PL/SQL是是由甲骨文公司在90年代初开发，以提高SQL的功能。PL/SQL是嵌入在Oracle数据库中的三个关键的编程语言之一（随着SQL本身和Java）。PL/SQL是Oracle数据库对SQL语句的扩展。在普通SQL语句的使用上增加了编程语言的特点，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。

以下是关于PL/ SQL值得注意的事实：

* PL/SQL是一个完全可移植的，高性能的事务处理语言。
* PL/SQL提供了内置的解释器以及OS独立编程环境。
* PL/ SQL也可直接调用的命令行SQL\* Plus接口。
* 直接调用也可以从外部编程语言调用数据库。
* PL/SQL的一般语法是基于ADA和Pascal编程语言。 除了甲骨文，PL/SQL在TimesTen内存数据库和IBM DB2也可用。

PL/SQL具有以下特点：

* PL/SQL紧密结合集成SQL。
* 它提供了广泛的错误检查。
* 它提供了大量的数据类型。
* 它提供了多种编程结构。
* 它支持通过函数和程序结构化编程。
* 它支持面向对象的编程。
* 它支持开发Web应用程序和服务器的页面。

PL/SQL具有以下优点：

* SQL 是结构化查询语言，是数据库操作的标准语言，而 PL/SQL 是 Oracle 对 SQL 的过程化扩展语言，强调将 SQL 与程序控制结构（如条件语句、循环等）紧密集成，用于实现更复杂的业务逻辑。
* PL/SQL支持静态和动态SQL。
* 静态SQL支持DML操作和事务PL/SQL块控制。
* 动态SQL是SQL允许嵌入PL/SQL块的DDL语句。
* PL/SQL允许一次发送语句的整块到数据库。这降低了网络流量，并提供高性能的应用程序。
* PL/SQL给编程人员高的生产效率，因为它可以查询，转换并在数据库中更新数据。
* PL/SQL强劲的功能，如异常处理，封装，数据隐藏和面向对象数据类型可以节省设计和调试的时间。
* 编写PL/SQL应用程序是完全可移植的。
* PL/SQL提供了高的安全级别。
* PL/SQL提供了访问预定义SQL包。
* PL/SQL提供了面向对象的编程支持。
* PL/ SQL提供了用于开发Web应用程序和服务器页面的支持。

**PL/SQL 基本语法**

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

SQL

PL/SQL是块结构语言; PL/SQL程序划分成几个部分，并在每个部分中写入逻辑代码块。每个块由三个子部分组成

* 声明部分 - 此部分是以关键字DECLARE开头。这是一个可选部分，并定义了程序中要使用的所有变量，游标，子程序和其他元素。
* 可执行命令部分 - 此部分包含在关键字BEGIN和END之间，这是一个强制性部分。它由程序的可执行PL/SQL语句组成。它应该有至少一个可执行代码行，它可以只是一个NULL命令，表示不执行任何操作。
* 异常处理部分 - 此部分以关键字EXCEPTION开头。这是一个可选部分，它包含处理程序中错误的异常。

每个PL/SQL语句以分号(;)结尾。 使用BEGIN和END可以将PL/SQL块嵌套在其他PL/SQL块中。

**Hello World示例：**

DECLARE

message varchar2(20):= 'Hello, World!';

BEGIN

dbms\_output.put\_line(message);

END;

/

SQL

END;行表示PL/SQL块的结尾。要从SQL命令行运行代码，需要在代码的最后一行之后键入/字符。当上述代码在SQL提示符下执行时，它会产生以下结果：

Hello World

PL/SQL procedure successfully completed.

Shell

**PL/SQL注释**

PL/SQL支持单行和多行注释。注释中的所有字符都被PL/SQL编译器忽略。 PL/SQL单行注释以分隔符开头 --(双连字符)，多行注释由/\*和\*/括起来。

**PL/SQL 数据类型**

* 标量(SCALAR)类型 - 它是没有内部组件的单个值，例如：NUMBER，DATE或BOOLEAN等。
* 大对象(LOB)类型 - 指向与其他数据项(例如：文本，图形图像，视频剪辑和声音波形)分开存储的大对象的指针。
* 复合类型 - 具有可单独访问的内部组件的数据项。例如，集合和记录。
* 引用类型 - 指向其他数据项。

**PL/SQL标量数据类型和子类型**

数字 执行算术运算的数值。

字符 表示单个字符或字符串的字母数字值。

布尔 执行逻辑运算的逻辑值。

日期时间 用于表示日期和时间的值

**数字：**

|  |  |  |
| --- | --- | --- |
| 1 | PLS\_INTEGER | 带符号整数：-2,147,483,648至2,147,483,647，以32位表示 |
| 2 | BINARY\_INTEGER | 带符号整数：-2,147,483,648至2,147,483,647，以32位表示 |
| 3 | BINARY\_FLOAT | 单精度IEEE 754格式浮点数 |
| 4 | BINARY\_DOUBLE | 双精度IEEE 754格式浮点数 |
| 5 | NUMBER(prec, scale) | 在1E-130到(但不包括)1.0E126范围内的绝对值的定点或浮点数。NUMBER变量也可以表示0 |
| 6 | DEC(prec, scale) | ANSI特定定点类型，最大精度为38位十进制数字 |
| 7 | DECIMAL(prec, scale) | IBM具体定点类型，最大精度为38位十进制数字 |
| 8 | NUMERIC(pre, secale) | 浮点型，最大精度为38位十进制数 |
| 9 | DOUBLE PRECISION | ANSI特定浮点类型，最大精度为126位二进制数字(大约38位十进制数字) |
| 10 | FLOAT | ANSI和IBM特定浮点类型，最大精度为126位二进制数字(大约38位十进制数字) |
| 11 | INT | ANSI特定整数类型，最大精度为38位十进制数 |
| 12 | INTEGER | ANSI和IBM特定整数类型，最大精度为38位十进制数 |
| 13 | SMALLINT | ANSI和IBM特定整数类型，最大精度为38位十进制数 |
| 14 | REAL | 浮点型，最大精度为63位二进制数字(约十八位数) |

**字符：**

|  |  |  |
| --- | --- | --- |
| 1 | CHAR | 固定长度字符串，最大大小为32,767字节 |
| 2 | VARCHAR2 | 最大大小为32,767字节的可变长度字符串 |
| 3 | RAW | 最大大小为32,767字节的可变长度二进制或字节字符串，不由PL/SQL解释 |
| 4 | NCHAR | 固定长度的国家字符串，最大大小为32,767字节 |
| 5 | NVARCHAR2 | 可变长度的国家字符串，最大大小为32,767字节 |
| 6 | LONG | 最大长度为32,760字节的可变长度字符串 |
| 7 | LONG RAW | 最大大小为32,760字节的可变长度二进制或字节字符串，不由PL/SQL解释 |
| 8 | ROWID | 物理行标识符，普通表中的行的地址 |
| 9 | UROWID | 通用行标识符(物理，逻辑或外部行标识符) |

**布尔值：**

BOOLEAN数据类型存储逻辑运算中使用的逻辑值。逻辑值为布尔值:TRUE,FALSE以及NULL值。

注意！！但是，SQL没有类似于BOOLEAN的数据类型。 因此，布尔值不能用于：

* SQL语句
* 内置SQL函数(如:TO\_CHAR)
* 从SQL语句调用PL/SQL函数

**日期时间：**

|  |  |  |  |
| --- | --- | --- | --- |
| YEAR | -4712至9999(不包括第0年) | 任意非零整数 |  |
| MONTH | 01 ~ 12 | 01 ~ 11 |  |
| DAY | 01至31(限于MONTH和YEAR的值，根据本地日历的规则) | 任何非零整数 |  |
| HOUR | 00 ~ 23 | 00 ~ 23 |  |
| MINUTE | 00 ~ 59 | 00 ~ 59 |  |
| SECOND | 00 ~ 59.9(n)，其中9(n)是时间分秒的精度 | 00 ~ 59.9(n)，其中9(n)是间隔分数秒的精度 |  |
| TIMEZONE\_HOUR | -12至14(范围适应夏令时更改) | 不适用 |  |
| TIMEZONE\_MINUTE | 00 ~ 59 | 不适用 |  |
| TIMEZONE\_REGION | 在动态性能视图V$TIMEZONE\_NAMES找到 | 不适用 |  |
| TIMEZONE\_ABBR | 在动态性能视图V$TIMEZONE\_NAMES找到 | 不适用 |  |

**PL/SQL大对象(LOB)数据类型：**

| **数据类型** | **描述** | **大小** |
| --- | --- | --- |
| BFILE | 用于在数据库外的操作系统文件中存储大型二进制对象。 | 取决于系统，但不得超过4GB。 |
| BLOB | 用于在数据库中存储的大型二进制对象 | 8TB至128TB |
| CLOB | 用于在数据库中存储大字符数据。 | 8TB至128TB |
| NCLOB | 用于在数据库中存储大块NCHAR数据。 | 8TB至128TB |

**PL/SQL中的NULL**

PL/SQL中的NULL值表示丢失或未知数据，它们不是整数，字符或任何其他特定数据类型。 请注意！！NULL与空数据字符串或空字符值\0不同。可以将一个null值分配给其它变量，但不能等同于任何东西，包括其自身(null)。

**PL/SQL 变量**

PL/SQL变量声明：必须在声明部分或包中声明PL/SQL变量作为全局变量。当声明一个变量时，PL/SQL为变量的值分配内存，并且存储位置由变量名称标识。

声明变量的语法是：

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

SQL

其中，variable\_name是PL/SQL中的有效标识符，datatype必须是有效的PL/SQL数据类型或任何用户定义的数据类型，比如：

sales number(10, 2);

pi CONSTANT BINARY\_DOUBLE := 3.1415;

name varchar2(25);

greetings varchar2(20) DEFAULT 'Have a Good Day';

1. 声明这是一个数值型的变量，名为sales。最大总位数为10小数位保留2位
2. 这是一个 常量 pi 的声明（值不能被修改）（CONSTANT关键字）
3. 这是一个 字符串变量 name 的声明，最多可以存储 25 个字符
4. 这是一个 字符串变量 greetings 的声明，默认值为Have a Good Day

注意！！给变量设置默认值用:= 或 DEFAULT，如果不设置默认是null

**PL/SQL变量作用域**

局部变量 - 内部块中声明的变量，外部块不可访问。

全局变量 - 在最外部块或包中声明的变量。

比如：

DECLARE

-- Global variables 全局

num1 number := 95;

num2 number := 85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: ' || num1);

dbms\_output.put\_line('Outer Variable num2: ' || num2);

DECLARE

-- Local variables 局部

num1 number := 195;

num2 number := 185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: ' || num1);

dbms\_output.put\_line('Inner Variable num2: ' || num2);

END;

END;

/

SQL

**将SQL查询结果分配给PL/SQL变量**

可以使用SQL的SELECT INTO语句将值分配给PL/SQL变量。 对于SELECT列表中的每个项目，INTO列表中必须有一个对应的类型兼容变量。

**也就是说，**当我们不知道数据库中某一列的数据类型时，也可以直接用 %type 来将从数据库内查询出来的数据类型赋给定义的变量

DECLARE

c\_id customers.id%type := 1;

c\_name customers.Name%type;

c\_addr customers.address%type;

c\_sal customers.salary%type;

BEGIN

SELECT name, address, salary INTO c\_name, c\_addr, c\_sal

FROM customers

WHERE id = c\_id;

dbms\_output.put\_line

('Customer ' ||c\_name || ' from ' || c\_addr || ' earns ' || c\_sal);

END;

/

SQL

上述代码的运行结果：

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully

**PL/SQL 算术运算符**

|  |  |  |
| --- | --- | --- |
| + | 两个操作数相加 | A + B = 15 |
| - | 从第一个减去第二个操作数 | A - B = 5 |
| \* | 将两个操作数相乘 | A \* B = 50 |
| / | 从第一个除以第二个操作数 | A / B = 2 |
| \*\* | 指数运算符，提出一个操作数到其他的幂值 | A \*\* B = 100000 |

**PL/SQL 关系运算符**

|  |  |  |
| --- | --- | --- |
| = | 检查两个操作数的值是否相等，如果是，则条件成立。 | (A = B)为假 |
| !=,<>，~= | 检查两个操作数的值是否相等，如果两个值不相等则条件成为真。 | (A != B)为真 |
| > | 检查左操作数的值是否大于右操作数的值，如果是，则条件成为真。 | (A > B) 为假 |
| < | 检查左操作数的值是否小于右操作数的值，如果是，则条件成为真。 | (A < B) 条件为真。 |
| >= | 检查左操作数的值是否大于或等于右操作数的值，如果是，则条件成为真。 | (A >= B) 为假 |
| <= | 检查左操作数的值是否小于或等于右操作数的值，如果是，则条件成为真。 | (A <= B) 为真 |

**PL/SQL 比较运算符**

|  |  |  |
| --- | --- | --- |
| LIKE | LIKE运算符将字符，字符串或CLOB值与模式进行比较，如果值与模式匹配，则返回TRUE，否则返回FALSE。 | 如果'Zara Ali' LIKE 'Z％A\_i'返回一个布尔值true，而'Nuha Ali' LIKE'Z％A\_i'返回一个布尔值。 |
| BETWEEN | BETWEEN运算符测试值是否在指定范围内。x BETWEEN a AND b表示x >= a和x <= b。 | 如果x = 10，那么在5到20之间则x返回true，x在5和10之间则x返回true，但是x在11和20之间返回false。 |
| IN | IN运算符测试集成员数据。 x IN(set)表示x等于集合中的任何成员数据。 | 如果x ='m'，则在('a'，'b'，'c')中x返回false，而在('m'，'n'，'o')中x返回true。 |
| IS NULL | IS NULL运算符如果其操作数为NULL返回值为TRUE，如果不为NULL则返回FALSE。 涉及NULL值的比较总是产生NULL。 | 如果x ='m'，则is null'返回false 。 |

**PL/SQL 运算符优先级**

|  |  |
| --- | --- |
| \*\* | 指数幂运算符 |
| +, - | 标识符，负数 |
| \*, / | 乘法，除法 |
| +, -, ΙΙ | 加，减，连接 |
| NOT | 逻辑否定 |
| AND | 连词(逻辑与) |
| OR | 包含(逻辑或) |

**PL/SQL IF-THEN语句**

IF-THEN语句的语法是：

IF condition THEN

S;

END IF;

SQL

比如：

DECLARE

a number(2) := 10;

BEGIN

a:= 10;

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

dbms\_output.put\_line('a is less than 20 ' );

END IF;

dbms\_output.put\_line('value of a is : ' || a);

END;

/

SQL

当上述代码在SQL提示符下执行时，它会产生以下结果：

a is less than 20

value of a is : 10

PL/SQL procedure successfully completed.

Shell

**PL/SQL IF-THEN-ELSE语句**

IF-THEN-ELSE语句的语法是：

IF condition THEN

S1;

ELSE

S2;

END IF;

SQL

**PL/SQL IF-THEN-ELSIF语句**

PL/SQL编程语言中的IF-THEN-ELSIF语句的语法是：

IF(boolean\_expression 1)THEN

S1; -- Executes when the boolean expression 1 is true

ELSIF( boolean\_expression 2) THEN

S2; -- Executes when the boolean expression 2 is true

ELSIF( boolean\_expression 3) THEN

S3; -- Executes when the boolean expression 3 is true

ELSE

S4; -- executes when the none of the above condition is true

END IF;

SQL

注意！！PL/SQL编程语言中要用ELSIF而不是ELSEIF

**PL/SQL CASE语句**

PL/SQL中的case语句的语法是：

CASE selector

WHEN 'value1' THEN S1;

WHEN 'value2' THEN S2;

WHEN 'value3' THEN S3;

...

ELSE Sn; *-- default case*

END CASE;

SQL

**PL/SQL 可搜索CASE语句**

**可搜索的CASE语句没有选择器，语句中的WHEN子句包含给出布尔值的搜索条件。**

PL/SQL中可搜索的case语句的语法是：

CASE

WHEN selector = 'value1' THEN S1;

WHEN selector = 'value2' THEN S2;

WHEN selector = 'value3' THEN S3;

...

ELSE Sn; -- default case

END CASE;

SQL

**可搜索CASE语句可以应对更复杂的条件：**

**比如如下是一个普通case语句：**

DECLARE

status\_code CHAR := 'A';

status\_desc VARCHAR2(20);

BEGIN

status\_desc := CASE status\_code

WHEN 'A' THEN 'Active'

WHEN 'I' THEN 'Inactive'

WHEN 'P' THEN 'Pending'

ELSE 'Unknown'

END;

DBMS\_OUTPUT.PUT\_LINE(status\_desc);

END;

**将它改写成可搜索case语句：**

DECLARE

status\_code CHAR := 'A';

status\_desc VARCHAR2(20);

BEGIN

status\_desc := CASE

WHEN status\_code = 'A' THEN 'Active'

WHEN status\_code = 'I' THEN 'Inactive'

WHEN status\_code = 'P' THEN 'Pending'

WHEN status\_code IS NULL THEN 'Not Set'

WHEN status\_code IN ('X', 'Y') THEN 'Special Case'

ELSE 'Unknown'

END;

DBMS\_OUTPUT.PUT\_LINE(status\_desc);

END;

注意！！虽然两种写法逻辑一致，但可搜索case可以加如上的复杂条件

**PL/SQL 嵌套IF-THEN-ELSE语句**

**在PL/SQL编程中嵌套IF-ELSE语句总是合法的，也就是说可以在一个IF或ELSE IF语句中使用另一个IF或ELSE IF语句。**

PL/SQL中嵌套IF-ELSE语句的语法是：

IF( boolean\_expression 1)THEN

-- executes when the boolean expression 1 is true

IF(boolean\_expression 2) THEN

-- executes when the boolean expression 2 is true

sequence-of-statements;

END IF;

ELSE

-- executes when the boolean expression 1 is not true

else-statements;

END IF;

SQL

**PL/SQL 基本LOOP循环**

**基本循环结构包含LOOP和END LOOP语句之间的语句序列。通过每次迭代，执行语句序列，然后在循环顶部继续控制。（内部多为if条件）**

LOOP

Sequence of statements;

END LOOP;

SQL

**例如：**

SET SERVEROUTPUT ON SIZE 1000000;

*-- 启用 DBMS\_OUTPUT 输出功能，用于在控制台上显示输出内容*

*-- 设置输出缓冲区大小为 1,000,000 字节（最多 1MB）*

*-- 如果没加 SET SERVEROUTPUT ON，则无法在控制台看到输出*

DECLARE

x number := 10;

BEGIN

LOOP

dbms\_output.put\_line(x);

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms\_output.put\_line('After Exit x is: ' || x);

END;

/

SQL

**上文意为当x大于50时退出（exit）循环，也可以用exit when来替代if then（更简洁）：**

LOOP

dbms\_output.put\_line(x);

x := x + 10;

exit WHEN x > 50;

END LOOP;

**PL/SQL while循环**

**只要给定条件为真，PL/SQL编程语言中的WHILE LOOP语句重复执行目标语句。**

WHILE LOOP语句的语法如下：

WHILE condition LOOP

sequence\_of\_statements

END LOOP;

SQL

**与单loop循环的区别：**不用再用exit控制循环，直到while内的条件不满足时，循环自动停止

**PL/SQL for循环**

**FOR LOOP语句是一种重复控制结构，可以有效地编写一个需要执行特定次数的循环。**

如何使用FOR LOOP语句：

FOR counter IN initial\_value .. final\_value LOOP

sequence\_of\_statements;

END LOOP;

SQL

注意！！循环变量或计数器的initial\_value和final\_value可以是文字，变量或表达式，但必须对数字求值。 否则，PL/SQL引发预定义的异常VALUE\_ERROR。而且，initial\_value不必为1; 但是，循环计数器增量(或减量)必须为1。

比如：

SET SERVEROUTPUT ON SIZE 100000;

DECLARE

a number(2);

BEGIN

FOR a in 10 .. 15 LOOP

dbms\_output.put\_line('value of a: ' || a);

END LOOP;

END;

/

SQL

输出为：

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

也就是说，当a满足在10到15之间这个条件时，执行循环体，并每次将a的值增加1

**反转FOR LOOP语句（每次递减）**

默认情况下，迭代从初始值**递增**到最终值。但是也可以使用REVERSE关键字来反转这个顺序。在这种情况下，每次迭代后，循环计数器**递减**。

比如：

SET SERVEROUTPUT ON SIZE 100000;

DECLARE

a number(2);

BEGIN

FOR a in REVERSE 10 .. 15 LOOP

dbms\_output.put\_line('value of a: ' || a);

END LOOP;

END;

/

SQL

输出为：

value of a: 15

value of a: 14

value of a: 13

value of a: 12

value of a: 11

value of a: 10

**PL/SQL 嵌套循环**

**PL/SQL允许在一个循环中使用另一个循环。**

比如：

LOOP

Sequence of statements1

LOOP

Sequence of statements2

END LOOP;

END LOOP;

SQL

**PL/SQL 字符串**

PL/SQL中的字符串实际上是一个具有可选大小规格的字符序列。字符可以是数字，字母，空白，特殊字符或全部的组合

PL/SQL提供三种字符串：

* 固定长度字符串 - 在这样的字符串中，程序员在声明字符串时指定长度。该字符串的右边填充规定的长度。
* 可变长度字符串 - 在这样的字符串中，指定字符串的最大长度达32,767，并且不会填充。
* 字符大对象(CLOB) - 这些可变长度字符串最多可达128TB。

**声明字符串变量：**

DECLARE

name varchar2(20);

company CHAR(30);

introduction clob;

注意！！CHAR(n)为固定长度，长度始终为 n，不足补空格，VARCHAR2(n)为可变长度，仅占用实际字符长度

**PL/SQL字符串函数和运算符**

PL/SQL提供用于连接两个字符串的级联运算符(||)。以及如下函数：

|  |  |  |
| --- | --- | --- |
| 1 | ASCII(x); | 返回字符x的ASCII值。 |
| 2 | CHR(x); | 返回ASCII值为x的字符。 |
| 3 | CONCAT(x, y); | 连接两个字符串x和y，并返回连接后的字符串。 |
| 4 | INITCAP(x); | 将x中每个单词的初第一个字母转换为大写，并返回该字符串。 |
| 5 | INSTR(x, find\_string [, start] [, occurrence]); | 在x字符串中搜索find\_string子串并返回找到的位置。 |
| 6 | INSTRB(x); | 返回字符串x在另一个字符串中第一次再现的位置，但返回值(以字节为单位)。 |
| 7 | LENGTH(x); | 返回x中的字符数，也是计算字符串的长度。 |
| 8 | LENGTHB(x); | 返回单字节字符集的字符串长度(以字节为单位)。 |
| 9 | LOWER(x); | 将x字符串中的字母转换为小写，并返回此小写字符串。 |
| 10 | LPAD(x, width [, pad\_string]) ; | 使用空格垫放在x字符串的左边，以使字符串的长度达到宽度字符。 |
| 11 | LTRIM(x [, trim\_string]); | 修剪x字符串左边的字符。 |
| 12 | NANVL(x, value); | 如果x匹配NaN特殊值(而不是数字)，则返回值，否则返回x字符串。 |
| 13 | NLS\_INITCAP(x); | 与INITCAP(x)函数相同，只不过它可以使用NLSSORT指定的其他排序方法。 |
| 14 | NLS\_LOWER(x) ; | 与LOWER(x)函数相同，除了可以使用NLSSORT指定的不同排序方法。 |
| 15 | NLS\_UPPER(x); | 与UPPER()函数相同，除了可以使用NLSSORT指定的不同排序方法。 |
| 16 | NLSSORT(x); | 更改排序字符的方法。必须在任何NLS()函数之前指定; 否则，将使用默认排序。 |
| 17 | NVL(x, value); | 如果x为null则返回value值; 否则返回x。 |
| 18 | NVL2(x, value1, value2); | 如果x不为null则返回值value1; 如果x为null，则返回value2。 |
| 19 | REPLACE(x, search\_string, replace\_string); | 在x字符串中搜索search\_string并将其替换为replace\_string。 |
| 20 | RPAD(x, width [, pad\_string]); | 使用空格垫放在x字符串的右边，以使字符串的长度达到宽度字符。 |
| 21 | RTRIM(x [, trim\_string]); | 从右边修剪x字符串。 |
| 22 | SOUNDEX(x) ; | 返回一个包含x的语音表示的字符串。 |
| 23 | SUBSTR(x, start [, length]); | 返回x字符串从指定start位置开始到一个可选指定长度(length)范围内的子字符串。 |
| 24 | SUBSTRB(x); | 与SUBSTR()相同，除了参数以字节表示，还支持单字节字符系统的字符。 |
| 25 | TRIM([trim\_char FROM) x); | 修剪x字符串的左边和右边的字符。 |
| 26 | UPPER(x); | 将x中的字母转换为大写，并返回此大写后的字符串。 |

**PL/SQL 数组**

PL/SQL中的数组是，一种称为VARRAY的数据结构，它可以存储相同类型的元素的固定大小顺序集合。varray用于存储有序的数据集合，但通常最好将数组视为相同类型变量的集合。

**创建Varray类型**

**1，在模式(schema)级创建VARRAY类型的基本语法是：**

CREATE OR REPLACE TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

SQL

其中：

* varray\_type\_name是一个有效的属性名称；
* n是varray中元素的数量(最大值)；
* element\_type是数组元素的数据类型。

注意！！此时创建的varray是**数据库级别 (schema)级** 的，即全局作用域

1. **在PL/SQL块中创建VARRAY类型的基本语法是**

TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

SQL

注意！！此时创建的varray是**块级别**的，即块级作用域

**可以使用 ALTER TYPE 语句来更改 Oracle 中已定义的 VARRAY 类型的最大大小。**

比如：

CREATE OR REPLACE TYPE grades\_varray AS VARRAY(5) OF NUMBER;

现在想把最大容量从 5 改为 10，可以使用：

ALTER TYPE grades\_varray MODIFY LIMIT 10;

**数组运用实例：**

SET SERVEROUTPUT ON SIZE 99999;

*-- PL/SQL 匿名块开始*

DECLARE

*-- 定义一个 VARRAY 类型，最多存储 5 个名字，每个名字最长 10 个字符*

TYPE namesarray IS VARRAY(5) OF VARCHAR2(10);

*-- 定义一个 VARRAY 类型，最多存储 5 个整数，表示学生成绩*

TYPE grades IS VARRAY(5) OF INTEGER;

*-- 声明变量 names，类型为 namesarray*

names namesarray;

*-- 声明变量 marks，类型为 grades*

marks grades;

*-- 用于存储学生总数*

total INTEGER;

BEGIN

*-- 初始化 names 数组，赋值为 5 个学生名字*

names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

*-- 初始化 marks 数组，赋值为 5 个对应的学生分数*

marks := grades(98, 97, 78, 87, 92);

*-- 用.COUNT方法获取 names 数组中学生的总数，存入 total 变量*

total := names.COUNT;

*-- 输出学生总数*

DBMS\_OUTPUT.PUT\_LINE('Total ' || total || ' Students');

*-- 使用 FOR 循环遍历每位学生，输出他们的名字和对应的分数*

FOR i IN 1 .. total LOOP

DBMS\_OUTPUT.PUT\_LINE('Student: ' || names(i) || ' Marks: ' || marks(i));

END LOOP;

END;

/

注意！！

* varrays是一维数组。
* varray在声明时自动为NULL，并且必须在引用元素之前初始化它。
* 不同于java，PL/SQL中没有.length方法！！只能使用.COUNT来获取数组长度。
* 虽然可以用数组索引（下标）引用数组内的元素，但不能用[]！！要用()。
* 不同于java中索引起始值为0，在Oracle环境中，varrays的起始索引始终为1。

**PL/SQL 存储过程**

在 PL/SQL 中，存储过程（Stored Procedure） 是一种预编译、存储在数据库中的程序模块，用于封装可重用的业务逻辑。相当于封装好的一个方法。

可以在以下几个地方中创建一个子程序：

* 在模式(schema)级别中
* 一个程序包中
* 在PL/SQL块中

在模式(schema)级别中，子程序是一个独立的子程序。它是使用CREATE PROCEDURE或CREATE FUNCTION语句创建的。它存储在数据库中，可以使用DROP PROCEDURE或DROP FUNCTION语句进行删除。

在包中创建的子程序是打包的子程序。它存储在数据库中，只有当使用DROP PACKAGE语句删除程序包时，才能将其删除。我们将在“PL/SQL程序包”一章中讨论程序包的应用。

在PL/SQL块中。 PL/SQL提供两种子程序：

1. 函数 - 这些子程序返回单个值; 主要用于计算和返回值。
2. 存储过程(程序) - 这些子程序不直接返回值; 主要用于执行动作。

**创建存储过程（模式(schema)级别中）**

简化语法如下：

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

AS

BEGIN

< procedure\_body >

END procedure\_name;

*-- end后面的名称可以省略*

SQL

其中：

* procedure-name是要创建的存储过程的名称。
* [OR REPLACE]选项允许修改现有的过程。
* 可选参数列表包含参数的名称，模式和类型。IN表示将从外部传入的参数，OUT表示返回值，IN OUT表示可读写参数（调用时传入，修改后传出）
* procedure-body包含可执行部分。
* 使用AS关键字而不是IS关键字来创建存储过程。

比如：

SET SERVEROUTPUT ON SIZE 99999;

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

*-- 执行存储过程*

exec greetings;

*-- 或者*

EXECUTE greetings;

*-- exec和execute两种写法是完全等价的，没有功能上的区别*

SQL

**除了用EXECUTE关键字调用，也可以在另外一个PL/SQL块中调用（直接用名称调用）：**

BEGIN

greetings;

END;

/

SQL

**传递参数的方法**

假设在PL/SQL块中有这样一个名为findmin的存储过程；

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

...

END;

BEGIN

...

findMin(a, b, c);

...

END;

/

SQL

此时，实际参数(实参)可以通过三种方式传递：

1. 位置符号，使a b c按照顺序分别对应x y z：

findMin(a, b, c);

SQL

1. 命名符号，手动给形参匹配实参

findMin(x => a, y => b, z => c);

SQL

1. 混合符号，上述两种方式混合：

findMin(a, b, c, m => d);

SQL

注意！！混合符号的情况下，命名符号 => 不能放在最前面，否则系统找不到正确顺序

比如：

findMin(x => a, b, c, d);

这就是不合法的。

**PL/SQL 函数**

1. **创建函数**

简化语法如下：

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

SQL

* function-name是指定要创建的函数的名称。
* [OR REPLACE]选项指示是否允许修改现有的函数。
* 可选参数列表包含参数的名称，模式和类型。 IN表示将从外部传递的值，OUT表示将用于返回过程外的值的参数。
* 函数必须包含一个返回(RETURN)语句。
* RETURN子句指定要从函数返回的数据类型。
* function-body包含可执行部分。
* 使用AS关键字代替IS关键字，用来创建独立的函数。

1. **调用函数**

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('当前客户的总数为: ' || c);

END;

/

SQL

1. **PL/SQL递归函数**

**程序或子程序可能会调用另一个子程序。当子程序调用自身时，它被称为递归调用，该过程称为递归。**

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f := 1;

ELSE

f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 10;

factorial := fact(num);

dbms\_output.put\_line(' 数字 '|| num || ' 的阶乘积是： ' || factorial);

END;

/

SQL

**函数（Function）和过程（Procedure）的区别总结**

|  |  |  |
| --- | --- | --- |
| 比较项 | 函数（Function） | 过程（Procedure） |
| 是否必须返回值 | ✅ 必须返回一个值 | ❌ 不返回值（可通过 OUT 参数返回多个值）或SELECT INTO语句未返回任何行，则返回TRUE。 否则返回FALSE。 |
| 调用方式 | 可以直接在表达式中调用（如 := func(...)） | 不能嵌入表达式中，需单独调用（EXEC proc） |
| 调用场景 | 适合有计算逻辑并需要结果返回时 | 适合执行操作、更新表、批处理等业务逻辑 |
| 可否用在 SQL 语句中 | ✅ 可以（前提是无副作用、无 DML） | ❌ 不可以 |
| 参数类型支持 | 仅支持 IN（语法上可以有 OUT/IN OUT 参数） | 支持 IN、OUT、IN OUT |
| 返回值数量 | 只能返回一个值（用 RETURN） | 可通过多个 OUT 参数返回多个值 |

**PL/SQL 游标**

可以命名一个游标，以便在程序中引用它来获取和处理SQL语句返回的行，一次处理一个(行)。PL/SQL中有两种类型的游标 -

* **隐式游标**
* **显式游标**

**隐式游标**

当执行SQL语句时，如果语句没有显式游标，则Oracle会自动创建隐式游标。程序员无法控制隐式游标及其信息。

每当发出DML语句(INSERT，UPDATE和DELETE)时，隐式游标与此语句相关联。 对于INSERT操作，游标保存需要插入的数据。对于UPDATE和DELETE操作，游标标识将受到影响的行。

在PL/SQL中，可以将最近的隐式游标引用为SQL游标，它始终具有%FOUND，%ISOPEN，%NOTFOUND和%ROWCOUNT等属性。

|  |  |  |
| --- | --- | --- |
| 1 | %FOUND | 如果INSERT，UPDATE或DELETE语句影响一行或多行，或SELECT INTO语句返回一行或多行，则返回TRUE，否则返回FALSE。 |
| 2 | %NOTFOUND | 与%FOUND的逻辑相反。 如果INSERT，UPDATE或DELETE语句没有影响任何行，或SELECT INTO语句未返回任何行，则返回TRUE。 否则返回FALSE。 |
| 3 | %ISOPEN | 由于Oracle在执行关联的SQL语句后会自动关闭SQL游标，因此总是为隐式游标返回FALSE。 |
| 4 | %ROWCOUNT | 返回受INSERT，UPDATE或DELETE语句，或者受SELECT INTO语句影响的行数。 |

比如：

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

IF sql%notfound THEN

dbms\_output.put\_line('没有找到客户信息~');

ELSIF sql%found THEN

total\_rows := sql%rowcount;

dbms\_output.put\_line('一共有：' || total\_rows || ' 个客户的工资被更新！ ');

END IF;

END;

/

SQL

也就是说，当一个sql语句被执行时，会自动创建一个名为sql的隐式游标，可以直接调用

**显式游标**

显式游标是用于获得对上下文区域的更多控制的程序员定义的游标。应在PL/SQL块的声明部分中定义一个显式游标。它是在一个返回多行的SELECT语句中创建的。语法是：

CURSOR c\_customers IS

*-- 在is后面写select语句*

SELECT id, name, address FROM customers;

SQL

打开游标，将为游标分配内存，并使其准备好将SQL语句返回的行记录数据提取到其中。例如，打开上面定义的游标，如下所示：

OPEN c\_customers;

SQL

获取游标，一次仅访问一行。 例如，从上面打开的游标中获取行，如下所示代码：

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

SQL

关闭游标，意味着释放分配的内存。例如，关闭上面打开的游标，如下所示：

CLOSE c\_customers;

SQL

以下是一个完整的例子来说明显式游标的概念：

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

c\_id customers.id%type;

c\_name customers.name%type;

c\_addr customers.address%type;

CURSOR c\_customers is

SELECT id, name, address FROM customers;

BEGIN

OPEN c\_customers;

LOOP

FETCH c\_customers into c\_id, c\_name, c\_addr;

EXIT WHEN c\_customers%notfound;

dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);

END LOOP;

CLOSE c\_customers;

END;

/

SQL

**PL/SQL 记录**

记录是可以容纳不同种类的数据项的数据结构。 记录由不同的字段组成，类似于数据库表的一行。PL/SQL可以处理以下类型的记录：

* 基于表的记录
* 基于游标的记录
* 用户定义的记录

**基于表的记录**

%ROWTYPE属性使程序员能够创建基于表和基于游标的记录。

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

customer\_rec customers%rowtype;

BEGIN

SELECT \* into customer\_rec

FROM customers

WHERE id = 5;

dbms\_output.put\_line('客户ID: ' || customer\_rec.id);

dbms\_output.put\_line('客户姓名: ' || customer\_rec.name);

dbms\_output.put\_line('客户地址: ' || customer\_rec.address);

dbms\_output.put\_line('客户薪资: ' || customer\_rec.salary);

END;

/

SQL

也就是说：声明一个记录变量 customer\_rec，其结构与 customers 表中的一行完全一致。customer\_rec 变量包含了 customers 表中所有列。比如：id、name、address、salary 等。可以通过 customer\_rec.id、customer\_rec.name 等方式来访问对应字段的值

注意！！在 PL/SQL 中，基于表的记录（如使用 %ROWTYPE）一次只能存储并处理一行数据。如果你需要处理多行数据，有两种常见做法：

1. 使用基于游标的记录 （见后文）
2. 用for...in循环：（in后面直接加查询语句）

BEGIN

FOR customer\_rec IN (

SELECT \* FROM customers WHERE salary > 5000

) LOOP

DBMS\_OUTPUT.PUT\_LINE('客户姓名: ' || customer\_rec.name);

END LOOP;

END;

/

**基于游标的记录**

游标记录可以逐行处理查询结果，每次 FETCH 一行，通常和 %ROWTYPE 配合使用：

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

CURSOR customer\_cur is

SELECT id, name, address

FROM customers;

customer\_rec customer\_cur%rowtype;

BEGIN

OPEN customer\_cur;

LOOP

FETCH customer\_cur into customer\_rec;

EXIT WHEN customer\_cur%notfound;

DBMS\_OUTPUT.put\_line(customer\_rec.id || ' ' || customer\_rec.name);

END LOOP;

END;

/

SQL

1. 定义一个游标 customer\_cur，它用于选出 customers 表中的 id、name 和 address 三列。
2. 使用 %ROWTYPE 定义一个记录变量 customer\_rec，它的结构和 customer\_cur 返回的一行一致，即包含 id、name 和 address 三列。
3. 启动一个循环：每次从游标中取出一行数据赋值给 customer\_rec。如果没有更多数据了，%NOTFOUND 为 TRUE，退出循环。

**用户定义的记录**

PL/SQL提供了一个用户自定义的记录类型，允许程序员定义不同的记录结构。比如，要跟踪记录图书信息，例如可能要跟踪每本书的以下属性：

* 标题
* 作者
* 学科
* 图书ID

定义记录的语法如下：

DECLARE

TYPE books IS RECORD

(

title varchar(50),

author varchar(50),

subject varchar(100),

book\_id number

);

book1 books;

book2 books;

SQL

即：

1. 定义了一个名为books的记录类型
2. 定义了两本书，book1和book2，类型为books记录

需要访问字段时，用点(.)运算符：

BEGIN

-- Book 1 specification

book1.title := 'C Programming';

book1.author := 'TanHao';

book1.subject := 'C Programming Tutorial';

book1.book\_id := 1920122;

-- Book 2 specification

book2.title := 'Telecom Billing';

book2.author := 'LiDawei';

book2.subject := 'Telecom Billing Tutorial';

book2.book\_id := 2032942;

END;

/

SQL

**PL/SQL 异常处理**

程序执行过程中的错误情况是一个例外(异常)。PL/SQL支持程序员在程序中使用EXCEPTION块捕获这些发生错误的条件，并针对错误情况采取适当的措施。PL/SQL中有两种异常：

* 系统定义的异常
* 用户定义的异常

异常的语法：

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

*-- 这是一个通用的异常处理语句，必须放在最后，用来捕获所有未明确处理的异常。*

exception3-handling-statements

END;

SQL

**系统预定义异常（Predefined Exception）：**

|  |  |  |
| --- | --- | --- |
| 异常名 | 错误码 | 说明 |
| NO\_DATA\_FOUND | ORA-01403 | 查询没有返回数据（如 SELECT INTO 未找到记录）。 |
| TOO\_MANY\_ROWS | ORA-01422 | SELECT INTO 返回了多于一行数据 |
| ZERO\_DIVIDE | ORA-01476 | 被零除 |
| INVALID\_CURSOR | ORA-01001 | 无效的游标操作 |
| CURSOR\_ALREADY\_OPEN | ORA-06511 | 打开的游标又被打开 |
| DUP\_VAL\_ON\_INDEX | ORA-00001 | 唯一索引冲突 |
| VALUE\_ERROR | ORA-06502 | 数据类型或长度不匹配（如数值超界、字符串太长等） |
| INVALID\_NUMBER | ORA-01722 | 字符串转换为数字失败 |
| LOGIN\_DENIED | ORA-01017 | 登录失败，用户名或密码错误 |
| STORAGE\_ERROR | ORA-06500 | 内存或存储相关错误 |

比如：

EXCEPTION

WHEN no\_data\_found THEN

dbms\_output.put\_line('没有找到符合条件的客户信息!');

WHEN others THEN

dbms\_output.put\_line('Error!');

**引发自定义异常**

只要有内部数据库错误，数据库服务器就会自动产生(引发)异常，但程序员可以使用命令RAISE明确地主动引发异常。以下是引发异常的简单语法：

DECLARE

exception\_name EXCEPTION;

*-- 定义一个名为exception\_name的异常*

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

*-- 使用if语句，当condition时，引发（raise）exception\_name这个异常*

EXCEPTION

WHEN exception\_name THEN

statement;

*-- 在异常板块内，自定义exception\_name这个异常要做的操作*

END;

SQL

**PL/SQL 触发器**

触发器是存储的程序，在发生以下这些事件时会自动执行或触发：

* 数据库操作(DML)语句(DELETE，INSERT或UPDATE)
* 数据库定义(DDL)语句(CREATE，ALTER或DROP)。
* 数据库操作(SERVERERROR，LOGON，LOGOFF，STARTUP或SHUTDOWN)。

创建触发器的语法是：

CREATE [OR REPLACE ] TRIGGER trigger\_name

*-- 使用trigger\_name创建或替换现有的触发器*

{BEFORE | AFTER | INSTEAD OF }

*-- 指定语句执行之前/之后执行触发器。INSTEAD OF子句用于在视图上创建触发器*

{INSERT [OR] | UPDATE [OR] | DELETE}

*-- 这指定了在哪些DML操作时触发*

[OF col\_name]

*-- 这指定了将要更新的列名称*

ON table\_name

*-- 这指定了与触发器关联的表的名称*

[REFERENCING OLD AS o NEW AS n]

*-- 这允许各种DML语句(如INSERT，UPDATE和DELETE)引用新值和旧值*

[FOR EACH ROW]

*-- 这指定了一个行级别的触发器，即每处理一行就触发一次*

*-- 没有这一句则为表级触发器，每执行一次语句只触发一次*

WHEN (condition)

*-- 这为触发器触发的行提供了一个条件。该子句仅对行级触发器有效*

DECLARE

Declaration-statements

BEGIN

*-- 这里写触发器的具体执行逻辑*

Executable-statements

EXCEPTION

Exception-handling-statements

END;

SQL

比如：

SET SERVEROUTPUT ON SIZE 999999;

CREATE OR REPLACE TRIGGER display\_salary\_changes

*-- 在customers表上创建一个触发器，当工资发生变化时（update）打印出变化值*

BEFORE UPDATE ON customers

FOR EACH ROW

*-- 行级触发器，每一行变化都会触发打印*

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

*-- :OLD和:NEW引用不可用于表级触发器，只能用于行级触发器*

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

SQL

**PL/SQL 包**

包是模式对象，将逻辑上相关的PL/SQL类型，变量和子程序分组。

一个包将有两个强制性的部分：

* 包规范/格式（包头）
* 包体或定义（包体）

**包规范（包头）**

包规范是一个包的对外接口。它声明了可以从包外部引用的类型，变量，常量，异常，游标和子程序。也就是说，一个包头声明了所有我们希望暴露给外部的所有包内元素，一般包括变量，函数，过程。反过来，如果一个包内元素没有在包头中被声明，即使它存在于包体中，它也不能被外部访问。

**包体**

包体具有包规范中声明的各种方法代码和其他私有声明，这些声明对包之外的代码是隐藏的。即，包体定义了包中所有子程序的实现。

**注意！！**

* 包头（规范）和包体（实现）是“分开定义”的结构，不能写在一个create语句中
* 但必须搭配使用，才能完整实现包的功能
* 包规范中声明的变量 是全局共享变量，多个会话共享一个副本
* 包体中声明的变量 是私有变量，仅包体可见

比如：

*-- 第一步：创建包头*

CREATE OR REPLACE PACKAGE my\_pkg AS

PROCEDURE do\_something;

END my\_pkg;

/

*-- 第二步：创建包体*

CREATE OR REPLACE PACKAGE BODY my\_pkg AS

PROCEDURE do\_something IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Doing something...');

END;

END my\_pkg;

/

*-- 第三步：用（.）来使用包内的方法或过程或变量*

BEGIN

my\_pkg.do\_something;

END;

/

**PL/SQL 集合**

集合是具有相同数据类型的有序元素组。 每个元素都由一个唯一的下标来表示它在集合中的位置。PL/SQL提供了三种集合类型：

* 索引表或关联数组
* 嵌套的表
* 可变大小的数组或Varray类型（前文已经讨论过）

| **集合类型** | **元素个数** | **下标类型** | **密集或稀疏** | **在哪创建** | **是否为对象类型属性** |
| --- | --- | --- | --- | --- | --- |
| 关联数组(或索引表) | 无界 | 字符串或整数 | 任意一种 | 只在PL/SQL块中 | No |
| 嵌套表 | 无界 | 整数 | 开始密集，可以变得稀疏 | 在PL/SQL块或模式级别 | Yes |
| 可变大小数组(Varray) | 有界 | 整数 | 总是密集 | 在PL/SQL块或模式级别 | Yes |

**索引表**

索引表(也称为关联数组)是一组键值对。类似于js的对象，每个键都是唯一的，用来定位相应的值。键可以是整数或字符串。

使用以下语法创建索引表，在这里，正在创建一个名为table\_name，以type\_name为类型的索引表，其中的键的数据类型是subscript\_type，关联的值的数据类型是element\_type：

TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;

table\_name type\_name;

SQL

示例：

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);

*-- 定义索引表的类型为salary，值是number类型，键是VARCHAR2类型*

salary\_list salary;

*-- 定义一个salary类型的索引表叫salary\_list*

name VARCHAR2(20);

BEGIN

*-- 将键值对插入索引表，用“索引表名（键）”的形式访问表内元素*

salary\_list('Rajnish') := 62000;

salary\_list('Minakshi') := 75000;

salary\_list('Martin') := 100000;

salary\_list('James') := 78000;

*-- 逐行打印索引表*

name := salary\_list.FIRST;

*-- .first方法拿取索引表的最小键（如果是字符就按字典序（ASCII 顺序）排序）*

*-- 这里就会拿到James*

WHILE name IS NOT null LOOP

dbms\_output.put\_line

('Salary of ' || name || ' is ' || TO\_CHAR(salary\_list(name)));

name := salary\_list.NEXT(name);

*-- .next()方法拿到当前键的下一个键，也是按照排序*

END LOOP;

END;

/

SQL

**嵌套表**

嵌套表就像一个具有任意数量元素的一维数组。但是，嵌套表与数组在以下几个方面不同：

* 数组是一个有声明数量的元素集合，但是一个嵌套表没有。嵌套表的大小可以动态增加。也就是说嵌套表不用声明其大小。
* 数组总是密集的，即它总是具有连续的下标。 嵌套数组最初是密集的，但是当从其中删除元素时，它可能变得稀疏。即可以有空位：1，2，3，10

使用以下语法创建嵌套表：

TYPE type\_name IS TABLE OF element\_type [NOT NULL];

table\_name type\_name;

SQL

声明类似于索引表的声明，只是没有INDEX BY子句。

具体示例：

SET SERVEROUTPUT ON SIZE 99999;

DECLARE

TYPE names\_table IS TABLE OF VARCHAR2(10);

TYPE grades IS TABLE OF INTEGER;

names names\_table;

marks grades;

total integer;

BEGIN

names := names\_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

*-- 直接向嵌套表names内插入元素*

Marks := grades(98, 97, 78, 87, 92);

*-- 直接向嵌套表Marks内插入元素*

total := names.count;

dbms\_output.put\_line('Total '|| total || ' Students');

FOR i IN 1 .. total LOOP

dbms\_output.put\_line('Student:'||names(i)||', Marks:' || marks(i));

end loop;

END;

/

SQL

当上面的代码在SQL提示符下执行时，它会产生以下结果：

Total 5 Students

Student:Kavita, Marks:98

Student:Pritam, Marks:97

Student:Ayan, Marks:78

Student:Rishav, Marks:87

Student:Aziz, Marks:92

PL/SQL 过程已成功完成。

Shell

**常用集合方法：**

| **编号** | **方法** | **目的** |
| --- | --- | --- |
| 1 | EXISTS(n) | 如果集合中的第n个元素存在，则返回TRUE; 否则返回FALSE。 |
| 2 | COUNT | 返回集合当前包含的元素的数量。 |
| 3 | LIMIT | 检查集合的最大容量(大小)。 |
| 4 | FIRST | 返回集合中的最小索引。 |
| 5 | LAST | 返回集合中的最大索引。 |
| 6 | PRIOR(n) | 返回集合中索引n之前的索引编号。 |
| 7 | NEXT(n) | 返回索引n的下一个索引号。 |
| 8 | EXTEND | 追加一个空(null)元素到集合。 |
| 9 | EXTEND(n) | 将n个空(null)元素追加到集合中。 |
| 10 | EXTEND(n,i) | 将第i个元素的n个副本追加到集合中。 |
| 11 | TRIM | 删除一个集合末尾的元素。 |
| 12 | TRIM(n) | 删除集合末尾的n个元素。 |
| 13 | DELETE | 删除集合中的所有元素，将COUNT设置为0。 |
| 14 | DELETE(n) | 使用数字键或嵌套表从关联数组中删除第n个元素。 如果关联数组有一个字符串键，则删除键值对应的元素。 如果n为空，则DELETE(n)不执行任何操作。 |
| 15 | DELETE(m,n) | 从关联数组或嵌套表中移除m..n范围内的所有元素。 如果m大于n，或者m或n为空，则DELETE(m，n)将不执行任何操作。 |

**PL/SQL 事务**

数据库事务是由一个或多个相关SQL语句组成的原子工作单元。它被称为原子操作，因为构成事务的SQL语句带来的数据库修改可以共同提交，即永久化到数据库或从数据库回滚(撤销)。

成功执行的SQL语句和提交的事务不一样。即使成功执行SQL语句，除非提交包含语句的事务，否则可以回滚该语句，并且可以撤消语句所做的所有更改。

**开始和结束事务**

事务有开始和结束。当发生以下事件之一时，事务即开始：

* 连接到数据库后执行第一个SQL语句。
* 在事务完成后发出的每个新的SQL语句。

事务在下列事件之一发生时结束：

* 发出了COMMIT或ROLLBACK语句。
* 发出DDL语句，例如：CREATE TABLE语句; 因为在这种情况下，自动执行COMMIT。
* 发布DCL语句，如:GRANT声明; 因为在这种情况下，自动执行COMMIT。
* 用户从数据库断开连接。
* 用户通过发出EXIT命令从SQL \* PLUS退出，COMMIT自动执行。
* SQL \* Plus异常终止，会自动执行ROLLBACK。
* DML语句失败; 在这种情况下，会自动执行ROLLBACK来撤消该DML语句。
* 用户通过发出EXIT命令从SQL \* PLUS退出，COMMIT自动执行。
* SQL \* Plus异常终止，会自动执行ROLLBACK。
* DML语句失败; 在这种情况下，会自动执行ROLLBACK来撤消该DML语句。

**回滚事务**

使用ROLLBACK命令可以撤销还未COMMIT的所有更改。

ROLLBACK [TO SAVEPOINT <savepoint\_name>];

SQL

**保存点 - Savepoints**

可以将代码分为数个存档点，方便我们进行回滚，语法如下：

SAVEPOINT < savepoint\_name >;

SQL

例如：

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

SAVEPOINT sav1;

UPDATE CUSTOMERS

SET SALARY = SALARY + 1000;

ROLLBACK TO sav1;

*-- 此语句回滚到保存点sav1，即撤销第一次update操作*

UPDATE CUSTOMERS

SET SALARY = SALARY + 1000

WHERE ID = 7;

COMMIT;

*-- commit之后就无法再回滚了*

SQL