# YashanDB 开发手册

*v23.2.2.0*

2024年5月


深圳计算科学研究院

深圳崖山科技有限公司

# PL参考手册

PL全称为Procedural Language/Structured Query Language，过程化SQL语言，它是一种建立在普通SQL语言之上的编程语言。

面向读者：

PL参考手册适用于使用PL语言进行开发的所有角色，包括：

- 基于数据库PL语言的开发者
- 使用数据库PL语言的分析者
- 数据库管理员

要有效地使用本文档，需要具备以下方面的知识储备：

- 基本编程能力，掌握变量、控制、函数调用等概念
- 结构化查询语言
- YashanDB数据库的基础知识，如系统视图

手册约定：

- 符号 说明
- [] 表示包含一个或多个可选项。
- {} 表示包含两个及以上候选，必须在其中选取一个。
- | 分割中括号或者花括号中的两个或两个以上选项。
- ... 表示其之前的元素可以被重复。
- 大写 表示系统关键字。
- 小写 表示用户标识符或者需要用户输入的子句缩写。

关于本手册中使用的示例说明：

- 示例中的SQL、PL语句在yasql工具中运行，并以yasql工具的输出结果作为样例进行展示。
- 示例中使用DBMS_OUTPUT.PUT_LINE语句向yasql工具端打印输出，为达到输出效果，需要保证在yasql工具里已运行set serveroutput on来打开控制数据开关，详见工具手册yasql基本功能使用指导。
- 系统对浮点类型及NUMBER类型数据输出时的显示宽度默认为10，可通过SET NUMWIDTH来调节显示宽度。
- 关于日期的显示格式：为尽量模拟实际业务，本手册示例库的日期格式被设置为'YYYY-MM-DD HH24:MI:SS'，如DATE_FORMAT配置参数不是按此格式设置，所看到的日期格式将与本手册不一致。
- 示例使用与SQL参考手册中一致的样例表。

> **Note**：
> 没有特殊说明规则的话，PL语句不输入参数或输入NULL，表示不对这个参数做任何处理。

# PL概述

PL语言是SQL语言在YashanDB数据库上的过程扩展语言，是一种可移植的高性能可编程语言，本文将介绍PL语言的优点及主要特性。

## PL的优势

PL相较于其他编程语言具有以下优点：

- SQL集成
- 支持可编程
- 高性能
- 可移植性

### SQL集成

PL语言与结构化查询语言SQL的结合非常紧密，具体表现在以下方面：

- PL语言中允许静态SQL操作，即直接使用所有的DQL、DML数据操作，事务控制语句，以及所有内置函数、高级包函数、运算符和伪列。
- PL语言中允许通过动态SQL方式进行所有SQL操作。
- PL语言完全支持SQL中定义的所有数据类型。
- PL语言中支持游标变量，提供了灵活的游标OPEN、FETCH、CLOSE、赋值、入参、出参、返回给客户端等操作。
- PL语言中，可以为变量的数据类型指定数据库表的列或行的数据类型，而无需显式指定该数据类型。
- PL语言中，运行DQL查询时将会一次处理查询结果集的一行。
- PL语言可以定位自定义函数，该函数可以在DQL语句中直接使用。
- PL对SQL的操作产生异常时，均可以通过异常模块的编程进行捕获。

### 支持可编程

结构化查询语言SQL与数据库的结合非常紧密，主要用于描述如何访问和处理数据库，包括数据的增删改查操作。但SQL语言的可编程并不强，PL语言则是针对该弱点进行扩展的语言。

- PL语言支持变量声明、赋值、缺省表达式，实参和形参等特点。
- PL语言中的变量除了普通数据类型声明的标量形式，还可以支持普通数据类型或者用户自定义类型UDT定义的数组、OBJECT、NEST TABLE等向量形式。
- PL语言中提供了循环、条件、跳转、空行等可编程语言逻辑。
- PL语言提供了异常捕获模块。
- PL语言提供了存储过程、自定义函数、触发器、匿名块等多种数据库对象形态，提供不同的使用场景。

### 高性能

PL语言通过可编程逻辑和SQL集成，可以直接在数据库实现应用逻辑操作，从而显著的减少应用程序和数据库之间的结果集交互流量。

- 通过绑定变量的方式，在PL语言中使用DML/DQL操作，PL编译器会将SQL语句中出现的PL变量转换为绑定变量。
- YashanDB数据库在每次运行相同的SQL语句时，将复用SQL语句的编译缓存，从而提高性能。
- 在PL语言中使用动态SQL时，不会自动创建绑定变量，但可以通过EXECUTE IMMEDIATE语句中的USING子句显式指定绑定参数。
- PL的程序在首次调用将进行编译，编译完成进入PL POOL缓存池，后续复用，从而提高性能。

### 可移植性

PL语言是用于数据库开发的可移植标准语言，可以在运行数据库的任何操作系统和平台上使用PL语言。

## PL的主要特点

PL语言结合了SQL语言的数据操作能力和过程语言的处理能力，主要包括以下特点：

- 语句块
- 变量和常量
- PL数据类型
- 丰富的语句形式
- 丰富的对象形态
- 异常捕获模块

## 语句块

PL语言的基本单元是语句块，它对相关的声明和语句进行分组。

语句块可由不同关键字划分为不同类别，具体含义如下：

1. DECLARE：声明部分。
2. BEGIN - END：可执行部分（必需）。
3. EXCEPTION：异常处理部分。

YashanDB中语句块支持通过使用标签的方式来获取别名，从而使该语句块具有一次性使用的特性，同时语句块也支持嵌套。

## 变量和常量

PL语句块的声明部分允许声明变量和常量，然后在可以使用表达式的任何地方使用它们。

变量和常量的区别在于程序运行时，变量的值可以进行更改，而常量的值不能。

当多个语句块嵌套时，如果发生变量名冲突，缺省情况将以最近一层语句块声明的变量为准。

PL支持如下两种引用变量的使用：

- PL可通过%ROWTYPE类型将声明的变量视为一个RECORD，且该RECORD的成员来自于%ROWTYPE所引用对象的成员。
- PL可通过指定%TYPE类型使变量的数据类型与其他某个变量或列字段的数据类型相同。

## PL数据类型

PL支持使用所有SQL数据类型进行声明，可声明成普通标量，或者声明成数组、OBJECT、NEST TABLE等向量，还支持通过自定义类型，以及类型级联嵌套进行增强。

## 丰富的语句形式

- PL语言的控制语句提供以下能力：条件选择语句，可针对不同的数据值运行不同的语句；循环语句，可使用一系列不同的数据值重复相同的语句；顺序控制语句，允许转到指定的、标记的语句。
- PL语言的静态SQL、动态SQL，隐式游标属性能力。
- PL语言的游标相关语句，以及游标属性能力。
- PL支持表达式、运算符、内置函数、内置高级包、伪列等的直接调用能力。
- PL的系统异常、自定义异常的声明和使用。

## 丰富的对象形态

- 将PL语句块单独执行即为匿名块。
- 支持定义PROCEDURE，并提供给PL语句块执行，不直接返回结果，可以提供出入参。
- 支持定义自定义函数，包括PL语言的自定义函数、外置JAVA语言的自定义函数和外置C语言的自定义函数。自定义函数可以提供给SQL语言直接调用执行，也可以提供给PL语句块执行，会直接返回结果。
- 支持定义TRIGGER，目前提供在DML语句执行前后进行语句级或者行级触发。
- 支持定义自定义高级包，这是一个模式对象，它对逻辑相关的PL类型、变量、常量、存储过程、自定义函数、游标和异常进行分组。包被存储在数据库中，可以被直接访问使用。数据库本身也提供了很多高级包，具体请查询内置高级包章节。

## 异常捕获模块

PL的异常捕获模块主要用于检测和处理错误。

当SQL操作发生错误时，同时会造成PL中出现异常，此时异常捕获模块会检测该错误。

# PL语言基础

PL的词法单元是其最小的的单个组件，包括：

- 分隔符
- 标识符
- 字面量
- 编译指示
- 注释

## 分隔符

分隔符用于分割词法元素，可以是一个字符或多个字符的组合。以下是PL中包含的分隔符：

| 分隔符 | 含义 |
| --- | --- |
| + | 加法运算符 |
| := | 赋值运算符 |
| % | 属性标识 |
| ' | 字符串分隔符 |
| . | 对象分隔符 |
| \|\| | 连接运算符 |
| / | 除法运算符 |
| ( | 左括号 |
| ) | 右括号 |
| , | 逗号分隔符 |
| << | 左标签分隔符 |
| >> | 右标签分隔符 |
| /* | 多行注释开始符 |
| */ | 多行注释结束符 |
| * | 乘法运算符 |
| " | 双引号 |
| .. | 范围运算符 |
| = | 等号 |
| <> | 不等号 |
| != | 不等号 |
| < | 小于号 |
| > | 大于号 |
| <= | 小于等于号 |
| >= | 大于等于号 |
| -- | 单行注释符 |
| ; | 分号 |

| 分隔符 | 含义 |
| --- | --- |
| - | 减法运算符 |

## 标识符

标识符用于标识PL语法单元，包含：

- 字面量
- 游标
- 异常
- 关键字
- 标签
- 包
- 保留关键字
- 子过程体
- 类型
- 变量

详细见标识符。

## 字面量

字面量的意思是不变的值，它以字符串的形式直接出现在SQL和PL语句中，详细见字面量。

## 编译指令

编译指令（PRAGMA）用于指示编译阶段的指令。

## 注释

注释用于增强PL程序的可读性，在编译过程会忽略注释。

**单行注释**

在注释文字前用”--“标识。

**多行段落注释**

注释文字使用”/.../“包含。

# 变量

变量是用来存储数据及在程序中传递数据值的一个容器。在PL过程体中使用的变量必须先进行声明，且某些类型的变量可在声明的同时为其赋一个初始值。

## 变量声明

一个声明用于定义一个对象，指定数据类型，分配存储空间，并命名存储位置，以便后续引用。声明必须出现在块、子程序或包的声明部分。

变量可以被显式声明和隐式声明：

- **显示声明变量**

对于过程体中使用的全局变量，必须在BEGIN关键字前显示声明。

- **隐式声明变量**

对于PL语句中的FOR Statement，其counter变量即为隐式声明的局部变量，可直接使用而不需要在BEGIN关键字前显示声明。

## 变量初始化和赋值

在声明变量时，可同时对其进行初始化赋值，例如：`var INT := 0;`。

### 赋值语句赋值

将常数/变量、或常数/变量的四则运算组成的表达式，通过`:=`赋值给变量。

示例

```
DECLARE
    a INT;
BEGIN
    a := 1;
END;
/
```

### SQL赋值

将SQL（包括静态SQL和动态SQL）的结果集，通过INTO赋值给变量，具体请参考DML Statement、EXECUTE Statement描述。

### 游标赋值

通过FETCH语句对游标取值并赋给变量，具体请参考FETCH Statement。

### 出参赋值

通过将变量作为OUT或IN OUT参数传递给存储过程或自定义函数，通过给出参赋值的方式给变量赋值，具体请参考形参和实参中的OUT参数和IN OUT参数。

## 变量类型

隐式声明变量时其类型已知且由系统指定，显式声明则需在定义时明确指定变量类型，不同类型的变量声明格式不同，YashanDB PL包括的变量类型有：

- **普通变量**：

普通变量的声明格式为："变量名称 [CONSTANT] 数据类型[NOT NULL] [:=初始值];"。

其中，数据类型同SQL中定义的普通标量数据类型。

当数据类型为字符型时，须同时指定其Size属性，例如VARCHAR(10)，但输入CHAR默认等同于CHAR(1)。CHAR和VARCHAR中可指定的最大Size为32000。Size后可选长度单位CHAR和BYTE，分别为字符长度单位和字节长度单位。

- 对于VARCHAR（SIZE[CHAR|BYTE]），指定不超过32000个字节的初始值。若不指定长度单位，默认使用BYTE。

- 对于CHAR（SIZE[CHAR|BYTE]），使用CHAR长度单位时初始值会优先按照字符数进行空格补齐，最大补齐至32000个字节；使用BYTE长度单位时初始值按照字节数进行空格补齐，最大补齐至32000个字节。若不指定长度单位，默认使用BYTE。

当指定NOT NULL属性时，须同时为变量指定不为NULL的初始值。 当指定CONSTANT属性时，表示所声明的变量为一个常量，常量在声明时必须赋值且之后不能再赋值，具体见常量章节介绍。

- **游标**：

游标是一种特殊的PL变量，具体可分为显式与动态两种类型，其声明和使用方式见游标文档说明。

- **RECORD**：

一个RECORD由一组数据项组成，每个数据项有自己的名称和数据类型，例如一个表的多个列项可以组成一个RECORD，其声明和使用方式见RECORD文档说明。

- **VARRAY**：

集合变量，用于定义数组，可以为复合（嵌套）数组。

- **NESTED TABLE**：

集合变量，用于定义嵌套表。

- **%TYPE**：

引用变量，用于对某个变量或某个列字段类型的自动引用，使声明变量的数据类型与该变量或列字段的数据类型相同。

- **%ROWTYPE**：

引用变量，用于对某个表或视图结构类型的自动引用，使声明变量为一个包含该表或视图所有列项的RECORD。

- **EXCEPTION**：

自定义异常变量，该变量只能被用于PL的异常处理机制中。

# 常量

将变量声明为一个常量的语法为：

```
变量名 CONSTANT datatype [NOT NULL] (:=|DEFAULT) expression;
```

指定CONSTANT即表示声明的是一个常量，此时必须同时对其进行初始化赋值，且在过程体中不允许对其赋值。

expression可以为NULL、数值、字符串、及另一个已声明的变量（不要求变量初始化赋值，此时等同于NULL）。

如在声明常量时指定了NOT NULL属性，则不允许为其初始化赋值NULL。

定义一个RECORD时，其成员不能定义为常量。

示例

```
DECLARE
a INT;
a1 INT := 3;
b CONSTANT INT := a;
c CONSTANT INT := NULL;
d CONSTANT INT NOT NULL DEFAULT a1;
BEGIN
a := 4;
a1 := 4;
DBMS_OUTPUT.PUT_LINE('b is '||b);
DBMS_OUTPUT.PUT_LINE('c is '||c);
DBMS_OUTPUT.PUT_LINE('d is '||d);
END;
/

--result
b is
c is
d is 3
```

# 游标

游标（Cursor）被用于指向一条SQL语句一行或多行的结果集，提供在结果集中分行浏览数据的能力。

YashanDB中可由用户声明和使用的游标分为如下几种：

- 显式游标：使用CURSOR...IS命令定义的游标，在其定义时静态绑定了一条SQL语句，不可更改与赋值。
- 动态游标：使用TYPE...IS REF CURSOR定义的游标变量，可以与不同的SQL语句动态绑定（某一时刻只能与一条SQL语句绑定），只需要这些SQL语句的返回类型与游标变量定义的类型兼容即可。
- sys_refcursor：系统预定义的动态游标类型，可直接调用，使用方法与动态游标一致。

分布式部署中不可使用游标变量，但支持使用隐式游标相关属性。

YashanDB在同一时间可打开的游标数目限制为300个。

游标所绑定的SQL中不能有FOR UPDATE语句。

## 显式游标

通过游标定义语句实现对显式游标的声明，之后即可在过程体中执行打开游标、推进游标、关闭游标等操作。

**cursor_definition::=**

```
syntax::= CURSOR cursor ["(" (cursor_parameters) {"," (cursor_parameters)} ")"] [RETURN rowtype] IS select_statement ";"
```

**cursor_parameters**

定义显式游标的参数，该参数用于给绑定的SQL语句提供变量输入，参数定义格式为：

*参数名称 数据类型 [缺省值]*

其中，数据类型里不应该包含长度、精度等属性。

可以为游标的参数定义一个缺省值，之后，允许在打开游标时不输入参数值，系统将会以此缺省值作为参数值传入，当仍输入参数值时，则会以输入的值作为参数值传入。

当游标的参数为多个时，应该将定义缺省值的参数放在最后面，但在使用"=>"传入参数值时，无需此顺序要求。

**RETURN rowtype**

定义显式游标的返回值，必须指定为一个RECORD类型变量，且其列项必须与绑定SQL语句的查询列项相匹配。

RETURN rowtype语句可省略，则系统将隐式地生成一个返回值变量，其列项即为绑定SQL语句的查询列项。

**select_statement**

SELECT查询语句，但不可包含FOR UPDATE，且不可指定占位符（绑定参数）。

示例（单机、共享集群部署）

```
DECLARE
  CURSOR cur(c1 CHAR) IS
  SELECT area_no, area_name FROM area WHERE area_no=c1;
  TYPE record1 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20)
  );
  rec record1;
BEGIN
  OPEN cur(c1=>'01');
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2);
  CLOSE cur;
END;
/
```

```
--result
01 华东


-- 游标for循环
DECLARE
CURSOR  emp_cur IS SELECT * FROM employees;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR v_emp_rec IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.employee_no || '     ' || v_emp_rec.employee_name);
    END LOOP;
END;
/

--result
EMPNO    ENAME
-----    -------
0101000001     Mask
0101000002     John
0201010011     Anna
0201008003     Jack
0201008004     Jim
```

## 动态游标

动态游标要求首先定义一个引用游标类型，并按此类型声明一个游标变量，之后即可在过程体中执行打开游标、游标取值、关闭游标等操作。

**ref_cursor_type_definition::=**

```
syntax::= TYPE type IS REF CURSOR [RETURN ((table_or_view|cursor|cursor_variable) "%ROWTYPE"|
                record "%TYPE"|
                record_type)] ";"
```

**cursor_variable_declaration::=**

```
syntax::= cursor_variable type ";"
```

**RETURN子句**

RETURN子句显示地定义了动态游标所持有的返回值，该返回值必须通过一个RECORD类型来记录，包括：

- %ROWTYPE引用的表、视图、显式游标或游标变量（必须为一个显式定义了返回值的游标变量）。
- %TYPE引用的一个已定义的RECORD类型变量。
- 一个已定义的RECORD类型变量。

打开游标的操作中，若被打开的SQL语句的查询列与此处RECORD所记录的成员无法进行隐式类型转换时，将无法对该SQL语句赋值给游标变量。

本语句可省略，此时声明的是一个没有指定结果集的游标变量，可以实现绑定不同的SQL语句且返回不同类型的RECORD。

示例（单机、共享集群部署）

```
--未显式定义返回值的动态游标声明
DECLARE
  TYPE cursor IS REF CURSOR;
  cur cursor;
  TYPE record1 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20)
  );
  rec1 record1;
  TYPE record2 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
```

```
    c3 INT
  );
  rec2 record2;
BEGIN
  OPEN cur FOR SELECT area_no, area_name FROM area WHERE area_no='01';
  FETCH cur INTO rec1;
  DBMS_OUTPUT.PUT_LINE(rec1.c1 ||' '||rec1.c2);
  CLOSE cur;
  OPEN cur FOR SELECT area_no, area_name, LENGTH(area_name) FROM area WHERE area_no='01';
  FETCH cur INTO rec2;
  DBMS_OUTPUT.PUT_LINE(rec2.c1 ||' '||rec2.c2||' '||rec2.c3);
  CLOSE cur;
END;
/
--result
01 华东
01 华东 2

--显式定义返回值的动态游标声明
DECLARE
  TYPE cursor1 IS REF CURSOR RETURN area%ROWTYPE;
  cur1 cursor1;
  TYPE cursor2 IS REF CURSOR RETURN cur1%ROWTYPE;
  cur2 cursor2;
  TYPE record IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 VARCHAR(20)
  );
  rec record;
BEGIN
  OPEN cur2 FOR SELECT * FROM area WHERE area_no='01';
  FETCH cur2 INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  CLOSE cur2;
END;
/
--result
01 华东 Shanghai
```

## sys_refcursor

sys_refcursor为系统预定义的不指定返回值的动态游标，可被直接用于游标变量的声明，其用法与动态游标一致。

**游标参数**

基于系统预定义的sys_refcursor，可以将游标作为PL参数使用：

- 作为存储过程或自定义函数的IN参数。
- 作为存储过程的OUT参数。
- 作为自定义函数的返回值。

示例（单机、共享集群部署）

```
CREATE OR REPLACE PROCEDURE getArea(cur IN OUT SYS_REFCURSOR) AS
BEGIN
  OPEN cur FOR SELECT * FROM area WHERE area_no='01';
END;
/

CREATE OR REPLACE FUNCTION getArea_func RETURN SYS_REFCURSOR AS
  cur SYS_REFCURSOR;
BEGIN
  OPEN cur FOR SELECT * FROM area WHERE area_no='02';
  RETURN cur;
END;
/
```

```
DECLARE
  TYPE record IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 VARCHAR(20)
  );
  rec record;
  cur SYS_REFCURSOR;
BEGIN
  cur := getArea_func;
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  CLOSE cur;
  getArea(cur);
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  CLOSE cur;
END;
/

--result
02 华西 Chengdu
01 华东 Shanghai
```

## 游标间赋值

YashanDB支持直接将一个游标变量赋值给另一个游标变量，规则如下：

- 显式游标无法作为任何赋值表达式的左值或右值。
- 两个返回值列项相同的动态游标可相互赋值。
- 无返回值的游标作为左值时可被任意非显式游标赋值。
- 有返回值的游标作为左值时将会判断右值游标的返回值列项，若右值游标未绑定SQL语句，或者其返回值列项与左值游标一致，则可赋值。

示例（单机、共享集群部署）

```
DECLARE
  cur0 SYS_REFCURSOR;
  TYPE cursor IS REF CURSOR RETURN area%ROWTYPE;
  cur cursor;
  TYPE record IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 VARCHAR(20)
  );
  rec record;
BEGIN
  OPEN cur FOR SELECT * FROM area WHERE area_no='01';
  cur0 := cur;
  FETCH cur0 INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  CLOSE cur;
END;
/
--result
01 华东 Shanghai
```

## 游标的属性

游标有四个属性，如下：

| 属性 | 返回值类型 | 作用 |
| --- | --- | --- |
| %isopen | 布尔型 | 判断游标是否打开 |
| %found | 布尔型 | 判断游标是否获取到值 |

| 属性 | 返回值类型 | 作用 |
| --- | --- | --- |
| %notfound | 布尔型 | 判断游标是否没有获取到值 |
| %rowcount | BIGINT | 当前成功执行的数据行数 |

上述属性只能在过程性语句中使用，不能使用在SQL语句中，使用格式如下：

*cursor_name%isopen*、*cursor_name%found*、*cursor_name%notfound*、*cursor_name%rowcount*

其中cursor_name为游标的名称，使用示例可查看FETCH Statement。

**隐式游标属性**

在过程体中执行某条SQL语句时，系统为其生成了一个隐式的游标，隐式游标无需用户声明和管理，但用户可以通过上述游标属性来操作其结果集，使用格式如下：

*SQL%isopen*、*SQL%found*、*SQL%notfound*、*SQL%rowcount*

示例

```
BEGIN
  UPDATE area SET area_name='cursor example';
  IF SQL%found THEN
    DBMS_OUTPUT.PUT_LINE('Total '||SQL%rowcount||' lines are deleted.');
  END IF;
  ROLLBACK;
END;
/

--result
Total 5 lines are deleted.
```

# 引用变量

本文档描述%TYPE和%ROWTYPE两种引用变量类型。

## %TYPE

在声明变量时，通过指定%TYPE可以使变量的数据类型与其他某个变量或列字段的数据类型相同，当变量用于存储其他某个变量或列字段的值时，使用%TYPE可以在声明时不必知道对方的数据类型直接进行引用，且如果对方的数据类型发生变更时，无需改动声明。

除用于声明变量外，%TYPE还可以作为PROCEDURE的参数，FUNCTION的参数或者返回值。

当%TYPE引用的对象为列字段时，将只引用该列字段的数据类型，其他如NOT NULL等约束项不会被继承，且在列字段上定义的DEFAULT值也不会被继承。

当%TYPE引用的对象为其他某个变量时，该变量上的NOT NULL属性将被继承，但初始值不会被继承，基于此规则，如果引用的变量指定了NOT NULL，则必须为%TYPE变量定义初始值。

**变量类型**

可以被%TYPE引用的变量包括：

- 普通变量
- 游标
- RECORD
- RECORD的成员，为record_name.member_name形式
- label与上述变量的结合，为label.变量名形式

声明%TYPE类型变量的格式为：

```
变量名 (被引用的变量名|表名.列名)%TYPE[:=初始值];
```

**匹配规则**

对于上述中出现的label.变量名、record_name.member_name和表名.列名三种字面表现相同的形式，系统定义如下匹配规则以确定%TYPE的引用对象：

- 按record_name.member_name>label.变量名>表名.列名的优先级进行匹配。
- 当"."前面的item匹配上时，上条优先级规则不再有效，而是只对本种形式进行匹配，此时如"."后面的item未匹配成功，则系统报错。

示例

```
DROP TABLE IF EXISTS table_type;
CREATE TABLE table_type(a VARCHAR(10));
INSERT INTO table_type VALUES('helloworld');
COMMIT;

--tb_type与tb_recode匹配，但a与aa不匹配，此时报错而不执行label.变量名匹配
<<tb_type>>
DECLARE
TYPE tb_record IS RECORD (aa CHAR(12));
a VARCHAR(11);
tb_type tb_record;
b tb_type.a%TYPE;
BEGIN
b := 'helloworld';
DBMS_OUTPUT.PUT_LINE(b);
END;
/
YAS-04253 PL/SQL compiling errors:
[7:3] YAS-05246 invalid tb_type.a%type
[9:1] YAS-04243 invalid identifier "B"
[10:22] YAS-04243 invalid identifier "B"
```

## %ROWTYPE

通过%ROWTYPE声明的变量为一个RECORD，且该RECORD的成员来自于%ROWTYPE所引用对象的成员，包括成员的名称和类型。

可以被%ROWTYPE引用的对象有：

- 表/视图，可通过别名引用
- 游标

其中，引用游标表示引用的是该游标的返回值RECORD，包括显式定义的返回值，以及未显式定义返回值时，系统隐式生成的返回值（即游标绑定的SQL语句的列项）。

RECORD的成员为被引用对象所拥有的全部列或成员，无法进行部分引用。

示例

```
DECLARE
a area%ROWTYPE;
TYPE cursor IS REF CURSOR RETURN area%ROWTYPE;
cur cursor;
ano_cur cur%ROWTYPE;
BEGIN
a.area_no := '88';
a.area_name := 'ref column';
DBMS_OUTPUT.PUT_LINE('first record:' || a.area_no || ',' || a.area_name);
OPEN cur FOR SELECT * FROM area;
FETCH cur INTO ano_cur;
DBMS_OUTPUT.PUT_LINE('third record:' || ano_cur.area_no || ',' || ano_cur.area_name);
CLOSE cur;
END;
/

--output
first record:88,ref column
third record:01,华东
```

# RECORD

在声明RECORD类型之前，需要先进行RECORD类型的结构定义，包括用户自定义和系统隐形定义：

- 用户自定义：通过TYPE ... IS RECORD语法显式定义REORD的结构成员信息。
- %TYPE：将某个变量声明为引用变量%TYPE类型时，如被引用的变量为一个RECORD或游标，则系统自动继承被引用变量的结构定义，声明的变量为一个RECORD集合变量。
- %ROWTYPE：将某个变量声明为引用变量%ROWTYPE类型时，系统自动继承被引用对象的行结构定义，声明的变量为一个RECORD集合变量。

## 用户自定义RECORD

用户自定义RECORD需要显式指定RECORD的类型名称、成员名称、成员数据类型信息。其中数据类型不能定义为游标类型。

定义语法如下：

```
syntax::= TYPE record_type IS RECORD "(" (member_name (dataType|recodeType) [":=" default_value])
{"," (member_name (dataType|recodeType) [":=" default_value])} ")" ";"
```

RECORD变量声明语法如下：

```
syntax::= variable_name record_type ";"
```

**dataType**

PL中定义的用户自定义类型和SQL中定义的普通标量数据类型。当数据类型为字符型时，须同时指定其Size属性，例如VARCHAR(10)，但输入CHAR默认等同于CHAR(1)；可指定的最大Size为32000。

**recodeType**

在一个RECORD结构中嵌套另一个RECORD，YashanDB支持最多16层嵌套（当用户自定义RECORD前使用了Label标签时，RECORD最多嵌套15层）。嵌套的RECORD不能定义为引用游标%TYPE类型。

**default_value**

为成员变量定义一个默认值。

## RECORD赋值

### 通过成员变量赋值

通过"."操作符访问RECORD变量的member成员，对成员分别赋值，或通过对象初始化方式对成员批量赋值。

示例

```
DECLARE
TYPE name IS RECORD (
first VARCHAR(20),
last VARCHAR(20) := 'Smith'
);
name1 name;
name2 name;
name3 name;
BEGIN
name1.first := 'Jane';
name1.last := 'Smith';
DBMS_OUTPUT.PUT_LINE('name1: ' || name1.first || ' ' || name1.last);
DBMS_OUTPUT.PUT_LINE('name2: ' || name2.first || ' ' || name2.last);
--初始化方式
name3 := name('Lisa','Stone');
DBMS_OUTPUT.PUT_LINE('name3: ' || name3.first || ' ' || name3.last);
END;
/
```

```
--result
name1: Jane Smith
name2:  Smith
name3: Lisa Stone
```

## RECORD变量间赋值

两个RECORD变量间的赋值规则为：

- 源和目标均为用户自定义的RECORD类型时：只能在声明为同一个recode_type的变量间赋值。
- 源和目标其中一个为系统隐形定义的RECORD类型时，只有两边的成员数量和数据类型均一致时，才可以赋值。其中数据类型一致的含义为：
  - 显式的一致，即系统不进行隐式的数据类型转换。
  - 不要求精度、长度等属性一致。

示例

```
--如下不能赋值
DECLARE
TYPE name_1 IS RECORD (
first VARCHAR(20),
last VARCHAR(20) := 'Smith'
);
TYPE name_2 IS RECORD (
first VARCHAR(20),
last VARCHAR(20) := 'Smith'
);
name1 name_1;
name2 name_2;
BEGIN
name1 := name2;
END;
/
YAS-04253 PL/SQL compiling errors:
[13:10] YAS-00014 illegal conversion from NAME_2 to NAME_1

CREATE TABLE name_1 (c1 NUMBER, c2 CHAR(4));
--如下可以赋值
DECLARE
TYPE name_1 IS RECORD (
first NUMBER(6),
last VARCHAR(40)
);
name1 name_1;
name2 name_1%ROWTYPE;
BEGIN
name1 := name2;
END;
/
--如下不能赋值
DECLARE
TYPE name_1 IS RECORD (
first INT,
last CHAR(4)
);
name1 name_1;
name2 name_1%ROWTYPE;
BEGIN
name1 := name2;
END;
/
YAS-04253 PL/SQL compiling errors:
[9:10] YAS-05286 illegal conversion
```

## 通过SQL语句赋值

通过SQL语句执行RECORD变量的赋值操作（作为源或目标）时，要求源和目标的结构成员一致，且数据类型兼容，即能满足系统的隐式数据类型转换。

### INSERT语句

通过INSERT INTO语句将一个RECORD变量插入到表中，此时只能对表的所有列项整体插入，不能分开列项插入，且不支持returning_clause。

示例

```
DECLARE
TYPE area_type IS RECORD (
area_no VARCHAR(5),
area_name VARCHAR(10),
dhq VARCHAR(5)
);
a area_type;
BEGIN
a.area_no := '09';
a.area_name := 'example';
a.dhq := 'exa';
INSERT INTO area VALUES a;
END;
/
```

### SELECT语句

通过SELECT...INTO语句将查询结果赋值给一个RECORD变量，此时只能将SELECT后的列项整体赋给一个RECORD变量，不能分开赋值，且查询结果必须为单行记录。

示例

```
DECLARE
TYPE area_type IS RECORD (
area_no NUMBER,
area_name VARCHAR(10)
);
a area_type;
BEGIN
SELECT area_no,area_name INTO a
FROM area
WHERE area_no = '09';
DBMS_OUTPUT.PUT_LINE(a.area_no || ',' || a.area_name);
END;
/

--result
9,example
```

# 集合变量

YashanDB包含如下两种集合变量：

- VARRAY：可变长数组，一个数组包含了一组相同类型的成员，这些成员在数组里有序存放。
- NESTED TABLE：嵌套表，一个嵌套表包含了多行的相同类型成员，像物理表记录存储一样，这些行成员采取了无序存放的方式。

数组与嵌套表的显著区别为，数组在定义时必须声明其最大长度，实际长度（即成员个数）在此范围内动态改变；而嵌套表在资源允许范围内不存在最大长度限制，实际长度（即行成员个数）可一直动态扩展。

此外，使用嵌套表可以很方便地实现多维数组的定义和操作。数组或嵌套表元素的数据类型可以是除游标类型外的任何PL数据类型。

## 集合变量声明

声明集合变量前需进行集合类型的定义，除可以在过程体内部直接定义集合类型外，还可以作为UDT在过程体外部定义。在过程体内部定义的语法为：

```
varray_definition::=
TYPE varray_type IS (VARRAY|([VARYING] ARRAY)) "(" size_limit ")" OF datatype [NOT NULL]

nested_table_definition::=
TYPE nested_table_type IS TABLE OF ["("] datatype [NOT NULL] [")"]
```

对集合变量的声明语法为：

```
syntax::= variable_name (varray_type|nested_table_type) ";"
```

**size_limit**

在声明数组类型时，必须指定数组成员的个数上限，值范围[1,int32]。

**datatype**

成员的数据类型，可指定为PL支持的所有类型，或UDT。当数据类型为字符型时，须同时指定其Size属性，例如VARCHAR(10)，但输入CHAR默认等同于CHAR(1)；可指定的最大Size为32000。

**NOT NULL**

使用NOT NULL指定集合的成员不能设置为NULL，省略时，默认集合中的成员可以设置为NULL。

## 集合变量赋值

集合变量赋值包括初始化赋值和过程中赋值。

1.通过如下格式为集合成员赋值：

```
variable_name := (varray_type|nested_table_type) (expr1,expr2,...);

variable_name(index) := expr;
```

当成员类型为普通标量类型时，expr为一个YashanDB认可的通用表达式，否则为该成员类型所规定的赋值格式。

2.可以将一个已初始化值的集合变量赋值给另一个集合变量。

3.可以通过数组相关内置函数对数组进行赋值，例如STRING_TO_ARRAY、ARRAY_APPEND、ARRAY_REMOVE、ARRAY_REPLACE等。

## 集合变量访问

一个已被赋值的集合变量可以通过如下语法访问：

```
syntax::= (varray_type|nested_table_type) "(" expr ")"
```

expr为一个YashanDB认可的通用表达式，其结果必须为一个数字，表示数组内索引。数组内索引以1为首位。

示例

```
CREATE OR REPLACE TYPE udt_object FORCE AS OBJECT (
    area_no CHAR(2),
    MAP MEMBER FUNCTION showArea RETURN VARCHAR
) NOT FINAL;
/
CREATE OR REPLACE TYPE BODY udt_object AS
    MAP MEMBER FUNCTION showArea RETURN VARCHAR AS
    BEGIN
        RETURN self.area_no;
    END;
END;
/

--通过VARRAY定义单维+嵌套数组并访问
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    v_char1 char_array := char_array('1',TO_CHAR(2),'');
    v_char2 char_array := v_char1;
    TYPE array_array IS ARRAY(3) OF char_array;
    v_array array_array;
    TYPE udt_array IS ARRAY(3) OF udt_object;
    v_udt udt_array;
    TYPE rec IS RECORD(area udt_object, branch CHAR(4));
    TYPE rec_array IS ARRAY(3) OF rec;
    v_rec rec_array;
BEGIN
    DBMS_OUTPUT.PUT_LINE('char array:'||v_char2(1)||v_char2(TO_NUMBER('2'))||v_char2(3));
    v_array := array_array(v_char1,v_char2);
    DBMS_OUTPUT.PUT_LINE('array array:'||v_array(1)(1)||v_array(1)(2)||v_array(2)(1)||v_array(2)(2));
    v_udt := udt_array(udt_object('01'),udt_object('02'));
    DBMS_OUTPUT.PUT_LINE('udt array:'||v_udt(1).area_no||' '||v_udt(2).area_no);
    v_rec := rec_array(rec(udt_object('03'),'0301'),rec(udt_object('04'),'0401'));
    DBMS_OUTPUT.PUT_LINE('record array:'||v_rec(1).area.area_no||'.'||v_rec(1).branch||
    ' '||v_rec(2).area.area_no||'.'||v_rec(2).branch);
END;
/
--result
char array:12
array array:1212
udt array:01 02
record array:03.0301 04.0401

--通过NESTED TABLE定义多维数组并访问
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_table char_table;
BEGIN
    v_table := char_table(char_array('1',TO_CHAR(2),''),char_array('1',TO_CHAR(2),''));
    DBMS_OUTPUT.PUT_LINE('nested table: '||v_table(1)(1)||v_table(1)(2)||' '||v_table(2)(1)||v_table(2)(2));
END;
/
--result
nested table: 12 12
```

## 集合变量操作方法

除上述通过索引直接访问集合成员的方法外，YashanDB还内置了一系列方法，用于在过程体内对集合变量执行指定的操作。

需注意的是，除EXISTS外，其他方法都必须在集合变量初始化后才可以操作。

### LIMIT

```
syntax::= (varray_type|nested_table_type) "." "limit" ["()"]
```

LIMIT为一个函数，用于获得集合的成员数量上限，对于嵌套表，函数返回NULL。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array limit: '||v_char.LIMIT);
    DBMS_OUTPUT.PUT_LINE('nested table limit: '||v_t_char.LIMIT);
END;
/

--result
array limit: 5
nested table limit:
```

## COUNT

```
syntax::= (varray_type|nested_table_type) "." "count" ["()"]
```

COUNT为一个函数，用于获得集合的当前成员数量。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array count: '||v_char.count());
    DBMS_OUTPUT.PUT_LINE('nested table limit: '||v_t_char.count());
END;
/

--result
array count: 3
nested count limit: 3
```

## FIRST

```
syntax::= (varray_type|nested_table_type) "." "first" ["()"]
```

FIRST为一个函数，用于获得集合当前第一个成员的索引。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array first: '||v_char.first());
    DBMS_OUTPUT.PUT_LINE('nested table first: '||v_t_char.first());
END;
/
```

```
--result
array first: 1
nested table first: 1
```

## LAST

```
syntax::= (varray_type|nested_table_type) "." "last" ["()"]
```

LAST为一个函数，用于获得集合当前最后一个成员的索引。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array last: '||v_char.last());
    DBMS_OUTPUT.PUT_LINE('nested table last: '||v_t_char.last());
END;
/

--result
array last: 3
nested table last: 3
```

## NEXT

```
syntax::= (varray_type|nested_table_type) "." "next" "(" expr ")"
```

NEXT为一个函数，用于获得输入索引在集合内的下一个索引。

expr为一个YashanDB认可的通用表达式，其结果必须为一个数字（为NULL时函数返回NULL），负数按0处理。当expr+1的值超出集合上下限范围时，函数返回空。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array next0: '||v_char.NEXT(0));
    DBMS_OUTPUT.PUT_LINE('array next-9: '||v_char.NEXT(-9));
    DBMS_OUTPUT.PUT_LINE('array next3: '||v_char.NEXT(3));
    DBMS_OUTPUT.PUT_LINE('nested table next0: '||v_t_char.NEXT(0));
    DBMS_OUTPUT.PUT_LINE('nested table next-9: '||v_t_char.NEXT(-9));
    DBMS_OUTPUT.PUT_LINE('nested table next3: '||v_t_char.NEXT(3));
END;
/

--result
array next0: 1
array next-9: 1
array next3:
nested table next0: 1
nested table next-9: 1
nested table next3:
```

## PRIOR

```
syntax::= (varray_type|nested_table_type) "." "prior" "(" expr ")"
```

PRIOR为一个函数，用于获得输入索引在集合内的下一个索引。

expr为一个YashanDB认可的通用表达式，其结果必须为一个数字（为NULL时函数返回NULL），负数按0处理。当expr-1的值超出集合上下限范围时，函数返回空。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array prior-9: '||v_char.PRIOR(-9));
    DBMS_OUTPUT.PUT_LINE('array prior1: '||v_char.PRIOR(1));
    DBMS_OUTPUT.PUT_LINE('array prior4: '||v_char.PRIOR(4));
    DBMS_OUTPUT.PUT_LINE('nested table prior-9: '||v_t_char.PRIOR(-9));
    DBMS_OUTPUT.PUT_LINE('nested table prior1: '||v_t_char.PRIOR(1));
    DBMS_OUTPUT.PUT_LINE('nested table prior4: '||v_t_char.PRIOR(4));
END;
/

--result
array prior-9:
array prior1:
array prior4: 3
nested table prior-9:
nested table prior1:
nested table prior4: 3
```

## EXISTS

```
syntax::= (varray_type|nested_table_type) "." "exists" "(" expr ")"
```

EXISTS为一个函数，用于判断输入索引位置是否有成员存在。

expr为一个YashanDB认可的通用表达式，其结果必须为一个数字（为NULL时函数返回NULL）。

示例

```
DECLARE
    TYPE char_array IS ARRAY(5) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    DBMS_OUTPUT.PUT_LINE('array exists-9: '||v_char.EXISTS(-9));
    DBMS_OUTPUT.PUT_LINE('array exists1: '||v_char.EXISTS(1));
    DBMS_OUTPUT.PUT_LINE('array exists4: '||v_char.EXISTS(4));
    DBMS_OUTPUT.PUT_LINE('nested table exists-9: '||v_t_char.EXISTS(-9));
    DBMS_OUTPUT.PUT_LINE('nested table exists1: '||v_t_char.EXISTS(1));
    DBMS_OUTPUT.PUT_LINE('nested table exists4: '||v_t_char.EXISTS(4));
END;
/

--result
array exists-9: false
array exists1: true
array exists4: false
nested table exists-9: false
nested table exists1: true
nested table exists4: false
```

## EXTEND

```
syntax::= (varray_type|nested_table_type) "." "extend" ["(" [expr1 ["," expr2]] ")"]
```

EXTEND为一个存储过程，用于拓展集合的当前成员数量，扩展值为expr2所在成员值，拓展个数为expr1个（对于数组，拓展后成员数量不能超过数组上限）。

expr1/expr2为一个YashanDB认可的通用表达式，其结果必须为一个非负数字（为NULL时表示extend(0)）。expr2省略时，表示拓展空值，expr1省略时，表示拓展1个。

示例

```
DECLARE
    TYPE char_array IS ARRAY(10) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','3');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    v_char.EXTEND();
    v_char.EXTEND(1);
    v_char.EXTEND(1,1);
    v_t_char.EXTEND();
    v_t_char.EXTEND(1);
    v_t_char.EXTEND(1,1);
    DBMS_OUTPUT.PUT_LINE('array count: '||v_char.count());
    DBMS_OUTPUT.PUT_LINE('nested table count: '||v_t_char.count());
END;
/

--result
array count: 6
nested table count: 6
```

## TRIM

```
syntax::= (varray_type|nested_table_type) "." "trim" ["(" [expr] ")"]
```

TRIM为一个存储过程，用于删除集合的末尾expr个成员（删除数量不能超过集合当前成员数量）。

expr为一个YashanDB认可的通用表达式，其结果必须为一个非负数字（为NULL时表示trim(0)）。expr省略时，表示删除1个成员。

示例

```
DECLARE
    TYPE char_array IS ARRAY(10) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','3');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    v_char.trim('');
    v_char.trim;
    v_char.trim(2);
    v_t_char.trim('');
    v_t_char.trim;
    v_t_char.trim(2);
    DBMS_OUTPUT.PUT_LINE('array count: '||v_char.count());
    DBMS_OUTPUT.PUT_LINE('nested table count: '||v_t_char.count());
END;
/

--result
```

```
array count: 0
nested table count: 0
```

## DELETE

```
syntax::= (varray_type|nested_table_type) "." "delete" ["()"]
```

DELETE为一个存储过程，用于清空集合中的所有成员。

示例

```
DECLARE
    TYPE char_array IS ARRAY(10) OF CHAR;
    TYPE char_table IS TABLE OF char_array;
    v_char char_array := char_array('1','2','3');
    v_t_char char_table := char_table(v_char, v_char, v_char);
BEGIN
    v_char.DELETE;
    v_t_char.DELETE;
    DBMS_OUTPUT.PUT_LINE('array count: '||v_char.count());
    DBMS_OUTPUT.PUT_LINE('nested table count: '||v_t_char.count());
END;
/

--result
array count: 0
nested table count: 0
```

# 自定义异常

自定义异常变量被用于用户自定义的异常处理中。

在一个自定义异常变量被声明后，可以在过程体中对该异常进行初始化、抛出和接收操作，详见异常处理章节描述。

使用EXCEPTION关键字定义一个异常变量，定义格式为：

*exception_name EXCEPTION;*

其中exception_name为异常名称，长度不超过64字节。异常名称可与YashanDB系统预定义的异常重名。

# 标识符

PL标识符用于标识PL语法单元，包含：

- 字面量
- 游标
- 异常
- 关键字
- 标签
- 包
- 保留关键字
- 过程体对象
- 类型
- 变量

详细见标识符。

## 引用标识符

引用标识符时，使用的名称可以是定义的标识符名称，有时还需要包含其存储位置的名称（如包名）和schema名称。

## 标识符的作用域和可见性

PL单元中可以引用标识符的区域被称为标识符的作用域。PL单元中可以引用标识符而无需对其进行限定的区域被称为标识符的可见性。

对于声明标识符的PL单元，该标识符是本地标识符。如果该单元有子单元，则该标识符对于子单元是全局标识符。如果子单元声明了一个同名的本地标识符，那么在子单元内部，两个标识符均在其作用域内，但只有本地标识符是可见的，不需要限定。若需要引用全局标识符，则需要使用声明它的单元名称进行限定。

对于同一级别的其他单元中声明的标识符，不在其作用域内，不可引用。

不能在同一PL单元中两次声明同一标识符。可以在两个不同的PL单元声明相同的标识符，它们是独立的对象。

# 表达式

表达式是一个或多个值、运算符和SQL函数的组合，其结果为一个值。

一个表达式始终返回唯一值。

操作数可以是变量、常量、常数、运算符、函数调用、占位符或其他表达式。因此，表达式可以任意复杂。

操作数的数据类型决定表达式的数据类型。每次计算表达式时，都会生成该数据类型的单个值。该结果的数据类型就是表达式的数据类型。

过程体的各个地方都可能使用到表达式，例如存储过程/函数等的实参、变量赋值、SQL语句等，这些表达式在使用时均遵循YashanDB的通用表达式语法及规则，同时增加了如下一些限制：

- 表达式作为列字段时，不能使用*代表所有列字段。
- 表达式中不能使用窗口函数。
- 非SELECT的DML语句中的表达式不能使用ROWNUM、ROWID、ROWSCN等伪列，不能使用聚集函数。

# PL名称解析

PL名称解析将介绍PL引擎如何解析对标识符的模糊引用。

## 限定名称和点符号

当一个名称项属于另一个命名项时，您可以（有时必须）使用点表达法使用父项名称来限定子项的名称。例如：

| 引用 | 名称限定 | 语法 | | 记录的字段 | 记录名称 | record_name.field_name | | 包变量 | 包名称 | package_name.var_name |

如果在命名的PL单元中声明了一个标识符，可以使用该单元的名称来限定其声明中的名称（简单名称），使用以下语法：

*unit_name.simple_identifier_name*

如果标识符不可见，则必须限定其名称，标识符可见性具体请参考标识符。

如果标识符属于另一个模式，则必须使用模式名称限定其名称。例如：

*schema_name.package_name*

# PL对象

PL对象是基于PL语言按语法规则组织的一种数据库对象，如存储过程、匿名块、自定义函数等。

一个典型的PL对象的结构为：

*头部*

IS/AS

*过程体*

其中，头部包含名称、参数、返回类型等内容；过程体则是实现PL语言的主要载体，主要包括如下内容：

- 变量定义及管理
- 运算
- 控制
- 传递（变量、参数）
- 特有的数据处理

## 创建PL对象

包括对PL对象的定义和编译。

对PL对象的定义是通过编写PL语言来描述其结构的过程。

对PL对象的编译是使用YashanDB的编译器对上面定义的内容进行分析、翻译等操作后生成可执行代码的过程。

对于使用CREATE PROCEDURE、CREATE FUNCTION等SQL语句创建的PL对象，其在创建后将被保存在数据库中，即对其实现持久化。

对于只有过程体，无对象名称和参数定义，不需要通过SQL语句创建的PL对象，如匿名块，其在创建后不会保存，而是直接运行。

## 运行PL对象

当匿名块创建后，或者当持久化的PL对象被调用（在SQL或PL内部调用，或通过CALL/EXEC等SQL语句调用）时，YashanDB的执行器将会找到该对象编译好的代码体并运行，执行各类PL语句所定义的操作，生成结果返回，或者抛出异常。

## 删除PL对象

对于不再使用的PL对象，可通过DROP PROCEDURE、DROP FUNCTION等SQL语句将其从数据库中删除。

## 重编译PL对象

通过ALTER FUNCTION、ALTER PACKAGE、ALTER PROCEDURE、ALTER TRIGGER等SQL语句，可以对自定义函数、自定义高级包、存储过程、触发器等进行重编译。

如果重编译的对象有任何依赖的对象失效，系统将首先重编译这些依赖的对象。

对一个PL对象重编译成功后，该对象将被置为有效状态。如果重编译失败，系统返回相应报错，该对象变为无效状态，且系统会同时失效依赖于该对象的其他对象。

## PL对象权限控制

YashanDB通过一套系统权限机制（见产品安全手册系统权限管理）对PL对象的创建和使用进行安全控制，具体包括如下权限：

- CREATE/DROP PROCEDURE：创建/删除存储过程、函数、高级包、类型的权限。
- CREATE/ALTER/DROP ANY PROCEDURE：创建/修改/删除所有用户（sys用户除外）下的存储过程、函数、高级包、类型的权限。
- EXECUTE ANY PROCEDURE：执行所有用户（sys用户除外）下的存储过程、函数、高级包的权限。
- CREATE TRIGGER：创建/修改/删除触发器的权限。
- CREATE/ALTER/DROP TRIGGER：创建/修改/删除所有用户（sys用户除外）下的触发器的权限。
- ADMINISTER DATABASE TRIGGER：管理数据库级别触发器的权限。
- CREATE/DROP TYPE：创建/删除自定义类型。
- CREATE/ALTER/DROP ANY TYPE：创建/修改/删除所有用户（sys用户除外）下的自定义类型。

- EXECUTE ANY TYPE：执行所有用户（sys用户除外）下的自定义类型。
- UNDER ANY TYPE: 在非最终的OBJECT类型下创建子类型。

## invoker_rights_clause

上述权限定义了某个用户对PL对象的权限，该用户可能是PL对象的所属用户，或者为调用PL执行的用户。由于PL过程体中存在SQL语句，相应地会存在执行SQL时的权限检查。

invoker_rights_clause用于在定义PL对象时同时定义其AUTHID属性，指定此种情形下的执行权限模式。

**invoker_rights_clause::=**

```
syntax::= AUTHID (CURRENT_USER | DEFINER)
```

在定义一个PL对象的语句中，invoker_rights_clause子句只能出现一次。

在独立的存储过程、函数中用于指定该对象的AUTHID属性。

在高级包中用于指定该高级包下所有子程序的AUTHID属性。

在类型中只用于指定Object UDT的方法的AUTHID属性，不能在定义Varray/Table UDT时指定，且子类型必须与父类型指定相同的AUTHID属性。

定义匿名块和触发器时不能使用本语句，而是采用固定的AUTHID属性：

- 匿名块的AUTHID属性为CURRENT_USER
- 触发器的AUTHID属性为DEFINER

### current_user

表示过程体执行时的默认模式是调用PL对象的用户，且对该用户执行过程体中SQL语句的权限检查。

### definer

默认行为。表示过程体中在执行时的默认模式是PL对象所属的用户，且对该用户执行过程体中SQL语句的权限检查。

> **Note**：
> 通过dba/all/USER_PROCEDURES视图可查看PL对象的执行权限模式（AUTHID属性）。
> CURRENT_USER当前为语法兼容模式，即无论是否指定AUTHID属性，系统均按DEFINER模式执行权限检查。

# 存储过程

存储过程是数据库里的一种PL对象。

分布式部署中不能创建和运行存储过程。

## 创建存储过程

使用CREATE PROCEDURE语句创建存储过程，其语法定义为：

**create procedure::=**

```
syntax::= CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] PROCEDURE procedure_head_clause (IS|AS) procedure_body_clause ";"
```

**procedure_head_clause::=**

```
syntax::= [schema "."] procedure_name ["(" (argument_define) {"," (argument_define)} ")"] [invoker_rights_clause]
```

**invoker_rights_clause::=**

```
syntax::= AUTHID (CURRENT_USER|DEFINER)
```

**procedure_body_clause::=**

```
syntax::= [variable_declare] BEGIN plsql_statements END [procedure_name]
```

### or replace

当要创建的存储过程已经存在时，将其进行重建。

### editionable | noneditionable

用于语法兼容，无实际含义。

### procedure_name

存储过程的名称，不可省略，且需符合YashanDB的对象命名规范。

### argument_define

定义存储过程的参数（形参），可省略，此时procedure_name()=procedure_name。最多可以指定4095个参数。

PL参数描述。

### invoker_rights_clause

见首页invoker_rights_clause部分描述。

### variable_declare

声明PL过程体的全局变量，可省略。

PL声明。

### plsql_statements

定义过程体中的执行语句，为PL语句中的一项或多项。

示例（单机、共享集群部署）

```
CREATE OR REPLACE PROCEDURE ya_proc(i INT) IS
BEGIN
CASE i
WHEN 1 THEN
DBMS_OUTPUT.PUT_LINE('hello');
WHEN 2 THEN
DBMS_OUTPUT.PUT_LINE('world');
END CASE;
END ya_proc;
/
```

# 运行存储过程

## sql语句中调用

通过CALL、EXEC语句调用存储过程，其语法定义为：

```
syntax::= (CALL|EXEC) procedure_name ["(" arguments ")"]
```

arguments为与argument_define一一对应的实参。

示例

```
exec ya_proc(1);

--result
hello
```

## 过程体中调用

格式为：*procedure_name(arguments);*。

arguments为与argument_define一一对应的实参。

示例（单机、共享集群部署）

```
BEGIN
  ya_proc(1);
  ya_proc(2);
END;
/

--result
hello
world
```

# 删除存储过程

使用DROP PROCEDURE语句删除存储过程。

# 修改存储过程

使用ALTER PROCEDURE语句修改存储过程。

# 匿名块

匿名块是数据库里的一种特殊的PL对象，与其他对象不同的是，它不会被持久化，创建后立即运行，不能被调用。

## 创建匿名块

匿名块无名称、参数等定义，只包含过程体，其语法定义为：

```
syntax::= [DECLARE variable_declare] BEGIN plsql_statements END ";"
```

### variable_declare

声明PL过程体的全局变量，可省略。

PL声明。

### plsql_statements

定义过程体中的执行语句，为PL语句中的一项或多项。

示例

```
DECLARE
i INT :=1;
BEGIN
CASE i
WHEN 1 THEN
DBMS_OUTPUT.PUT_LINE('hello');
WHEN 2 THEN
DBMS_OUTPUT.PUT_LINE('world');
END CASE;
END;
/
```

# 自定义函数

自定义函数是数据库里的一种PL对象，简称UDF。

分布式部署中不能创建和使用UDF。

## 创建自定义函数

使用CREATE FUNCTION语句创建自定义函数，其语法定义为：

**create function::=**

```
syntax::= CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] FUNCTION function_head_clause (IS|AS) function_body_clause ";"
```

**function_head_clause::=**

```
syntax::= [schema "."] function_name ["("  (argument_define) {"," (argument_define)}  ")"]
 RETURN return_datatype [invoker_rights_clause]
```

**invoker_rights_clause::=**

```
syntax::= AUTHID (CURRENT_USER|DEFINER)
```

**function_body_clause::=**

```
syntax::= [variable_declare] BEGIN plsql_statements END [function_name]
```

### or replace

当要创建的自定义函数已经存在时，将其进行重建。

### editionable | noneditionable

用于语法兼容，无实际含义。

### function_name

自定义函数的名称，不可省略，且需符合YashanDB的对象命名规范。

### argument_define

自定义函数的参数（形参），可省略，此时function_name()=function_name。最多可以指定4095个参数。

PL参数描述。

### return_datatype

自定义函数的返回值类型，此处为隐式形参，因此不可以为其指定长度、精度等属性。

### invoker_rights_clause

见首页invoker_rights_clause部分描述。

### variable_declare

声明PL过程体的全局变量，可省略。

PL变量声明。

### plsql_statements

定义过程体中的执行语句，为PL语句中的一项或多项。

过程体中的每个执行路径（流程控制语句的分支，异常处理单元的分支）里都必须包含至少一个RETURN Statement。

示例（单机、共享集群部署）

```
CREATE OR REPLACE FUNCTION ya_func(i INT) RETURN VARCHAR
IS
BEGIN
CASE i
WHEN 1 THEN
RETURN 'hello';
WHEN 2 THEN
RETURN 'world';
END CASE;
END ya_func;
/
```

# 运行自定义函数

## sql语句中调用

与内置函数的调用方法相同，作为SQL表达式使用，例如：

*SELECT function_name(arguments) FROM dual；*

arguments为与argument_define一一对应的实参，且argument_define不能为IN OUT或OUT形参。

示例（单机、共享集群部署）

```
SELECT ya_func(1)||' '||ya_func(2) res FROM dual;
RES
----------------------------------------------------------------
hello world
```

## 过程体中调用

与内置函数的调用方法相同，作为表达式在PL语句中使用，或者用于给变量赋值，例如：

*variable_name := procedure_name(arguments);*

arguments为与argument_define一一对应的实参。

示例（单机、共享集群部署）

```
DECLARE
  a VARCHAR(10);
  b VARCHAR(10);
BEGIN
  a := ya_func(1);
  b := ya_func(2);
  DBMS_OUTPUT.PUT_LINE(a||' '||b);
END;
/

--result
hello world
```

# 删除自定义函数

使用DROP FUNCTION语句删除自定义函数。

# 修改自定义函数

使用ALTER FUNCTION语句修改自定义函数。

# 自定义高级包

自定义高级包是数据库里的一种PL对象，简称UDP。

分布式部署中不能创建和使用UDP。

## 创建UDP

UDP的创建分为两部分：

- 定义PACKAGE HEAD：PACKAGE HEAD用于声明PUBLIC属性的变量、类型、游标和子过程体对象（存储过程、自定义函数），在UDP被成功创建后，这些变量、类型、游标和子过程体对象将可以作为UDP的成员被其他外部程序所引用。
- 定义PACKAGE BODY：PACKAGE BODY用于定义过程体，PACKAGE BODY除可以直接使用PACKAGE HEAD中声明的变量外，也可以自行声明新的变量，但这些变量为该UDP私有（PRIVATE），即只能在PACKAGE BODY中使用，而不可以被外部其他过程体所引用。

PACKAGE HEAD和PACKAGE BODY通过不同语法分别定义，且YashanDB并不严格要求HEAD和BODY的创建顺序，如在HEAD还未创建时先定义BODY，系统将仍然创建该UDP，但会因为HEAD声明不存在而抛出编译错误，该UDP也无法被执行和调用，直到HEAD被创建。

系统对PACKAGE HEAD和PACKAGE BODY还存在如下约束要求：

- HEAD和BODY内部不允许出现同名的变量或者子过程体对象声明；
- HEAD中的子过程体对象声明，在BODY中必须有对应的过程体定义；
- 对于子过程体对象，HEAD中只能对其进行声明，BODY中只能对其进行过程体定义。

使用CREATE PACKAGE语句可以分别创建PACKAGE HEAD或者PACKAGE BODY，语法定义如下：

**create package::=**

```
syntax::= CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] PACKAGE (package_head_clause|package_body_clause) END
[package_name]";"
```

**package_head_clause::=**

```
syntax::= [schema "."] package_name [invoker_rights_clause] (IS|AS) [pragma_clause] package_item_clause
```

**invoker_rights_clause::=**

```
syntax::= AUTHID (CURRENT_USER|DEFINER)
```

**pragma_clause::=**

```
syntax::= PRAGMA SERIALLY_REUSABLE";"
```

**package_item_clause::=**

```
syntax::= (variable_declare|pl_declare) {(variable_declare|pl_declare)}
```

variable_declare

**pl_declare::=**

```
syntax::= (PROCEDURE|FUNCTION) pl_name ["(" (argument_define) {"," (argument_define)} ")"] [RETURN return_datatype]
```

**package_body_clause::=**

```
syntax::= BODY [schema "."] package_name (IS|AS) [package_item_clause] package_subprogram_clause
```

**package_subprogram_clause::=**

```
syntax::= (IS|AS) [variable_declare] BEGIN plsql_statements END [pl_name]
```

## or replace

当要创建的UDP已经存在时，将其进行重建。

## editionable | noneditionable

用于语法兼容，无实际含义。

## package_name

UDP的名称，不可省略，且需符合YashanDB的对象命名规范。

## package_head_clause

创建一个UDP的HEAD。

## invoker_rights_clause

见PL对象中invoker_rights_clause部分描述。

## pragma_clause

可选项，表示UDP的变量每次被引用时，是否将其重置为变量声明时定义的缺省值，默认不重置。

在YashanDB里，在UDP中声明的变量均为SESSION级别的全局变量，这意味着在同一个会话中，变量赋值相互影响，而不同会话中的变量则相互独立。通过设置SERIALLY_REUSABLE可以在每次引用变量时让其恢复初始值。

示例（单机、共享集群部署）

```
DROP PACKAGE IF EXISTS calc_fee;

CREATE OR REPLACE PACKAGE calc_fee
AS
c NUMBER := 100;
PROCEDURE branch_quantity(date_from date);
END;
/

CREATE OR REPLACE PACKAGE BODY calc_fee AS
PROCEDURE branch_quantity(date_from DATE) IS
BEGIN
 DBMS_OUTPUT.PUT_LINE(c);
 c := c+1;
 DBMS_OUTPUT.PUT_LINE(c);
END;
END calc_fee;
/

--第一次执行
exec calc_fee.branch_quantity(sysdate);
100
101
--第二次执行
exec calc_fee.branch_quantity(sysdate);
101
102

--设置pragma_clause
CREATE OR REPLACE PACKAGE calc_fee
AS
PRAGMA SERIALLY_REUSABLE;
c NUMBER := 100;
PROCEDURE branch_quantity(date_from date);
END;
```

```
/

CREATE OR REPLACE PACKAGE BODY calc_fee AS
PRAGMA SERIALLY_REUSABLE;
PROCEDURE branch_quantity(date_from date) IS
BEGIN
DBMS_OUTPUT.PUT_LINE(c);
c := c+1;
DBMS_OUTPUT.PUT_LINE(c);
END;
END calc_fee;
/

--第一次执行
exec calc_fee.branch_quantity(sysdate);
100
101
--第二次执行
exec calc_fee.branch_quantity(sysdate);
100
101
```

### package_item_clause

UDP的成员声明，包括变量声明和子过程体对象声明，一个UDP（HEAD或BODY）里可声明的成员数量最多为1024个。

- 变量声明：支持在声明章节中列出的各类变量的声明，但对于游标，允许显式定义游标，不允许显式声明游标。可交叉声明多个不同类变量且顺序无限制。
- 子过程体对象声明：支持声明存储过程和自定义函数。允许交叉声明多个存储过程和自定义函数，且顺序无限制。

变量声明必须位于子过程体对象声明的前面，且定义的成员先后顺序决定了后一个成员是否可见前面成员。

### package_body_clause

创建一个UDP的BODY。

### package_subprogram_clause

UDP里的过程体定义，包括PROCEDURE或FUNCTION的定义，定义语法同存储过程和自定义函数章节里的过程体语法描述。

## 加载udp的变量

UDP的变量无法在外部SQL语句中直接访问，而是通过在过程体（内部或外部）中加载使用，加载格式为：

*[schema.][package_name.]variable_name;*

在被外部过程体调用时，package_name不可省略。

在YashanDB中，UDP的变量与UDP进行绑定管理，如果UDP的HEAD或者BODY发生变更，该UDP的所有变量将被重置。此规则与Oracle数据库存在差异。

## 运行udp的子过程体对象

**SQL语句中调用子函数**

通过SQL语句调用UDP的子函数，允许出现在SQL语句的列项、条件项或查询选项（如ORDER BY/LIMIT/OFFSET）等各个位置。以出现在查询列为例的调用格式为：

*SELECT package_name.subpl_name[(arguments)] FROM table_name；*

**CALL/EXEC语句中调用子存储过程**

通过CALL/EXEC语句可调用UDP的子存储过程，调用格式为：

*(CALL|EXEC) package_name.subpl_name[(arguments)]；*

**过程体中调用子存储过程**

通过在过程体（内部或外部）中调用子存储过程，调用格式为：

*[package_name.]procedure_name[(arguments)];*

在被外部过程体调用时，package_name不可省略。

示例（单机、共享集群部署）

```
--创建无HEAD的UDP
DROP PACKAGE IF EXISTS calc_fee;

--创建UDP的BODY，由于未声明相关变量会返回错误，实际上该BODY已创建成功
CREATE OR REPLACE PACKAGE BODY calc_fee AS
PROCEDURE calc_rev(prefix CHAR) IS
 str1 VARCHAR(100) := 'select sum(revenue_total) from finance_info where substr(branch,0,2)=:a';
BEGIN
 EXECUTE IMMEDIATE str1 INTO c USING prefix;
 DBMS_OUTPUT.PUT_LINE(c);
END;
END calc_fee;
/
--result
YAS-05282 package body compiling "CALC_FEE" errors:

--创建calc_fee的HEAD（此步骤后无需再次创建BODY）
CREATE OR REPLACE PACKAGE calc_fee
AS
c NUMBER := 100;
PROCEDURE calc_rev(prefix CHAR);
END calc_fee;
/

--调用UDP
DECLARE
 c CHAR(2);
BEGIN
 c := '01';
 DBMS_OUTPUT.PUT_LINE(c);
 DBMS_OUTPUT.PUT_LINE(calc_fee.c);
 calc_fee.calc_rev(c);
END;
/
--result
01
100
76666
```

## 删除自定义高级包

使用DROP PACKAGE语句删除自定义高级包。

示例（单机、共享集群部署）

```
--创建UDF，调用上例中的UDP子过程体对象
CREATE OR REPLACE FUNCTION calc(c CHAR) RETURN NUMBER IS
BEGIN
  calc_fee.calc_rev(c);
  RETURN calc_fee.c;
END;
/
SELECT calc('01') FROM dual;
 CALC('01')
-----------
      76666

--删除UDP
DROP PACKAGE calc_fee;
SELECT calc('01') FROM dual;
[1:8]YAS-04253 PL/SQL compiling errors:
```

```
[3:3] YAS-04243 invalid identifier "CALC_FEE"."CALC_REV"
[4:10] YAS-04243 invalid identifier "CALC_FEE"."C"
```

## 修改自定义高级包

使用ALTER PACKAGE语句修改自定义高级包。

# 自定义类型

自定义类型是数据库里的一种PL对象，简称UDT。

分布式部署中不能创建和使用UDT。

UDT可以定义的数据结构种类包括：

- 对象结构Object：面向对象概念的抽象数据类型（Abstract Data Type），包含属性和方法，UDT之间可以继承或嵌套。需要创建类型和类型主体两部分，且必须先创建类型，再创建类型主体。
- 数组结构Varray： 一种需定义最大数量、包含可变长度的多个元素的集合类型 ，除可被创建为一个独立的UDT对象外，还可以作为集合变量中的数组变量直接在过程体内进行声明。只需要创建类型部分。
- 嵌套表结构Nested Table： 一种未定义最大数量、包含多行的可变长度的多个元素的集合类型 ，除可被创建为一个独立的UDT对象外，还可以作为集合变量的嵌套表变量直接在过程体内进行声明。只需要创建类型部分。

**UDT的初始化**

系统对每一个自定义类型自动创建一个同名的构造函数，在对象实例化时进行初始化。

**UDT的实例化**

将数据类型声明为UDT的一个变量，即为该UDT的实例，对变量进行初始化赋值即为该UDT的实例化。

**UDT的依赖**

当一个UDT被表或者其他UDT引用时，该UDT就成为被依赖对象，无法重建、修改或删除（除非使用FORCE选项）。

对于继承性依赖，如父类型强制修改或重建，子类型也将自动重建。

对于嵌套性依赖，如被依赖的UDT强制修改时，依赖的UDT将自动失效。

**UDT的权限**

拥有CREATE TYPE权限可以在自己的模式下创建UDT，拥有CREATE ANY TYPE权限才可以在其它模式下创建UDT。拥有EXECUTE ANY TYPE权限才可以使用其它模式的UDT。

在其它模式下创建UDT时，UDT所属用户必须直接拥有相关权限，此时UDT所属用户从角色继承的权限不会生效。

## 创建UDT类型

通过CREATE TYPE语句创建一个UDT类型，包括UDT的定义和编译，如果UDT编译失败，依然会被创建，但在使用时将报错。

create type**::=**

```
syntax::=  CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] TYPE [schema "."] type_name
[FORCE] [invoker_rights_clause]
(object_base_type_def | object_subtype_def)
```

invoker_rights_clause**::=**

```
syntax::= AUTHID (CURRENT_USER|DEFINER)
```

object_base_type_def**::=**

```
syntax::= ( AS | IS ) (object_type_def | varray_type_spec | nested_table_type_def)
```

object_type_def**::=**

```
syntax::= OBJECT  [ "(" (attribute datatype) { ","  (attribute datatype)}
[ "," element_spec ] ")" ] [ [NOT]  FINAL ]
```

element_spec::=

```
syntax::=( ( constructor_spec | subprogram_spec | map_order_function_spec )) {( ( constructor_spec | subprogram_spec |
map_order_function_spec ))}
```

constructor_spec::=

```
syntax::= CONSTRUCTOR FUNCTION type_name [ "(" [SELF IN OUT datatype ","] (parameter datatype) { "," (parameter datatype)}
")" ] RETURN SELF AS RESULT
```

subprogram_spec::=

```
syntax::= ( MEMBER | STATIC ) ( procedure_spec | function_spec )
```

map_order_function_spec::=

```
syntax::=  ( MAP | ORDER ) MEMBER function_spec
```

varray_type_spec::=

```
syntax::= (VARRAY | ( [VARYING] ARRAY ) ) "(" size_limit ")" OF datatype [ NOT NULL ]
```

nested_table_type_def::=

```
syntax::= TABLE OF "(" datatype [NOT NULL] ")"
```

object_subtype_def::=

```
syntax::= UNDER [ schema "." ] supertype [ "(" (attribute datatype) { "," (attribute datatype)} [ "," element_spec ] ")" ] [
[NOT] FINAL ]
```

### or replace

当要创建的UDT已经存在时，将其进行重建。

### editionable | noneditionable

用于语法兼容，无实际含义。

### schema

包含UDT的模式名称，可省略，则默认为当前登录用户的模式。

### type_name

要创建的UDT的名称，在同一模式下的UDT不能有相同的名称。不可省略，且需符合YashanDB的对象命名规范。

### force

当类型名称存在且被其它对象依赖时，REPLACE FORCE选项可以强制替换UDT的定义。 如果类型被表依赖时，不能使用REPLACE FORCE进行替换。

### invoker_rights_clause

见首页invoker_rights_clause部分描述。

### object_base_type_def

创建模式级别的UDT。

### object_type_def

指定OBJECT关键字可以创建一个Object类型的UDT。

构成Object的成员变量称为属性，定义Object行为的成员子程序称为方法。

一个UDT必须至少包含一个属性，可以不包含任何方法。

#### attribute

Object包含的属性名称。每个Object类型的UDT至少包含一个属性，且属性的名称在该UDT内唯一。

#### datatype

属性的数据类型，可以是系统内置的数据类型，或之前已定义的某个UDT（此种情况称为UDT嵌套，当前类型将对该UDT形成依赖）。

#### element_spec

指定Object包含的方法，可以为构造方法、静态方法和成员方法。

方法参数的默认值最大长度是8000。

##### 构造方法

每个UDT都会存在一个由系统创建的与类型同名的默认构造函数，用于UDT的初始化工作。

同时，用户也可以自定义一个构造函数，其名称必须与类型同名。 当自定义构造函数的参数/属性个数、类型与默认的构造函数完全一致时，系统将先调用自定义构造函数。

示例（单机、共享集群部署）

```
CREATE OR REPLACE TYPE udt_object AS OBJECT (
    area_no CHAR(2),
    CONSTRUCTOR FUNCTION udt_object(SELF IN OUT udt_object,a VARCHAR) RETURN SELF AS RESULT
);
/

CREATE OR REPLACE TYPE BODY udt_object AS
    CONSTRUCTOR FUNCTION udt_object(SELF IN OUT udt_object,a VARCHAR) RETURN SELF AS RESULT AS
    BEGIN
        self.area_no := '0'||a;
        RETURN;
    END;
END;
/

DECLARE
    obj1 udt_object;
BEGIN
    obj1 := udt_object(3);
    DBMS_OUTPUT.PUT_LINE('area no is: '||obj1.area_no);
END;
/

--result
area no is: 03
```

##### 静态方法

静态方法为UDT的全局方法，无self参数，不能由对象实例（声明为该UDT的变量）调用。

可以声明为一个静态的存储过程或静态的函数。

示例（单机、共享集群部署）

```
CREATE OR REPLACE TYPE udt_object_static AS OBJECT (
    area_no CHAR(2),
    STATIC FUNCTION showAreaStatic(a VARCHAR) RETURN VARCHAR
);
/
CREATE OR REPLACE TYPE BODY udt_object_static AS
    STATIC FUNCTION showAreaStatic(a VARCHAR) RETURN VARCHAR IS
    BEGIN
        RETURN '0'||a;
    END;
END;
/

DECLARE
    obj udt_object_static;
BEGIN
    obj := udt_object_static(3);
    DBMS_OUTPUT.PUT_LINE('area_no is: '||udt_object_static.showAreaStatic(obj.area_no));
END;
/

--result
area_no is: 03
```

**成员方法**

成员方法为对象实例可调用的方法，默认存在一个self参数（函数为IN，存储过程为IN OUT）。

可以声明为一个成员存储过程或成员函数，当为成员函数时，可声明为如下两类特殊的成员函数：

- MAP函数：用于将对象实例映射为标量数值，MAP函数不可以定义参数，且返回类型不可以为LOB型。
- ORDER函数：用于对两个对象实例进行比较，ORDER函数必须声明一个类型为其所在UDT的参数，且只能为一个。其返回类型只能为INT。

一个UDT中，MAP函数和ORDER函数不能同时存在，且最多只能存在一个。

示例（单机、共享集群部署）

```
--map函数
CREATE OR REPLACE TYPE udt_object_member FORCE AS OBJECT (
    area_no CHAR(2),
    MAP MEMBER FUNCTION showArea RETURN VARCHAR
) NOT FINAL;
/
CREATE OR REPLACE TYPE BODY udt_object_member AS
    MAP MEMBER FUNCTION showArea RETURN VARCHAR AS
    BEGIN
        RETURN self.area_no;
    END;
END;
/

DECLARE
    obj1 udt_object_member;
    obj2 udt_object_member;
BEGIN
    obj1 := udt_object_member(3);
    obj2 := udt_object_member('a');
    IF obj1.showArea > obj2.showArea THEN
        DBMS_OUTPUT.PUT_LINE('compare result: '||obj1.area_no);
    ELSE
        DBMS_OUTPUT.PUT_LINE('compare result: '||obj2.area_no);
    END IF;
END;
/

--result
compare result: a
```

```
--order函数
CREATE OR REPLACE TYPE udt_object_order AS OBJECT (
    area_no CHAR(2),
    ORDER MEMBER FUNCTION orderArea(a udt_object_order) RETURN INT
);
/
CREATE OR REPLACE TYPE BODY udt_object_order AS
    ORDER MEMBER FUNCTION orderArea(a udt_object_order) RETURN INT AS
    BEGIN
        CASE
        WHEN self.area_no = a.area_no THEN RETURN 0;
        WHEN self.area_no > a.area_no THEN RETURN 1;
        WHEN self.area_no < a.area_no THEN RETURN -1;
        END CASE;
    END;
END;
/

DECLARE
    obj1 udt_object_order;
    obj2 udt_object_order;
BEGIN
    obj1 := udt_object_order(3);
    obj2 := udt_object_order('a');
    DBMS_OUTPUT.PUT_LINE('compare result: '||obj1.orderArea(obj2));
END;
/

--result
compare result: -1
```

**[not] final**

指定NOT FINAL表示可以为此类型创建子类型。默认为FINAL，即不能为此类型创建子类型。

### varray_type_spec

指定VARRAY或[VARYING] ARRAY关键字可以创建一个数组类型的UDT。数组是一组有序元素的集合，长度限定，每个元素都为datatype所指定的数据类型。

除datatype可以为普通标量数据类型，及已定义的UDT外，数组类型UDT的定义、使用及操作方法均与在过程体内声明的集合变量类型中的数组类型一致。

示例（单机、共享集群部署）

```
-- 创建一个数组类型，此类型的元素是上例中的udt_object，数组的元素上限是10个。
CREATE OR REPLACE TYPE udt_varray AS VARRAY(10) OF udt_object;
/

-- 通过默认构造函数给udt_varray类型的变量赋值。
DECLARE
    arr udt_varray;
BEGIN
    arr := udt_varray(udt_object(1),udt_object(2),udt_object(3));
    DBMS_OUTPUT.PUT_LINE('attribute of varray element:' || arr(1).area_no);
END;
/

--result
attribute of varray element:1
```

### nested_table_type_def

指定TABLE关键字可以创建一个嵌套表类型的UDT。嵌套表是一个不限长度的多行元素的集合，每个元素都为datatype所指定的数据类型。

除datatype可以为普通标量数据类型，及已定义的UDT外，嵌套表类型UDT的定义、使用及操作方法均与在过程体内声明的集合变量类型中的嵌套表类型一致。

示例（单机、共享集群部署）

```
-- 创建一个嵌套表类型，此类型的元素是上例中的udt_varray。
CREATE OR REPLACE TYPE udt_nested_table AS TABLE OF udt_varray NOT NULL;
/

-- 通过默认构造函数给udt_nested_table类型的变量赋值,并将其赋值给另一个变量。
DECLARE
    nested_table udt_nested_table;
    nested_table2 udt_nested_table;
BEGIN
    nested_table := udt_nested_table(udt_varray(udt_object(1),udt_object(2),udt_object(3)),udt_varray(NULL));
    nested_table2 := nested_table;
    DBMS_OUTPUT.PUT_LINE('Records of nested_table2 is: '||nested_table2.count);
END;
/

--result
Records of nested_table is: 2
```

### object_subtype_def

创建一个已存在类型的子类型（此种情况称为UDT继承，子类型将继承父类型的属性和方法）。

子类型除属性名称不能与父类型中声明的任何属性和方法名称相同外，其他语法与创建父类型一致。

#### supertype

已经存在的父类型名称，父类型必须是一个NOT FINAL的Object UDT。

示例（单机、共享集群部署）

```
--创建前例中udt_object_member的子类型
CREATE OR REPLACE TYPE udt_object_child UNDER udt_object_member(branch_no CHAR(4));
/

DECLARE
    obj udt_object_child;
BEGIN
    obj := udt_object_child('02','0201');
    DBMS_OUTPUT.PUT_LINE('area is:'||obj.showArea||' branch is:'||obj.branch_no);
END;
/

--result
area is:02 branch is:0201
```

## 创建udt类型主体

对于在Object类型中定义的方法，必须在类型主体中对应实现才能使用。 通过CREATE TYPE BODY语句创建一个UDT类型主体，类型主体的名称必须和已创建的UDT的名称一样，否则该类型主体是无效状态。

**create type body::=**

```
syntax::= CREATE [ OR REPLACE ] [ EDITIONABLE | NONEDITIONABLE ] TYPE BODY [ schema "." ] type_name ( IS | AS ) element_spec
element_spec_body  { " " element_spec element_spec_body} END ";"
```

**element_spec_body::=**

```
syntax::= (IS|AS) [variable_declare] BEGIN plsql_statements END ";"
```

### or replace

当要创建的UDT主体已经存在时，将其进行重建。

### editionable | noneditionable

用于语法兼容，无实际含义。

### schema

包含UDT主体的模式名称，可省略，则默认为当前登录用户的模式。

### type_name

要创建的UDT主体的名称，在同一模式下的UDT主体不能有相同的名称。不可省略，且需符合YashanDB的对象命名规范。

### element_spec

在UDT中已声明的方法。

### element_spec_body

方法的实现过程，语法同存储过程和自定义函数章节里的过程体语法描述。

示例（单机、共享集群部署）

```
-- 创建一个OBJECT类型的自定义类型udt_object，此类型有4个属性，2个方法，一个成员存储过程，一个静态函数。
CREATE OR REPLACE TYPE udt_object FORCE IS OBJECT (
    branch_no CHAR(4),
    branch_name VARCHAR2(200),
    area_no CHAR(2),
    address VARCHAR2(200),
    MEMBER PROCEDURE showBranch,
    STATIC FUNCTION showObjectType(p1 VARCHAR2) RETURN VARCHAR2);
/

CREATE OR REPLACE TYPE BODY udt_object AS
MEMBER PROCEDURE showBranch IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('branch_name:  ' || branch_name);
    DBMS_OUTPUT.PUT_LINE('address:      ' || SELF.address);
END;
STATIC FUNCTION showObjectType(p1 VARCHAR2) RETURN VARCHAR2 IS
v1 VARCHAR2(64);
BEGIN
    v1 := p1;
    DBMS_OUTPUT.PUT_LINE('This is STATIC METHOD of udt_object TYPE.');
    RETURN v1;
END;
END;
/

-- 通过默认构造函数给udt_object类型的变量赋值，调用变量的成员存储过程，调用udt_object类型的静态函数。
DECLARE
    obj1 udt_object;
BEGIN
    obj1 := udt_object('0101','Beijing','01','North Street');
    obj1.showBranch();
    DBMS_OUTPUT.PUT_LINE(udt_object.showObjectType('Shenzhen'));
END;
/

-- 创建一个数组类型，此类型的元素是自定义类型udt_object，数组的元素上限是10个。
CREATE OR REPLACE TYPE udt_varray_element AS VARRAY(10) OF udt_object;
/
```

```
-- 通过默认构造函数给udt_varray_element类型的变量赋值。
DECLARE
    arr1 udt_varray_element;
BEGIN
    arr1 := udt_varray_element(udt_object('0101','Beijing','11','North Street'),
udt_object('0102','Shanghai','22','Pedestrian street of Nanjing Road'));
    DBMS_OUTPUT.PUT_LINE('attribute of varray element:' || arr1(1).address);
    -- method OF varray element
    arr1(1).showBranch();
    DBMS_OUTPUT.PUT_LINE('varray element count(INIT):' || arr1.COUNT);
    arr1.TRIM();
    DBMS_OUTPUT.PUT_LINE('varray element count(TRIM):' || arr1.COUNT);
    arr1.EXTEND(2, 1);
    DBMS_OUTPUT.PUT_LINE('varray element count(EXTEND):' || arr1.COUNT);
    DBMS_OUTPUT.PUT_LINE('attribute of varray element(copyed):' || arr1(3).address);
END;
/
```

## 删除udt

通过DROP TYPE BODY可以删除一个UDT类型主体。

通过DROP TYPE可以删除一个UDT类型，同时删除它存在的类型主体。

## 修改udt

通过ALTER TYPE可以修改一个UDT类型。

# 定时任务

定时任务（JOB）是数据库里的一种PL对象。分布式部署中不能创建定时任务。

系统提供如下内置高级包用于创建和管理定时任务：

- DBMS_JOB
- DBMS_SCHEDULER

一个定时任务包含如下三个基本要素：

- JOB唯一标识
- JOB需要执行的任务
- JOB执行时间及频率

DBMS_JOB基于上述三个要素，提供一系列存储过程实现对一个JOB的直接操作管理。DBMS_SCHEDULER相比DBMS_JOB提供了更丰富和灵活的功能，如可设置任务名称，任务结束时间，结束后自动删除等。

在调用高级包执行创建或管理定时任务的各种操作后，用户可通过如下配置参数或系统视图监控定时任务的执行状况和各种属性：

**JOB_QUEUE_PROCESSES**

该配置参数用于指定后台运行JOB的线程数量，当JOB_QUEUE_PROCESSES=0时，系统中所有的JOB都不会被自动执行；当某一时间需要自动执行的JOB数量大于当前空闲的JOB线程数时，JOB需要排队等待 。

**JOB/SCHEDULER视图**

JOB视图：DBA_JOBS/ALL_JOBS/USER_JOBS。

SCHEDULER视图：DBA_SCHEDULER_JOBS/ALL_SCHEDULER_JOBS/USER_SCHEDULER_JOBS。

查看定时任务的各项属性，跟踪定时任务的执行情况，例如查看当前用户下处于调度中的任务队列（未被停止，或状态为有效且在有效期内）：

示例（单机、共享集群部署）

```
SELECT job, next_date, failures
FROM USER_JOBS
WHERE broken='N';
  JOB NEXT_DATE                           FAILURES
----- ------------------------------- ------------
 1616 2022-06-22 22:44:33.000000                 0
 1617 2022-06-22 22:44:38.000000                 0
 1618 2022-06-21 22:55:06.000000                 0

SELECT job_name,next_run_date,failure_count
FROM USER_SCHEDULER_JOBS
WHERE enabled=true
AND (START_DATE IS NULL OR SYSDATE>START_DATE)
AND (END_DATE IS NULL OR SYSDATE<END_DATE);
JOB_NAME             NEXT_RUN_DATE                   FAILURE_COUNT
-------------------- ------------------------------- -------------
DBMS_JOB$_1616       2022-06-22 22:44:33.000000                  0
DBMS_JOB$_1617       2022-06-22 22:44:38.000000                  0
SCHE_EXAMPLE         2022-06-21 22:55:06.000000                  0
```

# 触发器

触发器（Trigger）是数据库里的一种PL对象。创建一个触发器即创建了一个可执行的过程体，但与存储过程和函数不同的是，触发器过程体不可以被用户显式调用，而是由系统依据某个事件来触发执行。

一个触发器包含如下要素：

- 触发操作：触发执行的内容，为一个过程体。
- 触发事件：可以由系统判断的触发过程体执行的事件，即对表的INSERT/UPDATE/DELETE等DML操作。 可以定义单个触发事件，也可以定义多个触发事件的组合（OR逻辑组合） 。
- 触发时机： 触发过程体执行的时间点，包括BEFORE（触发事件发生前执行）和AFTER（触发事件发生后执行）两种。
- 触发对象：触发事件所基于的对象，即某张表。
- 触发类型：包括语句级触发（触发事件发生时，执行一次过程体）和行级触发（触发事件发生时，对其影响的每一行数据均执行一次过程体）两种类型。
- 触发条件：对于行级触发器，可以由WHEN语句指定一个条件表达式，在触发事件发生且条件表达式结果为true时，过程体才会被执行。

在表上定义触发器可以在对该表执行DML操作时，及时地进行一些错误拦截，或者操作记录，或者业务逻辑处理，相比存储过程的优势是实时性以及可以获得当时刻的数据信息，但其会对DML操作产生关联影响，如执行效率或本身问题导致DML操作失败等。

除触发器外，另一个在对表执行DML操作即被触发的功能为约束。约束是为了保证数据完整性而执行的字段级别的数据检查，相比约束，触发器使用的过程性语句可以实现更复杂的数据处理。对一个同时定义了约束和触发器的表执行DML操作时，系统的处理顺序为：

1. 执行BEFORE语句级触发器。

2. 对DML操作影响的每一行：

   2.1 执行BEFORE行级触发器。

   2.2 执行DML操作，同时执行非FOREIGN KEY的约束项检查。

   2.3 执行AFTER行级触发器。

   2.4 执行FOREIGN KEY检查。

3. 执行AFTER语句级触发器。

触发器仅适用于HEAP表。

## 创建触发器

包括触发器的定义和编译，如果触发器编译失败，依然会被创建，但是会在执行时出错。执行失败的触发器会阻止DML语句事件上的所有触发器，直到它被禁用、删除、或被重建且编译成功。

使用CREATE TRIGGER语句创建触发器，其语法定义为：

create_trigger**::=**

```
syntax::= CREATE [ OR REPLACE ] [EDITIONABLE|NONEDITIONABLE] TRIGGER [schema"."] trigger_name  simple_dml_trigger
```

simple_dml_trigger**::=**

```
syntax::= ( BEFORE | AFTER ) dml_event_clause [ referencing_clause ] [ FOR EACH ROW ]  [trigger_ordering_clause]
[ ENABLE | DISABLE ] [ WHEN ( condition ) ] trigger_body_clause
```

dml_event_clause**::=**

```
syntax::= ( DELETE | INSERT | (UPDATE [ OF (column){","(column)}] )){OR ( DELETE | INSERT | (UPDATE [ OF (column){","
(column)}] )} ON [ schema"."] ( table )
```

referencing_cause**::=**

```
syntax::= REFERENCING ( OLD [ AS ] old | NEW [ AS ] new ) {( OLD [ AS ] old | NEW [ AS ] new )}
```

trigger_ordering_clause**::=**

```
syntax::=  FOLLOWS  ( [ schema "." ]  trigger )  { ","  ( [ schema "." ]  trigger )}
```

trigger_body_clause**::=**

```
syntax::= [DECLARE variable_declare] BEGIN plsql_statements END
```

## or replace

当要创建的触发器已经存在时，将其进行重建。

## editionable | noneditionable

用于语法兼容，无实际含义。

## schema

包含触发器的模式名称，可省略，则默认为当前登录用户的模式。

## trigger_name

要创建的触发器的名称，在同一模式下的触发器不能有相同的名称。不可省略，且需符合YashanDB的对象命名规范。

## simple_dml_trigger

本语句用于定义触发器的各项要素。

### before|after

定义触发时机，在触发事件发生之前（之后）运行触发器。对于行级触发器，触发器在操作每个受影响的行之前（之后）运行。

### dml_event_clause

定义触发事件及触发对象。

触发器在表上创建，其触发事件可以是DML语句DELETE、INSERT、UPDATE的OR组合。

对于MERGE语句，可以通过在MERGE分解后的INSERT、UPDATE、DELETE语句上分别创建触发器，以实现对MERGE语句的触发操作。

**条件谓词**

由于触发事件可以由多个触发语句组合，当某一条语句导致触发事件发生时，触发器可以通过下述条件谓词来识别出具体是哪一条语句。

| 条件谓词 | 是TRUE的场景 |
| --- | --- |
| INSERTING | 由INSERT语句触发 |
| UPDATING | 由UPDATE语句触发 |
| UPDATING ('*column*') | 由指定列的UPDATE语句触发 |
| DELETING | 由DELETE语句触发 |

**delete**
从表中删除数据时运行触发器。

**insert**
向表中添加数据时运行触发器。

**update [ of column [, column ] ]**
更改指定列字段的值时运行触发器。未指定列字段时，更改表的任何列的值均会运行触发器。

不能指定为LOB类型的列字段。

**schema**
要在其上创建触发器的表的模式名称。可省略，则默认为当前登录用户的模式。

不能为SYS模式下的对象创建触发器。

**table**
要在其上创建触发器的表的名称。

不能为私有临时表创建触发器。

### referencing_clause

系统提供当前数据的新旧值供引用，默认使用OLD/NEW来识别，也可以通过本语句为其定义新的名称。当系统中存在名称为OLD或NEW的表对象时，应该通过本语句定义新的名称，以避免由于名称混淆出现错误。

### for each row

定义触发器为行级触发器，表示系统将为触发事件影响并满足触发条件的每一行数据执行一次触发器。 省略此子句时，触发器默认定义为语句级触发器，表示系统将在触发事件发生时执行一次触发器。

### trigger_order_clause

当为某个对象的某个触发事件定义了多个触发器时，使用本语句可以指定它们的相对运行关系。

FOLLOWS命令表示要创建的触发器必须在指定的触发器之后运行。指定的触发器必须存在，并且已成功编译，但是不需要启用。

### enable|disable

指定创建的触发器为启用或禁用状态，不指定则默认为启用。

### when (condition)

定义触发条件，只针对行级触发器，当在行上计算的condition为真时，则为这一行运行触发器。

需注意的是，触发条件影响的是触发器的运行，而不会影响触发事件，即DML语句的运行，无论触发条件结果为true还是false，DML语句都将执行。

condition的语法与通用SQL语法condition一致，但不能包含子查询和自定义函数。其中，对于列字段的引用需要按OLD/NEW（或由referencing_clause定义的名称）区分新旧值，引用格式为OLD.column，NEW.column。

### trigger_body_clause

定义触发操作，即一个过程体。

**declare variable_declare**
声明PL过程体的全局变量，可省略。

PL变量描述。

**pl_statements**
定义过程体中的执行语句，为PL语句中的一项或多项，但任何语句或由语句调用的子程序，均不可以包含DDL和DCL语句内容。

其中，对于触发对象的列字段的引用需要按OLD/NEW（或由referencing_clause定义的名称）区分新旧值，引用格式及取值定义见下表：

| | :OLD.column | :NEW.column |
|---|---|---|
| INSERT | NULL | INSERT的值 |
| DELETE | DELETE前的值 | NULL |

| | :OLD.column | :NEW.column |
| --- | --- | --- |
| UPDATE | UPDATE前的值 | UPDATE后的值 |

对于新旧值的引用还存在如下规则：

- 不能在语句级触发器中读写':NEW'和':OLD'，包括BEFORE和AFTER。
- 可以在BEFORE行级触发器中读或写':OLD'和':NEW'。
- 可以在AFTER行级触发器中读':OLD'和':NEW'。
- 不能在AFTER行级触发器中写':OLD'和':NEW'。

示例（HEAP表）

```
-- 在 product 表上创建行级前触发器t，在INSERT、DELETE、UPDATE时分别打印不同的信息。
-- 为old和new指定新的相关性名称。
CREATE OR REPLACE TRIGGER tri
    BEFORE
    INSERT OR UPDATE OF product_name, price OR DELETE
    ON product
    REFERENCING OLD AS told NEW AS tnew
    FOR EACH ROW
    ENABLE
    WHEN (told.product_no<>11003 OR tnew.product_no <> 11003)
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting. id is ' || :tnew.product_no);
    ELSIF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting. id is ' || :told.product_no);
    ELSIF UPDATING('price') THEN
        DBMS_OUTPUT.PUT_LINE('Updating price. id is ' || :tnew.product_no);
        DBMS_OUTPUT.PUT_LINE('Updating old price:' || :told.price);
        DBMS_OUTPUT.PUT_LINE('Updating new price:' || :tnew.price);
        :tnew.price := :tnew.price + 10;
        DBMS_OUTPUT.PUT_LINE('Updating final price:' || :tnew.price);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Updating Others. id is ' || :tnew.product_no);
    END IF;
END;
/

--执行DML语句
--1.主键约束项在BEFORE行级触发器之后运行
INSERT INTO product VALUES ('11001','product001',8,10);
Inserting. id is 11001
YAS-02030 unique constraint violated
--2.when条件为false，触发器不运行，但DML语句仍运行
DELETE product WHERE product_no=11001;
Deleting. id is 11001
ROLLBACK;
--3.更新price
UPDATE product SET price=20 WHERE product_no=11002;
Updating price. id is 11002
Updating old price:16
Updating new price:20
Updating final price:30
```

## 创建自治事务触发器

在上述创建触发器的语法中，由于触发事件的DML语句和触发操作的过程体处于同一事务中，所以在过程体中不允许执行DCL和DDL语句，以避免错误地提交或回滚。

当存在必须要提交事务的情况，例如触发器用于操作记录，记录的日志表为另一张独立表且在获取到触发事件的新旧值、系统时间等信息就应该及时提交，此时可以通过创建自治事务触发器来将触发操作独立成一个事务来运行。

定义自治事务触发器的语句为：*pragma autonomous_transaction;*。

示例（HEAP表）

```
--创建记录日志表
CREATE TABLE product_log
(product_no CHAR(5),
 update_date DATE,
 newprice NUMBER,
 oldprice NUMBER);

-- 自治事务触发器。在 product 表上创建行级前触发器 log_product，在更新 product.price 列时将旧值和新值写入product_log表。
CREATE OR REPLACE TRIGGER log_product
    BEFORE
    DELETE
    ON product
    FOR EACH ROW
DECLARE
PRAGMA autonomous_transaction;
BEGIN
    DBMS_OUTPUT.PUT_LINE('old.price is ' || :old.price);
    DBMS_OUTPUT.PUT_LINE('new.price is ' || :NEW.price);
    INSERT INTO product_log VALUES (:old.product_no, SYSDATE, :NEW.price, :old.price);
    COMMIT;
END;
/

--删除product表的数据，触发log_product及tri，日志表将独立提交
DELETE product WHERE product_no=11002;
Deleting, id is 11002

--回滚DML操作，product数据被恢复，但日志表仍记录当时操作
ROLLBACK;
SELECT price FROM product WHERE product_no=11002;
      PRICE
-----------
         30
SELECT * FROM product_log  WHERE product_no=11002;
PRODUCT_NO UPDATE_DATE                       NEWPRICE    OLDPRICE
---------- ------------------------------- ----------- -----------
11002      2022-06-26                                        16
```

## 启用/禁用触发器

使用ALTER TRIGGER语句可以启用或禁用一个已创建的触发器。

## 删除触发器

使用DROP TRIGGER语句可以删除一个已存在的触发器。

此外，当触发器的触发对象，即表被删除（DROP TABLE）时，该触发器也将被删除。

# 自定义库

自定义库是数据库中的一种PL对象，用于数据库调用第三方库文件。

分布式部署中不能创建自定义库。

## 创建自定义库

使用CREATE LIBRARY语句创建自定义库，其语法定义为：

**create library::=**

```
syntax::= CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] LIBRARY plsql_library_source ";"
```

**plsql_library_source::=**

```
syntax::= [ schema "."] library_name ( IS | AS ) "'" full_path_name "'"
```

### or replace

当要创建的自定义库已经存在时，将其进行重建。

### editionable | noneditionable

用于语法兼容，无实际含义。

### library_name

自定义库的名称，不可省略，且需要符合YashanDB的对象命名规范。

### 'full_path_name'

库文件的完整路径，需要为字符串，支持绝对路径和相对路径。长度需在0-255字节范围，如'/home/yasdb/example/UDFexample.class'。

示例（单机、共享集群部署）

```
CREATE OR REPLACE LIBRARY ya_lib IS
'/home/yasdb/example/UDFexample.class';
/
```

## 使用自定义库

外置UDF中可使用call_spec子句调用自定义库，详细使用方法见外置UDF。

示例（单机、共享集群部署）

```
CREATE OR REPLACE FUNCTION udf_func(argu INT) RETURN VARCHAR IS
LANGUAGE java
NAME 'example.UDFexample.execJdbcexample(int) return string'
LIBRARY ya_lib;
/
```

## 删除自定义库

使用DROP_LIBRARY语句删除自定义库。

# 外置UDF

YashanDB支持在PL中调用外部Java方法或者外部C函数，实现数据库中的外置UDF功能。分布式部署中不可使用此功能。

## 环境要求

使用外置UDF（JAVA）要求数据库服务器已安装JDK（1.8及其以上版本），并配置如下环境变量：

```
#如下路径需更换为实际的jdk安装路径
export LD_LIBRARY_PATH=/etc/jdk-18.0.2/lib/server:$LD_LIBRARY_PATH
```

## 创建自定义库

使用外置UDF首先需要将外置UDF所需要的自定义库创建到数据库中。

Java语言的外置UDF，YashanDB支持创建包含自定义Java函数的.class文件或者jar包的自定义库。

C语言的外置UDF，YashanDB支持创建包含C函数的.dll（Windows）和.so（Linux）动态库文件。

YashanDB通过CREATE LIBRARY中的CREATE LIBRARY语句创建自定义库。

创建自定义库时不检查库文件是否存在，在执行外置UDF时才检查并加载库文件。

示例（单机、共享集群部署）

```
--创建Java语言的自定义库
CREATE OR REPLACE LIBRARY ya_java_lib IS
'/home/yasdb/example/UDFexample.class';
/
```

UDFexample.class对应UDFexample.java文件内容如下：

```
package example;

public class UDFexample {
    public static String execJdbcexample(int ctrls) {
        switch (ctrls) {
        case 1:return "Hello";
        case 2:return "World";
        default:return "!";
        }
    }

    public static void main(String[] args) {
       String a = execJdbcexample(1);
    }
}
```

```
--创建C语言的自定义库
CREATE OR REPLACE LIBRARY ya_c_lib IS
'/home/yasdb/example/libUDFexample.so';
/
```

libUDFexample.so对应的UDFexample.c文件内容如下：

```
#include "string.h"
#include "yacli.h"

YacResult udfExample(YacHandle hProc)
{
    YacInt32  id;
```

```
    YacChar   buf[1024];

    YAC_CALL(yepGetInt32(hProc, 0, &id, NULL));
    (YacVoid)snprintf(buf, 1024, "Hello World, id: %d", id);
    YAC_CALL(yepReturnString(hProc, buf));

    return YAC_SUCCESS;
}
```

当class文件、jar包、so文件或dll文件内容发生变化时，需重新生成自定义库。

## 定义Java语言的外置UDF

通过创建一个函数，实现对Java语言的外置UDF的定义，语法如下：

```
syntax::= CREATE [OR REPLACE] FUNCTION [schema "."] function_name ["("  (argument_define) {"," (argument_define)}  ")"]
RETURN return_datatype (IS|AS) call_spec ";"
```

其中，为实现向外置UDF传递参数和获得返回值，对函数定义的参数和返回值应与java_method里的参数和返回值一致，此处一致的具体含义为：

- 参数个数及顺序一致。
- 数据类型兼容，即显式地满足YashanDB和java数据类型对应关系，或者通过隐式转换后满足对应关系，对应关系见下面描述。

```
syntax::= LANGUAGE JAVA NAME "java_string_literal_name" LIBRARY ([schema"."]library_name | "'" library_path "'")
```

### LANGUAGE

指定外置UDF所用的语言，这里为JAVA。

### NAME

指定需引用的java函数的函数签名，java_package.java_class部分长度需小于128字节，其格式为：

*java_package.java_class.java_method(argu1_datatype,argu2_datatype...) [return return_datatype]*

此处引用的java函数必须为static的java方法，且该方法的参数和返回类型必须在如下列示的java数据类型范围内：

- bool：对应YashanDB的BOOLEAN
- byte：对应YashanDB的TINYINT
- char：对应YashanDB的INT（取值范围[0,65535]，即0-16位无符号整数）
- short：对应YashanDB的SMALLINT
- int：对应YashanDB的INT
- long：对应YashanDB的BIGINT
- float：对应YashanDB的FLOAT
- double：对应YashanDB的DOUBLE
- string：对应YashanDB的VARCHAR

### library_name

指定已经创建的自定义库名称。

### library_path

通过LOADJAVA高级包程序加载的自定义库路径，YashanDB已提供直接创建自定义库的功能，此方式不再建议使用。

示例（单机、共享集群部署）

```
CREATE OR REPLACE FUNCTION udf_func_java(argu INT) RETURN VARCHAR IS
LANGUAGE java
NAME 'example.UDFexample.execJdbcexample(int) return string'
LIBRARY ya_java_lib;
/

CREATE OR REPLACE FUNCTION udf_func_java(argu INT) RETURN VARCHAR IS
```

```
LANGUAGE java
NAME 'example.UDFexample.execJdbcexample(int) return string'
LIBRARY '/home/yasdb/example/UDFexample.class';
/
```

## 定义C语言的外置UDF

通过创建一个函数，实现对C语言的外置UDF的定义，语法如下：

```
syntax::= CREATE [OR REPLACE] FUNCTION [schema "."] function_name ["("  (argument_define) {"," (argument_define)}  ")"]
RETURN return_datatype (IS|AS) call_spec ";"
```

**call_spec::=**

```
syntax::= (LANGUAGE C | EXTERNAL) (([NAME c_string_literal_name] LIBRARY [schema"."] library_name)|(LIBRARY [schema"."]
library_name [NAME c_string_literal_name]))
```

### language

指定外置UDF所用的语言，这里为C。

### external

指定为C语言的外置UDF（建议使用LANGUAGE指定外置UDF所使用的语言）。

### name

通过c_string_literal_name指定需引用的C函数的函数名，默认转为全大写，如需保留大小写，请使用双引号包围，如果省略NAME，默认使用外置UDF的名称。

c_string_literal_name需要为合法的标识符，其最大长度为64字节。

### library_name

指定已经创建的自定义库名称。

示例（单机、共享集群部署）

```
CREATE OR REPLACE FUNCTION udf_func_c(argu INT) RETURN VARCHAR IS
LANGUAGE C
NAME "udfExample"
LIBRARY ya_c_lib;
/
```

## 编写c语言外置udf指导

C语言外置UDF是调用C语言编写的函数，将C语言文件编译为动态库文件后，在创建为自定义库被数据库调用。

C函数有唯一参数hProc，不可修改其值。通过YashanDB提供的一系列yepGet、yepOutPut、yepReturn接口实现获取入参、设置出参、设置返回值功能。

使用yepGet接口时，需根据入参类型调用相同类型或可以隐式转换类型的yepGet接口。

使用yepOutPut接口时，需根据入参类型调用相同类型的yepOutPut接口。

使用yepReturn接口时，需根据返回值类型调用相同类型或者可以隐式转换类型的yepReturn接口，且自定义函数必须调用yepReturn接口设置返回值。

接口参数id为函数定义的参数序号，起始值为0。

接口参数lenOrInd用于获取入参长度或是否为NULL值，若其值返回YAC_NULL_DATA，则说明入参为NULL，否则返回入参的字节长度。lenOrInd可为NULL。

YashanDB提供给编写C语言动态库的接口如下：

| 接口名称 | 接口函数声明 | 接口使用说明 |
| --- | --- | --- |
| yepGetCharsetId | YacResult yepGetCharsetId(YacHandle hProc, YacUint16* charsetId) | 获取数据库的字符集编号，赋值给charsetId。 |
| yepGetBool | YacResult yepGetBool(YacHandle hProc, YacInt32 id, YacBool* v, YacInt32* lenOrInd) | 获取第id个BOOLEAN类型入参的值，赋值给v。 |
| yepGetInt8 | YacResult yepGetInt8(YacHandle hProc, YacInt32 id, YacInt8* v, YacInt32* lenOrInd) | 获取第id个TINYINT类型入参的值，赋值给v。 |
| yepGetInt16 | YacResult yepGetInt16(YacHandle hProc, YacInt32 id, YacInt16* v, YacInt32* lenOrInd) | 获取第id个SMALLINT类型入参的值，赋值给v。 |
| yepGetInt32 | YacResult yepGetInt32(YacHandle hProc, YacInt32 id, YacInt32* v, YacInt32* lenOrInd) | 获取第id个INT类型入参的值，赋值给v。 |
| yepGetInt64 | YacResult yepGetInt64(YacHandle hProc, YacInt32 id, YacInt64* v, YacInt32* lenOrInd) | 获取第id个BIGINT类型入参的值，赋值给v。 |
| yepGetFloat | YacResult yepGetFloat(YacHandle hProc, YacInt32 id, YacFloat* v, YacInt32* lenOrInd) | 获取第id个FLOAT类型入参的值，赋值给v。 |
| yepGetDouble | YacResult yepGetDouble(YacHandle hProc, YacInt32 id, YacDouble* v, YacInt32* lenOrInd) | 获取第id个DOUBLE类型入参的值，赋值给v。 |
| yepGetNumber | YacResult yepGetNumber(YacHandle hProc, YacInt32 id, YacNumber* v, YacInt32* lenOrInd) | 获取第id个NUMBER类型入参的值，赋值给v。 |
| yepGetDate | YacResult yepGetDate(YacHandle hProc, YacInt32 id, YacDate* v, YacInt32* lenOrInd) | 获取第id个DATE类型入参的值，赋值给v。 |
| yepGetTimestamp | YacResult yepGetTimestamp(YacHandle hProc, YacInt32 id, YacTimestamp* v, YacInt32* lenOrInd) | 获取第id个TIMESTAMP类型入参的值，赋值给v。 |
| yepGetYMInterval | YacResult yepGetYMInterval(YacHandle hProc, YacInt32 id, YacYMInterval* v, YacInt32* lenOrInd) | 获取第id个INTERVAL YEAR TO MONTH类型入参的值，赋值给v。 |
| yepGetDSInterval | YacResult yepGetDSInterval(YacHandle hProc, YacInt32 id, YacDSInterval* v, YacInt32* lenOrInd) | 获取第id个INTERVAL DAY TO SECOND类型入参的值，赋值给v。 |
| yepGetString | YacResult yepGetString(YacHandle hProc, YacInt32 id, YacChar* str, YacUint32 bufSize, YacInt32* lenOrInd) | 获取第id个字符串类型（CHAR、VARCHAR）入参的值，需要提前分配内存，传入指针str和bufSize。 |
| yepGetBytes | YacResult yepGetBytes(YacHandle hProc, YacInt32 id, YacUint8* bytes, YacUint32 bufSize, YacInt32* lenOrInd) | 获取第id个RAW类型入参的值，需要提前分配内存，传入指针bytes和bufSize。 |
| yepOutputNull | YacResult yepOutputNull(YacHandle hProc, YacInt32 id) | 将第id个出参设置为NULL。 |
| yepOutputBool | YacResult yepOutputBool(YacHandle hProc, YacInt32 id, YacBool v) | 将第id个BOOLEAN类型出参的值设置为v。 |
| yepOutputInt8 | YacResult yepOutputInt8(YacHandle hProc, YacInt32 id, YacInt8 v) | 将第id个TINYINT类型出参的值设置为v。 |
| yepOutputInt16 | YacResult yepOutputInt16(YacHandle hProc, YacInt32 id, YacInt16 v) | 将第id个SMALLINT类型出参的值设置为v。 |
| yepOutputInt32 | YacResult yepOutputInt32(YacHandle hProc, YacInt32 id, YacInt32 v) | 将第id个INT类型出参的值设置为v。 |
| yepOutputInt64 | YacResult yepOutputInt64(YacHandle hProc, YacInt32 id, YacInt64 v) | 将第id个BIGINT类型出参的值设置为v。 |
| yepOutputFloat | YacResult yepOutputFloat(YacHandle hProc, YacInt32 id, YacFloat v) | 将第id个FLOAT类型出参的值设置为v。 |
| yepOutputDouble | YacResult yepOutputDouble(YacHandle hProc, YacInt32 id, YacDouble v) | 将第id个DOUBLE类型出参的值设置为v。 |

| 接口名称 | 接口函数声明 | 接口使用说明 |
| --- | --- | --- |
| yepOutputNumber | YacResult yepOutputNumber(YacHandle hProc, YacInt32 id, YacNumber* v) | 将第id个NUMBER类型出参的值设置为v。 |
| yepOutputDate | YacResult yepOutputDate(YacHandle hProc, YacInt32 id, YacDate v) | 将第id个DATE类型出参的值设置为v。 |
| yepOutputTimestamp | YacResult yepOutputTimestamp(YacHandle hProc, YacInt32 id, YacTimestamp* v) | 将第id个TIMESTAMP类型出参的值设置为v。 |
| yepOutputYMInterval | YacResult yepOutputYMInterval(YacHandle hProc, YacInt32 id, YacYMInterval v) | 将第id个INTERVAL YEAR TO MONTH类型出参的值设置为v。 |
| yepOutputDSInterval | YacResult yepOutputDSInterval(YacHandle hProc, YacInt32 id, YacDSInterval v) | 将第id个INTERVAL DAY TO SECOND类型出参的值设置为v。 |
| yepOutputString | YacResult yepOutputString(YacHandle hProc, YacInt32 id, YacChar* str) | 将第id个字符串类型（CHAR、VARCHAR）类型出参的值设置为str指向的字符串。 |
| yepOutputBytes | YacResult yepOutputBytes(YacHandle hProc, YacInt32 id, YacUint8* bytes, YacUint32 size) | 将第id个RAW类型出参的值设置为bytes指向的字节，大小为size。 |
| yepReturnNull | #define yepReturnNull(hProc) yepOutputNull(hProc, YEP_RETURN) | 将返回值设置为NULL。 |
| yepReturnBool | #define yepReturnBool(hProc, value) yepOutputBool(hProc, YEP_RETURN, value) | 将返回值设置为BOOLEAN类型，值为value。 |
| yepReturnInt8 | #define yepReturnInt8(hProc, value) yepOutputInt8(hProc, YEP_RETURN, value) | 将返回值设置为TINYINT类型，值为value。 |
| yepReturnInt16 | #define yepReturnInt16(hProc, value) yepOutputInt16(hProc, YEP_RETURN, value) | 将返回值设置为SMALLINT类型，值为value。 |
| yepReturnInt32 | #define yepReturnInt32(hProc, value) yepOutputInt32(hProc, YEP_RETURN, value) | 将返回值设置为INT类型，值为value。 |
| yepReturnInt64 | #define yepReturnInt64(hProc, value) yepOutputInt64(hProc, YEP_RETURN, value) | 将返回值设置为BIGINT类型，值为value。 |
| yepReturnFloat | #define yepReturnFloat(hProc, value) yepOutputFloat(hProc, YEP_RETURN, value) | 将返回值设置为FLOAT类型，值为value。 |
| yepReturnDouble | #define yepReturnDouble(hProc, value) yepOutputDouble(hProc, YEP_RETURN, value) | 将返回值设置为DOUBLE类型，值为value。 |
| yepReturnNumber | #define yepReturnNumber(hProc, value) yepOutputNumber(hProc, YEP_RETURN, value) | 将返回值设置为NUMBER类型，值为value。 |
| yepReturnDate | #define yepReturnDate(hProc, value) yepOutputDate(hProc, YEP_RETURN, value) | 将返回值设置为DATE类型，值为value。 |
| yepReturnTimestamp | #define yepReturnTimestamp(hProc, value) yepOutputTimestamp(hProc, YEP_RETURN, value) | 将返回值设置为TIMESTAMP类型，值为value。 |
| yepReturnYMInterval | #define yepReturnYMInterval(hProc, value) yepOutputYMInterval(hProc, YEP_RETURN, value) | 将返回值设置为INTERVAL YEAR TO MONTH类型，值为value。 |
| yepReturnDSInterval | #define yepReturnDSInterval(hProc, value) yepOutputDSInterval(hProc, YEP_RETURN, value) | 将返回值设置为INTERVAL DAY TO SECOND类型，值为value。 |
| yepReturnString | #define yepReturnString(hProc, value) yepOutputString(hProc, YEP_RETURN, value) | 将返回值设置为VARCHAR类型，值为value指向的字符串。 |
| yepReturnBytes | #define yepReturnBytes(hProc, value, size) yepOutputBytes(hProc, YEP_RETURN, value, size) | 将返回值设置为RAW类型，值为bytes指向的字节，大小为size。 |

## 运行外置udf

对于已成功定义的外置UDF，其调用方法与函数一致。

示例（单机、共享集群部署）

```
SELECT udf_func_java(1) FROM dual;
UDF_FUNC_JAVA(1)
----------------------------------------------------------------
Hello

SELECT udf_func_java(2) FROM dual;
UDF_FUNC_JAVA(2)
----------------------------------------------------------------
World

SELECT udf_func_c(1) FROM dual;
UDF_FUNC_C(1)
----------------------------------------------------------------
Hello World, id: 1
```

## 删除外置udf

删除定义了某个外置UDF的函数，即实现对这个外置UDF的删除。

## 删除自定义库

确认外置UDF使用的自定义库不再使用时，可对其进行删除，使用DROP_LIBRARY语句删除自定义库。

# 内置高级包

YashanDB提供一系列内置高级包，其中封装了与系统逻辑相关的各子程序供调用，按程序使用用途分类如下：

- 系统标准功能、事务处理相关

  DBMS_LOB

  DBMS_LOCK

  DBMS_METADATA

  DBMS_MVIEW

  DBMS_OUTPUT

  DBMS_RANDOM

  DBMS_ROWID

  DBMS_STANDARD

  DBMS_SQL

  UTL_FILE

  UTL_ENCODE

- 定时任务相关

  DBMS_IJOB

  DBMS_JOB

  DBMS_SCHEDULER

- 故障诊断相关

  DBMS_HM

- 性能优化相关

  DBMS_AWR

  DBMS_STATS

  DBMS_PARAM

- 资源管理相关

  DBMS_RESOURCE_MANAGER

- 审计相关

  DBMS_AUDIT_MGMT

- 实用程序相关

  DBMS_UTILITY

  OWA_UTIL

  DBMS_PICKLER

基于上述内置高级包，用户可以在SQL客户端执行实时的功能操作，具体参考各高级包的功能介绍，也可以使用自定义高级包功能进行更多定制开发。

> **Note**:
> 对于列存表，只允许在过程体中调用内置高级包。

# DBMS_AUDIT_MGMT

DBMS_AUDIT_MGMT包提供了一组内置的存储过程，用于创建和管理审计清理任务。需要注意的是，所有相关程序的参数值必须按序输入。

分布式部署中，清理审计日志需使用sys用户在每个节点本地进行操作。

## SET_LAST_ARCHIVE_TIMESTAMP

```
DBMS_AUDIT_MGMT.SET_LAST_ARCHIVE_TIMESTAMP(
   audit_trail_type IN INTEGER,
   last_archive_time IN TIMESTAMP);
```

SET_LAST_ARCHIVE_TIMESTAMP程序用于设置清理时间点，成功设置的清理时间点可以在DBA_AUDIT_MGMT_LAST_ARCH_TS视图中查询。

| 参数 | 描述 |
| --- | --- |
| audit_trail_type | 审计清理类型，当前只支持DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED。 |
| last_archive_time | 清理时间点，不能超过当前时间。 |

示例

```
--设置清理时间点
BEGIN
    DBMS_AUDIT_MGMT.SET_LAST_ARCHIVE_TIMESTAMP(
        DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED ,
        SYSDATE);
END;
/
```

## CLEAN_AUDIT_TRAIL

```
DBMS_AUDIT_MGMT.CLEAN_AUDIT_TRAIL(
    audit_trail_type IN INTEGER,
    use_last_arch_timestamp IN BOOLEAN DEFAULT TRUE);
```

CLEAN_AUDIT_TRAIL程序用于进行审计日志内容清理。

| 参数 | 描述 |
| --- | --- |
| audit_trail_type | 审计清理类型，当前只支持DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED。 |
| use_last_arch_timestamp | 是否使用清理时间，值为TRUE或FALSE。若为TRUE则只删除小于清理时间点的审计数据，若为FALSE则删除所有审计数据。 |

示例

```
--执行审计日志清理，清除审计清理时间点之前的审计数据。
BEGIN
  DBMS_AUDIT_MGMT.CLEAN_AUDIT_TRAIL (
        DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED,
        true);
END;
/
```

## CREATE_PURGE_JOB

```
DBMS_AUDIT_MGMT.CREATE_PURGE_JOB(
    audit_trail_type IN INTEGER,
    audit_trail_start_time IN timestamp,
    audit_trail_purge_interval IN varchar,
    audit_trail_purge_name IN VARCHAR,
    use_last_arch_timestamp IN BOOLEAN DEFAULT TRUE);
```

CREATE_PURGE_JOB程序用于创建一个审计清理定时任务，成功创建的定时任务将可以在DBA_AUDIT_MGMT_CLEANUP_JOBS视图中查询，审计清理定时任务创建时默认状态为开启。

| 参数 | 描述 |
| --- | --- |
| audit_trail_type | 审计清理类型，当前只支持DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED。 |
| audit_trail_start_time | 定时任务开始时间。 |
| audit_trail_purge_interval | 表达式文本，用于计算定时任务下次执行的时间。 |
| audit_trail_purge_name | 定时任务名称。 |
| use_last_arch_timestamp | 是否使用清理时间，值为TRUE或FALSE。若为TRUE则只有小于清理时间点的数据被删除，若为FALSE则会清理所有的审计记录。 |

**定期更新审计清理时间点**

使用审计清理定时任务处理审计数据清理时，默认将清理时间点之前的审计数据做清理。在数据库使用期间，审计日志不断积累，需要根据实际需要定期更新清理时间点。 可以通过YashanDB提供的DBMS_SCHEDULER.CREATE_JOB进行定期更新审计清理时间点。所创建的任务计划可在DBA_SCHEDULER_JOBS中查询。

示例

```
--将审计清理时间点设置为30天前，即只保留最近30天的审计日志，每天执行一次刷新
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
     'update_audit_archive_time',
    'PLSQL_BLOCK',
    'BEGIN DBMS_AUDIT_MGMT.SET_LAST_ARCHIVE_TIMESTAMP(DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED, sysdate-30);END;' ,
    0,
    SYSDATE,
    'sysdate+1',
    NULL,
    'DEFAULT_JOB_CLASS',
    TRUE,
    FALSE,
    'update audit archive time');
END;
/
```

**创建审计清理定时任务**

创建审计清理时间点定时任务后，创建审计清理定时任务。

示例

```
--创建自动清理任务，5个小时后开始第一次执行，每次执行时间间隔为1天
BEGIN
  DBMS_AUDIT_MGMT.CREATE_PURGE_JOB (
      DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED,
      SYSDATE + 5/24,
      'sysdate + 1',
      'audit_job',
     TRUE);
END;
/
```

## SET_PURGE_JOB_STATUS

```
DBMS_AUDIT_MGMT.SET_PURGE_JOB_STATUS(
    audit_trail_purge_name IN VARCHAR,
    audit_trail_status_value IN INTEGER);
```

SET_PURGE_JOB_STATUS程序用于设置定时任务的启停标志，被停止的定时任务将不再被系统进行任务调度。

| 参数 | 描述 |
| --- | --- |
| audit_trail_purge_name | 定时任务名称，可以通过DBA_AUDIT_MGMT_CLEANUP_JOBS视图查询。 |
| audit_trail_status_value | 是否停止定时任务，DBMS_AUDIT_MGMT.PURGE_JOB_ENABLE表示启动，DBMS_AUDIT_MGMT.PURGE_JOB_DISABLE表示停止。 |

示例

```
--停止审计清理定时任务
BEGIN
  DBMS_AUDIT_MGMT.SET_PURGE_JOB_STATUS (
        'audit_job',
        DBMS_AUDIT_MGMT.PURGE_JOB_DISABLE);
END;
/
```

## SET_PURGE_JOB_INTERVAL

示例

```
DBMS_AUDIT_MGMT.SET_PURGE_JOB_INTERVAL(
    audit_trail_purge_name IN VARCHAR,
    audit_trail_interval_value IN VARCHAR);
```

SET_PURGE_JOB_INTERVAL程序用于修改审计日志定时任务下次执行时间。

| 参数 | 描述 |
| --- | --- |
| audit_trail_purge_name | 定时任务名称，可以通过DBA_AUDIT_MGMT_CLEANUP_JOBS视图查询。 |
| audit_trail_interval_value | 表达式文本，用于计算定时任务下次执行的时间。 |

示例

```
--修改审计清理定时任务的执行间隔为7天
BEGIN
DBMS_AUDIT_MGMT.SET_PURGE_JOB_INTERVAL(
     'audit_job',
     'sysdate + 7');
END;
/
```

## GET_LAST_ARCHIVE_TIMESTAMP

```
DBMS_AUDIT_MGMT.GET_LAST_ARCHIVE_TIMESTAMP(
    audit_trail_type IN PLS_INTEGER)
```

```
RETURN TIMESTAMP;
```

GET_LAST_ARCHIVE_TIMESTAMP程序获取审计清理时间点。

| 参数 | 描述 |
| --- | --- |
| audit_trail_type | 审计清理类型，当前只支持DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED。 |

示例

```
--获取审计清理时间点
DECLARE
 last_time TIMESTAMP;
BEGIN
 last_time := DBMS_AUDIT_MGMT.GET_LAST_ARCHIVE_TIMESTAMP(DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED);
 IF last_time IS NOT NULL THEN
  DBMS_OUTPUT.PUT_LINE('last archive timestamp is: ' || TO_CHAR(last_time));
 ELSE
   DBMS_OUTPUT.PUT_LINE('last archive timestamp is null');
 END IF;
END;
/

--result
last archive timestamp is: 2022-11-05 17:12:34.000000
```

## CLEAR_LAST_ARCHIVE_TIMESTAMP

```
DBMS_AUDIT_MGMT.CLEAR_LAST_ARCHIVE_TIMESTAMP(
    audit_trail_type IN PLS_INTEGER);
```

CLEAR_LAST_ARCHIVE_TIMESTAMP程序用于清空审计清理时间点。

| 参数 | 描述 |
| --- | --- |
| audit_trail_type | 审计清理类型，当前只支持DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED。 |

示例

```
--清空审计清理时间点
BEGIN
    DBMS_AUDIT_MGMT.CLEAR_LAST_ARCHIVE_TIMESTAMP
    (DBMS_AUDIT_MGMT.AUDIT_TRAIL_UNIFIED);
END;
/
```

**删除审计清理定时任务**

```
DBMS_AUDIT_MGMT.DROP_PURGE_JOB (
    audit_trail_purge_name IN VARCHAR);
```

DROP_PURGE_JOB程序用于删除清理定时任务。

| 参数 | 描述 |
| --- | --- |
| audit_trail_purge_name | 定时任务名称。 |

## 异常说明

调用DBMS_AUDIT_MGMT高级包进行审计清理相关处理时，若参数设置不正确会抛出如下异常：

| 错误码 | 错误内容 | 解释 |
| --- | --- | --- |
| YAS-30000 | invalid value of argument AUDIT_TRAIL_TYPE | 审计清理类型的设置非法 |
| YAS-30001 | invalid value of argument AUDIT_TRAIL_STATUS_VALUE | 审计清理job状态的设置非法 |
| YAS-30002 | invalid value of argument LAST_ARCHIVE_TIME | 审计清理时间点的设置非法 |
| YAS-30003 | invalid value of argument AUDIT_TRAIL_PURGE_INTERVAL | 下次执行时间的设置非法 |
| YAS-30004 | Cleanup job already existed for the given audit trail type | 给定类型的审计清理job已存在 |

# DBMS_AWR

DBMS_AWR包提供了一组内置的存储过程/函数，用于实现性能报告相关功能。

调用DBMS_AWR高级包下的所有子程序时，都需要以SYS用户连接数据库，否则报错。

## CREATE_SNAPSHOT

```
DBMS_AWR.CREATE_SNAPSHOT ();
```

CREATE_SNAPSHOT子程序为存储过程，无参数，用于创建一个快照。

成功执行本程序后，系统将在WRM$_SNAPSHOT表中新增一条快照记录，该记录包含快照ID，快照时间等信息。

示例（单机、共享集群部署）

```
--创建一次快照
EXEC DBMS_AWR.CREATE_SNAPSHOT();
--继续创建一次快照
EXEC DBMS_AWR.CREATE_SNAPSHOT();

--从WRM$_SNAPSHOT表中查询最近的两次快照信息，包括数据库ID、快照ID、实例标识等信息
SELECT * FROM
(SELECT dbid,snap_id,instance_number FROM wrm$_snapshot ORDER BY snap_id DESC)
WHERE ROWNUM<3;
       DBID     SNAP_ID INSTANCE_NUMBER
----------- ----------- ---------------
 2621752453         169               1
 2621752453         168               1
```

## AWR_REPORT

```
DBMS_AWR.AWR_REPORT(
    l_dbid     IN NUMBER,
    l_inst_num IN NUMBER,
    l_bid      IN NUMBER,
    l_eid      IN NUMBER,
    l_options  IN NUMBER   DEFAULT 0,
    l_format   IN VARCHAR2 DEFAULT 'html');
```

AWR_REPORT子程序为存储过程，用于生成性能报告。

本程序在成功执行后，将直接输出报告的文本内容，用户可将输出内容粘贴到一个html文件，用于查看和保存。

| 参数 | 描述 |
| --- | --- |
| l_dbid | 数据库ID |
| l_inst_num | 实例标识符，该参数单机部署下值为1，集群下在实例上查V$INSTANCE.INSTANCE_NUMBER |
| l_bid | 起始快照ID |
| l_eid | 终止快照ID |
| l_options | 保留字段 |
| l_format | 保留字段 |

示例（单机、共享集群部署）

```
--从WRM$_SNAPSHOT表获取两个snap_id值
EXEC DBMS_AWR.AWR_REPORT(2621752453,1,168,169);
```

## CLEAN_SNAPSHOT

```
DBMS_AWR.CLEAN_SNAPSHOT ();
```

CLEAN_SNAPSHOT子程序为存储过程，无参数，用于检查之前生成的快照是否超过设置的保存时间（默认是7天），超过则进行清理。

示例（单机、共享集群部署）

```
EXEC DBMS_AWR.CLEAN_SNAPSHOT();
```

## DROP_SNAPSHOT_RANGE

```
DBMS_AWR.DROP_SNAPSHOT_RANGE (
    low_snap_id      IN NUMBER,
    high_snap_id     IN NUMBER,
    dbid             IN NUMBER DEFAULT NULL);
```

DROP_SNAPSHOT_RANGE子程序为存储过程，通过给定参数确定所需要删除快照的范围，删除相应区间的快照。

| 参数 | 描述 |
| --- | --- |
| low_snap_id | 快照起始ID |
| high_snap_id | 快照终止ID |
| dbid | 数据库ID，可省略，则自动填充当前的数据库ID |

示例（单机、共享集群部署）

```
EXEC DBMS_AWR.DROP_SNAPSHOT_RANGE(35, 40);
```

## MODIFY_SNAPSHOT_SETTINGS

```
DBMS_AWR.MODIFY_SNAPSHOT_SETTINGS(
    retention       IN NUMBER   DEFAULT NULL,
    interval        IN NUMBER   DEFAULT NULL,
    dbid            IN NUMBER   DEFAULT NULL);
```

MODIFY_SNAPSHOT_SETTINGS子程序为存储过程，通过修改快照的相关属性，控制快照的生成时间间隔（用于生成快照的自动任务中）及保存时间。

| 参数 | 描述 |
| --- | --- |
| retention | 快照保存时间，单位分钟，如保存一天可输入：24*60（最小值一天，最大值100年） |
| interval | 快照生成时间间隔，单位分钟，如20分钟生成一次快照，可输入：20（最小值10分钟，最大值100年） |
| dbid | 数据库ID，可省略，则自动填充当前的数据库ID |

示例（单机、共享集群部署）

```
--将快照保存时间设为3天，将快照生成间隔时间设为2小时
EXEC DBMS_AWR.MODIFY_SNAPSHOT_SETTINGS(3*24*60,2*60);
```

# DBMS_HM

DBMS_HM包提供了一组内置的存储过程/函数，用于实现YashanDB的健康检查相关功能。

## RUN_CHECK

```
DBMS_HM.RUN_CHECK (
    check_name   IN   VARCHAR,
    run_name     IN   VARCHAR DEFAULT 'NULL',
    input_params IN   VARCHAR DEFAULT 'NULL');
```

RUN_CHECK为一个存储过程，通过给定参数运行指定的检查器，执行健康检查。

| 参数 | 描述 |
| --- | --- |
| check_name | 要调用的检查器名称。通过V$HM_CHECK视图可以查询到当前系统中所有的检查器信息。 |
| run_name | 用户指定的唯一标识此检查运行的名称，为NULL时，系统根据当前健康检查的ID创建一个默认名称。通过V$HM_RUN视图可以查看该名称信息。 |
| input_params | 检查器要求的参数信息，由名称=值对组成，多参数使用 ; 分隔。（示例：'Data Block Integrity Check'检查器可能的输入参数：'BLC_DF_NUM=1;BLC_BL_NUM=23456'）<br>每个检查器都具有与之关联的明确定义的输入参数。这些输入参数及其类型、默认值和描述可以通过V$HM_CHECK_PARAM视图获取。（示例：查询'Data Block Integrity Check' 检查器的参数信息：SELECT a.* FROM V$HM_CHECK_PARAM a, V$HM_CHECK b WHERE a.check_id = b.id AND b.name = 'Data Block Integrity Check';） |

示例

```
BEGIN
  DBMS_HM.RUN_CHECK('Redo File Check','hm1','RF_NUM=1');
END;
/
```

## GET_RUN_REPORT

```
DBMS_HM.GET_RUN_REPORT (
    run_name IN VARCHAR);
```

GET_RUN_REPORT为一个函数，返回指定检查器运行的报告。

| 参数 | 描述 |
| --- | --- |
| check_name | 检查运行的名称。通过V$HM_RUN视图可以查看当前所有运行的检查信息。 |

示例

```
SELECT DBMS_HM.GET_RUN_REPORT('hm1') FROM dual;
DBMS_HM.GET_RUN_REPO
------------------------------------------------------------
 Run Name                    : hm1
 Run Id                      : 3
 Check Name                  : Redo File Check
 Mode                        : MANUAL
 status                      : COMPLETED
 Start Time                  : 2022-07-19 01:50:27
 End Time                    : 2022-07-19 01:50:27
 Error Encountered           : 0
```

```
  Source Incident Id          : 0
  Number of Incidents Created : 0

Input Parameters for the Run
  RF_NUM=1
```

# DBMS_IJOB

DBMS_IJOB包提供了一组内置的存储过程，用于对指定用户创建和管理定时任务。需要注意的是，调用这些存储过程所执行的对定时任务的所有操作，均只有在执行COMMIT后才会生效。

## SUBMIT

```
DBMS_IJOB.SUBMIT(
    JOB OUT BINARY_INTEGER
    LUSER IN VARCHAR2,
    PUSER IN VARCHAR2,
    CUSER IN VARCHAR2 DEFAULT CURUSER,
    NEXT_DATE IN DATE DEFAULT SYSDATE,
    INTERVAL IN VARCHAR DEFAULT NULL,
    BROKEN IN BOOLEAN DEFAULT FALSE,
    WHAT IN VARCHAR2 NOT NULL
    CS_LAB IN VARCHAR2,
    CS_HI IN VARCHAR2,
    CS_LO IN VARCHAR2,
    NLSENV IN VARCHAR2,
    ENV IN VARCHAR2,
);
```

SUBMIT程序可以给指定的用户创建一个新的定时任务，成功创建的定时任务将可以在DBA_JOBS/ALL_JOBS/USER_JOBS视图中查询。

> **Note**：
> 本程序只能被如下用户调用执行：
> - SYS用户
> - DBA角色的用户
> - 拥有all privileges权限的用户

| 参数 | 描述 |
| --- | --- |
| JOB | 创建的任务编号 |
| LUSER | 选填，不生效 |
| PUSER | 选填，不生效 |
| CUSER | 任务执行的角色，默认获取当前登录用户 |
| NEXT_DATE | 定时任务下次执行的时间，默认立即执行 |
| INTERVAL | 表达式文本，用于计算定时任务下次执行的时间，表达式为NULL表示定时任务只执行一次。通过表达式计算出的时间必须为将来时间或者NULL |
| BROKEN | 任务是否中断，默认FLASE |
| WHAT | 定时任务要执行的PL文本，可以是匿名块或者存储过程，必须以分号结束 |
| CS_LAB | 选填，不生效 |
| CS_HI | 选填，不生效 |
| CS_LO | 选填，不生效 |
| NLSENV | 选填，不生效 |
| ENV | 选填，不生效 |

示例

```
--在SALES用户下创建存储过程
CREATE TABLE tbl_job(id VARCHAR2(30), name VARCHAR2(30));
CREATE OR REPLACE PROCEDURE proce_t IS
BEGIN
INSERT INTO tbl_job(id, name)
VALUES('1', TO_CHAR(SYSDATE, 'yyyy-mm-dd hh24:mi:ss'));
COMMIT;
END proce_t;
/

--以SYS用户执行，为SALES用户创建JOB
DECLARE
    job_id INT;
BEGIN
    SYS.DBMS_IJOB.SUBMIT(
    job_id,
 cuser=>'SALES',
 INTERVAL=> 'sysdate+1/24/60',
 what=> 'proce_t;');
COMMIT;
END;
/


--在SALES用户下查看job运行结果：每分钟插入一条数据
SELECT * FROM tbl_job;
ID                               NAME
-------------------------------- --------------------------------
1                                2022-11-08 19:50:57
1                                2022-11-08 19:51:57
1                                2022-11-08 19:52:57
1                                2022-11-08 19:53:57

--在SALES用户下删除JOB（从user_jobs视图获取该JOB的ID）
exec DBMS_JOB.REMOVE(1925);
COMMIT;
```

# DBMS_JOB

DBMS_JOB包提供了一组内置的存储过程，用于创建和管理定时任务。需要注意的是，调用这些存储过程所执行的对定时任务的所有操作，均只有在执行COMMIT后才会生效。

## SUBMIT

```
DBMS_JOB.SUBMIT(
    job OUT BIGINT,
    what IN VARCHAR,
    next_date IN DATE DEFAULT SYSDATE,
    interval IN VARCHAR DEFAULT NULL,
    no_parse IN BOOLEAN DEFAULT FALSE,
    instance IN BINARY_INTEGER DEFAULT 0,
    force IN BOOLEAN DEFAULT FALSE);
```

SUBMIT程序用于创建一个新的定时任务，成功创建的定时任务将可以在DBA_JOBS/ALL_JOBS/USER_JOBS视图中查询。

| 参数 | 描述 |
| --- | --- |
| job | 系统为定时任务分配的对象ID，BIGINT类型。 |
| what | 定时任务要执行的PL文本，可以是匿名块或者存储过程，必须以分号结束。 |
| next_date | 定时任务下次执行的时间。 |
| interval | 表达式文本，用于计算定时任务下次执行的时间，表达式为NULL表示定时任务只执行一次。通过表达式计算出的时间必须为将来时间或者NULL。 |
| no_parse | 标识在创建定时任务时，是否需要解析what指定的PL文本，true表示不解析，false表示会解析校验。 |
| instance | 定时任务执行的实例，默认值0表示在任意实例上执行。 |
| force | 为FALSE时，设置的instance必须是正在运行的。为TRUE时，任意整数的instance都可以创建。 |

说明：

- 共享集群部署中通过查询V$INSTANCE视图的INSTANCE_NUMBER字段可获得所在实例的instance值，其他部署模式下的instance值恒为1。
- 单机部署下输入非1的instance值在force=true时可以成功创建定义定时任务，但该任务并不会被执行。

示例

```
CREATE TABLE job_table(time_ss CHAR(2),job_type VARCHAR(10),job_name VARCHAR(20));
--创建存储过程
CREATE OR REPLACE PROCEDURE job_proc IS
BEGIN
  INSERT INTO job_table VALUES (
    TO_CHAR(SYSDATE, 'SS'),
    'job',
    'job example'
    );
  COMMIT;
END;
/

DECLARE
  jobid BIGINT;
BEGIN
  DBMS_JOB.SUBMIT(jobid,
    'begin job_proc; end;',
    SYSDATE,
    'SYSDATE+1'
    );
```

```
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Job:'||jobid||' is created!');
END;
/

--result
Job:1681 is created!
```

## BROKEN

```
DBMS_JOB.BROKEN(
    job IN BIGINT,
    broken IN BOOLEAN,
    next_date IN DATE DEFAULT SYSDATE);
```

BROKEN程序用于设置定时任务的停止标志，被停止的定时任务将不再被系统进行任务调度，需要注意的是，BROKEN程序不能作用于一个正在执行的定时任务。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| broken | 是否停止定时任务，true表示停止，false表示不停止。 |
| next_date | 定时任务下次执行的时间。 |

示例

```
--停止JOB
EXEC DBMS_JOB.BROKEN(1681,true);
COMMIT;

--重启被停止的JOB
EXEC DBMS_JOB.BROKEN(1681,false);
COMMIT;
```

## CHANGE

```
DBMS_JOB.CHANGE(
    job IN BIGINT,
    what IN VARCHAR,
    next_date IN DATE,
    interval IN VARCHAR,
    instance IN INTEGER DEFAULT NULL,
    force IN BOOLEAN DEFAULT FALSE);
```

CHANGE程序用于修改定时任务的相关属性，其中，使用NULL参数值表示对属性仍保持原值。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| what | 定时任务要执行的PL文本，可以是匿名块或者存储过程，必须以分号结束。 |
| next_date | 定时任务下次执行的时间。 |
| interval | 表达式文本，用于计算定时任务下次执行的时间，表达式为NULL表示定时任务只执行一次。通过表达式计算出的时间必须为将来时间或者NULL。 |
| instance | 定时任务在集群环境下执行的实例，默认值NULL表示不会更改job的instance。 |
| force | 为FALSE时，设置的instance必须是正在运行的。为TRUE时，任意整数的instance都可以创建。 |

示例

```
--修改JOB的执行间隔为1小时
EXEC DBMS_JOB.CHANGE(1681,'begin job_proc; end;',NULL,'SYSDATE+1/24');
COMMIT;

EXEC DBMS_JOB.CHANGE(1681,'begin job_proc; end;',NULL,'SYSDATE+1/24', 3);
YAS-06806 job associated instance number 3 is not valid

EXEC DBMS_JOB.CHANGE(1681,'begin job_proc; end;',NULL,'SYSDATE+1/24', 1, TRUE);
COMMIT;
```

## INSTANCE

```
DBMS_JOB.INSTANCE(
    job IN BIGINT,
    instance IN INTEGER,
    force IN BOOLEAN DEFAULT FALSE);
```

INSTANCE程序用于设置定时任务关联的实列。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| instance | 定时任务在集群环境下执行的实例。 |
| force | 为FALSE时，设置的instance必须是正在运行的。为TRUE时，任意整数的instance都可以创建。 |

示例

```
EXEC DBMS_JOB.INSTANCE(1681, 1);
COMMIT;

EXEC DBMS_JOB.INSTANCE(1681, 1, TRUE);
COMMIT;
```

## INTERVAL

```
DBMS_JOB.INTERVAL(
    job IN BIGINT,
    interval IN VARCHAR);
```

INTERVAL程序用于修改定时任务执行的频率。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| interval | 表达式文本，用于计算定时任务下次执行的时间，表达式为NULL表示定时任务只执行一次。通过表达式计算出的时间必须为将来时间或者NULL。 |

示例

```
--修改JOB的执行间隔为2小时
EXEC DBMS_JOB.INTERVAL(1681,'SYSDATE+1/12');
COMMIT;
```

## NEXT_DATE

```
DBMS_JOB.NEXT_DATE(
    job IN BIGINT,
    next_date IN DATE);
```

NEXT_DATE程序用于修改定时任务下次执行的时间。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| next_date | 定时任务下次执行的时间。 |

示例

```
--修改JOB的下次执行时间为1分钟后
EXEC DBMS_JOB.NEXT_DATE(1681,SYSDATE+1/24/60);
COMMIT;
```

## RUN

```
DBMS_JOB.RUN(
    job IN BIGINT,
    force IN BOOLEAN DEFAULT FALSE);
```

RUN程序用于手动执行一次定时任务。定时任务的状态会变为RUNNING。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| force | 为FALSE时只能执行当前实例的定时任务。为TRUE时，可以执行非当前实例的定时任务。 |

示例

```
EXEC DBMS_JOB.RUN(1681);
COMMIT;
```

## WHAT

```
DBMS_JOB.WHAT(
    job IN BIGINT,
    what IN VARCHAR);
```

WHAT程序用于修改定时任务的执行内容。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |
| what | 定时任务要执行的PL文本，可以是匿名块或者存储过程，必须以分号结束。 |

示例

```
EXEC DBMS_JOB.WHAT(1681,'begin job_proc; commit; end;');
COMMIT;
```

## REMOVE

```
DBMS_JOB.REMOVE(
    job IN BIGINT);
```

REMOVE程序用于删除一个非执行状态下的定时任务，执行删除后将无法在DBA_JOBS/ALL_JOBS/USER_JOBS视图中查询到该定时任务数据。

| 参数 | 描述 |
| --- | --- |
| job | 定时任务的对象ID，可以通过DBA_JOBS/ALL_JOBS/USER_JOBS视图查询。 |

示例

```
EXEC DBMS_JOB.REMOVE(1681);
COMMIT;
```

# DBMS_LOB

DBMS_LOB包提供了一组内置的函数，用于LOB类型（或可与LOB进行隐式转换的类型）相关的运算。DBMS_LOB包不适用于分布式部署。

## COMPARE

```
DBMS_LOB.COMPARE (
   LOB_1            IN BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW,
   LOB_2            IN BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW,
   AMOUNT           IN BIGINT DEFAULT MAX(LENGTH(LOB_1), LENGTH(LOB_2)),
   OFFSET_1         IN BIGINT DEFAULT 1,
   OFFSET_2         IN BIGINT DEFAULT 1)
  RETURN INTEGER;
```

COMPARE函数用于比较两个LOB数据的大小，返回一个值为0、-1或者1的INTEGER类型数据。

| 参数 | 描述 |
| --- | --- |
| LOB_1 | 用于比较的第一个LOB数据，类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW。 |
| LOB_2 | 用于比较的第二个LOB数据，类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW。 |
| AMOUNT | 用于比较的数据长度，函数从偏移位置开始获取该长度的字节数（对于BLOB/RAW）或字符数（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR）进行比较。缺省取LOB_1与LOB_2数据长度的较大值。 |
| OFFSET_1 | LOB_1用于比较的偏移量。缺省为1，即从第1个字节（对于BLOB/RAW）或字符（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR）开始获取AMOUNT长度的数据进行比较。 |
| OFFSET_2 | LOB_2用于比较的偏移量。缺省为1，即从第1个字节（对于BLOB/RAW）或字符（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR）开始获取AMOUNT长度的数据进行比较。 |

本函数的主要规则为：

- 如果LOB_1和LOB_2在OFFSET和AMOUNT参数指定的范围内完全匹配，函数返回0；如果LOB_1小于LOB_2，函数返回-1；如果LOB_1大于LOB_2，函数返回1。
- 当LOB_1和LOB_2中的一个为BLOB/RAW类型，而另一个为CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR类型时，函数返回错误。
- 当LOB_1和LOB_2中的一个为CLOB/CHAR/VARCHAR类型，而另一个为NCLOB/NCHAR/NVARCHAR类型时，函数会将CLOB/CHAR/VARCHAR类型的字符串转码为国家字符集形态，再和NCLOB/NCHAR/NVARCHAR类型进行比较。
- 出现如下情况之一时，函数返回NULL：
  - AMOUNT < 1
  - OFFSET_1 < 1
  - OFFSET_2 < 1
  - 任一个参数输入了NULL
- AMOUNT、OFFSET_1、OFFSET_2可以为数值型或字符型，函数将隐式转换为BIGINT，该转换对NUMBER类型的小数做截断处理，其他类型的小数按四舍五入处理。
- NCLOB/NCHAR/NVARCHAR按国家字符集UTF-16进行编码，固定按2B一个字符进行计算。

示例（HEAP表）

```
CREATE TABLE clob_compare(C1 CLOB, C2 CLOB);
INSERT INTO clob_compare VALUES('124SD', '12332');
--对C1、C2进行完全比较
SELECT DBMS_LOB.COMPARE(C1, C2) compare_res FROM clob_compare;
 COMPARE_RES
------------
           1
--比较C1、C2的前两个字符
SELECT DBMS_LOB.COMPARE(C1, C2, 2) compare_res FROM clob_compare;
 COMPARE_RES
------------
```

```
           0
--比较C1的第2、3位字符和C2的第5位字符
SELECT DBMS_LOB.COMPARE(C1, C2, 2, 2, 5) compare_res FROM clob_compare;
 COMPARE_RES
------------
           1

CREATE TABLE raw_compare(C1 RAW(10), C2 RAW(10));
INSERT INTO raw_compare VALUES('AB', 'A234');
--对C1、C2进行完全比较
SELECT DBMS_LOB.COMPARE(C1, C2) compare_res FROM raw_compare;
 COMPARE_RES
------------
           1
--比较C1、C2的第一个字节
SELECT DBMS_LOB.COMPARE(C1, C2, 1) compare_res FROM raw_compare;
 COMPARE_RES
------------
           1
--C1未取到数据，而C2获取到第2个字节的比较
SELECT DBMS_LOB.COMPARE(C1, C2, 2, 2, 2) compare_res FROM raw_compare;
 COMPARE_RES
------------
          -1
```

## GET_LENGTH

```
DBMS_LOB.GET_LENGTH (
   LOB_LOC    IN  BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW)
RETURN BIGINT;
```

GET_LENGTH函数用于获取指定LOB数据的长度，返回一个BIGINT类型的数值。

| 参数 | 描述 |
| --- | --- |
| LOB_LOC | 输入参数，类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW。 |

本函数的主要规则为：

- 输入参数为NULL时，函数返回NULL。
- 输入参数为BLOB/RAW类型时，函数返回的是按字节的长度；输入参数为CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR时，函数返回的是按字符的长度。
- NCLOB/NCHAR/NVARCHAR按国家字符集UTF-16进行编码，固定按2B一个字符进行计算。

示例（HEAP表）

```
CREATE TABLE get_length(C1 CLOB, C2 BLOB);
INSERT INTO get_length VALUES('1U8WQHSD', 'AF1234');
SELECT DBMS_LOB.GET_LENGTH(C1) cl_len,
       DBMS_LOB.GET_LENGTH(C2) c2_len
FROM get_length;
               CL_LEN               C2_LEN
-------------------- --------------------
                   8                    3
```

## GETLENGTH

同GET_LENGTH。

## SUB_STR

```
DBMS_LOB.SUB_STR (
   LOB_LOC     IN    BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW,
   AMOUNT      IN    INTEGER DEFAULT 32000,
   OFFSET      IN    BIGINT DEFAULT 1)
RETURN RAW/VARCHAR2;
```

SUB_STR函数用于读取给定LOB数据的子串，返回RAW或者VARCHAR类型的数据。

| 参数 | 描述 |
| --- | --- |
| LOB_LOC | 要读取的LOB数据，类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW。 |
| AMOUNT | 要读取的字节数（对于BLOB/RAW）或字符数（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR），该值范围为[1, 32000]，缺省为32000。 特例情况，因为NCLOB/NCHAR/NVARCHAR按国家字符集UTF-16进行编码，固定按2B一个字符进行计算，该值范围在[16000, 32000]，实际按16000最大字符数处理。 |
| OFFSET | 偏移量。缺省为1，即从第1个字节（对于BLOB/RAW）或字符（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR）开始读取AMOUNT长度的数据。 |

本函数的主要规则为：

- 函数对LOB_LOC读取从OFFSET开始的AMOUNT长度的字串，单位为字节（对于BLOB/RAW）或字符（对于CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR）。
- 当LOB_LOC为BLOB/RAW类型时，函数返回RAW类型；当LOB_LOC为CLOB/CHAR/VARCHAR类型时，函数返回VARCHAR类型。当LOB_LOC为NCLOB/NCHAR/NVARCHAR类型时，函数返回NVARCHAR类型。
- 出现如下情况之一时，函数返回NULL：
  - AMOUNT < 1
  - AMOUNT > 32000
  - OFFSET < 1
  - 任一个参数输入了NULL

示例（HEAP表）

```
CREATE TABLE sub_str(C1 CLOB, C2 BLOB);
INSERT INTO sub_str VALUES('1U8WQHSD', 'AF1234');
--读取全部的C1和C2
SELECT DBMS_LOB.SUB_STR(C1) sub_c1,
       DBMS_LOB.SUB_STR(C2) sub_c2
FROM sub_str;
SUB_C1              SUB_C2
------------------- -------------------
1U8WQHSD            AF1234

--从C1/C2的第3个字符/字节开始读取2个字符/字节
SELECT DBMS_LOB.SUB_STR(C1, 2, 3) sub_c1,
       DBMS_LOB.SUB_STR(C2, 2, 3) sub_c2
FROM sub_str;
SUB_C1              SUB_C2
------------------- -------------------
8W                  34
```

## SUBSTR

同SUB_STR。

## CREATETEMPORARY

```
DBMS_LOB.CREATETEMPORARY (
   lob_loc  IN OUT  BLOB/CLOB/NCLOB,
   cache    IN      BOOLEAN DEFAULT FALSE,
   dur      IN      INTEGER DEFAULT 10);
```

CREATETEMPORARY过程用于创建一个长度为0的临时LOB。

| 参数 | 描述 |
| --- | --- |
| lob_loc | LOB变量，参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。 |
| cache | 保留字段，使用缺省值。BOOLEAN类型或可隐式转换为BOOLEAN类型的参数。不可为NULL。 |
| dur | 保留字段，使用缺省值。INTEGER类型或可隐式转换为INTEGER类型的参数的数值范围与INTEGER类型的数值范围相同。不可为NULL。 |

示例（单机、共享集群部署）

```
DECLARE
  clob1 CLOB;
  len INT;
BEGIN
  DBMS_LOB.CREATETEMPORARY(clob1);
  len := DBMS_LOB.GETLENGTH(clob1);
  DBMS_OUTPUT.PUT_LINE(len);
END;
/
--result
0
```

## FREETEMPORARY

```
DBMS_LOB.FREETEMPORARY (
   lob_loc  IN OUT  BLOB/CLOB/NCLOB);
```

FREETEMPORARY过程用于释放一个临时LOB。临时LOB释放后长度为0，在DBMS_LOB高级包中视为无效的临时LOB。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 临时LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/NCLOB类型。不可为NULL。 |

示例（单机、共享集群部署）

```
DECLARE
  clob1 CLOB;
BEGIN
  DBMS_LOB.CREATETEMPORARY(clob1);
  DBMS_LOB.FREETEMPORARY(clob1);
  DBMS_OUTPUT.PUT_LINE(LENGTH(clob1));
END;
/
--result
0
```

## ISTEMPORARY

```
DBMS_LOB.ISTEMPORARY (
   lob_loc  IN  BLOB/CLOB/NCLOB)
RETURN INTEGER;
```

ISTEMPORARY过程用于判断输入的LOB是否为临时LOB。 如果给定的参数是有效的临时LOB，则返回1。 如果给定的参数是无效的临时LOB或者不是临时LOB，则返回0。 如果给定的参数为NULL，则返回NULL。

| 参数 | 描述 |
| --- | --- |
| lob_loc | LOB变量，参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。 |

示例（单机、共享集群部署）

```
CREATE TABLE isnot_templob(id INT, c1 CLOB);
INSERT INTO isnot_templob VALUES(1, 'lob');

DECLARE
  clob1 CLOB;
  istemp INT;
BEGIN
  --valid TEMPORARY LOB
  DBMS_LOB.CREATETEMPORARY(clob1);
  istemp := DBMS_LOB.ISTEMPORARY(clob1);
  DBMS_OUTPUT.PUT_LINE(istemp);
  --invalid TEMPORARY LOB
  DBMS_LOB.FREETEMPORARY(clob1);
  istemp := DBMS_LOB.ISTEMPORARY(clob1);
  DBMS_OUTPUT.PUT_LINE(istemp);
  --IS NOT TEMPORARY LOB
  SELECT c1 INTO clob1 FROM isnot_templob WHERE id = 1;
  istemp := DBMS_LOB.ISTEMPORARY(clob1);
  DBMS_OUTPUT.PUT_LINE(istemp);
END;
/
--result
1
0
0
```

## OPEN

```
DBMS_LOB.OPEN (
   lob_loc    IN OUT  BLOB/CLOB/NCLOB,
   open_mode  IN      INTEGER);
```

OPEN过程用于按指定的模式打开一个LOB。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待打开的LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。不可为NULL。 |
| open_mode | LOB打开的模式。输入DBMS_LOB.LOB_READONLY，以只读模式打开LOB；输入DBMS_LOB.LOB_READWRITE，以读写模式打开LOB。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'read lob';
  amount INT := 3;
  buffer VARCHAR(10);
BEGIN
  DBMS_LOB.OPEN(clob1, DBMS_LOB.LOB_READONLY);
  DBMS_LOB.READ(clob1, amount, 6, buffer);
  DBMS_OUTPUT.PUT_LINE(buffer);
  DBMS_LOB.CLOSE(clob1);
END;
/
```

```
--result
lob
```

## CLOSE

```
DBMS_LOB.CLOSE (
   lob_loc  IN OUT  BLOB/CLOB/NCLOB);
```

CLOSE过程用于关闭一个打开的LOB。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待关闭的LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/NCLOB类型。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob';
BEGIN
  DBMS_LOB.OPEN(clob1, DBMS_LOB.LOB_READWRITE);
  DBMS_LOB.WRITE(clob1, 5, 5, 'write');
  DBMS_OUTPUT.PUT_LINE(clob1);
  DBMS_LOB.CLOSE(clob1);
END;
/
--result
lob write
```

## ISOPEN

```
DBMS_LOB.ISOPEN (
   lob_loc  IN  BLOB/CLOB/NCLOB)
RETURN INTEGER;
```

ISOPEN函数用于判断一个LOB是否处于打开状态。如果LOB处于打开状态，则返回1，否则返回0。

| 参数 | 描述 |
| --- | --- |
| lob_loc | LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob';
  isopen INT;
BEGIN
  isopen := DBMS_LOB.ISOPEN(clob1);
  DBMS_OUTPUT.PUT_LINE(isopen);
  DBMS_LOB.OPEN(clob1, DBMS_LOB.LOB_READWRITE);
  isopen := DBMS_LOB.ISOPEN(clob1);
  DBMS_OUTPUT.PUT_LINE(isopen);
  DBMS_LOB.CLOSE(clob1);
  isopen := DBMS_LOB.ISOPEN(clob1);
  DBMS_OUTPUT.PUT_LINE(isopen);
END;
/
--result
0
```

```
1
0
```

## READ

```
DBMS_LOB.READ (
   lob_loc  IN      BLOB,
   amount   IN OUT  BIGINT,
   offset   IN      BIGINT,
   buffer   OUT     RAW);

DBMS_LOB.READ (
   lob_loc  IN      CLOB,
   amount   IN OUT  BIGINT,
   offset   IN      BIGINT,
   buffer   OUT     VARCHAR);

DBMS_LOB.READ (
   lob_loc  IN      NCLOB,
   amount   IN OUT  BIGINT,
   offset   IN      BIGINT,
   buffer   OUT     NVARCHAR);
```

READ过程用于读取LOB指定部分的数据。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待读取的LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。不可为NULL。 |
| amount | 输入为要读取的BLOB字节数或CLOB/NCLOB字符数，至少为1，不能超过32000；输出为实际读取到的BLOB字节数或CLOB/NCLOB字符数。 |
| offset | 读取起点在LOB中的偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始，不能超过LOB末尾。不可为NULL。 |
| buffer | 输出读取到的指定部分LOB数据。当读取BLOB时，参数的数据类型可以为RAW/BLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型；当读取CLOB时，参数的数据类型可以为CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR/RAW类型。 |

示例（单机、共享集群部署）

```
DECLARE
  clob1 CLOB := 'read lob';
  amount INT := 10;
  buffer VARCHAR(10);
BEGIN
  DBMS_LOB.READ(clob1, amount, 6, buffer);
  DBMS_OUTPUT.PUT_LINE(buffer);
  DBMS_OUTPUT.PUT_LINE(amount);
END;
/
--result
lob
3
```

## WRITE

```
DBMS_LOB.WRITE (
   lob_loc  IN OUT  BLOB,
   amount   IN      BIGINT,
   offset   IN      BIGINT,
   buffer   IN      RAW);

DBMS_LOB.WRITE (
```

```
    lob_loc  IN OUT  CLOB,
    amount   IN      BIGINT,
    offset   IN      BIGINT,
    buffer   IN      VARCHAR);

DBMS_LOB.WRITE (
    lob_loc  IN OUT  NCLOB,
    amount   IN      BIGINT,
    offset   IN      BIGINT,
    buffer   IN      NVARCHAR);
```

WRITE过程用于在LOB的指定位置写入指定数量的字节或字符。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待写入的LOB变量，不可为无效的临时LOB。在写入LOB列时需要拥有UPDATE权限。不可为NULL。 |
| amount | 写入BLOB的字节数或写入CLOB/NCLOB的字符数，至少为1，不能超过32000。不可为NULL。 |
| offset | 写入起点在LOB中的偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始。不可为NULL。 |
| buffer | 待写入LOB的数据，若为LOB类型，不可为无效的临时LOB。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob';
  buffer VARCHAR(10) := 'write lob';
BEGIN
  DBMS_LOB.WRITE(clob1, 5, 5, buffer);
  DBMS_OUTPUT.PUT_LINE(clob1);
END;
/
--result
lob write
```

## APPEND

```
DBMS_LOB.APPEND (
   dest_lob  IN OUT  BLOB,
   src_lob   IN      BLOB);

DBMS_LOB.APPEND (
   dest_lob  IN OUT  CLOB,
   src_lob   IN      CLOB);

DBMS_LOB.APPEND (
   dest_lob  IN OUT  NCLOB,
   src_lob   IN      NCLOB);
```

APPEND过程用于将源LOB的数据附加到目标LOB的末尾。

| 参数 | 描述 |
| --- | --- |
| dest_lob | 目标LOB变量，不可为无效的临时LOB。在更新LOB列时需要拥有UPDATE权限。参数的数据类型可以为BLOB/RAW/CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| src_lob | 源LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/RAW/CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |

示例（HEAP表）

```
DECLARE
  dest_clob CLOB := 'lob';
  src_clob CLOB := ' append';
BEGIN
  DBMS_LOB.APPEND(dest_clob, src_clob);
  DBMS_OUTPUT.PUT_LINE(dest_clob);
END;
/
--result
lob append
```

## WRITEAPPEND

```
DBMS_LOB.WRITEAPPEND (
   lob_loc  IN OUT  BLOB,
   amount   IN      BIGINT,
   buffer   IN      RAW);

DBMS_LOB.WRITEAPPEND (
   lob_loc  IN OUT  CLOB,
   amount   IN      BIGINT,
   buffer   IN      VARCHAR);

DBMS_LOB.WRITEAPPEND (
   lob_loc  IN OUT  NCLOB,
   amount   IN      BIGINT,
   buffer   IN      NVARCHAR);
```

WRITEAPPEND过程用于将指定数量的字节或字符附加到LOB的末尾。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待附加数据的LOB变量，不可为无效的临时LOB。在更新LOB列时需要拥有UPDATE权限。参数的数据类型可以为BLOB/CLOB/VARCHAR/RAW/NCLOB/NVARCHAR类型。不可为NULL。 |
| amount | 附加的BLOB字节数或CLOB/NCLOB字符数，至少为1，不能超过32000。不可为NULL。 |
| buffer | 待附加到LOB末尾的数据，不可为无效的临时LOB。参数的数据类型可以为RAW/BLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob';
  buffer VARCHAR(20) := ' append lob';
BEGIN
  DBMS_LOB.WRITEAPPEND(clob1, 7, buffer);
  DBMS_OUTPUT.PUT_LINE(clob1);
END;
/
--result
lob append
```

## COPY

```
DBMS_LOB.COPY (
  dest_lob     IN OUT  BLOB,
  src_lob      IN      BLOB,
  amount       IN      BIGINT,
  dest_offset  IN      BIGINT DEFAULT 1,
  src_offset   IN      BIGINT DEFAULT 1);

DBMS_LOB.COPY (
```

```
  dest_lob     IN OUT  CLOB,
  src_lob      IN      CLOB,
  amount       IN      BIGINT,
  dest_offset  IN      BIGINT DEFAULT 1,
  src_offset   IN      BIGINT DEFAULT 1);

DBMS_LOB.COPY (
  dest_lob     IN OUT  NCLOB,
  src_lob      IN      NCLOB,
  amount       IN      BIGINT,
  dest_offset  IN      BIGINT DEFAULT 1,
  src_offset   IN      BIGINT DEFAULT 1);
```

COPY过程用于将指定部分的源LOB数据拷贝到目标LOB的指定位置。

| 参数 | 描述 |
| --- | --- |
| dest_lob | 目标LOB变量，不可为无效的临时LOB。在更新LOB列时需要拥有UPDATE权限。参数的数据类型可以为BLOB/RAW/CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| src_lob | 源LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/RAW/CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| amount | 拷贝的BLOB字节数或CLOB/NCLOB字符数，至少为1。不可为NULL。 |
| dest_offset | 目标LOB偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始。不可为NULL。 |
| src_offset | 源LOB偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob';
BEGIN
  DBMS_LOB.COPY(clob1, 'copy lob', 4, 5, 1);
  DBMS_OUTPUT.PUT_LINE(clob1);
END;
/
--result
lob copy
```

## ERASE

```
DBMS_LOB.ERASE (
   lob_loc   IN OUT  BLOB/CLOB/NCLOB,
   amount    IN OUT  BIGINT,
   offset    IN      BIGINT DEFAULT 1);
```

ERASE过程用于擦除指定部分的LOB数据。

| 参数 | 描述 |
| --- | --- |
| lob_loc | LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。在擦除LOB列时需要拥有UPDATE权限。不可为NULL。 |
| amount | 擦除的BLOB字节数或CLOB/NCLOB字符数，至少为1。不可为NULL。 |
| offset | 待擦除的数据在LOB中的偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'loberaseerase';
  amount INT := 5;
BEGIN
  DBMS_LOB.ERASE(clob1, amount, 4);
  DBMS_OUTPUT.PUT_LINE(clob1);
END;
/
--result
lob     erase
```

## TRIM

```
DBMS_LOB.TRIM (
   lob_loc  IN OUT  BLOB/CLOB/NCLOB,
   newlen   IN      BIGINT);
```

TRIM过程用于将LOB修剪至指定长度。

| 参数 | 描述 |
| --- | --- |
| lob_loc | LOB变量，不可为无效的临时LOB。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/NCLOB/NCHAR/NVARCHAR/RAW类型。在修剪LOB列时需要拥有UPDATE权限。不可为NULL。 |
| newlen | 修剪后LOB的新长度，BLOB以字节为单位，CLOB/NCLOB以字符为单位，至少为0，不能超过LOB的长度。不可为NULL。 |

示例（HEAP表）

```
DECLARE
  clob1 CLOB := 'lob trim newlen';
BEGIN
  DBMS_LOB.TRIM(clob1, 8);
  DBMS_OUTPUT.PUT_LINE(clob1);
END;
/
--result
lob trim
```

## INSTR

```
DBMS_LOB.INSTR (
   lob_loc  IN  BLOB,
   pattern  IN  RAW,
   offset   IN  BIGINT DEFAULT 1,
   nth      IN  BIGINT DEFAULT 1)
RETURN BIGINT;

DBMS_LOB.INSTR (
   lob_loc  IN  CLOB,
   pattern  IN  VARCHAR,
   offset   IN  BIGINT DEFAULT 1,
   nth      IN  BIGINT DEFAULT 1)
RETURN BIGINT;

DBMS_LOB.INSTR (
   lob_loc  IN  NCLOB,
   pattern  IN  NVARCHAR,
   offset   IN  BIGINT DEFAULT 1,
   nth      IN  BIGINT DEFAULT 1)
RETURN BIGINT;
```

INSTR函数用于返回pattern在LOB中第n次出现的位置，从LOB的指定位置开始匹配。如果第n次匹配失败，则返回0。

| 参数 | 描述 |
| --- | --- |
| lob_loc | 待匹配的LOB变量。参数的数据类型可以为BLOB/CLOB/CHAR/VARCHAR/RAW/NCLOB/NCHAR/NVARCHAR类型。 |
| pattern | 待匹配的模式。当匹配BLOB时，参数的数据类型可以为RAW/BLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。当匹配CLOB/NCLOB时，参数的数据类型可以为RAW/CLOB/NCLOB/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| offset | 匹配起点在LOB中的偏移量，BLOB以字节为单位，CLOB/NCLOB以字符为单位，从1开始。 |
| nth | 匹配次数。 |

示例（单机、共享集群部署）

```
DECLARE
  clob1 CLOB := 'lob instr lob instr';
  pos BIGINT;
BEGIN
  pos := DBMS_LOB.INSTR(clob1, 'instr', 4, 2);
  DBMS_OUTPUT.PUT_LINE(pos);
END;
/
--result
15
```

## 异常

| 异常 | 描述 |
| --- | --- |
| INVALID_ARGVAL | 参数超出数值范围（除数字溢出外）。 |
| UNOPENED_FILE | 关闭未打开的LOB。 |

示例（单机、共享集群部署）

```
--invalid_argval exception
DECLARE
  c1 CLOB:='12345';
BEGIN
  DBMS_LOB.WRITE(c1,0,1,'6');
EXCEPTION
  WHEN DBMS_LOB.INVALID_ARGVAL THEN
    DBMS_OUTPUT.PUT_LINE('argument value out of range');
END;
/
--result
argument value out of range

--unopened_file exception
DECLARE
  c1 CLOB:='12345';
BEGIN
  DBMS_LOB.CLOSE(c1);
EXCEPTION
  WHEN DBMS_LOB.UNOPENED_FILE THEN
    DBMS_OUTPUT.PUT_LINE('try to close unopened lob');
END;
/
--result
try to close unopened lob
```

# DBMS_LOCK

DBMS_LOCK包提供了一组内置的存储过程/函数，用于实现锁相关功能。

## SLEEP

```
DBMS_LOCK.SLEEP(
    second IN NUMBER);
```

SLEEP为存储过程，根据输入的参数值暂停对应的时间，单位为秒。

执行本程序时，系统将按参数指定的秒数进行休眠，如需在秒数结束前停止休眠，须在终端执行CTRL+C、Shutdown等指令进行强制退出。

| 参数 | 描述 |
| --- | --- |
| second | 指定休眠的时间，不可输入空，数值范围必须为[0,21474836.47]，支持入参类型隐式转换。 |

示例

```
-- 数值型
exec DBMS_LOCK.SLEEP (10);

-- 字符串型
exec DBMS_LOCK.SLEEP ('10');

-- BIT型
DECLARE
vsql VARCHAR(256);
vbit BIT(8);
BEGIN
vsql := 'exec DBMS_LOCK.SLEEP (?)';
vbit := 5;
EXECUTE IMMEDIATE vsql USING vbit;
END;
/

-- 布尔型
DECLARE
vsql VARCHAR(256);
vbool BOOLEAN;
BEGIN
vsql := 'exec DBMS_LOCK.SLEEP (?)';
vbool := 'true';
EXECUTE IMMEDIATE vsql USING vbool;
END;
/
```

# DBMS_METADATA

DBMS_METADATA包提供了一组内置程序接口，用于从数据库字典中检索元数据。

## GET_DDL

```
DBMS_METADATA.GET_DDL (
object_type     IN VARCHAR,
name            IN VARCHAR,
schema          IN VARCHAR DEFAULT NULL,
version         IN VARCHAR,
model           IN VARCHAR,
transform       IN VARCHAR)
RETURN CLOB;
```

GET_DDL函数用于获取指定对象的元数据，该对象必须是一个可创建的对象。

本函数将返回与指定对象含义相同的创建语句（可能包含多条语句），但对于PL对象不返回'/'结束符，对于SQL对象不返回';'结束符。

| 参数 | 描述 |
| --- | --- |
| object_type | 要检索的对象的类型，不能为NULL，可输入的值有：<br>VIEW<br>FUNCTION<br>TRIGGER<br>PROCEDURE<br>PACKAGE<br>TYPE<br>TABLE：TAC表和EXTERNAL表不能获取创建语句<br>MATERIALIZED VIEW |
| name | 对象的名称，不能为NULL。使用双引号命名的对象按实际大小写，否则需使用全大写名称 |
| schema | 对象所属用户。当此参数为NULL时，默认为当前登录用户 |
| version | 提取元数据的版本。不能为NULL，选填，不生效 |
| model | 使用的模型。不能为NULL，选填，不生效 |
| transform | 转换的名称。不能为NULL，选填，不生效 |

示例（HEAP表、LSC表）

```
-- 1、VIEW
CREATE VIEW v_area_meta AS SELECT * FROM area;

--查询v_area对象的元数据
SELECT DBMS_METADATA.GET_DDL('view', 'V_AREA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------
CREATE OR REPLACE VIEW "SALES"."V_AREA"("AREA_NO","AREA_NAME","DHQ") AS SELECT "AREA_NO","AREA_NAME","DHQ" FROM area

-- 按查询结果执行语句
CREATE OR REPLACE VIEW "SALES"."V_AREA"("AREA_NO","AREA_NAME","DHQ") AS SELECT "AREA_NO","AREA_NAME","DHQ" FROM area;

-- 2、PROCEDURE
CREATE OR REPLACE EDITIONABLE PROCEDURE P1_METADATA
AS
c INT;
BEGIN
FOR i IN 1..10 LOOP
 SELECT 1 INTO c FROM dual;
 DBMS_OUTPUT.PUT_LINE ('c:'||c);
```

```
END LOOP;
END;
/

-- 查询P1_METADATA对象的元数据
SELECT DBMS_METADATA.GET_DDL('PROCEDURE','P1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
------------------------------------------------------------------
CREATE OR REPLACE PROCEDURE "SALES"."P1_METADATA"AS
c INT;
BEGIN
FOR i IN 1..10 LOOP
 SELECT 1 INTO c FROM dual;
 DBMS_OUTPUT.PUT_LINE ('c:'||c);
END LOOP;
END;

-- 3、FUNCTION
CREATE OR replace FUNCTION F1_METADATA(c INT DEFAULT 0) RETURN INT AS
BEGIN
 RETURN c;
 EXCEPTION
  WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE ('Unexpected error');
END;
/

-- 查询F1_METADATA对象的元数据
SELECT DBMS_METADATA.GET_DDL('FUNCTION','F1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
------------------------------------------------------------------
CREATE OR REPLACE FUNCTION "SALES"."F1_METADATA"(c INT DEFAULT 0) RETURN INT AS
BEGIN
 RETURN c;
 EXCEPTION
  WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE ('Unexpected error');
END;

-- 4、TRIGGER
CREATE TABLE IF NOT EXISTS test_trigger (col1 INT);
CREATE OR replace TRIGGER Tri_METADATA after INSERT ON test_trigger
BEGIN DBMS_OUTPUT.PUT_LINE('test follows table'); END;
/

-- 查询TRI_METADATA对象的元数据
SELECT DBMS_METADATA.GET_DDL('TRIGGER','TRI_METADATA') FROM dual;
DBMS_METADATA.GET_DD
------------------------------------------------------------------
CREATE OR REPLACE TRIGGER "SALES"."TRI_METADATA"after INSERT ON test_trigger
BEGIN DBMS_OUTPUT.PUT_LINE('test follows table'); END;
ALTER TRIGGER "SALES"."TRI_METADATA" ENABLE

-- 5、PACKAGE
DROP PACKAGE IF EXISTS pkg_METADATA;
CREATE OR replace PACKAGE pkg_METADATA AS
n1 VARCHAR(200) := 'abc';
TYPE rec_type IS RECORD(id INT,name VARCHAR(200) DEFAULT 'unknown');
rec_var1 rec_type;
PROCEDURE p1_METADATA;
END pkg_METADATA;
/

CREATE OR REPLACE PACKAGE BODY pkg_METADATA IS
PROCEDURE P1_METADATA IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('c:');
END;
END pkg_METADATA;
/
```

```
-- 查询PKG_METADATA对象的元数据
SELECT DBMS_METADATA.GET_DDL('PACKAGE','PKG_METADATA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------------
CREATE OR REPLACE PACKAGE "SALES"."PKG_METADATA"AS
n1 VARCHAR(200) := 'abc';
type rec_type is record(id int,name varchar(200) default 'unknown');
rec_var1 rec_type;
procedure p1_METADATA;
END pkg_METADATA;
CREATE OR REPLACE PACKAGE BODY "SALES"."PKG_METADATA"IS
PROCEDURE P1_METADATA IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('c:');
END;
END pkg_METADATA;

-- 6、TABLE

-- NORMAL TABLE
DROP TABLE IF EXISTS t1_metadata;
CREATE TABLE t1_metadata (col1 INT, col2 FLOAT, col3 VARCHAR(20), col4 NUMBER(20,6), col5 INTERVAL day(9) TO second(6), col6
INTERVAL year(9) TO month);

-- 查询T1_METADATA表的元数据
SELECT DBMS_METADATA.GET_DDL('TABLE', 'T1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------------
CREATE TABLE "SALES"."T1_METADATA"
("COL1" INTEGER,
"COL2" FLOAT,
"COL3" VARCHAR(20),
"COL4" NUMBER(20, 6),
"COL5" INTERVAL DAY(9) TO SECOND(6),
"COL6" INTERVAL YEAR(9) TO MONTH
) PCTFREE 8 INITRANS 2 MAXTRANS 255
LOGGING
TABLESPACE "USERS"
SEGMENT CREATION DEFERRED
ORGANIZATION HEAP

-- TEMPORARY TABLE
DROP TABLE t1_metadata;
CREATE GLOBAL TEMPORARY TABLE t1_metadata (col1 CLOB) ON COMMIT DELETE ROWS;

-- 查询T1_METADATA表的元数据
SELECT DBMS_METADATA.GET_DDL('TABLE', 'T1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------------
CREATE GLOBAL TEMPORARY TABLE "SALES"."T1_METADATA"
("COL1" CLOB
) ON COMMIT DELETE ROWS
LOB ("COL1") STORE AS (
TABLESPACE "TEMP" ENABLE STORAGE IN ROW)
ORGANIZATION HEAP

-- PARTITION TABLE
DROP TABLE t1_metadata;
CREATE TABLE t1_metadata (col1 INT)
PARTITION BY RANGE(col1)
(PARTITION p1 VALUES LESS than(1),
PARTITION p2 VALUES LESS than(MAXVALUE));

-- 查询T1_METADATA表的元数据
SELECT DBMS_METADATA.GET_DDL('TABLE', 'T1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------------
CREATE TABLE "SALES"."T1_METADATA"
("COL1" INTEGER
```

```
) PCTFREE 8 INITRANS 2 MAXTRANS 255
LOGGING
TABLESPACE "USERS"
SEGMENT CREATION DEFERRED
PARTITION BY RANGE ("COL1")
(PARTITION "P1" VALUES LESS THAN (1)
PCTFREE 8 INITRANS 2 MAXTRANS 255
TABLESPACE "USERS",
PARTITION "P2" VALUES LESS THAN (MAXVALUE)
PCTFREE 8 INITRANS 2 MAXTRANS 255
TABLESPACE "USERS")
ORGANIZATION HEAP

-- LSC TABLE
DROP TABLE t1_metadata;
CREATE TABLE t1_metadata (col1 INT, col2 INT)
ORGANIZATION LSC
COMPRESSION lz4 high
ORDER BY (col2, col1)
MCOL TTL '1' month;

-- 查询T1_METADATA表的元数据
SELECT DBMS_METADATA.GET_DDL('TABLE', 'T1_METADATA') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------
CREATE TABLE "SALES"."T1_METADATA"
("COL1" INTEGER COMPRESSION LZ4 HIGH ENCODING PLAIN,
"COL2" INTEGER COMPRESSION LZ4 HIGH ENCODING PLAIN
) PCTFREE 8 INITRANS 2 MAXTRANS 255
LOGGING
TABLESPACE "USERS"
SEGMENT CREATION DEFERRED
COMPRESSION LZ4 HIGH
ORDER BY ("COL2","COL1") ASC NULLS FIRST
MCOL TTL '2678400' SECOND
ORGANIZATION LSC
ENABLE ROW MOVEMENT

-- 7、MATERIALIZED VIEW

-- 创建MV1物化视图
CREATE TABLE MV1_metadata(col1 INT, col2 INT);
CREATE MATERIALIZED VIEW MV1 BUILD IMMEDIATE REFRESH COMPLETE ON DEMAND START WITH SYSDATE+1/24 NEXT SYSDATE+3/24 AS SELECT *
FROM MV1_metadata;

-- 查询MV1物化视图的元数据
SELECT DBMS_METADATA.GET_DDL('MATERIALIZED VIEW', 'MV1') FROM dual;
DBMS_METADATA.GET_DD
--------------------------------------------------------------
CREATE MATERIALIZED VIEW "SALES"."MV1" ("COL1","COL2")
TABLESPACE SYSTEM
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND START WITH SYSDATE+1/24 NEXT SYSDATE+3/24
DISABLE QUERY REWRITE
AS SELECT "COL1","COL2" FROM MV1_metadata
```

# DBMS_MVIEW

DBMS_MVIEW包提供了一组内置的存储过程，用于物化视图刷新相关操作。

## REFRESH

```
DBMS_MVIEW.REFRESH(
    tab IN VARCHAR,
    method IN VARCHAR,
    refresh_after_errors IN BOOLEAN DEFAULT FALSE,
    purge_option IN INTEGER DEFAULT 1,
    parallelism IN INTEGER DEFAULT 0,
    atomic_refresh IN BOOLEAN DEFAULT TRUE,
    nested IN BOOLEAN DEFAULT FALSE,
    out_of_place IN BOOLEAN DEFAULT FALSE);
```

REFRESH程序用于指定一个物化视图手动刷新数据。

| 参数 | 描述 |
| --- | --- |
| tab | 物化视图名字。 |
| method | 物化视图刷新模式，默认为C（全量刷新），可省略。 |
| refresh_after_errors | 兼容参数，仅列表兼容。 |
| purge_option | 兼容参数，仅列表兼容。 |
| parallelism | 并行度兼容参数，仅列表兼容。 |
| atomic_refresh | 指定TRUE时，刷新行为是delete全表+insert into select；指定为FALSE时，刷新行为是truncate全表+insert into select |
| nested | 嵌套刷新，兼容参数，仅列表兼容。 |
| out_of_place | 兼容参数，仅列表兼容。 |

示例（单机HEAP表）

```
--创建物化视图并指定为手动刷新模式
CREATE MATERIALIZED VIEW TEST_MVIEW
REFRESH ON DEMAND
AS SELECT * FROM area;

--执行高级包刷新物化视图数据
EXEC DBMS_MVIEW.REFRESH('TEST_MVIEW');
```

# DBMS_OUTPUT

DBMS_OUTPUT包提供了一组内置的存储过程，主要用于调试代码时输出变量、表达式的值以及生成报表等功能。

- 启用DBMS_OUTPUT包通过set serveroutput on 命令实现。
- 停用DBMS_OUTPUT包通过set serveroutput off命令实现，默认情况下是停用的。
- 每行输出大小不能超过32000bytes。

DBMS_OUTPUT包含如下子程序：

| 子程序 | 描述 |
| --- | --- |
| PUT | 向缓冲区输入文本，不输出换行符。 |
| PUT_LINE | 向缓冲区输入文本和一个换行符，将缓冲区中的所有文本输出，之后清空换行符。 |

示例

```
SET serveroutput ON;
BEGIN
   DBMS_OUTPUT.PUT('yashanDB');
   DBMS_OUTPUT.PUT_LINE('hello world!');
   DBMS_OUTPUT.PUT('coming');
END;
/

--result
yashanDBhello world!
coming
```

# DBMS_PARAM

DBMS_PARAM包提供了一组变量、函数和存储过程，用于设置数据库的推荐参数。 拥有ALTER SYSTEM权限的用户可以根据业务类型和系统资源状况，使用该高级包查看和应用推荐参数。

## OPTIMIZE

```
DBMS_PARAM.OPTIMIZE (
    apply_parameter     BOOL,
    table_type          VARCHAR,
    os_memory_limit     NUMBER,
    os_cpu_limit        NUMBER,
    data_path           VARCHAR,
    redo_path           VARCHAR
);
```

此存储过程用于生成推荐参数，旧的推荐参数将会被覆盖。

生成推荐参数的过程中，系统会测试datafile和redofile所在的磁盘性能，因此建议在没有业务的情况下执行，以免影响推荐参数的结果准确性。

此存储过程的所有参数都有默认值，最少输入0个参数，最多输入6个参数。

| 参数 | 描述 |
| --- | --- |
| apply_parameter | 生成推荐参数后，是否立刻写入配置文件，默认为FALSE |
| table_type | 主要业务的表类型，可选[HEAP,TAC,LSC]，默认为HEAP |
| os_memory_limit | 内存限制百分比，默认为100，即分配所有内存 |
| os_cpu_limit | CPU限制百分比，默认为100，即分配所有CPU |
| data_path | datafile所在路径，默认为''，系统会自动获取datafile路径 |
| redo_path | redofile所在路径，默认为''，系统会自动获取redofile路径 |

注意事项：

1. 数据库最小可用内存为1.5G，最小可用CPU核数为1，如果分配的内存或CPU过小，将按最小值计算推荐参数，而不会报错。
2. NOMOUNT模式下，data_path和redo_path默认都是$YASDB_DATA/dbfiles，MOUNT和OPEN模式下，data_path默认为随机一个datafile所在路径，redo_path默认为随机一个redofile所在路径。
3. 写入配置文件后，需要重启数据库才能生效参数。
4. 如果同一台服务器上部署多个数据库实例，建议手动设置os_memory_limit，将系统内存按比例划分，防止内存不足。
5. 在生成推荐参数时，会在相应目录创建名为"_io_diag.tmp"的临时文件，正常情况下程序结束后自动删除临时文件。异常宕机时可能会残留，需要手动清理。

示例

```
-- 使用默认参数生成推荐参数，不写入配置文件。
BEGIN
    DBMS_PARAM.OPTIMIZE();
END;
/

-- 使用TAC表类型，分配80%的内存，100%的CPU，生成推荐参数后，写入配置文件。
EXEC DBMS_PARAM.OPTIMIZE(True, 'TAC', 80);

-- 使用默认的HEAP表类型，分配100%的内存，100%的CPU，生成推荐参数后，不写入配置文件。
EXEC DBMS_PARAM.OPTIMIZE(NULL, NULL, 100, 100);

-- 使用LSC表类型，分配100%的内存，100%的CPU，指定datafile和redofile的路径，生成推荐参数后，不写入配置文件。
EXEC DBMS_PARAM.OPTIMIZE(NULL, 'LSC', NULL, NULL, '/home/yashan/data', '/home/yashan/redo');
```

## SHOW_RECOMMEND

```
DBMS_PARAM.SHOW_RECOMMEND();
```

SHOW_RECOMMEND为一个函数，返回最近一次推荐参数的报告。

示例

```
SELECT DBMS_PARAM.SHOW_RECOMMEND() FROM dual;

DBMS_PARAM.SHOW_RECO
------------------------------------------------------------
********** Recommended Settings For LSC Table ************
+--------------------------------+-------------+-------------+---------+
|            name                |  current    |  recommend  | restart |
+--------------------------------+-------------+-------------+---------+
| DATA_BUFFER_SIZE               |       256M  |        128M |  True   |
| VM_BUFFER_SIZE                 |        32M  |        840M |  True   |
| WORK_AREA_STACK_SIZE           |      1024K  |          2M |  True   |
| WORK_AREA_POOL_SIZE            |       128M  |         64M |  True   |
| WORK_AREA_HEAP_SIZE            |         2M  |          2M |  True   |
| SHARE_POOL_SIZE                |       256M  |       1680M |  True   |
| LARGE_POOL_SIZE                |        32M  |        384M |  True   |
| MAX_PARALLEL_WORKERS           |       128   |         32  |  True   |
| SCOL_DATA_BUFFER_SIZE          |       128M  |        128M |  True   |
| SCOL_DATA_PRELOADERS           |         2   |          8  |  True   |
| COLUMNAR_WORK_AREA_HEAP_SIZE   |        64M  |         32M |  True   |
| COLUMNAR_VM_BUFFER_SIZE        |         2G  |      17645M |  True   |
| COLUMNAR_BULK_SIZE             |      1024   |       4096  |  True   |
| COMPRESSION                    |       LZ4   |        LZ4  |  True   |
| PQ_POOL_SIZE                   |       128M  |       3361M |  True   |
| MAX_SESSIONS                   |      1024   |        285  |  True   |
| MAX_WORKERS                    |         0   |        100  |  True   |
| TAB_QUEUE_WINDOW_SIZE          |         4   |        210  |  True   |
| BLOOM_FILTER_FACTOR            |        .3   |         .5  |  True   |
| DEGREE_OF_PARALLEL             |         1   |          4  |  True   |
| MMS_DATA_LOADERS               |         4   |          8  |  True   |
| CHECKPOINT_INTERVAL            |    100000   |        256M |  False  |
| CHECKPOINT_TIMEOUT             |       300   |         60  |  False  |
| REDOFILE_IO_MODE               |     DSYNC   |       DSYNC |  True   |
| DATAFILE_IO_MODE               |   DEFAULT   |     DEFAULT |  True   |
| COMMIT_LOGGING                 | IMMEDIATE   |   IMMEDIATE |  False  |
| RECOVERY_PARALLELISM           |        16   |          8  |  True   |
| REDO_BUFFER_SIZE               |        8M   |         32M |  True   |
+--------------------------------+-------------+-------------+---------+
| total memory                   |                           25064M |
+--------------------------------+-------------+-------------+---------+
Note: You can execute 'DBMS_PARAM.APPLY_RECOMMEND()' to apply the recommend parameters.
      After applying the parameters, you need to restart the database.
```

## APPLY_RECOMMEND

```
DBMS_PARAM.APPLY_RECOMMEND();
```

此存储过程应用推荐参数，将最近一次的推荐参数写入配置参数文件。执行后必须重启数据库，以便让参数生效。

示例

```
EXEC DBMS_PARAM.APPLY_RECOMMEND();
```

## SHOW_MEMORY_LIMIT

```
DBMS_PARAM.SHOW_MEMORY_LIMIT();
```

SHOW_MEMORY_LIMIT为一个函数，返回当前配置项下，系统使用内存的上限值。

示例

```
SELECT DBMS_PARAM.SHOW_MEMORY_LIMIT() FROM dual;

DBMS_PARAM.SHOW_MEMO
------------------------------------------------------------
********** Default table type: LSC ************
+--------------------------------+----------+--------+----------+
|            name                | setting  |  num   |  memory  |
+--------------------------------+----------+--------+----------+
| DATA_BUFFER_SIZE               |     256M |      1 |     256M |
| VM_BUFFER_SIZE                 |      32M |      1 |      32M |
| WORK_AREA_POOL_SIZE            |     128M |      1 |     128M |
| SHARE_POOL_SIZE                |     256M |      1 |     256M |
| LARGE_POOL_SIZE                |      32M |      1 |      32M |
| SCOL_DATA_BUFFER_SIZE          |     128M |      1 |     128M |
| COLUMNAR_VM_BUFFER_SIZE        |     128M |      1 |     128M |
| PQ_POOL_SIZE                   |     128M |      1 |     128M |
| REDO_BUFFER_SIZE               |       8M |      2 |      16M |
| MAX_WORKERS                    |       16 |        |          |
| MAX_PARALLEL_WORKERS           |      128 |        |          |
| WORK_AREA_STACK_SIZE           |    1024K |    144 |     144M |
| private log buffer size        |     128K |    144 |      18M |
| MAX_SESSIONS                   |      128 |        |          |
| WORK_AREA_HEAP_SIZE            |       2M |    128 |     256M |
| reserved memory for db         |     200M |      1 |     200M |
+--------------------------------+----------+--------+----------+
| memory limit                   |          |        | 1.68164G |
+--------------------------------+----------+--------+----------+
```

# DBMS_PICKLER

DBMS_PICKLER包提供了一个内置的函数，用于查询UDT元数据信息。

## GET_TYPE_SHAPE

```
DBMS_PICKLER.GET_TYPE_SHAPE (
    FULLTYPENAME     IN  VAECHAR,
    TYPOID           OUT BIGINT,
    VERSION          OUT INT,
    TDS              OUT RAW,
    INSTANTIABLE     OUT VARCHAR,
    SUPERTYPE_OWNER  OUT VARCHAR,
    SUPERTYPE_NAME   OUT VARCHAR,
    ATTR_RC          OUT SYS_REFCURSOR,
    SUBTYPE_RC       OUT SYS_REFCURSOR
) return INT;
```

GET_TYPE_SHAPE函数实现了从`SYS.USER$`等基础表中查询类型信息及其子类型信息。

函数参数信息：

| 参数 | 描述 | 类型 | IN/OUT |
| --- | --- | --- | --- |
| 返回值 | tdsFlag，标识TDS数据是否通过LOB发送，非0表示LOB发送。 | INT | OUT |
| FULLTYPENAME | 自定义类型全名称，oracle将其作为in参数。 | VARCHAR | IN/OUT |
| TYPOID | 自定义类型 oid。 | BIGINT | OUT |
| VERSION | 类型版本号，`alter type`后自增，初始版本为1。 | INT | OUT |
| TDS | `Type Descriptor Source`，包含`attr numbers`，n个属性`type code`；如果`tdsFlag`为1，标识该字段是LOB发送。 | RAW | OUT |
| INSTANTIABLE | 是否可实例化，通常为YES。 | VARCHAR | OUT |
| SUPERTYPE_OWNER | 父类型所属schema。 | VARCHAR | OUT |
| SUPERTYPE_NAME | 父类型名称。 | VARCHAR | OUT |
| ATTR_RC | 属性元数据信息 cursor。 | SYS_REFCURSOR | OUT |
| SUBTYPE_RC | 子类型元数据信息cursor。 | SYS_REFCURSOR | OUT |

ATTR_RC属性元数据信息：

| 字段 | 说明 |
| --- | --- |
| 1 | 固定值，1，`image format`。 |
| NAME | 属性名称。 |
| ATTRIBUTE | 属性序号，表示类型的第几个属性；从1开始。 |
| DTypeName | 类型名称，比如VARCHAR2，NUMBER，NVARCHAR2，CLOB。 |
| USER.NAME | 属性类型（是自定义类型）owner的名称，普通类型为null。 |
| ATTR_TOID | 属性的`type id`，如果属性类型也是自定义类型。 |
| ATTRTYPINS | 类型是否可实例化，通常都是YES。 |
| SUP_TYP_OWNER | 父类型的schema，如果属性类型有父类型。 |

| 字段 | 说明 |
| --- | --- |
| SUP_TYP_NAME | 父类型的 `type name` ，如果属性类型有父类型。 |

SUBTYPE_RC子类型元数据信息：

| 字段 | 说明 |
| --- | --- |
| 1 | 固定值，1， `image format` 。 |
| NAME | 子类型owner的名称。 |
| NAME | 子类型名称。 |
| OID | 子类型的 `type oid` 。 |

示例：

```
-- 创建UDT类型
CREATE OR replace TYPE SALES.UDT_VARRAY_TYP IS object(id INT, value VARCHAR(20));
/

-- 启用dbms_output
SET serveroutput ON;
-- 匿名块
DECLARE
   tds_flag INT;
   full_type_name VARCHAR(255) := 'SALES.UDT_VARRAY_TYP';
   typeoid BIGINT;
   version INT;
   tds RAW(8000);
   instantiable VARCHAR(255);
   sup_owner VARCHAR(255);
   sup_name VARCHAR(255);
   attr_rc SYS_REFCURSOR;
   subtype_rc SYS_REFCURSOR;
BEGIN
   tds_flag := sys.dbms_pickler.get_type_shape(full_type_name, typeoid, version, tds, instantiable, sup_owner, sup_name,
attr_rc, subtype_rc);
   DBMS_OUTPUT.PUT_LINE('tds flag: ' || tds_flag);
   DBMS_OUTPUT.PUT_LINE(' full_type_name: ' || full_type_name);
   DBMS_OUTPUT.PUT_LINE(' typeoid: ' || typeoid);
   DBMS_OUTPUT.PUT_LINE(' version: ' || version);
   DBMS_OUTPUT.PUT_LINE(' tds: ' || tds);
   DBMS_OUTPUT.PUT_LINE(' instantiable: ' || instantiable);
   DBMS_OUTPUT.PUT_LINE(' sup_owner: ' || sup_owner);
   DBMS_OUTPUT.PUT_LINE(' sup_name: ' || sup_name);
   DBMS_SQL.RETURN_RESULT(attr_rc);
   DBMS_SQL.RETURN_RESULT(subtype_rc);
END;
/

-- result
tds flag: 0
 full_type_name: SALES.UDT_VARRAY_TYP
 typeoid: 65581
 version: 1
 tds: 1C000000000000000102000400010000000001A0002000000000000
 instantiable: YES
 sup_owner:
 sup_name:


ResultSet #1
```

```
1 NAME                                                          ATTRIBUTE# DECODE(A.ATTR_TOID,0
DECODE(BITAND(A.PROP
ATTR_TOID ATTRTYPINS SUP_TYP_OWNER                                                  SUP_TYP_NAME
------------ ------------------------------------------------------------------ ------------ --------------------------------
------------------------------ ---------------------------------------------------------------------- -------------------- -------
--- ------------------------------------------------------------------ ----------------------------------------------------------
--------
1 ID                                                                  1 INTEGER
4 YES
1 VALUE                                                               2 VARCHAR
26 YES


ResultSet #2

1 NAME                                                          NAME
TOID
------------ ------------------------------------------------------------------ ------------------------------------------------
----------------- ---------------------
```

# DBMS_RANDOM

DBMS_RANDOM包提供了一组内置的存储过程/函数，主要用于生成随机数字与字符。

## INITIALIZE

用一个种子值来初始化DBMS_RANDOM包。

示例

```
DBMS_RANDOM.INITIALIZE(
   seed IN INTEGER);
```

示例

```
BEGIN
DBMS_RANDOM.INITIALIZE(100);
END;
/
```

## NORMAL

返回服从正态分布的随机数，无参数，'()'可省略。

```
DBMS_RANDOM.NORMAL();
```

示例

```
SELECT DBMS_RANDOM.NORMAL FROM dual;
DBMS_RANDOM.NORMAL
------------------
        -1.9243543
```

## RANDOM

返回随机数，无参数，'()'可省略。

```
DBMS_RANDOM.RANDOM();
```

示例

```
BEGIN
DBMS_RANDOM.INITIALIZE(100);
FOR i IN 1 .. 5 LOOP
 DBMS_OUTPUT.PUT_LINE(DBMS_RANDOM.RANDOM);
END LOOP;
END;
/
--result
592679413
2139893194
61410547
1016237248
628080577
```

## SEED

重置种子值。

```
DBMS_RANDOM.SEED(
  seed IN INTEGER);
```

示例

```
BEGIN
DBMS_RANDOM.SEED('15');
END;
/
```

## STRING

随机生成字符串。

```
DBMS_RANDOM.STRING(
    opt  IN  CHAR,
    len  IN  NUMBER);
```

| 参数 | 描述 |
| --- | --- |
| opt | 字符串格式：<br>* 'x','X'：返回包括数字与大写字母的字符串<br>* 'u','U'：返回只有大写字母的字符串<br>* 'l','L'：返回只有小写字母的字符串<br>* 'a','A'：返回包括大小写字母的字符串<br>* 'p','P'：返回任一ASCII码字符 |
| len | 字符串长度 |

示例

```
SELECT DBMS_RANDOM.STRING('l',10) value FROM dual;
VALUE
----------------------------------------------------------------
elwtenyjkj

SELECT DBMS_RANDOM.STRING('p',10) value FROM dual;
VALUE
----------------------------------------------------------------
g)6*gN/?Yl
```

## VALUE

生成一个指定范围的38位随机小数（小数点后38位），参数可省略，表示不指定范围，则生成范围为 `[0,1)` 的38位随机数。

```
DBMS_RANDOM.VALUE (
   low IN NUMBER,
   high IN NUMBER);
```

示例

```
SET num 40;
SELECT DBMS_RANDOM.VALUE FROM dual;
                        DBMS_RANDOM.VALUE
----------------------------------------
  .875661952392119437547428402478673589O6
```

```
SELECT DBMS_RANDOM.VALUE(1,10) FROM dual;
                    DBMS_RANDOM.VALUE(1,
----------------------------------------
  5.67403276094096544220600403896672745 21
```

# DBMS_RESOURCE_MANAGER

DBMS_RESOURCE_MANAGER包提供了一组资源管理的存储过程/函数，用于创建和删除资源使用组、资源映射及相关操作。

> **Note**:
> - 调用DBMS_RESOURCE_MANAGER高级包下的所有子程序时，都需要以SYS用户连接数据库，否则报错。
> - 分布式部署下，高级包DBMS_RESOURCE_MANAGER下的所有过程/函数只允许逐条调用，且只能在CN节点上操作。

## CREATE_CONSUMER_GROUP

```
DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    CONSUMER_GROUP IN VARCHAR(64),
    COMMENT        IN VARCHAR(2000));
```

该程序用于创建一个资源使用组。

系统包含如下默认资源使用组：

- SYS_GROUP：包括系统用户sys和系统进程，不允许修改与删除SYS_GROUP的相关配置与映射关系。
- DEFAULT_CONSUMER_GROUP：不存在资源映射关系的资源使用者默认划分至该组。

YashanDB最多支持用户自定义创建126个资源使用组。

| 参数 | 描述 |
| --- | --- |
| CONSUMER_GROUP | 资源使用组，名称唯一且符合YashanDB的对象命名规范 |
| COMMENT | 注释 |

示例

```
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    'RESGROUP1',
    'GROUP FOR CPU RESOURCE1');
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    'RESGROUP2',
    'GROUP FOR CPU RESOURCE2');
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    'RESGROUP3',
    'GROUP FOR CPU RESOURCE3');
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP4',
    COMMENT => 'GROUP FOR CPU RESOURCE4');
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP5',
    COMMENT => 'GROUP FOR CPU RESOURCE5');
EXEC DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP6',
    COMMENT => 'GROUP FOR CPU RESOURCE6');
```

## SET_CONSUMER_GROUP_MAPPING

```
DBMS_RESOURCE_MANAGER.SET_CONSUMER_GROUP_MAPPING(
    ATTRIBUTE      IN VARCHAR(64),
    VALUE          IN VARCHAR(64),
    CONSUMER_GROUP IN VARCHAR(64));
```

该程序用于把资源使用者映射到资源使用组，无法修改SYS用户与SYS_GROUP组的映射关系。

| 参数 | 描述 |
| --- | --- |
| ATTRIBUTE | 映射属性<br>仅支持USER，即资源使用者以用户为维度 |
| VALUE | 待映射的用户名，必须为已存在的用户 |
| CONSUMER_GROUP | 待映射的资源使用组 |

如果映射的资源使用组没有对应的计划指令，则表现与DEFAULT_CONSUMER_GROUP保持一致；当创建新的计划指令时，需要用户重新登录会话才能生效。

示例

```
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.SET_CONSUMER_GROUP_MAPPING(
    'USER',
    'SALES1',
    'RESGROUP1');
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.SET_CONSUMER_GROUP_MAPPING(
    ATTRIBUTE => 'USER',
    VALUE => 'SALES2',
    CONSUMER_GROUP => 'RESGROUP2');
```

## DELETE_CONSUMER_GROUP_MAPPING

```
DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP_MAPPING(
    ATTRIBUTE IN VARCHAR(64),
    VALUE     IN VARCHAR(64));
```

该程序用于删除资源使用者到资源使用组的映射，无法删除SYS用户与SYS_GROUP组的映射关系。

| 参数 | 描述 |
| --- | --- |
| ATTRIBUTE | 映射属性，仅支持USER |
| VALUE | 映射关系中的用户名，不能为SYS用户 |

示例

```
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP_MAPPING(
    'USER',
    'SALES1');
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP_MAPPING(
    ATTRIBUTE => 'USER',
    VALUE => 'SALES2');
```

## CREATE_PLAN_DIRECTIVE

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    PLAN                  IN VARCHAR(64),
    GROUP_OR_SUBPLAN      IN VARCHAR(64),
    MGMT_P1               IN INTEGER DEFAULT NULL,
    MAX_UTILIZATION_LIMIT IN INTEGER DEFAULT NULL,
    UTILIZATION_LIMT      IN INTEGER DEFAULT NULL,
```

```
    SHARES               IN INTEGER DEFAULT NULL,
    PARALLEL_SERVER_LIMIT IN INTEGER DEFAULT NULL);
```

该程序用于创建资源计划指令。

| 参数 | 描述 |
| --- | --- |
| PLAN | 资源指令计划 |
| GROUP_OR_SUBPLAN | 资源使用组 |
| MGMT_P1 | CPU共享模式下的使用份额，须为[1,100]中的整数，默认值NULL表示最小值1 |
| MAX_UTILIZATION_LIMIT | CPU最大使用上限，须为[1,100]中的整数，默认值NULL表示最大值100 |
| UTILIZATION_LIMIT | 与字段MAX_UTILIZATION_LIMIT相同表现 |
| SHARES | 与字段MGMT_P1相同表现 |
| PARALLEL_SERVER_LIMIT | 资源使用组可以使用的最大并行资源百分比，须为[0,100]中的整数，默认值NULL表示最大值100 |

使用说明：

- 当同时设置MGMT_P1和SHARES参数时，以shares参数为准。
- 当同时设置MAX_UTILIZATION_LIMIT和UTILIZATION_LIMIT参数时，以UTILIZATION_LIMIT参数为准。
- 最大CPU使用率计算公式为 `MAX_UTILIZATION_LIMIT * CPU个数` ，例如某用户映射的资源使用组中MAX_UTILIZATION_LIMIT为10，环境中各节点的CPU个数为2，则该用户在每个节点的最大CPU使用率为20%。
- 为保障SYS_GROUP具备足够的资源可用，当该组的共享资源（MGMT_P1）较少或较多时，系统会在用户输入时做动态调整。若SYS_GROUP的MGMT_P1全局占比小于40%或大于60%时，系统会将SYS_GROUP的MGMT_P1动态调整为剩余资源组的MGMT_P1总和，此时SYS_GROUP的MGMT_P1值可能会超过100（可查看视图DBA_RSRC_PLAN_DIRECTIVES的MGMT_P1字段，全局占比计算公式：`SYS_GROUP的MGMT_P1 / 所有资源使用组的MGMT_P1总和)` ）。
- 较低版本的数据库升级后，如果原本已将MGMT_P1或MAX_UTILIZATION_LIMIT参数配置为0，实际表现与当前版本中将其对应配置为100相同。

示例

```
-- 不配置可选参数
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP1');
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP2',
    10,
    20,
    30,
    40,
    50);
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP3',
    MGMT_P1 => 10,
    MAX_UTILIZATION_LIMIT => 20,
    UTILIZATION_LIMIT => 30,
    SHARES => 40,
    PARALLEL_SERVER_LIMIT => 50);
-- 只配置MGMT_P1和MAX_UTILIZATION_LIMIT参数
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP4',
    MGMT_P1 => 10,
    MAX_UTILIZATION_LIMIT => 20);
-- 只配置PARALL_SERVER_LIMIT参数
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
```

```
    GROUP_OR_SUBPLAN => 'RESGROUP5',
    PARALLEL_SERVER_LIMIT => 50);
-- 指定参数名后的参数只能继续指定参数名，以下示例会报错
EXEC DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP6',
    10);

YAS-04253 PL/SQL compiling errors:
[1:104] YAS-00003 invalid parameter, reason: param => input order error
```

## UPDATE_PLAN_DIRECTIVE

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    PLAN                 IN VARCHAR(64),
    GROUP_OR_SUBPLAN     IN VARCHAR(64),
    MGMT_P1              IN INTEGER DEFAULT NULL,
    MAX_UTILIZATION_LIMIT IN INTEGER DEFAULT NULL,
    UTILIZATION_LIMT     IN INTEGER DEFAULT NULL,
    SHARES               IN INTEGER DEFAULT NULL,
    PARALLEL_SERVER_LIMIT IN INTEGER DEFAULT NULL);
```

该程序用于更新资源计划指令。

| 参数 | 描述 |
| --- | --- |
| PLAN | 资源指令计划 |
| GROUP_OR_SUBPLAN | 资源使用组 |
| MGMT_P1 | CPU共享模式下的使用份额，须为[1,100]中的整数，默认值NULL表示最小值1 |
| MAX_UTILIZATION_LIMIT | CPU最大使用上限，须为[1,100]中的整数，默认值NULL表示最大值100 |
| UTILIZATION_LIMIT | 与字段MAX_UTILIZATION_LIMIT相同表现 |
| SHARES | 与字段MGMT_P1相同表现 |
| PARALLEL_SERVER_LIMIT | 资源使用组可以使用的最大并行资源百分比，须为[0,100]中的整数，默认值NULL表示最大值100 |

使用说明：

- 可选参数不设置值或输入为NULL时，不对该参数做任何处理。

示例

```
-- 不配置可选参数
EXEC DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP1');
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP2',
    10,
    20,
    30,
    40,
    50);
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP3',
    MGMT_P1 => 10,
    MAX_UTILIZATION_LIMIT => 20,
```

```
    UTILIZATION_LIMIT => 30,
    SHARES => 40,
    PARALLEL_SERVER_LIMIT => 50);
-- 只配置MGMT_P1和MAX_UTILIZATION_LIMIT参数
EXEC DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP4',
    MGMT_P1 => 10,
    MAX_UTILIZATION_LIMIT => 20);
-- 只配置PARALL_SERVER_LIMIT参数
EXEC DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP5',
    PARALLEL_SERVER_LIMIT => 50);
```

## DELETE_PLAN_DIRECTIVE

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    PLAN             IN VARCHAR(64),
    GROUP_OR_SUBPLAN IN VARCHAR(64));
```

该程序用于删除资源计划指令。

| 参数 | 描述 |
| --- | --- |
| PLAN | 资源指令计划 |
| GROUP_OR_SUBPLAN | 资源使用组 |

示例

```
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP1');
EXEC DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP2');
EXEC DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    'RESPLAN',
    'RESGROUP3');
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP4');
EXEC DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE(
    PLAN => 'RESPLAN',
    GROUP_OR_SUBPLAN => 'RESGROUP5');
```

## DELETE_CONSUMER_GROUP

```
DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    CONSUMER_GROUP IN VARCHAR(64));
```

该程序用于删除一个资源使用组。

| 参数 | 描述 |
| --- | --- |
| CONSUMER_GROUP | 资源使用组 |

示例

```
-- 忽略参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    'RESGROUP1');
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    'RESGROUP2');
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    'RESGROUP3');
-- 指定参数名
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP4');
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP5');
EXEC DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP(
    CONSUMER_GROUP => 'RESGROUP6');
```

# DBMS_ROWID

DBMS_ROWID包提供了一组内置的函数，用于获取ROWID的数据块号blockId、相对文件号fileId和行号dir信息。

## ROWID_BLOCK_NUMBER

```
DBMS_ROWID.ROWID_BLOCK_NUMBER (
   row_id      IN  ROWID,
   ts_type_in  IN  VARCHAR DEFAULT 'SMALLFILE')
RETURN NUMBER;
```

ROWID_BLOCK_NUMBER函数用于获取指定ROWID所对应的数据块号blockId，返回值为NUMBER类型。

| 参数 | 描述 |
| --- | --- |
| row_id | 待解释的ROWID。该参数的数据类型可为ROWID/UROWID/RAW/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| ts_type_in | 保留字段，使用缺省值，默认值为'SMALLFILE'，不区分大小写，且仅接受SMALLFILE、BIGFILE。该参数的数据类型可以为VARCHAR类型和可以隐式转换为VARCHAR类型的其他数据类型，详情参阅数据类型转换。 |

示例

```
CREATE TABLE tbl_rowid(ID INT);
INSERT INTO tbl_rowid VALUES(1);
INSERT INTO tbl_rowid VALUES(2);

SELECT ROWID, DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID)
FROM tbl_rowid
WHERE ID = 2;
--result
ROWID                                      DBMS_ROWID.ROWID_BLO
------------------------------------------ --------------------
2801:0:0:3572:1                                            3572
```

## ROWID_RELATIVE_FNO

```
DBMS_ROWID.ROWID_RELATIVE_FNO (
   row_id      IN  ROWID,
   ts_type_in  IN  VARCHAR DEFAULT 'SMALLFILE')
RETURN NUMBER;
```

ROWID_RELATIVE_FNO函数用于获取指定ROWID所对应的相对文件号fileId，返回值为NUMBER类型。

| 参数 | 描述 |
| --- | --- |
| row_id | 待解释的ROWID。该参数的数据类型可为ROWID/UROWID/RAW/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |
| ts_type_in | 保留字段，使用缺省值，默认值为'SMALLFILE'，不区分大小写，且仅接受SMALLFILE、BIGFILE。该参数的数据类型可以为VARCHAR类型和可以隐式转换为VARCHAR类型的其他数据类型，详情参阅数据类型转换。 |

示例

```
SELECT ROWID, DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID)
FROM tbl_rowid
WHERE ID = 2;
--result
ROWID                                      DBMS_ROWID.ROWID_REL
```

```
---------------------------------------- --------------------
2801:0:0:3572:1                                             0
```

## ROWID_ROW_NUMBER

```
DBMS_ROWID.ROWID_ROW_NUMBER (
   row_id      IN  ROWID)
RETURN NUMBER;
```

ROWID_ROW_NUMBER函数用于获取指定ROWID所对应的行号dir，返回值为NUMBER类型。

| 参数 | 描述 |
| --- | --- |
| row_id | 待解释的ROWID。该参数的数据类型可为ROWID/UROWID/RAW/CHAR/VARCHAR/NCHAR/NVARCHAR类型。 |

示例

```
SELECT ROWID, DBMS_ROWID.ROWID_ROW_NUMBER(ROWID)
FROM tbl_rowid
WHERE ID = 2;
--result
ROWID                                       DBMS_ROWID.ROWID_ROW
---------------------------------------- --------------------
2801:0:0:3572:1                                             1
```

# 异常说明

| 异常 | 描述 |
| --- | --- |
| ROWID_INVALID | 无效的rowid |

示例

```
--rowid_invalid exception
DECLARE
  BLK NUMBER;
  RID VARCHAR(100) := 'aaa';
BEGIN
  BLK := DBMS_ROWID.ROWID_BLOCK_NUMBER(RID);
EXCEPTION
  WHEN DBMS_ROWID.ROWID_INVALID THEN
    DBMS_OUTPUT.PUT_LINE('invalid rowid: ' || RID);
END;
/
--result
invalid rowid: aaa
```

# DBMS_SCHEDULER

DBMS_SCHEDULER包提供了一组内置的存储过程，用于创建和管理定时任务。

## CREATE_JOB

```
DBMS_SCHEDULER.CREATE_JOB (
    job_name IN VARCHAR,
    job_type IN VARCHAR,
    job_action IN VARCHAR ,
    number_of_arguments IN INTEGER DEFAULT 0,
    start_date IN TIMESTAMP DEFAULT NULL,
    repeat_interval IN VARCHAR DEFAULT NULL,
    end_date IN TIMESTAMP DEFAULT NULL,
    job_class IN VARCHAR DEFAULT 'DEFAULT_JOB_CLASS',
    enabled IN BOOLEAN DEFAULT FALSE,
    auto_drop IN BOOLEAN DEFAULT TRUE,
    comments IN VARCHAR DEFAULT NULL);
```

CREATE_JOB程序用于创建一个新的定时任务，成功创建的定时任务将可以在DBA_SCHEDULER_JOBS/ALL_SCHEDULER_JOBS/USER_SCHEDULER_JOBS视图中查询。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式，且需符合YashanDB的对象命名规范。 |
| job_type | job_action的类型。'PLSQL_BLOCK'表示匿名块，'STORED_PROCEDURE'表示存储过程。 |
| job_action | 定时任务要执行的PL文本，可以是具体的匿名块或存储过程，必须以分号结束，且应当与job_type匹配。 |
| number_of_arguments | 保留字段，使用缺省值。 |
| start_date | 定时任务开始执行的时间，NULL表示立即执行。 |
| repeat_interval | 表达式文本，用于计算定时任务下次执行的时间，表达式为NULL表示定时任务只执行一次。根据日期时间型算术计算规则，repeat_interval参数所输入的间隔以天为单位，例如SYSDATE+1表示下次执行任务时间为当前时间的后一天。通过表达式计算出的时间必须为将来时间或NULL。 |
| end_date | 定时任务执行结束的时间，超过该时间后定时任务将不再自动执行。 |
| job_class | 保留字段，使用缺省值。 |
| enabled | 定时任务是否生效，默认为false，不生效。只有生效的定时任务才会自动执行。 |
| auto_drop | 当定时任务完成后是否自动删除。<br>满足以下条件之一时，定时任务会被标记为完成：<br>* 当前时间已超过定时任务的end_date。<br>* 用户设置了定时任务的最大执行次数，且定时任务自动执行的次数已达到最大执行次数。<br>* 不重复执行（未设置repeat_interval）的定时任务已执行了一次。 |
| comments | 为定时任务添加的描述信息，默认为NULL。 |

示例

```
EXEC DBMS_SCHEDULER.CREATE_JOB(
    'sche_example',
    'PLSQL_BLOCK',
    'begin insert into area values(TO_CHAR(SYSDATE,''SS''),''sche'',''sche example''); commit; end;',
    0,
    SYSDATE+10/24/60,
    'SYSDATE+1',
    NULL,
    'DEFAULT_JOB_CLASS',
```

```
    true,
    true,
    NULL
    );
```

## RUN_JOB

```
DBMS_SCHEDULER.RUN_JOB(
    job_name             IN VARCHAR,
    use_current_session IN BOOL DEFAULT TRUE);
```

RUN_JOB程序会手动执行一次定时任务，定时任务的状态会变为RUNNING，手动执行的任务无法通过STOP_JOB程序停止。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式。 |
| use_current_session | 当参数值为true时，在当前session下手动执行定时任务；当参数值为false时，另起一个后台线程手动执行定时任务。 |

示例

```
EXEC DBMS_SCHEDULER.RUN_JOB('sales.sche_example');
```

## DISABLE

```
DBMS_SCHEDULER.DISABLE(
    job_name         IN VARCHAR,
    force            IN BOOL DEFAULT FALSE,
    commit_semantics IN VARCHAR DEFAULT 'STOP_ON_FIRST_ERROR');
```

DISABLE程序用于使一个非执行状态下的定时任务失效，失效后的定时任务将不再被系统调度。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式。 |
| force | 保留字段，使用缺省值。 |
| commit_semantics | 保留字段，使用缺省值。 |

示例

```
EXEC DBMS_SCHEDULER.DISABLE('sales.sche_example');
```

## ENABLE

```
DBMS_SCHEDULER.ENABLE(
    job_name IN VARCHAR,
    commit_semantics IN VARCHAR DEFAULT 'STOP_ON_FIRST_ERROR');
```

ENABLE程序用于使一个定时任务生效。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式。 |

| 参数 | 描述 |
| --- | --- |
| commit_semantics | 保留字段，使用缺省值。 |

示例

```
EXEC DBMS_SCHEDULER.ENABLE('sales.sche_example');
```

## SET_ATTRIBUTE

```
DBMS_SCHEDULER.SET_ATTRIBUTE(
    job_name IN VARCHAR,
    attribute IN VARCHAR,
    value IN {BOOLEAN|DATE|TIMESTAMP|VARCHAR|INTEGER},
    value2 IN VARCHAR DEFAULT NULL);
```

SET_ATTRIBUTE程序用于设置定时任务的属性。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式。 |
| attribute | 要修改的属性名称，可以为'auto_drop','comment','end_date','job_action','repeat_interval','start_date','instance_id'。 |
| value | 要修改的属性值，其类型需与attribute对应类型一致。 |
| value2 | 语法兼容，无实际含义。 |

示例（分布式部署）

```
-- 设置sche_example任务的执行频率
EXEC DBMS_SCHEDULER.SET_ATTRIBUTE('sche_example','repeat_interval','SYSDATE+0.003');

-- 设置sche_example任务的结束时间为1分钟后
EXEC DBMS_SCHEDULER.SET_ATTRIBUTE('sche_example','end_date',SYSDATE+1/24/60);

-- 设置sche_example任务为完成后自动删除
EXEC DBMS_SCHEDULER.SET_ATTRIBUTE('sche_example','auto_drop',true);
```

## STOP_JOB

示例

```
DBMS_SCHEDULER.STOP_JOB(
    job_name            IN VARCHAR,
    force               IN BOOLEAN DEFAULT FALSE,
    commit_semantics    IN VARCHAR DEFAULT 'STOP_ON_FIRST_ERROR');
```

STOP_JOB程序用于停止正在运行的定时任务，但无法停止通过RUN_JOB程序手动执行的定时任务。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式，如果是设置多个定时任务名称，中间用逗号分隔。 |
| force | 语法兼容，无实际含义。 |
| commit_semantics | 语法兼容，无实际含义。 |

示例

```
EXEC DBMS_SCHEDULER.STOP_JOB('sales.sche_example');

EXEC DBMS_SCHEDULER.STOP_JOB('sche_example, sales.sche_example2');
```

## DROP_JOB

```
DBMS_SCHEDULER.DROP_JOB(
    job_name         IN VARCHAR,
    force            IN BOOL DEFAULT FALSE,
    defer            IN BOOL DEFAULT FALSE,
    commit_semantics IN VARCHAR DEFAULT 'STOP_ON_FIRST_ERROR');
```

DROP_JOB程序用于删除一个非执行状态下的定时任务。

| 参数 | 描述 |
| --- | --- |
| job_name | 定时任务名称，可以为schema.job_name格式。 |
| force | 保留字段，使用缺省值。 |
| defer | 保留字段，使用缺省值。 |
| commit_semantics | 保留字段，使用缺省值。 |

示例

```
EXEC DBMS_SCHEDULER.DROP_JOB('sales.sche_example');
```

# DBMS_SQL

DBMS_SQL包提供了一组内置接口，用于执行动态SQL，包括DDL和DML。

## RETURN_RESULT

示例

```
DBMS_SQL.RETURN_RESULT (
rc              IN OUT SYS_REFCURSOR,
to_client       IN BOOLEAN DEFAULT TRUE);
```

RETURN_RESULT为存储过程，用于执行指定SQL并返回结果集给客户端（例如接口程序），客户端获得结果集游标后，可以执行fetch语句查询结果。

当在yasql中执行本存储过程时，将直接输出结果集内容。

| 参数 | 描述 |
| --- | --- |
| rc | 动态游标 |
| to_client | 是否返回结果集给客户端；默认为true，如果传入false并且是执行动态SQL时，结果集不返回给客户端。 |

**Note**：

- 当前只有查询结果集可以返回。
- 执行返回结果集后，rc参数对应的游标不能再访问。
- 客户端将缓存结果集全部结果，但依赖客户端机的内存大小。
- 过程体执行过程中如果发生错误，错误点之前返回的结果集，客户端依然可以获取到。

示例

```
CREATE OR REPLACE PROCEDURE return_rs_proc IS
cur1 SYS_REFCURSOR;
cur2 sys_refcursor;
BEGIN
  OPEN cur1 FOR SELECT area_no,area_name FROM area WHERE area_no='01';
  DBMS_SQL.RETURN_RESULT(cur1,false);
  OPEN cur2 FOR SELECT branch_no,branch_name FROM branches WHERE area_no='01';
  DBMS_SQL.RETURN_RESULT(cur2);
END;
/

--执行静态SQL
exec return_rs_proc;

ResultSet #1

AREA_NO AREA_NAME
------- ------------------------------------------------------------
01      华东


ResultSet #2

BRANCH_NO BRANCH_NAME
--------- ------------------------------------------------------------
0101      上海
0102      南京
0103      福州
0104      厦门
```

```
--执行动态SQL，此时由于to_client为false，第一个结果集不会返回
BEGIN
  EXECUTE IMMEDIATE 'begin return_rs_proc; end;';
END;
/

ResultSet #1

BRANCH_NO BRANCH_NAME
--------- ----------------------------------------------------------------
0101      上海
0102      南京
0103      福州
0104      厦门
```

# DBMS_STANDARD

DBMS_STANDARD包提供了一组内置的存储过程/函数，主要用于PL中的异常处理，事务处理，及外置java函数控制。

调用DBMS_STANDARD包的子程序时，可以省略DBMS_STANDARD，直接使用子程序名。

## RAISE_APPLICATION_ERROR

示例

```
DBMS_STANDARD.RAISE_APPLICATION_ERROR(
    errCode         IN INTEGER,
    errMsg          IN VARCHAR,
    reserveErrStack IN BOOLEAN DEFAULT FALSE);
```

RAISE_APPLICATION_ERROR存储过程用于在PL程序中手动抛出异常。

| 参数 | 描述 |
| --- | --- |
| errCode | 错误号，范围在[30000, 99999]。 |
| errMsg | 错误消息，长度不大于1024bytes。错误消息字符串的空白符后缀会被移除。 |
| reserveErrStack | 保留字段，仅语法兼容。 |

示例

```
CREATE OR replace PROCEDURE account_status (
    due_date DATE,
    today DATE) IS
BEGIN
    IF due_date < today THEN -- explicitly RAISE EXCEPTION
      DBMS_STANDARD.RAISE_APPLICATION_ERROR(29999+1, 'Account past due.');
    END IF;
END;
/

DECLARE
    past_due EXCEPTION; -- DECLARE EXCEPTION
    PRAGMA EXCEPTION_INIT (past_due, 30000);
BEGIN
    account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
    TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));
EXCEPTION
    WHEN past_due THEN -- handle EXCEPTION
        DBMS_OUTPUT.PUT_LINE('SUCCESS!');
END;
/

--result
SUCCESS!
```

## LOADJAVA

示例

```
DBMS_STANDARD.LOADJAVA(
    javaPath IN VARCHAR);
```

LOADJAVA存储过程用于加载一个java文件，在外置UDF场景中使用。

YashanDB会在当前用户下自动创建一个自定义库对象，对象的名称由输入的路径按照一定规则生成，区分大小写，需要符合YashanDB的对象命名规范。

YashanDB已提供直接创建自定义库的功能，此方式不再建议使用。

| 参数 | 描述 |
| --- | --- |
| javaPath | 需要加载的class文件或者jar路径，长度不超过255字节。 |

示例

```
call DBMS_STANDARD.LOADJAVA('/mnt/d/com/test/Hello.class');
```

## DROPJAVA

```
DBMS_STANDARD.DROPJAVA(
    javaPath IN VARCHAR);
```

DROPJAVA存储过程用于卸载一个java文件，在外置UDF场景中使用。

| 参数 | 描述 |
| --- | --- |
| javaPath | 需要卸载的class文件或者jar路径，长度不超过255字节。 |

示例

```
call DBMS_STANDARD.DROPJAVA('/mnt/d/com/test/Hello.class');
```

## INSERTING

INSERTING函数用于在INSERT语句中执行的触发器内返回TRUE，该函数无参数，使用请参考触发器中的例程。

## DELETING

DELETING函数用于在DELETE语句中执行的触发器内返回TRUE，该函数无参数，使用请参考触发器中的例程。

## UPDATING

示例

```
DBMS_STANDARD.UPDATING(
    colName IN VARCHAR);
```

UPDATING函数用于在UPDATE语句中执行的触发器内更新的列与输入参数匹配时返回TRUE。

| 参数 | 描述 |
| --- | --- |
| colName | UPDATE操作中更新的列名。当更新的列与colName一致时返回TRUE。当没有设置colName时，更新任何列都返回TRUE。 |

使用请参考触发器中的例程。

# DBMS_STATS

DBMS_STATS包提供了一组变量、函数和存储过程，用于管理优化器的统计信息。

除GATHER_TABLE_STATS、GATHER_SCHEMA_STATS和GATHER_DATABASE_STATS外，其他函数/存储过程不适用于分布式部署。

## 手动设置统计信息

### SET_TABLE_STATS

```
DBMS_STATS.SET_TABLE_STATS (
    ownname     VARCHAR,
    tabname     VARCHAR,
    partname    VARCHAR,
    numrows     NUMBER,
    numblks     NUMBER,
    avgrlen     NUMBER
);
```

此存储过程用于手动设置表/AC或者分区的统计信息，旧的统计信息将会被覆盖，主要在测试需要或者统计信息修正场景中使用。

此存储过程不适用于组合分区表。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表/AC名 |
| partname | 分区名 |
| numrows | 表/AC或者分区的行数 |
| numblks | 表/AC或者分区的数据块数 |
| avgrlen | 表/AC或者分区的平均行长度 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.SET_TABLE_STATS( 'SALES', 'SALES_INFO_RANGE', 'P_SALES_INFO_RANGE_1', 10, 1, 15);
END;
/
```

### SET_INDEX_STATS

```
DBMS_STATS.SET_INDEX_STATS (
    ownname      VARCHAR,
    indname      VARCHAR,
    partname     VARCHAR,
    numrows      NUMBER,
    numlblks     NUMBER,
    numdist      NUMBER,
    numfirstdist NUMBER,
    avglblk      NUMBER,
    avgdblk      NUMBER,
    clstfct      NUMBER,
    indlevel     NUMBER
);
```

此存储过程用于手动设置索引或者分区的统计信息，旧的统计信息将会被覆盖，主要在测试需要或者统计信息修正场景中使用。

此存储过程不适用于组合分区表的索引信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| indname | 索引名 |
| partname | 索引分区名 |
| numrows | 索引键个数 |
| numlblks | 叶子块个数 |
| numdist | 索引键唯一值个数 |
| numfirstdist | 组合索引第一列的键唯一值个数 |
| avglblk | 平均每个叶子块包含键的个数 |
| avgdblk | 平均每个叶子块指向数据块的个数 |
| clstfct | 索引的聚集因子 |
| level | BTree的高度 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.SET_INDEX_STATS('SALES', 'IDX_FINANCE_INFO_1', null, 100, 10, 10, 1, 10, 10, 1, 1);
END;
/
```

## SET_COLUMN_STATS

```
DBMS_STATS.SET_COLUMN_STATS (
    ownname    VARCHAR,
    tabname    VARCHAR,
    colname    VARCHAR
    partname   VARCHAR,
    distinct   NUMBER,
    density    NUMBER,
    nullcnt    NUMBER,
    avgclen    NUMBER,
);
```

此存储过程用于手动设置表/AC或者分区的列统计信息，旧的统计信息将会被覆盖，主要在测试需要或者统计信息修正场景中使用。

此存储过程不适用于组合分区表。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表/AC名 |
| colname | 列名 |
| partname | 分区名 |
| distinct | 列distinct数量 |
| density | 列density，取值范围为[0,1] |
| nullcnt | 列NULL值数量 |

| 参数 | 描述 |
| --- | --- |
| avgclen | 平均列长度 |

示例（单机、共享集群部署）

```
BEGIN
  DBMS_STATS.SET_COLUMN_STATS( 'SALES', 'SALES_INFO_RANGE', 'AMOUNT', 'P_SALES_INFO_RANGE_1', 10, 0.2, 15, 10);
END;
/
```

## 删除统计信息

### DELETE_TABLE_STATS

```
DBMS_STATS.DELETE_TABLE_STATS (
    ownname           VARCHAR,
    tabname           VARCHAR,
    partname          VARCHAR DEFAULT NULL,
    cascade_parts     BOOLEAN DEFAULT TRUE,
    cascade_columns   BOOLEAN DEFAULT TRUE,
    cascade_indexes   BOOLEAN DEFAULT TRUE,
    force             BOOLEAN DEFAULT FALSE
);
```

此存储过程用于手动删除表的统计信息，主要在清理不准确的统计信息场景中使用。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表/AC名 |
| partname | 分区名 |
| cascade_parts | 是否删除与表相关联的分区的统计信息 |
| cascade_columns | 是否删除与表相关联的列的统计信息 |
| cascade_indexes | 是否删除与表相关联的索引的统计信息 |
| force | 是否强制删除被锁定的统计信息 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.DELETE_TABLE_STATS('SALES', 'SALES_INFO_RANGE', 'P_SALES_INFO_RANGE_1', true, true, true, false);
END;
/
```

### DELETE_INDEX_STATS

```
DBMS_STATS.DELETE_INDEX_STATS (
    ownname           VARCHAR,
    indname           VARCHAR,
    partname          VARCHAR DEFAULT NULL,
    cascade_parts     BOOLEAN DEFAULT TRUE,
    force             BOOLEAN DEFAULT FALSE
);
```

此存储过程用于手动删除索引的统计信息，主要在清理不准确的统计信息场景中使用。

此存储过程不适用于组合分区表的索引信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| indname | 索引名 |
| partname | 索引分区名 |
| cascade_parts | 是否删除与索引相关联的分区的统计信息 |
| force | 是否强制删除被锁定的统计信息 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.DELETE_INDEX_STATS('SALES', 'IDX_FINANCE_INFO_1', null, true, false);
END;
/
```

## DELETE_SCHEMA_STATS

```
DBMS_STATS.DELETE_SCHEMA_STATS (
    ownname           VARCHAR,
    force             BOOLEAN DEFAULT FALSE
);
```

此存储过程用于手动删除某个schema下所有对象（表、AC、列、索引）的统计信息，主要在清理不准确的统计信息场景中使用。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| force | 是否强制删除被锁定的统计信息 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.DELETE_SCHEMA_STATS('SALES', false);
END;
/
```

## DELETE_COLUMN_STATS

```
DBMS_STATS.DELETE_COLUMN_STATS (
    ownname         VARCHAR,
    tabname         VARCHAR,
    colname         VARCHAR,
    partname        VARCHAR,
    type            VARCHAR,
    cascade_parts   BOOLEAN DEFAULT TRUE,
    force           BOOLEAN DEFAULT FALSE
);
```

此存储过程用于手动删除列的统计信息，主要在清理不准确的统计信息场景中使用。

| 参数 | 描述 |
| --- | --- |

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表/AC名 |
| colname | 列名 |
| partname | 分区名 |
| type | 删除类型（'ALL'：同时删除column统计信息和直方图，'HISTOGRAM'：只删除直方图） |
| cascade_parts | 是否删除与列相关联的分区的统计信息 |
| force | 是否强制删除被锁定的统计信息 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.DELETE_COLUMN_STATS('SALES', 'SALES_INFO_RANGE', 'AMOUNT', 'P_SALES_INFO_RANGE_1', 'ALL', true, false);
END;
/
```

## 收集统计信息

### GATHER_TABLE_STATS

```
DBMS_STATS.GATHER_TABLE_STATS (
    ownname             VARCHAR,
    tabname             VARCHAR,
    partname            VARCHAR,
    estimate_percent    NUMBER,
    block_sample        BOOLEAN,
    method_opt          VARCHAR,
    degree              NUMBER,
    granularity         VARCHAR,
    cascade             BOOLEAN
);
```

此存储过程用于收集表/AC或分区的统计信息。

对于组合分区表，不可指定收集某个分区/子分区，仅收集其GLOBAL级统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表/AC名 |
| partname | 分区名 |
| estimate_percent | 采样率，见统计信息选项说明 |
| block_sample | 是否采用块级采样 |
| method_opt | 列统计信息选项，见统计信息选项说明 |
| degree | 并行度，对于大表增大并行度可以提升统计信息收集的效率，见统计信息选项说明 |
| granularity | 分区统计粒度，见统计信息选项说明 |
| cascade | 是否收集索引统计信息，见统计信息选项说明 |

示例

```
BEGIN
    DBMS_STATS.GATHER_TABLE_STATS('SALES', 'SALES_INFO_RANGE', 'P_SALES_INFO_RANGE_1', 0.2, FALSE, 'FOR ALL COLUMNS SIZE
AUTO', 4, 'AUTO', TRUE);
END;
/
```

## GATHER_INDEX_STATS

```
DBMS_STATS.GATHER_INDEX_STATS (
    ownname             VARCHAR,
    indname             VARCHAR,
    partname            VARCHAR,
    estimate_percent    NUMBER,
    degree              NUMBER,
    granularity         VARCHAR,
);
```

此存储过程用于收集索引或索引分区的统计信息。

对于组合分区索引，不可指定收集某个分区/子分区，仅收集其GLOBAL级统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| indname | 索引名 |
| partname | 索引分区名，NULL表示统计所有分区 |
| estimate_percent | 采样率百分比，见统计信息选项说明 |
| degree | 并行度，见统计信息选项说明 |
| granularity | 分区统计粒度，见统计信息选项说明 |

示例（单机、共享集群部署）

```
BEGIN
    DBMS_STATS.GATHER_INDEX_STATS('SALES', 'IDX_FINANCE_INFO_1', '', 0.2, 2,'ALL');
END;
/
```

## GATHER_SCHEMA_STATS

```
DBMS_STATS.GATHER_SCHEMA_STATS (
    ownname           VARCHAR,
    estimate_percent  NUMBER,
    block_sample      BOOLEAN,
    method_opt        VARCHAR,
    degree            NUMBER,
    granularity       VARCHAR,
    cascade           BOOLEAN
);
```

此存储过程用于收集指定用户下所有对象（表、AC、列、索引）的统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |

| 参数 | 描述 |
| --- | --- |
| estimate_percent | 采样率百分比，见统计信息选项说明 |
| block_sample | 是否采用块级采样 |
| method_opt | 列统计信息选项，见统计信息选项说明 |
| degree | 并行度，见统计信息选项说明 |
| granularity | 分区统计粒度，见统计信息选项说明 |
| cascade | 是否收集索引统计信息，见统计信息选项说明 |

```
exec DBMS_STATS.GATHER_SCHEMA_STATS('SALES', 1, TRUE, 'FOR ALL COLUMNS SIZE AUTO', 1, 'ALL', TRUE);
```

## GATHER_DATABASE_STATS

```
DBMS_STATS.GATHER_DATABASE_STATS (
    options             VARCHAR,
    estimate_percent    NUMBER,
    degree              NUMBER,
    method_opt          VARCHAR,
    granularity         VARCHAR,
    cascade             BOOLEAN,
    gather_sys          BOOLEAN
);
```

此存储过程用于收集数据库的统计信息。

| 参数 | 描述 |
| --- | --- |
| options | 见统计信息选项说明 |
| estimate_percent | 采样率，见统计信息选项说明 |
| degree | 并行度，见统计信息选项说明 |
| method_opt | 列统计信息选项，见统计信息选项说明 |
| granularity | 分区统计粒度，见统计信息选项说明 |
| cascade | 是否收集索引统计信息，见统计信息选项说明 |
| gather_sys | 是否收集系统表的统计信息 |

示例

```
exec DBMS_STATS.GATHER_DATABASE_STATS('GATHER AUTO', 1, 2,  'FOR ALL COLUMNS SIZE AUTO', 'AUTO', TRUE, FALSE);
```

## FLUSH_DATABASE_MONITORING_INFO

```
DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO
```

此存储过程用于持久化表/AC的变化信息。

示例（单机、共享集群部署）

```
exec DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO;
```

## 锁定/解锁统计信息

### LOCK_PARTITION_STATS

```
DBMS_STATS.LOCK_PARTITION_STATS (
    ownname             VARCHAR,
    tabname             VARCHAR,
    partname            VARCHAR
);
```

此存储过程用于锁定表中某个表分区的统计信息，锁定后该分区的统计信息不再被修改。

此存储过程不适用于组合分区表。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |
| partname | 分区名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.LOCK_PARTITION_STATS('SALES', 'SALES_INFO_RANGE', 'P_SALES_INFO_RANGE_1');
```

### LOCK_TABLE_STATS

```
DBMS_STATS.LOCK_TABLE_STATS (
    ownname             VARCHAR,
    tabname             VARCHAR
);
```

此存储过程用于锁定某张表的统计信息，锁定后该表的统计信息不再被修改。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.LOCK_TABLE_STATS('SALES', 'SALES_INFO');
```

### LOCK_SCHEMA_STATS

```
DBMS_STATS.LOCK_SCHEMA_STATS (
    ownname             VARCHAR
);
```

此存储过程用于锁定某个schema下所有表的统计信息，锁定后该schema下所有表的统计信息不再被修改，但该schema下在此之后新建的表并不锁定。

| 参数 | 描述 |
| --- | --- |

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.LOCK_SCHEMA_STATS('SALES');
```

## UNLOCK_PARTITION_STATS

```
DBMS_STATS.UNLOCK_PARTITION_STATS (
    ownname             VARCHAR,
    tabname             VARCHAR,
    partname            VARCHAR
);
```

此存储过程用于解锁表中某个表分区的统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |
| partname | 分区名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.UNLOCK_PARTITION_STATS('SALES', 'SALES_INFO', 'P_SALES_INFO_1');
```

## UNLOCK_TABLE_STATS

```
DBMS_STATS.UNLOCK_TABLE_STATS (
    ownname             VARCHAR,
    tabname             VARCHAR
);
```

此存储过程用于解锁某张表的统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.UNLOCK_TABLE_STATS('SALES', 'SALES_INFO');
```

## UNLOCK_SCHEMA_STATS

```
DBMS_STATS.UNLOCK_SCHEMA_STATS (
    ownname             VARCHAR
);
```

此存储过程用于解锁某个schema下所有表的统计信息。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.UNLOCK_SCHEMA_STATS('SALES');
```

## 设置统计信息选项

在调用统计信息收集程序时，YashanDB提供一系列选项，如分区粒度、采样率等，供用户指定，以获取更精准可用的数据，这些选项的值可通过如下三种方式设置：

- 作为收集程序的参数，在调用时指定，此种方式优先级最高。
- 通过SET_*_PREFS程序进行通用设定，当未指定选项参数，或选项参数为NULL时，系统使用此通用设定值。
- 当未按上述两种方式指定值时，由系统给出默认值。

其中，某个选项一旦由SET_*_PREFS程序设定，即可通过ALL_TAB_STAT_PREFS、DBA_TAB_STAT_PREFS、USER_TAB_STAT_PREFS视图查看该选项在表上对应的值。

下表列示这些选项：

| 选项 | 含义 | 系统默认值 |
| --- | --- | --- |
| STALE_PERCENT | 统计信息失效的变化率阈值 | 0.1 |
| CASCADE | 收集表的统计信息时，是否一并收集索引的统计信息 | FALSE |
| DEGREE | 统计信息收集的并行度 | 1 |
| ESTIMATE_PERCENT | 采样率，取值为0或者在[0.0001-1]之内。当取值为0时，将由数据库自动决定采样率。 | 1 |
| GRANULARITY | 分区表收集策略<br>1. ALL：收集表和分区<br>2. GLOBAL：只收集表<br>3. GLOBAL AND PARTITION：收集表和分区<br>4. PARTITION：只收集分区<br>5. SUBPARTITION：保留选项。只收集子分区<br>6. AUTO：由分区的类型自动决定 | GLOBAL |
| METHOD_OPT | 统计信息收集选项<br>1. FOR ALL [INDEXED] COLUMNS [size_clause]：收集所有列或索引列的统计信息<br>2. FOR COLUMNS [column_clause] [size_clause]：收集指定列的统计信息<br>column_clause：(col1,col2,col3)<br>size_clause：SIZE {integer \| AUTO}用于指定直方图信息<br>1. integer：直方图的bucket数量，范围为[1, 2048]<br>2. AUTO：由数据库决定是否生成直方图<br>SET_SCHEMA_PREFS、SET_DATABASE_PREFS、SET_GLOBAL_PREFS只能设置FOR ALL [INDEXED] COLUMNS [size_clause]。 | FOR ALL COLUMNS SIZE AUTO |
| OPTIONS | 数据库统计信息收集选项<br>1. GATHER：收集所有表的统计信息<br>2. GATHER AUTO：默认选项，由数据库决定收集的表<br>3. GATHER STALE：只收集无效的统计信息<br>4. GATHER EMPTY：只收集空的统计信息<br>该选项仅能通过SET_GLOBAL_PREFS设置。 | GATHER |

### SET_TABLE_PREFS

```
DBMS_STATS.SET_TABLE_PREFS (
    ownname             VARCHAR,
    tabname             VARCHAR,
    pname               VARCHAR,
    pvalue              VARCHAR
);
```

此存储过程用于设置表的统计信息选项，方便用户控制表的统计信息收集、失效等行为。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |
| pname | 统计信息选项名 |
| pvalue | 统计信息选项值 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.SET_TABLE_PREFS('SALES', 'SALES_INFO', 'DEGREE', '4');
```

## SET_SCHEMA_PREFS

```
DBMS_STATS.SET_SCHEMA_PREFS (
    ownname             VARCHAR,
    pname               VARCHAR,
    pvalue              VARCHAR
);
```

此存储过程用于设置schema下当前所有表的统计信息选项，对后续新建的表不生效。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| pname | 统计信息选项名 |
| pvalue | 统计信息选项值 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.SET_SCHEMA_PREFS('SALES', 'DEGREE', '4');
```

## SET_DATABASE_PREFS

```
DBMS_STATS.SET_DATABASE_PREFS (
    pname               VARCHAR,
    pvalue              VARCHAR,
    add_sys             BOOLEAN
);
```

此存储过程用于设置数据库下当前所有表的统计信息选项，对后续新建的表不生效。

| 参数 | 描述 |
| --- | --- |
| pname | 统计信息选项名 |

| 参数 | 描述 |
| --- | --- |
| pvalue | 统计信息选项值 |
| add_sys | 是否设置SYS用户下表的统计信息选项，默认为FALSE |

示例（单机、共享集群部署）

```
exec DBMS_STATS.SET_DATABASE_PREFS('DEGREE', '4', FALSE);
```

### SET_GLOBAL_PREFS

```
DBMS_STATS.SET_GLOBAL_PREFS (
    pname               VARCHAR,
    pvalue              VARCHAR
);
```

此存储过程用于设置全局的统计信息选项，对现有没有设置该选项的表和后续新建的表生效。

| 参数 | 描述 |
| --- | --- |
| pname | 统计信息选项名 |
| pvalue | 统计信息选项值 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.SET_GLOBAL_PREFS('DEGREE', '4');
```

## 删除统计信息选项

### DELETE_TABLE_PREFS

```
DBMS_STATS.DELETE_TABLE_PREFS (
    ownname             VARCHAR,
    tabname             VARCHAR,
    pname               VARCHAR
);
```

此存储过程用于删除表的统计信息选项，方便用户控制表的统计信息收集、失效等行为。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| tabname | 表名 |
| pname | 统计信息选项名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.DELETE_TABLE_PREFS('SALES', 'SALES_INFO_RANGE', 'DEGREE');
```

### DELETE_SCHEMA_PREFS

```
DBMS_STATS.DELETE_SCHEMA_PREFS (
    ownname             VARCHAR,
    pname               VARCHAR
);
```

此存储过程用于删除某个schema下当前所有表的统计信息选项，方便用户控制表的统计信息收集、失效等行为。

| 参数 | 描述 |
| --- | --- |
| ownname | 用户名 |
| pname | 统计信息选项名 |

示例（单机、共享集群部署）

```
exec DBMS_STATS.DELETE_SCHEMA_PREFS('SALES', 'DEGREE');
```

## 获取统计信息选项的值

### GET_PREFS

```
DBMS_STATS.GET_PREFS (
    pname               VARCHAR,
    ownname             VARCHAR DEFAULT NULL,
    tabname             VARCHAR DEFAULT NULL)
RETURN VARCHAR;
```

此存储过程用于获取表的统计信息选项的值。

| 参数 | 描述 |
| --- | --- |
| pname | 统计信息选项名 |
| ownname | 用户名 |
| tabname | 表名 |

示例（单机、共享集群部署）

```
SELECT DBMS_STATS.GET_PREFS('DEGREE', 'SALES', 'SALES_INFO_RANGE') FROM DUAL;
```

# DBMS_UTILITY

DBMS_UTILITY包提供了一组内置的存储过程/函数，提供了各种实用程序。

## FORMAT_CALL_STACK

```
DBMS_UTILITY.FORMAT_CALL_STACK()
  RETURN VARCHAR;
```

FORMAT_CALL_STACK返回当前过程体调用栈的格式化输出，无参数，'()'可以省略，返回值最大2000个字节。

示例

```
CREATE OR REPLACE PROCEDURE PROC_CALL AS
BEGIN
DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_CALL_STACK);
END;
/

CALL PROC_CALL;

--result
----- PL/SQL Call Stack ----
object handle     line number  object  name
0x7fffe2e39010              3  procedure  REGRESS.PROC_CALL
0x7fffdfb3bd90              1  anonymous block
```

## FORMAT_ERROR_STACK

```
DBMS_UTILITY.FORMAT_ERROR_STACK
  RETURN VARCHAR;
```

FORMAT_ERROR_STACK返回当前错误栈的格式化输出，无参数，'()'可以省略，返回值最大2000个字节。

示例

```
DECLARE
N INT;
BEGIN
N := 10 / 0;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_ERROR_STACK);
END;
/

--result
YAS-00011 divided by zero
```

## EXEC_DDL_STATEMENT

```
DBMS_UTILITY.EXEC_DDL_STATEMENT (
  parse_string IN VARCHAR2
  );
```

EXEC_DDL_STATEMENT用于执行DDL语句，其中parse_string为待执行的语句。

示例

```
DECLARE
BEGIN
    DBMS_UTILITY.EXEC_DDL_STATEMENT('CREATE TABLE UTILITY_TABLE(A INT, B INT)');
END;
/
SELECT * FROM UTILITY_TABLE;

--result
           A            B
------------ ------------
```

# OWA_UTIL

OWA_UTIL包提供了一组内置子程序，用于执行返回过程体调用信息等操作。

## WHO_CALLED_ME

示例

```
OWA_UTIL.WHO_CALLED_ME(
   owner          OUT      VARCHAR2,
   name           OUT      VARCHAR2,
   lineno         OUT      NUMBER,
   caller_t       OUT      VARCHAR2);
```

WHO_CALLED_ME程序通过出参的方式返回调用它的PL过程体单元信息。

| 参数 | 描述 |
| --- | --- |
| owner | 过程体单元的所有者。 |
| name | 过程体单元的名字。 |
| lineno | 进行调用的过程体单元内的调用位置行号。 |
| caller_t | 进行调用的过程体单元的类型。 |

示例

```
CREATE OR replace PROCEDURE proc1 AS
owner    VARCHAR(200);
name     varchar(200);
lineno   number;
caller_t varchar(200);
BEGIN
OWA_UTIL.WHO_CALLED_ME(owner, name, lineno, caller_t);
DBMS_OUTPUT.PUT_LINE('owener: ' || owner || ' name: ' || name || ' lineno: ' || lineno || ' caller_t: '|| caller_t);
END;
/

BEGIN
proc1();
END;
/
owener:  name:  lineno: 2 caller_t: ANONYMOUS BLOCK
```

# UTL_ENCODE

UTL_ENCODE包提供将RAW类型数据编码为标准编码格式的函数以及对应的解码函数，标准编码格式更便于数据在服务器之间传输。上述函数均遵循已发布的编码标准。

## BASE64_ENCODE

```
UTL_ENCODE.BASE64_ENCODE (R IN RAW) RETURN RAW;
```

BASE64_ENCODE函数采用BASE64将输入的RAW类型数据进行编码并输出编码后的数据。

| 参数 | 描述 |
| --- | --- |
| R | 要编码的RAW字符串。 |

示例

```
SELECT UTL_ENCODE.BASE64_ENCODE('0000') FROM DUAL;
```

## BASE64_DECODE

```
UTL_ENCODE.BASE64_DECODE (R IN RAW) RETURN RAW;
```

BASE64_DECODE函数用于读取BASE64编码的原始输入字符串，并将其解码为原始值。

| 参数 | 描述 |
| --- | --- |
| R | 包含BASE64编码数据的RAW字符串。 |

示例

```
SELECT UTL_ENCODE.BASE64_DECODE('4141413D') FROM DUAL;
```

# UTL_FILE

UTL_FILE包为用户提供了读写系统文件的能力，用户可以通过UTL_FILE操作系统文件。

## FOPEN

```
UTL_FILE.FOPEN (
   location     IN VARCHAR2,
   filename     IN VARCHAR2,
   open_mode    IN VARCHAR2,
   max_linesize IN BINARY_INTEGER DEFAULT 1024)
  RETURN FILE_TYPE;
```

FOPEN函数用于打开一个文件，可以指定一行的最大长度max_lineSize的范围在1-32000不指定的话默认为1024。在一个会话下默认打开50个文件，最多支持的打开文件数由 SESSION_MAX_OPEN_FILE指定。

| 参数 | 描述 |
| --- | --- |
| location | 文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| filename | 操作的文件名字。 |
| open_mode | 打开的方式，支持如下方式：<br>* a：表示以追加方式打开文件，默认在文件末尾追加，若不存在指定文件会创建新文件。<br>* w：表示以写方式打开文件，此时会创建一个新的文件并覆盖旧文件。<br>* r：表示以只读方式打开文件。 |
| max_linesize | 对文件读写的一行的最大长度。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    HANDLE_A UTL_FILE.FILE_TYPE;
    HANDLE_W UTL_FILE.FILE_TYPE;
BEGIN
    HANDLE_R := UTL_FILE.FOPEN('/data','yashan.txt','r',100);
    HANDLE_A := UTL_FILE.FOPEN('/data','yashan.txt','a');
    HANDLE_W := UTL_FILE.FOPEN('/data','yashan.txt','w',32000);
END;
/
```

## FCOPY

```
UTL_FILE.FCOPY (
   src_location    IN VARCHAR2,
   src_filename    IN VARCHAR2,
   dest_location   IN VARCHAR2,
   dest_filename   IN VARCHAR2,
   start_line      IN BINARY_INTEGER DEFAULT 1,
   end_line        IN BINARY_INTEGER DEFAULT NULL);
```

FCOPY复制源文件到一个新创建的目标文件中。如果不指定start_line和end_line，将默认复制整个源文件。源文件以只读模式打开，目标文件以只写方式打开。将复制源文件的start_line到end_line的内容到目标文件中。

| 参数 | 描述 |
| --- | --- |

| 参数 | 描述 |
| --- | --- |
| src_location | 源文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| src_filename | 源文件名字。 |
| dest_location | 目标文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| dest_filename | 目标文件名字。 |
| start_line | 从源文件的第几行开始复制，默认为1。 |
| end_line | 复制到源文件的第几行，默认为NULL，表示源文件的最后一行。 |

示例

```
BEGIN
    UTL_FILE.FCOPY('/data','yashan.txt','/data','yashan1.txt');
END;
/
```

## Fflush

```
UTL_FILE.FFLUSH (
   file  IN FILE_TYPE);
```

Fflush函数将指定的打开的文件句柄对应的还没有刷盘的数据刷到磁盘上。一般情况下，数据先写到缓冲区再写到磁盘上。

| 参数 | 描述 |
| --- | --- |
| file | FOPEN成功执行后返回的file值。 |

示例

```
DECLARE
    HANDLE_W UTL_FILE.FILE_TYPE;
    BUFFER_W VARCHAR2(200);
    writenNum INT;
BEGIN
      writenNum := 0;
      BUFFER_W := LPAD('*',150 ,'*');
        HANDLE_W := UTL_FILE.FOPEN('/data','yashan.txt','w',200);
    FOR i IN 1 .. 2 LOOP
      UTL_FILE.PUT_LINE(HANDLE_W,BUFFER_W);
    END LOOP;
    UTL_FILE.FFLUSH(HANDLE_W);
END;
/
```

## FGET_ATTR

```
UTL_FILE.FGETATTR(
   location     IN VARCHAR2,
   filename     IN VARCHAR2,
   fexists      OUT BOOLEAN,
   file_length  OUT NUMBER,
   block_size   OUT BINARY_INTEGER);
```

FGET_ATTR函数返回磁盘文件的属性。

| 参数 | 描述 |
| --- | --- |
| location | 源文件所在的位置，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| filename | 文件的名字。 |
| fexists | 一个表示文件是否存在的BOOL值。 |
| file_length | 文件的字节总长度，如果文件不存在，返回NULL。 |
| block_size | 文件系统块的长度，如果文件不存在，返回NULL。 |

示例

```
DECLARE
    HANDLE_R  UTL_FILE.FILE_TYPE;
    BUFFER_W  VARCHAR2(1000);
    fexists   BOOLEAN;
    flen      NUMBER;
    blockSize INT;
BEGIN
    UTL_FILE.FGETATTR('/data','yashan.txt',fexists,flen,blockSize);
    DBMS_OUTPUT.PUT_LINE('yashan.txt:');
    IF fexists THEN
    DBMS_OUTPUT.PUT_LINE(flen);
    DBMS_OUTPUT.PUT_LINE(blockSize);
    END IF;
  END;
/

--result
yashan.txt:
302
0
```

## Frename

```
UTL_FILE.FRENAME (
   src_location     IN   VARCHAR2,
   src_filename     IN   VARCHAR2,
   dest_location    IN   VARCHAR2,
   dest_filename    IN   VARCHAR2,
   overwrite        IN   BOOLEAN DEFAULT FALSE);
```

Frename函数用于重命名一个已经存在的文件。

| 参数 | 描述 |
| --- | --- |
| src_location | 源文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| src_filename | 源文件的名字。 |
| dest_location | 目标文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| dest_filename | 重命名的文件的名字。 |
| overwrite | 如果目标文件已经存在，是否要覆盖。 |

示例

```
BEGIN
    UTL_FILE.FRENAME('/data','yashan1.txt','/data','yashan2.txt');
END;
/
```

## Fremove

```
UTL_FILE.FREMOVE (
   location IN VARCHAR2,
   filename IN VARCHAR2);
```

Fremove函数用于删除磁盘的文件，如果有足够的权限的话可以成功删除。

| 参数 | 描述 |
| --- | --- |
| location | 文件所在的路径，仅支持起始为“.”或“/”的路径。<br>若被赋值起始为“.”的路径，表示相对于环境变量YASDB_DATA的路径。 |
| filename | 文件的名字。 |

示例

```
BEGIN
    UTL_FILE.FREMOVE('/data','yashan2.txt');
END;
/
```

## Fseek

```
UTL_FILE.FSEEK (
   file             IN OUT  UTL_FILE.FILE_TYPE,
   absolute_offset  IN      PLS_INTEGER DEFAULT NULL,
   relative_offset  IN      PLS_INTEGER DEFAULT NULL);
```

Fseek用来将当前的文件指针向前或者向后移动指定的字节数，只能用于以读方式打开的文件句柄。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |
| absolute_offset | 文件指针移动的绝对偏移值，默认为NULL。 |
| relative_offset | 文件指针移动的相对偏移，基于当前的偏移值，如果<0则向前移动n个字节，如果>0则向后移动n个字节。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    BUFFER_R  VARCHAR2(500);
BEGIN
    IF UTL_FILE.IS_OPEN(HANDLE_R) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE HAS OPENED');
    ELSE
        HANDLE_R := UTL_FILE.FOPEN('/data','yashan.txt','r',500);
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE FIRST OPEN');
    END IF;
```

```
    UTL_FILE.FSEEK(HANDLE_R,10,0);
    UTL_FILE.GET_LINE(HANDLE_R,BUFFER_R,200);
    DBMS_OUTPUT.PUT_LINE(BUFFER_R);
END;
/

--result
HANDLE_R FILE FIRST OPEN
********************************************************************************************************************************************
**************
```

## GET_LINE

```
UTL_FILE.GET_LINE (
   file        IN  FILE_TYPE,
   buffer      OUT VARCHAR2,
   len         IN  PLS_INTEGER DEFAULT NULL);
```

GET_LINE函数用于从文件中读取一行，但是读取的一行的长度不能超过FOPEN函数中设置的max_linesize。 另外，GET_LINE函数只支持在以只读模式打开的文件句柄中使用。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |
| buffer | 从文件中读取的一行数据放置的缓冲区。 |
| len | 从文件中读取几个字节，默认为NULL。如果是NULL，yasdb会将该值设置为max_linesize。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    BUFFER_R  VARCHAR2(500);
BEGIN
    IF UTL_FILE.IS_OPEN(HANDLE_R) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE HAS OPENED');
    ELSE
        HANDLE_R := UTL_FILE.FOPEN('/data','yashan.txt','r',500);
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE FIRST OPEN');
    END IF;

    UTL_FILE.FSEEK(HANDLE_R,10,0);
    UTL_FILE.GET_LINE(HANDLE_R,BUFFER_R,200);
    DBMS_OUTPUT.PUT_LINE(BUFFER_R);
END;
/

--result
HANDLE_R FILE FIRST OPEN
********************************************************************************************************************************************
**************
```

## IS_OPEN

```
UTL_FILE.IS_OPEN(
   FILE IN FILE_TYPE
)
```

IS_OPEN用于判断当前fileHandle对应的文件是否已经打开。

| 参数 | 描述 |
| --- | --- |
| FILE_TYPE | 对文件进行操作的句柄。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    BUFFER_R VARCHAR2(200);
BEGIN
    IF UTL_FILE.IS_OPEN(HANDLE_R) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE HAS OPENED');
    ELSE
        HANDLE_R := UTL_FILE.FOPEN('/data','yashan.txt','r',200);
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE FIRST OPEN');
    END IF;
END;
/

--result
HANDLE_R FILE FIRST OPEN
```

## NEW_LINE

```
UTL_FILE.NEW_LINE (
   file     IN FILE_TYPE,
   lines    IN BINARY_INTEGER := 1);
```

NEW_LINE函数用于对文件句柄指定的文件增加一行或者多行换行，这个函数和PUT函数配合的效果相当于PUT_LINE。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |
| lines | 增加几个换行。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    HANDLE_W UTL_FILE.FILE_TYPE;
    BUFFER_R VARCHAR2(200);
    writenNum INT;
BEGIN
    writenNum := 0;
    HANDLE_R := UTL_FILE.FOPEN('/data','yashan.txt','r',200);
    HANDLE_W := UTL_FILE.FOPEN('/data','yashan1.txt','w',300);

    FOR i IN 1 .. 2 LOOP
      UTL_FILE.GET_LINE(HANDLE_R,BUFFER_R,110);
      DBMS_OUTPUT.PUT_LINE(BUFFER_R);

      UTL_FILE.PUT(HANDLE_W,BUFFER_R);
      IF i = 2 THEN
        UTL_FILE.NEW_LINE(HANDLE_W);
        UTL_FILE.FFLUSH(HANDLE_W);
      END IF;
    END LOOP;
    UTL_FILE.FFLUSH(HANDLE_W);
END;
/
```

```
--result
************************************************************************************************************************
**************************************
```

## PUT

```
UTL_FILE.PUT (
   file      IN FILE_TYPE,
   buffer    IN VARCHAR2);
```

PUT函数将buffer中的数写入到file指定的文件中，但是PUT函数中使用的file必须是以写模式打开的。PUT函数不会为数据添加换行符，如果要添加换行符可以使用NEW_LINE或者PUT_LINE函数。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |
| buffer | 需要写入到文件的数据存放的位置。 |

示例

```
DECLARE
    HANDLE_W UTL_FILE.FILE_TYPE;
    BUFFER_W VARCHAR2(200);
    writenNum INT;
BEGIN
    writenNum := 0;
    BUFFER_W := LPAD('*', 20, '*');
    IF UTL_FILE.IS_OPEN(HANDLE_W) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_W FILE HAS OPENED');
    ELSE
        HANDLE_W := UTL_FILE.FOPEN('/data','yashan1.txt','w',130);
        DBMS_OUTPUT.PUT_LINE('HANDLE_W FILE FIRST OPEN');
    END IF;
    FOR i IN 1 .. 2 LOOP
      UTL_FILE.PUT(HANDLE_W,BUFFER_W);
      IF i = 2 THEN
        UTL_FILE.NEW_LINE(HANDLE_W);
        UTL_FILE.FFLUSH(HANDLE_W);
      END IF;
    END LOOP;

    UTL_FILE.FFLUSH(HANDLE_W);
END;
/

--result
HANDLE_W FILE FIRST OPEN
```

## PUT_LINE

```
UTL_FILE.PUT_LINE (
   file      IN FILE_TYPE,
   buffer    IN VARCHAR2,
   autoflush IN BOOLEAN DEFAULT FALSE);
```

PUT函数将buffer中的数写入到file指定的文件中。文件必须以写打开，PUT_LINE会在数据后面自动添加换行符。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |

| 参数 | 描述 |
| --- | --- |
| buffer | 需要写入到文件的缓冲buffer。 |
| autoflush | bool值，用于表示在写之后是否立即刷到磁盘，默认是false。 |

示例

```
DECLARE
    HANDLE_W UTL_FILE.FILE_TYPE;
    BUFFER_W VARCHAR2(200);
    writenNum INT;
BEGIN
    writenNum := 0;
    BUFFER_W := LPAD('*', 20, '*');
    IF UTL_FILE.IS_OPEN(HANDLE_W) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_W FILE HAS OPENED');
    ELSE
        HANDLE_W := UTL_FILE.FOPEN('/data','yashan1.txt','w',130);
        DBMS_OUTPUT.PUT_LINE('HANDLE_W FILE HAS OPENED');
    END IF;
    FOR i IN 1 .. 2 LOOP
      UTL_FILE.PUT_LINE(HANDLE_W,BUFFER_W,false);
    END LOOP;
    UTL_FILE.FFLUSH(HANDLE_W);
END;
/

--result
HANDLE_W FILE HAS OPENED
```

## FGETPOS

```
UTL_FILE.FGETPOS (
   file      IN FILE_TYPE);
```

FGETPOS函数返回文件中当前打开文件的相对偏移位置（以字节为单位）。

| 参数 | 描述 |
| --- | --- |
| file | 文件句柄。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    BUFFER_R VARCHAR2(500);
    POS      INTEGER;
BEGIN
    IF UTL_FILE.IS_OPEN(HANDLE_R) THEN
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE HAS OPENED');
    ELSE
        HANDLE_R := UTL_FILE.FOPEN('/data','yashan1.txt','r',500);
        DBMS_OUTPUT.PUT_LINE('HANDLE_R FILE FIRST OPEN');
    END IF;

    UTL_FILE.GET_LINE (HANDLE_R, BUFFER_R);
    POS := UTL_FILE.FGETPOS(HANDLE_R);
    DBMS_OUTPUT.PUT_LINE('BEFORE_SEEK_POS:'||POS);

    UTL_FILE.FSEEK(HANDLE_R, NULL, -10);
```

```
    POS := UTL_FILE.FGETPOS(HANDLE_R);
    DBMS_OUTPUT.PUT_LINE('AFTER_SEEK_POS:'||POS);
    UTL_FILE.FCLOSE (HANDLE_R);
END;
/

--result
HANDLE_R FILE FIRST OPEN
BEFORE_SEEK_POS:21
AFTER_SEEK_POS:11
```

## FCLOSE

```
UTL_FILE.FCLOSE (
   file IN OUT FILE_TYPE);
```

FCLOSE函数用于关闭通过FILE_TYPE指定的文件。

| 参数 | 描述 |
| --- | --- |
| file | FOPEN成功执行后返回的file值。 |

示例

```
DECLARE
    HANDLE_R UTL_FILE.FILE_TYPE;
    BUFFER_R VARCHAR2(200);
BEGIN
        HANDLE_R := UTL_FILE.FOPEN('/data','yashan1.txt','r',200);
        UTL_FILE.FCLOSE(HANDLE_R);
END;
/
```

## FCLOSE_ALL

```
UTL_FILE.FCLOSEALL (
  );
```

FCLOSE_ALL函数用于关闭当前session下所有打开的file handle，多数用于紧急情况下的文件句柄清理。

示例

```
DECLARE
   BEGIN
   UTL_FILE.FCLOSE_ALL();
   END;
   /
```

## 异常说明

| 参数 | 描述 |
| --- | --- |
| INVALID_PATH | 文件所在的路径不存在。 |
| INVALID_MODE | 文件打开模式不对。 |
| INVALID_FILETYPE | 文件句柄无效。 |
| INVALID_OPERATION | 操作错误。 |

| 参数 | 描述 |
| --- | --- |
| READ_ERROR | 在读操作时发生了错误。 |
| WRITE_ERROR | 在写操作时发生了错误。 |
| INVALID_MAXLINESIZE | 无效的max_linesize值，正常范围在1-32000。 |
| INVALID_FILENAME | 无效的文件名。 |
| ACCESS_DENIED | 文件拒绝被访问。 |
| INVALID_OFFSET | 无效的偏移地址，当absolute_offset和relative_offset同时为NULL，或absolute_offset<0，或没有一个可以有效地指定需要跳转到文件的哪个位置。 |
| DELETE_FAILED | 删除失败。 |
| RENAME_FAILED | 重命名失败。 |

示例

```
DECLARE
       HANDLE_R UTL_FILE.FILE_TYPE;
        fexists   BOOLEAN;
    flen      NUMBER;
    blockSize INT;
    BUFFER_R VARCHAR(10);
    BEGIN
        UTL_FILE.FREMOVE('./a','t2.txtrttt');
       DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
      EXCEPTION WHEN UTL_FILE.INVALID_PATH THEN
       DBMS_OUTPUT.PUT_LINE('test invalid path success!');
       DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
    WHEN UTL_FILE.INVALID_FILENAME   THEN
       DBMS_OUTPUT.PUT_LINE('test invalid fileName success!');
   END;
   /

--result
test invalid path success!
330:YAS-00330 invalid path
```

# 参数

在编程语言中，参数用于从外部向一个程序块传递值或变量，YashanDB PL中的参数被应用于如下程序块：

- 存储过程/函数，包括独立的存储过程/函数，或被定义在高级包、UDT等PL对象里的存储过程/函数。

创建或重建存储过程/函数时，参数中有错误依然会创建或重建该对象，但在后续调用中会返回错误。

存储过程/函数不一定需要参数，使用参数则可以在调用存储过程/函数时，对其运行内容进行控制以达到特定的目的。

此类情况使用参数需要在定义存储过程/函数时进行参数声明（形参），并在调用存储过程/函数时进行值或变量传递（实参）。

- 在EXECUTE Statement或OPEN Statement中运行的动态SQL。

动态SQL是在运行时才进行构造的SQL语句，使用参数可以让构造的SQL语句模板化，简化PL编程，且可以让优化器按SQL语句模板（结合变量窥视）生成执行计划，避免对每次运行硬解析，极大地提高SQL执行性能。

此类情况使用参数需要在SQL语句中使用占位符进行参数绑定（相当于存储过程/函数中的形参），并通过USING语法进行值或变量传递（相当于存储过程/函数中的实参）。

# 形参和实参

在PL里，参数可区分为形参和实参两个概念。

形参：在定义PL对象时指定的参数，位于头部结构里PL对象名后面的括号中。

实参：在调用PL对象时指定的参数，位于调用时指定的PL对象名后面的括号中。

PL执行器将把实参与形参按照如下三种方式之一进行一一对应：

- 实参直接输入值或变量，与形参按位置对应。
- 实参使用"=>"指定形参的名称进行对应。
- 前两种方式混合，但是在有多个形参时，如果有一个形参对应的实参是按名称指定，则其后的形参对应的实参都要使用按名称方式指定。

如果实参的数据类型与形参不一致，系统会对其进行到形参数据类型的转换，转换规则参考CAST函数描述，如转换失败将抛出错误。

> **Note**：
> 与CAST中将 `CHAR` 默认为 `CHAR(1)` 不同，在PL中的参数转换无此规则。

示例

```
--此处的argu称为形参
CREATE OR REPLACE PROCEDURE ya_proc(argu1 FLOAT, argu2 VARCHAR DEFAULT 'shenzhen') AS
BEGIN
DBMS_OUTPUT.PUT_LINE(TO_CHAR(argu1)||'------'||argu2);
END;
/

--此处的argu称为实参，该值将被赋给形参
SET NUMWIDTH 20;
exec ya_proc('4');

--result
4.0E+000------shenzhen

--无法将实参转换到对应的形参数据类型时，抛出错误
exec ya_proc('shenzhen');

--result

YAS-00008 type convert error : not a valid number
```

## 形参的格式

形参的格式为：*参数名称 [参数类型] [NOCOPY] 数据类型 [缺省值]*。

其中，数据类型里不应该包含长度、精度等属性。

## 形参的类型

包括IN、OUT和IN OUT三种类型。在定义PL对象时，可以为每一个形参显式地指定一个类型，或者不指定而使用默认的IN类型。

### IN参数

IN参数用于向过程体传递一个值，过程体不能修改该参数的值。此时对应的实参可以为常量或变量。

### OUT参数

OUT参数用于向调用器返回一个值。此时对应的实参必须为变量，且实参的值不会被赋给形参，而是过程体执行完成后，将形参值赋给实参用于返回。

### IN OUT参数

IN OUT参数向过程体传递一个初始值，并向调用器返回一个更新后的值。此时对应的实参必须为变量，且实参的值赋给形参，在过程体内部可读可写；过程体执行结束后，将形参值赋给实参用于返回。

### NOCOPY

对于OUT和IN OUT参数类型，可同时指定NOCOPY关键字，该关键字用于语法兼容，无实际含义。

## 形参的缺省值

只能为IN参数指定缺省值，且缺省值需符合该形参的数据类型要求。

对于指定了缺省值的形参，其对应实参可省略，前提是所有指定了缺省值的形参应该定义在括号最后面位置，且应该同时都指定或者都省略，否则不能保证实参对应到正确的形参。但在实参使用"=>"传入参数值时，无需此前提要求。

示例

```
--创建含三种类型形参的存储过程
CREATE OR REPLACE PROCEDURE ya_proc(
argu_a INT,
argu_b IN INT,
argu_c OUT INT,
argu_d IN OUT BINARY_FLOAT
) IS
BEGIN
DBMS_OUTPUT.PUT_LINE('Inside procedure ya_proc:');
DBMS_OUTPUT.PUT('IN argu_a = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(argu_a), 'NULL'));
DBMS_OUTPUT.PUT('IN argu_b = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(argu_b), 'NULL'));
DBMS_OUTPUT.PUT('OUT argu_c = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(argu_c), 'NULL'));
DBMS_OUTPUT.PUT_LINE('IN OUT argu_d = ' || TO_CHAR(argu_d));
argu_c := argu_a+10;
argu_d := 10/argu_b;
END;
/

--创建匿名块调用ya_proc,展示三种类型形参在执行存储过程前、中、后时的值
DECLARE
aa INT := 1;
bb INT := 2;
cc INT := 3;
dd BINARY_FLOAT := 4;
ee INT;
ff BINARY_FLOAT := 5;
BEGIN
DBMS_OUTPUT.PUT_LINE('Before invoking procedure ya_proc:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
ya_proc (aa,
bb,
cc,
dd
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure ya_proc:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

DBMS_OUTPUT.PUT_LINE('Before invoking procedure ya_proc:');
```

```
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
ya_proc (1,
(bb+3)*4,
ee,
ff
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure ya_proc:');
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
END;
/

--result
Before invoking procedure ya_proc:
aa = 1
bb = 2
cc = 3
dd = 4.0E+000
Inside procedure ya_proc:
IN argu_a = 1
IN argu_b = 2
OUT argu_c = NULL
IN OUT argu_d = 4.0E+000
After invoking procedure ya_proc:
aa = 1
bb = 2
cc = 11
dd = 5.0E+000
Before invoking procedure ya_proc:
ee = NULL
ff = 5.0E+000
Inside procedure ya_proc:
IN argu_a = 1
IN argu_b = 20
OUT argu_c = NULL
IN OUT argu_d = 5.0E+000
After invoking procedure ya_proc:
ee = 11
ff = 5.0E-001
```

# 绑定参数

如本章首页所述，绑定参数被使用在动态SQL中，其中占位符相当于存储过程/函数中的形参，通过USING语法指定的值或变量相当于存储过程/函数中的实参，区别在于：

1. 占位符表示的形参不能定义参数类型、缺省值等信息，形参的参数类型依据实参传入值或变量确定。
2. 实参只能直接输入值或变量，与形参按位置一一对应，与占位符名称无任何关联。
3. 与存储过程/函数在声明形参时指定IN/OUT/IN OUT类型不同，绑定参数在实参传递变量时才进行指定，且只有动态SQL语句存在返回值时，才可以指定为OUT/IN OUT类型，否则默认为IN类型。
4. 基于''和NULL无数据类型，实参的值不能为''和NULL。

## 占位符

占位符由`:`+标识符构成，标识符建议使用数字或字母。

同一SQL语句在实参传递时按照占位符的位置进行参数对应，如果SQL语句绑定不同值时，应该使用不同的占位符名称，否则将编译错误。

示例

```
--定义一个存储过程，其中a2参数在过程体被赋值输出
CREATE OR REPLACE PROCEDURE ya_block(a1 VARCHAR,a2 IN OUT VARCHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no in (:a,:b) and rownum=1';
BEGIN
--str1中不存在返回值，不能将绑定变量a1,a2指定OUT/IN OUT类型进行参数传递
EXECUTE IMMEDIATE str1 INTO no,name USING a1,a2;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
a2 := '00';
END;
/

DECLARE
a1 VARCHAR(20):='01';
a2 VARCHAR(20):='02';
str2 VARCHAR(200) := 'INSERT INTO area(area_no) VALUES(:a) RETURN TO_NUMBER(area_no), NVL(area_name, ''area''||area_no) INTO
:b, :c';
BEGIN
DBMS_OUTPUT.PUT_LINE('before-a1:'||a1||',a2:'||a2);
--ya_block的a2参数存在返回值，可以被定义为OUT/IN OUT类型
EXECUTE IMMEDIATE 'begin ya_block(:b,:c); end;' USING a1,IN OUT a2;
DBMS_OUTPUT.PUT_LINE('plsql.after-a1:'||a1||',a2:'||a2);

--str2中定义了RETURN值，可以将绑定变量a1,a2指定OUT/IN OUT类型进行参数传递
EXECUTE IMMEDIATE str2 USING '09',out a1,out a2;
DBMS_OUTPUT.PUT_LINE('sql.after-a1:'||a1||',a2:'||a2);
END;
/

--result
before-a1:01,a2:02
区域号码:01区域名称:华东
plsql.after-a1:01,a2:00
sql.after-a1:9,a2:area09
```

## 变量窥视

当使用绑定参数时，YashanDB优化器可以将相似的SQL语句合并成一个执行计划，避免多次的硬解析。

变量窥视是在此功能上的进一步优化，实现了：

1. 依据实参的数据类型推导SQL语句中占位符的数据类型。
2. 依据实参传递的绑定变量进行动态窥视，生成最合适的执行计划，而不是只采用依据某一个绑定变量所生成的执行计划。

YashanDB的变量窥视功能不为用户感知，在动态SELECT语句中使用绑定参数时自动调用，提升系统运行效率。

23.1版本开始支持变量窥视之后，一部分绑定变量的场景进行类型推导时会出现与23.1之前的版本不同的现象。

示例

```
CREATE TABLE t1 (id INT, data CHAR(255));
INSERT INTO t1 VALUES(1, 'test');
COMMIT;
SET serveroutput ON;

DECLARE
   var VARCHAR(100) := 'test';
   count INT;
BEGIN
   SELECT COUNT(*) INTO count FROM t1 WHERE data = var;
   DBMS_OUTPUT.PUT_LINE(count);
END;
/
```

这个匿名块用例第5行是使用绑定变量的方式将var变量传入SQL语句中执行。

在23.1之前的版本，没有变量窥视，会将待绑定的unknow变量从最近的运算数据类型中推导一个类型。如该例子里，会将var传入之前的unknow数据类型推导为与data相同的类型，即char类型。待var变量传入以后，会将var变量转为char类型再与data比较。此时使用的是char类型与char类型的比较规则，空格是不敏感的，具体可查看字符型，所以用例where条件匹配成功，结果为1。

在23.1及之后的版本，有变量窥视，会将var数据类型的真实数据类型传入，data与var类型比较时，是char类型与varchar类型比较，此时空格是敏感的，具体可查看字符型。所以用例where条件匹配失败，结果为0。

# PL语句

YashanDB定义了一系列PL语句，按照这些语句指定的格式进行编程，并由YashanDB内置编译器进行编译，生成可运行的PL对象，如存储过程、函数等，满足在数据库层面实现各类复杂的业务逻辑。

## PL中的表达式

PL中的表达式是一个或多个值、运算符和SQL函数的组合，其计算结果为一个值，具体见表达式。

## PL中的静态SQL语句

在PL中可编写如下SQL语句：

- DML：INSERT/UPDATE/SELECT/MERGE/DELETE等。
- DCL：COMMIT/ROLLBACK/SAVEPOINT等。

这些SQL语句与非过程体中的普通SQL语句语法一致，但存在如下约束：

- 在SQL语句中出现的标识符与过程体的变量同名时，将优先使用过程体的变量名称，系统将该标识符替换为绑定参数占位符`:Bi`。
- 编写SELECT语句必须有INTO子句，且select_list的个数必须INTO子句变量或RECORD成员相对应。
- COMMIT/ROLLBACK只能使用缺省的COMMIT/ROLLBACK语法，不能为其指定任何选项。

## PL中的动态SQL语句

在PL中可以通过EXECUTE Statement执行动态SQL语句。

这些SQL语句与非过程体中的普通SQL语句语法一致，但存在如下约束：

- 动态SQL语句中每个绑定变量的占位符，都需要有对应的变量在USING子句。
- 对于动态SQL中的SELECT语句，如查询结果为一行数据，SQL中必须要有INTO子句；如查询结果为多行数据，SQL中必须要有BULK COLLECTION INTO子句。
- 对于动态SQL中的匿名块，除存储过程、函数、游标等需要输入的参数外，匿名块的其他地方不能出现占位符（绑定参数）。

## PL中的控制语句

- 循环控制语句：FOR循环语句、FORALL批量循环语句、WHILT循环语句和LOOP循环语句。
- 顺序控制语句：GOTO语句、CONTINUE语句和EXIT语句。
- 条件选择控制语句：IF语句和CASE语句。
- 过程体结束控制语句: RETURN语句。

# CASE Statement

CASE Statement为条件选择控制语句，包含如下两种形式：

- CASE selector WHEN
- CASE WHEN condition

## CASE selector WHEN

格式为：

*CASE selector*
*WHEN selector_value_1 THEN statements_1*
*WHEN selector_value_2 THEN statements_2*
*...*
*WHEN selector_value_n THEN statements_n*
*_[* ELSE *_else_statements ]*
*END CASE;*

其中，selector、selector_value_1 、selector_value_2、...、selector_value_n为表达式形式。

含义为：

首先计算CASE语句里的selector的值，如果值为NULL，则跳过所有WHEN语句向下寻找ELSE语句，如能找到ELSE语句，则执行else_statements ，否则报错。

如果selector的值不为NULL，则将此值与向下第一个WHEN语句里的selector_value_1的值进行比较：

- 如果selector的值与selector_value_1的值相等，则执行第一个WHEN语句里THEN之后的statements_1，然后跳转到END CASE结束；
- 如果selector的值与selector_value_1的值不相等，则跳转到第二个WHEN语句，执行与第一个WHEN语句相同的操作；
- ......
- 如果selector的值与selector_value_n的值相等，则执行第n个WHEN语句里THEN之后的statements_n，然后跳转到END CASE结束；
- 如果selector的值与selector_value_n的值不相等，则跳转到下一条语句：

如果下一条语句为ELSE语句，则执行else_statements，然后跳转到END CASE结束。

如果下一条语句不为ELSE语句，则抛出错误。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc() IS
i INT:=0;
BEGIN
    CASE i
    WHEN 1 THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    WHEN 2 THEN
        DBMS_OUTPUT.PUT_LINE('world');
    END CASE;
END ya_proc;
/

exec ya_proc;

--result
YAS-05210 case not found

-- 包含ELSE语句
CREATE OR REPLACE PROCEDURE ya_proc(a INT) IS
BEGIN
    CASE a
    WHEN 2 THEN
        DBMS_OUTPUT.PUT_LINE('loop2');
    WHEN 3 THEN
```

```
        DBMS_OUTPUT.PUT_LINE('loop3');
    ELSE
        DBMS_OUTPUT.PUT_LINE('loop');
    END CASE;
END;
/

exec ya_proc(3);

--result
loop3
```

## CASE WHEN condition

格式为：

*CASE*
*WHEN condition_1 THEN statements_1*
*WHEN condition_2 THEN statements_2*
*...*
*WHEN condition_n THEN statements_n*
*_[ ELSE _else_statements ]*
*END CASE;*

其中，condition_1 、condition_2、...、condition_n为能产生布尔值结果的条件表达式。

含义为：

首先判断第一个WHEN语句里的condition_1条件：

- 如果condition_1的值为true，执行第一个WHEN语句里THEN之后的statemnets_1，然后跳转到END CASE结束；
- 如果condition_1的值为false，跳转到第二个WHEN语句，执行与第一个WHEN语句相同的操作；
- ......
- 如果condition_n的值为true，执行第n个WHEN语句里THEN之后的statemnets_n，然后跳转到END CASE结束；
- 如果condition_n的值为false，则跳转到下一条语句。

如果下一条语句为ELSE语句，则执行else_statements，然后跳转到END CASE结束。

如果下一条语句不为ELSE语句，则抛出错误。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc() IS
BEGIN
    CASE WHEN true THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    WHEN false THEN
        DBMS_OUTPUT.PUT_LINE('world');
    END CASE;
END;
/

exec ya_proc;

--result
hello
```

# CLOSE Statement

CLOSE Statement为关闭游标语句，系统对显式游标和动态游标的关闭方法一致，且格式均为：

*CLOSE cursor_name;*

其中，cursor_name为显式游标或动态游标的名称。

关闭游标表示关闭游标指向的结果集。如果游标并非打开状态，或者重复关闭，系统将返回invalid cursor错误。

示例

```
DECLARE
  CURSOR cur IS SELECT 1 FROM dual;
BEGIN
  OPEN cur;
  CLOSE cur;
  IF cur%isopen THEN
    CLOSE cur;
  END IF;
END;
/
```

# CONTINUE Statement

CONTINUE Statement为顺序控制语句，其语法形式为：

*CONTINUE [label_name] [WHEN condition]；*

其中，label_name为可选项，标识需跳过迭代的当前或外层循环；WHEN condition为可选项，condition为能产生布尔结果的表达式。

CONTINUE语句用于跳过循环的当前迭代，跳转到循环开始处，进行下一次循环的执行。需注意：

- label_name标识需跳过迭代的当前或外层循环，若label_name对应的标签不标识当前或外层循环，或为不存在等非法情况，将报错处理。label_name可省略，则CONTINUE语句执行时跳过当前循环的当前迭代。
- WHEN condition子句中的条件结果用于确定是否执行CONTINUE语句，true为执行，false为不执行。当省略WHEN condition子句时，默认执行CONTINUE语句。

示例

```
DECLARE
i INT;
BEGIN
    i := 0;
    <<loop1>>
    WHILE i < 3 LOOP
        i := i + 1;
        CONTINUE loop1 WHEN i = 2;
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    END LOOP loop1;
    DBMS_OUTPUT.PUT_LINE ('The End');
END;
/

--result
This is: 1
This is: 3
The End


DECLARE
    a INT;
BEGIN
    a := 1;
    LOOP
        a :=a + 1;
        DBMS_OUTPUT.PUT_LINE('loop1');
        CONTINUE WHEN a < 5;
        DBMS_OUTPUT.PUT_LINE('loop2');
        EXIT;
    END LOOP;
END;
/

--result
loop1
loop1
loop1
loop1
loop2
```

# DCL Statement

DCL Statement为静态SQL执行语句。

## COMMIT|ROLLBACK

COMMIT/ROLLBACK只能使用缺省的COMMIT/ROLLBACK语法，不能为其指定任何选项。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
no VARCHAR(10);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
BEGIN


EXECUTE IMMEDIATE str1 INTO no,name USING '03';        --动态SQL
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);

UPDATE area SET area_no='00' WHERE area_no='03';    --静态SQL
COMMIT;         --静态SQL

EXECUTE IMMEDIATE str1 INTO no,name USING '03' ;        --动态SQL,由于执行UPDATE语句，此语句将无结果返回
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
END;
/

exec ya_proc;

--result
区域号码:03区域名称:华南
[9:1]YAS-05206 no data found
```

## SAVEPOINT

SAVEPOINT的语法形式为：

*BEGIN*
*SAVEPOINT savepoint_name;*
*...*
*ROLLBACK TO savepoint_name;*
*...*
*END;*

示例

```
--程序捕获到ZERO_DIVIDE异常，执行rollback，之前插入到area表的数据被回滚，所以第二次执行str1语句时抛出未找到数据的异常
DECLARE
a INT := 0;
b INT := 1;
c INT;
no VARCHAR(10);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=''99''';
BEGIN
SAVEPOINT start_trans;
c := 2;
INSERT INTO area VALUES('99','other',DEFAULT);
EXECUTE IMMEDIATE str1 INTO no,name ;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
c := b/a;
EXCEPTION
```

```
WHEN ZERO_DIVIDE THEN
ROLLBACK TO start_trans;
EXECUTE IMMEDIATE str1 INTO no,name ;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
END;
/

--result
区域号码:99区域名称:other
[18:1]YAS-05206 no data found
```

# DML Statement

DML Statement为静态SQL执行语句。其语法形式与常规DML类的SQL语句一致，且可以结合INTO语法为变量赋值。

在过程体中，DML语句可直接执行，或通过EXECUTE Statement调用执行，此时同样可以结合INTO语法为变量赋值。

## 常规用法

不与INTO结合时，静态SQL不能为SELECT语句。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
no VARCHAR(10);
name VARCHAR(20);
BEGIN
UPDATE area SET area_no='00' WHERE area_no='03';
DELETE FROM area WHERE area_no='03';
INSERT INTO area
SELECT area_no+20, area_name, dhq
FROM area
WHERE area_no IN ('01','02','03');
COMMIT;
END;
/

exec ya_proc;
```

## select...into

本语句可以将SELECT语句从数据库中查询到结果赋值给变量，但查询的结果只能是一行记录，不能是零行或多行记录。

对于不能确保获得一行结果集的SELECT语句，需通过游标实现结果集赋值。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
no VARCHAR(10);
name VARCHAR(20);
BEGIN
SELECT area_no,area_name INTO no,name FROM area WHERE area_no='01';
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);

--下面语句将抛出NO_DATA_FOUND异常，通过EXCEPTION Statement将其捕获
SELECT area_no,area_name INTO no,name FROM area WHERE area_no='09';
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('warning:no data found!');
END;
/

exec ya_proc;

--result
区域号码:01区域名称:华东
warning:no data found!
```

## select...bulk collect into

本语句可以将SELECT语句从数据库中查询的结果赋值给集合变量，查询的结果可以是零行或任意数量行。

对使用字符串类型作键的关联数组，不允许使用BULK COLLECT子句。

BULK COLLECT INTO的目标对象必须是集合类型。

示例

```
DECLARE
  TYPE area_tb IS TABLE OF area%rowtype;
  type_001 area_tb;
BEGIN
  SELECT * BULK COLLECT INTO type_001 FROM area ORDER BY 1,2;
  FOR i IN type_001.first..type_001.last LOOP
    DBMS_OUTPUT.PUT_LINE('type_001('||i||'): ' || type_001(i).area_no  || CHR(9) || type_001(i).area_name );
  END LOOP;
END;
/

--result
type_001(1): 01 华东
type_001(2): 02 华西
type_001(3): 03 华南
type_001(4): 04 华北
type_001(5): 05 华中
```

## insert into...return...into

本语句在执行INSERT操作的同时，指定一个返回的结果集，并将其赋值给变量。其中RETURN也可以写为RETURNING。

结果集可以包含一列或多列数据，每一列为一个YashanDB通用表达式，但不能为伪列、聚集函数和窗口函数。

在此种用法中，INSERT语句只能为insert_values_clause语法，不可以为多行或者子查询插入。

列存表无法使用insert into...return...into语句。

示例（HEAP表）

```
CREATE OR REPLACE PROCEDURE ya_proc IS
no INT;
area_name VARCHAR(20);
BEGIN
INSERT INTO area(area_no) VALUES('09')
RETURN TO_NUMBER(area_no), NVL(area_name, 'area'||area_no) INTO no, area_name;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||area_name);
END;
/

exec ya_proc;

--result
区域号码:9区域名称:area09
```

# EXCEPTION Statement

EXCEPTION Statement为异常处理语句，即exception handler，其语法形式为：

*EXCEPTION*

*WHEN exception_name1 THEN*
*statements*

*WHEN exception_name2 THEN statements*

*...*

*WHEN OTHERS THEN*

*statements;*

其中，WHEN后的exception_name可以使用or来连接多个异常，但是OTHERS不可与其他exception_name使用or连接。

在一个程序块'BEGIN END;'中，本语句只能出现一次。可以通过程序块的嵌套，来实现多层EXCEPTION Statement，此时，外层可以捕获内层执行过程中抛出的异常，且对于内层未捕获到的异常，外层可以继续捕获，直到该异常被成功捕获到，或者达到最外层仍未捕获到时，报出错误。

同一层的exception_name对应的错误码不可重复，不同层的exception_name对应的错误码则可以相同。

## exception_name

已被定义的异常名称，用于过程体里抛出异常和exception handler捕获异常时进行一致识别的标识。

异常名称可以为系统预定义的异常名称，或者用户自定义的异常名称，具体描述见异常处理文档。

## exception handler

YashanDB对过程体中的异常处理，规则如下：

- 在执行过程中出错，并且错误码跟EXCEPTION Statement中某个WHEN后面的exception_name匹配，则执行其对应THEN后面的statements。
- 在执行过程中出错，并且错误码无法匹配到EXCEPTION Statement中所有WHEN后面的exception_name，则将其匹配到OTHERS，执行对应THEN后面的statements。
- 在执行过程中出错，并且错误码无法匹配到当前和外层过程体的EXCEPTION Statement中所有WHEN后面的exception_name，则作为未处理异常抛出。
- 执行过程中，如果出错并且捕获了异常；那在异常处理中出现的新错误，会将原先捕获的错误覆盖。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno IN OUT VARCHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
BEGIN
DBMS_OUTPUT.PUT_LINE(vno);
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
IF vno='01' THEN
vno := TO_NUMBER(vno)/0;
END IF;
IF vno='02' THEN
UPDATE area SET area_no='00' WHERE area_no=vno;             --由于'02'在子表上存在外键约束值，此语句将导致外键约束错误
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('no data found');
WHEN ZERO_DIVIDE THEN
BEGIN
DBMS_OUTPUT.PUT_LINE('first:0 divide');
vno := TO_NUMBER(vno)/0;
EXCEPTION
```

```
WHEN NO_DATA_FOUND OR ZERO_DIVIDE THEN
DBMS_OUTPUT.PUT_LINE('second:0 divide');
END;
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('unknown error');
END;
/

DECLARE
 a VARCHAR(2);
BEGIN
DBMS_OUTPUT.PUT_LINE('--------------------------');
a := '10';
ya_proc(a);                         --NO_DATA_FOUND
DBMS_OUTPUT.PUT_LINE('--------------------------');
a := '01';
ya_proc(a);                         --ZERO_DIVIDE
DBMS_OUTPUT.PUT_LINE('--------------------------');
a := '02';
ya_proc(a);                         --OTHERS
END;
/
--result
--------------------------
10
no data found
--------------------------
01
num:01name:华东
first:0 divide
second:0 divide
--------------------------
02
num:02name:westofchina
unknown error
```

# EXECUTE Statement

EXECUTE Statement为动态SQL执行语句，其常规的语法形式为：

*EXECUTE IMMEDIATE v_sql;*

其中，v_sql为SQL语句形式的字符串变量或常量。在YashanDB里，v_sql除了可以为普通SQL语句外，还可以为"BEGIN..END;"形式的匿名块。

## 静态SQL与动态SQL

静态SQL：在过程体创建时就已经确定好的SQL语句，例如DML Statement和DCL Statement。

动态SQL：在过程体运行时才能确定的SQL语句，例如查询的字段、条件等信息来自于用户的输入，或者只能根据程序逻辑来动态构造。

EXECUTE Statement里的v_sql为动态SQL。

## 常规用法

*EXECUTE IMMEDIATE v_sql;*

执行v_sql字符串里包含的SQL语句。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
BEGIN
EXECUTE IMMEDIATE 'alter table area drop DHQ';
END;
/
exec ya_proc;
```

## INTO用法

*EXECUTE IMMEDIATE v_sql INTO v1,v2,...;*

'v1,v2...'为已声明的变量。

执行v_sql字符串里包含的SQL语句（只能是SELECT语句），并将查询结果赋值给变量，'v1,v2...'与SELECT语句的查询项一一对应。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
no VARCHAR(10);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where rownum=1';
BEGIN
EXECUTE IMMEDIATE str1 INTO no,name ;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
END;
/
exec ya_proc;

--result
区域号码:01区域名称:华东
```

## USING用法

*EXECUTE IMMEDIATE v_sql  USING [IN|OUT|IN OUT] a1,a2,...;*

a1,a2...为变量或常量形式的绑定参数。

> **Note**： v_sql如果使用 `:name` 形式占位符且多处同名，按位置绑定值时需要绑定多次。通常同名占位符位置绑定的是相同值。

IN|OUT|IN OUT为绑定参数类型，默认为IN类型。规则如下：

- IN参数可以为变量、变量表达式或常量；
- OUT和IN OUT参数只能为已声明的变量；
- 当v_sql里包含了对另一个过程体的调用时，对绑定参数的类型使用与该过程体定义的参数类型一致。

当USING和INTO同时使用时，INTO语句应该位于USING语句前面。

示例

```
--v_sql为DML语句
CREATE OR REPLACE PROCEDURE ya_proc() IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no in (:a,:b) and rownum=1';
BEGIN
EXECUTE IMMEDIATE str1 INTO no,name USING '01','02';
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
END;
/
exec ya_proc;

--result
区域号码:01区域名称:华东
```

示例（HEAP表）

```
--insert into...return into...
CREATE OR REPLACE PROCEDURE ya_proc IS
no INT;
area_name VARCHAR(20);
str1 VARCHAR(200) := 'INSERT INTO area(area_no) VALUES(:a) RETURN TO_NUMBER(area_no), NVL(area_name, ''area''||area_no) INTO
:b, :c';
BEGIN
EXECUTE IMMEDIATE str1 USING '09',out no,out area_name;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||area_name);
END;
/
exec ya_proc;

--result
区域号码:9区域名称:area09
```

示例

```
--v_sql为调用过程体语句
--定义一个存储过程，其中a2的参数在过程体被赋值输出
CREATE OR REPLACE PROCEDURE ya_block(a1 VARCHAR,a2 IN OUT VARCHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no in (:a,:b) and rownum=1';
BEGIN
EXECUTE IMMEDIATE str1 INTO no,name USING a1,a2;
DBMS_OUTPUT.PUT_LINE('区域号码:'||no||'区域名称:'||name);
a2 := '00';
END;
/
--通过匿名块中调用
DECLARE
a1 VARCHAR(2):='01';
a2 VARCHAR(2):='02';
BEGIN
DBMS_OUTPUT.PUT_LINE('a1:'||a1||',a2:'||a2);
ya_block(a1,a2);
DBMS_OUTPUT.PUT_LINE('a1:'||a1||',a2:'||a2);
END;
```

```
/
--result
a1:01,a2:02
区域号码:01区域名称:华东
a1:01,a2:00

--通过动态SQL调用
CREATE OR REPLACE PROCEDURE ya_proc IS
a1 VARCHAR(2):='01';
a2 VARCHAR(2):='02';
BEGIN
DBMS_OUTPUT.PUT_LINE('a1:'||a1||',a2:'||a2);
EXECUTE IMMEDIATE 'begin ya_block(:a,:b); end;' USING a1,IN OUT a2;
DBMS_OUTPUT.PUT_LINE('a1:'||a1||',a2:'||a2);
END;
/
exec ya_proc;
--result
a1:01,a2:02
区域号码:01区域名称:华东
a1:01,a2:00
```

# EXIT Statement

EXIT Statement为顺序控制语句，其语法形式为：

*EXIT [label_name] [WHEN condition]；*

其中，label_name为可选项，标识需退出的当前或外层循环；WHEN condition为可选项，condition为能产生布尔结果的表达式。

EXIT语句用于退出循环，跳转到循环结束处，继续向下执行。需注意：

- label_name标识需退出的当前或外层循环，若label_name对应的标签不标识当前或外层循环，或为不存在等其他非法情况，将报错处理。label_name可省略，则EXIT语句执行时退出当前循环。
- WHEN condition子句中的条件结果确定是否执行EXIT语句，true为执行，false为不执行。若省略WHEN condition，默认将执行EXIT语句。
- EXIT语句只能出现在循环中，包括WHILE Statement、FOR Statement和LOOP Statement。

示例

```
DECLARE
i INT;
BEGIN
i := 0;
<<loop1>>
LOOP
i := i + 1;
EXIT loop1 WHEN i = 3;
DBMS_OUTPUT.PUT_LINE ('This is: '||i);
END LOOP loop1;
DBMS_OUTPUT.PUT_LINE ('The End');
END;
/

--result
This is: 1
This is: 2
The End
```

# FETCH Statement

FETCH Statement为游标取值语句，系统对显式游标和动态游标的取值方法一致，且格式均为：

*FETCH cursor_name [BULK COLLECT] INTO result_variable[,result_variable] limit numeric_expression;*

其中，cursor_name为显式游标或动态游标的名称，result_variable为一个已声明的变量，用于接收结果集；numeric_expression是数值文本、数值变量或数值表达式，用于限制FETCH获取的行数。

BULK COLLECT为可选项，当使用该项时，会将FETCH的检索结果一次性存储进一个或多个集合变量中。

当需要取值结果集的指定列项时，result_variable可以被声明为一个普通数据类型的变量（或其集合），可同时指定多个列项取值。

当需要取值整个结果集时，result_variable必须被声明为相同列结构的RECORD类型变量（或其集合）或等同于结果集列的多个数据类型变量（或其集合）。

上述两种情况中所定义的列项数据类型均须与结果集列项的数据类型兼容，兼容的含义为当数据类型不一致时系统可以成功执行隐式转换。

游标取值表示将游标指向查询结果集中的行记录，并将其输出到result_variable变量空间中。

## fetch..into

每执行一次FETCH Statement，游标指针移动一次。

游标指针只能往前移动，不能回退。

当前游标指针只能指向某一行，即每移动一次可获取到一行记录。

当游标指针已经走到结束，继续执行FETCH不会导致错误，但游标也不会取到值。

使用RECORD变量承载结果集：

示例

```
DECLARE
  TYPE cursor IS REF CURSOR RETURN area%ROWTYPE;
  cur cursor;
  TYPE record IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 VARCHAR(20)
  );
  rec record;
BEGIN
  OPEN cur FOR SELECT * FROM area;
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' '||rec.c2||' '||rec.c3);
  CLOSE cur;
END;
/

--result
01 华东 Shanghai
02 华西 Chengdu
```

使用多个变量承载结果集：

示例

```
DECLARE
  TYPE cursor IS REF CURSOR RETURN area%ROWTYPE;
  cur cursor;
  c1 CHAR(2);
```

```
  c2 VARCHAR(20);
  c3 VARCHAR(20);
BEGIN
  OPEN cur FOR SELECT * FROM area;
  FETCH cur INTO c1, c2, c3;
  DBMS_OUTPUT.PUT_LINE(c1 ||' '||c2||' '||c3);
  FETCH cur INTO c1, c2, c3;
  DBMS_OUTPUT.PUT_LINE(c1 ||' '||c2||' '||c3);
  CLOSE cur;
END;
/

--result
01 华东 Shanghai
02 华西 Chengdu
```

针对多行结果集，更简便的办法是利用系统提供的游标属性，结合循环语句来进行游标取值。

下表为YashanDB提供的四种游标属性：

| 属性 | 返回值类型 | 作用 |
| --- | --- | --- |
| %isopen | 布尔型 | 判断游标是否打开 |
| %found | 布尔型 | 判断游标是否获取到值 |
| %notfound | 布尔型 | 判断游标是否没有获取到值 |
| %rowcount | BIGINT | 当前成功执行的数据行数 |

示例

```
DECLARE
  CURSOR cur IS SELECT * FROM area;
  c1 CHAR(2);
  c2 VARCHAR(20);
  c3 VARCHAR(20);
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO c1,c2,c3;
    EXIT WHEN cur%notfound;
    DBMS_OUTPUT.PUT_LINE('Line ' || cur%rowcount || ':' );
    DBMS_OUTPUT.PUT_LINE(c1 ||' '||c2||' '||c3);
  END LOOP;
  CLOSE cur;
END;
/

--result
Line 1:
01 华东 Shanghai
Line 2:
02 华西 Chengdu
Line 3:
03 华南 Guangzhou
Line 4:
04 华北 Beijing
Line 5:
05 华中 Wuhan
```

## fetch..bulk collect into

只需执行一次FETCH Statement即可将整个结果集提取到一个或多个集合变量中。

limit用于限制提取的行数。

示例

```
DECLARE
  TYPE cursor IS REF CURSOR RETURN area%ROWTYPE;
  cur cursor;
  TYPE areaType IS TABLE OF area%rowtype;
  areaVar areaType;
BEGIN
  OPEN cur FOR SELECT * FROM area;
  FETCH cur BULK COLLECT INTO areaVar;
  FOR i IN areaVar.first..areaVar.last LOOP
    DBMS_OUTPUT.PUT_LINE(areaVar(i).area_no  ||' '||areaVar(i).area_name ||' '||areaVar(i).DHQ );
  END LOOP;
  CLOSE cur;
END;
/

--result
01 华东 Shanghai
02 华西 Chengdu
03 华南 Guangzhou
04 华北 Beijing
05 华中 Wuhan
```

# FOR Statement

FOR Statement为循环控制语句，其语法形式分为普通用法和for cursor用法两种。

## 普通用法

*for counter in [reverse] lower_bound .. upper_bound loop*

*.....*

*end loop [label_name];*

其中，counter为计数器，lower_bound为计数器下限值，upper_bound为计数器上限值；reverse为控制参数，使用此参数表示将计数器从默认的递增改为递减模式。

counter、lower_bound、upper_bound均为BIGINT类型，取值范围同BIGINT值域。

label_name为可选项，作为一个标签（Label）标识循环体，提升可读性，使用label_name需在循环体开始处有相匹配的标签定义，否则将报错。

### counter

counter为FOR Statement里隐式声明的局部变量，只能在FOR语句内部使用，不可被继承类型，退出FOR语句后失效。

counter可以与过程体全局变量、或者其他循环控制语句内部的局部变量重名，此时 ：

- 在FOR Statement内部，counter表示的是本语句局部变量，且只能由_for counter in [reverse] lower_bound .. upper_bound loop_对其赋值。
- 当需要在FOR Statement内部访问其他的同名变量时，必须使用标签.counter来识别，标签的描述参见GOTO Statement。

### lower_bound|upper_bound

lower_bound和upper_bound可以为常量或变量，以默认递增模式说明如下：

- 当lower_bound<upper_bound时，从lower_bound开始循环赋值到counter，一直到upper_bound结束循环语句；
- 当lower_bound=upper_bound时，lower_bound赋值到counter，循环语句只会执行一次；
- 当lower_bound>upper_bound时，直接跳转到END LOOP结束循环语句。

示例

```
<<G>>
DECLARE
i INT :=5;
BEGIN
<<F1>>
FOR i IN 1..9 LOOP
IF i>5 THEN
DBMS_OUTPUT.PUT_LINE(G.i);     --访问过程体的全局变量
DBMS_OUTPUT.PUT_LINE(i || ':hello');
ELSE
  FOR i IN 1..2 LOOP
  DBMS_OUTPUT.PUT_LINE(F1.i);    --访问上一级FOR语句的局部变量
  DBMS_OUTPUT.PUT_LINE(i || ':world');
  END LOOP;
END IF;
END LOOP;
END;
/

--result
1
1:world
1
2:world
2
1:world
2
2:world
```

```
3
1:world
3
2:world
4
1:world
4
2:world
5
1:world
5
2:world
5
6:hello
5
7:hello
5
8:hello
5
9:hello


DECLARE
BEGIN
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE('loop:'||i);
    END LOOP;
END;
/

--result
loop:1
loop:2
loop:3
```

## for 隐式游标用法

*for counter in (select xx from table) loop*

*.....*

*end loop [label_name];*

其中， counter为隐式声明的局部变量，类型为其后的SELECT查询生成的默认RECORD类型，RECORD成员类型与SELECT查询列的类型一致，RECORD成员个数为查询列个数，不可被继承类型。

本用法为FOR循环支持隐式游标的用法，执行过程为每次循环时，FETCH查询语句的一行数据，将其填充到RECORD类型的counter，直到FETCH数据结束，循环也相应结束。

示例

```
--SELECT查询
DECLARE
resultStr VARCHAR(200);
BEGIN
FOR file IN (SELECT area_no,area_name FROM area WHERE ROWNUM<5) LOOP
resultStr := resultStr||file.area_no||file.area_name;
DBMS_OUTPUT.PUT_LINE(resultStr);
END LOOP;
DBMS_OUTPUT.PUT_LINE(resultStr);
END;
/

--result
01华东
01华东02华西
01华东02华西03华南
```

```
01华东02华西03华南04华北
01华东02华西03华南04华北
```

## for 显式游标用法

*for counter in cursor_x[()] loop*

*.....*

*end loop [label_name];*

其中， cursor_x为一个已定义的显式游标，包括本地定义和全局定义显式游标，支持有参数和无参数。

此处的显式游标用法与正常使用显式游标一致，counter为隐式声明的局部变量，类型为其后的显式游标查询生成的默认RECORD类型，RECORD成员类型与显式查询列的类型一致，RECORD成员个数为查询列个数，不可被继承类型。

本用法为for循环支持显式游标的用法，执行过程为：在for循环的第一次隐式调用显式游标的open操作，之后每次循环隐式调用显式游标的fetch操作，循环结束时隐式调用显式游标的close操作。在for循环体中不允许显式调用作为索引的显式游标的open/fetch/close操作，否则均做报错处理。

示例

```
--显式游标无参数
DECLARE
CURSOR c1 IS SELECT * FROM area ORDER BY area_no;
resultStr VARCHAR(200);
BEGIN
FOR v_sal IN c1 LOOP
resultStr := resultStr||v_sal.area_no||v_sal.area_name;
DBMS_OUTPUT.PUT_LINE(resultStr);
END LOOP;
END;
/

--result
01华东
01华东02华西
01华东02华西03华南
01华东02华西03华南04华北
01华东02华西03华南04华北05华中

--显式游标有参数
DECLARE
CURSOR c1(id INT) IS SELECT branch_no,branch_name FROM branches WHERE area_no = id ORDER BY branch_no;
resultStr VARCHAR(200);
BEGIN
FOR v_sal IN c1(1) LOOP
resultStr := resultStr||v_sal.branch_no||v_sal.branch_name;
DBMS_OUTPUT.PUT_LINE(resultStr);
END LOOP;
END;
/

--result
0101上海
0101上海0102南京
0101上海0102南京0103福州
0101上海0102南京0103福州0104厦门
```

# FORALL Statement

FORALL Statement为批量执行语句。

## 语句定义

**forall::=**

```
syntax::= forall index in lower_bound..upper_bound [SAVE EXCEPTIONS] dml_statement
```

其中，index为索引，lower_bound为索引下限值，upper_bound为索引上限值。 index、lower_bound、upper_bound均为INTEGER类型，取值范围同INTEGER值域。

SAVE EXCEPTIONS为可选项，它能使在某些dml statement执行失败的的情况下继续执行FORALL语句，并将错误信息保存至FORALL特有的游标内，当整个FORALL语句执行完后，抛出一个统一错误码（YAS-06817）。

FORALL Statement的执行体必须是一个单独的dml statement，比如INSERT，UPDATE或DELETE。

**index::=**

index为FORALL Statement里隐式声明的局部变量，只能在FORALL语句内部使用，退出FORALL语句后失效。

**lower_bound|upper_bound::=**

lower_bound和upper_bound为数值文本、数值变量或数值表达式，并按照步进1递增：

- 当lower_bound < upper_bound时，从lower_bound开始循环赋值给index，一直到upper_bound即结束循环语句。
- 当lower_bound = upper_bound时，lower_bound赋值给index，循环语句只会执行一次。
- 当lower_bound > upper_bound时，直接结束FORALL语句。

## 游标定义

**SQL%BULK_EXCEPTIONS**

在YAS-06817的异常处理流程中，可以从隐式游标属性SQL%BULK_EXCEPTIONS获取有关每个DML语句失败的信息。

- SQL%BULK_EXCEPTIONS.COUNT：记录失败的DML语句的条数。
- SQL%BULK_EXCEPTIONS(i).ERROR_INDEX：记录第i个失败的DML语句在所有DML语句中的位置。
- SQL%BULK_EXCEPTIONS(i).ERROR_CODE：记录第i个失败的DML语句的错误码。

**SQL%BULK_ROWCOUNT**

FORALL执行完毕后，可以从隐式游标属性SQL%BULK_ROWCOUNT获取每条DML影响的行数。

SQL%BULK_ROWCOUNT(i)：记录第i条DML语句执行完毕后所影响的行数。

示例

```
DECLARE
    TYPE numListType IS TABLE OF CHAR(2);
    numList numListType := numListType('01','02','03');
BEGIN
    forall i IN 1..6/1.9 save exceptions
        UPDATE area SET DHQ  = DHQ  || '3out_of_size' WHERE area_no  = numList(i);
    EXCEPTION
        WHEN others THEN
            DBMS_OUTPUT PUT_LINE('SQL%BULK_EXCEPTIONS.COUNT : ' || SQL%BULK_EXCEPTIONS.COUNT );

            FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
            DBMS_OUTPUT PUT_LINE('SQL%BULK_EXCEPTIONS('||i||').ERROR_CODE : ' || SQL%BULK_EXCEPTIONS(i) ERROR_CODE );
            DBMS_OUTPUT PUT_LINE('SQL%BULK_EXCEPTIONS('||i||').ERROR_INDEX :' || SQL%BULK_EXCEPTIONS(i) ERROR_INDEX );
            END LOOP;
```

```
            FOR i IN 1..3 LOOP   ----4 out
            DBMS_OUTPUT.PUT_LINE('SQL%BULK_ROWCOUNT('||i||') : ' || SQL%BULK_ROWCOUNT(i) );
            END LOOP;
END;
/

--result
SQL%BULK_EXCEPTIONS.COUNT : 1
SQL%BULK_EXCEPTIONS(1).ERROR_CODE : 4008
SQL%BULK_EXCEPTIONS(1).ERROR_INDEX :3
SQL%BULK_ROWCOUNT(1) : 1
SQL%BULK_ROWCOUNT(2) : 1
SQL%BULK_ROWCOUNT(3) : 0
```

# GOTO Statement

GOTO Statement为顺序控制语句，其语法形式为：

*<<label_name>> statement*

*...*

*GOTO label_name；*

其中，<<>>用于为其后面的statement定义一个标签（Label），label_name为定义的标签的名字。

## <<label_name>>

用于为其后面的statement定义一个标签（Label），label_name为定义的标签的名称。

在同一个PL过程体里，标签的名称不能重复，否则报错。

标签可以位于过程体的任何位置，但如下位置除外：

- END IF前面
- END LOOP前面
- END CASE前面
- END前面
- CASE Statement中的WHEN前面
- EXCEPTION前面
- 过程体结束符'/'前面

当为如下statement定义标签时：

- 以DECLARE开头的程序块，此时除了用于表示跳转位置外，与其连续使用的label_name还可作为这个程序块的名称来使用，例如通过label_name.variable_name，可以访问在这个程序块里声明的变量。
- 以FOR开头的循环语句，此时除了用于表示跳转位置外，label_name还可作为这个FOR语句的名称来使用，例如通过label_name.variable_name，可以访问在这个FOR语句里声明的变量。

## GOTO label_name

GOTO label_name表示跳转到label_name标签所指向的statement，并从此statement开始往下执行。

当GOTO label_name位于某个PL语句的子句里面时，只能向外层跳转，而不能向同一层或更里层跳转。

示例

```
--label\_name未与DECLARE开头的程序块连续使用
CREATE OR REPLACE PROCEDURE ya_proc(a VARCHAR, b VARCHAR) AS
BEGIN
    <<label1>>
    DBMS_OUTPUT.PUT_LINE(a);
    <<label2>>
        DECLARE
        a INT:=3;
        b INT:=3;
        BEGIN
            SELECT * INTO label1.a FROM dual;
            SELECT * INTO label2.b FROM dual;
        DBMS_OUTPUT.PUT_LINE('a+1 = '|| a);
        DBMS_OUTPUT.PUT_LINE('b+1 = '|| b);
        END;
    END;
/
YAS-04253 PL/SQL compiling errors:
[10:27] YAS-04243 invalid identifier "LABEL1"."A"

--label\_name与DECLARE开头的程序块连续使用
CREATE OR REPLACE PROCEDURE ya_proc(a VARCHAR, b VARCHAR) AS
```

```
BEGIN
    <<label1>>
    <<label2>>
        DECLARE
        a INT:=3;
        b INT:=3;
        BEGIN
            SELECT * INTO label1.a FROM dual;
            SELECT * INTO label2.b FROM dual;
        DBMS_OUTPUT.PUT_LINE('a+1 = '|| a);
        DBMS_OUTPUT.PUT_LINE('b+1 = '|| b);
        END;
    END;
/

CREATE OR REPLACE PROCEDURE ya_proc() IS
BEGIN
    CASE WHEN true THEN
    <<a>>
        DBMS_OUTPUT.PUT_LINE('hello');
    WHEN false THEN
        DBMS_OUTPUT.PUT_LINE('world');
    END CASE;
    GOTO a;
END;
/
YAS-04253 PL/SQL compiling errors:
[9:1] YAS-05223 invalid goto statement cannot jump to to label A


CREATE OR REPLACE PROCEDURE ya_proc() IS
BEGIN
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE(i);
        GOTO a;
    END LOOP;
    <<a>>
    CASE WHEN true THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    WHEN false THEN
        DBMS_OUTPUT.PUT_LINE('world');
    END CASE;
END;
/

exec ya_proc;

--result
1
hello
```

# IF Statement

IF Statement为条件选择控制语句，包含如下三种形式：

- IF THEN
- IF THEN ELSE
- IF THEN ELSIF

## IF THEN

格式为：

*IF condition THEN statements END IF;*

其中，condition为能产生布尔值结果的表达式。

含义为：

判断IF语句里的condition条件：

- 如果condition的值为true，执行THEN后面的Statements，然后继续执行到END IF结束；
- 如果condition的值为false，不执行任何操作，跳转到END IF结束。

示例

```
DECLARE
i INT;
BEGIN
    i := 0;
    IF i < 5 THEN
        i := i + 1;
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    END IF;
END;
/

--result
This is: 1
```

## IF THEN ELSE

格式为：

*IF condition THEN statements*
*ELSE statements*
*END IF;*

其中，condition为能产生布尔值结果的表达式。

含义为：

首先判断IF语句里的condition条件：

- 如果condition的值为true，执行IF语句里THEN后面的Statements，然后跳转到END IF结束；
- 如果condition的值为false，跳转到ELSE语句，执行ELSE语句里的statements，然后继续执行到END IF结束。

示例

```
DECLARE
i INT;
BEGIN
    i := 5;
    IF i < 5 THEN
        i := i + 1;
```

```
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    ELSE
        DBMS_OUTPUT.PUT_LINE ('This is: '|| (i-5));
    END IF;
END;
/

--result
This is: 0
```

## IF THEN ELSIF

格式为：

*IF condition_1 THEN statements_1*
*ELSIF condition_2 THEN statements_2*
*.....*

*ELSIF condition_n THEN statements_n*
*[ELSE else_statements]*
*END IF;*

其中，condition_1、*condition_2* 、...、*condition_n* 为能产生布尔值结果的表达式。

含义为：

首先判断IF语句里的condition_1条件：

- 如果condition_1的值为true，执行IF语句里THEN后面的Statements_1，然后跳转到END IF结束；
- 如果condition_1的值为false，跳转到ELSIF，执行与IF语句相同的操作；
- .....
- 如果condition_n的值为true，执行ELSIF语句里THEN后面的Statements_n，然后跳转到END IF结束；
- 如果condition_n的值为false，跳转到下一条语句：

如果下一条语句为ELSE语句，则执行else_statements，然后跳转到END IF结束。

如果下一条语句为END IF语句，则不执行任何操作结束。

示例

```
DECLARE
i INT;
BEGIN
    i := 5;
    IF i < 5 THEN
        i := i + 1;
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    ELSIF i>5 THEN
        DBMS_OUTPUT.PUT_LINE ('This is: '|| (i-5));
    ELSE
        DBMS_OUTPUT.PUT_LINE ('This is: '|| (i+5));
    END IF;
END;
/

--result
This is: 10
```

# LOOP Statement

LOOP Statement为循环控制语句，其语法形式为：

*loop*

*.....*

*end loop [label_name];*

其中，label_name为可选项，作为一个标签（Label）标识循环体，提升可读性，使用label_name需在循环体开始处有相匹配的标签定义，否则将报错。

LOOP Statement无条件进入循环语句，并在结束后再次返回到循环语句的顶部循环执行，除非存在顺序控制语句或引发异常退出循环，因此，在循环语句里必须存在GOTO Statement、EXIT Statement、CONTINUE Statement等顺序控制语句，否则将导致执行进入死循环。

示例

```
DECLARE
i INT;
BEGIN
    i := 0;
    <<loop1>>
    LOOP
        i := i + 1;
        IF i=3 THEN
            GOTO a;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    END LOOP loop1;
    <<a>>
    DBMS_OUTPUT.PUT_LINE ('The End');
END;
/

--result
This is: 1
This is: 2
The End
```

示例

```
DECLARE
a INT :=1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('loop:'||a);
        a := a+1;
        EXIT WHEN a > 2;
    END LOOP;
END;
/

--result
loop:1
loop:2
```

# OPEN Statement

OPEN Statement为打开游标的语句，打开游标表示执行游标所绑定（静态绑定或动态绑定）的SQL语句，并获得查询结果集。

## 打开显式游标

打开一个显式游标的语法格式为：

*OPEN cursor[cursor_parameters];*

其中cursor为显式游标名称，cursor_parameters为显式游标参数的输入值。

如显式游标在声明阶段为某个参数定义了缺省值，则此处其对应的cursor_parameters可省略，否则每个声明的参数在本语句里必须有对应的输入值。

声明游标阶段的cursor_parameters与打开游标阶段的cursor_parameters按顺序一一对应，除非通过"=>"指定了参数的传递顺序。

显式游标在被打开后将保留对应的查询结果集，在将其关闭之前不允许再次执行打开操作。

示例

```
DECLARE
  CURSOR cur(areano CHAR DEFAULT '02', branchno CHAR, orderdate DATE DEFAULT SYSDATE) IS
  SELECT area, branch, order_no FROM orders_info
  WHERE area = areano
  AND branch = branchno
  AND order_date < orderdate;
  TYPE record1 IS RECORD (
    c1 CHAR(2),
    c2 CHAR(4),
    c3 VARCHAR(20)
  );
  rec record1;
BEGIN
  DBMS_OUTPUT.PUT_LINE('First open:');
  OPEN cur(branchno =>'0201');
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' ,'||rec.c2 ||' ,'||rec.c3);
  CLOSE cur;
  DBMS_OUTPUT.PUT_LINE('Second open:');
  OPEN cur('02','0201');
  FETCH cur INTO rec;
  DBMS_OUTPUT.PUT_LINE(rec.c1 ||' ,'||rec.c2 ||' ,'||rec.c3);
  CLOSE cur;
END;
/

--result
First open:
02 ,0201 ,20010102020001
Second open:
02 ,0201 ,20010102020001
```

## 打开动态游标

打开一个动态游标的语法格式为：

*OPEN cursor_variable FOR select_statement;*

其中，cursor_variable为游标变量名称；select_statement为该游标要绑定的SQL语句，该SQL语句可以是静态SQL语句，也可以是通过变量承载的动态SQL语句。当使用动态SQL语句时可以使用USING子句加入绑定参数。此时语法格式变为：

*OPEN cursor_variable FOR string_variable USING bind_variable_list;*

一个动态游标可以被多次打开（可以绑定不同的SQL语句，且返回值类型也可以不同），且不用显式指定关闭游标语句。但在最终不再使用该游标时，应该显式地执行关闭游标操作。

静态SQL语句：

示例

```
DECLARE
  TYPE cursor IS REF CURSOR;
  cur cursor;
  TYPE record1 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20)
  );
  rec1 record1;
  TYPE record2 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 INT
  );
  rec2 record2;
BEGIN
  OPEN cur FOR SELECT area_no, area_name FROM area WHERE area_no='01';
  FETCH cur INTO rec1;
  DBMS_OUTPUT.PUT_LINE(rec1.c1 ||' '||rec1.c2);
  OPEN cur FOR SELECT area_no, area_name, LENGTH(area_name) FROM area WHERE area_no='02';
  FETCH cur INTO rec2;
  DBMS_OUTPUT.PUT_LINE(rec2.c1 ||' '||rec2.c2||' '||rec2.c3);
  CLOSE cur;
END;
/

--result
01 华东
02 华西 2
```

动态SQL语句：

示例

```
DECLARE
  TYPE cursor IS REF CURSOR;
  cur cursor;
  TYPE record1 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20)
  );
  rec1 record1;
  TYPE record2 IS RECORD (
    c1 CHAR(2),
    c2 VARCHAR(20),
    c3 INT
  );
  rec2 record2;
  sql1 CHAR(200) := 'SELECT area_no, area_name FROM area WHERE area_no = :1';
  sql2 CHAR(200) := 'SELECT area_no, area_name, LENGTH(area_name) FROM area WHERE area_no= :1';
  bind1 CHAR(5) := '01';
  bind2 CHAR(5) := '02';
BEGIN
  OPEN cur FOR sql1 USING bind1;
  FETCH cur INTO rec1;
  DBMS_OUTPUT.PUT_LINE(rec1.c1 ||' '||rec1.c2);
  OPEN cur FOR sql2 USING bind2;
  FETCH cur INTO rec2;
  DBMS_OUTPUT.PUT_LINE(rec2.c1 ||' '||rec2.c2||' '||rec2.c3);
  CLOSE cur;
END;
/

--result
```

```
01 华东
02 华西 2
```

# PRAGMA statement

自治事务是由PL中由另一个事务（主事务）启动的独立事务。自治事务不与主事务共享锁，资源或提交依赖关系，可独立执行SQL操作并进行提交或回滚。

## 声明自治事务

声明自治事务可以在程序开头使用如下命令：PRAGMA AUTONOMOUS_TRANSACTION。声明后即开始AUTONOMOUS ROUTINE，其中可包含多个COMMIT以及ROLLBACK。

声明自治事务遵循以下规则：

- 自治事务必须在最外层的DECLARE中的声明。
- 自治事务不是一个嵌套的事务，其提交跟回滚操作不影响主事务。
- 一个自治例程可包含多个自治事务。
- 例程中与事务相关的语句最后必须显式地使用COMMIT或ROLLBACK，否则会导致报错并将语句内容回滚。
- 自治事务执行异常或没有COMMIT/ROLLBACK的场景下会将异常进行上抛，当外层有相应的EXCEPTION时可以对异常进行处理。
- 匿名块，过程体，触发器以及函数可以声明为自治事务，其余类型不可使用。

示例

```
DROP TABLE IF EXISTS address;
CREATE TABLE address (c1 VARCHAR2(27));
CREATE OR replace PROCEDURE Autonomous_Insert
AS
   PRAGMA autonomous_transaction;
BEGIN
    INSERT INTO address VALUES ( 'Autonomous Insert' );
    INSERT INTO address VALUES ( 'Autonomous Insert' );
    COMMIT;
END;
/
BEGIN
    INSERT INTO address VALUES ( 'Commit Block' );
    Autonomous_Insert;
    ROLLBACK;
END;
/
SELECT * FROM address;
--result
C1
----------------------------
Autonomous Insert
Autonomous Insert

DROP TABLE IF EXISTS address;
CREATE TABLE address (c1 VARCHAR2(27));
CREATE OR replace PROCEDURE Autonomous_Insert
AS
   PRAGMA autonomous_transaction;
BEGIN
    INSERT INTO address VALUES ( 'Autonomous Insert' );
END;
/
BEGIN
    INSERT INTO address VALUES ( 'Roll Back Block' );
    Autonomous_Insert;
    ROLLBACK;
END;
/
YAS-05232 active autonomous transaction detected and rolled back
```

# RETURN Statement

RETURN Statement为过程体结束控制语句，执行此语句时将会立即结束当前的过程体运行，并返回Statement内容到调用器。

## RETURN Statement in function

在自定义函数中使用RETURN Statement的语法形式为：

*RETURN v_result;*

其中，v_result可以为变量、常量或表达式。

此语句不可以省略，即过程体中的每个执行路径（流程控制语句的分支，异常处理单元的分支）里都必须包含至少一个RETURN Statement，且v_result不可以省略，但v_result的值可以为NULL。

示例

```
CREATE OR REPLACE FUNCTION ya_func RETURN INT IS
a VARCHAR(10) := '100';
BEGIN
    DBMS_OUTPUT.PUT_LINE('ya_func');
    RETURN TO_NUMBER(a)+10;
END;
/

SELECT TO_CHAR(ya_func) FROM dual;

--result

ya_func
TO_CHAR(YA_FUNC)
-------------------
110


-- EXCEPTION 异常处理
CREATE OR REPLACE FUNCTION ya_func (name_in IN VARCHAR) RETURN DATE IS
entry_out DATE;
BEGIN
    SELECT entry_date INTO entry_out FROM employees WHERE department NOT IN (SELECT deparment_no FROM department);
    RETURN(entry_out);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE ('NO_DATA_FOUND');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE ('Unexpected error');
END;
/


--result
SELECT ya_func('jack') FROM dual;
NO_DATA_FOUND

[11:1]YAS-05228 function must has return sql
```

## RETURN Statement in others

在存储过程、匿名块、自定义高级包(body)、触发器中使用RETURN Statement的语法形式为：

*RETURN ;*

其中，return后不可跟变量、常量或表达式。

此语句可以省略，出现时用于表示后面的语句不再执行，结束程序。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc IS
A VARCHAR(10) := '100';
BEGIN
    DBMS_OUTPUT.PUT_LINE('ya_proc 1');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('ya_proc 2');
END;
/

--result
CALL ya_proc();
ya_proc 1
```

# WHILE Statement

WHILE Statement为循环控制语句，其语法形式为：

*while condition loop*

*.....*

*end loop [label_name];*

其中，condition为能产生布尔结果的表达式。

label_name为可选项，作为一个标签（Label）标识循环体，提升可读性，使用label_name需循环体开始处需有相匹配的标签定义，否则将报错。

WHILE Statement首先判断condition的结果，为true则进入循环语句，并在结束后再次进行判断condition的结果，直到condition的结果为false时结束循环语句，因此：

- 在WHILE Statement前，应该保证condition的结果为true，否则无法进入循环语句执行。
- 在循环语句里，必须存在使condition结果为false的语句，或者存在GOTO Statement、EXIT Statement、CONTINUE Statement等顺序控制语句，否则将导致执行进入死循环。

示例

```
DECLARE
    i INT;
BEGIN
    i := 0;
    WHILE i < 5 LOOP
        i := i + 1;
        IF i=3 THEN
            GOTO a;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('This is: '||i);
    END LOOP;
    <<a>>
    DBMS_OUTPUT.PUT_LINE ('The End');
END;
/

--result
This is: 1
This is: 2
The End



CREATE OR REPLACE PROCEDURE ya_proc(b INT) IS
    a INT := 1;
BEGIN
    WHILE a<=b LOOP
        DBMS_OUTPUT.PUT_LINE ('while:' || a);
        a := a+1;
    END LOOP;
END;
/

exec ya_proc(3);

--result
while:1
while:2
while:3
```

# PL异常处理

在PL程序开发中，需要使用异常处理机制，对各类异常状况进行终止程序或错误规避的逻辑处理，而不是直接抛出错误信息和中止程序。

YashanDB提供如下三类异常：

- 系统预定义的异常，这类异常通常是一些触发了程序内部逻辑的错误，由系统抛出，过程体中可对其进行捕获，例如NO_DATA_FOUND未获得查询结果，LOGIN_DENIED鉴权不通过等。
- 用户自定义的异常，这类异常通常是开发人员依据业务逻辑规则所制定的错误，且须在过程体中自行抛出和捕获，例如身份证号码的合法性判断等。
- 不可预知的其他异常，开发人员无法预知到程序运行时可能会产生的错误，但可以通过OTHERS将其捕获，并进行友好型提示等逻辑处理。

无论上述何种异常，必须在过程体中进行主动捕获，才能接着对其进行逻辑处理，否则，系统将依据错误机制进行相应处理并抛出错误码和错误消息。

## 系统预定义的异常

YashanDB预定义了如下可以在过程体中被识别的异常名称：

| 内置异常名称 | 说明 |
| --- | --- |
| COLLECTION_IS_NULL | 访问的数据集未初始化。 |
| CURSOR_ALREADY_OPEN | 重复打开游标。 |
| DUP_VAL_ON_INDEX | 发生索引冲突。 |
| INVALID_CURSOR | 使用无效的游标。 |
| INVALID_NUMBER | 使用非法数字。 |
| LOGIN_DENIED | 被拒绝登录。 |
| NO_DATA_FOUND | 在存储过程中查询数据为空。 |
| ROWTYPE_MISMATCH | 数据列类型不匹配。 |
| STORAGE_ERROR | 存储错误。 |
| SYS_INVALID_ROWID | 使用非法ROWID。 |
| TIMEOUT_ON_RESOURCE | 访问资源超时。 |
| TOO_MANY_ROWS | 在PL中指定select查询的数据返回了多行。 |
| VALUE_ERROR | 发生数值异常。 |
| ZERO_DIVIDE | 发生除零异常。 |

系统预定义的异常由系统抛出，过程体中只需要进行捕获处理，具体操作可参考EXCEPTION Statement。

## 用户自定义的异常

用户自定义的异常需由用户在过程体中自行生成异常和抛出异常。分为如下两种方式：

**1.直接生成异常**

此方式通过RAISE_APPLICATION_ERROR在需要的地方直接指定错误码和错误消息生成异常，并抛出该错误码和错误消息，无需指定异常名称，并且不对异常进行捕获。

**2.异常定义->异常初始化->异常抛出->异常接收**

此方式需首先进行异常的变量声明（异常定义），之后可对该异常进行错误码绑定（异常初始化），异常抛出和异常捕获（异常接收）等操作。

- 异常定义：声明自定义异常变量，具体操作可参考自定义异常文档说明。
- 异常初始化：对异常绑定错误码，不执行此操作则由系统自动生成错误码。
- 异常抛出：可执行RAISE（捕获后向外层抛出），RAISE exception_name（抛出供本层捕获）两种不同的抛出。

- 异常接收：在异常句柄中捕获该异常并执行逻辑处理，具体操作可参考EXCEPTION Statement。

## OTHERS异常

对于开发人员无法预知的异常，或者过程体中未进行主动捕获的异常，都可以全部归集到OTHERS里进行处理，具体操作可参考EXCEPTION Statement。

## 错误码

YashanDB制定了一套错误码，用于对用户需要感知的错误进行抛出，此时用户只能接收到错误信息且程序被中止。异常处理机制则对指定的错误给予用户对其进行捕获处理的机会，每个异常需要对应相应的错误码。如下为YashanDB的错误码框架：

| 类别 | 错误号区间范围 | 对应方式 |
| --- | --- | --- |
| 系统预定义的错误 | (0 , 20000] | -- |
| 系统预定义的异常 | [20000 , 21000) | 系统生成 |
| 用户自定义的异常 | [21000 , 30000) | EXCEPTION变量声明 |
| 用户自定义的异常 | [30000 , 99999] | RAISE_APPLICATION_ERROR |
| 用户自定义的异常 | (0 , 99999] | EXCEPTION_INIT |

其中，系统预定义的错误见参考手册错误码文档说明。

### RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR为系统提供的异常处理函数，其语法格式为：

*RAISE_APPLICATION_ERROR(errCode, errMsg);*

**errCode**

错误号，需指定为上表范围内的值，否则在执行程序时系统将抛出YAS-04426错误。

**errMsg**

错误消息。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno CHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
BEGIN
IF vno>'10' THEN
RAISE_APPLICATION_ERROR(40000,'no data found for '||vno);
END IF;
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
END;
/

exec ya_proc('99');
--result
YAS-40000 no data found for 99
```

### DBMS_AUDIT_MGMT 自定义异常

调用DBMS_AUDIT_MGMT高级包进行处理，若参数设置不正确会抛出异常。

| 错误码 | 错误内容 | 解释 |
| --- | --- | --- |

| 错误码 | 错误内容 | 解释 |
| --- | --- | --- |
| YAS-30000 | invalid value of argument AUDIT_TRAIL_TYPE | 审计清理类型的设置非法 |
| YAS-30001 | invalid value of argument AUDIT_TRAIL_STATUS_VALUE | 审计清理job状态的设置非法 |
| YAS-30002 | invalid value of argument LAST_ARCHIVE_TIME | 审计清理时间点的设置非法 |
| YAS-30003 | invalid value of argument AUDIT_TRAIL_PURGE_INTERVAL | 下次执行时间的设置非法 |
| YAS-30004 | Cleanup job already existed for the given audit trail type | 给定类型的审计清理job已存在 |

## EXCEPTION_INIT

EXCEPTION_INIT为系统内置程序块，使用该程序块的语法格式为：

*PRAGMA EXCEPTION_INIT(exception_name,errCode)*;

对用户声明的EXCEPTION变量，使用EXCEPTION_INIT可以对其绑定一个错误码，否则系统将按上表范围自动生成一个错误码。

**exception_name**

已被声明的变量名称。

**errCode**

绑定的错误码，需指定为上表范围内的值，否则编译报错。

> **Caution**：
> 禁止定义2017错误码：
> 2017错误码表示DC失效，依据YashanDB内部处理逻辑，在DC失效时会进行reparse后再执行。如手动raise这个错误码且无异常句柄接收，将导致系统陷入不断reparse和执行的死循环。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno CHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
err EXCEPTION;
PRAGMA EXCEPTION_INIT(err, 45000);
BEGIN
IF vno>'10' THEN
RAISE err;
END IF;
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
EXCEPTION
WHEN err THEN
DBMS_OUTPUT.PUT_LINE('no data found for '||vno);
END;
/

exec ya_proc('99');
--result
no data found for 99
```

## SQLCODE&SQLERRM

SQLCODE和SQLERRM为系统提供的异常处理函数，用于获取异常对应的错误号和错误消息，其语法格式为：

*SQLCODE[()]*

*SQLERRM[(errCode)]*

SQLCODE和SQLERRM只能用于过程性语句中，不能用于SQL语句中。当被用于非异常处理的语句中时，SQLCODE函数将返回0值，SQLERRM函数将返回"YAS-00000 normal, successful completion"。

当被用于异常处理语句中时，SQLCODE函数将返回当前捕获到的异常错误号，SQLERRM函数返回规则如下：

- 不指定errCode参数时，返回当前错误对应的错误消息。
- 指定errCode参数时，如errCode超出上表指定范围，返回"%d non-yas exception"，否则查找errCode对应的错误消息并返回。
- 当前错误号或errCode未定义错误消息时，返回"YAS-%05d message %05d not found“。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno CHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
err EXCEPTION;
PRAGMA EXCEPTION_INIT(err, 45000);
BEGIN
DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
IF vno>'10' THEN
RAISE err;
END IF;
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
EXCEPTION
WHEN err THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
END;
/

exec ya_proc('99');
--result
0:YAS-00000 normal, successful completion
45000:YAS-45000 message 45000 not found
```

## STANDARD

STANDARD用于表明其后面的exception_name为系统预定义的异常名称，当用户自定义的异常可能与系统预定义的异常重名时，使用STANDARD可以准确识别出系统预定义的异常。语法格式为：

*STANDARD.exception_name*

exception_name必须为系统预定义异常的名称，否则将出现编译错误。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno CHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
no_data_found EXCEPTION;
PRAGMA EXCEPTION_INIT(no_data_found, 45000);
BEGIN
IF vno>'10' THEN
RAISE no_data_found;
ELSE
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
END IF;
EXCEPTION
WHEN STANDARD.NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('Standard exception.');
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
END;
/
```

```
exec ya_proc('99');
--result
45000:YAS-45000 message 45000 not found

exec ya_proc('09');
--result
Standard exception.
```

## RAISE

RAISE语句用于显式地抛出异常，其语法格式为：

*RAISE [exception_name];*

当不指定exception_name时，该语句必须位于异常句柄（EXCEPTION Statment）中，表示将当前的异常继续向外层抛出。

exception_name可以为用户自定义的异常，也可以是系统预定义的异常。

当本层的异常句柄并没有捕获RAISE后的exception_name时，该异常将向外层抛出，对与用户自定义的异常同名的系统预定义的异常，则需要使用STANDARD捕获。

示例

```
CREATE OR REPLACE PROCEDURE ya_proc(vno CHAR) IS
no VARCHAR(2);
name VARCHAR(20);
str1 VARCHAR(100) := 'select area_no,area_name from area where area_no=:a';
no_data_found EXCEPTION;
BEGIN
IF vno>'10' THEN
BEGIN
DBMS_OUTPUT.PUT_LINE('First:');
RAISE no_data_found;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE;
END;
ELSE
DBMS_OUTPUT.PUT_LINE('Second:');
EXECUTE IMMEDIATE str1 INTO no,name USING vno;
DBMS_OUTPUT.PUT_LINE('num:'||no||'name:'||name);
END IF;
EXCEPTION
WHEN STANDARD.NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(SQLCODE||':'||SQLERRM);
END;
/

exec ya_proc('99');
--result
First:
21000:YAS-21000 message 21000 not found

exec ya_proc('09');
--result
Second:
5206:YAS-05206 no data found
```

# PL源文本加密

可以通过对PL对象进行源代码加密，避免其他人通过数据字典显示该文本。可以加密的PL对象有：

- Package head
- Package body
- Function
- Procedure

对于其他对象，例如trigger，可先将其封装进一个procedure，再通过procedure进行包装加密。

## 加密工具yaswrap

可以通过加密工具yaswrap对PL源代码进行加密操作，具体请参考yaswrap。