

Module07-02

数据库开发：Oracle PL/SQL

- SQL 语句
- Oracle PL/SQL
- MySQL Procedure
- C++ OTL
- 数据建模工具

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ 关于 PL/SQL 语言

- ◆ PL/SQL 是一种程序语言，即过程化 SQL 语言（Procedural Language/SQL. PL/SQL 是 Oracle 数据库对 SQL 语句的扩展。在 SQL 语句的使用上增加了过程化编程语言的特点，所以 PL/SQL 就是把数据操作和查询语句组织在 PL/SQL 代码的过程性单元中，通过逻辑判断、循环等操作实现复杂的功能或者计算的程序语言。

■ 为什么使用 PL/SQL 语言

- ◆ 使用 PL/SQL 可以编写具有很多高级功能的程序，虽然通过多个 SQL 语句可能也能实现同样的功能，但是相比而言，PL/SQL 具有一些更为明显的优点：
 - 能够使一组 SQL 语句的功能更具模块化程序特点；
 - 采用了过程性语言控制程序的结构；
 - 可以对程序中的错误进行自动处理，使程序能够在遇到错误的时候不会被中断；
 - 具有较好的可移植性，可以移植到另一个 Oracle 数据库中；
 - 集成在数据库中，调用更快；
 - 减少了网络的交互，有助于提高程序性能。

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ PL/SQL 语言元素

```
DECLARE
    /* 声明 */
    v_name      EMPLOYEE.name%TYPE;
    v_result    INTEGER;
BEGIN
    /* 执行块开始 */
    v_result := 1;
    SELECT name INTO v_name
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    /* 其它语句 */
EXCEPTION
    /* 异常处理 */
    WHERE NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-01043, 'No data');
END;
```

变量名

变量类型

1 声明

注释

赋值语句

2 执行块

数据

3 异常处理

4 块

- PL/SQL 程序的组成
 - ◆ 块 (Block)
 - ◆ 变量名 (Variable name)
 - ◆ 数据类型 (Data type)
 - ◆ 变量声明 (Declare)
 - ◆ 赋值语句 (Assignment)
 - ◆ 表达式 (Expression)
 - ◆ 注释 (Comment)

■ 块的构成

- ◆ 标准的语句块由 3 部分组成
 - 声明部分 (Declare)：声明块中用到的变量、游标和类型，或是声明局部的过程或函数，而这些过程和函数值在该语句块中有效
 - 执行部分：语句块要完成的任务，由 SQL 语句和过程性语句组成
 - 异常处理部分：处理程序执行过程中遇到的错误，在 PL/SQL 程序中，警告和错误信息被称之为异常
- ◆ 上述的 3 个部分中，只有执行部分是必须的，声明和异常处理是可选的

■ 块的类型

◆ 块可分为以下 3 类：

- 无名块：动态构造，且只能被执行一次
- 命名块：在无名块前加上名称，当使用嵌套块时，为了区分多级嵌套层次关系，可以使用命名块加以区分。
- 子程序：包括存储在数据库中的存储过程、函数、包，这些块一旦定义后，便可随时调用
- 触发器：特殊的子程序，当触发它的事件发生时调用触发器，触发的事件很多，比如对表中的数据执行 delete、update、insert 等等操作

■ 无名块示例

SET SERVEROUTPUT ON; -- 注意打开该选项，否则下面的输出语句无效果

```
DECLARE
    v_name      EMPLOYEE.empl_name%TYPE;
BEGIN
    SELECT empl_name INTO v_name
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    DBMS_OUTPUT.PUT_LINE(v_name);
END;
/
```

■ 命名块示例

```
SET SERVEROUTPUT ON; -- 注意打开该选项, 否则下面的输出语句无效果

<<blk_outer>>
DECLARE
    v_dept_id DEPARTMENT.depart_id%TYPE;
    v_dname DEPARTMENT.depart_name%TYPE;
BEGIN
    <<blk_inner>>
    BEGIN
        SELECT dep_depart_id INTO v_dept_id FROM EMPLOYEE
        WHERE lower(empl_name)=lower('&name'); -- 从标准输入获取
name
    END blk_inner; -- 此处名称可有可无

    SELECT depart_name INTO v_dname FROM DEPARTMENT
    WHERE depart_id=v_dept_id;
    DBMS_OUTPUT.PUT_LINE('Department name: '||v_dname);
END blk_outer; -- 此处名称可有可无
/
```

■ 标识符

- ◆ 所谓标识符是指用于命名：变量、子程序、游标等
- ◆ 一个合法的标识符必须符合以下要求：
 - 标识符只能由：[a-zA-Z0-9\$_#_] 字符组成
 - 第一个字符必须是字母
 - 长度不超过 30 个字符
- ◆ 注意：PL/SQL 的标识符不区分大小写！如果希望区分大小写，则需将标识符用” ” 包括起来：
 - name 与 NAME、Name、NAme 等是同一个标识符
 - 而 “Name” 则是与上述标识符不同的名称，大小写区分

■ 变量名

- ◆ 一个合法的变量名必须是一个合法的标识符
- ◆ 变量名不可与 PL/SQL 关键字相同，如 BEGIN、END 等

- 数据类型简介
 - ◆ 标量类型（简单类型）
 - ◆ 复合类型
 - ◆ LOB 类型（大对象类型）

■ 标量类型

◆ 常用的标量类型：

- 数值类型 (Numeric) :
 - number(p,s)、integer 等
- 字符类型 (Character)
 - varchar2(n)、char(n) 等
- 日期类型 (Date Time)
 - date、timestamp 等
- 布尔类型 (Boolean)
 - 如文字量：true、false

■ 复合类型 - Record

- ◆ Record 的定义：在使用记录类型之前必须先正确定义。
- Record 定义语法：

```
TYPE record_type IS RECORD (  
    field1  type1 [NOT NULL] [:=expr1],  
    field2  type2 [NOT NULL] [:=expr2],  
    ...  
    fieldn  typen [NOT NULL] [:=exprn]  
);
```

-- 上述语句中的赋值操作符 := 可以由关键字 default 代替

■ 复合类型 - Record

◆ Record 示例

```
DECLARE
    TYPE Rec_Empl_Extra IS RECORD (
        has_email      NUMBER(1) NOT NULL := 1,
        bonus_level    NUMBER(1) DEFAULT 3,
        dept_name      DEPARTMENT.depart_name%TYPE
    );

    -- 使用 Rec_Empl_Extra 定义变量
    extra1 Rec_Empl_Extra;
    extra2 Rec_Empl_Extra;

BEGIN
    /* 其它代码 */
    extra1.has_email := 0;    -- 为字段赋值
    extra2 := extra1;        -- 将一个对象为另一个对象赋值

    /* 其它代码 */

END;
/
```

■ LOB 类型

- ◆ LOB 为大对象类型， PL/SQL 支持以下大对象类型：
 - BFILE：文件定位器，指向只读的、操作系统的文件系统中的二进制大文件，该对象的存储不再数据库中
 - BLOB：LOB 定位器，指向数据库内部的二进制大对象
 - CLOB：LOB 定位器，指向数据库内部的字符大对象
 - NCLOB：LOB 定位器，指向数据库内部的 Unicode 字符大对象

■ 变量声明的语法

- ◆ 变量在块的声明部分声明，语法如下：

```
variable_name [CONSTANT] type [NOT NULL] [:= default_value];
```

- ◆ 上面各个部分：
 - *variable_name* : 变量名，必须是一个合法的标识符
 - *type* : 变量的类型
 - **CONSTANT** : 指示该变量是一个常量，在声明时必须赋初值
 - **NOT NULL** : 指示该变量不可为 NULL，在声明时必须赋初值
 - *default_value* : 变量的初始值

■ 变量的作用域

```
DECLARE
    v_age    NUMBER(3);
BEGIN
    DECLARE
        v_name VARCHAR2(16);
    BEGIN
        /* 其它代码 */
    END;
    /* 其它代码 */
END;
/
```

v_name 作用域

v_age 作用域

■ 局部变量可以屏蔽全局变量

```
<<outer_scope>>
DECLARE
    v_age    NUMBER(3);
BEGIN
    <<inner_scope>>
    DECLARE
        v_name  VARCHAR2(16);
        v_age    NUMBER(3);  -- 屏蔽全局的 v_age
    BEGIN
        v_age := 18; -- inner_scope.v_age

        /* 要在该作用域访问全局的 v_age : */
        outer_scope.v_age := 22;
    END;

    v_age := v_age + 5; -- outer_scope.v_age
    /* 其它代码 */
END;
/
```

- 赋值语句的语法

```
variable_name := expression;
```

■ 表达式分类：

◆ 算术表达式

- 如 `v_salary + v_salary * 1.2`

◆ 字符表达式

- 如 `v_name := 'Doug ' || 'Paul';`

◆ 逻辑表达式

- 如 `v_salary <= 1000 AND v_name like '_h%'`

■ PL/SQL 注释：

◆ 单行注释

- 单行注释以： -- 开头，如
- -- 这是一个单行注释

◆ 多行注释

- 多行注释如：
- /* 这是多行注释的第一行，
这是多行注释的第二行。 */

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

- 关于流程控制语句：
 - ◆ 同其它语言类似， PL/SQL 也支持流程控制，常见的流程控制语句分为以下 2 类：
 - 分支语句
 - 循环语句

- 分支语句：
 - ◆ 常见的分支语句分为：
 - IF .. THEN
 - IF .. THEN .. ELSE
 - IF .. THEN .. ELSIF
 - CASE WHEN

■ IF .. THEN 语句

◆ 语法:

```
IF condition_expr THEN  
    statements ...  
END IF;
```

■ IF .. THEN 语句

◆ 示例:

```
SET SERVEROUTPUT ON;

DECLARE
    v_email EMPLOYEE.email%TYPE;
BEGIN
    SELECT email INTO v_email
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    IF v_email IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE(v_email);
    END IF;
END;
/
```

- IF .. THEN .. ELSE 语句

- ◆ 语法:

```
IF condition_expr THEN  
    statements ...  
ELSE  
    statements ...  
END IF;
```

■ IF .. THEN .. ELSE 语句

◆ 示例:

```
SET SERVEROUTPUT ON;

DECLARE
    v_email EMPLOYEE.email%TYPE;
BEGIN
    SELECT email INTO v_email
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    IF v_email IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE(v_email);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Email not set.');
```


■ IF .. THEN .. ELSIF 语句

◆ 语法:

```
IF condition_expr1 THEN
    statements ...
ELSIF condition_expr2 THEN
    statements ...
ELSE
    statements ...
END IF;
```

■ IF .. THEN .. ELSIF 语句

◆ 示例:

```
DECLARE
    v_salary EMPLOYEE.salary%TYPE;
    v_sal_level NUMBER(1);
BEGIN
    SELECT salary INTO v_salary
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    IF v_salary < 3000 THEN
        v_sal_level := 1;
    ELSIF v_salary >= 3000 AND v_salary < 5000 THEN
        v_sal_level := 2;
    ELSE
        v_sal_level := 3;
    END IF;
END;
/
```

■ CASE WHEN 语句

- ◆ CASE WHEN 语句有 2 种： Simple Case 和 Searched Case
- ◆ Simple Case 语法：

```
-- Simple case
CASE switch-expr
  WHEN switch-value1 THEN
    statement ...
  [ WHEN switch-valueN THEN
    statement ... ]
  ELSE
    statement ...
END CASE;
```

■ CASE WHEN 语句

◆ Searched Case 语法:

```
-- Searched case
CASE
    WHEN switch-expr1 THEN
        statement ...
    [ WHEN switch-exprN THEN
        statement ... ]
    ELSE
        statement ...
END CASE;
```

■ CASE WHEN 语句

◆ Simple CASE 示例:

```
DECLARE
    v_location  DEPARTMENT.location%TYPE;
BEGIN
    SELECT location INTO v_location
    FROM DEPARTMENT
    WHERE depart_id = 'RND';

    CASE v_location
        WHEN 'Beijing' THEN
            DBMS_OUTPUT.PUT_LINE('Northern China. ');
        WHEN 'Shanghai' THEN
            DBMS_OUTPUT.PUT_LINE('Eastern China. ');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Unknown. ');
    END CASE;
END;
/
```

■ CASE WHEN 语句

◆ Searched CASE 示例:

```
DECLARE
    v_salary EMPLOYEE.salary%TYPE;
    v_sal_level NUMBER(1);
BEGIN
    SELECT salary INTO v_salary
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    CASE
        WHEN v_salary < 3000 THEN
            v_sal_level := 1;
        WHEN v_salary >= 3000 AND v_salary < 5000 THEN
            v_sal_level := 2;
        ELSE
            v_sal_level := 3;
    END CASE;
END;
/
```

■ 循环语句：

- ◆ 常见的循环语句分为：
 - LOOP .. EXIT
 - WHILE .. LOOP
 - FOR .. LOOP

■ LOOP 语句

◆ 语法:

```
[ <<loop-label>> ]  
LOOP  
    statements ...  
    IF condition THEN  
        EXIT [loop-label];  
    END IF;  
END LOOP;
```

} 或其它分支语句

■ LOOP 语句

◆ 示例:

```
DECLARE
    n    NUMBER(2) := 0;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Round #' || TO_CHAR(n));
        EXIT WHEN n = 10;
        n := n + 1;
    END LOOP;
END;
/
```

■ WHILE .. LOOP 语句

◆ 语法:

```
WHILE condition LOOP  
    statements ...  
END LOOP;
```

■ WHILE .. LOOP 语句

◆ 示例:

```
DECLARE
    n    NUMBER(2) := 0;
BEGIN
    WHILE n <= 10 LOOP
        DBMS_OUTPUT.PUT_LINE('Round #' || TO_CHAR(n));
        n := n + 1;
    END LOOP;
END;
/
```

■ FOR .. LOOP 语句

◆ 语法:

```
FOR loop-variable IN begin-value .. end-value LOOP  
    statements ...  
END LOOP;
```

◆ 示例:

```
BEGIN  
    FOR n IN 0 .. 10 LOOP  
        DBMS_OUTPUT.PUT_LINE( 'Round #' || TO_CHAR(n) );  
    END LOOP;  
END;  
/
```

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ 什么是异常

- ◆ PL/SQL 中出现的警告和错误统称为异常，针对异常进行处理则称为异常处理
- ◆ 异常有预定义异常、自定义异常。PL/SQL 的预定义异常都在 STANDARD 包中定义，可以直接使用，不需在声明块中定义和声明，而用户自定义的异常需在声明块中先定义，才能使用。

■ 常见的预定义异常

Oracle 错误	异常	产生异常的时机
ORA-0001	DUP_VAL_ON_INDEX	违反唯一性约束
ORA-0051	TIMEOUT_ON_RESOURCE	等待资源超时
ORA-0061	TRANSACTION_BACKED_OUT	由于死锁导致事务回滚。
ORA-1001	INVALID_CURSOR	执行了非法的游标操作。如试图关闭一个已经关闭的游标
ORA-1012	NOT_LOGGED_ON	没有连接到 Oracle
ORA-1017	LOGIN_DENIED	登录被拒绝
ORA-1403	NO_DATA_FOUND	SELECT...INTO 语句没有返回行数为 0，或试图引用一个没有被赋值的 PL/SQL 表元素
ORA-1422	TOO_MANY_ROWS	SELECT...INTO 语句返回多行
ORA-1476	ZERO_DIVIDE	除 0
ORA-1722	INVALID_NUMBER	将字符串转成数值型时出现错误
ORA-6500	STORAGE_ERROR	内存溢出
ORA-6501	PROGRAM_ERROR	内部错误

■ 常见的预定义异常（续）

Oracle 错误	异常	产生异常的时机
ORA-6502	VALUE_ERROR	类型转换出现的错误
ORA-6504	ROWTYPE_MISMATCH	一个主游标变量和 PL/SQL 游标变量类型不匹配
ORA-6511	CURSOR_ALREADY_OPEN	试图打开一个已经打开的游标
ORA-6530	ACCESS_INTO_NULL	试图给一个 NULL 对象赋值
ORA-6531	COLLECTION_IS_NULL	试图对一个 NULL 值的 PL/SQL 表或变长数组执行除 EXISTS 以外的操作
ORA-6532	SUBSCRIPT_OUTSIDE_LIMIT	引用的嵌套表或变长数组索引超出了其声明的范围
ORA-6533	SUBSCRIPT_BEYOND_COUNT	引用的嵌套表或变长数组索引大于嵌套表中的元素个数

■ 异常的声明

- ◆ 用户自定义的异常必须在声明块中先定义，而后才能使用
- ◆ 异常不是变量，不能象变量一样进行赋值等操作
- ◆ 异常的作用域和可视范围同变量的规则

异常声明示例

```
DECLARE
    e_invalid_salary EXCEPTION;  -- 声明一个自定义异常

    /* 其它代码 */
```

■ EXCEPTION_INIT

- ◆ 将自定义的异常与 Oracle 错误关联：
 - 首先必须先定义自定义异常
 - 然后通过 EXCEPTION_INIT 将该异常与 Oracle 错误关联
- ◆ 语法：

```
-- exception_name 须是已经定义的异常
```

```
PRAGMA EXCEPTION_INIT(exception_name, Oracle_Error_Number);
```

◆ 示例：

```
DECLARE
```

```
    e_too_large EXCEPTION;
```

```
-- ORA-01401: inserted value too large for column
```

```
PRAGMA EXCEPTION_INIT(e_too_large, -01401);
```

```
/* 其它代码 */
```

■ 异常产生方式

- ◆ 1，预定义异常的产生：当遇到预定义异常时，自动产生
- ◆ 2，自定义异常的产生：需由用户显式通过 RAISE 子句产生
- ◆ 3，自定义、但关联到 Oracle 错误的异常的产生：同预定义异常，自动产生

■ 异常产生方式示例：自定义异常的产生

```
DECLARE
    v_salary          INTEGER;
    e_invalid_salary  EXCEPTION;
BEGIN
    SELECT salary INTO v_salary
    FROM EMPLOYEE
    WHERE empl_id = 'ACC0001';

    IF v_salary < 5000 THEN
        RAISE e_invalid_salary;
    END IF;
EXCEPTION
    WHEN e_invalid_salary THEN
        UPDATE EMPLOYEE SET salary = 5000
        WHERE empl_id = 'ACC0001';
END;
/
```

■ 异常处理语法

```
EXCEPTION
    WHEN exception1 THEN
        statements ...
    WHEN exception2 THEN
        statements ...
    WHEN exceptionN THEN
        statements ...
    WHEN OTHERS THEN
        statements ...
END;
```

◆ 说明:

- 最后的 WHEN OTHERS THEN 检测有助于提高应用的健壮性。

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ 关于游标 (Cursor)

- ◆ Oracle 数据库执行的任意 SQL 语句均有一个包含该 SQL 语句的信息和返回值的私有 SQL 区域，而游标则是指向特定 SQL 语句的私有 SQL 区域的名字（指针）
- ◆ 游标分：
 - 静态游标：关联静态 SQL 语句，即编译器确定的 SQL 语句，静态游标只能用于 DML 语句
 - 动态游标：关联动态 SQL 语句，即运行期确定的 SQL 语句，动态游标用于 DDL 和 DCL 语句
- ◆ 根据游标的声明方式分：
 - 显式游标
 - 隐式游标

■ 显式游标的定义

◆ 语法:

```
CURSOR cursor_name [(parameter[,parameter] ...)]  
    [RETURN return-type]  
IS  
    select-statement;
```

-- parameter 的格式:

```
cursor-parameter-name [IN] data-type [{:= | DEFAULT} expr]
```

◆ 游标定义示例:

```
DECLARE  
  
    CURSOR cur_empl_info IS  
        SELECT empl_id, empl_name, salary  
        FROM EMPLOYEE  
        WHERE dep_depart_id = 'RND';  
  
    /* 其它代码 */
```


■ 显式游标的操作

- ◆ 显式游标相关的 3 个操作：
 - open : 打开游标
 - fetch : 从游标中取出结果
 - close : 关闭游标
- ◆ 操作语法：

```
-- 打开游标
OPEN cursor_name [(parameter[,parameter] ...)];

-- 从游标中提取内容
FETCH cursor_name INTO variable-list;
-- 或
FETCH cursor_name INTO record-object;

-- 关闭游标
CLOSE cursor_name;
```

■ 显式游标的属性

- ◆ %FOUND
 - 表示 FETCH 语句成功取出一条记录
- ◆ %NOTFOUND
 - 与 %FOUND 相反
- ◆ %ISOPEN
 - 查看游标是否被打开
- ◆ %ROWCOUNT
 - 指示总计 FETCH 多少行记录

■ 显式游标的操作示例

```
DECLARE
    CURSOR cur_empl_info IS      -- 定义游标
        SELECT empl_id, empl_name, salary
        FROM EMPLOYEE
        WHERE dep_depart_id = 'RND';

    TYPE rec_empl_info IS RECORD (    -- 定义记录类型
        v_empl_id      EMPLOYEE.empl_id%TYPE,
        v_empl_name     EMPLOYEE.empl_name%TYPE,
        v_salary        EMPLOYEE.salary%TYPE
    );

    v_info rec_empl_info;  -- 创建记录对象
```

■ 显式游标的操作示例（续）

```
BEGIN
    OPEN cur_empl_info;
    LOOP
        FETCH cur_empl_info INTO v_info;

        EXIT WHEN cur_empl_info%NOTFOUND;

        -- 输出: fetch 的行数、 empl_id、 empl_name、 salary
        DBMS_OUTPUT.PUT_LINE('Employee #' ||
                               TO_CHAR(cur_empl_info%ROWCOUNT) || ':' ||
                               v_info.v_empl_id || v_info.v_empl_name ||
                               TO_CHAR(v_info.v_salary));
    END LOOP;
    CLOSE cur_empl_info;
END;
/
```

■ 隐式游标

- ◆ 显式游标仅仅是用来控制返回多行的 SELECT 语句，而隐式游标则是一种处理所有 SQL 语句的私有 SQL 区域的指针，隐式游标也叫 SQL 游标
- ◆ 隐式游标用于处理 DELETE、UPDATE、INSERT 以及返回一行的 SELECT..INTO 语句

■ 隐式游标的属性

◆ %FOUND

- 当 INSERT、UPDATE、DELETE 语句影响的行数大于 0 时，SELECT .. INTO 语句返回一行时，返回 true，否则 false
- 当 SELECT .. INTO 语句返回大于一行时，抛出 TO_MANY_ROWS 异常，返回 0 时，抛出 NO_DATA_FOUND，二者均不会返回 true

◆ %NOTFOUND

- 与 %FOUND 相反

■ 隐式游标的属性（续）

◆ %ISOPEN

- 由于隐式游标于语句结束后自动关闭，所以该属性总是为 false

◆ %ROWCOUNT

- 指示 INSERT、UPDATE、DELETE 语句影响的行数，SELECT .. INTO 返回的行数。
- 当 SELECT .. INTO 语句返回大于一行时，抛出 TO_MANY_ROWS 异常，返回 0 时，抛出 NO_DATA_FOUND，二者均不会去判断该属性的值

■ 隐式游标示例 1

```
-- 该例程先尝试更改某条记录，如果该记录不存在，则添加一条新记录
BEGIN
    UPDATE DEPARTMENT SET location = 'Shenzhen'
    WHERE depart_id = 'PRD';

    IF SQL%NOTFOUND THEN -- 或使用下面的语句
-- IF SQL%ROWCOUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('No such department, add it!');
        INSERT INTO DEPARTMENT
        VALUES ('PRD', 'Public Relation', 'Shenzhen');
    END IF;
END;
/
```


■ 隐式游标示例 2

```
DECLARE
    v_name  DEPARTMENT.depart_name%TYPE;
BEGIN
    SELECT depart_name INTO v_name
    FROM DEPARTMENT
    WHERE depart_id = 'ANG';

    IF SQL%FOUND THEN
        DELETE FROM DEPARTMENT
        WHERE depart_id = 'ANG';
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No such department');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Too many department matched');
END;
/
```

■ 游标变量

- ◆ 游标变量是一种动态游标，其使用方式为，先定义一个游标类型，但不事先关联到某个 select 语句，而是在 open 操作中关联 select 语句
- ◆ open 之后的操作与静态显式游标无异

■ 游标变量声明

◆ 语法

```
TYPE cursor_variable IS REF CURSOR RETURN return-type;
```

◆ 注意:

- 声明过程中并没有关联到某个 select 语句

- 打开游标变量
 - ◆ 语法

```
OPEN cursor_variable FOR select-statement;
```

■ 游标变量操作示例

```
-- 注意游标变量定义的 RETURN 只接受 RECORD 类型的返回值
DECLARE
    v_dept  DEPARTMENT%ROWTYPE; -- 定义一个 Record 类型的变量
    TYPE cur_dept_type
        IS REF CURSOR RETURN v_dept%TYPE; -- 定义一个游标变量类型

    cur_dept cur_dept_type; -- 声明游标变量
BEGIN
    OPEN cur_dept FOR -- 打开游标
        SELECT * FROM DEPARTMENT WHERE depart_id = 'RND';

    LOOP
        FETCH cur_dept INTO v_dept;
        EXIT WHEN cur_dept%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(cur_dept%ROWCOUNT) ||
                               ' ' || v_dept.depart_name);
    END LOOP;
    CLOSE cur_dept; -- 关闭游标
```

■ 游标变量操作示例

```
OPEN cur_dept FOR -- 再次打开游标
    SELECT * FROM DEPARTMENT WHERE depart_id = 'ACC';

LOOP
    FETCH cur_dept INTO v_dept;
    EXIT WHEN cur_dept%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(cur_dept%ROWCOUNT) ||
                          ' ' || v_dept.depart_name);
END LOOP;
CLOSE cur_dept; -- 再次关闭游标
END;
/
```

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ 关于存储过程与函数

- ◆ PL/SQL 的存储过程和函数，实际上是命名的块 (Block)，不过这些过程和函数被存储在 Oracle 服务器中，并且事先已经编译完成，可以被反复调用
- ◆ 存储过程和函数是比较特别的命名块，结构上可分为 2 个部分：参数定义、体（常规语句块）定义

■ 创建存储过程语法

```
CREATE [OR REPLACE] PROCEDURE [schema.]procedure_name  
    [(argument [{IN | OUT | IN OUT}] data-type [,...])]  
    {IS | AS}  
    PL/SQL-Body;
```

■ 关于存储过程的参数

- ◆ 存储过程和函数的参数有 3 种形式：
 - IN：传入的参数，该参数是只读的，在过程体内不能对其赋值
 - OUT：输出的参数（类似 C++ 语言中的按引用传参），在过程体内只能被赋值，不能获取其值
 - IN OUT：既可以作为传入参数，也可以作为输出参数
 - 如果没有指定参数形式，默认为 IN
- ◆ 参数的默认值
 - 存储过程的参数可以有默认值。
- ◆ 参数的数据类型：
 - 所有参数的数据类型部分，不能带长度信息，如：
 - 错误的方式： `v_name VARCHAR2(12)`
 - 正确的方式： `v_name VARCHAR2`

■ 存储过程示例

```
CREATE OR REPLACE PROCEDURE test_proc (  
    arg1      IN OUT  EMPLOYEE.dep_depart_id%TYPE,  
    arg2      IN      EMPLOYEE.empl_name%TYPE,  
    arg3      OUT      INTEGER) AS  
    -- 过程内部的变量在 AS 和 BEGIN 之间定义  
    v_sal     EMPLOYEE.salary%TYPE;  
BEGIN  
    SELECT salary INTO v_sal FROM EMPLOYEE  
    WHERE dep_depart_id = arg1 AND empl_name = arg2;  
  
    arg3 := 0; -- Success  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        arg3 := 1;  
    WHEN TOO_MANY_ROWS THEN  
        arg3 := 2;  
    WHEN OTHERS THEN  
        arg3 := 3;  
END test_proc;  
/
```

■ 存储过程的调用

◆ 传参方式

- 按存储过程声明时的参数顺序传参，如为过程 test_proc 传参：

```
SET SERVEROUTPUT ON;
DECLARE
    v_did    EMPLOYEE.dep_depart_id%TYPE;
    v_ename  EMPLOYEE.empl_name%TYPE;
    v_res    INTEGER;
BEGIN
    v_did := 'ACC';
    v_ename := 'Luo jin';

    test_proc(v_did, v_ename, v_res); -- 调用存储过程
    DBMS_OUTPUT.PUT_LINE('Result is: ' || to_char(v_res));
END;
/
```

■ 存储过程的调用

◆ 传参方式（续）

- 按命名表示方式为存储过程传参，如为过程 test_proc 传参：

```
SET SERVEROUTPUT ON;
DECLARE
    v_did    EMPLOYEE.dep_depart_id%TYPE;
    v_ename  EMPLOYEE.empl_name%TYPE;
    v_res    INTEGER;
BEGIN
    v_did := 'ACC';
    v_ename := 'Luo jin';

    /* 注意以下传参的语法：将实参与 test_proc 的形参关联起来，
       这样就不必按存储过程声明时的参数顺序传参 */
    test_proc(arg1=>v_did, arg3=>v_res, arg2=>v_ename);
    DBMS_OUTPUT.PUT_LINE('Result is: ' || to_char(v_res));
END;
/
```

■ 存储过程的调用

- ◆ 通过 exec 或 call 指令调用存储过程
 - 如果某个存储过程的所有参数是 IN 类型，则在 sqlplus 下可以使用 exec 或 call（call 为 ODBC 通用方式）调用。
- ◆ 示例

```
CREATE OR REPLACE PROCEDURE proc1 (  
    arg1 INTEGER, -- 参数形式没指定，则默认为 IN  
    arg2 VARCHAR2) AS  
BEGIN  
    FOR n IN 0..arg1 LOOP  
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || ':' || arg2);  
    END LOOP;  
END proc1;  
/
```

```
call proc1 (10, 'hello');  
-- 或  
exec proc1 (10, 'hello');
```

■ 函数与存储过程：

- ◆ 两者的定义类似
- ◆ 存储过程接受 IN、OUT、IN OUT 形式的参数，但函数只接受 IN 形式的参数
- ◆ 函数必须声明返回值类型，在函数体中必须有返回语句
- ◆ 两者都存储在数据库中，在块中调用
- ◆ 存储过程只能作为一个 PL/SQL 块调用，而函数则可以在表达式中调用（作为表达式的一个部分）
- ◆ 两者的声明、执行、异常处理部分也有差异

■ 创建函数语法

```
CREATE [OR REPLACE] FUNCTION [schema.]function_name  
    [(argument [{IN | OUT | IN OUT}] data-type [...])]  
    RETURN return-datatype {IS | AS}  
    PL/SQL-Body;
```


■ 函数定义示例

-- 函数 calc_level 接受一个参数：员工 id，返回该员工的工资级别

```
CREATE OR REPLACE FUNCTION calc_level (  
    arg_eid EMPLOYEE.empl_id%TYPE)  
    RETURN INTEGER AS  
    v_lev    INTEGER;  
    v_sal    EMPLOYEE.salary%TYPE;  
BEGIN  
    SELECT salary INTO v_sal  
    FROM EMPLOYEE WHERE empl_id = arg_eid;  
  
    CASE  
        WHEN v_sal < 3000 THEN v_lev := 1;  
        WHEN v_sal BETWEEN 3000 AND 3999 THEN v_lev := 2;  
        WHEN v_sal BETWEEN 4000 AND 4999 THEN v_lev := 3;  
        ELSE v_lev := 4;  
    END CASE;
```

■ 函数定义示例（续）

```
        RETURN v_lev;  
  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RETURN 0;  
    WHEN OTHERS THEN  
        RETURN -1;  
END calc_level;  
/
```

■ 函数调用示例

```
-- 调用函数 calc_level()

DECLARE
    v_lev    INTEGER;
BEGIN
    v_lev := calc_level('ACC0001');
    DBMS_OUTPUT.PUT_LINE('Salary level of ACC0001 is: ' ||
TO_CHAR(v_lev));
END;
/
```

■ 使用 DROP 命令删除存储过程和函数

-- 删除存储过程

```
DROP PROCEDURE procedure-name;
```

-- 删除函数

```
DROP FUNCTION function-name;
```

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

■ 关于触发器

- ◆ 触发器 (Trigger) 是由某些数据库事件触发而执行的例程，我们可以将触发器将特定的数据库事件关联起来，当这些数据库事件发生时，触发器例程将被执行
- ◆ 触发的事件有以下 3 种类型：
 - DML 事件：如对表执行 DELETE、INSERT、UPDATE 操作
 - DDL 事件：如 CREATE、DROP、ALTER 操作的执行
 - 数据库事件：数据库级别的事件
- ◆ 触发器的典型应用：
 - 操作的监控
 - 数据同步
 - ...

- 创建触发器的语法 (DDL 和数据库事件)

```
CREATE [OR REPLACE] TRIGGER trigger_name  
{ BEFORE | AFTER } trigger_event  
ON [ DATABASE | schema ]  
[FOLLOWS other_trigger][DISABLE]  
[WHEN trigger_condition]  
trigger_body;
```

■ 创建触发器的语法 (DML 事件)

```
CREATE [OR REPLACE] TRIGGER trigger_name
{ BEFORE | AFTER } trigger_event  -- #1, #2
ON {table_or_view_reference |
NESTED TABLE nested_table_column OF view}
[REFERENCING [OLD AS old] [NEW AS new]  -- #3
[PARENT AS parent]]
[FOR EACH ROW ]  -- #4
[FOLLOWS other_trigger] [DISABLE]
[COMPOUND TRIGGER]
[WHEN trigger_condition]
trigger_body;
```

本次仅讨论 DML 事件相关的触发器

■ 创建触发器的语法 (DML 事件)

◆ 补充说明:

- #1 : BEFORE / AFTER 表示在事件发生前 / 后触发
- #2 : trigger_event 对于 DML 事件而言为: INSERT / UPDATE / DELETE
- #3 : NEW / OLD : 指事件产生后 (即 INSERT/UPDATE/DELETE) 表中的记录的新 / 旧值, 对于 UPDATE 而言, 有 NEW 和 OLD , 对于 INSERT 操作而言只有 NEW 值, 对于 DELETE 操作而言只有 OLD 值
- #4 : FOR EACH ROW 指定触发的级别为行级别 (即每改变一行触发一次), 如果不指定, 默认为表级别, 即对于一个表而言, 一次操作更改多行也只触发一次

注意: 如果 FOR EACH ROW 没指定, 则 #3 的 NEW / OLD 不存在

■ 说明

- ◆ 如同存储过程和函数，触发器能完成很多复杂的任务
- ◆ 以下将通过一个示例来演示触发器的使用
- ◆ 该示例将使用触发器来监控对表 EMPLOYEE 的修改性操作
- ◆ 针对 EMPLOYEE 任意一次修改性的操作将被记录到表 OP_LOG

■ 示例相关的表和 Sequence

```
-- 操作日志表
DROP TABLE op_log;
CREATE TABLE op_log (
    log_seq      INTEGER NOT NULL PRIMARY KEY,
    who          VARCHAR2(32)      NOT NULL,
    operation    VARCHAR2(16)      NOT NULL,
    rec_id       VARCHAR2(32)      NOT NULL,
    op_time      DATE              NOT NULL
);

-- Sequence , 用于产生表 op_log 的主键
DROP SEQUENCE seq_op_log;
CREATE SEQUENCE seq_op_log
INCREMENT BY    1
START WITH      1
MAXVALUE        999999
NOCYCLE
NOCACHE;
/
```

■ 触发器定义：形式 1

```
CREATE OR REPLACE TRIGGER trigger_empl
  AFTER INSERT OR UPDATE OR DELETE ON EMPLOYEE
  FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO op_log
    VALUES (seq_op_log.NEXTVAL, USER,
            'insert', :NEW.empl_id, SYSDATE);
  ELSIF UPDATING THEN
    INSERT INTO op_log
    VALUES (seq_op_log.NEXTVAL, USER,
            'update', :NEW.empl_id, SYSDATE);
  ELSIF DELETING THEN
    INSERT INTO op_log
    VALUES (seq_op_log.NEXTVAL, USER,
            'delete', :OLD.empl_id, SYSDATE);
  END IF;
END;
/
```

■ 触发器定义：形式 2

- ◆ 可以一个 trigger 关注一个事件，如下面以 INSERT 事件为例：

```
CREATE OR REPLACE TRIGGER trigger_empl_insert
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
BEGIN
  INSERT INTO op_log
  VALUES (seq_op_log.NEXTVAL, USER,
          'insert', :NEW.empl_id, SYSDATE);
END;
/
```

■ Oracle PL/SQL :

- ◆ 概要
- ◆ 语言元素
- ◆ 流程控制
- ◆ 异常处理
- ◆ 游标 (Cursor)
- ◆ 存储过程与函数
- ◆ 触发器 (Trigger)
- ◆ 包 (Package)

- 关于 PL/SQL 中包 (Package)
 - ◆ 包将一系列相关的对象逻辑上组织在一起。
 - ◆ 借助包的机制，可以将接口与实现分离
 - ◆ 包内可以声明存储过程、函数、游标、其它自定义类型（如 record ）以及变量
 - ◆ 包由两个独立的部分组成：包头和包体

■ 包头定义的语法

```
CREATE [OR REPLACE] PACKAGE package_name
    [ AUTHID { CURRENT_USER | DEFINER } ]
    { IS | AS }
    [definitions of public TYPEs
    ,declarations of public variables, types, and objects
    ,declarations of exceptions
    ,pragmas
    ,declarations of cursors, procedures, and functions
    ,headers of procedures and functions]
END [package_name];
```


■ 包头定义示例

```
CREATE OR REPLACE PACKAGE attendance_sys_procs AS
    TYPE rec_empl IS RECORD ( -- 定义 record 类型
        empl_no EMPLOYEE.empl_id%TYPE,
        sal_lev INTEGER
    );

    FUNCTION func1( -- 声明一个函数
        arg1      IN   INTEGER,
        arg2      IN   VARCHAR2
    ) RETURN VARCHAR2;

    PROCEDURE proc2 ( -- 声明一个存储过程
        eid        IN   VARCHAR2,
        result      OUT  INTEGER
    );

    /* 其它声明 */
END attendance_sys_procs;
/
```

■ 包体定义的语法

```
CREATE [OR REPLACE] PACKAGE BODY package_name
    { IS | AS }
    [definitions of private TYPEs
    ,declarations of private variables, types, and objects
    ,full definitions of cursors
    ,full definitions of procedures and functions]
[BEGIN
    executable_statements
[EXCEPTION
    exception_handlers]]
END [package_name];
```

■ 包体定义示例

```
CREATE OR REPLACE PACKAGE BODY attendance_sys_procs AS
    FUNCTION func1( -- 定义函数
        arg1      IN   INTEGER,
        arg2      IN   VARCHAR2
    ) RETURN VARCHAR2 AS -- 内部变量定义, 略
BEGIN
    -- 应用逻辑、EXCEPTION, 略
    RETURN 'Hill';
END func1;

PROCEDURE proc2 ( -- 定义存储过程
    eid          IN   VARCHAR2,
    result       OUT  INTEGER
) AS -- 内部变量定义, 略
BEGIN
    -- 应用逻辑、EXCEPTION, 略
    result := 0;
END proc2;
END attendance_sys_procs;
/
```

■ 包的使用

- ◆ 使用已经定义的包中的对象，需在对象之前加上包的名称，如使用包 attendance_sys_procs 内的存储过程 proc2 仅需在过程名之前冠以包名即可：

```
attendance_sys_procs.proc2(...);
```

- ◆ 包中存储过程、函数等对象的调用与普通存储过程等方式一致。