DBMaster

ESQL/C 程序员参考手册

SYSCOM

SYSCOM Computer Engineering Co./Corporate Headquarters

B1, 2-7F No. 115 Emei Street, Wanhua District,
Taipei City 108, Taiwan (R.O.C.)

www.dbmaker.com

www.dbmaker.com.tw/service

Document No.645049-237484/DBM54CN-M03312017-ESQL

发行日期: 2017-03-31

目录

1 简介

欢迎使用E*SQL*\C用户使用手册。*DBMaster*是一个功能强大且使用灵活的*SQL*数据库管理系统（*DBMS*），它支持交互式的结构化查询语言（SQL），Microsoft开放式数据库连接（ODBC）标准接口，以及嵌入式的ESQL/C语言。其独特的开放式结构和ODBC界面允许您通过使用多种编程工具来自由地构建客户应用程序，或者使用现有的*ODBC*应用程序来查询数据库。

DBMaster可以很容易地从个人使用的“单用户数据库”升级到企业级的“分布式数据库”。无论您使用的是单一使用者数据库还是商用数据库，DBMaster都能提供给您最先进的安全性、完整性和可靠性管理。广泛的跨平台支持特性则在您需要作硬件升级时，提供给您最佳的调整弹性。

*DBMaster*提供出色的多媒体处理能力来存储、查询、恢复和操作各种类型的多媒体数据。*DBMaster*中的二进制大型对象（*BLOBs*）通过其较高的安全性以及溃损恢复机制确保了多媒体数据的完整性。在源应用程序中编辑独立文件时，文件对象（*FOs*）用来管理多媒体数据。

本手册包含了*ESQL\C*的基本操作，并提供了系统说明以便您更好地管理数据库。用户手册的内容专为设计人员与数据库管理员编制，您仅需了解关系型数据库的工作方式即可。此外，用户还应具有在*Windows*和/或*UNIX*环境下操作的经验，本手册的信息同样适用于有经验的用户参考使用。

本手册展示了使用ESQL\C维护数据库的不同命令与过程。尽管手册适用于*Windows*环境下的*DBMaster*，但是它依然具备*UNIX*平台下的性能。为了更加清晰，我们使用了统一的示例贯穿整个手册。

*SQL*是双重功效的语言。它既作为能够访问数据库的交互式工具（交互式*SQL*），同时也是应用程序用来访问数据库的数据库编程语言。

一般情况下，所有主流的RDBMS都提供各自的用户界面来使用*SQL*，例如*DBMaster*提供的是dmSQL/C。用户可以直接在该工具中输入*SQL*语法来访问和维护数据库。

另外，大多数RDBMS同样为用户提供两个在程序中使用*SQL*的方法。分别为数据库应用程序界面和嵌入式*SQL*。DBMaster同时提供多种Java工具，详细内容请参考*其它相关文件*章节内容并选择您所需的手册。

1.1 其它相关文件

除了本书之外，我们还为您提供了其它用户手册和参考文献，帮助您更深入地了解DBMaster数据库管理系统。

您可以根据自己的需要，参考相关手册：

- 有关DBMaster的性能和功能信息，请参考*DBMaster指南*。
- 有关设计、管理和维护DBMaster数据库的信息，请参考*数据库管理员手册*。
- 有关管理DBMaster的相关信息，请参考*服务器管理工具用户手册*。
- 有关配置DBMaster的信息，请参考*配置管理工具用户手册*。
- 有关DBMaster功能的相关信息，请参考*数据库管理工具用户手册*。
- 有关DCI COBOL接口的相关信息，请参考*DBMaster DCI用户手册*。
- 有关dmSQL中使用的SQL语言，请参考*SQL命令与函数参考手册*。
- 有关dmSQL命令行工具使用的相关信息，请参考*dmSQL使用手册*。
- 有关ODBC API和JDBC API的相关信息，请参考*ODBC程序员参考手册和JDBC程序员参考手册*。
- 有关错误和警告信息的内容，请参考*错误信息参考手册*。

1.2 技术支持

在软件试用期间，Syscom电脑有限公司（“Syscom”）将为您提供30天的免费email支持和电话支持。当软件注册后，我们还会再为您提供30天的免费技术支持。如此一来，您就可以获得60天的免费支持。不仅如此，在您购买软件后，Syscom对任何问题都会以email的方式为您提供技术支持。

您除了可以获得免费的技术支持外，还可以以20%的零售价购买其它产品。要想获得更多的详细资料和价格信息，请与sales@dbmaker.com保持联系。

您可以通过任何一种方式（普通信件、电话或email）与Syscom技术支持保持联系，请登录至：www.casemaker.com/support 以获取详细信息。在与Syscom技术支持联系之前，请先查询当前数据库的常见问题解答。

无论您以何种方式与CASEMaker的技术支持联系时，请务必写上以下有效信息：

- 产品名称和版本号
- 注册号
- 注册的用户名和地址
- 供应商/发行者的地址
- 操作平台和计算机系统配置
- 错误发生前执行的动作
- 如果可以，请提供错误信息和编号
- 其它一些相关信息

1.3 文档协定

为方便用户的阅读和使用，本手册使用了标准的排版约定，注释、程序、示例和命令行都用缩进排版的方式进行了特别的设置。

协定	说明
斜体字	斜体字表示必须输入的信息占位符，例如用户名和表名。此字符应该用实际的名称来替换。有时，也会使用斜体字来介绍新的关键字，或强调重点。
黑体字	黑体字表示文件名、数据库名、表名称、字段名、用户名和其它数据库对象。它也用于强调程序执行步骤中的菜单命令。
关键字	文字段落中，SQL语言使用的关键字都是以大写字母出现的。
小符号	文档中出现的小写字符表示键盘上的按键，两个键名之间的加号（+）表示在按住第一个键不放的同时，再按第二个键。两个键名之间的逗号（，）表示释放第一个键以后，再按第二个键。
注意	包含一些重要的信息。
➲ **程序**	表示后面跟随的是程序的执行步骤或连续的项目。很多任务都是通过这种方式描述，给用户提供一个逻辑顺序步骤得以效仿。
➲ **示例**	例子用来阐明描述，通常包括屏幕上出现的文本，用户也可以将这些例子输入到计算机中，通过屏幕看到运行结果。当然，示例还包括一些原型和语法。
命令行	包括文本，这些命令都可以输入计算机中，显示在屏幕上。通常用于显示SQL命令的输入输出或dmconfig.ini中的内容。

表 1-1文档协定

2 ESQL基础

本章将介绍如何使用ESQL/C预编译器来编译ESQL和C程序。要在专有的应用程序中使用SQL可以通过使用应用程序接口实现，应用程序接口是一组函数调用的集合，它可向DBMS提交*SQL*语句并返回查询结果。*DBMaster*提供了符合工业标准的开放式数据库连接API-ODBC。

嵌入式SQL语言（ESQL）使用另外一种方法。SQL语句在格式上稍作改变，就可以直接写入应用程序的源代码。这种混合的源代码通过SQL预编译后可存储在数据库中，产生的主语言函数调用用来执行存储命令。

您可以编写C应用程序来通过ESQL命令访问DBMS，*DBMaster*的ESQL/C预处理程序为C编译器提供包含SQL命令的应用程序，然后预处理程序将SQL命令转换成C语句，通过C语言命令来执行数据库操作。

➲ **创建并且运行ESQL/C应用程序：**

1. 通过内嵌的SQL语言来设计并书写程序。
2. 使用DBMaster ESQL/C预编译器dmppcc来预处理程序。
3. 编译并链接由预处理器生成的程序。
4. 执行程序。

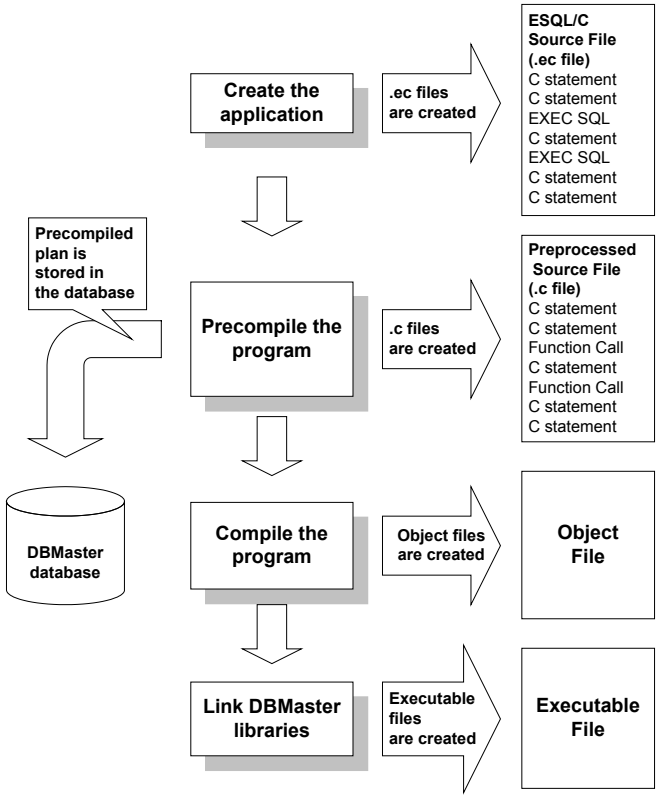

表 2-1 ESQL程序流程图

2.1 使用dmppcc完成预编译

*DBMaster*的预处理命令为*dmppcc*。输入的ESQL/C文件后缀为“.ec”，*dmppcc*输出的是一个C语言文件。在预编译阶段，*dmppcc*为每一个ESQL DML语句创建了一个存储命令，并且将源代码ESQL语句转化为函数引用以调用存储命令。

选项	评论
-d 数据库名	需要
-u 用户名	需要
-p 密码	需要
-o 输出文件	输出文件名，默认值为input_file.c。
-l 日志文件名	Dmppcc错误信息日志，默认值为stderr。
-j 项目名	如果没有指定，将使用模块名作为项目名。
-m 模块名	如果没有指定，将使用输入文件名作为模块名。
-s	将错误检验的单选项设置为on，默认值为off。
-cl	设置sql检查级别为limit，默认值为FULL。用户可以参照一个不存在的表。
-cs	设置sql检查级别的语法，默认为FULL。dmppcc只检查SQL语法而不检查表或字段信息。
-n	在数据库服务器端，Dmppcc将不存储SQL命令和执行计划。
-v	显示版本
-h	显示帮助信息
-sp	编译存储过程源

表 2-2 dmppcc命令选项

单独选择选项

➲ **示例1**

使用单独选择选项：

```
dmppcc -s ex1.ec
```

➲ **示例2**

开启单独选择选项并且在预处理时确认检查运行时间：

```
EXEC SQL SELECT salary FROM emp_table WHERE emp_id = 1000 INTO :var_salary;
```

如果单独选择错误检查项被开启，并且输出的结果记录数大于1，将会返回错误提示。

➲ **示例3**

返回的错误：

```
singleton SELECT can only retrieve at most one row
```

如果该检查项没有开启，那么单独选择查询将会返回第一条记录至主参数，而并不会检查其它记录。

SQLCHECK

当SQLCHECK选项设置为LIMIT (set -cl)，并且如果EXEC SQL语句中的表不存在，它能够使ESQL预处理器继续预处理ESQL/C程序，而不返回任何错误。

当SQLCHECK选项设置为SYNTAX (set -cs)，并且当用户在EXEC SQL中输入正确语法时，它能够使ESQL预处理器继续执行ESQL/C程序，而不返回错误。

它并不会检查表、字段或者安全性的相关语义信息。当该项检查开启时，dmppcc不会链接到数据库，但是用户仍要为dmppcc提供数据库名称来检查DBMaster配置文件的相关信息。

当该选项开启时，dmppcc不会存储数据库的预编译计划，dmppcc仍会检查语法错误。该选项主要为多用户环境设计，当多个人预处理不同的文件，而这些文件通过同一链接到一个可执行文件，并且测试执行文件功能，再

次预处理但并不会链接来测试可执行文件。用户可能会收到“可执行文件过期，请重新编译。”的错误提示。

如果您收到这类错误信息并且确定其他用户正在预处理这个文件或者当收到“锁超时”的错误信息时，请使用该选项。

➲ **语法**

```
FULL(default)/LIMIT(-cl)/SYNTAX(-cs)
```

强制预编译参数

➲ **示例1**

当预编译ESQL/C程序时，需要设置相关选项：

```
dmppcc -d test_db -u db_user_id -p db_user_passwd esql_source.ec
```

test_db –u =数据库名称，db_user_id＝用户名，db_user_passwd=用户密码

这些参数必须在命令行工具中使用或dmppcc将开启并查找本地的dmppcc.ini文件，您也可以在dmppcc.ini文件中写入dmppcc参数。

➲ **示例2**

dmppcc.ini文件的语法：

```
DATABASE = database_name
USER = user_id
PASSWORD = password
SELECT_ERROR = yes/no
OUTFILE = output_filename
LOGFILE = log_filename
PROJECT = project_name
MODULE  = module_name
SQLCHECK = FULL/LIMIT/SYNTAX
```

*dmppcc*创建的“.c”文件是一个普通的C源代码文件。您可以将其它“.c”或者“.o”文件合并来创建最终的可执行文件。

➲ **示例3**

使用一个预源码程序ex1.ec并且通过用户“john”和密码“johnspwd”来访问数据库**TESTDB**。

```
dmppcc -d TESTDB -u john -p johnspwd example.ec
```

➲ **示例4**

使用C编译器来编译输出文件ex1.c，并且通过ESQL库以及其它*DBMaster*数据库作为可执行来链接它：

```
cc -I. -I$INCDIR -c example.c
```

如果文件example.c通过*dmppcc*编译器进行了预处理并编译为example.o，我们可以通过其它对象文件作为可执行文件来链接。

➲ **示例5**

将–o作为可执行文件来链接其它文件。

```
acc -o driver example.o otherap.o  -L$LIBDIR -ldmapic -lm
```

您也可以参考在*DBMaster*的samples路径下的Makefile：

~*DBMaster*/$VERSION/samples/ESQLC。

3 ESQL语法

ESQL/C预处理程序会预处理ESQL 源程序中所有前缀为“EXEC *SQL*”或“$”的语句，SQL语句在C应用程序中可以放置在任意位置。然而，ESQL预处理程序不能在宏定义中执行EXEC SQL语句，而且也不会预处理头文件中的EXEC SQL语句。SQL语句必须有前缀“EXEC *SQL*” 或“$”，并且每个关键字必须在同一行中且以分号(;)结束。

➲ **示例**

```
if (c1 > 0) EXEC SQL COMMIT WORK;
```

3.1 静态/动态语法

如果SQL语句在预处理时是已知的，那么ESQL程序将使用静态的ESQL语法。当执行删除、插入、更新或者查询等操作，在相关表和查询条件已知时，只有输入参数的值在执行操作时会被更改，对于这种语法，*DBMaster*会自动检查其安全性，并将它编译为一个可执行计划，并且将该计划存储在数据库中。

如果SQL语句在编写和预处理时是未知的，那么将使用动态的ESQL。在第七章*安全性*中您可以了解到如何管理存储在数据库中的信息。

预处理时，当应用程序完全或者部分*SQL*语句未知时，可以使用动态ESQL语法。*SQL*语句是由用户提供的（例如dmsqlc），整体*SQL*语句可由查询工具（QBE）生成。*DBMaster*不会在预处理时为动态ESQL语句编译SQL语句。因此，所有编译和安全性检查操作都可在运行时间内执行。详细信息将在第6章*动态ESQL*中详细说明。

当C语句中参考到主变量时，处理方法与其它C变量的方法相同。当EXEC SQL语句参考主变量时，必须以冒号(:)开始。

➲ **示例1**

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
main ( )
    {
     EXEC SQL BEGIN DECLARE SECTION;
     char sql_string[255];
     EXEC SQL END DECLARE SECTION;

     EXEC SQL CONNECT TO testdb john johnspwd;

     /* get user input for SQL statement                                  */
     user_input(sql_string);
     /* Dynamic ESQL statement without host variable                      */
     EXEC SQL PREPARE statement_name FROM :sql_string;
     EXEC SQL EXECUTE statement_name;
     EXEC SQL DISCONNECT;
    }
```

➲ **示例2**

静态的ESQL语法：

```
EXEC SQL CONNECT TO database_name user_name password
EXEC SQL DISCONNECT [database_name]
EXEC SQL INCLUDE {DBENVCA | SQLCA | SQLDA}
EXEC SQL WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}
                    {CONTINUE| STOP | GO TO label| GOTO label | DO action}
EXEC SQL BEGIN DECLARE SECTION, EXEC SQL END DECLARE SECTION
EXEC SQL [AT DATABASE_NAME] any SQL statement
EXEC SQL [AT DATABASE_NAME] DECLARE cursor_name CURSOR FOR sql_query_statement
EXEC SQL [AT DATABASE_NAME] OPEN cursor_name [USING host_variable_[indicator]_list]
EXEC SQL [AT  DATABASE_NAME] FETCH cursor_name [INTO host_variable_[indicator]_list]
EXEC SQL [AT DATABASE_NAME] CLOSE cursor_name
```

➲ **示例3**

静态的ESQL/C程序格式：

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
main ( )
    {
     EXEC SQL BEGIN DECLARE SECTION;
     int  emp_id;
     char emp_name[20], emp_addr[50];
     EXEC SQL END DECLARE SECTION;
     EXEC SQL CONNECT TO testdb john johnspwd;
     /* get user input for host var                                    */
     user_input(&emp_id, emp_name, emp_addr);
     EXEC SQL INSERT INTO emp_table VALUES (:emp_id, :emp_name, :emp_addr);
     EXEC SQL DISCONNECT;
    }
```

3.2 变量

您可以从ESQL语句的主程序中使用主变量，在C应用程序和DBMaster数据库间传递数据。一个主变量通常与另一个被称为指示变量的变量相结合。当主变量被赋值时，如果该值为空或已被截断，那么这个指示变量将注册这个指定的值。

声明段

EXEC SQL中参考的任意主变量和指示变量都必须在声明段中声明，声明段由EXEC SQL语句中包含的C变量声明，由BEGIN DECLARE SECTION和END DECLARE SECTION组成。

一个应用程序中允许有任意数量的声明段。如果函数的主变量或指示变量在EXEC SQL语句中被参考，那么每一个函数都应该有其专门的声明段。如果在声明段中无法找到主变量和指示变量时，DBMaster的dmppcc将会返回一个错误提示。

➲ **示例**

声明段开始：

```
EXEC SQL BEGIN DECLARE SECTION;
varchar hoEmpNo[8];                          /* A host variable       */
int    inEmpNo;                              /* An indicator variable */
EXEC SQL END DECLARE SECTION;
```

主变量数据类型

一个单独C变量可以在主变量中声明。除非一维字符数组被用来指定字符缓冲的长度外，C结构和单元不允许出现在主变量中。fileobj数据类型不能用来指明一维数组，其它类型的C数组在一个单独的FETCH语句中为获得超过一个数值的一行集合时，可以作为一维数组。用户可以在一个单独的FETCH语句中使用二维数组来返回多个CHAR或者BINARY类型数值。

ESQL/C类型	类型定义	定义
char var_name[n]	char[n]	fix length char input (n) char output (n-1) char+ NULL terminate
binary var_name[n]	char[n]	fix length binary input (n) char output (n) char
short	short	short integer
int	int	integer
long	long	long integer
float	float	float
double	double	double

表 3-1 esqltype.h中的ESQL类型定义

下列SQL数据类型可以在声明段中使用：

ESQL/C类型	类型定义	定义
date	typedef struct date_s { short year; unsigned short month; unsigned short day;} eq_date;	Date
time	typedef struct time_s { unsigned short hour; unsigned short minute; unsigned short second; } eq_time;	Time

ESQL/C类型	类型定义	定义
timestamp	typedef struct timestamp_s { short year; unsigned short month; unsigned short day; unsigned short hour; unsigned short minute; unsigned short second; unsigned long fraction;} eq_timestamp;	Timestamp
varchar var_name[n]	typedef struct varchar_s {long len; char arr[n]; } varchar;	变量长度字符 输入：如果用户没有指明长度，那么遇空终止的字符串 输出：(n-1)个字符＋遇空终止
varcptr var_name[n]	typedef struct varcptr_s {long len; char *arr; } varcptr;	变量长度的字符，用户必须分配长度值。 输入/输出与varchar相同
varbinary var_name[n]	typedef struct varbinary_s {long len;char arr[n];} varbinary;	变量长度的二进制，用户必须分配长度值。 输入/输出与binary相同
varbptr	typedef struct varbpr_s { long len; /* must assign a buffer length */ char *arr; /* must assign a valid buffer ptr address */ } varbptr;	变量长度的二进制， 输入/输出与varbinary相同

表 3-2 SQL数据类型的声明段

Varchar类型是一个以空值结束的可变长度字符串，**varbinary**是可变长度的二进制字符串且不以空值结束。在**varchar**或**varbinary**的长度域内，通过分配实际长度来改变输入输出变量的长度。缓冲区的长度与缓冲地址不会因为**varcptr**或**varbptr**的类型而受到限制，请记得在程序中使用它们之前要先分配长度和地址。

对于非标准的C数据类型，*dmppcc*会将它们转化为C结构以供C编译器识别。例如varchar被分解为一个具有组长度和字符数组元素的C结构。

BLOB数据类型	类型定义	定义
Longvarchar	typedef struct longvarchar_s { long bufsize; char *buf; } longvarchar;	BLOB （文本数据）
Longvarbinary	typedef struct longvarbinary_s { long bufsize; char *buf; } longvarbinary;	BLOB (二进制数据)
fileobj	typedef struct fileobj_s { long type; char fname[MAX_FNAME_LEN]; } fileobj;	BLOB (文件对象) 默认类型是ESQL_STORE_FILE_CONTENT。您还可以将类型设置为ESQL_STORE_FILE_NAME只存储服务器端的文件名。

表 3-3 Blob数据类型

FILEOBJ文件对象

如果文件类型没有设置，那么默认的类型为ESQL_STORE_FILE_CONTENT，并且数据库服务器会存储用户指定的文件内容，即使文件在插入后被删除也没有关系。

然而，如果您设置的文件类型为ESQL_STORE_FILE_NAME，那么数据库仅会在fname字段存储指定的文件名；您务必确保数据库服务器可以找到并访问该文件。如果您删除了这个文件，当需要参考这个文件时，*DBMaster*会返回一个错误提示。文件类型的设置仅在fileobj类型作为输入参数时才会有作用。作为一个输出参数，数据库会不断尝试将数据复制到指定文件中。

➲ **示例1**

Fileobj类型作为输入主变量：

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE TABLE t1 (c1 file);
strcpy(fname1.name , "u:\image_path\test1.gif");
/* This INSERT statement will store all the content of file test.gif   */
EXEC SQL INSERT INTO t1 VALUES (:fname1);
fname1.type = ESQL_STORE_FILE_NAME;
strcpy(fname1.name , "u:\image_path\test2.gif");
EXEC SQL INSERT INTO t1 VALUES (:fname1);
```

➲ **示例2**

Fileobj类型作为表t2中c1的long varchar输出变量：

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
strcpy(fname1.name , "u:\image_path\test1.gif");
/* This SELECT statement will fetch the blob data from server site and put into user's
local file.   */
strcpy(fname1.name , "u:\local_path\test1.gif");
EXEC SQL select c1 from t2 into :fname1;
```

➲ **示例3**

使用游标时文件对象作为输出变量：

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
int idx1=0;
strcpy(fname1.name , "u:\image_path\test1.gif");
EXEC SQL DECLARE myCur1 CURSOR FOR select c1 from t2 into :fname1;
While (1)
{
idx1++;
/* This FETCH statement will fetch the blob data from server site and store it into
user's local file.  If you do not change output file name, in the next FETCH statement,
the output file  will be overwritten. */
sprintf(fname1.name , "test%d.gif", idx1);
EXEC SQL FETCH myCur1;
if (SQLCODE)
  { /* Break while loop when no more data or there's error */
  if (SQLCODE != SQL_SUCCESS_WITH_INFO)
         break;
  }
}
```

主变量

一旦主变量确定了，数据就可以被*DBMaster*存储在变量中以供应用程序调用。主变量同样可以用来插入、删除数据，或者用于WHERE与HAVING子句。

在C应用程序中使用主变量时需注意以下几点：

- 根据C语言的一般语法和DBMaster ESQL支持的数据类型语法来声明主变量。
- *INTO*子句的主变量不多于*SELECT*子句中字段名称的数量，并且输入或输出主变量的数据类型要与参数或投影字段的数据类型相兼容。
- 使用与字段数值类型相一致的主变量。

- 在SQL语句中使用主变量用冒号(:)作为前缀；而在应用程序中使用时则无需冒号。

➲ **示例1**

```
EXEC SQL BEGIN DECLARE SECTION;
char emp_id[20];                                /* employee ID           */
char emp_tel[20];                               /* employee telephone #  */
int indvalue = SQL_NTS;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT emp_id, emp_tel FROM emp_tab INTO :emp_id :indvalue,
:emp_tel :indvalue;
```

➲ **示例2**

```
EXEC SQL BEGIN DECLARE SECTION;
char emp_id[20];                                /* employee ID           */
char emp_tel[20];                               /* employee telephone #  */
int indvalue = SQL_NTS;  /* indicates input parameter is null terminated */
EXEC SQL END DECLARE SECTION;
strcpy(emp_id, "john Smith");
strcpy(emp_tel, "765-4321");
EXEC SQL INSERT INTO emp_tab VALUES (:emp_id :indvalue, :emp_tel :indvalue);
```

➲ **示例3**

使用输出主变量hoDeptNo与输入主变量hoEmpNo，已经参照select标准对代码进行初始化。

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, hoEmpNo;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT deptNo FROM Employee WHERE empNo = :hoEmpNo INTO :hoDeptNo;
```

变量作用域

以与其它C变量相同的方式，在声明段中的声明变量作用域为EXTERN或STATIC。

➲ **示例1**

要参考ESQL数据类型所支持的外部（全局）变量，需在ESQL变量声明之前增加“**extern**”。

```
EXEC SQL BEGIN DECLARE SECTION;
extern int var1;
EXEC SQL END DECLARE SECTION;
```

➲ **示例2**

要设置本地（静态）变量，需在ESQL变量声明之前增加“**static**”。

```
EXEC SQL BEGIN DECLARE SECTION;
static int var1;
EXEC SQL END DECLARE SECTION;
```

➲ **示例3**

要参考函数的输入变量值或者返回从一个ESQL或SQL查询语句中获得的值，复制变量数据或重新指定ESQL变量数据结构的指针。

```
/* Method 1, copy the value to esql host variable */
func1(int input_emp_id, char *output_telno)
    {
     EXEC SQL BEGIN DECLARE SECTION;
     int  emp_id;
     char telno[15];
     EXEC SQL END DECLARE SECTION;
     emp_id = input_emp_id;                  /* copy the input value */
     EXEC SQL SELECT telno FROM emp_tab WHERE emp_id = :emp_id INTO :telno;
     strcpy(output_telno, telno);
    }
/* Method 2, reassign the buffer pointer */
func1(char *input_emp_name, char *output_telno)
    {
     EXEC SQL BEGIN DECLARE SECTION;
     varcptr p_name, p_telno;
     EXEC SQL END DECLARE SECTION;
     p_name.len  = strlen(input_emp_name);   /* input string length  */
     p_name.arr  = input_emp_name;
     p_telno.len = 15;                       /* output string length */
     p_telno.arr = output_telno;
```

```
    EXEC SQL SELECT telno FROM emp_tab WHERE emp_name = :p_name INTO :p_telno;
  }
```

指示变量

指示变量是应用程序中处理空值和切断主变量的一种可选方法。使用指示变量时，指示变量在SQL语句中紧随主变量之后，请注意指示变量只能取整型数值。

指示变量indDeptNo直接跟在主变量hoDeptNo之后，并且以冒号为前缀。

➲ **示例1**

声明指示变量：

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, indDeptNo, hoEmpNo;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT deptNo FROM Employee
                      WHERE empNo = :hoEmpNo
                      INTO :hoDeptNo :indDeptNo;
```

指示值	数据库返回的值
SQL_NULL_DATA	返回NULL值，输出主变量是不确定的。
0或更大	指示变量将被设置为初始输出主变量缓冲区的长度。如果指示变量为获取BLOB数据，指示变量值设置为初始输出主变量的缓冲长度，以在数据库中优先获取blob数据。

表3-4指示变量的取值例1

➲ **示例2**

在数据库中使用指示变量来输入一个空值，相关的主变量可以忽略。

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, indDeptNo;
EXEC SQL END DECLARE SECTION;
inDeptNo = SQL_NULL_DATA;
```

```
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo :inDeptNo);
```

以上示例将会在字段hoDeptNo中插入空值，并且hoDeptNo的值将被忽略。

指示值	数据库接受的值
SQL_NULL_DATA	NULL值
0或更大	输入主变量的实际缓冲区长度，CHAR和BINARY类型。当提供了指示变量时，VARCHAR/VARBINARY数据类型中设置的主变量的长度可以忽略。
SQL_NTS	CHAR和BINARY类型的缓冲为空而中止。VARCHAR/VARBINARY数据类型中设置的主变量长度可以忽略。

表 3-5　指示变量的取值例2

3.3 状态代码

有三种声明方式应对于DBMaster的预处理器或运行程序所需要的特殊数据结构，这些声明可以在Include变量中使用。声明sqlca与sqlda的语法与声明dbenvca的语法相同。Sqlca用于传达状态码而sqlda用在动态ESQL中。

- dbenvca
- sqlca
- sqlda

dbenvca

Dbenvca声明是DBMaster在应用程序使用的环境变量，必须在程序中声明。

➲ 语法

```
EXEC SQL INCLUDE [EXTERN] dbenvca;
```

➲ 示例

```
file1.c
EXEC SQL INCLUDE DBENVCA;
main()
    {
     ...
     EXEC SQL .
     ...
    }
file2.c
EXEC SQL INCLUDE EXTERN DBENVCA;
func1()
    {
     ...
    }
```

SQLCA

每一条执行的*SQL*命令的状态代码都返回到*SQL*通信区(SQLCA)。*DBMaster*使用该数据结构中的变量将状态信息传递给C程序。以便如果出现问题，C程序将会对信息进行分析和处理。

声明SQLCA

SQLCA结构变量必须能够被整个ESQL/C程序所访问。SQLCA与DBENVCA都是全局变量，必须在ESQL/C源程序中声明。当SQLCA遇到下列语句时会自动被预处理器声明。

➲ 示例

自动声明SQLCA：

```
EXEC SQL INCLUDE [EXTERN] SQLCA;
```

SQLCA返回的状态

数据库服务器中的状态信息通过SQLCA返回，分析数据和处理错误或警告是应用程序的职责。

指示应用程序检查SQLCA的状态码并且处理错误和警告的途径有两种。您可以使用C代码或者SQL来编写命令，或使用WHENEVER命令，在C代码进行预处理时产生错误处理代码。

➲ 示例

SQLCA语法定义：

```
#define MAX_ERR_STR_LEN 256
/*----------------------------------------------------------------
* SQLCA - the SQL Communications Area (SQLCA)
*----------------------------------------------------------------*/
typedef struct sqlca
    {
     unsigned char  sqlcaid[8];          /* the string "SQLCA   "     */
     long           sqlcabc;             /* length of SQLCA, in bytes */
     long           sqlcode;             /* SQLstatus code            */
     long           sqlerrml;            /* length of sqlerrmc data   */
```

```
    unsigned char  sqlerrmc[MAX_ERR_STR_LEN]; /* name of object cause error */
    unsigned char  sqlerrp[8];          /* diagnostic information    */
    long           sqlerrd[6];          /* various count and error code */
    unsigned char  sqlwarn[8];          /* warning flag array           */
    unsigned char  sqlext[8];           /* extension to sqlwarn array   */
   } sqlca_t;
#define SQLCODE   sqlca.sqlcode   /* SQL status code  */
#define SQLWARN0 sqlca.sqlwarn[0] /* master warning flag */
#define SQLWARN1 sqlca.sqlwarn[1] /* string truncated */
```

DBMaster错误、警告以及其它代码都存储在sqlca.sqlerrd[0]中，获取的行数存储在sqlca.sqlerrd[3]中。当在FETCH语句中获取大于一行的记录时，您可以参考它们。更多详细信息请参考*错误信息参考手册*。

SQL代码	含义
SQL_SUCCESS or 0	成功
SQL_ERROR or (-1)	错误
SQL_NO_DATA_FOUND or 100	没有满足查询条件的记录
SQL_SUCCESS_WITH_INFO or 1	警告

表 3-6 示例代码

3.4 WHENEVER语句

WHENEVER语句可用于处理错误。当ESQL/C预处理器遇到WHENEVER语句时，会产生C语言错误处理代码，该错误处理代码的执行会依赖于*SQL*语句的输出。

每个条件的默认值都是CONTINUE。WHENEVER语句影响应用程序中所有紧随其后的*SQL*语句，一直到下一个相同情况的WHENEVER为止。换言之，最近的WHENEVER语句设定将影响所有紧随其后的EXEC SQL语句，直至文件结束，除非中间还出现其它的WHENEVER语句。

为了防止出现无限循环的可能，在错误处理中请不要忘记设置WHENEVER *check_case* CONTINUE选项，在其它包含EXEC *SQL*语句的函数中，如果EXEC SQL语句同样需要使用WHENEVER语句进行错误处理时，也同样需要设置。

在WHENEVER语句中使用这些条件来指导应用程序：

- *STOP* –回滚并断开与数据库的连接，并且当SQL语句的返回状态达到指定条件时，终止应用程序。
- *CONTINUE* –使之前*WHENEVER*语句的条件设置无效，并且继续执行直到引起错误的下一条语句。
- *GOTO label_name* –在您的应用程序中，为错误处理程序指示执行标签。
- *DO c_action_statement* –当返回状态与指定条件相同时，执行指定动作。

条件	描述
SQLERROR	当SQLCODE是SQL_ERROR
SQLWARNING	当SQLCODE是SQL_SUCCESS_WITH_INFO
NOT FOUND	当SQLCODE是SQL_NO_DATA_FOUND

表 3-7 示例代码

➲ **示例1**

```
WHENEVER SQLERROR DO break;
WHENEVER SQLWARNING DO print_warning();
 while ()
{
EXEC SQL …..;
EXEC SQL…..;
}
```

➲ **示例2**

```
int func()
    {
     ...
     EXEC SQL WHENEVER SQLERROR goto error_handle;
     EXEC SQL INSERT INTO emp_table VALUES (:emp_id, :emp_name, :emp_addr);
     ...
     ...
     return 0;
error_handle:
     EXEC SQL WHENEVER SQLERROR CONTINUE;
     printf("ERR:%s\n", sqlcode.sqlerrmc);
    return -1;
    }
```

4 数据操作

本章将给您提供应用程序中使用的ESQL示例，包括如何使用主变量、指示变量以及NULL值。您可以在交互式SQL的可用ESQL应用程序中输入同样的SQL语句，关于交互式的SQL，本手册也增加了一些描述。

在ESQL应用程序中，所有*SQL*命令都可以用EXEC *SQL*作为前缀，包括：COMMIT、ROLLBACK、CONNECT、DISCONNECT、INSERT、SELECT、UPDATE、DELETE…但是在ESQL应用程序中并不常使用DDL SQL语句。

ESQL中使用的附加嵌入式*SQL*语句示例都是声明类语句，例如：BEGIN (END) DECLARE SECTION、DECLARE CURSOR、INCLUDE以及WHENEVER，另外还有执行语句，例如：CLOSE、DESCRIBE、EXECUTE (IMMEDIATE)、FETCH、OPEN以及PREPARE。这些可执行语句只能在嵌入式*SQL*中使用。

4.1 数据操作

在进行INSERT、UPDATE以及DELETE操作时，只使用主变量用于将数据从应用程序传递到数据库中。除了在声明段中声明的主变量，在使用之前对每一个输入的主变量都要进行初始化。

在*SQL*语法中测试WHERE子句的NULL值关键字为'IS NULL'，不能使用指示变量来指示WHERE子句的NULL值。

➲ **示例1**

```
EXEC SQL BEGIN DECLARE SECTION;
int   hoDeptNo, inDeptNo;
varchar  hoDeptName[8];
EXEC SQL BEGIN DECLARE SECTION;
/* Use host variable hoDeptNo to input data into database              */
hoDeptNo = 1001;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo);
```

➲ **示例2**

在数据库中输入NULL值，忽略相应的主变量：

```
inDeptNo = SQL_NULL_DATA;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo :inDeptNo);
```

➲ **示例3**

使用SQL伪数据类型的主变量例如varchar，不是C语言中直接支持的：

```
strcpy(hoDeptName.arr, 'Human Resource');
hoDeptName.len = strlen(hoDeptName.arr);
hoDeptNo = 1001;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES (:hoDeptName, :hoDeptNo);
```

➲ **示例4**

使用UPDATE与DELETE输入主变量：

```
strcpy(hoDeptName.arr, 'Human Resource');
```

```
hoDeptName.len = strlen(hoDeptName.arr);
hoDeptNo = 1001;
EXEC SQL UPDATE Department SET DeptNo = :hoDeptNo WHERE DeptName = :hoDeptName;
EXEC SQL DELETE FROM Department WHERE DeptNo = :hoDeptNo + 1;
```

4.2 检索一行数据

在数据库中检索一行数据给输出的主变量，并将值传递给应用程序。在SELECT语句的INTO子句中使用输出主变量。

下例中的hoDeptName是一个输出主变量，hoDeptNo是一个输入主变量。指示变量inDeptName附属于输出主变量来检查查询返回是否为NULL或被截断的值，0值表示正常结果。

SQL_NULL_DATA说明主变量返回的是一个空值。当值大于0时意味着主变量被截断了，并且指示变量的值是初始值的长度。

例如，hoDeptName的长度是8，数据库中的结果长度为12，那么指示变量将会是12，并且主变量将包含初始值的前8个字符。

在嵌入的SELECT语句中，您可以拥有标准SQL子句（WHERE、GROUP BY、ORDER BY、HAVING…）的完整范围，您也可以在WHERE子句或者HAVING子句中使用输入主变量。

每一个SELECT语句准确地返回一行记录，如果一个查询返回超过一行记录时可使用游标（参见下文），在每一个SELECT语句后检查sqlca.sqlcode，查看返回多少行。如果只有SQLCA中的sqlcode为SQL_NO_DATA_FOUND，那么将不返回任何数据。当SELECT语句返回多行记录时，DBMaster预处理程序*dmppcc*将返回一个错误。

➲ **示例1**

```
EXEC SQL BEGIN DECLARE SECTION;
int   hoDeptNo, inDeptNo, inDeptName;
varchar  hoDeptName[8];
EXEC SQL BEGIN DECLARE SECTION;
hoDeptNo = 1001;
EXEC SQL SELECT DeptName FROM Department
          WHERE DeptNo = :hoDeptNo
            INTO :hoDeptName :inDeptName;
```

➲ **示例2**

如果设置了该选项，那么当结果大于一行时将会返回一个错误。

```
dmppcc -d TESTDB -u john -p johnspwd -s ex1.ec
```

4.3 事务处理

使用EXEC SQL COMMIT和EXEC SQL ROLLBACK命令来控制事务的完整性。高级应用程序使用SAVEPOINT与ROLLBACK TO SAVEPOINT选项来更好地进行数据处理。COMMIT WORK与ROLLBACK WORK会取消一个事务中的所有保存点。如果您没有使用COMMIT WORK以及ROLLBACK WORK来退出程序，那么事务中的所有变化都将被撤销。

您可以在配置文件中设置AUTOCOMMIT连接选项。如果AUTOCOMMIT连接选项设置为ON，那么所有已经执行的SQL语句都会立刻提交并且ESQL应用程序无法执行回滚操作。在执行应用程序之前，请确定将AUTOCOMMIT选项设置为off，或者在连接数据库后，使用下面的语法在ESQL源文件中转换AUTOCOMMIT选项的on/off。

➲ **示例**

使用SET AUTOCOMMIT ON/OFF选项：

```
EXEC SQL SET AUTOCOMMIT {ON|OFF}
```

4.4 动态连接语法

有时您可能会通过ESQL/C应用程序来访问多个数据库。可以编写几个ESQL程序，通过不同的数据库名进行预处理，然后链接所有程序作为可执行文件或者在SQL语句前增加“AT数据库名称”，在执行SQL语句前指明数据库名称。

➲ 示例

```
EXEC SQL BEGIN DECLARE SECTION;
 char db1[20];
 char usr1[10];
 char pwd1[10];
 int c1;
EXEC SQL END DECLARE SECTION;
/* GetDBInfo ()is user function which will pass back database name, user, password */
GetDBInfo(db1, usr1, pwd1);
EXEC SQL CONNECT TO :db1 :usr1 :pwd1;
EXEC SQL AT :db1 select c1 from t1 into :c1;
EXEC SQL DISCONNECT :db1;
```

4.5 使用游标

当一个查询返回多条结果时，程序必须以不同的方式执行查询。多行查询在操作上分为两步：程序开始查询但没有立即返回数据，然后在使用游标时，这些程序一次需要一条数据行。

应用程序中使用的游标是一个数据选择器，它可以对数据库中指定的行进行操作。下列操作演示的是DECLARE、OPEN、FETCH 以及CLOSE语句。

➲ **使用游标：**

1. 为游标分配空间、声明游标以及相关的**SELECT**语句。
2. 执行相关的**SELECT**语句，并开启游标。

   注意 *实际包括三个步骤：分析SQL语句、绑定主变量、开始执行语句。*
3. Fetch、delete或update主变量中的一行数据并且执行它。

   注意 *一直重复上述步骤到获取所有行为为止。*
4. 关闭游标。

声明游标

除返回单一记录的情形之外，应用程序中的每一条SELECT命令都要声明一个新游标。

INTO主变量子句同样可以在FETCH语句中定义。如果您想使用所有获取数据的可行方法（例如 FETCH NEXT、PREVIOUS、LAST、FIRST、ABSOLUTE以及RELATIVE），在声明一个游标时请指明SCROLL。

➲ **示例1**

DECLARE语法：

```
EXEC SQL DECLARE cursor_name [SCROLL] CURSOR FOR select_statement
        [INTO :output_host_var :indicator_var [, :output_host_var :indicator_var]]
```

➲ **示例2**

要在一个单独的FETCH语句中获取超过一行的记录，请指明SCROLL关键字。

```
EXEC SQL DECLARE vendCursor SCROLL CURSOR FOR
          SELECT vendorName FROM vendors
           WHERE vendorNumber = :inputNo;
```

➲ **示例3**

或者您可以在DECLARE语句中使用INTO子句。

```
EXEC SQL DECLARE vendCursor CURSOR FOR
          SELECT vendorName FROM vendors
           WHERE vendorNumber = :inputNo
            INTO :vendName;
```

开启一个游标

游标必须以开启状态来对指定的行内容进行操作，OPEN命令位于 DECLARE 命令中的游标名之前。

当执行OPEN命令后，游标找到并指向符合搜索条件的结果集的第一行。如果在游标中有输入主变量，您必须在OPEN语句之前或在OPEN语句中

为所有输入主变量指定值。以上示例中，因为输入主变量在DECLARE语句中做了定义，因此无需在OPEN游标语句中再次指明输入主变量。

➲ **示例1**

OPEN语法：

```
EXEC SQL OPEN cursor_name
       [USING :input_host_var [:indicator_var]
           [, :input_host_var [:indicator_var]]]
```

➲ **示例2**

开启游标的SQL命令：

```
EXEC SQL OPEN vendCursor;
```

使用游标返回数据

使用游标返回数据是通过FETCH命令来实现的。在循环语句中，FETCH在结果集中接近游标并且返回当前行，将SELECT列表中指定的字段值复制到INTO子句中指定的主变量中。

INTO子句在FETCH语句中可以被省略。nth_position以及num_rows可以作为一个主变量或者一个常整数。只有SCROLL选项定义了游标时，PREVIOUS、FIRST、LAST、ABSOLUTE nth_position、RELATIVE nth_position 以及 num_rows ROWS才可用。

您必须对每一行都使用一个FETCH命令以返回所有行。当结果集中的所有行都被获取后，DBMaster会将SQLCA的SQLCODE字段值设置为SQL_NO_DATA_FOUND以表示再没有找到更多的行。指示变量可同主变量一起声明以检测null值。

如果在FETCH语句中或是在INTO排列的主变量中，指定num_rows ROW语法，您可以在一个FETCH语句中获取更多的数据。

➲ **示例1**

FETCH语法：

```
EXEC SQL FETCH [NEXT | PREVIOUS | FIRST | LAST | ABSOLUTE nth_position | RELATIVE
nth_position] [num_rows ROWS]
         cursor_name
```

```
        [INTO :input_host_var [:indicator_var]
              [, :input_host_var [:indicator_var], ...]]
```

➲ **示例2**

从游标中获取结果的SQL命令：

```
EXEC SQL FETCH vendCursor INTO :vendName;
```

➲ **示例3**

获取行的命令：

```
EXEC SQL BEGIN DECLARE SETION;
int nrows;
int host1[50];
int host2[50];
char host3[50][100]; /* 50: means the maximum available fetched data values,100: means
the maximum data buffer length of each element. */
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE myCur SCROLL CURSOR FOR SELECT c1,c2,c3 FROM table1
INTO :host1, :host2, :host3;
EXEC SQL OPEN myCur;
nrows = 50;
/* loop until no data found */
while (SQLCODE != SQL_NO_DATA_FOUND)
{
     EXEC SQL FETCH :nrows ROWS myCur;  /* This will fetch 50 rows into host1, host2,
host3, because the maximum fetched  */
     for (j = 0; j < sqlca.sqlerrd[3]; j++) /* print out according to the number of
returned rows */
      printout(host1, host2, host3);
}
EXEC SQL CLOSE myCur;
```

➲ **示例4**

返回结果集的下一行，游标默认获取的值为NEXT。

```
EXEC SQL FETCH cur1;            /* default is fetch next */
EXEC SQL FETCH NEXT cur1 INTO :c1, :c2;
```

➲ **示例5**

PREVIOUS命令返回结果集的前一行。

```
EXEC SQL FETCH PREVIOUS cur1 INTO :c1, :c2;
```

➲ **示例6**

在结果集中，FIRST命令移动游标到第一行，并返回第一行。

```
EXEC SQL FETCH FIRST cur1 INTO :c1, :c2;
```

➲ **示例7**

LAST命令将游标移动到结果集的最后一行，并返回最后一行。

```
EXEC SQL FETCH LAST cur1 INTO :c1, :c2;
```

➲ **示例8**

ABSOLUTE n命令返回结果集中的第n行。如果n是一个负值，返回行将会是结果集中最后一行开始倒数的第n行。

```
n = 10;
EXEC SQL FETCH ABSOLUTE :n cur1 INTO :c1, :c2;
```

➲ **示例9**

RELATIVE n 命令返回在当前获取的行之后的第n行。如果n是一个负数，返回的行将会是游标所在位置倒数的第n行。

```
n = 10;
EXEC SQL FETCH RELATIVE :n cur1 INTO :c1, :c2;
```

使用游标删除数据

使用游标时，您一次只能删除表中的一行，任意其它行必须独立找回后再删除。

在连续删除时请不要使用COMMIT WORK命令，执行这个命令时会关闭游标并且终止删除。您可以在AUTOCOMMIT模式下实现这一功能。在使用DELETE或UPDATE WHERE CURRENT OF游标语句前请关闭AUTOCOMMIT模式。

➲ **删除一行：**

1. 使用SELECT命令声明游标。
2. 使用OPEN与FETCH命令开启游标并且指向被删除的行。
3. 执行删除命令。
4. 删除其它行时，使用FETCH命令重新确定游标位置。

➲ **示例**

```
EXEC SQL DELETE FROM supplier WHERE CURRENT OF vendCursor;
```

使用游标更新数据

您可以使用游标一次选择一行来进行更新，复位游标可更新其它行。

在连续更新时，请不要使用COMMIT WORK命令，并且确保AUTOCOMMIT模式处于关闭状态，否则它们都会关闭游标并且终止更新过程。

➲ **更新一行：**

1. 使用SELECT命令来声明游标。
2. 使用OPEN和FETCH命令来开启游标并且指向要被更新的行。
3. 执行UPDATE命令。
4. 要更新另一行时，请使用另一个FETCH命令来重新确定游标位置。

➲ **示例**

```
EXEC SQL UPDATE supplier SET price = price + 10
          WHERE CURRENT OF vendCursor;
```

关闭游标

COMMIT WORK和ROLLBACK WORK命令会关闭游标。您可以使用CLOSE CURSOR命令来明确地关闭一个游标。当您不需要游标时，可以关闭它以释放更多的资源。游标被删除后就不能再执行使用、开启或者取回命令了。

➲ **示例1**

关闭游标的语法：

```
EXEC SQL CLOSE cursor_name
```

➲ **示例2**

使用SQL命令关闭游标：

```
EXEC SQL CLOSE vendCursor;
```

5 BLOB数据

当在准备阶段BLOB数据大小未知，或者缓冲区不够大时，您需要每次PUT或GET部分BLOB数据，直到所有数据都被返回。

如果您可以给每一个程序分配足够的缓冲空间或者不在乎是否能够得到全部BLOB数据时，您就可以使用源ESQL语法来获取BLOB字段。

因为无需按照顺序来GET BLOB数据，我们可以分配一列来GET。使用GET BLOB语句时，数据无需从左列至右列按顺序被返回（小列数至大列数）。如果不需要，同样无需GET全部的BLOB数据。

PUT BLOB必须开启第一个至最后一个BLOB列，并且指明何时开启和停止。如果您没有PUT每一列BLOB数据或者没有指明何时结束PUT BLOB，那么BLOB字段将不会插入到数据库中。

因此当您断开连接或者从程序中退出时，操作将会终止。如果您断开数据库，最后一句被取消的错误就会显示。

5.1 PUT BLOB语句

➲ **使用PUT BLOB语句：**

1. PREPARE一个ESQL语句并且声明BLOB主变量作为一个问题的标记，以表示该BLOB主变量将会被限制。

   PREPARE语法如下：

```
EXEC SQL PREPARE stmt_name FROM "SQL SYNTAX"
| :sql_string_host_variable];
```

   Dmppcc会将“**SQL SYNTAX**”作为静态的ESQL语法处理，分析语法并且在预处理时存储执行计划。

   当它包含 :sql_string_host_variable时，Dmppcc会将PREPARE作为动态ESQL语法处理。语法在运行时才会被分析，请参考*动态ESQL*章节以获取更多详细内容。

   如果您要稍后使用GET或PUT BLOB语法来处理BLOB数据，请使用“?”作为BLOB的主变量的定义。

   将emp_pic插入到emp_table中：

```
EXEC SQL PREPARE stmt1 FROM

        "INSERT INTO emp_table (emp_id, emp_pic) VALUES
(:emp_id, ?)";
```

2. 执行这条ESQL语句：

```
EXEC SQL EXECUTE stmt_name;
```

   示例

```
EXEC SQL EXECUTE stmt1;
```

3. 声明何时开始输入BLOB数据：

```
EXEC SQL BEGIN PUT BLOB FOR stmt_name;
```

   示例

```
EXEC SQL BEGIN PUT BLOB FOR stmt1;
```

4. 定义在输入BLOB数据和填写bufsize与bufptr信息时，使用哪个BLOB变量。PUT BLOB命令必须是从第一列到最后一列未绑定的连续BLOB数据。

```
EXEC SQL PUT BLOB FOR stmt_name USING :host_var [:indicator_var]

/*host_var's data type must be longvarchar or longvarbinary)      */
```

示例

```
strcpy(buf, "This is a test");

b1.buf     = buf;

b1.bufsize = strlen(buf);

EXEC SQL PUT BLOB FOR stmt1 USING :b1;
```

5. 重复第四步直到没有BLOB数据可以PUT为止。

6. 如果要PUT的BLOB数据多于1列，请声明何时开始输入下一个BLOB列，并返回到第四步。

```
EXEC SQL PUT NEXT BLOB FOR stmt_name;
```

示例

```
EXEC SQL PUT NEXT BLOB FOR stmt1;
```

7. 如果整个BLOB列都已经PUT到数据库中，请声明PUT BLOB操作已经结束。

```
EXEC SQL END PUT BLOB FOR stmt_name;
```

示例

```
EXEC SQL END PUT BLOB FOR stmt1;
```

➲ 语法

完整的PUT BLOB语句：

```
 * Prepare an insert statement; the input BLOB columns should use a
 * question mark to denote it.
*****************************************************************/
EXEC SQL PREPARE stmt1 FROM "insert into emp_table \
```

```
             (emp_id, emp_pic, emp_memo) values (:id, ?, ?)";
id = 1000+j;
/*****************************************************************
* Execute this statement.
*****************************************************************/
EXEC SQL execute stmt1;

/*****************************************************************
* If there are 300 characters to put into emp_pic, and we want to
* put it 100 characters at a time.
*****************************************************************/
pic_buffer.bufsize  = 100;         /* max buffer size                */
/* you must allocate enough buffer as indicated in field bufsize, or
*point to a valid buffer pointer */
pic_buffer.buf  = user_buf;
/*****************************************************************
* Begin PUT BLOB.
*****************************************************************/
EXEC SQL BEGIN PUT BLOB FOR stmt1;
/*****************************************************************
* Loop 3 times to put data.
*****************************************************************/
for (i=0; i < 3; i++)
    {
     sprintf(pic_buffer.buf, "user's picture %dth's data........ ", i);
     EXEC SQL PUT BLOB FOR stmt1 USING :pic_buffer;
    }

/*****************************************************************
* Now start to put the next BLOB column's data.
*****************************************************************/
EXEC SQL put next blob FOR stmt1;
/*****************************************************************
* If there are 200 characters to put into emp_memo, and we want to
* put it 100 characters at a time.
*****************************************************************/
memo_buffer.bufsize  = 100;         /* max buffer size               */
/* you must allocate enough buffer as indicated in field bufsize, or
*point to a valid buffer pointer */
```

```
memo_buffer.buf  = user_buf;
/*****************************************************************
* loop 2 times to put data
*****************************************************************/
for (i=0; i < 2; i++)
    {
     sprintf(memo_buffer.buf, "user's memo %dth's data..........", i);
     EXEC SQL PUT BLOB FOR stmt1 USING :memo_buffer;
    }
/*****************************************************************
* end PUT BLOB
*****************************************************************/
EXEC SQL END PUT BLOB FOR stmt1;
```

5.2 GET BLOB语句

➲ **使用GET BLOB语句步骤如下：**

1. Prepare一个ESQL语句并声明BLOB主变量作为一个问题的标记（意味着BLOB主变量还没有限制）。

```
EXEC SQL PREPARE stmt_name FROM "SQL SYNTAX";
```

与动态ESQL语法有所不同的是，SQL SYNTAX在预处理时是已知的，并且必须包括限制的主变量名；如果是BLOB，在您要使用GET/PUT BLOB方法时，必须先定义'?'。

假设我们要从emp_table表中获取emp_pic：

示例

```
EXEC SQL PREPARE stmt1 FROM "select emp_pic from emp_table into ?";
```

2. 为准备好的语句声明一个游标：

```
EXEC SQL DECLARE cursor_name CURSOR FOR stmt_name;
```

示例

```
EXEC SQL DECLARE myCur CURSOR FOR stmt1;
```

3. 开启游标。

示例

```
EXEC SQL OPEN myCur;
```

4. 获取游标。

示例

```
EXEC SQL FETCH myCur;
```

5. 您无需声明何时开始GET BLOB，但您必须在BLOB主变量中定义列数以及变量缓冲的大小和有效的缓冲指示器。

```
EXEC SQL GET BLOB COLUMN :blobcol_num  FOR stmt_name USING :host_var
[:indicator_var]
```

语法1

```
EXEC SQL BEGIN DECLARE SECTION;

int nCol;

longvarchar b1;

EXEC SQL END DECLARE SECTION;

nCol = 1;         /* assign nCol as the column order in projection
*/

b1.bufsize = 50;  /* if you want to get 50 bytes at a time        */

b1.buf  = buf;    /* assign a valid buffer pointer to b1          */

EXEC SQL GET BLOB COLUMN :nCol FOR stmt2 USING :b1
```

注意 *在获取BLOB列之前，您可以指明一个指示器变量来获取剩余缓冲的大小。*

语法2

```
EXEC SQL PREPARE stmt1 FROM "select c6 from d1 into ?";

EXEC SQL EXECUTE stmt1;

/* fetch BLOB with size = 0 first, to know total size with indicator
*/

nCol    = 1;

b1.bufsize = 0;

b1.buf     = wkbuf;

EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;

if (SQLCODE == 0)

     printf("left size is %d\n", i_b1);

/* loop fetch BLOB with size = 2, and print out   "last" remain size
*/
```

```
for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
    {
     b1.bufsize = 2;       /* get 1 char plus null ptr at a time */
     EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
     if (SQLCODE != SQL_NO_DATA_FOUND)
          printf("b1 = %s, last remain size is %d\n", b1.buf, i_b1);
     chkErr();
    }
```

注意 *当您用DBMaster获取BLOB数据分配内存时，可以设置bufsize=DB_ALLOCATE_MEMORY。当您调用下一条FETCH或CLOSE CURSOR语句时， DBMaster会释放BLOB占用的内存。请小心使用该选项，因为给BLOB分配足够的内存可能会引起系统内存不够的问题。另外，在下一条FETCH或 CLOSE语句之后，由于数据库会释放内存，BLOB变量指示器将会指定一个无效的地址，这将会在您的程序中引起存储器清除或者一个错误的执行结果。*

6. 如果还有更多的BLOB数据要GET，请按照之前步骤继续。

示例

```
EXEC SQL PREPARE stmt1 FROM "select c6 from d1 into ?";
EXEC SQL EXECUTE stmt1;
/* fetch BLOB with size = 0 first, to know total size with indicator
*/
nCol    = 1;
b1.bufsize = 0;
b1.buf     = wkbuf;
EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
```

```
if (SQLCODE == 0)
     printf("left size is %d\n", i_b1);
/* loop fetch BLOB with size = 2, and print out   "last" remain size
*/
for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
    {
     b1.bufsize = 2;       /* get 1 char plus null ptr at a time */
     EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
     if (SQLCODE != SQL_NO_DATA_FOUND)
          printf("b1 = %s, last remain size is %d\n", b1.buf, i_b1);
     chkErr();
    }
```

7. 如果所有的数据都找回或者您不想GET剩下的BLOB数据，您可以继续FETCH剩下游标的结果缓冲区直至搜索不到其它行为止。

示例

```
#define MAX_BUF_SIZE 101
/* Prepare a SELECT statement, the output BLOB column should use    */
/* a question mark to denote it.                                    */
EXEC SQL PREPARE stmt1 FROM "select emp_id, emp_pic from emp_table
                                into :id, ?";
/* Declare a cursor to associate it with this statement.            */
EXEC SQL DECLARE myCur CURSOR FOR stmt1;
/* Open this cursor.                                                */
EXEC SQL OPEN myCur;
/* To get one MAX_BUF_SIZE character at a time, we must fill the
```

```
* following field first.                                          */
nCol = 2;                   /* second output column in projection.*/
pic_buffer.bufsize  = MAX_BUF_SIZE;   /* max buffer size          */
/* You must allocate enough buffer as indicated in field bufsize, or
   point to a valid buffer pointer.                               */
pic_buffer.buf  = user_buf;
/* loop fetch result.                                             */
while (1)
    {
     EXEC SQL FETCH myCur INTO :id, ?; /* fetch cursor            */
     if (SQLCODE)
         {
          if (SQLWARN0 != 'W')
              break;
         }
     printf("emp_id = %d\n", id);
     printf("emp_pic = ");
     /* loop get BLOB data until no data is found.                */
     for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
         {
          EXEC SQL GET BLOB COLUMN :nCol FOR stmt1
                              USING :pic_buffer :pic_ind;
          if (SQLCODE != SQL_NO_DATA_FOUND)
              {
```

```
            if (pic_ind == SQL_NULL_DATA)
                printf("(null)");
            else
                {
                 for(j = 0; j < MAX_BUF_SIZE-1; j++)
                 printf("%c", pic_buffer.buf[j]);
                }
           }
        }
     printf("\n");
    }

/* Close the cursor.                                          */
EXEC SQL CLOSE myCur;
```

6 动态ESQL

您可以通过两种方式书写ESQL语句，常用的最简单方法是静态的嵌入式语言，就是在预编译前将SQL语句作为源程序的一部分，这是静态的ESQL。

尽管静态ESQL非常有用，但是在您书写程序时必须清楚SQL语句的准确语法。一些应用程序需要具有调整SQL语句的功能以便响应用户在运行时的输入，动态ESQL则可以实现该功能。在动态ESQL中，程序将SQL语句作为字符串对待，并且在运行时传递给数据库。在预编译时，全部或部分动态ESQL语句都是未知的，完整的语句将在运行时在内存中构建。

动态ESQL语句根据是否是SELECT语句以及主变量是否明确来分为四种类型。每种动态ESQL使用不同的方法来编译ESQL语句。

类型	特征	方法
1	无查询，无输入主变量	立即执行
2	无查询，明确输入主变量的数目	预备/执行
3	查询，明确输入与输出主变量的数目	游标指针
4	不明确输入与输出主变量的数目	SQLDA

表 6-1ESQL语句分类

6.1 第一种动态ESQL类型

动态ESQL的第一种类型是没有输入主变量的非SELECT语句。在下面简单示例中，您可以使用EXECUTE IMMEDIATE来处理动态ESQL。

➲ **示例**

动态ESQL的第一种类型：

```
EXEC SQL BEGIN DECLARE SECTION;
varchar upd_str[100];
EXEC SQL END DECLARE SECTION;
sprintf(upd_str.arr, "UPDATE part SET qty = qty -1 WHERE ");
gets (update_condition);                    /* get dynamic upd condition */
strcat (upd_str.arr, update_condition);     /* construct dynamic SQL     */
upd_str.len = strlen(upd_str.arr);

EXEC SQL EXECUTE IMMEDIATE FROM :upd_str;   /* execute it                */
EXEC SQL COMMIT WORK;
```

6.2 第二种动态ESQL类型

动态ESQL的第二种类型稍微复杂一些，是已知输入主变量数的非SELECT语句。如果输入的主变量数在预处理时没有确定，那么您必须使用动态ESQL的第四种类型（见本章稍后章节）。

➲ **构建第二种动态ESQL类型的应用程序：**

1. 在声明段中声明所有输入主变量。
2. Prepare下列语句：

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

3. 为所有输入主变量和指示变量赋值。
4. 执行下列语句：

```
EXEC SQL EXECUTE statement_name USING :input_var1, :input_var2;
```

➲ **示例**

构建第二种动态ESQL类型应用：

```
EXEC SQL BEGIN DECLARE SECTION;
varchar del_str[80];
int  ord_number;
EXEC SQL END DECLARE SECTION;
char del_condition[80];
/* there is an input variable :iVord, it is a place holder               */
sprintf(del_str.arr, "DELETE FROM order WHERE ordid = :iVord AND ");
gets (del_condition);               /* get the DYNAMIC delete condition */
strcat (del_str.arr, del_condition); /* construct dynamic SQL            */
del_str.len = strlen(del_str.arr);
EXEC SQL PREPARE del_stmt FROM :del_str;
/* please note the relationship between the input host                   */
/* variable ord_number and placeholder iVord.                          */
gets (ord_number);                  /* set host variable value          */
EXEC SQL EXECUTE del_stmt USING :ord_number;
EXEC SQL COMMIT WORK;
```

6.3 第三种动态ESQL类型

动态ESQL的第三种类型是知道一定数量的输入输出主变量的SELECT语句。如果输入或输出主变量数在预编译时是未知的，就会成为第四种动态ESQL。

➲ **处理第三种动态ESQL:**

1. 在主变量中Prepare一个语句串，它包括一个对每一个输入变量都以“？”作为占位符的SQL语句。
2. 执行ESQL PREPARE语句，并指明语句名和语句串。

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

3. 执行ESQL DECLARE CURSOR语句，并指明指针名和语句名。

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
```

4. 为每一个输入变量指定一个值。
5. 为输入变量开启指针。
6. 在一个循环语句中，为输出变量返取回结果。
7. 关闭指针。

➲ **示例**

第三种动态ESQL类型的处理过程：

```
EXEC SQL BEGIN DECLARE SECTION;
varchar sel_str[100];
int  ord_num, ord_date, custor_num;
EXEC SQL END DECLARE SECTION;
/* 1. Put a build statement string inside a host variable, including one */
/*    placeholder for each input variable.                               */
gets(condition);
sprintf(sel_str.arr, "SELECT Ordid, Orddate FROM order WHERE CusId = :c \
                        AND %s", condition);
sel_str.len = strlen(sel_str.arr);
/* 2. Prepare the statement.                                             */
EXEC SQL PREPARE sel_stmt FROM :sel_str;
```

```
/* 3. Declare the cursor.                                                */
EXEC SQL DECLARE emp_cursor CURSOR FOR sel_stmt;
/* 4. Specify a value for each input variable.                           */
gets (custor_num);
/* 5. Open the cursor with a list of input variables.                    */
EXEC SQL OPEN emp_cursor USING :custor_num;
/* 6. In a loop, fetch the result to a list of output variables.         */
do
    {
     EXEC SQL FETCH emp_cursor INTO :ord_num, :ord_date;
     printf("ord_num = %d ord_date = %d\n", ord_num, ord_date);
    } while (sqlca.sqlcode == SQL_SUCCESS ||
             sqlca.sqlcode == SQL_SUCCESS_WITH_INFO)
/* 7. Close the cursor.                                                  */
EXEC SQL CLOSE emp_cursor;
```

6.4 第四种动态ESQL类型

动态ESQL的第四种类型是在预编译时没有定义输入输出主变量的SQL语句。该类型必须使用SQLDA描述符和主变量。第四种类型有许多步骤，然而在特定情况下一些步骤可以省略，比如输出主变量的数量未知，但输入主变量的数目是已知的，反之亦然。

SQLDA描述符

SQLDA描述符是一个区域，在这个区域中，应用程序和*DBMaster*存储每一个主变量的序号、值、长度、数据类型以及名称，并在动态ESQL语句中用来指示变量值。

SQLDA包含主变量的数量以及相关描述等信息。主变量的数量等于当前SQL语句中输入输出主变量的数目。描述信息包括两部分：字段信息和主变量信息。

描述命令

动态ESQL的第四种类型在用户输入语句前不知道SQL语句中包含的字段数，因此，应用程序不知道在数据库中需要传递多少个输入输出主变量的信息。

这就是为什么我们使用描述符区域在SQLDA中存储主变量信息的原因。动态ESQL准备好后，应用程序就会使用描述命令来获取*DBMaster*有多少个字段用来输入（DESCRIBE BIND VARIABLE）和多少个输出字段用于输出（DESCRIBE SELECT LIST），以及它们是什么。

描述命令会将输入的字段数（也就是主变量）与这些字段的数据放入描述符中。在循环语句中，应用程序会检查输入输出字段的类型，然后为这些主变量分配空间。

通过SQLDA传递信息

DBMaster支持两种分配和释放SQLDA的功能。

```
allocate_descriptor_storage()
free_descriptor_storage()
```

如果出现错误，信息将会存储到SQLCA中。

➲ **原型**

```
int allocate_descriptor_storage(long maxNumber, char **descriptor_name);
```

➲ **示例**

为一个SQLDA描述符'desc1'分配最大10个数。

```
char  *desc1;
long   maxNumber = 10;
/* SQLCODE is macro of sqlca.code                                          */
/* support error_handle() is a function for error handling                */
allocate_descriptor_storage(maxNumber, &desc1);
if (SQLCODE == SQL_ERROR) error_handle();
```

➲ **示例**

释放SQLDA描述符'desc1'：

```
free_descriptor_storage(desc1);
if (SQLCODE == SQL_ERROR) error_handle();
```

SQLDA包括主变量和字段信息。DBMaster 在描述时间内设置字段信息，主变量的信息由应用程序设置，DBMaster 或应用程序设置指示变量的信息。应用程序可以使用SetSQLDA() 函数来设定主变量的信息，并使用GetSQLDA()函数来获取字段信息。如果出现错误，信息将会存储在SQLCA 中。

➲ **原型**

SetSQLDA命令原型如下：

```
int SetSQLDA(char *descriptor_name, short host_variable_number,
             short option, long option_value);
```

➲ **原型**

GetSQLDA命令原型如下：

```
int GetSQLDA(char *descriptor_name, short projection_column_number,
             short option, void *option_value);
```

选项	描述
描述符名称	SQLDA描述符由allocate_descriptor_storage()分配
主变量数	主变量数目
映射字段数	映射字段列表数目
选项	一个关键字
选项值	有效的选项值

表 6-2 函数参数

当选项是SQLDA_NUM_OF_HV或SQLDA_MAX_FETCH_ROWS时，host_variable_number和projection_column_number没有使用。选项符号用于指定您要SET或者GET什么样的信息类型。

选项	描述
SQLDA_DATABUF	该选项值是数据缓冲指示器
SQLDA_DATABUF_LEN	该选项值是数据缓冲区的最大值
SQLDA_DATABUF_TYPE	该选项值是数据缓冲器的类型
SQLDA_BLOB_FLAG	指定主变量将以BLOB方法处理
SQLDA_PUT_DATA_LEN	该选项值是主变量输入数据的长度
SQLDA_INDICATOR	该选项值是指示变量值
SQLDA_MAX_FETCH_ROWS	该选项值是获取的最大行数
SQLDA_STORE_FILE_TYPE	该选项值是存储文件的类型（ESQL_STORE_FILE_CONTENT或ESQL_STORE_FILE_NAME）
SQLDA_COLTYPE	该选项值是主变量的字段类型
SQLDA_COLLEN	该选项值是主变量的字段长度
SQLDA_COLPREC	该选项值是主变量的精度
SQLDA_COLSCALE	该选项值是主变量字段的取值范围

选项	描述
SQLDA_COLNULLABLE	该选项值是可以包含空值的字段主变量
SQLDA_COLNAME	该选项是主变量的字段名
SQLDA_COLNAME_LEN	该选项值是主变量的字段名长度
SQLDA_NUM_OF_HV	该选项值是当前SQL语句包含的所有主变量数目

表 6-3 SET和 GET选项

选项	选项值
SQLDA_NUM_OF_HV	0 ~ 252的任意整数
SQLDA_MAX_FETCH_ROWS	正整数
SQLDA_DATABUF	有效指示器
SQLDA_DATABUF_LEN	正整数
SQLDA_DATABUF_TYPE	参见下面“数据缓冲类型”表
SQLDA_STORE_FILE_TYPE	ESQL_STORE_FILE_CONTENT ESQL_STORE_FILE_NAME
SQLDA_COLTYPE	参见下面“字段数据类型”表
SQLDA_COLLEN	与SQLDA_COLTYPE相关的正整数
SQLDA_COLPREC	与SQLDA_COLTYPE相关的正整数
SQLDA_COLSCALE	与SQLDA_COLTYPE相关的正整数
SQLDA_COLNULLABLE	SQL_NO_NULLS-字段不能为空值 SQL_NULLABLE-字段可以为空值
SQLDA_COLNAME	字符串
SQLDA_COLNAMELEN	1 ~ 18的任一整数

选项	选项值
SQLDA_INDICATOR	SQL_NULL_DATA –输入空数据 SQL_DEFAULT_PARAM –输入默认值 SQL_NTS – 遇空终止前输入数据 Positive integer –输入数据的实际长度
SQLDA_BLOB_FLAG	SQLDA_BLOB_ON SQLDA_BLOB_OFF
SQLDA_PUT_DATA_LEN	正整数

表 6-4 数据类型和可选值

SQLDA_DATABUF_TYPE的值将会告知*DBMaster*由SQLDA_DATABUF指定的数据缓冲区的数据类型。

SQLDA_DATABUF_TYPE	C TYPE OF SQLDA_DATABUF
SQL_C_CHAR	字符指示器（打印字符串）
SQL_C_LONG	长整型
SQL_C_SHORT	短整型
SQL_C_FLOAT	浮点型
SQL_C_DOUBLE	双字符型
SQL_C_BINARY	字符指示器（不打印字符串）
SQL_C_DATE	ESQL自定义类型‘eq_date’（参见esqltype.h）
SQL_C_TIME	ESQL自定义类型‘eq_ time’（参见esqltype.h）
SQL_C_TIMESTAMP	ESQL自定义类型‘eq_ timestamp’（参见esqltype.h）
SQL_C_FILE	字符串（SQLDA_DATABUF的值是一个文件名）

表 6-5 缓冲数据类型

SQLDA_COLTYPE	字段相关数据类型
SQL_CHAR	Char
SQL_VARCHAR	Varchar
SQL_LONGVARCHAR	Long varchar
SQL_BINARY	binary
SQL_LONGVARBINARY	Long varbinary
SQL_INTEGER	int
SQL_SMALLINT	smallint
SQL_REAL	float
SQL_DOUBLE	double
SQL_DECIMAL	decimal
SQL_DATE	date
SQL_TIME	time
SQL_TIMESTAMP	timestamp
SQL_FILE	file

表 6-6 字段数据类型

您可以使用GetSQLDA()获取所有相关选项的值。

只有下列选项可以被设定：

- SQLDA_COLTYPE
- SQLDA_COLLEN
- SQLDA_COLPREC
- SQLDA_COLSCALE
- SQLDA_COLNULLABLE
- SQLDA_COLNAME
- SQLDA_COLNAME_LEN

- SQLDA_NUM_OF_HV

下表列出的是SQLDA选项的设置方以及设置时间的详细信息。请记住应用程序第一次使用DESCRIBE来询问*DBMaster*字段信息。*DBMaster*会先设置字段信息，然后应用程序再设置主变量信息并且要求*DBMaster*执行该语句。

关键字	设置方	设置时间
SQLDA_NUM_OF_HV	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_DATABUF	应用程序	OPEN/EXECUTE前
SQLDA_DATABUF_LEN	应用程序	OPEN/EXECUTE前
SQLDA_DATABUF_TYPE	应用程序	OPEN/EXECUTE前
SQLDA_STORE_FILE_TYPE	应用程序	OPEN/EXECUTE前
SQLDA_COLTYPE	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_COLLEN	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_COLPREC	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_COLSCALE	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_COLNULLABLE	DBMaster	DESCRIBE BIND VARIABLES时
SQLDA_COLNAME	未使用	--
SQLDA_COLNAME_LEN	未使用	--
SQLDA_INDICATOR	应用程序	OPEN/EXECUTE前
SQLDA_MAX_FETCH_ROWS	应用程序	open前
SQLDA_BLOB_FLAG	应用程序	OPEN/EXECUTE前
SQLDA_PUT_DATA_LEN	应用程序	PUT BLOB前

表 6-7 输入主变量

描述符领域	设置方	设置时间
SQLDA_NUM_OF_HV	DBMaster	DESCRIBE SELECT LIST中
SQLDA_DATABUF	应用程序	FETCH前
SQLDA_DATABUF_LEN	应用程序	FETCH前
SQLDA_DATABUF_TYPE	应用程序	FETCH前
SQLDA_STORE_FILE_TYPE	未使用	DESCRIBE SELECT LIST中
SQLDA_COLTYPE	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLLEN	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLPREC	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLSCALE	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLNULLABLE	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLNAME	DBMaster	DESCRIBE SELECT LIST中
SQLDA_COLNAME_LEN	DBMaster	DESCRIBE SELECT LIST中
SQLDA_INDICATOR	DBMaster	FETCH中
SQLDA_BLOB_FLAG	应用程序	FETCH之前
SQLDA_PUT_DATA_LEN	未使用	--

表 6-8 输出主变量

应用程序步骤

构建动态ESQL应用程序的第四种类型时有许多步骤。如果您确定没有输入输出主变量时，则可以省略很多步骤。

➲ **构建动态ESQL应用程序的第四种类型：**

1. 声明描述变量。
2. 用maxNumber为动态输入输出主变量分配SQLDA。
3. 执行SQL PREPARE语句，指明语句名和语句串。

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

4. 为第三步中的语句声明游标。

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name; // step 4 and
5 for input host variable.
```

注意 *如果有输入主变量时，您只需要执行第5步即可。*

5. 描述第3步中的输入主变量，并将信息输入到已绑定的描述符中。

```
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name INTO
descriptor_name;
```

   a) 设置输入主变量的长度。
   b) 设置输入主变量的类型。
   c) 为输入主变量的值分配存储空间。
   d) 设置输入主变量的值。
   e) 设置输入指示变量的值。

6. 声明第4步时开启游标，并指明游标使用的描述变量。

```
EXEC SQL OPEN cursor_name USING DESCRIPTOR descriptor_name; // step
7 and 8 for output host variable.
```

7. 描述第三步语句中的输出映射字段并且将这些信息反馈给已绑定的描述符。

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
descriptor_name;
```

8. 设置输出主变量的长度和数据类型，并且为输出主变量的值分配存储空间。

9. 通过第4步中声明的游标获取数据，并且将这些值返回给已绑定的描述符的数据缓冲区。

```
EXEC SQL FETCH cursor_name USING descriptor_name;
```

10. 关闭游标。

```
EXEC SQL CLOSE cursor_name;
```

11. 释放SQLDA占用的存储空间（SQLDA中的pData域）。

12. 重新分配指示器。

➲ **示例1**

动态ESQL第四种类型的应用：

```
#define maxNumber 10
#define STRING_LEN 128
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
/* 0. declare descriptor variables                                    */
char *input_descriptor, *select_descriptor;
long  nHv=0, nCol=0;
char *pColName, *pData;
long  colType, colScale, colNullable;
long   colLen, colNameLen, dataType, colPrec;
/* connect to database                                                */
EXEC SQL CONNECT TO :dbname :user :password;
/* 1. allocate SQLDA for dynamic in/out host variables by maxNumber   */
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
/* 2. EXEC SQL PREPARE statement_name FROM :statement_string          */
gets(stmt_str.arr);
```

```
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
/* 3. EXEC SQL DECLARE cursor_name CURSOR FOR statement_name            */
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
/* 4. EXEC SQL DESCRIBLE BIND VARIABLES FOR statement_name INTO         */
/*    input_descriptor                                                  */
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
GetSQLDA(input_descriptor, 0, SQLDA_NUM_OF_HV, &nHv);
if (SQLCODE == SQL_ERROR) error_handle();
printf("There are %d returned input host variables: \n\n", nHv);
/* 5. set length of input host variables, set dataType of input host    */
/*    variables,allocate storage for value of input host variables, set */
/*    value of input host variables, set value of input indicates, set  */
/*    type, len, allocate buffer, value                                 */
for (i = 1; i <= nHv; i++)
    {
     pData = malloc(STRING_LEN);
     SetSQLDA(input_descriptor, i, SQLDA_DATABUF, pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     SetSQLDA(input_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
     if (SQLCODE == SQL_ERROR) error_handle();
     strcpy(pData, "dynamic ESQL/C example");
     datalen = strlen(pData);
     SetSQLDA(input_descriptor, i, SQLDA_INDICATOR, datalen);
     if (SQLCODE == SQL_ERROR) error_handle();
    }
/* 6. EXEC SQL OPEN cursor_name USING DESCRIPTOR input_descriptor       */
EXEC SQL OPEN demo_cursor USING DESCRIPTOR input_descriptor;
/* 7. EXEC SQL DESCRIBLE SELECT LIST FOR statement_name INTO            */
/*    select_descriptor                                                 */
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
GetSQLDA(select_descriptor, 0, SQLDA_NUM_OF_HV, &nCol);
if (SQLCODE == SQL_ERROR) goto error_label;
printf("There are %d returned columns: \n\n", nCol);
for (i = 1; i <= nCol; i++)
    {
     printf("column %d : \n");
     GetSQLDA(select_descriptor, i, SQLDA_COLNAME_LEN, &colNameLen);
     if (SQLCODE == SQL_ERROR) error_handle();
```

```
    GetSQLDA(select_descriptor, i, SQLDA_COLNAME, &pColName);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLTYPE, &colType);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLLEN, &colLen);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLPREC, &colPrec);
 if  (SQLCODE == SQL_ERROR) error_handing(); )
    GetSQLDA(select_descriptor, i, SQLDA_COLSCALE, &colScale);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLNULLABLE, &colNullable);
    if (SQLCODE == SQL_ERROR) error_handle();
    printf(" column name length = %ld \n", colNameLen);
    printf(" column name = %s \n", pColName);
    printf(" column type = %ld \n", colType);
    printf(" column length = %ld \n", colLen);
    printf(" column precision = %ld \n", colPrec);
    printf(" column scale = %ld \n", colScale);
    printf(" column nullable = %ld \n", colNullable);
   }
/* 8. set length of output host variables, set dataType of output host   */
/* variables, allocate storage for value of output host variable         */
for (i = 1; i <= nCol; i++)
   {
    GetSQLDA(select_descriptor, i, SQLDA_COLTYPE, &colType);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLLEN, &colLen);
    if (SQLCODE == SQL_ERROR) error_handle();
    switch (colType)
        {
         case SQL_CHAR:
              pData = malloc(colLen+1);
              SetSQLDA(select_descriptor, i, SQLDA_DATABUF, pData);
              if (SQLCODE == SQL_ERROR) error_handle();
              SetSQLDA(select_descriptor, i, SQLDA_DATABUF_LEN, colLen+1);
              /* '+1' for null terminate                                  */
              if (SQLCODE == SQL_ERROR) error_handle();
              SetSQLDA(select_descriptor,i,SQLDA_DATABUF_TYPE,SQL_C_CHAR);
              if (SQLCODE == SQL_ERROR) error_handle();
```

```
                break;
            case SQL_INTEGER:
                pData = malloc(4);
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF, pData);
                if (SQLCODE == SQL_ERROR) error_handle();
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF_LEN, 4);
                if (SQLCODE == SQL_ERROR) error_handle();
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_LONG);
                if (SQLCODE == SQL_ERROR) error_handle();
                break;
            }
        }
/* 9. EXEC SQL FETCH cursor_name USING select_descriptor                */
while (1)
    {
     EXEC SQL FETCH demo_cursor USING select_descriptor;
if (SQLCODE != SQL_SUCCESS && SQLCODE != SQL_SUCCESS_WITH_INFO)
         break;
     for (i = 1; i <= nCol; i++)
         {
          GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
          if (SQLCODE == SQL_ERROR) error_handle();
          GetSQLDA(select_descriptor, i, SQLDA_DATABUF_TYPE, &dateType);
          if (SQLCODE == SQL_ERROR) error_handle();
          switch (dataType)
              {
               case SQL_C_CHAR:
                    printf(" %s ", pData);
                    break;
                case SQL_C_LONG:
                    printf(" %ld ",*(long *)pData);
                    break;
              }
         }
    }   /* end of while loop */

/* 10. EXEC SQL CLOSE cursor_name                                       */
EXEC SQL CLOSE demo_cursor;
```

```
/* 11. free user buffer                                                    */
for (i = 1; i <= nHv; i++)
    {
     GetSQLDA(input_descriptor, i, SQLDA_DATABUF, &pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     free(pData);
    }
for (i = 1; i <= nCol; i++)
    {
     GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     free(pData);
    }
/* 12. De_allocate descriptor                                              */
free_descriptor_storage(input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
free_descriptor_storage(select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
```

➲ **示例2**

使用SQLDA返回一个FETCH语句中的多行数据。

```
#include "testdb.h"
/*************************************************************/
 *   The example will show how to write a dynamic ESQL program using SQLDA
 *   and to query with an unknown number of input and output host variables
 *
 *   Table customer in database STORE is used in the following demo example.
 *   Schema of table customer is (cid int, lname(32), fname(32)).
 *************************************************************/
#define MAX_ENTRY   10
#define STRING_LEN  128
#define CONDITION   103
#define MAX_FETCH_ROWS   10
#define NUM_OF_FETCH_ROWS 5
/*************************************************************
*  include header
*************************************************************/
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL INCLUDE SQLDA;
```

```
   EXEC SQL INCLUDE DBENVCA;

main()
{

   /*************************************************************
         *  error handling
*****************************************************/
   EXEC SQL WHENEVER SQLERROR GOTO error_label;

   /*************************************************************
         *  declare SQL host variables
*****************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
   varchar cuser[8], passwd[8];
   varchar demoquery[64];
   varchar democursor[8];
   char dbname[18];                  /* char type is fix length string */
   int     nFetchRows = NUM_OF_FETCH_ROWS;
   EXEC SQL END DECLARE SECTION;

   /*************************************************************
         *  declare variables
*****************************************************/
   int        i, rc = 0;
   char       fgConn = 0;
   long       datalen, colLen;
   long       nHv=0, nCol=0;
   char      *input_descriptor;
   char      *select_descriptor;
   char      *pData, *pColName;
   int       count;
      /*************************************************************
         *  connect to database
*****************************************************/
   strcpy(dbname, TEST_DBNAME);       /* get db,user,password info */
   strcpy(cuser.arr, "SYSADM");
   cuser.len = strlen(cuser.arr);
   strcpy(passwd.arr, "");
```

```
   passwd.len = strlen(passwd.arr);
      EXEC SQL CONNECT TO :dbname :cuser :passwd;
   fgConn = 1;
   EXEC SQL SET AUTOCOMMIT OFF;
      /*****************************************************************
         *  step1. allocate SQLDA storage for input and output host variables
*****************************************************/
   rc = allocate_descriptor_storage(MAX_ENTRY, &input_descriptor);
   if (rc < 0) goto error_label;
   rc = allocate_descriptor_storage(MAX_ENTRY, &select_descriptor);
   if (rc < 0) goto error_label;
      /*****************************************************************
         *  clear all tuples of table customer
*****************************************************/
   EXEC SQL DELETE FROM customer;
   /*****************************************************************
         *  insert data for test
*****************************************************/
   EXEC SQL INSERT INTO customer VALUES(1000, 'aaa', 'test1');
   EXEC SQL INSERT INTO customer VALUES(2000, 'bbb', 'test2');
   EXEC SQL INSERT INTO customer VALUES(3000, 'ccc', 'test3');
   EXEC SQL INSERT INTO customer VALUES(4000, 'ddd', 'test4');
   EXEC SQL INSERT INTO customer VALUES(5000, 'eee', 'test5');
   EXEC SQL INSERT INTO customer VALUES(6000, 'fff', 'test6');
   EXEC SQL INSERT INTO customer VALUES(7000, 'ggg', 'test7');
   EXEC SQL INSERT INTO customer VALUES(8000, 'hhh', 'test8');
   EXEC SQL INSERT INTO customer VALUES(9000, 'iii', 'test9');
   EXEC SQL INSERT INTO customer VALUES(10000, 'jjj', 'test10');
   EXEC SQL INSERT INTO customer VALUES(11000);
   exec sql commit work;

   /*****************************************************************
         *  specify the query demoquery with host variable, you can also ask
         *  user to input the query string from the terminal at run time
*****************************************************/
   sprintf(demoquery.arr, "%s %s",
                   "SELECT cid, lname, memo, memo2 FROM customer",
                   "WHERE cid > ?");
   demoquery.len = strlen(demoquery.arr);
```

```
    /*****************************************************************
       *  step2. prepare the query
*****************************************************/
   EXEC SQL PREPARE demo_stmt FROM :demoquery;
    /*****************************************************************
       *  step3. declare cousor for the query
*****************************************************/
   EXEC SQL DECLARE democursor SCROLL CURSOR FOR demo_stmt;
    /*****************************************************************
       *  step4. describle input host variables information (including number,
       *         type, length, ...) and put them into SQLDA input_descriptor
       *         for reference
*****************************************************/
   EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;

   rc = GetSQLDA(input_descriptor, 0, SQLDA_NUM_OF_HV, &nHv);
   if (rc < 0) goto error_label;
   printf("There are %d input host variables in the query \n\n", nHv);
    /*****************************************************************
       *  step5. set data type and buffer length of input buffer , and
       *         allocate it. Then, set these values.
       *         (note: assume the input data is character string less than
       *                128 bytes.)
*****************************************************/
   for (i = 1; i <= nHv; i++)
          {
          pData = MALLOC(STRING_LEN);
          rc = SetSQLDA(input_descriptor, i, SQLDA_DATABUF, pData);
          if (rc < 0) goto error_label;
          rc = SetSQLDA(input_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
          if (rc < 0) goto error_label;
          sprintf(pData, "%d", CONDITION);
          datalen = strlen(pData);
          rc = SetSQLDA(input_descriptor, i, SQLDA_INDICATOR, datalen);
          if (rc < 0) goto error_label;
          }
   /*****************************************************************
       *  step6. open cursor using data and information of
       *         SQLDA input_descriptor
```

```
*****************************************************/
   EXEC SQL OPEN democursor USING DESCRIPTOR input_descriptor;
   /************************************************************
         *  step7. describle output host variables information
         *         (including projection number, column name, column type,
         *         column length, ....) and put them into SQLDA select_descriptor
         *         for reference
*****************************************************/
   EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;

   rc = GetSQLDA(select_descriptor, 0, SQLDA_NUM_OF_HV, &nCol);
   if (rc < 0) goto error_label;
   printf("There are %d columns returned \n\n", nCol);
   printColInfo(select_descriptor);
   /*******************************************************
         *  step8. bind output host variables buffer information (including buffer
         *         type, buffer length) and allocate buffers for each output host
         *         variables.
*****************************************************/
   bindBufInfo(select_descriptor);
   /************************************************************
         *  step9. fetch data by cursor and print out the results, including:
         *         column name and data
*****************************************************/
   for (i = 1; i <= nCol; i++)
          {
          rc = GetSQLDA(select_descriptor, i, SQLDA_COLNAME, &pColName);
          if (rc < 0) goto error_label;
          printf("   %s    ", pColName);
          }
   printf("\n");
      for (i = 1; i <= nCol; i++)
          printf("========= ");
   printf("\n");

        do
                {
                EXEC SQL FETCH :nFetchRows ROWS democursor USING select_descriptor;
                if (sqlca.sqlcode == SQL_NO_DATA_FOUND)
```

```
                        break;
#if 0
                EXEC SQL GET BLOB column 2 FOR demo_stmt;
                EXEC SQL GET BLOB column 3 FOR demo_stmt;
#endif
                /* print out result by buffer data type */
                printResult(select_descriptor);
                printf("\n");
                } while(sqlca.sqlcode == SQL_SUCCESS ||
                            sqlca.sqlcode == SQL_SUCCESS_WITH_INFO);
        printf("\n");
   /*************************************************************
        *  step10. close cursor
*****************************************************/
        EXEC SQL CLOSE democursor;
error_label:
   /*************************************************************
        *  print out error information
*****************************************************/
        if (sqlca.sqlcode)
           {
           printf("SQLSTATE: %ld \n", sqlca.sqlcode);
           printf("error code: %ld \n", sqlca.sqlerrd[0]);
           printf("error message: %s \n", sqlca.sqlerrmc);
           }
  /*******************************************************
        *  step11. deallocate SQLDA storage
*****************************************************/
        for (i = 1; i <= nHv; i++)
           {
           rc = GetSQLDA(input_descriptor, i, SQLDA_DATABUF, &pData);
           FREE(pData);
           }
        for (i = 1; i <= nCol; i++)
           {
           rc = GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
           FREE(pData);
           }
        if (input_descriptor)
```

```
          {
          rc = free_descriptor_storage(input_descriptor);
          if (rc < 0)
                 {
                 printf("SQLSTATE: %ld \n", sqlca.sqlcode);
                 printf("error code: %ld \n", sqlca.sqlerrd[0]);
                 printf("error message: %s \n", sqlca.sqlerrmc);
                 }
          }
       if (select_descriptor)
          {
          rc = free_descriptor_storage(select_descriptor);
          if (rc < 0)
                 {
                 printf("SQLSTATE: %ld \n", sqlca.sqlcode);
                 printf("error code: %ld \n", sqlca.sqlerrd[0]);
                 printf("error message: %s \n", sqlca.sqlerrmc);
                 }
          }       /************************************************************
   *  disconnect from database
   ************************************************************/
       EXEC SQL WHENEVER SQLERROR CONTINUE;
       if (fgConn)
          EXEC SQL DISCONNECT;
}
/*************************************************************
 *  printColInfo() --
 *    print out column information from SQLDA by describle SELECT LIST
 *************************************************************/
void printColInfo(char *desc)
{
   int i, rc=0;
   long nCol=0;
   char *pColName;
   long   colType, colPrec, colScale, colNullable;
   long   colLen, colNameLen;
   rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);

   for (i = 1; i <= nCol; i++)
```

```
        {
        printf("column %ld information :\n", i);
        rc = GetSQLDA(desc, i, SQLDA_COLNAME_LEN, &colNameLen);
        rc = GetSQLDA(desc, i, SQLDA_COLNAME, &pColName);
        if (pColName != NULL)
                {
                printf("column name = %s \n", pColName);
                printf("column name length = %d \n", colNameLen);
                }

        rc = GetSQLDA(desc, i, SQLDA_COLTYPE, &colType);
        switch (colType)
                {
                case SQL_CHAR:
                   printf("column type = char \n");
                        break;
                case SQL_DECIMAL:
                        printf("column type = decimal \n");
                        break;
                case SQL_INTEGER:
                        printf("column type = integer \n");
                        break;
                case SQL_SMALLINT:
                        printf("column type = smallint \n");
                        break;
                case SQL_FLOAT:
                case SQL_REAL:
                        printf("column type = float \n");
                        break;
                case SQL_DOUBLE:
                        printf("column type = double \n");
                        break;
                case SQL_VARCHAR:
                        printf("column type = varchar \n");
                        break;
                case SQL_DATE:
                        printf("column type = date \n");
                        break;
                case SQL_TIME:
```

```
                        printf("column type = time \n");
                        break;
                case SQL_TIMESTAMP:
                        printf("column type = timestamp \n");
                        break;
                case SQL_LONGVARCHAR:
                        printf("column type = longvarchar \n");
                        break;
                case SQL_BINARY:
                        printf("column type = binary \n");
                        break;
                case SQL_LONGVARBINARY:
                        printf("column type = longvarbinary \n");
                        break;
                case SQL_FILE:
                        printf("column type = file \n");
                        break;
                default:
                        break;
                }

          rc = GetSQLDA(desc, i, SQLDA_COLLEN, &colLen);
          printf("column length = %ld \n", colLen);
          rc = GetSQLDA(desc, i, SQLDA_COLPREC, &colPrec);
          printf("column precision = %ld \n", colPrec);
          rc = GetSQLDA(desc, i, SQLDA_COLSCALE, &colScale);
          printf("column scale = %d \n", colScale);
          rc = GetSQLDA(desc, i, SQLDA_COLNULLABLE, &colNullable);
          if (colNullable == 0)
                  printf("column is not nullable \n");
          else
                  printf("column is nullable \n");
          printf("\n");
          }
}
/*************************************************************
 *  bindBufInfo() --
 *    set output host variables buffer information (including buffer type,
 *    buffer length) and allocate buffers for each output host variables.
```

```
 ***************************************************************/
void bindBufInfo(char *desc)
{
   int  i, rc=0;
   char *pData;
   long   nCol=0, colType, dataType;
   long   dataLen;
   int  nFetch = MAX_FETCH_ROWS;
   rc = SetSQLDA(desc, 0, SQLDA_MAX_FETCH_ROWS, nFetch);
   rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);
   for (i = 1; i <= nCol; i++)
          {
          rc = GetSQLDA(desc, i, SQLDA_COLTYPE, &colType);
          switch (colType)
                 {
                 case SQL_CHAR:
                 case SQL_VARCHAR:
                 case SQL_LONGVARCHAR:
                 case SQL_BINARY:
                 case SQL_LONGVARBINARY:
                 case SQL_FILE:
                 case SQL_DECIMAL:
                 case SQL_DATE:
                 case SQL_TIME:
                 case SQL_TIMESTAMP:
                    pData = MALLOC(STRING_LEN*nFetch);
                         dataLen = STRING_LEN-1; /* for null terminate */
                         dataType = SQL_C_CHAR;
                         break;
                 case SQL_INTEGER:
                         pData = MALLOC(4*nFetch);
                         dataLen = 4;
                         dataType = SQL_C_LONG;
                         break;
                 case SQL_SMALLINT:
                         pData = MALLOC(2*nFetch);
                         dataLen = 2;
                         dataType = SQL_C_SHORT;
                         break;
```

```
                case SQL_FLOAT:
                case SQL_REAL:
                        pData = MALLOC(4*nFetch);
                        dataLen = 4;
                        dataType = SQL_C_FLOAT;
                        break;
                case SQL_DOUBLE:
                        pData = MALLOC(8*nFetch);
                        dataLen = 8;
                        dataType = SQL_C_DOUBLE;
                        break;
                default:
                        break;
                }
          rc = SetSQLDA(desc, i, SQLDA_DATABUF, pData);
          rc = SetSQLDA(desc, i, SQLDA_DATABUF_LEN, dataLen);
          rc = SetSQLDA(desc, i, SQLDA_DATABUF_TYPE, dataType);
          }
}
/*************************************************************
 *  printResult() --
 *    print out column data by their data type
*************************************************************/
void printResult(char *desc)
{
   int   i,j, rc=0;
   long   nCol=0, dataType[MAX_ENTRY+1];
   long   *ind[MAX_ENTRY+1];
   long   *pind;
   char *pData[MAX_ENTRY+1];
   int   retRows = pSQLCA->sqlerrd[3];
   int   dataLen[MAX_ENTRY+1];
   int   maxFetch;
   rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);
   rc = GetSQLDA(desc, 0, SQLDA_MAX_FETCH_ROWS, &maxFetch);
   for (i = 1; i <= nCol; i++)
          {
      rc = GetSQLDA(desc, i, SQLDA_INDICATOR, &ind[i]);
          rc = GetSQLDA(desc, i, SQLDA_DATABUF_TYPE, &dataType[i]);
```

```
          rc = GetSQLDA(desc, i, SQLDA_DATABUF_LEN, &dataLen[i]);
          rc = GetSQLDA(desc, i, SQLDA_DATABUF, &pData[i]);
      }
   for (j = 0; j < retRows; j++)
      {
      for (i = 1; i <= nCol; i++)
         {
         if (*ind[i] == SQL_NULL_DATA)
            printf("NULL  ");
         else
            switch (dataType[i])
               {
               case SQL_C_CHAR:
                    printf(" %s ", pData[i]);
                    break;
               case SQL_C_LONG:
                    printf(" %l5d ", *(long *)pData[i]);
                    break;
               case SQL_C_SHORT:
                    printf(" %5d ", *(short *)pData[i]);
                    break;
               case SQL_C_FLOAT:
                    printf(" %f ", *(float *)pData[i]);
                    break;
               case SQL_C_DOUBLE:
                    printf(" %lf ", *(double *)pData[i]);
                    break;
               default:
                    break;
               }
         ind[i]++;
         pData[i] += dataLen[i];
         }
      printf("\n");
      }
}
```

➲ **将输入主变量的值传递给SQLDA，请参考下列步骤：**

- 为主变量分配有效的数据缓冲区，并且根据缓冲区的数据类型设置数据缓冲区的输入值。
- 设置数据缓冲区的指针、类型、长度以及指示器。

注意 *如果您不设置指示值，DBMaster就会使用数据缓冲区的长度来作为实际输入数据的长度。*

➲ **要获取SQLDA中输出主变量的值，请参考下列步骤：**

- 为主变量分配一个有效的数据缓冲区。
- 设置数据缓冲区的指针、类型以及长度。

6.5 动态ESQL BLOB界面

在动态ESQL中使用PUT BLOB或GET BLOB机制。另外，动态ESQL的第四种类型和静态ESQ在输入或获取BLOB数据时，使用相同的BLOB机制。

ESQL中有两种BLOB数据类型——内存存储与文件存储。存储在内存中的BLOB被当做BLOB数据，而存储在文件中的BLOB则作为文件对象来对待。动态ESQL的第四种数据类型如何PUT和GET BLOB数据和文件对象都将在本节稍后部分中介绍。除了动态ESQL第四种数据类型的常规步骤外，一些有关BLOB接口的额外步骤也必须考虑。

存储文件对象

您可以通过内容或对象名来存储文件对象。存储文件内容时，内容将作为数据存储到数据库中，并且存储后外部文件将不再与数据库相关。存储文件名时，文件名会以数据的形式存储在数据库中，而文件对象的内容仍然存储在外部文件中。

使用SetSQLDA()功能中的SQLDA_STORE_FILE_TY PE来指明存储的文件对象类型。用ESQL_STORE_FILE_CONTENT来设置option_value意味着SQLDA_DATABUF指定的文件内容将会存储在数据库中，使用ESQL_STORE_FILE_NAME来设置option_value意味着SQLDA_DATABUF指定的文件名称将存储在数据库中。在*DBMaster*中，SQLDA_STORE_FILE_TYPE的默认值是ESQL_STORE_FILE_CONTENT。

➲ **存储文件对象需要在SQLDA中进行如下设置：**

- 为文件名分配一个字符数据缓冲，文件名的最大长度为*MAX_FNAME_LEN = 79*，并且在数据缓冲区中设置文件名。
- 在SQLDA设置数据缓冲SQLDA_DATABUF的指针。
- 用SQL_C_FILE在SQLDA数据缓冲来设置数据缓冲类型SQLDA_DATABUF_TYPE。

- 在SQLDA使用ESQL_STORE_FILE_CONTENT或ESQL_STORE_FILE_NAME来设置文件类型SQLDA_STORE_FILE_TYPE。
- 在SQLDA中用实际文件名长度来设置指示变量SQLDA_INDICATOR。

➲ 示例

使用用户表**cid**、**cname**、**memo**：将数据从文件对象mary_memo.foPUT到“**memo**”字段。

```
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *input_descriptor;
char *pData;
long  datalen;
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "INSERT INTO customer VALUES(1, 'mary', ?)");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
pData = malloc(MAX_FNAME_LEN);
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_FILE);
if (SQLCODE == SQL_ERROR) error_handle();
/* Suppose we will put ‘file name’ into database */
SetSQLDA(input_descriptor. 1, SQLDA_STORE_FILE_TYPE, ESQL_STORE_FILE_NAME);
If (SQLCODE == SQL_SRROR) error_handing();)
/* Suppose mary's memo data is stored in file 'mary_memo.fo'            */
strcpy(pData, "mary_memo.fo");
datalen = strlen(pData);
SetSQLDA(input_descriptor, 1, SQLDA_INDICATOR, datalen);
if (SQLCODE == SQL_ERROR) error_handle();

EXEC SQL EXECUTE demo_stmt USING DESCRIPTOR input_descriptor;
```

```
/* suppose FreeSQLDA() can free SQLDA and all buffers allocated by application
*/
FreeSQLDA(input_descriptor);
```

获取文件对象

如果您想要获取一个文件对象，请在SQLDA中进行如下设置：

- 为文件名分配一个字符数据缓冲（文件名长度的最大值为MAX_FNAME_LEN = 79），并且在数据缓冲区中设置文件名（文件将会存储“**memo**”字段的数据）。
- 通过使用SQLDA_DATABUF命令来为SQLDA设置数据缓冲的指针。
- 在SQLDA中通过SQL_C_FILE设置数据缓冲类型SQLDA_DATABUF_TYPE。
- 在SQLDA中设置数据缓冲区的最大长度SQLDA_DATABUF_LEN。

在这个示例中，数据将会从“**memo**”字段取回并放置在文件对象（mary_memo.fo）中。

➲ 示例

```
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *select_descriptor;
char *pData;
long  datalen;
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "SELECT memo FROM customer WHERE cname = 'mary'");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
EXEC SQL OPEN demo_cursor;
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
pData = malloc(MAX_FNAME_LEN);
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF, pData);
```

```
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_FILE);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_LEN, MAX_FNAME_LEN);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(pData, "mary_memo.fo");
EXEC SQL FETCH demo_cursor USING select_descriptor;
/* support PrintFile() can print out content of file                    */
printf("mary memo content - " \n);
PrintFile("mary_memo.fo");
EXEC SQL CLOSE demo_cursor;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application
*/
FreeSQLDA(select_descriptor);
```

放置BLOB数据

如果您想使用动态ESQL的第四种类型来PUT BLOB数据，可在SQLDA中设置以下选项。

在使用EXECUTE命令前：

- 为输入数据分配一个数据缓冲。
- 在SQLDA中通过SQLDA_DATABUF命令为数据缓冲设置指针。
- 在SQLDA中使用SQLDA_DATABUF_TYPE命令来设置数据缓冲类型。
- 为SQLDA设置BLOB标记SQLDA_BLOB_FLAG，以指明数据库中将会PUT的主变量。
- 在“*BEGIN PUT BLOB*”之前将输入的数据填入数据缓冲。
- 指明数据SQLDA_PUT_DATA_LEN在SQLDA中的长度。

➲ **示例**

数据将从数据缓冲区Data PUT到“**memo**”字段。

```
#define maxbufsize 256
#define maxNumber 10
```

```
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *input_descriptor;
char *pData;
long  datalen;
boolean fgEnd = FALSE;
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "INSERT INTO customer VALUES(1, 'mary', ?)");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
pData = malloc(maxbufsize);
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_BLOB_FLAG, SQLDA_BLOB_ON);
if (SQLCODE == SQL_ERROR) error_handle();
EXEC SQL EXECUTE demo_stmt USING DESCRIPTOR input_descriptor;
EXEC SQL BEGIN PUT BLOB FOR demo_stmt;
/* support for mary's memo data is retrieved through function getInData(),  */
/* if no more input data will be retrieved, fgEnd will be assigned to TRUE  */
while(!fgEnd)
    {
     getInData(pData, maxbufsize, fgEnd);
     datalen = strlen(pData);
     SetSQLDA(input_descriptor, 1, SQLDA_PUT_DATA_LEN, datalen);
     if (SQLCODE == SQL_ERROR) error_handle();
     EXEC SQL PUT BLOB FOR demo_stmt;
    }

EXEC SQL END PUT BLOB FOR demo_stmt;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application */
FreeSQLDA(input_descriptor);
```

获取BLOB数据

如果您想使用动态ESQL的第四种类型来获取BLOB数据，可在SQLDA中设置以下选项。

在获取BOLB数据前：

- 为存储获取的数据分配一个数据缓冲。
- 在SQLDA中设置数据缓冲SQLDA_DATABUF的指针。
- 在SQLDA中设置数据缓冲类型SQLDA_DATABUF_TYPE。
- 在FETCH前设置数据缓冲区的最大长度SQLDA_DATABUF_LEN。
- 为存储的返回数据分配数据缓冲。
- 在*SQLDA*中设置数据缓冲指针。
- 在*SQLDA*中设置数据缓冲类型。
- 在*SQLDA*中设置BLOB标记来指明要获取数据的字段。
- 在*SQLDA*中设置数据缓冲的最大长度。
- GET BLOB数据前，指明GET DATA长度以获取SQLDA中的数据。

➲ 示例

在数据缓冲“**pData**”中，从“**memo**”字段中GET数据：

```
#define maxbufsize 256
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *select_descriptor;
char *pData;
long  datalen;
boolean fgEnd = FALSE;
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "SELECT memo FROM customer WHERE cname = 'mary');
stmt_str.len = strlen(stmt_str.arr);
```

```
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
EXEC SQL OPEN demo_cursor;
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
SetSQLDA(select_descriptor, 1, SQLDA_BLOB_FLAG, SQLDA_BLOB_ON);
if (SQLCODE == SQL_ERROR) error_handle();
EXEC SQL FETCH demo_cursor USING select_descriptor;
pData = malloc(maxbufsize);
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_LEN, maxbufsize);
if (SQLCODE == SQL_ERROR) error_handle();
printf("mary memo content - " \n);
do
        {
        EXEC SQL GET BLOB COLUMN 1 FOR demo_stmt;
If (SQLCODE ! = SQL_SUCCESS && SQLCODE != SQL_SUCCESS_WITH_INFO)
        Break;
        Printf(" %s ", pData);
        }
)
while (SQLCODE == SQL_SUCCESS || SQLCODE == SQL_SUCCESS_WITH_INFO)
    {
     EXEC SQL GET BLOB COLUMN 1 FOR demo_stmt;
     printf(" %s ", pData);
    }
EXEC SQL CLOSE demo_cursor;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application */
FreeSQLDA(select_descriptor);
```

➲ **在使用GET BLOBGET字段数据前，如何GET字段的总数据长度：**

- 设置SQLDA_DATABUF_LEN为0。
- 从字段中GET BLOB数据。
- 获取SQLDA_INDICATOR的值，该值是字段全部数据的长度。

7 项目与模块管理

当使用*DBMaster* ESQL/C处理器*dmppcc*来预处理一个ESQL/C源文件时，需要提供数据库名、用户名和密码。在连接数据库和预处理文件时，用于预处理ESQL/C源文件的用户名必须拥有connect和resource权限。

如果在ESQL/C应用程序声明段中已经声明，那么dbname、dbuser以及dbusr_passwd将作为标识符或者字符类型的主变量。如果用户没有设置密码，那么dbusr_passwd可以被忽略。在预处理和执行ESQL/C程序时，不必使用相同的数据库用户名。例如，银行数据库用户“**acc_dba**”开发“**bank**”程序。

➲ **语法**

```
dmppcc -d test_db -u db_user_id -p db_user_passwd esql_source.ec
```

➲ **示例1**

当编写ESQL/C源文件时，您需要在源文件中增加CONNECT语句，使得应用程序能够连接到数据库。

```
EXEC SQL CONNECT TO dbname dbuser dbusr_passwd;
```

参数	语法
Dbname	[identifier | :host_variable_identifier]
Dbuser	[identifier | :host_variable_identifier]
dbusr_passwd	[| identifier | :host_variable_identifier]

表7-1 Connect语句参数和语法

➲ **示例2**

在shell命令中，键入银行数据库中的用户acc_dba。

```
dmppcc -d bank -u acc_dba -p acc_dba connect.ec
```

➲ **示例3**

以下为连接文件connect.ec的内容：

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
connect()
    {
     /* use the clerk account to connect to the bank database           */
     EXEC SQL CONNECT TO bank clerk pd_clerk;
     if (SQLCODE) return SQLCODE;
     ...
     do_clerk_operation();
     ....
    }
```

7.1 项目和模块管理

当预处理ESQL/C源文件时，DBMaster会创建一个执行计划并且将所有相关信息存储在数据库一个被称为模块的组件中。如果您没有指明模块名称，那么默认的模块名将会是ESQL/C源文件名。

当用户试图访问一个旧版的执行计划时会产生错误。当其他ESQL开发者再次预处理相同的ESQL程序时，dmppcc会删除先前存储的计划，并且创建一个新的存储计划。如果您在执行ESQL程序时经常遇到“执行可能已经过期，请重新构建”的错误提示，那么您应该编译相关的.c文件并且重新连接执行。

然而，尽管这是个可执行版本的错误，但是当许多研发人员在同一个ESQL应用程序中开发不同ESQL模块时，可能仍想忽略该错误。如果您处于开发阶段，则可以使用“-n”选项来忽略这个错误信息。为了使性能最优化并且减少ESQL项目管理的问题，您应当在完成程序编码后删除“-n”选项。

任何研发人员的应用系统都可以包含多个ESQL/C模块。如果研发人员试图单独管理（赋予/收回或者删除权限）每一个模块时，负担将会加重。项目的目的就是让研发小组一起完成ESQL/C模块，并且更容易地组织应用系统。在预处理ESQL/C源文件后，dmppcc会存储项目名和模块名。如果没有指明项目名称，默认的项目名会和模块名相同。

在预处理任何ESQL源文件时，如果数据库中不存在项目，那么DBMaster会自动创建一个项目来存储模块。如果项目已经存在，那么DBMaster会自动为该项目关联一个新模块，每一个模块只能和一个项目关联。

您可以通过SQL语句来参考数据库的系统表SYSPROJECT，以查看ESQL的项目和模块信息。

➲ **示例**

```
select * from SYSPROJECT;
```

字段	含义
PROJECT_NAME	ESQL项目名
PROJECT_OWNER	ESQL项目所有者
MODULE_NAME	ESQL模块名
MODULE_OWNER	ESQL模块所有者
MODULE_SOURCE	模块源文件名
REF_CMD	相关命令数量（dmppcc内部使用）

表 7-2 SYSPROJECT系统表

ESQL执行计划的信息存储在SYSCMDINFO系统表中，系统表不仅存储ESQL执行计划，还包括其它存储命令和存储程序的执行计划。您可以参考存储命令手册以获取更多信息。

如果您在dmppcc中设置选项–cs 或 –n，DBMaster将不存储执行计划、模块、项目或者相关表的用户名。为防止不同ESQL/C预处理程序使用者和程序执行者引起错误，当您在设置–cs 或 –n选项时，我们建议您在SQL语句中为每一张表输入一个用户名。

字段	含义
MODULENAME	ESQL模块名
CMDNAME	ESQL预处理程序产生的命令名
CMDOWNER	ESQL预处理程序产生的命令的所有者
STATEMENT	原SQL语句
DATA	执行计划数据
DESCPARM	描述参数
STATUS	有效或无效

表 7-3 SYSCMDINFO系统表

删除一个项目

由于项目是为了维护ESQL各模块之间的关系，当项目不再有用时，您可以使用DROP PROJECT语句来删除所有相关的执行计划和项目信息。

您也可以从数据库中删除项目中的模块和所有存储命令。当一个项目只包含一个模块时，删除模块的同时，该项目也会从数据库中删除。

只有项目的所有者或数据库管理员可以删除一个项目或者模块，并且可以赋予或收回其他用户的执行权。

➲ **示例**

删除项目的语法：

```
DROP PROJECT project_name;
```

加载或卸载项目或模块

您可以在dmSQL中使用LOAD、UNLOAD PROJECT或MODULE函数来加载或卸载相关的项目或任意指定模块。更多详细信息请参考*dmSQL用户手册*的*UNLOAD/LOAD语法*。

➲ **示例1**

UNLOAD语法：

```
UNLOAD PROJECT FROM [ower_pattern.]project_pattern TO script_name
UNLOAD MODULE [ower_pattern.]module_pattern FROM PROJECT [owner_name.]project_name
TO script_name.
```

➲ **示例2**

使用UNLOAD命令：

```
UNLOAD PROJECT FROM project1 TO project.scr;
UNLOAD MODULE module1 FROM PROJECT project1 TO module.scr;
```

➲ **示例3**

LOAD语法：

```
LOAD PROJECT FROM script_name.
LOAD MODULE FROM script_name.
```

➲ **示例4**

使用LOAD命令：

```
LOAD PROJECT FROM project.scr;
LOAD MODULE FROM module.scr;
```

注意 *UNLOAD/LOAD函数只能在dmSQL中使用。*

赋予或收回项目权限

项目其他用户的执行权限可以被赋予或者收回。当任何一个没有权限的用户执行项目时，将会在运行时返回错误。因为数据库中的项目是为了研发人员组织或管理模块的，一个应用系统可以连接不同的项目中的不同模块。在本例中，应用程序用户只能执行被赋予权限的部分。

➲ **语法**

GRANT和REVOKE项目权限语法：

```
GRANT EXECUTE ON PROJECT project_name TO auth_user_list;
REVOKE EXECUTE ON PROJECT project_name FROM auth_user_list;
```

ESQL的授权信息存储在SYSAUTHEXE系统表中，您可以在该表中查看用户授权信息。

字段	含义
OBJNAME	项目名
OWNER	项目所有者
OBJTYPE	项目、存储命令或存储过程
GRANTEE	授权的用户

表 7-4 SYSAUTHEXE系统表