一、 ESQL/C资料.....3
1.1 第一节 什么是嵌入SQL语言？ 3
1.1.1 嵌入SQL程序的组成元素 3
1.1.2 什么是静态SQL和动态SQL? 5
1.1.3 什么是SQLCA? 5
1.1.4 什么是SQLDA? 5
1.2 第二节 SYBASE SQL Server嵌入式SQL语言..... 5
1.2.1 一个嵌入SQL语言的简单例子 5
1.2.2 嵌入SQL的处理过程 7
1.2.3 嵌入SQL语句总览 7
1.2.4 动态SQL语句 20
1.2.5 两个例子程序 30
1.3 第三节 IBM DB2 嵌入SQL语言 42
1.3.1 一个简单示例 42
1.3.2 嵌入SQL语句 45
1.3.3 DB2 的嵌入SQL程序处理过程 57
1.3.4 DB2 的动态SQL嵌入语句 62
1.4 第四节 ORACLE数据库的嵌入SQL语言..... 82
1.4.1 基本的SQL语句 82
1.4.2 嵌入PL/SQL 87
1.4.3 动态SQL语句 88
1.5 第五节INFORMIX的嵌入SQL/C语言..... 107
1.5.1 一个简单的入门例子 107
1.5.2 宿主变量 109
1.5.3 嵌入SQL的处理过程 114
1.5.4 动态SQL语言 115
1.6 第六节Microsoft SQL Server7 嵌入式SQL语言 .. 127
1.6.1 一个嵌入SQL语言的简单例子 127
1.6.2 嵌入SQL的处理过程 128
1.6.3 嵌入SQL语句 132
1.6.4 动态SQL语句 140
1.6.5 API.....151
二、 ESQL编程使用说明.....152
2.1 第一章 ESQL介绍..... 152
2.1.1 ESQL中的基本概念.....153
2.1.2 ESQL程序的组成和运行.....153
2.2 第二章 ESQL 程序的基本结构..... 154
2.2.1 程序首部 154

2.2.2 程序体 155
2.3 第三章 查 询 159
2.3.1 SELECT 语句 159
2.3.2 游标的使用 161
2.3.3 定位修改和删除语句 163
2.4 第四章 提交/回滚事务 166
2.4.1 逻辑工作单元 166
2.4.2 COMMIT 语句 167
2.4.3 ROLLBACK语 句 167
2.4.4 DISCONNECT 语句 167
2.5 第五章 错误检测和恢复 167
2.5.1 USERCA的结构 168
2.6 第六章 使用说明书 169
2.6.1 启动Cobase： 169
2.6.2 退出Cobase: 169
2.6.3 交互式SQL（Interactive SQL）访问 170
2.6.4 嵌入式SQL（Enbeded SQL）编程方式 170
2.6.5 补充说明 171

一、 ESQL/C资料

1.1第一节 什么是嵌入SQL语言？

SQL 是一种双重式语言，它既是一种用于查询和更新的交互式数据库语言，又是一种应用程序进行数据库访问时所采取的编程式数据库语言。SQL 语言在这两种方式中的大部分语法是相同的。在编写访问数据库的程序时，必须从普通的编程语言开始（如 C 语言），再把 SQL 加入到程序中。所以，嵌入式 SQL 语言就是将 SQL 语句直接嵌入到程序的源代码中，与其他程序设计语言语句混合。专用的 SQL 预编译程序将嵌入的 SQL 语句转换为能被程序设计语言（如 C 语言）的编译器识别的函数调用。然后，C 编译器编译源代码为可执行程序。

各个数据库厂商都采用嵌入 SQL 语言，并且都符合 ANSI/ISO 的标准。所以，如果采用合适的嵌入 SQL 语言，那么可以使得你的程序能够在各个数据库平台上执行（即：源程序不用做修改，只需要用相应数据库产品的预编译器编译即可）。当然，每个数据库厂商又扩展了 ANSI/ISO 的标准，提供了一些附加的功能。这样，也使得每个数据库产品在嵌入 SQL 方面有一些区别。本章的目标是，对所有的数据库产品的嵌入 SQL 做一个简单、实用的介绍。

当然，嵌入 SQL 语句完成的功能也可以通过应用程序接口（API）实现。通过 API 的调用，可以将 SQL 语句传递到 DBMS，并用 API 调用返回查询结果。这个方法不需要专用的预编译程序。

1.1.1 嵌入SQL程序的组成元素

我们以 IBM 的 DB2 嵌入 SQL 为例，来看看嵌入 SQL 语句的组成元素。

例 1、连接到 SAMPLE 数据库，查询 LASTNAME 为 JOHNSON 的 FIRSTNAME 信息。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"
#include <sqlca.h>
EXEC SQL INCLUDE SQLCA; (1)
main()
{
   EXEC SQL BEGIN DECLARE SECTION; (2)
   char firstname[13];
   char userid[9];
   char passwd[19];
   EXEC SQL END DECLARE SECTION;
   EXEC SQL CONNECT TO sample; (3)
   EXEC SQL SELECT FIRSTNME INTO :firstname (4)
      FROM employee
      WHERE LASTNAME = 'JOHNSON';(4)
   printf( "First name = %s\n", firstname );
   EXEC SQL CONNECT RESET; (5)
   return 0;
}
```

上面是一个简单的静态嵌入 SQL 语句的应用程序。它包括了嵌入 SQL 的主要部分：

（1）中的 include SQLCA 语句定义并描述了 SQLCA 的结构。SQLCA 用于应用程序和数据库之间的通讯，其中的 SQLCODE 返回 SQL 语句执行后的结果状态。

（2）在 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间定义了宿主变量。宿主变量可被 SQL 语句引用，也可以被 C 语言语句引用。它用于将程序中的数据通过 SQL 语句传给数据库管理器，或从数据库管理器接收查询的结果。在 SQL 语句中，主变量前均有“：”标志以示区别。

（3）在每次访问数据库之前必须做 CONNECT 操作，以连接到某一个数据库上。这时，应该保证数据库实例已经启动。

（4）是一条选择语句。它将表 employee 中的 LASTNAME 为“JOHNSON”的行数据的 FIRSTNAME 查出，并将它放在 firstname 变量中。该语句返回一个结果。可以通过游标返回多个结果。当然，也可以包含 update、insert 和 delete 语句。

（5）最后断开数据库的连接。

从上例看出，每条嵌入式 SQL 语句都用 EXEC SQL 开始，表明它是一条 SQL 语句。这也是告诉预编译器在 EXEC SQL 和“；”之间是嵌入 SQL 语句。如果一条嵌入式 SQL 语句占用多行，在 C 程序中可以用续行符“\”。

1.1.2 什么是静态SQL和动态SQL?

嵌入 SQL 语言，分为静态 SQL 语言和动态语言两类。静态 SQL 语言，就是在编译时已经确定了引用的表和列。宿主变量不改变表和列信息。可以使用主变量改变查询参数值，但是不能用主变量代替表名或列名。

动态 SQL 语言就是：不在编译时确定 SQL 的表和列，而是让程序在运行时提供，并将 SQL 语句文本传给 DBMS 执行。静态 SQL 语句在编译时已经生成执行计划。而动态 SQL 语句，只有在执行时才产生执行计划。动态 SQL 语句首先执行 PREPARE 语句要求 DBMS 分析、确认和优化语句，并为其生成执行计划。DBMS 还设置 SQLCODE 以表明语句中发现的错误。当程序执行完“PREPARE”语句后，就可以用 EXECUTE 语句执行执行计划，并设置 SQLCODE，以表明完成状态。

1.1.3 什么是SQLCA?

应用程序执行时，每执行一条 SQL 语句，就返回一个状态符和一些附加信息。这些信息反映了 SQL 语句的执行情况，它有助于用户分析应用程序的错误所在。这些信息都存放在 sqlca.h 的 sqlca 结构中。如果一个源文件中包含 SQL 语句，则必须要在源程序中定义一个 SQLCA 结构，而且名为 SQLCA。最简单的定义方法是在源文件中加入一些语句：EXEC SQL INCLUDE sqlca.h。每个数据库产品都提供了 SQLCA 结构。

1.1.4 什么是SQLDA?

我们知道，动态 SQL 语句在编译时可能不知道有多少列信息。在嵌入 SQL 语句中，这些不确定的数据是通过 SQLDA 完成的。SQLDA 的结构非常灵活，在该结构的固定部分，指明了多少列等信息，在该结构的后面有一个可变长的结构，说明每列的信息。在从数据库获得数据时，就可以采用 SQLDA 来获得每行的数据。各个数据库产品的 SQLDA 结构都不完全相同。

1.2 第二节 SYBASE SQL Server嵌入式SQL语言

1.2.1 一个嵌入SQL语言的简单例子

我们首先来看一个简单的嵌入式 SQL 语言的程序（C 语言）：用 sa（口令为 password）连接数据库服务器，并将所有书的价格增加 10%。这个例子程序如下：

例 1、

```
/*建立通讯区域*/
Exec sql include sqlca;
main()
{
   /*声明宿主变量*/
   EXEC SQL BEGIN DECLARE SECTION;
   char user[30],passwd[30];
   EXEC SQL END DECLARE SECTION;
   /*错误处理*/
   EXEC SQL WHENEVER SQLERROR CALL err_p();
   /*连接到 SQL SERVER 服务器*/
   printf("\nplease enter your userid ");
   gets(user);
   printf("\npassword ");
   gets(passwd);
   exec sql connect :user identified by :passwd;
   exec sql use pubs2;
   EXEC SQL update titles set price=price*1.10;
   EXEC SQL commit work;
   /*断开数据库服务器的连接*/
   Exec sql disconnect all;
   return (0);
}
/*错误处理程序*/
err_p()
{
   printf("\nError occurred: code %d.\n%s", \
   sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

从上面这个例子，我们看出嵌入 SQL 的基本特点是：

1、每条嵌入式 SQL 语句都用 EXEC SQL 开始，表明它是一条 SQL 语句。这也是告诉预编译器在 EXEC SQL 和“；”之间是嵌入 SQL 语句。

2、如果一条嵌入式 SQL 语句占用多行，在 C 程序中可以用续行符“\”，在 Fortran 中必须有续行符。其他语言也有相应规定。

3、每一条嵌入 SQL 语句都有结束符号，如：在 C 中是“；”。

4、嵌入 SQL 语句的关键字不区分大小写。

5、可以使用“/*….*/”来添加注释。也可以使用“--”来添加注释。

1.2.2 嵌入SQL的处理过程

嵌入 SQL 的处理过程如下图所示：

图 6-1 SYBASE SQL SERVER 嵌入 SQL 程序处理过程

嵌入 SQL 程序的后缀为.cp。嵌入 SQL 处理的第一步是预编译。预编译器（cpre.exe）处理过程分为两个小步：

第一小步：语法分析。检查嵌入 SQL 语句的语法正确性。

第二小步：增加一些预编译器变量，并注释了所有的嵌入的 SQL 语句，将嵌入 SQL 语句转换为对 client-library 中函数的调用（注意：在连接时，编译后的代码需要使用 client-library 中的库文件）。如果在编译时，设置一些选项，则生成存储过程。预编译后可能产生 3 个文件：一个 C 文件（肯定产生），一个列表文件（需要在编译时设置选项，才能产生）和一个 isql 脚本文件（需要在编译时设置选项，才能产生）。列表文件包含了输入文件的源语句和一些警告信息和错误信息。Isql 脚本文件包含了预编译器产生的存储过程脚本文件。这个存储过程是用 T-SQL 写的。总之，预编译器的处理方法是，注释了嵌入的 SQL 语句，用一些特定的函数代替。

第二步是 C 源程序的编译和链接。cl 是编译和链接命令的集成命令，编译的结果是产生.obj，在链接时，将 C 的系统库和 SQL Server 提供的库文件同目标文件连接在一起。最后生成.exe。也可以使用 SET LIB 语句设置库文件的环境信息。

1.2.3 嵌入SQL语句总览

除了 print、readtext 和 writetext 外，大多数的 Transact-SQL 语句都可以在嵌入 SQL 中使用。嵌入 SQL 语句的语法为：“exec sql [at connection_name] sql_statement; ”。那么，你可以用 Transact-SQL 语句来替代 sql_statement 就可以完成嵌入 SQL 的编写。(同 T-SQL 相比，嵌入 SQL 提供了：自动数据类型转换、动态 SQL、SQLCA 数据结构等功能。)

但是，也有一些嵌入式 SQL 所特有的语句，有些嵌入式 SQL 语句的名字同 Transact-SQL 语句相同，但是语句的语法有所不同。

嵌入 SQL 语句应该包含五个步骤：

1）、通过 SQLCA 建立应用程序和 SQL SERVER 的 SQL 通信区域。

2）、声明宿主变量。

3）、连接到 SQL SERVER。

4）、通过 SQL 语句操作数据。

5）、处理错误和结果信息。

嵌入式 SQL 语句分为静态 SQL 语句和动态 SQL 语句两类。下面我们按照功能讲解这些语句。本节讲解静态 SQL 语句的作用。动态 SQL 语句将在下一节讲解。同动态 SQL 相关的一些语句也在下一节中讲解。

1.2.3.1 宿主变量

1）、声明方法

宿主变量（host variable）就是在嵌入式 SQL 语句中引用主语言说明的程序变量（如例中的 user[31]变量）。如：

```
EXEC SQL BEGIN DECLARE SECTION;
char user[31],passwd[31];
EXEC SQL END DECLARE SECTION;
……
exec sql connect :user identified by :passwd;
…….
```

在嵌入式 SQL 语句中使用主变量前，必须采用 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间给主变量说明。这两条语句不是可执行语句，而是预编译程序的说明。主变量是标准的 C 程序变量。嵌入 SQL 语句使用主变量来输入数据和输出数据。C 程序和嵌入 SQL 语句都可以访问主变量。

另外，在定义宿主变量时也可以使用 client-library 定义的数据类型，如：CS_CHAR。这些定义存放在 cspublic.h 文件中。如：

```
EXEC SQL BEGIN DECLARE SECTION;
CS_CHAR user[30],passwd[30];
EXEC SQL END DECLARE SECTION;
```

client-library 定义的数据类型共有：CS_BINARY、CS_BIT、 CS_BOOL、 CS_CHAR、CS_DATETIME、CS_DATETIME4、 CS_DECIMAL、 CS_FLOAT、 CS_REAL、CS_IMAGE、 CS_INT、CS_MONEY、 CS_MONEY4、 CS_NUMERIC、CS_RETCODE、 CS_SMALLINT、 CS_TEXT、 CS_TINYINT、CS_VARBINARY、 CS_VARCHAR、 CS_VOID。

为了便于识别主变量，当嵌入式 SQL 语句中出现主变量时，必须在变量名称前标上冒号（：）。冒号的作用是，告诉预编译器，这是个主变量而不是表名或列名。不能在声明时，初始化数组变量。

由上可知，SYBASE SQL SERVER 使用宿主变量传递数据库中的数据和状态信息到应用程序，应用程序也通过宿主变量传递数据到 SYBASE 数据库。根据上面两种功能，宿主变量分为输出宿主变量和输入宿主变量。在 SELECT INTO 和 FETCH 语句之后的宿主变量称作“输出宿主变量”，这是因为从数据库传递列数据到应用程序。如：

```
exec sql begin declare section;
CS_CHAR id[5];
exec sql end declare section;
exec sql select title_id into :id from titles
where pub_id = "0736" and type = "business";
```

除了 SELECT INTO 和 FETCH 语句外的其他 SQL 语句（如：INSERT、UPDATE 等语句）中的宿主变量，称为“输入宿主变量”。这是因为从应用程序向数据库输入值。如：

```
exec sql begin declare section;
CS_CHAR id[7];
CS_CHAR publisher[5];
```

```
    exec sql end declare section;
    ...
    exec sql delete from titles where title_id = :id;
    exec sql update titles set pub_id = :publisher
    where title_id = :id;
```

另外，也可以通过宿主变量获得存储过程的执行状态信息。如：

```
    exec sql begin declare section;
    CS_SMALLINT retcode;
    exec sql end declare section;
    exec sql begin transaction;
    exec sql exec :retcode = update_proc;
    if (retcode != 0)
    {
    exec sql rollback transaction;
```

也可以通过宿主变量获得存储过程的返回值。如：

```
exec sql exec a_proc :par1 out, :par2 out;
```

2）、主变量的数据类型

SYBASE SQL SERVER 支持的数据类型与程序设计语言支持的数据类型之间有很大差别。这些差别对主变量影响很大。一方面，主变量是一个用程序设计语言的数据类型说明并用程序设计语言处理的程序变量；另一方面，在嵌入 SQL 语句中用主变量保存数据库数据。所以，在嵌入 SQL 语句中，必须映射 C 数据类型为合适的 SQL Server 数据类型。必须慎重选择主变量的数据类型。在 SQL SERVER 中，预编译器能够自动转换兼容的数据类型。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int hostvar1 = 39;
char *hostvar2 = "telescope";
float hostvar3 = 355.95;
EXEC SQL END DECLARE SECTION;

EXEC SQL UPDATE inventory
   SET department = :hostvar1
   WHERE part_num = "4572-3";

EXEC SQL UPDATE inventory
   SET prod_descrip = :hostvar2
   WHERE part_num = "4572-3";

EXEC SQL UPDATE inventory
   SET price = :hostvar3
   WHERE part_num = "4572-3";
```

在第一个 update 语句中，department 列为 smallint 数据类型（integer ），所以应该把 hostvar1 定义为 int 数据类型（integer）。这样的话，从 C 到 SQL Server 的 hostvar1 可以直接映射。在第二个 update 语句中，prod_descip 列为 varchar 数据类型，所以应该把 hostvar2 定义为字符数组。这样的话，从 C 到 SQL Server 的 hostvar2 可以从字符数组映射为 varchar 数据类型。在第三个 update 语句中，price 列为 money 数据类型。在 C 语言中，没有相应的数据类型，所以用户可以把 hostvar3 定义为 C 的浮点变量或字符数据类型。SQL Server 可以自动将浮点变量转换为 money 数据类型（输入数据），或将 money 数据类型转换为浮点变量（输出数据）。

注意的是，如果数据类型为字符数组，那么 SQL Server 会在数据后面填充空格，直到填满该变量的声明长度（CS_CHAR 数据类型除外）。

下表列出了 C 的数据类型和 SQL SERVER 数据类型的一些兼容关系：

可兼容的 C 数据类型	分配的 SQL Server 数据类型	SYBASE 提供的数据类型描述
Short	Smallint	CS_SMALLINT2 字节整数
Int	Smallint	CS_SMALLINT2 字节整数
Long	Int	CS_INT4 字节整数
Float	Real	CS_REAL4 字节浮点数
Double	Float	CS_FLOAT8 字节浮点数
Char	Carchar[X]	VARCHARCS_CHAR 字符数据类型
Unsigned char	BinaryVarbinary	CS_BINARYBinary 数据类型
Unsigned char	tinyint	CS_TINYINT1 字节整数
无	Datetime	CS_DATETIME8 字节 datetime 类型
无	Smalldatetime	CS_DATETIME44 字节 datetime 类型
无	Decimal	CS_DECIMALDecimal 数据类型
无	numeric	CS_NUMERICNumeric 数据类型
无	Money	CS_MONEY8 字节 money 类型
无	smallmoney	CS_MONEY44 字节 money 类型
Unsigned char	Text	CS_TEXT 文本数据类型
Unsigned char	image	CS_IMAGE 图象数据类型
无	boolean	CS_BITBit 数据类型

因为 C 没有 date 或 time 数据类型，所以 SQL Server 的 date 或 time 列将被转换为字符。缺省情况下，使用以下转换格式：mm dd yyyy hh:mm:ss[am | pm]。你也可以使用字符数据格式将 C 的字符数据存放到 SQL Server 的 date 列上。你也可以使用 Transact-SQL 中的 convert 语句来转换数据类型。如：SELECT CONVERT(char, date, 8) FROM sales。

下表是从 SQL SERVER 数据类型到 C 的数据类型的转换关系：

SQL_SERVER 数据类型 C 数据类型

CS_TINYINT CS_SMALLINT CS_INT CS_REAL CS_CHAR CS_MONEY CS_DATETIME

char 可以 可以 可以 可以 可以 可以 可以

varchar 可以 可以 可以 可以 可以 可以 可以

bit 可以 可以 可以 可以 可以 可以

binary 可以 可以 可以 可以 可以 可以

tinyint 可以 可以 可以可以 可以 可以
smallint 可以 可以 可以 可以可以 可以
int 可以 可以 可以可以 可以 可以
float 可以 可以 可以 可以 可以可以
money 可以 可以 可以 可以 可以 可以
datetime 可以 可以
decimal 可以 可以 可以 可以 可以 可以
numeric 可以 可以 可以 可以 可以 可以

下表是从 C 的数据类型到 SQL SERVER 数据类型的转换关系：

C 数据类型	SQL_SERVER 数据类型									
	tinyint	bit	smallint	int	float	char	money	datetime	decimal	numeric
Unsigned char	可以	可以	可以	可以	可以	需要自己转换	可以	可以	可以	
Short int		可以	可以	可以	可以	可以	需要自己转换	可以	可以	可以
Long int		可以	可以	可以	可以	可以	需要自己转换	可以	可以	可以
float double		可以	可以	可以	可以	可以	需要自己转换	可以	可以	可以
char	需要自己转换	需要自己转换	需要自己转换	需要自己转换	需要自己转换	可以	需要自己转换	可以	需要自己转换	需要自己转换
money	可以	可以	可以	可以	可以	可以	可以	可以	可以	
datetime	需要自己转换	可以								

3）、主变量和 NULL

大多数程序设计语言（如 C）都不支持 NULL。所以对 NULL 的处理，一定要在 SQL 中完成。我们可以使用主机指示符变量（host indicator variable）来解决这个问题。在嵌入式 SQL 语句中，主变量和指示符变量共同规定一个单独的 SQL 类型值。如：

```
EXEC SQL SELECT price INTO :price :price_nullflag FROM titles
    WHERE au_id = "mc3026"
```

其中，price 是主变量，price_nullflag 是指示符变量。

使用指示符变量的语法为：: host_variable [[indicator] : indicator_variable]。其中，indicator 可以不写。针对宿主变量是输出宿主变量，还是输入宿主变量。指示符变量共分两种情况。

情况 1：同输出宿主变量一起使用，则 indicator_varibale 为：

l-1。表示相应列值为 NULL。表示主变量应该假设为 NULL。（注意：宿主变量的实际值是一个无关值，不予考虑）。

l0。表示非 NULL 值。该变量存放了非 NULL 的列值。

l>0。表示宿主变量包含了列值的截断值。该指示变量存放了该列值的实际长度。

下面是一个同输出宿主变量一起使用的指示变量的例子：

```
exec sql begin declare section;
CS_CHAR id[6];
CS_SMALLINT indic;
CS_CHAR pub_name[41];
exec sql end declare section;

exec sql select pub_id into :id indicator :indic
   from titles
   where title like "%Stress%";

if (indic == -1)
{
   printf("\npub_id is null");
}
else
{
   exec sql select pub_name into :pub_name
   from publishers where pub_id = :id;
   printf("\nPublisher: %s", pub_name);
}
```

情况 2：同输入宿主变量一起使用，则 indicator_varibale 为：

l-1。表示主变量应该假设为 NULL。（注意：宿主变量的实际值是一个无关值，不予考虑）。应该将 NULL 赋值给相应列。

l0。表示非 NULL 值。该变量存放了非 NULL 值。应该将宿主变量的值赋值给相应列。

对于以下语句：

```
EXEC SQL SELECT price INTO :price :price_nullflag FROM titles
    WHERE au_id = "mc3026"
```

如果不存在 mc3026 写的书，那么 price_nullflag 为-1，表示 price 为 NULL；如果存在，则 price 为实际的价格。下面我们再看一个 update 的例子：

```
EXEC SQL UPDATE closeoutsale
    SET temp_price = :saleprice :saleprice_null, listprice = :oldprice;
```

如果 saleprice_null 是-1，则上述语句等价为：

```
EXEC SQL UPDATE closeoutsale
    SET temp_price = null, listprice = :oldprice;
```

我们也可以在指示符变量前面加上“INDICATOR”关键字，表示后面的变量为指示符变量。如：

```
EXEC SQL UPDATE closeoutsale
    SET temp_price = :saleprice INDICATOR :saleprice_null;
```

指示符变量也是宿主变量，定义指示符变量同定义宿主变量一样。它应该是一个 2 个字节的整数（short 或 CS_SMALLINT）。

1.2.3.2 连接数据库

在程序中，使用 CONNECT 语句来连接数据库。该语句的完整语法为：

```
exec sql connect : user [identified by : password]
   [at : connection_name] [using : server]
   [labelname labelname labelvalue labelvalue...]
```

其中，

lserver 为服务器名。如省略，则为本地服务器名。

lconnection_name 为连接名。可省略。如果你仅仅使用一个连接，那么无需指定连接名。可以使用 SET CONNECTION 来使用不同的连接。

luser 为登录名。

lpassword 为密码。

如：使用 my_id 用户和 passes 密码连接到 SYBASE 服务器。

```
exec sql begin declare section;
CS_CHAR user[16];
CS_CHAR passwd[16];
CS_CHAR server[BUFSIZ];
exec sql end declare section;
strcpy(server,"SYBASE");
strcpy(passwd,"passes");
strcpy(user, "my_id");
exec sql connect :user identified by :passwd using :server;
```

请看下面这些例子来理解连接名的使用方法。

```
...
exec sql begin declare section;
CS_CHAR user[16];
CS_CHAR passwd[16];
CS_CHAR name;
CS_INT value, test;
CS_CHAR server_1[BUFSIZ];
CS_CHAR server_2[BUFSIZ];
exec sql end declare section;
...
strcpy (server_1, "sybase1");
strcpy (server_2, "sybase2");
strcpy(user, "my_id");
strcpy(passwd, "mypass");
exec sql connect :user identified by :passwd
   at connection_2 using :server_2;
exec sql connect :user identified by :passwd
   using :server_1;
/* 下面这个语句使用了"server_1"的连接*/
exec sql select royalty into :value from authors
where author = :name;
if (value == test)
{
   /* 下面这个语句使用了"connection_2"连接 */
   exec sql at connection_2 update authors
      set column = :value*2
      where author = :name;
}
```

在嵌入 SQL 语句中，使用 DISCONNECT 语句断开数据库的连接。其语法为：

```
DISCONNECT [connection_name | ALL | CURRENT]
```

其中，connection_name 为连接名。ALL 表示断开所有的连接。CURRENT 表示断开当前连接。断开连接会回滚当前事务、删除临时表、关闭游标和释放锁等。

1.2.3.3 数据的查询和修改

可以使用 SELECT INTO 语句查询数据，并将数据存放在主变量中。如：查询 lastname 为 stringer 的 firstname 信息。

```
EXEC SQL SELECT au_fname INTO :first_name
   from authors where au_lname = "stringer";
```

使用 DELETE 语句删除数据。其语法类似于 Transact-SQL 中的 DELETE 语法。如：

```
EXEC SQL DELETE FROM authors WHERE au_lname = 'White'
```

使用 UPDATE 语句可以更新数据。其语法类似 Transact-SQL 中的 UPDATE 语法。如：

```
EXEC SQL UPDATE authors SET au_fname = 'Fred' WHERE au_lname = 'White'
```

使用 INSERT 语句可以插入新数据。其语法就是 Transact-SQL 中的 INSERT 语法。如：

```
EXEC SQL INSERT INTO homesales (seller_name, sale_price)
    real_estate('Jane Doe', 180000.00);
```

多行数据的查询和修改请参见下一节——游标。

1.2.3.4 游标的使用

用嵌入式 SQL 语句查询数据分成两类情况。一类是单行结果，一类是多行结果。对于单行结果，可以使用 SELECT INTO 语句；对于多行结果，你必须使用 cursor（游标）来完成。游标(Cursor)是一个与 SELECT 语句相关联的符号名，它使用户可逐行访问由 SQL Server 返回的结果集。先请看下面这个例子，这个例子的作用是逐行打印 staff 表的 id、name、dept、 job、years、salary 和 comm 的值。

```
.....
EXEC SQL DECLARE C1 CURSOR FOR
SELECT id, name, dept, job, years, salary, comm FROM staff;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
   /* SQLCODE will be zero if data is successfully fetched */
   EXEC SQL FETCH c1 INTO :id, :name, :dept, :job, :years, :salary, :comm;
   if (SQLCODE == 0)
      printf("%4d %12s %10d %10s %2d %8d %8d",
             id, name, dept, job, years, salary, comm);
}
EXEC SQL CLOSE c1;
......
```

从上例看出，你首先应该定义游标结果集，即定义该游标的 SELECT 语句返回的行的集合。然后，使用 FETCH 语句逐行处理。

值得注意的是，嵌入 SQL 语句中的游标定义选项同 Transact-SQL 中的游标定义选项有些不同。必须遵循嵌入 SQL 语句中的游标定义选项。

1）、声明游标：

如：

```
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT id, name, dept, job, years, salary, comm FROM staff;
```

其中，C1 是游标的名称。

2）、打开游标

如：

```
EXEC SQL OPEN c1;
```

完整语法为：

EXEC SQL OPEN 游标名 [USING 主变量名 | DESCRIPTOR 描述名]。

关于动态 OPEN 游标的描述见第四节。

3）、取一行值

如：

EXEC SQL FETCH c1 INTO :id, :name, :dept, :job, :years, :salary, :comm;

关于动态 FETCH 语句见第四小节。

4）、关闭游标

如：

EXEC SQL CLOSE c1;

关闭游标的同时，会释放由游标添加的锁和放弃未处理的数据。在关闭游标前，该游标必须已经声明和打开。另外，程序终止时，系统会自动关闭所有打开的游标。

也可以使用 UPDATE 语句和 DELETE 语句来更新或删除由游标选择的当前行。使用 DELETE 语句删除当前游标所在的行数据的具体语法如下：

DELETE [FROM] {table_name | view_name} WHERE CURRENT OF cursor_name

其中，

ltable_name 是表名，该表必须是 DECLARE CURSOR 中 SELECT 语句中的表。

lview_name 是视图名，该视图必须是 DECLARE CURSOR 中 SELECT 语句中的视图。

lcursor_name 是游标名。

请看下面这个例子，逐行显示 firstname 和 lastname，询问用户是否删除该信息，如果回答“是”，那么删除当前行的数据。

```
EXEC SQL DECLARE c1 CURSOR FOR
SELECT au_fname, au_lname FROM authors ;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
   EXEC SQL FETCH c1 INTO :fname, :lname;
   if (SQLCODE == 0)
   {
      printf("%12s %12s\n", fname, lname);
      printf("Delete? ");
      scanf("%c", &reply);
      if (reply == 'y')
      {
         EXEC SQL DELETE FROM authors WHERE CURRENT OF c1;
         printf("delete sqlcode= %d\n", SQLCODE(ca));
      }
   }
}
EXEC SQL CLOSE c1;
```

1.2.3.5 SQLCA

DBMS 是通过 SQLCA（SQL 通信区）向应用程序报告运行错误信息。SQLCA 是一个含有错误变量和状态指示符的数据结构。通过检查 SQLCA，应用程序能够检查出嵌入式 SQL 语句是否成功，并根据成功与否决定是否继续往下执行。预编译器自动会在嵌入 SQL 语句中插入 SQLCA 数据结构。在程序中可以使用 EXEC SQL INCLUDE SQLCA，目的是告诉 SQL 预编译程序在该程序中包含一个 SQL 通信区。也可以不写，系统会自动加上 SQLCA 结构。

下表是 SQLCA 结构中的变量和作用：

变量	数据类型	作用
sqlcaid	char	包含“sqlca”的字符串
sqlcabc	long	SQLCA 的长度
sqlcode	long	包含最近一次语句执行的返回代码
sqlwarn[0] 到 sqlwarn[7]	char	警告标志。如果是“W”，那么表示有警报信息。
sqlerrm.sqlerrmc[ ]	char	错误信息。
sqlerrm.sqlerrml	long	错误信息的长度。
sqlerrp	char	检测错误或警告信息的过程。
sqlerrd[6]	long	警告或错误的详细信息。[2]中存放影响行的个数。

下面仔细讲解几个重要的变量。

1）、SQLCODE

SQLCA 结构中最重要的部分是 SQLCODE 变量。在执行每条嵌入式 SQL 语句时，DBMS 在 SQLCA 中设置变量 SQLCODE 值，以指明语句的完成状态：

1、0 该语句成功执行，无任何错误或报警。

2、<0 出现了严重错误。

3、>0 出现了报警信息。

4、100 没有数据存在。在 FETCH 语句中，表示到达结果集的末尾。在 UPDATE、DELETE、INSERT 语句中，表示没有满足条件的数据。

例：显示错误信息。

```
printf("\nError occurred: code %d.\n%s",
    sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
```

在 SYBASE SQL SERVER 中，也可以单独定义 SQLCODE。如：

```
long SQLCODE;
exec sql open cursor pub_id;
while (SQLCODE == 0)
{
    exec sql fetch pub_id into :pub_name;
    ......
```

}

2）、SQLSTATE

SQLSTATE 变量也是 SQLCA 结构中的成员。它同 SQLCODE 一样，都是返回错误信息。SQLSTATE 是在 SQLCODE 之后产生的。这是因为，在制定 SQL2 标准之前，各个数据库厂商都采用 SQLCODE 变量来报告嵌入式 SQL 语句中的错误状态。但是，各个厂商没有采用标准的错误描述信息和错误值来报告相同的错误状态。所以，标准化组织增加了 SQLSTATE 变量，规定了通过 SQLSTATE 变量报告错误状态和各个错误代码。因此，目前使用 SQLCODE 的程序仍然有效，但也可用标准的 SQLSTATE 错误代码编写新程序。值得注意的是，Open client emebeded SQL/C11.1.x 并不完全支持 SQLSTATE。

SQLSTATE 是一个字符串参数。具体含义如下：

值	作用
00XXX	成功
01XXX	警告
02XXX	不存在数据
其他值	错误

1.2.3.6 WHENEVER

在每条嵌入式 SQL 语句之后立即编写一条检查 SQLCODE/SQLSTATE 值的程序，是一件很繁琐的事情。为了简化错误处理，可以使用 WHENEVER 语句。该语句是 SQL 预编译程序的指示语句，而不是可执行语句。它通知预编译程序在每条可执行嵌入式 SQL 语句之后自动生成错误处理程序，并指定了错误处理操作。

用户可以使用 WHENEVER 语句通知预编译程序去如何处理三种异常处理：

lWHENEVER SQLERROR action：表示一旦 sql 语句执行时遇到错误信息，则执行 action，action 中包含了处理错误的代码（SQLCODE<0）。

lWHENEVER SQLWARNING action:表示一旦 sql 语句执行时遇到警告信息,则执行 aciton，即 action 中包含了处理警报的代码（SQLCODE=1）。

lWHENEVER NOT FOUND action：表示一旦 sql 语句执行时没有找到相应的元组，则执行 action，即 action 包含了处理没有查到内容的代码（SQLCODE=100）。

针对上述三种异常处理，用户可以指定预编译程序采取以下三种行为（action）：

lWHENEVER …GOTO：通知预编译程序产生一条转移语句。

lWHENEVER…CONTINUE：通知预编译程序让程序的控制流转入到下一个主语言语句。

lWHENEVER…CALL：通知预编译程序调用函数。

其完整语法如下：

WHENEVER {SQLWARNING | SQLERROR | NOT FOUND} {CONTINUE | GOTO stmt_label | CALL function()}

例：WHENEVER 的作用

```
EXEC SQL WHENEVER sqlerror GOTO errormessage1;
EXEC SQL DELETE FROM homesales
```

```
        WHERE equity < 10000;
    EXEC SQL DELETE FROM customerlist
        WHERE salary < 40000;
    EXEC SQL WHENEVER sqlerror CONTINUE;
    EXEC SQL UPDATE homesales
        SET equity = equity - loanvalue;
    EXEC SQL WHENEVER sqlerror GOTO errormessage2;
    EXEC SQL INSERT INTO homesales (seller_name, sale_price)
        real_estate('Jane Doe', 180000.00);
.
.
.
    errormessage1:
        printf("SQL DELETE error: %ld\n, sqlcode);
        exit();

    errormessage2:
        printf("SQL INSERT error: %ld\n, sqlcode);
        exit();
```

WHENEVER 语句是预编译程序的指示语句。在上面这个例子中，由于第一个 WHENEVER 语句的作用，前面两个 DELETE 语句中任一语句内的一个错误会在 errormessage1 中形成一个转移指令。由于一个 WHENEVER 语句替代前面 WHENEVER 语句，所以，嵌入式 UPDATE 语句中的一个错误会直接转入下一个程序语句中。嵌入式 INSERT 语句中的一个错误会在 errormessage2 中产生一条转移指定。

从上面例子看出，WHENEVER/CONTINUE 语句的主要作用是取消先前的 WHENEVER 语句的作用。WHENEVER 语句使得对嵌入式 SQL 错误的处理更加简便。应该在应用程序中普遍使用，而不是直接检查 SQLCODE 的值。

1.2.3.7 批处理

嵌入 SQL 也支持批处理。如：

```
    exec sql insert into TABLE1 values (:val1)
    insert into TABLE2 values (:val2)
    insert into TABLE3 values (:val3);
```

SYBASE SQL SERVER 将在 EXEC SQL 和“；”之间的所有 T-SQL 语句作为一个批来处理。在上例中，会将这 3 个语句作为一组来处理。

1.2.3.8 事务

SYBASE SQL SERVER 预编译器能够处理两种事务模式：ANSI/ISO 事务模式和 T-SQL 模式。在 T-SQL 模式中，除非有 begin transaction 外，每个语句都会做提交。可以在编译时设置事务模式。ANSI/ISO 模式是系统的缺省模式。嵌入 SQL 的事务语法和 T-SQL 的事务语法是相同的。

2.3.8.1 T-SQL 事务模式

1）、开始事务

```
exec sql [at connect_name] begin transaction [ transaction_name];
```

2）、保存事务回滚点

```
exec sql [at connect_name] save transaction [ savepoint_name];
```

3）、提交事务

```
exec sql [at connect_name] commit transaction [ transaction_name];
```

4）、回滚事务

```
exec sql [at connect_name] rollback transaction
      [ savepoint_name | transaction_name];
```

2.3.8.2 ANSI/ISO 事务模式

该模式没有 begin transaction 和 save transaction。在应用程序中，只要遇到以下语句，就表示事务开始：delete、insert、select、update、open 和 exec。当遇到 commit work 或 rollback work，就表示事务结束。也就是说，commit 和 rollback 表示当前事务结束，下一个事务开始。

1.2.4 动态SQL语句

前一节中讲述的嵌入SQL语言都是静态SQL语言,即在编译时已经确定了引用的表和列。主变量不改变表和列信息。在上几节中，我们使用主变量改变查询参数，但是不能用主变量代替表名或列名。否则，系统报错。动态 SQL 语句就是来解决这个问题。

动态 SQL 语句的目的是，不是在编译时确定 SQL 的表和列，而是让程序在运行时提供，并将 SQL 语句文本传给 DBMS 执行。静态 SQL 语句在编译时已经生成执行计划。而动态 SQL 语句，只有在执行时才产生执行计划。动态 SQL 语句首先执行 PREPARE 语句要求 DBMS 分析、确认和优化语句，并为其生成执行计划。DBMS 还设置 SQLCODE 以表明语句中发现的错误。当程序执行完“PREPARE”语句后，就可以用 EXECUTE 语句执行执行计划，并设置 SQLCODE，以表明完成状态。

使用动态 SQL，共分成四种方法：

方法 支持的 SQL 语句 实现方法

1 该语句内不包含宿主变量，该语句不是查询语句 execute immediate

2 该语句内包含输入宿主变量 ，该语句不是查询语句 prepare 和 execute

3 包含已知数目的输入宿主变量或列的查询 prepare 和 fetch

4 包含未知数目的输入宿主变量或列的查询 prepare 和 fetch，用描述符

按照功能和处理上的划分，动态 SQL 应该分成两类来解释：动态修改和动态查询。方法 1 和方法 2 完成动态修改（参见 2.4.1）。方法 3 和方法 4 完成了动态查询（参见 2.4.2 和 2.4.3）。

1.2.4.1 动态修改

方法 1 和方法 2 完成动态修改。对于方法 1，表示要执行一个完整的 T-SQL 语句，该语句没有宿主变量，不是一个查询语句。因为没有宿主变量来带入不同的参数，所以不能通过方法 1 来重复执行修改语句。具体语法为：

exec sql [at connection_name] execute immediate

{: host_variable | string};

其中，host_variable 和 string 是存放完整 T-SQL 语句。

例：提示用户输入被更新书的条件，然后组合成为一个完整的 SQL 语句，并执行更新。

```
exec sql begin declare section;
CS_CHAR sqlstring[200];
exec sql end declare section;

char cond[150];
exec sql whenever sqlerror call err_p();
exec sql whenever sqlwarning call warn_p();
strcpy(sqlstring,"update titles set price=price*1.10 where ");
printf("Enter search condition:");
scanf("%s", cond);
strcat(sqlstring, cond);

exec sql execute immediate :sqlstring;
exec sql commit work;
```

对于方法 2，可以执行一个包含输入宿主变量的动态修改语句。该方法要使用 PREPARE 语句和 EXECUTE 语句。PREPARE 语句是动态 SQL 语句独有的语句。其语法为：

PREPARE 语句名 FROM 宿主变量|字符串

该语句接收含有 SQL 语句串的宿主变量，并把该语句送到 DBMS。DBMS 编译语句并生成执行计划。在语句串中包含一个“？”表明参数，当执行语句时，DBMS 需要参数来替代这些“？”。PREPRARE 执行的结果是，DBMS 用语句名标志准备后的语句。SQL SERVER 编译后的语句以临时存储过程的形式存放在缓冲区中。语句名类似于游标名，是一个 SQL 标识符。在执行 SQL 语句时，EXECUTE 语句后面是这个语句名。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char prep[] = "INSERT INTO mf_table VALUES(?,?,?)";
char name[30];
char car[30];
double num;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE prep_stat FROM :prep;
while (SQLCODE == 0)
{
    strcpy(name, "Elaine");
    strcpy(car, "Lamborghini");
    num = 4.9;
    EXEC SQL EXECUTE prep_stat USING :name, :car, :num;
}
```

在这个例子中，prep_stat 是语句名，prep 宿主变量的值是一个 INSERT 语句，包含了三个参数（3 个“？”）。PREPARE 的作用是，DBMS 编译这个语句并生成执行计划，并把语句名标志这个准备后的语句。值得注意的是，PREPARE 中的语句名的作用范围为整个程序，所以不允许在同一个程序中使用相同的语句名在多个 PREPARE 语句中。

EXECUTE 语句是动态 SQL 独有的语句。它的语法如下：

EXECUTE 语句名 USING 宿主变量 | DESCRIPTOR 描述符名

请看上面这个例子中的“EXEC SQL EXECUTE prep_stat USING :name, :car, :num;”语句，它的作用是，请求 DBMS 执行 PREPARE 语句准备好的语句。当要执行的动态语句中包含一个或多个参数标志时，在 EXECUTE 语句必须为每一个参数提供值，如：:name、:car 和:num。这样的话，EXECUTE 语句用宿主变量值逐一代替准备语句中的参数标志（“？”），从而，为动态执行语句提供了输入值。

使用主变量提供值，USING 子句中的主变量数必须同动态语句中的参数标志数一致，而且每一个主变量的数据类型必须同相应参数所需的数据类型相一致。各主变量也可以有一个伴随主变量的指示符变量。当处理 EXECUTE 语句时，如果指示符变量包含一个负值，就把 NULL 值赋予相应的参数标志。除了使用主变量为参数提供值，也可以通过 SQLDA 提供值（见节 2.4.4）。

1.2.4.2 动态游标

使用动态游标可以完成方法 3。

游标分为静态游标和动态游标两类。对于静态游标，在定义游标时就已经确定了完整的 SELECT 语句。在 SELECT 语句中可以包含主变量来接收输入值。当执行游标的 OPEN 语句时，主变量的值被放入 SELECT 语句。在 OPEN 语句中，不用指定主变量，因为在 DECLARE CURSOR 语句中已经放置了主变量。请看下面静态游标的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char szLastName[] = "White";
char szFirstName[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE author_cursor CURSOR FOR
    SELECT au_fname FROM authors WHERE au_lname = :szLastName;

EXEC SQL OPEN author_cursor;
EXEC SQL FETCH author_cursor INTO :szFirstName;
```

动态游标和静态游标不同。以下是动态游标使用的句法（请参照本小节后面的例子来理解动态游标）。

1）、声明游标：

对于动态游标，在 DECLARE CURSOR 语句中不包含 SELECT 语句。而是，定义了在 PREPARE 中的语句名，PREPARE 语句规定与查询相关的语句名称。具体语法为：

exec sql [at connection_name] declare cursor_name

cursor for statement_name;

如：EXEC SQL DECLARE author_cursor CURSOR FOR select_statement;

值得注意的是，声明动态游标是一个可执行语句，应该在 PREPARE 语句后执行。

2）、打开游标

完整语法为：OPEN 游标名 [USING 主变量名 | DESCRIPTOR 描述名]

在动态游标中,OPEN 语句的作用是使 DBMS 定位相关的游标在第一行查询结果前。当 OPEN 语句成功执行完毕后，游标处于打开状态，并为 FETCH 语句做准备。OPEN 语句执行一条由 PREPARE 语句预编译的语句。如果动态查询正文中包含有一个或多个参数标志时，OPEN 语句必须为这些参数提供参数值。USING 子句的作用就是规定参数值。可以使用主变量提供参数值，也可以通过描述名（即 SQLDA）提供参数值。如：EXEC SQL OPEN author_cursor USING :szLastName;。

3）、取一行值

FETCH 语法为：FETCH 游标名 INTO USING DESCRIPTOR 描述符名。

动态 FETCH 语句的作用是，把游标移到下一行，并把这一行的各列值送到 SQLDA 中。注意的是，静态 FETCH 语句的作用是用主变量表接收查询到的列值。在方法 3 中，使用的是静态 FETCH 语句获得值。动态 FETCH 语句只在方法 4 中使用。

4）、关闭游标

如：EXEC SQL CLOSE c1;

关闭游标的同时，会释放由游标添加的锁和放弃未处理的数据。在关闭游标前，该游标必须已经声明和打开。另外，程序终止时，系统会自动关闭所有打开的游标。

总之,在动态游标的 DECLARE CURSOR 语句中不包含 SELECT 语句。而是,定义了在 PREPARE 中的语句名，用 PREPARE 语句规定与查询相关的语句名称。当 PREPARE 语句中的语句包含了参数，那么在 OPEN 语句中必须指定提供参数值的主变量或 SQLDA。动态 DECLARE CURSOR 语

句是一个可执行语句。该子句必须在 OPEN、FETCH、CLOSE 语句之前使用。请看下面这个例子，描述了完成方法 3 的五个步骤：PREPARE、DECLARE、OPEN、FETCH 和 CLOSE。

```
......
EXEC SQL BEGIN DECLARE SECTION;
char szCommand[] = "SELECT au_fname FROM authors WHERE au_lname = ?";
char szLastName[] = "White";
char szFirstName[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE select_statement FROM :szCommand;
EXEC SQL DECLARE author_cursor CURSOR FOR select_statement;
EXEC SQL OPEN author_cursor USING :szLastName;
EXEC SQL FETCH author_cursor INTO :szFirstName;
EXEC SQL CLOSE author_cursor;
......
```

下面是一个实现方法 3 的实际例子。提示用户输入排序的条件，并把符合条件的书信息显示出来。

```
......
exec sql begin declare section;
CS_CHAR sqlstring[200];
CS_FLOAT bookprice,condprice;
CS_CHAR booktitle[200];
exec sql end declare section;

char orderby[150];

exec sql whenever sqlerror call err_p();
exec sql whenever sqlwarning call warn_p();

strcpy(sqlstring,
    "select title,price from titles\
    where price>? order by ");

printf("Enter the order by clause:");
scanf("%s", orderby);
strcat(sqlstring, orderby);

exec sql prepare select_state from :sqlstring;
exec sql declare select_cur cursor for select_state;

condprice = 10; /* 可以提示用户输入这个值*/

exec sql open select_cur using :condprice;
exec sql whenever not found goto end;
for (;;)
{
    exec sql fetch select_cur
       into :booktitle,:bookprice;
    printf("%20s %bookprice=%6.2f\n",
       booktitle, bookprice);
}

exec sql close select_cur;
exec sql commit work;
......
```

1.2.4.3 SQLDA

要实现方法 4，则需要使用 SQLDA（也可以使用 SQL Descriptors，请读者参阅帮助信息）。可以通过 SQLDA 为嵌入 SQL 语句提供不确定的输入数据和从嵌入 SQ 语句中输出不确定数据。理解 SQLDA 的结构是理解动态 SQL 的关键。

我们知道，动态 SQL 语句在编译时可能不知道有多少列信息。在嵌入 SQL 语句中，这些不确定的数据是通过 SQLDA 完成的。SQLDA 的结构非常灵活，在该结构的固定部分，指明了多少列等信息（如下图中的 sqld=2，表示为两列信息），在该结构的后面，有一个可变长的结构（sd_column 结构），说明每列的信息。

SQLDA 结构
Sd_Sqld=2
Sd_column
……

Sd_datafmt
Sd_Sqllen
Sd_sqldata
…..

Sd_datafmt
Sd_Sqllen
Sd_Sqldata
…..

图 6-2 SQLDA 结构示例

具体 SQLDA 的结构在 sqlda.h 中定义，是：

```
typedef struct _sqlda
{
CS_SMALLINT sd_sqln;
CS_SMALLINT sd_sqld;
struct _sd_column
{
CS_DATAFMT sd_datafmt;
CS_VOID *sd_sqldata;
CS_SMALLINT sd_sqlind;
CS_INT sd_sqllen;
CS_VOID*sd_sqlmore;
} sd_column[1];
} syb_sqlda;
typedef syb_sqlda SQLDA;
```

从上面这个定义看出，SQLDA 是一种由两个不同部分组成的可变长数据结构。从位于 SQLDA 开端的 sd_sqln 到 sd_sqld 为固定部分，用于标志该 SQLDA，并规定这一特定的 SQLDA 的长度。而后是一个或多个 sd_column 结构 ，用于标志列数据或参数。当用 SQLDA 把参数送到执行语句时，每一个参数都是一个 sd_column 结构；当用 SQLDA 返回输出列信息时，每一列都是一个 sd_column 结构。具体每个元素的含义为：

lSd_Sqln。分配的 sd_column 结构的个数。等价于可以允许的最大输入参数的个数或输出列的个数。

lSd_Sqld。目前使用的 sd_column 结构的个数。

lSd_column[].sd_datafmt。标志同列相关的 CS_DATAFMT 结构。

lSd_column[].sd_Sqldata。指向数据的地址。注意，仅仅是一个地址。

lSd_column[].sd_sqllen。sd_sqldata 指向的数据的长度。

lSd_column[].sd_Sqlind。代表是否为 NULL。如果该列不允许为 NULL，则该字段不赋值；如果该列允许为 NULL，则：该字段若为 0，表示数据值不为 NULL，若为-1，表示数据值为 NULL。

lSd_column[].sd_sqlmore。保留为将来使用。

下面我们来看一个具体的例子。这个例子是通过 output_descriptor 查询数据库中的数据，是通过 input_descriptor 传递参数。这个例子的作用是，模拟一个动态查询，并显示查询结果。动态查询的执行过程如下：

1）、如同构造动态 UPDATE 语句或 DELETE 语句的方法一样，程序在缓冲器中构造一个有效的 SELECT 语句。

2）、程序用 PREPARE 语句把动态查询语句送到 DBMS，DBMS 准备、确认和优化语句，并生成一个应用计划。

3）、动态 DECLARE CURSOR 语句说明查询游标，动态 DECLARE CURSOR 语句规定与动态 SELECT 语句有关的语句名称。如：例子中的 statement。

4）、程序用 DESCRIBE 语句请求 DBMS 提供 SQLDA 中描述信息，即告诉程序有多少列查询结果、各列名称、数据类型和长度。DESCRIBE 语句只用于动态查询。具体见下一节。

5）、为 SQLDA 申请存放一列查询结果的存储块（即：sqldata 指向的数据区），也为 SQLDA 的列的指示符变量申请空间。程序把数据区地址和指示符变量地址送入 SQLDA，以告诉 DBMS 向何处回送查询结果。

6）、动态格式的 OPEN 语句。即打开存放查询到的数据集（动态 SELECT 语句产生的数据）的第一行。

7）、动态格式的 FETCH 语句把游标当前行的结果送到 SQLDA。（动态 FETCH 语句和静态 FETCH 语句的不同是：静态 FETCH 语句规定了用主变量接收数据；而动态 FETCH 语句是用 SQLDA 接收数据。）并把游标指向下一行结果集。

8）、CLOSE 语句关闭游标。

具体程序如下：

```
exec sql include sqlca;
exec sql include sqlda;
...
/*input_ descriptor 是通过 SQLDA 传递参数，output_descriptor 是通过 SQLDA 返回
列数据*/
```

```
SQLDA *input_descriptor, *output_descriptor;
CS_SMALLINT small;
CS_CHAR character[20];
/*申请空间*/
input_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
/*设置参数的最大个数*/
input_descriptor->sqlda_sqln = 3;
/*申请空间*/
output_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
/*设置列数的最大值*/
output_descriptor->sqlda_sqln = 3;
*p_retcode = CS_SUCCEED;
/*连接数据库服务器*/
exec sql connect "sa" identified by password;
/* 创建一张 example 表，并插入一些例子数据，用于演示 SQLDA 的使用*/
exec sql drop table example;
exec sql create table example (fruit char(30), number int);
exec sql insert example values ('tangerine', 1);
exec sql insert example values ('pomegranate', 2);
exec sql insert example values ('banana', 3);
/* 准备和描述查询语句*/
exec sql prepare statement from
"select fruit from example where number = ?";
/*describe 语句的作用是，将查询所需要的参数信息存放在 input_descriptor 中*/
exec sql describe input statement using descriptor input_descriptor;
/*设置 SQLDA 中指向参数数据的地址信息（sqldata）和数据长度（sqlda_sqllen）*/
input_descriptor->sqlda_column[0].sqlda_datafmt.datatype =CS_SMALLINT_TYPE;
input_descriptor->sqlda_column[0].sqlda_sqldata = &small;
input_descriptor->sqlda_column[0].sqlda_sqllen = sizeof(small);
small = 2;
/*将查询语句的列信息存放在 output_descriptor 中*/
exec sql describe output statement using descriptor output_descriptor;
if (output_descriptor->sqlda_sqld != 1 ||
   output_descriptor->sqlda_column[0].sqlda_datafmt.datatype !=
CS_CHAR_TYPE)
FAIL;
else
   printf("first describe output \n");
/*设置存放列数据的地址信息*/
output_descriptor->sqlda_column[0].sqlda_sqldata = character;
output_descriptor->sqlda_column[0].sqlda_datafmt.maxlength = 20;
```

```
/*通过 input_descriptor 将输入参数带入查询语句,并将结果通过 output_descriptor
带出*/
exec sql execute statement into descriptor output_descriptor \
using descriptor input_descriptor;
/*打印结果---单行结果*/
printf("expected pomegranate, got %s\n",character);
/*释放申请的内存空间*/
exec sql deallocate prepare statement;
/* 多行结果示例。对多行查询语句做准备和描述操作*/
exec sql prepare statement from \
"select number from example where fruit = ?";
/*为多行结果声明游标*/
exec sql declare c cursor for statement;
exec sql describe input statement using descriptor input_descriptor;
/*设置查询的参数地址信息*/
input_descriptor->sqlda_column->sqlda_sqldata = character;
input_descriptor->sqlda_column->sqlda_datafmt.maxlength =CS_NULLTERM;
/*设置参数值为 banana，也可以提示用户输入这些信息*/
strcpy(character, "banana");
input_descriptor->sqlda_column->sqlda_sqllen = CS_NULLTERM;
/*打开游标*/
exec sql open c using descriptor input_descriptor;
/*设置输出列的信息*/
exec sql describe output statement using descriptor output_descriptor;
/*设置存放数据的地址信息*/
output_descriptor->sqlda_column->sqlda_sqldata = character;
output_descriptor->sqlda_column->sqlda_datafmt.datatype =CS_CHAR_TYPE;
output_descriptor->sqlda_column->sqlda_datafmt.maxlength = 20;
output_descriptor->sqlda_column->sqlda_sqllen = 20;
output_descriptor->sqlda_column->sqlda_datafmt.format =
(CS_FMT_NULLTERM | CS_FMT_PADBLANK);
exec sql fetch c into descriptor output_descriptor;
/*打印列的数据*/
printf("expected pomegranate, got %s\n", character);
exec sql commit work;
……
```

上面这个例子是典型的动态查询程序。该程序中演示了 PREPARE 语句和 DESCRIBE 语句的处理方式，以及为程序中检索到的数据分配空间。要注意程序中如何设置 sqlda_column 结构中的的各个变量。这个程序也演示了 OPEN、FETCH 和 CLOSE 语句在动态查询中的应用。值得注意的是，FETCH 语句只使用了 SQLDA，不使用主变量。由于程序中预先申请了

sqlda_column 结构中的 SQLDATA 空间，所以 DBMS 知道将查询到的数据保存在何处。该程序还考虑了查询数据为 NULL 的处理。

值得注意的是，SQDA 结构不是 SQL 标准。每个数据库厂商的实现方式有可能不同。

1.2.4.4 DESCRIBE语句

该语句只有动态 SQL 才有。该语句是在 PREPARE 语句之后，在 OPEN 语句之前使用。该语句的作用是，设置 SQLDA 中的描述信息，如：列名、数据类型和长度等。DESCRIBE 语句的语法为：

DESCRIBE 语句名 INTO 描述符名

如：exec sql describe output statement using descriptor output_descriptor;。

在执行 DESCRIBE 前，用户必须给出 SQLDA 中的 SQLN 的值（表示最多有多少列），该值也说明了 SQLDA 中最多有多少个 sqlda_column 结构。然后，执行 DESCRIBE 语句，该语句填充每一个 sqlda_column 结构。每个 sqlda_column 结构中的相应列为：

lSd_datafmt 结构：列名等信息。

lSd_sqllen 列：给出列的长度。

注意，sd_sqldata 列不填充。由程序在 FETCH 语句之前，给出数据缓冲器地址和指示符地址。

1.2.5 两个例子程序

1.2.5.1 TELECOM程序

该程序是模拟电信费用查询。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#if defined ( DB2 )
#define SQLNOTFOUND 100
#include <sql.h>
#elif defined ( ORA7 )
#define SQLNOTFOUND 1403
#endif
#if defined (SYBASE)
#define SQLNOTFOUND100
#endif

EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
char user[30];
char passwd[30];
char Usr_name[61];
char Dev_no[9];
long Call_flg;
char Called_arno[11];
char Called_no[15];
char Call_dat[21];
double Call_dur;
double Call_rate;
double Call_fee;
double Add_fee;
char as_dev_no[9];
EXEC SQL END DECLARE SECTION;

void main()
{
    char statusbuf[1024], s[30];
    /*连接到 SQL SERVER 服务器*/
    printf("\nplease enter your userid ");
    gets(user);
    printf("\npassword ");
    gets(passwd);
    exec sql connect :user identified by :passwd;
```

```
    exec sql use pubs2;

    /*输入想要查询的电话号码*/
    printf("\nPlease enter the telephone number:");
    gets(as_dev_no );

    /*声明游标*/
    EXEC SQL DECLARE c1 CURSOR FOR
       SELECT bas_infot.Usr_name, auto10a_list.Dev_no, auto10a_list.Call_flg,
          auto10a_list.Called_arno,auto10a_list.Called_no,
          auto10a_list.Call_dat,auto10a_list.Call_dur, auto10a_list.Call_rate,
          auto10a_list.Call_fee,
       FROM auto10a_list, bas_infot
       WHERE ( auto10a_list.Dev_no = bas_infot.Dev_no )
           AND auto10a_list.Dev_no = :as_dev_no;

    /*打开游标，指向查询相关电话信息的结果集*/
    EXEC SQL OPEN c1; /* :rk.2:erk. */
    do
    {
       /*取出一行数据到各个变量*/
       EXEC SQL FETCH c1 INTO
           :Usr_name, :Dev_no, :Call_flg, :Called_arno, :Called_no, :Call_dat,
           :Call_dur, :Call_rate, :Call_fee, :Add_fee;
       if( (sqlca.sqlcode == SQLNOTFOUND) || (sqlca.sqlcode <0) )
           break;

       /*显示数据*/
       printf("%s,%s,%d,%s,%s,%s,%7.0f,%8.3f,%7.2f,%6.2f\n",
           Usr_name, Dev_no, Call_flg, Called_arno, Called_no, Call_dat,
           Call_dur, Call_rate, Call_fee, Add_fee );
    }while(1);

    EXEC SQL CLOSE c1;
    EXEC SQL DEALLOCATE CURSOR c1;

    Exec sql disconnect all;
    return (0);
}
```

1.2.5.2 ADHOC程序

该程序的功能是：用户输入任意 SQL 语句，并执行和打印结果。

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

/* Defines for BINDING */
/*初试化 SQLDA*/
int init_da (SQLDA **DAPointer, int DAsqln);

/*为存放列数据的 sd_column 结构申请空间*/
int alloc_host_vars (SQLDA *sqldaPointer);

/*释放 SQLDA 所申请的空间*/
void free_da (SQLDA *sqldaPointer);

/*获取列名信息*/
char * readColName (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer);

/*获取列数据*/
char * readCol (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer);

/*定义最大列数*/
#define MAX_COLUMNS255
#define MAX_CHAR_FOR_DOUBLE20
#define MAX_CHAR_FOR_LONG15
#define MAX_CHAR_FOR_DATETIME30
#define MAX_CHAR_FOR_DEFAULT100

EXEC SQL INCLUDE SQLCA ;
EXEC SQL INCLUDE SQLDA ;

#define SQLSTATE sqlca.sqlstate
#define SQLCODE sqlca.sqlcode

/*处理 SQL 语句*/
int process_statement( char * ) ;



int main()
{
    int rc ;
```

```
    char st[1024];
    char tmpstr[1024];

    /*获得 SQL 语句*/
    printf("Please enter the any sql statement:");
    gets( st);

    /* 处理该语句 */
    rc = process_statement( st ) ;

    /*打印处理结果*/
    printf( "%d", rc);
    printf("the sqlcode is %d",SQLCODE);
}


/*****************************************************************************
* FUNCTION : process_statement
* 处理 SQL 语句
*****************************************************************************/
int process_statement ( char * sqlInput )
{
    int counter = 0 ;

    SQLDA * sqldaPointer ;
    short sqlda_d ; /* Total columns */

    short idx;
    char buffer[4096];
    char varname[1024];
    char colnamelist[4096];

    EXEC SQL BEGIN DECLARE SECTION ;
    char st[1024] ;
    EXEC SQL END DECLARE SECTION ;

    strcpy( st, sqlInput ) ;

    /* 为 SQLDA 结构申请空间" */
    if (init_da( &sqldaPointer, MAX_COLUMNS ) == -1)
```

```
{
   return -1;
}

/*准备 SQL 语句*/
EXEC SQL PREPARE statement1 from :st ;
if (SQLCODE < 0)
{
   free_da(sqldaPointer);
   return SQLCODE;
}

/*获取查询列的信息到 SQLDA 结构*/
EXEC SQL DESCRIBE statement1 USING DESCRIPTOR sqldaPointer ;
/* 如果 SQLCODE 为 0，则表示为 SELECT 语句 */
if ( SQLCODE != 0 )
{
   free_da(sqldaPointer);
   return SQLCODE;
} /* end if */

sqlda_d = sqldaPointer->sd_sqld ;
if ( sqlda_d > 0 )
{
   /* 为存放列数据的 sd_column 结构申请空间 */
   if (alloc_host_vars( sqldaPointer ) == -1)
   {
      free_da(sqldaPointer);
      return -1;
   }

   /*声明游标*/
   EXEC SQL DECLARE pcurs CURSOR FOR statement1 ;

   /打开游标*/
   EXEC SQL OPEN pcurs ;
   if (SQLCODE < 0)
      return SQLCODE;

   /*取一行数据到 SQLDA 结构*/
```

```
    EXEC SQL FETCH pcurs INTO DESCRIPTOR sqldaPointer;
    if (SQLCODE < 0)
    {
        EXEC SQL CLOSE pcurs ;
        return SQLCODE;
    }
    /*显示列标题 */
    colnamelist[0] = 0;

    for ( idx=0; idx< sqlda_d; idx++)
    {
        strcat(colnamelist, readColName(sqldaPointer, idx, buffer));
        if (idx < sqlda_d -1)
            strcat(colnamelist, ",");
    }
    /* 显示行数据*/
    while ( SQLCODE == 0 )
    {
        counter++ ;
        for ( idx=0; idx< sqlda_d; idx++)
        printf("%s",readCol(sqldaPointer, idx, buffer));
        EXEC SQL FETCH pcurs INTO DESCRIPTOR sqldaPointer ;
    } /* endwhile */

    /*关闭游标*/
    EXEC SQL CLOSE pcurs ;
    EXEC SQL DEALLOCATE CURSOR pcurs;

    /* 释放为 SQLDA 申请的空间 */
    free_da( sqldaPointer ) ;
}
else
{ /* 不是 SELECT 语句*/
    EXEC SQL EXECUTE statement1 ;
    free_da( sqldaPointer ) ;
    if (SQLCODE < 0)
    return SQLCODE;
} /* end if */

return( 0 ) ;
```

```
} /* end of program : ADHOC.CP */


/*****************************************************************************
*
PROCEDURE : init_da
*为 SQLDA 分配空间。使用 SQLDASIZE 获得 SQLDA 的大小。如果返回-1，则表示分配
*空间不成功。
*****************************************************************************/
int init_da (SQLDA **DAPointer, int DAsqln)
{
    int idx;
    *DAPointer = (SQLDA *)malloc(SYB_SQLDA_SIZE(DAsqln));
    if (*DAPointer == NULL)
       return (-1);

    memset (*DAPointer, '\0', SYB_SQLDA_SIZE(DAsqln));
    (*DAPointer)->sd_sqln = DAsqln;
    (*DAPointer)->sd_sqld = 0;

    return 0;
}


/*****************************************************************************
*
FUNCTION : alloc_host_vars
*为存放列数据的 sd_column 结构申请空间。如果返回-1，则表示不能获得足够内存。
*****************************************************************************/
int alloc_host_vars (SQLDA *sqldaPointer)
{
    short idx;
    for (idx = 0; idx < sqldaPointer->sd_sqld; idx++)
    {
       switch (sqldaPointer->sd_column[idx].sd_datafmt.datatype )
       {
       case CS_CHAR_TYPE:
       case CS_VARCHAR_TYPE:
              sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
              sqldaPointer->sd_column[idx].sd_sqldata = (char *)
```

```
malloc( sqldaPointer->sd_column[idx].sd_sqllen + 1 );
                sqldaPointer->sd_column[idx].sd_sqllen ++;
                sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
            break;
        case CS_TINYINT_TYPE:
        case CS_SMALLINT_TYPE:
        case CS_INT_TYPE:
        case CS_VOID_TYPE:
        case CS_USHORT_TYPE:
                sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
                sqldaPointer->sd_column[idx].sd_sqldata = (char *)
malloc( MAX_CHAR_FOR_LONG );
                sqldaPointer->sd_column[idx].sd_sqllen = MAX_CHAR_FOR_LONG;
                sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
            break;
        case CS_REAL_TYPE:
        case CS_FLOAT_TYPE:
        case CS_BIT_TYPE:
        case CS_MONEY_TYPE:
        case CS_MONEY4_TYPE:
                sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
                sqldaPointer->sd_column[idx].sd_sqldata = (char *)
malloc( MAX_CHAR_FOR_DOUBLE );
                sqldaPointer->sd_column[idx].sd_sqllen = MAX_CHAR_FOR_DOUBLE;
                sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
            break;
        case CS_DATETIME_TYPE:
        case CS_DATETIME4_TYPE:
                sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
                sqldaPointer->sd_column[idx].sd_sqldata = (char *)
malloc( MAX_CHAR_FOR_DATETIME );
                sqldaPointer->sd_column[idx].sd_sqllen = MAX_CHAR_FOR_DATETIME;
                sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
            break;
        case CS_NUMERIC_TYPE:
        case CS_DECIMAL_TYPE:
                sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
                sqldaPointer->sd_column[idx].sd_sqldata = (char *)
malloc( sqldaPointer->sd_column[idx].sd_datafmt.precision + 3 );
                sqldaPointer->sd_column[idx].sd_sqllen =
```

```
sqldaPointer->sd_column[idx].sd_datafmt.precision + 3;
              sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
           break;
       default:
              sqldaPointer->sd_column[idx].sd_datafmt.datatype = CS_CHAR_TYPE;
              sqldaPointer->sd_column[idx].sd_sqldata = (char *)
malloc( MAX_CHAR_FOR_DEFAULT );
              sqldaPointer->sd_column[idx].sd_sqllen = MAX_CHAR_FOR_DEFAULT;
              sqldaPointer->sd_column[idx].sd_datafmt.format = CS_FMT_NULLTERM;
           break;
       } /* endswitch */
       if (sqldaPointer->sd_column[idx].sd_sqldata == NULL)
       {
           return (-1);
       }
    } /* endfor */

    return 0;
}


/*****************************************************************************
*
FUNCTION : free_da
* 释放 SQLDA 申请的空间。
*****************************************************************************/
void free_da (SQLDA *sqldaPointer)
{
    short idx;
    for (idx = 0; idx < sqldaPointer->sd_sqld; idx++)
    {
       free (sqldaPointer->sd_column[idx].sd_sqldata);
    } /* endfor */

    free (sqldaPointer);
}


/*****************************************************************************
*
```

```
PROCEDURE : readColName
* 返回列名
*****************************************************************************/
char * readColName (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer)
{
    strcpy(buffer, sqldaPointer->sd_column[sd_columnIndex].sd_datafmt.name);
    return buffer;
}


/*****************************************************************************
*
PROCEDURE : readCol
* 返回列数据。
*****************************************************************************/
char * readCol (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer)
{
    short numBytes;
    short idx, ind ; /* Array idx variables */

    /* Variables for decoding packed decimal data */
    char tmpstr[1024];
    short collen;
    char *dataptr;

    /* 检查是否为 NULL */
    if ( sqldaPointer->sd_column[sd_columnIndex].sd_sqlind )
    {
       buffer[0] = 0;
       return buffer;
    }

    /*返回列数据到 buffer 变量*/
    strcpy( buffer, (char *)
sqldaPointer->sd_column[ sd_columnIndex ].sd_sqldata);
    return buffer;
}
/* COMMENT OUT OFF */
```

1.3 第三节 IBM DB2 嵌入SQL语言

DB2 支持 SQL 嵌入到 C/C++、JAVA、COBOL、FORTRAN 和 REXX 等语言。本节以 SQL 嵌入 C/C++为例子，讲解静态的嵌入 SQL 编程和动态的嵌入 SQL 编程。

静态 SQL 嵌入 C 语言编程是指，应用程序在书写时，每个 SQL 语句的大部分都已确定下来（如：查询的表、列和语句的格式等），唯一不确定的是查询语句中某些特定变量的值，这些值可以在执行时由变量传进去，但是，值的类型要事先确定。

1.3.1 一个简单示例

首先，我们来看一个嵌入静态 SQL 语句的 C 程序。

例 1、连接到 SAMPLE 数据库，查询 LASTNAME 为 JOHNSON 的 FIRSTNAME 信息。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"
#include <sqlca.h>

EXEC SQL INCLUDE SQLCA;  (1)

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) \
       return 1;

int check_error (char eString[], struct sqlca *caPointer)
{
    char eBuffer[1024];
    char sBuffer[1024];
    short rc, Erc;

    if (caPointer->sqlcode != 0)
    {
       printf ("--- error report ---\n");
       printf ("ERROR occured : %s.\nSQLCODE : %ld\n", eString,
       caPointer->sqlcode);
    }

    return 0;
}

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;  (2)
    char firstname[13];
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf( "Sample C program: STATIC\n" );

    if (argc == 1)
    {
```

```
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3)
    {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;  (3)
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else
    {
        printf ("\nUSAGE: static [userid passwd]\n\n");
        return 1;
    } /* endif */

    EXEC SQL SELECT FIRSTNME INTO :firstname
        FROM employee
        WHERE LASTNAME = 'JOHNSON'; (4)

    CHECKERR ("SELECT statement");  (5)

    printf( "First name = %s\n", firstname );

    EXEC SQL CONNECT RESET;  (6)

    CHECKERR ("CONNECT RESET");

    return 0;
}/* end of program : STATIC.SQC */
```

上面是一个简单的静态嵌入 SQL 语句的应用程序。它包括了静态嵌入 SQL 的主要部分。

（1）中的 include SQLCA 语句定义并描述了 SQLCA 的结构。SQLCA 用于应用程序和数据库之间的通讯，其中的 SQLCODE 返回 SQL 语句执行后的结果状态。

（2）在 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间定义了主变量。主变量可被 SQL 语句引用，也可以被 C 语言语句引用。它用于将程序中的数据通过 SQL 语句传给数据库管理器，或从数据库管理器接收查询的结果。在 SQL 语句中，主变量前均有“：”标志以示区别。

（3）在每次访问数据库之前必须做 CONNECT 操作，以连接到某一个数据库上。这时，应该保证数据库实例已经启动。

（4）是一条选择语句。它将表 employee 中的 LASTNAME 为“JOHNSON”的行数据的 FIRSTNAME 查出，并将它放在 firstname 变量中。该语句返回一个结果。可以通过游标返回多个结果。

（5）在该程序中通过调用宏 CHECKERR（即调用函数 check_error）来返回 SQL 语句执行的结果。Check_error 函数在下面讲解。

（6）最后断开数据库的连接。

从上例看出，每条嵌入式 SQL 语句都用 EXEC SQL 开始，表明它是一条 SQL 语句。这也是告诉预编译器在 EXEC SQL 和“；”之间是嵌入 SQL 语句。如果一条嵌入式 SQL 语句占用多行，在 C 程序中可以用续行符“\”。

1.3.2 嵌入SQL语句

1.3.2.1 宿主变量

1）、声明方法

宿主变量就是在嵌入式 SQL 语句中引用主语言说明的程序变量（如上例中的 firstname 变量）。如：

```
……
EXEC SQL SELECT FIRSTNME INTO :firstname (4)
   FROM employee
   WHERE LASTNAME = 'JOHNSON';
……
```

在嵌入式 SQL 语句中使用宿主变量前，必须采用 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间给宿主变量说明。这两条语句不是可执行语句，而是预编译程序的说明。宿主变量是标准的 C 程序变量。嵌入 SQL 语句使用宿主变量把数据库中查询到的值返回给应用程序（称为输出宿主变量），也用于将程序中给定的值传到 SQL 语句中（称为输入宿主变量）。显然，C 程序和嵌入 SQL 语句都可以访问宿主变量。

在使用宿主变量前，请注意以下几点：

l 宿主变量的长度不能超过 30 字节。开始的字母不能是 EXEC 和 SQL。

l 宿主变量必须在被引用之前定义。

l 一个源程序文件中可以有多个 SQL 说明段。

l 宿主变量名在整个程序中必须是唯一的。

2）、宿主变量的数据类型

宿主变量是一个用程序设计语言的数据类型说明并用程序设计语言处理的程序变量;另外，在嵌入 SQL 语句中用宿主变量保存数据库数据。所以，在嵌入 SQL 语句中，必须映射 C 数据类型为合适的 DB2 数据类型。必须慎重选择宿主变量的数据类型。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
short hostvar1 = 39;
char *hostvar2 = "telescope";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL UPDATE inventory
   SET department = :hostvar1
   WHERE part_num = "4572-3";
EXEC SQL UPDATE inventory
   SET prod_descrip = :hostvar2
   WHERE part_num = "4572-3";
```

在第一个 update 语句中，department 列为 smallint 数据类型，所以应该把 hostvar1 定义为 short 数据类型。这样的话，从 C 到 DB2 的 hostvar1 可以直接映射。在第二个 update 语句中，prod_descip 列为 varchar 数据类型，所以应该把 hostvar2 定义为字符数组。这样的话，从 C 到 DB2 的 hostvar2 可以从字符数组映射为 varchar 数据类型。

下表列出了 C 的数据类型和 DB2 的数据类型的一些转换关系：

DB2 数据类型 C 数据类型

Smallintshort

IntegerLong

Decimal（p，s）无

DoubleDouble

DateChar[11]

TimeChar[9]

TimestampChar[27]

Char（X）Char[X+1]

Varchar（X）Char[X+1]

Graphic（X）Wchar_t[X+1]

Vargraphic（X）Wchar_t[X+1]

因为 C 没有 date 或 time 数据类型，所以 DB2 的 date 或 time 列将被转换为字符。缺省情况下，使用以下转换格式：mm dd yyyy hh:mm:ss[am | pm]。你也可以使用字符数据格式将 C 的字符数据存放到 DB2 的 date 列上。对于 DECIMAL 数据类型，在 C 语言中也没有对应的数据类型。但可以使用 char 数据类型实现。

3）、宿主变量和 NULL

大多数程序设计语言（如 C）都不支持 NULL。所以对 NULL 的处理，一定要在 SQL 中完成。我们可以使用主机指示符变量来解决这个问题。在嵌入式 SQL 语句中，宿主变量和指示符变量共同规定一个单独的 SQL 类型值。指示变量和前面宿主变量之间用一个空格相分隔。如：

```
EXEC SQL SELECT price INTO :price :price_nullflag FROM titles
   WHERE au_id = "mc3026"
```

其中，price 是宿主变量，price_nullflag 是指示符变量。指示符变量的值为：

l-1。表示宿主变量应该假设为 NULL。（注意：宿主变量的实际值是一个无关值，不予考虑）。

l=0。表示宿主变量不是 NULL。

l>0。表示宿主变量不是 NULL。而且宿主变量对返回值作了截断，指示变量存放了截断数据的长度。

所以，上面这个例子的含义是：如果不存在 mc3026 写的书，那么 price_nullflag 为-1，表示 price 为 NULL；如果存在，则 price 为实际的价格。

指示变量也是一种宿主变量，也需要在程序中定义，它对应数据库系统中的数据类型为 SMALLINT。为了便于识别宿主变量，当嵌入式 SQL 语句中出现宿主变量时，必须在变量名称前标上冒号（：）。冒号的作用是，告诉预编译器，这是个宿主变量而不是表名或列名。

1.3.2.2 单行查询

单行查询是通过 SELECT INTO 语句完成。当这条语句执行时，查询的结果送入 INTO 所标志的变量中。如果 SQLCODE 是 100，或者 SQLSTATE 是 02000，则说明没有查询到结果或返回结果为 NULL，这时，宿主变量不改变，否则，宿主变量中将包含查询的结果。如：

```
……
EXEC SQL SELECT FIRSTNME INTO :firstname
   FROM employee
   WHERE LASTNAME = 'JOHNSON';
……
```

1.3.2.3 多行查询

对于多行结果，必须使用游标来完成。游标是一个与 SELECT 语句相关联的符号名，它使用户可逐行访问由 DB2 返回的结果集。下面这个例子演示了游标的使用方法。这个例子的作用是，逐行打印出每个经理的名字和部门。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) \
       return 1;

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char pname[10];
    short dept;
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf( "Sample C program: CURSOR \n" );
    if (argc == 1)
    {
       EXEC SQL CONNECT TO sample;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3)
    {
       strcpy (userid, argv[1]);
       strcpy (passwd, argv[2]);
       EXEC SQL CONNECT TO sample USER :userid USING :passwd;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else
    {
       printf ("\nUSAGE: cursor [userid passwd]\n\n");
       return 1;
    } /* endif */

    EXEC SQL DECLARE c1 CURSOR FOR //(1)
```

```
        SELECT name, dept FROM staff WHERE job='Mgr'
            FOR UPDATE OF job;

    EXEC SQL OPEN c1; //(2)
    CHECKERR ("OPEN CURSOR");

    do
    {
       EXEC SQL FETCH c1 INTO :pname, :dept; //(3)
       if (SQLCODE != 0)
           break;

       printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
           pname, dept );
    } while ( 1 );

    EXEC SQL CLOSE c1;//(4)
    CHECKERR ("CLOSE CURSOR");

    EXEC SQL ROLLBACK;
    CHECKERR ("ROLLBACK");

    printf( "\nOn second thought -- changes rolled back.\n" );

    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");
    return 0;
}
/* end of program : CURSOR.SQC */
```

在上面这个程序中，

（1）定义了一个游标，并指明游标的名字为 C1，同时给出了相对于游标的查询语句和游标类型（UPDATE）。

（2）打开游标。系统执行查询语句，建立结果表，将游标指针指向第一条记录之前。

（3）FETCH 语句将指针的下一条记录取出，将记录中的数据存放在相应的宿主变量中。同时指针下移。

（4）用 CLOSE 关闭游标。

1.3.2.4 插入、删除和修改操作

DB2 中的插入、删除和修改操作同 SQL 语句中 INSERT、DELETE 和 UPDATE 语句类似。只需在相应的 SQL 语句前加上 EXEC SQL 即可。请看下面这个例子：

例、将 staff 表中所有工作为“Mgr”的职工的工作改变为“clerk”，并将 staff 表中所有工作为“sale”的职工信息删除。最后插入一新行。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlenv.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;//(1)

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0)\
          return 1;

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;//(2)
    char statement[256];
    char userid[9];
    char passwd[19];
    char jobUpdate[6];
    EXEC SQL END DECLARE SECTION;

    printf( "\nSample C program: UPDAT \n");
    if (argc == 1)
    {
       EXEC SQL CONNECT TO sample;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3)
    {
       strcpy (userid, argv[1]);
       strcpy (passwd, argv[2]);
       EXEC SQL CONNECT TO sample USER :userid USING :passwd;//(3)
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else
    {
       printf ("\nUSAGE: updat [userid passwd]\n\n");
       return 1;
    } /* endif */
```

```
    strcpy (jobUpdate, "Clerk");

    EXEC SQL UPDATE staff SET job = :jobUpdate WHERE job = 'Mgr';//(4)
    CHECKERR ("UPDATE STAFF");

    printf ("All 'Mgr' have been demoted to 'Clerk'!\n" );
    strcpy (jobUpdate, "Sales");

    EXEC SQL DELETE FROM staff WHERE job = :jobUpdate;//(5)
    CHECKERR ("DELETE FROM STAFF");

    printf ("All 'Sales' people have been deleted!\n");

    EXEC SQL INSERT INTO staff
       VALUES (999, 'Testing', 99, :jobUpdate, 0, 0, 0);//(6)
    CHECKERR ("INSERT INTO STAFF");

    printf ("New data has been inserted\n");

    EXEC SQL ROLLBACK;//(7)
    CHECKERR ("ROLLBACK");

    printf( "On second thought -- changes rolled back.\n" );

    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");

    return 0;
}
/* end of program : UPDAT.SQC */
```

上述语句：

（1）包含 SQLCA 结构。该结构用于将 SQL 语句执行的结果信息返回给应用程序。

（2）宿主变量定义。

（3）连接到 DB2 的 SAMPLE 数据库。

（4）UPDATE 语句将 staff 表中所有工作为“Mgr”的职工的工作改变为“clerk”。

（5）DELETE 语句将 staff 表中所有工作为“sale”的职工信息删除。

（6）INSERT 语句插入一新行。

指定位置的 UPDATE 语句和 DELETE 语句

游标操作除了可以将多行的查询结果返回给应用程序，它还可以与 UPDATE 语句和 DELETE 相结合，根据游标当前的位置，对指针所指的这个行数据执行 UPDATE 操作和 DELETE 操作。它的语法为：

UPDATE…. WHERE CURRENT OF cursor_name

DELETE [FROM] {table_name | view_name} WHERE CURRENT OF cursor_name

请看下面这个例子。这个例子的作用是：将部门大于 40 的员工的工作改变为“clerk”，部门小于等于 40 的员工信息删除。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0)
          return 1;


int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char pname[10];
    short dept;
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf( "Sample C program: OPENFTCH\n" );
    if (argc == 1)
    {
       EXEC SQL CONNECT TO sample;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3)
    {
       strcpy (userid, argv[1]);
       strcpy (passwd, argv[2]);
       EXEC SQL CONNECT TO sample USER :userid USING :passwd;
       CHECKERR ("CONNECT TO SAMPLE");
```

```
}
else
{
   printf ("\nUSAGE: openftch [userid passwd]\n\n");
   return 1;
} /* endif */

EXEC SQL DECLARE c1 CURSOR FOR
SELECT name, dept FROM staff WHERE job='Mgr'
   FOR UPDATE OF job;

EXEC SQL OPEN c1;
CHECKERR ("OPEN CURSOR");

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;
      CHECKERR ("UPDATE STAFF");
   }
   else
   {
      printf ("%-10.10s in dept. %2d will be DELETED!\n",
      pname, dept);

      EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
      CHECKERR ("DELETE");
   } /* endif */
} while ( 1 );

EXEC SQL CLOSE c1;
CHECKERR ("CLOSE CURSOR");
```

```
    EXEC SQL ROLLBACK;
    CHECKERR ("ROLLBACK");

    printf( "\nOn second thought -- changes rolled back.\n" );

    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");

    return 0;
}
/* end of program : OPENFTCH.SQC */
```

在通过游标进行 UPDATE 和 DELETE 操作时，相应的游标在定义时必须加上 FOR UPDATE 子句。操作时，游标必须打开，而且正指向一行数据。

1.3.2.5 SQLCA

应用程序执行时，每执行一条 SQL 语句，就返回一个状态符和一些附加信息。这些信息反映了 SQL 语句或 API 的执行情况，它有助于用户分析应用程序的错误所在。这些信息都存放在一个定义在 sqlca.h 的 sqlca 结构中。如果一个源文件中后 SQL 语句，则必须要在源程序中定义一个 SQLCA 结构，而且名为 SQLCA。最简单的定义方法是在源文件中加入一些语句：

```
EXEC SQL INCLUDE sqlca.h
```

下面，我们首先看看 SQLCA 的结构：

```
SQL_STRUCTURE sqlca
{
   _SQLOLDCHAR sqlcaid[8]; /* Eyecatcher = 'SQLCA ' */
   long sqlcabc; /* SQLCA size in bytes = 136 */
   #ifdef DB2_SQL92E
   long sqlcade; /* SQL return code */
   #else
   long sqlcode; /* SQL return code */
   #endif
   short sqlerrml; /* Length for SQLERRMC */
   _SQLOLDCHAR sqlerrmc[70]; /* Error message tokens */
   _SQLOLDCHAR sqlerrp[8]; /* Diagnostic information */
   long sqlerrd[6]; /* Diagnostic information */
   _SQLOLDCHAR sqlwarn[11]; /* Warning flags */
   #ifdef DB2_SQL92E
   _SQLOLDCHAR sqlstat[5]; /* State corresponding to SQLCODE */
   #else
   _SQLOLDCHAR sqlstate[5]; /* State corresponding to SQLCODE */
```

```
    #endif
};
```

结构中各个字段的作用是：

lSqlcaid：包含字符串“SQLCA”。

lSqlcabc：包含 SQLCA 结构的长度。

lSqlcode：该值反映了 SQL 语句执行后的状态，0 表示 SQL 执行成功；<0 表示 SQL 语句执行出错；>0 反映了一些特殊情况（如：没有查询结果）。不同的数据库产品，该值代表的含义可能不同。

lSqlerrml：sqlerrmc 域中数据的实际长度。

lSqlerrmc：由 0 个或多个字符串组成，它对返回的值给予一个更详细的解释。

lSqlerrp：包含一些对用户没有用的信息。

lSqlwarn：包含了一些警告信息。

lSqlstate：长度为 5 的字符串。它表示 SQL 语句执行的结果。它的每一个含义是遵循 ANSI/SQL 92 标准。各个数据库产品的 sqlstate 域的含义都是相同的。

为了更方便地读取 sqlca 中 SQL 语句执行后的结果或错误，DB2 提供了一个函数——sqlaintp,它在 sql.h 中声明:sqlaintp(msgbuf,bufsize,linesize,sqlcaptr)。其中 msgbuf 中存放信息；bufsize 中存放了 msgbuf 的长度；linesize 中存放了两个执行符之间的字符长度。函数的返回值为正，代表信息的长度；为负代表没有信息返回。

下面这个例子解释了 sqlca 和 sqlaintp 的使用方法：

```
int check_error (char eString[], struct sqlca *caPointer)
{
    char eBuffer[1024];
    char sBuffer[1024];
    short rc, Erc;
    /*通过 SQLCODE 来判断是否出错*/
    if (caPointer->sqlcode != 0)
    {
       printf ("--- error report ---\n");
       printf ("ERROR occured : %s.\nSQLCODE : %ld\n", eString,
          caPointer->sqlcode);
       /* 获取 SQLSTATE 信息*/
       rc = sqlogstt (sBuffer, 1024, 80, caPointer->sqlstate);
       /*获取调用 API 的错误信息*/
       Erc = sqlaintp (eBuffer, 1024, 80, caPointer);
       /* Erc 中存放了 eBuffer 的长度*/
       if (Erc > 0)
          printf ("%s", eBuffer);
       if (caPointer->sqlcode < 0)
       {  /*错误信息*/
```

```
            if (rc == 0)
            {
                printf ("\n%s", sBuffer);
            }
            printf ("--- end error report ---\n");
            return 1;
        }
        else
        {  /* 仅仅是警告信息 */
            if (rc == 0)
            {
                printf ("\n%s", sBuffer);
            }
            printf ("--- end error report ---\n");
            printf ("WARNING - CONTINUING PROGRAM WITH WARNINGS!\n");
            return 0;
        } /* endif */
    } /* endif */

    return 0;
}
```

在每条 SQL 语句执行后都返回一个 SQLCA 结构，SQLCA 结构中记载了 SQL 语句执行后的结果信息。用户可以根据返回信息执行各种操作。DB2 也提供了 WHENEVER 语句。具体可参见 SQL SERVER 中的 WHENEVER。但是，在 DB2 中，没有 WHENEVER…CALL 这个处理。

1.3.2.6 事务

所谓事务,就是一系列应用程序和数据库之间交互操作的集合。一旦一个事务开始执行，则事务中的操作要么全部执行，要么全部不执行。

1 事务开始：DB2 事务是隐式开始的，除了下列的一些语句，其他任何一个可执行的 SQL 语句都隐式地开始一个事务。

BEGIN DECLARE SECTION END DECLARE SECTION

DECLARE CURSOR INCLUDE SQLCA INCLUDE SQLDA

WHENEVER

1 事务结束：事务由一个可执行的 SQL 语句开始，后面执行的所有 SQL 语句都将属于同一个事务，该事务一直遇到 COMMIT 或 ROLLBACK 命令时才结束。

COMMIT 操作的作用是，结束当前的事务，事务对数据库所做的修改永久化。ROLLBACK 的作用是，结束当前的事务，将被修改的数据恢复到事务执行以前的状态，即取消事务执行产生的影响。当程序结束时，系统自动隐式地执行 COMMIT 操作，如果系统检测到死锁等故障，则隐式地执行 ROLLBACK 操作。

1.3.3 DB2 的嵌入SQL程序处理过程

嵌入 SQL 程序处理，由一个源程序创建为一个可执行文件的过程。如下图所示：

图 6-3 嵌入 SQL 处理过程

从上图看出，首先对源文件做预编译（precompiler），生成两个部分文件：一部分是纯的 C 程序源文件，它们和其他的 C 程序源文件一起，经过编译和连接生成可执行的程序（executable program）；而另一部分是 bind 文件或 package 文件。Bind 文件经过 binder 操作以后，也生成为 package 文件。所谓 package，实际上是 SQL 语句的访问计划。所以，预编译器将源程序中的 SQL 语句提出来，生成他们的访问计划，并将访问计划存放在数据库管理器中。当执行程序并遇到访问数据库的命令时，它将到数据库管理器中寻找属于它的访问计划，然后按照访问计划中所设计的方法对数据库进行访问。具体来说：

第一步、预编译

源程序生成以后，在源程序中嵌入了许多 SQL 语句，而 SQL 语句是宿主语言编译器所不认识的，所以在用宿主语言编译器进行编译、连接之前必须将 SQL 语句分离出来，这就是预编译所做的工作。DB2 中预编译操作是通过 PREP 命令执行的，PREP 命令首先将源程序中的所有有关 SQL 语句全部注释起来，对它进行分析和语法检查。如果源程序中的 SQL 语句全部书写正确，则将这些 SQL 语句转换成 C 语言可以识别的一系列的 API 函数。这些函数可以在函数执行时访问数据库，然后将源文件中所有用于生成数据库管理器的 PACKAGE 的数据提出组合成一个 BIND 文件。也可以直接生成一个 PACKAGE，但这相当于在预编译后又执行了一次 BIND 操作。在预编译时，对整个源程序文件中的所有变量做统一处理而不根据变量的生命期来处理，所以宿宿主变量在整个程序中是唯一的。下面讲解预编译的步骤：

1）、连接到一个数据库，该操作是为 BIND 做准备。操作如下：

db2 connect to cicstest

2）、执行预编译命令，假设源文件为 adhoc.sqc，则：

db2 prep adhoc.sqc bindfile

下面我们对预编译的几种输出文件进行讨论。

l 预编译后生成的 C 语言源文件：该文件中原有的 SQL 语句，已经全部加上注释并转换成了 C 语言可以识别的 API 调用。

lBIND 文件：如果在预编译时使用 BINDFILE 选项，则生成 BIND 文件，BIND 文件的后缀为.bnd，BIND 文件可以在将来使用 BIND 命令来生成 PACKAGE。

db2 bind adhoc.bnd

如果在预编译时，只生成 BIND 文件，那么即使在预编译时，不能访问某些数据库对象，系统也只是报警，而不会报错。如果使用 PACKAGE 选项，则生成 PACKAGE。如果有 MESSAGE 选项，则生成信息文件，它包含了所有的返回信息，如：警报、错误等。它便于程序员对源程序做进一步的修改。

第二步、编译和连接

在预编译后，程序中只有 C 语言语句，它们都可以为 C 语言的编译器所识别。所以，可以按照一般的方法进行编译和连接，但在将 SQL 语句转换以后，在 C 语言程序中，又引入了

许多一般的 C 语言系统所没有的 INCLUDE 文件和函数库，这些均在 DB2 的 SDK 中。所以，要生成可执行的程序，就必须安装 DB2 的 SDK，并且做以下设置：

```
set INCLUDE=$(DB2PATH)\include;%include%
set LIB=$(DB2PATH)\lib;%LIB%
```

下面是编译和连接：

```
cl -o adhoc.exe adhoc.c
```

生成的可执行文件必须与数据库管理器中的 PACKAGE 相结合，才能执行。

下面对 BIND 做进一步解释。

PACKAGE 是 DB2 为 SQL 语句制定的访问计划。通过 precompile 之后，源程序中的 SQL 语句部分就被分离出来。PACKAGE 就是根据具体的 SQL 语句和数据库中的信息生成的针对每条 SQL 语句的访问计划，它存放在 DB2 数据库服务器上。应用程序执行到 SQL 语句时，就到相应的服务器上去找它们的 PACKAGE，数据库服务器根据 PACKAGE 执行具体的数据库操作。所以，如果一个应用程序访问了多个数据库服务器，则该应用程序应在它访问到的数据库服务器上均生成相应的 PACKAGE。因此，当遇到这种情况时，推荐的方式是将源程序分成若干个文件，每个文件只访问一个服务器，然后分别进行预编译。

执行 PREP 命令时，加上选项 PACKAGE 或不注明 BINDFILE、SYNTAX 或 SQLFLAG 选项，这时 BIND 操作将自动进行。

直接使用 BIND 命令从 BIND 文件中生成 PACKAGE 存放在数据库管理器中。BIND 完成的功能是，从数据库中找到 SQL 语句所涉及的表，查看 SQL 语句中提到的表名及属性是否与数据库中的表名和属性相匹配，以及应用程序开发者是否有权限查询或修改应用程序中所涉及到的表及属性。这就是为什么在预编译之前要连接到相应的数据库上的原因。在做 BIND 操作后，在数据库管理器中就生成一个 PACKAGE，PACKAGE 的名字与源程序的文件名字相同。

一个源文件在数据库中可有多个 PACKAGE 存在。为了区分不同的 PACKAGE，DB2 中引入了时间戳的概念。在 PREP 执行时，系统对生成的修改过的 C 语言程序和 BIND 文件以及 PACKAGE 中都加入了一个时间戳。BIND 文件在生成 PACKAGE 时也将时间戳传递下来，修改过的 C 语言程序在生成可执行程序时，同样也将时间戳传递下去。当应用程序运行时，可执行程序是通过时间戳找到相应的 PACKAGE，如果时间戳不匹配，则说明版本更新，需要做 BIND。

本章第一个例子被编译后所生成的 C 程序：

```
static char sqla_program_id[40] =
{111,65,65,66,65,73,65,70,89,65,78,71,90,72,32,32,68,69,77,79,
68,66,50,32,67,65,51,54,75,75,67,81,48,49,49,49,49,50,32,32};
#include "sqladef.h"
static struct sqla_runtime_info sqla_rtinfo =
{{'S','Q','L','A','R','T','I','N'}, sizeof(wchar_t), 0, {' ',' ',' ',' '}};
#line 1 "demodb2.sqc"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"
#include <sqlca.h>
/*
```

```
EXEC SQL INCLUDE SQLCA;
*/
/* SQL Communication Area - SQLCA - structures and constants */
#include "sqlca.h"
struct sqlca sqlca;
#line 6 "demodb2.sqc"
#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;
int check_error (char eString[], struct sqlca *caPointer) {
char eBuffer[1024];
char sBuffer[1024];
short rc, Erc;
if (caPointer->sqlcode != 0) {
printf ("--- error report ---\n");
printf ("ERROR occured : %s.\nSQLCODE : %ld\n", eString,
caPointer->sqlcode);
}
return 0;
}
int main(int argc, char *argv[]) {
/*
EXEC SQL BEGIN DECLARE SECTION;
*/
#line 21 "demodb2.sqc"
char firstname[13];
char userid[9];
char passwd[19];
/*
EXEC SQL END DECLARE SECTION;
*/
#line 25 "demodb2.sqc"
printf( "Sample C program: STATIC\n" );
if (argc == 1) {
/*
EXEC SQL CONNECT TO sample;
*/
{
#line 28 "demodb2.sqc"
sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
#line 28 "demodb2.sqc"
sqlaaloc(2,1,1,0L);
{
```

```
struct sqla_setd_list sql_setdlist[1];
#line 28 "demodb2.sqc"
sql_setdlist[0].sqltype = 460; sql_setdlist[0].sqllen = 7;
#line 28 "demodb2.sqc"
sql_setdlist[0].sqldata = (void*)"sample";
#line 28 "demodb2.sqc"
sql_setdlist[0].sqlind = 0L;
#line 28 "demodb2.sqc"
sqlasetd(2,0,1,sql_setdlist,0L);
}
#line 28 "demodb2.sqc"
sqlacall((unsigned short)29,4,2,0,0L);
#line 28 "demodb2.sqc"
sqlastop(0L);
}
#line 28 "demodb2.sqc"
CHECKERR ("CONNECT TO SAMPLE");
}
else if (argc == 3) {
strcpy (userid, argv[1]);
strcpy (passwd, argv[2]);
/*
EXEC SQL CONNECT TO sample USER :userid USING :passwd;
*/
{
#line 34 "demodb2.sqc"
sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
#line 34 "demodb2.sqc"
sqlaaloc(2,3,2,0L);
{
struct sqla_setd_list sql_setdlist[3];
#line 34 "demodb2.sqc"
sql_setdlist[0].sqltype = 460; sql_setdlist[0].sqllen = 7;
#line 34 "demodb2.sqc"
sql_setdlist[0].sqldata = (void*)"sample";
#line 34 "demodb2.sqc"
sql_setdlist[0].sqlind = 0L;
#line 34 "demodb2.sqc"
sql_setdlist[1].sqltype = 460; sql_setdlist[1].sqllen = 9;
#line 34 "demodb2.sqc"
sql_setdlist[1].sqldata = (void*)userid;
```

```
#line 34 "demodb2.sqc"
sql_setdlist[1].sqlind = 0L;
#line 34 "demodb2.sqc"
sql_setdlist[2].sqltype = 460; sql_setdlist[2].sqllen = 19;
#line 34 "demodb2.sqc"
sql_setdlist[2].sqldata = (void*)passwd;
#line 34 "demodb2.sqc"
sql_setdlist[2].sqlind = 0L;
#line 34 "demodb2.sqc"
sqlasetd(2,0,3,sql_setdlist,0L);
}
#line 34 "demodb2.sqc"
sqlacall((unsigned short)29,5,2,0,0L);
#line 34 "demodb2.sqc"
sqlastop(0L);
}
#line 34 "demodb2.sqc"
CHECKERR ("CONNECT TO SAMPLE");
}
else {
printf ("\nUSAGE: static [userid passwd]\n\n");
return 1;
} /* endif */
/*
EXEC SQL SELECT FIRSTNME INTO :firstname
FROM employee
WHERE LASTNAME = 'JOHNSON';
*/
{
#line 44 "demodb2.sqc"
sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
#line 44 "demodb2.sqc"
sqlaaloc(3,1,3,0L);
{
struct sqla_setd_list sql_setdlist[1];
#line 44 "demodb2.sqc"
sql_setdlist[0].sqltype = 460; sql_setdlist[0].sqllen = 13;
#line 44 "demodb2.sqc"
sql_setdlist[0].sqldata = (void*)firstname;
#line 44 "demodb2.sqc"
sql_setdlist[0].sqlind = 0L;
```

```
#line 44 "demodb2.sqc"
sqlasetd(3,0,1,sql_setdlist,0L);
}
#line 44 "demodb2.sqc"
sqlacall((unsigned short)24,1,0,3,0L);
#line 44 "demodb2.sqc"
sqlastop(0L);
}
#line 44 "demodb2.sqc"
CHECKERR ("SELECT statement");
printf( "First name = %s\n", firstname );

/*
EXEC SQL CONNECT RESET;
*/
{
#line 47 "demodb2.sqc"
sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
#line 47 "demodb2.sqc"
sqlacall((unsigned short)29,3,0,0,0L);
#line 47 "demodb2.sqc"
sqlastop(0L);
}
#line 47 "demodb2.sqc"
CHECKERR ("CONNECT RESET");
return 0;
}
/* end of program : STATIC.SQC */
```

生成的 BIND 文件为 demodb2.bnd。

1.3.4 DB2 的动态SQL嵌入语句

所谓静态 SQL 的编程方法，就是指在预编译时 SQL 语句已经基本确定，即访问的表或视图名、访问的列等信息已经确定。但是，有时整个 SQL 语句要到执行的时候才能确定下来，而且 SQL 语句所访问的对象也要到执行时才能确定。这就需要通过动态 SQL 语句完成。

1.3.4.1 基本方法

执行动态 SQL 语句的程序，主要有三条语句来完成：

1）、PREPARE 语句。由于动态 SQL 语句在执行时才能确定，所以 DB2 中使用一个字符型的宿主变量来存放相应的 SQL 语句。但是，这条语句在预编译时不存在，因而在预编译时无法编译这个 SQL 语句。但是，存放在宿主变量中的语句必须转化为可执行的格式后才能执行。PREPARE 命令的作用是完成编译的工作。即相当于静态 SQL 中的 BIND 操作，在数据库管理器中生成 PACKAGE，并给出一个语句的名字。在后面的 DESCRIBE、EXECUTE 和 OPEN 等命令都使用这个名字来访问相应的 SQL 语句。SQL 语句在做 PREPARE 操作时，不可使用宿主变量来表示参数,但可以在相应的位置上使用“？”来表示该位置上应该有一个参数。PREPARE 还将生成相应的 SQLDA 结构。

2）、EXECUTE 语句。该语句的作用是执行已经做过 PREPARE 操作的 SQL 语句。在 EXECUTE 语句中，将针对 PREPARE 语句中的每个参数标志给出相应的参数值，这些参数值可以有宿主变量传递，他们的类型应该匹配。EXECUTE 命令不能做 SELECT 操作，这是因为 EXECUTE 语句无法返回结果，要执行 SELECT 操作，应该通过游标完成。

EXECUTE IMMEDIATE 语句是 PERPARE 语句和 EXECUTE 语句的综合，它对语句做 PREPARE，生成可执行模式，在执行。

3）、DESCRIBE 语句。将执行过的 PREPARE 的 SQL 语句结果信息存放在 SQLDA 结构。

例、查询表名不是 STAFF 的表信息。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0)
          return 1;


int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char table_name[19];
    /*st[80]宿主变量存放 SQL 语句*/
    char st[80];
    char parm_var[19];
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf( "Sample C program: DYNAMIC\n" );

    if (argc == 1)
    {
       EXEC SQL CONNECT TO sample;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3)
    {
       strcpy (userid, argv[1]);
       strcpy (passwd, argv[2]);
       EXEC SQL CONNECT TO sample USER :userid USING :passwd;
       CHECKERR ("CONNECT TO SAMPLE");
    }
    else
    {
       printf ("\nUSAGE: dynamic [userid passwd]\n\n");
```

```
        return 1;
    } /* endif */

    strcpy( st, "SELECT tabname FROM syscat.tables" );
    strcat( st, " WHERE tabname <> ?" );

    /*对 st 做 PREPARE 操作，st 中的“？”表示参数*/
    EXEC SQL PREPARE s1 FROM :st;
    CHECKERR ("PREPARE");

    /*定义游标*/
    EXEC SQL DECLARE c1 CURSOR FOR s1;
    strcpy( parm_var, "STAFF" );

    /*打开游标，并用 parm_var 代替 SELECT 语句中的“？”参数*/
    EXEC SQL OPEN c1 USING :parm_var;
    CHECKERR ("OPEN");

    do
    {
       /*用 FETCH 语句从结果集中取出结果*/
       EXEC SQL FETCH c1 INTO :table_name;
       if (SQLCODE != 0) break;
       printf( "Table = %s\n", table_name );
    } while ( 1 );

    /*关闭游标*/
    EXEC SQL CLOSE c1;
    CHECKERR ("CLOSE");

    EXEC SQL COMMIT;
    CHECKERR ("COMMIT");

    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");

    return 0;
}
/* end of program : DYNAMIC.SQC */
```

从上面这个例子看出，动态 SQL 语句同静态 SQL 语句的不同之处在于，要使用 PREPARE 语句操作具体的 SQL 语句，然后使用 EXECUTE 或者游标等方式执行。

1.3.4.2 动态游标

1）、动态游标的 DECLARE 语句

动态游标对应的SQL语句应该是一个用PREPARE操作从文本形式转换趁个可执行形式的语句。在上例子中： EXEC SQL DECLARE c1 CURSOR FOR s1; 其中 s1 就是 PREPARE 操作后的可执行语句，c1 是游标的名字。

2）、动态游标的 OPEN 语句

动态 OPEN 操作的作用是：将宿主变量或 SQLDA 结构中的值取出，填充到 PREPARE 操作后的 SQL 语句中标有“？”的参数的位置，并执行查询。宿主变量或 SQLDA 结构中的值应该与参数有一一对应的关系，而且数据类型符合。如上例子中：

```
EXEC SQL OPEN c1 USING :parm_var;
```

3）、动态游标的 FETCH 语句

FETCH 语句是从结果集中取出一行，将结果送入宿主变量列表或 SQLDA 结构中。关于 SQLDA 结构，见下节。如上例子中：

```
EXEC SQL FETCH c1 INTO :table_name;
```

1.3.4.3 SQLDA

SQLDA 结构的作用与宿主变量相同，可用于应用程序与数据库之间的数据交换，它适用于：在编写程序时不确定要使用的变量个数和数据类型或不确定输出数据的列数，如：动态输入的 SELECT 语句，就必须使用 SQLDA 结构来获得查询的数据。SQLDA 用于描述数据的类型、长度、变量的值和数据项的个数。在应用程序中使用 SQLDA，必须在程序中定义 SQLDA。即加入以下语句：EXEC SQL INCLUDE SQLDA。

1）、SQLDA 结构

SQLDA 的定义存放在 sqlda.h 中。具体为：

```
SQL_STRUCTURE sqlname               /* Variable Name */
{
    short length;                   /* Name length [1..30] */
    _SQLOLDCHAR data[30];           /* Variable or Column name */
};

SQL_STRUCTURE sqlvar /* Variable Description */
{
    short sqltype;                  /* Variable data type */
    short sqllen;                   /* Variable data length */
    _SQLOLDCHAR *SQL_POINTER sqldata; /* Pointer to variable data value */
    short *SQL_POINTER sqlind;      /* Pointer to Null indicator */
    struct sqlname sqlname;         /* Variable name */
};

SQL_STRUCTURE sqlda
{
    _SQLOLDCHAR sqldaid[8];         /* Eye catcher = 'SQLDA ' */
    long sqldabc;                   /* SQLDA size in bytes=16+44*SQLN */
    short sqln;                     /* Number of SQLVAR elements */
    short sqld;                     /* # of columns or host vars. */
    struct sqlvar sqlvar[1];        /* first SQLVAR element */
};
```

下图形象的描述了用 SQLDA 来存放两列数据。

SQLDA 结构

Sqld=2

sqlvar

……

Sqltype=500

Sqllen

sqldata

…..

Sqltype=501

Sqllen

Sqldata

…..

图 6-4 SQLDA 结构示例

从上面这个定义看出，SQLDA 是一种由两个不同部分组成的可变长数据结构。从位于 SQLDA 开端的 sqldaid 到 sqld 为固定部分，用于标志该 SQLDA，并规定这一特定的 SQLDA 的长度。而后是一个或多个 sqlvar 结构 ，用于标志列数据。当用 SQLDA 把参数送到执行语句时，每一个参数都是一个 sqlvar 结构；当用 SQLDA 返回输出列信息时，每一列都是一个 sqlvar 结构。具体每个元素的含义为：

lSqldaid。用于输入标志信息，如：“SQLDA”。

lSqldabc。SQLDA 数据结果的长度。应该是 16+44*SQLN。Sqldaid、sqldabc、sqln 和 sqld 的总长度为 16 个字节。而 sqlvar 结构的长度为 44 个字节。

lSqln。分配的 Sqlvar 结构的个数。等价于输入参数的个数或输出列的个数。

lSqld。目前使用的 sqlvar 结构的个数。

lSqltype。代表参数或列的数据类型。它是一个整数数据类型代码。如：500 代表 smallint。具体每个整数的含义见下表：

SQL 列类型	SQLTYPE 值	SQLTYPE 的字符名
DATE	384/385	SQL_TYP_DATE/SQL_TYP_NDATE
TIME	388/389	SQL_TYP_TIME/SQL_TYP_NTIME
TIMESTAMP	392/393	SQL_TYP_STAMP/SQL_TYP_NSTAMP
n/a2	400/401	SQL_TYP_CGSTR/SQL_TYP_NCGSTR
BLOB	404/405	SQL_TYP_BLOB/SQL_TYP_NBLOB
CLOB	408/409	SQL_TYP_CLOB/SQL_TYP_NCLOB
DBCLOB	412/413	SQL_TYP_DBCLOB/SQL_TYP_NDBCLOB
VARCHAR	448/449	SQL_TYP_VARCHAR/SQL_TYP_NVARCHAR
CHAR	452/453	SQL_TYP_CHAR/SQL_TYP_NCHAR
LONG VARCHAR	456/457	SQL_TYP_LONG/SQL_TYP_NLONG
n/a3	460/461	SQL_TYP_CSTR/SQL_TYP_NCSTR
VARGRAPHIC	464/465	SQL_TYP_VARGRAPH/SQL_TYP_NVARGRAPH
GRAPHIC	468/469	SQL_TYP_GRAPHIC/SQL_TYP_NGRAPHIC
LONG VARGRAPHIC	472/473	SQL_TYP_LONGRAPH/SQL_TYP_NLONGRAPH
FLOAT	480/481	SQL_TYP_FLOAT/SQL_TYP_NFLOAT
REAL4	480/481	SQL_TYP_FLOAT/SQL_TYP_NFLOAT
DECIMAL5	484/485	SQL_TYP_DECIMAL/SQL_TYP_DECIMAL
INTEGER	496/497	SQL_TYP_INTEGER/SQL_TYP_NINTEGER
SMALLINT	500/501	SQL_TYP_SMALL/SQL_TYP_NSMALL
n/a	804/805	SQL_TYP_BLOB_FILE/SQL_TYPE_NBLOB_FILE
n/a	808/809	SQL_TYP_CLOB_FILE/SQL_TYPE_NCLOB_FILE
n/a	812/813	SQL_TYP_DBCLOB_FILE/SQL_TYPE_NDBCLOB_FILE
n/a	960/961	SQL_TYP_BLOB_LOCATOR/SQL_TYP_NBLOB_LOCATOR
n/a	964/965	SQL_TYP_CLOB_LOCATOR/SQL_TYP_NCLOB_LOCATOR
n/a	968/969	SQL_TYP_DBCLOB_LOCATOR/SQL_TYP_NDBCLOB_LOCATOR

我们知道，SQLDA 的作用是应用程序与数据库之间交换数据。对于从数据库向应用程序输出数据，则 SQLDA 存放了每列的信息，如：数据类型、长度和值。输出数据要同 FETCH 语句结合；对于从应用程序向数据库输入数据，则要同 OPEN 或 EXECUTE 操作结合。请看下面这个 ADHOC 例子，来理解 SQLDA 的作用。这个例子很经典，它的功能是处理任意输入的 SQL 语句，并返回结果。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlcodes.h>

/*包含 SQLDA 结构的定义*/
#include <sqlda.h>
#include "util.h"

/*包含 SQLCA 结构的定义*/
EXEC SQL INCLUDE SQLCA;

/* 'check_error' is a function found in the util.c program */
#define CHECKERR(CE_STR) check_error (CE_STR, &sqlca)
#define SQLSTATE sqlca.sqlstate

/*初试化 SQLDA*/
int init_da (SQLDA **DAPointer, int DAsqln);

/*为存放列数据的 sd_column 结构申请空间*/
int alloc_host_vars (SQLDA *sqldaPointer);

/*释放 SQLDA 所申请的空间*/
void free_da (SQLDA *sqldaPointer);

/*获取列名信息*/
char * readColName (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer);

/*获取列数据*/
char * readCol (SQLDA *sqldaPointer, short sd_columnIndex, char * buffer);

/*处理 SQL 语句*/
int process_statement (char[1000]);
```

```
#define MAX_COLUMNS 255

int main(void)
{
    int rc;
    char sqlInput[255];
    char st[1000]="";
    char Transaction;
    char tmpstr[1024];

    /*定义宿主变量*/
    EXEC SQL BEGIN DECLARE SECTION; (3)
    char server[9];
    char userid[9];
    char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf("Sample C program : ADHOC interactive SQL\n");

    /* 初试化与数据库的连接 */
    do
    {
       printf("input the server (database) which you wish to attach to : \n");
       gets (sqlInput);
       strcpy(server,sqlInput);
       printf("input your userid : \n");
       gets (sqlInput);
       strcpy (userid, sqlInput);
       printf("input your passwd : \n");
       gets (sqlInput);
       strcpy (passwd, sqlInput);
       printf("CONNECTING TO %s\n",server);
       /*连接数据库*/
       EXEC SQL CONNECT TO :server USER :userid USING :passwd;
       /*检查连接是否成功*/
       CHECKERR("CONNECT TO DATABASE");
    } while (SQLCODE != 0); /* enddo */

    printf("CONNECTED TO %s\n",server);
```

```
/* Enter the continuous command line loop. */
while ( 1 )
{
   /*提示用户输入要操作的 SQL 语句*/
   printf ("Enter an SQL statement or 'quit' to Quit :\n");
   gets (sqlInput);
   if (strcmp(sqlInput, "quit") == 0)
   {
      break;
   }
   else if (strlen(sqlInput) == 0)
   {
      /* Don't process the statement */
      printf ("\tNo characters entered.\n");
   }
   else if (sqlInput[strlen(sqlInput) - 1] == '\\')
   {
      /* 查看是否有续行 */
      strcpy (st, "\0");
      do
      {
         strncat (st, sqlInput, strlen(sqlInput) -1);
         gets (sqlInput);
      } while (sqlInput[strlen(sqlInput) - 1] == '\\');
   strcat (st, sqlInput);
   /* 处理 SQL 语句 */
   rc = process_statement (st);
   }
   else
   {
      strcpy (st, sqlInput);
      /* 处理输入的 SQL 语句*/
      rc = process_statement (st);
   } /* end if */
} /* end while */

printf ("Enter 'c' to COMMIT or Any Other key to ROLLBACK the transaction :\n");

Transaction = getc(stdin);
if (Transaction == 'c')
```

```
    {
       printf("COMMITING the transactions.\n");

       /提交结果*/
       EXEC SQL COMMIT;
       CHECKERR ("COMMIT");
    } else
    {
       /* 撤消语句的执行结果*/
       printf("ROLLING BACK the transactions.\n");

       EXEC SQL ROLLBACK;
       CHECKERR ("ROLLBACK");
    }; /* endif */

    /*断开数据库的连接*/
    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");

    return 0;
}


/*****************************************************************************
* 函数 : process_statement
* This function processes the inputted statement and then prepares the
* procedural SQL implementation to take place.
*****************************************************************************/
int process_statement (char sqlInput[1000])
{
    int counter = 0;
    struct sqlda *sqldaPointer;
    short sqlda_d ; /* Total columns */
    short idx;
    char buffer[4096];
    char varname[1024];
    char colnamelist[4096];

    /*声明一个宿主变量，用于存放 SQL 语句*/
    EXEC SQL BEGIN DECLARE SECTION;
```

```
char st[1000];
EXEC SQL END DECLARE SECTION;

/*向宿主变量中存放 SQL 语句*/
strcpy(st, sqlInput);

/* 分配 SQLDA 空间，以存放查询结果 */
init_da (&sqldaPointer, 1);

EXEC SQL PREPARE statement1 from :st;
if (CHECKERR ("PREPARE") != 0)
   return SQLCODE;

/*获得返回结果的描述信息，填入 SQLDA 结构*/
EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer;

/* 判断 DESCRIBE 是否正确执行*/
if (SQLCODE != 0 &&
   SQLCODE != SQL_RC_W236 &&
   SQLCODE != SQL_RC_W237 &&
   SQLCODE != SQL_RC_W238 &&
   SQLCODE != SQL_RC_W239)
{
   /* An unexpected warning/error has occured. Check the SQLCA. */
   if (CHECKERR ("DESCRIBE") != 0)
      return SQLCODE;
} /* end if */

/*如果 SQLDA 结构中的 sqld 大于 0，则表明是一个 SELECT 语句，sqld 值是列的个数*/
if (sqldaPointer->sqld > 0)
{
   /*判断是否有 LOB 列，若是，则需要双倍的 SQLDA 空间*/
   if (strncmp(SQLSTATE, "01005", sizeof(SQLSTATE)) == 0)
   {
      /* this output contains columns that need a DOUBLED SQLDA */
      SETSQLDOUBLED (sqldaPointer, SQLDOUBLED);
      init_da (&sqldaPointer, sqldaPointer->sqld * 2);
   }
   else
   {
```

```
    /*否则，只需要一个 SQLDA */
    init_da (&sqldaPointer, sqldaPointer->sqld);
} /* end if */

/* 对 SQLDA 重新赋值*/
EXEC SQL DESCRIBE statement1 INTO :*sqldaPointer;
if (CHECKERR ("DESCRIBE") != 0)
    return SQLCODE;

/* 给 SQLDA 分配合适的内存空间*/
alloc_host_vars (sqldaPointer);

/* 声明游标*/
EXEC SQL DECLARE pcurs CURSOR FOR statement1;

/*打开游标*/
EXEC SQL OPEN pcurs;
if (CHECKERR ("OPEN") != 0)
    return SQLCODE;

/*查询一行，存放在 SQLDA 中*/
EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer;
if (CHECKERR ("FETCH") != 0)
    return SQLCODE;

/* 从 SQLDA 中获得列标题信息，并显示之*/
colnamelist[0] = 0;

for ( idx=0; idx< sqlda_d; idx++)
{
    strcat(colnamelist, readColName(sqldaPointer, idx, buffer));
    If (idx < sqlda_d -1)
    strcat(colnamelist, ",");
}
printf("%s\n",colnamelist);

/*显示所有的行数据*/
while ( SQLCODE == 0 )
{
    counter++ ;
```

```
            for ( idx=0; idx< sqlda_d; idx++)
            printf(“%s”,readCol(sqldaPointer, idx, buffer));
            EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer ;
        } /* endwhile */

        /*关闭游标*/
        EXEC SQL CLOSE pcurs;
        if (CHECKERR ("CLOSE CURSOR") != 0) return SQLCODE;

        printf ("\n %d record(s) selected\n\n", counter);

        /* 释放 SQLDA 申请的空间 */
        free_da(sqldaPointer);
    }
    else
    { /*不是 SELECT 语句，则执行 SQL 语句 */
        EXEC SQL EXECUTE statement1;
        if (CHECKERR ("executing the SQL statement") != 0) return SQLCODE;
    } /* end if */

    return SQLCODE;
}
/* end of program : ADHOC.SQC */


/*************************************************************************
* PROCEDURE : init_da
*为 SQLDA 分配空间。其中 SQLDASIZE 的作用，是计算 SQLDA 的大小。返回-1，
*表示无法分配空间。
*************************************************************************/
int init_da (struct sqlda **DAPointer, int DAsqln)
{
    int idx;
    *DAPointer = (struct sqlda *) malloc (SQLDASIZE(DAsqln));
    if (*DAPointer == NULL)
        return (-1);

    memset (*DAPointer, '\0', SQLDASIZE(DAsqln));

    strncpy((*DAPointer)->sqldaid, "SQLDA ", sizeof ((*DAPointer)->sqldaid));
```

```
    (*DAPointer)->sqldabc = (long)SQLDASIZE(DAsqln);
    (*DAPointer)->sqln = DAsqln;
    (*DAPointer)->sqld = 0;

    return 0;
}
/*********************************************************************
* FUNCTION : alloc_host_vars
*为 sqlvar 结构申请空间。返回-1 表示申请失败。
*********************************************************************/
int alloc_host_vars (struct sqlda *sqldaPointer)
{
    short idx;
    unsigned int memsize =0;
    long longmemsize =0;
    int precision =0;
    for (idx = 0; idx < sqldaPointer->sqld; idx++)
    {
       switch (sqldaPointer->sqlvar[idx].sqltype )
       {
          case SQL_TYP_VARCHAR:
          case SQL_TYP_NVARCHAR:
          case SQL_TYP_LONG:
          case SQL_TYP_NLONG:
          case SQL_TYP_DATE:
          case SQL_TYP_NDATE:
          case SQL_TYP_TIME:
          case SQL_TYP_NTIME:
          case SQL_TYP_STAMP:
          case SQL_TYP_NSTAMP:
                 sqldaPointer->sqlvar[idx].sqltype = SQL_TYP_NCSTR;
                 sqldaPointer->sqlvar[idx].sqldata = (char *SQL_POINTER)
                 malloc ((sqldaPointer->sqlvar[idx].sqllen));
                 memsize = (sqldaPointer->sqlvar[idx].sqllen);
             break;
          case SQL_TYP_DECIMAL:
          case SQL_TYP_NDECIMAL:
                 precision = ((char *)&(sqldaPointer->sqlvar[idx].sqllen))[0];
                 sqldaPointer->sqlvar[idx].sqldata = (char *SQL_POINTER)
                 malloc ((precision + 2) /2);
```

```
                memsize = (precision +2) /2;
            break;
        default:
                sqldaPointer->sqlvar[idx].sqldata = (char *SQL_POINTER)
                malloc (sqldaPointer->sqlvar[idx].sqllen);
                memsize = sqldaPointer->sqlvar[idx].sqllen;
            break;
    } /* endswitch */

    if (sqldaPointer->sqlvar[idx].sqldata == NULL)
    {
        return (-1);
    }
    else
    {
        memset (sqldaPointer->sqlvar[idx].sqldata,'\0',memsize);
    } /* endif */

    /*为 sqlind 申请空间*/
    if ( sqldaPointer->sqlvar[idx].sqltype & 1 )
    {
        /* Allocate storage for short int */
        sqldaPointer->sqlvar[idx].sqlind = (short *)malloc(sizeof(short));

        /* Detect memory allocation error */
        if ( sqldaPointer->sqlvar[idx].sqlind == NULL )
        {
            return(-1) ;
        }
        else
        {
            /* initialize memory to zero */
            memset(sqldaPointer->sqlvar[idx].sqldata,'\0',sizeof(short));
        } /* endif */

    } /* endif */

} /* endfor */

return 0;
```

```
}


/******************************************************************************
* FUNCTION : free_da
* 释放 SQLDA 申请的空间。
******************************************************************************/
void free_da (struct sqlda *sqldaPointer)
{
    short idx;

    for (idx = 0; idx < sqldaPointer->sqld; idx++)
    {
       free (sqldaPointer->sqlvar[idx].sqldata);
       if (sqldaPointer->sqlvar[idx].sqltype & 1)
       {
          free (sqldaPointer->sqlvar[idx].sqlind);
       }
    } /* endfor */

    free (sqldaPointer);
}


/******************************************************************************
* PROCEDURE : readColName
* 返回列名信息
******************************************************************************/
char * readColName (struct sqlda *sqldaPointer, short sqlvarIndex, char * buffer)
{
    strncpy(buffer, sqldaPointer->sqlvar[sqlvarIndex].sqlname.data,
    sqldaPointer->sqlvar[sqlvarIndex].sqlname.length);

    return buffer;
}


/******************************************************************************
* PROCEDURE : readCol
* 返回一行数据
```

```
*****************************************************************************/
char * readCol (struct sqlda *sqldaPointer, short sqlvarIndex, char * buffer)
{
    short numBytes;
    short idx, ind ; /* Array idx variables */
    /* Variables for decoding packed decimal data */
    short bottom, point ;
    unsigned short top, precision, scale;
    char tmpstr[1024];
    short pos;
    short collen;
    char *dataptr;

    /* 检查是否为 null */
    if ( sqldaPointer->sqlvar[sqlvarIndex].sqltype & 1 &&\
    *(sqldaPointer->sqlvar[sqlvarIndex].sqlind) < 0 )

    {
       buffer[0] = 0;
       return buffer;
    }
    dataptr = (char *) sqldaPointer->sqlvar[ sqlvarIndex ].sqldata;
    collen = sqldaPointer->sqlvar[ sqlvarIndex ].sqllen;

    switch ( sqldaPointer->sqlvar[ sqlvarIndex ].sqltype )
    {
       case SQL_TYP_INTEGER: /* long */
       case SQL_TYP_NINTEGER: /* long with null indicator */
              sprintf(buffer, "%ld", * ( long *) dataptr ) ;
          break ;

       case SQL_TYP_SMALL: /* short */
       case SQL_TYP_NSMALL: /* short with null indicator */
              sprintf(buffer, "%d", * ( short *) dataptr ) ;
          break ;

       case SQL_TYP_DECIMAL: /* decimal */
       case SQL_TYP_NDECIMAL: /* decimal with null indicator */
              /* Determine the scale and precision */
              precision = ((char *)&(collen))[0];
```

```
            scale = ((char *)&(collen))[1];

    /*****************************************************************************
       *计算精度

    *****************************************************************************/
            if ((precision %2) == 0) precision += 1;
            /* Calculate the total number of bytes */
            idx = ( short ) ( precision + 2 ) / 2 ;
            point = precision - scale ;
            pos = 0;
            /* Determine the sign */
            bottom = *(dataptr + idx -1) & 0x000F ; /* sign */
            if ( (bottom == 0x000D) || (bottom == 0x000B) )
            {
            buffer[pos++]='-';
            }
            /* Decode and print the decimal number */
            for (pos=0, ind=0; ind < idx; ind++)
            {
            top = *(dataptr + ind) & 0x00F0 ;
            top = (top >> 4 ) ;
            bottom = *(dataptr + ind) & 0x000F ;
            if ( point-- == 0 ) buffer[pos++]='.' ;
            buffer[pos++]='0' + top ;

    /*****************************************************************************/
       /*忽略最后一位（符号位） */

    /*****************************************************************************/
            if ( ind < idx - 1 ) { /* sign half byte ? */
            if ( point-- == 0 ) buffer[pos++] = '.';
            buffer[pos++] = '0'+ bottom;
            }
            }
            buffer[pos] = 0;
          break ;

       case SQL_TYP_FLOAT: /* double */
       case SQL_TYP_NFLOAT: /* double with null indicator */
```

```
                sprintf(buffer, "%e", * (double *) dataptr) ;
            break ;

        case SQL_TYP_CHAR: /* fixed length character string */
        case SQL_TYP_NCHAR: /* fixed length character string with nullindicator
*/
                strncpy(buffer, dataptr, collen);
                buffer[collen] = 0;
                collen--;
                while ((collen >=0) && (buffer[collen] == ' '))
                    buffer[collen--] = 0;
            break;

        case SQL_TYP_LSTR: /* varying length character string, 1-byte length */
        case SQL_TYP_NLSTR: /* varying length character string, 1-byte length,
                with null indicator */
                /* Initialize blen to the value the length field in the varchar data
structure. */
                collen = *dataptr;
                /* Advance the data pointer beyond the length field */
                dataptr+=sizeof(char);
                strncpy(buffer, dataptr, collen);
                buffer[collen] = 0;
            break ;

        case SQL_TYP_CSTR: /* null terminated varying length character string */
        case SQL_TYP_NCSTR: /* null terminate varying length character
                    string with null indicator */
                strcpy(buffer, dataptr);
            break ;

        default:
                buffer[0] = 0;
    }

    return buffer;
}
/* COMMENT OUT OFF */
```

1.4 第四节 ORACLE数据库的嵌入SQL语言

1.4.1 基本的SQL语句

1.4.1.1 宿主变量和指示符

1）、声明方法

同其他数据库管理器一样，ORACLE 使用宿主变量传递数据库中的数据和状态信息到应用程序，应用程序也通过宿主变量传递数据到 ORACLE 数据库。根据上面两种功能，宿主变量分为输出宿主变量和输入宿主变量。在 SELECT INTO 和 FETCH 语句之后的宿主变量称作“输出宿主变量”，这是因为从数据库传递列数据到应用程序。除了 SELECT INTO 和 FETCH 语句外的其他 SQL 语句中的宿主变量，称为“输入宿主变量”。这是因为从应用程序向数据库输入值。如：INSERT、UPDATE 等语句。请看下面这个例子：

```
int emp_number;
char temp[20];
VARCHAR emp_name[20];
/* get values for input host variables */
printf("Employee number? ");
gets(temp);
emp_number = atoi(temp);
printf("Employee name? ");
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
    VALUES (:emp_number, :emp_name);
```

在上面这个例子中，其中的 emp_number 和 emp_name 就是宿主变量。值得注意的是，它同其他数据库的区别是，定义宿主变量可以不需要 BEGIN DECLARE SECTION 和 END DECLARE SECTION。

2）、指示符变量

大多数程序设计语言（如 C）都不支持 NULL。所以对 NULL 的处理，一定要在 SQL 中完成。我们可以使用主机指示符变量来解决这个问题。在嵌入式 SQL 语句中，主变量和指示符变量共同规定一个单独的 SQL 类型值。指示符变量是一个 2 字节的整数。

针对输入宿主变量和输出宿主变量，指示变量共有下面几种情况：

同输入宿主变量一起使用时：

-1 Oracle 将 null 赋值给列，即宿主变量应该假设为 NULL。

>=0 Oracle 将宿主变量的实际值赋值给列。

同输出宿主变量一起使用时：

-1 表示该列的输出值为 NULL。

0 Oracle 已经将列的值赋给了宿主变量。列值未做截断。

>0 Oracle 将列的值截断，并赋给了宿主变量。指示变量中存放了这个列的实际长度。

-2 Oracle 将列的值截断，并赋给了宿主变量。但是这个列的实际长度不能确定。

从数据库中查询数据时，可以使用指示符变量来测试 NULL：

```
EXEC SQL SELECT ename, sal
   INTO :emp_name, :salary
   FROM emp
   WHERE :commission INDICATOR :ind_comm IS NULL ...
```

注意，不能使用关系操作符来比较 NULL，这是因为 NULL 和任何操作都为 false。如：

```
EXEC SQL SELECT ename, sal
   INTO :emp_name, :salary
   FROM emp
   WHERE comm = :commission
```

如果 comm 列的某些行存在 NULL，则该 SELECT 语句不能返回正确的结果。应该使用下面这个语句完成：

```
EXEC SQL SELECT ename, sal
INTO :emp_name, :salary
FROM emp
WHERE (comm = :commission) OR ((comm IS NULL) AND
   (:commission INDICATOR :ind_comm IS NULL));
```

1.4.1.2 查询

如果是单行查询，则应该使用 SELECT INTO 语句。如果是多行查询，应该使用游标或宿主变量数组。如：单行查询的一个例子：

```
EXEC SQL SELECT ename, job, sal + 2000
   INTO :emp_name, :job_title, :salary
   FROM emp
   WHERE empno = :emp_number;
```

在嵌入 SQL 语句中，也可以使用子查询。如：

```
EXEC SQL INSERT INTO emp2 (empno, ename, sal, deptno)
   SELECT empno, ename, sal, deptno FROM emp
      WHERE job = :job_title;
```

1.4.1.3 修改数据

1）、插入数据

使用 INSERT 语句插入数据。其语法同 ANSI SQL 语法类似。如：

```
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno)
   VALUES (:emp_number, :emp_name, :salary, :dept_number);
```

2）、更新数据

使用 UPDATE 语句更新数据。其语法同 ANSI SQL 语法类似。如：

```
EXEC SQL UPDATE emp
   SET sal = :salary, comm = :commission
   WHERE empno = :emp_number;
```

3）、删除数据

使用 DELETE 语句删除数据。其语法同 ANSI SQL 语法类似。如：

```
EXEC SQL DELETE FROM emp
WHERE deptno = :dept_number;
```

1.4.1.4 游标

用嵌入式 SQL 语句查询数据分成两类情况。一类是单行结果，一类是多行结果。对于单行结果，可以使用 SELECT INTO 语句；对于多行结果，你必须使用游标来完成。游标是一个与 SELECT 语句相关联的符号名，它使用户可逐行访问由 ORACLE 返回的结果集。使用游标，应该包含以下四个步骤。

1）、定义游标

使用 DECLARE 语句完成。如：

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename FROM emp WHERE deptno = :dept_number;
```

值得注意的是，不能在同一个文件中定义两个相同名字的游标。游标的作用范围是全局的。

2）、打开游标

使用 OPEN 语句完成。如：

```
EXEC SQL OPEN emp_cursor;
```

3）、取一行值

使用 FETCH 语句完成。如：

```
EXEC SQL FETCH emp_cursor INTO :emp_name;
```

4）、关闭游标

使用 CLOSE 语句完成。它完成的功能是：释放资源，如占用内存，锁等。如：EXEC SQL CLOSE emp_cursor;

5）、使用游标修改数据

我们可以使用 CURRENT OF 子句来完成修改数据。如：

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename, sal FROM emp WHERE job = 'CLERK'
   FOR UPDATE OF sal;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;) {
   EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
   ...
   EXEC SQL UPDATE emp SET sal = :new_salary
```

```
        WHERE CURRENT OF emp_cursor;
    }
```

值得注意的是，在使用 CURRENT OF 子句来完成修改数据时，在 OPEN 时会对数据加上排它锁。这个锁直到有 COMMIT 或 ROLLBACK 语句时才释放。

以下是使用游标修改数据的一个完整例子：

```
...
/* 定义游标 */
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job
       FROM emp
       WHERE empno = :emp_number
    FOR UPDATE OF job;

/* 打开游标 */
EXEC SQL OPEN emp_cursor;

/* break if the last row was already fetched */
EXEC SQL WHENEVER NOT FOUND DO break;

/* 循环取值*/
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;

    /* 更新当前游标所在的行的数据*/
    EXEC SQL UPDATE emp
       SET job = :new_job_title
       WHERE CURRENT OF emp_cursor;
}
...
/* 关闭游标 */
EXEC SQL CLOSE emp_cursor;
EXEC SQL COMMIT WORK RELEASE;
...
```

下面这个例子完整演示了静态游标的使用方法。这个例子的作用是，获得部门编号，通过游标来显示这个部门中的所有雇员信息。

```
#include <stdio.h>

/* 声明宿主变量 */
char userid[12] = "SCOTT/TIGER";
char emp_name[10];
int emp_number;
int dept_number;
char temp[32];
void sql_error();

/*包含 SQLCA */
#include <sqlca.h>

main()
{
    emp_number = 7499;

    /* 处理错误*/
    EXEC SQL WHENEVER SQLERROR do sql_error("Oracle error");

    /* 连接到 Oracle 数据库*/
    EXEC SQL CONNECT :userid;
    printf("Connected.\n");

    /* 声明游标 */
    EXEC SQL DECLARE emp_cursor CURSOR FOR
       SELECT ename FROM emp WHERE deptno = :dept_number;

    printf("Department number? ");
    gets(temp);
    dept_number = atoi(temp);

    /* 打开游标*/
    EXEC SQL OPEN emp_cursor;
    printf("Employee Name\n");
    printf("-------------\n");

    /* 循环处理每一行数据，如果无数据，则退出*/
    EXEC SQL WHENEVER NOT FOUND DO break;
```

```
    while (1)
    {
       EXEC SQL FETCH emp_cursor INTO :emp_name;
       printf("%s\n", emp_name);
    }

    EXEC SQL CLOSE emp_cursor;
    EXEC SQL COMMIT WORK RELEASE;

    exit(0);
}


/错误处理程序*/
void sql_error(char *msg)
{
    char buf[500];
    int buflen, msglen;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;

    buflen = sizeof (buf);
    sqlglm(buf, &buflen, &msglen);
    printf("%s\n", msg);
    printf("%*.s\n", msglen, buf);

    exit(1);
}
```

1.4.2 嵌入PL/SQL

嵌入 PL/SQL 和嵌入 SQL 不同。嵌入 PL/SQL 提供了很多嵌入 SQL 不具有的优点，如：更好的性能、更灵活的表达方式。能够自己定义过程和函数。如：

```
PROCEDURE create_dept
(new_dname IN CHAR(14),
new_loc IN CHAR(13),
new_deptno OUT NUMBER(2)) IS
BEGIN
SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);
```

END create_dept;

其中的 IN/OUT，表示参数模式。IN 是传递参数值到过程，而 OUT 是从过程传递参数值到调用者。

但是，如果使用这些扩展的功能，也会造成同其他数据库厂商的嵌入 SQL 的不兼容。

1.4.3 动态SQL语句

1.4.3.1 ORACLE动态SQL语句的一些特点

ORACLE DBMS 进入市场的时间早于 DB2，其动态 SQL 支持是以 IBM 的 system/R 原型为基础的。因此，ORACLE 支持的动态 SQL 与 IBM 的 DB2 标准有不同。虽然 ORACLE 和 DB2 在很大程度上是兼容的，但是在使用参数标志、SQLDA 格式及支持数据类型转换等方面都有差异。

DB2 中不允许在 PREPARE 的动态语句中引用宿主变量,而是用问号来标志语句中的参数，然后用 EXECUTE 或 OPEN 语句来规定参数值。ORACLE 允许用户用宿主变量规定动态语句中的参数。

而且，ORACLE 支持的 DESCRIBE 语句同 DB2 有一些区别。如：

从已经 PREPARE 后的动态查询语句中获得对查询结果列的信息的语句为：

EXEC SQL DESCRIBE SELECT LIST FOR qrystmt INTO qry_sqlda;

等价于 DB2 的：

EXEC SQL DESCRIBE qrystmt INTO qry_sqlda;

从已经 PREPARE 后的动态查询语句中获得对查询参数的说明的语句为：

EXEC SQL DESCRIBE BIND LIST FOR qrystmt INTO qry_sqlda;

该 ORACLE 语句没有对应的 DB2 语句。用户只能按照当前需要的参数和 SQLDA 的结构对 SQLDA 赋值。然后再在 OPEN 语句或 EXECUTE 语句中使用 SQLDA 结构。

1.4.3.2 使用动态SQL的四种方法

使用动态 SQL，共分成四种方法：

方法 支持的 SQL 语句

1 该语句不包含宿主变量，该语句不是查询语句

2 该语句包含输入宿主变量 ，该语句不是查询语句

3 包含已知数目的输入宿主变量或列的查询

4 包含未知数目的输入宿主变量或列的查询

l 方法 1：使用 EXECUTE IMMEDIATE 命令实现，具体语法为：

EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };

其中，host_variable 和 string 是存放完整 T-SQL 语句。

请看下面这个例子。这个例子的作用是执行用户随意输入的合法的 SQL 语句。

```
char dyn_stmt[132];
...
for (;;)
```

```
{
   printf("Enter SQL statement: ");
   gets(dyn_stmt);
   if (*dyn_stmt == '\0')
      break;
   /* dyn_stmt now contains the text of a SQL statement */
   EXEC SQL EXECUTE IMMEDIATE :dyn_stmt;
}
...
```

EXECUTE IMMEDIATE 命令的作用是：分析该语句的语法，然后执行该语句。方法 1 适合于仅仅执行一次的语句。

l 方法 2：方法支持的语句可以包含输入宿主变量。这个语句首先做 PREPARE 操作，然后通过 EXECUTE 执行。PREPARE 语句的语法为：

EXEC SQL PREPARE statement_name FROM { :host_string | string_literal };

该语句接收含有 SQL 语句串的宿主变量，并把该语句送到 ORACLE。ORACLE 编译语句并生成执行计划。在语句串中包含一个“？”表明参数，当执行语句时，ORACLE 需要参数来替代这些“？”。PREPRARE 执行的结果是，DBMS 用语句名标志准备后的语句。在执行 SQL 语句时，EXECUTE 语句后面是这个语句名。EXECUTE 语句的语法为：

EXECUTE 语句名 USING 宿主变量 | DESCRIPTOR 描述符名

它的作用是，请求 ORACLE 执行 PREPARE 语句准备好的语句。当要执行的动态语句中包含一个或多个参数标志时，在 EXECUTE 语句必须为每一个参数提供值。这样的话，EXECUTE 语句用宿主变量值逐一代替准备语句中的参数标志（“？”或其他占位符），从而，为动态执行语句提供了输入值。

使用主变量提供值，USING 子句中的主变量数必须同动态语句中的参数标志数一致，而且每一个主变量的数据类型必须同相应参数所需的数据类型相一致。各主变量也可以有一个伴随主变量的指示符变量。当处理 EXECUTE 语句时，如果指示符变量包含一个负值，就把 NULL 值赋予相应的参数标志。除了使用主变量为参数提供值，也可以通过 SQLDA 提供值。

请看下面这个例子。这个例子的作用是删除用户指定的雇员信息。

```
...
int emp_number INTEGER;
char delete_stmt[120], search_cond[40];;
...
strcpy(delete_stmt, "DELETE FROM EMP WHERE EMPNO = :n AND ");
printf("Complete the following statement's search condition--\n");
printf("%s\n", delete_stmt);
gets(search_cond);
strcat(delete_stmt, search_cond);

EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
for (;;)
{
```

```
    printf("Enter employee number: ");
    gets(temp);
    emp_number = atoi(temp);
    if (emp_number == 0)
        break;
    EXEC SQL EXECUTE sql_stmt USING :emp_number;
}
```

l 方法三：是指查询的列数或输入宿主变量数在预编译时已经确定，但是数据库中的对象，如表、列名等信息未确定。这些对象名不能是宿主变量。这时，必须通过以下语句来完成：

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

如：下面这个例子演示用方法 3 完成动态查询：

```
char select_stmt[132] =
   "SELECT MGR, JOB FROM EMP WHERE SAL < :salary";
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL OPEN emp_cursor USING :salary;
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
EXEC SQL CLOSE emp_cursor;
```

l 方法四：在预编译时，查询的列数或者宿主变量的个数不能确定，因为不知道具体的返回个数，所以不能使用输出宿主变量。这是因为你不知道应该定义多少个宿主变量。这时，就需要 SQLDA 结构和 DESCRIBE 命令。SQLDA 包含了动态查询的列描述信息。对于输入宿主变量，也可以使用 SQLDA 来完成不确定的参数说明。要完成方法四，必须通过以下语句来完成：

```
EXEC SQL PREPARE statement_name FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
   INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
   [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
   INTO select_descriptor_name;
EXEC SQL FETCH cursor_name USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

在上述语句中，DESCRIBE SELECT LIST 的作用是将 PREPARE 后的动态查询语句的列名、数据类型、长度等信息保存在 SQLDA 中。DESCRIBE BIND VARIABLES 的作用是，检查 PREPARE

后的动态查询语句的每个占位符的名字、数据类型、长度等信息。并将它存放在 SQLDA 中，然后，使用 SQLDA 提示用户数据参数值。

值得注意的是，方法之间可以混合使用。如：在一个查询中，列的个数确定，但是查询中的占位符不确定，这时，你可以结合方法 3 和方法 4，即使用方法 3 的 FETCH 语句和方法 4 的 OPEN 语句，如：EXEC SQL FETCH emp_cursor INTO host_variable_list; 反之，如果查询中占位符的个数确定，而列数不确定，则你可以使用方法 3 的 OPEN 语句，如：EXEC SQL OPEN cursor_name [USING host_variable_list];

这里，我们讲解的是嵌入 SQL，对于嵌入 PL/SQL，有一些区别。简单来说，主要有两点：

l 预编译器将 PL/SQL 块中的所有宿主变量都作为输入宿主变量。

l 不能对 PL/SQL 块使用 FETCH 命令。

l 占位符不用声明，可以是任何名字。如：

```
INSERT INTO emp (empno, deptno) VALUES (:e, :d)
DELETE FROM dept WHERE deptno = :num OR loc = :loc
```

其中的 e、d、num 和 loc 就是占位符。

1.4.3.3 SQLDA

SQLDA 存放了输出数据的信息，或存放了输入数据的信息。可以使用 SQLSQLDAAlloc(runtime_context, size, name_length, ind_name_length)来分配空间。

SQLDA 结构的定义存放在 sqlda.h 文件中。它的内容为：

```
struct SQLDA
{
   long N; /* Descriptor size in number of entries */
   char **V; /*Ptr to Arr of addresses of main variables */
   long *L; /* Ptr to Arr of lengths of buffers */
   short *T; /* Ptr to Arr of types of buffers */
   short **I; /* Ptr to Arr of addresses of indicator vars */
   long F; /* Number of variables found by DESCRIBE */
   char **S; /* Ptr to Arr of variable name pointers */
   short *M; /* Ptr to Arr of max lengths of var. names */
   short *C; /* Ptr to Arr of current lengths of var. names */
   char **X; /* Ptr to Arr of ind. var. name pointers */
   short *Y; /* Ptr to Arr of max lengths of ind. var. names */
   short *Z; /* Ptr to Arr of cur lengths of ind. var. names */
};
```

其中，上述变量的含义为：

lN：可以容纳的列的最大数目或参数的最大数目。它对应于 DB2 的 SQLDA 的 SQLN 字段。

lF：当前 SQLDA 中的实际列数或参数个数。它对应于 DB2 的 SQLDA 的 SQLD 字段。

lT：指明数据类型。它对应于 DB2 的 SQLVAR 结构中的 SQLTYPE 字段。

lV：指向字符数组。该字符数组可能是列的数据，或传送参数的数据。它对应于 DB2 的 SQLVAR 结构中的 SQLDATA 字段。

lL：给出列或参数值的长度。它对应于 DB2 的 SQLVAR 结构中的 SQLLEN 字段。

lI：指向指示符变量，标志数据是否为 NULL。它对应于 DB2 的 SQLVAR 结构中的 SQLIND 字段。

lS：指向存放列名或参数名的字符数组。它对应于 DB2 的 SQLVAR 结构中的 SQLNAME 结构的 data[]。

lM：指向一个整数，该整数是 S 的申请长度。在 DB2 中，SQLVAR 结构中的 SQLNAME 结构的 data[30]的大小是固定的，即是 30。而 ORACLE 中是可变的。其大小为 M 指向的整数。

lC：指向一个整数，该整数是 S 的实际长度。它对应于 DB2 的 SQLVAR 结构中的 SQLNAME 结构的 length。

lX：指向一个字符数组。该字符数组存放了指示符变量的名称，指示符变量表示传递的参数是否为 NULL。DB2 中无相应的对应字段。这个缓冲区仅仅供 DESCRIBE BIND LIST 语句使用。

lY：指向一个整数，该整数是 X 的申请的最大长度。DB2 中无相应的对应字段。

lZ：指向一个整数，该整数是 X 的实际长度。DB2 中无相应的对应字段。

ORACLE 的数据类型分成两种情况：内部数据类型和外部数据类型。ORACLE 的内部数据类型是 ORACLE 在数据库中存放数据的类型，在使用 DESCRIBE SELECT LIST 命令，就返回内部数据类型代码。下表是所有的内部数据类型：

Oracle 内部数据类型 代码

VARCHAR2 1

NUMBER 2

LONG 8

ROWID 11

DATE 12

RAW 23

LONG RAW 24

CHARACTER (or CHAR) 96

MLSLABEL 106

外部数据类型是输入宿主变量和输出宿主变量存放数据的类型。DESCRIBE BIND VARIABLES 命令将 SQLDA 中的数据类型代码置为 0。所以，必须在 OPEN 语句前设置外部数据类型代码，以告诉 ORACLE 是什么外部数据类型。下表是具体的外部数据类型：

外部数据类型 代码 C 数据类型

VARCHAR2 1 char[n]

NUMBER 2 char[n] (n 22)

INTEGER 3 int

FLOAT 4 float

STRING 5 char[n+1]

VARNUM 6 char[n] (n 22)

DECIMAL 7 float

LONG 8 char[n]

VARCHAR 9 char[n+2]

ROWID 11 char[n]

DATE 12 char[n]

VARRAW 15 char[n]

RAW 23 unsigned char[n]

LONG RAW 24 unsigned char[n]

UNSIGNED 68 unsigned int

DISPLAY 91 char[n]

LONG VARCHAR 94 char[n+4]

LONG VARRAW 95 unsigned char[n+4]

CHAR 96 char[n]

CHARF 96 char[n]

CHARZ 97 char[n+1]

MLSLABEL 106 char[n]

当 ORACLE 从用户程序中接收参数值并向用户程序传送查询结果时，就在自己的内部数据格式与它所运行的计算机系统的数据格式之间自动进行数据转换。DESCRIBE SELECT L IST 命令可以返回 ORACLE 的内部数据类型。对于字符数据，内部数据类型同外部数据类型是相一致的；而有些内部数据类型对应到外部数据类型后，导致处理复杂化，如：你想将 NUMBER 数据类型的值处理为 C 中的 FLOAT，那么你可以设置相应的 T 值为 FLOAT(4)和 L 值为 FLOAT 的长度。在 FETCH 时，ORACLE 自动在内部数据类型和外部数据类型之间转换。

在 DB2 的 SQLVAR 结构中，列的说明信息、数据等存放在一个单独的 sqlvar 结构中。而在 ORACLE 数据库中，不存在一个单独的结构来说明每列的信息。而是通过数组的方式实现。如下图所示，描述了 1 个输入参数，参数名为 bonus。假设的最大参数个数为 3。

SQLDA 结构

N=3

V

L

T

I

F=1 describe 设置

S

N

C

X

Y

Z

图 6-5 SQLDA 结构示例

下面这个例子是一个 adhoc 程序。用户输入任何合法的 SQL 语句（可以带参数），该程序能够处理这个语句，并打印出结果。这个例子非常经典，说明使用 SQLDA 的两个功能。

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

/* 列的最大数目或宿主变量的最大个数*/
#define MAX_ITEMS 40

/* 列名的最大长度或指示符的最大长度*/
#define MAX_VNAME_LEN 30
#define MAX_INAME_LEN 30

#ifndef NULL
#define NULL 0
#endif

char *dml_commands[] = {"SELECT", "select", "INSERT", "insert",
       "UPDATE", "update", "DELETE", "delete"};

EXEC SQL BEGIN DECLARE SECTION;
char dyn_statement[1024];
EXEC SQL VAR dyn_statement IS STRING(1024);
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE sqlda;

SQLDA *bind_dp;
SQLDA *select_dp;

extern SQLDA *SQLSQLDAAlloc();
extern void sqlnul();

/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;

/* A global flag for the error routine. */
int parse_flag = 0;
```

```
main()
{
    int oracle_connect();
    int alloc_descriptors();
    int get_dyn_statement();
    int set_bind_variables();
    int process_select_list();
    int i;

    /*连接到数据库 */
    if (oracle_connect() != 0)
       exit(1);

    /* 为 SQLDA 分配空间*/
    if (alloc_descriptors(MAX_ITEMS, MAX_VNAME_LEN, MAX_INAME_LEN) != 0)
       exit(1);

    /* 处理 SQL 语句*/
    for (;;)
    {
       i = setjmp(jmp_continue);

       /* 获取 SQL 语句。输入"exit"表示退出 */
       if (get_dyn_statement() != 0)
          break;

       /* 对该 SQL 语句做 PREPARE 操作 */
       EXEC SQL WHENEVER SQLERROR DO sql_error();
       parse_flag = 1; /* Set a flag for sql_error(). */

       EXEC SQL PREPARE S FROM :dyn_statement;

       parse_flag = 0; /* Unset the flag. */

       /*声明游标*/
       EXEC SQL DECLARE C CURSOR FOR S;

       /* 提示用户输入参数值*/
       set_bind_variables();
```

```
        /* 打开游标 */
        EXEC SQL OPEN C USING DESCRIPTOR bind_dp;

        /* 处理语句，并输出结果*/
        process_select_list();

        /*输出处理的行数. */
        for (i = 0; i < 8; i++)
        {
            if (strncmp(dyn_statement, dml_commands[i], 6) == 0)
            {
                printf("\n\n%d row%c processed.\n",
                sqlca.sqlerrd[2],
                sqlca.sqlerrd[2] == 1 ? '\0' : 's');
                break;
            }
        }
    } /* end of for(;;) statement-processing loop */

    /* 释放申请的空间*/
    for (i = 0; i < MAX_ITEMS; i++)
    {
        if (bind_dp->V[i] != (char *) 0)
            free(bind_dp->V[i]);

        free(bind_dp->I[i]); /* MAX_ITEMS were allocated. */

        if (select_dp->V[i] != (char *) 0)
            free(select_dp->V[i]);

        free(select_dp->I[i]); /* MAX_ITEMS were allocated. */
    }

    SQLSQLDAFree(SQL_SINGLE_RCTX, bind_dp);
    SQLSQLDAFree(SQL_SINGLE_RCTX, select_dp);

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* 关闭游标*/
    EXEC SQL CLOSE C;
```

```
    EXEC SQL COMMIT WORK RELEASE;

    puts("\nHave a good day!\n");

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    return;
}


/*连接数据库函数*/
oracle_connect()
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR username[128];
    VARCHAR password[32];
    EXEC SQL END DECLARE SECTION;

    /提示用户输入用户名*/
    printf("\nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
    fflush(stdin);
    username.arr[strlen((char *) username.arr)-1] = '\0';
    username.len = strlen((char *) username.arr);

    /*提示用户输入口令*/
    printf("password: ");
    fgets((char *) password.arr, sizeof password.arr, stdin);
    fflush(stdin);
    password.arr[strlen((char *) password.arr) - 1] = '\0';
    password.len = strlen((char *) password.arr);

    EXEC SQL WHENEVER SQLERROR GOTO connect_error;

    /*连接数据库*/
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user %s.\n", username.arr);

    return 0;
```

```
    /*连接错误处理*/
connect_error:
    fprintf(stderr, "Cannot connect to ORACLE as user %s\n",
    username.arr);

    return -1;
}


/*为 SQLDA 分配空间*/
alloc_descriptors(size, max_vname_len, max_iname_len)
int size;
int max_vname_len;
int max_iname_len;
{
    int i;

    /*SQLSQLDAAlloc 的第一个参数是 SQL 语句的最大列数或输入宿主变量的最大个数。
    *第二个参数，是指列名的最大长度，或参数名的最大长度。
    *第三个参数，是指指示符变量名的最大长度。*/

    /*给 SQLDA 分配空间，下面这个 SQLDA 用于输入参数*/
    if ((bind_dp = SQLSQLDAAlloc(SQL_SINGLE_RCTX, size,
       max_vname_len, max_iname_len)) == (SQLDA *) 0)
    {
       fprintf(stderr, "Cannot allocate memory for bind descriptor.");
       return -1; /* Have to exit in this case. */
    }

    /*给 SQLDA 分配空间，下面这个 SQLDA 用于动态查询*/
    if ((select_dp = SQLSQLDAAlloc (SQL_SINGLE_RCTX, size,
       max_vname_len, max_iname_len)) == (SQLDA *) 0)
    {
       fprintf(stderr,"Cannot allocate memory for select descriptor.");
       return -1;
    }

    /*设置最大的列数，或最大的变量数*/
       select_dp->N = MAX_ITEMS;
```

```
    /* 给存放指示符变量值和存放数据的变量申请空间。*/
    for (i = 0; i < MAX_ITEMS; i++)
    {
       bind_dp->I[i] = (short *) malloc(sizeof (short));
       select_dp->I[i] = (short *) malloc(sizeof(short));
       bind_dp->V[i] = (char *) malloc(1);
       select_dp->V[i] = (char *) malloc(1);
    }

    return 0;
}


/*获得 SQL 语句，可略看*/
get_dyn_statement()
{
    char *cp, linebuf[256];
    int iter, plsql;
    int help();

    for (plsql = 0, iter = 1; ;)
    {
       if (iter == 1)
       {
          printf("\nSQL> ");
          dyn_statement[0] = '\0';
       }

       fgets(linebuf, sizeof linebuf, stdin);
       fflush(stdin);
       cp = strrchr(linebuf, '\n');

       if (cp && cp != linebuf)
          *cp = ' ';
       else if (cp == linebuf)
          continue;

       if ((strncmp(linebuf, "EXIT", 4) == 0) ||
          (strncmp(linebuf, "exit", 4) == 0))
       {
```

```
            return -1;
        }
        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn_statement, ';'))))
        {
            *cp = '\0';
            break;
        }
        else
        {
            iter++;
            printf("%3d ", iter);
        }

    }

    return 0;
}



/*设置宿主变量的信息*/
set_bind_variables()
```

```
{
    int i, n;
    char bind_var[64];

    /* 通过 DESCRIBE 语句，将处理语句的参数名、数据类型等信息存放在 bind_dp 中*/
    EXEC SQL WHENEVER SQLERROR DO sql_error();
    bind_dp->N = MAX_ITEMS; /* Init. count of array elements. */

    EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_dp;
    /* F 小于 0，表示 SQLSQLDAAlloc()分配的空间不够，即实际参数的个数超出预算的最大值. */

    if (bind_dp->F < 0)
    {
       printf("\nToo many bind variables (%d), maximum is %d.\n",
          -bind_dp->F, MAX_ITEMS);
       return;
    }

    /* 将 N（最大值）设置为实际的参数个数*/
    bind_dp->N = bind_dp->F;

    /* 提示用户输入参数值，并设置 SQLDA 的其他相关值，如：长度等。*/
    for (i = 0; i < bind_dp->F; i++)
    {
       printf ("\nEnter value for bind variable %.*s: ",
          (int)bind_dp->C[i], bind_dp->S[i]);

       fgets(bind_var, sizeof bind_var, stdin);

       /* 获得长度，去掉 NULL 结束符 */
       n = strlen(bind_var) - 1;

       /*设置参数长度 */
       bind_dp->L[i] = n;

       /* 分配存放参数数据的内存空间 */
       bind_dp->V[i] = (char *) realloc(bind_dp->V[i], (bind_dp->L[i] + 1));

       /* 将数据放在这个内存空间中 */
```

```
        strncpy(bind_dp->V[i], bind_var, n);

        /* 设置指示符变量的值*/
        if ((strncmp(bind_dp->V[i], "NULL", 4) == 0) ||
              (strncmp(bind_dp->V[i], "null", 4) == 0))
           *bind_dp->I[i] = -1;
        else
           *bind_dp->I[i] = 0;

        /* 设置数据类型为 CHAR，ORACLE 会根据列的数据类型自动转换 */
        bind_dp->T[i] = 1;
    }
}


/*处理语句*/
process_select_list()
{
    int i, null_ok, precision, scale;

    /*如果不是查询语句，则设置 F（即返回的列数）为 0*/
    if ((strncmp(dyn_statement, "SELECT", 6) != 0) &&
       (strncmp(dyn_statement, "select", 6) != 0))
    {
       select_dp->F = 0;
       return;
    }

    /* 如果是 SELECT 语句,则通过 DESCRIBE 函数返回列名、数据类型、长度和是否为 NULL
标志*/
    select_dp->N = MAX_ITEMS;
    EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_dp;

    /* 如果 F 小于 0。则表示比预定的列数要多。*/
    if (select_dp->F < 0)
    {
       printf("\nToo many select-list items (%d), maximum is %d\n",
          -(select_dp->F), MAX_ITEMS);
       return;
    }
```

```
/* 设置最大列数为实际列数*/
select_dp->N = select_dp->F;
/* 为每列分配空间。
SQLNumberPrecV6() 函数的作用是从 select_dp->L[i]获得精度和长度。
SQLColumnNullCheck() 函数的作用是检查该列是否为 NULL。*/

printf ("\n");
for (i = 0; i < select_dp->F; i++)
{
   /* 关闭最高位*/
   SQLColumnNullCheck (&(select_dp->T[i]),
      &(select_dp->T[i]), &null_ok);
   switch (select_dp->T[i])
   {
      case 1 : /* CHAR */
         break;

      case 2 : /* NUMBER , 获得精度和范围*/
         SQLNumberPrecV6 (SQL_SINGLE_RCTX, &(select_dp->L[i]), &precision,
            &scale);
            /* 如果精度为 0，则设置为最大值 40 */
            if (precision == 0)
            precision = 40;
            if (scale > 0)
            select_dp->L[i] = sizeof(float);
            else
            select_dp->L[i] = sizeof(int);
         break;

      case 8 : /* LONG*/
            select_dp->L[i] = 240;
         break;

      case 11 : /* ROWID datatype */
            select_dp->L[i] = 18;
         break;

      case 12 : /* DATE datatype */
            select_dp->L[i] = 9;
```

```
            break;

        case 23 : /* RAW datatype */
            break;

        case 24 : /* LONG RAW datatype */
                select_dp->L[i] = 240;
            break;
    }

    /* 申请空间给 SQLDA 来存放数据*/
    if (select_dp->T[i] != 2)
        select_dp->V[i] = (char *) realloc(select_dp->V[i],
        select_dp->L[i] + 1);
    else
        select_dp->V[i] = (char *) realloc(select_dp->V[i],
        select_dp->L[i]);

    /* 输出列名*/
    if (select_dp->T[i] == 2)
        if (scale > 0)
            printf ("%.*s ",select_dp->L[i]+3, select_dp->S[i]);
        else
            printf ("%.*s ", select_dp->L[i], select_dp->S[i]);
    else
        printf ("%-.*s ", select_dp->L[i], select_dp->S[i]);

    /* 除了 LONG RAW 和 NUMBER，其他数据类型转换为字符型数据类型*/
    if (select_dp->T[i] != 24 && select_dp->T[i] != 2)
        select_dp->T[i] = 1;

    /* 将 NUMBER 数据类型转换为浮点型数据类型或 int 数据类型*/
    if (select_dp->T[i] == 2)
        if (scale > 0)
            select_dp->T[i] = 4; /* float */
        else
            select_dp->T[i] = 3; /* int */
}
printf ("\n\n");
```

```
    /* 取出每一行数据*/
    EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;

    for (;;)
    {
       EXEC SQL FETCH C USING DESCRIPTOR select_dp;

       /*输出列数据。除了 float 和 int 数据类型，其他数据类型都被转为字符型*/
       for (i = 0; i < select_dp->F; i++)
       {
       if (*select_dp->I[i] < 0)
          if (select_dp->T[i] == 4)
             printf ("%-*c ",(int)select_dp->L[i]+3, ' ');
          else
             printf ("%-*c ",(int)select_dp->L[i], ' ');
       else
          if (select_dp->T[i] == 3) /* int datatype */
             printf ("%*d ", (int)select_dp->L[i],
                *(int *)select_dp->V[i]);
          else
             if (select_dp->T[i] == 4)/* float datatype*/
                printf ("%*.2f ", (int)select_dp->L[i],
                *(float *)select_dp->V[i]);
             else /* character string */
                printf ("%-*s ",
                (int)select_dp->L[i], select_dp->V[i]);
       }
       printf ("\n");
    }

    end_select_loop:

    return;
}



help()
{
    puts("\n\nEnter a SQL statement or a PL/SQL block");
```

```
    puts("at the SQL> prompt.");
    puts("Statements can be continued over several");
    puts("lines, except within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
    puts("Terminate a PL/SQL block");
    puts("(which can contain embedded semicolons)");
    puts("with a slash (/).");
    puts("Typing \"exit\" (no semicolon needed)");
    puts("exits the program.");
    puts("You typed \"?\" or \"help\"");
    puts(" to get this message.\n\n");
}


sql_error()
{
    int i;
    /* ORACLE error handler */
    printf ("\n\n%.70s\n",sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
       printf("Parse error at character offset %d.\n",
           sqlca.sqlerrd[4]);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;

    longjmp(jmp_continue, 1);
}
```

SQLDA 中的 sqldata 存放着指向数据的地址。你可以认为,如果查询出的数据是整数(如：258），那么这个地址是指向整数的地址（也就是说，INTEL 是高位在后，低位在前。所以第一个字节为 00000001，第二个字节为 00000010），如果查询出的数据是字符（‘2’），那么这个地址是指向字符的地址(也就是说，第一个字节为 00110010—2 的 ASCII 码)。又因为，sqldata 声明的是指向字符的指针，所以，你必须按照不同的数据类型做转换，即：对于整数，应该是（*（int *）sqldata)，告诉系统，sqldata 目前指向的数据应该按照整数来解释。如果按照字符来解释，那么第一个字符是 ASCII 值为 1 的字符，显然不正确。从数据库向 sqldata 赋值时，是直接赋值。如：*p=*q，*（P+1）=*（Q+1）。如果从数据库查询出数据为 258，则存放在 sqlda 中也是 258，存放格式为：第一个字节为 00000001，第二个字节为 00000010。你可以执行以下语句，来体会上述论述。

```
#include <stdio.h>
main()
{
    int li_i;
    int * lp_int;
    char * lp_char;
    char lc_char;
    li_i=258;
    lp_char=malloc(10);
    lp_int=lp_char;
    *lp_int=258;
    /*(lp_char+2)='\0';*/
    printf("*lp_char=%d\n",*((int *)lp_char));
    printf("*lp_char=%s\n",lp_char);
    return;
}
```

1.5 第五节INFORMIX的嵌入SQL/C语言

1.5.1 一个简单的入门例子

例 1、查询 customer 表中所有 lname 的第一个字符小于 C 的顾客信息。

```
#include <stdio.h>

/*定义两个常量*/
EXEC SQL define FNAME_LEN 15;
EXEC SQL define LNAME_LEN 15;

main()
{
    /*声明宿主变量*/
    EXEC SQL BEGIN DECLARE SECTION;
    char fname[ FNAME_LEN + 1 ];
    char lname[ LNAME_LEN + 1 ];
    EXEC SQL END DECLARE SECTION;

    printf( "DEMO1 Sample ESQL Program running.\n\n");

    /*出错处理，如果返回错误信息，则停止该程序*/
    EXEC SQL WHENEVER ERROR STOP;

    /*连接到 stores7 数据库*/
    EXEC SQL connect to 'stores7';

    /*声明一个游标*/
    EXEC SQL DECLARE democursor cursor for
       select fname, lname
           into :fname, :lname
       from customer
       where lname < 'C';

    /*打开游标*/
    EXEC SQL open democursor;

    /*如果 SQLSTATE 不等于“00”，那么表示到达了数据集的尾部（02），或者产生了错
误（大于 02）*/
    for (;;)
    {
       EXEC SQL fetch democursor;

       if (strncmp(SQLSTATE, "00", 2) != 0)
```

```
            break;
        printf("%s %s\n",fname, lname);
    }

    /*打印错误信息*/
    if (strncmp(SQLSTATE, "02", 2) != 0)
        printf("SQLSTATE after fetch is %s\n", SQLSTATE);

    /*关闭游标*/
    EXEC SQL close democursor;

    /*释放游标占用的资源*/
    EXEC SQL free democursor;

    /*断开数据库服务器的连接*/
    EXEC SQL disconnect current;

    printf("\nDEMO1 Sample Program over.\n\n");
}
```

从上面这个例子，我们看出嵌入 SQL 的基本特点是：

1、每条嵌入式 SQL 语句都用 EXEC SQL 开始，表明它是一条 SQL 语句。这也是告诉预编译器在 EXEC SQL 和“；”之间是嵌入 SQL 语句。

2、如果一条嵌入式 SQL 语句占用多行，在 C 程序中可以用续行符“\”，在 Fortran 中必须有续行符。其他语言也有相应规定。

3、每一条嵌入 SQL 语句都有结束符号，如：在 C 中是“；”。

4、嵌入 SQL 语句的关键字不区分大小写。

5、可以使用“/*….*/”来添加注释。

从上面这个例子看出，INFORMIX 数据库的嵌入 SQL 语句的格式同其他数据库基本相同。但是，它也有它自己本身的一些特点。本节把重点放在 INFORMIX 数据库所独有的一些语句或处理方式。

1.5.2 宿主变量

宿主变量就是在嵌入式 SQL 语句中引用主语言说明的程序变量。如：

```
EXEC SQL connect to :hostvar;
```

1）、定义宿主变量

方法 1：采用 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间给主变量说明。如：

```
EXEC SQL BEGIN DECLARE SECTION;
char fname[ FNAME_LEN + 1 ];
```

```
char lname[ LNAME_LEN + 1 ];
EXEC SQL END DECLARE SECTION;
```

方法 2：在每个变量的数据类型前加上“$”。如：

```
$int hostint;
$double hostdbl;
```

ESQL/C 对宿主变量的大小写敏感。但是，ESQL/C 的关键字、语句标志符、游标名大小写不敏感。在 SQL 语句中，除了使用“:”来标志宿主变量外，还可以使用“$”。当然，“:”是 ANSI 标准。如：EXEC SQL connect to $hostvar。对于注释，可以使用“--”，也可以使用标准的“/*…*/”。

2）、宿主变量和 NULL

方法 1：使用指示符变量。

方法 2：使用函数 risnull()和 rsetnull()。

3）、指示符变量

大多数程序设计语言（如 C）都不支持 NULL。所以对 NULL 的处理，一定要在 SQL 中完成。我们可以使用主机指示符变量来解决这个问题。在嵌入式 SQL 语句中，宿主变量和指示符变量共同规定一个单独的 SQL 类型值。指示变量和前面宿主变量之间用一个空格相分隔。如：

```
EXEC SQL select lname, company
   into :name INDICATOR :nameind, :comp INDICATOR :compind
```

nameind 是 name 变量的指示符，而 compind 是 comp 变量的指示符。

可以通过以下三种方法使用指示符变量：

方法 1、使用 INDICATOR 关键字。

```
:hostvar INDICATOR :indvar
```

方法 2、

```
:hostvar :indvar
```

方法 3、使用$符号。

```
$hostvar $indvar。
```

无论采用哪种方法，都是实现指示符变量的作用。即：当宿主变量 hostvar 应该返回 NULL 时，指示符变量为-1。当宿主变量 hostvar 应该返回不是 NULL 而且无需截断时，指示符变量为 0。当返回值太大而需要截断时，指示符变量是截断前数据的长度。SQLSTATE 会返回 01004 错误信息。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char name[16];
char comp[20];
short nameind;
short compind;
EXEC SQL END DECLARE SECTION;
EXEC SQL select lname, company
   into :name INDICATOR :nameind, :comp INDICATOR :compind
   from customer
   where customer_num = 105;
```

如果对应 105 的 company 为 NULL，则 compind 小于 0，如果 lname 的结果大于 15 个字节，那么 name 包含前 15 个字符。

4）、宿主变量的数据类型

INFROMIX ESQ/C 的宿主变量数据类型除了标准 C 的数据类型外，可以是它自己定义的数据类型。如：

lvarchar 数据类型

```
EXEC SQL BEGIN DECLARE SECTION;
varchar varc_name[n + 1];
EXEC SQL END DECLARE SECTION;
```

lint8 数据类型

```
EXEC SQL BEGIN DECLARE SECTION;
int8 int8_var1;
ifx_int8_t int8_var2;
EXEC SQL BEGIN DECLARE SECTION;
```

lfixchar 数据类型

```
EXEC SQL BEGIN DECLARE SECTION;
boolean flag;
fixchar my_boolflag;
int id;
EXEC SQL END DECLARE SECTION;
```

lDecimal 数据类型

```
#define DECSIZE 16
struct decimal
{
short dec_exp;
short dec_pos;
short dec_ndgts;
char dec_dgts[DECSIZE];
};
typedef struct decimal dec_t;
```

lDatetime 数据类型

```
EXEC SQL include datetime;
EXEC SQL BEGIN DECLARE SECTION;
datetime year; /* will cause an error */
datetime year to day year, today; /* ambiguous */
EXEC SQL END DECLARE SECTION;
```

lInterval hour 等数据类型

```
EXEC SQL BEGIN DECLARE SECTION;
interval day(3) to day accrued_leave, leave_taken;
interval hour to second race_length;
interval scheduled;
```

EXEC SQL END DECLARE SECTION;

l 其他数据类型

EXEC SQL BEGIN DECLARE SECTION;

loc_t my_simple_lo;

EXEC SQL END DECLARE SECTION;

my_simole_lo.loc_loctype = LOCMEMORY;

l 在 INFORMIX 数据库中，'\0'表示为 NULL。如：

id = 1;

flag = '\0'; /* valid boolean assignment to FALSE */

EXEC SQL insert into table2 values (:id, :flag); /* inserts FALSE */

在以 SQL 为基础的 DBMS 支持的数据类型与程序设计语言支持的数据类型之间有很大差别。如果你通过宿主变量从数据库取值，或者通过宿主变量向数据库插入值，都存在数据类型转换的问题。

下表列出了 C 的数据类型、ESQL/C 定义的数据类型和 SQL 数据类型的兼容关系：

SQL 数据类型 ESQL/C 定义的数据类型 C 的数据类型

BOOLEAN boolean

BYTE loc_t

CHAR(n)

CHARACTER(n) fixchar [n] 或 string [n+1] char [n + 1] 或 char *

DATE date 4-byte integer

DATETIME datetime 或 ordtime_t

DECIMAL decimal 或 dec_t

DEC decimal 或 dec_t

NUMERIC decimal 或 dec_t

MONEY decimal 或 dec_t

FLOAT double

DOUBLE double

PRECISION double

INT8 int8 或 ifx_int8_t

INTEGER 4-byte integer

INT 4-byte integer

INTERVAL interval or intrvl_t

LVARCHAR lvarchar char [n + 1] orchar *

NCHAR(n) fixchar [n] orstring [n+1] char [n + 1] orchar *

NVARCHAR(m) varchar[m+1] orstring [m+1] char [m+1]

SERIAL 4-byte integer

SERIAL8 int8 or ifx_int8_t

SMALLFLOAT float

REAL float

SMALLINT 2-byte integer

TEXT loc_t

VARCHAR(m,x) varchar[m+1] or string [m+1] char d[m+1]

BLOB ifx_lo_t

CLOB ifx_lo_t

LIST(e) collection

MULTISET(e) collection

Opaque data type lvarchar,fixed binary 或 var binary

ROW(...) row

SET(e) collection

下表是 INFORMIX 数据库服务器支持的数据类型和类型代码：

SQL 数据类型	类型代码	类型代码值
CHAR	SQLCHAR	0
SMALLINT	SQLSMINT	1
INTEGER	SQLINT	2
FLOAT	SQLFLOAT	3
SMALLFLOAT	SQLSMFLOAT	4
DECIMAL	SQLDECIMAL	5
SERIAL	SQLSERIAL	6
DATE	SQLDATE	7
MONEY	SQLMONEY	8
DATETIME	SQLDTIME	10
BYTE	SQLBYTES	11
TEXT	SQLTEXT	12
VARCHAR	SQLVCHAR	13
INTERVAL	SQLINTERVAL	14
NCHAR	SQLNCHAR	15
NVARCHAR	SQLNVCHAR	16
INT8	SQLINT8	17
SERIAL8	SQLSERIAL8	18
LVARCHAR	SQLLVARCHAR	43
BOOLEAN	SQLBOOL	45
SET	SQLSET	19
MULTISET	SQLMULTISET	20
LIST	SQLLIST	21
ROW	SQLROW	22
Varying-length opaque Type	SQLUDTVAR	40
Fixed-length opaque type	SQLUDTFIXED	41
SENDRECV(client-side only)	SQLSENDRECV	44

下表是 ESQL/C 定义的数据类型和类型代码，这些定义存放在各个头文件中。

ESQL/C 数据类型	类型代码	类型代码值
char	CCHARTYPE	100
short int	CSHORTTYPE	101
int4	CINTTYPE	102
long	CLONGTYPE	103
float	CFLOATTYPE	104
double	CDOUBLETYPE	105
dec_t 或 decimal	CDECIMALTYPE	107
fixchar	CFIXCHARTYPE	108
string	CSTRINGTYPE	109
date	CDATETYPE	110
dec_t 或 decimal	CMONEYTYPE	111
datetime 或 dtime_t	CDTIMETYPE	112
loc_t	CLOCATORTYPE	113
varchar	CVCHARTYPE	114
intrvl_t 或 interval	CINVTYPE	115
char	CFILETYPE	116
int8	CINT8TYPE	117
collection(Universal Data Option)	CCOLTYPE	118
lvarchar	CLVCHARTYPE	119
fixed binary	CFIXBINTYPE	120
var binary(Universal Data Option)	CVARBINTYPE	121
boolean	CBOOLTYPE	122
row(Universal Data Option)	CROWTYPE	123

INFORMIX 的 ESQL/C 提供了很多函数来处理数据类型，这些函数的参数就是 ESQL/C 定义的数据类型。如：dectoasc()的作用是转换数据类型是 decimal 的值为 ASCII。

1.5.3 嵌入SQL的处理过程

INFORMIX的预编译器为esql。嵌入SQL包含一些组件：嵌入SQL的库文件，提供访问数据库服务器、操作各种数据类型、出错信息的处理等函数。嵌入SQL的头文件（UNIX环境：$INFORMIXDIR/incl/esql下，WINDOWS环境：%INFORMIXDIR%\incl\esql下），提供程序用的数据结构、常数和宏的定义信息。Esql是预编译器。UNIX系统下，是finderr程序获得INFORMIX的错误信息，WINDOWS平台下是find error获得错误信息。还有一些GLS locale文件，提供

一些特定的locale信息。在WINDOWS平台下，还有另外一些文件，如：setnet32、ilogin、regcopy、esqlmf程序。

创建嵌入 SQL/C 的程序的一般步骤：程序的后缀可以是.ec 或.ecp。

1、定义宿主变量。

2、访问数据库。

3、操作。

4、完成后，使用 esql 命令来预编译。如：esql demo1.ec。在预编译后，程序中只有 C 语言语句，它们都可以为 C 语言的编译器所识别。所以，可以按照一般的方法进行编译和连接，但在将 SQL 语句转换以后，在 C 语言程序中，又引入了许多一般的 C 语言系统所没有的结构、变量和函数，因此应该设置 INCLUDE 和 LIB 的设置。最后生成的可执行文件。

1.5.4 动态SQL语言

所谓静态 SQL 的编程方法，就是指在预编译时 SQL 语句已经基本确定，即访问的表或视图名、访问的列等信息已经确定。但是，有时整个 SQL 语句要到执行的时候才能确定下来，而且 SQL 语句所访问的对象也要到执行时才能确定。这就需要通过动态 SQL 语句完成。动态 SQL 语句的处理步骤是：

1、组合 SQL 语句。

2、PREPARE。PREPARE 语句是动态 SQL 语句独有的语句。其语法为：

PREPARE 语句名 FROM 宿主变量|字符串

该语句接收含有 SQL 语句串的宿主变量，并把该语句送到 DBMS。DBMS 编译语句并生成执行计划。在语句串中包含一个“？”表明参数，当执行语句时，DBMS 需要参数来替代这些“？”。PREPRARE 执行的结果是，DBMS 用语句名标志编译后的语句。在执行 SQL 语句时，EXECUTE 语句后面是这个语句名。请看下面这个例子：

```
EXEC SQL prepare slct_id from
'select company from customer where customer_num = ?';
```

可以通过 SQLCA 检查 PREPARE 操作是否成功。

3、EXECUTE 或 OPEN。

EXECUTE 语句的语法如下：

EXECUTE 语句名 USING 宿主变量 | DESCRIPTOR 描述符名

它的作用是，请求 DBMS 执行 PREPARE 语句准备好的语句。当要执行的动态语句中包含一个或多个参数标志时，在 EXECUTE 语句必须为每一个参数提供值。这样的话，EXECUTE 语句用宿主变量值逐一代替准备语句中的参数标志（“？”），从而，为动态执行语句提供了输入值。

如果是多行查询，则使用游标，使用 OPEN USING 语句传递参数；如果是单行查询，则使用 SELECT INTO。如果是修改数据：则使用 EXECUTE USING 语句。如果知道参数个数，就可以使用宿主变量。如果不知道参数个数，则必须使用 DESCRIBE 语句。下表总结了动态 SQL 语句的处理方法：

语句类型是否有输入参数执行的方法

INSERT、DELETE、UPDATE 没有 EXECUTE

INSERT、DELETE、UPDATE 有（数据类型和个数确定）EXECUTE …USING

INSERT、DELETE、UPDATE 有（数据类型和个数不确定）EXECUTE...USINGSQL DESCRIPTOR 或 EXECUTE...USINGDESCRIPTOR

SELECT（返回多行）无 OPEN

SELECT（返回多行）有（数据类型和个数确定）OPEN…USING

SELECT（返回多行）有（数据类型和个数不确定）OPEN...USINGSQL DESCRIPTOR 或 OPEN...USINGDESCRIPTOR

SELECT（返回一行）无 EXECUTE...INTO

SELECT（返回一行，但是返回的数据类型和个数不确定）无 EXECUTE...INTODESCRIPTOR 或 EXECUTE...INTOSQL DESCRIPTOR

SELECT（返回一行）有 EXECUTE...INTO...USING

SELECT（返回一行，但是返回的数据类型和个数不确定）有 EXECUTE...INTO...USING SQLDESCRIPTOR 或 EXECUTE...INTO...USINGDESCRIPTOR

4、释放资源。

1.5.4.1 SQLDA

可以通过 SQLDA 为嵌入 SQL 语句提供输入数据和从嵌入 SQ 语句中输出数据。理解 SQLDA 的结构是理解动态 SQL 的关键。

我们知道，动态 SQL 语句在编译时可能不知道有多少列信息。在嵌入 SQL 语句中，这些不确定的数据是通过 SQLDA 完成的。SQLDA 的结构非常灵活，在该结构的固定部分，指明了多少列等信息（如下图中的 sqld=2，表示为两列信息），在该结构的后面，有一个可变长的结构（SQLVAR 结构），说明每列的信息。

SQLDA 结构

Sqld=2

sqlvar

Desc_name

Desc_occ

Desc_next

Sqltype=500

Sqllen

sqldata

…..

Sqltype=501

Sqllen

Sqldata

…..

图 6-6 SQLDA 结构示例

具体 SQLDA 的结构在 sqlda.h 中定义，是：

```
struct sqlvar_struct
{
    short sqltype;/* variable type*/
    short sqllen;/* length in bytes*/
    char *sqldata;/* pointer to data*/
    short *sqlind;/* pointer to indicator*/
    char *sqlname;/* variable name*/
    char *sqlformat;/* reserved for future use */
    short sqlitype;/* ind variable type*/
    short sqlilen;/* ind length in bytes*/
    char *sqlidata;/* ind data pointer*/
};
struct sqlda
{
    short sqld;
    struct sqlvar_struct *sqlvar;
    char desc_name[19];/* descriptor name */
    short desc_occ;/* size of sqlda structure */
    struct sqlda *desc_next;/* pointer to next sqlda struct */
};
#endif /* _SQLDA */
```

从上面这个定义看出,SQLDA 是一种由三个不同部分组成的可变长数据结构。位于 SQLDA 开端的 sqldaid 用于标志该 SQLDA 描述了多少列的信息；而后是一个或多个 sqlvar 结构 ，用于标志列数据。当用 SQLDA 把参数送到执行语句时，每一个参数都是一个 sqlvar 结构；当用 SQLDA 返回输出列信息时，每一列都是一个 sqlvar 结构。第三部分是 SQLDA 结构的描述信息部分。具体每个元素的含义为：

lSqld　　目前使用的 sqlvar 结构的个数。即输出列的个数。

lSqlvar　　指向 sqlvar_struct 结构。 即指向描述第一列信息的 sqlvar 结构。

lDesc_name　Sqlda 的名称。

lDesc_occ　Sqlda 结构的大小。

lDesc_next　指向下一个 SQLDA 结构。

lSqltype　　代表参数或列的数据类型。它是一个整数数据类型代码。具体每个整数的含义见第二节。

l Sqllen　　代表传送数据的长度。如：2，即代表二字节整数。如果是字符串，则该数据为字符串中的字符数量。

lSqldata　　指向数据的地址。注意，仅仅是一个地址。

lSqlind　　代表是否为 NULL。如果该列不允许为 NULL，则该字段不赋值；如果该列允许为 NULL，则：该字段若为 0，表示数据值不为 NULL，若为-1，表示数据值为 NULL。

lSqlname　　代表列名或变量名。它是一个结构。

包含 length 和 data。Length 是名字的长度；data 是名字。

lSqlformat　保留为以后使用。

lSqlitype　指定用户定义的指示符变量的数据类型。

lSqlilen　指定用户定义的指示符变量的长度。

lSqlidata　指向用户定义的指示符变量所存放的数据。

下面这个 ADHOC 程序非常经典，演示了 SQLDA 的作用。

模拟一个不确定的查询，然后通过 SQLDA 来获得数据，并打印出来。

```
EXEC SQL include locator.h;
EXEC SQL include sqltypes.h;

#define BLOBSIZE 32276;

main()
{
    int i = 0;
    int row_count;

    /**** Step 1: 声明一个 SQLDA 结构，来存放查询的数据 ********/
    struct sqlda *da_ptr;

    /*连接到数据库服务器*/
    EXEC SQL connect to 'stores7';
    if ( SQLCODE < 0 )
    {
       printf("CONNECT failed: %d\n", SQLCODE)
       exit(0);
    }

    /* 创建一个临时表，模拟一个不确定列和表的环境*/
    EXEC SQL create table blob_tab (int_col integer, blob_col byte);

    /* load_db 函数是往 blob_tab 表插入数据，读者不用关心它的代码*/
    load_db();

    /* PREPARE 查询语句 */
    EXEC SQL prepare selct_id 'select * from tab1';

    /* Step 2: 使用 describe 函数完成两个功能：一是为 sqlda 分配空间， 二是获取语
句信息，并存放在 SQLDA 结构中。*/
    EXEC SQL describe selct_id into da_ptr;

    /* Step 3: 初试化 sqlda 结构，如：为列分配空间，改变数据类型等。*/
    row_size = init_sqlda(da_ptr, 0);

    /* 为 PREPARE 的 SELECT 语句声明和打开游标*/
    EXEC SQL declare curs for selct_id;
```

```
    EXEC SQL open curs;

    while (1)
    {
       /* Step 4: 执行 fetch 操作，将一行数据存放在 sqlda 结构中*/
       EXEC SQL fetch curs using descriptor da_ptr;

       /* 是否到达最后一行？，若是，则退出。 */
       if ( SQLCODE == SQLNOTFOUND )
          break;

       /* Step 5: 从 SQLDA 中打印数据，使用 sqlca.sqlerrd[2]来获得查询的行数*/
       printf("\n===============\n");
       printf("FETCH %d\n", i++);
       printf("===============");
       print_sqlda(da_ptr, ((FetArrSize == 0) ? 1 : sqlca.sqlerrd[2]));
       /* Step 6: 循环执行 FETCH，直到处理完所有的行(SQLCODE 为 SQLNOTFOUND)*/
    }

    /* Step 7: 释放申请的内存空间，如游标、SQLDA、创建的临时表等*/
    EXEC SQL free selct_id;
    EXEC SQL close curs;
    EXEC SQL free curs;

    free_sqlda(da_ptr);

    cleanup_db();
}


/*****************************************************************************
* 函数: init_sqlda()
* 作用: 为 SQLDA 申请空间
* 返回值: 0 正确，否则有错误
*****************************************************************************/

int init_sqlda(in_da, print)
struct sqlda *in_da;
int print;
{
```

```
    int i, j, row_size=0, msglen=0, num_to_alloc;
    struct sqlvar_struct *col_ptr;
    loc_t *temp_loc;
    char *type;

    if (print)
       printf("columns: %d. ", in_da->sqld);

    /* Step 1: 获得一行数据的长度 */
    for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++, col_ptr++)

    /* msglen 变量存放查询数据的所有列的长度和。*/
    msglen += col_ptr->sqllen; /* get database sizes */

    /* 为 col_ptr->sqllen 重新赋值，该值是在 C 下的大小。如：在数据库中的字符串，
在 C 中应该多一个字节空间来存放 NULL 的结束符。*/
    col_ptr->sqllen = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);

    /*row_size 变量存放了在 C 程序中的所有列的长度和。这个值是应用程序为存放一行
数据所需要申请的内存空间*/
    row_size += col_ptr->sqllen;

    if (print) printf("Total row size = %d\n", row_size);

    /* Step 2: 设置 FetArrSize 值*/
    if (FetArrSize == -1) /* if FetArrSize not yet initialized */
    {
       if (FetBufSize == 0) /* if FetBufSize not set */
          FetBufSize = 4096; /* default FetBufSize */

       FetArrSize = FetBufSize/msglen;
    }
    num_to_alloc = (FetArrSize == 0)? 1: FetArrSize;

    /* 设置 sqlvar_struct 结构中的数据类型为相应的 C 的数据类型*/
    for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++, col_ptr++)
    {
       switch(col_ptr->sqltype)
       {
          case SQLCHAR:
```

```
            type = "char ";
            col_ptr->sqltype = CCHARTYPE;
        break;

    case SQLINT:
        type = "int ";
        col_ptr->sqltype = CINTTYPE;
      break;

    case SQLBYTES:
    case SQLTEXT:
        if (col_ptr->sqltype == SQLBYTES)
            type = "blob ";
        else
            type = "text ";
        col_ptr->sqltype = CLOCATORTYPE;

        /* Step 3 ：只有数据类型为 TEXT 和 BLOB 时，才执行。
        为存放 TEXT 或 BYTE 列数据申请空间*/
        temp_loc = (loc_t *)malloc(col_ptr->sqllen * num_to_alloc);
        if (!temp_loc)
        {
            fprintf(stderr, "blob sqldata malloc failed\n");
            return(-1);
        }

        col_ptr->sqldata = (char *)temp_loc;

        /* Step 4：只有数据类型为 TEXT 和 BLOB 时才执行。初试化 loc_t 结构*/
        byfill(temp_loc, col_ptr->sqllen*num_to_alloc ,0);
        for (j = 0; j< num_to_alloc; j++, temp_loc++)
        {
            temp_loc->loc_loctype = LOCMEMORY;
            temp_loc->loc_bufsize = BLOBSIZE;
            temp_loc->loc_buffer = (char *)malloc(BLOBSIZE);
            if (!temp_loc->loc_buffer)
            {
            fprintf(stderr, "loc_buffer malloc failed\n");
            return(-1);
            }
```

```
                temp_loc->loc_oflags = 0; /* clear flag */
            } /* end for */
        break;

    default: /* 其他数据类型*/
            fprintf(stderr, "not yet handled(%d)!\n", col_ptr->sqltype);
            return(-1);

    } /* switch */

    /* Step 5: 为指示符变量申请空间*/
    col_ptr->sqlind = (short *) malloc(sizeof(short) * num_to_alloc);
    if (!col_ptr->sqlind)
    {
        printf("indicator malloc failed\n");
        return -1;
    }

    /* Step 6 ： 为存放非 TEXT 和 BLOB 的数据类型的 sqldata 申请空间.注意的是，
申请的地址是(char *)，在输出数据时，要按照相应的数据类型做转换。*/
    if (col_ptr->sqltype != CLOCATORTYPE)
    {
        col_ptr->sqldata = (char *) malloc(col_ptr->sqllen * num_to_alloc);
        if (!col_ptr->sqldata)
        {
            printf("sqldata malloc failed\n");
            return -1;
        }
    }

    if (print)
        printf("column %3d, type = %s(%3d), len=%d\n", i+1, type,
            col_ptr->sqltype, col_ptr->sqllen);

  } /* end for */

  return msglen;
}
```

```
/*****************************************************************************
* 函数: print_sqlda
* 作用: 打印存放在 SQLDA 结构中的数据。
*****************************************************************************/
void print_sqlda(sqlda, count)
struct sqlda *sqlda;
int count;
{
    void *data;
    int i, j;
    loc_t *temp_loc;
    struct sqlvar_struct *col_ptr;
    char *type;
    char buffer[512];
    int ind;
    char i1, i2;

    /* 打印列数 (sqld) 和行数*/
    printf("\nsqld: %d, fetch-array elements: %d.\n", sqlda->sqld, count);

    /* 外循环：针对每一行数据循环处理 */
    for (j = 0; j < count; j ++)
    {
       if (count > 1)
       {
          printf("record[%4d]:\n", j);
          printf("col | type | id | len | ind | rin | data ");
          printf("| value\n");
          printf("-----------------------------------------");
          printf("-----------------\n");
       }
       /* 内循环: 针对每一列数据处理*/
       for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld;i++, col_ptr++)
       {
          data = col_ptr->sqldata + (j*col_ptr->sqllen);
          switch (col_ptr->sqltype)
          {
             case CFIXCHARTYPE:
             case CCHARTYPE:
```

```
                type = "char";
                if (col_ptr->sqllen > 40)
                sprintf(buffer, " %39.39s<..", data);
                else
                sprintf(buffer, "%*.*s", col_ptr->sqllen,
                col_ptr->sqllen, data);
                break;

            case CINTTYPE:
                type = "int";
                sprintf(buffer, " %d", *(int *) data);
                break;

            case CLOCATORTYPE:
                type = "byte";
                temp_loc = (loc_t *)(col_ptr->sqldata + (j * sizeof(loc_t)));
                sprintf(buffer, " buf ptr: %p, buf sz: %d,
                blob sz: %d", temp_loc->loc_buffer,
                temp_loc->loc_bufsize, temp_loc->loc_size);
                break;

            default:
                type = "??????";
                sprintf(buffer, " type not implemented: ",
                "can't print %d", col_ptr->sqltype);
                break;
        } /* end switch */

        i1 = (col_ptr->sqlind==NULL) ? 'X' :
               (((col_ptr->sqlind)[j] != 0) ? 'T' : 'F');
        i2 = (risnull(col_ptr->sqltype, data)) ? 'T' : 'F';

        printf("%3d | %-6.6s | %3d | %3d | %c | %c | ",
            i, type, col_ptr->sqltype, col_ptr->sqllen, i1, i2);

        printf("%8p |%s\n", data, buffer);
      } /* end for (i=0...) */
    } /* end for (j=0...) */
}
```

```
/*****************************************************************************
* 函数: free_sqlda
* 作用: 释放以下对象申请的内存空间
* o loc_buffer memory (used by TEXT & BYTE)
* o sqldata memory
* o sqlda structure
*****************************************************************************/
void free_sqlda(sqlda)
struct sqlda *sqlda;
{
    int i,j, num_to_dealloc;
    struct sqlvar_struc *col_ptr;
    loc_t *temp_loc;

    for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++, col_ptr++)
    {
       if ( col_ptr->sqltype = CLOCATORTYPE )
       {
          /* Free memory for blob buffer of each element in fetch array */
          num_to_dealloc = (FetArrSize == 0)? 1: FetArrSize;
          temp_loc = (loc_t *) col_ptr->sqldata;

          for (j = 0; j< num_to_dealloc; j++, temp_loc++)
             free(temp_loc->loc_buffer);
       }

       /* Free memory for sqldata (contains fetch array) */
       free(col_ptr->sqldata);
    }

    /* Free memory for sqlda structure */
    free(sqlda);
}
```

1.6 第六节Microsoft SQL Server7 嵌入式SQL语言

1.6.1 一个嵌入SQL语言的简单例子

我们首先来看一个简单的嵌入式 SQL 语言的程序（C 语言）：在 YANGZH 服务器的 pubs 数据库上查询 lastname 为“White”的 firstname。用 sa（口令为 password）连接数据库服务器。这个例子程序如下：

例 1、查询 lastname 为“White”的 firstname 的信息。

```
main()
{
   EXEC SQL BEGIN DECLARE SECTION;
   char first_name[50];
   char last_name[] = "White";
   EXEC SQL END DECLARE SECTION;

   EXEC SQL CONNECT TO YANGZH.pubs
   USER sa.password;
   EXEC SQL SELECT au_fname INTO :first_name
      from authors where au_lname = :last_name;
   printf("first name: %s\n",first_name);
   return (0);
}
```

从上面这个例子，我们看出嵌入 SQL 的基本特点是：

1、每条嵌入式 SQL 语句都用 EXEC SQL 开始，表明它是一条 SQL 语句。这也是告诉预编译器在 EXEC SQL 和“；”之间是嵌入 SQL 语句。

2、如果一条嵌入式 SQL 语句占用多行，在 C 程序中可以用续行符“\”，在 Fortran 中必须有续行符。其他语言也有相应规定。

3、每一条嵌入 SQL 语句都有结束符号，如：在 C 中是“；”。

如果你需要在 NT 平台编写 C 的嵌入 SQL 语句，那么你必须保证设置了以下环境：

lWindows NT Workstation 3.51 或以上；或者Windows NT Server 3.51 或以上

lVC++版本 2.0 或以上；或者其他 100%兼容的编译器和连接器。

lSQL Server6.0 或以上

如果你需要在Windows95/98 平台编写C的嵌入SQL语句，那么你必须保证设置了以下环境：

lWindows 95/98

lVC++版本 2.0 或以上；或者其他 100%兼容的编译器和连接器。

lSQL Server6.0 或以上

1.6.2 嵌入SQL的处理过程

下面我们来说明整个嵌入 SQL 的处理过程。以下是生成可执行文件的步骤：

1、在命令提示符下执行 …\VC98\bin\vcvar32.bat，作用是设置 C 的环境信息。

2、在命令提示符下执行：nmake -f demo.mk 。执行后，生成 demo.exe。

其中，demo.mk 为（在 NT 平台）：

```
demo.exe:demo.sqc
set include=e:\mssql7\devtools\include;%include%;
nsqlprep demo.sqc
cl -o demo.exe e:\mssql7\devtools\lib\sqlakw32.lib \
e:\mssql7\devtools\lib\caw32.lib demo.c
```

其中，

l“set include=e:\mssql7\devtools\include;%include%;”的作用是，说明整个头文件的路径信息。即包含 sqlca.h 和 sqlda.h 路径信息。在嵌入 SQL 程序中，无需使用“#include <sqlca.h>”和“#include <sqlda.h>”语句，这是因为 nsqlprep.exe 预编译器会自动将这些语句插入预编译后的 C 程序中。

l“nsqlprep demo.sqc”是 SQL Server7 的预编译处理。Nsqlprep.exe 是 SQL Server7 的预编译器。处理的结果产生 C 的程序，如 demo.c。demo.c 的程序为：

例 2、demo.c 程序

```
/* ===== demo.c =====*/

/* ===== NT doesn't need the following... */
#ifndef WIN32
#define WIN32
#endif
#define _loadds
#define _SQLPREP_
#include <sqlca.h>
#include <sqlda.h>
#include <string.h>
#define SQLLENMAX(x) ( ((x) > 32767) ? 32767 : (x) )
short ESQLAPI _loadds sqlaaloc(
unsigned short usSqlDaId,
unsigned short sqld,
unsigned short stmt_id,
void far *spare);
short ESQLAPI _loadds sqlxcall(
unsigned short usCallType,
unsigned short usSection,
unsigned short usSqldaInId,
```

```
unsigned short usSqlDaOutId,
unsigned short usSqlTextLen,
char far *lpszSQLText);
short ESQLAPI _loadds sqlacall(
unsigned short usCallType,
unsigned short usSection,
unsigned short usSqldaInId,
unsigned short usSqlDaOutId,
void far *spare);
short ESQLAPI _loadds sqladloc(
unsigned short usSqldaInId,
void far *spare);
short ESQLAPI _loadds sqlasets(
unsigned short cbSqlText,
void far *lpvSqlText,
void far *spare);
short ESQLAPI _loadds sqlasetv(
unsigned short usSqldaInId,
unsigned short sqlvar_index,
unsigned short sqltype,
unsigned short sqllen,
void far *sqldata,
void far *sqlind,
void far *spare);
short ESQLAPI _loadds sqlastop(
void far *spare);
short ESQLAPI _loadds sqlastrt(
void far *pid,
void far *spare,
void far *sqlca);
short ESQLAPI _loadds sqlausda(
unsigned short sqldaId,
void far *lpvSqlDa,
void far *spare);
extern struct tag_sqlca far sql_sqlca;
extern struct tag_sqlca far *sqlca;
struct sqla_program_id2 {
unsigned short length;
unsigned short rp_rel_num;
unsigned short db_rel_num;
unsigned short bf_rel_num;
```

```
    unsigned char sqluser[30];
    unsigned char sqlusername[30];
    unsigned char planname[256];
    unsigned char contoken[8];
    unsigned char buffer[8];
    };
    static struct sqla_program_id2 program_id =
    {340,2,0,0," ","","demo","VVVLKcBo"," "};
    static void far* pid = &program_id;
    #line 1 "demo.sqc"
    main()
    {

    #line 5
    /*
    EXEC SQL BEGIN DECLARE SECTION;
    */
    #line 5
    char first_name[50];
    char last_name[] = "White";

    #line 8
    /*
    EXEC SQL END DECLARE SECTION;
    */
    #line 8



    #line 10
    /*
    EXEC SQL CONNECT TO YANGZH.pubs
    USER sa.password;
    */
    #line 11
    #line 10
    {
    #line 10
    sqlastrt((void far *)pid, (void far *)0, (struct tag_sqlca far *)sqlca);
    #line 10
    sqlxcall(30, 1, 0, 0, 42, (char far *)" CONNECT TO YANGZH.pubs USER sa.password
");
```

```
    #line 10
    SQLCODE = sqlca->sqlcode;
    #line 10
    sqlastop((void far *)0L);
    #line 10
    }
    #line 12

    #line 12
    /*
    EXEC SQL SELECT au_fname INTO :first_name
    from authors where au_lname = :last_name;
    */
    #line 13
    #line 12
    {
    #line 12
    sqlastrt((void far *)pid, (void far *)0, (struct tag_sqlca far *)sqlca);
    #line 12
    sqlaaloc(1, 1, 2, (void far *)0);
    #line 12
    sqlasetv(1, 0, 462,(short) SQLLENMAX(sizeof(first_name)),(void far
*)&first_name, (void far *)0,0L);
    #line 12
    sqlaaloc(2, 1, 2, (void far *)0);
    #line 12
    sqlasetv(2, 0, 462, (short) SQLLENMAX(sizeof(last_name)), (void far
*)last_name, (void far *)0, (void far *)0L);
    #line 12
    sqlxcall(24, 2, 2, 1, 60, (char far *)" SELECT au_fname from authors where
au_lname = @p1 ");
    #line 12
    SQLCODE = sqlca->sqlcode;
    #line 12
    sqlastop((void far *)0L);
    #line 12
    }
    #line 14
    printf("first name: %s\n",first_name);
    return (0);
    }
```

long SQLCODE;

从这个程序看出，预编译器的处理方法是，注释了嵌入的 SQL 语句，用一些特定的函数代替，如 sqlxcall。这些函数调用 sqlakw32.dll，而 sqlakw32.dll 调用了 DB-Library（ntwdblib.dll）来访问 SQL Server 服务器。所以，必须保证应用程序能够访问到 sqlakw32.dll、ntwdblib.dll 和 dbnmpntw.dll。预编译器 nsqlprep.exe 有很多选项，具体这些选项信息可参见帮助。

l “cl -o demo.exe e:\mssql7\devtools\lib\sqlakw32.lib \

e:\mssql7\devtools\lib\caw32.lib demo.c”的作用是 C 源程序的编译和链接。cl 是编译和链接命令的集成命令，编译的结果是产生 demo.obj，在链接时，将 C 的系统库和 SQL Server 提供的库文件（sqlakw32.lib 和 caw32.lib）同目标文件连接在一起。最后生成 demo.exe。也可以使用“SET LIB=e:\mssql7\devtools\LIB;%LIB%”语句设置库文件的环境信息。

设置 SQL Server 相关的头文件和库文件环境信息，也可以执行 mssql7\devtools\samples\esqlc\setenv.bat 程序完成。

在运行 demo.exe 程序时，对于每个 SQL 语句，都调用相应的运行中的服务（如：sqlakw32.dll）。如果该语句是静态 SQL 语句，那么该服务执行 SQL 语句或执行一个已编译成功的存储过程（可以在编译时使用/SQLACCESS 选项为静态 SQL 语句创建存储过程）；如果该语句是动态 SQL 语句，那么该服务将 SQL 语句送到 SQL Server 上处理。这个服务调用 DB-Library，在客户和服务器之间传递数据。这些数据要么存放在主变量中，要么存放在 SQLDA 结构中。执行 SQL 语句的错误信息存放在 SQLCA 数据结构中。

下图总结了整个处理过程。

6-7 嵌入 SQL 程序处理过程

一个应用中的静态 SQL 语句，可以在运行时才发送到服务器端处理（类似动态 SQL 语句），或者生成执行计划（access plan）。一个执行计划就是一些存储过程。每个静态 SQL 语句可以生成一个存储过程。在预编译时，可以创建执行计划。如果在预编译时，服务器不可访问，那么预编译器创建绑定文件（bind file）。绑定文件就是用来创建执行计划中的存储过程的一些 Transact-SQL 脚本。在运行应用程序前，你可以通过 OSQL 执行绑定文件。上面这个例子，在预编译时，未指定“/DB”和“/PASS”选项（用于生成执行计划）也未指定“/BIND”选项（用于生成绑定文件），所以我们生成的应用程序对 SQL 语句的处理是采用类似动态 SQL 语句的处理方式，即在运行时才将语句送到服务器端处理。

Nsqlprep.exe 编译器的作用是，找出 SQL 语句，语法分析这些语句，创建执行计划或绑定文件，最终生成 C 程序。

当然，以上步骤的完整，也可以在 VC++（版本 2.0 以上）集成环境中完成。

1.6.3 嵌入SQL语句

下表是所有的嵌入式 SQL 语句，“*”表示嵌入式 SQL 语句的名字同 Transact-SQL 语句相同。

BEGIN DECLARE SECTION PREPARE

CLOSE* SELECT INTO*

CONNECT TO SET ANSI_DEFAULTS

DECLARE CURSOR* SET CONCURRENCY

DELETE (POSITIONED)* SET CONNECTION

DELETE (SEARCHED)* SET CURSOR_CLOSE_ON_COMMIT

DESCRIBE SET CURSORTYPE

DISCONNECT SET FETCHBUFFER

END DECLARE SECTION SET OPTION

EXECUTE* SET SCROLLOPTION

EXECUTE IMMEDIATE UPDATE (POSITIONED)*

FETCH* UPDATE (SEARCHED)*

GET CONNECTION WHENEVER

OPEN*

嵌入式 SQL 语句分为静态 SQL 语句和动态 SQL 语句两类。下面我们按照功能讲解这些语句。本节讲解静态 SQL 语句的作用。动态 SQL 语句将在下一节讲解。同动态 SQL 相关的一些语句也在下一节中讲解。

6.3.1 声明嵌入 SQL 语句中使用的 C 变量

1）、声明方法

主变量（host variable）就是在嵌入式 SQL 语句中引用主语言说明的程序变量（如例 1 中的 last_name[]变量）。如：

```
EXEC SQL BEGIN DECLARE SECTION;
char first_name[50];
char last_name[] = "White";
EXEC SQL END DECLARE SECTION;
……
EXEC SQL SELECT au_fname INTO :first_name
from authors where au_lname = :last_name;
…….
```

在嵌入式 SQL 语句中使用主变量前，必须采用 BEGIN DECLARE SECTION 和 END DECLARE SECTION 之间给主变量说明。这两条语句不是可执行语句，而是预编译程序的说明。主变量是标准的 C 程序变量。嵌入 SQL 语句使用主变量来输入数据和输出数据。C 程序和嵌入 SQL 语句都可以访问主变量。

值得注意的是，主变量的长度不能超过 30 字节。

2）、主变量的数据类型

在以 SQL 为基础的 DBMS 支持的数据类型与程序设计语言支持的数据类型之间有很大差别。这些差别对主变量影响很大。一方面，主变量是一个用程序设计语言的数据类型说明并用程序设计语言处理的程序变量；另一方面，在嵌入 SQL 语句中用主变量保存数据库数据。所以，在嵌入 SQL 语句中，必须映射 C 数据类型为合适的 SQL Server 数据类型。必须慎重选择主变量的数据类型。在 SQL SERVER 中，很多数据类型都能够自动转换。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int hostvar1 = 39;
char *hostvar2 = "telescope";
```

```
float hostvar3 = 355.95;
EXEC SQL END DECLARE SECTION;
EXEC SQL UPDATE inventory
SET department = :hostvar1
WHERE part_num = "4572-3";
EXEC SQL UPDATE inventory
SET prod_descrip = :hostvar2
WHERE part_num = "4572-3";
EXEC SQL UPDATE inventory
SET price = :hostvar3
WHERE part_num = "4572-3";
```

在第一个 update 语句中，department 列为 smallint 数据类型（integer ），所以应该把 hostvar1 定义为 int 数据类型（integer）。这样的话，从 C 到 SQL Server 的 hostvar1 可以直接映射。在第二个 update 语句中，prod_descip 列为 varchar 数据类型，所以应该把 hostvar2 定义为字符数组。这样的话，从 C 到 SQL Server 的 hostvar2 可以从字符数组映射为 varchar 数据类型。在第三个 update 语句中，price 列为 money 数据类型。在 C 语言中，没有相应的数据类型，所以用户可以把 hostvar3 定义为 C 的浮点变量或字符数据类型。SQL Server 可以自动将浮点变量转换为 money 数据类型（输入数据），或将 money 数据类型转换为浮点变量（输出数据）。

注意的是，如果数据类型为字符数组，那么 SQL Server 会在数据后面填充空格，直到填满该变量的声明长度。

在 ESQL/C 中，不支持所有的 unicode 数据类型（如：nvarchar、nchar 和 ntext）。对于非 unicode 数据类型，除了 datetime、smalldatetime、money 和 smallmoney 外（decimal 和 numeric 数据类型部分情况下不支持），都可以相互转换。

下表列出了 C 的数据类型和 datetime、smalldatetime、money、smallmoney、decimal 和 numeric 数据类型的一些转换关系：

C 数据类型分配的 SQL Server 数据类型 Datetime 或 smalldatetimeMoney 或 smallmoneyDecimal 或 numeric

shortSmallint 不可以不可以不可以

IntSmallint 不可以不可以不可以

LongInt 不可以不可以不可以

FloatReal 不可以不可以不可以

DoubleFloat 不可以不可以不可以

CharCarchar[X]可以可以可以

Void *pBinary(2)可以可以可以

Char bytetinyint 不可以不可以不可以

因为 C 没有 date 或 time 数据类型，所以 SQL Server 的 date 或 time 列将被转换为字符。缺省情况下，使用以下转换格式：mm dd yyyy hh:mm:ss[am | pm]。你也可以使用字符数据格式将 C 的字符数据存放到 SQL Server 的 date 列上。你也可以使用 Transact-SQL 中的 convert 语句来转换数据类型。如：SELECT CONVERT(char, date, 8) FROM sales。

3）、主变量和 NULL

大多数程序设计语言（如 C）都不支持 NULL。所以对 NULL 的处理，一定要在 SQL 中完成。我们可以使用主机指示符变量（host indicator variable）来解决这个问题。在嵌入式 SQL 语句中，主变量和指示符变量共同规定一个单独的 SQL 类型值。如：

```
EXEC SQL SELECT price INTO :price:price_nullflag FROM titles
WHERE au_id = "mc3026"
```

其中，price 是主变量，price_nullflag 是指示符变量。指示符变量共有两类值：

l-1。表示主变量应该假设为 NULL。（注意：主变量的实际值是一个无关值，不予考虑）。

l>0。表示主变量包含了有效值。该指示变量存放了该主变量数据的最大长度。

所以，上面这个例子的含义是：如果不存在 mc3026 写的书，那么 price_nullflag 为-1，表示 price 为 NULL；如果存在，则 price 为实际的价格。

下面我们再看一个 update 的例子：

```
EXEC SQL UPDATE closeoutsale
SET temp_price = :saleprice :saleprice_null, listprice = :oldprice;
```

如果 saleprice_null 是-1，则上述语句等价为：

```
EXEC SQL UPDATE closeoutsale
SET temp_price = null, listprice = :oldprice;
```

我们也可以在指示符变量前面加上“INDICATOR”关键字，表示后面的变量为指示符变量。如：

```
EXEC SQL UPDATE closeoutsale
SET temp_price = :saleprice INDICATOR :saleprice_null;
```

值得注意的是，不能在 WHERE 语句后面使用指示符变量。如：

```
EXEC SQL DELETE FROM closeoutsale
WHERE temp_price = :saleprice :saleprice_null;
```

你可以使用下面语句来完成上述功能：

```
if (saleprice_null == -1)
{
EXEC SQL DELETE FROM closeoutsale
WHERE temp_price IS null;
}
else
{
EXEC SQL DELETE FROM closeoutsale
WHERE temp_price = :saleprice;
}
```

为了便于识别主变量，当嵌入式 SQL 语句中出现主变量时，必须在变量名称前标上冒号（：）。冒号的作用是，告诉预编译器，这是个主变量而不是表名或列名。

6.3.2 连接数据库

在程序中，使用“CONNECT TO”语句来连接数据库。该语句的完整语法为：

```
CONNECT TO {[server_name.]database_name} [AS connection_name] USER
[login[.password] | $integrated]
```

其中，

lserver_name 为服务器名。如省略，则为本地服务器名。

ldatabase_name 为数据库名。

lconnection_name 为连接名。可省略。如果你仅仅使用一个连接，那么无需指定连接名。可以使用 SET CONNECTION 来使用不同的连接。

llogin 为登录名。

lpassword 为密码。

在上例中的“ EXEC SQL CONNECT TO YANGZH.pubs USER sa.password; ”，服务器是 YANGZH，数据库为 pubs，登录名为 sa，密码为 password。缺省的超时时间为 10 秒。如果指定连接的服务器没有响应这个连接请求，或者连接超时，那么系统会返回错误信息。我们可以使用“SET OPTION”命令设置连接超时的时间值。

在嵌入 SQL 语句中，使用 DISCONNECT 语句断开数据库的连接。其语法为：

DISCONNECT [connection_name | ALL | CURRENT]

其中，connection_name 为连接名。ALL 表示断开所有的连接。CURRENT 表示断开当前连接。请看下面这些例子来理解 CONNECT 和 DISCONNECT 语句。

```
EXEC SQL CONNECT TO caffe.pubs AS caffe1 USER sa;
EXEC SQL CONNECT TO latte.pubs AS latte1 USER sa;
EXEC SQL SET CONNECTION caffe1;
EXEC SQL SELECT name FROM sysobjects INTO :name;
EXEC SQL SET CONNECTION latte1;
EXEC SQL SELECT name FROM sysobjects INTO :name;
EXEC SQL DISCONNECT caffe1;
EXEC SQL DISCONNECT latte1;
```

在上面这个例子中，第一个 select 语句查询在 caffe 服务器上的 pubs 数据库。第二个 SELECT 语句查询在 latte 服务器上的 pubs 数据库。当然，你也可以使用“EXEC SQL DISCONNECT ALL; ”来断开所有的连接。

6.3.3 数据的查询和修改

可以使用 SELECT INTO 语句查询数据，并将数据存放在主变量中。如上例中的：

```
EXEC SQL SELECT au_fname INTO :first_name
from authors where au_lname = :last_name;
```

使用 DELETE 语句删除数据。其语法类似于 Transact-SQL 中的 DELETE 语法。如：

```
EXEC SQL DELETE FROM authors WHERE au_lname = 'White'
```

使用 UPDATE 语句可以更新数据。其语法就是 Transact-SQL 中的 UPDATE 语法。如：

```
`EXEC SQL UPDATE authors SET au_fname = 'Fred' WHERE au_lname = 'White'
```

使用 INSERT 语句可以插入新数据。其语法就是 Transact-SQL 中的 INSERT 语法。如：

```
EXEC SQL INSERT INTO homesales (seller_name, sale_price)
real_estate('Jane Doe', 180000.00);
```

多行数据的查询和修改请参见下一节——游标。

6.3.4 游标的使用

用嵌入式 SQL 语句查询数据分成两类情况。一类是单行结果，一类是多行结果。对于单行结果，可以使用 SELECT INTO 语句；对于多行结果，你必须使用 cursor（游标）来完成。游标(Cursor)是一个与 SELECT 语句相关联的符号名，它使用户可逐行访问由 SQL Server

返回的结果集。先请看下面这个例子，这个例子的作用是逐行打印 staff 表的 id、name、dept、 job、years、salary 和 comm 的值。

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT id, name, dept, job, years, salary, comm FROM staff;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
/* SQLCODE will be zero if data is successfully fetched */
EXEC SQL FETCH c1 INTO :id, :name, :dept, :job, :years, :salary, :comm;
if (SQLCODE == 0)
printf("%4d %12s %10d %10s %2d %8d %8d",
id, name, dept, job, years, salary, comm);
}
EXEC SQL CLOSE c1;
```

从上例看出，你首先应该定义游标结果集，即定义该游标的 SELECT 语句返回的行的集合。然后，使用 FETCH 语句逐行处理。

值得注意的是，嵌入 SQL 语句中的游标定义选项同 Transact-SQL 中的游标定义选项有些不同。必须遵循嵌入 SQL 语句中的游标定义选项。

1）、声明游标：

如：EXEC SQL DECLARE C1 CURSOR FOR

SELECT id, name, dept, job, years, salary, comm FROM staff;

2）、打开游标

如：EXEC SQL OPEN c1;

完整语法为：OPEN 游标名 [USING 主变量名 | DESCRIPTOR 描述名]。关于动态 OPEN 游标的描述见第四节。

3）、取一行值

如：EXEC SQL FETCH c1 INTO :id, :name, :dept, :job, :years, :salary, :comm;

关于动态 FETCH 语句见第四节。

4）、关闭游标

如：EXEC SQL CLOSE c1;

关闭游标的同时，会释放由游标添加的锁和放弃未处理的数据。在关闭游标前，该游标必须已经声明和打开。另外，程序终止时，系统会自动关闭所有打开的游标。

也可以使用 UPDATE 语句和 DELETE 语句来更新或删除由游标选择的当前行。使用 DELETE 语句删除当前游标所在的行数据的具体语法如下：

DELETE [FROM] {table_name | view_name} WHERE CURRENT OF cursor_name

其中，

ltable_name 是表名，该表必须是 DECLARE CURSOR 中 SELECT 语句中的表。

lview_name 是视图名，该视图必须是 DECLARE CURSOR 中 SELECT 语句中的视图。

lcursor_name 是游标名。

请看下面这个例子，逐行显示 firstname 和 lastname，询问用户是否删除该信息，如果回答“是”，那么删除当前行的数据。

```
EXEC SQL DECLARE c1 CURSOR FOR
SELECT au_fname, au_lname FROM authors FOR BROWSE;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
EXEC SQL FETCH c1 INTO :fname, :lname;
if (SQLCODE == 0)
{
printf("%12s %12s\n", fname, lname);
printf("Delete? ");
scanf("%c", &reply);
if (reply == 'y')
{
EXEC SQL DELETE FROM authors WHERE CURRENT OF c1;
printf("delete sqlcode= %d\n", SQLCODE(ca));
}
}
}
```

6.3.5 SQLCA

DBMS 是通过 SQLCA（SQL 通信区）向应用程序报告运行错误信息（见 3.4 中的例子）。SQLCA 是一个含有错误变量和状态指示符的数据结构。通过检查 SQLCA，应用程序能够检查出嵌入式 SQL 语句是否成功，并根据成功与否决定是否继续往下执行。预编译器自动在嵌入 SQL 语句中包含 SQLCA 数据结构（见第二节的例子 demo.c）。在程序中可以使用 EXEC SQL INCLUDE SQLCA，目的是告诉 SQL 预编译程序在该程序中包含一个 SQL 通信区。也可以不写，系统会自动加上 SQLCA 结构。

1）、SQLCODE

SQLCA 结构中最重要的部分是 SQLCODE 变量。在执行每条嵌入式 SQL 语句时，DBMS 在 SQLCA 中设置变量 SQLCODE 值，以指明语句的完成状态：

2、0 该语句成功执行，无任何错误或报警。

2、<0 出现了严重错误。

3、>0 出现了报警信息。

2）、SQLSTATE

SQLSTATE 变量也是 SQLCA 结构中的成员。它同 SQLCODE 一样，都是返回错误信息。SQLSTATE 是在 SQLCODE 之后产生的。这是因为，在制定 SQL2 标准之前，各个数据库厂商都采用 SQLCODE 变量来报告嵌入式 SQL 语句中的错误状态。但是，各个厂商没有采用标准的错误描述信息和错误值来报告相同的错误状态。所以，标准化组织增加了 SQLSTATE 变量，规定了通过 SQLSTATE 变量报告错误状态和各个错误代码。因此，目前使用 SQLCODE 的程序仍然有效，但也可用标准的 SQLSTATE 错误代码编写新程序。

6.3.6 WHENEVER

在每条嵌入式 SQL 语句之后立即编写一条检查 SQLCODE/SQLSTATE 值的程序，是一件很繁琐的事情。为了简化错误处理，可以使用 WHENEVER 语句。该语句是 SQL 预编译程序的指示语句，而不是可执行语句。它通知预编译程序在每条可执行嵌入式 SQL 语句之后自动生成错误处理程序，并指定了错误处理操作。

用户可以使用 WHENEVER 语句通知预编译程序去如何处理三种异常处理：

lWHENEVER SQLERROR action：表示一旦 sql 语句执行时遇到错误信息，则执行 action，action 中包含了处理错误的代码（SQLCODE<0）。

lWHENEVER SQLWARNING action:表示一旦 sql 语句执行时遇到警告信息,则执行 aciton，即 action 中包含了处理警报的代码（SQLCODE=1）。

lWHENEVER NOT FOUND：表示一旦 sql 语句执行时没有找到相应的元组，则执行 action，即 action 包含了处理没有查到内容的代码（SQLCODE=100）。

针对上述三种异常处理，用户可以指定预编译程序采取以下三种行为（action）：

lWHENEVER …GOTO：通知预编译程序产生一条转移语句。

lWHENEVER…CONTINUE：通知预编译程序让程序的控制流转入到下一个主语言语句。

lWHENEVER…CALL：通知预编译程序调用函数。

其完整语法如下：

WHENEVER {SQLWARNING | SQLERROR | NOT FOUND} {CONTINUE | GOTO stmt_label | CALL function()}

例：WHENEVER 的作用

```
EXEC SQL WHENEVER sqlerror GOTO errormessage1;
EXEC SQL DELETE FROM homesales
WHERE equity < 10000;
EXEC SQL DELETE FROM customerlist
WHERE salary < 40000;
EXEC SQL WHENEVER sqlerror CONTINUE;
EXEC SQL UPDATE homesales
SET equity = equity - loanvalue;
EXEC SQL WHENEVER sqlerror GOTO errormessage2;
EXEC SQL INSERT INTO homesales (seller_name, sale_price)
real_estate('Jane Doe', 180000.00);
.
.
.
errormessage1:
printf("SQL DELETE error: %ld\n, sqlcode);
exit();
errormessage2:
printf("SQL INSERT error: %ld\n, sqlcode);
exit();
```

WHENEVER 语句是预编译程序的指示语句。在上面这个例子中，由于第一个 WHENEVER 语句的作用，前面两个 DELETE 语句中任一语句内的一个错误会在 errormessage1 中形成一个转移指令。由于一个 WHENEVER 语句替代前面 WHENEVER 语句，所以，嵌入式 UPDATE 语句中的一个错误会直接转入下一个程序语句中。嵌入式 INSERT 语句中的一个错误会在 errormessage2 中产生一条转移指定。

从上面例子看出，WHENEVER/CONTINUE 语句的主要作用是取消先前的 WHENEVER 语句的作用。WHENEVER 语句使得对嵌入式 SQL 错误的处理更加简便。应该在应用程序中普遍使用，而不是直接检查 SQLCODE 的值。

1.6.4 动态SQL语句

前一节中讲述的嵌入SQL语言都是静态SQL语言,即在编译时已经确定了引用的表和列。主变量不改变表和列信息。在上几节中，我们使用主变量改变查询参数，但是不能用主变量代替表名或列名。否则，系统报错。动态 SQL 语句就是来解决这个问题。

动态 SQL 语句的目的是，不是在编译时确定 SQL 的表和列，而是让程序在运行时提供，并将 SQL 语句文本传给 DBMS 执行。静态 SQL 语句在编译时已经生成执行计划。而动态 SQL 语句，只有在执行时才产生执行计划。动态 SQL 语句首先执行 PREPARE 语句要求 DBMS 分析、确认和优化语句，并为其生成执行计划。DBMS 还设置 SQLCODE 以表明语句中发现的错误。当程序执行完“PREPARE”语句后，就可以用 EXECUTE 语句执行执行计划，并设置 SQLCODE，以表明完成状态。

按照功能和处理上的划分，动态 SQL 应该分成两类来解释：动态修改和动态查询。动态修改的例子参见 4.1。动态查询的例子参见 4.3。

6.4 .1 动态修改

动态修改使用 PREPARE 语句和 EXECUTE 语句。PREPARE 语句是动态 SQL 语句独有的语句。其语法为：

PREPARE 语句名 FROM 主变量

该语句接收含有 SQL 语句串的主变量，并把该语句送到 DBMS。DBMS 编译语句并生成执行计划。在语句串中包含一个“？”表明参数，当执行语句时，DBMS 需要参数来替代这些“？”。PREPRARE 执行的结果是，DBMS 把语句名赋给准备的语句。语句名类似于游标名，是一个 SQL 标识符。在执行 SQL 语句时，EXECUTE 语句后面是这个语句名。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char prep[] = "INSERT INTO mf_table VALUES(?,?,?)";
char name[30];
char car[30];
double num;
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE prep_stat FROM :prep;
while (SQLCODE == 0)
{
```

```
strcpy(name, "Elaine");
strcpy(car, "Lamborghini");
num = 4.9;
EXEC SQL EXECUTE prep_stat USING :name, :car, :num;
}
```

在这个例子中，prep_stat 是语句名，prep 主变量的值是一个 INSERT 语句，包含了三个参数（3 个“？”）。PREPARE 的作用是，DBMS 编译这个语句并生成执行计划，并把语句名赋给这个准备的语句。

值得注意的是，PREPARE 中的语句名的作用范围为整个程序，所以不允许在同一个程序中使用相同的语句名在多个 PREPARE 语句中。

EXECUTE 语句是动态 SQL 独有的语句。它的语法如下：

EXECUTE 语句名 USING 主变量 | DESCRIPTOR 描述符名

请看上面这个例子中的“EXEC SQL EXECUTE prep_stat USING :name, :car, :num;”语句，它的作用是，请求 DBMS 执行 PREPARE 语句准备好的语句。当要执行的动态语句中包含一个或多个参数标志时，在 EXECUTE 语句必须为每一个参数提供值，如：:name、:car 和:num。这样的话，EXECUTE 语句用主变量值逐一代替准备语句中的参数标志（“？”），从而，为动态执行语句提供了输入值。

使用主变量提供值，USING 子句中的主变量数必须同动态语句中的参数标志数一致，而且每一个主变量的数据类型必须同相应参数所需的数据类型相一致。各主变量也可以有一个伴随主变量的指示符变量。当处理 EXECUTE 语句时，如果指示符变量包含一个负值，就把 NULL 值赋予相应的参数标志。除了使用主变量为参数提供值，也可以通过 SQLDA 提供值（见节 4.4）。

6.4.2 动态游标

游标分为静态游标和动态游标两类。对于静态游标，在定义游标时就已经确定了完整的 SELECT 语句。在 SELECT 语句中可以包含主变量来接收输入值。当执行游标的 OPEN 语句时，主变量的值被放入 SELECT 语句。在 OPEN 语句中，不用指定主变量，因为在 DECLARE CURSOR 语句中已经放置了主变量。请看下面静态游标的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char szLastName[] = "White";
char szFirstName[30];
EXEC SQL END DECLARE SECTION;
EXEC SQL
DECLARE author_cursor CURSOR FOR
SELECT au_fname FROM authors WHERE au_lname = :szLastName;
EXEC SQL OPEN author_cursor;
EXEC SQL FETCH author_cursor INTO :szFirstName;
```

动态游标和静态游标不同。以下是动态游标使用的句法（请参照本小节后面的例子来理解动态游标）。

1）、声明游标：

对于动态游标，在 DECLARE CURSOR 语句中不包含 SELECT 语句。而是，定义了在 PREPARE 中的语句名，用 PREPARE 语句规定与查询相关的语句名称。

2）、打开游标

完整语法为：OPEN 游标名 [USING 主变量名 | DESCRIPTOR 描述名]

在动态游标中，OPEN 语句的作用是使 DBMS 在第一行查询结果前开始执行查询并定位相关的游标。当 OPEN 语句成功执行完毕后，游标处于打开状态，并为 FETCH 语句做准备。OPEN 语句执行一条由 PREPARE 语句预编译的语句。如果动态查询正文中包含有一个或多个参数标志时，OPEN 语句必须为这些参数提供参数值。USING 子句的作用是规定参数值。

3）、取一行值

FETCH 语法为：FETCH 游标名 USING DESCRIPTOR 描述符名。

动态 FETCH 语句的作用是，把这一行的各列值送到 SQLDA 中，并把游标移到下一行。（注意，静态 FETCH 语句的作用是用主变量表接收查询到的列值。）

在使用 FETCH 语句前，必须为数据区分配空间，SQLDATA 字段指向检索出的数据区。SQLLEN 字段是 SQLDATA 指向的数据区的长度。SQLIND 字段指出是否为 NULL。关于 SQLDA，见下一节。

4）、关闭游标

如：EXEC SQL CLOSE c1;

关闭游标的同时，会释放由游标添加的锁和放弃未处理的数据。在关闭游标前，该游标必须已经声明和打开。另外，程序终止时，系统会自动关闭所有打开的游标。

在动态游标的 DECLARE CURSOR 语句中不包含 SELECT 语句。而是，定义了在 PREPARE 中的语句名，用 PREPARE 语句规定与查询相关的语句名称。当 PREPARE 语句中的语句包含了参数，那么在 OPEN 语句中必须指定提供参数值的主变量或 SQLDA。动态 DECLARE CURSOR 语句是 SQL 预编译程序中的一个命令，而不是可执行语句。该子句必须在 OPEN、FETCH、CLOSE 语句之前使用。请看下面这个例子：

```
EXEC SQL BEGIN DECLARE SECTION;
char szCommand[] = "SELECT au_fname FROM authors WHERE au_lname = ?";
char szLastName[] = "White";
char szFirstName[30];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE author_cursor CURSOR FOR select_statement;
EXEC SQL PREPARE select_statement FROM :szCommand;
EXEC SQL OPEN author_cursor USING :szLastName;
EXEC SQL FETCH author_cursor INTO :szFirstName;
```

一个很实际的例子在 4.4 讲解。

6.4.3 SQLDA

可以通过 SQLDA 为嵌入 SQL 语句提供输入数据和从嵌入 SQ 语句中输出数据。理解 SQLDA 的结构是理解动态 SQL 的关键。

我们知道，动态 SQL 语句在编译时可能不知道有多少列信息。在嵌入 SQL 语句中，这些不确定的数据是通过 SQLDA 完成的。SQLDA 的结构非常灵活，在该结构的固定部分，指明了多少列等信息（如下图中的 sqld=2，表示为两列信息），在该结构的后面，有一个可变长的结构（SQLVAR 结构），说明每列的信息。

SQLDA 结构

Sqld=2

sqlvar

……

Sqltype=500

Sqllen

sqldata

…..

Sqltype=501

Sqllen

Sqldata

…..

图 6-8 SQLDA 结构示例

具体 SQLDA 的结构在 sqlda.h 中定义，是：

```
struct sqlda
{
unsigned char sqldaid[8]; // Eye catcher = 'SQLDA '
long sqldabc; // SQLDA size in bytes = 16+44*SQLN
short sqln; // Number of SQLVAR elements
short sqld; // Num of used SQLVAR elements
struct sqlvar
{
short sqltype; // Variable data type
short sqllen; // Variable data length
// Maximum amount of data < 32K
unsigned char FAR *sqldata; // Pointer to variable data value
short FAR *sqlind; // Pointer to null indicator
struct sqlname // Variable name
{
short length; // Name length [1..30]
unsigned char data[30]; // Variable or column name
} sqlname;
} sqlvar[1];
};
```

从上面这个定义看出，SQLDA 是一种由两个不同部分组成的可变长数据结构。从位于 SQLDA 开端的 sqldaid 到 dqld 为固定部分，用于标志该 SQLDA，并规定这一特定的 SQLDA 的长度。而后是一个或多个 sqlvar 结构 ，用于标志列数据。当用 SQLDA 把参数送到执行语句时，每一个参数都是一个 sqlvar 结构；当用 SQLDA 返回输出列信息时，每一列都是一个 sqlvar 结构。具体每个元素的含义为：

lSqldaid。用于输入标志信息，如：“SQLDA”。

lSqldabc。SQLDA 数据结果的长度。应该是 16+44*SQLN。Sqldaid、sqldabc、sqln 和 sqld 的总长度为 16 个字节。而 sqlvar 结构的长度为 44 个字节。

lSqln。分配的 Sqlvar 结构的个数。等价于输入参数的个数或输出列的个数。

lSqld。目前使用的 sqlvar 结构的个数。

lSqltype。代表参数或列的数据类型。它是一个整数数据类型代码。如：500 代表二字节整数。具体每个整数的含义见下表：

Sqltype 代码说明 SQL Server 数据类型例子

392/39326 字节长的包含日期和时间的字符串 Datetime,smalldatetimechar date1[27] = Mar 7 1988 7:12PM;

444/445BinaryBinary,varbinary,image,timestampchar binary1[4097];

452/453 小于 254 字节的字符串 Char,varcharchar mychar[255];

456/457 固定长度的长字符串 textstruct TEXTVAR { short len; char data[4097];} textvar;

480/4818 字节的浮点数 Floatdouble mydouble1;

482/4834 字节的浮点数 realfloat myfloat1;

496/4974 字节的整数 Intlong myint1;

500/5014 字节的整数 Smallint,tinyint,bitshort myshort1;

462/463NULL 结尾的字符串 Char,varchar,textchar mychar1[41]; char * mychar2;

lSqllen。代表传送数据的长度。如：2，即代表二字节整数。如果是字符串，则该数据为字符串中的字符数量。

lSqldata。指向数据的地址。注意，仅仅是一个地址。

lSqlind。代表是否为 NULL。如果该列不允许为 NULL，则该字段不赋值；如果该列允许为 NULL，则：该字段若为 0，表示数据值不为 NULL，若为-1，表示数据值为 NULL。

lSqlname。代表列名或变量名。它是一个结构。包含 length 和 data。Length 是名字的长度；data 是名字。

下面我们来看两个具体的例子。第一个例子是通过 SQLDA 查询数据库中的数据。第二个例子是通过 SQLDA 传递参数。

我们首先看一个动态查询的例子。这个例子的作用是，由用户输入表名，查询系统表获得该表的列信息，询问用户是否显示该列数据，若是，则显示；否则，不显示。动态查询的执行过程如下：

1）、如同构造动态 UPDATE 语句或 DELETE 语句的方法一样，程序在缓冲器中构造一个有效的 SELECT 语句。

2）、动态 DECLARE CURSOR 语句说明查询游标，动态 DECLARE CURSOR 语句规定与动态 SELECT 语句有关的语句名称。如：例子中的 querystmt。

3）、程序用 PREPARE 语句把动态查询语句送到 DBMS，DBMS 准备、确认和优化语句，并生成一个应用计划。

4）、程序用 DESCRIBE 语句请求 DBMS 提供 SQLDA 中描述信息，即告诉程序有多少列查询结果、各列名称、数据类型和长度。DESCRIBE 语句只用于动态查询。具体见下一节。

5）、为 SQLDA 申请存放一列查询结果的存储块（即：sqldata 指向的数据区），也为 SQLDA 的列的指示符变量申请空间。程序把数据区地址和指示符变量地址送入 SQLDA，以告诉 DBMS 向何处回送查询结果。

6）、动态格式的 OPEN 语句。即打开存放查询到的数据集（动态 SELECT 语句产生的数据）的第一行。

7）、动态格式的 FETCH 语句把游标当前行的结果送到 SQLDA。（动态 FETCH 语句和静态 FETCH 语句的不同是：静态 FETCH 语句规定了用主变量接收数据；而动态 FETCH 语句是用 SQLDA 接收数据。）并把游标指向下一行结果集。

8）、CLOSE 语句关闭游标。

```
Main()
{
exec sql include sqlca;
exec sql include sqlda;
exec sql begin declare section;
char stmbuf[2001];
char querytbl[32];
char querycol[32];
exec sql end declare section;
/*静态游标*/
exec sql declare tblcurs cursor for
select colname from syscolumns
where tblname = :querytbl;
int colcount = 0;
struct sqlda * qry_da;
struct sqlvar *qry_var;
int I;
char inbuf[101];
/*提示输入想要查询的表名*/
printf(“***Enter name of table for query: “);
gets(querytbl);
/*生成 SELECT 语句中的选择列表*/
strcpy(stmbuf,” select “);
/*设置错误处理过程*/
exec sql whenever sqlerror goto handle_error;
exec sql whenever not found goto no_more_columns;
/*查询系统表，获得列名信息*/
exec sql open tblcurs;
for( ; ; ) {
exec sql fetch tblcurs into :querycol;
printf(“Include column %s(y/n)?”,querycol);
gets(inbuf);
```

```
if(inbuf[0]= ='y'){
/*生成 SELECT 语句的各个列名*/
if (colcount++>0)
strcat(stmtbuf, ”,”);
strcat(stmtbuf,querycol);
}
}
no_more_column:
exec sql close tblcurs;
/*生成 SELECT 语句中的 FROM 部分*/
strcat(stmtbuf,” from”);
strcat(stmtbuf,querytbl);
/*分配 SQLDA 空间*/
qry_da=(SQLDA *) malloc(sizeof(SQLDA)+colcount * sizeof(SQLVAR));
qry_da ->sqln = colcount;
/*声明动态游标，以便逐行处理*/
exec sql declare qrycurs for querystmt;
/*准备查询语句*/
exec sql prepare querystmt from :stmtbuf;
/*获取 SQLDA 的描述信息*/
exec sql describe querystmt into qry_da;
/*为存放一列数据的 SQLDA 申请空间*/
for( I=0; I<colcount; I++){
qry_var=qry_da ->sqlvar + I;
qry_var -> sqldat = malloc(qry_var ->sqllen);
qry_var -> sqlind =malloc(sizeof(short));
}
/*动态 OPEN 语句，打开动态游标，即指向查询数据集的第一行*/
exec sql open qrycurs;
exec sql whenever not found goto no_more_data;
for( ; ;) {
/*将查询到的一行值放在 SQLDA 中*/
exec sql fetch sqlcurs using descriptor qry_da;
printf(“/n”);
for(I=0; I<colcount; I++){
qry_var = qry_da ->sqlvar +I;
/*显示列名信息*/
printf(“Column#%d(%s):”,I+1,qry_var ->sqlname);
/*查看该列是否为 NULL*/
if(*(qry_var ->sqlind)) !=0{
puts(“is NULL!\n”);
```

```
continue;
}
/*按照数据类型，将它显示出来*/
switch(qry_var -> sqltype) {
case 448:
case 449:
/*VARCHAR 数据类型，直接显示*/
puts(qry_var -> sqldata) ;
break;
case 496:
case 497:
/*4 位整数，转化，并显示出来*/
printf(“%ld”,*((int *) (qry_var->sqldata)));
break;
case 500:
case 501:
/*2 位整数，转化，并显示出来*/
printf(“%d”,*((short *)(qry_var->sqldata)));
break;
case 480:
case 481:
/*浮点数，转化，并显示出来*/
printf(“%lf”,*((double *)(qry_var->sqldata)));
break;
}
}
}
no_more_data:
printf(“\nEnd of data.\n”);
for (I=0;I<colcount;I++) {
qry_var=qry_da->sqlvar+I;
free(qry_var->sqldata);
free(qry_var->sqlind);
}
free(qry_da);
close qrycurs;
exit();
}
```

上面这个例子是典型的动态查询程序。该程序中演示了 PREPARE 语句和 DESCRIBE 语句的处理方式，以及为程序中检索到的数据分配空间。要注意程序中如何设置 SQLVAR 结构中的的各个变量。这个程序也演示了 OPEN、FETCH 和 CLOSE 语句在动态查询中的应用。值得注

意的是，FETCH 语句只使用了 SQLDA，不使用主变量。由于程序中预先申请了 SQLVAR 结构中的 SQLDA 和 SQLIND 空间，所以 DBMS 知道将查询到的数据保存在何处。该程序还考虑了查询数据为 NULL 的处理。

例 2、用 SQLDA 规定输入参数。该程序在运行开始时，可以选择要更新的列和值。由于用户能够在每次运行程序时选择不同的列，所以该程序必须用 SQLDA 把参数传送到 EXECUTE 语句。这个例子仅供参考，读者只需读懂即可。

```
Main()
{
#define COLCNT 6
exec sql include sqlca;
exec sql include sqlda;
exec sql begin declare section;
char stmbuf[2001];
exec sql end declare section;
char * malloc()
struct {
char prompt[31]; /*列名的全名*/
char name[31]; /*列名*/
short typecode; /*数据类型代码*/
short buflen; /*数据长度*/
char selected; /*是否更新标志，y 为是，n 为否*/
}columns[]={“Name”,” NAME”,449,16,'n',
 “Office”,” REP_OFFICE”,497,4,'n',
 “Manager”,” MANAGER”,497,4,'n',
 “Hire Date”,” HIRE_DATE”,449,12,'n',
 “Quota”,” QUOTA”,481,8,'n',
 “Sales”,” SALES”,481,8,'n'};
struct sqlda *parmda;
struct sqlvar *parmvar;
int parmcnt; /*参数个数*/
int empl_num; /*员工号*/
int I;
int j;
char inbuf[101]; /*用户输入信息*/
/* 询问用户更新哪些列*/
printf(“****Salesperson Update Program*****\n\n”);
parmcnt = 1;
for (I=0;I<COLCNT;I++) {
printf(“Update %s column(y/n)? “,column[I].prompt);
gets(inbuf);
if (inbuf[0]=='y'){
```

```
column[I].selected =  'y';
parmcnt + = 1;
}
}
/*根据要更新的列数，分配 SQLDA 空间*/
parmda=malloc(16+(44*parmcnt));
strcpy(parmda->sqldaid," SQLDA ");
parmda->sqldabc=(16+(44*parmcnt));
parmda->sqln=parmcnt;
/*开始生成更新语句*/
strcpy(stmbuf," update orders set  ");
j=0;
/*处理列名*/
for (I=0;I++;I<COLCNT) {
if (column[I].selected =='n')
continue;
if (parmcnt>0) strcat(stmbuf,',  ');
strcat(stmbuf,column[I].name);
strcat(stmbuf," =?"); /*生成动态 UPDATE 语句*/
/*为 sqlvar 指定参数的信息，并申请存放新值的空间*/
parmvar=parmda->sqlvar + j;
parmvar->sqltype = column[I].typecode;
parmvar->sqllen = column[I].buflen;
parmvar->sqldata = malloc(column[I].bufflen);
parmvar->sqlind=malloc(2);
strcpy(parmvar->sqlname.data,column[I].prompt);
j+ = 1;
}
/*生成 WHERE 语句*/
strcat(stmbuf,"  where empl_num = ?");
parmvar=parmda + parmcnt;
parmvar->sqltype=496;
parmvar->sqllen=4;
parmvar->sqldata =&empl_num;
parmvar->sqlind=0;
parmda->sqld=parmcnt;
/*编译动态 SQL 语句*/
exec sql prepare updatestmt from :stmbuf;
if (sqlca.sqlcode < 0 ) {
printf("PREPARE error:%ld\n",sqlca.sqlcode);
exit();
```

```
}
/*提示用户输入更新的新值，并将之存放在 SQLDA 中*/
for ( ; ; ) {
printf(“\nEnter Salesperson's Employee Number: “);
scanf(“%ld”,&empl_num);
if ( empl_num == 0) break;
for (j=0;j<(parmcnt  - 1 ); J++) {
parmvar=parmda + j;
printf(“Enter new value for %s: “,parmvar->sqlname.data);
gets(inbuf);
if ( inbuf[0]= =  ‘*’) {
/*如果用户输入*，表示为 NULL 值*/
*(parmvar->sqlind) = -1;
continue;
}
else{
*(parmvar -> sqlind)=0;
switch (parmvar -> sqltype) {
case 481:
sscanf(inbuf,"%lf",parmvar->sqldata);
break;
case 449:
stccpy(parmvar->sqldata,inbuf,strlen(inbuf));
parmvar->sqllen=strlen(inbuf);
break;
case 501:
sscanf(inbuf,"%ld",parmvar->sqldata);
break;
}
}
}
/*执行动态 SQL 语句，并经过 SQLDA 传递参数值*/
exec sql execute updatestmt using :parmda;
if (sqlca.sqlcode) <0 ) {
printf("execute error:%ld\n",sqlca.sqlcode);
exit();
}
}
exec execute immediate "COMMIT WORK";
if (sqlca.sqlcode)
printf("COMMIT error:%ld\n",sqlca.sqlcode);
```

```
else
printf("\n All update commited.\n");
exit();
}
```

在上述例子中，生成的动态 UPDATE 语句为：

```
update salesreps
set name =? , office=? , quota=?
Where empl_num=?
```

该语句规定了 4 个参数，程序分配了一个处理四个 sqlvar 结构的 SQLDA。用于提供参数值。

6.4.4 DESCRIBE 语句

该语句只有动态 SQL 才有。该语句是在 PREPARE 语句之后，在 OPEN 语句之前使用。该语句的作用是，设置 SQLDA 中的描述信息，如：列名、数据类型和长度等。DESCRIBE 语句的语法为：

DESCRIBE 语句名 INTO 描述符名

如：exec sql describe querystmt into qry_da;

在执行 DESCRIBE 前，用户必须给出 SQLDA 中的 SQLN 的值（表示有多少列），该值也说明了 SQLDA 中有多少个 SQLVAR 结构。然后，执行 DESCRIBE 语句，该语句填充每一个 SQLVAR 结构。每个 SQLVAR 结构中的相应列为：

lSQLNAME 结构：列名放在 DATA 字段，列的长度放在 LENGTH 字段。

lSQLTYPE 列：给出一个数据类型的整数代码。

lSQLLEN 列：给出列的长度。

lSQLDATA 列和 SQLIND 列：不填充。由程序在 FETCH 语句之前，给出数据缓冲器地址和指示符地址。

1.6.5 API

编写嵌入 SQL 程序完成的功能也可以通过 DB-Library 或 ODBC API 的函数调用来完成。下面这个例子是使用 DB-Library 来连接 SQL Server 并执行一个简单的查询。类似于第一节中的例 1。

```
#define DBNTWIN32
#include <sqlfront.h>
#include <sqldb.h>
main()
{
DBPROCESS *dbproc;
LOGINREC *login;
RETCODE r;
dbinit();
login = dblogin();
if (login == NULL)
```

```
return (1);
DBSETLUSER(login, "sa");
DBSETLPWD(login, "password");
dbproc = dbopen(login, "YANGZH");
dbfreelogin(login);
if (dbproc == NULL)
return (1);
dbuse(dbproc, "pubs");
dbcmd(dbproc,
"select au_fname from authors where au_lname = 'White'");
r = dbsqlexec(dbproc);
if (r == FAIL)
return (1);
while (1)
{
r = dbresults(dbproc);
if (r == SUCCEED)
{
/* Process the rows with dbnextrow() */
}
if ((r == FAIL) || (r == NO_MORE_RESULTS))
break;
}
return (0);
}
```

嵌入 SQL 的程序有一些缺点。各个数据库厂商提出的嵌入 SQL 语言各不相同。尤其在 SQLDA 的定义上。另外，Microsoft SQL SERVER 的嵌入 SQL 的 C 程序不是线程安全的。如果你在一个线程应用中使用 ESQL/C，那么应该仅仅在一个单一线程中调用 E/SQL。最好在主线程中使用。所以，在可能的情况下，应该使用 API 来代替嵌入 SQL。

二、 ESQL编程使用说明

2.1 第一章 ESQL介绍

本章对 ESQL 做一概括介绍,主要讨论怎么使用 ESQL、ESQL 的基本的概念和定义、ESQL 程序的各个部分和 ESQL 程序中语句的类型.

SQL 语言是非过程化语言,大部分语句的执行与其前面或后面的语句无关,而一些高级编程语言都是基于如循环,条件等结构的过程化语言,尽管 SQL 语言非常有力,但它却没有过程化能力.若把 SQL 语言嵌入到过程化的编程语言中,则利用这些结构,程序开发人员就能设计出更加灵活的应用系统,具有 SQL 语言和高级编程语言的良好特征,它将比单独使用 SQL 或 C 语言具有更强的功能和灵活性.

COBASE RDBMS 提供两种工具在主语言中编程来存取 COBASE 数据库中的数据.即高级语言预编译程序接口(ESQL)和高级语言的函数调用接口(CCI).目前这些工具仅支持 C 语言.

COBASE RDBMS 提供的 ESQL 工具把含有 SQL 语句的 C 程序转化为可存取和操纵 COBASE 数据库中数据的 C 程序,作为一编译器,ESQL 把输入文件中的 EXEC SQL 语句在输出文件中转化为适当的 CCI 函数调用.输出文件则可以正常的 C 程序的方式被编译、连接和执行.

2.1.1 ESQL中的基本概念

ESQL 中的基本概念主要有:

1.嵌入的 SQL 语句:

嵌入的 SQL 语句是指在应用程序中使用的 SQL 语句.该应用程序称作宿主程序,书写该程序的语言称作宿主语言.嵌入的 SQL 语句与交互式 SQL 语句在语法上没有太大的差别,只是嵌入式 SQL 语句在个别语句上有所扩充.如嵌入式 SQL 中的 SELECT 语句增加了 INTO 子句,以便与宿主语言变量打交道.此外,嵌入式 SQL 为适合程序设计语言的要求,还增加了许多语句,如游标的定义、打开和关闭语句等等.

2.执行性 SQL 语句和说明性 SQL 语句:

嵌入的 SQL 语句主要有两种类型:执行性 SQL 语句和说明性 SQL 语句.执行性 SQL 语句可用来连接 COBASE,定义、查询和操纵 COBASE 数据库中的数据,每一执行性语句真正对数据库进行操作 ,执行完成后,在 USERCA 中存放执行信息.说明性语句用来说明通讯域和 SQL 语句中用到的变量. 说明性语句不生成执行代码,对 USERCA 不产生影响.

3.事务:

事务是逻辑上相关的一组 SQL 语句.COBASE 把它们视作一个单元.为了保持数据库的一致性,一事务内的所有操作要么都做,要么都不做.

2.1.2 ESQL程序的组成和运行

在 ESQL 程序 中嵌入的 SQL 语句以 EXEC 作为起始标识,语句的结束以";"作为标识.在嵌入的 SQL 语句可以使用主语言(这时是 C 语言)的程序变量(即主变量),这时主变量名前加冒号(:)作为标志,以区别于字段名.

ESQL 程序包括两部分:程序首部和程序体.程序首部定义变量,为 ESQL 程序做准备, 程序体包括各种 SQL 语句来操纵 COBASE 数据库中的数据.

编制并运行 ESQL 程序比单独使用纯 C 语言多一个预编译过程,通常具有以下几个步骤:

1.编辑 ESQL 程序(可利用编辑软件如: EDLIN, WS 等进行编辑).程序保后缀为.ec.

2.使用 COBASE 的预编译器 ETE 对 ESQL 源程序进行预处理,该编译器将源程序中嵌入的 SQL 语言翻译成标准 C 语言,产生一个 C 语言编译器能直接进 行编译的文件.其文件的扩展名为.cpp。该 cpp 文件可以和普通的 cpp 文件一样被放入一个工程中被 C++编译器编译，连接最后运行。

对 COBASE 的预编译器的使用的详细说明见第六章.

2.2 第二章 ESQL 程序的基本结构

ESQL 程序由两部分组成:程序首部和程序体.

2.2.1 程序首部

每一个 ESQL 程序的开始,就是程序的首部,它包括以下三部分:

1.DECLARE 部分:

说明特殊的主变量,这些变量区别于纯 C 语言程序中的变量,COBASE 使用这些变量与程序之间相互作用.

2.INCLUDE USERCA 语句: 说明一个 SQL 语句的通讯域(USERCA),它提供了错误处理,其功能等价于代替 C 语言中的#INCLUDE 语句.

3.CONNECT 语句: 建立程序与 COBASE 之间的连接.

2.1.1DECLARE 部分:(描述部分)

在 DECLARE SECTION (描述部分),定义所有在 SQL 语句中用到的主变量,定义部分是以:

EXEC SQL BEGIN DECLARE SECTION;

和 EXEC SQL END DECLARE SECTION;

开始和结束的.

在这两个语句中,只可以定义 SQL 语句中用到的主变量,每个预编译单元只允许一个 BEGIN/END DECLARE SECTION (描述部分),但是一个程序可以包含许多独立的预编译单元。若一个主变量或指示变量在 ESQL 程序中的 SQL 语句中引用,但它没有在 描述部分中定义,则程序在预编译时就会出现错误信息.

在这两个语 句中可以定义的变量有六种类型:INT,SHORT,CHAR,FLOAT,NUMBER,DATE.其中 CHAR 型允许定义二维数组,其它类型只允许定义一维数组,不允许有指针类型 ,在此处变量可以赋值.

例如:

```
EXEC SQL BEGIN DECLARE SECTION;
int sno;
char sname[11];
```

short snameid;

EXEC SQL END DECLARE SECTION;

(1)主变量:

就是用在 SQL 语句中的主语言变量.主要用于程序变量和数据库内部交换数据之用,它们的数据类型必须是在主语言中描述部分里定义过的,而且它们的数据类型必须和 COBASE 数据库中已定义的表中的数据类型相匹配.

例如:

SELECT 姓名, 等级

FROM 供应商

INTO :sname, :status

WHERE 供应商号=:sno;

该语句表示,从供应商表中在供应商号与主变量 sno 一致的地方选择供应商姓名和供应商等级,COBASE 把结果传送到主变量 sname,status 中.

主变量使用规则:

1.必须在描述部分明确定义.

2.必须使用与其定义相同的大小写格式.

3.在 SQL 语句中使用主变量时必须在主变量前写一个冒号":",在纯 C 语言 语句中则不要在主变量前写冒号.

4.不能是 SQL 命令的保 留字.

5.在一条语句中只能使用一次.

2.1.2 SQL 通讯域

每个COBASE应用程序必须提供对错误 的处理,为了说明SQL通讯域(USERCA),必须在每个 COBASE 预编译程序中写上:

EXEC SQL INCLUDE USERCA;

USERCA 是一结构,每一嵌入的执行性 SQL 语言的执行情况在其执行完成后写入 USERCA 结构中的各变量中, 根据USERCA中的内容可以获得每一嵌入SQL语句执行后的信息,编制程序时就可以做适当的处理.对其的详细说明见第五章.

2.1.3 连接 COBASE

在 存取COBASE数据之前,每一个预编译程序必须与COBASE连接.连接时,程序必须提供用户名和口令,由 COBASE 进行校验,若口令和用户名正确,方可登录 COBASE,获得使用权,否则,COBASE 拒绝登录,则程序就不能使用数据库.缺省的用户名为“cobase”, 口令为“cobase”。

连接 COBASE 的格式如下:

EXEC SQL CONNECT <用户名>:<用户口令>.

CONNECT 语句必须是 ESQL 程序中第一条可执行的 SQL 语句.

2.2.2 程序体

程序体可以包含许多 SQL 语句,以查询或处理存储在 COBASE 数据库中的数据.

在应用程序中所包含的 SQL 语句,可以查询或操纵存储在 COBASE 中的数据,这些语句叫做数据操纵语言(DML),应用程序体也可以包含 DDL 语句,用来建立或定义数据结构,如表、视图或索引.在用户程序中写入的任何有效的 SQL 语句都可以被执行，只需要把完整的 SQL 语句按嵌入式的要求写入 C 语言的合适位置即可.嵌入在 C 语言中的 SQL 语句以 EXEC SQL 开始,以";"为结束标志.SQL 语句中可以嵌入主变量,主变量前应有":"标志.如下面例子都是嵌入式 SQL 语句:

```
EXEC SQL UPDATE 供应商
SET 姓名='李 红'
WHERE 供应商号='S1';
/* 把供应商号是 S1 的供应商姓名改为 '李 红'*/
EXEC SQL INSERT INTO 供应商(供应商号,姓名,等级,城市)
VALUES((:sno,:sname,:status,:city));
/* 根据宿主变量值插入供应商表中 */

EXEC SQL DELETE FROM 供应商
WHERE 等级 IS NULL;
/*删除供应商等级是空值的供应商*/
```

从上面例可以看出,静态的增,删,改语句与交互方式没有太大的差别,对于查询语句就没有那么简单,查询语句在下一章中介绍.

现用几个例题程序加以说明.

例题程序 1 (建立一表并向表中插入数据)

```
/*=============================================================================
=====
This is a sample program which include SQL sentence about
CREATE ,INSERT a table.
=============================================================================
===*/
EXEC SQL BEGIN DECLARE SECTION ;
CHAR sno[10],sname[10],city[10];
INT status;
EXEC SQL END DECLARE SECTION ;
EXEC SQL INCLUDE USERCA;
main()
{
int i;
EXEC SQL CONNECT "cobase:cobase" ;
EXEC SQL CREATE TABLE S_TEST /* Create a table named S_TEST */
(SNO CHAR (9),
SNAME CHAR (20),
STATUS INT ,CITY CHAR(10));
```

```
    printf(" create table success,insert?");
    printf(" 0 ---- no ");
    printf(" 1 ---- yes ");
    printf(" choice:");
    scanf("%d",&i);
    while(i)
    {
    printf("input sno:");
    scanf("%s",sno);
    printf("input sname:");
    scanf("%s",sname);
    printf("input status:");
    scanf("%d",&status);
    printf("input city:");
    scanf("%s",city);
    EXEC SQL INSERT INTO S_TEST(SNO,SNAME,STATUS,CITY)
    VALUES ((:sno,:sname,:status,:city));
    printf("continue?");
    printf(" 0 ----terminate ");
    printf(" 1 ----continue ");
    printf(" choice:");
    scanf("%d",&i);
    }
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT; /*log off database*/
    exit(0);
    }
```

例题程序 2（修改和删除表中的数据）

```
    /*=========================================================================
====
    This is a sample program which include SQL sentence about
    UPDATE,DELETE a table.
    =========================================================================
===*/
    EXEC SQL BEGIN DECLARE SECTION ;
    CHAR sno[10],sname[10],city[10];
    INT status;
    EXEC SQL END DECLARE SECTION ;
    EXEC SQL INCLUDE USERCA;
    main()
```

```
{
int i;
printf("Now exec connect ...");
EXEC SQL CONNECT "COBASE:COBASE" ;

printf(" update?");
printf(" 0 ---- no ");
printf(" 1 ---- yes ");
printf(" choice:");
scanf("%d",&i);
while(i)
{
printf("input sno:");
scanf("%s",sno);
printf("input sname:");
scanf("%s",sname);
printf("input status:");
scanf("%d",&status);
printf("input city:");
scanf("%s",city);
EXEC SQL UPDATE S_TEST SET sname=:sname,city=:city ,status=:status
WHERE sno=:sno;
printf(" continue?");
printf(" 0 ----no ");
printf(" 1 ----yes ");
printf(" choice:");
scanf("%d",&i);
}
printf(" delete?");
printf(" 0 ---- no ");
printf(" 1 ---- yes ");
printf(" choice:");
scanf("%d",&i);
while(i)
{
printf("input sno:");
scanf("%s",sno);
EXEC SQL DELETE FROM S_TEST WHERE sno=:sno;
printf(" continue?");
printf(" 0 ----no ");
printf(" 1 ----yes ");
```

```
printf(" chioce:");
scanf("%d",&i);
}
EXEC SQL COMMIT;
EXEC SQL DISCONNECT;
exit(0);
}
```

2.3 第三章 查 询

在 ESQL 程序中,查询可以分为两大类型:返回一行的查询和返回多行的查询.对于查询,我们不仅对其执行得成功与否感兴趣,其结果更为有用.多行查询要用到游标的概念,本章就介绍查询语句和游标的概念和使用.

2.3.1 SELECT 语句

SELECT 语句是用于完成查询功能的 SQL 语句,查询语句因为有返回的结果,故 ESQL 中的 SELECT 语句比 SQL 的 SELECT 语句多一 INTO 子句,INTO 子句的主变量表对应于程序中主变量,用于存放查询返回的结果.

SELECT 语句格式如下:

EXEC SQL SELECT <列名> [,<列名>[,...]] INTO <主变量表>

FROM <表名> [,<表名>[,...]] [WHERE <检索条件>];

其中:1.检索条件中允许有主变量和嵌套子查询语句.

2.INTO 后的主变量可以是数组.

3.主变量前要用":"标志.

执行该语句时,COBASE 找出表中满足检索条件的行,并把结果传送到 INTO 子句中所对应的主变量中.该语句的查询结果可以是一行或多行.WHERE 后的主变量叫输入主变量,它提供了查询所需的信息.INTO 子句中的主变量叫输出主变量,它保存 SELECT 语句运行后的结果.

例题程序 3 (返回一行的查询)

```
/*=============================================================================
====
This is a sample program which uses SELECT statement of Esql.
It is an example of how to do queries that return one row.
=============================================================================
===*/

EXEC SQL BEGIN DECLARE SECTION ;
INT status;
```

```
    CHAR sno[10],sname[10],city[10],isno[10];
    EXEC SQL END DECLARE SECTION ;
    EXEC SQL INCLUDE USERCA;
    main()
    {
    EXEC SQL CONNECT "cobase:cobase" ; /* log into COBASE */

    printf("input the sno for update:");
    scanf("%s",sno);
    EXEC SQL SELECT sno,sname,status,city
    INTO :sno , :sname ,:status,:city
    from S_TEST WHERE sno=:sno;
    printf("sno: %s ;",sno);
    printf("sname: %s ;",sname);
    printf("status: %d ;",status);
    printf("city: %s ;",city);
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;
    exit(0);

    }
```

例题程序 4 (返回多行的查询)

```
    /*=========================================================================
====
    This is a sample program which uses SELECT statement of Esql.
    It is an example of how to do queries that return more than
    one row.
    =========================================================================
===*/

    EXEC SQL BEGIN DECLARE SECTION ;
    INT status[10];
    CHAR sno[10][10],sname[10][15],city[10][20],isno[10];
    EXEC SQL END DECLARE SECTION ;
    EXEC SQL INCLUDE USERCA;
    main()
    { int i;
    EXEC SQL CONNECT "cobase:cobase" ; /* log into COBASE */
    for(i=0;i<10;i++)
    {
```

```
strcpy(sno[i],"ttttt");
strcpy(sname[i],"ttttt");
strcpy(city[i],"ttttt");
status[i]=1000;
}
EXEC SQL SELECT sno,sname,status,city
INTO :sno , :sname ,:status,:city
from S_TEST;
printf("sno sname status city");
for(i=0;i<10;i++)
{ printf("%8s %8s %8d %8s ",sno[i],sname[i],status[i],city[i]);
getchar();
}
EXEC SQL COMMIT;
EXEC SQL DISCONNECT;
}
```

2.3.2 游标的使用

我们知道 SQL 语言是一种面向集合的语言,而普通的高级语言则是面向记录的语言,要想使 SQL 语言能与宿主语言通讯,ESQL 提供了游标操作,能把 SQL 的集合操作结果,按单个记录方式取出,赋于主变量进行进一步的处理.

如果查询结果返回多行或不知返回多少行,就可使用带游标的 SELECT 语句.一个游标(CURSOR)是一个 COBASE 和 ESQL 使用的工作区域,COBASE 使用这个工作区存放着一个查询结果 .一个已命名的游标是和一条 SELECT 语句相关联.一个游标必须首先定义(同一个查询相关联),然后用三条可运行的 SQL 语句使用游标,以操纵数据.四条操纵游标的命令如下:

.DECLRE CURSOR

.OPEN CURSOR

.FETCH

.CLOSE CURSOR

DECLARE CURSOR 语句用来定义一游标,此时游标处于关闭状态.用 OPEN CURSOR 语句打开游标后,就可用它从相关的查询中取出多于一行的结果.所有满足查询条件的行组成一个集合,叫做游标活动集.通过 FETCH 取操作,活动集中的每一行可以一个一个的返回,当查询作完后,游标就可以用 CLOSE CURSOR 语句关闭.

3.2.1DECLARE CURSOR 定义游标语句:

ESQL 中的 DECLARE CURSOR 语句定义游标,赋给它一个与查询相关的游标名.该语句的格式为:

EXEC SQL DECLARE <游标名> CURSOR FOR

<SELECT 语句> [FOR UPDATE];

其中:(1)SELECT 语句应不含 INTO 子句.

(2)若无 FOR UPDATE 则无法 执行 UPDATE(定位)和 DELETE(定位)语句.

定义游标的 DECLARE 语句必须出现在程序中对游标进行操作的所有语句之前, ESQL 不能引 用没有说明的游标,游标的定义范围是整个程序.程序中可包含多个 DECLARE 语 句,这些语句定义了不同的游标,并把游标与不同的查询联系在一起,所以在同一个程序中的两个 DECLARE 语句中不能说明同一个游标名.

3.2.2 OPEN CURSOR 打开游标语句

ESQL 中 OPEN CURSOR 语句格式如下:

```
EXEC SQL OPEN <游标名>;
```

OPEN 语句决定了满足查询的行的集合,游标处于打开状态,它的活动集就是满足 WHERE 子句条件的行的集合.这时,游标处在活动集的第一行的 前面.

3.2.3 FETCH CURSOR 语句

ESQL 中的 FETCH CURSOR 语句读出活动集中的行,并把结果送到输出主变量,输出主变量是在相关的 FETCH 语句中定义的.其 格式如下:

```
EXEC SQL FETCH <游标名> INTO <主变量表>;
```

游标必须先定义,然后再打开,只有当游标处于打开状态时,才执行 FETCH 语句.在第一次运行 FETCH 时,游标从活动集的第一行前移到当前第一行,使这一行成为当前行.每次运行 FETCH 时游标在活动集中向前移,把选出的结果送到主变量表中指定的输出主变量中.

如果游标活动集中是空的,或所有的行已经被取走,COBASE 就返回一代码.(USERCA.SQLCODE==2000).

游标只可在活动集中向前移动,COBASE 无法取到已经用 FETCH 取过的行,要想再取这一行,就必须关闭游标,再重新打开它.

3.2.4 CLOSE CURSOR 关闭游标语句

当取完活动集中所有行后,必须关闭游标,以释放与该游标的关的资源.其格式如下:

EXEC SQL CLOSE <游 标名>;

例题程序 5 (使用游标的查询)

```
/*=============================================================================
====
This is a sample program which uses Cursor.Include DECLARE,
OPEN,FETCH and CLOSE cursor name.
It is an example of how to do queries that return more than
one row.
=============================================================================
===*/

EXEC SQL BEGIN DECLARE SECTION ;
```

```
INT status;
CHAR sno[10],sname[15],city[20];
EXEC SQL END DECLARE SECTION ;
EXEC SQL INCLUDE USERCA;
main()
{
int i;
printf("Now exec connect ...");
EXEC SQL CONNECT "cobase:cobase" ; /* log into COBASE */
printf("Now exec declare_open_fetch_close cursor ...");
/* Declare statement name ---"s1" for INSERT statement */
EXEC SQL DECLARE cursor1 CURSOR FOR SELECT * FROM S_TEST;
EXEC SQL OPEN cursor1 ;
printf("sno sname status city\n ");
do
{
EXEC SQL FETCH cursor1 INTO :sno,:sname,:status,:city ;
If (userca.sqlcode==2000) break;
printf("%8s %8s %8d %8s \n",sno,sname,status,city);
getchar();

}while(1);
EXEC SQL CLOSE cursor1 ;
EXEC SQL COMMIT;
EXEC SQL DISCONNECT;
}
```

2.3.3 定位修改和删除语句

COBASE 支持 SQL 格式" CURRENT OF CURSOR".这条语句将指向一个游标中最新取出的行,以用于修改和删除操作.该语句必须在取操作之后使用 ,它等同于存储一个 ROWID,并使用它.其格式如下:

```
(1) EXEC SQL
UPDATE <表名>
SET <列名> = <值表达式> | NULL
[,<列名> = <值表达式> | NULL ....]
WHERE CURRENT OF <游标名> ;
(2) EXEC SQL
UPDATE <表名>
SET ( <列名表> ) = ( <子查询> )
WHERE CURRENT OF <游标名> ;
```

(3) EXEC SQL DELETE FROM <表名>

WHERE CURRENT OF <游标名> ;

这些语句执行在游标名的当前行下更新或修改.其中在值表达式或子查询中出现的主变量前应有":"标志.

例题程序 6 (定位删除)

```
/*=========================================================================
=====
This is a sample program which uses DELETE STATEMENT at
CURRENT Cursor.
=========================================================================
===*/

EXEC SQL BEGIN DECLARE SECTION ;
INT status;
CHAR sno[10],sname[15],city[20];
EXEC SQL END DECLARE SECTION ;
EXEC SQL INCLUDE USERCA;
main()
{
int i;
printf("Now exec connect ...");
EXEC SQL CONNECT "COBASE:COBASE" ;
printf("Now exec declare cursor ...");
EXEC SQL DECLARE cursor1 CURSOR FOR
SELECT sno,sname,status,city FROM S_TEST FOR UPDATE;
printf("Now exec open cursor ...");
EXEC SQL OPEN cursor1;
printf("sno sname status city\n");
for(;;)
{
printf("Now exec fetch cursor ...\n");
EXEC SQL FETCH cursor1 INTO :sno,:sname,:status,:city;
if (userca.sqlcode==2000) break;
printf("%8s %8s %8d %8s \n",sno,sname,status,city);


//Delete the first record that be fetched
printf(" delete current ?(0/1)");
scanf("%d",&i);
if (i==1)
EXEC SQL DELETE FROM S_TEST
```

```
    WHERE CURRENT of cursor1;
    }
    EXEC SQL CLOSE cursor1;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;
    }
```

例题程序 7（定位修改）

```
    /*=========================================================================
====
    This is a sample program which uses UPDATE STATEMENT at
    CURRENT Cursor.
    =========================================================================
===*/

    EXEC SQL BEGIN DECLARE SECTION ;
    INT status;
    CHAR sno[10],sname[15],city[20];
    EXEC SQL END DECLARE SECTION ;
    EXEC SQL INCLUDE USERCA;
    main()
    {
    int i;
    strcpy(sno,"ttttt");
    strcpy(sname,"ttttt");
    strcpy(city,"ttttt");
    status=1000;
    EXEC SQL CONNECT "cobase:cobase" ;
    EXEC SQL DECLARE cursor1 CURSOR FOR
    SELECT SNO, SNAME, STATUS,CITY FROM S_TEST FOR UPDATE;
    EXEC SQL OPEN cursor1;

    printf("sno sname status city\n");
    for(;;)
    {
    EXEC SQL FETCH cursor1 INTO :sno,:sname,:status,:city;
    if (userca.sqlcode==2000) break;
    printf("%8s %8s %8d %8s \n",sno,sname,status,city);
    /* Delete the first record that be fetched */
    printf(" update current ?(0/1)");
```

```
scanf("%d",&i);
if (i==1)
{printf("input sno=");
scanf("%s",sno);
printf("\ninput sname=");
scanf("%s",sname);
printf("\ninput status=");
scanf("%d",&status);
printf("\ninput city=");
scanf("%s",city);
EXEC SQL UPDATE S_TEST SET sno=:sno,sname=:sname,status=:status,city=:city
WHERE CURRENT of cursor1;
EXEC SQL COMMIT;

}
}
EXEC SQL CLOSE cursor1;
EXEC SQL COMMIT;
EXEC SQL DISCONNECT;
}
```

2.4 第四章 提交/回滚事务

本章定义一事务或叫逻辑工作单元.为了保证数据库的一致性,在 ESQL 程序中,程序开发人员可以控制事务是否提交或回滚.一事务通常可以理解为一个完整程序对数据库进行的所有操作.一个事务也可以理解为是一个逻 辑工作单元.

2.4.1 逻辑工作单元

一个逻辑工作单元是一组 SQL 语句和插入的主语言码的集合.COBASE 把它们作为一个整体来处理.

在逻辑工作单元这一级上,COBASE 保证了数据的一致性,这就意味着要么完成所有的操作,要么每条语句都不执行.

如果在处理一个逻辑工作单元期间出现了系统或用户程序失败,那么 COBASE 就自动把数据恢复到该逻辑工作单元开始之前的状态,程序失败时,COBASE 检测完错误就恢复数据,若系统故障,COBASE 在重新启动时恢复数据.

当遇 到第一个可执行的SQL 语句(除了 CONNECT)时,就隐含着一个逻辑工作单元的开始,COMMIT 和ROLLBACK语句结束一个逻辑工作单元.在ESQL程序中,说明语句并不开始一个逻辑单元.

COMMIT 语句保证了当前逻辑单元上的所有操作都完整地提交给了数据库.ROLLBACK 语句取消对当逻辑工作单元所作的操作 ,把数据库恢复到当前逻辑工作单元开始前的状态.

2.4.2 COMMIT 语句

该语句结束当前逻辑工作单元,把在逻辑工作单元期们的所有变化提 交给数据库 .其格式如下:

EXEC SQL COMMIT;

在程序结束之前,应该明确地结束它的工作单元,否则,若程序成功结束后,COBASE 自动提交所有的变换,若程序非正常结束,就恢复到最近 没有提交的逻辑工作单元.

COMMIT 语句不影响主变量的内容 或主程序的控制流.

每条DDL 语句的执行,自动发出COMMIT 操作,这就是说 ,DDL 语句跟在DML 语句后面,那么以前的DML 语句就自动提交组数据库.一个DDL语句结束当前逻辑工作单元,释放该程序拥有的所有锁.

2.4.3 ROLLBACK语 句

该语句将数据库恢复到当前逻辑工作单元之前的状态,结束当前的逻辑工作单元.该语句不影响主变量的内容或主程序的控制流.其格式如下:

EXEC SQL ROLLBACK;

2.4.4 DISCONNECT 语句

当应用程序不再使用COBASE 数据库时,应该使用DISCONNECT 语句释放程序所有与COBASE 数据库有关的资源,并退出数据库,脱离COBASE 环境.其格式如下:

EXEC SQL DISCONNECT;

2.5 第五章 错误检测和恢复

本章我们介绍如何使用USERCA 来进行错误检测和处理。

2.5.1 USERCA的结构

USERCA 是 ESQL 程序用来传送执行信息的结构,每执行完一条执行性 SQL 语句, COBASE 都把执行信息写入USERCA 中,对于说明性SQL 语句,则无执行信息.谨慎的程序员应该在每一 SQL 语句执行完成后,检查 USERCA 结构中内容来确信语句的执行是否成功,并根据其中的信息作适当的处理.在 ESQL 中,USERCA 的结构如下:

```
typedef struct
char caid[10]; /* userca ID */
long calen; /* userca length */
long sqlcode; /* sql code */
long sqltype; /* sql statement type */
int sqlerrmlen; /* sql error message length */
char sqlerrmtext[80]; /* sql error message text */
int sqlreturnflag; /* sql return flag(def or data) */
long sqlpl; /* sql process lines (per fetch st) */
long sqlcoml; /* sql communication lines (per com) */
long sqltotal; /* select_total_lines */
char sqlwarn[7]; /* sql warnning flag */
short sqlstsave; /* sql_statement_save flag */
user_com_area;
struct user_com_area userca;
```

该结构的各元素的意义描述如下:

userca.caid 通讯区标识.

userca.calen 通讯区长度.

userca.sqlcode 记录每一 SQL 语句执行完成情况.其取值如下:

0 表示执行成功.

2000 表示没有返回行或最后一行已取完.

userca.sqltype SQL 语句的类型.

userca.sqlerrmlen 执行 SQL 语句错误 信息的长度.

userca.sqlerrmtext 执行 SQL 语句错误 信息的正文.

userca.sqlreturnflag

userca.sqlpl

userca.sqlcoml

uaseca.sqltotal

userca.sqlwarn

userca.sqlwarn[0] 警告检查位;

userca.sqlwarn[1] 返回值截断警告;

userca.sqlwarn[2] 在集函数中忽略空值警告;

userca.sqlwarn[3] SELECT_LIST 个数与 INTO 子句项
个数不符的警告;
userca.sqlwarn[4] DML 操作涉及每一行的警告;
userca.sqlwarn[5] SQL 语句引起事务回滚的警告;
userca.sqlwarn[6] DELETE 语句对于 FOR_UPDATE 的行操作警告;
userca.sqlstsave

2.6 第六章 使用说明书

Cobase支持两种方式对数据库中的数据进行访问—交互方式和嵌入C程序（ESQL）的方式。Cobase采用的是Client/Server结构，Client端将对数据库的各种访问请求发送到服务器方，交由服务器方处理。服务器对发送来的请求进行分析和处理，然后将执行结果发送回Client端。交互式（ISQL）和嵌入式C程序都是运行在Client端的进程，通过以网络方式和服务器建立连接来进行通讯。下面简要介绍这两种方式的使用。

2.6.1 启动Cobase：

无论使用这两种方式中的哪一种，在和数据库进行交互之前都要首先启动 Cobase 的 DBMS。

?;启动 Cobas 的 DBA 进程：运行 Cobase.exe 将启动 Cobase 的 DBA,这时屏幕将出现两个窗口，一个是控制窗口，一个是消息窗口。控制窗口用于完成对系统的控制，包括初始化系统，选择和系统的连接方式，及断开连接，退出系统等。以后的用户操作都在控制窗口中进行，消息窗口只用于显示一些系统信息。

?;初始化系统：第一次启动 Cobase 需对系统初始化。选中主菜单中的 File 菜单项，在弹出的子菜单中选中 Initialize，在弹出的对话框中选中“确定”即可完成对系统的初始化。该步骤只需在第一次进入系统时调用，或当你认为需要清除系统中已存在的所有数据，对整个系统初始化时使用。

?;选择和Cobase DBMS的连接方式：在控制窗口的主菜单中选中File，在弹出的子菜单中选择NetWork Share，以网络方式和Cobase建立连接。在弹出的对话框提示DBA启动成功后，进入下一步。

2.6.2 退出Cobase:

?;Cobase DBA Shutdown:在退出Cobase之前将DBA Shutdown。选中File菜单项，在弹出的子菜单中选择Normal Shutdown，那么DBA将shutdown。如果Client端的进程非正常终止，则选择Immediate Shutdown。如果再需要Cobase DBMS的服务，则须重新启动DBA，以网络方式和Cobase建立连接，然后启动服务器进程shadow.

?;退出 Cobase:选中 File 菜单项，在弹出的子菜单中选择 Exit。

2.6.3 交互式SQL（Interactive SQL）访问

交互式 SQL 提供了一种交互式的方法对数据库中的数据进行访问。在交互式的界面中只能执行交互式的 SQL 语句—DDL 语句,DML 语句，COMMIT 和 ROLLBACK。并且 DML 语句中不能含有主变量。交互式 SQL 不支持游标。ISQL 将输入的 SQL 语句发送给服务器方执行，最后负责从服务器方将数据取回来，显示在交互式的界面中。

?;启动 ISQL:执行 SISQL.EXE，启动 ISQL。

?;登录到 Cobase：在弹出的窗口菜单中选择 FILE，然后选择 Logon 子菜单项进行登录。登录的用户名为 cobase，口令也为 cobase。

?;进入 SQL 命令状态：在产生的窗口中，由三部分组成。第一部分标记为 SQL data，用于对查询结果的显示。第二部分标记为 Statistics，用于显示对 SQL 语句执行结果的反馈信息。第三部分标记为 SQL Command，用于输入要求执行的 SQL 语句。首先在 SQL Command 中输入“SQL”，要求以下进入 SQL 语句的执行状态。

?;执行 SQL 语句：在 SQL Command 编辑框中输入要求执行的 SQL 语句，一次一条，每个语句要求以分号结束。输入完毕后，单击“Execute”语句即被执行，执行结果将在 SQL data 或 Statistics 中被显示。

?;退出 ISQL：在 SQL Command 中输入“logout”，即可退出 ISQL，同时也将关闭服务器方的 shadow 进程。

2.6.4 嵌入式SQL（Enbeded SQL）编程方式

嵌入式 SQL 在前面的章节中已经介绍了。我们把在 C 语言中嵌入 SQL 语句的程序简称为 EC 程序。开发一个 EC 程序的基本步骤如下：

1. 编辑 ESQL 程序：可以使用文本编辑器如 VC 的编辑器编制一个 ESQL 程序，以.ec 作为文件的扩展名。

2. 预编译：使用 COBASE 的预编译器 ETE 对 ESQL 源程序进行预处理,该编译器将源程序中嵌入的 SQL 语句翻译成 C++语言形式的对 Cobase 库函数的调用,生成文件的扩展名为.cpp.

启动 ETE.exe,在弹出的对话框中,输入要进行预处理的.ec 文件。(该文件本身要以.ec 结尾，但在这里输入的文件名无须加上.ec 后缀，预编译器会自动查找以.ec 结尾的同名文件。)

ETE 的调用格式为:

ETE <filename>

<filename> 为含有嵌入式 SQL 语句(ESQL)的 C/C++语言文本文件名;

3. 生成项目：创建一个相应的项目，将预编译生成的.cpp 文件加入到该项目中。

4. 项目设置：选中 VC 的 Project/setting,在弹出的对话框中选择 Link 标签。

在 Object/Library Modules 文本框中加入库文件 wetelib.lib,wccilib.lib;

选中 Tools/Options,在弹出的对话框中选择 Directories/Library files,设置 各库文件的路径。

5.运行：

?;启动 Cobas：运行 Cobase.exe 将启动 Cobase,这时屏幕将出现两个窗口，一个是控制窗口，一个是消息窗口。控制窗口用于完成对系统的控制，包括初始化系统，选择和系统的连接方式，及断开连接，退出系统等。以后的用户操作都在控制窗口中进行，消息窗口只用于显示一些系统信息。

?;初始化系统：第一次启动 Cobase 需对系统初始化。选中主菜单中的 File 菜单项，在弹出的子菜单中选中 Initialize，即可完成对系统的初始化。该步骤只需在第一次进入系统时调用，或当你认为需要清除系统中已存在的所有数据，对整个系统初始化时使用。

?;和 Cobase 建立连接：在控制窗口的主菜单中选中 File，在弹出的子菜单中选择 NetWork Share，和 Cobase 建立连接。

?;运行你的应用程序

6.Shutdown:在应用程序终止之后,选择 File/NormalShutdown,和 DBMS 断开连接。如果你应用程序非正常终止，那么选择 File/ImmediateShutdown。重新建立连接只要再选中 NetWork Share 即可，无须退出 Cobase。

7．退出系统：成功 Shutdown 之后，选择 File/Exit 退出 Cobase。

2.6.5 补充说明

本次数据库上机实习的主要目的是熟悉关系数据库的设计和实现的基本原理,掌握使用 EC 编程的基本方法。

Cobase 是数据库教研室正在研制开发的第一个国产关系数据库管理系统，它还存在着很多不完善的地方，如果给大家的使用带来了一定的困难，希望大家能够谅解。

把 Cobase 作为本次上机实习的教学软件，我们是经过研究考虑的。Cobase 虽然在很多方面不够完善和坚固，对有些功能不能给予很好的支持，但是对于一般数据库的基本操作还是可以完成的。对本次上机实习来说，Cobase 提供的功能是完全可以满足作业提出的所有要求的。Cobase EC 提供的基本功能和使用方法在 ESQL.DOC 中有详细说明，大家可以参照其中给出的例子程序编制自己的应用程序。在完成作业之余，欢迎大家对其中基本功能存在的不足提出宝贵意见。

另外有几个问题需要大家注意：

1．在该版本中，ISQL 和 EC 应用程序不能同时启动；

2．用 NOTEPAD 编辑的文件缺省以.txt 结尾，所以预编译器找不到，请大家用 VC 的编辑器编辑，并存为.ec 结尾的文件。

3．在 Cobase 初始化的时候需要申请足够的空间，所以请大家检查 Cobase 所在的盘是否有足够的空间。

4．在运行过程中如果出现了意外操作，建议大家先将 Cobase Immediate Shutdown，再重新启动。

5．COBASE 程序目录 COBASE 应放在某一分区的根目录下，并且用户的应用程序必须与 COBASE 目录在同一分区下。

6．COBASE ISQL界面中用到了一个MicroSoft的控件msflxgrd.ocx，如果目标机上没有的话，应把该文件拷贝到系统目录 /windows/system32/ 下。该文件包含在cobase.zip中。

这次上机实习题目的要求都很基本，鉴于 Cobase 所提供功能的局限，希望大家不要把过多的精力放在优化和界面等方面。希望大家上机顺利，如果有问题请多于我们联系。我们的 Email 地址是 linda@db.pku.edu.cn ， leonhard@db.pku.edu.cn, gaoxm@cis.pku.edu.cn .