

ORACLE[®]
Enabling the Information Age™



実践 Oracle Objects for OLE



日本オラクル株式会社

はじめに

Oracle を使った業務アプリケーション開発で最も広く使われているのは VisualBasic(以下、VB)で、OFFICE 製品との連携の際には Excel がよく利用されています。これらの製品は共に ODBC や OO4O を介することで Oracle データベースに格納されているデータを活用することができます。ODBC と OO4O のどちらを使えば良いかに関しては、他の資料で詳細に述べられているので、本資料では述べません。

最近になってようやく、VB と Oracle での開発に関する技術書籍などが出版されるようになりました。しかし、特に、VB/OO4O/Oracle の開発における実践的なテクニックに関しては情報が少ないのが現状です。本資料はこの点に焦点をあて、実際の開発中に必要となる各種テクニック・知識に関して述べていきます。

尚、本資料では OO4O の基本的な利用方法については触れません。これらに関しては、日本オラクル株式会社のホームページ (www.oracle.co.jp)などをご覧ください。

Oracle, Oracle7, Oracle8 はオラクル社の登録商標です。

その他のすべての企業名と商品名は各社の登録商標または商標、製品名です。

なお、本文中では™・®は明記していません。

目次

I. VISUAL BASIC との連携	2
1. パフォーマンス	2
1-1. PL/SQL の利用	2
1-2. 配列の利用	9
1-3. カーソル変数・PL/SQL 表の利用	11
2. ダイナセットとロック	14
3. データ型	16
3-1. NUMBER 型	16
3-2. CHAR/VARCHAR2 型	18
3-3. DATE 型	18
4. データベースへの接続	23
4-1. いつ接続するか	23
4-2. セッション接続/切断のタイミング	24
II. EXCEL との連携	26
1. COPYTOCLIPBOARD メソッドの利用	26
III. よくある疑問点	28
1. 表名を動的に変えてINSERT 処理を行うには？	28
2. 番号を自動的に割り振ってくれる列はどう作成するの？	33
3. ORACLE から取得した列の値がNULL の場合はどうするの？	35
4. ダイナセットの作成時に制限はあるの？	36
5. 100 行の表の先頭 20 行のダイナセットを作成するには？	36
6. ダイナセットが更新可能かどうかを確認するには？	44
7. 表に DEFAULT 値を用いている場合は？	45
IV. 参考文献	47

I. Visual Basic との連携

1. パフォーマンス

どのようなアプリケーションでも、そのパフォーマンスを無視することはできません。では、VB/OO40/Oracle を用いた場合に、優れたパフォーマンスを引き出すにはどのようにしたらよいのでしょうか。アプリケーションのパフォーマンスを大きく向上させる方法には3つあります。

- ・ PL/SQL の利用
- ・ 配列の利用
- ・ カーソル変数・PL/SQL 表の利用

このほかにもパフォーマンス向上のテクニックに関しては様々なものが有りますが、本資料ではこの3つに関してその実装方法を紹介していきます。これら以外のものに関しては、別紙資料「パフォーマンスを考慮した VB/OO40/Oracle アプリケーション開発」をご覧ください。

1-1. PL/SQL の利用

PL/SQL は Oracle が独自に拡張した、手続き型のプロシージャ言語です。PL/SQL を用いて、If 分岐、LOOP 処理等を行うことができます。PL/SQL はデータベース・オブジェクトのトリガーやストアドプログラムを作成するのにも用いられます。また、クライアントから1つの PL/SQL ブロック(BEGIN ではじまり END で終わる)をサーバーに送り、その処理をすべてサーバー側で行わせることも可能です。パフォーマンスの向上のために VB から利用する PL/SQL には大きく分けて2種類があります。無名 PL/SQL ブロックとストアドプログラムです。ここでは、この2つについてその利用方法を紹介します。

無名 PL/SQL ブロック

無名 PL/SQL ブロックとはクライアント(VB)から Oracle に送る文字列(通常は SELECE...FROM...など)に PL/SQL を記述したものもことです。例えば、100 行の INSERT 文処理を行いたい場合、通常の INSERT 文でこれを実現しようとする、VB のプログラム上で 100 回のループを行い、そのループ内で INSERT 文を記述する形になります。

```
Private Sub Command1_Click()  
    sqlstmt = "insert into test (ID) values (:id_num)" '100 件 INSERT  
    OraDatabase.Parameters.Add "id_num", 1, ORAPARM_INPUT  
    OraDatabase.Parameters("id_num").ServerType=ORATYPE_NUMBER  
    Set OraSQL = OraDatabase.CreateSQL(sqlstmt, &H0&)  
    For ID = 1 To 100  
        OraDatabase.Parameters("id_num").Value = ID  
        result = OraSQL.Refresh  
    Next  
    OraDatabase.Parameters.Remove "id_num"  
End Sub
```

上記の例では、バインド変数を用いた SQL 文を VB から 100 回発行しています。つまり、100 回 INSERT 文が実行され、その結果が 100 回クライアントに戻されています。同様の処理を無名 PL/SQL ブロックを用いると以下ようになります。

```
Private Sub Command2_Click()  
    sqlstmt = "DECLARE " & _  
              "id_num INTEGER;" & _  
              "BEGIN " & _  
              "FOR id_num IN 1..100 LOOP " & _  
              "insert into test (ID) values  
(id_num);" & _  
              "END LOOP;" & _  
              "END;"  
    result = OraDatabase.ExecuteSQL(sqlstmt)
```

無名 PL/SQL ブロックを用いた場合、VB から発行される SQL 文(無名 PL/SQL ブロック)は 1 回のみになります。無名 PL/SQL ブロックが Oracle に送られ、Oracle 上で 100 回のループを行い、INSERT 文を実行しています。そして、その結果が 1 回クライアントに戻されます。同時実行ユーザーが少ない場合や、ネットワーク・トラフィックが少ない場合などには大きな違いは現れないかも知れませんが、同時実行ユーザー数が多い場合、ネットワーク・トラフィックが多い場合、WAN 環境などでネットワークの速度が遅い場合などにはこれらのパフォーマンスの差は大きくなります。

ストアドプログラムの実行

Oracle では PL/SQL を用いてストアドプログラムを作成することができます。"CREATE FUNCTION..."や"CREATE PROCEDURE..."などが実行されると、直ちにコンパイルされます。そしてコンパイルされた形でデータディクショナリに格納され、プログラムがコールされるたびにロードされるという動きをします。このため、無名 PL/SQL ブロックを用いる場合と比較して、コンパイルの時間が節約され、パフォーマンスが向上します。また、複数のユーザーが同じストアドプログラムを利用する場合でも、Oracle のシステム・グローバル領域(SGA)内の共有プールを用いて 1 つのストアドプログラムのコピーを複数のユーザーで共有することが可能です。

ストアドプログラムはその用途から以下の 3 つに分類されます。

- *ストアドプロシージャ
- *ストアドファンクション
- *ストアドパッケージ

通常、ストアドプロシージャは何らかのアクションを実行するために用いられ、ストアドファンクションは値を計算するのに用いられます。またストアドパッケージはストアドプロシージャ/ファンクションなどをまとめたスキーマオブジェクトです。

ストアドプロシージャ

ここでは VB からストアドプロシージャを実行する方法を紹介します。まず、Oracle 上にストアドプロシージャを作成する必要があります。以下のようにストアドプロシージャを作成してください。

```
CREATE OR REPLACE PROCEDURE sp_test
IS
    id_num INTEGER;
BEGIN
    FOR id_num IN 1..100 LOOP
        insert into test (ID) values (id_num);
    END LOOP;
END;
/
```

SQL*Plus などからこの SQL 文を発行します。これにより Oracle にストアプロシージャ sp_test が作成されます。作成時にエラーが発生した場合は、

```
SQL> show error
```

上記のように記述することで、エラーの発生個所を特定することができます。

このストアードプロシージャを VB から実行するには OraDatabase オブジェクトの ExecuteSQL メソッドを使います。実行する SQL 文は "BEGIN ... ; END ;" を用います。

```
Private Sub Command3_Click()  
    sqlstmt = "BEGIN sp_test; END;"  
    OraDatabase.ExecuteSQL (sqlstmt)  
End Sub
```

ストアードプロシージャが引数を持つ場合も同様にして VB から実行することが可能です。その場合、ストアードプロシージャの引数にバインド変数を割り当てその引数と VB の変数を関連付けます。

ストアードファンクション

ストアードファンクションは主に値の計算に利用されます。ストアードプロシージャと異なり、ストアードファンクションは RETURN 句によって返される戻り値を持ちます。ストアードファンクションを用いることで VB アプリケーションのパフォーマンスに著しく影響を与えることはあまりありません。ここでは、簡単なストアードファンクションの作成例とその実行方法を紹介します。

まず、以下のように Oracle 上にストアードファンクションを作成します。

```
CREATE OR REPLACE FUNCTION sf_test RETURN NUMBER  
IS  
    v_number    NUMBER;  
BEGIN  
    v_number := 1;  
    RETURN v_number;  
END;  
/
```

このファンクションは単に、変数 v_number に 1 を代入し、それを RETURN 句で戻り値として返しているだけです。このファンクションを VB から実行するには "SELECT ファンクション FROM dual" としてダイナセットを作成しその値を利用する形になります。単純な処理であれば、わざわざ Oracle 上にストアードファンクションを作成することなく、VB 上でファンクションを作成するほうが効率的です。次に実行例を示します。

```

Private Sub Command3_Click()
    Dim v_function As Integer
    sqlstmt = "SELECT sf_test FROM dual"
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H0&)
    v_function = OraDynaset.Fields(0).Value
    MsgBox "ファンクションの戻り値は" & v_function & "です。"
End Sub

```

ストアドパッケージ

先に述べたように、ストアドパッケージとはストアドプロシージャやストアドファンクションをまとめたスキーマオブジェクトです。例えば、先程作成例を紹介したストアドプロシージャ sp_test およびストアドファンクション sf_test をストアドパッケージにまとめておくとします。この場合には、まず、ストアドパッケージの仕様部を記述し、本体部に sp_test 及び sf_test を記述します。

次に例を挙げて説明します。まず作成するストアドパッケージの仕様部を作成します。

```

CREATE OR REPLACE PACKAGE pkg_test AS
    PROCEDURE sp_test;
    FUNCTION sf_test RETURN NUMBER;
END pkg_test;
/

```

次にストアドパッケージの本体部にストアドプロシージャ/ファンクションを実装します。

```

CREATE OR REPLACE PACKAGE BODY pkg_test AS
    PROCEDURE sp_test IS
        id_num INTEGER;
    BEGIN
        FOR id_num IN 1..100 LOOP
            insert into test (ID) values (id_num);
        END LOOP;
    END sp_test;
    FUNCTION sf_test RETURN NUMBER IS
        v_number NUMBER;
    BEGIN
        v_number := 1;
        RETURN v_number;
    END sf_test;
END pkg_test;
/

```

これでストアドパッケージが作成できました。ストアドパッケージ内のストアドプロシージャ/ファンクションをコールする際には、ストアドパッケージ名.ストアドプロシージャ/ファンクション名を用います。VB から利用する部分のそれぞれの例を次に示します。sqlstmt 文字列の中の呼び出すストアドプログラム名の指定方法が変わっています。

*ストアドプロシージャ

```
Private Sub Command3_Click()  
    sqlstmt = "BEGIN pkg_test.sp_test; END;"  
    OraDatabase.ExecuteSQL (sqlstmt)  
End Sub
```

*ストアドファンクション

```
Private Sub Command3_Click()  
    Dim v_function As Integer  
    sqlstmt = "SELECT pkg_test.sf_test FROM dual"  
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H0&)  
    v_function = OraDynaset.Fields(0).Value  
    MsgBox "ファンクションの戻り値は" & v_function & "です。 "  
End Sub
```

VB アプリケーションからストアドパッケージを利用するのは、1.1.3 で述べるカーソル変数および PL/SQL 表を用いる時がほとんどです。カーソル変数と PL/SQL 表の利用方法は後述します。

1-2. 配列の利用

開発者の方であれば、ほぼ間違いなく配列を用いてプログラムを組んだことがあると思います。Oracle には配列インターフェースが用意されており、バインド変数や PL/SQL 表などを用いた際にバインドするプログラム変数を配列にすることで処理を高速化することが可能です。配列を用いることにより得られる効果は、PL/SQL を活用して得られる効果と同様に、ネットワーク・トラフィックの減少などが上げられます。通常のバインド変数と配列バインド変数とではパフォーマンスに著しい差が出ます。

まず、配列を使わず単にバインド変数を用いた場合の 100 回の INSERT 処理の例を示します。

```
Private Sub Command1_Click()
    sqlstmt = "insert into test (ID) values (:id_num)" '100 件 INSERT
    OraDatabase.Parameters.Add "id_num", 1, ORAPARM_INPUT
    OraDatabase.Parameters("id_num").ServerType=ORATYPE_NUMB
ER
    Set OraSQL = OraDatabase.CreateSQL(sqlstmt, &H0&)
    For ID = 1 To 100
        OraDatabase.Parameters("id_num").Value = ID
        result = OraSQL.Refresh
    Next
    OraDatabase.Parameters.Remove "id_num"
End Sub
```

このサンプルは無名 PL/SQL ブロックで利用したサンプルです。同様の処理を配列を用いて行うと以下のようになります。

```
Private Sub Command3_Click()
    sqlstmt = "insert into test (ID) values (:id_num)" '100 件 INSERT
    OraDatabase.Parameters.AddTable "id_num", _
        ORAPARM_INPUT, ORATYPE_NUMBER, 100, 5
    Set OraParamArray = OraDatabase.Parameters("id_num")
    For ID = 1 To 100
        OraParamArray.put_Value ID, ID - 1
    Next ID
    OraDatabase.ExecuteSQL (sqlstmt)
    OraDatabase.Parameters.Remove "id_num"
End Sub
```

配列を用いた場合でも、配列の各要素に値をセットするために 100 回のループを行っていますが、SQL 文の発行は 1 回だけです。この処理の場合、配列を用いるだけで単純に考えてネットワークラウンドトリップが 100 分の 1 になっています。パフォーマンスの差は非常に大きくなります。

1-3. カーソル変数・PL/SQL 表の利用

PL/SQL ではカーソル変数および PL/SQL 表が利用できます。カーソル変数を利用することで、ストアードプロシージャから結果セットを戻すことが可能になります。これは、ちょうど Microsoft SQLServer ストアドプロシージャに SELECT 文を記述し結果セットを返す場合と良く似ています。PL/SQL で利用するデータ構造の一種で、PL/SQL 表は配列に似ています。例えば、複数の行をまとめて更新する時などは、PL/SQL 表を IN 引数にもつストアードプロシージャを用いることができます。

カーソル変数

ここではカーソル変数を戻すストアードプロシージャを使った OO4O のダイナセットを作成する方法を示します。まず、カーソル変数を利用するために必要なストアードパッケージおよびストアードプロシージャを記述します。

```
CREATE OR REPLACE PACKAGE pkg_ref AS
  CURSOR c1 IS SELECT ename FROM emp;
  TYPE empCur IS REF CURSOR RETURN c1%ROWTYPE;
  PROCEDURE GetEmpData(indeptno IN NUMBER,
                      EmpCursor in out empCur );
END;
/
```

まずこのように、パッケージの仕様部を作成します。ここでは、カーソルの宣言やプロシージャの定義を行っています。次に本体部を作成します。

```
CREATE OR REPLACE PACKAGE BODY pkg_ref AS
  PROCEDURE GetEmpData(indeptno IN NUMBER,
                      EmpCursor in out empCur ) IS
  BEGIN
    OPEN EmpCursor FOR SELECT ename FROM emp WHERE deptno
= indeptno;
    END GetEmpData;
END pkg_ref;
/
```

これで必要なストアードパッケージおよびストアードプロシージャが作成されました。これを VB から実行し、カーソル変数を結果セットとしてダイナセットを作成するには次のようにします。

```
Private Sub Command3_Click()  
    OraDatabase.Parameters.Add "DEPTNO", 10, ORAPARM_INPUT  
    OraDatabase.Parameters("DEPTNO").ServerType = _  
                                                ORATYPE_NUMBER  
    sqlstmt = "Begin pkg_ref.GetEmpData (:DEPTNO,:EmpCursor);  
end;"  
    Set OraDynaset = OraDatabase.CreatePlsqlDynaset _  
                                                (sqlstmt, "EmpCursor", 0&)
```

このようにして作成したダイナセットは通常の SELECT 文で作成したダイナセットと同様に扱うことができます。

PL/SQL 表

ここでは、PL/SQL 表を IN 引数に持つストアドプロシージャの実行例を示します。まず、ストアドパッケージおよびストアドプロシージャを作成します。

*仕様部

```
CREATE OR REPLACE PACKAGE pkg_ins IS
  TYPE t_val_tbl IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  PROCEDURE sp_ins(val_tbl t_val_tbl, i_count INTEGER);
END pkg_ins;
/
```

*本体部

```
CREATE OR REPLACE PACKAGE BODY pkg_ins IS
  PROCEDURE sp_ins(val_tbl t_val_tbl, i_count INTEGER)
  IS
    i_index INTEGER;
  BEGIN
    For i_index In 1..i_count LOOP
      INSERT INTO test(ID) VALUES (val_tbl(i_index));
    END LOOP;
  END sp_ins;
END pkg_ins;
/
```

VB からこのプロシージャを実行するには OraDatabase.AddTable メソッドを用います。ストアドプロシージャ sp_ins には引数が 2 つあり、1 つは PL/SQL 表、もう一つは INTEGER で INSERT する行数を指定します。次に例を示します。

```
Private Sub Command2_Click()
  sqlstmt = "BEGIN testPkg.testSP(:id_num, 100); END;" '100 件
INSERT
  OraDatabase.Parameters.AddTable "id_num", ORAPARM_INPUT, _
    ORATYPE_NUMBER, 100, 5
  Set OraParamArray = OraDatabase.Parameters("id_num")
  For ID = 1 To 100
    OraParamArray.put_Value ID, ID - 1
  Next ID
  OraDatabase.ExecuteSQL (sqlstmt)
  OraDatabase.Parameters.Remove "id_num"
```

2. ダイナセットとロック

ACCESS などを利用していたユーザーが、データ量の増加と共にそのパフォーマンスや堅牢性で限界を感じ、RDBMS に乗り換えるという例は数多くあります。この際に、ファイルシステムベースのデータベースである ACCESS と RDBMS とには、複数ユーザーでデータベースを共有して運用する場合に大きく異なる点があります。データのロックです。ロックに関しては、各 RDBMS のベンダーにより実装方法は異なりますが、基本的な概念は同じです。ユーザーA が更新中のデータはその更新が確定するまで(トランザクションが終了するまで)、ユーザーB が更新することはできないというものです。この場合、ユーザーA がロックを保持しているため、ユーザーB は更新処理に必要なロックの獲得待ちの状態になります。

作成した Dynaset を用いて、データを更新しようとする場合に、OO4O では Edit メソッドを発行します。通常、更新の際に必要なステップは以下の通りです。

```
Private Sub Command1_Click()  
    sqlstmt = "select empno, ename from emp"  
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H0&)  
    1  
    OraDynaset.Edit      2  
    OraDynaset.Fields(1).Value = "CLINTON"  
    OraDynaset.Update    3
```

この一連の処理において発行される SQL 文を紹介します。

1. CreateDynaset メソッド
 - SELECT empno, ename FROM emp
 - SELECT empno, ename, ROWID InternalRowid from emp
2. Edit メソッド
 - SELECT empno, ename FROM emp WHERE ROWID = :1 FOR UPDATE
3. Update メソッド
 - UPDATE emp set ename = :1 WHERE ROWID = :2

ヒント VB から Oracle にどのような SQL 文が発行されているかを知るために、"ALTER SESSION SET SQL_TRACE = TRUE"を実行します。そうすることにより、接続セッションから発行されている SQL 文を全て把握することができます。トレースファイルは%ORACLE_HOME%\RDBMS80\TRACE ディレクトリに作成されます。

例 : sqlstmt = "ALTER SESSION SET SQL_TRACE=TRUE"
 OraDatabase.ExecuteSQL (sqlstmt)

OO40 では Oracle の内部データ型である ROWID を用いて、ダイナセットの作成・更新やナビゲーションを行っています。これを取得しているのが、CreateDynaset メソッドです。Edit メソッド発行時には、データの更新のために "SELECT...FOR UPDATE" を用いて、更新する行のロックを獲得します。対象行が既に別のセッションでロックされている場合は、ここでロックの競合が起こります。NO_WAIT モードでない場合は、別セッションでロックされている業の処理が含まれるトランザクションがコミット/ロールバックされるまでロック待ちの状態になり、アプリケーションはフリーズした状態になります。これを避けるために、VB ではロック競合をした場合にはエラーを返し、すぐにアプリケーションに制御を戻す NO_WAIT オプションがよく利用されます。ロック競合の際に返されるエラー番号を元に処理を分岐させることが可能です。最後に Update メソッドの発行時に、実際の UPDATE 文を発行しデータの更新を行っています。

また、Dynaset を利用せずにデータを更新する場合があります。その場合、明示的にロックを確保し、そのデータに対して、UPDATE 文を発行するという形をとります。その場合更新するデータを "SELECT...FOR UPDATE" でロックし、ロック獲得後 UPDATE 文を OraDatabase.ExecuteSQL で発行します。

3. データ型

VB で用いられるデータ型と Oracle で用いられるデータ型は当然異なります。これらの違いはプログラム上でそのギャップを埋めなければなりません。ここでは、Oracle で頻繁に使われるデータ型である、NUMBER、CHAR、VARCHAR2、DATE とそれらに対応する VB のデータ型およびその取り扱いについて述べていきます。

DOA(Data Oriented Approach)では、まずデータベース内のデータ型が先に決定されます。それらをアプリケーション側から利用するために、開発ツールで用いることができるデータ型のどれと対応させるかが問題となります。ここでは、Oracle のデータ型と VB のデータ型の対応に関して述べます。VB のデータ型で頻繁に利用するものに、数値型、文字列型、日付型があります。

3-1. NUMBER 型

Oracle では数値型のデータはすべて NUMBER 型で表現されます。NUMBER 型は以下のように定義されています。

固定小数点数および浮動小数点数に使用する。1~9.99x(10 の 125 乗)までの正と負の数値およびゼロを格納。最大小数桁数 38 桁。次のように「精度」と「位取り」を指定する

```
column_name NUMBER(precision[精度]), scale[位取り])
```

マニュアル「Oracle8 Server アプリケーション開発者ガイド」の 5-6 にも記載されてますが、NUMBER 型の定義の例を次に示します。VB の数値型には Byte, Integer, Long, Single, Double があります。NUMBER(20) のような整数値しかとらないデータに関しては、Integer もしくは Long で対応できます。小数点以下の値を含む NUMBER(20, 5) のようなデータは、Single、Double を用いる必要があります。これらは以下のように Oracle のデータ型と対応付けられます。

*整数値で 32767 ~ -32767 の値を取る場合

VB : INTEGER ORACLE: NUMBER(5) もしくは NUMBER(4)
NUMBER(5) の場合は 32767 ~ -32767 になります。
NUMBER(4) の場合は 9999 ~ -9999 になります。

*整数値で 2147483647 ~ -2147483647 の値を取る場合。

VB : LONG ORACLE : NUMBER(10) もしくは
NUMBER(9)

NUMBER(10) の場合は

2147483647 ~ -2147483647 になります。

NUMBER(9) の場合は 999999999 ~ -999999999 になりま

す。

VB の Single 型、Double 型の定義は非常に細かい数値になっています。しかし、Oracle に格納されているデータを VB 上で扱う際に、Single および Double を利用するのは小数点以下の値を含む数値のデータを扱う場合がほとんどです。ここでは、現実的にわかりやすく、「数値が小数点何桁以下の値を持つ場合」という考え方で区別します。

*小数点以下が 6 桁まで

VB : SINGLE

ORACLE : NUMBER(?, 6) ---- ?は 6 以上

*小数点以下が 15 桁まで

VB : DOUBLE

ORACLE : NUMBER(?, 15) ---- ?は 15 以上

ただし、これらの Single/Double は 1.xxxxE+31
といった形で表現されることもあるので注意
が必要です。

3-2. CHAR/VARCHAR2 型

文字列のデータ型には固定長文字列の CHAR 型と可変長文字列の VARCHAR2 型があります。

これらに対応する VB のデータ型は String 型になります。文字列データの取り扱いで注意が必要なのは、例えば、Oracle の CHAR(10)の列に "TEST" という 4 文字を入れると、データベース内では "TEST" と 4 文字+6 つのスペースが格納されることです。ODBC などを用いた場合は、これらのスペースを TRIM 関数などを用いて明示的に切り取る必要があります。OO4O ではデフォルトで、これらのスペースを取り除いたデータを変数に格納することができます。オプションの指定を変えることでスペースを取り除かずに、この場合は 10 文字(6 つのスペースを含む)を、変数に格納することも可能です。

OO4O の Dynaset 作成時のオプションに NO_BLANKSTRIP を指定すると「データベースから取り出された文字列データから、後続するブランクを取り除かない」(OO4O マニュアル参照)で変数にデータを格納することができます。

3-3. DATE 型

日付型も頻繁に利用されるデータ型ですが、これは文字列型と違い少々厄介です。ほとんどの場合、アプリケーションの要件毎に日付データの書式(表示方法)が異なります。Oracle で使われている日付型の書式とアプリケーション内で利用している日付型の書式が一致していれば問題にはなりません。異なる場合には、書式の変換を行わないと、Oracle とのデータのやり取りができません。また、2000 年問題が騒がれている昨今、日付型の書式には注意が必要です。ここでは、西暦による年を 4 桁で表示し、月および日を "/" で区切る書式(例: "1999/04/04")を用いて説明します。

Oracle に格納されている DATE 型データは Oracle で用いる環境変数の一つである NLS_DATE_FORMAT に基いて扱われます。つまり、クライアントから "select * from dual" とした場合は、デフォルトでは "99-04-01" のように日付が表示されます。NLS_DATE_FORMAT を設定していない場合は、NLS_LANGUAGE パラメータにより書式が決定されます。このようなパラメータを意識しなくても、Oracle では、to_date() および to_char() 関数を用いることで Date 型の書式を表面的に変えることも可能です。

VB の Date 型の書式は、VB を実行している OS の設定により異なります。通常、Format() 関数などを使わず Date 型のデータを表示すると、"99/04/12 午後 4:21:14" となります。ここではまず日付のデータ(時刻データを除く)の取り扱いに関して説明します。

アプリケーションで Date 型のデータを使用する場合、どのような書式でそのデータを利用するかを決定する必要があります。ユーザーが日付データを入力する際の書式、および Oracle にデータを格納する際の書式です。データベースに格納されている Date 型データは書式情報を持ちません。VB の Date 型の変数も同様です。表示される書式はユーザー(アプリケーション開発者)の責任で決定・管理する必要があります。ここでは、"1999/04/01" のような書式を例に説明します。

まず、VB アプリケーションに Oracle の Date 型のデータを取り出す際の書式設定です。これには 2 つの方法があります。

1. クライアントの環境変数 NLS_DATE_FORMAT を "YYYY/MM/DD" に設定する。Windows クライアントの場合、レジストリに NLS_DATE_FORMAT 環境変数を追加・修正する必要があります。レジストリの [HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE] に NLS_DATE_FORMAT の環境変数が含まれていない場合は、[編集]-[新規作成]-[文字列] で "NLS_DATE_FORMAT" を追加し、その値に "YYYY/MM/DD" を設定します。
2. 利用中のセッションから "ALTER SESSION SET NLS_DATE_FORMAT='YYYY/MM/DD'" を発行する。
3. to_date(sysdate, 'YYYY/MM/DD') を用いる

上記のように、"YYYY/MM/DD" の書式で Oracle から Date 型のデータを取り出しても、VB の Date 型書式のデフォルトは "yy/mm/dd" ですので、アプリケーションで日付データを表示させる際には、西暦の年数が 2 桁で表示されてしまいます。これを回避し、常に 4 桁で西暦を表示するには、常に、format(v_Date, "yyyy/mm/dd") を用いるか、Oracle から取得したデータを文字列型に格納する方法があります。後者の場合、アプリケーション内で、日付の比較などを行うときに明示的に型変換(IsDate, CDate など)を行う必要が出てきます。

ここまで、データ検索時の方法を説明してきましたが、データの更新・挿入時にはどのようにすればよいのでしょうか。これは、2 つのパターンに分類されます。

1. ダイナセットを用いて更新・挿入を行う場合

実践 Oracle Objects for OLE

この場合、VB のデフォルトの書式である "yy/mm/dd" の形で OraDynaset 内の OraField オブジェクトの値を更新することで、問題なく Oracle の Date 型データも更新されます。

2. SQL 文を用いて更新・挿入を行う場合

この場合は、さらに 2 つの場合に分類されます。

a. バインド変数を用いる場合

この場合、バインド変数の ServerType を Date 型に設定します。

```
OraDatabase.Parameters.Add "V_DATE", txtString.Text, 1
```

```
OraDatabase.Parameters("V_DATE").ServerType = 12
```

b. 文字列として SQL 文に含める場合

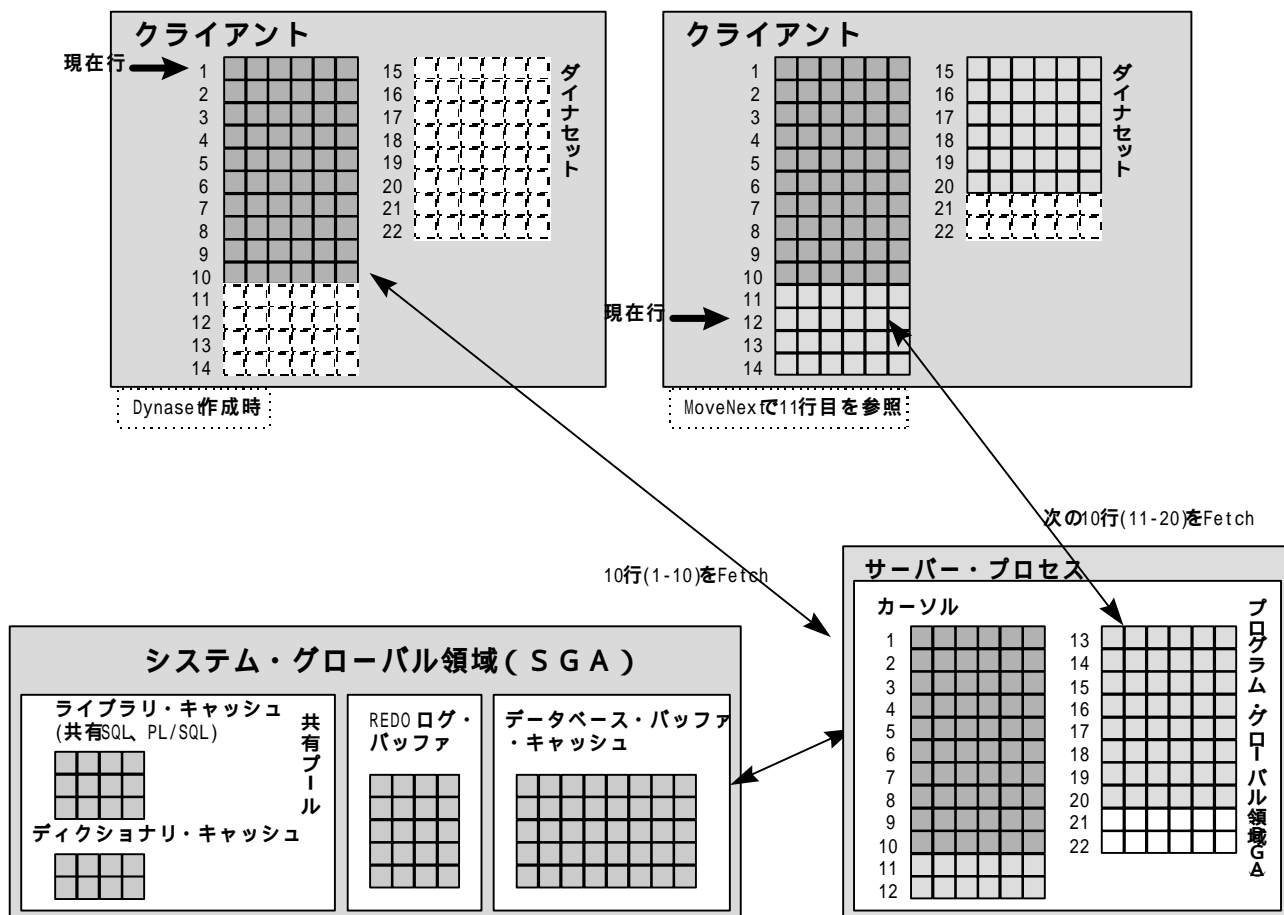
これは通常の VB の文字列の結合と同様です。 これらの場合でも Date 型のデータの更新・挿入は問題なく行えます。

ヒント Oracle に格納されているデータが"NULL"の場合、その値をそのまま VB に返すことはできません。VB 側で NULL 値に関する取り扱いのエラーが発生します。このエラーの回避方法は「III. よくある質問 - 3. Oracle から取得した列の値が NULL の場合はどうするの?」を参照してください。

おまけ : OO4O のデータ処理構造

OO4O を利用する際にサーバー側とクライアント側でどのようにデータがやり取りされているかを把握しておくことは、アプリケーション開発において重要な要素です。ここではその点について説明します。VB から OO4O を介してダイナセットを作成する時、およびダイナセット内をナビゲートしてデータ処理を行う際に、Oracle 上のデータがどのようにクライアント側のキャッシュに Fetch されるかを説明します。

1. クライアントから SELECT 文を発行(SELECT * FROM EMP)
2. サーバーで SELECT 文を解析
3. SELECT 文を実行し、PGA 内にカーソルを作成。(作成時のカーソルは先頭行に位置する)
4. クライアント側の OO4O の CreateDynaset で指定した FetchLimit の数(例: FetchLimit=10) の数の行をクライアントのローカルキャッシュにフェッチする。(例の場合、最初の 10 行がクライアントのローカルキャッシュに保持される。PGA 内のカーソルに位置は 11 行目)。この時、クライアントの CreateDynaset のカーソルポインタの位置は先頭行に位置する。
5. OraDynaset 内で MoveNext など、11 行目を参照しようとしたときに、2 回目の Fetch が発生する。この時、11 行目から 20 行目までがクライアントのローカルキャッシュに Fetch される。サーバーの PGA 内のカーソルの位置は 21 行目。



4. データベースへの接続

4-1. いつ接続するか

アプリケーションから Oracle に接続する処理はコストがかかる処理です。ユーザーにストレスを感じさせないレスポンスを備えたアプリケーションにするには Oracle への接続のタイミングをいつにするかが重要なポイントになります。アプリケーションから Oracle への接続の数もできるだけ少なくするのがより好ましい形態になります。単純なアプリケーションであれば、Oracle に対する接続は 1 つでほとんどの処理を行えます。別の Oracle インスタンスに接続する必要がある場合は、接続している Oracle インスタンスからデータベース・リンクを利用することで、別の Oracle インスタンスに格納されている表などを扱うことができます。この場合、アプリケーションの側からみると接続は 1 つですが、利用している Oracle インスタンスは 2 つという形になります。

では、この接続はプログラムのどの部分で行うのがよいのでしょうか。通常、アプリケーションを起動する際には、ある程度の時間がかかるとほとんどのユーザーが認識しています。Oracle への接続処理も比較的時間のかかるものですので、このタイミングで行うことにより、ユーザーにストレスを感じさせなくすることができます。VB のプロシージャでは [Form_Load] に当たります。アプリケーションが起動して最初に表示されるフォームの [Form_Load] プロシージャに以下のように記述します。

```
Private Sub Form_Load()  
    Set OraSession = CreateObject("OracleInprocServer.XOraSession")  
    Set OraDatabase = OraSession.OpenDatabase("dms_tech803", _  
        "scott/tiger", &H0&)  
End Sub
```

また、アプリケーション内で 1 つの接続を共有するので、0040 のオブジェクトである OraSession オブジェクトや OraDatabase オブジェクトは標準モジュール (.bas) ファイルなどで以下のように宣言する必要があります。

```
Public OraSession As OraSession  
Public OraDatabase As OraDatabase
```

0040 に付属のサンプルなどで利用されている ORACONST.bas 内にこれらを記述し、プロジェクトに含めることで、0040 で利用する各オブジェクトの定数を利用することも可能になります。

ヒント クライアントアプリケーションが異常終了してしまった場合でも、Oracle から見たクライアントとの接続は保持されたままになります。Oracle に接続する際には Net8 を介して接続しています。サーバー側の Net8 リスナーは一定時間クライアントからの処理要求がない場合、その接続を切断する設定ができます。これにはパラメータ EXPIRE_TIME(sqlnet.ora) を設定します。クライアント・アプリケーションがロックを保持したまま異常終了した場合に、そのロックが開放されず残ります。EXPIRE_TIME を設定していなければ、Oracle インスタンスの再起動、もしくはデータベース管理者が明示的にセッションを指定し、ロックを保持して異常終了したセッションを切断する必要があります。特に、このような場合に EXPIRE_TIME は有効です。

4-2. セッション接続/切断のタイミング

ここでは、コードレベルのどのタイミングで Oracle との接続が確立し、どのタイミングで切断されるかに関して紹介します。4-1 で紹介したように OO4O では Oracle に接続する際に、

```
Set OraSession = CreateObject("OracleInprocServer.XOraSession")
Set OraDatabase = OraSession.OpenDatabase("dms_tech803", _
    "scott/tiger", &H0&)
```

を必ず記述します。1 行目は、OLE インプロセスサーバーを初期化するコールです。この時点では Oracle とは接続されていません。2 行目の OpenDatabase メソッド発行時に OraDatabase オブジェクトが生成され、この時点で Oracle との接続が確立します。

次に切断のタイミングですが、OO4O には CloseDatabase というようなメソッドは存在しません。OraDatabase オブジェクトの生成で接続が確立されたことからわかりのように、OraDatabase オブジェクトが破棄されるタイミングで Oracle との接続が切断されます。

```
Set OraDatabase = Nothing
```

上記のように明示的にオブジェクトを Nothing している場合は、この文が実行された時点で Oracle との接続がなくなります。明示的に Nothing されていない場合は、OraDatabase オブジェクトの有効範囲を外れると切断されます。例えば以下の例では、Command5_Click() プロシージャの終了と同時に Oracle との接続が切断されます。

```
Private Sub Command5_Click()
    Dim OraDatabase1 As OraDatabase
    Set OraDatabase1 = OraSession.OpenDatabase("linux805", _
        "scott/tiger", 0&)
End Sub
```


II. Excel との連携

1. CopyToClipboard メソッドの利用

Oracle と Excel を連携する際にも OO4O を利用することが可能です。Excel のマクロ内に OO4O のコードを記述することにより、VB とほぼ同じ形でデータのやり取りが行えます。VB と異なる点は OO4O で Excel 用に提供されている CopyToClipboard メソッドです。このメソッドを用いることで、作成した OO4O のダイナセットのデータを一括して、Excel に貼り付けることが可能になります。

1 番目の引数にはクリップボードにコピーする行数を指定します。2 番目の引数には、列と列の間に挿入する列セパレータ(区切り文字)を指定し、3 番目の引数には、行と行の間に挿入する行セパレータ(行区切り文字)を指定します。このメソッドを利用する際に注意が必要なのは、CopyToClipboard メソッドは「ダイナセットの現在行から最終行まで(コピーする行数が指定された場合はその行数まで)クリップボードに行をコピーする」ことです。さらに、コピー終了後、ダイナセットの現在行を CopyToClipboard が発行された時点の行に戻すところです。

つまり、ダイナセットの現在行が「5」のときに、オプションを 20 行にして、CopyToClipboard メソッドを発行すると、5 行目から 25 行目までが、クリップボードにコピーされ、ダイナセットの現在行が 5 行目に戻されます。したがって、20 行ずつ Excel シートに貼り付けるには、先頭行から 20 行で CopyToClipboard をして Excel に張り付け、MoveNextn 20 で 21 行目にダイナセットの現在行を移し、先頭行から 20 行で CopyToClipboard をして Excel に張り付けることを繰り返すこととなります。CopyToClipboard メソッドを使わず、VB での処理のときと同様に 1 行ずつデータを貼り付けていく方法もありますが、上記の方法で貼り付けを行うほうがパフォーマンス的には優れています。

最後に、CopyToClipboard メソッド使用時の動きについて簡単にまとめます。

1. クライアントから SELECT 文を発行(SELECT * FROM EMP)
2. サーバーで SELECT 文を解析
3. SELECT 文を実行し、PGA 内にカーソルを作成。(作成時のカーソルは先頭行に位置する)
4. クライアント側の OO4O の CreateDynaset で指定した FetchLimit の数(例: FetchLimit=10) の数の行をクライアントのローカルキャッシュにフェッチする。(例の場合、最初の 10 行がクライアントのローカルキャッシュに保持される。PGA 内のカーソルに位置は 11 行目)この時、クライアントの CreateDynaset のカーソルポインタの位置は先頭行に位置する。

-
5. CopyToClipboard メソッド(全件)を発行する。OraDynaset 内のキャッシュされているデータの先頭行から順にクリップボードにコピーしていく。11 行目を参照しようとしたときに、2 回目の Fetch が発生する。この時、11 行目から 20 行目までがクライアントのローカルキャッシュに Fetch される。サーバーの PGA 内のカーソルの位置は 21 行目。
 6. これを繰り返し、全行がクリップボードにコピーされる。
 7. 全ての行がクリップボードにコピーされたら、OraDynaset のカーソルポインタの位置を CopyToClipboard メソッド発行時の位置に戻す。(この場合先頭行)
 8. CopyToClipboard 発行時に、オプションでコピーする行数が指定された場合でも動作は同じ

III. よくある疑問点

1. 表名を動的に変えてINSERT 処理を行うには？

動的 SQL 文は難しいというイメージがあります。しかし、VB などから OO4O を用いて SELECT 文を発行する場合は気にする必要はありません。OO4O では CreateDynaset メソッドや ExecuteSQL メソッドの発行時に指定した文字列に含まれる SQL 文が Oracle に送られ解析されるからです。言い換えれば、OO4O を用いる場合は常に動的 SQL を行っているとも言えるでしょう。その証拠に、OO4O の ExecuteSQL メソッドを用いれば DDL 文を実行することもできます。これは、OO4O は Oracle の最もネイティブな API である Oracle Call Interface をラッピングしたものであるからです。

しかしながら、PL/SQL を利用し、ストアードプロシージャの中で動的 SQL を行うには、Oracle から提供される DBMS_SQL パッケージを利用する必要があります。。ここでは、ストアードプロシージャを作成し、その IN 引数に表名及び INSERT する列の値を指定し、INSERT を実行します。以下に、サンプルを示します。なお、DBMS_SQL パッケージに関する詳細はマニュアル「Oracle8 Server アプリケーション開発者ガイド」を参照してください。

このサンプルでは、同じ構造を持つテーブルを 3 つ作成し、ストアードプロシージャの実行時にどのテーブルに対して INSERT を発行するかを決定し、それを引数としてストアードプロシージャに渡し、実行しています。まず、表を 3 つ作成します。

```
CREATE TABLE TEST1(  
pk      NUMBER(10) PRIMARY KEY,  
name    VARCHAR2(30));  
  
CREATE TABLE TEST2(  
pk      NUMBER(10) PRIMARY KEY,  
name    VARCHAR2(30));  
  
CREATE TABLE TEST3(  
pk      NUMBER(10) PRIMARY KEY,  
name    VARCHAR2(30));
```

次にストアプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE dynamic_sql
(table_name_in IN VARCHAR2, v_pk IN NUMBER, v_name IN
VARCHAR2)
IS
  cursor_id          INTEGER;
  rows_processed     INTEGER;
BEGIN
  cursor_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor_id, 'INSERT INTO ' || table_name_in || '
VALUES (:v_pk, :v_name)', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(cursor_id, 'v_pk', v_pk);
  DBMS_SQL.BIND_VARIABLE(cursor_id, 'v_name', v_name);
  rows_processed := DBMS_SQL.EXECUTE(cursor_id);
  DBMS_SQL.CLOSE_CURSOR(cursor_id);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_SQL.CLOSE_CURSOR(cursor_id);
END;
```

このストアプロシージャに対して、適切な引数を渡して実行することで INSERT 文が発行される表が動的に変わります。

動的 SQL を実装する際にもっとも厄介なのが SELECT 文中で FROM 句が動的に変化するものです。ここでは、東京 (ORDER_TOKYO)、大阪 (ORDER_OSAKA)、名古屋 (ORDER_NAGOYA) で都市別で格納されている注文データを全体の注文データ用の表 (ORDER_TOTAL) に格納するストアプロシージャを作成します。プロシージャの呼び出しの際に、アプリケーションからどの都市のデータを全体のデータに UPLOAD するかを引数として渡します。利用する表の構造とそのデータは以下の通りになります。

表:ORDER_TOTAL, ORDER_TOKYO, ORDER_OSAKA, ORDER_NAGOYA のすべてが同じ構造を持つ。

```
CREATE TABLE ORDER_XXXXXX
(ID          NUMBER(10) PRIMARY KEY,
ORDER_DATE  DATE,
QUANTITY    NUMBER(5),
PRICE       NUMBER(10));
```

サンプルデータ

```
SQL> SELECT * FROM order_tokyo;
```

ID	ORDER_DA	QUANTITY	PRICE
990410001	99-04-01	5	10000
990410002	99-04-01	1	2000
990410003	99-04-03	15	17000
990410004	99-04-04	3	5000
990410005	99-04-04	10	20000

```
SQL> SELECT * FROM order_osaka;
```

ID	ORDER_DA	QUANTITY	PRICE
990420001	99-04-02	2	3000
990420002	99-04-02	6	12000
990420003	99-04-09	7	17000
990420004	99-04-11	8	8000

```
SQL> SELECT * FROM order_nagoya;
```

ID	ORDER_DA	QUANTITY	PRICE
990430001	99-04-01	1	3500
990430002	99-04-01	2	2000
990430003	99-04-03	5	7000
990430004	99-04-04	3	5000
990430005	99-04-04	9	20000

上記の表およびデータを利用し、ストアプロシージャで動的 SQL を使い、ORDER_TOTAL 表にデータを Upload します。以下のようなストアプロシージャを作成し、引数にデータを Upload する表名を渡し、実行します。

```
CREATE OR REPLACE PROCEDURE UploadData(from_table IN
VARCHAR2) AS
  v_id          order_total.id%TYPE;
  v_date        order_total.order_date%TYPE;
  v_quantity    order_total.quantity%TYPE;
  v_price       order_total.price%TYPE;
  sel_cursor    INTEGER;
  ins_cursor    INTEGER;
  s_row_processed INTEGER;
  i_row_processed INTEGER;
  sql_code      NUMBER;
  sql_msg       VARCHAR2(55);
BEGIN
  sel_cursor := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(sel_cursor,
  'SELECT id, order_date, quantity, price FROM ' || from_table,
  dbms_sql.v7);
  DBMS_SQL.DEFINE_COLUMN(sel_cursor,1,v_id);
  DBMS_SQL.DEFINE_COLUMN(sel_cursor,2,v_date);
  DBMS_SQL.DEFINE_COLUMN(sel_cursor,3,v_quantity);
  DBMS_SQL.DEFINE_COLUMN(sel_cursor,4,v_price);
  s_row_processed := DBMS_SQL.EXECUTE(sel_cursor);
  LOOP
    IF DBMS_SQL.FETCH_ROWS(sel_cursor)>0 THEN
      DBMS_SQL.COLUMN_VALUE(sel_cursor,1,v_id);
      DBMS_SQL.COLUMN_VALUE(sel_cursor,2,v_date);
      DBMS_SQL.COLUMN_VALUE(sel_cursor,3,v_quantity);
      DBMS_SQL.COLUMN_VALUE(sel_cursor,4,v_price);
      ins_cursor := DBMS_SQL.OPEN_CURSOR;
      DBMS_SQL.PARSE(ins_cursor, 'INSERT INTO order_total VALUES
        (:vv_id, :vv_date, :vv_quantity, :vv_price)', dbms_sql.v7);
      DBMS_SQL.BIND_VARIABLE(ins_cursor,'vv_id',v_id);
      DBMS_SQL.BIND_VARIABLE(ins_cursor,'vv_date',v_date);
      DBMS_SQL.BIND_VARIABLE(ins_cursor,'vv_quantity',v_quantity);
      DBMS_SQL.BIND_VARIABLE(ins_cursor,'vv_price',v_price);
      i_row_processed := DBMS_SQL.EXECUTE(ins_cursor);
      DBMS_SQL.CLOSE_CURSOR(ins_cursor);
    ELSE
      EXIT;
    END IF;
  END LOOP;
```

<前ページから>

```
COMMIT;
DBMS_SQL.CLOSE_CURSOR(sel_cursor);
EXCEPTION
WHEN OTHERS THEN
  IF DBMS_SQL.IS_OPEN(sel_cursor) THEN
    DBMS_SQL.CLOSE_CURSOR(sel_cursor);
  END IF;
  sql_code := SQLCODE;
  sql_msg := SUBSTR(SQLERRM,1,55);
  INSERT INTO TEMP VALUES (sql_code,null,sql_msg);
  COMMIT;
END;
/
```

CやC++などでOCI(Oracle Call Interface)を使用した経験のある方にはなじみの深いプログラムの流れだと思います。動的 SQL を行う場合、基本的に、[カーソルのオープン] [文の解析] [入力変数のバインド] [選択リストの記述] [文の実行] [取得したデータのフェッチ] [カーソルのクローズ]というステップを踏まなければなりません。どのような場合に、どのステップが必須になるかはマニュアルの「PL/SQL ユーザーズガイドおよびリファレンス」や市販の関連書籍をご覧ください。

2. 番号を自動的に割り振ってくれる列はどう作成するの？

Microsoft 社の Access や SQLServer を利用している方のほとんど、ID 列 (IDENTITY プロパティなど) を主キーとした表を作成したことがあるかと思います。この ID 列 (オートナンバー型) には、表にデータが挿入されるたびに、その列内で一意な数値が順に自動的に割り振られ、挿入されます。Oracle ではこのオートナンバー型のようなデータ型は存在しません。これを実装するには、シーケンス (順序) とトリガーを用いることとなります。

次に例を示して、この ID 列の実装方法を説明します。実装には 3 つのステップが必要となります。

1. 表の作成
2. シーケンス (順序) の作成
3. トリガーの作成

まず、表を作成します。

```
CREATE TABLE TEST1(  
pk          number(10) primary key,  
name       varchar2(50)  
);
```

次にシーケンス (順序) を作成します。この順序は 1000 から始まって 1 ずつ増えます。

```
CREATE SEQUENCE test_pk1 INCREMENT BY 1 START WITH 1000;
```

最後にトリガーを表に対して作成します。test1 表に対する INSERT および UPDATE 文が発行され、それらが実行される直前に、1 行の処理毎に起動するトリガーです。

```
CREATE OR REPLACE TRIGGER trg_test1  
BEFORE INSERT OR UPDATE on test1  
FOR EACH ROW  
DECLARE  
    iCounter          test1.pk%TYPE;  
    cannot_change_counter EXCEPTION;  
BEGIN  
    IF INSERTING THEN  
        SELECT test_pk1.nextval INTO iCounter FROM dual;  
        :new.pk := iCounter;  
    END IF;
```

```
IF UPDATING THEN
    IF NOT (:new.pk = :old.pk) THEN
        RAISE cannot_change_counter;
    END IF;
END IF;
EXCEPTION
    WHEN cannot_change_counter THEN
        raise_application_error(-20000, 'Cannot Change Counter
Value!');
END;
```

これで ID 列と同じ機能が実装されました。
INSERT 文は、次のようになります。

```
INSERT INTO TEST1(name) VALUES('NAKASHIMA');
INSERT INTO TEST1(name) VALUES('KAWAKAMI');
INSERT INTO TEST1(name) VALUES('YAMADA');
INSERT INTO TEST1(name) VALUES('SUGAHARA');
INSERT INTO TEST1(name) VALUES('HAYASHI');
```

これらのインサート文発行後の表のデータは以下のようになります。

```
SQL> select * from test1;

SQL> pk      name
SQL> -----
SQL> 1000    NAKASHIMA
SQL> 1001    KAWAKAMI
SQL> 1002    YAMADA
SQL> 1003    SUGAHARA
SQL> 1004    HAYASHI
```

注意

Oracle の順序ではシーケンシャルに番号が割り振られますが、一度使った番号を再度割り振ることはできません。この為、欠番が発生します。欠番が発生しない番号を利用する必要がある場合は、別に表(欠番表)などを作成し、機能を実装する必要があります。

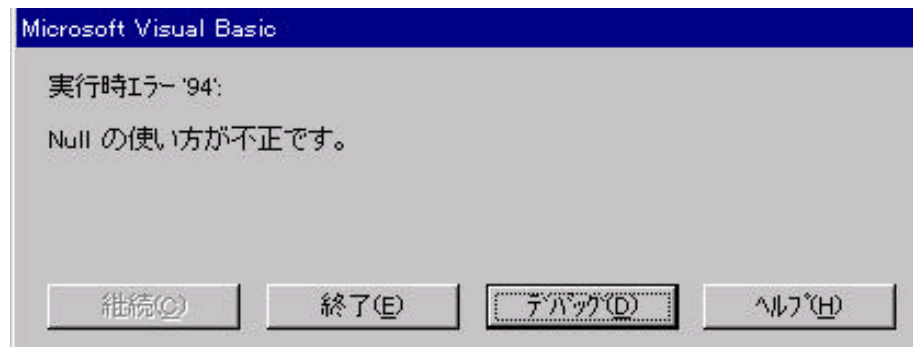
3. Oracle から取得した列の値がNULL の場合はどうするの？

ある表からダイナセットを作成する時に、元となる表のデータに NULL 値が格納されている場合、v_Variable = OraDynaset.Fields(1).Value などでのデータを参照して VB の変数に格納しようとする場合、VB のエラーが発生します。これを回避するために、NULL 値が含まれる可能性のある列(Field オブジェクト)のデータに関しては、変数に格納する前に NULL か否かのチェックをする必要があります。その方法を紹介します。

まず、NULL 値のチェックを行わない場合は、以下のようなコードとなります。

```
Private Sub Command1_Click()  
    Dim sqlstmt As String  
    Dim v_Variable(3) As String  
    Set OraSession = CreateObject("OracleInProcServer.XOraSession")  
    Set OraDatabase = OraSession.OpenDatabase("linux805", "scott/tiger", 0&)  
    sqlstmt = "select ename from emp where empno = 7698"  
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H12&)  
    Set OraFld1 = OraDynaset.Fields(0)  
    v_Variable = OraFld1.Value  
    MsgBox v_Variable
```

この場合、次のようなエラーが返されます。



このエラーを回避するために、コードを次のように書き換えます。

```
Private Sub Command1_Click()  
    Dim sqlstmt As String  
    Dim v_Variable(3) As String  
    Set OraSession = CreateObject("OracleInProcServer.XOraSession")  
    Set OraDatabase = OraSession.OpenDatabase("linux805", "scott/tiger", 0&)  
    sqlstmt = "select ename from emp where empno = 7698"  
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H12&)  
    Set OraFld1 = OraDynaset.Fields(0)  
    If IsNull(OraFld1.Value) Then  
        v_Variable = ""  
    Else  
        v_Variable = OraFld1.Value  
    End If  
    MsgBox v_Variable  
End Sub
```

こうすることにより取り出したデータが NULL の場合でも、String 型変数 v_Variable に長さ 0 の文字列が格納されます。

4. ダイナセットの作成時に制限はあるの？

OO4O ではダイナセットを作成する際の SQL 文の長さに制限があります。利用できる SQL 文の文字列の長さは最大 64K までです。これを超えるとエラーが返されます。

5. 100 行の表の先頭 20 行のダイナセットを作成するには？

ダイナセットを作成する際に、クライアントに Fetch してくる行数をしていることは各種パラメータで指定することができます。しかし、作成するダイナセットそのものの行数を指定するにはどのようにすればよいのでしょうか。これは、SQL 文レベルで実現できます。ROWNUM 疑似列を用います。例えば、scott/tiger の emp 表の最初の 3 件を取得したい場合の SQL 文は次のようになります。

```
SQL> select empno, ename from emp where rownum <= 3;  
  
EMPNO ENAME  
-----  
7369 SMITH  
7499 CLINTON  
7521 WARD
```

0040のダイナセット作成時に指定する SELECT 文を上記のように変えることで、作成されるダイナセットは 3 行しかデータを含まないものになります。

```
SQL> select empno, ename from emp;
```

```
EMPNO ENAME
-----
7369 SMITH
7499 CLINTON
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 KING
7844 TURNER
7876 ADAMS
7900 JAMES
7910 KENJI
```

emp 表のデータ

しかし、この ROWNUM 疑似列を利用する際には注意が必要です。SELECT 文でデータを取り出す際にソート処理を行うために、ORDER BY 句を使うことは頻繁に有ります。この ORDER BY 句と ROWNUM 疑似列は同時に使うことができないのです。同時に使うとどのような結果になってしまうかを説明します。行いたい処理は ENAME 列で ORDER BY を行い、最初の 3 行のみを取得したいとします。

```
EMPNO ENAME
-----
7876 ADAMS
7698 BLAKE
7782 CLARK
```

取得したい結果

先ほどの ROWNUM 疑似列と ORDER BY 句を併用して SELECT 文を実行すると以下のような結果になってしまいます。これは、ROWNUM 疑似列への番号の割当てが ORDER BY によるソートの前に行われ、ソート処理前のデータの最初の 3 行が選択され、その 3 つのデータの中で ORDER BY 句が行われてしまっています。

```
SQL> select empno, ename, rownum from emp
2  where rownum <= 3 order by ename;
```

```
EMPNO ENAME  ROWNUM
-----
7499  CLINTON    2
7369  SMITH      1
7521  WARD       3
```

取得したい結果とは全く異なるものが取得されました。このような検索はアプリケーションを開発する際に頻繁に利用されるものです。では、どのようにして期待通り結果が取得できるのでしょうか。SQL文レベルで実装するには複雑な SELECT 文が必要になります。ここに紹介する方法以外にも様々な方法がありますので、ご自分で頭をひねって考えてみてください。

まず、同一の 2 つの emp 表を単純結合します。結合した表のイメージは以下ようになります。

A.EMPNO	A.ENAME	B.EMPNO	B.ENAME
7369	SMITH	7369	SMITH
7369	SMITH	7499	CLINTON
7369	SMITH	7521	WARD
7369	SMITH	7566	JONES
7369	SMITH	7654	MARTIN
7369	SMITH	7698	BLAKE
7369	SMITH	7782	CLARK
7369	SMITH	7788	SCOTT
7369	SMITH	7839	KING
7369	SMITH	7844	TURNER
7369	SMITH	7876	ADAMS
7369	SMITH	7900	JAMES
7369	SMITH	7910	KENJI
7499	CLINTON	7369	SMITH
7499	CLINTON	7499	CLINTON
...
7521	WARD	7369	SMITH
7521	WARD	7499	CLINTON
...

ORDER BY 句を使って、ソートを行いたいのは ENAME 列ですので、A.ENAME と B.ENAME を比較します。例を挙げて説明します。A.ENAME が SMITH の行には B.ENAME が SMITH ~ KENJI までの 13 行があります。B.ENAME のアルファベット順が A.ENAME の SMITH と同じか若い列だけを取得します。イメージは以下のようになります。

A.EMPNO	A.ENAME	B.EMPNO	B.ENAME
7369	SMITH	7369	SMITH
7369	SMITH	7499	CLINTON
7369	SMITH	7566	JONES
7369	SMITH	7654	MARTIN
7369	SMITH	7698	BLAKE
7369	SMITH	7782	CLARK
7369	SMITH	7788	SCOTT
7369	SMITH	7839	KING
7369	SMITH	7876	ADAMS
7369	SMITH	7900	JAMES
7369	SMITH	7910	KENJI
7499	CLINTON	7900	CLINTON
7499	CLINTON	7910	BLAKE
7499	CLINTON	7369	CLARK
7499	CLINTON	7499	ADAMS

7698	BLAKE	7698	BLAKE
7698	BLAKE	7876	ADAMS
7782	CLARK	7698	BLAKE
7782	CLARK	7782	CLARK
7782	CLARK	7876	ADAMS
7876	ADAMS	7876	ADAMS
7900	JAMES	7499	CLINTON
7900	JAMES	7566	JONES
7900	JAMES	7844	TURNER
7900	JAMES	7698	BLAKE
7900	JAMES	7782	CLARK
7900	JAMES	7876	ADAMS
7900	JAMES	7900	JAMES
7910	KENJI	7499	CLINTON
7910	KENJI	7566	JONES
7910	KENJI	7698	BLAKE
7910	KENJI	7782	CLARK
7910	KENJI	7876	ADAMS
7910	KENJI	7900	JAMES
7910	KENJI	7910	KENJI

このイメージが表しているものは、A.ENAME の値の同じ行の数が A.ENAME 列のアルファベット順でのその値の順番を表していることです。例えば、ADAMS は 1 行しかないなので、A.ENAME 列の値のアルファベット順で 1 番目です。BLAKE は 2 行あるので、2 番目です。同様に KENJI は 7 番目になります。この数値 SQL 文上で求めるには以下のような SQL 文を発行します。

```
SQL> SELECT A.EMPNO, A.ENAME, COUNT(B.ENAME)
2 FROM EMP A, EMP B
3 WHERE A.ENAME >= B.ENAME
4 GROUP BY A.EMPNO, A.ENAME;
```

```
EMPNO ENAME COUNT(B.ENAME)
```

```
-----
7369 SMITH 11
7499 CLINTON 4
7521 WARD 13
7566 JONES 6
7654 MARTIN 9
7698 BLAKE 2
7782 CLARK 3
7788 SCOTT 10
7839 KING 8
7844 TURNER 12
7876 ADAMS 1
7900 JAMES 5
7910 KENJI 7
```

上のイメージを見るとわかるように、A.ENAME のアルファベット順序が COUNT(B.ENAME) に表されています。ここで、ENAME のアルファベット順が 3 番目までを表示したいので、COUNT(B.ENAME) が 3 以下のもののみを検索します。ここでは、GROUP BY 関数を用いているので HAVING 句を使います。

```
SQL> SELECT A.EMPNO, A.ENAME, COUNT(B.ENAME)
2 FROM EMP A, EMP B
3 WHERE A.ENAME >= B.ENAME
4 GROUP BY A.EMPNO, A.ENAME
5 HAVING COUNT(B.ENAME) <= 3;
```

```
EMPNO ENAME COUNT(B.ENAME)
```

```
-----
7698 BLAKE 2
7782 CLARK 3
7876 ADAMS 1
```

これで意図する行のみが検索されてきています。この結果に対して、ORDER BY 句を用います。また、表示したいのは EMPNO 列と ENAME 列のみですので、COUNT(B.ENAME)は除きます。

```
SQL> SELECT A.EMPNO, A.ENAME
2 FROM EMP A, EMP B
3 WHERE A.ENAME >= B.ENAME
4 GROUP BY A.EMPNO, A.ENAME
5 HAVING COUNT(B.ENAME) <= 3
6 ORDER BY A.ENAME;
```

```
EMPNO ENAME
-----
```

```
7876 ADAMS
7698 BLAKE
7782 CLARK
```

これで、ENAME 列でソートしたデータの最初の 3 行のみを検索できました。検索する行数を変更する場合は、“HAVING COUNT(B.ENAME) <= 3”の値を変更してください。

この方法で実装した場合の考慮事項は、この例の EMP 表が非常に大きな表の場合に検索のパフォーマンスに問題が出る可能性があります。

```
実行計画
-----
```

```
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=15 Card=1 Bytes=378)
1  0    FILTER
2  1    SORT (GROUP BY) (Cost=15 Card=1 Bytes=378)
3  2    NESTED LOOPS (Cost=13 Card=7 Bytes=378)
4  3      TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=12 Bytes=480)
5  3      TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=12 Bytes=168)
```

この SELECT 文では EMP 表に対して 2 回の全表検索を行い、それらの表をネストドープにより結合しています。全表検索は非常にコストの高い処理ですので、必要に応じて INDEX などを作成し、パフォーマンスチューニングを行う必要があります。

ヒント SQL*Plus などから簡単に、SQL 文の実行計画を見るには、
SQL> set autotrace on
を発行します。set autotrace on コマンド発行直後の SQL 文から、SQL 文の結果の後に実行計画が表示されます。

*注意

このコマンドを発行する前に、発行するユーザーで utlxplan.sql スクリプト(%ORACLE_HOME%\RDBMS80\ADMIN)を実行する必要があります。

6. ダイナセットが更新可能かどうかを確認するには？

0040 のオンラインマニュアルには以下のように記述されています。

1. SQL 文が単純な列リストまたは列リスト全体 (*) を参照する。
2. SQL 文でオプション引数の読み込み専用フラグを設定しない。
3. 選択された問合せ行に ROWID で参照することを Oracle が許可する。

上記の 3 つの条件がそろった時にダイナセットは更新可能になります。基本的に、ROWID で参照できる場合は更新可能になると考えてください。しかし、すべての SQL 文がこれに該当するかを調査するのは非常に手間のかかる作業なので、ここでは単純に、OraDynaset.Updatable プロパティをチェックすることで、更新可能かどうかを調べます。TRUE の場合は更新可能、FALSE の場合は読み取り専用です。

ヒント ダイナセット作成時のオプションの指定は、作成するダイナセットの用途に合わせることをお勧めします。例えば、ORADYN_READONLY を指定した場合、ダイナセットは読み取り専用になりますが、その分パフォーマンスは向上します。このように適切なオプションを指定することで、VB へのデータの受け渡しの方法を指定したり、パフォーマンスを向上させたりすることが可能です。以下に頻繁に用いられるオプションを紹介합니다。

* ORADYN_NO_BLANKSTRIP

CHAR(10) 型のデータで 5 文字しか含まれない文字列 "scott " (スペース 5 つ) を取り出す際に、後続するスペースを取り除かずに、VB の変数に格納する場合に用いる。(ODBC を利用している場合は、スペースは取り除かれない)

* ORADYN_READONLY

ダイナセットが読み取り専用になる。主にマスター系の表からダイナセットを作成する際に用いる。

* ORADYN_NOCACHE

ダイナセットのデータをクライアントにキャッシュしない。ダイナセット内のデータを参照後、MoveNext などデータを移動した際に、以前に参照したデータをローカルメモリにキャッシュしないため、メモリの節約になり、パフォーマンスが向上する。データを 1 度しか参照せず、配列の変数などに格納してしまう場合に用いる。

7. 表に DEFAULT 値を用いている場合は？

Oracle では DEFAULT 句を用いて列に対するデフォルトを指定することが可能です。その表に対して INSERT 文が発行され、DEFAULT 句を設定した列に値が指定されていない場合は、NULL の代わりに DEFAULT 句で指定した値が INSERT されます。

```
DROP TABLE TEST_DEFAULT;
CREATE TABLE TEST_DEFAULT(
ID          NUMBER(5),
TEST_DEF    NUMBER(10) DEFAULT 10
);

INSERT INTO TEST_DEFAULT (ID, TEST_DEF) VALUES (1, 21);
INSERT INTO TEST_DEFAULT (ID) VALUES (2);

SQL> SELECT * FROM TEST_DEFAULT;
```

ID	TEST_DEF
1	21
2	10

VB から OO4O を経由して Oracle に接続している場合には注意が必要です。

```
Set OraDatabase = OraSession.DbOpenDatabase("", "vb/vb", 0&)
```

上記のように接続した場合、その接続のオプションは 0&、つまり、ORADB_DEFAULT となります。このオプションでは、以下のように動作します。

*AddNew または Edit を使う場合、明示的に設定されていないフィールド (列) の値は NULL に設定される。NULL 値はサーバー列デフォルトを上書きする。

*Edit (「SELECT... FOR UPDATE」) を使う場合は、行ロックを待機する。

*非ブロックの SQL 機能は使えない。

ダイナセットを経由して、INSERT 処理や UPDATE 処理を行った場合は、TEST_DEF に設定されている DEFAULT 値が無効になり、NULL がセットされます。

*VB

```
Private Sub Command4_Click()  
    Dim sqlstmt As String  
    sqlstmt = "select * from test_default"  
    Set OraDynaset = OraDatabase.CreateDynaset(sqlstmt, &H0&)  
    OraDynaset.AddNew  
    OraDynaset.Fields(0).Value = 3  
    OraDynaset.Update  
End Sub
```

*SQL *Plus

```
SQL> SELECT * FROM TEST_DEFAULT;  
  
ID   TEST_DEF  
-----  
1           21  
2           10  
3
```

これを回避するには、OpenDatabase メソッド発行時のオプションに ORADB_ORAMODE を設定します。このモードでは Oracle の DEFAULT 値は有効になり、挿入したデータは以下のようになります。

```
SQL> SELECT * FROM TEST_DEFAULT;  
  
ID   TEST_DEF  
-----  
1           21  
2           10  
3           10
```

その他のオプションに関する詳細はオンラインマニュアルを参照してください。

IV. 参考文献

- 1) 「PL/SQL リリース 8.0 ユーザーズ・ガイドおよびリファレンス」
- 2) 「Oracle8 Enterprise Edition リリース 8.0.5 for Windows NT スタート・ガイド」
- 3) Steven Feuerstein, Bill Pribyl, 「オラクル PL/SQL プログラミング Oracle8 対応版」, オライリー・ジャパン, オーム社, 1998
- 4) 井上岳, 林孝光, 中島謙二, 「Oracle アプリケーション開発テクニック」, ソフト・リサーチ・センター, 1999

この文書はあくまでも参考資料であり、掲載されている情報は予告なしに変更されることがあります。この文書に関連して不都合が生じた場合も、米国オラクル社及び日本オラクル株式会社は一切保証せず、特に責任は負いかねますのでご容赦ください。また許可なく、改編、引用することを禁じます。

1999年4月30日 初版

日本オラクル株式会社 テクノロジーセンター
Design & Migration Services
Copyright© ORACLE CORPORATION JAPAN 1999

