* 切断与 ORACLE 连接. */
```

```
    if (ologof (&lda))
    {
        printf("Error logging off!\n");
        return -1;
    }
    return 0;
}

/* 建立一个无名的 PL/SQL,以调用提取数据的存储过程 */
dvoid
do_fetch (void)
{
    /* 给出无名 PL/SQL 块 */
    text *call_fetch = (text *) "\
    begin \
    calldemo.get_employees (:deptno, :t_size, :num_ret, :all_done,\
    :e_name, :job, :sal);\
    end;";
    sword table_size = MAX_ARRAY_SIZE;
    sword i, n, n_ret, done_flag;
    sword dept_num;
    sb2 n_ret_indp;
    ub2 n_ret_len, n_ret_rcode;
    ub4 n_ret_cursiz = 0;

    text emp_name [MAX_ARRAY_SIZE][VC_LENGTH];
    sb2 emp_name_indp[MAX_ARRAY_SIZE];
    ub2 emp_name_len[MAX_ARRAY_SIZE];
    ub2 emp_name_rcode[MAX_ARRAY_SIZE];
    ub4 emp_name_cursiz = (ub4) MAX_ARRAY_SIZE;

    text job[MAX_ARRAY_SIZE][VC_LENGTH];
    sb2 job_indp[MAX_ARRAY_SIZE];
    ub2 job_len[MAX_ARRAY_SIZE];
    ub2 job_rcode[MAX_ARRAY_SIZE];
    ub4 job_cursiz = (ub4) MAX_ARRAY_SIZE;

    float salary [MAX_ARRAY_SIZE];
    sb2 salary_indp[MAX_ARRAY_SIZE];
```

```
ub2 salary_len[MAX_ARRAY_SIZE];
ub2 salary_rcode[MAX_ARRAY_SIZE];
ub4 salary_cursiz = (ub4) MAX_ARRAY_SIZE;
/* 分析该无名 PL/SQL 块 */
if(oparse(&cda, call_ftch, -1,
    NO_PARSE_DEFER,V7_LNGFLG))
{
     oci_error();
     return;
}
/* 初始化结合数组 */
for (i = 0;i < MAX_ARRAY_SIZE;i++)
{
     emp_name_len[i] =VC_LENGTH;
     job-len[i] = VC_LENGTH;
     salary_len [i] = sizeof (float);
}
n_ret_len = sizeof (sword);

/* 结合部门号 IN 参数 */
if (obndrv (&cda, (text *) ":deptno", -1, (ub1 *) &dept_num,
    (sword) sizeof (sword), INT_TYPE, -1,
    (sb2 *)0, (text *) 0,-1,-1 ))
{
     oci_erro();
     return;
}
/* 结合表尺寸 IN 参数 */
if(obndrv (&cda, (text *) ":t_size", -1, (ub1 *) &table_size,
    (sword) sizeof (sword)
    INT_TYPE, -1, (sb2 *) 0, (text *) 0,-1,-1))
{
     oci_error();
     return;
}
/* 结合进行提取的 OUT 参数 */
if(obndrv (&cda, (text *) ":all_done", -1, (ub1 *) &done_flag,
    (sword) sizeof (sword),
    INT_TYPE ,-1 (sb2 *) 0, (text *) 0,-1,-1))
```

```
{
    oci_erro();
    return;
}

/* 用 obndra 结合 OUT n_ret */
if (obndra (&cda,
    (text *) ": num_ret",
    -1,
    (ub1 *)&n_rnt,
    (sword) sizeof (sword),
    INT_TYPE
    -1,
    &n_ret_indp,
    &n_ret_len,
    &n_ret_rcode,
    (ub4) 0, /* pass as 0, not 1, when binding a scalar */
    (ub4 *) 0, /* pass as the null pointer when scalar */
    (text *) 0,
    -1
    -1)),
{
    oci_error();
    return;
}
/* 结合职员名数组 */
if (obndra (&cda,
    (text *) ": e_name",
    -1,
    (ub1 *) emp_name,
    VC_LENGTH,
    VARCHAR2_TYPE
    -1,
    emp_name_indp,
    emp_name_len,
    emp_name_rcode,
    (ub4)MAX_ARRAY_SIZE,
    &emp_name_cursiz,
```

```
        (text *) 0,
        -1,
        -1))
{
        oci_error();
        return;
}
/* 结合 job 数组 */
if (obndra (&cda,
        (text *) ": job",
        -1,
        (ub1 *) job,
        VC_LENGTH,
        VARCHAR2_TYPE,
        -1,
        job_indp,
        job_len,
        job_rcode,
        (ub4) MAX_ARRAY_SIZE,
        & job_cursiz,
        (text *) 0,
        -1,
        -1))
{
        oci_error();
        return;
}
/* 结合 salary 数组 */
if (obndra (& cda,

        (text *) ": sal",
        -1,
        (ub1 *) salary,
        (sword) sizeof (float),
        FLOAT_TYPE,
        -1,
        salary_indp,
        salary_len,
        salary_rcode,
```

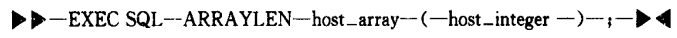
```
        (ub4) MAX_ARRAY_SIZE,
        &salary_cursiz,
        (text *) 0,
        -1,
        -1))
{
        ci_error();
        return;
}
printf("\nenter deptno: ");
scanf("%d, &dept_num);
fflush (stdin);
#ifdef DEBUG
printf("\nenter table size:");
scanf("%d", &table_size);
fflush(stdin);
#endif

for (;;)
{
        /* 执行提取 */
        if(oexec(&cda))
        {
                oci-error();
                return;
        }
        printf("\n%d row%c returned \n",
                n_ret, n_ret == 1?'\0' :'s');
        if (n_ret >0)
        {
                printf("\n%-*.*s%-*.*s%s\n",
                        VC_LENGTH ,VC_LENGTH, "Employee Name",
                        VC_LENGTH ,VC_LENGTH,"Job", " Salary ");
                for (i = 0; i< n_ret; i++)
                {
                        n = printf("%.*s",emp_name_len[i],emp_name[i]);
                        printf("%*c", VC_LENGTH - emp_name_len[i], '');
                        n = printf("%.*s", job_len[i],job[i];
                        printf("%*c",VC_LENGTH -job_len[i], '');
```

```
                printf ("8.2f\n", salary[i];
            }
        }
        if (one_flag !=0)
        {
            printf("\n");
            break;
        }
    }
    return;
}

/*错误处理 */
dvoid
oci_error(void)
{
    text msg[900];
    sword rv;

    rv = oerhms(&lda, cda.rc, msg, (sword ) sizeof (msg);

    printf("\n\n%.*s",rv,msg);
    printf("Processing OCI function %s\n",
            oci_func_table(int) cda.fcl);
    return;
}
```

附录 A. 嵌入 SQL 语句语法图

1 ARRAYLEN

ARRAYLEN command ::=



(a) 功能

用于限制传递给 PL/SQL 块的数组维数，它必须放置在说明段。

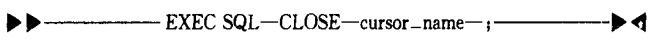
(b) 关键字及参数说明

host_array：宿主数组变量名

host_integer：是整型宿主变量

2 CLOSE

CLOSE command ::=



(a) 功能

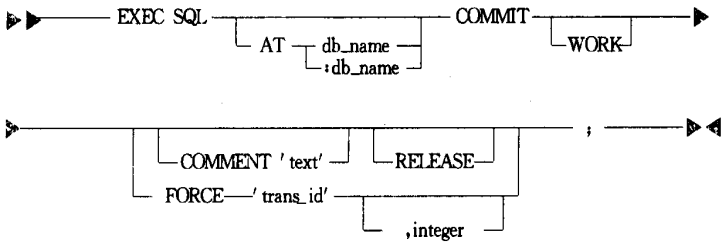
CLOSE 语句使一个光标不起作用，释放打开光标时所获得的资源，以及释放分析块。

(b) 关键字及参数说明

cursor_name：指出被关闭的光标名，该光标必须是当前打开的光标。

3 COMMIT

COMMIT command ::=



(a) 功能

事务提交。

(b). 关键字及参数说明

AT 子句：标识一个数据库，省略该子句时，是指缺省数据库。

WORK：是为支持与标准 SQL 一致性而设置的，使用它可增强程序可读性。

COMMENT：指出一个与当前事务相关的注释。'text' 是不超过 50 个字符的文字串。

RELEASE：该关键字释放所有资源并切断与 ORACLE 的连接。

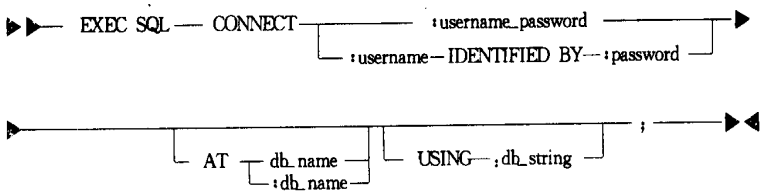
FORCE：手工提交一个不确定的分布事务。

db_name、:db_name：指定数句库连接名。

integer：系统变更号。

4 CONNECT

CONNECT command::=



(a) 功能

实现登录。

(b) 关键字及参数说明

AT 子句：标识一个数据库，省略该子句时，是指缺省数据库。

:username：用户标识名。

:password：用户口令。

:username_password：用户标识名/用户口令。

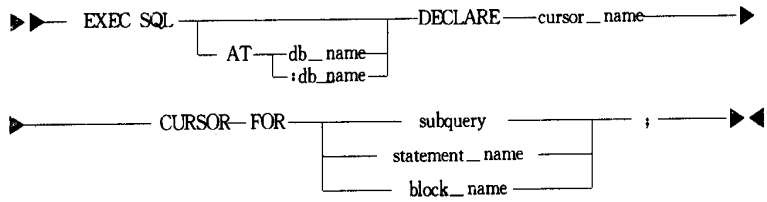
:db_string：是 SQL * Net 串，用于连接非缺省数据库。

db_name：字符串，表示数据库连接名(下同)。

:db_name：宿主变量，包含数据库连接名(下同)。

5 DECLARE CURSOR

DECLARE CURSOR ::=



(a) 功能：

说明和命名一个光标，并使它与一个 SQL 语句或 PL/SQL 块相联系。

(b) 关键字和参数说明

AT 子句：标识一个数据库，指明说明的光标是相对于该数据库的。数据库标识的方法同 INSERT 语 句。

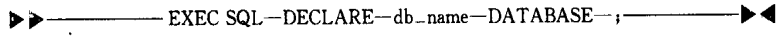
cursor_name：是光标名。

subquery：是与光标相关联的 SELECT 语句，该 SELECT 语句不包含 INTO 子句。

statement_name、block_name：标识与光标相联系的 SQL 语句或 PL/SQL 块名。该名字要在前面的 DECLARE STATEMENT 语句中说明。

6 DECLARE DATABASE

DECLARE DATABASE command::=



(a) 功能

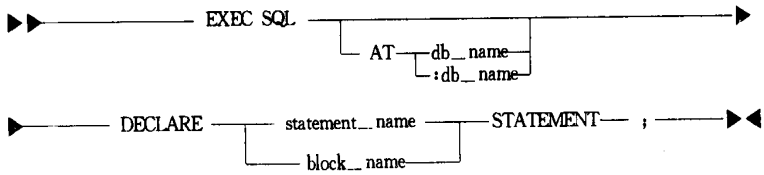
说明一个在 SQL 语句的 AT 子句中使用的非缺省数据库名。

(b) 关键字和参数说明

db_name:是一个标识符,标识数据库名。

7 DECLARE STATEMENT

DECLARE STATEMENT command::=



(a) 功能

该说明语句说明一个 SQL 语句或 PL/SQL 块的标识符。

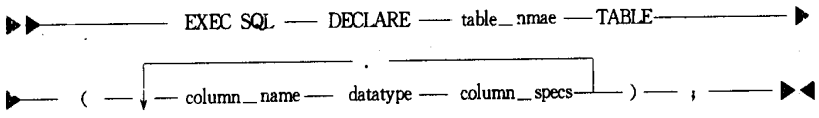
(b) 关键字和参数说明

AT 子句:标识一个数据库。其它参考 INSERT 语句。要说明的 SQL 语句和 PL/SQL 块名就是相对于该数据库的。

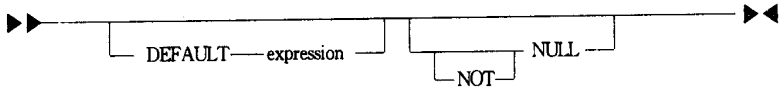
statement_name、block_name: 是为该 SQL 语句说明的标识符。

8 DECLARE TABLE

DELETE command::=



column_specs::=



(a)功能

该语句定义一个表或视图的结构,其中包含每列的数据类型、可选的缺省值和 Null (Not Null)说明。

(b) 关键字和参数说明

table_name:表或视图名。

column_name:列名。

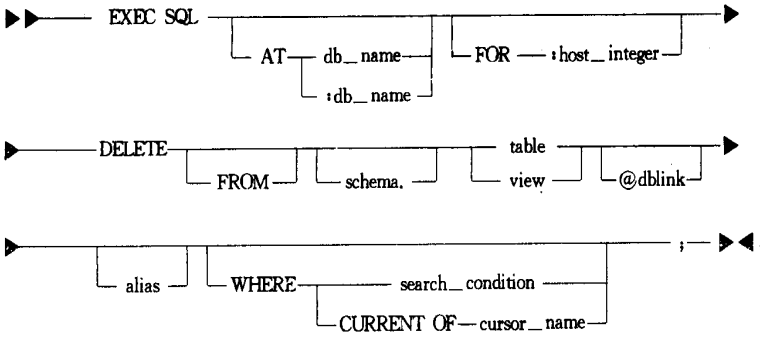
column_specs:指定缺省值 Null 或 Not Null。

9 DELETE

DELETE command::=

(a) 功能

该语句从一个表或视图的基本表中删除行。



(b) 关键字和参数说明

AT 子句:该子句标识要进行删除操作的数据库名。

FOR :host_integer:其说明同 INSERT 语句。

schema:是包含表或视图的模式。如果省 schema 时,则 ORACLE 假定表或视图是在用户所拥有的模式中。

table:是表名,指出从该表中删除行。

view:是视图名,如果指定 view 时,则 ORACLE 从该视图的基本表中删除行。

dblink:指定一个数据库链名,其说明同 INSERT 语句。如果省略时,ORACLE 假定表或视图分布在当地数据库上。

alias:是分配给该表的别名。一般用于具有关联查询的 DELETE 语句中。

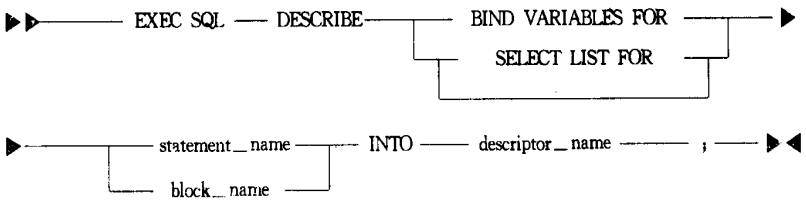
WHERE 子句:指出那些行被删除。

search_condition:是逻辑表达式,表明仅删除满足该条件的行。该条件能包含宿主变量和可选的指示器变量。

CURRENT OF 子句:指示仅删除由光标所提取的当前行。在 CURRENT OF 子句中所引用的光标必须在 FOR UPDATE OF 子句中说明,且必须打开和定位在某一行上。如果没有执行 FETCH 或 OPEN,则 CURRENT OF 子句不会影响行。该子句不能与宿主数组或动态 SQL 一起使用。

10 DESCRIBE

DESCRIBE command::=



(a) 功能

描述选择表项或虚拟输入变量。

(c)关键字和参数说明:

BIND VARIABLES:把 SQL 语句或 PL/SQL 块的输入宿主变量的信息保存到结合描述区中。

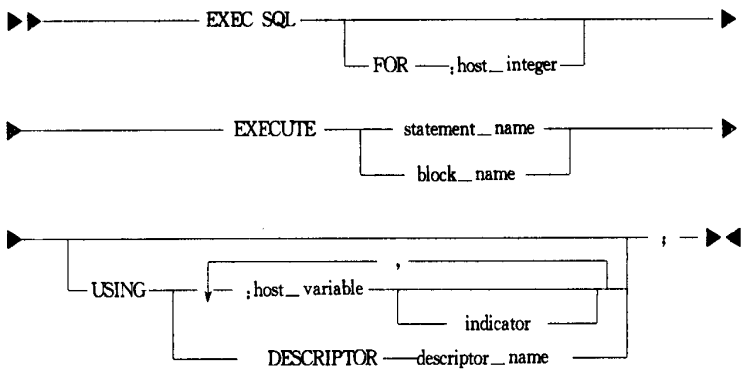
SELECT LIST：把 SELECT 语句的选择表项的信息保存到选择描述区中。

statement_name/block_name：标识预先用 PREPARE 语句分析的 SQL 语句或 PL/SQL 块。

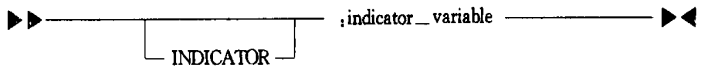
descriptor_name：是描述区的名字。

11 EXECUTE

EXCUTE command：：=



indicator：：=



(a) 功能

该语句执行一个预先分析过的动态 SQL 语句或 PL/SQL 块。

(b) 关键字及参数

FOR 子句：指出要处理的数组元素的最大个数。

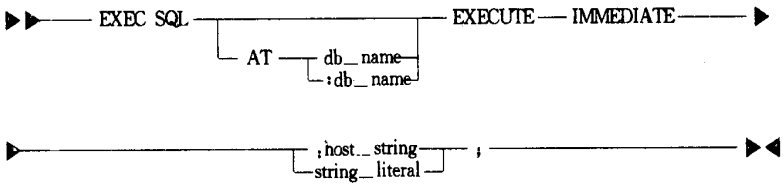
statement_name 和 block_name：说明同 DELETE。

UNING 子句：指出一个描述区或实宿主变量表，它们被用来替换被分析串中的虚拟宿主变量。

descriptor_name：标识一个在前面的 DESCRIBE 语句中引用过的描述区。

12 EXECUTE IMMEDIATE

EXECUTE IMMEDIATE command：：=



(a) 功能

用于分析和执行动态 SQL 语句。

(b) 关键字及参数说名

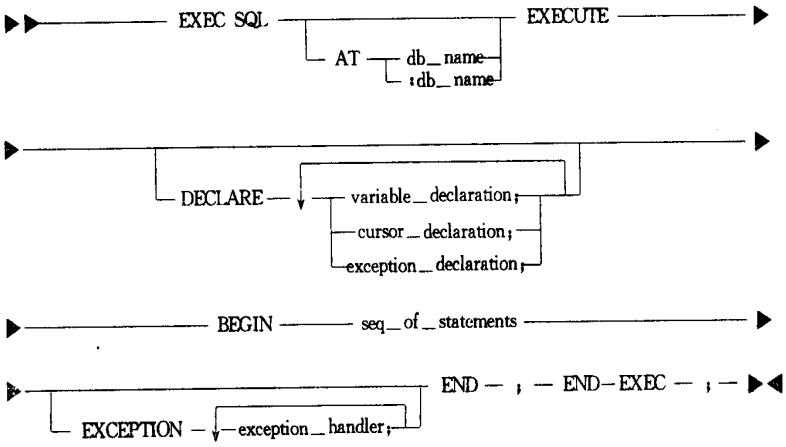
AT 子句：描述同前。

:host_string：是字符串宿主变量，用来保存动态 SQL 语句文本。

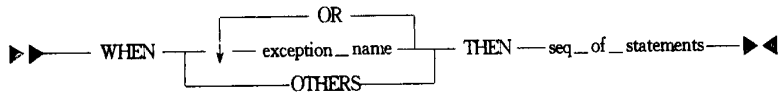
string_literal：是文字串，表示一个动态 SQL 语句文本。

13 EXECUTE psql_block

EXECUTE psql_block：：=



exception_handler：：=



(a) 功能：

执行一个嵌入 PL/SQL 块。

(b) 关键字和参数说明

db_name、:db_name：标识一个非缺省连接。

variable_declaration：说明 PL/SQL 常数和变量。

cursor_declaration：说明一个 PL/SQL 光标。

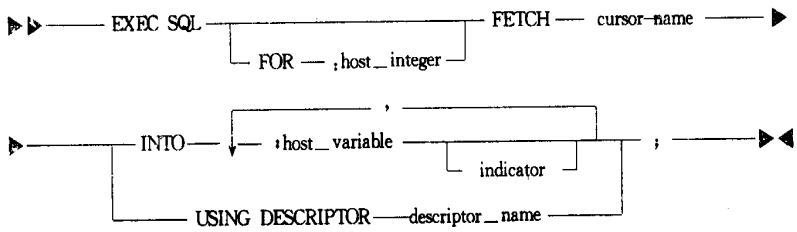
exception_declaration：说明一个 PL/SQL 例外(错误条件)。

sel_of_statemants：是 PL/SQL 语句的序号。

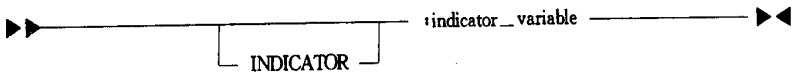
exception_name：标识一个 PL/SQL 的预定义或用户定义例外。

14 FETCH

FETCH command：：=



indicator::=



(a) 功能

从活动集中提取一行或多行，并把提取的值赋给输出宿主变量。

(b) 关键字和参数说明

FOR :host_integer 子句：

如果使用了数组宿主变量，则该子句限制提取的行数。如果省略该子句，则 ORACLE 提取的行数等于最小数组的维数。

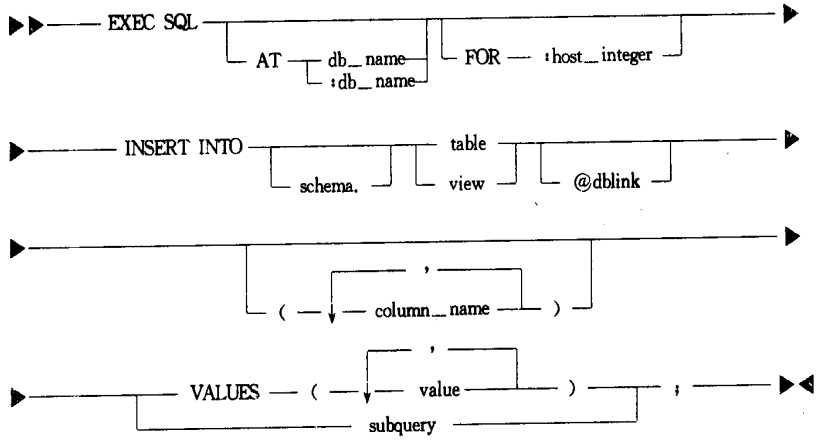
cursor_name：是光标名，FETCH 语句从该光标所标识的光标缓冲区中读取一行或多行。

INTO 子句：指出一个宿主变量和可选的指示器变量表。所提取的数据被赋给这些变量。

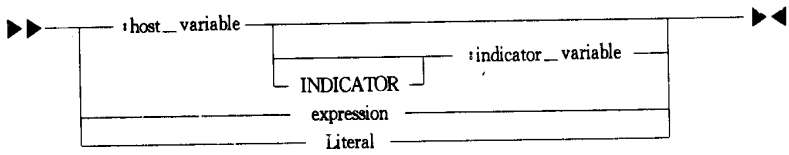
USING 子句：指出在前面的 DESCRIBE 语句中所引用的描述区。这个子句仅在动态 SQL 方法 4 中使用。

15 INSERT

INSERT command::=



value::=



(a) 功能

INSERT 语句向一个表或一个视图的基本表插入一些新行。

(b) 关键字和参数说明

AT 子句：该子句指出执行插入的数据库。该数据库可用两种方式标识：

• 使用 DECLARE DATABASE 语句中说明的标识符 db_name 来标识。

• 使用宿主变量（:db_name）来标识，该宿主变量的值是前面说明的 db_name。

如果省略该子句，则对缺省数据库插入。

FOR :host_integer 子句：该子句设置要处理的数组元素个数。它适于 DELETE、EXECUTE、FETCH、INSERT、OPEN 和 UPDATE 等语句。当不想对整个数组进行操作时，可用 FOR 子句限制要处理的元素数，使之正好满足所需要的元素个数。

宿主变量中设置的数一定不能超过最小数组维数，且大于 0，否则将产生错误信息，且没有行被处理。

如果省略该子句时，执行该语句所处理的数组元素数与最小数组的维数相同。

schema：是包含表和视图的模式。如果省略 schema 的话，ORACLE 假定该表和视图属于用户自己的模式。

table 和 view：table 是表名，行被插入到该表中。如果指定了 view，则把行插入到视图的基本表中。

dblink：是远程数据库的连接名。其中 table 和 view 就是该远程数据库中的表或视图。如果使用具有分布可选项的 ORACLE 的话。才能对远程的表或视图进行插入。

如果省略 dblink 时，则 ORACLE 假定 table 和 view 都在当地数据库上。

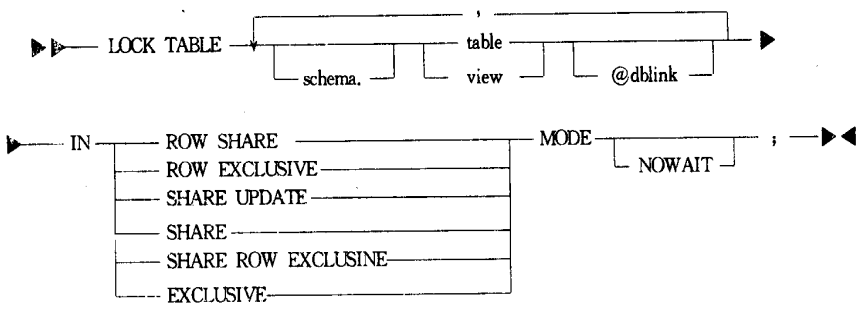
column_name：该项指定表或视图的一个列清单。VALUES 子句或 subquery 给出被插入行的列值，这些值被赋给列清单中所指定的相应列。表或视图中未被包括在该列清单中的列，插入建表时所指定的缺省值。如果省略了整个列清单，则 VALUES 子句或 query 必须指定表中全部列的值。

VALUES 子句：指定向表或视图插入的一行值。value 可以是常数、宿主变量、SQL 表达式或伪列（如 USER 和 SYSDATE）。VALUES 子句中值的个数必须等于列清单中的名字个数。如果省略列清单的话，VALUES 子句中必须包含表中的每一列值。

subquery：这是一个不带 INTO 子句的 SELECT 语句（叫子查询），其语法同 SQL 语言中的 SELECT 语句。它为 INSERT 语句提供要插入的值。

16 LOCK TABLE

LOCK TABLE Command：：=



(a) 功能

实现表封锁。

(b) 关键字及参数说明

Schema：是包含表或视图的模式，如果省略 schema，ORACLE 就假定该表或视图属于该用户所拥有的模式。

table/view：是要被封锁的表或视图的名字；如果指定 View，则 ORACLE 封锁视图的基本表。

dblink：是一个与远程 ORACLE7 数据库相连的数据库连接名。该句中指出的 table 或 View 被分配在该远程数据库上。

ROW SHARE：指定行共享方式的封锁，它允许并行访问被封锁的表，阻止用户以排它方式封锁整个表。ROW SHARE 与 SHARE UPDATE 是同义的。

ROW EXCLUSIVE：它与 ROW SHARE 基本相同，但阻止以 SHASRE 方式封锁。当更新、插入或删除时，自动获得行排它封锁。

SHARE：是共享封锁，它允许并行查询，但阻止对已封锁的表进行更新。

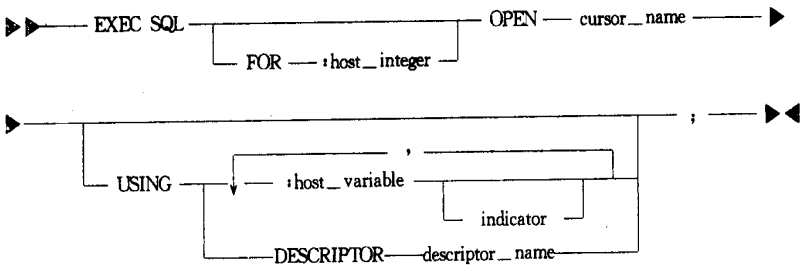
SHARE ROW EXLCLUSIVE：该封锁方式用于查阅整个表并允许其他用户查阅该表中的行，但阻止其他用户以 SHARE 方式封锁该表或更新行。

EXCLUSIVE：该封锁方式允许查询该已被封锁的表，但阻止对该表进行任何其它操作。

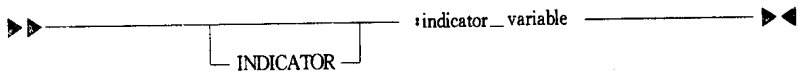
NOWAIT：如果指定的表已被其他用户封锁，则 ORACLE 直接把控制返回给用户程序，同时返回一个指明该表已被其他用户封锁的信息。如果省略该子句，则 ORACLE 一直等到该表可用，而且该等待未设置限制。

17 OPEN

OPEN command：：=



indicator：：=



(a) 功能

该语句打开一个光标；处理有关查询，并用 USING 子句中提供的宿主变量名替换该查询的 WHERE 子句中的虚拟宿主变量。

(b) 关键字及参数说明

cursor_name：是要打开的光标。

USING 子句：指出执行动态 SQL 语句所用的实宿主变量名，这些宿主变量替换动态 SQL 语句中的虚拟宿主变量。

:host_variable：是替换动态 SQL 语句中的虚拟宿主变量的实际宿主变量。

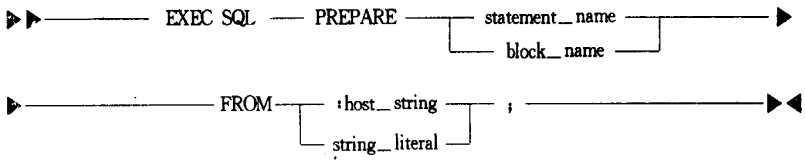
DESCRIPTOR:指定一个描述区，该描述区描述进行替换的实际宿主变量。

该描述必须在前面的 DESCRIBE 语句中初始化。

替换是按顺序逐个进行。在 OPEN 语句中指出的宿主变量名可与动态 SQL 语句中的虚拟宿主变量名不同。

18 PREPARE

PREPARE command::=



(a) 功能

该语句用于分析和命名一个动态 SQL 语句。

(b) 关键字及参数说明

statement_name:标识被分析的动态 SQL 语句。是供预编译程序使用的标识符，而不是宿主变量。

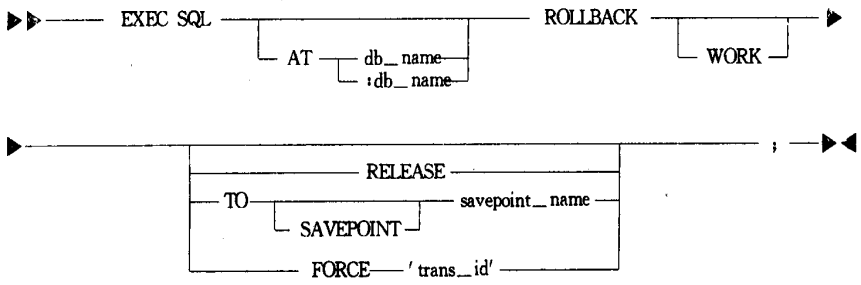
block_name:标识被分析的动态 PL/SQL 块。

:host_string:是一个宿主变量，它包含动态 SQL 语句或 PL/SQL 块的文本。

string_literal: 是一个文字串，表示动态 SQL 语句或 PL/SQL 块的文本。

19 ROLLBACK

ROLLBACK command ::=



(a) 功能

实现事务回滚。

(b) 关键字及参数说明

TO:指出把事务回滚到指定的保留点。

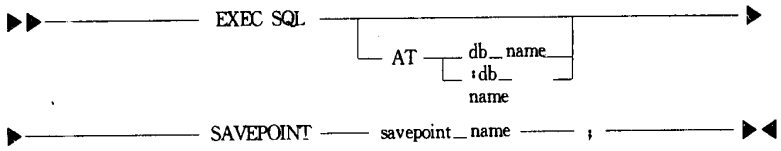
FORCE:

指出手工回滚一个不确定的分布事务。该事务是由包含局部或全程事务 ID 的'text'所标识，要找这些事务的 ID，就查询数据字典视图 DBA_2PC_PENDING。其它参数说明同 COMMIT 语句。

savepoint_name:保留点名。

20 SAVEPOINT

SAVEPOINT COmmand::=



(a) 功能

用于命名和标志事务处理中的一个当前点。

(b) 关键字及参数说明

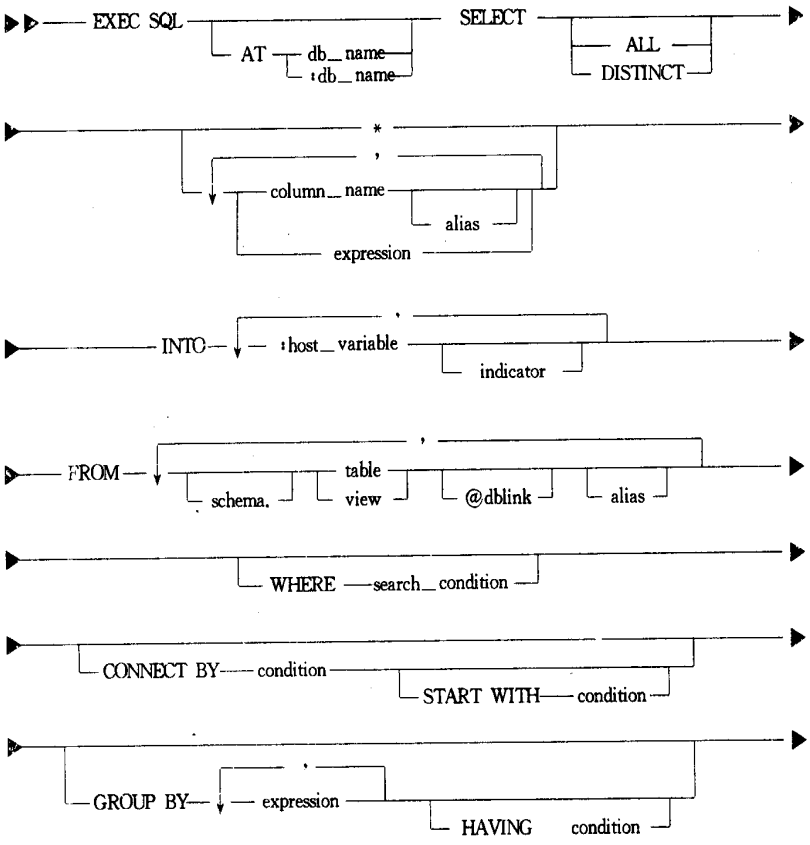
AT 子句：

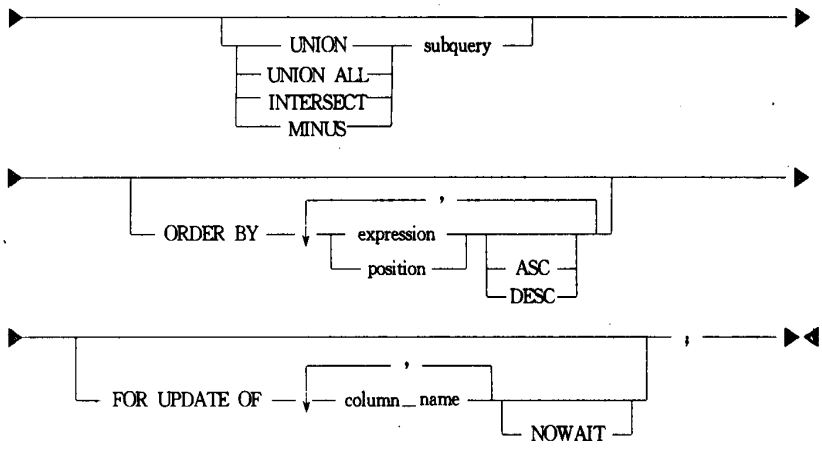
其含义同上所述。

savepoint_name：是要建立的保留点名。是标识符，不是变量。

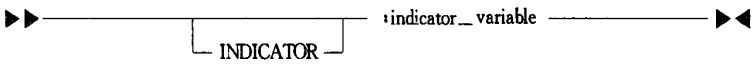
21 SELECT

SELECT command::=





indicator:=



(a) 功能

该语句从一个或多个数据库表、视图或快照中检索数据，并把选到的值赋给输出宿主变量。

(b) 关键字和参数说明

AT 子句:该子句标识被查询的数据库名。

返回值限制:

DISTINCT:消除返回行中的重复行。

ALL:返回全部选择行。缺省值是 ALL。

选择表项:

*:指出选择表或视图中的所有列。

column_name:指定要选择的列名

alias:是列的别名

expression:是选择表达式

INTO 子句:

指出与选择表项相对应的输出宿主变量。其中:host_variable 是输出宿主变量名，indicator 是与相应宿主变量相关的指示器变量。

FROM 子句:

指出从哪些表或视图上进行选择。其中:

schama:是包含被选择的表、视图的模式名。省略 schema 时，则假定是对该用户所拥有的模式进行查询。

table 和 view:指出被选择的表或视图的名字。

dblink:给出数据库的链名

alias:表或视图的别名

WHERE 子句:指出所选择的行应满足的条件。如果省略此子句时，则返回表或视图中的所有行。

CONNECT BY 和 START WITH 子句：

按层次顺序返回行。

GROUP BY：根据表达式的值来分组所选择的行。

HAVING 子句：

限制返回的行组是使指定条件为“真”的行组。如果省略该子句，则 ORACLE 返回所有行组。

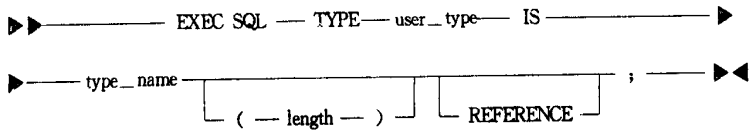
UNION、UNION ALL、INTERSECT 和 MINUS：组合两个 SELECT 语句返回的行。

ORDER BY 子句：使查询返回的行按指定条件排序。

FOR UPDATE OF 子句：用于实现行封锁。

22 TYPE

TYPE command::=



(a)功能

实现用户定义类型等价。

(b) 关键字和参数说明

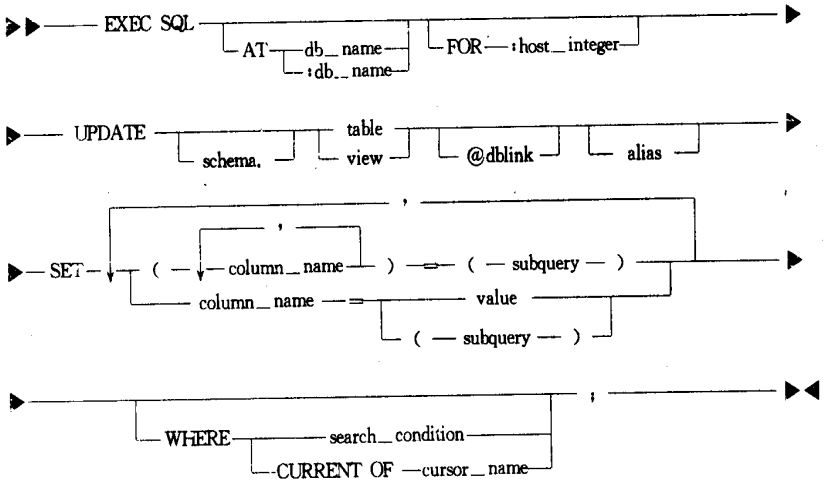
user_type：是用户定义类型。

type_name：ORACLE 外部数据类型。

length。该外部数据类型的长度。

23 UPDATE

UPDATE command::=



(a) 功能

该语句用于修改表或视图的基本表中现有的列值。

(b) 关键字和参数说明

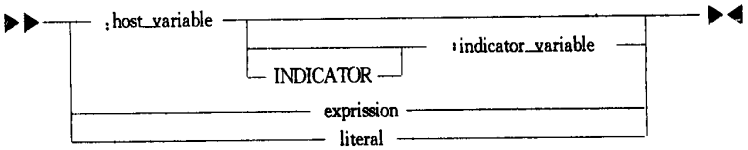
AT 子句:该子句指出要更新的数据库。

table 和 view:table 指出要更新的表名，view 指出要更新的视图基本表。

dblink:指出远程数据库的数据库连接名，要更新的数据库表或视图就分布在该远程数据库上。

column_name:指出要更新的数据库表或视图的列名。如省略 SET 子句中的 column_name，表示表或视图的列值不变。

value:是赋给相应列的新值，该表达式中可能包含宿主变量和可选的指示器变量。其语法如下:



WHERE 子句:该子句指出表或视图中的那些行被更新。如果省略 WHERE 子句，则 ORACLE 更新表或视图的所有行。

search_condition:指出仅更新使逻辑表达式 search_condition 取值为“真”的那些行。该表达式能包含宿主变量或 可选的指示器变量等。

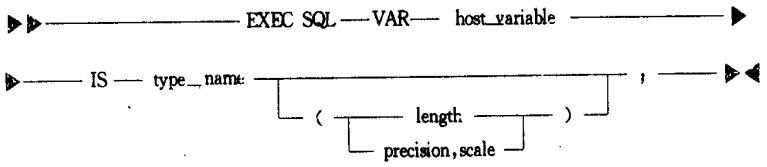
CURRENT OF 子句:

指出仅更新由光标所提取的最当前行。在使用 CURRENT OF 子句时，所用的光标必须用 FOR UPDATE 子句说明，而且必须被打开和定位在一行上。如果 FETCH 语句没执行，或光标没打开，则 CURRENT OF 子句不影响任何一行。CURRENT OF 子句不能与宿主数组或动态 SQL 一起使用。

其它参数说明同 INSERT 语句。

24 VAR

VAR command::=



(a) 功能

用于建立宿主变量与 ORACLE 外部数据类型等价。

(b) 关键字和参数说明

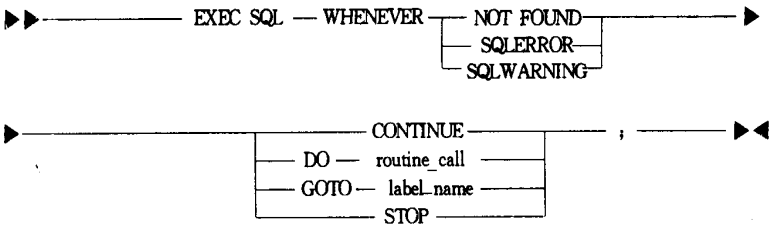
host_variable:主变量名。

type_name:ORACLE 外部数据类型。

length:该外部数据类型的长度。

precision,scale:该外部数据类型的精度和定标。

25 WHENEVER



(a) 功能

WHENEVER 语句用于指定 ORACLE 发现错误、警告或找不到等条件时采取的动作。

(b) 关键字和参数说明

NOT FOUND:当执行 SQL 数据操纵语句时，如果没有一行被处理，SQLCODE 就被设置为+1403 或+100，此时"NOT FOUND"条件满足。

WHENVER command::=

SQLERROR:当 SQL 语句的执行发生错误时，SQLCODE 被设置为负值。此时"SQLERROR"条件满足。

SQLWARNING:当把一个截短列值赋给一个输出宿主变量，或有其它例外发生时，SQLWARN[]的相应元素被设置，此时"SQLWARNING"条件满足。当 MODE={ANSI|ANSI14}时,SQLCA 是可选的。但是,当使用 WHENEVER SQLWARNING 时，必须说明 SQLCA。

CONTINUE:程序继续运行下一个语句(如果可能的话)。这是一个缺省动作，它等价于不使用 WHENEVER 语句。通常使用它来切断条件检查。

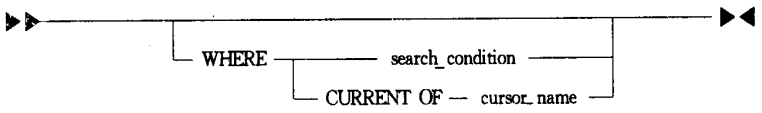
DO routine_call:当 SQL 语句的执行发生错误时，程序把控制传送给一个函数,当该函数执行结束时，又返回发生错误的 SQL 语句之后的语句。但不允许给函数传递参数，也不许有函数返回值。

GOTO lable_name:程序分支到 lable_name 所标的语句。

STOP:停止程序运行但不提交和 回滚事务。

26 WHERE

WHERE command::=



附录 B. OCI 库函数清单(关于 C 语言)

下面按字母顺序列出 C 语言的 OCI 库函数:

```
sword obndra(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, ub2 *alen, ub2 *arcode, ub4 maxsiz,
             ub4 *cursiz, text *fmt, sword fmtl, sword fmtt);

sword obndrn(struct cda_def *cursor, sword sqlvn, ubl *progv,
             sword progvl, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmtl, sword fmtt);

sword obndrv(struct cda_def *cursor, text *sqlvar, sword sqlvl,
             ub1 *progv, sword progvl, sword ftype, sword scale,
             sb2 *indp, text *fmt, sword fmtl, sword fmtt);

sword obreak(struct cda_def *lda);

sword ocan (struct cda_def *cursor);

sword oclose(struct cda_def *cursor);

sword ocof (struct cda_def *lda);

sword ocom (struct cda_def *lda);

sword ocon (struct cda_def *lda);

sword odefin(struct cda_def *cursor, sword pos, ub1 *buf,
             sword buf1, sword ftype, sword scale, sb2 *indp,
             text *fmt, sword fmt1, sword fmtt, ub2 *rlen,
             ub2 *rcode);
sword odessp(struct cda_def *cursor, text *objnam, size_t onlen,
             ub1 *rsvl, size_t rsulln, ub1 *rsv2, size_t rsv2ln,
             ub2 *ovrld, ub2 *pos, ub2 *level, text **argnam,
             ub2 *arnlen, ub2 *dtype, ub1 *defsup, ub1* mode,
             ub4 *dtsiz, sb2 *prec, sb2 *scale, ub1 *radix,
             ub4 *spare, ub4 *arrsiz);
```

```
sword odsc (struct cda_def *cursor, sword pos, sb2 *dbsize,
                    sb2 *fsize, sb2 *rcode, sb2 *dtype, sb1 *buf,
                    sb2 *buf1, sb2 *dsize);

sword odescr(struct cda_def *cursor, sword pos, sb4 *dbsize,
                    sb2 *dbtype, sb1 *cbuf, sb4 *cbuf1, sb4 *dsize,
                    sb2 *prec, sb2 *scale, sb2 *nullok);

sword oerhms(struct cda_def *lda, sb2 rcode, text *buf,
                    sword bufsiz);
sword oermsg(sb2 rcode, text *buf);

sword oexec (struct cda_def *cursor);

sword oexfet(struct cda_def *cursor, ub4 nrows,
                    sword cancel, sword exact);

sword oexn (struct cda_def *cursor,
                    sword iters, sword rowoff);

sword ofen (struct cda_def *cursor, sword nrows);

sword ofetch(struct cda_def *cursor);

sword oflng (struct cda_def *cursor, sword pos, ub1 *buf,
                    sb4 buf1, sword dtype, ub4 *ret1, sb4 offset);

sword ologof(struct cda_def *lda);

sword olon (struct cda_def *lda,text *uid, sword uidl,
                    text *pswd, sword pswdl, sword audit);

sword oopen (struct cda_def *cursor, struct cda_def *lda,
                    text *dbn, sword dbn1, sword arsize,
                    text *uid, sword uidl);

sword oopt (struct cda_def *cursor, sword rbopt, sword waitopt);
```

```
sword oname (struct cda_def *sursor, sword pos, sb1 *tbuf,
                sb2 *tbufl, sb1 *buf, sb2 *bufl);

sword oparse(struct cda_def *cursor, text *sqlstm, sb4 sqllen,
                sword defflg, ub4 lngflg);

sword orlon (struct cda_def *lda, ub1 *hda, text *uid,
                sword uidl, text *pswd, sword pswdl, sword audit);

sword orol (struct cda_def *lda);

sword osql3 (struct cda_def *cda, text *sqlstm, sword sqllen);

sword sqlld2(struct cda_def *lda, text *cname, sb4 *cnlen);

sword sqllda(struct cda_def *lda);
```

附录 C. ORACLE 的保留字和关键字

C.1 保留字

下面列出的字是 ORACLE 的保留字，它们对 ORACLE 有特殊的含义，但不能被重新定义。正因为如此，用户不能用它们来命名数据库对象，如列、表或索引等。

ACCESS	IDENTIFIED	RENAME
ADD	IMMEDIATE	RESOURCE
ALL	IN	REVOKE
ALTER	INCREMENT	ROW
AND	INDEX	ROWID
ANY	INITIAL	ROWLABEL
ARRAYLEN	INSERT	ROWNUM
AS	INTEGER	ROWS
ASC	INTERSECT	SELECT
AUDIT	INTO	SESSION
BETWEEN	IS	SET
BY	LEVEL	SHARE
CHAR	LIKE	SIZE
CHECK	LOCK	SMALLINT
CLUSTER	LONG	SQLBUF
COLUMN	MAXEXTENTS	START
COMMENT	MINUS	SUCCESSFUL
COMPRESS	MODE	SYNONYM
CONNECT	MODIFY	SYSDATE
CREATE	NOAUDIT	TABLE
CURRENT	NOCOMPRESS	THEN
DATE	NOT	TO
DECIMAL	NOTFOUND	TRIGGER
DEFAULT	NOWAIT	UID
DELETE	NULL	UNION
DESC	NUMBER	UNIQUE
DISTINCT	OF	UPDATE
DROP	OFFLINE	USER
ELSE	ON	VALLDATE
EXCLUSIVE	ONLINE	VALUES
EXISTS	OPTION	VARCHAR
FILE	OR	VARCHAR2
FLOAT	ORDER	VIEW
FOR	PCTFREE	WHENEVER
FROM	PRIOR	WHERE
GRANT	PRIVILEGES	WITH
GROUP	PUBLIC	
HAVING	RAW	

C.2 关键字

下面列出的字对 ORACLE 有特殊的含义，但还不是保留字，因此能重新定义。但其中某些可能最终将成为保留字。

ADMIN	ESCAPE	NEW	ROLE
AFTER	EVENTS	NOARCHIVELOG	ROLES
ALLOCATE	EXCEPT	NOCACHE	ROLLBACK
ANALYZE	EXCEPTIONS	NOCYCLE	SAVEPOINT
ARCHIVE	EXPLAIN	NOMAXVALUE	SCHEMA
ARCHIVELOG	EXECUTE	NOMINVALUE	SCN
AUTHORIZATION	EXTENT	NONE	SEGMENT
BACKUP	EXTERNALLY	NOORDER	SEQUENCE
BEGIN	FLUSH	NORESETLOGS	SHARED
BECOME	FREELIST	NORMAL	SNAPSHOT
BEFORE	FREELISTS	NOSORT	SOME
BLOCK	FORCE	NUMERIC	SORT
BODY	FOREIGN	ONLY	STATEMENT_ID
CACHE	FUNCTION	OBJNO	STATISTICS
CANCEL	GROUPS	OFF	STOP
CASADE	INCLUDING	OLD	STORAGE
CHANGE	INDICATOR	ONLY	SWITCH
CHARACTER	INITRANS	OPCODE	SYSTEM
CHECKPOINT	INSTANCE	OPTIMAL	TABLES
CLOSE	INT	OPEN	TABLESPACE
COMMIT	KEY	OWN	TEMPORARY
COMPILE	LAYER	PACKAGE	THREAD
CONSTRAINT	LINK	PARALLEL	TIME
CONSTRAINTS	LISTS	PCTINCREASE	TRACING
CONTENTS	LOGFILE	PCTUSED	TRANSACTION
CONTINUE	MANAGE	PLAN	TRIGGERS
CONTROLFILE	MANUAL	PRIMARY	TRUNCATE
CYCLE	MAX	PRIVATE	UNDER
DATABASE	MAXARCHLOGS	PROCEDURE	UNLIMITED
DATAFILE	MAXDATAFILES	PROFILE	UNTLL
DBA	MAXINSTANCES	QUOTA	USE
DEC	MAXLOGFILES	RBA	USING
DECLARE	MAXLOGMEMBERS	READ	WHEN
DISABLE	MAXTRANS	REAL	WRITE
DISMOUNT	MAXVALUE	RECOVER	WORK
DOUBLE	MIN	REFERENCES	
DUMP	MINEXTENTS	REFERENCING	
EACH	MINVALUE	RESETLOGS	
ENABLE	MOUNT	RESTRICTED	
END	NEXT	REUSE	