SUNDB数据库管理系统
V5.0-PSM手册

PSM Manual

目录

1. PSM概要

1.1 PSM的优点

与SQL紧密配合

SUNDB PSM可以与SUNDB SQL紧密配合使用

- 支持SUNDB SQL支持的所有数据类型
- 支持属性类型（attribute type）（％TYPE％ROWTYPE）因此可以灵活使用表和列（column）类型
- 支持SUNDB SQL支持的所有运算符和内置函数
- 支持SUNDB SQL支持的所有DMLDCL（COMMIT / ROLLBACK等）
- 声明Cursor并使用OPENFETCHCLOSE语句支持SQL SELECT语句
- 通过 dynamic SQL statement功能支持SQL DDL语句

提高性能

SUNDB PSM仅在服务器内部执行因此减少了用户应用程序与DBMS服务器之间的通信次数从而提高了整体性能

提高生产效率

SUNDB PSM语言类似于脚本语言因此可相对轻松地编写所需功能的代码通过将用户的业务逻辑模块化并创建为过程/函数可以大大减少编写客户端应用程序的时间

可移植性

通过SUNDB PSM编写的过程/函数同样可在ODBCJDBC嵌入式SQL和各种开发工具中使用此外与服务器或客户端的平台类型无关同样可以正常运行并且很容易移植到其他类型的平台上

易于管理

SUNDB PSM将存在于每个客户端的类似逻辑以一个模块体现在服务器上因此易于管理并且可在该模块的使用过程中根据需求进行更改

1.2 Language Elements

Data Types

SUNDB PSM提供SUNDB SQL中提供的所有内置数据类型（built-in data type）并支持用户自定义的record及collection类型

详细内容参考PSM 数据类型

Variables

SUNDB PSM中用户根据所需声明的变量均可在所有表达式中使用

详细内容参考下文

- 声明部分 (Declarative Part)
- Assignment

Control Structures

SUNDB PSM支持大多数通用脚本语言所提供的条件分歧语句GOTO等的无条件转移语句用于循环执行的loop语句

详细内容参考PSM Control Statements

Subprograms

SUNDB PSM subprogram为拥有名称且可重复执行的PSM块如果子程序有参数则每次调用时可使用不同的参数进行执行

子程序是过程或函数两种形式中的一个函数形式具有返回值另外支持在特定块中声明并仅在其内部使用的嵌套子程序 (nested subprogram)

详细内容参考 Using PSM Subprograms

1.3 Transaction Processing in PSM

数据库事务是由一个以上的SQL语句构成的无法分解的工作单位在SUNDB数据库中使用事务的SQL语句大致如下

- 除SELECT语句外的DML
- 所有DDL

事务在如下情况下开始

- 连接后第一次执行使用事务的SQL时
- COMMIT或ROLLBACK后第一次执行使用事务的SQL时

当用户执行COMMIT或ROLLBACK或终止连接时事务终止

用SUNDB PSM编写的子程序模块是本身不使用事务的语句因此在调用时不会生成新的事务

但是为了保证SQL语句的原子性（atomicity）子程序中可以使用的SQL类型根据调用相应子程序的情况而受到如下限制

- 用户直接使用CALL语句调用或执行anonymous block时
  - 由于无父类语句（statement）因此可在subprogram中使用所有类型的SQL并可执行COMMIT或ROLLBACK
- 在除SELECT外的DML语句（INSERT / UPDATE / DELETE等）中使用时
  - 由于父类语句（statement）具有事务因此可在 subprogram 中使用所有SQL语句但为了保证父类语句（statement）的原子性不允许执行 COMMIT 或 ROLLBACK
- 在SELECT语句中调用时
  - 由于父类语句（statement）不使用事务因此无法在调用的子程序中使用所有使用该事务的SQL不能使用COMMIT或ROLLBACK并且只能使用SELECT语句

2. PSM 数据类型

2.1 Built-in Data Types

SUNDB PSM同样支持SUNDB SQL中提供的所有基本数据类型

基本数据类型的种类如下

详细内容参考**数据类型**

数字类型

- NUMBER
- NUMERIC
- FLOAT
- NATIVE_INTEGER
- NATIVE_DOUBLE

CHARACTER STRING 类型

- CHARACTER (CHAR)
- CHARACTER VARYING (VARCHAR, VARCHAR2)

- CHARACTER LONG VARYING (LONG VARCHAR)

> Note:
> 虽然SUNDB SQL不支持但为了与其他数据库的兼容性SUNDB PSM提供未指定精度（precision）的VARCHAR(VARCHAR2, CHAR VARYING, CHARACTER VARYING)类型
>
> 该类型无法用于声明一般变量仅可在指定子程序的参数及返回类型游标的参数类型时使用指定该类型时则将对应参数或返回类型决定为在VARCHAR类型中拥有最大大小的类型（precision = 4000）

BINARY STRING类型

- BINARY
- BINARY VARYING (VARBINARY)
- BINARY LONG VARYING (LONG VARBINARY)

日期/时间类型

- DATE
- TIME [WITH/WITHOUT TIME ZONE]
- TIMESTAMP [WITH/WITHOUT TIME ZONE]

INTERVAL类型

- INTERVAL YEAR
- INTERVAL MONTH
- INTERVAL YEAR TO MONTH
- INTERVAL DAY
- INTERVAL HOUR
- INTERVAL MINUTE
- INTERVAL SECOND
- INTERVAL DAY TO HOUR
- INTERVAL DAY TO MINUTE
- INTERVAL DAY TO SECOND
- INTERVAL HOUR TO MINUTE
- INTERVAL HOUR TO SECOND
- INTERVAL MINUTE TO SECOND

BOOLEAN 类型

BOOLEAN

ROWID 类型

ROWID

Built-in数据型变量声明

变量声明可在anonymous block或过程（procedure）函数内部的各声明部分（declaration section）中进行

```
DECLARE
  V_MSG VARCHAR(20) := 'HELLO, WORLD!';
BEGIN
  DBMS_OUTPUT.PUT_LINE( 'My First Message Is : ' || V_MSG );
END;
/
```

详细内容参考Built-in Data Type References

2.2 Attribute Data Types

指定其他PSM变量或游标表或表的特定列（column）等类型时使用的数据类型

通过Attribute类型声明的变量或函数的对应对象（表等）被更改时PSM（procedure函数）自动按照变更后的类型重新编译并应用

%TYPE

用于指定其他变量或特定表的列（column）的类型 可以引用的对象如下

- 标量（scalar）类型(包含built-in数据类型)变量
- 用户自定义记录类型变量
- 记录类型变量的特定字段
- Collection类型变量
- Collection类型变量的特定字段
- 表的特定列（column）

如下使用%TYPE

```
CREATE TABLE EMP ( ID INTEGER, NAME VARCHAR(32) );
INSERT INTO EMP VALUES ( 1001, 'Tom Jackson' );
COMMIT;

DECLARE
  V_NAME EMP.NAME%TYPE;
BEGIN
  SELECT NAME INTO V_NAME FROM EMP;
```

```
  DBMS_OUTPUT.PUT_LINE( 'EMP.NAME = ' || V_NAME );
END;
/
```

%ROWTYPE

指定与特定表的结构或特定游标的返回类型相同的记录类型时使用可引用的对象如下

- Table
- Cursor
- Cursor variable

记录类型变量或collection类型变量不能为%ROWTYPE对象

使用%ROWTYPE的示例如下

```
DECLARE
  V_EMP EMP%ROWTYPE;
BEGIN
  SELECT * INTO V_EMP FROM EMP WHERE ID = 1001;
  DBMS_OUTPUT.PUT_LINE( 'Name of ID 1001 Is : ' || V_EMP.NAME );
END;
/
```

Constraint 属性继承

声明为“属性类型”的变量的约束属性继承了引用对象的约束属性如下所示

Attribute type	引用对象	NOT NULL	Default值
%TYPE	Scalar变量	O	X
	记录类型变量	O	O
	记录类型变量的特定字段	O	X
	Collection类型变量 - scalar element	O	X
	Collection类型变量 - 记录element	O	O
	Collection类型变量的特定字段	O	X
	表的特定列（column）	X	X
%ROWTYPE	表	X	X
	Cursor	X	X

Table 2-1 **属性类型约束继承**

2.3 User-defined Record Type

记录类型变量是由多个不同类型的字段组成的复杂结构类型 记录类型变量可以通过使用%ROWTYPE复制其他表或游标的类型或者声明适合用户特定目的的数据结构来生成

用户自定义记录类型可在PSM声明部分使用TYPE关键字如下定义各字段可选择性地指定非空约束（NOT NULL constraint）与默认值

```
DECLARE
  TYPE MY_EMP_TYPE IS RECORD
  (
    ID INTEGER := 99999,
    NAME VARCHAR(32) NOT NULL DEFAULT 'anonymous'
  );
  V_EMP MY_EMP_TYPE;
BEGIN
  SELECT ID, NAME INTO V_EMP.ID, V_EMP.NAME FROM EMP;
  DBMS_OUTPUT.PUT_LINE('ID = ' || V_EMP.ID);
  DBMS_OUTPUT.PUT_LINE('NAME = ' || V_EMP.NAME);
END;
/
```

以下为在嵌套(nested)过程或嵌套(nested)函数中使用的示例

```
DECLARE
  TYPE MY_EMP_TYPE IS RECORD
  (
    ID INTEGER := 99999,
    NAME VARCHAR(32) NOT NULL DEFAULT 'anonymous'
  );
  V_EMP MY_EMP_TYPE;
```

```
  PROCEDURE SET_EMP( A_EMP IN OUT MY_EMP_TYPE )
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('ID = ' || A_EMP.ID);
    DBMS_OUTPUT.PUT_LINE('NAME = ' || A_EMP.NAME);
    SELECT ID, NAME INTO A_EMP.ID, A_EMP.NAME FROM EMP;
  END;
BEGIN
  SET_EMP( V_EMP );
  DBMS_OUTPUT.PUT_LINE('ID = ' || V_EMP.ID);
  DBMS_OUTPUT.PUT_LINE('NAME = ' || V_EMP.NAME);
END;
```

自定义记录变量类型可用于一般局部变量嵌套(nested)过程或嵌套(nested)函数的参数或返回类型但不可用作schema-level 过程或schema-level 函数的参数或返回类型

2.4 User-defined Collection Type

自定义集合类型(user-defined collection type)指储存一个以上的数据的数组（array）结构SUNDB PSM支持collection type中可以存储为key/value 对（pair）的关联数组（associative array）类型

用于声明associative type的基本语法为如下

```
TYPE <type_name> IS TABLE OF <element_data_type> INDEX BY <index_key_data_type>
```

- <type_name>是用户为该类型指定的名称
- <element_data_type>指定构成数组（array）的value的数据类型
- <index_key_data_type>指定要搜索element的index key的数据类型

例如如果有对应（编号）的信息(姓名年龄)形式则在数据库中可创建并存储如下表

```
CREATE TABLE INFO
(
   NO INTEGER,
   NAME VARCHAR(20),
   AGE INTEGER
)
CREATE UNIQUE INDEX IDX_NO ON INFO (NO)
```

- 如果将上述信息保存为associative array则配置如下
  - Index _key属于编号(NO)
  - Element _data由名字(NAME)年龄(AGE)构成

如下定义PSM中实际存储该信息的associative array变量

```
TYPE rec IS RECORD (NAME VARCHAR(20), AGE INTEGER);
```

```
TYPE info IS TABLE OF rec INDEX BY INTEGER;
```

详细内容参考COLLECTION Variable Declaration（收集变量声明）

Associative Array

关联数组(associative array)类型变量是一种PSM变量具有在index子句中描述的数据类型的key并且可以key/ value的形式存储TABLE OF子句中描述的一个或多个元素数据类型值

- SUNDB PSM的关联数组类型的特征如下
  - 具有INDEX BY子句中描述的数据类型的搜索键并自动进行排序和存储
  - 由TABLE OF子句中描述的元素组成
  - 提供了称为收集方法的搜索功能
  - 当元素存储在现有索引键中时将以替换形式保存
  - 最大存储空间被限制在可用的MEMORY_TEMP_TBS_SIZE大小之内

以下为通过将元素类型声明为SQL数据类型来插入数据的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.



gSQL> DECLARE
  TYPE rec IS TABLE OF VARCHAR(20) INDEX BY VARCHAR(10);
```

```
  V1 rec;
BEGIN
  V1('aa') := 'Dog';
  V1('bb') := 'Cat';

  INSERT INTO T1 VALUES ( V1('aa'), V1('bb') );
END;
/

Anonymous PL block executed.



gSQL> SELECT * FROM T1;

C1  C2
--- ---
Dog Cat

1 row selected.
```

以下为将记录类型（record type）作为元素的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.




gSQL> DECLARE
```

```
  TYPE rec IS TABLE OF T1%ROWTYPE INDEX BY VARCHAR(10);
  V1 rec;
BEGIN
  V1('person1').C1 := 'seoul';
  V1('person1').C2 := '12';

  V1('person2').C1 := 'busan';
  V1('person2').C2 := '24';

  INSERT INTO T1 VALUES V1('person1'), V1('person2');
END;
/

Anonymous PL block executed.


gSQL> SELECT * FROM T1;

C1    C2
----- --
seoul 12
busan 24

2 rows selected.
```

- 以下为在SUNDB PSM的associative array的index key中指定的数据类型
  - INTEGER
  - LONG
  - CHAR
  - VARCHAR
- 以下为可以用作SUNDB PSM关联数组的元素类型的数据类型
  - SQL data type
  - %TYPE

- %ROWTYPE
- User-Defined Record Type

详细内容参考 Built-in Data Types, %TYPE, %ROWTYPE, User-defined Record Type

Assign Values to Collection Variables

以下规则适用于在集合（collection）变量之间进行分配

- 与<assign statement>相同的规则适用于元素分配
- 要分配整个collection变量collection变量的类型必须相同
- 分配collection变量的元素时根据元素数据类型它的分配方式如下
  - 仅在相同类型之间才允许分配用户定义的类型元素
  - 在其他情况下仅当在运行时构成元素的字段的数据类型之间具有兼容性时才可以进行分配

以下为由于assign类型不同而导致报错的情况

```
DECLARE
  TYPE udr1 IS RECORD (F1 INTEGER, F2 VARCHAR(20));
  TYPE udr2 IS RECORD (F1 INTEGER, F2 VARCHAR(20));

  TYPE rec1 IS TABLE OF udr1 INDEX BY VARCHAR(10);
  TYPE rec2 IS TABLE OF udr2 INDEX BY VARCHAR(10);

  V1 rec1;
  V2 rec2;
BEGIN
  V2('person1').F1 := 24;
  V2('person1').F2 := 'seoul korea';
```

```
  V1 := V2;

END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (14:9): ERR-HY000(17007): invalid expression
```

如上述示例虽然由user-defined type构成的element的各字段构成相同但由于变量的类型不同而导致无法执行assign

以下为使用相同的类型正常执行assign的情况

```
DECLARE
  TYPE udr1 IS RECORD (F1 INTEGER, F2 VARCHAR(20));

  TYPE rec1 IS TABLE OF udr1 INDEX BY VARCHAR(10);

  V1 rec1;
  V2 rec1;
BEGIN
  V2('person1').F1 := 24;
  V2('person1').F2 := 'seoul korea';

  V1 := V2;

END;
/

Anonymous PL block executed.
```

要按元素分配关联类型的元素该元素的数据类型必须兼容

参考以下示例

```
gSQL> CREATE TABLE T1
(
  c1 VARCHAR(20),
  c2 VARCHAR(20)
);

Table created.



gSQL> DECLARE
TYPE record_org1 IS RECORD (f1 VARCHAR(20), f2 VARCHAR(20));

TYPE rec1 IS TABLE OF t1%rowtype INDEX BY VARCHAR(10);
TYPE rec2 IS TABLE OF record_org1 INDEX BY VARCHAR(10);

v1 record_org1;
v2 rec1;
v3 rec2;
v4 t1%rowtype;
BEGIN

  -- From record_org1 type to %rowtype
  V2('first') := v1;

  -- From record_org1 type to record_org1 type
  V3('first') := v1;

  -- From t1%rowtype to record_org1 type
  V3('second') := V2('first');

END;
```

```
/

Anonymous PL block executed.
```

关联类型变量存储在易失性存储空间中如果没有足够的空间则必须扩展用户可访问的TEMP TABLESPACE

以下为由于空间不足而导致的错误的示例

```
DECLARE
TYPE rec IS TABLE OF t1%rowtype INDEX BY varchar(20);
v1 rec;
BEGIN
  BEGIN
    FOR i IN 1 .. 100000
    LOOP
      v1(i).c1 := i;
      v1(i).c2 := i;
    END LOOP;

    EXCEPTION WHEN OTHERS THEN
                dbms_output.put_line('error: count=' || v1.count());
                dbms_output.put_line('sqlcode=' || SQLCODE);
                dbms_output.put_line('sqlmsg =' || SQLERRM);
  END;
  dbms_output.put_line('v1.count=' || v1.count());
END;
/
error: count=95004
sqlcode=-14015
sqlmsg =[CSII][PSM][SUNDB]there is no extendible datafile in tablespace
'MEM_TEMP_TBS'
v1.count=95004

Anonymous PL block executed.
```

Collection Method

Collection method指为了便于操作collection type变量而提供的函数（function）或过程（procedure）

Associative array提供以下method

Method	类型	输入参数	返回类型	说明
FIRST	Function	X	Index key data type	返回第一个index key
LAST	Function	X	Index key data type	返回最后一个index key
COUNT	Function	X	INTEGER	返回element的数量
EXISTS	Function	O	BOOLEAN	返回是否存在index key
PRIOR	Function	O	Index key data type	返回输入index key之前的index key
NEXT	Function	O	Index key data type	返回输入index Key之后的index key
DELETE	Procedure	O	N/A	删除属于输入的index key的element

Table 2-2 Collection method

可按如下方式使用collection method

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.



gSQL> DECLARE
  TYPE rec IS TABLE OF T1%ROWTYPE INDEX BY VARCHAR(10);
```

```
  V1 rec;
BEGIN
  V1('person1').C1 := 'seoul';
  V1('person1').C2 := '12';

  V1('person2').C1 := 'busan';
  V1('person2').C2 := '24';

  V1('person3').C1 := 'Daegu';
  V1('person3').C2 := '36';

  -- First method
  DBMS_OUTPUT.PUT_LINE('First Index Key = ' || V1.first() );

  -- Last method
  DBMS_OUTPUT.PUT_LINE('Last Index Key = ' || V1.last() );

  -- Count method
  DBMS_OUTPUT.PUT_LINE('Count of element = ' || V1.count() );

  -- Prior Method
  DBMS_OUTPUT.PUT_LINE('Prior (person1) = ' || V1.prior('person1') ); -- return
NULL
  DBMS_OUTPUT.PUT_LINE('Prior (person3) = ' || V1.prior('person3') );

  -- Next Method
  DBMS_OUTPUT.PUT_LINE('Next (person1) = ' || V1.next('person1') );
  DBMS_OUTPUT.PUT_LINE('Next (person3) = ' || V1.next('person3') ); -- return
NULL

  -- Exists Method
  DBMS_OUTPUT.PUT_LINE('Exists (person2) = ' || V1.exists('person2') );
```

```
  -- Delete Method
  V1.delete('person2');

  -- Exists Method
  DBMS_OUTPUT.PUT_LINE('After delete, Exists (person2) = ' ||
V1.exists('person2') );
END;
/
First Index Key = person1
Last Index Key = person3
Count of element = 3
Prior (person1) =
Prior (person3) = person2
Next (person1) = person2
Next (person3) =
Exists (person2) = TRUE
After delete, Exists (person2) = FALSE

Anonymous PL block executed.
```

如果删除一个元素的删除过程找不到与输入的参数相对应的索引键（index key）则会发生以下错误

```
gSQL> DECLARE
  TYPE rec IS TABLE OF T1%ROWTYPE INDEX BY VARCHAR(10);
  V1 rec;
BEGIN
  V1('person1').C1 := 'seoul';
  V1('person1').C2 := '12';

  -- Call delete procedure
  V1.delete('person2');

END;
```

```
/


ERR-HY000(17045): no data found :
  V1.delete('person2');
  *
ERROR at line 9:
Anonymous PL block executed.
```

2.5 SYS_REFCURSOR

SYS_REFCURSOR为cursor variable的predefined type并用于声明cursor variable

以如下形式声明cursor variable时使用

```
cursor_variable_name SYS_REFCURSOR;
```

如下cursor variable可以与OPEN FOR, FETCH, CLOSE语句共同使用

```
DECLARE
  v1 VARCHAR(20);
  v2 VARCHAR(20);

  cv1 SYS_REFCURSOR;
  cv2 SYS_REFCURSOR;

BEGIN

  OPEN cv1 FOR SELECT * FROM T1;

  cv2 := cv1;

  FETCH cv2 INTO V1, V2;

  DBMS_OUTPUT.PUT_LINE('V1 = ' || V1 || ' , V2 = ' || V2);

END;
/
V1 = Seoul , V2 = 24
```

```
Anonymous PL block executed.
```

- 声明为SYS_REFCURSOR的变量按如下方式使用
  - 可以在多个游标变量(cursor variable)之间相互分配 位于Assign右侧的游标变量已经打开的游标将无法再使用
  - 不能将显式游标(explicit cursor assign)分配给游标变量
  - 不能将不同数据类型的变量分配给游标变量
  - 可用作嵌套函数/过程(nested function/ procedure)或模式级函数/过程(schema-level function/ procedure)的参数

详细内容参考Cursor VariableCursor Variable Declaration（游标变量声明）OPEN FOR Statement

3. PSM Control Statements

3.1 Assignment

使用赋值（assignment）运算符（':='）将右侧表达式计算的结果值插入到左侧变量中

Assignment Target

赋值运算符的左侧是赋值对象可以为如下item

- 变量（包含非IN类型的过程或函数）
- '? ' 或': V1'等绑定参数（仅在anonymous PL block中可用）

Assigning Expression

赋值运算符的右侧可以是可在PSM中使用的表达式PSM的表达式中可用的表达式如下

- 常数
- SUNDB SQL提供的所有运算符内置函数或伪列（pseudo column）
- 绑定到IN或IN OUT类型的绑定参数（ 仅在匿名PL块中可用）
- PSM变量
- PSM嵌套函数

- Schema-level函数

以下为各类型的assignment语法示例

```
CREATE OR REPLACE FUNCTION ADD_TEN( A1 INTEGER )
RETURN INTEGER
IS
BEGIN
  RETURN A1 + 10;
END;
/
COMMIT;

DECLARE
  FUNCTION ADD_ONE( A1 INTEGER )
  RETURN INTEGER
  IS
  BEGIN
    RETURN A1 + 1;
  END;
  V_NUM INTEGER;
  V_STR VARCHAR(10);
BEGIN
  V_NUM := 10;                  ❶ Numeric literal
  V_STR := 'ABC';               ❷ String literal
  :V_PARAM := V_STR || 'DEF';   ❸ PSM variable, SQL operator
  V_NUM := ADD_ONE( 100 ) + ADD_TEN( 1000 );
                                ❹ Nested function, schema-level function
END;
/
```

以下为不可在PSM的表达式中使用的表达式

- 表视图或相应对象的列
- 索引序列同义词(synonym)等SQL对象
- 子查询
- 嵌套程序（nested procedure）
- Schema-level程序（schema-level procedure）

以下为使用错误表达式时发生错误的示例

```
gSQL> CREATE SEQUENCE SEQ1;

Sequence created.

gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
  V1 INTEGER;
BEGIN
  V1 := SEQ1.NEXTVAL;
END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (4:9): ERR-42000(16074): sequence number not allowed here
```

如果需要给无法在PSM表达式中使用的对象赋值可以如下使用SELECT INTO语句

```
gSQL> DECLARE
  V1 INTEGER;
BEGIN
  SELECT SEQ1.NEXTVAL INTO V1 FROM DUAL;
```

```
END;
/

Anonymous PL block executed.
```

Assignment 兼容性

即使是可使用assignment语句的表达式的语法根据目标类型和表达式的最终结果类型可能成功也可能失败根据目标类型可使用的表达式类型为如下

目标类型	Expression最终结果类型
Scalar 类型 • Scalar 类型变量 • 记录类型变量的特定字段 • 具有key值的collection变量的scalar element	仅在标量类型具有可转换为目标类型的值的情况下才允许使用
Attribute记录类型(%ROWTYPE)	仅当字段数相同并且每个字段的值可以转换为目标序列号的字段时才允许使用
用户自定义记录类型	仅允许完全相同的类型
Collection 类型 • 未指定key值的collection类型变量	仅允许完全相同的类型

Table 3-1 Assignment 兼容性

即使是内部结构相同的用户自定义记录类型变量如果类型名称不同会发生以下错误

```
gSQL> DECLARE
```

```
  TYPE MY_REC1 IS RECORD ( F1 INTEGER := 1, F2 VARCHAR(10) := 'AAA' );
  TYPE MY_REC2 IS RECORD ( F1 INTEGER := 1, F2 VARCHAR(10) := 'AAA' );
  V1 MY_REC1;
  V2 MY_REC2;
BEGIN
  V2 := V1;
END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (7:9): ERR-HY000(17007): invalid expression
```

此外target是使用IN属性定义的参数变量或者是使用IN属性绑定的绑定参数也会发生错误

```
gSQL> DECLARE
  FUNCTION ADD_ONE( A1 IN INTEGER )
  RETURN INTEGER
  IS
  BEGIN
    A1 := A1 + 1;
    RETURN A1;
  END;
  V1 INTEGER;
BEGIN
  V1 := ADD_ONE( 10 );
END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (6:5): ERR-HY000(17024): (A1) cannot be used as assignment target
```

当在调用函数或过程时将实际参数值分配给形式参数变量时分配兼容性规则同样适用

3.2 PL Block

PL block 是构成PSM的最基本单位PL block 可以再次嵌套在另一个PL block 中并且PL block 中声明的所有项（item）只能在相应块的范围内（包括子块）引用

PL Block 结构

一个PL block 分为以下三个部分

详细内容参考 块(BEGIN .. END)

```
[DECLARE
❶ 声明部分 (Declarative Part)]
BEGIN
❷ Statements
[EXCEPTION
❸ Handlers]
END;
```

声明部分 (Declarative Part)

声明部分是声明要在PL块内使用的本地（local）项目的部分 如果没有要声明的item（例如局部变量）则它将作为可执行部分开始而无需定义声明部分

可以在声明部分声明的本地Item如下

- 变量 (参考: PSM 数据类型)
- 用户自定义的类型

- 游标
- 嵌套函数或过程(Subprogram)
- 用户自定义Exception

在一个PL块中声明的所有项目（item）的名称与类型无关都应是唯一的例如无法声明与变量同名的游标

```
gSQL> DECLARE
  C1 INTEGER;
  CURSOR C1 IS SELECT * FROM DUAL;
BEGIN
  OPEN C1;
  FETCH C1 INTO C1;
  CLOSE C1;
END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (3:10): ERR-HY000(17027): duplicated identifier
(2) at (5:8): ERR-HY000(17031): mismatch identifier type
(3) at (6:9): ERR-HY000(17031): mismatch identifier type
(4) at (7:9): ERR-HY000(17031): mismatch identifier type
```

但在下层PL块中可以声明与在上层PL块中声明的item具有相同名称的item此时该名称从当前位置开始向上层块进行搜索时指向最先找到的Item

```
gSQL> <<PP>>
DECLARE     ❶ Parent scope begin
  V1 VARCHAR(10) := 'AAA';
BEGIN
  <<CC>>
  DECLARE  ❷ Child Scope Begin
    V1 INTEGER := 99;
```

```
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'PP.V1 = ' || PP.V1 );
    DBMS_OUTPUT.PUT_LINE( 'CC.V1 = ' || CC.V1 );
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
  END;      ❸ Child scope end
END;        ❹ Parent scope end
/

PP.V1 = AAA
CC.V1 = 99
V1 = 99

Anonymous PL block executed.
```

可执行部分（Executable Part)

可执行部分处理声明的Item并执行其中包含的各种语句 可以执行的语句类型如下

- Control Statements (参考: PSM Control Statements)
- Cursor Statements (参考: PSM Control Statements)
- SQL Statements (Static/Dynamic) (参考: Using SQLs in PSM)

异常处理部分（Exception Handling Part)

定义处理执行过程中发生的各种异常（exception）的handler

如果在执行部分执行各种语句时发生异常它将搜索是否注册了一个处理程序来处理从对应的PL块位置到上层PL块发生的异常 如果存在它将执行相关处理程序的内容并初始化异常发生状态

```
DECLARE
```

```
  V1 INTEGER;
BEGIN
  SELECT C1 INTO V1 FROM T1 WHERE C1 = 100;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN   = ' || SQL%ISOPEN);
    DBMS_OUTPUT.PUT_LINE('SQL%FOUND    = ' || SQL%FOUND);
    DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND = ' || SQL%NOTFOUND);
    DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT = ' || SQL%ROWCOUNT);
END;
/
```

如果在搜索最高PL块之后都没有找到合适的处理器（handle）则将发生的异常返回给调用方并终止

```
gSQL> DECLARE
  v1 INTEGER;
BEGIN
  v1 := 1/ 0;
END;
/

ERR-HY000(17007): invalid expression :
  v1 := 1/ 0;
        *
ERROR at line 4:
ERR-22012(12122): divisor is equal to zero
```

有关用户自定义异常的声明内置处理程序（built-in Handler）的类型以及处理程序的安装等详细内容参考Error Handling

3.3 NULL Statement

用于指定在PSM中no-op等不起作用的语句

详细内容参考 NULL Statement

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  IF V1 < 10 THEN
    V1 := V1 + 1;
  ELSE
    NULL;  -- do nothing
  END IF;
END;
/
```

3.4 Testing Conditions

条件分支语句根据指定条件的TURE/FALSE执行语句SUNDB PSM提供IF和CASE两种条件分支语句

IF

IF语句执行指定的表达式条件中属于TRUE的语句表达式条件可以在IFELSIF之后进行描述按照执行时描述的顺序检查条件当表达式条件为TRUE时执行THEN子句下面的语句当IF和ELSIF条件都为假并描述ELSE子句时执行ELSE子句下面的语句

IF语句始终以END IF关键字结束

详细内容参考 IF Statement（IF声明）

```
DECLARE
V1 INTEGER := 1;
BEGIN
  -- IF COND
  IF V1 > 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'IF COND' );
  ELSIF V1 = 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'ELSIF COND' );
  ELSE
    DBMS_OUTPUT.PUT_LINE( 'ELSE COND' );
  END IF;

  -- ELSIF COND
  IF V1 = 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'IF COND' );
```

```
  ELSIF V1 > 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'ELSIF COND' );
  ELSE
    DBMS_OUTPUT.PUT_LINE( 'ELSE COND' );
  END IF;

  -- ELSE COND
  IF V1 = 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'IF COND' );
  ELSIF V1 < 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'ELSIF COND' );
  ELSE
    DBMS_OUTPUT.PUT_LINE( 'ELSE COND' );
  END IF;
END;
```

可描述或者不描述ELSIF子句或ELSE子句

如果未描述ELSE子句并且IF或者ELSIF的任何条件均不是ture则不执行任何语句并在不发生异常的情况下进行下一个statement

CASE

CASE 语句可从IF语句等多个语句的排列中选择一个语句组执行CASE语句以END CASE关键字结束可选择性的指定ELSE子句

如果找不到拥有合适条件的WHEN子句并且描述了ELSE子句则执行ELSE子句以下的语句组如果没有合适的条件又未描述ELSE子句则产生CASE_NOT_FOUND exception

详细内容参考 CASE Statement（CASE 声明）

CASE的WHEN子句按照描述的顺序进行评估（evaluate）在找到符合条件的WHEN子句并执行相应WHEN子句的语句组之后忽略所有的后续WHEN子句

CASE语句有以下两种形式：

- Simple CASE Statement
- Searched CASE Statement

Simple CASE Statement

Simple CASE statement使用CASE关键字后描述的选择表达式执行WHEN子句之后描述的表达式中具有与选择表达式相同值的WHEN子句的语句组后终止

```
DECLARE
V1 VARCHAR(1) := '2';
BEGIN
  CASE V1 WHEN '0' THEN DBMS_OUTPUT.PUT_LINE ('Result = 0');
          WHEN '1' THEN DBMS_OUTPUT.PUT_LINE ('Result = 1');
          WHEN '2' THEN DBMS_OUTPUT.PUT_LINE ('Result = 2');
          ELSE DBMS_OUTPUT.PUT_LINE ('Result = Other');
  END CASE;
END;
/
```

Searched CASE Statement

Searched CASE statement中没有selector expression但每个WHEN子句都有被评估为boolean类型的条件表达式执行时评估每个WHEN子句的表达式执行第一个为TRUE的WHEN子句的语句组后终止

```
DECLARE
V1 VARCHAR(1) := '2';
```

```
BEGIN
  CASE WHEN V1 = 0 THEN DBMS_OUTPUT.PUT_LINE ('Result = 0');
       WHEN V1 = 1 THEN DBMS_OUTPUT.PUT_LINE ('Result = 1');
       WHEN V1 = 2 THEN DBMS_OUTPUT.PUT_LINE ('Result = 2');
       ELSE DBMS_OUTPUT.PUT_LINE ('Result = OTHER');
  END CASE;
END;
/
```

3.5 Iterative Control

迭代控制（iterative control ）语句多次执行一系列语句SUNDB PSM提供的迭代控制语法的类型如下

- Basic loop
- FOR loop
- WHILE loop

Basic Loop

简单（Basic loop）语句包装了一系列语句这些语句将以LOOP和END LOOP关键字重复执行 该语句无限期重复执行内部语句并且可以使用顺序控制（sequential contol）语句EXIT或GOTO跳出循环

详细内容参考基本LOOP语句

以下为使用EXIT WHEN语句从相应位置跳出拥有最近的scope的basic loop的示例

详细内容参考EXIT Statement（退出声明）

```
gSQL> DECLARE
  V1 INTEGER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
    EXIT WHEN V1 > 10;   -- Escape Condition
  END LOOP;
END;
/

V1 = 1
```

```
V1 = 2
V1 = 3
V1 = 4
V1 = 5
V1 = 6
V1 = 7
V1 = 8
V1 = 9
V1 = 10

Anonymous PL block executed.
```

如果要在多个loop重叠的状态中跳出特定loop需在loop前指定 Label并在EXIT语句中将该label指定为target label

```
gSQL> DECLARE
  V1 INTEGER := 1;
BEGIN
  <<OUTER_LOOP>>
  LOOP
    <<INNER_LOOP>>
    LOOP
      DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
      V1 := V1 + 1;
      EXIT OUTER_LOOP WHEN V1 > 5;  -- Escape Condition
    END LOOP INNER_LOOP;
    DBMS_OUTPUT.PUT_LINE( 'END OF INNER LOOP' );
    V1 := V1 + 5;
  END LOOP OUTER_LOOP;
  DBMS_OUTPUT.PUT_LINE( 'END OF OUTER LOOP' );
END;
/
```

```
V1 = 1
V1 = 2
V1 = 3
V1 = 4
V1 = 5
END OF OUTER LOOP

Anonymous PL block executed.
```

FOR Loop

FOR loop语句按照指定范围的整数数量重复执行一系列语句

详细内容参考 FOR LOOP Statement

用户通过FOR关键字后面指定的名称生成索引变量

然后将IN关键字后面的范围操作符(..)左侧指定的下限值作为索引变量的初始值每次loop增加1个索引变量执行给定的一系列statement直到与范围操作符(..)右侧指定的上限值相同

```
BEGIN
  FOR I IN 0 .. 2 LOOP
    DBMS_OUTPUT.PUT_LINE( 'I = ' || I );
  END LOOP;
END;
/

I = 0
I = 1
I = 2

Anonymous PL block executed.
```

指定REVERSE关键字时以范围运算符(..)右侧指定的上限值为索引变量的初始值每次循环时索引变量减1直到达到与左侧指定的下限值相同

```
BEGIN
  FOR I IN REVERSE 0 .. 2 LOOP
    DBMS_OUTPUT.PUT_LINE( 'I = ' || I );
  END LOOP;
END;
/

I = 2
I = 1
I = 0

Anonymous PL block executed.
```

WHILE Loop

当指定条件为TRUE时WHILE loop语句反复执行给定的一系列语句

详细内容参考 WHILE LOOP Statement

```
DECLARE
V1 integer := 0;
BEGIN
  WHILE V1 < 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
  END LOOP;
END;
/
```

WHILE循环在执行body的语句之前先检查给定条件如果从一开始条件就为假则可能一次都不执行body的语句

3.6 Sequential Control

Sequential control语句将程序执行位置从当前执行位置移动到其他位置

SUNDB PSM提供以下三种 sequential control 语句

- GOTO
- CONTINUE
- EXIT

GOTO

GOTO语句将执行位置移动到拥有指定label的语句可在label的PSM内可执行的所有语句前指定GOTO移动的target label可以存在于GOTO语句之前或之后但只有当GOTO当前位置的语句为visible时才能执行GOTO语句

详细内容参考 GOTO Statement

Statement状态为visible的条件如下

- 在当前执行位置前后存在的sibling statement
- 当前位置的parent statement以及其前后的sibling statement

以下为跳转到当前位置的sibling statement的示例

```
gSQL> BEGIN

  <<PARENT_PREV>>
  DBMS_OUTPUT.PUT_LINE('PARENT PREV');

  <<PARENT>>
  IF 1 > 0 THEN
    <<SIBLING_PREV>>
    DBMS_OUTPUT.PUT_LINE('SIBLING PREV');

    <<CURRENT_POSITION>>
    GOTO SIBLING_NEXT;

    DBMS_OUTPUT.PUT_LINE('SIBLINGS');
    DBMS_OUTPUT.PUT_LINE('SIBLINGS');

    <<SIBLING_NEXT>>
    DBMS_OUTPUT.PUT_LINE('SIBLING NEXT');
  ELSE
    <<NON_SIBLING>>
    DBMS_OUTPUT.PUT_LINE('NON SIBLING');
  END IF;

  <<PARENT_NEXT>>
  DBMS_OUTPUT.PUT_LINE('PARENT NEXT');

END;
/

PARENT PREV
SIBLING PREV
SIBLING NEXT
PARENT NEXT
```

```
Anonymous PL block executed.
```

以下为跳转到当前位置的parent statement的示例

```
gSQL> BEGIN

  <<PARENT_PREV>>
  DBMS_OUTPUT.PUT_LINE('PARENT PREV');

  <<PARENT>>
  IF 1 > 0 THEN
    <<SIBLING_PREV>>
    DBMS_OUTPUT.PUT_LINE('SIBLING PREV');

    <<CURRENT_POSITION>>
    GOTO PARENT_NEXT;

    DBMS_OUTPUT.PUT_LINE('SIBLINGS');
    DBMS_OUTPUT.PUT_LINE('SIBLINGS');

    <<SIBLING_NEXT>>
    DBMS_OUTPUT.PUT_LINE('SIBLING NEXT');
  ELSE
    <<NON_SIBLING>>
    DBMS_OUTPUT.PUT_LINE('NON SIBLING');
  END IF;

  <<PARENT_NEXT>>
  DBMS_OUTPUT.PUT_LINE('PARENT NEXT');

END;
/
```

```
PARENT PREV
SIBLING PREV
PARENT NEXT

Anonymous PL block executed.
```

以下为在尝试跳转到非Sibling的Statement时报错的示例

```
gSQL> BEGIN

  <<PARENT_PREV>>
  DBMS_OUTPUT.PUT_LINE('PARENT PREV');

  <<PARENT>>
  IF 1 > 0 THEN
    <<SIBLING_PREV>>
    DBMS_OUTPUT.PUT_LINE('SIBLING PREV');

    <<CURRENT_POSITION>>
    GOTO NON_SIBLING;

    DBMS_OUTPUT.PUT_LINE('SIBLINGS');
    DBMS_OUTPUT.PUT_LINE('SIBLINGS');

    <<SIBLING_NEXT>>
    DBMS_OUTPUT.PUT_LINE('SIBLING NEXT');
  ELSE
    <<NON_SIBLING>>
    DBMS_OUTPUT.PUT_LINE('NON SIBLING');
  END IF;

  <<PARENT_NEXT>>
```

```
  DBMS_OUTPUT.PUT_LINE('PARENT NEXT');

END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (12:10): ERR-HY000(17010): can not find label name
```

同样如果尝试移动到当前语句的同级语句以外的语句则会发生错误

GOTO是使PSM程序逻辑灵活的好工具但与其他语言相同过多的使用会降低程序的可读性难以维护因此建议对于一般逻辑使用FOR或WHILE等的循环语法只在必要时使用GOTO

CONTINUE

CONTINUE语句在FOR或WHILE等的循环（Loop）中结束当前迭代（Iteration）后启动下一次迭代

详细内容参考CONTINUE Statement（CONTINUE 声明）

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  WHILE V1 <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
    IF V1 <= 2 THEN
      DBMS_OUTPUT.PUT_LINE( 'CONTINUE' );
      CONTINUE;
    ELSE
      EXIT;
    END IF;
```

```
    DBMS_OUTPUT.PUT_LINE( 'END-OF-WHILE' );
  END LOOP AAA;
END;
/
```

如果指定了target label则启动具有该label的循环的下一次迭代如未指定则启动最近的（内部的）循环的下一次迭代

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  FOR I IN 0..5 LOOP
    <<BBB>>
    WHILE V1 <= 10 LOOP
      DBMS_OUTPUT.PUT_LINE( 'I = ' || I );
      DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
      V1 := V1 + 1;
      IF V1 <= 2 THEN
        CONTINUE BBB;
      ELSE
        EXIT AAA;
      END IF;
    END LOOP BBB;
  END LOOP AAA;
END;
/
```

CONITINUE子句可以选择性的描述WHEN子句在这种情况下仅当WHEN子句后面的表达式为TRUE时终止当前迭代后启动下一次迭代

```
DECLARE
  V1 INTEGER :=0 ;
```

```
BEGIN
  LOOP
    V1 := V1 + 1;
    CONTINUE WHEN V1 < 5;
    EXIT;
  END LOOP;
END;
/
```

EXIT

EXIT语句跳出当前的循环并执行下一个语句

详细内容参考EXIT Statement（退出声明）

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  WHILE V1 <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    IF V1 > 5 THEN
      EXIT;
    END IF;
    V1 := V1 + 1;
  END LOOP;
END;
/
```

如果描述了目标循环则带有相应标签的循环将跳出

```
DECLARE
```

```
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  FOR I IN 0..5 LOOP
    <<BBB>>
    WHILE V1 <= 10 LOOP
      DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
      IF V1 >= 5 THEN
        EXIT AAA;
      END IF;
      V1 := V1 + 1;
    END LOOP BBB;
  END LOOP AAA;
END;
/
```

与CONTINUE语句相同EXIT语句也可以选择性的描述WHEN子句在这种情况下只有在指定条件为TRUE时才执行EXIT

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  FOR I IN 0..5 LOOP
    <<BBB>>
    WHILE V1 <= 10 LOOP
      DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
      EXIT AAA WHEN V1 >= 5;
      V1 := V1 + 1;
    END LOOP BBB;
  END LOOP AAA;
END;
/
```

3.7 Error Handling

根据发生的时间可将PSM中可能发生的error类型分类如下

- Errors at compile time
- Errors at run time

Errors at Compile Time

Compile error输出在验证过程/函数中描述的PSM语句的过程中发生的错误

如果出现语法错误则仅显示找到的第一个语法错误的位置如下所示因此用户必须在更正后重新编译以确保没有其他错误

```
CREATE OR REPLACE PROCEDURE PROC1
IS
BEGIN
  V1 = 1;
  V2 = 1;
END;
/

ERR-42000(40000): syntax error:
  V1 = 1;
.....^
Error at line 4
```

如果没有syntax错误SUNDB将在编译PSM statement的过程中进行验证并如下输出发生的错误

```
CREATE OR REPLACE PROCEDURE PROC1
IS
BEGIN
  V1 := 1;
  V2 := 2;
END;
/

ERR-01000(16409): Warning: Routine definition has compilation errors
ERR-HY000(17032): PSM compilation error :
(1) at (4:3): ERR-HY000(17006): unknown variable or column name (V1)
(2) at (5:3): ERR-HY000(17006): unknown variable or column name (V2)
Procedure created.
```

Compile error在SUNDB自身的错误输出缓冲区允许的大小范围内输出错误因此可能存在未输出的错误此时以如下形式显示存在未输出的错误

```
CREATE OR REPLACE PROCEDURE PROC1
IS
BEGIN
V1 := 1;
V2 := 2;
V3 := 2;
V4 := 2;
V5 := 2;
V6 := 2;
V7 := 2;
V8 := 2;
V10 := 2;
END;
/

ERR-01000(16409): Warning: Routine definition has compilation errors
```

```
ERR-HY000(17032): PSM compilation error :
(1) at (4:3): ERR-HY000(17006): unknown variable or column name (V1)
(2) at (5:3): ERR-HY000(17006): unknown variable or column name (V2)
(3) at (6:3): ERR-HY000(17006): unknown variable or column name (V3)
(4) at (7:3): ERR-HY000(17006): unknown variable or column name (V4)
(5) at (8:3): ERR-HY000(17006): unknown variable or column name (V5)
(6) at (9:3): ERR-HY000(17006): unknown variable or column name (V6)
(7) at (10:3): ERR-HY000(17006): unknown variable or column name (V7)
2 more errors...
```

以如下形式输出错误消息

```
(Index) at (Line_Number : Column_Position): ERR-SQLState( Internal_ErrorCode):
Detail_Error_Message
```

- Index
  - 表示发生错误的顺序
- Line_Number
  - 表示发生错误的源（source）的行（line）位置
- Column_Position
  - 表示发生错误源的列（column）位置
- ERR-SQLState
  - 表示标准SQLState
- Internal_ErrorCode
  - 内部错误代码
- Detail_Error_Message
  - 错误消息详情

Errors at Run Time

在正常执行PSM时由于多种原因PSM statement中也可能产生错误在这种情况下SUNDB通过提供异常处理（exception handling）等使用户可以控制错误

以下为执行时发生错误的示例如果在执行SUNDB PMS时发生错误则如下同时输出错误位置和PSM错误以及internal error即可能存在多个错误因此如果用户要在ODBC等开发过程中查看准确的错误消息则应通过SQLGetDiagRec等函数读取数据库中的所有错误

```
DECLARE
  V1 INTEGER;
BEGIN
  V1 := 1 / 0;
END;
/

ERR-HY000(17007): invalid expression :
  V1 := 1 / 0;
        *
ERROR at line 4:
ERR-22012(12122): divisor is equal to zero
```

如果在执行时发生错误则PSM立即停止并向用户通知该错误但是如果用户要直接控制错误并继续运行程序时则应进行异常处理（exception handling）

以下为对产生错误的位置进行异常处理（exception handling）使用户程序不中断运行的示例

```
DECLARE
  V1 INTEGER;
BEGIN
  BEGIN
```

```
    V1 := 1 / 0;
  EXCEPTION WHEN OTHERS
            THEN DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                 DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
  END;

  DBMS_OUTPUT.PUT_LINE('next code');
END;
/
SQLCODE = -12122
SQLERRM = [CSII][PSM][SUNDB]divisor is equal to zero
next code

Anonymous PL block executed.
```

详细内容参考异常处理（Exception Handling） SQLCODE Function SQLERRM Function

Cursor Attributes

Cursor attributes指在处理PSM的过程中为了获取数据库内部或用户指定的cursor状态而提供的属性（变量）

- Cursor分为如下
  - Implicit cursor
    - 数据库内部需要时被OPEN/FETCH/CLOSE的cursor
  - Explicit cursor
    - 用户明确declaration/definition后执行OPEN/FETCH/CLOSE的cursor

Implicit Cursor Attributes

PSM中的Implicit Cursor Attributes用于查看之前执行的SQL语句的处理状态Attributes的各个值可以用在条件式等的表达式中因此用户可以通过该值控制程序分支逻辑等

有关各属性的详细内容参考下表

Attribute name	返回类型	说明
ISOPEN	BOOLEAN	内部为close状态所以始终为FALSE
FOUND	BOOLEAN	如果前一个statement返回数据则为TRUE否则为FALSE
NOTFOUND	BOOLEAN	与FOUND相反的值
ROWCOUNT	INTEGER	受前一个statement影响的行数

Table 3-2 Implicit cursor attributes

使用的语法形式如下

```
SQL%Attribute_name

Attribute_name := ISOPEN | FOUND | NOTFOUND | ROWCOUNT
```

如下可查看各属性

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);
Table created.

gSQL> INSERT INTO T1 VALUES ('Seoul', '24');
1 row created.
```

```
gSQL> INSERT INTO T1 VALUES ('Pusan', '44');
1 row created.

gSQL> BEGIN
    UPDATE T1 SET C2 = C2 + 1;

    DBMS_OUTPUT.PUT_LINE('ISOPEN   = ' || SQL%ISOPEN );
    DBMS_OUTPUT.PUT_LINE('FOUND    = ' || SQL%FOUND );
    DBMS_OUTPUT.PUT_LINE('NOTFOUND = ' || SQL%NOTFOUND );
    DBMS_OUTPUT.PUT_LINE('ROWCOUNT = ' || SQL%ROWCOUNT );
END;
/
ISOPEN   = FALSE
FOUND    = TRUE
NOTFOUND = FALSE
ROWCOUNT = 2

Anonymous PL block executed.
```

Explicit Cursor Attributes

Explicit cursor attributes用于查询explicit cursor的状态

有关各属性的详细内容参考下表

属性	返回类型	说明
%ISOPEN	BOOLEAN	仅在cursor正常打开时为TRUE否则为FALSE
%FOUND	BOOLEAN	执行FETCH之前为NULL正常执行FETCH时为TRUE无数据时为FALSECLOSE后为NULL
%NOTFOUND	BOOLEAN	具有与%FOUND相反的值
%ROWCOUNT	INTEGER	在OPEN之前为NULL正常OPEN时为0并且每次成功执行FETCH时增加1

Table 3-3 Cursor attributes

在PSM内如下使用

```
Cursor_Name % Attribute_name
Attribute_name :=  ISOPEN
                 | FOUND
                 | NOTFOUND
                 | ROWCOUNT
```

以下为使用explicit cursor attribute的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);
Table created.

gSQL> INSERT INTO T1 VALUES ('Seoul', '24');
1 row created.

gSQL> INSERT INTO T1 VALUES ('Pusan', '44');
1 row created.

gSQL> DECLARE
  CURSOR C1 IS SELECT * FROM T1;
  v1 c1%ROWTYPE;
BEGIN
```

- 通过ISOPEN查看游标是否为打开的状态

```
IF C1%ISOPEN = FALSE
  THEN
      OPEN C1;
```

```
  END IF;



  LOOP
      FETCH C1 INTO v1;
```

- 通过NOTFOUND查看是否还有数据

```
EXIT WHEN C1%NOTFOUND;
```

- 通过ROWCOUNT查看fetch的row数

```
DBMS_OUTPUT.PUT_LINE('COUNT = ' || C1%ROWCOUNT );
  END LOOP;

  CLOSE C1;

END;
/
COUNT = 1
COUNT = 2

Anonymous PL block executed.
```

异常处理（Exception Handling）

PSM中的exception定义用户执行PSM的过程中产生的各种错误在PSM执行过程中若产生错误会立即中断并向用户返回错误异常处理（exception handling）功能使用户控制EXCEPTION情况并允许程序不间断运行

Exception Handler

如下可通过PSM BLOCK中定义的exception handler控制错误情况

```
DECLARE
  V1 INTEGER;
BEGIN
  BEGIN
    V1 := 1 / 0;
```

- Exception handler

```
EXCEPTION WHEN OTHERS
            THEN DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                 DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
  END;

  DBMS_OUTPUT.PUT_LINE('next code');
END;
/
SQLCODE = -12122
SQLERRM = [CSII][PSM][SUNDB]divisor is equal to zero
next code
```

以上示例中执行时程序被divide by zero中断但是由于exception handler发生错误时只中断相应的BLOCK并从下一个位置继续执行

Exception handler通过以下形式进行描述

```
BEGIN
EXCEPTION WHEN exception_name, [exception_name, ...] THEN pl_statements;
         [WHEN exception_name, [exception_name, ...] THEN pl_statements;
```

END

- Exception handler有以下特征
  - 在BLOCK中描述一个exception handler
  - 可以在exception handler中指定多个Exception_Name并定义其行为
  - Exception handler中的Exception_name不能重复
  - 仅对描述exception handler的PL block SCOPE中发生的错误执行
  - 表示所有例外情况的OTHERS应是唯一的并应在最后面进行描述
  - 如果exception handler正常完成则清除之前发生的错误必要时用户需将其存储在单独的PSM变量中

SUNDB中的exception具有以下特性

类型	Definer	Error code	Name	Raise implicitly	Raise explicitly
Predefined	Database	O	O	O	Optionally
User-defined	User	用户指定	用户指定	X	O

Table 3-4 EXCEPTION 类型

Note:

SUNDB中提供的Predefined exception或User-defined exception的详细内容参考Exception Declaration（例外声明）

Exception的传播

产生exception并且当前BLOCK中没有合适的exception handler则exception将传播到上层BLOCK的exception handler直到被处理

在如下代码中的LINE_1中发生错误则执行结果取决于exception handler是否正在运行

```
BEGIN
  ...
  BEGIN
    LINE_1
    EXCEPTION HANDLER_1
  END;
  LINE_2
  EXCEPTION HANDLER_2
END;
/
```

- 在EXCEPTION_HANDLER_1中处理时
  - 当handler的操作正常完成时将从LINE_2开始执行
- EXCEPTION_HANDLER_1中没有合适的handler时
  - LINE_2无法执行向EXCEPTION_HANDLER_2传递当前产生的EXCEPTION
    - 如果EXCEPTION_HANDLER_2中有合适的Handler则程序将在handler正常完成其操作后结束
    - 如果EXCEPTION_HANDLER_2中没有Handler则程序将停止并向用户传递当前错误

> Caution:
>
> 如果处理异常的exception handler中发生错误则handler中发生的错误代码将传播到上层BLOCK

```
DECLARE
  V1 INTEGER;
  E1 EXCEPTION;
  E2 EXCEPTION;
BEGIN
  BEGIN
    RAISE E1;
  EXCEPTION WHEN E1 THEN V1 := 1 / 0;    ❶ 产生新的错误
  END;
```

```
  EXCEPTION WHEN E1
            THEN DBMS_OUTPUT.PUT_LINE('User Exception');
            WHEN ZERO_DIVIDE
            THEN DBMS_OUTPUT.PUT_LINE('Zero divide');
END;
/
Zero divide

Anonymous PL block executed.
```

User-defined Exception

用户定义的异常（user-defined exception）是用户通过设置除预定义之外的异常名称和错误代码来使用的异常 用户定义的异常无法自动（implicitly）触发并且必须由用户使用RAISE语句显式触发

以下为使用RAISE statement触发exception的示例

```
DECLARE
  V1   INTEGER;
  high EXCEPTION;
  low  EXCEPTION;
BEGIN

  BEGIN
      V1 := 10;

      IF V1 > 10
      THEN
          RAISE high;
      ELSE
          RAISE low;
      END IF;
```

```
  EXCEPTION WHEN high
            THEN DBMS_OUTPUT.PUT_LINE('High');
            WHEN low
            THEN DBMS_OUTPUT.PUT_LINE('Low');
  END;
END;
/
Low
```

- User-defined exception的处理分为以下两种类型
  - 设置error code时
  - 未设置error code时

用户设置error code时操作如下

```
DECLARE
  V1 INTEGER;
  E1 EXCEPTION;
  PRAGMA EXCEPTION_INIT( E1, -12122);
BEGIN
  BEGIN
    V1 := 1 / 0;
  EXCEPTION WHEN E1
            THEN DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                 DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
  END;
END;
/
SQLCODE = -12122
SQLERRM = [CSII][PSM][SUNDB]divisor is equal to zero
```

存在ZERO_DIVIDE predefined exception但用户将错误代码设置为E1 exception并设置handler如上为了确保拥有不同错误代码的供应商之间的程序兼容性用户可以使用亲自重新设置exception的错误代码的exception handler

未设置error code的user-defined exception在exception handler中设置为以下错误代码和错误信息

```
DECLARE
  V1 INTEGER;
  E1 EXCEPTION;
BEGIN
  BEGIN
    RAISE E1;
  EXCEPTION WHEN E1
            THEN DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                 DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
  END;
END;
/
SQLCODE = 1
SQLERRM = [CSII][PSM][SUNDB]User-Defined Exception

Anonymous PL block executed.
```

User-defined exception的传播过程与Predefined exception的传播过程相同但是如果未在user-defined exception中设置error code则当传播到上层BLOCK时发送为unhandled user exception

```
DECLARE
  V1 INTEGER;
  E1 EXCEPTION;
  E2 EXCEPTION;
BEGIN
  BEGIN
    RAISE E1;
```

```
  EXCEPTION WHEN E2
            THEN DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                 DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
  END;
END;
/

ERR-HY000(17017): Unhandled user exception :
    RAISE E1;
          *
ERROR at line 7:
```

Unhandled user exception可在exception handler中描述user-defined exception或使用为predefined exception的OTHERS进行catch

PRAGMA EXCEPTION_INIT

用于使用user-defined exception时用户设置特定DBMS错误代码由于每个供应商的DBMS错误代码均不同因此可为了PSM代码的exception handler兼容性而使用

其语法如下

```
<PRAGMA EXCEPTION_INIT> ::=
    PARAGMA EXCEPTION_INIT ( exception_name, internal_errorcode )
```

- 具有以下约束事项
  - exception_name需提前声明
  - exception_name不能使用Predefined exception
  - internal_errorcode只能使用SUNDB的internal error code
  - 无法设置为表示SUCCESS的0值

以下为在适用NOT NULL constraints的PSM变量V1中assign NULL时将exception以user-defined exception进行handing的示例

```
DECLARE
  V1 VARCHAR(20) NOT NULL := 10;
  USER_EXCEPT EXCEPTION;
```

- 将PSM NOT NULL constrainst错误声明为用户自定义exception

```
PRAGMA EXCEPTION_INIT( USER_EXCEPT, -17009 );
BEGIN
```

- 触发exception情况

```
V1 := NULL;

  EXCEPTION WHEN USER_EXCEPT THEN DBMS_OUTPUT.PUT_LINE('Check Value');
END;
/
Check Value

Anonymous PL block executed.
```

设置非SUNDB的internal error code值时会发生以下错误

```
DECLARE
  E1 EXCEPTION;
  PRAGMA EXCEPTION_INIT( E1, 99999);
BEGIN
    NULL;
END;
/
```

```
ERR-HY000(17032): PSM compilation error :
(1) at (3:30): ERR-HY000(17021): Invalid error number for PRAGMA EXCEPTION_INIT
```

DBMS_STANDARD.RAISE_APPLICATION_ERROR

当只通过用户错误代码和消息引发异常而不明确定义用户异常时使用其属于DBMS_STANDARD模块即使省略了模块名称也可以调用该模块的例程

有以下三个参数

参数	说明
error_code IN NATIVE_INTEGER	任意生成的错误代码值仅可使用-20000到-20999范围内的值
error_message IN VARCHAR(4000)	发生错误时存储在SQLERRM中的消息
stack_flag IN BOOLEAN := FALSE	是否堆叠现有错误（TRUE）还是关于是否替换所有现有错误（FALSE）的flag默认值为FALSE

Table 3-5 RAISE_APPLICATION_ERROR

以下为一个简单示例

```
CREATE OR REPLACE PROCEDURE PROC1
IS
BEGIN
  RAISE_APPLICATION_ERROR (-20000, 'USER DEFINED ERROR TEST');
END;
/

Procedure created.

BEGIN
```

```
  PROC1;
EXCEPTION WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE( 'SQLCODE : ' || SQLCODE || ' SQLERRM : ' || SQLERRM );
END;
/
SQLCODE : -20000 SQLERRM : [CSII][PSM][SUNDB]USER DEFINED ERROR TEST

Anonymous PL block executed.
```

4. PSM Cursor Statements

4.1 Implicit Cursor

隐式游标（implicit cursor）是在数据库中由PSM statement定义并管理的游标PSM statement在每次执行SELECT语句或DML语句时使用隐式游标用户虽然无法控制隐式游标但可以通过隐式游标属性（implicit cursor attribute）获取查询信息

Implicit Cursor Attributes

隐式游标属性（implicit cursor attributes）引用最近执行的SELECT语句或DML语句的执行结果如果最近没有执行SELECT语句或DML语句则attributes值为NULL

属性	返回类型	说明
SQL%ISOPEN	BOOLEAN	因最近执行的SELECT语句或DML语句总是终止所以始终返回FALSE
SQL%FOUND	BOOLEAN	当最近执行的SELECT语句或DML语句中有返回结果时为TRUE否则为FALSE
SQL%NOTFOUND	BOOLEAN	拥有与SQL%FOUND相反的值
SQL%ROWCOUNT	INTEGER	为最近执行的SELECT语句或DML语句中返回的ROW的数量

Table 4-1 **隐式游标属性**

示例

- 没有最近执行的查询时

```
gSQL>
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN   = ' || SQL%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('SQL%FOUND    = ' || SQL%FOUND);
  DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND = ' || SQL%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT = ' || SQL%ROWCOUNT);
END;
/

SQL%ISOPEN   = FALSE
SQL%FOUND    =
SQL%NOTFOUND =
SQL%ROWCOUNT =
Anonymous PL block executed.
```

- 当存在最近执行查询时

```
gSQL>
CREATE TABLE t_month( month_char VARCHAR(15) , month_num INTEGER );
Table created.

gSQL>
INSERT INTO t_month VALUES( 'JANUARY' , 1 ), ( 'MARCH' , 3 ),
                          ( 'MAY' , 5 ), ( 'JULY' , 7 ),
                          ( 'SEPTEMBER' , 9 ), ( 'NOVEMBER' , 11 );
6 rows created.
```

```
gSQL>
DECLARE
  row_count INTEGER;
BEGIN
  SELECT COUNT(*) INTO row_count FROM t_month;

  DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN   = ' || SQL%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('SQL%FOUND    = ' || SQL%FOUND);
  DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND = ' || SQL%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT = ' || SQL%ROWCOUNT);
END;
/

SQL%ISOPEN   = FALSE
SQL%FOUND    = TRUE
SQL%NOTFOUND = FALSE
SQL%ROWCOUNT = 1
Anonymous PL block executed.
```

4.2 Explicit Cursor

显式游标（explicit cursor）是用户声明定义并管理的游标该游标必须由用户命名并连接到SELECT语句或DML语句

- 显式游标可如下使用
  - 用户通过open语句打开显式游标通过执行fetch语句获取查询结果最后通过close语句关闭游标
  - 显式游标可用于cursor for loop语句

显式游标不可用作expression即不可给游标分配值也不可用作procedure或routine的参数

- 显式游标示例

```
gSQL>
CREATE TABLE t_month( month_char VARCHAR(15), month_num INTEGER );
Table created.

gSQL>
INSERT INTO t_month VALUES( 'FEBRUARY' , 2 ), ( 'APRIL' , 4 ),
                          ( 'JUNE' , 6 ), ( 'AUGUST' , 8 ),
                          ( 'OCTOBER' , 10 ), ( 'DECEMBER' , 12 );
6 rows created.

gSQL>
DECLARE
  CURSOR cur_month IS SELECT * FROM t_month;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month;                                          ❶ OPEN explicit
```

```
cursor.

  LOOP                                                    ❷ 使用Loop获取多个ROW
    FETCH cur_month INTO v_month_char, v_month_num;       ❸ FETCH explicit
cursor.

    EXIT WHEN cur_month%NOTFOUND;                         ❹ 确认是否获取了结果集中的全部ROW

    DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num :
' || v_month_num );
  END LOOP;

  CLOSE cur_month;                                        ❺ CLOSE explicit
cursor.
END;
/

v_month_char : FEBRUARY , v_month_num : 2
v_month_char : APRIL , v_month_num : 4
v_month_char : JUNE , v_month_num : 6
v_month_char : AUGUST , v_month_num : 8
v_month_char : OCTOBER , v_month_num : 10
v_month_char : DECEMBER , v_month_num : 12
Anonymous PL block executed.
```

Declaring and Defining Explicit Cursors

要使用显式游标（explicit cursor）需在PSM declare section中进行声明(declaration) 和定义(definition)声明 (declaration) 显式游标时的游标名称参数信息和返回类型须与定义(definition)时的相同关于显式游标的syntax和详情请参考Explicit Cursor Declaration and Definition（显式游标声明和定义）

Explicit Cursor声明（declaration）及定义（definition）示例

```
gSQL>
DECLARE
  CURSOR cur_month1( p1 INTEGER ) RETURN t_month%rowtype;
  CURSOR cur_month1( p1 INTEGER ) RETURN t_month%rowtype IS SELECT * FROM
t_month WHERE month_num = p1;

  CURSOR cur_month2 IS SELECT * FROM t_month;
BEGIN
  NULL;
END;
/

Anonymous PL block executed.
```

Opening and Closing Explicit Cursors

- 声明（declaration）并定义（definition）显式游标（explicit cursor）后通过执行OPEN Statement执行cursor query
- 如果显式游标的查询为SELECT ... FOR UPDATE语句则对结果集的ROW加LOCK
- 如果显式游标的查询引用explicit cursor parameter或PSM variable则会影响查询结果
- 使用显式游标后执行CLOSE Statement（CLOSE 声明）关闭关闭显式游标后无法从结果集获取记录
- 显式游标OPEN后无法再次OPEN若想重新OPEN须先进行CLOSE后OPEN

Explicit Cursor OPEN和CLOSE示例

```
gSQL>
```

```
DECLARE
  CURSOR cur_month( p1 INTEGER ) IS SELECT * FROM t_month WHERE month_num = p1;
BEGIN
  OPEN cur_month( 2 );      -- Explicit Cursor OPEN

  CLOSE cur_month;          -- Explicit Cursor CLOSE
END;
/

Anonymous PL block executed.
```

Fetching Data with Explicit Cursors

- 在OPEN显式游标（explicit cursor）后可以执行FETCH Statement获取结果集的row
- Fetch statement搜索结果集的当前ROW将该ROW值保存在PSM variable或record type variable中然后将游标移动到下一个ROW
- 即使没有获取到ROWPSM FETCH Statement也不会引发异常
- 在基本LOOP语句中执行FETCH时使用%NOTFOUND属性检测终止条件

Explicit Cursor的FETCH示例

```
gSQL>
DECLARE
  CURSOR cur_month IS SELECT * FROM t_month;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month;
```

```
  LOOP
    FETCH cur_month INTO v_month_char, v_month_num;

    EXIT WHEN cur_month%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num :
' || v_month_num );
  END LOOP;

  CLOSE cur_month;
END;
/

v_month_char : FEBRUARY , v_month_num : 2
v_month_char : APRIL , v_month_num : 4
v_month_char : JUNE , v_month_num : 6
v_month_char : AUGUST , v_month_num : 8
v_month_char : OCTOBER , v_month_num : 10
v_month_char : DECEMBER , v_month_num : 12
Anonymous PL block executed.
```

When Explicit Cursor Queries Need Column Aliases

当显式游标（explicit cursor）的查询中包含表达式时该column始终需要一个别名（alias name)Fetch结果保存在引用显式游标创建的%ROWTYPE variable中可在引用时使用

在Explicit Cursor查询中使用别名示例

```
gSQL>
```

```
DECLARE
  CURSOR cur_month IS SELECT month_char, ( ' == ' || month_num ) as
equal_month_num FROM t_month;
  var cur_month%rowtype;
BEGIN
  OPEN cur_month;

  LOOP
    FETCH cur_month INTO var;

    EXIT WHEN cur_month%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE( var.month_char || var.equal_month_num );
  END LOOP;

  CLOSE cur_month;
END;
/

FEBRUARY == 2
APRIL == 4
JUNE == 6
AUGUST == 8
OCTOBER == 10
DECEMBER == 12
Anonymous PL block executed.
```

Explicit Cursors that Accept Parameters

可以声明（declaration）和定义（definition）有参数的显式游标（explicit cursor）

- 显式游标的参数
  - 仅允许使用IN bind type
  - 可以拥有DEFAULT value可以不指定Argument
  - 可以在Cursor的查询中被引用
  - 不可以在显式游标的范围外引用显式游标的参数

Explicit Cursor Parameter使用示例

```
gSQL>
DECLARE
  CURSOR cur_month( p1 IN INTEGER ) IS SELECT month_char, month_num FROM t_month
WHERE month_num = p1;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month(4);

  FETCH cur_month INTO v_month_char, v_month_num;
  DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num : '
|| v_month_num );
gSQL>
  CLOSE cur_month;
END;
/

v_month_char : APRIL , v_month_num : 4
Anonymous PL block executed.
```

- 显式游标参数的DEFAULT value示例

```
gSQL>
```

```
DECLARE
  CURSOR cur_month( p1 INTEGER DEFAULT 2 ) IS SELECT month_char, month_num FROM
t_month WHERE month_num = p1;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month;                    ❶ 可省略actual parameter.

  FETCH cur_month INTO v_month_char, v_month_num;
  DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num : '
|| v_month_num );

  CLOSE cur_month;

  OPEN cur_month(4);              ❷ 可指定actual parameter进行OPEN.

  FETCH cur_month INTO v_month_char, v_month_num;
  DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num : '
|| v_month_num );

  CLOSE cur_month;
END;
/

v_month_char : FEBRUARY , v_month_num : 2
v_month_char : APRIL , v_month_num : 4
Anonymous PL block executed.
```

Explicit Cursor Attributes

显式游标（explicit cursor）属性显示显式游标的状态可以在cursor name后添加属性用作显式游标属性

属性	返回类型	说明
explicit_cursor_name%ISOPEN	BOOLEAN	显式游标正常OPEN时为TRUE否则为FALSE
explicit_cursor_name%FOUND	BOOLEAN	Explicit cursor FETCH以前为NULL有数据时为TRUE无数据时为FALSE, CLOSE后为NULL
explicit_cursor_name%NOTFOUND	BOOLEAN	拥有与explicit_cursor_name%FOUND相反的值
explicit_cursor_name%ROWCOUNT	INTEGER	Explicit cursor OPEN以前为NULL正常OPEN时为0每次执行FETCH时增加1个CLOSE后为NULL

Table 4-2 显式游标属性

Explicit Cursor Parameter示例

- Explicit cursor OPEN前

```
gSQL>
DECLARE
  CURSOR cur_month IS SELECT month_char, month_num FROM t_month;
BEGIN
  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);
END;
/

cur_month%ISOPEN   = FALSE
```

```
cur_month%FOUND    =
cur_month%NOTFOUND =
cur_month%ROWCOUNT =
Anonymous PL block executed.
```

- Explicit cursor OPEN后

```
gSQL>
DECLARE
  CURSOR cur_month IS SELECT month_char, month_num FROM t_month;
BEGIN
  OPEN cur_month;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);

  CLOSE cur_month;
END;
/

cur_month%ISOPEN   = TRUE
cur_month%FOUND    =
cur_month%NOTFOUND =
cur_month%ROWCOUNT = 0
Anonymous PL block executed.
```

- Explicit cursor FETCH后

```
gSQL>
DECLARE
  CURSOR cur_month IS SELECT month_char, month_num FROM t_month LIMIT 3;
```

```
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month;


  DBMS_OUTPUT.PUT_LINE( '[ First Fetch ]' );

  FETCH cur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);


  DBMS_OUTPUT.PUT_LINE( '[ Second Fetch ]' );

  FETCH cur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);


  DBMS_OUTPUT.PUT_LINE( '[ Last Fetch ]' );

  FETCH cur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
```

```
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);


  DBMS_OUTPUT.PUT_LINE( '[ Over Last Fetch ]' );

  FETCH cur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);


  CLOSE cur_month;
END;
/

[ First Fetch ]
cur_month%ISOPEN   = TRUE
cur_month%FOUND    = TRUE
cur_month%NOTFOUND = FALSE
cur_month%ROWCOUNT = 1
[ Second Fetch ]
cur_month%ISOPEN   = TRUE
cur_month%FOUND    = TRUE
cur_month%NOTFOUND = FALSE
cur_month%ROWCOUNT = 2
[ Last Fetch ]
cur_month%ISOPEN   = TRUE
cur_month%FOUND    = TRUE
cur_month%NOTFOUND = FALSE
cur_month%ROWCOUNT = 3
[ Over Last Fetch ]
```

```
cur_month%ISOPEN   = TRUE
cur_month%FOUND    = FALSE
cur_month%NOTFOUND = TRUE
cur_month%ROWCOUNT = 3
Anonymous PL block executed.
```

- Explicit cursor CLOSE后

```
gSQL>
DECLARE
  CURSOR cur_month IS SELECT month_char, month_num FROM t_month;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN cur_month;

  FETCH cur_month INTO v_month_char, v_month_num;

  CLOSE cur_month;

  DBMS_OUTPUT.PUT_LINE('cur_month%ISOPEN   = ' || cur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('cur_month%FOUND    = ' || cur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%NOTFOUND = ' || cur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('cur_month%ROWCOUNT = ' || cur_month%ROWCOUNT);
END;
/

cur_month%ISOPEN   = FALSE
cur_month%FOUND    =
cur_month%NOTFOUND =
cur_month%ROWCOUNT =
Anonymous PL block executed.
```

4.3 Cursor Variable

Create Cursor Variables

若要创建游标变量（cursor variable）需声明提前定义的SYS_REFCURSOR type的variable或定义REF CURSOR TYPE声明该类型的variable详细内容参考Cursor Variable Declaration（游标变量声明）

- REF CURSOR type如下
  - Strong REF CURSOR type
    - 定义REF CURSOR type时指定return_type时为strong REF CURSOR type
    - Strong REF CURSOR type variable仅将查询集合类型与定义的return_type相同的查询连接到CURSOR VARIABLE的查询
    - 当定义的return_type相同时可以分配Strong REF CURSOR type variable
  - Weak REF CURSOR type
    - 定义REF CURSOR type时未指定return_type时为weak REF CURSOR type
    - SYS_REFCURSOR type也是weak REF CURSOR type
    - 与Strong REF CURSOR variable可更加灵活地使用
    - 在Weak REF CURSOR type variable中可以分配为SYS_REFCURSOR type variable反之也可以在SYS_REFCURSOR type variable中分配为weak REF CURSOR type variable

Create Cursor Variable示例

- REF CURSOR type定义（definition）

```
gSQL>
DECLARE
  TYPE strong_refcur IS REF CURSOR RETURN t_month%ROWTYPE;    ❶ strong ref
```

```
cursor type
  TYPE weak_refcur IS REF CURSOR;                          ❷ weak ref cursor
type
BEGIN
  NULL;
END;
/

Anonymous PL block executed.
```

- CURSOR VARIABLE声明（declaration）

```
gSQL>
DECLARE
  TYPE strong_refcur IS REF CURSOR RETURN t_month%ROWTYPE;  ❶ strong ref
cursor type
  TYPE weak_refcur IS REF CURSOR;                          ❷ weak ref cursor
type

  v_sys_refcursor     SYS_REFCURSOR;                       ❸ weak ref cursor
type variable
  v_strong_refcursor  strong_refcur;                       ❹ strong ref
cursor type variable
  v_weak_refcursor    weak_refcur;                         ❺ weak ref cursor
type variable
BEGIN
  NULL;
END;
/

Anonymous PL block executed.
```

Opening and Closing Cursor Variables

- 声明游标变量（cursor variable）后执行OPEN FOR Statement连接游标变量和要执行的SELECT语句或DML语句
- Cursor查询中可能会存在指定OPEN FOR语句的using子句中指定值的bind variable
- Cursor查询中存在FOR UPDATE语句时则对结果集的ROW加LOCK
- 对游标变量进行REOPEN时可以不进行CLOSE进行REOPEN时断开和之前查询的连接
- 游标变量使用结束后执行CLOSE Statement（CLOSE 声明）即可
- 不可以访问已CLOSE的游标变量的查询结果或引用游标变量属性

Cursor variable的OPEN和CLOSE示例

```
DECLARE
  v_refcur_month SYS_REFCURSOR;
BEGIN
  OPEN v_refcur_month FOR SELECT * FROM t_month;
  CLOSE v_refcur_month;
END;
/

Anonymous PL block executed.
```

- 使用using子句的cursor variable的OPEN示例

```
gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
  var            INTEGER;
BEGIN
```

```
  var := 1;

  OPEN v_refcur_month FOR 'SELECT * FROM t_month WHERE month_num = :v' USING IN
var;
END;
/

Anonymous PL block executed.
```

Fetching Data with Cursor Variables

在对游标变量（cursor variable）进行OPEN后可以执行FETCH Statement获取结果集的row游标变量返回的结果须与into子句兼容

Cursor Variable的FETCH示例

```
gSQL>
DECLARE
  TYPE ref_month IS REF CURSOR RETURN t_month%ROWTYPE;
  v_refcur_month ref_month;

  v_month        t_month%ROWTYPE;
BEGIN
  OPEN v_refcur_month FOR SELECT * FROM t_month;

  LOOP
    FETCH v_refcur_month INTO v_month;

    EXIT WHEN v_refcur_month%NOTFOUND;
```

```
    DBMS_OUTPUT.PUT_LINE( 'v_month.month_char : ' || v_month.month_char || ' ,
v_month.month_num : ' || v_month.month_num );
  END LOOP;

  CLOSE v_refcur_month;
END;
/

v_month.month_char : JANUARY , v_month.month_num : 1
v_month.month_char : MARCH , v_month.month_num : 3
v_month.month_char : MAY , v_month.month_num : 5
v_month.month_char : JULY , v_month.month_num : 7
v_month.month_char : SEPTEMBER , v_month.month_num : 9
v_month.month_char : NOVEMBER , v_month.month_num : 11
Anonymous PL block executed.
```

Assigning Value to Cursor Variables

游标变量（cursor variable）可以分配另一个游标变量或主机变量（host variable）值在source cursor variable为OPEN的状态下给target cursor variable分配值时两个游标变量指向相同的SQL在source cursor variable为非OPEN的状态下给target cursor variable分配值时两个游标变量都是非OPEN的状态

Assign Cursor Variable示例

```
gSQL>
DECLARE
  source_refcur_month SYS_REFCURSOR;
  target_refcur_month SYS_REFCURSOR;
```

```
  v_month_char   VARCHAR(15);
  v_month_num    INTEGER;
BEGIN

  DBMS_OUTPUT.PUT_LINE( 'OPEN source_refcur_month' );

  OPEN source_refcur_month FOR SELECT * FROM t_month;

  DBMS_OUTPUT.PUT_LINE( 'source_refcur_month%ISOPEN : ' ||
source_refcur_month%ISOPEN );
  DBMS_OUTPUT.PUT_LINE( 'target_refcur_month%ISOPEN : ' ||
target_refcur_month%ISOPEN );


  DBMS_OUTPUT.PUT_LINE( 'Assign source_refcur_month to target_refcur_month' );

  target_refcur_month := source_refcur_month;

  DBMS_OUTPUT.PUT_LINE( 'source_refcur_month%ISOPEN : ' ||
source_refcur_month%ISOPEN );
  DBMS_OUTPUT.PUT_LINE( 'target_refcur_month%ISOPEN : ' ||
target_refcur_month%ISOPEN );


  DBMS_OUTPUT.PUT_LINE( 'CLOSE source_refcur_month' );

  CLOSE source_refcur_month;

  DBMS_OUTPUT.PUT_LINE( 'source_refcur_month%ISOPEN : ' ||
source_refcur_month%ISOPEN );
  DBMS_OUTPUT.PUT_LINE( 'target_refcur_month%ISOPEN : ' ||
target_refcur_month%ISOPEN );
END;
```

```
/

OPEN source_refcur_month
source_refcur_month%ISOPEN : TRUE
target_refcur_month%ISOPEN : FALSE
Assign source_refcur_month to target_refcur_month
source_refcur_month%ISOPEN : TRUE
target_refcur_month%ISOPEN : TRUE
CLOSE source_refcur_month
source_refcur_month%ISOPEN : FALSE
target_refcur_month%ISOPEN : FALSE
Anonymous PL block executed.
```

Cursor Variable Attributes

游标变量（cursor variable）的属性与显式游标（explicit cursor）属性相同显示游标变量的状态 可以在游标变量名称后添加属性用作游标变量属性cursor_variable_name%ISOPEN属性以外的其它属性的情况在cursor variable OPEN前或CLOSE后发生'cursor is not defined'错误

属性	返回类型	说明
cursor_variable_name%ISOPEN	BOOLEAN	如果游标变量是OPEN的状态则为TRUE，否则为FALSE
cursor_variable_name%FOUND	BOOLEAN	如果FETCH后有数据则为TRUE没有则为FALSE
cursor_variable_name%NOTFOUND	BOOLEAN	FETCH后拥有与cursor_variable_name%FOUND相反的值
cursor_variable_name%ROWCOUNT	INTEGER	如果游标变量为OPEN则为0每次执行FETCH增加1个

Table 4-3 游标变量属性

Cursor Variable Attributes示例

- Cursor variable OPEN前

```
gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
BEGIN
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);
END;
/

v_refcur_month%ISOPEN   = FALSE
ERR-2F000(17036): cursor is not defined :
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
                        *
ERROR at line 5:
```

- Cursor variable OPEN后

```
gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
BEGIN
  OPEN v_refcur_month FOR SELECT * FROM t_month;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
```

```
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);

  CLOSE v_refcur_month;
END;
/

v_refcur_month%ISOPEN   = TRUE
v_refcur_month%FOUND    =
v_refcur_month%NOTFOUND =
v_refcur_month%ROWCOUNT = 0
Anonymous PL block executed.
```

- Cursor variable FETCH后

```
gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN v_refcur_month FOR SELECT * FROM t_month LIMIT 3;


  DBMS_OUTPUT.PUT_LINE( '[ First Fetch ]' );

  FETCH v_refcur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);
```

```
  DBMS_OUTPUT.PUT_LINE( '[ Second Fetch ]' );

  FETCH v_refcur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);


  DBMS_OUTPUT.PUT_LINE( '[ Last Fetch ]' );

  FETCH v_refcur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);


  DBMS_OUTPUT.PUT_LINE( '[ Over Last Fetch ]' );

  FETCH v_refcur_month INTO v_month_char, v_month_num;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);


  CLOSE v_refcur_month;
END;
/
```

```
[ First Fetch ]
v_refcur_month%ISOPEN   = TRUE
v_refcur_month%FOUND    = TRUE
v_refcur_month%NOTFOUND = FALSE
v_refcur_month%ROWCOUNT = 1
[ Second Fetch ]
v_refcur_month%ISOPEN   = TRUE
v_refcur_month%FOUND    = TRUE
v_refcur_month%NOTFOUND = FALSE
v_refcur_month%ROWCOUNT = 2
[ Last Fetch ]
v_refcur_month%ISOPEN   = TRUE
v_refcur_month%FOUND    = TRUE
v_refcur_month%NOTFOUND = FALSE
v_refcur_month%ROWCOUNT = 3
[ Over Last Fetch ]
v_refcur_month%ISOPEN   = TRUE
v_refcur_month%FOUND    = FALSE
v_refcur_month%NOTFOUND = TRUE
v_refcur_month%ROWCOUNT = 3
Anonymous PL block executed.
```

- Cursor variable CLOSE后

```
gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  OPEN v_refcur_month FOR SELECT * FROM t_month LIMIT 3;
```

```
  FETCH v_refcur_month INTO v_month_char, v_month_num;

  CLOSE v_refcur_month;

  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ISOPEN   = ' || v_refcur_month%ISOPEN);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%NOTFOUND = ' || v_refcur_month%NOTFOUND);
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%ROWCOUNT = ' || v_refcur_month%ROWCOUNT);
END;
/

gSQL>
v_refcur_month%ISOPEN   = FALSE
ERR-2F000(17036): cursor is not defined :
  DBMS_OUTPUT.PUT_LINE('v_refcur_month%FOUND    = ' || v_refcur_month%FOUND);
                        *
ERROR at line 13:
```

Cursor Variable as Routine Parameter

将游标变量（cursor variable）用作routine的参数时有助于传送查询结果若想在procedure或函数中对游标变量进行FETCH或CLOSE 参数须为IN或IN OUT type若想对游标变量进行OPEN参数须为OUT或IN OUT type在Package中声明ref cursor type有助于在不同的routine中传送游标变量的查询结果

用作Parameter的Cursor Variable示例

- 用作routine parameter的示例

```
gSQL>
```

```
CREATE OR REPLACE PROCEDURE p_open( p_refcur_month OUT SYS_REFCURSOR ) AS
BEGIN
  OPEN p_refcur_month FOR SELECT * FROM t_month;
END;
/

Procedure created.

gSQL>
CREATE OR REPLACE PROCEDURE p_fetch( p_refcur_month IN SYS_REFCURSOR ) AS
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  FETCH p_refcur_month INTO v_month_char, v_month_num;
  DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num : '
|| v_month_num );
END;
/

Procedure created.

gSQL>
DECLARE
  v_refcur_month SYS_REFCURSOR;
BEGIN
  p_open( v_refcur_month );
  DBMS_OUTPUT.PUT_LINE( 'v_refcur_month%ISOPEN : ' || v_refcur_month%ISOPEN );

  p_fetch( v_refcur_month );
  DBMS_OUTPUT.PUT_LINE( 'v_refcur_month%FOUND : ' || v_refcur_month%FOUND );

  CLOSE v_refcur_month;
  DBMS_OUTPUT.PUT_LINE( 'v_refcur_month%ISOPEN : ' || v_refcur_month%ISOPEN );
```

```
END;
/

v_refcur_month%ISOPEN : TRUE
v_month_char : JANUARY , v_month_num : 1
v_refcur_month%FOUND : TRUE
v_refcur_month%ISOPEN : FALSE
Anonymous PL block executed.
```

- Package的ref cursor type示例

```
gSQL>
CREATE OR REPLACE PACKAGE pkg_refcur AS
  TYPE ref_month IS REF CURSOR RETURN t_month%ROWTYPE;
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;

  PROCEDURE p_open( p_refcur_month OUT ref_month, p_month IN INTEGER );
  PROCEDURE p_fetch( p_refcur_month IN ref_month );
  PROCEDURE p_close( p_refcur_month IN ref_month );
END;
/

Package created.

gSQL>
CREATE OR REPLACE PACKAGE BODY pkg_refcur AS
  PROCEDURE p_open( p_refcur_month OUT ref_month, p_month IN INTEGER ) AS
  BEGIN
    IF p_month = 1 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 1;
    ELSIF p_month = 3 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 3;
```

```
    ELSIF p_month = 5 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 5;
    ELSIF p_month = 7 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 7;
    ELSIF p_month = 9 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 9;
    ELSIF p_month = 11 THEN
      OPEN p_refcur_month FOR SELECT * FROM t_month WHERE month_num = 11;
    ELSE
      DBMS_OUTPUT.PUT_LINE( 'NO ODD MONTH' );
    END IF;

    DBMS_OUTPUT.PUT_LINE( 'p_refcur_month%ISOPEN : ' || p_refcur_month%ISOPEN );
  END;

  PROCEDURE p_fetch( p_refcur_month IN ref_month ) AS
  BEGIN
    FETCH p_refcur_month INTO v_month_char, v_month_num;
    DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num :
' || v_month_num );
  END;

  PROCEDURE p_close( p_refcur_month IN ref_month ) AS
  BEGIN
    CLOSE p_refcur_month;
    DBMS_OUTPUT.PUT_LINE( 'p_refcur_month%ISOPEN : ' || p_refcur_month%ISOPEN );
  END;
END;
/

Package created.

gSQL>
```

```
DECLARE
  v_pkg_refcur_month pkg_refcur.ref_month;
BEGIN
  pkg_refcur.p_open( v_pkg_refcur_month, 5 );
  pkg_refcur.p_fetch( v_pkg_refcur_month );
  pkg_refcur.p_close( v_pkg_refcur_month );
END;
/

p_refcur_month%ISOPEN : TRUE
v_month_char : MAY , v_month_num : 5
p_refcur_month%ISOPEN : FALSE
Anonymous PL block executed.
```

Cursor Variable as Host Variable

游标变量（cursor variable）可以声明为主机变量（host variable）Host cursor variable用于传送服务器和客户端之间的查询结果服务器和客户端之间可以共享相同的查询结果集

要将游标变量用作主机变量需声明REF CURSOR type主机变量并将其传送到服务器服务器和客户端之间的游标变量调用没有限制在客户端中声明游标变量在服务器中进行OPEN后Fetch然后在客户端中FETCH后可以关闭

Host Cursor Variable示例

```
gSQL>
\var host_refcur_month REFCURSOR

gSQL>
```

```
BEGIN
  OPEN :host_refcur_month FOR SELECT * FROM t_month;
END;
/

Anonymous PL block executed.

gSQL>
DECLARE
  v_month_char VARCHAR(15);
  v_month_num  INTEGER;
BEGIN
  FETCH :host_refcur_month INTO v_month_char, v_month_num;
  DBMS_OUTPUT.PUT_LINE( 'v_month_char : ' || v_month_char || ' , v_month_num : '
|| v_month_num );
END;
/

v_month_char : JANUARY , v_month_num : 1
Anonymous PL block executed.

gSQL>
BEGIN
  CLOSE :host_refcur_month;
END;
/

Anonymous PL block executed.
```

5. Using PSM Subprograms

5.1 Anonymous PL Block

Anonymous PL block是不将PSM语句储存在数据库中并且一次性执行的SQL语句是SUNDB提供的正规SQL之一因此在SUNDB提供的ODBCJDBC或precompiler中也可与其他SQL一样使用语法与普通basic block语句相同详细内容参考块(BEGIN .. END)

```
<<Label>> ❶ (optional)
DECLARE    ❷ (optional)
❸ declare items (variables, cursors, types, ...) (optional)
BEGIN      ❹ (required)
❺ PSM statements to execute (required)
EXCEPTION ❻ Exception Handling Part (optional)
END;
```

使用anonymous PL block时应注意以下几点:

Interface	使用注意事项
通用	与模式级过程（Schema-level Procedure）或函数不同可以使用绑定参数（Bind Parameter） 也支持Prepare-Execute方式
ODBC	与常规SQL的使用方法完全相同
JDBC	有In OUT或OUT属性的绑定参数时应使用CallableStatement类的语句
precompiler (gpec)	无特殊事项
Interactive Command Tool (gsql)	输入anonymous PL block后应输入'/'< Enter >结束语句

5.2 Nested Procedure

嵌套过程（nested procedure）是在特定PL block中声明的过程类型子程序嵌套过程包含声明的PL block与仅可在其下层引用的scope

详细内容参考Procedure Declaration and Definition

```
DECLARE
  PROCEDURE PROC1( A1 INTEGER )  ❶ Define nested procedure
    IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE( 'A1 = ' || A1 );
    END;
BEGIN
  PROC1( 100 );  ❷ Call nested procedure
END;
/
```

嵌套子程序（nested subprogram）中可使用的item如下

- 嵌套子程序的参数（argument）变量
- 定义嵌套子程序的PL block与其上层scope中定义的变量和各种item（type, cursor,...）
- 为Anonymous PL block时bind parameter(例: '?', ':V1' 等)

嵌套过程支持forward declaration因此各单独描述declare和define使用前向声明可实现两个嵌套过程之间相互调用的逻辑

```
gSQL> DECLARE

  PROCEDURE PROC1( A1 INTEGER );  ❶ declare proc1
```

```
  PROCEDURE PROC2( A1 INTEGER )   ❷ define  proc2
  IS
  BEGIN
    IF A1 > 0 THEN
      DBMS_OUTPUT.PUT_LINE( '(proc2)A1 = ' || A1 );
      PROC1( A1 -1 );  ❸ call proc1
    END IF;
  END;

  PROCEDURE PROC1( A1 INTEGER )   ❹ define  proc1
  IS
  BEGIN
    IF A1 > 0 THEN
      DBMS_OUTPUT.PUT_LINE( '(proc1)A1 = ' || A1 );
      PROC2( A1 -1 );  ❺ call proc2
    END IF;
  END;

BEGIN
  PROC1(5);  ❻ call proc1
END;
/

(proc1)A1 = 5
(proc2)A1 = 4
(proc1)A1 = 3
(proc2)A1 = 2
(proc1)A1 = 1

Anonymous PL block executed.
```

SUNDB PSM将最大子语句深度（child statement depth）限制为50以防止无限交叉引用如果超过此值将如下报错（同样适用于nested functionschema-level procedure/ function）

```
gSQL> DECLARE
  PROCEDURE PROC1( A1 INTEGER );  ❶ declare proc1
  PROCEDURE PROC2( A1 INTEGER )   ❷ define  proc2
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'A1 = ' || A1 );
    PROC1( A1 -1 );
  END;
  PROCEDURE PROC1( A1 INTEGER )   ❸ define  proc1
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'A1 = ' || A1 );
    PROC2( A1 -1 );
  END;
BEGIN
  PROC1(100);
END;
/

ERR-42000(16411): maximum number of recursive SQL levels (50) exceeded.
```

5.3 Nested Function

嵌套函数（nested function）虽与嵌套过程相同但为具有函数类型的子程序

详细内容参考Function Declaration and Definition（函数声明和定义）

```
gSQL> DECLARE
  V1 INTEGER := 0;
  FUNCTION FUNC1( A1 INTEGER )
    RETURN INTEGER
    IS
    BEGIN
      RETURN A1 * 10;
    END;
BEGIN
  V1 := FUNC1( 10 );
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/

V1 = 100

Anonymous PL block executed.
```

嵌套函数可在可见范围内的所有PSM表达式中使用但不能在SQL语句中使用

```
gSQL> DECLARE
  V1 INTEGER := 0;
  FUNCTION FUNC1( A1 INTEGER )
    RETURN INTEGER
    IS
    BEGIN
```

```
      RETURN A1 * 10;
    END;
BEGIN
  SELECT FUNC1(10) INTO V1 FROM DUAL;
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/

ERR-HY000(17032): PSM compilation error :
(1) at (10:3): ERR-HY000(17079): a nested function not allowed in executing SQL
```

5.4 Schema-level Procedure

模式级子程序（schema-level subprogram）是存储在数据库的拥有名称的SQL对象Schema-level procedure是过程形式的模式级子程序

创建

Schema-level procedure通过以下语句创建必要时应在参数类型指定precision与scale值

详细内容参考CREATE PROCEDURE

```
CREATE OR REPLACE PROCEDURE PROC1( A1 INTEGER, A2 INTEGER )
IS
  V1 INTEGER;
BEGIN
  SELECT COUNT(*)
    INTO V1
    FROM T1
    WHERE T1.I1 >= A1 AND T1.I1 <= A2;
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/
```

创建的Schema-level Procedure的信息可通过INFORMATION_SCHEMA.ROUTINES表进行查看

详细内容参考ROUTINES

```
gSQL> SELECT SPECIFIC_NAME, ROUTINE_DEFINITION
  FROM  INFORMATION_SCHEMA.ROUTINES
  WHERE SPECIFIC_NAME = 'PROC1';
```

```
SPECIFIC_NAME ROUTINE_DEFINITION
------------- --------------------------------------------------
PROC1         PROCEDURE "PUBLIC"."PROC1" ( A1 INTEGER, A2 INTEGER )
              IS
                V1 INTEGER;
              BEGIN
                SELECT COUNT(*)
                  INTO V1
                  FROM T1
                  WHERE T1.I1 >= A1 AND T1.I1 <= A2;
                DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
              END;


1 row selected.
```

参数（argument）信息可通过INFORMATION_SCHEMA.PARAMETERS表进行查看

详细内容参考PARAMETERS

```
gSQL> SELECT P.PARAMETER_NAME, P.ORDINAL_POSITION
  FROM  INFORMATION_SCHEMA.ROUTINES     R,
        INFORMATION_SCHEMA.PARAMETERS   P
  WHERE R.SPECIFIC_NAME = 'PROC1'
    AND R.SPECIFIC_SCHEMA = P.SPECIFIC_SCHEMA
    AND R.SPECIFIC_NAME = P.SPECIFIC_NAME
  ORDER BY P.ORDINAL_POSITION;

PARAMETER_NAME ORDINAL_POSITION
-------------- ----------------
A1                            1
A2                            2
```

```
2 rows selected.
```

使用

其他PSM内部语句或CALL语句使用模式级过程（schema-level procedure）

其他PSM语句以与嵌套过程相同的格式调用模式级过程

```
gSQL> BEGIN
  PROC1( 2, 4 ); -- call schema-level procedure
END;
/

V1 = 3

Anonymous PL block executed.
```

CALL语句是执行给定PSM的SQL其执行方式如下

详细内容参考CALL Statement

```
gSQL> CALL PROC1( 2, 4 );
V1 = 3

Procedure Call complete.
```

为了支持ODBC或JDBC中使用的procedure call escape sequence对过程支持以下语句

```
{ CALL procedure_name( param1, param2, ... ) }
```

Procedure call escape sequence语句可像常规SQL一样使用

```
gSQL> { CALL PROC1(2, 4) };
V1 = 3


Procedure Call complete.
```

SUNDB的交互命令工具（interactive command tool）GSQL不支持通过工具自身命令/EXEC的执行过程

执行schema-level procedure时可指定以下权限相关选项当非definer的用户根据此选项执行PSM中的SQL时根据该用户的schema-path定义也可以引用具有不同schema的相同名称的表（声明item时使用的对象名（例: T1%ROWTYPE）始终被解析为definer）

- AUTHID DEFINER（默认）：变更为创建该procedure的用户后执行
- AuthIDCurrngUsAudio：不变更执行用户由当前用户执行

```
CREATE OR REPLACE PROCEDURE "PROC1"( A1 INTEGER, A2 INTEGER )
AUTHID CURRENT_USER
IS
  V1 INTEGER;
BEGIN
  SELECT COUNT(*)
    INTO V1
    FROM T1
    WHERE T1.I1 >= A1 AND T1.I1 <= A2;
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/
```

删除

Schema-level procedure通过以下DROP PROCEDURE语句删除

详细内容参考DROP PROCEDURE

```
DROP PROCEDURE PROC1;
```

与SUNDB中的其他DROP语句相同也支持IF EXISTS语句

```
DROP PROCEDURE IF EXISTS PROC1;
```

重新编译（Recompile）

在schema-level subprogram内部引用的对象发生更改时对应的子程序也会受到影响可能需要重新创建执行计划

在声明部分或参数中引用的对象

定义声明部分的各种Item或参数类型时若使用的对象发生更改则自动重新编译过程的计划

- %TYPE%ROWTYPE中使用的对象
- 显式游标（Explicit Cursor）定义语句中使用的对象

对象变更后首次执行该过程时根据以下顺序自动重新创建计划

1. 从计划缓存中获取该过程的计划
2. 验证对应计划的对象列表的过程中查找更改的对象
3. Discard当前计划并从字典中存储的过程定义语句开始创建新的计划
4. 在计划缓存中注册新创建的计划
5. 执行计划

Body的SQL中引用的对象

过程的计划中不存储body中使用的SQL计划是仅存储SQL text的状态执行过程时实时编译对应的SQL语句并生成计划后执行因此body中使用的SQL对象不会影响过程计划本身

但是如果对象发生了更改可能会变更与过程的现有接口的绑定数量及类型或删除column等因此如果不适当变更过程在执行时可能发生错误

5.5 Schema-level Function

Schema-level function是表达式中使用的函数类型的schema-level subprogram

创建

如下创建schema-level function必要时应在参数的类型指定precision与Scale值

详细内容参考CREATE FUNCTION

```
gSQL> CREATE OR REPLACE FUNCTION FUNC1( A1 INTEGER, A2 INTEGER )
RETURN INTEGER
IS
  V1 INTEGER;
BEGIN
  SELECT COUNT(*)
    INTO V1
    FROM T1
    WHERE T1.I1 >= A1 AND T1.I1 <= A2;
  RETURN V1;
END;
/

Function created.
```

创建的schema-level function与过程相同可在 INFORMATION_SCHEMA.ROUTINES与INFORMATION_SCHEMA.PARAMETERS表中查看

使用

Schema-level function可在使用常规SQL或PSM中的SQL对象的所有表达式中使用

```
gSQL> SELECT FUNC1( 2, 4 ) FROM DUAL;

FUNC1( 2, 4 )
-------------
            3

1 row selected.
```

如下也可在PSM中使用

```
gSQL> DECLARE
  V1 INTEGER;
BEGIN
  V1 := FUNC1( 2, 4 );
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/

V1 = 3

Anonymous PL block executed.
```

如下可使用CALL语句执行

详细内容参考CALL Statement

```
gSQL> \var V1 INTEGER
```

```
gSQL> CALL FUNC1( 2, 4 ) INTO :V1;

Procedure Call complete.

gSQL> \print V1

V1
--
 3
```

与schema-level procedure相同也支持procedure call escape sequence语句

```
{ ? = CALL function_name( param1, param2, ... ) }
```

```
gSQL> { :V1 = CALL FUNC1( 2, 4 ) };

Procedure Call complete.

gSQL> \print V1

V1
--
 3
```

删除

Schema-level Function通过DROP FUNCTION语句删除

详细内容参考DROP FUNCTION

```
gSQL> DROP FUNCTION FUNC1;

Function dropped.
```

5.6 Built-in Procedures

SUNDB PSM提供如下在实现过程及函数时用于调试（debugging）或处理异常（exception）的built-in procedure

Procedure	function
DBMS_OUTPUT.ENABLE(buffer_size IN NATIVE_INTEGER := 20000)	以指定的缓冲区大小激活消息日志记录功能 如果未指定缓冲区大小则默认为20000字节如果当前缓冲区已被激活则丢弃所有消息并创建新的缓冲区
DBMS_OUTPUT.DISABLE	禁用消息日志记录功能所有已记录的消息都将被丢弃
DBMS_OUTPUT.SET_LOG(file_path IN VARCHAR(4000))	记录消息时同时输出到指定路径中的文件如果指定为相对路径（第一个字符不是'/'的情况）则可在<SYSTEM_LOGGER_DIR>参数中的对应目录下找到目标文件 如下可指定permission • CALL DBMS_OUTPUT.SET_LOG('a.txt' , 640)
DBMS_OUTPUT.PUT_LINE(item IN VARCHAR(4000))	将通过指定表达式创建的消息包含新行字符存储到缓冲区
DBMS_OUTPUT.PUT(item IN VARCHAR(4000))	将通过指定表达式创建的消息存储到缓冲区
DBMS_OUTPUT.NEW_LINE	将新行字符存储到缓冲区
DBMS_OUTPUT.GET_LINE(line OUT VARCHAR(4000), status OUT NATIVE_INTEGER)	返回缓冲区中存储的消息中尚未读取的最早的一条消息如果存在该消息则status返回'0'反之返回'1'
DBMS_STANDARD.RAISE_APPLICATION_ERROR(error_code IN NATIVE_INTEGER, error_message IN VARCHAR(4000), stack_flag IN BOOLEAN := FALSE)	触发任意用户异常error_code应为-20000～-20999之间的值 TRUE时最后的stack_flag参数在已有的错误上累积该错误为FALSE时以该错误代替（replace）所有错误（可省略此时default为FALSE）
DBMS_LOCK.SLEEP(seconds IN NUMBER)	在指定时间内暂时终止会话

Procedure	function
DBMS_SQL.RETURN_RESULT(rc IN SYS_REFCURSOR)	将通过ref cursor执行的查询结果返回给客户端应用程序

Table 5-1 Built-in procedures

```
gSQL> DECLARE
V1 VARCHAR(1024);
V2 INTEGER;
BEGIN
  DBMS_OUTPUT.ENABLE(2000);
  DBMS_OUTPUT.PUT_LINE('TEST MSG');
  DBMS_OUTPUT.GET_LINE( V1, V2);
  DBMS_OUTPUT.PUT_LINE('V1 = ' || v1);
  DBMS_OUTPUT.PUT_LINE('V2 = ' || v2);
END;
/

V1 = TEST MSG

V2 = 0

Anonymous PL block executed.
```

消息日志记录功能以会话为单位进行管理可通过SUNDB的interactive command tool gsql中的 serveroutput 选项激活或禁用消息日志记录功能执行PSM后如果消息缓冲区中有内容则自动输出所有内容

```
gSQL> \set serveroutput on
gSQL> \var msg VARCHAR(4000)
gSQL> \var status NATIVE_INTEGER
gSQL> CALL DBMS_OUTPUT.PUT_LINE( 'aaa' );
aaa
```

```
Procedure Call complete.

gSQL> CALL DBMS_OUTPUT.PUT_LINE( 'bbb' );
bbb

Procedure Call complete.

gSQL> CALL DBMS_OUTPUT.GET_LINE( :msg, :status );

Procedure Call complete.

gSQL> \print msg

MSG
----
null

gSQL> \print status

STATUS
------
     1
```

6. Using SQLs in PSM

6.1 Static SQLs

概要

静态SQL是为使用SUNDB中支持的PSM变量扩展的SQL

可在SQL中所有可以使用bind parameter的表达式中使用PSM变量

- SELECT INTO Statement
  - 详细内容参考PSM Language Element References的SELECT INTO Statement
- Data Manipulation Language(DML)
  - INSERT Statement Extension
    - 详细内容参考PSM Language Element References的INSERT Statement Extension
  - INSERT INTO ... UPDATE Statement Extension
    - 详细内容参考PSM Language Element References的INSERT INTO ... UPDATE Statement Extension
  - UPDATE Statement Extension
    - 详细内容参考PSM Language Element References的UPDATE Statement Extension
  - DELETE Statement Extension
    - 详细内容参考PSM Language Element References的DELETE Statement Extension（DELETE 语句扩展）
- Transaction Control Language
  - COMMIT
    - 详细内容参考SQL References的COMMIT
  - ROLLBACK

        - 详细内容参考SQL References的ROLLBACK
    - SAVEPOINT
        - 详细内容参考SQL References的SAVEPOINT savepoint_specifier
    - LOCK TABLE
        - 详细内容参考SQL References的LOCK TABLE

使用示例

- 使用built-in data type的PSM变量时

```
gSQL>
DECLARE
  v_id      NUMBER;
  v_name    VARCHAR(50);
BEGIN
  -- SELELECT INTO Statement
  SELECT id , name
    INTO v_id , v_name
    FROM emp
   WHERE id = 201;

   DBMS_OUTPUT.PUT_LINE( '[SELECT INTO] ' || v_id || ' , ' || v_name );

  -- INSERT Statement Extension
  v_id   := 200;
  v_name := 'Jennifer Whalen';
  INSERT INTO emp VALUES( v_id , v_name );

  -- DELETE Statement Extension
  DELETE FROM emp
   WHERE id = v_id
```

```
  RETURNING name INTO v_name;

  DBMS_OUTPUT.PUT_LINE( '[DELETE] ' || v_id || ' , ' || v_name );

  -- UPDATE Statement Extension
  UPDATE emp
     SET id = v_id , name = v_name
   WHERE id = 200;

  -- Commit
  COMMIT;
END;
/

[SELECT INTO] 201 , Michael Hartstein
[DELETE] 200 , Jennifer Whalen
Anonymous PL block executed.
```

- 使用ROW type的PSM变量时

```
gSQL>
DECLARE
  TYPE rec_emp IS RECORD( f_id emp.id%TYPE , f_name emp.name%TYPE );
  v_rec_emp rec_emp;

BEGIN
  -- SELELECT INTO Statement
  SELECT id , name
    INTO v_rec_emp
    FROM emp
   WHERE id = 201;

   DBMS_OUTPUT.PUT_LINE( '[SELECT INTO] ' || v_rec_emp.f_id ||
```

```
                        ' , ' || v_rec_emp.f_name );

  -- INSERT Statement Extension
  v_rec_emp.f_id   := 200;
  v_rec_emp.f_name := 'Jennifer Whalen';
  INSERT INTO emp VALUES( v_rec_emp.f_id , v_rec_emp.f_name );

  -- DELETE Statement Extension
  DELETE FROM emp
   WHERE id = v_rec_emp.f_id
  RETURNING * INTO v_rec_emp;

   DBMS_OUTPUT.PUT_LINE( '[DELETE] ' || v_rec_emp.f_id ||
                        ' , ' || v_rec_emp.f_name );

  -- UPDATE Statement Extension
  UPDATE emp
     SET ROW = v_rec_emp
   WHERE id = 200;

  -- Commit
  COMMIT;
END;
/

[SELECT INTO] 201 , Michael Hartstein
[DELETE] 200 , Jennifer Whalen
Anonymous PL block executed.
```

Processing Query Result Sets

PSM中使用隐式游标（ implicit cursor）或显式游标（ explicit cursor）处理结果集

PSM如下定义隐式游标

- SELECT INTO
- Implicit Cursor FOR LOOP

PSM如下定义显式游标

- Explicit Cursor FOR LOOP
  - 用户定义的显式游标可在执行PSM语句期间使用

Processing Query Result Sets with SELECT INTO Statements

使用隐式游标通过执行SELECT INTO statement检索值并将其保存到PSM变量

Select into statement的结果集总是单行的

使用示例

```
gSQL>
DECLARE
  v_id   emp.id%TYPE;
  v_name emp.name%TYPE;
BEGIN
  SELECT id , name
    INTO v_id , v_name
    FROM emp
```

```
   WHERE id = 201;


   DBMS_OUTPUT.PUT_LINE( 'SQL%FOUND = ' || SQL%FOUND );
END;
/

SQL%FOUND = TRUE
Anonymous PL block executed.
```

Processing Query Result Sets with Cursor FOR LOOP Statements

Cursor For LOOP statement执行隐式游标和显式游标然后重复返回结果集的行

使用SELECT语句的cursor FOR LOOP statement称为implicit cursor FOR LOOP statementImplicit cursor FOR LOOP statement使用select statement所需的隐式游标返回结果集的行

Cursor FOR LOOP statement可以使用用户声明的显式游标
也可以在PSM块的其它语句中使用用户声明的显式游标
Cursor FOR LOOP statement隐式创建并使用游标返回的loop index类型的%ROWTYPE变量
Loop index是仅在cursor FOR LOOP statement执行期间才可以使用的变量
在Loop中执行的PSM statement可以使用loop index引用记录和字段
在创建loop index变量后打开用户指定的游标执行Cursor FOR LOOP statement
每次重复Loop时将行结果保存到loop index变量中
没有再要返回的行时关闭游标另外执行中发生例外时也会关闭游标

使用示例

- 在cursor FOR LOOP statement中使用SELECT statement时

```
gSQL>
BEGIN
```

```
  FOR tmp IN ( SELECT id , name , manager_id FROM emp ) LOOP
    DBMS_OUTPUT.PUT_LINE( 'id = ' || tmp.id ||
                          ' , name = ' || tmp.name ||
                          ' , manager_id = ' || tmp.manager_id );
  END LOOP;


END;
/

id = 200 , name = Jennifer Whalen   , manager_id = 101
id = 201 , name = Michael Hartstein , manager_id = 101
id = 202 , name = Pat Fay           , manager_id = 301
id = 203 , name = Susan Mavris      , manager_id = 201
id = 204 , name = Hermann Baer      , manager_id = 201
id = 205 , name = Shelley Higgins   , manager_id = 301
id = 206 , name = William Gietz     , manager_id = 201
Anonymous PL block executed.
```

- 在cursor FOR LOOP statement中使用显式游标时
  - 显式游标中没有参数时

```
gSQL>
DECLARE
  CURSOR cur1 IS SELECT id , name , manager_id FROM emp;
BEGIN
  FOR tmp IN cur1 LOOP
    DBMS_OUTPUT.PUT_LINE( 'id = ' || tmp.id ||
                          ' , name = ' || tmp.name ||
                          ' , manager_id = ' || tmp.manager_id );
  END LOOP;
END;
/
```

```
id = 200 , name = Jennifer Whalen   , manager_id = 101
id = 201 , name = Michael Hartstein , manager_id = 101
id = 202 , name = Pat Fay           , manager_id = 301
id = 203 , name = Susan Mavris      , manager_id = 201
id = 204 , name = Hermann Baer      , manager_id = 201
id = 205 , name = Shelley Higgins   , manager_id = 301
id = 206 , name = William Gietz     , manager_id = 201
Anonymous PL block executed.
```

- 显式游标中有参数时

```
gSQL>
DECLARE
  CURSOR cur1( p1 NUMBER ) IS SELECT id , name , manager_id
                                FROM emp
                               WHERE manager_id = p1;
BEGIN
  FOR tmp IN cur1( 201 ) LOOP
    DBMS_OUTPUT.PUT_LINE( 'id = ' || tmp.idgSQL>  ||
                          ' , name = ' || tmp.name ||
                          ' , manager_id = ' || tmp.manager_id );
  END LOOP;
END;
/

id = 203 , name = Susan Mavris      , manager_id = 201
id = 204 , name = Hermann Baer      , manager_id = 201
id = 206 , name = William Gietz     , manager_id = 201
Anonymous PL block executed.
```

Processing Query Result Sets with Explicit Cursors, OPEN, FETCH, and

CLOSE

声明并使用显式游标来按照需要控制结果集

声明显式游标后用户可以使用OPENFETCH和CLOSE statement管理结果集

这样使用PL statement的查询虽然看起来复杂但是具有可以如下灵活管理结果集的优点

- 可以使用多个显式游标并行处理结果集
- 可以在单个loop statement中并行处理结果集的多个行或跳过特定行
- 另外还可以使用多个loop statement划分结果集进行处理

详细内容参考Explicit Cursor

6.2 Dynamic SQL

与静态SQL不同动态SQL在执行时确定syntax

PSM可以通过EXECUTE IMMEDIATE或OPEN FOR语句在运行时执行用户创建的动态SQL

- 动态SQL用于以下情况
  - 在编译时无法确定SQL语句的情况（例如SQL语句需要根据条件发生变化的情况）
  - 需要执行静态SQL不支持的SQL的情况（例如DDL）
- 动态SQL在执行之前无法确定syntax因此根据语法错误目标对象的存在与否用户权限等验证过程可能会在运行过程中产生错误
- 可以在以下PSM语句中使用dynamic SQL
  - EXECUTE IMMEDIATE Statement（EXECUTE IMMEDIATE声明）
  - OPEN FOR Statement

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE可执行多种动态SQL操作 但是不能使用PSM提供的SQL扩展形式的语句

提供如下形式的语句

```
EXECUTE IMMEDIATE 'dynamic sql' [ USING [IN | OUT | INOUT] variable_list] [INTO variable_list] [RETURNING INTO variable_list]
```

可以使用EXECUTE IMMEDIATE语句提供的USINGINTORETURNING INTO语句将存储在PSM变量中的值应用于数据库或将数据库中的值存储到PSM变量

各个语句的使用方法有如下区别

- USING
  - 将PSM变量的值应用于SQL时使用IN-mode
  - 将SQL的处理结果保存到PSM变量时使用OUT-mode
    - 因此在以OUT-mode使用时必须描述为PSM变量 （不能使用表达式）
  - 如果未指定USING子句中描述的变量的绑定类型适用IN-mode
  - 以USING OUT返回SELECT的target时运行方式与使用SELECT INTO子句相同
  - 只能从数据库接收单行如果出现2条以上的结果将报错
  - USING IN子句中的PSM变量只能以scalar-type形式使用
  - USING OUT子句中的PSM变量可以是record-type但不能与其他类型混合使用
- INTO
  - 用于存储在数据库内部使用隐式游标处理的SELECT结果
  - 只有在以动态SQL的方式执行SELECT子句时使用（无法使用SELECT INTO子句）
  - PSM变量可以是record-type但不能与其他类型混合使用
- RETURING INTO
  - 用于存储通过INSERT / UPDATE / DELETE处理的数据的之前/之后
  - 只能保存单行
  - PSM变量可以是record-type但不能与其他类型混合使用

以下为使用动态SQL输出SELECT_INTO语句的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.


gSQL> INSERT INTO T1 VALUES ('Seoul', '24');
```

```
1 row created.

gSQL> INSERT INTO T1 VALUES ('Pusan', '44');

1 row created.


gSQL> DECLARE
  V1 VARCHAR(20);
  V2 VARCHAR(20);
BEGIN
  EXECUTE IMMEDIATE 'SELECT * INTO ?, ? FROM T1 WHERE C1 = ''Seoul'''
  USING OUT V1, OUT V2;

  DBMS_OUTPUT.PUT_LINE('V1 = ' || V1);
  DBMS_OUTPUT.PUT_LINE('V2 = ' || V2);
END;
/
V1 = Seoul
V2 = 24

Anonymous PL block executed.
```

以下是执行更新运算并将更新前的结果存储为RETURNING INTO中描述的变量的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.
```

```
gSQL> INSERT INTO T1 VALUES ('Seoul', '24');

1 row created.

gSQL> INSERT INTO T1 VALUES ('Pusan', '44');

1 row created.



gSQL> DECLARE
  V1 VARCHAR(20);
  V2 VARCHAR(20);
  V3 VARCHAR(20);
  V4 VARCHAR(20);
BEGIN
  V1 := 'Daegu';
  V2 := '50';

  EXECUTE IMMEDIATE
      'UPDATE T1 SET C1 = ? ,  C2 = ? WHERE C1 = ''Seoul'' RETURNING OLD *
INTO ?, ?'
  USING V1, V2 RETURNING INTO V3, V4;

  DBMS_OUTPUT.PUT_LINE('V3 = ' || V3);
  DBMS_OUTPUT.PUT_LINE('V4 = ' || V4);
END;
/
V3 = Seoul
V4 = 24

Anonymous PL block executed.
```

详细内容参考 EXECUTE IMMEDIATE Statement（EXECUTE IMMEDIATE声明）

OPEN FOR, FETCH and CLOSE

使用EXECUTE IMMEDIATE处理查询时不能从数据库返回一个或多个 如果要处理的SQL是动态SQL并且需要获取两个以上的结果集（例如游标）则可以使用OPEN FOR

可以以如下形式对OPEN FOR使用动态SQL

```
OPEN Cursor_variable FOR dynamic_sql [USING variable_list]
```

以下为通过dynmaic SQL执行OPEN FOR的示例

```
gSQL> CREATE TABLE T1
(
  C1 VARCHAR(20),
  C2 VARCHAR(20)
);

Table created.


gSQL> INSERT INTO T1 VALUES ('Seoul', '24');

1 row created.

gSQL> INSERT INTO T1 VALUES ('Pusan', '44');

1 row created.
```

```
gSQL> DECLARE
  v1 VARCHAR(20);
  v2 VARCHAR(20);
  v3 VARCHAR(20);

  cv1 SYS_REFCURSOR;
  sqlstr VARCHAR(1024);
BEGIN

    sqlstr := 'SELECT * FROM T1 WHERE C1 >= ?';

    v3 := 'AAAA';
    OPEN cv1 FOR sqlstr USING v3;

    FETCH cv1 INTO v1, v2;

    DBMS_OUTPUT.PUT_LINE('V1 = ' || V1 || ' , V2 = ' || V2);

    CLOSE cv1;
END;
/
V1 = Seoul , V2 = 24

Anonymous PL block executed.
```

- OPEN FOR statement的USING子句有以下约束
  - Binding mode
    - 仅限IN mode
  - 可用项
    - PSM变量
    - 以标量类型返回结果的值表达式

详细内容参考OPEN FOR Statement, FETCH Statement, CLOSE Statement（CLOSE 声明）

7. PSM Packages

本章介绍使用 package 对多个应用程序通用的数据和 PSM 代码进行模块化的方法

7.1 定义

Package是将有逻辑关系的PSM类型变量subprogram游标异常等项目捆绑在一起的schema对象Package经过编译过程存储到数据库从而使其他程序（其他packageprocedure外部程序等）能够引用共享执行package项目

为了能够引用所有package中有描述公开的多个item和subprogram的目录的specification

如果公开的item中有游标或公开的subprogram时对应package可以拥有package bodyBody包含如下内容

- 已公开的游标要执行的SQL语句或已公开的subprogram要执行的PSM code
- 不公开并且仅在内部使用的多个变量类型等item或subprogram
- 第一次使用package时（instantiation）仅执行一次的公开item初始化code (Initialization part)

Package specification的AUTHID语句指定以invoker权限执行对应package还是以definer权限执行（Default：definer）并指定引用unqualified对象时以invoker解析还是以definer解析

只有相应package的所有者和授权EXECUTE PACKAGE权限的用户才能检索package specification定义语句而package body定义语句只有相应package的所有者才能检索

7.2 优点

Package的以下功能向应用程序开发人员和运营人员提供很高的信任度和重复使用性

- 模块化功能
  - 将相关item和subprogram捆绑在一起因此更便于管理数据库并理解其结构
- 程序设计的便利性
  - 定义specification后即使不定义body也可以设计相关应用程序
- 封装
  - 可以隐藏package内部实现事项body的内容使package用户无法查看
  - 不需要重新编译使用相应package的SQL和应用程序也可以变更body
- 性能优化
  - 整个package一次性加载到内存因此可快速调用内存中的subprogram
- 权限管理的便利性
  - 可以将package中包含的所有公开item和subprogram的各个使用权限捆绑成一个对整个package的权限

Package不支持subprogram overloading因此无法在一个block scope内定义名称相同的函数

7.3 Specification

以下为可在package specification声明的公开item

- 用户自定义类型
  - Record or collection (Associative array)

  - 在package内声明的用户自定义类型仅可在PSM系列statement中使用
- 变量
  - 为了防止直接引用变量建议对各个变量定义get/ set subprogram
- Subprogram
  - Procedure
  - 函数 (Function)
- 显式游标 (Explicit cursor)
  - 可以在package specification仅定义名称和返回类型后在body定义游标要执行的SQL语句
  - 游标变量 (cursor variable)无法定义为package的公开item
- 用户自定义异常(Exception)

创建Package Specification

使用CREATE PACKAGE语句创建package specification

```
CREATE OR REPLACE PACKAGE MY_PKG
IS
  TYPE MY_REC IS RECORD (F1 INTEGER, F2 INTEGER );
END;
/

Package created.
```

使已定义的package的公开item可如下像package名称一样指定后在package外部引用

```
DECLARE
V1 MY_PKG.MY_REC;
BEGIN
    V1.F1 := 10;
    V1.F2 := 20;
```

```
    DBMS_OUTPUT.PUT_LINE( 'V1.F1 = ' || V1.F1 || ', V1.F2 = ' || V1.F2 );
END;
/
V1.F1 = 10, V1.F2 = 20
Anonymous PL block executed.
```

创建Package Body

Package有公开的游标或subprogram或需要初始化代码时可能需要声明body其他情况选择性地声明body

Package body有以下约束事项

- Package body要与package specification生成在相同的schema
- Package body要与package specification拥有相同的名称
- 在package specification声明的公开游标中如有未指定SQL语句的游标时必须在与其拥有相同名称记录类型参数的游标指定SQL后在body定义
- 与在package specification声明的公开subprogram拥有相同名称参数返回类型的subprogram应在body定义
- Package body中无法声明与在package specification定义的变量类型异常相同名称的变量类型异常

如下创建package body

```
CREATE TABLE emp( empno NUMBER, sal NUMBER, comm NUMBER );
Table created.

INSERT INTO emp VALUES( 3548, 6000, 1000 );
1 row created.

INSERT INTO emp VALUES( 9369, 5000, NULL );
1 row created.
```

```
INSERT INTO emp VALUES( 7294, 4000, 500 );
1 row created.

COMMIT;
Commit complete.


CREATE OR REPLACE PACKAGE emp_mgmt
IS
  PROCEDURE adjust_sal(v_flag VARCHAR, v_empno NUMBER, v_pct NUMBER);
  FUNCTION get_annual_sal(v_empno NUMBER) RETURN NUMBER;
END;
/

Package created.


CREATE OR REPLACE PACKAGE BODY emp_mgmt
IS
  PROCEDURE adjust_sal(v_flag VARCHAR, v_empno NUMBER, v_pct NUMBER) IS
  BEGIN
    IF v_flag = 'INCREASE' THEN
      UPDATE emp SET sal = sal + (sal * (v_pct / 100)) WHERE empno = v_empno;
    ELSE
      UPDATE emp SET sal = sal - (sal * (v_pct / 100)) WHERE empno = v_empno;
    END IF;
  END;
  FUNCTION get_annual_sal (v_empno NUMBER) RETURN NUMBER
  IS
    v_sal NUMBER;
  BEGIN
    SELECT (sal + NVL(comm,0)) * 12 INTO v_sal FROM emp WHERE empno = v_empno;
```

```
    RETURN v_sal;
  END;
END;
/

Package created.
```

可如下调用创建的package body的subprogram

```
call emp_mgmt.adjust_sal('INCREASE',7369, 10);

Procedure Call complete.


SELECT emp_mgmt.get_annual_sal(7294) FROM DUAL;
EMP_MGMT.GET_ANNUAL_SAL(7294)
-----------------------------
                        54000
1 row selected.
```

Package状态

Package分为stateful package和stateless package

Stateful package是由一个以上的变量或者游标组成的packagestateless package是仅由procedure或者function等subprogram组成的package

Stateful package在第一次引用package的变量或者游标时在会话上创建package instance此时执行以下操作

- Package变量初始化

- 执行initialize section

一般情况下package状态与会话的lifetime保持相同而stateful package在如下情况invalidate后初始化instance

- 执行REPLACE PACKAGE语句重新生成package时
- 执行ALTER PACKAGE语句recompile package时
- 如果在Package引用的对象之中发生了任何影响了package instance形象的变更（DDL）操作时

Stateless package不拥有状态因此即使变更相关对象的形象也会自动重新编译后执行但stateful package删除现有的package instance并根据更改的package信息重新创建package instance

所以应用程序中使用package时需要注意此种情况下的特殊处理情况

8. PSM Language Element References

8.1 Assignment Statement

功能

在PSM块内的变量或out-bind参数中存储值

语句

```
<assignment statement> ::=
    <assignment target> := <value expression>
    ;

<assignment target> ::=
      collection_variable ( index )
    | cursor_variable
    | :host_cursor_variable
    | out_parameter
    | :host_variable [ :indicator_variable ]
    | record_variable . field_name
    | scalar_variable
```

使用范围及访问权限

仅可在PSM中可用（例如：package, procedure, function）

仅可用于PL block的body区域

语句规则及参数

- collection_variable
  - 在DECLARE区域声明的COLLECTION类型的变量名称
- index
  - 在collection_variable的element中要选择一个的键值
- cursor_variable
  - 在DECLARE区域声明的游标名称
- host_variable
  - 从调用PSM的位置传过来的out或in-out类型的client-side变量名称
- indicator_variable
  - 用于检查host_variable的NULL值并查看实际值长度的indicator变量的名称
- record_variable
  - 已在DECLARE区域声明的RECORD类型的变量名称
- field_name
  - RECORD类型变量中包含的一个字段的名称
- scalar_variable
  - 在DECLARE区域已声明的普通标量（scalar）类型的变量名称

说明

Assignment语句的目标主要分为外部bind参数和内部PSM变量

- 外部参数
  - 主机变量
  - 主机游标变量
- 内部变量
  - out类型函数参数
  - 内部声明的变量
    - Scalar/ record/ multi-set类型
    - Record类型变量的特定字段
    - Multi-set变量的特定要素

除procedure/ function parameter外的所有变量均可以具有指定scope的名称

使用示例

以下为使用<assignment statement>的示例

- PSM内部变量的分配

```
gSQL> DECLARE
  V1 INTEGER := 0;
BEGIN
  FOR I IN 1..10 LOOP
    V1 := V1 + I;
  END LOOP;
```

```
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/

V1 = 55

Anonymous PL block executed.
```

- 主机变量的分配

```
gSQL> \var P1 INTEGER

gSQL>
BEGIN
  :P1 := 100;
END;
/

gSQL> \print P1
 P1
---
100

Anonymous PL block executed.
```

兼容性

<assignment statement> 语句对比标准SQL有以下区别

- 标准SQL中的<assignment statement>定义<singleton variable assignment>和<multiple variable

assignment>但SUNDB仅支持<singleton variable assignment>形式

- 标准SQL中的<assignment statement>以SET关键字开头但SUNDB不使用SET关键字
- 标准SQL中的<assignment statement>在目标和值之间使用<equal operator>（=）但SUNDB使用 :=

Feature ID	说明	是否支持
P002	Computational completeness	X
P006	Multiple assignment	X

Table 8-1 **标准SQL兼容性**

参考内容

详细内容参考如下

- Scalar Variable Declaration
- Record Variable Declaration
- COLLECTION Variable Declaration（收集变量声明）
- Cursor Variable Declaration（游标变量声明）

8.2 基本LOOP语句

功能

使用GOTO或者EXIT直到LOOP结束一直反复执行LOOP内的statement

语句

```
<basic loop statement> ::=
    LOOP { <SQL procedure statement> ; }... END LOOP [ loop_name ]
    ;
```

使用范围及访问权限

仅可在PSM中可用（例如： package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- loop_name
  - <basic loop statement>的标签名称.
  - 仅有注释的作用因此即使与实际<basic loop statement>的标签名不同也不会有问题

说明

基本循环语句（Basic loop）在LOOP内重复执行语句

由于基本循环语句（Basic loop ）是基于循环的语句因此GOTOEXIT和CONTINUE可以是由标签指示的目标语句

使用示例

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
    EXIT WHEN V1 > 2;
  END LOOP;
END;
/
V1 = 1
V1 = 2

Anonymous PL block executed.
```

兼容性

<basic loop statement>语句与标准SQL中的<loop statement>相同

Feature ID	说明	支持状态
P002	Computational completeness	O

Table 8-2 标准SQL兼容性

参考内容

详细内容参考如下

- CONTINUE Statement（CONTINUE 声明）
- EXIT Statement（退出声明）
- GOTO Statement

8.3 块(BEGIN .. END)

功能

创建新范围定义变量游标类型和exception等

语句

```
<PSM block> ::=
    [ DECLARE <declare item>... ] BEGIN <SQL procedure statement list> END
    ;

<declare item> ::=
    <variable declaration>
    | <explicit cursor declaration>
    | <explicit cursor definition>
    | <cursor variable declaration>
    | <type definition>
    | <exception declaration>
    | <exception init pragma>
    | <procedure declaration>
    | <procedure definition>
    | <function declaration>
    | <function definition>

<executable statement list> ::=
    [ <label list> ] { <SQL procedure statement> ; }...
```

```
<label list> ::=
    { << identifier >>  }...

<SQL procedure statement>
      <PSM Static SQL>
    | <PSM Dynamic SQL>
    | <PSM Control Statement>
```

使用范围及访问权限

仅可在PROCEDUREFUNCTION或Anonymous Block中使用

语句规则及参数

- Variable declaration
  - 声明要在块中使用的scalar/ record/ array类型的变量
- Explicit cursor declaration
  - 声明在块中使用的游标
- Explicit cursor definition
  - 定义在块中使用的游标
- Cursor variable declaration
  - 声明在块中使用的Cursor变量
- Type definition
  - 定义在块中声明变量时使用的用户自定义类型
- Exception declaration
  - 声明在块中使用的exception

- Exception init pragma
  - 设置用户定义的exception要处理的错误代码
- Procedure declaration
  - 声明在块中使用的procedure
- Procedure definition
  - 定义在块中使用的procedure
- Function declaration
  - 声明在块中使用的function
- Function definition
  - 定义在块中使用的function
- Static SQL
  - 可以在PSM中执行的DML（数据操作语言）和DCL（数据控制语言）
- Dynamic SQL
  - 指SUNDB PSM支持的动态查询处理语句
- PSM Control Statement
  - 指SUNDB PSM支持的各种流控制语句

说明

<psm block>是PSM的基本组成部分

块可以具有声明部分(declaration part)和异常处理部分(exception handling part)

块可以重叠重叠的块有一个新的子变量范围 父块不能引用子块的变量

使用示例

```
gSQL>
<<MAIN>>
```

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
  <<SUB1>>
  DECLARE
    V1 VARCHAR(10) := 'ABC';
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    DBMS_OUTPUT.PUT_LINE( 'SUB1.V1 = ' || SUB1.V1 );
    DBMS_OUTPUT.PUT_LINE( 'MAIN.V1 = ' || MAIN.V1 );
 END;
END;
/
V1 = 1
V1 = ABC
SUB1.V1 = ABC
MAIN.V1 = 1

Anonymous PL block executed.
```

兼容性

标准SQL的<compound statement>定义了指定新保存点的ATOMIC / NOT ATOMIC语句但SUNDB不支持

Feature ID	说明	备注
P002	Computational completeness	不支持ATOMIC语法

Table 8-3 标准SQL兼容性

参考内容

详细内容参考 PSM概要

8.4 CASE Statement

功能

执行指定的多个条件中返回TRUE的条件对应的执行语句列表

语句

```
<case statement> ::=
    <simple case statement>
    | <searched case statement>
    ;

<simple case statement> ::=
    CASE <case operand> <simple case statement when clause>...
    [ <case statement else clase> ]
    END CASE

<searched case statement> ::=
```

```
    CASE <searched case statement when clause>...
    [ <case statement else clase> ]
    END CASE

<simple case statement when clause> ::=
    WHEN <when operand>
        THEN <executable statement list>

<searched case statement when clause> ::=
    WHEN <search condition>
        THEN <executable statement list>
```

使用范围及访问权限

仅可在PSM中使用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- Case operand
  - 任何可以作为标量值评估的表达式
  - 但不能包含子查询语句
- When operand
  - 对比是否与<case operand>相同的表达式
  - 不能包含子查询语句
- Search condition
  - 在<searched case statement>评估为TRUE时执行的条件表达式

- 不能包含子查询语句

- Executable statement list
  - 可以在PSM中执行的所有语句的列表

说明

与IF语句类似评估条件并执行返回TRUE的WHEN子句的语句

按照顺序评估条件表达式并该条件表达式为TRUE时不评估后续条件表达式

没有符合条件表达式并且未描述ELSE子句时报错

使用示例

简单案例使用

```
gSQL> DECLARE
V1 integer := 0;
BEGIN
  SELECT 2 INTO V1 FROM DUAL;

  CASE V1 WHEN 0 THEN DBMS_OUTPUT.PUT_LINE ('Result = 0');
          WHEN 1 THEN DBMS_OUTPUT.PUT_LINE ('Result = 1');
          WHEN 2 THEN DBMS_OUTPUT.PUT_LINE ('Result = 2');
          ELSE DBMS_OUTPUT.PUT_LINE ('Result = OTHER');
  END CASE;
END;
/
Result = 2
```

```
Anonymous PL block executed.
```

搜索案例使用

```
gSQL> DECLARE
V1 integer := 0;
BEGIN
  SELECT 2 INTO V1 FROM DUAL;

  CASE WHEN V1 = 0 THEN DBMS_OUTPUT.PUT_LINE ('Result = 0');
       WHEN V1 = 1 THEN DBMS_OUTPUT.PUT_LINE ('Result = 1');
       WHEN V1 = 2 THEN DBMS_OUTPUT.PUT_LINE ('Result = 2');
       ELSE DBMS_OUTPUT.PUT_LINE ('Result = OTHER');
  END CASE;
END;
/
Result = 2

Anonymous PL block executed.
```

兼容性

标准SQL中的CASE语句定义行类型（列表类型）值之间的比较但SUNDB不支持

标准SQL中的CASE语句可以以列表定义在<when operand>以','区分的多个条件但SUNDB不支持

Feature ID	说明	备注
P002	Computational completeness	P004P008不支持

Feature ID	说明	备注
P004	Extended CASE statement	-
P008	Comma-separated predicates in simple CASE statement	-

Table 8-4 标准SQL兼容性

8.5 CLOSE Statement

功能

关闭打开状态的游标

语句

```
<close statement> ::=
    CLOSE cursor_name
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- cursor_name
  - 要关闭的游标的名称

说明

关闭open状态的游标

可以使用open语句再次打开关闭的游标

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
  CURSOR C1 IS SELECT I1 FROM T1;
  V1 T1%ROWTYPE;
BEGIN
  OPEN C1;
  FETCH C1 INTO V1;
```

```
  CLOSE C1;
END;
/

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- FETCH Statement
- OPEN Statement

8.6 Collection Method Invocation

功能

提供可搜索集合类型变量的方法

语句

```
<collection method> ::=
         variable_name . <method>
    ;

<method> ::=
        first ()
      | last  ()
      | prior ( expression )
      | next  ( expression )
      | count ()
      | exists ( expression )
      | delete ( expression )
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- 仅适用于声明为collection类型的变量
- FIRSTLASTCOUNT不能有参数
- 需要指定PRIORNEXTEXISTSDELETE等对象时必须指定参数
- 对于DELETE其操作类似于PSM语句不能以其他变量返回结果（不适用于expression）

说明

参考下表

功能名称	功能	返回值	是否需要参数
FIRST	返回最小的键	INDEX OF中指定的密钥类型	X
LAST	返回最大的键	INDEX OF中指定的密钥类型	X
PRIOR	返回小于输入键的键	INDEX OF中指定的密钥类型	O
NEXT	返回大于输入键的键	INDEX OF中指定的密钥类型	O
COUNT	返回存储的数量	INTEGER	X
DELETE	删除Key对应的值	N/A	O
EXISTS	返回key的存在与否	BOOLEAN	O

Table 8-5 **函数**

使用示例

```
DECLARE
TYPE rec IS TABLE OF VARCHAR(20) INDEX BY VARCHAR(20);
v1 rec;
v2 VARCHAR(20);
BEGIN
  DBMS_OUTPUT.PUT_LINE( '-----------------------------------');
  DBMS_OUTPUT.PUT_LINE( 'first = ' || v1.first);
  DBMS_OUTPUT.PUT_LINE( 'last = '  || v1.last);
  DBMS_OUTPUT.PUT_LINE( 'prior = ' || v1.prior('aaa'));
  DBMS_OUTPUT.PUT_LINE( 'next = '  || v1.next('aaa'));
  DBMS_OUTPUT.PUT_LINE( 'count = ' || v1.count());

  FOR I IN 1 .. 9
  LOOP
      v1('a' || to_char(i)) := 'a' || to_char(i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( '-----------------------------------');
  DBMS_OUTPUT.PUT_LINE( 'first = ' || v1.first);
  DBMS_OUTPUT.PUT_LINE( 'last = '  || v1.last);
  DBMS_OUTPUT.PUT_LINE( 'count = ' || v1.count());

  DBMS_OUTPUT.PUT_LINE( '-----------------------------------');
  DBMS_OUTPUT.PUT_LINE( 'Print all from first to last');
  v2 := v1.first;
  WHILE v2 IS NOT NULL
  LOOP
      DBMS_OUTPUT.PUT_LINE('Key= ' || v2 || ',Value=' || v1(v2));
      v2 := v1.next(v2);
```

```
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( '-----------------------------------');
  DBMS_OUTPUT.PUT_LINE( 'Print all from last to first');
  v2 := v1.last;
  WHILE v2 IS NOT NULL
  LOOP
      DBMS_OUTPUT.PUT_LINE('Key= ' || v2 || ',Value=' || v1(v2));
      v2 := v1.prior(v2);
  END LOOP;

  v1.delete(v1.first());
  DBMS_OUTPUT.PUT_LINE('count = ' || v1.count() );
  DBMS_OUTPUT.PUT_LINE('first = ' || v1.first() );

END;
/
-----------------------------------
first =
last =
prior =
next =
count = 0
-----------------------------------
first = a1
last = a9
count = 9
-----------------------------------
Print all from first to last
Key= a1,Value=a1
Key= a2,Value=a2
Key= a3,Value=a3
Key= a4,Value=a4
```

```
Key= a5,Value=a5
Key= a6,Value=a6
Key= a7,Value=a7
Key= a8,Value=a8
Key= a9,Value=a9
-----------------------------------
Print all from last to first
Key= a9,Value=a9
Key= a8,Value=a8
Key= a7,Value=a7
Key= a6,Value=a6
Key= a5,Value=a5
Key= a4,Value=a4
Key= a3,Value=a3
Key= a2,Value=a2
Key= a1,Value=a1
count = 8
first = a2

Anonymous PL block executed.
```

参考内容

详细内容参考：COLLECTION Variable Declaration（收集变量声明）

8.7 COLLECTION Variable Declaration

功能

声明collection变量

语句

```
<declare record variable> ::=
    variable_name <collectionType>
    ;

<Collection Type Definition> ::=
    TYPE <Type-Name> IS TABLE OF <Element-Type> INDEX BY <Index-Type>
    ;

<Element-Type> ::=
      Built-in SQL Data Type
    | User-Defined Type
    | %TYPE
    | %ROWTYPE

<Index-Type> ::=
      INTEGER
    | LONG
    | CHAR(n)
    | VARCHAR(n)
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的声明区域

语句规则及参数

- Type-name
  - 指定用户要使用的collection类型的名称.
- Element-type
  - 指定要存储在collection变量中的element类型
- Index-type
  - 指定存储在collection变量中的键的data type

说明

声明collection类型

使用示例

```
gSQL> DECLARE
TYPE rec IS TABLE OF VARCHAR(20) INDEX BY VARCHAR(20);
v1 rec;
v2 VARCHAR(20);
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE( 'first = ' || v1.first);
  DBMS_OUTPUT.PUT_LINE( 'last = '  || v1.last);
  DBMS_OUTPUT.PUT_LINE( 'prior = ' || v1.prior('aaa'));
  DBMS_OUTPUT.PUT_LINE( 'next = '  || v1.next('aaa'));
  DBMS_OUTPUT.PUT_LINE( 'count = ' || v1.count());

  FOR I IN 1 .. 10
  LOOP
      v1('a' || i) := 'a' || i;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( 'first = ' || v1.first);
  DBMS_OUTPUT.PUT_LINE( 'last = '  || v1.last);
  DBMS_OUTPUT.PUT_LINE( 'count = ' || v1.count());

  DBMS_OUTPUT.PUT_LINE( 'Print all from first to last');
  v2 := v1.first;
  WHILE v2 IS NOT NULL
  LOOP
      DBMS_OUTPUT.PUT_LINE('Key= ' || v2 || ',Value=' || v1(v2));
      v2 := v1.next(v2);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( 'Print all from last to first');
  v2 := v1.last;
  WHILE v2 IS NOT NULL
  LOOP
      DBMS_OUTPUT.PUT_LINE('Key= ' || v2 || ',Value=' || v1(v2));
      v2 := v1.prior(v2);
  END LOOP;

END;
```

```
/
first =
last =
prior =
next =
count = 0
first = a1
last = a9
count = 10
Print all from first to last
Key= a1,Value=a1
Key= a10,Value=a10
Key= a2,Value=a2
Key= a3,Value=a3
Key= a4,Value=a4
Key= a5,Value=a5
Key= a6,Value=a6
Key= a7,Value=a7
Key= a8,Value=a8
Key= a9,Value=a9
Print all from last to first
Key= a9,Value=a9
Key= a8,Value=a8
Key= a7,Value=a7
Key= a6,Value=a6
Key= a5,Value=a5
Key= a4,Value=a4
Key= a3,Value=a3
Key= a2,Value=a2
Key= a10,Value=a10
Key= a1,Value=a1

Anonymous PL block executed.
```

兼容性

标准SQL未定义

参考内容

详细内容参考 : Collection Method Invocation（集合方法调用）

8.8 CONTINUE Statement

功能

停止当前进行中的语句列表的执行并执行上层循环语句的下一个iteration

语句

```
<continue statement> ::=
    CONTINUE [ label_name ] [ WHEN condition ]
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

具有目标标签的语句必须是以下循环语句中的一个

- basic loop statement
- for loop statement
- while statement
- forall statement

语句规则及参数

- Label name
  - 可以是标识符链的形式
- Condition
  - 指定时仅在条件为TRUE时返回至loop statement

说明

停止执行当前进行中的语句列表并返回至上层循环语句

如果指定了label则返回至拥有该标签名称的上层循环语句

如果未指定label则返回至最近的上层循环语句

如果有多个拥有相同标签名称的上层语句则选择最接近的语句

只能返回至当前位置可见（在重叠范围内）的循环语句

如果指定了条件则仅在条件为TRUE时才返回该条件

如果未指定条件则无条件返回

使用示例

```
DECLARE
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  WHILE V1 <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
```

```
    IF V1 <= 2 THEN
      DBMS_OUTPUT.PUT_LINE( 'CONTINUE' );
      CONTINUE;
    ELSE
      EXIT;
    END IF;
    DBMS_OUTPUT.PUT_LINE( 'END-OF-WHILE' );
  END LOOP AAA;
END;
/
V1 = 1
CONTINUE
V1 = 2

Anonymous PL block executed.
```

兼容性

<continue statement>语句在功能上类似于标准SQL的<iterate statement>

但是<iterate statement>语句不提供WHEN条件功能

参考内容

详细内容参考如下

- EXIT Statement（退出声明）
- GOTO Statement

8.9 Cursor FOR LOOP Statement

功能

在PSM中执行循环的次数与用户声明的游标或查询生成的行数一样多

语句

```
<Cursor For Loop statement> ::=
       FOR <Variable_Name> IN <Cursor>
       LOOP
            { <SQL procedure statement> ; }...
       END LOOP [ Label_Name ]
       ;

<Cursor> ::=
       < ( Implict_Cursor_Query ) >
     | < Explict_Cursor_Name > [ ( [ <actual param> ] ) ]


<Implict_Cursor_Query> ::=
       SELECT statement
     | SELECT_FOR_UPDATE statement
     | INSERT_RETURNING_QUERY statement
     | UPDATE_RETURNING_QUERY statement
     | DELETE_RETURNING_QUERY statement
```

```
<actual param> ::=
      ( expression [ , expression ] .. )
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PSM中的主区域

语句规则及参数

在For∽Loop中声明的变量仅在循环范围内有效（不能在Cursor For Loop Block Scope之外引用该变量）

使用Cursor Name的情况

使用cursor name执行LOOP时必须提前声明游标

实际参数的详细内容参考 OPEN Statement

使用Cursor Query的情况

只能执行SUNDB内部以implicit cursor处理的查询如Select或returning query

说明

通过执行与游标生成的结果数量相同数量的loop执行循环内的PSM语句

如果在LOOP期间游标成为无效（例如：关闭）则不再执行循环并将其视为错误

指定explicit cursor name时该cursor已经打开则将其视为错误

自动生成FOR LOOP子句中指定的接收cursor结果的变量（生成为由游标执行结果返回的result set的row type类型）

但是如果未在用户的游标查询结果中指定非特定表的column的select target expression的alias等时可能会发生错误

使用示例

使用Explicit Cursor

```
DECLARE
CURSOR c1 IS SELECT * FROM T1;
BEGIN
  FOR rec IN C1
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'RowCount=' || c1%rowcount || ',C1=' || rec.c1 || ',
C2=' || rec.c2);
  END LOOP;
END;
/
RowCount=1,C1=1, C2=1
RowCount=2,C1=2, C2=2
RowCount=3,C1=3, C2=3
RowCount=4,C1=4, C2=4
```

```
RowCount=5,C1=5, C2=5
RowCount=6,C1=6, C2=6
RowCount=7,C1=7, C2=7
RowCount=8,C1=8, C2=8
RowCount=9,C1=9, C2=9
RowCount=10,C1=10, C2=10

Anonymous PL block executed.
```

使用Cursor Query

```
BEGIN
  FOR rec IN (select * from t1)
  LOOP
      DBMS_OUTPUT.PUT_LINE( 'RowCount=' || sql%rowcount || ',C1=' || rec.c1 ||
', C2=' || rec.c2);
  END LOOP;
END;
/
C1=1, C2=1
C1=2, C2=2
C1=3, C2=3
C1=4, C2=4
C1=5, C2=5
C1=6, C2=6
C1=7, C2=7
C1=8, C2=8
C1=9, C2=9
C1=10, C2=10

Anonymous PL block executed.
```

参考内容

详细内容参考如下

- Explicit Cursor Declaration and Definition（显式游标声明和定义）
- GOTO Statement
- EXIT Statement（退出声明）

8.10 Cursor Variable Declaration

功能

在PSM的DECLARE部分声明游标变量

语句

```
<cursor variable declaration> ::=
    variable_name <type>
    ;

<cursor type definition> ::=
      TYPE <type_name> IS REF CURSOR [ RETURN <return type> ]

<return type> ::=
      <table_name | view_name | cursor_name | cursor_variable > % ROWTYPE
    | <record_variable_name> % TYPE
    | <record_type_name>
```

使用范围及访问权限

仅在PSM中可用（例如： package, procedure, function）

仅适用于PSM中的声明区域

语句规则及参数

只能在游标变量之间进行指定游标变量的初始值或分配

说明

游标变量承担一种pointer作用指向不属于特定游标的游标

使用示例

```
DECLARE
TYPE rec IS RECORD (V1 VARCHAR(20), V2 VARCHAR(20));
TYPE cv IS REF CURSOR RETURN rec;
BEGIN
    NULL;
END;
/

Anonymous PL block executed.
```

参考内容

详细内容参考如下

- OPEN Statement
- FETCH Statement
- CLOSE Statement（CLOSE 声明）

8.11 DELETE Statement Extension

功能

可以使用PSM的record type变量将结果保存在RETURNING INTO子句中

语句

```
<PSM delete statement extension: searched> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        [ <returning into clause> ]
    ;

<delete statement: positioned> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        WHERE CURRENT OF cursor_name
    ;

<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]

<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>
```

```
<fetch first clause> ::=
    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]

<limit clause>
    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }

<returning into clause> ::=
    { RETURN | RETURNING } { * | { <value expression> [ [AS] alias_name] }
[, ...] } INTO variable_name [, ...]
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PSM的主区域

语句规则及参数

通过Returning Into返回的变量类型为record-type时不能与其他变量类型混合使用

说明

可以使用PSM record type变量将结果保存在RETURNING INTO子句中

使用示例

```
gSQL> CREATE TABLE T1( C1 VARCHAR(20), C2 VARCHAR(20));
Table created.

gSQL> COMMIT;
Commit complete.

gSQL> INSERT INTO T1 VALUES ('AAA', 'BBB'), ('BBB', 'CCC'), ('CCC', 'DDD');
3 rows created.

gSQL> COMMIT;
Commit complete.



gSQL> DECLARE
  rec t1%ROWTYPE;
BEGIN
  DELETE FROM T1 WHERE C1 = 'AAA' RETURNING * INTO rec;
  DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT=' || SQL%ROWCOUNT);
  DBMS_OUTPUT.PUT_LINE('rec.c1=' || rec.c1 || ', rec.c2=' || rec.c2);
END;
/
SQL%ROWCOUNT=1
rec.c1=AAA, rec.c2=BBB

Anonymous PL block executed.
```

参考内容

详细内容参考**删除数据**

8.12 EXCEPTION_INIT Pragma

功能

设置用户定义的exception要处理的error code

语句

```
< PRAGMA EXCEPTION_INIT > ::=
    PRAGMA EXCEPTION_INIT ( <Exception-Name>, <Internal-ErrorCode> )
    ;
```

使用范围及访问权限

仅可在PSM中可用（例如：package, procedure, function）

仅适用于PSM的声明区域

语句规则及参数

预定义异常（predefined exception）不能用于用作argument的exception name（无法声明predefined exception name）

对于同一个PL block DECLARE子句必须提前声明用作argument的exception name（不能引用其他BLOCK的exception name声明）

<Internal-ErrorCode>必须是DB SYSTEM中存在的内部错误代码（无法设置SUCCESS代码）

说明

用户指定声明与DB SYSTEM的错误代码对应的exception name

使用示例

```
gSQL> DECLARE
user_exception_1 EXCEPTION;
PRAGMA EXCEPTION_INIT( user_exception_1, -17001);
BEGIN
  RAISE USER_EXCEPTION_1;
  EXCEPTION WHEN user_exception_1 THEN DBMS_OUTPUT.PUT_LINE('User Exception_1');
END;
/
User Exception_1

Anonymous PL block executed.
```

兼容性

错误代码不兼容因为每个供应商的错误代码不同

参考内容

详细内容参考如下

- Exception Declaration
- Exception Handler（异常处理程序）

8.13 Exception Declaration

功能

声明PL block内的exception name

语句

```
< Exception-Declaration Statement > ::=
    <Exception-Name>  EXCEPTION
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PSM中的声明区域

语句规则及参数

无法声明预定义的异常名称

不能在相同的SCOPE的DECLARE子句中重复声明

说明

用户指定声明exception

使用示例

```
DECLARE
user_exception_1 EXCEPTION;
user_exception_2 EXCEPTION;
user_exception_3 EXCEPTION;
BEGIN
  RAISE USER_EXCEPTION_2;
  EXCEPTION WHEN user_exception_1 THEN DBMS_OUTPUT.PUT_LINE('User Exception_1');
            WHEN user_exception_2 THEN DBMS_OUTPUT.PUT_LINE('User Exception_2');
            WHEN user_exception_3 THEN DBMS_OUTPUT.PUT_LINE('User Exception_3');
END;
/
User Exception_2

Anonymous PL block executed.
```

兼容性

标准SQL异常声明如下但SUNDB支持上述语句

```
<condition declaration> ::=
DECLARE <condition name> CONDITION [ FOR <sqlstate value> ]
```

参考内容

详细内容参考如下

- Exception Declaration（例外声明）
- EXCEPTION_INIT Pragma

8.14 Exception Handler

功能

对执行PL/SQL过程中发生的DB SYSTEM上的错误引起的默认exception或用户指定发生的exception执行定义动作

语句

```
< Exception Handler Statement > ::=
       EXCEPTION < Exception_When_List >
       ;

< Exception_When_List > ::=
       WHEN < Exception_Name_List > THEN <excutable statement list>  [ WHEN
OTHERS THEN <excutable statement list> ]

< Exception_Name_List > ::=
        <Exception_Name> [ { OR <Exception_Name> }... ]
```

使用范围及访问权限

可以在PL块内使用

语句规则及参数

预定义异常的OTHERS不能使用OR与其他exception name一起描述

预定义异常的OTHERS不能在异常处理程序中重复描述应该在最后描述

说明

Exception 类型

Type	Definer	Has error	Has name	Raise implictly	Raise explictly
Predefined	System	Yes	Yes	Yes	Optionally
User-Defined	User	If user assign	If user assign	No	Yes

预定义异常有SUNDB中提前指定的exception name和error code

其他异常分为用户将SUNDB内部的错误代码设置为与预定义异常名称不同的名称（内部定义）和仅声明exception name而未指定单独的错误代码（用户定义）

Predefined Exception

名称	说明
CASE_NOT_FOUND	不满足CASE WHEN的所有条件或未定义ELSE子句
DUP_VAL_ON_INDEX	发生INDEX duplicate错误
INVALID_CURSOR	游标的状态不正确
INVALID_NUMBER	无法转换为数字
NO_DATA_FOUND	SELECT语句返回0条数据
ROWTYPE_MISMATCH	两个RowType变量具有不同的字段类型
TOO_MANY_ROWS	返回两条以上的row

名称	说明
VALUE_ERROR	类型不匹配（type mismatch）无效转换（invalid casting）等错误
ZERO_DIVIDE	尝试除以零
OTHERS	包含预定义中未定义的错误

Table 8-6 Predefined exception 类型

使用示例

```
gSQL> DECLARE
V1 INTEGER := 0;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Step1');
   V1 := 1 / 0;
   DBMS_OUTPUT.PUT_LINE('Step2');
   EXCEPTION WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE( 'Exception V1=' || V1);
END;
/
Step1
Exception V1=0

Anonymous PL block executed.
```

兼容性

不支持SQL标准语法

参考内容

详细内容参考如下

- EXCEPTION_INIT Pragma
- Exception Declaration（例外声明）

8.15 EXECUTE IMMEDIATE Statement

功能

在PSM中执行动态SQL.

语句

```
<EXECUTE IMMEDIATE statement> ::=
    EXECUTE IMMEDIATE <dynamic-sql> [ <binding-parameters> ]
    ;


<dynamic-sql> ::=
      single_quote_string
    | psm_variable
```

```
<binding-parameters> ::=
      <into-clause>
    | <using-clause>
    | <returning-into-clause>
    | <into-clause> <using-clause>
    | <using-clause> <returning-into-clause>



<into-clause> ::=
    INTO psm_variable [ {, psm_variable} ... ]


<using-clause> ::=
    USING [ <Bind-Type> ] <expression> [ {, [ <Bind-Type> ] <expression>} ... ]


<Bind-Type> ::=
     IN
   | OUT
   | IN OUT


<returing-into-clause> ::=
    RETURNING INTO psm_variable [ {, psm_variable} ... ]
```

使用范围及访问权限

仅在PSM中可用（例如： package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

Dynamic SQL（动态SQL）

- single_quote_string: 用单引号(')括起来的SQL语句（double_quote_string在PSM中被识别为变量）
- PSM-variable：存储在PSM中使用的变量中的SQL语句.

以下为在要执行的SQL语句中使用引号表示数据的示例

```
EXECUTE IMMEDIATE 'INSERT INTO t1 VALUES ( ''Tom'' ) '; -- Tom
EXECUTE IMMEDIATE 'INSERT INTO t1 VALUES ( ''Tom''''House'') '; -- Tom'House
EXECUTE IMMEDIATE 'INSERT INTO t1 VALUES ( ''''''Tom'') ';   -- 'Tom
EXECUTE IMMEDIATE 'INSERT INTO t1 VALUES ( CHR(39) || ''TOM'' ) '; -- 'Tom
```

在dynamic SQL中用户输入变量的部分可以使用 ？ 或 :V1 等标记

Dynamic SQL中描述的SQL语句应为有效的语句

INTO Clause（INTO 子句）

存在Dynamic-Sql执行结果返回并非通过marker绑定的SQL语句的处理结果 （在内部是implicit cursor fetch形式）

语句如下

```
EXECUTE IMMEDIATE 'SELECT ... FROM .. WHERE ...';
EXECUTE IMMEDIATE 'INSERT ... RETURNING ...';
EXECUTE IMMEDIATE 'UPDATE ... RETURNING ...';
```

```
EXECUTE IMMEDIATE 'DELETE ... RETURNING ...';
```

USING Clause（USING子句）

在动态SQL（dynamic-Sql）中使用输入变量时变量或表达式将在USING（可以省略IN）子句中列出

如果有动态SQL（dynamic-Sql）执行结果则USING OUT子句中列出的变量与结果列数一样多（当将结果作为INTO子句返回时）

可以使用USING OUT在以下语句中返回结果

```
EXECUTE IMMEDIATE 'SELECT x, y, z INTO :v1, :v2, :v3 ...';
EXECUTE IMMEDIATE 'INSERT INTO ...  RETURNING C1, C2 INTO :V1, :V2';
EXECUTE IMMEDIATE 'UPDATE T1 SET .. RETURNING C1, C2 INTO :V1, :V2';
EXECUTE IMMEDIATE 'DELETE FROM ...  RETURNING C1, C2 INTO :V1, :V2';
```

RETURNING Clause（RETURNING 子句）

将INSERT/ UPDATE/ DELETE RETURNING INTO子句用作dynamic SQL时SUNDB可以通过将USING子句中描述的变量绑定到OUT模式来返回结果

为了与其他DBMS兼容也可以使用RETURNING-INTO子句获取相同的结果

其他规则

- 对于DDL / DCL无法使用任何BIND子句（INTOUSINGRETURNING子句）
- 由于INTO子句和RETURNING INTO子句用作OUT因此无法指定单独的bind type且不能同时使用
- IN BIND类型仅可使用scalar type变量
- OUT BIND类型可以使用record type变量 但是不能混合标量和记录类型同时列出
- 不能通过INTO子句和USING OUT或RETURNING INTO来分别获取结果

说明

- Dynamic SQL将结果作为INTO子句中描述的变量返回时
  - 返回两条以上结果的查询会引起TOO_MANY_ROWS异常
  - 结果为0则引发NO_DATA_FOUND异常
- Dynamic SQL将结果作为USING或RETURNING子句中描述的变量返回时
  - 返回两条以上非ARRAY的变量时会引发TOO_MANY_ROWS异常
  - 如果结果为零条则不会发生错误
- DDL / DCL执行隐式游标SQL％Attribute变量的结果为如下
  - SQL%ROWCOUNT = 0
  - SQL%ISOPEN = FALSE
  - SQL%FOUND = FALSE
  - SQL%NOTFOUND = TRUE
- 其余的dynamic SQL存储符合该SQL语句处理结果的SQL%Attribute值

使用示例

```
gSQL> DECLARE
V1 INTEGER;
V2 VARCHAR(20);
BEGIN
    V1 := 1;
    V2 := 'abcdef';
    DBMS_OUTPUT.PUT_LINE('#INSERT');
    EXECUTE IMMEDIATE 'insert into t1 values (:a1, :a2)' USING v1, v2;
    EXECUTE IMMEDIATE 'select c1, c2 from t1 where c1 = 1' INTO v1, v2;
    DBMS_OUTPUT.PUT_LINE('C1='|| v1 || ', C2=' || v2);
```

```
    V1 := 1;
    V2 := 'xyz';
    DBMS_OUTPUT.PUT_LINE('#UPDATE');
    EXECUTE IMMEDIATE 'update t1 set c2 = :a1 where c1 = :a2' USING V2, V1;

    V1 := 1;
    V2 := '';
    DBMS_OUTPUT.PUT_LINE('#SELECT');
    EXECUTE IMMEDIATE 'select c1, c2 from t1 where c1 = :a1' INTO v1, v2 USING
v1;
    DBMS_OUTPUT.PUT_LINE('C1='|| v1 || ', C2=' || v2);

    V1 := 1;
    V2 := '';
    DBMS_OUTPUT.PUT_LINE('#DELETE');
    EXECUTE IMMEDIATE 'delete from t1 where c1 = :a1' USING v1;
END;
/
#INSERT
C1=1, C2=abcdef
#UPDATE
#SELECT
C1=1, C2=xyz
#DELETE

Anonymous PL block executed.
```

8.16 EXIT Statement

功能

在上层循环语句中从带有指定标签的循环语句中跳出并执行下一个语句

语句

```
<exit statement> ::=
    EXIT [ label_name ] [ WHEN condition ]
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block主区域

语句规则及参数

- Label name
  - 要跳出的循环语句的标签名称
  - 可以为标识符链的形式

- Condition
  - 如果指定条件则仅当条件为TRUE时才跳出循环语句

说明

- 停止执行当前语句列表并跳出上层循环语句
- 具有目标标签的语句应为以下循环语句中的一个
  - basic loop statement
  - for loop statement
  - while statement
  - forall statement
- 指定label时跳出拥有该标签名的上层循环语句
- 未指定label时跳出最接近的上层循环语句
- 如果有多个具有相同标签名的上层语句则选择最接近的语句
- 只能跳出在当前位置可见（在重叠范围内的）的循环语句
- 指定条件时仅在条件为TRUE时才跳出
- 未指定条件时无条件跳出

使用示例

```
gSQL> DECLARE
  V1 INTEGER := 1;
BEGIN
  <<AAA>>
  WHILE V1 <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    EXIT;
```

```
    V1 := V1 + 1;
  END LOOP AAA;
END;
/
V1 = 1

Anonymous PL block executed.
```

兼容性

标准SQL中不存在

8.17 Explicit Cursor Declaration and Definition

功能

在PSM的DECLARE部分声明游标

语句

```
<cursor declaration> ::=
    CURSOR cursor_name [ <cursor param spec> ] RETURN rowtype
    ;

<cursor param spec> ::=
      ( <cursor param decl> [ , <cursor param decl> ] .. )

<cursor param decl> ::=
      param_name [ IN ] datatype [ { ':=' | DEFAULT } expression ]

<cursor definition> ::=
    CURSOR cursor_name [ <cursor param spec> ] [ RETURN rowtype ]
    IS <cursor query>
    ;

<cursor query> ::==
    select statement
    | select for update_statement
    | insert returning query statement
```

```
| update returning query statement
| delete returning query statement
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的声明区域

语句规则及参数

Cursor Name

要声明的游标的名称

游标名称的长度应小于128个字节

在范围内必须是唯一的名称

RowType

定义游标的记录类型

定义游标时指定的选择目标数量应相同并且数据类型应兼容

未指定rowtype时则根据定义游标时描述的select_statement的SELECT目标自动指定合适的rowtype

Param Name

在特定游标中用于区分参数的名称

在该游标中必须是唯一的名称

如果名称与可引用的范围中的另一个变量相同则首先引用游标的参数

DataType

指定该参数的数据类型

可以使用SUNDB提供的所有内置类型和PSM中定义的类型

但是内置类型无法指定限制范围的语句（precision/ scale）并在内部指定为该数据类型的最大范围

Cursor Query

- Cursor可执行的query如下
  - Select语句
  - Select For Update语句
  - Insert Returning语句
  - Update Returning语句
  - Delete Returning语句
- SELECT INTO语句无法使用

说明

- <explicit cursor declaration>语句声明或定义游标
  - cursor declaration： 仅声明游标名称和格式
  - cursor definition： 详细定义游标的名称格式以及要执行的SELECT语句
- 可以在声明之后定义并使用显式游标也可以在没有声明的情况下直接定义使用
- 如果在声明之后定义并使用则游标名称参数spec记录类型定义必须完全一致
- 显式游标的声明和定义必须存在于同一个块中

- 当前游标进入定义的块时生成显式游标并在退出块时自动CLOSE并删除
- 特定时间点可生成的显式游标的最大数量受“MAXIMUM_NAMED_CURSOR_COUNT”参数的限制
- 可以在显式游标执行的<cursor query>语句中使用以下变量
  - 该游标的参数
  - 声明时可在范围内引用的所有PSM变量（非开放时间点的范围）
  - 外部绑定参数（Anonymous PL Block的情况）
- 执行用户自定义的<cursor query>的显式游标具有以下属性
  - IN_SENSITIVE(不受其他事务所更改的影响)
  - NON_SCROLLABLE (无法再次获取以前的记录)
  - READ_ONLY (只能进行read操作)
  - WITH-HOLD (执行COMMIT / ROLLBACK游标也不会自动关闭)

使用示例

```
gSQL> CREATE TABLE t1( c1 INTEGER, c2 VARCHAR( 6 ) );

Table created.

gSQL> COMMIT;

Commit complete.
```

- Insert Returning Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   CURSOR cur1( p1 INTEGER, p2 varchar(6) ) RETURN t1%ROWTYPE
      IS INSERT INTO t1 VALUES ( p1, p2 ) RETURNING *;
```

```
BEGIN
   OPEN cur1( 1, 'aaa' );

   FETCH cur1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE cur1;
END;
/

var1.c1 = 1 , var1.c2 = aaa
Anonymous PL block executed.
```

- Select Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   CURSOR cur1( p1 INTEGER, p2 varchar(6) ) RETURN t1%ROWTYPE IS
      SELECT * FROM t1 WHERE c1 = p1 AND c2 = p2;
BEGIN
   OPEN cur1( 1, 'aaa' );

   FETCH cur1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE cur1;
END;
/

var1.c1 = 1 , var1.c2 = aaa
```

```
Anonymous PL block executed.
```

- Update Returning Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   CURSOR cur1( p1 INTEGER, p2 varchar(6) ) RETURN t1%ROWTYPE IS
      UPDATE t1 SET c1 = p1, c2 = p2 RETURNING *;
BEGIN
   OPEN cur1( 3, 'ccc' );

   FETCH cur1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE cur1;
END;
/

var1.c1 = 3 , var1.c2 = ccc
Anonymous PL block executed.
```

- Delete Returning Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   CURSOR cur1( p1 INTEGER, p2 varchar(6) ) RETURN t1%ROWTYPE IS
      DELETE FROM t1 WHERE c1 = p1 AND c2 = p2 RETURNING *;
BEGIN
   OPEN cur1( 3, 'ccc' );
```

```
   FETCH cur1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE cur1;
END;
/

var1.c1 = 3 , var1.c2 = ccc
Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- OPEN FOR Statement
- FETCH Statement
- CLOSE Statement（CLOSE 声明）

8.18 FETCH Statement

功能

获取已打开的游标的一条记录

语句

```
<fetch statement> ::=
    FETCH cursor_name <into clause>
    ;

<into clause> ::=
    INTO { variable [ , variable ] .. | record }
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- Cursor name
  - 要获取的游标名称
- Variable
  - 存储获取结果中的一个列值的scalar类型变量或绑定参数
- Record
  - 存储一个获取结果的整个记录的记录类型变量

说明

从打开的游标中获取一条记录并将值复制到INTO子句中指定的变量

如果仅声明游标而未定义则会报错

该游标应为打开的状态

INTO子句中指定的变量类型应与获取的记录结果的数据类型相互兼容

INTO子句中指定的变量数应与游标SELECT目标的数量相同

但是INTO子句中指定的变量为记录类型时只能指定一个

而且该记录变量中的字段数应等于SELECT目标的数量

如果在没有要提取的记录时调用fetch则INTO子句中目标变量的值不变

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER );

Table created.
```

```
gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
  CURSOR C1 IS SELECT * FROM T1;
  V1 INTEGER := 0;
  V2 INTEGER := 0;
  CNT INTEGER := 0;
  TOTAL INTEGER := 0;
BEGIN
  FOR I IN 1..100 LOOP
    INSERT INTO T1 VALUES( I );
  END LOOP;

  COMMIT;

  OPEN C1;

  LOOP
    FETCH C1 INTO V1;

    CNT := CNT + 1;
    TOTAL := TOTAL + V1;
    V2 := V1;
    EXIT WHEN V1 = 100;
  END LOOP;

  CLOSE C1;

  DBMS_OUTPUT.PUT_LINE( 'CNT = ' || CNT || ' TOTAL = ' || TOTAL );
END;
```

```
/

CNT = 100 TOTAL = 5050

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- OPEN Statement
- CLOSE Statement（CLOSE 声明）

8.19 FOR LOOP Statement

功能

当索引变量具有指定值范围时每次将索引变量递增或递减1（REVERSE）并执行内部语句

语句

```
<for loop statement> ::=
    FOR index_variable_name IN [ REVERSE ] lower_bound .. upper_bound
    LOOP { <SQL procedure statement> ; }... END LOOP [ loop_name ]
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- Index variable name
    - 在FOR语句中用作索引的变量的名称在内部使用NATIVE_BIGINT类型的变量

- Lower bound
  - 应为整数型如果使用有浮动小数点的数字时类型转换时四舍五入
- Upper bound
  - 应为整数型如果使用有浮动小数点的数字时类型转换时四舍五入

说明

for loop语句执行内部的statement list时增加或减少索引变量的值

- 指定REVERSE时
  - 索引变量从upper_bound值开始递减1当索引变量的值变得小于lower_bound时终止执行for loop语句
  - upper_bound值小于lower_bound值时不执行内部的statement list
- 未指定REVERSE时
  - 索引变量从lower_bound值开始增加1当索引变量的值变得大于upper_bound时终止执行for loop语句
  - lower_bound值大于upper_bound值时不执行内部的statement list

使用示例

```
gSQL> BEGIN
  FOR I IN 0 .. 5 LOOP
    DBMS_OUTPUT.PUT_LINE( 'I = ' || I );
  END LOOP;
END;
/
```

```
I = 0
I = 1
I = 2
I = 3
I = 4
I = 5
Anonymous PL block executed.


gSQL> BEGIN
  FOR I IN REVERSE 0 .. 5 LOOP
    DBMS_OUTPUT.PUT_LINE( 'I = ' || I );
  END LOOP;
END;
/

I = 5
I = 4
I = 3
I = 2
I = 1
I = 0
Anonymous PL block executed.
```

兼容性

标准SQL未定义

参考内容

详细内容参考如下

- CONTINUE Statement（CONTINUE 声明）
- EXIT Statement（退出声明）
- GOTO Statement

8.20 Function Declaration and Definition

功能

声明并定义函数（function）

语句

```
<function declaration> ::=
        FUNCTION <function name> [ ( <parameter list> ) ] <return clause>
        ;

<function definition> ::=
        FUNCTION <function name> [ ( <parameter list> ) ] <return clause>
        { IS | AS }
        <item declaration>
        BEGIN
        <pl statement list>
        END [ <function name> ]
        ;

<parameter list> ::=
      <parameter name> [ <parameter mode> ] <datatype> [ <parameter default> ]
[ , ... ]

<parameter mode> ::=
      IN
```

```
    | OUT
    | IN OUT

<parameter default> ::=
      { := | DEFAULT } <value expression>

<return clause> ::=
      RETURN <datatype>
    | RETURN TABLE ( <table function column list> )

<table function column list> ::=
      <column name> <datatype> [ , ... ]
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的声明区域

语句规则及参数

function name

PL block中要创建的函数的名称应在PL block中是唯一的名称

即PL block中声明的PL item和function不能有相同的名称

函数名称的长度应小于128个字节

parameter name

定义函数的参数名称

各参数在函数中名称唯一

即函数的参数和PL item不能有相同的名称

参数名称的长度应小于128个字节

单个函数中可用参数的最大数量没有限制

parameter mode

设置各参数模式

参数模式为IN, OUT, IN OUT

未指定参数模式时默认模式为IN

parameter default

参数的默认值

执行函数时可省略指定了参数默认值的参数

未指定参数并省略时默认值为定义参数时指定的 <value expression>

<value expression>的数据类型应为参数的数据类型

在有<parameter default>的参数以后定义的所有参数必须有<parameter default>

return clause

定义函数的返回类型

<return clause>中如下定义

- RETURN <datatype>

- 定义函数返回的返回值的数据类型

- RETURN TABLE (<table function column list>)
  - 定义返回结果集的表类型

table function column list

Table function返回的结果集的column名称

Column名称的长度应小于128个字节

无Column数量限制

各column名称在<table function column list>中是唯一的

Column名称可以与参数和declare item的名称相同

<table function column list>中定义的column不可在函数的PL块中引用

Item Declaration

声明要在函数中使用的本地变量等的item

可以声明在PL block中可声明的所有item

PL Stmt List

是函数的主体部分列出了要执行的PL语句

说明

如下声明并定义函数

- PL块（例：匿名块procedure及函数的块块语句的块）
  - 作为PL块的item中之一声明并定义函数
  - 在PL块中声明并定义的函数称为内嵌函数（nested function）
  - 内嵌函数仅可在PL块范围内使用
- 包规格（package specification）
  - 声明包的公共函数（public function）
  - 包的公共函数可在数据库内使用
- 包主体（package body）
  - 定义在包规格中声明的公共函数
  - 声明并定义包的私有函数（private function）
  - 包的私有函数仅可在包范围内使用

函数的使用方法与schema-level function相同

使用示例

```
gSQL> CREATE TABLE t_score( c_grade INTEGER, c_score INTEGER );

Table created.

gSQL> INSERT INTO t_score VALUES ( 1 , 98 ) , ( 1 , 97 ) , ( 1 , 99 ),
                                 ( 2 , 95 ) , ( 2 , 98 ) , ( 2 , 92 ),
                                 ( 3 , 98 ) , ( 3 , 96 ) , ( 3 , 94 );

9 rows created.

gSQL> COMMIT;

Commit complete.
```

- 内嵌函数

```
gSQL>
DECLARE
  v_max_score INTEGER;

  FUNCTION nestedfunc RETURN INTEGER;

  FUNCTION nestedfunc RETURN INTEGER AS
    v_max INTEGER;
  BEGIN
    SELECT MAX( c_score )
      INTO v_max
      FROM t_score;

    RETURN v_max;
  END;
BEGIN
  v_max_score := nestedfunc;

  DBMS_OUTPUT.PUT_LINE( 'Max Score : ' || v_max_score );
END;
/

Max Score : 99
Anonymous PL block executed.
```

- 包函数

```
gSQL>
CREATE OR REPLACE PACKAGE pkg1 AS
  -- Declare Package Public Function
  FUNCTION func1( p_grade INTEGER, p_option VARCHAR ) RETURN INTEGER;
```

```
END;
/

Package created.

gSQL>
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  -- Declare Package Private Function
  FUNCTION func2( p_grade INTEGER, p_option VARCHAR ) RETURN INTEGER;

  -- Define Package Public Function
  FUNCTION func1( p_grade INTEGER, p_option VARCHAR ) RETURN INTEGER AS
    var INTEGER;
  BEGIN
    var := func2( p_grade, p_option );

    RETURN var;
  END;

  -- Define Package Private Function
  FUNCTION func2( p_grade INTEGER, p_option VARCHAR ) RETURN INTEGER AS
    var INTEGER;
  BEGIN
    IF p_option = 'MIN' THEN
      SELECT MIN(c_score) INTO var FROM t_score WHERE c_grade = p_grade;
    ELSIF p_option = 'MAX' THEN
      SELECT MAX(c_score) INTO var FROM t_score WHERE c_grade = p_grade;
    ELSIF p_option = 'AVG' THEN
      SELECT AVG(c_score) INTO var FROM t_score WHERE c_grade = p_grade;
    ELSE
      var := -1;
    END IF;
```

```
    RETURN var;
  END;
END;
/

Package created.

gSQL> SELECT pkg1.func1(  1 , 'MAX' ) FROM DUAL;

PKG1.FUNC1(  1 , 'MAX' )
------------------------
                      99

1 row selected.
```

兼容性

与schema level function相同

参考内容

详细内容参考CREATE FUNCTION

8.21 GOTO Statement

功能

尝试跳到在当前位置可访问的语句中拥有指定标签（label）的最近的语句

语句

```
<goto statement> ::=
    GOTO label_name
    ;
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- Label_name
  - 要跳转的语句的标签名称
  - 可以拥有标识符链形式

说明

跳转到带有该标签名称的语句并开始执行

如果有多个候选语句则跳转到最近的语句

仅可跳转到当前位置可见（存在于重叠的scope中）的语句

前向跳转和后向跳跃均可

使用示例

```
gSQL> DECLARE
  V1 INTEGER := 0;
BEGIN
  <<LABEL1>>
  IF V1 > 0 THEN
    GOTO LABEL2;
  END IF;
  V1 := V1 + 1;
  DBMS_OUTPUT.PUT_LINE('a');
  GOTO LABEL1;
  DBMS_OUTPUT.PUT_LINE('b');
  <<LABEL2>>
  DBMS_OUTPUT.PUT_LINE('c');
END;
/
a
c

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- EXIT Statement（退出声明）
- CONTINUE Statement（CONTINUE 声明）

8.22 IF Statement

功能

执行指定的多个条件中返回TRUE的条件的语句列表

语句

```
<if statement> ::=
    IF <search condition> <if statement then clause>
    [ <if statement elsif clase> ]
    [ <if statement else clase> ]
    END IF
    ;

<if statement then clause> ::=
    THEN <executable statement list>

<if statement elsif clause> ::=
    ELSIF <search condition> THEN <executable statement list>

<if statement elsif clause> ::=
    ELSE <executable statement list>
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

语句规则及参数

- Search condition
  - boolean类型可最终评价的表达式
- Executable statement list
  - SUNDB PSM支持的所有语句的列表

说明

评价与CASE语句类似的条件并执行返回TRUE的IFELSIF子句的语句列表

如果不满足所有条件且存在<if statement else clause>时执行该语句

ELSIF语句按照顺序依次进行评估条件表达式如果条件表达式为TRUE则不评估后续的条件表达式

使用示例

```
gSQL> DECLARE
  V1 INTEGER := 10;
BEGIN
  IF V1 > 0 THEN
```

```
    DBMS_OUTPUT.PUT_LINE( 'POSITIVE' );
  ELSIF V1 = 0 THEN
    DBMS_OUTPUT.PUT_LINE( 'ZERO' );
  ELSE
    DBMS_OUTPUT.PUT_LINE( 'NEGATIVE' );
  END IF;
END;
/
POSITIVE

Anonymous PL block executed.
```

兼容性

<if statement>语句的语句及操作与标准SQL相同

FeatureID	说明	支持状态
P002	Computational completeness	O

Table 8-7 标准SQL 兼容性

8.23 Implicit Cursor Attribute

功能

返回PSM中定义的implicit cursor的状态值

语句

```
<Implict cursor attribute> ::=
    SQL '%' { ISOPEN | FOUND | NOTFOUND | ROWCOUNT }
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

说明

- 存储上次执行的SQL语句的结果
  - ISOPEN： 显示当前游标是否为OPEN状态始终设置为FALSE
  - FOUND： 是否由上一个SQL结果返回数据
  - NOTFOUND： 与FOUND相反

- ROWCOUNT：受上一个SQL结果影响的记录数

使用示例

```
DECLARE
V1 INTEGER;
BEGIN
    SELECT COUNT(*) INTO V1 FROM T1;
    DBMS_OUTPUT.PUT_LINE('COUNT RET    = ' || V1);
    DBMS_OUTPUT.PUT_LINE('SQL%ISOPEN   = ' || SQL%ISOPEN);
    DBMS_OUTPUT.PUT_LINE('SQL%FOUND    = ' || SQL%FOUND);
    DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND = ' || SQL%NOTFOUND);
    DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT = ' || SQL%ROWCOUNT);
END;
/
COUNT RET    = 0
SQL%ISOPEN   = FALSE
SQL%FOUND    = TRUE
SQL%NOTFOUND = FALSE
SQL%ROWCOUNT = 1

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.24 INSERT Statement Extension

功能

为了在VALUES子句中描述PSM支持的record-type变量并输入数据而扩展insert statement的功能

语句

```
<PSM Insert Statement Extension Statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
           <Insert_source>
           [ <Returning_into_clause> ]
    ;

<Insert_source> ::=
      <value-list>
    | <from_subquery>
    | <from_default>


<from subquery> ::=
    <query_expression>


<from default> ::=
    DEFAULT VALUES
```

```
<Value-List> ::=
      VALUES <Value_item> [ , ...]


<value-Item> ::=
     PSM-Record-Type-Variable
     |  ( { <value expression> | DEFAULT } [, ...] )


<Returning_into_clause> ::=
      [ RETURN | RETURNING ] { * | { <value_expression> [ [AS] alias_name ] }
[, ...] INTO variable_name [, ...]
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function）

仅适用于PL block的主区域

不能在EXECUTE IMMEDIATE中的原始SQL语句中使用PSM insert extension语句

语句规则及参数

与insert语句的基本语句的运行方式相同但是除了在value item中使用括号连续列出值表达式的方式之外添加了描述PSM记录类型变量的功能

如果在Value_Item中使用不带括号的变量则必须描述PSM记录类型的变量

使用insert extension语句形式时不能混合使用非record type的变量

说明

除了通常的insert语句之外使用PSM的record type参数保存记录或通过returning into获取结果

Insert的详细内容参考如下

使用示例

```
gSQL> DECLARE
  rec t1%ROWTYPE;
BEGIN
    rec.i1 := 'AAA';
    rec.i2 := 'BBB';
    rec.i3 := 'CCC';

    INSERT INTO t1 (i1, i2, i3) VALUES rec ;
END;
/

Anonymous PL block executed.


gSQL> SELECT * FROM t1;

I1  I2  I3
--- --- ---
AAA BBB CCC
```

兼容性

标准SQL中未定义

8.25 INSERT INTO ... UPDATE Statement Extension

功能

从Insert into .. update statement扩展的功能将PSM支持的record type变量记述在values clause中来生成新的row或者记述到set clause中来更新row

语句

```
<PSM Upsert Statement Extention Statement > ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
        <duplicate key clause>
        [ <Returning_into_clause> ]
    ;

<insert source> ::=
      <values-list>
    | <from subquery>
```

```
    | <from_default>

<Value-List> ::=
      VALUES <Value_item> [ , ...]

<value-Item> ::=
         PSM-Record-Type-Variable
     |  ( { <value expression> | DEFAULT } [, ...] )

<from subquery> ::=
    <query expression>

<from default> ::=
    DEFAULT VALUES

<duplicate key clause>
    ON DUPLICATE KEY { DO NOTHING | <do update clause> }

<do update clause> ::=
    [DO] UPDATE <target-list>

<target-List> ::=
        SET <set clause> [, ...]
      | SET ROW = <psm_variable>

<set clause> ::=
      column_name = { <value expression> | DEFAULT }
    | ( column_name [, ...] ) = ( { <value expression> | DEFAULT } [, ...] )
    | ( column_name [, ...] ) = ( <query expression> )

<returning_into_clause> ::=
      [ RETURN | RETURNING ] { * | { <value_expression> [ [AS] alias_name ] }
[, ...] INTO variable_name [, ...]
```

使用范围及访问权限

仅可在PSM中使用（例：procedure, function, package）

仅适用于PL block的主区域

语句规则及参数

- <values clause>
  - 在Value item中除了将现有的value expression用括号连续排列的方法外还增加了描述PSM record type变量的功能
  - 如果在Value item中使用变量时没有加括号则必须描述PSM record type的变量
  - 将Value item进行扩展时则不能混合使用非record type的变量
- <target list>
  - SET ROW语句中使用的<PSM_Variable>必须是已经声明为record type的变量
- <returning into clause>
  - RETURNING INTO中要使用的<Variable>可以不是record type但是描述为record type时需要与其他数据类型的变量混合使用或者不能排列两个以上的record type变量

说明

Upsert statement的returning into语句在执行insert后会保存执行insert的结果执行update后会保存执行update的结果

使用示例

- 执行insert

```
gSQL> CREATE TABLE t1( c1 INTEGER UNIQUE, c2 INTEGER, c3 INTEGER );

Table created.

gSQL> DECLARE
  v_rec t1%ROWTYPE;
BEGIN
  v_rec.c1 := 1;
  v_rec.c2 := 1;
  v_rec.c3 := 1;

  INSERT INTO t1 VALUES v_rec ON DUPLICATE KEY UPDATE SET c1 = c1 + 1;
END;
/

Anonymous PL block executed.

gSQL> SELECT * FROM t1;

C1 C2 C3
-- -- --
 1  1  1

1 row selected.
```

- 执行update

```
gSQL> CREATE TABLE t1( c1 INTEGER UNIQUE, c2 INTEGER, c3 INTEGER );

Table created.

gSQL> INSERT INTO t1 VALUES ( 1 , 1 , 1 );

1 row created.

gSQL> DECLARE
  v_rec t1%ROWTYPE;
BEGIN
  v_rec.c1 := 2;
  v_rec.c2 := 2;
  v_rec.c3 := 2;

  INSERT INTO t1 VALUES (1, 1, 1) ON DUPLICATE KEY UPDATE SET ROW = v_rec;
END;
/

gSQL> SELECT * FROM t1;

C1 C2 C3
-- -- --
 2  2  2

1 row selected.
```

- 执行insert时returning语句

```
gSQL> CREATE TABLE t1( c1 INTEGER UNIQUE, c2 INTEGER, c3 INTEGER );

Table created.
```

```
gSQL> DECLARE
  v_rec1 t1%ROWTYPE;
  v_rec2 t1%ROWTYPE;
BEGIN
  v_rec1.c1 := 1;
  v_rec1.c2 := 1;
  v_rec1.c3 := 1;

  INSERT INTO t1 VALUES v_rec1 ON DUPLICATE KEY UPDATE SET c1 = c1 + 1 RETURNING
* INTO v_rec2;
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c1 : ' || v_rec2.c1 );
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c2 : ' || v_rec2.c2 );
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c3 : ' || v_rec2.c3 );
END;
/

v_rec2.c1 : 1
v_rec2.c2 : 1
v_rec2.c3 : 1
Anonymous PL block executed.

gSQL> SELECT * FROM t1;

C1 C2 C3
-- -- --
 1  1  1

1 row selected.
```

- 执行update时returning语句

```
gSQL> CREATE TABLE t1( c1 INTEGER UNIQUE, c2 INTEGER, c3 INTEGER );
```

```
Table created.

gSQL> INSERT INTO t1 VALUES ( 1 , 1 , 1 );

1 row created.

gSQL> DECLARE
  v_rec1 t1%ROWTYPE;
  v_rec2 t1%ROWTYPE;
BEGIN
  v_rec1.c1 := 2;
  v_rec1.c2 := 2;
  v_rec1.c3 := 2;

  INSERT INTO t1 VALUES (1, 1, 1) ON DUPLICATE KEY UPDATE SET ROW = v_rec1
RETURNING * INTO v_rec2;
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c1 : ' || v_rec2.c1 );
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c2 : ' || v_rec2.c2 );
  DBMS_OUTPUT.PUT_LINE( 'v_rec2.c3 : ' || v_rec2.c3 );
END;
/

v_rec2.c1 : 2
v_rec2.c2 : 2
v_rec2.c3 : 2
Anonymous PL block executed.

gSQL> SELECT * FROM t1;

C1 C2 C3
-- -- --
 2  2  2
```

```
1 row selected.
```

兼容性

标准SQL未定义

8.26 Explicit Cursor Attribute

功能

返回PSM中定义的游标的状态值

语句

```
<Explicit cursor attribute> ::=
    cursor_name '%' { ISOPEN | FOUND | NOTFOUND | ROWCOUNT }
```

使用范围及访问权限

只能在PSM中的主区域使用

语句规则及参数

- Cursor_name
  - 想要查看其状态值的游标名称

说明

- 返回指定游标的状态值
  - ISOPEN：当前游标是否为OPEN状态
  - FOUND：是否由最近的fetch返回数据
  - NOTFOUND：与FOUND相反
  - ROWCOUNT：自最近打开游标以来获取的记录数
- 根据游标的状态返回以下值

Attribute名称	OPEN 前	OPEN 后	FETCH 后	CLOSE 后
ISOPEN	FALSE	TRUE	TRUE	FALSE
FOUND	NULL	NULL	TRUE/FALSE	NULL
NOTFOUND	NULL	NULL	TRUE/FALSE	NULL
ROWCOUNT	NULL	NULL	N(数量)	NULL

Table 8-8 根据执行时间点的结果

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
  CURSOR C1 IS SELECT * FROM T1;
  V1 INTEGER := 0;
  V2 INTEGER := 0;
  TOTAL INTEGER := 0;
BEGIN
  FOR I IN 1..100 LOOP
    INSERT INTO T1 VALUES( I );
  END LOOP;

  COMMIT;

  IF NOT C1%ISOPEN THEN
    OPEN C1;
  END IF;

  LOOP
    FETCH C1 INTO V1;
    EXIT WHEN C1%NOTFOUND;

    TOTAL := TOTAL + V1;
```

```
    V2 := V1;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( 'COUNT = ' || C1%ROWCOUNT || ' TOTAL = ' || TOTAL );

  CLOSE C1;

END;
/

COUNT = 100 TOTAL = 5050

Anonymous PL block executed.
```

兼容性

标准SQL未定义

8.27 NULL Statement

功能

无任何功能的statement

语句

```
<null statement> ::=
    NULL
    ;
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅适用于PL block的主区域

说明

无任何功能的statement主要用于设置特定位置的标签

使用示例

```
gSQL> BEGIN
  FOR i in 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE( i );
    IF i > 5 THEN
      GOTO label1;
    END IF;
  END LOOP;
  <<label1>>
  NULL;
END;
/
1
2
3
4
5
6

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.28 OPEN Statement

功能

执行PSM中定义的游标的SELECT语句

语句

```
<open statement> ::=
    OPEN cursor_name [ <actual param spec> ]
    ;

<actual param spec> ::=
      ( expression [ , expression ] .. )
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅适用于PL block的主区域

语句规则及参数

- Cursor_name
  - 要打开的游标的名称

说明

执行定义的游标的SELECT或SELECT ... FOR UPDATE语句

如果仅声明游标而未定义则会报错

实际参数值必须与相应参数的数据类型相互兼容

实际参数的数量必须等于游标的参数数量

如果比游标的Parameter少其余的parameter都必须指定DEFAULT值

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I2 VARCHAR(10) );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> INSERT INTO T1 VALUES( 1, 'AAA' );

1 row created.
```

```
gSQL> INSERT INTO T1 VALUES( 2, 'BBB' );

1 row created.

gSQL> INSERT INTO T1 VALUES( 3, 'CCC' );

1 row created.

gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
  CURSOR C1( A1 INTEGER := 1, A2 VARCHAR DEFAULT 'AAA') IS SELECT * FROM T1
WHERE I1 = A1 AND I2 = A2;
  V1 INTEGER;
  V2 VARCHAR(10);
BEGIN
  OPEN C1( 1 );
  FETCH C1 INTO V1, V2;
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 || ' V2 = ' || V2 );

  CLOSE C1;
END;
/

V1 = 1 V2 = AAA

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- CLOSE Statement（CLOSE 声明）
- FETCH Statement

8.29 OPEN FOR Statement

功能

通过PSM中定义的cursor变量执行SELECT语句后打开一个游标

语句

```
<open statement> ::=
    OPEN cursor_variable_name FOR <cursor_query>
    ;

<cursor_query> ::==
      <static_cursor_query>
    | <dynamic_cursor_query> [ <using_clause> ]

<static_cursor_query> ::==
      select_statement
    | select_for_update_statement
    | insert_returning_query_statement
    | update_returning_query_statement
    | delete_returning_query_statement

<dynamic_cursor_query>
      single_quote_string
    | psm_variable
```

```
<using_clause> ::=
    USING [ <bind_type> ] <expression> [ {, [ <bind_type> ] <expression>} ... ]

<bind_type> ::=
     IN
   | OUT
   | IN OUT
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅适用于PL block的主区域

语句规则及参数

- <cursor_variable_name>
  - cursor_variable的名称
- <cursor_query>
  - cursor query可使用静态SQL和动态SQL
  - cursor query如下
    - select语句
    - select for update语句
    - insert returning语句
    - update returning语句
    - delete returning语句
  - 动态游标查询（dynamic cursor query）可以通过用单引号(')将sql括起来或者将sql保存到psm variable进行使用

- 以动态游标查询编写时可以使用using子句

- <using_clause>
  - 当编写动态游标查询时存在bind variable时在using子句中列出与bind variable数量一样多的variable或expression
  - Using子句的variable可以描述IN/ OUT/ IN OUT的bind type编写动态游标查询时bind type须与bind variable的bind type一致
  - 可以省略using子句的variable的bind type省略时default bind type为IN

说明

执行定义的cursor variable的cursor query

如果cursor variable在之前已经是open的状态则自动close后进行reopen后执行

使用示例

```
gSQL> CREATE TABLE t1( c1 INTEGER, c2 VARCHAR( 6 ) );

Table created.

gSQL> COMMIT;

Commit complete.
```

- Insert Returning Statement

```
gSQL>
DECLARE
```

```
   var1 t1%ROWTYPE;
   curvar1 SYS_REFCURSOR;
BEGIN
   OPEN curvar1 FOR INSERT INTO t1 VALUES ( 1 , 'aaa' ) RETURNING *;

   FETCH curvar1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE curvar1;
END;
/

var1.c1 = 1 , var1.c2 = aaa
Anonymous PL block executed.
```

- Select Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   curvar1 SYS_REFCURSOR;
BEGIN
   OPEN curvar1 FOR SELECT * FROM t1;

   FETCH curvar1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE curvar1;
END;
/
```

```
var1.c1 = 1 , var1.c2 = aaa
Anonymous PL block executed.
```

- Update Returning Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   curvar1 SYS_REFCURSOR;
BEGIN
   OPEN curvar1 FOR UPDATE t1 SET c1 = 3, c2 = 'ccc' RETURNING *;

   FETCH curvar1 INTO var1;
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE curvar1;
END;
/

var1.c1 = 3 , var1.c2 = ccc
Anonymous PL block executed.
```

- Delete Returning Statement

```
gSQL>
DECLARE
   var1 t1%ROWTYPE;
   curvar1 SYS_REFCURSOR;
BEGIN
   OPEN curvar1 FOR DELETE FROM t1 RETURNING *;

   FETCH curvar1 INTO var1;
```

```
   DBMS_OUTPUT.PUT_LINE( 'var1.c1 = ' || var1.c1 || ' , var1.c2 = ' ||
var1.c2 );

   CLOSE curvar1;
END;
/

var1.c1 = 3 , var1.c2 = ccc
Anonymous PL block executed.
```

- Dynamic Cursor Query和Using Clause

```
gSQL> INSERT INTO t1 VALUES( 100 , 'BBB' );
gSQL> COMMIT;

gSQL>
DECLARE
  v_int_in     INTEGER;
  v_varchar_in VARCHAR(6);
  v_out        t1%ROWTYPE;
  v_sql        VARCHAR(1024);
  cv           SYS_REFCURSOR;
BEGIN
  v_sql := 'SELECT * FROM t1 WHERE c1 = ? AND c2 = ?';

  v_int_in := 100;
  v_varchar_in := 'BBB';

  OPEN cv FOR v_sql USING IN v_int_in, v_varchar_in;

  FETCH cv INTO v_out;
  DBMS_OUTPUT.PUT_LINE( 'c1 : ' || v_out.c1 || ' , v_out.c2 : ' || v_out.c2 );
```

```
  CLOSE cv;
END;
/

c1 : 100 , v_out.c2 : BBB
Anonymous PL block executed.
```

兼容性

标准SQL中未定义

参考内容

详细内容参考如下

- Cursor Variable Declaration（游标变量声明）
- FETCH Statement
- CLOSE Statement（CLOSE 声明）

8.30 Procedure Call

功能

调用用户自定义procedurebuilt-in procedure或nested procedure

语句

```
<Procedure call> ::=
    proc_name [ ( expr { , expr } ... ) ]
    ;
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅适用于PL block的主区域

- 调用用户自定义procedure时应满足该procedure的以下权限中的一个
  - EXECUTE PROCEDURE
  - 对于procedure所属的schema的（EXECUTE PROCEDURE或CONTROL SCHEMA）ON SCHEMA
  - EXECUTE ANY PROCEDURE ON DATABASE

语句规则及参数

- Proc_name
  - 要执行的procedure的名称可如下使用

形式	语句	说明
单一identifier	procedure_name	调用指定名称的procedure 搜索顺序如下 1. nested procedure 2. schema-level procedure
identifier chain	lable_name.procedure_name	调用nested procedure
	schema_name.procedure_name	调用schema-level procedure
	package_name.procedure_name	调用built-in procedure

以指定的名称（proc_name）搜索后第一个发现的procedure的指定参数和数量及类型不合适时不会搜索其他procedure并且报错

说明

调用之前定义的用户自定义procedurebuilt-in procedure或nested procedure

调用时可省略定义默认值的参数值

在指定为OUT或IN-OUT的参数中使用procedure变量或bind parameter (?, :V1 等)可以得到返回的值

使用示例

```
gSQL> DECLARE
  PROCEDURE PROC1( A1 INTEGER )
    IS
    BEGIN
      PUT_LINE( 'A1 = ' || A1 );
    END;
BEGIN
  PROC1( 100 );
END;
/
A1 = 100

Anonymous PL block executed.
```

兼容性

标准SQL指定使用<call statement>

8.31 Procedure Declaration and Definition

功能

声明并定义procedure

语句

```
<procedure declaration> ::=
      PROCEDURE <procedure name> [ ( <parameter list> ) ]
      ;

<procedure definition> ::=
      PROCEDURE <procedure name> [ ( <parameter list> ) ]
      { IS | AS }
      <item declaration>
      BEGIN
      <pl statement list>
      END [ <procedure name> ]
      ;

<parameter list> ::=
      <parameter name> [ <parameter mode> ] <datatype> [ <parameter default> ]
[ , ... ]

<parameter mode> ::=
      IN
```

```
    | OUT
    | IN OUT

<parameter default> ::=
      { := | DEFAULT } <value expression>
```

使用范围及访问权限

仅可在PSM中使用(例：package, procedure, function)

仅适用于PL block的声明领域

语句规则及参数

procedure name

作为PL块中要创建的procedure的名称应在PL块内是唯一的名称

即PL块中声明的PL Item和procedure不能有相同的名称

Procedure名称的长度应小于128个字节

parameter name

定义Procedure的参数名称

各参数应在procedure中拥有唯一的名称

即procedure的参数和PL item不能有相同的名称

参数名称的长度应小于128 个字节

单个procedure可用参数的最大数量没有限制

parameter mode

设置各参数模式

参数模式为IN, OUT, IN OUT

未指定参数模式时默认模式为IN

parameter default

参数的默认值

执行procedure时可省略指定了参数默认值的参数

未指定并省略参数时默认值为定义参数时指定的<value expression>

<value expression>的数据类型应为参数的数据类型

在有<parameter default>的参数以后定义的所有参数须有<parameter default>

Item Declaration

声明Procedure内部使用的本地变量等的item

可以声明可在PL块中可声明的所有item

PL Stmt List

Procedure的主体部分列出要执行的PL statement

说明

如下声明和定义procedure

- PL块(例：匿名块procedure及函数的块块语句的块)
  - 作为PL块的item中之一声明并定义procedure
  - 在PL块中声明并定义的procedure称为嵌套程序（nested procedure）
  - 嵌套程序仅可在PL块范围内使用
- 包规格（Package specification）
  - 声明包的公共程序（public procedure）
  - 包的公共程序可在数据库内使用
- 包主体（Package body）
  - 定义在包规格中声明的公共程序
  - 声明并定义包的私有程序（private procedure）
  - 包的私有程序仅可在包范围内使用

Procedure的使用方法与schema-level procedure相同

使用示例

- 嵌套程序

```
gSQL>
DECLARE
  PROCEDURE proc1( p1 INTEGER ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'p1 = ' || p1 );
  END;
BEGIN
  proc1( 100 );
END;
/
```

```
p1 = 100
Anonymous PL block executed.
```

- 包程序

```
gSQL>
CREATE OR REPLACE PACKAGE pkg1 AS
  -- Declare Package Public Function
  PROCEDURE proc1( p1 INTEGER );
END;
/

Package created.

gSQL>
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  -- Declare Package Private Function
  PROCEDURE proc2( p1 INTEGER );

  -- Define Package Public Function
  PROCEDURE proc1( p1 INTEGER ) AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'p1 of proc1 : ' || p1 );

    proc2( p1 * p1 );
  END;

  -- Define Package Private Function
  PROCEDURE proc2( p1 INTEGER ) AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE( 'p1 of proc2 : ' || p1 );
  END;
END;
```

```
/

Package created.

gSQL> CALL pkg1.proc1( 10 );

p1 of proc1 : 10
p1 of proc2 : 100
Procedure Call complete.
```

兼容性

与schema level procedure相同

参考内容

详细内容参考 CREATE PROCEDURE

8.32 RAISE Statement

功能

显式引发用户定义的异常

语句

```
<RAISE Statement> ::=
    RAISE  <Exception-Name>
    ;
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅适用于PL block的主区域

语句规则及参数

- 需要raise的异常名应声明在raise所属的PL块或在上层PL块的DECLARE子句但是预定义异常可在未声明异常的情况下进行raise
- 如果在exception handler中使用raise语句则可以省略exception name此时上一个exception将传播到父

块

说明

如果未处理产生异常的PL块则将其传播到上层PL块

如果要raise的异常不在包含raise语句的PL块及上层PL块内的所有异常处理程序中时报错

RAISE exception从PL块传播到上层直到它被处理并且它不会传播到下层PL块的exception handler

Raise exception	Exception handler SCOPE	Exception handler	顶部传播
User exception without error code	相同范围	X	"unhandled exception" 错误传播到上级范围
User exception with errorcode	上级范围	X	传播user exception
User exception with errorcode	相同范围	X	传播用户定义的error code
User exception with error code	上级范围	X	传播用户定义的error code

Table 8-9 用户异常的传播

使用示例

```
gSQL> DECLARE
V1 INTEGER;
exception1   EXCEPTION;
exception100 EXCEPTION;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Step1');
    BEGIN
        RAISE exception1;
```

```
        DBMS_OUTPUT.PUT_LINE('Step2');
        EXCEPTION WHEN Exception100 THEN DBMS_OUTPUT.PUT_LINE('in Exception');
    END;
    DBMS_OUTPUT.PUT_LINE('Step3');
    EXCEPTION WHEN Exception1 THEN DBMS_OUTPUT.PUT_LINE('out Exception');
    DBMS_OUTPUT.PUT_LINE('Step4');
END;
/
Step1
out Exception
Step4

Anonymous PL block executed.
```

兼容性

SQL标准描述了<handler declaration>和<condition declaration>但不支持语句

参考内容

详细内容参考如下

- Exception Handler（异常处理程序）
- Exception Declaration（例外声明）

8.33 Record Variable Declaration

功能

在DECLARE区域中声明record type变量

语句

```
<declare record variable> ::=
    variable_name <recordType>
    ;

<recordType> ::=
     <tableName>%ROWTYPE
   | USER_DEFINED_DATA_TYPE
```

使用范围及访问权限

仅可在PSM内可用(例: package, procedure, function)

仅适用于PL block的声明的区域

语句规则及参数

- Variable_name
  - 要声明的变量的名称
  - 变量名的长度应小于128个字节
  - 应为在一个scope（PL block body）中的唯一的名称
  - 可以在重叠的下级PL block body中声明具有相同名称的变量
  - 为了明确使用重复声明的变量需使用scope_name.variable_name形式
  - 如果未指定范围名称并使用重复声明的变量则会自动使用最近的范围变量
- Record_type
  - 可以使用%ROWTYPE及用户定义的recordType
  - 关于recordType声明的详细内容参考User-defined Record Type

说明

- 声明的变量具有以下特征
  - 应为在一个scope（PL block body）中的唯一名称
  - 可以在重叠的下级PL block body中声明具有相同名称的变量
  - 为了明确使用重复声明的变量需使用scope_name.variable_name形式
  - 如果未明示范围名称并使用重复声明的变量则会自动使用最近的范围变量
- 声明的变量在相应的范围和下级范围中使用并且与嵌入式SQL不同不使用 ':' 符号
- 使用的变量对相应SQL或PSM控制语句以绑定参数（INOUT）运行

使用示例

```
gSQL> DECLARE
  TYPE MY_REC1 IS RECORD ( F1 INTEGER, F2 VARCHAR(10) );
  V1 MY_REC1;
BEGIN
  V1.F1 := 1;
  V1.F2 := 'AAA';
  INSERT INTO T1 VALUES( V1.F1, V1.F2 );
END;
/

Anonymous PL block executed.


gSQL> COMMIT;

Commit complete.


gSQL> SELECT * FROM T1;

I1 I2
-- ---
 1 AAA

1 row selected.
```

兼容性

标准SQL中未定义

8.34 RETURN Statement

功能

指定函数应返回的值后终止该函数Procedure不指定返回值并结束该procedure

语句

```
<return statement> ::=
    RETURN [ return_value_expr ]
    ;
```

使用范围及访问权限

仅在PSM内部可用(例: package, procedure, function)

仅适用于PL block的主区域

不可在table function的PL block中使用

语句规则及参数

- Return_value_expr
  - 要返回的值的表达式仅可在为函数时指定

说明

终止当前在执行的procedure/ function

对于函数（function）如果未执行并结束RETURN子句或者RETURN子句没有return_value_expr则会报错

不可在table function中使用

使用示例

```
gSQL> CREATE OR REPLACE FUNCTION FUNC1( A1 INTEGER, A2 INTEGER )

  RETURN INTEGER
  IS
    V1 INTEGER;
  BEGIN

    SELECT COUNT(*)
      INTO V1
```

```
      FROM T1
      WHERE T1.I1 >= A1 AND T1.I1 <= A2;

    RETURN V1;
  END;
  /

Function created.
```

兼容性

在标准SQL中进行了描述但是没有遵循法则（conformance rule）

8.35 RETURN TABLE Statement

功能

返回执行游标变量（cursor variable）的cursor query或select statement的结果集后终止函数

语句

```
<return table statement> ::=
    RETURN TABLE ( { <cursor variable name> | <select statement> } )
    ;
```

使用范围及访问权限

仅可在PSM中table function中使用

仅可在table function块的主区域使用

语句规则及参数

- <cursor variable name>
  - 返回执行指定的游标变量的cursor query的结果集
  - 游标变量必须open

- 游标变量的cursor query不能执行过fetch
- 仅允许使用Cursor query为选择语句的游标变量

- <select statement>
  - 仅可使用选择语句
  - 不可使用select for update statement, select into statement

说明

返回执行在RETURN TABLE语句中指定的游标变量的cursor query或选择语句的结果集

RETURN TABLE语句仅可在table function中使用

使用示例

```
gSQL> CREATE TABLE t_score( c_grade INTEGER, c_score INTEGER );

Table created.

gSQL> INSERT INTO t_score VALUES ( 1 , 98 ) , ( 1 , 97 ) , ( 1 , 99 ),
                                 ( 2 , 95 ) , ( 2 , 98 ) , ( 2 , 92 ),
                                 ( 3 , 98 ) , ( 3 , 96 ) , ( 3 , 94 );

9 rows created.

gSQL> COMMIT;

Commit complete.
```

- 游标变量示例

```
gSQL>
CREATE FUNCTION func_cursor_variable( p_option VARCHAR )
  RETURN TABLE( f_class NUMBER, f_score NUMBER )
AS
  s_cur SYS_REFCURSOR;
BEGIN
  IF p_option = 'MIN' THEN
    OPEN s_cur FOR SELECT c_grade, MIN(c_score) FROM t_score GROUP BY c_grade;
  ELSIF p_option = 'MAX' THEN
    OPEN s_cur FOR SELECT c_grade, MAX(c_score) FROM t_score GROUP BY c_grade;
  ELSIF p_option = 'AVG' THEN
    OPEN s_cur FOR SELECT c_grade, AVG(c_score) FROM t_score GROUP BY c_grade;
  ELSE
    OPEN s_cur FOR SELECT NULL, NULL FROM dual;
  END IF;

  RETURN TABLE( s_cur );
END;
/

Function created.

gSQL> COMMIT;

Commit complete.

-- 计算各学年的最低分数
gSQL> SELECT * FROM TABLE( func_cursor_variable( 'MIN' ) );

F_CLASS F_SCORE
------- -------
      1      97
      2      92
```

```
      3      94


3 rows selected.
```

- 选择语句示例

```
gSQL>
CREATE FUNCTION func_select( p_option VARCHAR )
    RETURN TABLE( f_class NUMBER, f_score NUMBER )
AS
BEGIN
  IF p_option = 'MIN' THEN
    RETURN TABLE ( SELECT c_grade, MIN(c_score) FROM t_score GROUP BY c_grade );
  ELSIF p_option = 'MAX' THEN
    RETURN TABLE ( SELECT c_grade, MAX(c_score) FROM t_score GROUP BY c_grade );
  ELSIF p_option = 'AVG' THEN
    RETURN TABLE ( SELECT c_grade, AVG(c_score) FROM t_score GROUP BY c_grade );
  ELSE
    RETURN TABLE ( SELECT NULL, NULL FROM dual );
  END IF;
END;
/

Function created.

-- 计算各学年的平均分数
gSQL> SELECT * FROM TABLE( func_select( 'AVG' ) );

F_CLASS F_SCORE
------- -------
      1      98
      2      95
      3      96
```

```
3 rows selected.
```

兼容性

标准SQL在<return statement> 语句中进行了描述

但是标准SQL未定义游标变量

8.36 RETURNING INTO clause

功能

Insert/ update/ delete处理的数据作为PSM变量返回

语句

```
<Insert, Delete Returning_into_clause> ::=
      [ RETURN | RETURNING ] { * | { <value_expression> [ [AS] alias_name ] }
[, ...] INTO variable_name [, ...]

<Update returning into clause> ::=
    { RETURN | RETURNING } [ NEW | OLD ] { * | { <value expression> [ [AS]
alias_name] } [, ...] } INTO Variable [, ...]
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅可在PL block的主区域使用

语句规则及参数

记录类型变量不能与其他类型混合使用

说明

通过returning into保存处理Insert / Update / Delete语句的before/ after record

使用示例

```
gSQL> DECLARE
  rec t1%ROWTYPE;
BEGIN
    INSERT INTO t1 VALUES (1, 2, 3) RETURNING * INTO rec ;
    UPDATE T1 SET ROW = rec RETURNING * INTO rec;
    DELETE FROM T1 RETURNING * INTO rec;
END;
/

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.37 %ROWTYPE Attribute

功能

声明变量时将其定义为与特定表或游标或游标变量的result set相同的结构和类型

语句

```
<rowtype attribute> ::=
    <identifier chain> % ROWTYPE
```

使用范围及访问权限

- 仅在PSM内可用(例: package, procedure, function)
- 仅在PL block的声明区域中可用
- 声明变量时仅可在<data type>部分使用
- 声明记录类型的字段时不能用于<data type>部分（不支持复杂数据类型）

语句规则及参数

- 标识符链如下
  - 要引用的表视图同义词的名称或游标或游标变量的名称

说明

- 参考对象搜索顺序
  - 游标或游标变量
  - base tableview或synonym
- 参考范围
  - 使用行属性进行引用时在表或游标的result set中仅引用列名和类型
  - 因此不引用NOT NULL constraint或DEFAULT值设置等

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER NOT NULL, I2 VARCHAR(10) );

Table created.

gSQL> INSERT INTO T1 VALUES( 123, '1234567890' );

1 row created.


gSQL> COMMIT;

Commit complete.


gSQL> DECLARE
  V1 T1%ROWTYPE;
BEGIN
```

```
  SELECT * INTO V1.I1, V1.I2 FROM T1;
  DBMS_OUTPUT.PUT_LINE( 'V1.I1 = ' || V1.I1 || ' V1.I2 = ' || V1.I2 );
END;
/
V1.I1 = 123 V1.I2 = 1234567890

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.38 Scalar Variable Declaration

功能

在DECLARE区域中声明scalar变量

语句

```
<declare scalar variable> ::=
    variable_name <data type> [ <variable initialize clause> ]
    ;

<variable initialize clause> ::=
      [ NOT NULL ] { DEFAULT | := } <value expression>
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅在PL block的声明区域中可用

语句规则及参数

- Variable_name
  - 要声明的变量的名称
  - 变量名的长度应小于128个字节
  - 应为在一个Scope（PL Block Body）中的唯一的名称
  - 可以在重叠的下级PL Block Body声明具有相同名称的变量
  - 为了明确使用重复声明的变量需使用scope_name.variable_name形式
  - 如果未指定范围名称并使用重复声明的变量则会自动使用最近的范围变量
- Data_Type
  - 可使用SUNDB提供的所有built-in **数据类型**
- Value_expression
  - 表示在变量指定的初始值
  - 可使用除了SUNDB支持的所有常数及多行函数外的所有表达式

说明

- 声明的变量具有以下特征
  - 应为在一个scope（PL block body）中唯一的名称
  - 可以在重叠的下级PL block body中声明具有相同名称的变量
  - 要明确使用重复声明的变量需使用scope_name.variable_name形式
  - 如果使用未指定范围名称的重复声明的变量则会自动使用最近的范围变量
- 声明的变量在相应的范围和下级范围中使用并且与嵌入式SQL不同不使用'：'符号
- 使用的变量对相应SQL或PSM控制语句以绑定参数（INOUT）操作

使用示例

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I2 VARCHAR(10) );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> DECLARE
V1 INTEGER := 100;
V2 INTEGER := -100;
V3 VARCHAR(10) := 'ABC';
BEGIN
  IF V1 > 50 THEN
    INSERT INTO T1 VALUES ( V1, V3 );
  ELSE
    INSERT INTO T1 VALUES ( V2, V3 );
  END IF;
END;
/

Anonymous PL block executed.

gSQL> SELECT * FROM T1;

 I1 I2
--- ---
100 ABC
```

```
1 row selected.
```

兼容性

- <declare scalar variable>语句与标准SQL有如下区别
  - 标准SQL的<SQL变量声明>在PL block主体中(在BEGIN之后)声明变量但SUNDB单独在声明部分进行声明
  - 标准SQL可以通过单个DECLARE语句声明多个相同类型的变量但在SUNDB一个语句只能声明一个变量
  - 标准SQL在设置初始值时仅使用DEFAULT语句但SUNDB还可以使用 := 分配符号

8.39 SELECT INTO Statement

功能

通过SELECT返回一行

语句

```
<select statement: single row> ::=
    SELECT [ <hint clause> ] [ <set quantifier> ] <select list>
        INTO <select target list>
        <table expression>
    ;

<select target list> ::=
    variable_name [, ...]
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅可在PL block的主区域内使用

语句规则及参数

除INTO子句外遵循与Select语句相同的规则

说明

- SELECT INTO用于返回一条记录
  - 如果结果为0则引发“NO_DATA_FOUND”异常
  - 如果有两条以上的结果则引发“TOO_MANY_ROWS”异常
- 可以通过PSM的记录类型变量获取结果
  - 如果使用记录类型变量则不能与其他类型的变量混合使用

使用示例

```
gSQL> CREATE TABLE T1 (c1 VARCHAR(20), c2 VARCHAR(20));

Table created.

gSQL> INSERT INTO T1 VALUES ('AAA', 'BBB'), ('BBB', 'CCC');

2 rows created.

gSQL> DECLARE
  v1 VARCHAR(20);
  v2 VARCHAR(20);
BEGIN
```

```
  SELECT * INTO v1, v2 FROM T1 WHERE c1 = 'AAA';
  DBMS_OUTPUT.PUT_LINE('V1=' || v1 || ', v2=' || v2);
END;
/
V1=AAA, v2=BBB

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.40 SQLCODE Function

功能

返回在PSM中执行的上一个statement的错误代码

语句

```
<SQLCODE function> ::= SQLCODE
```

使用范围及访问权限

仅在PSM内可用

语句规则及参数

没有单独的argument

说明

返回在PL/ SQL中执行的语句的错误代码

未分配错误代码的user-defined exception在被处理程序处理时返回1

当异常处理程序完成错误处理后返回0

使用示例

```
gSQL> DECLARE
V1 INTEGER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('SQLCODE=[' || SQLCODE || ']');
    DBMS_OUTPUT.PUT_LINE('SQLERRM=[' || SQLERRM || ']');
END;
/
SQLCODE=[0]
SQLERRM=[[CSII][PL/SQL][SUNDB]successful completion]

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.41 SQLERRM Function

功能

返回在PSM中执行的上一个statement的错误消息

语句

```
<SQLERRM function> ::= SQLERRM
```

使用范围及访问权限

仅在PSM内可用

语句规则及参数

没有单独的argument

说明

返回在PL/ SQL中执行的语句的错误消息

对于未分配Error code的user-defined exception在处理程序处理时返回user-defined exception

当异常处理程序完成错误处理后将显示成功完成（successful completion）消息

使用示例

```
gSQL> DECLARE

V1 INTEGER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('SQLCODE=[' || SQLCODE || ']');
    DBMS_OUTPUT.PUT_LINE('SQLERRM=[' || SQLERRM || ']');
END;
/
SQLCODE=[0]
SQLERRM=[[CSII][PL/SQL][SUNDB]successful completion]

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.42 %TYPE Attribute

功能

声明变量或定义RECORD类型的特定字段时可以将其类型定义为特定表的列或与其他变量相同的类型

语句

```
<type attribute> ::=
    <identifier chain> % TYPE
```

使用范围及访问权限

仅在PSM内可用(例: package, procedure, function)

仅在PL block的声明区域中可用

在声明变量或声明记录类型的字段时仅可用于<data type>部分

语句规则及参数

- Identifier_chain
  - 要引用表的列或现有声明变量（或变量字段）的名称

说明

参考范围

根据引用的对象类型的引用范围如下

- 当引用的对象是表的列时
  - 参照列的数据类型
  - 不参照列的constraint（例如：NOT NULL）
  - 不参照列的默认初始值
- 当引用的对象是其他变量（或变量的字段）时
  - 参照变量（或字段）的数据类型
  - 参照变量（或字段）的约束（NOT NULL）
  - 不参照变量（或字段）的默认初始值
- 引用对象的搜索顺序如下
  1. 变量（或字段）名称
  2. 列名

NOT NULL Constraints变量引用

当引用NOT NULL属性的变量时不会引用初始值因此必须指定新的初始值

使用当前记录类型的字段的type attribute时不支持字段初始值设置因此无法引用NOT NULL类型的字段

使用示例

```
gSQL> DECLARE
```

```
  V1 NUMBER(5,2) := 100.01;
  V2 V1%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
  DBMS_OUTPUT.PUT_LINE( 'V2 = ' || V2 );
END;
/
V1 = 100.01
V2 =

Anonymous PL block executed.
```

兼容性

标准SQL中未定义

8.43 UPDATE Statement Extension

功能

在PSM中除了列出UPDATE语句的更改对象列的方式外还可以使用record type变量来更改记录

在PSM中在UPDATE（searched）RETURNING INTO子句使用record type变量存储结果

语句

```
<update Extension statement : searched> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        <target-list>
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        [ <returning into clause> ]
    ;

<update statement: positioned> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        <Target-List>
        WHERE CURRENT OF cursor_name
    ;

<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]
```

```
<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>


<fetch first clause> ::=
    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]


<limit clause>
    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }

<returning into clause> ::=
    { RETURN | RETURNING } [ NEW | OLD ] { * | { <value expression> [ [AS]
alias_name] } [, ...] } INTO Variable [, ...]

<target-List> ::=
        SET <set clause> [, ...]
      | SET ROW = <psm_variable>

<set clause> ::=
      column_name = { <value expression> | DEFAULT }
    | ( column_name [, ...] ) = ( { <value expression> | DEFAULT } [, ...] )
    | ( column_name [, ...] ) = ( <query expression> )
```

使用范围及访问权限

仅在PSM中可用（例如：package, procedure, function ）

仅可在PL block的主区域使用

语句规则及参数

- 在UPDATE SET ROW语句中使用的<PSM_Variable>必须是声明为record type的变量
- UPDATE RETURNING INTO语句中使用的<Variable>可以不是record type
    - 但是如果将其描述为record type则不能与其他数据类型变量混合使用也不能列出两个或更多record type变量

说明

在PSM中通过record type变量更改记录或保存RETURNING INTO的结果

使用示例

```
gSQL> CREATE TABLE T1 (C1 VARCHAR(20), C2 VARCHAR(20));
Table created.

gSQL> INSERT INTO T1 VALUES ('AAA', 'BBB'), ('BBB', 'CCC'), ('CCC', 'DDD');
3 rows created.


gSQL> DECLARE
    v1 t1%ROWTYPE;
    v2 t1%ROWTYPE;
BEGIN

    v1.c1 := '1';
```

```
    v1.c2 := '2';

    UPDATE T1 SET ROW = v1 WHERE c1 = 'AAA' RETURNING * INTO v2;
    DBMS_OUTPUT.PUT_LINE('SQL%ROWCOUNT=' || SQL%ROWCOUNT );
    DBMS_OUTPUT.PUT_LINE('v2.c1=' || v2.c1 || ', v2.c2=' || v2.c2);
END;
/
SQL%ROWCOUNT=1
v2.c1=1, v2.c2=2

Anonymous PL block executed.

gSQL> SELECT * FROM T1 ORDER BY C1;

C1  C2
--- ---
1   2
BBB CCC
CCC DDD

3 rows selected.
```

兼容性

标准SQL中未定义

8.44 WHILE LOOP Statement

功能

当<search condition>返回TRUE时执行内部的语句

语句

```
<while loop statement> ::=
    WHILE <search condition>
    LOOP { <SQL procedure statement> ; }... END LOOP [ loop_name ]
    ;
```

使用范围及访问权限

仅在PSM内可用（例: package, procedure, function）

仅可在PL block的主区域内使用

语句规则及参数

- 搜索条件如下
  - 持续循环while loop的条件表达式

- 最终应返回boolean类型

描述

当<搜索条件>的评估结果为TRUE时while循环语句会执行内部的语句列表

使用示例

```
gSQL> DECLARE
V1 integer := 0;
BEGIN
  WHILE V1 < 10 LOOP
    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
    V1 := V1 + 1;
  END LOOP;
END;
/
V1 = 0
V1 = 1
V1 = 2
V1 = 3
V1 = 4
V1 = 5
V1 = 6
V1 = 7
V1 = 8
V1 = 9
```

```
Anonymous PL block executed.
```

兼容性

在标准<SQL中<while loop statement>语句被定义为<while statement>

标准SQL的< while statement>以DO ... END WHILE执行loop语句而SUNDB以LOOP ... END LOOP执行

参考内容

详细内容参考如下

- CONTINUE Statement（CONTINUE 声明）
- EXIT Statement（退出声明）
- GOTO Statement

9. PSM SQL References

9.1 ALTER FUNCTION

功能

重新编译函数

语句

```
<alter function statement> ::=
    ALTER FUNCTION function_name COMPILE
    ;
```

使用范围及访问权限

执行<alter function statement>语句时用户需要有以下权限中的一个

- 对应函数的所有者
- 对函数所属的schema有(ALTER PROCEDURE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY PROCEDURE ON DATABASE

语句规则及参数

- Function name
  - 要编译的函数的名称
  - 与schema_name.fun_name相同可定义function所属的Schema省略schema_name时使用执行语句的用户的默认schema名

说明

再次编译指定的schema-level function

使用示例

```
gSQL> ALTER FUNCTION FUNC1 COMPILE;

Function altered.
```

兼容性

在标准SQL中是更改执行CREATE时指定特征的语句在SUNDB中是重新生成函数计划的语句

Feature ID	说明	是否支持
F381	Extended schema manipulation	O

Table 9-1 标准SQL兼容性

参考内容

详细内容参考如下

- CREATE FUNCTION
- DROP FUNCTION

9.2 ALTER PACKAGE

功能

重新编译package

语句

```
<alter package statement> ::=
    ALTER PACKAGE package_name <package compile clause>
    ;

<package compile clause> ::=
    COMPILE [PACKAGE|SPECIFICATION|BODY]
```

使用范围及访问权限

执行<alter package statement>时用户需要有以下权限中的一个

- 对应package的所有者
- 对package所属的schema有(ALTER PACKAGE 或 CONTROL SCHEMA) ON SCHEMA
- ALTER ANY PACKAGE ON DATABASE

语句规则及参数

- Package name
  - 要编译的package的名称
  - 如schema_name.package_name可定义package所属的schema省略schema_name时使用执行语句的用户的默认schema名
- Package compile clause
  - PACKAGE：重新编译specification和body(Default)
  - SPECIFICATION：仅重新编译specification
  - BODY：仅重新编译body

说明

重新编译指定的package

编译的package的执行code存储在plan cache

使用示例

```
ALTER PACKAGE PKG1 COMPILE;
Package altered.
```

```
ALTER PACKAGE PKG1 COMPILE PACKAGE;
Package altered.
```

```
ALTER PACKAGE PKG1 COMPILE BODY;
Package altered.
```

兼容性

在标准SQL中是ALTER MODULE语句

参考

详细内容参考如下

- CREATE PACKAGE
- CREATE PACKAGE BODY
- DROP PACKAGE

9.3 ALTER PROCEDURE

功能

重新编译过程（procedure）

语句

```
<alter procedure statement> ::=
    ALTER PROCEDURE proc_name COMPILE
    ;
```

使用范围及访问权限

执行<alter procedure statement>语句时用户需要有以下权限中的一个

- 对应procedure的所有者
- 对procedure所属的Schema有(ALTER PROCEDURE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY PROCEDURE ON DATABASE

语句规则及参数

- Proc name
  - 要编译的procedure的名称
  - 与schema_name.proc_name相同可定义procedure所属的Schema省略schema_name时使用执行语句的用户的默认schema名

说明

重新编译指定的schema-level procedure

使用示例

```
gSQL> CREATE OR REPLACE PROCEDURE PROC1( A1 INTEGER )
IS
BEGIN
  INSERT INTO T1 VALUES( A1 );
END;
/

ERR-01000(16409): Warning: Routine definition has compilation errors
ERR-HY000(17032): PSM compilation error :
(1) at (5:15): ERR-17053: schema or table object does not exist
Procedure created.

gSQL> CALL PROC1(1);
```

```
ERR-HY000(17032): PSM compilation error :
(1) at (5:15): ERR-17053: schema or table object does not exist

gSQL> CREATE TABLE T1( I1 INTEGER );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> ALTER PROCEDURE PROC1 COMPILE;

Procedure altered.

gSQL> COMMIT;

Commit complete.

gSQL> CALL PROC1(2);

Procedure Call complete.

gSQL> SELECT * FROM T1;

I1
--
 2

1 row selected.
```

兼容性

在标准SQL中是更改在CREATE时指定特征的语句在SUNDB中是重新生成procedure计划的语句

Feature ID	说明	支持状态
F381	Extended schema manipulation	O

Table 9-2 标准SQL兼容性

参考内容

详细内容参考如下

- CREATE PROCEDURE
- DROP PROCEDURE

9.4 CALL Statement

功能

执行schema level procedure或函数

语句

```
<call statement> ::=
    <sql call statement> | <odbc procedure call escape sequence>
    ;

<sql call statement> ::=
    CALL proc_name [ ( value_expr [ , value_expr ] .. ) ]  [ INTO { '?' |
{ host_param [ indicator_param ] } } ]

<odbc procedure call escape sequence> ::=
    '{' [ ? = ] CALL proc_name [ ( value_expr [ , value_expr ] .. ) ] '}'
```

使用范围及访问权限

执行<call statement>语句的用户需要有以下权限中的一个

- 对应procedure的EXECUTE权限

- 对procedure所属的schema有(EXECUTE PROCEDURE或CONTROL SCHEMA) ON SCHEMA
- EXECUTE ANY PROCEDURE ON DATABASE

语句规则及参数

- proc_name
  - 要执行的procedure/ function的名称
  - 与schema_name.Proc_name相同可包含procedure所属的schema
- value_expr
  - 表示传递给procedure的参数值也可使用'?'或':V1'等绑定参数

说明

使用指定的参数执行schema level SQL procedure或函数

在<sql call statement>形式中函数使用INTO子句之后的主机变量表达式或dynamic绑定参数（?）返回结果值

<odbc过程调用转义序列>格式用于在ODBC / JDBC中调用PROCEDURE这是一种标准语句并且SUNDB在服务器中支持此语句（它也可以在gsql等工具中使用）函数使用前面的赋值表达式（？=）返回结果值

使用示例

Call Procedure

```
gSQL> CREATE OR REPLACE PROCEDURE PROC1
(
  A1 INTEGER
)
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('A1=' || A1);
END;
/

Procedure created.

gSQL> \var v1 INTEGER;
gSQL> \exec :v1 := 123;
gSQL> CALL PROC1(:v1);
A1=123

Procedure Call complete.
```

Call Function

```
CREATE OR REPLACE FUNCTION FUNC1
(
  A1 INTEGER
)
RETURN INTEGER
```

```
IS
BEGIN
    return A1;
END;
/

Function created.


gSQL> \var v1 INTEGER;
gSQL> \var v2 INTEGER;
gSQL> \exec :v1 := 123;
gSQL> CALL FUNC1(:v1) INTO :v2;
Procedure Call complete.

gSQL> \print v2;
 V2
---
123
```

兼容性

由于标准SQL中只能调用PROCEDURE因此未定义[INTO]子句下层子句

9.5 CREATE FUNCTION

功能

定义schema level function

语句

```
<create function statement> ::=
    CREATE [ OR REPLACE ] FUNCTION <function name> [ ( <parameter list> ) ]
       <return clause>
       [ <function characteristics> ]
       { IS | AS }
       <item declaration>
       BEGIN
       <pl statement list>
       END [ <function name> ]
    ;

<parameter list> ::=
      <parameter name> [ <parameter mode> ] <datatype> [ <parameter default> ]
[ , ... ]

<parameter mode> ::=
      IN
    | OUT
    | IN OUT
```

```
<parameter default> ::=
      { := | DEFAULT } <value expression>

<return clause> ::=
      RETURN <datatype>
    | RETURN TABLE ( <table function column list> )

<table function column list> ::=
      <column name> <datatype> [ , ... ]

<function characteristics> ::=
      DETERMINISTIC
    | AUTHID CURRENT_USER
    | AUTHID DEFINER
```

使用范围及访问权限

执行<create function statement>语句时用户需要满足以下条件

- 生成函数需要有以下权限中的一个
  - 对函数所属的Schema有(CREATE PROCEDURE或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY PROCEDURE ON DATABASE
- 使用OR REPLACE子句时如果已有函数则需要有可删除原有函数的如下权限中的一个
  - 该函数的所有者
  - 对函数所属的Schema有(DROP PROCEDURE E或CONTROL SCHEMA) ON SCHEMA
  - DROP ANY PROCEDURE ON DATABASE
- 执行语句的用户为生成函数的所有者

语句规则及参数

OR REPLACE

如果函数已存在则用新函数替换现有函数

function name

是要创建的函数的名称应在schema中是唯一的名称

与schema_name.function_name相同可定义函数所属的schema省略schema_name时使用执行语句的用户的默认schema名

函数名称的长度应小于128个字节

parameter name

定义函数的参数名称

各参数的名称在函数中应是唯一的名称

即函数的参数和PL item的名称不能相同

参数名称的长度应小于128个字节

单个函数中可用参数的最大数量没有限制

parameter mode

设置各参数模式

参数模式为IN, OUT, IN OUT

未指定参数模式时默认模式为IN

parameter default

参数的默认值

执行函数时可省略指定了参数默认值的参数

未指定并省略参数时默认值为定义参数时指定的<value expression>

<value expression>的数据类型应为参数的数据类型

在有<parameter default>的参数以后定义的所有参数必须有<parameter default>

return clause

定义函数的返回类型

<return clause>中如下定义

- RETURN <datatype>
  - 定义函数返回的返回值的数据类型
- RETURN TABLE (<table function column list>)
  - 定义返回结果集的表类型

table function column list

Table function返回的结果集的column名称

Column名称的长度应小于128个字节

无Column数量限制

各column名称在<table function column list>中是唯一的

Column名称可以与参数和declare item的名称相同

<table function column list>中定义的column不可在函数的PL块中引用

function characteristics

定义函数执行选项 对同一项目仅可描述一次

- DETERMINISTIC：输入相同的参数值时该函数始终返回相同的结果值
- AUTHID CURRENT_USER：函数执行期间执行的SQL根据invoker的授权进行分析及执行
- AUTHID DEFINER：函数执行期间执行的SQL根据定义者的授权进行分析及执行

Item Declaration

声明要在函数内使用的本地变量等item

可声明PL块中可声明的所有item

PL Stmt List

函数的主体部分列出要执行的PL语句

无法在函数的内部使用'？'或'：V1'等绑定参数

说明

定义schema-level SQL function可以在所有表达式中调用创建的函数

可在INFORMATION_SCHEMA的ROUTINES表中查看函数的定义函数的参数的定义可在INFORMATION_SCHEMA的PARAMETERS表中查看

由于缺少相关对象导致函数暂时不稳定时可使用ALTER FUNCTION 语句尝试重新创建计划

已创建的函数可用DROP FUNCTION删除

因无函数的最大创建数量限制所以在存储空间充足的前提下可一直创建

使用示例

```
gSQL> CREATE OR REPLACE FUNCTION FUNC1( A1 INTEGER, A2 INTEGER )
  RETURN INTEGER
  IS
    V1 INTEGER;
  BEGIN

    SELECT COUNT(*)
      INTO V1
      FROM T1
      WHERE T1.I1 >= A1 AND T1.I1 <= A2;

    RETURN V1;
  END;
  /

Function created.
```

兼容性

标准SQL未定义OR REPLACE子句

Feature ID	说明	支持状态
T471	Result sets return value	X

Feature ID	说明	支持状态
T341	Overloading of SQL-invoked functions and SQL-invoked procedures	X
S023	Basic structured types	X
S241	Transform functions	X
S024	Enhanced structured types	X
T571	Array-returning external SQL-invoked functions	X
T572	Multiset-returning external SQL-invoked functions	X
S201	SQL routines on arrays	X
S202	SQL-invoked routines on multisets	X
T323	Explicit security for external routines	X
S231	Structured type locators	X
S232	Array locators	X
S233	Multiset locators	X
T041	Basic LOB data type support	X
S027	Create method by specific method name	X
T041	Basic LOB data type support	X
T324	Explicit security for SQL routines	O
T326	Table functions	O
T651	SQL-schema statements in SQL routines	X
T652	SQL-dynamic statements in SQL routines	O
T653	SQL-schema statements in external routines	X
T654	SQL-dynamic statements in external routines	X
T655	Cyclically dependent routines	X
T272	Enhanced savepoint management	X
T522	Default values for IN parameters of SQL-invoked procedures	O
B121	Routine language Ada	X
B122	Routine language C	X
B123	Routine language COBOL	X
B124	Routine language Fortran	X
B125	Routine language MUMPS	X
B126	Routine language Pascal	X
B127	Routine language PL/I	X
B128	Routine language SQL	O
B129	Routine language Ada: VARCHAR and NUMERIC support	X

Table 9-3 标准SQL兼容性

参考内容

详细内容参考如下

- DROP FUNCTION
- ALTER FUNCTION

9.6 CREATE PACKAGE

功能

定义在package内使用的public item的spec

语句

```
<create package statement> ::=
    CREATE [ OR REPLACE ]
        PACKAGE package_name
        [ <package_characteristics> ]
        { IS | AS }
       <item_declaration>
    END
    ;

<package_characteristics> ::=
    AUTHID CURRENT_USER | AUTHID DEFINER

<item_declaration> ::=    <variable declaration>
                        | <cursor declaration>
                        | <user-defined type declaration>
                        | <function declaration>
                        | <procedure declaration>
                        | <user exception declaration>
                        | <cursor definition>
```

使用范围及访问权限

执行<create package statement>语句时用户需要满足以下条件

- 生成package需要有以下权限中的一个
  - 对package所属的schema有(CREATE PACKAGE或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY PACKAGE ON DATABASE
- 使用OR REPLACE子句时如果package已存在则需要有可删除原有package的如下权限中的一个
  - 对应package的所有者
  - 对package所属的schema有(DROP PACKAGE或CONTROL SCHEMA) ON SCHEMA
- 执行语句的用户时生成的package的所有者

语句规则及参数

OR REPLACE

已有package时代替原有的package specification

PACKAGE NAME

要生成的package的名称应在schema中是唯一的

如schema_name.package_name可定义package所属的schema省略schema_name时使用执行语句的用户的默认schema名

Package名的长度应小于128字节

Package Characteristics

定义执行package的选项对同样的项目只能描述一次

- AUTHID CURRENT_USER: 在执行package内的item的过程中执行的SQL按照invoker的authorization解析并执行
- AUTHID DEFINER：在执行package内的item的过程中执行的SQL按照definer的authorization解析并执行

Item Declaration

声明可在package外部访问的public变量type游标游标变量function specprocedure spec

Package内的function和procedure必须通过create package body语句定义

在package内没有SQL的仅声明的游标必须通过create package body语句定义

另外package内的functionprocedure游标的argument和返回类型应与body语句内定义的一致

说明

生成Schema-level package specification

生成的package的所有public item可被其他procedure/ function/ package/ anonymous block引用

Package的生成信息可在DEFINITION_SCHEMA或INFORMATION_SCHEMA的MODULES表查看

Package内的公开的各个procedure/function目录可在DEFINITION_SCHEMA或INFORMATION_SCHEMA的ROUTINES表查看

Package内的公开的各个procedure/ function的parameter的定义可在DEFINITION_SCHEMA或INFORMATION_SCHEMA的PARAMETERS表查看

未公开的procedure/function可在MODULE_BODY表查看

Package内的procedure/function变量引用的object的信息出现暂时不完整时可通过<alter package>语句尝试重新编译

使用示例

```
CREATE OR REPLACE PACKAGE PKG1
IS
  V1 INTEGER;
  PROCEDURE PROC1;
  FUNCTION FUNC1 RETURN INTEGER;
END;
/

Package created.
```

兼容性

在标准SQL中是CREATE MODULE语句

参考内容

详细内容参考如下

- CREATE PACKAGE BODY
- DROP PACKAGE
- ALTER PACKAGE

9.7 CREATE PACKAGE BODY

功能

生成要在package内使用的procedure/ function/游标的定义

语句

```
<create package body statement> ::=
    CREATE [ OR REPLACE ] PACKAGE BODY package_name
    { IS | AS }
       <item_declaration>
       <cursor_definition>
       <routine_definition>
    [ BEGIN <initialization part> ]
    END
    ;

<item_declaration> ::=    VARIABLE_DECLARATION
                        | CURSOR_DECLARATION
                        | USER_DEFINED_TYPE_DEFINITION
                        | FUNCTION_DECLARATION
                        | PROCEDURE_DECLARATION
                        | USER_EXCEPTION_DECLARATION

<routine_definition> ::= FUNCTION_DEFINITION
                       | PROCEDURE_DEFINITION
```

```
<initialization part> ::= <pl_stmt_list>
```

使用范围及访问权限

执行<create package body statement> 语句时用户需要满足以下条件

- 生成package body需要有以下权限中的一个
  - 对package所属的schema有(CREATE PACKAGE或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY PACKAGE ON DATABASE
- 应为已定义Package spec的状态
- 使用OR REPLACE子句时如果package body已存在则需要有可删除原有package的如下权限中的一个
  - 对应package的所有者
  - 对package所属的schema有(DROP PACKAGE或CONTROL SCHEMA) ON SCHEMA
  - DROP ANY PACKAGE ON DATABASE
- 执行语句的用户为生成的package的所有者

语句规则及参数

OR REPLACE

已有package body时代替原有的 package body definition

PACKAGE NAME

要生成的package body的名称应使用与生成package spec相同的名称应在schema中是唯一的

如schema_name.package_name可定义package所属的schema省略schema_name时使用执行语句的用户的默认schema名

Package名的长度应小于128字节

Item Declaration

声明在package内使用的PSM identifier（变量type游标function specprocedure specexception等）

在package spec内声明的routine必须在package body定义

在package spec内没有SQL的仅声明的游标必须在package body定义

另外package内的functionprocedure游标的argument和返回类型应与body语句内定义的一致

Initialization Part

描述生成package instance的过程中为了内部变量等的初始化而仅执行一次的语句

说明

生成Schema-level package body

生成的package body的所有public item无法被其他procedure/ function/ package/ anonymous block引用仅在对应package body内使用

Package body的生成信息可在DEFINITION_SCHEMA或INFORMATION_SCHEMA的MODULES_BODY表查看

Package body内的各个private item目录不可在DEFINITION_SCHEMA或INFORMATION_SCHEMA查看

Package内的procedure/function/变量引用的object的信息出现暂时不完整时可通过<alter package body>语句尝试重新编译

生成的package body可通过<drop package>或 <drop package body>语句删除

使用示例

```
CREATE OR REPLACE PACKAGE PKG1
IS
  V1 INTEGER;
  PROCEDURE PROC1;
  FUNCTION FUNC1 RETURN INTEGER;
END;
/

Package created.

CREATE OR REPLACE PACKAGE BODY PKG1
IS
  FUNCTION FUNC1 RETURN INTEGER
  IS
  BEGIN
      RETURN V1;
  END;

  PROCEDURE PROC1
  IS
  BEGIN
      IF V1 IS NULL
      THEN
          V1 := 10;
      ELSE
          V1 := V1 + 10;
      END IF;
  END;
```

```
END;
/

Package created.
```

兼容性

标准SQL未定义

参考内容

详细内容可参考如下

- CREATE PACKAGE
- DROP PACKAGE
- ALTER PACKAGE

9.8 CREATE PROCEDURE

功能

定义schema-level procedure

语句

```
<create procedure statement> ::=
      CREATE [ OR REPLACE ] PROCEDURE <procedure name> [ ( <parameter list> ) ]
      { IS | AS }
      <item declaration>
      BEGIN
      <pl statement list>
      END [ <procedure name> ]
      ;

<parameter list> ::=
      <parameter name> [ <parameter mode> ] <datatype> [ <parameter default> ]
[ , ... ]

<parameter mode> ::=
      IN
    | OUT
    | IN OUT

<parameter default> ::=
```

```
      { := | DEFAULT } <value expression>

<procedure characteristics> ::=
      AUTHID CURRENT_USER
    | AUTHID DEFINER
```

使用范围及访问权限

执行<create procedure statement>语句时用户需要满足以下条件

- 生成procedure需要有以下权限中的一个
  - 对procedure所属的Schema有(CREATE PROCEDURE或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY PROCEDURE ON DATABASE
- 使用OR REPLACE子句时如果已有procedure则需要有可删除原有procedure的如下权限中的一个
  - 该procedure的所有者
  - 对procedure所属的Schema有(DROP PROCEDURE或CONTROL SCHEMA) ON SCHEMA
  - DROP ANY PROCEDURE ON DATABASE
- 执行语句的用户为生成的procedure的所有者

语句规则及参数

OR REPLACE

已经存在procedure时用新的procedure代替原来的procedure

procedure name

作为要创建的procedure的名称应在schema中拥有唯一的名称

与schema_name.procedure_name相同可定义procedure所属的schema

省略schema_name时使用执行语句的用户的默认schema名称

procedure名称的长度应小于128个字节

parameter name

定义procedure的参数名称

各参数的名称在procedure中应是唯一的名称

即procedure的参数和PL item不能有相同的名称

参数名称的长度应小于128个字节

单个procedure中可用参数的最大数量没有限制

parameter mode

设置各参数模式

参数模式为IN, OUT, IN OUT

未指定参数模式时默认模式为IN

parameter default

参数的默认值

执行procedure时可省略指定了参数默认值的参数

未指定并省略参数时默认值为定义参数时指定的<value expression>

<value expression>的数据类型应为参数的数据类型

在有<parameter default>的参数以后定义的所有参数须有<parameter default>

procedure characteristics

定义procedure执行选项 对同一项目只能描述一次

- AUTHID CURRENT_USER：procedure执行期间执行的SQL根据invoker的授权进行解释及执行
- AUTHID DEFINER：procedure执行期间执行的SQL根据定义者的授权进行解释及执行

Item Declaration

声明要在procedure内部使用的本地变量等的item

可声明PL块中可声明的所有item

PL Stmt List

Procedure的主体部分列出要执行的PL语句

无法在procedure内使用'？'或'：V1'等绑定参数

说明

定义schema-level SQL procedure可在CALL语句匿名块或其他procedure/ 函数中调用创建的procedure

可在INFORMATION_SCHEMA的ROUTINES表中查看procedure的定义

Procedure的参数定义可在INFORMATION_SCHEMA的PARAMETERS表中查看

由于缺少相关对象导致procedure暂时不稳定时可使用ALTER PROCEDURE语句尝试重新创建计划

可通过DROP PROCEDURE删除已创建的procedure

因为没有创建procedure数量的限制所以在存储空间充足的前提下可持续创建

使用示例

```
gSQL> CREATE OR REPLACE PROCEDURE PROC1( A1 INTEGER, A2 INTEGER )

  IS
    V1 INTEGER;
  BEGIN

    SELECT COUNT(*)
      INTO V1
      FROM T1
      WHERE T1.I1 >= A1 AND T1.I1 <= A2;

    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
  END;
  /

Procedure created.
```

兼容性

标准SQL没有定义以下子句

Feature ID	说明	支持状态
T471	Result sets return value	X
T341	Overloading of SQL-invoked functions and SQL-invoked procedures	X
S023	Basic structured types	X
S241	Transform functions	X
S024	Enhanced structured types	X
T571	Array-returning external SQL-invoked functions	X
T572	Multiset-returning external SQL-invoked functions	X
S201	SQL routines on arrays	X
S202	SQL-invoked routines on multisets	X
T323	Explicit security for external routines	X
S231	Structured type locators	X
S232	Array locators	X
S233	Multiset locators	X
T041	Basic LOB data type support	X
S027	Create method by specific method name	X
T041	Basic LOB data type support	X
T324	Explicit security for SQL routines	O
T326	Table functions	O
T651	SQL-schema statements in SQL routines	X
T652	SQL-dynamic statements in SQL routines	O
T653	SQL-schema statements in external routines	X
T654	SQL-dynamic statements in external routines	X
T655	Cyclically dependent routines	X
T272	Enhanced savepoint management	X
T522	Default values for IN parameters of SQL-invoked procedures	O
B121	Routine language Ada	X
B122	Routine language C	X
B123	Routine language COBOL	X
B124	Routine language Fortran	X
B125	Routine language MUMPS	X
B126	Routine language Pascal	X
B127	Routine language PL/I	X
B128	Routine language SQL	O

Feature ID	说明	支持状态
B129	Routine language Ada: VARCHAR and NUMERIC support	X

Table 9-4 标准SQL兼容性

参考内容

详细内容参考如下

- DROP PROCEDURE
- ALTER PROCEDURE

9.9 DROP FUNCTION

功能

删除函数

语句

```
<drop function statement> ::=
    DROP FUNCTION [ IF EXISTS ] func_name
    ;
```

使用范围及访问权限

执行<drop function statement>语句时用户需要有以下权限中的一个

- 对应函数的所有者
- 对函数所属的Schema有(DROP PROCEDURE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY PROCEDURE ON DATABASE

语句规则及参数

IF EXISTS

函数不存在时也不报错

FUNC NAME

要删除的函数名称

与schema_name.func_name相同可定义函数所属的schema省略schema_name时使用执行语句的用户的默认schema名

说明

删除指定的schema-level function

使用示例

```
gSQL> CREATE OR REPLACE FUNCTION FUNC1
RETURN INTEGER
 IS
    V1 INTEGER;
  BEGIN
    V1 := 10;
    RETURN V1;
  END;
  /
```

```
Function created.


COMMIT;

Commit complete.
gSQL> DROP FUNCTION FUNC1;

Function dropped.
```

兼容性

标准SQL没有定义以下子句

Feature ID	说明	支持状态
F032	CASCADE drop behavior	X
S024	Enhanced structured types	X

Table 9-5 标准SQL兼容性

参考内容

详细内容参考如下

- CREATE FUNCTION
- ALTER FUNCTION

9.10 DROP PACKAGE

功能

删除package（仅删除body或删除spec/body）

语句

```
<drop package statement> ::=
    DROP PACKAGE [BODY] [ IF EXISTS ] package_name
    ;
```

使用范围及访问权限

执行<drop package statement>语句时 用户需要有以下权限中的一个

- 对应package的所有者
- 对package所属的schema有(DROP PACKAGE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY PACKAGE ON DATABASE

语句规则及参数

BODY

仅删除拥有指定名称的package的body对象未指定*BODY* 这个关键字时删除package specification和body全部

IF EXISTS

即使没有package也不报错

PACKAGE NAME

要删除的package的名称

如schema_name.package_name可定义package所属的schema

省略schema_name时使用执行语句的用户的默认shcema名

说明

删除指定的package object

使用示例

```
CREATE PACKAGE PKG1
```

```
  IS
    V1 INTEGER;
    FUNCTION FUNC1 (A1 INTEGER) RETURN INTEGER;
  END;
  /

Package created.

DROP PACKAGE IF EXISTS PKG1;

Package dropped.
```

兼容性

标准SQL中是DRO MODULE语句

参考内容

详细内容参考如下

- CREATE PACKAGE
- CREATE PACKAGE BODY
- ALTER PACKAGE

9.11 DROP PROCEDURE

功能

删除procedure

语句

```
<drop procedure statement> ::=
    DROP PROCEDURE [ IF EXISTS ] proc_name
    ;
```

使用范围及访问权限

执行<drop procedure statement>语句时用户需要有以下权限中的一个

- 对应procedure的所有者
- 对procedure所属的Schema有(DROP PROCEDURE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY PROCEDURE ON DATABASE

语句规则及参数

IF EXISTS

Procedure不存时也不报错

PROC NAME

要删除的procedure名称

与schema_name.proc_name相同可定义procedure所属的schema省略schema_name时使用执行语句的用户的默认schema名

说明

删除指定的schema-level procedure

使用示例

```
gSQL> CREATE OR REPLACE PROCEDURE PROC1( A1 INTEGER, A2 INTEGER )
  IS
    V1 INTEGER;
  BEGIN

    SELECT COUNT(*)
      INTO V1
```

```
      FROM T1
      WHERE T1.I1 >= A1 AND T1.I1 <= A2;

    DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
  END;
  /

Procedure created.


COMMIT;

Commit complete.


gSQL> DROP PROCEDURE PROC1;

Procedure dropped.
```

兼容性

标准SQL没有定义以下子句

Feature ID	说明	支持状态
F032	CASCADE drop behavior	X
S024	Enhanced structured types	X

Table 9-6 标准SQL 兼容性

参考内容

详细内容参考如下

- CREATE PROCEDURE
- ALTER PROCEDURE