ストアドプロシージャの呼び出し方

この機能は、Release10.5 以降で利用可能です。

Oracle、DataServer、Informix、MS SQL データベースを使用している場合、NXJ アプリケーションからストアドプロシージャの呼び出し、準備、実行を簡単に行える埋込み型 SQL を利用することができます。ネイティブの JDBC 構文の場合とは違って、ストアドプロシージャの呼び出し（ストアドファンクションおよびストアドパッケージも同様）には、追加の Java プログラミングを必要としません。しかしながら、JDBC 構文のネイティブなサポートも、NXJ 環境内で行っています。（注 : 他のデータベースに対する NXJ ストアドプロシージャのサポートは、将来のリリースにおいて追加されます。）

構文の呼び出し

ストアドプロシージャまたはストアドファンクションは、コンパイル時に利用可能であり、メタデータは JDBC ドライバから取得できなければなりません。

ストアドプロシージャ / ファンクションを実行するために、NXJ プログラミング言語スクリプトで以下の構文を使用します。

```
EXEC SQL [USING CONNECTION <connection-name>]
CALL [<schema-name>.][<package-name>.]<function-name>
  ([<argument-list>])
  [RETURNING <return-variable> ]
  [INTO <result-set>|<object-array>|<variable-list>
    ( ;| EXECUTING
      {
      <code-statements>
      }
    )
  ]
```

表 1　　CALL 文の引数 (1 of 2)

構文のエレメント	説明
USING CONNECTION <connection-name>	ストアドプロシージャが実行される接続の名前を指定します。
CALL [<schema-name>.][<package-name>.]<function-name>	実行するストアドファンクションの名前を指定します。スキーマ名はオプションです。指定しない場合は、指定された接続のデフォルトスキーマが使用されます（使用しているデータベースのベンダーが定義する）。プロシージャが Oracle パッケージの中で定義されている場合、プロシージャ名にはそのパッケージ名が指定されなければなりません。
<argument-list>	ストアドファンクションのためのにカンマで区切られた引数のリストです。引数の数およびタイプは、ストアドファンクションで定義されているものと一致しなければなりません。IN パラメータは、Java 式の可能性があり、OUT および IN OUT パラメータは、変数リファレンスでなければなりません。
RETURNING <return-variable>	戻された値を受け取るための変数を指定します。一般的に、この句はリザルトセットでない戻り値を扱うために使用されます。RETURNING 句が指定されて、<return-variable> がリザルトセットである場合、INTO 句が指定されることはありません（指定した場合、コンパイルエラーになります）。 Informix ストアドプロシージャでは、RETURNING 句は使えません。INTO 句を使わなければなりません。

表 1　　CALL 文の引数 (2 of 2)

構文のエレメント	説明
INTO <result-set>|<object-array>|<variable-list>	一般的に、INTO <object-array>|<variable-list> 句は、ストアドプロシージャから戻された値を処理するために、EXECUTING 句とともに使用されます。INTO <result-set> の値がリザルトセットの場合、EXECUTING 句は使用することができません。INTO の値がリザルトセットの場合、EXEC SQL ITERATE <result-set> 句が使用されなければなりません（下記の構文を参照してください）。INTO <result-set> の使用は、戻された変数がリザルトセットである RETURNING <return-variable> の使用と機能的に等しくなります。 INTO 句に、Object[] 変数や変数のリストを指定して、EXECUTING 句が指定されない場合は、リザルトセットの最初の行のデータ値が、Object[] 変数または変数のリストに割り当てられます。 Object[] に行の値を入力する場合、NXJ はリザルトセットの列数に等しい長さで新しい Object[] を割り当てます。インデックス 0 は、1 つ目の列の値に割り当てられ、インデックス 1 は、2 つ目の列の値に割り当てられます。 変数リストに行の値を入力する場合、1 つ目の変数は、1 つ目の列の値に割り当てられ、2 つ目の変数は、2 つ目の列の値に割り当てられます。リストの変数の数とタイプがリザルトセットの列と一致していない場合、ランタイム例外が発生することに注意してください。
EXECUTING { <code-statements> }	リザルトセットの各行のために、一旦、実行されるコード文のブロックを指定します。

注 – リザルトセットを処理するには、INTO <variable-list> EXECUTING 構文を使用することを推奨します。

注 – Oracle を除く、ほとんどの JDBC ドライバはリザルトセットの処理が終わるまで、ストアドプロシージャ / ストアドファンクションの戻り値と OUT パラメータを更新しません。ストアドプロシージャが OUT と RETURN <result-set> の両方を持つ場合、OUT 値は EXECUTING 句で利用できません。

データベースがリザルトセットを処理する前に行う OUT パラメータ受け渡しをサポートする場合、リザルトセットを処理する前に（つまり、あらゆるコード文を実行する前）OUT 変数を割り当てるために以下を行うことができます。

例 : Process RETURN カーソルに INTO <variable list> を使用

```
//stock_price is an AMOUNT field type
// or a variable
// function that has one IN (NUMERIC) and output
// cursor with 3 columns
// String, Amount and Date
    EXEC SQL CALL hr.sp_get_stocks(stock_price)
    INTO ric,price,spupdated
    EXECUTING
    {
    // populating single multi_valued field “nraise”
    // a non-target selected set
    // on Data View “dataview1”
        dataview1.clearToAdd();
        dataview1.nraise = ric + "  "
      + price.toString() + "  "
      + spupdated.toString();
        dataview1.updateCurrentRecord();
    }
```

以下のストアドファンクションの例はカーソルを戻します。

```
CREATE OR REPLACE  FUNCTION "HR"."SP_GET_STOCKS" (v_price IN
    NUMBER)
    RETURN Types.ref_cursor
AS
    stock_cursor types.ref_cursor;
BEGIN
    OPEN stock_cursor FOR
    SELECT ric,price,updated FROM stock_prices
    WHERE  price < v_price;

    RETURN stock_cursor;
END;
```

例 : IN と OUT パラメータ

```
EXEC SQL CALL HR.EMPLOYEE_PKG.GetEmployeeDetails
(1,em_first_name,em_last_name,em_salary,em_start_date);
    session.displayToMessageBox("Response: " +
     em_first_name + " " + em_last_name + " "
     + em_salary + "   " + em_start_date);
```

以下のストアドプロシージャの例は、IN パラメータと OUT パラメータを使用します。

```
  PROCEDURE GetEmployeeDetails(
  i_emID INemployee.em_id%TYPE,
    o_FirstName  OUT employee.em_first_name%TYPE,
    o_LastNameOUTemployee.em_last_name%TYPE,
    o_SalaryOUTemployee.em_salary%TYPE,
    o_StartDateOUTemployee.em_start_date%TYPE)
  IS
  BEGIN
  SELECT em_first_name,
         em_last_name,
         em_salary,
         em_start_date
    INTO o_FirstName,
         o_LastName,
         o_Salary,
         o_StartDate
    FROM employee
   WHERE em_id = i_emID;
  END GetEmployeeDetails;
```

リザルトセットは ITERATE 句でトラバースする

以下の構文は、OUT パラメータとして 1 つ以上のリザルトセットを持つプロシージャを実行するときに参考になります。

```
EXEC SQL
      ITERATE <resultset-variable-name>
      INTO (<variable-list>|<object-array>)
      ( ; |
```

```
        EXECUTING
        {
      <code-statements>
        }
      )
```

構文引数の説明については、2 **ページの表** 1 「CALL **文の引数**」 **を参照してください。**

以下の例は、ITERATE **で** RETURNING <result-set> **カーソルを使用します。**

```
  // Variables defined on Form
  //    NullableStringVariable ric;
    //    NullableAmountVariable price;
    //    NullableDateVariable spupdated;
// Declare ResultSet “rs” for RETURNING
    ResultSet rs;
    EXEC SQL CALL HR.sp_get_stocks(stock_price)
    RETURNING rs;
      // Iterate through the returned result set
    EXEC SQL ITERATE rs
    INTO ric,price,spupdated
    EXECUTING
    {
     // populating single multi_valued field “nraise”
     // a non-target selected set
     // on Data View “dataview1”
         dataview1.clearToAdd();
         dataview1.nraise = ric + "  "
       + price.toString() + "  "
       + spupdated.toString();
         dataview1.updateCurrentRecord();
    }
```

ダイナミックストアドプロシージャ

多くのストアドプロシージャは、動的に実行時に準備、実行される必要があります。

例えば、OUT **パラメータとして** Oracle INDEX BY TABLE **を使用する場合、文はダイナミック構文を使用して準備されなければなりません。これにより、**IN **と** OUT **パラメータを設定、取得が可能になります。**Dynamic **パラメータの詳細と構文に関しては、**NXJ Javadoc **の** NXJParameter **を参照してください。**

注 – INDEX BY TABLE **の** OUT **パラメータを持つ** Oracle **のダイナミックストアドプロシージャの使用は、**OCI JDBC **インタフェースの使用を必要とします。**OCI **インタフェースを使用するには、アプリケーションサーバ上に** Oracle **クライアントのインストールが必要になります。**

構成 :

```
Type: Other JDBC Databases
  Jar/Zip File: C:\Unify\NXJ\lib\jdbcDrivers\ojdbc14.zip
  User Name: xxxxx     Password : xxxx
  Driver: odbc.jdbc.driver.OracleDriver
  URL: jdbc:oracle:oci:@servername
```

“servername” **は、**Oracle Network Client **で構成されなければなりません。**

PREPARE CALL **構文**

PREPARE CALL **文は、以下の構文を持ちます。**

```
EXEC SQL [ USING CONNECTION <connection-name> ]
      PREPARE CALL <string-expression>
      INTO <NXJPreparedCall>;
```

表 2　　PREPARE CALL 文の引数

構文エレメント	説明
USING CONNECTION <connection-name>	ストアドプロシージャが実行される接続名。
CALL <string-expression>	ストアドプロシージャの名前の文字列式。実行時にストアドプロシージャのメタデータが明らかに取得できない場合、例外がスローされます。
INTO <NXJPreparedCall>	メタデータ情報のクエリ、およびストアドプロシージャを実行するためのハンドルを提供する NXJPreparedCall 変数。

例

```
NXJPreparedCall myCall;
   EXEC SQL USING CONNECTION ociOracle PREPARE CALL
  "HR.EMPLOYEE_PKG.GetEmployeeList" INTO myCall;
```

注 – EXEC SQL EXECUTE 文を呼び出す前に、registerIndexTableOutParameter() 呼び出しを使用して OUT パラメータを登録しなければなりません。未登録の PL/SQL インデックステーブル OUT パラメータがある場合、EXEC SQL EXECUTE 文は、SQLException をスローします。

```
// Get the NXJ Parameters for the prepared call
      NXJParameter[] params = myCall.getOutputParameterData();
// you must register the estimated array size
// value for each parameter in this example there are 3 OUT
parameters.
      params[0].registerIndexTableOutParameter(100);
```

```
params[1].registerIndexTableOutParameter(100);
params[2].registerIndexTableOutParameter(100);
```

EXECUTE 構文

ストアドプロシージャが準備されて、OUT パラメータが登録された後、ストアドプロシージャは以下の構文を使用して実行することができます。

```
EXEC SQL
   EXECUTE <NXJPreparedCall>
   [ USING (<input-object-array>|<expression-list>)
   [ RETURNING (<result-set>|<output-object-array.|
              <variable-list>)
   ;
```

表 3　　EXECUTE 文の引数

構文エレメント	説明
EXECUTE <NXJPreparedCall>	実行する NXJPreparedCall を指定します。
USING (<input-object-array>|<expression-list>)	ストアドプロシージャの IN パラメータに値を提供するために使用します。Object[] が提供される場合、各 IN パラメータごとに 1 つのエレメントがなければなりません。カンマで区切られた式のリストが提供される場合、各 IN パラメータごとに 1 つの式がなければなりません。
INTO (<result-set>|<output-object-array>|<variable-list>)	ストアドプロシージャの OUT パラメータ用の値をどこに置くかを指定するために使用します。 変数のリストの場合、ストアドプロシージャから戻される各 OUT パラメータごとに 1 つの変数がなければなりません。変数の数とタイプは、ストアドプロシージャからの出力に一致しなければなりません。さもなければ、ランタイム例外がスローされます。変数のリストは、リザルトセット変数を含みます。

Object[] が指定される場合、NXJ は OUT パラメータの数に等しい長さで新しい Object[] を割り当てます。インデックス 0 は、1 つ目の OUT パラメータの値に割り当てられます。インデックス 1 は、2 つ目の OUT パラメータの値に割り当てられます。

ストアドプロシージャが戻り値を持つ場合、最初の OUT パラメータが割り当てられると考えられます。

例

PREPARE のサンプルストアドプロシージャ。

```
// Package
  TYPE tblEMID IS TABLE OF NUMBER(3)INDEX BY BINARY_INTEGER;
  TYPE tblFirstName IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
  TYPE tblLastName  IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
  TYPE tblSalary    IS TABLE OF NUMBER(6)    INDEX BY BINARY_INTEGER;
  TYPE tblStartDate IS TABLE OF DATE         INDEX BY BINARY_INTEGER;

  PROCEDURE GetEmployeeList(
    o_emID OUT tblEMID,
    o_FirstName OUT tblFirstName,
    o_LastNameOUTtblLastName)

// Package Body
  PROCEDURE GetEmployeeList(
    o_emID   OUT  tblEMID,
    o_FirstNameOUT tblFirstName,
    o_LastNameOUTtblLastName)
  IS
    CURSOR employee_cur  IS
      SELECT em_id,em_first_name, em_last_name
        FROM employee;
    recCount NUMBER DEFAULT 0;
  BEGIN
    FOR EmployeeRec IN employee_cur LOOP
      recCount:= recCount + 1;
      o_emID(recCount):= EmployeeRec.em_id;
      o_FirstName(recCount):= EmployeeRec.em_first_name;
      o_LastName(recCount):= EmployeeRec.em_last_name;
    END LOOP;
  END GetEmployeeList;
```

EXEC SQL PREPARE CALL **を使用する** NXJ **例。**

```
// using TABLE INDEX BY return values:
// i.e.  TYPE tblEMID  IS TABLE OF NUMBER(3)INDEX BY BINARY_INTEGER;
// NOTE: Oracle TABLE INDEX BY requires the OCI JDBC connection
// which requires the Oracle Client be installed on the
// Application Server.
// Configuration is:
   //  Type: Other JDBC Databases
   //  Jar/Zip File: C:\Unify\NXJ\lib\jdbcDrivers\ojbdbc14.jar
   //  User Name: xxxxx     Password : xxxx
   //  Driver: odbc.jdbc.driver.OracleDriver
   //  URL: jdbc:oracle:oci:@servername
   //  NOTE: This server MUST be configured in the Oracle Network
   // Client

// Define the NXJ Prepared Call
  NXJPreparedCall call;
     EXEC SQL USING CONNECTION ociOracle PREPARE CALL
  "HR.EMPLOYEE_PKG.GetEmployeeList" into call;
// Get the NXJ Parameters for the prepared call
     NXJParameter[] params = call.getOutputParameterData();
// you must register the estimated array size
// value for each parameter
    params[0].registerIndexTableOutParameter(100);
    params[1].registerIndexTableOutParameter(100);
    params[2].registerIndexTableOutParameter(100);
// Define variables for the return values
   int[] firstArray;
   String[] secondArray;
   String[] thirdArray;
// Execute the call
   EXEC SQL EXECUTE call RETURNING firstArray, secondArray,
      thirdArray;
// Loop through the returned arrays and in this example
//   add them to a selected set.
   for (int i = 0; i < firstArray.length; i++)
   {
       testview1.clearToAdd();
       testview1.nraise = secondArray[i] + " " + thirdArray[i];
```

```
       testview1.updateCurrentRecord();
    }
// Close the prepared call
   call.close();
```

ダイナミックパラメータ

NXJPreparedCall と NXJParameter は、事前に準備された文で利用されるクエリストアドプロシージャメタデータにインタフェースを提供する NXJ クラスです。これらのインタフェースは、以下のメソッドを定義します。

NXJPrepreparedCall

NXJPreparedCall
    Void close()

例 :

```
  NXJPrepareCall sp_call;
  Sp_call.close();
```

NXJParameter[] getInputParameterData

NXJParameter[] getInputParameterData()
  throws SQLException;

ストアドプロシージャへの入力パラメータのメタデータを取得します。

例 :

```
NXJParameter[] params = call.getInputParameterData();
```

NXJParameter[] getOutputParameterData

NXJParameter[] getOutputParameterData()
    throws SQLException;

ストアドプロシージャの出力パラメータのメタデータを取得します。プロシージャが戻り値を持つ場合、配列の最初のエレメントがそのメタデータを提供します。ストアドプロシージャが暗黙的なリザルトセットを返す場合、配列の最初のエレメントがそのメタデータを提供します。ストアドプロシージャが暗黙的なリザルトセットと整数の戻り値の両方を持つ場合、暗黙的なリザルトセットは最初のエレメントがそのメタデータを提供します。整数の戻り値
（getVendorReturnCode() **によって取得できる**）**があるかどうかを決定するために、**hasVendorReturnCode() **を使用します。**

hasVendorReturnCode()

boolean hasVendorReturnCode();

この NXJPreparedCall **インスタンスが、暗黙的なリザルトセットと整数の戻り値の両方を返すストアドプロシージャを呼び出す場合、**true **を返します。**

getVendorReturnCode ()

int getVendorReturnCode();

ストアドプロシージャのために整数の戻り値を返します。hasVendorReturnCode() **が** false **を返す場合、**IllegalStateException **をスローします。**

NXJParameter

以下のメソッドは、メタデータ情報のためにあります。

String getName();

- **パラメータの名前を得ます。**

int getSqlType();

- パラメータの java sql タイプを得ます。

int getExtendedType();

パラメータの拡張されたタイプを得ます。標準タイプの場合、NXJExtendedSQLType.NOT_EXTENDED が返されます。

String getTypeName();

- データベースに識別されているタイプ名を得ます。

int getLength();

- このパラメータが保持できるデータのバイト長を得ます。

intgetNullable();

- 以下の内の 1 つが返ります。

NXJParameter.NO_NULLS

NXJParameter.NULLABLE

NXJParameter.NULLABLE_UNKNOWN

intgetPrecision();

- パラメータの精度を得ます。

shortgetRadix();

- パラメータの基数を得ます。

shortgetScale();

- パラメータのスケールを得ます。

以下のメソッドは、Oracle の PL/SQL インデックステーブル用のものであり、OUT パラメータが PL/SQL インデックステーブルの場合にのみ使用することができます。つまり、OUT パラメータの NXJParameter.getExtenedType() は、NXJExtentedSQLTypes.INDEX_BY_TABLE を返します。

int getIndexTableElemSqlType();

- インデックステーブル内でエレメントの sql タイプを得ます。

int getIndexTableElemMaxLen();

- インデックステーブル内でエレメントの最大の長さを得ます。

エレメント sql タイプが VARCHAR, CHAR, BINARY でない場合、この値は 0 です。

registerOracleArrayOutParameter

registerOracleArrayOutParameter() は、OUT Oracle 配列パラメータ毎に呼び出されなければなりません。これらの配列は、Oracle Array タイプの “User Types-> Array Types” です。

例 :

```
outputParameterData[0].registerOracleArrayOutParameter("MY_ARRAY");
```

registerIndexTableOutParameter

registerIndexTableOutParameter() メソッドは、NXJParameter.getExtenedType() メソッドが NXJExtentedSQLTypes.INDEX_BY_TABLE を返すすべての OUT パラメータ毎に呼び出されなければなりません。NXJ が EXEC SQL EXECUTE 文を処理するとき、未登録のインデックステーブル OUT パラメータがある場合、SQLException をスローします。

void registerIndexTableOutParameter(int maxLen);

int maxLen - これは、配列の最大サイズが返されることを示します。

NXJParameter についての詳細は、Javadoc を参照してください。

このインタフェースは、NXJ によってサポートされる標準ではない SQL タイプを定義します。これらの値は、NXJParameter.getExtendedType() によって返されます。このリリースには、以下の拡張されたタイプがあります。

NOT_EXTENDED

- NXJParameter が、NXJ によってサポートされる拡張された SQL タイプではないことを示します。ユーザは、パラメータタイプを識別するために、NXJParameter.getSqlType() を使用する必要があります。

RESULT_SET

- NXJParameter が、ResultSet であることを示します。

INDEX_BY_TABLE

- NXJParameter が、Oracle PL/SQL インデックステーブルであることを示します。

例:

```
NXJPreparedCall sp_call;
    EXEC SQL USING CONNECTION ociOracle
        PREPARE CALL "HR.EMPLOYEE_PKG.GetEmployeesAboveSalary"
  INTO sp_call;
    NXJParameter[] inparams = sp_call.getInputParameterData();
    session.displayToMessageBox("input: " +
inparams[0].getSqlType());
    NXJParameter[] outparams = sp_call.getOutputParameterData();
    for (int index = 0; index < outparams.length; index++)
    {
        NXJParameter outparam1 = outparams[index];
   session.displayToMessageBox(" OUT: " + param1.getName()
  + " Type: " + outparam1.getIndexTableElemSqlType());

        if (outparam1.getExtendedType() ==
com.unify.nxj.mgr.dataConnection.NXJExtendedSQLTypes.INDEX_BY_TABLE)
        {
            outparam1.registerIndexTableOutParameter(1000);
        }
    }
    // Define variables for the return values
    int[] o_emID;
    int myinput = 10000;
    String[] o_FirstName;
    String[] o_lastName;
    float[] o_Salary;
    java.sql.Timestamp[] o_startDate;
```

グローバル変数

以下のように、パッケージのグローバル変数（特にオラクル）にアクセスすることができます。

```
EXEC SQL [ USING CONNECTION <connection-name> ]
        GET VAR <string-expression>
        INTO <return-variable>;
```

表 4　　GET VAR 文引数

構文エレメント	説明
GET VAR <string-expression>	戻されるパッケージ変数（例えば、"MY_PACKAGE.MY_VARIABLE") を識別する文字列式。
INTO <return-variable>	戻される変数。

以下のように、パッケージのグローバル変数（特にオラクル）を設定します。

```
EXEC SQL [ USING CONNECTION <connection-name> ]
        SET VAR <string-expression>
        USING <input-expression>;
```

表 5　　SET VAR 文引数

構文エレメント	説明
SET VAR <string-expression>	設定されるパッケージ変数（例えば、"MY_PACKAGE.MY_VARIABLE"）を識別する文字列式。
USING <input-expression>	パッケージ変数に割り当てる値。

注 – コンパイル時には、グローバル変数の存在有無、またはグローバル変数とホスト変数タイプ間のデータタイプの不整合をチェックすることができません。代わりに、これらのエラー状況は、ランタイム例外に終わるでしょう。従って、このようなエラーの場合には実行時の例外が発生してしまいます。

例 :

```
// As defined in Oracle Package: EMPLOYEE_PKG
   // myString VARCHAR2(40);
   String setfield = “hello UNIFY”;
```

```
EXEC SQL
  SET VAR "HR.EMPLOYEE_PKG.myString"
  USING setfield;

 EXEC SQL
  GET VAR "HR.EMPLOYEE_PKG.myString"
  INTO setfield;
 session.displayToMessageBox(setfield);
```

Oracle の raise_application_error()

Oracle ストアドプロシージャが、raise_application_error() を呼び出すときは常に、Oracle JDBC ドライバによってスローされている SQLException で終わります。例外メッセージは、詳細なメッセージを提供します。SQLException.getErrorCode() を使って、raise_application_error() に渡されるエラーコードを取得することができます。

その他の Oracle 例

Oracle ストアドプロシージャとの間でデータを受け渡しする 1 つの方法として、カーソルを利用することができます。カーソル変数の PL/SQL タイプは、REF CURSOR です。例 :

```
CREATE OR REPLACE PACKAGE types
AS
    TYPE ref_cursor IS REF CURSOR;
END;
```

カーソルは、1 つ以上の行を返す SQL 文の作業領域です。それは、リザルトセットから 1 つずつ行をフェッチすることを可能にします。カーソルの使用は特に難しいことはありませんが、PL/SQL ストアドプロシージャからリザルトセットを返すためにカーソル変数を使用しなければなりません。カーソル変数は、基本的にカーソルのポインタなので、ストアドプロシージャへのパラメータのように、カーソルそのものではありませんがそのリファレンスとして同様に扱うことができます。

NXJ の 2 つの異なる方法で PL/SQL カーソルを利用することができます。1 つ目は、自動的に NXJ 変数にカーソルの列の割り当てを行ないます。2 つ目は、NXJ リザルトセットにカーソルを割り当てた後に、リザルトセットによって割り当てを繰り返すか、さもなければカーソルの列を直接操作します。

Oracle スクリプト

```
CREATE TABLE STOCK_PRICES(
  RIC VARCHAR(6) PRIMARY KEY,
  PRICE NUMBER(7,2),
  UPDATED DATE )
  TABLESPACE USERS
/
INSERT INTO STOCK_PRICES(RIC, PRICE, UPDATED)
VALUES('MSFT', 69.20, SYSDATE)
/
INSERT INTO STOCK_PRICES(RIC, PRICE, UPDATED)
VALUES('RSAS', 30.18, SYSDATE)
/
INSERT INTO STOCK_PRICES(RIC, PRICE, UPDATED)
VALUES('AMZN', 15.50, SYSDATE)
/
INSERT INTO STOCK_PRICES(RIC, PRICE, UPDATED)
VALUES('SUNW', 16.25, SYSDATE)
/
INSERT INTO STOCK_PRICES(RIC, PRICE, UPDATED)
VALUES('ORCL', 14.50, SYSDATE)
/
COMMIT
/
CREATE OR REPLACE PACKAGE Types
AS
    TYPE ref_cursor IS REF CURSOR;
END;
/

CREATE OR REPLACE FUNCTION sp_get_stocks(v_price IN NUMBER)
    RETURN types.ref_cursor
AS
    stock_cursor types.ref_cursor;
BEGIN
    OPEN stock_cursor FOR
        SELECT ric,price,updated FROM stock_prices
```

```
        WHERE price < v_price;

    RETURN stock_cursor;
END;
```

NXJ コマンド

```
COMMAND sp_get_stocks
{
    //stock_price is an AMOUNT field type or a variable
    // Variables “ric”,”price”, and “spupdated” are defined as
    // NullableStringVariable ric;
    // NullableAmountVariable price;
    // NullableDateVariable spupdated;
    EXEC SQL CALL hr.sp_get_stocks(stock_price)
    INTO ric,price,spupdated
    EXECUTING
    {
        session.displayToMessageBox(": " + ric + "  " + price + "  "
+ spupdated);
    }
}

COMMAND sp_get_stocks2
{
    DataViewHelper.clearSet(testview1); // this is just clearing out
the selected set
    // Define the Result Set
    ResultSet rs;
    EXEC SQL CALL HR.sp_get_stocks(stock_price)
//  NOTE: Can also use RETURNING rs;
    INTO rs;
      // Iterate through the returned result set
    EXEC SQL ITERATE rs
    INTO ric,price,spupdated
    EXECUTING
    {
        testview1.clearToAdd();
        testview1.nraise = ric + "  " + price.toString() + "  " +
spupdated.toString();
        testview1.updateCurrentRecord();
    }
```

```
}
```

Oracle INDEX_BY_TABLE : パッケージスクリプト

```
CREATE OR REPLACE  PACKAGE "HR"."EMPLOYEE_PKG" AS
  TYPE tblEMID      IS TABLE OF NUMBER(3)    INDEX BY
BINARY_INTEGER;
  TYPE tblFirstName IS TABLE OF VARCHAR2(30) INDEX BY
BINARY_INTEGER;
  TYPE tblLastName  IS TABLE OF VARCHAR2(30) INDEX BY
BINARY_INTEGER;
  TYPE tblSalary    IS TABLE OF NUMBER(6)    INDEX BY
BINARY_INTEGER;
  TYPE tblStartDate IS TABLE OF DATE         INDEX BY
BINARY_INTEGER;
  myNumber NUMBER;
  myString VARCHAR2(40);

  PROCEDURE GetEmployeeList(
      o_emID           OUT tblEMID,
    o_FirstName      OUT tblFirstName,
    o_LastName     OUT tblLastName);

  PROCEDURE GetEmployeesAboveSalary(
 i_MinimumSalary   INemployee.em_salary%TYPE,
      o_emID       OUT tblEMID,
    o_FirstName OUT tblFirstName,
    o_LastNameOUTtblLastName,
    o_SalaryOUTtblSalary);


/*****************************************************************
  * Retrieves a single employees details using standard arguments

*****************************************************************/
  PROCEDURE GetEmployeeDetails(
i_emID            INemployee.em_id%TYPE,
    o_FirstName OUT         employee.em_first_name%TYPE,
    o_LastNameOUTemployee.em_last_name%TYPE,
    o_SalaryOUTemployee.em_salary%TYPE,
    o_StartDateOUTemployee.em_start_date%TYPE);
```

```
END Employee_Pkg;
```

Oracle INDEX_BY_TABLE: パッケージボディスクリプト

```
CREATE OR REPLACE  PACKAGE BODY "HR"."EMPLOYEE_PKG" AS

  PROCEDURE GetEmployeeList(
      o_emID         OUT      tblEMID,
    o_FirstName    OUT tblFirstName,
    o_LastName     OUTtblLastName)
  IS
    CURSOR employee_cur  IS
      SELECT em_id,em_first_name, em_last_name
        FROM employee;
    recCount NUMBER DEFAULT 0;
  BEGIN
    FOR EmployeeRec IN employee_cur  LOOP
      recCount             := recCount + 1;
      o_emID(recCount)     := EmployeeRec.em_id;
      o_FirstName(recCount):= EmployeeRec.em_first_name;
      o_LastName(recCount) := EmployeeRec.em_last_name;

    END LOOP;

  END GetEmployeeList;

  PROCEDURE GetEmployeesAboveSalary(
 i_MinimumSalary   INemployee.em_salary%TYPE,
      o_emID       OUT tblEMID,
    o_FirstName       OUT tblFirstName,
    o_LastNameOUTtblLastName,
    o_SalaryOUTtblSalary)
  IS
    CURSOR employee_cur (curMinSalary NUMBER) IS
      SELECT em_id,
             em_first_name, em_last_name,
             em_salary, em_start_date
        FROM employee
       WHERE em_salary > curMinSalary;
     recCount NUMBER DEFAULT 0;

BEGIN
```

```
     FOR EmployeeRec IN employee_cur(i_MinimumSalary) LOOP
       recCount:= recCount + 1;
       o_emID(recCount)     := EmployeeRec.em_id;
       o_FirstName(recCount):= EmployeeRec.em_first_name;
       o_LastName(recCount) := EmployeeRec.em_last_name;
       o_Salary(recCount)   := EmployeeRec.em_salary;
     END LOOP;
  END GetEmployeesAboveSalary;


/*****************************************************************
  * Retrieves a single employees details using standard arguments

*****************************************************************/
  PROCEDURE GetEmployeeDetails(
i_emID INemployee.em_id%TYPE,
    o_FirstName  OUT       employee.em_first_name%TYPE,
    o_LastName OUTemployee.em_last_name%TYPE,
    o_Salary OUTemployee.em_salary%TYPE,
    o_StartDate OUTemployee.em_start_date%TYPE)
  IS
  BEGIN
 SELECT em_first_name,
        em_last_name,
        em_salary,
        em_start_date
    INTO o_FirstName,
         o_LastName,
         o_Salary,
         o_StartDate
    FROM employee
   WHERE em_id = i_emID;
  END GetEmployeeDetails;
END Employee_Pkg;
```

NXJ コマンド

```
// ********************* GetEmployeeList ********************
// Define the NXJ Prepared Call
  NXJPreparedCall call;
     EXEC SQL USING CONNECTION ociOracle PREPARE CALL
"HR.EMPLOYEE_PKG.GetEmployeeList" into call;
```

```
// Get the NXJ Parameters for the prepared call
     NXJParameter[] params = call.getOutputParameterData();
// you must register the estimated array size
// value for each parameter
    params[0].registerIndexTableOutParameter(100);
    params[1].registerIndexTableOutParameter(100);
    params[2].registerIndexTableOutParameter(100);
// Define variables for the return values
   int[] firstArray;
   String[] secondArray;
   String[] thirdArray;
// Execute the call
   EXEC SQL EXECUTE call RETURNING firstArray, secondArray,
thirdArray;
// Loop through the returned arrays and in this example
// add them to a selected set.
   for (int i = 0; i < firstArray.length; i++)
   {
       testview1.clearToAdd();
       testview1.nraise = secondArray[i] + " " + thirdArray[i];
       testview1.updateCurrentRecord();
    }
// Close the prepared call
   call.close();

//************** GetEmployeesAboveSalary *************

   NXJPreparedCall sp_call;
    EXEC SQL USING CONNECTION ociOracle
      PREPARE CALL "HR.EMPLOYEE_PKG.GetEmployeesAboveSalary"
into sp_call;
    NXJParameter[] inparams = sp_call.getInputParameterData();
    NXJParameter[] params = sp_call.getOutputParameterData();
for (int index = 0; index < params.length; index++)
    {
        NXJParameter param1 = params[index];
        if (param1.getExtendedType() ==
com.unify.NXJ.mgr.dataConnection.NXJExtendedSQLTypes.INDEX_BY_TABLE)
        {
            param1.registerIndexTableOutParameter(1000);
        }
    }
    // Define variables for the return values
    int[] o_emID;
    int myinput = 10000;
    String[] o_FirstName;
```

```
    String[] o_lastName;
    float[] o_Salary;

    EXEC SQL EXECUTE sp_call
        USING myinput
        RETURNING o_emID,o_FirstName,o_lastName,o_Salary;
    DataViewHelper.clearSet(testview1);
// Loop through the returned arrays and in this example
//   add them to a selected set.
    for (int i = 0; i < o_emID.length; i++){
        testview1.clearToAdd();
        testview1.nraise = o_FirstName[i] + " " + o_lastName[i]
+ " " + o_Salary[i];
        testview1.updateCurrentRecord();
    }
// Close the prepared call

    sp_call.close();

//************** GetEmployeesList *************

 NXJPreparedCall call;
 EXEC SQL USING CONNECTION ociOracle
PREPARE CALL "HR.EMPLOYEE_PKG.GetEmployeeList"
INTO call;
// Get the NXJ Parameters for the prepared call
 NXJParameter[] params = call.getOutputParameterData();
// you must register the estimated array size
// value for each parameter
for (int index = 0; index < params.length; index++)
{
   NXJParameter param1 = params[index];
   if (param1.getExtendedType() ==
com.unify.NXJ.mgr.dataConnection.NXJExtendedSQLTypes.INDEX_BY_TABLE)
   {
       param1.registerIndexTableOutParameter(1000);
   }
}

// Define variables for the return values
int[] firstArray;
String[] secondArray;
String[] thirdArray;
// Execute the call
EXEC SQL
EXECUTE call
```

```
RETURNING firstArray, secondArray, thirdArray;
// Loop through the returned arrays and in this example
//   add them to a selected set.
for (int i = 0; i < firstArray.length; i++)
{
  testview1.clearToAdd();
  testview1.nraise = secondArray[i] + " " + thirdArray[i];
  testview1.updateCurrentRecord();
}
// Close the prepared call
call.close();
```

MS SQL の制限

NXJ アプリケーションで使用される MS SQL のストアドプロシージャには以下の使用上の制限があります。

- ストアドプロシージャには、compute 文を含めることはできません。
- ストアドプロシージャには、複数のリザルトセットを返すことはできません。（すなわち、複数の SELECT 文を使ってはいけません）
- NXJ は、ダイナミック CALL 文の RETURNING 句で指定される最初の変数はリザルトセットであるものとして扱います。

  もしストアドプロシージャがリザルトセットを返さない場合には、最初の変数には Null が設定されなければなりません。この場合は、リザルトセットから行データを取得する前にアプリケーションは最初の変数が Null かどうかをチェックする必要があります。