

PL/SQL 程序设计



```
Text editor
1.pl/sql 基本的语法格式
2.记录类型 type ... is record(,,,);
3.流程控制: |
  3.1 条件判断 (两种)
    方式一: if ... then elsif then ... else ... end if;
    方式二: case ... when ... then ... end;
  3.2 循环结构 (三种)
    方式一: loop ... exit when ... end loop;
    方式二: while ... loop ... end loop;
    方式三: for i in ... loop ... end loop;
  3.3 goto 、exit
4.游标的使用 (类似于java中的Iterator)
5.异常的处理 (三种方式)
6.会写一个存储函数 (有返回值)、存储过程 (没有返回值)
7.会写一个触发器
```

目 录

| | | |
|--------|-------------------------|----|
| 第一章 | PL/SQL 程序设计简介..... | 3 |
| §1.2 | SQL 与 PL/SQL | 3 |
| §1.2.1 | 什么是 PL/SQL?..... | 3 |
| §1.2.1 | PL/SQL 的好处..... | 3 |
| §1.2.2 | PL/SQL 可用的 SQL 语句 | 4 |
| §1.3 | 运行 PL/SQL 程序 | 4 |
| 第二章 | PL/SQL 块结构和组成元素 | 5 |
| §2.1 | PL/SQL 块 | 5 |
| §2.2 | PL/SQL 结构 | 5 |
| §2.3 | 标识符 | 5 |
| §2.4 | PL/SQL 变量类型 | 6 |
| §2.4.1 | 变量类型 | 6 |
| §2.4.2 | 复合类型 | 7 |
| §2.4.3 | 使用%ROWTYPE | 9 |
| §2.4.4 | PL/SQL 表(嵌套表) | 9 |
| §2.5 | 运算符和表达式(数据定义)..... | 10 |
| §2.5.1 | 关系运算符 | 10 |
| §2.5.2 | 一般运算符 | 10 |
| §2.5.3 | 逻辑运算符 | 11 |
| §2.6 | 变量赋值 | 11 |
| §2.6.1 | 字符及数字运算特点 | 11 |
| §2.6.2 | BOOLEAN 赋值 | 11 |
| §2.6.3 | 数据库赋值 | 11 |
| §2.6.4 | 可转换的类型赋值 | 12 |
| §2.7 | 变量作用范围及可见性 | 12 |
| §2.8 | 注释 | 12 |
| §2.9 | 简单例子 | 13 |
| §2.9.1 | 简单数据插入例子 | 13 |
| §2.9.2 | 简单数据删除例子 | 13 |
| 第三章 | PL/SQL 流程控制语句 | 14 |
| §3.1 | 条件语句 | 14 |
| §3.2 | CASE 表达式 | 15 |
| §3.3 | 循环 | 15 |
| §3.3 | 标号和 GOTO | 17 |
| §3.4 | NULL 语句 | 18 |
| 第四章 | 游标的使用 | 19 |
| §4.1 | 游标概念 | 19 |
| §4.1.1 | 处理显式游标 | 19 |
| §4.1.2 | 处理隐式游标 | 23 |

| | |
|---|----|
| §4.1.3 关于 NO_DATA_FOUND 和 %NOTFOUND 的区别 | 24 |
| §4.1.4 游标修改和删除操作..... | 24 |
| 第五章 异常错误处理 | 26 |
| §5.1 异常处理概念 | 26 |
| §5.1.1 预定义的异常处理 | 26 |
| §5.1.2 非预定义的异常处理..... | 27 |
| §5.1.3 用户自定义的异常处理..... | 28 |
| §5.2 在 PL/SQL 中使用 SQLCODE, SQLERRM..... | 29 |
| 第六章 存储函数和过程 | 31 |
| §6.1 引言 | 31 |
| §6.2 创建函数 | 31 |
| §6.3 存储过程 | 35 |
| §6.3.1 创建过程..... | 35 |
| §6.3.2 调用存储过程 | 36 |
| §6.3.3 AUTHID..... | 37 |
| §6.3.4 开发存储过程步骤 | 38 |
| §6.3.5 删除过程和函数 | 38 |
| 第七章 包的创建和应用 | 40 |
| §7.1 引言 | 40 |
| §7.2 包的定义 | 40 |
| §7.3 包的开发步骤..... | 41 |
| §7.4 包定义的说明..... | 41 |
| §7.5 子程序重载 | 49 |
| §7.6 删除包 | 51 |
| §7.7 包的管理 | 51 |
| 第八章 触发器 | 52 |
| §8.1 触发器类型 | 52 |
| §8.1.1 DML 触发器 | 52 |
| §8.1.2 替代触发器..... | 52 |
| §8.1.3 系统触发器..... | 52 |
| §8.2 创建触发器 | 53 |
| §8.2.1 触发器触发次序 | 54 |
| §8.2.2 创建 DML 触发器..... | 54 |
| §8.2.3 创建替代(INSTEAD OF)触发器 | 54 |
| §8.2.3 创建系统事件触发器..... | 56 |
| §8.2.4 系统触发器事件属性..... | 56 |
| §8.2.5 使用触发器谓词 | 57 |
| §8.2.6 重新编译触发器 | 57 |
| §8.3 删除和使能触发器 | 57 |

第一章 PL/SQL 程序设计简介

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。

本章主要重点：

- PL/SQL 概述
- PL/SQL 块结构
- PL/SQL 流程
- 运算符和表达式
- 游标
- 异常处理
- 数据库存储过程和函数
- 包
- 触发器

§1.2 SQL 与 PL/SQL

§1.2.1 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。

§1.2.1 PL/SQL 的好处

§1.2.1.1 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。

§1.2.1.2 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，也可以向服务器发 SQL 命令或激活服务器端的 PL/SQL 程序运行。

§1.2.2 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。
- 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。

§1.3 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。

在命令窗口中执行需要先执行
set serveroutput on

```
--declare
--声明的变量、类型、游标
begin
--程序的执行部分（类似于java里的main()方法）
  dbms_output.put_line('helloworld');
--exception
--针对begin块中出现的异常，提供处理的机制
--when .... then ...
--when .... then ...
end;
```

```
declare
--声明变量
  v_sal varchar2(20);
begin
--sql语句的操作: select ... into ... from ... where ...
  select salary into v_sal from employees where employee_id = 100;
--打印
  dbms_output.put_line(v_sal);
end;
```

```
declare
--声明变量
  v_sal employees.salary%type;
  v_email employees.email%type;
  v_hire_date employees.hire_date%type;
begin
--sql语句的操作: select ... into ... from ... where ...
  select salary, email, hire_date into v_sal, v_email, v_hire_date from employees where employee_id = 100;
--打印
  dbms_output.put_line(v_sal || ',' || v_email || ',' || v_hire_date);
end;
```

第二章 PL/SQL 块结构和组成元素

§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 执行部分是必须的。

PL/SQL 块可以分为三类：

1. **无名块**：动态构造，只能执行一次。
2. **子程序**：存储在数据库中的**存储过程、函数**及包等。当在数据库上建立好后可以在其它程序中调用它们。
3. **触发器**：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。

§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 KING 的记录；

```
DECLARE
  Ename varchar2(20) := 'KING';
BEGIN
  DELETE FROM 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 | Name_ _cursor | Emp_cursor |
| 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。下表是 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 | 小数, NUMBER 的子类型 高精度实数 整数, NUMBER 的子类型 | | |

| | | | | |
|---------|---|--|---|-----------|
| | precision Integer Int Numeric Real Small int | 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 | | |

§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

记录类型是把逻辑相关的数据作为一个单元存储起来, 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。

定义记录类型语法如下:

```
TYPE record_type IS RECORD(
    Field1 type1  [NOT NULL]  [:= exp1 ],
    Field2 type2  [NOT NULL]  [:= exp2 ],
    ...
    Fieldn typen  [NOT NULL]  [:= expn ] );
```

例 1 :



```
declare
--声明一个记录类型 声明record
type emp_record is record(
v_sal employees.salary%type,
v_email employees.email%type,
v_hire_date employees.hire_date%type
);
--定义一个记录类型的成员变量 用record产生一个变量
v_emp_record emp_record;
begin
--sql语句的操作: select ... into ... from ... where ...
select salary,email,hire_date into v_emp_record from employees where employee_id = 100;
--打印 使用这个record
dbms_output.put_line(v_emp_record.v_sal||','||v_emp_record.v_email||','||v_emp_record.v_hire_date);
end;
```



```
declare
    type test_rec is record(
        l_name varchar2(30),
        d_id number(4));
    v_emp test_rec;
begin
    v_emp.l_name := 'Tom';
    v_emp.d_id := 1234;
    dbms_output.put_line(v_emp.l_name || ', ' || v_emp.d_id);
end;
```

或

```
declare
    type test_rec is record(
        l_name varchar2(30),
        d_id number(4));
    v_emp test_rec;
begin
    select last_name, department_id into v_emp
    from employees
    where employee_id = 200;

    dbms_output.put_line(v_emp.l_name || ', ' || v_emp.d_id);
end;
```

提示: 1) `DBMS_OUTPUT.PUT_LINE` 过程的功能类似于 Java 中的 `System.out.println()` 直接将输出结果送到标准输出中。

2) 在使用上述过程之前必须将 SQL * PLUS 的环境参数 `SERVEROUTPUT` 设置为 `ON`, 否则将看不到输出结果: **set serveroutput on**

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。

§2.4.2.2 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的类型相同,或者与数据库表的某个列的数据类型相同,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。

例 2:

```
declare
    type test_rec is record(
        l_name employees.last_name%type,
        d_id employees.department_id%type);
    v_emp test_rec;
begin
    select last_name,
           department_id into v_emp
    from employees where employee_id = 200;
    dbms_output.put_line(v_emp.l_name || ', ' || v_emp.d_id);
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。

例 3:

```
declare
    v_emp employees%rowtype;
begin
    select * into v_emp
    from employees where employee_id = 200;
    dbms_output.put_line(v_emp.last_name
                        || ', ' || v_emp.department_id
                        || ', ' || v_emp.hire_date);
end;
```

§2.4.4 PL/SQL 表(嵌套表)

PL/SQL 程序可使用嵌套表类型创建具有一个或多个列和无限行的变量, 这很像数据库中的表. 声明嵌套表类型的一般语法如下:

TYPE type_name IS TABLE OF
 {datatype | {variable | table.column} % type | table%rowtype};

| 方法 | 描述 |
|---------------|---|
| EXISTS(n) | Return TRUE if the nth element in a PL/SQL table exists; |
| COUNT | Returns the number of elements that a PL/SQL table currently contains; |
| FIRST LAST | Return the first and last (smallest and lastest) index numbers in a PL/SQL table. Returns NULL if the PL/SQL table is empty. |
| PRIOR(n) | Returns the index number that precedes index n in a PL/SQL table; |
| NEXT(N) | Returns the index number that succeeds index n in a PL/SQL table; |
| TRIM | TRIM removes one element from the end of a PL/SQL table. TRIM(n) removes n element from the end of a PL/SQL table. |

| | |
|--------|--|
| DELETE | <p>DELETE removes all elements from a PL/SQL table.</p> <p>DELETE(n) removes the nth elements from a PL/SQL table.</p> <p>DELETE(m, n) removes all elements in the range m to n from a PL/SQL table.</p> |
|--------|--|

例 4:

```

declare
    type dep_table_type is table of departments%rowtype;
    my_dep_table dep_table_type := dep_table_type();
begin
    my_dep_table.extend(5);

    for i in 1 .. 5 loop
        select * into my_dep_table(i)
        from departments
        where department_id = 200 + 10 * i;
    end loop;

    dbms_output.put_line(my_dep_table.count());
    dbms_output.put_line(my_dep_table(1).department_id);
end;
```

说明: 1) 在使用嵌套表之前必须先使用该集合的构造器初始化它. PL/SQL 自动提供一个带有相同名字的构造器作为集合类型.

2) 嵌套表可以有任意数量的行. 表的大小在必要时可动态地增加或减少: extend(x) 方法添加 x 个空元素到集合末尾; trim(x) 方法为去掉集合末尾的 x 个元素.

§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 | 取反,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: $NULL + \langle \text{数字} \rangle = NULL$

空值加（连接）字符, 结果为字符: $NULL || \langle \text{字符串} \rangle = \langle \text{字符串} \rangle$

§2.6.2 BOOLEAN 赋值

布尔值只有 TRUE, FALSE 及 NULL 三个值。

§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 9:

```
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。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。

§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); -- 工资变量。
```

- 使用 **/* */** 来加一行或多行注释，如：

```
/* **** */
/* 文件名: department_salary.sql */
/* **** */
```

提示：被解释存放在数据库中的 **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 11:

/* 本例子仅是一个简单的插入，不是实际应用。 */

```
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 12:

/* 本例子仅是一个简单的删除例子，不是实际应用。 */

```
DECLARE
    v_empno   number(4) := 8888;
BEGIN
    DELETE FROM emp WHERE empno=v_empno;
    COMMIT;
END;
```

第三章 PL/SQL 流程控制语句

介绍 PL/SQL 的流程控制语句, 包括如下三类:

- 控制语句: IF 语句
- 循环语句: LOOP 语句, EXIT 语句
- 顺序语句: GOTO 语句, NULL 语句

§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 emp.empno%TYPE;
    v_salary emp.sal%TYPE;
    v_comment VARCHAR2(35);
BEGIN
    SELECT sal INTO v_salary FROM emp WHERE empno=v_empno;
    IF v_salary<1500 THEN
        v_comment:= 'Fairly less';
```

```
ELSIF v_salary < 3000 THEN
    V_comment := 'A little more';
ELSE
    V_comment := 'Lots of salary';
END IF;
DBMS_OUTPUT.PUT_LINE(V_comment);

END;
```

§3.2 CASE 表达式 case表达式 selector不可以是离散的值，then中不能有赋值操作。

```
CASE selector
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2

    WHEN expressionN THEN resultN
    [ ELSE resultN+1]
END;
```

例 2:

```
DECLARE
    V_grade char(1);
    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;
```

§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 3.

```
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(相较 1，推荐使用 2)

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

例 4.

```
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

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

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。

for循环是[]闭区间的，使用reverse可以反着输出，从大到小进行循环

例 5.

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

END;

例 6.

```
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;
```

§3.3 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 */
```

例 7:

```
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 I_ENDOfLOOP;
```

```
        END IF;
```

```
    END LOOP;
```

```
    <<I_ENDOfLOOP>>
```

```
        DBMS_OUTPUT.PUT_LINE('V_counter 的当前值为:'||V_counter);
```

```
END ;
```

§3.4 NULL 语句

在 PL/SQL 程序中，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，可以使某些语句变得有意义，提高程序的可读性。如：

```
DECLARE
    ...
BEGIN
    ...
    IF v_num IS NULL THEN
        GOTO print1;
    END IF;
    ...
    <<print1>>
    NULL; -- 不需要处理任何数据。
END;
```

第四章 游标的使用

在 PL/SQL 程序中，对于处理多行记录的事务经常使用游标来实现。

§4.1 游标概念

为了处理 SQL 语句，ORACLE 必须分配一片叫上下文(context area)的区域来处理所必需的信息，其中包括要处理的行的数目，一个指向语句被分析以后的表示形式的指针以及查询的活动集(active set)。

游标是一个指向上下文的句柄(handle)或指针。通过游标，PL/SQL 可以控制上下文区和处理语句时上下文区会发生些什么事情。

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]...)] **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。

- **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};

- 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
  --定义游标
  cursor c_cursor is select last_name, salary
                      from employees
                      where rownum < 11
                      order by salary;
  v_name employees.last_name%type;
  v_sal  employees.salary%type;
begin
  --打开游标
  open c_cursor;
  --提取游标数据
  fetch c_cursor into v_name, v_sal;

  while c_cursor %found loop
    dbms_output.put_line(v_name || ': ' || v_sal);
    fetch c_cursor into v_name, v_sal;
  end loop;
  --关闭游标
  close c_cursor;
end;
```

例 2. 游标参数的传递方法。

```
declare
  --定义游标
  cursor c_cursor(emp_no number default 11) is select last_name, salary
                      from employees
                      where rownum < emp_no
                      order by salary;
  v_name employees.last_name%type;
  v_sal  employees.salary%type;
begin
  --打开游标
  open c_cursor;
  --提取游标数据
  fetch c_cursor into v_name, v_sal;

  while c_cursor %found loop
    dbms_output.put_line(v_name || ': ' || v_sal);
    fetch c_cursor into v_name, v_sal;
  end loop;
  --关闭游标
  close c_cursor;
end;
```

或

```
declare
  --定义游标
  cursor c_cursor(emp_no number default 11) is select last_name, salary
        from employees
        where rownum < emp_no
        order by salary;
  v_name employees.last_name%type;
  v_sal  employees.salary%type;
begin
  --打开游标
  open c_cursor(emp_no => 20);
  --提取游标数据
  fetch c_cursor into v_name, v_sal;

  while c_cursor %found loop
    dbms_output.put_line(v_name || ': ' || v_sal);
    fetch c_cursor into v_name, v_sal;
  end loop;
  --关闭游标
  close c_cursor;
end;
```

2.游标属性

- %FOUND** 布尔型属性，当最近一次读记录时成功返回,则值为 **TRUE**;
- %NOTFOUND** 布尔型属性，与**%FOUND** 相反;
- %ISOPEN** 布尔型属性，当游标已打开时返回 **TRUE**;
- %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 3000 的员工工资调为 3000。

```
declare
  v_eid 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_eid, v_sal;
    exit when c_cursor %notfound;
    if v_sal <= 3000 then
      update employees set salary = 3000
        where employee_id = v_eid;
      dbms_output.put_line('员工: ' || v_eid || '工资已更新');
    end if;
  end loop;
  dbms_output.put_line('记录数为: ' || c_cursor %rowcount);
  close c_cursor;
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 4：

```
declare  
    cursor c_emp is select last_name, salary sal  
                    from employees;  
begin  
    for v_emp in c_emp loop  
        dbms_output.put_line(v_emp.last_name  
                             || ', ' || v_emp.sal);  
    end loop;  
end;
```

例 5：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_emp(dep_id number default 50) is  
        select last_name, salary sal  
        from employees  
        where department_id = dep_id;  
begin  
    for v_emp in c_emp loop  
        dbms_output.put_line(v_emp.last_name  
                             || ', ' || v_emp.sal);  
    end loop;  
end;
```

或

```
declare
    cursor c_emp(dep_id number default 50) is
        select last_name, salary sal
        from employees
        where department_id = dep_id;
begin
    for v_emp in c_emp(80) loop
        dbms_output.put_line(v_emp.last_name
                               || ', ' || v_emp.sal);
    end loop;
end;
```

例 6: 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 v_emp in (select last_name, salary from employees) loop
        dbms_output.put_line(v_emp.last_name
                               || ', ' || v_emp.salary);
    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%

隐式游标属性

- SQL%FOUND 布尔型属性,当最近一次读记录时成功返回，则值为 TRUE;
- SQL%NOTFOUND 布尔型属性,与%FOUND 相反;
- SQL %ROWCOUNT 数字型属性, 返回已从游标中读取得记录数;
- SQL %ISOPEN 布尔型属性, 取值总是 FALSE。SQL 命令执行完毕立即关闭隐式游标。

例 7: 更新指定员工信息，如果该员工没有找到，则打印“查无此人”信息。


```
declare
    v_name employees.last_name%type;
    v_id employees.employee_id%type := &v_id;
begin
    update employees
    set last_name = 'xx'
    where employee_id = v_id;

    if SQL%NOTFOUND then
        dbms_output.put_line('查无此人');
    end if;
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 游标修改和删除操作

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 ... FROM ... 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 8：从 EMPLOYEES 表中查询某部门的员工情况，将其工资最低定为 3000;

```
declare
  v_dep_id employees.department_id%type := &v_dep_id;
  cursor emp_cursor is
    select last_name, salary
    from employees
    where department_id = v_dep_id
    for update nowait;
begin
  for emp_rec in emp_cursor loop
    if emp_rec.salary < 3000 then
      update employees set salary = 3000
      where current of emp_cursor;
    end if;
  end loop;
end;
```

第五章 异常错误处理

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。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 | 试图破坏一个唯一性限制 |
| 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 | 试图打开一个已存在的游标 |
| 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 300，则加 100；对 NO_DATA_FOUND 异常, TOO_MANY_ROWS 进行处理。

```

declare
    v_empno employees.employee_id%type := &v_empno;
    v_sal employees.salary%type;
begin
    select salary into v_sal
    from employees
    where employee_id = v_empno
    for update;

    if v_sal <= 3000 then
        update employees set salary = salary + 3000
        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 || '员工工资不需要更新');
    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('其他错误');
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 PRAGMA 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。

```
declare
    v_depno dept.deptno%type := &v_depno;
    deptno_remaining exception;

    -- -2292 是违反一致性约束的错误代码
    pragma exception_init(deptno_remaining, -2292);
begin
    delete from dept where deptno = v_depno;
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；若该员工不存在则抛出用户自定义异常：no_result

```
declare
    v_empid employees.employee_id%type := &v_empid;
    no_result exception;
begin
    update employees
    set salary = salary + 100
    where employee_id = v_empid;

    if sql%notfound then
        raise no_result;
    end if;
exception
    when no_result then
        dbms_output.put_line('数据更新失败');
    when others then
        dbms_output.put_line('出现其他异常');
end;
```

§5.2 在 PL/SQL 中使用 SQLCODE, SQLERRM

SQLCODE 返回错误代码数字

SQLERRM 返回错误信息。

如: SQLCODE=-100 → SQLERRM='no_data_found'
SQLCODE=0 → SQLERRM='normal, successful completion'

例 5. 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 6. 查询 ORACLE 错误代码;

```
BEGIN
    INSERT INTO emp(empno, ename, hiredate, deptno)
    VALUES(2222, 'Jerry', SYSDATE, 20);
    DBMS_OUTPUT.PUT_LINE('插入数据记录成功!');
    INSERT INTO emp(empno, ename, hiredate, deptno)
```

```
VALUES(2222, 'Jerry', 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;
```

第六章 存储函数和过程

§6.1 引言

1. **ORACLE** 提供可以把 **PL/SQL** 程序存储在数据库中，并可以在任何地方来运行它。这样就叫**存储过程或函数**。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。**

§6.2 创建函数

1. 建立内嵌函数

语法如下：

```
CREATE [OR REPLACE] FUNCTION function_name  
    [ (argument [ { IN | IN OUT } ] Type,  
      argument [ { IN | OUT | IN OUT } ] Type ]  
    [ AUTHID DEFINER | CURRENT_USER ]  
RETURN return_type
```

{ **IS** | **AS** }

<类型.变量的说明>

BEGIN

FUNCTION_body

EXCEPTION

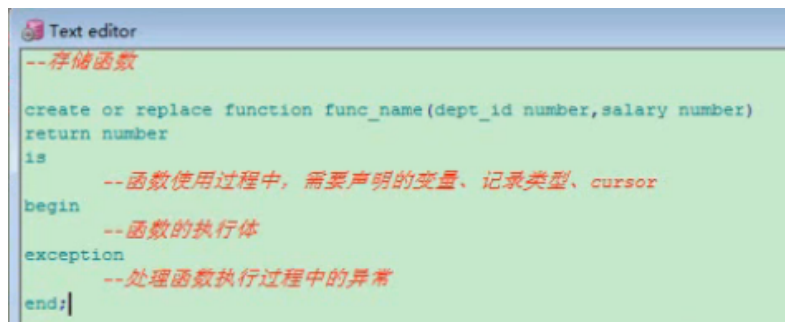
其它语句

END;

说明：

- 1) **OR REPLACE** 为可选。有了它，可以或者创建一个新函数或者替换相同名字的函数，而不会出现冲突
- 2) 函数名后面是一个可选的参数列表，其中包含 **IN**, **OUT** 或 **IN OUT** 标记。参数之间用逗号隔开。**IN** 参数标记表示传递给函数的值在该函数执行中不改变；**OUT** 标记表示一个值在函数中进行计算并通过该参数传递给调用语句；**IN OUT** 标记表示传递给函数的值可以变化并传递给调用语句。若省略标记，则参数隐含为 **IN**。
- 3) 因为函数需要返回一个值，所以 **RETURN** 包含返回结果的数据类型。

例1. 不带参数的函数



```
Text editor  
--存储函数  
  
create or replace function func_name(dept_id number,salary number)  
return number  
is  
    --函数使用过程中，需要声明的变量、记录类型、cursor  
begin  
    --函数的执行体  
exception  
    --处理函数执行过程中的异常  
end;
```



```
create or replace function test_fun
return date
is
v_date date;
begin
select sysdate into v_date
from dual;

dbms_output.put_line('我是函数哈^^');

return v_date;
end;
```

执行该函数

```
declare
v_date date;
begin
v_date := test_fun();
dbms_output.put_line(v_date);
end;
```

对于函数中有输出语句的，用select执行与用plsql块执行输出的顺序是不一样的。用select会先得到返回值，再输出；用plsql块会先输出，再得到返回值。

```
--函数的 helloworld: 返回一个 "helloworld" 的字符串
create or replace function hello_world1(v_logo varchar2)
return varchar2
is
begin
dbms_output.put_line('人家是函数的啦，么么');
return 'helloworld '||v_logo;
end;
```

```
begin
dbms_output.put_line(hello_world1('atguigu'));
end;
```

```
人家是函数的啦，么么
helloworld atguigu
```

```
SQL> select hello_world1('atguigu')from dual;

HELLO_WORLD1('ATGUIGU')
-----
helloworld atguigu

人家是函数的啦，么么
```

例2. 获取某部门的工资总和：

```
create or replace function get_salary(
dep_id employees.department_id%type,
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 = dep_id;

return v_sum;
exception
when no_data_found then
dbms_output.put_line('您需要数据不存在');
when others then
dbms_output.put_line(sqlcode || ' : ' || sqlerrm);
end;
```

2. 内嵌函数的调用

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称为**位置表示法**，格式为：

argument_value1[,argument_value2 ...]

例 3：计算某部门的工资总和：

```
declare
    v_num number;
    v_sum number;
begin
    v_sum := get_salary(80, v_num);
    dbms_output.put_line('80 号部门的工资总和: '
                        || v_sum || ', 人数: '
                        || v_num);
end;
```

第二种参数传递格式称为**名称表示法**，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 4：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, dep_id => 80);
    dbms_output.put_line('80 号部门的工资总和: '
                        || v_sum || ', 人数: '
                        || v_num);
end;
```

第三种参数传递格式称为**混合表示法**：

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 5：

```
declare
    v_num number;
    v_sum number;
begin
    v_sum := get_salary(80, emp_count => v_num);
    dbms_output.put_line('80 号部门的工资总和: '
                        || v_sum || ', 人数: '
                        || v_num);
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 6:

```
create or replace function get_salary(
    dep_id employees.department_id%type default 50,
    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 = dep_id;

    return v_sum;
exception
    when no_data_found then
        dbms_output.put_line('您需要的数据不存在');
    when others then
        dbms_output.put_line(sqlcode || ': ' || sqlerrm);
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
    v_num number;
    v_sum number;
begin
    v_sum := get_salary(emp_count => v_num);
    dbms_output.put_line('50 号部门的工资总和: '
        || v_sum || ', 人数: '
        || v_num);
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

[(argument [{ **IN** | IN OUT }] Type,
argument [{ **IN** | **OUT** | IN OUT }] Type)
[AUTHID DEFINER | CURRENT_USER]
{ **IS** | **AS** }
<类型.变量的说明>

BEGIN

<执行部分>

EXCEPTION

<可选的异常错误处理程序>

END;

例 7. 删除指定员工记录;

```
create or replace procedure del_emp(  
    v_empid in employees.employee_id%type)  
is  
  
    no_result exception;  
  
begin  
    delete from employees  
    where employee_id = v_empid;  
  
    if sql%notfound then  
        raise no_result;  
    end if;  
  
    dbms_output.put_line('编号为: ' || v_empid || '的员工已被除名');  
exception  
    when no_result then  
        dbms_output.put_line('您要删除的数据不存在');  
    when others then  
        dbms_output.put_line(sqlcode || '--' || sqlerrm);  
end;
```

例 8. 插入员工记录;

```
create or replace procedure insert_emp(  
    v_empno emp.empno%type,  
    v_name emp.ename%type,  
    v_depno emp.deptno%type)  
  
    is  
  
    empno_remaining exception;  
  
    pragma exception_init(empno_remaining, -1);  
begin  
    insert into emp(empno, ename, deptno)  
        values(v_empno, v_name, v_depno);  
    dbms_output.put_line('插入数据成功!');  
exception  
    when empno_remaining then  
        dbms_output.put_line('违反完整性约束!');  
    when others then  
        dbms_output.put_line(sqlcode || '--' || sqlerrm);  
end;
```

§6.3.2 调用存储过程

ORACLE 使用 **EXECUTE** 语句来实现对存储过程的调用：
EXEC[UTE] Procedure_name(parameter1, parameter2...);

例 9：查询指定员工记录；

```
create or replace procedure query_emp(  
    v_empid employees.employee_id%type,  
    v_name out employees.last_name%type,  
    v_sal out employees.salary%type)  
    is  
  
begin  
    select last_name, salary into v_name, v_sal  
    from employees  
    where employee_id = v_empid;  
  
    dbms_output.put_line('员工号为: ' || v_empid || '的员工已经找到');  
exception  
    when no_data_found then  
        dbms_output.put_line('你要查询的数据不存在');  
    when others then  
        dbms_output.put_line(sqlcode || '--' || sqlerrm);  
end;
```

调用方法:

```
declare
    v1 employees.last_name%type;
    v2 employees.salary%type;
begin
    query_emp(200, v1, v2);
    dbms_output.put_line('姓名: ' || v1 || ', 工资: ' || v2);

    query_emp(201, v1, v2);
    dbms_output.put_line('姓名: ' || v1 || ', 工资: ' || v2);
end;
```

例 10. 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(
    v_deptid employees.department_id%type default 10,
    v_salsum out employees.salary%type,
    v_empcount out number)
is

begin
    select sum(salary), count(*) into v_salsum, v_empcount
    from employees
    where department_id = v_deptid;

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;
```

调用方法:

```
declare
    v_num number;
    v_sum number;

begin
    proc_demo(v_salsum => v_sum, v_empcount => v_num);
    dbms_output.put_line('10号部门的工资总额为: ' || v_sum
        || ', 人数为: ' || v_num);
end;
```

§6.3.3 AUTHID

在创建存储过程时，可使用 AUTHID CURRENT_USER 或 AUTHID DEFINER 选项，以表明在执行该过程时 Oracle 使用的权限。

- 1) 如果使用 AUTHID CURRENT_USER 选项创建一个过程，则 **Oracle** 用调用该过程的用户权限执行该过程。为了成功执行该过程，调用者必须具有访问该存储过程中引用的所有数据库对象

所必须的权限

- 2) 如果用默认的 AUTHID DEFINER 选项创建过程, 则 **Oracle 使用过程所有者的特权执行该过程**. 为了成功执行该过程, **过程的所有者必须具有访问该存储过程中引用的所有数据库对象所必须的权限**. 想要简化应用程序用户的特权管理, 在创建存储过程时, 一般选择 AUTHID DEFINER 选项 — 这样就不必授权给需要调用的此过程的所有用户了.

§6.3.4 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:

§6.3.4.1 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,, 需将源码存为文本格式。

§6.3.4.2 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
```

§6.3.4.3 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。

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

§6.3.4.5 与过程相关数据字典

USER_SOURCE, ALL_SOURCE, DBA_SOURCE, USER_ERRORS

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**。

```
DESCRIBE Procedure_name;
```

§6.3.5 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;
```


第七章 包的创建和应用

§7.1 引言

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，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所访问。

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（通过使用私有变量），而且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：

包定义 (PACKAGE)：包定义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。

包定义和包主体分开编译，并作为两部分分开的对象存放在数据库字典中，详见数据字典 user_source, all_source, dba_source。

§7.2 包的定义

包定义的语法如下：

创建包定义：

```
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]...]
  [公有子程序声明[公有子程序声明]...]
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。

创建包主体：

```
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]...]

[私有子程序声明和定义[私有子程序声明和定义]...]

[公有游标定义[公有游标定义]...]

[公有子程序定义[公有子程序定义]...]

BEGIN

PL/SQL 语句

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_pack, 该包中包含一个记录变量 DeptRec、两个函数和一个过程。

```
CREATE OR REPLACE PACKAGE demo_pack
IS
  DeptRec dept%ROWTYPE;
  FUNCTION add_dept(
    dept_no NUMBER, dept_name VARCHAR2, location VARCHAR2)
    RETURN NUMBER;
  FUNCTION remove_dept(dept_no NUMBER)
    RETURN NUMBER;
  PROCEDURE query_dept(dept_no IN NUMBER);
END demo_pack;
```

包主体的创建方法，它实现上面所声明的包定义

```
CREATE OR REPLACE PACKAGE BODY demo_pack
IS
  FUNCTION add_dept
    (dept_no NUMBER, dept_name VARCHAR2, location VARCHAR2)
    RETURN NUMBER
```

```
IS
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 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 remove_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 remove_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('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');
    WHEN OTHERS THEN
```

```
DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);  
END query_dept;
```

```
BEGIN  
    Null;  
END demo_pack;
```

对包内共有元素的调用格式为：包名.元素名称

调用 demo_pack 包内函数对 dept 表进行插入、查询和修改操作，并通过 demo_pack 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_pack.add_dept(90,'Administration', 'Beijing');  
    IF var =-1 THEN  
        DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);  
    ELSIF var =0 THEN  
        DBMS_OUTPUT.PUT_LINE('该部门记录已经存在！');  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('添加记录成功！');  
        Demo_pack.query_dept(90);  
        DBMS_OUTPUT.PUT_LINE(demo_pack.DeptRec.deptno||'---'||  
            demo_pack.DeptRec.dname||'---'||demo_pack.DeptRec.loc);  
        var := demo_pack.remove_dept(90);  
        IF var =-1 THEN  
            DBMS_OUTPUT.PUT_LINE(SQLCODE||'----'||SQLERRM);  
        ELSIF var=0 THEN  
            DBMS_OUTPUT.PUT_LINE('该部门记录不存在！');  
        ELSE  
            DBMS_OUTPUT.PUT_LINE('删除记录成功！');  
        END IF;  
    END IF;  
END;
```

例 2：创建包 emp_package

```
CREATE OR REPLACE PACKAGE emp_package  
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_package;
```

```
CREATE OR REPLACE PACKAGE BODY emp_package
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_package;

DECLARE
  E_table emp_package.emp_table_type;
BEGIN
  Emp_package.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 emp_mgmt:

```
CREATE SEQUENCE empseq
  START WITH 1000
  INCREMENT BY 1
  ORDER NOCYCLE;

CREATE SEQUENCE deptseq
  START WITH 50
  INCREMENT BY 10
  ORDER NOCYCLE;

CREATE OR REPLACE PACKAGE emp_mgmt
AS
  FUNCTION hire(ename VARCHAR2, job VARCHAR2, mgr NUMBER, sal NUMBER,
    comm NUMBER, deptno NUMBER)
    RETURN NUMBER;
  FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
    RETURN NUMBER;
  PROCEDURE remove_emp(empno NUMBER);
  PROCEDURE remove_dept(deptno NUMBER);
```

```
PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER);
PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER);
END emp_mgmt;

CREATE OR REPLACE PACKAGE BODY emp_mgmt
AS
    tot_emps NUMBER;
    tot_depts NUMBER;
    no_sal EXCEPTION;
    no_comm EXCEPTION;
FUNCTION hire(ename VARCHAR2, job VARCHAR2, mgr NUMBER,
    sal NUMBER, comm NUMBER, deptno NUMBER)
    RETURN NUMBER IS
    new_empno NUMBER(4);
BEGIN
    SELECT empseq.NEXTVAL INTO new_empno FROM dual;
    INSERT INTO emp
        VALUES (new_empno, ename, job, mgr, sysdate, sal, comm, deptno);
    tot_emps:=tot_emps+1;
    RETURN(new_empno);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END hire;

FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
    RETURN NUMBER IS
    new_deptno NUMBER(4);
BEGIN
    SELECT deptseq.NEXTVAL INTO new_deptno FROM dual;
    INSERT INTO dept VALUES (new_deptno, dname, loc);
    Tot_depts:=tot_depts;
    RETURN(new_deptno);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END create_dept;

PROCEDURE remove_emp(empno NUMBER) IS
    No_result EXCEPTION;
BEGIN
    DELETE FROM emp WHERE emp.empno=remove_emp.empno;
    IF SQL%NOTFOUND THEN
        RAISE no_result;
```

```
END IF;
tot_emps:=tot_emps-1;
EXCEPTION
WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('你需要的数据不存在!');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END remove_emp;

PROCEDURE remove_dept(deptno NUMBER) IS
    No_result EXCEPTION;
    e_deptno_remaining EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_deptno_remaining, -2292);
    /* -2292 是违反一致性约束的错误代码 */
BEGIN
    DELETE FROM dept WHERE dept.deptno=remove_dept.deptno;
    IF SQL%NOTFOUND THEN
        RAISE no_result;
    END IF;
    Tot_depts:=tot_depts-1;
EXCEPTION
WHEN no_result THEN
    DBMS_OUTPUT.PUT_LINE('你需要的数据不存在!');
WHEN e_deptno_remaining THEN
    DBMS_OUTPUT.PUT_LINE('违反数据完整性约束!');
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END remove_dept;

PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER) IS
    curr_sal NUMBER(7, 2);
BEGIN
    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
            WHERE emp.empno=increase_sal.empno;
    END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('你需要的数据不存在!');
WHEN no_sal THEN
    DBMS_OUTPUT.PUT_LINE('此员工的工资不存在!');
```

```
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END increase_sal;

PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER) IS
    curr_comm NUMBER(7,2);
BEGIN
    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('你需要的数据不存在!');
    WHEN no_comm THEN
        DBMS_OUTPUT.PUT_LINE('此员工的奖金不存在!');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('发生其它错误!');
END increase_comm;

END EMP_MGMT;
```

例 4：利用游标变量创建包 Curvarpack。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。

```
CREATE OR REPLACE PACKAGE CurVarPack 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;

CREATE OR REPLACE PACKAGE BODY CurVarPack 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 CurVarPack;
```

--定义一个过程

```
CREATE OR REPLACE PROCEDURE OpenCurType(
    Cv IN OUT CurVarPack.CurType,
    Tab CHAR)
AS
BEGIN
    --由于 CurVarPack.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 tab = 'D' THEN
        OPEN cv FOR SELECT * FROM dept;
    ELSE
        OPEN cv FOR SELECT * FROM emp;
    END IF;
END OpenCurType;
```

--定义一个应用

```
DECLARE
    DeptRec Dept%ROWTYPE;
    EmpRec Emp%ROWTYPE;
    Cv1 Curvarpack.deptcurtype;
    Cv2 Curvarpack.curtype;
BEGIN
    DBMS_OUTPUT.PUT_LINE('游标变量强类型定义应用');
    Curvarpack.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');
CurVarPack.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');
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');
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;
```

§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。所谓重载是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:

```
CREATE OR REPLACE PACKAGE demo_pack1
IS
    DeptRec dept%ROWTYPE;
    V_sqlcode NUMBER;
    V_sqlerr VARCHAR2(2048);
    FUNCTION query_dept(dept_no IN NUMBER)
        RETURN INTEGER;
    FUNCTION query_dept(dept_no IN VARCHAR2)
        RETURN INTEGER;
END demo_pack1;

CREATE OR REPLACE PACKAGE BODY demo_pack1
```

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

FUNCTION check_dept(dept_no VARCHAR2)
    RETURN INTEGER
IS
    Flag INTEGER;
BEGIN
    SELECT COUNT(*) INTO flag FROM dept WHERE deptno=dept_no;
    IF flag > 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_pack1;
```

§7.6 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 PACKAGE demo_pack;
DROP PACKAGE demo_pack1;
DROP PACKAGE emp_mgmt;
DROP PACKAGE emp_package;
```

§7.7 包的管理

DBA_SOURCE, USER_SOURCE, USER_ERRORS, DBA-OBJECTS

第八章 触发器

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，它与存储过程不同的是，存储过程通过其它程序来启动运行或直接启动运行，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。

§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。

§8.1.3 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**：即在何种情况下触发 TRIGGER；例如：**INSERT, UPDATE, DELETE。**
- **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。
- **触发器本身**：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。

§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 ...]]}
ON [schema.] table_name
[FOR EACH ROW ]
[WHEN condition]
trigger_body;
```

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 语句操作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。

WHEN 子句说明触发约束条件。Condition 为一个逻辑表达式时,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。

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, SAVEPOINT 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;

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 | 有效 | 有效 |
| NEW | 有效 | 有效 | 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 del_emp_trigger
before delete on emp for each row
begin
    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.hiredate);
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 ...]]}  
ON [schema.] view_name  
[FOR EACH ROW ]  
[WHEN condition]  
trigger_body;
```

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。

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 语句操做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。

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:

```
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;

§8.2.3 创建系统事件触发器

ORACLE 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 [schema.] trigger_name
{BEFORE|AFTER}
{ddl_event_list | database_event_list}
ON { DATABASE | [schema.] SCHEMA }
[WHEN_clause]
trigger_body;
```

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 |
|-------------------|-------|------------|
| 启动 STARTUP | 之后 | 实例启动时激活 |
| 关闭 SHUTDOWN | 之前 | 实例正常关闭时激活 |
| 服务器错误 SERVERERROR | 之后 | 只要有错误就激活 |
| 登录 LOGON | 之后 | 成功登录后激活 |
| 注销 LOGOFF | 之前 | 开始注销时激活 |
| 创建 CREATE | 之前，之后 | 在创建之前或之后激活 |
| 撤消 DROP | 之前，之后 | 在撤消之前或之后激活 |
| 变更 ALTER | 之前，之后 | 在变更之前或之后激活 |

§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 |
|----------------------------|---------------|---|
| Sysevent | VARCHAR2 (20) | 激活触发器的事件名称 |
| Instance_num | NUMBER | 数据库实例名 |
| 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 |

§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。**

● 触发器的状态

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;