DBMaster

ODBC 程序员参考手册

SYSCOM

SYSCOM Computer Engineering Co./Corporate Headquarters

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

www.dbmaker.com

www.dbmaker.com.tw/service

Document No.645049-237460/DBM54CN-M03312017-ODBC

发行日期: 2017-03-31

对于本手册中没有体现的关于产品最新功能的描述，请在安装完成SYSCOM DBMaster 软件后阅读 README.TXT 文件。

注册商标
SYSCOM，SYSCOM 图标和 DBMaster 是SYSCOM 公司的注册商标。
Microsoft，MS-DOS，Windows 和 Windows NT 是 Microsoft 公司的注册商标。
UNIX 是 The Open Group 的注册商标。
ANSI 是美国国家标准化组织的注册商标。

手册中提到的其他产品名称或许是它们各自持有者的注册商标，仅仅是为提供此信息。
SQL 是行业语言，并不为任何公司或任何组织所有。

注意事项
本手册中有关软件的描述，均以该软件所提供的使用许可为基础。

对于授权许可的详细信息，请与您的经销商联系。关于计算机产品的特殊用途的市场性与适用性，经销商不会给予任何说明和保证。因外界因素如地震、过热、过冷和潮湿而引起产品的任何损坏以及由于使用不正确的电压和不兼容的软硬件而引起的损失和损坏，经销商概不负责。

虽然该手册的内容已经过仔细核对，但错误再所难免。若手册再有改动，不另行通知。还请见谅。

目录

1 简介 1-1

1.1 其它相关文件 **1-3**

1.2 技术支持 **1-4**

1.3 文档协定 **1-5**

2 示例程序 2-1

2.1 库模式 **2-2**

2.2 必需文件 **2-4**

头文件 2-4

链接库 2-4

2.3 范例**ODBC**应用程序 **2-6**

2.4 编译和链接 **2-8**

2.5 示例程序 **2-9**

3 数据库连接 3-1

3.1 环境句柄 **3-3**

3.2 连接句柄 **3-4**

3.3 连接到一个数据源 **3-5**

SQLConnect 3-5

SQLDriverConnect 3-8

多重连接 3-12
3.4 连接选项 3-13
SQLSetConnectOption 3-13
SQLGetConnectOption 3-14
3.5 释放句柄 3-15
SQLDisconnect 3-15
SQLFreeConnect 3-15
SQLFreeEnv 3-15
4 SQL语句 4-1
4.1 SQL语言 4-3
SQL在ODBC中的作用 4-3
基础的SQL语句 4-4
数据定义语言 4-4
数据操作语言 (DML) 4-5
4.2 执行SQL语句 4-10
SQLAllocStmt 4-10
SQLExecDirect 4-11
SQLRowCount 4-11
SQLFreeStmt 4-12
SQLPrepare和SQLExecute 4-14
4.3 参数 4-15
参数函数 4-15
在SQLExecDirect中使用参数 4-24
清除绑定的参数 4-25
4.4 输入大数据 4-26
如何输入大数据 4-26
取消执行SQLPutData 4-30
在文件对象中存放大数据 4-31
4.5 Get和Set选项 4-33
5 返回结果 5-1
5.1 使用ODBC查询 5-3

绑定存储单元和获取数据 5-3
结果字段的属性 5-5
结果字段的更多信息 5-12
清除绑定字段 5-14
5.2 游标 5-16
何时使用游标 5-16
获取游标名称 5-16
使用游标 5-17
设置游标名称 5-19
5.3 获取大数据 5-20
SQLGetData 5-22
停止SQLGetData操作 5-26
绑定字段以返回文件对象 5-26
获取文件对象的文件名 5-27
5.4 处理结果集 5-28
Rowsets 5-28
程序流程 5-28
存储单元绑定 5-29
定位游标 5-35
SQLExtendedFetch参数 5-35
返回值&错误处理 5-40
使用SQLSetPos修改表 5-42
字段指示器 5-48
SQLPutData 5-48
使用 SQLSetPos 5-52
6 错误处理 6-1
6.1 获取错误信息 6-2
ODBC中普通错误代码的定义 6-2
如何使用SQLError 6-2
错误列 6-6
6.2 目录函数 6-8
查找模式 6-8

SQLTables 6-9
SQLColumns 6-12
SQLStatistics 6-14
SQLSpecialColumns 6-15
6.3 系统信息 6-18
SQLGetTypeInfo 6-18
SQLGetInfo 6-21
SQLGetFunctions 6-22
SQLGetDiagRec 6-23
6.4 程序信息 6-24
SQLProcedureColumns 6-24
SQLProcedures 6-27
7 事务控制 7-1
7.1 事务和保存点 7-2
7.2 终止一个事务 7-5
7.3 自动提交和手动提交 7-7
8 ODBC 3.0函数 8-1
8.1 禁止使用的函数 8-2
8.2 修改过的函数 8-4
SQLCancel 8-4
SQLColumns 8-4
SQLFetch 8-5
SQLGetData 8-5
SQLGetFunctions 8-5
SQLGetInfo 8-6
SQLProcedureColumns 8-6
8.3 新函数 8-7
SQLAllocHandle 8-7
SQLBulkOperations 8-8
SQLCloseCursor 8-10
SQLColAttribute 8-10

SQLCopyDesc 8-13
SQLEndTran 8-15
SQLFetchScroll 8-15
SQLForeignKeys 8-17
SQLFreeHandle 8-18
SQLGetConnectAttr 8-19
SQLGetDescField 8-20
SQLGetDescRec 8-23
SQLGetDiagField 8-24
SQLGetDiagRec 8-26
SQLGetEnvAttr 8-26
SQLGetStmtAttr 8-27
SQLPrimaryKeys 8-29
SQLSetConnectAttr 8-30
SQLSetDescField 8-31
SQLSetDescRec 8-34
SQLSetEnvAttr 8-36
SQLSetStmtAttr 8-36

8.4 ODBC支持64位 8-39

ODBC函数 8-39

9 Unicode支持 9-1

9.1 Unicode编码接口 9-2

Unicode 函数 9-2

A 函数序列差异 1

A.1 SQLRowCount 2

A.2 SQLGetCursorName 3

B 函数特性差异 1

B.1 SQLPutData 2

B.2 SQLColumns 3

B.3 SQLTables 4

B.4 SQLDriverConnect 5

B.5 SQLBindParameter 6
B.6 DELETE/UPDATE位置 7
B.7 SQLSetConnectOption 8
B.8 SQLGetConnectOption 11
C ODBC 3.0错误 1
C.1 SQLParamData 2
C.2 SQLPrepare 3
D 数据类型 1
D.1 ODBC SQL数据类型 2
D.2 ODBC C数据类型 4
D.3 默认的ODBC C数据类型 6
D.4 精度、刻度、长度与显示大小 7
D.5 Data类型转化 9
SQL转化为C数据类型 9
C转化为SQL数据类型 14
E ODBC日志函数 1

1 简介

欢迎使用ODBC程序员手册。DBMaster是一个功能强大且使用灵活的SQL数据库管理系统(DBMS)，它支持交互式的结构化查询语言(SQL)、Microsoft开放式数据库连接(ODBC)标准接口以及嵌入式的ESQL/C语言。由于DBMaster完全遵循开放式的架构以及标准的ODBC接口，而市面上的绝大多数开发工具都支持标准ODBC，所以用户可以有更多的选择用来编译自定义应用程序。

DBMaster可以很容易地从个人使用的“单用户数据库”升级到企业级的“分布式数据库”。无论用户的数据库是何种结构，DBMaster先进的安全性、完整性和可靠性都能保证用户重要数据的安全。另外，DBMaster的跨平台支持特性则在用户需要作硬件升级时，提供了最佳的扩展弹性。

DBMaster提供了卓越的多媒体处理能力，可以让用户存储、查询、恢复和操纵各种类型的多媒体数据。利用DBMaster提供的二进制大型对象(BLOB)，可以让用户的多媒体数据完全享有DBMaster先进的安全性和灾难恢复功能。而利用文件对象(File Object)数据类型，也可以让用户的应用程序能够直接编辑DBMaster数据库的外部文件。

本手册主要针对那些为DBMaster创建前后端应用程序的程序员。首先用户应该对C语言（此后简称C）有足够的了解，如果想编译和执行示例程序，同时还应该安装一个可用的C语言开发工具。

C程序的信息不在本手册的范围之内，若用户遇到此类问题可参考C语言开发手册。如果在开发工具里编译运行示例程序时遇到此类问题，用户可参考开发工具文件或求助于开发工具供应商。

本手册将介绍DBMaster ODBC API以及如何使用DBMaster ODBC API为数据库构造一个前后端应用程序的概要。由于本手册仅是一个ODBC程序的介绍，ODBC的概念和执行也许不会被全部适用，但是所有提到的概念将以足够的深度被使用，以便用户理解在示例程序中发生了什么以及发生的原因。

每一章都介绍了一组关系函数和它们的选项，并解释用户可能遇到的DBMaster ODBC API和Microsoft ODBC 2.1规格（关于DBMaster ODBC 3.0 API的信息，请参考ODBC 3.0函数的第8章）之间的不同。用户将学习到如何使用函数以及如何使每个函数都适合所有程序。

本手册中提供的示例和图表可用来帮助用户理解当前信息。使用C编写的示例程序能够通过任何一种C/C++编译器来编译。

尽管本手册提供了所有的DBMaster ODBC函数信息，但它并不是Microsoft ODBC 3.0 API的一个综合参考。当使用本手册时，如果用户需要一个所有函数和语法转换的详细信息参考操作时，您会发现本手册的有用之处。Microsoft Press推荐参考的是*ODBC3.0程序员参考手册*。

1.1 其它相关文件

除了本手册外，DBMaster还提供了一整套的数据库管理系统（DBMS）手册，用户可以根据特定需要参考以下手册。

- 有关DBMaster性能和功能的介绍，请参考*DBMaster指南*。
- 有关设计、管理和维护DBMaster数据库的信息，请参考*数据库管理员手册*。
- 有关如何管理DBMaster的信息，请参考*服务器管理工具用户手册*。
- 有关DBMaster的配置信息，请参考*配置管理工具用户手册*。
- 有关DBMaster功能的信息，请参考*数据库管理工具用户手册*。
- 有关dmSQL命令行工具的使用方法，请参考*dmSQL使用手册*。
- 有关DBMaster SQL语言的语法和使用的相关信息，请参考*SQL命令与函数参考手册*。
- 有关嵌入式ESQL/C语言的语法和使用，请参考*ESQL/C程序员参考手册*。
- 有关DBMaster的错误信息及警告信息，请参考*错误信息参考手册*。
- 有关DCI COBOL接口的详细信息，请参考*DBMasterDCI用户手册*。
- 有关JDBC API的信息，请参考*JDBC程序员参考手册*。

1.2 技术支持

在软件评估期间，Syscom会为用户提供30天的免费email支持和电话支持。当软件注册后，我们还会再为用户提供30天的免费技术支持。因此，用户就可以获得60天的免费技术支持。在30天或60天的免费技术支持到期后，Syscom仍将继续通过电子邮件的方式为用户提供技术支持。

用户除了可以获得免费的技术支持外，还可以以20%的零售价购买其它产品。具体详情和价格请与sales@dbmaker.com保持联系。

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

无论用户以何种方式与Syscom的技术支持联系时，请务必写上以下有效信息：

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

1.3 文档协定

为方便用户的阅读和使用，本手册使用了一种标准的排版约定：注释、程序、示例和命令行都用缩进排版的方式进行了特别的设置。

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

表1- 1文档协定

2 示例程序

在安装ODBC软件和DBMaster ODBC驱动之后，就可以使用DBMaster构建一个简单的ODBC程序。本章将指导用户如何使用DBMaster ODBC驱动来编译并链接一个ODBC程序。

2.1 库模式

您可以使用DBMaster函数库或ODBC函数库创建一个ODBC应用程序。如果应用程序需要使用低级系统调用操作系统、文件系统或特定硬件驱动器，那么用户也可以使用自己的函数库，*图2-1*和*图2-2*将展示这些库的接口模式。

*图2-1*模拟*ODBC*驱动管理器的使用，图*2-2*模拟DBMaster驱动器而非ODBC驱动管理器的使用。当使用*ODBC*驱动管理器时，用户可以连接DBMaster以外的数据源。若不使用*驱动管理器*，用户的应用程序将直接连接到DBMaster函数库以达到更高的性能，但却丧失了连接到其它数据源的能力。

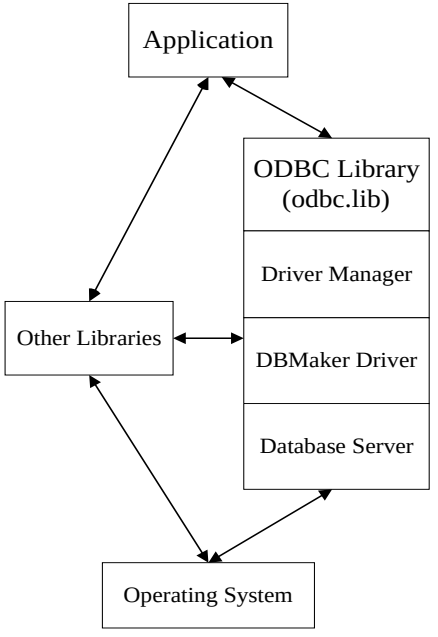

图2- 1: 当使用ODBC驱动管理时的ODBC库模式

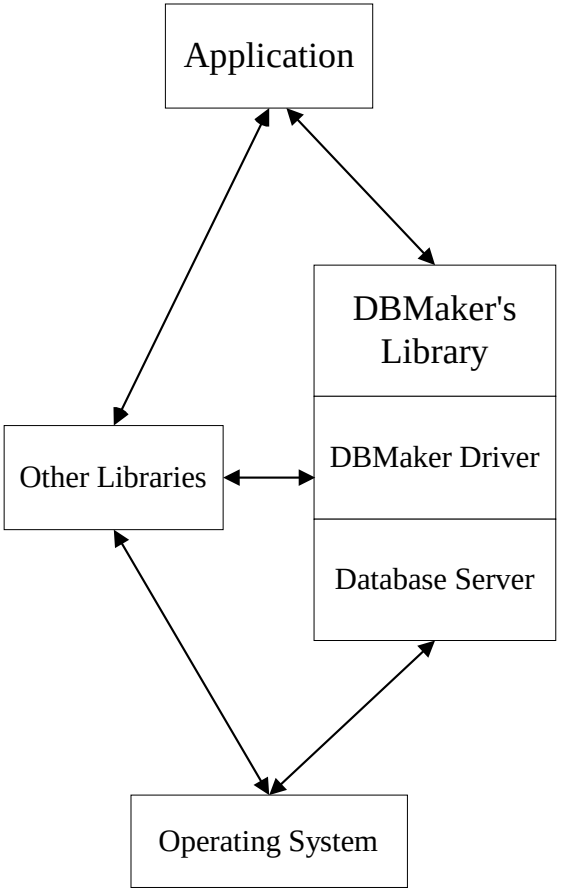

图2-2: 当直接使用DBMaster驱动时的ODBC库模式

2.2 必需文件

使用DBMaster ODBC驱动器构建一个ODBC程序时，可在makefile文件里指定头文件和链接库。以下我们使用DBMaster 5.4版本的C为例。

头文件

当创建一个ODBC应用程序，以下的头文件是必需的：***SQL.h***、***SQLext.h***、***SQLopt.h***和***SQLunix.h***(***SQLunix.h***仅在UNIX下需要)。

SQL.h和***SQLext.h***是标准ODBC所包含的文件，DBMaster在UNIX下为一些特定驱动器选项分别提供***SQLopt.h***和***SQLunix.h***。因为***SQLext.h***也包括***SQL.h***，*所以*用户仅需在程序里含有***SQLext.h***就可以。

头文件在所有平台是一样的，除了***SQLunix.h***，它只被应用于UINX平台运行的程序。

在Microsoft Windows系统，如果DBMaster安装在默认路径下，那么可在*c:\dbmaster\5.4\include*路径下找到这些文件，否则这些文件将位于*d:\install_directory\include*目录下，其中*d:*是DBMaster驱动器的安装磁盘，*install_directory*是路径。

在UINX环境，这些文件位于*~dbmaster/5.4/include*路径下。

链接库

当使用DBMaster创建ODBC应用程序时，连接库是必需的，使用哪个链接库将取决于应用程序所运行的平台。

WINDOWS 平台

对于Windows ODBC应用程序：

如果使用驱动管理器链接ODBC SDK的库***odbc.lib***，***odbc.lib***通常位于*c:\odbcsdk\lib\odbc.lib*路径下。用户必须首先在***odbc.ini***里注册DBMaster驱动管理器才能正确加载DBMaster驱动器。在这种情况下，

用户不需要指定makefile文件里的***dmapi54.lib***，驱动管理器将自动加载所需的DLL。

注意 *您可以不使用驱动管理器来链接由DBMaster提供的dmapi54.lib库，该库通常位于c:\dbmaster\5.4\lib路径下。*

这是一个位于*install_directory\bin*目录下， ODBC程序的动态链接库***dmapi54.dll***，但是程序只需要链接到库***odbc.lib*** (通过**驱动管理器**) 或***dmapi54.lib*** (不通过**驱动管理器**)。当这些库文件都已经链接上后，程序将自动调用该DLL。

UNIX平台

在UNIX平台，当创建一个客户端/服务器端ODBC程序时，用户必须链接***libdmapic.a***文件。

2.3 范例ODBC应用程序

在UINX环境下可以参考以下示例程序。

➲ **示例**

连接到一个数据源后，通过SQLGetInfo找回DBMS版本：

```
#include <stdio.h>


#include "sqlext.h"
#include "sqlopt.h"
#include "sqlunix.h"

#define  STR_LEN 30
HENV     henv;                          /* environment handle         */
HDBC     hdbc;                          /* connection handle          */
HSTMT    hstmt;                         /* statement handle           */
SDWORD   retcode;                       /* return code                */
UCHAR    info[STR LEN];                 /* info string for SQLGetInfo */
SHORT    cbInfoValue;



retcode = SQLAllocEnv(&henv);
retcode = SQLAllocConnect(henv,&hdbc);
retcode = SQLConnect(hdbc, (SQLCHAR *)"TEST", SQL_NTS, (SQLCHAR *)"SYSADM",
                     SQL_NTS, (SQLCHAR *) "",SQL_NTS);


if (retcode != SQL_SUCCESS)
    goto EXIT;
retcode = SQLGetInfo(hdbc,SQL_DBMS_VER, &info, STR_LEN,&cbInfoValue);
```

```
if (retcode != SQL_SUCCESS)
    goto EXIT;
printf("Current DBMS version is %s\n", info);
EXIT:
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
return;
```

2.4 编译和链接

以下示例使用***acc***编译器，其中的***$dir***代表DBMaster目录。

➲ **示例 1**

在UINX命令行中键入如下命令，编译示例程序(***example.c***)：

```
sh> acc -c example.c -I$dir/dbmaster/include
```

现在通过DBMaster库***libdmapis.a***链接示例程序(***example.o***)。

➲ **示例 2**

创建一个执行文件成为example：

```
sh> acc -o example example.o -L$dir/dbmaster/lib -ldmapis
```

2.5 示例程序

附加的ODBC示例程序是在***samples***目录下由DBMaster提供的。用户能够使用目录里的makefile文件来创建和执行它们。首先更改*dbmaster/5.4/samples/odbc*目录，然后输入“make ex1”来为示例程序***ex1.c***创建可执行的**ex1**。

在Windows平台下有一些不同的C语言开发工具，如Microsoft Visual C++、Borland C++和Watcom C++。为了对不同的开发工具使用不同的方法设置包含文件和链接库目录，用户需要编辑您的makefile文件。例如：在Visual C++中用户必须打开一个新的工程编辑***.mak***和***.def***文件。

在*c:/dbmaster/5.4/samples/odbc*路径下，DBMaster为Visual C++提供一个示例makefile文件。

3 数据库连接

在任何一个ODBC应用程序中，用户必须设置合适的ODBC环境并且在执行SQL语法或执行查询前连接一个数据源。同样的，用户必须从数据库断开并且当程序停止时为ODBC环境释放所分配的内存。本章将介绍设置连接数据源所需的函数。

本章用户将学习到以下内容：

- 通过分配环境来初始化ODBC环境并且使用*SQLAllocEnv*和*SQLAllocConnect*函数来连接句柄。
- 使用*SQLConnect*函数建立一个到预定数据源的连接或者使用*SQLDriverConnect*函数建立一个到位置数据源的连接。
- 通过设置*SQLGetConnectOption*和*SQLSetConnectOption*选项来连接数据源。
- 使用*SQLDisconnect*函数断开数据源连接。

  当程序结束时，使用*SQLFreeConnect*和*SQLFreeEnv*函数来释放连接和环境句柄。

注意 *本章将描述用户分配环境并连接句柄，使用DBMaster (ODBC 3.0)获取并设置不同的连接选项，更多信息请参考ODBC3.0函数的第八章。*

以下程序流程图适用于所有使用六个ODBC函数的程序。

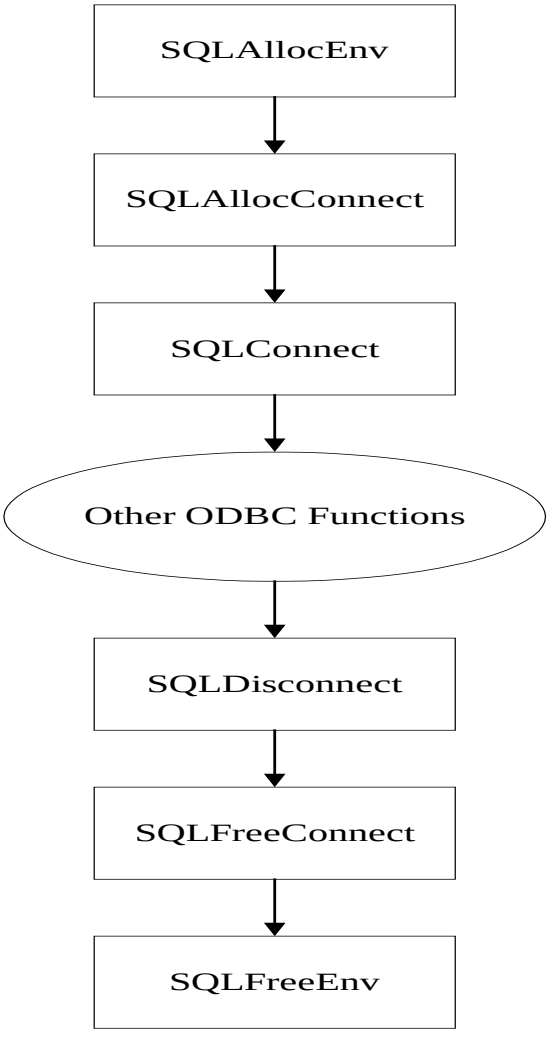

图3-1: 连接和断开一个数据源的程序流程图

3.1 环境句柄

在一个ODBC应用程序里，SQLAllocEnv函数在调用其它ODBC函数之前，被调用来设置ODBC环境。当用户调用SQLAllocEnv函数时，DBMaster驱动器将为环境信息分配内存空间并且返回一个环境句柄到应用程序中。

环境句柄识别内存区域，DBMaster驱动将用于保存关于ODBC环境的全局信息，它可能包括如一系列有效连接句柄和当前有用的连接句柄信息。在应用程序中只可以分配一个环境句柄。

➲ **原型**

SQLAllocEnv：

```
RETCODE SQLAllocEnv(HENV FAR * phenv)
```

➲ **示例 1**

分配一个环境句柄，声明HENV类型的变量：

```
HENV henv1;
```

➲ **示例 2**

调用SQLAllocEnv并传递HENV变量的地址：

```
retcode = SQLAllocEnv(&henv1);
```

环境句柄是目前的一个有效句柄，用户能够在随后的应用程序中使用它。如果应用程序调用SQLAllocEnv得到一个有效的环境句柄，驱动器将覆盖先前的环境句柄内容。

3.2 连接句柄

分配一个环境句柄后，在连接任何一个ODBC数据源之前，一个连接句柄将被分配。它将在一个ODBC程序里为每一个连接识别内存存储并且包括诸如数据库名和用户名此类信息。SQLAllocConnect函数为一个连接句柄分配内存。

➲ **原型**

SQLAllocConnect：

```
RETCODE SQLAllocConnect(HENV henv, HDBC * phdbc)
```

➲ **示例 1**

分配一个连接句柄，声明HDBC类型的变量：

```
HDBC hdbc1;
```

➲ **示例 2**

调用SQLAllocConnect并传递变量的地址：

```
retcode = SQLAllocConnect(henv1, &hdbc1);
```

3.3 连接到一个数据源

在试图访问包含所需要的数据之前连接一个数据源。要连接一个数据源，那么必须先指定该带有有效连接句柄（**hdbc**）的数据源。无论是否通过dmconfig.ini文件，数据源连接都能在客户端被执行。

SQLConnect

使用SQLConnect在一个数据源和一个有效连接句柄之间建立一个连接。

➲ **原型**

SQLConnect：

```
RETCODE SQLConnect(
                    HDBC         hdbc,
                    UCHAR  FAR * szDSN,
                    SWORD        cbDSN,
                    UCHAR  FAR * szUID,
                    SWORD        cbUID,
                    UCHAR  FAR * szAuthStr,
                    SWORD        cbAuthStr);
```

一个应用程序必须通过以下设置来使用SQLConnect：

- 一个有效的连接句柄通常不被连接到另一个数据源
- 数据源名称和该名称的长度
- 用户名和其长度
- 密码和其长度

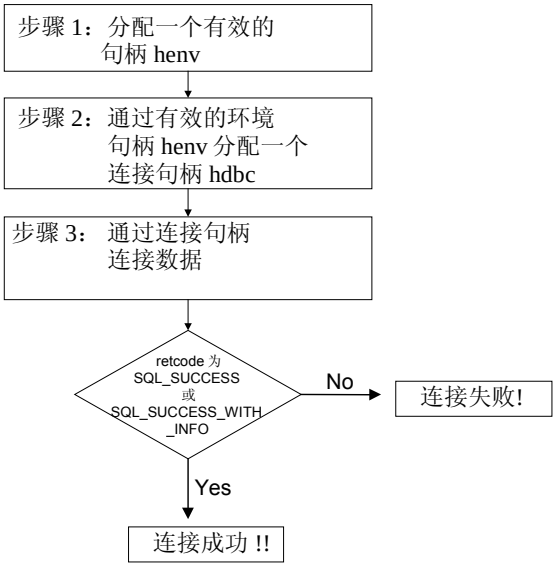

图3-2: 连接一个数据源的流程图

1. 在第一步，**SQLAllocEnv**被调用分配一个环境句柄**henv** 。
2. 在第二步，调用**SQLAllocConnect**来分配一个带有有效环境句柄**henv** 的有效连接句柄**hdbc**。
3. 在第三步，调用**SQLConnect**来连接一个带有有效连接句柄**hdbc**的数据源。
4. 若代码返回的是**SQL_SUCCESS**或**SQL_SUCCESS_WITH_INFO**，说明连接已经被建立。

➲ **示例 1**

通过用户MYNAME和密码PASS来连接数据源TEST_DB：

```
retcode = SQLConnect(hdbc, (SQLCHAR *)"TEST DB", SQL_NTS,
                           (SQLCHAR *)"MYNAME",  SQL_NTS,
                           (SQLCHAR *)"PASS",    SQL_NTS);
```

SQL_NTS意味着字符串是以**NULL**结束，并且驱动器会计算字符串的长度。当使用SQLConnect时，数据源名称是一个必须指定的参数，但是用户名和密码是可以选择的。

在DBMaster中，可将默认的用户名和密码设置于**dmconfig.ini**文件中以省略它们在SQLConnect的连接字符串中的使用，然后驱动器会在**dmconfig.ini**中获得指定的用户名和密码。

➲ **示例 2**

在*dmconfig.ini*中设置DB_UsrID=MYNAME和DB_PasWD=PASS然后调用TEST_DB：

```
retcode = SQLConnect(hdbc, (SQLCHAR *)"TEST_DB", SQL_NTS, (SQLCHAR *)"",
                          SQL_NTS, (SQLCHAR *)"", SQL_NTS);
```

当一个应用程序调用**SQLConnect**时，**驱动管理器**将使用数据源名称(**test_db**)从**ODBC.INI**文件适当的部分读取驱动器DLL的名称，然后加载驱动DLL并传递用户名和密码参数到驱动器。客户端不需要配置文件。

在没有DMCONFIG.INI配置文件的情况下使用SQLCONNECT

DBMaster将提供在客户端无需配置文件即可连接数据库的功能。通过使用SQLConnect()连接字符串和合适的关键字，配置文件指定的设置可以被执行。SQLConnect()的另一个参数具备接受一个特殊连接字符串的能力。

若在SQLConnect()连接字符串中没有关键字被指定并且客户端没有配置文件存在，DBMaster将为每一个关键字使用默认值。

可以使用以下关键字：

- DSN：数据源名称。
- CTIMO：连接超时（请参考dmconfig.ini的DB_CTIMO定义）。
- ATCMT：自动提交打开或关闭（请参考dmconfig.ini的DB_ ATCMT定义）。
- STRSZ：STRING类型数据返回的长度，仅通过UDF使用。
- STROP：此关键字指定在应用字符串串联操作符(||)之前是否删除空格。
- SVADR：远数据源地址。
- PTNUM：端口号。

关键字**DSN**必须位于SQLConnect字符串的第一个位置，**SVADR**和**PTNUM**必须位于第二和第三位置，其它的关键字没有特别规定位置。

以上所提及的SQLConnect原型中，szDSN的最初意思是“数据源名称”，但是现在它被改为**连接字符串**。输入的字符串的格式为“keyword1=value1; keyword2=value2…”。

➲ **示例 1**

```
SQLConnect(hdbc, "DSN=TEST_DB;SVADR=172.0.0.1;PTNUM=12345;",
                                    SQL_NTS, ...);
```

➲ **示例 2**

```
SQLConnect(hdbc, "DSN=DBSAMPLE;SVADR=172.0.0.1;PTNUM=12345;ATCMT=0;
                                  CTIMO=2;", SQL_NTS, ...);
```

SQLDriverConnect

使用SQLDriverConnect连接一个预先指定的数据源，驱动管理器用于显示所有可用的数据源以提供给用户一个数据源连接列表。

➲ **原型**

SQLDriverConnect：

```
RETCODE SQLDriverConnect(
                          HDBC        hdbc,
                          HWND        hwnd,
                          UCHAR  FAR *szConnStrIn,
                          SWORD       cbConnStrIn,
                          UCHAR  FAR *szConnStrOut,
                          SWORD       cbConnStrOutMax,
                          SWORD  FAR *pcbConnStrOut,
                          UWORD       fDriverCompletion);
```

一个应用程序必须通过以下信息执行SQLDriverConnect：

- 一个有效的连接句柄尚未和数据源相关联。
- 一个有效的窗口句柄为对话框提供一个主窗口。
- 输入连接字符串(*szConnStrIn*)和它的长度。连接字符串有自己的特殊语法（请参考连接字符串章节），并且包括连接一个数据源所需的指定信息。如果输入的信息不完整，*SQLDriverConnect*将在发送连接信息到数据库之前弹出一个对话框，提示用户输入更多的信息。
- 输出连接字符串(*szConnStrOut*)和它的长度，这是最终发送至数据库驱动器的连接信息。
- 当提示数据源信息时，提示标记(*fDriverCompletion*)将支配所使用的方针。

注意 *当使用SQLDriverConnect时的连接流程与图5中所描述的大致相同。首先，用户必须分配环境和连接句柄，然后SQLDriverConnect函数可被调用连接一个数据源。*

输入连接串

输入的连接字符串指定所需信息来连接一个数据源。

➲ **原型**

Keyword value pairs：

```
KEYWORD=VALUE;
```

通常用到的关键字：

- *DSN* —数据源名称的名称
- *UID* —用户名
- *PWD* —密码

➲ **示例**

```
DSN=TEST_DB; UID=myname; PWD=abc;

DSN=TEST_DB; UID=myname;

UID=myname;
```

注意 *如果输出连接字符串有多个DSN，UID或PWD，DBMaster将使用第一个。*

在没有DMCONFIG.INI配置文件的情况下使用SQLDRIVERCONNECT

SQLDriverConnect在客户端也可以不通过配置文件而被使用。如果在连接中没有指定关键字，DBMaster将为每一个关键字使用默认值。SQLDriverConnect连接字符串和dmconfig.ini配置文件在客户端是可用的。

如上所述的SQLDriverConnect原型中，参数**szConnStrIn**作为连接字符串被使用，包括新关键字。

➲ **示例**

```
SQLDriverConnect (hdbc, hwin, "DSN=TEST_DB; UID=SYSADM;PWD=x123
                                ;SVADR=172.0.0.1;
                  PTNUM=12345;ATCMT=0;CTIMO=2;", SQL_NTS, .);
```

提示标记

此提示标记显示了无论驱动管理器还是DBMaster驱动，都需要使用一个对话框来从用户处获得连接信息。

➲ **示例**

提示标记的可能值：

```
SQL_DRIVER_PROMPT
SQL_DRIVER_COMPLETE
SQL_DRIVER_COMPLETE_REQUIRED
SQL_DRIVER_NOPROMPT
```

当提示标记值设置为**SQL_DRIVER_COMPLETE**或**SQL_DRIVER_COMPLETE_REQUIRED**时，驱动管理器执行这些操作：

- 如果*DSN*在输入的连接字符串中被指定，它将复制这个连接字符串并且传递字符串到驱动器。

- 如果*DSN*在输入的字符串中没有被指定，驱动管理器将显示数据源对话框以供用户选择一个数据源。
- 驱动管理器构建从对话框返回的数据源名称，并且在输入连接字符串中找到其余的*UID*或*PWD*值。

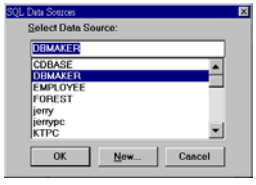

图3-3: 数据源对话框

如果从对话框返回的数据源名称为空，驱动管理器将指定**DSN=Default**（若默认数据源区存在于***ODBC.INI***）。

➲ **DBMaster数据库驱动将根据以下条件执行动作：**

1. 如果输入的连接字符串包含足够的信息(用户ID和密码)，驱动器将连接这个数据源。
2. 如果成功，它将复制输入的连接字符串到输出的连接字符串里。
3. 如果用户ID、密码或两个都丢失，DBMaster驱动将显示一个对话框，允许用户从输入的连接字符串中填写这些丢失的值。
4. 在用户离开对话框之后连接数据源。
5. 在输入的连接字符串中用DSN值创建一个连接字符串并且通过对话框返回此信息，然后在输出连接字符串中输出此连接字符串。

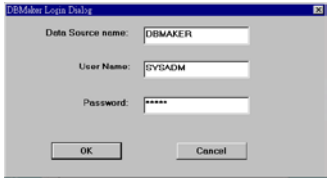

图3-4: UID和PWD对话框

注意 *当提示标记设置为**SQL_DRIVER_PROMPT**时，DBMaster驱动操作与设置**SQL_DRIVER_COMPLETE**或**SQL_DRIVER_COMPLETE_REQUIRED**是相同的。ODBC驱动管理器将会*

弹出一个对话框来提示用户是否为数据源在输入字符串中提供一个DSN。

多重连接

用户可以轻松地在一个应用程序中同时连接多个数据源，但是一些程序需要多次连接到相同的数据库。例如一项任务（程序）可能有两个窗口，每一个窗口都有一个到相同数据库的连接，一个窗口用于扫描表时，另一个窗口可用于更新另一张表。DBMaster中的SQLConnect命令允许程序多次连接到同一个数据源，但是所有连接必须使用相同的用户名，并且所有数据库更改相关联的连接时必须属于同一个活跃事务。

➲ **示例**

通过使用具有相同用户(*user1*, *pass1*)的有效句柄两次连接数据源*DBI*：

```
retcode = SQLAllocConnect(henv, &hdbc1);

retcode = SQLAllocConnect(henv, &hdbc2);


retcode = SQLConnect(hdbc1, (SQLCHAR *)"DB1", SQL_NTS, (SQLCHAR *)"user1",
                     SQL_NTS, (SQLCHAR *)"pass1", SQL_NTS);

retcode = SQLConnect(hdbc2, (SQLCHAR *)"DB1", SQL_NTS, (SQLCHAR *)"user1",
                     SQL_NTS, (SQLCHAR *)"pass1",SQL_NTS);
```

但是，如果用户想尝试用两个用户(*user1, pass1*)和(*user2, pass2*)在一个程序中连接相同的数据源，则会返回错误信息。

在一个程序中多重连接相同的数据源并不是必需的，有很多方法能够解决此类问题，如在一个连接句柄下设多个句柄语法。

3.4 连接选项

一个到数据源的连接有许多属性来控制它的行为。例如**SQL_AUTOCOMMIT**选项决定了是否所有数据库操作都会被自动提交。

SQLSetConnectOption

系统为每一个选项定义了一个默认值，但用户能够通过SQLSetConnectOption来为一个连接指定不同的值。

➲ **原型**

SQLSetConnectOption：

```
RETCODE SQLSetConnectOption (
                                HDBC    hdbc,
                                UWORD   fOption,
                                UDWORD  vParam);
```

其中hdbc是一个有效的连接句柄，fOption是为连接设置的选项，vParam是fOption指定的值。

在DBMaster中，自动提交模式默认为开启状态。

➲ **示例1**

关闭一个连接的自动提交模式：

```
retcode = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT,
                                    SQL_AUTOCOMMIT_OFF);
```

选项值**SQL_AUTOCOMMIT_OFF**是**SQL_AUTOCOMMIT**选项的一个新值。将**SQL_AUTOCOMMIT**设置为关闭，调用一个外部的SQLTransact **(hdbc, COMMIT)**来提交事务的所有更改。

➲ **示例2**

打开自动索引后台程序：

```
retcode = SQLSetConnectOption(hdbc, SQL_LOAD_AUTOINDEX,
```

```
                                        SQL_LOAD_ON);
```

注意 *用户可以在创建一个连接之前或之后，使用SQLSetConnectOption来设置连接选项，这些选项在连接句柄存在时起作用。一旦这些选项被设置，它们将应用于所有与连接相关的语法。但是DBMaster中的选项值SQL_CONNECT_MODE必须在用户创建连接之前被设置。*

SQLGetConnectOption

用户可以使用SQLGetConnectOption来获取一个连接选项的当前值。

➲ **原型**

SQLGetConnectOption：

```
RETCODE SQLGetConnectOption (
                                HDBC    hdbc,
                                UWORD   fOption,
                                PTR     vParam)
```

此处的hdbc是一个有效的连接句柄，fOption是设置用户想重新获取的值，vParam指出获取的选项值的存放位置。

➲ **示例**

在变量commitval中为连接hdbc放置与SQL_AUTOCOMMIT相关联的值：

```
retcode = SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &commitval);
```

3.5 释放句柄

在中断用户的程序之前，用户应该释放位于连接和环境中的所有资源。对于每一个SQLConnect，SQLAllocConnect和SQLAllocEnv函数，与之相应的SQLDisconnect，SQLFreeConnect和SQLFreeEnv函数允许用户释放相应的本地资源。

SQLDisconnect

SQLDisconnect关闭与一个特定连接句柄相关联的连接，并且释放该连接下的所有语句句柄。（语句句柄将在*第四章*进行说明。）

➲ **原型**

SQLDisconnect：

```
RETCODE SQLDisconnect(HDBC hdbc)
```

SQLFreeConnect

在断开连接后，一个程序应该调用SQLFreeConnect来释放连接句柄和所有与句柄相关的内存。如果用户想试图使用SQLFreeConnect来释放一个打开的连接句柄，驱动器将返回错误信息。用户需要在调用SQLFreeConnect.之前关闭连接（通过调用SQLDisconnect）。

➲ **原型**

SQLFreeConnect：

```
RETCODE SQLFreeConnect(HDBC hdbc)
```

SQLFreeEnv

SQLFreeEnv释放环境句柄和所有相关内存。在调用SQLFreeEnv之前，一个程序必须调用SQLFreeConnect来释放henv下分配的任意hdbc。

➲ 原型

SQLFreeEnv：

```
RETCODE SQLFreeEnv(HENV henv)
```

4 SQL语句

本章将详细描述如何使用ODBC函数来执行DBMaster支持的SQL语句。首先介绍SQL查询语言，然后解释下列内容：

- 使用函数*SQLAllocStmt*和*SQLFreeStmt*分配和释放语句句柄。
- 使用函数*SQLExecDirect*直接执行SQL语句，使用函数*SQLPrepare*准备执行语句以及使用函数*SQLExecute*执行准备好的语句。
- 使用函数*SQLRowCount*返回UPDATE、INSERT或DELETE语句影响的记录数。
- 在执行期间而非准备期间，使用参数向SQL命令传递数据值。
- 使用函数*SQLNumParams*返回SQL语句中的参数个数，使用函数*SQLDescribeParam*返回与准备的SQL语句相关联的参数的描述信息。
- 使用函数*SQLBindParameter*在SQL语句中绑定缓存与参数，使用函数*SQLPutData*和*SQLParamData*在小空间内输入大数据项。
- 使用函数*SQLGetStmtOption*返回语句选项的当前设定，使用函数*SQLCancel*取消语句处理。

下图列出了4-3 ~ 4-6的主题，以及当编写应用程序使用ODBC访问数据库时，它们与发生的状态转换之间的关系。

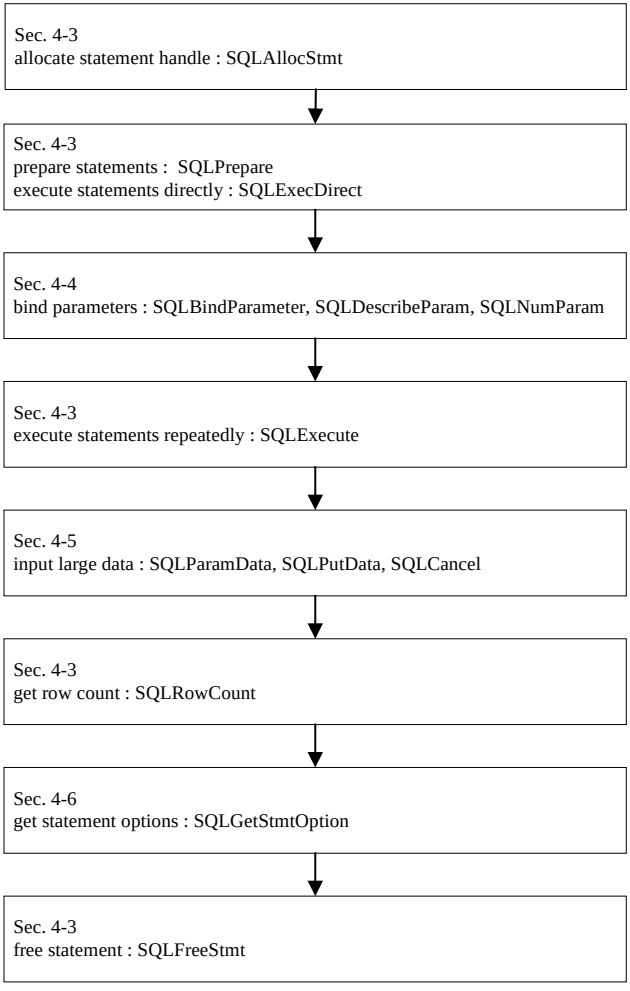

图4-1: 本章章节以及它们的状态转换关系

4.1 SQL语言

结构化查询语言（SQL）是一种行业标准查询语言，用来对存储在关系数据库中的数据进行定义、组织、管理和查询。与C和Pascal等传统的过程语言不同的是，您不必明确定义如何执行数据库操作，您只需使用类似英语的SQL语法简单地向数据库提出一个请求，数据库就会选择最好的方式处理这个请求，并在完成时向您返回结果。

虽然查询仍然是它最重要的功能，但SQL提过的功能远远超过了简单的数据查询。实际上，SQL分为三部分，分别是数据定义语言（DDL）、数据操作语言(DML)*和*数据控制语言 (DCL)。每一部分都有自己特定的作用，将它们合起来一起使用，您就可以执行DBMS提供的所有功能，包括：

- **数据定义 —** 定义数据的结构和组织以及数据间的关系。
- **数据操作 —**返回数据库中的数据，并通过增加新数据、删除旧数据和修改当前数据等更新数据库。
- **数据控制 —**保护数据防止未授权的访问，定义数据完整性以防止冲突发生。

本节只提供SQL的简明概述，有关更多详细信息请参考*SQL命令与函数参考手册*。

SQL在ODBC中的作用

当ODBC驱动器获得一个SQL语句后，它会将该SQL请求传递给数据库引擎。然后数据库引擎会根据SQL语句来构造、存储和查询存储在硬盘上的数据。

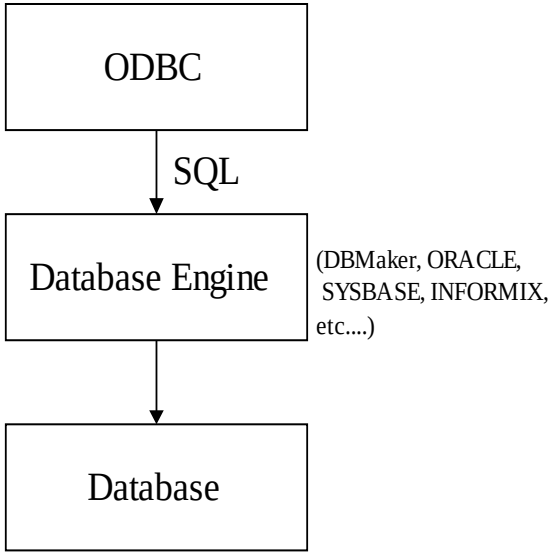

图4-2: 使用ODBC时SQL的作用

基础的SQL语句

SQL语句可以分为*DDL*、*DML*和*DCL*语句。有关DBMaster支持的SQL语言和SQL语法的详细讨论请参考*SQL命令与函数参考手册*。

数据定义语言

数据库模式由一组称为SQL数据定义语言（DDL）的SQL语句来处理，DDL使用CREATE、DROP或ALTER命令来定义、删除或修改数据库对象的定义。这里将简明介绍CREATE TABLE语句。

CREATE TABLE命令

数据库中包含很多表，每个表中都存有信息。表由行（记录）和列（字段）组成。可以使用CREATE TABLE命令在数据库中创建新表。

➲ **示例 1**

CREATE TABLE的基本语法：

```
CREATE TABLE table-name (column-name data-type, ... )
```

ANSI/ISO SQL标准规定了DBMS最少应该支持的数据类型数量。基本上所有的商业SQL产品都支持这些类型或提供具有相等功能的类似数据类型。

➲ **示例 2**

CREATE TABLE命令：

```
CREATE TABLE account (
                      fno       serial,       /* account number         */
                      lname     char(10),     /* account last name      */
                      fname     char(10),     /* account first name     */
                      branch    integer,      /* belong to branch       */
                      balance   integer,      /* account balance        */
                      altno     char(12),
                      stamp     long varbinary, /* account's stamp image  */
                      photo     long varbinary, /* account's photo image  */
                      memo      long varchar    /* account's memo          /
                      );
```

数据操作语言 (DML)

返回或操作数据库中的数据是由一组称为SQL数据操作语言（DML）的SQL语句来处理的。基本的DML语句是SELECT、INSERT、DELETE和UPDATE。

从数据库中返回数据（SELECT）

可以使用SELECT语句从数据库中返回数据或结果集。

➲ **示例 1**

SELECT语句的基本语法：

```
SELECT item_list FROM table_list WHERE search_condition;
```

数据类型	描述
CHAR(len)	Fixed-length character string
VARCHAR(len)	Variable-length character string
BINARY(len)	Binary data
OID	Object ID
FILE	BLOB object (file)
LONG VARCHAR	BLOB object (text)
LONG VARBINARY	BLOB object (binary)
SERIAL [(integer)]	Auto-increment integer
SMALLINT	Small integer number
INTEGER [INT]	Integer number
FLOAT	Low-precision floating point number
DOUBLE	High-precision floating point number
DECIMAL [DEC]	Decimal numbers (use default precision and scale)
DECIMAL(precision,scale)	Default precision is 17 and default scale is 6
DATE	Date
TIME	Time
TIMESTAMP	Timestamp
Other	Domain

图4-3: DBMaster数据类型

基础的SELECT 语句包括三部分：SELECT、FROM和WHERE。每部分的功能如下：

- *SELECT* — 指定查询语句要返回的字段或经过计算的字段。
- *FROM* — 指定*SELECT*列表中包含的项目所在的表。
- *WHERE* — 指定选择记录时必须满足的搜索条件。

WHERE子句可以包含多个搜索条件：

- 比较符 (=, >, <, >=, <=, <>, !=)
- 范围（BETWEEN 和 NOT BETWEEN）
- 列表 (IN和NOT IN)
- 字符串匹配(LIKE和NOT LIKE)
- BLOB匹配（MATCH和NOT MATCH）

- 未知值（IS NULL和IS NOT NULL）
- 逻辑组合 (AND, OR)
- 否定 (NOT)

➲ **示例 2**

执行一个查询查找账户余额大于$10,000的所有用户：

```
select lname, fname, balance from account where balance > 10000
```

修改数据库中的数据

增加、删除或更新记录可以修改数据库中包含的数据。对于每种操作，DBMS都会返回所影响的记录数。

添加数据

INSERT语句用来向表中添加一条新记录。

➲ **示例 1**

INSERT语句的基本语法如下：

```
INSERT INTO table_name(column_names) VALUES value_list
```

INSERT语句由两部分组成：INSERT INTO和VALUES。

每个部分的功能如下：

- *INSERT INTO* — 指定您要插入记录的表。也可选择性的包含一个字段列表来指定数据只能插入到这些字段，列表中未包含的字段会插入**NULL**值。
- *VALUES* — 指定要插入的数据值。您可以使用常数或参数插入值。

如上所述，值的列表中可以包含常数或参数。常数可以是能以文本形式表达的任意数字、文本或日期，如'John'、 'Monday'、 123、54.823等。下面给出一个使用常数的INSERT命令的例子。

➲ **示例 2**

在数据库中为John Smith增加一个新账户：

```
INSERT INTO account (lname,fname,branch,balance)
       VALUES ('john','smith',101,10000)
```

在值的列表中，可以用问号(?)代表参数数据，实际值可以在过后插入。如果在准备时数据值未知或当用户想保存准备时间时，可以使用参数。下面给出一个使用参数的INSERT命令的示例，例中向数据库插入记录，但当前值未知。

➲ **示例 3**

要插入的实际值可以在执行前绑定：

```
INSERT INTO account (lname,fname,branch) VALUES (?,?,?)
```

注意 *要学习如何准备语句和绑定参数，请参考第五章。*

删除数据

DELETE语句将从表中删除一条或多条记录。

➲ **示例 1**

DELETE语句的基本语法：

```
DELETE FROM table_name WHERE search_condition
```

DELETE语句由两部分组成：DELETE FROM和WHERE。

这两部分的功能如下：

- *DELETE FROM* — 指定您要从哪个表中删除记录。
- *WHERE* — 指定删除记录必须满足的条件。

DELETE语句中使用的WHERE子句可包含SELECT语句的WHERE子句中允许使用的任意搜索条件。

➲ **示例 2**

从数据库中删除John Smith的账户：

```
DELETE FROM account where fname = 'john' and lname = 'smith'
```

更新数据

UPDATE语句改变表中的现有记录。

➲ **示例 1**

UPDATE语句的基本语法：

```
UPDATE table_name SET column_names expression WHERE search_condition
```

UPDATE子句由三部分组成：UPDATE、SET 和 WHERE。

各部分的功能如下：

- *UPDATE* — 指定您要更新的记录所在的表。
- *SET* — 指定您要更改的字段和定义每个字段如何更改的表达式。
- *WHERE* — 指定更改字段时必须满足的搜索条件。

UPDATE语句中使用的WHERE子句可包含SELECT语句的WHERE子句中允许使用的任意搜索条件。

➲ **示例 2**

给账户大于1000美元的账户增加6%的利息：

```
UPDATE account SET balance = balance * 1.06 where balance > 1000
```

注意 *要查找多少行被插入，删除或更新了，请参考函数SQLRowCount。*

4.2 执行SQL语句

本节将指导用户如何编写一个简单的ODBC程序。如前面章节提到的，每个SQL语句都可在程序中通过ODBC执行。如下例，假设我们已经成功连接到数据库。

➲ **示例 1**

使用下面的SQL语句建构一个简单的表：

```
CREATE TABLE account (lname char(10), fname char(10), branch integer)

INSERT INTO account VALUES('Mulder', 'Fox', 11240)
```

➲ **示例 2**

相应的ODBC语句将是：

```
retcode = SQLAllocStmt(hdbc, &htmt);

retcode = SQLExecDirect(htmt, (SQLCHAR *)"CREATE TABLE account (lname
                        CHAR(10), fname CHAR(10), branch integer)", SQL_NTS);

retcode = SQLExecDirect(htmt, (SQLCHAR *)"INSERT INTO account
                        VALUES('Mulder', 'Fox', 11240)", SQL_NTS);
```

SQLAllocStmt

所有使用SQL语句的ODBC函数都需要一个语句句柄。语句句柄是一个指向系统控制区（DCCA的一部分）中某一位置的指针，系统控制区中保存着一个SQL语句的所有信息。因此，在通过SQLExecDirect执行SQL语句前，需要使用SQLAllocStmt来分配语句句柄。

➲ **原型**

SQLAllocStmt：

```
RETCODE SQLAllocStmt(HDBC hdbc, HSTMT FAR *phstmt)
```

如果返回的代码是**SQL_SUCCESS**，那么您就从驱动成功分配了一个有效的语句句柄。然后可进行下一个ODBC函数SQLExecDirect了。

SQLExecDirect

SQLExecDirect用于直接执行SQL语句。许多ODBC教材中称之为*直接执行*，以区别于另一种执行方式*准备执行*。准备执行将在后面介绍。

➲ 原型

SQLExecDirect：

```
RETCODE SQLExecDirect(
                        HSTMT     hstmt,
                        UCHAR FAR *szSqlStr,
                        SWORD     cbSqlStr)
```

第一个参数hstmt是一个有效的语句句柄，第二个参数是要执行的SQL语句字符串，最后一个参数是SQL语句的字符串长度或如果szSqlStr指向一个以空字符结尾的字符串时，为**SQL_NTS**。

SQLExecDirect的执行分两个阶段。首先，通过检查引用对象的名称和语法，选择访问计划，将语句转换为内部可执行的形式来编译（准备）SQL语句。然后在第二个阶段，执行这个可执行的形式真正访问数据库。

如果SQL语句是诸如SELECT * FROM account 这样的查询，那么将会产生所选记录的结果集，用户需要使用SQLFetch从结果集（见第五章）中一条一条地返回数据。如果SQL语句是一个INSERT、DELETE或UPDATE语句，可以使用SQLRowCount来查看影响了多少行记录。

SQLRowCount

该函数返回被语句句柄中执行的INSERT、DELETE或UPDATE语句所影响的记录数。

➲ 原型

SQLRowCount：

```
RETCODE SQLRowCount(
                      HSTMT      hstmt,
```

```
                    SDWORD FAR *pcrow)
```

如果hstmt与UPDATE语句一起使用，那么**pcrow**会返回执行UPDATE语句后更新的记录数。

➲ **示例**

使用SQLRowCount：

```
SDWORD count;

SDWORD retcode;

retcode = SQLAllocStmt(hdbc, &hstmt);

retcode = SQLExecDirect(hstmt, (SQLCHAR *)"CREATE TABLE account (lname
                        char(10), fname char(10), branch integer, balance
                        money)",SQL_NTS);

/* insert three records into account table                          */

retcode = SQLExecDirect(hstmt, (SQLCHAR *) "INSERT INTO account
                        VALUES('Mulder','Fox', 11240, 10000.00)", SQL_NTS);

retcode = SQLExecDirect(hstmt, (SQLCHAR *) "INSERT INTO account
                        VALUES('Scully','Dana', 11330, 20000.00)", SQL NTS);

retcode = SQLExecDirect(hstmt, (SQLCHAR *) "INSERT INTO account
                        VALUES('Skinner','Walter', 11240, 30000.00)",
                        SQL_NTS);

/* if branch is 11240, add 1000 to balance                          */

retcode = SQLExecDirect(hstmt, (SQLCHAR *) "UPDATE account SET balance =
                        balance + 1000.00 WHERE branch = 11240", SQL_NTS);

/* get the number of updated rows from count in the example.        */

/* Count will be two.                                               */

retcode = SQLRowCount(hstmt, &count);
```

如果hstmt没有与INSERT、DELETE或UPDATE语句一起使用，那么DBMaster返回的记录数将是-1。

SQLFreeStmt

您可以使用SQLFreeStmt来关闭或删除语句句柄。

➲ **原型**

SQLFreeStmt：

```
RETCODE SQLFreeStmt (
                      HSTMT     hstmt,
                      UWORD     fOption)
```

第一个参数hstmt是一个有效的语句句柄，第二个是可选项，用来指定如何释放语句句柄。两个用于释放语句句柄的常用选项是**SQL_CLOSE** 和**SQL_DROP**。如果语句不是select语句，那么语句句柄可以重新利用。

➲ **示例 1**

一个重新利用的语句句柄：

```
SQLExecDirect(hstmt1, (SQLCHAR *)"INSERT ...");
SQLExecDirect(hstmt1, (SQLCHAR *)"CREATE ...");
SQLExecDirect(hstmt1, (SQLCHAR *)"INSERT ...");
```

如果是一个select语句，在再次使用之前，您需要先关闭该句柄。

➲ **示例 2**

在重新利用之前，先关闭一个select语句：

```
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT * FROM account", SQL_NTS);
...
retcode = SQLFreeStmt(htmt,SQL_CLOSE);
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"INSERT INTO account
                      VALUES('Mulder','Fox', 11240)", SQL_NTS);
```

您可以使用**SQL_CLOSE**关闭一个语句句柄以备后续使用，这样就不必在每次select语句之后执行语句时，都分配一个新的语句句柄。如果您心存怀疑，可以使用**SQL_DROP**选项来删除语句句柄后再分配新的句柄。删除动作会释放所有与句柄相关的资源，删除之后，您就不能再使用该句柄了。

SQLPrepare和SQLExecute

正如在SQLExecDirect一节提到的，准备执行分两个阶段：准备阶段和执行阶段。如果您想重复地执行一条语句，可以使用准备执行来提高效率。

准备执行将语句的执行周期分为两个部分，分别是使用ODBC函数调用进行准备(SQLPrepare)和执行(SQLExecute)。主要思想是只将语句准备为可执行的形式一次，然后可执行多次。

➲ **原型**

SQLPrepare：

```
RETCODE SQLPrepare(
                    HSTMT     hstmt,
                    UCHAR     *szSqlStr,
                    UDWORD     cbSqlStr)
```

➲ **原型**

SQLExecute：

```
RETCODE SQLExecute(HSTMT     hstmt)
```

在SQLPrepare中，hstmt是有效的语句句柄，szSqlStr是要执行的SQL语句字符串。cbSqlStr是字符串szSqlStr的长度或当字符串是以空终止时为**SQL_NTS**，当使用参数时，准备执行将更加有利，如下节所述。

4.3 参数

本节将介绍参数。在ODBC程序中，SQL语句中使用参数用于在执行期间向SQL命令传递数据值。该概念类似于嵌入式SQL中的主变量。

当遇到以下情景，可在SQL语句中使用参数：

- 在准备期间，不知道参数值。
- 应用程序需要使用不同的参数值多次执行同样的SQL语句（例如，一个程序可能需要使用字符串来获得所有输入值，然后将这些值插入到数据库的表中。这样，它可以使用参数标记来接受所有值，然后将这些值转化为相应的字段类型。所以，驱动可以将它们正确地插入到数据库中）。
- 应用程序需要在不同的数据类型间转换参数值。
- 要执行的存储过程带有输出参数。

➲ **示例**

向名为*account*的表中插入5条记录：

```
INSERT INTO account (lname, fname, branch) VALUES (?,?,?)
```

在该语句中，'?'是参数标记。通过使用参数，应用程序只需准备该语句1次，然后使用不同的参数值执行5次。

参数函数

处理参数的函数有3个，分别是SQLBindParameter、SQLDescribeParam和 SQLNumParams。

- ***SQLBindParameter-*** 用来绑定存储单元与参数标记，指定存储单元的数据类型、精度和广度。
- ***SQLNumParams*** –应用程序用来获取SQL语句中的参数个数，在交互式的动态SQL程序中尤其实用。

- ***SQLDescribeParam-*** 用来描述指定参数的属性，像长度或精度等。应用于动态SQL程序。

SQLBINDPARAMETER

➲ **原型**

SQLBindParameter：

```
RETCODE SQLBindParameter(
              HSTMT      hstmt,
              UWORD      ipar,
              SWORD      fParamType,
              SWORD      fCType,
              SWORD      fSqlType,
              UDWORD     cbColDef,
              SWORD      ibScale,
              PTR        rgbValue,
              SDWORD     cbValueMax,
              SDWORD FAR *pcbValue)
```

应用程序需要传递下列信息到SQLBindParameter：

- *hstmt* — 语句句柄
- *ipar* —第i个参数
- *fParamType* — 参数类型，input/output
- *fCType* —参数主语言类型
- *fSQLType* — SQL字段类型
- *cbColDef* — 字段精度
- *ibScale* — 字段的广度
- *rgbValue* — 存储地址

- *cbValueMax* — 存储缓存的长度。当参数类型是output时，使用 *cbValueMax* 。如果返回的数据是在C中有固定长度的类型，那么忽略该值。当参数类型是input时，不使用该参数。
- *pcbValue* —rgbValue中的参数长度

下例说明了在调用SQLExecute之前，是如何使用SQLBindParameter来绑定记录值将具有不同值的多行记录插入到数据库中的。注意：在第三次调用SQLExecute时，通过设定SQLBindParameter的pcbValue为**SQL_NULL_DATA**，字段***branch***将插入**NULL**值。

➲ 示例

使用SQLBindParameter：

```
#define LENGTH 18

UCHAR  lname[LENGTH], fname[LENGTH];

UDWORD branch no;

SDWORD retcode, cblname, cbfname, cbbranch;

retcode = SQLPrepare(hstmt, (SQLCHAR *)"INSERT INTO account
                      (lname,fname,branch) VALUES (?,?,?)",SQL_NTS);

err exit(hstmt, retcode);            /* exit if error               */

cblname = SQL NTS;                   /* null terminated string      */

cbfname = 0;

cgbranch = 0;

retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,

                           SQL CHAR, LENGTH, 0, lname, 0, &cblname);

retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,

                           SQL_CHAR, LENGTH, 0, fname, 0, &cbfname);


retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_LONG,

                           SQL_INTEGER, 0, 0, branch_no, 0, &cbbranch);

   strcpy((char *)lname, "Mulder");

   cblname = strlen((char *)lname);

   strcpy((char *)fname, "Fox");
```

```
cbfname = strlen((char *)fname);
branch no = 11240;
retcode = SQLExecute(hstmt);
strcpy((char *)lname, "Scully");
cblname = strlen((char *)lname);
strcpy((char *)fname, "Dana");
cbfname = strlen((char *)fname);
branch_no = 11251;
retcode = SQLExecute(hstmt);
/* insert a NULL data to branch column for the third customer whose   */
/* branch number is unknown                                           */
strcpy((char *)lname, "Angus");
cblname = strlen((char *)lname);
strcpy((char *)fname, "MacGyver");
cbfname = strlen((char *)fname);
cbbranch = SQL_NULL_DATA;      /* indicate NULL for branch column */
retcode = SQLExecute(hstmt);
```

➲ **设置参数值：**

1. 调用SQLBindParameter绑定存储单元与参数标记。
2. 在存储单元中存入参数值。

注意 *在调用SQLPrepare之前或之后，第一步都可以完成。但必须在调用SQLExecute之前完成。第二步需要在SQLExecute之前完成，因为驱动需要参数值来执行SQL语句。*

SQLBindParameter中fParamType可使用三种参数类型：

- ***SQL_PARAM_INPUT –*** 该参数称为input参数。
- ***SQL_PARAM_INPUT_OUTPUT –*** 该参数可以是input类型，也可为output类型。
- ***SQL_PARAM_OUTPUT –***该参数称为output参数。

如何使用INPUT参数执行SQLBINDPARAMETER

该参数存储在存储单元rgbValue中。当将参数值放入rgbValue中，用户需要使用在SQLBindParameter的fCType参数中指定的C数据类型。

SQLBindParameter 的pcbValue参数只是一个指针，该指针指向一个包含参数长度的缓存，但是缓存还可以用于其它目的。

在调用SQLExecute或SQLExecDirect之前，pcbValue中存储的可能值有：

- 参数的长度，只对字符或二进制C类型有效。
- ***SQL_NTS*** - 指示该参数值是NULL结尾的字符串。
- ***SQL_NULL_DATA*** **-** 指示该参数值是**NULL**，如前面的代码示例中所示。
- ***SQL_DEFAULT_PARAM*** **-** 指示使用了字段的默认值。
- ***SQL_DATA_AT_EXEC*** 或 ***SQL_LEN_DATA_AT_EXEC*** **-** 指示参数值需要由*SQLPutData*来传递。这将在*4.4*节详细介绍。

SQLNUMPARAMETER

➲ **原型**

SQLNumParams：

```
RETCODE SQLNumParams(
                      HSTMT       hstmt,
                      SWORD  FAR *pcpar)
```

当调用了该函数，驱动会将SQL语句中的参数个数放入到缓存pcpar中。如果SQL语句中未包含参数，该值将是0。注意，该函数只能在SQL语句准备好后调用（也就是已经调用了SQLPrepare）。

SQLDESCRIBEPARAM

➲ **原型**

SQLDescribeParam：

```
RETCODE SQLDescribeParam(
                          HSTMT       hstmt,
                          UWORD       ipar,
                          SWORD  FAR *pfSqlType,
                          UDWORD FAR *pcbColDef,
                          SWORD  FAR *pibScale,
                          SWORD  FAR *pfNullable)
```

如果应用程序中包含了SQLBindParameter需要的所有信息，那么可以直接调用SQLBindParameter。然而在调用SQLBindParameter进行设置之前，应用程序可能会缺少参数的详细信息。例如，在动态SQL程序中的图形化查询工具，要执行的SQL语句直到运行时才能决定。这种程序需要同时包含参数的详细信息才能够绑定参数。

SQLDescribeParam返回与准备的SQL语句相关的参数标记的描述。

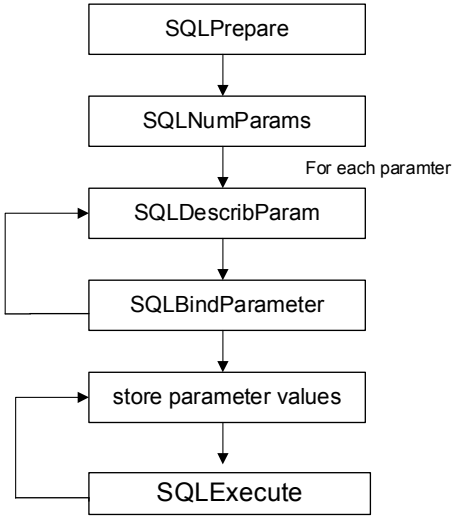

图4-4: 使用input参数时的程序流程

下例说明在动态SQL程序中SQLDescribeParam、SQLNumParams和SQLBindParameter的用法。参数标记序号从左向右排序，初始值为1。

➲ **示例 1**

在动态SQL程序中使用input参数：

```
#define  BUFFER LEN  256        /* length of the SQL string buffer   */
#define  MAX_PARAMS   32        /* allowed max number of parameters  */
UCHAR   str[BUFFER_LEN];
SDWORD  retcode;
SWORD   i, nparam;
SWORD   partype[MAX_PARAMS], parscale[MAX_PARAMS], parnull[MAX_PARAMS];
SWORD   parCtype[MAX_PARAMS];
UDWORD  parlen[MAX PARAMS], outlen[MAX PARAMS];
char    *parbuf[MAX_PARAMS];
BEGIN:                              /* begin label                    */
getSQLString(&str);                 /* get input SQL statement string  */
retcode = SQLPrepare(hstmt,str,SQL NTS);    /* prepare the input SQL string
                        */
err_exit(hstmt, retcode);           /* exit if error                  */
retcode = SQLNumParams(hstmt,&nparam); /* get number of parameters    */
err exit(hstmt, retcode);           /* exit if error                  */
if (nparam > 0)                     /* parameters found in input string*/
   {
    printf("There are %d parameters \n",nparam);
    for (i = 0; i < nparam; i++)  /* describe parameters and set them*/
        {
         retcode = SQLDescribeParam(hstmt, i+1, &partype[i], &parlen[i],
                                    &parscale[i], &parnull[i]);
         err exit(hstmt, retcode);          /* check return code, exit if
                        error*/
```

```
        /* allocate storage location for the parameter according to the
                    */
        /* parameter type, length and scale, reuse the storage if possible
                    */
        allocParamStorage(partype[i], parlen[i], parscale[i], &parbuf[i]);
        /*get C type corresponding to SQL type                    */
        getSQLCtype(partype, &parCtype);
        /* bind the parameter to storage location                  */
        retcode = SQLBindParameter(hstmt, i+1, SQL_PARAM_INPUT,
                                   parCtype[i], partype[i],
                                   parlen[i], parscale[i],
                                   parbuf[i], BUFFER_LEN,
                                   &outlen[i]);
        err_exit(hstmt, retcode);/* check return code, exit if error*/
       }
   for (i = 0; i < nparam; i++)
      {
       /* get parameter values and store them in bound storage
                    */
       getParamValue(nparam, parCtype[i], partype[i], parlen[i],
                     parscale[i],
                    &parnull[i], parbuf[i]);
      }
   }                            /* end of if statement            */
retcode = SQLExecute(hstmt);    /* excute prepared SQL statement  */
err_exit(hstmt, retcode);
if (user_Quit())                /* user wants to quit             */
   return;
else
   goto BEGIN;                  /* go to BEGIN for next SQL string */
```

只有要执行的存储过程带有输出参数时才需要使用输出参数。后面会介绍使用输出参数来执行带有输出参数的存储过程。存储过程是一个用户定义的函数。一旦创建了存储过程，它将以可执行形式作为数据库对象存储在数据库中。在交互式的SQL窗口中，用户可以像执行命令一样执行存储过程，或者在应用程序、触发事件或另一个存储过程中调用存储过程。这里，我们只介绍在ODBC应用程序中如何执行存储过程。有关存储过程的更多信息请参考*数据库管理员手册*。

下例说明了如何调用带有SQL_PARAM_OUTPUT参数类型的SQLBindParameter函数。SQLBindParameter函数准备了一个缓冲，用来存储当带有指定的input城市值，如输入taipei时执行的存储过程的返回值。

➲ **示例 2**

使用输出参数：

```
SDWORD personNumber = 10000;
SDWORD retcode, avlen;
retcode = SQLPrepare(hstmt, (SQLCHAR *)"CALL getNumber('Taipei',?)",
err_exit(hstmt, retcode);         /* exit if error                    */
retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_LONG,
                           SQL_INTEGER, 0, 0, &personNumber, 0,
                           &avlen);
err_exit(hstmt, retcode);         /* exit if error                    */
retcode = SQLExecute(hstmt);      /* excute prepared SQL statement    */
printf("total %ld employees live on Taipei \n", personNumber);
```

如果用户想编写带有输出参数的动态SQL程序，可以使用SQLNumParams、SQLDescribeParam、SQLProcedures和SQLProcedureColumns。使用SQLProcedures可以获得存储在数据源中的程序名称的列表，然后使用SQLProcedureColumns来返回有关程序参数的信息。

在SQLExecDirect中使用参数

如前面阐述过的，当程序需要多次执行一个SQL语句时，它可以先调用SQLPrepare，然后再多次调用SQLExecute，而不是为每次执行都要准备同样的SQL语句。

通过使用SQLExecDirect，参数也可用于只执行一次的SQL语句中。然而，用户无论如何必须在调用SQLExecDirect之前绑定参数并设置参数值。所以，这样就失去了使用SQLPrepare和SQLExecute所带来的所有优势。

除非输入的数据是特殊类型，如BLOB，否则没有必须使用SQLExecDirect。对于BLOB数据，使用**SQL_DATA_AT_EXEC**或**SQL_LEN_DATA_AT_EXEC**选项在执行期间绑定参数。直到语句被执行后，数据值才被设置。

➲ **示例**

使用SQLExecDirect：

```
#define  LENGTH   18
UCHAR   lname[LENGTH], fname[LENGTH];
UDWORD  branch_no;
SDWORD  retcode, cblname, cbfname, cbbranch;
retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,
                           SQL CHAR, LENGTH, 0, lname, 0, &cblname);
retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, LENGTH, 0, fname, 0, &cbfname);
retcode = SQLBindParameter(hstmt, 3, SQL PARAM INPUT, SQL C LONG,
                           SQL INTEGER, 0, 0, branch no, 0, &cbbranch);
strcpy((char *)lname, "Bill");
cblname = strlen((char *)lname);
strcpy((char *)fname, "Skinner");
cblname = strlen((char *)fname);
branch_no = 11243;
```

```
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"INSERT INTO account (lname, fname,
                        branch) VALUES (?,?,?)", SQL NTS);
err_exit(hstmt, retcode);
```

清除绑定的参数

在调用SQLBindParameter绑定一个存储单元后，它可以重复利用。在下例中，三个存储单元被绑定到INSERT语句中的三个参数标记，直到明确被释放。

当程序调用带有**SQL_RESET_PARAMS**或**SQL_DROP选项**的SQLFreeStmt时，存储单元被释放。注意：这三个存储单元属于同一个语句句柄，当调用SQLFreeStmt后，绑定在该句柄中的所有存储单元都被释放。如果程序使用了**SQL_RESET_PARAMS** 选项，它可重置语句句柄而绑定一个不同的存储单元。

➲ **示例**

清除绑定的参数：

```
#define  LENGTH  18
UCHAR  lname [LENGTH], fname[LENGTH];
UDWORD branch_no;
SDWORD retcode, cblname, cbfname, cbbranch;
retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, LENGTH, 0, lname, 0, &cblname);
retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL CHAR, LENGTH, 0, fname, 0, &cbfname);
retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_LONG,
                           SQL_INTEGER, 0, 0, &branch_no, 0, &cbbranch);
... (use the three parameters to execute some SQL commands)
/* reset the parameters */
retcode = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
```

当程序在**SQLFreeStmt**语句中使用**SQL_DROP**选项，句柄将被释放，而成为无效句柄。

4.4 输入大数据

DBMaster提供两种方式来向数据库输入BLOB数据。一种是使用小的固定大小的缓存来读取部分BLOB数据到数据库中。通过多次重复这个动作，整个BLOB就被输入到数据库中而不用使用独占大缓存区。DBMaster还提供了文件对象类型，允许用户将BLOB数据存储在外部文件中。

如何输入大数据

当需要向long varchar或long varbinary字段输入大量的数据时，可以使用函数SQLPutData和SQLParamData将数据分成较小的块来存储。因此，除非数据需要一次全部输入，否则并不是所有的数据都需要大的缓存区。

➲ **原型**

SQLParamData：

```
RETCODE SQLParamData(
                      HSTMT      hstmt,
                      PTR   FAR  *prgbValue)
```

➲ **原型**

SQLPutData：

```
RETCODE SQLPutData(
                    HSTMT   hstmt,
                    PTR     rgbValue,
                    SDWORD  cbValue)
```

SQLParamData用来检查参数是否需要数据。然后使用SQLPutData输入数据。这个过程持续进行，直到SQL语句中的所有参数的数据都被输入。

➲ **向数据库中输入大对象：**

1. 绑定参数 — 设置SQLBindParameter函数的pcbValue参数为SQL_DATA_AT_EXEC或SQL_LEN_DATA_AT_EXEC。这使ODBC驱动知道您要在执行期间使用SQLPutData为该参数赋值。
2. 执行SQL命令— 用SQLExecDirect或SQLExecute执行SQL语句。当执行期间有参数需要获得数据，将会返回SQL_NEED_DATA。
3. 找到执行期间需要获得数据的第一个参数 — 调用SQLParamData来指示执行期间需要获得数据的第一个参数开始接收数据。
4. 调用 SQLPutData — 准备缓存区的第二个数据段，调用SQLPutData将其传递给数据库中正在等待数据的参数，重复这个步骤，直到该参数的所有数据都已传递。
5. 调用 SQLParamData — 如果返回的代码是SQL_NEED_DATA，执行期间需要获得数据的下一个参数 准备接收数据，用户需要回到第4步。如果返回的代码是SQL_SUCCESS 或SQL_SUCCESS_WITH_INFO，执行期间需要获得数据的所有参数的数据都已传递完成，SQL语句执行完毕。

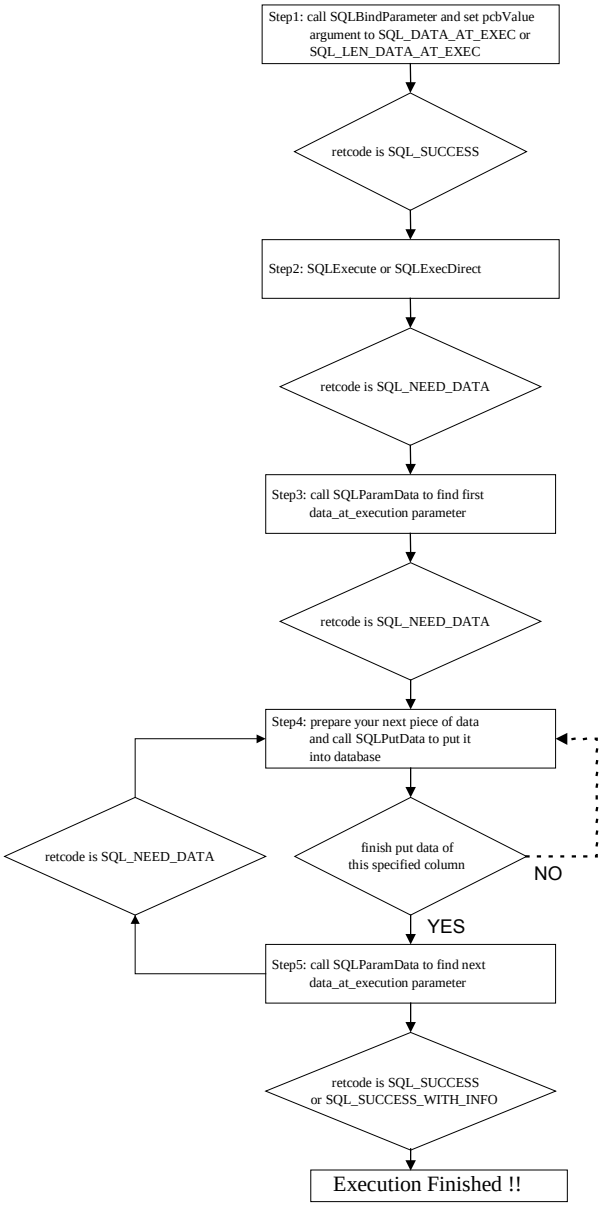

图4-5:输入大数据的操作流程

在下面的例子中，向表***account***中插入一条记录。**InitUserData()**打开用户的数据文件，文件中包含用户的***photograph***、***signature*** 和***memo***字段，这些字段都要输入到表***account***中。**GetUserData()**从用户数据文件中返回下一个**MAX_DATA_SIZE**数据到数据缓存InputData中，直到所有的数据都被读取。

➲ 示例

使用SQLParamData和SQLPutData插入数据：

```
#define  MAX_DATA_SIZE  2048


SDWORD  cbPhoto, cbStamp, cbMemo;
SDWORD  DataLen;
PTR     pParm, DataFile;
UCHAR   InputData[MAX_DATA_SIZE];
SDWORD  retcode;
retcode = SQLPrepare(hstmt, (UCHAR *)"INSERT INTO account
                            VALUES('Mark','Greene',2, 40000.00,
                    'xxxx', ?, ?, ?)",SQL_NTS);
if (retcode == SQL_SUCCESS)
   {
    /* Bind the parameters. Set cbPhoto, cbStamp and cbMemo with     */
    /* SQL_DATA_AT_EXEC to let ODBC know that values of these parameters */
    /*  will be provided at statement execution time      */
    cbPhoto = cbStamp = cbMemo = SQL DATA AT EXEC;
    retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
                               SQL_LONGVARBINARY, 0, 0, NULL, 0,
                               &cbPhoto);
    retcode = SQLBindParameter(hstmt, 2, SQL PARAM INPUT, SQL C BINARY,
                               SQL_LONGVARBINARY, 0, 0, NULL, 0,
                               &cbStamp);
    retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR,
```

```
                                SQL_LONGVARCHAR, 0, 0, NULL, 0,
                                &cbMemo);
    retcode = SQLExecute(hstmt);
    if (retcode == SQL_NEED_DATA)  /* any large data ? */
        {
         Parm = 0;
         while ((retcode = SQLParamData(hstmt, &pAddr)) == SQL_NEED_DATA)
                {          /* need to put large data for this column */
                 Parm++;   /* in a loop to get data and put data in blob */
                 InitUserData(Parm, DataFile);
                 while (GetUserData(Parm, DataFile, InputData, &DataLen))
                        retcode = SQLPutData(hstmt, InputData, DataLen);
                }
         if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
             printf("insert a record of account table success !! \n");
        }
   }
```

取消执行SQLPutData

如果在输入数据过程中有错误发生，或者不想继续输入数据，可以通过函数SQLCancel取消该过程。

➲ **原型**

SQLCancel：

```
RETCODE SQLCancel(HSTMT hstmt)
```

调用SQLCancel，整个语句将被中止。取消当前执行的语句后，你可以再次调用SQLExecute或SQLExecDirect。

在文件对象中存放大数据

文件对象是DBMaster支持的功能强大的大对象数据类型。它与**LONG VARCHAR**或**LONG VARBINARY**数据类似，只是以外部对象的形式存放在文件系统上。定义字段的数据类型为**FILE**便可创建一个FO字段。

➲ **示例 1**

定义*photograph*字段为FO字段：

```
create table student (name char(20), photograph file)
```

因为FO被当做BLOB数据对待，所以您可以使用BLOB的插入方法向FO字段插入数据。通过设置fSQLType为**SQL_LONGVARCHAR**或**SQL_LONGVARBINARY**，DBMaster会为每个FO字段创建一个新文件，并从输入缓存(当 fCType为**SQL_C_CHAR**)或客户端(当fCType为**SQL_C_FILE**)复制数据。

可能用户不想创建另一个文件，而是将FO字段链接到服务器端现有的文件，如在CD-ROM上的文件。如果想链接到服务器端的文件，设置fCType为**SQL_C_CHAR**，fSqlType为**SQL_FILE**。使用该方法后，在调用SQLExecute之前复制文件名到缓存中。

FSQL类型	**FO**字段	**FC**类型	数据源
SQL_LONGVARCHAR or SQL_LONGVARBINARY (Same as BLOB)	在服务器端新建一个文件	SQL_C_FILE	从客户端复制数据。
		SQL_C_CHAR SQL_C_BINARY	从输入缓存复制数据。
SQL_FILE	链接到服务器端存在的文件。	SQL_CHAR	从输入缓存获得文件名。

在下面的例子中，为名为***Mary***的用户插入了一条新记录，同时她的照片存储在FO中，文件已经存在于服务器端，所以可以链接该文件到数据库中。

➲ **示例 2**

在文件对象中放置大数据：

```
UCHAR  pPhotoFlName[80];

SDWORD retcode;

SDWORD cbPhoto;

retcode = SQLPrepare(hstmt, (SQLCHAR *)"INSERT INTO student
                      VALUES('Mary', ?)", SQL_NTS);

cbPhoto = SQL_NTS;

retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,

                          SQL_FILE, 80, 0, pPhotoFlName, 0, &cbPhoto);

strcpy((char *)pPhotoFlName, "/disk1/sys/fo/photo");       /* pass the file
                      name */


retcode = SQLExecute(hstmt);
```

因为FO以外部文件存储，编辑该文件的程序仍然可以直接对文件进行操作。

4.5 Get和Set选项

在一个语句句柄中，一条语句的当前设定可以使用SQLGetStmtOption函数获得。

➲ **原型**

SQLGetStmtOption：

```
RETCODE SQLGetStmtOption(
                          HSTMT  hstmt,
                          UWORD  fOption,
                          PTR    pvParam);
```

➲ **原型**

SQLSetStmtOption：

```
RETCODE SQLSetStmtOption(
                                    HSTMT   hstmt,
                                    WORD    fOption,
                                    UDWORD  vParm);
```

hstmt是一个有效语句句柄，fOption是要返回的选项，pvParam是与fOption相关的值。根据fOption的值，pvParam将返回一个32位的整型值或一个指向以NULL结尾字符串的指针。

DBMaster提供了一些可用于这两个函数中的选项。

选项	描述
SQL_GET_INCREMENT_VALUE	获取SERIAL字段值。
SQL_GET_CURRENT_OID	获取最新插入/获取的元组的OID。
SQL_GET_BACKUP_ID	获取备份ID。
SQL_DUMP_PLAN	获取dump计划选项。
SQL_PLAN	获取查询指向的计划。
SQL_PLAN_LEN	获取计划长度。

图4-6 同SQLGetStmtOption一起使用的扩展语句选项

选项	描述	允许值
SQL_DUMP_PLAN	设置dump计划选项	SQL_DUMP_PLAN_ON SQL_DUMP_PLAN_OFF

图4-7 同SQLSetStmtOption一起使用的扩展语句选项

在下例中，表***account***包含**SERIAL**类型字段。**SERIAL**字段的值不需要明确地指定，它是可以自动增加的。然后在插入一条记录后，用户可能想知道刚刚插入的记录值。可以通过调用SQLGetStmtOption并设置选项fOption为**SQL_GET_INCREMENT_VALUE**来获得。

➲ **示例 1**

使用SQLGetStmtOption获取SERIAL字段值：

```
/* insert a record into table account where the value of each field     */
/* is its default value                                                 */
SDWORD val;
SDWORD retcode;
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"INSERT INTO ACCOUNT VALUES ()",
                        SQL_NTS);
/* get the serial number that was just inserted                         */
retcode = SQLGetStmtOption(hstmt, SQL_GET_INCREMENT_VALUE, &val);
```

在本例中，**val**是前面INSERT语句中刚刚插入到表account中的**SERIAL**字段值。有关**SERIAL**定义和使用的详细信息请参考*SQL命令与函数参考手册*。

SQLGetStmtOption的另一个特殊用法是获取刚插入/返回元组的OID。继续前面的例子，可以提交带有fOption **SQL_GET_CURRENT_OID** 的SQLGetStmtOption来获取刚刚插入数据库中的记录的OID。

➲ **示例 2**

使用SQLGetStmtOption获取当前对象的OID：

```
UCHAR  oid[17];
SDWORD retcode;
```

```
/* insert a record into account table */
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"INSERT INTO ACCOUNT VALUES ()",
                        SQL_NTS);
/* get the OID of the record just inserted */
retcode = SQLGetStmtOption(hstmt, SQL_GET_CURRENT_OID, &oid);
```

OID是DBMaster中对象的唯一ID。您可以使用OID来唯一指定一个数据库对象。

➲ **示例 3**

在查询的WHERE子句总是有OID：

```
SELECT * FROM account WHERE OID = ?
```

注意 *有关OID数据类型的更多信息，请参考SQL命令与函数参考手册或数据库管理员手册。*

扩展语句选项**SQL_DUMP_PLAN**, **SQL_PLAN_LEN**和 **SQL_PLAN** 用来获取DBMaster查询分析器为准备的SQL语句产生的查询计划。

➲ 获取准备的SQL语句的查询计划：

1. 通过调用SQLSetStmtOption开启SQL_DUMP_PLAN选项。
2. 通过调用带有fOption SQL_PLAN_LEN 的SQLGetStmtOption获取计划字符串的长度。
3. 根据计划字符串分配缓存，然后调用带有fOption SQL_PLAN 的SQLGetStmtOption来获取计划字符串。

➲ **示例 4**

使用SQLSetStmtOption和SQLGetStmtOption获取SQL语句的查询计划：

```
SDWORD planlen;
UCHAR  *planstr;
SDWORD retcode;
/* turn on the dump plan option                                          */
retcode = SQLSetStmtOption(hstmt, SQL_DUMP_PLAN, SQL_DUMP_PLAN_ON);
```

```
/* prepare a SQL JOIN statement                                          */

retcode = SQLPrepare(hstmt, (SQLCHAR *)"SELECT * FROM account, branch WHERE
                     account.branchId = branch.branchId", SQL_NTS);

/* get the plan string length                                            */

retcode = SQLGetStmtOption(hstmt, SQL_PLAN_LEN, &planlen);

/* allocate a buffer for the plan string according to the plan           */

/* string length                                                         */

planstr = (UCHAR *)malloc(planlen);

/* get the plan string                                                   */

retcode = SQLGetStmtOption(hstmt, SQL_PLAN, planstr);
```

5 返回结果

执行查询来返回数据是数据库的最重要的功能之一。

本章讲述如何执行下列操作：

- 使用函数*SQLBindCol* 和*SQLFetch*通过绑定字段到存储单元来一行一行地返回数据。
- 使用函数*SQLNumResultCols*、*SQLDescribeCol*和*SQLColAttributes*获取结果集中的字段信息（如类型和长度）。
- 使用游标执行在查询语句返回结果集的固定位置*DELETE*和*UPDATE* 。同时，还有使用函数*SQLGetCursorName*和*SQLSetCursorName*获取或设置游标名称。
- 使用函数*SQLGetData*一块一块地返回大数据对象***(LONG VARCHAR*或*LONG VARBINARY)***和文件对象。一块一块地返回大数据对象和文件对象允许您使用较小的缓存，而不是使用一次返回数据所需的较大的缓存空间。
- 通过函数*SQLExtendedFetch*和*SQLSetPos*，使用rowsets在结果集中前后浏览。

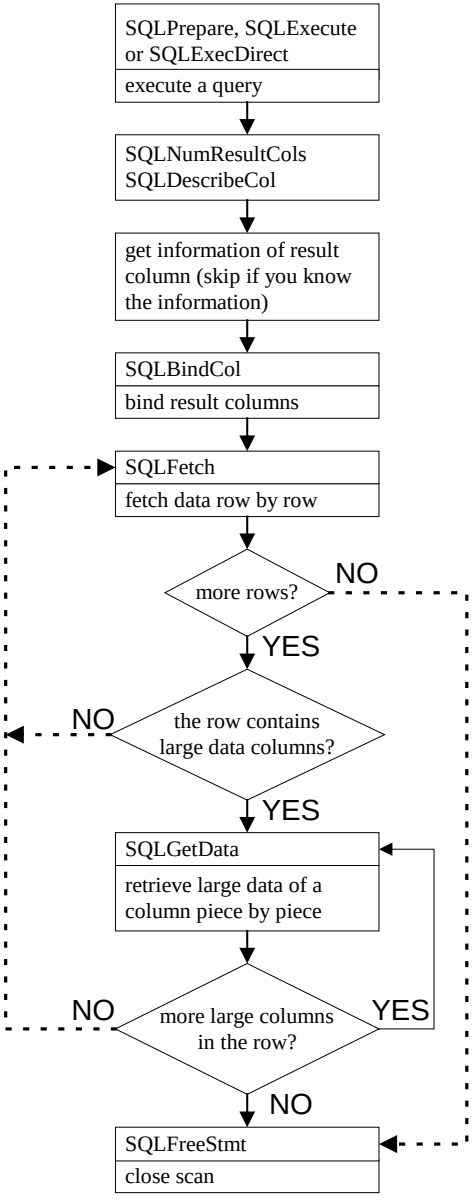

图5-1 从数据库中返回数据的程序流程

5.1 使用ODBC查询

当应用程序需要从数据库中返回数据时，最常用的方法就是使用SQL SELECT语句执行查询。本节将介绍如何使用ODBC函数执行查询，一行一行地返回数据。

绑定存储单元和获取数据

假设我们要从表***account***中取得branch为11240的用户的last name、first name和branch信息。

➲ **示例**

Query查询：

```
SELECT lname, fname, branch FROM account WHERE branch = 11240
```

在准备和执行了这个查询后，我们就准备一行一行地获取数据。如果该查询的所有映射字段的信息(如字段类型、精度等)都已经清楚，那么可以使用SQLBindCol和SQLFetch来获取结果。

- ***SQLBindCol -*** 用来将存储单元和一列数据联系起来。*SELECT*中的*SQLBindCol* 的作用类似于*INSERT*语句中的*SQLBindParameter*。
- ***SQLFetch -*** 用来从结果集中获取一行数据。驱动返回所有绑定字段的值到*SQLBindCol*指定的存储单元中。

➲ **原型**

SQLBindCol：

```
RETCODE SQLBindCol(
                    HSTMT       hstmt,
                    UWORD       icol,
                    SWORD       fCType,
                    PTR         rgbValue,
                    SDWORD      cbValueMax,
                    SDWORD FAR *pcbValue)
```

应用程序要在存储单元和结果字段中建立联系，需要传递下列信息到SQLBindCol。

- *fCType* — 数据即将被转换为的数据类型。
- *rgbValue* — 数据的输出缓存地址。应用程序必须分配这个缓存并确保该缓存足够大，能够容纳返回的指定数据类型。
- *cbValueMax* — 输出缓存的长度。如果返回的数据是在C类型中有固定长度的类型，如整型值，那么该值可以忽略。
- *pcbValue* — 用来返回可用数据字节数的存储缓存地址。注意：如果获取的数据是***NULL***，那么驱动会向该参数存储***SQL_NULL_DATA*** 。

映射中的每个字段都被绑定后，调用SQLFetch**()**来获取一行记录。

➲ **原型**

SQLFetch：

```
RETCODE SQLFetch(HSTMT hstmt)
```

下例使用ODBC执行前面使用SQLBindCol和SQLFetch执行的查询。

➲ **示例**

获取表***account***中branch为11240的所有用户信息：

```
#define  LENGTH  18


UCHAR       lname[LENGTH], fname[LENGTH];

UDWORD      branch no;

SDWORD      retcode, cblname, cbfname, cbbranch;

retcode = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT lname, fname, branch FROM
                        account WHERE branch = 11240", SQL NTS);

if (retcode == SQL SUCCESS)

   {

    retcode = SQLBindCol(hstmt,1, SQL_C_CHAR, lname, LENGTH, &cblname);

    retcode = SQLBindCol(hstmt,2, SQL_C_CHAR, fname, LENGTH, &cbfname);
```

```
    retcode = SQLBindCol(hstmt,3, SQL_C_LONG, &branch_no, 0, &cbbranch);
   }
/* fetch data one row at a time and print out the result data        */
/* stop when no more data or error returned from SQLFetch            */
while (TRUE)
   {
    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        {
         if (cblname == SQL NULL DATA)            /* check null data */
            printf("last name: NULL\n");
         else
            printf("last name: %s\n", lname);
         if (cbfname == SQL NULL DATA)
            printf("first name: NULL\n");
         else
            printf("first name: %s\n", fname);
         if (cbbranch == SQL NULL DATA)
            printf("branch no: NULL\n");
         else
            printf("branch no: %d\n", branch no);
        }
    else                        /* if no more data or errors returned */
         break;
   }
```

结果字段的属性

一些程序事先可能不知道要插入的数据类型。同样，在动态SQL中，程序也可能无法提前预知获取的结果数据。如果这样的话，函数SQLNumResultCols和SQLDescribeCol可帮助提供更多的信息。

SQLNumResultCols用来获取结果集中结果字段的数量。
SQLDescribeCol用来描述结果字段的属性，包括名称、SQL类型、精度、宽度以及是否允许**NULL**值。

➲ **原型**

SQLNumResultCols：

```
RETCODE SQLNumResultCols(
                          HSTMT       hstmt,
                          SWORD  FAR *pccol)
```

SQLNumResultCols的参数描述：

Hstmt：StatementHandle句柄。

Pccol：返回列数。

➲ **原型**

SQLDescribeCol：

```
RETCODE SQLDescribeCol(
                        HSTMT       hstmt,
                        UWORD       icol,
                        UCHAR  FAR *szColName,
                        SWORD       cbColNameMax,
                        SWORD  FAR *pcbColName,
                        SWORD  FAR *pfSqlType,
                        UDWORD FAR *pcbColDef,
                        SWORD  FAR *pibScale,
                        SWORD  FAR *pfNullable)
```

SQLDescribeCol的参数的描述：

hstmt：StatementHandle句柄。

icol：需要得到的列序号，从1开始计算。

szColName：得到列的名称。

cbColNameMax：指明ColumnName参数的最大长度。

pcbColName：返回列名称的长度。

pfSqlType：得到列的ODBC数据类型。

pcbColDef：得到列的长度。

pibScale：当该列为数字类型时返回小数点后的数据位数。

pfNullable：指明该列是否允许为空。

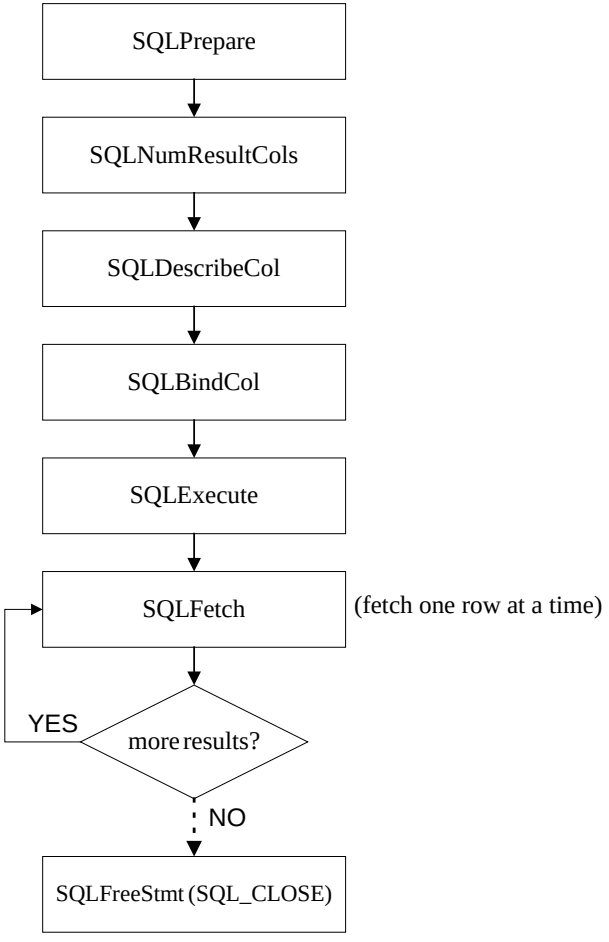

图5-2 获取一个结果集的程序流程

在准备完一个查询后，用户可以调用SQLNumResultCols来找出查询中有多少个结果字段。然后调用SQLDescribeCol获取每个字段需要多少内存，读取的内存数将再用于调用SQLBindCol。最后通过调用SQLFetch一次获取一个结果记录。

下面的例子展示了在执行一个查询后，使用SQLNumResultCols、SQLDescribeCol、SQLBindCol和SQLFetch来获取结果集的过程。

➲ 示例

执行一个查询后获取结果集：

```
#define  BUFFER LEN  256       /* length of the query string buffer*/
#define  MAX_COLS    128       /* allowed max number of columns */

UCHAR       str[BUFFER LEN];
SDWORD      retcode, TRUE = 1;
SWORD       i, ncol;
SWORD       coltype[MAX_COLS], colscale[MAX_COLS], colnull[MAX_COLS];
SWORD       colCtype[MAX COLS];
UDWORD      collen[MAX_COLS], outlen[MAX_COLS];
char       *colbuf[MAX_COLS];
char szColName[MAX_COLS][BUFFER_LEN];      // Column name
SWORD nLen[MAX COLS];          // Column name length
SWORD rgbValue[MAX_COLS];     //

BEGIN:                          /* begin label                  */

getQueryString(&str);           /* get user input query string   */

/* prepare the input query                                              */
retcode = SQLPrepare(hstmt,(SQLCHAR *)str, SQL NTS);

err_exit(hstmt, retcode);       /* exit if error                */

retcode = SQLNumResultCols(hstmt,&ncol); /* get number of result columns */
err_exit(hstmt, retcode);       /* exit if error                */
```

```
if (ncol > 0)                  /* still columns in input query  */
   {
    printf("There are %d result columns \n",ncol);

    for (i = 0; i < ncol; i++)    /* describe columns and bind them */
        {
        retcode = SQLDescribeCol(hstmt, i+1, szColName[i], BUFFER_LEN,
                      &nLen[i], &coltype[i], &collen[i],
                                &colscale[i], &colnull[i]);
        err_exit(hstmt, retcode); /* exit if error                  */

        /* allocate storage location for column according to its,   */
        /* type, length and scale, reuse the storage if possible    */
        allocColumnStorage(coltype[i], collen[i], colscale[i], &colbuf[i]);

        /* get corresponding SQL C type                             */
        getSQLCtype(coltype, &colCtype);

        /* bind the column storage                                  */
        retcode = SQLBindCol(hstmt, i+1, colCtype[i], &rgbValue[i],
                             BUFFER_LEN, &outlen[i]);
        err_exit(hstmt, retcode);/* exit if error                   */
       }                         /* end of for                      */
  }                              /* end of if                       */

retcode = SQLExecute(hstmt);     /* execute the prepared query      */

err exit(hstmt, retcode);        /* exit if error                   */
```

```
/* fetch one row at a time until no more data is in the result set,*/
/* if the column data is null, add a mark in the column buffer, then output
                        */
/* all the column data (print to file or standard output)           */

while(TRUE)
     {
      retcode = SQLFetch(hstmt);

      if (retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO)
         {
          for (i = 0; i < ncol; i++)
              {/* if data is NULL, specify NULL in column buffer    */
               if (outlen[i] == SQL NULL DATA)
                   MarkNullColumn(colbuf[i]);
               /* output column data                                */
               outputColumnData(ncol, colCtype[i], coltype[i],
                                      collen[i],   colscale[i],
                                      colnull[i],  colbuf[i]);
              }
         }
      else
         break;
      }

retcode = SQLFreeStmt(hstmt, SQL_CLOSE); /* close the cursor associated  */
                                         /* in the statement hstmt  */

err exit(hstmt, retcode);                /* exit if error           */
```

```
if (user_Quit())                          /* user wants to quit      */
   return;
else
   goto BEGIN;                            /* go to BEGIN for next query   */
```

结果字段的更多信息

尽管通过调用SQLDescribeCol可以获取字段的一些属性，然而程序可能还需了解更多字段信息。ODBC为此提供了函数SQLColAttributes。

SQLCOLATTRIBUTES

SQLColAttributes用户返回一个字段的描述信息，这个信息用于指定的描述类型。

➲ **原型**

SQLColAttributes：

```
RETCODE SQLColAttributes(
                          HSTMT       hstmt,
                          UWORD       icol,
                          UWORD       fDescType,
                          PTR         rgbDesc,
                          SWORD       cbDescMax,
                          SWORD  FAR *pcbDesc,
                          SDWORD FAR *pfDesc)
```

ODBC中包括的描述符类型：

- SQL_COLUMN_COUNT
- SQL_COLUMN_NAME
- SQL_COLUMN_TYPE
- SQL_COLUMN_LENGTH
- SQL_COLUMN_PRECISION

- SQL_COLUMN_SCALE
- SQL_COLUMN_DISPLAY_SIZE
- SQL_COLUMN_NULLABLE
- SQL_COLUMN_UNSIGNED
- SQL_COLUMN_MONEY
- SQL_COLUMN_UPDATABLE
- SQL_COLUMN_AUTO_INCREMENT
- SQL_COLUMN_CASE_SENSITIVE
- SQL_COLUMN_SEARCHABLE
- SQL_COLUMN_TYPE_NAME
- SQL_COLUMN_TABLE_NAME
- SQL_COLUMN_OWNER_NAME
- SQL_COLUMN_QUALIFIER_NAME
- SQL_COLUMN_LABEL

注意 *有关每个选项具体意义的详细信息，请参考微软ODBC程序员手册。*

例如，如果fDescType的值是**SQL_COLUMN_TYPE**，SQLColAttributes会返回指定字段的SQL类型。虽然这样的信息也可通过SQLDescribeCol获得，但SQLColAttributes可提供SQLDescribeCol不能提供的其它功能。

SQLColAttributes与SQLDescribeCol的主要区别：

- *SQLDescribeCol* 一次提供一个字段的指定信息，而*SQLColAttributes*只能得到一个描述符的值。
- *SQLColAttributes* 提供更确切、更详细的字段信息。如果驱动添加了更多的驱动描述符或在未来版本中ODBC定义了新的描述符，它还能够扩展。

例如，程序想要知道一个字段是否是大小写敏感，它可以使用SQLColAttributes函数并设置描述符的选项为**SQL_COLUMN_CASE_SENSITIVE**来查找。

➲ **示例**

SQLColAttributes获得字段的详细信息：

```
#define  TRUE   1
#define  FALSE  0
SDWORD   CSflag;              /* case-sensitive flag                */
SDWORD   retcode;

retcode = SQLExecDirectSQLPrepare(hstmt, (SQLCHAR *)"SELECT lname, fname,
                                  branch FROM account WHERE branch = 11240",
                                  SQL_NTS);

retcode = SQLColAttributes(hstmt, 1, SQL COLUMN CASE SENSITIVE,
                           NULL, 0, NULL, &CSflag);
if (CSflag == TRUE)
    printf("Column 1 is case-sensitive\n");
```

清除绑定字段

在调用SQLBindCol将存储单元绑定到字段后，它还可以重复利用。在下面的例子中，三个存储单元被绑定到SELECT语句中的三个字段中。

➲ **示例**

调用带有选项**SQL_UNBIND**的SQLFreeStmt来释放语句句柄绑定的所有字段：

```
Retcode = SQLFreeStmt(hstmt, SQL_UNBIND);
```

现在，hstmt中的所有绑定存储单元都被释放了，程序可以重新利用这个语句句柄再绑定其它不同的存储区域。如果程序想释放某个绑定字段，可调用SQLBindCol然后传递一个**NULL**指针到rgbValue参数。

如果程序不需要重新使用语句句柄，那么可以调用带有**SQL_DROP**选项的SQLFreeStmt。这样的话，所有存储单元，以及现有的游标，未决的结果和语句句柄使用的所有资源都将被释放。

5.2 游标

使用游标工具，用户可以在结果集中根据条件逐行进行处理。在给定的结果集中的某个单行上，程序可以执行多个操作，通过查询的执行在结果集上开启游标。

何时使用游标

当程序需要在结果集的给定行上执行更新或删除操作时，可以使用游标。例如，程序可能需要从结果集返回一些记录，然后显示在屏幕上，然后响应用户的更新或删除数据请求。

如果用户要使用游标更新数据，那么用于产生结果集的SELECT语句中必须明确指出FOR UPDATE或FOR UPDATE OF column_list (例如，SELECT * FROM account FOR UPDATE)。如果语句没有声明FOR UPDATE，那么默认的游标类型是只读游标，不允许执行游标更新或游标删除。

➲ **示例 1**

使用游标的UPDATE语句：

```
UPDATE tablename SET column = value [, column = value...]
       WHERE CURRENT OF cursorname
```

➲ **示例 2**

使用游标的DELETE语句：

```
DELETE FROM tablename WHERE CURRENT OF cursorname
```

获取游标名称

当调用SQLAllocStmt分配语句句柄时，ODBC驱动会自动产生一个以**SQL_CUR**开头的名称。可以使用SQLGetCursorName获取与给定的语句句柄相关联的游标全名。

➲ 原型

SQLGetCursorName：

```
RETCODE SQLGetCursorName(
                          HSTMT      hstmt,
                          UCHAR FAR *szCursor,
                          SWORD      cbCursorMax,
                          SWORD FAR *pcbCursor)
```

使用游标

下面显示了如果在定位更新中使用SQLGetCursorName，定位更新包括SELECT和UPDATE语句的两个不同的hstmt。

➲ 示例

在定位更新中使用游标更新John Smith的branch号：

```
#define  NAME_LEN    21
#define  CURSOR_LEN  20
HSTMT   hstmtSelect;
HSTMT   hstmtUpdate;
UCHAR   szLname[NAME_LEN],szFname[NAME_LEN],cursorName[CURSOR_LEN];
SWORD   cursorLen;
SDWORD  sBranch, cbName, cbBranch;
SQLLEN  cbLname,cbFname;
CHAR updsql[256];
/* Allocate the statement handles                                          */
retcode = SQLAllocStmt(hdbc, &hstmtSelect);
retcode = SQLAllocStmt(hdbc, &hstmtUpdate);
/* SELECT the result set and bind its columns to local storage             */
/* NOTE:   This is a select FOR UPDATE                                     */
retcode = SQLExecDirect(hstmtSelect, (SQLCHAR *)"SELECT lname,fname, branch
FROM account FOR UPDATE", SQL_NTS);
```

```
retcode = SQLBindCol(hstmtSelect, 1, SQL_C_CHAR, szLname, NAME_LEN,
                     &cbLname);
retcode = SQLBindCol(hstmtSelect, 2, SQL_C_CHAR, szFname, NAME_LEN,
                     &cbFname);
retcode = SQLBindCol(hstmtSelect, 3, SQL C LONG, &sBranch, 0,
                     &cbBranch);
/* get the cursor name of the select for use in the update statement     */
retcode = SQLGetCursorName(hstmtSelect, cursorName, CURSOR_LEN,
                           &cursorLen);
/* Read through the result set until the cursor is positioned on the row */
/* for John Smith                                                        */
do
   retcode = SQLFetch(hstmtSelect);
while ((retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO) &&
       (strcmp((char *)szFname, "John") != 0 && strcmp((char *)szLname,
                    "Smith") != 0
       && sBranch == 2100));
/* Perform a positioned update of John Smith's branch number             */
/* NOTE: the cursorName in the CURRENT OF clause                         */
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
   {
         sprintf((char *)updsql, "UPDATE account SET branch = 2101 WHERE
                      CURRENT OF %s", cursorName);
     retcode = SQLExecDirect(hstmtUpdate, updsql, SQL_NTS);


    }
```

设置游标名称

用户可以使用SQLSetCursorName设置一个活跃语句句柄的游标名称。在执行SELECT语句之前，必须使用SQLSetCursorName改变游标名称。

➲ **原型**

SQLSetCursorName：

```
RETCODE SQLSetCursorName(
                            HSTMT      hstmt,
                            UCHAR FAR *szCursor,
                            SWORD      cbCursor)
```

5.3 获取大数据

正如前面所描述的，获得字段数据的最常规方式是使用SQLBindCol绑定一个本地缓存。在SQLFetch期间，字段数据自动存储在绑定的缓存中。如果用户确保有足够缓存空间，可以使用绑定的方法来返回大数据。

另一种方式是使用SQLGetData一次获得足够大的一个缓存。如果使用SQLGetData，就不能绑定字段，否则SQLFetch会自动将字段数据传递到绑定的缓存中。

要关联结果字段与存储单元，程序需要传递下列信息到SQLGetData：

- *fcType* — 数据即将被转换为的数据类型。
- *rgbValue* — 数据的输出缓存地址。应用程序必须分配这个缓存并确保该缓存足够大，能够容纳返回的指定类型的数据。
- *cbValueMax* — 输出缓存的长度。如果返回的数据是在C类型中有固定长度的类型，如整型，那么该值可以忽略。
- *pcbValue* — 在当前调用*SQLGetData*之前，用来返回可用数据的字节数的存储缓存的地址。

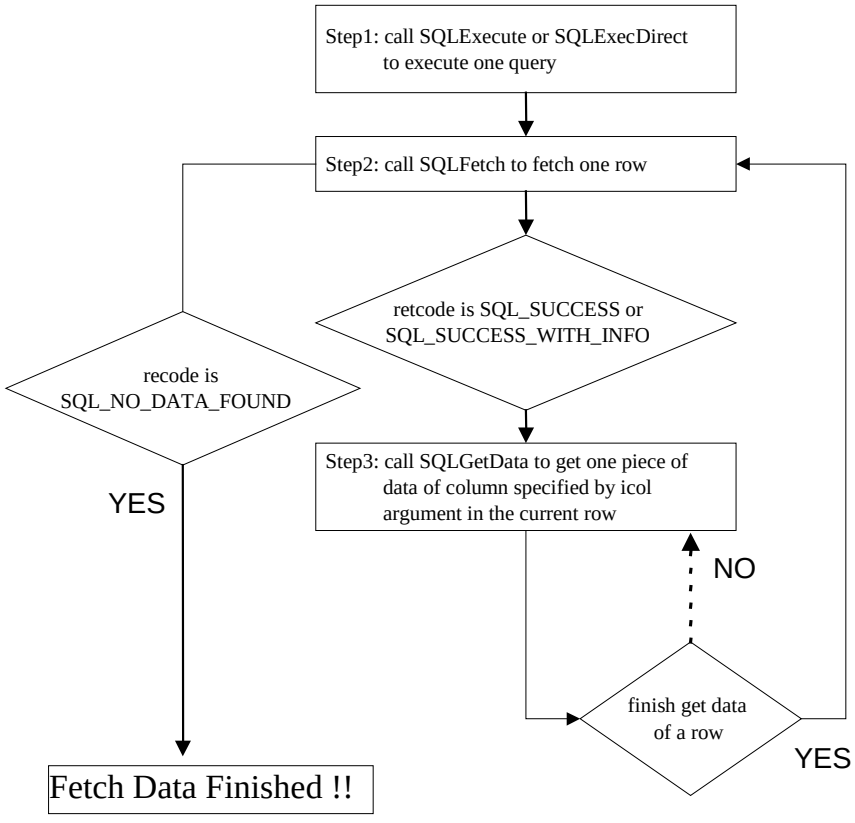

图5-3 返回大数据的程序流程

SQLGetData

当要从**LONG VARCHAR**或**LONG VARBINARY**字段返回大数据时，可以使用函数SQLGetData来一块一块地返回数据。以这种方式，就不必为返回整个字段而准备较大的缓存。

➲ **原型**

SQLGetData：

```
RETCODE SQLGetData(
                    HSTMT hstmt,
                    UWORD icol,
                    SWORD fCType,
                    PTR rgbValue,
                    SDWORD cbValueMax,
                    SDWORD FAR * pcbValue)
```

➲ **从数据库中返回大数据对象：**

1. 执行SQL命令— 用SQLExecDirect或SQLExecute执行SQL查询。
2. 获取数据 — 调用SQLFetch获取下一行数据。如果SQLFetch返回的是SQL_NO_DATA_FOUND，那么查询结果集中的所有记录都已返回。如果返回的代码是SQL_SUCCESS或SQL_SUCCESS_WITH_INFO，说明还有大数据需要返回，请到下一步。
3. 获取大数据对象 — 调用SQLGetData获取当前行中icol参数指定的未绑定字段的一块数据。重复这步，直到SQLGetData返回SQL_NO_DATA_FOUND。如果想从下一行获取数据，请回到上一步。

在下面的例子中，我们从表***account***中的所有记录中返回字段：***fname***、***photo***和***memo***用来显示它们，字段***photo***和***memo***包含大对象。使用绑定方法获取***fname***字段的值，SQLGetData方法获取字段***photo***和***memo***的值。

正如以前所述，用户即可绑定字段与存储单元，又可以使用SQLGetData返回数据。这对于所有数据都实用，如果您愿意，还可以对常规数据类型如整型用SQLGetData。但是由于这包含了不必要的额外程序，所以并不实用。

➲ 示例

```
#define  MAX BINARY SIZE  1024
#define  MAX CHAR SIZE     256
#define  MAX_NAME_SIZE      21

SDWORD cbFname, cbPhoto, cbMemo, DataLen;
UCHAR  FnameBuf[MAX NAME SIZE], PhotoBuf[MAX BINARY SIZE],
       MemoBuf[MAX_CHAR_SIZE];
PTR    PhotoDataFile, MemoDataFile;
SDWORD retcode, TRUE=1;
retcode = SQLExecDirect(hstmt, (UCHAR *)"SELECT fname, photo, memo
                                         FROM account", SQL_NTS);
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, FnameBuf, MAX_NAME_SIZE,
                     &cbFname);
while (TRUE)
      {
        retcode = SQLFetch(hstmt);

        /* After calling SQLFetch, the value of bound column fname is    */
        /* automatically stored in user buffer FnameBuf             */
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
           {
            /* InitDataFile() opens user's data files for storing       */
            /* photo and memo column data                               */
            InitDataFile(PhotoDataFile);
            InitDataFile(MemoDataFile);
```

```
/* The size of remaining data before this SQLGetData is      */
/* returned in cbPhoto. This SQLGetData will retrieve        */
/* MAX_BINARY_SIZE data of column photo from the database,   */
/* and put it into binary buffer PhotoBuf.                   */
while(TRUE)
     {
      retcode = SQLGetData(hstmt, 2, SQL_C_BINARY,
                           PhotoBuf,
      MAX BINARY SIZE, &cbPhoto);
      if (retcode == SQL SUCCESS
          || retcode == SQL_SUCCESS_WITH_INFO)
          {
           /* GetToFile() moves data from buffer PhotoBuf    */
           /* to user file PhotoDataFile                     */
           GetToFile(PhotoDataFile, PhotoBuf);
           printf("%ld more bytes remains in Photo column \n",
                  cbPhoto - MAX BINARY SIZE);
          }
      else
           break;
      }
/* Use SQLGetData to get memo data and put in MemoDataFile   */
while (TRUE)
      {
       retcode = SQLGetData(hstmt, 3, SQL C CHAR, MemoBuf,
       MAX_CHAR_SIZE, &cbMemo);
       if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WITH_INFO)
          {
           GetToFile(MemoDataFile, MemoBuf);
```

```
                    }
                 else
                      break;
                }

        /* Display data on screen                                    */
        Display(FnameBuf);
        DisplayLargeData(PhotoDataFile);
        DisplayLargeData(MemoDataFile);
       }
  else if (retcode == SQL_NO_DATA_FOUND)
       {
        printf("no data found \n");
        break;
       }
  else
      {
       printf("error \n");
       break;
      }
  }
```

对于大部分程序，在显示之前，通常将大字段的所有数据返回和存储到临时文件中。对于必须一次全部显示的静态数据是这样的，对于视频、音频或数据页等的显示可以不需要临时文件。使用双缓存模式同时返回和显示数据，如下图所示。

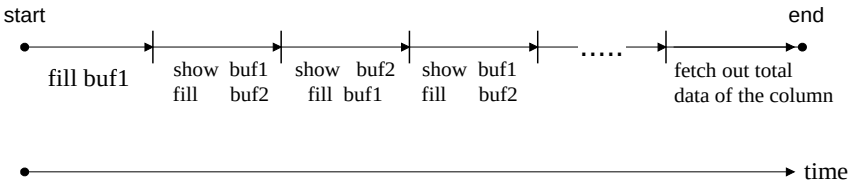

图5-4 用双缓存获取大数据

停止SQLGetData操作

如果从数据库返回数据的过程中有错误发生，或者不想继续返回数据，可以设置函数SQLFreeStmt的**SQL_CLOSE**选项停止该返回过程。还可以调用带有**SQL_CLOSE**选项的SQLFreeStmt函数关闭游标和取消所有未决的结果。然后通过在同样的查询中再次调用SQLExecute或SQLExecDirect来重新开启游标返回数据。

绑定字段以返回文件对象

如果要返回大数据对象并把它放到客户文件中，可以使用绑定文件的方法来实现。这个方法中，设置SQLBindCol的参数fCType为**SQL_C_FILE**，将文件名放置于缓存中。这会指示DBMaster创建一个文件存放大对象数据。

下面的例子中使用该方法返回一个照片，并通过在SQLBindCol的参数fCType中绑定**SQL_C_FILE**将照片放置到客户文件中，然后在缓存中准备文件名。在调用SQLFetch之后，照片被复制到文件中。

➲ **示例**

```
UCHAR  pPhotoFlName[80];
SDWORD retcode;
SQLLEN cbPhoto;
retcode = SQLBindCol(hstmt, 1, SQL_C_FILE, pPhotoFlName, 80, &cbPhoto);
strcpy((char *)pPhotoFlName, "/disk1/usr/fo/photo");
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT photograph FROM student
                                WHERE name = 'mary'", SQL_NTS);
```

```
retcode = SQLFetch(hstmt);  /* a new file is created and data copied */
```

获取文件对象的文件名

如第4章所描述的，文件对象在服务器端以外部文件的形式存储。用户可以使用上述3个方法中的任意一种来返回FO数据，但绑定客户文件(**SQL_C_FILE**)的方法通常在客户端创建一个新文件。如果只对FO的文件名有兴趣，那么可以使用内嵌函数**FILENAME()**来获得文件名。

➲ 示例

使用**SQL_C_CHAR**绑定字段：

```
UCHAR  pPhotoFlName[80];
SDWORD retcode;
SQLLEN cbPhoto;
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, pPhotoFlName, 80, &cbPhoto);
retcode = SQLExecDirect(hstmt, (SQLCHAR *)"SELECT FILENAME(photograph) FROM \
                                 student WHERE name = 'mary'", SQL_NTS);
retcode = SQLFetch(hstmt);   /* file name of FO goes to pPhotoFlName */
```

目前，有很多多媒体工具可以处理以操作系统文件存储的多媒体数据。如果多媒体数据存储在字段**LONG VARCHAR**或**LONG VARBINARY**，您需要先从DBMaster中获取数据，然后转给多媒体工具可以接收的文件。如果存储为文件对象，您只需从DBMaster获得文件名称然后传递给工具即可。

5.4 处理结果集

应用程序使用SELECT语句来查询底层的数据库。除前面章节讲过的SQLFetch函数外，DBMaster还提供了SQLExtendedFetch，能够在SELECT命令返回的结果集中轻松地进行前后浏览，除此之外，还提供了SQLSetPos来进一步修改SQLExtendedFetch的结果集。

Rowsets

*rowset*就像结果集的一个窗口，可以通过它来浏览结果集的详细信息。通常*rowset*是结果集的子集，与结果集有同样的元组顺序。

用户可以使用函数SQLExtendedFetch获取一个rowset。然而在调用SQLExtendedFetch之前，必须先分配一个缓存来绑定字段，然后设置想要取回的元组数。

可以使用不同的参数调用SQLExtendedFetch将rowset窗口前后移动到结果集中的任意位置。例如，如果rowset大小是10，选项SQL_FETCH_FIRST移动窗口到结果集的开头处，然后读取前10条元组到rowset。应用程序通过调用SQLSetStmtOption函数并设置SQL_ROWSET_SIZE选项来负责设置rowset大小，SQL_ROWSET_SIZE的默认值是***1***。在调用SQLExtendedFetch之前，应用程序有责任使用SQLBindCol分配足够大的缓存空间来绑定字段。

程序流程

除了要为rowset分配缓存，程序流程与SQLFetch相似。在调用SQLExtendedFetch期间，可以改变rowset 大小，但是必须确保字段输出缓存和字段状态数组足够大。必须再次调用SQLBindCol重新绑定任何新分配的字段输出缓存和字段状态数组。唯一的区别是：SQLExtendedFetch的数组参数**rgfRowStatus**用来在rowset中记录行的状态，它的大小与rowset大小一致，只能通过重新调用SQLExtendedFetch来解除绑定。

存储单元绑定

可以使用函数SQLBindCol来为从结果集中获取的数据绑定输出缓存(**rgbValue**)和字段状态(**pcbValue**)。因为可获取的元组数可以是rowset大小的任意值，所以必须参考rowset大小为输出缓存和字段状态分配足够的空间。否则，函数SQLExtendedFetch可能调用失败，而将输出数据放置于错误的地址中。

➲ **原型**

SQLBindCol：

```
RETCODE SQLBindCol(
                    HSTMT       hstmt,
                    UWORD       icol,
                    SWORD       fCType,
                    PTR         rgbValue,
                    SDWORD      cbValueMax,
                    SDWORD FAR *pcbValue)
```

有两种方式为包含大于1个元组的rowset绑定输出缓存和字段状态：*以列方式*绑定和*以行方式*绑定。

以列方式绑定

使用SQLSetStmtOption设置SQL_BIND_TYPE为BIND_BY_COLUMN来指定以列方式绑定。如果正在使用列方式绑定，那么rowset中所有元组的相同字段的缓存将是连续的。也就是说，用户一次为一个列分配足够大的缓存空间。例如，下面的一段代码绑定了表中的两个字段(int和char(5))。

注意 *如果使用以列方式绑定，那么SQLFetch是SQLExtendedFetch的一个特例。*

➲ 示例

```
#define ROWSET SIZE   6                    /* rowset size(6 tuples) */
#define NAME_LEN     30                    /* length of NAME column */
#define AGE_LEN       4                    /* length of AGE column  */
SDWORD  retcode;
char    *c1 rgbValue, *c2 rgbValue;
SDWORD  *c1_pcbValue, *c2_pcbValue;
UWORD   *rgfRowStatus;
UDWORD   crow;
int      irow;
/* set rowset size and exit if there is an error                          */
retcode = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);
err exit(hstmt, retcode);
/* set the binding type and exit if there is an error                     */
retcode = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, SQL_BIND_BY_COLUMN);
err_exit(hstmt, retcode);
/* allocate buffer for the column data (rowset) and column status arrays */
c1 rgbValue = (char *)malloc(ROWSET SIZE*NAME LEN);     /* c1 data   */
c1_pcbValue = (SDWORD *)malloc(ROWSET_SIZE*sizeof(SDWORD)); /* c1 status */
c2_rgbValue = (char *)malloc(ROWSET_SIZE*AGE_LEN);      /* c2 data   */
c2 pcbValue = (SDWORD *)malloc(ROWSET SIZE*sizeof(SDWORD)); /* c2 status */
/* allocate row status array for the rowset                               */
rgfRowStatus = (UWORD *)malloc(ROWSET_SIZE*sizeof(UWORD));
/* prepare the input query and exit if there is an error                  */
retcode = SQLPrepare(hstmt, (SQLCHAR *)"select NAME, AGE from employee",
                     SQL_NTS);
err_exit(hstmt, retcode);
/* bind the columns and exit if there is an error                         */
retcode = SQLBindCol(hstmt, 1, SQL C CHAR, c1 rgbValue, NAME LEN,
                    c1_pcbValue);
```

```
err_exit(hstmt, retcode);
retcode = SQLBindCol(hstmt, 2, SQL C CHAR, c2 rgbValue, AGE LEN,
                     c2_pcbValue);
err_exit(hstmt, retcode);
/* execute the prepared query and exit if there is an error             */
retcode = SQLExecute(hstmt);
err_exit(hstmt, retcode);
/* fetch 6 tuples at once until there is no more data in the result set  */
while(TRUE)
    {
     retcode = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &crow,
                                rgfRowStatus);
     if (retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO)
         {
          printf("**** %d fetched rows in rowset ****\n", crow);
          /* print tuples in rowset                                    */
          for (irow=0; irow<ROWSET SIZE; irow++)
              {
               printf("row %d of rowset - ",irow+1);
               switch(rgfRowStatus[irow])
                   {
                    case SQL_ROW_SUCCESS:
                    printf("(NAME: %s) , ",c1_rgbValue + irow * NAME_LEN);
                    printf("(AGE : %s)   ",c2_rgbValue + irow * AGE_LEN);
                    printf("[SUCCESS] \n");
                    break;
                    case SQL_ROW_NOROW:
                    printf(" [NO ROW] \n");
                    break;
                    case SQL_ROW_ERROR:
```

```
                    printf("[ROW ERROR] \n");
                    break;
                   }
               }
          }
           else
                break;
    }
/* close cursor associated with hstmt and exit if there is an error      */
retcode = SQLFreeStmt(hstmt, SQL_CLOSE);
err_exit(hstmt, retcode);
```

以行方式绑定

如果使用SQLSetStmtOption，设置SQL_BIND_TYPE为BIND_BY_COLUMN外的其它值，那么缓存将一行一行被绑定。这样的话，该值将作为一个元组的必要输出缓存的长度。缓存值是所有列的输出值和状态值。如果字段是已知的，应用程序通常会定义一个结构来包括所有字段的输出缓存和状态缓存。例如，下面的代码段为带有两个字段(int和char(5))的表绑定字段。

➲ **示例**

```
#define ROWSET_SIZE   6                    /* rowset size(6 tuples) */
#define NAME_LEN     30                    /* length of NAME column */
#define AGE_LEN       4                    /* length of AGE column  */
SDWORD  retcode;
char    *c1_rgbValue, *c2_rgbValue, *tup_rgbValue, *tup_prn;
SDWORD  *c1_pcbValue, *c2_pcbValue;
UWORD   *rgfRowStatus;
UDWORD   crow;
int      irow, tup_len;
/* set the rowset size and exit if there is an error                     */
```

```
retcode = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);
err exit(hstmt, retcode);
/*calculate the length of one row                                          */
tup_len = NAME_LEN + sizeof(SDWORD) + AGE_LEN + sizeof(SDWORD);
/* set the binding type to row-wise and exit if there is an error          */
retcode = SQLSetStmtOption(hstmt, SQL BIND TYPE, tup len);
err_exit(hstmt, retcode);
/* allocate buffer for the column data(rowset) and column status arrays    */
tup rgbValue = (char *)malloc(ROWSET SIZE*tup len);
/* allocate row status array for rowset                                    */
rgfRowStatus = (UWORD *)malloc(ROWSET_SIZE*sizeof(UWORD));
/* prepare the input query and exit if there is an error                   */
retcode = SQLPrepare(hstmt, (SQLCHAR *)"select NAME, AGE from employee",
                     SQL NTS);
err_exit(hstmt, retcode);
/* bind the columns and exit if there is an error                          */
c1 rgbValue = tup rgbValue;
c1_pcbValue = (SDWORD *)(c1_rgbValue + NAME_LEN);
c2_rgbValue = c1_rgbValue + NAME_LEN + sizeof(SDWORD);
c2_pcbValue = (SDWORD *)(c2_rgbValue + AGE_LEN);
retcode = SQLBindCol(hstmt, 1, SQL C CHAR, c1 rgbValue, NAME LEN,
                     c1_pcbValue);
err_exit(hstmt, retcode);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, c2_rgbValue, AGE_LEN,
                     c2 pcbValue);
err_exit(hstmt, retcode);
/* execute the prepared query and exit if there is an error                */
retcode = SQLExecute(hstmt);
err exit(hstmt, retcode);
/* fetch 6 tuples at once until there is no more data in the result set    */
```

```
while(TRUE)
    {
     retcode = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &crow,
                                rgfRowStatus);
     if (retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO)
         {
          tup_prn = tup_rgbValue;
          printf("**** %d fetched rows in rowset ****\n", crow);
          /* print tuples in rowset                                    */
          for (irow=0; irow<ROWSET SIZE; irow++)
              {
               printf("row %d of rowset - ",irow+1);
               switch(rgfRowStatus[irow])
                   {
                    case SQL_ROW_SUCCESS:
                    printf("(NAME: %s) , ",tup_prn);
                    printf("(AGE : %s)   ",tup prn + NAME LEN +
                       sizeof(SDWORD));
                    printf("[SUCCESS] \n");
                    break;
                    case SQL ROW NOROW:
                    printf(" [NO ROW] \n");
                    break;
                    case SQL_ROW_ERROR:
                    printf("[ROW ERROR] \n");
                    break;
                   }
               /* print next tuple */
               tup prn = tup prn + tup len;
              }
```

```
        }
         else
              break;
    }
/* close cursor associated with hstmt and exit if there is an error      */
retcode = SQLFreeStmt(hstmt, SQL CLOSE);
err_exit(hstmt, retcode);
```

定位游标

如果游标存在，SQLExtendedFetch会将游标定位在rowset的第一行。可通过下列方式使用SQLExtendedFetch：

- 从另一个语句句柄定位*UPDATE* 和*DELETE* 语句。用户可以调用SQL*ExtendedFetch*将游标定位于一行上，然后使用定位*DELETE*语句从目标表中的结果集中删除记录。例如，*DELETE … WHERE CURRENT OF …*。
- *SQLGetData*。可以调用SQL*GetData*从那些未绑定的字段中获取数据。在调用*SQLGetData*之前，rowset大小需要设置为1。
- *SQLSetPos*带有*SQL_DELETE*、*SQL_REFRESH*、*SQL_UPDATE*选项。

在第一次调用SQLExtendedFetch之前，游标位于结果集的开始处，显示为未定义的。使用不同的选项可以将游标放置于结果集的开始处或结果集的结尾来代替当前位置。

SQLExtendedFetch参数

➲ **原型**

SQLExtendedFetch：

```
RETCODE SQLExtendedFetch(
                          HSTMT         hstmt,
                          UWORD         fFetchType,
```

```
                        SDWORD        irow,
                        UDWORD FAR   *pcrow,
                        UWORD  FAR   *rgfRowStatus)
```

SQLExtendedFetch可能的返回值有：

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE

下表显示了ODBC应用程序的不同请求的rowset和返回的代码。在使用rowset缓存中的内容之前，ODBC程序需要检查SQLExtendedFetch返回的每一行的代码和记录状态。

请求的Rowset	返回代码	游标位置	返回的Rowset
在结果集开始之前	sql_no_data_found	在结果集开始之前	无。Rowset缓存的内容未定义。
与结果集的起始处重叠	sql_success	Rowset的第一行	结果集中的第一个rowset。
结果集中	sql_success	Rowset的第一行	请求的rowset。
与结果集的结尾处重叠	sql_success	Rowset的第一行	Rowser中的行与结果集重叠，数据返回。Rowser中的行与结果集未重叠，行状态(rgfRowStatus)为SQL_ROW_NOROW并且rowset缓存的内容未定义。
在结果集的结尾处	sql_no_data_found	Rowset的结尾	无。Rowset缓存的内容未定义。

重叠的情形（第二个和第四个）不是对称的。例如，假设一个结果集有100行，rowset大小为5。下表显示了当获取类型是SQL_FETCH_RELATIVE时，针对不同的**irow**值，SQLExtendedFetch返回的rowset和代码。

当前rowset	Irow	返回代码	新的Rowset
1 to 5	-5	SQL_NO_DATA_FOUND	无
1 to 5	-3	SQL_SUCCESS	1- 5
96 to 100	5	SQL_NO_DATA_FOUND	无
96 to 100	3	SQL_SUCCESS	99和100.对于rowset中的第3.4.5行，相应的行状态都被设置为SQL_ROW_NOROW。

fFetchType参数

fFetchType参数用于为rowset定义决定窗口位置（在结果集中）的类型。

fFetchType的有效值如下：

- SQL_FETCH_FIRST
- SQL_FETCH_LAST
- SQL_FETCH_NEXT
- SQL_FETCH_PRIOR
- SQL_FETCH_ABSOLUTE
- SQL_FETCH_RELATIVE
- SQL_FETCH_BOOKMARK

当参数**fFetchType**使用值SQL_FETCH_ABSOLUTE或SQL_FETCH_RELATIVE时，会应用到参数**irow**。因为并不是相对于当前rowset获取的，所以SQL_FETCH_FIRST、SQL_FETCH_LAST和SQL_FETCH_ABSOLUTE返回的rowset不依赖于刚刚执行的SQLExtendedFetch的fFetchType值。

根据前面rowset **fFetchType**获取的rowset的其它值有：

- *SQL_FETCH_FIRST*：获取结果集中的第一个rowset。
- *SQL_FETCH_LAST*：获取结果集中最后一个完整的rowset。
- *SQL_FETCH_ABSOLUTE*：获取结果集中以第**irow**行开始的rowset。

  如果**irow** > 0，获取以第**irow**行开始的rowset。

  如果**irow** < 0，获取第**irow**+result set size+1行开始的rowset。

  例如，如果**irow** = -1，那么返回rowset的起始行为结果集的最后一行。如果**irow**小于0，用户可以从结果集末尾倒数找到返回rowset的第一行。

  如果**irow** = 0，返回SQL_NO_DATA_FOUND并将游标置于结果集的开始之前。（到reset）
- *SQL_FETCH_NEXT*：获取下一个rowset。如果当前游标位于结果集的开始之前（例如初始条件），将等同于*SQL_FETCH_FIRST*。
- *SQL_FETCH_PRIOR*：获取前一个rowset。如果当前游标位于结果集的结尾之后（例如初始条件），将等同于*SQL_FETCH_* LAST 。
- *SQL_FETCH_RELATIVE*：从当前rowset的开始处，获取以**irow**行开始的rowset。如果游标置于结果集的开始处之前。

  irow > 0：获取以**irow**行起始的rowset。这等同于SQL_FETCH_ABSOLUTE 值。

  irow < 0：不改变游标，返回SQL_NO_DATA_FOUND 。

  如果游标置于结果集的结尾之后：

  irow < 0：获取以**irow**+result set size+1行开始的rowset。这等同于SQL_FETCH_ABSOLUTE 。

  irow > 0：不改变游标，返回SQL_NO_DATA_FOUND 。

  irow = 0：刷新（重新获取）当前rowset。

- *SQL_FETCH_BOOKMARK*：获取以SQL_ATTR_FETCH_BOOKMARK_PTR属性语句指定的书签为起始的rowset。

irow参数

irow参数指定要获取的行数。如果**fFetchType**参数设置为SQL_FETCH_ABSOLUTE或SQL_FETCH_RELATIVE，用户只需使用**irow**，否则可忽略该值。

pcrow参数

pcrow参数指定从一个**pcrow**调用实际返回的记录数。**pcrow**的有效值为0到rowset大小。

rgfRowStatus参数

rgfRowStatus参数是rowset中所有记录的一组状态值。由ODBC应用程序分配。

SQLExtendedFetch 可设置的状态值是：

- *SQL_ROW_NOROW*：该行中的数据未定义。
- *SQL_ROW_SUCCESS*：使用SQLExtendedFetch成功获取该行数据。
- *SQL_ROW_ERROR*：使用SQL*ExtendedFetch*获取该行数据时发生错误。

例如，如果rowset大小是10，而SQLExtendedFetch只获取了9行记录(例如设置**fFetchType**为SQL_FETCH_ABSOLUTE，**irow**为 -9)，那么最后一行的状态值将是SQL_ROW_NOROW，其它行的状态值是SQL_ROW_SUCCESS。

SQLSetPos用来处理SQLExtendedFetch返回的rowset中的记录，它的值可以为：

- *SQL_ROW_UPDATED*：该行是被更新的。
- *SQL_ROW_DELETED*：该行是被删除的。

- SQL_ROW_ADDED：该行是被添加的。

返回值&错误处理

SQLExtendedFetch一次可获取多条记录。每一个成功获取的记录或给出提示信息的记录状态值都是SQL_ROW_SUCCESS。如果遇到错误，值将是SQL_ROW_ERROR并停止获取。Rowset中的后续记录将被标记为SQL_ROW_NOROW。下面的4个例子中使用的rowset值为5。

➲ **示例 1**

Rowset中的所有记录都被获取：

```
     +------+-----------------+
row1 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row2 | data | SQL ROW SUCCESS |
     +------+-----------------+
row3 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row4 | data | SQL ROW SUCCESS |
     +------+-----------------+
row5 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
```

➲ **示例 2**

结果集结尾附近的rowset中的所有记录都被获取：

```
     +------+-----------------+
row1 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row2 | data | SQL ROW SUCCESS |
     +------+-----------------+
row3 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
```

```
row4 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row5 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
```

➲ **示例 3**

在获取第二行时发生错误：

```
     +------+-----------------+
row1 | data | SQL ROW SUCCESS |
     +------+-----------------+
row2 | xxxx | SQL_ROW_ERROR   |
     +------+-----------------+
row3 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
row4 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row5 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
```

➲ **示例 4**

没有返回记录：

```
     +------+-----------------+
row1 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row2 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row3 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
row4 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row5 | xxxx | SQL_ROW_NOROW   |
```

```
    +------+-----------------+
```

SQLExtendedFetch的返回值取决于rowset中所有记录的值。用户应该检查每一个行的每个字段的状态数组。

SQLExtendedFetch有如下可能返回值：

- *SQL_SUCCESS* - 如果没有发现错误或警告信息，至少有一条记录被返回。
- *SQL_NO_DATA_FOUND* - 如果没有记录返回。
- *SQL_SUCCESS_WITH_INFO* - 如果发现警告信息但没有错误发生。如果调用了**SQLError**，将返回最后一条警告信息的详细情况。
- *SQL_ERROR* - 如果发现错误。接下来所有的**SQLExtendedFetch**调用都会返回同样错误。ODBC程序不会继续访问该结果集。（例如：虽然锁在稍后会被释放，但锁超时被视为错误。需要SQL*Execute*来重新产生结果集。）

使用SQLSetPos修改表

如果结果集来自于一个表，结果集中的每个记录都可以唯一地映射到目标表中的某一行。然后rowset中的每一行，将通过OID与目标表中的物理记录建立关联。

➲ **示例 1**

从下列查询语句返回的rowset都被更新：

```
create table t1 (c1 int, c2 int, c3 char(5))
select * from t1;
select * from t1 where c1 > 10;
select c1 from t1 where c2 < 20;
select c2, c1 from t1;
```

➲ **示例 2**

从下列查询语句返回的rowset是不可更新的：

```
create table t1 (c1 int, c2 int, c3 char(5))
```

```
select * from t1,t2 ;

select c1+c2 from t1 ;

select c1*2 from t1 ;
```

SQLSetPos目前只支持对一个表上的简单扫描进行修改。只有这样的表达式可以被修改。像**c1*2**、**c1+1**和**c1+c2**这样的表达式不能修改。

如果需要，对于非映射中的字段可以使用字段默认值。例如通过SQLSetPos插入的记录。绑定字段是映射的子集，而映射又是在对单一表的简单扫描中的表模式的子集，可使用SQLPutData处理未绑定字段。

SQLSetPos 的参数

➲ **原型**

SQLSetPos：

```
RETCODE SQLSetPos(
                    HSTMT    hstmt,
                    UWORD    irow,
                    UWORD    fOption,
                    UWORD    fLock)
```

SQLSetPos的可能返回值如下：

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

因为SQLSetPos处理rowset，所以必须在调用SQLExtendedFetch之后调用。要被处理的rowset来自于前面的SQLExtendedFetch。注意，参数**rgfRowStatus**（行状态数组）由SQLSetPos（根据不同选项）设置，并由相应的SQLExtendedFetch传递而来。在SQLSetPos访问行数据时，ODBC程序需要再次检查SQLSetPos返回值和行记录数组。如果结果集是不可修改的，SQLSetPos会返回错误。

由SQLSetPos带有不同选项设置的行状态值有：

- SQL_ROW_SUCCESS
- SQL_ROW_ERROR
- SQL_ROW_NOROW
- SQL_ROW_UPDATED
- SQL_ROW_DELETED
- SQL_ROW_ADDED

irow 参数

irow是**fOption**指定的操作将要处理的rowset中的记录数，如果值是0，操作将应用于rowset中的所有行。

fOption参数

适用于SQLExtendedFetch返回的rowset的操作有：

- SQL_POSITION
- SQL_REFRESH
- SQL_UPDATE
- SQL_DELETE
- SQL_ADD

SQL_POSITION为那些需要游标的操作将游标定位于rowset的记录上，该选项不改变记录状态数组。

如果**irow** = 0：游标定位于整个rowset上。

如果**irow** = n：游标定位于第“n”行（n =1 或 <= rowset size）。

如果该选项用于将游标定位于多个行处，那么定位语句仅在所选行的第一个行上执行。

➲ **示例**

```
/* hstmtS is for SQLExtendedFetch, SQLSetPos and SQLSetCursorName */
```

```
/* hstmtU is for positioned statement                              */
/* use hstmtS to query */
#define ROWSET_SIZE   6                    /* rowset size(6 tuples) */
#define NAME_LEN     30                    /* length of NAME column */
#define BDAY LEN  30                       /* length of BIRTHDAY column */
char   szName[ROWSET SIZE][NAME LEN], szBirthday[ROWSET SIZE][BDAY LEN];
SDWORD cbName, cbBirthday;
char szReply[10];
SQLULEN irow,crow;
rc=SQLSetStmtOption(hstmtS, SQL ROWSET SIZE, ROWS);
rc=SQLSetCursorName(hstmtS, (UCHAR *)"C1", SQL_NTS);
rc=SQLExecDirect(hstmtS, (UCHAR *)"SELECT NAME, BIRTHDAY FROM EMPLOYEE
                                   FOR UPDATE OF BIRTHDAY", SQL NTS);
rc=SQLBindCol(hstmtS, 1, SQL C CHAR, szName, NAME LEN, &cbName);
rc=SQLBindCol(hstmtS, 2, SQL_C_CHAR, szBirthday, BDAY_LEN, &cbBirthday);

/* use hstmtS (through SQLExtendedFetch) to browse all rows */
while (1) {
rc=SQLExtendedFetch(hstmtS, SQL_FETCH_NEXT, 0, &crow,
                  rgfRowStatus);
    if (rc == SQL ERROR || rc == SQL NO DATA FOUND)
        break;
    for (irow = 0; irow < crow; irow++) {
        if (rgfRowStatus[irow] != SQL_ROW_DELETED)
              printf("%d %10s : %30s\n", irow+1, szName[irow],
                      szBirthday[irow]);
    }/*for*/
    /* read user input for line number and data to update */
    /* use hstmtS to position cursor for hstmtU           */
    /* use hstmtU to execute positioned update statement  */
```

```
    while (TRUE) {
      printf("\nRow number to update? (0 to quit)");
      gets((char *)szReply);
      irow = atoi((char *)szReply);
      if (irow > 0 && irow <= crow) {
          printf("\nNew birthday? ");
          gets((char *)szBirthday[irow-1]);
          rc=SQLSetPos(hstmtS, irow, SQL_POSITION, SQL_LOCK_NO_CHANGE);
          rc=SQLPrepare(hstmtU,
               (UCHAR *)"UPDATE EMPLOYEE SET BIRTHDAY=? WHERE CURRENT OF C1",
                      SQL_NTS);
          rc=SQLBindParameter(hstmtU, 1, SQL_PARAM_INPUT,
                              SQL C CHAR, SQL CHAR, BDAY LEN, 0,
                              szBirthday[irow-1], 0, NULL);
          rc=SQLExecute(hstmtU);
          rc=SQLFreeStmt(hstmtU,SQL_CLOSE);
      } else if (irow == 0) {
          break;
      }
    }/*wh*/
}/*wh*/
```

SQL_REFRESH用于刷新rowset中的行记录。这会重新获取同样的窗口进入rowset缓存中。最新被获取的行状态被设置为SQL_ROW_SUCCESS，不包含在结果集中的行状态设置为SQL_ROW_NOROW。

如果**irow** = 0，游标定位于整个rowset上并刷新。

如果**irow** = n，DBMaster不支持带有SQL_REFRESH设置的**irow**为其他值。

如果该选项成功，那么刷新行的状态将设为SQL_ROW_SUCCESS。如果刷新后的rowset中布满了由SQL_DELETE选项产生的空洞，可以在结果集中前后移动窗口。

SQL_UPDATE用于更新行数据。目标表中的相应行被来自rowset缓存中的数据更新。成功更新的行的状态设置为SQL_ROW_UPDATED。不能更新标记为SQL_ROW_DELETED的行。

如果**irow** = 0，游标定位于整个rowset上并更新。

如果**irow** = n，游标定位于第n行，更新第n行。

SQL_DELETE用于在目标表中删除rowset映射的相应行。不能删除标记为SQL_ROW_DELETED的行。如果记录被删除（设置为SQL_ROW_DELETED），不能再对其执行下列操作：定位UPDATE/DELETE语句、调用SQLGetData、调用除选项SQL_POSITION之外SQLSetPos（对于设置为SQL_ROW_DELETED的行，您只能调用带有选项SQL_SET_POSITION的SQLSetPos）。

如果**irow** = 0，游标定位于整个rowset上并删除。

如果**irow** = n，游标定位于第n行，删除n行。

SQL_ADD用于添加行记录。被添加的行的状态设置为SQL_ROW_ADDED。如果应用了该选项，rowset便被当做用于插入数据的输入缓存。目标表中没有相应的记录映射到rowset。这是唯一允许**irow**值大于rowset size的选项，且该选项不改变游标的位置（没有定位游标）。当插入未绑定到rowset缓存的字段时，使用默认值（如果有默认值）或NULL值（如果没有默认值）。

如果**irow** = 0，rowset中所有记录都被添加。

如果**irow** = 1 ~ rowset size：添加**irow**值行。

如果**irow** > *rowset* size：依然从rowset缓存的起始处，带有合适的偏移量来寻找**irow**。例如，如果 **irow** = rowset size + 1，或者 **irow** = rowset size + 2，添加行**irow**。

ODBC应用程序分配的缓存空间要多于rowset缓存空间，rowset缓存空间由SQL_ROWSET_SIZE指定。这简化了ODBC应用程序。如果没有分配

额外的缓存且大于rowset size，那么会因为试图访问非法内存而产生错误。

fLock 参数

锁定或解锁目标表中的相应操作行。

fLoke的有效值是：

- *SQL_LOCK_NO_CHANGE*：不更改行的锁定模式。

字段指示器

当ODBC应用程序想要向字段中插入NULL值，唯一的接口便是字段指示器（状态）。没有可用的主变量（例如INSERT INTO t1 VALUES (?,?)中的主变量）可以从其返回信息或指定允许插入NULL值。使用SQLExtendedFetc和SQLSetPos，SQLBindCol的字段指示器是用来指定获取信息和修改信息的唯一接口。

例如，如果ODBC程序想要更新一个带有NULL值（或插入NULL值）的字段，那么在调用带有SQL_UPDATE或SQL_INSERT的选项的SQLSetPos之前，相应的字段指示器要设置为SQL_NULL_DATA，如果使用默认值，方法相同，只是将字段指示器设置为SQL_DEFAULT_PARAM。

SQLPutData

如果使用SQLExtendedFetch，那么SQLGetData是唯一获取未绑定字段（大部分都是BLOB或文件对象字段）的方式。同样，您可以调用带有SQLPutData的SQLSetPos来修改那些未绑定字段（大部分都是BLOB或文件对象字段）。没有字段指示器可用，您只能使用SQLPutData的**cbValue**参数。在执行SQLPutData或SQLGetData之前，rowset size必须是1。

对于非映射字段，带有SQL_ADD选项的SQLSetPos将使用默认值。

➲ **示例 1**

创建表：

```
create table t1 (c1 int, c2 int, c3 char(5) default 'col3')
```

➲ **示例 2**

一个Select查询：

```
select c1, c2 from t1
```

➲ **示例 3**

使用下面的调用修改表**t1**：

```
/* bind columns c1, c2 , execute and fetch */
SQLBindCol(hstmt, 1, SQL_C_CHAR, c1_rgbValue, c1_len ,c1_pcbValue);
SQLBindCol(hstmt, 2, SQL_C_CHAR, c2_rgbValue, c2_len ,c2_pcbValue);
SQLExecute(hstmt);
SQLExtendedFetch(hstmt, SQL_FETCH_FIRST, 0, &crow, rgfRowStatus);

/* specify c1, c2 values in first row of rowset  */
/* first row of rowset is used as input buffer   */
SQLSetPos(hstmt, 1, SQL_ADD, SQL_LOCK_NO_CHANGE);  /* default for c3 */
/* specify c1,c2 values in first row of rowset   */
/* first row of rowset is used as input buffer   */
/* update the row (in table t1) corresponding to */
/* first row in rowset */
SQLSetPos(hstmt, 1, SQL_UPDATE, SQL_LOCK_NO_CHANGE);/* c3 is not changed */
```

字段**c3**不是映射字段，在rowset中只能找到**c1**和**c2**。换句话说，要插入一个额外字段，SQLSetPos从rowset中获取**c1**和**c2**的值，为字段**c3**使用默认值**col3**。要更新rowset中的第一行在表**t1**中的相应行，请确定**c3**没有被改变，因为**c3**不在映射中。

对于绑定字段，SQLSetPos从rowset（绑定缓存）中获取它需要(为选项SQL_ADD和SQL_UPDATE)的所有输入数据。

对每个未绑定字段，如果它不是BLOB(LONG VARCHAR或LONG VARBINARY)或文件类型，那么如果SQLSetPos需要的话仍然使用默认

值。不能使用SQLPutData为这类型的字段插入值，因为在执行SQLSetPos时使用了默认值。

对于未绑定的BLOB/文件对象字段，必须使用SQLPutData进行修改。在SQLPutData之前和SQLSetPos之后，仍然需要执行SQLParamData找到所有未绑定的BLOB/文件对象字段。

BLOB(LONG VARCHAR 和 LONG VARBINARY) 字段：

- 要输入*NULL*值；调用SQL*PutData*将参数**cbValue**设置为*SQL_NULL_DATA*。
- 要输入*默认*值；调用SQL*PutData*将参数**cbValue**设置为*SQL DEFAULT_PARAM*。
- 要输入数据；调用SQL*PutData*，在参数**rgbValue**和***SQL_NTS***中输入数据。或在参数**cbValue**中输入**rgbValue**的长度。要输入数据，*LONG VARCHAR* 和*LONG VARBINARY*数据类型的*SQL_C_TYPE*为*SQL_C_LONGVARCHAR* 和*SQL_C_LONGVARBINARY*。

文件类型字段：

- 要输入*NULL*值；调用SQL*PutData*将参数**cbValue**设置为*SQL_NULL_DATA*。
- 要输入*默认*值；调用SQL*PutData*将参数**cbValue**设置为*SQL DEFAULT_PARAM*。

➲ **示例 4**

执行SQLSetPos，make该调用，然后执行SQLPutData输入数据：

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_DATA|col);
```

这指示了正在进行的调用SQLPutData从参数rgbValue中插入数据到字段**col**中，字段**col**是映射中的文件对象类型字段。且是映射目标文件中的文件对象字段的索引。选项SQL_SPOS_FO_DATA会强迫系统使用SQL_C_CHAR或SQL_LONGVARCHAR来绑定输入数据，同时会为该类型的数据输入自动产生一个系统文件。

➲ **示例 5**

执行SQLSetPos，调用以下函数，然后执行SQLPutData输入用户文件：

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_SFILE|col)
```

这指示了正在进行的调用SQLPutData插入一个名由**rgbValue**指定的用户文件到字段**col**，**col**是映射中的文件对象字段。选项SQL_SPOS_FO_SFILE会强迫系统使用SQL_C_CHAR或SQL_ FILE来绑定输入数据。

➲ **示例 6**

执行SQLSetPos，调用以下函数，然后执行SQLPutData输入系统文件：

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_CFILE|col)
```

这指示了正在进行的调用**SQLPutData**插入一个名由**rgbValue**指定的系统文件到字段**col**，**col**是映射中的文件对象字段。选项**SQL_SPOS_FO_CFILE**会强迫系统使用**SQL_LONGVARCHAR**或SQL_ FILE来绑定输入数据。

下列演示如何使用**SQLSetPos和SQLPutData**输入BLOB和文件对象数据：

➲ **示例 7**

创建表模式：

```
create table t1 (c1 int, c2 long varchar, c3 file, c4 int default 10)
```

➲ **示例 8**

一个 Select查询：

```
select c2, c3, c4 from t1
```

➲ **示例 9**

使用代码：

```
/* do not bind any column                                              */
SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, 1);
       :
```

```
/* execute and fetch                                                    */
SQLExecute(hstmt);
SQLExtendedFetch(hstmt, SQL_FETCH_FIRST, 0, &crow, rgfRowStatus);
/* call SQLSetPos to insert one tuple                                   */
/* SQLSetPos returns SQL NEED DATA                                      */
SQLSetPos(hstmt, 1, SQL ADD, SQL LOCK NO CHANGE);  /* default(10) for c4 */
/* input null for c2(long varchar)                                      */
SQLParamData(hstmt, (void *)&paranum);
SQLPutData(hstmt, buf,SQL NULL DATA);
/* input user file for c3(file)                                         */
SQLParamData(hstmt, (void *)&paranum);
/* specify user file input and place file name in sbuf                  */
SQLSetStmtOption(hstmt, SQL SPOS FO, SQL SPOS FO SFILE|2);   /* 2 for c3 */
       :
/* input user file for c3(file)                                         */
SQLPutData(hstmt, sbuf, strlen(sbuf));
```

使用 SQLSetPos

SQLSetPos可同时修改多条记录，返回值的规则与SQLExtendedFetch类似。对记录的每一个成功操作将根据所使用的选项进行标记。如果发现错误，行将被标记为SQL_ROW_ERROR，除非发生严重的错误如中止事务等，否则操作将不会停止。

规则如下：

- 如果没有错误或警告发生，返回*SQL_SUCCESS*。
- 如果没有返回记录，返回*SQL_NO_DATA_FOUND*（仅对选项*SQL_REFRESH*）。
- 如果操作期间发生错误或警告，返回*SQL_SUCCESS_WITH_INFO*。可以调用获取完整的错误信息。如果只有警告而没有错误信息，那么只记录最后一条警告。

- 如果发生严重错误，返回*SQL_ERROR*。

注意，在错误发生元组之前的SQLSetPos部分结果是未完成的。也就是说还不是原子操作。如果开启自动提交模式，那么每个SQLSetPos调用(除带有选项SQL_REFRESH 和SQL_POSITION之外)会在成功后自动提交。

使用 SQLSETPOS的限制

从带子查询的查询中返回的结果集不能修改，对这类的结果集不能调用SQLSetPos。

6 错误处理

读完之前的章节，用户应该能够构建一个ODBC程序。但是当调用ODBC函数出现问题时，用户需要做些什么呢？本章介绍了错误发生时如何获取错误信息，以及一些ODBC目录函数以允许用户从系统目录（系统表）中获取信息。同时也涵盖了一些别的ODBC函数，包括用于获取有关数据源系统信息的函数，比如支持的数据类型、built-in函数和ODBC函数。

在本章用户将学习到：

- 当使用*SQLError*调用一个*ODBC*函数失败时获取详细的错误信息。
- 通过使用诸如*SQLTables*、*SQLColumns*、*SQLStatistics*和*SQLSpecialColumns*返回如表模式和统计信息的目录信息。
- 通过使用*SQLGetTypeInfo*、*SQLGetInfo*和*SQLGet*函数获取关于数据源的系统信息。

注意 *收集错误信息的方式是不同的。*

6.1 获取错误信息

当一个程序执行ODBC函数并返回错误代码时，就需要详细的错误信息来了解引起该错误的原因。这一节将说明如何使用SQLError函数来获取错误信息。

ODBC中普通错误代码的定义

在调用一个ODBC函数之后，用户可能会得到以下返回代码：

- ***SQL_SUCCESS*** — ODBC函数执行成功。
- ***SQL_SUCCESS_WITH_INFO*** — ODBC函数执行成功，但是一些警告信息被返回。
- ***SQL_NO_DATA_FOUND*** — 没有更多的信息被获取。
- ***SQL_ERROR*** —发生错误并且函数调用失败。
- ***SQL_INVALID_HANDLE*** —发现一个无效句柄并且函数调用失败。
- ***SQL_NEED_DATA*** — 驱动器显示程序必须发送参数值。

如果一个程序调用任意ODBC函数（除了SQLError本身）并且返回的代码是**SQL_ERROR**或**SQL_SUCCESS_WITH_INFO**，它将调用SQLError来获取额外的错误信息。

如何使用SQLError

SQLError用于获取输入句柄中的错误信息，包括错误消息、错误状态和驱动器的内部错误代码。驱动器内部错误代码是通过每个驱动器定义的，对于不同的驱动器错误代码也可能不同。（内部DBMaster错误代码，请参考*附录C*）

当通过之前的ODBC函数返回的错误代码是**SQL_ERROR**或**SQL_SUCCESS_WITH_INFO**时，程序将调用SQLError。

➲ 原型

SQLError：

```
RETCODE  SQLError(
                  HENV        henv,
                  HDBC        hdbc,
                  HSTMT       hstmt,
                  UCHAR  FAR *szSqlState,
                  SDWORD FAR *pfNativeError,
                  UCHAR  FAR *szErrorMsg,
                  SWORD       cbErrorMsgMax,
                  SWORD  FAR *pcbErrorMsg)
```

SQLError参数列表中的这三个句柄并不是完全需要传递到SQLError。ODBC驱动器将从最右边的non-null句柄中找到相关的返回码。

- ***henv*** —环境句柄。
- ***hdbc*** —数据库连接句柄。
- ***hstmt*** —语句句柄。
- ***szSqlState*** —SQLSTATE，作为5个字符的字符串返回，并且以空字符终止。前两个字符指示错误类别，紧随着的三个字符指示子类别。这些值直接与X/Open SQL CAE规范和ODBC规范中定义的SQLSTATE值相对应，而与特定于IBM®和产品的 SQLSTATE值相矛盾。
- ***pfNativeError*** —本机错误代码。
- ***szErrorMsg*** —指向一个缓冲区的指针，该缓冲区包含由实现定义的消息文本。
- ***cbErrorMsgMax*** —缓冲区szErrorMsg的最大长度（即分配的长度）。
- ***pcbErrorMsg*** —指向可以返回到szErrorMsg缓冲区的总字节数的指针。

➲ **示例 1**

```
SQLError(henv, hdbc, hstmt, ...)
```

➲ **示例 2**

返回**有关hdbc的错误信息**：

```
SQLError(SQL_NULL_ENV, hdbc, SQL_NULL_STMT, ...)
```

这个驱动将返回与hdbc相关的错误信息。用户应该确认程序传递正确的句柄到SQLError，这样他们所需的错误信息才能成功被获取。

若没有错误信息被获取，SQLError将返回**SQL_NO_DATA_FOUND**。在每次调用SQLError并返回错误信息后，句柄中的此错误信息将被删除，这就意味着调用一个ODBC函数的错误信息只能被获取一次。

通过SQLError返回SQL访问组的SQL CAE规范(1992)和X/Open定义**SQLSTATE**值。这些值是由两个字符的class值后跟随一个三个字符的subclass值总共五个字符串组成。例如，class值01是一个警告，与此相应的返回代码是**SQL_SUCCESS_WITH_INFO**。请参考*Microsoft ODBC程序手册*以获取ODBC中**SQLSTATE**值定义的详细信息。

➲ **例**

SQLError with SQLState：

```
#define  MSG LEN  256       /* error message buffer length          */
HENV   henv;                /* environment handle                   */
HDBC   hdbc;                /* connection handle                    */
HSTMT  hstmt;               /* statement handle                     */
SDWORD retcode, retcode1;   /* return code                          */
UCHAR  sqlState[6];         /* buffer to store SQLSTATE             */
SDWORD nativeErr;           /* native error code                    */
UCHAR  errMsg[MSG LEN];     /* buffer to store error message        */
SWORD  realMsgLen;          /* real length of returned error message */
retcode = SQLAllocEnv(&henv);
```

```
retcode = SQLAllocConnect(&hdbc);
/* Use specified DB NAME(data source name), uid (user id),              */
/* pwd (password) to connect to a data source.  If any warnings or      */
/* errors are detected, call SQLError and pass hdbc to retrieve         */
/* error information from the connection handle with other handles      */
/* set to NULL.  Then print the error information and return.           */
retcode = SQLConnect(hdbc, DB_NAME, SQL_NTS, uid, SQL_NTS, pwd,
                          SQL_NTS);
if (retcode != SQL SUCCESS)    /* warning or error returned        */
    {
     retcode1 = SQLError(SQL_NULL_HENV, hdbc, SQL_NULL_HSTMT, sqlState,
                         &nativeErr, errMsg, MSG_LEN, &realMsgLen);
     print err(sqlState, nativeErr, errMsg, realMsgLen);
     return;
    }
/* Get SQL command string and execute it.  If any warnings or errors    */
/* are detected, call SQLError and pass hstmt to retrieve error         */
/* information from the statement handle, then print the error          */
/* information and return.                                              */

retcode = execute cmd(hstmt);   /* execute a SQL command            */
if (retcode != SQL_SUCCESS)     /* warning or error returned        */
    {
     retcode1 = SQLError(SQL_NULL_HENV, SQL_NULL_HDBC, hstmt, sqlState,
                         &nativeErr, errMsg, MSG LEN, &realMsgLen);
     print_err(sqlState, nativeErr, errMsg, realMsgLen);
     return;
    }
```

错误列

ODBC允许多个错误代码存储于一个错误序列中并与一个句柄相关联。SQLError可被调用多次来依次获取错误代码。通常DBMaster将只为*数据库一致性检查*(DBCC)操作返回多个错误代码，这些错误都存储于错误序列中。

一个应用能多次调用SQLError直到所有错误序列中的错误被获取。一旦所有错误被获取，SQLError将返回**SQL_NO_DATA_FOUND**。

➲ **示例**

统计表一致性检查的应用程序：

```
#define  MSG_LEN  256       /* error message buffer length           */
UCHAR   sqlState[6];        /* buffer to store SQLSTATE              */
SDWORD  nativeErr;          /* native error code                     */
UCHAR   errMsg[MSG LEN];    /* buffer to store error message         */
SWORD   realMsgLen;         /* real length of returned error message */
SWORD   count;
SWORD   retcode;
retcode = SQLExecDirect (hstmt, (SQLCHAR *)"check table account", SQL NTS);

do {
    retcode = SQLError(SQL NULL HENV, SQL NULL HDBC, hstmt, sqlState,
                       &nativeErr, errMsg, MSG LEN, &realMsgLen);

    if (retcode == SQL_NO_DATA_FOUND)
        {
         printf("check error queue finish \n\n");
         break;
         }
    count++;
    printf("-->Error %d :\n", count);
```

```
    printf("   SQLSTATE = %s \n", sqlState);
    printf("   native error = %ld \n", nativeErr);
    printf("   error message = %s \n", errMsg);
    printf("   error message length = %d \n", realMsgLen);
   }
while ((retcode == SQL_Error) || (retcode == SQL_SUCCESS_WITH_INFO));
```

6.2 目录函数

关系数据库中的一些系统表可用来记录表、字段和权限等信息，这就是所谓的*目录*。此目录用于读取数据库中表和索引的模式信息。

所有目录函数都在同样方式下工作。当调用目录函数时，可使用参数指定信息并返回一个结果集，用户可从此结果集中获取数据。

本节将介绍四个经常使用的目录函数：SQLTables、SQLColumns、SQLStatistics以及SQLSpecialColumns。

- ***SQLTables*** - 在数据库中获取一列表或视图名称。
- ***SQLColumns*** - 获取有关指定表的字段信息。
- ***SQLStatistics*** - 获得表和与这些表相关联的索引统计信息。
- ***SQLSpecialColumns*** - 获得表中唯一识别行的最佳字段集。

查找模式

目录函数中的一些参数通过查找模式来选择想要得到的对象。最简单的查找模式用于精确匹配用户所寻找的项目字符串。另外，用户能够在一个查找模式中使用通配符来进行更强大的查找。DBMaster支持以下符号：下划线(_)、百分号(%)以及斜杠(\)。

- 下划线(_)可用于匹配一个字符。
- 百分号*(%)*用于匹配0或更多字符。
- 换码符*(\)*允许符号%或_在查找模式中当做文字字符使用。在查找模式中若使用斜杠\作为一个文字字符，应该使用双斜杠*(\\)*。

例如，若一个表名的查找模式是%A%，此函数将返回表名称中包含字符A的所有表；若一个表名的查找模式是_A_，此函数将返回表名称为三个字符长并且A在中间的所有表；若一个表名的查找模式是%，此函数将返回所有表。

如果用户想重新获得表名为***TAB_TEST***的信息，可使用***TAB_TEST***作为查询模式，然后会在结果集中获得诸如***TAB1TEST***、***TAB2TEST***等表信息。但这并不是用户想要的，解决该问题的方法是在元字符前加一个斜杠如：***TAB_TEST***。

注意 *当通过C编译器传递此字符串时，用户必须指定TAB_TEST而不是TAB_TEST。这是因为C编译器会将“\”当做一个符号。请参考以下SQLTables的示例。*

SQLTables

当用户连接一个数据库并想确定关于所有或者一个特殊表集的信息时，可调用已指定标准的SQLTables来解决此问题。

➲ **原型**

SQLTables：

```
RETCODE SQLTables (
                    HSTMT   hstmt,
                    UCHAR   *szTableQualifier,
                    SWORD   cbTableQualifier,
                    UCHAR   *szTableOwner,
                    SWORD   cbTableOwner,
                    UCHAR   *szTableName,
                    SWORD   cbTableName,
                    UCHAR   *szTableType,
                    SWORD   cbTableType)
```

SQLTables的参数是：

- *hstmt* —重新获得结果的一个有效语法句柄。
- *szTableQualifier* — DBMaster不支持，它应该是***NULL***或空字符串。
- *cbTableQualifier* — *szTableQualifier*长度应该为0。

- *szTableOwner* — 指出所有者名称。所有者是创建表或视图的用户，它可以作为查找模式的字符串或者是一个***NULL***值。使用***NULL***值表示所有用户。
- *cbTableOwner* — *szTableOwner*的长度或***SQL_NTS***。
- *szTableName* — 指出表或者视图的名称。它可以作为查找模式的字符串或是一个***NULL***值。使用***NULL***值表示所有名称。
- *cbTableName* — *szTableName*的长度或***SQL_NTS***。
- *szTableType* — 表类型列表*(TABLE和/或VIEW)*。
- *cbTableType* — *szTableType*长度。

注意 *用作查找模式的字符串可存在于**szTableQualifier**、**szTableOwner**和**szTableName**中。这三个参数和它们相对应的字符长度参数**cbTableQualifier**、**cbTableOwner**和**cbTableName**也会出现在其它三个目录参数中：**SQLColumns**、**SQLStatistics**以及**SQLSpecialColumns**。*

SQLTables返回一个由以下字段组成的结果集：

字段号	字段名称	数据类型
1	TABLE_QUALIFIER	VARCHAR(128)
2	TABLE_OWNER	VARCHAR(128)
3	TABLE_NAME	VARCHAR(128)
4	TABLE_TYPE	VARCHAR(128)
5	REMARKS	VARCHAR(254)

SQLTables根据用户标准返回一个结果集。例如：当szTableName是%A%并且szTableOwner是_A_时，此结果集将包括所有名称中含有字符A以及所有者名称为三个字符并且A在中间的表。如果用户想获取数据库中所有表的名称，只需要设置szTableQualifier、szTableOwner以及szTableName为**NULL**即可。

实际上，用户可以考虑SQLTables作为一种使用SQLExecDirect执行查询的方法。这就意味着需要使用**SQLFetch**来获得结果集。在使用SQLFetch之前，用户应该通过SQLBindCol在结果集中绑定字段。

以下代码给出了一个SQLTables的示例。假设有两个名为***TAB_TEST1***和***TAB_TEST2***的表。在调用SQLTables之后，用户将得到***TAB_TEST1***和***TAB_TEST2***的信息， 顺序为***TABLE_TYPE***、***TABLE_QUALIFIER***、***TABLE_OWNER*** 和***TABLE_NAME***。

➲ 示例

```
HDBC   hdbc;
HSTMT  hstmt;
UCHAR  tabQualifier[255], tabOwner[255], tabName[255];
UCHAR  tabType[255], remarks[255];

SDWORD lenTabQualifier, lenTabOwner, lenTableName;
SDWORD lenTableType, lenRemarks;
SDWORD retcode;

...

retcode = SQLAllocStmt(hdbc,&hstmt);

retcode = SQLTables(hstmt,
                    (UCHAR FAR *)NULL, 0,            /* tabQualifier  */
                    (UCHAR FAR *)NULL, 0,            /* tabOwners     */
                    (UCHAR FAR *)"DB\\ %", SQL NTS,  /* table name    */
                    (UCHAR FAR *)"TABLE", SQL_NTS);  /* table type    */

/* Bind columns in result set to storage locations                      */

retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, tabQualifier, 255,
                     &lenTabQualifier);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, tabOwner, 255, &lenTabOwner);
retcode = SQLBindCol(hstmt, 3, SQL_C_CHAR, tabName, 255, &lenTableName);
```

```
retcode = SQLBindCol(hstmt, 4, SQL_C_CHAR, tabType, 255, &lenTableType);
retcode = SQLBindCol(hstmt, 5, SQL C CHAR, remarks, 255, &lenRemarks);
while ((retcode = SQLFetch(hstmt)) == SQL_SUCCESS)
      {                    /* print out the record in the result set */
       printf("column 1 : table qualifier = %s\n", tabQualifier);
       printf("column 2 : table owner = %s\n", tabOwner);
       printf("column 3 : table name = %s\n", tabName);
       printf("column 4 : table type = %s\n", tabType);
       printf("column 5 : remarks = %s\n", remarks);
      }
...
```

注意 *当函数返回一个结果集时，用户应该使用SQLBindCol和SQLFetch来获得结果集中的行。SQLTables、SQLColumns、SQLStatistics和SQLSpecialColumns就是这样的函数。*

SQLColumns

用户能够使用SQLTables来获取数据库中表的信息。同样也能够使用**SQLColumns**函数来获取一个特殊表中字段的信息。

➲ **SQLColumns原型是：**

```
RETCODE SQLColumns (
                    HSTMT    hstmt,
                    UCHAR   *szTableQualifier,
                    SWORD    cbTableQualifier,
                    UCHAR   *szTableOwner,
                    SWORD    cbTableOwner,
                    UCHAR   *szTableName,
                    SWORD    cbTableName,
                    UCHAR   *szColumnName,
                    SWORD    cbColumnName);
```

函数**szTableQualifier**、**cbTableQualifier**、**szTableOwner**、**cbTableOwner**、**szTableName**和**cbTableName**的定义与**SQLTables**基本相同。**szColumnName**指字段名称的查询模式字符串，**cbColumnName**是**szColumnName**的长度。

正如**SQLTables**函数，将返回匹配上述参数标准的一个结果集，并且包含字段信息。

以下是此结果集的字段列表：

字段号	字段名称	数据类型	说明
1	TABLE_QUALIFIER	VARCHAR(128)	
2	TABLE_OWNER	VARCHAR(128)	
3	TABLE_NAME	VARCHAR(128)	NOT NULL
4	COLUMN_NAME	VARCHAR(128)	NOT NULL
5	DATA_TYPE	SMALLINT	NOT NULL
6	TYPE_NAME	VARCHAR(128)	NOT NULL
7	PRECISION	INTEGER	
8	LENGTH	INTEGER	
9	SCALE	SMALLINT	
10	RADIX	SMALLINT	
11	NULLABLE	SMALLINT	NOT NULL
12	REMARKS	VARCHAR(254)	

结果集排列顺序为：***TABLE_QUALIFIER***、***TABLE_OWNER*** 和 ***TABLE_NAME***。用户应该使用SQLBindCol在结果集中绑定这些字段，然后使用SQLFetch获得结果。

SQLStatistics

SQLStatistics用于找回指定表以及与这些表相关联的索引的统计列表。

➲ **原型**

SQLStatistics：

```
RETCODE SQLStatistics (
                        HSTMT    hstmt,
                        UCHAR   *szTableQualifier,
                        SWORD    cbTableQualifier,
                        UCHAR   *szTableOwner,
                        SWORD    cbTableOwner,
                        UCHAR   *szTableName,
                        SWORD    cbTableName,
                        UWORD    fUnique,
                        UWORD    fAccuracy)
```

函数**szTableQualifier**、**cbTableQualifier**、**szTableOwner**、**cbTableOwner**、**szTableName**和**cbTableName**的定义与**SQLTables**以及**SQLColumns**基本相同。**fUnique**用于定义返回索引的类型，**fAccuracy**用于定义结果集中***CARDINALITY***和***PAGES***字段的重要性。

注意 ***fUnique****有两个选项：SQL_INDEX_UNIQUE或SQL_INDEX_ALL。****Accuracy****也有两个选项SQL_ENSURE或SQL_QUICK。*

以下是结果集中的字段列表：

字段号	字段名称	数据类型	说明
1	TABLE_QUALIFIER	VARCHAR(128)	
2	TABLE_OWNER	VARCHAR(128)	
3	TABLE_NAME	VARCHAR(128)	NOT NULL
4	NON_UNIQUE	SMALLINT	
5	INDEX_QUALIFIER	VARCHAR(128)	
6	INDEX_NAME	VARCHAR(128)	

字段号	字段名称	数据类型	说明
7	TYPE	SMALLINT	NOT NULL
8	SEQ_IN_INDEX	SMALLINT	
9	COLUMN_NAME	VARCHAR(128)	
10	COLLATION	CHAR(1)	
11	CARDINALITY	INTEGER	
12	PAGES	INTEGER	
13	FILTER_CONDITION	VARCHAR(128)	

TYPE的字段值是**SQL_TABLE_STAT**或**SQL_INDEX_OTHER**。**SQL_TABLE_STAT**显示包含表统计的行，并且***NON_UNIQUE***、***INDEX_QUALIFIER***、***INDEX_NAME***、***SEQ_IN_INDEX***、***COLUMN_NAME***、***COLLATION***和***FILTER_CONDITION***字段将是**NULL**。另一方面，**SQL_INDEX_OTHER**显示包含索引统计的行。

与SQLTables和SQLColumns相似，用户需要SQLBindCol和SQLFetch在结果集中重新获取数据。在结果集中字段的顺序为：***NON_UNIQUE***、***TYPE***、***INDEX_QUALIFIER***、***INDEX_NAME***和***SEQ_IN_INDEX***。关于相似函数的代码示例，请参考SQLTables示例。

SQLSpecialColumns

从此函数名称可以看出，SQLSpecialColumns返回在表中唯一指定行的特定字段。

➲ **原型**

SQLSpecialColumns：

```
RETCODE SQLSpecialColumns (
                            HSTMT    hstmt,
                            UWORD    fColType,
                            UCHAR   *szTableQualifier,
                            SWORD    cbTableQualifier,
                            UCHAR   *szTableOwner,
                            SWORD    cbTableOwner,
```

```
                        UCHAR   *szTableName,
                        SWORD    cbTableName,
                        UWORD    fScope,
                        UWORD    fNullable);
```

Hstmt是一个有效的语句句柄，szTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableName的所有定义与SQLTables基本相同。fColType指定返回的字段类型，fScope是特定字段的最小所需范围，fNullable确定返回的特殊字段能否有**NULL**值。

注意 ***fColType****有两个选项：SQL_BEST_ROWID和SQL_ROWVER。*
fScope*有三个选项：SQL_SCOPE_CURROW、SQL_SCOPE_TRANSACTION和SQL_SCOPE_SESSION。*
fNullable*有两个选项：SQL_NO_NULLS和SQL_NULLABLE。*

下表列出了结果集中的字段：

字段号	字段名称	数据类型	说明
1	SCOPE	SMALLINT	
2	COLLUMN_NAME	VARCHAR(128)	NOT NULL
3	DATA_TYPE	SMALLINT	NOT NULL
4	TYPE_NAME	VARCHAR(128)	NOT NULL
5	PRECISION	INTEGER	
6	LENGTH	INTEGER	
7	SCALE	SMALLINT	
8	PSEUDO_COLUMN	SMALLINT	

DBMaster提供一个特殊的行指示器*OID*，它与Oracle中的*ROWID*或Ingres中*TID*的相似。OID在表中被当做一个pseudo-column，因为像SELECT * FROM ACCOUNT的查询将不会返回这样的字段名，但是用户还能在一个选择列表中使用OID或WHERE子句来获取用户想明确指定的记录。

一旦用户在fColType中指定**SQL_BEST_ROWID**，那么通过SQLSpecialColumns返回的结果集将包含一个字段名为OID的行。用户可以使用这个特殊字段来重新选择fScope定义范围内的行。SELECT语

句保证结果中没有行或者有一行。有关相似函数的代码示例，请参考SQLTables。

如果***fColType***、***fScope*** 或***fNullable***参数指定了DBMaster不支持的字符，那么SQLSpecialColumns将返回一个没有行的rowset。通过***hstmt***并发调用SQLFetch或SQLExtendedFetch将返回***SQL_NO_DATA_FOUND***。

6.3 系统信息

用户可以使用SQLGetTypeInfo、SQLGetInfo以及SQLGetFunctions获取关于数据源的系统信息。这些ODBC函数在下面章节中会通过示例图示说明。

SQLGetTypeInfo

用户可以使用SQLGetTypeInfo来获取数据源支持的数据类型信息。

➲ **原型**

SQLGetTypeInfo：

```
RETCODE  SQLGetTypeInfo (HSTMT hstmt, SWORD fSqlType)
```

当为fSqlType赋值时，SQLGetTypeInfo在将会在结果集中返回相关的类型信息。用户可以使用SQLBindCol来为结果集绑定输出存储并使用SQLFetch来获取输出存储中的结果。fSqlType可以是SQL数据类型—**SQL_CHAR**、**SQL_DECIMAL**、**SQL_INTEGER**等。

结果集为：

字段号	字段名称	数据类型	说明
1	TYPE_NAME	VARCHAR(128)	NOT NULL
2	DATA_TYPE	SMALLINT	NOT NULL
3	PRECISION	INTEGER	
4	LITERAL_PREFIX	VARCHAR(128)	
5	LITERAL_SUFFIX	VARCHAR(128)	
6	CREATE_PARAMS	VARCHAR(128)	
7	NULLABLE	SMALLINT	NOT NULL
8	CASE_SENSITIVE	SMALLINT	NOT NULL
9	SEARCHABLE	SMALLINT	NOT

字段号	字段名称	数据类型	说明
			NULL
10	UNSIGNED_ATTRIBUTE	SMALLINT	
11	MONEY	SMALLINT	NOT NULL
12	AUTO_INCREMENT	SMALLINT	
13	LOCAL_TYPE_NAME	VARCHAR(128)	
14	MINIMUM_SCALE	SMALLINT	
15	MAXIMUM_SCALE	SMALLINT	

以下将使用带有**SQL_ALL_TYPES**的SQLGetTypeInfo作为fSqlType的值来获取数据源支持的所有数据类型。

➲ 示例

```
UCHAR  name[30], prefix[30], suffix[30], params[30], local name[30];
SWORD  type, nullable, case sen, searchable, unsign, money, auto inc;
SWORD  min_scale, max_scale;
UDWORD prec;
SDWORD len[15], retcode;

/* bind all columns */
retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  name,       30, &len[1]);
retcode = SQLBindCol(hstmt,  2, SQL C SHORT, &type,       0, &len[2]);
retcode = SQLBindCol(hstmt,  3, SQL_C_LONG,  &prec,       0, &len[3]);
retcode = SQLBindCol(hstmt,  4, SQL_C_CHAR,  prefix,     30, &len[4]);
retcode = SQLBindCol(hstmt,  5, SQL_C_CHAR,  suffix,     30, &len[5]);
retcode = SQLBindCol(hstmt,  6, SQL C CHAR,  params,     30, &len[6]);
retcode = SQLBindCol(hstmt,  7, SQL_C_SHORT, &nullable,   0, &len[7]);
retcode = SQLBindCol(hstmt,  8, SQL_C_SHORT, &case_sen,   0, &len[8]);
retcode = SQLBindCol(hstmt,  9, SQL_C_SHORT, &searchable, 0, &len[9]);
retcode = SQLBindCol(hstmt, 10, SQL C SHORT, &unsign,     0, &len[10]);
retcode = SQLBindCol(hstmt, 11, SQL_C_SHORT, &money,      0, &len[11]);
```

```
retcode = SQLBindCol(hstmt, 12, SQL_C_SHORT, &auto_inc,   0, &len[12]);
retcode = SQLBindCol(hstmt, 13, SQL C CHAR,  local name, 30, &len[13]);
retcode = SQLBindCol(hstmt, 14, SQL_C_SHORT, &min_scale,  0, &len[14]);
retcode = SQLBindCol(hstmt, 15, SQL_C_SHORT, &max_scale,  0, &len[15]);

/* tell odbc driver to get all type information                          */
printf("tell odbc driver to get all SQL type information \n");
SQLGetTypeInfo(hstmt,SQL_ALL_TYPES);

/* fetch all type information                                             */
do {
    retcode = SQLFetch(hstmt);
    switch (retcode)
          {
           case SQL_SUCCESS_WITH_INFO:
           case SQL_SUCCESS:
                print type info(); /* print type info such as name,type,*/
                                   /* prec, prefix, ...             */
                break;
           case SQL_NO_DATA_FOUND:
                break;
           default:
                print error
          }
   }while (retcode != SQL_NO_DATA_FOUND);
```

SQLGetInfo

用户可以使用SQLGetInfo来获取数据源的一般信息。

➲ **原型**

SQLGetInfo：

```
RETCODE  SQLGetInfo (
                     HDBC       hdbc,
                     UWORD      fInfoType,
                     PTR        rgbInfoValue,
                     SWORD      cbInfoValueMax,
                     SWORD FAR *pcbInfoValue)
```

在fInfoType中赋值代表了用户想知道的信息类型，并且给出输出存储rgbInfoValue和其存储大小ValueMax。SQLGetInfo将在rgbInfoValue中返回获取的信息，并在pcbInfoValue中返回获取信息的大小。

➲ **示例 1**

检查数据源是否支持字符串函数CONCAT：

```
UDWORD  bitmask;
SDWORD  retcode;
retcode = SQLGetInfo(hdbc, SQL_STRING_FUNCTIONS, (PTR) &bitmask,
                     sizeof(bitmask), NULL);
if (bitmask & SQL FN STR CONCAT)
    printf ("the data source supports CONCAT\n");
else
    printf ("the data source does not support CONCAT\n");
```

如果用户想知道表允许的字段最大数，可以在示例2中尝试这些代码。

示例 2

检查表中所允许的字段最大数：

```
UWORD  maxNCol;
SDWORD retcode;
retcode = SQLGetInfo(hdbc, SQL_MAX_COLUMNS_IN_TABLE, (PTR) &maxNCol,
                     sizeof(maxNCol),NULL);
printf ("In this data source, a table can have %d columns at most\n",
        (int) maxNCol );
```

SQLGetFunctions

用户可以使用SQLGetFunctions来检查数据源支持什么ODBC函数。

➲ **原型**

SQLGetFunctions：

```
RETCODE  SQLGetFunctions (
                          HDBC hdbc,
                          UWORD fFunction,
                          UWORD FAR *pfExists)
```

输入参数fFunction指定了ODBC函数的种类。fFunction的值可以为**SQL_API_SQLCANCEL**、**SQL_API_SQLFETCH**、**SQL_PUTDATA**等，**SQLCancel**、**SQLFetch**、**SQLPutData**为所有ODBC函数。

例如，用户可以在fFunction中赋予参数**SQL_API_SQLCANCEL**来检查数据源是否支持SQLCancel。

在提交SQLGetFunctions后，为了确定ODBC函数的存在，用户可以在pfExists中检查Boolean值，pfExists是对于一个单独Boolean值或Boolean值列表的指示器。

➲ **示例 1**

检查数据源是否支持SQLExecDirect：

```
UWORD  fExecDirect;
```

```
SDWORD retcode;


retcode = SQLGetFunctions (hdbc, SQL_API_SQLEXECDIRECT, &fExecDirect);


if (fExecDirect)
   printf ("the data source supports SQLExecDirect\n");
else
   printf ("the data source does not support SQLExecDirect\n");
```

➲ **示例 2**

检查数据源是否支持SQLTables：

```
UWORD  fExecDirect;
SDWORD retcode;
retcode = SQLGetFunctions (hdbc, SQL_API_SQLTABLES, &fExecDirect);
if (fExecDirect)
   printf ("the data source supports SQLExecDirect\n");
else
   printf ("the data source does not support SQLExecDirect\n");
```

SQLGetDiagRec

用户可以使用ODBC的SQLGetDiagRec()函数来获得返回代码和信息。

6.4 程序信息

用户可以使用SQLProcedureColumns和SQLProcedures来获取存储过程信息，这些ODBC函数在下面的章节中会通过示例图示说明。

SQLProcedureColumns

用户可以使用SQLProcedureColumns来重新获得有关输入输出参数列表的信息，以及为指定程序产生结果集的定义字段内容。驱动器将把这些信息作为结果集返回。

➲ **原型**

SQLProcedureColumns：

```
RETCODE  SQLProcedureColumns (
                              HSTMT hstmt,
                              UCHAR *szProcQualifier,
                              SWORD   cbProcQualifier,
                              UCHAR  *szProcOwner,
                              SWORD   cbProcOwner,
                              UCHAR  *szProcName,
                              SWORD   cbProcName,
                              UCHAR  *szColumnName,
                              SWORD   cbColumnName)
```

被用作查找模式的字符串可存在于***szProcQualifier***、***szProcOwner***和***szProcName***中。这三个参数和它们相对应的字符串长度参数在SQL*Procedures*中以***cbProcQualifier***、***cbProcOwner***和***cbProcName***出现。

SQLProcedureColumns的参数为：

- *hstmt* — 找回结果的一个有效语法句柄。

- *szProcQualifier* — DBMaster不支持，应该为**NULL**或一个空的字符串。
- *cbProcQualifier* — szProcQualifier的长度，应该为0。
- *szProcOwner* — 指出所有者名，此所有者为创建该程序的用户。它可以是一个用作查找模式的字符串或是**NULL**值。使用**NULL**值来显示所有用户。
- *cbProcOwner* — szProcOwner的长度或**SQL_NTS**。
- *szProcName* — 指出程序名。它可以是一个用作查找模式的字符串或**NULL**值。使用**NULL**值来显示所有程序。
- *cbProcName* — szProcName的长度或**SQL_NTS**。
- *szColumnName* — 指出字段名称。它可以是一个用作查找模式的字符串或**NULL**值。使用**NULL**值来显示所有字段。
- *cbColumnName* — szColumnName的长度。

返回一个由以下字段组成的结果集：

字段号	字段名称	数据类型	说明
1	PROCEDURE_QUALIFIER	VARCHAR(128)	
2	PROCEDURE_OWNER	VARCHAR(128)	
3	PROCEDURE_NAME	VARCHAR(128)	NOT NULL
4	COLUMN_NAME	VARCHAR(128)	NOT NULL
5	COLUMN_TYPE	SMALLINT	NOT NULL
6	DATA_TYPE	SMALLINT	NOT NULL
7	TYPE_NAME	VARCHAR(128)	NOT NULL
8	PRECISION	INTEGER	
9	LENGTH	INTEGER	
10	SCALE	SMALLINT	
11	RADIX	SMALLINT	
12	NULLABLE	SMALLINT	NOT NULL
13	REMARK	VARCHAR(254)	

下例使用SQLProcedureColumns获取关于数据库用户Tom的存储过程**employee**的所有信息的结果集。

➲ 示例

```
UCHAR  catalog[30], schema[30], procName[30], colName[30];
UCHAR  typeName[30], remark[30];
SWORD  colType, dataType, scale, radix, nullable;
UDWORD prec, length, len[13];
SDWORD retcode;
/* bind all columns                                                    */
retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  catalog,   30, &len[1]);
retcode = SQLBindCol(hstmt,  2, SQL_C_CHAR,  schema,    30, &len[2]);
retcode = SQLBindCol(hstmt,  3, SQL C CHAR,  procName,  30, &len[3]);
retcode = SQLBindCol(hstmt,  4, SQL C CHAR,  colName,   30, &len[4]);
retcode = SQLBindCol(hstmt,  5, SQL_C_SHORT, &colType,   0, &len[5]);
retcode = SQLBindCol(hstmt,  6, SQL_C_SHORT, &dataType,  0, &len[6]);
retcode = SQLBindCol(hstmt,  7, SQL C CHAR,  typeName,  30, &len[7]);
retcode = SQLBindCol(hstmt,  8, SQL C LONG,  &prec,      0, &len[8]);
retcode = SQLBindCol(hstmt,  9, SQL_C_LONG,  &length,    0, &len[9]);
retcode = SQLBindCol(hstmt, 10, SQL_C_SHORT,  &scale,    0, &len[10]);
retcode = SQLBindCol(hstmt, 11, SQL C SHORT,  &radix,    0, &len[11]);
retcode = SQLBindCol(hstmt, 12, SQL C SHORT, &nullable,  0, &len[12]);
retcode = SQLBindCol(hstmt, 13, SQL_C_CHAR,  remark,    30, &len[13]);
retcode = SQLProcedureColumns(hstmt, NULL, 0, "Tom", SQL_NTS,
                              "employee", SQL NTS, NULL, 0);
while ((retcode = SQLFetch(hstmt)) == SQL_SUCCESS)
      { /* print out each column's content in the result set */
       printf("column 1 :  procedure qualifier = %s\n", catalog);
       printf("column 2 :  procedure owner = %s\n", schema);
       printf("column 3 :  procedure name = %s\n", procName);
       printf("column 4 :  column name = %s\n", colName);
       printf("column 5 :  column type = %d\n", colType);
       printf("column 6 :  data type = %d\n", dataType);
```

```
       printf("column 7 :  type name = %s\n", typeName);
       printf("column 8 :  precision = %d\n", prec);
       printf("column 9 :  length = %d\n", length);
       printf("column 10 : scale = %d\n", scale);
       printf("column 11 : radix = %d\n", radix);
       printf("column 12 : nullable = %d\n", nullable);
       printf("column 13 : remark = %s\n", remark);
      }
...
```

SQLProcedures

用户可以使用SQLProcedures获取存储在数据源中的存储过程名称列表。

➲ **原型**

SQLProcedures：

```
RETCODE SQLProcedures (
                        HSTMT   hstmt,
                        UCHAR   *szProcQualifier,
                        SWORD   cbProcQualifier,
                        UCHAR   *szProcOwner,
                        SWORD   cbProcOwner,
                        UCHAR   *szProcName,
                        SWORD   cbProcName);
```

hstmt是一个有效的语法句柄。szTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableName所有定义与**SQLProcedureColumns**相似。SQLProcedures返回作为标准结果集的结果，顺序为PROCEDURE_QUALIFIER、PROCEDURE_OWNER和PROCEDURE_NAME。

下表列出了结果集中的字段：

字段号	字段名称	数据类型	说明
1	PROCEDURE_QUALIFIER	VARCHAR(128)	
2	PROCEDURE_OWNER	VARCHAR(128)	
3	PROCEDURE_NAME	VARCHAR(128)	NOT NULL
4	NUM_INPUT_PARAMS	N/A	
5	NUM_OUTPUT_PARAMS	N/A	
6	NUM_RESULT_SETS	N/A	
7	REMARKS	VARCHAR(254)	
8	PROCEDURE_TYPE	SMALLINT	

下例将显示如何使用SQLProcedures来获取由**Tom**创建的所有存储过程。如果用户想要在数据库中重新获取所有存储过程，可以在程序所有者的字段中使用一个null值。

➲ **示例**

```
UCHAR  catalog[30], schema[30], procName[30];
UCHAR  remark[30];
SWORD  type;
SDWORD len[5];
SDWORD retcode;
/* bind all columns                                                    */
retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  catalog,  30, &len[1]);
retcode = SQLBindCol(hstmt,  2, SQL C CHAR,  schema,   30, &len[2]);
retcode = SQLBindCol(hstmt,  3, SQL_C_CHAR,  procName, 30, &len[3]);
retcode = SQLBindCol(hstmt,  7, SQL_C_CHAR,  remark,   30, &len[4]);
retcode = SQLBindCol(hstmt,  8, SQL_C_SHORT, &type,     0, &len[5]);
retcode = SQLProcedures(hstmt, NULL, 0,
                              "Tom", SQL_NTS,
                          NULL, 0);
while ((retcode = SQLFetch(hstmt)) == SQL_SUCCESS)
      { /* print out each column's content in the result set */
       printf("column 1 :  procedure qualifier = %s\n", catalog);
```

```
    printf("column 2 :  procedure owner = %s\n", schema);
    printf("column 3 :  procedure name = %s\n", procName);
    printf("column 7 :  remark = %s\n", remark);
    printf("column 8 :  type = %d\n", type);
   }
```

7 事务控制

本章将描述事务和检查点的概念以及它们的特性，还将介绍如何使用ODBC函数来结束一个事务以及为事务控制设置选项。

在本章您将学习到如下内容：

- 使用函数*SQLSetConnectOption*和*SQLGetConnectOption*来设置和使用两种不同的提交模式：自动提交和手动提交。
- 使用函数*SQLTransact*终止一个事务，同时解释当事务终止时发生的各种影响。

7.1 事务和保存点

事务是一个或多个SQL语句的序列，这个序列组成一个逻辑工作单元。事务中的每个SQL语句执行其中的一部分任务，是整个任务不可或缺的必要组成部分。只有当事务中的所有SQL语句都执行成功时，任务才被完成。

➲ **管理银行账户中的存款，程序应该执行以下操作：**

1. 查询账户表，确保账户名称有效。
2. 查询分支表，确保分支号正确。
3. 查询出纳表，检查出纳是否存在。
4. 为该笔存款向历史表中插入一笔记录。
5. 在账户表中更新该账户名称的余额，然后为该笔存款增加金额。
6. 在出纳表中更新该出纳的余额。
7. 在分支表中更新该分支的余额。

注意 *这7个操作组成一个完整的事务，每个操作都是一个SQL语句。如果任意一个语句失败，整个事务都要取消，否则会产生数据不一致的情况。*

➲ **一个事务的一般流程如下：**

1. 开始一个事务。
2. 执行语句。
3. 如果任何语句失败，回滚。
4. 如果所有语句都成功，提交。

当连接到DBMaster，一个事务便自动产生了。用户可以根据自己的需要执行任意多个SQL语句。在处理完所有SQL语句后，提交整个事务，同时包括那些由DML操作(INSERT、DELETE或UPDATE)的所有改变，调用ODBC函数SQLTransact并带有参数**SQL_COMMIT**。另一方面，如果您想中止一个任务，可以调用ODBC函数SQLTransact带有参数**SQL_ROLLBACK**。一个事务停止后，DBMaster会自动开始一个新的事务。

有时如果事务非常长，用户可以使用检查点把长事务分割成多个部分以便管理。检查点是一个逻辑标记用来在事务中的某个特定点进行声明。使用检查点，就可以撤销某个特定点之后的所有更新，而不用撤销整个事务。

例如，如果您执行一个由15条SQL语句组成的事物，您在第10条和第11条语句间标记了一个检查点，如果在执行第12条语句时发生错误，那么可以回滚到检查点。然后您只需修改错误发生的语句，重做第11和第12条语句便可，而不用重做当前事务中的所有语句。

➲ **示例**

```
statement 1;
...
statement 5;
SAVEPOINT SVP1;      -> point A: define the first savepoint
statement 6;
...
statement 10;
SAVEPOINT SVP2;      -> point B: define the second savepoint
statement 11;
statement 12;        -> error occurs
ROLLBACK TO SVP2;    -> point C: when error occurs, rollback to nearest
savepoint
/* at this point, all the statements before SVP2 are preserved */
/* only statement 11 and 12 need to be re-executed.            */

statement 13;

statement 14;

statement 15;
```

```
COMMIT WORK;          -> if all statements are ok, commit the transaction
```

在这个例子中，我们可以看到检查点是如何帮助我们管理长事务的。在DBMaster中，您可以在一个事务中最多定义32个检查点。在事务终止后，所有定义的检查点都被清除。

注意，事务中的检查点ID必须唯一，例如：如果在点A定义了一个名为***SVP1***的检查点，那么就不能在点B再定义一个名为***SVP1***的检查点。另一个需要注意的方面是，当回滚到前面定义的检查点时，那么该点之后的所有保存点都将取消。

例如，上例中的点C，如果回滚到检查点***SVP1***，那么***SVP2***将被取消而不能再使用。然而，您可以定义一个名为***SVP2***的新检查点。

7.2 终止一个事务

如前面章节介绍的，用户可以使用SQLTransact提交或回滚事务。

➲ **原型**

SQLTransact：

```
RETCODE SQLTransact(
                    HENV        henv,
                    HDBC        hdbc,
                    UWORD       fType);
```

其中fType是**SQL_COMMIT**或**SQL_ROLLBACK**，正如他们的名字所暗示的，**SQL_COMMIT**提交事务，而 **SQL_ROLLBACK**回滚事务。

在DBMaster中，除非在事务终止后，连接选项**SQL_CB_MODE**的值设为**SQL_CB_PRESERVE**。否则，用户回滚到定义的检查点，那么当前连接句柄中所有与语句句柄关联的未决结果都将清除。

➲ **示例**

```
SQLAllocEnv(&henv);
SQLAllocConnect(henv,&hdbc);
/* connect to a database */
SQLConnect(hdbc, ...)
SQLAllocStmt(hdbc, &hstmt1);
SQLAllocStmt(hdbc, &hstmt2);
..
/* fetch one tuple from account table */
SQLExecDirect(hstmt1, "select * from account", SQL NTS);
SQLBindCol(hstmt1, 1, ....)
SQLBindCol(hstmt1, 2, ....)
SQLFetch(hstmt1);
/* fetch one tuple from branch table */
```

```
SQLExecDirect(hstmt2, "select * from branch", SQL_NTS);
SQLBindCol(hstmt2, 1, ....)
SQLBindCol(hstmt2, 2, ....)
SQLFetch(hstmt2);
/* Commit the transaction */
SQLTransact(henv, hdbc, SQL_COMMIT);
```

当提交了事务，与hstmt1和hstmt2相关联的结果集中未获取的数据将被清除（假设账户表和分支表中都多于一行数据）。

7.3 自动提交和手动提交

一般情况，应用程序需要控制事务的终止，这样就需要手动提交模式。

ODBC定义了许多连接选项，其中一个便是**SQL_AUTOCOMMIT**。该连接选项指示是否开启自动提交模式。**SQL_AUTOCOMMIT**的默认值是开启，这意味着每个语句都是自动提交的。

➲ **示例 1**

开始事务处理，使用ODBC函数SQLSetConnectOption关闭**SQL_AUTOCOMMIT**选项。

```
SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF)
```

调用该函数后，用户可控制提交动作了。如果自动提交模式为关闭，且当用户调用SQLDisconnect时，事务未提交，DBMaster会回滚该事物并返回警告。

➲ **示例 2**

要获取当前的自动提交模式值，可使用SQLGetConnectOption：

```
SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &optVal);
```

如果optVal中的选项值是**SQL_AUTOCOMMIT_ON**，那么在每个SQL语句成功执行后，都会自动被提交。

8 ODBC 3.0函数

目前在DBMaster中内构的应用程序接口是与ODBC3.0兼容的。为了与ODBC3.0兼容，在添加了一些新函数和禁止使用一些旧函数的同时，还修改了部分函数。因此，一些函数行为可能与之前版本有所不同，一些函数可能不能使用。有关ODBC3.0函数的完整描述和如何使用，请参考*微软ODBC3.0程序员参考手册*。

8.1 禁止使用的函数

下面的函数或函数参数值是DBMaster API (ODBC 3.0)中禁止使用的。DBMaster目前保留那些函数以备向后兼容，但并不保证以后的版本一定会出现这些函数。

- SQLAllocConnect—SQLAllocConnect 已经被函数SQLAllocHandle取代，HandleType为SQL_HANDLD_DBC。在以后版本中用户要使用SQLAllocHandle。
- SQLAllocEnv—SQLAllocEnv已经被函数SQLAllocHandle取代，HandleType为SQL_HANDLD_ENV 。在以后版本中用户要使用SQLAllocHandle。
- SQLAllocStmt—SQLAllocStmt已经被函数SQLAllocHandle取代，HandleType为SQL_HANDLD_STMT。在以后版本中用户要使用SQLAllocHandle 。
- SQLColAttributes—SQLColAttribute已经取代了SQLColAttributes函数。在以后版本中，用户要使用SQLColAttribute而不是SQLColAttributes。
- SQLExtendedFetch—SQLFetchScroll已经取代了SQLExtendedFetch。在以后版本中，用户要使用SQLFetchScroll而不是SQLExtendedFetch。
- SQLFreeConnect—SQLFreeConnect已经被SQLFreeHandle取代，HandleType为SQL_HANDLE_DBC。在以后版本中，用户要使用SQLFreeHandle。
- SQLFreeEnv—SQLFreeEnv已经被SQLFreeHandle取代，HandleType为SQL_HANDLE_ENV。在以后版本中，用户要使用SQLFreeHandle。
- SQLFreeStmt—SQLFreeStmt的选项参数的SQL_DROP值已经被SQLFreeHandle取代，HandleType为SQL_HANDLE_STMT。在以后版本中，用户要使用SQLFreeHandle。

- SQLGetConnectOption—SQLGetConnectAttr已经取代了SQLGetConnectOption函数。在以后版本中，用户要使用SQLGetConnectAttr而不是SQLGetConnectOption。
- SQLGetStmtOption—SQLGetStmtAttr已经取代了SQLGetStmtOption函数。在以后版本中，用户要使用SQLGetStmtAttr 而不是SQLGetStmtOption 。
- SQLSetConnectOption-- SQLSetConnectAttr已经取代了SQLSetConnectOption函数。在以后版本中，用户要使用SQLSetConnectAttr而不是SQLSetConnectOption 。
- SQLSetPos—SQLSetPos函数的fOption参数的SQL_ADD值已经被SQLBulkOperations函数的Operation的SQL_ADD值取代。在以后版本中，用户要使用SQLBulkOperations而不是SQLSetPos。
- SQLSetStmtOption—SQLSetStmtAttr已经取代了SQLSetStmtOption函数。在以后版本中，用户要使用SQLSetStmtAttr ，而不是SQLSetStmtOption 。
- SQLTransact—SQLTransact已经被SQLEndTran函数所取代。在以后版本中，用户要使用SQLEndTran而不是SQLTransact。

8.2 修改过的函数

下面的函数在DBMaster API(ODBC 3.0)中已经修改。这些函数的行为与DBMaster3.01和早期的API版本(ODBC 2.0)会有些不同。不过，如果您使用的DBMaster3.5版本之前的客户端软件，那么这些函数的行为将保持不变。

SQLCancel

SQLCancel函数被DBMaster API(ODBC 3.0)完全支持。在之前的DBMaster版本(ODBC 2.0)中，当一个语句调用SQLCancel函数而没有执行任何处理时，它的效果等同于调用SQLFreeStmt带有SQL_CLOSE选项。在DBMaster中，调用SQLCancel函数而没有执行任何处理是无效的。如果这时有游标开启并想关闭它，您应该调用SQLCloseCursor。

SQLColumns

SQLColumns函数被DBMaster API(ODBC 3.0)完全支持。现在，无论客户端使用的是ODBC2.0或3.0 API，函数SQLColumns都将返回18个字段。下表列出了当前DBMaster和之前DBMaster版本对调用该函数会返回的字段名称。

DBMaster 3.5-4.x (ODBC 3.0)	DBMaster 2.0x, 3.0x (ODBC 2.0)
TABLE_CAT	TABLE_QUALIFIER
TABLE_SCHEM	TABLE_OWNER
TABLE_NAME	TABLE_NAME
COLUMN_NAME	COLUMN_NAME
DATA_TYPE	DATA_TYPE
TYPE_NAME	TYPE_NAME
COLUMN_SIZE	PRECISION
BUFFER_LENGTH	LENGTH
DECIMAL_DIGITS	SCALE
NUM_PREC_RADIX	RADIX
NULLABLE	NULLABLE

DBMaster 3.5-4.x (ODBC 3.0)	DBMaster 2.0x, 3.0x (ODBC 2.0)
REMARKS	—
COLUMN_DEF	—
SQL_DATA_TYPE	—
SQL_DATETIME_SUB	—
CHAR_OCTET_LENGTH	—
ORDINAL_POSITION	—
IS_NULLABLE	—

SQLFetch

SQLFetch函数被DBMaster API(ODBC 3.0)完全支持。当前的DBMaster，SQLFetch函数可支持带有多行的rowsets。早期的版本只支持单行的操作。

SQLGetData

SQLGetData函数被DBMaster API(ODBC 3.0)完全支持。当前的DBMaster，SQLGetData函数可支持带有多行的rowsets。早期的版本只支持单行的操作。

SQLGetFunctions

SQLGetFunctions函数被DBMaster API(ODBC 3.0)完全支持。在DBMaster中，用户可调用SQLGetFunctions函数，*FunctionId*参数的值可以是SQL_API_ODBC3_ALL_FUNCTIONS或SQL_API_ALL_FUNCTIONS。SQL_API_ODBC3_ALL_FUNCTIONS被ODBC3.0应用程序用来决定是支持ODBC3.0还是早期版本，而SQL_API_ALL_FUNCTIONS被ODBC2.0应用程序用来决定是支持ODBC2.0或早期版本。

如果*FunctionId*的值是SQL_API_ODBC3_ALL_FUNCTIONS，*SupportedPtr*会指向一个用来决定支持ODBC3.0还是早期版本的4000-bit位图。ODBC3.0或2.0中都可以使用SQL_API_ODBC3_ALL_FUNCTIONS。如果*FunctionID*的值是

SQL_API_ALL_FUNCTIONS，那么*SupportedPtr*将返回一个有100个元素的数组。可使用该数组来决定支持ODBC2.0还是早期版本。

SQLGetInfo

SQLGetInfo函数被DBMaster API(ODBC 3.0)完全支持。

SQLProcedureColumns

SQLProcedureColumns函数被DBMaster API(ODBC 3.0)完全支持。无论客户端使用的是ODBC2.0或3.0 API，DBMaster SQLProcedureColumns函数将会返回19个字段。下表列出了当前DBMaster和之前DBMaster版本调用该函数会返回的字段名称。

DBMaster 3.5-4.x (ODBC 3.0)	DBMaster 2.0x, 3.0x (ODBC 2.0)
PROCEDURE_CAT	PROCEDURE _QUALIFIER
PROCEDURE _SCHEM	PROCEDURE _OWNER
PROCEDURE _NAME	PROCEDURE _NAME
COLUMN_NAME	COLUMN_NAME
COLUMN_TYPE	COLUMN_TYPE
DATA_NAME	DATA_NAME
TYPE_NAME	TYPE_NAME
COLUMN_SIZE	PRECISION
BUFFER_LENGTH	LENGTH
DECIMAL_DIGITS	SCALE
NUM_PREC_RADIX	RADIX
NULLABLE	NULLABLE
REMARKS	REMARK
COLUMN_DEF	—
SQL_DATA_TYPE	—
SQL_DATETIME_SUB	—
CHAR_OCTET_LENGTH	—
ORDINAL_POSITION	—
IS_NULLABLE	—

8.3 新函数

本节列出了DBMaster中的所有新函数、每个函数支持的选项以及这些是否被ODBC3.0标准完全支持还是部分支持。

SQLAllocHandle

SQLAllocHandle函数被ODBC API(ODBC 3.0)完全支持。它是分配环境、连接、语句或指示句柄的一般函数，它替代了ODBC2.0中的SQLAllocConnect, SQLAllocEnv和SQLAllocStmt。

➲ 原型

SQLAllocHandle：

```
RETCODE SQLAllocHandle(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   InputHandle,
                    SQLHANDLE * OutputHandlePtr);
```

下列使用SQLAllocHandle函数分配环境、连接和语句句柄。

➲ 示例

```
SQLHANDLE  henv, hdbc, hstmt;
SQLRETURN  retcode;
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(void*)SQL OV ODBC3, 0);
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
retcode = SQLConnect(hdbc, (SQLCHAR*)  "test", SQL_NTS,
                              (SQLCHAR*) "Sysadm", SQL_NTS,
                              (SQLCHAR*) "coffee", SQL NTS);
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
…
```

SQLBulkOperations

SQLBulkOperations函数被ODBC API(ODBC 3.0)完全支持。SQLBulkOperations函数执行批量的插入和书签操作，包括更新、删除和通过书签操作获取记录等。

➲ **原型**

SQLBulkOperations：

```
RETCODE SQLBulkOperations(
                    SQLHSTMT     StatementHandle,
                    SQLUSMALLINT Operation);
```

下表列出了SQLBulkOperations函数的选项以及是否支持这些选项。

操作	支持?
SQL_ADD	Y
SQL_UPDATE_BY_BOOKMARK	Y
SQL_DELETE_BY_BOOKMARK	Y
SQL_FETCH_BY_BOOKMARK	Y

下例使用SQLBulkOperations函数，带有SQL_ADD选项来向表Employee中插入2行记录。

➲ **示例**

```
SQLRETCODE  retcode;
SQLHANDLE   hstmt;
SQLUINTEGER CustID[2];
SQLCHAR     Name[2][18], Address[2][100], Phone[2][11];
SQLINTEGER  CustIDInd[2], NameInd[2], AddressInd[2], PhoneInd[2];
/* execute a query  */
    retcode = SQLExecDirect(hstmt, (SQLCHAR *)"insert into Customers
                        values(?,?,?,?)", SQL_NTS);
/* set necessary statement attributes */
retcode = SQLSetStmtAttr(hstmt, SQL ATTR CURSOR TYPE, (void
                        *)SQL_CURSOR_DYNAMIC, SQL_IS_INTEGER);
```

```
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_CONCURRENCY, (void
                        *)SQL CONCUR LOCK, SQL IS INTEGER);

retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (void
                        *)2,SQL_IS_INTEGER);

/* binding columns */

retcode = SQLBindCol(hstmt, 1, SQL C LONG, CustID,  0,  CustIDInd);

retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, Name,    18,  NameInd);

retcode = SQLBindCol(hstmt, 3, SQL_C_CHAR, Address, 100, AddressInd);

retcode = SQLBindCol(hstmt, 4, SQL C CHAR, Phone,   11,  PhoneInd);

/* execute a query  */

SQLExecDirect(hstmt, (SQLCHAR *) "select * from Customers", SQL_NTS);

/* prepare data for insertion */

CustID[0] = 1;

CustID[1] = 2;

strcpy((char *)Name[0], "Jackson");

strcpy((char *)Name[1], "Clinton");

strcpy((char *)Address[0], "107 Castlewood, Cary, NC11256");

strcpy((char *)Address[1], "305 N. Frances St., Madison, WI95868");

strcpy((char *)Phone[0], "02-78923423");

strcpy((char *)Phone[1], "03-7893933");

NameInd[0] = strlen((char*)Name[0]);

NameInd[1] = strlen((char*)Name[1]);

AddressInd[0] = strlen((char*)Address[0]);

AddressInd[1] = strlen((char*)Address[1]);

PhoneInd[0] = strlen((char*)Phone[0]);

PhoneInd[1] = strlen((char*)Phone[1]);

/* insert data */

retcode = SQLBulkOperations(hstmt, SQL_ADD);

…
```

SQLCloseCursor

SQLCloseCursor函数被ODBC API（ODBC 3.0）完全支持。它用来关闭语句总的游标，取消未决的结果。

➲ **原型**

SQLCloseCursor：

```
RETCODE SQLCloseCursor(
                    SQLHSTMT     StatementHandle);
```

SQLColAttribute

SQLColAttribute函数被ODBC API（ODBC 3.0）完全支持。
SQLColAttribute函数会返回结果集中有关字段的信息，或者为字符串32位的值或整型值。

➲ **原型**

SQLColAttribute：

```
RETCODE SQLColAttribute(
                    SQLHSTMT      StatementHandle,
                    SQLUSMALLINT  ColumnNumber,
                    SQLUSMALLINT  FieldIdentifier,
                    SQLPOINTER    CharacterAttributePtr,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT * StringLengthPtr,
                    SQLPOINTER    NumericAttributePtr);
```

下表列出该函数返回的描述符类型。

字段标示符	目的
SQL_DESC_AUTO_UNIQUE_VALUE	指示一个字段是否为SERIAL类型，(SQL_TRUE) 或(SQL_FALSE)。
SQL_DESC_BASE_COLUMN_NAME	基础字段名称。

字段标示符	目的
SQL_DESC_BASE_TABLE_NAME	基础表名称。
SQL_DESC_CASE_SENSITIVE	用来指定对字段的比较和操作是否是大小写敏感的，（SQL_TRUE）或(SQL_FALSE)。
SQL_DESC_CATALOG_NAME	包含字段的表目录。
SQL_DESC_CONCISE_TYPE	日期、时间和间隔时间类型的简明结构。
SQL_DESC_COUNT	结果集中允许的字段数。
SQL_DESC_DISPLAY_SIZE	要显示字段的数据所需要的最多字符数。
SQL_DESC_FIXED_PREC_SCALE	决定字段是否有固定的精度和非零广度，（SQL_TRUE）或(SQL_FALSE)。
SQL_DESC_LABEL	字段标签或标题。如果字段没有标签，返回字段名称。
SQL_DESC_LENGTH	字符串或二进制数据类型的最大或实际长度。
SQL_DESC_LITERAL_PREFIX	DBMaster为字段所包含的数据类型的文字认定的前缀。
SQL_DESC_LITERAL_SUFFIX	DBMaster为字段所包含的数据类型的文字认定的后缀。
SQL_DESC_LOCAL_TYPE_NAME	数据类型的本地化（本机语言）名称，可能与常规名称不同。
SQL_DESC_NAME	字段同义字。如果字段没有同义字，将返回字段名。
SQL_DESC_NULLABLE	决定字段是否可以包含NULL值，(SQL_NULLABLE)或(SQL_NO_NULLS)。如果未知字段是否可以包含NULL值，返回SQL_NULLABLE_UNKNOWN。

字段标示符	目的
SQL_DESC_NUM_PREX_RADIX	包含2 - 如果SQL_DESC_TYPE是近似数值数据类型，并且SQL_DESC_PRECISION包含字节数。 10 - 如果SQL_DESC_TYPE是个确切的数值数据类型，并且SQL_DESC_PRECISION包含小数位数。 0 - 如果SQL_DESC_TYPE包含非数值数据类型。
SQL_DESC_OCTET_LENGTH	字符串或二进制数据类型的字节长度。
SQL_DESC_PRECISION	数值数据类型的精度。
SQL_DESC_SCALE	数值数据类型的广度。
SQL_DESC_SCHEMA_NAME	包含该字段的表模式。
SQL_DESC_SEARCHABLE	决定在WHERE语句中该字段是否可以使用比较运算符(SQL_PRED_SEARCHABLE)，是否可以使用除LIKE之外的所有比较运算符(SQL_PRED_BASIC)，或只能使用LIKE谓词(SQL_PRED_CHAR)，或WHERE子句中根本不能使用该字段。
SQL_DESC_TABLE_NAME	包含该字段的表名称。
SQL_DESC_TYPE	用来指定字段数据类型的数值。
SQL_DESC_TYPE_NAME	用来指定字段类型名称的字符串（依赖于数据源）。
SQL_DESC_UNNAMED	决定SQL_DESC_NAM中的值是否为字段名/同义字，(SQL_DESC_NAMED)或(SQL_DESC_UNNAMED)。
SQL_DESC_UNSIGNED	决定字段是否无正负之分，(SQL_TRUE) 或 (SQL_FALSE)。
SQL_DESC_UPDATABLE	描述结果集中的字段是否可以被更新（不是基础表中的字段）。

SQLCopyDesc

SQLCopyDesc函数被ODBC API（ODBC 3.0）完全支持。
SQLCopyDesc函数可将描述符信息从一个描述符句柄复制到另一个句柄。

➲ 原型

SQLCopyDesc：

```
RETCODE SQLCopyDesc(
                    SQLHDESC      SourceDescHandle,
                    SQLHDESC      TargetDescHandle);
```

在下例中，使用描述符操作复制PartInfo表的域到Backup表中。为此，复制hstmt1的IRD域到hstmt2的IPD域，hstmt1的ARD域到hstmt2的APD域。

➲ 示例

```
/* the structure of a record row */
typedef struct{
                        SQLINTEGER  PartID;
                        SQLINTEGER  PartIDInd;
                        UCHAR    Description[100];
                        SQLINTEGER  DescriptionInd;
                        DOUBLE      Price;
        SQLINTEGER  PriceInd;
}PartInfo;
PartInfo parts[20];
SQLHANDLE  hstmt1, hstmt2;
SQLHANDLE  ird1, ard1, ipd2, apd2;
SQLRETURN retcode;
/* get ARD and IRD of hstmt1 */
SQLGetStmtAttr(hstmt1, SQL_ATTR_APP_ROW_DESC, &ard1, 0, NULL);
```

```
SQLGetStmtAttr(hstmt1, SQL_ATTR_IMP_ROW_DESC, &ird1, 0, NULL);

/* get APD and IPD of hstmt2 */

SQLGetStmtAttr(hstmt2, SQL_ATTR_APP_PARAM_DESC, &apd2, 0, NULL);

SQLGetStmtAttr(hstmt2, SQL_ATTR_IMP_PARAM_DESC, &ipd2, 0, NULL);

/* set necessary statement attributes on hstmt1 */

SQLSetStmtAttr(hstmt1, SQL ATTR ROW BIND TYPE, (SQLPOINTER)sizeof(PartInfo),
                       0);

SQLSetStmtAttr(hstmt1, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)20, 0);

/* execute a select statement */

SQLExecDirect(hstmt1, (SQLCHAR *) "select * from PartInfo", SQL NTS);

/* binding columns */

SQLBindCol(hstmt1, 1, SQL_C_LONG, &parts[0].PartID,     0,
                       &parts[0].PartIDInd);

SQLBindCol(hstmt1, 2, SQL C CHAR, parts[0].Description, 100,
                       &parts[0].DescriptionInd);

SQLBindCol(hstmt1, 3, SQL_C_DOUBLE, &parts[0].Price,    0,
                       &parts[0].PriceInd);

/* calling SQLCopyDesc */

SQLCopyDesc(ard1, apd2);

SQLCopyDesc(ird1, ipd2);

/* prepare an insert statement on hstmt2 */

SQLPrepare(hstmt2, (SQLCHAR *) "insert into Backup values(?,?,?)", SQL_NTS);

retcode = SQLFetchScroll(hstmt1, SQL_FETCH_NEXT, 0);

while(retcode == SQL_SUCCESS){

                       SQLExecute(hstmt2);

                       retcode = SQLFetchScroll(hstmt1, SQL_FETCH_NEXT, 0);

}

…
```

SQLEndTran

SQLEndTran函数被ODBC API(ODBC 3.0)完全支持。SQLEndTran会请求DBMaster服务器为与连接到相关所有语句上的活跃操作执行提交或回滚，或与环境相关的连接。DBMaster支持的*CompletionType*参数值为SQL_COMMIT和SQL_ROLLBACK。

➲ 原型

SQLEndTran：

```
RETCODE SQLEndTran(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   Handle,
                    SQLSMALLINT CompletionType);
```

SQLFetchScroll

SQLFetchScroll函数被ODBC API(ODBC 3.0)完全支持。
SQLFetchScroll函数从结果集中的相对或绝对位置，或通过书签获取rowset数据，并返回所有绑定字段。对于*FetchOrientation*参数，DBMaster支持下列值：SQL_FETCH_NEXT、SQL_FETCH_PRIOR、SQL_FETCH_FIRST、SQL_FETCH_LAST、SQL_FETCH_ABSOLUTE、SQL_FETCH_BOOKMARK和SQL_FETCH_RELATIVE。

➲ 原型

SQLFetchScroll：

```
RETCODE SQLFetchScroll(
                    SQLHSTMT    StatementHandle,
                    SQLSMALLINT Handle,
                    SQLINTEGER  FetchOffset);
```

以下代码段显示了如何使用SQLFetchScroll函数带有SQL_NEXT选项来获取整个结果集。要获取行状态，用户需要使用与SQLExtendedFetch不同的SQLSetStmtAttr函数并带有SQL_ATTR_ROW_STATUS_PTR。

➲ 示例

```
#define LENGTH 18

#define ROWSET SIZE 5

HSTMT hstmt;

RETCODE   retcode;

UCHAR  empid[ROWSET_SIZE][LENGTH];

UCHAR  name[ROWSET SIZE][LENGTH];

FLOAT    salary[ROWSET_SIZE];

SQLINTEGER   empidInd[ROWSET_SIZE], nameInd[ROWSET_SIZE],
                      salaryInd[ROWSET SIZE];

SQLUSMALLINT status[ROWSET SIZE];

SQLUSMALLINT i;

/* set row status pointer */

retcode = SQLSetStmtAttr(hstmt, SQL ATTR ROW STATUS PTR, status, 0);

/* set rowset size */

retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, , (void
                      *)ROWSET_SIZE, SQL_IS_INTEGER);

/* execute a statement */

SQLExecDirect(hstmt, (SQLCHAR *) "SELECT * FROM EMPLOYEE", SQL NTS);

/* binding columns */

SQLBindCol(hstmt, 1, SQL_C_CHAR, empid, LENGTH, empidInd);

SQLBindCol(hstmt, 2, SQL C CHAR, name, LENGTH, nameInd);

SQLBindCol(hstmt, 3, SQL_C_FLOAT, salary, 0, salaryInd);

retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);

for (i = 0; i < ROWSET_SIZE; i++)

  {    if (status[i] == SQL ROW SUCCESS)

      {

        printf("tuple %ld  is - Employee ID : %s, Employee Name : %s,
                      Salary : %f \n", i+1, empid[i], name[i], salary[i]);

      }

    else {
```

```
                    printf("fetch tuple %ld error \n", i+1);
    }
  }
…
```

SQLForeignKeys

SQLForeignKeys函数被ODBC API(ODBC 3.0)完全支持。
SQLForeignKeys函数会返回指定表的外键列表，或参照指定表的其它表中的外键列表。

➲ **原型**

SQLForeignKeys：

```
RETCODE SQLForeignKeys(
                    SQLHSTMT    StatementHandle,
                    SQLCHAR *   PKCatalogName,
                    SQLSMALLINT NameLength1,
                    SQLCHAR *   PKSchemaName,
                    SQLSMALLINT NameLength2,
                    SQLCHAR *   PKTableName,
                    SQLSMALLINT NameLength3,
                    SQLCHAR *   FKCatalogName,
                    SQLSMALLINT NameLength4,
                    SQLCHAR *   FKSchemaName,
                    SQLSMALLINT NameLength5,
                    SQLCHAR *   FKTableName,
                    SQLSMALLINT  NameLength6);
```

下例使用2个表：ORDER (ORDERID, CUSTID, OPENDATE)和CUSTOMER (CUSTID, NAME, ADDRESS, PHONE)。

在*ORDER*表中，CUSTID指示该笔买卖与哪个顾客成交。表CUSTOMER中有外键参考ORDERID。

本例调用SQLForeignKeys来获取那些参照表ORDER主键的其它表中的外键。

➲ 示例

```
#define TAB_LEN 18

#define COL_LEN 18

/* Bind the columns that describe the primary and foreign keys */

SQLBindCol(hstmt, 3, SQL_C_CHAR, pkTable, TAB_LEN, &pkTableInd);

SQLBindCol(hstmt, 4, SQL_C_CHAR, pkCol,   COL_LEN, &pkColInd);

SQLBindCol(hstmt, 7, SQL_C_CHAR, fkTable, TAB_LEN, &fkTableInd);

SQLBindCol(hstmt, 8, SQL C CHAR, fkCol,   COL LEN, &fkColInd);

/* Get the names of columns in the primary key. */

retcode = SQLForeignKeys(hstmt, NULL, 0, NULL, 0, (SQLCHAR *) "ORDER",
                      SQL_NTS, NULL, 0, NULL, 0, NULL, 0);

while (retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO){

                      retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);

                      printf("Primary Table : %s, Primary Column : %s \n",
                      pkTable, pkCol);

                      printf("Foreign Table : %s, Foreign Column : %s \n",
                      fkTable, fkCol);

}

/* close the cursor */

SQLCloseCursor(hstmt);

…
```

SQLFreeHandle

SQLFreeHandle函数被ODBC API(ODBC 3.0)完全支持。
SQLFreeHandle函数是释放之前使用函数SQLAllocHandle分配的，并与环境、连接、语句或描述符句柄相关的资源。

➲ 原型

SQLFreeHandle：

```
RETCODE SQLFreeHandle(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   Handle);
```

下面代码段显示如何使用SQLFreeHandle函数释放环境句柄。

➲ 示例

```
SQLHANDLE   henv;
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
…
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

SQLGetConnectAttr

SQLGetConnectAttr函数被ODBC API（ODBC 3.0）部分支持。SQLGetConnectAttr获取一个数据库连接的属性。该函数替代了ODBC 2.0中的SQLGetConnectOption函数。

➲ 原型

SQLGetConnectAttr：

```
RETCODE SQLGetConnectAttr(
                    SQLHDBC      ConnectionHandle,
                    SQLINTEGER   Attribute,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER  *StringLengthPtr);
```

下表列出了该函数可获取的属性列表，以及这些属性是否被DBMaster支持。

属性	是否支持
SQL_ATTR_ACCESS_MODE	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_AUTO_IPD	Y
SQL_ATTR_AUTOCOMMIT	Y

属性	是否支持
SQL_ATTR_CONNECTION_TIMEOUT	Y
SQL_ATTR_CURRENT_CATALOG	Y
SQL_ATTR_LOGIN_TIMEOUT	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_ODBC_CURSORS	N
SQL_ATTR_PACKET_SIZE	N
SQL_ATTR_QUIET_MODE	N
SQL_ATTR_TRACE	N
SQL_ATTR_TRACEFILE	N
SQL_ATTR_TRANSLATE_LIB	N
SQL_ATTR_TRANSLATE_OPTION	N
SQL_ATTR_TXN_ISOLATION	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_MAX_ROWS	Y

SQLGetDescField

SQLGetDescField函数被ODBC API(ODBC 3.0)完全支持，SQLGetDescField获取一个描述符记录的单个域值。

➲ **原型**

SQLGetDescField：

```
RETCODE SQLGetDescField(
                    SQLHDESC     DescriptorHandle,
                    SQLSMALLINT  RecNumber,
                    SQLSMALLINT  FieldIdentifier,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```

下表列出了在DBMaster中用该函数可返回的描述符域。在表中，“G”代表获取，“S”代表设置，“I”代表无效，SQL_DESC_ARRAY_SIZE只支持值为1。

域描述符	ARD	APD	IRD	IPD
SQL_DESC_ALLOC_TYPE	G	G	G	G
SQL_DESC_ARRAY_SIZE	G/S	G/S	I	I
SQL_DESC_ARRAY_STATUS_PTR	G/S	G/S	G/S	G/S
SQL_DESC_BIND_OFFSET_PTR	G/S	G/S	I	I
SQL_DESC_BIND_TYPE	G/S	G/S	I	I
SQL_DESC_COUNT	G/S	G/S	G	G/S
SQL_DESC_ROW_PROCESSED_PTR	I	I	G/S	G/S
SQL_DESC_AUTO_UNIQUE_VALUE	I	I	G	I
SQL_DESC_BASE_COLUMN_NAME	I	I	G	I
SQL_DESC_BASE_TABLE_NAME	I	I	G	I
SQL_DESC_CASE_SENSITIVE	I	I	G	I
SQL_DESC_CATALOG_NAME	I	I	G	I
SQL_DESC_CONCISE_TYPE	G/S	G/S	G	G/S
SQL_DESC_DATA_PTR	G/S	G/S	I	I
SQL_DESC_DATETIME_INTERVAL_ CODE	G/S	G/S	G	G/S
SQL_DESC_DATETIME_INTERVAL_ PRECISION	G/S	G/S	G	G/S
SQL_DESC_DISPLAY_SIZE	I	I	G	I
SQL_DESC_FIXED_PREC_SCALE	I	I	G	G
SQL_DESC_INDICATOR_PTR	G/S	G/S	I	I
SQL_DESC_LABLE	I	I	G	I
SQL_DESC_LENGTH	G/S	G/S	G	G/S
SQL_DESC_LITERAL_PREFIX	I	I	G	I
SQL_DESC_LITERAL_SU FFIX	I	I	G	I
SQL_DESC_LOCAL_TYPE_NAME	I	I	G	G
SQL_DESC_NAME	I	I	G	G
SQL_DESC_NULLABLE	I	I	G	G
SQL_DESC_NUM_PREC_RADIX	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH_PTR	G/S	G/S	I	I
SQL_DESC_PARAMETER_TYPE	I	I	I	G/S
SQL_DESC_PRECISION	G/S	G/S	G	G/S
SQL_DESC_SCALE	G/S	G/S	G	G/S
SQL_DESC_SCHEMA_NAME	I	I	G	I
SQL_DESC_SEARCHABLE	I	I	G	I
SQL_DESC_TABLE_NAME	I	I	G	I

域描述符	ARD	APD	IRD	IPD
SQL_DESC_TYPE	G/S	G/S	G	G/S
SQL_DESC_TYPE_NAME	I	I	G	G
SQL_DESC_UNNAMED	I	I	G	I
SQL_DESC_UNSIGNED	I	I	G	G
SQL_DESC_UPDATABLE	I	I	G	I

下面代码段演示了如何使用SQLGetDescField函数获取字段信息。

➲ **示例**

```
#define LEN 19

SQLHANDLE   hstmt, ird;

SQLINTEGER  count, index;

UCHAR    colName[LEN], typeName[LEN];

SQLSMALLINT prec, scale;

SQLUINTEGER length;

/* execute the statement */

SQLExecDirect(hstmt, (SQLCHAR *) "select * from EMPLOYEE", SQL_NTS);

/* get the ird descriptors */

 SQLGetStmtAttr(hstmt, SQL ATTR IMP ROW DESC, &ird, SQL IS POINTER, NULL);

/* get the number of columns */

SQLGetDescField(ird, 0, SQL_DESC_COUNT, &count, 0, NULL);

for (index = 1; index <= count; index++)

{

        SQLGetDescField(ird, index, SQL_DESC_NAME, colName, LEN, NULL);

        SQLGetDescField(ird, index, SQL_DESC_TYPE_NAME, typeName, LEN, NULL);

        SQLGetDescField(ird, index, SQL DESC PRECISION, &prec, 0, NULL);

        SQLGetDescField(ird, index, SQL DESC SCALE, &scale, 0, NULL);

        SQLGetDescField(ird, index, SQL_DESC_LENGTH, &length, 0, NULL);

        printf("Column No : %d, Name : %s, Type : %s, Length : %d,
Precision : %d, Scale : %d \n", index, colName, typeName, length, prec,
scale);

}
```

```
...
```

SQLGetDescRec

SQLGetDescRec函数被ODBC API（ODBC 3.0）完全支持。
SQLGetDescRec函数返回描述符记录中的多个域的设置或域值。

➲ 原型

SQLGetDescRec：

```
RETCODE SQLGetDescRec(
                    SQLHDESC      DescriptorHandle,
                    SQLSMALLINT   RecNumber,
                    SQLCHAR*      Name,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  StringLengthPtr,
                    SQLSMALLINT*  TypePtr,
                    SQLSMALLINT*  SubTypePtr,
                    SQLINTEGER*   LengthPtr,
                    SQLSMALLINT*  PrecisionPtr,
                    SQLSMALLINT*  ScalePtr,
                    SQLSMALLINT*  NullablePtr);
```

下面代码段演示了如何使用SQLGetDescRec函数获取字段信息。

➲ 示例

```
#define LEN 19
SQLHANDLE   hstmt, ird;
SQLINTEGER  count, index;
UCHAR    colName[LEN];
SQLSMALLINT type, prec, scale, nullable;
SQLUINTEGER length;
/* execute the statement */
SQLExecDirect(hstmt, (SQLCHAR *) "select * from EMPLOYEE", SQL_NTS);
```

```
/* get the ird descriptors */

 SQLGetStmtAttr(hstmt, SQL ATTR IMP ROW DESC, &ird, SQL IS POINTER, NULL);

/* get the number of columns */

SQLGetDescField(ird, 0, SQL_DESC_COUNT, &count, 0, NULL);

for (index = 1; index <= count; index++)

{

SQLGetDescRec(ird, index, colName, LEN, NULL, &type, NULL, &length, &prec,
                      &scale, &nullable);

printf("Column No. : %d, Name : %s, Type : %d, Length : %d, Precision : %d,
                      Scale : %d, Nullable : %d \n", index, colName, type,
                      length, prec, scale, NULL);

}

…
```

SQLGetDiagField

SQLGetDiagField函数被ODBC API(ODBC 3.0)部分支持。
SQLGetDiagField函数从指定句柄的诊断数据结构的记录中返回域的当前值。该域包括错误、警告和状态信息。

➲ **原型**

SQLGetDiagField：

```
RETCODE SQLGetDiagField(

                  SQLSMALLINT   HandleType,

                  SQLHANDLE     Handle,

                  SQLSMALLINT   RecNumber,

                  SQLSMALLINT   DiagIdentifier,

                  SQLPOINTER    DiagInfoPtr,

                  SQLSMALLINT   BufferLength,

                  SQLSMALLINT*  StringLengthPtr);
```

下表显示了诊断数据结构需要的域标示符以及是否被DBMaster支持。

诊断描述符	是否支持
SQL_DIAG_CURSOR_ROW_COUNT	N
SQL_DIAG_DYNAMIC_FUNCTION	N
SQL_DIAG_DYNAMIC_FUNCTION_CODE	N
SQL_DIAG_NUMBER	Y
SQL_DIAG_RETURNCODE	Y
SQL_DIAG_ROW_COUNT	Y
SQL_DIAG_CLASS_ORIGIN	Y
SQL_DIAG_CONNECTION_NAME	Y
SQL_DIAG_MESSAGE_TEXT	Y
SQL_DIAG_SERVER_NAME	Y
SQL_DIAG_SQLSTATE	Y
SQL_DIAG_SUBCLASS_ORIGIN	Y
SQL_DIAG_COLUMN_NUMBER	N
SQL_DIAG_NATIVE	N
SQL_DIAG_ROW_NUMBER	N

下面代码段显示如何使用SQLGetDiagField函数和SQLGetDiagRec函数来获取错误信息。

➲ **示例**

```
SQLHANDLE   hstmt;

RETCODE  retcode;

SQLINTEGER  num, index, nativerc;

UCHAR    state[20], errmsg[200];

SQLSMALLINT retLen;

/* execute a statement */

retcode = SQLExecDirect(hstmt, (SQLCHAR *) "SELECT * FROM EMPLOYEE",
                      SQL_NTS);

/* if error, get error info */

if (retcode != SQL_SUCCESS){

        SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_NUMBER, &num, 0,
NULL);
```

```
        for (index = 1; index <= num; index++){
             SQLGetDiagRec(SQL HANDLE STMT, hstmt, index, state, &nativerc,
errmsg, 200, &retLen);
        }
}
….
```

SQLGetDiagRec

SQLGetDiagRec函数被ODBC API(ODBC 3.0)完全支持。SQLGetDiagRec函数返回诊断记录中多个常用域的值，包括SQLSTATE、标准错误代码以及诊断信息文本。DBMaster支持*HandleType*参数为下列值：SQL_HANDLE_ENV、SQL_HANDLE_DBC、SQL_HANDLE_STMT和SQL_HANDLE_DESC。

➲ **原型**

SQLGetDiagRec：

```
RETCODE SQLGetDiagRec(
                    SQLSMALLINT   HandleType,
                    SQLHANDLE     Handle,
                    SQLSMALLINT   RecNumber,
                    SQLCHAR *     Sqlstate,
                    SQLINTEGER *  NativeErrorPtr,
                    SQLCHAR *     MessageText,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT * TextLengthPtr);
```

SQLGetEnvAttr

SQLGetEnvAttr函数被ODBC API(ODBC 3.0)部分支持，SQLGetEnvAttr函数获取有关环境句柄的属性。

➲ 原型

SQLGetEnvAttr：

```
RETCODE SQLGetEnvAttr(
                    SQLHENV      EnvironmentHandle,
                    SQLINTEGER   Attribute,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```

下表列出了可以使用函数返回的属性列表，以及是否被DBMaster支持。

属性	是否支持
SQL_ATTR_ODBC_VERSION	Y
SQL_ATTR_OUTPUT_NTS	Y
SQL_ATTR_CONNECTION_POOLING	N
SQL_ATTR_CP_MATCH	N

SQLGetStmtAttr

SQLGetStmtAttr函数被ODBC API(ODBC 3.0)部分支持。
SQLGetStmtAttr函数为语句设置属性，它替代了ODBC 2.0中的SQLGetStatementOption函数。

➲ 原型

SQLGetStmtAttr：

```
RETCODE SQLGetStmtAttr(
                    SQLHSTMT     StatementHandle,
                    SQLINTEGER   Attribute,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```

下表列出使用该函数可返回的属性以及DBMaster是否支持。

属性	是否支持?
SQL_ATTR_APPPARAM_DESC	Y
SQL_ATTR_APP_ROW_DESC	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_CONCURRENCY	Y
SQL_ATTR_CURSOR_SCROLLABLE	Y
SQL_ATTR_CURSOR_SENSITIVITY	Y
SQL_ATTR_CURSOR_TYPE	Y
SQL_ATTR_ENABLE_AUTO_IPD	Y
SQL_ATTR_FETCH_BOOKMARK_PTR	Y
SQL_ATTR_IMP_PARAM_DESC	Y
SQL_ATTR_IMP_ROW_DESC	Y
SQL_ATTR_KEYSET_SIZE	Y
SQL_ATTR_MAX_LENGTH	Y
SQL_ATTR_MAX_ROWS	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_NOSCAN	Y
SQL_ATTR_PARAM_BIND_OFFSET_PTR	Y
SQL_ATTR_PARAM_BIND_TYPE	Y
SQL_ATTR_PARAM_OPERATION_PTR	Y
SQL_ATTR_PARAM_STATUS_PTR	Y
SQL_ATTR_PARAMS_PROCESSED_PTR	Y
SQL_ATTR_PARAMSET_SIZE	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_RETRIEVE_DATA	Y
SQL_ATTR_ROW_ARRAY_SIZE	Y
SQL_ATTR_ROW_BIND_OFFSET_PTR	Y
SQL_ATTR_ROW_BIND_TYPE	Y
SQL_ATTR_ROW_NUMBER	N
SQL_ATTR_ROW_OPERATION_PTR	Y
SQL_ATTR_ROW_STATUS_PTR	Y
SQL_ATTR_ROWS_FETCHED_PTR	Y
SQL_ATTR_SIMULATE_CURSOR	Y
SQL_ATTR_USE_BOOKMARK	Y

SQLPrimaryKeys

SQLPrimaryKeys函数被ODBC API(ODBC 3.0)完全支持，SQLPrimaryKeys返回组成主键的字段名称作为结果集。

➲ 原型

SQLPrimaryKeys：

```
RETCODE SQLPrimaryKeys(
                    SQLHSTMT      StatementHandle,
                    SQLCHAR *     CatalogName,
                    SQLSMALLINT   NameLength1,
                    SQLCHAR *     SchemaName,
                    SQLSMALLINT   NameLength2,
                    SQLCHAR *     TableName,
                    SQLSMALLINT   NameLength3);
```

本例使用表CUSTOMER (CUSTID, NAME, ADDRESS, PHONE)，CUSTID是主键。

调用SQLPrimaryKeys函数获取表CUSTOMER的主键信息。

➲ 示例

```
#define TAB LEN 19
#define COL_LEN 19
UCHAR     pkTable[TAB_LEN], fkTable[TAB_LEN];
UCHAR     pkCol[COL_LEN];     /* Primary key column */
SQLHANDLE    hstmt;
SQLINTEGER   pkTableInd, pkColInd;
RETCODE   retcode;
/* Bind the columns that describe the primary and foreign keys */
SQLBindCol(hstmt, 3, SQL C CHAR, pkTable, TAB LEN, &pkTableInd);
SQLBindCol(hstmt, 4, SQL_C_CHAR, pkCol,   COL_LEN, &pkColInd);
/* Get the names of columns in the primary key. */
```

```
retcode = SQLPrimaryKeys(hstmt, NULL, 0, NULL, 0, (SQLCHAR *) "CUSTOMER",
                         SQL NTS);
while (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){
        retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
        printf("Table : %s, Column : %s \n", pkTable, pkCol);
}
/* close the cursor */
SQLCloseCursor(hstmt);
…
```

SQLSetConnectAttr

SQLSetConnectAttr函数被ODBC API(ODBC 3.0)部分支持。SQLSetConnectAttr设置数据库连接的属性，它替代了ODBC 2.0中的SQLSetConnectOption函数。SQLSetConnectAttr的原型为：

```
RETCODE SQLSetConnectAttr(
                    SQLHDBC    ConnectionHandle,
                    SQLINTEGER Attribute,
                    SQLPOINTER ValuePtr,
                    SQLINTEGER StringLength);
```

下表列出了可以使用该函数设置的属性，以及DBMaster是否支持。

下列属性适用：SQL_ATTR_ASYNC_ENABLE只支持SQL_ASYNC_ENABLE_OFF，SQL_ATTR_CONNECTION_TIMEOUT只支持 *0值* (没有超时)，SQL_ATTR_METADATA_ID只支持 *SQL_FALSE*，SQL_ATTR_MAX_ROWS 只支持 *0值* (所有行)。

属性	是否支持?
SQL_ATTR_ACCESS_MODE	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_AUTO_IPD	Y
SQL_ATTR_AUTOCOMMIT	Y
SQL_ATTR_CONNECTION_TIMEOUT	Y

属性	是否支持?
SQL_ATTR_CURRENT_CATALOG	Y
SQL_ATTR_LOGIN_TIMEOUT	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_ODBC_CURSORS	N
SQL_ATTR_PACKET_SIZE	N
SQL_ATTR_QUIET_MODE	N
SQL_ATTR_SYSTEM_DEFAULT	Y
SQL_ATTR_TRACE	N
SQL_ATTR_TRACEFILE	N
SQL_ATTR_TRANSLATE_LIB	N
SQL_ATTR_TRANSLATE_OPTION	N
SQL_ATTR_TXN_ISOLATION	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_MAX_ROWS	Y

SQLSetDescField

SQLSetDescField函数被ODBC API(ODBC 3.0)完全支持，SQLSetDescField函数设置描述符记录的单个域的值。

➲ 原型

SQLSetDescField：

```
RETCODE SQLSetDescField(
                    SQLHDESC    DescriptorHandle,
                    SQLSMALLINT RecNumber,
                    SQLSMALLINT FieldIdentifier,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  BufferLength);
```

下表列出了在DBMaster中使用该函数可设置的描述符域。表中，*G* 代表获取、*S*代表设置、*I* 代表无效。SQL_DESC_ARRAY_SIZE只支持值为1。

域描述符	ARD	APD	IRD	IPD
SQL_DESC_ALLOC_TYPE	G	G	G	G
SQL_DESC_ARRAY_SIZE	G/S	G/S	I	I
SQL_DESC_ARRAY_STATUS_PTR	G/S	G/S	G/S	G/S
SQL_DESC_BIND_OFFSET_PTR	G/S	G/S	I	I
SQL_DESC_BIND_TYPE	G/S	G/S	I	I
SQL_DESC_COUNT	G/S	G/S	G	G/S
SQL_DESC_ROW_PROCESSED_PTR	I	I	G/S	G/S
SQL_DESC_AUTO_UNIQUE_VALUE	I	I	G	I
SQL_DESC_BASE_COLUMN_NAME	I	I	G	I
SQL_DESC_BASE_TABLE_NAME	I	I	G	I
SQL_DESC_CASE_SENSITIVE	I	I	G	I
SQL_DESC_CATALOG_NAME	I	I	G	I
SQL_DESC_CONCISE_TYPE	G/S	G/S	G	G/S
SQL_DESC_DATA_PTR	G/S	G/S	I	I
SQL_DESC_DATETIME_INTERVAL_ CODE	G/S	G/S	G	G/S
SQL_DESC_DATETIME_INTERVAL_ PRECISION	G/S	G/S	G	G/S
SQL_DESC_DISPLAY_SIZE	I	I	G	I
SQL_DESC_FIXED_PREC_SCALE	I	I	G	G
SQL_DESC_INDICATOR_PTR	G/S	G/S	I	I
SQL_DESC_LABLE	I	I	G	I
SQL_DESC_LENGTH	G/S	G/S	G	G/S
SQL_DESC_LITERAL_PREFIX	I	I	G	I
SQL_DESC_LITERAL_SU FFIX	I	I	G	I
SQL_DESC_LOCAL_TYPE_NAME	I	I	G	G
SQL_DESC_NAME	I	I	G	G
SQL_DESC_NULLABLE	I	I	G	G
SQL_DESC_NUM_PREC_RADIX	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH_PTR	G/S	G/S	I	I
SQL_DESC_PARAMETER_TYPE	I	I	I	G/S
SQL_DESC_PRECISION	G/S	G/S	G	G/S
SQL_DESC_SCALE	G/S	G/S	G	G/S

域描述符	ARD	APD	IRD	IPD
SQL_DESC_SCHEMA_NAME	I	I	G	I
SQL_DESC_SEARCHABLE	I	I	G	I
SQL_DESC_TABLE_NAME	I	I	G	I
SQL_DESC_TYPE	G/S	G/S	G	G/S
SQL_DESC_TYPE_NAME	I	I	G	G
SQL_DESC_UNNAMED	I	I	G	I
SQL_DESC_UNSIGNED	I	I	G	G
SQL_DESC_UPDATABLE	I	I	G	I

下面代码段显示了函数SQLSetDescField中常用的选项。为简化程序，表EMPLOYEE中只有两个字段(NAME, SALARY)。

➲ 示例

```
SQLHANDLE   hstmt, ard;
SQLSMALLINT status[2];
UCHAR    name[2][18];
FLOAT       salary[2];
RETCODE  retcode;

/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL ATTR APP ROW DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, (SQLCHAR *) "SELECT * FROM EMPLOYEE", SQL_NTS);

/* set the row size, status and the binding type */
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_SIZE, (void *)2, 0);
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_STATUS_PTR, status, 0);
SQLSetDescField(ard, 0, SQL DESC BIND TYPE, SQL BIND BY COLUMN, 0);
/* set necessary fields to fetch the record */
SQLSetDescField(ard, 1, SQL_DESC_CONCISE_TYPE, (void *)SQL_C_CHAR, 0);
SQLSetDescField(ard, 1, SQL_DESC_OCTET_LENGTH, (void *)19, 0);
```

```
SQLSetDescField(ard, 1, SQL_DESC_DATA_PTR, name, 19);
SQLSetDescField(ard, 2, SQL DESC CONCISE TYPE, (void *)SQL C FLOAT, 0);
SQLSetDescField(ard, 2, SQL_DESC_DATA_PTR, salary, 0);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL SUCCESS) {
        printf("Employee Name : %s, Salary : %f \n", name[0], salary[0]);
       retcode = SQLFetch(hstmt);
}
…
```

SQLSetDescRec

SQLSetDescRec函数被ODBC API(ODBC 3.0)完全支持。
SQLSetDescRec函数用来设置多个描述符域的值，这些值可能影响字段或参数的数据类型和绑定缓存。

➲ **原型**

SQLSetDescRec：

```
RETCODE SQLSetDescField(
                    SQLHDESC     DescriptorHandle,
                    SQLSMALLINT  RecNumber,
                    SQLSMALLINT  Type,
                    SQLSMALLINT  SubType,
                    SQLINTEGER   Length,
                    SQLSMALLINT  Precision,
                    SQLSMALLINT  Scale,
                    SQLPOINTER   DataPtr,
                    SQLINTEGER * StringLengthPtr,
                    SQLINTEGER * IndicatorPtr);
```

下面的代码段显示了如何使用函数SQLSetDescRec来绑定字段。为简化程序，表EMPLOYEE中只有两个字段(NAME, SALARY)。

➲ 示例

```
SQLHANDLE   hstmt, ard;
SQLSMALLINT status[2];
UCHAR   name[2][19];
SQLINTEGER nameInd[2];
FLOAT      salary[2];
RETCODE retcode;
int        i;
SQLINTEGER sLen;
/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL ATTR APP ROW DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, (SQLCHAR *) "SELECT * FROM EMPLOYEE", SQL_NTS);
/* set the row size, status and the binding type */
SQLSetDescField(ard, 0, SQL DESC ARRAY SIZE, (void *)2, 0);
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_STATUS_PTR, status, 0);
SQLSetDescField(ard, 0, SQL_DESC_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
/* set necessary fields to fetch the record */
SQLSetDescRec(ard, 1, SQL C CHAR, 0, 19, 0, 0, name, &sLen, nameInd);
SQLSetDescRec(ard, 2, SQL_C_FLOAT, 0, 0, 0,0, salary, 0, NULL);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL SUCCESS) {
     printf("Employee Name : %s, Salary : %f \n", name[0], salary[0]);
     retcode = SQLFetch(hstmt);
}
…
```

SQLSetEnvAttr

SQLSetEnvAttr函数被ODBC API(ODBC 3.0)部分支持。SQLSetEnvAttr函数为环境句柄设置属性。

➲ 原型

SQLSetEnvAttr：

```
RETCODE SQLSetEnvAttr(
                    SQLHENV     EnvironmentHandle,
                    SQLINTEGER  Attribute,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  StringLength);
```

下表列出了使用该函数设置的属性，以及DBMaster是否支持。SQL_ATTR_OUTPUT_NTS只支持SQL_TRUE（常以NULL结束）。

属性	是否支持?
SQL_ATTR_ODBC_VERSION	Y
SQL_ATTR_OUTPUT_NTS	Y
SQL_ATTR_CONNECTION_POOLING	N
SQL_ATTR_CP_MATCH	N

SQLSetStmtAttr

SQLSetStmtAttr函数被ODBC API(ODBC 3.0)部分支持。SQLSetStmtAttr函数为语句设置属性，它取代了ODBC 2.0中的SQLSetStatementOption函数。

➲ 原型

SQLSetStmtAttr：

```
RETCODE SQLSetStmtAttr(
                    SQLHSTMT    StatementHandle,
                    SQLINTEGER  Attribute,
                    SQLPOINTER  ValuePtr,
```

```
SQLINTEGER  StringLength);
```

下表列出了可以使用该函数设置的属性以及DBMaster是否支持。
SQL_ATTR_ASYNC_ENABLE和
SQL_ATTR_PARAMS_PROCESSED_PTR只支持
SQL_ASYNC_ENABLE_OFF；SQL_ATTR_CONCURRENCY只支持
SQL_CONCUR_READ_ONLY和SQL_CONCUR_LOCK；
SQL_ATTR_CURSOR_SENSITIVITY只支持SQL_UNSPECIFIED；
SQL_ATTR_KEYSET_SIZE和SQL_ATTR_MAX_LENGTH只支持0值；
SQL_ATTR_METADATA_ID只支持 SQL_FALSE；
SQL_ATTR_NOSCAN只支持SQL_NOSCAN_ON；
SQL_ATTR_PARAMSET_SIZE只支持1值。

SQL_ATTR_QUERY_TIMEOUT只支持0值；

SQL_ATTR_RETRIEVE_DATA只支持SQL_RD_ON；
SQL_ATTR_SIMULATE_CURSOR只支持SQL_SC_UNIQUE。

属性	是否支持?
SQL_ATTR_APPPARAM_DESC	Y
SQL_ATTR_APP_ROW_DESC	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_CONCURRENCY	Y
SQL_ATTR_CURSOR_SCROLLABLE	Y
SQL_ATTR_CURSOR_SENSITIVITY	Y
SQL_ATTR_CURSOR_TYPE	Y
SQL_ATTR_ENABLE_AUTO_IPD	Y
SQL_ATTR_FETCH_BOOKMARK_PTR	Y
SQL_ATTR_IMP_PARAM_DESC	N
SQL_ATTR_IMP_ROW_DESC	N
SQL_ATTR_KEYSET_SIZE	Y
SQL_ATTR_MAX_LENGTH	Y
SQL_ATTR_MAX_ROWS	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_NOSCAN	Y
SQL_ATTR_PARAM_BIND_OFFSET_PTR	Y

属性	是否支持?
SQL_ATTR_PARAM_BIND_TYPE	Y
SQL_ATTR_PARAM_OPERATION_PTR	Y
SQL_ATTR_PARAM_STATUS_PTR	Y
SQL_ATTR_PARAMS_PROCESSED_PTR	Y
SQL_ATTR_PARAMSET_SIZE	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_RETRIEVE_DATA	Y
SQL_ATTR_ROW_ARRAY_SIZE	Y
SQL_ATTR_ROW_BIND_OFFSET_PTR	Y
SQL_ATTR_ROW_BIND_TYPE	Y
SQL_ATTR_ROW_NUMBER	N
SQL_ATTR_ROW_OPERATION_PTR	Y
SQL_ATTR_ROW_STATUS_PTR	Y
SQL_ATTR_ROWS_FETCHED_PTR	Y
SQL_ATTR_SIMULATE_CURSOR	Y
SQL_ATTR_USE_BOOKMARK	Y

8.4 ODBC支持64位

ODBC函数

与微软数据访问组件MDAC 2.7 SDK一起装载的开放式数据库连接头和库与之前的ODBC版本相比，做了些改动，现在版本可以允许程序员在新的64位平台上进行开发。

只要确保您的代码是使用下列ODBC定义类型，就可以在基于WIN64或WIN32 macros 64位和32位平台上编译代码。

以下几点请特别注意：

虽然指针大小由4字节变为8字节，而整型和长整型仍保持为4字节。INT64和UINT64类型定义为8字节整型。ODBC中为编译64位已转换为SQLLEN和SQLULEN。因此，之前和参数SQLINTEGER和SQLUINTEGER一起定义的ODBC函数也被相应的更改，情况列举如下。

ODBC中的一些函数参数被声明为指针类型。在32位ODBC中，指针类型通常被用为传递整型数据，也根据调用关系指向缓存。当然，这是可能的，因为指针和整型具有相同的长度。在64位Windows平台上不存在此情况。

一些描述符字段可以通过各种各样的SQLSet…和SQLGet…函数进行设置和返回已经改变为能够适应64位值，其它还仍然为32位值。在调用这些方法时请小心使用合适的缓存大小进行设置和返回那些域。发生改变的一些特殊描述符字段列在本文档的最后部分进行描述。

函数声明变化

下列函数签名已经更改以适应64位编程，满足使用新类型需要。黑体字部分是改变的特殊参数。

- SQLBindCol (SQLHSTMT StatementHandle, SQLUSMALLINT ColumnNumber, SQLSMALLINT TargetType, SQLPOINTER TargetValue, **SQLLEN BufferLength**, **SQLLEN * StrLen_or_Ind**);

- SQLBindParam (SQLHSTMT StatementHandle, SQLUSMALLINT ParameterNumber, SQLSMALLINT ValueType, SQLSMALLINT ParameterType, **SQLULEN LengthPrecision**, SQLSMALLINT ParameterScale, SQLPOINTER ParameterValue, **SQLLEN *StrLen_or_Ind**);
- SQLBindParameter (SQLHSTMT hstmt, SQLUSMALLINT ipar, SQLSMALLINT fParamType, SQLSMALLINT fCType, SQLSMALLINT fSqlType, **SQLULEN cbColDef**, SQLSMALLINT ibScale, SQLPOINTER rgbValue, **SQLLEN cbValueMax**, **SQLLEN *pcbValue**);
- SQLColAttribute (SQLHSTMT StatementHandle, SQLUSMALLINT ColumnNumber, SQLUSMALLINT FieldIdentifier, SQLPOINTER CharacterAttribute, SQLSMALLINT BufferLength, SQLSMALLINT * StringLength, **SQLLEN*** NumericAttribute)
- SQLColAttributes (SQLHSTMT hstmt, SQLUSMALLINT icol, SQLUSMALLINT fDescType, SQLPOINTER rgbDesc, SQLSMALLINT cbDescMax, SQLSMALLINT *pcbDesc, **SQLLEN * pfDesc**);
- SQLDescribeCol (SQLHSTMT StatementHandle, SQLUSMALLINT ColumnNumber, SQLCHAR *ColumnName, SQLSMALLINT BufferLength, SQLSMALLINT *NameLength, SQLSMALLINT *DataType, **SQLULEN *ColumnSize**, SQLSMALLINT *DecimalDigits, SQLSMALLINT *Nullable);
- SQLDescribeParam (SQLHSTMT hstmt, SQLUSMALLINT ipar, SQLSMALLINT *pfSqlType, **SQLULEN *pcbParamDef**, SQLSMALLINT *pibScale, SQLSMALLINT *pfNullable);
- SQLExtendedFetch(SQLHSTMT hstmt, SQLUSMALLINT fFetchType, **SQLLEN irow**, **SQLULEN * pcrow**, SQLUSMALLINT * rgfRowStatus)
- SQLFetchScroll (SQLHSTMT StatementHandle, SQLSMALLINT FetchOrientation, **SQLLEN FetchOffset**);

- SQLGetData (SQLHSTMT StatementHandle, SQLUSMALLINT ColumnNumber, SQLSMALLINT TargetType, SQLPOINTER TargetValue, **SQLLEN BufferLength**, **SQLLEN *StrLen_or_Ind**);
- SQLGetDescRec (SQLHDESC DescriptorHandle, SQLSMALLINT RecNumber, SQLCHAR *Name, SQLSMALLINT BufferLength, SQLSMALLINT *StringLength, SQLSMALLINT *Type, SQLSMALLINT *SubType, **SQLLEN *Length**, SQLSMALLINT *Precision, SQLSMALLINT *Scale, SQLSMALLINT *Nullable);
- SQLParamOptions(SQLHSTMT hstmt, **SQLULEN crow**, **SQLULEN * pirow**)
- SQLPutData (SQLHSTMT StatementHandle, SQLPOINTER Data, **SQLLEN StrLen_or_Ind**);
- SQLRowCount (SQLHSTMT StatementHandle, **SQLLEN* RowCount**);
- SQLSetConnectOption(SQLHDBC ConnectHandle, SQLUSMALLINT Option, **SQLULEN Value**);
- SQLSetPos (SQLHSTMT hstmt, **SQLSETPOSIROW irow**, SQLUSMALLINT fOption, SQLUSMALLINT fLock);
- SQLSetDescRec (SQLHDESC DescriptorHandle, SQLSMALLINT RecNumber, SQLSMALLINT Type, SQLSMALLINT SubType, **SQLLEN Length**, SQLSMALLINT Precision, SQLSMALLINT Scale, SQLPOINTER Data, **SQLLEN *StringLength**, **SQLLEN *Indicator**);
- SQLSetParam (SQLHSTMT StatementHandle, SQLUSMALLINT ParameterNumber, SQLSMALLINT ValueType, SQLSMALLINT ParameterType, **SQLULEN LengthPrecision**, SQLSMALLINT ParameterScale, SQLPOINTER ParameterValue, **SQLLEN *StrLen_or_Ind**);

- SQLSetScrollOptions (SQLHSTMT hstmt, SQLUSMALLINT fConcurrency, **SQLLEN crowKeyset**, SQLUSMALLINT crowRowset);
- SQLSetStmtOption (SQLHSTMT StatementHandle, SQLUSMALLINT Option, **SQLULEN Value**);

通过指针调用ODBC API返回的值

下列ODBC函数调用中，指针所指向的数据是由驱动返回的，返回数据的内容由其它输入参数决定。一些情况下，这些方法可能返回64位(8-byte)整型值，而不是传统的32位(4-byte)整型值。这些情况如下：

SQLColAttribute
当*FieldIdentifier*参数的值为如下一种时，**NumericAttribute*返回64位值：

SQL_DESC_DISPLAY_SIZE
SQL_DESC_LENGTH
SQL_DESC_OCTET_LENGTH
SQL_DESC_COUNT

SQLColAttributes
当*fDescType*参数的值为如下一种时，**pfDesc*返回64位值：

SQL_COLUMN_DISPLAY_SIZE
SQL_COLUMN_LENGTH
SQL_COLUMN_COUNT

SQLGetConnectAttr
当*Attribute*参数的值为如下一种时，*Value*返回64位值：

SQL_ATTR_QUIET_MODE

SQLGetConnectOption
当*Attribute*参数的值为如下一种时，*Value*返回64位值：

SQL_ATTR_QUIET_MODE

SQLGetDescField
当*FieldIdentifier*参数的值为如下一种时，**ValuePtr*返回64位值：

SQL_DESC_ARRAY_SIZE

SQLGetDiagField
当*DiagIdentifier*参数的值为如下一种时，**DiagInfoPtr*返回64位值：

SQL_DIAG_CURSOR_ROW_COUNT
SQL_DIAG_ROW_COUNT
SQL_DIAG_ROW_NUMBER

SQLGetInfo
当*InfoType*参数的值为如下一种时，**InfoValuePtr*返回64位值：

SQL_DRIVER_HENV
SQL_DRIVER_HDBC
SQL_DRIVER_HLIB
当*InfoType*参数的值为如下一种时，**InfoValuePtr* 的输入和输出都是64位值：

SQL_DRIVER_HSTMT
SQL_DRIVER_HDESC

SQLGetStmtAttr
当*Attribute*参数的值为如下一种时，**ValuePtr*返回64位值：

SQL_ATTR_APP_PARAM_DESC
SQL_ATTR_APP_ROW_DESC
SQL_ATTR_IMP_PARAM_DESC

SQL_ATTR_IMP_ROW_DESC
SQL_ATTR_MAX_LENGTH
SQL_ATTR_MAX_ROWS
SQL_ATTR_PARAM_BIND_OFFSET_PTR
SQL_ATTR_ROW_ARRAY_SIZE
SQL_ATTR_ROW_BIND_OFFSET_PTR
SQL_ATTR_ROW_NUMBER
SQL_ATTR_ROWS_FETCHED_PTR
SQL_ATTR_KEYSET_SIZE

SQLGetStmtOption
当*Option*参数的值为如下一种时，**Value*返回64位值：

SQL_MAX_LENGTH
SQL_MAX_ROWS
SQL_ROWSET_SIZE
SQL_KEYSET_SIZE

SQLSetConnectAttr
当*Attribute*参数的值为如下一种时，*Value*返回64位值：

SQL_ATTR_QUIET_MODE

SQLSetConnectOption
当*Attribute*参数的值为如下一种时，*Value*返回64位值：

SQL_ATTR_QUIET_MODE

SQLSetDescField
当*FieldIdentifier*参数的值为如下一种时，**ValuePtr*返回64位值：

SQL_DESC_ARRAY_SIZE

SQLSetStmtAttr
当*Attribute*参数的值为如下一种时，**ValuePtr*返回64位值：

SQL_ATTR_APP_PARAM_DESC
SQL_ATTR_APP_ROW_DESC
SQL_ATTR_IMP_PARAM_DESC
SQL_ATTR_IMP_ROW_DESC
SQL_ATTR_MAX_LENGTH
SQL_ATTR_MAX_ROWS
SQL_ATTR_PARAM_BIND_OFFSET_PTR
SQL_ATTR_ROW_ARRAY_SIZE
SQL_ATTR_ROW_BIND_OFFSET_PTR
SQL_ATTR_ROW_NUMBER
SQL_ATTR_ROWS_FETCHED_PTR
SQL_ATTR_KEYSET_SIZE

SQLSetConnectAttr
当*Option*参数的值为如下一种时，**Value*返回64位值：

SQL_MAX_LENGTH
SQL_MAX_ROWS
SQL_ROWSET_SIZE
SQL_KEYSET_SIZE

不支持的SQL类型

下列4个SQL类型仍然只支持32位。在MDAC2.7中这些类型不能为任何参数使用，如果在64位平台上使用那些类型会导致编译失败。

```
#ifdef WIN32

typedef SQLULEN SQLROWCOUNT;

typedef SQLULEN SQLROWSETSIZE;

typedef SQLULEN SQLTRANSID;

typedef SQLLEN SQLROWOFFSET;

#endif
```

SQLSETPOSIROW的定义已经改变，可为32位和64位编译：

```
#ifdef _WIN64
typedef UINT64 SQLSETPOSIROW;
#else
#define SQLSETPOSIROW SQLUSMALLINT
#endif
```

SQLLEN和SQLULEN的定义改变，可进行64位编译：

```
#ifdef _WIN64
typedef INT64 SQLLEN;
typedef UINT64 SQLULEN;
#else
#define SQLLEN SQLINTEGER
#define SQLULEN SQLUINTEGER
#endif
```

尽管SQL_C_BOOKMARK在ODBC3.0中取消了，为在2.0客户端中进行64位编译，该值更改为：

```
#ifdef _WIN64
#define SQL_C_BOOKMARK SQL_C_UBIGINT
#else
#define SQL_C_BOOKMARK SQL_C_ULONG
#endif
```

BOOKMARK类型在新头中的定义完全不同：

```
typedef SQLULEN        BOOKMARK;
```

9 Unicode支持

DBMaster目前支持内部Unicode数据，所以用户可以在数据库中存储和执行多语言数据，多语言数据必须作为Unicode数据被传递。

提供数据类型NCHAR、NVARCHAR和NCLOB来支持Unicode数据的存储。此外，目前也支持宽字节的Unicode函数。

9.1 Unicode编码接口

从ODBC函数中输入的数据可以是UTF-16LE或UTF-8编码。用户可以使用一个连接选项来定义输入的Unicode编码规则，默认的编码是UTF-16LE。当用户设置此选项为UTF-8时，DBMaster会假设所有的输入字符串是UTF-8编码的，并且这些Unicode字符串也以UTF-8编码输出。Windows应用程序以及如Visual Basic的开发工具使用UTF-16LE编码；如果用户的程序使用的是UTF-8(通常为Unix)，则在您的程序中通过调用ODBC设置连接选项函数来设置连接选项。

DBMaster提供两种Unicode编码来输入和输出字符串数据，它们分别是UTF-8和UTF-16。用户可以调用选项SQL_CLI_UCODE_TYPE的SQLSetConnectAttr，它的值为SQL_CLI_UTYPE_UTF16和SQL_CLI_UTYPE_UTF8，用来设置输入/输出Unicode字符串编码类型。

SQL_CLI_UCODE_TYPE的默认值为SQL_CLI_UTYPE_UTF16 (UTF-16LE)。

Unicode 函数

以下为DBMaster所支持的ODBCUnicode函数：

SQLColAttributeW

SQLColAttributesW

SQLConnectW

SQLDescribeColW

SQLErrorW

SQLExecDirectW

SQLGetConnectAttrW

SQLGetCursorNameW

SQLGetDescFieldW

SQLGetDescRecW

SQLGetDiagFieldW

SQLGetDiagRecW

SQLPrepareW

SQLSetConnectAttrW

SQLSetCursorNameW

SQLSetDescNameW

SQLSetStmtAttrW

SQLGetStmtAttrW

SQLColumnsW

SQLGetConnectOptionW

SQLGetInfoW

SQLGetTypeInfoW

SQLSetConnectOptionW

SQLSpecialColumnsW

SQLStatisticsW

SQLTablesW

SQLDriverConnectW

SQLForeignKeysW

SQLPrimaryKeysW

SQLProcedureColumnsW

SQLProceduresW

相关UnicodeODBC类型：

DBMaster支持以下数据类型：C类型中SQL_C_WCHAR和SQL类型中的SQL_WCHAR、SQL_WVARCHAR、SQL_WLONGVARCHAR 。这些数据类型为相关ODBC函数所提供，例如SQLBindCol、SQLBindParameter、SQLGetData、SQLPutData等。

SQL_C_WCHAR的输入/输出参数字段长度以字节为单位，SQL_C_WCHAR的精确值以字符为单位。

A 函数序列差异

以下附录显示了DBMaster ODBC API与Microsoft ODBC 3.0 API的函数序列差异。

A.1 SQLRowCount

SQLRowCount能够在状态S1、S2和S3中被成功调用。DBMaster不会返回错误，而ODBC 3.0会返回错误代码S1010。

A.2 SQLGetCursorName

如果您在使用SQLSetCursorName设置指针名之前调用SQLGetCursorName函数，DBMaster将不会返回错误S1015，因为DBMaster在分配一个声明前会自动生成一个指针名。

B 函数特性差异

下列附录展示了DBMaster ODBC API与Microsoft ODBC 3.0 API的函数特性差异。这些差异包括DBMaster连接和声明时有用的扩展选项，它们与标准的函数相比略有不同。

B.1 SQLPutData

ODBC 3.0允许SQLPutData函数向字段传递部分CHAR、BINARY、LONG VARCHAR以及LONG VARBINARY类型的数据。DBMaster可限制用于SQLPutData函数的数据类型为LONG VARCHAR与LONG VARBINARY。

B.2 SQLColumns

DBMaster不允许您使用SQLColumns函数来获取临时表的信息。

B.3 SQLTables

DBMaster不允许您使用SQL SQLTables函数来获取临时表的信息。

B.4 SQLDriverConnect

在QLDriverConnect函数中，SQL_DRIVER_PROMPT和SQL_DRIVER_COMPLETE_REQUIRED的提示标记与SQL_DRIVER_COMPLETE提示标记的行为相同。提示行为是由SQLDriverConnect的**fDriverComplete**参数控制的。

B.5 SQLBindParameter

对于SQLBindParameter函数，当**pcbValue**设置为SQL_LEN_DATA_AT_EXEC（长度）的行为与**pcbValue**设置为SQL_DATA_AT_EXEC的行为相同时，SQL_LEN_DATA_AT_EXEC的长度值将被忽略。

B.6 DELETE/UPDATE位置

一条SQL语句通常引用指针名称来定位DELETE或UPDATE。如果由指针名称指定的指针没有开启，那么DBMaster将会探测并且在执行时间而非准备时间返回错误34000。

B.7 SQLSetConnectOption

SQLSetConnectOption函数中，DBMaster拥有一些扩展的连接选项。当您使用DBMaster数据源时，这些选项允许您设置高级连接选项。

选项	描述	允许的值
SQL_TXN_ ISOLATION	为当前连接设置交易保护级别	SQL_TXN_REPEATABLE_ READ SQL_TXN_READ_ UNCOMMITTED SQL_TXN_SERIALIZEABLE SQL_TXN_FORCE_READ_ UNCOMMITTED
SQL_DB_ MODE	为当前数据库连接设置单用户或多用户版本	SQL_SINGLE SQL_MULTI
SQL_ JOURNAL_ MODE	设置书写或关闭数据库日志	SQL_JOURNAL_ON SQL_JOURNAL_OFF
SQL_LOCK_ TIMEOUT	在返回应用程序之前，为锁设置秒数	秒数
SQL_ BACKUP_ MODE	告知数据库使用哪种方式的备份	SQL_BACKUP_DATA SQL_BACKUP_BLOB SQL_BACKUP_OFF
SQL_DATE_ INPUT_ FORMAT	告知数据库返回哪种数据格式	SQL_DATE_FORMAT_0(mm/dd/yy) SQL_DATE_FORMAT_1(mm-dd/yy) SQL_DATE_FORMAT_2(dd/mon/yy) SQL_DATE_FORMAT_3(dd-mon-yy) SQL_DATE_FORMAT_4(mm/dd/yyyy) SQL_DATE_FORMAT_5(mm-dd-yyyy) SQL_DATE_FORMAT_6(yyyy/mm/dd) SQL_DATE_FORMAT_7(yyyy-mm-dd) SQL_DATE_FORMAT_8(dd/mon/yyyy)

选项	描述	允许的值
		SQL_DATE_FORMAT_9(dd-mon-yyyy) SQL_DATE_FORMAT_10(dd.mm.yyyy) SQL_DATE_FORMAT_11(yy/mm/dd) SQL_DATE_FORMAT_12(yy-mm-dd) SQL_DATE_IN_DEFAULT
SQL_DATE_OUTPUT_FORMAT	告知数据库当返回日期类型的数据时使用哪种格式。	SQL_DATE_FORMAT_0(mm/dd/yy) SQL_DATE_FORMAT_1(mm-dd/yy) SQL_DATE_FORMAT_2(dd/mon/yy) SQL_DATE_FORMAT_3(dd-mon-yy) SQL_DATE_FORMAT_4(mm/dd/yyyy) SQL_DATE_FORMAT_5(mm-dd-yyyy) SQL_DATE_FORMAT_6(yyyy/mm/dd) SQL_DATE_FORMAT_7(yyyy-mm-dd) SQL_DATE_FORMAT_8(dd/mon/yyyy) SQL_DATE_FORMAT_9(dd-mon-yyyy) SQL_DATE_FORMAT_10(dd.mm.yyyy) SQL_DATE_FORMAT_11(yy/mm/dd) SQL_DATE_FORMAT_12(yy-mm-dd) SQL_DATE_OUT_DEFAULT
SQL_TIME_INPUT_FORMAT	告知数据库接受的时间格式。	SQL_DATE_FORMAT_0(hh:mm:ss.fff) SQL_DATE_FORMAT_1(hh:mm:ss) SQL_DATE_FORMAT_2(hh:mm) SQL_DATE_FORMAT_3(hh) SQL_DATE_FORMAT_4(hh:mm:ss.fff tt) SQL_DATE_FORMAT_5(hh:mm:ss tt) SQL_DATE_FORMAT_6(hh:mm tt) SQL_DATE_FORMAT_7(hh tt) SQL_DATE_FORMAT_8(tt hh:mm:ss.fff) SQL_DATE_FORMAT_9(tt hh:mm:ss) SQL_DATE_FORMAT_10(tt hh:mm) SQL_DATE_FORMAT_11(tt hh)

选项	描述	允许的值
		SQL_TIME_IN_DEFAULT
SQL_TIME_OUTPUT_FORMAT	告知数据库返回时间类型的数据格式	SQL_DATE_FORMAT_0(hh:mm:ss.fff) SQL_DATE_FORMAT_1(hh:mm:ss) SQL_DATE_FORMAT_2(hh:mm) SQL_DATE_FORMAT_3(hh) SQL_DATE_FORMAT_4(hh:mm:ss.fff tt) SQL_DATE_FORMAT_5(hh:mm:ss tt) SQL_DATE_FORMAT_6(hh:mm tt) SQL_DATE_FORMAT_7(hh tt) SQL_DATE_FORMAT_8(tt hh:mm:ss.fff) SQL_DATE_FORMAT_9(tt hh:mm:ss) SQL_DATE_FORMAT_10(tt hh:mm) SQL_DATE_FORMAT_11(tt hh) SQL_TIME_OUT_DEFAULT
SQL_SYSINFO_CLEAR	要求数据库清除系统信息	无
SQL_CB_MODE	暗示COMMIT/ROLLBACK操作在连接时如何影响指针	SQL_CB_CLOSE SQL_CB_RESERVER SQL_CB_DELETE

B.8 SQLGetConnectOption

SQLGetConnectOption函数有一些外部连接选项属于DBMaster专有。这些选项允许您在使用DBMater数据源时获取一些有用的高级连接选项信息。

扩展的选项为：

选项	描述
SQL_TXN_ISOLATION	为当前hdbc获取交易保护级别
SQL_DB_MODE	获取当前数据库模式
SQL_JOURNAL_MODE	获取当前数据库日志写入模式
SQL_LOCK_TIMEOUT	在返回至应用程序前获取锁的等待秒数
SQL_BACKUP_MODE	获取当前数据库备份模式
SQL_DATE_INPUT_FORMAT	获取当前数据库日期类型数据输入格式
SQL_DATE_OUTPUT_FORMAT	获取当前数据库日期类型数据输出格式
SQL_TIME_INPUT_FORMAT	获取当前时间类型数据的输入格式
SQL_TIME_OUTPUT_FORMAT	获取当前时间类型数据的输出格式
SQL_CB_MODE	获取在连接时COMMIT/ROLLBACK操作对指针的影响方式
SQL_CONNECT_ID	获取当前连接ID

C ODBC 3.0错误

下列附件展示了微软ODBC 3.0 API的一些错误。

C.1 SQLParamData

在调用SQLParamData之后，如果当前状态是S8，驱动将会返回SQL_NEED_DATA，而不是ODBC 3.0定义的SQL_SUCCESS。

C.2 SQLPrepare

如果当前状态为S2，如果结果集为空，调用SQLPrepare后的状态将会变为S3。

如果当前状态为S3，当没有创建结果集时，调用SQLPrepare之后的状态将会变为S2。

D 数据类型

一个驱动器可以将数据源指定的SQL数据类型映射成为ODBC SQL数据类型和驱动器指定的SQL数据类型。每一个SQL数据类型对应一个ODBC C 数据类型。程序能够在SQLBindCol、SQLGetData或者SQLBindParameter中通过**fCType**参数指定正确的C数据类型。在发送数据至数据源之前，驱动器将转换已指定的C数据类型。从数据源重新获得数据之前，驱动器将其转换为指定的C数据类型。

本章包含下列主题：

- *ODBC* SQL数据类型
- *ODBC C*数据类型
- 默认的*ODBC C*数据类型
- 精度、数值范围、长度以及SQL数据类型的显示大小
- 数据类型的转化

D.1 ODBC SQL数据类型

下表展示了ODBCSQL数据类型（字段**fSqlType**）以及相关的DBMaster SQL数据类型（SQL数据类型字段），数据类型的描述也在下表中列出。

FSql类型	SQL数据类型	描述
SQL_CHAR	CHAR(n)	字符串固定长度n(1 <= n <= 3992)
SQL_VARCHAR	VARCHAR(n)	变量长度字符串最大长度n(1 <= n <= 3992)
SQL_LONGVARCHAR	LONG VARCHAR	变量长度字符数，最大长度2 GB
SQL_DECIMAL	DECIMAL(p,s) DECIMAL(p) DECIMAL	正负号准确，数值精确度为p和刻度s (1<= p <= 17; 0 <= s <= p) (6<= p <= 17; s = 6) (p = 17; s = 6)
SQL_SMALLINT	SMALLINT	正负号准确，数值精确度为5和刻度0
SQL_INTEGER	INTEGER	正负号准确，数值精确度为10和刻度0
SQL_REAL	FLOAT	有正负之分、数值尾数精确到7位(10^{-38} - 10^{38})。DBMaster 和ODBC 2.0在定义这种数值时略有不同。
SQL_FLOAT	DOUBLE	有正负之分、数值尾数精确到15位(10^{-38} to 10^{38})
SQL_DOUBLE	DOUBLE	有正负之分、数值尾数精确到15位(10^{-38} to 10^{38})
SQL_FILE	FILE	DBMaster指定数据类型，将FILE字段的数据作为外部文件存储，p（精度）是数字总数，而s（刻度）是小数点后的数字数。

FSql类型	SQL数据类型	描述
SQL_BINARY	BINARY(n)	二进制数确定的长度n(1 <= n <= 3992)。
SQL_LONGVARBINARY	LONG VARBINARY	二进制数变量长度最大为2 GB
SQL_DATE	DATE	日期数值
SQL_TIME	TIME	时间数值
SQL_TIMESTAMP	TIMESTAMP	包括日期数值和时间数值
SQL_WCHAR	WCHAR(n)	Unicode字符串确定的长度n(1<=*n*<=1996)
SQL_WLONGVARCHAR	WLONGVARCHAR	Unicode变量长度字符数据，最长为2GB（1GB为字符长度）
SQL_WVARCHAR	WVARCHAR(n)	Unicode变量长度字符串最大长度为n(1<=*n*<=1996)

D.2 ODBC C数据类型

ODBC C数据类型是应用程序为存储数据而使用的数据类型。C数据类型的使用在SQLBindCol， SQLGetData 和 SQLBindParameter 函数中通过**fCType**参数进行指定。

有效的年值范围为1到9999，有效的月值范围为1到12。有效的天值范围为1-每月中的天数，有效的时间值范围为0~23，有效的分钟值的范围是0~59，有效地秒数值的范围是0~59，有效的分数值是0~999,999,999。（例如500,000,000代表半秒，1, 000,000是千分之一秒，1,000是百万分之一秒（一微秒），1是一秒的十亿分之一（十亿分之一秒）。）

下表列出了有效的**fCType**值，执行每一个**fCType**值的ODBC C数据类型以及在windows与UNIX环境中相应的C类型。

FCType	ODBC C Typedef	C Type	字节
SQL_C_CHAR	UCHAR FAR *	unsigned char FAR *	4
SQL_C_SSHORT	SWORD	short int	2
SQL_C_SHORT	SWORD	short int	2
SQL_C_USHORT	UWORD	unsigned short int	2
SQL_C_SLONG	SDWORD	long int	4
SQL_C_LONG	SDWORD	long int	4
SQL_C_ULONG	UDWORD	unsigned long int	4
SQL_C_FLOAT	SFLOAT	float	4
SQL_C_DOUBLE	SDOUBLE	double	8
SQL_C_BINARY	UCHAR FAR *	unsigned char FAR *	4
SQL_C_DATE	DATE_STRUCT	struct tagDATE_STRUCT { SWORD year; UWORD month; UWORD day; }	6
SQL_C_TIME	TIME_STRUCT	struct	6

FCType	ODBC C Typedef	C Type	字节

		tagTIME_STRUCT {
		UWORD hour;
		UWORD minutes;
		UWORD second;
		}	

SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	struct tagTIMESTAMP_STRUCT {
		SWORD year;
		UWORD month;
		UWORD day;
		UWORD hour;
		UWORD minutes;
		UWORD second;
		UDWORD fraction;
		}
			16

SQL_C_WCHAR	SQLWCHAR FAR*	unsigned short FAR*	4
SQL_C_BOOKMARK	UDWORD	unsigned long int	4
SQL_C_DEFAULT	参考下一章节*默认的ODBC C数据类型*		

D.3 默认的ODBC C数据类型

如果一个应用程序为SQLBindCol、SQLGetData或者SQLBindParameter中的fCType参数指定为**SQL_C_DEFAULT**，驱动器会认为C 数据类型的输入或输出缓存与SQL数据类型的字段或参数所绑定的缓存是一致的。下例表显示每一个ODBC SQL数据类型默认的C数据类型。定义此数据类型时DBMaster与 ODBC 2.0略有不同。

SQL数据类型	默认的C数据类型
SQL_CHAR	SQL_C_CHAR
SQL_VARCHAR	SQL_C_CHAR
SQL_LONGVARCHAR	SQL_C_CHAR
SQL_DECIMAL	SQL_C_CHAR
SQL_FILE1	SQL_C_CHAR
SQL_SMALLINT	SQL_C_SSHORT
SQL_INTEGER	SQL_C_SLONG
SQL_REAL	SQL_C_FLOAT
SQL_FLOAT	SQL_C_FLOAT
SQL_DOUBLE	SQL_C_DOUBLE
SQL_BINARY	SQL_C_BINARY
SQL_LONGVARBINARY	SQL_C_BINARY
SQL_DATE	SQL_C_DATE
SQL_TIME	SQL_C_TIME
SQL_TIMESTAMP	SQL_C_TIMESTAMP
SQL_WCHAR	SQL_C_WCHAR
SQL_WLONGVARCHAR	SQL_C_WCHAR
SQL_WVARCHAR	SQL_C_WCHAR

D.4 精度、刻度、长度与显示大小

SQLColAttributes、SQLColumns与SQLDescribeCol返回一个表中字段的精度、刻度、长度与显示大小。SQLDescribeParam返回一个SQL语句中参数的精度或刻度。SQLBindParameter为一个SQL语句的参数设置精度或刻度。SQLGetTypeInfo在数据源中返回一个SQL数据类型的最大精度与最小最大刻度。

“—”意味着值无意义或者相应类型没有确定。例如，刻度不适合dates类型，并且**SQL_LONGVARCHAR**的精度仅仅涉及存储的数据字节数而无法被确定。

Date、time以及timestamp的精度设定为与它们最大的显示大小相同，即为date、time和timestamp各自长度最大值格式（dd/mon/yyyy、hh:mm:ss.nnn tt、dd/mon/yyyy hh:mm:ss.nnn tt），time与timestamp的刻度是3，意味着小数部分能够被DBMaster准确接受。

以下列表展示精度、刻度、长度与每一个ODBC SQL类型的显示大小。

FSql类型	SQL数据类型	精确度	刻度	长度	显示大小
SQL_CHAR	CHAR(n)	n	—	n	n
SQL_VARCHAR	VARCHAR(n)	n	—	n	n
SQL_LONGVARCHAR	LONG VARCHAR	—	—	—	—
SQL_DECIMAL	DECIMAL(p,s)	p	S	(p+3)/2	p+2
SQL_FILE	FILE	79	—	79	—
SQL_SMALLINT	SMALLINT	5	0	2	6
SQL_INTEGER	INTEGER	10	0	4	11
SQL_REAL	FLOAT	7	—	4	13
SQL_FLOAT	FLOAT	7	—	4	13
SQL_DOUBLE	DOUBLE	15	—	8	22
SQL_BINARY	BINARY(n)	n	—	n	2n
SQL_LONGVARBINARY	LONG VARBINARY	—	—	—	—
SQL_DATE	DATE	11	—	6	11
SQL_TIME	TIME	15	3	6	15
SQL_TIMESTAMP	TIMESTAMP	27	3	16	27
SQL_WCHAR	WCHAR(n)	n	—	2n	2n
SQL_WLONGVARCHAR	WLONGVARCHAR	n	—	2n	2n
SQL_WVARCHAR	WVARCHAR(n)	—	—	—	—

D.5 Data类型转化

本节将介绍两种数据类型的转化：一个是SQL转化为C数据，另一个是C转化为SQL数据。每一小节会具体说明指定数据类型转化的结果。

SQL转化为C数据类型

在使用SQLFetch找回数据之前，您必须指明找回的数据类型在SQLBindCol的fCType参数中已被转化。最终，驱动会存储SQLBindCol中参数**rgbValue** 指定的数据，SQLGetData也是同样。下列表展示了DBMater提供的从SQL到C的数据转化。

SQL Data Type \ C Data Type	SQL_C_CHAR	SQL_C_SSHORT	SQL_C_SHORT	SQL_C_SLONG	SQL_C_LONG	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_BINARY	SQL_C_DATE	SQL_C_TIME	SQL_C_TIMESTAMP	SQL_C_FILE	SQL_C_WCHAR
SQL_CHAR	●							○					○
SQL_VARCHAR	●							○					○
SQL_LONGVARCHAR	●							○				○	○
SQL_DECIMAL	●	○	○	○	○	○	○						○
SQL_FILE	●											○	○
SQL_SMALLINT	○	○	●	○	○	○	○						○
SQL_INTEGER	○	○	○	○	●	○	○						○
SQL_REAL	○	○	○	○	○	●	○						○
SQL_FLOAT	○	○	○	○	○	●	○						○
SQL_DOUBLE	○	○	○	○	○	○	●						○
SQL_BINARY	○							●					○
SQL_LONGVARBINARY	○							●				○	○
SQL_DATE	○							○	●		○		○
SQL_TIME	○							○		●	○		○
SQL_TIMESTAMP	○							○	○	○	●		○
SQL_WCHAR	○							○					●
SQL_WLONGVARCHAR	○							○				○	●
SQL_WVARCHAR	○							○					●

● - default conversion

○ - supported conversion

下面三页图表展示了DBMaster如何将SQL数据转化为C数据。输出数据与返回的代码与用户设置的date输出格式相同。如果data的输出格式是yyyy-mm-dd，那么用户缓冲的大小最少为11个字节，否则就会返回错误。输出数据和返回的代码与date用户设置的date输出格式相同，如果date输出格式为mm-dd-yy，那么用户缓冲大小最少为9个字节，否则也会返回错误。输出数据和返回代码与用户设置的输出格式相同，如果输

出的时间格式有小数部分，如hh:mm:ss.fff，并且用户缓冲大小不够输出的小数部分时，将会返回一个警告信息。

如果时间输出格式包含一个秒的域：例如hh:mm:ss，但是用户缓冲不足以放置秒的数值时，就会返回错误。假设一个用户指明时间输出格式为hh tt，即使分和秒的数值丢失，也不会返回错误，这是因为用户仅想得到小时的信息。Timestamp格式的规则包括日期格式的规则和时间格式的规则，因此，如果日期输出格式为mm/dd/yy并且时间输出格式为hh:mm:ss，那么缓冲大小最少为18(8 + 1 + 8 + 1)，否则会返回错误信息。

假设字段类型为FILE，使用SQL_C_CHAR来绑定这个字段将会获取用户缓冲区中这个文件的内容，例如abcdefg是文件“homework”的内容，那么abcdefg将会输出到用户缓冲区中。与7的例子相同，但是如果您使用SQL_C_FILE来绑定这个字段，它将会获取文件的内容到用户缓冲区指定的新文件（如student）中。

SQL数据类型	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0	N/A
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0	01004
SQL_CHAR	abcdef	SQL_C_BINARY	6	abcdef	N/A
SQL_CHAR	abcdef	SQL_C_BINARY	5	abcde	01004
SQL_VARCHAR	same as SQL_CHAR				
SQL_LONGVARCHAR	same as SQL_CHAR				
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0	N/A
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0	01004
SQL_DECIM	1234.56	SQL_C_C	4	—	22003

SQL数据类型	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
AL		HAR			
SQL_DECIMAL	1234.56	SQL_C_FLOAT	—	1234.56	N/A
SQL_DECIMAL	1234.56	SQL_C_SHORT	—	1234	01004
SQL_SMALLINT	7890	SQL_C_CHAR	5	7890\0	N/A
SQL_SMALLINT	7890	SQL_C_CHAR	4	—	22003
SQL_SMALLINT	7890	SQL_C_LONG	—	7890	N/A
SQL_INTEGER	65000	SQL_C_LONG	—	65000	N/A
SQL_INTEGER	65000	SQL_C_SSHORT	—	—	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	—	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	—	1.234567	N/A
SQL_DOUBLE	1.2345678	SQL_C_SHORT	—	1	01004
SQL_FLOAT	Same as SQL_DOUBLE				
SQL_REAL	Same as SQL_DOUBLE				
SQL_DATE	1995-11-29	SQL_C_CHAR	11	1995-11-29\0	N/A
SQL_DATE	1995-11-29	SQL_C_CHAR	10	—	22003
SQL_DATE	1995-11-29	SQL_C_CHAR	10	11-29-95\0	N/A
SQL_TIME	22:11:33.	SQL_C_C	19	22:11:33.3	N/A

SQL数据类型	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
	32	HAR		20\0	
SQL_TIME	22:11:33.32	SQL_C_CHAR	12	22:11:33.32\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	11	22:11:33.3\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	10	22:11:33\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	9	22:11:33\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	8	—	22003
SQL_TIME	22:11:33.32	SQL_C_CHAR		10 PM\0	N/A
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	24	1995-11-29 23:54:38.234\0	N/A
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	22	1995-11-29 23:54:38.2\0	01004
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	19	—	22003
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	19	11/29/95 23:54:38\0	N/A
SQL_BINARY	ABCDEF	SQL_C_C	15	656667686	N/A

SQL数据类型	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
	G	HAR		96A6B\0	
SQL_BINARY	ABCDEFG	SQL_C_CHAR	14	65666768696A6\0	01004
SQL_BINARY	ABCDEFG	SQL_C_BINARY	7	ABCDEFG	N/A
SQL_BINARY	ABCDEFG	SQL_C_BINARY	6	ABCDEF	01004
SQL_LONGVARBINARY	same as SQL_BINARY				
SQL_FILE	abcdefg	SQL_C_CHAR	8	abcdefg\0	N/A
SQL_FILE	abcdefg	SQL_C_FILE	8	student\0	N/A

C转化为SQL数据类型

当一个应用程序调用SQLExecute或者SQLExecDirect时，驱动器从应用程序中存储位置返回与SQLBindParameter绑定的参数数据。若需要，在驱动将数据传至数据源之前，它会将SQLBindParameter 中fCType参数指定的数据转化为SQLBindParameter 中fSqlType参数指定的数据类型。对于执行中的数据参数，应用程序通过SQLPutData传递这些参数。下表表明DBMaster支持ODBC C数据类型转化为ODBC SQL数据类型。

C DATA TYPE \ SQL DATA TYPE	SQL_CHAR	SQL_VARCHAR	SQL_LONGVARCHAR	SQL_DECIMAL	SQL_FILE	SQL_SMALLINT	SQL_INTEGER	SQL_REAL	SQL_FLOAT	SQL_DOUBLE	SQL_BINARY	SQL_LONGVARBINARY	SQL_DATE	SQL_TIME	SQL_TIMESTAMP	SQL_WCHAR	SQL_WVARCHAR	SQL_WLONGVARCHAR
SQL_C_CHAR	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_C_SSHORT				○		○	○	○	○	○								
SQL_C_SHORT[1]				○		●	○	○	○	○								
SQL_C_SLONG				○		○	○	○	○	○								
SQL_C_LONG[1]				○		○	●	●	○	○								
SQL_C_FLOAT				○		○	○	○	○	○								
SQL_C_DOUBLE				○		○	○	○	●	●								
SQL_C_BINARY	○	○	○	○		○	○	○	○	○	●	●	○	○	○	○	○	○
SQL_C_DATE	○	○	○										●		○	○	○	○
SQL_C_TIME	○	○	○											●	○	○	○	○
SQL_C_TIMESTAMP	○	○	○										○	○	●	○	○	○
SQL_C_FILE			○									○						
SQL_C_WCHAR	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●

● - default conversion
○ - supported conversion

下面三页图表展示了DBMater如何将C数据转化为SQL数据。这种数据类型的转化与之前表中的数据类型相同。如果服务器端有一个名为'homework'的文件，然后使用SQL_C_CHAR绑定一个参数的话，这样会将指定的FILE字段与文件'homework'链接。QL_C_SSHORT、SQL_C_SHORT、 SQL_C_SLONG、SQL_C_LONG、SQL_C_FLOAT与SQL_C_DOUBLE是数字ODBC C数字类型。SQL_C_DATE不能被转化为SQL_TIME。

当SQL_C_DATE转化为SQL_TIMESTAMP时，timestamp的时间部分设置为零。SQL_C_TIME不能转化为SQL_DATE。Timestamp的日期部分在SQL_TIME转化为SQL_C_TIMESTAMP时将被设置为当前日期。如果时间部分非零，在SQL_C_TIMESTAMP转化为SQL_DATE时将会返回

“数据被删减”的警告。如果小数部分非零，当SQL_C_TIMESTAMP转化为SQL_TIME时会返回“数据被删减”的警告。

由于timestamp在小数点后只有三位数是确定的，当小数点后超过三位数时将会返回警告信息。如果客户端有一个名为'homework'的文件，文件内容为abcdefg，使用SQL_C_FILE来绑定一个参数将会将内容插入到BLOB或者FILE字段中。

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcdef	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6, 2	1234.56	N/A
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5, 1	1234.5	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6, 3	—	N/A
SQL_C_CHAR	1234\0	SQL_INTEGER	—	1234	N/A
SQL_C_CHAR	1234.56\0	SQL_INTEGER	—	1234	01004
SQL_C_CHAR	12345678912\0	SQL_INTEGER	—	—	22003
SQL_C_CHAR	abcdef\0	SQL_INTEGER	—	—	22005
SQL_C_CHAR	abcdef\0	SQL_SMALLINT	—	—	22005
SQL_C_CHAR	1234.56\0	SQL_FLOAT	—	1234.56	N/A
SQL_C_CHAR	1234.5678\0	SQL_FLOAT	—	1234.5678	N/A
SQL_C_CHAR	1.23456e+4\0	SQL_FLOAT	—	1234.56	N/A
SQL_C_CHAR	abcdef\0	SQL_FLOAT	—	—	22005
SQL_C_CHAR		SQL_DOUBLE	same as SQL_FLOAT		

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
SQL_C_CHAR	666768696A6b\0	SQL_BINARY	6	BCDEF	N/A
SQL_C_CHAR	666768696A6\0	SQL_BINARY	6	BCDEF	N/A
SQL_C_CHAR	666768696A6b\0	SQL_BINARY	5	BCDEF	01004
SQL_C_CHAR	HHKKLLMM\0	SQL_BINARY	6	—	22005
SQL_C_CHAR	1995-11-29\0	SQL_DATE	—	1995-11-29	N/A
SQL_C_CHAR	1995-11-29 00:00:00.00\0	SQL_DATE	—	1995-11-29	N/A
SQL_C_CHAR	1995-11-29 23:54:38.23\0	SQL_DATE	—	1995-11-29	01004
SQL_C_CHAR	1995/22/33	SQL_DATE	—	—	22008
SQL_C_CHAR	17:18:19.123	SQL_TIME	—	17:18:19.123	N/A
SQL_C_CHAR	1995-11-29 17:18:19.123	SQL_TIMESTAMP	—	1995-11-29 17:18:19.123	N/A
SQL_C_CHAR	homework\0	SQL_FILE	9	Abcdefg	N/A
SQL_C_SHORT	7890	SQL_SMALLINT	—	7890	N/A
SQL_C_SSHO	7890	SQL_VARCHA	—	—	S1C00

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
RT		R			
SQL_C_SSHORT	7890	SQL_BINARY	—	—	07006
SQL_C_SSHORT	7890	SQL_SMALLINT	—	7890	N/A
SQL_C_SSHORT	7890	SQL_INTEGER	—	7890	N/A
SQL_C_USHORT	7890	SQL_INTEGER	—	7890	N/A
SQL_C_LONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_SLONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_ULONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_FLOAT	1234.00	SQL_INTEGER	—	1234	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	—	1234	01004
SQL_C_FLOAT	1234567.24	SQL_SMALLINT	—	—	22003
SQL_C_FLOAT	1234.56	SQL_DECIMAL	6, 2	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_DECIMAL	5, 1	1234.5	01004
SQL_C_FLOAT	1234.56	SQL_DECIMAL	6, 3	—	22003
SQL_C_DOUBLE	same as SQL_C_FLOAT				
SQL_C_BINA	ABCDEF	SQL_BINARY	6	ABCDEF	N/A

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
RY					
SQL_C_BINARY	ABCDEF	SQL_BINARY	5	ABCDE	01004
SQL_C_DATE	1996,3,8	SQL_DATE	—	1996,3,8	N/A
SQL_C_DATE	1996,3,8	SQL_TIME	—	—	07006
SQL_C_DATE	1996,3,8	SQL_TIMESTAMP	—	1996,3,8,0,0,0,0	N/A
SQL_C_TIME	13,14,15	SQL_DATE	—	—	07006
SQL_C_TIME	13,14,15	SQL_TIME	—	13,14,15	N/A
SQL_C_TIME	13,14,15	SQL_TIMESTAMP	—	1996,3,8,13,14,15	N/A
SQL_C_TIMESTAMP	1996,3,8,13,14,15,160000000	SQL_DATE	—	1996,3,8	01004
SQL_C_TIMESTAMP	1996,3,8,13,14,15,160000000	SQL_TIME	—	13,14,15	01004
SQL_C_TIMESTAMP	1996,3,8,13,14,15,160000000	SQL_TIMESTAMP	—	1996,3,8,13,14,15,160000000	N/A
SQL_C_TIMESTAMP	1996,3,8,13,14,15,167800000	SQL_TIMESTAMP	—	1996,3,8,13,14,15,167000000	01004
SQL_C_FILE	homework\0	SQL_LONGVARCHAR	6	Abcdefg	N/A

E ODBC日志函数

本章列出了用于跟踪记录ODBC函数DBMaster支持的所有配置文件选项。您可以通过微软windows平台的ODBC驱动管理器来跟踪ODBC函数，但是通过驱动管理器自身调用的函数不能被记录。另外，微软的日志函数只能设置为日志开启/关闭来跟踪所有ODBC函数或全部不跟踪，并且只能用于windows平台下。DBMaster支持更有效的日志函数，它不仅可以设置日志开启/关闭，还可以指定用户想要记录的ODBC函数。

您可以在配置文件中设置一个名为“DM_COMMON_OPTION”特殊的段，然后在这个段中设置关键字的不同选项来开启或关闭ODBC日志函数。

注意：如果您决定推迟使用ODBC日志函数，请您一定要记得删除“DM_COMMON_OPTION”段，或者在配置文件中设置LG_Trace = 0以关闭跟踪。否则日志函数将会记录您在数据源的任何动作。

下表展示了配置文件中日志函数的设置选项，以及它们的描述和取值。

关键字选项	描述	取值	默认值
LG_Trace	设置日志函数的跟踪选项	0-关闭ODBC日志。 1-开启ODBC日志。	0
LG_Path	设置日志文件的输出路径。如果路径无效将不会输出日志。	输出日志文件路径。	C:\odbclog.log (for Win32). ./odbclog.log (for Unix).
LG_Time	设置是否记录每一次调用ODBC函数的时间。	0-不记录每一次调用ODBC函数的时间。 1-记录每一次调用ODBC函数的时间。	0
LG_PTFun	设置记录日志的函数列表。这个列表是字符串，它包含零或者一些ODBC函数的名称，由逗号隔开。	函数列表串（例如SQLGetDiagRec，SQLCloseCursor）。	未定义（所有函数都被记入日志）。

关键字选项	描述	取值	默认值
LG_NPFun	设置未记入日志的函数列表。该列表是字符串，它包含零或者一些ODBC函数的名称，由逗号隔开。仅当LG_PTFun在日志文件中没有被定义时，此关键字才有效。	函数列表串（例如SQLGetDiagRec，SQLCloseCursor）。	（空字符串，所有函数都被记入日志）。