达梦数据库

达梦技术手册

DM8_PROC 使用手册

Service manual of DM8_PROC

前言

概述

本文档主要介绍 DM 对于 PRO*C 的支持，包括 DM 支持的嵌入式 SQL 语法、PRO*C 程序的编写以及 dpc_new 工具的使用等。

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

本章概要介绍 PRO*C 与嵌入式 SQL 的基本概念，以及 PRO*C 程序的工作机制即 DM 的预编译系统。

1.1 功能简介

SQL 语言作为结构化的查询语言，可以完成对数据库的定义、查询、更新、控制、维护、恢复、安全管理等一系列操作，充分体现了关系数据库的特征。但 SQL 语言是非过程性语言，本身没有过程性结构，大多数语句都是独立执行，与上下文无关，而绝大多数完整的应用都是过程性的，需要根据不同的条件来执行不同的任务。因此，单纯用 SQL 语言很难实现这样的应用。为此，DM 数据库提供了 SQL 的两种使用方式：一种是交互方式，另一种是嵌入方式。

嵌入方式是将 SQL 语言嵌入到高级语言中，这样一来，既发挥了高级语言数据类型丰富、处理方便灵活的优势，又以 SQL 语言弥补了高级语言难以描述数据库操作的不足，从而为用户提供了建立大型管理信息系统和处理复杂事务所需要的工作环境。在这种方式下使用的 SQL 语言称为嵌入式 SQL，而嵌入 SQL 的高级语言称为主语言或宿主语言。DM 数据库允许 C 作为嵌入方式的主语言。在 DM 系统中，我们将嵌有 SQL 语句的 C 语言程序称为 PRO*C 程序。

嵌入在主语言程序中的 SQL 语句并不能直接被主语言编译程序识别，必须对这些 SQL 语句进行预处理，将其翻译成主语言语句，生成由主语言语句组成的目标文件，然后再由编译程序编译成可执行文件，执行该文件，方可得到用户所需要的结果。

DM 的 PRO*C 嵌入工作方式支持的功能如下：

- ✓ 支持国家和军用标准关系数据库语言 SQL；
- ✓ 支持嵌入 SQL 语言的多模块程序设计；
- ✓ 提供对用户透明的查询优化功能；
- ✓ 支持对远程数据库的访问。

1.2 预编译系统的结构与功能

1.2.1 预编译系统的结构

预编译系统的结构如下图所示。

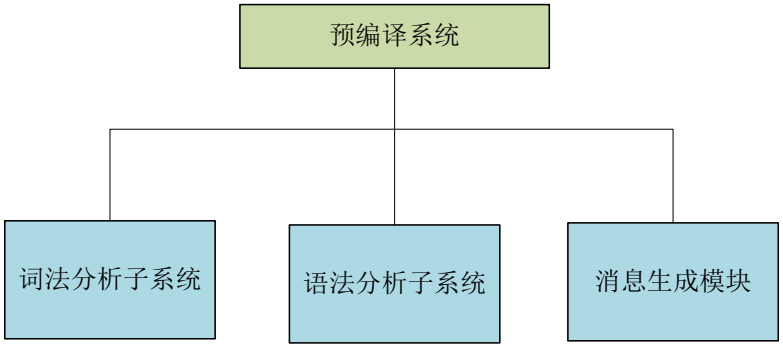

图 1.1 预编译系统的结构

1.2.2 预编译系统的功能

预编译系统主要包括词法分析、语法分析、消息生成等几个子系统。

- **词法分析子系统**

在嵌入工作方式下，词法分析子系统从指定的主语言源程序中逐段找出嵌入的 SQL 声明节或语句段，将它们进行分解，得到一个个单词，顺序填入单词表，供语法分析时使用；而对非 SQL 嵌入段的主语言正文则直接写入目标文件。在交互工作方式下，词法分析子系统只需要接受用户输入的单个 SQL 语句或 SQL 语句块，将它们分解成单词串并顺序填入单词表。

- **语法分析子系统**

语法分析子系统在词法分析的基础上，对单词表中所存入的 SQL 单词串按 SQL 语言文本进行初步的合法性检查，并对各种单词进行分类，识别语句中的嵌入变量并进行相应的语法检查。

- **消息生成模块**

在 DM 中，客户端对数据库的操作都是以消息方式传送给数据库服务器，因此预编译系统应具有将 SQL 语句转换为消息的功能，这就是消息生成。消息生成的任务是将 SQL 语句翻译成主语言消息生成语句和消息发送及回收语句，并将它们写入目标文件。需要说明的

是，预编译系统并不直接生成消息，而是利用对 DPI 接口的函数调用实现，这样可以进行必要的封装，模块性也更强。

1.2.3 预编译系统的处理流程

在嵌入工作方式下，预编译系统的功能是：对嵌入的 SQL 语句段和 SQL 声明节进行全面的词法、语法检查，然后将 SQL 语句翻译成主语言语句（即 DPI 函数调用）写入目标文件中，使目标文件成为一个纯由主语言组成的程序。整个预编译的处理流程如图所示：

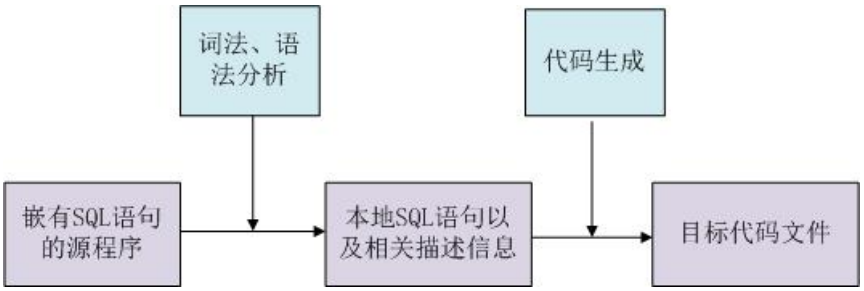

图 1.2 预编译系统的处理流程

1.3 预编译系统配置

1.3.1 预编译系统包含的程序和文件

预编译系统提供的软件有：

- 预编译命令行运行程序 dpc_new.exe；
- 编译时要使用的文件 dpc_dll.h、DPI.h、DPItypes.h、sqlca.h；若兼容 ORACLE，则需另外添加 sqlca_ora.h、sqlda_ora.h、dpc_ora_dll.h；若兼容 DB2，则需另外添加 sqlca_db2.h、sqlda_db2.h；
- 连接时需要的库文件 dmdpc.lib（Windows 操作系统）或者 libdmdpc.a（Linux）；
- 执行时需要的动态库文件 dmdpc.dll（Windows 操作系统）或者 libdmdpc.so（Linux）。

1.3.2 预编译命令的使用方法

预编译时，在命令提示符窗口中输入带参数的 dpc_new 命令，语法如下：

```
dpc_new parameter=value {parameter=value}
```

dpc_new 支持的参数含义如下表所示。

表 1.1 dpc_new 参数

参数	介绍
FILE	.pc 文件的完整路径，必须指定
TYPE	指定生成文件类型。C 代表 C 源文件，CPP 代表 C++源文件；默认为 C
MODE	编译模式。STD 表示标准编译模式；DM 表示达梦编译模式；ORACLE 表示兼容部分 ORACLE 语法的编译模式；DB2 表示兼容部分 DB2 语法的编译模式。默认 STD 编译模式
MACRO	解析宏，如果源 pc 文件中有#define 定义的宏，在文件中使用了宏，若不加此命令，文件中的宏将不能识别
DEFINE	条件编译宏，如果源 pc 文件中有#defWIN32 等条件编译宏，可以根据需要在 DEFINE 命令中设置对应的条件编译宏，来解析对应的内容
INCLUDE_DIR	包含的头文件目录，指定 pc 文件中 include 的头文件所在的目录
WRITE_DIRECT	解析的文件是否立即可见，缺省为 N。当为 N 时，dpc_new 工具解析 pc 文件时解析完部分就会立即写入 C 文件；当为 Y 时，整个 pc 文件解析完，才会一起写入生成的 C 文件
OCI_CONNECT	是否使用 OCI 连接，缺省为 N
INCLUDE_SQLCA	是否包含 sqlca，缺省为 N
CHAR_MAP	字符串的处理方式，目前只支持 STRING、CHARZ
HELP	打印帮助信息

例 1：

假设已有文件为 test.pc，则编译命令如下。

```
./dpc_new FILE=test.pc
```

如果编译成功将会生成 test.c 文件，否则 dpc_new 工具会报错。

例 2：

本例说明如何根据设置的 define 命令，来解析对应的内容。

假设原始文件 test.pc 内容如下：

```
#include <stdio.h>

#include <time.h>

#ifdef WIN32

#include <Windows.h>

#include <PROCESS.H>
```

```
typedef HANDLE     os_thread_t;
typedef  LPTHREAD_START_ROUTINE os_thread_fun_t;
#else
#include <pthread.h>
#endif
```

使用如下命令进行编译：

```
./dpc_new FILE=test.pc DEFINE=WIN32
```

则生成的 test.c 的内容为如下所示。可以看到，因为 DEFINE 参数指明了使用 WIN32 条件编译宏，因此编译结果中正确解析了源文件中“ifdef WIN32”部分。

```
#include <stdio.h>
#include <time.h>
#include <Windows.h>
#include <PROCESS.H>
typedef HANDLE     os_thread_t;
typedef  LPTHREAD_START_ROUTINE os_thread_fun_t;
```

若没有使用 DEFINE=WIN32 参数，使用如下命令进行编译

```
./dpc_new FILE=test.pc
```

则生成的 test.c 的内容为：

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
```

例 3：

本例说明 MACRO 参数的作用。

假设原始文件 test.pc 内容如下

```
#define BUFFERMAX  20971520   /* 缓存 1024*1024*20 */
unsigned char buffer[BUFFERMAX];
```

如果编译命令中不加 MACRO=Y，那么在嵌入式 SQL 语句中使用 buffer 变量就无法识别 BUFFERMAX。使用 MACRO=Y 后，生成的 C 文件将所有使用 BUFFERMAX 地方用 20971520 替代。上面的 pc 将解析为下面内容：

```
#include "dpc_dll.h"
```

```
/* Thread Safety */
typedef void * sql_context;
typedef void * SQL_CONTEXT;

unsigned char buffer [ 20971520 ] ;
```


注意：

如果指定了INCLUDE_DIR目录，则默认MACRO=Y

1.3.3 编译目标代码文件时的编译选项

_OciConnect：目标文件需要使用 dci.dll 时，包含头文件 dci.h，在编译时应该加上该选项。

DM64：如果在 64 位机器上编译，需要加上该选项。由于 dpc_new 在编译时需要使用 DPItypes.h，而在 DPItypes.h 中定义了 slength 和 ulength 类型，如下所示：

```
#ifdef DM64
typedef sdint8      slength;
typedef udint8      ulength;
#define SLENGTH_MAX SDINT8_MAX
#define ULENGTH_MAX UDINT8_MAX
#else
typedef sdint4      slength;
typedef udint4      ulength;
#define SLENGTH_MAX SDINT4_MAX
#define ULENGTH_MAX UDINT4_MAX
#endif
```

因此，如果 64 位环境下编译应用时没加 DM64 宏，会导致 DPI 接口调用异常。

2 预编译概念

本章主要介绍嵌入式 SQL 中的一些关键概念与术语，并简单介绍了开发 PRO*C 程序的步骤。本文档中从本章开始的所有示例，除了例子中特别建的表外，其余都使用 DM 示例库中的表。

2.1 嵌入式 SQL 关键概念

2.1.1 嵌入式 SQL 语句

嵌入式 SQL 是指在应用程序里直接嵌入 SQL 语句。因为嵌入 SQL，应用程序又叫宿主程序，编写应用程序的高级语言又叫宿主语言。例如，PRO*C/C++能够在 C 和 C++宿主程序嵌入一些 SQL 语句。

可嵌入的 SQL 语句包括 DDL 与 DML 语句，以及一些事务控制语句，这些语句都是嵌入的可执行语句。可执行语句在编译时会调用 libdmdpc.so 库接口，通过它们连接数据库，进行数据定义及查询操作数据库数据，以及处理事务，它们能存在 C/C++语言中任何可执行语句能嵌入的地方。表 2.1 列出了常用的嵌入可执行语句。

表 2.1 常用的嵌入可执行语句

可执行语句	作用
ALLOCATE	初始化变量
ALTER，CREATE TABLE	DDL
DELETE,UPDATE, INSERT,SELECT	DML
COMMIT,ROLLBACK,SAVEPOINT,SET TRANSACTION	事务控制
DESCRIBE，EXECUTE，PREPARE	动态 SQL

除了可执行语句，可嵌入 SQL 还包括一些在宿主程序与 DM 数据库之间传输数据的指令。这些指令在编译时不需要调用 libdmdpc.so 库接口，也不会操作数据库数据，用户可以使用它们声明通信区域和宿主变量。表 2.2 列出了常用的嵌入指令。

表 2.2 常用的嵌入指令

指令	作用

BEGIN DECLARE SECTION END DECLARE SECTION	声明宿主变量
INCLUDE	包含其他文件
TYPE	类型定义
WHENEVER	捕获运行错误

2.1.2 嵌入式 SQL 语法

在 PRO*C 程序中能够自由地使用 SQL 语句与 C 语句，也可以在 SQL 语句中使用 C 变量和结构。在 PRO*C 中使用 SQL 语句必须在 SQL 语句前加上关键字“EXEC SQL”并且以分号结束。DM 的预编译命令行运行程序 dpc_new 将所有带有“EXEC SQL”的语句转换为对 libdmdpc.so 库中接口的调用。

许多嵌入式 SQL 语句与交互式 SQL 语句的区别只在于能够使用程序变量或者是增加了额外的“EXEC SQL”关键字，下面的例子比较了交互式与嵌入式 COMMIT 语句的区别，他们的作用是一样的。

```
COMMIT;            --交互式
EXEC SQL COMMIT;    --嵌入式
```

2.1.3 静态 SQL 与动态 SQL

大多数程序使用静态的 SQL 处理固定的语句，在这种情况下，你必须在运行前事先知道 SQL 语句和事务的组成，以及哪些 SQL 命令会被执行，哪些表的结构会变动，哪些列会被更新等等。

然而，在某些情况下可能要在运行时才能确定要执行的有效的 SQL 语句，因此你在运行前可能不知道所有的 SQL 命令、数据库表、牵涉到的列等，只有在程序执行时才能构造出完整的 SQL。动态 SQL 是一种高级编程技术，能够让程序在运行时处理执行过程中临时生成的 SQL 语句。

2.1.4 嵌入 PL/SQL 块

PRO*C 把 PL/SQL 语句块视作单一的嵌入 SQL 语句，任何 SQL 语句能嵌入的地方也能嵌入 PL/SQL 块。在宿主程序中嵌入 PL/SQL 块，必须使用关键字 EXEC SQL EXECUTE 和 ENDEXEC 将 PL/SQL 语句块括起来。

2.1.5 宿主变量与指示符

宿主变量是数据库与应用程序联系的关键，宿主变量是在 C 中声明的 SQL 与 C 程序都能使用的变量。程序使用输入宿主变量向数据库传递数据，数据库使用输出宿主变量传输数据或其他状态信息给应用程序。宿主变量能够在任何 SQL 表达式可以使用的地方使用。在 SQL 语句中，宿主变量必须以“:”作为前缀以便于同关键字区分开来。

应用程序中也能用结构来包含多个宿主变量，在 SQL 语句中使用结构时也要用“:”作为前缀，DM 会把结构里的每个成员都当宿主变量处理。

每个宿主变量后都能跟一个指示符变量。指示符变量是一个整型的宿主变量，其作用是用来指示宿主变量的取值情况。在 SQL 语句中，指示符变量必须加上“:”前缀并且只能紧跟在它指示的宿主变量后面。可以在宿主变量与指示变量中间加上关键字 INDICATOR 使含义更明确，也可以省略。

如果宿主变量是定义在结构体中，你只需要定义一个指示符的结构，里面的指示变量与宿主结构中的宿主变量一一对应，再在 SQL 语句使用指示结构即可。与非结构指示变量一样，也需要紧跟在宿主结构之后，且前缀加上“:”，也可在中间加上关键字 INDICATOR。

2.1.6 DM 数据类型

一般来说，应用程序输入数据到数据库，数据库输出数据到程序。数据库将输入的数据存储到到表中，输出数据存放到宿主变量里。

存储数据必须知道数据类型，以确定存储格式。数据在 DM 数据库表中的存储格式为 DM 内部数据类型；数据在宿主变量中的存储格式为外部数据类型。DM 数据库能同时识别这两种数据类型，且在二者间进行转换。

2.1.7 宿主数组

PRO*C 允许定义数组宿主变量和数组结构宿主变量，并且在单个 SQL 中使用。使用数组可以同时操作大量数据，也可以在结构中使用数组宿主变量。

2.1.8 事务

PRO*C 中嵌入的 SQL 也适用交互式 SQL 中事务的概念，可以通过 COMMIT、ROLLBACK、SAVEPOINT 等语句对事务进行控制。

2.1.9 错误与警告

当嵌入式 SQL 执行出错或产生警告时，PRO*C 提供了三种方法来捕获错误与警告：SQLCA 结构、WHENEVER 语句、SQLCODE 与 SQLSTATE。具体使用方法见后面章节介绍。

2.2 开发嵌入式程序的步骤

举例说明嵌有 SQL 语句的 C 语言程序 PRO*C 的运行过程：

1) 编写嵌有 SQL 语句的源程序 test.pc；
2) 使用 DM 预编译工具 dpc_new 编译 test.pc，生成 test.c；
3) C 编译器编译链接上述文件，以及 dpc_dll.h、DPI.h、DPItypes.h、sqlca.h、dmdpc.lib、dmdpc.dll，生成 test.exe 可执行程序。若在 Linux 下则将 dmdpc.lib、dmdpc.dll 改为 libdmdpc.a 和 libdmdpc.so。

2.3 程序编写

本节简单介绍一些嵌入式程序编写的语法、惯例、限制等，具体在后续章节中会有详细介绍。

■ **程序注释**

可以在嵌入的 SQL 语句中使用 C 类型注释（/*…*/），也可以在一行的结尾使用 ANSI 标准的注释（//…）。

例如：

```
EXEC SQL SELECT NAME,/*名字*/ SEX
INTO :name, :sex  -- output host variables
FROM PERSON.PERSON
WHERE PERSONID = :personid;
```

■ 声明节

一个声明节的组成如下例所示：

```
EXEC SQL BEGIN DECLARE SECTION;
/* 声明宿主变量在这里  */
char username[20],password[20],servername[20];
    ...
EXEC SQL END DECLARE SECTION;
```

声明节以“EXEC SQL BEGIN DECLARE SECTION;”语句开始，以“EXEC SQL END DECLARE SECTION;”结束。在声明节中能够存在的内容包括有以下语句：

- ✓ 宿主变量与指示符变量的定义
- ✓ 非宿主的 C 变量
- ✓ C 的注释
- ✓ EXEC SQL TYPE 语句

■ 引号

C 使用单引号表示单个字符，SQL 使用单引号界定字符串，例如：

```
EXEC SQL SELECT NAME, SEX FROM EMP WHERE PHONE = '123456789';
```

C 使用双引号界定字符串，SQL 使用双引号界定包含小写或特殊字符的标识符，例如：

```
EXEC SQL CREATE TABLE "test"(c1 INT);
```

■ 行延长

在 SQL 语句中能够使用反斜杠延长一行字符串到另一行，例如：

```
EXEC SQL INSERT INTO PERSON.PERSON (NAME, PHONE) VALUES ('XQ', '123\
456');
```

3 嵌入式程序的组成

3.1 一个简单的嵌入式程序结构分析

首先，从一个简单的程序入手来认识嵌入式程序的组成和基本语法。

例如，下面是一个名为 test.pc 的简单的嵌入式程序。这个程序查询并输出“人员信息”表中人员编号为“1”的人员姓名和联系电话。

```
/* test.pc */
#include <stdio.h>
/*宿主变量的定义 */
EXEC SQL BEGIN DECLARE SECTION;
 char username[20],password[20],servername[20];
varchar person_name[50];
varchar person_phone[25];
EXEC SQL END DECLARE SECTION;


void main(void)
{
 printf(" Please input username:");
 scanf("%s",username);
 printf(" Please input password :");
 scanf("%s",password);
 printf(" Please input servername :");
 scanf("%s",servername);
/*登录数据库*/
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
/*对数据库操作 */
EXEC SQL SELECT name, phone
INTO :person_name,:person_phone FROM person.person
```

```
WHERE personid = '1';
printf("\n 对应的人员姓名为： %s\n", person_name);
printf("\n 对应的联系电话为： %s\n", person_phone);
/*退出数据库*/
EXEC SQL LOGOUT;
}
```

一般说来，pc 文件由以下几个部分构成：

1. 宿主变量定义

在上面的例子中，宿主变量定义为嵌入在如下语句中的部分：

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE SECTION;
```

2. 登录数据库

```
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
```

3. 数据库操作

登录成功之后，就可以对数据库进行各种操作。

4. 退出登录

```
EXEC SQL LOGOUT;
```

下面将对以上各部分加以详细说明。

3.2 宿主变量的定义

嵌入的 SQL 语句可以使用主语言的变量来输入/输出数据。例如在 SELECT 语句中，需要将查找到的数据通过 INTO 子句送到某些变量中，或者 WHERE 子句需要用某些变量的值作条件。我们称 SQL 语句中使用到的变量为宿主变量。在 SQL 中所使用的宿主变量均必须在嵌入的 SQL 声明节中加以说明。

3.2.1 声明节语句

■ **声明节开始语句**

语法：

```
EXEC SQL BEGIN DECLARE SECTION;
```

功能：表示声明节的开始。

使用说明：后面必须有一个声明节结束语句与之相对应。

■ **声明节结束语句**

语法：

```
EXEC SQL END DECLARE SECTION;
```

功能：表示声明节的结束。

使用说明：前面必须有一个声明节开始语句与之相对应。

■ **使用声明节应注意的问题**

在声明节开始语句与结束语句中间，采用 C 语言定义变量的方式定义宿主变量。在声明节中定义的变量可以被 C 语句使用，而不必在声明节之外重新定义这些变量，否则会引起变量重复定义的错误。

一个程序模块中可以出现多个声明节，各个声明节可以出现在 C 程序的说明语句可出现的位置上。PRO*C 规定，一个模块中所有的声明节中的宿主变量不得同名，而不论其实质是全局变量还是局部变量。预编译时，相应声明节的 C 语句是去掉了“EXECSQL BEGIN DECLARE SECTION”和“EXEC SQL END DECLARE SECTION”两个语句后的变量定义语句，其位置顺序均保持不变。

3.2.2 常规数据类型变量的定义

在声明节中，可以使用的常规数据类型包括：char、short、int、long、float、double 等 C 数据类型。定义这些类型变量的方法与 C 语言定义相应变量的方法基本上是相同的。由于 DM 采用的是一次一个元组的处理方式，因而限制了数组的定义和使用，即

不能使用数组的某个元素。作为一种特例，在声明节中定义字符数组，作为字符串指针使用。如： char name[20]等价于 char *name ，其中 name 是指向长度为 20 个字节空间的字符指针（等价于指针变量）。

在定义常规类型的变量时，与 C 语言一样，可以加上存储类型修饰符，包括 extern、static 和 auto。

定义语句的语法如下：

```
<宿主变量定义>::=<常规数据类型变量定义>

<常规数据类型变量定义>::=[<存储类型>] <C 类型说明><变量声明>[{,<变量声明>}...]

<存储类型>::=extern|static|auto

<C 类型说明>::= char | short | int | long | float | double 等

<变量声明>::= <变量名说明> [= <常量表达式>]

<变量名说明>::=<变量名> | *<变量名> | <变量名>[无符号整数]
```

例如：

常规数据类型变量定义。

```
......
EXEC SQL BEGIN DECLARE SECTION;
char shop_no[20],shop_name[20];
short flag;
EXEC SQL END DECLARE SECTION;
......
```

预编译后，产生的结果为：

```
char shop_no[20],shop_name[20];
short flag;
......
```

3.2.3 宿主变量的使用

3.2.3.1 常规使用方法

在 SQL 语句中引用的宿主变量都应在声明节中定义，而且定义应先于引用。

在 SQL 语句中使用宿主变量时，在引用的宿主变量名前必须加上并且只能加上冒号“:”，而不论说明该变量时变量名前有无“*”，或者是否采用数组形式。在 C 语句中使用宿主变量时，不能够加上“:”。

例如：

查询人员编号为“1”的人员姓名及联系电话。

```
EXEC SQL BEGIN DECLARE SECTION;
varchar person_name[50];
varchar person_phone[25];
varchar person_id[10];
EXEC SQL END DECLARE SECTION;
......
strcpy(person_id,"1");
EXEC SQL SELECT name, phone
INTO :person_name, :person_phone FROM person.person
WHERE personid =:person_id;
printf("%s %s ", person_name, person_phone);
```

执行结果：

```
李丽 02788548562
```

3.2.3.2 指示符的使用

在 SQL 语句中，在一个宿主变量之后，可以再跟一个数据类型为整型 (short，long，int) 的被称为指示变量的宿主变量，其作用是指明该变量之前的一个宿主变量的取值情况。具体地说，它有如下两个作用，分别举例说明：

- ✓ **指示 SQL 语句中输入变量的值是否为空值。**

例如：

使用指示变量插入空值。利用预置指示变量的值为-1，表明插入一个空值。

```
EXEC SQL BEGIN DECLARE SECTION;
varchar person_name[50];
varchar person_phone[25];
varchar person_id[10];
short  value_indicator;
EXEC SQL END DECLARE SECTION;
......
strcpy(person_name,"李丽");
person_phone ='02788548562';
value_indicator=-1;
EXEC SQL INSERT INTO person.person (name, phone)
VALUES(:person_name,:person_phone INDICATOR :value_indicator);
/*该语句等价于：
EXEC SQL INSERT INTO person.person (name, phone)
VALUES('李丽', NULL);  */
```

在前一种插入语句中，person_phone 带指示变量 value_indicator，由于 value_indicator 的值为-1，尽管 person_phone 的值为 02788548562，系统仍不使用该值作插入值，而认为 person_phone 的插入值为空。如果希望 person_phone 的插入值为 02788548562，只需改变 value_indicator 的值为 0 即可。

在后一个插入语句中，person_phone 未使用指示变量，它的插入值在插入语句中固定为 NULL，如果希望 person_phone 的插入值为 02788548562，则要修改该插入语句。因此，前一种使用指示变量的方法比后一种方法灵活，同时也给用户编程带来方便。

✓ **指示执行 SQL 语句后，返回结果是否为空值或有截取等情况。**

在 INTO 子句中，使用指示变量表明该 SQL 语句执行时，对被指示的宿主变量的赋值情况。

一个指示变量的取值有三种：取值为 0，说明返回值非空且赋给宿主变量时不发生截舍；取值为-1，说明返回的值为空值；取值>0，说明返回的值为非空值，但在赋给宿主变量时，字符串的值被截舍，这时指示变量的值为截断之前的长度。

3.2.3.3 使用宿主变量应注意的问题

宿主变量的命名是有大小写区别的。字符序列相同，大小写不同的变量名代表不同的变量。

宿主变量的命名不应是 C 语句的关键字，以免产生误用或错误。使用宿主变量时应注意：SQL 语句引用一个宿主变量时，宿主变量名前加一个冒号；C 语句中使用宿主变量时，不加冒号；引用宿主变量时，可以使用一个相关的指示变量。

另外，由于 C 语言所允许的数据类型与 SQL 语言所采用的数据类型有一定的差别，因而在输入输出数据时，在两种数据类型之间要做一定的转换工作。DM 支持 BYTE、NUMERIC 与 short、int、float、double 之间的转换，但应注意数据转换时产生的溢出问题。

3.2.4 VARCHAR 宿主变量的使用

可以使用 VARCHAR 类型声明可变长度的字符串，如果需要处理 VARCHAR 类型列的数据的输入输出，使用 VARCHAR 类型变量比使用 C 字符串类型会更加方便。“VARCHAR”可以是大写也可以是小写，但是不能大小写混用。

VARCHAR 类型经过预编译被转换为 C 的结构。

例如：

```
VARCHAR   username[20];
```

经过预编译后转换成：

```
struct
{
    unsigned short  len;
    unsigned char   arr[20];
} username;
```

其中 len 是数据的实际长度，arr 存放的是字符串的数据。在 SQL 语句中使用 VARCHAR 宿主变量与使用 C 类型宿主变量一样。

例如：

```
EXEC SQL SELECT NAME INTO :username FROM PERSON.PERSON;
```

3.2.5 游标变量的使用

游标在 PRO*C 中作为一种对象变量使用，具体的使用方法见第 6.4 节介绍。

3.2.6 CONTEXT 变量

运行上下文 CONTEXT 变量实际上是一个句柄，指向客户端内存的一段区域，对应一个 DM8 数据库连接，包含 0 个或多个游标的状态与信息等。

使用 CONTEXT 变量一般包括以下几个步骤：

1. 使用 sql_context 定义上下文宿主变量

例如：

```
sql_context my_context ;
```

2. 使用 ALLOCATE 语句初始化 context 变量

例如：

```
EXEC SQL CONTEXT ALLOCATE :my_context ;
```

3. 使用 context 变量，context 使用后直到下一个 CONTEXT USE 才会切换到新的上下文。

例如：

```
EXEC SQL CONTEXT USE :my_context ;
```

4. 使用 FREE 语句释放 context 变量

例如：

```
EXEC SQL CONTEXT FREE :my_context ;
```

CONTEXT 变量的具体使用可参看后续“多线程支持”章节。

3.2.7 结构宿主变量

可以在嵌入式程序中定义 C 结构宿主变量，然后在 SQL 语句中引用结构变量，需要注意结构中的成员必须与查询或插入 SQL 语句中表达式的顺序一致。

例 1：使用结构进行数据插入。

```
typedef struct
{
```

```
charsex[2];
    char   name[51];
    char   email[51];
    char phone[26];
} person_record;
person_record new_person;

/* 为new_person 赋值 */
......
EXEC SQL INSERT INTO  PERSON.PERSON(SEX, NAME, EMAIL, PHONE)
    VALUES (:new_person);
```

结构中的成员也可以是数组，这样用来批量执行，可同时操作多行。

例 2：一次向 PERSON 表插入三行数据。

```
typedef struct
{
    char  sex[3][2];
    char   name[3][51];
    char   email[3][51];
    char phone[3][26];
} person_record;
person_record new_person;

/* 为new_person 赋值 */
......
EXEC SQL INSERT INTO  PERSON.PERSON(SEX, NAME, EMAIL, PHONE)
    VALUES (:new_person);
```

当需要使用指示符变量时，由于宿主变量都包含在结构里，所以必须再定义一个结构包含各个宿主变量对应的指示符变量，并且成员顺序必须与宿主变量的结构一致。

例 3：

```
struct
```

```
{
    short sex_ind;
    short name_ind;
    short email_ind;
    short phone_ind;
} person_record_ind;

/* 为person_record_ind 赋值 */
......
EXEC SQL INSERT INTO  PERSON.PERSON(SEX, NAME, EMAIL, PHONE)
VALUES (:person_record_ind);
```

例 4：一个完整的使用结构的示例。

```
/*
 *  This program connects to DM, declares and opens a cursor,
 *  fetches the name, email, and phone of all
 *  people, displays the results, then closes the cursor.
 */
#include <stdio.h>

#define UNAME_LEN      20
#define PWD_LEN        40
EXEC SQL INCLUDE SQLCA;

typedef struct
{
    char     name[50];
    char     email[50];
    char     phone[50];
}person_info;
person_info person_rec_ptr;
```

```
EXEC SQL BEGIN DECLARE SECTION;
char     username[50];
char     password[50];
char     servername[50];
EXEC SQL END DECLARE SECTION;

/* Declare function to handle unrecoverable errors. */
void sql_error();

main()
{
/* Connect to DM. */
    strcpy(username, "SYSDBA");
    strcpy(password, "SYSDBA");
    strcpy(servername, "192.168.0.89:5289");
    EXEC SQL WHENEVER SQLERROR DO sql_error("DM error--");
    EXEC SQL CONNECT :username IDENTIFIED BY :password USING :servername;
    printf("\nConnected to dm as user: %s\n", username);

/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. 'salespeople' is a SQL identifier,
 * not a (C) host variable.
 */
    EXEC SQL DECLARE salespeople CURSOR FOR
        SELECT NAME, EMAIL, PHONE FROM PERSON;

/* Open the cursor. */
    EXEC SQL OPEN salespeople;

/* Get ready to print results. */
```

```
    printf("\n\nThe company's salespeople are--\n\n");
    printf("Salesperson   EMAIL   PHONE\n");
    printf("-----------   ------   ----------\n");

/* Loop, fetching all salesperson's statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
 */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        EXEC SQL FETCH salespeople INTO :person_rec_ptr;
        printf("%s %s %s\n", person_rec_ptr.name,
                person_rec_ptr.email, person_rec_ptr.phone);
    }

/* Close the cursor. */
    EXEC SQL CLOSE salespeople;
    printf("\nArrivederci.\n\n");
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

void
sql_error(msg)
char *msg;
{
    char err_msg[512];
size_t buf_len, msg_len;
```

```
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

3.2.8 指针变量

也可以在嵌入式 SQL 中使用指针类型的变量。

例如：

```
char  *int_ptr;
EXEC SQL SELECT NAME INTO :int_ptr FROM PERSON.PERSON;
```

3.3 可执行的 SQL 语句

在 C 程序中嵌入的可执行 SQL 语句包括普通的 SQL 语句和数据库登录/退出语句，其中数据库登录/退出不是 SQL 标准语句，为嵌入式程序扩展的可执行 SQL 语句。

3.3.1 数据库登录语句

数据库登录语句的语法如下：

```
EXEC SQL LOGIN :username PASSWORD :pwd SERVER :servername;
```

```
或 EXEC SQL CONNECT :username IDENTIFIED BY :pwd [USING :servername];
或 EXEC SQL CONNECT :conninfo;
```

功能：用于数据库的登录。

使用说明：

- 语句中 username 是长度最多为 128 个字符的指针，其值为注册用户的名字；
- pwd 表示相应的用户口令，口令的长度最多为 128 个字符；
- servername 为服务器的名字， 名字的长度最多为 128 个字符；
- servername 能带有端口号，例如“192.168.0.89:5236”，conninfo 包含用户名密码以及 IP 和端口号，例如 SYSDBA/SYSDBA@192.168.0.89:5236。

在用户程序中，登录语句的逻辑位置必须先于所有可执行的 SQL语句，只有执行登录语句之后，才能执行其它可执行的 SQL语句。如果登录失败，则终止程序的运行。

3.3.2 数据库退出语句

数据库退出语句的语法如下：

```
EXEC SQL LOGOUT;
或 EXEC SQL COMMIT RELEASE;
```

功能：断开与数据库的连接。

使用说明：在一个程序运行结束后，最好有一个 LOGOUT 语句，用于切断客户程序与 DM 数据库服务器之间的联系，有利于服务器释放相应的空间。在执行 LOGOUT 语句时，隐含有对最后一个事务的提交。

3.3.3 普通 SQL 语句

能在 C 程序中嵌入的可执行普通 SQL 语句包括建表、建索引等 DDL 语句；SELECT、INSERT、DELETE、UPDATE 等 DML 语句；RETURNING 语句和游标操作语句等。

■ SELECT 语句

查询是最常用的 SQL 操作之一，可以将从数据库中检索到的值赋给相应的目标变量。

常用语法格式如下，详细的 SELECT 语句语法介绍请参考《DM8_SQL 语言使用手册》。

```
EXEC SQL SELECT [ALL|DISTINCT]<选择清单> INTO<选择目标清单> FROM <表引用> [,<表引用>...] [WHERE <搜索条件>][<GROUP BY 子句>][<HAVING 子句>];
```

例如下面的语句对 PERSON.PERSON 表进行查询。

```
EXEC SQL SELECT NAME,PHONE INTO :name,:phone FROM PERSON.PERSON WHERE PERSONID=:personid;
```

查询语句的 INTO、WHERE、CONNECT BY、START WITH、GROUP BY、HAVING、ORDER BY、FOR UPDATE 等语法在嵌入 SQL 中也同样支持。

■ INSERT 语句

使用 INSERT 语句向数据库插入数据，例如下面的语句通过宿主变量向 PERSON.PERSON 表插入数据。

```
EXEC SQL INSERT INTO PERSON.PERSON(NAME, PHONE) VALUES(:name, :phone);
```

■ DELETE 语句

使用 DELETE 语句删除数据库表中的数据，例如下面的语句通过宿主变量删除 PERSON.PERSON 表中的特定数据。

```
EXEC SQL DELETE FROM PERSON.PERSON WHERE PERSONID=:personid;
```

■ UPDATE 语句

使用 UPDATE 语句更新数据库表中指定列的值，例如下面的语句通过宿主变量更新 PERSON.PERSON 表中指定行的 PHONE 与 EMAIL 列的值。

```
EXEC SQL UPDATE PERSON.PERSON SET PHONE=:phone,EMAIL=:email WHERE PERSONID=:PERSONID;
```

■ DML RETURNING 语句

INSERT、DELETE、UPDATE 语句后面可以跟 RETURNING 子句，RETURNING 子句的语法如下：

```
<RETURNING | RETURN><expr {,expr}>
```

```
INTO <:hv [[INDICATOR]:iv] {, :hv [[INDICATOR]:iv]}>
```

其中 hv 表示宿主变量，iv 表示指示符变量。


注意：

RETURN后的表达式与INTO后的宿主变量数必须一致

■ **DDL 语句**

可使用 DDL 语句进行数据定义，包括创建用户、创建表、修改表、创建索引等等。

例如下面的语句创建表 T。

```
EXEC SQL CREATE TABLE T(C1 INT, C2 INT);
```

3.3.4 游标语句

使用游标一般包括四个步骤：

1. 声明游标：DECLARE CURSOR
2. 打开游标：OPEN CURSOR
3. 使用游标获取数据：FETCH
4. 关闭游标：CLOSE CURSOR

使用游标必须先声明，声明游标实际上是定义了一个游标工作区，并给该工作区分配了一个指定名字的指针(即游标)。游标工作区用以存放满足查询条件行的集合，因此它是一说明性语句，是不可执行的。在打开游标时，就可从指定的基表中取出所有满足查询条件的行送入游标工作区并根据需要分组排序，同时将游标置于第一行的前面以备读出该工作区中的数据。当对行集合操作结束后，应关闭游标，释放与游标有关的资源。

3.3.4.1 声明游标语句

声明一个游标，给它一个名称，同时也可定义与之相联系的 SELECT 语句。

语法格式为：

```
DECLARE 游标名 CURSOR [WITH HOLD] [FAST | NOFAST]  FOR < <SELECT 语句>|<SQL 语句名> >;
```

参数说明：

- 游标名：指明声明的游标的名称；
- <SELECT 语句>： 被称为查询说明，指明对应于被定义的游标的 SELECT 语句，不能包括 INTO 子句；
- <SQL 语句名>： PREPARE 语句的语句名。详细请参考 6.6.1.1 动态 SQL 语句的表示方法 。

使用说明：

- 用户必须在其他的嵌入式 SQL 语句引用该游标之前定义它。游标说明的范围在整个预编译单元内，并且每一个游标的名字必须在此范围内唯一；
- Oracle 兼容模式下默认提交时不关闭游标，非 Oracle 兼容模式下默认提交时关闭游标。WITH HOLD 游标与兼容模式无关，提交时都不关闭；
- FAST 属性指定游标是否为快速游标。缺省时默认为 NO FAST 对应的普通游标。若定义游标时设置 FAST 属性将游标定义为快速游标，该游标在执行过程中会提前返回结果集，速度上提升明显，但是存在以下的使用约束：
  a) 使用快速游标的 PLSQL 语句块中不能修改快速游标所涉及的表；
  b) 禁止将快速游标赋值给其它游标对象；
  c) 快速游标上不能创建引用游标；
  d) 不支持快速游标更新和删除；
  e) 快速游标不支持 next 以外的 fetch 方向。

例如：声明一个游标 cheap_book。

```
EXEC SQL DECLARE  cheap_book CURSOR FOR
   SELECT  NAME, AUTHOR, PUBLISHER, NOWPRICE
   FROM   PRODUCTION.PRODUCT
   WHERE  NOWPRICE < 15.0000;
```

3.3.4.2 打开游标语句

打开一个已经声明过的游标。

语法格式为：

```
格式一：OPEN <游标名>
格式二：OPEN <游标变量> FOR <查询表达式> | <表达式子句>;
```

```
<表达式子句>::=<表达式> [USING <绑定参数> {, <绑定参数>}]
```

格式一对应声明游标语句的格式一

格式二对应声明游标语句的格式二

参数说明：

- 游标名：指明被打开的游标的名称

例如，打开游标 cheap_book。

```
EXEC SQL OPEN cheap_book;
```

3.3.4.3 拨动游标语句

使游标移动到指定的一行，若游标名后跟随有 INTO 子句，则将游标当前指示行的内容取出分别送入 INTO 后的各变量中。

语法格式为：

```
FETCH [[NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n] [FROM] ] <游标名>
[INTO <赋值对象>{,<赋值对象>>}];
```

参数说明：

- 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。

使用说明：

- 当游标被打开后，若不指定游标的移动位置，第一次执行 FETCH 语句时，游标下移，指向工作区中的第一行，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 后的变量个数、类型必须与定义游标语句中、SELECT 后各值表达式的个数、类型一一对应。

例 1，对于上一节中打开的游标，若连续执行以下语句两次：

```
EXEC SQL FETCH  cheap_book  INTO  :a, :b, :c, :d;
```

由于是两次连续执行 FETCH 语句且结果均放在变量:a、:b、:c、:d 中，这样，第一次的结果已被第二次的结果冲掉，最后的结果为：

a='老人与海'；b='海明威'；c='上海出版社'；d=6.1000

该例说明：当需要连续取出工作区的多行数据时，应将 FETCH 语句置入高级语言的循环结构中。

例 2，当设置 ABSOLUTE 属性时，n 值是从 1 开始的。如执行如下语句：

```
EXEC SQL FETCH ABSOLUTE 3 cheap_book INTO :a, :b, :c, :d;
```

结果为：

a='突破英文基础词汇'; b='刘毅'; c='外语教学与研究出版社'；d=11.1000

3.3.4.4 关闭游标语句

关闭指定的游标，收回它所占的资源。

语法格式为：

```
CLOSE  <游标名>;
```

参数说明：

- 游标名：指明被关闭的游标的名称

例如，关闭之前打开的 cheap_book 游标。

```
EXEC SQL CLOSE  cheap_book;
```

3.3.4.5 游标定位删除更新语句

■ **可更新游标**

在嵌入式 SQL 中，通过游标对基表进行修改和删除时要求该游标表必须是可更新的。可更新游标的条件是：游标定义中给出的查询说明必须是可更新的。DM 对查询说明是可更新的有这样的规定：

- 查询说明的 FROM 后只带一个表名，且该表必须是基表或者是可更新视图；
- 查询说明是单个基表或单个可更新视图的行列子集，SELECT 后的每个值表达式只能是单纯的列名，如果基表上有聚集索引键，则必须包含所有聚集索引键；

- 查询说明不能带 GROUP BY 子句、HAVING 子句、ORDER BY 子句；
- 查询说明不能嵌套子查询。

不满足以上条件的游标表是不可更新的。3.3.4.1 节例子中声明的游标 cheap_book 就是一个可更新游标。

■ 游标定位删除语句

游标定位删除语句的语法格式为：

```
DELETE FROM <表引用> [WHERE CURRENT OF <游标名>];
<表引用>::= [<模式名>.] <基表或视图名> | <外部连接表>
<基表或视图名>::=<基表名> | <视图名>
```

例如，使用之前声明的游标 cheap_book，下面的语句将游标拨动到指定的位置，并进行游标定位删除。

```
EXEC SQL OPEN  cheap_book;
EXEC SQL FETCH  ABSOLUTE  2  cheap_book;
EXEC SQL DELETE  FROM  PRODUCTION.PRODUCT  WHERE  CURRENT  OF  cheap_book;
```

■ 游标定位更新语句

游标定位更新语句的语法格式为：

```
UPDATE  <表引用>
SET <列名>=<值表达式>{,<列名>=<值表达式>}
[WHERE CURRENT OF <游标名>];
<表引用>::= [<模式名>.] <基表或视图名> | <外部连接表>
<基表或视图名>::=<基表名> | <视图名>
```

例如，使用之前声明的游标 cheap_book，下面的语句将游标拨动到指定的位置，并进行游标定位更新。

```
EXEC SQL OPEN  cheap_book;
EXEC SQL FETCH ABSOLUTE 3 cheap_book;
EXEC SQL UPDATE  PRODUCTION.PRODUCT  SET  NOWPIRCE=13.0000  WHERE  CURRENT
OF  cheap_book;
```

3.3.5 嵌入式程序中的异常处理

3.3.5.1 嵌入的异常声明语句

在 C 程序中，任何可出现说明语句的位置，皆可嵌入异常声明处理语句，其作用是指明在该异常声明处理语句的作用域内每个 SQL 操纵语句在发生操作异常时的处理方法。下面介绍异常声明语句的格式、作用域、异常动作的执行。

1. 语法格式

<嵌入的异常声明>::= EXEC SQL WHENEVER <条件><异常动作>;

<条件>::= SQLERROR | NOT FOUND

<异常动作>::= STOP| CONTINUE|GOTO <标号> |GO TO <标号> |DO<用户代码>

其中：<标号>为标准的 C 语言的标号。

2. WHENEVER 语句的作用域

一个 WHENEVER 语句的作用域是从该语句出现的位置开始，到下一个指明相同条件的 WHENEVER 语句出现（若没有下一个相同条件的 WHENEVER 语句，则到文件结束）之前的所有 SQL 语句。这种起始、终止位置是指源程序的物理位置，与该程序逻辑执行顺序无关。

3. 异常动作的说明

如果<异常动作>为 STOP，则停止操作；

如果<异常动作>为 CONTINUE，即使产生异常，也不需要作异常处理。使用 CONTINUE 的主要作用是取消前面与之具有相同条件的异常声明处理语句的作用；

如果动作为 GOTO<标号>，则程序转移到标号处执行；

如果动作为 DO<用户代码>，则程序继续执行用户的 C 代码。

4. 动作的触发条件

当一个 SQL 语句执行后返回的 SQLCODE < 0 时，SQLERROR 为真；当一个 SQL 语句执行后返回的 SQLCODE = 100，NOT FOUND 为真。

当嵌入的异常声明的条件得到满足时，则触发异常动作的执行。

5. 预编译的处理及举例

在对含有异常声明处理语句的程序进行预编译时，所有的异常声明处理语句都被删除。根据其作用域，在相应的 SQL 语句的调用函数之后，生成根据 SQLCODE 作相应处理的语句。

例如，按人员编号查询人员姓名、联系电话。设该文件名为 test.pc。

```
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE;
char SQLSTATE[6];
varchar person_name[50];
varchar person_phone[25];
varchar person_id[10];
EXEC SQL END DECLARE SECTION;
//①定义游标
EXEC SQL DECLARE C1 CURSOR FOR
SELECT NAME, PHONE FROM PERSON.PERSON WHERE PERSONID=:person_id;
EXEC SQL WHENEVER SQLERROR GOTO err_proc;
strcpy(person_id,"1");
EXEC SQL OPEN C1;       // ②打开游标
EXEC SQL WHENEVER NOT FOUND GOTO aaa;
for(;;)
{//③拨动游标
EXEC SQL FETCH C1 INTO :person_name,:person_phone;
printf("%s,%s ", person_name, person_phone);
}
......
aaa:
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL CLOSE C1;     //④关闭游标
exit(0);
err_proc:
printf("error = %d",SQLCODE);
```

```
exit(1);
```

若打开游标出错，则转至标号 err_proc 处执行。同样在取值过程中出错，也要转至 err_proc 处执行。若游标 C1 移至游标表的最后一行之后，则返回的 SQLCODE 为 100，程序将跳转到标号 aaa 处的语句处执行。

在对该程序段预编译后，生成对应 C 文件 test.c。程序 test.c 内容为：

```
#include "dpc_dll.h"


/* Thread Safety */
typedef void * sql_context;
typedef void * SQL_CONTEXT;


int  SQLCODE;
char SQLSTATE[6];
char    person_name[50];
char    person_phone[25];
char    person_id[10];
//①定义游标
/*EXEC SQL DECLARE C1 CURSOR FOR
SELECT name, phone
FROM person.person WHERE personid=:person_id;*/


{
       void*  handle = NULL;
       char   curname[128];
       char*  sqlstmt = NULL;
       sqlstmt = "SELECT name, phone FROM person.person WHERE personid=\n\
 ?;";
       dpc_alloc_bind_items(1, &handle);
       dpc_bind_item(handle, 1, "person_id", (void*)person_id, DPC_C_VARCHAR,
10, 10, 10, NULL, 0, -1);
```

```
	strcpy(curname, "C1");
	dpc_declare_cursor_ex(curname, sqlstmt, handle);
	dpc_free_bind_items(handle);
	dpc_get_rt_info(&SQLCODE, SQLSTATE);
}

strcpy(person_id,"1");

// ②打开游标
/*EXEC SQL OPEN C1;*/
{
	dpc_open_cursor_ex("C1",NULL, 1);
	dpc_get_rt_info(&SQLCODE, SQLSTATE);
	if (dpc_get_rt_info(&SQLCODE, SQLSTATE) < 0)
		goto  err_proc;
}

for(;;)
{//③拨动游标
/*EXEC SQL FETCH C1 INTO :person_name,:person_phone;*/
{
	void*  handle_using = NULL;
	void*  handle_into = NULL;
	dpc_alloc_bind_items(2, &handle_into);
	dpc_bind_item(handle_into, 1, "person_name", (void*)person_name,
DPC_C_VARCHAR, 50, 50, 50, NULL, 0, -1);
	dpc_bind_item(handle_into, 2, "person_phone", (void*)person_phone,
DPC_C_VARCHAR, 25, 25, 25, NULL, 0, -1);
	dpc_fetch("C1", handle_into, handle_using, "", "", 1, DPC_FETCH_NEXT,
DPC_C_INT, NULL, 4, 1);
```

```
       dpc_free_bind_items(handle_using);
       dpc_free_bind_items(handle_into);
       dpc_get_rt_info(&SQLCODE, SQLSTATE);
       if (dpc_get_rt_info(&SQLCODE, SQLSTATE) < 0)
              goto  err_proc;
       if (dpc_get_rt_info(&SQLCODE, SQLSTATE) == dpc_get_data_not_found())
              goto  aaa;
}

printf("%s,%s ", person_name, person_phone);
}
......
aaa:

//④关闭游标
/*EXEC SQL CLOSE C1;*/
{
       dpc_close_cursor_by_name("C1");
       dpc_get_rt_info(&SQLCODE, SQLSTATE);
       if (dpc_get_rt_info(&SQLCODE, SQLSTATE) < 0)
              goto  err_proc;
}

exit(0);
err_proc:
printf("error = %d",SQLCODE);
exit(1);
```

如果预编译出错，则给出错误信息说明。

3.3.5.2 通过 SQLCODE 和 SQLSTATE 获取 SQL 语句执行的信息

在 C 文件中可以定义两个特殊的变量 SQLSTATE 和 SQLCODE：

```
long SQLCODE;
char SQLSTATE[6];
```

这两个变量记录了当前 SQL 语句执行情况的信息。也就是说执行了一个 SQL 语句之后，可以通过返回的 SQLCODE 和 SQLSTATE 来判断语句执行是否发生异常；如果发生了异常，那么具体是哪一种异常。这些信息对于了解程序运行情况和程序流程的控制是很有用的。

在 DM 嵌入式程序中 SQLCODE 是一个 long 型变量，可能的值有：

- SQLCODE < 0，SQL 语句执行发生异常；
- SQLCODE = 0，SQL 语句执行成功；
- SQLCODE > 0，SQL 语句执行中产生警告。如 100 表示找不到数据（如删除一个空表中的记录等）。

DM 嵌入式程序中，SQLSTATE 是一个 char[6]型的变量。其前两个字符表示类别代码，随后的三个字符表示子类的代码。例如当 SQLSTATE = "22012"时，22 是类别代码，表示 SQL 语句发生了数据异常；012 是该类别下属的子类代码，表示发生的是被零除的数据异常。同样是数据异常，当 SQLSTATE = "22008"时表示的则是发生了日期时间数据溢出。DM 遵循 SQL92 标准，所以 SQLSTATE 中代码所表示的具体含义可以参考 SQL92 标准中的 SQLSTATE 部分。

3.3.6 数据类型支持

DM 嵌入式编程支持的数据类型见下表。

表 3.1 DM 嵌入式程序支持的数据类型

数据类型	DM_TYPE
DSQL_CHAR	1
DSQL_VARCHAR	2
DSQL_BIT	3
DSQL_TINYINT	5
DSQL_SMALLINT	6
DSQL_INT	7

DSQL_BIGINT	8
DSQL_DEC	9
DSQL_FLOAT	10
DSQL_DOUBLE	11
DSQL_BLOB	12
DSQL_DATE	14
DSQL_TIME	15
DSQL_TIMESTAMP	16
DSQL_BINARY	17
DSQL_VARBINARY	18
DSQL_CLOB	19
DSQL_TIME_TZ	22
DSQL_TIMESTAMP_TZ	23
DSQL_INTERVAL_YEAR	100
DSQL_INTERVAL_MONTH	101
DSQL_INTERVAL_DAY	102
DSQL_INTERVAL_HOUR	103
DSQL_INTERVAL_MINUTE	104
DSQL_INTERVAL_SECOND	105
DSQL_INTERVAL_YEAR_TO_MONTH	106
DSQL_INTERVAL_DAY_TO_HOUR	107
DSQL_INTERVAL_DAY_TO_MINUTE	108
DSQL_INTERVAL_DAY_TO_SECOND	109
DSQL_INTERVAL_HOUR_TO_MINUTE	110
DSQL_INTERVAL_HOUR_TO_SECOND	111
DSQL_INTERVAL_MINUTE_TO_SECOND	112

关于数据类型的长度、精度与刻度，需要注意以下几点：

- DSQL_CHAR、DSQL_VARCHAR 的长度为其定义的长度。char、varchar2 类型会被创建为 DSQL_CHAR 类型；
- DSQL_BIT、DSQL_TINYINT、DSQL_SMALLINT、DSQL_INT、DSQL_BIGINT 的精度分别为 3、3、5、10、19，刻度为 0；
- DSQL_DEC 的精度与刻度根据定义来确定。dec 的精度与刻度都为 0；当 dm_svc.conf 中的 DEC2DOUB 参数为 TRUE 时，转换为 double 类型，精度为 53；
- DSQL_CLOB、DSQL_BLOB 类型的长度并不限制，默认返回-1；
- 创建 DSQL_FLOAT 需要使用 real 关键字，精度为 24；float 与 double 类型都会被创建为 DSQL_DOUBLE 类型，精度为 53。

3.4 编写嵌入式程序的注意事项

PRO*C 程序和 C 程序相类似，其编写也遵守标准 C 程序的规定。除此之外，还应注意下列问题：

1. 编程者应该在程序中定义一个类型为长整型的全局变量 SQLCODE。该变量可以在嵌入的变量声明节中定义，也可以在声明节之外定义。定义该变量的物理位置应先于所有可执行的 SQL 语句。SQLCODE 将保存一条 SQL 语句执行后所产生的返回码。
2. 应广泛使用嵌入的异常声明处理语句，以便及时根据返回结果进行相应的处理。异常声明语句可出现在程序中的任何语句可出现的位置上。
3. 宿主变量分全局变量与局部变量，在程序开始处定义的为全局变量，在子函数内定义的为局部变量，其作用域范围与相应位置定义的主语言变量相同，检查作用域的工作由主语言编译程序承担，预编译系统未作此项检查。
4. 最好将宿主变量定义为全局变量。
5. 应广泛使用指示变量。当查询的结果为空值时，若相应的目标变量之后无指示变量，SQLCODE 将被置为-5000。
6. 游标声明是一种特殊的说明性语句。我们规定该语句只能出现在可执行的 C 语句出现的位置上。在一个程序中，不能够定义同名游标，游标也不能跨模块使用。游标声明语句的物理位置应先于使用该游标的打开、拨动与关闭语句。在一个程序中，可以多次打开、关闭同一个游标。
7. 在一个程序中，第一条可执行的 SQL 语句必须是“LOGIN”语句。
8. 在程序运行结束前，最好有一条语句“LOGOUT”，注销对数据库的使用。
9. 最好能够根据 SQL 语句的执行结果进行错误处理。
10. 在对一个 PRO*C 程序进行预编译时（例如原文件为 test.pc），在生成的 C 文件中会自动生成如下语句：

```
#include "dpc_dll.h"
#include "sqlca.h"
static sqlca_t sqlca;
```

11. 一个 PRO*C 程序可以含有多个事务。在 DM 中，一个事务是被当作单个实体处理的 SQL 语句序列。在进行嵌入式程序设计时，应该注意正确及时的使用数据库的

提交语句或回滚语句，一个模块中的事务最好安排在本模块内提交或回滚。在一个事务中，要么每个 SQL 语句的操作结果都被提交使之变为长久生效，要么同时被回滚使其改变被取消。

12. 如果多个用户进程因为争夺资源而发生死锁，DM 服务器将取消某一个用户进程的工作，该进程的当前事务被终止。

4 Oracle 兼容

DM PRO*C 对 Oracle PRO*C 的常用语法进行了兼容，在 dpc_new 预编译命令中将 MODE 参数置为 ORACLE 表示使用兼容 ORACLE 语法的编译模式。使用兼容 Oracle 模式时，在编译时还需要使用 sqlca_ora.h、sqlda_ora.h、dpc_ora_dll.h。

4.1 简单的 Oracle 嵌入式程序结构分析

首先，通过一个简单的 Oracle 嵌入式程序入手，对 Oracle 嵌入式程序的结构及基本语法进行分析。

例如，以下名为 test_ora.pc 的嵌入式程序，完成对表 test_ora 进行全表及条件查询并打印相关信息的功能。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//定义列和绑定变量的最大个数
#define MAX_ITEMS 40
//定义列名的最大值
#define MAX_VNAME_LEN 30
#define MAX_INAME_LEN 30
int alloc_descriptor(int size,int max_vname_len,int max_iname_len);
void set_bind_v();
void set_select_v();
void free_da();
void sql_error(char *msg);
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
//宿主变量定义：
EXEC SQL BEGIN DECLARE SECTION;
```

```
    char sql_statement[256]= "select * from test_ora";
    char type_statement[256]="select f1,f2 from test_ora where f1=1";
    int f1 = 1;
    int i;
    VARCHAR username[10];
    VARCHAR password[10];
    VARCHAR servername[20];
EXEC SQL END DECLARE SECTION;
        SQLDA *bind_p;
        SQLDA *select_p;
       int main()
       {
           strcpy(username.arr,"SYSDBA");
           username.len = strlen(username.arr);
           username.arr[username.len] = '\0';
           strcpy(password.arr,"SYSDBA");
           password.len = strlen(password.arr);
           password.arr[password.len] = '\0';
           EXEC SQL CONNECT :username identified by :password
using :servername;
           printf("\n [OK Connected!] \n\n");
           EXEC SQL WHENEVER SQLERROR DO sql_error("<ERROR>");
           alloc_descriptor(MAX_ITEMS,MAX_VNAME_LEN,MAX_INAME_LEN);

EXEC SQL CREATE TABLE TEST_ORA(f1 INT, f2 INT);
       EXEC SQL     INSERT INTO TEST_ORA VALUES(1,2);
        EXEC SQL  INSERT INTO TEST_ORA VALUES(1,4);
           EXEC SQL PREPARE S from :type_statement;
           EXEC SQL DECLARE C1 CURSOR FOR S;
           set_bind_v();
```

```
          EXEC SQL OPEN C1 USING DESCRIPTOR bind_p;
          EXEC SQL DESCRIBE SELECT LIST for S INTO select_p;
set_select_v();
          printf("f1\t\tf2\n");
          printf("------------------------------------------------------------
\n");
          for(;;)
          {    EXEC SQL WHENEVER NOT FOUND DO break;


               EXEC SQL FETCH C1 USING DESCRIPTOR select_p;
               for(i = 0;i<select_p->F;i++){
                  printf("%s ",select_p->V[i]);
              }
               printf("\n");
          }
          free_da();
          EXEC SQL CLOSE C1;
         printf("\n------------------------------------------------------\n");
          alloc_descriptor(MAX_ITEMS,MAX_VNAME_LEN,MAX_INAME_LEN);
          EXEC SQL PREPARE S from :sql_statement;
          EXEC SQL DECLARE C CURSOR FOR S;
          set_bind_v();
          EXEC SQL OPEN C USING DESCRIPTOR bind_p;
          EXEC SQL DESCRIBE SELECT LIST for S INTO select_p;
            set_select_v();
          for(;;)
          {    EXEC SQL WHENEVER NOT FOUND DO break;
               EXEC SQL FETCH C USING DESCRIPTOR select_p;
               for(i = 0;i<select_p->F;i++)
                  printf("%s ",select_p->V[i]);
```

```
                printf("\n");
            }
            free_da();
            EXEC SQL CLOSE C;
               EXEC SQL DROP TABLE TEST_ORA;
            EXEC SQL COMMIT WORK RELEASE;
            exit(0);
        }
//分配描述符空间：
int alloc_descriptor(int size,int max_vname_len,int max_iname_len)
        {
            int i;
            if((bind_p=SQLSQLDAAlloc(0,size,max_vname_len,max_iname_len))==(SQL
DA*)0)
            {
                printf("can't allocate memory for bind_p.");
                return -1;
            }
            if((select_p=SQLSQLDAAlloc(0,size,max_vname_len,max_iname_len))==(S
QLDA*)0)
            {
                printf("can't allocate memory for select_p.");
                return -1;
            }
            return 0;
    }
        //绑定变量的设置：
        void set_bind_v()
        {
            int i;
```

```
            EXEC SQL WHENEVER SQLERROR DO sql_error("<ERROR>");
            bind_p ->N = MAX_ITEMS;
            EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_p;
            if(bind_p->F<0)
            {
                printf("Too Many bind varibles");
                return;
            }
            bind_p->N = bind_p->F;
            for(i=0;i<bind_p->N;i++)
            {
                bind_p->T[i] = 1;
            }
        }
//选择列处理
        void set_select_v()
        {
                   int i,null_ok,precision,scale;
             EXEC SQL DESCRIBE SELECT LIST for S INTO select_p;
             if(select_p->F<0)
            {
                printf("Too Many column varibles");
                return;
            }
            select_p->N = select_p->F;
            //对格式作处理
            for(i = 0;i<select_p->N;i++)
            {
                sqlnul(&(select_p->T[i]), &(select_p->T[i]), &null_ok);//检查类
型是否为空
```

```
            switch (select_p->T[i])
            {
               case 1://VARCHAR2
                    break;
               case 2://NUMBER
                    sqlprc(&(select_p->L[i]), &precision, &scale);
                    if (precision == 0)
                      precision = 40;
                    select_p->L[i] = precision + 2;
                    break;
               case 8://LONG
                     select_p->L[i] = 240;
                     break;
               case 11://ROWID
                     select_p->L[i] = 18;
                     break;
               case 12://DATE
                     select_p->L[i] = 9;
                     break;
               case 23://RAW
                     break;
               case 24://LONGRAW
                     select_p->L[i] = 240;
              break;
       }
            select_p->V[i] = (char *)realloc(select_p->V[i], select_p-
>L[i]+1);
            select_p->V[i][select_p->L[i]] ='\0';//加上终止符
            select_p->T[i] = 1;//把所有类型转换为字符型
          }
```

```
      }
      //释放内存 SQLDA 的函数：
      void free_da()
      {
          SQLSQLDAFree(0,bind_p);
          SQLSQLDAFree(0,select_p);
  }
//错误处理
      void sql_error(char *msg)
      {
          printf("\n%s %s\n", msg,(char *)sqlca.sqlerrm.sqlerrmc);
          EXEC SQL ROLLBACK RELEASE;
          exit(0);
      }
```

由此可见，Oracle 嵌入式程序通常由以下部分组成：

1．SQLDA 申请与释放

```
int alloc_descriptor(int size,int max_vname_len,int max_iname_len);
void free_da();
```

详见 4.2 SQLDA/SQLCA。

2．SQLDA、SQLCA 引用

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
```

详见 4.2 SQLDA/SQLCA。

3．宿主变量定义

```
//宿主变量定义：
EXEC SQL BEGIN DECLARE SECTION;
    char sql_statement[256]= "select * from test_ora";
```

```
    char type_statement[256]="select f1,f2 from test_ora where f1=1";
    int f1 = 1;
    int i;
    VARCHAR username[10];
    VARCHAR password[10];
EXEC SQL END DECLARE SECTION;
```

详见 3.2 宿主变量定义。

4．数据库登录

```
EXEC SQL CONNECT :username identified by :password;
```

5．数据库操作

登录成功之后，便可对数据库进行各种操作。

6．异常处理

详见 3.3.5 嵌入式程序中的异常处理。

4.2 SQLDA/SQLCA

SQLDA、SQLCA 是 Oracle 用于记录 PRO*C 程序在执行过程中的描述信息及诊断信息。为了实现与 Oracle 的兼容，DM 在 dpc_dll 中对其进行实现，保留了 SQLDA、SQLCA 各成员变量在 Oracle 中的原始含义，并分别用于 Oracle PRO*C 程序在 DM 上执行的描述及诊断信息的记录。

SQLDA 本身并不存在，用户需先申请 SQLDA 描述空间才可使用，且使用完毕后，需进行释放。如有必要，用户可申请多个 SQLDA，方便使用。

另外，SQLDA、SQLCA 的相关声明与定义不会在预编译时自动引入，用户需在 PRO*C 文件的头部手动引入，代码如下所示：

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
```

以下详细介绍 DM 实现的 SQLDA、SQLCA 的相关操作函数。

1．SQLDA 空间的申请

函数定义：

1) SQLDA*

```
sqlald(
    int         max_n,
    size_t      max_name,
    size_t      max_i
);
```

2) SQLDA *

```
SQLSQLDAAlloc(
    void        *context,
    int   max_n,
    size_t      max_name,
    size_t      max_i
);
```

参数说明：

参数名称	含义说明
context	运行时上下文指针，DM 中作 0 处理
max_n	变量的最大个数
max_name	变量名称的最大长度
max_i	指示器(indicator)名称的最大长度

2．SQLDA 空间的释放

函数定义：

1) void

```
sqlclu(
    SQLDA*      da
);
```

2) void

```
SQLSQLDAFree(
    void*       context,
    SQLDA*      da
```

```
);
```

参数说明：

参数名称	含义说明
context	运行时上下文指针，DM 中作 0 处理
da	待释放的 SQLDA 变量名

3. Precision 和 Scale 抽取

函数定义：

1)
```
void
sqlprc(
    udint4*     length,
    sdint4*     precision,
    sdint4*     scale
);
```

2)
```
void
SQLNumberPrecV6(
    void*       context,
    udint4*     length,
    sdint4*     precision,
    sdint4*     scale
);
```

参数说明：

参数名称	含义说明
context	运行时上下文指针，DM 中作 0 处理
length	长整型变量指针，指向存放 Oracle NUMBER 类型值的长度;长度存储在 L[i]，scale 和 precision 分别存放在低字节与高字节
precision	整型变量指针，指向返回 Oracle NUMBER 类型值的 precision 整型变量
scale	整型变量指针，指向返回 Oracle NUMBER 类型值的 scale 整型变量

4. NULL/NOT NULL 数据类型处理

函数定义：

1)
```
void
```

```
sqlnul(
    udint2*     value_type,
    udint2*     type_code,
    sdint4*     null_status
);
```

2)
```
void
SQLColumnNullCheck(
    void*       context,
    udint2*     value_type,
    udint2*     type_code,
    sdint4*     null_status
);
```

参数说明：

参数名称	含义说明
context	运行时上下文指针，DM 中作 0 处理
value_type	无符号 short 整型变量指针，指向 select-list 列数据类型码； select-list 列数据类型存储在 T[i]中
type_code	无符号 short 整型变量指针，指向返回 select-list 列数据类型码的变量， 返回值中高位被清空
null_status	整型变量指针，指向返回的 select-list 列的 null 状态；1 代表允许为 null， 0 代表不允许为 null

5．错误信息获取

函数定义：

1)
```
void
sqlglm(
    char*       message_buffer,
    size_t*     buffer_size,
    size_t*     message_length
);
```

2)
```
void
```

```
SQLErrorGetText(
    void*       context,
    char*       message_buffer,
    size_t*     buffer_size,
    size_t*     message_length
);
```

参数说明：

参数名称	含义说明
context	运行时上下文指针，DM 中作 0 处理
message_buffer	指向存放错误信息缓存区
buffer_size	指向存放缓存区大小变量的指针
message_length	指向返回错误信息最大长度变量的指针

4.3 可执行的 SQL 语句

DM 与 Oracle 在登录数据库、基本 SQL 语句语法的支持方面存在差异，因此，为方便用户使用，DM 对以下 Oracle 语法实现支持，其他可执行语句详见 3.3 节描述。

1. 数据库登录语句

数据库登录语句的语法如下：

```
EXEC SQL CONNECT :username IDENTIFIED BY :pwd [USING :servername];
```

功能：用于数据库的登录。

使用说明：

- 语句中 username 是长度最多为 128 个字符的指针，其值为注册用户的名字；
- pwd 表示相应的用户口令，口令的长度最多为 128 个字符；
- servername 为服务器的名字或 IP 地址，可选，名字的长度最多为 128 个字符。


注意：

在用户程序中，登录语句的逻辑位置必须先于所有可执行的 SQL语句，只有执行登录语句之后，才能执行其它可执行的 SQL语句。如果登录失败，则终止程序的运行。

2. 基本 SQL 语句

嵌入基本 SQL 语句的语法如下：

```
EXEC SQL FOR :array_size INSERT/UPDATE /DELETE-CLAUSE…;
或者
EXEC SQL FOR integer INSERT/UPDATE /DELETE-CLAUSE…;
```

功能：用于执行指定次数的 INSERT/UPDATE/DELETE 操作。

使用说明：

- array_size 为整形变量；
- integer 为整形数字；

当插入操作的 VALUES-语句包含宿主变量数组；或当更新操作的 SET-语句和 WHERE-语句包含宿主变量数组；或删除操作的 WHERE-语句包含宿主变量数组，且:array_size、integer 指示的值不大于宿主变量数组的最小长度时，:array_size、integer 代表后面语句的执行次数。其他情况为非法的执行语句，该类错误在 DM 预编译过程中不做处理。

具体使用示例可参见 6.5.5 节。

4.4 预编译命令 OPTION

语法：

```
EXEC ORACLE OPTION (:option = :option_value);
```

功能：兼容 Oracle 的 OPTION 预编译命令。

使用说明：

- option 表示预编译选项，目前只支持 CHAR_MAP；
- option_value 表示预编译选项的值，目前支持的 CHAR_MAP 选项其值仅有 STRING、CHARZ 和 CHARF。

例如：

```
CREATE TABLEchar_test (c1 int,c2 varchar(100));
INSERT INTO char_test VALUES(1,'abcde');
COMMIT;
```

下面的程序片段说明了 CHAR_MAP 取值 STRING、CHARZ 和 CHARF 时执行结果的不同。

```
……
```

```
char ch_array[5];
short ind;
strncpy(ch_array, "12345", 5);
EXEC ORACLE OPTION (char_map=charz);
EXEC SQL SELECT c2 INTO :ch_array:ind FROM char_test WHERE c1=1 ;
/* 结果为ch_array = { 'A', 'B', 'c', 'd', '\0' }，ind=5 */

strncpy (ch_array, "12345", 5);
EXEC ORACLE OPTION (char_map=string) ;
EXEC SQL SELECT c2 INTO :ch_array:ind FROM char_test WHERE c1=1 ;
/* 结果为ch_array = { 'A', 'B', 'c', 'd', '\0' }，ind=5 */

strncpy( ch_array, "12345", 5);
EXEC ORACLE OPTION (char_map=charf);
EXEC SQL SELECT c2 INTO :ch_array:ind FROM char_test WHERE c1=1 ;
/* 结果为ch_array = { 'A', 'B', 'c', 'd', 'e' }，ind=0 */
……
```

4.5 数据类型映射

指定 Oracle 兼容的预编译模式时，数据类型的映射如下表所示。

表 4.1 Oracle 兼容数据类型映射

数据类型	DM_TYPE	ORACLE_TYPE 兼容	ANSI 标准类型
DSQL_CHAR	1	96	1
DSQL_VARCHAR	2	1	12
DSQL_BIT	3	2	14
DSQL_TINYINT	5	2	-1
DSQL_SMALLINT	6	2	5
DSQL_INT	7	2	4
DSQL_BIGINT	8	2	-2
DSQL_DEC	9	2	3
DSQL_FLOAT	10	2	6
DSQL_DOUBLE	11	2	8

DSQL_BLOB	12	113	不支持
DSQL_DATE	14	12	9
DSQL_TIME	15	187	9
DSQL_TIMESTAMP	16	187	9
DSQL_BINARY	17	23	-3
DSQL_VARBINARY	18	23	-3
DSQL_CLOB	19	112	-4
DSQL_TIME_TZ	22	186	9
DSQL_TIMESTAMP_TZ	23	188	9
DSQL_INTERVAL_YEAR	100	189	10
DSQL_INTERVAL_MONTH	101	189	10
DSQL_INTERVAL_DAY	102	190	10
DSQL_INTERVAL_HOUR	103	190	10
DSQL_INTERVAL_MINUTE	104	190	10
DSQL_INTERVAL_SECOND	105	190	10
DSQL_INTERVAL_YEAR_TO_MONTH	106	189	10
DSQL_INTERVAL_DAY_TO_HOUR	107	190	10
DSQL_INTERVAL_DAY_TO_MINUTE	108	190	10
DSQL_INTERVAL_DAY_TO_SECOND	109	190	10
DSQL_INTERVAL_HOUR_TO_MINUTE	110	190	10
DSQL_INTERVAL_HOUR_TO_SECOND	111	190	10
DSQL_INTERVAL_MINUTE_TO_SECOND	112	190	10

5 DB2 兼容

DM PRO*C 对 DB2 PRO*C 的常用语法进行了兼容，在 dpc_new 预编译命令中将 MODE 参数置为 DB2 表示使用兼容 DB2 语法的编译模式。使用兼容 DB2 模式时，在编译时还需要使用 sqlca_db2.h、sqlda_db2.h。

5.1 简单的 DB2 嵌入式程序结构分析

首先通过一个简单的 DB2 嵌入式程序入手，对 DB2 嵌入式程序的结构及基本语法进行分析。与 Oracle、DM 嵌入式程序不同的是，DB2 嵌入式程序的扩展名不是.pc，而是.sqc。

例如，以下名为 update.sqc 的嵌入式程序，完成对表 staff 相关操作的功能。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL INCLUDE SQLCA;
int main(int argc, char *argv[]) {
     EXEC SQL BEGIN DECLARE SECTION;
          char pname[10];
          short dept;
          char username[19],password[19],servername[19];
     EXEC SQL END DECLARE SECTION;
     printf(" Please input Servername :");
     scanf("%s",servername);
     printf("Input Username: ");
     scanf("%s", username);
     printf("Input Password: ");
     scanf("%s",password);
     EXEC SQL CONNECT TO :servername USER :username USING :password;
```

```
    EXEC SQL CREATE TABLE STAFF (name varchar(10), dept int, job
varchar(40));
    EXEC SQL INSERT INTO STAFF VALUES('Jack',80,'Mgr');
    EXEC SQL INSERT INTO STAFF VALUES('Joan',30,'Mgr');
    EXEC SQL DECLARE c1 CURSOR FOR
    SELECT name, dept FROM staff WHERE job='Mgr'FOR UPDATE OF job;
    EXEC SQL OPEN c1;
    do {
        EXEC SQL FETCH c1 INTO :pname, :dept;
        if (SQLCODE != 0) break;
        if (dept > 40) {
            printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
            EXEC SQL UPDATE staff SET job = 'Clerk'
            WHERE CURRENT OF c1;
        }
        else {
            printf ("%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
            EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
        } /* endif */
    } while ( 1 );
    EXEC SQL CLOSE c1;
    EXEC SQL DROP TABLE STAFF;
    EXEC SQL ROLLBACK;
    printf( "\nOn second thought -- changes rolled back.\n" );
    EXEC SQL CONNECT RESET;
    return 0;
}
```

与其他 PRO*C 文件类似，DB2 的 sqc 文件由以下几个部分组成：

1. SQLDA/SQLCA 引用

```
EXEC SQL INCLUDE SQLCA;
```

详见 5.2 SQLDA/SQLCA。

2. 宿主变量定义

```
EXEC SQL BEGIN DECLARE SECTION;
     char pname[10];
     short dept;
     char userid[9];
     char passwd[19];
EXEC SQL END DECLARE SECTION;
```

详见 3.2 宿主变量定义。

3. 登录数据库

```
EXEC SQL CONNECT TO localhost USER :userid USING :passwd;
```

4. 数据操作

登录成功之后，便可对数据库进行各种操作。

5. 退出登录

```
EXEC SQL CONNECT RESET;
```

操作完毕后，从数据库退出登录。

5.2 SQLDA/SQLCA

DB2 中的 SQLDA、SQLCA 是用于记录 PRO*C 程序在执行过程中的描述信息及诊断信息。为了实现与 DB2 的兼容，DM 在 dpc_dll 中对其进行实现，保留了 SQLDA、SQLCA 各成员变量在 DB2 中的原始含义，并分别用于 DB2 PRO*C 程序在 DM 上执行的描述及诊断信息的记录。

SQLDA、SQLCA 的相关声明与定义在预编译时不会自动引入，用户需在 PRO*C 文件的头部手动引入，代码如下所示：

```
EXEC SQL INCLUDE SQLCA;

EXEC SQL INCLUDE SQLDA;
```

5.3 可执行的 SQL 语句

DM 与 DB2 登录数据库的语法不同，为此，DM 对 DB2 登录数据库语句语法实现了支持，其语法格式如下：

```
EXEC SQL CONNECT TO :servername USER :username USING :psw;

或 EXEC SQL CONNECT TO servername USER :username USING :psw;
```

功能：用于数据库的登录。

使用说明：

- 语句中 username 是长度最多为 128 个字符的指针或者为具体名称(如 localhost)，其值为注册用户的名字；
- pwd 表示相应的用户口令，口令的长度最多为 128 个字符；
- servername 为服务器的名字，名字的长度最多为 128 个字符。


注意：

在用户程序中，登录语句的逻辑位置必须先于所有可执行的 SQL语句，只有执行登录语句之后，才能执行其它可执行的 SQL语句。如果登录失败，则终止程序的运行。

其他可执行语句详见 3.3 节描述。

5.4 数据类型映射

指定 DB2 兼容的预编译模式时，数据类型的映射如下表所示。

表 5.1 DB2 兼容数据类型映射

数据类型	DM_TYPE	DB2_TYPE 兼容	ANSI 标准类型
DSQL_CHAR	1	-109	1
DSQL_VARCHAR	2	-108	12
DSQL_BIT	3	不支持	14
DSQL_TINYINT	5	不支持	-1
DSQL_SMALLINT	6	-121	5
DSQL_INT	7	-120	4

DSQL_BIGINT	8	-119	-2
DSQL_DEC	9	-117	3
DSQL_FLOAT	10	-116	6
DSQL_DOUBLE	11	-116	8
DSQL_BLOB	12	-106	不支持
DSQL_DATE	14	-100	9
DSQL_TIME	15	-101	9
DSQL_TIMESTAMP	16	-103	9
DSQL_BINARY	17	不支持	-3
DSQL_VARBINARY	18	不支持	-3
DSQL_CLOB	19	-107	-4
DSQL_TIME_TZ	22	不支持	9
DSQL_TIMESTAMP_TZ	23	不支持	9
DSQL_INTERVAL_YEAR	100	-100	10
DSQL_INTERVAL_MONTH	101	-100	10
DSQL_INTERVAL_DAY	102	-100	10
DSQL_INTERVAL_HOUR	103	-100	10
DSQL_INTERVAL_MINUTE	104	-100	10
DSQL_INTERVAL_SECOND	105	-100	10
DSQL_INTERVAL_YEAR_TO_MONTH	106	-100	10
DSQL_INTERVAL_DAY_TO_HOUR	107	-100	10
DSQL_INTERVAL_DAY_TO_MINUTE	108	-100	10
DSQL_INTERVAL_DAY_TO_SECOND	109	-100	10
DSQL_INTERVAL_HOUR_TO_MINUTE	110	-100	10
DSQL_INTERVAL_HOUR_TO_SECOND	111	-100	10
DSQL_INTERVAL_MINUTE_TO_SECOND	112	-100	10

6 DM 嵌入式 SQL 高级功能

6.1 SSL 连接

DM 支持 SSL 加密通信方式，可以通过配置 dm_svc.conf 文件实现 PRO*C 程序的 SSL 通信。dm_svc.conf 是 DM 数据库安装时生成的一个配置文件，在 Windows 操作平台下此文件位于%SystemRoot%\system32 目录，在 Linux 平台下此文件位于/etc 目录。

配置方法如下：

1. 打开 SSL

语法如下：

```
ENABLE_SSL=(参数)
```

参数说明：

- 1：启动 SSL 配置；0：关闭 SSL 配置

2. 配置具体用户的 SSL

PRO*C 程序连接数据库时，自动将用户名与配置文件中配置了 SSL 连接的用户名进行匹配，如果能匹配成功，则对其使用 SSL 连接。

配置具体用户 SSL 的语法如下：

```
SSL_CONFIG=((USER=(用户名 1)SSL_PATH=(SSL 路径 1)SSL_PWD=(SSL key1))(USER=(用户名 2)SSL_PATH=(SSL 路径 2)SSL_PWD=(SSL key2)))
```

参数说明：

- USER：需要使用 SSL 登录的用户名，最大长度 128 字节
- SSL_PATH：指定用户的 SSL 路径，最大长度 256 字节
- SSL_PWD：指定用户 SSL 的 key，最大长度 128 字节
- 完整的一个用户的 SSL 配置外面需要用括号扩起来，以分隔多个配置。

例如，配置两个 SSL 用户，SYSDBA 和 SYSSSO，配置文件中相关配置项如下设置。

```
ENABLE_SSL=(1)
SSL_CONFIG=((USER=(SYSDBA)SSL_PATH=( C:\dmdbms\bin\client_ssl\SYSDBA)SSL_PWD=
(SYSDBA))(USER=(SYSSSO)SSL_PATH=(C:\dmdbms\bin\client_ssl\SYSSSO)SSL_PWD=(SYS
SSO)))
```

dm_svc.conf 配置文件还可以进行其他配置项的配置，这里只介绍了 SSL 相关配置，其他配置项可参考《DM8 系统管理员手册》

6.2 PL/SQL 块

PRO*C 把 PL/SQL 语句块视作单一的嵌入 SQL 语句，任何 SQL 语句能嵌入的地方也能嵌入 PL/SQL 块。在宿主程序中嵌入 PL/SQL 块，必须使用关键字 EXEC SQL EXECUTE 和 ENDEXEC 将 PL/SQL 语句块括起来。

语法如下：

```
EXEC SQL EXECUTE
  BEGIN
plsql_block;
  END;
END-EXEC;
```

例如：

```
EXEC SQL EXECUTE
    BEGIN
SELECT NAME, EMAIL, PHONE
            INTO :person_name:ind_name, :person_email, :person_phone
            FROM PERSON.PERSON
            WHERE PERSONID = :person_number;
    END;
END-EXEC;
```

6.3 使用大字段句柄处理 LOB 类型

DM PRO*C 中的大字段句柄，对应 DM 数据库服务器的大字段类型（BLOB、CLOB、LONGVARBINARY、LONGVARCHAR、IMAGE、TEXT）。大字段句柄分为 OCIBlobLocator 和 OCIClobLocator。

在使用大字段句柄前，需要先定义、申请、绑定大字段句柄；使用结束后，需要关闭和释放大字段句柄。

1. 定义大字段句柄

像定义变量一样定义大字段句柄，例如下面的语句定义了一个大字段句柄 blob。

```
OCIBlobLocator *blob;
```

2. 申请大字段句柄

使用 ALLOCATE 申请大字段句柄，例如下面的语句申请大字段句柄 blob。

```
EXEC SQL ALLOCATE :blob;
```

3. 绑定大字段句柄

绑定大字段句柄就是通过 SQL 语句将数据库的大字段列绑定到大字段句柄变量，使本地的大字段变量与数据库的大字段列进行关联。可以通过 SELECT INTO 语句和 RETURNING 语句进行绑定。

例如：

```
EXECSQL INSERT INTO PRODUCTION.PRODUCT(NAME,AUTHOR,PUBLISHER,PUBLISHTIME,\
PRODUCTNO,PRODUCT_SUBCATEGORYID,SATETYSTOCKLEVEL,ORIGINALPRICE,NOWPRICE,\
DISCOUNT,DESCRIPTION,PHOTO,SELLSTARTTIME) VALUES('噼里啪啦丛书续集',\
'(日)佐佐木洋子','21 世纪出版社','1901-01-01','9787539125992',35,'10','58',\
'42','6.1','11','书',empty_blob(),'2006-03-20')RETURNING PHOTO INTO :blob;


EXEC SQL SELECT PHOTO INTO :blob FROM PRODUCTION.PRODUCT where NAME='噼里啪啦丛书续集';
```

4. 使用大字段句柄进行读写

使用大字段句柄读数据到本地缓存区的语法如下：

```
EXEC SQLLOB READ :amt FROM :src [AT :src_offset] INTO :buffer
 [WITH LENGTH :buflen] ;
```

参数说明：

- amt：输入/输出参数，输入表示将要读取的字节数，输出表示已经读取的字节数
- src：已绑定的大字段句柄
- src_offset：大字段读取的偏移量
- buffer：读取到的本地缓存区
- buflen：缓存区的长度

使用大字段句柄写数据到大字段中的语法如下：

```
EXEC SQL LOB WRITE [APPEND] [ONE ]:amt FROM :buffer [WITH LENGTH :buflen]
 INTO :dst [AT :dst_offset] ;
```

参数说明：

- amt：输入/输出参数，输入表示将要写入的字节数，输出表示已经写入的字节数
- dst：已绑定的大字段句柄
- dst_offset：大字段写入的偏移量
- buffer：本地缓存区地址
- buflen：缓存区的长度
- APPEND：表示从大字段的末尾开始写入

5．关闭大字段句柄

使用 CLOSE 关闭大字段句柄，例如下面的语句关闭大字段句柄 blob。

```
EXEC SQL LOB CLOSE :blob;
```

6．释放大字段句柄

使用 FREE 释放大字段句柄，例如下面的语句释放大字段句柄 blob。

```
EXEC SQL FREE :blob;
```

7．获取大字段的长度

可以使用如下所示语句将大字段句柄对应的大字段长度写入到变量中。

```
EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :itotal;
```

下面给出两个使用大字段句柄处理 LOB 类型的实例。

例 1：读取大字段数据。

```
#include <stdio.h>
#include "DCI.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
char     username[50];
char     password[50];
char     servername[50];
int    c1=1;
unsigned int iBufLen = 4;
unsigned int offset = 1;//1_based
unsigned int itotal;
unsigned char buf_blob[5];
unsigned char tmp[1024];
OCIBlobLocator *blob;
EXEC SQL END DECLARE SECTION;
/* Declare function to handle unrecoverable errors. */
void sql_error();

main()
{
/* Connect to DM. */
    strcpy(username, "SYSDBA");
    strcpy(password, "SYSDBA");
```

```
    strcpy(servername, "192.168.0.89:5289");
    EXEC SQL WHENEVER SQLERROR DO sql_error("DM error--");

    EXEC SQL CONNECT :username IDENTIFIED BY :password USING :servername;
    printf("\nConnected to dm as user: %s\n", username);
 printf("\n\n   BEGIN BLOB  select into    \n");
    EXEC SQL update PRODUCTION.PRODUCT set PHOTO='abcdabcdabcd';
    EXEC SQL commit;

    EXEC SQL ALLOCATE :blob;
    EXEC SQL SELECT PHOTO into :blob FROM PRODUCTION.PRODUCT where NAME='红楼
梦';
    EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :itotal;
    printf("itotal %d\n",itotal);
    while (1)
    {
           printf("offset %d\n",offset);
           EXEC SQL LOB READ :iBufLen FROM :blob AT :offset INTO :buf_blob;
           printf("%s\n",buf_blob);
           memcpy(tmp + offset - 1, buf_blob, iBufLen);
           offset = offset + iBufLen;
           if (offset >= itotal)
               break;
    }
    EXEC SQL FREE :blob;
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

void
```

```
sql_error(msg)
char *msg;
{
    char err_msg[512];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

例 2：写入大字段

```
EXEC SQL BEGIN DECLARE SECTION;
       OCIBlobLocator *blob;
       unsigned int offset = 1;
       unsigned int itotal = 2;
       unsigned char buf_blob[10]="AF";
EXEC SQL END DECLARE SECTION;
       EXEC SQL ALLOCATE :blob;
       EXECSQL INSERT INTO
PRODUCTION.PRODUCT(NAME,AUTHOR,PUBLISHER,PUBLISHTIME,\
PRODUCTNO,PRODUCT_SUBCATEGORYID,SATETYSTOCKLEVEL,ORIGINALPRICE,NOWPRICE,\
```

```
DISCOUNT,DESCRIPTION,PHOTO,SELLSTARTTIME) VALUES('噼里啪啦丛书续集',\
'(日)佐佐木洋子','21 世纪出版社','1901-01-01','9787539125992',35,'10','58',\
'42','6.1','11','书',empty_blob(),'2006-03-20')RETURNING PHOTO INTO :blob;
       EXEC SQL LOB WRITE :itotal FROM :buf_blob INTO :blob AT :offset;
       EXEC SQL FREE :blob;
       EXEC SQL COMMIT RELEASE;
```

6.4 游标变量

除了 3.3.4 节中介绍的游标语句，DM 嵌入式 SQL 还支持游标变量，将游标作为一种对象变量使用。使用时基本的步骤为：定义、申请、打开、获取数据、关闭、释放游标。

1. 定义游标变量，变量类型为 sql_cursor。

例如：

```
sql_cursor          person_cv;
sql_cursor          person_cursor;
sql_cursor          *person;/*游标变量指针*/
```

2. 申请游标变量。

例如：

```
EXEC SQL ALLOCATE :person_cv;
```

3. 打开游标变量

- 可以使用 PL/SQL 语句块打开，如：

```
EXEC SQL EXECUTE
   BEGIN
      OPEN :person_cv FOR SELECT * FROM person.person;
   END;
END-EXEC;
```

- 也使用包打开游标，如：

在服务器端定义包体：

```
CREATE PACKAGE demo_cur_pkg AS
    TYPE personName IS RECORD (name VARCHAR2(51));
```

```
    TYPE cur_type IS REF CURSOR RETURN personName;
    PROCEDURE open_person_cur (
                curs     IN OUT cur_type,
personid IN     NUMBER);
END;

CREATE PACKAGE BODY demo_cur_pkg AS
    CREATE PROCEDURE open_person_cur (
                curs     IN OUT cur_type,
personid IN     NUMBER) IS
    BEGIN
        OPEN curs FOR
            SELECT name FROM person.person
                WHERE personid = personid
                ORDER BY name ASC;
    END;
END;
```

在客户端使用此包：

```
EXEC SQL EXECUTE
    begin
        demo_cur_pkg.open_person_cur(:person_cursor, :personid);
    end;
END-EXEC;
```

4. 获取数据。

例如：

```
EXEC SQL FETCH :person_cv INTO :output_person_rec;
```

5. 释放游标变量。

例如：

```
EXEC SQL FREE :person_cv;
```

使用游标进行查询时，需要注意：

- 当返回值的数据类型与宿主变量的数据类型不一致时，DM 将返回值转换成宿主变量的类型。这种转换只局限于数值转换。不论数据类型如何，如果返回给宿主变量的值是 NULL，那么相应指示符变量被置为-1。如果没有与之相应的指示符变量，那么 SQLCODE 被设置为-5000。数值型数据转换，包括 short、int、double、float 四种数值型数据的转换，若发生溢出错误，将给出警告信息；
- 如果当前游标已经指向查询的最后一条记录，使用 FETCH 语句将会导致返回错误代码(SQLCODE=100)。

例：下面是一个使用游标变量的例子

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char     username[50];
char     password[50];
char     servername[50];
EXEC SQL END DECLARE SECTION;
typedef struct _EMP_CV
   {
   int personid;
   char sex[2] ;
   char name[51];
   char email[51];
   char phone[26];} V_EMP_CV;
EXEC SQL BEGIN DECLARE SECTION;
   SQL_CURSOR emp_cv;
   V_EMP_CV   output_emp_rec;
EXEC SQL END DECLARE SECTION;

/* Declare function to handle unrecoverable errors. */
```

```
void sql_error();


main()
{
/* Connect to DM. */
    strcpy(username, "SYSDBA");
    strcpy(password, "SYSDBA");
    strcpy(servername, "192.168.0.89:5289");


    EXEC SQL WHENEVER SQLERROR DO sql_error("DM error--");


    EXEC SQL CONNECT :username IDENTIFIED BY :password USING :servername;
    printf("\nConnected to dm as user: %s\n", username);
EXEC SQL ALLOCATE :emp_cv;
EXEC SQL EXECUTE
   BEGIN
      OPEN :emp_cv FOR SELECT * FROM person.person;
   END;
END-EXEC;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
for (;;)
{
   EXEC SQL FETCH :emp_cv INTO :output_emp_rec;
   printf("personid=%d; sex=%s; name=%s; email=%s;
phone=%s;\n",output_emp_rec.personid,output_emp_rec.sex,\
   output_emp_rec.name,output_emp_rec.email,output_emp_rec.phone);
}
EXEC SQL CLOSE :emp_cv;
EXEC SQL FREE :emp_cv;
```

```
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}
void
sql_error(msg)
char *msg;
{
    char err_msg[512];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

6.5 批量执行

DM 的 PRO*C 支持批量操作，可以批量查询、打开、获取和执行，批量操作描述信息的相关信息需要增加语法 FOR:size，其中 size 用于指定批量的行数，可以是常量，也可以是变量。

6.5.1 SELECT 批量操作

1. 使用数组宿主变量

当知道查询会返回的行数时，可以使用简单的 SELECT INTO 语句，你可以使用大于等于行数的数组宿主变量。

例：下例中从 PERSON.PERSON 表中查询 3 条记录放入数组宿主变量，而数组宿主变量可以存放 4 个元素。

```
int cc1[4];
VARCHAR cc2[4][200];
VARCHAR cc3[4][200];
EXEC SQL  SELECT top 3 personid,name,phone into :cc1,:cc2,:cc3 FROM
person.person;
```

2. 使用数组宿主变量+游标 FETCH

如果不知道查询结果的数据行数时，可以使用数组宿主变量+游标 FETCH，每次 FETCH 的行数等于宿主数组的大小。

例如，下例中每次 fetch 10 行。最后一次 fetch 是余下的数据可能小于 10 行。

```
int   personid[10];
char name[10][51];

EXEC SQL DECLARE person_cursor CURSOR FOR
    SELECT personid, name FROM person.person;

EXEC SQL OPEN person_cursor;

EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH person_cursor
```

```
        INTO :personid, :name;
    /* process batch of rows */
    ...
}
...
```

3. 使用 sqlca.sqlerrd[2]

对于 INSERT、UPDATE、DELETE 和 SELECT INTO 语句，sqlca.sqlerrd[2]记录了操作处理的数据行数。对于游标 FETCH 操作 sqlca.sqlerrd[2]记录的是当前总共 FETCH 的行数。你可以利用 sqlca.sqlerrd[2]的值进行批量操作的循环控制。

例如，下面代码片段中 sqlca.sqlerrd[2]用来记录已经 FETCH 的行数。

```
int  personid[50];
char name[50][51];

int rows_to_fetch, rows_before, rows_this_time;
EXEC SQL DECLARE person_cursor CURSOR FOR
    SELECT personid, name
    FROM person.person
    WHERE personid = 2;
EXEC SQL OPEN person_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
/* initialize loop variables */
rows_to_fetch = 10;   /* number of rows in each "batch" */
rows_before = 0;      /* previous value of sqlerrd[2]  */
rows_this_time = 10;

while (rows_this_time == rows_to_fetch)
{
    EXEC SQL FOR :rows_to_fetch
    FETCH person_cursor
```

```
        INTO :personid, :name;
    rows_this_time = sqlca.sqlerrd[2] - rows_before;
    rows_before = sqlca.sqlerrd[2];
}
...
```

4．批量获取描述信息

通过描述符可以获取动态 SQL 语句的输出列类型与长度，关于描述符的具体说明见 6.6 节。DM 支持批量获取描述信息。

例如：

```
char   sqlstr[500];
int    array_size=5;
int    col_cnt;
int    i;
char   c_name[50];
int    c1_data[5];
char   c2_data[5][21];
char   c3_data[5][50];
char   c4_data[5][50];
char   c5_data[5][50];
int    c1_type,c2_type,c3_type,c4_type,c5_type;
int    c1_length,c2_length,c3_length,c4_length,c5_length;

EXEC SQL drop table testdes;
EXEC SQL create table testdes(c1 int,c2 varchar(20),c3 varbinary(20),c4
clob,c5 blob);
EXEC SQL insert into testdes values(1, 'dm', 0x12ab, 'dameng', 0x34ef);
EXEC SQL insert into testdes values(2, 'dm', 0x12ab, 'dameng', 0x34ef);
EXEC SQL insert into testdes values(3, 'dm', 0x12ab, 'dameng', 0x34ef);
EXEC SQL insert into testdes values(4, 'dm', 0x12ab, 'dameng', 0x34ef);
```

```
EXEC SQL insert into testdes values(5, 'dm', 0x12ab, 'dameng', 0x34ef);


strcpy(sqlstr, "select c1,c2,c3,c4,c5 from testdes");


EXEC SQL PREPARE stmt FROM :sqlstr;
EXEC SQL DECLARE cur_stmt CURSOR FOR stmt;
EXEC SQL OPEN cur_stmt;


/***************
 * 绑定输出变量
 ***************/
EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'out';
EXEC SQL DESCRIBE OUTPUT stmt USING DESCRIPTOR 'out';


c1_type=4;
c1_length=4;
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:c1_type, LENGTH=:c1_length;
c2_type=1;
c2_length=10;
EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE=:c2_type, LENGTH=:c2_length;
c3_type=1;
c3_length=4;
EXEC SQL SET DESCRIPTOR 'out' VALUE 3 TYPE=:c3_type, LENGTH=:c3_length;
c4_type=1;
c4_length=10;
EXEC SQL SET DESCRIPTOR 'out' VALUE 4 TYPE=:c4_type, LENGTH=:c4_length;
c5_type=1;
c5_length=4;
EXEC SQL SET DESCRIPTOR 'out' VALUE 5 TYPE=:c5_type, LENGTH=:c5_length;
```

```
EXEC SQL GET DESCRIPTOR 'out' :col_cnt=COUNT;
printf("col_cnt:%d\n",col_cnt);

for(i=1;i<=col_cnt;i++)
{
       memset(c_name, 0, 10);
       EXEC SQL GET DESCRIPTOR 'out' VALUE :i :c_name=NAME;
       printf("%s  ",c_name);
}
printf("\n------------------------\n");

EXEC SQL FOR :array_size FETCH cur_stmt INTO DESCRIPTOR 'out';
EXEC SQL FOR :array_size GET DESCRIPTOR 'out' VALUE 1 :c1_data=DATA;
EXEC SQL FOR :array_size GET DESCRIPTOR 'out' VALUE 2 :c2_data=DATA;
EXEC SQL FOR :array_size GET DESCRIPTOR 'out' VALUE 3 :c3_data=DATA;
EXEC SQL FOR :array_size GET DESCRIPTOR 'out' VALUE 4 :c4_data=DATA;
EXEC SQL FOR :array_size GET DESCRIPTOR 'out' VALUE 5 :c5_data=DATA;

for(i=0;i<array_size;i++)
       printf("%d    %s    %s    %s    %s\n",c1_data[i],c2_data[i],c3_data[i],
c4_data[i],c5_data[i]);

EXEC SQL CLOSE cur_stmt;
EXEC SQL DEALLOCATE DESCRIPTOR 'out';
EXEC SQL DEALLOCATE DESCRIPTOR 'in';

EXEC SQL COMMIT WORK;
return 0;
```

6.5.2 INSERT 批量操作

同样地，可以使用宿主数组作为 INSERT 语句的输入变量，只需要在执行插入操作前给数组赋值即可。如果数组数据不足也可以使用 FOR 语句来指定批量插入的行数，未指定 FOR 则默认插入行数等于数组大小。

例如，下面代码片段使用数组进行批量插入。

```
char    name[50][51];
char    phone[50][21];
char    email[50][51];
/* 为数组宿主变量赋值*/
...
EXEC SQL INSERT INTO PERSON.PERSON (NAME, PHONE, EMAIL)
    VALUES (:name, :phone, :email);
```

6.5.3 UPDATE 批量操作

也可以使用数组作为 UPDATE 语句的宿主变量，与 INSERT 批量操作类似。

例如，下面代码片段使用数组进行批量更新。

```
int   person_number[50];
char  phone[50][51];
/* 为数组宿主变量赋值*/
...
EXEC SQL UPDATE PERSON.PERSON SET PHONE = :phone
    WHERE PERSONID = :person_number;
```

在上面例子中 PERSONID 是表 PERSON.PERSON 的主键，每个 PERSONID 对应一行数据。批量更新也可以是条件数组中每条数据对应多行。

例如:

```
char   sex [2][51];
char   email[2][51];
/* 为数组宿主变量赋值 */
```

```
...
EXEC SQL UPDATE person.person SET email = :email
    WHERE sex = :sex;
```

6.5.4 DELETE 批量操作

同样可以使用数组作为 DELETE 语句的宿主变量，与 INSERT 批量操作类似。

例如，下面代码片段使用数组进行批量删除。

```
int   person_number[50];
char  phone[50];
/* 为数组宿主变量赋值*/
...
EXEC SQL DELETE FROM PERSON.PERSON  WHERE PERSONID = :person_number;
```

在上面例子中 PERSONID 是表 PERSON.PERSON 的主键，每个 PERSONID 对应一行数据。批量删除也可以是条件数组中每条数据对应多行。

例如：

```
char  sex [2][51];
...
EXEC SQL DELETE FROM  person.person WHERE sex = :sex;
```

6.5.5 FOR 语法

前面介绍的的 SELECT、UPDATE、INSERT 和 DELETE 批量操作都可以使用 FOR 语法指定具体批量操作的行数。

例如，下面的代码片段使用 FOR 语法指定 INSERT 批量操作处理的行数。

```
char  name[100][51];
char email[100][51];
int   rows_to_insert;
/* 为数组宿主变量赋值*/
...
rows_to_insert = 25;              /* 设置 FOR 子句变量*/
```

```
EXEC SQL FOR :rows_to_insert      /* 将插入 25 条记录*/
    INSERT INTO PERSON.PERSON (NAME, EMAIL)
    VALUES (:name, :email);
```

6.5.6 使用结构数组

当批量操作同时涉及到多个列时，可以将这些列集成到一个结构体中，然后定义结构数组，在嵌入式 SQL 中使用结构数组进行查询或插入更新，这样也能实现批量的效果。

在嵌入式 SQL 中使用结构数组有以下限制:

- 不能在 PL\SQL 语句块中使用结构数组
- 不能在 where 条件中使用结构数组
- 不能在 update 语句中使用结构数组

例如：下例中批量查询同时查询 PERSON.PERSON 的三个列，因此定义了一个对应这三个列的结构数组 person_rec，同时还使用了对应这个结构数组的指示变量数组。

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
struct
{
    int    personid;
    char   name[51];
    char  email[51];
} person_rec[5];
struct
{
    short    ind_personid;
    short   ind_name;
    short  ind_email;
} ind_person_rec[5];
char     username[50];
```

```
char     password[50];
char     servername[50];
int  num_ret;
EXEC SQL END DECLARE SECTION;

/* Declare function to handle unrecoverable errors. */
void sql_error();
void print_rows();

main()
{
/* Connect to DM. */
    strcpy(username, "SYSDBA");
    strcpy(password, "SYSDBA");
    strcpy(servername, "192.168.0.89:5289");
    EXEC SQL WHENEVER SQLERROR DO sql_error("DM error--");
    EXEC SQL CONNECT :username IDENTIFIED BY :password USING :servername;
    printf("\nConnected to dm as user: %s\n", username);

/* Declare a cursor for the FETCH. */
    EXEC SQL DECLARE c1 CURSOR FOR
        SELECT personid, name, email FROM person.person where personid<15;
    EXEC SQL OPEN c1;
    num_ret = 0;

/* Array fetch loop - ends when NOT FOUND becomes true. */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
```

```
        EXEC SQL FETCH c1 INTO :person_rec :ind_person_rec;
        print_rows(sqlca.sqlerrd[2] - num_ret);
        num_ret = sqlca.sqlerrd[2];        /* Reset the number. */
    }
/* Print remaining rows from last fetch, if any. */
    if ((sqlca.sqlerrd[2] - num_ret) > 0)
        print_rows(sqlca.sqlerrd[2] - num_ret);
    EXEC SQL CLOSE c1;
    printf("\nAu revoir.\n\n\n");
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

void
print_rows(n)
int n;
{
    int i;

    printf("\npersonid   name         email");
    printf("\n------   --------         ------\n");

    for (i = 0; i < n; i++)
        printf("%d;%s; %s\n", person_rec[i].personid,
               person_rec[i].name, person_rec[i].email);
}

void
sql_error(msg)
char *msg;
```

```
{
    char err_msg[512];
    size_t buf_len, msg_len;

    printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

6.6 动态 SQL 语句

动态 SQL 技术是一种先进的程序设计技术。

前面介绍的嵌入式 SQL 语句为编程提供了一定的灵活性，使用户可以在程序运行过程中根据实际需要输入 WHERE 子句或 HAVING 子句中的某些变量的值。这些 SQL 语句的共同特点是，语句中主变量的个数与数据类型在预编译时都是确定的，只是主变量的值是程序运行过程中动态输入的，我们称这类嵌入式 SQL 语句为静态 SQL 语句。

静态 SQL 语句提供的编程灵活性在许多情况下仍显得不足，有时候需要编写更为通用的程序。例如查询人员信息表，经理想查询所有人员姓名、性别，员工想查询联系电话，也就是说查询条件是不确定的，要查询的属性列也是不确定的，这就无法用一条静态 SQL 语句实现了。因为在这些情况下，SQL 语句不能完整地写出来，而且这类语句在每次执行时都还有可能变化，只有在程序执行时才能构造完整。象这种在程序执行过程中临时生成的 SQL 语句叫动态 SQL 语句。

实际上，如果在预编译时下列信息不能确定，就必须使用动态 SQL 技术：

- SQL 语句正文；
- 主变量个数；
- 主变量的数据类型；
- SQL 语句中引用的数据库对象（例如，列、索引、基本表、视图等）。

动态 SQL 方法允许在程序运行过程中临时“组装”SQL 语句，主要有三种形式：

1. 语句可变；
2. 条件可变；
3. 数据对象、查询条件均可变。

这几种动态形式几乎覆盖所有的可变要求。为了实现上述三种可变形式，SQL 提供了相应的语句，例如，EXECUTE IMMEDIATE、PREPARE、EXECUTE、DESCRIBE 等。

6.6.1 DM 动态 SQL 语句

6.6.1.1 动态 SQL 语句的表示方法

一般来说，应该使用一个字符串变量来表示一个动态 SQL 语句的文本，但该文本部分包括 EXEC SQL 子句。


注意：

下列语句不能作动态SQL语句：CLOSE、 DECLARE、DESCRIBE、EXECUTE、FETCH、INCLUDE、OPEN、WHENEVER、PREPARE。

在大多数情况下，动态 SQL 语句的文本中可能包含虚拟宿主变量，这些变量只为实宿主变量保留位置。

例如，下列例子中的问号就是只为实宿主变量保留位置：

```
DELETE FROM EMP WHERE NO=? AND NAME=?;
```

6.6.1.2 生成动态 SQL 语句的方法

在 DM 嵌入式 SQL 中，生成动态 SQL 语句的方法有四种，下面分别介绍。

1. 方法一

用立即执行语句执行动态 SQL 语句。

语法格式如下：

```
EXEC SQL EXECUTE IMMEDIATE <SQL 动态语句文本>;
```

功能：动态地准备和执行一条语句。该语句首先分析动态语句文本，检查是否有错误，如果有错误则不执行它，并在 SQLCODE 中返回错误码；如果没发现错误则执行它。

使用说明：用该方法处理的 SQL 语句一定不是 SELECT 语句，而且不包含任何虚拟的输入宿主变量。

例如，下列 SQL 动态语句文本串是合法的：

```
CREATE TABLE person.person
 (
"personid" integer identity(1,1) not null,/*人员编号*/
"sex" char(1) not null,/*人员性别*/
"name" varchar(50) not null,/*人员姓名*/
"email" varchar(50),/*邮箱地址*/
"phone" varchar(25),/*联系电话*/
cluster primary key("personid")
)
INSERT INTO person.person VALUES('F' , '李丽', 'lily@sina.com', '02788548562');
```

如果有语句 EXEC SQL EXECUTE IMMEDIATE :string，则该语句将分析和执行宿主变量 string 中的 SQL 语句。

例如，有语句 EXEC SQL EXECUTE IMMEDIATE INSERT INTO person.person VALUES('F' , '李丽', 'lily@sina.com', '02788548562')。该语句直接分析和执行 INSERT INTO person.person VALUES('F' , '李丽', 'lily@sina.com', '02788548562')。

对于仅执行一次的动态语句，用立即执行语句最合适。而对于重复多次执行的动态 SQL 语句，用立即执行语句则会降低效率，应选用方法二。

2．方法二

方法二首先用准备语句，准备将要执行的动态 SQL 语句，然后用执行语句 EXECUTE 执行已准备好的动态 SQL 语句。在该方法中，动态 SQL 语句的处理分三步：

1） 构造一个动态 SQL 语句；

2） 用 PREPARE 语句来分析和命名该动态 SQL 语句；

3） 用 EXECUTE 语句来执行它。

下面分别介绍 PREPARE、EXECUTE 语句。

■ 准备语句

语法格式如下：

```
EXEC SQL PREPARE <SQL 语句名> FROM <SQL 动态语句文本>;
```

<SQL 语句名>标识被分析的动态 SQL 语句，它是供预编译程序使用的标识符，而不是宿主变量；<SQL 动态语句文本>包含动态 SQL 语句。

功能：准备一个语句执行。

使用说明：用该方法处理的 SQL 语句，不能是 SELECT 语句。该语句可能包含虚拟输入宿主变量（用问号表示），而且变量的类型是已知的。

例如：

```
INSERT INTO person.person VALUES(?, ?, ?, ?);
DELETE FROM person.person WHERE personid =?;
```

例如，用 stmt 标识被分析的 SQL 语句，”?”表示一个虚拟输入宿主变量。

```
strcpy(sql_stmt, "DELETE FROM person.person WHERE personid =?");
EXEC SQL PREPARE stmt FROM :sql_stmt;
```

■ 执行语句

语法格式如下：

```
EXEC SQL EXECUTE <SQL 语句名> [<结果使用子句>];
```

功能：执行一个由 PREPARE 准备好的动态 SQL 语句。

使用说明：<SQL 语句名>标识被分析的动态 SQL 语句。<结果使用子句>指出一个实宿主变量表，用于替换虚宿主变量，且<结果使用子句>中的实宿主变量要与被分析的动态 SQL 语句中的虚宿主变量在类型、次序上相对应，个数相匹配。

例如，执行上例准备的 SQL 语句。

```
EXEC SQL EXECUTE stmt USING :personid;
```

其中，personid 是一个实输入宿主变量，它替换前面所述的动态 SQL 语句中的“?”。

3．方法三

用方法三处理的 SQL 语句一定是 SELECT 语句，而且它包含的选择表项或虚拟输入宿主变量的个数或类型在预编译时都已知。

例如：

```
SELECT NAME, PHONE FROM PERSON.PERSON WHERE PERSONID=?
```

在方法三中，动态 SQL 语句的处理过程如下：

1） 构造一个动态 SQL 语句；

2） 用 PREPARE 语句来分析和命名该动态 SQL 语句，其描述同法二；

3） 用游标语句来执行它。

执行动态 SQL 查询的游标语句包括动态声明游标、动态打开游标、动态拨动游标、动态关闭游标，其中除了动态打开游标，其余的使用方法与 3.3.4 节中介绍的相同。

下面介绍一下动态打开游标语句，语法格式为：

```
EXEC SQL OPEN <游标名> [USING : host_variable [,.]];
```

功能：打开动态游标。

使用说明：<游标名>是要打开的游标的名字，host_variable 是实际宿主变量，用于替换动态 SQL 语句中的虚拟宿主变量。

6.6.1.3 动态 SQL 实例

下面的例子示范了三种生成动态 SQL 的方法。在运行本程序之前，应该先确定登录的用户名下是否已经存在表 PERSON，如果这个表已经存在，那么程序将在执行一开始的建表语句时发生异常而退出登录。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL BEGIN DECLARE SECTION;
long SQLCODE;
```

```
char SqlStr[200];
char username[19];
char password[19];
char servername[20];
varchar person_no[10],person_name[50],person_phone[25];
EXEC SQL END DECLARE SECTION;
void main()
{
int i;
char person_id[3][6]={"1","2","3"};
char person_name[3][11]={"李丽","王刚","李勇",};
char person_phone[3][12]={"02788548562","13055173012","15955173024"};
sprintf(servername,"localhost");
sprintf(username,"SYSDBA");
sprintf(password,"SYSDBA");
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
EXEC SQL WHENEVER SQLERROR GOTO sql_err;
/* 用方法一建表 */
strcpy(SqlStr,"CREATE TABLE person (\"personid\" integer identity(1,1) not
null,\"name\" varchar(50) not null,\"phone\" varchar(25))");
EXEC SQL EXECUTE IMMEDIATE :SqlStr;
/* 用方法二插入一组值 */
strcpy(SqlStr,"INSERT INTO person (\"name\", \"phone\") VALUES(?,?)");
EXEC SQL PREPARE INS1 FROM :SqlStr;
for(i=0;i<3;i++)
{
strcpy(person_name,person_name[i]);
strcpy(person_phone,person_phone[i]);
EXEC SQL EXECUTE INS1 USING :person_name,:person_phone;
EXEC SQL COMMIT WORK;
```

```
}
/* 用方法三查询并显示结果 */
strcpy(SqlStr,"SELECT \"name\", \"phone\" FROM person WHERE \"personid\" =?");
strcpy(person_no,"1");
EXEC SQL PREPARE S1 FROM :SqlStr;
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1 USING :person_no;
printf("The resultis:%s\n",person_no);
EXEC SQL FETCH C1 INTO :person_name,:person_phone;
printf("NO=%s,NAME=%s\n\n",person_phone, person_name);
EXEC SQL CLOSE C1;
EXEC SQL DROP TABLE PERSON CASCADE;
sql_err:
printf("Error Code:%d\n",SQLCODE);
EXEC SQL ROLLBACK;
EXEC SQL LOGOUT;
exit(1);
}
```

6.6.2 ANSI 动态 SQL 语句

DM 支持 ANSI 标准的动态 SQL 语法，以下面的 SQL 为例：

```
SELECT PERSONID ,NAME FROM PERSON.PERSON WHERE EMAIL=:email_data;
```

使用 ANSI 动态 SQL 的步骤是：

1. 声明变量，包含一个字符串变量存放待执行的 SQL 语句；
2. 为输入输出变量分配描述符；
3. 准备（PREPARE）SQL 语句；
4. 为输入描述符描述输入；
5. 设置输入描述符（上述 SQL 中有一个输入宿主变量 email_data）；
6. 声明打开一个动态游标；

7. 设置输出描述符（上述 SQL 中有两个输出变量：personid 和 name）

8. 反复 FETCH 数据，使用 GET DESCRIPTOR 检索 personid 和 name 对应输出描述符的 data 域；

9. 关闭游标，释放描述符。

ANSI 动态 SQL 语句中描述符起着重要的作用，下面主要介绍描述符相关操作，其他的声明 SQL 语句变量、准备语句等在前面都已介绍过了。

1. 分配描述符

在动态 SQL 语句中使用描述符前，必须先分配描述符。

语法格式为：

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] <:desc_nam | string_literal>
  [WITH MAX <:occurrences | numeric_literal>];
```

参数说明：

- GLOBAL 描述符能够在其他模块中使用，LOCAL 描述符只能在当前模块中使用；
- 描述符的名字可以是一个宿主变量，也可以是字符串；
- occurrences 是描述符最多可以绑定的变量数，可以使用宿主变量或者直接在 SQL 中使用数字表示。

2. 获取语句描述信息

描述符分为输入变量的描述符和输出变量的描述符。DESCRIBE 输入描述符可描述语句的绑定变量，DESCRIBE 输出描述符能描述输出列的类型跟长度。

语法格式为：

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] <sql_statement>   USING [SQL] DESCRIPTOR
[GLOBAL | LOCAL] <:desc_nam | string_literal>;
```

3. 设置描述符

使用 SET DESCRIPTOR 语句设置输入描述符属性。

语法格式为:

```
EXEC SQL [FOR <array_size>] SET DESCRIPTOR  <:desc_nam | string_literal> <
<COUNT = :hv0> | <VALUE item_number> >
```

```
< [REF] item_name1 = :hv1 > {, < [REF] item_nameN = :hvN > } ;
```

参数说明：

- FOR <array_size>: 数组大小，oracle 兼容模式下支持设置 DATA，INDICATOR 为数组宿主变量。FOR 的详细用法请参考 6.5.5 FOR 语法 。
- :desc_nam：前面 ALLOCATE DESCRIPTOR 分配描述符名称，以宿主变量形式书写。例如，char c1="out_lobInsert";……SET DESCRIPTOR :c1……。
- string_literal：前面 ALLOCATE DESCRIPTOR 分配描述符名称，以字符串形式书写。例如，……SET DESCRIPTOR 'out_lobInsert'……。
- COUNT：输入或输出绑定变量个数，可以是一个宿主变量，也可以直接使用数字。
- VALUE item_number：动态 SQL 绑定变量位置。
- hv1 .. hvN： 设置的用来绑定宿主变量。
- item_nameI：描述符属性名，详细参见表 6.1。
- REF :仅语法上支持，不起任何作用。且只能在设置 DATA, INDICATOR 属性时使用。

SET DESCRIPTOR 可以设置的描述符属性如下表所示。

表 6.1 SET DESCRIPTOR 可设置的描述符属性名

属性名	介绍
TYPE	ANSI 列类型码。详细参见表 6.2
LENGTH	列数据的最大长度
INDICATOR	对应列的指示符
DATA	对应要设置存放列数据的宿主变量

其中，ANSI 的列类型码见下表。

表 6.2 ANSI 数据类型编码

数据类型	列类型码
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2

REAL	7
SMALLINT	5
CLOB (oracle 模式下兼容)	112
BLOB (oracle 模式下兼容)	113

4．提取描述符

在执行 FETCH 操作后，可以通过 GET DESCRIPTOR 获取返回的查询结果。

语法格式如下：

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] <:desc_nam | string_literal>

   VALUE<item_number> :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH ;
```

参数说明：

- desc_nam 可以是字符串也可以是宿主变量；
- item_number 可以是数字也可以是宿主变量；
- hv1、hv2、hv3 必须是宿主变量；
- 可以只获取 DATA、INDICATOR、RETURNED_LENGTH 其中一个或多个。

GET DESCRIPTOR 可以查询的描述符相关属性如下表所示。

表 6.3 GET DESCRIPTOR 可以查询的描述符属性

属性	介绍
TYPE	类型码
LENGTH	列长度
OCTET_LENGTH	列按字符串输出最大字节数
RETURNED_LENGTH	实际数据长度
RETURNED_OCTET_LENGTH	列按字符串输出实际字节数
PRECISION	精度
SCALE	标度
NULLABLE	列是否可以为空
INDICATOR	指示数据是否为 NULL
DATA	指定数据存放宿主变量
NAME	列名
CHARACTER_SET_NAME	列的字符集

例如，下面的代码片段示例如何使用 ANSI 动态 SQL，分配了输入描述符‘in’与输出描述符‘out’，准备并执行一个查询语句，用 SET DESCRIPTOR 设置描述符，用 GET DESCRIPTOR 获取描述符的相关属性值。

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
char* dyn_statement = "SELECT name, personid FROM person.person WHERE
personid = :personid" ;
int personid_type_in = 3, personid_len_in = 4, personid_data_in = 10 ;
int name_type = 97, name_len = 50 ;
char name_data[51];
int personid_type = 3, personid_len = 4 ;
int personid_data ;
long SQLCODE = 0 ;
char     username[50];
char     password[50];
char     servername[50];
EXEC SQL END DECLARE SECTION;

void sql_error();

main()
{
/* Connect to DM. */
    strcpy(username, "SYSDBA");
    strcpy(password, "SYSDBA");
    strcpy(servername, "192.168.0.89:5289");

    EXEC SQL WHENEVER SQLERROR DO sql_error("DM error--");
```

```
    EXEC SQL CONNECT :username IDENTIFIED BY :password USING :servername;
    printf("\nConnected to dm as user: %s\n", username);



EXEC SQL ALLOCATE DESCRIPTOR 'in' ;
EXEC SQL ALLOCATE DESCRIPTOR 'out' ;
EXEC SQL PREPARE s FROM :dyn_statement ;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' ;
EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE = :personid_type_in,
   LENGTH = :personid_len_in, DATA = :personid_data_in ;
EXEC SQL DECLARE c CURSOR FOR s ;
EXEC SQL OPEN c USING DESCRIPTOR 'in';
EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out' ;
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :name_type,
    LENGTH = :name_len, DATA = :name_data ;
EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE = :personid_type,
    LENGTH = :personid_len, DATA = :personid_data ;


EXEC SQL WHENEVER NOT FOUND DO BREAK ;
while (SQLCODE == 0)
{
   EXEC SQL FETCH c INTO DESCRIPTOR 'out' ;
   EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :name_data = DATA ;
   EXEC SQL GET DESCRIPTOR 'out' VALUE 2 :personid_data = DATA ;
   printf("\nname = %s personid = %d", name_data, personid_data) ;
}
EXEC SQL CLOSE c ;
EXEC SQL DEALLOCATE DESCRIPTOR 'in' ;
EXEC SQL DEALLOCATE DESCRIPTOR 'out' ;
```

```
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

void
sql_error(msg)
char *msg;
{
    char err_msg[512];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

6.7 多线程支持

PRO*C 支持使用嵌入式 SQL 语句与指令开发多线程应用。

6.7.1 多线程应用的运行上下文环境

上下文环境资源包括一个或多个连接到一个或多个服务，一个或多个游标对应一个连接，以及它们的状态信息。不仅仅可以每个线程与连接使用多个上下文，也可以将上下文从一个线程传给另一个线程使用。

例如，一个多线程应用中定义一个线程 T1，执行了一个查询，返回前 10 行数据给应用，T1 结束。又定义一个线程 T2，将 T1 中使用的上下文 CONTEXT 传给 T2,在 T2 中使用同样的游标处理方式就能获取接下来的 10 行数据了。

6.7.2 上下文的两种使用方式

CONTEXT 在多线程应用中有两种使用方式：

1. 多个线程共享一个 CONTEXT；
2. 多个线程各自单独使用一个 CONTEXT。


注意：

不论使用哪种方式，在同一时刻多个线程不能使用同一个 CONTEXT。

- **多个线程使用同一个 CONTEXT，必须互斥访问**
- **多个线程各自使用一个 CONTEXT 时，可以各自自由使用**

6.7.3 多线程嵌入式 SQL 与指令

多线程应用中需要使用下面这些 SQL 与指令：

- EXEC SQL ENABLE THREADS;
- EXEC SQL CONTEXT ALLOCATE :context_var;
- EXEC SQL CONTEXT USE <:context_var | DEFAULT>;
- EXEC SQL CONTEXT FREE :context_var;

这些 EXEC SQL 语句中的 context_var 是上下文句柄，必须声明为 sql_context。语法格式为：

```
sql_context <context_variable>;
```

USE DEFAULT 表示使用默认的全局的上下文，直到下一个 CONTEXT USE 使用后才会切换上下文。

1. EXEC SQL ENABLE THREADS

该语句用来初始化多线程环境，必须是应用的第一条可执行的嵌入 SQL 语句。

2. EXEC SQL CONTEXT ALLOCATE

该语句用来初始化 CONTEXT 变量内存

3. EXEC SQL CONTEXT USE

该语句使用指定的运行上下文，使用之前必须先 ALLOCATE。EXEC SQL CONTEXT USE 指令与 C 规则一样有自己的范围，全局的上下文能在多个模块使用，局部的只能在当前模块使用。

例如，在下面的代码片段中，function2()中的 UPDATE 语句使用的是全局上下文 ctx1。

```
sql_context ctx1;            /* declare global context ctx1     */


function1()
{
   sql_context :ctx1;         /* declare local context ctx1      */
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...  /* local ctx1 used for this stmt   */
   ...
}


function2()
{
   EXEC SQL UPDATE ...       /* global ctx1 used for this stmt */
}
```

4. EXEC SQL CONTEXT FREE

该语句用来释放指定的 CONTEXT。

下面的两个代码片段分别示例使用 CONTEXT 的两种方式。

例 1，示例使用多个 CONTEXT，每个线程使用单独的 CONTEXT。

```
main()
{
   sql_context ctx1,ctx2;
EXEC SQL ENABLE THREADS;
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT ALLOCATE :ctx2;
   ...
   thread_create(..., function1, ctx1);
   thread_create(..., function2, ctx2);
   ...
   EXEC SQL CONTEXT FREE :ctx1;
   EXEC SQL CONTEXT FREE :ctx2;
   ...
}

void function1(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
   /* 使用 ctx 执行可执行 SQL 语句*/
   ...
}

void function2(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
   /* 使用 ctx 执行可执行 SQL 语句*/
   ...
```

```
}
```

例 2，示例多个线程使用同一个 context。因为 function1()和 function2()有可能同时执行，在使用 EXEC SQL 时应该使用互斥量。

```
mutex_t mutex;
main()
{
   sql_context ctx;
   EXEC SQL CONTEXT ALLOCATE :ctx;
   ...
/* spawn thread, execute function1 (in the thread) passing ctx  */
   thread_create(..., function1, ctx);
/* spawn thread, execute function2 (in the thread) passing ctx  */
   thread_create(..., function2, ctx);
   ...
}

void function1(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
   mutex_lock(&mutex);
/* Execute SQL statements on runtime context ctx.               */
   ...
   mutex_unlock(&mutex);
}

void function2(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
   mutex_lock(&mutex);
/* Execute SQL statements on runtime context ctx.               */
```

```
    ...
    mutex_unlock(&mutex);
}
```

6.7.4 多线程 PRO*C 程序注意事项

DM 提供的 PRO*C 是线程安全的，同时需要保证 PRO*C 应用编写也线程安全，如全局变量的使用等。除此之外需要注意以下两点：

- 对于静态或全局宿主变量要注意线程安全
- 避免在多个线程里同时使用同一个 CONTEXT

6.8 PRO*C 与 OCI 环境关联

DM 兼容 Oracle 的 SQLEnvGet 和 SQLSvcCtxGet 接口，可以用来把 OCI 环境句柄与 PRO*C 的 CONTEXT 上下文环境关联起来，进而可以在 PRO*C 应用中使用 OCI 句柄进行操作。

6.8.1 SQLEnvGet

SQLEnvGet 返回与 PRO*C 的 CONTEXT 关联的 OCI 环境句柄指针。

函数定义：

```
sword SQLEnvGet(
dvoid *rctx,
OCIEnv **oeh
);
```

功能说明：返回与 rctx 对应的 oeh OCI 环境句柄指针

参数说明：

- rctx：输入参数，上下文变量 CONTEXT,SQL_CONTEXT 类型
- oeh：输出参数，要返回的 OCI 环境句柄

返回值：成功返回 0，失败返回-1

6.8.2 SQLSvcCtxGet

SQLSvcCtxGet 用来获取与 PRO*C 上下文变量 sql_context 相关的 OCI 的上下文句柄，在 OCI 接口中能直接使用返回的 OCI 上下文句柄。

函数定义：

```
sword SQLSvcCtxGet(
dvoid *rctx,
text *dbname,
sb4 dbnamelen,
OCISvcCtx **svc
);
```

功能说明：返回与 rctx 关联的 svc 给 OCISvcCtx 句柄

参数说明：

- rctx：输入参数，sql_context 类型指针变量
- dbname：输入参数，逻辑上的连接名
- dbnamelen：输入参数，连接名长度
- svc：输出参数，OCISvcCtx 指针的地址

返回值：成功返回 0，失败返回-1

6.8.3 编写与 OCI 关联的 PRO*C 程序

编写与 OCI 关联的 PRO*C 程序的步骤主要如下：

1. include OCI 接口的头文件 DCI.h
2. 声明 OCI 环境句柄 OCIEnv *oeh
3. 声明 OCI 上下文句柄 OCISvcCtx *svc
4. 声明 OCI 的错误句柄 OCIError *err
5. 使用 PRO*C 连接数据库
6. 使用 SQLEnvGet 获取 OCI 的环境句柄，与 PRO*C 的上下文关联

   在这一步中，单线程应用与多线程应用获取 OCI 环境句柄的方式有一些不同：

   - 单线程应用：

```
retcode = SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
```

- 多线程应用：

```
sql_context ctx1;

...

EXEC SQL CONTEXT ALLOCATE :ctx1;

EXEC SQL CONTEXT USE :ctx1;

...

EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;

...

retcode = SQLEnvGet(ctx1, &oeh);
```

7. 使用获取到的环境句柄分配 OCI 错误句柄

```
retcode = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,

(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
```

8. 使用 SQLSvcCtxGet 获取 OCI 要使用的上下文句柄

在这一步中，单线程应用与多线程应用获取上下文句柄的方式有一些不同：

- 单线程应用：

```
retcode = SQLSvcCtxGet(SQL_SINGLE_RCTX, (text *)dbname, (ub4)dbnlen, &svc);
```

- 多线程应用：

```
sql_context ctx1;

...

EXEC SQL ALLOCATE :ctx1;

EXEC SQL CONTEXT USE :ctx1;

...

EXEC SQL CONNECT :uid IDENTIFIED BY :pwd AT :dbname USING :hst;

...

retcode = SQLSvcCtxGet(ctx1, (text *)dbname, (ub4)strlen(dbname), &svc);
```

6.8.4 PRO*C 中使用 OCI 实例

下面给出了一个在 PRO*C 中使用 OCI 的程序实例。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <DCI.h>

EXEC SQL INCLUDE SQLCA;
typedef struct _tagORAConnection
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
  OCISession *usrhp;
  OCISvcCtx *svchp;
  OCITrans* txnhp;                /* 事务句柄 */
  void *  ctx;                    /* pro*c 连接上下文*/
  FILE* fpTracefile;              /*trace 文件的句柄*/
} ORAConnection;

EXEC SQL BEGIN DECLARE SECTION;
char    username1[20],password1[20],servername1[50];
int * nrc;
#define BUFFERMAX  20971520   /* 缓存 1024*1024*20 */
unsigned char buffer[BUFFERMAX];
int nrsize;
OCIClobLocator *clob1;
OCIClobLocator *clob2;
int offset =1;
int nlength;
int errcnt=0;
EXEC SQL END DECLARE SECTION;
```

```
void
CHECKOK()
{
  if(sqlca.sqlcode!=0)
  {
     printf("fail! [%d] %s\n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
     errcnt = errcnt +1;
  }
}

int CLIAppInit_EC(char dbAlias[],
                  char user[],
                  char pswd[],
                  ORAConnection *conn)
{
  sword rc;

  rc = OCIHandleAlloc((dvoid *) conn->envhp, (dvoid **) & conn->srvhp,
                      OCI_HTYPE_SERVER, 0, (dvoid **) 0);
  if (rc != 0)
    return -1;

  rc = OCIHandleAlloc((dvoid *) conn->envhp, (dvoid **) & conn->errhp,
                      OCI_HTYPE_ERROR, 0, (dvoid **) 0);
  if (rc != 0)
    return -1;

  rc = OCIServerAttach(conn->srvhp, conn->errhp, (text *) dbAlias,
                       strlen(dbAlias), OCI_DEFAULT);
```

```
if (rc != 0)
{
  OCIHandleFree((dvoid *) conn->envhp, OCI_HTYPE_ENV);
  return -1;
}

OCIHandleAlloc((dvoid *) conn->envhp, (dvoid **) &conn->svchp,
               OCI_HTYPE_SVCCTX, 0, (dvoid **) 0);
OCIAttrSet((dvoid *) conn->svchp, OCI_HTYPE_SVCCTX, (dvoid *) conn->srvhp,
           0, OCI_ATTR_SERVER, conn->errhp);
OCIHandleAlloc((dvoid *)conn->envhp, (dvoid **) &conn->usrhp,
               OCI_HTYPE_SESSION, (size_t)0, (dvoid **) 0);
OCIAttrSet((dvoid *) conn->usrhp, OCI_HTYPE_SESSION, (dvoid *) user,
           (ub4)strlen(user), OCI_ATTR_USERNAME, conn->errhp);
OCIAttrSet((dvoid *) conn->usrhp, OCI_HTYPE_SESSION, (dvoid *) pswd,
           (ub4)strlen(pswd), OCI_ATTR_PASSWORD, conn->errhp);

rc = OCISessionBegin(conn->svchp, conn->errhp, conn->usrhp,
                     OCI_CRED_RDBMS, OCI_DEFAULT);
if (rc != 0)
{
  OCIHandleFree((dvoid *) conn->envhp, OCI_HTYPE_ENV);
  return -2;
}

OCIAttrSet((dvoid *)conn->svchp, OCI_HTYPE_SVCCTX,
           (dvoid *)conn->usrhp, 0, OCI_ATTR_SESSION, conn->errhp);
OCIHandleAlloc((dvoid *) conn->envhp, (dvoid **) &conn->txnhp,
               OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *) conn->svchp, OCI_HTYPE_SVCCTX,
```

```
            (dvoid *) conn->txnhp, 0, OCI_ATTR_TRANS, conn->errhp);


  printf("  Connected to %s.\n", dbAlias);
  return 0;
} /* CLIAppInit */


int main()
{
   FILE *fp;
   OCIEnv *oeh;
   OCISvcCtx *svc;
   OCIError *err;
   sword rc;
   ORAConnection *_conn;

   EXEC SQL BEGIN DECLARE SECTION;
   sql_context ctx;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL ENABLE THREADS;
   EXEC SQL CONTEXT ALLOCATE :ctx;
   EXEC SQL CONTEXT USE :ctx;
   printf("\n\n   BEGIN sqlenvget_main sqlenvget  \n");
   _conn = calloc(1, sizeof(ORAConnection));
   if (_conn == NULL)
   {
     printf("calloc failed");
     return -1;
   }
```

```
printf("%s\n","STEP11");
memset(servername1, 0, sizeof(servername1));
memset(username1, 0, sizeof(username1));
memset(password1, 0, sizeof(password1));
strcpy(username1,"SYSDBA");
strcpy(password1,"SYSDBA");
strcpy(servername1,"192.168.0.157:5290");

EXEC SQL CONNECT :username1 IDENTIFIED BY :password1 USING :servername1;

printf("%s\n","STEP12");
rc=SQLEnvGet(ctx, &_conn->envhp);
rc = CLIAppInit_EC(servername1,
                   username1,
                   password1,
                   _conn);
if (rc != 0)
{
  goto END_FLAG;
}
rc=SQLSvcCtxGet(ctx, (text *)0, (ub4)0, &_conn->svchp);

fp = NULL;
fp = fopen("data.txt","rb");
if( fp == NULL )
{
   printf("%s\n","open file failed!");
   *nrc = 2;
   goto END_FLAG;
}
```

```
EXEC SQL DROP TABLE DTMODULE;
EXEC SQL WHENEVER SQLERROR DO CHECKOK();
EXEC SQL ALLOCATE :clob1;
EXEC SQL ALLOCATE :clob2;

EXEC SQL CREATE TABLE DTMODULE(c1 int,c2 clob);
EXEC SQL INSERT INTO DTMODULE (c1,c2)VALUES(0,empty_clob());
EXEC SQL COMMIT;
EXEC SQL SELECT c2 INTO :clob1 FROM DTMODULE WHERE c1=0 FOR UPDATE;

offset = 1;
while((nlength =fread(buffer,1,BUFFERMAX,fp)) !=0 )
{
   EXEC SQL LOB WRITE :nlength FROM :buffer INTO :clob1 at :offset;
   offset = offset + nlength;
}
nlength=0;
EXEC SQL LOB DESCRIBE :clob1 GET LENGTH INTO :nlength;

fclose(fp);
EXEC SQL COMMIT;

EXEC SQL SELECT C2 INTO :clob2 from DTMODULE WHERE c1=0;

nlength=0;
EXEC SQL LOB DESCRIBE :clob2 GET LENGTH INTO :nlength;
printf("nlength=%d",nlength);

EXEC SQL COMMIT RELEASE;
```

```
END_FLAG:
   if(_conn->txnhp)
   OCIHandleFree((dvoid *) _conn->srvhp, (ub4) OCI_HTYPE_SERVER);
   if(_conn->srvhp)
   OCIHandleFree((dvoid *) _conn->srvhp, (ub4) OCI_HTYPE_SERVER);
   if(_conn->svchp)
   OCIHandleFree((dvoid *) _conn->svchp, (ub4) OCI_HTYPE_SVCCTX);
   if(_conn->errhp)
   OCIHandleFree((dvoid *) _conn->errhp, (ub4) OCI_HTYPE_ERROR);
   if(_conn->usrhp)
   OCIHandleFree((dvoid *) _conn->usrhp, (ub4) OCI_HTYPE_SESSION);
   if(_conn->envhp)
   OCIHandleFree((dvoid *) _conn->envhp, (ub4) OCI_HTYPE_ENV);
   free(_conn);
   _conn = NULL;

   printf("\n\n   END sqlenvget_main sqlenvget  \n");

   return(0);
}
```

6.9 修改当前连接的自动提交属性

DM PRO*C 提供 dpc_autocommit()和 dpc_autocommit_off()接口，用于修改当前连接的自动提交属性。

函数定义：

```
void dpc_autocommit()
```

功能说明：修改当前连接为自动提交

函数定义：

```
void dpc_autocommit_off()
```

功能说明：修改当前连接为非自动提交

7 PRO*C 程序实例

7.1 SELECT 语句

使用 SELECT 语句，可以将从数据库中检索到的值赋给相应的目标变量。

数据库管理系统执行一个 SELECT 语句，实际上是导出查询说明对应的导出表。查询说明是 SELECT 语句中滤除 INTO 子句后的对应语句。如果导出表的基数为 0(结果集为空集)，则返回 SQLCODE 为 100；如果基数大于 1，则返回 SQLCODE 小于 0；如果基数为 1，则返回 SQLCODE 为 0，相应的结果置于选择目标清单中。

如果查询结果多于一个元组，返回一个错误信息(SQLCODE<0)，此时应该使用游标查询。

例如，在人员信息表中，查询人员编号为‘1’的姓名、联系电话。

```
EXEC SQL BEGIN DECLARE SECTION;
varchar person_name[50];
varchar person_phone[25];
varchar person_id[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT name, phone
INTO :person_name, :person_phone
FROM person.person WHERE personid ='1';
......
```

7.2 插入、更新、删除语句

7.2.1 插入语句

例如，设有另一表 person_1(name, phone)。将人员信息表中人员编号为‘1’的姓名、联系电话插入到 person_1 中。

```
#include <stdio.h>
```

```
EXEC SQL BEGIN DECLARE SECTION;
char username[19];
char password[19];
char servername[19];
EXEC SQL END DECLARE SECTION;
void main( )
{
printf("Input Username: ");
scanf("%s", username);
printf("Input Password: ");
scanf("%s", password);
printf(" Please input servername :");
scanf("%s",servername);
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
EXEC SQL INSERT INTO person_1
SELECT name, phone FROM person.person
WHERE personid ='1';
EXEC SQL COMMIT WORK;
EXEC SQL LOGOUT;
}
```

7.2.2 更新语句

更新的方式有两种，一种是游标定位更新，另一种是搜索更新。

游标定位更新对游标所指的当前行的某些属性值进行更新。当打开一个游标时，游标指针处于游标表的第一行之前。因而应先执行 FETCH 操作，再对当前数据进行更新。

例如，在人员信息表中，使用游标定位更新修改人员编号为‘1’的姓名和联系电话。

```
#include <stdio.h>
EXEC SQL BEGIN DECLARE SECTION;
char username[19],password[19],servername[19];
```

```
varchar old_id[10],new_id[10];
varchar old_name[50],new_name[50];
varchar old_phone[25],new_phone[25];
EXEC SQL END DECLARE SECTION;
void main( )
{
strcpy(new_id,"1");
strcpy(new_name,"李丽 1");
strcpy(new_phone,"01088548562");
printf("Input Username: ");
scanf("%s",username);
printf("Input Password: ");
scanf("%s",password);
printf(" Please input servername :");
scanf("%s",servername);
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
EXEC SQL DECLARE C1 CURSOR FOR
SELECT name, phone
FROM person.person WHERE personid='1';
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOT FOUND GOTO endexit;
for(;;)
{
EXEC SQL FETCH C1 INTO:old_name,:old_phone;
printf("The current tuple is: %s %s", old_name,
old_phone);
......
/* 新的数值 new_no, new_name, new_addr */
EXEC SQL UPDATE person.person
SET name =:new_name, phone =:new_phone
```

```
WHERE CURRENT OF C1;

}

endexit:

EXEC SQL CLOSE C1;

EXEC SQL COMMIT WORK;

EXEC SQL LOGOUT;

}
```

例如，下面的例子使用搜索更新将人员编号为‘1’的元组的联系电话改为"01088548562"。

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;

char username[19],password[19],servername[19];

EXEC SQL END DECLARE SECTION;

void main( )

{

printf("Input Username:");

scanf("%s",username);

printf("Input Password:");

scanf("%s",password);

printf(" Please input servername :");

scanf("%s",servername);

EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;

EXEC SQL UPDATEperson.person

SET phone = '01088548562'

WHERE personid='1';

EXEC SQL COMMIT WORK;

EXEC SQL LOGOUT;

}
```

7.2.3 删除语句

删除的方式有两种，一种是游标定位删除，另一种是搜索删除。

定位删除是指将游标所指的当前行删除。

例如，游标查询人员信息表，询问是否要删除元组。

```
......
EXEC SQL DECLARE C1 CURSOR FOR
SELECT name, phone FROM person.person ;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOT FOUND GOTO endexit;
for(;;)
{
EXEC SQL FETCH C1 INTO :person_name,:person_phone;
printf(" Tuple is %s %s ", person_name, person_phone);
printf(" delete <Yes / No > ? ");
fflush(stdin);
scanf("%c",&select);
if(select==' y' || select=='Y')
EXEC SQL DELETE FROM person.person WHERE CURRENT OF C1;
}
endexit:
EXEC SQL CLOSE C1;
EXEC SQL COMMIT WORK;
EXEC SQL LOGOUT;
......
```

搜索删除是指对符合某种条件的元组进行删除操作。

例如，使用搜索删除将人员信息表中人员编号为‘1’的元组删除。

```
......
EXEC SQL DELETE FROM person.person WHERE personid='1';
EXEC SQL COMMIT WORK;
```

```
......
```

7.3 日期、时间数据类型的使用

日期、时间数据类型用于存储日期和时间信息，虽然日期和时间信息可以表示成字符串和数值数据类型，但日期和时间类型具有特殊的关联特性。下面我们通过一个例子介绍日期、时间、时间间隔类型的使用。

例如：

```
#include <stdio.h>
#include <stdlib.h>
EXEC SQL BEGIN DECLARE SECTION;
char username[19],password[19],servername[19];
int year1,month1,day1;
EXEC SQL END DECLARE SECTION;
void main( )
{
printf(" Please input Servername :");
scanf("%s",servername);
printf("Input Username: ");
scanf("%s", username);
printf("Input Password: ");
scanf("%s",password);
EXEC SQL LOGIN :username PASSWORD :password SERVER :servername;
/* 建存储日期时间的基表 */
EXEC SQL CREATE TABLE DATETIME1(
COL1 DATE,
COL2 TIME(6),
COL3 TIME(6),
COL4 TIMESTAMP(6),
COL5 TIMESTAMP(6),
```

```
COL6 INTERVAL YEAR TO MONTH,
COL7 INTERVAL YEAR(4) TO MONTH,
COL8 INTERVAL SECOND(6),
COL9 INTERVAL YEAR,
COL10 INTERVAL HOUR,
COL11 INTERVAL DAY(3) TO HOUR,
COL12 INTERVAL HOUR(5) TO SECOND(6)
);
/* 插入日期、时间值 */
EXEC SQL INSERT INTO DATETIME1
VALUES(DATE '1998-06-12',
TIME '08:09:20.123456',
TIME '09:10:21.234567',
TIMESTAMP '1999-07-13 10:11:22.345678',
TIMESTAMP '2000-08-14 11:12:23.456789',
INTERVAL '0022-11'YEAR TO MONTH,
INTERVAL '2000'YEAR,
INTERVAL '21.12345' SECOND,
INTERVAL '0055'YEAR,
INTERVAL '15'HOUR,
INTERVAL '110 22'DAY TO HOUR,
INTERVAL '19:22:10. 654321'HOUR TO SECOND
);
/* 更新日期、时间值 */
EXEC SQL UPDATE DATETIME1
SET COL1=(INTERVAL '0010-10' YEAR TO MONTH - INTERVAL
'0020' YEAR)*2+COL1;
/* 调用日期、时间函数 */
EXEC SQL SELECT EXTRACT(YEAR FROM COL1),
EXTRACT(MONTH FROM COL1),
```

```
EXTRACT(DAY FROM COL1)
INTO :year1,:month1,:day1
FROM DATETIME1;
printf("\n The day is %d-%d-%d \n",year1,month1,day1);
EXEC SQL COMMIT WORK;
EXEC SQL LOGOUT;
exit(0);
}
```

7.4 多线程

6.7 节中介绍了 DM 的 PRO*C 对多线程应用的支持，下面的例子生成了多个线程，这些线程使用各自的上下文环境并发执行。

```
#include <stdio.h>
#include <time.h>
#ifdef WIN32
#include <Windows.h>
#include <PROCESS.H>
#include <winbase.h>
typedef HANDLE     os_thread_t;
typedef  LPTHREAD_START_ROUTINE os_thread_fun_t;
#else
#include <pthread.h>
#endif
#define NUM_TH 10

struct parameters
{
   sql_context ctx;
   int  personid;
```

```
};
typedef struct parameters parameters;


EXEC SQL INCLUDE sqlca;
void select_person(parameters *param);


EXEC SQL BEGIN DECLARE SECTION;
       char servername[50];
       char username[50];
       char pwd[50];
EXEC SQL END DECLARE SECTION;


void
CHECKOK()
{
       if(sqlca.sqlcode!=0)
       {
              printf("fail! [%d] %s\n",sqlca.sqlcode,sqlca.sqlerrm.sqlerrmc);
       }
}


void
main()
{
    int i;
    int rt;
    sql_context ctx[NUM_TH];
    parameters param[NUM_TH];
#ifdef WIN32
    DWORD   code;
```

```
      os_thread_t thread_id[NUM_TH];
#else
    pthread_t thread_id[NUM_TH];
#endif


    printf("\n\n   BEGIN proc027_main1 dynamic bind number  \n");
       EXEC SQL ENABLE THREADS;
         strcpy(username, "SYSDBA");
    strcpy(pwd, "SYSDBA");
    strcpy(servername, "192.168.0.89:5236");


    for(i=0;i<NUM_TH;i++)
    {
       EXEC SQL CONTEXT ALLOCATE :ctx[i];
       param[i].ctx=ctx[i];
       param[i].personid=i+1;
#ifdef WIN32
       thread_id[i] =
_beginthreadex(NULL,1024*1024,(os_thread_fun_t)select_person,&param[i],0,NULL
);
#else
    rt = pthread_create(&thread_id[i],NULL,(void*)select_person,&param[i]);
#endif
    }
#ifdef WIN32
    while(TRUE)
    {
       for(i = 0; i < NUM_TH; i++)
       {
              rt = GetExitCodeThread(thread_id[i], &code);
```

```
            if(code == STILL_ACTIVE && rt != 0)     //if there is alive
thread, then sleep, and check again from beginning
                    break;
      }
      Sleep(100);
      if(i == NUM_TH)
            break;
    }
#else
   for(i=0;i<NUM_TH;i++)
   {
        pthread_join(thread_id[i],NULL);
   }
#endif

      printf("\n\n   END MAIN  \n");
}

void
select_person(parameters *param)
{
EXEC SQL BEGIN DECLARE SECTION;
      int          SQLCODE;
      char   SQLSTATE[6];
      SQL_CONTEXT ctx;
      int    personid;
      int     personid_out;
      char     name[51];
EXEC SQL END DECLARE SECTION;
```

```
  ctx=param->ctx;
  personid=param->personid;
  EXEC SQL CONTEXT USE :ctx;

/******************** BEGIN select_person  多线程并发 ********************/
       printf("\n  BEGIN select_person      \n");
       EXEC SQL CONNECT :username IDENTIFIED BY :pwd USING :servername;

  EXEC SQL SELECT personid,name into :personid_out,:name FROM person.person
where personid=:personid;

       CHECKOK();

       printf("personid_out=%d,name=%s
namelen=%d\n",personid_out,name,strlen(name));

       EXEC SQL CLOSE C1;
       EXEC SQL COMMIT RELEASE;
       EXEC SQL CONTEXT FREE :ctx;

       /******************** END select_person ********************/
}
```

附录 PRO*C 错误码汇编

运行预编译命令行工具 dpc_new.exe 时，可能会就预编译过程中的一些错误进行报错提示。

错误码码值域如下：

1）dpc 警告错误码值域为：（100,103）

2）dpc 错误错误码值域为：（-30000, -30015）

表 dpc 错误码

名称	代码	解释
DPC_WARN_NO_DATA	100	没有数据
DPC_WARN_STR_TRUNC	101	字符串截断
DPC_EC_INVALID_ITEM_AREA	103	描述空间不足
DPC_EC_STR_TRUNC	-30000	绑定字符参数截断
DPC_EC_INVALID_CURSOR_STATUS	-30001	无效的游标状态
DPC_EC_MEMORY_ALLOC_FAIL	-30002	内存分配失败
DPC_EC_INVALID_PARAM_INDEX	-30003	无效的参数信息，参数过大
DPC_EC_DESC_ALREADY_EXISTS	-30004	描述符已存在
DPC_EC_DESC_NOT_EXISTS	-30005	描述符信息不存在
DPC_EC_INVALID_PREPARE_NAME	-30006	无效的 Prepare 名称
DPC_EC_OUT_OF_MEMORY	-30007	内存分配错误
DPC_EC_INVALID_PARAM_TYPE	-30008	无效的参数类型
DPC_EC_INVALID_CONDITION_NUM	-30009	无效的序号
DPC_EC_PRE_STMT_NOT_CUR_SPEC	-30010	当前语句没有结果集
DPC_EC_INVALID_PRE_STATEMENT	-30011	无效的准备语句
DPC_EC_NOT_MATCH_COL	-30012	列不匹配
DPC_EC_NOT_MATCH_PARAM	-30013	参数不匹配
DPC_EC_PARAM_NEED_DATA	-30014	参数需要数据
DPC_EC_INVALIAD_COUNT	-30015	无效的个数

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