# DBMaster

## ODBC プログラマー参照編

SYSCOM

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive
Santa Clara, CA 95050, U.S.A.

**Contact Information:**
CASEMaker US Division
E-mail : info@casemaker.com
Europe Division
E-mail : casemaker.europe@casemaker.com
Asia Division
E-mail : casemaker.asia@casemaker.com(Taiwan)
E-mail : info@casemaker.co.jp(Japan)

www.casemaker.com
www.casemaker.com/support


Document No. 645049-231277/DBM50J-M01312008-ODBC
発行日:2008-01-31



本文には記されていない新しい機能についての説明は、CASEMakerのDBMasterをインストールしてからREADME.TXTを読んでください。

**登録商標**
CASEMaker、CASEMakerのロゴは、CASEMaker社の商標または登録商標です。
DBMasterは、Syscom Computer Engineering社の商標または登録商標です。
Microsoft、MS-DOS、Windows、Windows NTは、Microsoft社の商標または登録商標です。
UNIXは、The Open Groupの商標または登録商標です。
ANSIは、American National Standards Institute, Incの商標または登録商標です。

ここで使用されているその他の製品名は、その所有者の商標または登録商標で、情報として記述しているだけです。SQLは、工業用語であって、いかなる企業、企業集団、組織、組織集団の所有物でもありません。

**注意事項**
本書で記述されるソフトウェアは、ソフトウェアと共に提供される使用許諾書に基づきます。

保証については、ご利用の販売店にお問い合わせ下さい。販売店は、特定用途への本コンピュータ製品の商品性や適合性について、代表または保証しません。販売店は、突然の衝撃、過度の熱、冷気、湿度等の外的な要因による本コンピュータ製品へ生じたいかなる損害に対しても責任を負いません。不正な電圧や不適合なハードウェアやソフトウェアによってもたらされた損失や損害も同様です。

本書の記載情報は、その内容について十分精査していますが、その誤りについて責任を負うものではありません。本書は、事前の通知無く変更することがあります。

# 目次

# 1 はじめに

ODBCプログラマー参照編にようこそ。DBMasterは、強力かつ柔軟なSQLデータベース管理システム (DBMS)です。会話型構造の問い合わせ言語 (SQL)、Microsoft のオープンデータベース結合(ODBC) 互換インタフェース、およびC 言語対応埋め込みSQL (ESQL/C)をサポートします。唯一の公開アーキテクチャであるODBC インタフェースは、多種多様なプログラミングツールを使用して顧客アプリケーションを構築し、既存のODBC-適合アプリケーションを用いてデータベースに問い合わせることを可能にします。

DBMasterは、シングルユーザーの個人データベースから、企業全体に分散するデータベースまでに容易にスケール化することができます。どのようなデータベース構成を選択しても、重要データの安全性は、DBMasterのセキュリティ、整合性、信頼性の先進的機能によって確実に保証されます。広範なクロス・プラットフォームのサポートは、現在あるハードウェアの性能を高め、需要の変化に応じてより強力なハードウェアに拡大し、グレードアップすることを可能にします。

DBMasterは、優れたマルチメディア処理機能を提供し、あらゆるタイプのマルチメディアデータを保存、回収、操作を可能にします。バイナリラージオブジェクト(BLOB) は、DBMaster の先進的セキュリティと損傷リカバリ機能を全面的に利用して、マルチメディアデータの整合性を確実にします。ファイルオブジェクト(FO) は、マルチメディアデータを管理する一方で、既存のアプリケーションで各ファイルを編集できる機能を保持します。

本書は、DBMaster用のフロントエンドのアプリケーションを開発し、アプリケーションでODBC関数を使用する方法を習得しようとしているプログラマーに向けて書かれました。ユーザーの方は、Cプログラミング言語についての知識と、サンプル・プログラムをコンパイル/実行することができるC開発ツールをお持ちであると想定しています。

C プログラミングに関する情報は、本書の対象外です。この範疇で問題に遭遇した場合は、Cプログラミングのマニュアルを参照して下さい。お使いの開発ツールでサンプル・プログラムをコンパイル／実行している際に発生した問題については、開発ツールのマニュアル又は開発ツールのベンダーに問い合せて下さい。

本書では、DBMaster ODBC APIとDBMaster ODBC APIを使った、データベースのためのフロント・エンドのアプリケーションを構築するための概要について説明します。本書では、ODBCプログラミングの紹介するだけを目的としているので、ODBCの全ての概念と実践をカバーしているわけではありません。但し、記述されている全概念は、サンプル・プログラムで何が何故起こるのかを理解するために充分な内容をカバーしています。

各章で、関連関数のグループとそのオプションについて説明し、DBMaster ODBC APIとMicrosoft ODBC 2.1 仕様の間で遭遇するかもしれない違いについて説明します（DBMaster のODBC 3.0 APIについての詳細は、8章の「ODBC 3.0関数」を参照して下さい）。各関数の使用方法について、プラグラムにどのように各関数を当てはめるのかを習得することができます。

掲載する内容をより理解しやすくために、例と図を用いています。サンプル・プログラムはCで記述し、適当なC/C++コンパイラでコンパイルすることができます。

本書は、DBMasterの全ODBC関数に関する情報を紹介しており、Microsoft ODBC 3.0 APIを完全に網羅しているわけではないものの、全ての関数や状態遷移の詳細情報を参照して業務に役立てることができることと思います。推奨マニュアルは、Microsoft Pressの「Microsoft ODBC 3.0 プログラマーズリファレンス」です。

## 1.1 その他のマニュアル

DBMasterには、本マニュアル以外にも多くのユーザーガイドや参照編があります。特定のテーマについての詳細は、以下の書籍を参照して下さい。

- *DBMasterの能力と機能性についての概要は、*「DBMaster入門編」*を参照して下さい。*
- *DBMasterの設計、管理、保守についての詳細は、*「データベース管理者参照編」*をご覧下さい。*
- *DBMasterの管理についての詳細は、*「JServer Managerユーザーガイド」*を参照して下さい。*
- *DBMasterの環境設定についての詳細は、*「JConfiguration Tool参照編」*をご覧下さい。*
- *DBMasterの機能についての詳細は、*「JDBA Toolユーザーガイド」*を参照して下さい。*
- *DBMasterで使用しているdmSQLのインターフェースについての詳細は、*「dmSQLユーザーガイド」*を参照して下さい。*
- *DBMasterで採用しているSQL言語についての詳細は、*「SQL文と関数参照編」*を参照して下さい。*
- *ESQLプログラムについての詳細は、*「ESQL/Cプログラマー参照編」*をご覧下さい。*
- *エラーと警告メッセージについての詳細は、*「エラー・メッセージ参照編」*をご覧下さい。*
- *ネイティブDCI APIについての詳細は、*「DCI ユーザーガイド」*を参照して下さい。*

## 1.2 字体の規則

本書は、標準の字体規則を使用しているので、簡単かつ明確に読むことができます。

| | |
|---|---|
| *斜体* | 斜体は、ユーザー名や表名のような特定の情報を表します。斜体の文字そのものを入力せず、実際に使用する名前をそこに置き換えてください。斜体は、新しく登場した用語や文字を強調する場合にも使用します。 |
| **太字** | 太字は、ファイル名、データベース名、表名、カラム名、関数名やその他同様なケースに使用します。操作の手順においてメニューのコマンドを強調する場合にも、使用します。 |
| キーワード | 文中で使用するSQL言語のキーワードは、すべて英大文字で表現します。 |
| 小さい英大文字 | 小さい英大文字は、キーボードのキーを示します。2つのキー間のプラス記号 (+) は、最初のキーを押したまま次のキーを押すことを示します。キーの間のコンマ(,)は、最初のキーを放してから次のキーを押すことを示します。 |
| 注 | 重要な情報を意味します。 |
| ➲ プロシージャ | 一連の手順や連続的な事項を表します。ほとんどの作業は、この書式で解説されます。ユーザーが行う論理的な処理の順序です。 |
| ➲ 例 | 解説をよりわかりやすくするために与えられる例です。一般的に画面に表示されるテキストと共に表示されます。 |
| コマンドライン | 画面に表示されるテキストを意味します。この書式は、一般的にdmSQLコマンドやdmconfig.iniファイルの内容の入/出力を表示します。 |

# 2 サンプル･アプリケーション

ODBCソフトウェアとDBMasterのODBCドライバをインストールした後、DBMasterを使った簡単なODBCプログラムを作ります。この章では、コンパイルの方法やODBCプログラムをDBMasterのODBCドライバへリンクする方法について説明します。

## 2.1 ライブラリ・モデル

DBMasterの関数ライブラリやODBCの関数ライブラリを使って、ODBCのアプリケーションを作成します。アプリケーション・プログラムでオペレーティング・システム、ファイル・システム、特定ハードウェア・ドライバへの低レベルのシステム呼び出しを必要とする場合、独自の関数ライブラリを使う必要があるかもしれません。*図2-1*と*図2-2* は、これらのライブラリのあるインターフェースのモデルを示しています。

*図2-1* は、ODBCドライバ・マネージャを使用したモデルです。*図 2-2* は、ODBC ドライバ・マネージャを使用せずに直接DBMasterのドライバを使ったモデルです。*ODBC ドライバ・マネージャ*を使用した場合、DBMaster 以外のデータソースに接続することができます。 *ドライバ・マネージャ*を使用しない場合、ご使用のアプリケーションはDBMasterの関数ライブラリに直接リンクし、パフォーマンスは向上します。但し、他のデータソースへ接続する機能が無くなります。

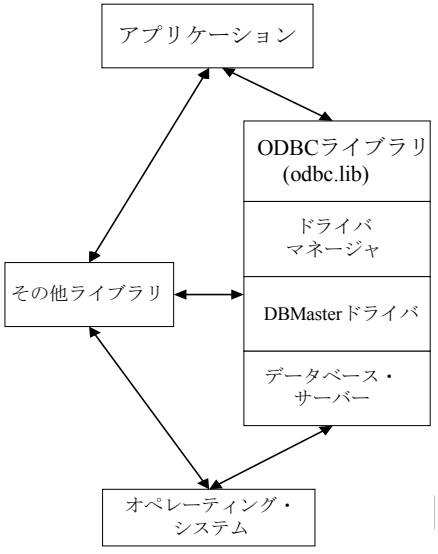


*図 2-1: ODBC ドライバマネージャを使用したODBCのライブラリ・モデル*

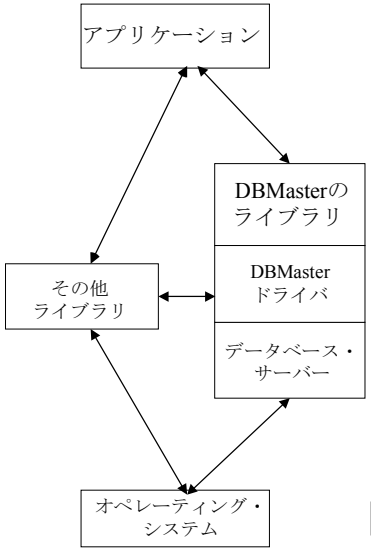


*図 2-2: DBMasterのドライバを直接使ったODBCのライブラリ・モデル*

# 2.2 必要なファイル

DBMasterのODBCドライバを使ってODBCプログラムを構築するためには、メイクファイルにヘッダー・ファイルとリンク・ライブラリを指定する必要があります。以下の解説では、例としてDBMaster version 4.0とC言語を使います。

## ヘッダー・ファイル

ODBCアプリケーションを作成する時、ヘッダー・ファイル***SQL.h***、***SQLext.h***、***SQLopt.h***、***SQLunix.h*** (***SQLunix.h***はUNIXでのみ必要)が必要です。

***SQL.h***と***SQLext.h*** は、ODBCの標準インクルード・ファイルです。更にドライバ独自のオプションやUNIXアプリケーション用に、DBMasterにはそれぞれ***SQLopt.h***と***SQLunix.h***があります。***SQL.h***は、***SQLext.h***にもインクルードされているので、プログラムには***SQLext.h***のみインクルードします。

これらヘッダー・ファイルは、全てのプラットフォームで共通ですが、***SQLunix.h***は、UNIXプラットフォームで運用されているプログラムでのみ使用します。

Microsoft Windowsでは、これらのファイルはDBMasterが初期設定でインストールされるディレクトリ、***c:\DBMaster4.0\include\c*** のディレクトリにあります。或いは、DBMasterがインストールされたドライブ***d:*** の ***d:\install_directory\include*** にあります。

UNIX環境では、これらのファイルは***~DBMaster/4.0/include*** ディレクトリにあります。

## リンク・ライブラリ

DBMasterで使用するODBCアプリケーションを作成するためには、リンク・ライブラリも必要です。使用されるリンク・ライブラリは、アプリケーションが駆動しているプラットフォームによります。

### WINDOWS 95/98/NT/2000

Windows ODBCアプリケーション:

ドライバマネージャを使う場合、一般に***c:\odbcsdk\lib\odbc.lib***にあるODBC SDKのライブラリ***odbc.lib***にリンクします。まず、ドライバマネージャがDBMasterのドライバを正しくロードできるようにするために、***odbc.ini*** にDBMasterを登録します。この場合、メイクファイルに***dmapi40.lib***を定義する必要はありません。ドライバマネージャは、必要なDLLを自動的にロードします。

*注:* *ドライバマネージャを使用しない場合、DBMasterにあるライブラリ* **dmapi40.lib** *にリンクします。このライブラリは通常* **c:\DBMaster4.0lib**. *にあります。*

***c:\windows\system*** ディレクトリに、ODBCプログラムで使用するダイナミック・リンク・ライブラリ***dmapi40.dll***があります。但し、プログラマがリンクに必要とするライブラリは、***odbc.lib*** ( ***ドライバマネージャ使用***)、又は***dmapi40.lib*** ( ***ドライバマネージャ不使用***)です。これらのライブラリをリンクした後、アプリケーションはこのDLLを自動的に呼び出します。

### UNIX

UNIXプラットフォームでは、*クライアント/サーバーのODBCアプリケーションを作成する際に、****libdmapic.a****ファイルにリンクする必要があります。*

## 2.3 サンプルODBCアプリケーション

UNIX環境の以下のサンプル・プログラムについて説明します。

➲ 例、データソースに接続し、SQLGetInfoを使ってDBMSのバージョンを回収:

```
#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 */

retcode = SQLAllocEnv(&henv);
retcode = SQLAllocConnect(&hdbc);

retcode = SQLConnect(hdbc, "TEST", SQL NTS, "SYSADM", SQL NTS, "",
                     SQL NTS);

if (rc != SQL SUCCESS)
    goto EXIT;

rc = SQLGetInfo(hdbc,SQL DBMS VER, &info, STR LEN,&cbInfoValue);

if (rc != SQL SUCCESS)
    goto EXIT;

printf("Current DBMS version is %s\n", info);

EXIT:
    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
    return;
```

## 2.4 コンパイルとリンク

サンプルでは、コンパイラ***acc*** とDBMasterディレクトリの***$dir*** を使用します。

➲ 例 1、UNIXのコマンド・ラインに以下を入力し、サンプル・プログラム (***example.c***)をコンパイルする:

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

ここで、***example.o***をDBMasterのライブラリ***libdmapis.a***にリンクします。

➲ 例 2、exampleという名前の実行ファイルを作成する:

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

## 2.5 サンプル・プログラム

補足のODBCサンプル・プログラムが、DBMasterの***samples***ディレクトリに用意されています。このディレクトリのメイクファイルを使って、サンプルプログラムを構築、実行することができます。まず、***DBMaster/samples*** ディレクトリに変更し、“make ex1”と入力し、サンプル・プログラム***ex1.c***. のための実行ファイル***ex1***を構築します。

Windowsには、Microsoft Visual C++、Borland C++、Watcom C++のような様々な種類のC言語の開発ツールがあります。異なる開発ツールの異なる方法で、ディレクトリをインクルード・ファイルやリンク・ライブラリにセットするために、メイクファイルを編集する必要があるかもしれません。例えば、Visual C++では、新しいプロジェクトを開き、***.mak***と***.def***ファイルを編集します。

***c:/DBMaster/version/samples/c*** ディレクトリにVisual C++用のサンプル・メイクファイルがあります。

# 3 データベース接続

どのようなODBCアプリケーションでも、SQL文を実行/問合せを行う前に、ODBC環境を適切にセットアップし、データソースに接続する必要があります。同様に、プログラムを終了するときには、データベースから切断し、ODBC環境に割り当てたメモリを解放します。この章では、データソースへの接続をセットアップするために必要な関数を紹介します。

本章では、以下について説明します。

- *SQLAllocEnvとSQLAllocConnect関数を使った、環境ハンドルと接続ハンドルの割り当てと、ODBC環境の初期化。*
- *SQLConnect関数を使った決まったデータソースへの接続の新設、データソースが不明な場合のSQLDriverConnect関数の使い方。*
- *SQLGetConnectOptionとSQLSetConnectOption関数を使った、データソースのための接続オプションの取得と設定。*
- *SQLDisconnect関数を使った、データソースからの切断。*
- *アプリケーションを終了する際に、SQLFreeConnectとSQLFreeEnv関数を使った、環境ハンドルと接続ハンドルの解放。*

*注:* *この章で説明している以外の方法で、DBMaster 4.0 (ODBC 3.0)を使って、環境ハンドルと接続ハンドルを割り当て、接続オプションを別々に取得/セットすることができます。詳細については、８章の「ODBC 3.0関数」を参照して下さい。*

## 3.1 環境ハンドル

ODBCアプリケーションでは、他のODBC関数を呼び出す前に、まずSQLAllocEnv関数を呼び出し、ODBC環境をセットアップします。SQLAllocEnv関数を呼び出すと、DBMasterのドライバは環境用のメモリ域を割り当て、アプリケーションに*環境ハンドル*を戻します。

環境ハンドルは、DBMasterドライバがODBC環境に関するグローバル情報を保存するために使用するメモリ域を識別します。これには、有効な接続ハンドルの一覧や現在のアクティブ接続ハンドルのような情報があります。アプリケーションに割り当てる環境ハンドルは1つだけです。

➲ SQLAllocEnv関数の原型:

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

➲ 例 1、環境ハンドルを割り当て、変数HENVを宣言する:

```
HENV henv1;
```

➲ 例 2、SQLAllocEnvを呼び出し、HENV変数のアドレスを渡す:

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

環境ハンドルは有効なハンドルです。後からアプリケーションで使用することができます。アプリケーションがSQLAllocEnvを有効な環境ハンドルへのポインタで呼び出した場合、ドライバは環境ハンドルの以前の内容に上書きします。

## 3.2 接続ハンドル

いかなるODBCデータソースに接続する場合にも、環境ハンドルを割り当てた後、接続ハンドルを割り当てます。接続ハンドルは、ODBCアプリケーションの各接続のためのメモリ域を識別します。データベース名やユーザー名のような情報を含みます。SQLAllocConnect関数は、接続ハンドル用のメモリを割り当てます。

➲ SQLAllocConnect関数の原型:

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

➲ 例 1、接続ハンドルを割り当て、変数HDBCを宣言する:

```
HDBC hdbc1;
```

➲ 例 2、SQLAllocConnectを呼び出し、変数のアドレスを渡す:

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

# 3.3 データソースへの接続

データソースにあるデータにアクセスする前に、データソースに接続します。データソースに接続する次の手順は、データソースを指定することです。

## 指定したデータソースへの接続

SQLConnectを使って、データソースと有効な接続ハンドルの間に接続を新設します。

➲ SQLConnectの原型:

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

アプリケーションで、SQLConnectに以下の情報を与えます。

- *現在、別のデータソースへ接続していない有効な接続ハンドル*
- *データソース名とその名前の長さ*
- *ユーザーIDとその長さ*
- *パスワードとその長さ*

**SQLConnectの使用手順を示したプログラムのフローチャート**

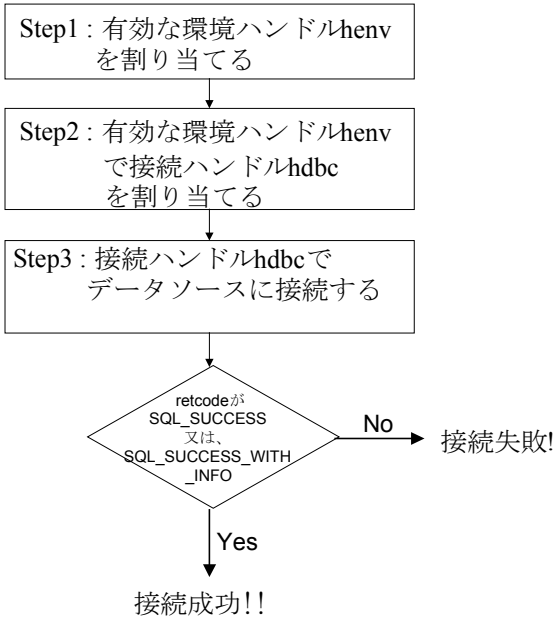


*図 3-1: データソースへの接続のためのプログラム・フロー*

1. SQLAllocEnvを呼び、環境ハンドルhenvを割り当てます。
2. SQLAllocConnectを呼び、環境ハンドルhenvで接続ハンドルhdbcを割り当てます。
3. SQLConnectを呼び、接続ハンドルhdbcでデータソースに接続します。
4. 戻りコードが**SQL_SUCCESS**、又は**SQL_SUCCESS_WITH_INFO**の場合、接続は正しく新設されました。

➲ 例 1、ユーザー*MYNAME* 、パスワード*PASS*で、データソース*TEST_DB*に接続する:

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

**SQL_NTS**は、文字列の最後に**NULL**文字が入っていることを意味し、ドライバはその文字列の長さを計算します。SQLConnectを使用する際に、データソース名の引数は必ず必要ですが、ユーザーIDとパスワードはオプションです。

DBMasterでは、***dmconfig.ini***ファイルの中で初期設定のユーザー名とパスワードを設定し、SQLConnectのユーザー名とパスワードを無視するようにすることができます。ドライバは、***dmconfig.ini***で指定されたユーザー名とパスワードを使用します。この方法では、プログラムの中にユーザー名とパスワードを指定する必要がありません。

➲ 例 2、*dmconfig.ini* に**DB_USRID=MYNAME**と**DB_PASWD=PASS**をセットしてから、TEST_DBに接続する:

```
retcode = SQLConnect(hdbc, "TEST DB", SQL NTS, "", SQL NTS,
                           "", SQL_NTS);
```

アプリケーションが**SQLConnect**を呼び出す時、***ドライバ マネージャ*** は、データソース名**(test_db)**を使って、***ODBC.INI***ファイルの適切なセクションからドライバDLLの名前を読み込みます。その後、ドライバDLLをロードし、そのドライバにユーザー名とパスワード引数を渡します。

## SQLDriverConnectを使ったデータソースへの接続

あらかじめ決められた特定のデータソースに接続する時には、SQLConnectを使用します。但し、使用できるデータソースを全て表示させる場合には、SQLDriverConnectを使用して、ユーザーに接続するデータソースを選択させます。

➲ SQLDriverConnectの原型:

```
RETCODE SQLDriverConnect(
                          HDBC        hdbc,
                          HWND        hwnd,
                          UCHAR  FAR *szConnStrIn,
                          SWORD       cbConnStrIn,
                          UCHAR  FAR *szConnStrOut,
                          SWORD       cbConnStrOutMax,
```

```
                                SWORD  FAR *pcbConnStrOut,
                                UWORD       fDriverCompletion);
```

アプリケーションで、SQLDriverConnectに以下の情報を与えます。

- *現在、別のデータソースへ接続していない接続ハンドル*
- *ダイアログボックス用の親ウィンドウを与えるウィンドウ・ハンドル*
- *入力接続文字列(szConnStrIn)とその長さ。接続文字列には、特有の構文があります(接続文字列の節を参照して下さい)。又、データソースに接続するために必要な特殊な情報が含まれています。入力情報が完全でない場合、SQLDriverConnectはデータベース・ドライバへ情報を送信する前に、ユーザーから更に情報を要求するためにダイアログボックスを使います。*
- *出力接続文字列(szConnStrOut)とその長さ。これはデータベース・ドライバへ送信する接続の最後の部分です。*
- *データソース情報を催促する際に使用する、方針を決定するプロンプト・フラグ(fDriverCompletion)。*

*注:* *SQLDriverConnectを使った接続フローは、図3-1の接続フローと同じです。まず、環境ハンドルと接続ハンドルを割り当てます。それから、SQLDriverConnect関数を呼び出し、データソースに接続します。*

## 入力接続文字列

入力接続文字列は、データソースに接続するために必要な情報を指定します。

➲ キーワードと値の組み合わせの原型:

```
KEYWORD=VALUE;
```

最も頻繁に使用するキーワードは以下のとおりです。

- *DSN — データソース名*
- *UID — ユーザー名*
- *PWD — パスワード*

➲ 例:

```
DSN=TEST DB; UID=myname; PWD=abc;
DSN=TEST DB; UID=myname;
UID=myname;
```

*注:* *入力接続文字列に複数の*DSN、UID、PWD*がある場合、最初の値が採用されます。*

## プロンプト・フラグ

プロンプト・フラグは、*ドライバ マネージャや**DBMaster*のドライバがユーザーから接続情報を受け取るダイアログボックスを使用する必要があるかどうかを示します。

➲ 例、プロンプト・フラグに使用される値:

```
SQL DRIVER PROMPT
SQL DRIVER COMPLETE
SQL DRIVER COMPLETE REQUIRED
SQL_DRIVER_NOPROMPT
```

プロンプト・フラグの値が、**SQL_DRIVER_COMPLETE**、又は**SQL_DRIVER_COMPLETE_REQUIRED**にセットされた場合、*ドライバマネージャは以下の条件に基づくアクションを実行します。*

- *DSNが入力接続文字列で指定されている場合、その入力接続文字列をコピーし、ドライバにその文字列を渡します。*
- *DSNが入力接続文字列で指定されていない場合、ドライバマネージャはデータソース・ダイアログボックスを表示して、データソースをユーザーに選択させます。*
- *ドライバマネージャ は、ダイアログボックスから戻されたデータソース名と入力接続文字列にあるその他のUID*やPWD*値と合わせ、ドライバに渡します。*

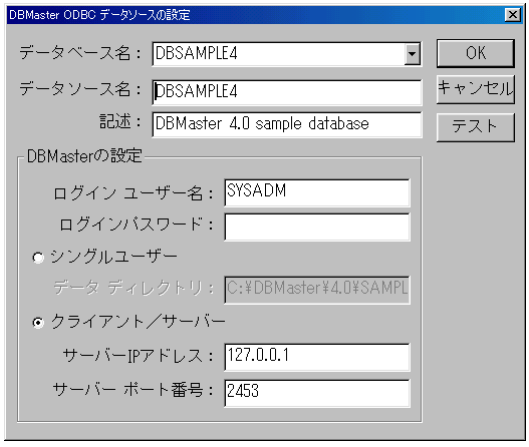


*図 3-2: データソース・ダイアログボックス*

ダイアログボックスから戻されたデータソース名が空白の場合、ドライバマネージャは、キーワードー値の組み合わせを**DSN=Default**に指定します(初期設定データソースのセクションは***ODBC.INI***にあります)。

➲ DBMasterのドライバは、以下の条件に基づくアクションを実行します。:

1. 入力接続文字列に充分な情報(ユーザーIDとパスワード)がある場合、ドライバはデータソースに接続します。
2. 成功した場合、入力接続文字列を出力接続文字列にコピーします。
3. ユーザーID、又はパスワード、或いはその両方が欠けている場合、DBMasterドライバは、ダイアログボックスを表示し、ユーザーに入力接続文字列で欠けている値を入力させます。
4. ユーザーがダイアログボックスを閉じた後、データソースに接続します。
5. 入力接続文字列のDSNの値とダイアログボックスから戻された情報で接続文字列を構築します。それからその接続文字列を出力接続文字列に渡します。

*注:* *プロンプト・フラグが**SQL_DRIVER_PROMPT**にセットされている場合、DBMasterのドライバのふるまいは、**SQL_DRIVER_COMPLETE**や**SQL_DRIVER_COMPLETE_REQUIRED**にセットされている時と同じようになります。ODBC ドライバマネージャは、入力接続文字列にDSNがある無いに関わらず、常にデータソース情報を催促するダイアログボックスを使用します。*

## 複数の接続

1つのアプリケーションから同時に、複数のデータソースに簡単に接続することができます。但し、アプリケーションによっては、同じデータベースに複数回接続する必要があります。例えば、あるタスク(プロセス)に2つのウィンドウがあるかもしれません。そして双方のウィンドウとも同じデータベースへの接続があり、1つのウィンドウは１つの表をスキャンするために使用し、もう一方は別の表を更新するために使用します。DBMasterのSQLConnectコマンドは、1つのプログラムで同じデータソースへ何度も接続することが可能ですが、全ての接続で同じユーザー名を使用し、接続に関係するデータベースの全変更は、同じアクティブ・トランザクションで行わなければなりません。

➲ 例、2つのハンドルと1ユーザー(*user1*, *pass1*)で、データソース*DB1*に2度接続する:

```
retcode = SQLAllocConnect(henv, &hdbc1);
retcode = SQLAllocConnect(henv, &hdbc2);

retcode = SQLConnect(hdbc1, "DB1", SQL NTS, "user1", SQL NTS, "pass1",
                     SQL NTS);
retcode = SQLConnect(hdbc2, "DB1", SQL NTS, "user1", SQL NTS, "pass1",
                     SQL_NTS);
```

但し、2つのユーザー(***user1***, ***pass1***)と(***user2***, ***pass2***)を使って、1つの処理で同じデータソースに接続しようとすると、エラーが発生します。

1回の処理で同じデータソースに何度も接続する必要はありません。この場合の最適な方法は、1つの接続ハンドルで複数のステートメント・ハンドルを持つことです。

*注:　各接続が独立した接続（統合を除く）として扱われるようにする場合は、**dmconfig.ini**で**DB_DIFCO=0**をセットします。DB_DIFCOについての詳細は、「データベース管理者参照編」を参照して下さい。*

# 3.4 接続オプション

データソースへの接続には、そのふるまいを決定する多くの属性が含まれます。例えば、**SQL_AUTOCOMMIT**オプションは、データベースの全操作を自動的にコミットするかどうかを決定します。

## SQLSetConnectOption

各オプションには、システムが定義する初期設定値がありますが、接続用に別の値を指定する場合は、SQLSetConnectOptionを使うことができます。

➲ SQLSetConnectOptionの原型:

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

Hdbcが有効な接続ハンドルの場合、fOptionは接続のためにセットするオプションです。vParamは、fOption用に指定した値です。

DBMasterでは、自動コミット・モードの初期設定値はONです。

➲ 例、接続の自動コミット・モードをOFFにする:

```
retcode = SQLSetConnectOption(hdbc, SQL AUTOCOMMIT,
                                    SQL_AUTOCOMMIT_OFF);
```

オプション値**SQL_AUTOCOMMIT_OFF**は、オプション**SQL_AUTOCOMMIT**の新しい値です。**SQL_AUTOCOMMIT**でOFFにセットすると、トランザクションの全変更をコミットするために、明示的なSQLTransact **(hdbc, COMMIT)**の呼び出しが必要です。

*注:* *接続する前後いつでも、SQLSetConnectOptionを使用して接続オプションをセットすることができます。これらのオプションは、接続ハンドルが存在している間有効です。これらのオプションを一旦セットすると、接続に関係する全ての文に適用されます。オプション* SQL_CONNECT_MODE*は唯一の例外で、接続前に必ずセットする必要があります。*

## SQLGetConnectOption

接続オプションの現在の値が知りたい場合、SQLGetConnectOptionを使ってその値を取得することができます。

➲ SQLGetConnectOptionの原型:

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

hdbcが有効な接続ハンドルの場合、fOptionは回収しようする値を持つオプションです。vParamは、オプションの値を受け取る場所を示します。

➲ 例、変数**commitval**に接続**hdbc**のためのオプション**SQL_AUTOCOMMIT**の値を代入する:

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

# 3.5 ハンドルの解放

アプリケーションを終了する前に、接続とその環境のために割り当てた全リソースを解放します。SQLConnect、SQLAllocConnect、SQLAllocEnvの各関数に対応する、SQLDisconnect、SQLFreeConnect、SQLFreeEnvの関数を使って、それぞれの割り当てたリソースを解放することができます。

## SQLDisconnect

SQLDisconnectは、ある特定の接続ハンドルに関係する接続を閉じ、その接続の全てのステートメント・ハンドルを解放します。(ステートメント・ハンドルは、4章で説明します)。

- SQLDisconnectの原型:

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

## SQLFreeConnect

プログラムが接続を閉じた後、SQLFreeConnectを呼び出して接続ハンドルを解放し、そのハンドルに関係する全メモリを解放します。
SQLFreeConnectを使って、繋がった接続ハンドルを解放しようとする場合、ドライバはエラーを戻します。SQLFreeConnectを呼び出す前に接続を閉じる(SQLDisconnectを呼び出す)必要があります。

- SQLFreeConnectの原型:

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

## SQLFreeEnv

SQLFreeEnvは環境ハンドルを解放し、関係する全てのメモリを解放します。SQLFreeEnvを呼び出す前に、アプリケーションでSQLFreeConnectを呼び出し、環境ハンドルhenvに関係する全ての接続ハンドルhdbcも解放しなければなりません。

- SQLFreeEnvの原型:

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

## プログラムのフローチャート

**6つのODBC関数を使ったアプリケーションのプログラム・フロー**

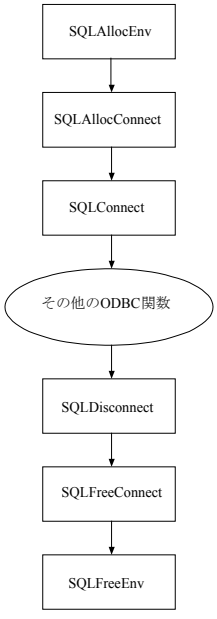


*図 3-3: データソースに接続/切断するためのプログラム・フロー*

# 4 SQL文

この章では、DBMasterがサポートするSQL文を実行するODBC関数の使い方についての詳細を説明します。SQL問合せ言語を紹介につづいて、以下の各項目の実行方法を解説します。

- *関数SQLAllocStmtとSQLFreeStmtを使ったステートメント・ハンドルを割り当てと解放。*
- *関数SQLExecDirectを使って直接SQL文を実行し、実行のためにSQL文を準備し、関数SQLPrepareとSQLExecuteを使って準備した文を実行します。*
- *関数SQLRowCountを使って、UPDATEやINSERT、或いはDELETE文で影響される行数を戻します。*
- *準備時の替わりに実行時に、SQL文をデータ値に渡すためにパラメータを使用します。*
- *関数SQLNumParamsを使ってSQL文にパラメータ数を戻し、関数SQLDescribeParamを使って準備したSQL文に関係するパラメータ・マーカーの内容を戻します。*
- *関数SQLBindParameterを使ってSQL文のパラメータ・マーカーにバッファをバインドし、関数SQLPutDataとSQLParamDataを使って、細かい単位に大きなデータ・アイテムを入力します。*
- *関数SQLGetStmtOptionを使って文オプションの現設定を戻し、関数SQLCancelを使って文処理をキャンセルします。*

以下のダイアグラムは、4-3～4-6節のトピックスと、ODBCを使ったデータベースにアクセスするアプリケーションを記述する時に発生する状態遷移との関係を表しています。

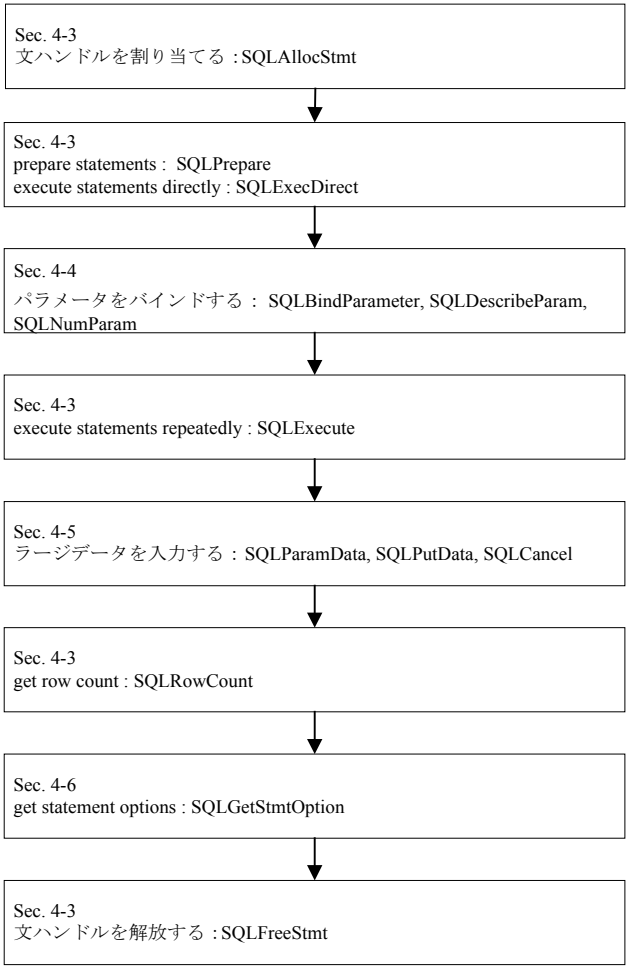


*図 4-1: 本章のトピックスと状態遷移との関係*

# 4.1 SQL言語

*SQL(Structured Query Language)*は、リレーショナル・データベースに保存されているデータを定義、組織、管理、回収するために使用する工業標準問い合せ言語です。CやPascalのような従来の手続き型のプログラミング言語と異なり、データベースをどのように操作するかを明確に定義する必要がありません。英語のようなSQL構文を使って、データベースに要求を入力するだけで、データベースはその要求を処理する最適の方法を決定し、結果を戻します。

SQLに備わっている関数は、その最も重要な関数であるデータ回収だけにとどまりません。SQLは、実際にはDDL *(Data Definition Language)*、DML *(Data Manipulation Language)*、DCL *(Data Control Language)*の3つの部分に分けられます。それぞれが特定の役割を果たし、それらと共にDBMSに備わる全関数を実行することができます。その関数には、以下のようなものがあります。

- ***データ定義*** *— データとデータ間の関係の構造、組織を定義させます。*
- ***データ操作*** *—データベースから既存データを回収、新規データを追加してデータベースを更新、古いデータを削除、以前保存したデータを修正させます。*
- ***データ制御*** *—不正アクセスからデータを保護し、データ破損を防ぐために整合性制約を定義させます。*

この節では、SQLのおまかな概要を説明します。より詳しい内容については、「SQL文と関数参照編」をご覧下さい。

## ODBCでのSQLの役割

ODBCのドライバがSQL文を取得すると、データベース・エンジンにそのSQLリクエストを伝えます。データベース・エンジンはSQL文に応じて、ディスクにデータを構築、保存、回収します。

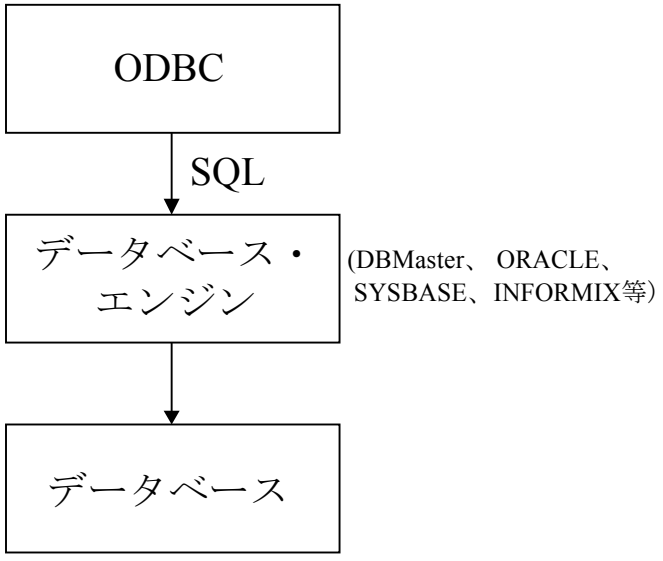


*図 4-2: ODBCを使った時のSQLの役割*

## 基本SQL文

SQL文は、DDL文、DML文、DCL文に分けることができます。DBMasterで使用しているSQL言語とその構文についての完全な内容については、「SQL文と関数参照編」を参照して下さい。

## DDL (データ記述言語)

データベースのスキーマは、DDL *(データ記述言語)*と呼ばれるSQL文で取り扱います。DDLでは、データベース・オブジェクトの定義を定義、削除、修正するために、CREATE文、DROP文、ALTER文を使います。CREATE TABLE文について簡単に説明します。

### CREATE TABLE文

データベースには、多くの表があります。そして各表には情報が保管されています。表は、行(レコード)とカラム(フィールド)で構成されていま

す。CREATE TABLE文を使うと、データベースに新しい表を作成することができます。

➲ 例 1、CREATE TABLE文の基本構文:

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

ANSI/ISO SQL標準は、DBMSがサポートするべき最低限のデータ型を指定しています。ほとんどの商業用のSQL製品は、これらのデータ型をサポート、又は同等の機能を持つ別のデータ型を提供しています。

➲ 例 2、CREATE TABLE文:

```
CREATE TABLE account (
                        no        serial,   /* account number         */
                        lname     name,     /* account last name      */
                        fname     name,     /* account first name     */
                        branch    integer,  /* belong to branch       */
                        balance   money,    /* account balance        */
                        altno     char(12),
                        stamp     image,    /* account's stamp image  */
                        photo     image,    /* account's photo image  */
                        memo      text      /* account's memo          /
                        );
```

## DML (データ操作言語)

データベースのデータの回収と操作を取り扱うのは、DML(データ操作言語)と呼ばれるSQL文の集まりです。基本DML文は、SELECT文、INSERT文、DELETE文、UPDATE文です。

### データベースからデータを回収する (SELECT)

SELECT文を使うと、データベースからデータを回収し、ユーザーに結果セットを返すことができます。

➲ 例 1、SELECT文の基本構文:

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

| データ型 | 説明 |
|---|---|
| **CHAR(サイズ)** | 固定長文字列 |
| **VARCHAR(サイズ)** | 可変長文字列 |
| **BINARY(サイズ)** | バイナリ・データ |
| **OID** | オブジェクトID |
| **FILE** | BLOBオブジェクト(ファイル) |
| **LONG VARCHAR** | BLOBオブジェクト(テキスト) |
| **LONG VARBINARY** | BLOBオブジェクト(バイナリ) |
| **SERIAL [(整数)]** | 自動加算の整数 |
| **SMALLINT** | 小さい整数値 |
| **INTEGER [INT]** | 整数の数 |
| **FLOAT** | 低精度浮動点数 |
| **DOUBLE** | 高精度浮動点数 |
| **DECIMAL [DEC]**<br>**DECIMAL(精度、スケール)** | 10進数の数(初期設定精度とスケールを使用)<br>初期設定精度は17、初期設定スケールは6 |
| **DATE** | 日付 |
| **TIME** | 時間 |
| **TIMESTAMP** | タイムスタンプ |
| その他 | ドメイン |

*表 4-1: DBMasterのデータ型*

基本SELECT文は、SELECT、FROM、WHEREの3つの要素で構成されています。これら各要素の役割は、以下のとおりです。

- ***SELECT*** *— 問合せで回収するカラム、計算するカラムを指定します。*
- ***FROM*** *—SELECTリストのアイテムが存在する表を指定します。.*
- ***WHERE*** *— 行の抽出に合致する検索条件を指定します。*

WHERE句には、以下のような複数の検索条件を指定することができます。

- *比較演算子 (=、>、<、>=、<=、<>、!=)*
- *範囲 (BETWEENとNOT BETWEEN)*
- *リスト (INとNOT IN)*
- *文字列一致 (LIKEとNOT LIKE)*
- *BLOB一致 (MATCHとNOT MATCH)*
- *不明な値 (IS NULLと IS NOT NULL)*
- *論理的な組み合わせ (AND、OR)*
- *否定 (NOT)*

➲ 例 2、残高が$10,000を超える顧客全員を抽出する問合せを実行する:

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

## データベースのデータを修正する

行を追加、削除、更新して、データベースにあるデータを修正することができます。DBMSは、これらの各操作で影響を受ける行数を返します。

## データベースにデータを追加する

INSERT文は、表に新しい行を追加します。

➲ 例 1、INSERT文の基本構文:

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

INSERT文は、INSERT INTOとVALUESの2つの要素で構成されています。これらの各要素の役割は、以下のとおりです。

- *INSERT INTO — 行を挿入したい表を指定します。データを挿入するカラムだけを指定するカラム・リストをオプションで追加することができます。このリストにないカラムには、**NULL**値が挿入されます。*

- *VALUES —挿入するデータ値を指定します。定数、又はパラメータを使って値を挿入します。*

上述のように、値のリストには*定数*か*パラメータ*を用います。定数は、'John'、'Monday'、123、54.823等といったテキスト形式で表現される数字やテキスト、或いは日付値です。定数を使ったINSERT文の例は、以下のとおりです。

➲ 例 2、データベースに、John Smithの新しい口座を追加する:

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

パラメータ・データは、値リストの中では疑問符 (?) で表し、後でその値を代入します。準備時にデータ値がわからない時に、パラメータを使用します。或いは、準備時間を節約したい場合にも利用します。パラメータを使ったINSERT文の例は、以下のとおりです。この例では、データベースに行を挿入しますが、現在その挿入する値はわかりません。

➲ 例 3、実際に挿入される値は、実行前にバインドすることができます:

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

*注:　　文の準備とODBC関数へのパラメータのバインド方法については、本章の4節を参照して下さい。*

## データベースからデータを削除する

DELETE文は、表から行を削除します。

➲ 例 1、DELETE文の基本構文:

```
DELETE FROM table_name WHERE search_condition
```

DELETE文は、DELETE FROMとWHEREの2つの要素で構成されています。

各要素の役割は、以下のとおりです。

- *DELETE FROM — 行を削除する表を指定します。*
- *WHERE — 行を削除する条件を指定します。*

DELETE文で使用するWHERE句には、SELECT文のWHERE句で使用するいくつかの検索条件も含まれます。

➲ 例 2、データベースからJohn Smithの口座を削除する:

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

### データベースのデータを更新する

UPDATE文は、表にある既存の行のデータを変更します。

➲ 例 1、UPDATE文の基本構文:

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

UPDATE文には、UPDATE、SET、WHEREの3つの要素で構成されています。各要素の役割は、以下のとおりです。

- ***UPDATE*** *— 行を更新する表を指定します。*
- ***SET*** *— 変更するカラムと変更内容を定義した式を指定します。*
- ***WHERE*** *— 行を更新するための条件を指定します。*

UPDATE文で使用するWHERE句には、SELECT文のWHERE句で使用するいくつかの検索条件も含まれます。

➲ 例 2、残高が$1000を超える顧客全員の残高に6%の利息を追加する:

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

*注:* *挿入、削除、更新する行数をチェックするには、SQLRowCount関数を用います。*

## 4.2 SQL文を実行する

この節では、簡単なODBCプログラムを書くための手順を紹介します。前節で説明したように、全てのSQL文はプログラムでODBCを介して実行されます。以下の例は、既にデータベースに接続しているものと想定しています。

➲ 例 1、次のSQL文で簡単な表を構築する:

```
CREATE TABLE account (lname name, fname name, branch integer)
```

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

➲ 例 2、対応するODBC文:

```
retcode = SQLAllocStmt(hdbc, &htmt);
retcode = SQLExecDirect(htmt,"CREATE TABLE account (lname name, fname
                              name, branch integer)", SQL NTS);
retcode = SQLExecDirect(htmt,"INSERT INTO account VALUES('Mulder',
                             'Fox', 11240)", SQL_NTS);
```

## SQLAllocStmt

ODBC関数にSQL文を使う場合、ステートメント・ハンドルが必ず必要です。ステートメント・ハンドルは、SQL文の全情報があるシステム制御域(DCCAの一部)の場所へのポインタです。そのため、SQLExecDirectでSQL文を実行する前に、SQLAllocStmtを呼び出して、ステートメント・ハンドルを割り当てる必要があります。

➲ SQLAllocStmtの原型:

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

戻り値が**SQL_SUCCESS**の場合、ドライバから有効なステートメント・ハンドルを割り当てたことを意味し、次のODBC関数SQLExecDirectに進むことができます。

## SQLExecDirect

SQLExecDirectは、SQL文を直接実行するために使用します。ODBCのほとんどのマニュアルでは、これを*直接実行*と呼んでいます。対照的なもう1つの方法として*間接実行*が知られています。間接実行は、のちほど説明します。

➲ SQLExecDirectの原型:

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

第1引数hstmtは、ステートメント・ハンドルです。第2引数は実行するSQL文の文字列です。最後の引数は、SQL文の文字列の長さ、又は**SQL_NTS** (szSqlStrの最後がNull文字の場合)です。

SQLExecDirectは、2段階で実行されます。まず、オブジェクト名と文法をチェックしてSQL文をコンパイル(準備)し、アクセス計画を選び、その文を内部の実行可能フォームに変換します。次に、その実行可能フォームを実行し、実際にデータベースへアクセスします。

SQL文が、“SELECT * FROM account”のような問合せの場合、選択した行の結果セットが生成され、SQLFetchを使って返されたデータ行を結果セットから取得します(*5章を参照*)。*SQL*文が、INSERT文、DELETE文、UPDATE文の場合、SQLRowCountを使って関連する行数を確認することができます。

## SQLRowCount

この関数は、ステートメント・ハンドルで実行したINSERT文、DELETE文、UPDATE文に関連する行の数を返します。

➲ SQLRowCountの原型:

```
RETCODE SQLRowCount(
                    HSTMT      hstmt,
                    SDWORD FAR *pcrow)
```

hstmtがUPDATE文に関係する場合、**pcrow**はUPDATE文の実行後に更新した行数を返します。

➲ 例、SQLRowCountを使う:

```
SDWORD count;
SDWORD retcode;

retcode = SQLAllocStmt(hdbc, &hstmt);
retcode = SQLExecDirect(hstmt,"CREATE TABLE account (lname name,
                        fname name, branch integer, balance money)",
                        SQL NTS);
```

```
/* insert three records into account table
*/

retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Mulder',
                        'Fox', 11240, 10000.00)", SQL NTS);
retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Scully',
                        'Dana', 11330, 20000.00)", SQL NTS);
retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Skinner',
                        'Walter', 11240, 30000.00)", SQL NTS);

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

retcode = SQLExecDirect(hstmt, "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);
```

hstmtがINSERT文、DELETE文、UPDATE文と関係が無い場合、行数は1になります。

## SQLFreeStmt

SQLFreeStmtを使って、ステートメント・ハンドルを閉じたり、削除したりすることができます。

➲ SQLFreeStmtの原型:

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

第1引数(hstmt)は、ステートメント・ハンドルです。第2引数は、どのようにステートメント・ハンドルを解放するかを指定するオプションです。ステートメント・ハンドルを解放する際に一般的に使用されるオプションは、**SQL_CLOSE**と**SQL_DROP**です。

文がselect文でない場合、ステートメント・ハンドルを再使用することができます。

➲ 例 1、ステートメント・ハンドルの再使用:

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

文がselect文の場合、ステートメント・ハンドルを再び使う前にそのステートメント・ハンドルを閉じる必要があります。

➲ 例 2、select文を再使用する前に閉じる:

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

**SQL_CLOSE**を使ってステートメント・ハンドルを閉じると、文を実行しようとする度に新しいステートメント・ハンドルを割り当てる必要がありません。但し、定かではない場合は、**SQL_DROP**オプションを使って、ステートメント・ハンドルを削除し、新しいハンドルを割り当てます。ハンドルを削除すると、そのステートメント・ハンドルに関連する全てのリソースを解放します。削除した後にそのハンドルを再使用することはできません。

## SQLPrepareとSQLExecute

SQLExecDirectの節で説明したように、用意した実行文は、準備と実行の2段階で実行されます。1つの文を繰り返し実行したい場合、準備した実行文を使って、パフォーマンスを向上することができます。

用意した実行文は、ODBC関数の準備(SQLPrepare)と実行(SQLExecute)を使って、文の実行寿命を2つの部分に分けます。この考えは、文を一度だけ実行可能フォームに準備し、それを繰り返し実行するということです。

➲ SQLPrepareの原型:

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

➲ SQLExecuteの原型:

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

hstmtは、ステートメント・ハンドルです。szSqlStrは、実行するSQL文の文字列です。cbSqlStrは、その文字列szSqlStrの長さ、又は**SQL_NTS(**文字列の最後がNull文字の場合)です。用意した実行文は、次の節で説明するパラメータと合わせる場合に最も有効です。

## 4.3 パラメータ

パラメータは、ODBCアプリケーションの実行時にSQL実行文にデータ値を渡すために、SQL文の中で使用します。この概念は、埋め込みSQLで使用するホスト変数に似ています。

SQL文でパラメータを使用するケースには、以下のようなものがあります。

- *パラメータの値が、記述時に不明である場合。*
- *同じSQL文を異なるパラメータ値で何度も実行する必要がある場合。（例えば、データベースの表にデータを挿入する場合、アプリケーションは全ての入力データを文字列で取得する必要があるかもしれません、この場合、パラメータ・マーカーを使用して、全ての入力データを受け入れてから、これらのデータを対応するカラムのデータ型に変換します。つまりドライバは、これらのデータをデータベースに正確に挿入することができます。）*
- *アプリケーションで、パラメータ値を異なるデータ型間で変換する必要がある場合。*
- *ストアド・プロシージャで、出力引数と共に実行する必要がある場合。*

➲ 例、表*account*に5つの行を挿入する:

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

この文では、'?'がパラメータ・マーカーです。パラメータを使用すると、アプリケーションではこの文を一度準備するだけです。そして、別々のパラメータ値で5回その準備文を実行します。

## パラメータ関数

パラメータを扱うODBC関数には、SQLBindParameter、SQLDescribeParam、SQLNumParamsの3つの関数があります。

- **SQLBindParameter-** *パラメータ・マーカーに格納場所をバインドし、格納場所のデータ型、精度、スケールを指定するために使用します。*
- **SQLNumParams** -*SQL文の中のパラメータ数を取得するために使用します。インタラクティブな動的SQLアプリケーションにとりわけ有効です。*
- **SQLDescribeParam-***指定したパラメータの属性（サイズや精度）を評価するために使用します。動的SQLアプリケーションにも使います。*

### SQLBINDPARAMETER

➲ SQLBindParameterの原型:

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

```
                SDWORD FAR *pcbValue)
```

SQLBindParameterには、以下の情報を与えます。

- *hstmt — ステートメント・ハンドル*
- *ipar — i番目のパラメータ*
- *fParamType —パラメータの種類（入力/出力）*
- *fCType — パラメータのホスト言語の型*
- ***f**SqlType — SQLカラムのデータ型*
- *cbColDef —カラムの精度*
- *ibScale — カラムのスケール*
- *rgbValue — ストレージのアドレス*
- *cbValueMax —ストレージ・バッファの長さ*

  *注:　パラメータの種類が出力の際に、このフィールドを使います。戻りデータに整数値のようなCの固定長がある場合、その値は無視されます。パラメータが入力の場合は使用されません。*

- *pcbValue —rgbValueにあるパラメータの長さ*

以下の例は、SQLExecuteを呼び出す前にSQLBindParameterを使ってパラメータに行の値をバインドし、別々の値を持つ各行をデータベースに挿入する方法を表しています。SQLExecuteを3度目に呼び出す際、SQLBindParameterのpcbValueを**SQL_NULL_DATA**にセットすることで、カラム***branch***には**NULL**が挿入されます。

➲ 例、*SQL*BindParameterの使用例:

```
#define LENGTH 18
UCHAR  lname[LENGTH], fname[LENGTH];
UDWORD branch no;
SDWORD retcode, cblname, cbfname, cbbranch;

retcode = SQLPrepare(hstmt,"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(lname, "Mulder");
   cblname = strlen(lname);
   strcpy(fname, "Fox");
   cbfname = strlen(fname);
   branch no = 11240;
   retcode = SQLExecute(hstmt);

   strcpy(lname, "Scully");
   cblname = strlen(lname);
   strcpy(fname, "Dana");
   cbfname = strlen(fname);
   branch no = 11251;
   retcode = SQLExecute(hstmt);

   /* insert a NULL data to branch column for the third customer whose
*/
   /* branch number is unknown
*/
   strcpy(lname, "Angus");
   cblname = strlen(lname);
   strcpy(fname, "MacGyver");
   cbfname = strlen(fname);
```

```
cbbranch = SQL_NULL_DATA;      /* indicate NULL for branch column */
retcode = SQLExecute(hstmt);
```

➲ パラメータ値をセットする:

1. SQLBindParameterを呼び出し、格納場所をパラメータ・マーカーにバインドします。
2. 格納場所にパラメータ値を代入します。

*注:* *1番目のステップは、SQLPrepareを呼び出す前後いずれでも可能ですが、SQLExecuteを呼び出す前に行う必要があります。ドライバがSQL文を実行する際にパラメータ値が必要なので、SQLExecuteの前に2番目のステップを行います。*

SQLBindParameterの引数fParamTypeには、3種類のオプションがあります。

- *SQL_PARAM_INPUT –入力パラメータ。*
- *SQL_PARAM_INPUT_OUTPUT – 入力/出力の両方。*
- *SQL_PARAM_OUTPUT - 出力パラメータ。*

## 入力パラメータを使ったSQLBINDPARAMETERの使用方法

パラメータは、格納場所rgbValueに保管されます。パラメータ値をrgbValueに代入すると、SQLBindParameterの引数fCTypeで指定したCデータ型を使わなければなりません。

SQLBindParameterの引数pcbValueは、単なるパラメータ長を含んだバッファへのポインタが、他の目的に使用することがあります。

SQLExecuteやSQLExecDirectの呼び出しの前に、pcbValueに格納される可能性のある値は、以下のとおりです。

- *文字列やバイナリのCデータにのみ役に立つパラメータの長さ。*
- ***SQL_NTS** – パラメータ値の最後がNull文字であることを表します。*
- ***SQL_NULL_DATA-**前述のコード例のように、パラメータ値が**NULL**であることを表します。*

- ***SQL_DEFAULT_PARAM-*** *そのカラムの初期設定値を使用することを表します。*
- ***SQL_DATA_AT_EXEC****、又は****SQL_LEN_DATA_AT_EXEC-****パラメータのデータが**SQLPutData**に送信されることを表します。これについては、*節4.4 *で説明します。*

## SQLNUMPARAMETER

➲ SQLNumParamsの原型:

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

SQLNumParams関数が呼び出されると、ドライバはバッファpcparに*SQL*文のパラメータ数を代入します。SQL文にパラメータが無い場合、この数は0になります。この関数は、*SQL*文が準備された後（例えば、SQLPrepareが呼び出された後)にのみ呼び出すことができることに注意して下さい。

## SQLDESCRIBEPARAM

➲ SQLDescribeParamの原型:

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

SQLBindParameterのために必要な情報がアプリケーションに全てある場合、直接SQLBindParameterを呼び出すことができます。逆に、アプリケーションには、SQLBindParameterを呼び出す際にパラメータ情報が不足しているかもしれません。例えば、グラフィカルな問合せツールのような動的SQLアプリケーションでは、使用するSQL文は実行時までわかりません。このようなアプリケーションでは、パラメータをバインドできる詳しいパラメータ情報を一度に取得する必要があります。

SQLDescribeParamは、コンパイルしたSQL文に関連する特定のパラメータ・マーカーの詳細を返します。

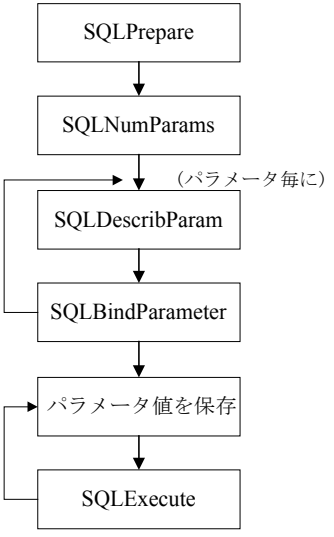


*図 4-3: 入力パラメータを使ったプログラム・フロー*

以下の例は、SQLBindParameter、SQLNumParams、SQLDescribeParamを使った動的SQLアプリケーションの例です。パラメータ・マーカーの番号は、左から右の順で1から始まります。

➲ 例 1、入力パラメータを使った動的SQLアプリケーションの例:

```
#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(cmdp,&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(cmdp, 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(cmdp, 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 */
```

出力パラメータは、出力引数でストアド・プロシージャを実行する時のみ使用します。のちほど、出力引数でストアド・プロシージャを実行するための出力パラメータの使用方法について解説します。ストアド・プロシージャは、ユーザー定義関数です。一旦ストアド・プロシージャを作成すると、データベースのオブジェクトとしてデータベースに実行可能な形式で保存されます。インタラクティブなSQLのウィンドウで、コマンドとしてストアド・プロシージャを実行することができます。また、アプリケーション・プログラム、トリガー、別のストアド・プロシージャでそれを呼び出すことができます。ここでは、ODBCアプリケーション・プログラムでのストアド・プロシージャの実行方法についてのみ説明します。ストアド・プロシージャについての詳細は、「データベース管理者参照編」を参照して下さい。

以下の例は、オプションをSQL_PARAM_OUTPUTに指定して、SQLBindParameter関数を呼び出す方法です。ストアド・プロシージャを指

定した入力都市(Taipei)で実行すると、SQLBindParameterはバッファを用意し、戻り値を保存します。

➲ 例 2、出力パラメータの使用例:

```
SDWORD personNumber;
SDWORD retcode, avlen;

retcode = SQLPrepare(hstmt,"CALL getNumber(“Taipei”,?)",SQL NTS);
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);
```

出力パラメータのある動的SQLプログラムを記述する場合、SQLNumParams、SQLDescribeParam、SQLProcedures、SQLProcedureColumnsを使うことができます。SQLProceduresは、データソースに保存されているプロシージャ名の一覧を取得します。SQLProcedureColumnsは、プロシージャのパラメータに関する情報を取得します。SQLProceduresとSQLProcedureColumnsについての詳細は、本書の6.4節を参照して下さい。

## SQLExecDirectでパラメータを使う

前述のように、アプリケーションで複数回SQL文を実行したい場合、まずSQLPrepareを呼び出し、それから実行の度に同じSQL文をコンパイルする替わりに、SQLExecuteを実行の度に呼び出します。

パラメータは、SQLExecDirectを使って一度だけ実行するSQL文でも使用することができます。但し、SQLExecDirectを呼び出す前にパラメータをバインドし、パラメータ値をセットしなければなりません。つまり、

SQLPrepareとSQLExecuteでパラメータを使う場合の利点が全く無くなります。

一般的に、入力するデータがBLOBのような特殊なタイプでない限り、パラメータにSQLExecDirectを使う必要はありません。BLOBの場合、**SQL_DATA_AT_EXEC**、又は**SQL_LEN_DATA_AT_EXEC**オプションを使って、実行時にパラメータをバインドします。但し、データ値はその文が実行されるまでセットしません。(ラージ・データ要素の入力についての詳細は、*4.4* 節を参照して下さい。)

➲ 例、SQLExecDirectでのパラメータの使用例:

```
#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(lname, "Bill");
cblname = strlen(lname);
strcpy(fname, "Skinner");
cblname = strlen(fname);
branch no = 11243;

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

```
err_exit(hstmt, retcode);
```

## バインドしたパラメータを取り消す

SQLBindParameterを呼び出して格納場所がバインドした後、それを例のように繰り返し利用することができます。例では、3つの格納場所がINSERT文で3つのパラメータ・マーカーにバインドされ、明確にアンバインドされるまで、バインドされたままになります。

**SQL_RESET_PARAMS**オプション、又は**SQL_DROP**オプションでSQLFreeStmtを呼び出した時、格納場所はアンバインドされます。同一のステートメント・ハンドルに属する3つの格納場所は、SQLFreeStmtが呼び出される時、全てアンバインドされることに注意して下さい。オプション**SQL_RESET_PARAMS**を使う場合、ステートメント・ハンドルをリセットし、別の格納場所をバインドします。

➲ 例、バインドしたパラメータを取り消す例:

```
#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);
```

**SQLFreeStmt**にオプション**SQL_DROP**を使う場合、ステートメント・ハンドルは解放され無効になります。

# 4.4 ラージ・データを入力する

DBMasterには、データベースにBLOBを入力する方法が2つあります。1つは、小さい固定サイズのバッファを使い、BLOBデータの1部分を読み込み、データベースにそれを入力します。この手順を数回繰り返すことで、1つの大きなバッファを使う事無く、BLOB全体を入力できます。もう1つは、外部ファイルにBLOBを保存することができるファイル・オブジェクトのデータ型を使用します。

## ラージ・データの挿入方法

long varcharやlong varbinaryカラムに大量のデータを入力する時、SQLPutData関数とSQLParamData関数を使って、小さなセグメントにデータを入力します。これにより、一度に全てのデータを入力する場合のような、全データを保存するための大きなバッファは必要ありません。

➲ SQLParamDataの原型:

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

➲ SQLPutDataの原型:

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

SQLParamDataは、パラメータにデータが必要かどうかをチェックするために使用します。その後、SQLPutDataを使ってデータを入力します。SQL文の全てのパラメータにデータが入力されるまで、この手順が続きます。

➲ データベースにラージ・データ・オブジェクトを入力する:

1. パラメータをバインドする —SQLBindParameter関数の引数pcbValueをSQL_DATA_AT_EXEC、又はSQL_LEN_DATA_AT_EXECのいずれかにセットします。これは、ユーザーが実行時にSQLPutDataを使って、このパラメータに値を与えることをODBCドライバに伝えます。

2. SQLコマンドを実行する —SQLExecDirect、又はSQLExecuteで、SQL文を実行します。実行時にデータを要求するパラメータがある場合、SQL_NEED_DATAが返ります。

3. 実行時にデータを要求する最初のパラメータを見つける —SQLParamDataを呼び出して、データ回収を始める実行時にデータを要求する最初のパラメータを表示します。

4. **SQLPutData** を呼ぶ— バッファにデータの次のセグメントを用意し、*SQL*PutDataを呼び出して現在データを待機しているパラメータ用にそれをデータベースに送信します。このパラメータへのデータを全て送信するまで、この手順を繰り返します。

5. **SQLParamData**を呼ぶ— 戻りコードがSQL_NEED_DATAの場合、実行時にデータを要求する次のパラメータは、データを回収する用意ができていますので、Step 4に戻ります。戻りコードがSQL_SUCCESS、又はSQL_SUCCESS_WITH_INFOの場合、それから実行時にデータを要求する全てのパラメータの全データが送信され、SQL文の実行が完了します。

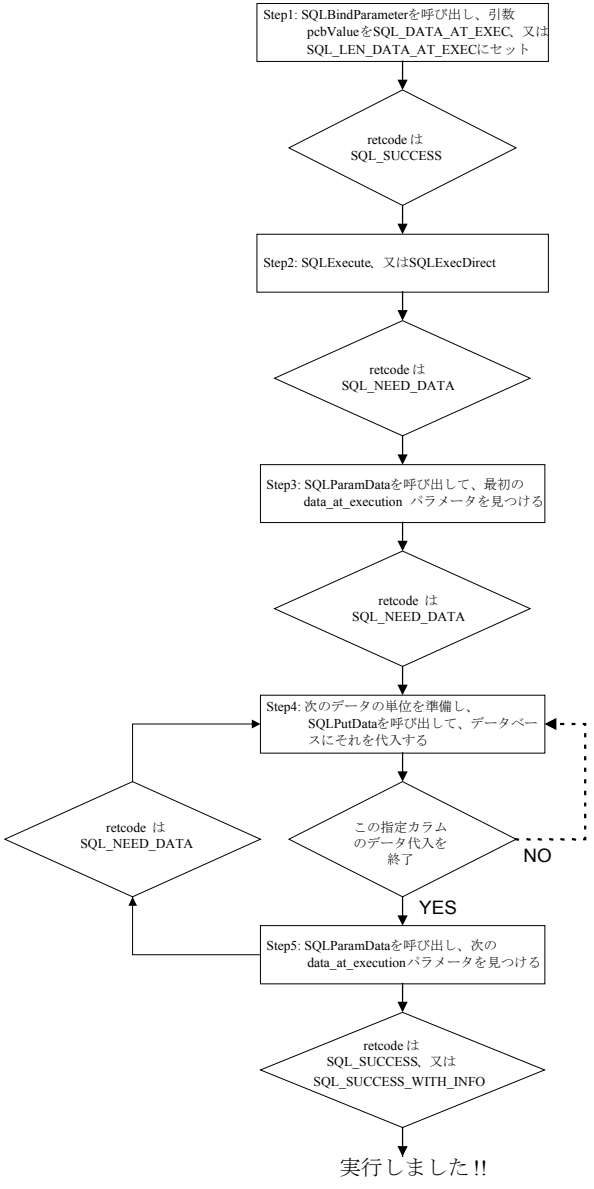


*図 4-4: ラージ・データを入力するためのプロセス・フロー*

次の例では、表***account***にレコードを挿入します。**InitUserData()**は、***account*** 表に入力される顧客の***photograph***、***signature*** 、***memo***フィールドを含む顧客のデータ・ファイルを開きます。**GetUserData()**は、ユーザーのデータ・ファイルから次の**MAX_DATA_SIZE**バイトのデータを、全データを読み込むまでデータ・バッファInputDataに回収します。**SQLPutData**は、データをバッファからデータベースに送信します。

➲ 例、SQLParamDataとSQLPutDataでデータを挿入する:

```
#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 LONGVARBIANRY, 0, 0, NULL, 0,
                               &cbPhoto);
    retcode = SQLBindParameter(hstmt, 2, SQL PARAM INPUT, SQL C BINARY,
                               SQL LONGVARBIANRY, 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");
        }
   }
```

## SQLPutDataの実行を取り消す

データベースにデータを入力中にエラーが起こった時、或いはデータ入力を中止する場合、SQLCancel関数を使ってその処理を取り消すことができます。

➲ SQLCancelの原型:

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

SQLCancelを呼び出して、文全体を中止することができます。現在実行している文を取り消した後、再びSQLExecuteやSQLExecDirectを呼び出すことができます。

## FOにラージ・データを配置する

*FO（ファイル・オブジェクト）*は、DBMasterがサポートする強力なラージ・オブジェクトのデータ型です。FOは、**LONG VARCHAR**や**LONG VARBINARY**データ型に似ていますが、使用しているホスト・ファイル・システムに外部ファイルとして保存します。**FILE**データ型にカラムを定義すると、FOカラムを作成することになります。

➲ 例 1、*photograph*カラムをFOカラムに定義する

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

FOはBLOBデータとして扱われるので、BLOBの挿入方法を使ってFOカラムにデータを挿入することができます。引数fSQLTypeを**SQL_LONGVARCHAR**、又は**SQL_LONGVARBINARY**にセットすると、DBMasterは各FOカラムに新しいファイルを作成し、データを入力バッファから（fCTypeが**SQL_C_CHAR**の時）、或いはクライアント側のファイルから（fCTypeが**SQL_C_FILE**の時）コピーします。

他のファイルを作成する替わりに、CD-ROMにあるファイルのような、サーバー側にある既存のファイルにFOカラムをリンクしようと考えるかもしれません。サーバー側のファイルにリンクする場合、引数fCTypeを**SQL_C_CHAR**にセットし、fSqlTypeを**SQL_FILE**にセットします。この方法を用いると、SQLExecuteを呼び出す前にバッファにファイル名をコピーします。

| FSQLTYPE | FOカラム | FCTYPE | データソース |
|---|---|---|---|
| **SQL_LONGVARCHAR**、又は**SQL_LONGVARBINARY** (BLOBと同じ) | サーバーに新しいファイルを作成 | **SQL_C_FILE** | クライアントのファイルからデータをコピー |
| | | **SQL_C_CHAR**<br>**SQL_C_BINARY** | 入力バッファからデータをコピー |
| **SQL_FILE** | サーバーの既存ファイルにリンク | SQL_CHAR | 入力バッファからファイル名を取得 |

次の例では、顧客***Mary***の氏名とファイル・オブジェクトとして保存する彼女の写真を新しいレコードとして挿入します。そのファイルは既にサーバーに存在するので、このファイルをデータベースにリンクするだけです。

➲ 例 2、ファイル・オブジェクトにラージ・データを代入する:

```
UCHAR  pPhotoFlName[80];
SDWORD retcode;

retcode = SQLPrepare(hstmt, "INSERT INTO student VALUES('Mary', ?)",
                     SQL NTS);
retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,
                           SQL FILE, 80, 0, pPhotoFlName, 0, SQL NTS);

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

retcode = SQLExecute(hstmt);
```

FOは外部ファイルに保存されるので、ファイルの編集に使用するアプリケーションは、ファイル・ディレクトリでそのまま使用することができます。

## 4.5 Get/Setオプション

ステートメント・ハンドルにある文オプションの現在の設定は、SQLGetStmtOption関数で取得することができます。

➲ SQLGetStmtOptionの原型:

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

➲ SQLSetStmtOptionの原型:

```
RETCODE SQLSetStmtOption{
                           HSTMT   hstmt,
                           WORD    fOption,
```

```
                                UDWORD  vParm);
```

hstmtが有効なステートメント・ハンドルである時、fOptionは回収するオプションで、pvParamはfOptionに関連する値です。fOptionの値によって、32-bitの整数値やNull文字で終わる文字列へのポインタがpvParamに返されます。

DBMasterには、これら2つの関数に使用することができるオプションがいくつかあります。

SQLGetStmtOption関数の拡張文オプション

| オプション | 解説 |
|---|---|
| **SQL_GET_INCREMENT_VALUE** | **SERIAL**カラムの値を取得 |
| **SQL_GET_CURRENT_OID** | 挿入/フェッチした直前のタプルのOIDを取得 |
| **SQL_GET_BACKUP_ID** | バックアップIDを取得 |
| **SQL_DUMP_PLAN** | ダンプ計画オプションを取得 |
| **SQL_PLAN** | 問合せ実行計画を取得 |
| **SQL_PLAN_LEN** | 計画長を取得 |

SQLSetStmtOption関数の拡張文オプション:

| オプション | 解説 | 許容値 |
|---|---|---|
| **SQL_DUMP_PLAN** | ダンプ計画オプションをセット | SQL_DUMP_PLAN_ON<br>SQL_DUMP_PLAN_OFF |

次の例の表***account*** には、**SERIAL**データ型のカラムがあります。**SERIAL**カラムの値は自動的に加算され、ユーザーはそれを指定する必要がありません。但し、レコード挿入の後に、挿入されたばかりの**SERIAL**カラムの値を知る必要があるかもしれません。この値は、SQLGetStmtOptionを呼び

出して、引数fOptionを**SQL_GET_INCREMENT_VALUE**にして取得することができます。

➲ 例 1、SQLGetStmtOptionを使ったSERIALカラムの値を取得する:

```
/* insert a record into table account where the value of each field
*/
/* is its default value
*/


SDWORD val;
SDWORD retcode;


retcode = SQLExecDirect(hstmt, "INSERT INTO ACCOUNT VALUES ()",
                        SQL NTS);


/* get the serial number that was just inserted
*/
retcode = SQLGetStmtOption(hstmt, SQL_GET_INCREMENT_VALUE, &val);
```

この例では、変数valは直前のINSERT文で表accountに挿入された**SERIAL**カラムの値です。**SERIAL**データ型の定義と用途については、「SQL文と関数参照編」を参照して下さい。

SQLGetStmtOptionのもう1つの特殊な用途は、挿入/フェッチされた直前のタプルのOIDを取得することです。上記の例に続けて、SQLGetStmtOption関数で引数fOptionに**SQL_GET_CURRENT_OID**を与え、データベースに挿入されたばかりのレコードのOIDを取得することができます。

➲ 例2、SQLGetStmtOptionを使って現在のオブジェクトのOIDを取得する:

```
UCHAR  oid[8];
SDWORD retcode;


/* insert a record into account table */
retcode = SQLExecDirect(hstmt, "INSERT INTO ACCOUNT VALUES ()",
                        SQL NTS);


/* get the OID of the record just inserted */
```

```
retcode = SQLGetStmtOption(hstmt, SQL_GET_CURRENT_OID, &oid);
```

OIDは、DBMaster内のオブジェクトに付けられている一意のIDです。データベースでオブジェクトを一意に識別するためにOIDを使います。

➲ 例3、問合せのWHERE句にOIDを使う:

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

*注:　　OIDのデータ型についての詳細は、「SQL文と関数参照編」、又は「データベース管理者参照編」をご覧下さい。*

拡張文オプションの**SQL_DUMP_PLAN**、**SQL_PLAN_LEN**、**SQL_PLAN**は、準備したSQL文のために、DBMaster問合せオプティマイザが生成した問合せ計画を取得するために使用します。

➲ 準備したSQL文の計画を取得する:

1. SQLSetStmtOptionを呼び出して、SQL_DUMP_PLANオプションをONにします。
2. 関数SQLGetStmtOptionを呼び出して、引数fOptionをSQL_PLAN_LENにし、計画文字列の長さを取得します。
3. 計画文字列に応じたバッファを割り当ててから、SQLGetStmtOptionを呼び出し、引数fOptionをSQL_PLANにして計画文字列を取得します。

➲ 例 4、SQLSetStmtOptionとSQLGetStmtOptionを使ったSQL文の問合せ計画の取得例:

```
SDWORD planlen;
UCHAR  *planstr;
SDWORD retcode;


/* turn on the dump plan option
*/
retcode = SQLSetStmtOption(hstmt, SQL DUMP PLAN, SQL DUMP PLAN ON);


/* prepare an SQL JOIN statement
*/
retcode = SQLPrepare(hstmt, "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 = malloc(planlen);


/* get the plan string
*/

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

# 5 結果の回収

問合せを実行してデータを回収することは、データベースの最も大切な機能の1つです。

本章では、以下の実行方法について説明します。

- *関数SQLBindColとSQLFetchを使って、格納場所にカラムをバインドし、行ごとにデータを回収。*
- *関数SQLNumResultCols、SQLDescribeCol、SQLColAttributesを使って、結果セットに、データ型や長さのようなカラム情報を取得。*
- *カーソルを使って、問合せから取得した結果セットに配置したDELETE、又は配置したUPDATEを実行。*
- *関数SQLGetDataを使って1部分ごとに、ラージ・データ・オブジェクト(**LONG VARCHAR**や**LONG VARBINARY**)とファイル・オブジェクトを回収。1部分づつラージ・オブジェクトやファイル・オブジェクトを回収すると、一度にデータを回収する場合に比べ比較的小さいバッファですみます。*
- *行セットを使って、関数SQLExtendedFetchとSQLSetPosの問合せで取得した結果セットを前後に閲覧。*

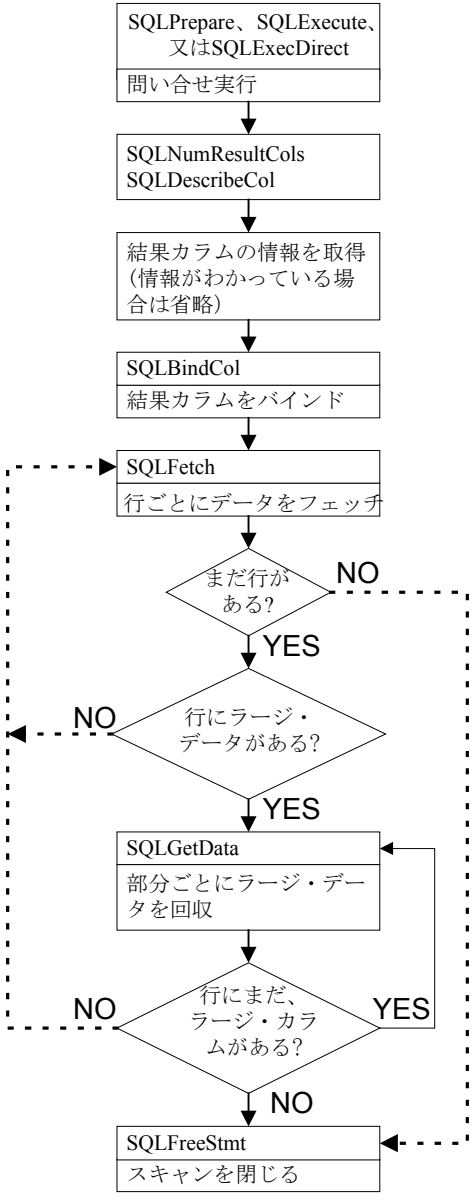


*図 5-1: データベースからデータを回収するためのプログラム・フロー*

# 5.1 ODBCを使った問合せ

アプリケーションでデータベースからデータを回収する最も一般的な方法は、SQLのSELECT文を使った問合せを実行することです。この節では、ODBC関数を使って問合せを実行し、行ごとに結果データをフェッチする方法について解説します。

## 格納場所のバインドとデータのフェッチ

表***account***から支店11240の顧客の姓と名前、支店情報を取得すると想定します。

➲ 例、問合せ:

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

この問合せを準備、実行した後、行ごとにデータをフェッチする用意ができます。この問合せのプロジェクションにある結果カラムの全情報が、わかっている場合(カラムのデータ型、精度、スケール等)、SQLBindColとSQLFetchを使って、結果をフェッチします。

- **SQLBindCol-** *データのカラムと格納場所を関連付けるために使用します。SELECT文のSQLBindColの役割は、INSERT文のSQLBindParameterの役割に似ています。*
- **SQLFetch-** *結果セットからデータの行をフェッチするために使用します。ドライバは、SQLBindColで定義した格納場所にバインドした全カラムのデータを返します。*

➲ SQLBindColの原型:

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

アプリケーションは、結果カラムと格納場所を関連付けるために、以下の情報をSQLBindColに与えます。

- *fCType — 結果セットのCデータ型*
- *rgbValue — データのための出力バッファのアドレス。アプリケーションでこのバッファを割り当て、バッファが指定したデータ型で回収したデータを格納するために充分な大きさであることを明確にします。*
- *cbValueMax — 出力バッファの最大長。この値は、戻りデータに整数値のようなCの固定長がある場合は無視されます。*
- *pcbValue — 利用できるデータのバイト数を戻すために使用されるストレージ・バッファのアドレス。フェッチしたデータが**NULL**の場合、ドライバはこの引数に**SQL_NULL_DATA**を保存します。*

プロジェクションにある各カラムをバインドした後、*SQL*Fetch()を呼び出しデータ行をフェッチします。

➲ SQLFetchの原型:

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

以下の例は、SQLBindColとSQLFetchで先に説明した問合せ実行します。

➲ 例、表***account***から支店11240の全ての顧客情報をフェッチする:

```
#define  LENGTH  18

UCHAR       lname[LENGTH], fname[LENGTH];
UDWORD      branch no, TRUE = 1;
SDWORD      retcode, cblname, cbfname, cbbranch;

retcode = SQLExecDirect(hstmt, "SELECT lname, fname, branch FROM
                                        account WHERE branch = 11240",
                                        SQL NTS);
if (retcode == SQL SUCESS)
   {
    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, &cbbrach);
   }
/* 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;
   }
```

## 結果カラムの特性

*節4.4*で説明したように、アプリケーションによっては、どのようなデータ型が挿入されるか前もってわからないことがあります。同様に、あらかじめフェッチする結果データがわからない可能性があります。そのような場合、SQLNumResultColsとSQLDescribeCol関数が詳しい情報を提供するの

に役立ちます。SQLNumResultColsは、結果セットにある結果カラムの数を取得するために使用します。SQLDescribeColは、結果カラムの特性を詳述するために使用します。特性には、その名前、SQLの種類、精度、スケール、カラムがNull値可かどうかといった情報が含まれます。

- SQLNumResultColsの原型:

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

- SQLDescribeColの原型:

```
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)
```

- このケースのプログラム・フローは以下のとおりです。

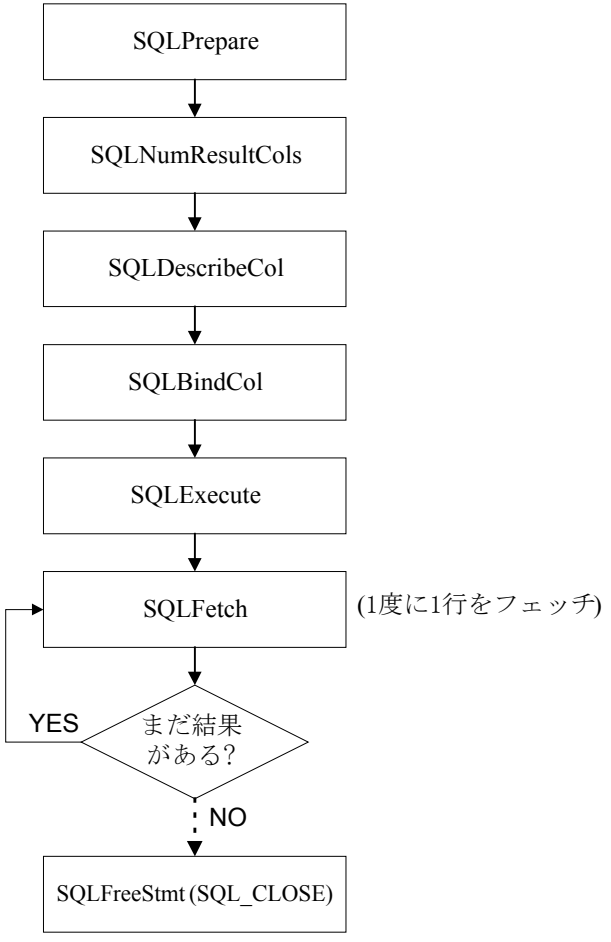


*図 5-2:* SQLNumResultColsとSQLDescribeColを使って結果セットを取得するためのプログラム・フロー

問合せを準備した後、SQLNumResultColsを呼び出して、その問合せにいくつの結果カラムがあるかを見つけます。次にSQLBindColとSQLDescribeColを交互に呼び出して、各カラムが必要とするメモリがどのぐらいであるかという情報を取得します。最後に、SQLFetchを呼び出して、1度に1行の結果データをフェッチします。

以下に示した例では、SQLNumResultCols、SQLDescribeCol、SQLBindCol、SQLFetchを使って、問合せ実行後に結果セットをフェッチします。

➲ 例、問合せを実行後、結果セットをフェッチする:

```
#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];

BEGIN:                          /* begin label                  */

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

/* prepare the input query
*/
retcode = SQLPrepare(hstmt,queryStr, 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, &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], coltype[i],
                             collen[i], colscale[i], colbuf[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
*/
```

## 結果カラムの詳細

SQLDescribeColを呼び出して、カラムの特性について一定の情報を取得することができても、アプリケーションに与える必要のある付加的なカラム情報はまだあります。ODBCには、この目的を達成する関数SQLColAttributesがあります。

### SQLCOLATTRIBUTES

SQLColAttributesは、カラムの記述子情報を返すために使用します。これは、指定した種類の記述子情報です。

➲ SQLColAttributesの原型:

```
RETCODE SQLColAttributes(
                           HSTMT       hstmt,
                           UWORD       icol,
                           UWORD       fDescType,
```

```
                        PTR          rgbDesc,
                        SWORD        cbDescMax,
                        SWORD  FAR *pcbDesc,
                        SDWORD FAR *pfDesc)
```

ODBCで定義できる記述子の種類は、以下のとおりです。

- **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:　　各オプションの意味についての詳細は、「Microsoft ODBC プログラマーズリファレンス」を参照して下さい。*

例えば、fDescTypeの値が**SQL_COLUMN_TYPE**の場合、SQLColAttributesは、指定したカラムのSQLのデータ型を返します。この情報は、SQLDescribeColを使っても取得することができますが、SQLColAttributesはSQLDescribeColでは取得できないその他の情報も提供します。

SQLColAttributesとSQLDescribeColの大きな違いは以下のとおりです。

- *SQLDescribeColは、一度に1つのカラムに指定した複数の値を提供しますが、SQLColAttributesは1つの記述子の値のみ取得します。*
- *SQLColAttributesは、特定のより詳細のカラム情報を提供します。ドライバで更にドライバ特有の記述子を追加したり、ODBCで将来のバージョンに新しい記述子を定義したり場合、拡張することも可能です。*

例えば、カラムが大文字・小文字を識別するかどうかを知る必要があるアプリケーションでは、SQLColAttributes関数を使って、記述子のオプションを**SQL_COLUMN_CASE_SENSITIVE**にし、見つけることができます。

➲ 例、SQLColAttributesを使った詳細なカラム情報の取得例:

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

retcode = SQLPrepare(hstmt, "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");
```

## バインドしたカラムを取り消す

SQLBindColを呼び出して格納場所をカラムにバインドした後、それを繰り返し再利用することができます。先述の例では、3つの格納場所をSELECT文の3つのカラムにバインドしました。

➲ 例、**SQL_UNBIND**オプションにしたSQLFreeStmtを呼び出し、ステートメント・ハンドルに関係するバインドした全カラムをアンバインドする:

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

ここで、hstmtでバインドした全ての格納場所が取り消されます。そして、アプリケーションはステートメント・ハンドルを再利用することができ、別のストレージ域にバインドすることができます。アプリケーションで1つのバインドしたカラムをアンバインドしようとする場合、替わりにSQLBindColを呼び出し、引数rgbValueにNullポインタを渡します。

アプリケーションでステートメント・ハンドルの再利用をする必要がない場合、**SQL_DROP**オプションのSQLFreeStmtを呼びます。この場合、全ての格納場所が取り消され、既存のカーソル、中断した結果、ステートメント・ハンドルが使用しているリソースも同様に解放されます。

# 5.2 カーソル

カーソルは、行の条件処理において行ごとに結果セットを操作することを可能にします。アプリケーションは、与えられた結果セットの個々の行で複数の操作を実行することができます。問合せを実行することで結果セットにカーソルを開くことができます。

## カーソルを使うタイミング

カーソルは、プログラムが結果セットの特定の行で更新/削除の操作を実行する必要がある時に使用します。例えば、プログラムは、問合せ結果から行を回収し、それらを画面に表示し、データを更新/削除するユーザーの要求に応えることができます。

カーソルを使ってデータを更新しようと考える場合、結果セットを生成するために使用するSELECT文にはっきりとFOR UPDATE、又はFOR UPDATE OF column_listと指定する必要があります(例、SELECT * FROM account FOR UPDATE)。SELECT文にFOR UPDATEが宣言されていない場合、カーソルの初期設定の読み込み専用になり、カーソル更新や削除はできません。

➲ 例 1、カーソルを使ったUPDATE文:

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

➲ 例 2、カーソルを使ったDELETE文:

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

## カーソル名を取得する

ODBCドライバは、SQLAllocStmtを呼び出しステートメント・ハンドルを割り当てる時に、**SQL_CUR**で始まる名前を自動的に生成します。SQLGetCursorNameを使って、ステートメント・ハンドルが割り当てたカーソルのフルネームを取得することができます。

➲ SQLGetCursorNameの原型:

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

*注:　　カーソル名の最大長は、18文字です。*

## カーソルを使う

以下の例は、SELECT文とUPDATE文に2種類のhstmtがあるプログラムでどのようにSQLGetCursorNameを使用するかを示しています。

➲ 例、John Smithの支店番号を更新するためにUPDATE文でカーソルを使う:

```
#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;
```

```
/* 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, "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 INTEGER, &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(szFname, "John") != 0 && strcmp(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(updsql, "UPDATE account SET branch = 2101 WHERE
                     CURRENT OF %s", cursorName);
    retcode = SQLExecDirect(hstmtUpdate, updsql, SQL_NTS);
   }
```

## カーソル名を設定する

SQLSetCursorNameを使って、アクティブのステートメント・ハンドルのカーソル名をセットすることができます。SQLSetCursorNameでSELECT文を実行する前にカーソル名を変更します。

➲ SQLSetCursorNameの原型:

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

# 5.3 ラージ・データをフェッチする

## SQLGetData

**LONG VARCHAR**、又は**LONG VARBINARY**のカラムからラージ・データを回収する必要がある時、SQLGetData関数を使ってセグメント毎にデータを回収することができます。この方法では、カラム全体を回収するための大きなバッファを準備する必要がありません。

➲ SQLGetDataの原型:

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

格納場所と結果セットを関連付けるために、以下の情報をSQLGetDataに与えます。

- *fcType — 結果セットのCデータ型*
- *rgbValue — データ用の出力バッファのアドレス。アプリケーションは、このバッファを割り当て、指定したデータ型で回収したデータを格納するために充分なバッファ・サイズを確保します。*
- *cbValueMax — 出力バッファの最大長さ。戻りデータが整数データのようにCの固定サイズを持つ場合、この値は無視されます。*
- *pcbValue —SQLGetDataへのカレント・コールの前に、利用できる(残っている)データのバイト数を返すために使用する格納バッファのアドレス。*

前述のように、カラムのデータを取得する一般的な方法は、SQLBindColを使って、カラムにローカル・バッファをバインドすることです。カラムのデータは、SQLFetchの際に自動的にバインドされたバッファに保存されます。バッファの大きさが充分であることが確かな場合、ラージ・データを回収するためにバインド方法を使うことができます。SQLGetDataを使った別の方法は、毎回データが一杯入ったバッファを取得します。SQLGetDataを使う場合は、カラムをバインドしません。さもないと、SQLFetchはバインドしたバッファに自動的にカラムのデータを送信します。

➲ データベースからラージ・データ・オブジェクトを回収する:

1. SQLコマンドを実行します。 — SQLExecDirectかSQLExecuteで、SQL問合せを実行します。
2. データをフェッチします。 — SQLFetchを呼び出して、次の行データを取得します。SQLFetchがSQL_NO_DATA_FOUNDを返した場合、問合せの結果セットにある全行が返されたことを意味します。戻りコードがSQL_SUCCESS、又はSQL_SUCCESS_WITH_INFOの場合、フェッチするラージ・データがありますので、次のステップに進んで下さい。
3. ラージ・データ・オブジェクトを取得します。 — SQLGetDataを呼び出して、現在の行にある引数icolで指定したアンバインドされたカラムの一部のデータを取得します。SQLGetDataがSQL_NO_DATA_FOUNDを戻

すまで、このステップを繰り返します。次の行からデータをフェッチしようとする場合は、前のステップに戻って下さい。

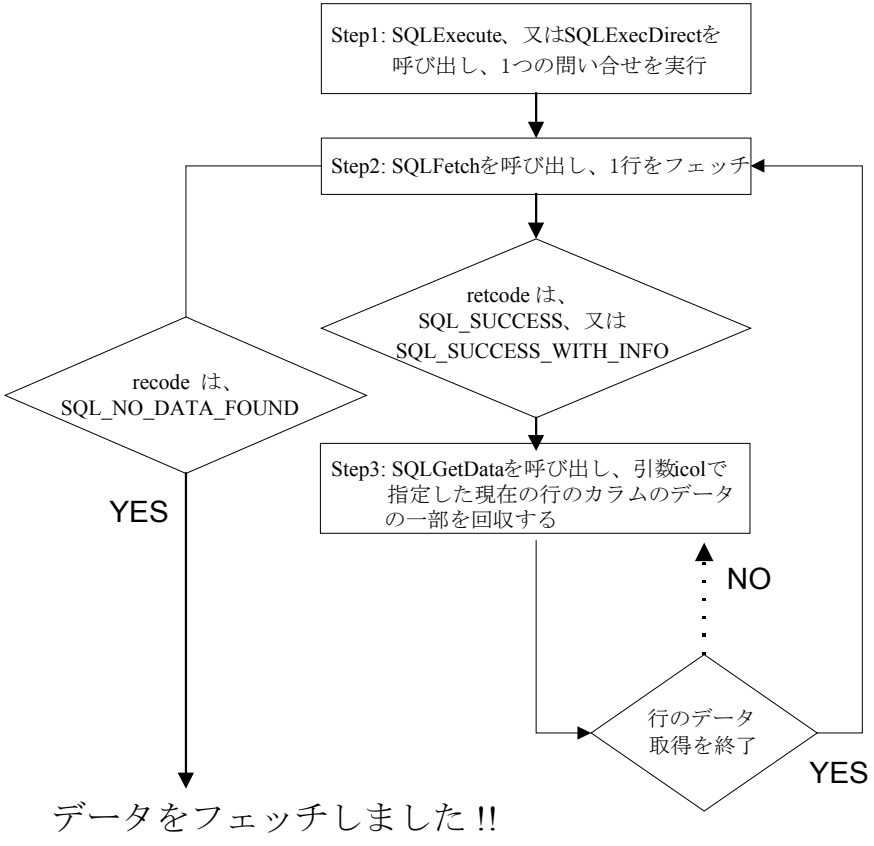


*図 5-3: ラージ・データ回収のためのプログラム・フロー*

以下の例では、表***account***にある全行から、カラム***fname***、***photo***、***memo***をフェッチし表示します。カラム***photo***と***memo***には、ラージ・オブジェクトがあります。バインド方法を使って、***fname***カラムの値を取得し、SQLGetDataを使って***photo***と***memo***カラムの値を取得します。

既に説明したように、格納場所にカラムをバインドすることもできますし、データ回収にSQLGetDataを使うこともできます。これは、全てのデータに対して言えることです。そのように希望すれば、整数値のような通常のデータ型にSQLGetDataを使うこともできます。但し、これには余計なプログラミング負担が伴う上に必然性もないので、現実的な選択ではありません。

➲ 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 WTIH 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 WTIH 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 CAHR, MemoBuf,
                   MAX CHAR SIZE, &cbMemo);

                   if (retcode == SQL SUCCESS
                       || retcode == SQL SUCCESS WTIH INFO)
                      {
                       GetToFile(MemoDataFile, MemoBuf);
                       }
                   else
                        break;
                  }

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

ほとんどのアプリケーションでは、ラージ・カラムにある全データは、通常一時ファイルに回収・保存されてから、表示されるます。このケースが、一度に全てを表示する必要がある静的なデータであるのに対し、ストリーミング（オーディオ、ビデオ）データ、又はページ（ラージ・テキス

ト）データは、一時ファイルが無くても表示することができます。ダイアグラムに示すように、ダブル・バッファ・スキーマを使って、データの回収と表示を同時に行うことができます。

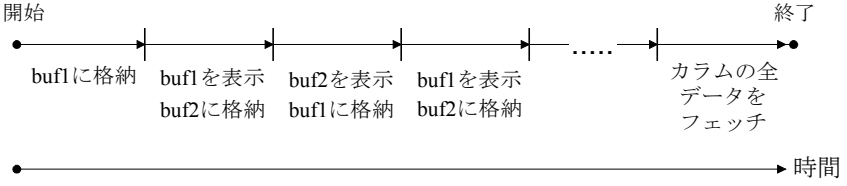


*図 5-4: ダブルバッファを使ったラージ・データの取得*

## SQLGetData操作を中止する

データベースからデータを回収している時にエラーが発生、或いはデータの回収をやめる場合、SQLFreeStmt関数で回収処理を中止することができます。SQLFreeStmt関数を呼び出し、オプションを**SQL_CLOSE**にセット、カーソルを閉じ保留になっている全結果を破棄します。再びSQLExecute、又はSQLExecDirectを呼び出して、同じ問合せでデータを回収する際に、このカーソルを再度開くことができます。

## FOを回収するためにカラムをバインドする

ラージ・データ・オブジェクトを回収し、それをクライアントのファイルに入れる場合、*ファイル・バインド*方法を使ってそれを行うことができます。この方法は、SQLBindCol関数の引数fCTypeを**SQL_C_FILE**にセットし、バッファにファイル名を挿入します。これにより、ファイルが作成され、その中にラージ・オブジェクト・データをコピーされます。

以下の例では、この方法を使って写真を回収し、それをSQLBindCol関数の引数fCTypeの**SQL_C_FILE**にバインドし、バッファpPhotoFlNameにそのファイル名を準備することで、クライアント・ファイルに代入します。*SQL*Fetchを呼び出すと、その写真はファイルにコピーされます。

➲ 例:

```
UCHAR  pPhotoFlName[80];
```

```
SDWORD retcode;

retcode = SQLBindCol(hstmt, 1, SQL C FILE, pPhotoFlName, 80, &cbPhoto);

strcpy(pPhotoFlName, "/disk1/usr/fo/photo");

retcode = SQLExecDirect(hstmt, "SELECT photograph FROM student
                                WHERE name = 'mary'", SQL NTS);

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

## FOのファイル名をフェッチする

4章で解説したように、FO（ファイル・オブジェクト）はサーバーに外部ファイルとして保存されます。FOデータを回収する場合は、上記の3つの方法のいずれかを使うことができます。但し、クライアント・ファイル(**SQL_C_FILE**)をバインドする方法は、常にクライアント側に新しいファイルを作成します。FOのファイル名のみが必要な場合は、組み込み関数**FILENAME()**を使って、ファイル名を取得することができます。

➲ 例、**SQL_C_CHAR**を使ってカラムにバインドする:

```
UCHAR  pPhotoFlName[80];
SDWORD retcode;


retcode = SQLBindCol(hstmt, 1, SQL C CHAR, pPhotoFlName, 80, &cbPhoto);

retcode = SQLExecDirect(hstmt, "SELECT FILENAME(photograph) FROM
                                 student WHERE name = 'mary'", SQL NTS);

retcode = SQLFetch(hstmt);   /* file name of FO goes to pPhotoFlName */
```

今日、ほとんどのマルチメディア・ツールは、オペレーティング・システムのファイルとして保存されているマルチメディア・データを処理しています。マルチメディア・データが**LONG VARCHAR**や、**LONG VARBINARY**のカラムに保存している場合、DBMasterからそのデータをフェッチし、それをマルチメディア・ツールで使えるファイルに再指定しなければなりません。それをファイル・オブジェクトとして保存した場

合、DBMasterからファイル名を取得し、その名前をツールに渡すだけです。

## 5.4 結果セットを操作する

アプリケーションでSELECT文を使って、基礎のデータベースに問合せをすることができます。前節のSQLFetch関数に加え、DBMasterではSQLExtendedFetchを使って、SELECT文で得た結果セットを簡単に前後にブラウズすることも可能です。またSQLSetPosを使って、SQLExtendedFetchで得た結果セットをさらに修正することができます。

### 行セット

*行セット*は、結果セットでウィンドウのような役目を果たします。これを使うと、結果セットの詳細を閲覧することができます。行セットは、常に結果セットの副セットで、同じタプル順になります。

SQLExtendedFetch関数を使って、行セットをフェッチすることができます。但し、SQLExtendedFetchを呼び出す前に、バッファを割り当て、結果カラムをバインドし、フェッチしようとしているタプル数(行セットのサイズ)をセットする必要があります。

様々なオプションでSQLExtendedFetchを呼び出すと、結果セットの行セットをあらゆるポジションに前後移動させることができます。例えば、行セットのサイズが10、オプションSQL_FETCH_FIRSTの場合は、結果セットの始めにウィンドウを移動し、行セットに最初の10タプルを読み込みます。アプリケーションでは、オプションをSQL_ROWSET_SIZEにしたSQLSetStmtOption関数を呼び出して、行セットのサイズを設定する必要があります。SQL_ROWSET_SIZEオプションの初期設定値は、1です。SQLExtendedFetchを呼び出す前に、SQLBindColでカラムをバインドするための十分なバッファ・スペースを割り当てる作業も必要です。

## プログラム・フロー

プログラム・フローは、行セット用のバッファの割り当てを除くとSQLFetchに似ています。行セットのサイズをSQLExtendedFetchコールの間に変更することができます。但し、カラムの出力バッファとカラムのステータス配列が各カラムに対して充分な大きさであることを必ず確認します。新たに割り当てたカラム出力バッファやカラムのステータス配列は全て、再度SQLBindColを呼び出し、再バインドしなければなりません。唯一の例外は、行セットにある行の状態を記録するために使用し、行セットと同じサイズのSQLExtendedFetchの**rgfRowStatus**配列引数が、SQLExtendedFetchを再び呼び出すことによって、再バインドされる時のみです。

## 格納場所のバインド

SQLBindCol関数を使って、結果セットからフェッチしたデータ(行セット)の出力バッファ(**rgbValue**)とカラムのステータス(**pcbValue**)をバインドすることができます。フェッチしたタプル数は、行セットのサイズの範囲内なので、行セットのサイズに応じた出力バッファとカラムのステータス配列のために十分なスペースを割り当てる必要があります。さもないと、SQLExtendedFetch関数は失敗し、不法なアドレス域に出力タプル・データを配置することになります。

➲ SQLBindColの原型:

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

複数のタプルがあると思われる行セットの、出力バッファとカラムのステータス配列をバインドする方法が2つあります。*カラム方向バインド*と*行方向バインド*です。

## カラム方向バインド

SQLSetStmtOption関数のSQL_BIND_TYPEをBIND_BY_COLUMNにセットして、カラム方向バインドであることを指定します。カラム方向バインドを使っている場合、行セットの全タプルにある同じカラム用のバッファは、連番になります。これは、一度に1カラムに充分なバッファを割り当てていることになります。例えば、2つのカラム(intとchar(5))がある行でカラムをバインドするためのコード・フラグメントは、以下のとおりです。

*注: カラム方向バインドが使用されている場合、SQLFetchはSQLExtendedFetchの特殊なケースです。*

➲ 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, "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);
```

## 行方向バインド

SQLSetStmtOption関数のSQL_BIND_TYPEがBIND_BY_COLUMN以外の値にセットされている時、バッファは行ごとにバインドされます。このBIND_BY_COLUMN以外の値は、1タプルに必要な出力バッファのサイズとして使用されます。そのバッファ値は、全カラムのためのカラムの出力値とカラムのステータスです。カラムが既にわかっている場合、アプリケーションは全カラム出力バッファとステータス・バッファで構成された構造を頻繁に定義します。例えば、2つのカラム(int and char(5))がある表のカラムをバインドするためのコード・フラグメントは、以下のとおりです。

➲ 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, "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);
```

## カーソルの配置

カーソルが存在する場合、SQLExtendedFetchは行セットの最初の行にカーソルを配置します。SQLExtendedFetchは、以下で使用されます。

- *他のステートメント・ハンドルからの位置付けUPDATE/DELETE文。SQLExtendedFetchを呼び出し、行にカーソルを配置し、位置付けDELETE文を使って、目的の表の結果セットからその行を削除することができます。例、DELETE ... WHERE CURRENT OF。*
- *SQLGetData。SQLGetDataを呼び出して、バインドされていないカラムのデータを取得することができます。SQLGetDataを呼び出す前に、その行セットのサイズを1にセットします。*
- *オプションにSQL_DELETE、SQL_REFRESH、SQL_UPDATEをセットしているSQLSetPos。*

初めてSQLExtendedFetchを呼び出す前に、定義されていないように見える結果セットの始めの前にカーソルが配置されます。異なるオプションを使うと、既存の行の替わりに、結果セットの始めの前、又は結果セットの後ろにカーソルが配置されます。

## SQLExtendedFetchの引数

➲ SQLExtendedFetchの原型:

```
RETCODE SQLExtendedFetch(
                          HSTMT         hstmt,
                          UWORD         fFetchType,
                          SDWORD        irow,
                          UDWORD FAR   *pcrow,
                          UWORD  FAR   *rgfRowStatus)
```

**SQLExtendedFetch**の戻り値は、以下のとおりです。

- *SQL_SUCCESS*
- *SQL_SUCCESS_WITH_INFO*
- *SQL_NO_DATA_FOUND*
- *SQL_ERROR*
- *SQL_INVALID_HANDLE*

以下の表は、ODBCアプリケーションが要求する行セットと、その時に返された行セットと戻りコードを表しています。ODBCのアプリケーションで行セットのバッファの内容を使う前に、SQLExtendedFetchの戻りコードと、行セットにある各行の行状態をチェックします。

| 要求する行セット | 戻りコード | カーソル位置 | 戻り行セット |
|---|---|---|---|
| 結果セットの1行目の前 | SQL_NO_DATA_FOUND | 結果セットの1行目の前 | 無し。行セットのバッファの内容が定義されません。 |
| 結果セットの1行目に重ねる | SQL_SUCCESS | 行セットの1行目 | 結果セットの最初の行セット。 |
| 結果セットの真中 | SQL_SUCCESS | 行セットの1行目 | 要求した行セット。 |
| 結果セットの最終行に重ねる | SQL_SUCCESS | 行セットの1行目 | 結果セットに重なっている行セットにある行のデータが戻ります。結果セットの外にある行セットの行では、その行状態(rgfRowStatus)は、SQL_ROW_NOROWにあり、その部分の行セット・バッファの内容は、定義されません。 |
| 結果セットの最終行の後 | SQL_NO_DATA_FOUND | 行セットの最終行の後 | 無し。行セットのバッファの内容は定義され |

| 要求する行セット | 戻りコード | カーソル位置 | 戻り行セット |
|---|---|---|---|
| 行の後 | | 最終行の後 | ません。 |

重なるケース(2番目と4番目)は、対照にはなりません。例えば、結果セットに100行あり、その行セットが5であると想定します。以下の表は、フェッチのタイプがSQL_FETCH_RELATIVE(このオプションの定義は下記を参照して下さい)の時、SQLExtendedFetchを呼び出しirowの値を様々に換えた場合に、戻される行セットと戻りコードを表しています。

| 現在の行セット | IROW | 戻りコード | 新しい行セット |
|---|---|---|---|
| 1から5 | -5 | SQL_NO_DATA_FOUND | 無し |
| 1から5 | -3 | SQL_SUCCESS | 1から5 |
| 96から100 | 5 | SQL_NO_DATA_FOUND | 無し |
| 96から100 | 3 | SQL_SUCCESS | 99と100。行セットの行3、4、5では、対応する行の状態は全てSQL_ROW_NOROWにセットされます。 |

### FFETCHTYPE引数

**fFetchType**引数は、結果セットでの位置を決定する方法を指定します。

**fFetchType**の有効値は、以下のとおりです。

- *SQL_FETCH_FIRST*
- *SQL_FETCH_LAST*
- *SQL_FETCH_NEXT*
- *SQL_FETCH_PRIOR*
- *SQL_FETCH_ABSOLUTE*

- *SQL_FETCH_RELATIVE*
- *SQL_FETCH_BOOKMARK*

**irow**引数は、**fFetchType**引数にSQL_FETCH_ABSOLUTE、又はSQL_FETCH_RELATIVEをセットする時に適用します。SQL_FETCH_FIRST、SQL_FETCH_LAST、SQL_FETCH_ABSOLUTEで戻される行セットは、現在の行セットと関係無くフェッチされるので、直前のSQLExtendedFetch呼び出しのfFetchType値に依存しません。

**fFetchType**引数をその他の値にセットすると、直前の行セットに従って行セットをフェッチします。

- *SQL_FETCH_FIRST: 結果セットの最初の行セットをフェッチします。*
- *SQL_FETCH_LAST: 結果セットの最後の完全な行セットをフェッチします。*
- *SQL_FETCH_ABSOLUTE: 結果セットの行**irow**で始まる行セットをフェッチします。*

  **irow** > 0の場合、行**irow**で始まる行セットをフェッチします。

  **irow** < 0の場合、行**irow** + 結果セットのサイズ + 1で始まる行セットをフェッチします。

  例、**irow** = -1の場合、返される行セットの開始行は、結果セットの最後の行です。**irow**が0より小さい場合、結果セットの最終行から逆に数えて、戻された行セットの最初の行を見つけることができます。

  **irow** = 0の場合、SQL_NO_DATA_FOUNDを返し、結果セットの１行目の前にカーソルを配置します(リセット)。

- *SQL_FETCH_NEXT: 次の行セットをフェッチします。現在カーソルが結果セットの１行目の前に位置している場合(初期状態)、これはSQL_FETCH_FIRSTと同じ結果になります。*

- *SQL_FETCH_PRIOR: 1つ前の行セットをフェッチします。現在カーソルが結果セットの最終行の後に位置している場合、これは、SQL_FETCH_LASTと同じ結果になります。*
- *SQL_FETCH_RELATIVE:現在の行セットの1行目から行**irow**の行セットをフェッチします。カーソルが結果セットの1行目の前に位置している場合は以下のようになります。*

  **irow** > 0: 行**irow**で始まる行セットをフェッチします。これはSQL_FETCH_ABSOLUTE値と同等です。

  **irow** < 0: カーソルはそのままで、SQL_NO_DATA_FOUNDが戻ります。

  カーソルが結果セットの最終行の後に位置している場合は、以下のようになります。

  **irow** < 0: 行**irow**+結果セットのサイズ+1で始まる行セットをフェッチします。これは、SQL_FETCH_ABSOLUTEの値と同等です。

  **irow** > 0: カーソルはそのままで、SQL_NO_DATA_FOUNDが戻ります。

  **irow** = 0: 現在の行セットを更新(再フェッチ)します。

- *SQL_FETCH_BOOKMARK: SQL_ATTR_FETCH_BOOKMARK_PTRステートメント属性によって指定されたブックマークで始まる行セットをフェッチします。*

### IROW引数

**irow**引数は、フェッチする行数を指定します。**fFetchType**引数をSQL_FETCH_ABSOLUTE、或いはSQL_FETCH_RELATIVEにセットした場合のみ、**irow**を使います。それ以外では、この値は無視されます。

### PCROW引数

**pcrow**引数は、**SQLExtendedFetch**呼び出しで実際にフェッチした行数(行セットのバッファ)を指定します。**pcrow**の有効値の範囲は、0から行セットのサイズの間です。

### RGFROWSTATUS引数

**rgfRowStatus**引数は、行セットにある全行の状態値の配列です。この配列は、ODBCアプリケーションによって割り当てられます。

SQLExtendedFetchによってセットされる値は、以下のとおりです。

- *SQL_ROW_NOROW:この行のデータは、定義されていません。*
- *SQL_ROW_SUCCESS:この行のデータをフェッチすることに成功しました。*
- *SQL_ROW_ERROR:この行をフェッチする時にエラーが発生しました。*

例えば、行セットのサイズが10で、SQLExtendedFetchで9行のみフェッチした場合(例、**fFetchType**をSQL_FETCH_ABSOLUTEにセットし、**irow**を9にセット)、最後の行の状態値はSQL_ROW_NOROWになり、他の行の状態値はSQL_ROW_SUCCESSになります。

SQLSetPosは、SQLExtendedFetchでフェッチした行セットにある行を操作するために使用します。SQLSetPosに応じて、**rgfRowStatus**引数には以下の値が返ります。

- *SQL_ROW_UPDATED: 行が更新されました。*
- *SQL_ROW_DELETED: 行が削除されました。*
- *SQL_ROW_ADDED: 行が追加されました。*

## 値の戻しとエラー処理

SQLExtendedFetch は、一度に複数の行をフェッチします。フェッチした行、或いはフェッチできたものの警告が受けた行は、行状態値として値SQL_ROW_SUCCESSが与えられます。エラーが発生した場合、フェッチした行には、値SQL_ROW_ERRORが与えられ、フェッチを停止します。その後に続く行セットの行は、SQL_ROW_NOROWにマークされます。

以下の4つの例は、サイズ5の行セットを使用した例です。

➲ 例 1、行セットの全行がフェッチされます:

```
     +------+-----------------+
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 |
     +------+-----------------+
```

➲ 例 2、結果セットの終端付近で、行セットの全行をフェッチしました:

```
     +------+-----------------+
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   |
     +------+-----------------+
```

➲ 例 3、2番目の行をフェッチする際にエラーが発生しました:

```
     +------+-----------------+
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   |
     +------+-----------------+
```

➲ 例 4、フェッチされる行は全くありません:

```
     +------+-----------------+
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の戻り値は、行セットにある全行の値によります。各行にあるカラムの状態は、カラム状態配列をチェックします。

SQLExtendedFetchの戻り値は、以下のとおりです。

- *SQL_SUCCESS- エラーや警告が無く、最低1行がフェッチされました。*
- *SQL_NO_DATA_FOUND- フェッチする行がありません。*
- *SQL_SUCCESS_WITH_INFO- エラーは無いものの、警告を受けました。警告情報のために**SQLError** が呼ばれ、前回の警告の詳細が戻されます。*
- *SQL_ERROR- エラーが見つかりました。後に続く**SQLExtendedFetch** 呼び出し全てで、同じエラーになります。そしてODBCアプリケーションは、それ以上結果セットにアクセスすることはできません。(例えば、ロックが解放された後でも、ロック・タイムアウトが発生します。SQLExecuteを実行し、結果セットを再生成する必要があります。)*

## SQLSetPosを使った表の修正

結果セットが単一表から生成された場合、結果セットの各行を一意に目的の表の1行にマップすることができます。結果セットの副セットである行セットの各行は、OID(オブジェクトID)を通じて目的の行の1物理行と関連付けられます。

➲ 例 1、以下の問合せ文からの行セットは、全て更新可能です:

```
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;
```

➲ 例 2、以下の問合せ文からの行セットは、更新不可能です:

```
create table t1 (c1 int, c2 int, c3 char(5))
     select * from t1,t2 ;
     select c1+c2 from t1 ;
     select c1*2 from t1 ;
```

現在のところSQLSetPosは、単一表での単純スキャンの修正のみサポートしています。**c1**のような式のみ修正することができます。**c1*2、c1+1、c1+c2**のような式は、修正することができません。

必要であれば、カラムの初期設定値は、プロジェクションにないカラムに適用します。例、SQLSetPosを経由する行の挿入。バインドしたカラムは、単一表で単純なスキャンで表スキーマの副セットに逆比例する、プロジェクションの副セットです。アンバインドしたカラムを操作するためだけに、SQLPutDataを使うことができます。

### SQLSETPOSの引数

➲ SQLSetPosの原型:

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

SQLSetPosの戻り値は以下のとおりです。

- *SQL_SUCCESS*
- *SQL_SUCCESS_WITH_INFO*
- *SQL_NEED_DATA*
- *SQL_ERROR*
- *SQL_INVALID_HANDLE*

SQLSetPosは行セットを操作するので、SQLExtendedFetchを呼び出した後に呼び出す必要があります。操作される行セットは、前回のSQLExtendedFetchで得たものです。**rgfRowStatus**引数（行状態配列）は、SQLSetPos（異なるオプションに応じて）でセットし、対応するSQLExtendedFetchから暗に渡されるすることに留意して下さい。SQLSetPosが行データにアクセスする前に、ODBCアプリケーションは再度、SQLSetPos戻り値と行状態配列をチェックする必要があります。結果セットが修正不可能な場合、SQLSetPosはエラーを返します。

異なるオプション値のSQLSetPosでセットされる可能性のある行状態の値は、以下のとおりです。

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

### IROW引数

**irow**は、**fOption**で指定した操作が実行される行セットにある行数です。**irow**の値が0の場合、その操作は行セットの全行に適用されます。

### FOPTION引数

SQLExtendedFetchから取得した行セットに適用する操作。

有効値は、以下のとおりです。

- *SQL_POSITION*
- *SQL_REFRESH*
- *SQL_UPDATE*
- *SQL_DELETE*
- *SQL_ADD*

SQL_POSITIONは、カーソルが必要な操作のために、行セットの行にカーソルを配置します。このオプションは、行状態配列を全く変えません。

**irow** = 0の場合: 行セット全体にカーソルを配置します。

**irow** = nの場合: 行nにカーソルを配置します。(n =1、又は<= 行セットのサイズ)。

このオプションが、複数の行にカーソルを配置するために使用される場合、位置付け文は選択された行の最初の行でのみ実行されます。

➲ 例:

```
/* hstmtS is for SQLExtendedFetch, SQLSetPos and SQLSetCursorName */
/* hstmtU is for positioned statement                              */

/* use hstmtS to query */
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は、行セットの行データを更新します。これは、行セットのバッファに同じウィンドウを再度フェッチすることです。新たにフェッ

チされた行の行状態は、SQL_ROW_SUCCESSにセットされ、結果セットにない行はSQL_ROW_NOROWにセットされます。

**irow** = 0の場合、行セット全体にカーソルを配置し、更新します。

**irow** = nの場合、DBMasterではSQL_REFRESHの**irow**に別の値をセットするサポートをしていません。

このオプションが成功した場合、更新した行の行状態は、SQL_ROW_SUCCESSにセットされます。更新した行セットがSQL_DELETEオプションで作成された「holes」で一杯の場合、ウィンドウ（行セット）は結果セットの中で前後に移動します。

SQL_UPDATEは行データを更新します。目的の表の適用行は、行セットのバッファの行データで更新されます。問題無く更新された行の行状態は、SQL_ROW_UPDATEDにセットされます。SQL_ROW_DELETEDにマークされた行を更新することはできません。

**irow** = 0の場合、行セット全体にカーソルを配置し、更新します。

**irow** = nの場合、行nにカーソルを配置し、更新されます。

SQL_DELETEは、行セットによってマップされた目的の表にある適用行を削除します。SQL_ROW_DELETEDにマークされた行を削除することはできません。行が削除された場合(SQL_ROW_DELETEDにセット)、それらに次の操作を実行することはできません。つまり、位置付けUPDATE/DELETE文、SQLGetDataの呼び出し、SQL_POSITION以外のオプションでのSQLSetPosの呼び出しを行うことはできません。(SQL_ROW_DELETEDにセットされた行で、SQL_SET_POSITIONのSQLSetPosのみ呼び出すことができます)。

**irow** = 0の場合、行セット全体にカーソルを配置し、削除します。

**irow** = nの場合、行nにカーソルを配置し、削除します。

SQL_ADDは、行データを追加します。追加した行の行状態は、SQL_ROW_ADDEDにセットされます。このオプションが適用されると、行セットは挿入されるデータ用のユーザー入力バッファとして使用されるだけです。目的の表には、行セットにある行にマップされる行はありません。これは、行セットのサイズより大きいirowを認めるだけのオプション

です。このオプションは、カーソル位置を変えません(カーソルの配置は行われません)。行セットのバッファにバインドされていなカラムを挿入する時、初期設定値(適用可能であれば)が使用されます。又はNULL値が使用されます(初期設定値が利用できない場合)。

**irow** = 0の場合、行セットの全行に追加します。

**irow** = 1～行セットのサイズの場合、行**irow**を追加します。

**irow** > 行セットのサイズの場合、行**irow**は適切なオフセットで行セットのバッファの最初から見つけることができます。例えば、**irow** = 行セットのサイズ + 1、或いは**irow** = 行セット・サイズ + 2の場合、行**irow**を追加します。

ODBCアプリケーションは、SQL_ROWSET_SIZEで指定した行セットのバッファのスペースよりも大きいバッファ・スペースを割り当てます。これにより、ODBCアプリケーションのプログラミングはより単純になります。予備のバッファを割り当てず、**irow**が行セットのサイズより大きくなる場合、不正なメモリ・アドレスにアクセスしようとして、アプリケーション・プログラムのエラーになるかもしれません。

### FLOCK引数

目的の表で適用する操作行をロック、又はロック解除します。

fLockの有効値は以下のとおりです。

- SQL_LOCK_NO_CHANGE: *行のロック・モードを変更しません。*

## カラム・インジケータ

ODBCアプリケーションでカラムにNULL値を挿入したい場合、唯一のインターフェースはカラム・インジケータ(状態)です。情報を回収したり、カラムにNull値の挿入を認めるよう指定したりすることができるホスト変数(INSERT INTO t1 VALUES (?,?)のようなホスト変数)はありません。SQLExtendedFetchとSQLSetPosと使用する場合、SQLBindColのカラム・インジケータは、フェッチと修正する情報を指定する唯一のインターフェースです。

例えば、ODBCアプリケーションでカラムをNull値に更新しようとする場合(或いはそのカラムにNull値を挿入する場合)、対応するカラム・インジケータは、SQL_UPDATEかSQL_INSERTオプションでSQLSetPosを呼び出す前に、SQL_NULL_DATAにセットする必要があります。初期設定値が適用される場合、同じ方法が使用されます。但し、カラム・インジケータはSQL_DEFAULT_PARAMにセットしなければなりません。

## SQLPutData

SQLExtendedFetchを使用する場合、SQLGetDataはアンバインドしたカラム(主にBLOB/ファイル・オブジェクト・カラム)をフェッチする唯一の方法です。同様に、SQLPutDataを使用する場合、SQLSetPosを呼び出してアンバインドされたカラム(主にBLOB/ファイル・オブジェクト・カラム)を修正することができます。使用するカラム・インジケータはありません。SQLPutDataの引数**cbValue**のみ使用することができます。行セットのサイズは、SQLGetData又はSQLPutDataを実行する前に1にします。

非プロジェクション・カラムには、初期設定値はSQL_ADDオプションのSQLSetPosによって適用されます。

➲ 例 1、表を作成する:

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

➲ 例 2、select問合せ:

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

➲ 例 3、次の呼び出しで、表t1を修正する:

```
/* 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
*/
```

カラム**c3** はプロジェクションにありません。カラム**c1**と**c2**のみ行セットで見つけることができます。言い換えると、1つ予備のタプルを挿入する場合、SQLSetPosはカラム**c1**と**c2**の値は行セットから取得し、カラムc3には初期設定値'col3'を使います。t1の対応する行を*行セット*の1行目に更新するために、**c3**がプロジェクションに無いので、カラムc3が変更されなかったことを確実にします。

バインドされたカラムでは、SQLSetPosは行セット(バインドされたバッファ)から、(SQL_ADDとSQL_UPDATEオプションのために)必要な全入力データを取得します。

各アンバインド・カラムでは、BLOB (LONG VARCHAR、又はLONG VARBINARY)でもファイル・オブジェクト型でもない時、SQLSetPosに必要な場合、初期設定値が適用されます。SQLSetPosを実行する時に初期設定値が使用されるので、SQLPutDataを使って、この種類のカラムにデータを配置することはできません。

アンバインドされたBLOB/ファイル・オブジェクト・カラムでは、それらを修正するためにSQLPutDataを使う必要があります。SQLPutDataの前とSQLSetPosの後ろに、アンバインドされた全BLOB/ファイル・オブジェクトのカラムを見つけるためにSQLParamDataを実行する必要があります。

BLOB(LONG VARCHARとLONG VARBINARY) カラムは、

- *NULL値を入力するために、引数**cbValue**のSQLPutDataを呼び出し、SQL_NULL_DATAにセットします。*
- *初期設定値を入力するために、引数**cbValue**のSQLPutData を呼び出し、SQL_DEFAULT_PARAMにセットします。*

- *データを入力するために、引数**rgbValue**に入力データを入れたSQLPutDataと、SQL_NTS、又は引数**cbValue**の**rgbValue**の長さを呼び出します。データを入力するために、LONG VARCHARとLONG VARBINARYデータ型のためのSQL_C_TYPEは、SQL_C_LONGVARCHARとSQL_C_LONGVARBINARYです。*

ファイル・オブジェクト・カラムは、

- *NULL値を入力するために、引数**cbValue**のSQLPutDataを呼び出し、SQL_NULL_DATAにセットします。*
- *初期設定値を入力するために、引数**cbValue**のSQLPutDataを呼び出し、SQL_DEFAULT_PARAMにセットします。*

➲ 例 4、SQLSetPosを実行し、呼び出しをし、SQLPutDataを実行してデータを入力する:

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

これは以下のSQLPutDataコールが、プロジェクションにあるファイル・オブジェクトのカラム**col**に、引数rgbValueからデータを挿入することを表しています。**col**は、プロジェクションのターゲット・ファイルにあるオブジェクト・カラムの索引です。オプションSQL_SPOS_FO_DATAは、入力データをバインドするために、システムに強制的にSQL_C_CHAR、又はSQL_LONGVARCHARを使わせます。この種のデータ入力のために、システム・ファイルが自動的に作成されます。

➲ 例 5、SQLSetPosを実行し、呼び出しをし、SQLPutDataを実行してユーザー・ファイルを入力する:

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

これは以下のSQLPutDataコールが、プロジェクションにあるファイル・オブジェクトのカラム**col**に、引数**rgbValue**でファイル名が指定されたユーザー・ファイルを挿入します。オプションSQL_SPOS_FO_SFILEは、入力データをバインドするために、システムに強制的にSQL_C_CHAR、又はSQL_FILEを使わせます。

➲ 例 6、SQLSetPosを実行し、呼び出しをし、SQLPutDataを実行してシステム・ファイルを入力する:

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

これは以下の**SQLPutData**コールが、プロジェクションにあるファイル・オブジェクトのカラム**col**に、引数**rgbValue**でファイル名が指定されたシステム・ファイルを挿入します。オプションSQL_SPOS_FO_CFILEは、入力データをバインドするために、システムに強制的にSQL_C_FILE、又はSQL_LONGVARCHARを使わせます。

以下は、BLOBとファイル・オブジェクトのデータを入力するために、どのように**SQLSetPos**と**SQLPutData**を使うのかを示してします。

➲ 例 7、表スキーマを作成する:

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

➲ 例 8、select問合せ:

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

➲ 例 9、コードを使う:

```
/* do not bind any column
*/

SQLSetStmtOption(hstmt, SQL ROWSET SIZE, 1);

       :

/* execute and fetch
*/

SQLExecute(hstmt);

SQLExtendedFetch(hstmt, SQL FETCH FIRST, 0, &crow, rqfRowStatus);


/* 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));
```

## SQLSetPosを使う

SQLSetPosは、一度に複数の行を修正します。そのため、戻り値のルールは、SQLExtendedFetchに似ています。行に実行した操作(或いは警告を受けた操作)は、使用するオプションに応じてマークされます。エラーが発生してトランザクションが中止するような重大なエラーでない限り、操作は停止しません。その行はSQL_ROW_ERRORにマークされます。

この法則は、以下のとおりです。

- *エラーや警告が発生しない場合、SQL_SUCCESSを返します。*
- *フェッチする行が無い場合、SQL_NO_DATA_FOUNDを返します(SQL_REFRESHオプションのみ)。*
- *操作中に、エラーや警告が見つかった場合、SQL_SUCCESS_WITH_INFOを返します。SQLErrorを呼び出して、完全なエラー情報を取得することができます。警告だけでエラーが無い場合、最後の警告のみが記録されます。*
- *操作中に重大なエラーが見つかった場合、SQL_ERRORを返します。*

エラーが発生したタプルの前に実行したSQLSetPosの部分的な結果は、元に戻されないことに注意して下さい。つまり、この関数が核操作ではありません。SQLSetPosの呼び出し(オプションSQL_REFRESHとSQL_POSITIONを除く)は、自動コミット・モードがONの場合、操作を実行した後それをコミットします。

## SQLSetPosの制限

副問合せのある問合せから得た結果セットは、修正不可能です。このような結果セットの場合、SQLSetPosを呼び出すことはできません。

# 6　エラー操作

前章までを熟読すると、ODBCプログラムが構築できるようになります。但し、ODBC関数を呼び出している際に、問題が起こった場合はどのように対処すればいいのでしょう？本章では、エラーが発生した時にそのエラー情報を取得する方法について説明します。また、ユーザーがシステム・カタログ（システム表）から情報を取得することができるODBCカタログ関数についても解説します。その他のODBC関数もここに掲載しています。サポートしているデータ型、組み込み関数、ODBC関数のような、データソースについてのシステム情報を取得するために使用する関数についても紹介します。

この章では、以下について説明します。

- *ODBC関数へのコールが失敗した際の、SQLError関数を使ったエラー情報の詳細の取得。*
- *SQLTables*、*SQLColumns*、*SQLStatistics*、*SQLSpecialColumns*のようなカタログ関数を使った、*表スキーマや統計情報のようなカタログ情報の回収。*
- *SQLGetTypeInfo*、*SQLGetInfo*、SQLGetFunctions*関数を使った、データソースについてのシステム情報の取得。*

*注:*　*エラー情報は、DBMaster 4.0 (ODBC 3.0)を使って、本章で説明している場所から個々に回収します。詳細については、8章の「ODBC 3.0関数」*を参照して下さい。

# 6.1 エラー情報を回収する

アプリケーションでODBC関数を実行した際にエラー・コードが戻る場合、エラーを引き起した原因を見極めるために、詳細なエラー情報が必要です。この節では、エラー情報を回収する*SQL*Error関数の使い方について説明します。

## ODBCに定義されている一般エラー・コード

ODBC関数を呼び出した際に、受け取る可能性のある戻りコードは以下のとおりです。

- ***SQL_SUCCESS*** *—ODBC関数は順調に実行されました。*
- ***SQL_SUCCESS_WITH_INFO*** *—ODBC関数は順調に実行されましたが、警告情報が返されました。*
- ***SQL_NO_DATA_FOUND*** *— フェッチできるデータがもうありません。*
- ***SQL_ERROR*** *— エラーが発生し、関数は実行されませんでした。*
- ***SQL_INVALID_HANDLE*** *— 無効なハンドルが検出され、関数は実行されませんでした。*
- ***SQL_NEED_DATA*** *— ドライバは、アプリケーションがパラメータのデータ値を送信するよう、表示しています。*

アプリケーションでODBC関数（SQLError自身を除く）を呼び出した際に、戻りコードが**SQL_ERROR**、又は**SQL_SUCCESS_WITH_INFO**である場合、SQLErrorを呼び出し、追加のエラー情報を取得することができます。

## SQLErrorの使用方法

SQLErrorは、エラー・メッセージ、エラー状態、ドライバのネイティブ・エラー・コードを含む入力ハンドルのエラー情報を取得するために使用します。ドライバのネイティブ・エラー・コードは、各ドライバで定義され

るエラー・コードです。これは、ドライバによって異なるかもしれません(DBMasterのネイティブ・エラー・コードは、*付録C*を参照して下さい)。

前回のODBC関数が返したエラー・コードが、**SQL_ERROR**、又は**SQL_SUCCESS_WITH_INFO**の時、アプリケーションはSQLErrorを呼び出します。

➲ SQLErrorの原型:

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

SQLErrorの引数にある3つのハンドル全てをSQLErrorに渡す必要はありません。ODBCドライバは、最も後方の非nullハンドルから関係する戻りコードを見つけます。

➲ 例:

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

➲ 返されたエラー情報は**hstmt**にあります:

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

ドライバはhdbcに関連するエラー情報を戻します。必要なエラー情報を間違い無く回収するために、アプリケーションが正しいハンドルをSQLErrorに渡すかを確認する必要があります。

回収するエラー情報が無い場合、SQLErrorは**SQL_NO_DATA_FOUND**を返します。SQLErrorを呼び出す度にエラー情報が返され、ハンドルにあるエラー情報は消去されます。これは、1つのODBC関数呼び出しのエラー情報が一度しか回収できないことを意味します。

SQL Access Group SQL CAE specification (1992)とX/Openは、SQLErrorが戻す**SQLSTATE**値を定義しています。その値は、2文字のクラス値と3文字のサブクラス値が後ろに続く5文字の文字列です。例えば、クラス値01が警告で、対応する戻りコードは、**SQL_SUCCESS_WITH_INFO**です。ODBCで定義される***SQL*STATE**値についての詳細な情報は、*「Microsoft ODBCプログラミング・リファレンス」*を参照して下さい。

➲ 例、SQLStateのあるSQLError:

```
#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  realMsqLen;          /* 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 warninqs 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_HDBC, 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 HDBC, SQL NULL HDBC, hstmt, sqlState,
                          &nativeErr, errMsg, MSG LEN, &realMsgLen);
     print err(sqlState, nativeErr, errMsg, realMsgLen);
     return;
    }
```

## エラー行列

ODBCでは、複数のエラー・コードをエラー行列に保存し、1つのハンドルに関連付けることが可能です。1件づつエラー・コードを回収するために、複数回、SQLErrorを呼び出します。*Database Consistency Checking* (DBCC)操作では、DBMasterは現時点では複数のエラーを返すだけです。これらのエラーは、エラー行列に保存されます。

エラー行列にあるエラーを全てフェッチするまで、アプリケーションでSQLErrorを何度も呼び出すことができます。全てのエラーがフェッチされると、SQLErrorは**SQL_NO_DATA_FOUND**を返します。

➲ 例、表accountの整合性のアプリケーション・チェック:

```
#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 = SQLExecute(hstmt, "check table account", SQL NTS);

do {
    retcode = SQLError(SQL NULL HDBC, hdbc, SQL NULL 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_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO));
```

## 6.2 カタログ関数

リレーショナル・データベース内には、表、カラム、権限等、*カタログ*と呼ばれる情報を記録するいくつかのシステム表があります。データベースにある表や索引のスキーマを知りたい場合、カタログを見て情報を入手することができます。

ODBCには、カタログ情報を回収するための関数がたくさんあります。これらの関数は、カタログ関数と呼ばれています。この節では、頻繁に使用される4つのカタログ関数、SQLTables、SQLColumns、SQLStatistics、SQLSpecialColumnsについて説明します。

- **SQLTables** – *データベースにある表やビューの一覧を取得します。*
- **SQLColumns –** *指定した表についてのカラム情報を取得します。*
- **SQLStatistics –** *これらの表に関係する表と索引についての統計情報を取得します。*
- **SQLSpecialColumns –** *表にある行を一意に識別するオプションのカラムの集まりを取得します。*

カタログ関数は、全て同じ働き方をします。カタログ関数を呼び出す時に、パラメータを使って情報を指定すると、結果セットが返ります。その結果セットからデータをフェッチすることができます。

## 検索パターン

カタログ関数の引数によっては、希望するオブジェクトを選択するために検索パターンを使うことができます。最も簡単な検索パターンは、探している物を正確に一致する文字列です。加えて、検索パターンにワイルドカードのメタキャラクタを使って、より強力な検索を行うこともできます。DBMasterでは、アンダーバー(_)、パーセント(%)、バックスラッシュ(\)を使うことができます。

- *アンダーバー(_)は、任意の1文字に相当します。*
- *パーセント(%)は、1以上の文字に相当します。*
- *バックスラッシュ(\)は、メタキャラクタの(%)や(_)を検索パターンのリテラル文字として使用できるようにします。検索パターンでバックスラッシュ(\)をリテラル文字として使用するためには、それを2度(\\)使います。*

例えば、表名の検索パターンが%A%の場合、関数は名前に文字Aを含む全表を返します。表名の検索パターンが_A_の場合、関数は名前が3文字で

真中に文字Aがある全表を返します。表名の検索パターンが%の場合、関数は全表を返します。

表名が***TAB_TEST***の情報を回収する場合、検索パターンとして***TAB_TEST***を使うと、結果セットに表***TAB1TEST***と表***TAB2TEST***なども取得することになります。これは、本来の希望と異なります。この問題を解決するために、***TAB_TEST***のようにメタキャラクタの前にバックスラッシュを置きます。

*注:* *Cコンパイラを通じてこの文字列を渡すとき、*TAB_TEST*の替わりに*TAB_TEST*を指定する必要があります。これは、Cコンパイラも"\"をエスケープ文字として扱うからです。SQLTablesの例は以下を参照して下さい。*

## SQLTables

データベースに接続し、表の全て或いは特定の部分についての情報を判断しようとする時、SQLTablesを呼び出す時に条件を付加すると、その回答を得ることができます。

➲ SQLTablesの原型:

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

SQLTablesのための引数は、以下のとおりです。

- *hstmt — 回収した結果のためのステートメント・ハンドル*
- *szTableQualifier —DBMasterではサポートしていませんので、**NULL**又は空の文字列にします。*

- *cbTableQualifier —szTableQualifierの長さ。0にします。*
- *szTableOwner — 所有者名へのポイント。所有者は、表やビューの作成者です。検索パターンや、**NULL**値の文字列です。**NULL**値を使うと、全所有者を表示します。*
- *cbTableOwner — szTableOwnerの長さ、又は**SQL_NTS**。*
- *szTableName — 表名やビュー名へのポイント。検索パターンや、**NULL**値の文字列です。**NULL**値を使うと、全部の名前を表示します。*
- *cbTableName —szTableNameの長さ、又は**SQL_NTS**。*
- *szTableType — 表の種類のリスト(表/ビュー)。*
- *cbTableType — szTableTypeの長さ。*

*注:* *検索パターンとして使用する文字列が、**szTableQualifier**、**szTableOwner**、**szTableName**.にあります。これらの3つの引数と対応する文字列の長さの引数**cbTableQualifier**、**cbTableOwner**、**cbTableName**は、他の3つのカタログ関数、SQLColumns、SQLStatistics 、SQLSpecialColumnsにも登場します。*

SQLTablesは、以下のカラムで構成される結果セットを返します。

| カラムNo. | カラム名 | データ型 |
|---|---|---|
| 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**は、ユーザーの与える条件に応じて結果セットを返します。例えば、szTableNameが%A%で、szTableOwnerが_A_の時、結果セットには、表の名前に文字Aを含まれ、所有者名の名前が3文字で真中の文字がAである表が全てあります。データベースにある全ての表の名前を取得する

場合は、szTableQualifier、szTableOwner、szTableNameを**Null**にセットするだけです。

実際SQLTablesは、SQLExecDirectを使って問合せを実行する方法のようにみなすことができます。これは、結果セットを取得するために**SQLFetch**を使う必要があることを意味します。SQLFetchを使う前に、SQLBindColを使って結果セットにカラムをバインドしなければなりません。

以下のコードは、SQLTablesの例です。表名が***TAB_TEST1***と***TAB_TEST2***の2つの表があると想定します。SQLTablesを呼び出して、***TABLE_TYPE***、***TABLE_QUALIFIER***、***TABLE_OWNER***、***TABLE_NAME***の順序で、***TAB_TEST1***と***TAB_TEST2***の情報を取得します。

➲ 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:　　関数が結果セットを返すと、SQLBindColとSQLFetchを使って、結果セットの行を取得する必要があります。SQLTables、SQLColumns、SQLStatistics、SQLSpecialColumnsのいずれの関数でも同様です。*

## SQLColumns

SQLTables関数を使ってデータベースにある表の名前を取得するように、SQLColumns関数を使うと、特定の表にあるカラム情報を取得することができます。

➲ SQLColumnsの原型:

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

SzTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableNameは、*SQL*Tablesと同じ定義です。SzColumnNameは、カラム名の検索パターンの文字列にポイントします。cbColumnNameは、szColumnNameの長さです。

SQLTables関数と同様、上記の引数にある条件に合致する結果セットは、カラム情報と共に返されます。

以下の表は、結果セットのカラムの一覧です。

| カラム No. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *TABLE_QUALIFIER* | **VARCHAR(128)** | |
| 2 | *TABLE_OWNER* | **VARCHAR(128)** | |
| 3 | *TABLE_NAME* | **VARCHAR(128)** | **非NULL** |
| 4 | *COLUMN_NAME* | **VARCHAR(128)** | **非NULL** |
| 5 | *DATA_TYPE* | **SMALLINT** | **非NULL** |
| 6 | *TYPE_NAME* | **VARCHAR(128)** | **非NULL** |
| 7 | *PRECISION* | **INTEGER** | |
| 8 | *LENGTH* | **INTEGER** | |
| 9 | *SCALE* | **SMALLINT** | |
| 10 | *RADIX* | **SMALLINT** | |
| 11 | *NULLABLE* | **SMALLINT** | **非NULL** |
| 12 | *REMARKS* | **VARCHAR(254)** | |

結果セットは、***TABLE_QUALIFIER***、***TABLE_OWNER***、***TABLE_NAME***の順です。SQLBindColを使って結果セットにそのカラムをバインドし、SQLFetchを使って結果をフェッチする必要があります。

## SQLStatistics

SQLStatisticsは、指定した表とその表に関連する索引の統計の一覧を回収します。

➲ SQLStatisticsの原型:

```
RETCODE SQLStatistics (
                        HSTMT    hstmt,
                        UCHAR   *szTableQualifier,
```

```
                    SWORD    cbTableQualifier,
                    UCHAR   *szTableOwner,
                    SWORD    cbTableOwner,
                    UCHAR   *szTableName,
                    SWORD    cbTableName,
                    UWORD    fUnique,
                    UWORD    fAccuracy)
```

szTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableNameは、SQLTablesとSQLColumnsと同じ定義です。fUniqueは返される索引の種類を指定するために使用し、fAccuracyは結果セットの***CARDINALITY***と***PAGES***カラムの重要性を指定するために使用します。

*注:* ***fUnique****には、SQL_INDEX_UNIQUEとSQL_INDEX_ALLの2つのオプションがあります。****fAccuracy****にも、SQL_ENSUREとSQL_QUICKの2つのオプションがあります。*

以下の表は、結果セットのカラム一覧です。

| カラム No. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *TABLE_QUALIFIER* | **VARCHAR(128)** | |
| 2 | *TABLE_OWNER* | **VARCHAR(128)** | |
| 3 | *TABLE_NAME* | **VARCHAR(128)** | **非NULL** |
| 4 | *NON_UNIQUE* | **SMALLINT** | |
| 5 | *INDEX_QUALIFIER* | **VARCHAR(128)** | |
| 6 | *INDEX_NAME* | **VARCHAR(128)** | |
| 7 | *TYPE* | **SMALLINT** | **非NULL** |
| 8 | *SEQ_IN_INDEX* | **SMALLINT** | |
| 9 | *COLUMN_NAME* | **VARCHAR(128)** | |
| 10 | *COLLATION* | **CHAR(1)** | |
| 11 | *CARDINALITY* | **INTEGER** | |
| 12 | *PAGES* | **INTEGER** | |

| カラム No. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 13 | *FILTER_CONDITION* | **VARCHAR(128)** | |

***TYPE*** カラムには、値**SQL_TABLE_STAT**、又は**SQL_INDEX_OTHER**のいずれかがあります。**SQL_TABLE_STAT**は、表の統計を含む行を表示し、***NON_UNIQUE***、***INDEX_QUALIFIER***、***INDEX_NAME***、***SEQ_IN_INDEX***、***COLUMN_NAME***、***COLLATION***、***FILTER_CONDITION***カラム（索引に使用）は、**NULL**になります。一方、**SQL_INDEX_OTHER**は、索引の統計を含む行を表示します。

SQLTablesとSQLColumnsと同様に、SQLBindColとSQLFetchで結果セットのデータを回収する必要があります。結果セットのカラムの並びは、***NON_UNIQUE***、***TYPE***、***INDEX_QUALIFIER***、***INDEX_NAME***、***SEQ_IN_INDEX***です。関数のコード例は、SQLTablesのコード例を参照して下さい。

## SQLSpecialColumns

関数名に示されるように、SQLSpecialColumnsは表の行を一意に指定する特別なカラムを返します。

➲ SQLSpecialColumnsの原型:

```
RETCODE SQLSpecialColumns (
                            HSTMT    hstmt,
                            UWORD    fColType,
                            UCHAR   *szTableQualifier,
                            SWORD    cbTableQualifier,
                            UCHAR   *szTableOwner,
                            SWORD    cbTableOwner,
                            UCHAR   *szTableName,
                            SWORD    cbTableName,
                            UWORD    fScope,
                            UWORD    fNullable);
```

hstmtはステートメント・ハンドル、szTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableNameは、*SQL*Tablesと

同じ定義です。fColTypeは、返すカラムのタイプを指定します。fScopeは、特別なカラムに最低限必要なスコープです。fNullableは、**NULL**値を持つことができる特別なカラムを返すかどうかを決定します。

*注:* ***fColType****には、SQL_BEST_ROWIDとSQL_ROWVERの2つのオプションがあります。****fScope****には、SQL_SCOPE_CURROW、SQL_SCOPE_TRANSACTION、SQL_SCOPE_SESSIONの3つのオプションがあります。****fNullable****には、SQL_NO_NULLSとSQL_NULLABLE.の2つのオプションがあります。*

以下の表は、結果セットのカラムの一覧です。

| カラムNo. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *SCOPE* | **SMALLINT** | |
| 2 | *COLLUMN_NAME* | **VARCHAR(128)** | **非NULL** |
| 3 | *DATA_TYPE* | **SMALLINT** | **非NULL** |
| 4 | *TYPE_NAME* | **VARCHAR(128)** | **非NULL** |
| 5 | *PRECISION* | **INTEGER** | |
| 6 | *LENGTH* | **INTEGER** | |
| 7 | *SCALE* | **SMALLINT** | |
| 8 | *PSEUDO_COLUMN* | **SMALLINT** | |

DBMasterには、Oracleの*ROWID*や、Ingresの*TID*に似た、特別な行識別子*OID*があります。*OID*は、表で仮想カラムとして扱われます。つまり、SELECT * FROM ACCOUNTのような問合せでは、そのようなカラム名を返しませんが、selectリストやWHERE句で*OID*を使い明示的に指定することで、そのレコードをフェッチすることができます。

一旦、fColTypeに**SQL_BEST_ROWID**を指定すると、SQLSpecialColumnsで返された結果セットには、単純にカラム名が*OID*の行が含まれます。この特別なカラムを使って、fScopeに定義したスコープの中の行を再度selectすることができます。SELECT文の結果は、必ず全く行が無いか、或いは1行のいずれかです。関数のコード例は、SQLTablesを参照して下さい。

*注:* **fColType**、**fScope**、**fNullable***の引数に*、DBMaster *でサポートしていない特性を指定した場合*、SQLSpecialColumns*は行の無い行セットを返します。***hstmt** *での*、SQLFetch*や*SQLExtendedFetch*の後に続く呼び出しは*、**SQL_NO_DATA_FOUND** *を返します。*

# 6.3 システム情報

SQLGetTypeInfo、SQLGetInfo、SQLGetFunctionsを使って、データソースについてのシステム情報を取得することができます。これらのODBC関数は、以下の節で例を紹介します。

## SQLGetTypeInfo

SQLGetTypeInfoを使うと、データソースでサポートするデータ型についての情報を取得することができます。

➲ SQLGetTypeInfoの原型:

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

fSqlTypeに値を与えると、SQLGetTypeInfoは結果セットに関連する種類の情報を返します。SQLBindColを使って結果セットに出力格納場所をバインドし、SQLFetchを使って出力格納場所に結果をフェッチすることができます。fSqlTypeは、**SQL_CHAR**、**SQL_DECIMAL**、**SQL_INTEGER**等のようなSQLデータ型を取ることができます。

結果セットは以下のとおりです。

| カラムNo. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *TYPE_NAME* | **VARCHAR(128)** | **非NULL** |
| 2 | *DATA_TYPE* | **SMALLINT** | **非NULL** |
| 3 | *PRECISION* | **INTEGER** | |
| 4 | *LITERAL_PREFIX* | **VARCHAR(128)** | |
| 5 | *LITERAL_SUFFIX* | **VARCHAR(128)** | |
| 6 | *CREATE_PARAMS* | **VARCHAR(128)** | |
| 7 | *NULLABLE* | **SMALLINT** | **非NULL** |
| 8 | *CASE_SENSITIVE* | **SMALLINT** | **非NULL** |
| 9 | *SEARCHABLE* | **SMALLINT** | **非NULL** |
| 10 | *UNSIGNED_ATTRIBUTE* | **SMALLINT** | |
| 11 | *MONEY* | **SMALLINT** | **非NULL** |
| 12 | *AUTO_INCREMENT* | **SMALLINT** | |
| 13 | *LOCAL_TYPE_NAME* | **VARCHAR(128)** | |
| 14 | *MINIMUM_SCALE* | **SMALLINT** | |
| 15 | *MAXIMUM_SCALE* | **SMALLINT** | |

以下は、データソースでサポートされている全てのデータ型をフェッチするために、SQLGetTypeInfoを呼び出し、fSqlTypeの値に**SQL_ALL_TYPES**を与えています。

➲ 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

SQLGetInfoを使って、データソースについての一般情報を取得することができます。

➲ SQLGetInfoの原型:

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

fInfoTypeに与える値は、知りたい情報の種類です。rgbInfoValueの値は出力格納場所を、cbInfoValueMaxの値は格納場所のサイズを意味します。SQLGetInfoは、rgbInfoValueにフェッチした情報を返し、pcbInfoValueにフェッチした情報のサイズを返します。

➲ 例 1、データソースが文字列関数CONCATをサポートしているかどうかをチェックする:

```
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");
```

表に認められている最大カラムを知りたい場合、次のコードを使うことができます。

➲ 例 2、表に認められている最大カラム数をチェックする:

```
UWORD  maxNCol;
SDWORD retcode;

retcode = SQLGetInfo(hdbc, SQL MAX COLUMNS IN TABLE, (PTR) &maxNCol,
                     sizeof(maxNCol));

printf ("In this data source, a table can have %d columns at most\n",
        (int) maxNCol );
```

## SQLGetFunctions

SQLGetFunctionsを使って、データソースがサポートしているODBC関数をチェックすることができます。

➲ SQLGetFunctionsの原型:

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

引数fFunctionは、ODBC関数の種類を指定します。fFunctionの値は、**SQL_API_SQLCANCEL**、**SQL_API_SQLFETCH**、**SQL_PUTDATA**等です。— **SQLCancel**、**SQLFetch**、**SQLPutData**は、全てODBC関数です。

例えば、fFunctionに引数**SQL_API_SQLCANCEL**を与えると、データソースがSQLCancelをサポートしているかどうかをチェックすることができます。

ODBC関数の存在を判断するためにSQLGetFunctionsを実行した後、pfExistsのBoolean値をチェックすることができます。pfExistsは、単体のBoolean値へのポインタ、或いはBoolean値のリストです。

➲ 例 1、データソースがSQLExecDirectをサポートするかどうかをチェックする:

```
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");
```

➲ 例 2、データソースがSQLTablesをサポートするかどうかをチェックする:

```
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");
```

# 6.4 プロシージャ情報

SQLProcedureColumnsとSQLProceduresを使うと、ストアド・プロシージャ情報を取得することができます。以下の節で例とともに解説します。

## SQLProcedureColumns

SQLProcedureColumnsを使って、入力と出力パラメータについての情報を回収することができます。指定したプロシージャの結果セットを形成する定義したカラムの内容も同様です。ドライバは、結果セットとしてその情報を返します。

➲ SQLProcedureColumnsの原型:

```
RETCODE  SQLProcedureColumns (
                              HSTMT hstmt,
                              UCHAR *szProcQualifier,
                              SWORD   cbProcQualifier,
                              UCHAR  *szProcOwner,
```

```
                                SWORD   cbProcOwner,
                                UCHAR  *szProcName,
                                SWORD   cbProcName,
                                UCHAR  *szColumnName,
                                SWORD   cbColumnName)
```

SQLProcedureColumnsの引数は以下のとおりです。

- *hstmt — 回収した結果のためのステートメント・ハンドル。*
- *szProcQualifier —DBMasterではサポートしていませんので、**NULL**か空の文字列にします。*
- *cbProcQualifier —szProcQualifierの長さ。0にします。*
- *szProcOwner — 所有者へのポイント。所有者はプロシージャを作成者です。検索パターンや、**NULL**値の文字列になります。**NULL**値を使うと全所有者を表示します。*
- *cbProcOwner —szProcOwnerの長さ、又は**SQL_NTS**。*
- *szProcName — プロシージャへのポイント。検索パターンや、**NULL**値の文字列になります。**NULL**値を使うと全プロシージャを表示します。*
- *cbProcName — szProcNameの長さ、又は**SQL_NTS**。*
- *szColumnName — カラム名へのポイント。検索パターンや、**NULL**値の文字列になります。**NULL**値を使うと全カラムを表示します。*
- *cbColumnName —szColumnNameの長さ。*

*注:* *検索パターンが使用される文字列は、**szProcQualifier**、**szProcOwner**、**szProcName**. です。これら3つの引数に対応する文字列の長さの引数は、**cbProcQualifier**、**cbProcOwner**、**cbProcName**は、*SQLProcedures. です。

SQLProcedureColumnsは、次のカラムから成る結果セットを返します。

| カラムNo. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *PROCEDURE_QUALIFIER* | **VARCHAR(128)** | |

| カラムNo. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 2 | *PROCEDURE_OWNER* | **VARCHAR(128)** | |
| 3 | *PROCEDURE_NAME* | **VARCHAR(128)** | **非NULL** |
| 4 | *COLUMN_NAME* | **VARCHAR(128)** | **非NULL** |
| 5 | *COLUMN_TYPE* | **SMALLINT** | **非NULL** |
| 6 | *DATA_TYPE* | **SMALLINT** | **非NULL** |
| 7 | *TYPE_NAME* | **VARCHAR(128)** | **非NULL** |
| 8 | *PRECISION* | **INTEGER** | |
| 9 | *LENGTH* | **INTEGER** | |
| 10 | *SCALE* | **SMALLINT** | |
| 11 | *RADIX* | **SMALLINT** | |
| 12 | *NULLABLE* | **SMALLINT** | **非NULL** |
| 13 | *REMARK* | **VARCHAR(254)** | |

以下は、SQLProcedureColumnsを使って、データベースのユーザーTomのストアド・プロシージャ"employee”についての全ての情報をフェッチします。この情報は、指定したプロシージャのための結果セットを形成します。

➲ 例:

```
UCHAR  cataloq[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

SQLProceduresを使うと、データソースに保存されているプロシージャ名の一覧を取得することができます。

➲ SQLProceduresの原型:

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

hstmtはステートメント・ハンドルです。szTableQualifier、cbTableQualifier、szTableOwner、cbTableOwner、szTableName、cbTableNameは、全て**SQLProcedureColumns**と同じ定義です。SQLProceduresは、PROCEDURE_QUALIFIER、PROCEDURE_OWNER、PROCEDURE_NAMEの順に、標準結果セットとして結果を返します。

以下の表は、結果セットのカラムの一覧です。

| カラム No. | カラム名 | データ型 | コメント |
|---|---|---|---|
| 1 | *PROCEDURE_QUALIFIER* | **VARCHAR(128)** | |
| 2 | *PROCEDURE_OWNER* | **VARCHAR(128)** | |
| 3 | *PROCEDURE_NAME* | **VARCHAR(128)** | **非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** | |

次の例は、SQLProceduresを使って“Tom”によって作成されたプロシージャを全てフェッチする方法を示しています。データベースにある全プロシージャを回収する場合は、プロシージャの所有者の欄をNull値にします。

➲ 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 トランザクション制御

この章では、トランザクションとセーブポイントの概念とその特徴について説明します。また、ODBC関数を使ったトランザクションの終了方法とトランザクション制御のためのオプションの設定方法について解説します。

本章では、以下について説明します。

- *SQLSetConnectOption関数とSQLGetConnectOption関数を使って、自動コミットと手動コミットのセットと使用。*
- *SQLTransact関数を使ったトランザクションの終了。トランザクションが終了する際の影響についても説明します。*

## 7.1 トランザクションとセーブポイント

トランザクションは、作業の論理的な単位を形成する1つ以上のSQL文のシーケンスです。トランザクションの各SQL文は、作業の一部分を実行し、全部分がその作業には必要なものです。トランザクションの全SQL文が順調に実行された時のみ、その作業が完了したとみなされます。

➲ 銀行の口座で預金を管理するプログラムの例:

1. 口座名が有効かどうかを確認するために、account表に問合せます。
2. 支店番号が有効かどうかを確認するために、branch表に問合せます。
3. 出納係が存在するかどうかをチェックするためにteller表に問合せます。
4. この預金の履歴表にレコードを挿入します。

5. account表のこの口座名の残高を更新し、この預金に金額を追加します。
6. teller表のこの出納係の残高を更新します。
7. branch表にあるこの支店の残高を更新します。

*注: これら7つの操作は、1つの完成されたトランザクションを形成し、各操作は1つのSQL文です。これらの文のうち1つが失敗した場合、このトランザクション全体の実行を放棄しなければなりません。さもないと、データが不整合になります。*

➲ トランザクションの一般的なフロー:

1. トランザクション開始。
2. 命令文の実行。
3. いずれかの文が失敗した場合、変更をロールバックします。
4. 全ての文が成功した場合、変更をコミットします。

DBMasterに接続すると、トランザクションが自動的に開始し、必要なSQL文を実行することができます。これらのSQL文が処理された後、DML操作（INSERT、DELETE、UPDATE）による全変更を含むトランザクションをコミットするために、ODBC関数のSQLTransact（オプション**SQL_COMMIT**）を呼び出します。逆に、そのトランザクションをアボートする場合、ODBC関数のSQLTransact（オプション**SQL_ROLLBACK**）を呼び出します。1つのトランザクションが終了すると、DBMasterは自動的に新しいトランザクションを開始します。

トランザクションが非常に長い場合は、セーブポイントを使ってトランザクションを数パートに分けることができます。そうすることで、トランザクション全体の管理が容易になります。セーブポイントは、トランザクションの中の指定した地点で宣言することができる、論理的なマーカーです。セーブポイントを使うと、トランザクション全体をやり直す事無く、指定した地点以降の全ての更新を元に戻すことができるようになります。

例えば、15のSQL文で構成されているトランザクションを実行する場合、10番目と11番目の間にセーブポイントをマークすると、12番目の文を実行する際にエラーが発生した場合、そのセーブポイントまでロールバックすることができます。これにより、現在のトランザクションにある全ての文

を再び実行する替わりに、エラーが発生した文を直し、10番目、11番目、12番目の文を再試行するだけで済みます。

➲ 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
```

この例では、セーブポイントが長いトランザクションの管理にいかに役立つかがわかると思います。DBMasterでは、1つのトランザクションに最高で32までセーブポイントを定義することができます。トランザクションが終了すると、このトランザクションで定義した全てのセーブポイントは消去します。

セーブポイントIDは、トランザクションで一意でなければならないことに注意して下さい。例えば、***SVP1***というセーブポイント名をA地点に設定した場合、B地点に***SVP1***という名前の別のセーブポイントを設けることはで

きません。セーブポイントを使う際に覚えていなければならないもう1つの重要な事は、予め定義したセーブポイントにロールバックする時に、この地点以降に定義したセーブポイントは、全て破棄されるということです。

例えば、上記の例のC地点のセーブポイント***SVP1***にロールバックした場合、***SVP2***は破棄され、それ以降使用することはできません。但し、***SVP2***という名前の新しいセーブポイントを定義できるようになります。

## 7.2 トランザクションを終了する

前節で説明したように、SQLTransactを使ってトランザクションをコミットしたり、ロールバックしたりすることができます。

➲ SQLTransactの原型:

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

引数fTypeは、**SQL_COMMIT**、又は**SQL_ROLLBACK**のいずれかです。それらの名前に象徴されるように、**SQL_COMMIT**はトランザクションをコミットし、**SQL_ROLLBACK**はトランザクションをロールバックします。

DBMasterでは、接続オプション**SQL_CB_MODE**の値が**SQL_CB_PRESERVE** にセットされない限り、トランザクションが終了（コミット、ロールバックのいずれでも）、又はユーザーが定義したセーブポイントにロールバックすると、現在の接続ハンドルにあるステートメント・ハンドルと関係のある全ての保留の結果は、消去されます。

➲ 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);
```

トランザクションがコミットされる時、結果セットにあるhstmt1とhstmt2に関連するフェッチされていないデータは、消去されます（accountとbranch表に複数の行があると想定しています）。

## 7.3 自動／手動コミット

アプリケーション・プログラムでは一般的に、トランザクションの終了が制御できるように計画されます。この場合、手動のコミットモードが必要です。

ODBCでは多くの接続オプションを定義します。その1つが**SQL_AUTOCOMMIT**です。この接続オプションは、自動コミット・モードがONかOFFかを表示します。**SQL_AUTOCOMMIT**オプションの初期設定値はONです。これは、どのSQL文も自動的にコミットされることを意味します。

- SQLSetConnectOption関数の**SQL_AUTOCOMMIT**オプションをOFFにする:.

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

この関数呼び出しの後、ユーザーがコミット作業をコントロールします。自動コミット・モードがOFFの時に、トランザクションがコミットされない場合、ユーザーがSQLDisconnectを呼び出す際に、トランザクションはロールバックされ警告が返ります。

- SQLGetConnectOptionを使って、現在の自動コミット・モードの値を取得する:

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

optValの値が、**SQL_AUTOCOMMIT_ON**の場合、各*SQL*文は実行された後、自動的にコミットされます。

# 8 ODBC 3.0関数

DBMasterで使用しているAPI (Application Program Interface)は、現在ODBC 3.0に互換しています。ODBC 3.0 に互換するために、新たに関数が追加され、古い関数のいくつかは削除されました。その他は修正されました。結果として、その他の関数の使用を控える一方、既存の関数のいくつかは、DBMasterの全バージョンと異なるふるまいをするかもしれません。ODBC 3.0関数の完全な解説と使用方法については、「Microsoft ODBC 3.0 プログラマーリファレンス」を参照して下さい。

## 8.1 削除された関数

以下の関数や、その引数の値は、DBMaster API (ODBC 3.0)から削除されました。現在DBMasterでは、これらの関数を以前のバージョンとの互換性を保つために維持していますが、今後のバージョンでそれらが使用される保証はありません。

- *SQLAllocConnect—SQLAllocConnectは、SQLAllocHandle関数のHandleType SQL_HANDLD_DBCに替わりました。今後この関数には、SQLAllocHandleを使用します。*
- *SQLAllocEnv—SQLAllocEnvは、SQLAllocHandle関数のHandleType SQL_HANDLD_ENVに替わりました。今後この関数には、SQLAllocHandleを使用します。*
- *SQLAllocStmt—SQLAllocStmtは、SQLAllocHandle関数のHandleType SQL_HANDLD_STMTに替わりました。今後この関数には、SQLAllocHandleを使用します。*

- *SQLColAttributes—SQLColAttributesは、SQLColAttribute関数に替わりました。今後この関数には、SQLColAttruibuteを使用します。*
- *SQLExtendedFetch—SQLExtendedFetchは、SQLFetchScroll関数に替わりました。今後この関数には、SQLFetchScrollを使用します。*
- *SQLFreeConnect—SQLFreeConnectは、SQLFreeHandle関数のHandleType SQL_HANDLE_DBCに替わりました。今後この関数には、SQLFreeHandleを使用します。*
- *SQLFreeEnv—SQLFreeEnvは、SQLFreeHandle関数のHandleType SQL_HANDLE_ENVに替わりました。今後この関数には、SQLFreeHandleを使用します。*
- *SQLFreeStmt—SQLFreeStmtの引数OptionのSQL_DROP値は、SQLFreeHandle関数のHandleType SQL_HANDLE_STMTに替わりました。今後この関数には、SQLFreeHandleを使用します。*
- *SQLGetConnectOption—SQLGetConnectOptionは、SQLGetConnectAttr関数に替わりました。今後この関数には、SQLGetConnectAttrを使用します。*
- *SQLGetStmtOption-- SQLGetStmtOptionは、 SQLGetStmtAttr関数に替わりました。今後この関数には、SQLGetStmtAttrを使用します。*
- *SQLSetConnectOption—SQLSetConnectOptionは、SQLSetConnectAttr関数に替わりました。今後この関数には、SQLSetConnectAttrを使用します。*
- *SQLSetPos—SQLSetPos関数の引数fOptionのSQL_ADD値は、SQLBulkOperations関数のOperation値のSQL_ADD値に替わりました。今後この関数には、SQLBulkOperationsを使用します。*
- *SQLSetStmtOption-- SQLSetStmtOptionは、SQLSetStmtAttr関数に替わりました。今後この関数には、SQLSetStmtAttrを使用します。*
- *SQLTransact—SQLTransactは、SQLEndTranに替わりました。今後この関数には、SQLEndTranを使います。*

# 8.2 修正された関数

以下の関数が、DBMaster API (ODBC 3.0)で修正されました。これらの関数のふるまいは、DBMaster 3.01以前のAPI (ODBC 2.0)と若干異なります。但し、3.5より前のバージョンのクライアント・ソフトを使用している場合、これらの関数のふるまいは、変わりません。

## SQLCancel

SQLCancel関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。DBMaster (ODBC 2.0)の以前のバージョンでは、SQLCancel関数を呼び出す文で実行されるプロセスが無い時、SQL_CLOSEオプションでSQLFreeStmtを呼び出すことと同じ効果が得られます。DBMasterでは、SQLCancel関数を呼び出した文に実行される処理が無い場合、この関数の影響はありません。カーソルをオープンし、クローズする場合は、SQLCancelの替わりにSQLCloseCursorを呼び出します。

## SQLColumns

SQLColumns関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLColumns関数は、クライアントがODBC2.0、3.0 APIを使っているに関わらず、18カラムを戻すようになりました。次の表は、DBMasterの新旧のバージョンで、SQLColumns関数によって戻されるカラム名の一覧です。

| 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 |

| DBMASTER 3.5~4.X (ODBC 3.0) | DBMASTER 2.0X, 3.0X (ODBC 2.0) |
|---|---|
| BUFFER_LENGTH | LENGTH |
| DECIMAL_DIGITS | SCALE |
| NUM_PREC_RADIX | RADIX |
| NULLABLE | NULLABLE |
| REMARKS | — |
| COLUMN_DEF | — |
| SQL_DATA_TYPE | — |
| SQL_DATETIME_SUB | — |
| CHAR_OCTET_LENGTH | — |
| ORDINAL_POSITION | — |
| IS_NULLABLE | — |

## SQLFetch

SQLFetch関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。DBMasterでは、SQLFetch関数は、複数行の*行セット*をサポートするようになりました。DBMasterの初期のバージョンでは、SQLFetch関数では単一行の操作のみをサポートしていました。

## SQLGetData

SQLGetData関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。DBMasterでは、SQLGetData関数は、複数行の*行セット*をサポートするようになりました。DBMasterの初期のバージョンでは、SQLGetData関数では単一行の操作のみをサポートしていました。

## SQLGetFunctions

SQLGetFunctions関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。DBMasterでは、*FunctionId*パラメータの値をSQL_API_ODBC3_ALL_FUNCTIONSやSQL_API_ALL_FUNCTIONSにして

SQLGetFunctions関数をを呼び出すことができます。ODBC 2.0以前の関数をサポートを判断するために、ODBC 2.0 アプリケーションでは、SQL_API_ALL_FUNCTIONSが使用されるのに対し、ODBC 3.0以前の関数のサポートを判断するために、ODBC 3.0 アプリケーションでは、SQL_API_ODBC3_ALL_FUNCTIONSが使用されます。

*FunctionId*の値がSQL_API_ODBC3_ALL_FUNCTIONSの場合、*SupportedPtr*は、ODBC 3.0又は以前の関数がサポートされているかどうかを判断するために使用される4000-bit bitmapにポイントします。ODBC 3.0、或いは2.0ドライバいずれでも、SQL_API_ODBC3_ALL_FUNCTIONSを使用することができます。*FunctionID*の値がSQL_API_ALL_FUNCTIONSの場合、*SupportedPtr* は、ODBC 2.0関数がサポートされているかどうかを判断するために使用する100要素の配列を戻します。

## SQLGetInfo

SQLGetInfo関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。

## SQLProcedureColumns

SQLProcedureColumns関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。DBMasterのSQLProcedureColumns関数は、クライアントがODBC 2.0、又は3.0 APIを使っているかに関わらず、19カラムを戻すようになりました。次の表は、DBMasterの新旧のバージョンで、SQLProcedureColumns関数によって戻されるカラム名の一覧です。

| 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 |

| DBMaster 3.5 - 4.x (ODBC 3.0) | DBMaster 2.0x, 3.0x (ODBC 2.0) |
|---|---|
| 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 新しい関数

この節では、DBMasterの新しい関数と各関数で使用できるオプションについて、又その関数がODBC 3.0関数を完全に或いは部分的にサポートしているかどうかを説明します。

## SQLAllocHandle

SQLAllocHandle関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。これは、環境/接続/文/ディスクリプタハンドルを割り当てる総称的な関数です。ODBC 2.0関数のSQLAllocConnect、SQLAllocEnv、SQLAllocStmtに相当します。

➲ SQLAllocHandle関数の原型:

```
RETCODE SQLAllocHandle(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   InputHandle,
```

```
                    SQLHANDLE * OutputHandlePtr);
```

次の例は、SQLAllocHandle関数を使って、環境/接続/文ハンドルを割り当てます。

➲ 例:

```
SQLHANDLE  henv, hdbc, hstmt;
SQLRETURE  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関数は、DBMaster API (ODBC 3.0)で前面的にサポートされています。SQLBulkOperations関数は、バルク挿入や、ブックマーク操作による更新/削除/フェッチを含むブックマーク操作を実行します。

➲ SQLBulkOperations関数の原型:

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

次の表は、SQLBulkOperations関数のオプションのリストと、それらがサポートされているかを示しています。

| OPERATION | サポートの有無 |
|---|---|
| SQL_ADD | Y |
| SQL_UPDATE_BY_BOOKMARK | Y |
| SQL_DELETE_BY_BOOKMARK | Y |

| OPERATION | サポートの有無 |
|---|---|
| SQL_FETCH_BY_BOOKMARK | Y |

次の例は、SQLBulkOperations関数のオプションの値をSQL_ADDにして、表Employeeに2行のデータを挿入しています。

➲ 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], PhoneIbd[2];


/* set necessary statement attributes */
retCode = SQLSetStmtAttr(hstmt, SQL ATTR CURSOR TYPE,
SQL CURSOR DYNAMIC);
retcode = SQLSetStmtAttr(hstmt, SQL ATTR CONCURRENCY, SQL CONCUR LOCK);
retcode = SQLSetStmtAttr(hstmt, SQL ROW ARRAY SIZE, 2);
/* 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, "select * from Customers", SQL NTS);
/* prepare data for insertion */
CustID[0] = 1;
CustID[1] = 2;
strcpy(Name[0], "Jackson");
strcpy(Name[1], "Clinton");
strcpy(Address[0], "107 Castlewood, Cary, NC11256");
strcpy(Address[1], "305 N. Frances St., Madison, WI95868");
strcpy(Phone[0], "02-78923423");
strcpy(Phone[1], "03-7893933");
/* insert data */
```

```
retcode = SQLBulkOperations(hstmt, SQL_ADD);
…
```

## SQLCloseCursor

SQLCloseCursor関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLCloseCursor関数は、文でオープンしたカーソルを閉じ、中断している結果を廃棄します。

➲ SQLCloseCursor関数の原型:

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

## SQLColAttribute

SQLColAttribute関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLColAttribute関数は、文字列、32ビット値、或いは整数値で結果セットにカラムについての情報を戻します。

➲ SQLColAttribute関数の原型:

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

次の表は、この関数が戻すディスクリプタの種類です。

| FIELDIDENTIFIER | 目的 |
|---|---|

| FIELDIDENTIFIER | 目的 |
|---|---|
| SQL_DESC_AUTO_UNIQUE_VALUE | カラムがSERIALカラム（SQL_TRUE）か、否（SQL_FALSE）を判断します。 |
| SQL_DESC_BASE_COLUMN_NAME | 基のカラム名。 |
| SQL_DESC_BASE_TABLE_NAME | 基の表名。 |
| SQL_DESC_CASE_SENSITIVE | カラムの対照と比較が大文字と小文字を識別するか(SQL_TRUE)、否か(SQL_FALSE)を判断します。 |
| SQL_DESC_CATALOG_NAME | カラムのある表のカタログ。 |
| SQL_DESC_CONCISE_TYPE | date、time、intervalデータ型の簡単なデータ型。 |
| SQL_DESC_COUNT | 結果セットで使用できるカラムの数。 |
| SQL_DESC_DISPLAY_SIZE | カラムのデータを表示するために必要な最大文字数。 |
| SQL_DESC_FIXED_PREC_SCALE | カラムに固定長の精度と非ゼロのスケールがあるか(SQL_TRUE)、否か(SQL_FALSE)を判断します。 |
| SQL_DESC_LABEL | カラムのラベルやタイトル。カラムにラベルが無い場合、カラム名が戻ります。 |
| SQL_DESC_LENGTH | 文字列やバイナリデータ型の最大サイズや実際の文字サイズ。 |
| SQL_DESC_LITERAL_PREFIX | DBMasterがカラムに含まれるデータ型のリテラル用の接頭辞として認識することができる文字。 |
| SQL_DESC_LITERAL_SUFFIX | DBMasterがカラムに含まれるデータ型のリテラル用の接尾辞として認識することができる文字。 |
| SQL_DESC_LOCAL_TYPE_NAME | 標準名と異なる可能性のあるローカライズされたデータ型の名前（ネイティブ言語）。 |
| SQL_DESC_NAME | カラムの別名。カラムに別名が無い場合、カラム名が戻ります。 |

| FieldIdentifier | 目的 |
|---|---|
| SQL_DESC_NULLABLE | カラムにNULL値があるか(SQL_NULLABLE)、否か(SQL_NO_NULLS)を判断します。カラムにNULL値があるかどうかが不明の場合は、SQL_NULLABLE_UNKNOWNが戻ります。 |
| SQL_DESC_NUM_PREX_RADIX | SQL_DESC_TYPEが大まかなデータ型で、SQL_DESC_PRECISIONにビット数が含まれる場合、2。SQL_DESC_TYPEが正確な数のデータ型で、SQL_DESC_PRECISIONにdecimal桁の数字が含まれる場合、10。SQL_DESC_TYPEに非数値データ型が含まれる場合、0。 |
| SQL_DESC_OCTET_LENGTH | 文字列やバイナリデータ型のサイズ。 |
| SQL_DESC_PRECISION | Numericデータ型の精度。 |
| SQL_DESC_SCALE | Numericデータ型のスケール。 |
| SQL_DESC_SCHEMA_NAME | カラムのある表のスキーマ。 |
| SQL_DESC_SEARCHABLE | WHERE句でそのカラムを比較演算子と共に使用することができるか(SQL_PRED_SEARCHABLE)を判断します。(SQL_PRED_BASIC)は、LIKE以外の全比較演算子と使用します。(SQL_PRED_CHAR)は、LIKE述語とのみ使用します。(SQL_PRED_NONE)は、WHERE句で全く使用することができません。 |
| SQL_DESC_TABLE_NAME | カラムのある表の名称。 |
| SQL_DESC_TYPE | カラムのデータ型を指定するNumeric値。 |

| FIELDIDENTIFIER | 目的 |
|---|---|
| SQL_DESC_TYPE_NAME | データ型の名前を指定する文字列(データ-ソース依存)。 |
| SQL_DESC_UNNAMED | SQL_DESC_NAMEの値がカラム名/別名(SQL_DESC_NAMED)か、否か(SQL_DESC_UNNAMED)を判断します。 |
| SQL_DESC_UNSIGNED | カラムが符号無し(SQL_TRUE)か、否か(SQL_FALSE)を判断します。 |
| SQL_DESC_UPDATABLE | 結果セットにあるカラム更新されたかどうかを表します。（基の表にあるカラムではない） |

## SQLCopyDesc

SQLCopyDesc関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLCopyDesc関数は、あるディスクリプタ・ハンドルから別のものへディスクリプタ情報をコピーします。

➲ SQLCopyDesc関数の原型:

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

次の例では、PartInfo表のフィールドをBackup表にコピーするためにディスクリプタ演算が使用されます。これを行うために、hstmt1にあるIRDのフィールドをhstmt2にあるIPDのフィールドに、hstmt1にあるARDのフィールドをhstmt2にあるAPDのフィールドにコピーします。

➲ 例:

```
/* the structure of a record row */
typedef struct{
        SQLINTEGER  PartID;
        SQLINTEGER  PartIDInd;
        SQLUCHAR    Description[100];
```

```
        SQLINTEGER  DescriptionInd;
        DOUBLE      Price;
        SQLINTEGER  PriceInd;
}PartInfo;
PartInfo parts[20];
SQLHANDLE  hstmt1, hstmt2;
SQLHANDLE  ird1, ard1, ipd2, apd2;
SQLRETCODE 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, "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, arp2);
SQLCopyDesc(ird1, ipd2);
/* prepare an insert statement on hstmt2 */
SQLPrepare(hstmt2, "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関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLEndTranは、接続に関連する全文や、環境に関連する全接続にある全てのアクティブ演算をコミット、又はロールバックを実行するよう、DBMasterサーバーに要求します。DBMasterは、*CompletionType*引数に次の値をサポートしています。SQL_COMMITとSQL_ROLLBACKです。

➲ SQLEndTran関数の原型:

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

## SQLFetchScroll

SQLFetchScroll関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLFetchScroll関数は、相関位置、或いは絶対位置で、ブックマークによって、或いは結果セットからデータの行セットをフェッチし、バインドされた全カラムを戻します。DBMasterでは、*FetchOrientation*引数に次の値をサポートしています。SQL_FETCH_NEXT、SQL_FETCH_PRIOR、SQL_FETCH_FIRST、SQL_FETCH_LAST、SQL_FETCH_ABSOLUTE、SQL_FETCH_BOOKMARK、SQL_FETCH_RELATIVE.です。

➲ SQLFetchScroll関数の原型:

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

次のコードの一部は、結果セット全体をフェッチするために、オプションがSQL_NEXTのSQLFetchScroll関数をどのように使用するかを示しています。行の状態を取得するために、SQLExtendedFetch関数と異なる、オプションSQL_ATTR_ROW_STATUS_PTRのSQLSetStmtAttr関数を使用します。

➲ 例:

```
#define LENGTH 18
#define ROWSET SIZE 5
SQLHandle hstmt;
RETCODE   retcode;
SQLUCHAR  empid[ROWSET SIZE][LENGTH]
SQLUCHAR  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 ROW ARRAY SIZE, ROWSET SIZE);
/* execute a statement */
SQLExecDirect(hstmt, "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関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLForeignKeys関数は、特定の表に外部キーのリストを、或いは特定の表にある主キーを参照する別の表にある外部キーのリストを戻します。

➲ SQLForeignKeys関数の原型:

```
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);
```

この例には、2つの表ORDER (ORDERID, CUSTID, OPENDATE)とCUSTOMER (CUSTID, NAME, ADDRESS, PHONE)があります。

*ORDER*表には、CUSTIDは販売した顧客を識別します。外部キーは、CUSTOMER表のORDERIDを参照します。

この例は、ORDER表の主キーを参照する他の表にある外部キーを取得するためにSQLForeignKeysを呼び出します。

➲ 例:

```
#define TAB LEN 18
#define COL LEN 18

SQLUCHAR     pkTable[TAB LEN+1], fkTable[TAB LEN+1];
SQLUCHAR     pkCol[COL LEN+1],   fkCol[COL LEN+1];
SQLHANDLE    hstmt;
SQLINTEGER   pkTableInd, fkTableInd, pkColInd, fkColInd;
SQLRETCODE   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);
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, "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関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLFreeHandle関数は、SQLAllocHandle関数で割り当てた環境/接続/文/ディスクリプタ・ハンドルに関連するリソースを解放します。

➲ SQLFreeHandle関数の原型:

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

次のコードの一部は、SQLFreeHandle関数をどのように使用して環境ハンドルを解放するのかを示しています。

➲ 例:

```
SQLHANDLE   henv;
SQLAllocHandle(SQL HANDLE ENV, SQL NULL HANDLE, &henv);
…
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

## SQLGetConnectAttr

SQLGetConnectAttr関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLSetConnectAttr関数は、データベース接続の属性を取得します。この関数は、ODBC 2.0のSQLGetConnectOption関数に相当します。

➲ SQLGetConnectAttr関数の原型:

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

次の表は、この関数を使って回収することができる属性と、DBMasterがサポートしているかどうかを表しています。

| 属性 | サポートの有無 |
|---|---|
| SQL_ATTR_ACCESS_MODE | Y |
| SQL_ATTR_ASYNC_ENABLE | Y |
| SQL_ATTR_AUTO_IPD | Y |
| SQL_ATTR_AUTOCOMMIT | Y |

| 属性 | サポートの有無 |
|---|---|
| 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関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLGetDescField関数は、ディスクリプタ・レコードの単一フィールドの値を取得します。

➲ SQLGetDescField関数の原型:

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

次の表は、DBMasterでこの関数を使って回収することができるディスクリプタ・フィールドを表しています。表の「G」はGet、「S」はSet、「I」は無効を意味します。SQL_DESC_ARRAY_SIZEは、1つの値のみサポートします。

| FIELDIDENTIFIER | 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 |

| FIELDIDENTIFIER | ARD | APD | IRD | IPD |
|---|---|---|---|---|
| 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 |
| 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 |

次のコードの一部は、カラム情報を取得するために、SQLGetDescFieldをどのように使用するかを示しています。

➲ 例:

```
#define LEN 19
SQLHANDLE   hstmt, ird;
SQLINTEGER  count, index;
SQLUCHAR    colName[LEN], typeName[LEN];
SQLSMALLINT prec, scale;
SQLUINTEGER length;

/* execute the statement */
SQLExecDirect(hstmt, "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, &len, 0, NULL);
        printf("Column No : %d, Name : %s, Type : %s, Length : %d,
Precision : %d, Scale : %d \n", index, colName, typeName, len, prec,
scale);
}
…
```

## SQLGetDescRec

SQLGetDescRec関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLGetDescRec関数は、ディスクリプタ・レコードにある複数のフィールドの設定や値を戻します。

➲ SQLGetDescRec関数の原型:

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

次のコードの一部は、カラム情報を取得するために、SQLGetDescRec関数をどのように使用するかを示しています。

➲ 例:

```
#define LEN 19
SQLHANDLE   hstmt, ird;
SQLINTEGER  count, index;
SQLUCHAR    colName[LEN];
SQLSMALLINT type, prec, scale, nullable;
SQLUINTEGER length;

/* execute the statement */
SQLExecDirect(hstmt, "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, &len,
&prec, &scale, &nullable);
        printf("Column No. : %d, Name : %s, Type : %d, Length : %d,
Precision : %d, Scale : %d, Nullable : %d \n", index, colName, type,
len, prec, scale, null);
}
…
```

## SQLGetDiagField

SQLGetDiagField関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLGetDiagField関数は、特定の診断データ構造にあるレコードから単一フィールドの現在の値を戻します。このフィールドには、エラー、警告、状態の情報があります。

➲ SQLGetDiagField関数の原型:

```
RETCODE SQLGetDiagField(
                    SQLSMALLINT   HandleType,
```

```
                    SQLHANDLE     Handle,
                    SQLSMALLINT   RecNumber,
                    SQLSMALLINT   DiagIdentifier,
                    SQLPOINTER    DiagInfoPtr,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  StringLengthPtr);
```

次の表は、診断データ構造から要求されるフィールドの識別子と、DBMasterがポートしているかどうかを表しています。

| DIAGIDENTIFIER | サポートの有無 |
|---|---|
| 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 |

次のコードの一部は、エラー情報を取得するために、SQLGetDiagField関数とSQLGetDiagRec関数をどのように使用するかを示しています。

➲ 例:

```
SQLHANDLE   hstmt;
SQLRETCODE  retcode;
SQLINTEGER  num, index, nativerc;
SQLUCHAR    state[20], errmsg[200];
SQLSMALLINT retLen;
```

```
/* execute a statement */
retcode = SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL NTS);
/* if error, get error info */
if (retcode != SQL SUCCESS){
        SQLGetDiagField(SQL HANDLE STMT, hstmt, 0, SQL DIAG NUM, &num,
0, NULL);
        for (index = 1; index <= num; index++){
             SQLGetDiagRec(SQL HANDLE STMT, hstmt, index, state,
&nativerc, errmsg, 200, &retLen);
        }
}
….
```

## SQLGetDiagRec

SQLGetDiagRec関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLGetDiagRec関数は、SQLSTATE、ネイティブ・エラー・コード、診断メッセージ・テキストを含む、よく使用される診断レコードの様々なフィールドの値を戻します。DBMasterでは、*HandleType*引数に次の値をサポートしています。SQL_HANDLE_ENV、SQL_HANDLE_DBC、SQL_HANDLE_STMT、SQL_HANDLE_DESCです。

➲ SQLGetDiagRec関数の原型:

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

## SQLGetEnvAttr

SQLGetEnvAttr関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLGetEnvAttr関数は、環境ハンドルの属性を取得します。

➲ SQLGetEnvAttr関数の原型:

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

次の表は、この関数を使って回収することができる属性と、DBMasterがサポートしているかどうかを表しています。

| 属性 | サポートの有無 |
|---|---|
| SQL_ATTR_ODBC_VERSION | Y |
| SQL_ATTR_OUTPUT_NTS | Y |
| SQL_ATTR_CONNECTION_POOLING | N |
| SQL_ATTR_CP_MATCH | N |

## SQLGetStmtAttr

SQLGetStmtAttr関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLGetStmtAttr関数は、文の属性をセットします。この関数は、ODBC 2.0のSQLGetStatementOption関数に相当します。

➲ SQLGetStmtAttr関数の原型:

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

次の表は、この関数を使って回収することができる属性と、DBMasterがサポートしているかどうかを表しています。

| 属性 | サポートの有無 |
|---|---|
| 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関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLPrimaryKeys関数は、結果セットに表の主キーを形成するカラム名を戻します。

➲ SQLPrimaryKeys関数の原型:

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

この例の表CUSTOMERにはカラム（CUSTID、NAME、ADDRESS、PHONE）があり、CUSTOMER表の主キーはCUSTIDです。

この例では、CUSTOMER表の主キー情報を取得するために、SQLPrimaryKeysを呼び出しています。.

➲ 例:

```
#define TAB LEN 19
#define COL LEN 19

SQLUCHAR     pkTable[TAB LEN], fkTable[TAB LEN];
SQLHANDLE    hstmt;
SQLINTEGER   pkTableInd, pkColInd;
SQLRETCODE   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, "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関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLSetConnectAttr関数は、データベース接続の属性をセットします。この関数は、ODBC 2.0のSQLSetConnectOption関数に相当します。

➲ SQLSetConnectAttr関数の原型 :

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

次の表は、この関数を使ってセットされる属性と、DBMasterがサポートしているかどうかを表しています。

以下の属性は次のように適用されます。SQL_ATTR_ASYNC_ENABLEは、SQL_ASYNC_ENABLE_OFFのみサポートします。SQL_ATTR_CONNECTION_TIMEOUTは、0の値（タイムアウト無し）のみサポートします。SQL_ATTR_METADATA_IDは、SQL_FALSEのみサポートします。SQL_ATTR_MAX_ROWSは、0の値（全ての行）のみサポートします。

| 属性 | サポートの有無 |
|---|---|
| SQL_ATTR_ACCESS_MODE | Y |
| SQL_ATTR_ASYNC_ENABLE | Y |

| 属性 | サポートの有無 |
|---|---|
| SQL_ATTR_AUTO_IPD | Y |
| SQL_ATTR_AUTOCOMMIT | Y |
| 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 |

## SQLSetDescField

SQLSetDescField関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLSetDescField関数は、ディスクリプタ・レコードの単一のフィールドの値をセットします。

➲ SQLSetDescField関数の原型:

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

次の表は、DBMasterでこの関数を使ってセットされるディスクリプタ・フィールドを表してします。表の「G」はGet、「S」はSet、「I」は無効を意味します。SQL_DESC_ARRAY_SIZEは、1つの値のみサポートします。

| FIELDIDENTIFIER | 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 |
| SQL_DESC_TYPE | G/S | G/S | G | G/S |

| FIELDIDENTIFIER | ARD | APD | IRD | IPD |
|---|---|---|---|---|
| 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 |

次のコードの一部は、SQLSetDescField関数で一般的に使用されるオプションを示しています。プログラムを単純にするために、表EMPLOYEEには、2つのカラム（NAME, SALARY）しかありません。

➲ 例:

```
SQLHANDLE   hstmt, ard;
SQLSMALLINT status[2];
SQLUCHAR    name[2][18];
Float       salary[2];
SQLRETCODE  retcode;
int         i;
/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL ATTR APP ROW DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL NTS);
/* set the row size, status and the binding type */
SQLSetDescField(ard, 0, SQL DESC ARRAY SIZE, 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, SQL C CHAR, 0);
SQLSetDescField(ard, 1, SQL DESC LENGTH, 18, 0);
SQLSetDescField(ard, 1, SQL DESC DATA PTR, name, 18);
SQLSetDescField(ard, 2, SQL DESC CONCISE TYPE, SQL C FLOAT, 0);
SQLSetDescField(ard, 2, SQL DESC DATA PTR, salary, 0);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL SUCCESS) {
        for (i = 0; i <2; i++)
```

```
        {
        printf("Employee Name : %s, Salary : %f \n", name[i],
salary[i]);
        }
        retcode = SQLFetch(hstmt);
}
…
```

## SQLSetDescRec

SQLSetDescRec関数は、DBMaster API (ODBC 3.0)で完全にサポートされています。SQLSetDescRec関数は、データ型、又はカラムやパラメータにバインドされるバッファに影響する複数のディスクリプタ・フィールドの値をセットします。

➲ SQLSetDescRec関数の原型:

```
RETCODE SQLSetDescField(
                    SQLHDESC     DescriptorHandle,
                    SQLSMALLINT  RecNumber,
                    SQLSMALLINT  Type,
                    SQLSMALLINT  SubType,
                    SQLINTEGER   Length,
                    SQLSMALLINT  Precision,
                    SQLSMALLINT  Scale,
                    SQLPOINTER   DataPtr,
                    SQLINTEGER * StringLengthPtr,
                    SQLINTEGER * IndicatorPtr);
```

次のコードの一部は、カラムをバインドするために、SQLSetDescRec関数をどのように使用するかを示しています。プログラムを単純にするために、表EMPLOYEEには、2つのカラム（NAME, SALARY）しかありません。

➲ 例:

```

SQLHANDLE   hstmt, ard;
```

```
SQLSMALLINT status[2];
SQLUCHAR   name[2][19];
SQLINTEGER nameInd[2];
Float      salary[2];
SQLRETCODE retcode;
int        i;
/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL ATTR APP ROW DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL NTS);
/* set the row size, status and the binding type */
SQLSetDescField(ard, 0, SQL DESC ARRAY SIZE, 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, 0, 0, 0, name, 19, nameInd);
SQLSetDescRec(ard, 2, SQL C FLOAT, 0, 0, 0,, salary, 0, NULL);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL SUCCESS) {
for (i = 0; i < 2; i++)
        {
        printf("Employee Name : %s, Salary : %f \n", name[i],
salary[i]);
        }
        retcode = SQLFetch(hstmt);
}
…
```

## SQLSetEnvAttr

SQLSetEnvAttr関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLSetEnvAttr関数は、環境ハンドルの属性をセットします。

➲ SQLSetEnvAttr関数の原型:

```
RETCODE SQLSetEnvAttr(
                    SQLHENV     EnvironmentHandle,
                    SQLINTEGER  Attribute,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  StringLength);
```

次の表は、この関数を使ってセットされる属性と、DBMasterがサポートしているかどうかを表しています。SQL_ATTR_OUTPUT_NTSは、SQL_TRUE（常にNULL終端）のみをサポートしています。

| 属性 | サポートの有無 |
|---|---|
| SQL_ATTR_ODBC_VERSION | Y |
| SQL_ATTR_OUTPUT_NTS | Y |
| SQL_ATTR_CONNECTION_POOLING | N |
| SQL_ATTR_CP_MATCH | N |

## SQLSetStmtAttr

SQLSetStmtAttr関数は、DBMaster API (ODBC 3.0)で部分的にサポートされています。SQLSetStmtAttr関数は、文の属性をセットします。この関数は、ODBC 2.0のSQLSetStatementOption関数に相当します。

➲ SQLSetStmtAttr関数の原型:

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

次の表は、この関数を使ってセットされる属性と、DBMasterがサポートしているかどうかを表しています。SQL_ATTR_ASYNC_ENABLEとSQL_ATTR_PARAMS_PROCESSED_PTRは、SQL_ASYNC_ENABLE_OFFのみをサポートしています。SQL_ATTR_CONCURRENCYは、SQL_CONCUR_READ_ONLYとSQL_CONCUR_LOCKのみをサポートしています。SQL_ATTR_CURSOR_SENSITIVITYは、SQL_UNSPECIFIEDのみ

をサポートしています。SQL_ATTR_KEYSET_SIZEと
SQL_ATTR_MAX_LENGTHは、0の値のみをサポートしています。
SQL_ATTR_METADATA_IDは、SQL_FALSEのみをサポートしています。
SQL_ATTR_NOSCANは、SQL_NOSCAN_ONのみをサポートしています。
SQL_ATTR_PARAMSET_SIZEは、1つの値のみサポートしています。
SQL_ATTR_PARAMSET_SIZEは、1つの値のみサポートしています。
SQL_ATTR_QUERY_TIMEOUTは、0の値のみをサポートしています。
SQL_ATTR_RETRIEVE_DATAは、SQL_RD_ONのみサポートしています。SQL_ATTR_SIMULATE_CURSORは、SQL_SC_UNIQUEのみサポートしています。

| 属性 | サポートの有無 |
|---|---|
| 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 |
| 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 |

| 属性 | サポートの有無 |
|---|---|
| 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 |

# 9 Unicodeサポート

DBMasterは現在ネイティブのunicodeデータを支援します。したがって、ユーザは、データベース中に多言語から成るデータを格納し、処理することができます。しかしながら、多言語のデータはunicodeデータとして渡されなくてはなりません。

NCHAR、NVARCHARおよびNCLOBのデータ型は、unicodeデータの格納を支援するために供給されています。更に、たくさんのunicode関数も支援されています。

## 9.1 Unicodeエンコーディング・インターフェイス

ODBC機能から渡された入力データはエンコード化されたUTF-16LEあるいはUTF-8でありえます。入力Unicodeエンコーディング規則を指定する接続オプションを使用してもかまいません。デフォルトのエンコーディングはUTF-16LEです。あなたがUTF-8にこのオプションをセットした時、DBMasterは入力ストリングがすべてエンコーディングUTF-8であると仮定します。また、Unicodeストリングは UTF8として出力されます。Windowsアプリケーション、およびVisual Basicのような開発ツールはUTF-16LEエンコーディングを使用しています；あなたのAPがUTF8(ほとんどのUnixアプリケーション)を使用する場合は、あなたのプログラム中のODBC接続設定オプションを呼び出すことにより、接続オプションをセットしてください。

ストリング・データを入力し出力するため、DBMasterは2のunicodeエンコーディング型、UTF-8およびUTF-16を支援しています。入力/出力unicodeストリング・エンコーディング型をセットするために、関数SQLSetConnectAttrのオプションSQL_CLI_UCODE_TYPEの値をSQL_CLI_UTYPE_UTF16か、SQL_CLI_UTYPE_UTF8にセットして呼び出すことができます。

SQL_CLI_UCODE_TYPEの初期設定値はSQL_CLI_UTYPE_UTF16 (UTF-16LE)です。

## Unicode関数：

DBMasterは次のODBC Unicode関数を支援しています。

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

関連するUnicode ODBCタイプ

Unicode支援については、DBMasterは次のデータ型を支援します：CのためのSQL_C_WCHAR、SQL のためのSQL_WCHAR、SQL_WVARCHAR、そしてSQL_WLONGVARCHAR。これらのデータ・タイプは、SQLBindCol、SQLBindParameter、SQLGetData、SQLPutData等のような関連するODBC関数のために支援されています。

SQL_C_WCHARのための入力パラメーター/出力カラムの長さはバイトで指定されます。SQL_C_WCHARの精度はキャラクタの中で指定されます。

# A 関数シーケンスの比較

この付録は、DBMaster ODBC APIとMicrosoft ODBC 3.0 API間の関数シーケンスの違いを列記しています。

## A.1 SQLRowCount

SQLRowCountは、状態S1、S2、S3で問題無く呼ぶことができます。DBMasterではエラーになりません。ODBC 3.0ではエラーS1010が発生します。

## A.2 SQLGetCursorName

SQLSetCursorName関数でカーソル名を設定する前に、SQLGetCursorName関数を呼び出しても、DBMasterはエラーS1015を返しません。これは、DBMasterがステートメント・ハンドルを割り当てる際に、自動的にカーソル名を生成するためです。

# B 関数プロパティの比較

この付録は、DBMasterのODBC APIとMicrosoft ODBC 3.0 API間の関数プロパティの違いを紹介します。これらの違いには、標準ODBC関数と若干異なるふるまいを持つ、DBMasterで使用できる関数や文のために役立つ拡張オプションが含まれます。

## B.1 SQLPutData

ODBC 3.0では、SQLPutData関数を使って、CHAR、BINARY、LONG VARCHAR、LONG VARBINARYのデータ型のカラムにデータを部分的に送信することができます。一方DBMasterでは、SQLPutData関数で使用できるデータ型はLONG VARCHARとLONG VARBINARYに限られています。

## B.2 SQLColumns

DBMasterでは、SQLColumns関数を使って一時表から情報を回収することはできません。

## B.3 SQLTables

DBMasterでは、SQLTables関数を使って一時表から情報を回収することはできません。

## B.4 SQLDriverConnect

SQLDriverConnect関数では、プロンプト・フラグSQL_DRIVER_PROMPTとSQL_DRIVER_COMPLETE_REQUIREDのふるまいは、プロンプト・フラグSQL_DRIVER_COMPLETEのふるまいと同じです。プロンプトのふるまいは、SQLDriverConnectの**fDriverComplete**引数でコントロールします。

## B.5 SQLBindParameter

SQLBindParameter関数では、引数**pcbValue**がSQL_LEN_DATA_AT_EXEC(長さ)にセットされた時のふるまいは、引数**pcbValue**がSQL_DATA_AT_EXECにセットされた時のふるまいと同じです。SQL_LEN_DATA_AT_EXECの長さの値は、無視されます。

## B.6 位置付けDELETE/UPDATE

位置付けDELETEや、位置付けUPDATEに含まれる*SQL*文は、通常カーソル名を参照します。カーソル名が示すカーソルがオープンしていない場合、準備時ではなく実行時にDBMasterはそれを指摘し、エラー34000が返ります。

## B.7 SQLSetConnectOption

SQLSetConnectOption関数のいくつかの拡張オプションは、DBMaster特有のものです。DBMasterのデータソースを使う際に、これらの高度な接続オプションをセットすることができます。

拡張オプションは以下のとおりです。

| オプション | 解説 | 許容値 |
|---|---|---|
| SQL_TXN_ ISOLATION | 現在の接続のトランザクション隔離レベ | SQL_TXN_REPEATABLE_ READ<br>SQL_TXN_READ_ UNCOMMITTED |

| オプション | 解説 | 許容値 |
|---|---|---|
| | ルをセット | SQL_TXN_SERIALIZEABLE<br>SQL_TXN_FORCE_READ_<br>UNCOMMITTED |
| SQL_DB_<br>MODE | 現在接続しているデータベースをシングル/マルチユーザー・バージョンにセット | SQL_SINGLE<br>SQL_MULTI |
| SQL_<br>JOURNAL_<br>MODE | データベースのジャーナル書き込みをON/OFFにセット | SQL_JOURNAL_ON<br>SQL_JOURNAL_OFF |
| SQL_LOCK_T<br>IMEOUT | ロックを待機する秒数をセット | 秒数 |
| SQL_<br>BACKUP_<br>MODE | 使用するバックアップの種類をデータベースに通知 | SQL_BACKUP_DATA<br>SQL_BACKUP_BLOB<br>SQL_BACKUP_OFF |
| SQL_DATE_<br>INPUT_<br>FORMAT | 回収するデータ形式をデータベースに通知 | SQL_DATE_FORMAT_0(mm/dd/yy)<br>SQL_DATE_FORMAT_1(mm-dd/yy)<br>SQL_DATE_FORMAT_2(dd/mon/yy)<br>SQL_DATE_FORMAT_3(dd-mon-yy)<br>SQL_DATE_FORMAT_4(mm/dd/yyyy)<br>SQL_DATE_FORMAT_5(mm-dd-yyyy)<br>SQL_DATE_FORMAT_6(yyyy/mm/dd)<br>SQL_DATE_FORMAT_7(yyyy-mm-dd)<br>SQL_DATE_FORMAT_8(dd/mon/yyyy)<br>SQL_DATE_FORMAT_9(dd-mon-yyyy)<br>SQL_DATE_FORMAT_10(dd.mm.yyyy)<br>SQL_DATE_FORMAT_11(yy/mm/dd)<br>SQL_DATE_FORMAT_12(yy-mm-dd)<br>SQL_DATE_IN_DEFAULT |

| オプション | 解説 | 許容値 |
|---|---|---|
| SQL_DATE_OUTPUT_FORMAT | 日付データ型を返す時に使用するデータ形式をデータベースに通知 | SQL_DATE_FORMAT_0(mm/dd/yy)<br>SQL_DATE_FORMAT_1(mm-dd/yy)<br>SQL_DATE_FORMAT_2(dd/mon/yy)<br>SQL_DATE_FORMAT_3(dd-mon-yy)<br>SQL_DATE_FORMAT_4(mm/dd/yyyy)<br>SQL_DATE_FORMAT_5(mm-dd-yyyy)<br>SQL_DATE_FORMAT_6(yyyy/mm/dd)<br>SQL_DATE_FORMAT_7(yyyy-mm-dd)<br>SQL_DATE_FORMAT_8(dd/mon/yyyy)<br>SQL_DATE_FORMAT_9(dd-mon-yyyy)<br>SQL_DATE_FORMAT_10(dd.mm.yyyy)<br>SQL_DATE_FORMAT_11(yy/mm/dd)<br>SQL_DATE_FORMAT_12(yy-mm-dd)<br>SQL_DATE_OUT_DEFAULT |
| SQL_TIME_INPUT_FORMAT | 回収する時間形式をデータベースに通知 | SQL_DATE_FORMAT_0(hh:mm:ss.fff)<br>SQL_DATE_FORMAT_1(hh:mm:ss)<br>SQL_DATE_FORMAT_2(hh:mm)<br>SQL_DATE_FORMAT_3(hh)<br>SQL_DATE_FORMAT_4(hh:mm:ss.fff tt)<br>SQL_DATE_FORMAT_5(hh:mm:ss tt)<br>SQL_DATE_FORMAT_6(hh:mm tt)<br>SQL_DATE_FORMAT_7(hh tt)<br>SQL_DATE_FORMAT_8(tt hh:mm:ss.fff)<br>SQL_DATE_FORMAT_9(tt hh:mm:ss)<br>SQL_DATE_FORMAT_10(tt hh:mm)<br>SQL_DATE_FORMAT_11(tt hh)<br>SQL_TIME_IN_DEFAULT |

| オプション | 解説 | 許容値 |
|---|---|---|
| SQL_TIME_OUTPUT_FORMAT | 時間データ型を返す時に使用する時間形式をデータベースに通知 | SQL_DATE_FORMAT_0(hh:mm:ss.fff)<br>SQL_DATE_FORMAT_1(hh:mm:ss)<br>SQL_DATE_FORMAT_2(hh:mm)<br>SQL_DATE_FORMAT_3(hh)<br>SQL_DATE_FORMAT_4(hh:mm:ss.fff tt)<br>SQL_DATE_FORMAT_5(hh:mm:ss tt)<br>SQL_DATE_FORMAT_6(hh:mm tt)<br>SQL_DATE_FORMAT_7(hh tt)<br>SQL_DATE_FORMAT_8(tt hh:mm:ss.fff)<br>SQL_DATE_FORMAT_9(tt hh:mm:ss)<br>SQL_DATE_FORMAT_10(tt hh:mm)<br>SQL_DATE_FORMAT_11(tt hh)<br>SQL_TIME_OUT_DEFAULT |
| SQL_SYSINFO_CLEAR | システム情報を消去するかどうかをデータベースに確認 | 無し |
| SQL_CB_MODE | COMMIT/ROLLBACK操作で、接続のカーソルに与える影響を表示 | SQL_CB_CLOSE<br>SQL_CB_RESERVER<br>SQL_CB_DELETE |

## B.8 SQLGetConnectOption

SQLGetConnectOption関数のいくつかの拡張オプションは、DBMaster特有のものです。これらの高度な接続オプションで、DBMasterのデータソースを使う際に情報を取得することができます。

拡張オプションは、以下のとおりです。

| オプション | 解説 |
|---|---|
| SQL_TXN_ISOLATION | 現在のhdbcのトランザクション隔離レベルを取得 |
| SQL_DB_MODE | 現在のデータベース・モードを取得 |
| SQL_JOURNAL_MODE | 現在のデータベース・ジャーナル書き込みモードを取得 |
| SQL_LOCK_TIMEOUT | ロックを待機する秒数を取得 |
| SQL_BACKUP_MODE | 現在のデータベース・バックアップ・モードを取得 |
| SQL_DATE_INPUT_FORMAT | 現在の日付データ型の入力フォーマットを取得 |
| SQL_DATE_OUTPUT_FORMAT | 現在の日付データ型の出力フォーマットを取得 |
| SQL_TIME_INPUT_FORMAT | 現在の時間データ型の入力フォーマットを取得 |
| SQL_TIME_OUTPUT_FORMAT | 現在の時間データ型の出力フォーマットを取得 |
| SQL_CB_MODE | COMMIT/ROLLBACK操作が接続のカーソルに与える影響を取得 |
| SQL_CONNECT_ID | 現在の接続IDを取得 |

# C Microsoft ODBC 3.0 プログラマーズリファレンスの誤記述

この付録は、Microsoft ODBC 3.0 プログラマーズリファレンスの記述ミスのリストです。

## C.1 SQLParamData

現在の状態がS8の場合、SQLParamDataを呼び出した後、ドライバはODBC 3.0で定義されたSQL_SUCCESSでは無く、SQL_NEED_DATAを返します。

## C.2 SQLPrepare

現在の状態がS2の場合、結果セットが空であると予想される場合SQLPrepareを呼び出した後、その状態はS3に遷移するはずです。

現在の状態がS3の場合、SQLPrepareを呼び出した後に結果セットが生成されない場合、その状態はS2に遷移するはずです。

# D　データ型

ドライバは、データソース特有のSQLデータ型をODBC SQLデータ型やドライバ特有のSQLデータ型にマップします。各SQL データ型は、ODBC C のデータ型に対応します。アプリケーションでは、SQLBindColや SQLGetData、或いはSQLBindParameterで、**fCType**引数と共に正しいCのデータ型を指定します。データをデータソースに送信する前に、ドライバは指定したCのデータ型から変換し、データソースからデータを回収する前に、ドライバは指定したCのデータ型に変換します。

この付録では、以下のトピックについて解説します。

- *ODBC SQLのデータ型*
- *ODBC C のデータ型*
- *ODBCの初期設定のCデータ型*
- *SQLのデータ型の精度、スケール、サイズ、表示サイズ*
- *データ型の変換*

# D.1 ODBC SQLのデータ型

以下の表は、ODBC SQLのデータ型(**fSqlType**の欄)とそれに対応するDBMasterのSQLのデータ型(SQL データ型の欄)のマッピングのリストです。データ型の詳細も合わせて説明します。

| FSQLTYPE | SQLデータ型 | 説明 |
|---|---|---|
| **SQL_CHAR** | **CHAR(n)** | 固定長nの文字列(1 <= n <= 3992) |
| **SQL_VARCHAR** | **VARCHAR(n)** | 最大長nの可変長文字列(1 <= n <= 3992) |
| **SQL_LONGVARCHAR** | **LONG VARCHAR** | 最大長2GBの可変長文字 |
| **SQL_DECIMAL** | **DECIMAL(p,s)** [1]<br>**DECIMAL(p)**<br>**DECIMAL** | 精度pとスケールs、符号付の正確な数値<br>(1<= p <= 17; 0 <= s <= p)<br>(6<= p <= 17; s = 6)<br>(p = 17; s = 6) |
| **SQL_SMALLINT** | **SMALLINT** | 精度5とスケール0、符号付の正確な数値<br>(32,768 <= n <= 32,767) |
| **SQL_INTEGER** | **INTEGER** | 精度10とスケール0、符号付の正確な数値<br>($-2^{31}$ <= n <= $2^{31}$ -1) |
| **SQL_REAL**[2] | **FLOAT** | 仮数部の精度7、符号付の概数 ($10^{-38}$～$10^{38}$) |
| **SQL_FLOAT** | **DOUBLE** | 仮数部の精度15、符号付の概数 (10^-308～10^308) |
| **SQL_DOUBLE** | **DOUBLE** | 仮数部の精度15、符号付の概数 (10^-308～10^308) |
| **SQL_FILE**[3] | **FILE** | DBMaster特有のデータ型。**FILE**カラムのデータを外部ファイルとして保存。 |

[1] p (精度)は合計桁数、s (スケール)は小数点の右側の桁数です。

[2] このデータ型の定義は、DBMasterとODBC 2.0で僅かに異なります。

[3] DBMasterでサポートしている拡張データ型は、ODBC 2.0には含まれていません。

| FSQLTYPE | SQLデータ型 | 説明 |
|---|---|---|
| **SQL_BINARY** | **BINARY(n)** | 固定長nのバイナリ・データ (1 <= n <= 3992) |
| **SQL_LONGVARBINARY** | **LONG VARBINARY** | 最大長2GBの可変長バイナリ・データ |
| **SQL_DATE** | **DATE** | 日付データ |
| **SQL_TIME** | **TIME** | 時間データ |
| **SQL_TIMESTAMP** | **TIMESTAMP** | 日付と時間両方を含むデータ |
| **SQL_WCHAR** | **WCHAR(n)** | 固定ストリング最大長 *n* のUnicode文字ストリング（1<=*n*<=1996） |
| **SQL_WLONGVARCHAR** | **WLONGVARCHAR** | 最大長2GBの可変長Unicode文字データ（文字最大長 1GB） |
| **SQL_WVARCHAR** | **WVARCHAR(n)** | ストリング最大長*n*の可変長Unicode文字ストリング（1<=*n*<=1996） |

## D.2 ODBCのCデータ型

ODBCのCデータ型は、アプリケーションでデータを保存するために使用します。引数**fCType**があるSQLBindParameter、SQLBindCol、SQLGetData関数の中で指定されます。以下の表は、fCTypeの有効値、各fCType値を取るODBCのCデータ型、WindowsとUNIX環境に対応するCデータ型の一覧です。

| FCTYPE | ODBC C Typedef | Cデータ型 | バイト |
|---|---|---|---|
| **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** | **SQL_C_SLONG**と同様 | | |

| FCTYPE | ODBC C Typedef | Cデータ型 | バイト |
|---|---|---|---|
| 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 {<br>SWORD year;[1]<br>UWORD month;[2]<br>UWORD day;[3]<br>} | 6 |
| SQL_C_TIME | TIME_STRUCT | Struct tagTIME_STRUCT {<br>UWORD hour;[4]<br>UWORD minutes;[5]<br>UWORD second[6];<br>} | 6 |
| SQL_C_TIMESTAMP | TIMESTAMP_STRUCT | struct tagTIMESTAMP_STRUCT<br>{<br>SWORD year;[1]<br>UWORD month;[2]<br>UWORD day;[3]<br>UWORD hour;[4]<br>UWORD minutes;[5] | 16 |

[1]yearの有効値の範囲は、1～9999です。

[2]monthの有効値の範囲は、1～12です。

[3] dayの有効値の範囲は、1～その月の日数です。

[4] hourの有効値の範囲は、0～23です。

[5] minuteの有効値の範囲は、0～59です。

[6] secondの有効値の範囲は、0～59です。

| FCTYPE | ODBC C Typedef | Cデータ型 | バイト |
|---|---|---|---|
| | | **UWORD** second;[6]<br>**UDWORD** fraction[1];<br>} | |
| **SQL_C_WCHAR** | **SQLWCHAR FAR*** | **unsigned short FAR*** | 4 |
| **SQL_C_BOOKMARK** | **UDWORD** | **unsigned long int** | 4 |
| **SQL_C_DEFAULT** | 次節の「初期設定のCデータ型」を参照して下さい。 | | |

# D.3 ODBCの初期設定のCデータ型

SQLBindCol、SQLGetData、SQLBindParameterの引数fCTypeに**SQL_C_DEFAULT**を指定した場合、出力/入力バッファのCデータ型は、バッファがバインドされるカラム又はパラメータのSQLデータ型に対応しているとドライバでは想定します。以下の表は、各ODBC SQLのデータ型に対する初期設定のCデータ型を示しています。

| SQLデータ型 | 初期設定のCデータ型 |
|---|---|
| **SQL_CHAR** | **SQL_C_CHAR** |
| **SQL_VARCHAR** | **SQL_C_CHAR** |
| **SQL_LONGVARCHAR** | **SQL_C_CHAR** |
| **SQL_DECIMAL** | **SQL_C_CHAR** |
| **SQL_FILE**[2] | **SQL_C_CHAR** |
| **SQL_SMALLINT** | **SQL_C_SSHORT** |
| **SQL_INTEGER** | **SQL_C_SLONG** |

[1] fractionの有効値の範囲は、0～999,999,999です。例えば、500,000,000は0.5秒を表します。1,000,000は1000分の1秒、1,000は100万分の1秒、1は10億分の1秒を表します。

[2]このデータ型の定義は、DBMasterとODBC 2.0で僅かに異なります。

| SQLデータ型 | 初期設定のCデータ型 |
|---|---|
| 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、スケール、サイズ、表示サイズ

SQLColAttributes、SQLColumns、SQLDescribeColは、表にあるカラムの精度、スケール、サイズ、表示サイズを返します。SQLDescribeParamは、SQL文のパラメータの精度やスケールを返します。SQLBindParameterは、SQL文のパラメータに精度やスケールをセットします。SQLGetTypeInfoは、データソースにあるSQLのデータ型の最大精度と最小/最大スケールを返します。以下の表は、各ODBC SQLデータ型の精度、スケール、サイズ、表示サイズです。‘—‘ は、意味の無い値や対応するデータ型を判断できない値を表します。例えば、スケールは日付データには適用できません。また、**SQL_LONGVARCHAR**の精度は、何バイトのデータが保存されているかを参照するだけなので、判断することはできません。

| FSQLTYPE | SQLのデータ型 | 精度[1] | スケール[2] | サイズ | 表示サイズ |
|---|---|---|---|---|---|
| **SQL_CHAR** | **CHAR(n)** | n | — | n | n |
| **SQL_VARCHAR** | **VARCHAR(n)** | n | — | n | n |
| **SQL_LONGVARCHAR** | **LONG VARCHAR** | — | — | — | — |
| **SQL_DECIMAL**[3] | **DECIMAL(p,s)** | p | S | (p+3)/2 | p+2 |
| **SQL_FILE**[3] | **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)** | — | — | — | — |

[1] 日付、時間、タイムスタンプの精度は、全て日付、時間、タイムスタンプの形式 (dd/mon/yyyy、hh:mm:ss.nnn tt、dd/mon/yyyy hh:mm:ss.nnn tt)の最大長である、最大表示サイズで定義されます。

[2] 時間とタイムスタンプのスケールは3です。分数部の桁数はDBMasterによって正確に許容されることを意味します。

[3] このデータ型の定義は、DBMasterとODBC 2.0で僅かに異なります。

# D.5 データ型の規則

以下の2節でデータ型の変換について説明します。1つはSQLからCへのデータ変換で、もう1つはCからSQLのデータ変換です。各節で、データ型の変換結果を示す例を紹介します。

## SQLからCのデータ変換

SQLFetchでデータを回収する前に、回収したデータを変換するデータ型をSQLBindColの引数fCTypeで指定します。ドライバは、SQLBindColの引数**rgbValue**で指定した場所にデータを保存します。SQLGetDataでも似通った状況になります。次のページの表は、DBMasterに備わっているSQLデータ型からCデータ型へのデータ型変換のリストです。

| 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_TIMESTAM | SQL_C_FILE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | ○ | | | | | | | ○ | ○ | ○ | ● | |

● - 初期設定変換
○ - サポートする変換

次の3ページにわたる表は、如何にしてDBMasterでSQLデータ型をCデータ型に変換するかを表しています。ODBCの標準との違いは、各ページの下部の脚注に記述しています。

| SQLデータ型 | SQLのデータ | Cのデータ型 | Cのｻｲｽﾞ | Cのデータ | SQL STATE |
|---|---|---|---|---|---|
| **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** | **SQL_CHAR**と同様 | | | | |
| **SQL_LONGVARCHAR** | **SQL_CHAR**と同様 | | | | |
| **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_DECIMAL** | 1234.56 | **SQL_C_CHAR** | 4 | – | **22003** |
| **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** | **SQL_DOUBLE**と同様 | | | | |
| **SQL_REAL** | **SQL_DOUBLE**と同様 | | | | |
| **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**[1] |

[1]出力データと戻りコードは、ユーザーがセットするデータ出力形式に対応しています。日付の出力形式がyyyy-mm-ddの場合、ユーザーのバッファ・サイズは最低11バイトにセットする必要があります。さもないと、エラーになります。

| SQLデータ型 | SQLのデータ | Cのデータ型 | Cのｻｲｽﾞ | Cのデータ | SQL STATE |
|---|---|---|---|---|---|
| **SQL_DATE** | 1995-11-29 | **SQL_C_CHAR** | 10 | 11-29-95\0 | N/A[1] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 19 | 22:11:33.320\0 | N/A |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 12 | 22:11:33.32\0 | **01004**[2] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 11 | 22:11:33.3\0 | **01004**[2] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 10 | 22:11:33\0 | **01004**[1] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 9 | 22:11:33\0 | **01004**[2] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | 8 | — | **22003**[3] |
| **SQL_TIME** | 22:11:33.32 | **SQL_C_CHAR** | | 10 PM\0 | N/A[4] |
| **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**[5] |
| **SQL_TIMESTAMP** | 1995-11-29 | **SQL_C_CHAR** | 19 | — | **22003**[1] |

[1]出力データと戻りコードは、ユーザーがセットするデータ出力形式に対応しています。日付の出力形式がmm-dd-yyの場合、ユーザーのバッファ・サイズは最低9バイトにセットする必要があります。さもないと、エラーになります。

[2]出力データと戻りコードは、ユーザーがセットするデータ出力形式に対応しています。時間の出力形式をhh:mm:ss.fffのように分数部を指定し、ユーザーのバッファが分数桁を代入するスペースが足りない場合、データが切り捨てられるという警告メッセージが戻ります。

[3] 時間の出力形式にhh:mm:ssのように秒の欄があり、ユーザーのバッファに秒値を代入するスペースが足りない場合、エラーが戻ります。

[4] 時間の出力形式にhh ttを指定した場合、minuteとsecondデータが無くなっても、ユーザーは時間の情報だけが必要なのでエラーは発生しません。

[5] 3と同じ理由

| SQLデータ型 | SQLのデータ | Cのデータ型 | Cのｻｲｽﾞ | Cのデータ | SQL STATE |
|---|---|---|---|---|---|
| | 23:54:38.234 | | | | |
| **SQL_TIMESTAMP** | 1995-11-29 23:54:38.234 | **SQL_C_CHAR** | 19 | 11/29/95 23:54:38\0 | N/A[2] |
| **SQL_BINARY** | ABCDEFG | **SQL_C_CHAR** | 15 | 65666768696A6B\0 | N/A |
| **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** | **SQL_BINARY**と同様 | | | | |
| **SQL_FILE** | abcdefg | **SQL_C_CHAR** | 8 | abcdefg\0 | N/A[3] |
| **SQL_FILE** | abcdefg | **SQL_C_FILE** | 8 | student\0 | N/A[4] |

## CからSQLのデータ変換

アプリケーションでSQLExecuteやSQLExecDirectを呼び出す時、ドライバはアプリケーションにある格納場所からSQLBindParameterにバインドするパラメータのデータを回収します。必要に応じて、ドライバがデータソースにデータを送信する前に、SQLBindParameterのfCType引数で指定したデータ型から、SQLBindParameterのfSqlType引数で指定したデータ型に変換

[1]時間の出力形式にhh ttを指定した場合、minuteとsecondデータが無くなっても、ユーザーは時間の情報だけが必要なのでエラーは発生しません。

[2] タイムスタンプ形式の規則は、上記の日付形式の規則と時間形式の規則を合わせたものです。つまり、日付形式がmm/dd/yyで、時間形式がhh:mm:ssの場合、バッファ・サイズは最低18 (8 + 1 + 8 + 1)必要です。さもないと、エラーになります。

[3] カラムのデータ型がFILEの場合、SQL_C_CHARでこのカラムをバインドすると、ユーザーのバッファにこのファイルの内容をフェッチします。例えば、abcdefgが'homework'というファイルの内容の場合、abcdefgはユーザーのバッファの中に代入されます。

[4] 7と同様の例、但しSQL_C_FILEを使ってこのカラムをバインドする場合、ユーザーのバッファで指定した新規ファイル(例 student)の中にそのファイルの内容をフェッチします。

します。data-at-executionパラメータでは、アプリケーションはSQLPutDataと共にパラメータを送信します。次の表は、DBMasterでサポートするODBC C のデータ型からODBC SQLのデータ型への変換を示しています。

| 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_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 | | | ○ | | | | | | | | | ○ | | | |

次の3ページにわたる表は、DBMasterがどのようにCのデータ型をSQLのデータ型に変換するかを示しています。DBMasterとODBC 3.0の仕様の違いは、ページの下部に記述しています。

[1] このデータ型のふるまいは、表の以前のデータ型と同じです。

| Cのデータ型 | Cのデータ | SQLのデータ型 | COL長 | SQLのデータ | SQL STATE |
|---|---|---|---|---|---|
| SQL_C_CHAR | Abcdef\0 | SQL_CHAR | 6 | abcdef | N/A |
| SQL_C_CHAR | abcdef\0 | SQL_CHAR | 5 | abcde | 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 | SQL_FLOATと同様 | | |
| SQL_C_CHAR | 666768696A6b\0 | SQL_BINARY | 6 | BCDEFG | 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 |

| Cのデータ型 | Cのデータ | SQLのデータ型 | COL長 | SQLのデータ | SQL STATE |
|---|---|---|---|---|---|
| SQL_C_CHAR | 1995/22/33 | SQL_DATE | — | — | 22008 |
| SQL_C_CHAR | 17:18:19.123 | SQL_TIME | — | 17:18:19.123 | N/A[1] |
| 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[2] |
| SQL_C_SHORT | 7890 | SQL_SMALLINT | — | 7890 | N/A |
| SQL_C_SSHORT | 7890 | SQL_VARCHAR | — | — | S1C00 |
| SQL_C_SSHORT | 7890 | SQL_BINARY | — | — | 07006 |
| SQL_C_SSHORT | 7890 | SQL_SMALLINT | — | 7890 | N/A[3] |
| 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 | SQL_C_FLOATと同様 | | | | |
| SQL_C_BINARY | ABCDEF | SQL_BINARY | 6 | ABCDEF | N/A |
| SQL_C_BINARY | ABCDEF | SQL_BINARY | 5 | ABCDE | 01004 |
| SQL_C_DATE | 1996,3,8 | SQL_DATE | — | 1996,3,8 | N/A |

[1] 日付の入力形式に従います。

[2] 'homework'というサーバーに1つのファイルがある場合、SQL_C_CHARを使って、指定するFILE カラム・リンクをファイル'homework'にリンクさせるパラメータをバインドします。

[3] SQL_C_SSHORT、SQL_C_SHORT、SQL_C_SLONG、SQL_C_LONG、SQL_C_FLOAT、SQL_C_DOUBLEは、数値のODBC Cデータ型です。

| Cのデータ型 | Cのデータ | SQLのデータ型 | Col長 | SQLのデータ | SQL STATE |
|---|---|---|---|---|---|
| SQL_C_DATE | 1996,3,8 | SQL_TIME | — | — | 07006[1] |
| SQL_C_DATE | 1996,3,8 | SQL_TIMESTAMP | — | 1996,3,8,0,0,0,0 | N/A[2] |
| SQL_C_TIME | 13,14,15 | SQL_DATE | — | — | 07006[3] |
| 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[4] |
| SQL_C_TIMESTAMP | 1996,3,8,13,14,15,160000000 | SQL_DATE | — | 1996,3,8 | 01004[5] |
| SQL_C_TIMESTAMP | 1996,3,8,13,14,15,160000000 | SQL_TIME | — | 13,14,15 | 01004[6] |
| 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[7] |
| SQL_C_FILE | homework\0 | SQL_LONGVARCHAR | 6 | abcdefg | N/A[1] |

[1] SQL_C_DATEは、SQL_TIMEに変換できません。

[2] SQL_C_DATEをSQL_TIMESTAMPに変換する時、タイムスタンプの時間の部分はゼロにセットされます。

[3] SQL_C_TIMEは、SQL_DATEに変換できません。

[4] SQL_TIMEをSQL_C_TIMESTAMPに変換する時、タイムスタンプの日付部分は、現在の日付にセットされます。

[5] SQL_C_TIMESTAMPをSQL_DATEに変更する時、時間部分がゼロの場合、「データが切り捨てられました」の警告メッセージが戻ります。

[6] SQL_C_TIMESTAMPをSQL_TIMEに変換する時、分数部分がゼロの場合、「データが切り捨てられました」の警告メッセージが戻ります。

[7] decimalの後に4桁以上ある場合、タイムスタンプの分数部分のdecimalの後に3桁しかないと想定され、警告メッセージが戻ります。

[1] クライアント側に'homework'というファイルが1つあり、その内容がabcdefgの場合、SQL_C_FILEを使って、BLOBカラムかFILEカラムにその内容を挿入するパラメータをバインドします。

# E　ODBCログ関数

この章では、ODBC関数のトレース・ログについてDBMasterがサポートしている全環境設定を列記しています。windowsプラットフォームでMicrosoftのODBCドライバマネージャを経由してODBC関数をトレースすることができます。但し、ドライバマネージャ自身に呼ばれた関数のログはとりません。加えてMicrosoftのログ関数は、全ODBC関数をトレースするようにログをON/OFFにセットするか、或いはそれら全てをトレースしないかのいずれかにしかセットできません。また、windowsのプラットフォームでしか使用することができません。DBMasterでは、ログをON/OFFにセットできるだけでなく、ログを取るODBC関数を指定することができるより強力なログ機能を備えています。

dmconfig.iniに、“DM_COMMON_OPTION”という名前の特殊なセクションをセットし、このセクションにODBCのログ機能を有効/無効にするための様々なキーワードをセットする必要があります。

注: あとからODBC関数のログを取らないことにした場合は、必ず“DM_COMMON_OPTION”セクションを削除するか、トレーシングをOFFにするためにLG_TRACE＝0にセットして下さい。さもないと、このログ機能は、データソースに実行したアクション全てを記録します。

 

以下の表は、**dmconfig.ini**のログ機能でセットすることができるオプションとその解説と値の一覧です。

| オプションのキーワード | 解説 | 値 | 初期設定値 |
|---|---|---|---|
| LG_TRACE | ログ機能にトレース・フラグをセット | 0-ODBCログ機能をOFFにする<br>1-ODBCログ機能をONにする | 0 |
| LG_PATH | 出力ログファイルへのパスをセット。パスが無効の場合、ログは出力されません。 | 出力ログファイルへのパス | C:\odbclog.log (Win32の場合)<br>./odbclog.log (Unixの場合) |
| LG_TIME | 各ODBC関数コールが要した時間を記録するかどうかをセット。 | 0-各ODBC関数コールが要した時間をログしない。<br>1-各ODBC関数コールが要した時間をログする。 | 0 |
| LG_PTFUN | ログを取る関数の一覧をセット。この一覧は、0又はカンマ(,)で区切った複数のODBC関数名を含む文字列です。 | 関数一覧の文字列。(例、SQLGetDiagRec, SQLCloseCursor, ”) | 定義無し。(全関数のログがとられます。) |
| LG_NPFUN | ログを取らない関数の一覧をセット。この一覧 | 関数一覧の文字列(例、SQLGetDiagRec, | “” (空の文字列、全関数のログが取られ |

| オプションのキーワード | 解説 | 値 | 初期設定値 |
|---|---|---|---|
| | は、0又はカンマ(,)で区切った複数のODBC関数名を含む文字列です。このキーワードは、dmconfig.iniにLG_PTFUNが定義されていない場合のみ有効です。 | SQLCloseCursor,”) | ます。) |

 