ODBC 数据库开发

目录

第 1 章　介绍

在文章的开头做一个习惯性的介绍。

本文从 2002 年 11 月开始写，基本上在 2002 年 12 月时完成，当时本来作为一本书的一个章节，后来由于某些原因没有完成该书。这段时间将本文内容进行了一些整理，放在网上希望能够给大家一些帮助。

本文的内容主要是关于 ODBC 的功能，所有内容都与 ODBC 3.X 版本兼容。

本文简要介绍了 ODBC 的历史和发展，也介绍了 ODBC 的基本的常用功能。大致包括：

- 使用 ODBC 进行数据库连接
- 利用 ODBC 直接执行 SQL 语句
- ODBC 光标类型介绍
- 利用滚动光标或非滚动光标进行结果集查询
- 存储过程的调用与参数绑定
- SQL 语句的准备执行方式
- BLOB 数据字段的查询和修改

本文的数据库利用了 MS SQL Server，ODBC 在使用时是与数据库无关的所以所有例程都可以运行在其他数据库上，例如 Oracle。其实利用 Access 数据库来进行练习也是可以的，但是由于 Access 不能支持存储过程，所以我没有使用 Access 数据库。

由于例程代码没有找到，所以没有就没有办法提供，但是文中的代码都比较详细而且有具体的解释。

书中有很多错误和不足之处希望大家能够容忍和包含，也欢迎来信指出。

闻怡洋 2003 年 07 月 01 日

http://www.vchelp.net

第 2 章　ODBC API 访问数据库

2.1 ODBC 简要介绍

2.1.1 在没有 ODBC 以前

请允许我将那时候成为第二黑暗时代，第一黑暗时代是没有数据库的时代。

ODBC 的出现结束了数据库开发的无标准时代。在没有 ODBC 以前不同的数据库的开发所采用的标准是不统一的。一般来讲不同的数据库厂商都有自己的数据库开发包，这些开发包支持两种模式的数据库开发：预编译的嵌入模式（例如 Oracle 的 ProC，SQL Server 的 ESQL）和 API 调用（例如 Oracle 的 OCI）。

对于一个开发人员来讲使用预编译方式开发是极其痛苦的，我就有过这样的经历，所有的 SQL 语句要写在程序内部，并且遵守一定的规则，然后由数据库厂商的预编译工具处理后形成 C 代码，最后由 C 编译器进行编译。预编译的最大问题就在于无法动态的生成 SQL 语句，我想作为一个程序员是很难接受的。

接下来的是使用 API 进行开发，和预编译相比算是前进了一大步。数据库厂商提供了开发包，你通过各种 API 函数就可以连接数据库，执行查询、修改、删除，操纵光标，执行存储过程等。对于程序员来讲有了更多的自由，而且可以创建自己的开发包。但是这一切的开发只能针对同一种数据库。

Oracle 的 OCI 是一个非常优秀的 C 语言开发包，在 ODBC 中就在很多地方参照了 OCI 的设计。

2.1.2 ODBC 介绍

ODBC(Open Database Connectivity)是由微软公司提出的一个用于访问数据库的统一界面标准，随着客户机/服务器体系结构在各行业领域广泛应用，多种数据库之间的互连访问成为一个突出的问题，而 ODBC 成为目前一个强有力的解决方案。ODBC 之所以能够操作众多的数据库，是由于当前绝大部分数据库全部或部分地遵从关系数据库概念，ODBC 看待这些数据库时正是着眼了这些共同点。虽然支持众多的数据库，但这并不意味 ODBC 会变得复杂，ODBC 是基于结构化查询语言(SQL)，使用 SQL 可大大简化其应用程序设计接口(API)，由于 ODBC 思想上的先进性，而且没有同类标准或产品与之竞争，因而越来越受到众多厂家和用户的青睐。目前，ODBC 已经成为客户机/服务器系统中的一个重要支持技术。

在 1994 年时 ODBC 有了第一个版本，这种名为 Open Data Base Connection（开放式数据库互连）的技术很快通过了标准化并且得到各个数据库厂商的支持。ODBC 在当时解决了两个问题，一个是在 Windows 平台上的数据库开发，另一个是建立一个统一的标准，只要数据厂商提供的开发包支持这个标准，那么开发人员通过 ODBC 开发的程序可以在不同的数据库之间自由转换。这对开发人员来说的确值得庆贺。

ODBC 参照了 X/OpenData Management: SQL Call-Level Interface 和 ISO/ICE1995 Call-Level Interface 标准，在 ODBC 版本 3.X 中已经完全实现了这两个标准的所有要求。所以本书所有内容都基于 ODBC 3.0 以上版本。

最开始时支持 ODBC 的数据库只有 SQL Server，ACCESS，FoxPro，这些都时微软的产品，他们能够支持 ODBC 一点也不奇怪，但是那时候 Windows 的图形界面已经成为了客户端软件最理想的载体，所以各大数据厂商也在不久后发布了针对 ODBC 的驱动程序。

在 Windows 3.X 和 Windows 95 的时候 ODBC 并不作为系统的组成部分出现，使用前必须另行安装。但到了 Windows 98 的时候，当你安装好操作系统后，ODBC 不需要另行安装了，因为它已经成为了操作系统的一部分。这对很多拒绝 ODBC 的人来说又少了一个借口。

作为一个程序员，至少是我，我实在找不出什么理由不为 ODBC 欢呼。此外 ODBC 的结构很简单和清晰，学习和了解 ODBC 的机制和开发方法对学习 ADO 等其他的数据库访问技术会有所帮助。

2.1.3 ODBC 结构

图 2.1 显示了 ODBC 的结构。

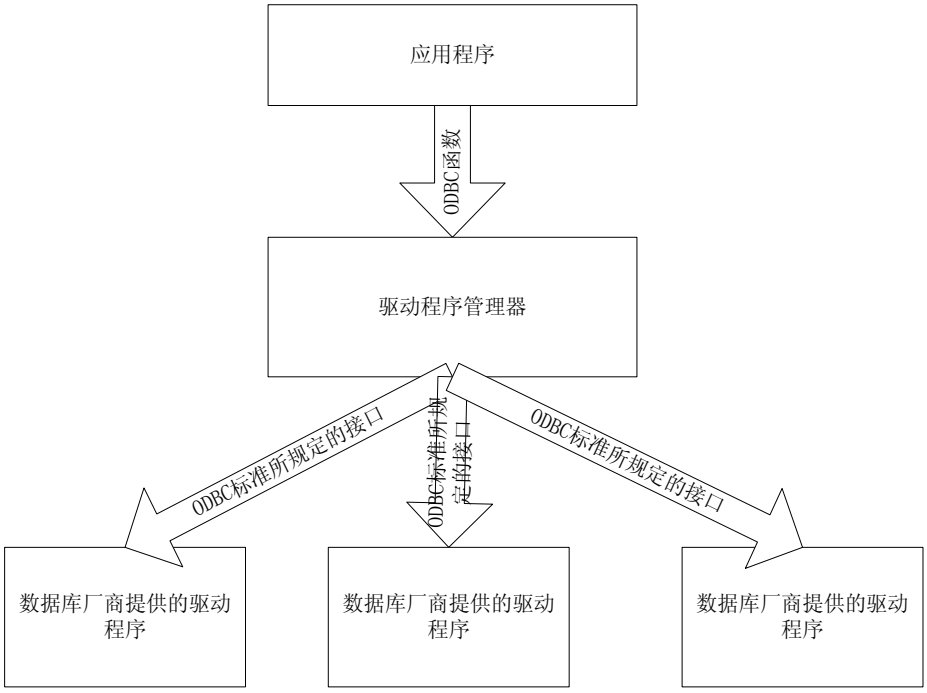


图 2.1

应用程序(Application)

应用程序本身不直接与数据库打交道，主要负责处理并调用 ODBC 函数，发送对数据库的 SQL 请求及取得结果。

驱动程序管理器(Driver Manager)

驱动程序管理器是一个带有输入程序的动态链接库(DLL)，主要目的是加载驱动程序，处理 ODBC 调用的初始化调用，提供 ODBC 调用的参数有效性和序列有效性。

驱动程序(Driver)

驱动程序是一个完成 ODBC 函数调用并与数据库相互影响的 DLL，这些驱动程序可以处理对于特定的数据的数据库访问请求。对于应用驱动程序管理器送来的命令，驱动程序再进行解释形成自己的数据库所能理解的命令。驱动程序将处理所有的数据库访问请求，对于应用程序来讲不需要关注所使用的是本地数据库还上网络数据库。

2.1.4 ODBC 的一致性

ODBC 接口的优势之一为互操作性，程序设计员可以在不指定特定数据源情况下创建 ODBC 应用程序。从应用程序角度方面，为了使每个驱动程序和数据源都支持相同的 ODBC 函数调用和 SQL 语句集，ODBC 接口定义了一致性级别，即 ODBC API 一致性和 ODBC SQL 语法一致性。SQL 一致性规定了对 SQL 语句语法的要求，而 API 一致性规定了驱动程序需要实现的 ODBC 函数。一致性级别通过建立标准功能集来帮助应用程序和驱动程序的开发者，应用程序可以很容易地确定驱动程序是否提供了所需的功能，驱动程序可被开发以支持应用程序选项，而不用考虑每个应用程序的特定请求。

2.2 使用 ODBC 进行数据库开发基本知识介绍

2.2.1 建立 ODBC DSN

DSN（Data Source Name）是用于指定 ODBC 与相关的驱动程序相对应的一个入口，所有 DSN 的信息由系统进行管理，一般来讲当应用程序要使用 ODBC 访问数据库时，就需要指定一个 DSN 以便于连接到一个指定的 ODBC 驱动程序。

在控制面板中打开 ODBC 管理器，回看到如图 2.2 的界面。

图 2.2

DSN 共分为三类：

- 用户 DSN：对当前登录用户可见，只能够用于当前计算机。
- 系统 DSN：对当前系统上所有用户可见，包括 NT 中的服务。
- 文件 DSN：DSN 信息存放在文件中，对能够访问到该文件的用户可见。

一个使用 Access 数据库的 DSN 中的信息如下：

[ODBC]

DRIVER=Driver do Microsoft Access (*.mdb)

UID=admin

DefaultDir=C:\www.vchelp.net\DB

DBQ=C:\www.vchelp.net\DB\chat.mdb

对于文件 DSN 来讲这些信息存放在文件中，对于用户 DSN 和系统 DSN 来讲这些信息存放在注册表内。你可以通过创建文件 DSN 来查看每种 DSN 对应的信息内容。

下面的例子将告诉你如何添加一个 SQL Server 的 DSN。

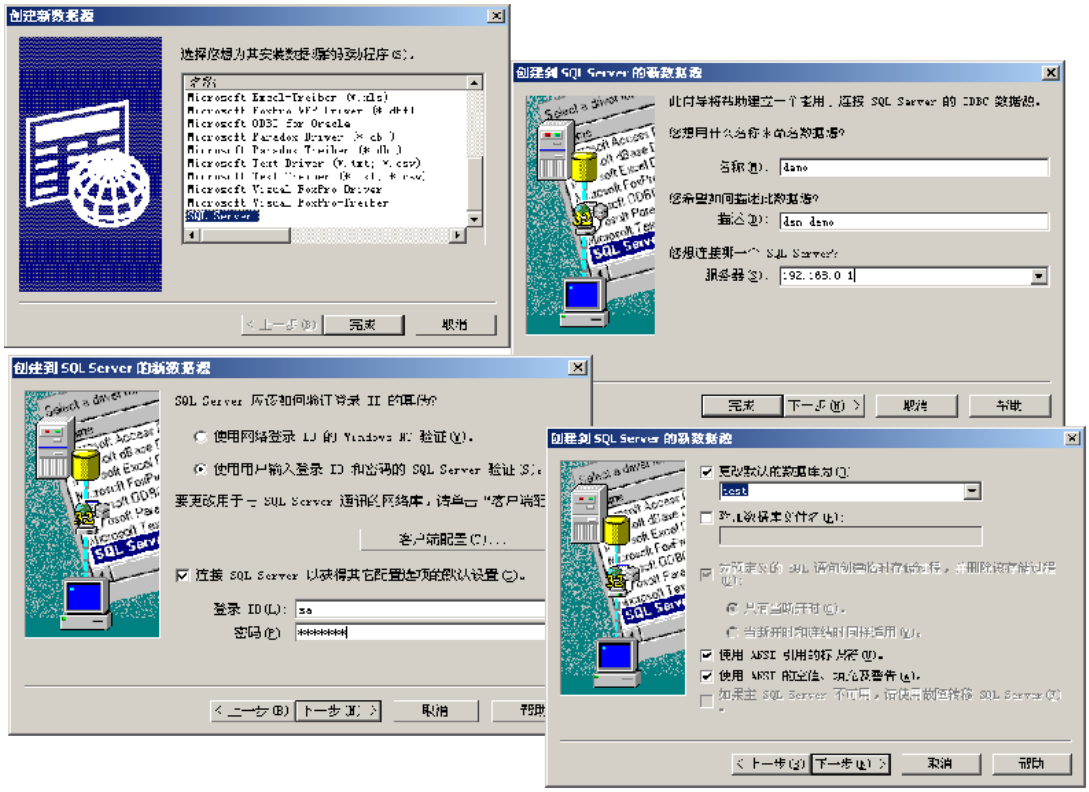


图 2.3

图 2.3 中的四个步骤分别是：

- 选择 SQL Server 作为驱动程序
- 输入 DSN 名称和 SQL Server 服务器地址或别名
- 输入用户和口令进行连接
- 选择默认数据库并完成

2.2.2 使用 ODBC 所需要的文件

你需要下面的文件：

- sql.h：包含有基本的 ODBC API 的定义。
- sqlext.h：包含有扩展的 ODBC 的定义。
- odbc32.lib：库文件。

这些文件在 VC6，VC7 都已经随开发工具提供了，不需要另外安装。

此外所有的 ODBC 函数都以 SQL 开始，例如 SQLExecute，SQLAllocHandle。

2.2.3 SQL 语句执行方式介绍

在 ODBC 中 SQL 语句的执行方式分为两种，直接执行和准备执行。

直接执行是指由程序直接提供 SQL 语句，例如：Select * from test_table 并调用 SQLExecDirect 执行，准备执行是指先提供一个 SQL 语句并调用 SQLPrepare，然后当语句准备好后调用 SQLExecute 执行前面准备好的语句。准备执行多用于数据插入和数据删除，在进行准备时将由 ODBC 驱动程序对语句进行分析，在实际执行时可以避免进行 SQL 语句分析所花费的时间，所以在进行大批量数据操作时速度会比直接执行有明显改善。在后面的章节中我会详细介绍准备执行与行

列绑定与参数替换的用法。

2.2.4 获取 SQL 语句执行的结果

对于 SQL 查询语句，ODBC 会返回一个光标，与光标对应的是一个结果集合（可以理解为一个表格）。开发人员利用光标来浏览所有的结果，你可以利用 ODBC API 函数移动光标，并且获取当前光标指向的行的列字段的数值。此外还可以通过光标来对光标当前所指向的数据进行修改，而修改会直接反映到数据库中。

对于数据更新语句，如插入，删除和修改，在执行后可以得到当前操作所影响的数据的行数。

2.2.5 程序执行的基本流程图

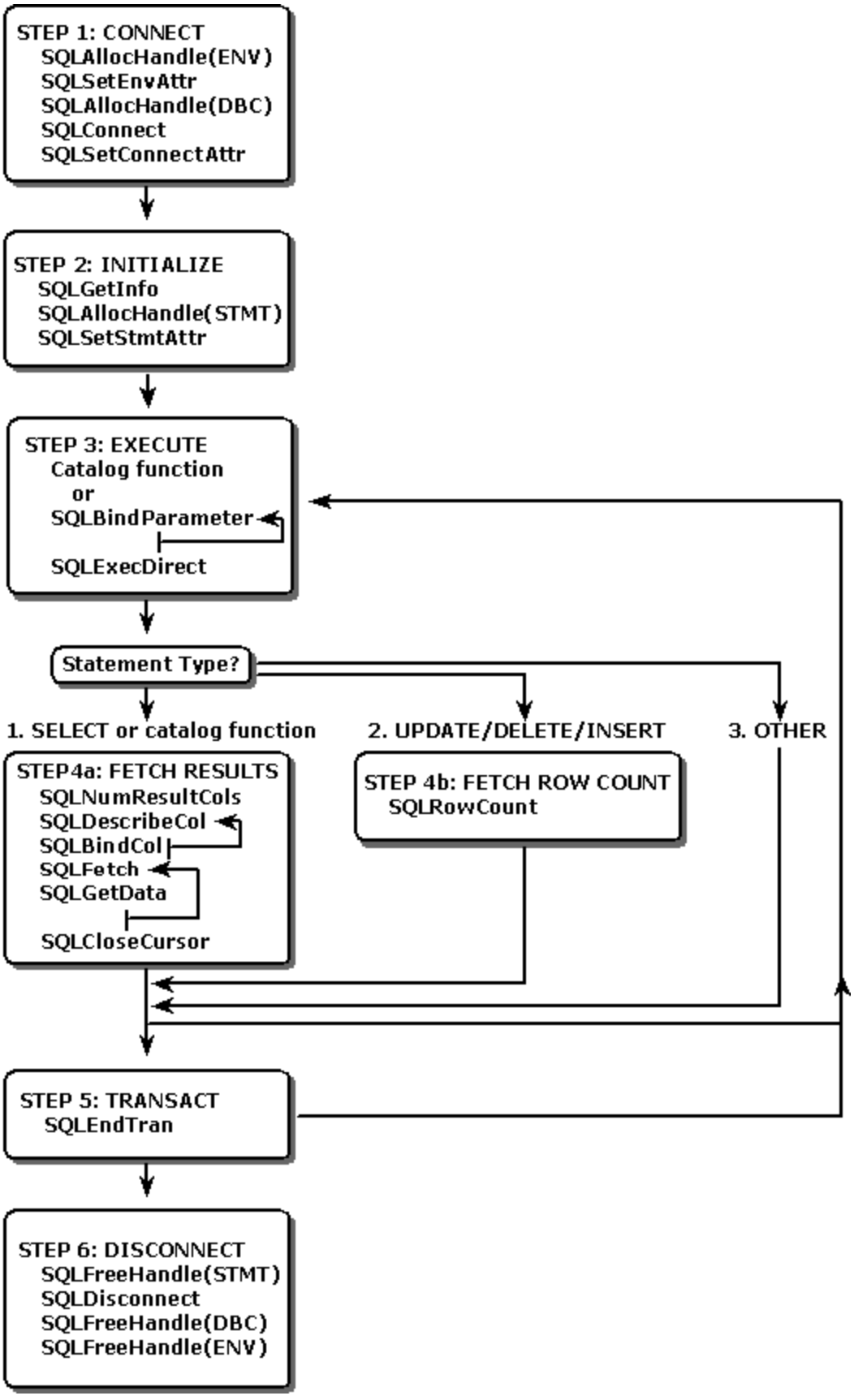


图 2.4

图 2.4 中是一个基本的使用 ODBC API 的一个流程，你现在并不理解上面所有的函数的作用，这没有关系。但希望能够通过这幅图给你一个最初的映象，那就是使用 ODBC API 开发并不复杂。

2.2.6 数据类型定义

在使用 ODBC 开发时一个重要的问题就是数据转换的问题，在 ODBC 中存在下面的几类数据：

- 数据库中 SQL 语言表达数据的类型
- ODBC 中表达数据的类型
- C 语言中表达数据的类型

在程序运行过程中数据需要经历两次转换：C 语言的数据或结构类型与 ODBC 的数据类型的转换，ODBC 与 SQL 间数据类型的转换。所以 ODBC 所定义的数据类型起到了中间桥梁的作用，在 ODBC 的驱动程序调用自己的 DBMS 数据库访问接口时就需要对数据类型进行转换。我们所需要关注的是 C 语言的数据类型和 ODBC 数据类型间的转换关系。

从下图中可以看到 ODBC 中定义的数据类型和 SQL 语言中数据类型的对应关系，所以通过下表我们可以将 ODBC 和 SQL 语言间的数据一一对应，在后面的文字中我们不再区分 ODBC 数据类型和 SQL 语言数据类型。

ODBC 数据类型名称	**SQL 语言数据类型名称**
SQL_CHAR	CHAR(*n*)
SQL_VARCHAR	VARCHAR(*n*)
SQL_LONGVARCHAR	LONG VARCHAR
SQL_WCHAR	WCHAR(*n*)
SQL_WVARCHAR	VARWCHAR(*n*)
SQL_WLONGVARCHAR	LONGWVARCHAR
SQL_DECIMAL	DECIMAL(*p*,*s*)
SQL_NUMERIC	NUMERIC(*p*,*s*)
SQL_SMALLINT	SMALLINT
SQL_INTEGER	INTEGER
SQL_REAL	REAL
SQL_FLOAT	FLOAT(*p*)
SQL_DOUBLE	DOUBLE PRECISION
SQL_BIT	BIT
SQL_TINYINT	TINYINT
SQL_BIGINT	BIGINT
SQL_BINARY	BINARY(*n*)
SQL_VARBINARY	VARBINARY(*n*)
SQL_LONGVARBINARY	LONG VARBINARY
SQL_TYPE_DATE[6]	DATE

SQL_TYPE_TIME[6]	TIME(*p*)
SQL_TYPE_TIMESTAMP[6]	TIMESTAMP(*p*)
SQL_GUID	GUID

图 2.5

使用 C/C++语言开发，那么必定会在与 ODBC 语言间存在数据的转换的问题，因为 ODBC 所存在的一些数据类型在 C 语言中是不存在的。在 ODBC 以宏定义的方式定义了 C 语言和 ODBC 中使用的数据类型：

C 语言数据类型名称	**ODBC 数据类型定义**	**C 语言实际类型**
SQL_C_CHAR	SQLCHAR *	unsigned char *
SQL_C_SSHORT[j]	SQLSMALLINT	short int
SQL_C_USHORT[j]	SQLUSMALLINT	unsigned short int
SQL_C_SLONG[j]	SQLINTEGER	long int
SQL_C_ULONG[j]	SQLUINTEGER	unsigned long int
SQL_C_FLOAT	SQLREAL	float
SQL_C_DOUBLE	SQLDOUBLE, SQLFLOAT	double
SQL_C_BIT	SQLCHAR	unsigned char
SQL_C_STINYINT[j]	SQLSCHAR	signed char
SQL_C_UTINYINT[j]	SQLCHAR	unsigned char
SQL_C_SBIGINT	SQLBIGINT	_int64[h]
SQL_C_UBIGINT	SQLUBIGINT	unsigned _int64[h]
SQL_C_BINARY	SQLCHAR *	unsigned char *
SQL_C_BOOKMARK[i]	BOOKMARK	unsigned long int[d]
SQL_C_VARBOOKMARK	SQLCHAR *	unsigned char *
SQL_C_TYPE_DATE[c]	SQL_DATE_STRUCT	struct tagDATE_STRUCT { SQLSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; } DATE_STRUCT;[a]
SQL_C_TYPE_TIME[c]	SQL_TIME_STRUCT	struct tagTIME_STRUCT { SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; } TIME_STRUCT;[a]
SQL_C_TYPE_TIMESTAMP[c]	SQL_TIMESTAMP_STRUCT	struct tagTIMESTAMP_STRUCT { SQLSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; SQLUSMALLINT hour; SQLUSMALLINT minute; SQLUSMALLINT second; SQLUINTEGER fraction;[b] } TIMESTAMP_STRUCT;[a]
SQL_C_NUMERIC	SQL_NUMERIC_STRUCT	struct tagSQL_NUMERIC_STRUCT { SQLCHAR precision; SQLSCHAR scale; SQLCHAR sign[g]; SQLCHAR val[SQL_MAX_NUMERIC_LEN];[e], [f] } SQL_NUMERIC_STRUCT;
SQL_C_GUID	SQLGUID	struct tagSQLGUID { DWORD Data1; WORD Data2; WORD Data3; BYTE Data4[8]; } SQLGUID;[k]

图 2.6

所以在 ODBC 的开发过程中不要使用 int ， float 之类的 C 语言的实际类型来定义变量而应该使用 ODBC 定义的数据类型来定义变量，如：SQLINTEGER，SQLFLOAT。

2.2.7 ODBC 句柄

ODBC 中的句柄分为三类：环境句柄，数据库连接句柄，SQL 语句句柄。

通过图 2.4 看出，在使用 ODBC 功能时必须先申请环境句柄，然后在环境句柄的基础上创建数据库连接，最后在数据连接的基础上执行 SQL 语句。

2.3 为本章的例程创建 DSN 与数据库表

为了后面的例子能够顺利执行，请创建一个名称为“test”的 DSN，并且使用下面的语句在数据库中创建表和插入基本的数据，这个例子和以后的例子中我们使用 SQL Server 作为数据库，你需要连接到 SQL Server 上执行下面的语句来创建表和插入数据。

```
Create table test_t1(iID int primary key , tmJoin datetime , szName varchar(40) ,fTall float );
Insert into test_t1 values(1, '2002-1-1 15:25' , 'user_1',1.56 );
Insert into test_t1 values(2, '2002-1-2 12:25' , 'user_2',1.53 );
Insert into test_t1 values(3, '2002-1-3 13:25' , 'user_3',1.76 );
```

2.4 ODBC 的基本功能介绍

2.4.1 所需要了解的 ODBC API

2.4.1.1 SQLAllocHandle 创建 ODBC 句柄

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

第一个参数 HandleType 的取值可以为：

- SQL_HANDLE_ENV：申请环境句柄。
- SQL_HANDLE_DBC：申请数据库连接句柄。
- SQL_HANDLE_STMT：申请 SQL 语句句柄，每次执行 SQL 语句都申请语句句柄，并且在执行完成后释放。

第二个参数为输入句柄，第三个参数为输出句柄，也就是是你在第一参数指定的需要申请的句柄。

根据 1.2.7 节的说明，在使用 ODBC 功能时必须先申请环境句柄，然后在环境句柄的基础上创建数据库连接，最后在数据连接的基础上执行 SQL 语句。所以可能的调用方式有三种。

```
SQLAllocHandle(SQL_HANDLE_ENV,NULL,&hEnv);
SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,(SQLPOINTER) SQL_OV_ODBC3, SQL_IS_INTEGER);
SQLAllocHandle(SQL_HANDLE_DBC,hEnv,&hDBC);
SQLAllocHandle(SQL_HANDLE_STMT,hDBC,&hSTMT);
```

请注意，在创建环境句柄后请务必调用：

SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,(SQLPOINTER) SQL_OV_ODBC3, SQL_IS_INTEGER);

将 ODBC 设置成为版本 3，否则某些 ODBC API 函数不能被支持。

2.4.1.2 ODBC API 的返回值

ODBC API 的返回值定义为：SQLRETURN。在成功时返回值为：SQL_SUCCESS, SQL_SUCCESS_WITH_INFO；在失败时返回错误代码。

一点需要注意的是如果 ODBC 返回值为：SQL_SUCCESS_WITH_INFO 并不表明执行完全成功，而是表明执行成功但是带有一定错误信息。当执行错误时 ODBC 返回的是一个错误信息的结果集，你需要遍历结果集合中所有行，这点和后面讲到的查询 SQL 语句执行结果集的思路很类似。

在 ODBC 可以利用 SQLGetDiagRec 来得到错误描述信息：

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

RecNumber：指明需要得到的错误状态行，从 1 开始逐次增大。

Sqlstate，NativeErrorPtr，MessageText：返回错误状态，错误代码和错误描述。

BufferLength：指定 MessageText 的最大长度。

TextLengthPtr：指定返回的 MessageText 中有效的字符数。

函数的返回值可能为：SQL_SUCCESS，SQL_SUCCESS_WITH_INFO，SQL_ERROR，SQL_INVALID_HANDLE，SQL_NO_DATA。在没有返回错误的情况下你需要反复调用此函数，并顺次增大 RecNumber 参数的值，直到函数返回 SQL_NO_DATA，以得到所有的错误描述。

示例，得到 STMT 句柄上的错误信息：

```
SQLCHAR SqlState[6],SQLStmt[100],Msg[SQL_MAX_MESSAGE_LENGTH];
SQLINTEGER    NativeError;
SQLSMALLINT   i, MsgLen;
int i = 1;
while ((rc2 = SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, SqlState, &NativeError, Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA)
{
//显示错误的代码
    i++;
}
```

下面的函数来自 MS 文档，用于显示一个错误的详细情况，你可以在你的程序中直接使用：

```
void ProcessLogMessages(
            SQLSMALLINT plm_handle_type,
//出现错误时所使用的 ODBC 句柄类型，取值为：SQL_HANDLE_ENV， SQL_HANDLE_DBC，
```

```
SQL_HANDLE_STMT
                SQLHANDLE plm_handle, //出现错误时所使用的 ODBC 句柄
                char *logstring, //标题字符串
                int ConnInd //指明句柄是否为 DBC 句柄
    )
    {
      RETCODE     plm_retcode = SQL_SUCCESS;
      UCHAR     plm_szSqlState[MAXBUFLEN] = "",
            plm_szErrorMsg[MAXBUFLEN] = "";
      SDWORD     plm_pfNativeError = 0L;
      SWORD     plm_pcbErrorMsg = 0;
      SQLSMALLINT   plm_cRecNmbr = 1;
      SDWORD     plm_SS_MsgState = 0, plm_SS_Severity = 0;
      SQLINTEGER   plm_Rownumber = 0;
      USHORT     plm_SS_Line;
      SQLSMALLINT   plm_cbSS_Procname, plm_cbSS_Srvname;
      SQLCHAR     plm_SS_Procname[MAXNAME], plm_SS_Srvname[MAXNAME];

      printf(logstring);

      while (plm_retcode != SQL_NO_DATA_FOUND) {
        plm_retcode = SQLGetDiagRec(plm_handle_type, plm_handle,
          plm_cRecNmbr, plm_szSqlState, &plm_pfNativeError,
          plm_szErrorMsg, MAXBUFLEN - 1, &plm_pcbErrorMsg);

        // Note that if the application has not yet made a
        // successful connection, the SQLGetDiagField
        // information has not yet been cached by ODBC
        // Driver Manager and these calls to SQLGetDiagField
        // will fail.
        if (plm_retcode != SQL_NO_DATA_FOUND) {
          if (ConnInd) {
            plm_retcode = SQLGetDiagField(
              plm_handle_type, plm_handle, plm_cRecNmbr,
              SQL_DIAG_ROW_NUMBER, &plm_Rownumber,
              SQL_IS_INTEGER,
              NULL);
            plm_retcode = SQLGetDiagField(
              plm_handle_type, plm_handle, plm_cRecNmbr,
              SQL_DIAG_SS_LINE, &plm_SS_Line,
              SQL_IS_INTEGER,
              NULL);
            plm_retcode = SQLGetDiagField(
              plm_handle_type, plm_handle, plm_cRecNmbr,
```

```
            SQL_DIAG_SS_MSGSTATE, &plm_SS_MsgState,
            SQL_IS_INTEGER,
            NULL);
          plm_retcode = SQLGetDiagField(
            plm_handle_type, plm_handle, plm_cRecNmbr,
            SQL_DIAG_SS_SEVERITY, &plm_SS_Severity,
            SQL_IS_INTEGER,
            NULL);
          plm_retcode = SQLGetDiagField(
            plm_handle_type, plm_handle, plm_cRecNmbr,
            SQL_DIAG_SS_PROCNAME, &plm_SS_Procname,
            sizeof(plm_SS_Procname),
            &plm_cbSS_Procname);
          plm_retcode = SQLGetDiagField(
            plm_handle_type, plm_handle, plm_cRecNmbr,
            SQL_DIAG_SS_SRVNAME, &plm_SS_Srvname,
            sizeof(plm_SS_Srvname),
            &plm_cbSS_Srvname);
        }
        printf("szSqlState = %s\n",plm_szSqlState);
        printf("pfNativeError = %d\n",plm_pfNativeError);
        printf("szErrorMsg = %s\n",plm_szErrorMsg);
        printf("pcbErrorMsg = %d\n\n",plm_pcbErrorMsg);
        if (ConnInd) {
          printf("ODBCRowNumber = %d\n", plm_Rownumber);
          printf("SSrvrLine = %d\n", plm_Rownumber);
          printf("SSrvrMsgState = %d\n",plm_SS_MsgState);
          printf("SSrvrSeverity = %d\n",plm_SS_Severity);
          printf("SSrvrProcname = %s\n",plm_SS_Procname);
          printf("SSrvrSrvname = %s\n\n",plm_SS_Srvname);
        }
      }
      plm_cRecNmbr++; //Increment to next diagnostic record.
    } // End while.
}
```

2.4.1.3 SQLConnect 连接数据库

```
SQLRETURN SQLConnect(
    SQLHDBC     ConnectionHandle,
    SQLCHAR *     ServerName,
    SQLSMALLINT     NameLength1,
    SQLCHAR *     UserName,
    SQLSMALLINT     NameLength2,
```

SQLCHAR * Authentication,

SQLSMALLINT NameLength3);

ConnectionHanlde：为 DBC 句柄，也就是前面提到到利用：

SQLAllocHandle(SQL_HANDLE_DBC,hEnv,&hDBC);申请的句柄。

ServerName：为 ODBC 的 DSN 名称。

NameLength1：指明参数 ServerName 数据的长度。

UserName：数据库用户名。

NameLength2：指明参数 UserName 数据的长度。

Authentication：数据库用户密码。

NameLength3：指明参数 Authentication 数据的长度。

关于 ServerName，UserName，Authentication 参数长度可以直接指定也可以指定为 SQL_NTS 表明参数是以 NULL 字符结尾。

示例代码：

retcode = SQLConnect(hdbc, (SQLCHAR*) "odbc_demo", SQL_NTS,(SQLCHAR*) "user", SQL_NTS, (SQLCHAR*) "password", SQL_NTS);

2.4.1.4 SQLExecDirect 直接执行 SQL 语句

SQLRETURN SQLExecDirect(

SQLHSTMT StatementHandle,

SQLCHAR * StatementText,

SQLINTEGER TextLength);

StatementHandle：SQL 语句句柄，也就是前面提到的利用：

SQLAllocHandle(SQL_HANDLE_STMT,hDBC,&hSTMT);申请的句柄。

StatementText：SQL 语句。

TextLength：参数 StatementText 的长度，可以使用 SQL_NTS 表示字符串以 NULL 字符结尾。

如果函数执行成功，你将会得到一个结果集，否则将返回错误信息。

SQLExecDirect 函数除可以执行 Select 语句外，还可以执行 Insert，Update，Delete 语句，在执行修改 SQL 语句后可以利用 SQLRowCount 函数来得到被更新的记录的数量。

2.4.1.5 SQLFetch 移动光标

SQLRETURN SQLFetch(SQLHSTMT StatementHandle);

在你调用 SQLExecDirect 执行 SQL 语句后，你需要遍历结果集来得到数据。StatementHandle 是 STMT 句柄，此句柄必须是被执行过。

当调用 SQLFetch 函数后，光标会被移动到下一条记录处，当光标移动到记录集的最后一条，函数将会返回 SQL_NO_DATA。

要遍历所有的结果集可以利用下面的方法：

```
while(SQL_NO_DATA != SQLFetch(hSTMT)) //移动光标，一直到集合末尾
{//得到结果
}
```

2.4.1.6 SQLGetData 得到光标处的某列的值

```
SQLRETURN SQLGetData(
    SQLHSTMT      StatementHandle,
    SQLUSMALLINT      ColumnNumber,
    SQLSMALLINT      TargetType,
    SQLPOINTER      TargetValuePtr,
    SQLINTEGER      BufferLength,
    SQLINTEGER *      StrLen_or_IndPtr);
```

StatementHanlde：STMT 句柄。

ClumnNumber：列号，以 1 开始。

TargetType：数据缓冲区（TargetValuePtr）的 C 语言数据类型，请参照图 2.6。

BufferLength：数据缓冲区（TargetValuePtr）的长度。

StrLen_or_IndPtr：返回当前得到的字段的字节长度。

下面是通过 SQLFetch 和 SQLGetData 得到记录集的例子：

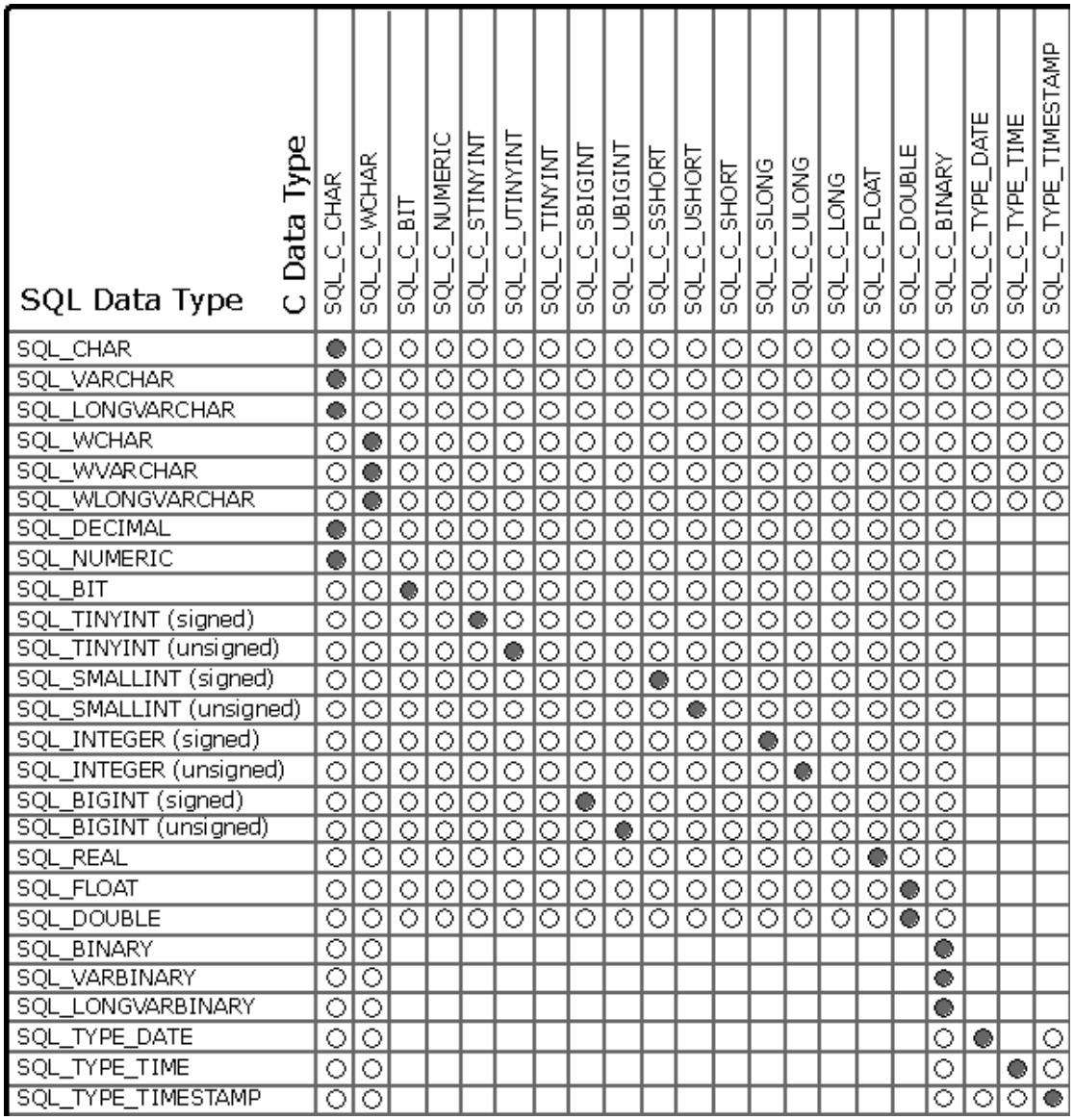
```
//假设 SQL = SELECT CUSTID, NAME, PHONE FROM CUSTOMERS
  SQLINTEGER   sCustID
  SQLCHAR   szName[50], szPhone[50];
  SQLINTEGER cbName, cbAge, cbBirthday;//用来保存得到的数据的长度
  while (TRUE) {//循环得到所有行
    retcode = SQLFetch(hstmt);//移动光标
    if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO) {
      printf("error SQLFetch\n");
    }
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){
      /*得到当前光标处每列的值 */
      SQLGetData(hstmt, 1, SQL_C_ULONG, &sCustID, 0, &cbCustID);
     //此处并没有指明 BufferLength 参数的值，是因为数据类型是定长的 LONG 型
      SQLGetData(hstmt, 2, SQL_C_CHAR, szName, 50, &cbName);
      SQLGetData(hstmt, 3, SQL_C_CHAR, szPhone, 50,&cbPhone);
      printf(out, "%5d %s %s", sCustID,  szName, szPhone);
    } else {
      break;
    }
  }
```

SQLGetData 的另一个用处就是用于得到一些变长字段的实际长度，例如 VARCHAR 字段，TEXT 字段。例如：

```
SQLGetData(hstmt, 2, SQL_C_CHAR, szName, 0, &cbName);
```

当你将 BufferLength 参数置为 0，则会在 StrLen_or_IndPtr 参数中返回字段的实际长度。但请注意第四个参数必须是一个合法的指针，不能够为 NULL。

此外在得到字段的值时还存在一个数据类型转换的问题，比如说数据库内的字段类型为：INTEGER，那么你可以使用 SQL_C_INTEGER，SQL_C_CHAR，SQL_C_ULONG 数据类型来得到该字段的值。图 2.7 就说明了所有可能存在的转换关系。



SQL Data Type \ C Data Type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_BIT	SQL_C_NUMERIC	SQL_C_STINYINT	SQL_C_UTINYINT	SQL_C_TINYINT	SQL_C_SBIGINT	SQL_C_UBIGINT	SQL_C_SSHORT	SQL_C_USHORT	SQL_C_SHORT	SQL_C_SLONG	SQL_C_ULONG	SQL_C_LONG	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_BINARY	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	SQL_C_TYPE_TIMESTAMP
SQL_CHAR	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_VARCHAR	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_LONGVARCHAR	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_WCHAR	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_WVARCHAR	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_WLONGVARCHAR	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_DECIMAL	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			
SQL_NUMERIC	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			
SQL_BIT	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			
SQL_TINYINT (signed)	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○			
SQL_TINYINT (unsigned)	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○			
SQL_SMALLINT (signed)	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○			
SQL_SMALLINT (unsigned)	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○			
SQL_INTEGER (signed)	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○			
SQL_INTEGER (unsigned)	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○			
SQL_BIGINT (signed)	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○			
SQL_BIGINT (unsigned)	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○			
SQL_REAL	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○			
SQL_FLOAT	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○			
SQL_DOUBLE	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○			
SQL_BINARY	○	○																●			
SQL_VARBINARY	○	○																●			
SQL_LONGVARBINARY	○	○																●			
SQL_TYPE_DATE	○	○																○	●		○
SQL_TYPE_TIME	○	○																○		●	○
SQL_TYPE_TIMESTAMP	○	○																○	○	○	●

图 2.7

参考前面的图 2.5 和图 2.6 就可以看出 ODBC 可以在 SQL 的数据类型和 C 的数据类型间提供转换。

请注意我讲解图 2.5 时提到的时 ODBC 数据类型和 SQL 语言数据类型之间的对应关系，而这里提到的是数据之间的转换关系。

图中用圆点标出的交叉点为允许的类型转换，其中实心圆点标出的是默认的数据转换类型，空心圆点标出的是允许的数据转换类型（允许并不表明一定可以，例如 Char 类型可以转换为 SQL_C_INTEGER，但是并不是总能成功，当字符为'123'时可以转换为整数 123，而字符为'odbc'时就不能成功）。例如数据库中的 Char，VarChar 类型默认都是对应 SQL_C_CHAR 类型。你还可以看到所有的数据库字段类型都可以转换为 SQL_C_CHAR 类型，例如整数 123 可以转换为"123"，而浮点数 1.23 可以转换为"1.23"，日期的 2002 年 10 月 1 日可以转换为"2002-10-1"。所以在初学 ODBC 并且对性能要求不是非常高时可以用字符类型来得到数据库字段的值，这样做会比较方便。

2.4.1.7 SQLNumResultCols 得到结果集中列数

```
SQLRETURN SQLNumResultCols(
    SQLHSTMT     StatementHandle,
```

SQLSMALLINT * ColumnCountPtr);

StatementHandle：STMT 句柄

ColumnCountPtr：返回列数

2.4.1.8 SQLDescribeCol 得到结果集中列的描述

SQLRETURN SQLDescribeCol(
SQLHSTMT StatementHandle,
SQLSMALLINT ColumnNumber,
SQLCHAR * ColumnName,
SQLSMALLINT BufferLength,
SQLSMALLINT * NameLengthPtr,
SQLSMALLINT * DataTypePtr,
SQLUINTEGER * ColumnSizePtr,
SQLSMALLINT * DecimalDigitsPtr,
SQLSMALLINT * NullablePtr);

StatementHandle：STMT 句柄。

ColumnNumber：需要得到的列的序号，从 1 开始计算。

ColumnName：得到列的名称。

BufferLength：指明 ColumnName 参数的最大长度。

NameLengthPtr：返回列名称的长度。

DataTypePtr：得到列的 ODBC 数据类型，请参照图 2.5。

ColumnSizePtr：得到列的长度。

DecimalDigitsPtr：当该列为数字类型时返回小数点后数据的位数。

NullablePtr：指明该列是否允许为空值。

2.4.1.9 SQLRowCount 执行 SQL 语句后得到影响的行数

SQLRETURN SQLRowCount(
SQLHSTMT StatementHandle,
SQLINTEGER * RowCountPtr);

你可以通过 SQLExecDirect 执行 SQL 语句来插入，修改和删除数据，在执行插入，修改和删除的 SQL 语句后就可以通过 SQLRowCount 函数来得到被影响的数据的行数。

Insert into test_t1 values(4, '2002-1-4 11:25' , 'user_4',1.86);

2.5 ODBC 的其他功能介绍

2.5.1 ODBC 连接句柄的参数设置

在上一小节中提到了 SQLSetConnectAttr 这个函数，这里需要对这个函数进行一些简单的讲解，你可以通过调用 SQLSetConnectAttr 在数据库连接建立或建立后设置连接的一些属性。

SQLSetConnectAttr 的函数原型如下：

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

ConnectionHandle：提供 DBC 连接句柄。

Attribute：指定需要设置的属性类型，在这里设置为值 SQL_ATTR_AUTOCOMMIT。

ValuePtr：提供参数值。

StringLength：指定参数的长度，当参数为整数是设置为 SQL_IS_INTEGER，当参数为字符串时设置为字符串长度或者为 SQL_NTS 。

这里讲一下常用的参数 Attribute 可能的取值和 ValuePtr 对应的取值：

Attribute	ValuePtr	作用
SQL_ATTR_CONNECTION_DEAD	提供一个合法的 SQLINTEGER 指针作为输出参数	检查连接是否已经断开 返回：SQL_CD_TRUE 表明连接已经断开，SQL_CD_FALSE 表明连接还保持。
SQL_ATTR_CONNECTION_TIMEOUT	设置一个合法的整数	建立与数据库连接时最大等待的超时秒数，ValuePtr 设置为 0 表明不使用超时控制。
SQL_ATTR_LOGIN_TIMEOUT	设置一个合法的整数	建立与数据库用户登录时最大等待的超时秒数，ValuePtr 设置为 0 表明不使用超时控制。
SQL_ATTR_TRACE	整数，取值为：SQL_OPT_TRACE_OFF，SQL_OPT_TRACE_ON	设置是否跟踪 ODBC 驱动程序中函数的调用。
SQL_ATTR_TRACEFILE	设置一个合法的文件名字符串，表明记录跟踪记录的文件名。	设置记录函数调用的文件名称。

2.5.2 ODBC 语句句柄的参数设置

如同数据库连接句柄一样，语句句柄也可以设置参数。函数为：

```
SQLRETURN SQLSetStmtAttr(
    SQLHSTMT     StatementHandle,
    SQLINTEGER    Attribute,
    SQLPOINTER    ValuePtr,
    SQLINTEGER    StringLength);
```

ConnectionHandle：提供 STMT 连接句柄。

Attribute：指定需要设置的属性类型，在这里设置为值 SQL_ATTR_AUTOCOMMIT。

ValuePtr：提供参数值。

StringLength：指定参数的长度，当参数为整数是设置为 SQL_IS_INTEGER，当参数为字符串时设置为字符串长度或者为 SQL_NTS 。

这里讲一下常用的参数 Attribute 可能的取值和 ValuePtr 对应的取值：

Attribute	ValuePtr	作用
SQL_ATTR_ASYNC_ENABLE	整数，取值为： SQL_ASYNC_ENABLE_OFF， SQL_ASYNC_ENABLE_ON	是否使用异步执行功能
SQL_ATTR_QUERY_TIMEOUT	设置一个合法的整数	SQL 语句执行时的超时秒数，设置为 0 表示无超时
SQL_ATTR_CURSOR_TYPE	整数，取值为： SQL_CURSOR_FORWARD_ONLY， SQL_CURSOR_STATIC， SQL_CURSOR_DYNAMIC， SQL_CURSOR_KEYSET_DRIVEN	设置光标的类型

2.5.3 ODBC 中使用可以滚动的光标

2.5.3.1 ODBC 光标类型

从上面的函数 SQLSetStmtAttr 可以看到我们在 ODBC 中可以使用不同的光标类型，那么这些光标之间有什么区别。

- 向前光标：SQL_CURSOR_FORWARD_ONLY，光标仅仅向前滚动。
- 静态光标：SQL_CURSOR_STATIC，结果集的数据是静态的，这就是说明在执行查询后，返回的结果集的数据不会再改变，即使是有其他程序更新了数据库中的记录，结果集中的记录也不会发生改变。
- 动态光标：SQL_CURSOR_DYNAMIC，在光标打开以后，当结果集中的行所对应的数据值发生变化时，其变化能够能够反映到光标所对应的结果集上，这些变化包括：字段的修改，添加，结果集中行的顺序变化。但是请注意如果行别删除则无法在当前结果集中反映出，因为被删除的行不再出现在当前的结果集中。动态光标所对应的结果集在数据发生变化时会被重建。例如，假设动态光标已获取到了两行，然后，另一应用程序更新了这两行中的一行，并删除了另一行，如果动态游标再试图获取那些行，它将不能检测已删除的行（因为当前结果集中只有一行，但是不要利用这个办法去检测被删除的行，因为出现这种情况还可能是因为行的数据被改变后不能再满足查询条件），而是返回已更新行的新值。
- 键集光标：SQL_CURSOR_KEYSET_DRIVEN，和上面的动态光标所不同的是键集光标能够检测到行的删除和修改，但是无法检测到检测到行的添加和结果集顺序变化。因为在光标创建时就创建了整个结果集，结果集合中记录和顺序已经被固定，这一点和静态光标一样。所以键集光标可以说是一种介于静态光标和动态光标之间的光标类型。

需要说明的是并不是每个 DBMS 的 ODBC 驱动程序都能够支持所有的这几种光标，具体情况可以通过 SQLGetInfo 函数进行查询。

2.5.3.2 利用可滚动光标进行查询

前面介绍的 SQLFetch 函数只能够让光标向前移动，但在很多时候我们需要光标能够前后移动。我们需要利用另一个函数 SQLFetchScroll，但是再这之前请利用 SQLSetStmtAttr 正确设置光标类型。SQLFetchScroll 的原型如下：

SQLRETURN SQLFetchScroll(
SQLHSTMT StatementHandle,
SQLSMALLINT FetchOrientation,
SQLINTEGER FetchOffset);

与 SQLFetch 不同的是多了后面两个参数。

FetchOrientation：表明滚动的方式，允许的值如下：

FetchOrientation	含义
SQL_FETCH_NEXT	滚动到下一行，这时候调用相当与 SQLFetch，参数 FetchOffset 将被忽略
SQL_FETCH_PRIOR	滚动到上一行，参数 FetchOffset 将被忽略
SQL_FETCH_FIRST	滚动到第一行，参数 FetchOffset 将被忽略
SQL_FETCH_LAST	滚动到最后一行，参数 FetchOffset 将被忽略
SQL_FETCH_ABSOLUTE	滚动到参数 FetchOffset 指定的绝对行
SQL_FETCH_RELATIVE	由当前位置滚动到参数 FetchOffset 指定的相对行，FetchOffset 大于 0 表示向前滚动，FetchOffset 小于 0 表示向后滚动

FetchOffset：表明光标滚动的位置。

光标滚动后，获取数据的方法和 SQLFetch 相同。滚动时如果指定的位置超出结果集区域会返回错误。

2.5.4 存储过程的执行与参数的绑定

在 ODBC 中调用存储过程需要遵守 ODBC 的约定，SQL 语句的格式为：

{[?=]call procedure-name[([parameter][,[parameter]]...)]}

在这里必须讲解问号在 SQL 语句中的作用，问号在 SQL 语句中代表参数的含义，这个参数可以是作为输入，也可以作为输出或者是既输入又输出，参数在 SQL 语句执行后进行指定，后面的章节中会进一步讲解参数的用法。

其中第一问号的作用是取得存储过程的返回值，在执行的过程中必须将此参数绑定到某一个变量，在执行结束后被绑定的变量中就会被赋值。

后面的参数可以是在 SQL 语句中直接指定，例如：

{ call insert_this (1 , ‘myName’) }

也可以利用参数来指定例如：

{ call insert_this (1 , ?) }

参数的值会在 SQL 语句执行的过程中进行指定。

执行存储过程直接利用 SQLExecDirect API 函数就可以了。但是存储过程中很多时候会返回结果，这些结果必须用参数的形式才能够得到，所以下面要介绍一下如何利用参数绑定的方法来得到存储过程的返回值，此外利用参数绑定还可以动态的向存储过程提供参数。

但是进行参数绑定就需要使用另一个 ODBC API：

SQLRETURN SQLBindParameter(
SQLHSTMT StatementHandle,
SQLUSMALLINT ParameterNumber,
SQLSMALLINT InputOutputType,
SQLSMALLINT ValueType,
SQLSMALLINT ParameterType,
SQLUINTEGER ColumnSize,

SQLSMALLINT DecimalDigits,
SQLPOINTER ParameterValuePtr,
SQLINTEGER BufferLength,
SQLINTEGER * StrLen_or_IndPtr);

StatementHandle：执行 SQL 语句 STMT 句柄。

ParameterNumber：指明要将变量与第几个参数绑定，从 1 开始计算。

InputOutputType：指明是输入还是输出参数。可以取值的范围为：SQL_PARAM_INPUT，SQL_PARAM_OUTPUT，SQL_PARAM_INPUT_OUTPUT。

ValueType：指明用于和参数绑定的 C 语言数据类型。

ParameterType：指明在存储过程中 ODBC 数据类型。

ColumnSize：指明接收数据的宽度，对于字符串和结构需要指明数据的宽度，而对于普通的变量如 SQLINTEGER，SQLFLOAT 等设置为 0 就可以了。

DecimalDigits：当数据类型为 SQL_NUMERIC，SQL_DECIMAL 时指明数字小数点的精度，否则填 0。

ParameterValuePtr：在作为输入参数指明参数的指针，在作为输出参数时指明接收数据的变量指针。

BufferLength：指明参数指针所指向的缓冲区的字节数大小。对于字符串和结构需要指明大小，而对于普通的变量如 SQLINTEGER，SQLFLOAT 等设置为 0 就可以了。

StrLen_or_IndPtr：作为输入参数时指明数据的字节数大小，对于普通的定长变量如 SQLINTEGER，SQLFLOAT 等设置为 0 就可以了，对于字符号串需要在此参数中指定字符串数据的长度，或者设置为 SQL_NULL_DATA 表明此参数为空值，或者设置为 SQL_NTS 表明字符串以 NULL 字符结尾，对于结构需要指明结构的长度。当作为输出参数时，当 SQL 执行完毕后会在这个参数中返回拷贝的缓冲区的数据的字节数。

2.5.4.1 例子一：调用含有输出参数的存储过程

在 SQL Server 中建立存储过程：

```
create proc p_f
    @myid int output,@myname varchar(20) output,@mytall float output,@mytall2 dec(30,10) output,
    @inid int,@inname varchar(10),@intall float,@intall2 dec(30,10)
as
    print 'do it'
    set @myid=1+isnull(@inid,-1)
    set @myname='test'+isnull(@inname,'_NULL')
    set @mytall=100.100 + @intall
    set @mytall2=100.100 + @intall2
    return 1024
```

这个存储过程的前四个参数是输出参数，后四个参数是输入参数。并且最后返回 1024 作为返回值。四个参数分别是不同的数据类型：整数，字符串，浮点数和带有 10 位小数的 30 位数字。

这个例子中我们不使用后面四个参数，只演示输出参数的使用方法，后四个输出参数直接在 SQL 语句中指定，程序代码如下。

```
void OnTestP4()
{
//省略分配句柄和连接数据库的部分
        SQLCHAR szOutput[40],szOutput2[40];
```

```
SQLINTEGER iOutput=0,iReturnVal=0;
SQLFLOAT fOutput=0;
//只使用输出参数
SQLCHAR szSQL[100]="{?=call p_f(?,?,?,?,1,'_OK',1000.001,1000.001)}";
//用于保存各个返回的输出参数的字节数
SQLINTEGER cb1,cb2,cb3,cb4,cb5;
//分配 STMT 句柄
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
//绑定参数
retcode = SQLBindParameter(hstmt1, 1, SQL_PARAM_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iReturnVal, 0, &cb1);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para1() Failed\n\n", TRUE);
}
retcode = SQLBindParameter(hstmt1, 2, SQL_PARAM_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iOutput, 0, &cb2);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para2() Failed\n\n", TRUE);
}
retcode = SQLBindParameter(hstmt1, 3, SQL_PARAM_OUTPUT, SQL_C_CHAR,SQL_VARCHAR, 20, 0, szOutput, 20, &cb3);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para3() Failed\n\n", TRUE);
}
retcode = SQLBindParameter(hstmt1, 4, SQL_PARAM_OUTPUT, SQL_C_DOUBLE,SQL_FLOAT, 0, 0, &fOutput, 0, &cb4);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para4() Failed\n\n", TRUE);
}
retcode = SQLBindParameter(hstmt1, 5, SQL_PARAM_OUTPUT, SQL_C_CHAR,SQL_DECIMAL, 25, 10, szOutput2, 40, &cb5);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para5() Failed\n\n", TRUE);
}
//执行 SQL 语句，调用存储过程
retcode = SQLExecDirect (hstmt1,szSQL, SQL_NTS);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"SQLExecute() Failed\n\n", TRUE);
```

```
            }
            else
                    //得到结果集
                    while ( ( retcode = SQLMoreResults(hstmt1) ) != SQL_NO_DATA ) ;

            TRACE("%d %d %s %f %s\n",iReturnVal,iOutput,szOutput,fOutput,szOutput2);
            SQLFreeHandle(SQL_HANDLE_STMT,hstmt1);
    //省略释放句柄部分
    }
```

我们现在来分别分析返回值和四个参数绑定的调用。

绑定返回值和第一个整数：

retcode = SQLBindParameter(hstmt1, 1, SQL_PARAM_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iReturnVal, 0, &cb1);

ParameterNumber ＝1，2

InputOutputType ＝ SQL_PARAM_OUTPUT，指明作为输出参数。

ValueType ＝ SQL_C_LONG，指明 C 语言中数据类型为 int 。

ParameterType ＝ SQL_INTEGER，指明 SQL 数据类型为 Integer。

ColumnSize ＝ 0

DecimalDigits ＝ 0

ParameterValuePtr ＝ &iReturnVal , &iOutput 为保存返回参数的缓冲区（变量）指针。

BufferLength ＝ 0

StrLen_or_IndPtr ＝ &cb1，cb1 未赋初值，用于得到返回参数的字节大小。

绑定第二个字符串：

retcode = SQLBindParameter(hstmt1, 3, SQL_PARAM_OUTPUT, SQL_C_CHAR,SQL_VARCHAR, 20, 0, szOutput, 20, &cb3);

ParameterNumber ＝3

InputOutputType ＝ SQL_PARAM_OUTPUT，指明作为输出参数。

ValueType ＝SQL_C_CHAR，指明 C 语言中数据类型为 char[ ] 。

ParameterType ＝ SQL_VARCHAR，指明 SQL 数据类型为 Varchar。

ColumnSize ＝ 20 ，指明字符串宽度为 20。

DecimalDigits ＝ 0

ParameterValuePtr ＝ szOutput 为保存返回参数的缓冲区（变量）指针。

BufferLength ＝ 20 ，指明缓冲区的最大字节数。

StrLen_or_IndPtr ＝ &cb3，cb3 未赋初值，用于得到返回参数的字节大小。

绑定第三个浮点数：

retcode = SQLBindParameter(hstmt1, 4, SQL_PARAM_OUTPUT, SQL_C_DOUBLE,SQL_FLOAT, 0, 0, &fOutput, 0, &cb4);

ParameterNumber ＝4

InputOutputType ＝ SQL_PARAM_OUTPUT，指明作为输出参数。

ValueType ＝SQL_C_DOUBLE，指明 C 语言中数据类型为 double 。

ParameterType ＝ SQL_FLOAT，指明 SQL 数据类型为 float 或 double。

ColumnSize ＝ 0

DecimalDigits ＝ 0

ParameterValuePtr ＝&fOutput 为保存返回参数的缓冲区（变量）指针。

BufferLength ＝ 0

StrLen_or_IndPtr ＝ &cb4，cb4 未赋初值，用于得到返回参数的字节大小。

绑定第四个数字：

retcode = SQLBindParameter(hstmt1, 5, SQL_PARAM_OUTPUT, SQL_C_CHAR,SQL_DECIMAL, 25, 10, szOutput2, 40, &cb5);

ParameterNumber ＝4

InputOutputType ＝ SQL_PARAM_OUTPUT，指明作为输出参数。

ValueType ＝SQL_C_ CHAR，指明 C 语言中数据类型为 char[ ]，虽然可以使用 SQL_NUMERIC_STRUCT 结构来进行参数绑定，不过 ODBC 中默认的类型是 SQL_C_CHAR（见图 2.6）。

ParameterType ＝ SQL_ DECIMAL，指明 SQL 数据类型为 Dec 或 Number。

ColumnSize ＝ 25 ，指明数据的宽度，请注意在存储过程中定义的参数为 dec(30, 10)表明参数为 30 位，而实际上返回的不足 30 位，所以这里可以指定一个比较小的值，但是如果返回的数的位数如果超过 25 位，将无法得到正确结果。

DecimalDigits ＝ 10 ，指明小数点后位数为 10 位。

ParameterValuePtr ＝&fOutput 为保存返回参数的缓冲区（变量）指针。

BufferLength ＝4 0 ，指明缓冲区的最大字节数。

StrLen_or_IndPtr ＝ &cb5，cb5 未赋初值，用于得到返回参数的字节大小。

2.5.4.2 例子二：调用含有输入和输出参数的存储过程

在这个例子里我们会处理输入参数，输出参数绑定和既输入又输出的参数绑定。此外我们还可以看看如何传递空值。

这个例子所用到的存储过程是在前一个存储过程的基础上修改的，前面的参数完全相同，而最后 4 个参数为既输入又输出的参数。

```
create proc p_f2
    @myid int output,@myname varchar(20) output,@mytall float output,@mytall2 dec(30,10) output,
    @inid int,@inname varchar(10),@intall float,@intall2 dec(30,10),
    @ioid int output,@ioname varchar(10) output,@iotall float output,@iotall2 dec(30,10) output
as
    print 'do it'
    set @myid=1+isnull(@inid,-1)
    set @myname='Test2'+@inname
    set @mytall=200.200 + @intall
    set @mytall2=200.200 + @intall2

    set @ioid=@ioid+1
    set @ioname='test'+@ioname
    set @iotall=200.200 + @iotall
```

```
    set @iotall2=200.200 + @iotall2
    return 2048
```

代码如下。

```
void OnTestP6()
{
        SQLCHAR szOutput[40],szOutput2[40];
        SQLINTEGER iOutput=0,iReturnVal=0;
        SQLFLOAT fOutput=0;

        SQLCHAR szInput[40]="_OK3",szInput2[40]="9876543210.9876543210";
        SQLINTEGER iInput=1000;
        SQLFLOAT fInput=1000.1;

        SQLCHAR szIO[40]="_IO",szIO2[40]="102030.102030";
        SQLINTEGER iIO=2000;
        SQLFLOAT fIO=3000.3;

        SQLCHAR szSQL[100]="{?=call p_f2(?,?,?,?,?,?,?,?,?,?,?,?,?)}";
        SQLINTEGER cb1,cb2,cb3,cb4,cb5;
        SQLINTEGER cb6=0,cb7=SQL_NTS,cb8=0,cb9=SQL_NTS;
        SQLINTEGER cb10=0,cb11=SQL_NTS,cb12=0,cb13=SQL_NTS;
        //分配 STMT 句柄
        retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
        // output 此部分代码在前面出现过，此处省略
        // input
        cb6= SQL_NULL_DATA;
        retcode = SQLBindParameter(hstmt1, 6, SQL_PARAM_INPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, NULL,
0, &cb6);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
                ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para6() Failed\n\n", TRUE);
        }
        retcode = SQLBindParameter(hstmt1, 7, SQL_PARAM_INPUT, SQL_C_CHAR,SQL_VARCHAR, 20, 0,
szInput, 20, &cb7);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
                ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para7() Failed\n\n", TRUE);
        }
        retcode = SQLBindParameter(hstmt1, 8, SQL_PARAM_INPUT, SQL_C_DOUBLE,SQL_FLOAT, 0, 0,
&fInput, 0, &cb8);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
                ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para8() Failed\n\n", TRUE);
        }
```

```
		retcode = SQLBindParameter(hstmt1, 9, SQL_PARAM_INPUT, SQL_C_CHAR,SQL_DECIMAL, 25, 10, szInput2, 40, &cb9);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para9() Failed\n\n", TRUE);
		}
		// input & output
		retcode = SQLBindParameter(hstmt1, 10, SQL_PARAM_INPUT_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iIO, 0, &cb10);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para10() Failed\n\n", TRUE);
		}
		retcode = SQLBindParameter(hstmt1, 11, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR,SQL_VARCHAR, 20, 0, szIO, 20, &cb11);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para11() Failed\n\n", TRUE);
		}
		retcode = SQLBindParameter(hstmt1, 12, SQL_PARAM_INPUT_OUTPUT, SQL_C_DOUBLE,SQL_FLOAT, 0, 0, &fIO, 0, &cb12);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para12() Failed\n\n", TRUE);
		}
		retcode = SQLBindParameter(hstmt1, 13, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR,SQL_DECIMAL, 25, 10, szIO2, 40, &cb13);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para13() Failed\n\n", TRUE);
		}
		// execute
		retcode = SQLExecDirect (hstmt1,szSQL, SQL_NTS);
		if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
		{
			ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"SQLExecute() Failed\n\n", TRUE);
		}
		else
			while ( ( retcode = SQLMoreResults(hstmt1) ) != SQL_NO_DATA ) ;

		TRACE("%d %d %s %f %s\n",iReturnVal,iOutput,szOutput,fOutput,szOutput2);
		TRACE("%d %s %f %s\n",iIO,szIO,fIO,szIO2);
		SQLFreeHandle(SQL_HANDLE_STMT,hstmt1);
	}
```

首先讲一下如何提供 NULL 值，大家看看第 5 个参数的调用语句。

cb6= SQL_NULL_DATA;

retcode = SQLBindParameter(hstmt1, 6, SQL_PARAM_INPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, NULL, 0, &cb6);

关键在与最后一个参数设置为 SQL_NULL_DATA。而 cb7 = SQL_NTS 表示字符串以 NULL 字符结尾，你可以将 cb7 赋值为 strlen(szInput) 函数也可以正常执行。

你对比输入参数，输出参数和既输入又输出参数的调用方式，你会发现非常类似，区别主要在最后一个参数是否需要先赋值和执行完成后是否会返回数据。

输出参数：retcode = SQLBindParameter(hstmt1, 2, SQL_PARAM_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iOutput, 0, &cb2);

输入参数：retcode = SQLBindParameter(hstmt1, 6, SQL_PARAM_INPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, NULL, 0, &cb6);

输入输出参数：retcode = SQLBindParameter(hstmt1, 10, SQL_PARAM_INPUT_OUTPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iIO, 0, &cb10);

2.5.5 SQL 的准备与执行

在 ODBC 中除支持 SQL 语句的直接执行（利用 SQLExecDirect）外还可以使用准备－执行的方式，在准备阶段 ODBC 会分析 SQL 语句并分配各种资源，在执行阶段才正式执行刚才准备好的 SQL 语句。在这中模式下会使用下面的两个 API 函数。

准备需要执行的 SQL 语句：

```
SQLRETURN SQLPrepare(
    SQLHSTMT     StatementHandle,
    SQLCHAR *    StatementText,
    SQLINTEGER    TextLength);
```

StatementHandle：STMT 句柄。

StatementText：包含 SQL 语句的字符串。

TextLength：SQL 语句的长度，或者使用 SQL_NTS 表示 SQL 语句以 NULL 字符结尾。

执行经过准备的 SQL 语句：

```
SQLRETURN SQLExecute(SQLHSTMT  StatementHandle);
```

在执行 SQL 语句时可以利用 SQLPrepare 和 SQLExecute 的组合来代替 SQLExecDirect 函数。

使用 SQLExecute 执行和 SQLExecDirect 执行后的 STMT 句柄可以进行相同的操作，例如使用 SQLFetch 取得结果集。不过准备－执行模式一般用来执行各种大批量的数据修改，而不是用来执行查询语句。此外对于那些需要反复执行的 SQL 语句利用准备－执行模式会比反复执行 SQLExecDirect 速度快。当执行数据修改时可以采用前面提到过的参数绑定，并且每执行一次重新后修改参数的值。

首先在数据库内添加下面的测试表：create table test_insert(testid int,testname varchar(20)) 。

下面的例子使用准备－执行的模式来插入数据，第一个 SQL 语句是删除语句，第二个语句是插入语句，利用参数绑定来修改每次插入的实际值。在每次执行后利用 SQLRowCount 函数得到修改的数据行的数量。

```
void OnTestP7()
{
        SQLCHAR szInput[20];
        SQLINTEGER iInput=0,iRowCount=0;
        SQLCHAR szSQL1[200]="delete from test_insert",szSQL2[100]="insert into test_insert values(?,?)";
```

```
SQLINTEGER cb1=0,cb2=SQL_NTS;

//delete
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
retcode = SQLPrepare(hstmt1,szSQL1,SQL_NTS);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"delete SQLPrepare() Failed\n\n", TRUE);
}
retcode = SQLExecute(hstmt1);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"delete SQLExecute() Failed\n\n", TRUE);
}
SQLRowCount(hstmt1,&iRowCount);
printf("delete effect row count = %d\n",iRowCount);
SQLFreeHandle(SQL_HANDLE_STMT,hstmt1);
// insert
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
retcode = SQLPrepare(hstmt1,szSQL2,SQL_NTS);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"Insert SQLPrepare() Failed\n\n", TRUE);
}
retcode = SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_LONG,SQL_INTEGER, 0, 0, &iInput,
0, &cb1);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para1() Failed\n\n", TRUE);

}
retcode = SQLBindParameter(hstmt1, 2, SQL_PARAM_INPUT, SQL_C_CHAR,SQL_VARCHAR, 20, 0,
szInput, 20, &cb2);
if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
{
        ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind para2() Failed\n\n", TRUE);

}
for(int i=0;i<20;i++)
{//循环插入
        iInput = i;
        sprintf((char*)szInput,"name:%d",20-i);
        retcode = SQLExecute(hstmt1);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
```

```
                    ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"insert SQLExecute() Failed\n\n",
TRUE);
                }
                SQLRowCount(hstmt1,&iRowCount);
                printf("insert(%d,%s) effect row count = %d\n",iInput,szInput,iRowCount);
            }
            SQLFreeHandle(SQL_HANDLE_STMT,hstmt1);
    }
```

2.5.6 通过列绑定获取字段数据

在从结果集中读取字段值时可以利用 SQLGetData，但为了速度可以利用列绑定的方式，在每次移动光标后让 ODBC 将数据传送到指定的变量中。

```
SQLRETURN SQLBindCol(
    SQLHSTMT     StatementHandle,
    SQLUSMALLINT     ColumnNumber,
    SQLSMALLINT     TargetType,
    SQLPOINTER     TargetValuePtr,
    SQLINTEGER     BufferLength,
    SQLLEN *     StrLen_or_Ind);
```

StatementHandle：STMT 句柄。

ColumnNumber：列的位置，从 1 开始计算。

ValueType：指明用于和参数绑定的 C 语言数据类型。

ParameterType：指明在存储过程中 ODBC 数据类型。

BufferLength：指明参数指针所指向的缓冲区的字节数大小。对于字符串和结构需要指明大小，而对于普通的变量如 SQLINTEGER，SQLFLOAT 等设置为 0 就可以了。

StrLen_or_IndPtr：返回拷贝的缓冲区的数据的字节数。

```
void OnTestP8()
{
        SQLCHAR szOutput[20];
        SQLINTEGER iOutput=0;
        SQLCHAR szSQL[100]="select testid,testname from test_insert";
        SQLINTEGER cb1=0,cb2=SQL_NTS;

        retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
        retcode = SQLExecDirect(hstmt1,szSQL, SQL_NTS);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
            ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"SQLExecute() Failed\n\n", TRUE);
        }
        retcode = SQLBindCol(hstmt1, 1, SQL_C_LONG,&iOutput, 0, &cb1);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
```

```
        {
                ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind1 Failed\n\n", TRUE);
        }
        retcode = SQLBindCol(hstmt1, 2, SQL_C_CHAR,szOutput, 20, &cb2);
        if ( (retcode != SQL_SUCCESS) && (retcode != SQL_SUCCESS_WITH_INFO) )
        {
                ProcessLogMessages(SQL_HANDLE_STMT, hstmt1,"bind2 Failed\n\n", TRUE);
        }
        while ( (retcode = SQLFetch(hstmt1) ) != SQL_NO_DATA )
        {
                // SQLGetData(hstmt1, 1, SQL_C_LONG, &iOutput, 0, &cb1);
                // SQLGetData(hstmt1, 2, SQL_C_CHAR, szOutput, 20, &cb2);
                printf("id=%d name=\"%s\"\n", iOutput, szOutput);
        }
        SQLFreeHandle(SQL_HANDLE_STMT,hstmt1);
}
```

2.5.7 ODBC 中 BLOB（Binary Large Object）字段数据的处理

2.5.7.1 BLOB 字段数据的查询

前面提到过的 API 函数 SQLGetData 可以用于得到普通的字段，也可以得到类似于 TEXT，IMAGE 等大数据量的字段。如果字段的长度大于缓冲区的长度，那么需要多次调用 SQLGetData 来获取字段所有的数据。在获得最后一次数据后 SQLGetData 会返回 SQL_NO_DATA 。SQLGetData 的最后一个参数会返回剩余的数据的长度，记住这个长度是指在调用 SQLGetData 之前剩余数据的长度而不是调用之后。

假设你的测试数据如下：

```
create table test_blob (user_id int identity(1,1),user_memo text not null)
insert into test_blob values
('abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefg')
```

现在我们用一个长度为 150 的缓冲区来分多次得到字段 user_memo 的实际内容。这里我们使用了 SQLServer 中的一个函数 DATALENGTH，这个函数的作用是得到字段的字节长度，用于和你实际得到的数据长度来进行比较。

```
void DoSelectBlob(void)
{
    #define BUFFER_LENGTH           150 //定义数据区的长度

    RETCODE retcode;
    SQLCHAR      bBinaryPtr[BUFFER_LENGTH]; //保存 user_memo TEXT 类型字段的值
    SQLINTEGER   iUserID,iDataLen; //保存 user_id INTEGER 字段的值
    SQLINTEGER   cb1, cb2 ,cb3; //保存数据长度
```

```
        SQLCHAR szSQL[] = "SELECT user_id,DATALENGTH(user_memo),user_memo FROM test_blob";
        //SQL 语句中选出的第二个字段是在 SQLServer 中计算数据长度的函数，用于得到 user_memo 字段的数据长度
        //删除了分配句柄的代码
        retcode = SQLExecDirect(hstmt1,szSQL,SQL_NTS);
        retcode = SQLBindCol(hstmt1, 1, SQL_C_ULONG, &iUserID, 0, &cb1);
        retcode = SQLBindCol(hstmt1, 2, SQL_C_ULONG, &iDataLen, 0, &cb2);
        //通过列绑定来获得第一和第二个字段的值
        while ((retcode = SQLFetch(hstmt1)) != SQL_NO_DATA)
        {//循环得到所有记录
                printf("user_id = %d , data length = %d\n",iUserID,iDataLen);
                while ((retcode = SQLGetData(hstmt1, 3, SQL_C_BINARY, bBinaryPtr, BUFFER_LENGTH, &cb3)) !=
SQL_NO_DATA)
                //多次调用 SQLGetData 一直等到所有数据被取出
                {

                        int iNumBytes = (cb3 > BUFFER_LENGTH)? BUFFER_LENGTH : cb3;
                        //判断当前有多少数据被读出
                        printf("\tget data length = %d , left data length = %d\n",iNumBytes,cb3);
                }
        }
        //删除了释放句柄的代码
    }
```

运行后产生类似下面的输出：

```
user_id = 1 , data length = 336
     get data length = 150 , left data length = 336
     get data length = 150 , left data length = 186
     get data length = 36 , left data length = 36
user_id = 2 , data length = 364
     get data length = 150 , left data length = 364
     get data length = 150 , left data length = 214
     get data length = 64 , left data length = 64
```

2.5.7.2 BLOB 字段数据的更新

前面讲过的 SQLBindParameter 可以在数据插入时提供数据，但对于 BLOB 大数据要一次提供足够的缓冲区会引起内存紧张，所对于大数据来讲会也需要分多次来提供数据。

首先看两个函数说明，SQLParamData 和 SQLPutData 的说明如下：

```
SQLRETURN SQLParamData(
    SQLHSTMT     StatementHandle,
    SQLPOINTER *     ValuePtrPtr);
```

StatementHandle：指定 STMT 句柄，检查在此句柄上执行的 SQL 语句是否需要提供参数。

ValuePtrPtr：得到调用 SQLBindParameter 时指定的参数数据指针。

```
SQLRETURN SQLPutData(
```

SQLHSTMT StatementHandle,
SQLPOINTER DataPtr,
SQLINTEGER StrLen_or_Ind);

StatementHandle：指定 STMT 句柄。

ValuePtrPtr：指明数据区地址。

StrLen_or_Ind：指明本次提供的数据的大小。

在插入和更新大数据时，可以按照下面的顺序来进行：

- 调用 SQLPrepare 准备 SQL 语句，需要插入或修改的列也必须以参数的形式出现在 SQL 语句中。
- 调用 SQLBindParameter 绑定参数，并将参数长度设置为：SQL_LEN_DATA_AT_EXEC(length)。
- 调用 SQLExecute 执行已经准备好的语句，调用应该返回：SQL_NEED_DATA，表明需要提供参数数据。
- 调用 SQLParamData 检查是否需要为 SQL 语句提供参数，调用应该返回：SQL_NEED_DATA。
- 循环提供数据时调用 SQLPutData 来提供数据，此时每次提供的数据量不能超过调用 SQLBindParameter 时设置的数据长度。
- 再次调用 SQLParamData ，表明本次参数的所有数据都已经传递完毕，并且检查是否还有下一个参数需要提供数据，如果有返回上一步提供数据。

这里我们还利用上一节的 test_blob 表来演示 BLOB 数据的插入。

```
void DoPutBlob(void)
{
#define BUFFER_LENGTH                    150 //定义数据区的长度

    RETCODE retcode;
    SQLCHAR     bBinaryPtr[BUFFER_LENGTH]; //提供 user_memo TEXT 类型字段的数据
    SQLPOINTER  pToken;
    SQLINTEGER  cb1,cb2; //保存数据长度
    SQLCHAR szSQL[] = "insert into test_blob(user_memo) values(?)";
    retcode = SQLPrepare(hstmt1,szSQL, SQL_NTS);

    cb1 = SQL_LEN_DATA_AT_EXEC(BUFFER_LENGTH);//设置数据长度
    retcode = SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_LONGVARCHAR,
BUFFER_LENGTH, 0, (SQLPOINTER) bBinaryPtr, 0, &cb1);
    retcode = SQLExecute(hstmt1);
    retcode = SQLParamData(hstmt1, &pToken);
    while (retcode == SQL_NEED_DATA)
    {
        for(int i=0;i<3;i++)
        {//生成测试数据
            memset(bBinaryPtr,'A'+i,BUFFER_LENGTH);
            cb2 = BUFFER_LENGTH - 10*i;//每次可以提供不同量的数据
            retcode = SQLPutData(hstmt1, bBinaryPtr, cb2); //每次提供的数据量不同
        }
        retcode = SQLParamData(hstmt1, &pToken); //本次数据传送完成
    }
```

}

2.5.8 ODBC 对事务的支持

ODBC 中实现对支持事务，在 ODBC 中支持自动提交模式和手工提交模式，当使用自动提交模式时每执行一次 SQL 语句 ODBC 会自动提交本次所进行的修改；当使用手工提交模式时，你必须显示的提交或回滚当前的操作才能够使修改生效。在 ODBC 2.X 中使用 SQLTransact 来开始和结束事务，但是在 3.X 中不再使用，而使用 SQLSetConnectAttr 来设置 ODBC 调用的事务特性，并使用 SQLEndTran 结束事务。

如果要设置为自动提交模式使用下面的方式调用：

SQLSetConnectAttr (hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER) SQL_AUTOCOMMIT_ON, SQL_IS_POINTER);

如果要设置为手工提交模式使用下面的方式调用：

SQLSetConnectAttr (hdbc, SQL_ATTR_AUTOCOMMIT, (SQLPOINTER) SQL_AUTOCOMMIT_OFF, SQL_IS_POINTER);

SQLSetConnectAttr 的函数原型如下：

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

ConnectionHandle：提供 DBC 连接句柄。

Attribute：指定需要设置的属性类型，在这里设置为值 SQL_ATTR_AUTOCOMMIT，需要强制转换位指针类型。

ValuePtr：提供参数值。

StringLength：指定参数的长度，当参数为整数是设置为 SQL_IS_INTEGER，当参数为字符串时设置为字符串长度或者为 SQL_NTS 。

在手工提交模式下，需要使用 SQLEndTran 来提交或回滚当前事务。

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

HandleType：指定句柄的类型，设置为值：SQL_HANDLE_DBC。

Handle：提供 DBC 连接句柄。

CompletionType：设置为 SQL_COMMIT 或者 SQL_ROLLBACK 表明提交或者回滚。

2.5.9 多线程

在 ODBC 3.X 中规定了 ODBC 的驱动程序必须是线程安全的，也就是说应用程序可以在不同的线程中使用同一个 ODBC 句柄进行操作，而对于句柄的同步和串行操作必须由驱动程序自己完成。所以在 ODBC 3.X 中已经不推荐使用异步执行的方式。

2.5.10SQL 语句的异步执行

虽然 ODBC 3.X 中不推荐使用异步执行，但是在某些情况下异步执行还是有用处的，比如在执行过程中可以可以取

消上一次执行的 SQL 语句。当使用 SQLSetStmtAttr 开启了异步执行功能后，在调用 SQLExecDirect 或 SQLExecute 后 ODBC 会返回 SQL_STILL_EXECUTING，然后你需要反复调用，一直等到返回值不为 SQL_STILL_EXECUTING，则表明 SQL 语句已经执行完毕。如果在过程中需要取消现有的 SQL 语句的执行，则调用 SQLCancel 。

```
SQLRETURN SQLCancel( SQLHSTMT     StatementHandle);
```

下面是一段示范代码：

```
SQLSetStmtAttr(hstmt1, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, 0);
while ((rc=SQLExecDirect(hstmt1,"SELECT * FROM test_blob",SQL_NTS))==SQL_STILL_EXECUTING)
{//一直等待执行完成
    // 执行其他的操作
    // 或者调用 SQLCancel( hstmt1); 取消现有的执行
}
```

第 3 章　结束语

今天是 2003/08/30，到 ODBC 的出现到今天已经有 8、9 年的时间了。现在已经有很多数据库访问技术出现了，那么使用 ODBC 是否已经显得很不合时宜？我个人觉得虽然有很多技术出现但是 ODBC 一直是我个人喜欢的方式，这也许就是使用 C 语言的人的共性，使用 API 函数开发的方式是最简单和最容易入手的，和纷繁复杂的类和接口比起来，API 函数更能够让人觉得容易亲近。就我自己来讲我也多次和朋友就 ODBC 和其他的技术如 ADO 进行比较，ADO 和 ODBC 功能类似，但是 ADO 最初出现是主要面向 VB 语言的，所以 ODBC 就方便和实用来讲更适合面向代码性的 C/C++语言。

通过学些 ODBC API 我们可以更多的了解数据库访问的基本机制，这一点来说对很多喜欢追究的朋友来讲尤其有吸引力。而就 VC 开发数据库应用来讲，因为 VC 的强项并不是这方面，所以 MFC 中的 CRecordset 虽然功能简单，却还是被大多数人使用。当然 ADO 也是一个很好的选择，而且现在也有越来越多的人在使用 ADO。至于其他 M$提出过的数据库访问技术 DAO，RDO 等现在都已经较少使用。

此外 M$在 dotNET Framework V1.0 中没有包含对 ODBC 的支持，所以没有在类库中提供 ODBC 类，但是事后发现外界的呼声很大，所以在 dotNet Framework V1.1 中提供了对 ODBC 的支持。见命名空间 System.Data.Odbc（只是对功能进行了封装，ODBC 的本身功能并没有增强）。因为对很多人来说 ADO.NET 并不比 ODBC 优越。

最后强调一点，本文只是包含 ODBC 的很少一部分功能，我认为这些是基本的常用功能，其他例如利用光标修改记录，多记录集查询，批拷贝（ODBC 3.X 中新功能），高性能光标（ODBC 3.X 中新功能），Unicode 支持等我都没有提及（ODBC 的内容很丰富也很全面）。所以本文只能作为一个 ODBC 的基本教材，不能作为一个 ODBC 的参考书。大家有兴趣可以参考其他的一些 ODBC 资料，特别是 MSDN 中的 ODBC Programmer's Reference，那是最权威的 ODBC 资料（个人意见：除了这本书外市面上没有其他一本书能够把 ODBC 讲全面而且清楚）。