ORACLE PL/SQL 编程详解

SQL 语言只是访问、操作数据库的语言，并不是一种具有流程控制的程序设计语言，而只有程序设计语言才能用于应用软件的开发。PL /SQL 是一种高级数据库程序设计语言，该语言专门用于在各种环境下对 ORACLE 数据库进行访问。由于该语言集成于数据库服务器中，所以 PL/SQL 代码可以对数据进行快速高效的处理。除此之外，可以在 ORACLE 数据库的某些客户端工具中，使用 PL/SQL 语言也是该语言的一个特点。本章的主要内容是讨论引入 PL/SQL 语言的必要性和该语言的主要特点，以及了解 PL/SQL 语言的重要性和数据库版本问题。还要介绍一些贯穿全书的更详细的高级概念，并在本章的最后就我们在本书案例中使用的数据库表的若干约定做一说明。

1.1 SQL 与 PL/SQL

1.1.1 什么是 PL/SQL?

PL/SQL 是 Procedure Language & Structured Query Language 的缩写。ORACLE 的 SQL 是支持 ANSI(American national Standards Institute)和 ISO92 (International Standards Organization)标准的产品。PL/SQL 是对 SQL 语言存储过程语言的扩展。从 ORACLE6 以后，ORACLE 的 RDBMS 附带了 PL/SQL。它现在已经成为一种过程处理语言，简称 PL/SQL。目前的 PL/SQL 包括两部分，一部分是数据库引擎部分；另一部分是可嵌入到许多产品（如 C 语言，JAVA 语言等）工具中的独立引擎。可以将这两部分称为：数据库 PL/SQL 和工具 PL/SQL。两者的编程非常相似。都具有编程结构、语法和逻辑机制。工具 PL/SQL 另外还增加了用于支持工具（如 ORACLE Forms）的句法，如：在窗体上设置按钮等。本章主要介绍数据库 PL/SQL 内容。

1.2 PL/SQL 的优点或特征

1.2.1 有利于客户/服务器环境应用的运行

对于客户/服务器环境来说，真正的瓶颈是网络上。无论网络多快，只要客户端与服务器进行大量的数据交换。应用运行的效率自然就回受到影响。如果使用 PL/SQL 进行编程，将这种具有大量数据处理的应用放在服务器端来执行。自然就省去了数据在网上的传输时间。

1.2.2 适合于客户环境

PL/SQL 由于分为数据库 PL/SQL 部分和工具 PL/SQL。对于客户端来说，PL/SQL 可以嵌套到相应的工具中，客户端程序可以执行本地包含 PL/SQL 部分，也可以向服务发 SQL 命令或激活服务器端的 PL/SQL 程序运行。

1.2.3 过程化

PL/SQL 是 Oracle 在标准 SQL 上的过程性扩展，不仅允许在 PL/SQL 程序内嵌入 SQL 语句，而且允许使用各种类型的条件分支语句和循环语句，可以多个应用程序之间共享其解决方案。

1.2.4 模块化

PL/SQL 程序结构是一种描述性很强、界限分明的块结构、嵌套块结构，被分成单独的过程、函数、触发器，且可以把它们组合为程序包，提高程序的模块化能力。

1.2.5 运行错误的可处理性

使用 PL/SQL 提供的异常处理（EXCEPTION），开发人员可集中处理各种 ORACLE 错误和 PL/SQL 错误，或处理系统错误与自定义错误，以增强应用程序的健壮性。

1.2.6 提供大量内置程序包

ORACLE 提供了大量的内置程序包。通过这些程序包能够实现 DBS 的一些低层操作、高级功能，不论对 DBA 还是应用开发人员都具有重要作用。

当然还有其它的一些优点如：更好的性能、可移植性和兼容性、可维护性、易用性与快速性等。

1.3 PL/SQL 可用的 SQL 语句

PL/SQL 是 ORACLE 系统的核心语言，现在 ORACLE 的许多部件都是由 PL/SQL 写成。在 PL/SQL 中可以使用的 SQL 语句有：

INSERT，UPDATE，DELETE，SELECT INTO，COMMIT，ROLLBACK，SAVEPOINT。

提示：在 PL/SQL 中只能用 SQL 语句中的 DML 部分，不能用 DDL 部分，如果要在 PL/SQL 中使用 DDL(如 CREATE table 等)的话，只能以动态的方式来使用。

- ORACLE 的 PL/SQL 组件在对 PL/SQL 程序进行解释时，同时对在其所使用的表名、列名及数据类型进行检查。
- PL/SQL 可以在 SQL*PLUS 中使用。
- PL/SQL 可以在高级语言中使用。
- PL/SQL 可以在 ORACLE 的开发工具中使用(如：SQL Developer 或 Procedure Builder 等)。
- 其它开发工具也可以调用 PL/SQL 编写的过程和函数，如 Power Builder 等都可以调用服务器端的 PL/SQL 过程。

1.4 运行 PL/SQL 程序

PL/SQL 程序的运行是通过 ORACLE 中的一个引擎来进行的。这个引擎可能在 ORACLE 的服务器端，也可能在 ORACLE 应用开发的客户端。引擎执行 PL/SQL 中的过程性语句，然后将 SQL 语句发送给数据库服务器来执行。再将结果返回给执行端

2.1 PL/SQL 块

PL/SQL 程序由三个块组成，即声明部分、执行部分、异常处理部分。

PL/SQL 块的结构如下：

```
DECLARE
  --声明部分: 在此声明 PL/SQL 用到的变量,类型及游标，以及局部的存储过程和函数
BEGIN
  -- 执行部分: 过程及 SQL 语句 , 即程序的主要部分
EXCEPTION
  -- 执行异常部分: 错误处理
END;
```

其中：执行部分不能省略。

PL/SQL 块可以分为三类：

1. **无名块或匿名块（anonymous）：**动态构造，只能执行一次，可调用其它程序，但不能被其它程序调用。

2. **命名块（named）：**是带有名称的匿名块，这个名称就是标签。

3. **子程序（subprogram）：**存储在数据库中的存储过程、函数等。当在数据库上建立好后可以在其它程序中调用它们。

4. **触发器 （Trigger）：**当数据库发生操作时，会触发一些事件，从而自动执行相应的程序。

5. **程序包（package）：**存储在数据库中的一组子程序、变量定义。在包中的子程序可以被其它程序包或子程序调用。但如果声明的是局部子程序，则只能在定义该局部子程序的块中调用该局部子程序。

2.2 PL/SQL 结构

- PL/SQL 块中可以包含子块；
- 子块可以位于 PL/SQL 中的任何部分；
- 子块也即 PL/SQL 中的一条命令；

2.3 标识符

PL/SQL 程序设计中的标识符定义与 SQL 的标识符定义的要求相同。要求和限制有：

- 标识符名不能超过 30 字符；
- 第一个字符必须为字母；
- 不分大小写；
- 不能用'-'(减号);
- 不能是 SQL 保留字。

提示: 一般不要把变量名声明与表中字段名完全一样,如果这样可能得到不正确的结果.

例如：下面的例子将会删除所有的纪录，而不是'EricHu'的记录；

```
DECLARE
  ename varchar2(20) :='EricHu';
BEGIN
 DELETE FROM scott.emp WHERE ename=ename;
END;
```

变量命名在 PL/SQL 中有特别的讲究，建议在系统的设计阶段就要求所有编程人员共同遵守一定的要求，使得整个系统的文档在规范上达到要求。下面是建议的命名方法：

标识符	命名规则	例子
程序变量	V_name	V_name
程序常量	C_Name	C_company_name
游标变量	Cursor_Name	Cursor_Emp
异常标识	E_name	E_too_many
表类型	Name_table_type	Emp_record_type
表	Name_table	Emp
记录类型	Name_record	Emp_record
SQL*Plus 替代变量	P_name	P_sal
绑定变量	G_name	G_year_sal

2.4 PL/SQL 变量类型

在前面的介绍中，有系统的数据类型，也可以自定义数据类型。下表给出 ORACLE 类型和 PL/SQL 中的变量类型的合法使用列表：

2.4.1 变量类型

在 ORACLE8i 中可以使用的变量类型有：

类型	*子类*	*说　　明*	*范　　围*	*ORACLE 限制*
CHAR	Character String Rowid Nchar	定长字符串 民族语言字符集	0→32767 可选,确省=1	2000

VARCHAR2	Varchar, String NVARCHAR2	可变字符串 民族语言字符集	0→32767 4000	4000
BINARY_INTEGER		带符号整数,为整数计算优化性能		
NUMBER(p,s)	Dec Double precision Integer Int Numeric Real Small int	小数, NUMBER 的子类型 高精度实数 整数, NUMBER 的子类型 整数, NUMBER 的子类型 与 NUMBER 等价 与 NUMBER 等价 整数, 比 integer 小		
LONG		变长字符串	0->2147483647	32,767 字节
DATE		日期型	公元前 4712 年 1 月 1 日至公元后 4712 年 12 月 31 日	
BOOLEAN		布尔型	TRUE, FALSE,NULL	不使用
ROWID		存放数据库行号		
UROWID		通用行标识符，字符类型		

例 1. 插入一条记录并显示；

```
DECLARE
  Row_id ROWID;
  info   VARCHAR2(40);
BEGIN
 INSERT INTO scott.dept VALUES (90, '财务室', '海口')
 RETURNING rowid, dname||':'||to_char(deptno)||':'||loc
 INTO row_id, info;
 DBMS_OUTPUT.PUT_LINE('ROWID:'||row_id);
 DBMS_OUTPUT.PUT_LINE(info);
END;
```

其中：

RETURNING 子句用于检索 INSERT 语句中所影响的数据行数，当 INSERT 语句使用 VALUES 子句插入数据时，RETURNING 字句还可将列表达式、ROWID 和 REF 值返回到输出变量中。在使用 RETURNING 子句是应注意以下几点限制：

1．不能与 DML 语句和远程对象一起使用；

2．不能检索 LONG 类型信息；

3．当通过视图向基表中插入数据时，**只能与单基表视图**一起使用。

例 2. 修改一条记录并显示

```
DECLARE
  Row_id ROWID;
  info   VARCHAR2(40);
BEGIN
 UPDATE dept SET deptno=100 WHERE DNAME='财务室'
 RETURNING rowid, dname||':'||to_char(deptno)||':'||loc
 INTO row_id, info;
 DBMS_OUTPUT.PUT_LINE('ROWID:'||row_id);
 DBMS_OUTPUT.PUT_LINE(info);
END;
```

其中：

RETURNING 子句用于检索被修改行的信息。当 UPDATE 语句修改单行数据时，RETURNING 子句可以检索被修改行的 ROWID 和 REF 值，以及行中被修改列的列表达式，并可将他们存储到 PL/SQL 变量或复合变量中；当 UPDATE 语句修改多行数据时，RETURNING 子句可以将被修改行的 ROWID 和 REF 值，以及列表达式值返回到复合变量数组中。在 UPDATE 中使用 RETURNING 子句的限制与 INSERT 语句中对 RETURNING 子句的限制相同。

例 3. 删除一条记录并显示

```
DECLARE
  Row_id ROWID;
  info   VARCHAR2(40);
BEGIN
 DELETE dept WHERE DNAME='办公室'
 RETURNING rowid, dname||':'||to_char(deptno)||':'||loc
 INTO row_id, info;
```

```
 DBMS_OUTPUT.PUT_LINE('ROWID:'||row_id);
 DBMS_OUTPUT.PUT_LINE(info);
END;
```

其中：

RETURNING 子句用于检索被删除行的信息：当 DELETE 语句删除单行数据时，RETURNING 子句可以检索被删除行的 ROWID 和 REF 值，以及被删除列的列表达式，并可将他们存储到 PL/SQL 变量或复合变量中；当 DELETE 语句删除多行数据时，RETURNING 子句可以将被删除行的 ROWID 和 REF 值，以及列表达式值返回到复合变量数组中。在 DELETE 中使用 RETURNING 子句的限制与 INSERT 语句中对 RETURNING 子句的限制相同。

2.4.2 复合类型

ORACLE 在 PL/SQL 中除了提供象前面介绍的各种类型外,还提供一种称为复合类型的类型---记录和表.

2.4.2.1 记录类型

记录类型类似于 C 语言中的结构数据类型，它把逻辑相关的、分离的、基本数据类型的变量组成一个整体存储起来，它必须包括至少一个标量型或 RECORD 数据类型的成员，称作 PL/SQL RECORD 的域(FIELD)，其作用是存放互不相同但逻辑相关的信息。在使用记录数据类型变量时，需要先在声明部分先定义记录的组成、记录的变量，然后在执行部分引用该记录变量本身或其中的成员。

定义记录类型语法如下:

```
TYPE record_name IS RECORD(
 v1 data_type1  [NOT NULL]  [:= default_value ],
 v2 data_type2  [NOT NULL]  [:= default_value ],
 ......
 vn data_typen  [NOT NULL]  [:= default_value ] );
```

例 4 ：

```
DECLARE
 TYPE test_rec IS RECORD(
     Name VARCHAR2(30) NOT NULL := '胡勇',
     Info VARCHAR2(100));
 rec_book test_rec;
BEGIN
 rec_book.Name :='胡勇';
 rec_book.Info :='谈 PL/SQL 编程;';
```

```
  DBMS_OUTPUT.PUT_LINE(rec_book.Name||' '||rec_book.Info);
END;
```

可以用 SELECT 语句对记录变量进行赋值,只要保证记录字段与查询结果列表中的字段相配即可。

例 5 ：

```
DECLARE
--定义与 hr.employees 表中的这几个列相同的记录数据类型
  TYPE RECORD_TYPE_EMPLOYEES IS RECORD(
     f_name   hr.employees.first_name%TYPE,
     h_date   hr.employees.hire_date%TYPE,
     j_id     hr.employees.job_id%TYPE);
--声明一个该记录数据类型的记录变量
  v_emp_record RECORD_TYPE_EMPLOYEES;

BEGIN
  SELECT first_name, hire_date, job_id INTO v_emp_record
  FROM employees
  WHERE employee_id = &emp_id;

  DBMS_OUTPUT.PUT_LINE('雇员名称：'||v_emp_record.f_name
        ||' 雇佣日期：'||v_emp_record.h_date
        ||' 岗位：'||v_emp_record.j_id);
END;
```

一个记录类型的变量只能保存从数据库中查询出的一行记录，若查询出了多行记录，就会出现错误。

2.4.2.2 数组类型

数据是具有相同数据类型的一组成员的集合。每个成员都有一个唯一的下标，它取决于成员在数组中的位置。在 PL/SQL 中，数组数据类型是 VARRAY。

定义 VARRY 数据类型语法如下：

TYPE varray_name IS VARRAY(size) OF element_type [NOT NULL];

varray_name 是 VARRAY 数据类型的名称，size 是下整数，表示可容纳的成员的最大数量，每个成员的数据类型是 element_type。默认成员可以取空值，否则需要使用 NOT NULL 加以限制。对于 VARRAY 数据类型来说，必须经过三个步骤，分别是：

定义、声明、初始化。

例 6 ：

```
DECLARE
--定义一个最多保存 5 个 VARCHAR(25)数据类型成员的 VARRAY 数据类型
   TYPE reg_varray_type IS VARRAY(5) OF VARCHAR(25);
--声明一个该 VARRAY 数据类型的变量
   v_reg_varray REG_VARRAY_TYPE;

BEGIN
--用构造函数语法赋予初值
   v_reg_varray := reg_varray_type
         ('中国', '美国', '英国', '日本', '法国');

   DBMS_OUTPUT.PUT_LINE('地区名称：'||v_reg_varray(1)||'、'
                                    ||v_reg_varray(2)||'、'
                                    ||v_reg_varray(3)||'、'
                                    ||v_reg_varray(4));
      DBMS_OUTPUT.PUT_LINE(' 赋 予 初 值 NULL 的 第 5 个 成 员 的 值 ：
'||v_reg_varray(5));
--用构造函数语法赋予初值后就可以这样对成员赋值
   v_reg_varray(5) := '法国';
   DBMS_OUTPUT.PUT_LINE('第 5 个成员的值：'||v_reg_varray(5));
END;
```

2.4.2.3 使用%TYPE

定义一个变量，其数据类型与已经定义的某个数据变量(尤其是表的某一列)的数据类型相一致，这时可以使用%TYPE。

使用%TYPE 特性的优点在于：

- 所引用的数据库列的数据类型可以不必知道；
- 所引用的数据库列的数据类型可以实时改变，容易保持一致，也不用修改 PL/SQL 程序。

例 7：

```
DECLARE
   -- 用%TYPE 类型定义与表相配的字段
   TYPE T_Record IS RECORD(
        T_no emp.empno%TYPE,
        T_name emp.ename%TYPE,
        T_sal emp.sal%TYPE );
```

```
    -- 声明接收数据的变量
    v_emp T_Record;
BEGIN
    SELECT empno, ename, sal INTO v_emp FROM emp WHERE empno=7788;
    DBMS_OUTPUT.PUT_LINE
      (TO_CHAR(v_emp.t_no)||' '||v_emp.t_name||'   ' || TO_CHAR(v_em
p.t_sal));
END;
```

例 8：

```
DECLARE
    v_empno emp.empno%TYPE :=&no;
    Type t_record is record (
         v_name    emp.ename%TYPE,
         v_sal     emp.sal%TYPE,
         v_date    emp.hiredate%TYPE);
    Rec t_record;
BEGIN
    SELECT ename, sal, hiredate INTO Rec FROM emp WHERE empno=v_em
pno;
     DBMS_OUTPUT.PUT_LINE(Rec.v_name||'---'||Rec.v_sal||'--'||Rec.v
_date);
END;
```

2.4.3 使用%ROWTYPE

PL/SQL 提供%ROWTYPE 操作符，返回一个记录类型，其数据类型和数据库表的数据结构相一致。

使用%ROWTYPE 特性的优点在于：

- 所引用的数据库中列的个数和数据类型可以不必知道；
- 所引用的数据库中列的个数和数据类型可以实时改变，容易保持一致，也不用修改 PL/SQL 程序。

例 9：

```
DECLARE
    v_empno emp.empno%TYPE :=&no;
    rec emp%ROWTYPE;
BEGIN
    SELECT * INTO rec FROM emp WHERE empno=v_empno;
```

```
    DBMS_OUTPUT.PUT_LINE('姓名:'||rec.ename||'工资:'||rec.sal||'工
作时间:'||rec.hiredate);
END;
```

2.4.4 LOB 类型

ORACLE 提供了 LOB (Large OBject)类型，用于存储大的数据对象的类型。ORACLE 目前主要支持 BFILE, BLOB, CLOB 及 NCLOB 类型。

BFILE (Movie)

存放大的二进制数据对象，这些数据文件不放在数据库里，而是放在操作系统的某个目录里，数据库的表里只存放文件的目录。

BLOB(Photo)

存储大的二进制数据类型。变量存储大的二进制对象的位置。大二进制对象的大小<=4GB。

CLOB(Book)

存储大的字符数据类型。每个变量存储大字符对象的位置，该位置指到大字符数据块。大字符对象的大小<=4GB。

NCLOB

存储大的 NCHAR 字符数据类型。每个变量存储大字符对象的位置，该位置指到大字符数据块。大字符对象的大小<=4GB。

2.4.5 BIND 变量

绑定变量是在主机环境中定义的变量。在 PL/SQL 程序中可以使用绑定变量作为他们将要使用的其它变量。为了在 PL/SQL 环境中声明绑定变量，使用命令 VARIABLE。

例如：

```
VARIABLE return_code NUMBER
VARIABLE return_msg VARCHAR2(20)
```

可以通过 SQL*Plus 命令中的 PRINT 显示绑定变量的值。例如：

```
PRINT return_code
PRINT return_msg
```

例 10：

```
VARIABLE result NUMBER;
BEGIN
  SELECT (sal*10)+nvl(comm, 0) INTO :result FROM emp
  WHERE empno=7844;
END;
--然后再执行
PRINT result
```

2.4.6 PL/SQL 表(TABLE)

定义记录表（或索引表）数据类型。它与记录类型相似，但它是对记录类型的扩展。它可以处理多行记录，类似于高级中的二维数组，使得可以在 PL/SQL 中模仿数据库中的表。

定义记录表类型的语法如下：

```
TYPE table_name IS TABLE OF element_type [NOT NULL]
INDEX BY [BINARY_INTEGER | PLS_INTEGER | VARRAY2];
```

关键字 INDEX BY 表示创建一个主键索引，以便引用记录表变量中的特定行。

方法	*描述*
EXISTS(n)	如果集合的第 n 个成员存在，则返回 true
COUNT	返回已经分配了存储空间即赋值了的成员数量
FIRST LAST	FIRST：返回成员的最低下标值 LAST： 返回成员的最高下标值
PRIOR(n)	返回下标为 n 的成员的前一个成员的下标。如果没有则返回 NULL
NEXT(N)	返回下标为 n 的成员的后一个成员的下标。如果没有则返回 NULL
TRIM	TRIM：删除末尾一个成员 TRIM(n) ：删除末尾 n 个成员
DELETE	DELETE：删除所有成员 DELETE(n) ：删除第 n 个成员 DELETE(m, n) ：删除从 n 到 m 的成员
EXTEND	EXTEND：添加一个 null 成员 EXTEND(n)：添加 n 个 null 成员 EXTEND(n,i)：添加 n 个成员，其值与第 i 个成员相同

LIMIT	返回在 varray 类型变量中出现的最高下标值

例 11：

```
DECLARE
  TYPE dept_table_type IS TABLE OF
       dept%ROWTYPE INDEX BY BINARY_INTEGER;
  my_dname_table dept_table_type;
  v_count number(2) :=4;
BEGIN
  FOR int IN 1 .. v_count LOOP
    SELECT * INTO my_dname_table(int) FROM dept WHERE deptno=int*
10;
  END LOOP;
  FOR int IN my_dname_table.FIRST .. my_dname_table.LAST LOOP
   DBMS_OUTPUT.PUT_LINE('Department number: '||my_dname_table(int)
.deptno);
   DBMS_OUTPUT.PUT_LINE('Department name: '|| my_dname_table(int).
dname);
  END LOOP;
END;
```

例 12：按一维数组使用记录表

```
DECLARE
--定义记录表数据类型
   TYPE reg_table_type IS TABLE OF varchar2(25)
   INDEX BY BINARY_INTEGER;
--声明记录表数据类型的变量
   v_reg_table REG_TABLE_TYPE;

BEGIN
   v_reg_table(1) := 'Europe';
   v_reg_table(2) := 'Americas';
   v_reg_table(3) := 'Asia';
   v_reg_table(4) := 'Middle East and Africa';
   v_reg_table(5) := 'NULL';

   DBMS_OUTPUT.PUT_LINE('地区名称：'||v_reg_table (1)||'、'
                                    ||v_reg_table (2)||'、'
                                    ||v_reg_table (3)||'、'
                                    ||v_reg_table (4));
   DBMS_OUTPUT.PUT_LINE('第 5 个成员的值：'||v_reg_table(5));
END;
```

例 13：按二维数组使用记录表

```
DECLARE
--定义记录表数据类型
   TYPE emp_table_type IS TABLE OF employees%ROWTYPE
   INDEX BY BINARY_INTEGER;
--声明记录表数据类型的变量
   v_emp_table EMP_TABLE_TYPE;
BEGIN
   SELECT first_name, hire_date, job_id INTO
    v_emp_table(1).first_name,v_emp_table(1).hire_date, v_emp_tabl
e(1).job_id
   FROM employees WHERE employee_id = 177;
   SELECT first_name, hire_date, job_id INTO
    v_emp_table(2).first_name,v_emp_table(2).hire_date, v_emp_tabl
e(2).job_id
   FROM employees WHERE employee_id = 178;

   DBMS_OUTPUT.PUT_LINE('177 雇员名称：'||v_emp_table(1).first_name
             ||'  雇佣日期：'||v_emp_table(1).hire_date
             ||'  岗位：'||v_emp_table(1).job_id);
   DBMS_OUTPUT.PUT_LINE('178 雇员名称：'||v_emp_table(2).first_name
             ||'  雇佣日期：'||v_emp_table(2).hire_date
             ||'  岗位：'||v_emp_table(2).job_id);
END;
```

2.5 运算符和表达式(数据定义)

2.5.1 关系运算符

运算符	意义
=	等于
<> , != , ~= , ^=	不等于
<	小于
>	大于
<=	小于或等于
>=	大于或等于

2.5.2 一般运算符

运算符	意义
+	加号
-	减号
*	乘号
/	除号
:=	赋值号

=>	关系号
..	范围运算符
||	字符连接符

2.5.3 逻辑运算符

运算符	意义
IS NULL	是空值
BETWEEN AND	介于两者之间
IN	在一列值中间
AND	逻辑与
OR	逻辑或
NOT	取返,如 IS NOT NULL, NOT IN

2.6 变量赋值

在 PL/SQL 编程中，变量赋值是一个值得注意的地方，它的语法如下：

```
variable  := expression ;
```

variable 是一个 PL/SQL 变量, expression 是一个 PL/SQL 表达式.

2.6.1 字符及数字运算特点

空值加数字仍是空值：NULL + < 数字> = NULL

空值加（连接）字符，结果为字符：NULL || <字符串> = < 字符串>

2.6.2 BOOLEAN 赋值

布尔值只有 TRUE, FALSE 及 NULL 三个值。如：

```
DECLARE
  bDone BOOLEAN;
BEGIN
  bDone := FALSE;
  WHILE NOT bDone LOOP
  Null;
  END LOOP;
END;
```

2.6.3 数据库赋值

数据库赋值是通过 SELECT 语句来完成的，每次执行 SELECT 语句就赋值一次，一般要求被赋值的变量与 SELECT 中的列名要一一对应。如：

例 14：

```
DECLARE
  emp_id     emp.empno%TYPE :=7788;
  emp_name   emp.ename%TYPE;
  wages      emp.sal%TYPE;
BEGIN
  SELECT ename, NVL(sal,0) + NVL(comm,0) INTO emp_name, wages
  FROM emp WHERE empno = emp_id;
  DBMS_OUTPUT.PUT_LINE(emp_name||'----'||to_char(wages));
END;
```

提示：不能将 SELECT 语句中的列赋值给布尔变量。

2.6.4 可转换的类型赋值

- **CHAR 转换为 NUMBER：**

使用 TO_NUMBER 函数来完成字符到数字的转换，如：

```
    v_total := TO_NUMBER('100.0') + sal;
```

- **NUMBER 转换为 CHAR：**

使用 TO_CHAR 函数可以实现数字到字符的转换，如：

```
    v_comm := TO_CHAR('123.45') || '元' ;
```

- **字符转换为日期：**

使用 TO_DATE 函数可以实现　字符到日期的转换，如：

```
    v_date := TO_DATE('2001.07.03','yyyy.mm.dd');
```

- **日期转换为字符**

使用 TO_CHAR 函数可以实现日期到字符的转换，如：

```
    v_to_day := TO_CHAR(SYSDATE, 'yyyy.mm.dd hh24:mi:ss') ;
```

2.7 变量作用范围及可见性

在 PL/SQL 编程中，如果在变量的定义上没有做到统一的话，可能会隐藏一些危险的错误，这样的原因主要是变量的作用范围所致。变量的作用域是指变量的有效作用范围，与其它高级语言类似，PL/SQL 的变量作用范围特点是：

- 变量的作用范围是在你所引用的程序单元（块、子程序、包）内。即从声明变量开始到该块的结束。
- 一个变量（标识）只能在你所引用的块内是可见的。
- 当一个变量超出了作用范围，PL/SQL 引擎就释放用来存放该变量的空间（因为它可能不用了）。
- 在子块中重新定义该变量后，它的作用仅在该块内。

例 15：

```
DECLARE
   Emess char(80);
BEGIN

   DECLARE
      V1 NUMBER(4);
   BEGIN
      SELECT empno INTO v1 FROM emp WHERE LOWER(job)='president';
      DBMS_OUTPUT.PUT_LINE(V1);
   EXCEPTION
      When TOO_MANY_ROWS THEN
         DBMS_OUTPUT.PUT_LINE ('More than one president');
   END;

   DECLARE
      V1 NUMBER(4);
   BEGIN
      SELECT empno INTO v1 FROM emp WHERE LOWER(job)='manager';
   EXCEPTION
      When TOO_MANY_ROWS THEN
          DBMS_OUTPUT.PUT_LINE ('More than one manager');
   END;

EXCEPTION
   When others THEN
      Emess:=substr(SQLERRM,1,80);
```

```
      DBMS_OUTPUT.PUT_LINE(emess);
END;
```

2.8 注释

在 PL/SQL 里，可以使用两种符号来写注释，即：

- 使用双‘-‘(减号) 加注释

PL/SQL 允许用 – 来写注释，它的作用范围是只能在一行有效。如：

```
V_Sal  NUMBER(12,2); -- 人员的工资变量。
```

- 使用 /* */ 来加一行或多行注释，如：

```
/*********************************************/

/* 文件名:  department_salary.sql              */

/* 作 者:  Kenny                              */

/* 时 间:  2011-5-9                           */

/*********************************************/
```

提示：被解释后存放在数据库中的 PL/SQL 程序，一般系统自动将程序头部的注释去掉。只有在 PROCEDURE 之后的注释才被保留；另外程序中的空行也自动被去掉。

2.9 简单例子

2.9.1 简单数据插入例子

例 16：

```
/*********************************************/
/* 文件名:  test.sql                          */
/* 说 明:
       一个简单的插入测试，无实际应用。*/
/* 作 者:  kenny                              */
/* 时 间:  2011-5-9                           */
/*********************************************/
DECLARE
  v_ename    VARCHAR2(20) := 'Bill';
  v_sal        NUMBER(7,2) :=1234.56;
  v_deptno    NUMBER(2) := 10;
```

```
  v_empno    NUMBER(4) := 8888;
BEGIN
  INSERT INTO emp ( empno, ename, JOB, sal, deptno , hiredate )
  VALUES (v_empno, v_ename, 'Manager', v_sal, v_deptno,
            TO_DATE('1954.06.09','yyyy.mm.dd') );
  COMMIT;
END;
```

2.9.2 简单数据删除例子

例 17：

```
/*********************************************/
/* 文件名:  test_deletedata.sql      */
/* 说 明:
       简单的删除例子，不是实际应用。 */
/* 作 者:  kenny                     */
/* 时 间:  2011-5-9                  */
/*********************************************/
DECLARE
  v_ename    VARCHAR2(20) := 'Bill';
  v_sal        NUMBER(7,2) :=1234.56;
  v_deptno    NUMBER(2) := 10;
  v_empno    NUMBER(4) := 8888;
BEGIN
  INSERT INTO emp ( empno, ename, JOB, sal, deptno , hiredate )
VALUES ( v_empno, v_ename, 'Manager', v_sal, v_deptno,
TO_DATE('1954.06.09','yyyy.mm.dd') );
  COMMIT;
END;
 DECLARE
  v_empno    number(4) := 8888;
BEGIN
  DELETE FROM emp WHERE empno=v_empno;
  COMMIT;
END;
```

3.1 条件语句

```
IF  〈布尔表达式〉  THEN
    PL/SQL  和  SQL 语句
END  IF;
------------------------
```

```
IF  〈布尔表达式〉  THEN
    PL/SQL  和  SQL 语句
ELSE
    其它语句
END  IF;
------------------------
```

```
IF <布尔表达式> THEN
 PL/SQL 和 SQL 语句
ELSIF < 其它布尔表达式> THEN
 其它语句
ELSIF < 其它布尔表达式> THEN
 其它语句
ELSE
 其它语句
END IF;
```

提示: ELSIF 不能写成 ELSEIF

例 1:

```
DECLARE
   v_empno  employees.employee_id%TYPE :=&empno;
   V_salary employees.salary%TYPE;
   V_comment VARCHAR2(35);
BEGIN
  SELECT salary INTO v_salary FROM employees
  WHERE employee_id = v_empno;
  IF v_salary < 1500 THEN
     V_comment:= '太少了,加点吧~!';
  ELSIF v_salary <3000 THEN
     V_comment:= '多了点,少点吧~!';
  ELSE
     V_comment:= '没有薪水~!';
  END IF;
  DBMS_OUTPUT.PUT_LINE(V_comment);
  exception
    when no_data_found then
      DBMS_OUTPUT.PUT_LINE('没有数据~!');
    when others then
      DBMS_OUTPUT.PUT_LINE(sqlcode || '---' || sqlerrm);
END;
```

例 2:

```
DECLARE
  v_first_name  VARCHAR2(20);
  v_salary NUMBER(7,2);
BEGIN
  SELECT first_name, salary INTO v_first_name, v_salary FROM employees
  WHERE employee_id = &emp_id;
  DBMS_OUTPUT.PUT_LINE(v_first_name||'雇员的工资是'||v_salary);
  IF v_salary < 10000 THEN
    DBMS_OUTPUT.PUT_LINE('工资低于 10000');
  ELSE
    IF 10000 <= v_salary AND v_salary < 20000 THEN
      DBMS_OUTPUT.PUT_LINE('工资在 10000 到 20000 之间');
    ELSE
      DBMS_OUTPUT.PUT_LINE('工资高于 20000');
    END IF;
  END IF;
END;
```

例 3:

```
DECLARE
  v_first_name  VARCHAR2(20);
  v_hire_date DATE;
  v_bonus NUMBER(6,2);
BEGIN
  SELECT first_name, hire_date INTO v_first_name, v_hire_date FROM employees
  WHERE employee_id = &emp_id;
  IF v_hire_date > TO_DATE('01-1 月-90') THEN
    v_bonus := 800;
  ELSIF v_hire_date > TO_DATE('01-1 月-88') THEN
    v_bonus := 1600;
  ELSE
    v_bonus := 2400;
  END IF;
  DBMS_OUTPUT.PUT_LINE(v_first_name||'雇员的雇佣日期是'||v_hire_date
                       ||'、奖金是'||v_bonus);
END;
```

3.2 CASE 表达式

```
CASE  条件表达式
    WHEN  条件表达式结果 1  THEN
          语句段 1
    WHEN  条件表达式结果 2  THEN
          语句段 2
    ......
    WHEN  条件表达式结果 n  THEN
          语句段 n
    [ELSE  条件表达式结果]
END;
```

```
CASE
     WHEN  条件表达式 1  THEN
           语句段 1
     WHEN  条件表达式 2  THEN
           语句段 2
     ......
     WHEN  条件表达式 n  THEN
           语句段 n
     [ELSE  语句段]
 END;
```

例 4:

```
DECLARE
 V_grade char(1) := UPPER('&p_grade');
 V_appraisal VARCHAR2(20);
BEGIN
 V_appraisal :=
 CASE v_grade
  WHEN 'A' THEN 'Excellent'
  WHEN 'B' THEN 'Very Good'
  WHEN 'C' THEN 'Good'
  ELSE 'No such grade'
 END;
 DBMS_OUTPUT.PUT_LINE('Grade:'||v_grade||'  Appraisal: '|| v_appraisal);
END;
```

例 5:

```
DECLARE
  v_first_name employees.first_name%TYPE;
  v_job_id employees.job_id%TYPE;
  v_salary employees.salary%TYPE;
  v_sal_raise NUMBER(3,2);
BEGIN
  SELECT first_name,   job_id,   salary INTO
       v_first_name, v_job_id, v_salary
  FROM employees WHERE employee_id = &emp_id;
  CASE
    WHEN v_job_id = 'PU_CLERK' THEN
      IF v_salary < 3000 THEN v_sal_raise := .08;
      ELSE v_sal_raise := .07;
      END IF;
    WHEN v_job_id = 'SH_CLERK' THEN
      IF v_salary < 4000 THEN v_sal_raise := .06;
      ELSE v_sal_raise := .05;
      END IF;
    WHEN v_job_id = 'ST_CLERK' THEN
      IF v_salary < 3500 THEN v_sal_raise := .04;
      ELSE v_sal_raise := .03;
      END IF;
    ELSE
      DBMS_OUTPUT.PUT_LINE('该岗位不涨工资: '||v_job_id);
  END CASE;
  DBMS_OUTPUT.PUT_LINE(v_first_name||'的岗位是'||v_job_id
                       ||'、的工资是'||v_salary
                       ||'、工资涨幅是'||v_sal_raise);
END;
```

3.3 循环

1. 简单循环

```
LOOP
  要执行的语句;
  EXIT WHEN <条件语句> --条件满足，退出循环语句
END LOOP;
```

例 6.

```
DECLARE
  int NUMBER(2) :=0;
BEGIN
  LOOP
    int := int + 1;
```

```
    DBMS_OUTPUT.PUT_LINE('int 的当前值为:'||int);
    EXIT WHEN int =10;
  END LOOP;
END;
```

2. WHILE 循环

```
WHILE <布尔表达式> LOOP
  要执行的语句;
END LOOP;
```

例 7.

```
DECLARE
 x NUMBER :=1;
BEGIN
  WHILE x<=10 LOOP
    DBMS_OUTPUT.PUT_LINE('X 的当前值为:'||x);
     x:= x+1;
  END LOOP;
END;
```

3. 数字式循环

```
[<<循环标签>>]
FOR 循环计数器 IN [ REVERSE ] 下限 .. 上限 LOOP
 要执行的语句;
END LOOP [循环标签];
```

每循环一次，循环变量自动加 1；使用关键字 REVERSE，循环变量自动减 1。跟在 IN REVERSE 后面的数字必须是从小到大的顺序，而且必须是整数，不能是变量或表达式。可以使用 EXIT 退出循环。

例 8.

```
BEGIN
  FOR int  in 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE('int 的当前值为: '||int);
  END LOOP;
END;
```

例 9.

```
CREATE TABLE temp_table(num_col NUMBER);
```

```
DECLARE
   V_counter NUMBER := 10;
BEGIN
  INSERT INTO temp_table(num_col) VALUES (v_counter );
  FOR v_counter IN 20 .. 25 LOOP
    INSERT INTO temp_table (num_col ) VALUES ( v_counter );
  END LOOP;
  INSERT INTO temp_table(num_col) VALUES (v_counter );
  FOR v_counter IN REVERSE 20 .. 25 LOOP
    INSERT INTO temp_table (num_col ) VALUES ( v_counter );
  END LOOP;
END ;

DROP TABLE temp_table;
```

例 10：

```
DECLARE
  TYPE jobids_varray IS VARRAY(12) OF VARCHAR2(10); --定义一个 VARRAY 数据类型
  v_jobids JOBIDS_VARRAY; --声明一个具有 JOBIDS_VARRAY 数据类型的变量
  v_howmany NUMBER; --声明一个变量来保存雇员的数量

BEGIN
  --用某些 job_id 值初始化数组
  v_jobids := jobids_varray('FI_ACCOUNT', 'FI_MGR', 'ST_CLERK', 'ST_MAN');

  --用 FOR...LOOP...END LOOP 循环使用每个数组成员的值
  FOR i IN v_jobids.FIRST..v_jobids.LAST LOOP

  --针对数组中的每个岗位，决定该岗位的雇员的数量
    SELECT count(*) INTO v_howmany FROM employees WHERE job_id = v_jobids(i);
    DBMS_OUTPUT.PUT_LINE ( '岗位'||v_jobids(i)||
                '总共有'|| TO_CHAR(v_howmany) || '个雇员');
  END LOOP;
END;
```

例 11 在 While 循环中嵌套 loop 循环

```
/*求 100 至 110 之间的素数*/
DECLARE
  v_m NUMBER := 101;
  v_i NUMBER;
  v_n NUMBER := 0;
BEGIN
```

```
  WHILE v_m < 110 LOOP
    v_i := 2;
    LOOP
      IF mod(v_m, v_i) = 0 THEN
        v_i := 0;
        EXIT;
      END IF;

      v_i := v_i + 1;
      EXIT WHEN v_i > v_m - 1;
    END LOOP;

    IF v_i > 0 THEN
      v_n := v_n + 1;
      DBMS_OUTPUT.PUT_LINE('第'|| v_n || '个素数是' || v_m);
    END IF;

    v_m := v_m + 2;
  END LOOP;
END;
```

3.4 标号和 GOTO

PL/SQL 中 GOTO 语句是无条件跳转到指定的标号去的意思。语法如下：

```
GOTO label;
.....
<<label>>   /*标号是用<<   >>括起来的标识符 */
```

注意，在以下地方使用是不合法的，编译时会出错误。

- 跳转到非执行语句前面。
- 跳转到子块中。
- 跳转到循环语句中。
- 跳转到条件语句中。
- 从异常处理部分跳转到执行。
- 从条件语句的一部分跳转到另一部分。

例 12:

```
DECLARE
  V_counter NUMBER := 1;
BEGIN
  LOOP
```

```
    DBMS_OUTPUT.PUT_LINE('V_counter 的当前值为:'||V_counter);
    V_counter := v_counter + 1;
  IF v_counter > 10 THEN
     GOTO labelOffLOOP;
  END IF;
  END LOOP;
  <<labelOffLOOP>>
    DBMS_OUTPUT.PUT_LINE('V_counter 的当前值为:'||V_counter);
END;
```

例 13:

```
DECLARE
  v_i NUMBER := 0;
  v_s NUMBER := 0;
BEGIN
  <<label_1>>
  v_i := v_i + 1;
  IF v_i <= 1000 THEN
     v_s := v_s + v_i;
     GOTO label_1;
  END IF;
  DBMS_OUTPUT.PUT_LINE(v_s);
END;
```

3.5 NULL 语句

在 PL/SQL 程序中，NULL 语句是一个可执行语句，可以用 null 语句来说明“不用做任何事情”的意思，相当于一个占位符或不执行任何操作的空语句，可以使某些语句变得有意义，提高程序的可读性，保证其他语句结构的完整性和正确性。如：

例 14:

```
DECLARE
   ...
BEGIN
   ...
   IF v_num IS NULL THEN
   GOTO labelPrint;
   END IF;
 ...
 <<labelPrint>>
 NULL; --不需要处理任何数据。
END;
```

例 15:

```
DECLARE
  v_emp_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_salary employees.salary%TYPE;
  v_sal_raise NUMBER(3,2);
BEGIN
  v_emp_id := &emp_id;
  SELECT first_name, salary INTO v_first_name, v_salary
  FROM employees WHERE employee_id = v_emp_id;
  IF v_salary <= 3000 THEN
    v_sal_raise := .10;
    DBMS_OUTPUT.PUT_LINE(v_first_name||'的工资是'||v_salary
                         ||'、工资涨幅是'||v_sal_raise);
  ELSE
    NULL;
  END IF;
END;
```

4.1 游标概念

在 PL/SQL 块中执行 SELECT、INSERT、DELETE 和 UPDATE 语句时，ORACLE 会在内存中为其分配上下文区（Context Area），即缓冲区。游标是指向该区的一个指针，或是命名一个工作区（Work Area），或是一种结构化数据类型。它为应用等量齐观提供了一种对具有多行数据查询结果集中的每一行数据分别进行单独处理的方法，是设计嵌入式 SQL 语句的应用程序的常用编程方式。

在每个用户会话中，可以同时打开多个游标，其数量由数据库初始化参数文件中的 OPEN_CURSORS 参数定义。

对于不同的 SQL 语句，游标的使用情况不同：

SQL 语句	*游标*
非查询语句	隐式的
结果是单行的查询语句	隐式的或显示的
结果是多行的查询语句	显示的

4.1.1 处理显式游标

1. 显式游标处理

显式游标处理需四个 PL/SQL 步骤:

- **定义/声明游标：**就是定义一个游标名，以及与其相对应的 SELECT 语句。

  格式：

```
CURSOR cursor_name[(parameter[, parameter]…)]
    [RETURN datatype]
IS
  select_statement;
```

游标参数只能为输入参数，其格式为：

```
parameter_name [IN] datatype [{:= | DEFAULT} expression]
```

在指定数据类型时，不能使用长度约束。如 NUMBER(4),CHAR(10) 等都是错误的。

[RETURN datatype]是可选的，表示游标返回数据的数据。如果选择，则应该严格与 select_statement 中的选择列表在次序和数据类型上匹配。一般是记录数据类型或带“%ROWTYPE”的数据。

- **打开游标：**就是执行游标所对应的 SELECT 语句，将其查询结果放入工作区，并且指针指向工作区的首部，标识游标结果集合。如果游标查询语句中带有 FOR UPDATE 选项，OPEN 语句还将锁定数据库表中游标结果集合对应的数据行。
  格式：

```
OPEN cursor_name[([parameter =>] value[, [parameter =>] value]…)];
```

在向游标传递参数时，可以使用与函数参数相同的传值方法，即位置表示法和名称表示法。PL/SQL 程序不能用 OPEN 语句重复打开一个游标。

- **提取游标数据：**就是检索结果集合中的数据行，放入指定的输出变量中。
  格式：

```
FETCH cursor_name INTO {variable_list | record_variable };
```

执行 FETCH 语句时，每次返回一个数据行，然后自动将游标移动指向下一个数据行。当检索到最后一行数据时，如果再次执行 FETCH 语句，将操作失败，并将游标属性%NOTFOUND 置为 TRUE。所以每次执行完 FETCH 语句后，检查游标属性%NOTFOUND 就可以判断 FETCH 语句是否执行成功并返回一个数据行，以便确定是否给对应的变量赋了值。

- 对该记录进行处理；
- 继续处理，直到活动集合中没有记录；
- **关闭游标：**当提取和处理完游标结果集合数据后，应及时关闭游标，以释放该游标所占用的系统资源，并使该游标的工作区变成无效，不能再使用 FETCH 语句取其中数据。关闭后的游标可以使用 OPEN 语句重新打开。
  格式：

```
CLOSE cursor_name;
```

注：定义的游标不能有 INTO 子句。

例 1. 查询前 10 名员工的信息。

```
DECLARE
  CURSOR c_cursor
  IS SELECT first_name || last_name, Salary
  FROM EMPLOYEES
```

```
  WHERE rownum<11;
  v_ename  EMPLOYEES.first_name%TYPE;
  v_sal    EMPLOYEES.Salary%TYPE;
BEGIN
  OPEN c_cursor;
  FETCH c_cursor INTO v_ename, v_sal;
  WHILE c_cursor%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(v_ename||'---'||to_char(v_sal) );
    FETCH c_cursor INTO v_ename, v_sal;
  END LOOP;
  CLOSE c_cursor;
END;
```

例 2. 游标参数的传递方法。

```
DECLARE
  DeptRec    DEPARTMENTS%ROWTYPE;
  Dept_name  DEPARTMENTS.DEPARTMENT_NAME%TYPE;
  Dept_loc   DEPARTMENTS.LOCATION_ID%TYPE;
  CURSOR c1 IS
  SELECT DEPARTMENT_NAME, LOCATION_ID FROM DEPARTMENTS
  WHERE DEPARTMENT_ID <= 30;

  CURSOR c2(dept_no NUMBER DEFAULT 10) IS
    SELECT DEPARTMENT_NAME, LOCATION_ID FROM DEPARTMENTS
    WHERE DEPARTMENT_ID <= dept_no;
  CURSOR c3(dept_no NUMBER DEFAULT 10) IS
    SELECT * FROM DEPARTMENTS
    WHERE DEPARTMENTS.DEPARTMENT_ID <=dept_no;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO dept_name, dept_loc;
    EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(dept_name||'---'||dept_loc);
    END LOOP;
    CLOSE c1;

    OPEN c2;
    LOOP
      FETCH c2 INTO dept_name, dept_loc;
      EXIT WHEN c2%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(dept_name||'---'||dept_loc);
    END LOOP;
```

```
  CLOSE c2;

  OPEN c3(dept_no =>20);
  LOOP
    FETCH c3 INTO deptrec;
    EXIT WHEN c3%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE(deptrec.DEPARTMENT_ID||'---'||deptrec.DEPARTMENT_NAME||'
---'||deptrec.LOCATION_ID);
  END LOOP;
  CLOSE c3;
END;
```

2.游标属性

游标属性	*属性作用*
Cursor_name%FOUND	布尔型属性，当最近一次提取游标操作 FETCH 成功则为 TRUE,否则为 FALSE；
Cursor_name%NOTFOUND	布尔型属性，与%FOUND 相反；
Cursor_name%ISOPEN	布尔型属性，当游标已打开时返回 TRUE；
Cursor_name%ROWCOUNT	数字型属性，返回已从游标中读取的记录数。

例 3：给工资低于 1200 的员工增加工资 50。

```
DECLARE
  v_empno  EMPLOYEES.EMPLOYEE_ID%TYPE;
  v_sal    EMPLOYEES.Salary%TYPE;
  CURSOR c_cursor IS SELECT EMPLOYEE_ID, Salary FROM EMPLOYEES;
BEGIN
  OPEN c_cursor;
  LOOP
    FETCH c_cursor INTO v_empno, v_sal;
    EXIT WHEN c_cursor%NOTFOUND;
    IF v_sal<=1200 THEN
       UPDATE EMPLOYEES SET Salary=Salary+50 WHERE EMPLOYEE_ID=v_empno;
       DBMS_OUTPUT.PUT_LINE('编码为'||v_empno||'工资已更新!');
    END IF;
  DBMS_OUTPUT.PUT_LINE('记录数:'|| c_cursor %ROWCOUNT);
  END LOOP;
  CLOSE c_cursor;
END;
```

例 4：没有参数且没有返回值的游标。

```
DECLARE
  v_f_name employees.first_name%TYPE;
  v_j_id   employees.job_id%TYPE;
  CURSOR c1     --声明游标,没有参数没有返回值
  IS
    SELECT first_name, job_id FROM employees
    WHERE department_id = 20;
BEGIN
  OPEN c1;      --打开游标
  LOOP
    FETCH c1 INTO v_f_name, v_j_id;   --提取游标
    IF c1%FOUND THEN
      DBMS_OUTPUT.PUT_LINE(v_f_name||'的岗位是'||v_j_id);
    ELSE
      DBMS_OUTPUT.PUT_LINE('已经处理完结果集了');
      EXIT;
    END IF;
  END LOOP;
  CLOSE c1;  --关闭游标
END;
```

例 5：有参数且没有返回值的游标。

```
DECLARE
  v_f_name employees.first_name%TYPE;
  v_h_date employees.hire_date%TYPE;
  CURSOR c2(dept_id NUMBER, j_id VARCHAR2) --声明游标,有参数没有返回值
  IS
    SELECT first_name, hire_date FROM employees
    WHERE department_id = dept_id AND job_id = j_id;
BEGIN
  OPEN c2(90, 'AD_VP'); --打开游标,传递参数值
  LOOP
    FETCH c2 INTO v_f_name, v_h_date;   --提取游标
    IF c2%FOUND THEN
      DBMS_OUTPUT.PUT_LINE(v_f_name||'的雇佣日期是'||v_h_date);
    ELSE
      DBMS_OUTPUT.PUT_LINE('已经处理完结果集了');
      EXIT;
    END IF;
  END LOOP;
  CLOSE c2;  --关闭游标
END;
```

例 6：有参数且有返回值的游标。

```
DECLARE
  TYPE emp_record_type IS RECORD(
      f_name   employees.first_name%TYPE,
      h_date   employees.hire_date%TYPE);
  v_emp_record EMP_RECORD_TYPE;

  CURSOR c3(dept_id NUMBER, j_id VARCHAR2) --声明游标,有参数有返回值
       RETURN EMP_RECORD_TYPE
  IS
    SELECT first_name, hire_date FROM employees
    WHERE department_id = dept_id AND job_id = j_id;
BEGIN
  OPEN c3(j_id => 'AD_VP', dept_id => 90);  --打开游标,传递参数值
  LOOP
    FETCH c3 INTO v_emp_record;    --提取游标
    IF c3%FOUND THEN
      DBMS_OUTPUT.PUT_LINE(v_emp_record.f_name||'的雇佣日期是'
                  ||v_emp_record.h_date);
    ELSE
      DBMS_OUTPUT.PUT_LINE('已经处理完结果集了');
      EXIT;
    END IF;
  END LOOP;
  CLOSE c3;   --关闭游标
END;
```

例 7：基于游标定义记录变量。

```
DECLARE
  CURSOR c4(dept_id NUMBER, j_id VARCHAR2) --声明游标,有参数没有返回值
  IS
    SELECT first_name f_name, hire_date FROM employees
    WHERE department_id = dept_id AND job_id = j_id;
  --基于游标定义记录变量，比声明记录类型变量要方便，不容易出错
   v_emp_record c4%ROWTYPE;
BEGIN
  OPEN c4(90, 'AD_VP');  --打开游标,传递参数值
  LOOP
    FETCH c4 INTO v_emp_record;    --提取游标
    IF c4%FOUND THEN
      DBMS_OUTPUT.PUT_LINE(v_emp_record.f_name||'的雇佣日期是'
                  ||v_emp_record.hire_date);
```

```
    ELSE
      DBMS_OUTPUT.PUT_LINE('已经处理完结果集了');
      EXIT;
    END IF;
  END LOOP;
  CLOSE c4;  --关闭游标
END;
```

3. 游标的 FOR 循环

PL/SQL 语言提供了游标 FOR 循环语句，自动执行游标的 OPEN、FETCH、CLOSE 语句和循环语句的功能；当进入循环时，游标 FOR 循环语句自动打开游标，并提取第一行游标数据，当程序处理完当前所提取的数据而进入下一次循环时，游标 FOR 循环语句自动提取下一行数据供程序处理，当提取完结果集合中的所有数据行后结束循环，并自动关闭游标。

格式：

```
FOR index_variable IN cursor_name[(value[, value]…)] LOOP
 -- 游标数据处理代码
END LOOP;
```

其中：

index_variable 为游标 FOR 循环语句隐含声明的索引变量，该变量为记录变量，其结构与游标查询语句返回的结构集合的结构相同。在程序中可以通过引用该索引记录变量元素来读取所提取的游标数据，index_variable 中各元素的名称与游标查询语句选择列表中所制定的列名相同。如果在游标查询语句的选择列表中存在计算列，则必须为这些计算列指定别名后才能通过游标 FOR 循环语句中的索引变量来访问这些列数据。

注：不要在程序中对游标进行人工操作；不要在程序中定义用于控制 FOR 循环的记录。

例 8：

```
DECLARE
  CURSOR c_sal IS SELECT employee_id, first_name || last_name ename, salary
  FROM employees ;
BEGIN
  --隐含打开游标
  FOR v_sal IN c_sal LOOP
  --隐含执行一个 FETCH 语句
      DBMS_OUTPUT.PUT_LINE(to_char(v_sal.employee_id)||'---'|| v_sal.ename||'---'||to_char(v
_sal.salary)) ;
  --隐含监测 c_sal%NOTFOUND
  END LOOP;
--隐含关闭游标
END;
```

例 9：当所声明的游标带有参数时，通过游标 FOR 循环语句为游标传递参数。

```
DECLARE
 CURSOR c_cursor(dept_no NUMBER DEFAULT 10)
 IS
   SELECT department_name, location_id FROM departments WHERE department_id <= dept_no
;
BEGIN
   DBMS_OUTPUT.PUT_LINE('当 dept_no 参数值为 30：');
   FOR c1_rec IN c_cursor(30) LOOP       DBMS_OUTPUT.PUT_LINE(c1_rec.department_name||'--
-'||c1_rec.location_id);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(CHR(10)||'使用默认的 dept_no 参数值 10：');
   FOR c1_rec IN c_cursor LOOP       DBMS_OUTPUT.PUT_LINE(c1_rec.department_name||'---'||
c1_rec.location_id);
   END LOOP;
END;
```

例 10：PL/SQL 还允许在游标 FOR 循环语句中使用子查询来实现游标的功能。

```
BEGIN
   FOR c1_rec IN(SELECT department_name, location_id FROM departments) LOOP       DBMS_O
UTPUT.PUT_LINE(c1_rec.department_name||'---'||c1_rec.location_id);
   END LOOP;
END;
```

4.1.2 处理隐式游标

显式游标主要是用于对查询语句的处理，尤其是在查询结果为多条记录的情况下；而对于非查询语句，如修改、删除操作，则由 ORACLE 系统自动地为这些操作设置游标并创建其工作区，这些由系统隐含创建的游标称为隐式游标，隐式游标的名字为 SQL，这是由 ORACLE 系统定义的。对于隐式游标的操作，如定义、打开、取值及关闭操作，都由 ORACLE 系统自动地完成，无需用户进行处理。用户只能通过隐式游标的相关属性，来完成相应的操作。在隐式游标的工作区中，所存放的数据是与用户自定义的显示游标无关的、最新处理的一条 SQL 语句所包含的数据。

格式调用为： SQL%

注：INSERT, UPDATE, DELETE, SELECT 语句中不必明确定义游标。

隐式游标属性

属性	*值*	*SELECT*	*INSERT*	*UPDATE*	*DELETE*
SQL%ISOPEN		FALSE	FALSE	FALSE	FALSE
SQL%FOUND	TRUE	有结果		成功	成功
SQL%FOUND	FALSE	没结果		失败	失败

SQL%NOTFUOND	TRUE	没结果		失败	失败
SQL%NOTFOUND	FALSE	有结果		成功	失败
SQL%ROWCOUNT		返回行数，只为 1	插入的行数	修改的行数	删除的行数

例 11: 删除 EMPLOYEES 表中某部门的所有员工，如果该部门中已没有员工，则在 DEPARTMENT 表中删除该部门。

```
DECLARE
   V_deptno department_id%TYPE :=&p_deptno;
BEGIN
   DELETE FROM employees WHERE department_id=v_deptno;
   IF SQL%NOTFOUND THEN
      DELETE FROM departments WHERE department_id=v_deptno;
   END IF;
END;
```

例 12: 通过隐式游标 SQL 的%ROWCOUNT 属性来了解修改了多少行。

```
DECLARE
  v_rows NUMBER;
BEGIN
--更新数据
  UPDATE employees SET salary = 30000
  WHERE department_id = 90 AND job_id = 'AD_VP';
--获取默认游标的属性值
  v_rows := SQL%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE('更新了'||v_rows||'个雇员的工资');
--回退更新，以便使数据库的数据保持原样
  ROLLBACK;
END;
```

4.1.3 关于 NO_DATA_FOUND 和 %NOTFOUND 的区别

SELECT … INTO 语句触发 NO_DATA_FOUND；
当一个显式游标的 WHERE 子句未找到时触发%NOTFOUND；
当 UPDATE 或 DELETE 语句的 WHERE 子句未找到时触发 SQL%NOTFOUND；在提取循环中要用 %NOTFOUND 或%FOUND 来确定循环的退出条件，不要用 NO_DATA_FOUND.**4.1.4 使用游标更新和删除数据**
游标修改和删除操作是指在游标定位下，修改或删除表中指定的数据行。这时，要求游标查询语句中必须使用 FOR UPDATE 选项，以便在打开游标时锁定游标结果集合在表中对应数据行的所有列和部分列。

为了对正在处理(查询)的行不被另外的用户改动，ORACLE 提供一个 FOR UPDATE 子句来对所选择的行进行锁住。该需求迫使 ORACLE 锁定游标结果集合的行，可以防止其他事务处理更新或删除相同的行，直到您的事务处理提交或回退为止。

语法：

```
SELECT column_list FROM table_list FOR UPDATE [OF column[, column]…] [NOWAIT]
```

如果另一个会话已对活动集中的行加了锁，那么 SELECT FOR UPDATE 操作一直等待到其它的会话释放这些锁后才继续自己的操作，对于这种情况，当加上 **NOWAIT** 子句时，如果这些行真的被另一个会话锁定，则 OPEN 立即返回并给出：

```
ORA-0054 : resource busy  and  acquire with nowait specified.
```

如果使用 FOR UPDATE 声明游标，则可在 DELETE 和 UPDATE 语句中使用 `WHERE CURRENT OF cursor_name` 子句，修改或删除游标结果集合当前行对应的数据库表中的数据行。

例 13：从 EMPLOYEES 表中查询某部门的员工情况，将其工资最低定为 1500；

```
DECLARE
  V_deptno employees.department_id%TYPE :=&p_deptno;
  CURSOR emp_cursor
 IS
 SELECT employees.employee_id, employees.salary
  FROM employees WHERE employees.department_id=v_deptno
 FOR UPDATE NOWAIT;
BEGIN
  FOR emp_record IN emp_cursor LOOP
  IF emp_record.salary < 1500 THEN
    UPDATE employees SET salary=1500
  WHERE CURRENT OF emp_cursor;
  END IF;
  END LOOP;
--  COMMIT;
END;
```

例 14：将 EMPLOYEES 表中部门编码为 90、岗位为 AD_VP 的雇员的工资都更新为 2000 元；

```
DECLARE
    v_emp_record employees%ROWTYPE;
    CURSOR c1
    IS
        SELECT * FROM employees FOR UPDATE;
```

```
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_emp_record;
        EXIT WHEN c1%NOTFOUND;
        IF v_emp_record.department_id = 90 AND
            v_emp_record.job_id = 'AD_VP'
        THEN
            UPDATE employees SET salary = 20000
            WHERE CURRENT OF c1;    --更新当前游标行对应的数据行
        END IF;
    END LOOP;
    COMMIT;     --提交已经修改的数据
    CLOSE c1;
END;
```

4.2 游标变量

与 游标一样，游标变量也是一个指向多行查询结果集合中当前数据行的指针。但与游标不同的是，游标变量是动态的，而游标是静态的。游标只能与指定的查询相连， 即固定指向一个查询的内存处理区域，而游标变量则可与不同的查询语句相连，它可以指向不同查询语句的内存处理区域（但不能同时指向多个内存处理区域，在某 一时刻只能与一个查询语句相连），只要这些查询语句的返回类型兼容即可。

4.2.1 声明游标变量

游标变量为一个指针，它属于参照类型，所以在声明游标变量类型之前必须先定义游标变量类型。在 PL/SQL 中，可以在块、子程序和包的声明区域内定义游标变量类型。
语法格式为：

```
TYPE ref_type_name IS REF CURSOR
 [ RETURN return_type];
```

其中：ref_type_name 为新定义的游标变量类型名称；
  return_type 为游标变量的返回值类型，它必须为记录变量。
在定义游标变量类型时，可以采用强类型定义和弱类型定义两种。强类型定义必须指定游标变量的返回值类型，而弱类型定义则不说明返回值类型。
    声明一个游标变量的两个步骤：
    步骤一：定义一个 REF CURSOU 数据类型，如：
    TYPE ref_cursor_type IS REF CURSOR;
    步骤二：声明一个该数据类型的游标变量，如：
    cv_ref REF_CURSOR_TYPE;

例：创建两个强类型定义游标变量和一个弱类型游标变量：

```
DECLARE
  TYPE deptrecord IS RECORD(
```

```
    Deptno departments.department_id%TYPE,
    Dname departments.department_name%TYPE,
    Loc departments.location_id%TYPE
  );
  TYPE deptcurtype IS REF CURSOR RETURN departments%ROWTYPE;
  TYPE deptcurtyp1 IS REF CURSOR RETURN deptrecord;
  TYPE curtype IS REF CURSOR;
  Dept_c1 deptcurtype;
  Dept_c2 deptcurtyp1;
  Cv curtype;
```

4.2.2 游标变量操作

与游标一样，游标变量操作也包括打开、提取和关闭三个步骤。

1. 打开游标变量

打开游标变量时使用的是 OPEN…FOR 语句。格式为：

```
OPEN {cursor_variable_name | :host_cursor_variable_name}
FOR select_statement;
```

其中：cursor_variable_name 为游标变量，host_cursor_variable_name 为 PL/SQL 主机环境（如 OCI: ORACLE Call Interface，Pro*c 程序等）中声明的游标变量。OPEN…FOR 语句可以在关闭当前的游标变量之前重新打开游标变量，而不会导致 CURSOR_ALREAD_OPEN 异常错误。新打开游标变量时，前一个查询的内存处理区将被释放。

2. 提取游标变量数据

使用 FETCH 语句提取游标变量结果集合中的数据。格式为：

```
FETCH {cursor_variable_name | :host_cursor_variable_name}
INTO {variable [, variable]… | record_variable};
```

其中：cursor_variable_name 和 host_cursor_variable_name 分别为游标变量和宿主游标变量名称；variable 和 record_variable 分别为普通变量和记录变量名称。

3. 关闭游标变量

CLOSE 语句关闭游标变量，格式为：

```
CLOSE {cursor_variable_name | :host_cursor_variable_name}
```

其中：cursor_variable_name 和 host_cursor_variable_name 分别为游标变量和宿主游标变量名称，如果应用程序试图关闭一个未打开的游标变量，则将导致 INVALID_CURSOR 异常错误。

例 15：强类型参照游标变量类型

```
DECLARE
  TYPE emp_job_rec IS RECORD(
```

```
    Employee_id employees.employee_id%TYPE,
    Employee_name employees.first_name%TYPE,
    Job_title employees.job_id%TYPE
  );
  TYPE emp_job_refcur_type IS REF CURSOR RETURN emp_job_rec;
  Emp_refcur emp_job_refcur_type ;
  Emp_job emp_job_rec;
BEGIN
  OPEN emp_refcur FOR
   SELECT employees.employee_id, employees.first_name||employees.last_name, employees.job_id
 FROM employees
 ORDER BY employees.department_id;

  FETCH emp_refcur INTO emp_job;
  WHILE emp_refcur%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(emp_job.employee_id||': '||emp_job.employee_name||' is a '||emp_job.job_title);
  FETCH emp_refcur INTO emp_job;
  END LOOP;
END;
```

例 16：弱类型参照游标变量类型

```
PROMPT
PROMPT 'What table would you like to see?'
ACCEPT tab PROMPT '(D)epartment, or (E)mployees:'

DECLARE
  Type refcur_t IS REF CURSOR;
  Refcur refcur_t;
  TYPE sample_rec_type IS RECORD (
    Id number,
    Description VARCHAR2 (30)
  );
  sample sample_rec_type;
  selection varchar2(1) := UPPER (SUBSTR ('&tab', 1, 1));
BEGIN
  IF selection='D' THEN
    OPEN refcur FOR
  SELECT departments.department_id, departments.department_name FROM departments;
    DBMS_OUTPUT.PUT_LINE('Department data');
  ELSIF selection='E' THEN
    OPEN refcur FOR
```

```
   SELECT employees.employee_id, employees.first_name||' is a '||employees.job_id FROM emp
loyees;
      DBMS_OUTPUT.PUT_LINE('Employee data');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Please enter ''D'' or ''E''');
      RETURN;
   END IF;
   DBMS_OUTPUT.PUT_LINE('----------------------');
   FETCH refcur INTO sample;
   WHILE refcur%FOUND LOOP
      DBMS_OUTPUT.PUT_LINE(sample.id||': '||sample.description);
      FETCH refcur INTO sample;
   END LOOP;
   CLOSE refcur;
END;
```

例 17：使用游标变量（没有 RETURN 子句）

```
DECLARE
--定义一个游标数据类型
  TYPE emp_cursor_type IS REF CURSOR;
--声明一个游标变量
  c1 EMP_CURSOR_TYPE;
--声明两个记录变量
  v_emp_record employees%ROWTYPE;
  v_reg_record regions%ROWTYPE;

BEGIN
  OPEN c1 FOR SELECT * FROM employees WHERE department_id = 20;
  LOOP
    FETCH c1 INTO v_emp_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_emp_record.first_name||'的雇佣日期是'
                  ||v_emp_record.hire_date);
  END LOOP;
--将同一个游标变量对应到另一个 SELECT 语句
  OPEN c1 FOR SELECT * FROM regions WHERE region_id IN（1，2）;
  LOOP
    FETCH c1 INTO v_reg_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_reg_record.region_id||'表示'
                  ||v_reg_record.region_name);
  END LOOP;
```

```
  CLOSE c1;
END;
```

例 18：使用游标变量（有 RETURN 子句）

```
DECLARE
--定义一个与 employees 表中的这几个列相同的记录数据类型
  TYPE emp_record_type IS RECORD(
      f_name   employees.first_name%TYPE,
      h_date   employees.hire_date%TYPE,
      j_id     employees.job_id%TYPE);
--声明一个该记录数据类型的记录变量
  v_emp_record EMP_RECORD_TYPE;
--定义一个游标数据类型
  TYPE emp_cursor_type IS REF CURSOR
      RETURN EMP_RECORD_TYPE;
--声明一个游标变量
  c1 EMP_CURSOR_TYPE;
BEGIN
  OPEN c1 FOR SELECT first_name, hire_date, job_id
          FROM employees WHERE department_id = 20;
  LOOP
    FETCH c1 INTO v_emp_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('雇员名称：'||v_emp_record.f_name
          ||' 雇佣日期：'||v_emp_record.h_date
          ||' 岗位：'||v_emp_record.j_id);
  END LOOP;
  CLOSE c1;
END;
```

即使是写得最好的 PL/SQL 程序也会遇到错误或未预料到的事件。一个优秀的程序都应该能够正确处理各种出错情况，并尽可能从错误中恢复。任何 ORACLE 错误（报告为 ORA-xxxxx 形式的 Oracle 错误号）、PL/SQL 运行错误或用户定义条件（不一写是错误），都可以。当然了，PL/SQL 编译错误不能通过 PL/SQL 异常处理来处理，因为这些错误发生在 PL/SQL 程序执行之前。

ORACLE 提供异常情况(EXCEPTION)和异常处理(EXCEPTION HANDLER)来实现错误处理。

5.1 异常处理概念

异常情况处理(EXCEPTION)是用来处理正常执行过程中未预料的事件,程序块的异常处理预定义的错误和自定义错误,由于 PL/SQL 程序块一旦产生异常而没有指出如何处理时,程序就会自动终止整个程序运行.

有三种类型的异常错误：

1． 预定义（Predefined)错误

ORACLE 预定义的异常情况大约有 24 个。对这种异常情况的处理，无需在程序中定义，由 ORACLE 自动将其引发。

2． 非预定义（Predefined)错误

即其他标准的 ORACLE 错误。对这种异常情况的处理，需要用户在程序中定义，然后由 ORACLE 自动将其引发。

3． 用户定义(User_define) 错误

程序执行过程中，出现编程人员认为的非正常情况。对这种异常情况的处理，需要用户在程序中定义，然后显式地在程序中将其引发。

异常处理部分一般放在 PL/SQL 程序体的后半部,结构为:

```
EXCEPTION
  WHEN first_exception THEN  <code to handle first exception >
  WHEN second_exception THEN  <code to handle second exception >
  WHEN OTHERS THEN  <code to handle others exception >
END;
```

异常处理可以按任意次序排列,但 OTHERS 必须放在最后.

5.1.1 预定义的异常处理

预定义说明的部分 ORACLE 异常错误

错误号	异常错误信息名称	说明
ORA-0001	Dup_val_on_index	违反了唯一性限制
ORA-0051	Timeout-on-resource	在等待资源时发生超时
ORA-0061	Transaction-backed-out	由于发生死锁事务被撤消
ORA-1001	Invalid-CURSOR	试图使用一个无效的游标
ORA-1012	Not-logged-on	没有连接到 ORACLE
ORA-1017	Login-denied	无效的用户名/口令
ORA-1403	No_data_found	SELECT INTO 没有找到数据
ORA-1422	Too_many_rows	SELECT INTO 返回多行
ORA-1476	Zero-divide	试图被零除
ORA-1722	Invalid-NUMBER	转换一个数字失败
ORA-6500	Storage-error	内存不够引发的内部错误
ORA-6501	Program-error	内部错误
ORA-6502	Value-error	转换或截断错误
ORA-6504	Rowtype-mismatch	宿主游标变量与 PL/SQL 变量有不兼容行类型
ORA-6511	CURSOR-already-OPEN	试图打开一个已处于打开状态的游标
ORA-6530	Access-INTO-null	试图为 null 对象的属性赋值
ORA-6531	Collection-is-null	试图将 Exists 以外的集合(collection)方法应用于一个 null pl/sql 表上或 varray 上
ORA-6532	Subscript-outside-limit	对嵌套或 varray 索引得引用超出声明范围以外
ORA-6533	Subscript-beyond-count	对嵌套或 varray 索引得引用大于集合中元素的个数.

对这种异常情况的处理，只需在 PL/SQL 块的异常处理部分，直接引用相应的异常情况名，并对其完成相应的异常错误处理即可。

例 1：更新指定员工工资，如工资小于 1500，则加 100；

```
DECLARE
  v_empno employees.employee_id%TYPE := &empno;
  v_sal   employees.salary%TYPE;
BEGIN
  SELECT salary INTO v_sal FROM employees WHERE employee_id = v_empno;
  IF v_sal<=1500 THEN
     UPDATE employees SET salary = salary + 100 WHERE employee_id=v_empno;
     DBMS_OUTPUT.PUT_LINE('编码为'||v_empno||'员工工资已更新!');
  ELSE
     DBMS_OUTPUT.PUT_LINE('编码为'||v_empno||'员工工资已经超过规定值!');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('数据库中没有编码为'||v_empno||'的员工');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('程序运行错误!请使用游标');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

5.1.2 非预定义的异常处理

对于这类异常情况的处理，首先必须对非定义的 ORACLE 错误进行定义。步骤如下：

1. 在 PL/SQL 块的定义部分定义异常情况：

   <异常情况> EXCEPTION;

2. 将其定义好的异常情况，与标准的 ORACLE 错误联系起来，使用 EXCEPTION_INIT 语句：

   PRAGMA EXCEPTION_INIT(<异常情况>, <错误代码>);

3. 在 PL/SQL 块的异常情况处理部分对异常情况做出相应的处理。

例 2：删除指定部门的记录信息，以确保该部门没有员工。

```
INSERT INTO departments VALUES(50, 'FINANCE', 'CHICAGO');

DECLARE
  v_deptno departments.department_id%TYPE := &deptno;
```

```
  deptno_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT(deptno_remaining, -2292);
  /* -2292 是违反一致性约束的错误代码 */
BEGIN
  DELETE FROM departments WHERE department_id = v_deptno;
EXCEPTION
  WHEN deptno_remaining THEN
    DBMS_OUTPUT.PUT_LINE('违反数据完整性约束!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

5.1.3 用户自定义的异常处理

当与一个异常错误相关的错误出现时，就会隐含触发该异常错误。用户定义的异常错误是通过显式使用 RAISE 语句来触发。当引发一个异常错误时，控制就转向到 EXCEPTION 块异常错误部分，执行错误处理代码。

对于这类异常情况的处理，步骤如下：

1. 在 PL/SQL 块的定义部分定义异常情况：

   <异常情况> EXCEPTION;

2. RAISE <异常情况>；

3. 在 PL/SQL 块的异常情况处理部分对异常情况做出相应的处理。

例 3：更新指定员工工资，增加 100；

```
DECLARE
  v_empno employees.employee_id%TYPE :=&empno;
  no_result  EXCEPTION;
BEGIN
  UPDATE employees SET salary = salary+100 WHERE employee_id = v_empno;
  IF SQL%NOTFOUND THEN
    RAISE no_result;
  END IF;
EXCEPTION
  WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('你的数据更新语句失败了!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

5.1.4 用户定义的异常处理

调用 DBMS_STANDARD(ORACLE 提供的包)包所定义的 RAISE_APPLICATION_ERROR 过程，可以重新定义异常错误消息，它为应用程序提供了一种与 ORACLE 交互的方法。RAISE_APPLICATION_ERROR 的语法如下：

```
RAISE_APPLICATION_ERROR(error_number,error_message,[keep_errors] );
```

这里的 error_number 是从 –20,000 到 –20,999 之间的参数，
error_message 是相应的提示信息(< 2048 字节)，
keep_errors 为可选，如果 keep_errors =TRUE ,则新错误将被添加到已经引发的错误列表中。如果 keep_errors=FALSE(缺省),则新错误将替换当前的错误列表。

例 4：创建一个函数 get_salary，该函数检索指定部门的工资总和，其中定义了-20991 和 -20992 号错误，分别处理参数为空和非法部门代码两种错误：

```
CREATE TABLE errlog(
 Errcode NUMBER,
 Errtext CHAR(40));

CREATE OR REPLACE FUNCTION get_salary(p_deptno NUMBER)
RETURN NUMBER
AS
 v_sal NUMBER;
BEGIN
 IF p_deptno IS NULL THEN
  RAISE_APPLICATION_ERROR(-20991, '部门代码为空');
 ELSIF p_deptno<0 THEN
  RAISE_APPLICATION_ERROR(-20992, '无效的部门代码');
 ELSE
  SELECT SUM(employees.salary) INTO v_sal FROM employees
  WHERE employees.department_id=p_deptno;
  RETURN v_sal;
 END IF;
END;
```

```
DECLARE
 V_salary NUMBER(7,2);
 V_sqlcode NUMBER;
 V_sqlerr VARCHAR2(512);
 Null_deptno EXCEPTION;
 Invalid_deptno EXCEPTION;
 PRAGMA EXCEPTION_INIT(null_deptno,-20991);
 PRAGMA EXCEPTION_INIT(invalid_deptno, -20992);
```

```
BEGIN
 V_salary :=get_salary(10);
 DBMS_OUTPUT.PUT_LINE('10 号部门工资：' || TO_CHAR(V_salary));

 BEGIN
  V_salary :=get_salary(-10);
 EXCEPTION
  WHEN invalid_deptno THEN
   V_sqlcode :=SQLCODE;
   V_sqlerr  :=SQLERRM;
   INSERT INTO errlog(errcode, errtext)
   VALUES(v_sqlcode, v_sqlerr);
   COMMIT;
 END inner1;

 V_salary :=get_salary(20);
 DBMS_OUTPUT.PUT_LINE('部门号为 20 的工资为：'||TO_CHAR(V_salary));

 BEGIN
  V_salary :=get_salary(NULL);
 END inner2;

 V_salary := get_salary(30);
 DBMS_OUTPUT.PUT_LINE('部门号为 30 的工资为：'||TO_CHAR(V_salary));

 EXCEPTION
  WHEN null_deptno THEN
   V_sqlcode :=SQLCODE;
   V_sqlerr  :=SQLERRM;
   INSERT INTO errlog(errcode, errtext) VALUES(v_sqlcode, v_sqlerr);
   COMMIT;
  WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END outer;
```

例 5：定义触发器，使用 RAISE_APPLICATION_ERROR 阻止没有员工姓名的新员式记录插入：

```
CREATE OR REPLACE TRIGGER tr_insert_emp
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
 IF :new.first_name IS NULL OR :new.last_name is null THEN
  RAISE_APPLICATION_ERROR(-20000,'Employee must have a name.');
 END IF;
```

```
END;
```

5.2 异常错误传播

由于异常错误可以在声明部分和执行部分以及异常错误部分出现，因而在不同部分引发的异常错误也不一样。

5.2.1 在执行部分引发异常错误

当一个异常错误在执行部分引发时，有下列情况：

- 如果当前块对该异常错误设置了处理，则执行它并成功完成该块的执行，然后控制转给包含块。
- 如果没有对当前块异常错误设置定义处理器，则通过在包含块中引发它来传播异常错误。然后对该包含块执行步骤 1)。

5.2.2 在声明部分引发异常错误

如果在声明部分引起异常情况，即在声明部分出现错误，那么该错误就能影响到其它的块。比如在有如下的 PL/SQL 程序：

```
DECLARE
   name varchar2(12):='EricHu';
   其它语句
BEGIN
   其它语句
EXCEPTION
   WHEN OTHERS THEN
   其它语句
END;
```

例子中，由于 Abc number(3)='abc'; 出错，尽管在 EXCEPTION 中说明了 WHEN OTHERS THEN 语句，但 WHEN OTHERS THEN 也不会被执行。 但是如果在该错误语句块的外部有一个异常错误，则该错误能被抓住，如：

```
BEGIN
   DECLARE
   name varchar2(12):='EricHu';
   其它语句
  BEGIN
   其它语句
  EXCEPTION
   WHEN OTHERS THEN
   其它语句
   END;
EXCEPTION
```

```
WHEN OTHERS THEN
  其它语句
END;
```

5.3 异常错误处理编程

在一般的应用处理中，建议程序人员要用异常处理，因为如果程序中不声明任何异常处理，则在程序运行出错时，程序就被终止，并且也不提示任何信息。下面是使用系统提供的异常来编程的例子。

5.4 在 PL/SQL 中使用 SQLCODE, SQLERRM 异常处理函数

由于 ORACLE 的错信息最大长度是 512 字节，为了得到完整的错误提示信息，我们可用 SQLERRM 和 SUBSTR 函数一起得到错误提示信息，方便进行错误，特别是如果 WHEN OTHERS 异常处理器时更为方便。

SQLCODE 返回遇到的 Oracle 错误号,

SQLERRM 返回遇到的 Oracle 错误信息.

如: SQLCODE=-100 ➔ SQLERRM='no_data_found '

SQLCODE=0 ➔ SQLERRM='normal, successful completion'

例 6. 将 ORACLE 错误代码及其信息存入错误代码表

```
CREATE TABLE errors (errnum NUMBER(4), errmsg VARCHAR2(100));

DECLARE
  err_msg  VARCHAR2(100);
BEGIN
  /* 得到所有 ORACLE 错误信息 */
  FOR err_num IN -100 .. 0 LOOP
    err_msg := SQLERRM(err_num);
    INSERT INTO errors VALUES(err_num, err_msg);
  END LOOP;
END;
DROP TABLE errors;
```

例 7. 查询 ORACLE 错误代码;

```
BEGIN
  INSERT INTO employees(employee_id, first_name,last_name,hire_date,department_id)
  VALUES(2222, 'Eric','Hu', SYSDATE, 20);
  DBMS_OUTPUT.PUT_LINE('插入数据记录成功!');

  INSERT INTO employees(employee_id, first_name,last_name,hire_date,department_id)
  VALUES(2222, '胡','勇', SYSDATE, 20);
```

```
  DBMS_OUTPUT.PUT_LINE('插入数据记录成功!');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

例 8. 利用 ORACLE 错误代码，编写异常错误处理代码；

```
DECLARE
  empno_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT(empno_remaining, -1);
  /* -1 是违反唯一约束条件的错误代码 */
BEGIN
  INSERT INTO employees(employee_id, first_name,last_name,hire_date,department_id)
  VALUES(3333, 'Eric','Hu', SYSDATE, 20);
  DBMS_OUTPUT.PUT_LINE('插入数据记录成功!');

  INSERT INTO employees(employee_id, first_name,last_name,hire_date,department_id)
  VALUES(3333, '胡','勇',SYSDATE, 20);
  DBMS_OUTPUT.PUT_LINE('插入数据记录成功!');
EXCEPTION
  WHEN empno_remaining THEN
    DBMS_OUTPUT.PUT_LINE('违反数据完整性约束!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

6.1 存储过程函数引言

过程与函数（另外还有包与触发器）是命名的 PL/SQL 块（也是用户的方案对象），被编译后存储在数据库中，以备执行。因此，其它 PL/SQL 块可以按名称来使用他们。所以，可以将商业逻辑、企业规则写成函数或过程保存到数据库中，以便共享。

过程和函数统称为 PL/SQL 子程序，他们是被命名的 PL/SQL 块，均存储在数据库中，并通过输入、输出参数或输入/输出参数与其调用者交换信息。过程和函数的唯一区别是函数总向调用者返回数据，而过程则不返回数据。在本节中，主要介绍：

1. 创建存储过程和函数。
2. 正确使用系统级的异常处理和用户定义的异常处理。
3. 建立和管理存储过程和函数。

6.2 创建函数

1. 创建函数

语法如下：

```
CREATE [OR REPLACE] FUNCTION function_name
 (arg1 [ { IN | OUT | IN OUT }] type1 [DEFAULT value1],
 [arg2 [ { IN | OUT | IN OUT }] type2 [DEFAULT value1]],
 ……
 [argn [ { IN | OUT | IN OUT }] typen [DEFAULT valuen]])
 [ AUTHID DEFINER | CURRENT_USER ]
RETURN return_type
 IS | AS
   <类型.变量的声明部分>
BEGIN
   执行部分
   RETURN expression
EXCEPTION
   异常处理部分
END function_name;
```

- IN,OUT,IN OUT 是形参的模式。若省略，则为 IN 模式。IN 模式的形参只能将实参传递给形参，进入函数内部，但只能读不能写，函数返回时实参的值不变。OUT 模式的形参会忽略调用时的实参值（或说该形参的初始值总是 NULL），但在函数内部可以被读或写，函数返回时形参的值会赋予给实参。IN OUT 具有前两种模式的特性，即调用时，实参的值总是传递给形参，结束时，形参的值传递给实参。调用时，对于 IN 模式的实参可以是常量或变量，但对于 OUT 和 IN OUT 模式的实参必须是变量。

- 一般，只有在确认 `function_name` 函数是新函数或是要更新的函数时，才使用 OR REPALCE 关键字，否则容易删除有用的函数。

例 1. 获取某部门的工资总和：

```
--获取某部门的工资总和
CREATE OR REPLACE
FUNCTION get_salary(
 Dept_no NUMBER,
 Emp_count OUT NUMBER)
 RETURN NUMBER
IS
 V_sum NUMBER;
BEGIN
 SELECT SUM(SALARY), count(*) INTO V_sum, emp_count
   FROM EMPLOYEES WHERE DEPARTMENT_ID=dept_no;
 RETURN v_sum;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('你需要的数据不存在!');
```

```
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END get_salary;
```

2. 函数的调用

函数声明时所定义的参数称为形式参数，应用程序调用时为函数传递的参数称为实际参数。应用程序在调用函数时，可以使用以下三种方法向函数传递参数：

第一种参数传递格式：位置表示法。

即在调用时按形参的排列顺序，依次写出实参的名称，而将形参与实参关联起来进行传递。用这种方法进行调用，形参与实参的名称是相互独立，没有关系，强调次序才是重要的。

格式为：

```
  argument_value1[,argument_value2 …]
```

例 2： 计算某部门的工资总和：

```
DECLARE
 V_num NUMBER;
 V_sum NUMBER;
BEGIN
 V_sum :=get_salary(10, v_num);
 DBMS_OUTPUT.PUT_LINE('部门号为:10 的工资总和：'||v_sum||'，人数为：'||v_num);
END;
```

第二种参数传递格式：名称表示法。

即在调用时按形参的名称与实参的名称，写出实参对应的形参，而将形参与实参关联起来进行传递。这种方法，形参与实参的名称是相互独立的，没有关系，名称的对应关系才是最重要的，次序并不重要。

格式为：

```
argument => parameter [,…]
```

其中：argument 为形式参数，它必须与函数定义时所声明的形式参数名称相同
parameter 为实际参数。

在这种格式中，形势参数与实际参数成对出现，相互间关系唯一确定，所以参数的顺序可以任意排列。

例 3：计算某部门的工资总和：

```
DECLARE
 V_num NUMBER;
  V_sum NUMBER;
BEGIN
  V_sum :=get_salary(emp_count => v_num, dept_no => 10);
  DBMS_OUTPUT.PUT_LINE('部门号为:10 的工资总和：'||v_sum||'，人数为：'||v_num);
END;
```

第三种参数传递格式：组合传递。

即在调用一个函数时，同时使用位置表示法和名称表示法为函数传递参数。采用这种参数传递方法时，使用位置表示法所传递的参数必须放在名称表示法所传递的参数前面。也就是说，无论函数具有多少个参数，只要其中有一个参数使用名称表示法，其后所有的参数都必须使用名称表示法。

例 4：

```
CREATE OR REPLACE FUNCTION demo_fun(
 Name VARCHAR2,--注意 VARCHAR2 不能给精度，如：VARCHAR2(10)，其它类似
 Age INTEGER,
 Sex VARCHAR2)
 RETURN VARCHAR2
AS
 V_var VARCHAR2(32);
BEGIN
 V_var := name||'：'||TO_CHAR(age)||'岁.'||sex;
 RETURN v_var;
END;
```

```
DECLARE
 Var VARCHAR(32);
BEGIN
 Var := demo_fun('user1', 30, sex => '男');
 DBMS_OUTPUT.PUT_LINE(var);

 Var := demo_fun('user2', age => 40, sex => '男');
```

```
 DBMS_OUTPUT.PUT_LINE(var);

 Var := demo_fun('user3', sex => '女', age => 20);
 DBMS_OUTPUT.PUT_LINE(var);
END;
```

无论采用哪一种参数传递方法，实际参数和形式参数之间的数据传递只有两种方法：传址法和传值法。所谓传址法是指在调用函数时，将实际参数的地址指针传递给形式参数，使形式参数和实际参数指向内存中的同一区域，从而实现参数数据的传递。这种方法又称作参照法，即形式参数参照实际参数数据。输入参数均采用传址法传递数据。

传值法是指将实际参数的数据拷贝到形式参数，而不是传递实际参数的地址。默认时，输出参数和输入/输出参数均采用传值法。在函数调用时，ORACLE 将实际参数数据拷贝到输入/输出参数，而当函数正常运行退出时，又将输出形式参数和输入/输出形式参数数据拷贝到实际参数变量中。

3. 参数默认值

在 CREATE OR REPLACE FUNCTION 语句中声明函数参数时可以使用 DEFAULT 关键字为输入参数指定默认值。

例 5：

```
CREATE OR REPLACE FUNCTION demo_fun(
 Name VARCHAR2,
 Age INTEGER,
 Sex VARCHAR2 DEFAULT '男')
 RETURN VARCHAR2
AS
 V_var VARCHAR2(32);
BEGIN
 V_var := name||'： '||TO_CHAR(age)||'岁.'||sex;
 RETURN v_var;
END;
```

具有默认值的函数创建后，在函数调用时，如果没有为具有默认值的参数提供实际参数值，函数将使用该参数的默认值。但当调用者为默认参数提供实际参数时，函数将使用实际参数值。在创建函数时，只能为输入参数设置默认值，而不能为输入/输出参数设置默认值。

```
DECLARE
 var VARCHAR(32);
BEGIN
```

```
 Var := demo_fun('user1', 30);
 DBMS_OUTPUT.PUT_LINE(var);
 Var := demo_fun('user2', age => 40);
 DBMS_OUTPUT.PUT_LINE(var);
 Var := demo_fun('user3', sex => '女', age => 20);
 DBMS_OUTPUT.PUT_LINE(var);
END;
```

6.3 存储过程

6.3.1 创建过程

建立存储过程

在 ORACLE SERVER 上建立存储过程,可以被多个应用程序调用,可以向存储过程传递参数,也可以向存储过程传回参数.

创建过程语法:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
([arg1 [ IN | OUT | IN OUT ]] type1 [DEFAULT value1],
 [arg2 [ IN | OUT | IN OUT ]] type2 [DEFAULT value1]],
 ......
 [argn [ IN | OUT | IN OUT ]] typen [DEFAULT valuen])
   [ AUTHID DEFINER | CURRENT_USER ]
{ IS | AS }
 <声明部分>
BEGIN
 <执行部分>
EXCEPTION
 <可选的异常错误处理程序>
END procedure_name;
```

说明：相关参数说明参见函数的语法说明。

例 6. 用户连接登记记录；

```
CREATE TABLE logtable (userid VARCHAR2(10), logdate date);

CREATE OR REPLACE PROCEDURE logexecution
IS
BEGIN
INSERT INTO logtable (userid, logdate) VALUES (USER, SYSDATE);
END;
```

例 7. 删除指定员工记录；

```
CREATE OR REPLACE
PROCEDURE DelEmp
(v_empno IN employees.employee_id%TYPE)
AS
No_result EXCEPTION;
BEGIN
  DELETE FROM employees WHERE employee_id = v_empno;
  IF SQL%NOTFOUND THEN
    RAISE no_result;
  END IF;
  DBMS_OUTPUT.PUT_LINE('编码为'||v_empno||'的员工已被删除!');
EXCEPTION
  WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END DelEmp;
```

例 8. 插入员工记录:

```
CREATE OR REPLACE
PROCEDURE InsertEmp(
  v_empno     in employees.employee_id%TYPE,
  v_firstname in employees.first_name%TYPE,
  v_lastname  in employees.last_name%TYPE,
  v_deptno    in employees.department_id%TYPE
  )
AS
  empno_remaining EXCEPTION;
  PRAGMA EXCEPTION_INIT(empno_remaining, -1);
  /* -1 是违反唯一约束条件的错误代码 */
BEGIN
   INSERT INTO EMPLOYEES(EMPLOYEE_ID, FIRST_NAME, LAST_NAME, HIRE_DATE,DEPARTMENT_
ID)
  VALUES(v_empno, v_firstname,v_lastname, sysdate, v_deptno);
  DBMS_OUTPUT.PUT_LINE('温馨提示:插入数据记录成功!');
EXCEPTION
  WHEN empno_remaining THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:违反数据完整性约束!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END InsertEmp;
```

例 9. 使用存储过程向 departments 表中插入数据。

```
CREATE OR REPLACE
PROCEDURE insert_dept
 (v_dept_id IN departments.department_id%TYPE,
  v_dept_name IN departments.department_name%TYPE,
  v_mgr_id IN departments.manager_id%TYPE,
  v_loc_id IN departments.location_id%TYPE)
IS
  ept_null_error EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_null_error, -1400);
  ept_no_loc_id EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_no_loc_id, -2291);
BEGIN
  INSERT INTO departments
  (department_id, department_name, manager_id, location_id)
  VALUES
  (v_dept_id, v_dept_name, v_mgr_id, v_loc_id);
  DBMS_OUTPUT.PUT_LINE('插入部门'||v_dept_id||'成功');
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    RAISE_APPLICATION_ERROR(-20000, '部门编码不能重复');
  WHEN ept_null_error THEN
    RAISE_APPLICATION_ERROR(-20001, '部门编码、部门名称不能为空');
  WHEN ept_no_loc_id THEN
    RAISE_APPLICATION_ERROR(-20002, '没有该地点');
END insert_dept;
```

```
/*调用实例一:
DECLARE
  ept_20000 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20000, -20000);
  ept_20001 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20001, -20001);
  ept_20002 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20002, -20002);
BEGIN
  insert_dept(300, '部门 300', 100, 2400);
  insert_dept(310, NULL, 100, 2400);
  insert_dept(310, '部门 310', 100, 900);
EXCEPTION
  WHEN ept_20000 THEN
    DBMS_OUTPUT.PUT_LINE('ept_20000 部门编码不能重复');
  WHEN ept_20001 THEN
```

```
    DBMS_OUTPUT.PUT_LINE('ept_20001 部门编码、部门名称不能为空');
  WHEN ept_20002 THEN
    DBMS_OUTPUT.PUT_LINE('ept_20002 没有该地点');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('others 出现了其他异常错误');
END;

调用实例二：
DECLARE
  ept_20000 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20000, -20000);
  ept_20001 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20001, -20001);
  ept_20002 EXCEPTION;
  PRAGMA EXCEPTION_INIT(ept_20002, -20002);
BEGIN
  insert_dept(v_dept_name => '部门 310', v_dept_id => 310,
          v_mgr_id => 100, v_loc_id => 2400);
  insert_dept(320, '部门 320', v_mgr_id => 100, v_loc_id => 900);
EXCEPTION
  WHEN ept_20000 THEN
    DBMS_OUTPUT.PUT_LINE('ept_20000 部门编码不能重复');
  WHEN ept_20001 THEN
    DBMS_OUTPUT.PUT_LINE('ept_20001 部门编码、部门名称不能为空');
  WHEN ept_20002 THEN
    DBMS_OUTPUT.PUT_LINE('ept_20002 没有该地点');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('others 出现了其他异常错误');
END;
*/
```

6.3.2 调用存储过程

存储过程建立完成后，只要通过授权，用户就可以在 SQLPLUS 、ORACLE 开发工具或第三方开发工具中来调用运行。对于参数的传递也有三种：按位置传递、按名称传递和组合传递，传递方法与函数的一样。ORACLE 使用 EXECUTE 语句来实现对存储过程的调用：

```
EXEC[UTE] procedure_name( parameter1, parameter2…);
```

例 10：

```
EXECUTE logexecution;
```

例 11：查询指定员工记录；

```
CREATE OR REPLACE
PROCEDURE QueryEmp
(v_empno IN  employees.employee_id%TYPE,
 v_ename OUT employees.first_name%TYPE,
 v_sal   OUT employees.salary%TYPE)
AS
BEGIN
     SELECT last_name || last_name, salary INTO v_ename, v_sal
   FROM employees
   WHERE employee_id = v_empno;
     DBMS_OUTPUT.PUT_LINE('温馨提示:编码为'||v_empno||'的员工已经查到!');
EXCEPTION
     WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
    WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END QueryEmp;
```

```
--调用
 DECLARE
   v1 employees.first_name%TYPE;
   v2 employees.salary%TYPE;
 BEGIN
  QueryEmp(100, v1, v2);
  DBMS_OUTPUT.PUT_LINE('姓名:'||v1);
  DBMS_OUTPUT.PUT_LINE('工资:'||v2);
  QueryEmp(103, v1, v2);
  DBMS_OUTPUT.PUT_LINE('姓名:'||v1);
  DBMS_OUTPUT.PUT_LINE('工资:'||v2);
  QueryEmp(104, v1, v2);
  DBMS_OUTPUT.PUT_LINE('姓名:'||v1);
  DBMS_OUTPUT.PUT_LINE('工资:'||v2);
END;
```

例 12. 计算指定部门的工资总和，并统计其中的职工数量。

```
CREATE OR REPLACE
PROCEDURE proc_demo
(
 dept_no NUMBER DEFAULT 10,
  sal_sum OUT NUMBER,
  emp_count OUT NUMBER
 )
IS
BEGIN
   SELECT SUM(salary), COUNT(*) INTO sal_sum, emp_count
 FROM employees WHERE department_id = dept_no;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END proc_demo;
```

```
DECLARE
V_num NUMBER;
V_sum NUMBER(8, 2);
BEGIN
 Proc_demo(30, v_sum, v_num);
DBMS_OUTPUT.PUT_LINE('温馨提示:30 号部门工资总和：'||v_sum||',人数：'||v_num);
 Proc_demo(sal_sum => v_sum, emp_count => v_num);
DBMS_OUTPUT.PUT_LINE('温馨提示:10 号部门工资总和：'||v_sum||',人数：'||v_num);
END;
```

在 PL/SQL 程序中还可以在块内建立本地函数和过程，这些函数和过程不存储在数据库中，但可以在创建它们的 PL/SQL 程序中被重复调用。本地函数和过程在 PL/SQL 块的声明部分定义，它们的语法格式与存储函数和过程相同，但不能使用 CREATE OR REPLACE 关键字。

例 13： 建立本地过程，用于计算指定部门的工资总和，并统计其中的职工数量；

```
DECLARE
V_num NUMBER;
V_sum NUMBER(8, 2);
PROCEDURE proc_demo
 (
```

```
   Dept_no NUMBER DEFAULT 10,
   Sal_sum OUT NUMBER,
   Emp_count OUT NUMBER
 )
IS
BEGIN
   SELECT SUM(salary), COUNT(*) INTO sal_sum, emp_count
   FROM employees WHERE department_id=dept_no;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('你需要的数据不存在!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END proc_demo;
```

```
--调用方法：
BEGIN
   Proc_demo(30, v_sum, v_num);
DBMS_OUTPUT.PUT_LINE('30 号部门工资总和：'||v_sum||'，人数：'||v_num);
   Proc_demo(sal_sum => v_sum, emp_count => v_num);
DBMS_OUTPUT.PUT_LINE('10 号部门工资总和：'||v_sum||'，人数：'||v_num);
END;
```

6.3.3 AUTHID

过程中的 AUTHID 指令可以告诉 ORACLE ，这个过程使用谁的权限运行．默任情况下，存储过程会作为调用者的过程运行，但是具有设计者的特权．这称为设计者权利运行．

例 14：建立过程，使用 AUTOID DEFINER；

```
Connect HR/qaz
DROP TABLE logtable;
CREATE table logtable (userid VARCHAR2(10), logdate date);

CREATE OR REPLACE PROCEDURE logexecution
   AUTHID DEFINER
IS
BEGIN
  INSERT INTO logtable (userid, logdate) VALUES (USER, SYSDATE);
END;

GRANT EXECUTE ON logexecution TO PUBLIC;
```

```
CONNECT / AS SYSDBA
GRANT CONNECT TO testuser1 IDENTIFIED BY userpwd1;

CONNECT testuser1/userpwd1
INSERT INTO HR.LOGTABLE VALUES (USER, SYSDATE);
EXECUTE HR.logexecution

CONNECT HR/qaz
SELECT * FROM HR.logtable;
```

例 15：建立过程，使用 AUTOID CURRENT_USER；

```
CONNECT HR/qaz

CREATE OR REPLACE PROCEDURE logexecution
 AUTHID CURRENT_USER
IS
BEGIN
  INSERT INTO logtable (userid, logdate) VALUES (USER, SYSDATE);
END;

GRANT EXECUTE ON logexecution TO PUBLIC;

CONNECT testuser1/userpwd1
INSERT INTO HR.LOGTABLE VALUES (USER, SYSDATE);
EXECUTE HR.logexecution
```

6.3.4 PRAGMA AUTONOMOUS_TRANSACTION

ORACLE8i 可以支持事务处理中的事务处理的概念．这种子事务处理可以完成它自己的工作，独立于父事务处理进行提交或者回滚．通过使用这种方法，开发者就能够这样的过程，无论父事务处理是提交还是回滚，它都可以成功执行．

例 16：建立过程，使用自动事务处理进行日志记录；

```
DROP TABLE logtable;

CREATE TABLE logtable(
 Username varchar2(20),
 Dassate_time date,
 Mege varchar2(60)
);

CREATE TABLE temp_table( N number );
```

```
CREATE OR REPLACE PROCEDURE log_message(p_message varchar2)
 AS
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 INSERT INTO logtable VALUES ( user, sysdate, p_message );
 COMMIT;
END log_message;

BEGIN
 Log_message (‘About to insert into temp_table‘);
 INSERT INTO temp_table VALUES (1);
 Log_message (‘Rollback to insert into temp_table‘);
 ROLLBACK;
END;

SELECT * FROM logtable;
SELECT * FROM temp_table;
```

例 17：建立过程，没有使用自动事务处理进行日志记录；

```
CREATE OR REPLACE PROCEDURE log_message(p_message varchar2)
 AS
BEGIN
 INSERT INTO logtable VALUES ( user, sysdate, p_message );
 COMMIT;
END log_message;

BEGIN
 Log_message ('About to insert into temp_table');
 INSERT INTO temp_table VALUES (1);
 Log_message ('Rollback to insert into temp_table');
 ROLLBACK;
END;

SELECT * FROM logtable;
SELECT * FROM temp_table;
```

6.3.5 开发存储过程步骤

开发存储过程、函数、包及触发器的步骤如下：

6.3.5.1 使用文字编辑处理软件编辑存储过程源码

使用文字编辑处理软件编辑存储过程源码，要用类似 WORD 文字处理软件进行编辑时，要将源码存为文本格式。

6.3.5.2 在 SQLPLUS 或用调试工具将存储过程程序进行解释

在 SQLPLUS 或用调试工具将存储过程程序进行解释；

在 SQL>下调试，可用 START 或 GET 等 ORACLE 命令来启动解释。如：

```
SQL>START c:\stat1.sql
```

如果使用调式工具，可直接编辑和点击相应的按钮即可生成存储过程。

6.3.5.3 调试源码直到正确

我们不能保证所写的存储过程达到一次就正确。所以这里的调式是每个程序员必须进行的工作之一。在 SQLPLUS 下来调式主要用的方法是：

- 使用 SHOW ERROR 命令来提示源码的错误位置；
- 使用 user_errors 数据字典来查看各存储过程的错误位置。

6.3.5.4 授权执行权给相关的用户或角色

如果调式正确的存储过程没有进行授权，那就只有建立者本人才可以运行。所以作为应用系统的一部分的存储过程也必须进行授权才能达到要求。在 SQL*PLUS 下可以用 GRANT 命令来进行存储过程的运行授权。

GRANT 语法：

```
GRANT system_privilege | role
TO user | role | PUBLIC [WITH ADMIN OPTION]

GRANT object_privilege | ALL ON schema.object
TO user | role | PUBLIC [WITH GRANT OPTION]
```

--例子：

```
CREATE OR REPLACE PUBLIC SYNONYM dbms_job FOR dbms_job

GRANT EXECUTE ON dbms_job TO PUBLIC WITH GRANT OPTION
```

6.3.5.5 与过程相关数据字典

USER_SOURCE, ALL_SOURCE, DBA_SOURCE, USER_ERRORS,

ALL_PROCEDURES,USER_OBJECTS,ALL_OBJECTS,DBA_OBJECTS

相关的权限:

```
CREATE ANY PROCEDURE

DROP ANY PROCEDURE
```

在 SQL*PLUS 中，可以用 DESCRIBE 命令查看过程的名字及其参数表。

```
DESC[RIBE] Procedure_name;
```

6.3.6 删除过程和函数

1．删除过程

可以使用 DROP PROCEDURE 命令对不需要的过程进行删除，语法如下：

```
DROP PROCEDURE [user.]Procudure_name;
```

2．删除函数

可以使用 DROP FUNCTION 命令对不需要的函数进行删除，语法如下：

```
DROP FUNCTION [user.]Function_name;
```

--删除上面实例创建的存储过程与函数

```
DROP PROCEDURE logexecution;
DROP PROCEDURE delemp;
DROP PROCEDURE insertemp;
DROP PROCEDURE fireemp;
DROP PROCEDURE queryemp;
DROP PROCEDURE proc_demo;
DROP PROCEDURE log_message;
DROP FUNCTION demo_fun;
DROP FUNCTION get_salary;
```

6.3.7 过程与函数的比较

使用过程与函数具有如下优点：

1、共同使用的代码可以只需要被编写和测试一次，而被需要该代码的任何应用程序（如：.NET、C++、JAVA、VB 程序，也可以是 DLL 库）调用。

2、这种集中编写、集中维护更新、大家共享（或重用）的方法，简化了应用程序的开发和维护，提高了效率与性能。

3、这种模块化的方法，使得可以将一个复杂的问题、大的程序逐步简化成几个简单的、小的程序部分，进行分别编写、调试。因此使程序的结构清晰、简单，也容易实现。

4、可以在各个开发者之间提供处理数据、控制流程、提示信息等方面的一致性。

5、节省内存空间。它们以一种压缩的形式被存储在外存中，当被调用时才被放入内存进行处理。并且，如果多个用户要执行相同的过程或函数时，就只需要在内存中加载一个该过程或函数。

6、提高数据的安全性与完整性。通过把一些对数据的操作放到过程或函数中，就可以通过是否授予用户有执行该过程或的权限，来限制某些用户对数据进行这些操作。

过程与函数的相同功能有：

1、 都使用 IN 模式的参数传入数据、OUT 模式的参数返回数据。
2、 输入参数都可以接受默认值，都可以传值或传引导。
3、 调用时的实际参数都可以使用位置表示法、名称表示法或组合方法。
4、 都有声明部分、执行部分和异常处理部分。
5、 其管理过程都有创建、编译、授权、删除、显示依赖关系等。

使用过程与函数的原则：

1、如果需要返回多个值和不返回值，就使用过程；如果只需要返回一个值，就使用函数。

2、过程一般用于执行一个指定的动作，函数一般用于计算和返回一个值。

3、可以 SQL 语句内部（如表达式）调用函数来完成复杂的计算问题，但不能调用过程。所以这是函数的特色。

7.1 程序包简介

程序包(PACKAGE，简称包)是一组相关过程、函数、变量、常量和游标等 PL/SQL 程序设计元素的组合，作为一个完整的单元存储在数据库中，用名称来标识包。它具有面向对象程序设计语言的特点，是对这些 PL/SQL 程序设计元素的封装。包类似于 c#和 JAVA 语言中的类，其中变量相当于类中的成员变量，过程和函数相当于类方法。把相关的模块归类成为包，可使开发人员利用面向对象的方法进行存储过程的开发，从而提高系统性能。

与高级语言中的类相同，包中的程序元素也分为公用元素和私用元素两种，这两种元素的区别是他们允许访问的程序范围不同，即它们的作用域不同。公用元素不仅可以被包中的函数、过程所调用，也可以被包外的 PL/SQL 程序访问，而私有元素只能被包内的函数和过程序所访问。

当然，对于不包含在程序包中的过程、函数是独立存在的。一般是先编写独立的过程与函数，待其较为完善或经过充分验证无误后，再按逻辑相关性组织为程序包。

程序包的优点

◆ **简化应用程序设计：**程序包的说明部分和包体部分可以分别创建各编译。主要体现 在以下三个方面：

1) 可以在设计一个应用程序时，只创建各编译程序包的说明部分，然后再编写引用该 程序包的 PL/SQL 块。

2) 当完成整个应用程序的整体框架后，再回头来定义包体部分。只要不改变包的说明部分，就可以单独调试、增加或替换包体的内容，这不会影响其他的应用程序。

3) 更新包的说明后必须重新编译引用包的应用程序，但更新包体，则不需重新编译引用包的应用程序，以快速进行进行应用程序的原形开发。

◆ **模块化：**可将逻辑相关的 PL/SQL 块或元素等组织在一起，用名称来唯一标识程序 包。把一个大的功能模块划分人适当个数小的功能模块，分别完成各自的功能。这样组织的程序包都易于编写，易于理解更易于管理。

◆ **信息隐藏：**因为包中的元素可以分为公有元素和私有元素。公有元素可被程序包内的过程、函数等的访问，还可以被包外的 PL/SQL 访问。但对于私有元素只能被包内的过程、函数等访问。对于用户，只需知道包的说明，不用了解包休的具体细节。

◆ **效率高：**程序包在应用程序第一次调用程序包中的某个元素时，ORACLE 将把整个程序包加载到内存中，当第二次访问程序包中的元素时，ORACLE 将直接从内在中读取，而不需要进行磁盘 I/O 操作而影响速度，同时位于内在中的程序包可被同一会话期间的其它应用程序共享。因此，程序包增加了重用性并改善了多用户、多应用程序环境的效率。

对程序包的优点可总结如下：在 PL/SQL 程序设计中，使用包不仅可以使程序设计模块化，对外隐藏包内所使用的信息（通过使用私用变量），而写可以提高程序的执行效率。因为，当程序首次调用包内函数或过程时，ORACLE 将整个包调入内存，当再次访问包内元素时，ORACLE 直接从内存中读取，而不需要进行磁盘 I/O 操作，从而使程序执行效率得到提高。

一个包由两个分开的部分组成：

包说明（PACKAGE）：包说明部分声明包内数据类型、变量、常量、游标、子程序和

异常错误处理等元素，这些元素为包的公有元素。

包主体（PACKAGE BODY）：包主体则是包定义部分的具体实现，它定义了包定义部分所声明的游标和子程序，在包主体中还可以声明包的私有元素。

包说明和包主体分开编译，并作为两部分分开的对象存放在数据库字典中，可查看数据字典 user_source, all_source, dba_source，分别了解包说明与包主体的详细信息。

7.2 程序包的定义

程序包的定义分为程序包说明定义和程序包主体定义两部分组成。

程 序包说明用于声明包的公用组件，如变量、常量、自定义数据类型、异常、过程、函数、游标等。包说明中定义的公有组件不仅可以在包内使用，还可以由包外其他 过程、函数。但需要说明与注意的是，我们为了实现信息的隐藏，建议不要将所有组件都放在包说明处声明，只应把公共组件放在包声明部分。包的名称是唯一的， 但对于两个包中的公有组件的名称可以相同，这种用“包名.公有组件名“加以区分。

包体是包的具体实现细节，其实现在包说明中声明的所有公有过程、函数、游标等。当然也可以在包体中声明仅属于自己的私有过程、函数、游标等。创建包体时，有以下几点需要注意：

- 包体只能在包说明被创建或编译后才能进行创建或编译。
- 在包体中实现的过程、函数、游标的名称必须与包说明中的过程、函数、游标一致，包括名称、参数的名称以及参数的模式（IN、OUT、IN OUT）。并建设按包说明中的次序定义包体中具体的实现。
- 在包体中声明的数据类型、变量、常量都是私有的，只能在包体中使用而不能被印刷体外的应用程序访问与使用。
- 在包体执行部分，可对包说明，包体中声明的公有或私有变量进行初始化或其它设置。

创建程序包说明语法格式:

```
CREATE [OR REPLACE] PACKAGE package_name
 [AUTHID {CURRENT_USER | DEFINER}]
 {IS | AS}
 [公有数据类型定义[公有数据类型定义]…]
 [公有游标声明[公有游标声明]…]
 [公有变量、常量声明[公有变量、常量声明]…]
 [公有函数声明[公有函数声明]…]
 [公有过程声明[公有过程声明]…]
END [package_name];
```

其中：AUTHID CURRENT_USER 和 AUTHID DEFINER 选项说明应用程序在调用函数时所使用的权限模式，它们与 CREATE FUNCTION 语句中 invoker_right_clause 子句的作用相同。

创建程序包主体语法格式:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
 {IS | AS}
 [私有数据类型定义[私有数据类型定义]…]
```

```
 [私有变量、常量声明[私有变量、常量声明]…]
 [私有异常错误声明[私有异常错误声明]…]
 [私有函数声明和定义[私有函数声明和定义]…]
 [私有函过程声明和定义[私有函过程声明和定义]…]
 [公有游标定义[公有游标定义]…]
 [公有函数定义[公有函数定义]…]
 [公有过程定义[公有过程定义]…]
BEGIN
 执行部分(初始化部分)
END package_name;
```

其中：在包主体定义公有程序时，它们必须与包定义中所声明子程序的格式完全一致。

7.3 包的开发步骤

与开发存储过程类似，包的开发需要几个步骤：

1. 将每个存储过程调式正确；
2. 用文本编辑软件将各个存储过程和函数集成在一起；
3. 按照包的定义要求将集成的文本的前面加上包定义；
4. 按照包的定义要求将集成的文本的前面加上包主体；
5. 使用 SQLPLUS 或开发工具进行调式。

7.4 包定义的说明

例 1:创建的包为 DEMO_PKG, 该包中包含一个记录变量 DEPTREC、两个函数和一个过程。实现对 dept 表的增加、删除与查询。

```
CREATE OR REPLACE PACKAGE  DEMO_PKG
IS
 DEPTREC DEPT%ROWTYPE;

 --Add dept...
 FUNCTION add_dept(
       dept_no    NUMBER,
       dept_name VARCHAR2,
       location  VARCHAR2)
 RETURN NUMBER;

 --delete dept...
 FUNCTION delete_dept(dept_no NUMBER)
 RETURN NUMBER;

 --query dept...
 PROCEDURE query_dept(dept_no IN NUMBER);
END DEMO_PKG;
```

包主体的创建方法，它实现上面所声明的包定义，并在包主体中声明一个私有变量 flag 和一个私有函数 check_dept，由于在 add_dept 和 remove_dept 等函数中需要调用 check_dpet 函数，所以，在定义 check_dept 函数之前首先对该函数进行声明，这种声明方法称作前向声明。

```
CREATE OR REPLACE PACKAGE BODY DEMO_PKG
IS
FUNCTION add_dept
(
  dept_no NUMBER,
  dept_name VARCHAR2,
  location VARCHAR2
)
RETURN NUMBER
IS
 empno_remaining EXCEPTION; --自定义异常
 PRAGMA EXCEPTION_INIT(empno_remaining, -1);
 /* -1 是违反唯一约束条件的错误代码 */
BEGIN
 INSERT INTO dept VALUES(dept_no, dept_name, location);
 IF SQL%FOUND THEN
   RETURN 1;
 END IF;
EXCEPTION
   WHEN empno_remaining THEN
     RETURN 0;
   WHEN OTHERS THEN
     RETURN -1;
END add_dept;

FUNCTION delete_dept(dept_no NUMBER)
RETURN NUMBER
IS
BEGIN
 DELETE FROM dept WHERE deptno = dept_no;
 IF SQL%FOUND THEN
  RETURN 1;
 ELSE
  RETURN 0;
 END IF;
EXCEPTION
 WHEN OTHERS THEN
  RETURN -1;
END delete_dept;
```

```
PROCEDURE query_dept
(dept_no IN NUMBER)
IS
BEGIN
    SELECT * INTO DeptRec FROM dept WHERE deptno=dept_no;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('温馨提示:数据库中没有编码为'||dept_no||'的部门');
    WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('程序运行错误,请使用游标进行操作!');
    WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);
END query_dept;

BEGIN
  Null;
END DEMO_PKG;
```

对包内共有元素的调用格式为：包名.元素名称

调用 DEMO_PKG 包内函数对 dept 表进行插入、查询和删除操作，并通过 DEMO_PKG 包中的记录变量 DEPTREC 显示所查询到的数据库信息：

```
DECLARE
  Var NUMBER;
BEGIN
  Var := DEMO_PKG.add_dept(90,'HKLORB', 'HAIKOU');
  IF var =-1 THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);
  ELSIF var =0 THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:该部门记录已经存在！');
  ELSE
    DBMS_OUTPUT.PUT_LINE('温馨提示:添加记录成功！');
    DEMO_PKG.query_dept(90);
    DBMS_OUTPUT.PUT_LINE(DEMO_PKG.DeptRec.deptno||'---'||
     DEMO_PKG.DeptRec.dname||'---'||DEMO_PKG.DeptRec.loc);
    var := DEMO_PKG.delete_dept(90);
    IF var =-1 THEN
      DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);
    ELSIF var=0 THEN
      DBMS_OUTPUT.PUT_LINE('温馨提示:该部门记录不存在！');
    ELSE
      DBMS_OUTPUT.PUT_LINE('温馨提示:删除记录成功！');
    END IF;
```

```
  END IF;
END;
```

例 2: 创建包 EMP_PKG，读取 emp 表中的数据

```
--创建包说明
CREATE OR REPLACE PACKAGE EMP_PKG
IS
 TYPE emp_table_type IS TABLE OF emp%ROWTYPE
 INDEX BY BINARY_INTEGER;

 PROCEDURE read_emp_table (p_emp_table OUT emp_table_type);
END EMP_PKG;

--创建包体
CREATE OR REPLACE PACKAGE BODY EMP_PKG
IS
PROCEDURE read_emp_table (p_emp_table OUT emp_table_type)
IS
I BINARY_INTEGER := 0;
BEGIN
  FOR emp_record IN ( SELECT * FROM emp ) LOOP
    P_emp_table(i) := emp_record;
    I := I + 1;
  END LOOP;
 END read_emp_table;
END EMP_PKG;
```

```
--执行
DECLARE
 E_table EMP_PKG.emp_table_type;
BEGIN
 EMP_PKG.read_emp_table(e_table);
 FOR I IN e_table.FIRST ..e_table.LAST LOOP
  DBMS_OUTPUT.PUT_LINE(e_table(i).empno||' '||e_table(i).ename);
 END LOOP;
END;
```

例 3: 创建包 MANAGE_EMP_PKG，对员工进行管理（新增员工、新增部门、删除指定员工、删除指定部门、增加指定员工的工资与奖金）：

```
--创建序列从 100 开始,依次增加 1
CREATE SEQUENCE empseq
```

```
START WITH 100
INCREMENT BY 1
ORDER NOCYCLE;

--创建序列从 100 开始,依次增加 10
CREATE SEQUENCE deptseq
START WITH 100
INCREMENT BY 10
ORDER NOCYCLE;

-- ****************************************
 -- 创建包说明
 -- 包  名：MANAGE_EMP_PKG
 -- 功能描述：对员工进行管理(新增员工,新增部门
 --        ,删除员工,删除部门,增加工资与奖金等)
 -- 创建人员：kenny
 -- 创建日期：2010-05-19
-- ****************************************
CREATE OR REPLACE PACKAGE MANAGE_EMP_PKG
AS
 --增加一名员工
 FUNCTION hire_emp
  (ename VARCHAR2, job VARCHAR2
  , mgr NUMBER, sal NUMBER
  , comm NUMBER, deptno NUMBER)
 RETURN NUMBER;

 --新增一个部门
 FUNCTION add_dept(dname VARCHAR2, loc VARCHAR2)
 RETURN NUMBER;

 --删除指定员工
 PROCEDURE remove_emp(empno NUMBER);
 --删除指定部门
 PROCEDURE remove_dept(deptno NUMBER);
 --增加指定员工的工资
 PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER);
 --增加指定员工的奖金
 PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER);
END MANAGE_EMP_PKG;--创建包说明结束

-- ****************************************
 -- 创建包体
 -- 包  名：MANAGE_EMP_PKG
```

```
 -- 功能描述：对员工进行管理(新增员工,新增部门
 --         ,删除员工,删除部门,增加工资与奖金等)
 -- 创建人员：kenny
-- ****************************************
CREATE OR REPLACE PACKAGE BODY MANAGE_EMP_PKG
AS
   total_emps  NUMBER; --员工数
   total_depts NUMBER; --部门数
   no_sal    EXCEPTION;
   no_comm   EXCEPTION;
 --增加一名员工
 FUNCTION hire_emp(ename VARCHAR2, job VARCHAR2, mgr NUMBER,
                   sal NUMBER, comm NUMBER, deptno NUMBER)
 RETURN NUMBER  --返回新增加的员工编号
 IS
   new_empno NUMBER(4);
 BEGIN
SELECT empseq.NEXTVAL INTO new_empno FROM dual;
SELECT COUNT(*) INTO total_emps FROM emp;--当前记录总数

   INSERT INTO emp
   VALUES (new_empno, ename, job, mgr, sysdate, sal, comm, deptno);
   total_emps:=total_emps+1;
 RETURN(new_empno);
 EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
 END hire_emp;

 --新增一个部门
 FUNCTION add_dept(dname VARCHAR2, loc VARCHAR2)
 RETURN NUMBER
 IS
   new_deptno NUMBER(4); --部门编号
 BEGIN
   --得到一个新的自增的员工编号
   SELECT deptseq.NEXTVAL INTO new_deptno FROM dual;
   SELECT COUNT(*) INTO total_depts FROM dept;--当前部门总数
   INSERT INTO dept VALUES (new_deptno, dname, loc);
   total_depts:=total_depts;
 RETURN(new_deptno);
 EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
```

```
END add_dept;

--删除指定员工
PROCEDURE remove_emp(empno NUMBER)
IS
 no_result EXCEPTION; --自定义异常
BEGIN
 DELETE FROM emp WHERE emp.empno=remove_emp.empno;
 IF SQL%NOTFOUND THEN
    RAISE no_result;
 END IF;
 total_emps:=total_emps - 1; --总的员工数减 1
EXCEPTION
  WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
END remove_emp;

--删除指定部门
PROCEDURE remove_dept(deptno NUMBER)
IS
  no_result EXCEPTION; --自定义异常
  exception_deptno_remaining EXCEPTION; --自定义异常
  /*-2292 是违反一致性约束的错误代码*/
  PRAGMA EXCEPTION_INIT(exception_deptno_remaining, -2292);
BEGIN
 DELETE FROM dept WHERE dept.deptno=remove_dept.deptno;

 IF SQL%NOTFOUND THEN
    RAISE no_result;
 END IF;
 total_depts:=total_depts-1; --总的部门数减 1
EXCEPTION
  WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
  WHEN exception_deptno_remaining THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:违反数据完整性约束!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
END remove_dept;

--给指定员工增加指定数量的工资
PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER)
```

```
IS
 curr_sal NUMBER(7, 2); --当前工资
BEGIN
 --得到当前工资
 SELECT sal INTO curr_sal FROM emp WHERE emp.empno=increase_sal.empno;

 IF curr_sal IS NULL THEN
   RAISE no_sal;
 ELSE
   UPDATE emp SET sal = sal + increase_sal.sal_incr --当前工资加新增的工资
   WHERE emp.empno = increase_sal.empno;
 END IF;
 EXCEPTION
   WHEN NO_DATA_FOUND THEN
     DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
   WHEN no_sal THEN
     DBMS_OUTPUT.PUT_LINE('温馨提示:此员工的工资不存在!');
   WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
END increase_sal;

--给指定员工增加指定数量的奖金
PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER)
IS
 curr_comm NUMBER(7,2);
BEGIN
 --得到指定员工的当前资金
 SELECT comm INTO curr_comm
 FROM emp WHERE emp.empno = increase_comm.empno;

 IF curr_comm IS NULL THEN
   RAISE no_comm;
 ELSE
  UPDATE emp SET comm = comm + increase_comm.comm_incr
  WHERE emp.empno=increase_comm.empno;
 END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:你需要的数据不存在!');
  WHEN no_comm THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:此员工的奖金不存在!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('温馨提示:发生系统错误!');
END increase_comm;
```

```
END MANAGE_EMP_PKG;--创建包体结束

--调用
SQL> variable empno number
SQL>execute  :empno:= manage_emp_pkg.hire_emp('HUYONG',PM,1455,5500,14,10)

PL/SQL procedure successfully completed
empno
---------
105
```

例 4：利用游标变量创建包 CURROR_VARIBAL_PKG。由于游标变量指是一个指针，其状态是不确定的，因此它不能随同包存储在数据库中，既不能在 PL/SQL 包中声明游标变量。但在包中可以创建游标变量参照类型，并可向包中的子程序传递游标变量参数。

```
-- ******************************************
 -- 创建包体
 -- 包  名：CURROR_VARIBAL_PKG
 -- 功能描述：在包中引用游标变量
 -- 创建人员：kenny
 - 创建日期：2010-05-09
 - ******************************************
CREATE OR REPLACE PACKAGE CURROR_VARIBAL_PKG AS
 TYPE DeptCurType IS REF CURSOR
 RETURN dept%ROWTYPE; --强类型定义

 TYPE CurType IS REF CURSOR;-- 弱类型定义

 PROCEDURE OpenDeptVar(
   Cv IN OUT DeptCurType,
   Choice INTEGER DEFAULT 0,
   Dept_no NUMBER DEFAULT 50,
   Dept_name VARCHAR DEFAULT '%');
END;

-- ******************************************
 -- 创建包体
 -- 包  名：CURROR_VARIBAL_PKG
 -- 功能描述：在包中引用游标变量
 -- 创建人员：胡勇
 -- 创建日期：2010-05-19
 -- ******************************************
CREATE OR REPLACE PACKAGE BODY CURROR_VARIBAL_PKG
```

```
AS
  PROCEDURE OpenDeptvar(
    Cv IN OUT DeptCurType,
    Choice INTEGER DEFAULT 0,
    Dept_no NUMBER DEFAULT 50,
    Dept_name VARCHAR DEFAULT '%')
  IS
  BEGIN
    IF choice =1 THEN
      OPEN cv FOR SELECT * FROM dept WHERE deptno <= dept_no;
    ELSIF choice = 2 THEN
      OPEN cv FOR SELECT * FROM dept WHERE dname LIKE dept_name;
    ELSE
      OPEN cv FOR SELECT * FROM dept;
    END IF;
  END OpenDeptvar;
END CURROR_VARIBAL_PKG;

--定义一个过程
CREATE OR REPLACE PROCEDURE UP_OpenCurType(
  Cv IN OUT CURROR_VARIBAL_PKG.CurType,
  FirstCapInTableName CHAR)
AS
BEGIN
  --CURROR_VARIBAL_PKG.CurType 采用弱类型定义
  --所以可以使用它定义的游标变量打开不同类型的查询语句
  IF FirstCapInTableName = 'D' THEN
    OPEN cv FOR SELECT * FROM dept;
  ELSE
    OPEN cv FOR SELECT * FROM emp;
  END IF;
END UP_OpenCurType;


--定义一个应用
DECLARE
  DeptRec Dept%ROWTYPE;
  EmpRec Emp%ROWTYPE;
  Cv1 CURROR_VARIBAL_PKG.deptcurtype;
  Cv2 CURROR_VARIBAL_PKG.curtype;
BEGIN
  DBMS_OUTPUT.PUT_LINE('游标变量强类型定义应用');
  CURROR_VARIBAL_PKG.OpenDeptVar(cv1, 1, 30);
  FETCH cv1 INTO DeptRec;
```

```
 WHILE cv1%FOUND LOOP
  DBMS_OUTPUT.PUT_LINE(DeptRec.deptno||':'||DeptRec.dname);
  FETCH cv1 INTO DeptRec;
 END LOOP;
 CLOSE cv1;

 DBMS_OUTPUT.PUT_LINE('游标变量弱类型定义应用');
 CURROR_VARIBAL_PKG.OpenDeptvar(cv2, 2, dept_name => 'A%');
 FETCH cv2 INTO DeptRec;
 WHILE cv2%FOUND LOOP
  DBMS_OUTPUT.PUT_LINE(DeptRec.deptno||':'||DeptRec.dname);
  FETCH cv2 INTO DeptRec;
 END LOOP;

 DBMS_OUTPUT.PUT_LINE('游标变量弱类型定义应用—dept 表');
 UP_OpenCurType(cv2, 'D');
 FETCH cv2 INTO DeptRec;
 WHILE cv2%FOUND LOOP
  DBMS_OUTPUT.PUT_LINE(deptrec.deptno||':'||deptrec.dname);
  FETCH cv2 INTO deptrec;
 END LOOP;

 DBMS_OUTPUT.PUT_LINE('游标变量弱类型定义应用—emp 表');
 UP_OpenCurType(cv2, 'E');
 FETCH cv2 INTO EmpRec;
 WHILE cv2%FOUND LOOP
  DBMS_OUTPUT.PUT_LINE(emprec.empno||':'||emprec.ename);
  FETCH cv2 INTO emprec;
 END LOOP;
 CLOSE cv2;
END;
----------运行结果----------
游标变量强类型定义应用
10:ACCOUNTING
20:RESEARCH
30:SALES
游标变量弱类型定义应用
10:ACCOUNTING
游标变量弱类型定义应用—dept 表
10:ACCOUNTING
20:RESEARCH
30:SALES
40:OPERATIONS
50:50abc
```

```
60:Developer
游标变量弱类型定义应用—emp 表
7369:SMITH
7499:ALLEN
7521:WARD
7566:JONES
7654:MARTIN
7698:BLAKE
7782:CLARK
7788:SCOTT
7839:KING
7844:TURNER
7876:ADAMS
7900:JAMES
7902:FORD
7934:MILLER

PL/SQL procedure successfully completed
```

7.5 子程序重载

PL/SQL 允许对包内子程序和本地子程序进行重载。所谓重载时指两个或多个子程序有相同的名称，但拥有不同的参数变量、参数顺序或参数数据类型。

例 5：

```
-- ******************************************
 -- 创建包说明
 -- 包  名：DEMO_PKG1
 -- 功能描述：创建包对子程序重载进行测试
 -- 创建人员：kenny
 -- 创建日期：2010-05-22
-- ******************************************
CREATE OR REPLACE PACKAGE DEMO_PKG1
IS
   DeptRec dept%ROWTYPE;
   V_sqlcode NUMBER;
   V_sqlerr VARCHAR2(2048);

 --两个子程序名字相同,但参数类型不同
   FUNCTION query_dept(dept_no IN NUMBER)
   RETURN INTEGER;

   FUNCTION query_dept(dept_no IN VARCHAR2)
```

```
  RETURN INTEGER;
END DEMO_PKG1;

-- ********************************************
 -- 创建包体
 -- 包  名：DEMO_PKG1
 -- 功能描述：创建包对子程序重载进行测试
 -- 创建人员：kenny
 -- 创建日期：2010-05-22
-- ********************************************
CREATE OR REPLACE PACKAGE BODY DEMO_PKG1
IS
 FUNCTION check_dept(dept_no NUMBER)
 RETURN INTEGER
 IS
  deptCnt INTEGER; --指定部门号的部门数量
 BEGIN
  SELECT COUNT(*) INTO deptCnt FROM dept WHERE deptno = dept_no;
  IF deptCnt > 0 THEN
   RETURN 1;
  ELSE
   RETURN 0;
  END IF;
 END check_dept;

 FUNCTION check_dept(dept_no VARCHAR2)
 RETURN INTEGER
 IS
  deptCnt INTEGER;
 BEGIN
  SELECT COUNT(*) INTO deptCnt FROM dept WHERE deptno=dept_no;
  IF deptCnt > 0 THEN
   RETURN 1;
  ELSE
   RETURN 0;
  END IF;
 END check_dept;

 FUNCTION query_dept(dept_no IN NUMBER)
 RETURN INTEGER
 IS
 BEGIN
  IF check_dept(dept_no) =1 THEN
   SELECT * INTO DeptRec FROM dept WHERE deptno=dept_no;
```

```
    RETURN 1;
   ELSE
    RETURN 0;
   END IF;
  END query_dept;

  FUNCTION query_dept(dept_no IN VARCHAR2)
   RETURN INTEGER
  IS
  BEGIN
   IF check_dept(dept_no) =1 THEN
    SELECT * INTO DeptRec FROM dept WHERE deptno = dept_no;
    RETURN 1;
   ELSE
    RETURN 0;
   END IF;
  END query_dept;

END DEMO_PKG1;
```

7.6 加密实用程序

ORACLE 提供了一个实用工具来加密或者包装用户的 PL/SQL，它会将用户的 PL/SQL 改变为只有 ORACLE 能够解释的代码版本.

WRAP 实用工具位于$ORACLE_HOME/BIN.

格式为：

```
WRAP INAME=<input_file_name> [ONAME=<output_file_name>]
```

```
wrap iname=e:\sample.txt
```

注意：在加密前，请将 PL/SQL 程序先保存一份，以备后用。

7.7 删除包

可以使用 DROP PACKAGE 命令对不需要的包进行删除，语法如下：

```
DROP PACKAGE [BODY] [user.]package_name;
```

```
DROP PROCEDURE OpenCurType; --删除存储过程
--删除我们实例中创建的各个包
DROP PACKAGE demo_pack;
```

```
DROP PACKAGE demo_pack1;
DROP PACKAGE emp_mgmt;
DROP PACKAGE emp_package;
```

7.8 包的管理

包与过程、函数一样，也是存储在数据库中的，可以随时查看其源码。若有需要，在创建包时可以随时查看更详细的编译错误。不需要的包也可以删除。

同样，为了避免调用的失败，在更新表的结构后，一定要记得重新编译依赖于它的程序包。在更新了包说明或包体后，也应该重新编译包说明与包体。语法如下：

```
ALTER PACKAGE package_name COMPILE [PACKAGE|BODY|SPECIFICATION];
```

也可以通过以下数据字典视图查看包的相关。

DBA_SOURCE, USER_SOURCE, USER_ERRORS, DBA-OBJECTS

如，我们可以用：select text from user_source where name = 'DEMO_PKG1';来查看我们创建的包的源码。

触发器是许多关系数据库系统都提供的一项技术。在 ORACLE 系统里，触发器类似过程和函数，都有声明，执行和异常处理过程的 PL/SQL 块。

8.1 触发器类型

触发器在数据库里以独立的对象存储，它与存储过程和函数不同的是，存储过程与函数需要用户显示调用才执行，而触发器是由一个事件来启动运行。即触发器是当某个事件发生时自动地隐式运行。并且，触发器不能接收参数。所以运行触发器就叫触发或点火（firing）。ORACLE 事件指的是对数据库的表进行的 INSERT、UPDATE 及 DELETE 操作或对视图进行类似的操作。ORACLE 将触发器的功能扩展到了触发 ORACLE，如数据库的启动与关闭等。所以触发器常用来完成由数据库的完整性约束难以完成的复杂业务规则的约束，或用来监视对数据库的各种操作，实现审计的功能。

8.1.1 DML 触发器

ORACLE 可以在 DML 语句进行触发，可以在 DML 操作前或操作后进行触发，并且可以对每个行或语句操作上进行触发。

8.1.2 替代触发器

由于在 ORACLE 里，不能直接对由两个以上的表建立的视图进行操作。所以给出了替代触发器。它就是 ORACLE 8 专门为进行视图操作的一种处理方法。

8.1.3 系统触发器

ORACLE 8i 提供了第三种类型的触发器叫系统触发器。它可以在 ORACLE 数据库系统的事件中进行触发，如 ORACLE 系统的启动与关闭等。

触发器组成:

- 触发事件：引起触发器被触发的事件。例如：DML 语句(INSERT, UPDATE, DELETE 语句对表或视图执行数据处理操作)、DDL 语句（如 CREATE、ALTER、DROP 语句在数据库中创建、修改、删除模式对象）、数据库系统事件（如系统启动或退出、异常错误）、用户事件（如登录或退出数据库）。
- 触发时间：即该 TRIGGER 是在触发事件发生之前（BEFORE）还是之后(AFTER)触发，也就是触发事件和该 TRIGGER 的操作顺序。
- 触发操作：即该 TRIGGER 被触发之后的目的和意图，正是触发器本身要做的事情。 例如：PL/SQL 块。
- 触发对象：包括表、视图、模式、数据库。只有在这些对象上发生了符合触发条件的触发事件，才会执行触发操作。
- 触发条件：由 WHEN 子句指定一个逻辑表达式。只有当该表达式的值为 TRUE 时，遇到触发事件才会自动执行触发器，使其执行触发操作。
- 触发频率：说明触发器内定义的动作被执行的次数。即语句级(STATEMENT)触发器和行级(ROW)触发器。
  语句级(STATEMENT)触发器：是指当某触发事件发生时，该触发器只执行一次；
  行级(ROW)触发器：是指当某触发事件发生时，对受到该操作影响的每一行数据，触发器都单独执行一次。

编写触发器时，需要注意以下几点：

- 触发器不接受参数。
- 一个表上最多可有 12 个触发器，但同一时间、同一事件、同一类型的触发器只能有一个。并各触发器之间不能有矛盾。
- 在一个表上的触发器越多，对在该表上的 DML 操作的性能影响就越大。
- 触发器最大为 32KB。若确实需要，可以先建立过程，然后在触发器中用 CALL 语句进行调用。
- **在触发器的执行部分只能用 DML 语句（SELECT、INSERT、UPDATE、DELETE），不能使用 DDL 语句（CREATE、ALTER、DROP）。**
- 触发器中不能包含事务控制语句(COMMIT，ROLLBACK，SAVEPOINT)。因为触发器是触发语句的一部分，触发语句被提交、回退时，触发器也被提交、回退了。
- 在触发器主体中调用的任何过程、函数，都不能使用事务控制语句。
- 在触发器主体中不能申明任何 Long 和 blob 变量。新值 new 和旧值 old 也不能向表中的任何 long 和 blob 列。

- 不同类型的触发器(如 DML 触发器、INSTEAD OF 触发器、系统触发器)的语法格式和作用有较大区别。

8.2 创建触发器

创建触发器的一般语法是:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER }
{INSERT | DELETE | UPDATE [OF column [, column …]]}
[OR {INSERT | DELETE | UPDATE [OF column [, column …]]}...]
ON [schema.]table_name | [schema.]view_name
[REFERENCING {OLD [AS] old | NEW [AS] new| PARENT as parent}]
[FOR EACH ROW ]
[WHEN condition]
PL/SQL_BLOCK | CALL procedure_name;
```

其中：

BEFORE 和 AFTER 指出触发器的触发时序分别为前触发和后触发方式，前触发是在执行触发事件之前触发当前所创建的触发器，后触发是在执行触发事件之后触发当前所创建的触发器。

FOR EACH ROW 选项说明触发器为行触发器。行触发器和语句触发器的区别表现在：行触发器要求当一个 DML 语句操走影响数据库中的多行数据时，对于其中的每个数据行，只要它们符合触发约束条件，均激活一次触发器；而语句触发器将整个语句操作作为触发事件，当它符合约束条件时，激活一次触发器。当省略 FOR EACH ROW 选项时，BEFORE 和 AFTER 触发器为语句触发器，而 INSTEAD OF 触发器则只能为行触发器。

REFERENCING 子句说明相关名称，在行触发器的 PL/SQL 块和 WHEN 子句中可以使用相关名称参照当前的新、旧列值，默认的相关名称分别为 OLD 和 NEW。触发器的 PL/SQL 块中应用相关名称时，必须在它们之前加冒号(:)，但在 WHEN 子句中则不能加冒号。

WHEN 子句说明触发约束条件。Condition 为一个逻辑表达时，其中必须包含相关名称，而不能包含查询语句，也不能调用 PL/SQL 函数。WHEN 子句指定的触发约束条件只能用在 BEFORE 和 AFTER 行触发器中，不能用在 INSTEAD OF 行触发器和其它类型的触发器中。

当一个基表被修改(INSERT, UPDATE, DELETE)时要执行的存储过程，执行时根据其所依附的基表改动而自动触发，因此与应用程序无关，用数据库触发器可以保证数据的一致性和完整性。

每张表最多可建立 12 种类型的触发器，它们是:

- BEFORE INSERT
- BEFORE INSERT FOR EACH ROW
- AFTER INSERT
- AFTER INSERT FOR EACH ROW

- BEFORE UPDATE
- BEFORE UPDATE FOR EACH ROW
- AFTER UPDATE
- AFTER UPDATE FOR EACH ROW

- BEFORE DELETE
- BEFORE DELETE FOR EACH ROW
- AFTER DELETE
- AFTER DELETE FOR EACH ROW

8.2.1 触发器触发次序

1. 执行 BEFORE 语句级触发器;
2. 对与受语句影响的每一行：
   - 执行 BEFORE 行级触发器
   - 执行 DML 语句
   - 执行 AFTER 行级触发器
3. 执行 AFTER 语句级触发器

8.2.2 创建 DML 触发器

触发器名与过程名和包的名字不一样，它是单独的名字空间，因而触发器名可以和表或过程有相同的名字，但在一个模式中触发器名不能相同。

DML 触发器的限制

- CREATE TRIGGER 语句文本的字符长度不能超过 32KB；
- 触发器体内的 SELECT 语句只能为 SELECT … INTO …结构，或者为定义游标所使用的 SELECT 语句。
- 触发器中不能使用数据库事务控制语句 COMMIT; ROLLBACK, SVAEPOINT 语句；

- 由触发器所调用的过程或函数也不能使用数据库事务控制语句；
- 触发器中不能使用 LONG, LONG RAW 类型；
- 触发器内可以参照 LOB 类型列的列值，但不能通过 :NEW 修改 LOB 列中的数据；

DML 触发器基本要点

- **触发时机：**指定触发器的触发时间。如果指定为 BEFORE，则表示在执行 DML 操作之前触发，以便防止某些错误操作发生或实现某些业务规则；如果指定为 AFTER，则表示在执行 DML 操作之后触发，以便记录该操作或做某些事后处理。
- **触发事件：**引起触发器被触发的事件，即 DML 操作（INSERT、UPDATE、DELETE）。既可以是单个触发事件，也可以是多个触发事件的组合（只能使用 OR 逻辑组合，不能使用 AND 逻辑组合）。
- **条件谓词：**当在触发器中包含多个触发事件（INSERT、UPDATE、DELETE）的组合时，为了分别针对不同的事件进行不同的处理，需要使用 ORACLE 提供的如下条件谓词。
  1）。**INSERTING：**当触发事件是 INSERT 时，取值为 TRUE，否则为 FALSE。
  2）。**UPDATING [（column_1,column_2,…,column_x）]：**当触发事件是 UPDATE 时，如果修改了 column_x 列，则取值为 TRUE，否则为 FALSE。其中 column_x 是可选的。
  3）。**DELETING：**当触发事件是 DELETE 时，则取值为 TRUE，否则为 FALSE。
  解发对象：指定触发器是创建在哪个表、视图上。
- **触发类型：**是语句级还是行级触发器。
- **触发条件：**由 WHEN 子句指定一个逻辑表达式，只允许在行级触发器上指定触发条件，指定 UPDATING 后面的列的列表。

问题：当触发器被触发时，要使用被插入、更新或删除的记录中的列值，有时要使用操作前、后列的值.

实现: :NEW 修饰符访问操作完成后列的值

:OLD 修饰符访问操作完成前列的值

特性	*INSERT*	*UPDATE*	*DELETE*
OLD	NULL	实际值	实际值
NEW	实际值	实际值	NULL

例 1: 建立一个触发器，当职工表 emp 表被删除一条记录时，把被删除记录写到职工表删除日志表中去。

```
CREATE TABLE emp_his AS SELECT * FROM EMP WHERE 1=2;
CREATE OR REPLACE TRIGGER tr_del_emp
  BEFORE DELETE --指定触发时机为删除操作前触发
```

```
  ON scott.emp
  FOR EACH ROW  --说明创建的是行级触发器
BEGIN
  --将修改前数据插入到日志记录表 del_emp ,以供监督使用。
  INSERT INTO emp_his(deptno , empno, ename , job ,mgr , sal , comm , hiredate )
     VALUES( :old.deptno, :old.empno, :old.ename , :old.job,:old.mgr, :old.sal, :old.comm, :old.hir
edate );
END;
DELETE emp WHERE empno=7788;
DROP TABLE emp_his;
DROP TRIGGER del_emp;
```

例 2：限制对 Departments 表修改（包括 INSERT,DELETE,UPDATE）的时间范围，即不允许在非工作时间修改 departments 表。

```
CREATE OR REPLACE TRIGGER tr_dept_time
BEFORE INSERT OR DELETE OR UPDATE
ON departments
BEGIN
     IF     (TO_CHAR(sysdate,'DAY')     IN     ('  星  期  六  ',  '  星  期  日
')) OR (TO_CHAR(sysdate, 'HH24:MI') NOT BETWEEN '08:30' AND '18:00') THEN
   RAISE_APPLICATION_ERROR(-20001, '不是上班时间，不能修改 departments 表');
 END IF;
END;
```

例 3：限定只对部门号为 80 的记录进行行触发器操作。

```
CREATE OR REPLACE TRIGGER tr_emp_sal_comm
BEFORE UPDATE OF salary, commission_pct
    OR DELETE
ON HR.employees
FOR EACH ROW
WHEN (old.department_id = 80)
BEGIN
 CASE
   WHEN UPDATING ('salary') THEN
     IF :NEW.salary < :old.salary THEN

       RAISE_APPLICATION_ERROR(-20001, '部门 80 的人员的工资不能降');
     END IF;
   WHEN UPDATING ('commission_pct') THEN

     IF :NEW.commission_pct < :old.commission_pct THEN
       RAISE_APPLICATION_ERROR(-20002, '部门 80 的人员的奖金不能降');
```

```
    END IF;
  WHEN DELETING THEN
     RAISE_APPLICATION_ERROR(-20003, '不能删除部门 80 的人员记录');
  END CASE;
END;

/*
实例：
UPDATE employees SET salary = 8000 WHERE employee_id = 177;
DELETE FROM employees WHERE employee_id in (177,170);
*/
```

例 4：利用行触发器实现级联更新。在修改了主表 regions 中的 region_id 之后（AFTER），级联的、自动的更新子表 countries 表中原来在该地区的国家的 region_id。

```
CREATE OR REPLACE TRIGGER tr_reg_cou
AFTER update OF region_id
ON regions
FOR EACH ROW
BEGIN
 DBMS_OUTPUT.PUT_LINE('旧的 region_id 值是'||:old.region_id
          ||'、新的 region_id 值是'||:new.region_id);
 UPDATE countries SET region_id = :new.region_id
 WHERE region_id = :old.region_id;
END;
```

例 5：在触发器中调用过程。

```
CREATE OR REPLACE PROCEDURE add_job_history
 ( p_emp_id        job_history.employee_id%type
  , p_start_date     job_history.start_date%type
 , p_end_date       job_history.end_date%type
  , p_job_id        job_history.job_id%type
  , p_department_id   job_history.department_id%type
  )
IS
BEGIN
 INSERT INTO job_history (employee_id, start_date, end_date,
                job_id, department_id)
 VALUES(p_emp_id, p_start_date, p_end_date, p_job_id, p_department_id);
END add_job_history;

--创建触发器调用存储过程...
```

```
CREATE OR REPLACE TRIGGER update_job_history
 AFTER UPDATE OF job_id, department_id ON employees
 FOR EACH ROW
BEGIN
 add_job_history(:old.employee_id, :old.hire_date, sysdate,
            :old.job_id, :old.department_id);
END;
```

8.2.3 创建替代(INSTEAD OF)触发器

创建触发器的一般语法是:

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF
{INSERT | DELETE | UPDATE [OF column [, column …]]}
[OR {INSERT | DELETE | UPDATE [OF column [, column …]]}…]
ON [schema.] view_name --只能定义在视图上
[REFERENCING {OLD [AS] old | NEW [AS] new| PARENT as parent}]
[FOR EACH ROW ] --因为 INSTEAD OF 触发器只能在行级上触发,所以没有必要指定
[WHEN condition]
PL/SQL_block | CALL procedure_name;
```

其中：

INSTEAD OF 选项使 ORACLE 激活触发器，而不执行触发事件。只能对视图和对象视图建立 INSTEAD OF 触发器，而不能对表、模式和数据库建立 INSTEAD OF 触发器。

FOR EACH ROW 选项说明触发器为行触发器。行触发器和语句触发器的区别表现在：行触发器要求当一个 DML 语句操走影响数据库中的多行数据时，对于其中的每个数据行，只要它们符合触发约束条件，均激活一次触发器；而语句触发器将整个语句操作作为触发事件，当它符合约束条件时，激活一次触发器。当省略 FOR EACH ROW 选项时，BEFORE 和 AFTER 触发器为语句触发器，而 INSTEAD OF 触发器则为行触发器。

REFERENCING 子句说明相关名称，在行触发器的 PL/SQL 块和 WHEN 子句中可以使用相关名称参照当前的新、旧列值，默认的相关名称分别为 OLD 和 NEW。触发器的 PL/SQL 块中应用相关名称时，必须在它们之前加冒号(:)，但在 WHEN 子句中则不能加冒号。

WHEN 子句说明触发约束条件。Condition 为一个逻辑表达时，其中必须包含相关名称，而不能包含查询语句，也不能调用 PL/SQL 函数。WHEN 子句指定的触发约束条件只能用在 BEFORE 和 AFTER 行触发器中，不能用在 INSTEAD OF 行触发器和其它类型的触发器中。

INSTEAD_OF 用于对视图的 DML 触发，由于视图有可能是由多个表进行联结(join)而成，因而并非是所有的联结都是可更新的。但可以按照所需的方式执行更新，例如下面情况：

例 1：

```
CREATE OR REPLACE VIEW emp_view AS
SELECT deptno, count(*) total_employeer, sum(sal) total_salary
FROM emp GROUP BY deptno;
```

在此视图中直接删除是非法：

```
SQL>DELETE FROM emp_view WHERE deptno=10;
DELETE FROM emp_view WHERE deptno=10
```

```
ERROR 位于第 1 行:
ORA-01732: 此视图的数据操纵操作非法
```

但是我们可以创建 INSTEAD_OF 触发器来为 DELETE 操作执行所需的处理，即删除 EMP 表中所有基准行：

```
CREATE OR REPLACE TRIGGER emp_view_delete
  INSTEAD OF DELETE ON emp_view FOR EACH ROW
BEGIN
  DELETE FROM emp WHERE deptno= :old.deptno;
END emp_view_delete;

DELETE FROM emp_view WHERE deptno=10;

DROP TRIGGER emp_view_delete;

DROP VIEW emp_view;
```

例 2：创建复杂视图，针对 INSERT 操作创建 INSTEAD OF 触发器，向复杂视图插入数据。

- 创建视图：

```
CREATE OR REPLACE FORCE VIEW "HR"."V_REG_COU" ("R_ID", "R_NAME", "C_ID", "C_NAME")
AS
 SELECT r.region_id,
   r.region_name,
   c.country_id,
   c.country_name
 FROM regions r,
   countries c
 WHERE r.region_id = c.region_id;
```

- 创建触发器：

```
CREATE OR REPLACE TRIGGER "HR"."TR_I_O_REG_COU" INSTEAD OF
```

```
INSERT ON v_reg_cou FOR EACH ROW DECLARE v_count NUMBER;
BEGIN
SELECT COUNT(*) INTO v_count FROM regions WHERE region_id = :new.r_id;
IF v_count = 0 THEN
  INSERT INTO regions
   (region_id, region_name
   ) VALUES
   (:new.r_id, :new.r_name
   );
END IF;

SELECT COUNT(*) INTO v_count FROM countries WHERE country_id = :new.c_id;
IF v_count = 0 THEN
  INSERT
  INTO countries
   (
    country_id,
    country_name,
    region_id
   )
   VALUES
   (
    :new.c_id,
    :new.c_name,
    :new.r_id
   );
END IF;
END;
```

创建 INSTEAD OF 触发器需要注意以下几点：

- 只能被创建在视图上，并且该视图没有指定 WITH CHECK OPTION 选项。
- 不能指定 BEFORE 或 AFTER 选项。
- FOR EACH ROW 子可是可选的，即 INSTEAD OF 触发器只能在行级上触发、或只能是行级触发器，没有必要指定。
- 没有必要在针对一个表的视图上创建 INSTEAD OF 触发器，只要创建 DML 触发器就可以了。

8.2.3 创建系统事件触发器

ORACLE10G 提供的系统事件触发器可以在 DDL 或数据库系统上被触发。DDL 指的是数据定义语言，如 CREATE 、ALTER 及 DROP 等。而数据库系统事件包括数据库服务器的启动或关闭，用户的登录与退出、数据库服务错误等。创建系统触发器的语法如下：

创建触发器的一般语法是:

```
CREATE OR REPLACE TRIGGER [sachema.]trigger_name
{BEFORE|AFTER}
{ddl_event_list | database_event_list}
ON { DATABASE | [schema.]SCHEMA }
[WHEN condition]
PL/SQL_block | CALL procedure_name;
```

其中: ddl_event_list：一个或多个 DDL 事件，事件间用 OR 分开；
database_event_list：一个或多个数据库事件，事件间用 OR 分开；

系统事件触发器既可以建立在一个模式上，又可以建立在整个数据库上。当建立在模式(SCHEMA)之上时，只有模式所指定用户的 DDL 操作和它们所导致的错误才激活触发器，默认时为当前用户模式。当建立在数据库(DATABASE)之上时，该数据库所有用户的 DDL 操作和他们所导致的错误，以及数据库的启动和关闭均可激活触发器。要在数据库之上建立触发器时，要求用户具有 ADMINISTER DATABASE TRIGGER 权限。

下面给出系统触发器的种类和事件出现的时机（前或后）：

事件	*允许的时机*	*说明*
STARTUP	AFTER	启动数据库实例之后触发
SHUTDOWN	BEFORE	关闭数据库实例之前触发（非正常关闭不触发）
SERVERERROR	AFTER	数据库服务器发生错误之后触发
LOGON	AFTER	成功登录连接到数据库后触发
LOGOFF	BEFORE	开始断开数据库连接之前触发
CREATE	BEFORE，AFTER	在执行 CREATE 语句创建数据库对象之前、之后触发
DROP	BEFORE，AFTER	在执行 DROP 语句删除数据库对象之前、之后触发
ALTER	BEFORE，AFTER	在执行 ALTER 语句更新数据库对象之前、之后触发
DDL	BEFORE，AFTER	在执行大多数 DDL 语句之前、之后触发
GRANT	BEFORE，AFTER	执行 GRANT 语句授予权限之前、之后触发
REVOKE	BEFORE，AFTER	执行 REVOKE 语句收权限之前、之后触犯发
RENAME	BEFORE，AFTER	执行 RENAME 语句更改数据库对象名称之前、之后触犯发
AUDIT /	BEFORE，	执行 AUDIT 或 NOAUDIT 进行审计或

NOAUDIT	AFTER	停止审计之前、之后触发

8.2.4 系统触发器事件属性

事件属性\事件	Startup/Shutdown	Servererror	Logon/Logoff	DDL	DML
事件名称	*	*	*	*	*
数据库名称	*				
数据库实例号	*				
错误号		*			
用户名			*	*	
模式对象类型				*	*
模式对象名称				*	*
列					*

除 DML 语句的列属性外，其余事件属性值可通过调用 ORACLE 定义的事件属性函数来读取。

函数名称	数据类型	说　　明
Ora_sysevent	VARCHAR2（20）	激活触发器的事件名称
Instance_num	NUMBER	数据库实例名
Ora_database_name	VARCHAR2（50）	数据库名称
Server_error(posi)	NUMBER	错误信息栈中 posi 指定位置中的错误号
Is_servererror(err_number)	BOOLEAN	检查 err_number 指定的错误号是否在错误信息栈中，如果在则返回 TRUE，否则返回 FALSE。在触发器内调用此函数可以判断是否发生指定的错误。
Login_user	VARCHAR2(30)	登陆或注销的用户名称
Dictionary_obj_type	VARCHAR2(20)	DDL 语句所操作的数据库对象类型
Dictionary_obj_name	VARCHAR2(30)	DDL 语句所操作的数据库对象

		名称
Dictionary_obj_owner	VARCHAR2(30)	DDL 语句所操作的数据库对象所有者名称
Des_encrypted_password	VARCHAR2(2)	正在创建或修改的经过 DES 算法加密的用户口令

例 1：创建触发器，存放有关事件信息。

```
DESC ora_sysevent
DESC ora_login_user

--创建用于记录事件用的表

CREATE TABLE ddl_event
(crt_date timestamp PRIMARY KEY,
 event_name VARCHAR2(20),
 user_name VARCHAR2(10),
 obj_type VARCHAR2(20),
 obj_name VARCHAR2(20));

--创建触犯发器
CREATE OR REPLACE TRIGGER tr_ddl
AFTER DDL ON SCHEMA
BEGIN
  INSERT INTO ddl_event VALUES
  (systimestamp,ora_sysevent, ora_login_user,
   ora_dict_obj_type, ora_dict_obj_name);
END tr_ddl;
```

例 2：创建登录、退出触发器。

```
CREATE TABLE log_event
(user_name VARCHAR2(10),
 address VARCHAR2(20),
 logon_date timestamp,
 logoff_date timestamp);

--创建登录触发器
CREATE OR REPLACE TRIGGER tr_logon
AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO log_event (user_name, address, logon_date)
```

```
  VALUES (ora_login_user, ora_client_ip_address, systimestamp);
END tr_logon;
--创建退出触发器
CREATE OR REPLACE TRIGGER tr_logoff
BEFORE LOGOFF ON DATABASE
BEGIN
  INSERT INTO log_event (user_name, address, logoff_date)
  VALUES (ora_login_user, ora_client_ip_address, systimestamp);
END tr_logoff;
```

8.2.5 使用触发器谓词

ORACLE 提供三个参数 INSERTING, UPDATING, DELETING 用于判断触发了哪些操作。

谓词	*行为*
INSERTING	如果触发语句是 INSERT 语句，则为 TRUE,否则为 FALSE
UPDATING	如果触发语句是 UPDATE 语句，则为 TRUE,否则为 FALSE
DELETING	如果触发语句是 DELETE 语句，则为 TRUE,否则为 FALSE

8.2.6 重新编译触发器

如果在触发器内调用其它函数或过程，当这些函数或过程被删除或修改后，触发器的状态将被标识为无效。当 DML 语句激活一个无效触发器时，ORACLE 将重新编译触发器代码，如果编译时发现错误，这将导致 DML 语句执行失败。

在 PL/SQL 程序中可以调用 ALTER TRIGGER 语句重新编译已经创建的触发器，格式为：

```
ALTER TRIGGER [schema.] trigger_name COMPILE [ DEBUG]
```

其中：DEBUG 选项要器编译器生成 PL/SQL 程序条使其所使用的调试代码。

8.3 删除和使能触发器

- **删除触发器：**

```
DROP TRIGGER trigger_name;
```

当删除其他用户模式中的触发器名称，需要具有 DROP ANY TRIGGER 系统权限，当删除建立在数据库上的触发器时，用户需要具有 ADMINISTER DATABASE TRIGGER 系统权限。

此外，当删除表或视图时，建立在这些对象上的触发器也随之删除。

- **禁用或启用触发器**

数据库 TRIGGER 的状态：

有效状态(ENABLE)：当触发事件发生时，处于有效状态的数据库触发器 TRIGGER 将被触发。

无效状态(DISABLE)：当触发事件发生时，处于无效状态的数据库触发器 TRIGGER 将不会被触发，此时就跟没有这个数据库触发器(TRIGGER) 一样。

数据库 TRIGGER 的这两种状态可以互相转换。格式为：

```
ALTER TIGGER trigger_name [DISABLE | ENABLE ];
```

--例：

```
ALTER TRIGGER emp_view_delete DISABLE;
```

ALTER TRIGGER 语句一次只能改变一个触发器的状态，而 ALTER TABLE 语句则一次能够改变与指定表相关的所有触发器的使用状态。格式为：

```
ALTER TABLE [schema.]table_name {ENABLE|DISABLE} ALL TRIGGERS;
```

--例：使表 EMP 上的所有 TRIGGER 失效：

```
ALTER TABLE emp DISABLE ALL TRIGGERS;
```

8.4 触发器和数据字典

相关数据字典：USER_TRIGGERS、ALL_TRIGGERS、DBA_TRIGGERS

```
SELECT TRIGGER_NAME, TRIGGER_TYPE, TRIGGERING_EVENT,
TABLE_OWNER, BASE_OBJECT_TYPE, REFERENCING_NAMES,
STATUS, ACTION_TYPE
FROM user_triggers;
```

8.5 数据库触发器的应用举例

例 1：创建一个 DML 语句级触发器，当对 emp 表执行 INSERT, UPDATE, DELETE 操作时，它自动更新 dept_summary 表中的数据。由于在 PL/SQL 块中不能直接调用 DDL 语句，所以，利用 ORACLE 内置包 DBMS_UTILITY 中的 EXEC_DDL_STATEMENT 过程，由它执行 DDL 语句创建触发器。

```
CREATE TABLE dept_summary(
Deptno NUMBER(2),
Sal_sum NUMBER(9, 2),
Emp_count NUMBER);
```

```
INSERT INTO dept_summary(deptno, sal_sum, emp_count)
 SELECT deptno, SUM(sal), COUNT(*)
FROM emp
GROUP BY deptno;
```

```
--创建一个 PL/SQL 过程 disp_dept_summary
--在触发器中调用该过程显示 dept_summary 标中的数据。
CREATE OR REPLACE PROCEDURE disp_dept_summary
IS
 Rec dept_summary%ROWTYPE;
 CURSOR c1 IS SELECT * FROM dept_summary;
BEGIN
 OPEN c1;
 FETCH c1 INTO REC;
 DBMS_OUTPUT.PUT_LINE('deptno    sal_sum    emp_count');
 DBMS_OUTPUT.PUT_LINE('-----------------------------------');
 WHILE c1%FOUND LOOP
   DBMS_OUTPUT.PUT_LINE(RPAD(rec.deptno, 6)||
    To_char(rec.sal_sum, '$999,999.99')||
    LPAD(rec.emp_count, 13));
   FETCH c1 INTO rec;
 END LOOP;
 CLOSE c1;
END;
BEGIN
 DBMS_OUTPUT.PUT_LINE('插入前');
 Disp_dept_summary();
 DBMS_UTILITY.EXEC_DDL_STATEMENT('
   CREATE OR REPLACE TRIGGER trig1
    AFTER INSERT OR DELETE OR UPDATE OF sal ON emp
   BEGIN
    DBMS_OUTPUT.PUT_LINE(''正在执行 trig1 触发器...'');
    DELETE FROM dept_summary;
    INSERT INTO dept_summary(deptno, sal_sum, emp_count)
    SELECT deptno, SUM(sal), COUNT(*)
    FROM emp GROUP BY deptno;
   END;
 ');
```

```
INSERT INTO dept(deptno, dname, loc)
```

```
 VALUES(90, 'demo_dept', 'none_loc');
 INSERT INTO emp(ename, deptno, empno, sal)
 VALUES(USER, 90, 9999, 3000);

 DBMS_OUTPUT.PUT_LINE('插入后');
 Disp_dept_summary();

 UPDATE emp SET sal=1000 WHERE empno=9999;
 DBMS_OUTPUT.PUT_LINE('修改后');
 Disp_dept_summary();

 DELETE FROM emp WHERE empno=9999;
 DELETE FROM dept WHERE deptno=90;

 DBMS_OUTPUT.PUT_LINE('删除后');
 Disp_dept_summary();
 DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig1');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);

END;
```

例 2：创建 DML 语句行级触发器。当对 emp 表执行 INSERT, UPDATE, DELETE 操作时，它自动更新 dept_summary 表中的数据。由于在 PL/SQL 块中不能直接调用 DDL 语句，所以，利用 ORACLE 内置包 DBMS_UTILITY 中的 EXEC_DDL_STATEMENT 过程，由它执行 DDL 语句创建触发器。

```
BEGIN
 DBMS_OUTPUT.PUT_LINE('插入前');
 Disp_dept_summary();
 DBMS_UTILITY.EXEC_DDL_STATEMENT(
  'CREATE OR REPLACE TRIGGER trig2_update
    AFTER UPDATE OF sal ON emp
    REFERENCING OLD AS old_emp NEW AS new_emp
    FOR EACH ROW
    WHEN (old_emp.sal != new_emp.sal)
  BEGIN
    DBMS_OUTPUT.PUT_LINE(''正在执行 trig2_update 触发器...'');
    DBMS_OUTPUT.PUT_LINE(''sal 旧值：''|| :old_emp.sal);
    DBMS_OUTPUT.PUT_LINE(''sal 新值：''|| :new_emp.sal);
    UPDATE dept_summary
      SET sal_sum=sal_sum + :new_emp.sal - :old_emp.sal
      WHERE deptno = :new_emp.deptno;
```

```
  END;'
);

DBMS_UTILITY.EXEC_DDL_STATEMENT(
 'CREATE OR REPLACE TRIGGER trig2_insert
   AFTER INSERT ON emp
   REFERENCING NEW AS new_emp
   FOR EACH ROW
 DECLARE
   I NUMBER;
 BEGIN
   DBMS_OUTPUT.PUT_LINE(''正在执行 trig2_insert 触发器...'');
   SELECT COUNT(*) INTO I
   FROM dept_summary WHERE deptno = :new_emp.deptno;
   IF I > 0 THEN
     UPDATE dept_summary
     SET sal_sum=sal_sum+:new_emp.sal,
     Emp_count=emp_count+1
     WHERE deptno = :new_emp.deptno;
   ELSE
     INSERT INTO dept_summary
     VALUES (:new_emp.deptno, :new_emp.sal, 1);
   END IF;
 END;'
);

DBMS_UTILITY.EXEC_DDL_STATEMENT(
 'CREATE OR REPLACE TRIGGER trig2_delete
   AFTER DELETE ON emp
   REFERENCING OLD AS old_emp
   FOR EACH ROW
 DECLARE
   I NUMBER;
 BEGIN
   DBMS_OUTPUT.PUT_LINE(''正在执行 trig2_delete 触发器...'');
   SELECT emp_count INTO I
   FROM dept_summary WHERE deptno = :old_emp.deptno;
   IF I >1 THEN
     UPDATE dept_summary
     SET sal_sum=sal_sum - :old_emp.sal,
     Emp_count=emp_count - 1
     WHERE deptno = :old_emp.deptno;
   ELSE
     DELETE FROM dept_summary WHERE deptno = :old_emp.deptno;
```

```
    END IF;
  END;'
 );

 INSERT INTO dept(deptno, dname, loc)
  VALUES(90, 'demo_dept', 'none_loc');
 INSERT INTO emp(ename, deptno, empno, sal)
  VALUES(USER, 90, 9999, 3000);
 INSERT INTO emp(ename, deptno, empno, sal)
  VALUES(USER, 90, 9998, 2000);
 DBMS_OUTPUT.PUT_LINE('插入后');
 Disp_dept_summary();

 UPDATE emp SET sal = sal*1.1 WHERE deptno=90;
 DBMS_OUTPUT.PUT_LINE('修改后');
 Disp_dept_summary();

 DELETE FROM emp WHERE deptno=90;
 DELETE FROM dept WHERE deptno=90;
 DBMS_OUTPUT.PUT_LINE('删除后');
 Disp_dept_summary();

 DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig2_update');
 DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig2_insert');
 DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig2_delete');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

例 3：利用 ORACLE 提供的条件谓词 INSERTING、UPDATING 和 DELETING 创建与例 2 具有相同功能的触发器。

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('插入前');
  Disp_dept_summary();
  DBMS_UTILITY.EXEC_DDL_STATEMENT(
    'CREATE OR REPLACE TRIGGER trig2
       AFTER INSERT OR DELETE OR UPDATE OF sal
ON emp
       REFERENCING OLD AS old_emp NEW AS new_emp
       FOR EACH ROW
    DECLARE
       I NUMBER;
```

```
    BEGIN
      IF UPDATING AND :old_emp.sal != :new_emp.sal THEN
      DBMS_OUTPUT.PUT_LINE(''正在执行 trig2 触发器...'');
        DBMS_OUTPUT.PUT_LINE(''sal 旧值： ''|| :old_emp.sal);
        DBMS_OUTPUT.PUT_LINE(''sal 新值： ''|| :new_emp.sal);
        UPDATE dept_summary
          SET sal_sum=sal_sum + :new_emp.sal - :old_emp.sal
        WHERE deptno = :new_emp.deptno;
      ELSIF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE(''正在执行 trig2 触发器...'');
        SELECT COUNT(*) INTO I
    FROM dept_summary
    WHERE deptno = :new_emp.deptno;
        IF I > 0 THEN
          UPDATE dept_summary
     SET sal_sum=sal_sum+:new_emp.sal,
        Emp_count=emp_count+1
     WHERE deptno = :new_emp.deptno;
      ELSE
     INSERT INTO dept_summary
      VALUES (:new_emp.deptno, :new_emp.sal, 1);
    END IF;
   ELSE
    DBMS_OUTPUT.PUT_LINE(''正在执行 trig2 触发器...'');
    SELECT emp_count INTO I
    FROM dept_summary WHERE deptno = :old_emp.deptno;
   IF I > 1 THEN
    UPDATE dept_summary
    SET sal_sum=sal_sum - :old_emp.sal,
    Emp_count=emp_count - 1
    WHERE deptno = :old_emp.deptno;
   ELSE
      DELETE FROM dept_summary
      WHERE deptno = :old_emp.deptno;
   END IF;
  END IF;
  END;'
);

INSERT INTO dept(deptno, dname, loc)
  VALUES(90, 'demo_dept', 'none_loc');
INSERT INTO emp(ename, deptno, empno, sal)
  VALUES(USER, 90, 9999, 3000);
INSERT INTO emp(ename, deptno, empno, sal)
```

```
  VALUES(USER, 90, 9998, 2000);
 DBMS_OUTPUT.PUT_LINE('插入后');
 Disp_dept_summary();

 UPDATE emp SET sal = sal*1.1 WHERE deptno=90;
 DBMS_OUTPUT.PUT_LINE('修改后');
 Disp_dept_summary();

 DELETE FROM emp WHERE deptno=90;
 DELETE FROM dept WHERE deptno=90;
 DBMS_OUTPUT.PUT_LINE('删除后');
 Disp_dept_summary();

 DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig2');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE||'---'||SQLERRM);
END;
```

例 4：创建 INSTEAD OF 触发器。首先创建一个视图 myview，由于该视图是复合查询所产生的视图，所以不能执行 DML 语句。根据用户对视图所插入的数据判断需要将数据插入到哪个视图基表中，然后对该基表执行插入操作。

```
DECLARE
  No NUMBER;
  Name VARCHAR2(20);
BEGIN
  DBMS_UTILITY.EXEC_DDL_STATEMENT('
    CREATE OR REPLACE VIEW myview AS
      SELECT empno, ename, ''E'' type FROM emp
      UNION
      SELECT dept.deptno, dname, ''D'' FROM dept
  ');
  -- 创建 INSTEAD OF 触发器 trigger3;
  DBMS_UTILITY.EXEC_DDL_STATEMENT('
    CREATE OR REPLACE TRIGGER trig3
      INSTEAD OF INSERT ON myview
      REFERENCING NEW n
      FOR EACH ROW
    DECLARE
      Rows INTEGER;
    BEGIN
      DBMS_OUTPUT.PUT_LINE(''正在执行 trig3 触发器...'');
      IF :n.type = ''D'' THEN
```

```
          SELECT COUNT(*) INTO rows
            FROM dept WHERE deptno = :n.empno;
          IF rows = 0 THEN
            DBMS_OUTPUT.PUT_LINE(''向 dept 表中插入数据…'');
            INSERT INTO dept(deptno, dname, loc)
              VALUES (:n.empno, :n.ename, ''none'');
          ELSE
            DBMS_OUTPUT.PUT_LINE(''编号为''|| :n.empno||
             ''的部门已存在，插入操作失败！'');
           END IF;
        ELSE
          SELECT COUNT(*) INTO rows
            FROM emp WHERE empno = :n.empno;
          IF rows = 0 THEN
            DBMS_OUTPUT.PUT_LINE(''向 emp 表中插入数据…'');
            INSERT INTO emp(empno, ename)
              VALUES(:n.empno, :n.ename);
          ELSE
            DBMS_OUTPUT.PUT_LINE(''编号为''|| :n.empno||
             ''的人员已存在，插入操作失败!'');
          END IF;
        END IF;
      END;
  ');

  INSERT INTO myview VALUES (70, 'demo', 'D');
  INSERT INTO myview VALUES (9999, USER, 'E');
  SELECT deptno, dname INTO no, name FROM dept WHERE deptno=70;
  DBMS_OUTPUT.PUT_LINE('员工编号：'||TO_CHAR(no)||'姓名：'||name);
  SELECT empno, ename INTO no, name FROM emp WHERE empno=9999;
  DBMS_OUTPUT.PUT_LINE('部门编号：'||TO_CHAR(no)||'姓名：'||name);
 DELETE FROM emp WHERE empno=9999;
 DELETE FROM dept WHERE deptno=70;
  DBMS_UTILITY.EXEC_DDL_STATEMENT('DROP TRIGGER trig3');
END;
```

例 5：利用 ORACLE 事件属性函数，创建一个系统事件触发器。首先创建一个事件日志表 eventlog，由它存储用户在当前数据库中所创建的数据库对象，以及用户的登陆和注销、数据库的启动和关闭等事件，之后创建 trig4_ddl、trig4_before 和 trig4_after 触发器，它们调用事件属性函数将各个事件记录到 eventlog 数据表中。

```
BEGIN
  -- 创建用于记录事件日志的数据表
```

```
  DBMS_UTILITY.EXEC_DDL_STATEMENT('
    CREATE TABLE eventlog(
      Eventname VARCHAR2(20) NOT NULL,
      Eventdate date default sysdate,
      Inst_num NUMBER NULL,
      Db_name VARCHAR2(50) NULL,
      Srv_error NUMBER NULL,
      Username VARCHAR2(30) NULL,
      Obj_type VARCHAR2(20) NULL,
      Obj_name VARCHAR2(30) NULL,
      Obj_owner VARCHAR2(30) NULL
    )
  ');

  -- 创建 DDL 触发器 trig4_ddl
  DBMS_UTILITY.EXEC_DDL_STATEMENT('
    CREATE OR REPLACE TRIGGER trig4_ddl
      AFTER CREATE OR ALTER OR DROP
ON DATABASE
    DECLARE
      Event VARCHAR2(20);
      Typ VARCHAR2(20);
      Name VARCHAR2(30);
      Owner VARCHAR2(30);
    BEGIN
      -- 读取 DDL 事件属性
      Event := SYSEVENT;
      Typ := DICTIONARY_OBJ_TYPE;
      Name := DICTIONARY_OBJ_NAME;
      Owner := DICTIONARY_OBJ_OWNER;
      --将事件属性插入到事件日志表中
      INSERT INTO scott.eventlog(eventname, obj_type, obj_name, obj_owner)
        VALUES(event, typ, name, owner);
    END;
  ');

  -- 创建 LOGON、STARTUP 和 SERVERERROR 事件触发器
  DBMS_UTILITY.EXEC_DDL_STATEMENT('
    CREATE OR REPLACE TRIGGER trig4_after
      AFTER LOGON OR STARTUP OR SERVERERROR
   ON DATABASE
    DECLARE
      Event VARCHAR2(20);
      Instance NUMBER;
```

```
        Err_num NUMBER;
        Dbname VARCHAR2(50);
        User VARCHAR2(30);
      BEGIN
        Event := SYSEVENT;
        IF event = ''LOGON'' THEN
          User := LOGIN_USER;
          INSERT INTO eventlog(eventname, username)
            VALUES(event, user);
        ELSIF event = ''SERVERERROR'' THEN
          Err_num := SERVER_ERROR(1);
          INSERT INTO eventlog(eventname, srv_error)
            VALUES(event, err_num);
        ELSE
          Instance := INSTANCE_NUM;
          Dbname := DATABASE_NAME;
          INSERT INTO eventlog(eventname, inst_num, db_name)
            VALUES(event, instance, dbname);
    END IF;
  END;
');

-- 创建 LOGOFF 和 SHUTDOWN 事件触发器
DBMS_UTILITY.EXEC_DDL_STATEMENT('
  CREATE OR REPLACE TRIGGER trig4_before
    BEFORE LOGOFF OR SHUTDOWN
    ON DATABASE
  DECLARE
    Event VARCHAR2(20);
    Instance NUMBER;
    Dbname VARCHAR2(50);
    User VARCHAR2(30);
  BEGIN
    Event := SYSEVENT;
    IF event = ''LOGOFF'' THEN
      User := LOGIN_USER;
      INSERT INTO eventlog(eventname, username)
        VALUES(event, user);
    ELSE
      Instance := INSTANCE_NUM;
      Dbname := DATABASE_NAME;
      INSERT INTO eventlog(eventname, inst_num, db_name)
        VALUES(event, instance, dbname);
    END IF;
```

```
   END;
 ');
END;

CREATE TABLE mydata(mydate NUMBER);
CONNECT SCOTT/TIGER

COL eventname FORMAT A10
COL eventdate FORMAT A12
COL username FORMAT A10
COL obj_type FORMAT A15
COL obj_name FORMAT A15
COL obj_owner FORMAT A10
SELECT eventname, eventdate, obj_type, obj_name, obj_owner, username, Srv_error
 FROM eventlog;

DROP TRIGGER trig4_ddl;
DROP TRIGGER trig4_before;
DROP TRIGGER trig4_after;
DROP TABLE eventlog;
DROP TABLE mydata;
```

8.6 数据库触发器的应用实例

用户可以使用数据库触发器实现各种功能：

- 复杂的审计功能；

例：将 EMP 表的变化情况记录到 AUDIT_TABLE 和 AUDIT_TABLE_VALUES 中。

```
CREATE TABLE audit_table(
   Audit_id    NUMBER,
   User_name VARCHAR2(20),
   Now_time DATE,
   Terminal_name VARCHAR2(10),
   Table_name VARCHAR2(10),
   Action_name VARCHAR2(10),
   Emp_id NUMBER(4));

CREATE TABLE audit_table_val(
   Audit_id NUMBER,
   Column_name VARCHAR2(10),
   Old_val NUMBER(7,2),
   New_val NUMBER(7,2));
```

```
CREATE SEQUENCE audit_seq
   START WITH 1000
   INCREMENT BY 1
   NOMAXVALUE
   NOCYCLE NOCACHE;

CREATE OR REPLACE TRIGGER audit_emp
   AFTER INSERT OR UPDATE OR DELETE ON emp
   FOR EACH ROW
DECLARE
   Time_now DATE;
   Terminal CHAR(10);
BEGIN
   Time_now:=sysdate;
   Terminal:=USERENV('TERMINAL');
   IF INSERTING THEN
      INSERT INTO audit_table
   VALUES(audit_seq.NEXTVAL, user, time_now,
         terminal, 'EMP', 'INSERT', :new.empno);
   ELSIF DELETING THEN
      INSERT INTO audit_table
   VALUES(audit_seq.NEXTVAL, user, time_now,
         terminal, 'EMP', 'DELETE', :old.empno);
   ELSE
      INSERT INTO audit_table
   VALUES(audit_seq.NEXTVAL, user, time_now,
         terminal, 'EMP', 'UPDATE', :old.empno);
      IF UPDATING('SAL') THEN
         INSERT INTO audit_table_val
            VALUES(audit_seq.CURRVAL, 'SAL', :old.sal, :new.sal);
      ELSE UPDATING('DEPTNO')
         INSERT INTO audit_table_val
            VALUES(audit_seq.CURRVAL, 'DEPTNO', :old.deptno, :new.deptno);
      END IF;
   END IF;
END;
```

- 增强数据的完整性管理；

例：修改 DEPT 表的 DEPTNO 列时，同时把 EMP 表中相应的 DEPTNO 也作相应的修改；

```
CREATE SEQUENCE update_sequence
```

```
   INCREMENT BY 1
   START WITH 1000
   MAXVALUE 5000 CYCLE;

ALTER TABLE emp
   ADD update_id NUMBER;

CREATE OR REPLACE PACKAGE integritypackage AS
   Updateseq NUMBER;
END integritypackage;

CREATE OR REPLACE PACKAGE BODY integritypackage AS
END integritypackage;

CREATE OR REPLACE TRIGGER dept_cascade1
   BEFORE UPDATE OF deptno ON dept
DECLARE
   Dummy NUMBER;
BEGIN
   SELECT update_sequence.NEXTVAL INTO dummy FROM dual;
   Integritypackage.updateseq:=dummy;
END;

CREATE OR REPLACE TRIGGER dept_cascade2
   AFTER DELETE OR UPDATE OF deptno ON dept
   FOR EACH ROW
BEGIN
   IF UPDATING THEN
      UPDATE emp SET deptno=:new.deptno,
    update_id=integritypackage.updateseq
      WHERE emp.deptno=:old.deptno AND update_id IS NULL;
   END IF;
   IF DELETING THEN
      DELETE FROM emp
         WHERE emp.deptno=:old.deptno;
   END IF;
END;

CREATE OR REPLACE TRIGGER dept_cascade3
   AFTER UPDATE OF deptno ON dept
BEGIN
   UPDATE emp SET update_id=NULL
      WHERE update_id=integritypackage.updateseq;
END;
```

```
SELECT * FROM EMP ORDER BY DEPTNO;
UPDATE dept SET deptno=25 WHERE deptno=20;
```

- 帮助实现安全控制；

例：保证对 EMP 表的修改仅在工作日的工作时间；

```
CREATE TABLE company_holidays(day DATE);

INSERT INTO company_holidays
   VALUES(sysdate);
INSERT INTO company_holidays
VALUES(TO_DATE('21-10 月-01', 'DD-MON-YY'));

CREATE OR REPLACE TRIGGER emp_permit_change
   BEFORE INSERT OR DELETE OR UPDATE ON emp
DECLARE
   Dummy NUMBER;
   Not_on_weekends EXCEPTION;
   Not_on_holidays EXCEPTION;
   Not_working_hours EXCEPTION;
BEGIN
   /* check for weekends */
IF TO_CHAR(SYSDATE, 'DAY') IN ('星期六', '星期日') THEN
   RAISE not_on_weekends;
END IF;
   /* check for company holidays */
SELECT COUNT(*) INTO dummy FROM company_holidays
   WHERE TRUNC(day)=TRUNC(SYSDATE);
IF dummy >0 THEN
   RAISE not_on_holidays;
END IF;
   /* check for work hours(8:00 AM to 18:00 PM */
IF (TO_CHAR(SYSDATE,'HH24')<8 OR TO_CHAR(SYSDATE, 'HH24')>18) THEN
  RAISE not_working_hours;
END IF;
EXCEPTION
  WHEN not_on_weekends THEN
   RAISE_APPLICATION_ERROR(-20324,
'May not change employee table during the weekends');
```

```
  WHEN not_on_holidays THEN
    RAISE_APPLICATION_ERROR(-20325,
'May not change employee table during a holiday');
  WHEN not_working_hours THEN
    RAISE_APPLICATION_ERROR(-20326,
'May not change employee table during no_working hours');
END;
```