

达梦技术手册

DM8_SQL 程序设计

Service manual of DM8_Sql_Program



前言

概述

DMSQL 程序是 DM 提供的一种过程化 SQL 语言。本文档介绍了 DMSQL 程序支持的各种语法、功能及其使用方法，并提供了大量示例。读者在阅读完本文档后可以自行设计较为复杂的 DMSQL 程序以实现复杂应用逻辑。

读者对象





本文档主要适用于 DM 数据库的：

- 开发工程师
- 测试工程师
- 技术支持工程师
- 数据库管理员

通用约定

在本文档中可能出现下列标志，它们所代表的含义如下：

表 0.1 标志含义

标志	说明
 警告：	表示可能导致系统损坏、数据丢失或不可预知的结果。
 注意：	表示可能导致性能降低、服务不可用。
 小窍门：	可以帮助您解决某个问题或节省您的时间。
 说明：	表示正文的附加信息，是对正文的强调和补充。

在本文档中可能出现下列格式，它们所代表的含义如下：

表 0.2 格式含义

格式	说明
宋体	表示正文。
Courier new	表示代码或者屏幕显示内容。
粗体	表示命令行中的关键字（命令中保持不变、必须照输的部分）或者正文中强调的内容。标题、警告、注意、小窍门、说明等内容均采用粗体。
<>	语法符号中，表示一个语法对象。
::=	语法符号中，表示定义符，用来定义一个语法对象。定义符左边为语法对象，右边为相应的语法描述。
	语法符号中，表示或者符，限定的语法选项在实际语句中只能出现一个。
{ }	语法符号中，大括号内的语法选项在实际的语句中可以出现 0...N 次 (N 为大于 0 的自然数)，但是大括号本身不能出现在语句中。
[]	语法符号中，中括号内的语法选项在实际的语句中可以出现 0...1 次，但是中括号本身不能出现在语句中。
关键字	关键字在 DM_SQL 语言中具有特殊意义，在 SQL 语法描述中，关键字以大写形式出现。但在实际书写 SQL 语句时，关键字既可以大写也可以小写。

访问相关文档

如果您安装了 DM 数据库，可在安装目录的“\doc”子目录中找到 DM 数据库的各种手册与技术丛书。

您也可以通过访问我们的网站 www.dameng.com 阅读或下载 DM 的各种相关文档。

联系我们

如果您有任何疑问或是想了解达梦数据库的最新动态消息，请联系我们：

网址：www.dameng.com

技术服务电话：400-648-9899

技术服务邮箱：tech@dameng.com

目录

1 概述	1
1.1 DMSQL 程序简介	1
1.2 使用 DMSQL 程序的优点	1
1.3 一个简单的 DMSQL 程序示例	2
2 DMSQL 程序数据类型与操作符	4
2.1 常规数据类型	4
2.1.1 数值数据类型	4
2.1.2 字符数据类型	7
2.1.3 多媒体数据类型	8
2.1.4 日期时间数据类型	9
2.1.5 BOOL/BOOLEAN 数据类型	12
2.2 %TYPE 和%ROWTYPE	12
2.3 记录类型	14
2.4 数组类型	16
2.4.1 静态数组类型	16
2.4.2 动态数组类型	17
2.4.3 复杂类型数组	19
2.5 集合类型	21
2.5.1 VARRAY	21
2.5.2 索引表	22
2.5.3 嵌套表	25
2.5.4 集合类型支持的方法	26
2.6 类类型	30
2.7 子类型	30
2.8 操作符	30
3 DMSQL 程序的定义、调用与删除	32
3.1 存储过程	32

3.2 存储函数	34
3.3 客户端 DMSQL 程序	37
3.4 参数	38
3.5 变量	40
3.6 使用 OR REPLACE 选项	43
3.7 调用权限子句	43
3.8 调用、重新编译与删除存储模块	43
3.8.1 调用存储模块	43
3.8.2 重新编译存储模块	45
3.8.3 删除存储模块	45
4 DMSQL 程序中的各种控制结构	47
4.1 语句块	47
4.2 分支结构	49
4.2.1 IF 语句	49
4.2.2 CASE 语句	53
4.2.3 SWITCH 语句	55
4.3 循环控制结构	56
4.3.1 LOOP 语句	56
4.3.2 WHILE 语句	57
4.3.3 FOR 语句	58
4.3.4 REPEAT 语句	59
4.3.5 FORALL 语句	60
4.3.6 EXIT 语句	61
4.3.7 CONTINUE 语句	64
4.4 顺序结构	66
4.4.1 GOTO 语句	66
4.4.2 NULL 语句	67
4.5 其他语句	68
4.5.1 赋值语句	68
4.5.2 调用语句	69

4.5.3 RETURN 语句	71
4.5.4 PRINT 语句	71
4.5.5 PIPE ROW 语句	72
5 DMSQL 程序中的 SQL 语句	74
5.1 普通静态 SQL 语句	74
5.1.1 数据操纵	74
5.1.2 数据查询	75
5.1.3 事务控制	76
5.2 游标	77
5.2.1 静态游标	77
5.2.2 动态游标	83
5.2.3 游标变量（引用游标）	85
5.2.4 使用游标更新、删除数据	86
5.2.5 使用游标 FOR 循环	87
5.3 动态 SQL	89
5.4 返回查询结果集	92
5.5 自治事务	93
5.5.1 定义自治事务	93
5.5.2 自治事务完整性与死锁检测	94
5.5.3 自治事务嵌套	95
6 DMSQL 程序异常处理	97
6.1 异常处理的优点	97
6.2 预定义异常	97
6.3 用户自定义异常	98
6.4 异常的抛出	101
6.5 内置函数 SQLCODE 和 SQLERRM	102
6.6 异常处理部分	103
7 基于 C、JAVA 语法的 DMSQL 程序	107
7.1 C 语法 DMSQL 程序	107
7.2 JAVA 语法 DMSQL 程序	109

8 DMSQL 程序调试	112
8.1 使用命令行工具 DMDBG 调试 DMSQL 程序	112
8.1.1 dmdbg 工具命令简介	112
8.1.2 使用 dmdbg 工具	113
8.2 使用图形化客户端工具 MANAGER 调试 DMSQL 程序	125

1 概述

1.1 DMSQL 程序简介

DMSQL 程序是达梦数据库对标准 SQL 语言的扩展，是一种过程化 SQL 语言。在 DMSQL 程序中，包括一整套数据类型、条件结构、循环结构和异常处理结构等，DMSQL 程序中执行 SQL 语句，SQL 语句中也可以使用 DMSQL 函数。

DMSQL 程序是一种技术，而不是一种独立的工具，它是和 DM 数据库服务器紧密结合在一起的。可以认为这种技术是执行 DMSQL 程序的一种机器，它可以接受任何有效的 DMSQL 程序，按照语言本身所规定的语义执行，并将结果返回给客户。

DMSQL 程序可以分为存储模块和客户端 DMSQL 程序两类。

用户可以使用 DMSQL 程序语言创建过程或函数，称为存储过程和存储函数。这些过程或函数像普通的过程或函数一样，有输入、输出参数和返回值，它们与表和视图等数据库对象一样被存储在数据库中，供用户随时调用。存储过程和存储函数在功能上相当于客户端的一段 SQL 批处理程序，但是在许多方面有着后者无法比拟的优点，它为用户提供了一种高效率的编程手段，成为现代数据库系统的重要特征。通常，我们将存储过程和存储函数统称为存储模块。

客户端 DMSQL 程序可以实现的功能与存储模块一致，不同的是客户端 DMSQL 程序并不创建一个具体的数据库对象。其处理方法为 DM 数据库服务器在预编译阶段将客户端 DMSQL 程序转化为虚过程。虚过程不需要存储，创建后立即执行，当执行的语句释放时，虚过程对象也一同被释放。客户端 DMSQL 程序只从语法上和存储模块兼容，完成和存储模块一样的功能，是一种编程手段。

1.2 使用 DMSQL 程序的优点

DMSQL 程序具有以下优点：

- ✓ 与 SQL 语言的完美结合

SQL 语言已成为数据库的标准语言，DMSQL 程序支持所有 SQL 数据类型和所有 SQL 函数，同时支持所有 DM 对象类型。在 DMSQL 程序中可以使用 SELECT、INSERT、DELETE、UPDATE 数据操作语句，事务控制语句，游标操纵语句以及通过动态 SQL 执行

DDL 语句。与 SQL 语言的完美结合使得 DMSQL 程序不仅能实现 SQL 的所有功能，且由于其自身的程序设计特性，能提供更加丰富、强大的功能。

✓ 提供更高的生产率

在使用 DMSQL 程序设计应用时，围绕存储过程/函数进行设计，可以避免重复编码，提高生产率；在自顶向下设计应用时，不必关心实现的细节；编程方便。从 DM7 开始，支持 C 和 JAVA 语言语法的 DMSQL 程序，这样在对自定义的 DMSQL 程序语法不熟悉的情况下也可以对数据库进行各种操作，对数据库的操作更加灵活，也更加容易。

✓ 提供更好的性能

DMSQL 存储模块在创建时被编译成伪码序列，在运行时不需要重新进行编译和优化处理，具有更快的执行速度，可以同时被多个用户调用，并能够减少操作错误。使用存储模块可减少应用对 DM 的调用，降低系统资源浪费，显著提高性能，尤其是对在网络上与 DM 通讯的应用更加显著。

✓ 便于维护

用户定义的存储模块在 DM 数据库中集中存放，用户可以根据需要随时查询、删除或重建它们，而调用这些存储模块的应用程序可以不作任何修改，或只做少量调整。存储模块能被其他的 DMSQL 程序或 SQL 命令调用，任何客户/服务器工具都能访问存储模块，具有很好的可重用性。

✓ 提供更高的安全性

存储模块可将用户与具体的内部数据操作进行隔离，提高数据库的安全性。如一个存储模块查询并修改一个表的某几个列，管理员将这个存储模块的执行权限授予某用户，而不必将表的访问和修改权限授予这个用户，保证用户只访问修改其需要的数据。

可以使用 DM 的管理工具管理存储在数据库中的存储模块的安全性，可以授予或撤销数据库其他用户执行存储模块的权限。

1.3 一个简单的 DMSQL 程序示例

下面通过一个简单的例子让读者对 DMSQL 程序有一个直观的了解。

```
CREATE OR REPLACE PROCEDURE RESOURCES.person_account AS
DECLARE
    person_count INT;
```

```
BEGIN

    SELECT COUNT(*) INTO person_count FROM RESOURCES.EMPLOYEE;

    DBMS_OUTPUT.PUT_LINE('公司总人数 ' || person_count);

    IF person_count < 5 THEN

        RAISE_APPLICATION_ERROR(-20001, '公司总人数过少!');

    ELSE

        NULL;

    END IF;

END;

/
```

该例子创建一个名为 RESOURCES.person_account 的存储过程，其中定义了变量，在执行部分执行一条 SELECT 语句，打印查询结果，并对查询结果进行判断，如果查询出的值<5，则抛出一个异常，否则不做任何处理。

这个例子用到了 DMSQL 程序的定义存储过程、变量定义、执行 DML 语句、控制语句、抛出异常等功能，而 DMSQL 程序的功能远不止这些，后续章节将对 DMSQL 程序的功能一一进行介绍。

2 DMSQL 程序数据类型与操作符

DMSQL 程序支持所有的 DM SQL 数据类型，包括：精确数值数据类型、近似数值数据类型、字符数据类型、多媒体数据类型、一般日期时间数据类型、时间间隔数据类型。

此外，为了进一步提高 DMSQL 程序的程序设计属性，DMSQL 程序还扩展支持了 %TYPE、%ROWTYPE、记录类型、数组类型、集合类型和类类型，用户还可以定义自己的子类型。

2.1 常规数据类型

2.1.1 数值数据类型

1. NUMERIC 类型

语法：

```
NUMERIC[(精度 [, 标度])]
```

功能：

NUMERIC 数据类型用于存储零、正负定点数。其中：精度是一个无符号整数，定义了总的数字数，精度范围是 1 至 38，标度定义了小数点右边的数字位数，定义时如省略精度，则默认是 16。如省略标度，则默认是 0。一个数的标度不应大于其精度。例如：NUMERIC(4,1) 定义了小数点前面 3 位和小数点后面 1 位，共 4 位的数字，范围在 -999.9 到 999.9。所有 NUMERIC 数据类型，如果其值超过精度，达梦数据库返回一个出错信息，如果超过标度，则多余的位截断。

如果不指定精度和标度，缺省精度为 38。

2. NUMBER 类型

语法：

```
NUMBER[(精度 [, 标度])]
```

功能：

与 NUMERIC 类型相同。

3. DECIMAL/DEC 类型

语法:

DECIMAL[(精度 [, 标度])]

DEC[(精度 [, 标度])]

功能:

与 NUMERIC 相似。

4. BIT 类型

语法:

BIT

功能:

BIT 类型用于存储整数数据 1、0 或 NULL，可以用来支持 ODBC 和 JDBC 的布尔数据类型。DM 的 BIT 类型与 SQL SERVER2000 的 BIT 数据类型相似。

5. INTEGER/INT 类型

语法:

INTEGER

INT

功能:

用于存储有符号整数，精度为 10，标度为 0。取值范围为： $-2147483648 (-2^{31}) \sim +2147483647 (2^{31}-1)$ 。

6. PLS_INTEGER 类型

语法:

PLS_INTEGER

功能:

与 INTEGER 相同。

7. BIGINT 类型

语法:

BIGINT

功能:

用于存储有符号整数，精度为 19，标度为 0。取值范围为： $-9223372036854775808 (-2^{63}) \sim 9223372036854775807 (2^{63}-1)$ 。

8. TINYINT 类型

语法:

TINYINT

功能:

用于存储有符号整数，精度为 3，标度为 0。取值范围为：-128~+127。

9. BYTE 类型

语法:

BYTE

功能:

与 TINYINT 相似，精度为 3，标度为 0。

10. SMALLINT 类型

语法:

SMALLINT

功能:

用于存储有符号整数，精度为 5，标度为 0。

11. BINARY 类型

语法:

BINARY[(长度)]

功能:

BINARY 数据类型指定定长二进制数据。缺省长度为 1 个字节，最大长度由数据库页面大小决定，具体可参考《DM8_SQL 语言使用手册》1.4.1 节。BINARY 常量以 0x 开始，后跟数据的十六进制表示，例如 0x2A3B4058。

12. VARBINARY 类型

语法:

VARBINARY[(长度)]

功能:

VARBINARY 数据类型指定变长二进制数据，用法类似 BINARY 数据类型，可以指定一个正整数作为数据长度。缺省长度为 8188 个字节，最大长度由数据库页面大小决定，具体可参考《DM8_SQL 语言使用手册》1.4.1 节。

13. REAL 类型

语法:

REAL

功能:

REAL 是带二进制的浮点数，但它不能由用户指定使用的精度，系统指定其二进制精度为 24，十进制精度为 7。取值范围 $-3.4E + 38 \sim 3.4E + 38$ 。

14. FLOAT 类型

语法:

FLOAT[(精度)]

功能:

FLOAT 是带二进制精度的浮点数，精度最大不超过 53，如省略精度，则二进制精度为 53，十进制精度为 15。取值范围为 $-1.7E + 308 \sim 1.7E + 308$ 。

15. DOUBLE 类型

语法:

DOUBLE[(精度)]

功能:

同 FLOAT 相似，精度最大不超过 53。

16. DOUBLE PRECISION 类型

语法:

DOUBLE PRECISION

功能:

该类型指明双精度浮点数，其二进制精度为 53，十进制精度为 15。取值范围 $-1.7E + 308 \sim 1.7E + 308$ 。

2.1.2 字符数据类型

1. CHAR/CHARACTER 类型

语法:

CHAR[(长度)]

CHARACTER[(长度)]

功能：

定长字符串，最大长度由数据库页面大小决定，具体可参考《DM8_SQL 语言使用手册》1.4.1 节。长度不足时，自动填充空格。

2. VARCHAR 类型

语法：

VARCHAR[(长度)]

功能：

可变长字符串，最大长度由数据库页面大小决定，具体可参考《DM8_SQL 语言使用手册》1.4.1 节。

2.1.3 多媒体数据类型

1. TEXT/LONGVARCHAR 类型

语法：

TEXT

LONGVARCHAR

功能：

变长字符串类型，其字符串的长度最大为 2G-1，可用于存储长的文本串。

2. IMAGE/LONGVARBINARY 类型

语法：

IMAGE

LONGVARBINARY

功能：

可用于存储多媒体信息中的图像类型。图像由不定长的像素点阵组成，长度最大为 2G-1 字节。该类型除了存储图像数据之外，还可用于存储任何其它二进制数据。

3. BLOB 类型

语法：

BLOB

功能:

BLOB 类型用于指明变长的二进制大对象, 长度最大为 2G-1 字节。

4. CLOB 类型

语法:

CLOB

功能:

CLOB 类型用于指明变长的字符串, 长度最大为 2G-1 字节。

5. BFILE 类型

语法:

BFILE

功能:

BFILE 用于指明存储在操作系统中的二进制文件, 文件存储在操作系统而非数据库中, 仅能进行只读访问。

2.1.4 日期时间数据类型

DM8SQL 程序支持的日期时间数据类型分为一般日期时间数据类型、时区数据类型和时间间隔数据类型三类。

2.1.4.1 一般日期时间数据类型

1. DATE 类型

语法:

DATE

功能:

DATE 类型包括年、月、日信息, 定义了 '-4712-01-01' 和 '9999-12-31' 之间任何一个有效的格里高利日期。

2. TIME 类型

语法:

TIME [(小数秒精度)]

功能:

TIME 类型包括时、分、秒信息,定义了一个在'00:00:00.000000'和'23:59:59.999999'之间的有效时间。TIME 类型的小数秒精度规定了秒字段中小数点后面的位数,取值范围为 0~6,如果未定义,缺省精度为 0。

3. **TIMESTAMP/DATETIME** 类型

语法:

TIMESTAMP [(小数秒精度)]

DATETIME [(小数秒精度)]

功能:

TIMESTAMP/DATETIME 类型包括年、月、日、时、分、秒信息,定义了一个在'-4712-01-01 00:00:00.000000'和'9999-12-31 23:59:59.999999'之间的有效格里高利日期时间。小数秒精度规定了秒字段中小数点后面的位数,取值范围为 0~6,如果未定义,缺省精度为 6。

2.1.4.2 时区数据类型

1. **TIME WITH TIME ZONE** 类型

语法:

TIME [(小数秒精度)] WITH TIME ZONE

功能:

描述一个带时区的 TIME 值,其定义是在 TIME 类型的后面加上时区信息。时区部分的实质是 INTERVAL HOUR TO MINUTE 类型,取值范围: -12:59 与+14:00 之间。例如: TIME '09:10:21 +8:00'。

2. **TIMESTAMP WITH TIME ZONE** 类型

语法:

TIMESTAMP [(小数秒精度)] WITH TIME ZONE

功能:

描述一个带时区的 `TIMESTAMP` 值，其定义是在 `TIMESTAMP` 类型的后面加上时区信息。时区部分的实质是 `INTERVAL HOUR TO MINUTE` 类型，取值范围：-12:59 与 +14:00 之间。例如：'2009-10-11 19:03:05.0000 -02:10'。

3. `TIMESTAMP WITH LOCAL TIME ZONE` 类型

语法：

`TIMESTAMP [(小数秒精度)] WITH LOCAL TIME ZONE`

功能：

描述一个本地时区的 `TIMESTAMP` 值，能够将标准时区类型 `TIMESTAMP WITH TIME ZONE` 类型转化为本地时区类型，如果插入的值没有指定时区，则默认为本地时区。

2.1.4.3 时间间隔数据类型

DM 支持两类十三种时间间隔类型：两类是年-月间隔类和日-时间隔类，它们通过时间间隔限定符区分，前者结合了日期字段年和月，后者结合了时间字段日、时、分、秒。由时间间隔数据类型所描述的值总是有符号的。

对时间间隔类型的介绍见表 2.1，需要查看更为详细的信息可参看《DM8_SQL 语言使用手册》1.4.3 节。

表 2.1 DMSQL 程序支持的时间间隔数据类型

类型名	类型描述
<code>INTERVAL YEAR(P)</code>	年间隔，即两个日期之间的年数字，P 为时间间隔的首项字段精度（后面简称为：首精度）
<code>INTERVAL MONTH(P)</code>	月间隔，即两个日期之间的月数字，P 为时间间隔的首精度
<code>INTERVAL DAY(P)</code>	日间隔，即为两个日期/时间之间的日数字，P 为时间间隔的首精度
<code>INTERVAL HOUR(P)</code>	时间隔，即为两个日期/时间之间的时数字，P 为时间间隔的首精度
<code>INTERVAL MINUTE(P)</code>	分间隔，即为两个日期/时间之间的分数字，P 为时间间隔的首精度
<code>INTERVAL SECOND(P,Q)</code>	秒间隔，即为两个日期/时间之间的秒数字，P 为时间间隔的首精度，Q 为时间间隔秒精度
<code>INTERVAL YEAR(P) TO MONTH</code>	年月间隔，即两个日期之间的年月数字，P 为时间间隔的首精度
<code>INTERVAL DAY(P) TO HOUR</code>	日时间隔，即为两个日期/时间之间的日时数字，P 为时间间隔的首精度
<code>INTERVAL DAY(P) TO MINUTE</code>	日时分间隔，即为两个日期/时间之间的日时分数字，P 为时间间隔的首精度

INTERVAL DAY (P) TO SECOND (Q)	日时分秒间隔，即为两个日期/时间之间的日时分秒数字，P 为时间间隔的首精度，Q 为时间间隔秒精度
INTERVAL HOUR (P) TO MINUTE	时分间隔，即为两个日期/时间之间的时分数字，P 为时间间隔的首精度
INTERVAL HOUR (P) TO SECOND (Q)	日时分秒间隔，即为两个日期/时间之间的日时分秒数字，P 为时间间隔的首精度，Q 为时间间隔秒精度
INTERVAL MINUTE (P) TO SECOND (Q)	分秒间隔，即为两个日期/时间之间的分秒间隔，P 为时间间隔的首精度，Q 为时间间隔秒精度

2.1.5 BOOL/BOOLEAN 数据类型

语法：

BOOL

BOOLEAN

功能：

布尔数据类型：TRUE 和 FALSE。DMSQL 程序的布尔类型和 INT 类型可以相互转化。如果变量或方法返回的类型是布尔类型，则返回值为 0 或 1。TRUE 和非 0 值的返回值为 1，FALSE 和 0 值返回为 0。

2.2 %TYPE 和%ROWTYPE

在许多时候，DMSQL 程序变量被用来处理存储在数据库表中的数据。这种情况下，变量应该拥有与表列相同的类型。例如表 T (ID INT, NAME VARCHAR(30)) 中字段 NAME 类型为 VARCHAR(30)。对应地在 DMSQL 程序中，我们可以声明一个变量：

```
DECLARE
    V_NAME VARCHAR(30);
```

但是如果 T 中的 NAME 字段定义发生了变化，比如变为 VARCHAR(100)。那么存储过程中的变量 V_NAME 也要相应修改为：

```
DECLARE
    V_NAME VARCHAR(100);
```

如果用户应用中有很多的变量以及 DMSQL 程序代码，这种处理可能是十分耗时和容易出错的。

为了解决上述问题，DMSQL 程序提供了%TYPE 类型。%TYPE 可以将变量同表列的类型进行绑定。例如：

```
DECLARE

    V_NAME T.NAME%TYPE;
```

通过使用%TYPE，V_NAME 将拥有 T 表的 NAME 列所拥有的类型。如果表 T 的 NAME 列类型定义发生变化，V_NAME 的类型也随之自动发生变化，而不需要用户手动修改。

例如，使用%TYPE 把变量 v1 的类型和表 PERSON.ADDRESS 的 ADDRESS1 列类型进行绑定。

```
DECLARE

    v_type PERSON.ADDRESS.ADDRESS1%TYPE;

BEGIN

    SELECT ADDRESS1 INTO v_type FROM PERSON.ADDRESS WHERE ADDRESSID=1;

    PRINT v_type;

END;

/
```

与%TYPE 类似，%ROWTYPE 将返回一个基于表定义的运算类型，它将一个记录声明为具有相同类型的数据库行。例如：

```
DECLARE

    V_TREC T % ROWTYPE;
```

将定义一个记录，该记录中的字段与表 T 中的行相对应。V_TREC 变量会拥有这样的结构：(ID INT, NAME VARCHAR(30))。如果表定义改变了，那么%ROWTYPE 定义的变量也会随之改变。

例如，使用%ROWTYPE 将变量 v_row 与表 PERSON.ADDRESS 的行相对应。

```
DECLARE

    v_row PERSON.ADDRESS%ROWTYPE;

    cur CURSOR;

BEGIN

    OPEN cur FOR SELECT * FROM PERSON.ADDRESS WHERE ADDRESSID=3;

    FETCH cur INTO v_row;

    PRINT v_row.ADDRESSID;
```

```
PRINT v_row.ADDRESS1;

CLOSE cur;

END;
```



在 DM8SQL 程序设计中使用%TYPE 和%ROWTYPE 是一种非常好的编程风格，它使得 DM8SQL 程序更加灵活，更适应于对数据库的处理。

2.3 记录类型

记录类型是由单行多列的标量类型构成复合类型，类似于 C 语言中的结构。记录类型提供了处理分立但又作为一个整体单元的相关变量的一种机制。例如：DECLARE V_ID INT; V_NAME VARCHAR(30); 这两个变量在逻辑上是相互关联的，因为它们分别对应表 T(ID INT, NAME VARCHAR(30)) 中的两个字段。如果为这样的变量声明一个记录类型，那么它们之间的关系就十分明显了。

定义记录类型的语法如下：

```
TYPE <记录类型名> IS RECORD

(<字段名><数据类型> [<default 子句>]{,<字段名><数据类型> [<default 子句>]});

<default 子句> ::= <default 子句 1> | <default 子句 2>

<default 子句 1> ::= DEFAULT <缺省值>

<default 子句 2> ::= := <缺省值>
```

通过将需要操作的表结构定义成一个记录，可以方便地对表中的行数据进行操作。在 DM8SQL 程序中使用记录，需要先定义一个 RECORD 类型，再用该类型声明变量，也可以使用上一小节介绍的%ROWTYPE 来创建与表结构匹配的记录。

可以单独对记录中的字段赋值，使用点标记引用一个记录中的字段（记录名.字段名）。

例如，下面的例子定义了一个记录类型 sale_person，声明一个该记录类型的变量 v_rec，使用点标记为 v_rec 的两个字段赋值，之后使用 v_rec 更新表的一行数据。

```
DECLARE

TYPE sale_person IS RECORD(

    ID SALES.SALESPERSON.SALESPERSONID%TYPE,

    SALESTHISYEAR SALES.SALESPERSON.SALESTHISYEAR%TYPE);
```

```

    v_rec sale_person;

BEGIN

    v_rec.ID := 1;

    v_rec.SALESTHISYEAR:= 5500;

    UPDATE SALES.SALESPERSON SET SALESTHISYEAR=v_rec.SALESTHISYEAR WHERE
SALESPERSONID =v_rec.ID;

END;

/

```

也可以将一个记录直接赋值给另外一个记录，此时两个记录中的字段类型定义必须完全一致。如下面的例子将表中的一行数据读取到一个记录中。然后，将记录 `v_rec1` 赋值给 `v_rec2`。

```

DECLARE

    TYPE t_rec IS RECORD( ID INT, NAME VARCHAR(50));

    TYPE t_rec_NEW IS RECORD( ID INT, NAME VARCHAR(50));

    v_rec1 T_REC;

    v_rec2 T_REC_NEW;

BEGIN

    SELECT PRODUCTID,NAME INTO v_rec1 FROM PRODUCTION.PRODUCT WHERE AUTHOR
LIKE '鲁迅';

    v_rec2 := v_rec1;

    PRINT v_rec2.ID;

    PRINT v_rec2.NAME;

END;

/

```

定义记录类型时，字段的数据类型除了可以是常规数据类型，还可以是常规数据类型后跟着 “[n]” 或 “[n1,n2...]” 表示一维或多维数组，如：

```

DECLARE

    TYPE T_REC IS RECORD( ID INT[3], NAME VARCHAR(30)[3]);

```

DM8SQL 程序还支持定义包含数组、集合和其他 RECORD 的 RECORD。例如下面是一个在 RECORD 定义中包含其他 RECORD 的例子，关于数组和集合的介绍请看后续小节。

```

DECLARE

    TYPE TimeType IS RECORD (hours INT, minutes INT );      --定义记录 TimeType

    TYPE MeetingType IS RECORD (

        day      DATE,

        time_of TimeType      -- 嵌套记录 TimeType

    );

BEGIN

    NULL;

END;

/

```

2.4 数组类型

DM8SQL 程序支持数组数据类型，包括静态数组类型和动态数组类型。

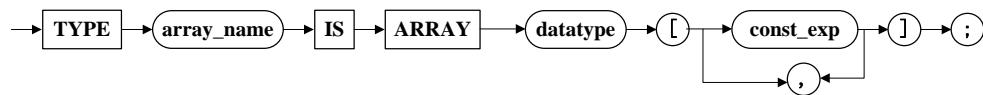


注意： DM数组下标的起始值为1。

2.4.1 静态数组类型

静态数组是在声明时已经确定了数组大小的数组，其长度是预先定义好的，在整个程序中，一旦给定大小后就无法改变。

定义静态数组类型的语法图例如下：



定义了静态数组类型后需要用这个类型申明一个数组变量然后进行操作。

理论上 DM 支持静态数组的每一个维度的最大长度为 65534，但是静态数组最大长度同时受系统内部堆栈空间大小的限制，如果超出堆栈的空间限制，系统会报错。

下面是一个使用静态数组的例子：

```

DECLARE

    TYPE Arr IS ARRAY VARCHAR[3];      --TYPE 定义一维数组类型

```

```

a Arr;                                --声明一维数组

TYPE Arr1 IS ARRAY VARCHAR[2,4]; --TYPE 定义二维数组类型

b Arr1;                                --声明二维数组

BEGIN

  FOR I IN 1..3 LOOP

    a[I] := I * 10;

    PRINT a[I];

  END LOOP;

  PRINT '-----';

  FOR I IN 1..2 LOOP

    FOR J IN 1..4 LOOP

      b[I][J] = 4*(I-1)+J;

      PRINT b[I][J];

    END LOOP;

  END LOOP;

END;

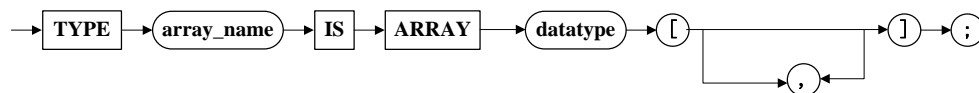
/

```

2.4.2 动态数组类型

与静态数组不同，动态数组可以随程序需要而重新指定大小，其内存空间是从堆（HEAP）上分配（即动态分配）的，通过执行代码而为其分配存储空间，并由 DM 自动释放内存。

动态数组与静态数组的定义方法类似，区别只在于动态数组没有指定下标，需要动态分配空间。定义动态数组类型的语法图例如下：



定义了动态数组类型后需要用这个类型申明一个数组变量，之后在 DMSQL 程序的执行部分需要为这个数组变量动态分配空间。动态分配空间语句如下所示：

```
数组变量名 := NEW 数据类型[常量表达式,...];
```


或者可以使用如下语句对多维数组的某一维度进行空间分配：

```
数组变量名 := NEW 数据类型[常量表达式][ ];
```

下面给出了一个使用动态数组的例子：

```
DECLARE

    TYPE Arr IS ARRAY VARCHAR[ ];

    a Arr;

BEGIN

    a := NEW VARCHAR[4]; --动态分配空间

    FOR I IN 1..4 LOOP

        a[I] := I * 4;

        PRINT a[I];

    END LOOP;

END;

/
```

对于多维动态数组，可以单独为每个维度动态分配空间，如下面的例子所示：

```
DECLARE

    TYPE Arr1 IS ARRAY VARCHAR[ , ];

    b Arr1;

BEGIN

    b := NEW VARCHAR[2][ ]; --动态分配第一维空间

    FOR I IN 1..2 LOOP

        b[I] := NEW VARCHAR[4]; --动态分配第二维空间

        FOR J IN 1..4 LOOP

            b[I][J] = I*10+J;

            PRINT b[I][J];

        END LOOP;

    END LOOP;

END;

/
```

也可以一次性为多维动态数组分配空间，则上面的例子可以写为：

```

DECLARE

    TYPE Arr1 IS ARRAY VARCHAR[,];

    b ARR1;

BEGIN

    b := NEW VARCHAR[2,4];

    FOR I IN 1..2 LOOP

        FOR J IN 1..4 LOOP

            b[I][J]= I*10+J;

            PRINT b[I][J];

        END LOOP;

    END LOOP;

END;

/

```

理论上 DM 支持动态数组的每一个维度的最大长度为 2147483646，但是数组最大长度同时受系统内部堆空间大小的限制，如果超出堆的空间限制，系统会报错。

2.4.3 复杂类型数组

除了普通数据类型的数组，DM 还支持自定义类型、记录类型和集合类型的数组。

在下面的例子中，定义了一个自定义类型（OBJECT 类型）的静态数组，存放图书的序号和名称。

```

CREATE OR REPLACE TYPE COMPLEX AS OBJECT(

    RPART      INT,

    IPART      VARCHAR(100)

);

/

DECLARE

    TYPE ARR_COMPLEX IS ARRAY SYSDBA.COMPLEX[3];

    arr ARR_COMPLEX;

```

```

BEGIN

    FOR I IN 1..3 LOOP

        SELECT SYSDBA.COMPLEX(PRODUCTID, NAME) INTO arr[I] FROM
PRODUCTION.PRODUCT WHERE PRODUCTID=I;

        PRINT arr[I].RPART || arr[I].IPART;

    END LOOP;

END;

/

```

也可以将上例中的对象类型改为记录类型，则 DMSQL 程序可写为：

```

DECLARE

    TYPE REC IS RECORD(ID INT, NAME VARCHAR(128));

    TYPE REC_ARR IS ARRAY REC[3];

    arr REC_ARR;

BEGIN

    FOR I IN 1..3 LOOP

        SELECT PRODUCTID, NAME INTO arr[I] FROM PRODUCTION.PRODUCT WHERE
PRODUCTID=I;

        PRINT arr[I].ID || arr[I].NAME;

    END LOOP;

END;

/

```

下面的例子定义了一个集合类型（以 VARRAY 为例）的数组，将员工的姓名、性别和职位信息存放到数组变量中。

```

DECLARE

    TYPE VARY IS VARRAY(3) OF varchar(100);

    TYPE ARR_VARY IS ARRAY VARY[8];

    arr ARR_VARY;

    v1,v2,v3 varchar(50);

BEGIN

```

```

FOR I IN 1..8 LOOP

    SELECT NAME,PERSON.SEX,TITLE INTO v1,v2,v3 FROM
PERSON.PERSON,RESOURCES.EMPLOYEE WHERE PERSON.PERSONID=EMPLOYEE.PERSONID AND
PERSON.PERSONID=I;

    arr[I] := VARY(v1,v2,v3);

    PRINT '*****工号'||I||'*****';

    FOR J IN 1..3 LOOP

        PRINT arr[I][J];

    END LOOP;

END LOOP;

END;

/

```

2.5 集合类型

DM8SQL 程序支持三种集合类型：VARRAY 类型、索引表类型和嵌套表类型。

2.5.1 VARRAY

VARRAY 是一种具有可伸缩性的数组，数组中的每个元素具有相同的数据类型。

VARRAY 在定义时由用户指定一个最大容量，其元素索引是从 1 开始的有序数字。

定义 VARRAY 的语法格式为：

```
TYPE<数组名> IS VARRAY (<常量表达式>) OF <数据类型>;
```

<常量表达式>表示数组的最大容量。

<数据类型>是 VARRAY 中元素的数据类型，可以是常规数据类型，也可以是其他自定义类型或对象、记录、其他 VARRAY 类型等，使得构造复杂的结构成为可能。

在定义了一个 VARRAY 数组类型后，再声明一个该数组类型的变量，就可以对这个数组变量进行操作了。如下面的代码片段所示：

```

TYPE my_array_type IS VARRAY(10) OF INTEGER;

v MY_ARRAY_TYPE;

```

使用 `v.COUNT()` 方法可以得到数组 `v` 当前的实际大小，`v.LIMIT()` 则可获得数组 `v` 的最大容量。需要注意的是，`VARRAY` 的元素索引总是连续的。

下面给出一个 `VARRAY` 的简单使用示例，查询人员姓名并将其存入一个 `VARRAY` 变量中。`VARRAY` 最初的实际大小为 0，使用 `EXTEND()` 方法可扩展 `VARRAY` 元素个数，具体在 2.5.4 节中介绍。

```
DECLARE

    TYPE MY_ARRAY_TYPE IS VARRAY(10) OF VARCHAR(100);

    v MY_ARRAY_TYPE;

BEGIN

    v:=MY_ARRAY_TYPE();

    PRINT 'v.COUNT()=' || v.COUNT();

    FOR I IN 1..8 LOOP

        v.EXTEND();

        SELECT NAME INTO v(I) FROM PERSON.PERSON WHERE PERSON.PERSONID=I;

    END LOOP;

    PRINT 'v.COUNT()=' || v.COUNT();

    FOR I IN 1..v.COUNT() LOOP

        PRINT 'v(' || i || ')= ' || v(i);

    END LOOP;

END;

/
```

2.5.2 索引表

索引表提供了一种快速、方便地管理一组相关数据的方法。索引表是一组数据的集合，它将数据按照一定规则组织起来，形成一个可操作的整体，是对大量数据进行有效组织和管理的的手段之一，通过函数可以对大量性质相同的数据进行存储、排序、插入及删除等操作，从而可以有效地提高程序开发效率及改善程序的编写方式。

索引表不需要用户指定大小，其大小根据用户的操作自动增长。

定义索引表的语法格式为：

```
TYPE <索引表名> IS TABLE OF<数据类型> INDEX BY <索引数据类型>;
```

“数据类型”指索引表存放的数据的类型，这个数据类型可以是常规数据类型，也可以是其他自定义类型或是对象、记录、静态数组，但不能是动态数组；“索引数据类型”则是索引表中元素索引的数据类型，DM 目前仅支持 INTEGER/INT 和 VARCHAR 两种类型，分别代表整数索引和字符串索引。对于 VARCHAR 类型，长度不能超过 1024。

用户可使用“索引-数据”对向索引表插入数据，之后可通过“索引”修改和查询这个数据，而不需要知道数据在索引表中实际的位置。

下面给出了一个非常简单的索引表的使用示例。

```
DECLARE

    TYPE Arr IS TABLE OF VARCHAR(100) INDEX BY INT;

    x Arr;

BEGIN

    x(1) := 'TEST1';

    x(2) := 'TEST2';

    x(3) := x(1) || x(2);

    PRINT x(3);

END;

/
```

索引表还可以用来管理记录，下面的例子索引表中存放的数据类型为 RECORD，展示了如何存入和遍历输出索引表的记录类型数据。

```
DECLARE

    TYPE Rd IS RECORD(ID INT, NAME VARCHAR(128));

    TYPE Arr IS TABLE OF Rd INDEX BY INT;

    x Arr;

    i INT;

    CURSOR C1;

BEGIN

    i := 1;

    OPEN C1 FOR SELECT PERSONID, NAME FROM PERSON.PERSON;
```

```

--遍历结果集，把每行的值都放入索引表中

LOOP

    IF C1%NOTFOUND THEN

        EXIT;

    END IF;

    FETCH C1 INTO x(i).ID, x(i).NAME;

    i := i + 1;

END LOOP;

--遍历输出索引表中的记录

i = x."FIRST"();

LOOP

    IF i IS NULL THEN

        EXIT;

    END IF;

    PRINT 'ID:' || CAST(x(i).ID AS VARCHAR(10)) || ', NAME:' || x(i).NAME;

    i = x."NEXT"(i);

END LOOP;

END;

/

```

下面的例子定义了一个二维索引表 `x`，展示了如何存入和遍历输出二维索引表的数据。

```

DECLARE

    TYPE Arr IS TABLE OF VARCHAR(100) INDEX BY BINARY_INTEGER;

    TYPE Arr2 IS TABLE OF Arr INDEX BY VARCHAR(100);

    x Arr2;

    ind_i INT;

    ind_j VARCHAR(10);

BEGIN

    --存入数据

    FOR I IN 1 .. 6 LOOP

        FOR J IN 1 .. 3 LOOP

```

```
x(I)(J) := CAST(I AS VARCHAR(100)) || '+' || CAST(J AS VARCHAR(10));

END LOOP;

END LOOP;

--遍历多维数组

ind_i := x."FIRST"();

LOOP

    IF ind_i IS NULL THEN

        EXIT;

    END IF;

    ind_j := x(ind_i)."FIRST"();

    LOOP

        IF ind_j IS NULL THEN

            EXIT;

        END IF;

        PRINT x(ind_i)(ind_j);

        ind_j := x(ind_i)."NEXT"(ind_j);

    END LOOP;

    ind_i := x."NEXT"(ind_i);

END LOOP;

END;

/
```

2.5.3 嵌套表

嵌套表类似于一维数组，但与数组不同的是，嵌套表不需要指定元素的个数，其大小可自动扩展。嵌套表元素的下标从 1 开始。

定义嵌套表的语法格式为：

```
TYPE <嵌套表名> IS TABLE OF <元素数据类型>;
```

元素数据类型用来指明嵌套表元素的数据类型，当元素数据类型为一个定义了某个表记录的对象类型时，嵌套表就是某些行的集合，实现了表的嵌套功能。

下面的例子定义了一个嵌套表，其结构与 SALES.SALESPERSON 表相同，用来存放今年销售额大于 1000 万的销售代表的信息。

```
DECLARE

    TYPE Info_t IS TABLE OF SALES.SALESPERSON%ROWTYPE;

    info Info_t;

BEGIN

    SELECT SALESPERSONID,EMPLOYEEID,SALESTHISYEAR,SALESLASTYEAR  BULK COLLECT
INTO info FROM SALES.SALESPERSON WHERE SALESTHISYEAR>1000;

END;

/
```

2.5.4 集合类型支持的方法

DM 为 VARRAY、索引表和嵌套表提供了一些方法，用户可以使用这些方法访问和修改集合与集合元素。

1. COUNT

语法：

<集合变量名>.COUNT

功能：

返回集合中元素的个数。

2. LIMIT

语法：

<VARRAY 变量名>.LIMIT

功能：

返回 VARRAY 集合的最大的元素个数，对索引表和嵌套表不适用。

3. FIRST

语法：

<集合变量名>.FIRST

功能：

返回集合中的第一个元素的下标号，对于 VARRAY 集合始终返回 1。

4. LAST

语法:

<集合变量名>.LAST

功能:

返回集合中最后一个元素的下标号，对于 VARRAY 返回值始终等于 COUNT。

5. NEXT

语法:

<集合变量名>.NEXT (<下标>)

参数:

指定的元素下标。

功能:

返回在指定元素 i 之后紧挨着它的元素的下标号，如果指定元素是最后一个元素，则返回 NULL。

6. PRIOR

语法:

<集合变量名>.PRIOR (<下标>)

参数:

指定的元素下标。

功能:

返回在指定元素 i 之前紧挨着它的元素的下标号，如果指定元素是第一个元素，则返回 NULL。

7. EXISTS

语法:

<集合变量名>.EXISTS (<下标>)

参数:

指定的元素下标。

功能:

如果指定下标对应的元素已经初始化，则返回 TRUE，否则返回 FALSE。

8. DELETE

语法:

<集合变量名>.DELETE ([<下标>])

参数:

待删除元素的下标。

功能:

下标参数省略时删除集合中所有元素，否则删除指定下标对应的元素，如果指定下标为 NULL，则集合保持不变。

9. DELETE

语法:

<集合变量名>.DELETE (<下标 1>, <下标 2>)

参数:

下标 1: 要删除的第一个元素的下标;

下标 2: 要删除的最后一个元素的下标。

功能:

删除集合中下标从下标 1 到下标 2 的所有元素。

10. TRIM

语法:

<集合变量名>.TRIM ([<n>])

参数:

删除元素的个数。

功能:

n 参数省略时从集合末尾删除一个元素，否则从集合末尾开始删除 n 个元素。本方法不适用于索引表。

11. EXTEND

语法:

<集合变量名>.EXTEND ([<n>])

参数:

扩展元素的个数。

功能:

n 参数省略时在集合末尾扩展一个空元素，否则在集合末尾扩展 n 个空元素。本方法不适用于索引表。

12. EXTEND

语法：

<集合变量名>.EXTEND(<n>,<下标>])

参数：

n: 扩展元素的个数；

下标: 待复制元素的下标。

功能：

在集合末尾扩展 n 个与指定下标元素相同的元素。本方法不适用于索引表。

下面的例子简单演示了如何使用集合类型的方法：

```
DECLARE

    TYPE IntList IS TABLE OF INT;

    v IntList := IntList(1,3,5,7,9);      -- 开始赋值 5 个元素.

BEGIN

    DBMS_OUTPUT.PUT_LINE

        ('最初 v 中共有 ' || v.COUNT || ' 个元素。');

    v.EXTEND(3); -- 在末尾增加三个元素

    DBMS_OUTPUT.PUT_LINE

        ('现在 v 中共有 ' || v.COUNT || ' 个元素。');

    v.DELETE(2);      -- 删掉第 2 个元素

    DBMS_OUTPUT.PUT_LINE

        ('现在 v 中共有 ' || v.COUNT || ' 个元素。');

    v.TRIM(2); -- 删除掉末尾的两个元素

    DBMS_OUTPUT.PUT_LINE

        ('现在 v 中共有 ' || v.COUNT || ' 个元素。');

END;

/
```

2.6 类类型

DM 支持类类型，类将结构化的数据及对其进行操作的过程或函数封装在一起。允许用户根据现实世界的对象建模，而不必再将其抽象成关系数据。关于类类型的详细介绍可以参考《DM8_SQL 语言使用手册》的相关章节。

DMSQL 程序中可以声明一个类类型的变量，初始化该变量后就可以访问类的成员变量，调用类的成员方法了。

2.7 子类型

子类型是其基数据类型的子集。子类型具有与基数据类型相同的操作性质，但是其有效值域是基数据类型的子集。

定义子类型的语法如下：

```
SUBTYPE <subtype_name> IS <base_type>[(<精度>,[<刻度>])] [NOT NULL];
```

例如：

```
DECLARE

    SUBTYPE COUNTER IS NUMBER(5);

    C COUNTER;

BEGIN

    NULL;

END;
```

在这个例子中定义了一个名称为 COUNTER 的子类型，其实际数据类型为 NUMBER(5)。子类型可用来防止变量超出规定的值域，也可以增强 DMSQL 程序的可读性。

可以在任何 DMSQL 程序块、子程序或包中定义自己的子类型。一旦定义了子类型，就可以声明该类型的变量、常量等，如上例中的变量 C。

2.8 操作符

与其他程序设计语言相同，DMSQL 程序有一系列操作符。操作符分为下面几类：

- 算术操作符
- 关系操作符

- 比较操作符
- 逻辑操作符

算术操作符如表 2.1 所示。

表 2.1 算术操作符

操作符	对应操作
+	加
-	减
*	乘
/	除

关系操作符主要用于条件判断语句或用于 WHERE 子句中，关系操作符检查条件和结果为 TRUE 或 FALSE。表 2.2、表 2.3、表 2.4 分别列出了 DM8SQL 程序中的关系操作符、比较操作符和逻辑操作符。

表 2.2 关系操作符

操作符	对应操作
<	小于操作符
<=	小于或等于操作符
>	大于操作符
>=	大于或等于操作符
=	等于操作符
!=	不等于操作符
<>	不等于操作符
:=	赋值操作符

表 2.3 比较操作符

操作符	对应操作
IS NULL	如果操作数为 NULL 返回 TRUE
LIKE	比较字符串值
BETWEEN	验证值是否在范围之内
IN	验证操作数在设定的一系列值中

表 2.4 逻辑操作符

操作符	对应操作
AND	两个条件都必须满足
OR	只要满足两个条件中的一个
NOT	取反

3 DMSQL 程序的定义、调用与删除

本章主要介绍如何使用 DMSQL 程序语言来定义存储模块及客户端 DMSQL 程序。为用户获得对 DMSQL 程序的整体印象，本节中提供了一些例子，对于例子中涉及到的各种控制语句，其详细的使用方法将在第五章进行介绍。

3.1 存储过程

定义存储过程的语法如下：

```
CREATE [OR REPLACE ] PROCEDURE<过程声明><AS_OR_IS><模块体>

<过程声明> ::= <存储过程名定义> [WITH ENCRYPTION] [ (<参数名><参数模式><参数类型> [<默认值表达式>] {,<参数名><参数模式><参数类型> [<默认值表达式>] }) ] [<调用权限子句>]

<存储过程名定义> ::= [<模式名>.]<存储过程名>

<AS_OR_IS> ::= AS | IS

<模块体> ::= [<声明部分>

    BEGIN

    <执行部分>

    [<异常处理部分>]

    END [<存储过程名>]

<声明部分> ::= [DECLARE] <声明定义> {<声明定义>}

<声明定义> ::= <变量声明> | <异常变量声明> | <游标定义> | <子过程定义> | <子函数定义>;

<执行部分> ::= <DMSQL 程序语句序列> {;<DMSQL 程序语句序列>}

<DMSQL 程序语句序列> ::= [<标号说明>] <DMSQL 程序语句>;

<标号说明> ::= <<<标号名>>>

<DMSQL 程序语句> ::= <SQL 语句> | <控制语句>

<异常处理部分> ::= EXCEPTION <异常处理语句> {;<异常处理语句>}
```

参数说明：

- <存储过程名>：指明被创建的存储过程的名字
- <模式名>：指明被创建的存储过程所属模式的名字，缺省为当前模式名

- <参数名>: 指明存储过程参数的名称
- <参数模式>: 参数模式可设置为 IN、OUT 或 IN OUT (OUT IN), 缺省为 IN 类型
- <参数类型>: 指明存储过程参数的数据类型
- <声明部分>: 由变量、游标和子程序等对象的声明构成, 可缺省
- <执行部分>: 由 SQL 语句和过程控制语句构成的执行代码
- <异常处理部分>: 各种异常的处理程序, 存储过程执行异常时调用, 可缺省
- <调用权限子句>: 指定该过程中的 SQL 语句默认的模式

DBA 或具有 CREATE PROCEDURE 权限的用户可以使用上述语法新创建一个存储过程。OR REPLACE 选项的作用是当同名的存储过程存在时, 首先将其删除, 再创建新的存储过程, 前提条件是当前用户具有删除原存储过程的权限, 如果没有删除权限, 则创建失败。使用 OR REPLACE 选项重新定义存储过程后, 由于不能保证原有对象权限的合法性, 所以全部去除。

WITH ENCRYPTION 为可选项, 如果指定 WITH ENCRYPTION 选项, 则对 BEGIN 到 END 之间的语句块进行加密, 防止非法用户查看其具体内容。加密后的存储过程的定义可在 SYS.SYSTEXTS 系统表中查询。

存储过程可以带有参数, 这样在调用存储过程时就需指定相应的实际参数, 如果没有参数, 过程名后面的圆括号和参数列表就可以省略了。关于参数使用的具体介绍见 3.4 节。

声明部分进行变量声明, 对其的具体介绍见 3.5 节。

可执行部分是存储过程的核心部分, 由 SQL 语句和流控制语句构成。支持的 SQL 语句包括:

- ✓ 数据查询语句 (SELECT)
- ✓ 数据操纵语句 (INSERT、DELETE、UPDATE)
- ✓ 游标定义及操纵语句 (DECLARE CURSOR、OPEN、FETCH、CLOSE)
- ✓ 事务控制语句 (COMMIT、ROLLBACK)
- ✓ 动态 SQL 执行语句 (EXECUTE IMMEDIATE)



注意:

SQL 语句必须以分号结尾, 否则语法分析报错。

异常处理部分用于处理存储过程在执行过程中可能出现的错误。

下面是一个定义存储过程的简单示例：

```

1      CREATE OR REPLACE PROCEDURE RESOURCES.proc_1(a IN OUT INT)  AS
2
3          b  INT:=10;
4
5      BEGIN
6
7          a:=a+b;
8
9          PRINT a;
10
11     EXCEPTION
12
13         WHEN OTHERS THEN NULL;
14
15     END;
16
17 /

```

该例子在模式 RESOURCES 下创建了一个名为 proc_1 的存储过程。例子中第 2 行是该存储过程的说明部分，这里声明了一个变量 b。注意在 DM8SQL 程序中说明变量时，变量的类型放在变量名称之后。第 4 行和第 5 行是该程序块运行时被执行的代码段，这里将 a 与 b 的和赋给参数 a。如果发生了异常，第 6 行开始的异常处理部分就对产生的异常情况进行处理。说明部分和异常处理部分都是可选的。如果用户在模块中不需要任何局部变量或者不想处理发生的异常，则可以省略这两部分。

3.2 存储函数

定义存储函数的语法如下：

```

CREATE [OR REPLACE ] FUNCTION <函数声明><AS_OR_IS><模块体>

<函数声明> ::= <存储函数名定义> [WITH ENCRYPTION] [FOR CALCULATE] [(<参数名><参数模式><参数类型> [<默认值表达式>] {,<参数名><参数模式><参数类型> [<默认值表达式>]}) ] RETURN
<返回数据类型> [<调用选项子句>] [PIPELINED]

<存储函数名定义> ::= [<模式名>.]<存储函数名>

<调用选项子句> ::= <调用选项> {<调用选项>}

<调用选项> ::= <调用权限子句> | DETERMINISTIC

<AS_OR_IS> ::= AS | IS

<模块体> ::= [<声明部分>]

```

BEGIN

<执行部分>

[<异常处理部分>]

END [存储函数名]

<声明部分> ::= [DECLARE] <声明定义> {<声明定义>}

<声明定义> ::= <变量声明> | <异常变量声明> | <游标定义> | <子过程定义> | <子函数定义>;

<执行部分> ::= <DMSQL 程序语句序列> {;<DMSQL 程序语句序列>}

<DMSQL 程序语句序列> ::= [<标号说明>] <DMSQL 程序语句>;

<标号说明> ::= <<<标号名>>>

<DMSQL 程序语句> ::= <SQL 语句> | <控制语句>

<异常处理部分> ::= EXCEPTION <异常处理语句> {;<异常处理语句>}

参数说明:

- <存储函数名>: 指明被创建的存储函数的名字
- <模式名>: 指明被创建的存储函数所属模式的名字, 缺省为当前模式名
- <参数名>: 指明存储函数参数的名称
- <参数模式>: 参数模式可设置为 IN、OUT 或 IN OUT (OUT IN), 缺省为 IN 类型
- <参数类型>: 指明存储函数参数的数据类型
- <返回数据类型>: 指明存储函数返回值的数据类型
- <调用权限子句>: 指定该过程中的 SQL 语句默认的模式
- PIPELINED: 指明函数为管道表函数

存储函数与存储过程在结构和功能上十分相似, 主要的差异在于:

- 存储过程没有返回值, 调用者只能通过访问 OUT 或 IN OUT 参数来获得执行结果, 而存储函数有返回值, 它把执行结果直接返回给调用者;
- 存储过程中可以没有返回语句, 而存储函数必须通过返回语句结束;
- 不能在存储过程的返回语句中带表达式, 而存储函数必须带表达式;
- 存储过程不能出现在一个表达式中, 而存储函数可以出现在表达式中。

FOR CALCULATE 指定存储函数为计算函数。计算函数中不支持对表进行 INSERT、DELETE、UPDATE、SELECT、上锁、设置自增列属性；对游标 DECLARE、OPEN、FETCH、CLOSE；事务的 COMMIT、ROLLBACK、SAVEPOINT、设置事务的隔离级别和读写属性；动态 SQL 的执行 EXEC、创建 INDEX、创建子过程。对于计算函数体内的函数调用必须是系统函数或者计算函数。计算函数可以被指定为表列的缺省值。

DETERMINISTIC 指定存储函数为确定性函数。在调用其的语句中，对于相同的参数返回相同的结果。如果要将一个函数作为表达式在函数索引中使用，必须指定该函数为确定性函数。当系统遇到确定性函数，它将会试图重用之前的计算结果，而不是重新计算。在确定性函数实现中，虽然没有限制不确定元素（如随机函数等）和 SQL 语句的使用，但是不推荐使用这些可能会导致结果不确定的内容。确定性函数中不支持 BOOLEAN 类型、复合类型或对象类型作为参数及返回值。

下面是一个定义存储函数的简单示例：

```

1      CREATE OR REPLACE FUNCTION RESOURCES.fun_1(a INT, b INT) RETURN INT AS
2
3      s  INT;
4
5      BEGIN
6
7          s:=a+b;
8
9          RETURN s;
10
11     EXCEPTION
12
13         WHEN OTHERS THEN NULL;
14
15     END;
16
17 /

```

这个例子在模式 RESOURCES 下创建一个名为 fun_1 的存储函数。第 1 行说明了该函数的返回类型为 INT 类型。第 4 行将两个参数 a、b 的和赋给了变量 s，第 5 行的 RETURN 语句则将变量 s 的值作为函数的返回值返回。

调用这个存储函数：

```
SELECT RESOURCES.fun_1(1,2);
```

查询结果为 3。

下面的例子创建了一个计算函数 F1，并在表 T 中使用其定义列的缺省值。

```
CREATE OR REPLACE FUNCTION F1 FOR CALCULATE
```

```
RETURN INT

IS

BEGIN

RETURN 1;

END;

/

--在表 T 中使用

CREATE TABLE T(C1 INT, C2 INT DEFAULT F1());

或者 CREATE TABLE T(C1 INT, C2 INT DEFAULT F1);
```

3.3 客户端 DMSQL 程序

客户端 DMSQL 程序不需要存储，创建后立即执行，执行完毕即被释放。

客户端 DMSQL 程序的定义语法与存储过程的定义语法类似，如下：

```
[<声明部分>]

BEGIN

<执行部分>

[<异常处理部分>]

END
```



注意： 客户端DMSQL程序的声明部分必须包含**DECLARE**。

读者可能已经发现，本书之前章节的很多示例都是使用的客户端 DMSQL 程序，对于不需要反复执行的脚本，使用客户端 DMSQL 程序是一个比较合适的选择。

客户端 DMSQL 程序无法被其他程序调用，但它可以调用包括存储过程和存储函数等在内的其他函数。例如：

```
DECLARE

    r INT:=0;

BEGIN

    SELECT RESOURCES.fun_1(1,2)*2 INTO r;

    CALL RESOURCES.proc_1(r);
```

```

EXCEPTION

    WHEN OTHERS THEN NULL;

END;

/

```

3.4 参数

存储模块及模块中定义的子模块都可以带参数，用来给模块传送数据及向外界返回数据。在存储过程或存储函数中定义一个参数时，必须说明名称、参数模式和数据类型。三种可能的参数模式是：IN (缺省模式)、OUT 和 IN OUT，意义分别为：

- IN：输入参数，用来将数据传送给模块；
- OUT：输出参数，用来从模块返回数据到进行调用的模块；
- IN OUT：既作为输入参数，也作为输出参数。

在存储模块中使用参数时要注意下面几点：

- 最多能定义不超过 1024 个参数；
- IN 参数能被赋值；
- OUT 参数的初值始终为空，无论调用该模块时对应的实参值为多少；
- 调用一个模块时，OUT 参数及 IN OUT 参数的实参必须是可赋值的对象。

下面的例子说明了不同模式的参数的使用方法。例子中在客户端 DMSQL 程序中定义了一个子过程 raise_salary，其三个参数分别为 IN，IN OUT 和 OUT 类型。调用 raise_salary 为工号为 emp_num 的员工加薪 bonus 元，在 raise_salary 中将加薪后的薪水值赋给 IN OUT 参数 bonus，将员工职位赋给 OUT 参数 title。

```

DECLARE

    emp_num    INT := 1;

    bonus      DEC(19,4) := 6000;

    title      VARCHAR(50);

    PROCEDURE raise_salary (emp_id  IN INT, --输入参数
                           amount  IN OUT DEC(19,4), --输入输出参数
                           emp_title OUT VARCHAR(50) --输出参数
    )

```

```

        )

    IS

    BEGIN

        UPDATE RESOURCES.EMPLOYEE SET SALARY = SALARY + amount WHERE
EMPLOYEEID = emp_id;

        SELECT TITLE,SALARY INTO emp_title,amount FROM RESOURCES.EMPLOYEE
WHERE EMPLOYEEID = emp_id;

        END raise_salary;

BEGIN

    raise_salary(emp_num, bonus, title);

    DBMS_OUTPUT.PUT_LINE

        ('工号: '||emp_num||'    '||'职位: '||title||'    '||'加薪后薪水: '||bonus);

END;

/

```

执行这个例子，将打印如下信息：

```
工号: 1    职位: 总经理加薪后薪水: 46000.0000
```

使用赋值符号“:=”或关键字 DEFAULT，可以为 IN 参数指定一个缺省值。如果调用时未指定参数值，系统将自动使用该参数的缺省值。例如：

```

CREATE PROCEDURE proc_def_arg(a varchar(10) default 'abc', b INT:=123) AS

BEGIN

    PRINT a;

    PRINT b;

END;

/

```

调用过程 PROC_DEF_ARG，不指定输入参数值：

```
CALL proc_def_arg;
```

系统使用缺省值作为参数值，打印结果为：

```
abc
```

```
123
```

也可以只指定第一个参数，省略后面的参数：

```
CALL proc_def_arg('我们');
```

系统对后面的参数使用缺省值，打印结果为：

```
我们
```

```
123
```

3.5 变量

变量的声明应在声明部分，其语法为：

```
<变量名>{,<变量名>} [CONSTANT] <变量类型> [NOT NULL] [<缺省值定义符><表达式>]
```

```
<缺省值定义符> ::= DEFAULT | ASSIGN | :=
```

声明一个变量需要给这个变量指定名字及数据类型。

变量名必须以字母开头，包含数字、字母、下划线以及\$、#符号，长度不能超过 128 字符，并且不能与 DM 的 DMSQL 程序保留字相同，变量名与大小写是无关的。

变量的数据类型可以是基本的 SQL 数据类型，也可以是 DMSQL 程序数据类型，比如一个游标、异常等。

用赋值符号“:=”或关键字 DEFAULT、ASSIGN，可以在定义时为变量指定一个缺省值。

在 DMSQL 程序的执行部分可以对变量赋值，赋值语句有两种方式：

- 直接赋值语句，语法为：

```
<变量名>:=<表达式>
```

或

```
SET <变量名>=<表达式>
```

- 通过 SQL SELECT INTO 或 FETCH INTO 给变量赋值，语法如下，具体说明见后续章节。

```
SELECT <表达式>{,<表达式>} [INTO <变量名>{,<变量名>}] FROM <表引用>{,<表引用>} ...;
```

或

```
FETCH [NEXT|PREV|FIRST|LAST|ABSOLUTE N|RELATIVE N] <游标名> [INTO <变量名>{,<变量名>}];
```

常量与变量相似，但常量的值在程序内部不能改变，常量的值在定义时赋予，它的声明方式与变量相似，但必须包含关键字 `CONSTANT`。

如果需要打印变量的值，则要调用 `PRINT` 语句或 `DBMS_OUTPUT.PUT_LINE` 语句，如果数据类型不一致，则系统会自动将它转换为 `VARCHAR` 类型输出。除了变量的声明外，变量的赋值、输出等操作都要放在 `DMSQL` 程序的可执行部分。

下面的例子说明了如何对变量进行定义与赋值。

```
DECLARE  -- 可以在这里赋值

    salary          DEC(19,4);

    worked_time     DEC(19,4) := 60;

    hourly_salary   DEC(19,4) := 1055;

    bonus           DEC(19,4) := 150;

    position        VARCHAR(50);

    province        VARCHAR(64);

    counter         DEC(19,4) := 0;

    done            BOOLEAN;

    valid_id        BOOLEAN;

    emp_rec1        RESOURCES.EMPLOYEE%ROWTYPE;

    emp_rec2        RESOURCES.EMPLOYEE%ROWTYPE;

    TYPE meeting_type IS TABLE OF INT INDEX BY INT;

    meeting         meeting_type;

BEGIN  -- 也可以在这里赋值

    salary := (worked_time * hourly_salary) + bonus;

    SELECT TITLE INTO position FROM RESOURCES.EMPLOYEE WHERE LOGINID='L3';

    province := 'ShangHai';

    province := UPPER('wuhan');

    done := (counter > 100);

    valid_id := TRUE;

    emp_rec1.employeeid := 1;

    emp_rec1.managerid := null;

    emp_rec1 := emp_rec2;
```



```
meeting(5) := 20000 * 0.15;

PRINT position||'来自'||province;

PRINT ('加班工资'||salary);

END;

/
```

变量只在定义它的语句块(包括其下层的语句块)内可见,并且定义在下一层语句块中的变量可以屏蔽上一层的同名变量。当遇到一个变量名时,系统首先在当前语句块内查找变量的定义;如果没有找到,再向包含该语句块的上一层语句块中查找,如此直到最外层。如下例:

```
DECLARE

    a INT :=5;

BEGIN

    DECLARE

        a VARCHAR(10);          /* 此处定义的变量 a 与上一层中的变量 a 同名 */

    BEGIN

        a:= 'ABCDEFGF';

        PRINT a;                /* 第一条打印语句 */

    END;

    PRINT a;                    /* 第二条打印语句 */

END;

/
```

先定义了一个整型变量 a,然后又在其下层的语句块中定义了一个同名的字符型变量 a。由于在该语句块中,字符型变量 a 屏蔽了整型变量 a,所以第一条打印语句打印的是字符型变量 a 的值,而第二条打印语句打印的则是整型变量 a 的值。执行结果如下:

```
ABCDEFGF
5
```

3.6 使用 OR REPLACE 选项

您可能已经发现，在前面的例子中，我们在创建存储模块的时候，都使用了 OR REPLACE 选项。使用 OR REPLACE 选项的好处是，如果系统中已经有同名的存储模块，服务器会删除原先的存储模块，再创建新的存储模块。如果不使用 OR REPLACE 选项，当创建的存储模块与系统中已有的存储模块同名时，服务器会报错。

当已存在同名的存储模块时，能成功使用 OR REPLACE 选项的前提条件是当前用户具有删除原存储过程的权限，如果没有删除权限，则创建失败。使用 OR REPLACE 选项重新定义存储过程后，由于不能保证原有对象权限的合法性，原对象权限全部去除。

3.7 调用权限子句

调用权限子句用于解析存储模块中的 SQL 语句中没有指定所在模式的对象名，是否在当前模式下运行。DM8 提供两种策略：定义者权限和调用者权限，系统默认使用定义者权限。

调用权限子句的语法如下：

```
AUTHID CURRENT_USER      -- sql 语句在当前模式下执行
```

或

```
AUTHID DEFINER           -- sql 语句在过程或函数所在的模式下执行
```

使用说明：

AUTHID CURRENT_USER：采用调用者权限，即 SQL 语句在当前模式下执行

AUTHID DEFINER：采用定义者权限，即 SQL 语句在过程或函数所在模式下执行

3.8 调用、重新编译与删除存储模块

3.8.1 调用存储模块

对存储过程的调用可通过 CALL 语句来完成，也可以什么也不加直接通过名字及相应的参数执行即可，两种方式没有区别。

对于存储函数，除了可以通过 CALL 语句和直接通过名字调用外，还可以通过 SELECT 语句来调用，且执行方式存在一些区别：

- 通过 CALL 和直接使用名字调用存储函数时，不会返回函数的返回值，仅执行其中的操作；
- 通过 SELECT 语句调用存储函数时，不仅会执行其中的操作，还会返回函数的返回值。SELECT 调用的存储函数不支持含有 OUT、IN OUT 模式的参数。

如下面的例子：

```
CREATE OR REPLACE FUNCTION proc(A INT) RETURN INT AS
DECLARE
    s INT;
    lv INT;
    rv INT;
BEGIN
    IF A = 1 THEN
        s = 1;
    ELSIF A = 2 THEN
        s = 1;
    ELSE
        rv = proc(A - 1);
        lv = proc(A - 2);
        s = lv + rv;
        print lv || '+' || rv || '=' || s;
    END IF;
    RETURN S;
END;
/
```

通过 CALL 来调用函数 proc：

```
CALL proc(3);
```

执行结果为：

```
1+1=2
```

使用 SELECT 来调用这个函数 proc：

```
SELECT proc(3);
```

则显示结果为：

行号	PROC (3)
1	2

可见，使用 CALL 调用存储函数会执行其中打印操作，但不返回结果集；而如果用 SELECT 调用的话就会输出返回值 55，这个相当于是结果集。

3.8.2 重新编译存储模块

存储模块中常常会访问或修改一些数据库表、索引等对象，而这些对象有可能已被修改甚至删除，这意味着对应的存储模块已经失效了。

若用户想确认一个存储模块是否还有效，可以重新编译该存储模块。

重新编译存储模块的语法如下：

```
ALTER PROCEDURE|FUNCTION <存储模块名定义> COMPILE [DEBUG];
```

```
<存储模块名定义> ::= [ <模式名>.]<存储模块名>
```

语法中的“DEBUG”没有实际作用，仅语法支持。



说明：

系统存储模块不能进行重新编译。

例如，下面的语句对存储过程 RESOURCES.person_account 进行重新编译。

```
ALTER PROCEDURE RESOURCES.person_account COMPILE;
```

3.8.3 删除存储模块

当用户需要从数据库中删除一个存储模块时，可以使用存储模块删除语句。其语法如下：

```
DROP PROCEDURE <存储过程名定义>;
```

```
<存储过程名定义> ::= [ <模式名>.]<存储过程名>
```

或

```
DROP FUNCTION <存储函数名定义>;
```

```
<存储函数名定义> ::= [ <模式名>.]<存储函数名>
```

当模式名缺省时，默认为删除当前模式下的存储模块，否则，应指明存储模块所属的模式。除了 DBA 用户外，其他用户只能删除自己创建的存储模块。

例如，下面的语句删除之前创建的存储过程 `RESOURCES.proc_1` 和存储函数

```
RESOURCES.fun_1;
```

```
DROP PROCEDURE RESOURCES.proc_1;
```

```
DROP FUNCTION RESOURCES.fun_1;
```

4 DMSQL 程序中的各种控制结构

DMSQL 程序提供了丰富的控制结构，包括分支结构、循环控制结构、顺序结构等。通过控制结构，可以编写更复杂的 DMSQL 程序。

4.1 语句块

语句块是 DMSQL 程序的基本单元。每个语句块由关键字 DECLARE、BEGIN、EXCEPTION 和 END 划分为声明部分、执行部分和异常处理部分。其中执行部分是必须的，说明和异常处理部分可以省略。您可能已经发现，事实上存储模块的模块体就是一个语句块。语句块可以嵌套，它可以出现在任何其他语句可以出现的位置。

语句块的语法如下：

```
[DECLARE <变量说明>{,<变量说明>} ;]

BEGIN

<执行部分>

[<异常处理部分>]

END
```

■ 声明部分

声明部分包含了变量和常量的数据类型和初始值。这个部分由关键字 DECLARE 开始。如果不需要声明变量或常量，那么可以忽略这一部分。游标的声明也放在这一部分。

■ 执行部分

执行部分是语句块中的指令部分，由关键字 BEGIN 开始，以关键字 EXCEPTION 结束，如果 EXCEPTION 不存在，那么将以关键字 END 结束。所有的可执行语句都放在这一部分，其他的语句块也可以放在这一部分。分号分隔每一条语句，使用赋值操作符:=或 SELECT INTO 或 FETCH INTO 给变量赋值，执行部分的错误将在异常处理部分解决，在执行部分中可以使用另一个语句块，这种程序块被称为嵌套块。

所有的 SQL 数据操作语句都可以用于执行部分；执行部分使用的变量和常量必须首先在声明部分声明；执行部分必须至少包括一条可执行语句，NULL 是一条合法的可执行语句；事务控制语句 COMMIT 和 ROLLBACK 可以在执行部分使用。数据定义语言 (Data

Definition Language) 不能在执行部分中使用, DDL 语句与 EXECUTE IMMEDIATE 一起使用。

■ 异常处理部分

异常处理部分是可选的, 在这一部分中处理异常或错误, 对异常处理的详细讨论我们在后面进行。

需要强调的一点是, 一个语句块意味着一个作用域范围。也就是说, 在一个语句块的声明部分定义的任何对象, 其作用域就是该语句块。请看下面的例子, 该例中有一个全局变量 x, 同时子语句块中又定义了一个局部变量 x。

```
CREATE OR REPLACE PROCEDURE PROC_BLOCK AS

    X INT := 0;

    COUNTER INT := 0;

BEGIN

    FOR I IN 1 .. 4 LOOP

        X := X + 1000;

        COUNTER := COUNTER + 1;

        PRINT CAST(X AS CHAR(10)) || CAST(COUNTER AS CHAR(10)) || 'OUTER LOOP';

/* 这里是一个嵌套的子语句块的开始 */

DECLARE

    X INT := 0;  -- 局部变量 x, 与全局变量 x 同名

BEGIN

FOR I IN 1 .. 4 LOOP

    X := X + 1;

    COUNTER := COUNTER + 1;

    PRINT CAST(X AS CHAR(10)) || CAST(COUNTER AS CHAR(10)) || 'INNER LOOP';

END LOOP;

END;

/* 子语句块结束 */

    END LOOP;

END;
```

执行这个存储过程:

```
CALL PROC_BLOCK;
```

执行结果为:

1000	1	OUTER LOOP
1	2	INNER LOOP
2	3	INNER LOOP
3	4	INNER LOOP
4	5	INNER LOOP
2000	6	OUTER LOOP
1	7	INNER LOOP
2	8	INNER LOOP
3	9	INNER LOOP
4	10	INNER LOOP
3000	11	OUTER LOOP
1	12	INNER LOOP
2	13	INNER LOOP
3	14	INNER LOOP
4	15	INNER LOOP
4000	16	OUTER LOOP
1	17	INNER LOOP
2	18	INNER LOOP
3	19	INNER LOOP
4	20	INNER LOOP

由执行结果可以看出，两个 x 的作用域是完全不同的。

4.2 分支结构

分支结构先执行一个判断条件，根据判断条件的执行结果执行对应的一系列语句。

4.2.1 IF 语句

IF 语句控制执行基于布尔条件的语句序列，以实现条件分支控制结构。

语法如下：

```
IF <条件表达式>
THEN <执行部分>;

[<ELSEIF_OR_ELSEIF><条件表达式> THEN <执行部分>;{<ELSEIF_OR_ELSEIF><条件表达式> THEN
<执行部分>;}]

[ELSE <执行部分>;]

END IF;

<ELSEIF_OR_ELSEIF> ::= ELSEIF | ELSEIF
```

考虑到不同用户的编程习惯，ELSEIF 子句的起始关键字既可写作 ELSEIF，也可写作 ELSEIF。

条件表达式中的因子可以是布尔类型的参数、变量，也可以是条件谓词。存储模块的控制语句中支持的条件谓词有：比较谓词、BETWEEN、IN、LIKE 和 IS NULL。

最简单的 IF 语句形如：

```
IF 条件 THEN
    代码
END IF;
```

如果条件成立，则执行代码，否则什么也不执行。

如果需要在条件不成立时执行另外的代码，则应使用 IF-ELSE，形如：

```
IF 条件 THEN
    代码 1
ELSE
    代码 2
END IF;
```

此时，若条件成立则执行代码 1，条件不成立则执行代码 2。

例如，下面例子的子过程 proc_if 中使用 IF 语句判断销售指标完成情况，从而计算 bonus 值。

```
DECLARE

PROCEDURE proc_if (
```

```

        sales  dec,

        quota  dec,

        emp_id dec
    )

IS

        bonus  dec:= 0;

BEGIN

        IF sales > (quota + 400) THEN

                bonus := (sales - quota)/4;

        ELSE

                bonus := 100;

        END IF;

        DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

        UPDATE RESOURCES.EMPLOYEE SET SALARY = SALARY + bonus      WHERE
EMPLOYEEID = emp_id;

        END proc_if ;

BEGIN

        proc_if (30100, 20000, 1);

        proc_if (15000, 10000, 2);

END;

/

```

执行结果为:

```

bonus = 2525

bonus = 1250

```

还可以通过 ELSEIF 或 ELSIF 进行 IF 语句的嵌套，形如：

```

IF 条件 1 THEN

    代码 1

ELSEIF 条件 2 THEN

    代码 2

```

```
...  
  
ELSE  
  
    代码 N  
  
END IF
```

在执行上面的 IF 语句时，首先判断条件 1，当条件 1 成立时执行代码 1，否则继续判断条件 2，条件成立则执行代码 2，否则继续判断下面的条件。如果前面的条件都不成立，则执行 ELSE 后面的代码 N。

例如，在前面的例子中再增加一种计算 bonus 的情况：

```
DECLARE  
  
    PROCEDURE proc_if (  
  
        sales  dec,  
  
        quota  dec,  
  
        emp_id dec  
  
    )  
  
    IS  
  
        bonus  dec:= 0;  
  
    BEGIN  
  
        IF sales > (quota + 400) THEN  
  
            bonus := (sales - quota)/4;  
  
        ELSE IF sales > quota THEN  
  
            bonus := 100;  
  
        ELSE  
  
            bonus :=0;  
  
        END IF;  
  
    END IF;  
  
    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);  
  
    UPDATE RESOURCES.EMPLOYEE SET SALARY = SALARY + bonus      WHERE  
EMPLOYEEID = emp_id;  
  
    END proc_if ;  
  
BEGIN
```

```
proc_if (30100, 20000, 1);  
  
proc_if (15000, 10000, 2);  
  
proc_if (9000, 10000, 3);  
  
END;  
  
/
```

执行结果为:

```
bonus = 2525  
  
bonus = 1250  
  
bonus = 0
```

4.2.2 CASE 语句

CASE 语句从系列条件中进行选择, 并且执行相应的语句块, 主要有下面两种形式:

■ 简单形式: 将一个表达式与多个值进行比较

语法如下:

```
CASE <条件表达式>  
  
WHEN <条件> THEN <执行部分>;  
  
{WHEN <条件> THEN <执行部分>;}  
  
[ ELSE <执行部分> ]  
  
END [CASE];
```

语句中的每个条件可以是立即值, 也可以是一个表达式。这种形式的 CASE 语句会选择第一个满足条件的对应的执行部分来执行, 剩下的则不会计算, 如果没有符合的条件, 它会执行 ELSE 子句中的执行部分, 但是如果 ELSE 子句不存在, 则不会执行任何语句。

例如:

```
CREATE OR REPLACE PROCEDURE PROC_CASE (GRADE CHAR(10)) AS  
  
DECLARE  
  
    appraisal VARCHAR2(20);  
  
BEGIN  
  
    appraisal :=  
  
    CASE GRADE
```

```
        WHEN NULL THEN 'IS NULL'

        WHEN 'A' THEN 'Excellent'

        WHEN 'B' THEN 'Good'

        WHEN 'C' THEN 'Fair'

        ELSE 'No such grade'

    END;

    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);

END;

/
```

■ 搜索形式：对多个条件进行计算，取第一个结果为真的条件

语法如下：

```
CASE

WHEN <条件表达式> THEN <执行部分>;

{ WHEN <条件表达式> THEN <执行部分>; }

[ ELSE <执行部分> ]

END [CASE];
```

搜索模式的 CASE 语句依次执行各条件表达式，当遇见第一个为真的条件时，执行其对应的执行部分，在第一个为真的条件后面的所有条件都不会再执行。如果所有的条件都不为真，则执行 ELSE 子句中的执行部分，如果 ELSE 子句不存在，则不执行任何语句。

例如：

```
CREATE OR REPLACE PROCEDURE PROC_CASE (GRADE CHAR(10)) AS

DECLARE

    appraisal VARCHAR2(20);

BEGIN

    appraisal :=

    CASE

        WHEN grade IS NULL THEN 'IS NULL'

        WHEN grade = 'A' THEN 'Excellent'

        WHEN grade = 'B' THEN 'Good'
```

```
        WHEN grade = 'C' THEN 'Fair'

        ELSE 'No such grade'

    END;

    DBMS_OUTPUT.PUT_LINE ('Grade ' ||grade|| ' is ' ||appraisal);

END;

/
```

4.2.3 SWITCH 语句

DM8SQL 程序支持 C 语法风格的 SWITCH 分支结构语句，关于 C 语法 DM8SQL 程序介绍请看 7.1 节。

SWITCH 语句的功能与简单形式的 CASE 语句类似，用于将一个表达式与多个值进行比较，并执行相应的语句块。

语法如下：

```
SWITCH (<条件表达式>)

{

CASE <常量表达式> : <执行部分>; BREAK;

{ CASE <常量表达式> : <执行部分>; BREAK; }

[DEFAULT : <执行部分>; ]

}
```

每个 CASE 分支的执行部分后应有“BREAK”语句，否则 SWITCH 语句在执行了对应分支的执行部分后会继续执行后续分支的执行部分。

如果没有符合的 CASE 分支，会执行 DEFAULT 子句中的执行部分，如果 DEFAULT 子句不存在，则不会执行任何语句。

例如：

```
{

    VARCHAR appraisal='B';

    SWITCH (appraisal)

    {

        CASE NULL: PRINT 'IS NULL'; BREAK;
```

```

        CASE 'A': PRINT 'Excellent'; BREAK;

        CASE 'B': PRINT 'Good'; BREAK;

        CASE 'C': PRINT 'Fair'; BREAK;

        DEFAULT: PRINT 'No such grade';

    }
}
/

```

4.3 循环控制结构

DM8SQL 程序支持五种类型的循环语句：LOOP 语句、WHILE 语句、FOR 语句、REPEAT 语句和 FORALL 语句。其中前四种为基本类型的循环语句：LOOP 语句循环重复执行一系列语句，直到 EXIT 语句终止循环为止；WHILE 语句循环检测一个条件表达式，当表达式的值为 TRUE 时就执行循环体的语句序列；FOR 语句对一系列的语句重复执行指定次数的循环；REPEAT 语句重复执行一系列语句直至达到条件表达式的限制要求。FORALL 语句对一条 DML 语句执行多次，当 DML 语句中使用数组或嵌套表时可进行优化处理，能大幅提升性能。

4.3.1 LOOP 语句

LOOP 语句的语法如下：

```

LOOP

<执行部分>;

END LOOP [标号名];

```

LOOP 语句实现对一语句系列的重复执行，是循环语句的最简单形式。LOOP 和 END LOOP 之间的执行部分将无限次地执行，必须借助 EXIT、GOTO 或 RAISE 语句来跳出循环。

例如，下面的例子在 LOOP 循环中打印参数 A 的值，并将 A 的值减 1，直到 A 小于等于 0 时跳出循环。

```

CREATE OR REPLACE PROCEDURE PROC_LOOP(a IN OUT INT) AS
BEGIN

    LOOP

```

```
        IF a<=0 THEN

            EXIT;

        END IF;

        PRINT a;

        a:=a-1;

    END LOOP;

END;

/

CALL PROC_LOOP(5);
```

4.3.2 WHILE 语句

WHILE 语句的语法如下：

```
WHILE <条件表达式> LOOP

<执行部分>;

END LOOP [标号名];
```

WHILE 循环语句在每次循环开始之前，先计算条件表达式，若该条件为 TRUE，执行部分被执行一次，然后控制重新回到循环顶部。若条件表达式的值为 FALSE，则结束循环。当然，也可以通过 EXIT 语句来终止循环。

例如，使用 WHILE 语句实现上一小节中 LOOP 循环相同的功能。

```
CREATE OR REPLACE PROCEDURE PROC_WHILE(a IN OUT INT) AS

BEGIN

    WHILE a>0 LOOP

        PRINT a;

        a:=a-1;

    END LOOP;

END;

/

CALL PROC_WHILE(5);
```


4.3.3 FOR 语句

FOR 语句的语法如下：

```
FOR <循环计数器> IN [REVERSE] <下限表达式> .. <上限表达式> LOOP  
  
<执行部分>;  
  
END LOOP [标号名];
```

循环计数器是一个标识符，它类似于一个变量，但是不能被赋值，且作用域限于 FOR 语句内部。下限表达式和上限表达式用来确定循环的范围，它们的类型必须和整型兼容。循环次数是在循环开始之前确定的，即使在循环过程中下限表达式或上限表达式的值发生了改变，也不会引起循环次数的变化。

执行 FOR 语句时，首先检查下限表达式的值是否小于上限表达式的值，如果下限数值大于上限数值，则不执行循环体。否则，将下限数值赋给循环计数器 (语句中使用了 REVERSE 关键字时，则把上限数值赋给循环计数器)；然后执行循环体内的语句序列；执行完后，循环计数器值加 1 (如果有 REVERSE 关键字，则减 1)；检查循环计数器的值，若仍在循环范围内，则重新继续执行循环体；如此循环，直到循环计数器的值超出循环范围。同样，也可以通过 EXIT 语句来终止循环。

例如，使用 FOR 语句实现前面 LOOP 语句和 WHILE 语句例子相同的功能。

```
CREATE OR REPLACE PROCEDURE PROC_FOR1 (a IN OUT INT) AS  
  
BEGIN  
  
    FOR I IN REVERSE 1 .. a LOOP  
  
        PRINT I;  
  
        a:=I-1;  
  
    END LOOP;  
  
END;  
  
/  
  
CALL PROC_FOR1(5);
```

FOR 语句中的循环计数器可与当前语句块内的参数或变量同名，这时该同名的参数或变量在 FOR 语句的范围内将被屏蔽。如下例所示：

```
DECLARE
```

```
v1 DATE:=DATE '2017-03-17';  
  
BEGIN  
  
    FOR v1 IN 3.. 5 LOOP  
  
        PRINT v1;  
  
    END LOOP;  
  
    PRINT v1;  
  
END;  
  
/
```

打印结果为:

```
3  
  
4  
  
5  
  
2017-03-17
```

4.3.4 REPEAT 语句

REPEAT 语句的语法如下:

```
REPEAT  
  
<执行部分>;  
  
UNTIL <条件表达式>;
```

REPEAT 语句先执行执行部分，然后判断条件表达式，若为 TRUE 则控制重新回到循环顶部，若为 FALSE 则退出循环。可以看出，REPEAT 语句的执行部分至少会执行一次。

例如:

```
DECLARE  
  
    a INT;  
  
BEGIN  
  
    a := 0;  
  
    REPEAT  
  
        a := a+1;  
  
        PRINT a;
```

```

    UNTIL a>10;

END;

/

```

4.3.5 FORALL 语句

FORALL 语句的语法如下：

```

FORALL<循环计数器> IN <bounds_clause> [SAVE EXCEPTIONS] <forall_dml_stmt>;

<bounds_clause> ::= <下限表达式>..<上限表达式>

                    | INDICES OF <集合> [BETWEEN ] <下限表达式> AND <上限表达式>

                    | VALUES OF <集合>

<forall_dml_stmt> ::= <INSERT 语句> | <UPDATE 语句> | <DELETE 语句> | <MERGE INTO
语句>

```

SAVE EXCEPTIONS：指定即使一些 DML 语句失败，直到 FORALL 执行结束才抛出异常。

INDICES OF <集合>：跳过集合中没有赋值的元素，可用于指向稀疏数组的实际下标。

VALUES OF <集合>：把该集合中的值作为下标。

下面的例子说明了如何使用 FORALL 语句：

```

CREATE TABLE t1_forall(C1 INT, C2 VARCHAR(50));

CREATE OR REPLACE PROCEDURE p1_forall AS
BEGIN
    FORALL I IN 1..10
        INSERT INTO t1_forall SELECT TOP 1 EMPLOYEEID, TITLE FROM
RESOURCES.EMPLOYEE;
END;

/

```

DM 可对 FORALL 中的 INSERT、UPDATE 和 DELETE 语句中的数组或嵌套表引用进行优化处理。需要注意的是，优化处理会影响游标的属性值，导致其不可使用。dm.ini 中的

USE_FORALL_ATTR 参数控制是否进行优化处理：值为 0 表示可以优化，不使用游标属性；值为 1 表示不优化，使用游标属性，默认值为 0。

4.3.6 EXIT 语句

EXIT 语句与循环语句一起使用，用于终止其所在循环语句的执行，将控制转移到该循环语句外的下一个语句继续执行。

EXIT 语句的语法如下：

```
EXIT [<标号名>] [WHEN <条件表达式>];
```

EXIT 语句必须出现在一个循环语句中，否则将报错。

当 EXIT 后面的标号名省略时，该语句将终止直接包含它的那条循环语句；当 EXIT 后面带有标号名时，该语句用于终止标号名所标识的那条循环语句。需要注意的是，该标号名所标识的语句必须是循环语句，并且 EXIT 语句必须出现在此循环语句中。当 EXIT 语句位于多重循环中时，可以用该功能来终止其中的任何一重循环。

下面的例子中 EXIT 后没有带标号，其所在的那层循环被终止。

```
DECLARE

    a INT;

    b INT;

BEGIN

    a := 0;

    LOOP

        FOR b in 1 .. 2 LOOP

            PRINT '内层循环' || b;

            EXIT WHEN a > 3;

        END LOOP;

        a := a + 2;

        PRINT '---外层循环' || a;

        EXIT WHEN a > 5;

    END LOOP;

END;
```

/

执行结果为：

内层循环 1

内层循环 2

---外层循环 2

内层循环 1

内层循环 2

---外层循环 4

内层循环 1

---外层循环 6

下面的例子中 EXIT 后带有标号，则其终止了指定的那层循环。

```
DECLARE
```

```
    a INT;
```

```
    b INT;
```

```
BEGIN
```

```
    a := 0;
```

```
    <<flag1>>
```

```
    LOOP
```

```
        FOR b in 1 .. 2 LOOP
```

```
            PRINT '内层循环' || b;
```

```
            EXIT flag1 WHEN a > 3;
```

```
        END LOOP;
```

```
        a := a + 2;
```

```
        PRINT '---外层循环' || a;
```

```
        EXIT WHEN a > 5;
```

```
    END LOOP;
```

```
END;
```

/

执行结果为：

```
内层循环 1
内层循环 2
---外层循环 2
内层循环 1
内层循环 2
---外层循环 4
内层循环 1
```

当 WHEN 子句省略时，EXIT 语句无条件地终止该循环语句；否则，先计算 WHEN 子句中的条件表达式，当条件表达式的值为真时，终止该循环语句。如下例所示：

```
DECLARE

    a INT;

    b INT;

BEGIN

    a := 0;

    LOOP

        FOR b in 1 .. 2 LOOP

            PRINT '内层循环' || b;

            EXIT;

        END LOOP;

        a := a + 2;

        PRINT '---外层循环' || a;

        EXIT;

    END LOOP;

END;

/
```

执行结果为：

```
内层循环 1
---外层循环 2
```

4.3.7 CONTINUE 语句

CONTINUE 语句的作用是退出当前循环，并且将语句控制转移到这次循环的下一循环迭代或者是一个指定标签的循环的开始位置并继续执行。

CONTINUE 语句的语法为：

```
CONTINUE [[标号名] WHEN <条件表达式>];
```

若 CONTINUE 后没有跟 WHEN 子句，则无条件立即退出当前循环，并且将语句控制转移到这次循环的下一循环迭代或者是一个指定标号名的循环的开始位置并继续执行。如下例所示：

```
DECLARE

    x INT:= 0;

BEGIN

    <<flag1>> -- CONTINUE 跳出之后，回到这里

    FOR I IN 1..4 LOOP

        DBMS_OUTPUT.PUT_LINE ('循环内部，CONTINUE 之前：  x = ' || TO_CHAR(x));

        x := x + 1;

        CONTINUE flag1;

        DBMS_OUTPUT.PUT_LINE ('循环内部，CONTINUE 之后：  x = ' || TO_CHAR(x));

    END LOOP;

    DBMS_OUTPUT.PUT_LINE (' 循环外部：  x = ' || TO_CHAR(x));

END;

/
```

执行结果为：

```
循环内部，CONTINUE 之前：  x = 0
循环内部，CONTINUE 之前：  x = 1
循环内部，CONTINUE 之前：  x = 2
循环内部，CONTINUE 之前：  x = 3
循环外部：  x = 4
```

若 CONTINUE 语句中包含 WHEN 子句，则当 WHEN 子句的条件表达式为 TRUE 时才退出当前循环，将语句控制转移到下一次循环迭代或者是一个指定标号名的循环的开始位置并继续执行。当每次循环到达 CONTINUE-WHEN 时，都会对 WHEN 的条件进行计算，如果条件为 FALSE，则 CONTINUE-WHEN 语句不执行任何动作，为了防止出现死循环，应将 WHEN 条件设置为一个肯定可以为 TRUE 的表达式。

下面的例子说明了 CONTINUE-WHEN 语句的使用。

```
DECLARE

    x INT:= 0;

BEGIN

    -- CONTINUE 跳出之后，回到这里

    FOR I IN 1..4 LOOP

        DBMS_OUTPUT.PUT_LINE ('循环内部，CONTINUE 之前： x = ' || TO_CHAR(x));

        x := x + 1;

        CONTINUE WHEN x > 3;

        DBMS_OUTPUT.PUT_LINE ('循环内部，CONTINUE 之后： x = ' || TO_CHAR(x));

    END LOOP;

    DBMS_OUTPUT.PUT_LINE (' 循环外部： x = ' || TO_CHAR(x));

END;

/
```

执行结果为：

```
循环内部，CONTINUE 之前： x = 0
循环内部，CONTINUE 之后： x = 1
循环内部，CONTINUE 之前： x = 1
循环内部，CONTINUE 之后： x = 2
循环内部，CONTINUE 之前： x = 2
循环内部，CONTINUE 之后： x = 3
循环内部，CONTINUE 之前： x = 3
循环外部： x = 4
```


4.4 顺序结构

4.4.1 GOTO 语句

GOTO 语句无条件地跳转到一个标号所在的位置，其语法为：

```
GOTO <标号名>;
```

GOTO 语句将控制权交给带有标号的语句或语句块。标号的定义在一个语句块中必须是唯一的，且必须指向一条可执行语句或语句块。

为了保证 GOTO 语句的使用不会引起程序的混乱，我们对 GOTO 语句的使用有下列限制：

- GOTO 语句不能跳入一个 IF 语句、CASE 语句、循环语句或下层语句块中；
- GOTO 语句不能从一个异常处理器跳回当前块，但是可以跳转到包含当前块的上层语句块。

下面的例子说明了如何正确使用 GOTO 语句。

```
DECLARE

    v_name    VARCHAR2(25);

    v_empid   INT := 1;

BEGIN

    <<get_name>>

    SELECT NAME INTO v_name FROM RESOURCES.EMPLOYEE A, PERSON.PERSON B WHERE
A.PERSONID=B.PERSONID AND A.EMPLOYEEID=v_empid;

    BEGIN

        DBMS_OUTPUT.PUT_LINE(v_name);

        v_empid := v_empid + 1;

        IF v_empid <=5 THEN

            GOTO get_name;

        END IF;

    END;

END;
```

下面两个例子则是非法使用 GOTO 语句的示例。

例 1：

```
BEGIN

...

GOTO update_row;      /* 错误, 企图跳入一个 IF 语句 */

...

IF valid THEN

...

<<update_row>>

UPDATE emp SET ...

END IF;

END;

/
```

例 2:

```
BEGIN

...

IF status = 'OBSOLETE' THEN

    GOTO delete_part; /* 错误, 企图跳入一个下层语句块 */

END IF;

...

BEGIN

...

<<delete_part>>

DELETE FROM parts WHERE ...

END;

END;

/
```

4.4.2 NULL 语句

NULL 语句的语法为:

```
NULL;
```

NULL 语句不做任何事情，只是用于保证语法的正确性，或增加程序的可读性。

例如，下面例子中的 NULL 语句说明程序已经考虑了其他的异常，只是并不做处理。

```
CREATE OR REPLACE FUNCTION func_null (  
    a INT,  
    b INT  
) RETURN INT  
AS  
BEGIN  
    RETURN (a/b);  
EXCEPTION  
    WHEN ZERO_DIVIDE THEN  
        DBMS_OUTPUT.PUT_LINE('除 0 错误');  
    WHEN OTHERS THEN  
        NULL;  
END;  
/
```

4.5 其他语句

4.5.1 赋值语句

赋值语句的语法为：

<赋值对象> := <值表达式>;

或

SET <赋值对象> = <值表达式>;

使用赋值语句可以给各种数据类型的对象赋值。被赋值的对象可以是变量，也可以是 OUT 参数或 IN OUT 参数。表达式的数据类型必须与赋值对象的数据类型兼容。

关于赋值语句的使用示例可以参考 3.5 节中的例子。



说明：

如果赋值对象和值表达式是类类型，赋值采用的是对象指向逻辑，赋值对象指向值表达式的对象，而不会创建新的对象。

4.5.2 调用语句

存储模块可以被别的存储模块或应用程序调用。同样，在存储模块中也可以调用其他存储模块。调用存储模块时，应给存储模块提供输入参数值，并获取存储模块的输出参数值。

调用语句的语法为：

```
[CALL] [<模式名>.]<存储模块名>[@dblink_name] [((<参数>{, <参数>}));
```

```
<参数> ::= <参数值>|<参数名=参数值>
```

使用说明：

- 如果被调用的存储模块不属于当前模式，必须在语句中指明存储模块的模式名；
- 参数的个数和类型必须与被调用的存储模块一致；
- 存储模块的输入参数可以是嵌入式变量，也可以是值表达式；存储模块的输出参数必须是可赋值对象，如嵌入式变量；
- “dblink_name”表示创建的 dblink 名字，如果添加了该选项，则表示调用远程实例的存储模块；
- 在调用过程中，服务器将以存储模块创建者的模式和权限来执行过程中的语句。

一般情况下，用户调用存储模块时通过实际参数位置关系和形式参数相对应，这种方式被称为“位置标识法”。系统还支持另一种存储模块调用方式：每个参数中包含形式参数和实际参数，这样的方法被称为“名字标识法”。对结果而言，这两种调用方式是等价的，但在具体使用时存在一些差异，如表 4.1 所示。

表 4.1 位置标识法与名字标识法的使用比较

位置标识法	名字标识法
依赖于实际参数的名称	清楚地说明了实际参数和形式参数间的对应关系
给出的实际参数必须依照形式参数的次序	指定实际参数时可不按照形式参数的顺序
调用比较简洁，符合大多数第三代语言的使用习惯	需要更多的编码，调用时需要指定形式参数和实际参数
使用参数缺省值时必须在参数列表的末尾	无论哪个参数拥有缺省值都不会对调用产生影响
维护的代价在于参数的定义次序发生了变化	维护的代价在于参数名的定义发生了变化

下面的例子分别使用位置标识法和名字标识法调用存储模块。

```
CREATE OR REPLACE PROCEDURE proc_call (a INT, b IN OUT INT) AS

    v1 INT:=a;

BEGIN
```

```
b:=0;

FOR C IN 1 .. v1 LOOP

    b:=b+c;

END LOOP;

print b;

END;

/

CALL proc_call(10,0);  --以位置标识法调用

CALL proc_call(b=0,a=10);  --以名字标识法调用
```

下面的例子演示了如何通过 DBLINK 调用远程实例的存储模块。

一，假设远程数据库（实例名 dmserver2，端口号 5237）已创建 DBLINK，名为 TEST_LINK。该远程数据库上存在存储过程 dm_get_next_val。

```
CREATE OR REPLACE PROCEDURE dm_get_next_val(a IN OUT INT) AS
BEGIN
    a := a + 1;
END;

/
```

二，在本地（实例名 dmserver，端口号 5236）通过 DBLINK 名字 TEST_LINK，调用上述远程过程：

```
DECLARE

    x INT;

BEGIN

    x := 1;

    dm_get_next_val@TEST_LINK(x);

    PRINT x;

END;

/
```

4.5.3 RETURN 语句

RETURN 语句的语法为:

```
RETURN [<返回值>];
```

RETURN 语句结束 DMSQL 程序的执行，将控制返回给 DMSQL 程序的调用者。如果是从存储函数返回，则同时将函数的返回值提供给调用环境。

除管道表函数外，函数的执行必须以 RETURN 语句结束。确切地说，函数中应至少有一个被执行的返回语句。由于程序中有分支结构，在编译时很难保证其每条路径都有返回语句，因此 DM 在函数执行时才对其进行检查，如果某个分支缺少 RETURN，DM 会自动为其隐式增加一条 RETURN NULL 语句。

例如:

```
CREATE OR REPLACE FUNCTION func_return(a INT) RETURN VARCHAR(10) AS
BEGIN
    IF (a<0) THEN
        RETURN ' a<0';
    ELSE
        PRINT '没有 RETURN';
    END IF;
END;
```

在这个例子中，当参数 a 大于或等于 0 时，函数 func_return 没有以 RETURN 语句结束。因此，执行下面的语句:

```
SELECT func_return(2);
```

执行结果为:

行号	FUNC_RETURN(2)

1	NULL

4.5.4 PRINT 语句

PRINT 语句的语法为:

```
PRINT <表达式>;
```

PRINT 语句用于从 DMSQL 程序中向客户端输出一个字符串，语句中的表达式可以是各种数据类型，系统自动将其转换为字符类型。

PRINT 语句便于用户调试 DMSQL 程序代码。当 DMSQL 程序的行为与预期不一致时，可以在其中加入 PRINT 语句来观察各个阶段的运行情况。

之前章节的示例中有很多已经使用了 PRINT 语句，这里不再另外举例。用户也可以使用 DM 系统包方法 DBMS_OUTPUT.PUT_LINE() 将信息打印到客户端。

4.5.5 PIPE ROW 语句

PIPE ROW 语句只能在管道表函数中使用，其语法为：

```
PIPE ROW ( <值表达式> );
```

管道表函数是可以返回行集合的函数，用户可以像查询数据库表一样查询它。目前 DM 管道表函数的返回值类型暂时只支持 VARRAY 类型和嵌套表类型。

PIPE ROW 语句将返回一行到管道表函数的结果行集中。如果值表达式是类类型的表达式，会复制一个对象输入到管道函数的结果集中，保证将同一个对象多次输入到管道函数的结果集中时，后文的修改不会影响前面的输入。

下面是一个关于管道表函数的例子：

```
CREATE TYPE mytype AS OBJECT (  
    COL1 INT,  
    COL2 VARCHAR (64)  
);  
  
/  
  
CREATE TYPE mytypelist AS TABLE OF mytype;  
  
/  
  
CREATE OR REPLACE FUNCTION func_piperow RETURN mytypelist PIPELINED  
IS  
    v_mytype mytype;  
  
BEGIN  
    FOR I IN 1 .. 5LOOP
```

```
        v_mytype := mytype (I, 'Row ' || I);

        PIPE ROW (v_mytype);

    END LOOP;

EXCEPTION

    WHEN OTHERS THEN NULL;

END;

/
```

查询管道表函数：

```
SELECT * FROM TABLE (FUNC_piperow);
```

查询结果为：

行号	COL1	COL2
1	1	Row 1
2	2	Row 2
3	3	Row 3
4	4	Row 4
5	5	Row 5

如果管道表函数中没有进行 PIPE ROW 操作，则管道表函数的返回结果为一个没有元素数据的集合，而不是 NULL，即返回的是空集而不是空。

5 DMSQL 程序中的 SQL 语句

使用 DMSQL 程序的主要目的是对 DM 数据库进行访问，因此 DMSQL 程序中支持使用 SQL 语句进行对数据库对象的 SELECT、INSERT、UPDATE、DELETE 等 DML 操作，还可以定义和操纵游标等。DMSQL 程序支持的 SQL 语句具体包括：

- ✓ 数据查询语句 (SELECT)
- ✓ 数据操纵语句 (INSERT、DELETE、UPDATE)
- ✓ 游标定义及操纵语句 (DECLARE、OPEN、FETCH、CLOSE)
- ✓ 事务控制语句 (COMMIT、ROLLBACK、SAVEPOINT)
- ✓ 动态 SQL 执行语句 (EXECUTE IMMEDIATE)

通过 SQL 语句及上一章介绍的各种控制结构，用户可以编写复杂的 DMSQL 程序，实现复杂逻辑的数据库访问应用。

5.1 普通静态 SQL 语句

5.1.1 数据操纵

在 DMSQL 程序中，可以直接使用 INSERT、DELETE 和 UPDATE 语句对数据库中的表进行增、删、改操作。

例如：

```
DECLARE

    category CONSTANT VARCHAR(50) := '天文';

BEGIN

    INSERT INTO PRODUCTION.PRODUCT_CATEGORY VALUES (category);

END;

/
```

上面的例子中在 INSERT 语句中直接使用了常量作为列值进行插入。在实际应用中，由于 DMSQL 程序的编程特性，更常见的使用方式是在 DML 语句中使用 DMSQL 程序中定义的变量，如下例所示。

```
CREATE OR REPLACE PROCEDURE proc_ins(category IN VARCHAR(50)) AS
BEGIN
    INSERT INTO PRODUCTION.PRODUCT_CATEGORY VALUES (category);
END;
/
CALL PROC_INS('动漫');
```

UPDATE 和 DELETE 语句的 WHERE 子句中也可以使用变量，例如：

```
DECLARE
    catogory1 VARCHAR(50);
    catogory2 VARCHAR(50);
BEGIN
    catogory1='动漫';
    catogory2='高等数学';
    DELETE PRODUCTION.PRODUCT_CATEGORY WHERE NAME=catogory1;
    UPDATE PRODUCTION.PRODUCT_CATEGORY SET NAME=catogory2 WHERE NAME='数学';
END;
/
```

5.1.2 数据查询

在 DMSQL 程序中可直接使用 SELECT 语句从数据库中查询数据。由于通常对数据进行查询的目的是为了进一步进行处理，而不仅仅是显示出来，所以 SELECT 语句可以采用下面的语法将查询到的数据赋给相应变量以便后面对它进行引用：

```
SELECT<列名>{,<列名>} INTO <变量>{,<变量>} FROM.....
```

与一般 SELECT 语句不同，上面语法所示的 SELECT 语句使用 INTO 子句将查询到的数据存放到变量中。查询的列与 INTO 子句中的变量在数目、类型上要一致，否则会报错。

下面的例子查询书名和出版社，存放到对应变量中并打印出来。

```
DECLARE
    p_name VARCHAR(50);
```

```
p_publish VARCHAR(50);

BEGIN

    SELECT NAME,PUBLISHER INTO p_name,p_publish FROM PRODUCTION.PRODUCT WHERE
AUTHOR LIKE '曹雪芹,高鹗';

    PRINT p_name;

    PRINT p_publish;

EXCEPTION

    WHENNO_DATA_FOUND OR TOO_MANY_ROWS THEN

        PRINT'NO_DATA_FOUND OR TOO_MANY_ROWS';

    WHEN OTHERS THEN

        PRINT 'ERROR OCCURS';

END;

/
```

SELECT...INTO 语句要求查询只能返回一条记录，执行时可能会发生两种例外情况：

- 没有查询到满足条件的记录，系统返回预定义异常 NO_DATA_FOUND；
- 存在多行满足条件的记录，系统返回预定义异常 TOO_MANY_ROWS，对于这样的异常必须做出相应处理，否则会影响 DMSQL 程序的正确执行。

5.1.3 事务控制

可以在 DMSQL 程序中直接使用 COMMIT、ROLLBACK 和 SAVEPOINT 语句进行事务控制：

- ✓ COMMIT[WORK]；语句提交一个事务
- ✓ ROLLBACK[WORK]；语句回滚一个事务
- ✓ SAVEPOINT <保存点名>；语句在事务中设置一个保存点
- ✓ ROLLBACK [WORK] TO SAVEPOINT <保存点名>；语句将事务回滚到指定的保存点

5.2 游标

5.1.2 节中介绍了 DMSQL 程序中使用 SELECT...INTO 语句将查询结果存放到变量中进行处理的方法，但这种方法只能返回一条记录，否则就会产生 TOO_MANY_ROWS 错误。为了解决这个问题，DMSQL 程序引入了游标，允许程序对多行数据进行逐条处理。

5.2.1 静态游标

静态游标是只读游标，它总是按照打开游标时的原样显示结果集，在编译时就能确定静态游标使用的查询。

静态游标又分为两种：隐式游标和显式游标。

5.2.1.1 隐式游标

隐式游标无需用户进行定义，每当用户在 DMSQL 程序中执行一个 DML 语句（INSERT、UPDATE、DELETE）或者 SELECT ...INTO 语句时，DMSQL 程序都会自动声明一个隐式游标并管理这个游标。

隐式游标的名称为“SQL”，用户可以通过隐式游标获取语句执行的一些信息。DMSQL 程序中的每个游标都有%FOUND、%NOTFOUND、%ISOPEN 和%ROWCOUNT 四个属性，对于隐式游标，这四个属性的意义如下：

- ✓ %FOUND：语句是否修改或查询到了记录，是返回 TRUE，否则返回 FALSE；
- ✓ %NOTFOUND：语句是否未能成功修改或查询到记录，是返回 TRUE，否则返回 FALSE；
- ✓ %ISOPEN：游标是否打开。是返回 TRUE，否返回 FALSE。由于系统在语句执行完成后会自动关闭隐式游标，因此隐式游标的%ISOPEN 属性永远为 FALSE；
- ✓ %ROWCOUNT：DML 语句执行影响的行数，或 SELECT...INTO 语句返回的行数。

例如，将孙丽的电话号码修改为 13818882888。

```
BEGIN

UPDATE PERSON.PERSON SET PHONE=13818882888 WHERE NAME='孙丽';

IF SQL%NOTFOUND THEN
```

```

        PRINT '此人不存在';

ELSE

        PRINT '已修改';

END IF;

END;

/

```

5.2.1.2 显式游标

显式游标指向一个查询语句执行后的结果集区域。当需要处理返回多条记录的查询时，应显式地定义游标以处理结果集的每一行。

使用显式游标一般包括四个步骤：

1. 定义游标：在 DMSQL 程序的声明部分定义游标，声明游标及其关联的查询语句；
2. 打开游标：执行游标关联的语句，将查询结果装入游标工作区，将游标定位到结果集的第一行之前；
3. 拨动游标：根据应用需要将游标位置移动到结果集的合适位置；
4. 关闭游标：游标使用完后应关闭，以释放其占有的资源。

下面对这四个步骤进行具体介绍。

■ 定义显示游标

在 DMSQL 程序的声明部分定义显示游标，其语法如下：

```
CURSOR <游标名> [FAST | NO FAST] <cursor 选项>;
```

或

```
<游标名> CURSOR [FAST | NO FAST] <cursor 选项>;
```

```
<cursor 选项> := <cursor 选项 1>|<cursor 选项 2>|<cursor 选项 3>|<cursor 选项 4>
```

```
<cursor 选项 1>:= <IS|FOR> {<查询表达式>|<连接表>}
```

```
<cursor 选项 2>:= <IS|FOR> TABLE <表名>
```

```
<cursor 选项 3>:= (<参数声明> {,<参数声明>}) IS <查询表达式>
```

```
<cursor 选项 4>:= [(<参数声明> {,<参数声明>})] RETURN <DMSQL 数据类型> IS <查询表达式>
```

```
<参数声明> ::= <参数名> [IN] <参数类型> [ DEFAULT|:= <缺省值> ]
```

<DMSQL 数据类型> ::= <普通数据类型>

| <变量名> %TYPE

| <表名> %ROWTYPE

| CURSOR

| REF <游标名>

语法中的“FAST”指定游标是否为快速游标。缺省为 NO FAST，为普通游标。快速游标提前返回结果集，速度上提升明显，但是存在以下的使用约束：

- FAST 属性只在显示游标中支持；
- 使用快速游标的 DMSQL 程序语句块中不能修改快速游标所涉及的表。这点需用户自己保证，否则可能导致结果不正确；
- 不支持游标更新和删除；
- 不支持 NEXT 以外的 FETCH 方向；
- 不支持快速游标作为函数返回值；
- MPP 环境下不支持对快速游标进行 FETCH 操作。

必须先定义一个游标，之后才能在别的语句中使用它。定义显示游标时指定游标名和与其关联的查询语句。可以指定游标的返回类型，也可以指定关联的查询语句中的 WHERE 子句使用的参数。

下面的程序片段说明了如何使用不同语法定义各种显示游标。

```
DECLARE

    CURSOR c1 IS SELECT TITLE FROM RESOURCES.EMPLOYEE WHERE MANAGERID = 3;

    CURSOR c2 RETURN RESOURCES.EMPLOYEE%ROWTYPE IS SELECT * FROM
RESOURCES.EMPLOYEE;

    c3 CURSOR IS TABLE RESOURCES.EMPLOYEE;

.....
```

■ 打开显示游标

打开一个显示游标的语法如下：

OPEN <游标名>;

指定打开的游标必须是已定义的游标，此时系统执行这个游标所关联的查询语句，获得结果集，并将游标定位到结果集的第一行之前。



注意:

当再次打开一个已打开的游标时，游标会被重新初始化，游标属性数据可能会发生变化。

■ 拨动游标

拨动游标的语法为:

```
FETCH [<fetch 选项> [FROM]] <游标名> [ [BULK COLLECT] INTO <主变量名>{,<主变量名>} ] [LIMIT <rows>];
```

```
<fetch 选项>::= NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n
```

被拨动的游标必须是已打开的游标。

fetch 选项指定将游标移动到结果集的某个位置:

- ✓ NEXT: 游标下移一行
- ✓ PRIOR: 游标前移一行
- ✓ FIRST: 游标移动到第一行
- ✓ LAST: 游标移动到最后一行
- ✓ ABSOLUTE n: 游标移动到第 n 行
- ✓ RELATIVE n: 游标移动到当前指示行后的第 n 行

FETCH 语句每次只获取一条记录，除非指定了“BULK COLLECT”。若不指定 FETCH 选项，则第一次执行 FETCH 语句时，游标下移，指向结果集的第一行，以后每执行一次 FETCH 语句，游标均顺序下移一行，使这一行成为当前行。

INTO 子句中的变量个数、类型必须与游标关联的查询语句中各 SELECT 项的个数、类型一一对应。典型的使用方式是在 LOOP 循环中使用 FETCH 语句将每一条记录数据赋给变量，并进行处理，使用 %FOUND 或 %NOTFOUND 来判断是否处理完数据并退出循环。如下例所示:

```
DECLARE

    v_name VARCHAR(50);

    v_phone VARCHAR(50);

    c1 CURSOR;

BEGIN
```

```
OPEN c1 FOR SELECT NAME,PHONE FROM PERSON.PERSON A,RESOURCES.EMPLOYEE B
WHERE A.PERSONID=B.PERSONID;

LOOP

    FETCH c1 INTO v_name,v_phone;

    EXIT WHEN c1%NOTFOUND;

    PRINT v_name || v_phone;

END LOOP;

CLOSE c1;

END;

/
```

使用 FETCH...BULK COLLECT INTO 可以将查询结果批量地、一次性地赋给集合变量。FETCH...BULK COLLECT INTO 和 LIMIT rows 配合使用，可以限制每次获取数据的行数。

下面的例子说明了 FETCH...BULK COLLECT INTO 的使用方法。

```
DECLARE

    TYPE V_rd IS RECORD(V_NAME VARCHAR(50),V_PHONE VARCHAR(50));

    TYPE V_type IS TABLE OF V_rd INDEX BY INT;

    v_info V_type;

    c1 CURSOR IS SELECT  NAME,PHONE FROM PERSON.PERSON A,RESOURCES.EMPLOYEE B
WHERE A.PERSONID=B.PERSONID;

BEGIN

    OPEN c1;

    FETCH c1 BULK COLLECT INTO v_info;

    CLOSE c1;

    FOR I IN 1..v_info.COUNT LOOP

        PRINT v_info(I).V_NAME ||v_info(I).V_PHONE;

    END LOOP;

END;

/
```


BULK COLLECT 可以和 SELECT INTO、FETCH INTO、RETURNING INTO 一起使用，BULK COLLECT 之后 INTO 的变量必须是集合类型。

■ 关闭游标

关闭游标的语法为：

```
CLOSE <游标名>;
```

游标在使用完后应及时关闭，以释放它所占用的内存空间。当游标关闭后，不能再从游标中获取数据，否则将报错。如果需要，可以再次打开游标。

5.2.1.1 节中介绍了隐式游标的属性，同样地，每一个显示游标也有 %FOUND、%NOTFOUND、%ISOPEN 和 %ROWCOUNT 四个属性，但这些属性的意义与隐式游标的有一些区别。

- ✓ %FOUND：如果游标未打开，产生一个异常。否则，在第一次拨动游标之前，其值为 NULL。如果最近一次拨动游标时取到了数据，其值为 TRUE，否则为 FALSE。
- ✓ %NOTFOUND：如果游标未打开，产生一个异常。否则，在第一次拨动游标之前，其值为 NULL。如果最近一次拨动游标时取到了数据，其值为 FALSE，否则为 TRUE。
- ✓ %ISOPEN：游标打开时为 TRUE，否则为 FALSE。
- ✓ %ROWCOUNT：如果游标未打开，产生一个异常。如游标已打开，在第一次拨动游标之前其值为 0，否则为最近一次拨动后已经取到的元组数。

下面的例子说明了显示游标属性的使用方法，对于基表 EMP_SALARY，输出表中的前 5 行数据。如果表中的数据不足 5 行，则输出表中的全部数据。

```
DECLARE

    CURSOR c1 FOR SELECT * FROM OTHER.EMP_SALARY;

    my_ename  CHAR(10);

    my_empno  NUMERIC(4);

    my_sal    NUMERIC(7,2);

BEGIN

    OPEN c1;
```

```

LOOP

    FETCH c1 INTO my_ename, my_empno, my_sal;

    EXIT WHEN c1%NOTFOUND;    /* 当游标取不到数据时跳出循环 */

    PRINT my_ename || ' ' || my_empno || ' ' || my_sal;

    EXIT WHEN c1%ROWCOUNT=5; /* 已经输出了 5 行数据，跳出循环*/

END LOOP;

CLOSE c1;

END;

/

```

5.2.2 动态游标

与静态游标不同，动态游标在声明部分只是先声明一个游标类型的变量，并不指定其关联的查询语句，在执行部分打开游标时才指定查询语句。动态游标的使用主要在定义和打开时与显式游标不同，下面进行详细介绍，拨动游标与关闭游标可参考 5.2.1.2 节中的介绍。

■ 定义动态游标

定义动态游标的语法如下：

```
CURSOR <游标名>;
```

■ 打开动态游标

打开动态游标的语法如下：

```

OPEN <游标名><for 表达式>;

<for 表达式>::=<for_item1>|<for_item2>

<for_item1>:::= FOR <查询表达式>

<for_item2>:::= FOR <表达式> [USING <绑定参数> {,<绑定参数>}]

```

动态游标在 OPEN 时通过 FOR 子句指定与其关联的查询语句。

下面的例子使用动态游标输出员工的姓名、工号和薪水。

```

DECLARE

    my_ename  CHAR(10);

    my_empno  NUMERIC(4);

```

```
my_sal      NUMERIC(7,2);

c1 CURSOR;

BEGIN

    OPEN c1 FOR SELECT * FROM OTHER.EMPSALARY;

    LOOP

        FETCH c1 INTO my_ename, my_empno, my_sal;

        EXIT WHEN c1%NOTFOUND;

        PRINT '姓名'||my_ename || '工号' || my_empno || ' 薪水' || my_sal;

    END LOOP;

    CLOSE c1;

END;

/
```

动态游标关联的查询语句还可以带有参数，参数以“?”指定，同时在打开游标语句中使用 USING 子句指定参数，且参数的个数和类型与语句中的“?”必须一一匹配。下面的例子说明了如何使用关联的语句中带有参数的动态游标。

```
DECLARE

    str VARCHAR;

    CURSOR csr;

BEGIN

    OPEN csr FOR 'SELECT LOGINID FROM RESOURCES.EMPLOYEE WHERE TITLE =? OR
TITLE =?' USING '销售经理','总经理';

    LOOP

        FETCH csr INTO str;

        EXIT WHEN csr%NOTFOUND;

        PRINT str;

    END LOOP;

    CLOSE csr;

END;

/
```

5.2.3 游标变量（引用游标）

游标变量不是真正的游标对象，而是指向游标对象的一个指针，因此是一种引用类型，也可以称为引用游标。

定义游标变量的语法为：

```
<游标变量名> CURSOR[:= <源游标名>];
```

或

```
TYPE <类型名> IS REF CURSOR [RETURN <DMSQL 数据类型>];
```

```
<游标变量名><类型名>;
```

如果定义引用游标的时候没有赋值。可以在<执行部分>中对它赋值。此时引用游标可以继承所有源游标的属性。如果源游标已经打开，则此引用游标也已经打开，引用游标指向的位置和源游标也是完全一样的。

还可以像使用动态游标一样，在打开引用游标时为其动态关联一条查询语句。

引用游标有以下几个特点：

- ✓ 引用游标不局限于一个查询，可以为一个查询声明或者打开一个引用游标，然后对其结果集进行处理，之后又可以将这个引用游标为其它的查询打开；
- ✓ 可以对引用游标进行赋值；
- ✓ 可以像用一个变量一样在一个表达式中使用引用游标；
- ✓ 引用游标可以作为一个子程序的参数；
- ✓ 可以使用引用游标在 DMSQL 程序的不同子程序中传递结果集。

下面的例子使用引用游标在子程序中传递结果集。

```
DECLARE

TYPE Emptytype IS REF CURSOR RETURN PERSON.PERSON%ROWTYPE;

emp Emptytype;

PROCEDURE process_emp(emp_v IN Emptytype) IS

    person PERSON.PERSON%ROWTYPE;

BEGIN

    LOOP

        FETCH emp_v INTO person;
```

```

        EXIT WHEN emp_v%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('姓名: '||person.NAME || ' 电话: ' ||
person.PHONE);

    END LOOP;

END;

BEGIN

    OPEN emp FOR SELECT A.* FROM PERSON.PERSON A,RESOURCES.EMPLOYEE B WHERE
A.PERSONID=B.PERSONID;

    process_emp(emp);

    CLOSE emp;

END;

/

```

5.2.4 使用游标更新、删除数据

可以使用游标更新或删除结果集中的数据。若需要使用游标更新或删除数据，则在游标关联的查询语句中一定要使用“FOR UPDATE 选项”。FOR UPDATE 选项出现在查询语句中，用于对要修改的行上锁，以防止用户在同一行上进行修改操作。关于 FOR UPDATE 选项的详细介绍可参考《DM8_SQL 语言使用手册》。

当游标拨动到需要更新或删除的行时，就可以使用 UPDATE/DELETE 语句进行数据更新/删除。此时必须在 UPDATE/DELETE 语句结尾使用“WHERE CURRENT OF 子句”，以限定删除/更新游标当前所指的行。语法如下：

```
< WHERE CURRENT OF 子句> ::=WHERE CURRENT OF <游标名>
```

例如，使用游标更新表中的数据。

```

DECLARE

    CURSOR csr is SELECT SALARY FROM RESOURCES.EMPLOYEE WHERE TITLE='销售经理'
FOR UPDATE;

BEGIN

    OPEN csr;

    IF csr%ISOPEN THEN

```

```
        FETCH csr;

        UPDATE RESOURCES.EMPLOYEE SET SALARY = SALARY + 10000 WHERE CURRENT
OF csr;

    ELSE

        PRINT 'CURSOR IS NOT OPENED';

    END IF;

    CLOSE csr;

END;

/
```

下面的例子使用游标删除表中的数据。

```
DECLARE

    CURSOR dcsr is SELECT EMPNO FROM OTHER.EMPSALARY WHERE ENAME='KING' FOR
UPDATE;

BEGIN

    OPEN dcsr;

    IF dcsr%ISOPEN THEN

        FETCH dcsr;

        DELETE FROM OTHER.EMPSALARY WHERE CURRENT OF dcsr;

    ELSE

        PRINT 'CURSOR IS NOT OPENED';

    END IF;

    CLOSE dcsr;

END;

/
```

5.2.5 使用游标 FOR 循环

游标通常与循环联合使用，以遍历结果集数据。实际上，DM8SQL 程序还提供了一种将两者综合在一起的语句，即游标 FOR 循环语句。游标 FOR 循环自动使用 FOR 循环依次读

取结果集中的数据。当 FOR 循环开始时，游标会自动打开（不需要使用 OPEN 方法）；每循环一次系统自动读取游标当前行的数据（不需要使用 FETCH）；当数据遍历完毕退出 FOR 循环时，游标被自动关闭（不需要使用 CLOSE），大大降低了应用程序的复杂度。

5.2.5.1 隐式游标 FOR 循环

隐式游标 FOR 循环的语法如下：

```
FOR <cursor_record> IN (<查询语句>)
LOOP
<执行部分>
END LOOP;
```

其中，<cursor_record>是一个记录类型的变量。它是 DMSQL 程序根据 SQL%ROWTYPE 属性隐式声明出来的，不需要显式声明。也不能显式声明一个与 <cursor_record>同名的记录，会导致逻辑错误。

FOR 循环不断地将行数据读入变量<cursor_record>中，在循环中也可以存取 <cursor_record>中的字段。

例如，下面的例子使用了隐式游标 FOR 循环。

```
BEGIN

    FOR v_emp IN (SELECT * FROM RESOURCES.EMPLOYEE)

    LOOP

        DBMS_OUTPUT.PUT_LINE(V_EMP.TITLE || '的工资' || V_EMP.SALARY);

    END LOOP;

END;

/
```

5.2.5.2 显式游标 FOR 循环

显式游标 FOR 循环的语法如下：

```
FOR <cursor_record> IN <游标名>
LOOP
```

<执行部分>

END LOOP;

显式游标 FOR 循环与隐式游标的语法和使用方式都非常相似，只是关键字“IN”后不指定查询语句而是指定显式游标名，<cursor_record>则为<游标名>%ROWTYPE 类型的变量。

下面的例子使用的是显式游标 FOR 循环。

```
DECLARE

    CURSOR cur_emp IS SELECT * FROM RESOURCES.EMPLOYEE;

BEGIN

    FOR V_EMP IN CUR_EMP LOOP

        DBMS_OUTPUT.PUT_LINE(V_EMP.TITLE || '的工资' || V_EMP.SALARY);

    END LOOP;

END;

/
```

5.3 动态 SQL

本章前面的小节中介绍的大都是 DMSQL 程序中的静态 SQL，静态 SQL 在 DMSQL 程序进行编译时是明确的 SQL 语句，处理的数据库对象也是明确的，这些语句在编译时就可进行语法和语义分析处理。与之对应，DMSQL 程序还支持动态 SQL，动态 SQL 指在 DMSQL 程序进行编译时是不确定的 SQL，编译时对动态 SQL 不进行处理，在 DMSQL 程序运行时才动态地生成并执行这些 SQL。

在应用中，常常需要根据用户的选择（如表名、列名、排序方式等）来生成 SQL 语句并执行，这些 SQL 不能在应用开发时确定，此时就需要使用动态 SQL。另外，在 DMSQL 程序中，DDL 语句只能通过动态 SQL 执行。

使用 EXECUTE IMMEDIATE 动态地准备和执行一条 SQL 语句，语法如下：

```
EXECUTE IMMEDIATE <SQL 动态语句文本> [USING <参数> {,<参数>}];
```

动态 SQL 的执行首先分析 SQL 动态语句文本，检查是否有错误，如果有错误则不执行，并在 SQLCODE 中返回错误码；如果没发现错误则继续执行。

**注意:****下列语句不能作动态SQL语句: CLOSE、DECLARE、FETCH、OPEN。**

动态 SQL 中可以使用参数, 并支持两种指定参数的方式: 用 “?” 表示参数和用 “:variable” 表示参数。

当用 “?” 表示参数时, 在指定参数值时可以是任意的值, 但参数值个数一定要与 “?” 的个数相同, 同时数据类型一定要匹配 (能够互相转换也可以), 不然会报数据类型不匹配的错误。

例如:

```
CREATE OR REPLACE PROCEDURE proc(cate IN INT, time IN DATE) AS
DECLARE
    str_sql varchar := 'SELECT NAME,PUBLISHER from PRODUCTION.PRODUCT WHERE
PRODUCT_SUBCATEGORYID = ? AND PUBLISHTIME> ?';
BEGIN
    EXECUTE IMMEDIATE str_sql USING cate,time;
EXCEPTION
    WHEN OTHERS THEN PRINT 'error';
END;
/
CALL proc(4, '2001-01-01');
```

当用 “:variable” 表示参数时, 若 SQL 动态语句文本为普通语句方式, 则系统将其转化为 “?” 进行处理。若 SQL 动态语句文本为脚本方式 (语句块方式或多条用分号分隔的 SQL 语句), 则保留 “:variable” 参数形式。此时需要注意, 若参数列表中有同名参数, 系统将这些同名参数视为同一个参数, 只需要对应传入一个参数值即可, 否则会报 “参数个数不匹配.error code = -3205” 的错误。

例如, 下面的 DMSQL 程序中 sql_str 采用普通语句方式, 其中的两个 :x 被转化为? 进行处理, 程序执行成功。

```
DECLARE
    sql_str VARCHAR := 'UPDATE PRODUCTION.PRODUCT SET SELLSTARTTIME=:x,
SELLENDTIME=:x WHERE NAME='红楼梦'';
```

```
BEGIN

    EXECUTE IMMEDIATE SQL_STR USING '2015-01-01','2018-01-01';

END;

/
```

而在下面的 DMSQL 程序中 sql_str 写为脚本方式，则将两个 :x 视为同名参数，因此在动态执行时指定两个实际参数会报错。

```
DECLARE

    sql_str VARCHAR := 'BEGIN UPDATE PRODUCTION.PRODUCT SET SELLSTARTTIME=:x,
SELLENDTIME=:x WHERE NAME=''红楼梦'';END;';

BEGIN

    EXECUTE IMMEDIATE SQL_STR USING '2015-01-01','2018-01-01';

END;

/

[-5402]:参数个数不匹配.
```

如果 SQL 动态语句文本为 SELECT 语句，且结果集只会是一条记录，可通过 INTO 语句来操作查询到的结果集，把查询到的结果存储到变量中。其语法如下：

```
EXECUTE IMMEDIATE <SQL 动态语句文本> [INTO <变量>{,<变量>}] [USING <参数>{,<参数>}];
```

关于变量的说明和当查询结果集不是一条记录时的处理可参考 5.1.2 节。

例如：

```
DECLARE

    p_name VARCHAR(50);

    p_publish VARCHAR(50);

    str_sql VARCHAR := 'SELECT NAME,PUBLISHER from PRODUCTION.PRODUCT WHERE
AUTHOR = ?';

BEGIN

    EXECUTE IMMEDIATE str_sql INTO p_name,p_publish USING '曹雪芹,高鹗';

    PRINT p_name;

    PRINT p_publish;
```

```

EXCEPTION

    WHEN OTHERS THEN PRINT 'error';

END;

/

```

5.4 返回查询结果集

在存储过程或客户端 DMSQL 程序中执行了不带 INTO 子句的查询语句时，系统将在调用结束时将该查询结果集返回给调用者。当出现多个查询语句时，只有第一个查询语句的结果集被返回，如果想查看其他结果集，可在 DISql 中输入命令“more”查看下一个结果集。

例如：

```

CREATE OR REPLACE PROCEDURE PRODUCTION.proc_result(p_publisher VARCHAR(50))
AS
BEGIN
    SELECT NAME,AUTHOR FROM PRODUCTION.PRODUCT WHERE PUBLISHER=p_publisher;

    SELECT NOWPRICE FROM PRODUCTION.PRODUCT WHERE PUBLISHER=p_publisher;

END;

/

```

调用 PRODUCTION.proc_result 过程，仅返回第一条查询语句的结果集：

```
SQL> CALL PRODUCTION.proc_result('中华书局');
```

行号	NAME	AUTHOR
1	红楼梦曹雪芹	高鹗
2	水浒传施耐庵	罗贯中

再调用 more 命令，返回第二条查询语句的结果集：

```
SQL> more
```

行号	NOWPRICE
1	15.2000
2	14.3000

**注意:****存储函数中不能包含返回结果集的查询语句。**

5.5 自治事务

通过将一个 DMSQL 语句块定义成自治事务，可以将该块中的 DML 语句和调用程序的事务环境隔离开。如此一来，该语句块就成为一个由其他事务启动的独立的事务，前一个事务被称为主事务。在自治事务块中，主事务是挂起的；等待自治事务完成后，会话自动切换回主事务。

一个定义了自治事务的语句块称为自治例程。

5.5.1 定义自治事务

定义自治事务，需要在 DMSQL 程序的声明部分添加如下语法的语句：

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

自治事务的定义语句可以放在 DMSQL 程序声明部分的任何地方，但推荐放在数据结构声明之前。

作为自治事务的 DMSQL 语句块可以是下面中的一种：

- ✓ 最顶层的（不是嵌套的）匿名 DMSQL 语句块
- ✓ 函数和过程，或者在一个包里定义或者是一个独立的程序
- ✓ 对象类型的方法
- ✓ 数据库触发器
- ✓ 嵌套子过程

例如：

```
CREATE TABLE test(c VARCHAR);

INSERT INTO testVALUES('主事务');

DECLARE

    PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN

    INSERT INTO test VALUES('自治事务');
```

```

COMMIT;  --一定要提交，否则会报错。详情请参考 5.5.2 节

END;

/

ROLLBACK;

```

查询表 test:

```
SELECT * FROM test;
```

行号	c
1	自治事务

可以看到，表 test 中有记录“自治事务”，其所在自治事务提交，不受主事务回滚影响，而主事务插入的记录“主事务”被回滚。

5.5.2 自治事务完整性与死锁检测

自治事务的语句块执行完成时，如果事务仍处于活动状态，则系统需要报错：检测到活动的自治事务处理，并将未提交事务回滚，所以需要在语句块结束处显式添加 COMMIT 或 ROLLBACK 语句。

另一方面，COMMIT 和 ROLLBACK 语句只是结束了活动的自治事务，但不会终止自治例程。实际上，可以在一个自治块中使用多个 COMMIT、ROLLBACK 语句。

例如：

```

CREATE TABLE test (c VARCHAR);

INSERT INTO test VALUES('主事务');

DECLARE

    PRAGMA AUTONOMOUS_TRANSACTION;

BEGIN

    INSERT INTO test VALUES('自治事务 1');

    ROLLBACK;

    INSERT INTO test VALUES('自治事务 2');

    COMMIT;

END;

```

```
/
ROLLBACK;
```

查询表 test:

```
SELECT * FROM test;

行号      C
-----
1          自治事务 2
```

自治事务相对主事务是完全独立的。由于执行自治事务时主事务处于挂起状态，如果自治事务需要的锁资源已经被主事务拥有则会产生死锁报错。

5.5.3 自治事务嵌套

自治块中可以调用其它被定义自治事务的过程，从而产生嵌套的自治事务。例如：

```
CREATE TABLE TEST (C VARCHAR);

CREATE OR REPLACE PROCEDURE PROC_AUTO1
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    DELETE TEST WHERE C='嵌套事务';
    COMMIT;
    INSERT INTO TEST VALUES ('自治嵌套事务');
COMMIT;
END;

/

CREATE OR REPLACE PROCEDURE PROC_AUTO2
IS
declare
PRAGMA AUTONOMOUS_TRANSACTION;
```

```
BEGIN

    CALL PROC_AUTO1;

INSERT INTO TEST VALUES ('自治语句块');

COMMIT;

END;

/

INSERT INTO TEST VALUES ('嵌套事务');

COMMIT;

INSERT INTO TEST VALUES ('主事务');

CALL PROC_AUTO2;

ROLLBACK;
```

查询表 test:

```
SELECT * FROM test;
```

行号	C
1	自治嵌套事务
2	自治语句块

DM 的 INI 参数 TRANSACTIONS 指定一个会话中可以并发的自治事务数量，其默认值为 75。如果实际生成自治事务数超出该参数限定值时，服务器将报错：“并发事务数超出上限”。该参数理论值上下限分别为 1000 和 1。

6 DMSQL 程序异常处理

DMSQL 程序在应用运行时不可避免地会发生一些错误，这些错误常常并不是 DMSQL 程序本身的设计和编码问题，而是由于应用运行时由于一些未预计的操作或硬件异常等导致产生超出 DMSQL 程序预计处理范围的数据等错误，我们将这些错误称为异常。应该在 DMSQL 程序中对异常进行处理，否则将造成应用的异常退出。

6.1 异常处理的优点

异常一般是在 DMSQL 程序执行发生错误时由服务器抛出，也可以在 DMSQL 块中由程序员在一定的条件下显式抛出。无论是哪种形式的异常，DMSQL 程序的执行都会被中止，程序控制转至 DMSQL 程序的异常处理部分。程序员可以在异常处理部分编写一段程序对异常进行处理，以避免 DMSQL 程序的异常退出。

异常被处理结束后，异常所处 DMSQL 块的执行便告结束。所以一旦发生异常，则在所处 DMSQL 块的可执行部分中，从发生异常的地方开始，后续的代码将不再执行。如果没有对异常进行任何处理，异常将被传递到调用者，由调用者统一处理。

6.2 预定义异常

为方便用户编程，DM 提供了一些预定义的异常，这些异常与常见的 DM 错误相对应，如表 6.1 所示。

表 6.1 DM 预定义异常

异常名	错误号	错误描述
INVALID_CURSOR	-4535	无效的游标操作
ZERO_DIVIDE	-6103	除 0 错误
DUP_VAL_ON_INDEX	-6602	违反唯一性约束
TOO_MANY_ROWS	-7046	SELECT INTO 中包含多行数据
NO_DATA_FOUND	-7065	数据未找到

此外，还有一个特殊的异常名 OTHERS，它处理所有没有明确列出的异常。OTHERS 对应的异常处理语句必须放在其他异常处理语句之后。

下面的例子中异常处理部分对指定的预定义异常进行处理。


```

DECLARE

    v_name VARCHAR(50);

    v_phone VARCHAR(50);

BEGIN

    SELECT  NAME,PHONE  INTO V_NAME,V_PHONE FROM PERSON.PERSON
A,RESOURCES.EMPLOYEE B WHERE A.PERSONID=B.PERSONID AND B.TITLE='销售代表';

    PRINT v_name||' '||v_phone;

EXCEPTION

    WHEN TOO_MANY_ROWS THEN

        PRINT 'SELECT INTO 中包含多行数据';

END;

/

```

6.3 用户自定义异常

除了 DM 预定义的异常外，用户还可以在 DMSQL 程序中自定义异常。程序员可以把一些特定的状态定义为异常，在一定的条件下抛出，然后利用 DMSQL 程序的异常机制进行处理。

DMSQL 程序支持两种自定义异常的方法。

■ 使用 **EXCEPTION FOR** 将异常变量与错误号绑定

语法如下：

```
<异常变量名> EXCEPTION [FOR <错误号> [, <错误描述>]];
```

在 DMSQL 程序的声明部分使用该语法声明一个异常变量。其中，FOR 子句用来为异常变量绑定错误号 (SQLCODE 值) 及错误描述。<错误号>必须是-20000 到-30000 间的负数值，<错误描述>则为字符串类型。如果未显式指定错误号，则系统在运行中在-10001 到-15000 区间中顺序为其绑定错误值。

例如：

```

CREATE OR REPLACE PROCEDURE proc_exception (v_personid INT)
IS

    v_name VARCHAR(50);

```

```

v_email VARCHAR(50);

e1 EXCEPTION FOR -20001, 'EMAIL 为空';

BEGIN

    SELECT NAME,EMAIL INTO v_name,v_email FROM PERSON.PERSON WHERE
PERSONID=v_personid;

    IF v_email='' THEN

        RAISE e1;

    ELSE

        PRINT v_name || v_email;

    END IF;

EXCEPTION

    WHEN E1 THEN

        PRINT v_name || ' ' ||SQLCODE||' ' ||SQLERRM;

END;

/

CALL proc_exception(2);

```

- 使用 **EXCEPTION_INIT** 将一个特定的错误号与程序中所声明的异常标示符关联起来语法如下：

```

<异常变量名> EXCEPTION;

PRAGMA EXCEPTION_INIT(<异常变量名>, <错误号>);

```

在 DMSQL 程序的声明部分使用这个语法先声明一个异常变量，再使用 **EXCEPTION_INIT** 将一个特定的错误号与这个异常变量关联起来。这样可以通过名字引用任意的 DM 服务器内部异常（DM 服务器内部异常可参考《DM8 程序员手册》附录 1），并且可以通过名字为异常编写一适当的异常处理。如果希望使用 **RAISE** 语句抛出一个用户自定义异常，则与异常关联的错误号必须是 -20000 到 -30000 之间的负数值。

下面的例子使用 **EXCEPTION_INIT** 将 DM 服务器的特定错误“-2206：缺少参数值”与异常变量 e1 关连起来。

```

CREATE OR REPLACE PROCEDURE proc_exception2(v_personid INT)
IS

```

```

TYPE Rec_type IS RECORD (V_NAME VARCHAR(50),V_EMAIL VARCHAR(50));

v_col Rec_type;

e1 EXCEPTION;

PRAGMA EXCEPTION_INIT(e1, -2206);

BEGIN

    SELECT NAME,EMAIL INTO v_col FROM PERSON.PERSON WHERE PERSONID=v_personid;

    IF v_col.V_EMAIL='' THEN

        RAISE e1;

    ELSE

        PRINT v_col.V_NAME || v_col.V_EMAIL;

    END IF;

EXCEPTION

    WHEN e1 THEN

        PRINT v_col.V_NAME || ' ' ||SQLCODE||': ' ||SQLERRM(SQLCODE);

END;

/

CALL proc_exception2(2);

```

下面的例子则使用 EXCEPTION_INIT 定义用户自定义异常，并使用 RAISE 语句抛出该异常。

```

CREATE OR REPLACE PROCEDURE proc_exception3 (v_personid INT)

IS

    TYPE Rec_type IS RECORD (V_NAME VARCHAR(50),V_EMAIL VARCHAR(50));

    v_col Rec_type;

    e2 EXCEPTION;

    PRAGMA EXCEPTION_INIT(e2, -25000);

BEGIN

    SELECT NAME,EMAIL INTO v_col FROM PERSON.PERSON WHERE PERSONID=v_personid;

    IF v_col.V_EMAIL='' THEN

        RAISE e2;

```

```

ELSE

    PRINT v_col.V_NAME || v_col.V_EMAIL;

END IF;

EXCEPTION

    WHEN e2 THEN

        PRINT v_col.V_NAME || '的邮箱为空';

END;

/

CALL proc_exception3(2);

```

6.4 异常的抛出

DM8SQL 程序运行时如果发生错误，系统会自动抛出一个异常。

此外，程序员还可以使用 RAISE 主动抛出一个异常。例如，当程序运行并不违反数据库规则，但是不满足应用的业务逻辑时，可以主动抛出一个异常并进行处理。

使用 RAISE 抛出异常分为有异常名和无异常名两种情况。

■ 有异常名

语法如下：

```
RAISE <异常名>
```

可以使用 RAISE 语句抛出一个系统预定义异常（参考 6.2 节）或用户自定义异常。

6.3 节中已经有使用 RAISE 语句的示例，这里不再举例。

■ 无异常名

如果没有在声明部分定义异常变量，也可以在执行部分使用 DM 提供的系统过程直接抛出自定义异常，语法如下：

```
RAISE_APPLICATION_ERROR (

    ERR_CODE IN INT,

    ERR_MSG IN VARCHAR(2000)

);
```

ERR_CODE：错误码，取值范围为-20000~-30000；

ERR_MSG：用户自定义的错误信息，长度不能超过 2000 字节。

例如：

```
CREATE OR REPLACE PROCEDURE proc_exception4 (v_personid INT)
IS
    TYPE Rec_type IS RECORD (V_NAME VARCHAR(50),V_EMAIL VARCHAR(50));
    v_col Rec_type;
BEGIN
    SELECT NAME,EMAIL INTO v_col FROM PERSON.PERSON WHERE PERSONID=v_personid;

    IF v_col.V_EMAIL='' THEN

        RAISE_APPLICATION_ERROR(-20001, '邮箱为空');

    ELSE

        PRINT v_col.V_NAME || v_col.V_EMAIL;

    END IF;
EXCEPTION
    WHEN OTHERS THEN

        PRINT SQLCODE || ' ' ||v_col.V_NAME || ':' ||SQLERRM;
END;
/
CALL proc_exception4(2);
```

6.5 内置函数 SQLCODE 和 SQLERRM

DM8SQL 程序提供了内置函数 SQLCODE 和 SQLERRM，程序员可以在异常处理部分通过这两个函数获取异常对应的错误码和描述信息。SQLCODE 返回错误码，为一个负数。SQLERRM 返回异常的描述信息，为字符串类型。

例如：

```
DECLARE

    e1 EXCEPTION FOR -20001, 'EMAIL 为空';
BEGIN

    RAISE e1;
EXCEPTION
```

```

WHEN e1 THEN

    PRINT  SQLCODE || ' ' || SQLERRM;

END;

/

```

若异常为 DM 服务器错误，则 SQLERRM 返回该错误的描述信息，否则 SQLERRM 的返回值遵循以下规则：

- 如果错误码在-15000 至-19999 间，返回 'User-Defined Exception'；
- 如果错误码在-20000 至-30000 之间，返回 'DM-<错误码绝对值>'；
- 如果错误码大于 0 或小于-65535，返回 '-<错误码绝对值>: non-DM exception'；
- 否则，返回 'DM-<错误码绝对值>: Message <错误码绝对值> not found;'。

如果程序员想查询某个 DM 服务器错误码对应的错误描述信息，也可以在执行部分或异常处理部分使用错误码参数调用 SQLERRM 函数，语法如下：

```

VARCHARSQLERRM(ERROR_NUMBER INT(4));

```

这种 SQLERRM 调用方式的返回值也遵循上面所述的规则。

例如：

```

BEGIN

    PRINT 'SQLERRM(-6815): ' || SQLERRM(-6815);

END;

```

执行结果为：

```

SQLERRM(-6815): 指定的对象数据库中不存在

```

6.6 异常处理部分

DMSQL 程序的异常处理部分对执行过程中抛出的异常进行处理，可以处理一个或多个异常。语法如下：

```

EXCEPTION {<异常处理语句>;}

```

```

<异常处理语句> ::= WHEN <异常处理器>

```

```

<异常处理器> ::= <异常名>{ OR <异常名>} THEN <执行部分>;

```

EXCEPTION 关键字表示异常处理部分的开始。如果在语句块的执行中出现异常，执行就被传递给语句块的异常处理部分。而如果在本语句块的异常处理部分没有相应的异常处理器对它进行处理，系统就会中止此语句块的执行，并将此异常传递到该语句块的上一层语句块或其调用者，这样一直到最外层。如果始终没有找到相应的异常处理器，则中止本次调用语句的处理，并向用户报告错误。

异常处理部分是可选的。但是如果出现 EXCEPTION 关键字时，必须至少有一个异常处理器。异常处理器可以按任意次序排列，只有 OTHERS 异常处理器必须在最后，它处理所有没有明确列出的异常。此外，同一个语句块内的异常处理器不允许处理重复的异常。

一个异常处理器可以同时多个异常进行统一处理，在 WHEN 子句中用 OR 分隔多个异常名。例如：

```
CREATE OR REPLACE PROCEDURE proc_exception5 (score INT) AS

    e1 EXCEPTION FOR -20001, '补考';

    e2 EXCEPTION FOR -20002, '不能补考';

BEGIN

    IF score BETWEEN 90 AND 100 THEN

        PRINT '考试通过';

    ELSEIF score BETWEEN 80 AND 90 THEN

        RAISE e1;

    ELSE

        RAISE e2;

    END IF;

EXCEPTION

    WHEN e1 OR e2 THEN    --同时处理多个异常

        PRINT '考试不通过';

END;

/

CALL proc_exception5(50);
```

下层语句块中出现的异常，如果在该语句块中没有对应的异常处理器，可以被上层语句块或其调用者的异常处理器处理。例如

```
DECLARE

    e1 EXCEPTION;

BEGIN

    PRINT '下层块之前';

    BEGIN

        RAISE e1;

    END;

    PRINT '下层块之后';

EXCEPTION

    WHEN e1 THEN

        PRINT '外层块处理下层块抛出的异常';

END;

/
```

执行结果为:

下层块之前

外层块处理下层块抛出的异常

要注意异常变量的有效范围。例如，下面的例子中在上层语句块中引用下层语句块定义的异常变量，DMSQL 程序无法运行通过，会报错。

```
BEGIN

    PRINT '下层块之前';

    DECLARE

        e1 EXCEPTION; /* 异常变量定义在下层语句块中 */

    BEGIN

        RAISE e1;

    END;

    PRINT '下层块之后';

EXCEPTION

    WHEN e1 THEN /* 报错。因为异常变量 e1 在此范围内没有定义 */

        PRINT '外层块处理下层块抛出的异常';
```



```
END;  
  
/
```

有时候，在异常处理器的执行中又可能出现新的异常。这时，系统将新出现的异常作为当前需要处理的异常，并向上层传递。例如

```
DECLARE  
  
    e1 EXCEPTION for -20000, 'EC_01';  
  
    e2 EXCEPTION for -30000, 'EC_02';  
  
BEGIN  
  
    BEGIN  
  
        RAISE e1; /* 下层语句块抛出异常 E1 */  
  
    EXCEPTION  
  
        WHEN e1 THEN  
  
            RAISE e2; /* 在 e1 的处理过程中，抛出 e2。但是下层块并没有处理 e2*/  
  
    END;  
  
EXCEPTION  
  
    WHEN e1 THEN /* 异常变量 e1 的异常处理器 */  
  
        PRINT 'EXCEPTION e1 CATCHED';  
  
    WHEN e2 THEN /* 异常变量 e2 的异常处理器 */  
  
        PRINT 'EXCEPTION e2 CATCHED';  
  
END;  
  
/
```

7 基于 C、JAVA 语法的 DMSQL 程序

在程序员们的印象中，过程化 SQL 语言是较复杂的，因为它有自己的语法规则，用户通常都不会去记那么多复杂的语法，只是在用的时候查看手册，所以当需要编写一个逻辑较复杂的 DMSQL 程序可能会碰到一些困难。

DM8 实现了用 C、JAVA 语言语法作为 DMSQL 程序的可选语法，这就为那些了解 C、JAVA 语言的人提供了很大的方便性，无需查看手册就可以很自如的完成一个语句块，对 SQL 程序员而言，这个功能无疑是他们梦寐以求的。

7.1 C 语法 DMSQL 程序

编写 C 语言语法的 DMSQL 程序，定义语句块时不需要用 BEGIN 及 END 把语句包含起来，而是直接用大括号括住即可。

下面给出两个 C 语法的 DMSQL 程序的例子来更具体地说明。

例 1:

```
{  
  
    int      i = 0;  
  
    VARCHAR v_name;  
  
    VARCHAR v_email;  
  
    for(i = 1; i < =7; i++)  
  
    {  
  
        SELECT  NAME,EMAIL INTO v_name,v_email FROM PERSON.PERSON WHERE  
PERSONID=i;  
  
        PRINT CONCAT(i,v_name);      //使用数据库函数 concat  
  
        PRINT v_email;  
  
    }  
}  
  
/
```

例 2:

```

{
    try
    {
        SELECT 1/0;
    }
    catch (EXCEPTION EX)
    {
        THROW NEW EXCEPTION(-20002, 'TEST');
    }
}
/

```

使用 C 语法的 DMSQL 程序时，可以很自由地调用一些系统内部函数（如上面例子中的 `concat()`）、存储函数、过程等等。可以定义像 C# 中的一些数据类型，如 `STRING` 类型，还可以定义 C 语言中的基本数据类型，如上面例子中的 `int`，另外还支持全部的 SQL 类型，DM 内部定义的类型包括 `EXCEPTION` 类、数组类型、游标类型等。

在 C 语法 DMSQL 程序中使用游标时需要在 `OPEN`、`FETCH` 及 `CLOSE` 后使用“`CURSOR`”关键字，下面是一个使用 C 语法 DMSQL 程序操作游标的例子。

```

{
    VARCHAR v_name;
    VARCHAR v_phone;

    int i=0;

    CURSOR csr IS SELECT  NAME,PHONE FROM PERSON.PERSON A,RESOURCES.EMPLOYEE
B WHERE A.PERSONID=B.PERSONID;

    string    str;

    {
        OPEN CURSOR csr;

        if (csr%ISOPEN)
        {
            PRINT 'CURSOR IS ALREADY FIRST';

            while(i<=csr%ROWCOUNT)

```

```

        {

            FETCH CURSOR csr INTO v_name,v_phone;

            PRINT csr%ROWCOUNT;  /* 从游标上取数据后, csr%ROWCOUNT 属性加 1 */

            PRINT CONCAT(v_name,v_phone);

            i++;

        }

        CLOSE CURSOR csr;

    }

else

    PRINT 'CURSOR IS NOT OPENED';

OPEN CURSOR csr;  /* 再次打开, 则会初始化各个属性 */

if (csr%ISOPEN)

{

    PRINT 'CURSOR IS  OPENED AGAIN';

    PRINT csr%ROWCOUNT;

}

else

    PRINT 'CURSOR IS NOT OPENED';

}

}

/

```

7.2 JAVA 语法 DMSQL 程序

当使用 JAVA 语法编写 DMSQL 程序时, 可以按 JAVA 语言风格定义一个类, 之后在 DMSQL 程序中就可以以 JAVA 语言风格创建这个类对象并调用类的属性和方法。

下面是一个使用 JAVA 语法的 DMSQL 程序示例。

```

create or replace java public class Class1 {

    int b = 5;

```

```

    Class1 (int a)

    {

        this.b = a;

    }

    int testFun(int a)

    {

        b = a;

        return b;

    }

}

/

DECLARE

    c1  class1;

    c2  class1;

BEGIN

    c1 = new class1( );

    PRINT c1.b;

    PRINT c1.testFun(9);

    c2 = new class1(8);

    PRINT c2.testFun(c2.b);

END;

/

```

在 JAVA 语法 DMSQL 程序中使用游标时需要在 OPEN、FETCH 及 CLOSE 后使用“CURSOR”关键字。下面的例子演示了如何使用 JAVA 语法的 DMSQL 程序操纵游标。

```

create or replace java public class cls_java

{

    VARCHAR v_name;

    VARCHAR v_phone;

    int I=0;

    int fun1()

```

```
{  
  
    CURSOR csr FOR SELECT  NAME,PHONE FROM PERSON.PERSON  
A,RESOURCES.EMPLOYEE B WHERE A.PERSONID=B.PERSONID;  
  
    OPEN CURSOR csr;  
  
    while(i<=csr%ROWCOUNT)  
  
    {  
  
        FETCH CURSOR csr INTO v_name,v_phone;  
  
        PRINT CONCAT(v_name,v_phone);  
  
        i++;  
  
    }  
  
    CLOSE CURSOR csr;  
  
}  
  
/  
  
DECLARE  
  
    c1  cls_java;  
  
BEGIN  
  
    c1 = new cls_java();  
  
    c1.fun1();  
  
END;  
  
/
```

8 DMSQL 程序调试

DM 提供了 DMSQL 程序调试功能，可调试直接执行的 DMSQL 程序或非 DDL 语句，以便开发人员定位 DMSQL 程序中存在的错误。

8.1 使用命令行工具 dmdbg 调试 DMSQL 程序

dmdbg 是 DM 提供的用于调试 DMSQL 程序的命令行工具，安装 DM8 数据库管理系统后，在安装目录的“bin”子目录下可找到 dmdbg 执行程序。



注意：

要使用 dmdbg 调试工具，首先要调用系统过程 `SP_INIT_DBG_SYS(1)` 创建调试所需要的包。

8.1.1 dmdbg 工具命令简介

dmdbg 在整个运行过程中可以处于初始状态 (S)、待执行 (W)、执行 (R)、调试 (D)、执行结束 (O) 等不同的状态：

- 初始状态 (S)：工具启动完成后，尚未设置调试语句；
- 待执行状态 (W)：设置调试语句后，等待用户执行；
- 执行状态 (R)：开始执行后，未中断而运行的过程；
- 调试状态 (D)：执行到断点或强制中断后进入交互模式；
- 执行结束 (O)：执行完当前设置的调试语句，并返回结果。

dmdbg 在不同的状态下可以执行不同的操作，如表 8.1 所示，其中备注表示不同的命令分别在哪儿几种状态下可以使用。

表 8.1 dmdbg 工具的调试命令

命令	含义	备注
LOGIN	登录	S/W/D/O
SQL	设置调试语句	S/W/O
B	设置断点	W/D/O
INFO B	显示断点信息	W/D/O
D	取消断点	W/D/O

R	执行语句	W/O
CTRL+C	中断执行	R
L	显示调试脚本	D
C	继续执行	D
N	单步执行	D
S	执行进入	D
F 或 FINISH	执行跳出	D
P	打印变量	D
BT	显示堆栈	D
UP	上移栈帧	D
DOWN	下移栈帧	D
KILL	结束当前执行	W/D/O
QUIT	退出调试工具	S/W/D/O

8.1.2 使用 dmdbg 工具

1. 登录

双击 dmdbg.exe 执行程序，进入 dmdbg 命令行工具窗口。在命令行输入 LOGIN 命令进行登录，会提示用户输入服务名、用户名、密码、端口号、ssl 路径和 ssl 密码，如下图所示。



图 8.1 dmdbg 工具登录界面

- server: 数据库服务名或 IP 地址。localhost 表示本地服务器。默认为 localhost;

- user name 和 password: 用户名和密码, 默认均为 SYSDBA, 密码不回显;
- port: 端口号, 默认为 5236;
- ssl path 和 ssl password: ssl 路径和 ssl 密码, 用于服务器通信加密, 不加密的用户不用设置, 缺省为不设置。

也可以在操作系统的命令行窗口中使用命令行方式登录 dmdbg, 格式如下:

```
DMDBG <用户名>/<口令>[@<服务器名|IP>[:<端口号>]]
```

例如:

```
DMDBG SYSDBA/SYSDBA@LOCALHOST:5236
```

2. 设置语句

命令 SQL 供用户设置需要执行的 SQL 语句, 其命令格式如下:

```
SQL <SQL 语句> ['/' ]
```

SQL 语句是非 DDL 语句或语句块。如果 SQL 是单条语句, 则以分号 “;” 结尾; 如果是语句块, 则语句块后必须以单独一行的斜杠符 “/” 结束。在调试中行数从 SQL 语句实际起始行开始计数。

在实际执行前, 可以多次调用 SQL 命令设置待调试的语句, 而 dmdbg 只是记录最后一次设置的语句。



说明:

如果需要调试自定义的存储函数, 需要和过程调用方式一样, 使用 **call** 调用。

下面我们用一个具体的示例来说明如何使用 SQL 命令设置语句, 并将使用这个例子贯穿后续的调试命令介绍。

首先, 对象和数据初始化, 在 DM 中使用下面的脚本创建表和存储过程:

```
CREATE TABLE STUDENT (ID INT, NAME VARCHAR(20));           --创建游标类型对应的表

INSERT INTO STUDENT VALUES(10010, 'ZHANGSAN');

INSERT INTO STUDENT VALUES(10011, 'LISI');

CREATE OR REPLACE PROCEDURE P0 AS                             --创建需要调试的存储过程
BEGIN
PRINT 'HELLO';
```

```

PRINT 'WORLD';

END;

/

CREATE CLASS MYCYCLE                                --定义类对象
AS
ID INT;
C1 INT;
FUNCTION MYCYCLE(ID INT , C1 INT) RETURN MYCYCLE;
END;

/

CREATE OR REPLACE CLASS BODY MYCYCLE                --定义类体
AS
FUNCTION MYCYCLE(ID INT, C1 INT) RETURN MYCYCLE
AS
BEGIN
    THIS.ID = ID;
    THIS.C1 = C1;
    RETURN THIS;
END;

END;

/

CREATE OR REPLACE PROCEDURE P AS                    --创建需要调试的存储过程
TYPE T_REC IS RECORD(ID INT, NAME VARCHAR(20));    --定义记录类型
TYPE ARR is ARRAY INT[4];                          --定义数组类型
R1 T_REC;                                           --声明记录类型
C1 CURSOR;                                         --声明游标类型
B ARR;                                             --声明数组类型

```

```

O MYCYCLE;                                --声明对象类型

I1 INT;                                    --声明整型

V1 VARCHAR(20);                            --声明字符串类型

BEGIN

R1.ID = 10012;

R1.NAME = 'WANGWU';

OPEN C1 FOR SELECT * FROM STUDENT;

LOOP

FETCH C1 INTO I1,V1;

EXIT WHEN C1%NOTFOUND;

END LOOP;

FOR I IN 1..4 LOOP

B[I] := I * 10;

PRINT B[I];

END LOOP;

O = NEW MYCYCLE(1, 2);

CALL P0;

END;

/

```

然后，按前面介绍的方法登录 dmdbg，使用 SQL 命令设置调试语句，如下：

```
DBG>SQL CALL P;
```

3. 设置断点

使用命令 B 设置调试断点，可以为指定的语句块设置执行中断位置。调试执行到断点设置位置后，将自动中断当前执行。

设置断点命令有两种不同格式：

```
B <方法名>
```

或

```
B [<方法名>:]<行号>
```

如果省略方法名，则为当前方法或语句块设置行号断点。包中的方法可以设置断点，但不能指定行号；如果断点设置在包上，则可以指定行号，但可以设置断点部分只能是该包体的初始化代码。

行号从 1 开始计数。对于方法来说，其开始的定义声明部分语句也包含在行号计数中。

对于断点的行号设置必须遵守以下原则：

- 定义声明部分不能设置断点；
- 如果一条语句跨多行，则断点应设置在语句最后一行；
- 如果多条语句在同一行，则执行中断在其中第一条语句执行前；
- 如果设置断点行号不能中断，则自动将其下移到第一个可以中断的行；
- 可以在调试过程中动态添加断点，如果在同一位置重复设置断点，则重复断点被忽略。

继续前面的例子，设置断点：

```
DBG>B 3                                --没有方法名，则在最顶层设置断点
No line 3 in method "@dbg_main". --由于顶层只有一行，无法设置断点，报错返回

DBG>B P:3                               --不能设置断点，则下移到第一条可以中断的行
Breakpoint 1 at SYSDBA.P, line 11@{R1.ID = 10012;}

DBG>B P:12                              --直接在指定行设置断点
Breakpoint 2 at SYSDBA.P, line 12@{R1.NAME = 'WANGWU';}
```

4. 显示断点

显示断点命令为：

```
INFO B
```

该命令显示所有已设置的断点。显示的每个断点包含以下信息：

- ✓ 序号：这是一个自增值，从 1 开始，设置断点时自动为其分配；
- ✓ 方法名：如果为语句块，则方法名为空；
- ✓ 行号。

继续前面的例子，显示断点：

```
DBG> INFO B

Num      Schema  Package  Method      What
```

```

1      SYSDBA  NULL    P      line 11@{R1.ID = 10012;}
2      SYSDBA  NULL    P      line 12@{R1.NAME = 'WANGWU';}

```

5. 取消断点

使用命令 D 取消断点，格式为：

```
D <断点序号>
```

断点序号可以通过 INFO B 命令获取，删除后断点的编号不会重用。取消断点命令可以在工具运行的任何时间调用。

继续前面的例子，取消断点：

```

DBG>D 1      --删除断点，成功

DBG>D 3      --删除不存在的断点，则报错
No breakpoint number 3.

DBG>INFO B   --显示断点信息

Num      Schema  Package Method  What
2        SYSDBA  NULL      P          line 12@{R1.NAME = 'WANGWU';}

```

6. 执行语句

执行语句命令格式为：

```
R
```

命令 R 执行设置的 SQL 语句。此命令无参数，执行 R 命令前必须已经设置过调试语句，且调试过程中命令 R 不允许重复使用。

如果设置了断点，则执行到断点指定的位置时中断，转入调试状态，并显示当前执行所在行的语句；否则，执行完成并显示结果。

继续前面的例子，执行语句：

```

DBG>R

Breakpoint 2, SYSDBA.P line: 12@{R1.NAME = 'WANGWU';}

DBG>R      --调试过程中不可以再次运行
The program is already running.

DBG>B P:23  --调试过程中可以设置断点

```

```
Breakpoint 3 at SYSDBA.P, line 23@{CALL P0;}
```

7. 显示脚本

使用命令 **L** 可以在 dmdbg 工具中显示指定方法的脚本。命令格式如下：

L <方法名>

方法名可以包含模式、包等前缀。在调试状态下，可以省略方法名，则显示当前正在运行的方法。此命令可以帮助用户准确设置断点位置。

L 命令每次显示 5 行代码，再次执行 **L** 命令时（不再需要带方法名），从已显示的一行开始显示。



说明：

命令 L 不能显示包的单个方法脚本。

继续前面的例子，显示脚本：

```
DBG>L SYSDBA.P
```

```
1      CREATE OR REPLACE PROCEDURE P AS
2      TYPE T_REC IS RECORD(ID INT, NAME VARCHAR(20));
3      TYPE ARR is ARRAY INT[4];
4      R1 T_REC;
5      C1 CURSOR;
```

```
DBG>L      --注意，第二次以后显示脚本不要带方法名，否则从第一行显示
```

```
6      B ARR;
7      O MYCYCLE;
8      I1 INT;
9      V1 VARCHAR(20);
10     BEGIN
```

8. 中断执行

中断执行命令为：

CTRL+C

在正常执行过程中，当监听到用户输入 CTRL+C，则 dmdbg 强制中断当前执行，转入调试状态。该命令对于调试死循环错误非常有效。

9. 继续执行

继续执行命令为：

C

在调试状态下，可以使用命令 C 继续 SQL 语句的执行，直到运行到断点或执行完成。

继续前面的例子，继续执行：

```
DBG>C
10
20
30
40
Breakpoint 3, SYSDBA.P line: 23@{CALL P0;}
```

10. 执行进入

执行进入命令为：

S

如果当前中断语句包含方法调用，则执行进入命令 S 可以将方法调用展开，执行到其语句块的第一条语句中断。如果当前行没有方法调用，则命令 S 与单步执行命令 N 等效。

继续前面的例子，执行进入：

```
DBG>S                                --执行进入存储过程 P0
SYSDBA.P0 line: 3      @ {PRINT 'HELLO';}
```



说明：

对于确定性函数，会直接进行结果重用，在重复调用时将不能执行进入调试。

11. 单步执行

单步执行命令为：

N

在调试状态下，单步执行命令 `N` 可以执行到当前层次下一个可中断的行。如果当前行包含多条语句，则所有该行语句都被执行。

继续前面的例子，单步执行：

```
DBG>N                                --单步执行存储过程 P0
SYSDBA.P0 line: 4      @ {PRINT 'WORLD';}
```

12. 执行跳出

执行跳出命令为：

```
F
或
FINISH
```

该命令可以执行当前函数的余下操作，如果剩余行有断点则执行到断点中断，否则返回到上层调用后中断。如果当前已经是最上层，则与单步执行命令 `N` 等效。

继续前面的例子，执行跳出：

```
DBG>F                                --跳出存储过程 P0
WORLD
SYSDBA.P line: 23      @ {CALL P0;}
```

13. 打印变量

打印变量命令的格式为：

```
P <变量名>
```

命令 `P` 可以打印指定的变量值。SQL 类型中，除了大字段数据，其它类型都可以打印。DM8SQL 数据类型往往为复杂数据类型，其显示格式如下：

- 结构类型

在大括号内显示结构所有成员的值，各成员值用 “,” 分隔。

- 数组类型

如果是数组，则可以与结构类型一样，依次显示数组所有成员的值，也可以只显示其中某一下标的值。

- 对象类型

可以打印对象中所有的静态成员，或者打印某一个静态成员，函数可以不打印。

- 游标类型

游标也是对象类型，其成员如表 8.2 所示：

表 8.2 游标对象成员

名称	数据类型	说明
ROWCOUNT	BIGINT	当前游标查询获取的总行数。如果游标未打开或查询失败时该值无效
FOUND	INT	最近一次 FETCH 操作是否取得数据，如果取得则为 1，否则为 0。如果游标未打开或查询失败时该值无效
ISOPEN	INT	游标是否打开。打开为 1，否则为 0
ROWCOUNT2	BIGINT	游标打开时确定的查询的结果集总行数，如果游标未打开或查询失败时该值无效

命令 P 只能打印当前方法所在栈帧中的变量，如果要打印其它层次方法栈帧的变量，首先需要移动到对应栈帧。打印变量的信息以\$开始，从 1 开始计数，每打印一个变量，自动增加 1。

继续之前的例子，重新开始执行，并打印变量：

```

DBG>C                                --结束之前的执行

0 rows affected

DBG>R                                --开始执行

Breakpoint 2, SYSDBA.P line: 12@{R1.NAME = 'WANGWU';}

DBG> P R1.ID                          --打印记录类型中单个字段

$1 = 10012

DBG> N                                ----单步执行后，R1 所有字段都被赋值

SYSDBA.P line: 13      @{OPEN C1 FOR SELECT * FROM STUDENT;}

DBG>P R1                             --打印记录类型 R1

$2 = {ID=10012, NAME=WANGWU}

DBG>C                                --继续运行，遇断点则中断

10

20

30

40

```

```

Breakpoint 3, SYSDBA.P line: 23@{CALL P0;}

DBG>P C1                                --打印游标的所有成员值

$3 = {ROWCOUNT=2, FOUND=0, ISOPEN=1, ROWCOUNT2=2}

DBG>P B                                  --打印数组

$4 = {10, 20, 30, 40}

DBG>P B[3]                              --数组下标以 1 开始

$5 = 30

DBG> P O.C1                             --打印对象类型中的成员

$6 = 2

DBG> P O                                 --打印对象类型

$7 = {ID=1, C1=2}

```

14. 显示堆栈

显示堆栈命令为:

```
BT
```

命令 BT 可以显示当前执行的堆栈状态。首先显示当前方法执行到的行数，然后从下至上显示各层方法名称及行号和调用参数值，输出参数不打印。

继续前面的例子，显示堆栈：

```

DBG>BT

#0      SYSDBA.P()      line: 23@{CALL P0;}      --#0 表示最下层
#1      @dbg_main      line: 1@{CALL P}      --表示栈帧只有两层

```

15. 下移栈帧

下移栈帧命令为:

```
DOWN
```

命令 DOWN 可以将当前调试栈帧下移一层。下移后，显示当前层执行到的语句行数。如果当前已经是堆栈最下层，则此命令无效。

继续前面的例子，下移栈帧：

```

DBG>DOWN                                --存储过程已到最下层，无法下移

```

```
Bottom (innermost) frame selected; you cannot go down.
```

16. 上移栈帧

上移栈帧命令为：

UP

命令 UP 可以将当前调试栈帧上移一层。上移后，显示当前层执行到的语句行数。如果当前已经是堆栈最上层，则此命令无效。

继续前面的例子，上移栈帧：

```
DBG>UP
#1      @dbg_main      line: 1@{CALL P;}

DBG>UP                                     --已经到最上层，无法上移
Initial frame selected; you cannot go up.

DBG> C                                     --执行结束，并显示结果
HELLO
WORLD

0 rows affected
```

17. 结束执行

结束执行命令为：

KILL

在调试过程中，用户可以使用命令 KILL 结束当前正在调试的 SQL 语句，并重新设置待调试的 SQL 语句。

KILL 命令结束执行与 R 命令执行结束的处理是不同的：

- ✓ KILL 之后，必须重新设置 SQL 语句，且断点信息被清空；
- ✓ R 返回结果结束执行之后，如果没有重新设置 SQL 语句，则再次运行的仍是之前设置的 SQL 语句，断点信息可以重用；如果重新设置了 SQL 语句，则断点信息亦被清空。

18. 退出工具

退出 dmdbg 工具的命令为：

QUIT

如果不希望继续调试，可以输入命令 QUIT。DMDBG 工具将中断连接，退出执行。

8.2 使用图形化客户端工具 Manager 调试 DMSQL 程序

DM 图形化客户端管理工具 Manager 也提供调试 DMSQL 程序的功能。在 WINDOWS 操作系统下的 DM8 数据库系统，可以通过选择“开始”-“程序”-“达梦数据库”-“客户端”，然后选择 DM 管理工具。或者在安装目录的“bin”子目录下可找到 manager.exe。

运行管理工具 Manager，其主界面如下图所示。

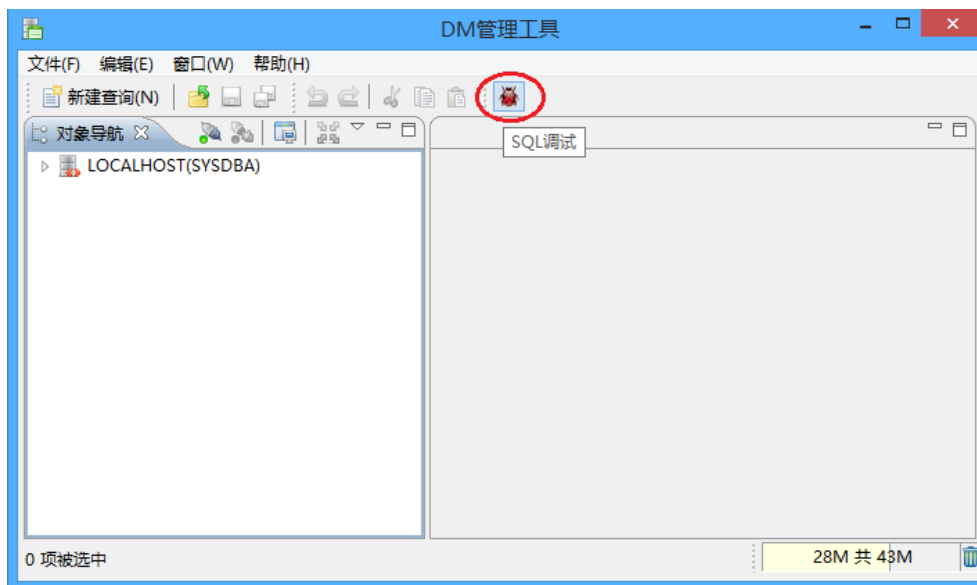



图 8.2 DM 管理工具主界面

点击  按钮进入 DMSQL 程序调试工具，首先需要登录数据库，如下图所示。

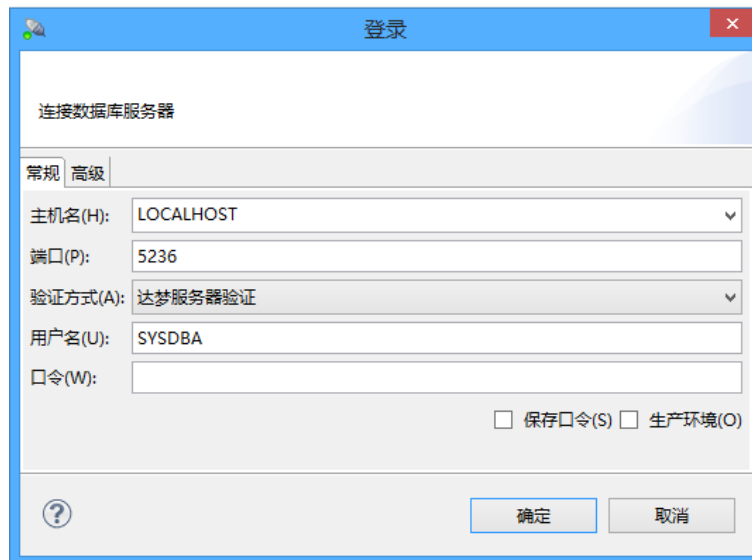


图 8.3 DMSQL 程序调试工具登录界面

成功登录数据库之后，出现 SQL 调试界面。然后点击  按钮，选择要调试的对象。

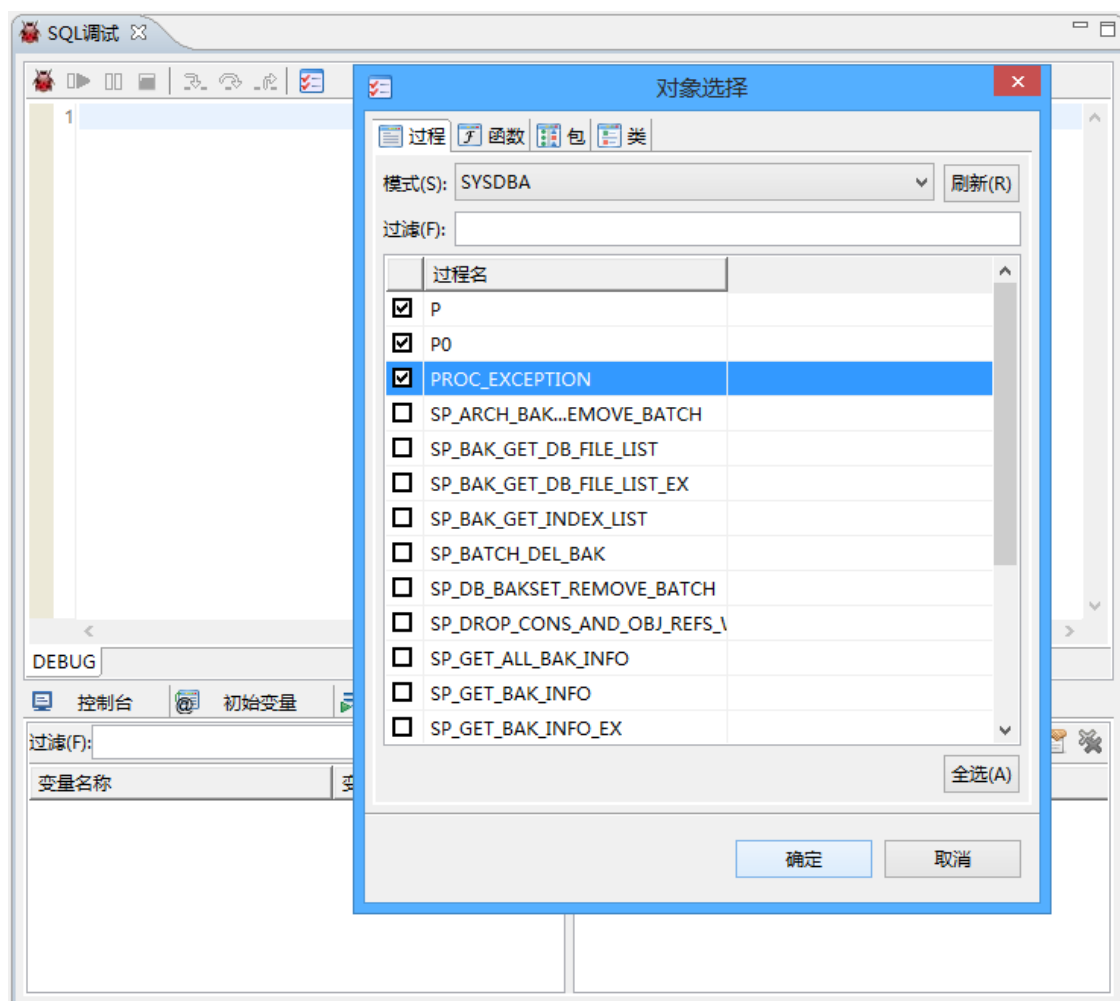


图 8.4 SQL 调试界面

之后，点击  按钮开始调试，进入调试状态之后  按钮变成灰色。

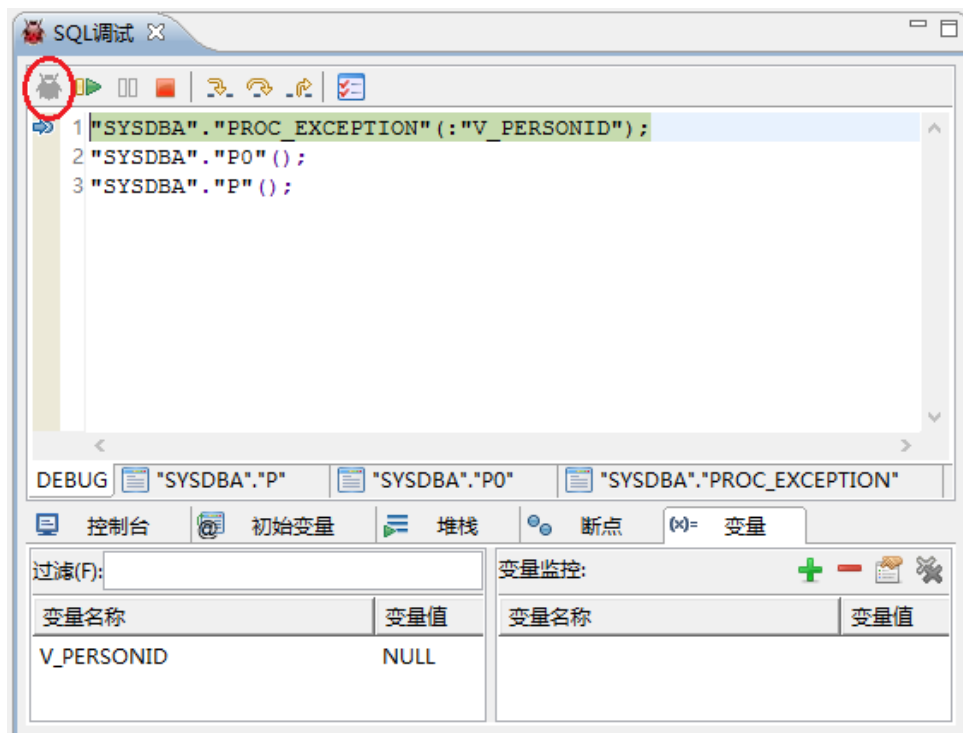


图 8.5 SQL 调试的调试状态

按钮 分别表示继续、暂停、停止； 分别表示进入 (F5)、下一步 (F6)、跳出 (F7)。例如，点击 按钮，就可以进入代码内部进行调试了。界面如下：

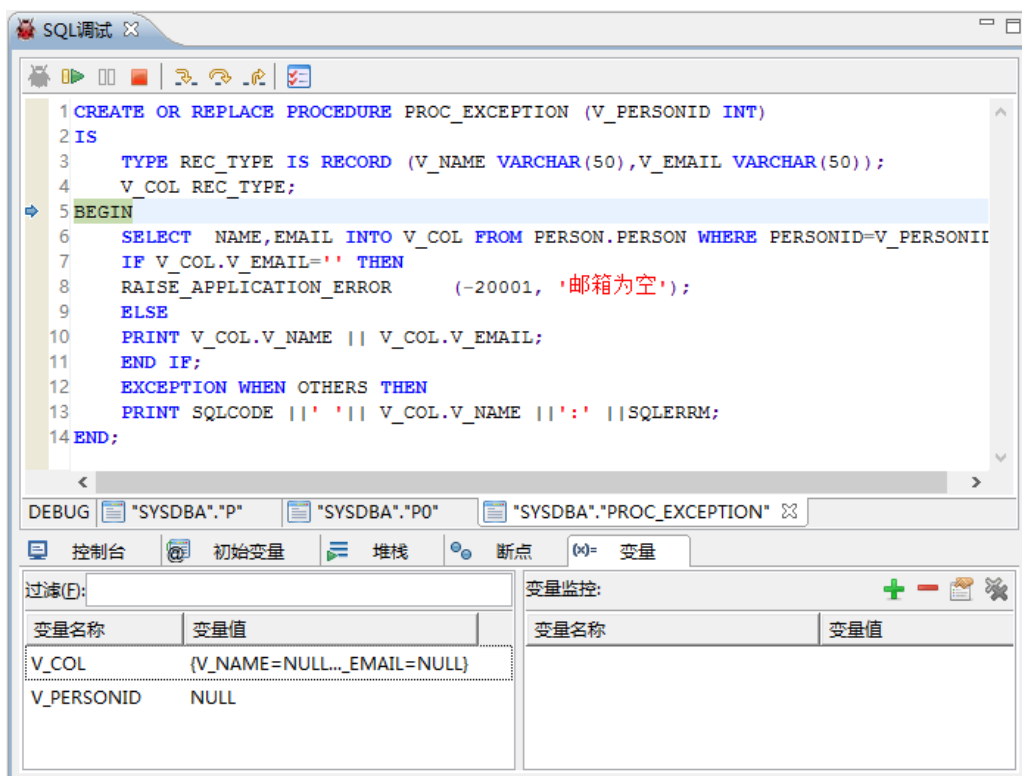


图 8.6 SQL 调试

咨询热线：400-991-6599

技术支持：dmtech@dameng.com

官网网址：www.dameng.com



武汉达梦数据库有限公司
Wuhan Dameng Database Co.,Ltd.

地址：武汉市东湖新技术开发区高新大道999号未来科技大厦C3栋16—19层

16th-19th Floor, Future Tech Building C3, No.999 Gaoxin Road, Donghu New Tech Development Zone,Wuhan,Hubei Province,China

电话：(+86) 027-87588000 传真：(+86) 027-87588810
