IBM DB2 10.1
for Linux, UNIX, and Windows

开发嵌入式 SQL 应用程序

S151-1763-00

IBM DB2 10.1
for Linux, UNIX, and Windows

开发嵌入式 SQL 应用程序

S151-1763-00

注意

使用此信息及其支持的产品前，请先阅读第 195 页的附录 B，『声明』下的常规信息。

修订版声明

此文档包含 IBM 的所有权信息。它在许可协议中提供，且受版权法的保护。本出版物中包含的信息不包括对任何产品的保证，且提供的任何语句都不需要如此解释。

您可在线或通过当地的 IBM 代表处订购 IBM 出版物。

- 要在线订购出版物，请转至 IBM 出版物中心，网址为: http://www.ibm.com/shop/publications/order
- 要查找当地的 IBM 代表处，请转至 IBM 全球联系人目录，网址为: http://www.ibm.com/planetwide/

要从美国或加拿大的 DB2 市场和销售部订购 DB2 出版物，请致电 1-800-IBM-4YOU（426-4968）。

您发送信息给 IBM 后，即授予 IBM 非独占权限，IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

目录

第 1 章 嵌入式 SQL 简介

嵌入式 SQL 数据库应用程序连接到数据库并执行嵌入式 SQL 语句。嵌入式 SQL 语句嵌入在主语言应用程序中。嵌入式 SQL 数据库应用程序支持以静态或动态方式来执行 SQL 语句的嵌入。

您可以使用下列主编程语言为 DB2® 开发嵌入式 SQL 应用程序：C、C++、COBOL、FORTRAN 和 REXX。

注：建议您不要使用 FORTRAN 和 REXX 对嵌入式 SQL 的支持，此支持将保持处于 DB2 Universal Database™ V5.2 级别。

构建嵌入式 SQL 应用程序时，必须在编译和链接应用程序前执行两个必要的步骤。

- 使用 DB2 预编译器来准备包含嵌入式 SQL 语句的源文件。

  PREP (PRECOMPILE) 命令用于调用 DB2 预编译器，后者读取源代码，解析嵌入式 SQL 语句并将其转换为 DB2 运行时服务 API 调用，最后将输出写入经过修改的新源文件。预编译器将生成 SQL 语句的存取方案，这些存取方案以程序包形式一起存储在数据库中。
- 将应用程序中的语句与目标数据库绑定。

  缺省情况下，在预编译（PREP 命令）期间执行绑定。如果要延迟绑定（例如，以后运行 BIND 命令），那么必须在执行 PREP 时指定 BINDFILE 选项以便生成绑定文件。

在预编译并绑定嵌入式 SQL 应用程序之后，即可使用特定于主语言的开发工具对其进行编译和链接。

为了帮助开发嵌入式 SQL 应用程序，您可以参考嵌入式 SQL 模板 (C)。另外，还可以在 %DB2PATH%\SQLLIB\samples 目录中找到有效的嵌入式 SQL 样本应用程序的示例。

注：%DB2PATH% 是指 DB2 安装目录。

静态和动态 SQL

可以通过两种方式中的一种来执行 SQL 语句：静态方式或动态方式。

以静态方式执行的 SQL 语句

对于以静态方式执行的 SQL 语句而言，语法在预编译时已完全确定。您必须完全地指定 SQL 语句的结构，该语句才会被认为是静态语句。例如，语句中引用的列和表的名称在预编译时必须完全确定。唯一可以在运行时指定的信息是该语句所引用的任何主变量的值。但是，仍必须对主变量信息（例如数据类型）进行预编译。请在运行应用程序之前预编译、绑定和编译以静态方式执行的 SQL 语句。静态 SQL 最适合于统计信息不会大幅更改的数据库。

以动态方式执行的 SQL 语句

以动态方式执行的 SQL 语句由应用程序在运行时构建和执行。适合于动态 SQL 的情况的一个不错示例是，提示最终用户输入 SQL 语句关键部分（例如要搜索的表和列的名称）的交互式应用程序。

在主语言中嵌入 SQL 语句

结构化查询语言（SQL）是一种标准化语言，可用于处理数据库对象以及它们包含的数据。尽管不同主语言之间存在差别，但嵌入式 SQL 应用程序全都由设置和发出 SQL 语句所需的三个主要元素构成:

1. 用于声明主变量的声明节。不必在声明节中包括 SQLCA 结构的声明。
2. 应用程序的主体，此主体由 SQL 语句的设置和执行组成。
3. 用于落实或回滚 SQL 语句所作的更改的逻辑布置。

对于每种主语言，适用于所有语言的一般准则与特定于各种语言的规则之间存在差别。

C 和 C++ 应用程序中的嵌入式 SQL 语句

嵌入式 SQL C 和 C++ 应用程序由三个用于设置和发出 SQL 语句的主要元素组成。

- 用于声明主变量的声明节。不必在声明节中包括 SQLCA 结构的声明。
- 应用程序的主体，此主体由 SQL 语句的设置和执行组成。
- 用于落实或回滚 SQL 语句所作的更改的逻辑布置。

正确的 C 和 C++ 元素语法

语句初始化符
EXEC SQL

语句字符串
任何有效的 SQL 语句

语句终止符
分号（;）

例如，要在 C 应用程序中以静态方式发出 SQL 语句，需要在应用程序代码中包括 EXEC SQL 语句:

```
EXEC SQL SELECT col INTO :hostvar FROM table;
```

以下示例说明如何使用主变量 stmt1 以动态方式发出 SQL 语句:

```
strcpy(stmt1, "CREATE TABLE table1(col1 INTEGER)");
EXEC SQL EXECUTE IMMEDIATE :stmt1;
```

在 C 和 C++ 应用程序中执行嵌入式 SQL 语句时，下列准则和规则适用:

- 可以在 EXEC SQL 语句初始化符所在的行开始 SQL 语句字符串。
- 不要将 EXEC SQL 分割为多行。
- 必须使用 SQL 语句终止符。如果未使用此终止符，那么预编译器将一直处理到应用程序中的下一个终止符。这可能会导致不确定的错误。
- 可以在语句初始化符之前或者语句终止符之后放置 C 和 C++ 注释。
- 可以将多个 SQL 语句以及 C 或 C++ 语句放在同一行。例如:

  ```
  EXEC SQL OPEN c1; if (SQLCODE >= 0) EXEC SQL FETCH c1 INTO :hv;
  ```

- 括在引号中的字符串可以包含回车符、换行符和制表符。SQL 预编译器将保留这些字符不变。

- 不要使用 #include 语句来包括那些包含 SQL 语句的文件。系统将在编译模块前对 SQL 语句进行预编译。预编译器将忽略 #include 语句。而是，请使用 SQL INCLUDE 语句来导入包含文件。
- 允许在嵌入式 SQL 语句（以动态方式发出的语句除外）所包含的任何行中包括 SQL 注释。
  - SQL 注释的格式是，以两个短划线（--）开头，后跟由零个或零个以上字符组成的字符串，并终止于行尾。
  - 不要在 SQL 语句终止符之后放置 SQL 注释。编译器将把这些 SQL 注释解释为 C 或 C++ 语法，从而导致编译错误。
  - 可以在静态语句字符串中任何允许空格的位置使用 SQL 注释。
  - 在静态和动态嵌入式 SQL 语句中，允许使用 C 和 C++ 注释定界符 /* */。
  - 在静态 SQL 语句中，不允许使用 // 样式的 C++ 注释。
- 在 C 和 C++ 应用程序中，SQL 字符串文字和定界标识可以跨换行符延续。要实现此目的，请在要中断的行的末尾使用反斜杠（\）。例如，要从 staff 表中的 NAME 列中选择 NAME 列等于“Sanders”的数据，请执行类似于以下样本代码的操作:

  ```
  EXEC SQL SELECT "NA\
  ME" INTO :n FROM staff WHERE name='Sa\
  nders';
  ```

  在作为 SQL 语句传递到数据库管理器的字符串中，不包括任何新行字符（例如回车符和换行符）。
- 空格字符（例如行尾符和制表符）的替换如下所示:
  - 当它们出现在引号外部但在 SQL 语句内部时，行尾符和制表符被替换为单个空格。
  - 当它们出现在引号内时，行尾符将消失，条件是字符串对于 C 程序能够正确地延续。制表符不会被修改。

  注意，随平台不同，用作行尾符和制表符的实际字符也将有所变化。例如，基于 UNIX 和 Linux 的系统使用换行符。

FORTRAN 应用程序中的嵌入式 SQL 语句

FORTRAN 应用程序中的嵌入式 SQL 语句由下面这三个元素组成:

正确的 FORTRAN 元素语法

语句初始化符
EXEC SQL

语句字符串
任何以空格作为定界符的有效 SQL 语句

语句终止符
源代码行末尾。

源代码行末尾充当语句终止符。如果该行延续，那么语句终止符是最后一个延续行的末尾。

例如:

```
EXEC SQL SELECT COL INTO :hostvar FROM TABLE
```

下列规则适用于 FORTRAN 应用程序中的嵌入式 SQL 语句:

- 只在第 7 列与第 72 列之间编码 SQL 语句。
- 使用满行的 FORTRAN 注释或 SQL 字符，但不要在 SQL 语句中使用 FORTRAN 行尾注释“!”字符。此注释字符可以在其他位置（包括主变量声明）使用。
- 编码嵌入式 SQL 语句时，请使用空格作为定界符，即使 FORTRAN 语句不要求使用空格作为定界符亦如此。
- 在每个 FORTRAN 源代码行中，只指定一个 SQL 语句。正常的 FORTRAN 延续规则适用于需要多个源代码行的语句。请不要将 EXEC SQL 语句初始化符分割为多行。
- 允许在嵌入式 SQL 语句所包含的任何行中包括 SQL 注释。在以动态方式执行的语句中，不允许包括这些注释。SQL 注释的格式是，以两个短划线（--）开头，后跟由零个或零个以上字符组成的字符串，并终止于行尾。
- 在嵌入式 SQL 语句中的*几乎*任何位置，都允许包括 FORTRAN 注释。例外情况如下所示:
  - 不允许在 EXEC 与 SQL 之间包括注释。
  - 在以动态方式执行的语句中，不允许包括注释。
  - 在嵌入式 SQL 语句中，不支持在行尾使用 ！来编码 FORTRAN 注释这一扩展功能。
- 在 SQL 语句中指定实数常量时，请使用指数表示法。数据库管理器将 SQL 语句中包含小数点的数字字符串解释为十进制常量，而不是解释为实数常量。
- 在第一个可执行的 FORTRAN 语句之前的 SQL 语句中，语句号无效。如果 SQL 语句有相关联的语句号，那么预编译器将在该 SQL 语句紧前面生成带有标号的 CONTINUE 语句。
- 在 SQL 语句中引用主变量时，请完全按照声明的方式来使用主变量。
- 空格字符（例如行尾符和制表符）的替换如下所示:
  - 当它们出现在引号外部但在 SQL 语句内部时，行尾符和制表符被替换为单个空格。
  - 当它们出现在引号内时，行尾符将消失，条件是字符串对于 FORTRAN 程序能够正确地延续。制表符不会被修改。

  注意，随平台不同，用作行尾符和制表符的实际字符也将有所变化。例如，基于 Windows 的平台使用回车符/换行符作为行尾符，而基于 UNIX 和 Linux 的平台只使用换行符。

COBOL 应用程序中的嵌入式 SQL 语句

COBOL 应用程序中的嵌入式 SQL 语句由下面这三个元素组成:

正确的 COBOL 元素语法

语句初始化符
EXEC SQL

语句字符串
任何有效的 SQL 语句

语句终止符
END-EXEC.

例如:

```
EXEC SQL SELECT col INTO :hostvar FROM table END-EXEC.
```

下列规则适用于 COBOL 应用程序中的嵌入式 SQL 语句:

- 必须将可执行的 SQL 语句放入 PROCEDURE DIVISION 节。就像 COBOL 语句一样，可以在 SQL 语句前面指定段名称。
- SQL 语句可以开始于区域 A（第 8 至 11 列）或区域 B（第 12 至 72 列）。
- 请使用语句初始化符 EXEC SQL 来开始每个 SQL 语句，并使用语句终止符 END-EXEC 来结束该语句。SQL 预编译器会将每个 SQL 语句作为注释包括在经过修改的源文件中。
- 必须使用 SQL 语句终止符。如果未使用此终止符，那么预编译器将一直处理到应用程序中的下一个终止符。这可能会导致不确定的错误。
- 允许在嵌入式 SQL 语句所包含的任何行中包括 SQL 注释。在以动态方式执行的语句中，不允许包括这些注释。SQL 注释的格式是，以两个短划线（--）开头，后跟由零个或零个以上字符组成的字符串，并终止于行尾。请不要在 SQL 语句终止符之后放置 SQL 注释，这是因为，它们会由于似乎是 COBOL 语言的组成部分而导致编译错误。
- 在大多数位置，允许包括 COBOL 注释。例外情况如下所示:
  - 不允许在 EXEC 与 SQL 之间包括注释。
  - 在以动态方式执行的语句中，不允许包括注释。
- SQL 语句与 COBOL 语言遵循相同的行延续规则。但是，请不要使 EXEC SQL 语句初始化符跨行。
- 不要使用 COBOL COPY 语句来包括那些包含 SQL 语句的文件。系统将在编译模块前对 SQL 语句进行预编译。预编译器将忽略 COBOL COPY 语句。而是，请使用 SQL INCLUDE 语句来导入包含文件。
- 要使字符串常量在下一行延续，延续行的第 7 列必须包含“-”，第 12 列或之后的列必须包含字符串定界符。
- SQL 算术运算符必须由空格定界。
- 空格字符（例如行尾符和制表符）的替换如下所示:
  - 当它们出现在引号外部但在 SQL 语句内部时，行尾符和制表符被替换为单个空格。
  - 当它们出现在引号内时，行尾符将消失，条件是字符串对于 COBOL 程序能够正确地延续。制表符不会被修改。

  注意，随平台不同，用作行尾符和制表符的实际字符也将有所变化。例如，基于 Windows 的平台使用回车符/换行符作为行尾符，而基于 UNIX 和 Linux 的系统只使用换行符。

REXX 应用程序中的嵌入式 SQL 语句

REXX 应用程序使用 API，这使它们能够使用 数据库管理器 API 和 SQL 提供的大部分功能。与使用编译型语言编写的应用程序不同，REXX 应用程序不需要进行预编译。而是，动态 SQL 处理程序处理所有 SQL 语句。通过将 REXX 与这些可调用的 API 相结合，您可以访问大多数 数据库管理器 功能。虽然 REXX 并不使用嵌入式 SQL 来直接支持某些 API，但可以从 REXX 应用程序中使用 DB2 命令行处理器来访问这些 API。

由于 REXX 是解释型语言，您会发现，与编译型主语言相比，开发和调试 REXX 应用程序原型更为容易。虽然使用 REXX 编码的数据库应用程序的性能不如使用编译型语言的数据库应用程序，但它们使您不必进行预编译、编译和链接或者使用其他软件就能创建数据库应用程序。

使用 SQLEXEC 例程来处理所有 SQL 语句。SQLEXEC 例程的字符串自变量由以下元素组成:

- SQL 关键字
- 预先声明的标识
- 语句主变量

通过将有效的 SQL 语句传递给 SQLEXEC 例程来发出每个请求。请使用以下语法:

```
CALL SQLEXEC '语句'
```

SQL 语句可以跨多行。语句的每一部分都应该括在单引号中，并且必须用逗号对附加的语句文本进行定界，如下所示:

```
CALL SQLEXEC 'SQL 文本',
             '附加文本',
                  .
                  .
                  .
             '最终文本'
```

以下代码是在 REXX 中嵌入 SQL 语句的示例:

```
statement = "UPDATE STAFF SET JOB = 'Clerk' WHERE JOB = 'Mgr'"
CALL SQLEXEC 'EXECUTE IMMEDIATE :statement'
IF ( SQLCA.SQLCODE < 0) THEN
   SAY 'Update Error:  SQLCODE = ' SQLCA.SQLCODE
```

在此示例中，将检查 SQLCA 结构的 SQLCODE 字段以确定更新是否成功。

下列规则适用于嵌入式 SQL 语句: 在 REXX 应用程序中

- 可以将下列 SQL 语句直接传递到 SQLEXEC 例程:
  - CALL
  - CLOSE
  - COMMIT
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - DECLARE
  - DESCRIBE
  - DISCONNECT
  - EXECUTE
  - EXECUTE IMMEDIATE
  - FETCH
  - FREE LOCATOR
  - OPEN
  - PREPARE
  - RELEASE
  - ROLLBACK
  - SET CONNECTION

对于其他 SQL 语句而言，必须通过将 EXECUTE IMMEDIATE 或者 PREPARE 和 EXECUTE 语句与 SQLEXEC 例程配合使用来动态地进行处理。

- 不能在 REXX 中的 CONNECT 和 SET CONNECTION 语句中使用主变量。
- 请按如下方式预定义游标名和语句名称:

  c1 到 c100

  游标名，*c1* 到 *c50* 的范围用于声明时未指定 WITH HOLD 选项的游标，*c51* 到 *c100* 用于声明时指定了 WITH HOLD 选项的游标。

  游标名标识用于 DECLARE、OPEN、FETCH 和 CLOSE 语句。它标识 SQL 请求中使用的游标。

  s1 到 s100

  语句名称，范围是 *s1* 到 *s100*。

  语句名称标识与 DECLARE、DESCRIBE、PREPARE 和 EXECUTE 语句配合使用。

  预先声明的标识必须用于游标名和语句名称。不允许用于其他名称。

- 声明游标时，游标名和语句名称在 DECLARE 语句中应该相互对应。例如，如果将 *c1* 用作游标名，那么必须使用 *s1* 作为语句名称。
- 请不要在 SQL 语句中使用注释。

注: REXX 不支持多线程数据库访问。

嵌入式 SQL 应用程序所支持的开发软件

DB2 数据库系统支持下列操作系统中用于嵌入式 SQL 应用程序的编译器、解释器和相关开发软件:

- AIX®
- HP-UX
- Linux
- Solaris
- Windows

可以根据嵌入式 SQL 源代码来构建 32 位和 64 位嵌入式 SQL 应用程序。

下列主语言要求使用特定的编译器来开发嵌入式 SQL 应用程序:

- C
- C++
- COBOL
- Fortran
- REXX

设置嵌入式 SQL 开发环境

在可以开始构建嵌入式 SQL 应用程序之前，请安装用于开发应用程序的主语言所支持的编译器并设置嵌入式 SQL 环境。

开始之前

- 在受支持的平台上安装 DB2 数据服务器
- 安装 DB2 客户机
- 安装受支持的嵌入式 SQL 应用程序开发软件 - 请参阅*数据库应用程序开发入门*中的『安装受支持的嵌入式 SQL 应用程序开发软件』

关于此任务

授权用户发出 **PREP** 命令和 **BIND** 命令。

要验证是否已正确地设置嵌入式 SQL 应用程序开发环境，请尝试构建和运行以下主题中的嵌入式 SQL 应用程序模板: 嵌入式 SQL 应用程序模板（C）。

第 2 章 设计嵌入式 SQL 应用程序

设计嵌入式 SQL 应用程序时，必须使用以静态方式或动态方式执行的 SQL 语句。静态 SQL 语句有两种风格：不包含主变量的语句（主要用于初始化和简单 SQL 示例）和使用了主变量的语句。动态 SQL 语句也有两种风格：它们可以不包含参数标记（CLP 之类的接口的典型情况），也可以包含参数标记（这使应用程序更为灵活）。

选择是使用以静态方式执行的语句还是以动态方式执行的语句取决于诸多因素，其中包括：嵌入式 SQL 应用程序的可移植性、性能和限制。

嵌入式 SQL 的授权注意事项

*授权*允许用户或组执行常规任务，例如连接到数据库、创建表或管理系统。*特权*使用户或组有权以指定方式访问一个特定的数据库对象。DB2® 使用一组特权为您在其中存储的信息提供保护。

大多数 SQL 语句要求对该语句所利用的数据库对象具有某种类型的特权。大多数 API 调用通常不要求对该调用所利用的数据库对象具有任何特权，但是，许多 API 要求您拥有必需的权限才能将其启动。您可以使用 DB2 API 从应用程序中执行 DB2 管理功能。例如，要在不需要绑定文件的情况下重新创建数据库中存储的程序包，可以使用 `sqlarbnd`（或 REBIND）API。

组提供了一种便利的方法为一组用户执行授权，而不必逐个地为每个用户授予或撤销特权。执行动态 SQL 语句时，将考虑组成员资格，但执行静态 SQL 语句时，情况并非如此。执行静态 SQL 语句时，考虑 `PUBLIC` 特权。例如，假定嵌入式 SQL 存储过程包含以静态方式绑定的 SQL 查询，并且，这些查询对名为 `STAFF` 的表执行。如果您尝试使用 `CREATE PROCEDURE` 语句来构建此过程，并且您的帐户属于对 `STAFF` 表具有 SELECT 特权的组，那么 `CREATE` 语句将失败并发生 SQL0551N 错误。要使 `CREATE` 语句正常工作，您的帐户直接需要 `STAFF` 表的 SELECT 特权。

设计应用程序时，请考虑用户运行该应用程序时所需的特权。用户所需的特权取决于下列各项：

- 该应用程序是使用动态 SQL（包括 JDBC 和 CLI）还是使用静态 SQL。有关发出语句时所需的特权的信息，请参阅该语句的描述。
- 该应用程序所使用的 API。有关 API 调用所需的特权和权限的信息，请参阅该 API 的描述。

组提供了一种便利的方法为一组用户执行授权，而不必逐个地为每个用户授予或撤销特权。通常，动态 SQL 语句考虑组成员资格，但静态 SQL 语句并非如此。将特权授予 PUBLIC 是这种一般情况的例外：处理静态 SQL 语句时，将考虑这些特权。

假定有两个用户 PAYROLL 和 BUDGET，他们需要对 STAFF 表执行查询。PAYROLL 负责向公司职员支付薪水，因此它需要在发放薪水支票时发出各种 SELECT 语句。PAYROLL 必须能够访问每个职员的薪水。BUDGET 负责确定支付薪水所需的金额。但是，BUDGET 不应能够查看任何特定职员的薪水。

由于 PAYROLL 发出许多不同的 SELECT 语句，因此，您为 PAYROLL 设计的应用程序可能会大量使用动态 SQL。动态 SQL 将要求 PAYROLL 对 STAFF 表具有 SELECT 特权。因为 PAYROLL 需要对表进行全面访问，所以此项要求不是问题。

但是，BUDGET 不应有访问每位职员薪水的权限。这意味着，您不应将 STAFF 表的 SELECT 特权授予 BUDGET。由于 BUDGET 需要访问 STAFF 表中所有薪水的总计，因此，您可以构建一个静态 SQL 应用程序来执行 SELECT SUM(SALARY) FROM STAFF，绑定该应用程序并将该应用程序的程序包的 EXECUTE 特权授予 BUDGET。这将使 BUDGET 能够获取必需的信息，而不会将该信息透露给不应查看该信息的 BUDGET。

在嵌入式 SQL 应用程序中执行静态和动态 SQL 语句

在嵌入式 SQL 应用程序中，既支持执行静态 SQL 语句也支持执行动态 SQL 语句。决定是以静态还是动态方式执行 SQL 语句要求了解程序包、如何在运行时发出 SQL 语句、主变量、参数标记以及这些内容与应用程序性能的关系。

嵌入式 SQL 程序中的静态 SQL

下面是以静态方式发出的 C 语句的示例:

```
/* 使用静态 SQL 从表中选择值以将其赋予主变量，然后进行打印 */
EXEC SQL SELECT id, name, dept, salary INTO :id, :name, :dept, :salary
           FROM staff WHERE id = 310;
```

嵌入式 SQL 程序中的动态 SQL

下面是以动态方式发出的 C 语句的示例:

```
/* 使用动态 SQL 来更新表中的列 */
strcpy(hostVarStmtDyn, "UPDATE staff SET salary = salary + 1000  WHERE dept = ?");
EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
EXEC SQL EXECUTE StmtDyn USING :dept;
```

嵌入式 SQL 动态语句

动态 SQL 语句接受字符串主变量和语句名作为自变量。主变量包含要以动态方式进行处理的 SQL 语句（文本形式）。预编译应用程序时，不会对语句文本进行处理。实际上，语句文本在应用程序预编译期间不必存在。而是，该 SQL 语句被视为主变量以便进行预编译，该变量将在应用程序执行期间被引用。

要将包含 SQL 文本的主变量转换为可执行形式，需要使用动态 SQL 支持语句。并且，动态 SQL 支持语句通过引用语句名对主变量执行操作。这些支持语句包括:

EXECUTE IMMEDIATE

准备并执行未使用任何主变量的语句。此语句可用于替代 PREPARE 和 EXECUTE 语句。

例如，请考虑以下 C 语句:

```
strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
   FROM EMP_ACT WHERE ACTNO >= 100");
EXEC SQL  EXECUTE IMMEDIATE :qstring;
```

PREPARE

将字符串形式的 SQL 语句转换为该语句的可执行形式，指定语句名，并将关于该语句的信息放入 SQLDA 结构（可选）。

EXECUTE

执行先前准备的 SQL 语句。可以在一个连接中反复执行该语句。

DESCRIBE

将关于已准备的语句的信息放入 SQLDA。

例如，请考虑以下 C 语句:

```
strcpy(hostVarStmt, "DELETE FROM org WHERE deptnumb = 15");
EXEC SQL PREPARE Stmt FROM :hostVarStmt;
EXEC SQL DESCRIBE Stmt INTO :sqlda;
EXEC SQL EXECUTE Stmt;
```

注: 动态 SQL 语句的内容与静态 SQL 语句遵循相同的语法，但存在下列例外情况:

- 此语句不能以 EXEC SQL 开头。
- 此语句不能以语句终止符结尾。在这方面有一个例外情况，即，CREATE TRIGGER 语句可以包含分号（;）。

确定何时在嵌入式 SQL 应用程序中以静态或动态方式执行 SQL 语句

在确定是以静态还是动态方式从嵌入式 SQL 应用程序中发出 SQL 语句之前，必须考虑诸多事项。下表列出了与使用静态和动态 SQL 语句关联的考虑事项。

注: 这些只是一般性建议。实际的选择取决于应用程序的要求、预期用途和工作环境。不确定时，最好的做法是先将语句作为静态 SQL 进行测试，然后再作为动态 SQL 进行测试并比较差别。

表 1. 比较静态 SQL 与动态 SQL

考虑事项	可能的最佳选项
SQL 语句所查询或操作的数据的统一性	
• 统一的数据分布	• 静态
• 略微不统一	• 任何一项
• 高度不统一的分布	• 动态
查询中的范围谓词的数量	
• 很少	• 静态
• 有一些	• 任何一项
• 很多	• 动态
反复执行 SQL 语句的可能性	
• 运行许多次（10 次或更多次）	• 任何一项
• 运行几次（不足 10 次）	• 任何一项
• 运行一次	• 静态
查询的性质	
• 随机	• 动态
• 永久	• 任何一项
SQL 语句的类型（DML/DDL/DCL）	
• 事务处理（仅限于 DML）	• 任何一项
• 混合（DML 和 DDL - DDL 影响程序包）	• 动态
• 混合（DML 和 DDL - DDL 不影响程序包）	• 任何一项

表 1. 比较静态 SQL 与动态 SQL (续)

考虑事项	可能的最佳选项
发出 RUNSTATS 命令的频率	
• 不经常	• 静态
• 正常频率	• 任何一项
• 频率非常高	• 动态

SQL 语句在运行前始终被编译。区别在于，动态 SQL 语句在运行时被编译，因此应用程序可能会由于在应用程序运行时与编译每个动态语句所使用的资源增加而变慢，而对于静态 SQL 而言，将在单一的初始编译阶段进行编译。

在混合环境中，静态 SQL 与动态 SQL 之间的选择还必须考虑使程序包失效的频率。如果 DDL 确实要使程序包失效，那么动态 SQL 的效率更高，这是因为，只有所发出的那些查询才会在它们下次使用时被重新编译。其他查询不会被重新编译。对于静态 SQL 而言，在使程序包失效后，将重新绑定整个程序包。

无论使用静态 SQL 还是动态 SQL，如果不起作用，总会有多次机会。例如，在主要引用以动态方式发出的 SQL 语句的应用程序中，可能有一个语句更适合作为静态 SQL 发出。在此情况下，为了使编码一致，可能最好也以动态方式发出该语句。注意，上表中的考虑事项大致按重要程度排序。

不要假定静态版本 SQL 语句始终快于同等的动态语句。在某些情况下，静态 SQL 更快，这是因为准备动态语句需要使用资源。在其他情况下，发出以动态方式准备的同一语句时的速度更快，这是因为优化器可以利用当前数据库统计信息，而不是利用先前执行绑定时获得的数据库统计信息。注意，如果事务只需几秒即可完成，那么静态 SQL 通常更快。要选择所使用的方法，您应该对这两种形式的绑定进行测试。

注: 静态 SQL 和动态 SQL 都有两种类型，即使用主变量的语句和不使用主变量的语句。这些类型是:

1. 不包含主变量的静态 SQL 语句

   这是一种比较罕见的情况，只有下列内容才会出现这种情况:

   - 初始化代码
   - 简单 SQL 语句

   从性能角度看，不包含主变量的简单 SQL 语句的性能较佳，其原因在于没有运行时性能增加，并且能够充分实现 DB2 优化器的功能。

2. 包含主变量的静态 SQL

   利用主变量的静态 SQL 语句被视为传统样式的 DB2 应用程序。静态 SQL 语句能够避免与 PREPARE 关联的运行时资源用途以及语句编译期间获取的目录锁定。不幸的是，由于优化器不知道完整的 SQL 语句，因此无法充分使用优化器的功能。对于高度不统一的数据分布，存在特定的问题。

3. 不包含参数标记的动态 SQL

   对于通常用于执行随需应变查询的 CLP 之类的接口，这种情况很典型。在 CLP 中，只能动态地发出 SQL 语句。

4. 包含参数标记的动态 SQL

动态 SQL 语句的关键优势是，由于存在参数标记，因此能够在语句（通常是选择或插入）的反复执行之间分摊语句准备成本。对于所有重复性动态 SQL 应用程序，这种分摊都有效。不幸的是，就像包含主变量的静态 SQL 一样，由于无法获得全面的信息，因此 DB2 优化器的某些部件将无法工作。

建议您使用包含主变量的静态 SQL 或者不包含参数标记的动态 SQL 作为最高效的选项。

嵌入式 SQL 应用程序的性能

开发数据库应用程序时，性能是重要的考虑因素。由于嵌入式 SQL 应用程序支持执行静态 SQL 语句以及混合执行静态 SQL 语句和动态 SQL 语句，因此性能较佳。静态 SQL 语句的编译方式决定了，开发者或数据库管理员必须执行一些步骤以确保嵌入式 SQL 应用程序的性能不会随时间推移而下降。

下列因素可能会影响嵌入式 SQL 应用程序的性能:

- 数据库模式随时间变化而更改
- 表的基数（表中的行数）随时间变化而更改
- 与 SQL 语句绑定的主变量值更改

嵌入式 SQL 应用程序的性能受这些因素影响的原因是，程序包在数据库具有一组特定特征时创建一次。程序包运行时访问方案的创建已考虑这些特征，这些访问方案定义数据库管理器如何最高效率地发出 SQL 语句。随着时间推移，数据库模式和数据可能会更改，从而使得运行时访问方案变得欠佳。这可能会导致应用程序性能下降。

因此，定期刷新所使用的信息以确保对程序包运行时存取方案进行良好的维护至关重要。

RUNSTATS 命令用于收集表和索引的当前统计信息，如果上次发出 RUNSTATS 命令后已执行大量更新活动或者已创建新索引，那么尤其应该运行此命令。这将为优化器提供用于确定最佳访问方案的最准确信息。

可以通过多种方法来提高嵌入式 SQL 应用程序的性能:

- 运行 RUNSTATS 命令以更新数据库统计信息。
- 将应用程序程序包与数据库重新绑定，以便根据已更新的统计信息来重新生成运行时访问方案，数据库将使用那些访问方案以物理方式检索磁盘上的数据。
- 在静态和动态程序中使用 REOPT 绑定选项。

对嵌入式 SQL 应用程序的 32 位和 64 位支持

在 32 位和 64 位平台上，都可以构建嵌入式 SQL 应用程序。但是，存在不同的构建和运行注意事项。构建脚本包含用于确定位宽度的检查。如果检测到位宽度为 64 位，那么将设置一组额外的开关以进行必要的更改。

DB2 数据库系统在本节稍后列示的 32 位和 64 位版本操作系统上受支持。在大多数情况下，在这些操作系统上的 32 位和 64 位环境中构建嵌入式 SQL 应用程序时，存在一些差别。

- AIX

- HP-UX
- Linux
- Solaris
- Windows

在 DB2 版本 9 中受支持的 32 位实例只有:

- 基于 x86 的 Linux
- 基于 x86 的 Windows
- 基于 x64 的 Windows（使用用于基于 x86 的 Windows 的 DB2 安装映像时）

在 DB2 版本 9 中受支持的 64 位实例只有:

- AIX
- Sun
- HP IPF
- 基于 x64 的 Linux
- 基于 POWER® 的 Linux
- 基于 System z® 的 Linux
- 基于 x64 的 Windows（使用用于 x64 的 Windows 安装映像时）
- 基于 IPF 的 Windows
- 基于 IPF 的 Linux

DB2 数据库系统支持在除 Linux IA64 和 Linux System z 以外的所有受支持 64 位操作系统环境中运行 32 位应用程序和例程。

对于每种主语言，在 32 位和/或 64 位平台中可能最好使用主变量。请参阅每种编程语言的各种数据类型。

有关嵌入式 SQL 应用程序的限制

每种受支持的主语言都有自己的一组限制和规范。C/C++ 使用称为“三字母词”的三字符序列来克服某些特殊字符的显示限制。COBOL 有一组规则来帮助使用面向对象的 COBOL 应用程序。FORTRAN 有一些将会影响到预编译过程的有趣领域，而 REXX 在某些领域（例如语言支持）有所限制。

使用 C 和 C++ 进行嵌入式 SQL 应用程序编程时有关字符集的限制

并非所有键盘都提供了 C 或 C++ 字符集中的某些字符。通过使用称为*三字母词*的三字符序列，可以将这些字符输入到 C 或 C++ 源程序中。在 SQL 语句中，不识别三字母词。预编译器识别主变量声明中的下列三字母词:

三字母词	定义
??(	左方括号“[”
??)	右方括号“]”
??<	左花括号“{”

??> 右花括号“}”

下列三字母词可以在 C 或 C++ 源程序中的其他位置出现:

三字母词
定义

??= 散列标记“#”

??/ 反斜杠“\”

??' 插入标记“^”

??! 竖线“|”

??– 颚化音符号“~”

使用 COBOL 进行嵌入式 SQL 应用程序编程时的限制

COBOL 应用程序中 API 调用的限制。

COBOL 应用程序中 API 调用的限制包括:

- 必须使用 `USAGE COMP-5` 子句来声明所有用作 API 调用中的值参数的整数变量。

在面向对象的 COBOL 程序中:

- 只能在编译单元中的第一个程序或类中使用 SQL 语句。存在此限制的原因是，预编译器将在它所遇到的第一个工作存储节中插入临时工作数据。
- 每个包含 SQL 语句的类都必须要有类级别的工作存储节，即使为空亦如此。此节用于存储预编译器所生成的数据定义。

使用 FORTRAN 进行嵌入式 SQL 应用程序编程时的限制

嵌入式 SQL 对 FORTRAN 的支持已在 DB2 V5 中定型，我们未计划将来对其进行增强。例如，FORTRAN 预编译器无法处理长度超过 18 个字节的 SQL 对象标识（例如表名）。要使用 DB2 数据库系统在 DB2 V5 之后引入的功能，例如长度为 19 到 128 个字节的表名，您必须使用除 FORTRAN 以外的语言来编写应用程序。

Windows 或 Linux 环境中的 DB2 实例不支持 FORTRAN 数据库应用程序开发。

FORTRAN 不支持多线程数据库访问。

某些 FORTRAN 编译器将列 1 包含“D”或“d”的行视为条件行。可以对这些行进行编译以便进行调试，也可以将其视为注释。预编译器始终将列 1 包含“D”或“d”的行视为注释。

某些 API 参数要求调用变量包含地址而不是值。数据库管理器提供了 GET ADDRESS、DEREFERENCE ADDRESS 和 COPY MEMORY API，这简化了您提供这些参数的能力。

下列各项将影响预编译过程:

- 预编译器只允许连续行的列 1-5 包含数字、空格和跳进字符。
- 在 `.sqf` 源文件中，不支持 Hollerith 常量。

使用 REXX 进行嵌入式 SQL 应用程序编程时的限制

有关 REXX 应用程序中嵌入式 SQL 的限制如下所示:

- 嵌入式 SQL 对 REXX 的支持已在 DB2 通用数据库版本 5 中定型，我们未计划将来对其进行增强。例如，REXX 无法处理长度超过 18 个字节的 SQL 对象标识（例如表名）。要使用 DB2 数据库系统在版本 5 之后引入的功能，例如长度为 19 到 128 个字节的表名，您必须使用除 REXX 以外的语言来编写应用程序。
- 在 REXX/SQL 中，不支持复合 SQL。
- REXX 不支持静态 SQL。
- 在日语或繁体中文 EUC 环境中，不支持 REXX 应用程序。

有关使用 XML 和 XQuery 开发嵌入式 SQL 应用程序的建议

在嵌入式 SQL 应用程序中使用 XML 和 XQuery 时，下列建议和限制适用。

- 应用程序必须以序列化字符串格式来访问所有 XML 数据。
  - 必须以序列化字符串格式来表示所有数据，包括数字数据和日期时间数据。
- 外部化的 XML 数据限长 2 GB。
- 所有包含 XML 数据的游标都是非分块游标（每个提取操作都将生成数据库服务器请求）。
- 每当字符主变量包含序列化 XML 数据时，都假定使用应用程序代码页作为数据的编码，并且该代码页必须与数据中存在的任何内部编码匹配。
- 您必须指定 LOB 数据类型作为 XML 主变量的基本类型。
- 以下建议和限制适用于静态 SQL:
  - 不能在 `SELECT INTO` 操作中使用字符主变量和二进制主变量来检索 XML 值。
  - 在期望将 XML 数据类型用于输入时，使用 CHAR、VARCHAR、CLOB 和 BLOB 主变量将导致执行具有缺省空格处理特征（`"STRIP WHITESPACE"`）的 `XMLPARSE` 操作。任何其他非 XML 主变量类型都将被拒绝。
  - 不支持静态 XQuery 表达式；尝试预编译 XQuery 表达式将失败并发生错误。只能通过 XMLQUERY 函数来发出 XQuery 表达式。
- 通过在表达式开头添加字符串“XQUERY”，可以动态地发出 XQuery 表达式。

嵌入式 SQL 应用程序中的并发事务和多线程数据库访问

某些操作系统的其中一项功能是，能够在单一进程中运行多个执行线程。多线程使应用程序能够处理异步事件，并使您能够更方便地创建事件驱动应用程序，而不必求助于轮询方案。

以下信息描述 DB2 数据库管理器如何使用多个线程，并列示了您应该牢记的一些设计准则。

如果您不熟悉与多线程应用程序的开发相关的术语（例如临界区和信标），请查阅操作系统的编程文档。

DB2 嵌入式 SQL 应用程序可以使用*上下文*从多个线程中执行 SQL 语句。上下文是一个环境，应用程序从该环境中运行所有 SQL 语句和 API 调用。所有连接、工作单元和其他数据库资源都与特定上下文相关联。每个上下文都与应用程序中的一个或多个线

程相关联。只有在 C 和 C++ 中，才支持开发包含线程安全代码的多线程嵌入式 SQL 应用程序。您可以编写自己的预编译器，它除了允许由语言提供的功能以外，还允许进行并发多线程数据库访问。

对于上下文中的每个可执行 SQL 语句，第一个运行时服务调用将始终尝试获取锁存器。如果此操作成功，那么它将继续进行处理。如果此操作由于同一上下文的另一个线程中的 SQL 语句已占用锁存器而失败，那么该调用将在发出信标时被阻塞并一直持续到该信标被发布为止，发布后，该调用将获取锁存器并继续进行处理。该锁存器将被占用到该 SQL 语句处理完毕，接着，它被上一个为该特定 SQL 语句生成的运行时服务调用释放。

最终结果是，上下文中的每个 SQL 语句都作为原子单元执行，尽管其他线程也可能尝试同时执行 SQL 语句。此操作确保内部数据结构不会同时被不同线程改变。API 也使用运行时服务所使用的锁存器；因此，API 与每个上下文中的运行时服务例程具有相同的限制。

可以在进程中的线程之间交换上下文，但不能在进程之间交换上下文。多上下文的其中一种用途是支持并发事务。

在对 DB2 数据库执行的线程应用程序的缺省实现中，数据库访问序列化由数据库 API 强制实施。如果一个线程执行数据库调用，那么其他线程进行的调用将被阻塞到第一个调用完成为止，即使后续调用访问与第一个调用无关的数据库对象亦如此。另外，进程中的所有线程共享落实作用域。只能通过独立的进程或者通过使用本主题中描述的 API 来实现真正的数据库并发访问。

DB2 数据库系统提供了可用于分配和处理独立环境（上下文）的 API，以便支持使用数据库 API 和嵌入式 SQL。每个上下文都是独立的实体，任何使用一个上下文的连接都与所有其他上下文无关（因此，与进程中的所有其他连接无关）。要在上下文中完成工作，必须先使上下文与线程相关联。在进行数据库 API 调用或者使用嵌入式 SQL 时，线程必须始终有上下文。

缺省情况下，所有 DB2 数据库系统应用程序都是多线程应用程序并能够使用多个上下文。您可以通过下列 DB2 API 来使用多个上下文。具体而言，应用程序可以为线程创建上下文、连接到每个线程的独立上下文或者与其拆离以及在线程之间传递上下文。如果应用程序未调用*任何*这些 API，那么 DB2 将自动地为应用程序管理多个上下文:

- `sqleAttachToCtx` - 连接到上下文
- `sqleBeginCtx` - 创建应用程序上下文并连接到该上下文
- `sqleDetachFromCtx` - 与上下文拆离
- `sqleEndCtx` - 拆离应用程序上下文并将其销毁
- `sqleGetCurrentCtx` - 获取当前上下文
- `sqleInterruptCtx` - 中断上下文

这些 API 在不支持应用程序线程化的平台上无效（即，它们是空操作）。

上下文在连接的持续期间不必与给定线程相关联。一个线程可以连接到上下文、连接到数据库并接着与该上下文拆离，然后，第二个线程可以连接到该上下文并使用现有的数据库连接继续执行工作。可以在进程中的线程之间传递上下文，但不能在进程之间进行传递。

即使使用新的 API，下列 API 也继续进行序列化:

- sqlabndx - 绑定
- sqlaprep - 预编译程序
- sqluexpr - 导出
- db2Import 和 sqluimpr - 导入

注:

1. 在支持多线程的平台上，CLI 自动使用多个上下文来实现线程安全的并发数据库访问。有需要时，用户可以显式禁用此功能，尽管建议不要这样做。
2. 缺省情况下，对于每个进程，AIX 不允许 32 位应用程序连接到 11 个以上的共享内存段，其中，最多 10 个共享内存段可用于 DB2 数据库连接。

   达到此限制时，DB2 数据库系统将对 SQL CONNECT 返回 SQLCODE -1224。如果本地用户通过 TP 监视器（TCP/IP）来运行两阶段落实，那么 DB2 Connect™ 也有“连接数不能超过 10 个”这一限制。

   您可以使用 AIX 环境变量 **EXTSHM** 来增加进程可以连接的共享内存段的最大数目。

   要将 **EXTSHM** 与 DB2 数据库系统配合使用，请按照以下列出的步骤操作:

   在客户机会话中:

   ```
   export EXTSHM=ON
   ```

   在启动 DB2 服务器时:

   ```
   export EXTSHM=ON
   db2set DB2ENVLIST=EXTSHM
   db2start
   ```

   在分区数据库环境上，还应向 `userprofile` 或 `usercshrc` 文件添加以下行:

   ```
   EXTSHM=ON
   export EXTSHM
   ```

   另一种方法是，将本地数据库或 DB2 Connect 移至另一台机器并以远程方式对其进行访问，或者通过将本地数据库或 DB2 Connect 数据库编目为具有本地机器 TCP/IP 地址的远程节点来借助 TCP/IP 回送功能对其进行访问。

有关使用多个线程的建议

从多线程应用程序中访问数据库时，请遵循下列准则:

将数据结构的改变序列化。

应用程序必须确保，在一个线程中处理 SQL 语句或数据库管理器例程时，该 SQL 语句或数据库管理器例程所使用的用户定义的数据结构不会被另一个线程改变。例如，当 SQLDA 正被一个线程中的 SQL 语句使用时，不要允许另一个线程重新分配该 SQLDA。

考虑使用不同的数据结构。

较为简单的方法是，对每个线程指定它自己的用户定义的数据结构，从而避免必须将数据结构的使用序列化。对于不仅由每个可执行 SQL 语句使用而是还由所有数据库管理器例程使用的 SQLCA 而言，尤其应该遵循此准则。对于 SQLCA，可以通过另外三种方法来避免此问题:

- 使用 EXEC SQL INCLUDE SQLCA，但在除第一个线程以外的任何其他线程所使用的任何例程开头添加 struct sqlca sqlca。
- 在每个包含 SQL 的例程内放置 EXEC SQL INCLUDE SQLCA，而不是将其置于全局作用域中。
- 将 EXEC SQL INCLUDE SQLCA 替换为 #include "sqlca.h"，然后在任何使用 SQL 的例程开头添加“struct sqlca sqlca”。

多线程 UNIX 应用程序的代码页以及国家或地区代码注意事项

本节仅适用于 C 和 C++ 嵌入式 SQL 应用程序。

在 AIX、Solaris 和 HP-UX 上，对用于数据库连接的代码页以及国家/地区或区域代码执行运行时查询时，使用的函数现在具有线程安全性。但是，这些函数可能会在使用了大量并发数据库连接的多线程应用程序中引起一些锁定争用并导致性能下降。

可以使用 *DB2_FORCE_NLS_CACHE* 环境变量来消除多线程应用程序中发生锁定争用的机会。*DB2_FORCE_NLS_CACHE* 设置为 TRUE 时，代码页以及国家或地区代码信息将在第一次被线程访问时保存下来。从那时起，高速缓存的信息将用于任何其他请求获取此信息的线程。通过保存此信息，可以消除锁定争用，在某些情况下还能提高性能。

如果应用程序在连接之间更改语言环境设置，那么您不应将 DB2_FORCE_NLS_CACHE 设置为 TRUE。如果发生这种情况，那么即使在语言环境设置更改后，也将返回原始语言环境信息。通常，多线程应用程序不会更改语言环境设置，这将确保应用程序保持具有线程安全性。

对多线程嵌入式 SQL 应用程序进行故障诊断

使用多个线程的应用程序比单线程应用程序更为复杂。

这种额外的复杂性有可能导致一些意外的问题。

编写多线程应用程序时，必须考虑以下上下文问题:

两个或更多个上下文之间的数据库依赖关系。

应用程序中的每个上下文都有自己的一组数据库资源，其中包括对数据库对象的锁定。此特征使得两个上下文有可能在访问同一个数据库对象时发生死锁。当数据库管理器检测到死锁时，向应用程序返回 SQLCODE -911，并回滚其工作单元。

两个或更多个上下文之间的应用程序依赖关系。

谨慎地使用任何将在上下文之间建立依赖关系的编程技术。锁存器、信标和临界区是可能会建立此类依赖关系的编程技术的示例。如果应用程序的两个上下文之间同时存在应用程序依赖关系和数据库依赖关系，那么该应用程序可能会发生死锁。如果某些依赖关系在数据库管理器外部，那么该死锁将不会被检测到，因此该应用程序将暂挂或挂起。

预防多个上下文之间发生死锁。

因为 数据库管理器 无法检测到线程之间的死锁，所以请以避免死锁的方式编码应用程序。

考虑 数据库管理器 无法检测到的死锁的一个示例是，应用程序有两个上下文，并且这两个上下文访问同一个数据结构。为了避免这两个上下文同时更改该数据结构时发生问题，该数据结构由信标提供保护。该样本上下文显示为以下伪码:

```
上下文 1
SELECT * FROM TAB1 FOR UPDATE....
UPDATE TAB1 SET....
获取信标
访问数据结构
释放信标
COMMIT

上下文 2
获取信标
访问数据结构
SELECT * FROM TAB1...
释放信标
COMMIT
```

假定第一个上下文成功地执行 SELECT 和 UPDATE 语句，而第二个上下文获取信标并访问数据结构。现在，第一个上下文尝试获取该信标，但它由于第二个上下文正占用该信标而无法完成操作。现在，第二个上下文尝试从表 TAB1 中读取行，但它由于第一个上下文已锁定数据库而停止。现在，应用程序处于这样一种状态: 上下文 1 无法在上下文 2 完成前完成，而上下文 2 正在等待上下文 1 完成。应用程序发生死锁，但由于 数据库管理器 不了解信标依赖关系，因此不会回滚任何一个上下文。未解决的依赖关系将使应用程序保持处于暂挂状态。

您可以通过多种方法来避免上一示例中可能发生的死锁。

- 在获取信标前释放所有锁定。

  更改上下文 1 的代码，以便在获取信标前执行落实。
- 不要在信标所保护的节中编码 SQL 语句。

  更改上下文 2 的代码，以便在执行 SELECT 前释放信标。
- 在信标中编码所有 SQL 语句。

  更改上下文 1 的代码，以便在运行 SELECT 语句前获取信标。虽然这种技术有效，但并不强烈建议采用此技术，这是因为信标将使得对数据库管理器进行的访问序列化，从而有可能导致使用多个线程的优势消失。
- 将 *locktimeout* 数据库配置参数设置为除 -1 以外的值。

  虽然除 -1 以外的值不会防止死锁，但它将允许执行继续。上下文 2 最终被回滚，这是因为它无法获取所请求获取的锁定。处理回滚错误时，上下文 2 应该释放信标。释放信标后，上下文 1 就能够继续，接着上下文 2 可以再次尝试其工作。

本节通过示例来描述用于避免死锁的技术，但您可以将这些技术应用于所有多线程应用程序。通常，请像对待任何受保护资源一样对待数据库管理器，您应该不会遇到与多线程应用程序相关的问题。

第 3 章 嵌入式 SQL 应用程序编程

嵌入式 SQL 应用程序编程涉及使用所选嵌入式 SQL 编程语言组装应用程序时所需执行的所有步骤。一旦确定嵌入式 SQL 就是能够满足编程需要的合适 API，并且在设计嵌入式 SQL 应用程序之后，您就已准备好进行嵌入式 SQL 应用程序编程。

先决条件:

- 选择是使用静态 SQL 语句还是动态 SQL 语句
- 设计嵌入式 SQL 应用程序

嵌入式 SQL 应用程序编程工作由下列子任务组成:

- 包括所需的头文件
- 选择受支持的嵌入式 SQL 编程语言
- 声明主变量以表示要包括在 SQL 语句中的值
- 连接到数据源
- 执行 SQL 语句
- 处理与 SQL 语句的执行相关的 SQL 错误和警告
- 与数据源断开连接

有了完整的嵌入式 SQL 应用程序之后，您就已准备好编译和运行应用程序: 编译嵌入式 SQL 应用程序。

嵌入式 SQL 源文件

在开发包含嵌入式 SQL 的源代码时，需要遵循每种受支持主语言的特定文件命名约定。

C 和 C++ 的输入和输出文件

缺省情况下，源应用程序可以具有下列扩展名:

.sqc 用于所有受支持操作系统上的 C 文件

.sqC 用于 UNIX 和 Linux 操作系统上的 C++ 文件

.sqx 用于 Windows 操作系统上的 C++ 文件

缺省情况下，相应的预编译器输出文件具有下列扩展名:

.c 用于所有受支持操作系统上的 C 文件

.C 用于 UNIX 和 Linux 操作系统上的 C++ 文件

.cxx 用于 Windows 操作系统上的 C++ 文件

您可以使用 OUTPUT 预编译选项来覆盖经过修改的输出源文件的名称和路径。如果使用 TARGET C 或 TARGET CPLUSPLUS 预编译选项，那么输入文件不需要特定的扩展名。

COBOL 的输入和输出文件

缺省情况下，源应用程序具有以下扩展名:

.sqb 用于所有操作系统上的 COBOL 文件

但是，如果使用了 TARGET 预编译选项（TARGET ANSI_COBOL、TARGET IBMCOB 或 TARGET MFCOB），那么输入文件可以具有您选择的任何扩展名。

缺省情况下，相应的预编译器输出文件具有下列扩展名:

.cbl 用于所有操作系统上的 COBOL 文件

但是，您可以使用 OUTPUT 预编译选项来指定经过修改的输出源文件的新名称和路径。

FORTRAN 的输入和输出文件

缺省情况下，源应用程序具有以下扩展名:

.sqf 用于所有操作系统上的 FORTRAN 文件

但是，如果将 TARGET 预编译选项与 FORTRAN 选项配合使用，那么输入文件可以具有您选择的任何扩展名。

缺省情况下，相应的预编译器输出文件具有下列扩展名:

.f 用于 UNIX 和 Linux 操作系统上的 FORTRAN 文件

.for 用于 Windows 操作系统上的 FORTRAN 文件

但是，您可以使用 OUTPUT 预编译选项来指定经过修改的输出源文件的新名称和路径。

使用 C 语言的嵌入式 SQL 应用程序模板

这是一个简单的嵌入式 SQL 应用程序，用于测试嵌入式 SQL 开发环境以及帮助您了解嵌入式 SQL 应用程序的基本结构。

嵌入式 SQL 应用程序必须具有以下结构:

- 包括必需的头文件
- 要包括在 SQL 语句中的值的主变量声明
- 数据库连接
- 执行 SQL 语句
- 处理与 SQL 语句的执行相关的 SQL 错误和警告
- 删除数据库连接

以下源代码演示使用 C 编写的嵌入式 SQL 应用程序的基本结构。

样本程序: template.sqc

```
#include <stdio.h>                                                    1
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>

EXEC SQL BEGIN DECLARE SECTION;                                       2
  short id;
  char name[10];
  short dept;
  double salary;
  char hostVarStmtDyn[50];
```

```
EXEC SQL END DECLARE SECTION;

int main()
 {
   int rc = 0;                                                          3
   EXEC SQL INCLUDE SQLCA;                                              4

   /* 连接到数据库 */
   printf("\n Connecting to database...");
   EXEC SQL CONNECT TO "sample";                                        5
   if (SQLCODE  <0)                                                     6
   {
      printf("\nConnect Error:  SQLCODE =
      goto connect_reset;
   }
   else
   {
      printf("\n Connected to database.\n");
   }

   /* 使用静态 SQL 来执行 SQL 语句（查询）；将单一结果行的值复制到主变量 */
   EXEC SQL SELECT id, name, dept, salary                               7
           INTO :id, :name, :dept, :salary
           FROM staff WHERE id = 310;
   if (SQLCODE  <0)                                                     6
   {
      printf("Select Error:  SQLCODE =
   }
   else
   {
      /* 将主变量值打印到标准输出 */
      printf("\n Executing a static SQL query statement, searching for
         \n the id value equal to 310\n");
      printf("\n ID   Name        DEPT       Salary\n");
      printf("
   }

   strcpy(hostVarStmtDyn, "UPDATE staff
                          SET salary = salary + 1000
                          WHERE dept = ?");
   /* 使用主变量和动态 SQL 来执行 SQL 语句（操作）*/
   EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
   if (SQLCODE  <0)                                                     6
   {
      printf("Prepare Error:  SQLCODE =
   }
   else
   {
      EXEC SQL EXECUTE StmtDyn USING :dept;                             8
   }
   if (SQLCODE  <0)                                                     6
   {
      printf("Execute Error:  SQLCODE =
   }

   /* 使用静态 SQL 和游标来读取已更新的行 */
   EXEC SQL DECLARE posCur1 CURSOR FOR
      SELECT id, name, dept, salary
      FROM staff WHERE id = 310;
   if (SQLCODE  <0)                                                   6
   {
      printf("Declare Error:  SQLCODE =
   }
   EXEC SQL OPEN posCur1;
   EXEC SQL FETCH posCur1 INTO :id, :name, :dept, :salary ;           9
   if (SQLCODE  <0)                                                   6
   {
```

```
   printf("Fetch Error:  SQLCODE =
}
else
{
   printf(" Executing an dynamic SQL statement, updating the
            \n salary value for the id equal to 310\n");
   printf("\n ID   Name        DEPT       Salary\n");
   printf("
}

EXEC SQL CLOSE posCur1;

/* 落实事务 */
printf("\n  Commit the transaction.\n");
EXEC SQL COMMIT;                                                    10
if (SQLCODE  <0)                                                    6
{
   printf("Error:  SQLCODE =
}

/* 与数据库断开连接 */
connect_reset :
   EXEC SQL CONNECT RESET;                                          11
   if (SQLCODE  <0)                                                 6
   {
      printf("Connection Error:  SQLCODE =
   }
return 0;
} /* 结束 main */
```

有关**样本程序**: **template.sqc** 的注释:

注解	描述
1	包含文件: 此伪指令将一个文件包括到源应用程序中。
2	声明节: 声明主变量，这些主变量用于存放 C 应用程序的 SQL 语句中引用的值。
3	局部变量声明: 此块声明要在应用程序中使用的局部变量。这些变量不是主变量。
4	包括 SQLCA 结构: 执行每个 SQL 语句后，都将更新 SQLCA 结构。此模板应用程序使用某些 SQLCA 字段进行错误处理。
5	连接到数据库: 使用数据库的初始步骤是，与该数据库建立连接。这里，通过执行 SQL 语句 CONNECT 来建立连接。
6	错误处理: 检查是否已发生错误。
7	执行查询: 通过执行此 SQL 语句，将表所返回的数据赋予主变量。在执行 SQL 语句后使用的 C 代码将主变量中的值打印到标准输出。
8	执行操作: 执行此 SQL 语句将更新表中由部门编号标识的一组行。准备过程 (EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;) 是一个步骤，此步骤将主变量值（例如此语句中引用的主变量值）与所要执行的 SQL 语句绑定。
9	执行操作: 在此行以及上一行中，此应用程序在静态 SQL 中使用游标来选择表中的信息并打印数据。在声明并打开游标之后，提取数据，最后关闭该游标。
10	落实事务: COMMIT 语句将工作单元中进行的数据库更改最终化。
11	最后，必须删除数据库连接。

嵌入式 SQL 应用程序所需的包含文件和定义

包含文件用于提供库中使用的函数和类型。必须包括这些文件，程序才能使用库函数。缺省情况下，这些文件将安装在 $HOME/sqllib/include 文件夹中。每种主语言都通过自己的方法来包括文件，并使用不同的文件扩展名。根据所指定语言的不同，必须采取特定的预防措施，例如指定文件路径。

C 和 C++ 嵌入式 SQL 应用程序的包含文件

特定于主语言的 C 和 C++ 包含文件（头文件）的文件扩展名为 .h。您可以通过两种方法来包括文件: EXEC SQL INCLUDE 语句和 #include 宏。预编译器将忽略 #include，而只处理通过 EXEC SQL INCLUDE 语句包括的文件。为了找到使用 EXEC SQL INCLUDE 包括的文件，DB2 C 预编译器将先搜索当前目录，然后搜索 DB2INCLUDE 环境变量所指定的目录。请考虑下列示例:

- EXEC SQL INCLUDE payroll;

  如果未将 INCLUDE 语句中指定的文件引在引号中（如上所示），那么 C 预编译器将在它查找的每个目录中依次搜索 payroll.sqc 和 payroll.h。在 UNIX 和 Linux 操作系统上，C++ 预编译器将在它查找的每个目录中依次搜索 payroll.sqC、payroll.sqx、payroll.hpp 和 payroll.h。在 Windows 32 位操作系统上，C++ 预编译器将在它查找的每个目录中依次搜索 payroll.sqx、payroll.hpp 和 payroll.h。

- EXEC SQL INCLUDE 'pay/payroll.h';

  如果将文件名括在引号中（如上所示），那么不会对该文件名添加扩展名。

  如果引号中的文件名未包含绝对路径，那么将使用 DB2INCLUDE 的内容来搜索文件（此内容被添加到 INCLUDE 文件名中指定的任何路径之前）。例如，在 UNIX 和 Linux 操作系统上，如果将 DB2INCLUDE 设置为“/disk2:myfiles/c”，那么 C 或 C++ 预编译器将依次搜索“./pay/payroll.h”、“/disk2/pay/payroll.h”和“./myfiles/c/pay/payroll.h”。该文件的实际路径将显示在预编译器消息中。在 Windows 操作系统上，请将以上示例中的反斜杠 (\) 替换为正斜杠。

  注意，如果预编译器选项 COMPATIBILITY_MODE 设置为 ORA，那么可使用双引号指定包含文件名称，例如，EXEC SQL INCLUDE "abc.h";。DB2 数据库管理器提供此功能是为了方便从其他数据库系统迁移嵌入的 SQL C 应用程序。

注: 命令行处理器将对 DB2INCLUDE 的设置进行高速缓存。要在发出任何 CLP 命令后更改 DB2INCLUDE 的设置，请输入 TERMINATE 命令，然后重新连接到数据库并进行预编译。

为了帮助使编译器错误重新与原始源代码相关，预编译器将在输出文件中生成 #line 宏。这将允许编译器使用源代码或者所包括源文件的文件名和行号（而不是预编译的输出源文件中的行号）来报告错误。

但是，如果指定了 PREPROCESSOR 选项，那么预编译器生成的所有 #line 宏都将引用外部 C 预处理器所生成的预处理文件。某些将会使源代码与对象代码相关的调试器和其他工具无法始终很好地使用 #line 宏。如果您想要使用的工具出现异常行为，那么在预编译时，请使用 NOLINEMACRO 选项（与 DB2 PREP 配合使用）。此选项将不允许生成 #line 宏。

下一节描述可以在应用程序中使用的包含文件。

SQLADEF（sqladef.h）

此文件包含预编译的 C 和 C++ 应用程序所使用的函数原型。

SQLCA（sqlca.h）

此文件定义 SQL 通信区（SQLCA）结构。SQLCA 包含数据库管理器所使用的变量，这些变量为应用程序提供关于 SQL 语句和 API 调用的执行情况的错误信息。

SQLCODES（sqlcodes.h）

此文件为 SQLCA 结构的 SQLCODE 字段定义常量。

SQLDA（sqlda.h）

此文件定义 SQL 描述符区域（SQLDA）结构。SQLDA 用于在应用程序与数据库管理器之间传递数据。

SQLEXT（sqlext.h）

此文件包含那些未包括在 X/Open 调用级接口规范中的 ODBC 第 1 级和第 2 级 API 的函数原型和常量，因此在得到 Microsoft 公司许可的情况下使用。

SQLE819A（sqle819a.h）

如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE819B（sqle819b.h）

如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850A（sqle850a.h）

如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850B（sqle850b.h）

如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932A（sqle932a.h）

如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5035（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932B（sqle932b.h）

如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5026（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLJACB（sqljacb.h）

此文件为 DB2 Connect 接口定义常量、结构和控制块。

SQLSTATE（sqlstate.h）

此文件为 SQLCA 结构的 SQLSTATE 字段定义常量。

SQLSYSTM（sqlsystm.h）

此文件包含数据库管理器 API 和数据结构所使用的特定于平台的定义。

SQLUDF（sqludf.h）

此文件定义用于编写用户定义的函数 (UDF) 的常量和接口结构。

SQLUV（sqluv.h）

此文件为异步读日志 API 以及表装入和卸载供应商所使用的 API 定义结构、常量和原型。

COBOL 嵌入式 SQL 应用程序的包含文件

特定于主语言的 COBOL 包含文件的文件扩展名为 .cbl。如果您使用 IBM® COBOL 编译器的“System/390® 主数据类型支持”功能部件，那么应用程序的 DB2 包含文件在以下目录中:

```
$HOME/sqllib/include/cobol_i
```

如果您使用提供的脚本文件来构建 DB2 样本程序，那么必须将脚本文件中指定的包含文件路径更改为 cobol_i 目录而不是 cobol_a 目录。

如果您**未**使用 IBM COBOL 编译器的“System/390 主数据类型支持”功能部件，或者使用此编译器的先前版本，那么应用程序的 DB2 包含文件在以下目录中:

```
$HOME/sqllib/include/cobol_a
```

为了找到包含文件，DB2 COBOL 预编译器将先搜索当前目录，然后搜索 DB2INCLUDE 环境变量所指定的目录。请考虑下列示例:

- `EXEC SQL INCLUDE payroll END-EXEC.`

  如果未将 INCLUDE 语句中指定的文件引在引号中（如先前示例所示），那么预编译器将在它查找的每个目录中依次搜索 payroll.sqb、payroll.cpy 和 payroll.cbl。

- `EXEC SQL INCLUDE 'pay/payroll.cbl' END-EXEC.`

  如果将文件名括在引号中（如上所示），那么不会对该文件名添加扩展名。

  如果引号中的文件名未包含绝对路径，那么将使用 DB2INCLUDE 的内容来搜索文件（此内容被添加到 INCLUDE 文件名中指定的任何路径之前）。例如，对于用于 AIX 的 DB2 数据库系统而言，如果 DB2INCLUDE 设置为“/disk2:myfiles/cobol”，那么预编译器将依次搜索“./pay/payroll.cbl”、“/disk2/pay/payroll.cbl”和“./myfiles/cobol/pay/payroll.cbl”。该文件的实际路径将显示在预编译器消息中。在 Windows 平台上，请将以前显示的示例中的反斜杠 (\) 替换为正斜杠。

注: DB2 命令行处理器将对 DB2INCLUDE 的设置进行高速缓存。要在发出任何 CLP 命令后更改 DB2INCLUDE 的设置，请输入 TERMINATE 命令，然后重新连接到数据库并进行预编译。

下面描述可以在应用程序中使用的包含文件:

SQLCA（sqlca.cbl）

此文件定义 SQL 通信区（SQLCA）结构。SQLCA 包含数据库管理器所使用的变量，这些变量为应用程序提供关于 SQL 语句和 API 调用的执行情况的错误信息。

SQLCA_92（sqlca_92.cbl）

此文件包含 SQL 通信区（SQLCA）结构的 FIPS SQL92 入门级相容版本。在编写符合 FIPS SQL92 入门级标准的 DB2 应用程序时，应该包括此文件以代

替 sqlca.cbl 文件。当 LANGLEVEL 预编译器选项设置为 SQL92E 时，DB2 预编译器将自动包括 sqlca_92.cbl 文件。

SQLCODES（sqlcodes.cbl）
此文件为 SQLCA 结构的 SQLCODE 字段定义常量。

SQLDA（sqlda.cbl）
此文件定义 SQL 描述符区域（SQLDA）结构。SQLDA 用于在应用程序与数据库管理器之间传递数据。

SQLEAU（sqleau.cbl）
此文件包含 DB2 安全性审计 API 所需的常量和结构定义。如果使用这些 API，那么需要在程序中包括此文件。此文件还包含审计跟踪记录中的字段的常量和关键字值定义。这些定义可供外部或供应商审计跟踪抽取程序使用。

SQLETSD（sqletsd.cbl）
此文件定义表空间描述符结构 SQLETSDESC，该结构将被传递到“创建数据库”API，即 sqlgcrea。

SQLE819A（sqle819a.cbl）
如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE819B（sqle819b.cbl）
如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850A（sqle850a.cbl）
如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850B（sqle850b.cbl）
如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932A（sqle932a.cbl）
如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5035（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932B（sqle932b.cbl）
如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5026（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQL1252A（sql1252a.cbl）
如果数据库的代码页是 1252（Windows Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQL1252B（sql1252b.cbl）
如果数据库的代码页是 1252（Windows Latin-1），那么此序列将根据主机

CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLSTATE（sqlstate.cbl）
此文件为 SQLCA 结构的 SQLSTATE 字段定义常量。

SQLUDF（sqludf.cbl）
此文件定义用于编写用户定义的函数 (UDF) 的常量和接口结构。

SQLUTBCQ（sqlutbcq.cbl）
此文件定义“表空间容器查询”数据结构 SQLB-TBSCONTQRY-DATA，此结构与 sqlgstsc、sqlgftcq 和 sqlgtcq 表空间容器查询 API 配合使用。

SQLUTBSQ（sqlutbsq.cbl）
此文件定义“表空间查询”数据结构 SQLB-TBSQRY-DATA，此结构与 sqlgstsq、sqlgftsq 和 sqlgtsq 表空间查询 API 配合使用。

FORTRAN 嵌入式 SQL 应用程序的包含文件

在 UNIX 和 Linux 操作系统上，特定于主语言的 FORTRAN 包含文件的文件扩展名为 .f，在 Windows 操作系统上，其扩展名为 .for。您可以通过两种方法来包括文件: EXEC SQL INCLUDE 语句和 FORTRAN INCLUDE 语句。预编译器将忽略 FORTRAN INCLUDE 语句，而只处理通过 EXEC SQL 语句包括的文件。为了找到包含文件，DB2 FORTRAN 预编译器将先搜索当前目录，然后搜索 DB2INCLUDE 环境变量所指定的目录。

请考虑下列示例:

- `EXEC SQL INCLUDE payroll`

  如果未将 INCLUDE 语句中指定的文件引在引号中（如先前的示例所示），那么预编译器将在它查找的每个目录中依次搜索 payroll.sqf 和 payroll.f（在 Windows 操作系统上，这是 payroll.for）。

- `EXEC SQL INCLUDE 'pay/payroll.f'`

  如果将文件名括在引号中（如上所示），那么不会对该文件名添加扩展名。（对于 Windows 操作系统，应该将文件指定为 'pay\payroll.for'。）

  如果引号中的文件名未包含绝对路径，那么将使用 DB2INCLUDE 的内容来搜索文件（此内容被添加到 INCLUDE 文件名中指定的任何路径之前）。例如，对于用于 UNIX 和 Linux 操作系统的 DB2，如果 DB2INCLUDE 设置为“/disk2:myfiles/fortran”，那么预编译器将依次搜索“./pay/payroll.f”、“/disk2/pay/payroll.f”和“./myfiles/cobol/pay/payroll.f”。该文件的实际路径将显示在预编译器消息中。在 Windows 操作系统上，请将先前显示示例中的反斜杠 (\) 替换为正斜杠，并将扩展名"for"替换为"f"。

注: DB2 命令行处理器将对 DB2INCLUDE 的设置进行高速缓存。要在发出任何 CLP 命令后更改 DB2INCLUDE 的设置，请输入 TERMINATE 命令，然后重新连接到数据库并进行预编译。

进行 DB2 数据库应用程序开发所需的 32 位 FORTRAN 头文件先前在 $INSTHOME/sqllib/include 中，现在则是在 $INSTHOME/sqllib/include32 中。

在版本 8.1 中，这些文件在 $INSTDIR/sqllib/include 目录中，该目录是指向下列其中一个目录的符号链接: $DB2DIR/include 或 $DB2DIR/include64，这取决于它是 32 位实例还是 64 位实例。

在版本 9.1 中，$DB2DIR/include 将包含所有包含文件（32 位和 64 位），而 $DB2DIR/include32 只包含 32 位 FORTRAN 文件和一个自述文件，以指示除 FORTRAN 以外，32 位包含文件与 64 位包含文件相同。

只有在 AIX、Solaris、HP-PA 和 HP-IPF 上，才存在 $DB2DIR/include32 目录。

可以在应用程序中使用下列 FORTRAN 包含文件。

SQLCA（sqlca_cn.f 和 sqlca_cs.f）

此文件定义 SQL 通信区（SQLCA）结构。SQLCA 包含数据库管理器所使用的变量，这些变量为应用程序提供关于 SQL 语句和 API 调用的执行情况的错误信息。

提供了两个用于 FORTRAN 应用程序的 SQLCA 文件。缺省文件 sqlca_cs.f 以 IBM SQL 兼容格式来定义 SQLCA 结构。使用 SQLCA NONE 选项进行预编译的 sqlca_cn.f 文件以能够提高性能的格式来定义 SQLCA 结构。

SQLCA_92（sqlca_92.f）

此文件包含 SQL 通信区（SQLCA）结构的 FIPS SQL92 入门级相容版本。在编写符合 FIPS SQL92 入门级标准的 DB2 应用程序时，应该包括此文件以代替 sqlca_cn.f 或 sqlca_cs.f 文件。当 LANGLEVEL 预编译器选项设置为 SQL92E 时，DB2 预编译器将自动包括 sqlca_92.f 文件。

SQLCODES（sqlcodes.f）

此文件为 SQLCA 结构的 SQLCODE 字段定义常量。

SQLDA（sqldact.f）

此文件定义 SQL 描述符区域（SQLDA）结构。SQLDA 用于在应用程序与数据库管理器之间传递数据。

SQLEAU（sqleau.f）

此文件包含 DB2 安全性审计 API 所需的常量和结构定义。如果使用这些 API，那么需要在程序中包括此文件。此文件还包含审计跟踪记录中的字段的常量和关键字值定义。这些定义可供外部或供应商审计跟踪抽取程序使用。

SQLE819A（sqle819a.f）

如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE819B（sqle819b.f）

如果数据库的代码页是 819（ISO Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850A（sqle850a.f）

如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE850B（sqle850b.f）

如果数据库的代码页是 850（ASCII Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932A（sqle932a.f）

如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5035（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLE932B（sqle932b.f）

如果数据库的代码页是 932（ASCII 日语），那么此序列将根据主机 CCSID 5026（EBCDIC 日语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQL1252A（sql1252a.f）

如果数据库的代码页是 1252（Windows Latin-1），那么此序列将根据主机 CCSID 500（EBCDIC 国际）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQL1252B（sql1252b.f）

如果数据库的代码页是 1252（Windows Latin-1），那么此序列将根据主机 CCSID 037（EBCDIC 美国英语）二进制整理顺序对非 FOR BIT DATA 字符串进行排序。此文件由 CREATE DATABASE API 使用。

SQLSTATE（sqlstate.f）

此文件为 SQLCA 结构的 SQLSTATE 字段定义常量。

SQLUDF（sqludf.f）

此文件定义用于编写用户定义的函数 (UDF) 的常量和接口结构。

声明用于进行错误处理的 SQLCA

您可以在应用程序中声明 SQLCA，以便数据库管理器可以向应用程序返回信息。

关于此任务

对程序进行预处理时，数据库管理器将插入主语言变量声明来替换 INCLUDE SQLCA 语句。系统与程序进行通信时，使用这些变量来存放警告标志、错误代码和诊断信息。

在执行每个 SQL 语句之后，系统将在 SQLCODE 和 SQLSTATE 中返回一个返回码。SQLCODE 是整数值，它提供有关语句执行情况的摘要信息；SQLSTATE 是字符字段，它提供 IBM 的各个关系数据库产品的公共错误代码。SQLSTATE 还符合 ISO/ANS SQL92 和 FIPS 127-2 标准。

注: FIPS 127-2 是指*数据库语言 SQL 的美国联邦信息处理标准出版物 127-2*。ISO/ANS SQL92 是指*美国国家标准数据库语言 SQL X3.135-1992* 和*国际标准 ISO/IEC 9075:1992，数据库语言 SQL*。

注意，如果 SQLCODE 小于 0，那么表示已发生错误，并且未处理该语句。如果 SQLCODE 大于 0，那么表示已发出警告，但仍处理该语句。

对于使用 C 或 C++ 编写的 DB2 应用程序，如果该应用程序由多个源文件组成，那么应该只有其中一个文件包含 EXEC SQL INCLUDE SQLCA 语句以避免多次定义 SQLCA。其余源文件必须使用下列各行:

```
#include "sqlca.h"
extern struct sqlca sqlca;
```

过程

要声明 SQLCA，请在程序中编码 INCLUDE SQLCA 语句:

- 对于 C 或 C++ 应用程序，请使用:

  ```
  EXEC SQL INCLUDE SQLCA;
  ```

- 对于 Java 应用程序，不能显式地使用 SQLCA。而是，使用 SQLException 实例方法来获取 SQLSTATE 和 SQLCODE 值。
- 对于 COBOL 应用程序，请使用:

  ```
  EXEC SQL INCLUDE SQLCA END-EXEC.
  ```

- 对于 FORTRAN 应用程序，请使用:

  ```
  EXEC SQL INCLUDE SQLCA
  ```

下一步做什么

如是应用程序必须符合 ISO/ANS SQL92 或 FIPS 127-2 标准，请不要使用上述语句或 INCLUDE SQLCA 语句。

使用 WHENEVER 语句进行错误处理

WHENEVER 语句导致预编译器生成源代码，以指示应用程序在执行期间发生错误、出现警告或找不到行时转至指定的标号。WHENEVER 语句将影响后续所有可执行的 SQL 语句，直到另一个 WHENEVER 语句改变这种情况为止。

WHENEVER 语句有三种基本格式:

```
EXEC SQL WHENEVER SQLERROR   操作
EXEC SQL WHENEVER SQLWARNING 操作
EXEC SQL WHENEVER NOT FOUND  操作
```

在这些语句中:

SQLERROR
标识 SQLCODE < 0 的任何情况。

SQLWARNING
标识 SQLWARN(0) = W 或 SQLCODE > 0 但不等于 100 的任何情况。

NOT FOUND
标识 SQLCODE = 100 的任何情况。

在每种情况中，*action* 可以是 CONTINUE 或 GO TO <label>:

CONTINUE
指示继续执行应用程序中的下一条指令。

GO TO *标号*
指示转至紧跟在 GO TO 后指定的标号之后的语句。（GO TO 可以是两个词，也可以是一个词 GOTO。）

如果未使用 WHENEVER 语句，那么缺省操作是，执行期间发生错误、警告或异常条件时继续进行处理。

必须在所要影响的 SQL 语句之前使用 WHENEVER 语句。否则，预编译器不知道应该为可执行的 SQL 语句生成附加的错误处理代码。在任何时候，这三种基本格式的任意组合都可以处于活动状态。声明这三种格式的顺序并不重要。

为了避免发生无穷循环，请确保在处理程序中执行任何 SQL 语句之前撤销 WHENEVER 处理。可以使用 WHENEVER SQLERROR CONTINUE 语句来执行撤销。

在嵌入式 SQL 应用程序中连接到 DB2 数据库

在使用数据库之前，必须与该数据库建立连接。嵌入式 SQL 提供了多种方法来包括用于建立数据库连接的代码。根据嵌入式 SQL 主编程语言的不同，可以通过一种或多种方法来完成此任务。

可以通过隐式或显式方式来建立数据库连接。隐式连接是指，假定连接所使用的用户标识是当前用户标识。对于数据库应用程序，建议您不要使用此类连接。强烈建议您使用要求指定用户标识和密码的显式数据库连接。

在 C 和 C++ 嵌入式 SQL 应用程序中连接到 DB2 数据库

使用 C 和 C++ 应用程序时，可以通过执行以下语句来建立数据库连接。

```
EXEC SQL CONNECT TO sample;
```

如果要使用特定的用户标识（herrick）和密码（mypassword），请使用以下语句:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword;
```

注意，如果预编译器选项 COMPATIBILITY_MODE 设置为 ORA，那么支持 CONNECT 语句的以下附加语法。DB2 数据库管理器提供此功能是为了方便从其他数据库系统迁移嵌入的 SQL C 应用程序。

```
EXEC SQL CONNECT [ username IDENTIFIED BY password ][ USING  dbname ] ;
```

下表描述了这些参数:

参数	描述
用户名	用于指定数据库用户名的主变量或字符串
密码	用于指定密码的主变量或字符串
数据库名称	用于指定数据库名称的主变量或字符串

在 COBOL 嵌入式 SQL 应用程序中连接到 DB2 数据库

使用 COBOL 应用程序时，通过执行以下语句来建立数据库连接。此语句使用缺省用户名来创建与 sample 数据库的连接。

```
EXEC SQL CONNECT TO sample END-EXEC.
```

如果要使用特定的用户标识（herrick）和密码（mypassword），请使用以下语句:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword END-EXEC.
```

在 FORTRAN 嵌入式 SQL 应用程序中连接到 DB2 数据库

使用 FORTRAN 应用程序时，通过执行以下语句来建立数据库连接。此语句使用缺省用户名来创建与 sample 数据库的连接。

```
EXEC SQL CONNECT TO sample
```

如果要使用特定的用户标识（herrick）和密码（mypassword），请使用以下语句:

```
EXEC SQL CONNECT TO sample USER herrick USING mypassword
```

在 REXX 嵌入式 SQL 应用程序中连接到 DB2 数据库

使用 REXX 应用程序时，通过执行以下语句来建立数据库连接。此语句使用缺省用户名来创建与 sample 数据库的连接。

```
CALL SQLEXEC 'CONNECT TO sample'
```

如果要使用特定的用户标识（herrick）和密码（mypassword），请使用以下语句:

```
CALL SQLEXEC 'CONNECT TO sample USER herrick USING mypassword'
```

嵌入式 SQL 应用程序中映射到 SQL 数据类型的数据类型

要在应用程序与数据库之间交换数据，请对所使用的变量使用正确的数据类型映射。当预编译器找到主变量声明时，它确定适当的 SQL 类型值。对于每种主语言，您都必须遵循一些该特定语言所特有的特殊映射规则。

C 和 C++ 嵌入式 SQL 应用程序中的受支持 SQL 数据类型

某些预定义的 C 和 C++ 数据类型与 DB2 数据库列类型相对应。只有这些 C 和 C++ 数据类型可以被声明为主变量。

下列各表列示与每种列类型等同的 C 和 C++ 类型。当预编译器找到主变量声明时，它确定适当的 SQL 类型值。数据库管理器使用此值对它自己与应用程序之间交换的数据进行转换。

表 2. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
SMALLINT （500 或 501）	short short int sqlint16	16 位带符号整数
INTEGER （496 或 497）	intlong long int sqlint32[2]	32 位带符号整数
BIGINT （492 或 493）	long long long __int64 sqlint64[3]	64 位带符号整数
REAL[5] （480 或 481）	float	单精度浮点数

表 2. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
DOUBLE[6] （480 或 481）	double	双精度浮点数
DECIMAL(*p*,*s*) （484 或 485）	没有精确的等同类型；请使用 double	压缩十进制 （考虑使用 CHAR 和 DECIMAL 函数将压缩数据字段作为字符数据进行处理。）
CHAR(1) （452 或 453）	char	单一字符
CHAR(*n*) （452 或 453）	没有精确的等同类型；请使用 char[*n*+*1*]，其中 n 的大小足以存放数据 1<=*n*<=254	定长字符串

VARCHAR(*n*)
（448 或 449）
	struct tag {
	short int;
	char[*n*]
	}
	
	1<=*n*<=32 672
		带有 2 字节字符串长度指示符而且并非以 null 结束的变长字符串

	另外，也可以使用 char[*n*+*1*]，其中 n 的大小足以存放数据 1<=*n*<=32 672	以 null 结束的变长字符串 **注**: 被赋予 SQL 类型 460/461。

LONG VARCHAR[8]
（456 或 457）
	struct tag {
	short int;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		带有 2 字节字符串长度指示符而且并非以 null 结束的变长字符串

CLOB(*n*) （408 或 409）	SQL 类型是 clob(*n*) 1<=*n*<=2 147 483 647	带有 4 字节字符串长度指示符而且并非以 null 结束的变长字符串
CLOB 定位器变量[7] （964 或 965）	SQL 类型是 clob_locator	标识驻留在服务器上的 CLOB 实体
CLOB 文件引用变量[7] （920 或 921）	SQL 类型是 clob_file	包含 CLOB 数据的文件的描述符
BLOB(*n*) （404 或 405）	SQL 类型是 blob(*n*) 1<=*n*<=2 147 483 647	带有 4 字节字符串长度指示符而且并非以 null 结束的变长二进制字符串

表 2. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
BLOB 定位器变量[7] （960 或 961）	SQL 类型是 blob_locator	标识服务器上的 BLOB 实体
BLOB 文件引用变量[7] （916 或 917）	SQL 类型是 blob_file	包含 BLOB 数据的文件的描述符
DATE （384 或 385）	以 null 结束的字符格式	至少允许 11 个字符，以便容纳 null 终止符
	VARCHAR 结构化格式	至少允许 10 个字符
TIME （388 或 389）	以 null 结束的字符格式	至少允许 9 个字符，以便容纳 null 终止符
	VARCHAR 结构化格式	至少允许 8 个字符
TIMESTAMP(*p*) [4]（392 或 393）	以 null 结束的字符格式	允许 20-33 个字符，以便容纳 null 终止符
	VARCHAR 结构化格式	允许 19-32 个字符

XML[8]
（988 或 989）
	struct {
	sqluint32 length;
	char data[*n*];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_CLOB
		XML 值

BINARY	unsigned char myBinField[4]; 1<= *n* <=255	二进制数据
VARBINARY	struct myVarBinField_t {sqluint16 length;char data[12];} myVarBinField; 1<= *n* <=32704	Varbinary 数据

只有在 DBCS 或 EUC 环境中使用 WCHARTYPE NOCONVERT 选项进行预编译的情况下，下列数据类型才可用。

表 3. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
GRAPHIC(1) （468 或 469）	sqldbchar	单一双字节字符
GRAPHIC(*n*) （468 或 469）	没有精确的等同类型；请使用 sqldbchar[*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=127	双字节定长字符串

表 3. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述

VARGRAPHIC(*n*)
（464 或 465）
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 336
		带有 2 字节字符串长度指示符而且并非以 null 结束的可变长度双字节字符串

	另外，也可以使用 sqldbchar[*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=16 336	以 null 结束的可变长度双字节字符串 **注**: 被指定给 SQL 类型 400/401。

LONG VARGRAPHIC[8]
（472 或 473）
	struct tag {
	short int;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		带有 2 字节字符串长度指示符而且并非以 null 结束的可变长度双字节字符串

只有在 DBCS 或 EUC 环境中使用 WCHARTYPE CONVERT 选项进行预编译的情况下，下列数据类型才可用。

表 4. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
GRAPHIC(1) （468 或 469）	wchar_t	• 单一宽字符（用于 C 类型） • 单一双字节字符（用于列类型）
GRAPHIC(*n*) （468 或 469）	没有精确的等同类型；请使用 wchar_t [*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=127	双字节定长字符串

VARGRAPHIC(*n*)
（464 或 465）
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	1<=*n*<=16 336
		带有 2 字节字符串长度指示符而且并非以 null 结束的可变长度双字节字符串

	另外，也可以使用 char[*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=16 336	以 null 结束的可变长度双字节字符串 **注**: 被指定给 SQL 类型 400/401。

表 4. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述

LONG VARGRAPHIC[8]
（472 或 473）
	struct tag {
	short int;
	wchar_t [*n*]
	}
	
	16 337<=*n*<=16 350
		带有 2 字节字符串长度指示符而且并非以 null 结束的可变长度双字节字符串

只有在 DBCS 或 EUC 环境中，下列数据类型才可用。

表 5. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
DBCLOB(*n*) （412 或 413）	SQL 类型是 dbclob(*n*) 1<=*n*<=1 073 741 823	带有 4 字节字符串长度指示符而且并非以 null 结束的可变长度双字节字符串
DBCLOB 定位器变量[7] （968 或 969）	SQL 类型是 dbclob_locator	标识驻留在服务器上的 DBCLOB 实体
DBCLOB 文件引用变量[7] （924 或 925）	SQL 类型是 dbclob_file	包含 DBCLOB 数据的文件的描述符

表 5. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述

注:

1. **SQL 列类型**下的第一个数字表示未提供指示符变量，第二个数字表示提供了指示符变量。指示符变量用于指示 NULL 值或者用于存放已截断的字符串的长度。这些是 SQLDA 的 SQLTYPE 字段对于这些数据类型将包含的值。
2. 为了实现平台兼容性，请使用 sqlint32。在 64 位 UNIX 和 Linux 操作系统上，“long”是 64 位整数。在 64 位 Windows 操作系统以及 32 位 UNIX 和 Linux 操作系统上，“long”是 32 位整数。
3. 为了实现平台兼容性，请使用 sqlint64。在受支持的 Windows 操作系统上使用 Microsoft 编译器时，DB2 数据库系统 sqlsystm.h 头文件具有 sqlint64 的类型定义“__int64”，在 32 位 UNIX 和 Linux 操作系统上定义为“long long”，在 64 位 UNIX 和 Linux 操作系统上定义为“long”。
4. 字符串的长度可以是 19 - 32 字节（不带 null 终止符），这取决于指定的秒小数位数。（可选）可以指定 TIMESTAMP 数据类型的小数秒部分，以使时间戳记精度为 0-12 位。

   将时间戳记值指定给秒小数位数不同的时间戳记变量时，该值将被截断或者填充 0 以便与时间戳记变量的格式匹配。
5. FLOAT(*n*)（其中，0 < *n* < 25）是 REAL 的同义词。在 SQLDA 中，REAL 与 DOUBLE 之间的差别是长度值（4 或 8）。
6. 下列 SQL 类型是 DOUBLE 的同义词:
   - FLOAT
   - FLOAT(*n*)（其中，24 < *n* < 54）是 DOUBLE 的同义词
   - DOUBLE PRECISION
7. 这不是列类型，而是主变量类型。
8. 只有 DESCRIBE 请求才会返回 SQL_TYP_XML/SQL_TYP_NXML 值。应用程序不能直接使用此值将应用程序资源与 XML 值绑定。
9. 建议您不要使用 LONG VARCHAR 和 LONG VARGRAPHIC 数据类型，在将来的发行版中，可能会除去这些数据类型。请改为选择 CLOB 或 DBCLOB 数据类型。

以下各项目是有关受支持的 C 和 C++ 数据类型的其他规则:

- 可以将数据类型 `char` 声明为 `char` 或 `unsigned char`。
- 数据库管理器将以 null 结束的变长字符串数据类型 char[*n*]（数据类型 460）作为 VARCHAR(*m*) 进行处理。
  - 如果 LANGLEVEL 为 SAA1，那么主变量长度 *m* 等于 char[*n*] 中的字符串长度 *n* 或者第一个 null 终止符（`\0`）之前的字节数（以较小者为准）。
  - 如果 LANGLEVEL 为 MIA，那么主变量长度 *m* 等于第一个 null 终止符（`\0`）之前的字节数。
- 数据库管理器将以 null 结束的可变长度图形字符串数据类型 `wchar_t[n]` 或 `sqldbchar[n]`（数据类型 400®）作为 VARGRAPHIC(*m*) 进行处理。
  - 如果 LANGLEVEL 为 SAA1，那么主变量长度 *m* 等于 `wchar_t[n]` 或 `sqldbchar[n]` 中的字符串长度 *n* 或者第一个图形 null 终止符之前的字符数（以较小者为准）。
  - 如果 LANGLEVEL 为 MIA，那么主变量长度 *m* 等于第一个图形 null 终止符之前的字符数。
- 不支持无符号数字数据类型。
- 由于 C 和 C++ 数据类型 `int` 的内部表示依赖于机器，因此不允许使用此数据类型。

C 和 C++ 嵌入式 SQL 应用程序中过程、函数和方法的数据类型

下表列示过程、UDF 和方法的 SQL 数据类型与 C 和 C++ 数据类型之间的受支持映射。

表 6. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
SMALLINT （500 或 501）	short	16 位带符号整数
INTEGER （496 或 497）	sqlint32	32 位带符号整数
BIGINT （492 或 493）	sqlint64	64 位带符号整数
REAL （480 或 481）	float	单精度浮点数
DOUBLE （480 或 481）	double	双精度浮点数
DECIMAL(*p*,*s*) （484 或 485）	不受支持	要传递十进制值，请将参数定义为能够从 DECIMAL 强制转换的数据类型（例如 CHAR 或 DOUBLE）并显式地将自变量强制转换为此类型。
CHAR(*n*) （452 或 453）	char[*n*+*1*]，其中 n 的大小足以存放数据 1<=*n*<=254	以 null 结束的定长字符串
CHAR(*n*) FOR BIT DATA （452 或 453）	char[*n*+*1*]，其中 n 的大小足以存放数据 1<=*n*<=254	定长字符串
VARCHAR(*n*) （448 或 449）（460 或 461）	char[*n*+*1*]，其中 n 的大小足以存放数据 1<=*n*<=32 672	以 null 结束的变长字符串

VARCHAR(*n*) FOR BIT DATA
（448 或 449）
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
		并非以 null 结束的变长字符串

表 6. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述

LONG VARCHAR[2]
（456 或 457）
	struct {
	sqluint16 length;
	char[*n*]
	}
	
	32 673<=*n*<=32 700
		并非以 null 结束的变长字符串

CLOB(*n*)
（408 或 409）
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		带有 4 字节字符串长度指示符而且并非以 null 结束的变长字符串

BLOB(*n*)
（404 或 405）
	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
		带有 4 字节字符串长度指示符而且并非以 null 结束的变长二进制字符串

DATE （384 或 385）	char[11]	以 null 结束的字符格式
TIME （388 或 389）	char[9]	以 null 结束的字符格式
TIMESTAMP(*p*) （392 或 393）	char[*p+21*]，其中 p 的大小足以存放数据 0<=*p*<=12	以 null 结束的字符格式
XML （988/989）	不受支持	将定义此描述符类型值（988/989），以便在 SQLDA 中用于描述和指示序列化格式的 XML 数据。另外，还可以使用现有的字符和二进制类型（包括 LOB 和 LOB 文件引用类型）来提取和插入数据（仅限于动态 SQL）

注: 只有在 DBCS 或 EUC 环境中使用 WCHARTYPE NOCONVERT 选项进行预编译的情况下，下列数据类型才可用。

表 7. 映射到 C 和 C++ 声明的 SQL 数据类型

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
GRAPHIC(*n*) （468 或 469）	sqldbchar[*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=127	以 null 结束的双字节定长字符串

表 7. 映射到 C 和 C++ 声明的 SQL 数据类型 (续)

SQL 列类型[1]	C 和 C++ 数据类型	SQL 列类型描述
VARGRAPHIC(*n*) （400 或 401）	sqldbchar[*n+1*]，其中 n 的大小足以存放数据 1<=*n*<=16 336	并非以 null 结束的可变长度双字节字符串

LONG VARGRAPHIC[2]
（472 或 473）
	struct {
	sqluint16 length;
	sqldbchar[*n*]
	}
	
	16 337<=*n*<=16 350
		并非以 null 结束的可变长度双字节字符串

DBCLOB(*n*)
（412 或 413）
	struct {
	sqluint32 length;
	sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
		带有 4 字节字符串长度指示符而且并非以 null 结束的变长字符串

注:

1. **SQL 列类型**下的第一个数字表示未提供指示符变量，第二个数字表示提供了指示符变量。指示符变量用于指示 NULL 值或者用于存放已截断的字符串的长度。这些是 SQLDA 的 SQLTYPE 字段对于这些数据类型将包含的值。
2. 建议您不要使用 LONG VARCHAR 和 LONG VARGRAPHIC 数据类型，在将来的发行版中，可能会除去这些数据类型。请改为选择 CLOB 或 DBCLOB 数据类型。

COBOL 嵌入式 SQL 应用程序中的受支持 SQL 数据类型

某些预定义的 COBOL 数据类型与 DB2 数据库列类型相对应。只有这些 COBOL 数据类型可以被声明为主变量。

下表列示每种列类型的等同 COBOL 类型。当预编译器找到主变量声明时，它确定适当的 SQL 类型值。数据库管理器使用此值对它自己与应用程序之间交换的数据进行转换。

并非主变量的每种可能数据描述都能被识别。COBOL 数据项必须与下表中描述的数据项一致。如果使用其他数据项，那么可能会发生错误。

表 8. 映射到 COBOL 声明的 SQL 数据类型

SQL 列类型[1]	COBOL 数据类型	SQL 列类型描述
SMALLINT （500 或 501）	01 名称 PIC S9(4) COMP-5.	16 位带符号整数
INTEGER （496 或 497）	01 名称 PIC S9(9) COMP-5.	32 位带符号整数
BIGINT （492 或 493）	01 名称 PIC S9(18) COMP-5.	64 位带符号整数

表 8. 映射到 COBOL 声明的 SQL 数据类型 (续)

SQL 列类型[1]	COBOL 数据类型	SQL 列类型描述
DECIMAL(*p*,*s*) （484 或 485）	01 名称 PIC S9(*m*)V9(*n*) COMP-3.	压缩十进制
REAL[2] （480 或 481）	01 名称 USAGE IS COMP-1.	单精度浮点数
DOUBLE[3] （480 或 481）	01 名称 USAGE IS COMP-2.	双精度浮点数
CHAR(*n*) （452 或 453）	01 名称 PIC X(*n*).	定长字符串
VARCHAR(*n*) （448 或 449）	01 名称. 49 长度 PIC S9(4) COMP-5. 49 名称 PIC X(*n*). 1<=*n*<=32 672	变长字符串
LONG VARCHAR[6] （456 或 457）	01 名称. 49 长度 PIC S9(4) COMP-5. 49 数据 PIC X(*n*). 32 673<=*n*<=32 700	可变长度长字符串
CLOB(*n*) （408 或 409）	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n). 1<=*n*<=2 147 483 647	大对象变长字符串
CLOB 定位器变量[4] （964 或 965）	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.	标识驻留在服务器上的 CLOB 实体
CLOB 文件引用变量[4] （920 或 921）	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.	包含 CLOB 数据的文件的描述符
BLOB(*n*) （404 或 405）	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n). 1<=*n*<=2 147 483 647	大对象变长二进制字符串
BLOB 定位器变量[4] （960 或 961）	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.	标识驻留在服务器上的 BLOB 实体
BLOB 文件引用变量[4] （916 或 917）	01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.	包含 BLOB 数据的文件的描述符
DATE （384 或 385）	01 标识 PIC X(10).	10 字节字符串

表 8. 映射到 COBOL 声明的 SQL 数据类型 (续)

SQL 列类型[1]	COBOL 数据类型	SQL 列类型描述
TIME （388 或 389）	01 标识 PIC X(8).	8 字节字符串
TIMESTAMP(*p*) （392 或 393）	01 标识 PIC X(*p*+*20*). 0<=*p*<=12	19 到 32 字节字符串 *p* 为 0 时，可以使用 19 个字节的字符串。
XML[5] （988 或 989）	01 名称 USAGE IS SQL TYPE IS XML AS CLOB (大小).	XML 值

只有在 DBCS 环境中，下列数据类型才可用。

表 9. 映射到 COBOL 声明的 SQL 数据类型

SQL 列类型[1]	COBOL 数据类型	SQL 列类型描述
GRAPHIC(*n*) （468 或 469）	01 名称 PIC G(*n*) DISPLAY-1.	双字节定长字符串
VARGRAPHIC(*n*) （464 或 465）	01 名称. 49 长度 PIC S9(4) COMP-5. 49 名称 PIC G(*n*) DISPLAY-1. 1<=*n*<=16 336	带有 2 字节字符串长度指示符的可变长度双字节字符串
LONG VARGRAPHIC[6] （472 或 473）	01 名称. 49 长度 PIC S9(4) COMP-5. 49 名称 PIC G(*n*) DISPLAY-1. 16 337<=*n*<=16 350	带有 2 字节字符串长度指示符的可变长度双字节字符串
DBCLOB(*n*) （412 或 413）	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB (n). 1<=*n*<=1 073 741 823	带有 4 字节字符串长度指示符的大对象可变长度双字节字符串
DBCLOB 定位器变量[4] （968 或 969）	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR.	标识驻留在服务器上的 DBCLOB 实体
DBCLOB 文件引用变量[4] （924 或 925）	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.	包含 DBCLOB 数据的文件的描述符

表 9. 映射到 COBOL 声明的 SQL 数据类型 (续)

SQL 列类型[1]	COBOL 数据类型	SQL 列类型描述

注:

1. **SQL 列类型**下的第一个数字表示未提供指示符变量，第二个数字表示提供了指示符变量。指示符变量用于指示 NULL 值或者用于存放已截断的字符串的长度。这些是 SQLDA 的 SQLTYPE 字段对于这些数据类型将包含的值。
2. FLOAT(*n*)（其中，0 < *n* < 25）是 REAL 的同义词。在 SQLDA 中，REAL 与 DOUBLE 之间的差别是长度值（4 或 8）。
3. 下列 SQL 类型是 DOUBLE 的同义词:
   - FLOAT
   - FLOAT(*n*)（其中，24 < *n* < 54）是 DOUBLE 的同义词。
   - DOUBLE PRECISION
4. 这不是列类型，而是主变量类型。
5. 只有 DESCRIBE 请求才会返回 SQL_TYP_XML/SQL_TYP_NXML 值。应用程序不能直接使用此值将应用程序资源与 XML 值绑定。
6. 建议您不要使用 LONG VARCHAR 和 LONG VARGRAPHIC 数据类型，在将来的发行版中，可能会除去这些数据类型。请改为选择 CLOB 或 DBCLOB 数据类型。

用于所支持 COBOL 数据类型的规则列表是:

- 在列示了 PIC S9 和 COMP-3/COMP-5 的位置，它们是必需的。
- 对于除 VARCHAR、LONG VARCHAR、VARGRAPHIC、LONG VARGRAPHIC 以及所有 LOB 变量类型以外的所有列类型，可以使用级别号 77 来代替 01。
- 为 DECIMAL(p,s) 列类型声明主变量时，请使用下列规则。请参阅以下样本:

  ```
  01 标识 PIC S9(m)V9(n) COMP-3
  ```

  – 使用 V 来指示小数点。
  – *n* 和 *m* 的值必须大于或等于 1。
  – *n* + *m* 的值不能超过 31。
  – *s* 的值等于 *n* 的值。
  – *p* 的值等于 *n* + *m* 的值。
  – 重复系数 *(n)* 和 *(m)* 是可选的。下列示例全部有效:

  ```
  01 标识 PIC S9(3)V COMP-3
  01 标识 PIC SV9(3) COMP-3
  01 标识 PIC S9V COMP-3
  01 标识 PIC SV9 COMP-3
  ```

  – 可以使用 PACKED-DECIMAL 来代替 COMP-3。
- COBOL 预编译器不支持数组。

FORTRAN 嵌入式 SQL 应用程序中的受支持 SQL 数据类型

某些预定义的 FORTRAN 数据类型与 DB2 数据库列类型相对应。只有这些 FORTRAN 数据类型可以被声明为主变量。

下表列示每种列类型的等同 FORTRAN 类型。当预编译器找到主变量声明时，它确定适当的 SQL 类型值。数据库管理器使用此值对它自己与应用程序之间交换的数据进行转换。

表 10. 映射到 FORTRAN 声明的 SQL 数据类型

SQL 列类型[1]	FORTRAN 数据类型	SQL 列类型描述
SMALLINT （500 或 501）	INTEGER*2	16 位带符号整数
INTEGER （496 或 497）	INTEGER*4	32 位带符号整数
REAL[2] （480 或 481）	REAL*4	单精度浮点数
DOUBLE[3] （480 或 481）	REAL*8	双精度浮点数
DECIMAL(*p*,*s*) （484 或 485）	没有精确的等同类型；请使用 REAL*8	压缩十进制
CHAR(*n*) （452 或 453）	CHARACTER**n*	长度为 *n* 的定长字符串，其中 *n* 为 1 到 254
VARCHAR(*n*) （448 或 449）	SQL 类型为 VARCHAR(*n*)，其中 *n* 是 1 到 32672	变长字符串
LONG VARCHAR[5] （456 或 457）	SQL 类型为 VARCHAR(*n*)，其中 *n* 是 32673 到 32700	可变长度长字符串
CLOB(*n*) （408 或 409）	SQL 类型为 CLOB (n)，其中 *n* 是 1 到 2147483647	大对象变长字符串
CLOB 定位器变量[4] （964 或 965）	SQL 类型为 CLOB_LOCATOR	标识驻留在服务器上的 CLOB 实体
CLOB 文件引用变量[4] （920 或 921）	SQL 类型为 CLOB_FILE	包含 CLOB 数据的文件的描述符
BLOB(*n*) （404 或 405）	SQL 类型为 BLOB(n)，其中 *n* 是 1 到 2147483647	大对象变长二进制字符串
BLOB 定位器变量[4] （960 或 961）	SQL 类型为 BLOB_LOCATOR	标识服务器上的 BLOB 实体
BLOB 文件引用变量[4] （916 或 917）	SQL 类型为 BLOB_FILE	包含 BLOB 数据的文件的描述符
DATE （384 或 385）	CHARACTER*10	10 字节字符串
TIME （388 或 389）	CHARACTER*8	8 字节字符串

表 10. 映射到 FORTRAN 声明的 SQL 数据类型 (续)

SQL 列类型[1]	FORTRAN 数据类型	SQL 列类型描述
TIMESTAMP(*p*) （392 或 393）	CHARACTER*19 到 CHARACTER*32	19 到 32 字节字符串
XML （988 或 989）	SQL_TYP_XML	FORTRAN 不支持 XML；应用程序能够重新获取所描述的类型，但将无法使用该类型。

注:

1. **SQL 列类型**下的第一个数字表示未提供指示符变量，第二个数字表示提供了指示符变量。指示符变量用于指示 NULL 值或者用于存放已截断的字符串的长度。这些是 SQLDA 的 SQLTYPE 字段对于这些数据类型将包含的值。
2. FLOAT(*n*)（其中，$0 < n < 25$）是 REAL 的同义词。在 SQLDA 中，REAL 与 DOUBLE 之间的差别是长度值（4 或 8）。
3. 下列 SQL 类型是 DOUBLE 的同义词:
   - FLOAT
   - FLOAT(*n*)（其中，$24 < n < 54$）是 DOUBLE 的同义词。
   - DOUBLE PRECISION
4. 这不是列类型，而是主变量类型。
5. 建议您不要使用 LONG VARCHAR 数据类型，在将来的发行版中，可能会除去此数据类型。请改为选择 CLOB 数据类型。

受支持的 FORTRAN 数据类型的规则是:

- 通过使用 VARCHAR 或 CLOB 主变量，可以定义长度超过 254 个字符的动态 SQL 语句。

REXX 嵌入式 SQL 应用程序中的受支持 SQL 数据类型

某些预定义的 REXX 数据类型与 DB2 数据库列类型相对应。只有这些 REXX 数据类型可以被声明为主变量。下表说明 SQLEXEC 和 SQLDBS 如何解释 REXX 变量以便将它们的内容转换为 DB2 数据类型。

表 11. 映射到 REXX 声明的 SQL 列类型

SQL 列类型[1]	REXX 数据类型	SQL 列类型描述
SMALLINT （500 或 501）	不带小数点的数字，范围为 -32768 到 32767	16 位带符号整数
INTEGER （496 或 497）	不带小数点的数字，范围为 -2147483648 到 2147483647	32 位带符号整数
REAL[2] （480 或 481）	采用科学记数法的数字，范围为 $-3.40282346 \times 10^{38}$ 到 $3.40282346 \times 10^{38}$	单精度浮点数
DOUBLE[3] （480 或 481）	采用科学记数法的数字，范围为 $-1.79769313 \times 10^{308}$ 到 $1.79769313 \times 10^{308}$	双精度浮点数
DECIMAL(*p*,*s*) （484 或 485）	带有小数点的数字	压缩十进制

表 11. 映射到 REXX 声明的 SQL 列类型 (续)

SQL 列类型[1]	REXX 数据类型	SQL 列类型描述
CHAR(*n*) （452 或 453）	带有前导和尾部引号（'）的字符串，在除去两个引号后，长度为 *n* 带有任何非数字字符（前导和尾部空格或者科学记数法中的 E 除外）并且长度为 *n* 的字符串	长度为 *n* 的定长字符串，其中 *n* 为 1 到 254
VARCHAR(*n*) （448 或 449）	等同于 CHAR(*n*)	长度为 *n* 的变长字符串，其中 *n* 的范围是 1 到 4000
LONG VARCHAR[5] （456 或 457）	等同于 CHAR(*n*)	长度为 *n* 的变长字符串，其中 *n* 的范围是 1 到 32700
CLOB(*n*) （408 或 409）	等同于 CHAR(*n*)	长度为 *n* 的大对象变长字符串，其中 *n* 的范围是 1 到 2147483647
CLOB 定位器变量[4] （964 或 965）	DECLARE *:变量名* LANGUAGE TYPE CLOB LOCATOR	标识驻留在服务器上的 CLOB 实体
CLOB 文件引用变量[4] （920 或 921）	DECLARE *:变量名* LANGUAGE TYPE CLOB FILE	包含 CLOB 数据的文件的描述符
BLOB(*n*) （404 或 405）	以 BIN 为前缀并带有前导和尾部撇号的字符串，除去前导 BIN 和两个撇号后，包含 *n* 个字符。	长度为 *n* 的大对象可变长度二进制字符串，其中 *n* 的范围是 1 到 2147483647
BLOB 定位器变量[4] （960 或 961）	DECLARE *:变量名* LANGUAGE TYPE BLOB LOCATOR	标识服务器上的 BLOB 实体
BLOB 文件引用变量[4] （916 或 917）	DECLARE *:变量名* LANGUAGE TYPE BLOB FILE	包含 BLOB 数据的文件的描述符
DATE （384 或 385）	等同于 CHAR(*10*)	10 字节字符串
TIME （388 或 389）	等同于 CHAR(*8*)	8 字节字符串
TIMESTAMP （392 或 393）	等同于 CHAR(*26*)	26 字节字符串
XML （988 或 989）	SQL_TYP_XML	REXX 不支持 XML；应用程序能够重新获取所描述的类型，但将无法使用该类型。

只有在 DBCS 环境中，下列数据类型才可用。

表 12. 映射到 REXX 声明的 SQL 列类型

SQL 列类型[1]	REXX 数据类型	SQL 列类型描述
GRAPHIC(*n*) （468 或 469）	以 G 或 N 为前缀并带有前导和尾部撇号的字符串，除去前导字符和两个撇号后，包含 *n* 个 DBCS 字符	长度为 *n* 的定长图形字符串，其中 *n* 为 1 到 127
VARGRAPHIC(*n*) （464 或 465）	等同于 GRAPHIC(*n*)	长度为 *n* 的变长图形字符串，其中 *n* 的范围是 1 到 2000
LONG VARGRAPHIC[5] （472 或 473）	等同于 GRAPHIC(*n*)	长度为 *n* 的可变长度长图形字符串，其中 *n* 的范围是 1 到 16350
DBCLOB(*n*) （412 或 413）	等同于 GRAPHIC(*n*)	长度为 *n* 的大对象变长图形字符串，其中 *n* 的范围是 1 到 1073741823
DBCLOB 定位器变量[4] （968 或 969）	DECLARE *:变量名* LANGUAGE TYPE DBCLOB LOCATOR	标识驻留在服务器上的 DBCLOB 实体
DBCLOB 文件引用变量[4] （924 或 925）	DECLARE *:变量名* LANGUAGE TYPE DBCLOB FILE	包含 DBCLOB 数据的文件的描述符

注:

1. **列类型**下的第一个数字表示未提供指示符变量，第二个数字表示提供了指示符变量。指示符变量用于指示 NULL 值或者用于存放已截断的字符串的长度。
2. FLOAT(*n*)（其中，$0 < n < 25$）是 REAL 的同义词。在 SQLDA 中，REAL 与 DOUBLE 之间的差别是长度值（4 或 8）。
3. 下列 SQL 类型是 DOUBLE 的同义词:
   - FLOAT
   - FLOAT(*n*)（其中，$24 < n < 54$）是 DOUBLE 的同义词。
   - DOUBLE PRECISION
4. 这不是列类型，而是主变量类型。
5. 建议您不要使用 LONG VARCHAR 和 LONG VARGRAPHIC 数据类型，在将来的发行版中，可能会除去这些数据类型。请改为使用 CLOB 或 DBCLOB 数据类型。

嵌入式 SQL 应用程序中的主变量

*主变量*是嵌入式 SQL 语句所引用的变量。这些变量用于在数据库服务器与嵌入式 SQL 应用程序之间交换数据值。嵌入式 SQL 应用程序还可以包含关系 SQL 查询的主变量声明。并且，主变量可用于包含所要执行的 XQuery 表达式。但是，没有任何机制可以将值传递到 XQuery 表达式中的参数。

请使用声明节中随主语言不同而有所变化的变量声明语法来声明主变量。

声明节是嵌入式 SQL 源代码文件顶部附近的嵌入式 SQL 应用程序部分，它由两个不可执行的 SQL 语句限定:

- BEGIN DECLARE SECTION
- END DECLARE SECTION

这些语句使预编译器能够找到变量声明。每个主变量声明都必须在这两个语句之间使用，否则那些变量将被视为仅仅是常规变量。

下列规则适用于主变量声明节:

- 除引用 SQLDA 结构的主变量以外，所有主变量在被引用前，都必须在源文件中结构良好的声明节中进行声明。
- 在一个源文件中，可以有多个声明节。
- 主变量的名称在源文件中必须唯一。这是因为，DB2 预编译器并不考虑随主语言不同而有所变化的变量作用域限定规则。因此，主变量只有一个作用域。

  注: 这并不表示 DB2 预编译器会将主变量的作用域更改为全局作用域以使它们在其定义所在的作用域外部可访问。

请考虑以下示例:

```
foo1(){
  .
  .
  .
  BEGIN SQL DECLARE SECTION;
  int x;
  END SQL DECLARE SECTION;
x=10;
  .
  .
  .
}


foo2(){
  .
  .
  .
  y=x;
  .
  .
  .
}
```

根据语言不同，此示例将无法通过编译，这是因为，在函数 `foo2()` 中未声明变量 `x`，或者未在 `foo2()` 中将 `x` 的值设置为 10。为了避免此问题，必须将 `x` 声明为全局变量，或者将 `x` 作为参数传递给函数 `foo2()`，如下所示:

```
  foo1(){
  .
  .
  .
    BEGIN SQL DECLARE SECTION;
    int x;
    END SQL DECLARE SECTION;
    x=10;
    foo2(x);
  .
  .
  .
  }


  foo2(int x){
  .
  .
  .
```

```
  y=x;
.
.
.
}
```

在嵌入式 SQL 应用程序中声明主变量

要在数据库服务器与应用程序之间传输数据，请在应用程序源代码中为关系 SQL 查询以及 XQuery 表达式的主变量声明之类的内容声明主变量。

关于此任务

下表提供嵌入式 SQL 主语言的主变量声明示例。

表 13. 主变量声明（按主语言排列）

语言	示例源代码

C 和 C++	`EXEC SQL BEGIN DECLARE SECTION;`
	`  short  dept=38, age=26;`
	`  double  salary;`
	`  char  CH;`
	`  char  name1[9], NAME2[9];`
	`  short  nul_ind;`
	`EXEC SQL END DECLARE SECTION;`

COBOL	`EXEC SQL BEGIN DECLARE SECTION END-EXEC.`
	`  01 age  PIC S9(4) COMP-5 VALUE 26.`
	`  01 DEPT  PIC S9(9) COMP-5 VALUE 38.`
	`  01 salary  PIC S9(6)V9(3) COMP-3.`
	`  01 CH  PIC X(1).`
	`  01 name1  PIC X(8).`
	`  01 NAME2  PIC X(8).`
	`  01 nul-ind  PIC S9(4) COMP-5.`
	`EXEC SQL END DECLARE SECTION END-EXEC.`

FORTRAN	`EXEC SQL BEGIN DECLARE SECTION`
	`  integer*2  age  /26/`
	`  integer*4  dept  /38/`
	`  real*8  salary`
	`  character  ch`
	`  character*8  name1,NAME2`
	`  integer*2  nul_ind`
	`EXEC SQL END DECLARE SECTION`

使用 db2dclgn 声明生成器来声明主变量

您可以使用声明生成器来生成数据库中给定表的声明。它将创建嵌入式 SQL 声明源文件，您可以方便地将该文件插入到应用程序中。db2dclgn 支持 C/C++、Java、COBOL 和 FORTRAN 语言。

关于此任务

要生成声明文件，请使用以下格式来输入 db2dclgn 命令:

```
db2dclgn -d 数据库名 -t 表名 [选项]
```

例如，要以 C 格式在输出文件 staff.h 中生成 SAMPLE 数据库中 STAFF 表的声明，请发出以下命令:

```
db2dclgn -d sample -t staff -l C
```

所生成的 staff.h 文件包含以下内容:

```
struct
{
  short id;
  struct
  {
    short length;
    char data[9];
  } name;
  short dept;
  char job[6];
  short years;
  double salary;
  double comm;
} staff;
```

嵌入式 SQL 应用程序中的列数据类型和主变量

每个 DB2 表中的每个列都在该列创建期间被指定 *SQL 数据类型*。有关如何对列指定这些类型的信息，请参阅 CREATE TABLE 语句。

注:

1. 每种受支持的数据类型都可以具有 NOT NULL 属性。这将被视为另一类型。
2. 您可以通过定义用户定义的单值类型（UDT）对数据类型进行扩展。UDT 是使用了其中一种内置 SQL 类型的表示的独立数据类型。

受支持的嵌入式 SQL 主语言具有与大部分数据库管理器数据类型对应的数据类型。在主变量声明中，只能使用这些主语言数据类型。当预编译器找到主变量声明时，它确定适当的 SQL 数据类型值。数据库管理器使用此值对它自己与应用程序之间交换的数据进行转换。

作为应用程序员，了解数据库管理器如何处理不同数据类型之间的比较和赋值十分重要。简而言之，数据类型在赋值和比较操作期间必须相互兼容，而无论数据库管理器是正在处理两种 SQL 列数据类型、两种主语言数据类型还是一种 SQL 列数据类型和一种主语言数据类型。

数据类型兼容性的一般规则是，所有受支持的主语言数字数据类型与所有数据库管理器数字数据类型相容并且赋值兼容，而所有主语言字符类型与所有数据库管理器字符类型相容；数字类型与字符类型不相容。但是，这个一般规则也有例外情况，这取决于主语言特性以及处理大对象时实施的限制。

在 SQL 语句中，DB2 将在兼容的数据类型之间进行转换。例如，在以下 SELECT 语句中，SALARY 和 BONUS 是 DECIMAL 列；但是，每位职员的薪水总计都作为 DOUBLE 数据返回:

```
SELECT EMPNO, DOUBLE(SALARY+BONUS) FROM EMPLOYEE
```

注意，执行此语句时，将在 DECIMAL 与 DOUBLE 数据类型之间进行转换。

要使查询结果在屏幕上更为易读，可以使用以下 SELECT 语句:

```
SELECT EMPNO, CHAR(SALARY+BONUS) FROM EMPLOYEE
```

以上示例中使用的 CAST 函数将返回数字的字符串表示。

要在应用程序中转换数据，请与编译器供应商联系，以获取其他支持此转换的例程、类、内置类型或 API。

即使应用程序代码页与数据库代码页不同，也可以对字符数据类型进行字符转换。

在嵌入式 SQL 应用程序中声明 XML 主变量

要在数据库服务器与嵌入式 SQL 应用程序之间交换 XML 数据，需要在应用程序源代码中声明主变量。

关于此任务

DB2 V9.1 引入了以树型格式在结构化节点集中存储 XML 数据的 XML 数据类型。具有此 XML 数据类型的列被描述为 SQL_TYP_XML 列 SQLTYPE，应用程序可以绑定各种特定于语言的数据类型，以便输入到这些列或参数以及从这些列或参数输出。您可以使用 SQL、SQL/XML 扩展或 XQuery 来直接访问 XML 列。XML 数据类型不仅仅适用于列。函数也可以具有 XML 值自变量以及生成 XML 值。同样，存储过程可以将 XML 值用作输入和输出参数。最后，XQuery 表达式生成 XML 值，这与它们是否访问 XML 列无关。

XML 数据的性质是字符，它的编码指定所使用的字符集。可以从包含 XML 文档的序列化字符串表示的基本应用程序类型来派生 XML 数据的编码，从而以外部方式确定此编码。另外，也可以通过内部方式来确定此编码，这要求解释数据。对于 Unicode 编码文档而言，建议使用由数据流开头的 Unicode 字符代码组成的字节顺序标记（BOM）。BOM 被用作特征符，用于定义字节顺序和 Unicode 编码格式。

除了使用 XML 主变量以外，还可以使用现有字符和二进制类型（包括 CHAR、VARCHAR、CLOB 和 BLOB）来提取和插入数据。但是，与 XML 主变量不同，它们不能用于隐式 XML 解析。而是，将插入并应用执行缺省空格删除操作的显式 XMLPARSE 函数。

有关开发嵌入式 SQL 应用程序的 XML 和 XQuery 限制

要在嵌入式 SQL 应用程序中声明 XML 主变量，请执行下列操作:

在应用程序的声明节中，将 XML 主变量声明为 LOB 数据类型:

- ```
  SQL TYPE IS XML AS CLOB(n) <主变量名>
  ```

  其中，<hostvar_name> 是一个 CLOB 主变量，它包含以应用程序的混合代码页编码的 XML 数据。

- ```
  SQL TYPE IS XML AS DBCLOB(n) <主变量名>
  ```

  其中，<hostvar_name> 是一个 DBCLOB 主变量，它包含以应用程序图形代码页编码的 XML 数据。

- ```
  SQL TYPE IS XML AS BLOB(n) <主变量名>
  ```

  其中，<主变量名> 是一个 BLOB 主变量，它包含以内部方式编码的 XML 数据[1]。

-
  ```

SQL TYPE IS XML AS CLOB_FILE <主变量名>

其中，<hostvar_name> 是一个 CLOB 文件，它包含以应用程序混合代码页编码的 XML 数据。

- SQL TYPE IS XML AS DBCLOB_FILE <主变量名>

  其中，<hostvar_name> 是一个 DBCLOB 文件，它包含以应用程序图形代码页编码的 XML 数据。

- SQL TYPE IS XML AS BLOB_FILE <主变量名>

  其中，<主变量名> 是一个 BLOB 文件，它包含以内部方式编码的 XML 数据[1]。

注:

1. 请参阅用于通过 XML 1.0 规范确定编码的算法（http://www.w3.org/TR/REC-xml/#sec-guessing-no-ext-info）。

在 SQLDA 中标识 XML 值

要指示基本类型存放 XML 数据，必须按如下方式更新 SQLVAR 的 sqlname 字段:

- sqlname.length 必须是 8
- sqlname.data 的前两个字节必须是 X'0000'
- sqlname.data 的第三个和第四个字节必须是 X'0000'
- sqlname.data 的第五个字节必须是 X'01'（仅当符合前两个条件时，此字节才被称为 XML 子类型指示符）
- 其余字节必须是 X'000000'

如果在 SQLTYPE 不是 LOB 的 SQLVAR 中设置 XML 子类型指示符，那么在运行时将返回 SQL0804 错误（RC 为 115）。

注: 只能从 DESCRIBE 语句中返回 SQL_TYP_XML。此类型不能用于任何其他请求。应用程序必须将 SQLDA 修改为包含有效的字符或二进制类型，并且必须适当地设置 sqlname 字段以指示数据为 XML。

使用 null 指示符变量来标识 null SQL 值

关于此任务

嵌入式 SQL 应用程序必须通过使 *null 指示符变量*与任何能够接收 null 的主变量相关联以准备接收 null 值。null 指示符变量由数据库管理器和主应用程序共享。因此，必须在应用程序中将此变量声明为与 SQL 数据类型 SMALLINT 相对应的主变量。

null 指示符变量紧跟在 SQL 语句中的主变量之后并以冒号为前缀。可以用空格将 null 指示符变量与主变量分隔开，但此分隔并非必需。但是，请不要在主变量与 null 指示符变量之间加入逗号。还可以通过在主变量及其 null 指示符之间放置可选的 INDICATOR 关键字来指定 null 指示符变量。

系统将检查 null 指示符变量是否为负数值。如果值不是负数，那么应用程序可以使用所返回的主变量值。如果值是负数，那么表明提取到的值为 null，因此不应使用该主变量。在这种情况下，数据库管理器不会更改该主变量的值。

注: 如果数据库配置参数 *dft_sqlmathwarn* 设置为“YES”，那么 null 指示符变量值可能是 -2。此值指示 null，这是因为对表达式进行求值时发生算术错误，或者尝试将数字结果值转换为主变量时发生溢出。

如果数据类型能够处理 null，那么应用程序必须提供 null 指示符。否则，可能会发生错误。如果未使用 null 指示符，那么将返回 SQLCODE -305（SQLSTATE 22002）。

如果 SQLCA 结构指示截断警告，那么可以检查 null 指示符变量以确定是否发生截断。如果 null 指示符变量包含正数值，那么表示已发生截断。

- 如果已截断 TIME 数据类型的秒部分，那么 null 指示符值将包含被截断的数据的秒部分。
- 对于除大对象（LOB）以外的所有其他字符串数据类型而言，null 指示符值表示所返回的数据的实际长度。用户定义的单值类型（UDT）的处理方式与其基本类型的处理方式相同。

处理 INSERT 或 UPDATE 语句时，数据库管理器将检查 null 指示符变量（如果存在的话）。如果此指示符变量的值为负数，那么数据库管理器将把目标列值设置为 null（如果允许使用 null 的话）。

如果 null 指示符变量为零或正数，那么数据库管理器将使用相关联主变量的值。

如果将字符串列的值赋予主变量时该值被截断，那么 SQLCA 结构中的 SQLWARN1 字段可能包含 X 或 W。如果 null 终止符被截断，那么此字段将包含 N。

仅当满足下列所有条件时，数据库管理器才返回值 X:

- 存在混合代码页连接，即，将字符串数据从数据库代码页转换到应用程序代码页时涉及更改数据长度。
- 游标是分块游标。
- null 指示符变量由应用程序提供。

null 指示符变量中返回的值将是结果字符串在应用程序代码页中的长度。

在除 null 终止符截断以外的所有其他涉及数据截断的情况下，数据库管理器将返回 W。在这种情况下，数据库管理器通过 null 指示符变量将一个值返回给应用程序，该值是结果字符串在选择列表项的代码页（应用程序代码页、数据库代码页或空代码页）中的长度。

在主语言中使用 null 指示符变量之前，请声明 null 指示符变量。在以下适用于 C 和 C++ 程序的示例中，可以将 null 指示符变量 cmind 声明为:

```
EXEC SQL BEGIN DECLARE SECTION;
  char cm[3];
  short cmind;
EXEC SQL END DECLARE SECTION;
```

下表提供受支持的主语言的示例:

表 14. 主语言提供的 null 指示符变量

语言	示例源代码

C 和 C++	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind;`
	`if ( cmind < 0 )`
	`  printf( "Commission is NULL\n" );`

COBOL	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind END-EXEC`
	`IF cmind LESS THAN 0`
	`  DISPLAY 'Commission is NULL'`

FORTRAN	`EXEC SQL FETCH C1 INTO :cm INDICATOR :cmind`
	`IF ( cmind .LT. 0 ) THEN`
	`  WRITE(*,*) 'Commission is NULL'`
	`ENDIF`

REXX	`  CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'`
	`  IF ( cmind < 0 )`
	`  SAY 'Commission is NULL'`

在嵌入式 SQL 应用程序中包括 SQLSTATE 和 SQLCODE 主变量

开始之前

错误信息在 SQLCA 结构的 SQLCODE 和 SQLSTATE 字段中返回，该结构在每个可执行 SQL 语句以及大部分数据库管理器 API 调用之后被更新。如果应用程序符合 FIPS 127-2 标准，那么可以在嵌入式 SQL 应用程序中声明名为 SQLSTATE 和 SQLCODE 的主变量以代替显式地声明 SQLCA 结构。

- 需要指定 PREP 选项 LANGLEVEL SQL92E。

关于此任务

在以下示例中，应用程序将检查 SQLCA 结构的 SQLCODE 字段以确定更新是否成功。

表 15. 在主语言中嵌入 SQL 语句

语言	样本源代码

C 和 C++	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr';`
	`if ( SQLCODE < 0 )`
	`  printf( "Update Error:  SQLCODE =`

COBOL	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr' END_EXEC.`
	`IF SQLCODE LESS THAN 0`
	`  DISPLAY 'UPDATE ERROR:  SQLCODE = ', SQLCODE.`

FORTRAN	`EXEC SQL UPDATE staff SET job = 'Clerk' WHERE job = 'Mgr'`
	`if ( sqlcode .lt. 0 ) THEN`
	`  write(*,*) 'Update error:  sqlcode = ', sqlcode`

在嵌入式 SQL 应用程序中引用主变量

关于此任务

一旦在嵌入式 SQL 应用程序代码中声明主变量，以后就可以在应用程序中引用该变量。在 SQL 语句中使用主变量时，请在其名称前面添加冒号（:）作为前缀。如果在主语言编程语法中使用主变量，请省略冒号。

请使用您所使用的主语言的语法来引用主变量。下表提供了示例。

表 16. 主变量引用（按主语言排列）

语言	示例源代码
C 或 C++	`EXEC SQL FETCH C1 INTO :cm;` `printf( "Commission = %f\n", cm );`
COBOL	`EXEC SQL FETCH C1 INTO :cm END-EXEC` `DISPLAY 'Commission = ' cm`
FORTRAN	`EXEC SQL FETCH C1 INTO :cm` `WRITE(*,*) 'Commission = ', cm`
REXX	`CALL SQLEXEC 'FETCH C1 INTO :cm'` `SAY 'Commission = ' cm`

示例: 在嵌入式 SQL 应用程序中引用 XML 主变量

下列样本应用程序说明如何在 C 和 COBOL 中引用 XML 主变量。

示例: 嵌入式 SQL C 应用程序:

为了便于您阅读，以下代码示例已进行格式化:

```
EXEC SQL BEGIN DECLARE;
  SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
  SQL TYPE IS CLOB( 10K ) clobBuf;
EXEC SQL END DECLARE SECTION;

// 作为 XML AS CLOB
// 写入 xmlBuf 的 XML 值将带有类似于以下内容的 XML 声明前缀:
// <?xml version = "1.0" encoding = "ISO-8859-1" ?>
// 注意: 编码名称取决于应用程序代码页
EXEC SQL SELECT xmlCol INTO :xmlBuf
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlBuf
   WHERE id = '001';

// 作为 XML AS BLOB
// 写入 xmlblob 的 XML 值将带有类似于以下内容的 XML 声明前缀:
// <?xml version = "1.0" encoding = "UTF-8"?>
EXEC SQL SELECT xmlCol INTO :xmlblob
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlblob
   WHERE id = '001';

// 作为 CLOB
// 输出将使用应用程序字符代码页进行编码,
// 但不会包含 XML 声明
EXEC SQL SELECT XMLSERIALIZE (xmlCol AS CLOB(10K)) INTO :clobBuf
```

```
   FROM myTable
   WHERE id = '001';
EXEC SQL UPDATE myTable
   SET xmlCol = XMLPARSE (:clobBuf PRESERVE WHITESPACE)
   WHERE id = '001';
```

示例: 嵌入式 SQL COBOL 应用程序:

为了便于您阅读，以下代码示例已进行格式化:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 xmlBuf USAGE IS SQL TYPE IS XML as CLOB(5K).
  01 clobBuf USAGE IS SQL TYPE IS CLOB(5K).
  01 xmlblob  USAGE IS SQL TYPE IS BLOB(5K).
EXEC SQL END DECLARE SECTION END-EXEC.

* 作为 XML
EXEC SQL SELECT xmlCol INTO :xmlBuf
   FROM myTable
   WHERE id = '001' END-EXEC.
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlBuf
   WHERE id = '001' END-EXEC.

* 作为 BLOB
EXEC SQL SELECT xmlCol INTO :xmlblob
   FROM myTable
   WHERE id = '001' END-EXEC.
EXEC SQL UPDATE myTable
   SET xmlCol = :xmlblob
   WHERE id = '001' END-EXEC.

* 作为 CLOB
EXEC SQL SELECT XMLSERIALIZE(xmlCol AS CLOB(10K)) INTO :clobBuf
   FROM myTable
   WHERE id= '001' END-EXEC.
EXEC SQL UPDATE myTable
   SET xmlCol = XMLPARSE(:clobBuf) PRESERVE WHITESPACE
   WHERE id = '001' END-EXEC.
```

C 和 C++ 嵌入式 SQL 应用程序中的主变量

主变量是 SQL 语句中引用的 C 或 C++ 语言变量。它们使应用程序能够与数据库管理器交换数据。对应用程序进行预编译之后，编译器将像使用任何其他 C 或 C++ 变量那样使用主变量。在命名、声明和使用主变量时，请遵循下列各节描述的规则。

长整型变量注意事项

在以手动方式构造 SQLDA 的应用程序中，当 sqlvar::sqltype==SQL_TYP_INTEGER 时，不能使用长整型变量。而是，必须使用 sqlint32 类型。此问题相当于在主变量声明中使用长整型变量，不同之处在于，对于以手动方式构造的 SQLDA，预编译器将不会发现此错误，因此将发生运行时错误。

任何用于访问 sqlvar::sqldata 信息的长整型和无符号长整型强制类型转换都必须更改为 sqlint32 和 sqluint32。sqloptions 和 sqla_option 结构的 Val 成员被声明为 sqluintptr。因此，将指针成员赋予 sqla_option::val 或 sqloptions::val 成员时，应该使用 sqluintptr 强制类型转换而不是无符号长整型强制类型转换。此更改不会在 64 位 UNIX 和 Linux 操作系统中引起运行时问题，但应该进行此更改以便为长整型类型只有 32 位的 64 位 Windows 应用程序作好准备。

多字节编码注意事项

某些字符编码方案（尤其是东亚地区的字符编码方案）要求使用多个字节来表示一个字符。数据的这种外部表示被称为字符的*多字节字符代码*表示，它包含双字节字符（由两个字节表示的字符）。由于 DB2 中的图形数据由双字节字符组成，因此将相应地选择主变量。

要处理包含双字节字符的字符串，应用程序使用数据的内部表示可能较为方便。这种内部表示被称为双字节字符的*宽字符代码*表示，这就是 C 或 C++ 数据类型 wchar_t 中的常用格式。遵循 ANSI C 和 X/Open 可移植性指南 4（XPG4）的子例程可用于处理宽字符数据以及将数据在宽字符格式与多字节格式之间进行转换。

注意，虽然应用程序可以处理多字节格式或宽字符格式的字符数据，但与数据库管理器进行的交互仅通过 DBCS（多字节）字符代码完成。即，将数据存储到 GRAPHIC 列以及从那些列中检索数据时，采用 DBCS 格式。我们提供了 WCHARTYPE 预编译器选项，以便允许与数据库引擎交换应用程序数据时，在宽字符格式与多字节格式之间进行转换。

C 和 C++ 嵌入式 SQL 应用程序中的主变量名

SQL 预编译器通过已声明的主变量名来标识那些变量。下列规则适用:

- 主变量名的长度不能超过 255 个字符。
- 主变量名不能使用保留供系统使用的前缀 SQL、sql、DB2 和 db2。例如:

```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* 允许 */
  char  sqlvar;     /* 不允许 */
  char  SQL_VAR;    /* 不允许 */
EXEC SQL END DECLARE SECTION;
```

- 预编译器支持的作用域规则与 C 和 C++ 编程语言相同。因此，您可以对两个分别存在于其自己的作用域中的不同变量使用同一名称。在以下示例中，对称为 empno 的变量的两次声明都是被允许的；第二次声明不会导致错误:

```
file: main.sqc
...
void scope1()
{
 EXEC SQL BEGIN DECLARE SECTION;

 short empno;

 EXEC SQL END DECLARE SECTION ;

 ...
}

void scope2()
{
 EXEC SQL BEGIN DECLARE SECTION;

 char[15 + 1] empno;    /* 此声明被允许 */

 EXEC SQL END DECLARE SECTION ;
 ...
}
```

C 和 C++ 嵌入式 SQL 应用程序中主变量的声明节

必须使用 SQL 声明节来标识主变量声明。此节向预编译器指示任何可以在后续 SQL 语句中引用的主变量。例如:

```
EXEC SQL BEGIN DECLARE SECTION;
  char  varsql;     /* 允许 */
EXEC SQL END DECLARE SECTION;
```

C 或 C++ 预编译器只能将一部分有效 C 或 C++ 声明识别为有效主变量声明。这些声明用于定义数字变量或字符变量。可以将主变量分组到单一主结构中。您可以将 C++ 类数据成员声明为主变量。

数字主变量可用作任何数字 SQL 输入或输出值的输入或输出变量。字符主变量可用作任何字符、日期、时间或时间戳记 SQL 输入或输出值的输入或输出变量。应用程序必须确保输出变量的长度足以包含它们所接收的值。

您可以在 SQL 声明节中定义、命名和使用主变量。在以下示例中，首先定义了称为 staff_record 的结构类型。然后，将名为 staff_detail 的变量声明为 staff_record 类型:

```
EXEC SQL BEGIN DECLARE SECTION;

typedef struct {
 short id;
 VARCHAR name[10+1];
 short years;
 double salary;
} staff_record;staff_record staff_detail;

EXEC SQL END DECLARE SECTION ;
...
SELECT id, name, years, salary
 FROM staff
 INTO :staff_detail
 WHERE id = 10;
...
```

可在声明节中使用 define 预处理器关键字，但不包括 # 前缀。# 字符由嵌入式 SQL 预处理器添加。例如:

```
EXEC SQL BEGIN DECLARE SECTION;

define MAX_VALUE 4096 ;

EXEC SQL END DECLARE SECTION ;
```

示例: C 和 C++ 嵌入式 SQL 应用程序的 SQL 声明节模板

以下示例是样本 SQL 声明节，包含为受支持的 SQL 数据类型声明的主变量:

```
   EXEC SQL BEGIN DECLARE SECTION;

.
.
.
        short     age = 26;              /* SQL 类型 500 */
        short     year;                  /* SQL 类型 500 */
        sqlint32  salary;                /* SQL 类型 496 */
        sqlint32  deptno;                /* SQL 类型 496 */
        float     bonus;                 /* SQL 类型 480 */
        double    wage;                  /* SQL 类型 480 */
        char      mi;                    /* SQL 类型 452 */
        char      name[6];               /* SQL 类型 460 */
        struct   {
```

```
          short len;
          char data[24];
         } address;               /* SQL 类型 448 */
struct   {
          short len;
          char data[32695];
         } voice;                 /* SQL 类型 456 */
sql type is clob(1m)
          chapter;                /* SQL 类型 408 */
sql type is clob_locator
          chapter_locator;        /* SQL 类型 964 */
sql type is clob_file
          chapter_file_ref;       /* SQL 类型 920 */
sql type is blob(1m)
          video;                  /* SQL 类型 404 */
sql type is blob_locator
          video_locator;          /* SQL 类型 960 */
sql type is blob_file
          video_file_ref;         /* SQL 类型 916 */
sql type is dbclob(1m)
          tokyo_phone_dir;        /* SQL 类型 412 */
sql type is dbclob_locator
          tokyo_phone_dir_lctr;   /* SQL 类型 968 */
sql type is dbclob_file
          tokyo_phone_dir_flref;  /* SQL 类型 924 */
sql type is varbinary(12)
          myVarBinField;          /* SQL 类型 908 */
sql type is binary(4)
          myBinField;             /* SQL 类型 912 */
struct   {
          short len;
          sqldbchar data[100];
         } vargraphic1;           /* SQL 类型 464 */
                                  /* Precompiled with
                                  WCHARTYPE NOCONVERT option */
struct   {
          short len;
          wchar_t data[100];
         } vargraphic2;           /* SQL 类型 464 */
                                  /* Precompiled with
                                  WCHARTYPE CONVERT option */
struct   {
          short len;
          sqldbchar data[10000];
         } long_vargraphic1;      /* SQL 类型 472 */
                                  /* Precompiled with
                                  WCHARTYPE NOCONVERT option */
struct   {
          short len;
          wchar_t data[10000];
         } long_vargraphic2;      /* SQL 类型 472 */
                                  /* Precompiled with
                                  WCHARTYPE CONVERT option */
sqldbchar graphic1[100];          /* SQL 类型 468 */
                                  /* Precompiled with
                                  WCHARTYPE NOCONVERT option */
wchar_t   graphic2[100];          /* SQL 类型 468 */
                                  /* Precompiled with
                                  WCHARTYPE CONVERT option */
char      date[11];               /* SQL 类型 384 */
char      time[9];                /* SQL 类型 388 */
char      timestamp[27];          /* SQL 类型 392 */
short     wage_ind;               /* Null 指示符 */
```

.

.
.

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

C 和 C++ 嵌入式 SQL 应用程序中的 SQLSTATE 和 SQLCODE 变量

使用值为 SQL92E 的 LANGLEVEL 预编译选项时，可以包括下面这两个声明作为主变量:

```
  EXEC SQL BEGIN DECLARE SECTION;
    char      SQLSTATE[6]
    sqlint32  SQLCODE;


  EXEC SQL END DECLARE SECTION;
```

在预编译步骤中，将采用 SQLCODE 声明。注意，使用此选项时，不能指定 INCLUDE SQLCA 语句。

在由多个源文件构成的应用程序中，可以在第一个源文件中定义 SQLCODE 和 SQLSTATE 变量，如以上示例所示。后续源文件应该按如下方式修改这些定义:

```
   extern sqlint32 SQLCODE;
   extern char     SQLSTATE[6];
```

C 数组主变量和指示符变量

如果将预编译器选项 COMPATIBILITY_MODE 设置为 ORA，那么可将 C 数组主变量和指示符变量数组用于 FETCH INTO 语句。

C 数组主变量

通过使用 C 数组主变量，可以声明一个游标并对数组变量进行批量访存，直到到达该行的结尾。

同一次访存中使用的数组变量需要具有相同的元素数，否则将使用为数组变量声明的最小元素数并显示警告。数组变量的大小各不相同（从 2 到 32K）。

在一次 FETCH 中，可检索的最大记录数是为数组变量声明的最大元素数。如果在第一次访存后有更多行可用，那么可以重复 FETCH 语句以获取下一组行。访存的总行数的累积和存储在 sqlca.sqlerrd[2] 中。

在以下示例中，声明了两个数组主变量 *empno* 和 *lastname*。每个数组主变量最多可容纳 100 个元素。因为仅有一个 FETCH 语句，所以此示例检索 100 个或更少的行。

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname FROM employee;

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname;  /* bulk fetch       */
```

```
 ...                                          /* 100 or less rows */
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

INDICATOR 变量数组

在 FETCH 语句中，可使用指示符变量数组来确定数组变量的任何元素是否为 NULL。如果指示符变量包含一个小于零的值，那么这标识相应的数组值为 NULL。

可使用关键字 INDICATOR 来标识指示符变量，如以下示例所示。

在以下示例中，声明了称为 *bonus_ind* 的指示符变量数组。它可以具有最多 100 个元素，与为数组变量 *bonus* 声明的数量相同。访存数据时，如果 *bonus* 的值为 NULL，那么 *bonus_ind* 中的值将为负数。

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
    short  edlevel[100];
    double bonus[100];
    short  bonus_ind[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname, edlevel, bonus
  FROM employee
  WHERE workdept = 'D21';

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname :edlevel,
         :bonus INDICATOR :bonus_ind
 ...
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

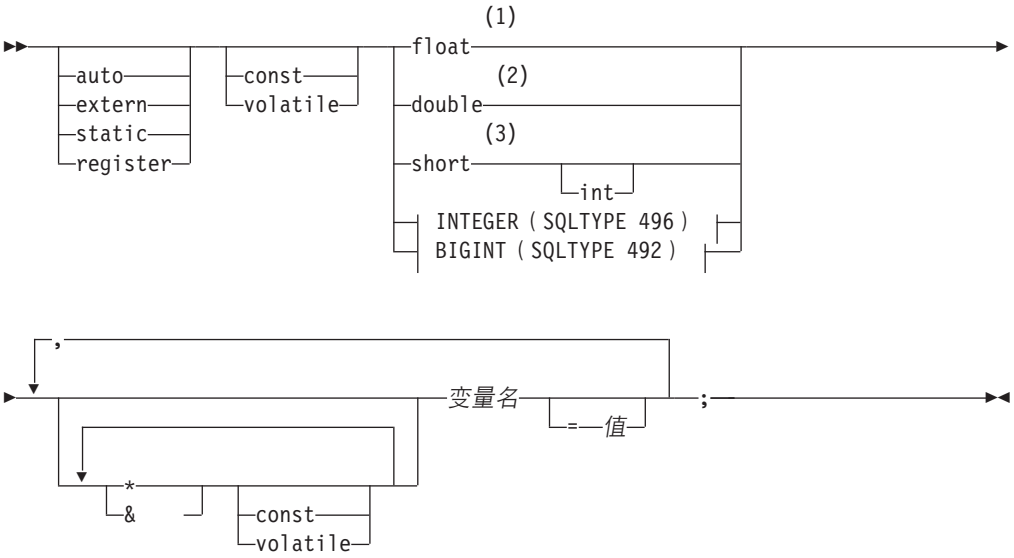
指示符变量可直接跟在其对应主变量后，而不是由 INDICATOR 关键字标识。在以下示例中，使用 `:bonus:bonus_ind` 而不是 `:bonus INDICATOR :bonus_ind`。

```
EXEC SQL FETCH empcr INTO :empno :lastname :edlevel, :bonus:bonus_ind
```

如果指示符数组变量的元素数与相应的主数组变量的元素数不匹配，那么将返回错误。

在 C 和 C++ 嵌入式 SQL 应用程序中声明数字主变量

以下是用于在 C 或 C++ 中声明数字主变量的语法。



INTEGER（SQLTYPE 496）

BIGINT（SQLTYPE 492）

注:

1 REAL（SQLTYPE 480），长度为 4

2 DOUBLE（SQLTYPE 480），长度为 8

3 SMALLINT（SQLTYPE 500）

4 为了最大程度地提高应用程序可移植性，请将 sqlint32 用于 INTEGER 主变量，并将 sqlint64 用于 BIGINT 主变量。缺省情况下，使用长整型主变量将在长整型为 64 位量的平台（例如 64 位 UNIX）上导致预编译器错误 SQL0402。请使用 PREP 选项 LONGERROR NO 来强制 DB2 接受长整型变量作为可接受的主变量类型并将其视为 BIGINT 变量。

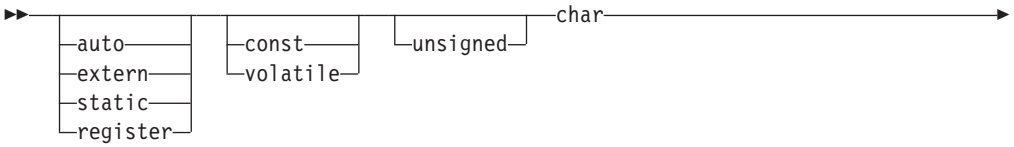
5 为了最大程度地提高应用程序可移植性，请将 sqlint32 和 sqlint64 用于 INTEGER 和 BIGINT 主变量。要使用 BIGINT 数据类型，平台必须支持 64 位整数值。缺省情况下，使用长整型主变量将在长整型为 64 位量的平台（例如 64 位

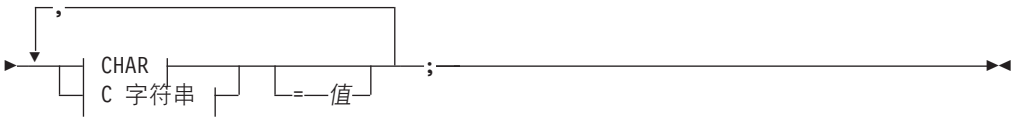
UNIX）上导致预编译器错误 SQL0402。请使用 PREP 选项 LONGERROR NO 来强制 DB2 接受长整型变量作为可接受的主变量类型并将其视为 BIGINT 变量。

在 C 和 C++ 嵌入式 SQL 应用程序中声明定长字符主变量、以 null 结束的字符主变量以及变长字符主变量

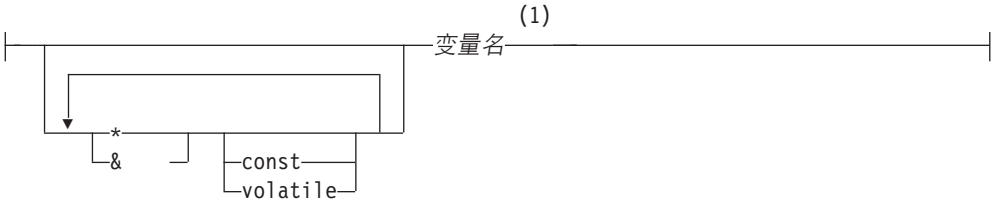
以下是在 C 或 C++ 中声明以 null 结束的定长字符主变量（格式 1）和变长字符主变量（格式 2）的语法。

格式 1: C 或 C++ 嵌入式 SQL 应用程序中以 null 结束的定长字符主变量的语法

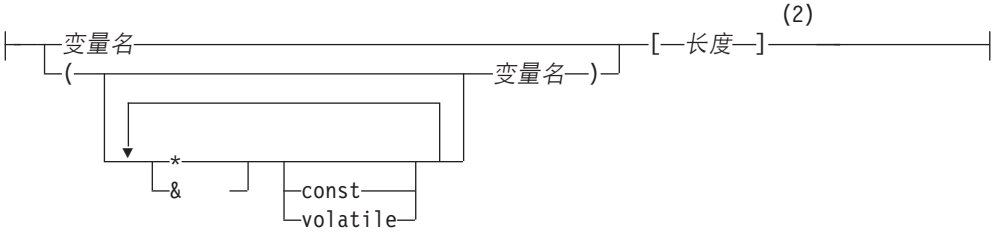




CHAR



C 字符串

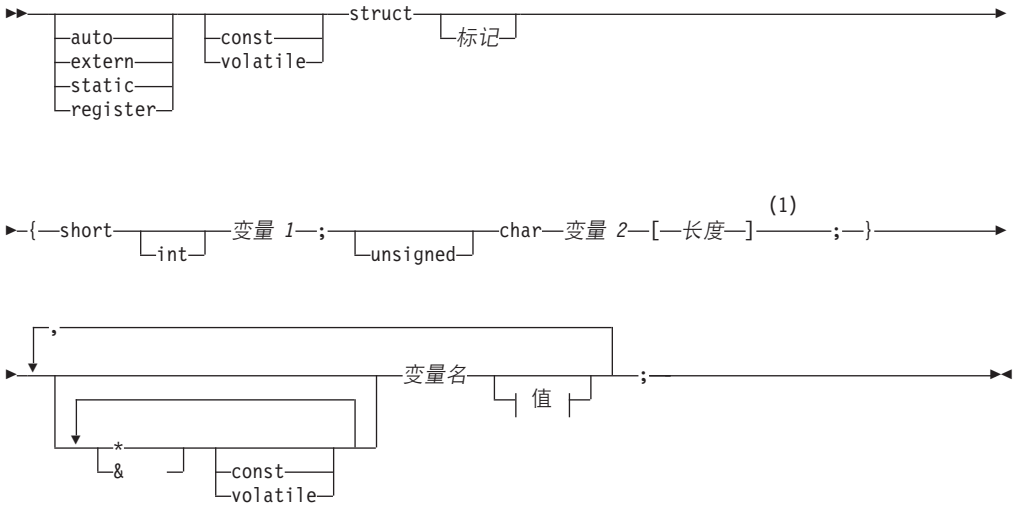


注:

1 CHAR（SQLTYPE 452），长度为 1

2 以 null 结束的 C 字符串（SQLTYPE 460）；长度可以是任何有效的常量表达式

格式 2: C 和 C++ 嵌入式 SQL 应用程序中变长字符主变量的语法



值

注:

1 在格式 2 中，长度可以是任何有效的常量表达式。进行求值后，它的值确定主变量是 VARCHAR（SQLTYPE 448）还是 LONG VARCHAR（SQLTYPE 456）。

变长字符主变量注意事项:

1. 虽然数据库管理器尽可能将字符数据转换为**格式 1** 或**格式 2**，但**格式 1** 与列类型 CHAR 或 VARCHAR 相对应，而**格式 2** 与列类型 VARCHAR 和 LONG VARCHAR 相对应。
2. 如果将**格式 1**与长度指定符 *[n]* 配合使用，那么在进行求值后，长度指定符的值不得大于 32672，并且该变量包含的字符串应该以 null 结束。
3. 如果使用**格式 2**，那么在进行求值后，长度指定符的值不得大于 32700。
4. 在**格式 2** 中，*变量 1* 和*变量 2* 必须是简单变量引用（无运算符），并且不能用作主变量（*变量名*是主变量）。
5. *变量名*可以是简单变量名，也可以包含运算符，例如 *变量名。有关更多信息，请参阅 C 和 C++ 中指针数据类型的描述。
6. 预编译器确定所有主变量的 SQLTYPE 和 SQLLEN。如果主变量出现在带有指示符变量的 SQL 语句中，那么在该语句的持续时间内，SQLTYPE 将被赋值为基本 SQLTYPE 加 1。
7. 预编译器允许某些在 C 或 C++ 语法方面无效的声明。如果您不确定特定的声明语法，请参阅编译器文档。

在 C 和 C++ 嵌入式 SQL 应用程序中声明图形主变量

要在 C 或 C++ 应用程序中处理图形数据，请使用基于 C 或 C++ 数据类型 `wchar_t` 或者 DB2 提供的 `sqldbchar` 数据类型的主变量。可以将这些类型的主变量赋予 GRAPHIC、VARGRAPHIC 或 DBCLOB 表列。例如，可以对表的 GRAPHIC 或 VARGRAPHIC 列中的 DBCS 数据进行更新或选择。

图形主变量共有三种有效格式:

- 单图形格式

  单图形主变量的 SQLTYPE 为 468/469，这与 SQL 数据类型 GRAPHIC(1) 等同。
- 以 null 结束的图形格式

  以 null 结束是指，图形字符串的最后一个字符的所有字节都包含二进制零“\0”。它们的 SQLTYPE 为 400/401。
- VARGRAPHIC 结构化格式

  如果 VARGRAPHIC 结构化主变量的长度介于 1 与 16336 字节之间，那么它们的 SQLTYPE 为 464/465。如果它们的长度介于 2000 与 16350 字节之间，那么它们的 SQLTYPE 为 472/473。

C 和 C++ 嵌入式 SQL 应用程序中用于图形数据的 wchar_t 和 sqldbchar 数据类型

虽然对于特定代码页而言，DB2 图形数据的大小和编码在各个平台之间一致，但 ANSI C 或 C++ wchar_t 数据类型的大小和内部格式取决于您使用的编译器和平台。但是，DB2 将 sqldbchar 数据类型的大小定义为 2 个字节，并且希望使其成为按数据库中的数据存储格式处理 DBCS 和 UCS-2 数据的可移植方法。

您可以使用 wchar_t 或 sqldbchar 来定义所有 DB2 C 图形主变量类型。如果使用 WCHARTYPE CONVERT 预编译选项来构建应用程序，那么必须使用 wchar_t。

注: 在 Windows 操作系统上指定 WCHARTYPE CONVERT 选项时，您必须注意，Windows 操作系统上的 wchar_t 是 Unicode。因此，如果 C 或 C++ 编译器的 wchar_t 不是 Unicode，那么 wcstombs() 函数调用可能会失败并且 SQLCODE 为 -1421（SQLSTATE=22504）。如果发生这种情况，那么您可以指定 WCHARTYPE NOCONVERT 选项并从程序中显式地调用 wcstombs() 和 mbstowcs() 函数。

如果使用 WCHARTYPE NOCONVERT 预编译选项来构建应用程序，那么应该使用 sqldbchar 以便最大程度地提高不同 DB2 客户机和服务器平台之间的可移植性。可以将 wchar_t 与 WCHARTYPE NOCONVERT 配合使用，但只能在 wchar_t 的长度被定义为 2 个字节的平台上这样做。

如果不正确地在主变量声明中使用 wchar_t 或 sqldbchar，那么您将在预编译期间接收到 SQLCODE 15（无 SQLSTATE）。

C 和 C++ 嵌入式 SQL 应用程序中用于图形数据的 WCHARTYPE 预编译器选项

通过使用 WCHARTYPE 预编译器选项，可以指定要在 C 或 C++ 应用程序中使用的图形字符格式。此选项使您能够灵活地选择是让图形数据采用多字节格式还是宽字符格式。WCHARTYPE 选项有两个可能的值:

CONVERT

如果选择 WCHARTYPE CONVERT 选项，那么将在图形主变量与数据库管理器之间转换字符代码。对于图形输入主变量而言，从宽字符格式到多字节 DBCS 字符格式的字符代码转换是在数据被发送到数据库管理器之前使用 ANSI C 函数 wcstombs() 执行的。对于图形输出主变量而言，从多字节 DBCS 字符格式到宽字符格式的字符代码转换是在接收自数据库管理器的数据被存储到主变量之前使用 ANSI C 函数 mbstowcs() 执行的。

使用 WCHARTYPE CONVERT 的优点是，它允许应用程序充分利用 ANSI C 机制来处理宽字符的字符串（L 文字和“wc”字符串函数等等），而不必在与数据库管理器进行通信前显式地将数据转换为多字节格式。缺点是，隐式转换在运行时可能会对应用程序性能产生影响，并可能会增加内存需求。

如果您选择 WCHARTYPE CONVERT，请使用 wchar_t 来声明所有图形主变量，而不要使用 sqldbchar。

如果您期望 WCHARTYPE CONVERT 行为，但应用程序无需进行预编译（例如 CLI 应用程序），请在编译时定义 C 预处理器宏 SQL_WCHART_CONVERT。这将确保 DB2 头文件中的某些定义使用数据类型 wchar_t 而不是 sqldbchar。

NOCONVERT（缺省值）

如果您选择 WCHARTYPE NOCONVERT 选项，或者未指定任何 WCHARTYPE 选项，那么不会在应用程序与数据库管理器之间执行任何隐式的字符代码转换操作。图形主变量中的数据作为未改动的 DBCS 字符在应用程序与数据库管理器之间进行发送/接收。这样做的优点是能够提高性能，但缺点是，应用程序不能在 wchar_t 主变量中使用宽字符数据，或者与数据库管理器进行交互时必须显式地调用 wcstombs() 和 mbstowcs() 函数将数据转换到多字节格式或者从多字节格式转换数据。

如果您选择 WCHARTYPE NOCONVERT，请使用 sqldbchar 类型来声明所有图形主变量，以便最大程度地提高迁移到其他 DB2 客户机/服务器平台的能力。

您必须遵循的其他准则如下所示:

- 由于 wchar_t 或 sqldbchar 支持用于处理 DBCS 数据，因此，使用此支持需要能够支持 DBCS 或 EUC 的硬件和软件。此支持只有在 DB2 Database for Linux, UNIX, and Windows的 DBCS 环境中才可用，或者用于在任何连接到 UCS-2 数据库的应用程序（包括单字节应用程序）中处理 GRAPHIC 数据。
- 不应在图形字符串中使用非 DBCS 字符以及可以转换为非 DBCS 字符的宽字符。*非 DBCS 字符*是指单字节字符以及非双字节字符。系统不会对图形字符串进行验证以确保它们的值只包含双字节字符代码点。图形主变量必须只包含 DBCS 数据，或者，如果 WCHARTYPE CONVERT 生效，那么可以包含转换为 DBCS 数据的宽字符数据。应该将混合双字节和单字节数据存储在字符主变量中。注意，设置 WCHARTYPE 选项不会对混合数据主变量产生影响。
- 在使用了 WCHARTYPE NOCONVERT 预编译选项的应用程序中，不应将 L 文字与图形主变量配合使用，这是因为，L 文字具有宽字符格式。L 文字是前缀为字母 L 并且数据类型为“wchar_t 数组”的 C 宽字符串文字。例如，L"DBCS 字符串" 是 L 文字。
- 在使用了 WCHARTYPE CONVERT 预编译选项的应用程序中，可以使用 L 文字来初始化 wchar_t 主变量，但不能在 SQL 语句中使用该文字。SQL 语句应该使用不受 WCHARTYPE 设置影响的图形字符串常量，而不应使用 L 文字。
- 设置 WCHARTYPE 选项将影响使用 SQLDA 结构以及主变量传递到数据库管理器以及从中传递的图形数据。如果 WCHARTYPE CONVERT 生效，那么通过 SQLDA 从应用程序接收的图形数据将被假定具有宽字符格式，并且将通过隐式的 wcstombs() 调用被转换为 DBCS 格式。同样，应用程序接收的图形输出数据在放入应用程序存储器之前将被转换为宽字符格式。
- 未防护的存储过程必须通过 WCHARTYPE NOCONVERT 选项进行预编译。您可以使用 CONVERT 或 NOCONVERT 选项对普通的受防护存储过程进行预编译，这将影响存储过程中包含的 SQL 语句所处理图形数据的格式。但是，无论在哪一种情况

下，任何通过 SQLDA 传递到存储过程的图形数据都必须具有 DBCS 格式。同样，通过 SQLDA 从存储过程传出的数据必须具有 DBCS 格式。

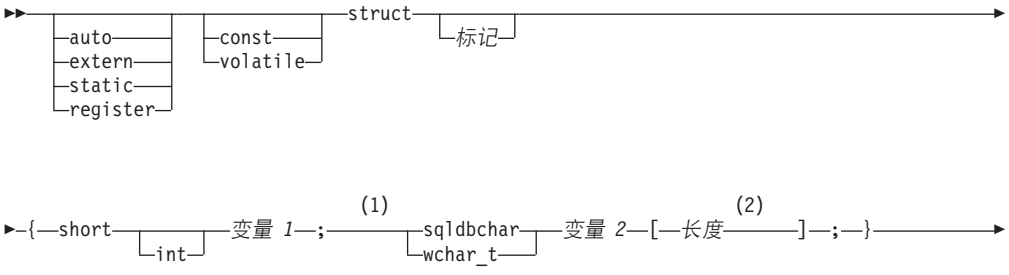
- 如果应用程序通过数据库应用程序远程接口（DARI）接口（sqleproc() API）来调用存储过程，那么输入 SQLDA 中的任何图形数据都必须具有 DBCS 格式或者具有 UCS-2 格式（如果已连接到 UCS-2 数据库），而与调用应用程序的 WCHARTYPE 设置状态无关。同样，输出 SQLDA 中的任何图形数据都将以 DBCS 格式返回或者以 UCS-2 格式返回（如果已连接到 UCS-2 数据库），而与 WCHARTYPE 设置无关。
- 如果应用程序通过 SQL CALL 语句来调用存储过程，那么将根据调用应用程序的 WCHARTYPE 设置对 SQLDA 进行图形数据转换。
- 传递到用户定义的函数 (UDF) 的图形数据将始终具有 DBCS 格式。同样，任何从 UDF 返回的图形数据都将被假定具有 DBCS 格式（对于 DBCS 数据库）和 UCS-2 格式（对于 EUC 和 UCS-2 数据库）。
- 通过使用 DBCLOB 文件引用变量存储在 DBCLOB 文件中的数据将以 DBCS 格式存储或者以 UCS-2 格式存储（对于 UCS-2 数据库）。同样，来自 DBCLOB 文件的输入数据将以 DBCS 格式检索或者以 UCS-2 格式检索（对于 UCS-2 数据库）。

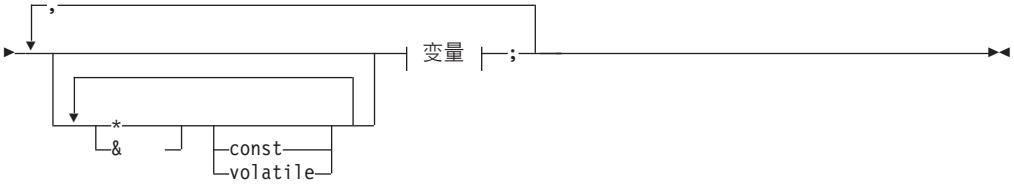
注:

1. 对于用于 Windows 操作系统的 DB2 而言，使用 Microsoft Visual C++ 编译器编译的应用程序支持 WCHARTYPE CONVERT 选项。但是，如果应用程序插入到 DB2 数据库的数据的代码页与数据库代码页不同，请不要将 CONVERT 选项与此编译器配合使用。在这种情况下，DB2 服务器通常会执行代码页转换；但是，Microsoft C 运行时环境不处理某些双字节字符的替换字符。这可能会导致发生运行时转换错误。
2. 如果使用 WCHARTYPE CONVERT 选项来预编译 C 应用程序，那么 DB2 将在应用程序的输入和输出图形数据通过转换函数时验证该数据。如果未使用 CONVERT 选项，那么将不会转换图形数据，因此不执行验证。在混合 CONVERT/NOCONVERT 环境中，如果 NOCONVERT 应用程序插入的无效图形数据被 CONVERT 应用程序提取，那么将出现问题。此数据将导致 CONVERT 应用程序中的 FETCH 执行的转换失败，并且 SQLCODE 为 -1421（SQLSTATE 22504）。

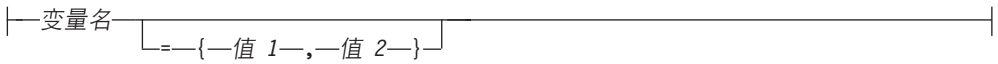
在 C 或 C++ 嵌入式 SQL 应用程序中声明结构化格式的 VARGRAPHIC 类型主变量

以下是用于声明 VARGRAPHIC 结构化格式的图形主变量的语法。





变量:



注:

1 要确定应该使用这两种图形类型中的哪种类型，请参阅 C 和 C++ 中 wchar_t 和 sqldbchar 数据类型的描述。

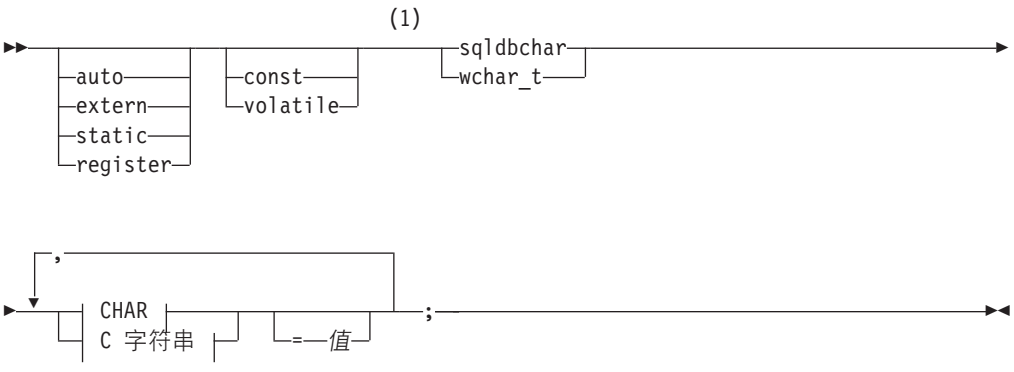
2 *长度*可以是任何有效的常量表达式。进行求值后，它的值确定主变量是 VARGRAPHIC（SQLTYPE 464）还是 LONG VARGRAPHIC（SQLTYPE 472）。长度的值必须大于或等于 1，并且不能大于 LONG VARGRAPHIC 的最大长度 16350。

图形声明（VARGRAPHIC 结构化格式）注意事项:

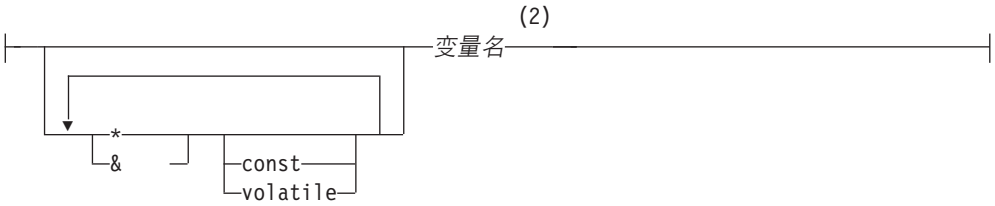
1. *变量 1* 和*变量 2* 必须是简单变量引用（没有运算符），并且不能用作主变量。
2. *值 1* 和*值 2* 是*变量 1* 和*变量 2* 的初始化器。如果使用了 WCHARTYPE CONVERT 预编译器选项，那么*值 1* 必须是整数，*值 2* 必须是宽字符的字符串文字（L 文字）。
3. 结构*标记*可以用来定义其他数据区，但它本身不能用作主变量。

在 C 和 C++ 嵌入式 SQL 应用程序中声明单一图形格式和以 null 结束的图形格式的 GRAPHIC 类型主变量

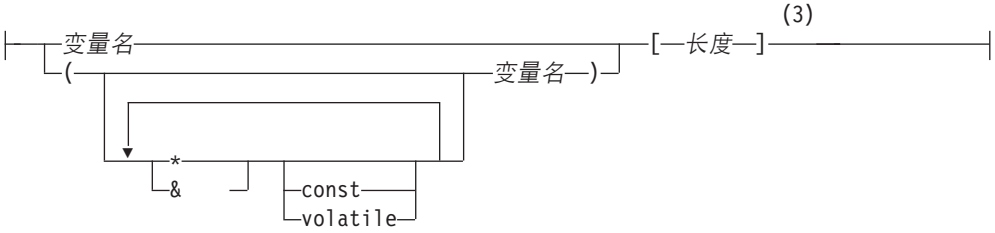
以下是用于声明单一图形格式和以 null 结束的图形格式的图形主变量的语法。



CHAR



C 字符串



注:

1 要确定应该使用这两种图形类型中的哪种类型，请参阅 C 和 C++ 中 wchar_t 和 sqldbchar 数据类型的描述。

2 GRAPHIC（SQLTYPE 468），长度为 1

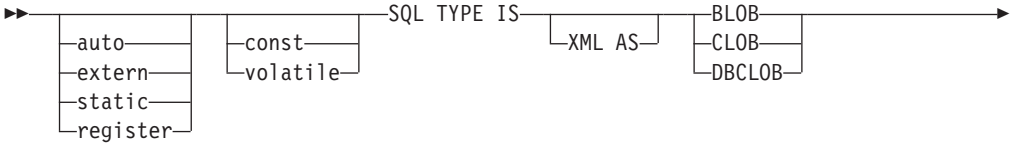
3 以 null 结束的图形字符串（SQLTYPE 400）

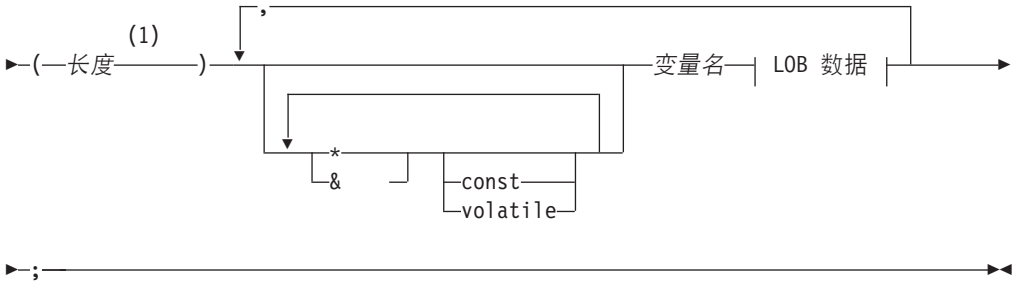
图形主变量注意事项:

1. 单一图形格式声明长度为 1 并且 SQLTYPE 为 468 或 469 的定长图形字符串主变量。
2. *值*是初始化器。如果使用了 WCHARTYPE CONVERT 预编译器选项，那么必须使用宽字符的字符串文字（L 文字）。
3. *长度*可以是任何有效的常量表达式，在进行求值后，它的值必须大于或等于 1，并且不能大于 VARGRAPHIC 的最大长度 16336。
4. 根据标准级别预编译选项设置的不同，对以 null 结束的图形字符串的处理方式也有所不同。

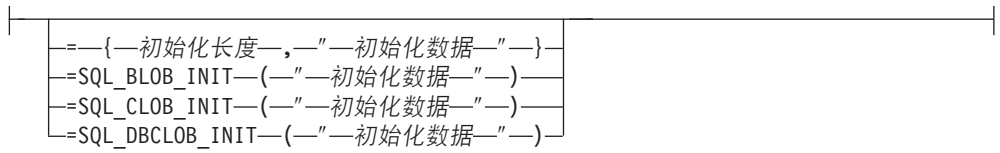
在 C 和 C++ 嵌入式 SQL 应用程序中声明大对象类型主变量

在 C 或 C++ 中声明大对象 (LOB) 主变量的语法是:





LOB 数据



注:

1 *长度*可以是任何有效的常量表达式，在此表达式中，可以使用常量 K、M 或 G。对于 BLOB 和 CLOB，长度值在求值后必须符合以下条件: 1 <= 长度 <= 2147483647。对于 DBCLOB，*长度*值在求值后必须符合以下条件: 1 <= 长度 <= 1073741823。

LOB 主变量注意事项:

1. SQL TYPE IS 子句用于对三种 LOB 类型进行区分，以便对传递到函数的 LOB 类型主变量执行类型检查和函数解析。
2. SQL TYPE IS、BLOB、CLOB、DBCLOB、K、M 和 G 可以是混合大小写。
3. 初始化字符串 *"初始化数据"* 所允许的最大长度是 32702 字节，其中包括字符串定界符（与预编译器中 C 和 C++ 字符串的现有限制相同）。
4. 初始化长度 *初始化长度* 必须是数字常量（例如，它不能包含 K、M 或 G）。
5. 必须指定 LOB 的长度: 即，不允许进行以下声明:

   ```
   SQL TYPE IS BLOB my_blob;
   ```

6. 如果未在声明中初始化 LOB，那么不会在预编译器生成的代码中执行初始化。
7. 如果初始化 DBCLOB，那么用户负责对字符串添加前缀“L”（指示宽字符的字符串）。

   注: 只有在选择了 WCHARTYPE CONVERT 预编译选项的情况下，才应该在预编译的程序中使用宽字符文字（例如 `L"Hello"`）。
8. 预编译器将生成可用于强制转换为主变量的类型的结构标记。

BLOB 示例:

声明:

```
static Sql Type is Blob(2M) my_blob=SQL_BLOB_INIT("mydata");
```

这将生成以下结构:

```
static struct my_blob_t {
      sqluint32        length;
      char             data[2097152];
 } my_blob=SQL_BLOB_INIT("mydata");
```

CLOB 示例:

声明:

```
 volatile sql type is clob(125m) *var1, var2 = {10, "data5data5"};
```

这将生成以下结构:

```
 volatile struct var1_t {
      sqluint32        length;
      char             data[131072000];
} * var1, var2 = {10, "data5data5"};
```

DBCLOB 示例:

声明:

```
SQL TYPE IS DBCLOB(30000) my_dbclob1;
```

在选择 WCHARTYPE NOCONVERT 选项的情况下进行预编译将生成以下结构:

```
struct my_dbclob1_t {
     sqluint32        length;
     sqldbchar        data[30000];
} my_dbclob1;
```

声明:

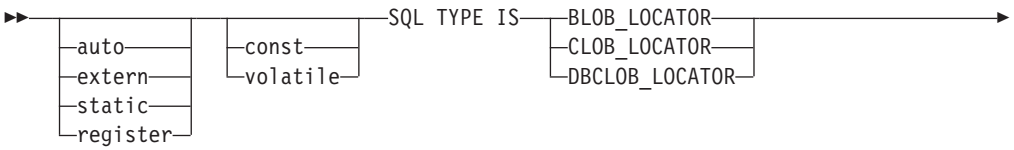
```
SQL TYPE IS DBCLOB(30000) my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

在选择 WCHARTYPE CONVERT 选项的情况下进行预编译将生成以下结构:

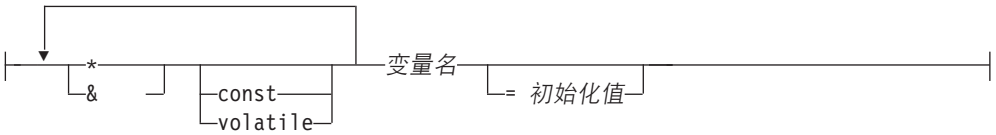
```
struct my_dbclob2_t {
     sqluint32        length;
     wchar_t          data[30000];
} my_dbclob2 = SQL_DBCLOB_INIT(L"mydbdata");
```

在 C 和 C++ 嵌入式 SQL 应用程序中声明大对象定位器类型主变量

在 C 或 C++ 中声明大对象 (LOB) 定位器主变量的语法是:



变量



LOB 定位器主变量注意事项:

1. SQL TYPE IS、BLOB_LOCATOR、CLOB_LOCATOR 和 DBCLOB_LOCATOR 可以混合大小写。
2. *初始化值*允许初始化指针和引用定位器变量。其他类型的初始化没有意义。

CLOB 定位器示例（其他 LOB 定位器类型声明类似）:

声明:

```
SQL TYPE IS CLOB_LOCATOR my_locator;
```

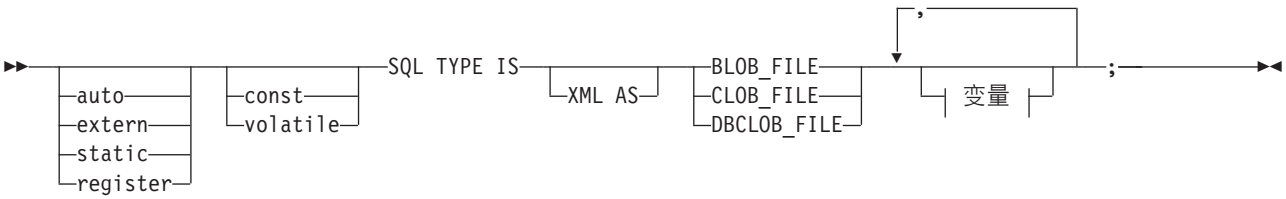
这将生成以下声明:

```
sqluint32 my_locator;
```

在 C 和 C++ 嵌入式 SQL 应用程序中声明文件引用类型主变量

在 C 或 C++ 中声明文件引用主变量的语法是:

C 或 C++ 中文件引用主变量的语法



变量

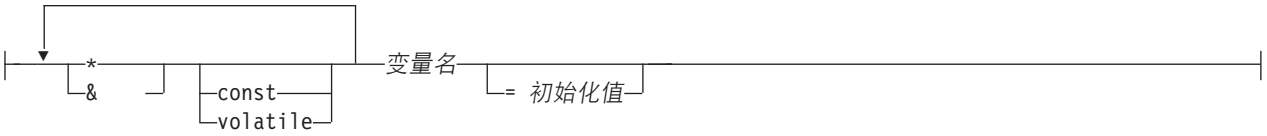


图 1. 语法图

注: SQL TYPE IS、BLOB_FILE、CLOB_FILE 和 DBCLOB_FILE 可以混合大小写。

CLOB 文件引用示例（其他 LOB 文件引用类型声明类似）:

声明:

```
static volatile SQL TYPE IS BLOB_FILE my_file;
```

这将生成以下结构:

```
static volatile struct {
     sqluint32        name_length;
     sqluint32        data_length;
     sqluint32        file_options;
              char    name[255];
} my_file;
```

注: 此结构等同于 sql.h 头中的 sqlfile 结构。参见第 74 页的图 1 以参考语法图。

在 C 和 C++ 嵌入式 SQL 应用程序中将主变量声明为指针

可以将主变量声明为指向特定数据类型的指针，但存在下列限制:

- 如果将主变量声明为指针，那么在同一个源文件中，不能将任何其他主变量声明为具有该名称。以下示例是不允许的:

  ```
  char mystring[20];
  char (*mystring)[20];
  ```

- 在声明指向以 null 结束的字符数组的指针时，请使用括号。在所有其他情况下，不允许使用括号。例如:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char  (*arr)[10];  /* 正确      */
    char  *(arr);      /* 不正确    */
    char  *arr[10];    /* 不正确    */
  EXEC SQL END DECLARE SECTION;
  ```

  第一个声明是指向 10 字节字符数组的指针。这是有效的主变量。第二个声明无效。在指向字符的指针中，不允许使用括号。第三个声明是指针数组。这不是受支持的数据类型。

  接受以下主变量声明:

  ```
  char *ptr;
  ```

  但是，此声明并不表示*长度不确定并且以 null 结束的字符串*。而是，此声明表示*指向定长单字符主变量的指针*。这可能并非您的本意。要定义可以指示不同字符串的指针主变量，请使用本主题上文所示的第一种声明格式。

- 在 SQL 语句中使用指针主变量时，应该在它们前面添加星号作为前缀并且星号数必须与声明时使用星号数相同，如以下示例所示:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
    char (*mychar)[20];    /* 指向 20 个字节的字符数组的指针 */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL SELECT column INTO :*mychar FROM table;   /* 正确 */
  ```

- 只有星号可用作基于主变量名的运算符。
- 主变量名的最大长度不受所指定星号数影响，这是因为，星号不会被视为名称的组成部分。
- 每当在 SQL 语句中使用指针变量时，应该保持优化级别预编译选项（OPTLEVEL）处于缺省设置 0（不优化）。这意味着，数据库管理器不执行任何 SQLDA 优化工作。

在 C++ 嵌入式 SQL 应用程序中将类数据成员声明为主变量

可以将类数据成员声明为主变量（而不是声明类或对象本身）。以下示例演示所使用的方法:

```
class STAFF
{
    private:
         EXEC SQL BEGIN DECLARE SECTION;
           char        staff_name[20];
           short int   staff_id;
           double      staff_salary;
         EXEC SQL END DECLARE SECTION;
         short       staff_in_db;
    .
    .
};
```

只能通过 C++ 编译器在类成员函数中提供的隐式 *this* 指针在 SQL 语句中直接访问数据成员。**不能**在 SQL 语句中显式地限定对象实例（例如 `SELECT name INTO :my_obj.staff_name ...`）。

如果在 SQL 语句中直接引用类数据成员，那么数据库管理器将使用 *this* 指针来解除引用。因此，应该保持优化级别预编译选项（OPTLEVEL）处于缺省设置 0（不优化）。

以下示例说明如何在 SQL 语句中直接使用已声明为主变量的类数据成员。

```
class STAFF
{
.
.
.
    public:

.
.
.

      short int hire( void )
      {
        EXEC SQL INSERT INTO staff ( name,id,salary )
          VALUES ( :staff_name, :staff_id, :staff_salary );
        staff_in_db = (sqlca.sqlcode == 0);
        return sqlca.sqlcode;
      }
};
```

在此示例中，INSERT 语句直接使用了类数据成员 `staff_name`、`staff_id` 和 `staff_salary`。由于它们已被声明为主变量（参见本节中的第一个示例），因此它们被 *this* 指针隐式地限定为当前对象。在 SQL 语句中，还可以通过 *this* 指针来引用不可访问的数据成员。要完成此任务，请使用指针或引用主变量间接地引用这些数据成员。

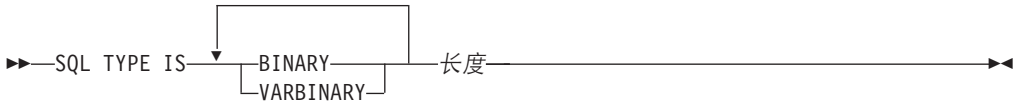
以下示例提供了新方法 *asWellPaidAs*，此方法接收另一个对象 *otherGuy*。此方法通过局部指针或引用主变量来间接地引用该对象的成员，这是因为，不能在 SQL 语句中直接引用那些成员。

```
short int STAFF::asWellPaidAs( STAFF otherGuy )
{
    EXEC SQL BEGIN DECLARE SECTION;
      short &otherID = otherGuy.staff_id
      double otherSalary;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL SELECT SALARY INTO :otherSalary
      FROM STAFF WHERE id = :otherID;
      if( sqlca.sqlcode == 0 )
```

```
            return staff_salary >= otherSalary;
         else
            return 0;
   }
```

在 C 和 C++ 嵌入式 SQL 应用程序中声明二进制类型主变量

在 C 和 C++ 中的二进制和 VARBINARY 定位器主变量的语法是:



示例

声明:

```
SQL TYPE IS BINARY(4) myBinField;
```

这将生成以下 C 代码:

```
 unsigned char myBinField[4];
```

其中，长度为 N（1<= N <=255）

声明:

```
SQL TYPE IS VARBINARY(12) myVarBinField;
```

这将生成以下 C 代码:

```
struct myVarBinField_t { sqluint16 length;
char data[12];
} myVarBinField;
```

其中，长度为 N（1<= N <=32704）

嵌入式 SQL 应用程序对 BINARY 和 VARBINARY 的支持

要在嵌入式应用程序中使用 BINARY 和 VARBINARY 数据类型，请使用声明节中声明的适当数据类型。对于 BINARY 数据，请将该数据复制到用户定义的变量并在 SQL 语句中使用该变量。对于 VARBINARY 数据，请在复制数据前将长度设置为适当的值。

以下示例说明如何在嵌入式应用程序中使用这两种数据类型:

```
EXEC SQL BEGIN DECLARE SECTION;
sql type is binary(50)  binary1 ;
sql type is varbinary(100)  binary2 ;
EXEC SQL END DECLARE SECTION;
char  strng1[50];
char  strng2[50];

memset( binary1, 0x00, sizeof(binary1) );
memset( binary2.data, 0x00, sizeof(binary2.data) );
strcpy( strng1, "AAAAAAZZZZZMMMMMMMMMJJJJJJJJJJJJJJ" );
strcpy( strng2, "BBBBBBBBBBBBBBCCCCCCCCCCCDDDDDDDDEEEEEEEEEEK" );
memcpy( binary1, strng1, strlen(strng1) );
memcpy( binary2.data, strng2, strlen(strng2) );
binary2.length = strlen(binary2.data);

EXEC SQL INSERT INTO test1 VALUES ( :binary1, :binary2 );
```

从数据库中检索数据时，将在相应的结构中正确地设置数据长度。

C 和 C++ 嵌入式 SQL 应用程序中的作用域解析和类成员运算符

在嵌入式 SQL 语句中，*不能*使用 C++ 作用域解析运算符“::”，也不能使用 C 和 C++ 成员运算符“.”或“->”。通过使用在 SQL 语句外部设置的局部指针或引用变量来指向具有所需作用域的变量，然后在 SQL 语句中使用该局部指针或引用变量来引用该变量，可以方便地完成同一任务。以下示例说明所要使用的正确方法:

```
EXEC SQL BEGIN DECLARE SECTION;
  char (& localName)[20] = ::name;
EXEC SQL END DECLARE SECTION;
EXEC SQL
  SELECT name INTO :localName FROM STAFF
  WHERE name = 'Sanders';
```

C 和 C++ 嵌入式 SQL 应用程序中的日语或繁体中文 EUC 和 UCS-2 注意事项

如果应用程序代码页是日语或繁体中文 EUC，或者应用程序连接到 UCS-2 数据库，那么您可以使用 CONVERT 或 NOCONVERT 选项以及 `wchar_t` 或 `sqldbchar` 图形主变量或者输入/输出 SQLDA 来访问数据库服务器上的 GRAPHIC 列。在本节中，*DBCS 格式*是指 EUC 数据的 UCS-2 编码方案。请考虑所列示的情况:

- 使用 CONVERT 选项

  DB2 客户机将图形数据从宽字符格式转换到应用程序代码页，接着转换到 UCS-2，然后再将输入 SQLDA 发送到数据库服务器。任何发送到数据库服务器的图形数据都以 UCS-2 代码页标识作为标记。混合字符数据以应用程序代码页标识作为标记。客户机从数据库检索图形数据时，该数据将以 UCS-2 代码页标识作为标记。DB2 客户机将数据从 UCS-2 转换到客户机应用程序代码页，然后转换为宽字符格式。如果使用输入 SQLDA 来代替主变量，那么您需要确保使用宽字符格式对图形数据进行编码。此数据将被转换为 UCS-2，然后发送到数据库服务器。这些转换将影响性能。

- 使用 NOCONVERT 选项

  DB2 假定图形数据使用 UCS-2 进行编码并以 UCS-2 代码页标识作为标记，因此不执行转换。DB2 假定图形主变量正被用作存储区。选择 NOCONVERT 选项时，从数据库服务器检索的图形数据将在使用 UCS-2 进行编码的情况下传递到应用程序。任何从应用程序代码页到 UCS-2 以及从 UCS-2 到应用程序代码页的转换都由您负责。被标记为 UCS-2 的数据将在不进行任何转换或改变的情况下发送到数据库服务器。

为了最大程度地减少转换，可以使用 NOCONVERT 选项并在应用程序中处理转换，或者不使用 GRAPHIC 列。对于 `wchar_t` 编码是双字节 Unicode 的客户机环境，例如 Windows 2000 或者 AIX V5.1 和更高版本，可以使用 NOCONVERT 选项并直接处理 UCS-2。在此类情况下，应用程序可能会处理大尾数法体系结构与小尾数法体系结构之间的差别。对于 NOCONVERT 选项，DB2 数据库系统将使用始终采用双字节大尾数法的 `sqldbchar`。

不要在转换到 UCS-2 之后（如果指定了 NOCONVERT）或者通过转换到宽字符格式（如果指定了 CONVERT）将 IBM eucJP/IBM eucTW CS0（7 位 ASCII）和 IBM eucJP CS2（片假名）数据指定给图形主变量。原因是，这两个 EUC 代码集中的字符从 UCS-2 转换到 PC DBCS 后都将成为单字节字符。

通常，虽然 eucJP 和 eucTW 将 GRAPHIC 数据存储为 UCS-2，但这些数据库中的 GRAPHIC 数据仍是非 ASCII eucJP 或 eucTW 数据。具体而言，对此类 GRAPHIC 数据填充的任何空格都是 DBCS 空格（在 UCS-2 中，这也称为表意空格，即 U+3000）。但是，对于 UCS-2 数据库，GRAPHIC 数据可以包含任何 UCS-2 字符，空格填充使用 UCS-2 空格 U+0020 完成。在编码应用程序以便从 UCS-2 数据库中检索 UCS-2 数据以及从 eucJP 和 eucTW 数据库中检索 UCS-2 数据时，请记住这种差别。

在 C 和 C++ 嵌入式 SQL 应用程序中使用 FOR BIT DATA 子句以二进制格式存储变量值

标准 C 或 C++ 字符串类型 460 不能用于指定了 FOR BIT DATA 的列。遇到 null 字符时，数据库管理器将截断此数据类型。请使用 VARCHAR（SQL 类型 448）或 CLOB（SQL 类型 408）结构。

在 C 和 C++ 嵌入式 SQL 应用程序中初始化主变量

在 C 和 C++ 声明节中，可以在单一行中声明并初始化多个变量。但是，必须使用“=”符来初始化变量，而不能使用括号进行初始化。以下示例说明声明节中的正确初始化方法和不正确初始化方法:

```
EXEC SQL BEGIN DECLARE SECTION;
  short my_short_2 = 5;       /* 正确   */
  short my_short_1(5);        /* 不正确 */
EXEC SQL END DECLARE SECTION;
```

C 和 C++ 嵌入式 SQL 应用程序中的宏扩展和声明节

C 或 C++ 预编译器无法直接处理声明节中的声明中使用的任何 C 宏。而是，必须先通过外部 C 预处理器对源文件进行预处理。要完成此任务，请通过 PREPROCESSOR 选项对预编译器指定用于调用 C 预处理器的确切命令。

如果您指定了 PREPROCESSOR 选项，那么预编译器先通过将 SQL INCLUDE 语句中引用的所有文件的内容合并到源文件来处理所有 SQL INCLUDE 语句。然后，预编译器使用您指定的命令来调用外部 C 预处理器并将经过修改的源文件用作输入。在余下的预编译过程中，将经过预处理的文件（预编译器始终期望此文件的扩展名为 .i）用作新的源文件。

预编译器生成的任何 #line 宏都不再引用原始源文件，而是引用经过预处理的文件。要使任何编译器错误重新与原始源文件相关，请保留经过预处理的文件中的注释。这可以帮助您找到原始源文件的各个节，包括头文件。C 预处理器通常提供了用于保留注释的选项，您可以在通过 PREPROCESSOR 选项指定的命令中包括该选项。不能让 C 预处理器输出任何 #line 宏本身，这是因为，它们可能会不正确地与预编译器生成的宏混合。

有关使用宏扩展的说明:

1. 通过 PREPROCESSOR 选项指定的命令必须包括所有需要的选项，但不能包括输入文件的名称。例如，对于 AIX 上的 IBM C，可以使用以下选项:

   ```
   xlC -P -DMYMACRO=1
   ```

2. 预编译器期望此命令生成一个经过预处理并且扩展名为 .i 的文件。但是，不能使用重定向功能来生成经过预处理的文件。例如，**不能**使用以下选项来生成经过预处理的文件:

   ```
   xlC -E > x.i
   ```

3. 外部 C 预处理器所遇到的任何错误都将在名称与原始源文件对应但扩展名为 .err 的文件中报告。

例如，可以采用以下方式在源代码中使用宏扩展:

```
#define SIZE 3

EXEC SQL BEGIN DECLARE SECTION;
  char a[SIZE+1];
  char b[(SIZE+1)*3];
  struct
  {
    short length;
    char  data[SIZE*6];
  } m;
  SQL TYPE IS BLOB(SIZE+1) x;
  SQL TYPE IS CLOB((SIZE+2)*3) y;
  SQL TYPE IS DBCLOB(SIZE*2K) z;
EXEC SQL END DECLARE SECTION;
```

使用 PREPROCESSOR 选项后，上述声明将解析为以下示例:

```
EXEC SQL BEGIN DECLARE SECTION;
  char a[4];
  char b[12];
  struct
  {
    short length;
    char  data[18];
  } m;
  SQL TYPE IS BLOB(4) x;
  SQL TYPE IS CLOB(15) y;
  SQL TYPE IS DBCLOB(6144) z;
EXEC SQL END DECLARE SECTION;
```

C 和 C++ 嵌入式 SQL 应用程序的声明节中的主结构支持

借助主结构支持，C 或 C++ 预编译器允许将主变量分组到单一主结构中。此功能使您能够方便地在 SQL 语句中引用该组主变量。例如，可以使用以下主结构来访问 SAMPLE 数据库中 STAFF 表的某些列:

```
struct tag
   {
     short id;
     struct
     {
       short length;
       char  data[10];
     } name;
     struct
     {
       short   years;
       double salary;
     } info;
   } staff_record;
```

主结构的字段可以是任何有效的主变量类型。有效类型包括所有数字、字符和大对象类型。另外，还支持嵌套的主结构，并且最多可嵌套 25 级。在以上显示的示例中，字段 info 是一个子结构，而字段 name 不是，这是因为它表示 VARCHAR 字段。同一原则也适用于 LONG VARCHAR、VARGRAPHIC 和 LONG VARGRAPHIC。另外，还支持指向主结构的指针。

在 SQL 语句中，可以通过两种方法来引用已分组到主结构中的主变量:

- 可以在 SQL 语句中引用主结构名。

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record
     FROM staff
     WHERE id = 10;
```

  预编译器会将对 staff_record 进行的引用转换为列表，该列表包含主结构中声明的所有字段并以逗号分隔。每个字段都由所有各级别的主结构名限定，以避免与其他主变量或字段发生命名冲突。这与以下方法等同。

- 可以在 SQL 语句中引用标准主变量名。

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record.id, :staff_record.name,
          :staff_record.info.years, :staff_record.info.salary
     FROM staff
     WHERE id = 10;
```

  对字段名的引用必须标准，即使没有任何其他同名主变量亦如此。另外，还可以引用限定的子结构。在以上示例中，可以使用 :staff_record.info 来替换 :staff_record.info.years, :staff_record.info.salary。

由于对主结构的引用（第一个示例）等同于该结构的字段的逗号分隔列表，因此，此类引用在某些情况下会引起错误。例如:

```
EXEC SQL DELETE FROM :staff_record;
```

在这里，DELETE 语句需要一个基于字符的主变量。但是，因为提供的是主结构，所以此语句将生成预编译时错误:

```
SQL0087N  主变量"staff_record"是结构，但在其使用位置不允许引用结构。
```

其他可能会导致发生 SQL0087N 错误的主结构用法包括 PREPARE、EXECUTE IMMEDIATE、CALL、指示符变量和 SQLDA 引用。在此类情况下，允许使用正好包含一个字段的主结构，这是因为，它们是对各个字段的引用（第二个示例）。

C 和 C++ 嵌入式 SQL 应用程序中的 null、截断指示符变量和指示符表

对于每个可以接收空值的主变量，请将**指示符变量**声明为 short 数据类型。

指示符表是要与主结构配合使用的指示符变量集合。您必须将其声明为短整数的数组。例如:

```
short ind_tab[10];
```

上一示例声明包含 10 个元素的指示符表。您可以按如下方式在 SQL 语句中使用此表:

```
EXEC SQL SELECT id, name, years, salary
     INTO :staff_record INDICATOR :ind_tab
     FROM staff
     WHERE id = 10;
```

下面列示此表中的每个主结构字段及其相应的指示符变量:

staff_record.id

ind_tab[0]

staff_record.name

ind_tab[1]

staff_record.info.years
ind_tab[2]

staff_record.info.salary
ind_tab[3]

注: 不能在 SQL 语句中逐个引用指示符表元素（例如 ind_tab[1]）。关键字 INDICATOR 是可选的。结构字段数与指示符数不必匹配；任何额外的指示符都不会被使用，未被赋予指示符的额外字段亦如此。

另外，还可以使用标量指示符变量来代替指示符表，以便为主结构的第一个字段提供指示符。这相当于使用只包含一个元素的指示符表。例如:

```
short scalar_ind;

EXEC SQL SELECT id, name, years, salary
          INTO :staff_record INDICATOR :scalar_ind
          FROM staff
          WHERE id = 10;
```

如果随主变量而不是主结构一起指定指示符表，那么将只使用指示符表的第一个元素（例如 ind_tab[0]）:

```
EXEC SQL SELECT id
          INTO :staff_record.id INDICATOR :ind_tab
          FROM staff
          WHERE id = 10;
```

如果在主结构中声明短整数数组:

```
struct tag
{
  short i[2];
} test_record;
```

在 SQL 语句中引用 test_record 时，该数组将扩展到它的元素中，这使 :test_record 相当于 :test_record.i[0], :test_record.i[1]。

C 和 C++ 嵌入式 SQL 应用程序中以 null 结束的字符串

在 C 和 C++ 中，以 null 结束的字符串具有它们自己的 SQLTYPE（用于字符的 460/461 以及用于图形的 468/469）。

在 C 和 C++ 中，根据 LANGLEVEL 预编译器选项值的不同，对以 null 结束的字符串的处理方式也有所不同。如果在 SQL 语句中指定具有其中一个 SQLTYPE 值并且被声明成长度为 *n* 的主变量，而且数据的字节数（对于字符类型）或双字节字符数（对于图形类型）是 *k*，那么:

- 如果 PREP 命令中指定的 LANGLEVEL 选项是 SAA1（缺省值）:

  对于输出:

如果...	**那么...**
k **>** *n*	将 *n* 个字符移至目标主变量，将 SQLWARN1 设置为“W”，并且 SQLCODE 为 0（SQLSTATE 01004）。不会在字符串中放置 null 终止符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 *k*。
k **=** *n*	将 *k* 个字符移至目标主变量，将 SQLWARN1 设置为“N”，并且

SQLCODE 为 0（SQLSTATE 01004）。不会在字符串中放置 null 终止符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 0。

k **<** *n* 将 *k* 个字符移至目标主变量，并在字符 *k + 1* 中放置 null 字符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 0。

对于输入:

当数据库管理器遇到具有其中一个 SQLTYPE 值并且未以 null 终止符结束的输入主变量时，它假定字符 *n+1* 将包含 null 终止符。

- 如果 PREP 命令中指定的 LANGLEVEL 选项是 MIA:

对于输出:

如果... 那么...

k **>=** *n*

将 *n - 1* 个字符移至目标主变量，将 SQLWARN1 设置为“W”，并且 SQLCODE 为 0（SQLSTATE 01501）。第 *n* 个字符被设置为 null 终止符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 *k*。

k + 1 **=** *n*

将 *k* 个字符移至目标主变量，并在字符 *n* 中放置 null 终止符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 0。

k + 1 **<** *n*

将 *k* 个字符移至目标主变量，在右端从字符 *k + 1* 开始追加 *n - k -1* 个空格，然后在字符 *n* 中放置 null 终止符。如果随主变量一起指定指示符变量，那么该指示符变量的值将设置为 0。

对于输入:

当数据库管理器遇到具有其中一个 SQLTYPE 值并且未以 null 字符结束的输入主变量时，将返回 SQLCODE -302（SQLSTATE 22501）。

正如先前定义的那样，在任何其他 SQL 上下文中指定时，SQLTYPE 为 460 并且长度为 *n* 的主变量将被视为长度为 *n* 的 VARCHAR 数据类型，SQLTYPE 为 468 并且长度为 *n* 的主变量将被视为长度为 *n* 的 VARGRAPHIC 数据类型。

COBOL 中的主变量

主变量是 SQL 语句中引用的 COBOL 语言变量。它们使应用程序能够与数据库管理器交换数据。对应用程序进行预编译之后，编译器将像使用任何其他 COBOL 变量那样使用主变量。在命名、声明和使用主变量时，请遵循下列各节描述的规则。

COBOL 中的主变量名

SQL 预编译器通过已声明的主变量名来标识那些变量。下列规则适用:

- 所指定变量名的长度不能超过 255 个字符。
- 主变量名不能使用保留供系统使用的前缀 SQL、sql、DB2 和 db2 开头。

- 在组主变量声明中，允许包括使用声明语法的 FILLER 项，这些项将被预编译器忽略。但是，如果在 SQL 声明节中多次使用 FILLER，那么预编译器将失败。不能在 VARCHAR、LONG VARCHAR、VARGRAPHIC 或 LONG VARGRAPHIC 声明中包括 FILLER 项。
- 可以在主变量名中使用连字符。

  SQL 将两旁为空格的连字符解释为减法运算符。在主变量名中使用连字符时，请不要包括空格。
- 在主变量声明中，允许指定 REDEFINES 子句。
- 在主变量声明节中，允许指定级别 88 声明，但此声明将被忽略。

COBOL 嵌入式 SQL 应用程序中主变量的声明节

必须使用 SQL 声明节来标识主变量声明。此节向预编译器指示任何可以在后续 SQL 语句中引用的主变量。例如:

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 dept             pic s9(4) comp-5.
01 userid           pic x(8).
01 passwd.
    EXEC SQL END DECLARE SECTION END-EXEC.
```

COBOL 预编译器只能识别一部分有效的 COBOL 声明。

示例: COBOL 嵌入式 SQL 应用程序的 SQL 声明节模板

以下代码是样本 SQL 声明节，包含为每种受支持的 SQL 数据类型声明的主变量。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*
  01 age        PIC S9(4) COMP-5.          /* SQL 类型 500 */
  01 divis      PIC S9(9) COMP-5.          /* SQL 类型 496 */
  01 salary     PIC S9(6)V9(3) COMP-3.     /* SQL 类型 484 */
  01 bonus      USAGE IS COMP-1.           /* SQL 类型 480 */
  01 wage       USAGE IS COMP-2.           /* SQL 类型 480 */
  01 nm         PIC X(5).                  /* SQL 类型 452 */
  01 varchar.
     49 leng    PIC S9(4) COMP-5.          /* SQL 类型 448 */
     49 strg    PIC X(14).                 /* SQL 类型 448 */
  01 longvchar.
     49 len     PIC S9(4) COMP-5.          /* SQL 类型 456 */
     49 str     PIC X(6027).               /* SQL 类型 456 */
  01 MY-CLOB USAGE IS SQL TYPE IS CLOB(1M).                /* SQL 类型 408 */
  01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR.    /* SQL 类型 964 */
  01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE.          /* SQL 类型 920 */
  01 MY-BLOB USAGE IS SQL TYPE IS BLOB(1M).                /* SQL 类型 404 */
  01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR.    /* SQL 类型 960 */
  01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.          /* SQL 类型 916 */
  01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(1M).            /* SQL 类型 412 */
  01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR. /* SQL 类型 968 */
  01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE.      /* SQL 类型 924 */
  01 MY-PICTURE PIC G(16000) USAGE IS DISPLAY-1.           /* SQL 类型 464 */
  01 dt         PIC X(10).                 /* SQL 类型 384 */
  01 tm         PIC X(8).                  /* SQL 类型 388 */
  01 tmstmp     PIC X(26).                 /* SQL 类型 392 */
  01 wage-ind   PIC S9(4) COMP-5.          /* SQL 类型 464 */
*
EXEC SQL END DECLARE SECTION END-EXEC.
```

COBOL 嵌入式 SQL 应用程序中的 BINARY/COMP-4 数据类型

DB2 COBOL 预编译器支持在允许使用整型主变量和指示符的位置使用 BINARY、COMP 和 COMP-4 数据类型，条件是目标 COBOL 预编译器将 BINARY、COMP 或 COMP-4 数据类型视为或者能够将其视为与 COMP-5 数据类型等同。在提供的示例中，这样的主变量和指示符与类型 COMP-5 一起显示。将 COMP、COMP-4、BINARY、COMP 和 COMP-5 视为等同并且受 DB2 支持的目标编译器包括:

- IBM COBOL Set for AIX
- Micro Focus COBOL for AIX

COBOL 嵌入式 SQL 应用程序中的 SQLSTATE 和 SQLCODE 变量

使用值为 SQL92E 的 LANGLEVEL 预编译选项时，可以包括下面这两个声明作为主变量:

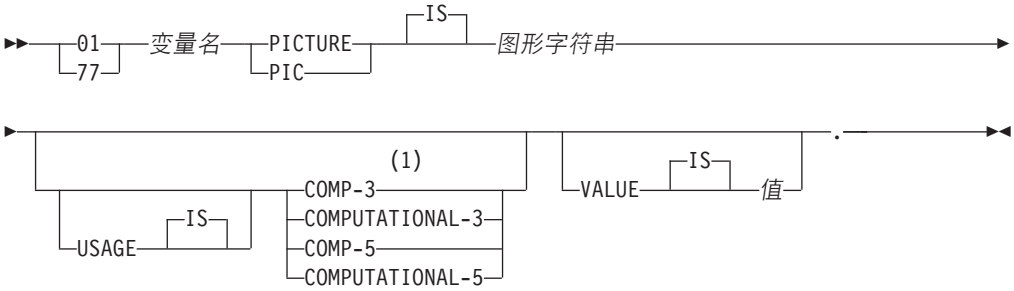
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PIC X(5).
01 SQLCODE  PIC S9(9) USAGE COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

如果未指定这两个声明，那么在预编译步骤中，将采用 SQLCODE 声明。可以使用级别 01（如以上示例所示）或级别 77 来声明 SQLCODE 和 SQLSTATE 变量。注意，使用此选项时，不应指定 INCLUDE SQLCA 语句。

对于由多个源文件构成的应用程序而言，可以在每个源文件中包括 SQLCODE 和 SQLSTATE 声明，如上所示。

在 COBOL 嵌入式 SQL 应用程序中声明数字主变量

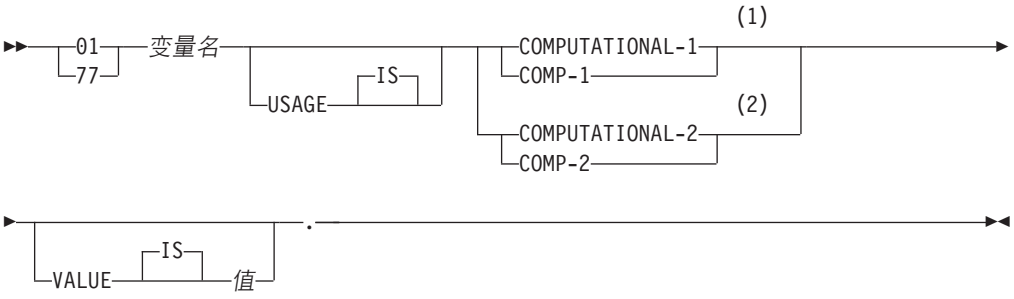
数字主变量的语法是:



注:

1 COMP-3 可以替换为 PACKED-DECIMAL。

浮点



注:

1 REAL (SQLTYPE 480), 长度为 4

2 DOUBLE (SQLTYPE 480), 长度为 8

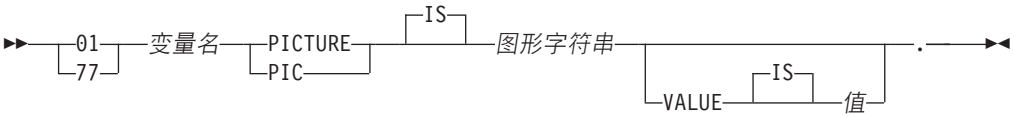
数字主变量注意事项:

1. *图形字符串*的格式必须是下列其中一项:
   - S9(m)V9(n)
   - S9(m)V
   - S9(m)
2. 9 可以展开, 例如指定“S999”代替“S9(3)”
3. *m* 和 *n* 必须是正整数。

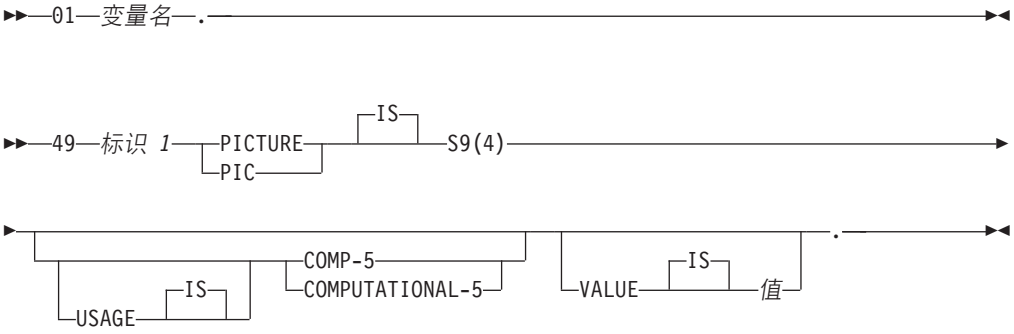
在 COBOL 嵌入式 SQL 应用程序中声明定长字符主变量和变长字符主变量

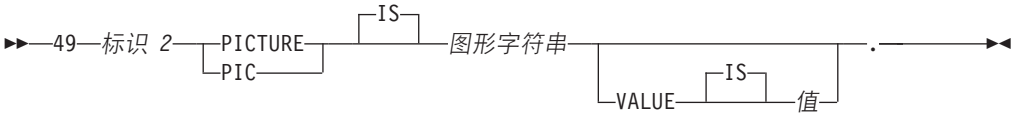
字符主变量的语法是:

固定长度



可变长度





字符主变量注意事项:

1. *图形字符串*的格式必须是 *X(m)*。另外，可以将 X 展开，例如指定“XXX”代替“X(3)”。
2. 对于定长字符串，*m* 是 1 到 254。
3. 对于变长字符串，*m* 是 1 到 32700。
4. 如果 *m* 大于 32672，那么主变量将被视为 LONG VARCHAR 字符串，它的使用可能会受限。
5. 在任何 PICTURE 子句中，请使用 X 和 9 作为图形字符。不允许使用其他字符。
6. 变长字符串由长度项和值项组成。您可以将可接受的 COBOL 名称用于长度项和字符项。但是，在 SQL 语句中，请通过集合体名称来引用变长字符串。
7. 在 CONNECT 语句中（如以下示例所示），在处理 COBOL 字符串主变量 dbname 和 userid 之前，将除去其中的任何尾部空格:

```
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

   但是，由于空格在密码中可能有意义，因此应该将 p-word 主变量声明为 VARCHAR 数据项，以使应用程序能够明确指示 CONNECT 语句的有效密码长度，如下所示:

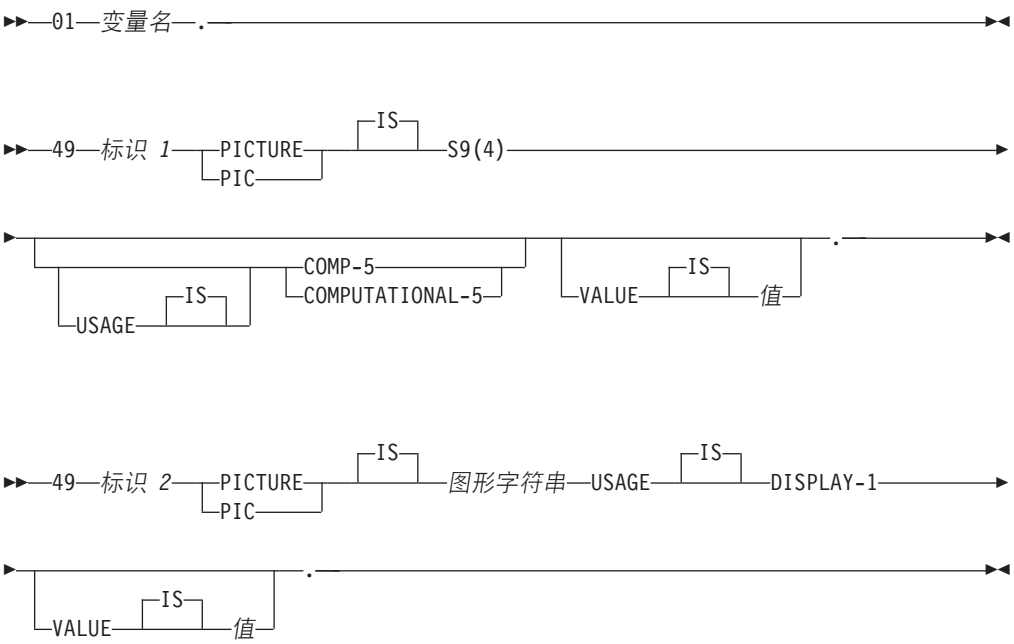
```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 dbname PIC X(8).
01 userid PIC X(8).
01 p-word.
    49 L PIC S9(4) COMP-5.
    49 D PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "sample" TO dbname.
    MOVE "userid" TO userid.
    MOVE "password" TO D OF p-word.
    MOVE 8          TO L of p-word.
EXEC SQL CONNECT TO :dbname USER :userid USING :p-word
END-EXEC.
```

在 COBOL 嵌入式 SQL 应用程序中声明定长图形主变量和变长图形主变量

以下是图形主变量的语法。

固定长度

可变长度

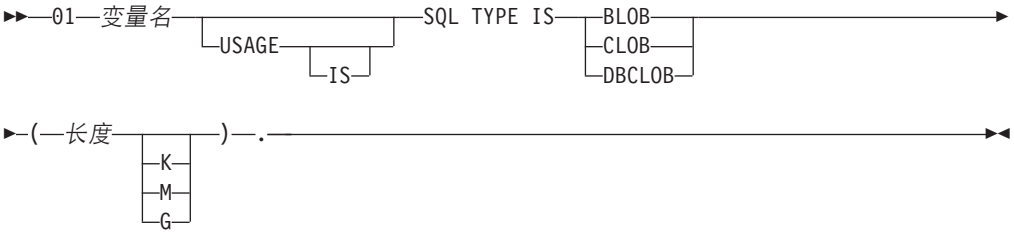


图形主变量注意事项:

1. *图形字符串*的格式必须是 *G(m)*。另外，可以将 G 展开，例如指定“GGG”代替“G(3)”。
2. 对于定长字符串，*m* 是 1 到 127。
3. 对于变长字符串，*m* 是 1 到 16350。
4. 如果 *m* 大于 16336，那么主变量将被视为 LONG VARGRAPHIC 字符串，它的使用可能会受限。

在 COBOL 嵌入式 SQL 应用程序中声明大对象类型主变量

在 COBOL 中声明大对象 (LOB) 主变量的语法是:



LOB 主变量注意事项:

1. 对于 BLOB 和 CLOB，1 <= LOB 长度 <= 2147483647。
2. 对于 DBCLOB，1 <= LOB 长度 <= 1073741823。
3. SQL TYPE IS、BLOB、CLOB、DBCLOB、K、M 和 G 可以是大写、小写或混合大小写。
4. 不允许在 LOB 声明中进行初始化。
5. 在预编译器生成的代码中，LENGTH 和 DATA 以主变量名作为前缀。

BLOB 示例:

声明:

```
01 MY-BLOB USAGE IS SQL TYPE IS BLOB(2M).
```

这将生成以下结构:

```
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-BLOB-DATA PIC X(2097152).
```

CLOB 示例:

声明:

```
01 MY-CLOB USAGE IS SQL TYPE IS CLOB(125M).
```

这将生成以下结构:

```
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-CLOB-DATA PIC X(131072000).
```

DBCLOB 示例:

声明:

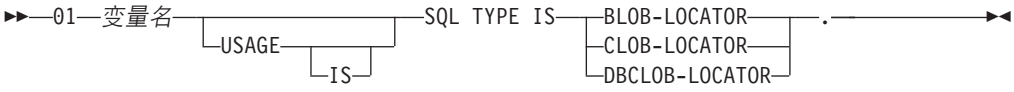
```
01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(30000).
```

这将生成以下结构:

```
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC S9(9) COMP-5.
   49 MY-DBCLOB-DATA PIC G(30000) DISPLAY-1.
```

在 COBOL 嵌入式 SQL 应用程序中声明大对象定位器类型主变量

在 COBOL 中声明大对象 (LOB) 定位器主变量的语法是:



LOB 定位器主变量注意事项:

1. SQL TYPE IS、BLOB-LOCATOR、CLOB-LOCATOR 和 DBCLOB-LOCATOR 可以是大写、小写或混合大小写。
2. 不允许对定位器进行初始化。

BLOB 定位器示例（其他 LOB 定位器类型类似）:

声明:

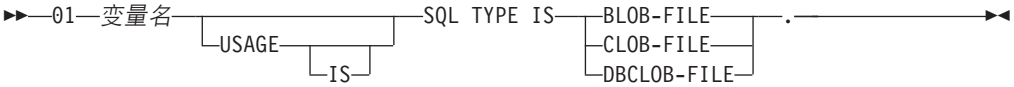
```
01 MY-LOCATOR USAGE SQL TYPE IS BLOB-LOCATOR.
```

这将生成以下声明:

```
01 MY-LOCATOR PIC S9(9) COMP-5.
```

在 COBOL 嵌入式 SQL 应用程序中声明文件引用类型主变量

在 COBOL 中声明文件引用主变量的语法是:



- SQL TYPE IS、BLOB-FILE、CLOB-FILE 和 DBCLOB-FILE 可以是大写、小写或混合大小写。

BLOB 文件引用示例（其他 LOB 类型类似）:

声明:

```
01 MY-FILE USAGE IS SQL TYPE IS BLOB-FILE.
```

这将生成以下声明:

```
01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).
```

在 COBOL 嵌入式 SQL 应用程序中使用 REDEFINES 对数据项进行分组

声明主变量时，可以使用 REDEFINES 子句。如果使用 REDEFINES 子句来声明组数据项的成员，并且在 SQL 语句中将该组数据项作为一个整体进行引用，那么任何包含 REDEFINES 子句的下级项都不会被展开。例如:

```
01 foo1.
 10 a pic s9(4) comp-5.
 10 a1 redefines a pic x(2).
 10 b pic x(10).
```

在 SQL 语句中，按以下方式引用 foo1:

```
... INTO :foo1 ...
```

此语句相当于:

```
... INTO :foo1.a, :foo1.b ...
```

即，在此类情况下，使用 REDEFINES 子句声明的下级项 a1 不会自动展开。如果 a1 明确，那么可以在 SQL 语句中显式地引用带有 REDEFINES 子句的下级，如下所示:

```
... INTO :foo1.a1 ...
```

或

```
... INTO :a1 ...
```

COBOL 嵌入式 SQL 应用程序中的日语或繁体中文 EUC 和 UCS-2 注意事项

在 eucJp 或 eucTW 代码集下运行或者已连接到 UCS-2 数据库的应用程序所发送的任何图形数据都将以 UCS-2 代码页标识作为标记。应用程序在将图形字符串发送到数据库服务器之前，必须先将其转换到 UCS-2。同样，任何应用程序从 UCS-2 数据库检索的图形数据或者在 EUC eucJP 或 eucTW 代码页下运行的应用程序从任何数据库检索的图形数据都将使用 UCS-2 进行编码。除非要向用户提供 UCS-2 数据，否则，这要求应用程序以内部方式从 UCS-2 转换到应用程序代码页。

应用程序负责转换到 UCS-2 或者从 UCS-2 进行转换，这是因为，必须在将数据复制到 SQLDA 之前或者从 SQLDA 复制此数据之后执行此转换。DB2 Database for Linux, UNIX, and Windows未提供任何可供应用程序访问的转换例程。而是，您必须使用操作系统提供的系统调用。对于 UCS-2 数据库而言，您还可以考虑使用 VARCHAR 和 VARGRAPHIC 标量函数。

在 COBOL 嵌入式 SQL 应用程序中使用 FOR BIT DATA 子句以二进制格式存储变量值

您可以使用 FOR BIT DATA 来声明某些数据库列。这些列用于存放二进制信息，它们通常包含字符。CHAR(*n*)、VARCHAR、LONG VARCHAR 和 BLOB 数据类型是可以包含二进制数据的 COBOL 主变量类型。在处理具有 FOR BIT DATA 属性的列时，请使用这些数据类型。

注: 建议您不要使用 LONG VARCHAR 数据类型，在将来的发行版中，可能会除去此数据类型。

COBOL 嵌入式 SQL 应用程序的声明节中的主结构支持

COBOL 预编译器支持在主变量声明节中声明组数据项。除其他方面的作用以外，这还使您能够方便地在 SQL 语句中引用一组基本数据项。例如，可以使用以下组数据项来访问 SAMPLE 数据库中 STAFF 表的某些列:

```
01 staff-record.
    05 staff-id       pic s9(4) comp-5.
    05 staff-name.
        49 l          pic s9(4) comp-5.
        49 d          pic x(9).
    05 staff-info.
        10 staff-dept pic s9(4) comp-5.
        10 staff-job  pic x(5).
```

声明节中的组数据项可将上述任何有效主变量类型作为下级数据项。这包括所有数字和字符类型以及所有大对象类型。可以对组数据项进行嵌套，并且最多可嵌套 10 级。注意，必须声明具有级别为 49 的下级项的 VARCHAR 字符类型，如以上示例所示。如果它们不处于 49 级，那么 VARCHAR 将被视为包含两个下级的组数据项，并且必须遵循有关声明和使用组数据项的规则。在以上示例中，`staff-info` 是组数据项，而 `staff-name` 是 VARCHAR。同一原则也适用于 LONG VARCHAR、VARGRAPHIC 和 LONG VARGRAPHIC。可以在 02 与 49 之间的任何级别声明组数据项。

可以采用四种方式来使用组数据项及其下级:

方法 1.

在 SQL 语句中，可以将整个组作为单一主变量进行引用:

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record
  FROM staff WHERE id = 10 END-EXEC.
```

预编译器会将对 staff-record 进行的引用转换为列表，该列表包含 staff-record 中声明的所有下级项以逗号分隔。每个基本项都由所有各级别的组名限定，以避免与其他项发生命名冲突。这与以下方法等同。

方法 2.

第二种使用组数据项的方法如下所示:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-record.staff-id,
  :staff-record.staff-name,
  :staff-record.staff-info.staff-dept,
  :staff-record.staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

注: 对 staff-id 的引用借助前缀 staff-record. 由其组名限定，而不是像纯 COBOL 中那样由 staff-record 的 staff-id 限定。

假定没有任何其他主变量与 staff-record 的下级同名，以上语句还可以按方法 3 的方式编码，从而消除显式的组限定。

方法 3.

在此方法中，以典型的 COBOL 方式引用下级项，而不将其限定到它们的特定组项:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-dept,
  :staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

就像在纯 COBOL 中一样，只要能够唯一地标识给定的下级项，此方法对于预编译器而言就可接受。例如，如果 staff-job 在多个组中出现，那么预编译器将发出错误以指示不明确的引用:

```
SQL0088N 主变量"staff-job"不明确。
```

方法 4.

要解决不明确的引用，可以对下级项进行部分限定，例如:

```
EXEC SQL SELECT id, name, dept, job
  INTO
  :staff-id,
  :staff-name,
  :staff-info.staff-dept,
  :staff-info.staff-job
  FROM staff WHERE id = 10 END-EXEC.
```

由于对组项的单独引用（方法 1）等同于该项的下级的逗号分隔列表，因此，此类引用在某些情况下会引起错误。例如:

```
EXEC SQL CONNECT TO :staff-record END-EXEC.
```

在这里，CONNECT 语句需要一个基于字符的主变量。如果改为指定 staff-record 组数据项，那么这个主变量将引起以下预编译错误:

```
SQL0087N  主变量"staff-record"是结构，但在其使用位置不允许引用结构。
```

其他可能会导致发生 SQL0087N 的组项用法包括 PREPARE、EXECUTE IMMEDIATE、CALL、指示符变量和 SQLDA 引用。在此类情况下，允许使用只有一个下级的组作为对各个下级的引用，如以上方法 2、3 和 4 所示。

COBOL 嵌入式 SQL 应用程序中的 null 指示符变量以及 null 或截断指示符变量表

应该将 null 指示符变量声明为 PIC S9(4) COMP-5 数据类型。

COBOL 预编译器支持声明 *null 指示符变量表*（称为指示符表），这些表可以方便地与组数据项配合使用。它们的声明方式如下:

```
01 <指示符表名>.
    05 <指示符名称> pic s9(4) comp-5
                       occurs <表大小> times.
```

例如:

```
01 staff-indicator-table.
    05 staff-indicator pic s9(4) comp-5
                      occurs 7 times.
```

可以利用以上第一种组项引用格式来高效地使用这个指示符表:

```
EXEC SQL SELECT id, name, dept, job
  INTO :staff-record :staff-indicator
  FROM staff WHERE id = 10 END-EXEC.
```

在这里，预编译器将检测到 staff-indicator 已被声明为指示符表，因此在处理 SQL 语句时将其展开为各个指示符引用。staff-indicator(1) 与 staff-record 的 staff-id 相关联，staff-indicator(2) 与 staff-record 的 staff-name 相关联，依此类推。

注: 如果指示符表中的指示符条目数比数据项中的下级数多 k 个（例如，如果 staff-indicator 有 10 个条目，那么 k 为 6），那么指示符表末尾的 k 个额外条目将被忽略。同样，如果指示符条目数比下级数少 k 个，那么组项中的最后 k 个下级将没有相关联的指示符。*注意，可以在 SQL 语句中引用指示符表中的各个元素*。

FORTRAN 中的主变量

主变量是 SQL 语句中引用的 FORTRAN 语言变量。它们使应用程序能够与数据库管理器交换数据。对应用程序进行预编译之后，编译器将像使用任何其他 FORTRAN 变量那样使用主变量。在命名、声明和使用主变量时，请遵循下列各节描述的规则。

FORTRAN 嵌入式 SQL 应用程序中的主变量名

SQL 预编译器通过已声明的主变量名来标识那些变量。下列建议适用:

- 所指定变量名的长度不能超过 255 个字符。
- 主变量名不能使用保留供系统使用的前缀 SQL、sql、DB2 和 db2 开头。

FORTRAN 嵌入式 SQL 应用程序中主变量的声明节

必须使用 SQL 声明节来标识主变量声明。此节向预编译器指示任何可以在后续 SQL 语句中引用的主变量。

FORTRAN 预编译器只能将一部分有效 FORTRAN 声明识别为有效主变量声明。这些声明用于定义数字变量或字符变量。数字主变量可用作任何数字 SQL 输入或输出值的输入或输出变量。字符主变量可用作任何字符、日期、时间或时间戳记 SQL 输入或输出值的输入或输出变量。程序员必须确保输出变量的长度足以包含它们将要接收的值。

示例: FORTRAN 嵌入式 SQL 应用程序的 SQL 声明节模板

以下示例是样本 SQL 声明节，包含为每种受支持的数据类型声明的主变量:

```
EXEC SQL BEGIN DECLARE SECTION
  INTEGER*2     AGE  /26/                        /* SQL 类型 500 */
  INTEGER*4     DEPT                            /* SQL 类型 496 */
  REAL*4        BONUS                           /* SQL 类型 480 */
  REAL*8        SALARY                          /* SQL 类型 480 */
  CHARACTER     MI                              /* SQL 类型 452 */
  CHARACTER*112 ADDRESS                         /* SQL 类型 452 */
  SQL TYPE IS VARCHAR (512) DESCRIPTION         /* SQL 类型 448 */
  SQL TYPE IS VARCHAR (32000) COMMENTS          /* SQL 类型 448 */
  SQL TYPE IS CLOB (1M) CHAPTER                 /* SQL 类型 408 */
  SQL TYPE IS CLOB_LOCATOR CHAPLOC              /* SQL 类型 964 */
  SQL TYPE IS CLOB_FILE  CHAPFL                 /* SQL 类型 920 */
  SQL TYPE IS BLOB (1M) VIDEO                   /* SQL 类型 404 */
  SQL TYPE IS BLOB_LOCATOR VIDLOC               /* SQL 类型 960 */
  SQL TYPE IS BLOB_FILE VIDFL                   /* SQL 类型 916 */
  CHARACTER*10  DATE                            /* SQL 类型 384 */
  CHARACTER*8   TIME                            /* SQL 类型 388 */
  CHARACTER*26  TIMESTAMP                       /* SQL 类型 392 */
  INTEGER*2     WAGE_IND                        /* SQL 类型 500 */
EXEC SQL END DECLARE SECTION
```

FORTRAN 嵌入式 SQL 应用程序中的 SQLSTATE 和 SQLCODE 变量

使用值为 SQL92E 的 LANGLEVEL 预编译选项时，可以包括下面这两个声明作为主变量:

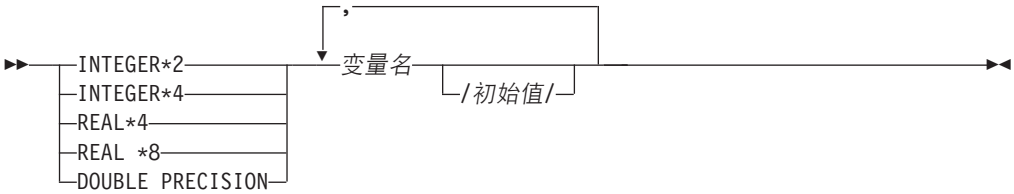
```
EXEC SQL BEGIN DECLARE SECTION;
  CHARACTER*5 SQLSTATE
  INTEGER     SQLCOD
  .
  .
  .
EXEC SQL END DECLARE SECTION
```

在预编译步骤中，将采用 SQLCOD 声明。名为 SQLSTATE 的变量也可以是 SQLSTA。注意，使用此选项时，不应指定 INCLUDE SQLCA 语句。

对于包含多个源文件的应用程序而言，可以在每个源文件中包括 SQLCOD 和 SQLSTATE 的声明，如上所示。

在 FORTRAN 嵌入式 SQL 应用程序中声明数字主变量

以下是 FORTRAN 中的数字主变量的语法。



数字主变量注意事项:

1. REAL*8 与 DOUBLE PRECISION 等同。
2. 对于 REAL*8 常量，请使用 E 代替 D 作为指数指示符。

在 FORTRAN 嵌入式 SQL 应用程序中声明定长字符主变量和变长字符主变量

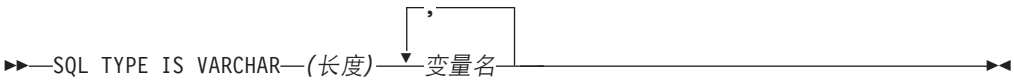
固定长度字符主变量的语法是:

固定长度

FORTRAN 中的字符主变量的语法: 固定长度

以下是可变长度字符主变量的语法。

可变长度



字符主变量注意事项:

1. *n* 的最大值为 254。
2. 长度介于 1 与 32672（包括首尾值）之间时，主变量的类型为 VARCHAR (SQLTYPE 448)。
3. 长度介于 32673 与 32700（包括首尾值）之间时，主变量的类型为 LONG VARCHAR (SQLTYPE 456)。
4. 在声明中，不允许初始化 VARCHAR 和 LONG VARCHAR 主变量。

VARCHAR 示例:

声明:

```
sql type is varchar(1000) my_varchar
```

这将生成以下结构:

```
   character    my_varchar(1000+2)
   integer*2    my_varchar_length
   character    my_varchar_data(1000)
   equivalence( my_varchar(1),
  +             my_varchar_length )
   equivalence( my_varchar(3),
  +             my_varchar_data )
```

应用程序可以同时处理 my_varchar_length 和 my_varchar_data; 例如, 用于设置或检查主变量的内容。在 SQL 语句中, 使用基本名称(在本例中, 这是 my_varchar)来引用整个 VARCHAR。

LONG VARCHAR 示例:

声明:

```
   sql type is varchar(10000) my_lvarchar
```

这将生成以下结构:

```
   character    my_lvarchar(10000+2)
   integer*2    my_lvarchar_length
   character    my_lvarchar_data(10000)
   equivalence( my_lvarchar(1),
  +             my_lvarchar_length )
   equivalence( my_lvarchar(3),
  +             my_lvarchar_data )
```

应用程序可以同时处理 my_lvarchar_length 和 my_lvarchar_data; 例如, 用于设置或检查主变量的内容。在 SQL 语句中, 使用基本名称(在本例中, 这是 my_lvarchar)来引用整个 LONG VARCHAR。

注: 在 CONNECT 语句中(例如在以下示例中), 在处理 FORTRAN 字符串主变量 dbname 和 userid 之前, 将除去其中的任何尾部空格:

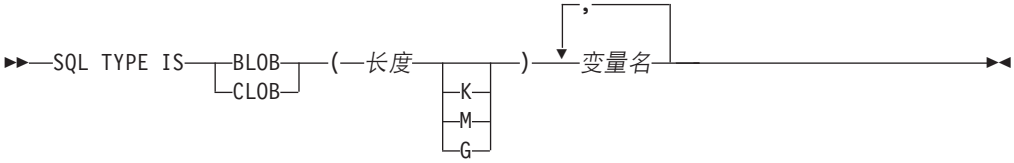
```
   EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

但是, 由于空格在密码中有意义, 因此, 您应该将密码的主变量声明为 VARCHAR, 并且应该设置长度字段以反映实际密码长度:

```
   EXEC SQL BEGIN DECLARE SECTION
     character*8 dbname, userid
     sql type is varchar(18) passwd
   EXEC SQL END DECLARE SECTION
   character*18 passwd_string
   equivalence(passwd_data,passwd_string)
   dbname = '样本'
   userid = '用户标识'
   passwd_length= 8
   passwd_string = '密码'
   EXEC SQL CONNECT TO :dbname USER :userid USING :passwd
```

在 FORTRAN 嵌入式 SQL 应用程序中声明大对象类型主变量

在 FORTRAN 中声明大对象 (LOB) 主变量的语法是:



LOB 主变量注意事项:

1. 在 FORTRAN 中，不支持 GRAPHIC 类型。
2. SQL TYPE IS、BLOB、CLOB、K、M 和 G 可以是大写、小写或混合大小写。
3. 对于 BLOB 和 CLOB，1 <= LOB 长度 <= 2147483647。
4. 不允许在 LOB 声明中初始化 LOB。
5. 在预编译器生成的代码中，“length”和“data”以主变量名作为前缀。

BLOB 示例:

声明:

```
sql type is blob(2m) my_blob
```

这将生成以下结构:

```
 character    my_blob(2097152+4)
 integer*4    my_blob_length
 character    my_blob_data(2097152)
 equivalence( my_blob(1),
+             my_blob_length )
 equivalence( my_blob(5),
+             my_blob_data )
```

CLOB 示例:

声明:

```
sql type is clob(125m) my_clob
```

这将生成以下结构:

```
 character    my_clob(131072000+4)
 integer*4    my_clob_length
 character    my_clob_data(131072000)
 equivalence( my_clob(1),
+             my_clob_length )
 equivalence( my_clob(5),
+             my_clob_data )
```

在 FORTRAN 嵌入式 SQL 应用程序中声明大对象定位器类型主变量

在 FORTRAN 中声明大对象 (LOB) 定位器主变量的语法是:

LOB 定位器主变量注意事项:

1. 在 FORTRAN 中，不支持 GRAPHIC 类型。

2. SQL TYPE IS、BLOB_LOCATOR 和 CLOB_LOCATOR 可以是大写、小写或混合大小写。
3. 不允许对定位器进行初始化。

CLOB 定位器示例（BLOB 定位器类似）：

声明:

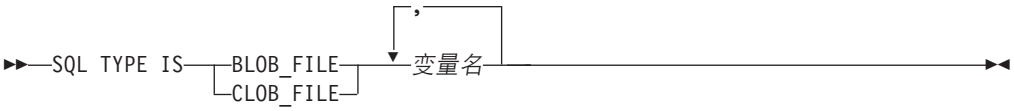
```
SQL TYPE IS CLOB_LOCATOR my_locator
```

这将生成以下声明:

```
integer*4 my_locator
```

在 FORTRAN 嵌入式 SQL 应用程序中声明文件引用类型主变量

在 FORTRAN 中声明文件引用主变量的语法是:



文件引用主变量注意事项:

1. 在 FORTRAN 中，不支持 GRAPHIC 类型。
2. SQL TYPE IS、BLOB_FILE 和 CLOB_FILE 可以是大写、小写或混合大小写。

BLOB 文件引用变量的示例（CLOB 文件引用变量类似）：

```
SQL TYPE IS BLOB_FILE my_file
```

这将生成以下声明:

```
 character     my_file(267)
 integer*4     my_file_name_length
 integer*4     my_file_data_length
 integer*4     my_file_file_options
 character*255 my_file_name
 equivalence(  my_file(1),
+              my_file_name_length )
 equivalence(  my_file(5),
+              my_file_data_length )
 equivalence(  my_file(9),
+              my_file_file_options )
 equivalence(  my_file(13),
+              my_file_name )
```

FORTRAN 嵌入式 SQL 应用程序中图形（多字节）字符集的注意事项

在 FORTRAN 中，不支持任何图形（多字节）主变量数据类型。而是，只通过 character 数据类型支持混合字符主变量。但是，您可以创建包含图形数据的用户 SQLDA。

FORTRAN 嵌入式 SQL 应用程序中的日语或繁体中文 EUC 和 UCS-2 注意事项

在 eucJp 或 eucTW 代码集下运行或者已连接到 UCS-2 数据库的应用程序所发送的任何图形数据都将以 UCS-2 代码页标识作为标记。应用程序在将图形字符串发送到数据库服务器之前，必须先将其转换到 UCS-2。同样，任何应用程序从 UCS-2 数据库检索

的图形数据或者在 EUC eucJP 或 eucTW 代码页下运行的应用程序从任何数据库检索的图形数据都将使用 UCS-2 进行编码。除非要向用户提供 UCS-2 数据，否则，这要求应用程序以内部方式从 UCS-2 转换到应用程序代码页。

应用程序负责转换到 UCS-2 或者从 UCS-2 进行转换，这是因为，必须在将数据复制到 SQLDA 之前或者从 SQLDA 复制此数据之后执行此转换。DB2 数据库系统未提供任何可供应用程序访问的转换例程。而是，您必须使用操作系统提供的系统调用。对于 UCS-2 数据库而言，您还可以考虑使用 VARCHAR 和 VARGRAPHIC 标量函数。

FORTRAN 嵌入式 SQL 应用程序中的 null 或截断指示符变量

必须将指示符变量声明为 INTEGER*2 数据类型。

REXX 中的主变量

主变量是 SQL 语句中引用的 REXX 语言变量。它们使应用程序能够与数据库管理器交换数据。对应用程序进行预编译之后，编译器将像使用任何其他 REXX 变量那样使用主变量。在命名、声明和使用主变量时，请遵循下列各节描述的规则。

REXX 嵌入式 SQL 应用程序中的主变量名

任何正确命名的 REXX 变量都可以用作主变量。变量名的长度可达 64 个字符。名称不能以句点结尾。主变量名可以包含数字、字母字符以及字符 @、_、!、.、? 和 $。

REXX 嵌入式 SQL 应用程序中的主变量引用

REXX 解释器将检查过程中每个不带引号的字符串。如果字符串代表当前 REXX 变量池中的某个变量，那么 REXX 将把该字符串替换为当前值。以下是在 REXX 中引用主变量方式的示例:

```
CALL SQLEXEC 'FETCH C1 INTO :cm'
SAY 'Commission = ' cm
```

要确保不将字符串转换为数字数据类型，请将该字符串括在单引号中，如以下示例所示:

```
VAR = '100'
```

REXX 将变量 *VAR* 设置为 3 字节字符串 100。如果要包括单引号作为字符串的组成部分，请遵循以下示例:

```
VAR = "'100'"
```

将数字数据插入到 CHARACTER 字段时，REXX 解释器将数字数据视为整数数据，因此必须显式并置数字字符串并将其括在单引号中。

预定义的 REXX 变量

SQLEXEC、SQLDBS 和 SQLDB2 将设置预定义的 REXX 变量作为某些操作的结果。这些变量是:

RESULT

每个操作都将设置此返回码。可能的值包括:

n 其中，*n* 是正数值，用于指示格式化消息中的字节数。单独的 GET ERROR MESSAGE API 将返回此值。

0 已执行 API。REXX 变量 SQLCA 包含 API 的完成状态。如果 SQLCA.SQLCODE 不为零，那么 SQLMSG 包含与该值相关联的文本消息。

-1 没有足够的内存，无法完成 API。未返回所请求的消息。

-2 SQLCA.SQLCODE 设置为 0。未返回任何消息。

-3 SQLCA.SQLCODE 包含无效的 SQLCODE。未返回任何消息。

-6 未能构建 REXX 变量 SQLCA。这表示没有足够的内存，或者 REXX 变量池由于某种原因而不可用。

-7 未能构建 REXX 变量 SQLMSG。这表示没有足够的内存，或者 REXX 变量池由于某种原因而不可用。

-8 未能从 REXX 变量池中提取 REXX 变量 SQLCA.SQLCODE。

-9 REXX 变量 SQLCA.SQLCODE 在提取期间被截断。此变量的最大长度是 5 个字节。

-10 未能将 REXX 变量 SQLCA.SQLCODE 由 ASCII 转换为有效的长整数。

-11 未能从 REXX 变量池中提取 REXX 变量 SQLCA.SQLERRML。

-12 REXX 变量 SQLCA.SQLERRML 在提取期间被截断。此变量的最大长度是 2 个字节。

-13 未能将 REXX 变量 SQLCA.SQLERRML 由 ASCII 转换为有效的短整数。

-14 未能从 REXX 变量池中提取 REXX 变量 SQLCA.SQLERRMC。

-15 REXX 变量 SQLCA.SQLERRMC 在提取期间被截断。此变量的最大长度是 70 个字节。

-16 未能设置对错误文本指定的 REXX 变量。

-17 未能从 REXX 变量池中提取 REXX 变量 SQLCA.SQLSTATE。

-18 REXX 变量 SQLCA.SQLSTATE 在提取期间被截断。此变量的最大长度是 2 个字节。

注: 只有 GET ERROR MESSAGE API 才会返回值 -8 到 -18。

SQLMSG

如果 SQLCA.SQLCODE 不为 0，那么此变量包含与错误代码相关联的文本消息。

SQLISL

隔离级别。可能的值包括:

RR 可重复读。

RS 读稳定性。

CS 游标稳定性。这是缺省值。

UR 未落实的读。

NC 不落实。（只有某些主机或 System i® 服务器才支持 NC。）

SQLCA

在处理 SQL 语句并调用 DB2 API 之后，将更新 SQLCA 结构。

SQLRODA

使用 CALL 语句调用的存储过程的输入/输出 SQLDA 结构。这也是使用数据库应用程序远程接口（DAPI）API 调用的存储过程的输出 SQLDA 结构。

SQLRIDA

使用数据库应用程序远程接口（DAPI）API 调用的存储过程的输入 SQLDA 结构。

SQLRDAT

使用数据库应用程序远程接口（DAPI）API 调用的服务器过程的 SQLCHAR 结构。

进行 REXX 嵌入式 SQL 应用程序编程时的注意事项

关于此任务

REXX 是解释型语言。因此，不使用预编译器、编译器或链接程序。而是，使用三个 DB2 API 来创建 REXX 语言的 DB2 应用程序。请使用这些 API 来访问 DB2 的不同元素。

SQLEXEC

支持 SQL 语言。

SQLDBS

支持类似于命令的 DB2 API 版本。

SQLDB2

支持特定于 REXX 的命令行处理器接口。请参阅 REXX 的 API 语法描述，以了解有关此接口的用法的详细信息和限制。

在应用程序中使用任何 DB2 API 或发出 SQL 语句之前，必须注册 SQLDBS、SQLDB2 和 SQLEXEC 例程。这将 REXX/SQL 入口点通知 REXX 解释器。在基于 Windows 的平台与 AIX 平台之间，注册方法略有不同。

请通过下列示例了解用于注册每个例程的正确语法:

Windows 操作系统上的样本注册

```
/* ------------ 向 REXX 注册 SQLDBS -------------------------*/
If Rxfuncquery('SQLDBS')  <> 0 then
    rcy = Rxfuncadd('SQLDBS','DB2AR','SQLDBS')
If rcy \= 0 then
    do
      say 'SQLDBS was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ------------ 向 REXX 注册 SQLDB2 -------------------------*/
If Rxfuncquery('SQLDB2')  <> 0 then
    rcy = Rxfuncadd('SQLDB2','DB2AR','SQLDB2')
If rcy \= 0 then
    do
      say 'SQLDB2 was not successfully added to the REXX environment'
      signal rxx_exit
    end

/* ----------------- 向 REXX 注册 SQLEXEC --------------------*/
If Rxfuncquery('SQLEXEC') <> 0 then
    rcy = Rxfuncadd('SQLEXEC','DB2AR','SQLEXEC')
If rcy \= 0 then
```

```
  do
    say 'SQLEXEC was not successfully added to the REXX environment'
    signal rxx_exit
  end
```

AIX 上的样本注册

```
 /* ------------ 向 REXX 注册 SQLDBS、SQLDB2 和 SQLEXEC --------*/
rcy = SysAddFuncPkg("db2rexx")
If rcy \= 0 then
  do
    say 'db2rexx was not successfully added to the REXX environment'
    signal rxx_exit
  end
```

在基于 Windows 的平台上，只需对所有会话执行一次 RxFuncAdd 命令。

在 AIX 上，应该在每个 REXX/SQL 应用程序中执行 SysAddFuncPkg。

基于 Windows 的平台和 AIX 的 REXX 文档中提供了有关 Rxfuncadd 和 SysAddFuncPkg API 的详细信息。

传递给 SQLEXEC、SQLDBS 和 SQLDB2 例程的语句或命令中的标记可以与 REXX 变量相对应。在这种情况下，REXX 解释器将在调用 SQLEXEC、SQLDBS 或 SQLDB2 之前替换变量的值。

要避免出现这种情况，请用引号（’ ’ 或 ″ ″）将语句字符串括起来。如果未使用引号，那么任何有冲突的变量名称都由 REXX 解释器解析，而不是被传递到 SQLEXEC、SQLDBS 或 SQLDB2 例程。

在 REXX 嵌入式 SQL 应用程序中声明大对象类型主变量

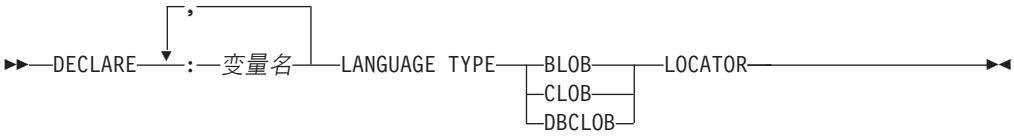
将 LOB 列提取到 REXX 主变量时，它将被存储为简单字符串（即，不计数的字符串）。其处理方式与所有基于字符的 SQL 类型（例如 CHAR、VARCHAR、GRAPHIC 和 LONG 等等）相同。对于输入，如果主变量的内容大小大于 32K，或者它符合下表中列出的其他设定条件，那么它将被赋予适当的 LOB 类型。

在 REXX SQL 中，LOB 类型根据主变量的字符串内容确定，如下所示:

主变量字符串内容	结果 LOB 类型
:hv1=’ordinary quoted string longer than 32K ...’	CLOB
:hv2=″’string with embedded delimiting quotation marks ″, ″longer than 32K...’″	CLOB
:hv3=″G’DBCS string with embedded delimiting single ″, ″quotation marks, beginning with G, longer than 32K...’″	DBCLOB
:hv4=″BIN’string with embedded delimiting single ″, ″quotation marks, beginning with BIN, any length...’″	BLOB

在 REXX 嵌入式 SQL 应用程序中声明大对象定位器类型主变量

在 REXX 中声明 LOB 定位器主变量的语法是:



必须在应用程序中声明 LOB 定位器主变量。REXX/SQL 遇到这些声明后，它将在程序的余下部分中将您声明的主变量视为定位器。定位器值以内部格式存储在 REXX 变量中。

示例:

```
CALL SQLEXEC 'DECLARE :hv1, :hv2 LANGUAGE TYPE CLOB LOCATOR'
```

在 REXX/SQL 中，可以使用格式如下的 FREE LOCATOR 语句来释放从引擎返回的 LOB 定位器所表示的数据:

FREE LOCATOR 语句的语法

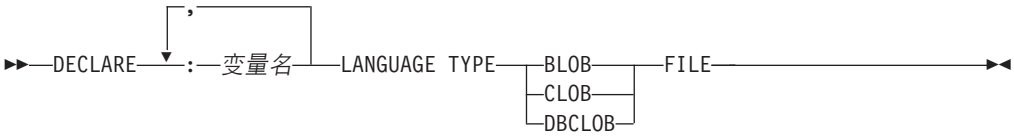
示例:

```
CALL SQLEXEC 'FREE LOCATOR :hv1, :hv2'
```

在 REXX 嵌入式 SQL 应用程序中声明文件引用类型主变量

必须在应用程序中声明 LOB 文件引用主变量。REXX/SQL 遇到这些声明后，它将在程序的余下部分中将您声明的主变量视为 LOB 文件引用。

在 REXX 中声明 LOB 文件引用主变量的语法是:



示例:

```
CALL SQLEXEC 'DECLARE :hv3, :hv4 LANGUAGE TYPE CLOB FILE'
```

REXX 中的文件引用变量包含三个字段。对于以前的示例，它们是:

hv3.FILE_OPTIONS。
由应用程序设置，用于指示如何使用该文件。

hv3.DATA_LENGTH。
由 DB2 设置，用于指示该文件的大小。

hv3.NAME。
由应用程序设置，用于指示该 LOB 文件的名称。

对于 FILE_OPTIONS，应用程序将设置下列关键字:

关键字（整数值）
含义

READ（2）
此文件用于输入。这是可以被打开、读取和关闭的常规文件。此文件中的数据长度（以字节计）在此文件被打开后由应用程序请求器代码计算。

CREATE（8）
在输出时，创建新文件。如果此文件已存在，那么将发生错误。此文件的长度（以字节计）将在文件引用变量结构的 DATA_LENGTH 字段中返回。

OVERWRITE（16）
在输出时，如果存在现有的文件，那么将其覆盖，否则创建新文件。此文件的长度（以字节计）将在文件引用变量结构的 DATA_LENGTH 字段中返回。

APPEND（32）
如果此文件已存在，那么在此文件末尾追加输出，否则创建新文件。对此文件添加的数据的长度（以字节计，但不是文件总长度）将在文件引用变量结构的 DATA_LENGTH 字段中返回。

注: 文件引用主变量是 REXX 中的复合变量，因此，除了声明这些变量以外，还必须设置 NAME、NAME_LENGTH 和 FILE_OPTIONS 字段的值。

在 REXX 嵌入式 SQL 应用程序中清除 LOB 主变量

在基于 Windows 的平台上，可能有必要显式地清除 REXX SQL LOB 定位器和文件引用主变量声明，这是因为，它们在应用程序结束后保持有效。发生这种情况的原因是，应用程序进程直到运行该进程的会话关闭后才会退出。如果未清除 REXX SQL LOB 声明，那么它们可能会在 LOB 应用程序执行完成后干扰同一会话中运行的其他应用程序。

用于清除声明的语法如下所示:

```
CALL SQLEXEC "CLEAR SQL VARIABLE DECLARATIONS"
```

您应该在 LOB 应用程序末尾编码此语句。注意，作为一项预防措施，可以在任何位置编码此语句以清除先前应用程序可能遗留的声明（例如，在 REXX SQL 应用程序开头编码此语句）。

REXX 嵌入式 SQL 应用程序中的 null 或截断指示符变量

REXX 中的指示符变量数据类型是不带小数点的数字。以下是 REXX 中使用 INDICATOR 关键字的指示符变量的示例。

```
CALL SQLEXEC 'FETCH C1 INTO :cm INDICATOR :cmind'
IF ( cmind < 0 )
   SAY 'Commission is NULL'
```

在以上示例中，检查 cmind 的值是否为负数。如果它不是负数，那么应用程序可以使用返回的值 cm。如果它是负数，那么提取的值是 NULL，并且不能使用 cm。在这种情况下，数据库管理器不会更改该主变量的值。

在嵌入式 SQL 应用程序中执行 XQuery 表达式

开始之前

您可以在表中存储 XML 数据并使用嵌入式 SQL 应用程序通过 XQuery 表达式来 XML 列。要访问 XML 数据，请使用 XML 主变量，而不要将数据强制转换为字符或二进制数据类型。如果未使用 XML 主变量，那么访问 XML 数据的最佳替代方法是使用 FOR BIT DATA 或 BLOB 数据类型以避免进行代码页转换。

- 在嵌入式 SQL 应用程序中声明 XML 主变量。

关于此任务

- 必须在静态 SQL SELECT INTO 语句中使用 XML 类型来检索 XML 值。
- 如果将 CHAR、VARCHAR、CLOB 或 BLOB 主变量用于期望 XML 值的输入，那么将在进行缺省空格（STRIP）处理的情况下对该值执行 XMLPARSE 函数操作。否则，必须使用 XML 主变量。

要在嵌入式 SQL 应用程序中直接发出 XQuery 表达式，请在该表达式开头添加“XQUERY”关键字。对于静态 SQL，请使用 XMLQUERY 函数。调用 XMLQUERY 函数时，XQuery 表达式不带“XQUERY”前缀。

下列示例返回样本数据库的表 CUSTOMER 中 XML 文档的数据。

示例 1：通过在 XQuery 表达式开头添加“XQUERY”关键字直接在 C 和 C++ 动态 SQL 中执行 XQuery 表达式

在 C 和 C++ 应用程序中，可以通过以下方法来发出 XQuery 表达式:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
  char stmt[16384];
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

sprintf( stmt, "XQUERY (for $a in db2-fn:xmlcolumn("CUSTOMER.INFO")
        /*:customerinfo[*:addr/*:city = "Toronto"]/@Cid return data($a))");

EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
  EXEC SQL FETCH c1 INTO :xmlblob;
  /* 显示结果 */
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;
```

示例 2：使用 XMLQUERY 函数和 XMLEXISTS 谓词在静态 SQL 中执行 XQuery 表达式

可以静态地对包含 XMLQUERY 函数的 SQL 语句进行准备，如下所示:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS XML AS BLOB( 10K ) xmlblob;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE C1 CURSOR FOR SELECT XMLQUERY(data($INFO/*:customerinfo/@Cid)')
FROM customer
WHERE XMLEXISTS('$INFO/*:customerinfo[*:addr/*:city = "Toronto"]');

EXEC SQL OPEN c1;

while( sqlca.sqlcode == SQL_RC_OK )
{
  EXEC SQL FETCH c1 INTO :xmlblob;
  /* 显示结果 */
```

```
}

EXEC SQL CLOSE c1;
EXEC SQL COMMIT;
```

示例 3: 在 COBOL 嵌入式 SQL 应用程序中执行 XQuery 表达式

在 COBOL 应用程序中，可以通过以下方法来发出 XQuery 表达式:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     01 stmt pic x(80).
   01 xmlBuff USAGE IS SQL TYPE IS XML AS BLOB (10K).
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "XQUERY (for $a in db2-fn:xmlcolumn("CUSTOMER.INFO")/*:customerinfo
            [*:addr/*:city = "Toronto"]/@Cid return data($a)))" TO stmt.
EXEC SQL PREPARE s1 FROM :stmt END-EXEC.
EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.
EXEC SQL OPEN c1 USING :host-var END-EXEC.

*调用 FETCH 和 UPDATE 循环。
Perform Fetch-Loop through End-Fetch-Loop
   until SQLCODE does not equal 0.

EXEC SQL CLOSE c1 END-EXEC.
EXEC SQL COMMIT END-EXEC.

Fetch-Loop Section.
    EXEC SQL FETCH c1 INTO :xmlBuff END-EXEC.
    if SQLCODE not equal 0
       go to End-Fetch-Loop.
* 显示结果
End-Fetch-Loop. exit.
```

在嵌入式 SQL 应用程序中执行 SQL 语句

对于以静态方式执行的语句以及以动态方式执行的语句，在嵌入式 SQL 应用程序中执行 SQL 语句的方法有所不同，尽管它们都使用 EXEC SQL 命令。您在嵌入式 SQL 应用程序的源代码中对静态语句进行硬编码。动态语句与静态语句的不同之处在于，它们在运行时进行编译，并可以使用输入参数进行准备。可以将读取的信息存储到称为“游标”的媒介，游标使用户能够随意滚动数据以及进行适当的更新。SQLCODE、SQLSTATE 和 SQLWARN 中的错误信息对于帮助您诊断应用程序的故障而言是一个非常有用的工具。

嵌入式 SQL 应用程序中的注释

任何应用程序中的注释对于使应用程序代码易于理解而言都至关重要。本节包含有关在嵌入式 SQL 应用程序中添加注释的信息。

C 和 C++ 嵌入式 SQL 应用程序中的注释

使用 C 和 C++ 应用程序时，可以在 EXEC SQL 块中插入 SQL 注释。例如:

```
/* 此处只允许 C 或 C++ 注释 */
 EXEC SQL
    -- 此处允许 SQL 注释、
    /* C 注释或 */
    // C++ 注释
    DECLARE C1 CURSOR FOR sname;
 /* 此处只允许 C 或 C++ 注释 */
```

COBOL 嵌入式 SQL 应用程序中的注释

使用 COBOL 应用程序时，可以在 EXEC SQL 块中插入 SQL 注释。例如:

```
*   请参阅 COBOL 文档以了解注释规则
*   此处只允许 COBOL 注释
    EXEC SQL
```

```
    -- 此处运行 SQL 注释或
*   整行的 COBOL 注释
    DECLARE C1 CURSOR FOR sname END-EXEC.
*   此处只允许 COBOL 注释
```

FORTRAN 嵌入式 SQL 应用程序中的注释

使用 FORTRAN 应用程序时，可以在 EXEC SQL 块中插入 SQL 注释。例如:

```
C     此处只允许 FORTRAN 注释
      EXEC SQL
      + -- 此处允许 SQL 注释和
C     整行的 FORTRAN 注释
      + DECLARE C1 CURSOR FOR sname
      I=7 ! 此处允许行尾 FORTRAN 注释
C     此处只允许 FORTRAN 注释
```

REXX 嵌入式 SQL 应用程序中的注释

在 REXX 应用程序中，不支持 SQL 注释。

在嵌入式 SQL 应用程序中执行静态 SQL 语句

通过使用以下方法，可以在主语言中以静态方式执行 SQL 语句:

- C 或 C++（**tbmod.sqc/tbmod.sqC**）

  下面这三个示例来自 **tbmod** 样本。请参阅此样本以获取完整的程序，此程序说明如何在 C 或 C++ 中修改表数据。

  以下示例说明如何插入表数据:

```
EXEC SQL INSERT INTO staff(id, name, dept, job, salary)
  VALUES(380, 'Pearce', 38, 'Clerk', 13217.50),
        (390, 'Hachey', 38, 'Mgr', 21270.00),
        (400, 'Wagland', 38, 'Clerk', 14575.00);
```

  以下示例说明如何更新表数据:

```
EXEC SQL UPDATE staff
  SET salary = salary + 10000
  WHERE id >= 310 AND dept = 84;
```

  以下示例说明如何从表中删除数据:

```
EXEC SQL DELETE
  FROM staff
  WHERE id >= 310 AND salary > 20000 AND job != 'Sales';
```

- COBOL（**updat.sqb**）

  下面这三个示例来自 **updat** 样本。请参阅此样本以获取完整的程序，此程序说明如何在 COBOL 中修改表数据。

  以下示例说明如何插入表数据:

```
EXEC SQL INSERT INTO staff
    VALUES (999, 'Testing', 99, :job-update, 0, 0, 0)
    END-EXEC.
```

  以下示例说明如何更新表数据，其中，job-update 是对源代码的声明节中声明的主变量的引用:

```
EXEC SQL UPDATE staff
    SET job=:job-update
    WHERE job='Mgr'
    END-EXEC.
```

以下示例说明如何从表中删除数据:

```
EXEC SQL DELETE
    FROM staff
    WHERE job=:job-update
    END-EXEC.
```

在嵌入式 SQL 应用程序中从 SQLDA 结构检索主变量信息

对于静态 SQL 而言，嵌入式 SQL 语句中使用的主变量在应用程序编译时已确定。对于动态 SQL 而言，嵌入式 SQL 语句以及主变量直到应用程序运行时才确定。因此，对于动态 SQL 应用程序而言，您需要处理应用程序中使用的主变量的列表。您可以使用 DESCRIBE 语句来获取任何已使用 PREPARE 准备的 SELECT 语句的主变量信息，并将该信息存储到 SQL 描述符区域（SQLDA）。

在应用程序中执行 DESCRIBE 语句时，数据库管理器将在 SQLDA 中定义主变量。在 SQLDA 中定义主变量之后，您可以使用 FETCH 语句通过游标对主变量赋值。

在动态执行的 SQL 程序中声明 SQLDA 结构

关于此任务

SQLDA 包含数目不定的 SQLVAR 条目实例，每个实例都包含一组用于描述一行数据中某个列的字段，如下图所示。SQLVAR 条目分为两类: 基本 SQLVAR 条目和辅助 SQLVAR 条目。

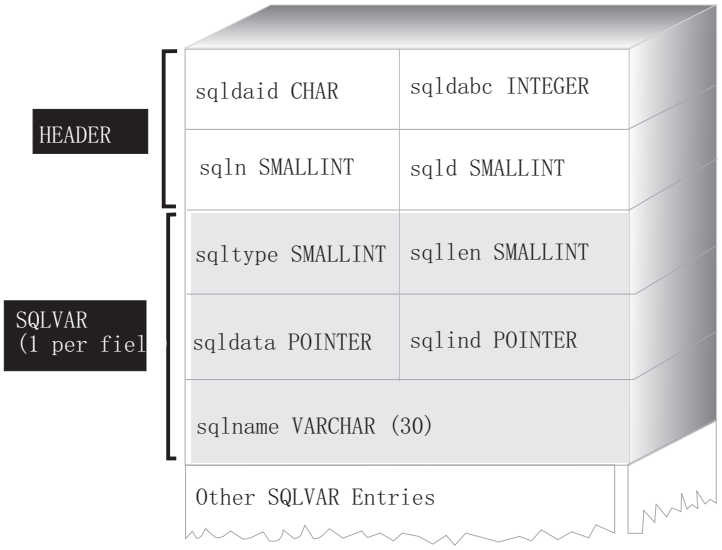


图 2. SQL 描述符区域（SQLDA）

由于所需的 SQLVAR 条目数取决于结果表中的列数，因此应用程序必须能够在需要时分配适当数目的 SQLVAR 元素。请使用下列其中一种方法:

过程

- 提供所需的最大 SQLDA（即，包含最大数目的 SQLVAR 条目）。结果表能够返回的最大列数是 255。如果所返回的任何列具有 LOB 类型或单值类型，那么 SQLN 中的值将翻倍，而存放信息所需的 SQLVAR 条目数将翻倍为 510。但是，由于大多数 SELECT 语句甚至无需检索 255 列，因此所分配的大部分空间都不会被使用。
- 提供包含较少 SQLVAR 条目的较小 SQLDA。在这种情况下，如果结果中的列数超过 SQLDA 所允许的 SQLVAR 条目数，那么不返回任何描述。而是，数据库管理器返回在 SELECT 语句中检测到的选择列表项的数目。应用程序将分配包含所需数目的 SQLVAR 条目的 SQLDA，然后使用 DESCRIBE 语句来获取列描述。
- 如果返回的任何列具有 LOB 或用户定义的类型，请提供包含准确数目的 SQLVAR 条目的 SQLDA。

下一步做什么

对于全部这三种方法，都存在有关应该分配多少初始 SQLVAR 条目这一问题。每个 SQLVAR 元素将使用多达 44 个字节的存储器（为 SQLDATA 和 SQLIND 字段分配的存储器未计算在内）。如果内存足够，那么第一种方法（提供具有最大大小的 SQLDA）较容易实现。

第二种方法（分配较小的 SQLDA）仅适用于支持动态分配内存的编程语言（例如 C 和 C++）。对于不支持动态分配内存的语言（例如 COBOL 和 FORTRAN），请使用第一种方法。

使用最小的 SQLDA 结构来准备以动态方式执行的 SQL 语句

请使用此处提供的信息作为有关如何为语句分配最小 SQLDA 结构的示例。

关于此任务

只能使用支持动态分配内存的编程语言（例如 C 和 C++）来分配较小的 SQLDA 结构。

假定应用程序声明了名为 minsqlda 并且未包含任何 SQLVAR 条目的 SQLDA 结构。SQLDA 的 SQLN 字段描述已分配的 SQLVAR 条目的数目。在这种情况下，必须将 SQLN 设置为 0。接着，要准备字符串 dstring 中的语句并将其描述输入到 minsqlda 中，请发出以下 SQL 语句（假定使用 C 语法，并假定 minsqlda 被声明为指向 SQLDA 结构的指针）：

```
EXEC SQL
  PREPARE STMT INTO :*minsqlda FROM :dstring;
```

假定 dstring 中包含的语句是将在每一行中返回 20 列的 SELECT 语句。在执行 PREPARE 语句（或 DESCRIBE 语句）之后，SQLDA 的 SQLD 字段包含所准备 SELECT 语句的结果表的列数。

在下列情况下，将设置 SQLDA 中的 SQLVAR 条目:

- SQLN >= SQLD，并且没有任何列具有 LOB 或单值类型。

  设置前 SQLD 个 SQLVAR 条目，并将 SQLDOUBLED 设置为空。
- SQLN >= 2*SQLD，并且至少一列具有 LOB 或单值类型。

  设置 2* SQLD 个 SQLVAR 条目，并且将 SQLDOUBLED 设置为 2。

- SQLD <= SQLN < 2*SQLD，并且至少一列具有单值类型，但不存在 LOB 列。

  设置前 SQLD 个 SQLVAR 条目，并将 SQLDOUBLED 设置为空。如果 SQLWARN 绑定选项是 YES，那么将发出警告 SQLCODE +237（SQLSTATE 01594）。

在下列情况下，不会设置 SQLDA 中的 SQLVAR 条目（要求分配其他空间和另一个 DESCRIBE）：

- SQLN < SQLD，并且没有任何列具有 LOB 或单值类型。

  不设置任何 SQLVAR 条目，并将 SQLDOUBLED 设置为空。如果 SQLWARN 绑定选项是 YES，那么将发出警告 SQLCODE +236（SQLSTATE 01005）。

  为成功的 DESCRIBE 分配 SQLD 个 SQLVAR 条目。
- SQLN < SQLD，并且至少一列具有单值类型，但不存在 LOB 列。

  不设置任何 SQLVAR 条目，并将 SQLDOUBLED 设置为空。如果 SQLWARN 绑定选项是 YES，那么将发出警告 SQLCODE +239（SQLSTATE 01005）。

  为成功的 DESCRIBE 分配 2*SQLD 个 SQLVAR 条目，包括单值类型的名称。
- SQLN < 2*SQLD，并且至少一列具有 LOB 类型。

  不设置任何 SQLVAR 条目，并将 SQLDOUBLED 设置为空。发出警告 SQLCODE +238（SQLSTATE 01005），而不考虑 SQLWARN 绑定选项的设置。

  为成功的 DESCRIBE 分配 2*SQLD 个 SQLVAR 条目。

BIND 命令的 SQLWARN 选项用来控制 DESCRIBE（或 PREPARE...INTO）是否将返回下列警告:

- SQLCODE +236（SQLSTATE 01005）
- SQLCODE +237（SQLSTATE 01594）
- SQLCODE +239（SQLSTATE 01005）。

建议应用程序代码始终考虑可能返回这些 SQLCODE 值时的情况。当选择列表包含 LOB 列，并且 SQLDA 中没有足够的 SQLVAR 条目时，将总是返回警告 SQLCODE +238（SQLSTATE 01005）。这是使应用程序了解到由于结果集包含 LOB 列而必须使 SQLVAR 条目数翻倍的唯一方法。

为动态执行的 SQL 语句分配包含足够 SQLVAR 条目的 SQLDA 结构

关于此任务

在确定结果表中的列数之后，请为具有完整大小的第二个 SQLDA 分配存储器。第一个 SQLDA 用于输入参数，具有完整大小的第二个 SQLDA 用于输出参数。

假定结果表包含 20 列（没有任何列是 LOB 列）。在这种情况下，必须分配至少包含 20 个 SQLVAR 元素的第二个 SQLDA 结构 `fulsqlda`（如果结果表包含任何 LOB 或单值类型，那么至少包含 40 个元素）。在此示例的其余部分中，假定结果表不包含 LOB 或单值类型。

在计算 SQLDA 结构的存储器需求时，请包括下列各项:

过程

- 固定长度的头，长度为 16 字节，包含 SQLN 和 SQLD 之类的字段
- 可变长度的 SQLVAR 条目数组，其中，每个元素的长度为 44 个字节（对于 32 位平台）或 56 个字节（对于 64 位平台）。

下一步做什么

fulsqlda 所需的 SQLVAR 条目数由 minsqlda 的 SQLD 字段指定。假定此值为 20。因此，需要为 fulsqlda 分配的存储量为:

```
16 + (20 * sizeof(struct sqlvar))
```

此值表示头的大小加上每个 SQLVAR 条目大小的 20 倍，总计为 896 个字节。

您可以使用 SQLDASIZE 宏来避免自己执行计算以及避免任何特定于版本的依赖关系。

在动态执行的 SQL 程序中描述 SELECT 语句

为第二个 SQLDA（在本示例中，名为 fulsqlda）分配足够的空间之后，必须编写应用程序代码以描述 SELECT 语句。

过程

请编写应用程序代码以执行下列步骤:

1. 在 fulsqlda 的 SQLN 字段中存储值 20（在本示例中，假定结果表包含 20 列，并且这些列都不是 LOB 列）。
2. 使用第二个 SQLDA 结构 fulsqlda 来获取有关 SELECT 语句的信息。有两种可用的方法:
   - 使用另一个 PREPARE 语句并指定 fulsqlda，而不要指定 minsqlda。
   - 使用 DESCRIBE 语句并指定 fulsqlda。

下一步做什么

最好使用 DESCRIBE 语句，这是因为可以避免再次准备语句的开销。DESCRIBE 语句复用先前在准备操作期间获取的信息来填写新的 SQLDA 结构。您可以发出以下语句:

```
EXEC SQL DESCRIBE STMT INTO :fulsqlda
```

执行此语句之后，每个 SQLVAR 元素都包含结果表中某一列的描述。

获取用于存放行的存储器

在应用程序可以使用 SQLDA 结构来提取结果表的行之前，应用程序必须先为该行分配存储器。

过程

对您的应用程序进行编码以执行下列任务:

1. 分析每个 SQLVAR 描述，以确定该列的值所需的空间量。

   注意，对于 LOB 值，在描述 SELECT 时，SQLVAR 中指定的数据类型是 SQL_TYP_*x*LOB。此数据类型与普通的 LOB 主变量相对应，即，一次性地将整个 LOB 存储到内存中。这对小型 LOB（大小可达几 MB）有效，但您不能将此数据类型用于大型 LOB（例如大小为 1 GB 的 LOB），这是因为堆栈无法分配足够的内

存。应用程序有必要将 SQLVAR 中的列定义更改为 SQL_TYP_*x*LOB_LOCATOR 或 SQL_TYPE_*x*LOB_FILE。（注意，更改 SQLVAR 的 SQLTYPE 字段也要求更改 SQLLEN 字段）。在更改 SQLVAR 中的列定义之后，应用程序可以为新类型分配正确的存储量。

2. 为该列的值分配存储器。
3. 将所分配存储器的地址存储到 SQLDA 结构的 SQLDATA 字段中。

下一步做什么

这些步骤的完成方法是，分析每个列的描述，并将每个 SQLDATA 字段的内容替换为大小足以存放该列中任何值的存储区的地址。对于并非 LOB 类型的数据项，长度属性根据每个 SQLVAR 条目的 SQLLEN 字段确定。对于类型为 BLOB、CLOB 或 DBCLOB 的项，长度属性根据第二个 SQLVAR 条目的 SQLLONGLEN 字段确定。

另外，如果指定的列允许 null 值，那么应用程序必须将 SQLIND 字段的内容替换为该列的指示符变量的地址。

在动态执行的 SQL 程序中处理游标

关于此任务

正确地分配 SQLDA 结构之后，可以打开与 SELECT 语句相关联的游标并提取行。

要处理与 SELECT 语句相关联的游标，请先打开该游标，然后通过指定 FETCH 语句的 USING DESCRIPTOR 子句来提取行。例如，C 应用程序可以具有以下行:

```
EXEC SQL OPEN pcurs
EMB_SQL_CHECK( "OPEN" ) ;
EXEC SQL FETCH pcurs USING DESCRIPTOR :*sqldaPointer
EMB_SQL_CHECK( "FETCH" ) ;
```

对于成功的 FETCH，可以将应用程序编写成获取 SQLDA 中的数据并显示列标题。例如:

```
display_col_titles( sqldaPointer ) ;
```

显示数据之后，应该关闭游标并释放任何动态分配的内存。例如:

```
EXEC SQL CLOSE pcurs ;
EMB_SQL_CHECK( "CLOSE CURSOR" ) ;
```

为动态执行的 SQL 程序分配 SQLDA 结构

为应用程序分配 SQLDA 结构，以便可以使用该结构将数据传递到该应用程序以及从该应用程序传递数据。

关于此任务

要在 C 程序中创建 SQLDA 结构，请在主语言中嵌入 INCLUDE SQLDA 语句或者包括 SQLDA 包含文件以获取结构定义。然后，因为 SQLDA 的大小不固定，所以应用程序必须声明指向 SQLDA 结构的指针并为其分配存储器。SQLDA 结构的实际大小取决于使用 SQLDA 传递的单值数据项的数目。

在 C 和 C++ 编程语言中，提供了用于简化 SQLDA 分配工作的宏。此宏的格式如下:

```
#define SQLDASIZE(n) (offsetof(struct sqlda, sqlvar) \
        + (n) x sizeof(struct sqlvar))
```

此宏的效果是，计算包含 n 个 SQLVAR 元素的 SQLDA 所需的存储器。

要在 COBOL 程序中创建 SQLDA 结构，可以嵌入 INCLUDE SQLDA 语句或使用 COPY 语句。如果您想要控制 SQLVAR 条目的最大数目，从而控制 SQLDA 所使用的存储量，请使用 COPY 语句。例如，要将 SQLVAR 条目的缺省数目由 1489 更改为 1，请使用以下 COPY 语句:

```
COPY "sqlda.cbl"
  replacing --1489--
  by --1--.
```

FORTRAN 语言并不直接支持自我定义的数据结构或者动态分配。我们未提供用于 FORTRAN 的 SQLDA 包含文件，这是因为，在 FORTRAN 中，不可能支持 SQLDA 作为数据结构。预编译器将忽略 FORTRAN 程序中的 INCLUDE SQLDA 语句。

但是，可以在 FORTRAN 程序中创建类似于静态 SQLDA 结构的内容，并在可以使用 SQLDA 的位置使用此结构。sqldact.f 文件包含帮助您在 FORTRAN 中声明 SQLDA 结构的常量。

执行 SQLGADDR 调用，以便将指针值赋予需要那些值的 SQLDA 元素。

下表列示包含一个 SQLVAR 元素的 SQLDA 结构的声明和用法。

语言	示例源代码

C 和 C++	`#include`
	`struct sqlda *outda = (struct sqlda *)malloc(SQLDASIZE(1));`
	
	`/* 声明用于存放实际数据的局部变量 */`
	`double sal = 0;`
	`short salind = 0;`
	
	`/* 初始化 SQLDA 的一个元素 */`
	`memcpy( outda->sqldaid,"SQLDA   ",sizeof(outda->sqldaid));`
	`outda->sqln = outda->sqld = 1;`
	`outda->sqlvar[0].sqltype = SQL_TYP_NFLOAT;`
	`outda->sqlvar[0].sqllen  = sizeof( double );.`
	`outda->sqlvar[0].sqldata = (unsigned char *)&sal;`
	`outda->sqlvar[0].sqlind  = (short *)&salind;`

语言	示例源代码
COBOL	

```
 WORKING-STORAGE SECTION.
 77 SALARY          PIC S99999V99 COMP-3.
 77 SAL-IND         PIC S9(4)     COMP-5.

    EXEC SQL INCLUDE SQLDA END-EXEC

* 或者，编码一种有用的方法来保存未使用的 SQLVAR 条目。
* COPY "sqlda.cbl" REPLACING --1489-- BY --1--.

      01 decimal-sqllen pic s9(4) comp-5.
      01 decimal-parts redefines decimal-sqllen.
         05 precision pic x.
         05 scale pic x.

* 初始化输出 SQLDA 的一个元素
      MOVE 1 TO SQLN
      MOVE 1 TO SQLD
      MOVE SQL-TYP-NDECIMAL TO SQLTYPE(1)

* 长度为 7 位精度，并且小数位数为 2

      MOVE x"07" TO PRECISION.
      MOVE x"02" TO SCALE.
      MOVE DECIMAL-SQLLEN TO O-SQLLEN(1).
      SET SQLDATA(1) TO ADDRESS OF SALARY
      SET SQLIND(1)  TO ADDRESS OF SAL-IND
```

语言	示例源代码
FORTRAN	

```
      include 'sqldact.f'

      integer*2  sqlvar1
      parameter ( sqlvar1 = sqlda_header_sz + 0*sqlvar_struct_sz )

C     声明输出 SQLDA -- 1 个变量
      character    out_sqlda(sqlda_header_sz + 1*sqlvar_struct_sz)

      character*8  out_sqldaid     ! 头
      integer*4    out_sqldabc
      integer*2    out_sqln
      integer*2    out_sqld

      integer*2    out_sqltype1    ! 第一个变量
      integer*2    out_sqllen1
      integer*4    out_sqldata1
      integer*4    out_sqlind1
      integer*2    out_sqlnamel1
      character*30 out_sqlnamec1

      equivalence( out_sqlda(sqlda_sqldaid_ofs), out_sqldaid )
      equivalence( out_sqlda(sqlda_sqldabc_ofs), out_sqldabc )
      equivalence( out_sqlda(sqlda_sqln_ofs), out_sqln       )
      equivalence( out_sqlda(sqlda_sqld_ofs), out_sqld       )
      equivalence( out_sqlda(sqlvar1+sqlvar_type_ofs), out_sqltype1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_len_ofs), out_sqllen1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_data_ofs), out_sqldata1 )
      equivalence( out_sqlda(sqlvar1+sqlvar_ind_ofs), out_sqlind1   )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_length_ofs),
     +             out_sqlnamel1                                    )
      equivalence( out_sqlda(sqlvar1+sqlvar_name_data_ofs),
     +             out_sqlnamec1                                    )

C     声明用于存放所返回的数据的局部变量。
      real*8       salary
      integer*2    sal_ind

C     初始化输出 SQLDA（头）
      out_sqldaid  = 'OUT_SQLDA'
      out_sqldabc  = sqlda_header_sz + 1*sqlvar_struct_sz
      out_sqln     = 1
      out_sqld     = 1
C     初始化 VAR1
      out_sqltype1 = SQL_TYP_NFLOAT
      out_sqllen1  = 8
      rc = sqlgaddr( %ref(salary), %ref(out_sqldata1) )
      rc = sqlgaddr( %ref(sal_ind), %ref(out_sqlind1) )
```

注: 此示例是针对 32 位 FORTRAN 编写的。

在不支持动态内存分配的语言中，必须在主语言中明确声明包含所需数目的 SQLVAR 元素的 SQLDA。请确保根据应用程序的需要声明足够的 SQLVAR 元素。

在动态执行的 SQL 程序中使用 SQLDA 结构来传输数据

关于此任务

使用 SQLDA 传输数据比使用主变量列表传输数据更为灵活。例如，可以使用 SQLDA 来传输在本机主语言中没有等同数据的数据，例如 C 语言中的 DECIMAL 数据。

请使用下表作为交叉引用列表，此列表说明数字值与符号名称之间的关系。

表 17. DB2 SQLDA SQL 类型

SQL 列类型	SQLTYPE 数字值	SQLTYPE 符号名称[1]
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
不适用[2]	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
不适用[3]	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
REAL[4]	480/481	SQL_TYP_FLOAT / SQL_TYP_NFLOAT
DECIMAL[5]	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
不适用	804/805	SQL_TYP_BLOB_FILE / SQL_TYPE_NBLOB_FILE
不适用	808/809	SQL_TYP_CLOB_FILE / SQL_TYPE_NCLOB_FILE
不适用	812/813	SQL_TYP_DBCLOB_FILE / SQL_TYPE_NDBCLOB_FILE
不适用	960/961	SQL_TYP_BLOB_LOCATOR / SQL_TYP_NBLOB_LOCATOR
不适用	964/965	SQL_TYP_CLOB_LOCATOR / SQL_TYP_NCLOB_LOCATOR
不适用	968/969	SQL_TYP_DBCLOB_LOCATOR / SQL_TYP_NDBCLOB_LOCATOR
XML	988/989	SQL_TYP_XML / SQL_TYP_XML

注: 在 sqllib 目录的 include 子目录中的 sql.h 包含文件中，您可以找到这些已定义的类型。（例如，对于 C 编程语言，文件为 sqllib/include/sql.h。）

1. 对于 COBOL 编程语言，SQLTYPE 名称不使用下划线（_），而是使用连字符（-）。
2. 这是以 null 结束的图形字符串。
3. 这是以 null 结束的字符串。
4. 在 SQLDA 中，REAL 与 DOUBLE 之间的差别是长度值（4 或 8）。
5. 精度存放在第一个字节中。小数位数存放在第二个字节中。

在动态执行的 SQL 程序中处理交互式 SQL 语句

关于此任务

您可以编写使用动态 SQL 的应用程序以处理任意的 SQL 语句。例如，如果应用程序接受用户提供的 SQL 语句，那么应用程序必须能够在事先不了解这些语句的情况下发

出这些语句。直到执行时才能确定的值可以由参数标记表示，这些参数标记由问号指示。参数标记允许用户与应用程序进行交互，这类似于静态 SQL 语句的主变量。

请将 PREPARE 和 DESCRIBE 语句与 SQLDA 结构配合使用，以使应用程序能够确定正在发出的 SQL 语句的类型并执行相应的操作。

在动态执行的 SQL 程序中确定语句类型

准备 SQL 语句时，可以通过检查 SQLDA 结构来确定关于语句类型的信息。此信息是在语句准备期间通过 INTO 子句放入 SQLDA 结构的，或者是通过对先前准备的语句发出 DESCRIBE 语句放入该结构的。

无论在哪一种情况下，数据库管理器都将在 SQLDA 结构的 SQLD 字段中放置一个值，以指示该 SQL 语句所生成的结果表中的列数。如果 SQLD 字段包含零（0），那么表明该语句*不是* SELECT 语句。由于该语句已准备完毕，因此可以通过 EXECUTE 语句立即执行。

如果该语句包含参数标记，那么必须指定 USING 子句。USING 子句可以指定主变量列表或者 SQLDA 结构。

如果 SQLD 字段大于零，那么表明该语句是 SELECT 语句并且必须按下列各节描述的方式进行处理。

在动态执行的 SQL 程序中处理可变列表 SELECT 语句

可变列表 SELECT 语句是指，要返回的列的数目和类型在预编译时未知。在此情况下，应用程序无法事先知道为了存放结果表的行而需声明的确切主变量。

过程

要处理可变列表 SELECT 语句，请对您的应用程序进行编码以执行以下步骤:

1. 声明 SQLDA。

   必须使用 SQLDA 结构来处理可变列表 SELECT 语句。
2. 使用 INTO 子句来准备语句。

   然后，应用程序确定所声明的 SQLDA 结构是否包含足够的 SQLVAR 元素。如果它没有足够的元素，那么应用程序将分配另一个包含所需数目 SQLVAR 元素的 SQLDA 结构，并使用新的 SQLDA 来发出另一个 DESCRIBE 语句。
3. 分配 SQLVAR 元素。

   为每个 SQLVAR 所需的主变量和指示符分配存储器。此步骤包括将您为数据和指示符变量分配的地址放入每个 SQLVAR 元素。
4. 处理该 SELECT 语句。

   打开与已准备的语句相关联的游标，然后使用正确分配的 SQLDA 结构来访存行。

保存来自最终用户的 SQL 请求

如果应用程序的用户可以从应用程序中发出 SQL 请求，那么您可能想保存这些请求。

关于此任务

如果应用程序允许用户保存任意的 SQL 语句，那么您可以将其保存到列数据类型为 VARCHAR、CLOB、VARGRAPHIC 或 DBCLOB 的表中。注意，VARGRAPHIC 和 DBCLOB 数据类型仅适用于双字节字符集（DBCS）和扩展 UNIX 代码（EUC）环境。

必须保存源 SQL 语句，而不能保存准备完毕的版本。这意味着，在执行表中存储的版本之前，必须检索并准备每个语句。实际上，应用程序根据字符串来准备 SQL 语句并动态地执行此语句。

通过使用参数标记为动态执行的 SQL 语句提供变量输入

在动态 SQL 语句中，由问号（?）或后跟名称的冒号（:*名称*）指示的参数标记用于替换主变量。

关于此任务

动态 SQL 语句不能包含主变量，这是因为只有在应用程序预编译期间才能获得主变量信息（数据类型和长度）；在执行期间，无法获得主变量信息。在动态 SQL 语句中，使用参数标记来代替主变量。参数标记由问号（?）或后跟名称的冒号（:*名称*）指示，它指示 SQL 语句中替换主变量的位置。

例如，假定要使用动态 SQL 语句来根据雇员编号值删除 TEMPL 表中的数据。您可以指定包含参数标记的 DELETE 语句，如下所示:

```
DELETE FROM TEMPL WHERE EMPNO = ?
```

要执行此语句，请为 EXECUTE 语句的 USING 子句指定主变量或 SQLDA 结构。主变量的内容将用来指定 EMPNO 的值。

如果未使用延迟准备功能，即，未设置注册表变量 **DB2_DEFERRED_PREPARE_SEMANTICS** 或者将其设置为 NO，那么参数标记的数据类型和长度取决于 SQL 语句中参数标记的上下文。如果无法根据使用参数标记的语句的上下文来确定参数标记的数据类型，请使用 CAST 规范来指定数据类型。您对其使用了 CAST 规范的参数标记被称为*带类型参数标记*。带类型参数标记被视为 CAST 规范中使用的数据类型的主变量。例如，语句 `SELECT ? FROM SYSCAT.TABLES` 无效，这是因为结果列的数据类型未知。但是，语句 `SELECT CAST(? AS INTEGER) FROM SYSCAT.TABLES` 有效，这是因为 CAST 规范指示该参数标记表示整数值；结果列的数据类型已知。

如果正在使用延迟准备功能，即，将注册表变量 **DB2_DEFERRED_PREPARE_SEMANTICS** 设置为 YES，那么语句的准备工作将推迟到您发出 OPEN 或 EXECUTE 语句时进行。准备语句时，假定参数标记的类型与 SQLDA 中相应主变量或信息的类型匹配。在语句编译期间，SQL 编译器可能会根据参数标记在语句中的上下文来优化参数标记的类型。如果正在使用延迟准备功能，那么语句 `SELECT ? FROM SYSCAT.TABLES` 有效，并且结果列的类型基于与参数标记绑定的主变量的类型。

如果 SQL 语句包含多个参数标记，那么 EXECUTE 语句的 USING 子句必须指定下列其中一种类型的信息:

- 主变量的列表（每个参数标记都有一个对应的变量）
- 一个 SQLDA（对于非 LOB 数据类型的每个参数标记，此 SQLDA 包含一个 SQLVAR 条目；或者，对于 LOB 数据类型的每个参数标记，此 SQLDA 包含两个条目）

主变量列表或 SQLVAR 条目根据语句中参数标记的顺序进行匹配，数据类型必须兼容。

注: 在动态 SQL 语句中使用参数标记类似于在静态 SQL 语句中使用主变量，即，优化器不使用分布统计信息，并且可能不会选择最佳的存取方案。

PREPARE 语句主题描述了适用于参数标记的规则。

以动态方式执行的 SQL 程序中的参数标记示例

下列示例说明如何在动态 SQL 程序中使用参数标记:

- C 和 C++（**dbuse.sqc/dbuse.sqC**）

  C 语言样本 **dbuse.sqc** 中的函数 DynamicStmtWithMarkersEXECUTEusingHostVars() 说明如何通过主变量使用参数标记来执行删除:

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;

/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

- COBOL（**varinp.sqb**）

  来自 COBOL 样本 **varinp.sqb** 的以下示例说明如何在搜索条件和更新条件中使用参数标记:

```
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 pname             pic x(10).
01 dept              pic s9(4) comp-5.
01 st                pic x(127).
01 parm-var          pic x(5).
   EXEC SQL END DECLARE SECTION END-EXEC.

   move "SELECT name, dept FROM staff
-         "   WHERE job = ? FOR UPDATE OF job" to st.
   EXEC SQL PREPARE s1 FROM :st END-EXEC.

   EXEC SQL DECLARE c1 CURSOR FOR s1 END-EXEC.

   move "Mgr" to parm-var.
   EXEC SQL OPEN c1 USING :parm-var END-EXEC

   move "Clerk" to parm-var.
   move "UPDATE staff SET job = ? WHERE CURRENT OF c1" to st.
   EXEC SQL PREPARE s2 from :st END-EXEC.

* call the FETCH and UPDATE loop.
   perform Fetch-Loop thru End-Fetch-Loop
      until SQLCODE not equal 0.

   EXEC SQL CLOSE c1 END-EXEC.
```

在嵌入式 SQL 应用程序中调用过程

通过构造并执行包含适当过程引用和参数的 CALL 语句，可以从嵌入式 SQL 应用程序中调用过程。可以从嵌入式 SQL 应用程序中静态或动态地发出 CALL 语句。但是，对

于每种编程语言，发出此命令的方法有所不同。无论使用哪种主语言，都必须声明过程中使用的每个主变量以使其与所需的数据类型匹配。

客户机应用程序和例程调用通过参数和结果集与过程交换信息。过程的参数由数据的传输方向定义（参数方式）。

过程的参数分为三类:

- IN 参数: 数据被传递到过程。
- OUT 参数: 数据由过程返回。
- INOUT 参数: 数据被传递到过程，在过程执行期间，该数据被过程所返回的数据替换。

参数的方式及其数据类型是在您使用 CREATE PROCEDURE 语句来注册过程时定义的。

在 C 和 C++ 嵌入式 SQL 应用程序中调用存储过程

在 C 和 C++ 嵌入式 SQL 应用程序中调用存储过程

DB2 支持在 SQL 过程中使用输入、输出以及输入输出参数。CREATE PROCEDURE 语句中的关键字 IN、OUT 和 INOUT 指示参数的方式或预期用法。IN 和 OUT 参数按值传递，INOUT 参数按引用传递。

使用 C 和 C++ 应用程序时，可以使用以下语句来调用存储过程 INOUT_PARAM:

```
EXEC SQL CALL INOUT_PARAM(:inout_median:medianind, :out_sqlcode:codeind,
                          :out_buffer:bufferind);
```

其中，inout_median、out_sqlcode 和 out_buffer 是主变量，而 medianind、codeind 和 bufferind 是 null 指示符变量。

注: 另外，还可以通过准备 CALL 语句以动态方式调用存储过程。

从 REXX 中调用存储过程

可以使用服务器上的任何受支持语言（AIX 系统上的 REXX 除外）来编写存储过程。（在 AIX 系统上，可以使用 REXX 来编写客户机应用程序，但与其他语言相同，在 AIX 上，它们不能调用使用 REXX 编写的存储过程。）

在嵌入式 SQL 应用程序中读取和滚动结果集

嵌入式 SQL 应用程序的其中一项最常用任务是检索数据。此任务通过 *SELECT 语句*完成，该语句是一种查询形式，它在数据库中搜索符合所指定搜索条件的表行。如果存在这样的行，那么将检索数据并将其放入主程序中的指定变量，在该位置，它可以用于所设计的任何用途。

注: 嵌入式 SQL 应用程序可以通过任何受支持的存储过程实现来调用存储过程，并可以检索输出参数值和输入输出参数值，但嵌入式 SQL 应用程序无法读取和滚动存储过程所返回的结果集。

在编写 SELECT 语句之后，您应该编码 SQL 语句以定义将信息传递到应用程序的方式。

您可以将 SELECT 语句的结果想像成一个由行和列组成的表，这与数据库中的表非常相似。如果只返回了一行，那么可以将结果直接传递给 SELECT INTO 语句所指定的主变量。

如果返回了多行，那么必须使用*游标*来每次提取一行。游标是由应用程序使用并具有名称的控制结构，用于指向有序行集中的特定行。

在嵌入式 SQL 应用程序中滚动先前检索的数据

关于此任务

应用程序从数据库检索数据时，FETCH 语句允许它向前滚动数据，但是，没有任何 SQL 语句允许向后滚动结果集（相当于向后 FETCH）。但是，CLI 和 DB2 通用 JDBC 驱动程序通过只读的可滚动游标来支持向后 FETCH。

过程

对于嵌入式 SQL 应用程序，您可以使用下列技术来滚动已检索的数据:

- 在应用程序内存中保存已访存的数据的副本，并通过一些编程技术对其进行滚动。
- 使用 SQL 来再次检索数据（通常使用第二个 SELECT 语句进行）。

在嵌入式 SQL 应用程序中保留已提取的数据的副本

在某些情况下，保留应用程序所提取的数据的副本非常有用。

过程

要保留数据的副本，您的应用程序可执行下列其中一项任务:

- 在虚拟存储器中保存已提取的数据。
- 将数据写入临时文件（如果该数据在虚拟存储器中装不下的话）。此方法的一个效果是，向后滚动的用户总是看到那些已提取的数据，即使数据库中的数据在此期间已被某个事务更改亦如此。
- 通过使用“可重复读”隔离级别，可以通过关闭并打开游标来再次检索从事务检索的数据。不允许其他应用程序更新结果集中的数据。隔离级别和锁定可能会影响用户更新数据的方式。

在嵌入式 SQL 应用程序中再次检索已提取的数据

用于再次检索数据的技术取决于您期望以何顺序再次查看数据。

过程

您可以通过使用下列任何方法来再次检索数据:

- 从头开始检索数据

  要从结果表开头再次检索数据，请关闭活动游标，然后将其重新打开。此操作将游标定位在结果表的开头。但是，除非应用程序对表挂起锁定，否则其他应用程序可能已对其进行更改，因此，结果表以前的第一行可能已不再是第一行。
- 从中间开始检索数据

  要从结果表中间的某个位置开始再次检索数据，请发出第二个 SELECT 语句并在该语句中声明第二个游标。例如，假定第一个 SELECT 语句是:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

现在，假定您要返回到以 DEPTNO = 'M95' 开头的行并从该位置开始按顺序执行访存。编码以下语句:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'M95'
  ORDER BY DEPTNO
```

此语句将游标定位在您所期望的位置。

- 按反向顺序检索数据

  升序是行的缺省顺序。如果对于 DEPTNO 的每个值只有一行，那么以下语句将指定行的唯一升序:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO
```

  要按方向顺序来检索相同的行，请指定降序，如以下语句所示:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO DESC
```

  第二个语句中的游标按第一个语句中的游标的反向顺序来检索行。仅当第一个语句指定了唯一的排序顺序时，检索顺序才有保证。

  要按相反顺序来检索行，可能最好构建两个基于 DEPTNO 列的索引，其中一个采用升序顺序，另一个采用降序顺序。

结果表中的行顺序差别

同一个 SELECT 语句的多个结果表中的行可能会以不同顺序出现。除非 SELECT 语句使用了 ORDER BY，否则数据库管理器认为行顺序并不重要。因此，如果存在多个具有相同 DEPTNO 值的行，那么第二个 SELECT 语句可能会以不同于第一个 SELECT 语句的顺序来检索这些行。唯一的保证是，这些行全都按 ORDER BY DEPTNO 子句的要求以部门号顺序出现。

即使再次发出同一个 SQL 语句并且主变量也相同，顺序方面也可能会有差别。例如，在这两次执行之间，可能更新了目录中的统计信息或者创建或删除了索引。然后，您可以再次发出该 SELECT 语句。

如果第二个 SELECT 语句包含第一个 SELECT 语句所没有的谓词，那么顺序更有可能会有变化；数据库管理器可能会选择使用基于新谓词的索引。例如，它可能为示例中的第一个语句选择基于 LOCATION 的索引，并为第二个语句选择基于 DEPTNO 的索引。由于按索引键的顺序提取行，因此，第二次的顺序不需要与第一次的顺序相同。

同样，执行两个相似的 SELECT 语句可能会产生不同的行顺序，即使统计信息未有变化并且未创建或删除任何索引亦如此。在示例中，如果存在许多不同的 LOCATION 值，那么数据库管理器可能会为这两个语句都选择基于 LOCATION 的索引。但是，按如下示例更改第二个语句中 DEPTNO 的值会导致 数据库管理器 选择 DEPTNO 的索引:

```
SELECT * FROM DEPARTMENT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >= 'Z98'
  ORDER BY DEPTNO
```

由于 SQL 语句格式与此语句中的值之间存在微妙的关系，因此，除非通过 ORDER BY 子句来唯一地确定顺序，否则永远不应假定两个不同的 SQL 语句将按相同的顺序返回行。

在嵌入式 SQL 应用程序中更新先前检索的数据

要向后滚动并更新先前检索的数据，您可以使用一组用来滚动先前检索的数据和更新已检索的数据的技术。

过程

要更新先前检索的数据，您可以执行下列两项操作中的一项:

- 如果已对所要更新的数据打开第二个游标，并且 SELECT 语句未使用任何受限元素，那么可以使用由游标控制的 UPDATE 语句。请在 WHERE CURRENT OF 子句中指定第二个游标。
- 在其他情况下，请使用带有 WHERE 子句的 UPDATE 并在该子句中指定行中所有的值或者指定表的主键。可以在对变量指定不同的值的情况下多次发出同一个语句。

在嵌入式 SQL 应用程序中使用游标来选择多行

为了允许应用程序检索一组行，SQL 使用了称为*游标*的机制。

关于此任务

为了帮助您理解游标的概念，假定数据库管理器构建了一个*结果表*用于存放所有通过执行 SELECT 语句检索到的行。游标通过标识或指向结果表的*当前行*，使此表中所有的行均可供应用程序使用。使用游标时，应用程序可以按顺序检索结果表中的每一行，直至符合数据结束条件，即，“找不到”条件 SQLCODE +100（SQLSTATE 02000）。作为执行 SELECT 语句的结果获取的行集可以由零行、一行或多行组成，这取决于满足搜索条件的行数。

过程

要处理某个游标，请执行以下操作:

1. 使用 DECLARE CURSOR 语句来指定光标。
2. 使用 OPEN 语句来执行查询并构建结果表。
3. 使用 FETCH 语句每次检索一行。
4. 如果有需要的话，使用 DELETE 或 UPDATE 语句来处理行。
5. 使用 CLOSE 语句来终止游标。

下一步做什么

应用程序可以同时使用多个游标。每个游标都需要它自己的一组 DECLARE CURSOR、OPEN、CLOSE 和 FETCH 语句。

在以静态方式执行的 SQL 应用程序中更新和删除所检索的数据

关于此任务

您可以更新和删除游标所引用的行。要使行可更新，与游标相对应的查询不能是只读查询。

要使用游标进行更新，请在 UPDATE 语句中使用 WHERE CURRENT OF 子句。使用 FOR UPDATE 子句来通知系统您要更新结果表的某些列。FOR UPDATE 中指定的列不必包含在全查询中；因此，可以更新游标未显式检索的列。如果指定了 FOR UPDATE 子句，但未指定列名，那么将认为外部全查询中第一个 FROM 子句所标识的表或视图中所有的列都可更新。指定的列数不应超过 FOR UPDATE 子句所需的列数。在某些情况下，在 FOR UPDATE 子句中指定额外的列可能会导致 DB2 访问数据时的效率下降。

使用游标执行删除的操作通过 DELETE 语句中的 WHERE CURRENT OF 子句完成。通常，删除游标的当前行时，不需要指定 FOR UPDATE 子句。唯一的例外情况是，对应用程序中的 SELECT 语句或 DELETE 语句使用动态 SQL，而该应用程序在 LANGLEVEL 设置为 SAA1 的情况下进行预编译并在指定了 BLOCKING ALL 的情况下进行绑定。在这种情况下，必须在 SELECT 语句中指定 FOR UPDATE 子句。

DELETE 语句将导致删除游标所引用的行。此删除操作将使游标定位在下一行之前，在对该游标执行其他 WHERE CURRENT OF 操作之前，必须发出 FETCH 语句。

以静态方式执行的 SQL 程序中的提取示例

以下样本使用游标对表执行选择，打开游标，然后从表中提取行。对于所提取的每一行，此程序根据简单条件确定是必须删除还是更新该行。

REXX 语言不支持静态 SQL，因此未提供样本。

- C 和 C++（**tbmod.sqc/tbmod.sqC**）

  以下示例使用游标对表执行选择，打开游标，提取、更新或删除表中的行，然后关闭游标。

```
EXEC SQL DECLARE c1 CURSOR FOR SELECT * FROM staff WHERE id >= 310;
  EXEC SQL OPEN c1;
  EXEC SQL FETCH c1 INTO :id, :name, :dept, :job:jobInd, :years:yearsInd, :salary,
     :comm:commInd;
```

  此样本几乎描述了所有可能的表数据修改情况。

- COBOL（**openftch.sqb**）

  以下示例来自样本 **openftch**。此示例使用游标对表执行选择，打开游标，然后从表中提取行。

```
 EXEC SQL DECLARE c1 CURSOR FOR
   SELECT name, dept FROM staff
   WHERE job='Mgr'
   FOR UPDATE OF job END-EXEC.

 EXEC SQL OPEN c1 END-EXEC


* call the FETCH and UPDATE/DELETE loop.
```

```
    perform Fetch-Loop thru End-Fetch-Loop
    until SQLCODE not equal 0.

   EXEC SQL CLOSE c1 END-EXEC.
```

在嵌入式 SQL 应用程序中检索错误消息

根据用于编写应用程序的语言不同，可以使用不同的方法来检索错误信息:

- C、C++ 和 COBOL 应用程序可以使用 GET ERROR MESSAGE API 来获取与传入的 SQLCA 相关的相应信息。

```
C 示例: UTILAPI.C 中的 SqlInfoPrint 过程
/******************************************************************************
** 1.1 - SqlInfoPrint - prints diagnostic information to the screen.
**
******************************************************************************/
int SqlInfoPrint( char * appMsg,
    struct sqlca * pSqlca,
    int line,
    char * file )
{   int  rc = 0;
    char sqlInfo[1024];
    char sqlInfoToken[1024];
    char sqlstateMsg[1024];
    char errorMsg[1024];
    if (pSqlca->sqlcode != 0 && pSqlca->sqlcode != 100)
    {  strcpy(sqlInfo, "");
       if( pSqlca->sqlcode < 0)
       {   sprintf( sqlInfoToken, "\n---- error report ----\n");
           strcat( sqlInfo, sqlInfoToken);
       }
       else
       {   sprintf( sqlInfoToken, "\n---- warning report ----\n");
           strcat( sqlInfo, sqlInfoToken);
       } /* endif */

       sprintf( sqlInfoToken, "  app. message      = %s\n", appMsg);
       strcat( sqlInfo, sqlInfoToken);
       sprintf( sqlInfoToken, "  line              = %d\n", line);
       strcat( sqlInfo, sqlInfoToken);
       sprintf( sqlInfoToken, "  file              = %s\n", file);
       strcat( sqlInfo, sqlInfoToken);
       sprintf( sqlInfoToken, "  SQLCODE           = %ld\n",
       pSqlca->sqlcode);
       strcat( sqlInfo, sqlInfoToken);

       /* get error message */
       rc = sqlaintp( errorMsg, 1024, 80, pSqlca);
       /* return code is the length of the errorMsg string */
       if( rc > 0)
       { sprintf( sqlInfoToken, "%s\n", errorMsg);
         strcat( sqlInfo, sqlInfoToken);
       }

       /* get SQLSTATE message */
       rc = sqlogstt( sqlstateMsg, 1024, 80, pSqlca->sqlstate);
       if (rc == 0)
       { sprintf( sqlInfoToken, "%s\n", sqlstateMsg);
         strcat( sqlInfo, sqlInfoToken);
       }

       if( pSqlca->sqlcode < 0)
       { sprintf( sqlInfoToken, "--- end error report ---\n");
         strcat( sqlInfo, sqlInfoToken);

         printf("%s", sqlInfo);
```

```
        return 1;
      }
      else
      { sprintf( sqlInfoToken, "--- end warning report ---\n");
        strcat( sqlInfo, sqlInfoToken);

        printf("%s", sqlInfo);
        return 0;
      } /* endif */
    } /* endif */
    return 0;
}
```

C 开发者还可以使用等价函数 sqlglm(), 该函数具有以下特征符:

```
sqlglm(char *message_buffer_ptr, int *buffer_size_ptr, int *msg_size_ptr)
```

COBOL 示例: 来自 CHECKERR.CBL

```
    *********************************
    * GET ERROR MESSAGE API called *
    *********************************
      call "sqlgintp" using
           by value buffer-size
           by value line-width
           by reference sqlca
           by reference error-buffer
        returning error-rc.
    ************************
    * GET SQLSTATE MESSAGE *
    ************************
      call "sqlggstt" using
           by value buffer-size
           by value line-width
           by reference sqlstate
           by reference state-buffer
        returning state-rc.
    if error-rc is greater than 0
      display error-buffer.

    if state-rc is greater than 0
      display state-buffer.

    if state-rc is less than 0
      display "return code from GET SQLSTATE =" state-rc.

    if SQLCODE is less than 0
      display "--- end error report ---"
      go to End-Prog.

    display "--- end error report ---"
    display "CONTINUING PROGRAM WITH WARNINGS!".
```

- REXX 应用程序使用 CHECKERR 过程。

```
/******   CHECKERR - Check SQLCODE *****/
CHECKERR:
  arg errloc

  if  ( SQLCA.SQLCODE = 0 ) then
    return 0
  else do
    say '--- error report ---'
    say 'ERROR occurred :' errloc
    say 'SQLCODE :' SQLCA.SQLCODE

   /*********************\
   * GET ERROR MESSAGE  *
   \*********************/
```

```
      call SQLDBS 'GET MESSAGE INTO :errmsg LINEWIDTH 80'
      say errmsg
      say '--- end error report ---'

      if (SQLCA.SQLCODE < 0 ) then
        exit
      else do
        say 'WARNING - CONTINUING PROGRAM WITH ERRORS'
        return 0
      end
    end
  return 0
```

SQLCODE、SQLSTATE 和 SQLWARN 字段中的错误信息

错误信息在 SQLCA 结构的 SQLCODE 和 SQLSTATE 字段中返回，该结构在每个可执行 SQL 语句以及大部分数据库管理器 API 调用之后被更新。

包含可执行 SQL 语句的源文件可以至少提供一个名为 sqlca 的 SQLCA 结构。SQLCA 结构在 SQLCA 包含文件中定义。未包含嵌入式 SQL 语句但调用了数据库管理器 API 的源文件还可以提供一个或多个 SQLCA 结构，但它们可以具有任意的名称。

如果应用程序符合 FIPS 127-2 标准，那么对于 C、C++、COBOL 和 FORTRAN 应用程序，可以将 SQLSTATE 和 SQLCODE 声明为主变量，以代替使用 SQLCA 结构。

SQLCODE 值 0 表示执行成功（可能存在 SQLWARN 警告情况）。正数值表示语句已成功地执行完成，但出现警告，例如主变量被截断。负数值表示发生错误情况。

另一个字段 SQLSTATE 包含标准化的错误代码，此错误代码在其他 IBM 数据库产品之间以及符合 SQL92 的数据库管理器之间保持一致。在实践中，当您关心可移植性时，应该使用 SQLSTATE 值，这是因为 SQLSTATE 值在许多数据库管理器之间保持一致。

SQLWARN 字段包含警告指示符数组，即使 SQLCODE 为零亦如此。如果 SQLWARN 数组中除第一个元素 SQLWARN0 以外的所有其他元素均为空，那么第一个元素也为空。如果除 SQLWARN0 以外的至少一个元素包含警告字符，那么 SQLWARN0 包含 W。

注：如果您想开发需要访问各种 IBM RDBMS 服务器的应用程序，那么应该：

- 有可能时，让应用程序检查 SQLSTATE 而不是 SQLCODE。
- 如果应用程序将使用 DB2 Connect，请考虑使用 DB2 Connect 提供的映射设施在不相似的数据库之间映射 SQLCODE 转换。

出口列表例程注意事项

请不要在出口列表例程中使用 SQL 或 DB2 API 调用。注意，在出口例程中，无法与数据库断开连接。

异常、信号和中断处理程序注意事项

异常、信号或中断处理程序是发生异常、信号或中断时获得控制权的例程。适用的处理程序的类型由操作环境确定。

Windows 操作系统

按 Ctrl-C 或 Ctrl-Break 将生成中断。

UNIX 操作系统

通常，按 Ctrl-C 将生成 SIGINT 中断信号。注意，可以方便地对键盘进行重新定义，以使 SIGINT 可以由机器上的另一个键序列生成。

请不要在异常、信号和中断处理程序中包括 SQL 语句。对于这些类型的错误情况，您通常想要执行 ROLLBACK 以避免数据不一致的风险。在发出 ROLLBACK 之前，调用 INTERRUPT API (sqleintr/sqlgintr)。此 API 将中断应用程序正在执行的当前 SQL 查询并允许 ROLLBACK 立即开始。

请参阅平台文档，以了解有关各种处理程序注意事项的具体详细信息。

与嵌入式 SQL 应用程序断开连接

使用数据库时，最后一个步骤是执行断开连接语句。本主题提供受支持主语言中的断开连接语句的示例。

在 C 和 C++ 嵌入式 SQL 应用程序中与 DB2 数据库断开连接

使用 C 和 C++ 应用程序时，通过发出以下语句来关闭数据库连接:

```
EXEC SQL CONNECT RESET;
```

在 COBOL 嵌入式 SQL 应用程序中与 DB2 数据库断开连接

使用 COBOL 应用程序时，通过发出以下语句来关闭数据库连接:

```
EXEC SQL CONNECT RESET END-EXEC.
```

在 REXX 嵌入式 SQL 应用程序中与 DB2 数据库断开连接

使用 REXX 应用程序时，通过发出以下语句来关闭数据库连接:

```
CALL SQLEXEC 'CONNECT RESET'
```

使用 FORTRAN 应用程序时，通过发出以下语句来关闭数据库连接:

```
EXEC SQL CONNECT RESET
```

第 4 章 构建嵌入式 SQL 应用程序

在创建嵌入式 SQL 应用程序的源代码之后，必须执行附加的步骤才能构建该应用程序。在开发新的嵌入式 SQL 数据库应用程序时，您应该考虑构建 64 位可执行文件。除编译和链接程序以外，还必须对其进行*预编译*和*绑定*。

预编译过程将嵌入式 SQL 语句转换为主语言编译器能够处理的 DB2 运行时 API 调用。缺省情况下，将在预编译时创建程序包。（可选）在预编译时，还可以创建绑定文件。绑定文件包含有关应用程序中的 SQL 语句的信息。以后，可以将绑定文件与 BIND 命令配合使用，以便为该应用程序创建程序包。

“绑定”是指根据绑定文件创建*程序包*并将其存储到数据库的过程。绑定文件必须与应用程序需要访问的每个数据库绑定。如果应用程序访问多个数据库，那么必须为每个数据库创建一个程序包。

要运行使用编译型主语言编写的应用程序，必须创建数据库管理器在执行时所需的程序包。下图显示这些步骤的顺序以及已编译的典型 DB2 应用程序的各个模块:

1. 创建包含带有嵌入式 SQL 语句的程序的源文件。
2. 连接到数据库，然后预编译每个源文件以便将嵌入式 SQL 源代码语句转换为数据库管理器能够使用的形式。

   由于应用程序中指定的 SQL 语句并非特定于主语言，因此数据库管理器提供了一种方法来转换 SQL 语法，以便由主语言进行处理。对于 C、C++、COBOL 或 FORTRAN 语言，此转换由使用 PRECOMPILE（或 PREP）命令调用的 DB2 预编译器处理。预编译器将嵌入式 SQL 语句直接转换为 DB2 运行时服务 API 调用。预编译器在处理源文件时，它专门查找 SQL 语句并避免处理非 SQL 主语言。
3. 通过使用主语言编译器，对经过修改的源文件（和其他不包含 SQL 语句的文件）进行编译。
4. 将对象文件与 DB2 和主语言库链接，以便生成可执行程序。

   编译和链接（步骤 3 和 4）将创建所需的对象模块。
5. 如果尚未在执行预编译时创建程序包，或者将要访问另一个数据库，请对绑定文件进行绑定以创建程序包。绑定操作将创建数据库管理器要在该程序运行时使用的程序包。
6. 运行该应用程序。该应用程序将使用访问方案来访问该数据库。

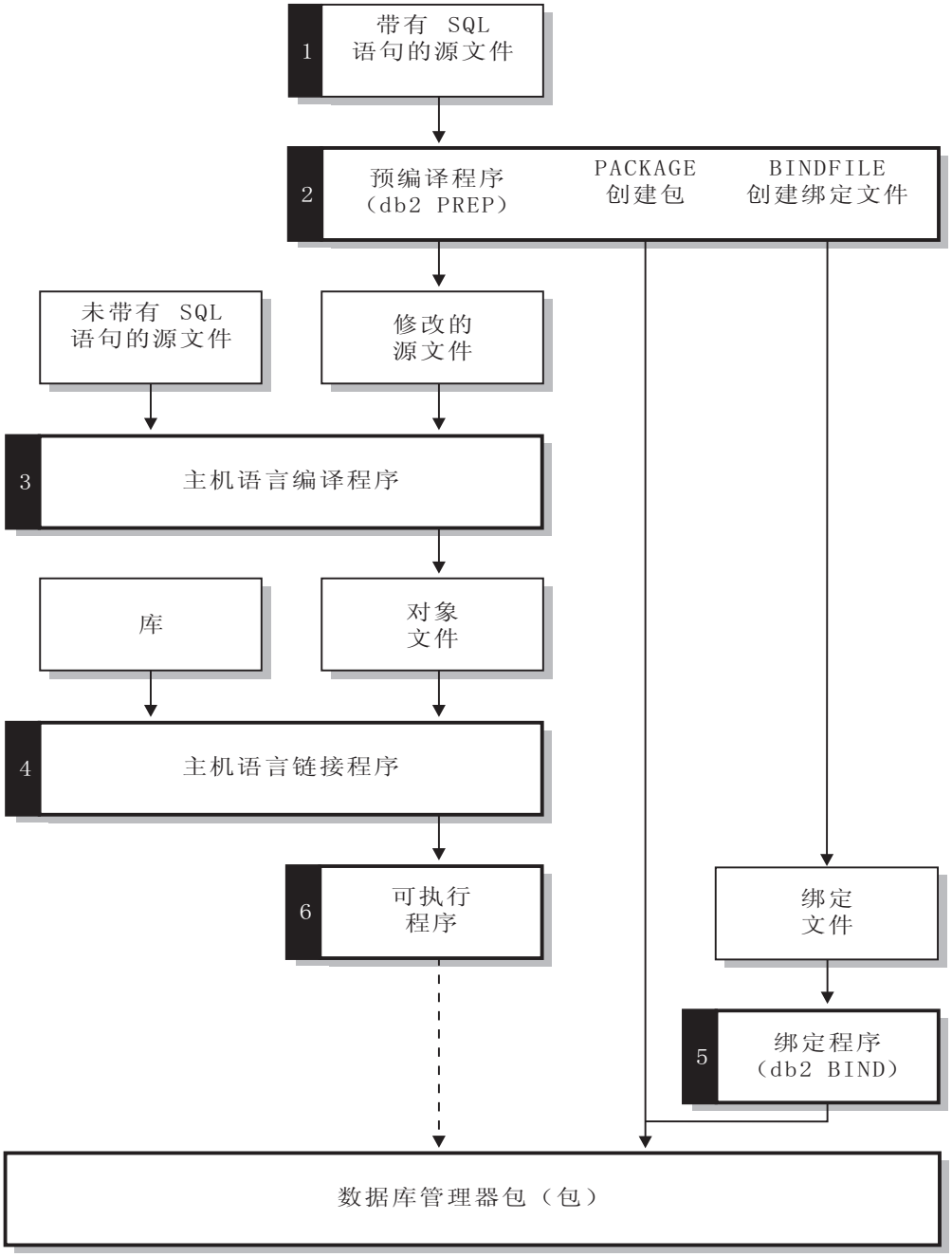


图 3. 对使用编译型主语言编写的程序进行准备

使用 PRECOMPILE 命令来预编译嵌入式 SQL 应用程序

创建嵌入式 SQL 应用程序的源文件之后，必须使用 **PREP** 命令（并指定特定于主语言的选项）对包含 SQL 语句的每个主语言文件进行预编译。

预编译器将源文件中包含的 SQL 语句转换为注释，并为那些语句生成 DB2 运行时 API 调用。

必须始终针对特定数据库预编译源文件，即使最终并不将该数据库与该应用程序配合使用亦如此。在实践中，您可以使用测试数据库来进行开发，在对应用程序进行充分测试之后，可以将它的绑定文件与一个或多个生产数据库绑定。此实践称为*延期绑定*。

注：不支持在版本比执行预编译的客户机低的客户机上运行嵌入式应用程序，这与该应用程序的编译位置无关。例如，在 DB2 V9.5 客户机上预编译嵌入式应用程序之后，不支持尝试在 DB2 V9.1 客户机上运行该应用程序。
如果应用程序使用的代码页与数据库代码页不同，那么您需要考虑执行预编译时使用哪个代码页。

如果应用程序使用用户定义的函数 (UDF) 或用户定义的单值类型 (UDT)，那么在预编译应用程序时，可能需要使用 **FUNCPATH** 参数。此参数指定用来为包含静态 SQL 的应用程序解析 UDF 和 UDT 的函数路径。如果未指定 **FUNCPATH**，那么缺省函数路径是 SYSIBM，SYSFUN，USER，其中 *USER* 是指当前用户标识。

在预编译应用程序之前，必须以隐式或显式方式连接到服务器。虽然您在客户机工作站上预编译应用程序，并且预编译器在客户机上生成经过修改的源代码和消息，但预编译器使用服务器连接来执行部分验证。

预编译器还将创建数据库管理器对数据库处理 SQL 语句时所需的信息。此信息存储在程序包和/或绑定文件中，这取决于您选择的预编译器选项。

下面是使用预编译器的典型示例。要预编译名为 filename.sqc 的 C 嵌入式 SQL 源文件，您可以发出以下命令以创建缺省名称为 filename.c 的 C 源文件以及缺省名称为 filename.bnd 的绑定文件:

```
DB2 PREP filename.sqc BINDFILE
```

预编译器生成多达四种类型的输出:

经过修改的源代码
在预编译器将 SQL 语句转换为 DB2 运行时 API 调用后，此文件是原始源文件生成的新版本。将为其指定适当的主语言扩展名。

程序包 如果使用 **PACKAGE** 参数（缺省值），或者未指定任何 **BINDFILE**、**SYNTAX** 或 **SQLFLAG** 参数，那么程序包将存储在所连接的数据库中。程序包只包含对此数据库发出特定源文件的静态 SQL 语句所需的所有信息。除非您使用 **PACKAGE USING** 参数来指定另一个名称，否则预编译器将使用源文件名的前 8 个字符来构造程序包名。

如果在未指定 **SQLERROR CONTINUE** 的情况下使用 **PACKAGE** 参数，那么预编译过程中使用的数据库必须包含源文件中的静态 SQL 语句所引用的所有数据库对象。例如，除非 SELECT 语句所引用的表存在于数据库中，否则无法预编译该语句。

借助 **VERSION** 参数，将对绑定文件（如果使用了 **BINDFILE** 参数）和程序包（如果在 **PREP** 时进行绑定或者单独进行绑定）指定特定的版本标识。多个具有相同名称和创建者的程序包版本可以同时存在。

绑定文件
如果使用 **BINDFILE** 参数，那么预编译器将创建一个绑定文件（扩展名为 .bnd），此文件包含创建程序包时所需的数据。以后，可以将此文件与 **BIND** 命令配合使用，以便将该应用程序与一个或多个数据库绑定。如果指定 **BINDFILE**

并且未指定 **PACKAGE** 参数，那么绑定将被延迟到您调用 **BIND** 命令时进行。注意，对于命令行处理器 (CLP) 而言，**PREP** 在缺省情况下不指定 **BINDFILE** 参数。因此，如果您正在使用 CLP，并且想延迟绑定，那么需要指定 **BINDFILE** 参数。

指定 **SQLERROR CONTINUE** 可创建程序包，即使在绑定 SQL 语句时发生错误也是如此。如果还指定了 **VALIDATE RUN**，那么可以在执行时以递增方式绑定那些由于授权或存在原因而未能绑定的语句。在运行时，尝试发出那些语句将生成错误。

消息文件

如果使用 **MESSAGES** 参数，那么预编译器会将消息重定向到指定的文件。这些消息包括用于描述预编译期间遇到的问题的警告和错误消息。如果未能成功地预编译源文件，请使用警告和错误消息来确定问题，更正源文件，然后再次尝试预编译源文件。如果没有使用 **MESSAGES** 参数，那么预编译消息将被写至标准输出。

对访问多个数据库服务器的嵌入式 SQL 应用程序进行预编译

要对访问多个服务器的应用程序进行预编译，可以执行下列其中一项任务:

- 将每个数据库的 SQL 语句分到不同的源文件中。请不要将不同数据库的 SQL 语句混合在同一个文件中。可以针对适当的数据库对每个源文件进行预编译。这是建议的方法。
- 在只使用动态 SQL 语句的情况下进行应用程序编码，并与程序将要访问的每个数据库绑定。
- 如果所有数据库看起来相同，即，它们的定义相同，那么可以将 SQL 语句分组到一个源文件中。

如果应用程序将通过 DB2 Connect 来访问主机应用程序服务器，那么这些过程也适用。请针对应用程序将要连接的服务器并使用可用于该服务器的 PREP 选项对该应用程序进行预编译。

嵌入式 SQL 应用程序的程序包和访问方案

预编译器将在数据库中生成一个程序包，另外，您还可以选择让它创建一个绑定文件。

程序包包含 DB2 优化器为应用程序中的静态 SQL 语句选择的访问方案。访问方案包含数据库管理器以优化器所确定的最高效方式发出静态 SQL 语句时所需的信息。对于动态 SQL 语句而言，优化器将在您运行应用程序时创建访问方案。

数据库中存储的程序包包含发出单一源文件中的特定 SQL 语句时所需的信息。对于预编译的每个用于构建应用程序的源文件，数据库应用程序都使用一个程序包。每个程序包都是独立的实体，而与同一个或其他应用程序所使用的任何其他程序包没有任何关系。要创建程序包，请在启用绑定的情况下对源文件运行预编译器，或者以后使用一个或多个绑定文件来运行绑定程序。

绑定文件包含创建程序包所需的 SQL 语句和其他数据。以后，您可以使用绑定文件来重新绑定应用程序，而不必先对该应用程序进行预编译。重新绑定操作将创建针对当前数据库情况进行优化的程序包。如果应用程序将要访问的数据库与您对其进行预编译时它访问的数据库不同，那么需要对该应用程序进行重新绑定。

使用 CURRENT PACKAGE PATH 专用寄存器进行程序包模式限定

程序包模式提供了以逻辑方式对程序包进行分组的方法。您可以通过不同的方法将程序包分组到模式中。某些实现对每个环境使用一个模式，例如生产模式和测试模式。其他实现对每个业务领域使用一个模式（例如 stocktrd 和 onlinebnk 模式）或者对每个应用程序使用一个模式（例如 stocktrdAddUser 和 onlinebnkAddUser）。您还可以对程序包进行分组以便执行一般的管理工作或者在程序包中提供变体（例如，维护应用程序的备份变体或者测试应用程序的新变体）。

将多个模式用于程序包时，数据库管理器必须确定要在哪个模式中查找程序包。为了完成此任务，数据库管理器将使用 CURRENT PACKAGESET 专用寄存器的值。您可以将此专用寄存器设置为单一模式名，以指示将要调用的任何程序包都属于该模式。如果应用程序使用不同模式中的程序包，那么可能必须在调用每个程序包之前发出 SET CURRENT PACKAGESET 语句（如果该程序包的模式与上一程序包的模式不同的话）。

注: 只有 DB2 版本 9.1 z/OS® 版（DB2 z/OS 版）才有 CURRENT PACKAGESET 专用寄存器，这允许您使用相应的 SET CURRENT PACKAGESET 语句显式设置值（单一模式名）。虽然 DB2 Database for Linux, UNIX, and Windows有 SET CURRENT PACKAGESET 语句，但没有 CURRENT PACKAGESET 专用寄存器。这意味着，对于 DB2 Database for Linux, UNIX, and Windows而言，无法在其他上下文（例如 SELECT 语句）中引用 CURRENT PACKAGESET。DB2 i 版未提供对 CURRENT PACKAGESET 的支持。

DB2 数据库服务器更为灵活，即，它能够在程序包解析期间考虑模式列表。模式列表类似于 CURRENT PATH 专用寄存器所提供的 SQL 路径。模式列表用于用户定义的函数、过程、方法和单值类型。

注: SQL 路径是 DB2 在尝试确定未限定的函数名、过程名、方法名或单值类型名的模式时应该考虑的模式名列表。

如果需要使程序包的多个变体（即，多组用于某个程序包的 BIND 选项）与一个已编译程序相关联，请考虑将用于 SQL 对象的模式的路径与用于程序包的模式的路径隔离。

CURRENT PACKAGE PATH 专用寄存器允许您指定程序包模式列表。其他 DB2 系列产品借助 CURRENT PATH 和 CURRENT PACKAGESET 之类的专用寄存器提供了类似的功能，这些专用寄存器将针对嵌套的过程和用户定义的函数进行入栈和出栈，而不会破坏调用应用程序的运行时环境。CURRENT PACKAGE PATH 专用寄存器为程序包模式解析提供了此功能。

许多安装将多个模式用于程序包。如果您未指定程序包模式列表，那么每次需要另一个模式中的程序包时，都必须发出 SET CURRENT PACKAGESET 语句（此语句最多可以包含一个模式名）。但是，如果在应用程序开头发出 SET CURRENT PACKAGE PATH 语句以指定模式名列表，那么不必在每次需要另一模式中的程序包时发出 SET CURRENT PACKAGESET 语句。

例如，假定存在下列程序包，并假定您想要调用以下程序包列表中第一个存在于服务器上的程序包：
SCHEMA1.PKG1、SCHEMA2.PKG2、SCHEMA3.PKG3、SCHEMA.PKG 和 SCHEMA5.PKG5。在采用 DB2 Database for Linux, UNIX, and Windows对 SET CURRENT PACKAGESET 语句的当前支持（即，接受单一模式名）的情况下，在尝试调用

每个程序包之前，必须发出 SET CURRENT PACKAGESET 语句以指定特定模式。对于此示例，需要发出 5 个 SET CURRENT PACKAGESET 语句。但是，通过使用 CURRENT PACKAGE PATH 专用寄存器，单一 SET 语句已足够。例如:

```
SET CURRENT PACKAGE PATH = SCHEMA1, SCHEMA2, SCHEMA3, SCHEMA, SCHEMA5;
```

注: 在 DB2 Database for Linux, UNIX, and Windows中，可以通过 db2cli.ini 文件、通过使用 SQLSetConnectAttr API、通过 SQLE-CLIENT-INFO 结构或者通过在嵌入式 SQL 程序中包括 SET CURRENT PACKAGE PATH 语句来设置 CURRENT PACKAGE PATH 专用寄存器。只有 DB2 z/OS 版的版本 8 或更高版本才支持 SET CURRENT PACKAGE PATH 语句。如果对 DB2 Database for Linux, UNIX, and Windows服务器或 DB2 i 版发出此语句，那么将返回 -30005。

可以使用多个模式来维护程序包的多个变体。这些变体对于帮助控制您在生产环境中所作的更改而言非常有用。另外，还可以使用程序包的不同变体来保留程序包的备份版本或者程序包的测试版本（例如，以便评估新索引的影响）。先前程序包版本的使用方式与备份应用程序（装入模块或可执行文件）相同，具体而言，用于提供恢复到先前版本的能力。

例如，假定 PROD 模式包含生产应用程序所使用的当前程序包，并且 BACKUP 模式用于存储那些程序包的备份副本。通过使用 PROD 模式进行绑定，将新版本的应用程序（以及程序包）提升到生产环境。通过使用备份模式（BACKUP）来绑定应用程序的当前版本，创建程序包的备份副本。然后，在运行时，您可以使用 SET CURRENT PACKAGE PATH 语句来指定在模式中查找程序包时应采用的顺序。假定已使用 BACKUP 模式来绑定应用程序 MYAPPL 的备份副本，并且生产环境中的当前应用程序版本已与 PROD 模式绑定（从而创建了程序包 PROD.MYAPPL）。要指定应该在 PROD 模式中的程序包变体可用时使用该变体（否则使用 BACKUP 模式中的变体），请对专用寄存器发出以下 SET 语句:

```
SET CURRENT PACKAGE PATH = PROD, BACKUP;
```

如果需要恢复为程序包的先前版本，那么可以使用 DROP PACKAGE 语句来删除该应用程序的生产版本，这将导致改为调用使用 BACKUP 模式绑定的旧应用程序版本（装入模块或可执行文件）。在这里，可以使用随操作系统平台不同而有所变化的应用程序路径技术。

注: 此示例假定程序包版本之间的差别仅仅是用于创建程序包的 BIND 选项有所不同（即，可执行文件代码没有差别）。

应用程序未使用 SET CURRENT PACKAGESET 语句来选择它所需的模式。而是，它允许 DB2 通过在 CURRENT PACKAGE PATH 专用寄存器中列示的模式中查找程序包来进行选择。

注: DB2 z/OS 版预编译过程在 DBRM 中存储一致性标记（可以使用 LEVEL 选项来设置此标记），在程序包解析期间，将进行检查以确保程序中的一致性标记与程序包匹配。同样，DB2 Database for Linux, UNIX, and Windows绑定过程在绑定文件中存储时间戳记。DB2 Database for Linux, UNIX, and Windows还支持 LEVEL 选项。

在不同模式中创建多个程序包版本的另一原因是，使不同的 BIND 选项生效。例如，可以将不同的限定符用于程序包中的未限定名称引用。

编写应用程序时，通常使用未限定的表名。这样就支持多个具有相同的表名和结构的表，但不同的限定符可以对不同的实例加以区分。例如，您可以在测试系统和生产系统中创建相同的对象，但这些对象的限定符有所不同（例如 PROD 和 TEST）。另一个示例是，应用程序将数据分布到不同 DB2 系统上的不同表中，并且每个表的限定符互不相同（例如 EAST、WEST、NORTH 和 SOUTH；COMPANYA 和 COMPANYB；Y1999、Y2000 和 Y2001）。对于 DB2 z/OS 版，请使用 BIND 命令的 QUALIFIER 选项来指定表限定符。使用 QUALIFIER 选项时，用户不必维护多个程序并让每个程序指定访问未限定的表所需的标准名称。而是，通过从应用程序中发出 SET CURRENT PACKAGESET 语句并指定单一模式名，可以在运行时访问正确的程序包。但是，如果使用 SET CURRENT PACKAGESET，那么仍需保留并修改多个应用程序：每个应用程序都通过自己的 SET CURRENT PACKAGESET 语句来访问所需的程序包。如果改为发出 SET CURRENT PACKAGE PATH 语句，那么可以列示所有模式。在执行时，DB2 可以选择正确的程序包。

注：DB2 Database for Linux, UNIX, and Windows还支持 QUALIFIER 绑定选项。但是，QUALIFIER 绑定选项只影响那些使用了 BIND 命令的 DYNAMICRULES 选项的静态 SQL 或程序包。

预编译器生成的时间戳记

在启用绑定的情况下预编译应用程序时，所生成的程序包和已修改源文件将具有匹配的时间戳记。这些时间戳记被单独地称为一致性标记。

如果存在多个版本的程序包（通过使用 **PRECOMPILE VERSION** 选项实现），那么每个版本都有相关联的时间戳记。应用程序运行时，将把程序包名、创建者和时间戳记发送到数据库管理器，数据库管理器将查找名称、创建者和时间戳记与该应用程序发送的信息匹配的程序包。如果不存在匹配的程序包，那么将下列两个 SQL 错误代码中的一个返回给应用程序:

- SQL0818N（时间戳记冲突）。如果找到名称和创建者匹配（但一致性标记不匹配）的单一程序包，并且该程序包的版本为“”（空字符串），那么将返回此错误。
- SQL0805N（找不到程序包）。在所有其他情况下，返回此错误。

请记住，将应用程序与数据库绑定时，除非使用 **PREP** 命令的 **PACKAGE USING** 参数覆盖缺省值，否则将把应用程序名的前 8 个字符用作程序包名。并且，除非 **PREP** 命令的 **VERSION** 参数指定了版本标识，那么版本标识将是“”（空字符串）。这意味着，如果在不更改版本标识的情况下使用同一名称来预编译和绑定两个程序，那么第二个程序包将替换第一个程序的程序包。运行第一个程序时，您将接收到时间戳记错误或“找不到程序包”错误，这是因为，经过修改的源文件的时间戳记不再与数据库中程序包的时间戳记匹配。“找不到程序包”错误也可能是由于使用了 **ACTION REPLACE REPLVER** 预编译或绑定选项所致，如以下示例所示:

1. 在指定 **VERSION** VER1 的情况下，预编译并绑定程序包 SCHEMA1.PKG。然后，生成相关联的应用程序 A1。
2. 在指定 **VERSION** VER2 **ACTION REPLACE REPLVER** VER1 的情况下，预编译并绑定程序包 SCHEMA1.PKG。然后，生成相关联的应用程序 A2。

   第二次预编译和绑定将生成 **VERSION** 为 VER2 的程序包 SCHEMA1.PKG，指定 **ACTION REPLACE REPLVER** VER1 将除去 **VERSION** 为 VER1 的程序包 SCHEMA1.PKG。

   尝试运行第一个应用程序将导致程序包不匹配，从而失败。

在以下示例中，将出现类似的症状:

1. 在指定 **VERSION** VER1 的情况下，预编译并绑定程序包 SCHEMA1.PKG。然后，生成相关联的应用程序 A1
2. 在指定 **VERSION** VER2 的情况下，预编译并绑定程序包 SCHEMA1.PKG。然后，生成相关联的应用程序 A2

   现在，可以同时运行应用程序 A1 和 A2，这两个应用程序将从程序包 SCHEMA1.PKG 的版本 VER1 和 VER2 中执行。例如，如果使用 SQL 语句 DROP PACKAGE SCHEMA1.PKG VERSION VER1 将第一个程序包删除，那么尝试运行应用程序 A1 将失败并发生“找不到程序包”错误。

如果预编译源文件但未创建程序包，那么生成的绑定文件和已修改源文件将具有匹配的时间戳记。为了运行该应用程序，将在一个单独的 **BIND** 步骤中绑定该绑定文件以创建程序包，并且将编译并链接经过修改的源文件。对于需要多个源模块的应用程序而言，必须为每个绑定文件执行绑定过程。

在此延迟绑定方案中，由于绑定文件包含的时间戳记与预编译期间在经过修改的源文件中存储的时间戳记相同，因此应用程序时间戳记与程序包时间戳记匹配。

预编译嵌入式 SQL 应用程序时生成的错误和警告

预编译时发生的嵌入式 SQL 错误由嵌入式 SQL 预编译器检测。嵌入式 SQL 预编译器能够检测语法错误，例如 SQL 语句缺少分号以及包含未声明的主变量。对于每个错误，都将生成相应的错误消息。

编译和链接包含嵌入式 SQL 的源文件

关于此任务

对嵌入式 SQL 源文件进行预编译时，PRECOMPILE 命令将生成经过修改的源文件，该文件具有适用于编程语言的文件扩展名。

请使用适当的主语言编译器来编译经过修改的源文件以及任何其他未包含 SQL 语句的源文件。语言编译器将每个经过修改的源文件转换为*对象模块*。

请参阅操作平台的编程文档，以了解有关缺省编译器选项的任何例外情况。请参阅编译器的文档，以获取可用编译器选项的完整描述。

主语言链接程序将创建可执行的应用程序。例如:

- 在 Windows 操作系统上，应用程序可以是可执行文件或动态链接库（DLL）。
- 在基于 UNIX 和 Linux 的操作系统上，应用程序可以是可执行的装入模块或共享库。

注: 在 Windows 操作系统上，虽然应用程序可以是 DDL，但 DLL 由应用程序而不是 DB2 数据库管理器直接装入。在 Windows 操作系统上，数据库管理器将嵌入式 SQL 存储过程和用户定义的函数作为 DLL 装入。

要创建可执行文件，请链接以下对象:

- 由语言编译器根据经过修改的源文件和其他未包含 SQL 语句的文件生成的用户对象模块。
- 语言编译器附带提供的主语言库 API。

- 包含操作环境的相应数据库管理器 API 的数据库管理器库。请参阅操作平台的相应编程文档，以了解数据库管理器 API 所需的数据库管理器库的具体名称。

将嵌入式 SQL 程序包与数据库绑定

“绑定”是指根据绑定文件创建程序包并将其存储到数据库的过程。

应用程序、绑定文件和程序包之间的关系

数据库应用程序使用程序包的某些原因与编译应用程序的原因相同：提高性能以及进行压缩。通过对 SQL 语句进行预编译，该语句在应用程序构建期间（而不是在运行时）被编译到程序包中。每个语句都被解析，程序包中存储的是解释效率更高的操作数字符串。在运行时，预编译器生成的代码将调用运行时服务数据库管理器 API 并提供输入或输出数据所需的任何变量信息，程序包中存储的信息将被执行。

预编译的优点仅适用于静态 SQL 语句。不会对使用 PREPARE 和 EXECUTE 或者 EXECUTE IMMEDIATE 以动态方式执行的 SQL 语句进行预编译；因此，在运行时，必须对它们执行整套处理步骤。

借助 DB2 绑定文件描述（**db2bfd**）实用程序，可以方便地显示绑定文件的内容，以便检查和验证其中的 SQL 语句。另外，还可以使用 DB2 绑定文件描述（**db2bfd**）实用程序来显示用于创建绑定文件的预编译选项。这对于确定与应用程序的绑定文件相关的问题而言非常有用。

可将 **BIND** 命令的 **GENERIC** 参数上的 STATICASDYNAMIC 字符串设置为“yes”，以指示 DB2 数据库管理器将所有语句存储在目录中并将它们标记为增量绑定。在运行时，当首次装入程序包时，数据库管理器使用当前会话环境（而不是程序包）来设置节条目和其他实体（将填充文本并访问程序包高速缓存）。此后，绑定文件中的语句的行为就如同您正在使用动态 SQL 一样。例如，各节将针对数据库定义语言失效、特殊专用寄存器更新等隐式地重新编译。DB2 数据库管理器提供此功能是为了方便从其他数据库系统迁移嵌入的 SQL C 应用程序。

DYNAMICRULES 绑定选项对动态 SQL 的影响

PRECOMPILE 命令和 **BIND** 命令参数 **DYNAMICRULES** 可确定在运行时对动态 SQL 应用的规则。

尤其是，**DYNAMICRULES** 参数可确定在运行时对以下动态 SQL 属性应用的值：

- 授权检查期间使用的授权标识。
- 用于对未限定的对象进行限定的限定符。
- 程序包是否可用于以动态方式准备下列语句：
  GRANT、REVOKE、ALTER、CREATE、DROP、COMMENT ON、RENAME、SET INTEGRITY 和 SET EVENT MONITOR STATE 语句。

除 **DYNAMICRULES** 值以外，程序包的运行时环境控制动态 SQL 语句在运行时的行为。两种可能的运行时环境是：

- 程序包作为独立程序的组成部分运行
- 程序包在例程上下文中运行

DYNAMICRULES 值与运行时环境共同确定动态 SQL 属性的值。该组属性值被称为动态 SQL 语句行为。四种行为如下所示:

运行行为

DB2 Database for Linux, UNIX, and Windows 使用执行程序包的用户的授权标识（最初连接到 DB2 数据库的标识）作为用于对动态 SQL 语句执行授权检查的值，并将该标识用作对动态 SQL 语句中的未限定对象引用进行隐式限定的初始值。

绑定行为

在运行时，DB2 Database for Linux, UNIX, and Windows 将使用所有适用于静态 SQL 的规则来进行授权和限定。即，使用程序包所有者的授权标识作为用于对动态 SQL 语句执行授权检查的值，并将该标识用作对动态 SQL 语句中的未限定对象引用进行隐式限定的程序包缺省限定符。

定义行为

仅当动态 SQL 语句包含在要在例程上下文中运行的程序包中，并且您已使用 **DYNAMICRULES** DEFINEBIND 或 **DYNAMICRULES** DEFINERUN 对该程序包进行绑定时，定义行为才适用。DB2 Database for Linux, UNIX, and Windows 使用例程定义者（而不是例程的程序包绑定者）的授权标识作为用于对动态 SQL 语句执行授权检查的值，并使用该标识对该例程中的动态 SQL 语句中的未限定对象引用进行隐式限定。

调用行为

仅当动态 SQL 语句包含在要在例程上下文中运行的程序包中，并且您已使用 **DYNAMICRULES** INVOKEBIND 或 **DYNAMICRULES** INVOKERUN 对该程序包进行绑定时，调用行为才适用。DB2 Database for Linux, UNIX, and Windows 使用调用例程时生效中的当前语句授权标识作为用于对动态 SQL 执行授权检查的值，并使用该标识对该例程中的动态 SQL 语句中的未限定对象引用进行隐式限定。下表对此作了摘要:

调用环境	使用的标识
任何静态 SQL	SQL 调用该例程的 **OWNER** 所在程序包的隐式或显式所有者值。
在视图或触发器的定义中使用	该视图或触发器的定义者。
运行行为程序包中的动态 SQL	用于对 DB2 数据库建立初始连接的标识。
定义行为程序包中的动态 SQL	使用了调用该例程的 SQL 所在程序包的例程的定义者。
调用行为程序包中的动态 SQL	调用该例程的当前授权标识。

下表显示实现每种动态 SQL 行为的 **DYNAMICRULES** 值与运行时环境组合。

表 18. DYNAMICRULES 和运行时环境如何确定动态 SQL 语句行为

DYNAMICRULES 值	独立程序环境中动态 SQL 语句的行为	例程环境中动态 SQL 语句的行为
BIND	绑定行为	绑定行为
RUN	运行行为	运行行为
DEFINEBIND	绑定行为	定义行为
DEFINERUN	运行行为	定义行为
INVOKEBIND	绑定行为	调用行为

表 18. DYNAMICRULES 和运行时环境如何确定动态 SQL 语句行为 (续)

DYNAMICRULES 值	独立程序环境中动态 SQL 语句的行为	例程环境中动态 SQL 语句的行为
INVOKERUN	运行行为	调用行为

下表列示每种动态 SQL 行为的动态 SQL 属性值。

表 19. 动态 SQL 语句行为的定义

动态 SQL 属性	动态 SQL 属性的设置: 绑定行为	动态 SQL 属性的设置: 运行行为	动态 SQL 属性的设置: 定义行为	动态 SQL 属性的设置: 调用行为
授权标识	**BIND OWNER** 命令参数的隐式或显式值	执行程序包的用户的标识	例程定义者（而不是例程的程序包所有者）	调用例程时的当前语句授权标识。
未限定的对象的缺省限定符	**BIND QUALIFIER** 命令参数的隐式或显式值	CURRENT SCHEMA 专用寄存器	例程定义者（而不是例程的程序包所有者）	调用例程时的当前语句授权标识。
可以执行 GRANT、REVOKE、ON、RENAME、SET INTEGRITY 和 SET EVENT MONITOR STATE	否 ALTER、CREATE、DROP、COMMENT	是	否	否

使用专用寄存器来控制语句编译环境

对于以动态方式准备的语句而言，语句编译环境由多个专用寄存器的值确定:

- CURRENT QUERY OPTIMIZATION 专用寄存器确定所使用的优化类。
- CURRENT PATH 专用寄存器确定用于进行 UDF 和 UDT 解析的函数路径。
- CURRENT EXPLAIN SNAPSHOT 寄存器确定是否捕获说明快照信息。
- CURRENT EXPLAIN MODE 寄存器确定是否为任何合格的动态 SQL 语句捕获说明表信息。这些专用寄存器的缺省值与用于相关绑定选项的缺省值相同。

使用 BIND 命令和现有绑定文件来重新创建程序包

“绑定”是指创建数据库管理器在应用程序执行时访问数据库所需的程序包的过程。缺省情况下，**PRECOMPILE** 命令将创建一个程序包。除非指定 **BINDFILE** 命令参数，否则绑定将在预编译时以隐式方式进行。**PACKAGE** 命令参数允许您为预编译时创建的程序包指定程序包名。

下面是一个典型的 **BIND** 命令用法示例。要将名为 filename.bnd 的绑定文件与数据库绑定，可以发出以下命令:

```
BIND filename.bnd
```

对于每个单独进行预编译的源代码模块，都将创建一个程序包。如果应用程序有 5 个源文件，其中 3 个需要进行预编译，那么将创建 3 个程序包或绑定文件。缺省情况下，为每个程序包指定的名称与 .bnd 文件所基于的源模块的名称相同，但此名称将被截断为 8 个字符。要以显式方式指定另一个程序包名，必须在 **PREP** 命令中使用 **PACKAGE USING** 参数。程序包的版本由 **VERSION** 预编译参数给定，并且缺省为空字符串。如果这个新创建的程序包的名称和模式与目标数据库中当前存在的某个程序包相同，但版本标识有所不同，那么将创建新程序包并保留先前的程序包。但是，如果存在与所绑定

程序包的名称、模式和版本匹配的程序包，那么该程序包将被删除并替换为正在绑定的新程序包。如果绑定时指定了 **ACTION ADD**，那么将不允许发生这种情况，而是返回错误（SQL0719）。

使用 REBIND 命令来重新绑定现有程序包

*重新绑定*是指为先前绑定的应用程序重新创建程序包的过程。如果程序包已被标记为无效或不可用，或者上次绑定后数据库统计信息已更改，那么必须重新绑定程序包。

但是，在某些情况下，您可能想重新绑定有效的程序包。例如，您可能想利用新创建的索引，或者在执行 **RUNSTATS** 命令后使用经过更新的统计信息。

程序包可以依赖于某些类型的数据库对象，例如表、视图、别名、索引、触发器、引用约束和表检查约束。如果程序包依赖于数据库对象（例如表、视图和触发器等等），并且该对象被删除，那么该程序包将被置于“无效”状态。如果被删除的对象是 UDF，那么该程序包将被置于不可用状态。

程序包被标记为不可用时，考虑到数据库模式中导致该程序包变为不可用的最新更改，下一次在此程序包中使用语句将导致使用非传统绑定语义隐式重新绑定程序包，以便能够解析为 SQL 对象。

对于程序包中的静态 DML，这些程序包可隐式重新绑定，或者通过发出 **REBIND** 命令（或对应 API）或 **BIND** 命令（或对应 API）显式重新绑定。隐式重新绑定是通过传统绑定语义执行的（如果该程序包被标记为无效），但它会在程序包标记为不可用时使用非传统绑定语义。

对于被修改为包含更多、更少或已更改的 SQL 语句的程序而言，您必须使用 **BIND** 命令来重新绑定程序包。如果需要将任何绑定选项的值更改为与最初绑定该程序包时使用的值不同，那么也必须使用 **BIND** 命令。**REBIND** 命令提供使用传统绑定语义解析的选项 (**RESOLVE CONSERVATIVE**) 或通过考虑新例程、数据类型或全局变量解析的选项（**RESOLVE ANY**，这是缺省选项）。仅当数据库管理器未将程序包标记为不可用时，才能使用 **RESOLVE CONSERVATIVE** 选项 (SQLSTATE 51028)。环境未明确要求使用 **BIND** 时，您应该使用 **REBIND**，这是因为，**REBIND** 的性能显著优于 **BIND**。

当目录中存在多个具有相同程序包名的版本时，每次只能对一个版本进行重新绑定。

在 IBM Data Studio V3.1 或更高版本中，可以使用以下工具的任务助手：重新绑定程序包. 任务助手可以指导您执行以下过程：设置选项、查看自动生成的命令以执行任务以及运行这些命令。有关更多详细信息，请参阅使用任务助手管理数据库。

绑定注意事项

如果应用程序代码页与数据库代码页不同，那么在进行绑定时，您可能必须考虑所要使用的代码页。

如果应用程序对任何 数据库管理器 实用程序 API（例如 **IMPORT**或 **EXPORT**）发出调用，那么您必须将提供的实用程序绑定文件与数据库绑定。

可以使用绑定选项来控制绑定期间执行的某些操作，如下列示例所示：

- **QUERYOPT** 绑定参数在绑定时利用特定的优化类。
- **EXPLSNAP** 绑定参数在说明表中存储合格 SQL 语句的说明快照信息。

- **FUNCPATH** 绑定参数正确地解析静态 SQL 中用户定义的单值类型和用户定义的函数。

如果绑定过程启动但一直不返回，那么可能表明其他连接到数据库的应用程序已挂起您所需的锁定。在此情况下，请确保没有任何应用程序连接到数据库。如果有任何应用程序连接到数据库，请断开服务器上所有应用程序的连接，绑定过程将继续进行。

如果应用程序将使用 DB2 Connect 来访问服务器，那么您可以使用可用于该服务器的**BIND**命令参数。

绑定文件并不与较低版本的 DB2 Database for Linux, UNIX, and Windows 兼容。在混合级别环境中，DB2 Database for Linux, UNIX, and Windows 只能使用可用于最低级别数据库环境的功能。例如，如果版本 8 客户机连接到版本 7.2 服务器，那么该客户机将只能使用版本 7.2 的功能。当绑定文件表达数据库的功能时，它们需遵守混合级别限制。

如果需要在较低级别的系统上重新绑定较高级别的绑定文件，那么您可以执行下列操作:

- 使用较低级别 IBM 数据服务器客户机 连接到较高级别服务器，然后创建可以被交付并与较低级别 DB2 Database for Linux, UNIX, and Windows 环境绑定的绑定文件。
- 在较低级别的生产环境中，使用较高级别 IBM 数据服务器客户机 来绑定在测试环境中创建的较高级别绑定文件。较高级别客户机将只传递适用于较低级别服务器的选项。

分块注意事项

如果要对嵌入式 SQL 应用程序关闭分块功能，但无法获得源代码，那么必须通过使用 **BIND** 命令并设置 **BLOCKING NO** 子句来重新绑定该应用程序。

必须通过使用 **BIND** 命令并设置 **BLOCKING ALL** 或 **BLOCKING UNAMBIGUOUS** 子句以请求进行分块来重新绑定现有的嵌入式 SQL 应用程序（如果尚未以此方式进行绑定）。从服务器检索行块之后，嵌入式应用程序将以每次一行的方式从服务器检索 LOB 值。

延迟绑定的优点

在启用绑定的情况下进行预编译只允许应用程序访问预编译过程中使用的数据库。但是，在延迟绑定的情况下进行预编译将允许应用程序访问许多数据库，这是因为，您可以针对每个数据库对绑定文件进行绑定。这种应用程序开发方法在本质上更为灵活，即，您只需对应用程序进行一次预编译，但可以随时将其与数据库绑定。

在执行期间使用 BIND API 允许应用程序绑定其自身，此操作可以在安装过程中进行，也可以在执行相关联的模块之前进行。例如，某个应用程序可以执行多项任务，并且只有其中一项任务要求使用 SQL 语句。您可以将此应用程序设计成，仅当此应用程序调用需要 SQL 语句的任务时，并且仅当相关联的程序包尚未存在时，才将其自身与数据库绑定，

延迟绑定方法的另一个优点是，它允许您在不必向最终用户提供源代码的情况下创建程序包。您可以随应用程序一起交付相关联的绑定文件。

使用 BIND 命令的 REOPT 选项时的性能改进

绑定选项 **REOPT** 能够显著提高嵌入式 SQL 应用程序的性能。

REOPT 对静态 SQL 的影响

绑定选项 **REOPT** 能够使包含主变量、全局变量或专用寄存器的静态 SQL 语句像以递增方式绑定的语句一样工作。这意味着，在执行 EXECUTE 或 OPEN 时（而不是在绑定时）编译这些语句。在此编译期间，将根据这些变量的实际值来选择访问方案。

如果指定 **REOPT ONCE**，那么在第一个 OPEN 或 EXECUTE 请求之后将对存取方案进行高速缓存，该存取方案将用于此语句的后续执行。如果指定 **REOPT ALWAYS**，那么每个 OPEN 和 EXECUTE 请求都将重新生成存取方案，并且将使用主变量、参数标记、全局变量和专用寄存器值的当前集合来创建此方案。

REOPT 对动态 SQL 的影响

如果指定选项 **REOPT ALWAYS**，那么数据库管理器将任何包含主变量、参数标记、全局变量或专用寄存器的语句的准备工作推迟到它遇到 OPEN 或 EXECUTE 语句时进行；即，在这些变量的值变为已知时进行。此时，将使用这些值来生成访问方案。对同一语句发出的后续 OPEN 或 EXECUTE 请求将重新编译该语句、使用变量值的当前集合来重新优化查询方案以及执行新生成的查询方案。指定 **REOPT ALWAYS** 时，语句集中器处于禁用状态。

选项 **REOPT ONCE** 具有类似的效果，但不同之处在于，只使用主变量、参数标记、全局变量和专用寄存器的值对方案进行一次优化。此方案将进行高速缓存并由后续请求使用。

绑定应用程序和实用程序（DB2 Connect 服务器）

必须将使用嵌入式 SQL 开发的应用程序与它们将要处理的数据库进行绑定。有关 IBM 数据服务器程序包绑定要求的信息，请参阅有关 DB2 CLI 绑定文件和程序包名称的主题。

每个应用程序应该对每个数据库执行一次绑定。在绑定过程中，为将要执行的每个 SQL 语句都存储了数据库存取方案。这些访问方案是由应用程序开发者提供的，它们包含在*绑定文件*中，该文件是在预编译期间创建。绑定是 IBM 大型机数据库服务器处理这些绑定文件的过程。

因为随 DB2 Connect 提供的一些实用程序是使用嵌入式 SQL 开发的，所以这些实用程序必须先绑定到 IBM 大型机数据库服务器，才能与该系统配合使用。如果您不使用 DB2 Connect 实用程序和接口，那么不需要将它们绑定到每个 IBM 大型机数据库服务器。在下列文件中包含了这些实用程序所需要的绑定文件的列表:

- `ddcsmvs.lst`（用于 System z）
- `ddcsvse.lst`（用于 VSE）
- `ddcsvm.lst`（用于 VM）
- `ddcs400.lst`（用于 IBM Power Systems™）

当将这些文件列表的其中一个与数据库进行绑定时，将导致每个单个实用程序都与该数据库进行绑定。

如果安装了 DB2 Connect 服务器产品，DB2 Connect 实用程序必须绑定到每个 IBM 大型机数据库服务器，才可与该系统配合使用。假定客户机也处于同一修订包级别，不管涉及多少客户机平台，您只需绑定一次实用程序。

例如，如果让 10 台 Windows 客户机和 10 台 AIX 客户机通过 Windows 服务器上的 DB2 Connect Enterprise Edition 连接至 DB2 z/OS 版，请执行下列其中一个步骤:

- 从其中一个 Windows 客户机绑定 `ddcsmvs.lst`。
- 从其中一个 AIX 客户机绑定 `ddcsmvs.lst`。
- 从 DB2 Connect 服务器绑定 `ddcsmvs.lst`。

本示例假定:

- 所有客户机都处于同一服务级别。如果它们不处于同一级别，那么可能需要从每个特定服务级别的客户机来进行绑定。
- 服务器与客户机处于同一服务级别。如果它们不处于同一级别，那么也需要从服务器进行绑定。

除了 DB2 Connect 实用程序之外，其他任何使用嵌入式 SQL 的应用程序还必须与它们想使用的每个数据库进行绑定。当执行未绑定的应用程序时，通常都将产生错误消息 SQL0805N。您可能想为需要绑定的所有应用程序创建一个附加的绑定列表文件。

对于每个要绑定至的 IBM 大型机数据库服务器，请执行下列步骤:

1. 确保您对 IBM 大型机数据库服务器管理系统具有足够的权限:

   System z

   需要的权限是:

   - SYSADM 或
   - SYSCTRL 或
   - BINDADD *和* CREATE IN COLLECTION NULLID

   注: 仅当软件包尚未存在时，BINDADD 和 CREATE IN COLLECTION NULLID 特权才能提供足够的权限。例如，您正在首次创建软件包的时候。

   如果软件包已经存在并且您正在再次绑定它们，那么完成该任务所需要的权限取决于是谁执行了最初的绑定。

   A) 如果最初是您自己执行了绑定，而您正在再次执行绑定，那么只要您具有先前所列示的任何权限，就将允许您完成绑定。

   B) 如果其他人执行了最初的绑定，而您正在执行第二次绑定，那么您将必须具有 SYSADM 或 SYSCTRL 权限才能完成绑定。如果您只具有 BINDADD 和 CREATE IN COLLECTION NULLID 权限，那么将不允许您完成绑定。如果您没有 SYSADM 或 SYSCTRL 特权，仍然可以创建软件包。在此情况下，您将需要对想替换的每个现有软件包都具有 BIND 特权。

   VSE 或 VM

   需要的权限是 DBA 权限。如果想在绑定（bind）命令中使用 GRANT 选项（以避免为每个 DB2 Connect 软件包单独授予访问权），那么 NULLID 用户标识必须有权为其他用户授予对下列表的权限:

   - system.syscatalog
   - system.syscolumns
   - system.sysindexes
   - system.systabauth
   - system.syskeycols

- system.syssynonyms
- system.syskeys
- system.syscolauth
- system.sysuserauth

在 VSE 或 VM 系统上，您可以发出:

```
grant select on table to nullid with grant option
```

IBM Power Systems

对 NULLID 集合的 *CHANGE 权限或更高权限。

2. 请发出类似以下命令的命令:

```
db2 connect to DBALIAS user USERID using PASSWORD
   db2 bind path@ddcsmvs.lst blocking all
              sqlerror continue messages ddcsmvs.msg grant public
   db2 connect reset
```

其中 *DBALIAS*、*USERID* 和 *PASSWORD* 适用于 IBM 大型机数据库服务器，ddcsmvs.lst 是 z/OS 的绑定列表文件，而 *path* 表示该绑定列表文件的位置。

例如，*drive*:\sqllib\bnd\ 适用于所有 Windows 操作系统，*INSTHOME*/sqllib/bnd/ 适用于所有 Linux 和 UNIX 操作系统，其中 *drive* 表示安装 DB2 Connect 的逻辑驱动器，*INSTHOME* 表示 DB2 Connect 实例的主目录。

可以使用 **bind** 命令的 grant 选项来对 PUBLIC 或指定的用户名或组标识授予 EXECUTE 特权。如果不使用 **bind** 命令的 grant 选项，那么必须逐个执行 GRANT EXECUTE (RUN)。

要了解绑定文件的软件包名称，输入下列命令:

```
ddcspkgn @bindfile.lst
```

例如:

```
ddcspkgn @ddcsmvs.lst
```

可能产生下列输出:

```
绑定文件                        软件包名称
------------------------------ ------------------------------
f:\sqllib\bnd\db2ajgrt.bnd     SQLAB6D3
```

要确定 DB2 Connect 的这些值，执行 **ddcspkgn** 实用程序，例如:

```
ddcspkgn @ddcsmvs.lst
```

另外，此实用程序可用来确定各个绑定文件的软件包名称，例如:

```
ddcspkgn bindfile.bnd
```

注:

a. 需要使用绑定选项 **sqlerror continue**；然而，当使用 DB2 工具或命令行处理器（CLP）来绑定应用程序时将自动为您指定此选项。指定了此选项，将把绑定错误转换为警告，因此，绑定一个包含错误的文件时仍然可以创建软件包。同样，这允许对多个服务器使用一个绑定文件，即使特定的服务器实现可能将另一个实现的 SQL 语法标志为无效。因此，对任何特定的 IBM 大型机数据库服务器绑定任何列表文件 ddcsxxx.lst 都应生成一些警告。

b. 如果您正在通过 DB2 Connect 连接至 DB2数据库，那么使用绑定列表 db2ubind.lst 并且不指定 **sqlerror continue**（它仅在连接至 IBM 大型机数据库服务器时才有效）。另外，要连接至 DB2数据库，建议您使用 DB2（而不是 DB2 Connect）提供的 DB2客户机。

3. 使用类似的语句来绑定每个应用程序或应用程序列表。
4. 如果您具有 DB2 前发行版的远程客户机，那么可能需要将这些客户机上的实用程序与 DB2 Connect 进行绑定。

程序包的存储与维护

您通过对应用程序进行预编译/绑定来创建程序包。程序包包含经过优化的访问方案，此方案用于监视应用程序中所有 SQL 语句的执行情况。三种用于处理程序包的特权是 CONTROL、EXECUTE 和 BIND 特权，它们用于对可接受的访问权级别进行过滤。通过在编译时指定 VERSION 选项，可以创建同一个程序包的多个版本。此选项帮助避免时间戳记不匹配错误，并允许同时运行应用程序的多个版本。

程序包版本控制

如果需要创建应用程序的多个版本，那么可以使用 **PRECOMPILE** 命令中的 **VERSION** 参数。此选项允许多个具有相同程序包名（即，程序包名和创建者名）的版本共存。

例如，假定有一个从 foo1.sqc 编译的称为 foo1 的应用程序。您将预编译程序包 foo1 并将其与数据库绑定，然后将该应用程序交付给用户。然后，用户可以运行该应用程序。要对该应用程序进行后续更改，您将更新 foo1.sqc，然后重复重新编译、绑定以及将该应用程序发送给用户这一过程。如果在第一次或第二次预编译 foo1.sqc 时没有指定 **VERSION** 参数，那么第一个程序包将被第二个程序包替换。任何尝试运行旧版本应用程序的用户都将接收到 SQLCODE -818，这表示发生时间戳记不匹配错误。

为了避免发生时间戳记不匹配错误，并且为了允许这两个版本的应用程序同时运行，请使用程序包版本控制功能。例如，在构建 foo1 的第一个版本时，请使用 **VERSION** 参数对其进行预编译，如下所示:

```
DB2 PREP FOO1.SQC VERSION V1.1
```

现在，可以运行此程序的第一个版本。在构建 foo1 的新版本时，请使用以下命令对其进行预编译:

```
DB2 PREP FOO1.SQC VERSION V1.2
```

现在，这个新版本的应用程序也能够运行，即使第一个应用程序的实例仍在执行亦如此。由于第一个程序包的程序包版本是 V1.1，并且第二个程序包的程序包版本是 V1.2，因此不会发生命名冲突：两个程序包都将存在于数据库中，并且这两个版本的应用程序都可用。

可以将 **PRECOMPILE** 或 **BIND** 命令的 **ACTION** 参数与 **PRECOMPILE** 命令的 **VERSION** 参数一起使用。可以使用 **ACTION** 参数控制添加或替换不同版本程序包的方式。

程序包特权不具有版本级的粒度。即，程序包特权的授予（GRANT）或撤销（REVOKE）将应用于所有具有相同名称和创建者的程序包版本。因此，在创建版本 V1.1 之后，如果将程序包 foo1 的程序包特权授予某个用户或组，那么分发版本 V1.2 时，该用户或组将对版本 V1.2 具有相同的特权。这种行为通常是必需的，这是因为，相同的用户和组通常对一个程序包的所有版本拥有相同的特权。如果您不想将相同的

程序包特权应用于应用程序的所有版本，那么不应使用 **PRECOMPILE VERSION** 参数来完成程序包版本控制。而是，应该使用不同的程序包名（将经过更新的源文件重命名，或者使用 **PACKAGE USING** 参数以显式方式将程序包重命名）。

对未限定的表名进行解析

可以使用下列其中一种方法来处理应用程序中未限定的表名:

- 借助于下列命令，通过使用不同的授权标识，每个用户都可以将其程序包与不同的 **COLLECTION** 参数绑定:

  ```
  CONNECT TO database-name USER 用户名
  BIND 文件名 COLLECTION 模式名
  ```

  在此示例中，*database-name* 是数据库的名称，*用户名* 是用户的名称，*文件名* 是将要绑定的应用程序的名称。注意，*用户名* 与 *模式名* 通常具有相同的值。然后，使用 SET CURRENT PACKAGESET 语句来指定要使用的程序包，从而指定要使用的限定符。如果未指定 **COLLECTION**，那么缺省限定符是绑定程序包时使用的授权标识。如果指定 **COLLECTION**，那么指定的 *schema_name* 是将要用于未限定对象的限定符。
- 创建公用别名以指向需要的表。
- 为每个用户创建与表同名的视图，以便正确地解析未限定的表名。
- 为每个用户创建一个别名以指向需要的表。

使用样本构建脚本来构建嵌入式 SQL 应用程序

在 UNIX 和 Linux 上，用于演示构建样本程序的文件被称为脚本文件，在 Windows 上，它们被称为批处理文件。通常，我们将其称为构建文件。它们包含用于受支持的平台编译器的建议编译和链接命令。

构建文件由 DB2 提供，用于与受支持平台相关的主语言。构建文件包含在该语言的样本所在的目录中。下表列示用于构建不同类型的程序的不同构建文件类型。除非另有指示，否则这些构建文件用于所有受支持平台上的受支持语言。在 Windows 上，构建文件的扩展名为 .bat（批处理），下表未列示此扩展名。在 UNIX 平台上，没有扩展名。

表 20. DB2 构建文件

构建文件	所构建的程序的类型
bldapp	应用程序
bldrtn	例程（存储过程和 UDF）
bldmc	C/C++ 多连接应用程序
bldmt	C/C++ 多线程应用程序
bldcli	sqlpl samples 子目录中 SQL 过程的 CLI 客户机应用程序。

注: 缺省情况下，用于根据源代码来构建可执行文件的 bldapp 样本脚本将构建 64 位可执行文件。

下表按平台和编程语言来列示构建文件以及它们所在的目录。在联机文档中，构建文件名以热链接方式链接到 HTML 源文件。用户还可以在相应的 samples 目录中访问文本文件。

表 21. 构建文件（按语言和平台排列）

平台 —> 语言	AIX	HP-UX	Linux	Solaris	Windows
C samples/c	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp.bat bldrtn.bat bldmt.bat bldmc.bat
C++ samples/cpp	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp bldrtn bldmt bldmc	bldapp.bat bldrtn.bat bldmt.bat bldmc.bat
IBM COBOL samples/cobol	bldapp bldrtn	不适用	不适用	不适用	bldapp.bat bldrtn.bat
Micro Focus COBOL samples/cobol_mf	bldapp bldrtn	bldapp bldrtn	bldapp bldrtn	bldapp bldrtn	bldapp.bat bldrtn.bat

在文档中，使用构建文件来构建应用程序和例程，这是因为，它们非常清晰地演示了 DB2 建议受支持编译器使用的编译和链接选项。通常，还有其他许多可用的编译和链接选项，并且用户可以自由地试验那些选项。请参阅编译器文档，以了解所提供的所有编译和链接选项。除构建样本程序以外，开发者还可以使用构建文件来构建他们自己的程序。样本程序可以用作可由用户修改的模板，以便帮助他们开发应用程序。

通过使用构建文件，您可以方便地构建具有编译器所允许的任何文件名的源文件。这与在文件中硬编码程序名的 Makefile 不同。Makefile 通过访问构建文件来编译和链接它们所构造的程序。构建文件使用 $1 变量（对于 UNIX 和 Linux）和 %1 变量（对于 Windows 操作系统）以内部方式替换程序名。通过使这些变量名的数目不断递增，可以替换其他可能需要的自变量。

由于每个构建文件都适合于构建特定类型的程序，例如独立应用程序、例程（存储过程和 UDF）或者更专门的程序类型（例如多连接程序和多线程程序），因此，构建文件使您能够快捷方便地进行试验。对于编译器所支持的每种特定程序类型，都提供了相应的构建文件类型。

每次构建程序时，都将自动覆盖构建文件所生成的对象文件和可执行文件，即使未曾修改源文件亦如此。使用 Makefile 时，情况并非如此。这意味着，开发者可以重建现有程序，而不必删除先前的对象文件和可执行文件或者修改源代码。

构建文件包含样本数据库的缺省设置。如果用户正在访问另一个数据库，那么他们只需提供另一个参数即可覆盖缺省值。如果他们始终如一地使用该数据库，那么可以硬编码此数据库名，从而替换构建文件本身中的 sample。

对于嵌入式 SQL 程序而言，除了在 Windows 上使用 IBM COBOL 预编译器的情况以外，构建文件将调用另一个文件 embprep，该文件包含用于嵌入式 SQL 程序的预编译和绑定步骤。这些步骤可能需要有关用户标识和密码的可选参数，这取决于嵌入式 SQL 程序的构建位置。

最后，开发者可以为了工作方便而修改构建文件。除了在构建文件中更改数据库名（以上描述的方法）以外，开发者还可以方便地在此文件中硬编码其他参数、更改编译和链接选项或者更改缺省的 DB2 实例路径。构建文件的简单、直接和具体性质使您能够方便地对其进行定制以满足您的需要。

错误检查实用程序

DB2 客户机提供了多个实用程序文件。这些文件提供了用于执行错误检查和打印错误信息的功能。对于每一种语言，都在 samples 目录中提供了相应的实用程序文件。将错误检查实用程序文件与应用程序配合使用时，它们将提供对您非常有帮助的错误信息，并可以简化 DB2 程序的调试工作。大多数错误检查实用程序使用 DB2 API GET SQLSTATE MESSAGE (SQLogstt) 和 GETERROR MESSAGE (sqlaintp) 来获取与执行程序时遇到的问题相关的 SQLSTATE 和 SQLCA 信息。CLI 实用程序文件 utilcli.c 不会使用这些 DB2 API; 而是会使用等价的CLI 语句。对于所有错误检查实用程序，都将打印描述性错误消息以使开发者能够迅速了解问题。某些 DB2 程序，例如例程（存储过程和用户定义的函数）不需要使用这些实用程序。

用于不同编程语言并受 DB2 支持的编译器所使用的错误检查实用程序文件如下所示:

表 22. 错误检查实用程序文件（按语言排列）

语言	非嵌入式 SQL 源文件	非嵌入式 SQL 头文件	嵌入式 SQL 源文件	嵌入式 SQL 头文件
C samples/c	utilapi.c	utilapi.h	utilemb.sqc	utilemb.h
C++ samples/cpp	utilapi.C	utilapi.h	utilemb.sqC	utilemb.h
IBM COBOL samples/cobol	checkerr.cbl	不适用	不适用	不适用
Micro Focus COBOL samples/cobol_mf	checkerr.cbl	不适用	不适用	不适用

为了使用实用程序函数，必须先编译实用程序文件，然后在创建目标系统的可执行文件期间链接其对象文件。samples 目录中的 Makefile 和构建文件都可以为需要错误检查实用程序的程序完成此工作。

以下示例演示如何在 DB2 程序中使用错误检查实用程序。utilemb.h 头文件为函数 SqlInfoPrint() 和 TransRollback() 定义了 EMB_SQL_CHECK 宏:

```
/* 用于执行嵌入式 SQL 检查的宏 */
#define EMB_SQL_CHECK(MSG_STR)                          \
SqlInfoPrint(MSG_STR, &sqlca, __LINE__, __FILE__);      \
if (sqlca.sqlcode < 0)                                  \
{                                                       \
  TransRollback();                                      \
  return 1;                                             \
}
```

SqlInfoPrint() 检查 SQLCODE 并打印任何与所遇到的特定错误相关的信息。它还指向源代码中发生错误的位置。TransRollback() 允许实用程序文件安全地回滚发生错误

的事务。它使用嵌入式 SQL 语句 EXEC SQL ROLLBACK。以下示例说明 C 程序 dbuse 如何通过使用宏并为 SqlInfoPrint() 函数的 MSG_STR 参数提供值 "Delete with host variables -- Execute" 来调用实用程序函数:

```
EXEC SQL DELETE FROM org
  WHERE deptnumb = :hostVar1 AND
        division = :hostVar2;
EMB_SQL_CHECK("Delete with host variables -- Execute");
```

EMB_SQL_CHECK 宏确保在 DELETE 语句失败时安全地回滚事务并打印适当的错误消息。

建议开发者在创建自己的 DB2 程序时使用并扩展这些错误检查实用程序。

构建使用 C 和 C++ 编写的应用程序和例程

产品附带提供了用于各种操作系统平台的构建脚本。通过使用这些文件，可以构建使用 C 和 C++ 编写的嵌入式 SQL 应用程序。除用于构建应用程序的构建脚本以外，还提供了用于构建例程（存储过程和用户定义的函数）的特定 bldrtn 脚本。对于使用 VisualAge® 编写的应用程序和例程而言，使用配置文件来构建应用程序。随教程以及客户机级或实例级示例不同，提供的 C 应用程序样本也有所不同，您可以在 sqllib/samples/c 目录（对于 UNIX）和 sqllib\samples\c 目录（对于 Windows）中找到这些样本。

C 和 C++ 的编译和链接选项

AIX C 嵌入式 SQL 和 DB2 API 应用程序的编译和链接选项:

使用 AIX IBM C 编译器来构建 C 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 bldapp 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

xlc IBM XL C/C++ 编译器。

$EXTRA_CFLAG
对于已启用 64 位支持的实例，此选项包含“-q64”；否则，它不包含任何值。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: $HOME/sqllib/include。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

xlc 使用编译器作为链接程序的前端。

$EXTRA_CFLAG
对于已启用 64 位支持的实例，此选项包含“-q64”；否则，它不包含任何值。

-o $1 指定可执行程序。

$1.o 指定程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

-ldb2 与 DB2 库链接。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。例如: `$HOME/sqllib/$LIB`。如果未指定 **-L** 选项，那么编译器将采用以下路径: `/usr/lib:/lib`。

请参阅编译器文档，以了解其他编译器选项。

AIX C++ 嵌入式 SQL 和 DB2 管理 API 应用程序的编译和链接选项:

使用 AIX IBM XL C/C++ 编译器来构建 C++ 嵌入式 SQL 和 DB2 管理 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp` 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

xlC IBM XL C/C++ 编译器。

EXTRA_CFLAG
对于已启用 64 位支持的实例，此选项包含"-q64"；否则，它不包含任何值。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: `$HOME/sqllib/include`。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

xlC 使用编译器作为链接程序的前端。

EXTRA_CFLAG
对于已启用 64 位支持的实例，此选项包含"-q64"；否则，它不包含任何值。

-o $1 指定可执行程序。

$1.o 指定程序对象文件。

utilapi.o
包括用于非嵌入式 SQL 程序的 API 实用程序对象文件。

utilemb.o
包括用于嵌入式 SQL 程序的嵌入式 SQL 实用程序对象文件。

-ldb2 与 DB2 库链接。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。例如: `$HOME/sqllib/$LIB`。如果未指定 **-L** 选项，那么编译器将采用以下路径: `/usr/lib:/lib`。

请参阅编译器文档，以了解其他编译器选项。

HP-UX C 应用程序的编译和链接选项:

使用 HP-UX C 编译器来构建 C 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp` 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

cc C 编译器。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持，那么此标志包含值 **+DD64**；如果启用了 32 位支持，那么此标志包含值 **+DD32**。

+DD64 必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32 必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

-Ae 启用 HP ANSI 扩展方式。

-I$DB2PATH/include
指定 DB2 包含文件的位置。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

cc 使用编译器作为链接程序的前端。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持，那么此标志包含值 **+DD64**；如果启用了 32 位支持，那么此标志包含值 **+DD32**。

+DD64 必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32 必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

-o $1 指定可执行文件。

$1.o 指定程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

$EXTRA_LFLAG
指定运行时路径。如果设置了此选项，那么对于 32 位，它包含值 `-Wl,+b$HOME/sqllib/lib32`，对于 64 位，它包含值 `-Wl,+b$HOME/sqllib/lib64`。如果未设置此选项，那么它不包含任何值。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。对于 32 位：`$HOME/sqllib/lib32`；对于 64 位：`$HOME/sqllib/lib64`。

-ldb2 与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

HP-UX C++ 应用程序的编译和链接选项:

使用 HP-UX C++ 编译器来构建 C++ 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 bldapp 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

aCC　HP aC++ 编译器。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持，那么此标志包含值 **+DD64**；如果启用了 32 位支持，那么此标志包含值 **+DD32**。

+DD64　必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32　必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

-ext　启用各种 C++ 扩展，其中包括“long long”支持。

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: $HOME/sqllib/include。

-c　只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

aCC　使用 HP aC++ 编译器作为链接程序的前端。

$EXTRA_CFLAG
如果 HP-UX 平台为 IA64 并且启用了 64 位支持，那么此标志包含值 **+DD64**；如果启用了 32 位支持，那么此标志包含值 **+DD32**。

+DD64　必须使用此选项才能为 IA64 上的 HP-UX 生成 64 位代码。

+DD32　必须使用此选项才能为 IA64 上的 HP-UX 生成 32 位代码。

-o $1　指定可执行文件。

$1.o　指定程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

$EXTRA_LFLAG
指定运行时路径。如果设置了此选项，那么对于 32 位，它包含值“-Wl,+b$HOME/sqllib/lib32”，对于 64 位，它包含值“-Wl,+b$HOME/sqllib/lib64”。如果未设置此选项，那么它不包含任何值。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。对于 32 位: $HOME/sqllib/lib32; 对于 64 位: $HOME/sqllib/lib64。

-ldb2　与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Linux C 应用程序的编译和链接选项:

使用 Linux C 编译器来构建 C 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 bldapp 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

$CC gcc 或 xlc_r 编译器。

$EXTRA_C_FLAGS
包含下列其中一个标志:
- -m31（仅限于 Linux for zSeries®），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库;或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-I$DB2PATH/include
指定 DB2 包含文件的位置。

-c 只执行编译；不进行链接。此脚本文件包含独立的编译和链接步骤。

链接选项:

$CC gcc 或 xlc_r 编译器；使用编译器作为链接程序的前端。

$EXTRA_C_FLAGS
包含下列其中一个标志:
- -m31（仅限于 Linux for zSeries），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库;或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-o $1 指定可执行文件。

$1.o 指定对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

$EXTRA_LFLAG
对于 32 位，包含值“-Wl,-rpath,$DB2PATH/lib32”；对于 64 位，包含值“-Wl,-rpath,$DB2PATH/lib64”。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位: $HOME/sqllib/lib32；对于 64 位: $HOME/sqllib/lib64。

-ldb2 与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Linux C++ 应用程序的编译和链接选项:

使用 Linux C++ 编译器来构建 C++ 嵌入式 SQL 和 DB2 API 应用程序时使用的编译和链接选项，如 bldapp 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

g++ GNU/Linux C++ 编译器。

$EXTRA_C_FLAGS
包含下列其中一个标志:
- -m31（仅限于 Linux for zSeries），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库;或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-I$DB2PATH/include
指定 DB2 包含文件的位置。

-c 只执行编译；不进行链接。此脚本文件包含独立的编译和链接步骤。

链接选项:

g++ 使用编译器作为链接程序的前端。

$EXTRA_C_FLAGS
包含下列其中一个标志:
- -m31（仅限于 Linux for zSeries），用于构建 32 位库;
- -m32（仅限于 Linux for x86、x64 和 POWER），用于构建 32 位库;
- -m64（仅限于 Linux for zSeries、POWER 和 x64），用于构建 64 位库;或者
- 不包含任何值（Linux for IA64），用于构建 64 位库。

-o $1 指定可执行文件。

$1.o 包括程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

$EXTRA_LFLAG
对于 32 位，包含值“-Wl,-rpath,$DB2PATH/lib32”；对于 64 位，包含值“-Wl,-rpath,$DB2PATH/lib64”。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位：`$HOME/sqllib/lib32`；对于 64 位：`$HOME/sqllib/lib64`。

-ldb2 与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Solaris C 应用程序的编译和链接选项:

以下是您使用 Forte C 编译器来构建 C 嵌入式 SQL 和 DB2 API 应用程序时，DB2 建议您使用的编译和链接选项（如 `bldapp` 构建脚本所示）。

bldapp 的编译和链接选项

编译选项:

cc C 编译器。

-xarch=$CFLAG_ARCH
此选项确保与 `libdb2.so` 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值的设置如下所示:

- “v8plusa”: Solaris SPARC 上的 32 位应用程序
- “v9”: Solaris SPARC 上的 64 位应用程序
- “sse2”: Solaris x64 上的 32 位应用程序
- “amd64”: Solaris x64 上的 64 位应用程序

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如: `$HOME/sqllib/include`。

-c 只执行编译；不进行链接。此脚本包含独立的编译和链接步骤。

链接选项:

cc 使用编译器作为链接程序的前端。

-xarch=$CFLAG_ARCH
此选项确保与 `libdb2.so` 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值设置为“v8plusa”（表示 32 位）或“v9”（表示 64 位）。

-mt 在支持多线程的情况下进行链接。此选项用于与 libdb2 链接。

注: 如果使用 POSIX 线程，那么 DB2 应用程序还必须使用 `-lpthread` 进行链接，而无论它们是否线程化应用程序。

-o $1 指定可执行文件。

$1.o 包括程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位：`$HOME/sqllib/lib32`；对于 64 位：`$HOME/sqllib/lib64`。

$EXTRA_LFLAG
指定 DB2 共享库在运行时的位置。对于 32 位，包含值“-R$DB2PATH/lib32”；对于 64 位，包含值“-R$DB2PATH/lib64”。

-ldb2 与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Solaris C++ 应用程序的编译和链接选项:

以下是您使用 Forte C++ 编译器来构建 C++ 嵌入式 SQL 和 DB2 API 应用程序时，DB2 建议您使用的编译和链接选项（如 `bldapp` 构建脚本所示）。

bldapp 的编译和链接选项

编译选项:

CC C++ 编译器。

-xarch=$CFLAG_ARCH
此选项确保与 `libdb2.so` 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值的设置如下所示:

- “v8plusa”: Solaris SPARC 上的 32 位应用程序
- “v9”: Solaris SPARC 上的 64 位应用程序
- “sse2”: Solaris x64 上的 32 位应用程序
- “amd64”: Solaris x64 上的 64 位应用程序

-I$DB2PATH/include
指定 DB2 包含文件的位置。例如：`$HOME/sqllib/include`。

-c 只执行编译；不进行链接。此脚本包含独立的编译和链接步骤。

链接选项:

CC 使用编译器作为链接程序的前端。

-xarch=$CFLAG_ARCH
此选项确保与 `libdb2.so` 链接时，编译器生成有效的可执行文件。$CFLAG_ARCH 的值设置为“v8plusa”（表示 32 位）或“v9”（表示 64 位）。

-mt 在支持多线程的情况下进行链接。此选项用于与 libdb2 链接。

注：如果使用 POSIX 线程，那么 DB2 应用程序还必须使用 `-lpthread` 进行链接，而无论它们是否线程化应用程序。

-o $1 指定可执行文件。

$1.o 包括程序对象文件。

utilemb.o
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.o
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如，对于 32 位：`$HOME/sqllib/lib32`；对于 64 位：`$HOME/sqllib/lib64`。

$EXTRA_LFLAG
指定 DB2 共享库在运行时的位置。对于 32 位，包含值“-R$DB2PATH/lib32”；对于 64 位，包含值“-R$DB2PATH/lib64”。

-ldb2 与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Windows C 和 C++ 应用程序的编译和链接选项:

在 Windows 上使用 Microsoft Visual C++ 编译器构建 C 和 C++ 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp.bat` 批处理文件中所示。

bldapp 的编译和链接选项

编译选项:

%BLDCOMP%
编译器的变量。缺省值为 `cl`，即 Microsoft Visual C++ 编译器。另外，还可以将其设置为 `icl`（用于 32 位和 64 位应用程序的 Intel C++ 编译器）或 `ecl`（用于 Itanium 64 位应用程序的 Intel C++ 编译器）。

-Zi 启用调试信息。

-Od 禁止优化。在关闭优化的情况下使用调试器较为容易。

-c 只执行编译；不进行链接。此批处理文件包含独立的编译和链接步骤。

-W2 输出警告、错误、严重和不可恢复错误消息。

-DWIN32
Windows 操作系统所必需的编译器选项。

链接选项:

link 使用链接程序进行链接。

-debug 包括调试信息。

-out:%1.exe
指定文件名。

%1.obj 包括对象文件。

utilemb.obj
如果是嵌入式 SQL 程序，那么包含用于检查错误的嵌入式 SQL 实用程序对象文件。

utilapi.obj
如果不是嵌入式 SQL 程序，那么包含用于检查错误的 DB2 API 实用程序对象文件。

db2api.lib
与 DB2 库链接。

使用样本构建脚本来构建 C 或 C++ 应用程序 (UNIX)

关于此任务

DB2 提供了用于编译和链接 C 或 C++ 嵌入式 SQL 和 DB2 管理 API 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib/samples/c 目录（对于 C 应用程序）和 sqllib/samples/cpp 目录（对于 C++ 应用程序）中。

构建文件 bldapp 包含用于构建 DB2 应用程序的命令。

第一个参数 $1 指定源文件的名称。这是唯一的必需参数，并且，未包含嵌入式 SQL 的 DB2 管理 API 程序只需要此参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个可选参数：第二个参数 $2 指定要连接的数据库的名称；第三个参数 $3 指定数据库的用户标识，而 $4 则指定密码。

对于嵌入式 SQL 程序，bldapp 将参数传递给预编译和绑定脚本 embprep。如果未提供数据库名，那么将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

下列示例说明如何构建和运行 DB2 管理 API 和嵌入式 SQL 应用程序。

构建和运行 DB2 管理 API 应用程序

要根据源文件 cli_info.c (C) 和 cli_info.C (C++) 来构建 DB2 管理 API 样本程序 cli_info，请输入：

```
bldapp cli_info
```

这将生成可执行文件 cli_info。

要运行这个可执行文件，请输入可执行文件名：

```
cli_info
```

构建和运行嵌入式 SQL 应用程序

- 可以通过三种方法来根据源文件 tbmod.sqc (C) 和 tbmod.sqC (C++) 构建嵌入式 SQL 应用程序 tbmod：
  1. 如果已连接到同一实例中的 sample 数据库，请输入：
     ```
     bldapp tbmod
     ```
  2. 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名：
     ```
     bldapp tbmod 数据库
     ```
  3. 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码：
     ```
     bldapp tbmod database userid password
     ```
     这将生成可执行文件 tbmod
- 可以通过三种方法来运行这个嵌入式 SQL 应用程序：

1. 如果要访问同一实例中的 sample 数据库，请输入可执行文件名:

   ```
   tbmod
   ```

2. 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名:

   ```
   tbmod 数据库
   ```

3. 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码:

   ```
   tbmod database userid password
   ```

在 Windows 上构建 C/C++ 应用程序

DB2 提供了用于编译和链接 DB2 API 和嵌入式 SQL C/C++ 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib\samples\c 和 sqllib\samples\cpp 目录中。

关于此任务

批处理文件 bldapp.bat 包含用于构建 DB2 API 和嵌入式 SQL 程序的命令。此文件可以接受多达 4 个参数，在批处理文件中，它们分别由变量 %1、%2、%3 和 %4 表示。

第一个参数 %1 指定源文件的名称。对于未包含嵌入式 SQL 的程序而言，这是唯一的必需参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个附加参数: 第二个参数 %2 指定要连接的数据库的名称; 第三个参数 %3 指定数据库的用户标识，而 %4 则指定密码。

对于嵌入式 SQL 程序，bldapp 将参数传递给预编译和绑定文件 embprep.bat。如果未提供数据库名，那么将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

过程

- **构建和运行嵌入式 SQL 应用程序**

  可以通过三种方法来根据 sqllib\samples\c 中的 C 源文件 tbmod.sqc 或 sqllib\samples\cpp 中的 C++ 源文件 tbmod.sqx 来构建嵌入式 SQL 应用程序 tbmod:

  – 如果已连接到同一实例中的 sample 数据库，请输入:

    ```
    bldapp tbmod
    ```

  – 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名:

    ```
    bldapp tbmod 数据库
    ```

  – 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码:

    ```
    bldapp tbmod database userid password
    ```

  这将生成可执行文件 tbmod.exe。

  可以通过三种方法来运行这个嵌入式 SQL 应用程序:

  – 如果要访问同一实例中的 sample 数据库，请输入可执行文件名:

    ```
    tbmod
    ```

  – 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名:

    ```
    tbmod 数据库
    ```

– 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码:

```
tbmod database userid password
```

- **构建和运行多线程应用程序**

  在 Windows 上编译 C/C++ 多线程应用程序时，必须指定 -MT 或 -MD 选项。-MT 选项将使用静态库 LIBCMT.LIB 进行链接，而 -MD 将使用动态库 MSVCRT.LIB 进行链接。使用 -MD 链接的二进制文件较小但依赖于 MSVCRT.DLL，使用 -MT 链接的二进制文件较大但独立于运行时。

  批处理文件 bldmt.bat 使用 -MT 选项来构建多线程程序。所有其他编译和链接选项都与用于构建常规独立应用程序的批处理文件 bldapp.bat 所使用的选项相同。

  要根据 samples\c\dbthrds.sqc 或 samples\cpp\dbthrds.sqx 源文件来构建多线程样本程序 dbthrds，请输入:

  ```
  bldmt dbthrds
  ```

  这将生成可执行文件 dbthrds.exe。

  可以通过三种方法来运行这个多线程应用程序:

  – 如果要访问同一实例中的 sample 数据库，只需输入可执行文件名（不必指定扩展名）:

    ```
    dbthrds
    ```

  – 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名:

    ```
    dbthrds 数据库
    ```

  – 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码:

    ```
    dbthrds database userid password
    ```

示例

下列示例说明如何构建和运行 DB2 API 和嵌入式 SQL 应用程序。

要根据 sqllib\samples\c 中的源文件 cli_info.c 或 sqllib\samples\cpp 中的源文件 cli_info.cxx 来构建 DB2 API 非嵌入式 SQL 样本程序 cli_info，请输入:

```
bldapp cli_info
```

这将生成可执行文件 cli_info.exe。您可以通过在命令行输入可执行文件名（不必指定扩展名）来运行可执行文件:

```
cli_info
```

通过配置文件来构建使用 VisualAge C++ 编写的嵌入式 SQL 应用程序

关于此任务

VisualAge C++ 提供了递增编译器和批处理方式编译器。虽然批处理方式编译器使用 Makefile 和构建文件，但递增编译器使用配置文件。有关这方面的更多信息，请参阅 VisualAge C++ V5.0 附带提供的文档。

DB2 为您可以使用 VisualAge C++ 编译器构建的不同类型 DB2 程序提供了配置文件。

要使用 DB2 配置文件，请先将一个环境变量设置为所要编译的程序的名称。然后，使用 VisualAge C++ 提供的命令来编译程序。

在 Windows 上构建 C/C++ 多连接应用程序

关于此任务

DB2 Database for Linux, UNIX, and Windows 提供了用于编译和链接 C 和 C++ 嵌入式 SQL 和 DB2 API 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib\samples\c 和 sqllib\samples\cpp 目录中。

批处理文件 bldmc.bat 包含用于构建 DB2 多连接程序的命令，它需要两个数据库。编译和链接选项与 bldapp.bat 文件中使用的选项相同。

第一个参数 %1 指定源文件的名称。第二个参数 %2 指定所要连接的第一个数据库的名称。第三个参数 %3 指定所要连接的第二个数据库。它们全都是必需参数。

注: 此构建脚本硬编码了缺省值“sample”和“sample2”作为数据库名（%2 和 %3），因此，如果您使用此构建脚本并接受这些缺省值，那么只需指定程序名（%1 参数）。如果您使用 bldmc.bat 脚本，那么必须指定全部三个参数。

可选参数并不是本地连接所必需的，但如果要从远程客户机连接到服务器，那么这些参数是必需的。这些参数是: %4 和 %5（用于指定第一个数据库的用户标识和密码）以及 %6 和 %7（用于指定第二个数据库的用户标识和密码）。

对于多连接样本程序 dbmcon.exe 而言，需要两个数据库。如果尚未创建 sample 数据库，那么您可以通过在 DB2 命令窗口的命令行中输入 db2sampl 来创建此数据库。可以使用下列其中一条命令来创建第二个数据库（在本例中，此数据库名为 sample2）:

如果以本地方式创建数据库:

```
db2 create db sample2
```

如果以远程方式创建数据库:

```
db2 attach to 节点名
db2 create db sample2
db2 detach
db2 catalog db sample2 as sample2 at node 节点名
```

其中，*节点名* 是数据库所在的节点。

多连接还要求 TCP/IP 侦听器处于运行状态。

过程

要确保 TCP/IP 侦听器处于运行状态，请执行以下操作:

1. 将环境变量 **DB2COMM** 设置为 TCP/IP，如下所示:

   ```
   db2set DB2COMM=TCPIP
   ```

2. 使用 services 文件中指定的 TCP/IP 服务名称来更新数据库管理器配置文件:

   ```
   db2 update dbm cfg using SVCENAME TCPIP_service_name
   ```

每个实例都在服务文件中列示 TCP/IP 服务名称。如果您找不到该文件或者不具有更改服务文件所需的文件许可权，请与系统管理员联系。

3. 停止然后重新启动数据库管理器，以使这些更改生效:

```
db2stop
db2start
```

结果

dbmcon.exe 程序是根据 samples\c 或 samples\cpp 目录中的 5 个文件创建的:

dbmcon.sqc 或 dbmcon.sqx
用于同时连接到这两个数据库的主源文件。

dbmcon1.sqc 或 dbmcon1.sqx
用于创建与第一个数据库绑定的程序包的源文件。

dbmcon1.h
主源文件 dbmcon.sqc 或 dbmcon.sqx 中包括的 dbmcon1.sqc 或 dbmcon1.sqx 的头文件，其作用是访问用于创建和删除与第一个数据库绑定的表的 SQL 语句。

dbmcon2.sqc 或 dbmcon2.sqx
用于创建与第二个数据库绑定的程序包的源文件。

dbmcon2.h
主源文件 dbmcon.sqc 或 dbmcon.sqx 中包括的 dbmcon2.sqc 或 dbmcon2.sqx 的头文件，其作用是访问用于创建和删除与第二个数据库绑定的表的 SQL 语句。

要构建多连接样本程序 dbmcon.exe，请输入:

```
bldmc dbmcon sample sample2
```

这将生成可执行文件 dbmcon.exe。

要运行这个可执行文件，请输入可执行文件名（不必指定扩展名）:

```
dbmcon
```

此程序演示对两个数据库执行一阶段落实。

构建使用 COBOL 编写的应用程序和例程

产品附带提供了用于各种操作系统平台的构建脚本。通过使用这些文件，可以构建使用 COBOL 编写的嵌入式 SQL 应用程序。除用于构建应用程序的构建脚本以外，还提供了用于构建例程（存储过程和用户定义的函数）的特定 bldrtn 脚本。在 Linux 上处理使用 Micro Focus COBOL 语言编写的应用程序时，请务必将编译器配置为能够访问某些 COBOL 共享库。您可以在 sqllib/samples/cobol 目录（对于 UNIX）和 sqllib\samples\cobol 目录（对于 Windows）中找到我们提供的 IBM COBOL 样本，对于 Micro Focus COBOL 的 samples 目录，请将路径末尾的“cobol”替换为“cobol_mf”。

COBOL 的编译和链接选项

AIX IBM COBOL 应用程序的编译和链接选项:

使用 IBM COBOL for AIX 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 bldapp 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

cob2 IBM COBOL for AIX 编译器。

-qpgmname\(mixed\)
指示编译器允许调用具有混合大小写名称的库入口点。

-qlib 指示编译器处理 COPY 语句。

-I$DB2PATH/include/cobol_a
指定 DB2 包含文件的位置。例如: $HOME/sqllib/include/cobol_a。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

链接选项:

cob2 使用编译器作为链接程序的前端。

-o $1 指定可执行程序。

$1.o 指定程序对象文件。

checkerr.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。例如: $HOME/sqllib/lib32。

-ldb2 与数据库管理器库链接。

请参阅编译器文档，以了解其他编译器选项。

AIX Micro Focus COBOL 应用程序的编译和链接选项:

在 AIX 上使用 Micro Focus COBOL 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 bldapp 构建脚本中所示。注意，可以通过设置 COBCPY 环境变量来找到 DB2 MicroFocus COBOL 包含文件，因此不需要在编译步骤中指定 -I 标志。请参阅 bldapp 脚本以获取示例。

bldapp 的编译和链接选项

编译选项:

cob MicroFocus COBOL 编译器。

-c 只执行编译；不进行链接。

$EXTRA_COBOL_FLAG="-C MFSYNC"
启用 64 位支持。

-x 与 -c 配合使用时，生成对象文件。

链接选项:

cob 使用编译器作为链接程序的前端。

-x 生成可执行程序。

-o $1 指定可执行程序。

$1.o 指定程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。例如: `$HOME/sqllib/lib32`。

-ldb2 与 DB2 库链接。

-ldb2gmf
与 Micro Focus COBOL 的 DB2 异常处理程序库链接。

请参阅编译器文档，以了解其他编译器选项。

HP-UX Micro Focus COBOL 应用程序的编译和链接选项:

在 HP-UX 上使用 Micro Focus COBOL 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp` 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

cob Micro Focus COBOL 编译器。

-cx 编译为对象模块。

$EXTRA_COBOL_FLAG
如果 HP-UX 平台启用了 IA64 和 64 位支持，那么此标志包含"-C MFSYNC"。

链接选项:

cob 使用编译器作为链接程序的前端。

-x 指定可执行程序。

$1.o 包括程序对象文件。

checkerr.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。

-ldb2 与 DB2 库链接。

-ldb2gmf
与 Micro Focus COBOL 的 DB2 异常处理程序库链接。

请参阅编译器文档，以了解其他编译器选项。

Solaris Micro Focus COBOL 应用程序的编译和链接选项:

以下是在 Solaris 上使用 Micro Focus COBOL 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时使用的编译和链接选项，如 `bldapp` 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

cob Micro Focus COBOL 编译器。

$EXTRA_COBOL_FLAG
对于 64 位支持，包含值“-C MFSYNC”；否则，它不包含任何值。

-cx 编译为对象模块。

链接选项:

cob 使用编译器作为链接程序的前端。

-x 指定可执行程序。

$1.o 包括程序对象文件。

checkerr.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 静态库和共享库在链接时的位置。例如: `$HOME/sqllib/lib64`。

-ldb2 与 DB2 库链接。

-ldb2gmf
与 Micro Focus COBOL 的 DB2 异常处理程序库链接。

请参阅编译器文档，以了解其他编译器选项。

Linux Micro Focus COBOL 应用程序的编译和链接选项:

以下是在 Linux 上使用 Micro Focus COBOL 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时使用的编译和链接选项，如 `bldapp` 构建脚本中所示。

bldapp 的编译和链接选项

编译选项:

cob Micro Focus COBOL 编译器。

-cx 编译为对象模块。

$EXTRA_COBOL_FLAG
对于 64 位支持，包含值“-C MFSYNC”；否则，它不包含任何值。

链接选项:

cob 使用编译器作为链接程序的前端。

-x 指定可执行程序。

-o $1 包括可执行文件。

$1.o 包括程序对象文件。

checkerr.o
包括用于执行错误检查的实用程序对象文件。

-L$DB2PATH/$LIB
指定 DB2 运行时共享库的位置。

-ldb2 与 DB2 库链接。

-ldb2gmf
与 Micro Focus COBOL 的 DB2 异常处理程序库链接。

请参阅编译器文档，以了解其他编译器选项。

Windows IBM COBOL 应用程序的编译和链接选项:

在 Windows 上使用 IBM VisualAge COBOL 编译器构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp.bat` 批处理文件中所示。

bldapp 的编译和链接选项

编译选项:

cob2 IBM VisualAge COBOL 编译器。

-qpgmname(mixed)
指示编译器允许调用具有混合大小写名称的库入口点。

-c 只执行编译；不进行链接。编译和链接是不同的步骤。

-qlib 指示编译器处理 COPY 语句。

-I*路径* 指定 DB2 包含文件的位置。例如: `-I"%DB2PATH%\include\cobol_a"`。

%EXTRA_COMPFLAG%
如果将“set IBMCOB_PRECOMP=true”取消注释，那么将使用 IBM COBOL 预编译器来预编译嵌入式 SQL。根据输入参数的不同，将使用下列其中一种格式来调用预编译器:

-q"SQL('database sample CALL_RESOLUTION DEFERRED')"
使用缺省的 sample 数据库进行预编译并推迟调用解析。

-q"SQL('database %2 CALL_RESOLUTION DEFERRED')"
使用用户指定的数据库进行预编译并推迟调用解析。

-q"SQL('database %2 user %3 using %4 CALL_RESOLUTION DEFERRED')"
使用用户指定的数据库、用户标识和密码进行预编译并推迟调用解析。这是用于远程客户机访问的格式。

链接选项:

cob2 使用编译器作为链接程序的前端。

%1.obj 包括程序对象文件。

checkerr.obj
包括错误检查实用程序对象文件。

db2api.lib
与 DB2 库链接。

请参阅编译器文档，以了解其他编译器选项。

Windows Micro Focus COBOL 应用程序的编译和链接选项:

在 Windows 上使用 Micro Focus COBOL 编译器来构建 COBOL 嵌入式 SQL 和 DB2 API 应用程序时，使用 DB2 中提供的编译和链接选项，如 `bldapp.bat` 批处理文件中所示。

bldapp 的编译和链接选项

编译选项:

cobol Micro Focus COBOL 编译器。

链接选项:

cbllink
使用链接程序进行链接编辑。

-l 与 lcobol 库链接。

checkerr.obj
与错误检查实用程序对象文件链接。

db2api.lib
与 DB2 API 库链接。

请参阅编译器文档，以了解其他编译器选项。

COBOL 编译器配置

在 AIX 上配置 IBM COBOL 编译器:

关于此任务

如果开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，并且您正在使用 IBM COBOL Set for AIX 编译器，那么需要这些步骤。

过程

- 使用 **PRECOMPILE** 命令来预编译应用程序时，请使用 `target ibmcob` 选项。
- 请不要在源文件中使用跳进字符。
- 可以在源文件的第一行使用 `PROCESS` 和 `CBL` 关键字来设置编译选项。
- 如果应用程序只包含嵌入式 SQL，但未包含 DB2 API 调用，那么不需要使用 `pgmname(mixed)` 编译选项。如果使用了 DB2 API 调用，那么必须使用 `pgmname (mixed)` 编译选项。
- 如果您正在使用 IBM COBOL Set for AIX 编译器的“System z 主机数据类型支持”功能部件，那么应用程序的 DB2 包含文件在以下目录中:

  ```
  $HOME/sqllib/include/cobol_i
  ```

  如果您正在使用提供的脚本文件来构建 DB2 样本程序，那么必须将脚本文件中指定的包含文件路径更改为指向 `cobol_i` 目录而不是 `cobol_a` 目录。

  如果您未使用 IBM COBOL Set for AIX 编译器的“System z 主机数据类型支持”功能部件，或正在使用此编译器的低级版本，那么应用程序的 DB2 包含文件在以下目录中:

  ```
  $HOME/sqllib/include/cobol_a
  ```

  指定 COPY 文件名以包括 `.cbl` 扩展名，如下所示:

  ```
  COPY "sql.cbl".
  ```

在 Windows 上配置 IBM COBOL 编译器:

关于此任务

如果您开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，并且正在使用 IBM VisualAge COBOL 编译器，那么必须牢记多个事项。

过程

- 在使用 DB2 预编译器来预编译应用程序，并使用命令行处理器命令 `db2 prep` 时，请使用 `target ibmcob` 选项。
- 请不要在源文件中使用跳进字符。
- 在源文件中使用 `PROCESS` 和 `CBL` 关键字来设置编译选项。只能在第 8 到 72 列中指定关键字。
- 如果应用程序只包含嵌入式 SQL，但未包含 DB2 API 调用，那么不需要使用 `pgmname(mixed)` 编译选项。如果使用了 DB2 API 调用，那么必须使用 `pgmname (mixed)` 编译选项。
- 如果您正在使用 IBM VisualAge COBOL 编译器的“System/390 主数据类型支持”功能部件，那么应用程序的 DB2 包含文件在以下目录中:

  ```
  %DB2PATH%\include\cobol_i
  ```

  如果您正在使用提供的批处理文件来构建 DB2 样本程序，那么必须将批处理文件中指定的包含文件路径更改为指向 `cobol_i` 目录而不是 `cobol_a` 目录。

  如果未使用 IBM VisualAge COBOL 编译器的“System/390 主数据类型支持”功能部件，或者正在使用此编译器的先前版本，那么应用程序的 DB2 包含文件在以下目录中:

  ```
  %DB2PATH%\include\cobol_a
  ```

  `cobol_a` 是缺省目录。
- 指定 COPY 文件名以包括 `.cbl` 扩展名，如下所示:

  ```
  COPY "sql.cbl".
  ```

在 Windows 上配置 Micro Focus COBOL 编译器:

关于此任务

如果您开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，并且正在使用 Micro Focus 编译器，那么必须牢记多个事项。

过程

- 使用 **PRECOMPILE** 命令来预编译应用程序时，请使用 `target mfcob` 选项。
- 通过使用以下命令，确保 LIB 环境变量指向 `%DB2PATH%\lib`:

  ```
  set LIB="%DB2PATH%\lib;%LIB%"
  ```
- Micro Focus COBOL 的 DB2 COPY 文件驻留在 `%DB2PATH%\include\cobol_mf` 中。请将 `COBCPY` 环境变量设置为包括此目录，如下所示:

  ```
  set COBCPY="%DB2PATH%\include\cobol_mf;%COBCPY%"
  ```

  您必须确保在 `System` 设置中永久设置上述环境变量。可以通过完成下列步骤进行检查:

  1. 打开**控制面板**

2. 选择**系统**
3. 选择**高级**选项卡
4. 单击**环境变量**
5. 在**系统变量**列表中查找必需的环境变量。如果这些环境变量不存在，请在**系统变量**列表中添加这些变量

通过用户设置、命令提示符或脚本来设置这些变量并不足够。

下一步做什么

必须使用调用约定 74 来调用所有 DB2 应用程序编程接口。DB2 COBOL 预编译器将在 SPECIAL-NAMES 段中自动插入 CALL-CONVENTION 子句。如果不存在 SPECIAL-NAMES 段，那么 DB2 COBOL 预编译器将创建该段，如下所示:

```
Identification Division
Program-ID. "static".
special-names.
    call-convention 74 is DB2API.
```

并且，每当调用 DB2 API 时，预编译器都会自动地在“call”关键字后面放置符号 DB2API，该符号用于标识调用约定。例如，每当预编译器根据嵌入式 SQL 语句来生成 DB2 API 运行时调用时，都将发生这种情况。

如果在未进行预编译的应用程序中调用 DB2 API，那么您应该以手动方式在应用程序中创建与上面给出的 SPECIAL-NAMES 段类似的 SPECIAL-NAMES 段。如果直接调用 DB2 API，那么您需要以手动方式在“call”关键字后面添加 DB2API 符号。

在 Linux 上配置 Micro Focus COBOL 编译器:

关于此任务

要运行 Micro Focus COBOL 例程，Linux 运行时链接程序必须能够访问某些 COBOL 共享库，并且 DB2 必须能够装入这些库。由于执行此装入操作的程序在使用 setuid 特权的情况下运行，因此，它只在 /usr/lib 中查找从属库。

为 COBOL 共享库创建指向 /usr/lib 的符号链接作为根目录。创建指向 /usr/lib 符号链接的最简单方法是，将所有 COBOL 库文件从 $COBDIR/lib 链接到 /usr/lib:

```
ln -s $COBDIR/lib/libcob* /usr/lib
```

其中，$COBDIR 是 Micro Focus COBOL 的安装位置，此位置通常是 /opt/lib/mfcobol。

下面是用于链接每个文件的命令（假定 Micro Focus COBOL 安装在 /opt/lib/mfcobol 中）:

```
ln -s /opt/lib/mfcobol/lib/libcobrts.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobrts_t.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobcrtn.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobcrtn.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobmisc_t.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobscreen.so /usr/lib
```

```
ln -s /opt/lib/mfcobol/lib/libcobscreen.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace_t.so /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace.so.2 /usr/lib
ln -s /opt/lib/mfcobol/lib/libcobtrace_t.so.2 /usr/lib
```

在每个 DB2 实例上，需要完成以下过程:

过程

- 使用 **PRECOMPILE** 命令来预编译应用程序时，请使用 target mfcob 选项。
- 在 Micro Focus COBOL 环境变量 COBCPY 中，必须包括 DB2 COBOL COPY 文件目录。COBCPY 环境变量指定 COPY 文件的位置。Micro Focus COBOL 的 DB2 COPY 文件驻留在数据库实例目录下的 sqllib/include/cobol_mf 中。

  要包括该目录，请输入:

  – 在 bash 或 Korn shell 中:

  ```
  export COBCPY=$HOME/sqllib/include/cobol_mf:$COBDIR/cpylib
  ```

  – 在 C shell 中:

  ```
  setenv COBCPY $HOME/sqllib/include/cobol_mf:$COBDIR/cpylib
  ```

- 更新环境变量:

  – 在 bash 或 Korn shell 中:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/sqllib/lib:$COBDIR/lib
  ```

  – 在 C shell 中:

  ```
  setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$HOME/sqllib/lib:$COBDIR/lib
  ```

- 设置 DB2 环境列表:

  ```
  db2set DB2ENVLIST="COBDIR LD_LIBRARY_PATH"
  ```

结果

注: 您可能想在 .bashrc、.kshrc（取决于所使用的 shell）、.bash_profile、.profile（取决于所使用的 shell）或 .login 中设置 COBCPY、COBDIR 和 LD_LIBRARY_PATH。

在 AIX 上配置 Micro Focus COBOL 编译器:

关于此任务

如果使用 Micro Focus COBOL 编译器来开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，请执行以下列出的步骤。

过程

- 使用 **PRECOMPILE** 命令来预编译应用程序时，请使用 target mfcob 选项。
- 在 Micro Focus COBOL 环境变量 COBCPY 中，必须包括 DB2 COBOL COPY 文件目录。COBCPY 环境变量指定 COPY 文件的位置。Micro Focus COBOL 的 DB2 COPY 文件在数据库实例目录下的 sqllib/include/cobol_mf 中。

  要包括该目录，请输入:

  – 在 bash 或 Korn shell 中:

  ```
  export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
  ```

– 在 C shell 中:

```
setenv COBCPY $COBCPY:$HOME/sqllib/include/cobol_mf
```

注: 您可能想在 .profile 或 .login 文件中设置 COBCPY。

在 HP-UX 上配置 Micro Focus COBOL 编译器:

关于此任务

如果您开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，并且正在使用 Micro Focus COBOL 编译器，那么必须牢记多个事项。

过程

- 使用 **PRECOMPILE** 命令来预编译应用程序时，请使用 target mfcob 选项。
- 在 Micro Focus COBOL 环境变量 COBCPY 中，必须包括 DB2 COBOL COPY 文件目录。COBCPY 环境变量指定 COPY 文件的位置。Micro Focus COBOL 的 DB2 COPY 文件驻留在数据库实例目录下的 sqllib/include/cobol_mf 中。

  要包括该目录:

  – 在 bash 或 Korn shell 中，请输入:

  ```
  export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
  ```

  – 在 C shell 中，请输入:

  ```
  setenv COBCPY ${COBCPY}:${HOME}/sqllib/include/cobol_mf
  ```

  注: 您可能想在 .profile 或 .login 文件中设置 COBCPY。

在 Solaris 上配置 Micro Focus COBOL 编译器:

关于此任务

如果您开发包含嵌入式 SQL 和 DB2 API 调用的应用程序，并且正在使用 Micro Focus COBOL 编译器，那么必须牢记多个事项。

过程

- 使用命令行处理器命令 **db2 prep** 来预编译应用程序时，请使用 target mfcob 选项。
- 在 Micro Focus COBOL 环境变量 COBCPY 中，必须包括 DB2 COBOL COPY 文件目录。COBCPY 环境变量指定 COPY 文件的位置。Micro Focus COBOL 的 DB2 COPY 文件驻留在数据库实例目录下的 sqllib/include/cobol_mf 中。

  要包括该目录，请输入:

  – 在 bash 或 Korn shells 中:

  ```
  export COBCPY=$COBCPY:$HOME/sqllib/include/cobol_mf
  ```

  – 在 C shell 中:

  ```
  setenv COBCPY $COBCPY:$HOME/sqllib/include/cobol_mf
  ```

  注: 您可能想在 .profile 文件中设置 COBCPY。

在 AIX 上构建 IBM COBOL 应用程序

关于此任务

DB2 提供了用于编译和链接 IBM COBOL 嵌入式 SQL 和 DB2 管理 API 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib/samples/cobol 目录中。

构建文件 bldapp 包含用于构建 DB2 应用程序的命令。

第一个参数 $1 指定源文件的名称。对于未包含嵌入式 SQL 的程序而言，这是唯一的必需参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个可选参数：第二个参数 $2 指定要连接的数据库的名称；第三个参数 $3 指定数据库的用户标识，而 $4 则指定密码。

对于嵌入式 SQL 程序，bldapp 将参数传递给预编译和绑定脚本 embprep。如果未提供数据库名，那么将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

要根据源文件 client.cbl 来构建非嵌入式 SQL 样本程序 client，请输入：

```
bldapp client
```

这将生成可执行文件 client。您可以通过输入以下命令对 sample 数据库运行这个可执行文件：

```
client
```

过程

- 可以通过三种方法根据源文件 updat.sqb 构建嵌入式 SQL 应用程序 updat：
  1. 如果已连接到同一实例中的 sample 数据库，请输入：

     ```
     bldapp updat
     ```
  2. 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名：

     ```
     bldapp updat 数据库
     ```
  3. 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码：

     ```
     bldapp updat database userid password
     ```

     这将生成可执行文件 updat。
- 可以通过三种方法来运行这个嵌入式 SQL 应用程序：
  1. 如果要访问同一实例中的 sample 数据库，请输入可执行文件名：

     ```
     updat
     ```
  2. 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名：

     ```
     updat 数据库
     ```
  3. 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码：

     ```
     updat database userid password
     ```

构建 UNIX Micro Focus COBOL 应用程序

关于此任务

DB2 提供了用于编译和链接 Micro Focus COBOL 嵌入式 SQL 和 DB2 管理 API 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib/samples/cobol_mf 目录中。

构建文件 bldapp 包含用于构建 DB2 应用程序的命令。

第一个参数 $1 指定源文件的名称。对于未包含嵌入式 SQL 的程序而言，这是唯一的必需参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个可选参数：第二个参数 $2 指定要连接的数据库的名称；第三个参数 $3 指定数据库的用户标识，而 $4 则指定密码。

对于嵌入式 SQL 程序，bldapp 将参数传递给预编译和绑定脚本 embprep。如果未提供数据库名，那么将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

要根据源文件 client.cbl 来构建非嵌入式 SQL 样本程序 client，请输入：

```
bldapp client
```

这将生成可执行文件 client。您可以通过输入以下命令对 sample 数据库运行这个可执行文件：

```
client
```

过程

- 可以通过三种方法根据源文件 updat.sqb 构建嵌入式 SQL 应用程序 updat：
  1. 如果已连接到同一实例中的 sample 数据库，请输入：

     ```
     bldapp updat
     ```
  2. 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名：

     ```
     bldapp updat 数据库
     ```
  3. 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码：

     ```
     bldapp updat database userid password
     ```

     这将生成可执行文件 updat。
- 可以通过三种方法来运行这个嵌入式 SQL 应用程序：
  1. 如果要访问同一实例中的 sample 数据库，请输入可执行文件名：

     ```
     updat
     ```
  2. 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名：

     ```
     updat 数据库
     ```
  3. 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码：

     ```
     updat database userid password
     ```

在 Windows 上构建 IBM COBOL 应用程序

关于此任务

DB2 提供了用于编译和链接 DB2 API 和嵌入式 SQL 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib\samples\cobol 目录中。

在 Windows 上，DB2 支持使用两款预编译器来构建 IBM COBOL 应用程序，即 DB2 预编译器和 IBM COBOL 预编译器。缺省情况下，使用 DB2 预编译器。您可以通过在所使用的批处理文件中取消注释适当的行来选择 IBM COBOL 预编译器。对 IBM COBOL 进行的预编译由编译器本身使用特定的预编译选项完成。

批处理文件 bldapp.bat 包含用于构建 DB2 应用程序的命令。此文件可以接受多达 4 个参数，在批处理文件中，它们分别由变量 %1、%2、%3 和 %4 表示。

第一个参数 %1 指定源文件的名称。对于未包含嵌入式 SQL 的程序而言，这是唯一的必需参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个可选参数：第二个参数 %2 指定要连接的数据库的名称；第三个参数 %3 指定数据库的用户标识，而 %4 则指定密码。

对于使用缺省的 DB2 预编译器的嵌入式 SQL 程序，bldapp.bat 将参数传递给预编译和绑定文件 embprep.bat。

对于使用 IBM COBOL 预编译器的嵌入式 SQL 程序，bldapp.bat 将把 .sqb 源文件复制到 .cbl 源文件。编译器使用特定的预编译选项对 .cbl 源文件执行预编译。

无论使用哪款预编译器，如果未提供数据库名，都将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

下列示例说明如何构建和运行 DB2 API 和嵌入式 SQL 应用程序。

要根据源文件 client.cbl 来构建非嵌入式 SQL 样本程序 client，请输入：

```
bldapp client
```

这将生成可执行文件 client.exe。您可以通过输入可执行文件名（不必指定扩展名）对 sample 数据库运行可执行文件：

```
client
```

过程

- 可以通过三种方法根据源文件 updat.sqb 构建嵌入式 SQL 应用程序 updat：
  1. 如果已连接到同一实例中的 sample 数据库，请输入：

     ```
     bldapp updat
     ```
  2. 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名：

     ```
     bldapp updat 数据库
     ```
  3. 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码：

     ```
     bldapp updat database userid password
     ```

     这将生成可执行文件 updat。
- 可以通过三种方法来运行这个嵌入式 SQL 应用程序：

1. 如果要访问同一实例中的 sample 数据库，请输入可执行文件名:

   ```
   updat
   ```

2. 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名:

   ```
   updat 数据库
   ```

3. 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码:

   ```
   updat database userid password
   ```

在 Windows 上构建 Micro Focus COBOL 应用程序

关于此任务

DB2 提供了用于编译和链接 DB2 API 和嵌入式 SQL 程序的构建脚本。这些脚本与可以使用这些文件构建的样本程序一起放在 sqllib\samples\cobol_mf 目录中。

批处理文件 bldapp.bat 包含用于构建 DB2 应用程序的命令。此文件可以接受多达 4 个参数，在批处理文件中，它们分别由变量 %1、%2、%3 和 %4 表示。

第一个参数 %1 指定源文件的名称。对于未包含嵌入式 SQL 的程序而言，这是唯一的必需参数。构建嵌入式 SQL 程序需要连接到数据库，因此，还提供了三个可选参数: 第二个参数 %2 指定要连接的数据库的名称; 第三个参数 %3 指定数据库的用户标识，而 %4 则指定密码。

对于嵌入式 SQL 程序，bldapp 将参数传递给预编译和绑定批处理文件 embprep.bat。如果未提供数据库名，那么将使用缺省数据库 sample。仅当从中构建程序的实例与数据库所在的实例不同时，才需要用户标识和密码参数。

下列示例说明如何构建和运行 DB2 API 和嵌入式 SQL 应用程序。

要根据源文件 client.cbl 来构建非嵌入式 SQL 样本程序 client，请输入:

```
bldapp client
```

这将生成可执行文件 client.exe。您可以通过输入可执行文件名（不必指定扩展名）对 sample 数据库运行可执行文件:

```
client
```

过程

- 可以通过三种方法根据源文件 updat.sqb 构建嵌入式 SQL 应用程序 updat:
  1. 如果已连接到同一实例中的 sample 数据库，请输入:

     ```
     bldapp updat
     ```

  2. 如果已连接到同一实例中的另一个数据库，那么还需输入数据库名:

     ```
     bldapp updat 数据库
     ```

  3. 如果已连接到另一实例中的数据库，那么还需输入该数据库实例的用户标识和密码:

     ```
     bldapp updat database userid password
     ```

     这将生成可执行文件 updat.exe。
- 可以通过三种方法来运行这个嵌入式 SQL 应用程序:

1. 如果要访问同一实例中的 sample 数据库，请输入可执行文件名（不必指定扩展名）：

   ```
   updat
   ```

2. 如果要访问同一实例中的另一个数据库，请输入可执行文件名和数据库名:

   ```
   updat 数据库
   ```

3. 如果要访问另一实例中的数据库，请输入可执行文件名、数据库名以及该数据库实例的用户标识和密码:

   ```
   updat database userid password
   ```

构建和运行使用 REXX 编写的嵌入式 SQL 应用程序

不必对 REXX 应用程序进行预编译、编译或链接。您可以在 Windows 操作系统和 AIX 操作系统上构建和运行 REXX 应用程序。

关于此任务

在 Windows 操作系统上，您的应用程序文件必须具有 .CMD 扩展名。创建应用程序完成后，可以直接从操作系统命令提示符运行该应用程序。在 AIX 上，应用程序文件可以具有任何扩展名。

过程

要构建和运行 REXX 应用程序，请执行以下操作:

- 在 Windows 操作系统上，应用程序文件可以具有任何名称。创建应用程序完成后，可以通过调用 REXX 解释器从操作系统命令提示符运行该应用程序，如下所示:

  ```
  REXX 文件名
  ```

- 在 AIX 上，可以使用以下两种方法中的任何一种方法来运行应用程序:
  - 在 shell 命令提示符处，输入 rexx *name*，其中 *name* 是您的 REXX 程序的名称。
  - 如果 REXX 程序的第一行包含“幻数”（#!）并标识了 REXX/6000 解释器所在的目录，那么您可以通过在 shell 命令提示符处输入 REXX 程序的名称来运行该程序。例如，如果 REXX/6000 解释器文件在 /usr/bin 目录中，请包括以下行作为 REXX 程序的第一行:

    ```
    #! /usr/bin/rexx
    ```

    然后，通过在 shell 命令提示符处输入以下命令使该程序可执行:

    ```
    chmod +x 名称
    ```

    通过在 shell 命令提示符处输入 REXX 程序的文件名来运行该程序。

注: 在 AIX 上，应该设置 **LIBPATH** 环境变量以包括 REXX SQL 库 db2rexx 所在的目录。例如:

```
export LIBPATH=/lib:/usr/lib:/$DB2PATH/lib
```

REXX 的绑定文件

我们提供了 5 个绑定文件来支持 REXX 应用程序。这些文件的名称包括在 DB2UBIND.LST 文件中。每个绑定文件都使用不同的隔离级别进行预编译；因此，数据库中存储了 5 个不同的程序包。

这 5 个绑定文件是:

DB2ARXCS.BND

支持“游标稳定性”隔离级别。

DB2ARXRR.BND

支持“可重复读”隔离级别。

DB2ARXUR.BND

支持“未落实的读”隔离级别。

DB2ARXRS.BND

支持“读稳定性”隔离级别。

DB2ARXNC.BND

支持“不落实”隔离级别。使用某些主机或 System i 数据库服务器时，将使用此隔离级别。对于其他数据库，此隔离级别的行为与“未落实的读”隔离级别相同。

注: 在某些情况下，可能有必要显式地将这些文件与数据库绑定。

使用 SQLEXEC 例程时，在采用“游标稳定性”隔离级别的情况下创建的程序包用作缺省包。如果您需要另外某种隔离级别，可以在连接到数据库之前使用 SQLDBS CHANGE SQL ISOLATION LEVEL API 来更改隔离级别。这将导致对 SQLEXEC 例程进行的后续调用与指定的隔离级别相关联。

基于 Windows 的 REXX 应用程序除非知道会话中的任何其他 REXX 程序均未更改设置，否则不能假定缺省隔离级别处于生效状态。在连接到数据库之前，REXX 应用程序应该显式地设置隔离级别。

在 Windows 上构建 Object REXX 应用程序

关于此任务

Object REXX 是 REXX 语言的面向对象版本。已对传统 REXX 添加面向对象的扩展，但其现有函数和指令未更改。Object REXX 解释器是其先前版本的增强版本，它提供了对下列各项的附加支持:

- 类、对象和方法
- 消息传递和多态性
- 单一继承和多重继承

Object REXX 与传统 REXX 完全兼容。在本节中，每当使用 REXX 时，所指均为 REXX 的所有版本，其中包括 Object REXX。

您不必对 REXX 程序进行预编译或绑定。

在 Windows 上，REXX 程序不必以注释开头。但是，为了实现可移植性，建议您使每个 REXX 程序都以起始于第一行第一列的注释开头。这样，就可以在其他平台上区分该程序与批处理命令:

```
/* Any comment will do. */
```

您可以在 `sqllib\samples\rexx` 目录中找到 REXX 样本程序。

要运行样本 REXX 程序 `updat`，请输入:

```
rexx updat.cmd
```

从命令行构建嵌入式 SQL 应用程序

从命令行构建嵌入式 SQL 应用程序涉及下列步骤:

1. 通过发出 **PRECOMPILE** 命令来预编译应用程序
2. 如果已创建绑定文件，请通过发出 **BIND** 命令将此文件与数据库绑定，以便创建应用程序程序包。
3. 对经过修改的应用程序源文件以及未包含嵌入式 SQL 的源文件进行编译，以便创建应用程序对象文件（.obj 文件）。
4. 通过使用链接命令将应用程序对象文件与 DB2 和主语言库链接，以便创建可执行程序。

构建使用 C 或 C++ 编写的嵌入式 SQL 应用程序（Windows）

在编写源文件之后，必须构建嵌入式 SQL 应用程序。

关于此任务

构建过程的某些步骤取决于您所使用的编译器。此过程的每个步骤所附带提供的示例说明如何使用 Microsoft Visual Studio 6.0 编译器（这是一个 C 编译器）来构建名为 myapp 的应用程序。您可以逐个运行此过程中的每个步骤，也可以通过从 DB2 命令窗口提示符运行批处理文件来同时运行这些步骤。要获取可用于构建 %**DB2PATH**%\SQLLIB\samples\c\ 目录中的嵌入式 SQL 样本应用程序的批处理文件的示例，请参考 %**DB2PATH**%\SQLLIB\samples\c\bldapp.bat 文件。此批处理文件通过调用另一个批处理文件 %**DB2PATH**%\SQLLIB\samples\c\embprep.bat 来预编译应用程序以及将该应用程序与数据库绑定。

- 活动的数据库连接
- 扩展名为 .sqc（C）或 .sqx（C++）并包含嵌入式 SQL 的应用程序源代码文件
- 受支持的 C 或 C++ 编译器
- 运行 **PRECOMPILE** 命令和 **BIND** 命令所需的权限或特权

过程

1. 通过发出 **PRECOMPILE** 命令来预编译应用程序。 例如:

```
C 应用程序: db2 PRECOMPILE myapp.sqc BINDFILE
C++ 应用程序: db2 PRECOMPILE myapp.sqx BINDFILE
```

   PRECOMPILE 命令将生成一个 .c 或 .C 文件（此文件包含 .sqc 或 .sqC 文件中源代码的经过修改的形式）以及一个应用程序包。如果使用 **BINDFILE** 选项，那么 **PRECOMPILE** 命令将生成绑定文件。在以上示例中，此绑定文件名为 myapp.bnd。

2. 如果已创建绑定文件，请通过发出 BIND 命令将此文件与数据库绑定，以便创建应用程序程序包。例如:

```
db2 bind myapp.bnd
```

   BIND 命令使应用程序包与数据库相关联并将该程序包存储在数据库中。

3. 对经过修改的应用程序源文件以及未包含嵌入式 SQL 的源文件进行编译，以便创建应用程序对象文件（.obj 文件）。 例如:

```
C 应用程序: cl -Zi -Od -c -W2 -DWIN32 myapp.c
C++ 应用程序: cl -Zi -Od -c -W2 -DWIN32 myapp.cxx
```

4. 通过使用链接命令将应用程序对象文件与 DB2 和主语言库链接，以便创建可执行程序。 例如:

```
link -debug -out:myapp.exe myapp.obj
```

第 5 章 部署和运行嵌入式 SQL 应用程序

嵌入式 SQL 应用程序可移植，并可以部署在远程机器上。您可以在一个位置编译此应用程序，然后在另一台机器上运行程序包以便使用新机器上的数据库。

有关链接到 libdb2.so 的限制

在某些 Linux 分发上，libc 开发 rpm 随附

```
/usr/lib/libdb2.so
```

或

```
/usr/lib64/libdb2.so
```

库。这个库用于 Sleepycat Software 的 Berkeley DB 实现，与 IBM DB2 数据库系统无关。

如果您未计划使用 Berkeley DB，那么可以在系统上永久地将这些库文件重命名或删除。

如果您想要使用 Berkeley DB，那么可以将包含这些库文件的文件夹重命名并将环境变量修改为指向新文件夹。

第 6 章 启用迁移兼容性功能

DB2 数据库管理器提供了方便从其他数据库系统迁移嵌入的 SQL C 应用程序的功能。

可通过将预编译器选项 COMPATIBILITY_MODE 设置为 ORA 来启用这些兼容性功能。例如，当您编译名为 tbsel.sqc 的文件时，使用以下命令将启用兼容性功能:

```
$ db2 PRECOMPILE tbsel.sqc BINDFILE COMPATIBILITY_MODE ORA
```

打开兼容性方式时，支持下列功能:

- 将 C 数组主变量用于 FETCH INTO 语句
- 将 INDICATOR 变量数组用于 FETCH INTO 语句
- 新的 CONNECT 语句语法
- 使用双引号通过 INCLUDE 语句指定文件名
- VARCHAR 类型的简单类型定义

此外，还支持嵌入式 SQL C 和嵌入式 SQL C++ 的以下功能，但不需要将预编译器选项 COMPATIBILITY_MODE 设置为 ORA:

- 使用 BIND 命令 GENERIC 选项的 STATICASDYNAMIC 字符串为会话中的程序包绑定提供真正的动态 SQL 行为
- 将字符串文字用于 PREPARE 语句
- 将 BREAK 操作用于 WHENEVER 语句

C 数组主变量

通过使用 C 数组主变量，可以声明一个游标并对数组变量进行批量访存，直到到达该行的结尾。

同一次访存中使用的数组变量需要具有相同的元素数，否则将使用为数组变量声明的最小元素数并显示警告。数组变量的大小各不相同（从 2 到 32K）。

在一次 FETCH 中，可检索的最大记录数是为数组变量声明的最大元素数。如果在第一次访存后有更多行可用，那么可以重复 FETCH 语句以获取下一组行。访存的总行数的累积和存储在 sqlca.sqlerrd[2] 中。

在以下示例中，声明了两个数组主变量 *empno* 和 *lastname*。每个数组主变量最多可容纳 100 个元素。因为仅有一个 FETCH 语句，所以此示例检索 100 个或更少的行。

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname FROM employee;

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
```

```
 EXEC SQL FETCH empcr INTO :empno :lastname;  /* bulk fetch       */
 ...                                          /* 100 or less rows */
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

INDICATOR 变量数组

在 FETCH 语句中，可使用指示符变量数组来确定数组变量的任何元素是否为 NULL。如果指示符变量包含一个小于零的值，那么这标识相应的数组值为 NULL。

可使用关键字 INDICATOR 来标识指示符变量，如以下示例所示。

在以下示例中，声明了称为 *bonus_ind* 的指示符变量数组。它可以具有最多 100 个元素，与为数组变量 *bonus* 声明的数量相同。访存数据时，如果 *bonus* 的值为 NULL，那么 *bonus_ind* 中的值将为负数。

```
EXEC SQL BEGIN DECLARE SECTION;
    char   empno[100][8];
    char   lastname[100][15];
    short  edlevel[100];
    double bonus[100];
    short  bonus_ind[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE empcr CURSOR FOR
 SELECT empno, lastname, edlevel, bonus
  FROM employee
  WHERE workdept = 'D21';

EXEC SQL OPEN empcr;

EXEC SQL WHENEVER NOT FOUND GOTO end_fetch;

while (1) {
 EXEC SQL FETCH empcr INTO :empno :lastname :edlevel,
         :bonus INDICATOR :bonus_ind
 ...
 ...
}
end_fetch:
EXEC SQL CLOSE empcr;
```

指示符变量不是由 INDICATOR 关键字标识，而是可以直接跟随在其相应的主变量之后，如以下示例中所示:

```
EXEC SQL FETCH empcr INTO :empno :lastname :edlevel, :bonus:bonus_ind
```

如果指示符数组变量的元素数与相应的主数组变量的元素数不匹配，那么将返回错误。

新的 CONNECT 语句语法

CONNECT 语句现在允许以下附加语法:

```
EXEC SQL CONNECT [ username IDENTIFIED BY password ][ USING  dbname ] ;
```

下表描述了这些参数:

参数	描述
用户名	用于指定数据库用户名的主变量或字符串
密码	用于指定密码的主变量或字符串
数据库名称	用于指定数据库名称的主变量或字符串

使用双引号指定包含文件名

可在 INCLUDE 伪指令中使用双引号来指定包含文件名（当 COMPATIBILITY MODE 未设置为 ORA 时，仅允许使用单引号）。例如:

```
EXEC SQL INCLUDE "abc.h";
```

VARCHAR 类型的简单类型定义

支持以下 VARCHAR 类型声明。预编译器将其展开为等价的 C 结构类型:

```
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR var_name [n+1];
EXEC SQL END DECLARE SECTION;
```

BIND 命令上 GENERIC 选项的 STATICASDYNAMIC 字符串

如果将 BIND 命令 GENERIC 选项上的 STATICASDYNAMIC 字符串设置为“yes”，那么 DB2 数据库管理器只会将所有语句存储在目录中并将它们标记为增量绑定。在运行时，当首次装入程序包时，数据库管理器使用当前会话环境（而不是程序包）来设置节条目和其他实体（将填充文本并访问程序包高速缓存）。

此后，绑定文件中的语句的行为就如同您正在使用动态 SQL 一样。例如，各节将针对数据库定义语言失效、特殊专用寄存器更新等隐式地重新编译。按如下方式定义该新语法:

```
DB2 BIND filename GENERIC 'STATICASDYNAMIC [YES|NO]'
```

将字符串文字用于 PREPARE 语句

PREPARE 语句由应用程序用于以动态方式准备 SQL 语句以便执行。PREPARE 语句根据语句的字符串格式（称为*语句字符串*）创建可执行的 SQL 语句。

对于嵌入式 C 和嵌入式 C++ 应用程序，除了能够根据主变量或表达式准备语句以外，*statement string* 还可以是字符串文字。

例如: `EXEC SQL PREPARE stmt_name FROM 'select empid from employee' ;`

WHENEVER 语句中的 BREAK 操作

WHENEVER 语句用于指定发生指定异常条件时要执行的操作。对于嵌入式 C 和嵌入式 C++ 应用程序，支持附加操作 BREAK。此操作会导致当前处理停止，例如，导致从 WHILE 循环中退出。

以下示例会导致紧随其后的语句在发生错误、警告或找不到数据时中断并退出循环:

```
EXEC SQL WHENEVER SQLERROR BREAK;
EXEC SQL WHENEVER SQLWARNING BREAK;
EXEC SQL WHENEVER NOT FOUND BREAK ;
```

附录 A. DB2 技术信息概述

DB2 技术信息以多种可以通过多种方法访问的格式提供。

您可以通过下列工具和方法获得 DB2 技术信息:

- DB2 信息中心
  - 主题（任务、概念和参考主题）
  - 样本程序
  - 教程
- DB2 书籍
  - PDF 文件（可下载）
  - PDF 文件（在 DB2 PDF DVD 中）
  - 印刷版书籍
- 命令行帮助
  - 命令帮助
  - 消息帮助

注: DB2 信息中心主题的更新频率比 PDF 书籍或硬拷贝书籍的更新频率高。要获取最新信息，请安装可用的文档更新或者参阅 ibm.com 上的 DB2 信息中心。

您可以在线访问 ibm.com 上的其他 DB2 技术信息，例如技术说明、白皮书和 IBM Redbooks® 出版物。请访问以下网址处的 DB2 信息管理软件资料库站点: http://www.ibm.com/software/data/sw-library/。

文档反馈

我们非常重视您对 DB2 文档的反馈。如果您想就如何改善 DB2 文档提出建议，请向 db2docs@ca.ibm.com 发送电子邮件。DB2 文档小组将阅读您的所有反馈，但无法直接给您答复。请尽可能提供具体的示例，这样我们才能更好地了解您所关心的问题。如果您要提供有关具体主题或帮助文件的反馈，请加上标题和 URL。

请不要使用以上电子邮件地址与 DB2 客户支持机构联系。如果您遇到文档无法解决的 DB2 技术问题，请与您当地的 IBM 服务中心联系以获得帮助。

硬拷贝或 PDF 格式的 DB2 技术库

下列各表描述 IBM 出版物中心（网址为 www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss）所提供的 DB2 资料库。可从 www.ibm.com/support/docview.wss?rs=71&uid=swg2700947 下载 PDF 格式的 DB2 V10.1 手册的英文版本和翻译版本。

尽管这些表标识书籍有印刷版，但可能未在您所在国家或地区提供。

每次更新手册时，表单号都会递增。确保您正在阅读下面列示的手册的最新版本。

注: *DB2 信息中心*的更新频率比 PDF 或硬拷贝书籍的更新频率高。

表 23. DB2 技术信息

书名	书号	是否提供印刷版	最近一次更新时间
Administrative API Reference	SC27-3864-00	是	2012 年 4 月
Administrative Routines and Views	SC27-3865-00	否	2012 年 4 月
Call Level Interface Guide and Reference Volume 1	SC27-3866-00	是	2012 年 4 月
Call Level Interface Guide and Reference Volume 2	SC27-3867-00	是	2012 年 4 月
Command Reference	SC27-3868-00	是	2012 年 4 月
数据库管理概念和配置参考	S151-1758-00	是	2012 年 4 月
Data Movement Utilities Guide and Reference	S151-1756-00	是	2012 年 4 月
数据库监视指南和参考	S151-1759-00	是	2012 年 4 月
数据恢复及高可用性指南与参考	S151-1755-00	是	2012 年 4 月
数据库安全性指南	S151-1753-01	是	2012 年 4 月
DB2 Workload Management Guide and Reference	SC27-3891-00	是	2012 年 4 月
开发 ADO.NET 和 OLE DB 应用程序	S151-1765-00	是	2012 年 4 月
开发嵌入式 SQL 应用程序	S151-1763-00	是	2012 年 4 月
Developing Java Applications	SC27-3875-00	是	2012 年 4 月
Developing Perl, PHP, Python, and Ruby on Rails Applications	SC27-3876-00	否	2012 年 4 月
开发用户定义的例程（SQL 和外部例程）	S151-1761-00	是	2012 年 4 月
数据库应用程序开发入门	G151-1764-00	是	2012 年 4 月
Linux 和 Windows 上的 DB2 安装和管理入门	G151-1769-00	是	2012 年 4 月
全球化指南	S151-1757-00	是	2012 年 4 月
安装 DB2 服务器	G151-1768-00	是	2012 年 4 月
安装 IBM Data Server Client	G151-1751-00	否	2012 年 4 月
消息参考第 1 卷	S151-1767-00	否	2012 年 4 月
消息参考第 2 卷	S151-1766-00	否	2012 年 4 月
Net Search Extender 管理和用户指南	S151-1078-00	否	2012 年 4 月

表 23. DB2 技术信息 (续)

书名	书号	是否提供印刷版	最近一次更新时间
分区和集群指南	S151-1754-00	是	2012 年 4 月
pureXML 指南	S151-1775-00	是	2012 年 4 月
Spatial Extender User's Guide and Reference	SC27-3894-00	否	2012 年 4 月
《SQL 过程语言: 应用程序启用和支持》	S151-1762-00	是	2012 年 4 月
SQL Reference Volume 1	SC27-3885-00	是	2012 年 4 月
SQL Reference Volume 2	SC27-3886-00	是	2012 年 4 月
Text Search Guide	SC27-3888-00	是	2012 年 4 月
故障诊断和调整数据库性能	S151-1760-00	是	2012 年 4 月
升级到 DB2 V10.1	S151-1770-00	是	2012 年 4 月
DB2 V10.1 新增内容	S151-1752-00	是	2012 年 4 月
XQuery 参考	S151-1774-00	否	2012 年 4 月

表 24. 特定于 DB2 Connect 的技术信息

书名	书号	是否提供印刷版	最近一次更新时间
DB2 Connect 安装和配置 DB2 Connect Personal Edition	S151-1773-00	是	2012 年 4 月
DB2 Connect 安装和配置 DB2 Connect 服务器	S151-1772-00	是	2012 年 4 月
DB2 Connect 用户指南	S151-1771-00	是	2012 年 4 月

从命令行处理器显示 SQL 状态帮助

DB2 产品针对可能充当 SQL 语句结果的条件返回 SQLSTATE 值。SQLSTATE 帮助说明 SQL 状态和 SQL 状态类代码的含义。

过程

要启动 SQL 状态帮助，请打开命令行处理器并输入:

```
? sqlstate or ? class code
```

其中，*sqlstate* 表示有效的 5 位 SQL 状态，*class code* 表示该 SQL 状态的前 2 位。例如，`? 08003` 显示 08003 SQL 状态的帮助，而 `? 08` 显示 08 类代码的帮助。

访问不同版本的 DB2 信息中心

您可以在 ibm.com® 上的不同信息中心中找到其他版本 DB2 产品的文档。

关于此任务

对于 DB2 V10.1 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v10r1。

对于 DB2 V9.8 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/。

对于 DB2 V9.7 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/。

对于 DB2 V9.5 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9r5。

对于 DB2 V9.1 主题，*DB2 信息中心* URL 是 http://publib.boulder.ibm.com/infocenter/db2luw/v9/。

对于 DB2 V8 主题，请转至 *DB2 信息中心* URL: http://publib.boulder.ibm.com/infocenter/db2luw/v8/。

更新安装在计算机或内部网服务器上的 DB2 信息中心

安装在本地的 DB2 信息中心必须定期进行更新。

开始之前

必须已安装 DB2 V10.1 信息中心。有关详细信息，请参阅安装 DB2 服务器中的“使用 DB2 安装向导来安装 DB2 信息中心”主题。所有适用于安装信息中心的先决条件和限制同样适用于更新信息中心。

关于此任务

可以自动或手动更新现有的 DB2 信息中心:

- 自动更新将更新现有的信息中心功能部件和语言。自动更新的一个优点是，与手动更新相比，信息中心的不可用时间较短。另外，自动更新可设置为作为定期运行的其他批处理作业的一部分运行。
- 可以使用手动更新方法来更新现有的信息中心功能部件和语言。自动更新可以缩短更新过程中的停机时间，但如果您想添加功能部件或语言，那么必须执行手动过程。例如，如果本地信息中心最初安装的是英语和法语版，而现在还要安装德语版；那么手动更新将安装德语版，并更新现有信息中心的功能和语言。但是，手动更新要求您手动停止、更新和重新启动信息中心。在整个更新过程期间信息中心不可用。在自动更新过程中，信息中心仅在更新完成后停止工作以重新启动信息中心。

此主题详细说明了自动更新的过程。有关手动更新的指示信息，请参阅“手动更新安装在您的计算机或内部网服务器上的 DB2 信息中心”主题。

过程

要自动更新安装在计算机或内部网服务器上的 DB2 信息中心:

1. 在 Linux 操作系统上，
   a. 浏览至信息中心的安装位置。缺省情况下，DB2 信息中心安装在 `/opt/ibm/db2ic/V10.1` 目录中。
   b. 从安装目录浏览至 `doc/bin` 目录。
   c. 运行 `update-ic` 脚本:

update-ic

2. 在 Windows 操作系统上，
   a. 打开命令窗口。
   b. 浏览至信息中心的安装位置。缺省情况下，DB2 信息中心安装在 <Program Files>\IBM\DB2 Information Center\V10.1 目录中，其中 <Program Files> 表示 Program Files 目录的位置。
   c. 从安装目录浏览至 doc\bin 目录。
   d. 运行 update-ic.bat 文件:

      update-ic.bat

结果

DB2 信息中心将自动重新启动。如果更新可用，那么信息中心会显示新的以及更新后的主题。如果信息中心更新不可用，那么会在日志中添加消息。日志文件位于 doc\eclipse\configuration 目录中。日志文件名称是随机生成的编号。例如，1239053440785.log。

手动更新安装在计算机或内部网服务器上的 DB2 信息中心

如果您已在本地安装 DB2 信息中心，那么可从 IBM 获取文档更新并进行安装。

关于此任务

手动更新安装在本地的 *DB2 信息中心*要求您:

1. 停止计算机上的 *DB2 信息中心*，然后以独立方式重新启动信息中心。如果以独立方式运行信息中心，那么网络上的其他用户将无法访问信息中心，因而您可以应用更新。DB2 信息中心的工作站版本总是以独立方式运行。
2. 使用“更新”功能部件来查看可用的更新。如果有您必须安装的更新，那么请使用“更新”功能部件来获取并安装这些更新。

   注: 如果您的环境要求在一台未连接至因特网的机器上安装 *DB2 信息中心*更新，请使用一台已连接至因特网并已安装 *DB2 信息中心*的机器将更新站点镜像至本地文件系统。如果网络中有许多用户将安装文档更新，那么可以通过在本地也为更新站点制作镜像并为更新站点创建代理来缩短每个人执行更新所需要的时间。
   如果提供了更新包，请使用“更新”功能部件来获取这些更新包。但是，只有在单机方式下才能使用“更新”功能部件。
3. 停止独立信息中心，然后在计算机上重新启动 *DB2 信息中心*。

注: 在 Windows 2008、Windows Vista 和更高版本上，稍后列示在此部分的命令必须作为管理员运行。要打开具有全面管理员特权的命令提示符或图形工具，请右键单击快捷方式，然后选择**以管理员身份运行**。

过程

要更新安装在您的计算机或内部网服务器上的 *DB2 信息中心*:

1. 停止 *DB2 信息中心*。
   - 在 Windows 上，单击**开始 > 控制面板 > 管理工具 > 服务**。右键单击 **DB2 信息中心**服务，并选择**停止**。

- 在 Linux 上，输入以下命令:

  ```
  /etc/init.d/db2icdv10 stop
  ```

2. 以独立方式启动信息中心。
   - 在 Windows 上:
     a. 打开命令窗口。
     b. 浏览至信息中心的安装位置。缺省情况下，*DB2 信息中心*安装在 *Program_Files*\IBM\DB2 Information Center\V10.1 目录中，其中 *Program Files* 表示 Program Files 目录的位置。
     c. 从安装目录浏览至 doc\bin 目录。
     d. 运行 help_start.bat 文件:

        ```
        help_start.bat
        ```

   - 在 Linux 上:
     a. 浏览至信息中心的安装位置。缺省情况下，*DB2 信息中心*安装在 /opt/ibm/db2ic/V10.1 目录中。
     b. 从安装目录浏览至 doc/bin 目录。
     c. 运行 help_start 脚本:

        ```
        help_start
        ```

   系统缺省 Web 浏览器将打开以显示独立信息中心。
3. 单击**更新**按钮（  ）。（必须在浏览器中启用 JavaScript。） 在信息中心的右边面板上，单击**查找更新**。 将显示现有文档的更新列表。
4. 要启动安装过程，请检查您要安装的选项，然后单击**安装更新**。
5. 在安装进程完成后，请单击**完成**。
6. 要停止独立信息中心，请执行下列操作:
   - 在 Windows 上，浏览至安装目录中的 doc\bin 目录并运行 help_end.bat 文件:

     ```
     help_end.bat
     ```

     注: help_end 批处理文件包含安全地停止使用 help_start 批处理文件启动的进程所需的命令。不要使用 Ctrl-C 或任何其他方法来停止 help_start.bat。
   - 在 Linux 上，浏览至安装目录中的 doc/bin 目录并运行 help_end 脚本:

     ```
     help_end
     ```

     注: help_end 脚本包含安全地停止使用 help_start 脚本启动的进程所需的命令。不要使用任何其他方法来停止 help_start 脚本。
7. 重新启动 *DB2 信息中心*。
   - 在 Windows 上，单击**开始 > 控制面板 > 管理工具 > 服务**。右键单击 **DB2 信息中心**服务，并选择**启动**。
   - 在 Linux 上，输入以下命令:

     ```
     /etc/init.d/db2icdv10 start
     ```

结果

更新后的 *DB2 信息中心*将显示新的以及更新后的主题。

DB2 教程

DB2 教程帮助您了解 DB2 数据库产品的各个方面。这些课程提供了逐步指示信息。

开始之前

您可以在信息中心中查看 XHTML 版的教程: http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/。

某些课程使用了样本数据或代码。有关其特定任务的任何先决条件的描述，请参阅教程。

DB2 教程

要查看教程，请单击标题。

*pureXML 指南*中的『**pureXML®**』

设置 DB2 数据库以存储 XML 数据以及对本机 XML 数据存储器执行基本操作。

DB2 故障诊断信息

我们提供了各种各样的故障诊断和问题确定信息来帮助您使用 DB2 数据库产品。

DB2 文档

您可以在*故障诊断和调整数据库性能*或者 *DB2 信息中心*的“数据库基础”部分中找到故障诊断信息，这些信息包含以下内容:

- 有关如何使用 DB2 诊断工具和实用程序来隔离和确定问题的信息。
- 一些最常见问题的解决方案。
- 旨在帮助您解决 DB2 数据库产品使用过程中可能会遇到的其他问题的建议。

IBM 支持门户网站

如果您遇到问题并且希望得到帮助以查找可能的原因和解决方案，请访问 IBM 支持门户网站。这个技术支持站点提供了指向最新 DB2 出版物、技术说明、授权程序分析报告（APAR 或错误修订）、修订包和其他资源的链接。可搜索此知识库并查找问题的可能解决方案。

访问 IBM 支持门户网站: http://www.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows

信息中心条款和条件

如果符合以下条款和条件，那么授予您使用这些出版物的许可权。

适用性: 用户需要遵循 IBM Web 站点的使用条款及以下条款和条件。

个人使用: 只要保留所有的专有权声明，您就可以为个人、非商业使用复制这些出版物。未经 IBM 明确同意，您不可以分发、展示或制作这些出版物或其中任何部分的演绎作品。

商业使用: 只要保留所有的专有权声明，您就可以仅在企业内复制、分发和展示这些出版物。未经 IBM 明确同意，您不可以制作这些出版物的演绎作品，或者在您的企业外部复制、分发或展示这些出版物或其中的任何部分。

权利：除非本许可权中明确授予，否则不得授予对这些出版物或其中包含的任何信息、数据、软件或其他知识产权的任何许可权、许可证或权利，无论是明示的还是暗含的。

IBM 保留根据自身的判断，认为对出版物的使用损害了 IBM 的权益（由 IBM 自身确定）或未正确遵循以上指示信息时，撤回此处所授予权限的权利。

只有您完全遵循所有适用的法律和法规，包括所有的美国出口法律和法规，您才可以下载、出口或再出口该信息。

IBM 对这些出版物的内容不作任何保证。这些出版物“按现状”提供，不附有任何种类的（无论是明示的还是暗含的）保证，包括但不限于暗含的关于适销和适用于某种特定用途的保证。

附录 B. 声明

本信息是为在美国提供的产品和服务编写的。有关非 IBM 产品的信息是基于首次出版此文档时的可获信息且会随时更新。

IBM 可能在其他国家或地区不提供本文档中讨论的产品、服务或功能特性。有关您当前所在区域的产品和服务的信息，请向您当地的 IBM 代表咨询。任何对 IBM 产品、程序或服务的引用并非意在明示或暗示只能使用 IBM 的产品、程序或服务。只要不侵犯 IBM 的知识产权，任何同等功能的产品、程序或服务，都可以代替 IBM 产品、程序或服务。但是，评估和验证任何非 IBM 产品、程序或服务，则由用户自行负责。

IBM 公司可能已拥有或正在申请与本文档内容有关的各项专利。提供本文档并未授予用户使用这些专利的任何许可。您可以用书面方式将许可查询寄往:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

有关双字节字符集 (DBCS) 信息的许可查询，请与您所在国家或地区的 IBM 知识产权部门联系，或用书面方式将查询寄往:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

本条款不适用英国或任何这样的条款与当地法律不一致的国家或地区: International Business Machines Corporation“按现状”提供本出版物，不附有任何种类的（无论是明示的还是暗含的）保证，包括但不限于暗含的有关非侵权、适销和适用于某种特定用途的保证。某些国家或地区在某些交易中不允许免除明示或暗含的保证。因此本条款可能不适用于您。

本信息中可能包含技术方面不够准确的地方或印刷错误。此处的信息将定期更改；这些更改将编入本资料的新版本中。IBM 可以随时对本资料中描述的产品和/或程序进行改进和/或更改，而不另行通知。

本信息中对非 IBM Web 站点的任何引用都只是为了方便起见才提供的，不以任何方式充当对那些 Web 站点的保证。那些 Web 站点中的资料不是此 IBM 产品资料的一部分，使用那些 Web 站点带来的风险将由您自行承担。

IBM 可以按它认为适当的任何方式使用或分发您所提供的任何信息而无须对您承担任何责任。

本程序的被许可方如果要了解有关程序的信息以达到如下目的: (i) 允许在独立创建的程序和其他程序（包括本程序）之间进行信息交换，以及 (ii) 允许对已经交换的信息进行相互使用，请与下列地址联系:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

只要遵守适当的条款和条件，包括某些情形下的一定数量的付费，都可获得这方面的信息。

本资料中描述的许可程序及其所有可用的许可资料均由 IBM 依据 IBM 客户协议、IBM 国际软件许可协议或任何同等协议中的条款提供。

此处包含的任何性能数据都是在受控环境中测得的。因此，在其他操作环境中获得的数据可能会有明显的不同。有些测量可能是在开发级的系统上进行的，因此不保证与一般可用系统上进行的测量结果相同。此外，有些测量是通过推算而估计的，实际结果可能会有差异。本文档的用户应当验证其特定环境的适用数据。

涉及非 IBM 产品的信息可从这些产品的供应商、其出版说明或其他可公开获得的资料中获取。IBM 没有对这些产品进行测试，也无法确认其性能的精确性、兼容性或任何其他关于非 IBM 产品的声明。有关非 IBM 产品性能的问题应当向这些产品的供应商提出。

所有关于 IBM 未来方向或意向的声明都可随时更改或收回，而不另行通知，它们仅仅表示了目标和意愿而已。

本信息可能包含在日常业务操作中使用的数据和报告的示例。为了尽可能完整地说明这些示例，示例中可能会包括个人、公司、品牌和产品的名称。所有这些名称都是虚构的，与实际商业企业所用的名称和地址的任何雷同纯属巧合。

版权许可:

本信息包括源语言形式的样本应用程序，这些样本说明不同操作平台上的编程方法。如果是为按照在编写样本程序的操作平台上的应用程序编程接口 (API) 进行应用程序的开发、使用、经销或分发，您可以任何形式对这些样本程序进行复制、修改、分发，而无须向 IBM 付费。这些示例并未在所有条件下作全面测试。因此，IBM 不能担保或暗示这些程序的可靠性、可维护性或功能。此样本程序“按现状”提供，且不附有任何种类的保证。对于使用此样本程序所引起的任何损坏，IBM 将不承担责任。

凡这些样本程序的每份拷贝或其任何部分或任何衍生产品，都必须包括如下版权声明:

©（*贵公司的名称*）（*年份*）. 此部分代码是根据 IBM 公司的样本程序衍生出来的。© Copyright IBM Corp. *（输入年份）*. All rights reserved.

商标

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other prod-

uct and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at 『Copyright and trademark information』 at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

索引

[B]

[C]

[D]

[E]

[F]

[G]

[H]

[J]

[K]

[L]

[P]

[Q]

[R]

[S]

[T]

[W]

[X]

[Y]

[Z]

[数字]

A

B

C

D

E

F

G

H

I

L

M

N

O

T

U

V

W

X

[特别字符]

Printed in China

S151-1763-00

Spine information:

IBM DB2 10.1 for Linux, UNIX, and Windows

开发嵌入式 SQL 应用程序