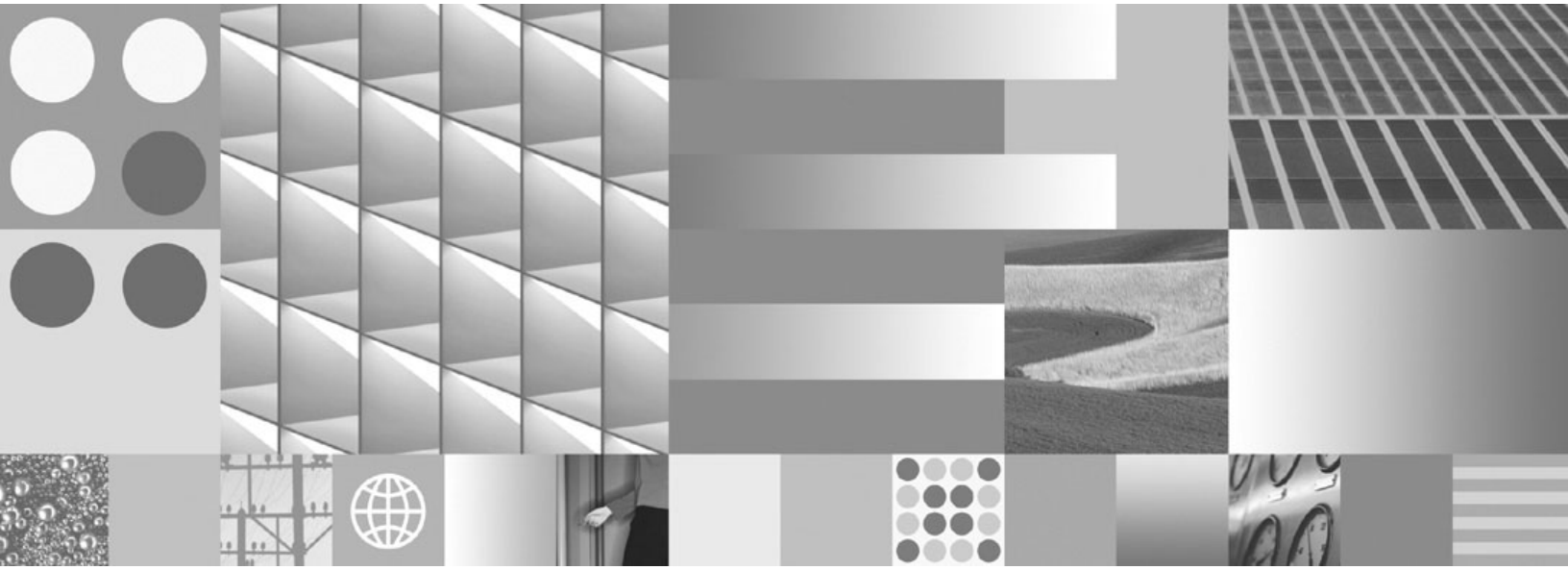
IBM DB2 9.7
for Linux, UNIX, and Windows

バージョン 9 リリース 7



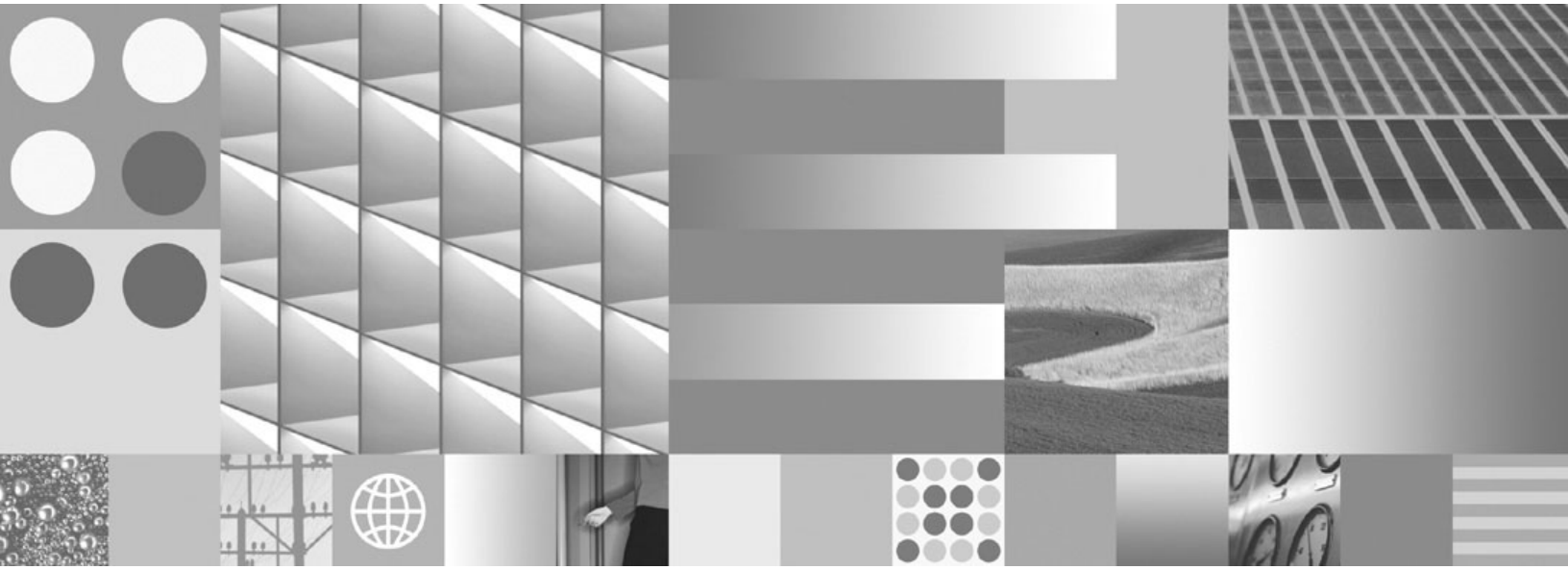
SQL プロシージャー言語: アプリケーションのイネーブルメントおよびサポート

2010 年 9 月更新版

SC88-5877-02
(英文原典：SC27-2470-02)

IBM DB2 9.7
for Linux, UNIX, and Windows

バージョン 9 リリース 7



SQL プロシージャー言語: アプリケーションのイネーブルメントおよびサポート

2010 年 9 月更新版

SC88-5877-02
(英文原典：SC27-2470-02)

ご注意

本書および本書で紹介する製品をご使用になる前に、451 ページの『付録 B. 特記事項』に記載されている情報をお読みください。

IBM 資料は、オンラインでご注文いただくことも、ご自分の国または地域の IBM 担当員を通してお求めいただくこともできます。

- オンラインで資料を注文するには、www.ibm.com/shop/publications/order にある IBM Publications Center をご利用ください。
- ご自分の国または地域の IBM 担当員を見つけるには、www.ibm.com/planetwide にある IBM Directory of Worldwide Contacts をお調べください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC27-2470-02
IBM DB2 9.7
for Linux, UNIX, and Windows
Version 9 Release 7
SQL Procedural Languages: Application Enablement and Support
Updated September, 2010

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2010.9

目次

第 1 章 SQL プロシージャー型言語 (SQL PL)

SQL プロシージャー型言語 (SQL PL) は SQL の言語拡張であり、SQL ステートメントでプロシージャー・ロジックをインプリメントするために使用できるステートメントと言語エレメントから成っています。 SQL PL は、変数や条件ハンドラーを宣言するためのステートメント、変数に値を割り当てるためのステートメント、およびプロシージャー・ロジックをインプリメントするためのステートメントを提供します。

インライン SQL PL

インライン SQL PL は SQL PL フィーチャーのサブセットであり、コンパウンド SQL (インライン化) ステートメントで使用することができます。コンパウンド SQL (インライン化) ステートメントは、単独で実行したり、トリガー、SQL 関数、または SQL メソッドの本体のインプリメントに使用したりできます。 DB2® CLP が基本 SQL スクリプト言語のサポートを提供する対話モードの場合は、ここからコンパウンド SQL (インライン化) ステートメントを単独で実行できます。

インライン SQL PL が「インライン」と記述されるのは、このロジックが、それを参照する SQL ステートメントに展開され、この SQL ステートメントを使って実行されるからです。

以下の SQL PL ステートメントは、インライン SQL PL ステートメントのセットの一部とみなされます。

- 変数に関連したステートメント
  - DECLARE <variable>
  - DECLARE <condition>
  - SET ステートメント (代入文)
- 条件ステートメント
  - IF
  - CASE 式
- ループ・ステートメント
  - FOR
  - WHILE
- 制御転送ステートメント
  - GOTO
  - ITERATE
  - LEAVE
  - RETURN
- エラー管理ステートメント
  - SIGNAL
  - GET DIAGNOSTICS

SQL プロシージャーでサポートされるこれ以外の SQL PL ステートメントは、コンパウンド SQL (インライン化) ステートメントではサポートされません。カーソルおよび条件ハンドラーはインライン SQL PL ではサポートされないため、RESIGNAL ステートメントもサポートされません。

インライン SQL PL ステートメントはコンパウンド SQL (インライン化) ステートメントで実行する必要があるため、PREPARE、EXECUTE、または EXECUTE IMMEDIATE ステートメントに対するサポートはありません。

また ATOMIC は、動的に準備または実行されるコンパウンド SQL (インライン化) ステートメント内で指定する必要があるため、すべてのメンバー・ステートメントを正常にコミットするか、またはまったくコミットしないかのどちらかでなければなりません。それゆえ、COMMIT および ROLLBACK ステートメントはどちらもサポートされません。

LOOP および REPEAT ステートメントに関しては、WHILE ステートメントを使用することにより、同等のロジックをインプリメントすることができます。

インライン SQL PL を持つスタンドアロン・スクリプトは、コマンド行プロセッサー (CLP) スクリプト内で動的に準備または実行されるコンパウンド SQL (インライン化) ステートメントの実行、または CLP プロンプトからの直接の実行から成ります。動的に準備または実行されるコンパウンド SQL (インライン化) ステートメントはキーワード BEGIN と END によってバインドされます。このステートメントはデフォルト以外の終了文字で終了しなければなりません。これには SQL PL およびそれ以外の SQL ステートメントを含めることができます。

インライン SQL PL ステートメントは、個別にコンパイルされるのではなく、それを参照する SQL ステートメント内に展開されるものなので、プロシージャー・ロジックを SQL プロシージャー内の SQL PL でインプリメントするのか、関数、トリガー、または動的に準備または実行されるコンパウンド SQL (コンパイル済み) ステートメント内のインライン SQL PL によってインプリメントするのかを計画するときには、パフォーマンスに関する考慮事項についていくらか検討する必要があります。

SQL プロシージャーの SQL PL

SQL PL ステートメントは、主に SQL プロシージャーで使用されます。SQL プロシージャーにはデータの照会と変更用の基本的な SQL ステートメントを含めることができますが、他の SQL ステートメントに制御フロー・ロジックをインプリメントするための SQL PL ステートメントも組み込むことができます。SQL PL ステートメントの完全セットは、SQL プロシージャーで使用できます。

また SQL プロシージャーは、パラメーター、変数、代入ステートメント、条件処理やエラー処理の強力なメカニズム、ネストされた呼び出しや再帰呼び出し、トランザクションおよびセーブポイントのサポート、さらにはプロシージャー呼び出し側またはクライアント・アプリケーションに複数の結果セットを戻す機能をサポートしています。

SQL PL を SQL プロシージャー内で使用すると、SQL でのプログラムを効率的に行えます。SQL プロシージャーが提供する高水準言語の SQL PL およびその他のフィーチャーにより、SQL PL を使用したプログラミングを高速かつ簡単に実行できます。

SQL プロシージャーで使用される SQL PL ステートメントの簡単な例として、以下を検討してください。

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
        INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
   UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
        WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
      SET salary = salary * 1.05, bonus = 500
        WHERE empno = empNum;
  ELSE
    UPDATE employee
      SET salary = salary * 1.03, bonus = 0
        WHERE empno = empNum;
  END IF;
END
```

インライン SQL PL 関数、SQL 関数、トリガー、およびコンパウンド SQL ステートメント

インライン SQL PL ステートメントは、コンパウンド SQL (コンパイル済み) ステートメント、コンパウンド SQL (インライン化) ステートメント、SQL 関数、およびトリガーで実行することができます。

コンパウンド SQL (インライン化) ステートメントとは、複数の SQL ステートメントをオプションのアトミック・ブロックにまとめるためのステートメントであり、このブロックでは、変数や条件処理エレメントを宣言することができます。この種のステートメントは、DB2 によって 1 つの SQL ステートメントとしてコンパイルされます。また、インライン SQL PL ステートメントを組み込むことも可能です。

SQL 関数およびトリガーの本体には、コンパウンド SQL (インライン化) ステートメントを収容することができ、また、一部のインライン SQL PL ステートメントを組み込むこともできます。

コンパウンド SQL (インライン化) ステートメントを単独で使用するのが便利なのは、最小限の制御フローで小単位のロジック作業を実行する一方で、そのデータ・フローはかなりの量になるような、短いスクリプトを作成する場合です。関数およびトリガー内で、そのようなオブジェクトの使用時にさらに複雑なロジックを実行できるようになります。

SQL PL を使用したコンパウンド SQL (インライン化) ステートメントの例として、以下を考察してみます。

```
BEGIN ATOMIC
  FOR row AS
    SELECT pk, c1, discretize(c1) AS v FROM source
  DO
    IF row.v is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
END
```

コンパウンド SQL (インライン化) ステートメントには、キーワード BEGIN および END がバインドされています。その一環として、SQL PL の一部を成す FOR および IF/ELSE 制御ステートメントの両方が使用されています。FOR ステートメントは、一連の定義済みの行に対する反復処理で使用されます。各行ごとに列の値が検査され、条件によっては、値に基づいて一連の値が別の表に挿入されます。

SQL PL を使用したトリガーの例として、以下を考察してみます。

```
CREATE TRIGGER validate_sched
NO CASCADE BEFORE INSERT ON c1_sched
FOR EACH ROW
MODE DB2SQL
Vs: BEGIN ATOMIC

  IF (n.ending IS NULL) THEN
    SET n.ending = n.starting + 1 HOUR;
  END IF;

  IF (n.ending > '21:00') THEN
    SIGNAL SQLSTATE '80000'  SET MESSAGE_TEXT =
           'Class ending time is after 9 PM';
  ELSE IF (n.DAY=1 or n.DAY-7) THEN
    SIGNAL SQLSTATE '80001' SET MESSAGE_TEXT =
           'Class cannot be scheduled on a weekend';
  END IF;
END vs;
```

このトリガーは、c1_sched という名前の表への挿入と同時に活動化してから、SQL PL を使用して、クラスの終了時刻を検査し、未指定の場合はその時刻を設定し、クラスの終了時刻が午後 9 時より遅い場合や、クラスが週末にスケジュールされている場合は、エラーを生じます。 SQL PL を使用したスカラー SQL 関数の例として、以下を考察してみます。

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

この単純な関数は、取引先を特定する入力パラメーターの値に基づいて、スカラー価格値を戻します。また、IF ステートメントも使用します。

出力パラメーター、結果セットの受け渡し、さらに高度な他のプロシージャー型のエレメントを必要とする複雑なロジックの場合は、SQL のプロシージャーのほうが適しているかもしれません。

SQL PL データ・タイプ

アンカー・データ・タイプ

アンカー・データ・タイプは、他のオブジェクトのデータ・タイプと同じになるように定義されているデータ・タイプです。基礎となるオブジェクト・データ・タイプが変更されると、アンカー・データ・タイプも変更されます。

以下のトピックでは、アンカー・データ・タイプの詳細について説明します。

アンカー・データ・タイプのフィーチャー

アンカー・タイプは、列、グローバル変数、SQL 変数、SQL パラメーター、または表やビューの行などの他の SQL オブジェクトに基づいてデータ・タイプを定義します。

アンカー・タイプ定義を使用して定義されたデータ・タイプは、アンカー先のオブジェクトへの従属関係を維持します。アンカー・オブジェクトのデータ・タイプを変更すると、アンカー・データ・タイプに影響を与えます。表またはビューの行にアンカー付けした場合、アンカー・データ・タイプはアンカー表またはアンカー・ビューの列によって定義されるフィールド付きの ROW となります。

このデータ・タイプは、変数に他のオブジェクト (表内の列など) と同じデータ・タイプを持たせる必要があるものの、対象のデータ・タイプを正確に把握していない場合に変数を宣言するときに役立ちます。

アンカー・データ・タイプは、以下のいずれかと同じタイプにできます。

- 表内の行
- ビュー内の行
- カーソル変数行定義
- 表内の列
- ビュー内の列
- ローカル変数。ローカル・カーソル変数または行変数を含む
- グローバル変数

アンカー・データ・タイプを指定できるのは、以下のいずれかを宣言または作成した場合だけです。

- SQL プロシージャー内のローカル変数 (行変数を含む)
- コンパイル済み SQL 関数内のローカル変数 (行変数を含む)
- ルーチン・パラメーター
- CREATE TYPE ステートメントを使用したユーザー定義カーソル・データ・タイプ。
  - DECLARE CURSOR ステートメントで参照することはできません。
- 関数からの戻りデータ・タイプ

- グローバル変数

アンカー・データ・タイプを定義するには、ANCHOR DATA TYPE TO 節 (または短縮形の ANCHOR 節) を指定して、データ・タイプを指定するようにします。アンカー・データ・タイプが行データ・タイプの場合、ANCHOR ROW OF 節または、そのいずれかの同義語を指定する必要があります。こうした節は、以下のステートメントでサポートされています。

- DECLARE
- CREATE TYPE
- CREATE VARIABLE
  - 今回のバージョンでは、グローバル変数をアンカー付けできるのは、他のグローバル変数、表内の列、またはビュー内の列に限られます。

アンカー・データ・タイプに関する制約事項

アンカー・データ・タイプの変数を宣言する前に、またはその使用に関連した問題をトラブルシューティングする際には、このデータ・タイプの使用に関する制約事項を検討してください。

PL/SQL %TYPE 属性を使用して指定したタイプを含め、アンカー・データ・タイプの使用には、以下の制約事項が適用されます。

- インライン SQL 関数では、アンカー・データ・タイプはサポートされていません。
- アンカー・データ・タイプは、ニックネームまたはニックネーム内の列は参照できません。
- アンカー・データ・タイプは、型付き表、型付き表の列、型付きビュー、または型付きビューの列は参照できません。
- アンカー・データ・タイプは、宣言済み一時表、または宣言済み一時表の列は参照できません。
- アンカー・データ・タイプは、緩やかに型付けされたカーソルに関連付けられた行定義は参照できません。
- アンカー・データ・タイプは、データベース・コード・ページとは異なるコード・ページのオブジェクト、またはデータベース照合とは異なる照合のオブジェクトを参照することはできません。

アンカー・データ・タイプ変数

アンカー変数は、データ・タイプがアンカー・データ・タイプであるローカル変数またはパラメーターです。

アンカー変数は、以下のコンテキストでサポートされています。

- SQL プロシージャー
  - SQL プロシージャーで、パラメーターおよびローカル変数がアンカー・データ・タイプとなるように指定できます。
- コンパイル済み SQL 関数

  - CREATE FUNCTION ステートメントを使用して作成された SQL 関数。その際、BEGIN ATOMIC 節ではなく BEGIN 節を指定して、アンカー・データ・タイプのパラメーターまたはローカル変数指定を組み込むことができるようにします。
- モジュール変数
  - アンカー変数は、モジュール内で定義された、パブリッシュ済みまたはパブリッシュされていない変数として指定できます。
- グローバル変数
  - グローバル変数は、アンカー・データ・タイプとして作成できます。

アンカー変数は、DECLARE ステートメントを使用して宣言します。

アンカー・データ・タイプのローカル変数の宣言

アンカー・データ・タイプのローカル変数またはパラメーターの宣言は、変数またはパラメーターのデータ・タイプとアンカー先オブジェクトのデータ・タイプとの整合性を保つ必要がある場合に実行するタスクです。

変数のアンカー先となるデータ・タイプのオブジェクトが定義されている必要があります。

1. 次のようにして DECLARE ステートメントを定式化します。
   a. 変数の名前を指定します。
   b. ANCHOR DATA TYPE TO 節を指定します。
   c. 変数がアンカーされるデータ・タイプのオブジェクトの名前を指定します。
2. サポートされる DB2 インターフェースから DECLARE ステートメントを実行します。

DECLARE ステートメントが正常に実行されると、指定されたアンカー・データ・タイプを使ってデータベース内に変数が定義されます。

以下の例のアンカー・データ・タイプ宣言では、v1 という名前の変数が、emp という表の c1 という列のデータ・タイプにアンカーされます。

```
DECLARE v1 ANCHOR DATA TYPE TO emp.c1;
```

変数が定義されたら、それに値を割り当てたり、参照したり、ルーチンへのパラメーターとして渡すことができます。

例: アンカー・データ・タイプの使用

アンカー・データ・タイプの使用例は、このデータ・タイプを使用する際のリファレンスとして役立ちます。

以下のトピックには、アンカー・データ・タイプの使用例が含まれています。

例: アンカー・データ・タイプの変数宣言:

アンカー・データ・タイプの宣言例を参照すると、変数を宣言する際に役立ちます。

以下は、v1 という名前の変数の宣言例で、この変数のデータ・タイプは表 staff 内の列 name と同じです。

```
DECLARE v1 ANCHOR DATA TYPE TO staff.name;
```

以下は、employee という名前の表に定義されている行にアンカー付けされている empRow1 というタイプを定義する CREATE TYPE ステートメントの例です。

```
CREATE TYPE empRow1 AS ROW ANCHOR DATA TYPE TO ROW OF employee;
```

タイプ empRow1 の宣言済み変数は、フィールド名が表の列名と同じです。

列 name のデータ・タイプが VARCHAR(128) の場合、変数 v1 のデータ・タイプも VARCHAR(128) になります。

例: SQL ルーチンにおけるアンカー・データ・タイプの使用:

SQL ルーチンにおけるアンカー・データ・タイプの使用例は、独自の SQL ルーチンを作成する際に参照すると役立ちます。

以下の一連の例では、SQL ルーチンにおけるアンカー・データ・タイプの様々なフィーチャーと使用法を示しています。アンカー・データ・タイプのフィーチャーが、それが含まれる SQL ルーチンのフィーチャーよりも多く示されています。

以下の例は、表内の列のデータ・タイプにアンカー付けされている宣言済み変数を示しています。

```
CREATE TABLE tab1(col1 INT, col2 CHAR)@

INSERT INTO tab1 VALUES (1,2)@

INSERT INTO tab1 VALUES (3,4)@

CREATE TABLE tab2 (col1a INT, col2a CHAR)@

CREATE PROCEDURE p1()
BEGIN
  DECLARE var1 ANCHOR tab1.col1;
  SELECT col1 INTO var1 FROM tab1 WHERE col2 = 2;
  INSERT INTO tab2 VALUES (var1, 'a');
END@

CALL p1()@
```

プロシージャー p1 が呼び出されると、特定の行の列 col1 の値が選択されて、同じタイプの変数 var1 に入れられます。

以下の CLP スクリプトには、関数に対するパラメーターとしてアンカー・データ・タイプを使用する方法を示す関数例が含まれています。

```
-- Create a table with multiple columns
CREATE TABLE tt1 (c1 VARCHAR(18), c2 CHAR(8), c3 INT, c4 FLOAT)
@

INSERT INTO tt1 VALUES ('aaabbb', 'ab', 1, 1.1)
@

INSERT INTO tt1 VALUES ('cccddd', 'cd', 2, 2.2)
@

SELECT c1, c2, c3, c4 FROM tt1
@

-- Creation of the function
CREATE FUNCTION func_a(p1 ANCHOR tt1.c3)
```

```
 RETURNS INT
 BEGIN
   RETURN p1 + 1;
 END
@

-- Invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(2)
@

-- Another invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(1)
@

DROP FUNCTION func_a
@

DROP TABLE tt1
@
```

関数 func_a が呼び出されると、関数はアンカー・データ・タイプ・パラメーターの値を使用して基本操作を実行します。

行タイプ

行データ・タイプは、ユーザー定義タイプで、それぞれに関連付けられたデータ・タイプを持つ名前付きフィールドの順序付きシーケンスを含みます。

行タイプは、SQL PL でのグローバル変数、SQL 変数、および SQL パラメーターのためのタイプとして使用できます。これを使用すると、データの行 (特に照会を使用して取得されたもの) に含まれる列を柔軟に操作できます。

行データ・タイプのフィーチャー

行データ・タイプのフィーチャーは、SQL PL コードを簡略化するのに役立ちます。

行データ・タイプは、SQL プロシージャー型言語で使用する場合にのみサポートされています。これは、それぞれが独自の名前とデータ・タイプを持つ複数のフィールドからなる構造体で、行の列値を、結果セットまたは他の同様のフォーマット済みデータで保管するのに使用できます。

このデータ・タイプを使用して以下が可能です。

- SQL プロシージャー型言語アプリケーション内でのロジックのコーディングを簡略化します。例えば、データベース・アプリケーションはレコードを一度に 1 つずつ処理し、一時的にレコードを格納するためにパラメーターと変数を必要とします。1 つの行データ・タイプによって、そうしたレコード値を処理および格納しないでも、複数のパラメーターと変数を置換できます。これにより、アプリケーションおよびルーチンにおいて行の値をパラメーターとして引き渡す機能がとても簡略化されます。
- 同様のデータ・タイプをサポートする他のプロシージャー型 SQL 言語で作成されたコードを、DB2 SQL PL に移植するのが容易になります。
- INSERT ステートメント、FETCH ステートメント、VALUES INTO ステートメント、および SELECT INTO ステートメントなどの、データ変更ステートメントおよび照会で行データを参照できます。

行データ・タイプは、CREATE TYPE (ROW) ステートメントを使用して作成する必要があります。定義済みのデータ・タイプの変数は、一度作成すると、DECLARE ステートメントを使用して SQL PL コンテキストで宣言できます。その後こうした変数は、行タイプの値を格納するために使用できます。

行フィールド値を明示的に割り当て、単一のドット「.」表記を使用して参照できます。

行データ・タイプに関する制約事項

行データ・タイプを使用する前に、またはその使用に関連している可能性のあるエラーをトラブルシューティングする際に、行データ・タイプの使用に関する制約事項に注意を払うのは大切です。

行データ・タイプには以下の制約事項が適用されます。

- 行データ・タイプでサポートされる最大フィールド数は、1012 です。
- 行データ・タイプを、CLP からプロシージャーおよび関数に対して入力パラメーター値として渡すことはできません。
- 行データ・タイプを、CLP に対してプロシージャーおよび関数から入出力または出力パラメーター値として渡すことはできません。
- 行データ・タイプ変数同士を直接比較することはできません。行タイプ変数を比較するには、各フィールドを比較できます。
- 以下のデータ・タイプは、行フィールドでサポートされていません。
  - XML データ・タイプ
  - LONG VARCHAR
  - LONG VARGRAPHIC
  - 構造化データ・タイプ
  - 行データ・タイプ
  - 配列データ・タイプ
- LOB タイプの 1 つ以上のフィールドが含まれる行タイプのグローバル変数は、サポートされていません。
- CAST 関数を使用してパラメーター値を行データ・タイプにキャストすることはできません。

データ・タイプの使用、権限、SQL の実行、データ・タイプの使用有効範囲、またはその他の問題に関連して、他の一般的な制約事項が適用される場合があります。

行変数

行変数は、ユーザー定義の行データ・タイプに基づく変数です。行変数は、宣言、値の割り当て、別の値への設定、または SQL プロシージャーとの間におけるパラメーターとしての受け渡しが可能です。行変数は、ベースとなる行データ・タイプのプロパティーを継承します。行変数は、結果セット内のデータの行を保持するために使用したり、他のタプル形式のデータを割り当てたりできます。

行変数は、DECLARE ステートメントを使用して SQL プロシージャー内で宣言できます。

行変数の作成

行変数を作成するには、最初に行タイプを作成してから、その行変数を宣言する必要があります。

以下のトピックでは、行データ・タイプと変数の作成方法が示されています。

行データ・タイプの作成:

行データ・タイプの作成は、行変数の作成の前提条件となります。

行データ・タイプを作成する前に、以下を行ってください。

- 9 ページの『行タイプ』 を読んでください
- 10 ページの『行データ・タイプに関する制約事項』 を読んでください

このタスクは、CREATE TYPE ステートメントの実行がサポートされている任意のインターフェースから実行できます。

データベース内に行データ・タイプを作成するには、SQL ステートメントの実行がサポートされている任意の DB2 インターフェースから、CREATE TYPE (ROW) ステートメントを正常に実行しなければなりません。

1. 以下のようにして、CREATE TYPE (ROW) ステートメントを定式化します。
   a. タイプの名前を指定します。
   b. 行の各フィールドに、名前とデータ・タイプを指定して、行の行フィールド定義を指定します。

   以下の例は、empRow 行データ・タイプと同じフォーマットを持つ結果セットに関連付けることができる行データ・タイプを作成する方法について示しています。

   ```
   CREATE TYPE empRow AS ROW (name VARCHAR(128), id VARCHAR(8));
   ```
2. サポートされる DB2 インターフェースから CREATE TYPE ステートメントを実行します。

CREATE TYPE ステートメントが正常に実行されると、その行データ・タイプがデータベース内に作成されます。ステートメントが正常に実行されない場合、ステートメントの構文を検証し、そのデータ・タイプが存在していないことも確かめます。

行データ・タイプが作成されると、このデータ・タイプに基づいて行変数を宣言できます。

行タイプのローカル変数の宣言:

行タイプの変数は、行データ・タイプの作成後に宣言できます。

行データ・タイプを作成する前に、以下を行ってください。

- 9 ページの『行タイプ』 を読んでください
- 10 ページの『行データ・タイプに関する制約事項』 を読んでください

行データ・タイプ変数を宣言できるのは、DECLARE ステートメントの実行がサポートされている SQL プロシージャーおよび関数が含まれている SQL PL コンテキストに限られます。

ローカル行変数を宣言するには、以下のステップを実行しなければなりません。

1. 次のようにして DECLARE ステートメントを定式化します。
   a. 変数の名前を指定します。
   b. 変数を定義する行データ・タイプを指定します。指定する行データ・タイプは、データベース内に定義済みでなければなりません。

   以下の例は、タイプ empRow の行変数を定義する DECLARE ステートメントの定式化方法を示しています。

   ```
   DECLARE r1 empRow;
   ```
2. サポートされるコンテキストで DECLARE ステートメントを実行します。

DECLARE ステートメントが正常に実行されると、行変数が作成されます。

行変数が作成される際、行の各フィールドは NULL 値に初期化されます。

行変数は、値を割り当てたり、参照したり、パラメーターとして受け渡したりできます。

行変数に対する値の割り当て

値は、いくつかの方法で行タイプの変数に割り当てることができます。行変数値は、別の行変数に割り当てることができます。変数フィールド値を割り当てて参照することが可能です。行のフィールド値を参照するには、単一ドット「.」表記を使用します。

以下のトピックには、行タイプ変数および行タイプ変数の配列に対して値を割り当てる方法が示されています。

行データ・タイプに対するサポートされている割り当て:

行または行フィールドに対する割り当てに関して、様々な値がサポートされています。

行変数またはパラメーターを宣言すると、行にある各フィールドには、値を割り当てるまでデフォルト値の NULL が入ります。

行変数には、以下のタイプの値を割り当てることができます。

- SET ステートメントを使用して、同じ行データ・タイプの別の行変数
  - 互換性のあるタイプの場合、行変数に割り当て可能なのは行変数値だけです。2 つの行変数がどちらも同じ行データ・タイプである場合、またはソース行変数が表またはビュー定義にアンカー付けされている場合には、それら 2 つの行変数には互換性があります。2 つの変数のタイプが互換性を持つには、フィールド名とフィールド・データ・タイプが同じであるだけでは不十分です。

    例えば、row1 という名前の行データ・タイプと、row2 という名前の別のデータ・タイプを作成し、どちらの定義も同じ場合、タイプ row1 の変数の値をタイプ row2 の変数に割り当てることはできません。または、タイプ row2 の変

数の値をタイプ row1 の変数に割り当てることもできません。ただし、タイプ row1 の変数 v1 の値を、やはりタイプ row1 の変数 v2 に割り当てることはできます。

- 行とエレメント数が同じで、行のフィールドと同じデータ・タイプのエレメントからなるタプル。
  - 以下は、行に割り当てるリテラル・タプルの例です。

    ```
    SET v1 = (1, 'abc')
    ```
- 行の値に解決する式
  - 行変数に割り当て可能な行の値に解決する式の例としては、VALUES ... INTO ステートメントにおける解決済み式があります。以下は、そのような割り当ての例です。

    ```
    VALUES (1, 'abc') INTO rv1
    ```
- 関数の戻りタイプ (ターゲット変数と行データ・タイプが同じ場合)。
  - 以下の例では、foo という名前の関数の戻りタイプがターゲット変数と同じ行データ・タイプです。

    ```
    SET v1 = foo()
    ```

    戻りデータ・タイプがアンカー・データ・タイプとして定義されている場合、アンカー・データ・タイプの割り当て規則が適用されます。
- 照会の単一行結果セット
  - 結果セットには行と同じエレメント数がなければならず、列は行のフィールドと同じデータ・タイプに割り当て可能である必要があります。以下は、このタイプの割り当て例です。

    ```
    SET v1 = (select c1, c2 from T)
    ```
- NULL
  - 行変数に対して NULL が割り当てられると、すべての行フィールドは NULL に設定されますが、行変数自体は NOT NULL のままです。

以下のタイプの値を行変数フィールドに割り当てることができます。

- リテラル
- パラメーター
- 変数
- 式
- NULL

以下の方法で、値を行フィールド値に割り当てられます。

- SET ステートメントの使用
- 行の値に解決する SELECT INTO ステートメントの使用
- 行の値に解決する FETCH INTO ステートメントの使用
- 行の値に解決する VALUES INTO ステートメントの使用

ROW データ・タイプは、SQL スカラー関数の戻りタイプとして指定できます。

SET ステートメントによる行変数に対する値の割り当て:

行変数に対して値を割り当てるには、SET ステートメントを使用できます。行変数には行の値を割り当てることができます。行フィールド値または式を行フィールドに割り当てることができます。

行の値と行変数がともに同じユーザー定義の行データ・タイプである場合には、SET ステートメントを使用して行の値を行変数に割り当てることができます。

以下の例は、行の値を、同じフォーマットの行変数に割り当てる方法について示しています。

```
SET empRow = newHire;
```

行の値 newHire は empRow 変数と同じフォーマットです。以下に示すように行フィールドの数とタイプが同じです。

```
empRow.lastName      /* VARCHAR(128) */
empRow.firstName     /* VARCHAR(128) */
empRow.id            /* VARCHAR(10)  */
empRow.hireDate      /* TIMESTAMP    */
empRow.dept          /* VARCHAR(3)   */

newHire.lastName     /* VARCHAR(128) */
newHire.firstName    /* VARCHAR(128) */
newHire.id           /* VARCHAR(10)  */
newHire.hireDate     /* TIMESTAMP    */
newHire.dept         /* VARCHAR(3)   */
```

行の値を、フォーマットが同じではない変数に割り当てようとすると、エラーが生じます。

行の値を割り当てる場合、行内の個々のフィールドに値を割り当てることができます。以下の例は、SET ステートメントを使用して empRow という名前の行のフィールドに値を割り当てる方法を示しています。

```
SET empRow.lastName = 'Brown';           // Literal value assignment

SET empRow.firstName = parmFirstName;    // Parameter value of same type assignment

SET empRow.id = var1;                    // Local variable of same type assignment

SET empRow.hiredate = CURRENT_TIMESTAMP; // Special register expression assignment

SET empRow.dept = NULL;                  // NULL value assignment
```

行の値を初期化するには、任意のサポート対象のフィールド割り当てを使用できます。

SELECT、VALUES、または FETCH ステートメントによる行変数に対する行の値の割り当て:

SELECT INTO ステートメント、VALUES INTO ステートメント、または FETCH INTO ステートメントを使用して、行タイプの変数に対して、行の値を割り当てることができます。ソース行の値のフィールド値は、ターゲット行変数のフィールド値に割り当て可能でなければなりません。

以下の例は、SELECT ステートメントを使用して、empRow という名前の行変数に 1 つの行の値を割り当てる方法を示しています。

```
SELECT * FROM employee
INTO empRow
WHERE id=5;
```

この SELECT 照会が複数の行の値を解決すると、エラーが生じます。

以下の例は、VALUES INTO ステートメントを使用して、empEmpBasics という名前の行変数に 1 つの行の値を割り当てる方法を示しています。

```
VALUES (5, 'Jane Doe', 10000) INTO empBasics;
```

以下の例は、FETCH ステートメントを使用して empRow という名前の行変数に行の値を割り当てる方法を示しています。このステートメントは、cur1 という名前のカーソルを参照し、このカーソルは変数 empRow と互換性のあるフィールド値を持つ行を定義します。

```
FETCH cur1 INTO empRow;
```

これらのそれぞれのステートメントを、他のさまざまな方法で使用することが可能です。

行変数および行フィールド値の比較

行変数は同じ行データ・タイプであっても直接比較することはできませんが、個々の行フィールドは比較できます。

行タイプ内の個々のフィールドは他の値と比較可能で、フィールドのデータ・タイプの比較規則が適用されます。

2 つの行変数を比較するには、それぞれの対応するフィールド値を比較する必要があります。

以下は、SQL PL 内の互換性のあるフィールド定義を持つ 2 つの行の値に関する比較の例です。

```
IF ROW1.field1 = ROW2.field1 AND
   ROW1.field2 = ROW2.field2 AND
   ROW1.field3 = ROW2.field3
THEN
   SET EQUAL = 1;
ELSE
   SET EQUAL = 0;
```

この例では、IF ステートメントを使用してプロシージャー型ロジックを実行します。このロジックは、フィールド値が等しい場合にはローカル変数 EQUAL を 1 に設定し、フィールド値が等しくない場合には 0 に設定します。

行の値の参照

SQL および SQL ステートメント内では、行の値および行フィールド値を参照できます。

以下のトピックは、行の値を参照できる箇所およびその方法について示しています。

行変数の参照:

行変数データ・タイプの参照がサポートされている場合には、どのような箇所でも行変数値を名前で参照できます。

サポートされている行変数参照コンテキストとしては、以下が挙げられます。

- SET ステートメントのソースまたはターゲット
- INSERT ステートメント
- SELECT INTO、VALUES INTO、または FETCH ステートメントのターゲット

以下は、SET ステートメントを使用して、ある行変数を、同じ定義を持つ別の行変数に割り当てる例です。

```
-- Referencing row variables as source and
   target of a SET statement
SET v1 = v2;
```

以下は、INSERT ステートメントで参照されている行変数の例で、このステートメントは 2 つの行を挿入します。行変数 v1 と v2 のフィールド定義のタイプは、INSERT ステートメントのターゲットである表の列定義と互換性があります。

```
-- Referencing row variables in an INSERT statement
INSERT INTO employee VALUES v1, v2;
```

以下は、FETCH ステートメントで参照されている行変数の例です。行変数 empRow には、カーソル c1 と関連付けられている結果セットと同じ列定義があります。

```
-- Referencing row variables in a FETCH statement
FETCH c1 INTO empRow;
```

以下は、SELECT ステートメントで参照されている、v3 という名前の行変数の例です。employee 表にある 2 つの列値が選択されて、変数 v3 の 2 つのフィールドに入れられます。

```
-- Referencing row variables in a SELECT statement
SELECT id, name INTO v3 FROM employee;
```

行変数内のフィールドの参照:

フィールド値は、複数のコンテキストで参照できます。

行フィールド値は、フィールドのデータ・タイプの値が許可される場合にはどの場所でも参照できます。行フィールドの参照は、以下のコンテキストでサポートされています。

- 以下のように、フィールドのデータ・タイプの値が許可されるすべての場所 (以下の場合に限定される訳ではありません)。
  - 割り当てのソースとして (SET ステートメント)
  - 割り当てのターゲットとして (SET ステートメント)
  - SELECT INTO、VALUES INTO、または FETCH INTO ステートメントのターゲットとして。

行変数内のフィールドの値を参照するには、単一ドット表記を使用します。フィールド値は、以下のように変数と関連付けられます。

```
<row-variable-name>.<field-name>
```

以下は、変数 employee のフィールド id にアクセスする方法の例です。

```
employee.id
```

以下に、サポートされている、行変数フィールド値に対する参照例を幾つか示します。

以下の例は、行変数 v1 内のフィールドにリテラル値を割り当てる方法を示しています。

```
-- Literal assignment to a row variable field
SET v1.c1 = 5;
```

以下の例は、複数の行変数フィールドにリテラル値および式値を割り当てる方法を示しています。

```
-- Literal assignment to fields of row variable
SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
```

以下の例は、INSERT ステートメントでフィールド値を参照する方法を示しています。

```
-- Field references in an INSERT statement
INSERT INTO employee
VALUES(v1.c1, 'Beth'),
      (emp.id, emp.name);
```

以下の例は、UPDATE ステートメントでフィールド値を参照する方法を示しています。

```
-- Field references in an UPDATE statement
UPDATE employee
SET name = 'Susan'
WHERE id = v1.c1;
```

以下の例は、SELECT INTO ステートメントでフィールド値を参照する方法を示しています。

```
-- Field references in a SELECT INTO statement
SELECT employee.firstname INTO v2.c1
FROM employee
WHERE name=emp.name;
```

INSERT での行変数の参照 ステートメント:

表全体の行を追加または変更するために、INSERT ステートメントで行変数を使用できます。

以下は、1 つの行を employee 表に挿入する INSERT ステートメントの例です。

```
INSERT INTO employee VALUES empRow;
```

INSERT ステートメントの場合、行変数のフィールド数は、暗黙的または明示的ターゲット列リスト内の列数と一致しなければなりません。

前述の INSERT ステートメントは、対応する行フィールドの値を表の各列に挿入します。従って、この INSERT ステートメントは、以下の INSERT ステートメントに相当します。

```
INSERT INTO employee VALUES (emp.id,
                             emp.name,
                             emp.salary,
                             emp.phone);
```

ルーチン・パラメーターとしての行の引き渡し

行タイプ値および行タイプ変数の配列を、パラメーターとしてプロシージャーおよび関数に引き渡すことができます。プロシージャーは、これらのデータ・タイプを IN、OUT、および INOUT パラメーターとしてサポートしています。

以下に、CHAR タイプを入力パラメーターとして受け入れ、出力行パラメーターのフィールドを変更してから戻すプロシージャーの例を示します。

```
CREATE PROCEDURE p(IN basicChar CHAR, OUT outEmpRow empRow)
BEGIN

  SET outEmpRow.field2 = basicChar;

END@
```

以下は、このプロシージャーを呼び出す CALL ステートメントの例です。

```
CALL p('1', myEmpRow)@
```

行データ・タイプのドロップ

行データ・タイプが不要になった場合、または既存の行データ・タイプの名前を再使用したい場合には、行データ・タイプをドロップします。

行データ・タイプをドロップする前に、以下の前提条件が満たされる必要があります。

- データベースへの接続を確立する必要があります。
- 行データ・タイプがデータベースに存在している必要があります。

行データ・タイプが不要になった場合、または既存の行データ・タイプの名前を再使用したい場合には、行データ・タイプをドロップします。 DROP ステートメントの実行をサポートする任意のインターフェースから、行をドロップすることができます。

1. ドロップ対象の行データ・タイプの名前を指定する DROP ステートメントを定式化します。
2. サポートされる DB2 インターフェースから DROP ステートメントを実行します。

以下の例は、simpleRow という名前の行データ・タイプをドロップする方法を示しています。

```
DROP TYPE simpleRow;
```

DROP ステートメントが正常に実行されると、行データ・タイプがデータベースからドロップされます。

例: 行データ・タイプの使用

行データ・タイプの使用例を参照すると、行データ・タイプの使用法と、どのような場合に使用できるかについての理解を深めるのに役立ちます。

以下のトピックには、行データ・タイプの使用法が示されています。

例: CLP スクリプトにおける行データ・タイプの使用:

行データ・タイプの最も一般的な使用法について示すため、DB2 CLP スクリプトにおける行データ・タイプの一部の基本的なフィーチャーについて説明します。

以下の DB2 CLP スクリプトは、行データ・タイプの使用法と、その関連操作について示しています。以下についてのデモンストレーションが含まれます。

- 行データ・タイプの作成
- 表の作成
- プロシージャーの作成。以下が含まれます。
  - 行データ・タイプ宣言
  - 特定の行フィールド値が含まれるタイプに対する値の挿入
  - 行フィールド値に基づく行の値の更新
  - 行フィールド値に対する値の SELECT INTO
  - 行に対する行の値の割り当て
  - パラメーターに対する行フィールド値の割り当て
- プロシージャーの呼び出し
- 行データ・タイプと表のドロップ

```
-- Creating row types

CREATE TYPE row01 AS ROW (c1 INTEGER)@

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))@

CREATE TABLE employee (id INTEGER, name VARCHAR(10))@

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
BEGIN
  DECLARE v1, v2 row01;
  DECLARE emp empRow;

  -- Assigning values to row fields
  SET v1.c1 = 5;
  SET (emp.id, emp.name) = (v1.c1 + 1, 'James');

  -- Using row fields in DML
  INSERT INTO employee
  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);

  UPDATE employee
  SET name = 'Susan' where id = v1.c1;

  -- SELECT INTO a row field
  SELECT id INTO v2.c1
  FROM employee
  WHERE name = emp.name;

  -- Row level assignment
  SET v1 = v2;

  -- Assignment to parameters
  SET (p0, p1) = (v1.c1, emp.id);

END@

CALL proc01(?, ?)@

SELECT * FROM employee@

DROP procedure proc01@
```

```
DROP TABLE employee@

-- Dropping row types
DROP TYPE empRow@

DROP TYPE row01@
```

このスクリプトを保管してから、以下のとおりに発行して、DB2 コマンド行から実行できます。

```
DB2 -td@ -vf <filename>;
```

このスクリプトを実行した際の出力は、以下のとおりです。

```
CREATE TYPE row01 AS ROW (c1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE TABLE employee (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
   BEGIN DECLARE v1, v2 row01;
   DECLARE emp empRow;
   SET v1.c1 = 5;
   SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
   INSERT INTO employee  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);
   UPDATE employee  SET name = 'Susan' where id = v1.c1;
   SELECT id INTO v2.c1 FROM employee WHERE name = emp.name;
   SET v1 = v2;
   SET (p0, p1) = (v1.c1, emp.id);
END

DB20000I  The SQL command completed successfully.

CALL proc01(?, ?)

  Value of output parameters
  --------------------------
  Parameter Name  : P0
  Parameter Value : 6

  Parameter Name  : P1
  Parameter Value : 6

  Return Status = 0

SELECT * FROM employee

ID          NAME
----------- ----------
          5 Susan
          6 James

  2 record(s) selected.


DROP procedure proc01
DB20000I  The SQL command completed successfully.

DROP TABLE employee
DB20000I  The SQL command completed successfully.

DROP TYPE empRow
```

```
DB20000I  The SQL command completed successfully.

DROP TYPE row01
DB20000I  The SQL command completed successfully.
```

例: SQL プロシージャー内における行データ・タイプの使用:

SQL プロシージャーで行データ・タイプを使用すると、レコード・データを取り出して、それをパラメーターとして渡すことができます。

このトピックでは、複数の SQL プロシージャーの定義を含んだ CLP スクリプトの例を示します。これは、多様な行の使用方法の一部です。

ADD_EMP という名前のプロシージャーは、1 つの行データ・タイプを入力パラメーターとして使用して、それを表に挿入します。

NEW_HIRE という名前のプロシージャーは、SET ステートメントを使用して、行変数に値を割り当て、行データ・タイプ値を CALL ステートメントにパラメーターとして渡します。このステートメントは、別のプロシージャーを呼び出します。

FIRE_EMP というプロシージャーは、表データの行を選択して行変数に入れ、行フィールド値を表に挿入します。

以下がこの CLP スクリプトで、その後に、冗長モードで CLP からこのスクリプトを実行した出力が続きます。

```
--#SET TERMINATOR @;
CREATE TABLE employee (id INT,
                       name VARCHAR(10),
                       salary DECIMAL(9,2))@

INSERT INTO employee VALUES (1, 'Mike', 35000),
                            (2, 'Susan', 35000)@

CREATE TABLE former_employee (id INT, name VARCHAR(10))@

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee@
CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
END@

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary)
    = (maxID + 1, newName, 30000);

  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END@

CREATE PROCEDURE FIRE_EMP (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
```

```
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END@

CALL NEW_HIRE('Adam')@

CALL FIRE_EMP(1)@

SELECT * FROM employee@

SELECT * FROM former_employee@
```

以下は、冗長モードで CLP からこのスクリプトを実行した出力です。

```
CREATE TABLE employee (id INT, name VARCHAR(10), salary DECIMAL(9,2))
DB20000I  The SQL command completed successfully.

INSERT INTO employee VALUES (1, 'Mike', 35000), (2, 'Susan', 35000)
DB20000I  The SQL command completed successfully.

CREATE TABLE former_employee (id INT, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary) = (maxID + 1, newName, 30000);

  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE FIRE_EMPLOYEE (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END
DB20000I  The SQL command completed successfully.

CALL NEW_HIRE('Adam')

  Return Status = 0
```

```
CALL FIRE_EMPLOYEE(1)

  Return Status = 0

SELECT * FROM employee

ID          NAME       SALARY
----------- ---------- -----------
          2 Susan         35000.00
          3 Adam          30000.00

  2 record(s) selected.


SELECT * FROM former_employee

ID          NAME
----------- ----------
          1 Mike

  1 record(s) selected.
```

例: SQL 関数での行データ・タイプの使用:

SQL 関数で行データ・タイプを使用して、レコード・データを構成、保管、または変更できます。

行データ・タイプに基づく変数を使用すると、表と同じフォーマットを持つ行の値を簡単に保持することができます。こうした仕方で使用すると、行変数を最初に使用する際に初期化するのに役立ちます。

以下は、DB2 CLP スクリプトの例です。このスクリプトには、表、行データ・タイプ、および関数を作成する SQL ステートメントが含まれています。作成される関数には、行変数、行参照、および UDF の呼び出しの宣言が組み込まれています。

```
CREATE TABLE t1 (deptNo VARCHAR(3),
                 reportNo VARCHAR(3),
                 deptName VARCHAR(29),
                 mgrNo VARCHAR (8),
                 location VARCHAR(128))@

 INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')@
 INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')@
 INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')@

 CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3),
                             r_reportNo VARCHAR(3),
                             r_depTName VARCHAR(29),
                             r_mgrNo VARCHAR (8),
                             r_location VARCHAR(128))@

 CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                             reportNo VARCHAR(3),
                             theName VARCHAR(29))
  RETURNS VARCHAR(128)
  BEGIN

    -- Declare a row variable
    DECLARE dept deptRow;

    -- Assign values to the fields of the row variable
    SET dept.r_deptno = theDeptNo;
    SET dept.r_reportNo = reportNo;
    SET dept.r_deptname = theName;
```

```
    SET dept.r_mgrno = '';
    SET dept.r_location = '';

    RETURN
      (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);

  END@

VALUES (getLocation ('789', 'MM3','Marketing-1'))@
```

この CLP スクリプトを実行すると表が作成され、その表に行が挿入され、行データ・タイプと UDF が作成されます。

関数 getLocation は SQL UDF で、行変数を宣言し、入力パラメーター値を使用して値を行変数フィールドに割り当てます。この関数は、この関数が戻すスカラー値を定義する SELECT ステートメント内の行変数のフィールドの 1 つを参照します。

このスクリプトの最後で VALUES ステートメントが実行されると、UDF が呼び出され、スカラー戻り値が戻されます。

以下は、CLP からこのスクリプトを実行した際の出力です。

```
CREATE TABLE t1 (deptNo VARCHAR(3), reportNo VARCHAR(3),
 deptName VARCHAR(29), mgrNo VARCHAR (8), location VARCHAR(128))
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')
DB20000I  The SQL command completed successfully.

CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3), r_reportNo VARCHAR(3), r_depTNa
me VARCHAR(29), r_mgrNo VARCHAR (8), r_location VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                            reportNo VARCHAR(3),
                            theName VARCHAR(29))
 RETURNS VARCHAR(128)
 BEGIN
   DECLARE dept deptRow;
   SET dept.r_deptno = theDeptNo;
   SET dept.r_reportNo = reportNo;
   SET dept.r_deptname = theName;
   SET dept.r_mgrno = '';
   SET dept.r_location = '';
 RETURN
   (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);
 END
DB20000I  The SQL command completed successfully.

VALUES (getLocation ('789', 'MM3','Marketing-1'))

1

------------------------------------------------------------------------------
----------------------------------------------
```

```
Toronto

  1 record(s) selected.
```

配列タイプ

配列タイプはユーザー定義のデータ・タイプで、単一のデータ・タイプのエレメントからなる順序付きセットで構成されます。

通常 配列タイプは、エレメントの数に上限が定義されており、添字として順序位置を使用します。

連想 配列タイプにはエレメントの数に関する特定の上限はなく、各エレメントには関連付けられた索引値があります。索引値のデータ・タイプは整数または文字ストリングにすることができますが、配列全体で同じデータ・タイプになります。

配列タイプは、単一のデータ・タイプの値のコレクションを柔軟に処理するために、SQL PL 内のグローバル変数、SQL 変数、および SQL パラメーター用のタイプとして使用できます。

配列と連想配列の比較

単純配列と連想配列は、いくつかの面で異なります。これらの相違点を理解すると、使用するデータ・タイプを適切に選択する際に役立ちます。

以下の表では、配列と連想配列の相違点について取り上げています。

表 1. 配列と連想配列の比較

配列	**連想配列**
単純配列の最大カーディナリティーは、単純配列の定義時に定義されます。索引 N に値を割り当てると、配列の現行カーディナリティーと N の間の索引を持つエレメントは、暗黙的に NULL に初期化されます。	ユーザー指定の最大カーディナリティーは存在せず、連想配列変数の宣言時に初期化されるエレメントはありません。最大カーディナリティーは、使用可能な空きメモリーによって制限されます。
単純配列の索引データ・タイプは整数値でなければなりません。	連想配列の索引タイプは、サポートされているデータ・タイプのいずれかまたはその集合とすることができます。
単純配列内の索引値は、整数値の連続する集合でなければなりません。	連想配列の場合、索引値は散在していて構いません。
単純配列の CREATE TYPE ステートメントでは、配列カーディナリティーの指定は必要ありません。例えば、以下のステートメントではカーディナリティーは指定しません。 `CREATE TYPE simple AS INTEGER ARRAY[];`	連想配列の CREATE TYPE ステートメントの場合、配列カーディナリティーの指定は必要としませんが、代わりに索引データ・タイプが必要となります。例えば、以下のステートメントでは、索引データ・タイプのカーディナリティーが INTEGER として指定されています。 `CREATE TYPE assoc AS INTEGER ARRAY[INTEGER];`

表 1. 配列と連想配列の比較 (続き)

配列	連想配列

単純配列に最初に割り当てを行うと、1 と配列に割り当てられた索引値との間の索引値を持つ配列エレメントが初期化されます。以下のコンパウンド SQL (コンパイル済み) ステートメントには、単純配列変数の宣言と、その変数への値の割り当てが含まれています。
`BEGIN`
` DECLARE mySimpleA simple;`
 	
`  SET mySimpleA[100] = 123;`
 	
`END`
 	
代入文を実行すると、mySimpleA のカーディナリティーは 100 です。値 1 から 99 までの索引を持つエレメントは暗黙的に NULL に初期化されます。
	連想配列に対する最初の割り当てにより、1 つの索引値を持つ 1 つのエレメントが初期化されます。 以下のコンパウンド SQL (コンパイル済み) ステートメントには、連想配列変数の宣言と、その変数への値の割り当てが含まれています。
	`BEGIN`
	`  DECLARE myAssocA assoc;`
	
	`  SET myAssocA[100] = 123;`
	`END`
	
	代入文の実行後、この配列のカーディナリティーは 1 です。

例

通常配列データ・タイプ

通常配列データ・タイプとは、データ・エレメントの順序付きコレクションがあり、各エレメントはコレクションにおけるその順序位置で参照できるようになっている構造のことをいいます。

N が配列におけるカーディナリティー (エレメント数) である場合、(索引と呼ばれる) 各エレメントに関連付けられた順序位置は、1 以上 N 以下の整数値です。配列内のエレメントは、すべて同じデータ・タイプになります。

配列データ・タイプのフィーチャー:

配列データ・タイプの多くのフィーチャーにより、SQL PL ロジックを理想に近い形で使用できます。

配列タイプとは、別のデータ・タイプの配列として定義されているデータ・タイプのことです。

すべての配列タイプには最大カーディナリティーがあり、これは CREATE TYPE ステートメントで指定されます。 A が最大カーディナリティー M を持つ配列タイプである場合、タイプ A の値のカーディナリティーは 0 から M まで (両端を含む) の値となります。 C などのプログラミング言語における配列の最大カーディナリティーとは異なり、SQL 配列の最大カーディナリティーはそれらの物理表現とは関連していません。むしろ、最大カーディナリティーは、添え字が境界内にあることを確実にするため、実行時にシステムにより使用されます。配列の値を示すために必要なメモリー量は、通常、その配列タイプの最大カーディナリティーにではなく、その配列のカーディナリティーに比例します。

配列が参照されているとき、配列のすべての値はメイン・メモリーに格納されます。そのため、大量のデータを含む配列は、大量のメイン・メモリーを消費します。

配列エレメント値は、エレメントの対応する索引値を指定すると取り出せます。

配列データ・タイプが役に立つのは、単一のデータ・タイプの値の集合を保管したい場合です。この値の集合を使用して、ルーチンへの値の受け渡しを大幅に簡略化できます。複数の、場合によってかなり多くのパラメーターを個別に受け渡す代わりに、単一の配列値を受け渡すことができるからです。

配列データ・タイプは、連想配列データ・タイプとは異なります。配列データ・タイプは値のシンプルなコレクションであるのに対して、連想配列は概念上は複数の配列からなる配列に似ています。つまり連想配列は、配列エレメントには 1 次索引がアクセスし、副配列エレメントには副索引がアクセスするといったように、ゼロ個以上の副配列エレメントが含まれる順序付けされた配列です。

配列データ・タイプに関する制約事項:

この配列データ・タイプを使用する前に、またはその宣言や使用に関連する問題をトラブルシューティングする場合、その制約事項に注意を払うのは大切です。

この配列データ・タイプには以下の制約事項が適用されます。

- 動的コンパウンド・ステートメントにおいて、配列データ・タイプを使用することはできません。
- SQL プロシージャー外で ARRAY_AGG 関数を使用することはできません。
- SQL プロシージャー外で UNNEST 関数を使用することはできません。
- Java™ プロシージャー以外の外部プロシージャーで配列データ・タイプのパラメーターを使用することはできません。
- ユーザー定義の配列データ・タイプ以外のデータ・タイプに配列をキャストすることはできません。
- 配列に指定されている以外のデータ・タイプのエレメントを含めることはできません。
- ターゲット配列のカーディナリティーよりも大きいカーディナリティーを持つ配列をキャストすることはできません。
- パラメーターまたは戻りタイプとして配列をメソッドで使用することはできません。
- パラメーターまたは戻りタイプとして配列をソース派生関数またはテンプレート関数で使用することはできません。
- パラメーターまたは戻りタイプとして配列を外部スカラー関数または外部表関数で使用することはできません。
- パラメーターまたは戻りタイプとして配列を SQL スカラー関数、SQL 表関数、または SQL 行関数で使用することはできません。
- 配列以外のデータ・タイプに、TRIM_ARRAY 関数の結果値を割り当てたり、キャストしたりすることはできません。
- 配列以外のデータ・タイプに、ARRAY コンストラクターまたは ARRAY 関数の結果値を割り当てたり、キャストしたりすることはできません。
-

配列変数:

配列変数は、ユーザー定義の配列データ・タイプに基づく変数です。配列変数は、宣言、値の割り当て、別の値への設定、または SQL プロシージャーとの間におけるパラメーターとしての受け渡しが可能です。

配列変数は、ベースとなる配列データ・タイプのプロパティーを継承します。配列変数は、同じデータ・タイプのデータの集合を保持するために使用します。

ローカル配列変数は、DECLARE ステートメントを使用して SQL プロシージャー内で宣言できます。

グローバル配列変数は、CREATE VARIABLE ステートメントを使用して作成できます。

配列変数の作成:

配列変数を作成するには、まず配列タイプを作成した後、ローカル配列変数を宣言するか、グローバル配列変数を作成する必要があります。

以下のトピックでは、配列データ・タイプおよび配列変数を作成する方法を示します。

配列データ・タイプの作成 (CREATE TYPE ステートメント):

配列データ・タイプの作成は、通常、配列データ・タイプの変数を作成するための前提条件として実行されるタスクです。

配列データ・タイプを作成する前に、CREATE TYPE ステートメントの実行に必要な特権があることを確認してください。

配列データ・タイプは、CREATE TYPE ステートメントの実行がサポートされる SQL PL コンテキストでのみ作成可能です。

制約事項

27 ページの『配列データ・タイプに関する制約事項』を参照してください。

1. CREATE TYPE ステートメントを定義します
   a. 配列データ・タイプの名前を指定します。
   b. AS キーワードを指定し、その後に配列エレメントのデータ・タイプのキーワード名を指定します。例: INTEGER、VARCHAR。
   c. ARRAY キーワード、および配列内の副索引のドメインを指定します。例えば 100 を指定した場合、有効な索引は 1 から 100 までになります。 この数値は配列のカーディナリティー (配列のエレメントの数) と同じです。
2. サポートされるインターフェースから CREATE TYPE ステートメントを実行します。

CREATE TYPE ステートメントが正常に実行されると、配列タイプが作成されます。

例 1:

```
CREATE TYPE simpleArray AS INTEGER ARRAY[100];
```

この配列データ・タイプには、1 から 100 までの範囲の整数値で索引が付けられる最大 100 個までの整数値を含めることができます。
例 2:

```
CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
```

この配列データ・タイプには、1 から 100 までの範囲の整数値で索引が付けられる、VARCHAR(20) データ・タイプ値として保管される電話番号値を最大 100 個まで含めることができます。

配列データ・タイプを作成した後、配列変数を宣言することができます。

配列タイプのローカル変数の宣言:

配列データ・タイプ値を一時的に保管または受け渡す必要がある場合には、配列データ・タイプを作成した後のタスクとして、配列データ・タイプの変数を宣言します。

行タイプのローカル変数を作成する前に、以下を行います。

- 配列データ・タイプを読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- 28 ページの『配列データ・タイプの作成 (CREATE TYPE ステートメント)』 を読んでください
- DECLARE ステートメントの実行に必要な特権があることを確認します。

配列データ・タイプの宣言を行うことができるコンテキストは、SQL プロシージャー、SQL 関数、およびトリガーの中です。

1. DECLARE ステートメントを定義します。
   a. 配列データ・タイプ変数の名前を指定します。
   b. 配列データ・タイプの作成時に使用した配列データ・タイプの名前を指定します。

   次のような CREATE TYPE ステートメントを使って配列データ・タイプが宣言された場合、

   ```
   CREATE TYPE simpleArray AS INTEGER ARRAY[10];
   ```

   このデータ・タイプの変数は、例えば次のように宣言されます。

   ```
   DECLARE myArray simpleArray;
   ```

   次のような CREATE TYPE ステートメントを使って配列データ・タイプが宣言された場合、

   ```
   CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
   ```

   このデータ・タイプの変数は、例えば次のように作成されます。

   ```
   DECLARE id_Phone_Toronto_List  id_Phone;
   ```

この配列には、1 から 100 までの範囲の整数値で索引が付けられる、VARCHAR(20) データ・タイプ値として保管される電話番号値を最大 100 個まで含めることができます。この変数名は、電話番号値がトロントの電話番号であることを示しています。

2. サポートされるコンテキストの中に DECLARE ステートメントを組み込みます。 CREATE PROCEDURE、CREATE FUNCTION、または CREATE TRIGGER ステートメントの中に入れることができます。
3. DECLARE ステートメントが入っているステートメントを実行します。

ステートメントが正常に実行されます。

DECLARE ステートメントのエラーが原因でそのステートメントが正常に実行されない場合は、次のようにします。

- DECLARE ステートメントの SQL ステートメント構文を確認して、ステートメントを再び実行します。
- 同じ名前を持つ他の変数が同じコンテキスト内で既に宣言されていないか確認します。
- 配列データ・タイプが正常に作成されたことを確認します。

連想配列変数を宣言した後、それらに値を割り当てることができます。

配列への値の割り当て:

配列に値を割り当てるには、いくつかの方法があります。以下のトピックでは、配列に値を割り当てる方法を示します。

副索引およびリテラル値を使用した配列値の割り当て:

副索引とリテラル値を使用して、連想配列に値を割り当てることができます。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- SET ステートメントを実行するのに必要な特権

このタスクを実行するのは、値が割り当てられた変数に対して条件付き SQL PL を実行する前、または変数をパラメーターとしてルーチンに渡す前などです。

1. SET ステートメントを定義します。
   a. 配列変数名を指定します。
   b. 割り当て記号「=」を指定します。
   c. ARRAY キーワードを指定して、値のペアを必須の大括弧の中に指定します。
2. SET ステートメントを実行します。

例えば、myArray という名前の配列にエレメント値を割り当てるには、次のようにします。

```
SET myArray[1]   =  123;
SET myArray[2]   =  124;
...
SET myArray[100] =  223;
```

SET ステートメントが正常に実行されると、配列エレメントが正常に定義されます。配列が作成されたことを検証するには、配列から値の取得を試行することができます。

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- データ・タイプが正常に作成されたことを確認する。

配列値の取得:

配列値を取得するには、いくつかの方法があります。以下のトピックでは、配列から値を取得する方法を示します。

索引を使用した配列値の取得:

配列を参照して副索引値を指定することにより、配列エレメントの値を直接取得することができます。

このタスクの前提条件として、以下が必要です。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- 配列参照を含む SET ステートメントまたは任意の SQL ステートメントを実行するために必要な特権

配列に保管された値にアクセスするには、SQL PL コード内でこのタスクを実行します。割り当て (SET) ステートメントの一部分として、または式の中で直接的に、配列エレメント値にアクセスすることができます。

1. SET ステートメントを定義します。
   a. 配列エレメントと同じデータ・タイプの変数名を指定します。
   b. 割り当て記号「=」を指定します。
   c. 配列の名前と大括弧を指定し、1 つの索引値を大括弧の中に指定します。
2. SET ステートメントを実行します。

配列値を取得する SET ステートメントの例を以下に示します。

```
SET mylocalVar = myArray[1];
```

SET ステートメントが正常に実行されると、配列エレメントの値がローカル変数に含まれます。

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- 変数が配列エレメントと同じデータ・タイプであることを確認する。
- 配列が正常に作成されており、存在していることを確認する。

配列エレメントの数の取得:

単純配列の配列エレメントの数を取得する最も簡単な方法は CARDINALITY 関数を使用することです。また、配列の許容される最大サイズを取得するには MAX_CARDINALITY 関数を使用できます。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- SET ステートメントを実行するのに必要な特権

配列内のエレメント数を示すカウント値にアクセスするには、SQL PL コード内でこのタスクを実行します。割り当て (SET) ステートメントの一部分として配列エレメント値にアクセスすることができます。または、式の中で値に直接アクセスすることもできます。

1. SET ステートメントを定義します。
   a. カーディナリティー値を保持する整数タイプの変数名を宣言して指定します。
   b. 割り当て記号「=」を指定します。
   c. CARDINALITY または MAX_CARDINALTIY 関数の名前を指定し、必須の括弧の中に配列の名前を指定します。
2. SET ステートメントを実行します。

SET ステートメントが正常に実行されると、配列のエレメント数を示すカウント値がローカル変数に含まれます。

以下は、このような割り当てを示す 2 つの SET ステートメントの例です。

```
SET card = CARDINALITY(arrayName);

SET maxcard = MAX_CARDINALITY(arrayName);
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数が整数データ・タイプであることを確認する。
- 配列が正常に作成されており、存在していることを確認する。

最初および最後の配列エレメントの取得 (FIRST 関数と LAST 関数):

単純配列内の最初および最後のエレメントを取得する最も簡単な方法は、FIRST および LAST 関数を使用することです。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- SET ステートメントを実行するのに必要な特権

配列の最初のエレメントに素早くアクセスするには、SQL PL コード内でこのタスクを実行できます。

次のようにして SET ステートメントを定義します。

1. 配列エレメントと同じタイプの変数を宣言して指定します。
2. 割り当て記号「=」を指定します。

3. FIRST または LAST 関数の名前を指定し、必須の括弧の中に配列の名前を指定します。

SET ステートメントが正常に実行されると、配列の最初または最後の索引値を示す値がローカル変数に含まれます。

電話番号の配列が以下のように定義されているとします。

```
firstPhone index    0              1              2              3
          phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

次のような SQL ステートメントを実行した場合、

```
SET firstPhoneIx = FIRST(phones);
```

変数 firstPhoneIx の値は 0 になります (この位置のエレメント値が NULL である場合でも)。

以下の SET ステートメントは、配列内の最初の位置にあるエレメント値にアクセスします。

```
SET firstPhone = A[FIRST(A)]
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数のデータ・タイプが正しいことを確認する。
- 配列が正常に作成されており、存在していることを確認する。

次および前の配列エレメントの取得:

単純配列内の次または前のエレメントを取得する最も簡単な方法は、PREV および NEXT 関数を使用することです。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください
- SET ステートメントを実行するのに必要な特権

配列の中の隣接するエレメント値に素早くアクセスするには、SQL PL コード内でこのタスクを実行します。

1. 次のようにして SET ステートメントを定義します。
   a. 配列エレメントと同じタイプの変数を宣言して指定します。
   b. 割り当て記号「=」を指定します。
   c. NEXT または PREV 関数の名前を指定し、必須の括弧の中に配列の名前を指定します。
2. SET ステートメントを実行します。

電話番号の配列が以下のように定義されているとします。

```
  firstPhone index  0              1              2              3
              phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

以下の SQL ステートメントは、変数 firstPhone の値を 0 に設定します。

```
SET firstPhone = FIRST(phones);
```

以下の SQL ステートメントは、変数 nextPhone の値を 1 に設定します。

```
SET nextPhone = NEXT(phones, firstPhone);
```

以下の SQL ステートメントは、変数 phoneNumber の値を、配列内で nextPhone の次の位置にある電話番号に設定します。これは、索引値 2 の位置にある配列エレメント値です。

```
SET phoneNumber = phones[NEXT(phones, nextPhone)];
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数のデータ・タイプが正しいことを確認する。
- 配列が正常に作成されており、存在していることを確認する。

配列のトリミング (TRIM_ARRAY 関数):

配列の末尾にある不要な配列エレメントを除去するには、TRIM_ARRAY 関数を使って配列のトリミング・タスクを実行します。

- 『配列データ・タイプ』を読んでください
- 『配列データ・タイプに関する制約事項』を読んでください
- SET ステートメントを実行するのに必要な特権

配列の末尾にある配列エレメントを素早く除去するには、SQL PL コード内でこのタスクを実行できます。

1. 次のようにして SET ステートメントを定義します。
   a. 変更対象の配列と同じタイプの配列変数を宣言して指定するか、同じ配列変数を再利用します。
   b. 割り当て記号「=」を指定します。
   c. TRIM_ARRAY 関数の名前を指定し、必須の括弧の中に、配列の名前とトリミング対象のエレメント数を指定します。
2. SET ステートメントを実行します。

SET ステートメントが正常に実行されると、phones 配列に含まれる値が更新されます。

電話番号の配列が以下のように定義されているとします。

```
   phones  index  0       1       2       3
             phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

以下を実行した後、

```
SET phones = TRIM_ARRAY ( phones,  2 );
```

配列 phones は次のように定義されます。

```
phones  index  0       1
         phone  '416-223-2233' '416-933-9333'
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数のデータ・タイプが正しいことを確認する。
- 配列が正常に作成されており、存在していることを確認する。

配列エレメントの削除 (ARRAY_DELETE):

ARRAY_DELETE 関数を使用すると、配列からエレメントを永続的に削除することができます。

- 『配列データ・タイプ』を読んでください
- 『配列データ・タイプに関する制約事項』を読んでください
- SET ステートメントを実行するのに必要な特権

配列の中のエレメントを削除するには、SQL PL コード内でこのタスクを実行します。

1. 次のようにして SET ステートメントを定義します。
   a. 配列エレメントと同じタイプの変数を宣言して指定します。
   b. 割り当て記号「=」を指定します。
   c. ARRAY_DELETE 関数の名前を指定し、必須の括弧の中に、配列の名前、および削除対象のエレメントの範囲を定義する副索引を指定します。
2. SET ステートメントを実行します。

SET ステートメントが正常に実行されると、phones 配列に含まれる値が更新されます。

電話番号の配列が以下のように定義されているとします。

```
phones  index  0            1            2            3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

次のような SQL ステートメントを実行した後、

```
SET phones = ARRAY_DELETE ( phones,  1, 2 );
```

配列 phones は次のように定義されます。

```
phones  index  0            3
         phone  '416-223-2233' '416-722-7227'
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数のデータ・タイプが正しいことを確認する。
- 配列が正常に作成されており、存在していることを確認する。

配列エレメントが存在するかどうかの判別:

ARRAY_EXISTS 関数を使用すると、配列エレメントが存在してそれに値が含まれるかどうかを判別するタスクを実行できます。

- 26 ページの『通常配列データ・タイプ』 を読んでください
- 27 ページの『配列データ・タイプに関する制約事項』 を読んでください

- ARRAY_EXISTS 関数が参照される IF ステートメントまたは任意の SQL ステートメントを実行するために必要な特権。

ある配列エレメントが配列の中に存在するかどうかを判別するには、SQL PL コード内でこのタスクを実行します。

1. 次のようにして IF ステートメントを定義します。
   a. ARRAY_EXISTS 関数を含む条件を定義します。
   b. THEN 節を指定して、条件が真の場合に実行する必要のあるロジックをそこに組み込みます。また、必要に応じて ELSE 節の値を追加します。
   c. END IF 節を使用して IF ステートメントを閉じます。
2. IF ステートメントを実行します。

電話番号の配列が以下のように定義されているとします。

```
phones  index  0             1             2             3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

以下を実行すると、変数 x が 1 に設定されます。

```
IF (ARRAY_EXISTS(phones, 2)) THEN
  SET x = 1;
END IF;
```

SET ステートメントが正常に実行されなかった場合、以下を行ってください。

- SET ステートメントの SQL ステートメント構文を確認して、SET ステートメントを再度実行する。
- ローカル変数のデータ・タイプが正しいことを確認する。
- 配列が正常に作成されており、存在していることを確認する。

SQL プロシージャーでの配列サポート:

SQL プロシージャーは、配列タイプのパラメーターと変数をサポートします。配列は、アプリケーションとストアード・プロシージャーの間で、または 2 つのストアード・プロシージャー間で一時的なデータの集合を受け渡すのに便利な方法です。

SQL ストアード・プロシージャーでは、配列は、標準的なプログラミング言語の配列として操作可能です。また、配列として表されるデータを表に簡単に変換したり、表列内のデータを配列に集約したりするなど、配列はリレーショナル・モデルに統合されます。以下の例では、配列の操作方法をいくつか示しています。どちらの例もコマンド行プロセッサー (CLP) スクリプトで、ステートメント終止符としてパーセント文字 (%) を使用しています。

例 1

この例には、sub と main の 2 つのプロシージャーが示されています。プロシージャー main は、配列コンストラクターを使用して 6 つの整数からなる 1 つの配列を作成します。このプロシージャーは、その後この配列をプロシージャー sum に渡します。プロシージャー sum は、入力配列内のすべての要素の合計を計算し、その結果をプロシージャー main に戻します。プロシージャー sum は、配列の副指標の使用法、および CARDINALITY 関数の使用法の例を示しています。この関数は、配列内の要素数を戻します。

```
create type intArray as integer array[100] %

create procedure sum(in numList intArray, out total integer)
begin
declare i, n integer;

set n = CARDINALITY(numList);

set i = 1;
set total = 0;

while (i <= n) do
set total = total + numList[i];
set i = i + 1;
end while;

end %

create procedure main(out total integer)
begin
declare numList intArray;

set numList = ARRAY[1,2,3,4,5,6];

call sum(numList, total);

end %
```

例 2

この例では、2 つの配列データ・タイプ (intArray および stringArray)、さらには 2 つの列 (id および name) を持つ persons 表を使用します。プロシージャー processPersons は、3 人の人物をこの表にさらに追加し、文字「o」が含まれる人物名を ID 順に並べた配列を返します。追加される 3 人の人物の ID と名前は、2 つの配列 (ids および names) として表されます。これらの配列は UNNEST 関数への引数として使用され、この関数はこうした配列を 2 列からなる表に変換します。その後、この表の要素が persons 表に挿入されます。最後に、このプロシージャーの最後の set ステートメントでは ARRAY_AGG 集約関数を使用して、出力パラメーターの値を計算します。

```
create type intArray as integer array[100] %
create type stringArray as varchar(10) array[100] %

create table persons (id integer, name varchar(10)) %
insert into persons values(2, 'Tom') %
insert into persons values(4, 'Jill') %
insert into persons values(1, 'Joe') %
insert into persons values(3, 'Mary') %

create procedure processPersons(out witho stringArray)
begin
declare ids intArray;
declare names stringArray;

set ids = ARRAY[5,6,7];
set names = ARRAY['Bob', 'Ann', 'Sue'];

insert into persons(id, name)
(select T.i, T.n from UNNEST(ids, names) as T(i, n));

set witho = (select array_agg(name order by id)
from persons
where name like '%o%');
end %
```

連想配列データ・タイプ

連想配列データ・タイプは、事前定義されたカーディナリティーを持たない汎用の配列を表すためのデータ・タイプです。連想配列には、同じデータ・タイプの 0 個以上のエレメントからなる順序付きセットが含まれます。それぞれのエレメントは索引値によって順序付けされ、参照が可能です。

連想配列の索引値は固有で、データ・タイプは同じですが、連続している必要はありません。

以下のトピックでは、連想配列データ・タイプの詳細について説明します。

連想配列のフィーチャー:

連想配列データ・タイプは、連想配列を表すために使用します。役に立つ多くのフィーチャーがあります。

連想配列データ・タイプは、以下の連想配列プロパティーをサポートしています。

- 連想配列用に指定された、定義済みカーディナリティーはありません。 これにより、1 つの集合を構成するエレメントの数を事前に把握していない場合に、最大サイズに注意を払うことなく、配列にエレメントを継続的に追加することができます。
- 添字値を整数以外のデータ・タイプにできます。連想配列索引の索引値として、VARCHAR および INTEGER がサポートされています。
- 索引値は必ずしも連続している必要はありません。位置によって索引付けされた標準的な配列とは対照的に、連想配列は他のデータ・タイプの値によって索引付けされた配列で、最小から最大までの考えうるすべて索引値に関して必須の索引エレメントというものは存在しません。これは、名前や電話番号を格納する集合を作成する場合などに役立ちます。対のデータをどのような順序でもその集合に追加できますし、索引として定義されている対となっているいずれのデータ項目を使用してもソートできます。
- 連想配列内のエレメントを、索引値の昇順でソートします。エレメントの掲載順序は重要ではありません。
- 連想配列データは、直接的な参照でも、使用可能な一群のスカラー関数を使用しても、アクセスおよび設定が可能です。
- SQL PL コンテキストで連想配列はサポートされています。
- 連想配列を使用して、同じ種類の値の集合をコレクションの形式で管理および引き渡すことができます。そのため、以下の必要がなくなります。
  - スカラー値へのデータの削減と、一度に 1 つのエレメント処理の使用。この処理により、ネットワーク・トラフィックの問題が生じる恐れがあります。
  - パラメーターとして渡されるカーソルの使用。
  - スカラー値へのデータの削減と、VALUES 節による集合としてそれらのデータの再構成。

連想配列データ・タイプに関する制約事項:

この配列データ・タイプを使用する前に、またはその宣言や使用に関連する問題をトラブルシューティングする場合、その制約事項に注意を払うのは大切です。

この配列データ・タイプには以下の制約事項が適用されます。

- 連想配列は、SQL PL コンテキストにおいてのみ宣言、作成、または参照できます。以下に、このデータ・タイプを使用できる SQL PL コンテキストをリストします。
  - モジュール内で定義されている SQL 関数に対するパラメーター。
  - モジュール内で定義されていないものの、モジュール内で定義されていない関数本体としてコンパウンド SQL (コンパイル済み) ステートメントがある SQL 関数に対するパラメーター。
  - モジュール内で定義されている SQL 関数からの戻りタイプ。
  - モジュール内で定義されていないものの、関数本体としてコンパウンド SQL (コンパイル済み) ステートメントがある SQL 関数からの戻りタイプ。
  - SQL プロシージャーに対するパラメーター。
  - モジュール内で定義されている SQL 関数で宣言済みのローカル変数。
  - モジュール内で定義されていないものの、関数本体としてコンパウンド SQL (コンパイル済み) ステートメントがある SQL 関数で宣言済みのローカル変数。
  - SQL プロシージャーで宣言済みのローカル変数。
  - トリガー本体としてコンパウンド SQL (コンパイル済み) ステートメントを使用する、トリガー内の宣言済みのローカル変数。
  - コンパウンド・コンパイル済み (SQL) ステートメント内の SQL ステートメントの式。
  - SQL PL コンテキスト内の SQL ステートメントの式。
  - グローバル変数。

  上記の SQL PL コンテキスト以外で使用されると無効です。
- 連想配列を表列のタイプとすることはできません。
- 索引値として NULL は許可されていません。
- 連想配列の最大サイズは、システム・リソースによって制限されます。
- 連想配列を、TRIM_ARRAY 関数に含めることはできません。連想配列値は、表列には保管できません。
- MAX_CARDINALITY 関数を連想配列とともに使用できますが、必ず NULL が戻ります。連想配列は特定の最大サイズを持たないためです。

連想配列データ・タイプの作成:

連想配列データ・タイプの作成は、通常、連想配列データ・タイプの変数を作成するための前提条件として実行されるタスクです。 CREATE TYPE (配列) ステートメントを実行することで、連想配列データ・タイプが作成されます。

CREATE TYPE ステートメントの実行に必要な特権があることを確認します。

連想配列データ・タイプは、いくつかのコンテキストでのみ使用可能です。

1. 次のようにして CREATE TYPE ステートメントを定義します。
   a. 連想配列データ・タイプの名前を指定します。 配列に保管されるデータの種類を明確に示すような名前を付けるのが適切です。例えば、いくつかの製品に関する情報を格納し、製品 ID が添字となるような配列の場合、Products

(製品) という名前にするのが適切でしょう。別の例として、グラフ関数の x 座標値が添字となるような配列の場合、y_coordinate (y 座標) という名前が適切でしょう。

a. AS キーワードを指定し、その後に配列エレメントのデータ・タイプのキーワード名を指定します (例えば INTEGER)。

b. ARRAY キーワードを指定します。 ARRAY 節の大括弧の中で、添字のデータ・タイプを指定します。なお、連想配列の場合、エレメントの数、および添字値のドメインに関する明示的な制限がないことに注意してください。

2. サポートされるインターフェースから CREATE TYPE ステートメントを実行します。

例 1:
以下の例に示す CREATE TYPE ステートメントは、20 個のエレメントとタイプ VARCHAR の添字を持つ assocArray という名前の配列を作成します。

```
CREATE TYPE assocArray AS INTEGER ARRAY[VARCHAR(20)];
```

例 2:
以下の例に示す基本的な連想配列の定義では、州の名前が索引として、その州都がエレメントとして使用されています。

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

ステートメントが正常に実行されると、配列データ・タイプがデータベースに作成されて、その配列データ・タイプは参照可能になります。

配列データ・タイプを作成した後、連想配列変数を作成することができます。

連想配列変数の宣言:

連想配列データ・タイプ値を一時的に保管または受け渡すことができるようにするには、連想配列データ・タイプを作成した後のタスクとして、連想配列変数を宣言します。ローカル変数は DECLARE ステートメントを使って宣言されます。グローバル変数は CREATE VARIABLE ステートメントを使って作成されます。

- 『連想配列データ・タイプ』を読んでください
- 『連想配列データ・タイプに関する制約事項』を読んでください
- 『連想配列データ・タイプの作成』を読んでください
- グローバル変数の場合、CREATE VARIABLE ステートメントを実行するための特権が必要です。ローカル変数の場合、DECLARE ステートメントの実行に必要な特権はありません。

連想配列変数を宣言および使用できるのは、行データのセットの保管がサポートされるコンテキストです。

1. 次のようにして、ローカル変数用の DECLARE ステートメントまたはグローバル変数用の CREATE TYPE ステートメントを定義します。

   a. 連想配列データ・タイプの名前を指定します。

   b. 連想配列データ・タイプの作成時に使用した連想配列データ・タイプの名前を指定します。

2. サポートされるインターフェースから CREATE TYPE ステートメントを実行します。

例 1:
連想配列データ・タイプが次のように定義されているとします。

```
CREATE TYPE Representative_Location AS VARCHAR(20) ARRAY[VARCHAR(30)];
```

このデータ・タイプの変数を宣言するには、次のような DECLARE ステートメントを使用できます。

```
DECLARE RepsByCity Representative_Location;
```

この配列には、固有の可変文字データ・タイプ値によって索引が付けられる、VARCHAR(20) データ・タイプ値として保管される連想配列エレメント値を、最大数まで含めることができます。変数名は、営業担当員の名前から成るセットに対して、それぞれ担当する都市の名前で索引が付けらけることを示しています。この配列では、同じ都市 (つまり添字値) で表される営業担当員の名前が 2 つ以上存在することはありません。

例 2:
次のように、州都の名前をエレメント値として保管し、州の名前を索引とする連想配列データ・タイプが定義されているとします。

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

このデータ・タイプの変数を作成するには、次のような CREATE VARIABLE ステートメントを使用できます。

```
CREATE VARIABLE capitals capitalsArray;
```

この配列には、固有の可変文字データ・タイプ値によって索引が付けられる、VARCHAR(20) データ・タイプ値として保管される連想配列エレメント値を、最大数まで含めることができます。変数名は、営業担当員の名前から成るセットに対して、それぞれ担当する都市の名前で索引が付けらけることを示しています。この配列では、同じ都市 (つまり添字値) で表される営業担当員の名前が 2 つ以上存在することはありません。

DECLARE ステートメントまたは CREATE VARIABLE ステートメントが正常に実行されると、配列データ・タイプが正常に定義されて参照可能になります。連想配列変数が作成されたことを検証するには、配列に値を割り当てるか、配列内の値を参照してみることができます。

DECLARE ステートメントまたは CREATE VARIABLE ステートメントが正常に実行されない場合は、DECLARE ステートメントの SQL ステートメント構文を確認して、ステートメントを再び実行します。 DECLARE ステートメントを参照してください。

副索引とリテラル値を使用した配列への値の割り当て:

連想配列変数を作成または宣言したら、それに値を割り当てることができます。連想配列に値を割り当てる 1 つの方法は、直接割り当てです。

- 『連想配列データ・タイプ』を読んでください
- 『連想配列データ・タイプに関する制約事項』を読んでください

- 連想配列変数が現在の有効範囲で使用されていることを確認してください。

代入文を使って連想配列変数エレメントに値を割り当てることができます。その際、配列の名前を示し、索引値を指定して、対応するエレメント値を割り当てます。

1. 連想配列変数の代入文を定義します。
   - 変数名、索引値、およびエレメント値を指定します。
   - 別の連想配列変数を指定します。
2. サポートされるインターフェースから代入文を実行します。

例 1:
以下の例は、変数宣言と、配列内の値を定義する一連の代入文を示しています。

```
DECLARE capitals capitalsArray;

SET capitals['British Columbia'] = 'Victoria';
SET capitals['Alberta'] = 'Edmonton';
SET capitals['Manitoba'] = 'Winnipeg';
SET capitals['Ontario'] = 'Toronto';
SET capitals['Nova Scotia'] = 'Halifax';
```

配列 capitals (州都) の中で、添字値は州の名前であり、連想配列エレメント値はそれぞれに対応する州都の名前です。連想配列は索引値に基づいて昇順でソートされます。連想配列の各エレメントに値が割り当てられる順序は、重要ではありません。

例 2:
連想配列変数に、同じ連想配列データ・タイプを持つ連想配列変数の値を割り当てることもできます。代入文を使用して、これを行うことができます。例えば、2 つの連想配列変数 capitalsA と capitalsB が次のように定義されているとします。

```
DECLARE capitalsA capitalsArray;
DECLARE capitalsB capitalsArray;

SET capitalsA['British Columbia'] = 'Victoria';
SET capitalsA['Alberta'] = 'Edmonton';
SET capitalsA['Manitoba'] = 'Winnipeg';
SET capitalsA['Ontario'] = 'Toronto';
SET capitalsA['Nova Scotia'] = 'Halifax';
```

以下の代入文を実行することにより、変数 capitalsA の値を変数 capitalsB に割り当てることができます。

```
SET capitalsB = capitalsA;
```

これを実行すると、capitalsB の値は capitalsA と同じになります。

代入文が正常に実行されると、値が正常に割り当てられて、新しい変数値が参照可能になります。

ステートメントが正常に実行されない場合は、SQL ステートメント構文を確認して訂正し、指定された変数が定義されていることを確認して、ステートメントを再び実行してください。

カーソル・タイプ

カーソル・タイプとなり得るのは、組み込みデータ・タイプ CURSOR、または組み込み CURSOR データ・タイプに基づくユーザー定義タイプです。またユーザー定義のカーソル・タイプを特定の行タイプを使用して定義し、関連するカーソルの結果行の属性を制限することもできます。

カーソル・タイプが (行によって指定されている) 行データ構造と関連付けられている場合、厳密に型付けされたカーソルと呼ばれます。厳密に型付けされたカーソル・データ・タイプの変数に割り当てて保管できるのは、定義が一致する結果セットだけです。結果セット定義がカーソル・データ・タイプ定義と関連付けられていない場合、カーソル・データ・タイプは緩やかに型付けされていると言います。緩やかに型付けされたカーソル・データ・タイプの変数には、どのような結果セットでも保管できます。

このカーソル・データ・タイプがサポートされているのは、SQL PL を使用する場合のみです。このカーソル・データ・タイプは、カーソル変数宣言で使用可能なカーソル・タイプ定義を作成するために本来使用されます。

このデータ・タイプを使用して以下が可能です。

- カーソル変数宣言を定義します。
- SQL プロシージャー型言語アプリケーション内でのロジックのコーディングを簡略化します。例えば、データベース・アプリケーションは、結果セットと呼ばれるレコードの集合を処理し、場合によっては同じ結果セットを別のコンテキストで参照および処理する必要があります。インターフェース間で定義済み結果セットを受け渡すには、複雑なロジックが求められる場合があります。カーソル・データ・タイプを使用するとカーソル変数の作成が可能で、カーソル変数を使用して、結果セットの保管、処理、およびパラメーターとしての受け渡しを行えます。
- 同様のデータ・タイプのコードを DB2 SQL PL に移植するのが容易になります。

カーソル・データ・タイプは、CREATE TYPE ステートメントを使用して作成する必要があります。このデータ・タイプの変数は、一度作成すると、宣言および参照が可能になります。カーソル変数は、行データ構造定義に割り当てることや、オープンやクローズ、別のカーソル変数の行の集合の割り当て、または SQL プロシージャーからのパラメーターとして引き渡しが可能です。

カーソル・データ・タイプの概要

カーソル・データ・タイプの概要では、カーソル・データ・タイプの種類、使用できる範囲について取り上げ、使用する際の制限事項と関連する特権についての情報も紹介します。

カーソル・データ・タイプの種類:

カーソル・データ・タイプには、大別すると緩やかに型付けされたカーソル・データ・タイプと厳密に型付けされたカーソル・データ・タイプの 2 つの種類があります。厳密に型付けされるか緩やかに型付けされるかのプロパティーは、データ・タイプの作成時に定義されます。このプロパティーは、各タイプの作成済み変数に維持されます。

厳密に型付けされたカーソル・データ・タイプおよび緩やかに型付けされたカーソル・データ・タイプの特性について以下に取り上げます。

厳密に型付けされたカーソル・データ・タイプ

厳密に型付けされたカーソル・データ・タイプは、行データ構造によって指定された結果セット定義を使用して作成されたカーソル・データ・タイプです。こうしたデータ・タイプが厳密に型付けされたと言われているのは、結果セット値が割り当てられる際、結果セットのデータ・タイプがチェック可能なためです。 カーソル・データ・タイプの結果セット定義は、行タイプ定義を指定すると定義できます。厳密に型付けされたカーソル・データ・タイプに割り当てて保管できるのは、定義が一致する結果セットだけです。割り当て時に厳密な型判定チェックが実行され、データ・タイプの不一致が見つかるとエラーが生じます。

厳密に型付けされたカーソル・データ・タイプの結果セット定義は、行データ・タイプ定義または SQL ステートメント定義によって提供できます。

以下は、カーソル・データ・タイプ定義の例で、rowType データ・タイプと同じ行フォーマットを持つ結果セットを戻すように定義されています。

```
CREATE TYPE cursorType AS rowType CURSOR@
```

rowType 行定義と同じデータ・タイプ定義を持つデータの列が含まれる結果セットのみを、cursorType カーソル・データ・タイプとなるように宣言された変数に正常に割り当てることができます。

以下は、表 T1 で定義されているのと同じ行フォーマットを持つ結果セットを戻すように定義されている、カーソル・データ・タイプ定義の例です。

```
CREATE TABLE T1 (C1 INT)

  CREATE TYPE cursorType AS ANCHOR ROW OF t1 CURSOR;
```

表 t1 の列定義と同じデータ・タイプ定義を持つデータの列が含まれる結果セットのみを、cursorType カーソル・データ・タイプとなるように宣言された変数に正常に割り当てることができます。

厳密に型付けされたカーソルに関連付けられている行定義は、アンカー・データ・タイプの定義として参照できます。以下はそのことを示す例です。

```
CREATE TYPE r1 AS ROW (C1 INT);
CREATE TYPE c1 AS RTEST CURSOR;

DECLARE c1 CTEST;
DECLARE r1 ANCHOR ROW OF CV1;
```

r1 という名前の行データ・タイプが定義され、r1 の行定義に関連付けられている c1 という名前のカーソル・タイプが定義されます。後半の SQL ステートメントは、SQL プロシージャーで出現する可能性のある変数宣言の例です。2 番目の変数宣言は、アンカー・データ・タイプとなるように定義された r1 という名前の変数用で、カーソル cv1 を定義するために使用された行タイプにアンカー付けされます。

緩やかに型付けされたカーソル・データ・タイプ

緩やかに型付けされたカーソル・データ・タイプは、いずれの行データ・タイプ定義とも関連付けられていません。緩やかに型付けされたカーソル変数に値を割り当てる際には、型判別チェックは実行されません。

CURSOR という名前の、システム定義の緩やかに型付けされたカーソル・データ・タイプがあり、緩やかに型付けされたカーソル変数またはパラメーターを宣言するために使用できます。以下は、システム定義の緩やかに型付けされたカーソル・データ・タイプ CURSOR に基づく、緩やかに型付けされたカーソル変数宣言の例です。

```
DECLARE cv1 CURSOR;
```

緩やかに型付けされたカーソル変数は、結果セットを不明な行定義で保管する必要がある場合に役立ちます。
緩やかに型付けされたカーソル変数を出力パラメーターとして戻すには、カーソルがオープンしていなければなりません。
今回のバージョンでは、緩やかに型付けされたカーソル・データ・タイプに基づく変数をアンカー・データ・タイプとして参照することはできません。
ユーザー定義の緩やかに型付けされたカーソル・データ・タイプを定義できます。

カーソル変数のその他の特性すべては、各タイプのカーソル変数で共通です。

カーソル・データ・タイプに関する制約事項:

カーソル・データ・タイプおよびカーソル変数に関する制約事項によって、カーソル変数の機能や、カーソル変数を定義および参照できる場面が制限されます。

カーソル・データ・タイプと変数に関する制約事項は、それらをインプリメントする前に注意を払うことが大切です。これらの制約事項は、カーソル変数がニーズに適合するかどうかを判別する上で重要となりますし、カーソル・データ・タイプと変数の使用に関連するエラーをトラブルシューティングする際に考慮すると役立ちます。

今回のバージョンでは、以下の制約事項がカーソル・データ・タイプに適用されます。

- カーソル・データ・タイプは、SQL プロシージャーのローカル・タイプとしてのみ作成できます。

今回のバージョンでは、以下の制約事項がカーソル変数に適用されます。

- カーソル変数をアプリケーション内で使用することはできません。カーソル変数は、SQL PL コンテキストにおいてのみ宣言および参照できます。
- カーソル変数は、読み取り専用カーソルです。
- カーソル変数を使用してアクセスする行を、更新することはできません。
- カーソル変数は、スクロール可能カーソルではありません。
- 厳密に型付けされたカーソル変数列を、アンカー・データ・タイプとして参照することはできません。
- グローバル・カーソル変数はサポートされていません。
- XML 列を、カーソル変数定義で参照することはできません。
- XQuery 言語ステートメントを使用して、厳密に型付けされたカーソルの結果セットを定義することはできません。

カーソル・データ・タイプの使用に関連した特権:

カーソル・データ・タイプおよび変数を作成できるユーザーを制限および制御するために、これらに関連した特定の特権が存在します。

カーソル・データ・タイプを作成するには、以下の特権が必要です。

- カーソル・データ・タイプを作成するために CREATE TYPE ステートメントを実行する特権。

既存のカーソル・データ・タイプに基づいてカーソル変数を宣言するためには、特権は不要です。

カーソル変数を初期化するため、カーソル変数が参照するカーソルをオープンするため、またはオープンされているカーソル変数参照から値をフェッチするには、DECLARE CURSOR ステートメントを実行する際に必要な特権と同様の特権が求められます。

カーソル変数

カーソル変数は、定義済みのカーソル・データ・タイプに基づくカーソルです。カーソル変数は未初期化にすることも、初期化することもできます。また、値の割り当て、別の値への設定、SQL プロシージャーからのパラメーターとしての受け渡しが可能です。カーソル変数は、ベースとなるカーソル・データ・タイプのプロパティーを継承します。カーソル変数は、厳密に型付けされたタイプにも緩やかに型付けされたタイプにもできます。カーソル変数は、カーソル・データ・タイプによって定義されたカーソルのコンテキストに対する参照を保持します。

カーソル変数は、DECLARE ステートメントを使用して SQL プロシージャー内で宣言できます。

カーソル述部

カーソル述部は、現行の有効範囲内に定義されたカーソルの状態を調べるために使用する SQL キーワードです。これを使用して、カーソルがオープンされたか、クローズされたか、またはカーソルに関連付けられている行があるかどうかを簡単に参照できます。

カーソル述部は、カーソルの状態を述部条件として使用できる場合にはいつでも、SQL および SQL PL ステートメントで参照できます。使用できるカーソル述部は、以下のとおりです。

IS OPEN
この述部は、カーソルがオープン状態にあるかどうかを判別するために使用できます。カーソルがパラメーターとして関数およびプロシージャーに渡されるときに有効な述部です。カーソルのオープンを試行する前に、この述部を使用してカーソルが既にオープンされていないかどうかを調べることができます。

IS NOT OPEN
この述部は、カーソルがクローズされているかどうかを判別するために使用できます。この値は、IS OPEN の逆ロジックです。この述部は、カーソルを実際にクローズする前にカーソルがクローズされているかどうかを判別するのに役立ちます。

IS FOUND

この述部は、FETCH ステートメントの実行後にカーソルが行を保有しているかどうかを検査するために使用できます。最後に実行された FETCH ステートメントが成功した場合、IS FOUND 述部の値は TRUE になります。最後に実行された FETCH ステートメントの結果、行が存在しなかった場合、戻される値は FALSE となります。以下の場合に、結果は不明となります。

- cursor-variable-name の値が NULL である
- cursor-variable-name の事前に宣言されているカーソルがオープンしていない。
- 事前に宣言されているカーソル上で最初の FETCH 操作が実行される前に、述部が評価された
- 最後の FETCH 操作でエラーが戻された

IS FOUND 述部は、ループ内でフェッチの実行を反復する SQL PL ロジック部分に有用です。この述部は、フェッチする残りの行があるかどうかを調べるためにも使用できます。フェッチする残りの行がないというエラー条件を検査する条件処理ルーチンの代わりとなる効果的な選択肢です。

IS FOUND を使用する代わりに、逆の値を持つ IS NOT FOUND を使用します。

例

以下のスクリプトは、プロシージャーのコンパイルおよび呼び出しを成功させるために必要な前提条件オブジェクトを定義し、さらに、カーソル述部の参照を行う SQL プロシージャーを定義します。

```
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)@

insert into t1 values (1,1,1),(2,2,2),(3,3,3) @

CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)@

CREATE TYPE myCursorType AS myRowType CURSOR@


CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
  DECLARE C1 cursor;
  DECLARE lvarInt INT;

  SET count = -1;
  SET c1 = CURSOR FOR SELECT c1 FROM t1;

  IF (c1 IS NOT OPEN) THEN
     OPEN c1;
  ELSE
     set count = -2;
  END IF;

  set count = 0;
  IF (c1 IS OPEN) THEN

    FETCH c1 into lvarInt;

    WHILE (c1 IS FOUND) DO
      SET count = count + 1;
      FETCH c1 INTO lvarInt;
```

```
    END WHILE;
  ELSE
     SET count = 0;
  END IF;

END@

CALL p()@
```

カーソル変数の作成

カーソル変数を作成するには、まずカーソル・タイプを作成した後、そのタイプに基づいてカーソル変数を作成する必要があります。以下のトピックでは、これらのタスクを行う方法について説明します。

CREATE TYPE ステートメントを使用したカーソル・データ・タイプの作成:

カーソル変数を作成するための前提条件として、カーソル・データ・タイプを作成する必要があります。カーソル・データ・タイプは CREATE TYPE (cursor) ステートメントを使って作成されます。

このタスクを実行するには、以下が必要です。

- CREATE TYPE (cursor) ステートメントを実行するための特権。
- 厳密に型付けされたカーソル・データ・タイプを作成する場合、行指定を準備するか、表、ビュー、またはカーソルの既存の行をベースとする必要があります。

CREATE TYPE (cursor) ステートメントによって定義されるカーソル・データ・タイプを SQL PL で使用すると、カーソル・データ・タイプのパラメーターとローカル変数を宣言することができます。 CREATE TYPE (cursor) ステートメントで row-type-name 節を指定すると、厳密に型付けされたカーソル・データ・タイプが作成されます。 row-type-name 節を省略した場合には、緩やかに型付けされたカーソル・データ・タイプが作成されます。

緩やかに型付けされたカーソル・データ・タイプを作成する代わりに、システム定義の緩やかに型付けされたカーソル・データ・タイプ CURSOR をカーソル変数の宣言時に使用することもできます。

```
CREATE TYPE weakCursorType AS CURSOR@
```

厳密に型付けされたカーソル・データ・タイプを作成するには、カーソルとの関連付けが可能な結果セットを定義する行データ・タイプ定義が存在しなければなりません。明示的に定義された行データ・タイプ、表またはビュー、あるいは厳密に型付けされたカーソルから行データ・タイプ定義を派生させることができます。行タイプ定義の例を以下に示します。

```
CREATE TYPE empRow AS ROW (name varchar(128), ID varchar(8))@
```

例えば以下のような表定義から、行タイプ定義を派生させることができます。

```
CREATE TABLE empTable AS ROW (name varchar(128), ID varchar(8))@
```

厳密に型付けされたカーソル・データ・タイプをデータベース内に定義するには、SQL ステートメントの実行をサポートする任意の DB2 インターフェースから CREATE TYPE (CURSOR) ステートメントを正常に実行する必要があります。

1. 次のようにして CREATE TYPE (CURSOR) ステートメントを定式化します。

a. タイプの名前を指定します。
b. 次のいずれかの方法で行定義を指定します。つまり、行データ・タイプの名前を参照するか、タイプを表またはビューにアンカーすることを指定するか、あるいは既存の厳密なカーソル・タイプに関連した結果セット定義にアンカーすることを指定するかのいずれかです。

2. サポートされる DB2 インターフェースから CREATE TYPE ステートメントを実行します。

CREATE TYPE ステートメントが正常に実行されると、カーソル・データ・タイプがデータベースに作成されます。

以下の例は、empRow 行データ・タイプと同じ形式の結果セットへの関連付けが可能な、緩やかに型付けされたカーソル・データ・タイプを作成する方法を示しています。

```
CREATE TYPE cursorType AS empRow CURSOR@
```

以下の例は、表 empTable と同じ形式の結果セットに関連付けることができるカーソル・データ・タイプの作成方法を示しています。

```
CREATE TYPE cursorType AS ANCHOR ROW OF empTable@
```

カーソル・データ・タイプを作成したら、このデータ・タイプに基づいてカーソル変数を宣言することができます。

カーソル・タイプのローカル変数の宣言:

カーソル・データ・タイプの作成が完了した後、カーソル・タイプのローカル変数を宣言することができます。

カーソル・データ・タイプ定義がデータベースに存在している必要があります。CREATE TYPE (CURSOR) ステートメントが正常に実行されると、カーソル・データ・タイプが作成されます。以下の例は、厳密に型付けされたカーソル・タイプ定義です。

```
CREATE TYPE cursorType AS empRow CURSOR;
```

この場合、SQL プロシージャーの中でローカル変数としてのみカーソル変数を宣言できます。厳密に型付けされたカーソル変数と緩やかに型付けされたカーソル変数の両方を宣言できます。

1. 次のようにして DECLARE ステートメントを定式化します。
   a. 変数の名前を指定します。
   b. 変数を定義するカーソル・データ・タイプを指定します。カーソル変数を緩やかに型付けする場合、ユーザー定義の緩やかに型付けされたカーソル・データ・タイプを指定するか、システム定義の緩やかに型付けされたカーソル・データ・タイプ CURSOR を使用する必要があります。厳密に型付けされたカーソル・データ・タイプに基づくカーソル変数にする場合は、変数を直ちに初期化することができます。

   以下の例で編成される DECLARE ステートメントは、初期化されない cursorType タイプのカーソル変数を定義します。

   ```
   DECLARE Cv1 cursorType@
   ```

以下の例で定式化される DECLARE ステートメントは、Cv1 という名前の既存のカーソル変数のタイプにアンカーされるタイプを使用するカーソル変数 Cv2 を定義します。

```
DECLARE Cv2 ANCHOR DATA TYPE TO Cv1@
```

以下の例で定式化される DECLARE ステートメントは、緩やかに型付けされたカーソル変数を定義します。

```
DECLARE Cv1 CURSOR@
```

2. サポートされるコンテキストで DECLARE ステートメントを実行します。

DECLARE ステートメントが正常に実行されると、カーソル変数が作成されます。

このカーソル変数を作成した後、カーソル変数に値を割り当てたり、カーソル変数を参照したり、パラメーターとして渡すことができます。

カーソル変数への値の割り当て

SET ステートメントを使用すると、いくつかの方法でさまざまな時点で結果セットをカーソル変数に割り当てることができます。

カーソル変数への照会結果セットの割り当て

SET ステートメントと CURSOR FOR キーワードを使用することにより、選択照会の結果セットをカーソル変数に割り当てることができます。以下の例では、T という表への照会に関連した結果セットが、表と同じ行定義を持つ c1 という名前のカーソル変数に割り当てられます。

T が次のように定義される場合、

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

次のように定義された、厳密に型付けされたカーソル変数 C1 を想定すると、

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
DECLARE c1 simpleCur;
```

次のように割り当てることができます。

```
SET c1 = CURSOR FOR SELECT * FROM T;
```

c1 の定義は表 T と互換であるため、厳密な型判定の検査が成功します。 c1 が緩やかに型付けされたカーソルであれば、データ・タイプ検査は実行されないため、この場合もこの割り当てが成功します。

カーソル変数へのリテラル値の割り当て

SET ステートメントと CURSOR FOR キーワードを使用することにより、選択照会の結果セットをカーソル変数に割り当てることができます。以下の例では、T という表への照会に関連した結果セットが、表と同じ行定義を持つ c1 という名前のカーソル変数に割り当てられます。

表 T が次のように定義されるとします。

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

行タイプ simpleRow、およびカーソル・タイプ simpleCur がそれぞれ次のように作成されるとします。

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
```

厳密に型付けされたカーソル変数 c1 がプロシージャー内で次のように宣言されるとします。

```
DECLARE c1 simpleCur;
```

次のようにして、リテラル値をカーソル c1 に割り当てることができます。

```
SET c1 = CURSOR FOR VALUES (1, 2, 3);
```

リテラル値はカーソル定義と互換であるため、厳密な型判定の検査が成功します。以下の例では、リテラル・データ・タイプがカーソル・タイプ定義と非互換であるため、リテラル値の割り当てが失敗します。

```
SET c1 = CURSOR FOR VALUES ('a', 'b', 'c');
```

カーソル変数値へのカーソル変数値の割り当て

2 つのカーソル変数の結果セット定義が同じである場合に限り、1 つのカーソル変数値を別のカーソル変数に割り当てることができます。以下に例を示します。

厳密に型付けされたカーソル変数 c1 および c2 が次のように定義されているとします。

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow

DECLARE c1 simpleCur;

DECLARE c2 simpleCur;
```

次のように c2 に値が割り当てられたとします。

```
SET c2 = CURSOR FOR VALUES (1, 2, 3);
```

次のようにして、c2 の結果セットをカーソル変数 c1 に割り当てることができます。

```
 SET c1 = c2;
```

カーソル変数に値を割り当てた後、カーソル変数およびカーソル変数フィールド値の割り当てや参照が可能になります。

カーソル変数の参照

結果セットの取得やアクセスに関連したカーソル操作を行う際、またはプロシージャーを呼び出してカーソル変数をパラメーターとして渡す際などに、さまざまな方法でカーソル変数を参照することができます。

SQL PL コンテキストにおいて、以下のステートメントを使用してカーソル変数を参照できます。

- CALL
- SET

- OPEN
- FETCH
- CLOSE

カーソル変数に関連した結果セットにアクセスする際には、多くの場合、OPEN、FETCH、および CLOSE ステートメントを共に使用します。 OPEN ステートメントは、カーソル変数に関連した結果セットを初期化するために使用されます。このステートメントの実行が成功すると、カーソル変数が結果セットに関連付けられて、結果セットの行がアクセス可能になります。 FETCH ステートメントは、カーソル変数によってアクセスされる現在行の特定の列値を取得するために使用されます。 CLOSE ステートメントは、カーソル変数の処理を終了するために使われます。

以下の例は、作成された行データ・タイプ定義、およびカーソル変数定義を含む SQL プロシージャー定義を示しています。 SQL プロシージャーの中で、カーソル変数と共に OPEN、FETCH、および CLOSE ステートメントが使用されています。

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE PROCEDURE P(OUT p1 INT, OUT p2 INT, PUT p3 INT, OUT pRow simpleRow)
LANGUAGE SQL
BEGIN

      CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow
      DECLARE c1 simpleCur;
      DECLARE localVar1 INTEGER;
      DECLARE localVar2 INTEGER;
      DECLARE localVar3 INTEGER;
      DECLARE localRow simpleRow;

      SET c1 = CURSOR FOR SELECT * FROM T;

      OPEN C1;

      FETCH c1 INTO localVar1, localVar2, localVar3;

      FETCH c1 into localRow;

      SET p1 = localVar1;

      SET p2 = localVar2;

      SET p3 = localVar3;

      SET pRow = localRow;

      CLOSE c1;

END;
```

また、CALL ステートメントでパラメーターとしてカーソル変数を参照することもできます。他のパラメーターと同様に、カーソル変数パラメーターは単に名前で参照されます。以下の例は、出力パラメーターである curVar というカーソル変数を参照する SQL プロシージャー内の CALL ステートメントを示しています。

```
CALL P2(curVar);
```

カーソルに取り出された行数の判別

cursor_rowCount スカラー関数を使用すると、カーソルに関連した行の数を効率的に判別することができます。この関数はカーソル変数をパラメーターとして取り、カーソルのオープン時から現在までに取り出された行数を示す整数値を出力として戻します。

cursor_rowCount 関数を使用する前に、以下の前提条件が満たされる必要があります。

- カーソル・データ・タイプが作成されている必要があります。
- カーソル・データ・タイプのカーソル変数が宣言されている必要があります。
- カーソルを参照する OPEN ステートメントが実行済みでなければなりません。

SQL PL コンテキストで cursor_rowCount 関数を使ってこのタスクを実行することができます。プロシージャー・ロジックの中で、あるカーソルに既に取り出された行数を示すカウント、または取り出された行数の合計カウントにアクセスする必要があるような場合に、これを実行します。 cursor_rowCount 関数を使用すると、取り出し済み行カウントに簡単にアクセスできます。この方法を使用しない場合、変数を宣言して SET ステートメントを繰り返し実行することにより、プロシージャー・ロジックのループの中でカウントを維持する必要があるでしょう。

制約事項

cursor_rowCount 関数は SQL PL コンテキストでのみ使用可能です。

1. cursor_rowCount スカラー関数を参照する SQL ステートメントを定式化します。 以下の例の SET ステートメントは、cursor_rowCount スカラー関数の出力を rows_fetched というローカル変数に割り当てます。

   ```
   SET rows_fetched = CURSOR_ROWCOUNT(curEmp)
   ```

2. cursor_rowCount 関数参照を含んでいる SQL ステートメントを、サポートされる SQL PL コンテキストに組み込みます。例えば CREATE PROCEDURE ステートメントや CREATE FUNCTION ステートメントの中に入れて、ステートメントをコンパイルします。
3.

ステートメントが正常にコンパイルされます。

以下の例は、cursor_rowCount 関数の参照を含む SQL プロシージャーです。

```
CREATE PROCEDURE p()
LANGUAGE SQL
BEGIN

SET rows_fetched = CURSOR_ROWCOUNT(curEmp)

END@
```

SQL プロシージャーを実行するか、SQL 関数を呼び出します。

例: カーソル変数の使用

カーソル変数の使用例を参照すると、カーソル変数を設計およびインプリメントする際に役立ちます。

SQL プロシージャーにおけるカーソル変数の使用:

カーソル変数の使用法を示す例を参照すると、可能なカーソル変数の使用法と使用箇所について理解を深めるのに役立ちます。

この例では、以下について説明します。

- CREATE TYPE ステートメントによる ROW データ・タイプの作成
- CREATE TYPE ステートメントによる、行データ・タイプ指定に基づく厳密に型付けされたカーソルの作成
- CREATE PROCEDURE ステートメントによる、出力カーソル・パラメーターを持つプロシージャーの作成
- CREATE PROCEDURE ステートメントによる、別のプロシージャーを呼び出し、入力パラメーターとしてカーソルを引き渡すプロシージャーの作成

この例を実行するための前提条件として、SAMPLE データベースが存在していなければなりません。このサンプル・データベースを作成するには、DB2 コマンド・ウィンドウから以下のコマンドを実行してください。

```
db2sampl;
```

以下は、SQL プロシージャーで使用するカーソル変数の中核となるフィーチャーについて示している CLP スクリプトの例です。このスクリプトには、行データ・タイプ定義、カーソル・タイプ定義、および 2 つの SQL プロシージャー定義が含まれています。プロシージャー P_CALLER には、カーソル変数定義と、P という名前のプロシージャーへの呼び出しが含まれています。プロシージャー P はカーソルを定義し、そのカーソルをオープンして、カーソルを出力パラメーター値として引き渡します。プロシージャー P_CALLER はカーソル・パラメーターを受け取り、カーソル値をローカル変数にフェッチしてから、そのローカル変数値に基づいて edlvel と lastname という 2 つの出力パラメーター値を設定します。

```
--#SET TERMINATOR @
update command options using c off @
connect to sample @

CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))@

CREATE TYPE myCursorType AS myRowType CURSOR@

CREATE PROCEDURE P(IN pempNo VARCHAR(8), OUT pcv1 CURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
  OPEN pcv1;

END@

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 CURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;
```

```
  SET edlevel = rv1.edlevel;
  SET lastname = rv1.name;

END @

CALL P_CALLER('000180',?,?) @
```

上記のスクリプトを実行すると、以下の出力が生成されます。

```
update command options using c off
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.

connect to sample

   Database Connection Information

 Database server        = DB2/LINUXX8664 9.7.0
 SQL authorization ID   = REGRESS5
 Local database alias   = SAMPLE


CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE TYPE myCursorType AS myRowType CURSOR@
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P(IN pempNo VARCHAR(8),OUT pcv1 CURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
  OPEN pcv1;

END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 CURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;

  SET EDLEVEL = rv1.edlevel;
  SET LASTNAME = rv1.name;

END
DB20000I  The SQL command completed successfully.

CALL P_CALLER('000180',?,?)

  Value of output parameters
  --------------------------
  Parameter Name  : EDLEVEL
  Parameter Value : 17

  Parameter Name  : LASTNAME
  Parameter Value : SCOUTTEN

  Return Status = 0
```

Boolean データ・タイプ

BOOLEAN タイプは組み込みデータ・タイプで、コンパウンド SQL (コンパイル済み) ステートメントにおけるローカル変数、グローバル変数、パラメーター、または戻りタイプとしてのみ使用できます。Boolean 値は、TRUE または FALSE の真の値を表します。Boolean 式または述部が不明な値になることもあります。その場合、NULL 値として表されます。

Boolean データ・タイプに関する制約事項

Boolean データ・タイプを使用する前に、またはその使用に関連する問題をトラブルシューティングする場合、その制約事項に注意を払うのは大切です。

Boolean データ・タイプには以下の制約事項が適用されます。

- Boolean データ・タイプは、以下としてのみ参照可能です。
  - SQL 関数で宣言されたローカル変数
  - SQL プロシージャー内で宣言されたローカル変数
  - トリガー本体としてコンパウンド SQL (コンパイル済み) ステートメントを使用する、トリガー内の宣言済みローカル変数
  - 関数本体としてコンパウンド SQL (コンパイル済み) ステートメントを使用する、SQL 関数に対するパラメーター
  - プロシージャー本体としてコンパウンド SQL (コンパイル済み) ステートメントを使用する、SQL プロシージャーに対するパラメーター
  - 戻りタイプ
  - モジュール内のグローバル変数
- Boolean データ・タイプを使用して、表またはビュー内の列のデータ・タイプを定義することはできません。
- システム定義値 TRUE と FALSE を値として参照し、表に挿入することはできません。
- Boolean データ・タイプは、外部ルーチンまたはクライアント・アプリケーションで参照することはできません。
- Boolean データ・タイプを、他のデータ・タイプにキャストすることはできません。
- Boolean データ・タイプを、SQL プロシージャーからの戻りコード値として戻すことはできません。
- Boolean データ・タイプの変数を割り当てることができるのは、TRUE、FALSE、または NULL のいずれか 1 つの値のみです。数値または他のデータ・タイプの割り当てはサポートされていません。
- 値を選択して Boolean データ・タイプの変数に入れたり、この変数に値をフェッチしたりすることはできません。
- Boolean データ・タイプは、結果セットを戻すことはできません。
- Boolean 変数は述部としては使用できません。例えば、以下の SQL 節はサポートされていません。

  ```
  IF (gb) THEN ...
  ```

述部の使用がサポートされているのは、UDF からの SET ステートメントおよび RETURN ステートメントにおいてのみです。

こうした制約事項ゆえにこのデータ・タイプを使用できない場合には、代わりに整数データ・タイプを使用し、それを値に割り当てることを考慮してください。例えば、TRUE の場合 1、FALSE の場合 0、NULL の場合 -1 などと割り当てます。

SQL ルーチン

SQL ルーチンとは、SQL プロシージャー型言語 (SQL PL) ステートメントを含む、SQL ステートメントのみを使用してロジックがインプリメントされたルーチンのことです。これには、ルーチン本体のロジックが、ルーチンの作成に使用される CREATE ステートメント内に含まれるという特徴があります。これは外部ルーチンと対照的です。外部ルーチンではそのルーチン・ロジックが、ライブラリー・ビルド形式のプログラミング・ソース・コードでインプリメントされています。一般的に、SQL ルーチンに含まれる、そしてこれが実行する SQL ステートメントの数は、外部ルーチンよりも少なめです。しかし、これをベスト・プラクティスに従ってインプリメントしたとき、全く同様の強力で高いパフォーマンスを発揮することができます。

SQL プロシージャー、SQL 関数、および SQL メソッドを作成することができます。これらすべてが SQL でインプリメントされている場合でも、それぞれのルーチンの機能タイプには異なるフィーチャーがあります。

SQL ルーチンの概要

SQL ルーチンとは、SQL プロシージャー型言語 (SQL PL) ステートメントを含む、SQL ステートメントのみを使用してロジックがインプリメントされたルーチンのことです。これには、ルーチン本体のロジックが、ルーチンの作成に使用される CREATE ステートメント内に含まれるという特徴があります。 SQL プロシージャー、SQL 関数、および SQL メソッドを作成することができます。これらすべてが SQL でインプリメントされている場合でも、それぞれのルーチンの機能タイプには異なるフィーチャーがあります。

SQL ルーチンのインプリメントを決定する前に、『ルーチンの概説』のトピックを読んで、SQL ルーチンとは何か、そのインプリメント方法、およびその使用方法を理解しておくことが大切です。その知識が身に付いたら、データベース環境内でルーチンを使用するタイミングと方法をよくわきまえた上で決定を下せるように、以下の概念トピックから SQL ルーチンの知識をさらに深めることができます。

- SQL プロシージャー
- SQL 関数
- SQL ルーチン開発のためのツール
- SQL プロシージャー型言語 (SQL PL)
- SQL PL およびインライン SQL PL の比較
- SQL PL のステートメントおよびフィーチャー
- サポートされているインライン SQL PL ステートメントおよびフィーチャー
- SQL プロシージャーまたは SQL 関数を使用する場合の判断
- SQL ルーチンに関する制約事項

SQL ルーチンについて学んだ後、次のタスクのいずれかを行うこともできます。

- SQL プロシージャーの開発
- SQL 関数の開発
- SQL メソッドの開発

SQL ルーチンの CREATE ステートメント

SQL ルーチンは、ルーチン・タイプに適した CREATE ステートメントを実行することによって作成されます。 CREATE ステートメントではルーチン本体も指定します。ルーチン本体は、SQL ルーチンの場合、SQL ステートメントまたは SQL PL ステートメントだけで記述する必要があります。 SQL プロシージャーの作成、デバッグ、実行には、 IBM® DB2 デベロップメント・センターを使用すると便利です。 SQL のプロシージャー、関数、メソッドは、 DB2 コマンド行プロセッサーで作成することも可能です。

SQL のプロシージャー、関数、メソッドには、それぞれの CREATE ステートメントがあります。ステートメントの構文はそれぞれ違いますが、共通のエレメントもいくつかあります。各ステートメントでは、ルーチン名を指定しなければなりません。パラメーターが必要であれば、パラメーターも指定しなければなりません。戻りタイプの指定も必要です。ルーチンに組み込むロジックに関する情報を DB2 に渡すための追加のキーワードも指定できます。 DB2 は、そのルーチン・プロトタイプと追加のキーワードを使用して、呼び出し時にルーチンを識別し、必要なフィーチャー・サポートと最適なパフォーマンスでルーチンを実行します。

DB2 デベロップメント・センターまたはコマンド行プロセッサーで SQL プロシージャーを作成するための具体的な情報や、関数やメソッドを作成するための具体的な情報については、以下の関連トピックを参照してください。

SQL ルーチンまたは外部ルーチンを使用する場合の判断

ルーチン・ロジックをインプリメントする場合、SQL ルーチンまたは外部ルーチンのインプリメントを選択できます。これら 2 つの各インプリメンテーションを選択するには、それぞれ理由があります。

どのような場合に SQL ルーチンまたは外部ルーチンのインプリメントを選択するかを判断するには、以下を読んで、選択を制限する要素があればそれがどのようなものかを判別してください。

- 以下の場合は、SQL ルーチンのインプリメントを選択します。
  - SQL PL および SQL ステートメントは、必要とするロジックをインプリメントするための適切なサポートを提供する。
  - ルーチン・ロジックが主にデータを照会または変更する SQL ステートメントで構成されており、パフォーマンスを重視する。データベース・データを照会または変更する SQL ステートメントの数と比較して、相対的に少量の制御フロー・ロジックを含むロジックは、一般に SQL ルーチン・インプリメンテーションの方がパフォーマンスは良くなります。 SQL PL は、データベース操作に関係したプロシージャー・ロジックのインプリメントに使用することを意図しており、プログラミング複合ロジックを主に意図しているわけではありません。

- 実行する必要がある SQL ステートメントが、外部ルーチン・インプリメンテーションで実行できる場合。
- オペレーティング・システム環境間で移植可能性が高いモジュールを作成し、プログラミング言語コード・コンパイラーおよびスクリプト・インタープリターへの依存を最小限にしたい。
- 高水準プログラミング言語を使用して、ロジックを迅速かつ簡単にインプリメントしたい。
- スクリプトまたはプログラミング言語を使用するよりも、SQL での作業に慣れている。
- データベース管理システム内のロジックを保護したい。
- リリースのアップグレードまたはオペレーティング・システムのアップグレード時の、ルーチン保守およびルーチン・パッケージ保守を最小化したい。
- ロジックをインプリメントするために必要なコードの量を最小化したい。
- メモリー管理、ポインター操作、または他の一般的なプログラミングの落とし穴となるリスクを最小化することで、インプリメントされるコードの最大限の安全性を確保したい。
- SQL PL の使用時に選択可能な特殊 SQL キャッシング・サポートを活用したい。

• 以下の場合は、外部プロシージャーのインプリメントを選択します。
  - ルーチン・ロジックが非常に複雑であり、少数の SQL ステートメントで構成されており、ルーチンのパフォーマンスを重視する。大量のストリング処理が関係する、またはデータベースにアクセスしない、複雑な数学アルゴリズムなどのロジックは、一般に外部ルーチン・インプリメンテーションの方がパフォーマンスは良くなります。
  - 実行する必要がある SQL ステートメントが、外部ルーチン・インプリメンテーションで実行できる場合。
  - ルーチン・ロジックがオペレーティング・システム呼び出しを実行する - これは外部ルーチンでしか実行できません。
  - ルーチン・ロジックがファイルからの読み取りまたはファイルへの書き込みを必要とする - これは外部ルーチンでしか実行できません。
  - サーバー・ファイル・システムに書き込む。これは注意深く実行してください。
  - データベース・サーバー上にあるアプリケーションまたはスクリプトを呼び出す。
  - SQL プロシージャーでサポートされていない特定の SQL ステートメントを発行する。
  - SQL PL 以外のプログラミング言語でのプログラミングに慣れている。

デフォルトとして、SQL ルーチンが必要を満たす場合は、それを使用してください。一般に、複合ロジックのインプリメントまたはデータベース・サーバー上のファイルまたはスクリプトへのアクセスが必要であれば、外部ルーチンの使用を決定することになります。特に、SQL PL の学習およびインプリメントは短時間で済み、簡単であることも理由になります。

SQL プロシージャーまたは SQL 関数を使用する場合の判断

SQL プロシージャーまたは SQL 関数で、SQL PL によるロジックのインプリメントを選択する場合、これら 2 つの各インプリメンテーションを選択するにはそれぞれ理由があります。

以下を読んで、SQL プロシージャーまたは SQL 関数の使用をどのような場合に選択するかを判断してください。
以下の場合は SQL 関数のインプリメントを選択します。

- 機能要件が SQL 関数により満たされ、SQL プロシージャーが提供するフィーチャーを後から必要とする見込みがない場合。
- パフォーマンスが優先され、ルーチン内に含まれるロジックが照会だけで構成されるか、または単一の結果セットだけを戻す場合。

  照会、または単一の結果セットの戻りしか含まれていない場合、SQL 関数のコンパイル方法により、SQL 関数は論理的に同等の SQL プロシージャーよりパフォーマンスがよくなります。

  SQL プロシージャーでは、SQL プロシージャーの作成時に各照会がパッケージ内の照会アクセス・プランの選択肢になるように、SELECT ステートメントおよび全選択ステートメントの形式の静的照会は個別にコンパイルされます。 SQL プロシージャーが再作成されるか、またはパッケージがデータベースに再バインドされるまで、このパッケージの再コンパイルはありません。これはつまり、照会のパフォーマンスが、SQL プロシージャー実行時より前の時点でデータベース・マネージャーが入手できる情報に基づいて決定されるので、最適化されていない可能性があるということを意味します。さらに、SQL プロシージャーでは、データを照会または変更するプロシージャー・フロー・ステートメントの実行と SQL ステートメントの実行との間でデータベース・マネージャーが転送を行う場合、小規模なオーバーヘッドが伴います。

  ただし、SQL 関数はそれらを参照する SQL ステートメント内で展開およびコンパイルされます。つまりこれは、ステートメントに応じて動的に実行される SQL ステートメントのコンパイルごとに、それらがコンパイルされることを意味します。 SQL 関数はパッケージとは直接関連付けられないので、データを照会または変更するプロシージャー・フロー・ステートメントの実行と SQL ステートメントの実行との間でデータベース・マネージャーが転送を行う場合、小規模なオーバーヘッドはありません。

以下の場合は SQL プロシージャーのインプリメントを選択します。

- SQL プロシージャーでのみサポートされる SQL PL フィーチャーが必要である場合。これには、出力パラメーター・サポート、カーソルの使用、複数の結果セットを呼び出し元に戻す機能、フル条件処理サポート、トランザクションおよびセーブポイント制御、その他のフィーチャーが含まれます。
- SQL プロシージャーでのみ実行できる非 SQL PL ステートメントを実行する場合。
- データを変更したいが、必要とする関数のタイプでデータの変更がサポートされていない場合。

必ずしもそうとは言えない場合もありますが、多くの場合、SQL プロシージャーは同等のロジックを実行する SQL 関数として簡単に再作成することができます。これは、小さなパフォーマンスの改善であってもそれらすべてを考慮した場合に、パフォーマンスを最大化する有効な方法です。

SQL ルーチンまたは動的に準備されたコンパウンド SQL ステートメントを使用する場合の判断

SQL PL やその他 SQL ステートメントのアトミック・ブロックの実装方法を決定する時に、SQL ルーチンと動的に準備されたコンパウンド SQL ステートメントのどちらを使用するかを選択する必要が生じるかもしれません。 SQL ルーチンはコンパウンド SQL ステートメントを内部的に使用しますが、どちらを使用するかの選択はその他の要因によって異なる可能性があります。

パフォーマンス

動的に準備されたコンパウンド SQL ステートメントが機能上の要件を満たせる場合、こちらを使用する方がよいでしょう。なぜなら、動的に準備されたコンパウンド SQL ステートメントに含まれる SQL ステートメントは、1 つのブロックとしてコンパイルされて実行されるからです。また一般的に、このステートメントのほうが、論理的にこれに相当する SQL プロシージャーに対する CALL ステートメントよりもパフォーマンスは良くなります。

SQL プロシージャーの作成時に、このプロシージャーはコンパイルされてパッケージが作成されます。 SQL プロシージャーのコンパイルの時点で、データへのアクセスに最適な実行パスが、そのパッケージの中に入れられます。動的に準備されたコンパウンド SQL ステートメントは、実行時にコンパイルされます。このステートメントに関しては、データへのアクセスに最適な実行パスは、最新のデータベース情報を使用して判別されます。すなわち、そのアクセス・プランのほうが、それより以前に作成された論理的にそれに相当する SQL プロシージャーのアクセス・プランよりも優れているので、パフォーマンスも向上する可能性があることを意味します。

必須論理の複雑さ

ロジックがかなり単純であって、しかも SQL ステートメント数が比較的少数の場合、動的に準備されたコンパウンド SQL ステートメント (ATOMIC を指定) 内、または SQL 関数内でインライン SQL PL を使用することを考慮してください。SQL プロシージャーも単純なロジックを扱うことはできますが、SQL プロシージャーを使用すると、プロシージャーの作成やその呼び出しなどいくらかのオーバーヘッドが生じるので、必要がなければ使用しないのが最善です。

実行する SQL ステートメントの数

1 つか 2 つの SQL ステートメントだけを実行する場合、SQL プロシージャーを使用する利点はないかもしれません。実際、その使用によって、このステートメントの実行に必要なパフォーマンス全体が否定的な影響を受ける可能性があります。1 つか 2 つの SQL ステートメントを実行するような場合は、動的に準備されたコンパウンド SQL ステートメントにおいてインライン SQL PL を使用する方が得策です。

原子性およびトランザクションの制御

別の考慮事項として、原子性があります。コンパウンド SQL (インライン化) ステートメントは、アトミックである必要があります。コンパウンド SQL (インライン化) ステートメントでは、コミットおよびロールバックはサポートされません。トランザクション制御が必要な場合や、セーブポイントへのロールバックのサポートが必要な場合、SQL プロシージャーを使用する必要があります。

セキュリティー

考慮事項には、セキュリティーもあります。 SQL プロシージャーを実行できるのは、そのプロシージャーに対する EXECUTE 特権を付与されたユーザーのみです。これは、特定のロジックをだれが実行できるかに関して制限を設けることが必要な場合に役立ちます。動的に準備されたコンパウンド SQL ステートメントを実行できるかどうかに関しても管理できます。しかし、SQL プロシージャーの実行許可によりセキュリティー制御の追加の層が提供されます。

フィーチャー・サポート

1 つ以上の結果セットを戻す場合、SQL プロシージャーを使用する必要があります。

モジュール性、長期性、および再利用

SQL プロシージャーは、データベースに永続的に保管されるデータベース・オブジェクトであり、これに対しては、複数のアプリケーションまたはスクリプトからの一貫性のある参照が可能です。動的に準備されたコンパウンド SQL ステートメントは、データベースには保管されないので、その中のロジックをすぐに再利用することはできません。

SQL プロシージャーが必要を満たす場合は、それを使用してください。一般的に、複合ロジックの実装、または SQL プロシージャーによってサポートされるフィーチャーの使用が要件となりますが、動的に準備されたコンパウンド SQL ステートメントでは使用できません。これは、SQL プロシージャーの使用を決定する要因となります。

SQL ユーザー定義関数としての SQL プロシージャーの再作成

データベース管理システムでのパフォーマンスを最大にするには、可能であれば、単純な SQL プロシージャーを SQL 関数として書き換えると有利な場合があります。プロシージャーと関数とでは、ルーチン本体が SQL PL を含められるコンパウンド・ブロックでインプリメントされているという事実が共通しています。両方とも、同じ SQL PL ステートメントは、BEGIN および END キーワードで区切られるコンパウンド・ブロック内に組み込まれます。

SQL プロシージャーを SQL 関数に変換する場合には、いくつかの注意事項があります。

- これを実行する主なそして唯一の理由は、ロジックのみがデータを照会する場合のルーチンのパフォーマンスを向上させることです。
- スカラー関数では、関数の出力パラメーターには値を直接割り当てることができないという事実に対処するために、戻り値を保持する変数を宣言することが必要

になる場合があります。ユーザー定義のスカラー関数の出力値は、関数の RETURN ステートメントでのみ指定されます。

- SQL 関数がデータを変更する場合、MODIFIES SQL 節を使用して明示的に作成し、データを変更する SQL ステートメントを含めることができるようにします。

続く例では、論理的に同等の SQL プロシージャーおよび SQL スカラー関数を示しています。これら 2 つのルーチンは、同じ入力値が指定されていれば機能的には同じ出力値を提供します。ただしそれぞれはいくらか異なる方法でインプリメントされ、呼び出されます。

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR(20),
                           IN Pid INT,
                           OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN

  IF Vendor = 'Vendor 1'
   THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
   THEN SET price = (SELECT Price FROM V2Table
                  WHERE Pid = GetPrice.Pid);
  END IF;
END
```

このプロシージャーは 2 つの入力パラメーター値を取り、入力パラメーター値に基づいて条件付きで決定された出力パラメーター値を戻します。これは IF ステートメントを使用しています。この SQL プロシージャーは、CALL ステートメントを実行して呼び出します。例えば、CLP から以下を実行できます。

```
CALL GetPrice( 'Vendor 1',  9456, ?)
```

この SQL プロシージャーは、以下のように論理的に同等な SQL 表関数として再作成できます。

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
    RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
              WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

この関数は 2 つの入力パラメーターを取り、入力パラメーター値に基づいて条件付きで決定された単一のスカラー値を戻します。関数が戻されるまで、戻される値を保持するために、これには price というローカル変数の宣言および使用が必要です。ただし SQL プロシージャーは出力パラメーターを変数として使用できます。機能的にはこれら 2 つのルーチンは同じロジックを実行します。

当然のことですが、それぞれのルーチンの実行インターフェースは異なるものです。 SQL プロシージャーを単に CALL ステートメントで呼び出す代わりに、SQL

関数は式が許可されている SQL ステートメント内で呼び出す必要があります。たいていの場合これは問題ではなく、ルーチンによって戻されるデータを即時に操作することを意図している場合は、実際には利点があります。ここで SQL 関数を呼び出す 2 つ方法の例を示します。

以下のように VALUES ステートメントを使用して呼び出すことができます。

```
VALUES (GetPrice('Vendor 1', 9456))
```

例えば値を表から選択し、関数の結果に基づいて行をフィルター処理する場合などは、以下のように SELECT ステートメントで呼び出すこともできます。

```
SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10
```

SQL プロシージャー

SQL プロシージャーとは、プログラミング・サブルーチンに似たやり方で呼び出せる論理をカプセル化するときに使用できる SQL を使用して、完全にインプリメントされたプロシージャーのことです。データベースまたはデータベース・アプリケーション・アーキテクチャーには、多数の便利な SQL プロシージャーの用法があります。 SQL プロシージャーを使用して、速やかな変換の照会、データの更新、または基本レポートの生成、アプリケーションのパフォーマンスの向上、アプリケーションのモジュール化、およびデータベースの全体的な設計の改善、およびデータベースのセキュリティーのための簡単なスクリプトを作成することができます。

SQL プロシージャーには、強力なルーチン・オプションである根拠となる多数のフィーチャーがあります。

SQL プロシージャーのインプリメントを決定する前に、まずルーチンを習得してから、『SQL プロシージャーの概要 (Overview of SQL procedures)』のトピックを読むことによって、SQL ルーチンに関する限りにおいて SQL プロシージャーとは何か、そのインプリメント方法、およびその使用方法を理解しておくことが大切です。

SQL プロシージャーのフィーチャー

SQL プロシージャーの特徴は、多数のフィーチャーにあります。 SQL プロシージャー

- 従来の静的および動的な SQL ステートメントにまつわる制御フロー論理のインプリメンテーションをサポートする SQL プロシージャー型言語のステートメントおよびフィーチャーを使用できる。
- DB2 ファミリー・ブランド全体のデータベース製品においてサポートされている。そのような製品では、DB2 バージョン 9 でサポートされているフィーチャーが、すべてではなくても多数サポートされます。
- インプリメントしやすい。簡明で高水準の厳密に型付けされた言語が使用されるからです。
- SQL プロシージャーのほうが、それと同等の外部プロシージャーよりも信頼性が高い。
- SQL99 ANSI/ISO/IEC SQL 標準に準拠している。
- 入力、出力、および入出力パラメーターの引き渡しモードをサポートする。
- 単純であっても強力な、条件およびエラーの処理モデルをサポートする。

- 呼び出し元またはクライアント・アプリケーションに対して、複数の結果セットを戻すことができる。
- 特殊変数としての SQLSTATE および SQLCODE 値に簡単にアクセスできる。
- データベース内に置かれるので、バックアップおよびリストアが自動的に行われる。
- CALL ステートメントがサポートされていれば、どこでも呼び出すことができる。
- 他の SQL プロシージャーまたは、他の言語でインプリメントされているプロシージャーに対するネストされたプロシージャー呼び出しをサポートする。
- 再帰をサポートする。
- 実行される SQL ステートメントのセーブポイントおよびロールバックをサポートするので、トランザクションの広範囲にわたる制御が可能になる。
- トリガーから呼び出すことができる。

SQL プロシージャーは、広範囲にわたるサポートを提供し、それは、上記に一覧で示したものに限定されません。このプロシージャーは、ベスト・プラクティスどおりにインプリメントされた場合、データベース・アーキテクチャー、データベース・アプリケーション設計、およびデータベース・システムのパフォーマンスにおいて不可欠な役割を果たすことができます。

SQL プロシージャーの設計

SQL プロシージャーの設計には、ユーザーの要件、SQL プロシージャー・フィーチャー、SQL フィーチャーの使用方法の理解、および設計にとって妨げとなり得る制約事項についての知識が必要です。 SQL プロシージャーの設計についての以下のトピックでは、SQL プロシージャー・フィーチャーを最善の仕方で使用する SQL プロシージャーを設計する方法を学習できます。

- SQL プロシージャーの各部分
- クロスプラットフォーム SQL ストアード・プロシージャーの考慮事項
- SQL プロシージャーでサポートされる SQL PL ステートメントおよび言語フィーチャー
- SQL プロシージャーの OLTP 考慮事項
- SQL プロシージャーのパフォーマンス
- SQL ユーザー定義関数としての SQL プロシージャーの再作成
- DB2 エラーおよび警告の処理

SQL プロシージャーの各部分: SQL プロシージャーを理解するには、SQL プロシージャーの各部分を理解することが助けになります。以下は SQL プロシージャーのほんの一部です。

- SQL プロシージャーの構造
- SQL プロシージャーのパラメーター
- SQL プロシージャーの変数
- SQL プロシージャーでの SQLCODE および SQLSTATE
- SQL プロシージャーのアトミック・ブロックおよび変数の有効範囲
- SQL プロシージャーのカーソル

- SQL PL のロジック・エレメント
- SQL プロシージャーの条件およびエラー・ハンドラー
- SQL プロシージャー内で実行できる SQL ステートメント

SQL プロシージャーの構造: SQL プロシージャーは、いくつかの論理パーツで構成されているので、SQL プロシージャーの開発では、そのようなパーツを構造化フォーマットでインプリメントする必要があります。そのフォーマットはきわめて単純明快であり、準拠するのも簡単ですが、これは、ルーチンの設計と意味を単純化するためのものです。

SQL プロシージャーの中心は、コンパウンド・ステートメントです。コンパウンド・ステートメントには、キーワード BEGIN および END が結び付けられています。このステートメントは、ATOMIC または NOT ATOMIC のどちらでもかまいません。デフォルトでは、これは NOT ATOMIC です。

コンパウンド・ステートメントでは、SQL ステートメントを使用して、複数のオプションの SQL PL オブジェクトを宣言および参照することができます。下図は、SQL プロシージャー内のコンパウンド・ステートメントの構造化フォーマットを図示しています。

```
label:  BEGIN
  Variable declarations
  Condition declarations
  Cursor declarations
  Condition handler declarations
  Assignment, flow of control, SQL statements and other compound statements
END label
```

この図は、1 つ以上のオプションのアトミック・コンパウンド・ステートメント (つまりブロック) で SQL プロシージャーを構成することができ、さらに、1 つの SQL プロシージャー内でのそのようなブロックのネストや逐次取り入れが可能なことを示しています。このようなアトミック・ブロックのいずれでも、オプションの変数、条件、およびハンドラーの宣言に対して規定された順序があります。それらは、SQL 制御のステートメントおよびその他の SQL ステートメントを使ってインプリメントされている手続き型論理と、カーソル宣言の導入の前に出現しなければなりません。 SQL プロシージャー本体内の一連の SQL ステートメントを使用して、どこでカーソルを宣言してもかまいません。

SQL プロシージャーに含まれている多くの SQL 制御ステートメントの場合と同じように、制御フローを明確にするため、SQL プロシージャーのアトミック・ブロックにもラベルを付けることができます。そうすれば、変数の参照および制御ステートメントの参照の転送時に、簡単に正確さを極めることができます。

上にリストした各エレメントを示している SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE DEL_INV_FOR_PROD (IN prod INT, OUT err_buffer VARCHAR(128))
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN

  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE integer DEFAULT 0;
  DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
  DECLARE cur1 CURSOR WITH RETURN TO CALLER
```

```
                    FOR SELECT * FROM Inv;

   A: BEGIN ATOMIC
        DECLARE EXIT HANDLER FOR NO_TABLE
          BEGIN
             SET ERR_BUFFER='Table Inv does not exist';
          END;

        SET err_buffer = '';

        IF (prod < 200)
          DELETE FROM Inv WHERE product = prod;
        ELSE IF (prod < 400)
          UPDATE Inv SET quantity = 0 WHERE product = prod;
        ELSE
          UPDATE Inv SET quantity = NULL WHERE product = prod;
        END IF;

    B:  OPEN cur1;

 END
```

SQL プロシージャー内の NOT ATOMIC コンパウンド・ステートメント

上記の例は、NOT ATOMIC コンパウンド・ステートメントを解説するものであり、SQL プロシージャーで使用されるデフォルトのタイプです。コンパウンド・ステートメント内で未処理エラー条件が発生した場合、そのエラーより前に完了したどの作業もロールバックされませんが、コミットもされません。ステートメントのグループをロールバックできるのは、ROLLBACK または ROLLBACK TO SAVEPOINT ステートメントを使用して、作業単位が明示的にロールバックされる場合のみです。また、必要であれば、COMMIT ステートメントを使用して、正常に完了したステートメントをコミットすることもできます。

以下は、NOT ATOMIC コンパウンド・ステートメントを使用した SQL プロシージャーの例です。

```
CREATE PROCEDURE not_atomic_proc ()
LANGUAGE SQL
SPECIFIC not_atomic_proc
 nap:  BEGIN NOT ATOMIC

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R11:TAA', 1);

 SIGNAL SQLSTATE '70000';

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R22:TBB', 1);

END nap
```

SIGNAL ステートメントを実行すると、処理されないエラーがこのステートメントで明示的に発行されます。その後即時にプロシージャーから戻ります。プロシージャーから戻った後は、エラーが発生したにもかかわらず、最初の INSERT ステートメントは正常に実行を完了し、行を c1_sched 表に挿入し終わっています。プロシージャーは、行挿入操作をコミットすることもロールバックすることもありません。その操作は、SQL プロシージャー呼び出しの属する作業単位が完了するまで、その状態のままになります。

SQL プロシージャー内の ATOMIC コンパウンド・ステートメント

名前から連想されるとおり、ATOMIC コンパウンド・ステートメントは、単体の統一体と考えることができます。その中で何らかの未処理エラー条件が発生した場合、その時点までに実行されたすべてのステートメントも失敗したとみなされて、ロールバックされます。

ATOMIC コンパウンド・ステートメントを他の ATOMIC コンパウンド・ステートメント内にネストすることはできません。

SAVEPOINT ステートメント、COMMIT ステートメント、または ROLLBACK ステートメントを ATOMIC コンパウンド・ステートメント内部から使用することはできません。これらがサポートされるのは、SQL プロシージャー内の NOT ATOMIC コンパウンド・ステートメントにおいてのみです。

以下は、ATOMIC コンパウンド・ステートメントを使用した SQL プロシージャーの例です。

```
CREATE PROCEDURE atomic_proc ()
LANGUAGE SQL
SPECIFIC atomic_proc

ap:  BEGIN ATOMIC

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R33:TCC', 1);

   SIGNAL SQLSTATE '70000';

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R44:TDD', 1);

END ap
```

SIGNAL ステートメントを実行すると、処理されないエラーがこのステートメントで明示的に発行されます。その後即時にプロシージャーから戻ります。最初の INSERT ステートメントは、正常に実行が完了したにもかかわらずロールバックされるので、このプロシージャーでは表には何も行が挿入されていないという結果になります。

ラベルおよび SQL プロシージャーのコンパウンド・ステートメント

コンパウンド・ステートメントおよびループを含め、選択によってはラベルを使用して、SQL プロシージャー内の任意の実行可能ステートメントに名前を付けることができます。他のステートメント内でラベルを参照することで、コンパウンド・ステートメントまたはループ外部に実行の流れをジャンプさせることができ、さらに、コンパウンド・ステートメントまたはループの先頭にジャンプすることもできます。ラベルを参照できるのは、GOTO、ITERATE、および LEAVE ステートメント内です。

必要があれば、コンパウンド・ステートメントの終わりに、対応するラベルを付けてもかまいません。終わりのラベルを付ける場合、先頭で使用したものと同じラベルでなければなりません。

どのラベルも、SQL プロシージャー本体内の固有ラベルでなければなりません。

また、ラベルを使用して、ストアード・プロシージャー内の複数のコンパウンド・ステートメントで同名の変数が宣言された場合に、紛らわしさが生じないようにすることができます。 SQL 変数の名前を修飾するのに、ラベルを使用することができます。

SQL プロシージャーのパラメーター: SQL プロシージャーは、プロシージャー内外への SQL 値の引き渡しのためのパラメーターをサポートしています。

いくつかの状況では SQL プロシージャーのパラメーターが有用となります。例えば特定の入力または入力スカラー値のセットを条件とするロジックをインプリメントする場合や、1 つ以上の出力スカラー値を戻す必要があり、結果セットは戻さないようにする場合などです。

SQL プロシージャーを設計または作成するときは、SQL プロシージャーのパラメーターのフィーチャーや制限について理解しておくことが得策です。

- DB2 では、SQL プロシージャーでの入力パラメーター、出力パラメーター、入出力パラメーターを任意指定で大量に使用することができます。パラメーターのモードまたは用途を指定するには、 CREATE PROCEDURE ステートメントのルーチン・シグニチャーの部分で IN、OUT、INOUT のいずれかのキーワードを使用します。 IN パラメーターと OUT パラメーターは値による受け渡し、INOUT パラメーターは参照による受け渡しになります。
- プロシージャーで複数のパラメーターを指定するときは、それぞれが固有の名前にならなければなりません。
- プロシージャー内でパラメーターと同じ名前の変数を宣言する場合は、プロシージャー内でネストしたラベル付きのアトミック・ブロックの中に宣言する必要があります。そうしない場合、DB2 は、あいまいな名前の参照を検出してしまいます。
- パラメーターのデータ・タイプに関係なく、SQL プロシージャーのパラメーターを SQLSTATE または SQLCODE という名前にすることはできません。

SQL プロシージャー内のパラメーターのリファレンスに関する詳細については、CREATE PROCEDURE (SQL) ステートメントを参照してください。

以下の SQL プロシージャー (myparams) は、 IN、INOUT、OUT の各パラメーター・モードの使用法を示した例です。 myfile.db2 という CLP ファイルに SQL プロシージャーを定義し、コマンド行を使用することが前提になります。

```
CREATE PROCEDURE myparams (IN p1 INT, INOUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
  SET p2 = p1 + 1;
  SET p3 = 2 * p2;
END@
```

パラメーター・マーカー: パラメーター・マーカー (通常、疑問符 (?) またはコロンの後に変数名が続く形 (:*var1*) で示される) は、ステートメント実行中に値が取得される SQL ステートメント内のプレースホルダーです。アプリケーションはパラメーター・マーカーをアプリケーション変数に関連付けます。ステートメントの実行中、これらの変数の値によって個々のパラメーター・マーカーがそれぞれ置き換えられます。処理中にデータ変換が発生することがあります。

パラメーター・マーカーの利点

何度も実行する必要のある SQL ステートメントの場合、SQL ステートメントを一回だけ準備し、パラメーター・マーカーを使って実行時に入力値を置換することにより、照会プランを再利用する方がたいてい効率的です。DB2® 9 では、パラメーター・マーカーは次の 2 つの方法のうち 1 つの方法で示されます。

- 最初のスタイルは「?」文字ですが、これは動的 SQL 実行 (動的組み込み SQL、CLI、Perl など) で使用されます。
- 2 番目のスタイルは、組み込み SQL の標準的な構造を表し、変数の名前にコロンの接頭部が付きます (:*var1*)。このスタイルは静的 SQL 実行で使用され、普通はホスト変数と呼ばれます。

どちらかのスタイルを使うことによって、アプリケーション変数が SQL ステートメント内で置換される場所を示します。パラメーター・マーカーは番号で参照され、1 から始まり、左から右へ順番に番号が付けられます。SQL ステートメントが実行される前に、アプリケーションは変数ストレージ域を SQL ステートメントに指定された各パラメーター・マーカーにバインドしなければなりません。さらに、バインドされる変数は有効なストレージ域でなければならず、準備ステートメントがデータベースに対して実行されるときに入力データ値を含んでいなければなりません。

以下の例では、2 つのパラメーター・マーカーを含んだ SQL ステートメントを示しています。

```
SELECT * FROM customers WHERE custid = ? AND lastname = ?
```

サポートされるタイプ

DB2 では型なしパラメーター・マーカーがサポートされ、SQL ステートメント内の特定の場所でこれを使用することができます。表 1 ではパラメーター・マーカー使用時の制約事項がリストされています。

表 2. パラメーター・マーカーの使用に関する制約事項

型なしパラメーター・マーカーの場所	データ・タイプ
式: 選択リスト内で単独	エラー
式: 算術演算子の両オペランド	エラー
述部: IN 述部の左側のオペランド	エラー
述部: 関係演算子の両オペランド	エラー
関数: 集約関数のオペランド	エラー

例

DB2® 9 は、効率的にデータにアクセスできるように、CLI/ODBC、JDBC、および ADO.NET を含め豊富な標準インターフェースのセットを提供しています。次のコード・スニペットは、各データ・アクセス API に対するパラメーター・マーカーの付いた準備ステートメントの使用法を示しています。

以下の表 t1 の表スキーマについて考慮してください。ここで、列 c1 が表 t1 の主キーです。

表 3. 表スキーマの例

列名	DB2 データ・タイプ	NULL 可能
c1	INTEGER	false
c2	SMALLINT	true
c3	CHAR(20)	true
c4	VARCHAR(20)	true
c5	DECIMAL(8,2)	true
c6	DATE	true
c7	TIME	true
c8	TIMESTAMP	true
c9	BLOB(30)	true

以下の例では、準備ステートメントを使って行を表 t1 に挿入する方法を示しています。

CLI の例

```
void parameterExample1(void)
{
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;
   SQLRETURN rc;
   TCHAR server[] = _T("C:¥¥mysample¥¥");
   TCHAR uid[] = _T("db2e");
   TCHAR pwd[] = _T("db2e");
   long p1 = 10;
   short p2 = 100;
   TCHAR p3[100];
   TCHAR p4[100];
   TCHAR p5[100];
   TCHAR p6[100];
   TCHAR p7[100];
   TCHAR p8[100];
   char  p9[100];
   long len = 0;

   _tcscpy(p3, _T("data1"));
   _tcscpy(p4, _T("data2"));
   _tcscpy(p5, _T("10.12"));
   _tcscpy(p6, _T("2003-06-30"));
   _tcscpy(p7, _T("12:12:12"));
   _tcscpy(p8, _T("2003-06-30-17.54.27.710000"));

   memset(p9, 0, sizeof(p9));
   p9[0] = 'X';
   p9[1] = 'Y';
   p9[2] = 'Z';

   rc = SQLAllocEnv(&henv);
   // check return code ...

   rc = SQLAllocConnect(henv, &hdbc);
   // check return code ...

   rc = SQLConnect(hdbc, (SQLTCHAR*)server, SQL_NTS,
    (SQLTCHAR*)uid, SQL_NTS, (SQLTCHAR*)pwd, SQL_NTS);
   // check return code ...
```

```
rc = SQLAllocStmt(hdbc, &hstmt);
// check return code ...

// prepare the statement
rc = SQLPrepare(hstmt, _T("INSERT INTO t1 VALUES (?,?,?,?,?,?,?,?,?)"), SQL_NTS);
// check return code ...

// bind input parameters
rc = SQLBindParameter(hstmt, (unsigned short)1, SQL_PARAM_INPUT,
 SQL_C_LONG, SQL_INTEGER, 4, 0, &p1, sizeof(p1), &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)2, SQL_PARAM_INPUT, SQL_C_LONG,
  SQL_SMALLINT, 2, 0, &p2, sizeof(p2), &len);
// check return code ...

len = SQL_NTS;
rc = SQLBindParameter(hstmt, (unsigned short)3, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_CHAR, 0, 0, &p3[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)4, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_VARCHAR, 0, 0, &p4[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)5, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_DECIMAL, 8, 2, &p5[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)6, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_DATE, 0, 0, &p6[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)7, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIME, 0, 0, &p7[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)8, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_TYPE_TIMESTAMP, 0, 0, &p8[0], 100, &len);
// check return code ...

len = 3;
rc = SQLBindParameter(hstmt, (unsigned short)9, SQL_PARAM_INPUT, SQL_C_BINARY,
  SQL_BINARY, 0, 0, &p9[0], 100, &len);
// check return code ...

// execute the prepared statement
rc = SQLExecute(hstmt);
// check return code ...

rc = SQLFreeStmt(hstmt, SQL_DROP);
// check return code ...

rc = SQLDisconnect(hdbc);
// check return code ...

rc = SQLFreeConnect(hdbc);
// check return code ...

rc = SQLFreeEnv(henv);
// check return code ...
```

C の例

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
```

```
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;

/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

JDBC の例

```
public static void parameterExample1() {

   String driver = "com.ibm.db2e.jdbc.DB2eDriver";
   String url    = "jdbc:db2e:mysample";
   Connection conn = null;
   PreparedStatement pstmt = null;

   try
   {
      Class.forName(driver);

      conn = DriverManager.getConnection(url);

      // prepare the statement
      pstmt = conn.prepareStatement("INSERT INTO t1 VALUES
                                                  (?, ?, ?, ?, ?, ?, ?, ?, ?)");

      // bind the input parameters
      pstmt.setInt(1, 1);
      pstmt.setShort(2, (short)2);
      pstmt.setString(3, "data1");
      pstmt.setString(4, "data2");
      pstmt.setBigDecimal(5, new java.math.BigDecimal("12.34"));
      pstmt.setDate(6, new java.sql.Date(System.currentTimeMillis() ) );
      pstmt.setTime(7, new java.sql.Time(System.currentTimeMillis() ) );
      pstmt.setTimestamp (8, new java.sql.Timestamp(System.currentTimeMillis() ) );
      pstmt.setBytes(9, new byte[] { (byte)'X', (byte)'Y', (byte)'Z' } );

      // execute the statement
      pstmt.execute();

      pstmt.close();

      conn.close();
   }
   catch (SQLException sqlEx)
   {
      while(sqlEx != null)
      {
          System.out.println("SQLERROR: ¥n" + sqlEx.getErrorCode() +
             ", SQLState: " + sqlEx.getSQLState() +
             ", Message: " + sqlEx.getMessage() +
             ", Vendor: " + sqlEx.getErrorCode() );
         sqlEx = sqlEx.getNextException();
      }
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
   }
}
```

ADO.NET の例 [C#]

```
public static void ParameterExample1()
{
   DB2eConnection conn = null;
   DB2eCommand cmd  = null;
   String connString   = @"database=.¥; uid=db2e; pwd=db2e";
   int i = 1;

   try
   {
      conn = new DB2eConnection(connString);

      conn.Open();

      cmd = new DB2eCommand("INSERT INTO t1 VALUES
                                     (?, ?, ?, ?, ?, ?, ?, ?, ?)", conn);

      // prepare the command
      cmd.Prepare();

      // bind the input parameters
      DB2eParameter p1 = new DB2eParameter("@p1", DB2eType.Integer);
      p1.Value = ++i;
      cmd.Parameters.Add(p1);

      DB2eParameter p2 = new DB2eParameter("@p2", DB2eType.SmallInt);
      p2.Value = 100;
      cmd.Parameters.Add(p2);

      DB2eParameter p3 = new DB2eParameter("@p3", DB2eType.Char);
      p3.Value = "data1";
      cmd.Parameters.Add(p3);

      DB2eParameter p4 = new DB2eParameter("@p4", DB2eType.VarChar);
      p4.Value = "data2";
      cmd.Parameters.Add(p4);

      DB2eParameter p5 = new DB2eParameter("@p5", DB2eType.Decimal);
      p5.Value = 20.25;
      cmd.Parameters.Add(p5);

      DB2eParameter p6 = new DB2eParameter("@p6", DB2eType.Date);
      p6.Value = DateTime.Now;
      cmd.Parameters.Add(p6);

      DB2eParameter p7 = new DB2eParameter("@p7", DB2eType.Time);
      p7.Value = new TimeSpan(23, 23, 23);
      cmd.Parameters.Add(p7);

      DB2eParameter p8 = new DB2eParameter("@p8", DB2eType.Timestamp);
      p8.Value = DateTime.Now;
      cmd.Parameters.Add(p8);

      byte []barr = new byte[3];
      barr[0] = (byte)'X';
      barr[1] = (byte)'Y';
      barr[2] = (byte)'Z';

      DB2eParameter p9 = new DB2eParameter("@p9", DB2eType.Blob);
      p9.Value = barr;
      cmd.Parameters.Add(p9);

      // execute the prepared command
      cmd.ExecuteNonQuery();
   }
   catch (DB2eException e1)
```

```
   {
      for (int i=0; i < e1.Errors.Count; i++)
      {
         Console.WriteLine("Error #" + i + "¥n" +
            "Message: " + e1.Errors[i].Message + "¥n" +
            "Native: " + e1.Errors[i].NativeError.ToString() + "¥n" +
            "SQL: " + e1.Errors[i].SQLState + "¥n");
      }
   }
   catch (Exception e2)
   {
       Console.WriteLine(e2.Message);
   }
   finally
   {
      if (conn != null && conn.State != ConnectionState.Closed)
      {
            conn.Close();
            conn = null;
      }
   }
}
```

SQL プロシージャーの変数 (DECLARE、SET ステートメント): SQL プロシージャーのローカル変数サポートにより、SQL プロシージャー・ロジックをサポートする SQL 値の割り当てと検索が可能です。

SQL プロシージャーの変数は DECLARE ステートメントを使って定義します。

値は SET ステートメントまたは SELECT INTO ステートメントを使って変数に割り当てることもできますし、変数の宣言時にデフォルト値として割り当てることもできます。変数には、リテラル、式、照会の結果、および特殊レジスターの値を割り当てることができます。

変数値を SQL プロシージャーのパラメーターに割り当てることができ (SQL プロシージャー内の他の変数)、ルーチン内で実行した SQL ステートメント内のパラメーターとして参照することができます。

以下の例は、変数値のさまざまな割り当ておよび検索方法を示しています。

```
CREATE PROCEDURE proc_vars()
SPECIFIC proc_vars
LANGUAGE SQL
BEGIN

  DECLARE v_rcount INTEGER;

  DECLARE v_max DECIMAL (9,2);

  DECLARE v_adate, v_another  DATE;

  DECLARE v_total INTEGER DEFAULT 0;           -- (1)

  DECLARE v_rowsChanged BOOLEAN DEFAULT FALSE; -- (2)

  SET v_total = v_total + 1                    -- (3)

  SELECT MAX(salary)                           -- (4)
    INTO v_max FROM employee;

  VALUES CURRENT_DATE INTO v_date;             -- (5)

  SELECT CURRENT DATE, CURRENT DATE            -- (6)
```

```
      INTO v_adate, v_another
  FROM SYSIBM.SYSDUMMY1;

  DELETE FROM T;
  GET DIAGNOSTICS v_rcount = ROW_COUNT;        -- (7)

  IF v_rcount > 0 THEN                         -- (8)
    SET is_done = TRUE;
  END IF;
END
```

行 (1) にあるように、変数を宣言するときに DEFAULT 節を使ってデフォルト値を指定することができます。行 (2) は、Boolean データ・タイプの変数 (デフォルト値 FALSE) の宣言を示しています。行 (3) は、SET ステートメントを使って単一の変数値を割り当てることができることを示しています。行 (4) が示すように、SELECT または FETCH ステートメントを INTO 節と組み合わせて実行することによって変数を設定することもできます。行 (5) および (6) は、VALUES INTO ステートメントを使って関数または特殊レジスターを評価して、値を変数または複数の変数に割り当てることができることを示しています。

GET DIAGNOSTICS ステートメントの結果を変数に割り当てることもできます。GET DIAGNOSTICS は、影響を受ける行 (UPDATE ステートメントの場合は更新される行、DELETE ステートメントの場合は削除される行) の数のハンドルを取得するため、または実行されたばかりの SQL ステートメントの戻り状況を取得するために使用できます。行 (7) は、直前で実行された DELETE ステートメントによって変更される行数を変数に割り当てる方法を示しています。

行 (8) は、1 つの論理を使用して、変数に割り当てられる値を判別する方法を示しています。この事例では、前の DELETE ステートメントの一部として行数が変更され、GET DIAGNOSTICS ステートメントの実行により変数 v_rcount にゼロより大きい値が割り当てられた場合、変数 is_done に値 TRUE が割り当てられます。

SQL プロシージャーでの SQLCODE および SQLSTATE 変数: エラー処理を実行したり、SQL プロシージャーのデバッグを補助したりするには、SQLCODE または SQLSTATE の値をテストしたり、これらの値を出力パラメーターまたは診断メッセージ・ストリングの一部として戻したり、これらの値を表に挿入したりして基本的なトレース・サポートを提供すると便利な場合があります。

SQL プロシージャー中の SQLCODE および SQLSTATE 値を使用するには、SQL プロシージャー本体で以下のような SQL 変数を宣言する必要があります。

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

ステートメントが実行されると DB2 は常にこれらの変数を暗黙で設定します。ハンドラーを用意された条件がステートメントによって生じた場合、ハンドラーの実行の開始時点で SQLSTATE および SQLCODE 変数の値を利用することができます。ただしこの変数は、ハンドラー内の最初のステートメントが実行されるとただちにリセットされます。そのため、ハンドラーの最初のステートメント内のローカル変数に SQLSTATE および SQLCODE の値をコピーしておくのが一般的な措置です。以下の例では、すべての条件で CONTINUE ハンドラーが使用されて、retcode という別の変数に SQLCODE 変数がコピーされます。次に実行可能ステートメント内で変数 retcode を使用して、プロシージャー・ロジックを制御したり、出力パラメーターとして値を返したりすることができます。

```
BEGIN
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE retcode INTEGER DEFAULT 0;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
     SET retcode = SQLCODE;

  executable-statements
END
```

注: SQL プロシージャーで SQLCODE または SQLSTATE 変数をアクセスする場合には、DB2 によって、後続するステートメントの SQLCODE 値は 0、また SQLSTATE 値は'00000' に設定されます。

SQL プロシージャー内のコンパウンド・ステートメントと変数の有効範囲: SQL プロシージャーには 1 つ以上のコンパウンド・ステートメントを含めることができます。それらは順次導入したり、別のコンパウンド・ステートメントの中にネストしたりできます。各コンパウンド・ステートメントによってそれぞれ新しいスコープが導入され、その中で変数が使用できたりできなかったりします。

コンパウンド・ステートメント内で宣言される変数を修飾し、固有に識別するためにラベルを使用できます。このため、コンパウンド・ステートメントを識別するラベルを使用するのは大切なことです。別のコンパウンド・ステートメントまたはネストされたコンパウンド・ステートメントの変数を参照する場合にはこれは特に重要になります。

以下の例には、変数 *a* の 2 つの宣言があります。 1 つのインスタンスは、 *lab1* というラベルの付いた外側のコンパウンド・ステートメント内の宣言であり、もう 1 つのインスタンスは、 *lab2* というラベルの付いた内側のコンパウンド・ステートメント内の宣言です。この記述のとおり、代入ステートメント内の *a* の参照先は、*lab2* というラベルの付いたコンパウンド・ブロックのローカル有効範囲内の宣言であると DB2 は判断します。しかし、変数 *a* によって、*lab1* というラベルの付いたコンパウンド・ステートメント・ブロック内の宣言を参照する場合は、そのコンパウンド・ブロック内の宣言を正しく参照するために、変数をそのブロックのラベルで修飾する必要があります。つまり、*lab1.a* というふうに修飾するということです。

```
CREATE PROCEDURE P1 ()
LANGUAGE SQL
  lab1: BEGIN
    DECLARE a INT DEFAULT 100;
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

      UPDATE T1
       SET T1.b = 5
        WHERE T1.b = a;  <-- Variable a refers to lab2.a
                             unless qualified otherwise

    END lab2;
 END lab1
```

SQL プロシージャー内の最も外側のコンパウンド・ステートメントをアトミックとして宣言できます。そのためには、BEGIN キーワードの後に ATOMIC キーワード

を追加します。アトミック・コンパウンド・ステートメント内のステートメントの実行時にエラーが発生した場合は、そのコンパウンド・ステートメント全体がロールバックされます。

SQL プロシージャーのカーソル: SQL プロシージャーでは、結果セット (データ行のセット) の定義と、1 行ずつのベースでの複雑な論理の実行は、カーソルによって可能になります。また、SQL プロシージャーは、この同じメカニックを使用して、結果セットを定義し、SQL プロシージャーの呼び出し元またはクライアント・アプリケーションに結果セットを直接戻すこともできます。

カーソルを、一連の行の中の 1 つの行を指すポインターとみなすことができます。カーソルは、一度に 1 つの行のみを参照できる一方で、必要に応じて結果セットの他の行に移動することもできます。

SQL プロシージャーでカーソルを使用するには、次のようにする必要があります。

1. 結果セットを定義するカーソルを宣言します。
2. カーソルをオープンし、結果セットを確立します。
3. 必要に応じて、一度に 1 行ずつ、カーソルからローカル変数にデータを取り出します。
4. 完了したなら、カーソルをクローズします。

カーソルを処理するには、次のような SQL ステートメントを使用する必要があります。

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

以下の例は、SQL プロシージャー内での読み取り専用カーソルの基本的な使用法を示しています。

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE p_sum INTEGER;
  DECLARE p_sal INTEGER;
  DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

   SET p_sum = 0;

   OPEN c;

   FETCH FROM c INTO p_sal;

   WHILE(SQLSTATE = '00000') DO
      SET p_sum = p_sum + p_sal;
      FETCH FROM c INTO p_sal;
   END WHILE;

   CLOSE c;

   SET sum = p_sum;

END%
```

SQL プロシージャー内でのカーソルのもっと複雑な使用例を以下に示します。この例は、カーソルおよび SQL PL ステートメントを結合して使用する方法を示しています。

SQL プロシージャー本体の SQL PL ロジック・エレメント: 順次実行は、プログラム実行が取ることのできる最も基本的なパスです。この方法を使用すると、プログラムはコードの最初の行から実行を開始し、その後次の行に移動し、コードの最後のステートメントが実行されるまで実行を継続します。このアプローチは非常に単純なタスクでは有効ですが、1 つの状況しか処理できないため、どちらかというと有用性に欠けます。多くの場合、プログラムには状況の変化に応じた対応が求められます。コードの実行パスを制御することにより、コードの特定の部分を使って複数の状況を賢明に処理できるようにすることが可能です。

SQL PL は、変数に対するサポートを提供し、ステートメントの実行順序の制御に使用できる制御ステートメントのフローを提供します。 IF や CASE などのステートメントは SQL PL ステートメントのブロックを条件的に実行するために使用され、WHILE や REPEAT などの他のステートメントは通常、タスクが完了するまでステートメントのセットを反復して実行するために使用されます。

SQL PL ステートメントのタイプは多数ありますが、それをソートできるカテゴリーはわずかです。

- 変数に関連したステートメント
- 条件ステートメント
- ループ・ステートメント
- 制御転送ステートメント

SQL プロシージャー内の変数に関連したステートメント: 変数に関連した SQL ステートメントは、変数の宣言と、変数に対する値の割り当てに使用されます。変数に関連したステートメントには、以下のようないくつかのタイプがあります。

- SQL プロシージャー内の DECLARE <variable> ステートメント
- SQL プロシージャー内の DECLARE <condition> ステートメント
- SQL プロシージャー内の DECLARE <condition handler> ステートメント
- SQL プロシージャー内の DECLARE CURSOR
- SQL プロシージャー内の SET (代入ステートメント)

上記のようなステートメントは、変数値を利用することになる他のタイプの SQL PL ステートメントおよび SQL ステートメントを利用するのに必要なサポートを備えています。

SQL プロシージャー内の条件ステートメント: 条件ステートメントを使用するのは、満たさなければならない特定の条件の状況に基づいて、どの論理を実行するかを定義する場合です。 SQL プロシージャーでサポートされている条件ステートメントには、以下の 2 つのタイプがあります。

- CASE
- IF

これらのステートメントは互いに似通っています。ただし、CASE ステートメントは、IF ステートメントを拡張したステートメントです。

SQL プロシージャーの CASE ステートメント: CASE ステートメントを使用して、満たさなければならない条件の状況に基づいて、条件付きで特定の論理を開始することができます。 CASE ステートメントには、次の 2 つのタイプがあります。

- 単純 CASE ステートメント (Simple CASE statement): リテラル値に基づいて特定の論理を開始するのに使用します。
- 検索 CASE ステートメント (Searched CASE statement): 式の値に基づいて特定の論理を開始するのに使用します。

CASE ステートメントの WHEN 節は、制御のフローを決定するために満たしていなければならない値を定義します。

simple-case-statement-when-clause を使用した CASE ステートメントを指定された SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE  v_workdept
    WHEN 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
    END CASE

END
```

searched-case-statement-when-clause を使用した CASE ステートメントの例を以下に示します。

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE
    WHEN v_workdept = 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN v_workdept = 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
  END CASE

END
```

上記のどちらの例も論理的には同等ですが、searched-case-statement-when-clause を使用する CASE ステートメントは非常に強力になりうることに注意することが重要です。ここでは、サポートされている任意の SQL 式を使用することができます。この式では、変数、パラメーター、特殊レジスター、その他に対する参照を使用することができます。

SQL プロシージャーの IF ステートメント: IF ステートメントを使用して、満たさなければならない条件の状況に基づいて、条件付きで特定の論理を開始することができます。 IF ステートメントは、searched-case-statement-when 節を指定された CASE ステートメントと論理的に同等です。

IF ステートメントは、オプションの ELSE IF 節およびデフォルトの ELSE 節の使用をサポートします。ステートメントの終わりを指示するための END IF 節が必要です。

IF ステートメントを使用しているプロシージャーの例を以下に示します。

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
                             INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
    UPDATE employee
    SET salary = salary * 1.10, bonus = 1000
      WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
    SET salary = salary * 1.05, bonus = 500
      WHERE empno = empNum;
  ELSE
    UPDATE employee
    SET salary = salary * 1.03, bonus = 0
      WHERE empno = empNum;
  END IF;
END
```

SQL プロシージャーのループ・ステートメント: ループ・ステートメントでは、条件が満たされるまで特定の論理を繰り返し実行することがサポートされます。 SQL PL では、以下のループ・ステートメントがサポートされます。

- FOR
- LOOP
- REPEAT
- WHILE

FOR ステートメントは他のステートメントとは異なります。つまり、FOR ステートメントは、定義済みの結果セットの行に対する反復操作を行うのに使用されるのに対して、その他のステートメントは、各条件が満たされるまで、一連の SQL ステートメントに対する反復操作を行うために使用されます。

どのループ制御ステートメントの場合も、識別のためのラベルを定義することができます。

SQL プロシージャーの FOR ステートメント: FOR ステートメントは、特殊なタイプのループ・ステートメントです。なぜなら、定義済みの読み取り専用の結果セット内の行に対する反復操作を行うのに使用されるからです。 FOR ステートメントを実行すると、FOR ループの反復ごとに、次の行が取り出された場合はその行が結果セットになるように、カーソルが暗黙で宣言されます。結果セット内に行がなくなるまで、ループは続行されます。

FOR ステートメントによって、カーソルのインプリメンテーションが単純化され、論理操作の実行対象になりうる一連の行の一連の列値を簡単に検索できるようになります。

単純な FOR ステートメントのみを使用した SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE P()
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE fullname CHAR(40);

  FOR v AS cur1 CURSOR FOR
             SELECT firstnme, midinit, lastname FROM employee
  DO
    SET fullname = v.lastname || ',' || v.firstnme
                   ||' ' || v.midinit;
    INSERT INTO tnames VALUES (fullname);
  END FOR;
END
```

注: CONCAT 関数を使用すれば、上記の例に示されているような論理を簡単にインプリメントすることができます。構文を分かりやすく説明するために、例は単純化されています。

for-loop-name は、FOR ステートメントをインプリメントするために生成される暗黙のコンパウンド・ステートメントのラベルを指定します。その場合、コンパウンド・ステートメントのラベルの規則が守られます。 for-loop-name を使用して、SELECT ステートメントから戻された結果セット内の列名を修飾することができます。

cursor-name は、結果セットから行を選択するときに使用されるカーソルに名前を付けるだけです。これを指定しないと、DB2 データベース・マネージャーによって自動的に固有なカーソル名が内部で生成されます。

SELECT ステートメントの列名は固有名でなければならず、1 つの表 (JOIN または UNION を行う場合は複数の表) を指定する FROM 節がなければなりません。 参照される表および列は、ループの実行より前に存在している必要があります。グローバル一時表および宣言済み一時表を参照することができます。

位置指定の更新と削除、および検索済み更新と削除は、FOR ループ中でサポートされます。確実に正しい結果を得るには、FOR ループのカーソル指定に FOR UPDATE 節を含めてください。

FOR ステートメントのサポートのために作成されたカーソルを、FOR ループ外で参照することはできません。

SQL プロシージャーの LOOP ステートメント: LOOP ステートメントは、特殊なタイプのループ・ステートメントです。なぜなら、終了条件の節がないからです。これは、別の論理 (一般的には制御ステートメントの転送) によって、制御フローをループ外のいずれかの地点に強制的にジャンプさせるまで繰り返し実行される一連のステートメントを定義します。

LOOP ステートメントは一般的に、LEAVE、GOTO、ITERATE、または RETURN のうちのいずれかのステートメントと一緒に使用されます。これらのステートメン

トは、ループの直後の地点、SQL プロシージャー内の指定のロケーション、ループの別の反復を開始するためのループの開始地点のいずれかに、強制的に制御を戻すことができますが、SQL プロシージャーを終了することもできます。これらのステートメントの使用時にどこへフローを渡すかを指示するときは、ラベルを使用します。

LOOP ステートメントが便利なのは、ループ内に複雑な論理が配置されていて、複数の終了方法を必要とする場合ですが、無限ループ・インスタンスが起きないように慎重に使用する必要があります。

制御を移すステートメントのない LOOP ステートメントを単独で使用すると、そのループ内に含まれている一連のステートメントは無限に実行されることになります。あるいは、制御フローを強制的に変更する条件ハンドラーが起動されることになるようなデータベース条件が発生するまで、または処理されない条件が発生したために SQL プロシージャーから強制的に戻ることになるまで、その一連のステートメントの実行が続けられます。

LOOP ステートメントを使用した SQL プロシージャーの例を以下に示します。そこでは、ITERATE および LEAVE ステートメントも使用されています。

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP

    FETCH c1 INTO v_deptno, v_deptname;

    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END  IF;

    INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;
END
```

SQL プロシージャーの WHILE ステートメント: WHILE ステートメントは、WHILE ループの先頭で条件が偽と評価されるまで実行される一連のステートメントを定義します。 while-loop-condition (式) は、ループの反復処理ごとに、事前に評価されます。

単純な WHILE ループを使用した SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE sum_mn (IN p_start INT
                        ,IN p_end INT
                        ,OUT p_sum INT)
```

```
SPECIFIC sum_mn
LANGUAGE SQL
smn: BEGIN

DECLARE v_temp INTEGER DEFAULT 0;
DECLARE v_current INTEGER;

SET v_current = p_start;

WHILE (v_current <= p_end) DO
  SET v_temp = v_temp + v_current;
  SET v_current = v_current + 1;
END WHILE;
p_sum = v_current;
END smn;
```

注: 数式を使用すれば、上記の例に示されているような論理を簡単にインプリメントすることができます。構文を分かりやすく説明するために、例は単純化されています。

SQL プロシージャーの REPEAT ステートメント: REPEAT ステートメントは、REPEAT ループの末尾で条件が真と評価されるまで実行される一連のステートメントを定義します。ループの反復の完了ごとに、repeat-loop-condition が評価されます。

WHILE ステートメントでは、最初の受け渡しで while-loop-condition が偽の場合はループには入りません。 REPEAT ステートメントは、それに代わる便利なステートメントです。ただし、while-loop 論理を REPEAT ステートメントに書き直せることに注意する必要があります。

以下に、SQL プロシージャーで使用される REPEAT ステートメントの例を示してあります。

```
CREATE PROCEDURE sum_mn2 (IN p_start INT
                         ,IN p_end INT
                         ,OUT p_sum INT)
SPECIFIC sum_mn2
LANGUAGE SQL
smn2: BEGIN

  DECLARE v_temp INTEGER DEFAULT 0;
  DECLARE v_current INTEGER;

  SET v_current = p_start;

  REPEAT
    SET v_temp = v_temp + v_current;
    SET v_current = v_current + 1;
  UNTIL (v_current > p_end)
  END REPEAT;
END
```

SQL プロシージャーでの制御ステートメントの転送: 制御ステートメントの転送を使用するのは、SQL プロシージャー内で制御のフローをリダイレクトするときです。この無条件ブランチを使用して、制御のフローをある地点から別の地点にジャンプさせることができます。ジャンプするのは、制御ステートメントの転送より前または後のどちらでもかまいません。 SQL プロシージャー内でサポートされている制御ステートメントの転送は、次のとおりです。

- GOTO

- ITERATE
- LEAVE
- RETURN

SQL プロシージャー内のどこでも制御ステートメントの転送を使用できますが、ITERATE および LEAVE は一般的に、LOOP ステートメントまたは他のループ・ステートメントと一緒に使用します。

SQL プロシージャーの GOTO ステートメント: GOTO ステートメントは、単純明快でしかも基本的な制御フローのステートメントであり、制御のフロー内での無条件の変更の原因となります。これは、SQL プロシージャーに定義されているラベルを使用して特定のユーザー定義ロケーションへのブランチを設定するのに使用します。

GOTO ステートメントの使用は、一般にプログラミングの手法としては粗悪なものと考えられており、推奨されていません。 GOTO を広範囲にわたって使用すると、判読不能なコードにつながる可能性が高くなります。プロシージャーが長大になるときは特にそうです。さらに、実行パスを制御するために利用できるより優れたステートメントがあるので、GOTO は必要ありません。 GOTO を使用することが必要になるような状況は特にありません。つまり、便宜上使用される場合のほうが多いということです。

GOTO ステートメントを使用した SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE adjust_salary ( IN p_empno CHAR(6),
                      IN p_rating INTEGER,
     OUT p_adjusted_salary DECIMAL (8,2) )
LANGUAGE SQL
BEGIN
  DECLARE new_salary DECIMAL (9,2);
  DECLARE service DATE;  -- start date

  SELECT salary, hiredate INTO v_new_salary, v_service
    FROM employee
       WHERE empno = p_empno;

  IF service > (CURRENT DATE - 1 year) THEN
    GOTO exit;
  END IF;

  IF p_rating = 1 THEN
    SET new_salary = new_salary + (new_salary * .10);
  END IF;

  UPDATE employee SET salary = new_salary WHERE empno = p_empno;

exit:
  SET p_adjusted_salary = v_new_salary;


END
```

この例は、GOTO ステートメントの効果的な使用方法を示しています。特定の論理を実行しない一方で、他の論理はこれまでどおり実行されるようにするために、プロシージャーまたはループのほぼ終了地点へスキップします。

GOTO ステートメントの使用時には、有効範囲に関する次のようないくつかの追加の考慮事項があることに注意する必要があります。

- GOTO ステートメントを FOR ステートメント内で定義する場合、ネストされている FOR ステートメントまたはネストされているコンパウンド・ステートメント内にあるのでない限り、同じ FOR ステートメントの内側でラベルを定義しなければなりません。
- GOTO ステートメントをコンパウンド・ステートメント内で定義する場合、ネストされている FOR ステートメントまたはネストされているコンパウンド・ステートメント内にあるのでない限り、同じコンパウンド・ステートメント内部でラベルを定義しなければなりません。
- GOTO ステートメントをハンドラーで定義する場合、他の有効範囲の規則に従って、同じハンドラーでラベルを定義しなければなりません。
- GOTO ステートメントをハンドラー外で定義する場合、ハンドラーの内部でラベルを定義してはなりません。
- GOTO ステートメントが到達できる有効範囲内でラベルが定義されていない場合、エラーが戻されます (SQLSTATE 42736)。

SQL プロシージャーの ITERATE ステートメント: ITERATE ステートメントを使用すると、ラベル付きの LOOP ステートメントの先頭に制御のフローが戻ります。

ITERATE ステートメントを使用した SQL プロシージャーの例を以下に示します。

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP
   FETCH c1 INTO v_deptno, v_deptname;
      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END  IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;

END
```

この例では、取り出された行内の列値が特定の値に一致したときに、ラベル ins_loop を指定して定義されている LOOP ステートメントに制御のフローを戻すために ITERATE ステートメントが使用されています。 ITERATE ステートメントの位置によって、department 表にはまったく値が挿入されなくなります。

SQL プロシージャーの LEAVE ステートメント: LEAVE ステートメントは、ループまたはコンパウンド・ステートメント外に制御のフローを移動するのに使用します。

LEAVE ステートメントを使用した SQL プロシージャーの例を以下に示します。

```
  CREATE  PROCEDURE ITERATOR()
  LANGUAGE SQL
  BEGIN
    DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';

    DECLARE c1 CURSOR FOR SELECT deptno, deptname
                          FROM department ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;

    OPEN c1;

    ins_loop: LOOP

      FETCH c1 INTO v_deptno, v_deptname;

      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;
END
```

この例では、LEAVE ステートメントを使用して、ラベル ins_loop を使って定義された LOOP ステートメントを終了します。このステートメントは、IF ステートメントにネストされているので、IF 条件が真になったときという条件付きで実行されます。この条件が真になるのは、カーソル内にもう行がなくなったときです。LEAVE ステートメントの位置によって、NOT FOUND エラーが発生した後はループの反復はもう実行されなくなります。

SQL プロシージャーの RETURN ステートメント: RETURN ステートメントは、SQL プロシージャーを無条件かつ即時に終了するときに使用されます。終了するために、ストアード・プロシージャーの呼び出し元に制御のフローを戻します。

RETURN ステートメントを実行すると、必ず整数値が戻される必要があります。戻り値を指定しない場合のデフォルト値は 0 です。通常、この値は、プロシージャーの実行が正常完了または失敗のどちらであったかを示すのに使用されます。値は、リテラルまたは変数、あるいは、整数値に評価される式のいずれかにすることができます。

ストアード・プロシージャー内で 1 つ以上の RETURN ステートメントを使用することができます。 RETURN ステートメントは、SQL プロシージャー本体内の宣言ブロックの後であればどこで使用してもかまいません。

それに代えてパラメーターを使用し、複数の出力値を戻すこともできます。パラメーター値は、実行する RETURN ステートメントより前に設定する必要があります。

RETURN ステートメントを使用した SQL プロシージャーの例を以下に示します。

```
CREATE PROCEDURE return_test (IN p_empno CHAR(6),
                              IN p_emplastname VARCHAR(15) )
LANGUAGE SQL
SPECIFIC return_test
BEGIN

  DECLARE v_lastname VARCHAR (15);

  SELECT lastname INTO v_lastname
    FROM employee
  WHERE empno = p_empno;

  IF v_lastname = p_emplastname THEN
    RETURN 1;
  ELSE
    RETURN -1;
  END IF;

END rt
```

この例では、employee 表に保管されている値にパラメーター *p_emplastname* が一致した場合、プロシージャーから 1 が戻されます。一致しなかった場合、-1 が戻されます。

SQL プロシージャーの条件ハンドラー: 条件ハンドラー は、ある条件が発生したときの SQL プロシージャーの振る舞いを決定します。一般的な条件、名前付き条件、または特定の SQLSTATE 値に対して 1 つ以上の条件ハンドラーを SQL プロシージャーで宣言することができます。

SQL プロシージャー内のステートメントによって SQLWARNING または NOT FOUND 条件が生じた場合、もしそれぞれの条件に対してハンドラーを宣言していたときは、それに対応するハンドラーに DB2 から制御が渡されます。そのような条件に対してハンドラーを宣言していなかった場合、 SQL プロシージャー本体の次のステートメントに DB2 から制御が渡されます。 SQLCODE および SQLSTATE 変数を宣言していた場合、それに対応する条件値がその変数内に入ります。

特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言していた場合に、 SQL プロシージャー内のステートメントが SQLEXCEPTION 条件を生じたときは、そのハンドラーに DB2 から制御が渡されます。 SQLSTATE および SQLCODE 変数を宣言していた場合、ハンドラーの実行が正常に完了した後のその変数の値はそれぞれ'00000' と 0 になります。

特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言していなかった場合に、 SQL プロシージャーのステートメントが SQLEXCEPTION 条件を生じたときは、 DB2 は SQL プロシージャーを終了してから呼び出し元に戻ります。

条件ハンドラー用のハンドラー宣言構文は、複合 SQL (プロシージャー) ステートメントに記述されます。

SQL プロシージャーからの結果セットの戻り:

SQL プロシージャーでは、カーソルを使用して、結果セットの行を繰り返す以上のことを実行できます。カーソルを使用して、結果セットを呼び出し側プログラムに戻すこともできます。結果セットは、SQL プロシージャー (ネストされたプロシー

ジャー呼び出しの場合)、または CLI アプリケーション・プログラミング・インターフェース、Java、CLI、または .NET CLR 言語を使用して、C でプログラミングされたクライアント・アプリケーションにより取得できます。

前提条件

- SQL プロシージャーを作成する権限

SQL プロシージャーから結果セットを戻すには、以下が必要です。

1. CREATE PROCEDURE ステートメントで DYNAMIC RESULT SETS 節を指定します。
2. WITH RETURN 節を使用してカーソルに対して DECLARE を実行します。
3. SQL プロシージャー内のカーソルをオープンします。
4. クライアント・アプリケーションに対してカーソルをオープンしたままにして、それをクローズしないようにしてください。

以下に示すのは、単一の結果セットだけを戻す SQL プロシージャーの例です。

```
CREATE PROCEDURE  read_emp()
SPECIFIC read_emp
LANGUAGE SQL
DYNAMIC RESULT SETS 1

Re:  BEGIN

  DECLARE c_emp CURSOR WITH RETURN FOR
    SELECT salary, bonus, comm.
    FROM employee
    WHERE job != 'PRES';

  OPEN c_emp;

END Re
```

SQL プロシージャーの戻りの前に CLOSE ステートメントを使用してカーソルがクローズされた場合、カーソルの結果セットは呼び出し元またはクライアント・アプリケーションには戻されません。

複数の結果セットを、複数のカーソルを使用して SQL プロシージャーから戻すことができます。複数のカーソルを戻すには、以下を実行する必要があります。

- CREATE PROCEDURE ステートメントで DYNAMIC RESULT SETS 節を指定します。戻される可能性がある結果セットの最大可能数を指定します。実際に戻される結果セットの数は、この数を超えることはできません。
- WITH RETURN 節を指定する、戻される各結果セットに対してカーソルを宣言します。
- 戻されるカーソルをオープンします。
- クライアント・アプリケーションに対してカーソルをオープンしたままにする - これはクローズしないでください。

戻される結果セットごとに 1 つのカーソルが必要です。

結果セットは、オープンされる順序で呼び出し元に戻されます。

結果セットを戻す SQL プロシージャーを作成すると、それを呼び出して結果セットを取得することができます。

同じカーソルの複数インスタンスを使用可能にすることによって、複数の結果セットを戻すこともできます。 WITH RETURN TO CLIENT を使用してカーソルに対して DECLARE を実行する必要があります。

以下の例では、WITH RETURN TO CLIENT を使用して、オープン・カーソルの複数インスタンスを使用可能にします。

```
CREATE PROCEDURE PROC(IN a INT)
BEGIN
  DECLARE index INTEGER DEFAULT 1;
  WHILE index < a DO
    BEGIN
      DECLARE cur CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM T WHERE pk = index;
      OPEN cur;
      SET index = index + 1;
    END;
  END WHILE;
END
@
```

SQL ルーチンでのプロシージャーの結果セットの受け取り:

SQL-bodied ルーチン (SQL を本体としてもつルーチン) 内から呼び出したプロシージャーから、結果セットを受け取ることができます。

呼び出したプロシージャーからいくつの結果セットが戻されるかを知っている必要があります。呼び出し元のルーチンが受け取る各結果セットごとに、結果セットを宣言しなければなりません。

SQL-bodied ルーチンからのプロシージャー結果セットを受け入れるには、次のようにします。

1. プロシージャーから戻される各結果セットごとに結果セット・ロケーターを宣言します。以下に例を示します。

   ```
   DECLARE result1 RESULT_SET_LOCATOR VARYING;
   DECLARE result2 RESULT_SET_LOCATOR VARYING;
   DECLARE result3 RESULT_SET_LOCATOR VARYING;
   ```

2. プロシージャーを呼び出します。以下に例を示します。

   ```
   CALL targetProcedure();
   ```

3. 結果セット・ロケーター変数 (上記で定義済み) を呼び出し元のプロシージャーに関連付けます。以下に例を示します。

   ```
   ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
      WITH PROCEDURE targetProcedure;
   ```

4. 呼び出し先のプロシージャーから渡される結果セット・カーソルをその結果セット・ロケーターに割り振ります。以下に例を示します。

   ```
   ALLOCATE rsCur CURSOR FOR RESULT SET result1;
   ```

5. 結果セットから行を取り出します。以下に例を示します。

   ```
   FETCH rsCur INTO ...
   ```

SQL プロシージャーの作成

SQL プロシージャーの作成は、DDL SQL ステートメントの実行により構成されるという点で、データベース・オブジェクトの作成と似ています。

SQL プロシージャーは、CREATE PROCEDURE ステートメントの実行により作成されます。これは、グラフィカル開発環境ツールを使用するか、または DB2 コマンド行プロセッサー (CLP)、DB2 コマンド・ウィンドウ、DB2 コマンド・エディター、あるいは他の DB2 インターフェースからステートメントを直接実行することにより実行できます。

SQL プロシージャーを作成する場合、プリコンパイラーおよびバインダーがプロシージャー・パッケージを生成する方法、および DB2 カタログ・ビュー内で SQL プロシージャー定義者を設定したり、他のパッケージ・オプションを設定するために使用する許可 ID を指定できます。

コマンド行からの SQL プロシージャーの作成:

- ユーザーには、 SQL プロシージャーの CREATE PROCEDURE ステートメントを実行するための特権が必要です。
- プロシージャーの SQL プロシージャー本体に組み込まれているすべての SQL ステートメントを実行するための特権が必要です。
- SQL プロシージャーの CREATE PROCEDURE ステートメント内で参照されているデータベース・オブジェクトは、そのステートメントの実行前に存在している必要があります。
- 次のステップでスクリプトを作成するときに使用する終了文字として、コマンド行プロセッサー (DB2 CLP) のデフォルトの終了文字、つまりセミコロン (';') 以外の代替の終了文字を選択します。

  これが必要なのは、ルーチンの CREATE ステートメントの本体に組み込まれている SQL ステートメントの終了と、 CREATE PROCEDURE ステートメントそのものの終了を CLP が区別するためです。 SQL ルーチン本体の中に組み込む SQL ステートメントの終了を示すためにはセミコロンを使用し、 CREATE ステートメントそのものの終了を示すためにはその選択した終了文字を使用する必要があります。また、 CLP スクリプトの中にさらに他の SQL ステートメントを組み込む場合は、その選択した終了文字でそれらのステートメントの終了を示します。

  例えば、以下の CREATE PROCEDURE ステートメントでは、myCLPscript.db2 という名前の DB2 CLP スクリプトの終了文字として、アットマーク ('@') を使用しています。

```
CREATE PROCEDURE UPDATE_SALARY_IF
(IN employee_number CHAR(6), IN rating SMALLINT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
     SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

  IF (rating = 1)
    THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
```

```
      ELSEIF (rating = 2)
        THEN UPDATE employee
          SET salary = salary * 1.05, bonus = 500
          WHERE empno = employee_number;
      ELSE UPDATE employee
          SET salary = salary * 1.03, bonus = 0
          WHERE empno = employee_number;
      END IF;
    END
@
```

- コマンド行から以下の CLP コマンドを使用して、プロシージャーの CREATE PROCEDURE ステートメントを含んだ DB2 CLP スクリプトを実行します。

  ```
  db2 -td terminating-character -vf CLP-script-name
  ```

  terminating-character は、実行する CLP スクリプト・ファイル *CLP-script-name* で使用している終了文字です。

  DB2 CLP オプション **-td** は、CLP 終止符のデフォルトを *terminating-character* に再設定するという指定です。 **-vf** は、CLP のオプションの冗長 (**-v**) オプションを使用するという指定です。このオプションを指定した場合、スクリプト内の各 SQL ステートメントやコマンドがそれぞれの実行時に画面に表示され、実行結果に関する出力も表示されることになります。 **-f** オプションは、コマンドのターゲットがファイルであるという指定です。

  最初のステップで示したスクリプトを実行するには、システム・コマンド・プロンプトから以下のコマンドを実行します。

  ```
  db2 -td@ -vf myCLPscript.db2
  ```

コンパイル済み SQL オブジェクトのプリコンパイル・オプションと BIND オプションのカスタマイズ:

SQL プロシージャー、コンパイル済み関数、コンパイル済みトリガー、およびコンパウンド SQL (コンパイル済み) ステートメント用のプリコンパイルおよび BIND オプションは、インスタンス全体に適用される DB2 レジストリー変数 DB2_SQLROUTINE_PREPOPTS を、次のコマンドで設定することによってカスタマイズできます。

```
db2set DB2_SQLROUTINE_PREPOPTS=<options>
```

これらのオプションは、 SET_ROUTINE_OPTS ストアード・プロシージャーを使用して、プロシージャー・レベルで変更できます。現行セッションで SQL プロシージャーを作成するために設定されているオプションの値は、 GET_ROUTINE_OPTS 関数を使用して取得できます。
特定のルーチンをコンパイルするために使用したオプションは、システム・カタログ表 ROUTINES.PRECOMPILE_OPTIONS の該当ルーチンに対応した行に保管されます。ルーチンが再度有効になる場合、これらの保管されているオプションも再有効化の際に使用されます。
ルーチンを作成した後で、SYSPROC.ALTER_ROUTINE_PACKAGE プロシージャーおよび SYSPROC.REBIND_ROUTINE_PACKAGE プロシージャーを使用してコンパイル・オプションを変更できます。変更済みオプションは、ROUTINES_PRECOMPILE_OPTIONS システム・カタログ表に反映されます。

注: SQL プロシージャーで、FETCH ステートメントで参照されるカーソル、および FOR ステートメントでの暗黙的なカーソルに対してカーソルのブロッキングは使用できません。 BLOCKING バインド・オプションに指定される値にかかわりなく、最適化された、非常に効率的な方法で一度に 1 行ずつデータを取得します。

例

この例に使用されている SQL プロシージャーは、CLP スクリプトで定義されます (以下を参照)。これらのスクリプトは sqlpl サンプル・ディレクトリーにはありませんが、 CREATE プロシージャー・ステートメントを独自のファイルにカット・アンド・ペーストすることによって、これらのファイルを簡単に作成できます。

これらの例では「expenses」という名前の表を使用します。これは、次のようにしてサンプル・データベース内に作成できます。

```
db2 connect to sample
db2 CREATE TABLE expenses(amount DOUBLE, date DATE)
db2 connect reset
```

初めに、日付用の ISO フォーマットの使用をインスタンス全体に適用される設定として指定します。

```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
db2stop
db2start
```

変更を有効にするには、DB2 をいったん停止してから再始動する必要があります。

次に、データベースに接続します。

```
db2 connect to sample
```

最初のプロシージャーは、CLP スクリプト maxamount.db2 で次のように定義されます。

```
CREATE PROCEDURE maxamount(OUT maxamnt DOUBLE)
BEGIN
  SELECT max(amount) INTO maxamnt FROM expenses;
END @
```

オプション DATETIME ISO および ISOLATION UR を指定して作成されます。

```
db2 "CALL SET_ROUTINE_OPTS(GET_ROUTINE_OPTS() || ' ISOLATION UR')"
db2 -td@ -vf maxamount.db2
```

次のプロシージャーは、CLP スクリプト fullamount.db2 で次のように定義されます。

```
CREATE PROCEDURE fullamount(OUT fullamnt DOUBLE)
BEGIN
  SELECT sum(amount) INTO fullamnt FROM expenses;
END @
```

オプション ISOLATION CS を指定して作成されます (ここでは、インスタンス全体に適用される DATETIME ISO 設定は使用しないことに注意してください)。

```
CALL SET_ROUTINE_OPTS('ISOLATION CS')
db2 -td@ -vf fullamount.db2
```

例の最後のプロシージャーは、CLP スクリプト perday.db2 で次のように定義されます。

```
CREATE PROCEDURE perday()
BEGIN
  DECLARE cur1 CURSOR WITH RETURN FOR
    SELECT date, sum(amount)
    FROM expenses
    GROUP BY date;

    OPEN cur1;
END @
```

最後の SET_ROUTINE_OPTS 呼び出しでは、引数として NULL 値を使用しています。これによって、DB2_SQLROUTINE_PREPOPTS レジストリーに指定されているグローバル設定がリストアされるため、最後のプロシージャーはオプション DATETIME ISO を指定して作成されます。

```
CALL SET_ROUTINE_OPTS(NULL)
db2 -td@ -vf perday.db2
```

SQL プロシージャーのパフォーマンスの改善

DB2 による SQL PL とインライン SQL PL のコンパイルの概要

SQL プロシージャーのパフォーマンスを改善する方法を説明する前に、 CREATE PROCEDURE ステートメントの実行時に DB2 が SQL プロシージャーをコンパイルする方法について取り上げる必要があります。

SQL プロシージャーの作成時に、 DB2 は、プロシージャー本体の中にある SQL 照会とプロシージャー・ロジックを分離します。 SQL 照会については、パフォーマンスの最大化のために、パッケージ内のセクションに静的にコンパイルします。静的にコンパイルした照会のセクションの主な中身は、 DB2 オプティマイザーがその照会のために選択したアクセス・プランです。パッケージとは、そのようなセクションの集合です。パッケージとセクションの詳細については、「DB2 SQL 解説書」を参照してください。一方、プロシージャー・ロジックは、ダイナミック・リンク・ライブラリーにコンパイルします。

プロシージャーの実行時に、プロシージャー・ロジックから SQL ステートメントに制御が移るたびに、 DLL と DB2 エンジンとの間で「コンテキストの切り替え」が発生します。 DB2 バージョン 8.1 以降、 SQL プロシージャーは「unfenced モード」で実行されます。つまり、 DB2 エンジンと同じアドレッシング・スペースで実行されるということです。したがって、ここで言う「コンテキストの切り替え」とは、オペレーティング・システム・レベルで発生する完全な「コンテキストの切り替え」ではなく、むしろ DB2 内の層の切り替えです。頻繁に呼び出されるプロシージャー (OLTP アプリケーション内のプロシージャーなど) や、多数の行を処理するプロシージャー (データ・クレンジングを実行するプロシージャーなど) でコンテキストの切り替えの数を減らせば、パフォーマンスにかなりの影響を与えることができます。

SQL PL を含んだ SQL プロシージャーは、個々の SQL 照会をパッケージ内の各セクションに静的にコンパイルすることによってインプリメントするのに対し、インライン SQL PL 関数は、その名が示すとおり、関数の本体を、関数を使用する照会の中にインライン化することによってインプリメントします。 SQL 関数内の各照会は、あたかも関数本体が 1 つの照会であるかのように一緒にコンパイルされま

す。このコンパイルは、その関数を使用するステートメントのコンパイルが行われるたびに発生します。ただし、SQL プロシージャーの場合とは異なり、 SQL 関数内のプロシージャー・ステートメントは、データ・フロー・ステートメントとは別の層で実行されるわけではありません。したがって、プロシージャー・ステートメントとデータ・フロー・ステートメントの間で制御が移るたびに、コンテキストの切り替えが発生するわけではないということです。

ロジック内に副作用がなければ SQL 関数を使用する

このように、プロシージャー内の SQL PL と関数内のインライン SQL PL とではコンパイルの方法が違うので、プロシージャー・コードが SQL データを照会するだけでデータを変更しない限り、つまり、データベース内外のデータに関する副作用がない限り、プロシージャー・コードは、プロシージャー内よりも関数内にあったほうが実行速度が上がると考えられます。

ただし、このようなメリットを生かせるのは、実行する必要のあるすべてのステートメントが SQL 関数内でサポートされている場合に限られます。 SQL 関数には、データベースを変更する SQL ステートメントを組み込めません。また、関数のインライン SQL PL として使用できるのは、 SQL PL のサブセットにすぎません。例えば、 CALL ステートメントの実行、カーソルの宣言、 SQL 関数による結果セットの生成などは実行できません。

以下に示すのは、パフォーマンスを最大化する目的で SQL 関数に変換するのに適している SQL PL を含んだ SQL プロシージャーの一例です。

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                           IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor eq; ssq;Vendor 1ssq;
    THEN SET price eq; (SELECT ProdPrice
                         FROM V1Table
                         WHERE Id = Pid);
  ELSE IF Vendor eq; ssq;Vendor 2ssq;
    THEN SET price eq; (SELECT Price FROM V2Table
                         WHERE Pid eq; GetPrice.Pid);
  END IF;
END
```

これを SQL 関数として記述すると、以下のようになります。

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
  DECLARE price DECIMAL(10,3);
  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice
                       FROM V1Table
                       WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
               WHERE Pid = GetPrice.Pid);
  END IF;
  RETURN price;
END
```

関数の呼び出しは、プロシージャーの呼び出しとは異なることも覚えておく必要があります。関数を呼び出すには、 VALUES ステートメントを使用するか、

SELECT ステートメントや SET ステートメントなどの中で式が有効な場所に関数を記述して呼び出します。以下はいずれも、この新しい関数を呼び出す方法として有効です。

```
VALUES (GetPrice('IBM', 324))

SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

SET price = GetPrice(Vname, Pid)
```

SQL PL プロシージャー内で 1 つのステートメントを使用すれば十分な場合に複数のステートメントを使用しない

基本的に SQL は簡潔に記述するほうが良いのですが、実際には簡潔でない SQL を記述してしまうこともよくあります。例えば、次のような SQL ステートメントがあるとしましょう。

```
INSERT INTO tab_comp VALUES (item1, price1, qty1);
INSERT INTO tab_comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

これは、以下の 1 つのステートメントとして記述できます。

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

この複数行の挿入ステートメントの実行にかかる時間は、元の 3 つのステートメントの実行にかかる時間のほぼ 3 分の 1 です。これだけを取り出したコードであれば、パフォーマンスの改善はごくわずかでしょうが、ループやトリガー本体などの中でこのコード断片を繰り返し実行する場合は、かなりの改善が期待できます。

同じように、以下のような一連の SET ステートメントがあるとしましょう。

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

これは、以下の 1 つの VALUES ステートメントとして記述できます。

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

この書き換えでは、元の一連のステートメントのセマンティクスをそのまま保持しています。ただし、元のいずれか 2 つのステートメントの間に依存関係が存在する場合は別です。この点を示す以下の例について考えてみましょう。

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

この 2 つのステートメントを以下のように書き換えるとしましょう。

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

この場合は、元のセマンティクスがそのまま保持されていません。 INTO キーワードの前の両方の式は「並列的に」評価されるからです。つまり、*B* に代入される値は *A* に代入される値に基づくというのが、元のステートメントで意図されているセマンティクスですが、書き換え後のコードにはそれが反映されていないということです。

複数の SQL ステートメントを 1 つの SQL 式にまとめる

SQL 言語には、他のプログラム言語と同じように、 2 種類の条件構造体が用意されています。つまり、プロシージャー型の構造体 (IF ステートメント、CASE ステートメント) と関数型の構造体 (CASE 式) です。 1 つの計算処理を表すためにどちらのタイプの構造体でも使用できる状況では、ほとんどの場合、どちらを使用するかは好みの問題です。ただし、 CASE 式によって記述したロジックは、 CASE ステートメントや IF ステートメントによって記述したロジックよりもコンパクトであり、効率的でもあります。

以下の SQL PL コード断片について考えてみましょう。

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price);
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice);
END IF;
```

この IF 節の条件は、 tab_comp.Val 列に挿入する値を決定するという目的のためだけに使用しています。プロシージャー層とデータ・フロー層の間のコンテキストの切り替えを避けるために、この同じロジックを CASE 式付きの 1 つの INSERT で記述すれば、以下のようになります。

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
            END);
```

CASE 式は、スカラー値が有効な場所であればどんなコンテキストでも使用できるというのは注目に値します。特に便利なのは、代入の右辺で使用できるということです。以下に例を示します。

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

これは、以下のように記述できます。

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

実際に、この例の場合はさらに優れた解決策があります。

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

SQL を分析し、その書き換えを検討するために時間を取ることのメリットを過小評価しないでください。パフォーマンス上のメリットは、プロシージャーの分析と書き換えにかけた時間の何倍もの価値があるはずです。

SQL の一括設定のセマンティクスを活用する

ループ、代入、カーソルなどのプロシージャー型の構造体を使用すれば、 SQL DML ステートメントだけでは記述できない計算処理を記述できます。その一方で、プロシージャー・ステートメントが手元にあると、実際には SQL DML ステートメントだけで計算処理を記述できる場合でも、プロシージャー・ステートメントに頼ってしまう危険があります。すでに見たとおり、プロシージャーによる計算処理は、 DML ステートメントによって記述した等価の計算処理よりもパフォーマンスが桁違いに落ちることがあります。以下のコード断片について考えてみましょう。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

まずループ本体は、『複数の SQL ステートメントを 1 つの SQL 式にまとめる』の項で取り上げた書き換えを適用することによって改善できます。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

しかし、よく見ると、このコード・ブロック全体は、サブ SELECT 付きの 1 つの INSERT として記述できます。

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

元のコードでは、 SELECT ステートメントの各行で、プロシージャー層とデータ・フロー層の間のコンテキストの切り替えが発生します。一方、書き換えた後のコードでは、コンテキストの切り替えがまったく発生しないので、オプティマイザーは計算処理全体をグローバルに最適化できます。

ただし、以下のように各 INSERT ステートメントの対象になっている表がそれぞれ異なる場合、これほど劇的な単純化は不可能です。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
  ELSE
```

```
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

それでも、以下のようにすれば、SQL の一括設定の機能を活用できます。

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

このようにカーソル・ループを除去するには時間がかかりますが、既存のプロシージャー・ロジックのパフォーマンスを改善できることを考えれば、そのための価値は十分にあると言えます。

DB2 オプティマイザーに常に最新の情報を提供する

プロシージャーの作成時に、個々の SQL 照会は、パッケージ内の各セクションにコンパイルされます。 DB2 オプティマイザーが照会の実行プランを選択するための基礎になるのは、特に表の統計 (表のサイズや、列内のデータ値の相対度数など) と、照会のコンパイルの時点で使用できる索引です。表にかなりの変更があった場合は、その表に関する統計を DB2 で収集するべきです。また、統計を更新した場合や、新しい索引を作成した場合は、その表を使用する SQL プロシージャーに関連するパッケージを再バインドして、最新の統計と索引に基づくプランを DB2 で作成するようにしてください。

表の統計を更新するには、RUNSTATS コマンドを使用します。 SQL プロシージャーに関連するパッケージを再バインドするには、 DB2 バージョン 8.1 に用意されている REBIND_ROUTINE_PACKAGE 組み込みプロシージャーを使用します。例えば、プロシージャー MYSCHEMA.MYPROC のパッケージを再バインドするには、以下のコマンドを使用できます。

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

「P」は、このパッケージがプロシージャーに対応していることを示し、 'ANY' は、関数とタイプの解決時に SQL パス内のすべての関数とタイプを対象にすることを示します。詳細については、「コマンド解説書」の『REBIND コマンド』の項目を参照してください。

配列の使用

配列を使用して、アプリケーションとストアード・プロシージャーの間でデータの集合を効果的に受け渡しし、リレーショナル表を使用せずに SQL プロシージャー内でデータの一時的な集合を格納し取り扱うことができます。SQL プロシージャー内で使用可能な配列の演算子を使って、データの保管と取り出しを効率的に行うことができます。アプリケーションが適度なサイズの配列を作成するなら、巨大な配列 (数メガバイト規模) を作成するよりも、はるかに良いパフォーマンスを得ることができます。これは配列全体がメイン・メモリーに格納されるためです。詳しい追加情報は、「*関連リンク*」セクションをご覧ください。

SQL 関数

SQL 関数とは、プログラミング・サブルーチンに似たやり方で呼び出せるロジックをカプセル化するときに使用できる、SQL を使用して完全にインプリメントされた関数のことです。 SQL スカラー関数および SQL 表関数を作成できます。

データベースまたはデータベース・アプリケーション・アーキテクチャーには、SQL 関数に便利な多数のアプリケーションがあります。 SQL 関数を使用して列データに演算子を作成すると、組み込み関数に関するサポートの拡張、アプリケーション・ロジックの一層のモジュール化、データベースの全体的な設計およびデータベース・セキュリティーの改善を行うことができます。

以下のトピックでは、SQL 関数の詳細について説明します。

SQL 関数のフィーチャー

SQL 関数は、多くの汎用フィーチャーによって特徴付けられます。

SQL 関数:

- 従来の静的および動的な SQL ステートメントにまつわる制御フロー論理のインプリメンテーションをサポートする SQL プロシージャー型言語のステートメントおよびフィーチャーを使用できる。
- DB2 ファミリー・ブランド全体のデータベース製品においてサポートされている。そのような製品では、DB2 バージョン 9 でサポートされているフィーチャーが、すべてではなくても多数サポートされます。
- インプリメントしやすい。簡明で高水準の厳密に型付けされた言語が使用されるからです。
- SQL 関数のほうが、それと同等の外部関数よりも信頼性が高い。
- 入力パラメーターをサポートしている。
- SQL スカラー関数はスカラー値を戻す。
- SQL 表関数は表結果セットを戻す。
- 単純であっても強力な、条件およびエラーの処理モデルをサポートする。
- 特殊変数としての SQLSTATE および SQLCODE 値に簡単にアクセスできる。
- データベース内に置かれるので、バックアップおよびリストアがバックアップ操作およびリストア操作の一部として自動的に行われる。
- SQL ステートメント内で式がサポートされている場合にはいつでも呼び出させる。
- 他の SQL 関数または、他の言語でインプリメントされている関数に対するネストされた関数呼び出しをサポートする。
- 再帰をサポートする (動的 SQL がコンパイル済み関数で使用される場合)。
- トリガーから呼び出すことができる。
- 多数の SQL ステートメントを SQL 関数に組み込める。ただし、例外があります。SQL 関数に組み込むことのできる SQL ステートメントと組み込むことができない SQL ステートメントの完全なリストに関しては、ルーチン内で実行できる SQL ステートメントを参照してください。

SQL 関数は、広範囲にわたるサポートを提供し、それは、上記に一覧で示したものに限定されません。このプロシージャーは、ベスト・プラクティスどおりにインプ

リメントされた場合、データベース・アーキテクチャー、データベース・アプリケーション設計、およびデータベース・システムのパフォーマンスにおいて不可欠な役割を果たすことができます。

SQL 関数の設計

SQL 関数の設計は、データベース内に SQL 関数を作成する前に行うタスクです。

SQL 関数を設計するには、SQL 関数のフィーチャーに精通することが大切です。以下のトピックでは、SQL 関数の設計に関する概念の詳細を説明します。

インライン化された SQL 関数およびコンパイル済み SQL 関数:

SQL 関数のインプリメンテーションのタイプには、インライン化された SQL 関数とコンパイル済み SQL 関数の 2 つがあります。

インライン化された SQL 関数

インライン化された SQL 関数とは、RETURN ステートメントまたはインライン・コンパウンド・ステートメントである本体を持つ、CREATE FUNCTION ステートメントを使用して作成される SQL 関数です。インライン・コンパウンド・ステートメントは、BEGIN ATOMIC キーワードと END キーワードを使用して定義します。

インライン化された SQL 関数には、SQL ステートメント、およびインライン SQL PL ステートメント (SQL PL ステートメントのサブセット) を含めることができます。

コンパイル済み SQL 関数

コンパイル済み SQL 関数とは、RETURN ステートメントまたはコンパイル済みコンパウンド・ステートメントである本体を持つ、CREATE FUNCTION ステートメントを使用して作成される SQL 関数です。コンパイル済みコンパウンド・ステートメントは、BEGIN キーワードと END キーワードを使用して定義します。

ATOMIC 節を省略すると、SQL 関数がコンパイルされるので、インライン化された SQL 関数に比べて、より多くの SQL PL フィーチャーを組み込んだり参照したりできます。コンパイル済み SQL 関数には、インライン化された SQL 関数ではサポートされていない以下のフィーチャーがあります。

- 以下のような SQL PL ステートメント
  - CASE ステートメント
  - REPEAT ステートメント
- カーソルの処理
- 動的 SQL
- 条件ハンドラー

SQL 関数に関する制約事項:

SQL 関数を作成する前に、またはそのインプリメンテーションと使用に関連する問題をトラブルシューティングする場合、その制約事項に注意を払うのは大切です。

SQL 関数には、以下の制約事項が適用されます。

- SQL 表関数にコンパイル済みコンパウンド・ステートメントを含めることはできません。
- パーティション・データベース環境では、コンパイル済みコンパウンド・ステートメントが含まれる SQL スカラー関数を呼び出すことはできません。
- 定義によると、SQL 関数には WITH RETURN 節を使用して定義されたカーソルを含めることはできません。
- パーティション・データベース環境では、コンパイル済み SQL スカラー関数を呼び出すことはできません。
- コンパイル済み SQL 関数では、構造化データ・タイプ、XML データ・タイプ、LONG VARCHAR データ・タイプ、および LONG VARGRAPHIC データ・タイプはサポートされていません。
- 今回のバージョンでは、コンパイル済み SQL 関数で DECLARE TYPE ステートメントを使用することはできません。

SQL スカラー関数の作成

データベースを設計するとき、またはアプリケーションを開発するときには、SQL スカラー関数を作成するタスクを実行できます。 SOL スカラー関数を作成する一般的なケースは、複数のアプリケーションの SQL ステートメント内または複数のデータベース・オブジェクト内で参照できるように、再使用可能なロジックをカプセル化すると明らかに役立つような場合です。

SQL 関数を作成する前に、

- 100 ページの『SQL 関数』 を読んでください
- 100 ページの『SQL 関数のフィーチャー』 を読んでください
- CREATE FUNCTION (スカラー) ステートメントの実行に必要な特権があることを確認します。

制約事項

101 ページの『SQL 関数に関する制約事項』を参照してください。

1. 次のようにして CREATE FUNCTION (スカラー) ステートメントを定義します。
   a. 関数の名前を指定します。
   b. それぞれの入力パラメーターの名前とデータ・タイプを指定します。
   c. RETURNS キーワードと、スカラー戻り値のデータ・タイプを指定します。
   d. 関数本体の導入部として BEGIN キーワードを指定します。注意すべき点として、新しい関数に BEGIN ATOMIC キーワードを使用することは推奨されません。
   e. 関数本体を指定します。 RETURN 節と、スカラー戻り値または変数を指定します。
   f. END キーワードを指定します。
2. サポートされるインターフェースから CREATE FUNCTION (スカラー) ステートメントを実行します。

CREATE FUNCTION (スカラー) ステートメントが正常に実行されると、スカラー関数が作成されます。

例 1:

以下は、コンパイルされる SQL 関数の例です。

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
  RETURNS  DECIMAL(10,3)
  LANGUAGE SQL
  MODIFIES SQL
  BEGIN
    DECLARE price DECIMAL(10,3);

    IF Vendor = 'Vendor 1'
      THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2'
      THEN SET price = (SELECT Price
                        FROM V2Table
           WHERE Pid = GetPrice.Pid);
    END IF;

  RETURN price;
END
```

この関数は 2 つの入力パラメーターを取り、入力パラメーター値に基づいて条件付きで決定された単一のスカラー値を戻します。関数から戻されるまでの間、戻される値を保持するために、price というローカル変数を宣言して使用する必要があります。

例 2:

以下の例は、コンパイルされる SQL 関数定義を示しています。これには、カーソル、条件処理ルーチン・ステートメント、および REPEAT ステートメントが含まれています。

```
CREATE FUNCTION exit_func(a INTEGER)
  SPECIFIC exit_func
  LANGUAGE SQL
  RETURNS INTEGER
  BEGIN
    DECLARE val INTEGER DEFAULT 0;

    DECLARE myint INTEGER DEFAULT 0;

    DECLARE cur2 CURSOR FOR
      SELECT c2 FROM udfd1
        WHERE c1 <= a
        ORDER BY c1;

    DECLARE EXIT HANDLER FOR NOT FOUND
      BEGIN
        SIGNAL SQLSTATE '70001'
        SET MESSAGE_TEXT =
          'Exit handler for not found fired';
      END;

  OPEN cur2;

  REPEAT
    FETCH cur2 INTO val;
    SET myint = myint + val;
  UNTIL (myint >= a)
```

```
    END REPEAT;

    CLOSE cur2;

    RETURN myint;

  END@
```

スカラー関数を作成した後、関数を呼び出してテストすることができます。

SQL 表関数の作成

SQL 表関数を作成するタスクは、いつでも実行できます。

SQL 表関数を作成する前に、CREATE FUNCTION (表) ステートメントの実行に必要な特権があることを確認してください。

制約事項

101 ページの『SQL 関数に関する制約事項』を参照してください。

1. 次のようにして CREATE FUNCTION (表) ステートメントを定義します。
   a. 関数の名前を指定します。
   b. それぞれの入力パラメーターの名前とデータ・タイプを指定します。
   c. ルーチン属性を指定します。
   d. RETURNS TABLE キーワードを指定します。
   e. 関数本体の導入部として BEGIN ATOMIC キーワードを指定します。
   f. 関数本体を指定します。
   g. RETURN 節と共に括弧を指定し、その中に、戻される結果セットを定義する照会を指定します。
   h. END キーワードを指定します。
2. サポートされるインターフェースから CREATE FUNCTION (表) ステートメントを実行します。

CREATE FUNCTION (表) ステートメントが正常に実行されると、表関数が作成されます。

例 1:

以下は、コンパイルされる SQL 表関数の例です。これは、従業員の給与データの更新内容を追跡および監査するために使用されます。

```
CREATE FUNCTION update_salary  (updEmpNum CHAR(4), amount INTEGER)
RETURNS TABLE (emp_lastname VARCHAR(10),
      emp_firstname VARCHAR(10),
      newSalary INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
NO EXTERNAL ACTION
NOT DETERMINISTIC
BEGIN ATOMIC

  INSERT INTO audit_table(user, table, action, time)
    VALUES (USER, 'EMPLOYEE',
    'Salary update. Values: ' || updEmpNum || ' ' || char(amount), CURRENT_TIMESTAMP);
```

```
  RETURN (SELECT lastname, firstname, salary
    FROM FINAL TABLE(UPDATE employee SET salary = salary + amount WHERE employee.empnum = updEmpNum));

END
```

この関数は updEmpNum によって指定された従業員の給料を amount で指定された金額で更新し、audit_table という名前の監査表、ルーチンを呼び出したユーザー、変更された表の名前、およびそのユーザーによって作成された変更のタイプも記録します。 FROM 節でデータ変更ステートメントを参照する SELECT ステートメントを使用して、更新された行の値を戻します。

例 2:

以下は、コンパイルされる SQL 表関数の例です。

```
CREATE TABLE t1(pk INT

CREATE TABLE t1_archive LIKE T1%

CREATE FUNCTION archive_tbl_t1(ppk INT)
  RETURNS TABLE(pk INT, c1 INT, date)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC

  DECLARE c1 INT;

  DECLARE date DATE;

  SET (c1, date) = (SELECT * FROM OLD TABLE(DELETE FROM t1 WHERE t1.pk = ppk));

  INSERT INTO T1_ARCHIVE VALUES (ppk, c1, date);

  RETURN VALUES (pk, c1, date);
END%
```

表関数を作成した後、関数を呼び出してテストすることができます。

コンパウンド・ステートメント

1 つのコンパウンド・ステートメントを他のステートメントとグループ化して、1 つの実行可能ブロックにします。コンパウンド・ステートメントは独自に実行することもできれば、プロシージャー、関数、メソッド、およびトリガーなどのデータベース・オブジェクトの定義に組み込むこともできます。コンパウンド・ステートメント用の異なる SQL ステートメントがあります。それぞれに当てはまる独自の相違点および制約事項があるためです。

コンパウンド・ステートメントは、インライン・コンパウンド・ステートメント (これまでは動的コンパウンド・ステートメント呼ばれていました) またはコンパイル済みコンパウンド・ステートメントのいずれかになります。これら 2 つのステートメントの違いを以下に記します。

インライン・コンパウンド・ステートメント

インライン・コンパウンド・ステートメントはアトミックで、BEGIN ATOMIC キーワードと END キーワードを使用して定義します。これら 2 つのキーワードの間で、他の SQL ステートメントを定義および実行できます。インライン・コンパウンド・ステートメントには、変数宣言、SQL ス

テートメント、およびインライン SQL PL ステートメントと呼ばれる SQL PL ステートメントのサブセットを含めることができます。

コンパイル済みコンパウンド・ステートメント
コンパイル済みコンパウンド・ステートメントはアトミックではなく、BEGIN キーワードと END キーワードを使用して定義します。これら 2 つのキーワードの間で、他の SQL ステートメントを定義および実行できます。コンパイル済みコンパウンド・ステートメントには、SQL ステートメントとすべての SQL PL ステートメントを含めることができます。

コンパイル済みコンパウンド・ステートメントで使用可能な追加フィーチャーを活用する場合には、インライン・コンパウンド・ステートメントではなくコンパイル済みコンパウンド・ステートメントを使用するよう選択できます。

コンパウンド・ステートメントの使用

コンパウンド・ステートメントは、DB2 コマンド行プロセッサーから実行可能な小さなスクリプトを作成するために主に使用できます。また、ルーチンまたはトリガーの本体を定義する際にも使用します。

コンパウンド・ステートメントに関する制約事項

コンパウンド・ステートメントを作成する前に、またはそのインプリメンテーションと使用に関連する問題をトラブルシューティングする場合、その制約事項に注意を払うのは大切です。

インライン化された SQL 関数には、以下の制約事項が適用されます。

- 一部の SQL PL ステートメントのみがサポートされています。
- DECLARE TYPE ステートメントはサポートされません。

コンパイル済み SQL 関数には、以下の制約事項が適用されます。

- SQL 表関数にコンパイル済みコンパウンド・ステートメントを含めることはできません。
- パーティション・データベース環境では、コンパイル済みコンパウンド・ステートメントが含まれる SQL スカラー関数を呼び出すことはできません。
- DECLARE TYPE ステートメントはサポートされていますが、構造化データ・タイプ、XML データ・タイプ、LONG VARCHAR データ・タイプ、および LONG VARGRAPHIC データ・タイプと一緒に使用することはできません。

コンパウンド・ステートメントの作成

複数の SQL ステートメントから成るスクリプトを実行する必要がある場合のタスクとして、コンパウンド・ステートメントを作成して実行することができます。

コンパウンド・ステートメントを作成する前に、

- 105 ページの『コンパウンド・ステートメント』 を読んでください
- コンパウンド・ステートメントの実行に必要な特権があることを確認します。

制約事項

コンパウンド・ステートメントの制約事項のリストについては、

- 106 ページの『コンパウンド・ステートメントに関する制約事項』

1. コンパウンド SQL ステートメントを定義します。
2. サポートされるインターフェースからコンパウンド SQL ステートメントを実行します。

SQL ステートメントが動的に実行される場合、正常に実行されます。

以下の例は、SQL PL を含んでいるインライン・コンパウンド SQL ステートメントです。

```
BEGIN
 FOR row AS
   SELECT pk, c1, discretize(c1) AS v FROM source
 DO
   IF row.v is NULL THEN
     INSERT INTO except VALUES(row.pk, row.c1);
   ELSE
     INSERT INTO target VALUES(row.pk, row.d);
   END IF;
 END FOR;
END
```

コンパウンド・ステートメントには、キーワード BEGIN および END がバインドされています。その一環として、SQL PL の一部を成す FOR および IF/ELSE 制御ステートメントの両方が使用されています。FOR ステートメントは、一連の定義済みの行に対する反復処理で使用されます。各行ごとに列の値が検査され、条件によっては、値に基づいて一連の値が別の表に挿入されます。

第 2 章 PL/SQL サポート

PL/SQL (Procedural Language/Structured Query Language) ステートメントを、DB2 のインターフェースを使用してコンパイルおよび実行できます。このサポートにより、既存の PL/SQL ソリューションを DB2 データ・サーバーで処理できるように準備するという煩雑な作業が軽減されます。

以下のインターフェースがサポートされます。

- DB2 コマンド行プロセッサー (CLP)
- DB2 CLPPlus
- IBM Data Studio
- IBM Optim™ Development Studio

デフォルトでは、これらのインターフェースから PL/SQL ステートメントを実行できません。DB2 データ・サーバーで、PL/SQL ステートメント実行のサポートを使用可能にする必要があります。

PL/SQL のフィーチャー

PL/SQL ステートメントおよびスクリプトを、DB2 のインターフェースを使用してコンパイルおよび実行できます。

以下の PL/SQL ステートメントを実行できます。

- 無名ブロック。例えば DECLARE...BEGIN...END
- CREATE OR REPLACE FUNCTION ステートメント
- CREATE OR REPLACE PACKAGE ステートメント
- CREATE OR REPLACE PACKAGE BODY ステートメント
- CREATE OR REPLACE PROCEDURE ステートメント
- CREATE OR REPLACE TRIGGER ステートメント
- DROP PACKAGE ステートメント
- DROP PACKAGE BODY ステートメント

PL/SQL プロシージャーおよび関数は、他の PL/SQL ステートメントや DB2 SQL PL ステートメントから呼び出すことができます。 PL/SQL プロシージャーは、CALL ステートメントを使用することにより、SQL PL から呼び出すことができます。

以下のステートメントおよび言語エレメントが、PL/SQL コンテキスト内でサポートされます。

- 型宣言 (このバージョンでは、型宣言はパッケージ内でのみサポートされます。プロシージャー、関数、トリガー、および無名ブロック内ではサポートされません。)
  - 連想配列
  - レコード・タイプ

  - VARRAY タイプ
- 変数宣言
  - %ROWTYPE
  - %TYPE
- 基本的なステートメント、節、およびステートメント属性
  - 割り当てステートメント
  - NULL ステートメント
  - RETURNING INTO 節
  - ステートメント属性 (SQL%FOUND、SQL%NOTFOUND、および SQL%ROWCOUNT を含む)
- 制御ステートメントおよび構造
  - CASE ステートメント
    - 単純 CASE ステートメント (Simple CASE statement)
    - 検索 CASE ステートメント (Searched CASE statement)
  - 例外処理
  - EXIT ステートメント
  - FOR ステートメント
  - GOTO ステートメント
  - IF ステートメント
  - LOOP ステートメント
  - WHILE ステートメント
- 静的カーソル
  - CLOSE ステートメント
  - カーソル FOR ループ・ステートメント
  - FETCH ステートメント (%ROWTYPE 変数への FETCH INTO を含む)
  - OPEN ステートメント
  - パラメーター付きカーソル
  - カーソル属性
- REF CURSOR サポート
  - type REF CURSOR の変数およびパラメーター
  - Strong REF CURSOR
  - OPEN FOR ステートメント
  - JDBC アプリケーションへの REF CURSOR の戻り
- エラー・サポート
  - RAISE_APPLICATION_ERROR プロシージャー
  - RAISE ステートメント
  - SQLCODE 関数
  - SQLERRM 関数

CLP スクリプトからの PL/SQL プロシージャーおよび関数の作成

DB2 コマンド行プロセッサー (CLP) スクリプトから PL/SQL プロシージャーおよび関数を作成できます。

1. CLP スクリプト・ファイルで PL/SQL プロシージャーまたは関数の定義を作成することができます。各ステートメントの末尾を改行およびスラッシュ文字 (/) にします。 その他のステートメント終了文字もサポートされています。
2. ファイルを保存します。 この例では、ファイル名は `script.db2` です。
3. CLP からスクリプトを実行します。 ステートメントの末尾にスラッシュ文字かセミコロンを使用した場合、以下のコマンドを発行します。

   ```
   db2 -td/ -vf script.db2
   ```

   スクリプト・ファイルで別のステートメント終了文字 (@ 文字など) を使用した場合は、コマンド・ストリング内でその文字を指定する必要があります。以下に例を示します。

   ```
   db2 -td@ -vf script.db2
   ```

構文エラーがない場合は、CLP スクリプトは正常に実行されるはずです。

以下の CLP スクリプトの例では、PL/SQL 関数およびプロシージャーを作成してから、PL/SQL プロシージャーを呼び出します。

```
CONNECT TO mydb
/

CREATE TABLE emp (
     name            VARCHAR2(10),
     salary          NUMBER,
     comm            NUMBER,
     tot_comp        NUMBER
)
/

INSERT INTO emp VALUES ('Larry', 1000, 50, 0)
/
INSERT INTO emp VALUES ('Curly', 200, 5, 0)
/
INSERT INTO emp VALUES ('Moe', 10000, 1000, 0)
/

CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp
/

CREATE OR REPLACE PROCEDURE update_comp(p_name IN VARCHAR) AS
BEGIN
    UPDATE emp SET tot_comp = emp_comp(salary, comm)
      WHERE name = p_name;
END update_comp
/

CALL update_comp('Curly')
/
```

```
SELECT * FROM emp
/

CONNECT RESET
/
```

このスクリプトによって生成される出力例を以下に示します。

```
CALL update_comp('Curly')

  Return Status = 0

SELECT * FROM emp

NAME       SALARY             COMM               TOT_COMP
---------- ------...---------- ----...---------- --------...-------
Larry                    1000                 50                  0
Curly                     200                  5               4920
Moe                     10000               1000                  0

  3 record(s) selected.
```

新しいプロシージャーまたは関数を呼び出してテストします。プロシージャーの場合は、CALL ステートメントを使用します。関数の場合は、これらの関数への参照を含む照会またはその他の SQL ステートメントを実行します。

PL/SQL サポートに関する制約事項

PL/SQL のコンパイルを行う前、あるいは PL/SQL のコンパイルや実行時の問題をトラブルシューティングする際には、PL/SQL のコンパイルのサポートに関する制約事項に留意することが重要です。

このバージョンでの制約事項は、以下のとおりです。

- 以下の製品エディションでは、PL/SQL ステートメントのコンパイルおよび実行はサポートされません。
  - DB2 Express-C
- パーティション・データベース環境では、PL/SQL 関数およびトリガーは作成できません。
- データベースが Unicode データベースとして定義されていない場合、PL/SQL ステートメントおよび PL/SQL コンテキストにおいて、NCLOB データ・タイプの使用はサポートされません。 Unicode データベースでは、NCLOB データ・タイプは、DB2 DBCLOB データ・タイプにマップされます。
- XMLTYPE データ・タイプはサポートされません。
- TYPE 宣言は、関数、プロシージャー、トリガー、および無名ブロックではサポートされません。
- FOR EACH STATEMENT オプションは、PL/SQL トリガーではサポートされません。

PL/SQL サンプル・スキーマ

PL/SQL の例のほとんどは、組織内の従業員を示す PL/SQL サンプル・スキーマに基づいています。

以下のスクリプト (plsql_sample.sql) により、PL/SQL サンプル・スキーマが定義されます。

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  functions, triggers, and so on.
--
--  Create and populate tables used in the documentation examples.
--
--  Create the 'dept' table
--
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) NOT NULL CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
--
--  Create the 'emp' table
--
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
--
--  Load the 'dept' table
```

```
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,
  'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,
  'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,
  'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,
  'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,
  'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,
  'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,
  'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,
  'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,
  'New Hire');

SET SQLCOMPAT PLSQL;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor
--
CREATE OR REPLACE PROCEDURE list_emp
IS
```

```
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'YYYY/MM/DD');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name. Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
```

```
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters. Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  After statement-level triggers that display a message after
--  an insert, update, or deletion to the 'emp' table. One message
--  per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_ins_audit_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' added employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
```

```
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_upd_audit_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' updated employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_del_audit_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' deleted employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
--
--  Before row-level triggers that display employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table
--
CREATE OR REPLACE TRIGGER emp_ins_sal_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
END;
/
CREATE OR REPLACE TRIGGER emp_upd_sal_trig
    BEFORE UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    sal_diff := :NEW.sal - :OLD.sal;
    DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
END;
/
CREATE OR REPLACE TRIGGER emp_del_sal_trig
    BEFORE DELETE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END;
/
--
--  Package specification for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE emp_admin
```

```
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters. Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
```

```
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/

SET SQLCOMPAT DB2;
```

難読化

難読化では、データベース・オブジェクトの DDL ステートメント (ルーチン、トリガー、ビュー、PL/SQL パッケージなど) の本体がエンコードされます。コードを難読化すると、ユーザーはコードを読めなくなるので知的財産の保護に役立ちます。その場合も、DB2 Database for Linux®, UNIX®, and Windows® はそのコードを理解できます。

DBMS_DDL モジュールには、ルーチン、トリガー、ビュー、または PL/SQL パッケージを難読化するための次の 2 つのルーチンが用意されています。

WRAP 関数

ルーチン、トリガー、PL/SQL パッケージ、または PL/SQL パッケージの本体定義を引数として受け取り、最初のヘッダーの後に、ステートメントの他の部分の難読化バージョンが続くストリングを作成します。例えば、入力が次のようなものだったとします。

```
CREATE PROCEDURE P(a INT)
BEGIN
  INSERT INTO T1 VALUES (a);
END
```

DBMS_DDL.WRAP 関数を使用した場合、結果は次のようになります。

```
CREATE PROCEDURE P(a INT) WRAPPED SQL09072
aBcDefg12AbcasHGJG6JKHhgkjFGHHkkkljljk878979HJHui99
```

DDL ステートメントの難読化された部分にコード・ページのインバリアント文字が含まれているので、このステートメントは何かしらのコード・ページに対して有効になります。

CREATE_WRAPPED プロシージャー
上記の WRAP 関数と同じ入力を受け取りますが、難読化したテキストを返すのではなく、データベースにオブジェクトが作成されます。このオブジェクトは、コンパイラーが処理できるように内部的には難読化されていませんが、SYSCAT.ROUTINES や SYSCAT.TRIGGERS のようなカタログ・ビューでは、TEXT 列の内容が難読化されます。

難読化されたステートメントを CLP スクリプトで使用でき、その他のクライアント・インターフェースを使用して動的 SQL としてサブミットすることもできます。

難読化は、以下のステートメントに使用可能です。

- db2look (-wrap オプションを使用)
- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE

db2look ツールは、-wrap オプションが使用されると上記のすべてのステートメントを難読化します。

ブロック (PL/SQL)

PL/SQL ブロック構造は、PL/SQL プロシージャー、関数、またはトリガー定義内に組み込むこともできますし、無名ブロック・ステートメントとして独立して実行することもできます。

PL/SQL ブロック構造および無名ブロック・ステートメントには、以下に示すセクションが 1 つ以上含まれます。

- オプションの宣言セクション
- 必須の実行可能セクション
- オプションの例外セクション

これらのセクションには、SQL ステートメント、PL/SQL ステートメント、データ・タイプおよび変数の宣言、またはその他の PL/SQL 言語エレメントを含めることができます。

無名ブロック・ステートメント (PL/SQL)

PL/SQL 無名ブロック・ステートメントとは、PL/SQL 制御ステートメントおよび SQL ステートメントを含めることができる、実行可能ステートメントのことです。これを使用すると、スクリプト言語にプロシージャー・ロジックをインプリメントできます。 PL/SQL コンテキストにおいて、このステートメントは DB2 データ・サーバーでコンパイルおよび実行できます。

無名ブロック・ステートメントは、データベース内に持続せず、オプションの宣言セクション、必須の実行可能セクション、およびオプションの例外セクションという、最大 3 つセクションで構成できます。

オプションの宣言セクションには、実行可能セクション内および例外セクション内でステートメントによって使用される変数、カーソル、およびタイプの宣言を含めることができます。この宣言セクションは実行可能 BEGIN-END ブロックの前に挿入されます。

オプションの例外セクションは、BEGIN-END ブロックの末尾に挿入できます。例外セクションは、キーワード EXCEPTION で開始し、そのセクションを配置したブロックの末尾まで続く必要があります。

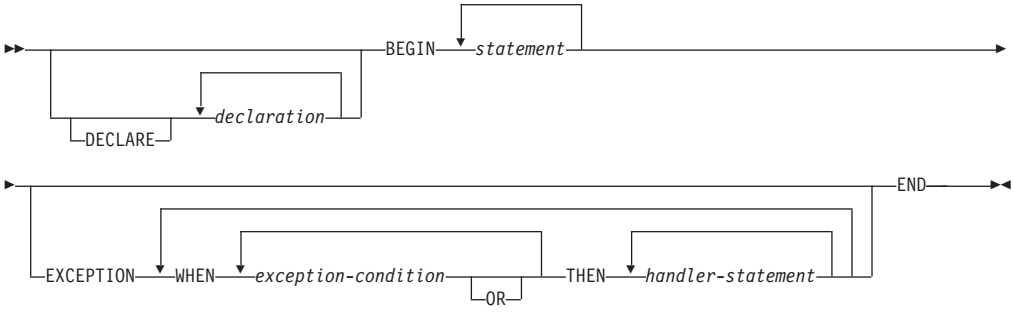
呼び出し方法

このステートメントは、対話式ツールや、CLP などのコマンド行インターフェースから実行可能です。このステートメントは、PL/SQL プロシージャー定義、関数定義、またはトリガー定義内に組み込むこともできます。これらのコンテキスト内では、このステートメントは、無名ブロック・ステートメントではなく、ブロック構造と呼ばれます。

許可

無名ブロックの呼び出しに特権は必要ありません。しかし、ステートメントの許可 ID によって保持される特権には、その無名ブロック内に組み込んだ SQL ステートメントの呼び出しに必要な特権をすべて含める必要があります。

構文



説明

DECLARE
DECLARE ステートメントを開始する、オプションのキーワード。これを使用して、データ・タイプ、変数、またはカーソルを宣言できます。このキーワードを使用するかどうかは、ブロックを配置するコンテキストによって異なります。

declaration
変数、カーソル、またはタイプの宣言を指定します。そのスコープはブロック内だけのローカルなものです。各宣言の末尾は、セミコロンでなければなりません。

BEGIN
実行可能セクションの開始を示す必須のキーワード。このセクションには、SQL または PL/SQL ステートメントを 1 つ以上含めることができます。
BEGIN-END ブロック内に、ネストした BEGIN-END ブロックを含めることができます。

statement
PL/SQL または SQL ステートメントを指定します。各ステートメントの末尾は、セミコロンでなければなりません。

EXCEPTION
例外セクションの開始を示す、オプションのキーワード。

WHEN *exception-condition*
1 つ以上のタイプの例外に関してテストする条件式を指定します。

THEN *handler-statement*
スローされた例外が *exception-condition* の例外と一致した場合に実行される、PL/SQL または SQL ステートメントを指定します。各ステートメントの末尾は、セミコロンでなければなりません。

END
ブロックを終了する必須のキーワード。

例

以下の例は、DB2 データ・サーバーでコンパイル可能な、最も単純な無名ブロック・ステートメントを示しています。

```
BEGIN
    NULL;
END;
```

以下の例では、DB2 CLP を介して対話式に入力できる無名ブロックを示します。

```
SET SERVEROUTPUT ON;

BEGIN
  dbms_output.put_line( 'Hello' );
END;
```

以下の例は、DB2 CLP を介して対話式に入力できる、宣言セクションを伴う無名ブロックを示しています。

```
SET SERVEROUTPUT ON;

DECLARE
```

```
   current_date DATE := SYSDATE;
BEGIN
     dbms_output.put_line( current_date );
END;
```

プロシージャー (PL/SQL)

DB2 データ・サーバーでは、PL/SQL プロシージャーのコンパイルおよび実行がサポートされます。 PL/SQL プロシージャーとは、CALL ステートメントまたはプロシージャー参照が有効であるコンテキスト内で呼び出し可能な PL/SQL プロシージャー・ロジックおよび SQL ステートメントが含まれる、データベース・オブジェクトのことです。

PL/SQL プロシージャーを作成するには、PL/SQL CREATE PROCEDURE ステートメントを実行します。これらのプロシージャーは、DB2 SQL DROP ステートメントを使用してデータベースからドロップできます。プロシージャーのインプリメンテーションを置き換える場合、それをドロップする必要はありません。CREATE PROCEDURE ステートメントを使用して、OR REPLACE オプションを指定すると、プロシージャーのインプリメンテーションを置き換えることができます。

CREATE PROCEDURE ステートメント (PL/SQL)

CREATE PROCEDURE ステートメントは、データベースに保管されるプロシージャーを定義します。

呼び出し方法

このステートメントは、DB2 コマンド行プロセッサー (CLP)、サポートされる対話式 SQL インターフェース、アプリケーション、またはルーチンから実行できます。

許可

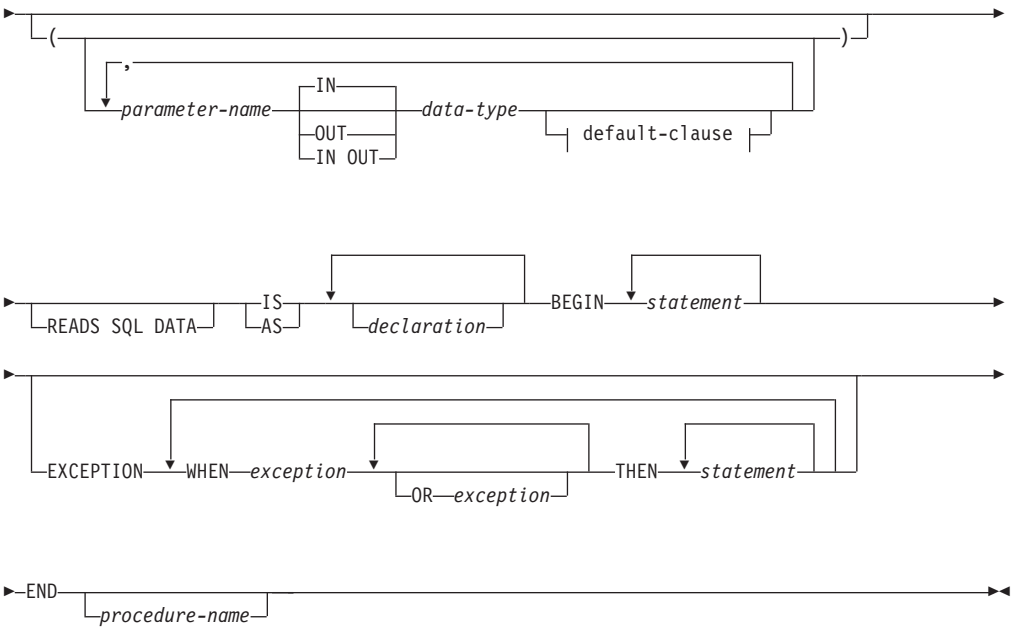
ステートメントの許可 ID によって保持される特権には、少なくとも以下のいずれか 1 つが含まれていなければなりません。

- プロシージャーのスキーマ名がない場合は、データベースに対する IMPLICIT_SCHEMA 権限
- プロシージャーのスキーマ名が既存のスキーマを参照する場合は、スキーマに対する CREATEIN 特権
- DBADM 権限

ステートメントの許可 ID によって保持される特権には、プロシージャー本体で指定される SQL ステートメントの呼び出しに必要なすべての特権も含まれていなければなりません。

OR REPLACE が指定されている場合は、ステートメントの許可 ID は、対応するプロシージャーの所有者でなければなりません (SQLSTATE 42501)。

構文



説明

PROCEDURE *procedure-name*

プロシージャーの識別名を指定します。*procedure-name* の非修飾書式は、最大長が 128 の SQL ID です。動的 SQL ステートメントでは、CURRENT SCHEMA 特殊レジスターの値を使用して、非修飾オブジェクト名を修飾します。静的 SQL ステートメントでは、QUALIFIER プリコンパイルまたはバインド・オプションが、非修飾オブジェクト名の修飾子を暗黙に指定します。*procedure-name* の修飾書式は、スキーマ名の後にピリオド文字と SQL ID が続きます。 2 部構成の名前を指定する場合は、スキーマ名の先頭を「SYS」にすることはできません。この文字を先頭にすると、エラーが戻されます (SQLSTATE 42939)。

名前 (暗黙修飾子または明示修飾子を含む) とパラメーターの数の組み合わせによって、カタログに記述されているプロシージャーを識別することはできません (SQLSTATE 42723)。非修飾名とパラメーターの数を組み合わせて使用すると、そのスキーマ内で固有になりますが、スキーマ間で固有にする必要はありません。

parameter-name

パラメーターの名前を指定します。パラメーター名は、このプロシージャーにとって固有でなければなりません (SQLSTATE 42734)。

data-type

サポートされる PL/SQL データ・タイプの 1 つを指定します。

READS SQL DATA

SQL データを変更しない SQL ステートメントを、プロシージャーに含められることを示します。この節は、DB2 の拡張機能です。

IS または AS

プロシージャー本体の定義の開始を示します。

declaration
1 つ以上の変数、カーソル、または REF CURSOR 型宣言を指定します。

BEGIN
実行可能ブロックの開始を示します。 BEGIN-END ブロックに EXCEPTION セクションを含めることができます。

statement
PL/SQL または SQL ステートメントを指定します。ステートメントの末尾は、セミコロンでなければなりません。

EXCEPTION
例外セクションの開始を示す、オプションのキーワード。

WHEN *exception-condition*
1 つ以上のタイプの例外に関してテストする条件式を指定します。

statement
PL/SQL または SQL ステートメントを指定します。ステートメントの末尾は、セミコロンでなければなりません。

END
ブロックを終了する必須のキーワード。オプションで、プロシージャーの名前を指定できます。

注

CREATE PROCEDURE ステートメントを難読化形式でサブミットできます。難読化されたステートメントでは、プロシージャー名のみを判読できます。ステートメントの他の部分は判読できないようにエンコードされますが、データベース・サーバーはその部分をデコードできます。難読化ステートメントの作成は、DBMS_DDL.WRAP 関数を呼び出すことによって行えます。

例

以下の例は、パラメーターがない単純なプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

以下の例では、IN および OUT パラメーターを持ち、ラベルが標準 PL/SQL 形式 (**<<label>>**) である GOTO ステートメントのプロシージャーを示します。

```
CREATE OR REPLACE PROCEDURE test_goto
( p1 IN INTEGER, out1 OUT VARCHAR2(30) )
IS
BEGIN
 <<LABEL2ABOVE>>
 IF p1 = 1 THEN
  out1 := out1 || 'one';
  GOTO LABEL1BELOW;
 END IF;
 if out1 IS NULL THEN
  out1 := out1 || 'two';
  GOTO LABEL2ABOVE;
 END IF;
```

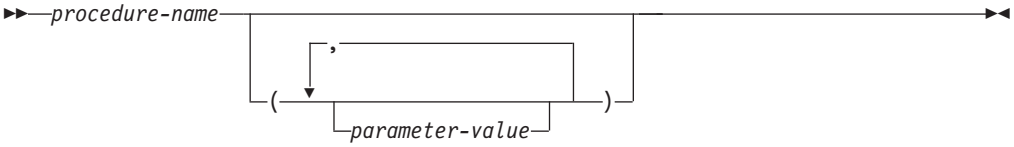
```
 out1 := out1 || 'three';

 <<LABEL1BELOW>>
 out1 := out1 || 'four';

END test_goto;
```

プロシージャー参照 (PL/SQL)

PL/SQL コンテキスト内における PL/SQL プロシージャーへの呼び出し参照を、DB2 データ・サーバーでコンパイルできます。

有効な PL/SQL プロシージャー参照は、プロシージャー名、次いでそのパラメーター (存在する場合) で構成されます。

構文



説明

procedure-name
プロシージャーの識別名を指定します。

parameter-value
パラメーター値を指定します。パラメーターを渡さない場合は、プロシージャーの呼び出しで括弧を付けても付けなくても構いません。

例

以下に、PL/SQL コンテキスト内で PL/SQL プロシージャーを呼び出す方法の例を示します。

```
BEGIN
    simple_procedure;
END;
```

DB2 データベース内に PL/SQL プロシージャーを作成した後、CALL ステートメントを使用してそれを呼び出すこともできます。このステートメントは、サポート対象の DB2 アプリケーション・プログラミング・インターフェースの使用により、DB2 SQL コンテキストおよびアプリケーションでサポートされます。

関数呼び出し構文のサポート (PL/SQL)

PL/SQL 代入文における関数呼び出し構文が、多数のプロシージャーでサポートされています。

これらのプロシージャーには、以下のものが含まれます。

- DBMS_SQL.EXECUTE
- DBMS_SQL.EXECUTE_AND_FETCH
- DBMS_SQL.FETCH_ROWS

- DBMS_SQL.IS_OPEN
- DBMS_SQL.LAST_ERROR_POSITION
- DBMS_SQL.LAST_ROW_COUNT
- DBMS_SQL.OPEN_CURSOR
- UTL_SMTP.CLOSE_DATA
- UTL_SMTP.COMMAND
- UTL_SMTP.COMMAND_REPLIES
- UTL_SMTP.DATA
- UTL_SMTP.EHLO
- UTL_SMTP.HELO
- UTL_SMTP.HELP
- UTL_SMTP.MAIL
- UTL_SMTP.NOOP
- UTL_SMTP.OPEN_DATA
- UTL_SMTP.QUIT
- UTL_SMTP.RCPT
- UTL_SMTP.RSET
- UTL_SMTP.VRFY

例

```
DECLARE
  cursor1 NUMBER;
  rowsProcessed NUMBER;
BEGIN
  cursor1 := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor1, 'INSERT INTO T1 VALUES (10)', DBMS_SQL.NATIVE);
  rowsProcessed := DBMS_SQL.EXECUTE(cursor1);
  DBMS_SQL.CLOSE_CURSOR(cursor1);
END;
/
```

```
DECLARE
  v_connection UTL_SMTP.CONNECTION;
  v_reply UTL_SMTP.REPLY;
BEGIN
  UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 25, v_connection, 10, v_reply);
  UTL_SMTP.HELO(v_connection,'127.0.0.1');
  UTL_SMTP.MAIL(v_connection, 'sender1@ca.ibm.com');
  UTL_SMTP.RCPT(v_connection, 'receiver1@ca.ibm.com');
  v_reply := UTL_SMTP.OPEN_DATA (v_connection);
  UTL_SMTP.WRITE_DATA (v_connection, 'Test message');
  UTL_SMTP.CLOSE_DATA (v_connection);
  UTL_SMTP.QUIT(v_connection);
END;
/
```

関数 (PL/SQL)

DB2 データ・サーバーでは、PL/SQL 関数のコンパイルおよび実行がサポートされます。 PL/SQL 関数とは、式が有効であるコンテキスト内で呼び出し可能な PL/SQL プロシージャー・ロジックおよび SQL ステートメントが含まれる、データベース・オブジェクトのことです。 PL/SQL 関数は、評価時に、その関数が組み込まれている式内で置換される値を返します。

PL/SQL 関数を作成するには、CREATE FUNCTION ステートメントを実行します。これらの関数は、DB2 SQL DROP ステートメントを使用してデータベースからドロップできます。関数のインプリメンテーションを置き換えたい場合であれば、ドロップする必要はありません。 CREATE FUNCTION ステートメントを使用して、OR REPLACE オプションを指定すると、関数のインプリメンテーションを置き換えることができます。

CREATE FUNCTION ステートメント (PL/SQL)

CREATE FUNCTION ステートメントは、データベースに保管される SQL スカラー関数を定義します。スカラー関数は、呼び出されるたびに単一値を返し、通常は SQL 式が有効な場合には必ず有効になります。 PL/SQL 関数は出力パラメーターをサポートしません。

呼び出し方法

このステートメントは、DB2 コマンド行プロセッサー、サポートされる対話式 SQL インターフェース、アプリケーション、またはルーチンから実行できます。

許可

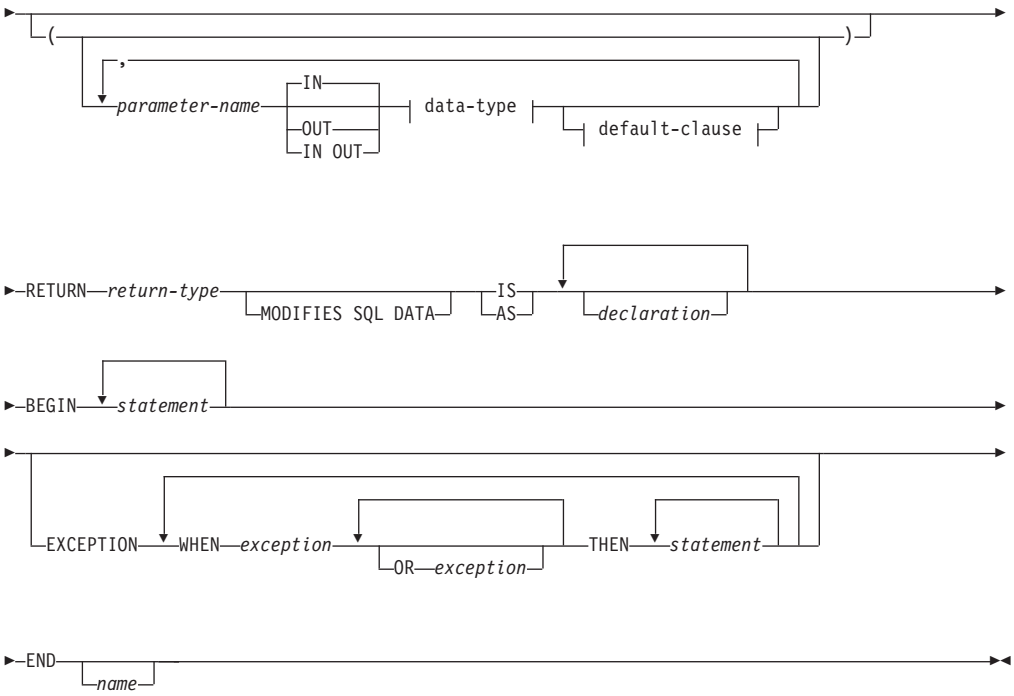
ステートメントの許可 ID によって保持される特権には、少なくとも以下のいずれか 1 つが含まれていなければなりません。

- 関数のスキーマ名がない場合は、データベースに対する IMPLICIT_SCHEMA 権限
- 関数のスキーマ名が既存のスキーマを参照する場合は、スキーマに対する CREATEIN 特権
- DBADM 権限

ステートメントの許可 ID によって保持される特権には、関数本体で指定される SQL ステートメントの呼び出しに必要なすべての特権も含まれていなければなりません。

OR REPLACE が指定されている場合は、ステートメントの許可 ID は、対応する関数の所有者でなければなりません (SQLSTATE 42501)。

構文



説明

CREATE FUNCTION ステートメントは、関数の名前、オプション・パラメーター、関数の戻りタイプ、および関数の本体を指定します。関数の本体は、BEGIN キーワードと END キーワードで囲まれるブロックです。そこには、定義済みの例外条件の発生時に行うアクションを定義する、オプションの EXCEPTION セクションを含めることができます。

OR REPLACE
スキーマ内に同じ名前の関数が既存の場合には新しい関数が既存の関数と置き換わることを示します。このオプションを指定しない場合、新しい関数によって、同じスキーマ内の同名の既存の関数を置き換えることはできません。

FUNCTION *name*
関数の ID を指定します。

parameter-name
パラメーターの名前を指定します。パラメーター・リストにある他の parameter-name と同じ名前は指定できません (SQLSTATE 42734)。

data-type
サポートされる PL/SQL データ・タイプの 1 つを指定します。

RETURN *return-type*
関数によって返されるスカラー値のデータ・タイプを指定します。

MODIFIES SQL DATA
関数でサポートされていないステートメントを除き、すべての SQL ステートメントを関数で実行できることを示します (SQLSTATE 38002 または 42985)。

この節は、DB2 の拡張機能です。ステートメントにおいて SQL データを変更する動的 SQL ステートメントが指定されている場合、これを使用する必要があ

ります。そうしないと、関数呼び出しの間に SQL データを変更しようとする動的ステートメントの実行が失敗します (SQLSTATE 38002)。

IS または AS
関数本体を定義するブロックを導入します。

declaration
1 つ以上の変数、カーソル、または REF CURSOR 型宣言を指定します。

statement
1 つ以上の PL/SQL プログラム・ステートメントを指定します。各ステートメントの末尾は、セミコロンでなければなりません。

exception
例外条件名を指定します。

注

PL/SQL 関数は、データベース・マネージャーで管理されないオブジェクトの状態を変更するアクションを行うことはできません。

CREATE FUNCTION ステートメントを難読化形式でサブミットできます。難読化されたステートメントでは、関数名のみを判読できます。ステートメントの他の部分は判読できないようにエンコードされますが、データベース・サーバーはその部分をデコードできます。難読化ステートメントの作成は、DBMS_DDL.WRAP 関数を呼び出すことによって行えます。

例

以下の例は、パラメーターがない基本的な関数を示しています。

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

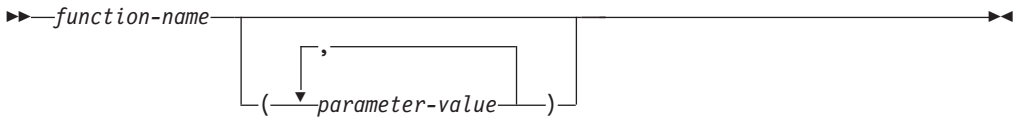
以下の例は、2 つの入力パラメーターがある関数を示しています。

```
CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

関数の参照 (PL/SQL)

式がサポートされる場合には、常に PL/SQL 関数を参照できます。

構文



説明

function-name
関数の ID を指定します。

parameter-value
パラメーターの値を指定します。

例

以下の例は、PL/SQL サンプル・スキーマで定義した SIMPLE_FUNCTION という名前の関数を、PL/SQL 無名ブロックから呼び出す方法を示しています。

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

以下の例は、SQL ステートメント内で関数を使用する方法を示しています。

```
SELECT
   empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
   emp_comp(sal, comm) "YEARLY COMPENSATION"
FROM emp
```

コレクション (PL/SQL)

PL/SQL コレクションの使用が DB2 データ・サーバーでサポートされています。PL/SQL *コレクション* とは、同じデータ・タイプを持つ、配列されたデータ・エレメントの集合のことです。その集合内の個々のデータ項目を、括弧を使用した添字表記法を使用して参照できます。

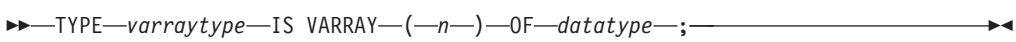
PL/SQL コンテキストにおいて、DB2 サーバーは、VARRAY コレクション・タイプおよび連想配列の両方をサポートします。

VARRAY コレクション・タイプの宣言 (PL/SQL)

VARRAY はコレクションの一種で、この中の各エレメントは、*添字* という正整数によって参照されます。 VARRAY の最大カーディナリティーは、タイプ定義で指定されます。

TYPE IS VARRAY ステートメントを使用して、VARRAY コレクション・タイプを定義します。

構文



説明

varraytype
配列型に割り当てる ID。

n 配列型のエレメントの最大数。

datatype
サポートされるデータ・タイプ (NUMBER、VARCHAR2、またはレコード・タイプなど)。%TYPE 属性および %ROWTYPE 属性もサポートされます。

例

以下の例では、EMP 表から従業員名を読み取り、タイプ VARRAY の配列変数にそれらの名前を格納してから、結果を表示します。 EMP 表には、ENAME という名前の 1 つの列が含まれます。このコードは DB2 スクリプト (script.db2) から実行します。スクリプト (db2 -tvf script.db2) を実行する前に、以下のコマンドを DB2 コマンド・ウィンドウから発行する必要があります。

```
db2set DB2_COMPATIBILITY_VECTOR=FFF
db2stop
db2start
```

スクリプトには、以下のコードが含まれます。

```
SET SQLCOMPAT PLSQL;

connect to mydb
/

CREATE PACKAGE foo
AS
    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    emp_arr         foo.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 5;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
/

DROP PACKAGE foo
/

connect reset
/
```

このスクリプトによって生成される出力例を以下に示します。

```
Curly
Larry
Moe
Shemp
Joe
```

CREATE TYPE (VARRAY) ステートメント (PL/SQL)

CREATE TYPE (VARRAY) ステートメントは、VARRAY データ・タイプを定義します。

呼び出し方法

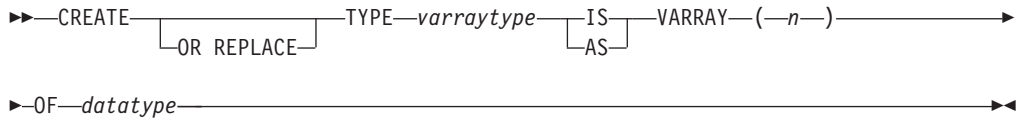
このステートメントは、DB2 コマンド行プロセッサー (CLP)、サポートされる対話式 SQL インターフェース、アプリケーション、またはルーチンから実行できます。

許可

ステートメントの許可 ID によって保持される特権には、少なくとも以下のいずれか 1 つが含まれていなければなりません。

- VARRAY タイプのスキーマ名がない場合、データベースに対する IMPLICIT_SCHEMA 権限
- VARRAY タイプのスキーマ名が既存のスキーマを参照する場合、スキーマに対する CREATEIN 特権
- DBADM 権限

構文



説明

OR REPLACE
スキーマ内に同じ名前のユーザー定義のデータ・タイプが既に存在していた場合、新しいデータ・タイプが既存のデータ・タイプと置き換わることを示します。このオプションを指定しない場合、新しいデータ・タイプによって、同じスキーマ内の同名の既存のデータ・タイプを置き換えることはできません。

varraytype
VARRAY タイプの識別子を指定します。*varraytype* の非修飾書式は、最大長が 128 の SQL ID です。 CURRENT SCHEMA 特殊レジスターの値を使用して、非修飾オブジェクト名を修飾します。 *varraytype* の修飾書式は、スキーマ名の後にピリオド文字と SQL ID が続きます。 2 部構成の名前を指定する場合は、スキーマ名の先頭を「SYS」にすることはできません。この文字を先頭にすると、エラーが戻されます (SQLSTATE 42939)。名前 (暗黙修飾子または明示修飾子を含む) は、カタログに記述されているユーザー定義のデータ・タイプを識別することはできません (SQLSTATE 42723)。非修飾名は、そのスキーマ内で固有になりますが、スキーマ間で固有にする必要はありません。

n 配列タイプの要素の最大数を指定します。あるシステム上の配列の最大カーディナリティーは、DB2 アプリケーションにとって使用可能なメモリーの総量によって制限されます。そのため、カーディナリティーの値が大きい配列 (2,147,483,647 まで) を作成できますが、すべての要素を使用できない可能性があります。

datatype
サポートされるデータ・タイプ (NUMBER、VARCHAR2、またはレコード・タイプなど) を指定します。%TYPE 属性および %ROWTYPE 属性もサポートされます。

例

以下の例では、各要素がデータ・タイプ NUMBER を持ち、最大で 10 個の要素を持つ VARRAY データ・タイプを作成します。

```
CREATE TYPE NUMARRAY1 AS VARRAY (10) OF NUMBER
```

連想配列 (PL/SQL)

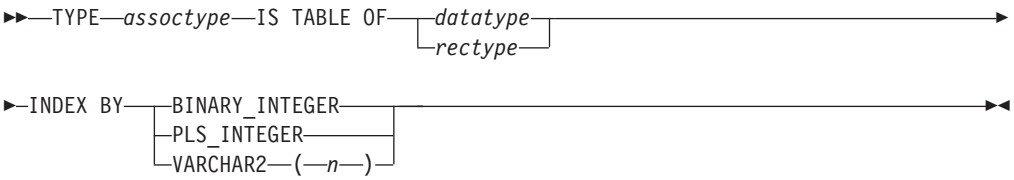
PL/SQL 連想配列とは、ユニーク・キーを値と関連付けるコレクション・タイプのことです。

連想配列には以下の特性があります。

- 連想配列タイプを定義した後でなければ、その配列タイプの配列変数を宣言できません。配列変数内でデータ操作が行われます。
- 配列を初期設定する必要はありません。配列エレメントに値を割り当てるだけです。
- 配列内のエレメント数について、定義された制限はありません。エレメントを追加するにつれて、エレメント数は動的に増加します。
- 配列は*疎* であっても構いません。すなわち、キーに対する値の割り当てに間隔があっても構いません。
- 値が割り当てられていない配列エレメントを参照しようとすると、例外が発生します。

TYPE IS TABLE OF ステートメントを使用して、連想配列タイプを定義します。

構文



説明

TYPE *assoctype*
配列タイプの ID を指定します。

datatype
スカラー・データ・タイプ (VARCHAR2 や NUMBER など) を指定します。%TYPE 属性もサポートされます。

rectype
既に定義されているレコード・タイプを指定します。%ROWTYPE 属性もサポートされます。

INDEX BY
この節によって導入されるデータ・タイプの 1 つによって、連想配列を索引付けすることを指定します。

BINARY_INTEGER
整数値データ。

PLS_INTEGER
整数値データ。

VARCHAR2 (*n*)
最大長 *n* の可変長文字ストリング。 %TYPE 属性の適用対象となるオブジェクトが、BINARY_INTEGER、PLS_INTEGER、または VARCHAR2 データ・タイプである場合、%TYPE 属性もサポートされます。

連想配列タイプの変数を宣言するには、*array-name assoctype* と指定します。ここで、*array-name* は連想配列に割り当てる ID を表し、*assoctype* は既に宣言されている配列タイプの ID を表します。

配列に含まれる特定のエレメントを参照するには、*array-name*(*n*) と指定します。ここで、*array-name* は既に宣言されている配列の ID を表し、*n* は *assoctype* の INDEX BY データ・タイプの値を表します。レコード・タイプから配列が定義されている場合は、参照は *array-name*(*n*).*field* となります。ここで、*field* は、この配列タイプを定義する元となった当該レコード・タイプ内で定義されています。このレコード全体を参照するには、*field* を省きます。

例

以下の例では、EMP 表から最初の 10 人分の従業員名を読み取り、それらを配列に格納した後、その配列の内容を表示します。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type1
IS
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
END pkg_test_type1
/

DECLARE
    emp_arr         pkg_test_type1.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END
/
```

このコードによって生成される出力例を以下に示します。

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

この例を変更して、配列定義でレコード・タイプを使用することができます。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type2
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type2
/

CREATE OR REPLACE PACKAGE pkg_test_type3
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type2.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type3
/

DECLARE
    emp_arr         pkg_test_type3.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

変更されたこのコードによって生成される出力例を以下に示します。

```
EMPNO    ENAME
-----    -------
1001     SMITH
1002     ALLEN
1003     WARD
1004     JONES
1005     MARTIN
1006     BLAKE
```

```
1007     CLARK
1008     SCOTT
1009     KING
1010     TURNER
```

この例をさらに変更して、emp_rec_typ レコード・タイプを使用する代わりに、emp%ROWTYPE 属性を使用して emp_arr_typ を定義するようにできます。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type4
IS
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
END pkg_test_type4
/

DECLARE
    emp_arr         pkg_test_type4.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

このケースでは、出力例は前述の例と同一になります。

最後の例として、以下のようにすると、レコードの各フィールドを個別に割り当てる代わりに、r_emp から emp_arr にレコード・レベルの割り当てを行うことができます。

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type5
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type5
/

CREATE OR REPLACE PACKAGE pkg_test_type6
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type5.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type6
/

DECLARE
    emp_arr         pkg_test_type6.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
```

```
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

コレクション・メソッド (PL/SQL)

コレクション・メソッドを使用すると、コレクションに関する情報を取得したり、コレクションを変更したりすることができます。

以下のコマンドを実行してから、 139 ページの表 4の例を実行するようにしてください。

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 connect to mydb
```

以下に示すように、MYDB データベースには 1 つの表 EMP が存在し、この表には 1 つの列 ENAME (VARCHAR(10) として定義) が含まれます。

```
db2 select * from emp

ENAME
----------
Curly
Larry
Moe
Shemp
Joe

  5 record(s) selected.
```

表 4. PL/SQL コンテキスト内で DB2 データ・サーバーによってサポート (または許容) されるコレクション・メソッド

コレクション・メソッド	説明	例

COUNT	コレクション内のエレメントの数を返します。	CREATE PACKAGE foo
		AS
		TYPE sparse_arr_typ IS TABLE OF NUMBER
		INDEX BY BINARY_INTEGER;
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		sparse_arr foo.sparse_arr_typ;
		BEGIN
		sparse_arr(-10) := -10;
		sparse_arr(0) := 0;
		sparse_arr(10) := 10;
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		sparse_arr.COUNT);
		END;
		/

表 4. PL/SQL コンテキスト内で DB2 データ・サーバーによってサポート (または許容) されるコレクション・メソッド (続き)

コレクション・メソッド	説明	例

DELETE	コレクションからすべてのエレメントを削除します。	CREATE PACKAGE foo
		AS
		TYPE names_typ IS TABLE OF VARCHAR2(10)
		INDEX BY BINARY INTEGER;
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		actor_names foo.names_typ;
		
		BEGIN
		
		actor_names(1) := 'Chris';
		actor_names(2) := 'Steve';
		actor_names(3) := 'Kate';
		actor_names(4) := 'Naomi';
		actor_names(5) := 'Peter';
		actor_names(6) := 'Philip';
		actor_names(7) := 'Michael';
		actor_names(8) := 'Gary';
		
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE(2);
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE(3, 5);
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		actor_names.DELETE;
		DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
		actor_names.COUNT);
		
		END;
		/

DELETE (*n*)	連想配列からエレメント *n* を削除します。 VARRAY コレクション・タイプから個々のエレメントを削除することはできません。	『DELETE』を参照してください。

表 4. PL/SQL コンテキスト内で DB2 データ・サーバーによってサポート (または許容) されるコレクション・メソッド (続き)

コレクション・メソッド	説明	例
DELETE (*n1*, *n2*)	*n1* から *n2* までのエレメントすべてを、連想配列から削除します。VARRAY コレクション・タイプから個々のエレメントを削除することはできません。	『DELETE』を参照してください。

EXISTS (*n*)	指定したエレメントが存在する場合は、TRUE を返します。	`CREATE PACKAGE foo`
		`AS`
		`  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);`
		`END;`
		`/`
		
		`SET SERVEROUTPUT ON`
		`/`
		
		`DECLARE`
		`  emp_arr  foo.emp_arr_typ;`
		`  CURSOR emp_cur IS SELECT ename FROM emp`
		`  WHERE ROWNUM <= 5;`
		`  i  INTEGER := 0;`
		`BEGIN`
		`  FOR r_emp IN emp_cur LOOP`
		`  i := i + 1;`
		`  emp_arr(i) := r_emp.ename;`
		`  END LOOP;`
		`  emp_arr.TRIM;`
		`  FOR j IN 1..5 LOOP`
		`  IF emp_arr.EXISTS(j) = true THEN`
		`  DBMS_OUTPUT.PUT_LINE(emp_arr(j));`
		`  ELSE`
		`  DBMS_OUTPUT.PUT_LINE('THIS ELEMENT`
		`  HAS BEEN DELETED');`
		`  END IF;`
		`  END LOOP;`
		`END;`
		`/`

EXTEND	コレクションに NULL エレメントを 1 つだけ付加します。	ノーオペレーション
EXTEND (*n*)	コレクションに NULL エレメントを *n* 個付加します。	ノーオペレーション
EXTEND (*n1*, *n2*)	コレクションに、*n2* 番目のエレメントのコピーを *n1* 個付加します。	ノーオペレーション

表 4. PL/SQL コンテキスト内で DB2 データ・サーバーによってサポート (または許容) されるコレクション・メソッド (続き)

コレクション・メソッド	説明	例

FIRST	コレクション内にある最小の索引番号を返します。	CREATE PACKAGE foo
		AS
		  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		  emp_arr foo.emp_arr_typ;
		  CURSOR emp_cur IS SELECT ename FROM emp
		  WHERE ROWNUM <= 5;
		  i INTEGER := 0;
		  k INTEGER := 0;
		  l INTEGER := 0;
		BEGIN
		
		  FOR r_emp IN emp_cur LOOP
		  i := i + 1;
		  emp_arr(i) := r_emp.ename;
		  END LOOP;
		
		  -- Use FIRST and LAST to specify the lower and
		  -- upper bounds of a loop range:
		  FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(j));
		  END LOOP;
		
		  -- Use NEXT(n) to obtain the subscript of
		  -- the next element:
		  k := emp_arr.FIRST;
		  WHILE k IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(k));
		  k := emp_arr.NEXT(k);
		  END LOOP;
		
		  -- Use PRIOR(n) to obtain the subscript of
		  -- the previous element:
		  l := emp_arr.LAST;
		  WHILE l IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(l));
		  l := emp_arr.PRIOR(l);
		  END LOOP;
		
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM;
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM(2);
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  DBMS_OUTPUT.PUT_LINE('Max. no. elements = ' ||
		  emp_arr.LIMIT);
		
		END;
		/

表 4. PL/SQL コンテキスト内で DB2 データ・サーバーによってサポート (または許容) されるコレクション・メソッド (続き)

コレクション・メソッド	説明	例
LAST	コレクション内にある最大の索引番号を返します。	『FIRST』を参照してください。
LIMIT	VARRAY の場合にはエレメントの最大数、ネストした表の場合には NULL を返します。	『FIRST』を参照してください。
NEXT (*n*)	指定したエレメントの直後に位置するエレメントの索引番号を返します。	『FIRST』を参照してください。
PRIOR (*n*)	指定したエレメントの直前に位置するエレメントの索引番号を返します。	『FIRST』を参照してください。
TRIM	コレクションの末尾から、エレメントを 1 つだけ削除します。連想配列コレクション・タイプからエレメントをトリムすることはできません。	『FIRST』を参照してください。
TRIM (*n*)	コレクションの末尾から、エレメントを *n* 個削除します。連想配列コレクション・タイプからエレメントをトリムすることはできません。	『FIRST』を参照してください。

変数 (PL/SQL)

変数を参照するには、前もってそれらを宣言しておく必要があります。

通常、ブロック内で使用する変数は、グローバル変数またはパッケージ・レベルの変数でない限り、ブロックの宣言セクションで定義する必要があります。宣言セクションには、ブロック内の PL/SQL ステートメントで使用できる変数、カーソル、および他のタイプの定義を含めます。変数宣言は、その変数に割り当てる名前、お

よびその変数のデータ・タイプで構成されます。オプションとして、変数宣言内で変数をデフォルト値に初期設定できます。

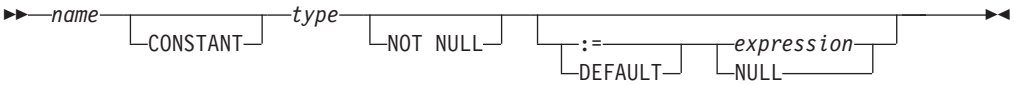
プロシージャーおよび関数に、入力値を渡すためのパラメーターを設定できます。プロシージャーには、出力値を渡すためのパラメーター、または入力値と出力値の両方を渡すためのパラメーターを設定することも可能です。

PL/SQL には、%TYPE 修飾子および %ROWTYPE 修飾子を使用して既存の列、行、またはカーソルのデータ・タイプと突き合わせるための、変数のデータ・タイプも含まれます。

変数宣言 (PL/SQL)

通常、ブロック内で使用する変数は、グローバル変数またはパッケージ・レベルの変数でない限り、ブロックの宣言セクションで定義する必要があります。変数宣言は、その変数に割り当てる名前、およびその変数のデータ・タイプで構成されます。オプションとして、変数宣言内で変数をデフォルト値に初期設定できます。

構文



説明

name
変数に割り当てる識別名を指定します。

CONSTANT
変数値を定数に指定します。デフォルトの式を割り当てる必要があります。アプリケーション・プログラム内で新規値を変数に割り当てることはできません。

type
変数のデータ・タイプを指定します。

NOT NULL
変数が NULL 値を取れないことを指定します。 NOT NULL を指定する場合、デフォルトの式を割り当てる必要があります。アプリケーション・プログラム内で変数を NULL にすることはできません。

DEFAULT
変数のデフォルト値を指定します。ブロックが入力されるたびに、このデフォルトは評価されます。例えば、タイプ DATE の変数に SYSDATE を割り当てた場合、変数は、プロシージャーまたは関数がプリコンパイルされた時刻ではなく、現在の呼び出し時刻に解決されます。

:= 代入演算子は DEFAULT キーワードの同義語です。しかし、この演算子を *expression* を付けずに指定した場合、変数は NULL 値に初期設定されます。

expression
ブロックが入力されるときに変数に割り当てられる初期値を指定します。

NULL
SQL 値 NULL を指定します。これには NULL 値が含まれます。

例

以下のプロシージャーは、ストリング式および数式で構成されたデフォルトを使用する変数宣言を示しています。

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
             || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

このプロシージャーを呼び出すことによって得られる出力例を以下に示します。

```
CALL dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

パラメーター・モード (PL/SQL)

PL/SQL プロシージャー・パラメーターは、IN、OUT、または IN OUT の 3 つのモードのいずれかに設定されます。 PL/SQL 関数パラメーターは、IN にしか設定できません。

- IN 仮パラメーターは、デフォルト値で明示的に初期設定されたのでない限り、その仮パラメーターを呼び出したときの実パラメーターに初期設定されます。 IN パラメーターは、呼び出されたプログラム内で参照できます。ただし、呼び出されたプログラムでは、新規値を IN パラメーターに割り当てることはできません。呼び出し側プログラムに制御が戻ると、実パラメーターには必ず、呼び出し前に設定された値が入っています。
- OUT 仮パラメーターは、それを呼び出したときの実パラメーターに初期設定されます。呼び出されたプログラムでは、仮パラメーターを参照し、そこに新規値を割り当てることができます。呼び出されたプログラムの終了時に例外が出されなかった場合、実パラメーターは、仮パラメーターが最後に設定されたときの値を取ります。処理済みの例外が発生すると、実パラメーターは、仮パラメーターが最後に設定されたときの値を取ります。処理できない例外が発生すると、実パラメーターの値は、呼び出し前の値のままとなります。
- IN パラメーターと同様、IN OUT 仮パラメーターは、それを呼び出したときの実パラメーターに初期設定されます。 OUT パラメーターと同様、IN OUT 仮パラメーターは、呼び出されたプログラムで変更可能です。呼び出されたプログラムの終了時に例外が出されなかった場合、仮パラメーターの最後の値が、呼び出し側プログラムの実パラメーターに渡されます。処理済みの例外が発生すると、実パラメーターは、仮パラメーターが最後に設定されたときの値を取ります。処理できない例外が発生すると、実パラメーターの値は、呼び出し前の値のままとなります。

これらの動作を要約すると、 146 ページの表 5のようになります。

表 5. パラメーター・モード

モード・プロパティー	IN	IN OUT	OUT
仮パラメーターの初期設定値	実パラメーター値	実パラメーター値	実パラメーター値
呼び出されたプログラムで仮パラメーターを変更可能	いいえ	はい	はい
呼び出されたプログラムの正常終了後に実パラメーターに格納される値	呼び出し前の元の実パラメーター値	仮パラメーターの最新値	仮パラメーターの最新値
呼び出されたプログラムで処理済みの例外が出された後に実パラメーターに格納される値	呼び出し前の元の実パラメーター値	仮パラメーターの最新値	仮パラメーターの最新値
呼び出されたプログラムで処理できない例外が出された後に実パラメーターに格納される値	呼び出し前の元の実パラメーター値	呼び出し前の元の実パラメーター値	呼び出し前の元の実パラメーター値

データ・タイプ (PL/SQL)

DB2 データ・サーバーでは、PL/SQL ブロック内での変数宣言に使用できる、多様なデータ・タイプがサポートされます。

表 6. PL/SQL で選択可能なサポート対象スカラー・データ・タイプ

PL/SQL データ・タイプ	DB2 SQL データ・タイプ	説明
BINARY_INTEGER	INTEGER	整数値データ
BLOB	BLOB (4096)	バイナリー・データ
BLOB(*n*)	BLOB (*n*) *n* は 1 から 2 147 483 647	バイナリー・ラージ・オブジェクト・データ
BOOLEAN	BOOLEAN	論理 Boolean 値 (TRUE または FALSE)
CHAR	CHAR(1)	長さ 1 の固定長文字ストリング・データ
CHAR(*n*)	CHAR (*n*) *n* は 1 から 254	長さ *n* の固定長文字ストリング・データ
CHAR VARYING (*n*)	VARCHAR(*n*)	最大長 *n* の可変長文字ストリング・データ
CHARACTER	CHARACTER (1)	長さ 1 の固定長文字ストリング・データ
CHARACTER (*n*)	CHARACTER (*n*) *n* は 1 から 254	長さ *n* の固定長文字ストリング・データ
CHARACTER VARYING (*n*)	VARCHAR (*n*) *n* は 1 から 32 672	最大長 *n* の可変長文字ストリング・データ
CLOB	CLOB (1M)	文字ラージ・オブジェクト・データ

表 6. PL/SQL で選択可能なサポート対象スカラー・データ・タイプ (続き)

PL/SQL データ・タイプ	**DB2 SQL データ・タイプ**	**説明**
CLOB(*n*)	CLOB (*n*) *n* は 1 から 2 147 483 647	長さ *n* の固定長長文字ストリング・データ
DATE	DATE [1]	日時データ (秒まで表示)
DEC	DEC (9, 2)	10 進数値データ
DEC (*p*)	DEC (*p*) *p* は 1 から 31	精度 *p* の 10 進数値データ
DEC (*p*, *s*)	DEC (*p*, *s*) *p* は 1 から 31、 *s* は 1 から 31	精度 *p* および位取り *s* の 10 進数値データ
DECIMAL	DECIMAL (9, 2)	10 進数値データ
DECIMAL (*p*)	DECIMAL (*p*) *p* は 1 から 31	精度 *p* の 10 進数値データ
DECIMAL (*p*, *s*)	DECIMAL (*p*, *s*) *p* は 1 から 31、 *s* は 1 から 31	精度 *p* および位取り *s* の 10 進数値データ
DOUBLE	DOUBLE	倍精度浮動小数点数
DOUBLE PRECISION	DOUBLE PRECISION	倍精度浮動小数点数
FLOAT	FLOAT	浮動小数点数値データ
FLOAT (*n*) *n* は 1 から 24	REAL	実数値データ
FLOAT (*n*) *n* は 25 から 53	DOUBLE	倍精度数値データ
INT	INT	符号付き 4 バイト整数値データ
INTEGER	INTEGER	符号付き 4 バイト整数値データ
LONG	CLOB (32760)	文字ラージ・オブジェクト・データ
LONG RAW	BLOB (32760)	バイナリー・ラージ・オブジェクト・データ
LONG VARCHAR	CLOB (32760)	文字ラージ・オブジェクト・データ
NATURAL	INTEGER	符号付き 4 バイト整数値データ
NCHAR	GRAPHIC (127)	固定長 GRAPHIC ストリング・データ
NCHAR (*n*) *n* は 1 から 2000	GRAPHIC (*n*) *n* は 1 から 127	長さ *n* の固定長 GRAPHIC ストリング・データ
NCLOB [2]	DBCLOB (1M)	2 バイト文字ラージ・オブジェクト・データ
NCLOB (*n*)	DBCLOB (2000)	最大長 *n* の 2 バイト長文字ストリング・データ

表 6. PL/SQL で選択可能なサポート対象スカラー・データ・タイプ (続き)

PL/SQL データ・タイプ	DB2 SQL データ・タイプ	説明
NVARCHAR2	VARGRAPHIC (2048)	可変長 GRAPHIC ストリング・データ
NVARCHAR2 (*n*)	VARGRAPHIC (*n*)	最大長 *n* の可変長 GRAPHIC ストリング・データ
NUMBER	NUMBER [3]	厳密な数値データ
NUMBER (*p*)	NUMBER (*p*) [3]	最大精度 *p* の厳密な数値データ
NUMBER (*p*, *s*)	NUMBER (*p*, *s*) [3] *p* は 1 から 31	最大精度 *p* および位取り *s* の厳密な数値データ
NUMERIC	NUMERIC (9.2)	厳密な数値データ
NUMERIC (*p*)	NUMERIC (*p*) *p* は 1 から 31	最大精度 *p* の厳密な数値データ
NUMERIC (*p*, s)	NUMERIC (*p*, *s*) *p* は 1 から 31、 *s* は 0 から 31	最大精度 *p* および位取り *s* の厳密な数値データ
PLS_INTEGER	INTEGER	整数値データ
RAW	BLOB (32767)	バイナリー・ラージ・オブジェクト・データ
RAW (*n*)	BLOB (*n*) *n* は 1 から 32 767	バイナリー・ラージ・オブジェクト・データ
SMALLINT	SMALLINT	符号付き 2 バイト整数データ
TIMESTAMP (0)	TIMESTAMP (0)	タイム・スタンプ情報を伴う日付データ
TIMESTAMP(*p*)	TIMESTAMP(*p*)	オプションの小数秒および精度 *p* を伴う日時データ
VARCHAR	VARCHAR (4096)	最大長 4096 文字の可変長文字ストリング・データ
VARCHAR(*n*)	VARCHAR(*n*)	最大長 *n* 文字の可変長文字ストリング・データ
VARCHAR2 (*n*)	VARCHAR2 (*n*) [4]	最大長 *n* 文字の可変長文字ストリング・データ

1. **DB2_COMPATIBILITY_VECTOR** レジストリー変数が DATE データ・タイプに設定されている場合、DATE は TIMESTAMP (0) と同等です。
2. 特定のデータベース環境における NCLOB データ・タイプの制約事項については、『PL/SQL サポートに関する制約事項』を参照してください。
3. このデータ・タイプは、**number_compat** データベース構成パラメーターが ON に設定されている場合にサポートされます。
4. このデータ・タイプは、**varchar2_compat** データベース構成パラメーターが ON に設定されている場合にサポートされます。

DB2 データ・サーバーでは、 146 ページの表 6に記載されているスカラー・データ・タイプに加えて、コレクション・タイプ、レコード・タイプ、および REF CURSOR タイプもサポートされます。

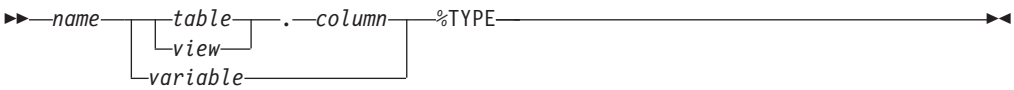
変数宣言内の %TYPE 属性 (PL/SQL)

PL/SQL 変数およびパラメーター宣言で使用する %TYPE 属性は、DB2 データ・サーバーでサポートされます。この属性を使用することにより、表の列と PL/SQL 変数の間でタイプの互換性が確実に維持されます。

ドット表記の修飾された列名、または既に宣言されている変数の名前を、%TYPE 属性の接頭部として指定する必要があります。この列または変数のデータ・タイプが、宣言する変数に割り当てられます。列または変数のデータ・タイプが変更されても、宣言コードを変更する必要はありません。

%TYPE 属性を仮パラメーターの宣言で使用することもできます。

構文



説明

name
宣言する変数または仮パラメーターの識別子を指定します。

table
参照する列がある表の識別子を指定します。

view
参照する列があるビューの識別子を指定します。

column
参照する表列またはビュー列の識別子を指定します。

variable
参照する、既に宣言されている変数の識別子を指定します。この変数は、その他の列属性 (NULL 可能属性など) を継承しません。

例

以下の例に示すプロシージャーでは、従業員番号を使用して EMP 表を照会し、同従業員のデータを表示し、同従業員が所属する部門内の全従業員の平均給与を調べ、次いで選択した同従業員の給与と部門の平均給与を比較します。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
```

```
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

このプロシージャーを、宣言セクションで EMP 表の各データ・タイプを明示的にコーディングしないように書き換えることができます。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

p_empno パラメーターは、%TYPE 属性を使用して定義する仮パラメーターの例です。 v_avgsal 変数は、表の列の代わりに別の変数を参照する %TYPE 属性の例です。

以下は、EMP_SAL_QUERY プロシージャーを呼び出すことによって生成される出力例です。

```
CALL emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

ユーザー定義レコード・タイプに基づくレコード変数 (PL/SQL)

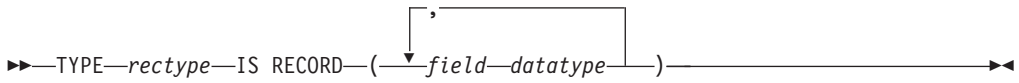
ユーザー定義のレコード・タイプ定義に基づく PL/SQL レコード変数宣言が、PL/SQL コンテキスト内で DB2 データ・サーバーによってサポートされています。

レコード・タイプ とは、1 つ以上の識別子、およびそれらに対応するデータ・タイプで構成されるレコードの定義のことです。レコード・タイプを単独で使用してデータを操作することはできません。既存のユーザー定義レコード・タイプに基づいて、PL/SQL レコード変数を宣言できます。また、PL/SQL の TYPE IS RECORD ステートメントを使用して、ユーザー定義レコード・タイプを作成できます。レコード・タイプ定義は、CREATE PACKAGE または CREATE PACKAGE BODY ステートメント内でのみサポートされます。

レコード変数 (またはレコード) とは、レコード・タイプのインスタンスのことです。レコード変数は、レコード・タイプから宣言します。レコードの各プロパティー (フィールド名およびフィールド・タイプなど) は、レコード・タイプから継承されます。

ドット表記を使用して、レコード内のフィールドを参照します。例えば、`record.field` とします。

構文



説明

TYPE *rectype* **IS RECORD**
レコード・タイプの識別子を指定します。

field
レコード・タイプのフィールドの識別子を指定します。

datatype
field の対応するデータ・タイプを指定します。 %TYPE 属性はサポートされますが、%ROWTYPE 属性はサポートされません。

例

以下の例は、ユーザー定義レコード・タイプを参照するパッケージを示しています。

```
CREATE OR REPLACE PACKAGE pkg7a
IS
TYPE t1_typ IS RECORD (
  c1 T1.C1%TYPE,
  c2 VARCHAR(10)
);
END;
```

レコード・タイプ宣言内の %ROWTYPE 属性 (PL/SQL)

%ROWTYPE 属性は、表またはビューの各列に対応したフィールドで構成されるタイプ・レコードの PL/SQL 変数を宣言するときに使用します。この属性は、DB2 データ・サーバーでサポートされます。 PL/SQL レコード内の各フィールドには、表内の対応する列のデータ・タイプが付与されます。

レコード とは、複数のフィールドからなる、名前が付いた順序付きコレクションのことです。*フィールド* は変数に似ていて、ID およびデータ・タイプがありますが、レコードにも属しており、レコード名を修飾子にしてドット表記の使用により参照する必要があります。

構文

説明

record
レコードの識別子を指定します。

table
レコード内のフィールドを定義するために使用する列定義を持つ表の識別子を指定します。

view
レコード内のフィールドを定義するために使用する列定義を持つビューの識別子を指定します。

%ROWTYPE
レコード・フィールドのデータ・タイプを、指定した表またはビューに関連付けられた列のデータ・タイプから取得することを指定します。レコード・フィールドは、その他の列属性 (NULL 可能属性など) を継承しません。

例

以下の例では、EMP 表の列の変数を個別に宣言する代わりに、%ROWTYPE 属性を使用してレコード (r_emp と命名) を作成する方法を示します。

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
```

```
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

基本的なステートメント (PL/SQL)

PL/SQL アプリケーションで使用できるプログラミング・ステートメントには、割り当て、DELETE、EXECUTE IMMEDIATE、INSERT、NULL、SELECT INTO、および UPDATE などがあります。

NULL ステートメント (PL/SQL)

NULL ステートメントは、何も実行しない実行可能ステートメントです。 NULL ステートメントは、実行可能ステートメントが必要であるものの、SQL 操作が求められていない場合 (例えば、IF-THEN-ELSE ステートメントのブランチ内など) に、いつでもプレースホルダーとして使用できます。

構文

例

以下の例は、DB2 データ・サーバーでコンパイル可能な、最も単純で有効な PL/SQL プログラムを示しています。

```
BEGIN
    NULL;
END;
```

以下の例は、IF...THEN...ELSE ステートメント内の NULL ステートメントを示しています。

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator     IN  NUMBER,
    p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
```

```
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

代入文 (PL/SQL)

代入文は既に宣言されている変数や OUT、または IN OUT の仮引数を式の値に設定します。

構文

説明

variable
既に宣言されている変数、OUT 仮パラメーター、または IN OUT 仮パラメーターの ID を指定します。

expression
単一値に評価される式を指定します。この値のデータ・タイプは、*variable* のデータ・タイプと互換性を持っている必要があります。

例

以下の例は、プロシージャーの実行可能セクション内の代入文を示しています。

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno       IN   NUMBER,
    p_base_annual  OUT  NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    p_base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || p_base_annual);
END
/
```

EXECUTE IMMEDIATE ステートメント (PL/SQL)

EXECUTE IMMEDIATE ステートメントは、SQL ステートメントの文字ストリング書式からステートメントの実行可能書式を準備してから、その SQL ステートメントを実行します。 EXECUTE IMMEDIATE は、PREPARE ステートメントと EXECUTE ステートメントの基本機能を兼ね備えます。

呼び出し方法

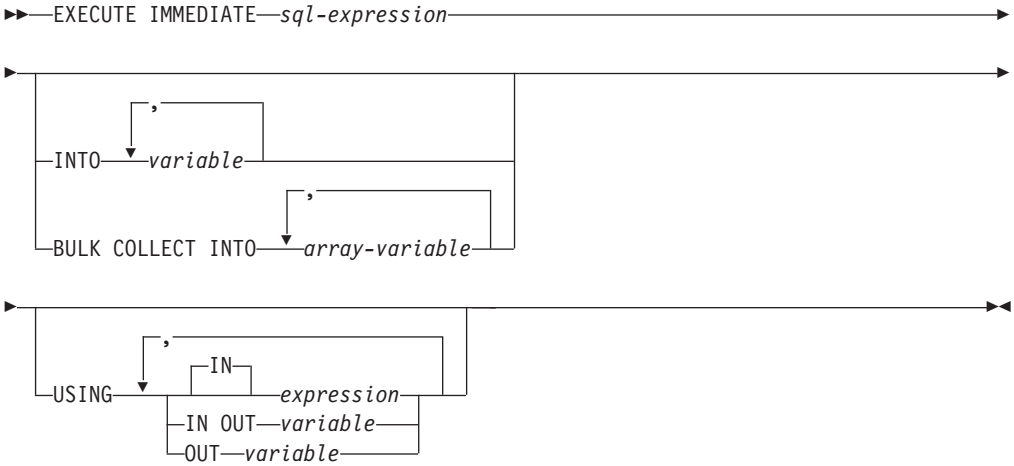
このステートメントは、PL/SQL コンテキスト内のみで指定できます。

許可

許可規則は、指定した SQL ステートメントに対して定義済みの規則です。

ステートメントの許可 ID は、DYNAMICRULES バインド・オプションの影響を受けることがあります。

構文



説明

sql-expression

実行されるステートメント・ストリングを返す式。この式は、文字ストリング・タイプを、最大ステートメント・サイズの 2 097 152 バイト未満で返す必要があります。 CLOB(2097152) には最大サイズのステートメントを含めることができますが、VARCHAR には含めることができないので注意してください。

ステートメント・ストリングは、次のいずれかの SQL ステートメントでなければなりません。

- ALTER
- CALL
- COMMENT
- COMMIT
- コンパウンド SQL (コンパイル済み)
- コンパウンド SQL (インライン化)
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- EXPLAIN

- FLUSH EVENT MONITOR
- FLUSH PACKAGE CACHE
- GRANT
- INSERT
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SELECT (EXECUTE IMMEDIATE ステートメントが BULK COLLECT INTO 節も指定した場合のみ)
- SET COMPILATION ENVIRONMENT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE
- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT SQL_CCFLAGS
- SET ROLE (DYNAMICRULES 実行動作がパッケージに対して有効な場合のみ)
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE (DYNAMICRULES 実行動作がパッケージに対して有効な場合のみ)
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA

- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET 変数
- TRANSFER OWNERSHIP (DYNAMICRULES 実行動作がパッケージに対して有効な場合のみ)
- TRUNCATE (DYNAMICRULES 実行動作がパッケージに対して有効な場合のみ)
- UPDATE

ステートメント・ストリングにステートメント終止符を含めることはできません。ただし、コンパウンド SQL ステートメントは例外で、セミコロン (;) を含めることによりコンパウンド・ブロック内でステートメントを区切ることができます。コンパウンド SQL ステートメントは一部の CREATE および ALTER ステートメント内で使用されるので、この場合にもセミコロンが含まれる可能性があります。

EXECUTE IMMEDIATE ステートメントの実行時に、指定されたステートメント・ストリングは構文解析され、エラーがないか検査されます。 SQL ステートメントが無効である場合は、実行されず、例外がスローされます。

INTO *variable*
対応するパラメーター・マーカーから出力値を受け取る変数の名前を指定します。

BULK COLLECT INTO *array-variable*
1 つ以上の変数を 1 つの配列データ・タイプとみなします。添字を順に割り当てながら、結果セットの順序でそれぞれの配列の 1 つの要素へ、照会の各行を割り当てます。

- 1 つの *array-variable* のみを指定した場合。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプではない場合、SELECT リストには必ず 1 つの列が必要となり、列データ・タイプは配列エレメントのデータ・タイプに割り当て可能でなければなりません。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプの場合、SELECT リストはレコード・タイプに割り当て可能でなければなりません。
- 複数の配列変数を指定した場合。
  - *array-variable* エレメントのデータ・タイプは、レコード・タイプであってはなりません。
  - SELECT リストの各列に *array-variable* が必要になります。
  - SELECT リストの各列のデータ・タイプは、対応する *array-variable* の配列エレメントのデータ・タイプに割り当て可能でなければなりません。

array-variable のデータ・タイプが通常配列の場合、最大カーディナリティーは照会によって戻される行数以上である必要があります。

sql-expression が SELECT ステートメントの場合のみ、この節を使用できます。

USING

IN *expression*
入力パラメーター・マーカーに渡される値を指定します。IN がデフォルトです。

IN OUT *variable*
対応するパラメーター・マーカーとの間で入力値を提供したり出力値を受け取ったりする変数の名前を指定します。

OUT *variable*
対応するパラメーター・マーカーから出力値を受け取る変数の名前を指定します。

評価される式または変数の数と順序は、*sql-expression* 内のパラメーター・マーカーの数と順序に一致し、互換性のあるタイプでなければなりません。

注

- ステートメント・キャッシングは、EXECUTE IMMEDIATE ステートメントの動作に影響します。

例

```
CREATE OR REPLACE PROCEDURE proc1( p1 IN NUMBER, p2 IN OUT NUMBER, p3 OUT NUMBER )
IS
BEGIN
  p3 := p1 + 1;
  p2 := p2 + 1;
END;
/

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' USING IN p1 + 10, IN OUT p3,
  OUT p2;

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' INTO p3, p2 USING p1 + 10, p3;
```

SQL ステートメント (PL/SQL)

PL/SQL コンテキスト内でサポートされる SQL ステートメントを使用すると、データを変更したり、ステートメントを実行する方法を指定したりすることができます。

表 7は、これらのステートメントのリストです。 PL/SQL コンテキストでの実行時におけるこれらのステートメントの動作は、対応する DB2 SQL ステートメントの動作に相当します。

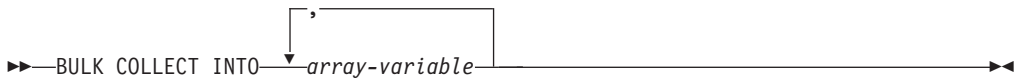
表 7. PL/SQL コンテキスト内で DB2 サーバーにより実行可能な SQL ステートメント

コマンド	説明
DELETE	表から行を削除します。
INSERT	表に行を挿入します。
MERGE	ソース (表参照の結果) からのデータを使用し、ターゲット (表またはビュー) を更新します。
SELECT INTO	表から行を取り出します。
UPDATE	表内の行を更新します。

BULK COLLECT INTO 節 (PL/SQL)

INTO キーワードの前にオプションの BULK COLLECT キーワードがある SELECT INTO ステートメントは、複数行を取り出し 1 つの配列へ格納します。

構文



説明

BULK COLLECT INTO *array-variable*

1 つ以上の変数を 1 つの配列データ・タイプとみなします。添字を順に割り当てながら、結果セットの順序でそれぞれの配列の 1 つの要素へ結果の各行を割り当てます。

- 1 つの *array-variable* のみを指定した場合。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプではない場合、SELECT リストには必ず 1 つの列が必要となり、列データ・タイプは配列エレメントのデータ・タイプに割り当て可能でなければなりません。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプの場合、SELECT リストはレコード・タイプに割り当て可能でなければなりません。
- 複数の配列変数を指定した場合。
  - *array-variable* エレメントのデータ・タイプは、レコード・タイプであってはなりません。
  - SELECT リストの各列に *array-variable* が必要になります。
  - SELECT リストの各列のデータ・タイプは、対応する *array-variable* の配列エレメントのデータ・タイプに割り当て可能でなければなりません。

array-variable のデータ・タイプが通常配列の場合、最大カーディナリティーは照会によって戻される行数以上である必要があります。

注

- どの BULK COLLECT INTO 節のバリエーションも、FETCH ステートメントおよび EXECUTE IMMEDIATE ステートメントがサポートされます。

例

以下の例は、プロシージャーから行の配列を戻すために BULK COLLECT INTO 節を使用するプロシージャーを示しています。プロシージャーおよび配列のタイプは、パッケージで定義されています。

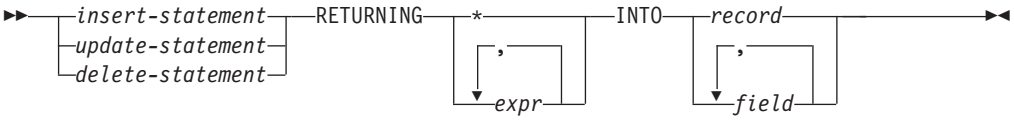
```
CREATE OR REPLACE PACAKGE bci_sample
IS
 TYPE emps_array IS VARRAY (30) OF VARCHAR2(6);

 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  );
END bci_sample;
```

```
CREATE OR REPLACE PACKAGE BODY bci_sample
IS
 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  )
 IS
  BEGIN
   SELECT empno BULK COLLECT INTO emps_dno
    FROM emp
    WHERE deptno=dno;
  END get_dept_empno;
END bci_sample;
```

RETURNING INTO 節 (PL/SQL)

オプションの RETURNING INTO 節を付加した INSERT、UPDATE、および DELETE ステートメントを、DB2 データ・サーバーでコンパイルできます。この節を PL/SQL コンテキスト内で使用すると、INSERT、UPDATE、または DELETE ステートメントの実行によって新たに追加、変更、または削除された値がそれぞれ取り込まれます。

構文



説明

insert-statement
有効な INSERT ステートメントを指定します。この INSERT ステートメントにより複数の行を含む結果セットが返された場合、例外が発生します。

update-statement
有効な UPDATE ステートメントを指定します。この UPDATE ステートメントにより複数の行を含む結果セットが返された場合、例外が発生します。

delete-statement
有効な DELETE ステートメントを指定します。この DELETE ステートメントにより複数の行を含む結果セットが返された場合、例外が発生します。

RETURNING *
INSERT、UPDATE、または DELETE ステートメントによる影響を受ける行の値すべてを割り当てることができることを指定します。

RETURNING *expr*
INSERT、UPDATE、または DELETE ステートメントによる影響を受ける行に対して評価を行う式を指定します。評価済みの結果は、指定のレコードまたはフィールドに割り当てられます。

INTO *record*
戻り値を、互換性のあるフィールドおよびデータ・タイプを持つレコードに格納することを指定します。これらのフィールドは、数、順序、およびデータ・タイ

プについて、RETURNING 節で指定した値と一致している必要があります。結果セットに行が含まれない場合、レコード内のフィールドは NULL 値に設定されます。

INTO *field*
戻り値を、互換性のあるフィールドおよびデータ・タイプを持つ一連の変数に格納することを指定します。これらのフィールドは、数、順序、およびデータ・タイプについて、RETURNING 節で指定した値と一致している必要があります。結果セットに行が含まれない場合、フィールドは NULL 値に設定されます。

例

以下の例は、RETURNING INTO 節を使用するプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

このプロシージャーは、次の出力例を返します。

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
```

```
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

以下の例は、レコード・タイプと共に RETURNING INTO 節を使用するプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

このプロシージャーは、次の出力例を返します。

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

ステートメント属性 (PL/SQL)

SQL%FOUND、SQL%NOTFOUND、および SQL%ROWCOUNT は、SQL ステートメントによる影響を見極めるために使用できる PL/SQL 属性です。

- SQL%FOUND 属性は Boolean 値を持ち、INSERT、UPDATE、または DELETE ステートメントによって少なくとも 1 行が影響を受けた場合、あるいは SELECT INTO ステートメントによって 1 行が取得された場合に、TRUE を返します。以下の例では、1 行が挿入され、状況メッセージが表示される無名ブロックを示します。

  ```
  BEGIN
      INSERT INTO emp (empno,ename,job,sal,deptno)
          VALUES (9001, 'JONES', 'CLERK', 850.00, 40);
      IF SQL%FOUND THEN
          DBMS_OUTPUT.PUT_LINE('Row has been inserted');
      END IF;
  END;
  ```

- SQL%NOTFOUND 属性は Boolean 値を持ち、INSERT、UPDATE、または DELETE ステートメントによる影響を受けた行がない場合、あるいは SELECT INTO ステートメントによって行が取得されなかった場合に、TRUE を返します。以下に例を示します。

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;
```

- SQL%ROWCOUNT 属性は整数値を持ちます。この値は、INSERT、UPDATE、または DELETE ステートメントによる影響を受けた行の数を表します。以下に例を示します。

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;
```

制御ステートメント (PL/SQL)

制御ステートメントとは、SQL を完全に補完するプロシージャーとして PL/SQL を機能させるプログラミング・ステートメントのことです。

いくつかの PL/SQL 制御ステートメントを DB2 データ・サーバーでコンパイルできます。

IF ステートメント (PL/SQL)

IF ステートメントを PL/SQL コンテキスト内で使用すると、特定の基準に基づいて SQL ステートメントを実行できます。

IF ステートメントには、以下に示す 4 つの形式があります。

- IF...THEN...END IF
- IF...THEN...ELSE...END IF
- IF...THEN...ELSE IF...END IF
- IF...THEN...ELSIF...THEN...ELSE...END IF

IF...THEN...END IF

このステートメントの構文は次のとおりです。

```
IF boolean-expression THEN
  statements
END IF;
```

IF...THEN ステートメントは、最も単純な形式の IF です。 THEN と END IF の間のステートメントは、条件が TRUE と評価された場合にのみ実行されます。以下の例では、IF...THEN ステートメントを使用することにより、歩合を受ける従業員を調べて、表示します。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
```

```
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

このプログラムは、次の出力例を生成します。

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

IF...THEN...ELSE...END IF

このステートメントの構文は次のとおりです。

```
IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

IF...THEN...ELSE ステートメントでは、条件が FALSE と評価された場合に実行する、別のステートメント一式を指定します。以下の例では、IF...THEN...ELSE ステートメントを使用することにより、従業員が歩合を受け取っていない場合に「Non-commission」というテキストが表示されるよう、前述の例を変更しています。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

このプログラムは、次の出力例を生成します。

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

IF...THEN...ELSE IF...END IF

このステートメントの構文は次のとおりです。

```
IF boolean-expression THEN
  IF boolean-expression THEN
    statements
ELSE
  IF boolean-expression THEN
    statements
END IF;
```

IF ステートメントをネストして、外側の IF ステートメントの条件が TRUE または FALSE のどちらに評価されるかに応じて、別の IF ステートメントが呼び出されるようにできます。以下の例では、外側の IF...THEN...ELSE ステートメントにより、従業員が歩合を受けるかどうかが検査されます。次いで、内側の IF...THEN...ELSE ステートメントにより、その従業員の報酬総額が、会社の平均より多いか少ないかを検査します。この形式の IF ステートメントを使用する場合、具体的には外側の IF ステートメントの ELSE 部分の内部に IF ステートメントをネストすることになります。したがって、ネストした各 IF に対して 1 つの END IF、さらに親の IF...ELSE に対して 1 つの END IF が必要です。 (なお、このプログラムのロジックは、カーソル宣言の SELECT ステートメント内で NVL 関数を使用して各従業員の年間報酬額を計算することにより、大幅に簡略化できます。しかし IF ステートメントの使用法を示すためにこの例を挙げています。)

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
```

```
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
--  Test whether the employee's compensation with commission exceeds
--  the company average
--
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
--
--  Test whether the employee's compensation without commission exceeds
--  the company average
--
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

このプログラムは、次の出力例を生成します。

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

IF...THEN...ELSIF...THEN...ELSE...END IF

このステートメントの構文は次のとおりです。

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements ] ...]
[ ELSE
  statements ]
END IF;
```

IF...THEN...ELSIF...ELSE ステートメントにより、1 つのステートメント内で多くの別の条件を検査できます。形式上、このステートメントは、必要な END IF は 1 つだけであることを除けば、ネストした IF... THEN... ELSE... IF...THEN ステートメントと同等です。以下の例では、IF...THEN...ELSIF...ELSE ステートメントを使用して、$25,000 刻みで報酬別に従業員数をカウントします。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

このプログラムは、次の出力例を生成します。

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

CASE ステートメント (PL/SQL)

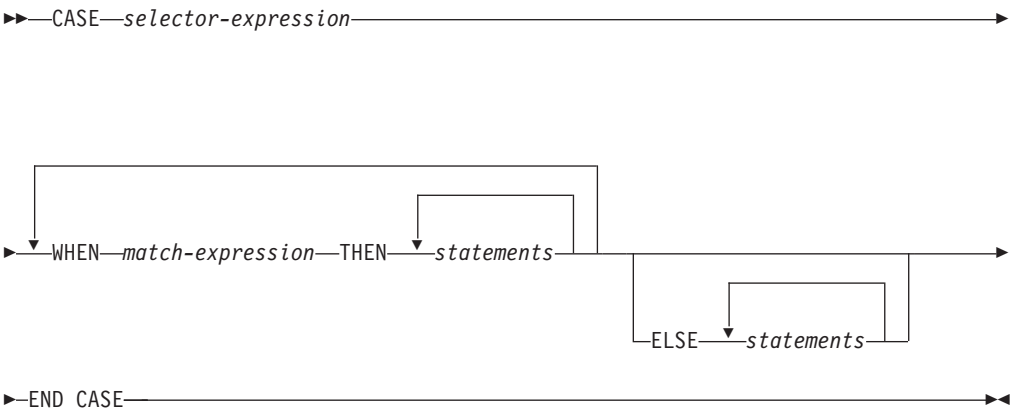
CASE ステートメントにより、指定した検索条件が真の場合に 1 つ以上のステートメント一式を実行します。 CASE は独立型のステートメントであり、式の一部として使用する必要がある CASE 式とは異なります。

CASE ステートメントには、単純 CASE ステートメント (Simple CASE statement) および検索 CASE ステートメント (Searched CASE statement) という 2 つの形式があります。

単純 CASE ステートメント (Simple CASE statement) (PL/SQL)

単純 CASE ステートメント (Simple CASE statement) では、式 (*選択子* という) を、1 つ以上の WHEN 節内に指定した別の式と突き合わせます。一致すると、対応する 1 つ以上のステートメントが実行されます。

構文



説明

CASE *selector-expression*

各 *match-expression* と互換性のあるデータ・タイプの値を持つ式を指定します。 *selector-expression* の値が 1 番目の *match-expression* と一致すると、対応する THEN 節内のステートメントが実行されます。一致するものがない場合は、対応する ELSE 節内のステートメントが実行されます。一致するものがなく、かつ ELSE 節がない場合は、例外がスローされます。

WHEN *match-expression*

CASE ステートメント内で評価する式を指定します。 *selector-expression* が *match-expression* と一致すると、対応する THEN 節内のステートメントが実行されます。

THEN

対応する Boolean 式が TRUE と評価された場合に実行するステートメントの開始を示すキーワード。

statements

1 つ以上の SQL または PL/SQL ステートメントを指定します。各ステートメントの末尾にセミコロンを付けます。

ELSE

CASE ステートメントにおけるデフォルトのケースの開始を示すキーワード。

例

以下の例では、単純 CASE ステートメントを使用することにより、部門番号に基づいて、部門の名前および場所を変数に割り当てます。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

このプログラムによって返される出力例を以下に示します。

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

検索 CASE ステートメント (Searched CASE statement) (PL/SQL)

検索 CASE ステートメント (Searched CASE statement) では、1 つ以上の Boolean 式を使用して、実行するステートメントを特定します。

構文

説明

CASE
CASE ステートメントにおける 1 番目の WHEN 節の開始を示すキーワード。

WHEN *boolean-expression*
評価される式を指定します。式の評価は、この式を定義した WHEN 節に制御フローが入る際に行われます。 *boolean-expression* が TRUE と評価されると、対応する THEN 節内のステートメントが実行されます。 *boolean-expression* が TRUE と評価されない場合は、対応する ELSE 節内のステートメントが実行されます。

THEN
対応する Boolean 式が TRUE と評価された場合に実行するステートメントの開始を示すキーワード。

statements
1 つ以上の SQL または PL/SQL ステートメントを指定します。各ステートメントの末尾にセミコロンを付けます。

ELSE
CASE ステートメントにおけるデフォルトのケースの開始を示すキーワード。

例

以下の例では、検索 CASE ステートメントを使用することにより、部門番号に基づいて、部門の名前および場所を変数に割り当てます。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
```

```
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

このプログラムによって返される出力例を以下に示します。

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

ループ (PL/SQL)

EXIT、FOR、LOOP、および WHILE ステートメントを使用すると、PL/SQL プログラム内で一連のコマンドを繰り返すことができます。

FOR (カーソル可変) ステートメント (PL/SQL)

カーソル FOR ループ・ステートメントは、既に宣言されているカーソルをオープンし、カーソル結果セット内の行をすべて取り出してから、カーソルをクローズします。

複数の SQL ステートメントを別個に使用する代わりにこのステートメントを使用することにより、カーソルをオープンし、結果セットの各行を取得するループ構成体を定義し、結果セットの終わりに達しているかのテストを行ってから、最後にカーソルをクローズすることができます。

呼び出し方法

このステートメントは、PL/SQL プロシージャー、関数、トリガー、または無名ブロック内で呼び出すことができます。

許可

SQL ステートメント内で行の式を参照するための特定の許可は必要ありません。しかし、ステートメントを正常に実行するには、カーソルの処理に必要な他の許可すべてが必要です。

構文

説明

FOR
FOR ループを続行するために真になる必要がある条件の開始を示します。

record
定義 cursor%ROWTYPE を使用して暗黙に宣言されたレコードに割り当てられた ID を指定します。

IN *cursor*
既に宣言されているカーソルの名前を指定します。

LOOP および END LOOP
ループ全体で毎回の反復中に実行される SQL ステートメントを含むループの開始と終了。

statements
1 つ以上の PL/SQL ステートメント。少なくとも 1 つのステートメントが必要です。

例

以下の例は、カーソル FOR ループが含まれるプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
  END LOOP;
END;
```

FOR (整数可変) ステートメント (PL/SQL)

FOR ステートメントを使用すると、一連の SQL ステートメントを複数回実行できます。

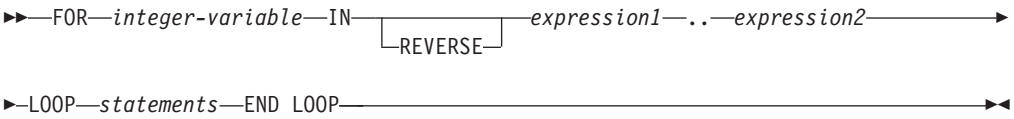
呼び出し方法

このステートメントは、PL/SQL プロシージャー、関数、または無名ブロック・ステートメント内に組み込むことができます。

許可

FOR ステートメントを呼び出すために特権は必要ありません。ただし、ステートメントの許可 ID は、FOR ステートメント内に組み込んだ SQL ステートメントの呼び出しに必要な特権を保持していなければなりません。

構文



説明

integer-variable
ループ処理中に使用される、自動的に定義された整数変数。 *integer-variable* の初期値は *expression1* です。最初の反復後、*integer-variable* の値は、以後の反復が開始されるたびに増分されます。 *expression1* および *expression2* は両方ともループ突入時に評価され、*integer-variable* が *expression2* に等しくなるとループ処理は停止します。

IN この後に、オプションの REVERSE キーワードと、ループにおける整数変数の範囲を定義する式が置かれます。

REVERSE
expression2 から *expression1* まで反復が続行されることを指定します。ループ内のステートメントを処理するには、REVERSE キーワードを指定するかどうかにかかわらず、*expression2* の値が *expression1* より大きくなければならないことに注意してください。

expression1
ループにおける整数変数の範囲の初期値を指定します。 REVERSE キーワードを指定する場合、*expression1* には、ループにおける整数変数の範囲の終了値を指定します。

expression2
ループにおける整数変数の範囲の終了値を指定します。 REVERSE キーワードを指定する場合、*expression2* には、ループにおける整数変数の範囲の初期値を指定します。

statements
ループが処理されるたびに実行される PL/SQL および SQL ステートメントを指定します。

例

以下の例は、無名ブロック内の基本的な FOR ステートメントを示しています。

```
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

この例によって生成される出力を以下に示します。

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

以下の例に示されているように、開始値が終了値より大きい場合は、ループ本体はまったく実行されませんが、エラーは返されません。

```
BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

この例では、ループ本体が実行されることはないので、出力は生成されません。

以下の例では、REVERSE キーワードを使用します。

```
BEGIN
  FOR i IN REVERSE 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

この例によって生成される出力を以下に示します。

```
Iteration # 10
Iteration # 9
Iteration # 8
Iteration # 7
Iteration # 6
Iteration # 5
Iteration # 4
Iteration # 3
Iteration # 2
Iteration # 1
```

FORALL ステートメント (PL/SQL)

FORALL ステートメントでは、配列のすべての要素または配列の要素の範囲に対して、データ変更ステートメントを実行します。

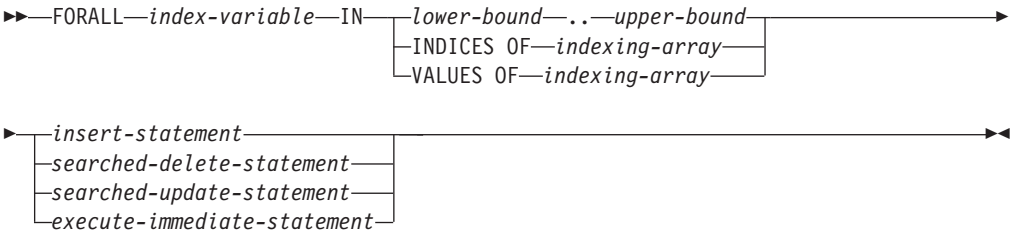
呼び出し方法

このステートメントは、PL/SQL ブロック内のみで指定できます。

許可

ステートメントの許可 ID によって保持される特権には、FORALL ステートメントで指定されるデータ変更ステートメントの呼び出しに必要なすべての特権が含まれていなければなりません。

構文



説明

index-variable
添字として使用される名前を識別します。これは、INTEGER として暗黙的に宣言され、FORALL ステートメントで参照のみ可能です。

lower-bound **..** *upper-bound*
index-variable に割り当て可能な索引値の範囲を識別します。*lower-bound* は *upper-bound* より小さくなります。この範囲は、*lower-bound* から 1 刻みで増分した *upper-bound* まで (upper-bound も含む) のすべての整数値を表します。

INDICES OF *indexing-array*
indexing–array によって示される配列の添字値を識別します。 *indexing-array* が連想配列の場合、添字値は *index-variable* に割り当て可能でなければなりませんが、疎の集合であっても構いません。

VALUES OF *indexing-array*
indexing–array によって示される配列の一連のエレメント値を識別します。エレメント値は *index-variable* に割り当て可能でなければなりませんが、順序付けられていない疎の集合であっても構いません。

insert-statement
それぞれの *index-variable* の値に対して効率的に実行する INSERT ステートメントを 1 つ指定します。

searched-delete-statement
それぞれの *index-variable* の値に対して効率的に実行する探索済み DELETE ステートメントを 1 つ指定します。

searched-update-statement
それぞれの *index-variable* の値に対して効率的に実行する探索済み UPDATE ステートメントを 1 つ指定します。

execute-immediate-statement
それぞれの *index-variable* の値に対して効率的に実行する EXECUTE IMMEDIATE ステートメントを 1 つ指定します。

注

- FORALL ステートメント処理は、アトミックではありません。FORALL ステートメントで反復処理が行われている間にエラーが発生した場合、既に処理されたどんなデータ変更操作も暗黙的にはロールバックされません。 FORALL ステートメントでエラーが発生した時には、アプリケーションで ROLLBACK ステートメントを使用し、トランザクション全体をロールバックすることができます。

例

以下は、基本的な FORALL ステートメントの例です。

```
FORALL x
  IN in_customer_list.FIRST..in_customer_list.LAST
  DELETE FROM customer
    WHERE cust_id IN in_customer_list(x);
```

EXIT ステートメント (PL/SQL)

EXIT ステートメントは、PL/SQL コード・ブロック内のループの実行を終了します。

呼び出し方法

このステートメントは、FOR、LOOP、または WHILE ステートメント内、または PL/SQL プロシージャー、関数、または無名ブロック・ステートメント内に組み込むことができます。

許可

EXIT ステートメントの呼び出しに特権は必要ありません。ただし、ステートメントの許可 ID は、FOR、LOOP、または WHILE ステートメント内に組み込んだ SQL ステートメントの呼び出しに必要な特権を保持していなければなりません。

構文

例

以下の例は、無名ブロック内の、EXIT ステートメントを伴う基本的な LOOP ステートメントを示しています。

```
DECLARE
  sum PLS_INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

LOOP ステートメント (PL/SQL)

LOOP ステートメントは、PL/SQL コード・ブロック内のステートメントのシーケンスを複数回実行します。

呼び出し方法

このステートメントは、PL/SQL プロシージャー、関数、または無名ブロック・ステートメント内に組み込むことができます。

許可

LOOP ステートメントの呼び出しに特権は必要ありません。ただし、ステートメントの許可 ID は、LOOP ステートメント内に組み込んだ SQL ステートメントの呼び出しに必要な特権を保持していなければなりません。

構文

説明

statements
1 つ以上の PL/SQL または SQL ステートメントを指定します。これらのステートメントは、ループを反復するたびに実行されます。

例

以下の例は、無名ブロック内の基本的な LOOP ステートメントを示しています。

```
DECLARE
  sum INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

WHILE ステートメント (PL/SQL)

WHILE ステートメントは、指定された式が真である限り、一連の SQL ステートメントを繰り返します。条件は、ループ本体に入る直前にその都度評価されます。

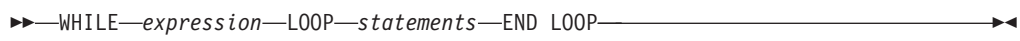
呼び出し方法

このステートメントは、PL/SQL プロシージャー、関数、または無名ブロック・ステートメント内に組み込むことができます。

許可

WHILE ステートメントを呼び出すために特権は必要ありません。ただし、ステートメントの許可 ID は、WHILE ステートメント内に組み込んだ SQL ステートメントの呼び出しに必要な特権を保持していなければなりません。

構文



説明

expression
ループを実行するかどうかを判別するために、ループ本体に入る直前にその都度評価される式を指定します。式が論理的に真の場合は、ループが実行されます。式が論理的に偽の場合は、ループ処理は終了します。 EXIT ステートメントを使用すると、式が真の場合でもループを終了させることができます。

statements
ループが処理されるたびに実行される PL/SQL および SQL ステートメントを指定します。

例

以下の例は、無名ブロック内の基本的な WHILE ステートメントを示しています。

```
DECLARE
  sum INTEGER := 0;
BEGIN
  WHILE sum < 11 LOOP
    sum := sum + 1;
  END LOOP;
END
```

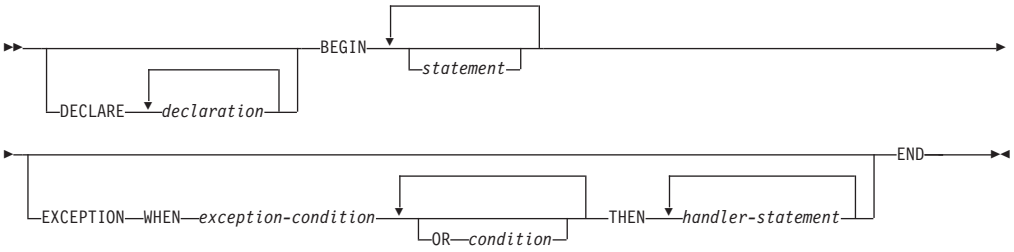
この無名ブロック内の WHILE ステートメントは、*sum* が 11 に等しくなるまで実行されてからループ処理が終了します。その後、無名ブロックの処理が完了します。

例外処理 (PL/SQL)

デフォルトでは、PL/SQL プログラム内でエラーが発生すると、プログラムの実行が停止します。 EXCEPTION セクションを使用することにより、エラーをトラップし、そこから回復することができます。

例外ハンドラーの構文は、BEGIN ブロックの構文を拡張したものです。

構文



エラーが発生しなかった場合、ブロックでは単に *statement* が実行され、END 後のステートメントに制御が渡されます。しかし、*statement* の実行中にエラーが発生すると、*statement* の残りの処理は中止され、EXCEPTION リストに制御が渡されます。発生したエラーと一致する最初の *condition* を見つけるために、リストが検索されます。一致するものが見つかると、対応する *handler-statement* が実行され、END 後のステートメントに制御が渡されます。一致するものが見つからないと、プログラムは実行を停止します。

handler-statement の実行中に新たなエラーが発生した場合は、周囲の EXCEPTION 節によってのみエラーをキャッチできます。

表 8は、使用可能な、システム定義条件を要約したものです。特別の条件名 OTHERS は、すべてのエラー・タイプと一致します。条件名には、大文字と小文字の区別がありません。

表 8. システム定義の例外条件名

条件名	説明
CASE_NOT_FOUND	CASE ステートメント内のいずれのケースも「TRUE」と評価されず、かつ ELSE 条件がありません。

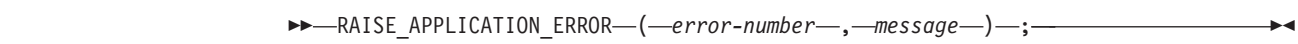
表 8. システム定義の例外条件名 (続き)

条件名	説明
CURSOR_ALREADY_OPEN	既にオープン済みのカーソルをオープンしようとしました。
DUP_VAL_ON_INDEX	索引キーに重複した値があります。
INVALID_CURSOR	オープンしていないカーソルにアクセスしようとしました。
INVALID_NUMBER	数値が無効です。
LOGIN_DENIED	ユーザー名またはパスワードが無効です。
NO_DATA_FOUND	選択基準を満たす行がありませんでした。
NOT_LOGGED_ON	データベース接続が存在しません。
OTHERS	例外セクション内の先行する条件によってキャッチされなかった例外のすべてが対象です。
SUBSCRIPT_BEYOND_COUNT	添字が、範囲外であるか、または存在しません。
SUBSCRIPT_OUTSIDE_LIMIT	添字式のデータ・タイプを添字タイプに割り当てることができません。
TOO_MANY_ROWS	複数の行が選択基準を満たしましたが、返すことが許可されているのは 1 行のみです。
VALUE_ERROR	値が無効です。
ZERO_DIVIDE	ゼロによる除算が試行されました。

アプリケーション・エラー生成 (PL/SQL)

RAISE_APPLICATION_ERROR プロシージャーにより、ユーザー定義のコードおよびエラー・メッセージをプログラムで使用できるようになります。これらを利用して、例外を特定できます。このプロシージャーは、PL/SQL コンテキスト内でのみサポートされます。

構文



説明

error-number

ベンダー固有の番号 (リテラルで表記) です。これは、DB2 エラー・コードにマップされてから、SQLCODE という変数に格納されます。

RAISE_APPLICATION_ERROR プロシージャーは、ユーザー定義の *error-number* 値として -20000 から -20999 までを受け入れます。エラー・メッセージで返される SQLCODE は、SQL0438N となります。 SQLSTATE は、クラス「UD」に加えて、*error-number* 値の末尾 3 桁に相当する 3 文字で構成されます。

message

ユーザー定義のメッセージで、最大長は 70 バイトです。

例

以下の例では、RAISE_APPLICATION_ERROR プロシージャーを使用して、欠落した従業員情報に固有のエラー・コードおよびエラー・メッセージを表示します。

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;

CALL verify_emp(7839);

SQLCODE: -438
SQLERRM: SQL0438N  Application raised error or warning with
 diagnostic text: "No manager for 7839".  SQLSTATE=UD030
```

RAISE ステートメント (PL/SQL)

RAISE ステートメントは、以前に定義された条件を発生させます。

構文

説明

condition
以前に定義された条件を指定します。

例

以下の例は、定義済みの条件を発生させるプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE raise_demo (inval NUMBER) IS
  evenno EXCEPTION;
  oddno  EXCEPTION;
```

```
BEGIN
  IF MOD(inval, 2) = 1 THEN
    RAISE oddno;
  ELSE
    RAISE evenno;
  END IF;
EXCEPTION
  WHEN evenno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is even');
  WHEN oddno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is odd');
END raise_demo;
/

SET SERVEROUTPUT ON;

CALL raise_demo;
```

Oracle-DB2 エラー・マッピング (PL/SQL)

PL/SQL のエラー・コードと例外名には、対応する DB2 エラー・コードと SQLSTATE 値があります。

これらのエラー・コード、例外名、および SQLSTATE 値を、表 9 にまとめています。

表 9. PL/SQL エラー・コードおよび例外名と DB2 エラー・コードおよび SQLSTATE 値のマッピング

plsqlCode	plsqlName	db2Code	db2State
-1	DUP_VAL_ON_INDEX	-803	23505
+100	NO_DATA_FOUND	+100	02000
-1012	NOT_LOGGED_ON	-1024	08003
-1017	LOGIN_DENIED	-30082	08001
-1476	ZERO_DIVIDE	-801	22012
-1722	INVALID_NUMBER	-420	22018
-1001	INVALID_CURSOR	-501	24501
-1422	TOO_MANY_ROWS	-811	21000
-6502	VALUE_ERROR	-433	22001
-6511	CURSOR_ALREADY_OPEN	-502	24502
-6532	SUBSCRIPT_OUTSIDE_LIMIT	-20438	428H1
-6533	SUBSCRIPT_BEYOND_COUNT	-20439	2202E
-6592	CASE_NOT_FOUND	-773	20000
-54		-904	57011
-60		-911	40001
-310		-206	42703
-595		-390	42887
-597		-303	42806
-598		-407	23502
-600		-30071	58015
-603		-119	42803
-604		-119	42803

表 9. *PL/SQL* エラー・コードおよび例外名と *DB2* エラー・コードおよび *SQLSTATE* 値のマッピング *(*続き*)*

plsqlCode	plsqlName	db2Code	db2State
-610		-20500	428HR
-611		-117	42802
-612		-117	42802
-613		-811	21000
-615		-420	22018
-616		-420	22018
-617		-418	42610
-618		-420	22018
-619		-418	42610
-620		-171	42815
-622		-304	22003
-623		-604	42611
-904		-206	42703
-911		-7	42601
-942		-204	42704
-955		-601	42710
-996		-1022	57011
-1119		-292	57047
-1002		+231	02000
-1403		-100	02000
-1430		-612	42711
-1436		-20451	560CO
-1438		-413	22003
-1450		-614	54008
-1578		-1007	58034
-2112		-811	21000
-2261		+605	01550
-2291		-530	23503
-2292		-532	23001
-3113		-30081	08001
-3114		-1024	08003
-3214		-20170	57059
-3297		-20170	57059
-4061		-727	56098
-4063		-727	56098
-4091		-723	09000
-6502		-304	22003
-6508		-440	42884
-6550		-104	42601

表 9. PL/SQL エラー・コードおよび例外名と DB2 エラー・コードおよび SQLSTATE 値のマッピング (続き)

plsqlCode	plsqlName	db2Code	db2State
-6553		-104	42601
-14028		-538	42830
-19567		-1523	55039
-30006		-904	57011
-30041		-1139	54047

カーソル (PL/SQL)

カーソル とは、結果セットにおける 1 行のデータを指し示したり選択したりするためにアプリケーション・プログラムで使用する、名前付き制御構造のことです。照会を一度にすべて実行する代わりに、カーソルを使用して、一度に 1 行ずつ照会結果セットの読み取りおよび処理を行うことができます。

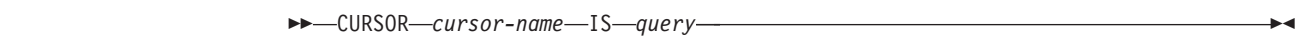
PL/SQL コンテキスト内のカーソルは、WITH HOLD カーソルとして扱われます。WITH HOLD カーソルについて詳しくは、『DECLARE CURSOR ステートメント』を参照してください。

DB2 データ・サーバーでは、PL/SQL 静的カーソルおよびカーソル変数の両方がサポートされます。

静的カーソル (PL/SQL)

静的カーソル とは、関連付けられた照会がコンパイル時に固定されるカーソルのことです。カーソルの宣言は、カーソルを使用する上での前提条件となります。PL/SQL コンテキスト内で PL/SQL 構文を使用して静的カーソルを宣言する機能が DB2 サーバーでサポートされています。

構文



説明

cursor-name
カーソルおよびその結果セットを参照するために使用できるカーソルの ID を指定します。

query
カーソルの結果セットを特定する SELECT ステートメントを指定します。

例

以下の例は、複数の静的カーソル宣言が含まれるプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     CURSOR emp_cur_1 IS SELECT * FROM emp;

     CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
```

```
     CURSOR emp_cur_3 IS SELECT empno, ename
                         FROM emp
                         WHERE deptno = 10
                         ORDER BY empno;
BEGIN
    OPEN emp_cur_1;
        ...
END;
```

パラメーター付きカーソル (PL/SQL)

パラメーター付きカーソルとは、オープン時に、渡されるパラメーター値を受け入れることができる静的カーソルのことです。

以下の例には、パラメーター付きカーソルが含まれています。このカーソルでは、EMP 表内における、渡されるパラメーター値で指定される値よりも給料が少ない各従業員の名前および給料が表示されます。

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

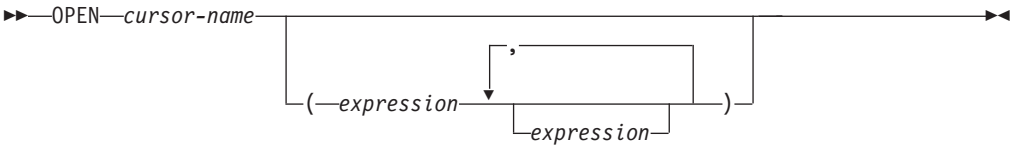
max_wage の値として 2000 が渡されると、給料が 2000 未満の従業員の名前および給料データのみが返されます。

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

カーソルのオープン (PL/SQL)

カーソルがオープンされるまでは、そのカーソルに関連付けられた結果セットを参照できません。

構文



説明

cursor-name
PL/SQL コンテキスト内で既に宣言されているカーソルの ID を指定します。オープン済みカーソルは指定できません。

expression
cursor-name がパラメーター付きカーソルである場合、オプションの実パラメーターを 1 つ以上指定します。実パラメーターの数は、対応する仮パラメーターの数と一致している必要があります。

例

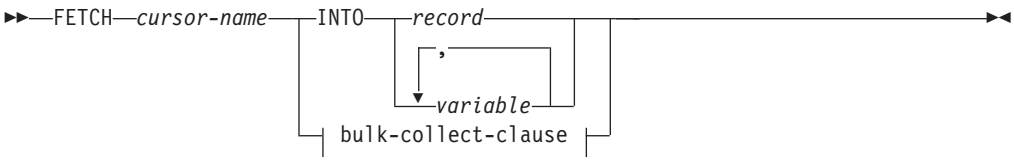
以下の例は、CURSOR_EXAMPLE プロシージャーの一部であるカーソルに対する OPEN ステートメントを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename
                        FROM emp
                        WHERE deptno = 10
                        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

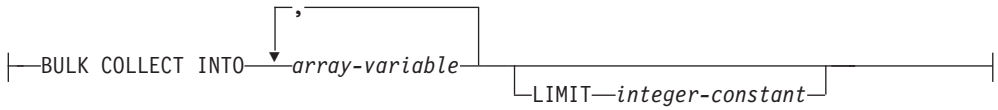
カーソルからの行のフェッチ (PL/SQL)

PL/SQL カーソルから行をフェッチするのに必要な FETCH ステートメントが、PL/SQL コンテキスト内で DB2 データ・サーバーによってサポートされています。

構文



bulk-collect-clause:



説明

cursor-name
静的カーソルまたはカーソル変数の名前。

record
既に定義されているレコードの ID。これは、ユーザー定義のレコードでも、%ROWTYPE 属性を使用して表から得られたレコード定義でも構いません。

variable
フェッチされた行からのフィールド・データを保持する PL/SQL 変数。変数は 1 つ以上定義できますが、順序および数において、カーソル宣言内で指定した照会の選択リストに返されるフィールドと一致している必要があります。選択リスト内の各フィールドのデータ・タイプは、レコード内の各フィールドのデータ・タイプ、または各変数のデータ・タイプと一致しているか、あるいはそれらに暗黙的に変換可能でなければなりません。

変数データ・タイプを、明示的に、または %TYPE 属性を使用して定義できます。

BULK COLLECT INTO *array-variable*
1 つ以上の変数を 1 つの配列データ・タイプとみなします。添字を順に割り当てながら、結果セットの順序でそれぞれの配列の 1 つの要素へ結果の各行を割り当てます。

- 1 つの *array-variable* のみを指定した場合。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプではない場合、カーソルの結果行には必ず 1 つの列が必要となり、列データ・タイプは配列エレメントのデータ・タイプに割り当て可能でなければなりません。
  - *array-variable* エレメントのデータ・タイプがレコード・タイプの場合、カーソルの結果行はレコード・タイプに割り当て可能でなければなりません。
- 複数の配列変数を指定した場合。
  - *array-variable* エレメントのデータ・タイプは、レコード・タイプであってはなりません。
  - カーソルの結果行の各列に *array-variable* が必要になります。
  - カーソルの結果行の各列のデータ・タイプは、対応する *array-variable* の配列エレメントのデータ・タイプに割り当て可能でなければなりません。

array-variable のデータ・タイプが通常配列の場合、最大カーディナリティーは照会によって戻される行数以上、または LIMIT 節で指定した *integer-constant* 以上である必要があります。

LIMIT *integer-constant*
ターゲット配列に格納される行数の限度を指定します。カーソル位置は、*integer-constant* 行順方向へ、または結果セットの末尾へ移動します。

例

以下の例は、FETCH ステートメントが含まれるプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

%TYPE 属性を使用してターゲット変数のデータ・タイプを定義した場合は、データベースの列のデータ・タイプを変更しても、PL/SQL アプリケーション・プログラム内のターゲット変数宣言を変更する必要はありません。以下の例は、%TYPE 属性を使用して定義した変数が含まれるプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

表内のすべての列をその定義順に取得する場合、%ROWTYPE 属性を使用して、FETCH ステートメントによって取得データが置かれるレコードを定義できます。その後レコード内の各フィールドには、ドット表記を使用してアクセスできます。以下の例は、%ROWTYPE を使用するレコード定義が含まれるプロシージャーを示しています。このレコードは、FETCH ステートメントのターゲットとして使用されています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     v_emp_rec       emp%ROWTYPE;
     CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
     OPEN emp_cur_1;
     FETCH emp_cur_1 INTO v_emp_rec;
     DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
     DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
END;
```

カーソルのクローズ (PL/SQL)

カーソルに関連付けられた結果セットからすべての行を取り出した後、カーソルをクローズする必要があります。カーソルのクローズ後は、結果セットを参照することはできません。

しかし、カーソルを再オープンし、新規結果セットの行をフェッチすることができます。

構文

説明

cursor-name
PL/SQL コンテキスト内で既に宣言されているオープン・カーソルの ID を指定します。

例

以下の例は、CURSOR_EXAMPLE プロシージャーの一部であるカーソルに対する CLOSE ステートメントを示しています。

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

カーソルと共に %ROWTYPE を使用する (PL/SQL)

%ROWTYPE 属性を使用すると、カーソルまたはカーソル変数からフェッチされた列のすべてに対応するフィールドを持つレコードを定義できます。各フィールドのデータ・タイプは、それに対応する列のデータ・タイプになります。

%ROWTYPE 属性には、接頭辞としてカーソル名またはカーソル変数名が付きます。構文は record cursor%ROWTYPE となります。ここで、*record* はレコードに割り当てられる ID です。また *cursor* は、現在のスコープ内で明示的に宣言されたカーソルです。

以下の例では、%ROWTYPE 属性と共にカーソルを使用して、EMP 表内の各従業員に関する部門情報を取得する方法を示します。

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

このプロシージャーの呼び出し (CALL emp_info;) により返される出力例を以下に示します。

```
SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

カーソル属性 (PL/SQL)

各カーソルには一式の属性が付与されています。これらの属性により、アプリケーション・プログラムはカーソルの状態を検査できます。

これらの属性は、%ISOPEN、%FOUND、%NOTFOUND、および %ROWCOUNT です。

%ISOPEN

この属性を使用すると、カーソルがオープン状態であるかどうかを判別できます。カーソルをパラメーターとして関数またはプロシージャーに渡す際、(カーソルのオープンを試行する前に) カーソルが既にオープン済みであるかどうかを把握しておくのは有益です。

%FOUND

この属性を使用すると、FETCH ステートメントの実行後に、カーソルに行が含まれるかどうかを判別できます。 FETCH ステートメントの実行に成功した場合、%FOUND 属性の値は TRUE です。 FETCH ステートメントの実行に失敗した場合、%FOUND 属性の値は FALSE です。以下の場合に、結果は不明となります。

- *cursor-variable-name* の値が NULL である。
- *cursor-variable-name* の事前に宣言されているカーソルがオープンしていない。
- 事前に宣言されているカーソルに対して最初の FETCH ステートメントが実行される前に、%FOUND 属性が評価される。
- FETCH ステートメントの実行により、エラーが返される。

%FOUND 属性は、フェッチする行が残っていない場合に戻るエラーの有無を検査する条件ハンドラーの使用に代わる、有効な手段です。

%NOTFOUND

この属性は、%FOUND 属性とは論理的に反対です。

%ROWCOUNT

この属性を使用すると、カーソルのオープン以降にフェッチされた行の数を特定できます。

表 10は、特定のカーソル・イベントに関連付けられた属性値の要約です。

表 10. カーソル属性値の要約

カーソル属性	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
OPEN 前	FALSE	未定義	未定義	「Cursor not open」例外
OPEN 後、最初の FETCH 前	TRUE	未定義	未定義	0
最初に成功した FETCH 後	TRUE	TRUE	FALSE	1
n 番目に成功した FETCH 後 (最終行)	TRUE	TRUE	FALSE	*n*
n+1 番目の FETCH 後 (最終行の後)	TRUE	FALSE	TRUE	*n*
CLOSE 後	FALSE	未定義	未定義	「Cursor not open」例外

カーソル変数 (PL/SQL)

カーソル変数 とは、照会の結果セットへのポインターが含まれるカーソルのことです。結果セットは、カーソル変数を使用して OPEN FOR ステートメントを実行することによって決まります。

静的カーソルとは異なり、カーソル変数は特定の照会と関連付けられていません。異なる照会が含まれる別々の OPEN FOR ステートメントを使用して、同じカーソル変数を何回もオープンできます。新しい結果セットが毎回作成され、カーソル変数によって使用可能となります。

SYS_REFCURSOR カーソル変数 (PL/SQL)

DB2 サーバーでは、あらゆる結果セットと関連付けることが可能な、SYS_REFCURSOR 組み込みデータ・タイプのカーソル変数の宣言がサポートされます。

SYS_REFCURSOR データ・タイプは、緩やかに型付けされた REF CURSOR タイプとして認知されています。厳密に型付けされた REF CURSOR タイプのカーソル変数においては、結果セットの指定が必要となります。

構文

説明

cursor-variable-name
カーソル変数の ID を指定します。

SYS_REFCURSOR
カーソル変数のデータ・タイプを、システム定義の SYS_REFCURSOR データ・タイプに指定します。

例

以下は、SYS_REFCURSOR 変数宣言の例です。

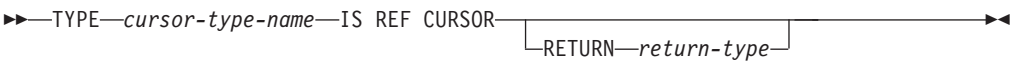
```
DECLARE emprefcur SYS_REFCURSOR;
```

ユーザー定義 REF CURSOR タイプ変数 (PL/SQL)

DB2 サーバーでは、ユーザー定義の REF CURSOR データ・タイプおよびカーソル変数の宣言がサポートされています。

PL/SQL コンテキスト内で TYPE 宣言を実行することにより、ユーザー定義 REF CURSOR タイプを定義できます。タイプの定義後に、このタイプのカーソル変数を宣言できます。

構文



説明

TYPE *cursor-type-name*
カーソル・データ・タイプの ID を指定します。

IS REF CURSOR
カーソルをユーザー定義 REF CURSOR データ・タイプに指定します。

RETURN *return-type*
カーソルに関連付ける戻りタイプを指定します。 *return-type* を指定すると、この REF CURSOR タイプは厳密に型付けされます。指定しないと、緩やかに型付けされます。

例

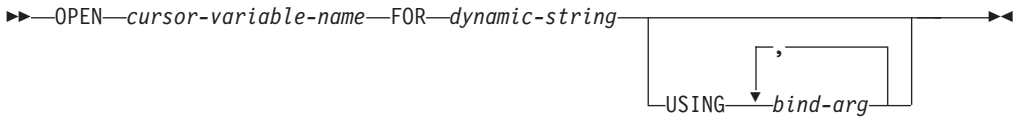
以下の例では、無名ブロックの DECLARE セクションにおけるカーソル変数宣言を示します。

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
BEGIN
    ...
END
```

カーソル変数による動的照会 (PL/SQL)

DB2 データ・サーバーでは、PL/SQL コンテキストにおける、OPEN FOR ステートメントによる動的照会がサポートされています。

構文



説明

OPEN *cursor-variable-name*
PL/SQL コンテキスト内で既に宣言されているカーソル変数の ID を指定します。

FOR *dynamic-string*
SELECT ステートメントが含まれるストリング・リテラルまたはストリング変数を指定します (終了のセミコロンは付けません)。ステートメントには、名前付きパラメーター (例えば、*:param1* など) を含めることができます。

USING *bind-arg*
1 つ以上のバインド引数を指定します。*dynamic-string* 内のプレースホルダーは、カーソルのオープン時に、バインド引数の値に置換されます。

例

以下の例は、ストリング・リテラルを使用する動的照会を示しています。

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下に、DEPT_QUERY プロシージャーによって生成される出力例を示します。

```
CALL dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

前述の例に示す照会を、バインド引数を加えて変更し、照会パラメーターを渡すようにすることができます。

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下の CALL ステートメントにより、前述の例で生成された出力と同じものが生成されます。

```
CALL dept_query(30, 1500);
```

以下のようにストリング変数を使用して SELECT ステートメントを渡すと、柔軟性が最も高くなります。

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
```

```
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

以下に、この形式の DEPT_QUERY プロシージャーによって生成される出力例を示します。

```
CALL dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

例: プロシージャーから REF CURSOR を返す (PL/SQL)

この例では、REF CURSOR 変数を定義およびオープンし、次いでプロシージャー・パラメーターとして渡す方法を示します。

次のようにカーソル変数を IN OUT パラメーターとして指定することにより、プロシージャーの呼び出し元で結果セットを使用できるようにします。

```
CREATE OR REPLACE PROCEDURE emp_by_job (
    p_job           VARCHAR2,
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
END;
```

以下の無名ブロックでは、プロシージャーの IN OUT パラメーターを、無名ブロックの宣言セクション内で宣言したカーソル変数に対して割り当てることにより、EMP_BY_JOB プロシージャーが呼び出されます。このカーソル変数を使用することにより、結果セットがフェッチされます。

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    emp_by_job(v_job, v_emp_refcur);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

以下に、無名ブロックの実行時に生成される出力例を示します。

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

例: カーソル操作のモジュール化 (PL/SQL)

以下の例では、カーソル変数に対する種々の操作を、別個のプログラムすなわち PL/SQL コンポーネントにモジュール化する方法を示します。

以下の例は、EMP 表内のすべての行を取得する照会用のカーソル変数をオープンするプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

次の例では、プロシージャーにより、指定された部門の行をすべて取得する照会用のカーソル変数をオープンします。

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

以下の例は、DEPT 表内のすべての行を取得する照会用のカーソル変数をオープンするプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_dept_refcur FOR SELECT deptno, dname FROM dept;
END;
```

次の例では、プロシージャーにより、従業員番号および従業員名で構成された、カーソル変数の結果セットをフェッチおよび表示します。

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
END;
```

以下の例は、部門番号および部門名で構成された、カーソル変数の結果セットをフェッチおよび表示するプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

以下の例は、カーソル変数をクローズするプロシージャーを示しています。

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

以下の例は、上述のプロシージャーを実行する無名ブロックを示しています。

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('***************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('***************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    open_dept(gen_refcur);
    fetch_dept(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('*****************');

    close_refcur(gen_refcur);
END;
```

以下に、無名ブロックの実行時に生成される出力例を示します。

```
ALL EMPLOYEES
EMPNO    ENAME
-----    -------
7369     SMITH
```

```
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

トリガー (PL/SQL)

PL/SQL トリガーとは、表に対する挿入、更新、または削除操作に応じて実行される一連のアクションがカプセル化および定義された、名前付きのデータベース・オブジェクトのことです。トリガーを作成するには、PL/SQL CREATE TRIGGER ステートメントを使用します。

トリガーのタイプ (PL/SQL)

DB2 データ・サーバーでは、PL/SQL コンテキスト内において行トリガーがサポートされます。

行トリガー は、トリガー・イベントの影響を受ける各行につき、1 回作動します。例えば、特定の表のトリガー・イベントとして削除操作を定義した場合に、単一の DELETE ステートメントによって 5 つの行をその表から削除すると、トリガーは 5 回、すなわち各行につき 1 回作動します。

トリガー・コード・ブロックは、トリガー・ステートメントによって各行が影響を受ける前または後に実行されます。

トリガー変数 (PL/SQL)

NEW および OLD は、明示的に定義せずに PL/SQL トリガーと共に使用できる、特殊変数です。

- NEW は、行トリガーでの挿入および更新操作における、表の新規行を表す疑似レコード名です。使用するには、`:NEW.column` とします。ここで *column* は、トリガーが定義された表の列の名前です。

– *BEFORE 行トリガー* で使用した場合、:NEW.*column* の初期の内容は、挿入される新規行内、または元の行を置き換える行内にある列値となります。
– *AFTER 行トリガー* で使用した場合、新しい列値は既に表に格納されています。

トリガー・コード・ブロックでは、:NEW.*column* を他の変数と同様に使用できます。 BEFORE 行トリガーのコード・ブロックで :NEW.*column* に値が割り当てられた場合、割り当てられた値は、挿入または更新された行内で使用されます。

- OLD は、行トリガーでの更新および削除操作における、表の元の行を表す疑似レコード名です。使用するには、`:OLD.column` とします。ここで *column* は、トリガーが定義された表の列の名前です。
  – *BEFORE 行トリガー* で使用した場合、:OLD.*column* の初期の内容は、削除される行内、または新規行によって置き換えられる元の行内にある列値となります。
  – *AFTER 行トリガー* で使用した場合、元の列値はもはや表に格納されていません。

  トリガー・コード・ブロックでは、:OLD.*column* を他の変数と同様に使用できます。 BEFORE 行トリガーのコード・ブロックで :OLD.*column* に値が割り当てられた場合、割り当てられた値は、トリガーのアクションに何も影響を及ぼしません。

トランザクションおよび例外 (PL/SQL)

トリガーは必ず、同じトランザクション (この中でトリガー・ステートメントが実行される) の一部として実行されます。

トリガー・コード・ブロック内で例外が発生しない場合、トリガー内のデータ操作言語 (DML) の結果は、トリガー・ステートメントが含まれるトランザクションがコミットされる場合にのみコミットされます。トランザクションがロールバックされると、トリガー内の DML の結果もロールバックされます。

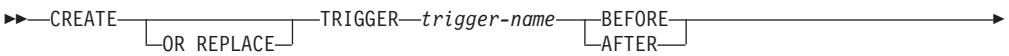
DB2 のロールバックは、アトミック・ブロック内において、または UNDO ハンドラーを使用することによってのみ行うことができます。トリガー・ステートメント自体は、それをカプセル化しているトランザクションのロールバックをアプリケーションが強制しない限り、ロールバックされません。

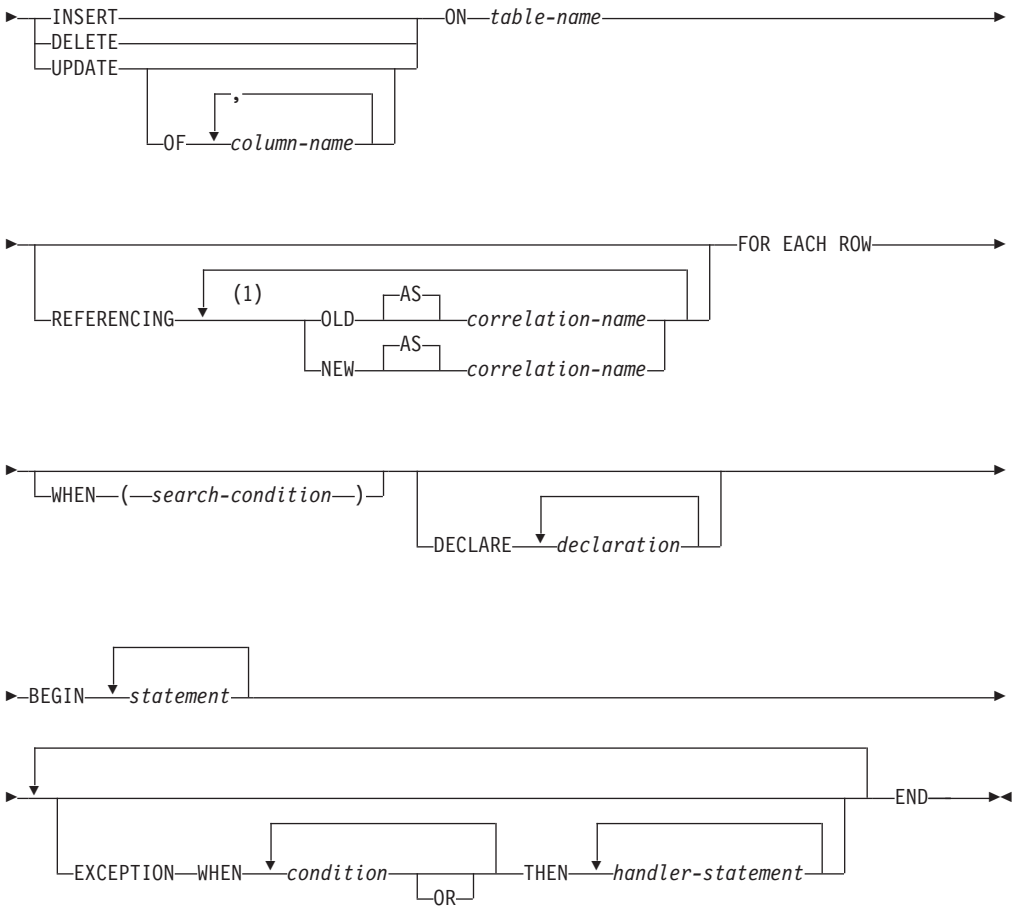
処理できない例外がトリガー・コード・ブロック内で発生すると、呼び出しステートメントがロールバックされます。

CREATE TRIGGER ステートメント (PL/SQL)

CREATE TRIGGER ステートメントは、データベース内で PL/SQL トリガーを定義します。

構文





注:

1 OLD および NEW はそれぞれ 1 回しか指定できません。

説明

OR REPLACE

トリガーの定義が現行のサーバーにある場合にそれを置き換えることを指定します。既存の定義が実際にドロップされてから、カタログ内で新しい定義が置き換えられます。トリガーの定義が現行のサーバーにない場合は、このオプションは無視されます。

trigger-name

トリガーの名前を指定します。この名前 (暗黙または明示のスキーマ名を含む) は、カタログ内で既に記述されているトリガーを特定するものであってはなりません (SQLSTATE 42710)。 2 部構成の名前を指定する場合は、スキーマ名の先頭を「SYS」にすることはできません (SQLSTATE 42939)。

BEFORE

サブジェクト表の実際の更新による変更がデータベースに適用される前に、関連したトリガー・アクションが適用されることを指定します。また、トリガーのトリガー・アクションにより他のトリガーがアクティブにならないことも指定します。

AFTER
サブジェクト表の実際の更新による変更がデータベースに適用された後に、関連したトリガー・アクションが適用されることを指定します。

INSERT
INSERT 操作がサブジェクト表に適用される際には必ず、トリガーに関連したトリガー・アクションが実行されることを指定します。

DELETE
DELETE 操作がサブジェクト表に適用される際には必ず、トリガーに関連したトリガー・アクションが実行されることを指定します。

UPDATE
UPDATE 操作がサブジェクト表に適用される際には必ず、トリガーに関連したトリガー・アクションが実行されることを指定します。この動作は、列が指定されているか暗黙指定されているかによって異なります。

オプションの *column-name* リストが指定されていない場合、表の列はすべて暗黙指定されます。したがって、*column-name* リストを省略すると、表の任意列の更新によってトリガーがアクティブになるよう暗黙指定されます。

OF *column-name*,...
それぞれの *column-name* には、基本表の列を指定する必要があります (SQLSTATE 42703)。トリガーが BEFORE トリガーの場合は、*column-name* に ID 列以外の生成された列を指定することはできません (SQLSTATE 42989)。 *column-name* のリストでは、同じ *column-name* を複数回使用することはできません (SQLSTATE 42711)。トリガーがアクティブになるのは、*column-name* のリストで指定した列の更新時のみです。

ON *table-name*
BEFORE トリガーまたは AFTER トリガーの定義のサブジェクト表を指定します。この名前は、基本表か、基本表に解決される別名を指定するものでなければなりません (SQLSTATE 42704 または 42809)。この名前は、カタログ表 (SQLSTATE 42832)、マテリアライズ照会表 (SQLSTATE 42997)、作成済みの一時表、宣言済み一時表 (SQLSTATE 42995)、またはニックネーム (SQLSTATE 42809) を指定するものであってはなりません。

REFERENCING
遷移変数 の場合は相関名を指定し、*遷移表* の場合は表名を指定します。相関名は、トリガー SQL 操作の影響を受ける行セット内の特定の行を指定します。表名は、影響を受ける行セット全体を指定します。以下のように *correlation-names* を指定して列を修飾すると、トリガー SQL 操作の影響を受ける各行をトリガー・アクションで使用できます。

OLD AS *correlation-name*
トリガー SQL 操作の前の時点での行の状態を指定する相関名を指定します。

NEW AS *correlation-name*
トリガー SQL 操作、および既に実行された BEFORE トリガーの SET ステートメントによって変更されるときの、行の状態を指定する相関名を指定します。

FOR EACH ROW
トリガー SQL 操作の影響を受けるサブジェクト表の行ごとに、トリガー・アクションが 1 回適用されることを指定します。

WHEN

(*search-condition*)
真、偽、または不明の条件を指定します。 *search-condition* を使用すると、特定のトリガー・アクションを実行する必要があるかどうかを判別することができます。関連したアクションは、指定した検索条件が真と評価される場合のみ実行されます。

declaration
変数宣言を指定します。

statement **または** *handler-statement*
PL/SQL プログラム・ステートメントを指定します。ネストしたブロックをトリガー本体に含めることができます。

condition
NO_DATA_FOUND などの例外条件名を指定します。

例

以下の例で示す BEFORE 行トリガーでは、部門 30 に属するすべての新しい従業員の歩合を計算した後に、その従業員のレコードを EMP 表に挿入します。

```
CREATE TABLE emp (
    name            VARCHAR2(10),
    deptno          NUMBER,
    sal             NUMBER,
    comm            NUMBER
)
/

CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END
/
```

トリガーのドロップ (PL/SQL)

DROP TRIGGER ステートメントを使用して、トリガーをデータベースから除去できます。

構文

説明

trigger-name
ドロップするトリガーの名前を指定します。

例: トリガー (PL/SQL)

PL/SQL トリガー定義を、DB2 データ・サーバーでコンパイルできます。以下の例は、有効なトリガーを作成し、PL/SQL トリガーのコンパイル・エラーをトラブルシューティングするために役立ちます。

BEFORE 行トリガー

以下の例で示す BEFORE 行トリガーでは、部門 30 に属するすべての新しい従業員の歩合を計算した後に、その従業員のレコードを EMP 表に挿入します。

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

このトリガーでは、2 人の新しい従業員の歩合を計算し、これらの値を新しい従業員の行の一部として挿入します。

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);

SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN         7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN         7782 01-APR-05       4500       1800         30
```

AFTER 行トリガー

以下の例では、3 つの AFTER 行トリガーを示します。

- 新しい従業員の行が EMP 表に挿入されると、トリガーの 1 つ (EMP_INS_TRIG) により、その従業員用の新規行が JOBHIST 表に追加され、当該アクションの記述を含む行が EMPCHGLOG 表に追加されます。
- 既存の従業員の行が更新されると、2 番目のトリガー (EMP_CHG_TRIG) により、JOBHIST の最新の行における ENDDATE 列 (ヌルの ENDDATE を持つ行と想定される) が現在の日付に設定され、その従業員の新規情報が入った JOBHIST の新規行が挿入されます。さらにこのトリガーにより、当該アクションの記述を含む行が、EMPCHGLOG 表に追加されます。
- 従業員の行が EMP 表から削除されると、3 番目のトリガー (EMP_DEL_TRIG) により、当該アクションの記述を含む行が EMPCHGLOG 表に追加されます。

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_ins_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
```

```
BEGIN
    v_action := 'Added';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Updated';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    v_chgdesc := '';
    IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
        v_chgdesc := v_chgdesc || 'name, ';
    END IF;
    IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
        v_chgdesc := v_chgdesc || 'job, ';
    END IF;
    IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
        v_chgdesc := v_chgdesc || 'salary, ';
    END IF;
    IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
        v_chgdesc := v_chgdesc || 'commission, ';
    END IF;
    IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
        v_chgdesc := v_chgdesc || 'department, ';
    END IF;
    v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
    UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
        AND enddate IS NULL;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_del_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Deleted';
    v_empno := :OLD.empno;
    v_deptno := :OLD.deptno;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

以下の例では、2 つの別個の INSERT ステートメントを使用して従業員の行を 2 つ追加してから、単一の UPDATE ステートメントを使用してそれら両方の行を更新します。 JOBHIST 表には、影響を受けた行ごとにトリガーのアクションが示されます。すなわち、2 人の新しい従業員についての新規採用 (New Hire) の項目が 2 つ、および歩合変更 (Changed commission) のレコードが 2 つ示されます。また EMPCHGLOG 表には、トリガーが合計 4 回、すなわち 2 つの行に対するアクションごとに 1 回ずつ作動したことも示されます。

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

     EMPNO STARTDATE ENDDATE   JOB              SAL       COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- ------------------
      9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40 New Hire
      9003 31-MAR-05           ANALYST         5000       5500         40 Changed commission
      9004 31-MAR-05           ANALYST         4500       4950         40 Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

単一の DELETE ステートメントを使用して両方の従業員を削除した後、EMPCHGLOG 表には、トリガーが 2 回、すなわち削除した従業員ごとに 1 回作動したことが示されます。

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

パッケージ (PL/SQL)

PL/SQL パッケージ定義が DB2 データ・サーバーでサポートされています。PL/SQL パッケージとは、関数、プロシージャー、変数、カーソル、ユーザー定義タイプ、およびレコードからなる名前付き集合のことです。共通の修飾子であるパッケージ名を使用して、これらの項目を参照します。

パッケージには以下の特性があります。

- パッケージは、関連した目的を持つ関数およびプロシージャーをまとめる便利な手段となります。パッケージの関数およびプロシージャーを使用する権限は、そのパッケージ全体に対して付与された 1 つの特権によって決まります。
- パッケージ内の特定の項目をパブリックとして宣言できます。パブリックのエンティティーは可視であり、該当するパッケージに対する EXECUTE 特権を保持する他のプログラムから参照可能です。パブリックの関数およびプロシージャーの場合は、それらのシグニチャーのみが可視です。これらの関数およびプロシージ

ャーの PL/SQL コードに、他からアクセスすることはできません。したがって、こうしたパッケージを利用するアプリケーションは、シグニチャー内で得られる情報のみに依存します。

- パッケージ内の他の項目はプライベートとして宣言できます。プライベートのエンティティーを参照および使用することは、当該パッケージ内の関数およびプロシージャーからは可能ですが、外部アプリケーションからは不可能です。

パッケージ・コンポーネント (PL/SQL)

パッケージは、パッケージ仕様部およびパッケージ本体という、2 つの主要コンポーネントで構成されます。

- *パッケージ仕様部* はパブリック・インターフェースで、当該パッケージの外部から参照可能なエレメントで構成されます。パッケージ仕様部を作成するには、CREATE PACKAGE ステートメントを実行します。
- *パッケージ本体* には、パッケージ仕様部内で宣言されたすべてのプロシージャーおよび関数が実際にインプリメントされたものに加え、プライベートのタイプ、変数、およびカーソルの宣言が入ります。パッケージ本体を作成するには、CREATE PACKAGE BODY ステートメントを実行します。

パッケージの作成 (PL/SQL)

パッケージ仕様部を作成すると、関連するデータ・タイプ、プロシージャー、および関数の定義を、データベースの単一のコンテキスト内にカプセル化できます。

パッケージはスキーマを拡張するもので、参照するオブジェクトに対してネーム・スペースのサポートを提供します。これは、実行可能コードを定義できるリポジトリーです。パッケージを使用する場合には、パッケージ内にインプリメントされたオブジェクトの参照または実行がパッケージ仕様部内に定義されている必要があります。

パッケージ仕様部の作成 (PL/SQL)

パッケージ仕様部では、パッケージの外部からどのパッケージ・オブジェクトを参照できるかを設定します。パッケージの外部から参照可能なオブジェクトは、そのパッケージのパブリック・エレメントといいます。

以下の例では、2 つの関数および 2 つのストアード・プロシージャーで構成される、EMP_ADMIN という名前のパッケージ仕様部を作成する方法を示します。

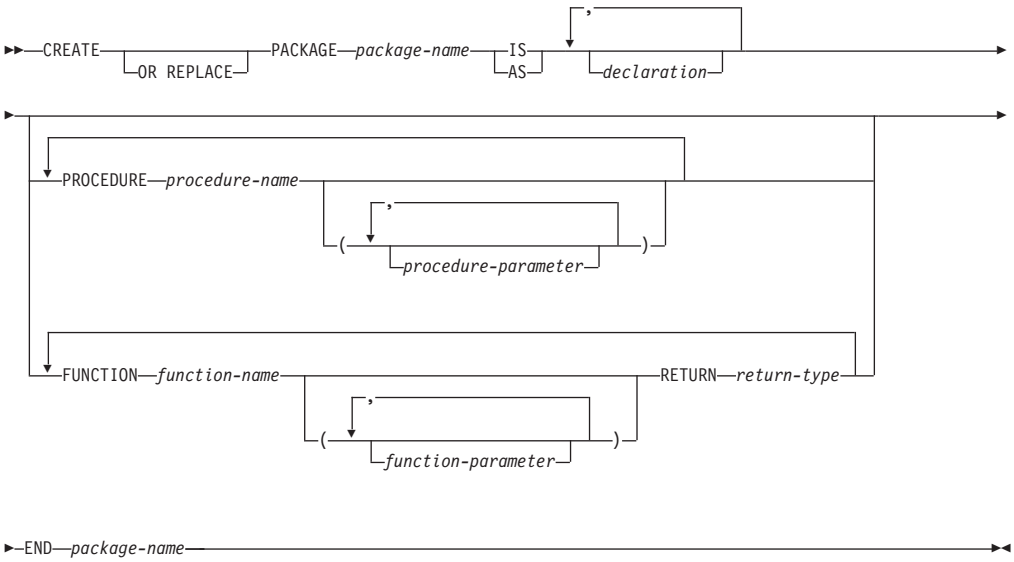
```
CREATE OR REPLACE PACKAGE emp_admin
IS

   FUNCTION get_dept_name (
      p_deptno        NUMBER DEFAULT 10
   )
   RETURN VARCHAR2;
   FUNCTION update_emp_sal (
      p_empno         NUMBER,
      p_raise         NUMBER
   )
   RETURN NUMBER;
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
```

```
      p_hiredate      DATE DEFAULT sysdate,
      p_comm          NUMBER DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER DEFAULT 10
   );
   PROCEDURE fire_emp (
      p_empno         NUMBER
   );

END emp_admin;
```

CREATE PACKAGE ステートメント (PL/SQL)

CREATE PACKAGE ステートメントにより、パッケージへのインターフェースを定義する、パッケージ仕様部が作成されます。

構文



説明

package-name
パッケージの ID を指定します。

declaration
パブリックな項目の ID を指定します。パブリックな項目には、構文 *package-name.item-name* を使用して、当該パッケージの外部からアクセス可能です。パブリックな項目は 0 個以上含めることができます。パブリックな項目の宣言は、プロシージャーまたは関数の宣言の前に配置する必要があります。*declaration* には、以下に示す項目を含めることができます。

- コレクション宣言
- EXCEPTION 宣言
- レコード宣言
- REF CURSOR およびカーソル変数宣言
- コレクション、レコード、または REF CURSOR タイプ変数の TYPE 定義

- 変数宣言

procedure-name
パブリック・プロシージャーの ID を指定します。パブリック・プロシージャーは、構文 *package-name.procedure-name*() を使用して、当該パッケージの外部から呼び出すことができます。

procedure-parameter
プロシージャーの仮パラメーターの ID を指定します。

function-name
パブリック関数の ID を指定します。パブリック関数は、構文 *package-name.function-name*() を使用して、当該パッケージの外部から呼び出すことができます。

function-parameter
関数の仮パラメーターの ID を指定します。入力 (IN モード) パラメーターを、デフォルト値で初期設定できます。

return-type
関数によって返される値のデータ・タイプを指定します。

注

CREATE PACKAGE ステートメントを難読化形式でサブミットできます。難読化されたステートメントでは、パッケージ名のみを判読できます。ステートメントの他の部分は判読できないようにエンコードされますが、データベース・サーバーはその部分をデコードできます。難読化ステートメントの作成は、DBMS_DDL.WRAP 関数を呼び出すことによって行えます。

パッケージ本体の作成 (PL/SQL)

パッケージ本体には、パッケージ仕様部内で宣言されたすべてのプロシージャーおよび関数がインプリメントされたものが入ります。

以下の例では、EMP_ADMIN パッケージ仕様部のパッケージ本体を作成する方法を示します。

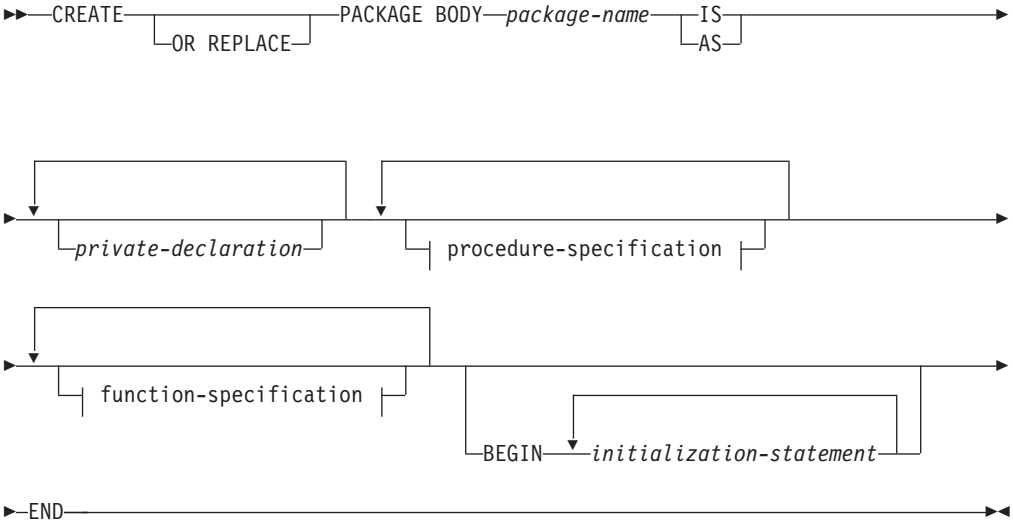
```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
   --
   --  Function that queries the 'dept' table based on the department
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
      p_deptno        IN NUMBER DEFAULT 10
   )
   RETURN VARCHAR2
   IS
      v_dname         VARCHAR2(14);
   BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
         RETURN '';
```

```
   END;
   --
   --  Function that updates an employee's salary based on the
   --  employee number and salary increment/decrement passed
   --  as IN parameters.  Upon successful completion the function
   --  returns the new updated salary.
   --
   FUNCTION update_emp_sal (
      p_empno         IN NUMBER,
      p_raise         IN NUMBER
   )
   RETURN NUMBER
   IS
      v_sal           NUMBER := 0;
   BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
         RETURN -1;
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
         DBMS_OUTPUT.PUT_LINE(SQLERRM);
         DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
         DBMS_OUTPUT.PUT_LINE(SQLCODE);
         RETURN -1;
   END;
   --
   --  Procedure that inserts a new employee record into the 'emp' table.
   --
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE    DEFAULT sysdate,
      p_comm          NUMBER  DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER  DEFAULT 10
   )
   AS
   BEGIN
      INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
         VALUES(p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
   END;
   --
   --  Procedure that deletes an employee record from the 'emp' table based
   --  on the employee number.
   --
   PROCEDURE fire_emp (
      p_empno         NUMBER
   )
   AS
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

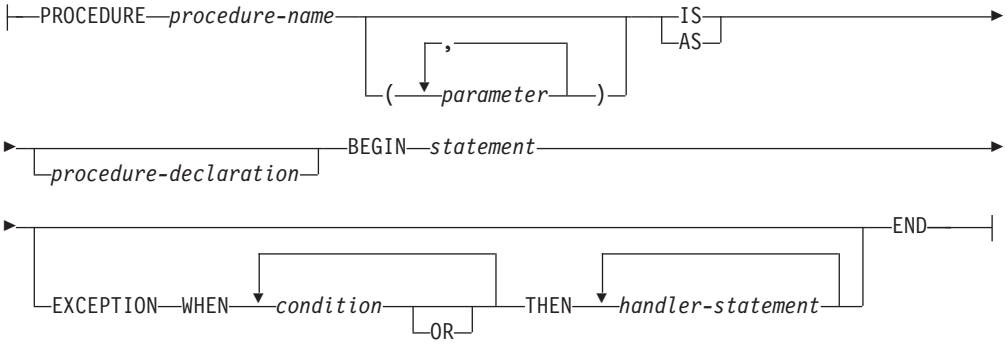
CREATE PACKAGE BODY ステートメント (PL/SQL)

CREATE PACKAGE BODY ステートメントで、パッケージ本体を作成します。パッケージ本体には、パッケージ仕様部内で宣言されたすべてのプロシージャーおよび関数がインプリメントされたものに加え、プライベートのタイプ、変数、およびカーソルの宣言が入ります。

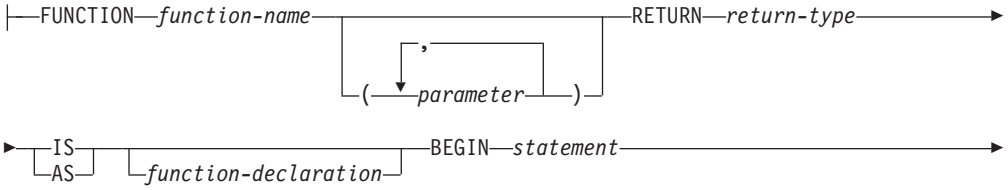
構文

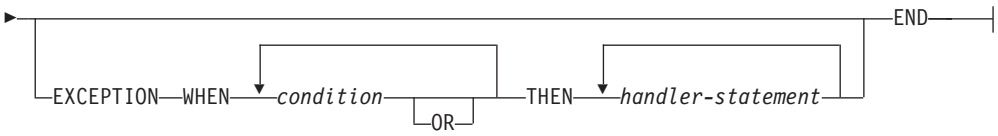


procedure-specification:



function-specification:





説明

package-name
作成する本体のパッケージの名前を指定します。同じ名前のパッケージ仕様部が存在しなければなりません。

private-declaration
パッケージ内のいずれのプロシージャーまたは関数からもアクセス可能なプライベート変数の名前を指定します。プライベート変数は 0 個以上含めることができます。 *private-declaration* には、以下に示す項目を含めることができます。

- 変数宣言
- レコード宣言
- コレクション宣言
- REF CURSOR およびカーソル変数宣言
- レコード、コレクション、または REF CURSOR タイプの変数の TYPE 定義

procedure-name
パッケージ仕様部およびそのシグニチャーで宣言されたパブリック・プロシージャーの名前を指定します。シグニチャーでは、仮パラメーターの名前、データ・タイプ、パラメーター・モード、仮パラメーターの順序、または仮パラメーターの数を指定できます。プロシージャー名およびパッケージ仕様部が、該当するパブリック・プロシージャーの宣言のシグニチャーと完全に一致する場合は、*procedure-name* によってこのパブリック・プロシージャーの本体が定義されます。

これらの条件のいずれも該当しない場合、*procedure-name* によって新規プライベート・プロシージャーが定義されます。

parameter
プロシージャーの仮パラメーターを指定します。

procedure-declaration
プロシージャー *procedure-name* 内からのみアクセス可能な宣言を指定します。これは PL/SQL ステートメントです。

statement
PL/SQL プログラム・ステートメントを指定します。

function-name
パッケージ仕様部およびそのシグニチャーで宣言されたパブリック関数の名前を指定します。シグニチャーでは、仮パラメーターの名前、データ・タイプ、パラメーター・モード、仮パラメーターの順序、または仮パラメーターの数を指定できます。関数名およびパッケージ仕様部が、該当するパブリック関数の宣言のシグニチャーと完全に一致する場合は、*function-name* によってこのパブリック関数の本体が定義されます。

これらの条件のいずれも該当しない場合、*function-name* によって新規プライベート関数が定義されます。

parameter
関数の仮パラメーターを指定します。

return-type
関数によって返される値のデータ・タイプを指定します。

function-declaration
関数 *function-name* 内からのみアクセス可能な宣言を指定します。これは PL/SQL ステートメントです。

statement
PL/SQL プログラム・ステートメントを指定します。

initialization-statement
パッケージ本体の初期設定セクション内のステートメントを指定します。初期設定セクションを指定する場合、そこには少なくとも 1 つのステートメントを含める必要があります。初期設定セクション内のステートメントは、パッケージが最初に参照されるとき、各ユーザー・セッションにつき 1 回実行されます。

注

CREATE PACKAGE BODY ステートメントを難読化形式でサブミットできます。難読化されたステートメントでは、パッケージ名のみを判読できます。ステートメントの他の部分は判読できないようにエンコードされますが、データベース・サーバーはその部分をデコードできます。難読化ステートメントの作成は、DBMS_DDL.WRAP 関数を呼び出すことによって行えます。

パッケージ・オブジェクトの参照 (PL/SQL)

パッケージ内に定義されたオブジェクトに対する参照を、パッケージ名で修飾しなければならない場合があります。

パッケージ仕様部内に宣言されたオブジェクトを参照するには、パッケージ名、ピリオド文字に続いてオブジェクトの名前を指定します。そのパッケージが現行スキーマ内に定義されていない場合は、スキーマ名も指定します。以下に例を示します。

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
schema.package_name.subprogram_name
```

例

以下の例には、EMP_ADMIN という名前のパッケージで定義された GET_DEPT_NAME という名前の関数に対する参照が含まれています。

```
select emp_admin.get_dept_name(10) from dept
```

ユーザー定義タイプを含むパッケージ (PL/SQL)

パッケージ内でユーザー定義タイプの宣言および参照が可能です。

以下の例では、EMP_RPT パッケージのパッケージ仕様部を示します。この定義には以下の宣言が含まれます。

- パブリックで使用可能なレコード・タイプである、EMPREC_TYP

- パブリックで使用可能な、緩やかに型付けされた REF CURSOR タイプである、EMP_REFCUR
- 2 つの関数、GET_DEPT_NAME および OPEN_EMP_BY_DEPT。後者の関数は、REF CURSOR タイプの EMP_REFCUR を返します
- 2 つのプロシージャー、FETCH_EMP および CLOSE_REFCUR。両者とも、緩やかに型付けされた REF CURSOR タイプを、仮パラメーターとして宣言します

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

関連付けられたパッケージ本体の定義には、以下のプライベート変数の宣言が含まれます。

- 静的カーソルである、DEPT_CUR
- 連想配列タイプである、DEPTTAB_TYP
- 連想配列変数である、T_DEPT
- 整数変数である、T_DEPT_MAX
- レコード変数である、R_EMP

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
```

```
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

このパッケージは、プライベート連想配列変数の T_DEPT、プライベート静的カーソルを使った DEPT_CUR の初期化部分を含んでいます。 T_DEPT は、関数 GET_DEPT_NAME において、部門名の参照表として機能します。 OPEN_EMP_BY_DEPT 関数は、指定した部門の従業員番号および従業員名を結果に設定した、REF CURSOR 変数を返します。その後、この REF CURSOR 変数をプロシージャー FETCH_EMP に渡すことにより、結果セットの個々の行を取り出し、リストすることができます。最後に、プロシージャー CLOSE_REFCUR を使用すると、この結果セットに関連付けられた REF CURSOR 変数をクローズできます。

以下の無名ブロックでは、パッケージ関数およびプロシージャーを実行します。宣言セクションには、パブリック REF CURSOR タイプである EMP_REFCUR を使用する、カーソル変数 V_EMP_CUR の宣言が含まれます。 V_EMP_CUR には、パッケージ関数とプロシージャーの間で渡される結果セットへのポインターが含まれます。

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
```

```
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

この無名ブロックによって生成される出力例を以下に示します。

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

以下の無名ブロックでは、同じ結果を得るための別の方法を示します。パッケージ・プロシージャー FETCH_EMP および CLOSE_REFCUR を使用する代わりに、無名ブロック内にロジックが直接コーディングされています。無名ブロックの宣言部分では、レコード変数の R_EMP、パブリックレコードタイプを使用して宣言された EMPREC_TYP の追加に注意してください。

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

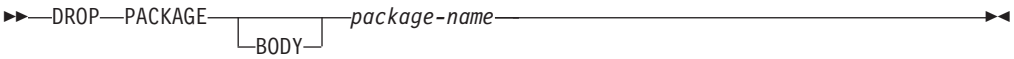
この無名ブロックによって生成される出力例を以下に示します。

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

パッケージのドロップ (PL/SQL)

パッケージが不要になったなら、ドロップできます。あるいは、パッケージを再利用する場合、パッケージ本体のみをドロップするという選択肢もあります。

構文



説明

BODY
パッケージ本体のみをドロップすることを指定します。このキーワードを省略すると、パッケージ仕様部とパッケージ本体の両方がドロップされます。

package-name
パッケージの名前を指定します。

例

以下の例では、EMP_ADMIN という名前のパッケージの本体のみをドロップする方法を示します。

```
DROP PACKAGE BODY emp_admin
```

以下の例では、パッケージの仕様と本体の両方をドロップする方法を示します。

```
DROP PACKAGE emp_admin
```

第 3 章 システム定義のモジュール

システム定義のモジュールでは、数多くの有用な操作を実行するための使いやすいプログラマチック・インターフェースが提供されます。

システム定義のモジュールを使用して、例えば次のような機能を実行できます。

- メッセージおよびアラートを、複数の接続にまたがって送受信する。
- オペレーティング・システムのファイル・システム上でファイルやディレクトリーに対して書き込み/読み取りを行う。
- さまざまなモニター情報が入ったレポートを生成する。

システム定義のモジュールは、SQL ベースのアプリケーション、DB2 コマンド行、またはコマンド・スクリプトから呼び出すことができます。

システム定義モジュールは、次の製品エディションではサポートされていません。

- DB2 Express®
- DB2 Express-C
- DB2 Personal Edition

DBMS_ALERT モジュール

DBMS_ALERT モジュールは、アラートの登録、アラートの送受信を行うための一連のプロシージャーを備えています。

アラートは、SYSTOOLS.DBMS_ALERT_INFO に保管されます。これは、データベースごとに初めてこのモジュールを参照する際に SYSTOOLSPACE に作成されます。

このモジュールのスキーマは SYSIBMADM です。

DBMS_ALERT モジュールには、以下のシステム定義ルーチンが含まれています。

表 11. DBMS_ALERT モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
REGISTER プロシージャー	現行セッションを登録して指定されたアラートを受信します。
REMOVE プロシージャー	指定されたアラートの登録を除去します。
REMOVEALL プロシージャー	すべてのアラートの登録を除去します。
SIGNAL プロシージャー	指定されたアラートのオカレンスをシグナル通知します。
SET_DEFAULTS プロシージャー	WAITONE および WAITANY プロシージャーのポーリング間隔を設定します。
WAITANY プロシージャー	任意の登録済みアラートの発生を待機します。
WAITONE プロシージャー	指定したアラートの発生を待機します。

使用上の注意

DBMS_ALERT モジュールのプロシージャーは、特定のイベント用にアラートを送信する場合に役立ちます。例えば、1 つ以上の表に対する変更の結果、トリガーが活動化されるときに、アラートを送信したい場合があります。

DBMS_ALERT モジュールでは、データベース構成パラメーター CUR_COMMIT を ON に設定する必要があります。

例

トリガー、TRIG1 が活動化される場合、接続 1 から接続 2 にアラートを送信します。まず、表およびトリガーを作成します。

```
CREATE TABLE T1 (C1 INT)@

CREATE TRIGGER TRIG1
AFTER INSERT ON T1
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN ATOMIC
CALL DBMS_ALERT.SIGNAL( 'trig1', NEW.C1 );
END@
```

接続 1 から INSERT ステートメントを発行します。

```
INSERT INTO T1 values (10)@
-- Commit to send messages to the listeners (required in early program)
CALL DBMS_ALERT.COMMIT()@
```

接続 2 から trig1 という名のアラートを受信するように登録し、アラートを待機します。

```
CALL DBMS_ALERT.REGISTER('trig1')@
CALL DBMS_ALERT.WAITONE('trig1', ?, ?, 5)@
```

この例では、以下の出力が結果として戻ります。

```
  Value of output parameters
  --------------------------
  Parameter Name  : MESSAGE
  Parameter Value : -

  Parameter Name  : STATUS
  Parameter Value : 1

  Return Status = 0
```

REGISTER プロシージャー - 指定されたアラートを受信するように登録する

REGISTER プロシージャーは、指定されたアラートを受信するように現行セッションを登録します。

構文

プロシージャー・パラメーター

name
アラートの名前を指定する、タイプ VARCHAR (128) の入力引数。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

REGISTER プロシージャーを使用して alert_test という名のアラートを登録し、シグナルを待機します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

REMOVE プロシージャー - 指定されたアラートの登録を除去する

REMOVE プロシージャーは、指定されたアラートの登録を現行セッションから除去します。

構文

プロシージャー・パラメーター

name
アラートの名前を指定する、タイプ VARCHAR (128) の入力引数。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

REMOVE プロシージャーを使用して、alert_test というアラートを除去します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

REMOVEALL プロシージャー - すべてのアラートの登録を除去する

REMOVEALL プロシージャーは、すべてのアラートの登録を現行セッションから除去します。

構文

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

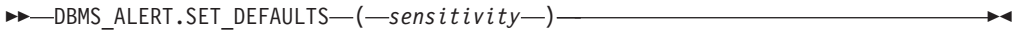
REMOVEALL プロシージャーを使用して、すべてのアラートの登録を除去します。

```
CALL DBMS_ALERT.REMOVEALL@
```

SET_DEFAULTS - WAITONE および WAITANY のポーリング間隔の設定

SET_DEFAULTS プロシージャーは、WAITONE および WAITANY プロシージャーによって使用されるポーリング間隔を設定します。

構文



プロシージャー・パラメーター

sensitivity
WAITONE および WAITANY プロシージャーがシグナルをチェックする時間間隔を秒単位で指定する、タイプ INTEGER の入力引数。値が指定されない場合には、時間間隔はデフォルトで 1 秒になります。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

SET_DEFAULTS プロシージャーを使用して、WAITONE および WAITANY プロシージャーのポーリング間隔を設定します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  DECLARE v_polling INTEGER DEFAULT 3;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.SET_DEFAULTS(v_polling);
  CALL DBMS_OUTPUT.PUT_LINE('Polling interval: ' || v_polling);
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

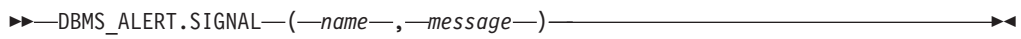
この例では、以下の出力が結果として戻ります。

```
Polling interval : 3
```

SIGNAL プロシージャー - 指定されたアラートの発生をシグナル通知する

SIGNAL プロシージャーは、指定されたアラートの発生をシグナル通知します。シグナルには、アラートとともに渡されるメッセージが含まれます。メッセージは、SIGNAL 呼び出しが発行されると、リスナー (アラート用に登録されたプロセス) に配布されます。

構文



プロシージャー・パラメーター

name
アラートの名前を指定する、タイプ VARCHAR (128) の入力引数。

message
情報をこのアラートとともに渡すように指定する、タイプ VARCHAR(32672)

の入力引数。 このメッセージは、アラートが発生すると、WAITANY または WAITONE プロシージャーによって戻すことができます。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

SIGNAL プロシージャーを使用して、alert_test というアラートの発生をシグナル通知します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_name   VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc1@
```

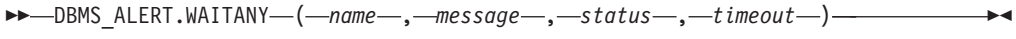
この例では、以下の出力が結果として戻ります。

```
Issued alert for alert_test
```

WAITANY プロシージャー - 任意の登録済みアラートを待機する

WAITANY プロシージャーは、任意の登録済みアラートの発生を待機します。

構文



プロシージャー・パラメーター

name
アラートの名前を含む、タイプ VARCHAR (128) の出力引数。

message
SIGNAL プロシージャーによって送信されるメッセージを含む、タイプ VARCHAR (32672) の出力引数。

status
プロシージャーによって戻される状況コードを含むタイプ INTEGER の出力引数。可能性があるのは以下の値です。

0 アラートが発生しました。

1 タイムアウトが発生しました。

timeout
アラートを待機する時間を秒単位で指定する、タイプ INTEGER の入力引数。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

1 つの接続から waitany.clp という CLP スクリプトを実行し、任意の登録済みアラートを受信します。

```
waitany.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30);
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER('alert_test');
  CALL DBMS_ALERT.REGISTER('any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITANY(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVEALL;
END@

call proc1@
```

別の接続から、signal.clp というスクリプトを実行して、any_alert という名前のアラートについてシグナルを発行します。

```
signal.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2
  BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'any_alert';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

スクリプト signal.clp では、以下の出力が結果として戻ります。

```
Issued alert for any_alert
```

スクリプト waitany.clp では、以下の出力が結果として戻ります。

```
Registered for alert alert_test and any_alert
Waiting for signal...
Alert name : any_alert
Alert msg : This is the message from any_alert
Alert status : 0
Alert timeout: 20 seconds
```

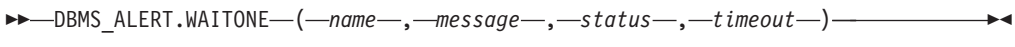
使用上の注意

WAITANY プロシージャーが呼び出された際にアラートが未登録の場合、プロシージャーは SQL0443N を戻します。

WAITONE プロシージャー - 指定されたアラートを待機する

WAITONE プロシージャーは、指定されたアラートの発生を待機します。

構文



プロシージャー・パラメーター

name
アラートの名前を指定する、タイプ VARCHAR (128) の入力引数。

message
SIGNAL プロシージャーによって送信されるメッセージを含む、タイプ VARCHAR (32672) の出力引数。

status
プロシージャーによって戻される状況コードを含むタイプ INTEGER の出力引数。可能性があるのは以下の値です。

0 アラートが発生しました。

1 タイムアウトが発生しました。

timeout
指定されたアラートを待機する時間を秒単位で指定する、タイプ INTEGER の入力引数。

許可

DBMS_ALERT モジュールに対する EXECUTE 特権。

例

waitone.clp という名前の CLP スクリプトを実行して、alert_test という名前のアラートを受信します。

```
waitone.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

異なる接続から、signalalert.clp という名前のスクリプトを実行して、alert_test という名前のアラートについてシグナルを発行します。

```
signalalert.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2
BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

スクリプト signalalert.clp では、以下の出力が結果として戻ります。

```
Issued alert for alert_test
```

スクリプト waitone.clp では、以下の出力が結果として戻ります。

```
Waiting for signal...
Alert name : alert_test
Alert msg : This is the message from alert_test
Alert status : 0
Alert timeout: 20 seconds
```

DBMS_DDL モジュール

DBMS_DDL モジュールは、ルーチン、トリガー、ビュー、PL/SQL パッケージなどの DDL オブジェクトを難読化する機能を提供します。難読化により、プロシージャーのロジックを公開せずに SQL オブジェクトをデータベースにデプロイメントできます。

これらのオブジェクトの DDL ステートメントは、ベンダー提供のインストール・スクリプトと DB2 カタログの両方において難読化されます。

このモジュールのスキーマは SYSIBMADM です。

表 12. DBMS_DDL モジュールには、以下のルーチンが含まれています。

ルーチン名	説明
WRAP 関数	引数として渡された DDL ステートメントの難読化バージョンを生成します。
CREATE_WRAPPED プロシージャー	DDL ステートメントを難読化形式でデータベースにデプロイします。

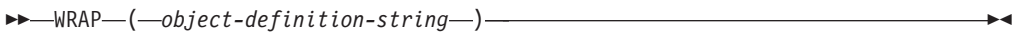
WRAP 関数 – DDL ステートメントの難読化

WRAP 関数は、判読可能な DDL ステートメントを、難読化された DDL ステートメントに変換します。

構文

難読化 DDL ステートメントでは、プロシージャー・ロジックに含まれる知的財産を簡単には抽出できないように、ロジックと組み込み SQL ステートメントがごち

ゃまぜにされます。 DDL ステートメントが外部ルーチン定義に相当する場合は、パラメーター・リストの後に続く部分がエンコードされます。



パラメーター

object-definition-string
DDL ステートメント・テキストを含んだ CLOB(2M) タイプのストリング。以下のいずれかの DDL ステートメント・テキストが可能です (SQLSTATE 5UA0O)。

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE PACKAGE (PL/SQL)
- CREATE PACKAGE BODY (PL/SQL)
- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE ADD FUNCTION
- ALTER MODULE PUBLISH FUNCTION
- ALTER MODULE ADD PROCEDURE
- ALTER MODULE PUBLISH PROCEDURE

結果は、入力ステートメントのエンコード・バージョンを含んだ CLOB(2M) タイプのストリングになります。エンコードの構成は、次のようになります。まず、元のステートメントの接頭部のルーチン・シグニチャー (トリガー、ビュー、またはパッケージの名前) までがあり、その後にキーワード WRAPPED が続きます。このキーワードの後に、この関数を実行するアプリケーション・サーバーに関する情報が続きます。この情報は *pppvvrrm* の形式をとります。

- *ppp* には、製品を DB2 Database for Linux®, UNIX®, and Windows® と特定するために文字 SQL が使用されます。
- *vv* は、2 桁のバージョン ID です (例えば「09」)。
- *rr* は、2 桁のリリース ID です (例えば「07」)。
- *m* は、1 文字の修正レベル ID です (例えば「0」)。

例えば、バージョン 9.7 のフィックスパック 2 は「SQL09072」と示されます。このアプリケーション・サーバー情報の後に、文字 (a から z、および A から Z)、数字 (0 から 9)、下線、コロンから成るストリングが続きます。難読化後も判読可能のままの接頭部の後にある入力ステートメントに対する構文検査は行われません。

エンコードされた DDL ステートメントは通常、平文形式のステートメントよりも長くなります。結果が SQL ステートメントの最大長を超えた場合は、エラーが発生します (SQLSTATE 54001)。

注: ステートメントのエンコードは、内容を難読化することを意図したものであり、強い暗号化の 1 形式と考えるべきではありません。

許可

DBMS_DDL モジュールに対する EXECUTE 特権。

例

1. 週 40 時間労働として時間給から年間給与を計算する関数の難読化バージョンを作成します。

```
VALUES(DDL.WRAP('CREATE FUNCTION ' ||
                    'salary(wage DECFLOAT) ' ||
                    'RETURNS DECFLOAT ' ||
                    'RETURN wage * 40 * 52'))
```

このステートメントの結果は、例えば次の形式のものになります。

```
CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

2. 複雑なデフォルトを設定するトリガーの難読化形式を作成します。

```
VALUES(DBMS_DDL.WRAP('CREATE OR REPLACE TRIGGER ' ||
                     'trg1 BEFORE INSERT ON emp ' ||
                     'REFERENCING NEW AS n ' ||
                     'FOR EACH ROW ' ||
                     'WHEN (n.bonus IS NULL) ' ||
                     'SET n.bonus = n.salary * .04'))
```

このステートメントの結果は、例えば次の形式のものになります。

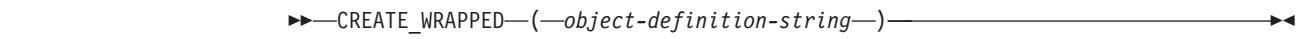
```
CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072 obfuscated-text
```

CREATE_WRAPPED プロシージャー - 難読化オブジェクトのデプロイ

CREATE_WRAPPED プロシージャーは、平文 DDL オブジェクト定義を難読化 DDL オブジェクト定義に変換した後、そのオブジェクトをデータベースにデプロイします。

構文

難読化 DDL ステートメントでは、プロシージャー・ロジックに含まれる知的財産を簡単には抽出できないように、ロジックと組み込み SQL ステートメントがエンコードされます。



パラメーター

object-definition-string
DDL ステートメント・テキストを含んだ CLOB(2M) タイプのストリング。以下のいずれかの DDL ステートメント・テキストが可能です (SQLSTATE 5UA0O)。

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE PACKAGE (PL/SQL)
- CREATE PACKAGE BODY (PL/SQL)

- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE ADD FUNCTION
- ALTER MODULE PUBLISH FUNCTION
- ALTER MODULE ADD PROCEDURE
- ALTER MODULE PUBLISH PROCEDURE

このプロシージャーは、入力を難読化 DDL ステートメント・ストリング変換した後、その DDL ステートメントを動的に実行します。現在の呼び出し側の権限だけでなく、呼び出し時に有効な特殊レジスター (PATH や CURRENT SCHEMA など) の値も使用されます。

エンコードの構成は、次のようになります。まず、元のステートメントの接頭部のルーチン・シグニチャー (トリガー、ビュー、またはパッケージの名前) までがあり、その後にキーワード WRAPPED が続きます。このキーワードの後に、このプロシージャーを実行するアプリケーション・サーバーに関する情報が続きます。この情報は「*pppvvrrm*」の形式をとります。

- *ppp* には、製品を DB2 Database for Linux®, UNIX®, and Windows® と特定するために文字 SQL が使用されます。
- *vv* は、2 桁のバージョン ID です (例えば「09」)。
- *rr* は、2 桁のリリース ID です (例えば「07」)。
- *m* は、1 文字の修正レベル ID です (例えば「0」)。

例えば、バージョン 9.7 のフィックスパック 2 は「SQL09072」と示されます。このアプリケーション・サーバー情報の後に、文字 (a から z、および A から Z)、数字 (0 から 9)、下線、コロンから成るストリングが続きます。難読化後も判読可能のままの接頭部の後にある入力ステートメントに対する構文検査は行われません。

エンコードされた DDL ステートメントは通常、平文形式のステートメントよりも長くなります。結果が SQL ステートメントの最大長を超えた場合は、エラーが発生します (SQLSTATE 54001)。

注: ステートメントのエンコードは、内容を難読化することを意図したものであり、強い暗号化の 1 形式と考えるべきではありません。

許可

DBMS_DDL モジュールに対する EXECUTE 特権。

例

1. 週 40 時間労働として時間給から年間給与を計算する難読化関数を作成します。

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE FUNCTION ' ||
                  'salary(wage DECFLOAT) ' ||
                  'RETURNS DECFLOAT ' ||
                  'RETURN wage * 40 * 52');
SELECT text FROM SYSCAT.ROUTINES
 WHERE routinename = 'SALARY'
    AND routineschema = CURRENT SCHEMA;
```

この CALL ステートメントの実行が正常に完了すると、ルーチン'SALARY' に対応する行の SYSCAT.ROUTINES.TEXT 列は、次のような形式のものになります。

```
CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

2. 複雑なデフォルトを設定する難読化トリガーを作成します。

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE OR REPLACE TRIGGER ' ||
                    'trg1 BEFORE INSERT ON emp ' ||
                    'REFERENCING NEW AS n ' ||
                    'FOR EACH ROW ' ||
                    'WHEN (n.bonus IS NULL) ' ||
                    'SET n.bonus = n.salary * .04');
SELECT text FROM SYSCAT.TRIGGERS
 WHERE trigname = 'TRG1'
   AND trigschema = CURRENT SCHEMA;
```

この CALL ステートメントの実行が正常に完了すると、トリガー 'TRG1' に対応する行の SYSCAT.TRIGGERS.TEXT 列は、次のような形式のものになります。

```
CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072  obfuscated-text
```

DBMS_JOB モジュール

DBMS_JOB モジュールは、ジョブの作成、スケジューリング、および管理のためのプロシージャーを提供します。

DBMS_JOB モジュールは、管理タスク・スケジューラー (ATS) に対する代替インターフェースを提供します。タスクを ATS に追加することによってジョブが作成されます。実際のタスク名は、DBMS_JOB.TASK_NAME_PREFIX プロシージャー名に、割り当てられたジョブ ID を連結することによって構成されます。例えば、1 がジョブ ID とすると、SAMPLE_JOB_TASK_1 のようになります。

ジョブは、既にデータベースに保管されているストアード・プロシージャーを実行します。 SUBMIT プロシージャーを使用して、ジョブ定義を作成および保管します。各ジョブにはジョブ ID が割り当てられ、それに関連付けられたストアード・プロシージャーと、ジョブを実行する時と頻度を記述した属性も、共に割り当てられます。

データベースで SUBMIT プロシージャーが最初に実行されるときに、必要に応じて SYSTOOLSPACE 表スペースが作成されます。

DBMS_JOB ルーチンに対してジョブ・スケジューリングを使用可能にするには、次のとおりに実行します。

```
db2set DB2_ATS_ENABLE=1
```

いつ、どれほどの頻度でジョブが実行されるかは、2 つの対話型パラメーター – **next_date** と **interval** によって決まります。 **next_date** パラメーターは、ジョブが実行される次回の日時を指定する日時値です。 **interval** パラメーターは、評価の結果として日時値になる関数日付を含むストリングです。ジョブを実行する直前に **interval** パラメーター内の式が評価され、その結果の値によって、ジョブで保管されている **next_date** 値が置換されます。それからジョブが実行されます。このようにして、毎回のジョブ実行の前に **interval** 内の式が再評価され、次回の実行のための **next_date** 日時が与えられます。

next_date パラメーターによって指定されるスケジュール済みジョブの最初の実行は、現在時刻よりも少なくとも 5 分後として設定する必要があり、各ジョブの実行間隔も少なくとも 5 分は必要です。

このモジュールのスキーマは SYSIBMADM です。

DBMS_JOB モジュールには、以下のシステム定義ルーチンが含まれています。

表 13. DBMS_JOB モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
BROKEN プロシージャー	示されたジョブが失敗しているかどうかを指定します。
CHANGE プロシージャー	ジョブのパラメーターを変更します。
INTERVAL プロシージャー	ジョブの実行のたびに再計算される日付関数によって、実行頻度を設定します。この値は、次回の実行日時になります。
NEXT_DATE プロシージャー	ジョブが実行される次回の日時を設定します。
REMOVE プロシージャー	データベースからジョブ定義を削除します。
RUN プロシージャー	失敗としてマーク付けされていても、ジョブを強制実行します。
SUBMIT プロシージャー	ジョブを作成し、ジョブ定義をデータベースに保管します。
WHAT プロシージャー	ジョブによって実行されるストアード・プロシージャーを変更します。

表 14. DBMS_JOB モジュールで使用可能なシステム定義定数

定数名	説明
ANY_INSTANCE	DBMS_JOB ルーチンのインスタンス引数でサポートされる唯一の値です。
TASK_NAME_PREFIX	この定数には、管理タスク・スケジューラーのタスク名を構成するために接頭部として使用されるストリングが入ります。

使用上の注意

各データベースに対する DBMS_JOB モジュールを通して最初のジョブがサブミットされるときに、管理タスク・スケジューラーのセットアップが以下のように行われます。

1. SYSTOOLSPACE 表スペースがまだ存在していなければ、作成します。
2. ATS 表およびビュー (例えば SYSTOOLS.ADMIN_TASK_LIST) を作成します。

スケジュール済みジョブをリストするには、次のように実行します。

```
db2 SELECT * FROM systools.admin_task_list
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

ジョブ実行の状況を表示するには、次のように実行します。

```
db2 SELECT * FROM systools.admin_task_status
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

例

例 1: 以下の例は、ストアード・プロシージャー job_proc を使用します。このストアード・プロシージャーは、単にタイム・スタンプを jobrun 表に挿入します。この表には、単一の VARCHAR 列があります。

```
CREATE TABLE jobrun (
    runtime         VARCHAR(40)
)@

CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END@
```

この例では、以下の出力が結果として戻ります。

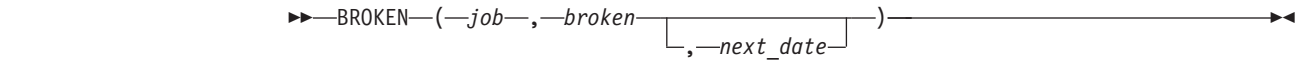
```
CREATE TABLE jobrun ( runtime         VARCHAR(40) )
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END
DB20000I  The SQL command completed successfully.
```

BROKEN プロシージャー - ジョブが失敗した状態かどうかを設定する

BROKEN プロシージャーは、ジョブが失敗した状態かどうかを設定します。

失敗したジョブは、RUN プロシージャーを使用しなければ実行できません。

構文



パラメーター

job
失敗しているかどうかが設定されるジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

broken
ジョブの状況を指定する、タイプ BOOLEAN の入力引数。 "true" に設定する場合、ジョブ状態は broken (失敗している) に設定されます。 "false" に設定する場合、ジョブ状態は not broken (失敗していない) に設定されます。失敗したジョブは、RUN プロシージャーを使わなければ実行できません。

next_date
ジョブが実行される日時を指定する、タイプ DATE のオプションの入力引数。デフォルトは SYSDATE です。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: ジョブ ID が 104 のジョブの状態を失敗に設定します。

```
CALL DBMS_JOB.BROKEN(104,true);
```

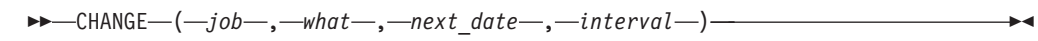
例 2: 状態を「失敗していない」に戻します。

```
CALL DBMS_JOB.BROKEN(104,false);
```

CHANGE プロシージャー - ジョブ属性の変更

CHANGE プロシージャーは、実行可能 SQL ステートメント、ジョブが次回に実行される日時、およびジョブの実行頻度を含む特定のジョブ属性を変更します。

構文



パラメーター

job
変更する属性を持つジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

what
実行可能 SQL ステートメントを指定する、タイプ VARCHAR(1024) の入力引数。既存の値をそのまま変更しない場合には、この引数を NULL に設定します。

next_date
ジョブが実行される次回の日時を指定する、タイプ TIMESTAMP(0) の入力引数。既存の値をそのまま変更しない場合には、この引数を NULL に設定します。

interval
評価の結果として、ジョブが実行される次回の日時を示す日付関数を指定する、タイプ VARCHAR(1024) の入力引数。既存の値をそのまま変更しない場合には、この引数を NULL に設定します。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

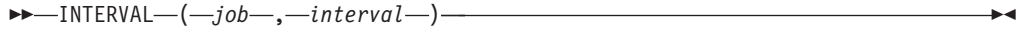
例 1: 次回は 2009 年 12 月 13 日に実行されるようにジョブを変更します。他のパラメーターは変更せず、そのままにします。

```
CALL DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-09','DD-MON-YY'),NULL);
```

INTERVAL プロシージャー - 実行頻度の設定

INTERVAL プロシージャーは、ジョブを実行する頻度を設定します。

構文



パラメーター

job
頻度が変更されるジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

interval
評価の結果として、ジョブが実行される次回の日時を示す日付関数を指定する、タイプ VARCHAR(1024) の入力引数。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

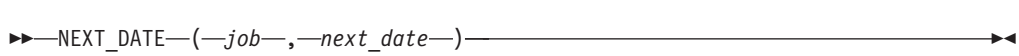
例 1: 週に 1 回実行するようにジョブを変更します。

```
CALL DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
```

NEXT_DATE プロシージャー - ジョブが実行される日時の設定

NEXT_DATE プロシージャーは、ジョブが実行される次回の日時を設定します。

構文



パラメーター

job
次回の実行日付を変更するジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

next_date
ジョブが次回に実行される日時を指定する、タイプ TIMESTAMP(0) の入力引数。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: 次回は 2009 年 12 月 14 日に実行されるようにジョブが変更します。

```
CALL DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-09','DD-MON-YY'));
```

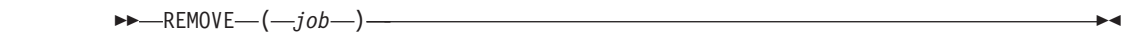
REMOVE プロシージャー - データベースからのジョブ定義の削除

REMOVE プロシージャーは、指定されたジョブをデータベースから削除します。

ジョブが今後再び実行されるようにするには、SUBMIT プロシージャーを使用してジョブを再サブミットする必要があります。

注: ジョブに関連付けられたストアード・プロシージャーは、ジョブが除去されるときに削除されません。

構文



パラメーター

job
データベースから除去されるジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: データベースからジョブを除去します。

```
CALL DBMS_JOB.REMOVE(104);
```

RUN プロシージャー - 失敗ジョブの強制実行

RUN プロシージャーは、ジョブが失敗状態であっても、ジョブを強制実行します。

構文

パラメーター

job
実行するジョブの ID を指定する、タイプ DECIMAL(20) の入力引数。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: ジョブを強制実行します。

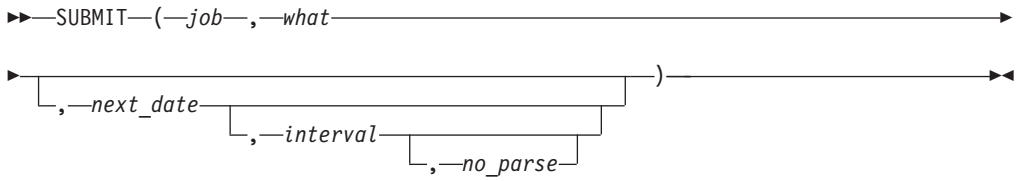
```
CALL DBMS_JOB.RUN(104);
```

SUBMIT プロシージャー - ジョブ定義を作成してデータベースに保管する

SUBMIT プロシージャーはジョブ定義を作成して、それをデータベースに保管します。

ジョブを構成するのは、ジョブ ID、実行されるストアード・プロシージャー、ジョブが最初に実行される時、およびジョブが実行される次回の日時を計算する日付関数です。

構文



パラメーター

job
ジョブに割り当てられた ID を指定する、タイプ DECIMAL(20) の出力引数。

what
動的に実行可能な SQL ステートメントの名前を指定する、タイプ VARCHAR(1024) の入力引数。

next_date
ジョブが実行される次回の日時を指定する、タイプ TIMESTAMP(0) のオプションの入力引数。デフォルトは SYSDATE です。

interval
評価の結果として、次回実行された後の実行の日時を示す日付関数を指定する、タイプ VARCHAR(1024) のオプションの入力引数。 *interval* が NULL に設定された場合には、ジョブは 1 回だけ実行されます。 NULL がデフォルトです。

no_parse
タイプ BOOLEAN のオプションの入力引数。 true に設定された場合には、ジョブ作成時に SQL ステートメントの構文検査を行わず、その代わりに、ジョブが最初に実行されるときにだけ構文検査を実行します。 false に設定された場合には、ジョブ作成時に SQL ステートメントの構文検査を行います。デフォルトは false です。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: 次の例では、ストアード・プロシージャー job_proc を使用してジョブを作成します。ジョブはまず、約 5 分以内に実行され、その後は *interval* 引数の SYSDATE + 1 の設定に従って、1 日に 1 回実行されます。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END@
```

このコマンドからの出力は、次のようになります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END
DB20000I  The SQL command completed successfully.

jobid: 1
```

WHAT プロシージャー - ジョブによって実行される SQL ステートメントの変更

WHAT プロシージャーは、指定されたジョブによって実行される SQL ステートメントを変更します。

構文

パラメーター

job
動的に実行可能な SQL ステートメントが変更される対象となるジョブ ID を指定する、タイプ DECIMAL(20) の入力引数。

what
動的に実行可能な SQL ステートメントを指定する、タイプ VARCHAR(1024) の入力引数。

許可

DBMS_JOB モジュールに対する EXECUTE 特権。

例

例 1: `list_emp` プロシージャーを実行するようにジョブを変更します。

```
CALL DBMS_JOB.WHAT(104,'list_emp;');
```

DBMS_LOB モジュール

DBMS_LOB モジュールは、ラージ・オブジェクトを操作する機能を提供します。

個々のプロシージャーおよび関数を説明している以下のセクションでは、ラージ・オブジェクトが BLOB である場合には、長さとオフセットはバイト単位で測ります。ラージ・オブジェクトが CLOB である場合には、長さとオフセットは文字単位で測ります。

DBMS_LOB モジュールは、10M バイトまでの LOB データをサポートします。

このモジュールのスキーマは SYSIBMADM です。

DBMS_LOB モジュールには、以下のルーチンが含まれています。

表 15. DBMS_LOB モジュール内で使用可能なシステム定義のルーチン

ルーチン名	説明
APPEND プロシージャー	1 つのラージ・オブジェクトを他のものに付加します。
CLOSE プロシージャー	オープンされているラージ・オブジェクトを閉じます。
COMPARE 関数	2 つのラージ・オブジェクトを比較します。
CONVERTTOBLOB プロシージャー	文字データをバイナリーに変換します。
CONVERTTOCLOB プロシージャー	バイナリー・データを文字に変換します。
COPY プロシージャー	1 つのラージ・オブジェクトを他のものにコピーします。
ERASE プロシージャー	ラージ・オブジェクトを消去します。
GET_STORAGE_LIMIT 関数	ラージ・オブジェクトのストレージ限度を取得します。
GETLENGTH 関数	ラージ・オブジェクトの長さを取得します。
INSTR 関数	オフセットで始まるラージ・オブジェクト内のパターンの n 番目の出現位置を取得します。
ISOPEN 関数	ラージ・オブジェクトがオープンされているかどうかを調べます。
OPEN プロシージャー	ラージ・オブジェクトをオープンします。
READ プロシージャー	ラージ・オブジェクトを読み取ります。
SUBSTR 関数	ラージ・オブジェクトの一部を取得します。
TRIM プロシージャー	ラージ・オブジェクトを指定の長さに切り取ります。
WRITE プロシージャー	ラージ・オブジェクトにデータを書き込みます。
WRITEAPPEND プロシージャー	ラージ・オブジェクトの終わりまで、バッファーからのデータを書き込みます。

次の表は、モジュールで使用可能なパブリック変数をリストしています。

表 16. DBMS_LOB パブリック変数

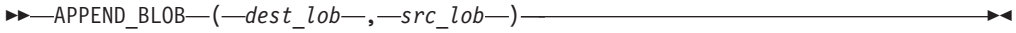
パブリック変数	データ・タイプ	値
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1

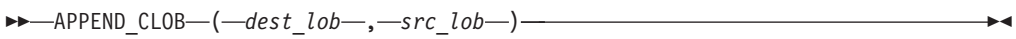
APPEND プロシージャー - 1 つのラージ・オブジェクトを別のものに付加する

APPEND プロシージャーは、1 つのラージ・オブジェクトを別のものに付加する機能を提供します。

注: 両方のラージ・オブジェクトが同じタイプでなければなりません。

構文





パラメーター

dest_lob
宛先オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプが BLOB(10M) または CLOB(10M) の入力または出力引数。 *src_lob* と同じデータ・タイプでなければなりません。

src_lob
ソース・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプが BLOB(10M) または CLOB(10M) の入力引数。 *dest_lob* と同じデータ・タイプでなければなりません。

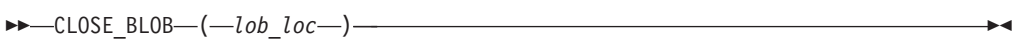
許可

DBMS_LOB モジュールに対する EXECUTE 特権。

CLOSE プロシージャー - オープンしているラージ・オブジェクトのクローズ

CLOSE プロシージャーはノーオペレーションです。

構文



パラメーター

lob_loc
クローズするラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。

許可

DBMS_LOB モジュールに対する EXECUTE 特権。

例

例 1:

COMPARE 関数 - 2 つのラージ・オブジェクトの比較

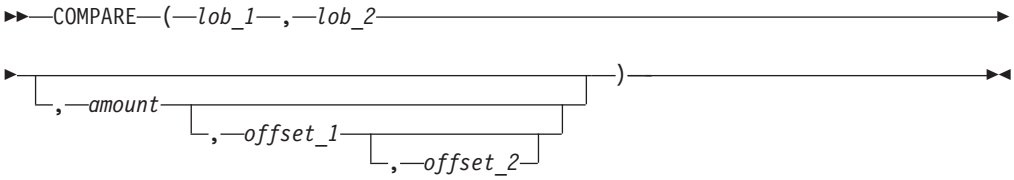
COMPARE 関数は、2 つのラージ・オブジェクトを対象に、指定されたオフセットの指定された長さで、バイトごとの厳密な比較を行います。

この関数は、以下の情報を戻します。

- 指定されたオフセットの指定された長さで両方のラージ・オブジェクトがまったく同じ場合は、ゼロ。
- オブジェクトが同じでない場合は、ゼロ以外。
- *amount*、*offset_1*、または *offset_2* がゼロより小さい場合は NULL。

注: 比較されるラージ・オブジェクトは、同じデータ・タイプでなければなりません。

構文



パラメーター

lob_1
比較される第 1 のラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。 *lob_2* と同じデータ・タイプでなければなりません。

lob_2
比較される第 2 のラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。 *lob_1* と同じデータ・タイプでなければなりません。

amount
タイプ INTEGER のオプション入力引数。ラージ・オブジェクトのデータ・タイプが BLOB の場合は、*amount* バイトについて比較が行われます。ラージ・オブジェクトのデータ・タイプが CLOB の場合は、*amount* 文字について比較が行われます。デフォルトはラージ・オブジェクトの最大サイズです。

offset_1
第 1 のラージ・オブジェクト内の比較開始位置を指定する、タイプ INTEGER のオプション入力引数。最初のバイト (または文字) がオフセット 1 となります。デフォルトは 1 です。

offset_2
第 2 のラージ・オブジェクト内の比較開始位置を指定する、タイプ INTEGER のオプション入力引数。最初のバイト (または文字) がオフセット 1 となります。デフォルトは 1 です。

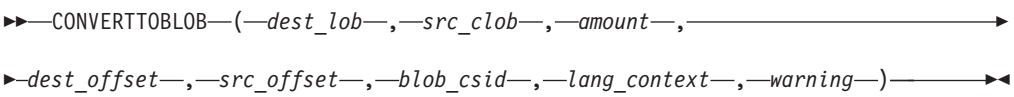
許可

DBMS_LOB モジュールに対する EXECUTE 特権。

CONVERTTOBLOB プロシージャー - 文字データをバイナリーに変換する

CONVERTTOBLOB プロシージャーは、文字データをバイナリーに変換する機能を提供します。

構文



パラメーター

dest_lob
文字データの変換先のラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) の入力または出力引数。

src_clob
変換される文字データのラージ・オブジェクト・ロケーターを指定する、タイプ CLOB(10M) の入力引数。

amount
変換される *src_clob* の文字数を指定する、タイプ INTEGER の入力引数。

dest_offset
変換先 BLOB 内の変換元 CLOB の書き込み開始位置 (バイト単位) を指定する、タイプ INTEGER の入力または出力引数。最初のバイトはオフセット 1 となります。

src_offset
変換元 CLOB 内の変換先 BLOB への変換開始位置 (文字単位) を指定する、タイプ INTEGER の入力または出力引数。最初の文字はオフセット 1 となります。

blob_csid
変換先 BLOB の文字セット ID を指定する、タイプ INTEGER の入力引数。この値は、データベース・コード・ページと一致しなければなりません。

lang_context
変換の言語コンテキストを指定する、タイプ INTEGER の入力引数。この値は 0 でなければなりません。

warning
常に 0 を戻す、タイプ INTEGER の出力引数。

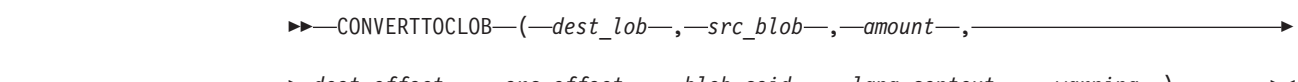
許可

DBMS_LOB モジュールに対する EXECUTE 特権。

CONVERTTOCLOB プロシージャー - バイナリー・データを文字に変換する

CONVERTTOCLOB プロシージャーは、バイナリー・データを文字に変換する機能を提供します。

構文



パラメーター

dest_lob
バイナリー・データの変換先のラージ・オブジェクト・ロケーターを指定する、タイプ CLOB(10M) の入力または出力引数。

src_clob
変換されるバイナリー・データのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) の入力引数。

amount
変換される *src_blob* の文字数を指定する、タイプ INTEGER の入力引数。

dest_offset
変換先 CLOB 内の変換元 BLOB の書き込み開始位置 (文字単位) を指定する、タイプ INTEGER の入力または出力引数。最初のバイトはオフセット 1 となります。

src_offset
変換元 BLOB 内の変換先 CLOB への変換開始位置 (バイト単位) を指定する、タイプ INTEGER の入力または出力引数。最初の文字はオフセット 1 となります。

blob_csid
変換元 BLOB の文字セット ID を指定する、タイプ INTEGER の入力引数。この値は、データベース・コード・ページと一致しなければなりません。

lang_context
変換の言語コンテキストを指定する、タイプ INTEGER の入力引数。この値は 0 でなければなりません。

warning
常に 0 を戻す、タイプ INTEGER の出力引数。

許可

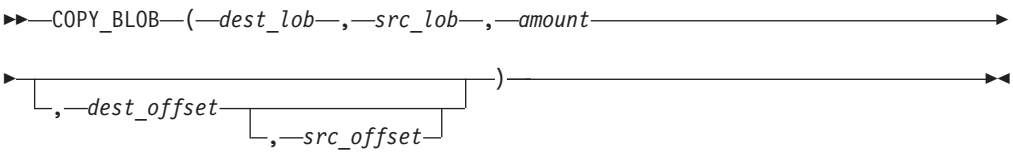
DBMS_LOB モジュールに対する EXECUTE 特権。

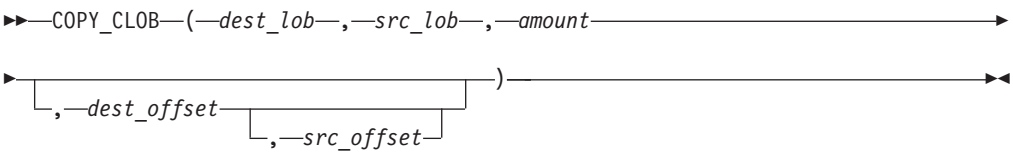
COPY プロシージャー - ラージ・オブジェクトを別のラージ・オブジェクトにコピーする

COPY プロシージャーは、ラージ・オブジェクトを別のラージ・オブジェクトにコピーする機能を提供します。

注: コピー元とコピー先のラージ・オブジェクトは、同じデータ・タイプでなければなりません。

構文





パラメーター

dest_lob
: *src_lob* のコピー先のラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。 *src_lob* と同じデータ・タイプでなければなりません。

src_lob
: *dest_lob* のコピー元のラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。 *dest_lob* と同じデータ・タイプでなければなりません。

amount
: コピーされる *src_lob* のバイト数または文字数を指定する、タイプ INTEGER の入力引数。

dest_offset
: コピー先ラージ・オブジェクト内のコピー元ラージ・オブジェクトの書き込み開始位置を指定する、タイプ INTEGER のオプション入力引数。最初の位置はオフセット 1 となります。デフォルトは 1 です。

src_offset
: コピー元ラージ・オブジェクト内のコピー先ラージ・オブジェクトへのコピー開始位置を指定する、タイプ INTEGER のオプション入力引数。最初の位置はオフセット 1 となります。デフォルトは 1 です。

許可

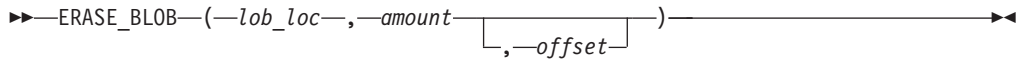
DBMS_LOB モジュールに対する EXECUTE 特権。

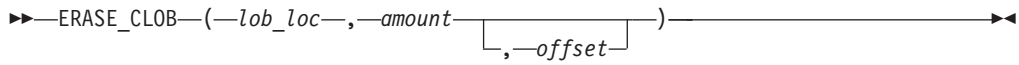
ERASE プロシージャー - ラージ・オブジェクトの部分消去

ERASE プロシージャーは、ラージ・オブジェクトの一部を消去する機能を提供します。

ラージ・オブジェクトを消去するということは、指定した部分をゼロ・バイト充てん文字 (BLOB の場合) またはスペース (CLOB の場合) に置き換えることを意味します。ラージ・オブジェクトの実際のサイズは変更されません。

構文





パラメーター

lob_loc
消去するラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。

amount
消去するバイト数または文字数を指定する、タイプ INTEGER の入力または出力引数。

offset
ラージ・オブジェクト内の消去開始位置を指定する、タイプ INTEGER のオプション入力引数。最初のバイトまたは文字が位置 1 となります。デフォルトは 1 です。

許可

DBMS_LOB モジュールに対する EXECUTE 特権。

GET_STORAGE_LIMIT 関数 - 最大許容ラージ・オブジェクトの限度を戻す

GET_STORAGE_LIMIT 関数は、最大許容ラージ・オブジェクトの限度を戻します。

この関数は、このデータベースにおけるラージ・オブジェクトの最大許容サイズを示す INTEGER 値を戻します。

構文

許可

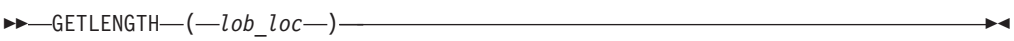
DBMS_LOB モジュールに対する EXECUTE 特権。

GETLENGTH 関数 - ラージ・オブジェクトの長さを戻す

GETLENGTH 関数は、ラージ・オブジェクトの長さを戻します。

この関数は、ラージ・オブジェクトの長さを示す INTEGER 値を戻します。BLOB の場合はバイト数、CLOB の場合は文字数です。

構文



パラメーター

lob_loc
長さを取得するラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。

許可

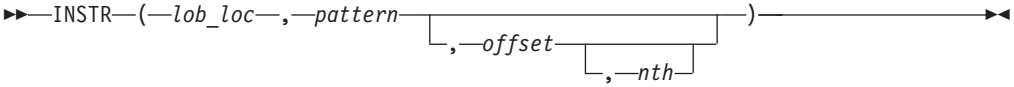
DBMS_LOB モジュールに対する EXECUTE 特権。

INSTR 関数 - 指定されたパターンの *n* 番目の出現位置を戻す

INSTR 関数は、ラージ・オブジェクト内の指定されたパターンの *n* 番目の出現位置を戻します。

この関数は、ラージ・オブジェクト内のパターンの n 番目 (*nth* で指定) の出現位置の INTEGER 値を戻します。この値は *offset* で指定された位置から始まります。

構文



パラメーター

lob_loc
pattern を検索するラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB または CLOB の入力引数。

pattern
ラージ・オブジェクトに対して突き合わせるバイトまたは文字のパターンを指定する、タイプ BLOB(32767) または VARCHAR(32672) の入力引数。 *lob_loc* が BLOB の場合、*pattern* は BLOB でなければなりません。また、*lob_loc* が CLOB の場合、*pattern* は VARCHAR でなければなりません。

offset
lob_loc 内の *pattern* 検索開始位置を指定する、タイプ INTEGER のオプション入力引数。最初のバイトまたは文字が位置 1 となります。デフォルト値は 1 です。

nth
offset で指定された位置からの *pattern* の検索回数を指定する、タイプ INTEGER のオプション引数。デフォルト値は 1 です。

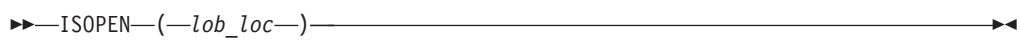
許可

DBMS_LOB モジュールに対する EXECUTE 特権。

ISOPEN 関数 - ラージ・オブジェクトがオープンされているかどうかのテスト

ISOPEN 関数は、常に INTEGER 値 1 を戻します。

構文



パラメーター

lob_loc
この関数でテストするラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。

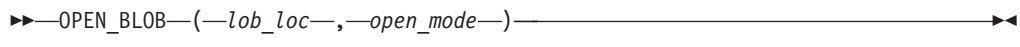
許可

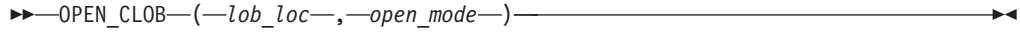
DBMS_LOB モジュールに対する EXECUTE 特権。

OPEN プロシージャー - ラージ・オブジェクトのオープン

OPEN プロシージャーはノーオペレーションです。

構文





パラメーター

lob_loc
オープンするラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。

open_mode
ラージ・オブジェクトをオープンするモードを指定する、タイプ INTEGER の入力引数。読み取り専用モードの場合は 0 に設定します (`lob_readonly`)。読み取り/書き込みモードの場合は 1 に設定します (`lob_readwrite`)。

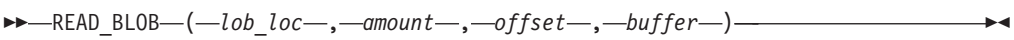
許可

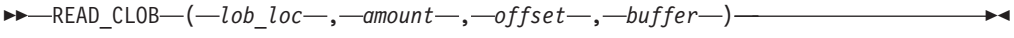
DBMS_LOB モジュールに対する EXECUTE 特権。

READ プロシージャー - ラージ・オブジェクトの部分読み取り

READ プロシージャーは、ラージ・オブジェクトの一部を読み取ってバッファーに入れる機能を提供します。

構文





パラメーター

lob_loc
読み取られるラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。

amount
読み取るバイト数または文字数を指定する、タイプ INTEGER の入力または出力引数。

offset
読み取り開始位置を指定する、タイプ INTEGER の入力引数。最初のバイトまたは文字が位置 1 となります。

buffer
ラージ・オブジェクトを受け取る変数を指定する、タイプ BLOB(32762) または VARCHAR(32672) の出力引数。 *lob_loc* が BLOB の場合、*buffer* は BLOB でなければなりません。 *lob_loc* が CLOB の場合、*buffer* は VARCHAR でなければなりません。

許可

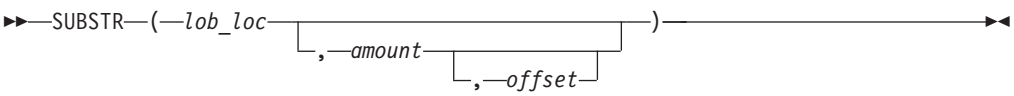
DBMS_LOB モジュールに対する EXECUTE 特権。

SUBSTR 関数 - ラージ・オブジェクトの一部を戻す

SUBSTR 関数は、ラージ・オブジェクトの一部を戻す機能を提供します。

この関数は、読み取ったラージ・オブジェクトの戻す部分の BLOB(32767) 値 (BLOB の場合) または VARCHAR 値 (CLOB の場合) を戻します。

構文



パラメーター

lob_loc
読み取られるラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力引数。

amount
戻されるバイト数または文字数を指定する、タイプ INTEGER のオプション入力引数。デフォルト値は 32,767 です。

offset
ラージ・オブジェクト内のデータを戻す開始位置を指定する、タイプ INTEGER のオプション入力引数。最初のバイトまたは文字が位置 1 となります。デフォルト値は 1 です。

許可

DBMS_LOB モジュールに対する EXECUTE 特権。

TRIM プロシージャー - 指定された長さになるようにラージ・オブジェクトを切り捨てる

TRIM プロシージャーは、指定された長さになるようにラージ・オブジェクトを切り捨てる機能を提供します。

構文

```
►►—TRIM_BLOB—(—lob_loc—,—newlen—)———————————————————————►◄

►►—TRIM_CLOB—(—lob_loc—,—newlen—)———————————————————————►◄
```

パラメーター

lob_loc
切り取られるラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。

newlen
ラージ・オブジェクトの切り取り限界の新しいバイト数または文字数を指定する、タイプ INTEGER の入力引数。

許可

DBMS_LOB モジュールに対する EXECUTE 特権。

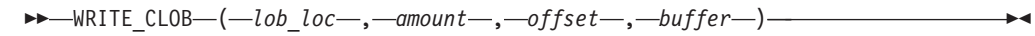
WRITE プロシージャー - ラージ・オブジェクトへのデータの書き込み

WRITE プロシージャーは、ラージ・オブジェクトにデータを書き込む機能を提供します。

ラージ・オブジェクト内の指定されたオフセットの指定された長さの既存データが、バッファーにあるデータで上書きされます。

構文

```
►►—WRITE_BLOB—(—lob_loc—,—amount—,—offset—,—buffer—)———————————►◄
```



パラメーター

lob_loc
上書きされるラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB(10M) または CLOB(10M) の入力または出力引数。

amount
ラージ・オブジェクトに書き込まれる *buffer* 内のバイト数または文字数を指定する、タイプ INTEGER の入力引数。

offset
ラージ・オブジェクトの先頭からの書き込み操作開始オフセットをバイト数または文字数で指定する、タイプ INTEGER の入力引数。ラージ・オブジェクトの開始値は 1 です。

buffer
ラージ・オブジェクトに書き込まれるデータが入っている、タイプ BLOB(32767) または VARCHAR(32672) の入力引数。 *lob_loc* が BLOB の場合、*buffer* は BLOB でなければなりません。 *lob_loc* が CLOB の場合、*buffer* は VARCHAR でなければなりません。

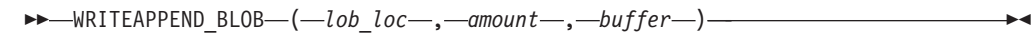
許可

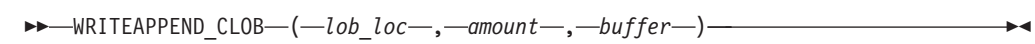
DBMS_LOB モジュールに対する EXECUTE 特権。

WRITEAPPEND プロシージャー - ラージ・オブジェクトの末尾へのデータの付加

WRITEAPPEND プロシージャーは、ラージ・オブジェクトの末尾にデータを追加する機能を提供します。

構文





パラメーター

lob_loc
データが付加されるラージ・オブジェクトのラージ・オブジェクト・ロケーターを指定する、タイプ BLOB または CLOB の入力または出力引数。

amount
ラージ・オブジェクトに付加される *buffer* 内のバイト数または文字数を指定する、タイプ INTEGER の入力引数。

buffer
ラージ・オブジェクトに付加されるデータが入っている、タイプ BLOB(32767)

または VARCHAR(32672) の入力引数。 *lob_loc* が BLOB の場合、*buffer* は BLOB でなければなりません。 *lob_loc* が CLOB の場合、*buffer* は VARCHAR でなければなりません。

許可

DBMS_LOB モジュールに対する EXECUTE 特権。

DBMS_OUTPUT モジュール

DBMS_OUTPUT モジュールは、メッセージ・バッファーにメッセージを書き込み (複数行のテキスト)、メッセージ・バッファーからメッセージを取得する一連のプロシージャーを提供します。これらのプロシージャーは、メッセージを標準出力に書き込む必要のあるアプリケーション・デバッグの際に役立ちます。

このモジュールのスキーマは SYSIBMADM です。

DBMS_OUTPUT モジュールには、以下のシステム定義ルーチンが含まれています。

表 17. DBMS_OUTPUT モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
DISABLE プロシージャー	メッセージ・バッファーを使用不可にします。
ENABLE プロシージャー	メッセージ・バッファーを使用可能にします。
GET_LINE プロシージャー	メッセージ・バッファーから 1 行のテキストを取得します。
GET_LINES プロシージャー	メッセージ・バッファーから 1 行以上のテキストを取得し、そのテキストをコレクションに書き込みます。
NEW_LINE プロシージャー	メッセージ・バッファーに行末文字シーケンスを書き込みます。
PUT プロシージャー	メッセージ・バッファーに行末文字シーケンスを含まないストリングを書き込みます。
PUT_LINE プロシージャー	メッセージ・バッファーに行末文字シーケンスを含む 1 行を書き込みます。

このモジュール内のプロシージャーを使用すると、メッセージ・バッファーを処理できます。標準出力に出力をリダイレクトするには、コマンド行プロセッサー (CLP) コマンド SET SERVEROUTPUT ON を使用します。

例

proc1 で PUT プロシージャーと PUT_LINE プロシージャーを使用して、メッセージ・バッファー内に 1 行のテキストを書き込みます。proc1 を初めて実行する時には、SET SERVEROUTPUT ON が指定されており、メッセージ・バッファー内の行が CLP ウィンドウに表示されます。proc1 を 2 回目に実行する時には、SET SERVEROUTPUT OFF が指定されており、メッセージ・バッファーから CLP ウィンドウに表示される行はありません。

```
CREATE PROCEDURE proc1( P1 VARCHAR(10) )
BEGIN
  CALL DBMS_OUTPUT.PUT( 'P1 = ' );
  CALL DBMS_OUTPUT.PUT_LINE( P1 );
END@

SET SERVEROUTPUT ON@

CALL proc1( '10' )@

SET SERVEROUTPUT OFF@

CALL proc1( '20' )@
```

この例では、結果として以下の出力が得られます。

```
CALL proc1( '10' )

  Return Status = 0

P1 = 10

SET SERVEROUTPUT OFF
DB20000I  The SET SERVEROUTPUT command completed successfully.

CALL proc1( '20' )

  Return Status = 0
```

DISABLE プロシージャー - メッセージ・バッファーを使用不可にする

DISABLE プロシージャーは、メッセージ・バッファーを使用不可にします。

このプロシージャーを実行すると、メッセージ・バッファー内のすべてのメッセージが破棄されます。PUT、PUT_LINE、または NEW_LINE プロシージャーの呼び出しは無視され、送信側にエラーは戻りません。

構文

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

以下の例を使用すると、現行セッションでメッセージ・バッファーが使用不可になります。

```
CALL DBMS_OUTPUT.DISABLE@
```

使用上の注意

メッセージ・バッファーが使用不可になった後にメッセージを送受信するには、ENABLE プロシージャーを使用します。

ENABLE プロシージャー - メッセージ・バッファーを使用可能にする

ENABLE プロシージャーは、メッセージ・バッファーを使用可能にします。単一セッションで、アプリケーションはメッセージ・バッファーにメッセージを書き込み、メッセージ・バッファーからメッセージを取得できます。

構文

プロシージャー・パラメーター

buffer_size
メッセージ・バッファーの最大長をバイト単位で指定する、タイプ INTEGER の入力引数。*buffer_size* に 2000 未満の値を指定すると、バッファー・サイズは 2000 に設定されます。この値が NULL の場合、デフォルトのバッファー・サイズは 20000 です。

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

以下の例を使用すると、メッセージ・バッファーが使用可能になります。

```
CALL DBMS_OUTPUT.ENABLE( NULL )@
```

使用上の注意

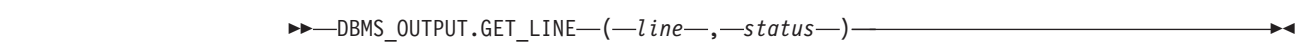
ENABLE プロシージャーを呼び出して、既存のメッセージ・バッファーのサイズを増やすことができます。古いバッファー内のメッセージは、拡張されたバッファーにコピーされます。

GET_LINE プロシージャー - メッセージ・バッファーから 1 行を取得する

GET_LINE プロシージャーは、メッセージ・バッファーから 1 行のテキストを取得します。テキストは、行末文字シーケンスによって終えなければなりません。

ヒント: メッセージ・バッファーに行末文字シーケンスを追加するためには、PUT_LINE プロシージャーを使用するか、PUT プロシージャーに対する一連の呼び出しを行った後に NEW_LINE プロシージャーを使用します。

構文



プロシージャー・パラメーター

line
メッセージ・バッファーから 1 行のテキストを戻す、タイプ VARCHAR(32672) の出力引数。

status
メッセージ・バッファーから 1 行を戻したかどうかを示す、タイプ INTEGER の出力引数。

- 0 は、1 行が戻されたことを示します。
- 1 は、戻された行がないことを示します。

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

GET_LINE プロシージャーを使用して、メッセージ・バッファーから 1 行のテキストを取得します。この例では、proc1 によってメッセージ・バッファーにテキストが 1 行書き込まれます。 proc3 はメッセージ・バッファーからテキストを取得し、それを messages という名前の表に挿入します。その後 proc2 を実行しますが、メッセージ・バッファーは使用不可なので、メッセージ・バッファーに追加されるテキストはありません。SELECT ステートメントが実行されると、proc1 によって追加されたテキストだけが戻されます。

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE line VARCHAR(32672);
  DECLARE status INT;

  CALL DBMS_OUTPUT.GET_LINE( line, status );
  while status = 0 do
    INSERT INTO messages VALUES ( line );
    CALL DBMS_OUTPUT.GET_LINE( line, status );
  end while;
END@

CALL proc1@

CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@
```

```
CALL proc3@

SELECT * FROM messages@
```

この例では、以下の出力が結果として戻ります。

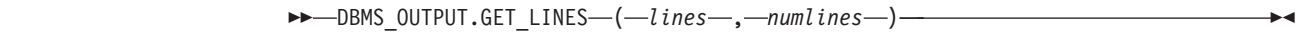
```
MSG
--------------------------------------------
PROC1 put this line in the message buffer.

  1 record(s) selected.
```

GET_LINES プロシージャー - メッセージ・バッファーから複数行を取得する

GET_LINES プロシージャーは、メッセージ・バッファーから 1 行以上のテキストを取得し、そのテキストをコレクションとして保管します。テキストの各行は、行末文字シーケンスによって終えなければなりません。

ヒント: メッセージ・バッファーに行末文字シーケンスを追加するためには、PUT_LINE プロシージャーを使用するか、PUT プロシージャーに対する一連の呼び出しを行った後に NEW_LINE プロシージャーを使用します。

構文



プロシージャー・パラメーター

lines
: メッセージ・バッファーから複数行のテキストを戻す、タイプ DBMS_OUTPUT.CHARARR の出力引数。タイプ DBMS_OUTPUT.CHARARR は、内部的には VARCHAR(32672) ARRAY[2147483647] 配列として定義されています。

numlines
: タイプ INTEGER の入出力引数。入力として使用される場合、メッセージ・バッファーから取り出される行数を指定します。出力として使用される場合、メッセージ・バッファーから取り出された実際の行数を示します。*numlines* の出力値が入力値より小さい場合、メッセージ・バッファーにはまったく行が残っていません。

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

GET_LINES プロシージャーを使用して、メッセージ・バッファーから複数行のテキストを取得し、配列にそのテキストを保管します。配列内のテキストは、表に挿入し、照会することができます。

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
```

```
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE lines DBMS_OUTPUT.CHARARR;
  DECLARE numlines INT;
  DECLARE i INT;

  CALL DBMS_OUTPUT.GET_LINES( lines, numlines );
  SET i = 1;
  WHILE i <= numlines DO
    INSERT INTO messages VALUES ( lines[i] );
    SET i = i + 1;
  END WHILE;
END@

CALL proc1@

CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@

CALL proc3@

SELECT * FROM messages@
```

この例では、以下の出力が結果として戻ります。

```
MSG
----------------------------------------
PROC1 put this line in the message buffer.
PROC1 put this line in the message buffer

  2 record(s) selected.
```

NEW_LINE プロシージャー - メッセージ・バッファーに行末文字シーケンスを書き込む

NEW_LINE プロシージャーは、メッセージ・バッファーに行末文字シーケンスを書き込みます。

構文

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

NEW_LINE プロシージャーを使用して、メッセージ・バッファーに行末文字シーケンスを書き込みます。この例では、行末文字シーケンスが後に続くテキストが出力として表示されます。SET SERVEROUTPUT ON が指定されているためです。ただし、メッセージ・バッファー内にあるテキストのうち、行末文字が後に続かないテキストは表示されません。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'i' );
  CALL DBMS_OUTPUT.PUT( 's' );
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'a' );
  CALL DBMS_OUTPUT.PUT( 't' );
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

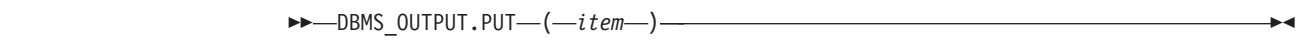
この例では、以下の出力が結果として戻ります。

```
This
```

PUT プロシージャー - メッセージ・バッファーに行の一部を書き込む

PUT プロシージャーは、メッセージ・バッファーにストリングを書き込みます。ストリングの末尾には、行末文字シーケンスは書き込まれません。

構文



プロシージャー・パラメーター

item
メッセージ・バッファーに書き込むテキストを指定する、タイプ VARCHAR(32672) の入力引数。

許可

DBMS_OUTPUT モジュールに対する EXECUTE 特権。

例

PUT プロシージャーを使用して、メッセージ・バッファーに行の一部を書き込みます。この例では、NEW_LINE プロシージャーがメッセージ・バッファーに行末文字シーケンスを追加します。proc1 を実行すると、SET SERVEROUTPUT ON が指定されているので、1 行のテキストが戻ります。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'H' );
  CALL DBMS_OUTPUT.PUT( 'e' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'o' );
  CALL DBMS_OUTPUT.PUT( '.' );
  CALL DBMS_OUTPUT.NEW_LINE;
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

この例では、以下の出力が結果として戻ります。

```
Hello.
```

使用上の注意

PUT プロシージャーを使用してメッセージ・バッファーにテキストを追加した後には、NEW_LINE プロシージャーを使用してメッセージ・バッファーに行末文字シーケンスを追加してください。そうしないと、このテキストは完全な行ではないので、GET_LINE プロシージャーと GET_LINES プロシージャーによっても戻りません。

PUT_LINE プロシージャー - メッセージ・バッファーに行全体を書き込む

PUT_LINE プロシージャーは、メッセージ・バッファーに行末文字シーケンスを含む 1 行を書き込みます。

構文

プロシージャー・パラメーター

item
メッセージ・バッファーに書き込むテキストを指定する、タイプ VARCHAR(32672) の入力引数。

許可

PUT_LINE プロシージャーに対する EXECUTE 特権。

例

PUT_LINE プロシージャーを使用して、行末文字シーケンスを含む行をメッセージ・バッファーに書き込みます。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE PROC1()
BEGIN
```

```
  CALL DBMS_OUTPUT.PUT( 'a' );
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT_LINE( 'b' );
END@

CALL PROC1@

SET SERVEROUTPUT OFF@
```

この例では、以下の出力が結果として戻ります。

```
a
b
```

DBMS_PIPE モジュール

DBMS_PIPE モジュールは、同じデータベースに接続されたセッション内またはセッション間のパイプを通して、メッセージを送信するための一連のルーチンを提供します。

このモジュールのスキーマは SYSIBMADM です。

DBMS_PIPE モジュールには、以下のシステム定義ルーチンが含まれています。

表 18. DBMS_PIPE モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
CREATE_PIPE 関数	プライベートまたはパブリック・パイプを明示的に作成します。
NEXT_ITEM_TYPE 関数	受信されたメッセージの次の項目のデータ・タイプを判別します。
PACK_MESSAGE 関数	項目をセッションのローカル・メッセージ・バッファーに入れます。
PACK_MESSAGE_RAW プロシージャー	タイプ RAW の項目をセッションのローカル・メッセージ・バッファーに入れます。
PURGE プロシージャー	指定されたパイプの受信されていないメッセージを除去します。
RECEIVE_MESSAGE 関数	指定されたパイプからメッセージを取得します。
REMOVE_PIPE 関数	作成されたパイプを明示的に削除します。
RESET_BUFFER プロシージャー	ローカル・メッセージ・バッファーをリセットします。
SEND_MESSAGE プロシージャー	指定されたパイプにメッセージを送信します。
UNIQUE_SESSION_NAME 関数	ユニークなセッション名を戻します。
UNPACK_MESSAGE プロシージャー	メッセージから次のデータ項目を取得し、これを変数に割り当てます。

使用上の注意

パイプは、プロシージャー呼び出し中に暗黙的または明示的に作成されます。プロシージャー呼び出しに、存在しないパイプ名への参照が含まれている場合は、*暗黙*

的パイプ が作成されます。例えば、「mailbox」という名前のパイプが SEND_MESSAGE プロシージャーに渡され、そのパイプがまだ存在しない場合は、「mailbox」という名前の新しいパイプが作成されます。 CREATE_PIPE 関数を呼び出し、パイプの名前を指定することによって、*明示的パイプ* が作成されます。

パイプはプライベートの場合とパブリックの場合があります。*プライベート・パイプ* は、パイプを作成したユーザーのみがアクセスできます。管理者であっても、他のユーザーが作成したプライベート・パイプにはアクセスできません。*パブリック・パイプ* は、DBMS_PIPE モジュールへのアクセス権限を持つすべてのユーザーがアクセスできます。パイプのアクセス・レベルを指定するには、CREATE_PIPE 関数を使用して、*private* パラメーターの値を指定します。「false」は、パイプがパブリックであることを指定します。「true」は、パイプがプライベートであることを指定します。値を指定しない場合は、デフォルトでは、プライベート・パイプが作成されます。すべての暗黙的パイプはプライベートです。

パイプを通してメッセージを送信するには、PACK_MESSAGE 関数を呼び出して、現行セッションでユニークなローカル・メッセージ・バッファーに、個別のデータ項目 (行) を書き出します。次に、SEND_MESSAGE プロシージャーを呼び出して、パイプを通してメッセージを送信します。

メッセージを受信するには、RECEIVE_MESSAGE 関数を呼び出して、指定されたパイプからメッセージを取得します。メッセージは、受信中のセッションのローカル・メッセージ・バッファーに書き込まれます。次に、UNPACK_MESSAGE プロシージャーを呼び出して、ローカル・メッセージ・バッファーから次のデータ項目を取得し、指定されたプログラム変数に割り当てます。パイプに複数のメッセージが含まれている場合は、RECEIVE_MESSAGE 関数は、メッセージを先入れ先出し法の順序で取得します。

それぞれのセッションは、PACK_MESSAGE 関数によって作成されたメッセージ、および RECEIVE_MESSAGE 関数によって取得されたメッセージごとに別個のメッセージ・バッファーを保守します。別個のメッセージ・バッファーを使用することによって、同じセッションでメッセージを構築し、受信することができます。ただし、RECEIVE_MESSAGE 関数が連続して呼び出された場合、最後の RECEIVE_MESSAGE 呼び出しからのメッセージのみが、ローカル・メッセージ・バッファーに保持されます。

例

接続 1 で、「pipe1」という名前のパイプを作成します。セッションのローカル・メッセージ・バッファーにメッセージを書き込み、pipe1 を通してそのメッセージを送信します。

```
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@
```

接続 2 で、メッセージを受信して解凍し、標準出力に表示します。

```
SET SERVEROUTPUT ON@

BEGIN
```

```
  DECLARE status   INT;
  DECLARE int1     INTEGER;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_INT( int1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'int1: ' || int1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@
```

この例では、以下の出力が結果として戻ります。

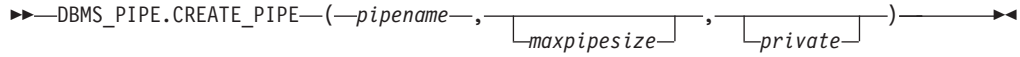
```
  varchar1: message1
```

CREATE_PIPE 関数 - パイプを作成する

CREATE_PIPE 関数は、指定された名前でパブリックまたはプライベート・パイプを明示的に作成します。

明示的なパブリックまたはプライベート・パイプについて詳しくは、DBMS_PIPE モジュールに関するトピックを参照してください。

構文



戻り値

この関数は、パイプが正常に作成された場合は、状況コード 0 を戻します。

関数のパラメーター

pipename
パイプの名前を指定する、タイプ VARCHAR (128) の入力引数。パイプについて詳しくは、255 ページの『DBMS_PIPE モジュール』を参照してください。

maxpipesize
パイプの最大容量をバイト単位で指定する、タイプ INTEGER のオプション入力引数。 デフォルトは 8192 バイトです。

private
パイプのアクセス・レベルを指定するオプション入力引数。

非パーティション・データベース環境の場合
値が「0」または「FALSE」の場合は、パブリック・パイプが作成されます。

値が「1」または「TRUE」の場合は、プライベート・パイプが作成されます。これはデフォルトです。

パーティション・データベース環境の場合
値が「0」の場合は、パブリック・パイプが作成されます。

値が「1」の場合は、プライベート・パイプが作成されます。これはデフォルトです。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

例 1: 名前付きメッセージであるプライベート・パイプを作成します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_status        INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('messages');
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
CREATE_PIPE status: 0
```

例 2 名前付きメールボックスであるパブリック・パイプを作成します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE v_status INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('mailbox',0);
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc2@
```

この例では、以下の出力が結果として戻ります。

```
CREATE_PIPE status: 0
```

NEXT_ITEM_TYPE 関数 - 次の項目のデータ・タイプ・コードを戻す

NEXT_ITEM_TYPE 関数は、受信されたメッセージの次のデータ項目のデータ・タイプを識別する整数コードを戻します。

受信されたメッセージは、セッションのローカル・メッセージ・バッファーに保管されます。UNPACK_MESSAGE プロシージャーを使用して、ローカル・メッセージ・バッファーから各項目を移動させ、次に NEXT_ITEM_TYPE 関数を使用して、次に使用可能な項目のデータ・タイプ・コードを戻します。メッセージに項目が残っていない場合は、コード 0 が戻されます。

構文

戻り値

この関数は、データ・タイプを表す以下のコードの 1 つを戻します。

表 19. NEXT_ITEM_TYPE データ・タイプ・コード

タイプ・コード	データ・タイプ
0	データ項目がこれ以上見つからない
6	INTEGER
9	VARCHAR
12	DATE
23	BLOB

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 で、メッセージを圧縮して送信します。proc2 で、メッセージを受信し、次に NEXT_ITEM_TYPE 関数を使用してこれを解凍し、そのタイプを判別します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
```

```
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

この例では、以下の出力が結果として戻ります。

```
varchar1: message1
```

PACK_MESSAGE 関数 - ローカル・メッセージ・バッファーにデータ項目を入れる

PACK_MESSAGE 関数は、セッションのローカル・メッセージ・バッファーにデータ項目を入れます。

構文

プロシージャー・パラメーター

item
式を含む、タイプ VARCHAR(4096)、DATE、または DECFLOAT の入力引数。この式によって戻される値は、セッションのローカル・メッセージ・バッファーに追加されます。

ヒント: タイプ RAW のデータ項目をローカル・メッセージ・バッファーに入れるには、PACK_MESSAGE_RAW プロシージャーを使用します。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

PACK_MESSAGE 関数を使用して Sujata に対するメッセージをローカル・メッセージ・バッファーに入れ、次に SEND_MESSAGE プロシージャーを使用して、このメッセージをパイプ上に送信します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
```

```
  DECLARE    v_status    INTEGER;
  DECLARE    status      INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
  SEND_MESSAGE status: 0
```

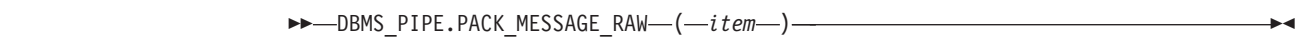
使用上の注意

PACK_MESSAGE 関数または PACK_MESSAGE_RAW プロシージャーを少なくとも一度呼び出してから、SEND_MESSAGE 呼び出しを発行する必要があります。

PACK_MESSAGE_RAW プロシージャー - ローカル・メッセージ・バッファーにタイプ RAW のデータ項目を入れる

PACK_MESSAGE_RAW プロシージャーは、セッションのローカル・メッセージ・バッファーにタイプ RAW のデータ項目を入れます。

構文



プロシージャー・パラメーター

item
式を指定する、タイプ BLOB(4096) の入力引数。この式によって戻される値は、セッションのローカル・メッセージ・バッファーに追加されます。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

PACK_MESSAGE_RAW プロシージャーを使用して、ローカル・メッセージ・バッファーにタイプ RAW のデータ項目を入れます。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_raw           BLOB(100);
  DECLARE v_raw2   BLOB(100);
  DECLARE v_status        INTEGER;
  SET v_raw = BLOB('21222324');
  SET v_raw2 = BLOB('30000392');
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw);
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw2);
  SET v_status = DBMS_PIPE.SEND_MESSAGE('datatypes');
```

```
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
SEND_MESSAGE status: 0
```

使用上の注意

PACK_MESSAGE 関数または PACK_MESSAGE_RAW プロシージャーを少なくとも一度呼び出してから、SEND_MESSAGE 呼び出しを発行する必要があります。

PURGE プロシージャー - 受信されていないメッセージをパイプから除去する

PURGE プロシージャーは、指定された暗黙的パイプの受信されていない除去を除去します。

ヒント: 明示的パイプを削除するには、REMOVE_PIPE 関数を使用します。

構文

プロシージャー・パラメーター

pipename
暗黙的パイプの名前を指定する、タイプ VARCHAR (128) の入力引数。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 では、パイプ上に 2 つのメッセージ、Message #1 および Message #2 を送信します。proc2 では、最初のメッセージを受信して、解凍し、パイプをパージします。proc3 を実行すると、使用可能なメッセージがないため、RECEIVE_MESSAGE 関数の呼び出しがタイムアウトになり、状況コード 1 が戻されます。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE   v_status        INTEGER;
  DECLARE   status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
```

```
  DECLARE   v_item          VARCHAR(80);
  DECLARE   v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.PURGE('pipe');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END@

CALL proc1@

CALL proc2@

CALL proc3@
```

この例では、以下の出力が結果として戻ります。

proc1 から

```
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

proc2 から

```
RECEIVE_MESSAGE status: 0
Item: Hi, Sujata
```

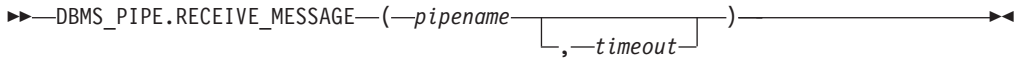
proc3 から

```
RECEIVE_MESSAGE status: 1
```

RECEIVE_MESSAGE 関数 - 指定されたパイプからメッセージを取得する

RECEIVE_MESSAGE 関数は、指定されたパイプからメッセージを取得します。

構文



戻り値

RECEIVE_MESSAGE 関数は、タイプ INTEGER の以下の状況コードの 1 つを戻します。

表 20. RECEIVE_MESSAGE 状況コード

状況コード	説明
0	成功
1	タイムアウト

関数のパラメーター

pipename
パイプの名前を指定する、タイプ VARCHAR (128) の入力引数。指定されたパイプが存在しない場合は、パイプが暗黙的に作成されます。パイプについて詳しくは、255 ページの『DBMS_PIPE モジュール』を参照してください。

timeout
待ち時間を秒単位で指定する、タイプ INTEGER のオプション入力引数。デフォルトは 86400000 (1000 日) です。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 では、メッセージを送信します。proc2 では、メッセージを受信して解凍します。メッセージが 1 秒以内に受信されない場合は、タイムアウトになります。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

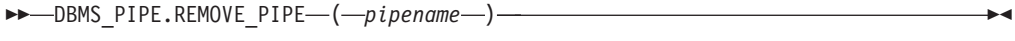
この例では、以下の出力が結果として戻ります。

```
RECEIVE_MESSAGE status: 0
Item: message1
```

REMOVE_PIPE 関数 - パイプを削除する

REMOVE_PIPE 関数は、作成されたパイプを明示的に削除します。CREATE_PIPE 関数によって作成された任意のパブリックまたはプライベート・パイプを削除するには、この関数を使用します。

構文



戻り値

この関数は、タイプ INTEGER の以下の状況コードの 1 つを戻します。

表 21. REMOVE_PIPE 状況コード

状況コード	説明
0	パイプは正常に除去されたか存在しません
NULL	例外がスローされました

関数のパラメーター

pipename
パイプの名前を指定する、タイプ VARCHAR (128) の入力引数。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 では、パイプ上に 2 つのメッセージ、Message #1 および Message #2 を送信します。proc2 では、最初のメッセージを受信して、解凍し、パイプを削除します。proc3 を実行すると、パイプが存在しないため、RECEIVE_MESSAGE 関数の呼び出しがタイムアウトになり、状況コード 1 が戻されます。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET  v_status = DBMS_PIPE.CREATE_PIPE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  SET status = DBMS_PIPE.REMOVE_PIPE('pipe1');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
```

```
END@

CALL proc1@

CALL proc2@

CALL proc3@
```

この例では、以下の出力が結果として戻ります。

proc1 から

```
CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

proc2 から

```
RECEIVE_MESSAGE status: 0
Item: Message #1
```

proc3 から

```
RECEIVE_MESSAGE status: 1
```

RESET_BUFFER プロシージャー - ローカル・メッセージ・バッファーをリセットする

RESET_BUFFER プロシージャーは、セッションのローカル・メッセージ・バッファーへのポインターをバッファーの先頭にリセットします。 バッファーをリセットすることによって、RESET_BUFFER 呼び出しより前にメッセージ・バッファーに存在していたあらゆるデータ項目が、以降の PACK_MESSAGE 呼び出しによって、上書きされます。

構文

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 では、PACK_MESSAGE 関数を使用して、Sujata という名前の従業員に関するメッセージをローカル・メッセージ・バッファーに書き込みます。
RESET_BUFFER プロシージャーを呼び出して、このメッセージを Bing に関するメッセージで置き換え、パイプに送信します。proc2 では、Bing に対するメッセージを受信して解凍します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
```

```
  CALL DBMS_PIPE.RESET_BUFFER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Bing');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@

CALL proc2@
```

この例では、以下の出力が結果として戻ります。

proc1 から

```
SEND_MESSAGE status: 0
```

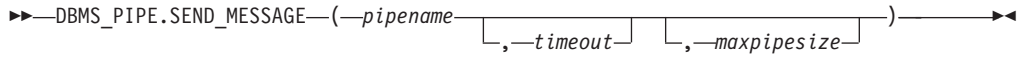
proc2 から

```
RECEIVE_MESSAGE status: 0
Item: Hi, Bing
Item: Can you attend a meeting at 9:30, tomorrow?
```

SEND_MESSAGE プロシージャー - メッセージを指定されたパイプに送信する

SEND_MESSAGE プロシージャーは、メッセージをセッションのローカル・メッセージ・バッファーから、指定されたパイプに送信します。

構文



戻り値

このプロシージャーは、タイプ INTEGER の以下の状況コードの 1 つを戻します。

表 22. SEND_MESSAGE 状況コード

状況コード	説明
0	成功
1	タイムアウト

プロシージャー・パラメーター

pipename
パイプの名前を指定する、タイプ VARCHAR (128) の入力引数。指定されたパ

イプが存在しない場合は、パイプが暗黙的に作成されます。パイプについて詳しくは、 255 ページの『DBMS_PIPE モジュール』を参照してください。

timeout
待ち時間を秒単位で指定する、タイプ INTEGER のオプション入力引数。デフォルトは 86400000 (1000 日) です。

maxpipesize
パイプの最大容量をバイト単位で指定する、タイプ INTEGER のオプション入力引数。デフォルトは 8192 バイトです。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 では、メッセージを送信します。proc2 では、メッセージを受信して解凍します。メッセージが 1 秒以内に受信されない場合は、タイムアウトになります。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

この例では、以下の出力が結果として戻ります。

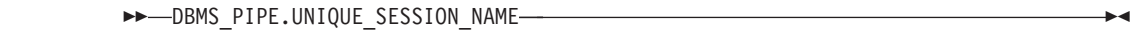
```
RECEIVE_MESSAGE status: 0
Item: message1
```

UNIQUE_SESSION_NAME 関数 - 固有のセッション名を戻す

UNIQUE_SESSION_NAME 関数は、現行セッションの固有の名前を戻します。

この関数を使用して、現行セッションと同じ名前を持つパイプを作成できます。このパイプを作成するには、UNIQUE_SESSION_NAME 関数によって戻された値を SEND_MESSAGE プロシージャーにパイプ名として渡します。現行セッションと同じ名前を持つ暗黙的パイプが作成されます。

構文



戻り値

この関数は、現行セッションの固有の名前を表すタイプ VARCHAR (128) の値を戻します。

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

現行セッションと同じ名前を持つパイプを作成します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    status          INTEGER;
  DECLARE    v_session       VARCHAR(30);
  SET v_session = DBMS_PIPE.UNIQUE_SESSION_NAME;
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE(v_session);
  CALL DBMS_OUTPUT.PUT_LINE('Sent message on pipe ' || v_session);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

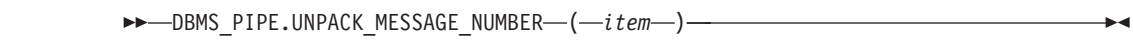
```
Sent message on pipe *LOCAL.myschema.080522010048
```

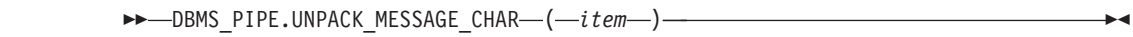
UNPACK_MESSAGE プロシージャー - データ項目をローカル・メッセージ・バッファーから取得する

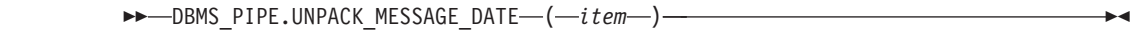
UNPACK_MESSAGE プロシージャーは、メッセージから次のデータ項目を取得し、変数に割り当てます。

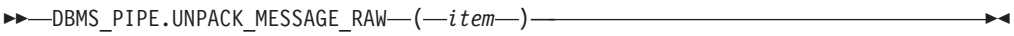
UNPACK_MESSAGE プロシージャーの 1 つを呼び出す前に、RECEIVE_MESSAGE プロシージャーを使用して、メッセージをローカル・メッセージ・バッファーに配置します。

構文









プロシージャー・パラメーター

item
データ項目をローカル・メッセージ・バッファーから受信するための変数を指定する、以下のいずれかのタイプの出力引数。

ルーチン	データ・タイプ
UNPACK_MESSAGE_NUMBER	DECFLOAT
UNPACK_MESSAGE_CHAR	VARCHAR(4096)
UNPACK_MESSAGE_DATE	DATE
UNPACK_MESSAGE_RAW	BLOB(4096)

許可

DBMS_PIPE モジュールに対する EXECUTE 特権。

例

proc1 で、メッセージを圧縮して送信します。proc2 では、メッセージを受信し、項目のタイプに基づいて適切なプロシージャーを使用することによって解凍し、メッセージを標準出力に表示します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
```

```
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

この例では、以下の出力が結果として戻ります。

```
varchar1: message1
```

DBMS_SQL モジュール

DBMS_SQL モジュールは、動的 SQL を実行するための一連のプロシージャーを提供します。したがって、さまざまなデータ操作言語 (DML) ステートメントやデータ定義言語 (DDL) ステートメントをサポートします。

このモジュールのスキーマは SYSIBMADM です。

DBMS_SQL モジュールには、以下のシステム定義ルーチンが含まれています。

表 23. DBMS_SQL モジュールで使用可能なシステム定義ルーチン

プロシージャー名	説明
BIND_VARIABLE_BLOB プロシージャー	IN または INOUT パラメーターの入力 BLOB 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを BLOB と定義します。
BIND_VARIABLE_CHAR プロシージャー	IN または INOUT パラメーターの入力 CHAR 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを CHAR と定義します。
BIND_VARIABLE_CLOB プロシージャー	IN または INOUT パラメーターの入力 CLOB 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを CLOB と定義します。
BIND_VARIABLE_DATE プロシージャー	IN または INOUT パラメーターの入力 DATE 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを DATE と定義します。
BIND_VARIABLE_DOUBLE プロシージャー	IN または INOUT パラメーターの入力 DOUBLE 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを DOUBLE と定義します。
BIND_VARIABLE_INT プロシージャー	IN または INOUT パラメーターの入力 INTEGER 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを INTEGER と定義します。

表 23. *DBMS_SQL* モジュールで使用可能なシステム定義ルーチン *(続き)*

プロシージャー名	説明
BIND_VARIABLE_NUMBER プロシージャー	IN または INOUT パラメーターの入力 DECFLOAT 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを DECFLOAT と定義します。
BIND_VARIABLE_RAW プロシージャー	IN または INOUT パラメーターの入力 BLOB(32767) 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを BLOB(32767) と定義します。
BIND_VARIABLE_TIMESTAMP プロシージャー	IN または INOUT パラメーターの入力 TIMESTAMP 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを TIMESTAMP と定義します。
BIND_VARIABLE_VARCHAR プロシージャー	IN または INOUT パラメーターの入力 VARCHAR 値を提供します。また、INOUT または OUT パラメーターの出力値のデータ・タイプを VARCHAR と定義します。
CLOSE_CURSOR プロシージャー	カーソルをクローズします。
COLUMN_VALUE_BLOB プロシージャー	タイプ BLOB の列の値を取り出します。
COLUMN_VALUE_CHAR プロシージャー	タイプ CHAR の列の値を取り出します。
COLUMN_VALUE_CLOB プロシージャー	タイプ CLOB の列の値を取り出します。
COLUMN_VALUE_DATE プロシージャー	タイプ DATE の列の値を取り出します。
COLUMN_VALUE_DOUBLE プロシージャー	タイプ DOUBLE の列の値を取り出します。
COLUMN_VALUE_INT プロシージャー	タイプ INTEGER の列の値を取り出します。
COLUMN_VALUE_LONG プロシージャー	タイプ CLOB(32767) の列の値を取り出します。
COLUMN_VALUE_NUMBER プロシージャー	タイプ DECFLOAT の列の値を取り出します。
COLUMN_VALUE_RAW プロシージャー	タイプ BLOB(32767) の列の値を取り出します。
COLUMN_VALUE_TIMESTAMP プロシージャー	タイプ TIMESTAMP の列の値を取り出します。
COLUMN_VALUE_VARCHAR プロシージャー	タイプ VARCHAR の列の値を取り出します。
DEFINE_COLUMN_BLOB プロシージャー	列のデータ・タイプを BLOB と定義します。
DEFINE_COLUMN_CHAR プロシージャー	列のデータ・タイプを CHAR と定義します。
DEFINE_COLUMN_CLOB プロシージャー	列のデータ・タイプを CLOB と定義します。
DEFINE_COLUMN_DATE プロシージャー	列のデータ・タイプを DATE と定義します。
DEFINE_COLUMN_DOUBLE プロシージャー	列のデータ・タイプを DOUBLE と定義します。

表 23. DBMS_SQL モジュールで使用可能なシステム定義ルーチン (続き)

プロシージャー名	説明
DEFINE_COLUMN_INT プロシージャー	列のデータ・タイプを INTEGER と定義します。
DEFINE_COLUMN_LONG プロシージャー	列のデータ・タイプを CLOB(32767) と定義します。
DEFINE_COLUMN_NUMBER プロシージャー	列のデータ・タイプを DECFLOAT と定義します。
DEFINE_COLUMN_RAW プロシージャー	列のデータ・タイプを BLOB(32767) と定義します。
DEFINE_COLUMN_TIMESTAMP プロシージャー	列のデータ・タイプを TIMESTAMP と定義します。
DEFINE_COLUMN_VARCHAR プロシージャー	列のデータ・タイプを VARCHAR と定義します。
DESCRIBE_COLUMNS プロシージャー	カーソルによって取り出された列の記述を戻します。
DESCRIBE_COLUMNS2 プロシージャー	32 文字より大きい列名が許可される点以外は DESCRIBE_COLUMNS と同じです。
EXECUTE プロシージャー	カーソルを実行します。
EXECUTE_AND_FETCH プロシージャー	カーソルを実行し、1 行をフェッチします。
FETCH_ROWS プロシージャー	カーソルから行をフェッチします。
IS_OPEN プロシージャー	カーソルがオープンされているかどうかを検査します。
LAST_ROW_COUNT プロシージャー	フェッチされた行の総数を戻します。
OPEN_CURSOR プロシージャー	カーソルをオープンします。
PARSE プロシージャー	DDL ステートメントを解析します。
VARIABLE_VALUE_BLOB プロシージャー	INOUT または OUT パラメーターの値を BLOB として取り出します。
VARIABLE_VALUE_CHAR プロシージャー	INOUT または OUT パラメーターの値を CHAR として取り出します。
VARIABLE_VALUE_CLOB プロシージャー	INOUT または OUT パラメーターの値を CLOB として取り出します。
VARIABLE_VALUE_DATE プロシージャー	INOUT または OUT パラメーターの値を DATE として取り出します。
VARIABLE_VALUE_DOUBLE プロシージャー	INOUT または OUT パラメーターの値を DOUBLE として取り出します。
VARIABLE_VALUE_INT プロシージャー	INOUT または OUT パラメーターの値を INTEGER として取り出します。
VARIABLE_VALUE_NUMBER プロシージャー	INOUT または OUT パラメーターの値を DECFLOAT として取り出します。
VARIABLE_VALUE_RAW プロシージャー	INOUT または OUT パラメーターの値を BLOB(32767) として取り出します。
VARIABLE_VALUE_TIMESTAMP プロシージャー	INOUT または OUT パラメーターの値を TIMESTAMP として取り出します。

表 23. DBMS_SQL モジュールで使用可能なシステム定義ルーチン (続き)

プロシージャー名	説明
VARIABLE_VALUE_VARCHAR プロシージャー	INOUT または OUT パラメーターの値を VARCHAR として取り出します。

次の表は、DBMS_SQL モジュールで使用可能なシステム定義のタイプと定数を示しています。

表 24. DBMS_SQL でのシステム定義のタイプと定数

Name	タイプまたは定数	説明
DESC_REC	タイプ	列情報のレコード。
DESC_REC2	タイプ	列情報のレコード。
DESC_TAB	タイプ	タイプ DESC_REC のレコードの配列。
DESC_TAB2	タイプ	タイプ DESC_REC2 のレコードの配列。
NATIVE	定数	PARSE プロシージャーの language_flag パラメーターに対して唯一サポートされる値。

使用上の注意

DBMS_SQL モジュールのルーチンは、動的 SQL ステートメントを構成および実行する際に役立ちます。例えば、「ALTER TABLE」や「DROP TABLE」などの DDL ステートメントや DML ステートメントを実行する、SQL ステートメントをオンザフライで構成して実行したり、動的 SQL を使用する関数を SQL ステートメント内から呼び出す場合に便利です。

BIND_VARIABLE_BLOB プロシージャー - 変数への BLOB 値のバインド

BIND_VARIABLE_BLOB プロシージャーは、SQL コマンドの中で BLOB 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文

```
►►—BIND_VARIABLE_BLOB—(—c—,—name—,—value—)———————————————————►◄
```

パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ BLOB(2G) の入力引数。

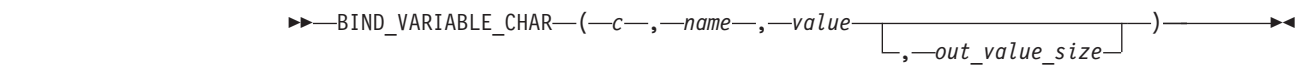
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_CHAR プロシージャー - 変数に CHAR 値をバインドする

BIND_VARIABLE_CHAR プロシージャーは、SQL コマンドの IN、INOUT、または OUT 引数に CHAR 値を関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ CHAR(254) の入力引数。

out_value_size
IN または INOUT 引数の長さ制限、および INOUT または OUT 引数の出力値の最大長を指定する、タイプ INTEGER のオプション入力引数。指定しない場合は、*value* の長さと見なされます。

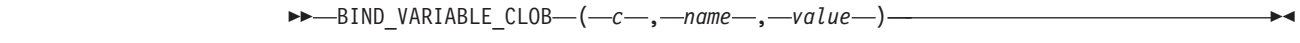
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_CLOB プロシージャー - 変数への CLOB 値のバインド

BIND_VARIABLE_CLOB プロシージャーは、SQL コマンドの中で CLOB 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ CLOB(2G) の入力引数。

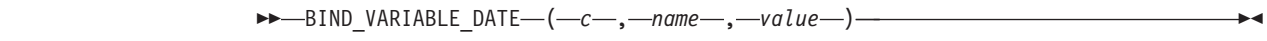
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_DATE プロシージャー - 変数への DATE 値のバインド

BIND_VARIABLE_DATE プロシージャーは、SQL コマンドの中で DATE 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ DATE の入力引数。

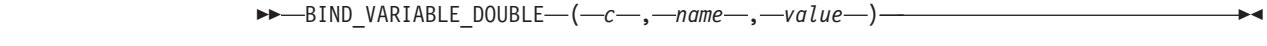
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_DOUBLE プロシージャー - 変数への DOUBLE 値のバインド

BIND_VARIABLE_DOUBLE プロシージャーは、SQL コマンドの中で DOUBLE 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ DOUBLE の入力引数。

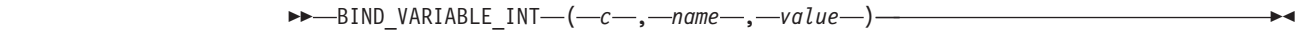
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_INT プロシージャー - 変数への INTEGER 値のバインド

BIND_VARIABLE_INT プロシージャーは、SQL コマンドの中で INTEGER 値を IN または INOUT バインド変数に関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ INTEGER の入力引数。

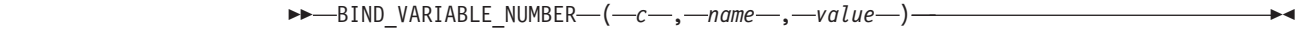
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_NUMBER プロシージャー - 変数に NUMBER 値をバインドする

BIND_VARIABLE_NUMBER プロシージャーは、SQL コマンドの IN、INOUT、または OUT 引数に NUMBER 値を関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ DECFLOAT の入力引数。

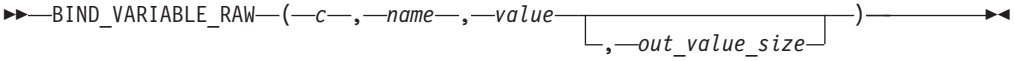
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_RAW プロシージャー - 変数に RAW 値をバインドする

BIND_VARIABLE_RAW プロシージャーは、SQL コマンドの IN、INOUT、または OUT 引数に RAW 値を関連付ける機能を提供します。

構文



パラメーター

c バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ BLOB(32767) の入力引数。

out_value_size
IN または INOUT 引数の長さ制限、および INOUT または OUT 引数の出力値の最大長を指定する、タイプ INTEGER のオプション入力引数。指定しない場合は、*value* の長さと見なされます。

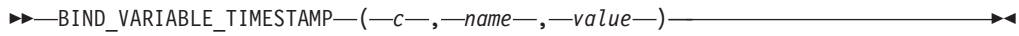
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_TIMESTAMP プロシージャー - 変数への TIMESTAMP 値のバインド

BIND_VARIABLE_TIMESTAMP プロシージャーは、SQL コマンドの中で TIMESTAMP 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文



パラメーター

c
バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ TIMESTAMP の入力引数。

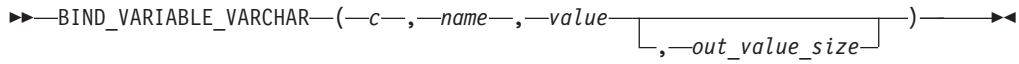
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

BIND_VARIABLE_VARCHAR プロシージャー - 変数への VARCHAR 値のバインド

BIND_VARIABLE_VARCHAR プロシージャーは、SQL コマンドの中で VARCHAR 値を IN、INOUT、または OUT 引数に関連付ける機能を提供します。

構文



パラメーター

c
バインド変数を含んだ SQL コマンドのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
SQL コマンド内のバインド変数の名前を指定する、タイプ VARCHAR (128) の入力引数。

value
割り当てる値を指定する、タイプ VARCHAR(32672) の入力引数。

out_value_size
IN または INOUT 引数の長さ限界、および INOUT または OUT 引数の出力値の最大長を指定する、タイプ INTEGER の入力引数。指定しない場合は、*value* の長さと見なされます。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

CLOSE_CURSOR プロシージャー - カーソルのクローズ

CLOSE_CURSOR プロシージャーは、オープンされているカーソルをクローズします。カーソルに割り振られたリソースは解放されて使用できなくなります。

構文

パラメーター

c クローズするカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

例

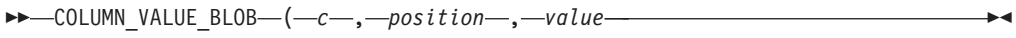
例 1: この例は、既にオープンされているカーソルをクローズする方法を示しています。

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

COLUMN_VALUE_BLOB プロシージャー - 変数に BLOB 列値を戻す

COLUMN_VALUE_BLOB プロシージャーは、カーソルから BLOB 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ BLOB(2G) の出力引数。

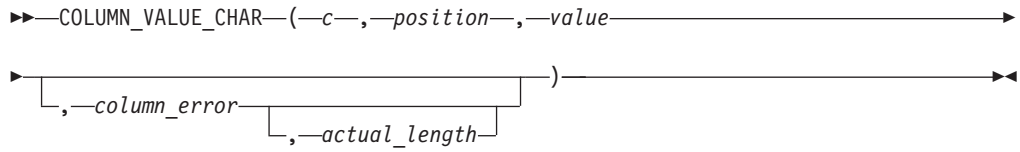
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_CHAR プロシージャー - 変数に CHAR 列値を戻す

COLUMN_VALUE_CHAR プロシージャーは、カーソルから CHAR 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
前のフェッチ呼び出しでカーソルによって戻されたデータを受け取る変数を指定する、タイプ CHAR の出力引数。

column_error
列に SQLCODE が関連付けられている場合はその SQLCODE を戻す、タイプ INTEGER のオプション出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER のオプション出力引数。

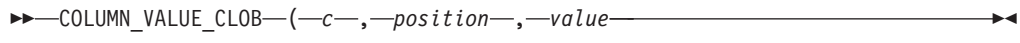
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_CLOB プロシージャー - 変数に CLOB 列値を戻す

COLUMN_VALUE_CLOB プロシージャーは、カーソルから CLOB 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ CLOB(2G) の出力引数。

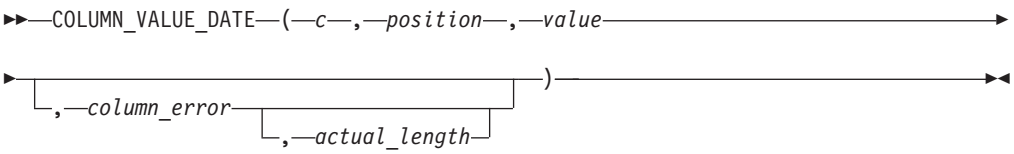
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_DATE プロシージャー - 変数に DATE 列値を戻す

COLUMN_VALUE_DATE プロシージャーは、カーソルから DATE 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ DATE の出力引数。

column_error
列に関連付けられた SQLCODE がある場合にはそれを返す、タイプ INTEGER の出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER の出力引数。

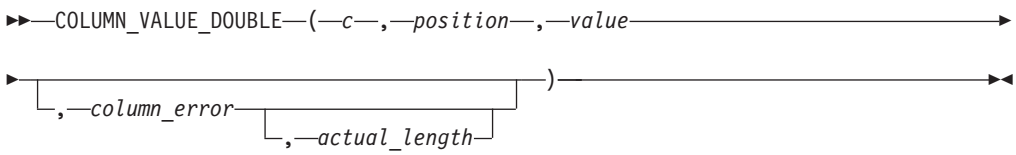
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_DOUBLE プロシージャー - 変数に DOUBLE 列値を戻す

COLUMN_VALUE_DOUBLE プロシージャーは、カーソルから DOUBLE 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ DOUBLE の出力引数。

column_error
列に関連付けられた SQLCODE がある場合にはそれを返す、タイプ INTEGER の出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER の出力引数。

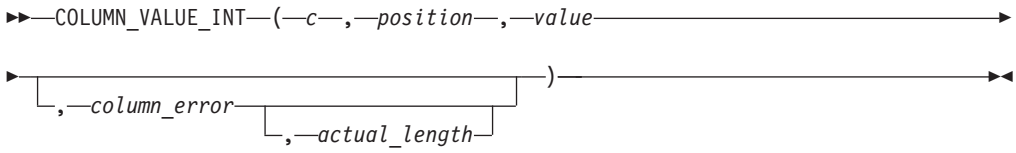
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_INT プロシージャー - 変数に INTEGER 列値を戻す

COLUMN_VALUE_INT プロシージャーは、カーソルから INTEGER 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ INTEGER の出力引数。

column_error
列に関連付けられた SQLCODE がある場合にはそれを返す、タイプ INTEGER の出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER の出力引数。

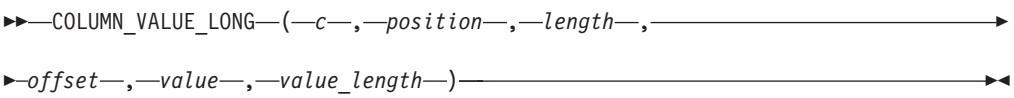
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_LONG プロシージャー - 変数に LONG 列値を戻す

COLUMN_VALUE_LONG プロシージャーは、カーソルから LONG 値の部分を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

length
offset から開始して取り出す LONG データの要求バイト数を指定する、タイプ INTEGER の入力引数。

offset
LONG 値内のデータ取り出し開始位置を指定する、タイプ INTEGER の入力引数。

value
前のフェッチ呼び出しでカーソルによって戻されたデータを受け取る変数を指定する、タイプ CLOB(32760) の出力引数。

value_length
戻されたデータの実際の長さを戻す、タイプ INTEGER の出力引数。

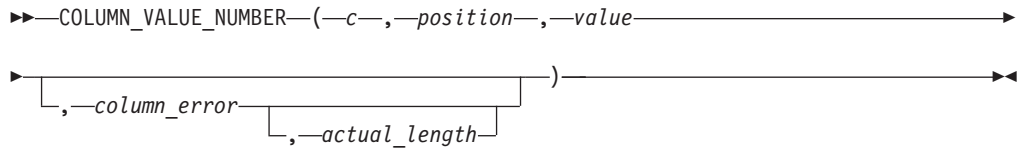
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_NUMBER プロシージャー - 変数に DECFLOAT 列値を戻す

COLUMN_VALUE_NUMBER プロシージャーは、カーソルから DECFLOAT 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
前のフェッチ呼び出しでカーソルによって戻されたデータを受け取る変数を指定する、タイプ DECFLOAT の出力引数。

column_error
列に SQLCODE が関連付けられている場合はその SQLCODE を戻す、タイプ INTEGER のオプション出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER のオプション出力引数。

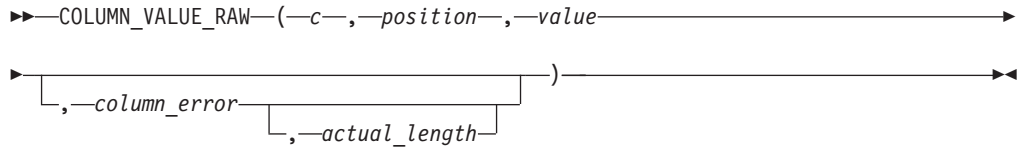
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_RAW プロシージャー - 変数に RAW 列値を戻す

COLUMN_VALUE_RAW プロシージャーは、カーソルから RAW 値を受け取る変数を定義します。

構文



パラメーター

c
定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
前のフェッチ呼び出しでカーソルによって戻されたデータを受け取る変数を指定する、タイプ BLOB(32767) の出力引数。

column_error
列に SQLCODE が関連付けられている場合はその SQLCODE を戻す、タイプ INTEGER のオプション出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER のオプション出力引数。

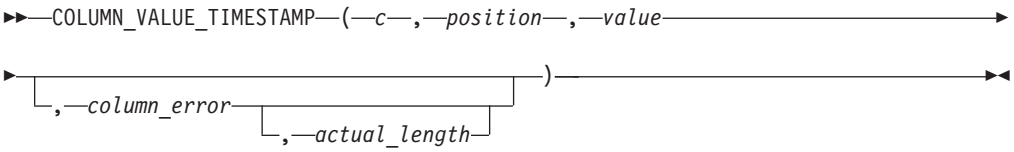
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_TIMESTAMP プロシージャー - 変数に TIMESTAMP 列値を戻す

COLUMN_VALUE_TIMESTAMP プロシージャーは、カーソルから TIMESTAMP 値を受け取る変数を定義します。

構文



パラメーター

c
定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ TIMESTAMP の出力引数。

column_error
列に関連付けられた SQLCODE がある場合にはそれを返す、タイプ INTEGER の出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER の出力引数。

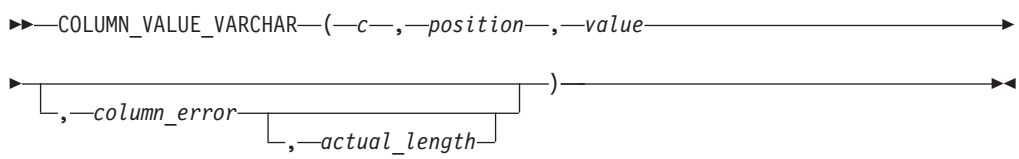
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

COLUMN_VALUE_VARCHAR プロシージャー - 変数に VARCHAR 列値を戻す

COLUMN_VALUE_VARCHAR プロシージャーは、カーソルから VARCHAR 値を受け取る変数を定義します。

構文



パラメーター

c 定義される変数にデータを戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

position
カーソル内の戻されるデータの位置を指定する、タイプ INTEGER の入力引数。カーソル内の最初の値が位置 1 となります。

value
以前のフェッチ呼び出しでカーソルによって返されたデータを受け取る変数を指定する、タイプ VARCHAR(32672) の出力引数。

column_error
列に関連付けられた SQLCODE がある場合にはそれを返す、タイプ INTEGER の出力引数。

actual_length
データの切り捨て前の実際の長さを戻す、タイプ INTEGER の出力引数。

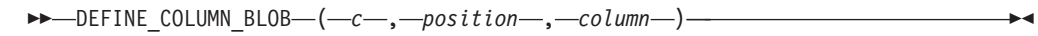
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_BLOB - SELECT リスト内の BLOB 列の定義

DEFINE_COLUMN_BLOB プロシージャーは、カーソルで返され取り出される、SELECT リスト内の BLOB 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ BLOB(2G) の入力引数。

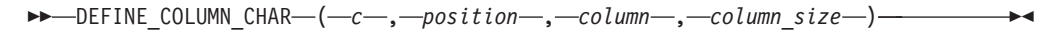
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_CHAR プロシージャー - SELECT リストの CHAR 列の定義

DEFINE_COLUMN_CHAR プロシージャーは、カーソルに戻されて取り出される、SELECT リストの CHAR 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ CHAR(254) の入力引数。

column_size
戻されるデータの最大長を指定する、タイプ INTEGER の入力引数。戻されるデータが *column_size* を超える場合は、文字数が *column_size* に切り捨てられます。

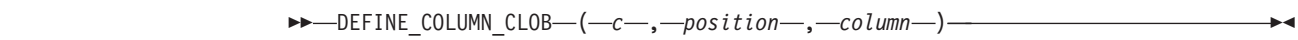
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_CLOB - SELECT リスト内の CLOB 列の定義

DEFINE_COLUMN_CLOB プロシージャーは、カーソルで返され取り出される、SELECT リスト内の CLOB 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ CLOB(2G) の入力引数。

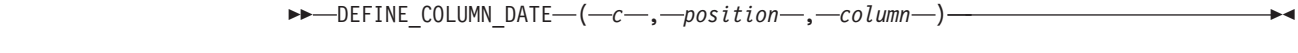
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_DATE - SELECT リスト内の DATE 列の定義

DEFINE_COLUMN_DATE プロシージャーは、カーソルで返され取り出される、SELECT リスト内の DATE 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ DATE の入力引数。

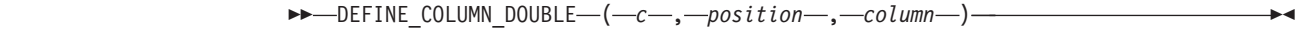
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_DOUBLE - SELECT リスト内の DOUBLE 列の定義

DEFINE_COLUMN_DOUBLE プロシージャーは、カーソルで返され取り出される、SELECT リスト内の DOUBLE 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ DOUBLE の入力引数。

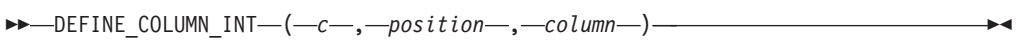
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_INT - SELECT リスト内の INTEGER 列の定義

DEFINE_COLUMN_INT プロシージャーは、カーソルで返され取り出される、SELECT リスト内の INTEGER 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ INTEGER の入力引数。

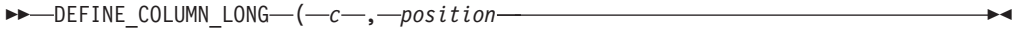
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_LONG プロシージャー - SELECT リストの LONG 列の定義

DEFINE_COLUMN_LONG プロシージャーは、カーソルに戻されて取り出される、SELECT リストの LONG 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

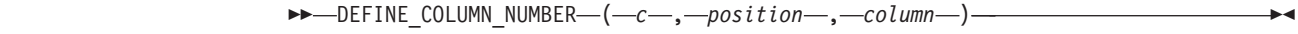
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_NUMBER プロシージャー - SELECT リストの DECFLOAT 列の定義

DEFINE_COLUMN_NUMBER プロシージャーは、カーソルに戻されて取り出される、SELECT リストの DECFLOAT 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ DECFLOAT の入力引数。

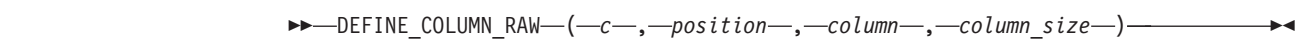
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_RAW プロシージャー - SELECT リストの RAW 列または式の定義

DEFINE_COLUMN_RAW プロシージャーは、カーソルに戻されて取り出される、SELECT リストの RAW 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ BLOB(32767) の入力引数。

column_size
戻されるデータの最大長を指定する、タイプ INTEGER の入力引数。戻されるデータが *column_size* を超える場合は、文字数が *column_size* に切り捨てられます。

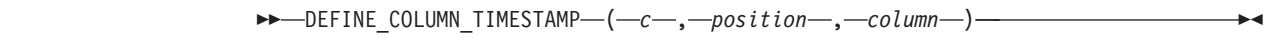
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_TIMESTAMP - SELECT リスト内の TIMESTAMP 列の定義

DEFINE_COLUMN_TIMESTAMP プロシージャーは、カーソルで返され取り出される、SELECT リスト内の TIMESTAMP 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ TIMESTAMP の入力引数。

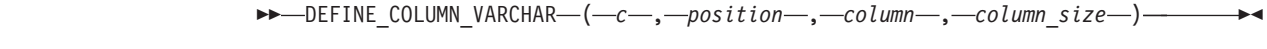
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DEFINE_COLUMN_VARCHAR プロシージャー - SELECT リスト内の VARCHAR 列の定義

DEFINE_COLUMN_VARCHAR プロシージャーは、カーソルで返され取り出される、SELECT リスト内の VARCHAR 列または式を定義します。

構文



パラメーター

c SELECT コマンドに関連付けられたカーソル・ハンドルを指定する、タイプ INTEGER の入力引数。

position
定義される、SELECT リスト内の列または式の位置を指定する、タイプ INTEGER の入力引数。

列 タイプ VARCHAR(32672) の入力引数。

column_size
戻されるデータの最大長を指定する、タイプ INTEGER の入力引数。戻されるデータが *column_size* を超える場合は、文字数が *column_size* に切り捨てられます。

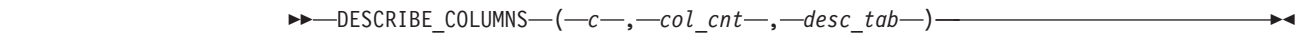
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DESCRIBE_COLUMNS プロシージャー - SELECT リストに含まれる列の記述の取得

DESCRIBE_COLUMNS プロシージャーは、カーソルから SELECT リストに含まれる列の記述を取得する機能を提供します。

構文



パラメーター

c 列が記述されるカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

col_cnt
カーソルの SELECT リスト内の列の数を戻す、タイプ INTEGER の出力引数。

desc_tab
列のメタデータを記述する、タイプ DESC_TAB の出力引数。 DESC_TAB 配列は、指定されたカーソルの各列の情報を提供します。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、ページ・サイズが 4K の USER TEMPORARY 表スペースを必要とします。この要件が満たされない場合は SQL0286N エラーを戻します。USER TEMPORARY 表スペースは、次のコマンドで作成できます。

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB は、列情報の入った DESC_REC レコードの配列です。

表 25. DESC_REC レコードによる DESC_TAB の定義

レコード名	説明
col_type	C および C++ 組み込み SQL アプリケーションにおいてサポートされる SQL データ・タイプで定義されている SQL データ・タイプ。
col_max_len	列の最大長。
col_name	列名。

表 25. *DESC_REC* レコードによる *DESC_TAB* の定義 (続き)

レコード名	説明
col_name_len	列名の長さ。
col_schema	常に NULL。
col_schema_name_len	常に NULL。
col_precision	データベースで定義されている列の精度。col_type がグラフィックまたは DBCLOB SQL データ・タイプを示す場合、この変数は列が保持できる 2 バイト文字の最大数を表します。
col_scale	データベースで定義されている列の位取り (DECIMAL、NUMERIC、TIMESTAMP にのみ適用)。
col_charsetid	常に NULL。
col_charsetform	常に NULL。
col_null_ok	NULL 可能標識。列が NULL 可能の場合は値が 1 に、それ以外の場合は 0 になります。

```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC AS ROW
(
  col_type INTEGER,
  col_max_len INTEGER,
  col_name VARCHAR(128),
  col_name_len INTEGER,
  col_schema_name VARCHAR(128),
  col_schema_name_len INTEGER,
  col_precision INTEGER,
  col_scale INTEGER,
  col_charsetid INTEGER,
  col_charsetform INTEGER,
  col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB AS DESC_REC ARRAY[INTEGER];
```

例

例 1: 次の例は、「EMP」表の empno、ename、hiredate、および sal 列を記述します。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE handle INTEGER;
  DECLARE col_cnt INTEGER;
  DECLARE col DBMS_SQL.DESC_TAB;
  DECLARE i INTEGER DEFAULT 1;
  DECLARE CUR1 CURSOR FOR S1;

  CALL DBMS_SQL.OPEN_CURSOR( handle );
  CALL DBMS_SQL.PARSE( handle,
      'SELECT empno, firstnme, lastname, salary
        FROM employee', DBMS_SQL.NATIVE );
  CALL DBMS_SQL.DESCRIBE_COLUMNS( handle, col_cnt, col );

  IF col_cnt > 0 THEN
    CALL DBMS_OUTPUT.PUT_LINE( 'col_cnt = ' || col_cnt );
```

```
    CALL DBMS_OUTPUT.NEW_LINE();
    fetchLoop: LOOP
      IF i > col_cnt THEN
        LEAVE fetchLoop;
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'i = ' || i );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name = ' || col[i].col_name );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name_len = ' ||
          NVL(col[i].col_name_len, 'NULL') );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name = ' ||
          NVL( col[i].col_schema_name, 'NULL' ) );

      IF col[i].col_schema_name_len IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = ' ||
            col[i].col_schema_name_len);
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_type = ' || col[i].col_type );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_max_len = ' || col[i].col_max_len );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_precision = ' || col[i].col_precision );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_scale = ' || col[i].col_scale );

      IF col[i].col_charsetid IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = ' || col[i].col_charsetid );
      END IF;

      IF col[i].col_charsetform IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = ' || col[i].col_charsetform );
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_null_ok = ' || col[i].col_null_ok );
      CALL DBMS_OUTPUT.NEW_LINE();
      SET i = i + 1;
    END LOOP;
  END IF;
END@
```

出力:

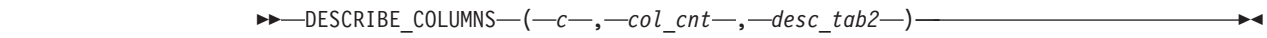
```
col_cnt = 4

i = 1
col[i].col_name = EMPNO
col[i].col_name_len = 5
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 452
col[i].col_max_len = 6
col[i].col_precision = 6
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 2
col[i].col_name = FIRSTNME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 12
```

```
col[i].col_precision = 12
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 3
col[i].col_name = LASTNAME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 15
col[i].col_precision = 15
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 4
col[i].col_name = SALARY
col[i].col_name_len = 6
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 484
col[i].col_max_len = 5
col[i].col_precision = 9
col[i].col_scale = 2
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 1
```

DESCRIBE_COLUMNS2 プロシージャー - SELECT リストに含まれる列名の記述の取得

DESCRIBE_COLUMNS2 プロシージャーは、カーソルから SELECT リストに含まれる列の記述を取得する機能を提供します。

構文



パラメーター

c
列が記述されるカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

col_cnt
カーソルの SELECT リスト内の列の数を戻す、タイプ INTEGER の出力引数。

desc_tab
列のメタデータを記述する、タイプ DESC_TAB2 の出力引数。 DESC_TAB2 配列は、指定されたカーソルの各列の情報を提供します。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、ページ・サイズが 4K の USER TEMPORARY 表スペースを必要とします。この要件が満たされない場合は SQL0286N エラーを戻します。USER TEMPORARY 表スペースは、次のコマンドで作成できます。

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB2 は、列情報の入った DESC_REC2 レコードの配列です。

表 26. DESC_REC2 レコードによる DESC_TAB2 の定義

レコード名	説明
col_type	C および C++ 組み込み SQL アプリケーションにおいてサポートされる SQL データ・タイプで定義されている SQL データ・タイプ。
col_max_len	列の最大長。
col_name	列名。
col_name_len	列名の長さ。
col_schema	常に NULL。
col_schema_name_len	常に NULL。
col_precision	データベースで定義されている列の精度。col_type がグラフィックまたは DBCLOB SQL データ・タイプを示す場合、この変数は列が保持できる 2 バイト文字の最大数を表します。
col_scale	データベースで定義されている列の位取り (DECIMAL、NUMERIC、TIMESTAMP にのみ適用)。
col_charsetid	常に NULL。
col_charsetform	常に NULL。
col_null_ok	NULL 可能標識。列が NULL 可能の場合は値が 1 に、それ以外の場合は 0 になります。

```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC2 AS ROW
(
col_type INTEGER,
col_max_len INTEGER,
col_name VARCHAR(128),
col_name_len INTEGER,
col_schema_name VARCHAR(128),
col_schema_name_len INTEGER,
col_precision INTEGER,
col_scale INTEGER,
col_charsetid INTEGER,
col_charsetform INTEGER,
col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB2 AS DESC_REC2 ARRAY[INTEGER];
```

EXECUTE プロシージャー - 解析済み SQL ステートメントの実行

EXECUTE 関数は、解析済み SQL ステートメントを実行します。

構文

パラメーター

c 実行する解析済み SQL ステートメントのカーソル ID を指定する、タイプ INTEGER の入力引数。

ret SQL コマンドが DELETE、INSERT、または UPDATE の場合に処理された行数を戻す、タイプ INTEGER の出力引数。それ以外の場合は 0 を戻します。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 次の無名ブロックは、「DEPT」表に行を挿入します。

```
SET SERVEROUTPUT ON@

CREATE TABLE dept (
  deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname )
)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

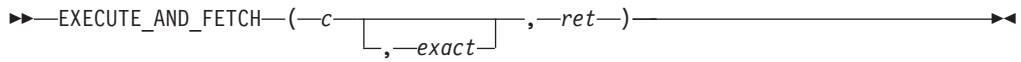
```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE dept
( deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
```

```
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname ) )
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
```

EXECUTE_AND_FETCH プロシージャー - 解析済み SELECT コマンドを実行して 1 行をフェッチする

EXECUTE_AND_FETCH プロシージャーは、解析済み SELECT コマンドを実行し、1 行をフェッチします。

構文



パラメーター

c 実行する SELECT コマンドのカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

exact
タイプ INTEGER のオプション引数。1 に設定した場合は、結果セットの行数が厳密に 1 に等しくなければ例外がスローされます。0 に設定すると、例外はスローされません。デフォルトは 0 です。*exact* が 1 に設定されていて結果セットに行がない場合は、NO_DATA_FOUND (SQL0100W) 例外がスローされます。 *exact* が 1 に設定されていて結果セットに複数の行がある場合は、TOO_MANY_ROWS (SQL0811N) 例外がスローされます。

ret 行が正常にフェッチされた場合は 1 を戻し、フェッチする行がない場合は 0 を戻す、タイプ INTEGER の出力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 次のストアード・プロシージャーは、EXECUTE_AND_FETCH 関数を使用して、従業員の名前で 1 人の従業員を検索します。従業員が見つからない場合、あるいは同じ名前の従業員が複数いる場合は、例外がスローされます。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@

CALL select_by_name( 'MARTIN' )@
```

この例では、以下の出力が結果として戻ります。

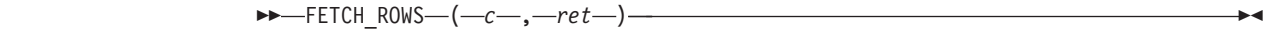
```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp
 ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
```

```
   ename    VARCHAR(10),
   job      VARCHAR(9),
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

CALL select_by_name( 'MARTIN' )

  Return Status = 0

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250.00
Commission: 1400.00
```

FETCH_ROWS プロシージャー - カーソルから行を取り出す

FETCH_ROWS 関数は、カーソルから行を取り出します。

構文



パラメーター

c 行がフェッチされるカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

ret 行が正常にフェッチされた場合は 1 を戻し、フェッチする行がない場合は 0 を戻す、タイプ INTEGER の出力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 以下の例は、「EMP」表から行をフェッチし、結果を表示します。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
```

```
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
        COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
       ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '    ' ||
            RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
            'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
            '9,999.99') || ' ' || TO_CHAR(NVL(v_comm, 0),
            '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

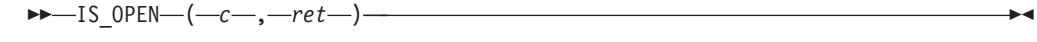
```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp (empno  DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename    VARCHAR(10), job      VARCHAR(9), mgr      DECIMAL(4),
    hiredate TIMESTAMP(0),
    sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
```

```
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
      COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
      ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
        10) || '  ' || TO_CHAR(v_hiredate,
        'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
        '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
         0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
```

IS_OPEN プロシージャー - カーソルがオープンしているかどうかを検査する

IS_OPEN 関数は、指定されたカーソルがオープンしているかどうかをテストする機能を提供します。

構文



パラメーター

c テストするカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

ret 指定されたファイルがオープンされている (TRUE) か、クローズされている (FALSE) かを示すタイプ BOOLEAN の出力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

LAST_ROW_COUNT プロシージャー - フェッチされた行の累積数を戻す

LAST_ROW_COUNT プロシージャーは、フェッチされた行の数を戻します。

構文

パラメーター

ret 現行セッションの今までにフェッチされた行の数を戻す、タイプ INTEGER の出力引数。 DBMS_SQL.PARSE を呼び出すと、カウンターがリセットされます。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 次の例は、LAST_ROW_COUNT プロシージャーを使用して、照会でフェッチされた行の総数を表示します。

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@
```

```
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
       10) || '  ' || TO_CHAR(v_hiredate,
       'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
       '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
       0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename    VARCHAR(10), job      VARCHAR(9),
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
```

```
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(
      v_empno || '   ' || RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
      'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
      '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
      0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

```
EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
Number of rows: 5
```

OPEN_CURSOR プロシージャー - カーソルのオープン

OPEN_CURSOR プロシージャーは、新しいカーソルを作成します。

動的 SQL ステートメントを解析して実行するには、カーソルを使用する必要があります。いったんオープンされたカーソルは、同じ SQL ステートメントや異なる SQL ステートメントで再度使用できます。再度使用するためにカーソルをクローズして再オープンする必要はありません。

構文

パラメーター

c 新しく作成されるカーソルのカーソル ID を指定する、タイプ INTEGER の出力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 次の例は、新しいカーソルを作成します。

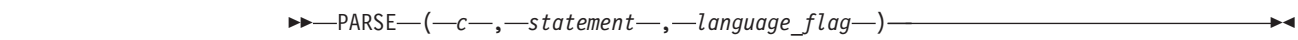
```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
END;
```

PARSE プロシージャー - SQL ステートメントの解析

PARSE プロシージャーは、SQL ステートメントを解析します。

SQL コマンドが DDL コマンドの場合には直ちに実行されるので、EXECUTE プロシージャーを実行する必要はありません。

構文



パラメーター

c
オープン・カーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

statement
解析対象の SQL ステートメント。

language_flag
Oracle 構文互換のための引数です。値として 1 または DBMS_SQL.native を使用してください。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

例

例 1: 次の無名ブロックは、job という名前の表を作成します。DDL ステートメントは PARSE プロシージャーによって直ちに実行されるので、別個の EXECUTE ステップを必要としないことに注意してください。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3),
     ' || 'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3), ' ||
    'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

例 2: 次の例は、job 表に 2 つの行を挿入します。

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
```

```
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
Number of rows processed: 1
```

例 3: 次の無名ブロックは、DBMS_SQL モジュールを使用して、2 つの INSERT ステートメントを含んだブロックを実行します。前の例では個々の INSERT ステートメントに終了セミコロンがなかったのに対し、ブロックの終わりに終了セミコロンがあることに注意してください。

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); '
       || 'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

この例では、以下の出力が結果として戻ります。

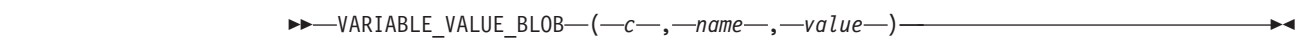
```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); ' ||
    'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
```

```
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

VARIABLE_VALUE_BLOB プロシージャー - BLOB INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_BLOB プロシージャーは、BLOB INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ BLOB(2G) の出力引数。

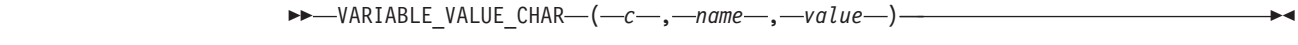
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_CHAR プロシージャー - CHAR INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_CHAR プロシージャーは、CHAR INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ CHAR(254) の出力引数。

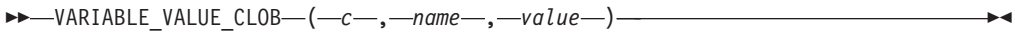
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_CLOB プロシージャー - CLOB INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_CLOB プロシージャーは、CLOB INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ CLOB(2G) の出力引数。

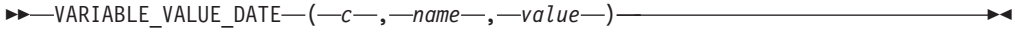
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_DATE プロシージャー - DATE INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_DATE プロシージャーは、DATE INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ DATE の出力引数。

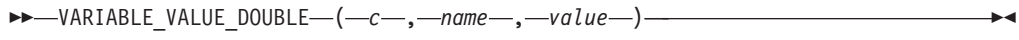
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_DOUBLE プロシージャー - DOUBLE INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_DOUBLE プロシージャーは、DOUBLE INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ DOUBLE の出力引数。

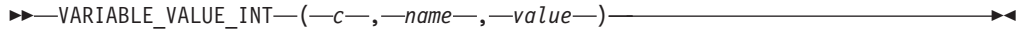
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_INT プロシージャー - INTEGER INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_INT プロシージャーは、INTEGER INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ INTEGER の出力引数。

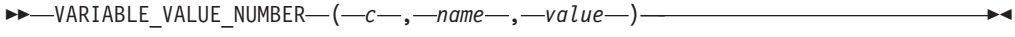
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_NUMBER プロシージャー - DECFLOAT INOUT または OUT パラメーターの値を戻す

VARIABLE_VALUE_NUMBER プロシージャーは、DECFLOAT INOUT または OUT パラメーターの値を戻す機能を提供します。

構文



パラメーター

c バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ DECFLOAT の出力引数。

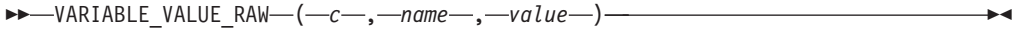
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_RAW プロシージャー - BLOB(32767) INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_RAW プロシージャーは、BLOB(32767) INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ BLOB(32767) の出力引数。

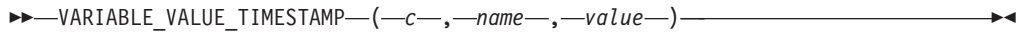
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_TIMESTAMP プロシージャー - TIMESTAMP INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_TIMESTAMP プロシージャーは、TIMESTAMP INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ TIMESTAMP の出力引数。

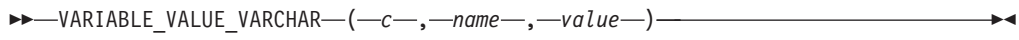
許可

DBMS_SQL モジュールに対する EXECUTE 特権。

VARIABLE_VALUE_VARCHAR プロシージャー - VARCHAR INOUT または OUT パラメーターの値を返す

VARIABLE_VALUE_VARCHAR プロシージャーは、VARCHAR INOUT または OUT パラメーターの値を返す機能を提供します。

構文



パラメーター

c
バインド変数を戻すカーソルのカーソル ID を指定する、タイプ INTEGER の入力引数。

name
バインド変数の名前を指定する入力引数。

value
値を受け取る変数を指定する、タイプ VARCHAR(32672) の出力引数。

許可

DBMS_SQL モジュールに対する EXECUTE 特権。

DBMS_UTILITY モジュール

DBMS_UTILITY モジュールは、さまざまなユーティリティー・プログラムを提供します。

このモジュールのスキーマは SYSIBMADM です。

DBMS_UTILITY モジュールには、以下のルーチンが含まれています。

表 27. DBMS_UTILITY モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
ANALYZE_DATABASE プロシージャー	データベースの表、クラスター、および索引を分析します。
ANALYZE_PART_OBJECT プロシージャー	パーティション表またはパーティション化索引を分析します。
ANALYZE_SCHEMA プロシージャー	スキーマの表、クラスター、および索引を分析します。
CANONICALIZE プロシージャー	ストリングを正規化します (例えば、空白文字を取り除く)。
COMMA_TO_TABLE プロシージャー	名前のコンマ区切りリストを名前の表に変換します。
COMPILE_SCHEMA プロシージャー	スキーマ内のプログラムをコンパイルします。
DB_VERSION プロシージャー	データベース・バージョンを取得します。
EXEC_DDL_STATEMENT プロシージャー	DDL ステートメントを実行します。
GET_CPU_TIME 関数	現在の CPU 時間を取得します。
GET_DEPENDENCY プロシージャー	指定されたオブジェクトに従属するオブジェクトを取得します。
GET_HASH_VALUE 関数	ハッシュ値を計算します。
GET_TIME 関数	現在時刻を取得します。
NAME_RESOLVE プロシージャー	指定された名前を解決します。
NAME_TOKENIZE プロシージャー	指定された名前をその構成要素部分に解析します。
TABLE_TO_COMMA プロシージャー	名前の表をコンマ区切りリストに変換します。
VALIDATE プロシージャー	無効データベース・オブジェクトを有効にします。

次の表は、DBMS_UTILITY モジュールで使用可能なシステム定義の変数とタイプを示しています。

表 28. DBMS_UTILITY パブリック変数

パブリック変数	データ・タイプ	説明
lname_array	TABLE	長い名前のリスト用。
uncl_array	TABLE	ユーザーおよび名前のリスト用。

LNAME_ARRAY は、完全修飾名など長い名前のリストの格納用です。

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE LNAME_ARRAY AS VARCHAR(4000) ARRAY[];
```

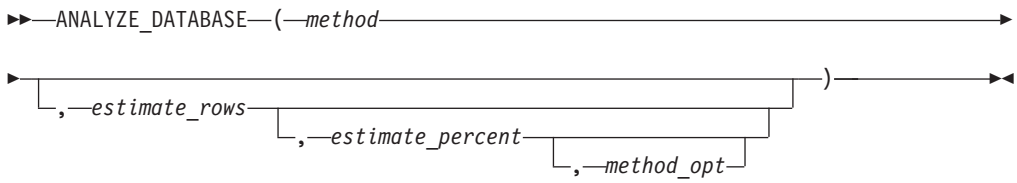
UNCL_ARRAY は、ユーザーおよび名前のリストの格納用です。

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE UNCL_ARRAY  AS VARCHAR(227)  ARRAY[];
```

ANALYZE_DATABASE プロシージャー - 表、クラスター、および索引に関する統計の収集

ANALYZE_DATABASE プロシージャーは、データベースの表、クラスター、および索引に関する統計を収集する機能を提供します。

構文



パラメーター

メソッド
実行する分析機能のタイプを指定する、タイプ VARCHAR(128) の入力引数。有効な値は以下のとおりです。

- ESTIMATE - *estimate_rows* で指定された行数または *estimate_percent* で指定された行のパーセンテージのどちらかに基づいて、見積もり統計を収集します。
- COMPUTE - 正確な統計を計算します。
- DELETE - データ・ディクショナリーから統計を削除します。

estimate_rows
見積もり統計のベースとする行数を指定する、タイプ INTEGER のオプション入力引数。 *method* を ESTIMATE にする場合は、*estimate_rows* か *estimate_percent* のどちらかを指定する必要があります。デフォルト値は NULL です。

estimate_percent
見積もり統計のベースとする行のパーセンテージを指定する、タイプ INTEGER のオプション入力引数。 *method* を ESTIMATE にする場合は、*estimate_rows* か *estimate_percent* のどちらかを指定する必要があります。デフォルト値は NULL です。

method_opt
分析するオブジェクト・タイプを指定する、タイプ VARCHAR(1024) のオプション入力引数。以下のキーワードの任意の組み合わせが有効です。

- [FOR TABLE]
- [FOR ALL [INDEXED] COLUMNS] [SIZE n]
- [FOR ALL INDEXES]

デフォルトは NULL です。

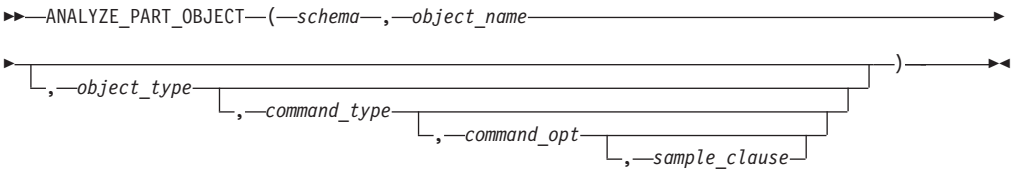
許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

ANALYZE_PART_OBJECT プロシージャー - パーティション表またはパーティション化索引に関する統計の収集

ANALYZE_PART_OBJECT プロシージャーは、パーティション化された表または索引に関する統計を収集する機能を提供します。

構文



パラメーター

schema
オブジェクトが分析されるスキーマのスキーマ名を指定する、タイプ VARCHAR(128) の入力引数。

object_name
分析対象のパーティション化オブジェクトの名前を指定する、タイプ VARCHAR(128)の入力引数。

object_type
分析対象オブジェクトのタイプを指定する、タイプ CHAR のオプション入力引数。有効な値は以下のとおりです。

- T – 表
- I – 索引

デフォルトは T です。

command_type
実行する分析機能のタイプを指定する、タイプ CHAR のオプション入力引数。有効な値は以下のとおりです。

- E - *sample_clause* 節で指定された行数または行のパーセンテージに基づいて、見積もり統計を収集します。
- C - 正確な統計を計算します。
- V - パーティションの構造と整合性の妥当性検査を行います。

デフォルト値は E です。

command_opt
統計計算のオプションを指定する、タイプ VARCHAR(1024) のオプション入力引数。 *command_type* が E または C の場合、以下のキーワードの任意の組み合わせをこの引数に使用できます。

- [FOR TABLE]

- [FOR ALL COLUMNS]
- [FOR ALL LOCAL INDEXES]

command_type が V の場合は、*object_type* が T のときに、この引数に CASCADE を指定できます。デフォルト値は NULL です。

sample_clause
タイプ VARCHAR(128) のオプション入力引数。 *command_type* が E の場合に、この引数に以下の節を含めます。この節は、見積もりのベースとする行数または行のパーセンテージを指定します。

```
SAMPLE n { ROWS | PERCENT }
```

デフォルト値は `SAMPLE 5 PERCENT` です。

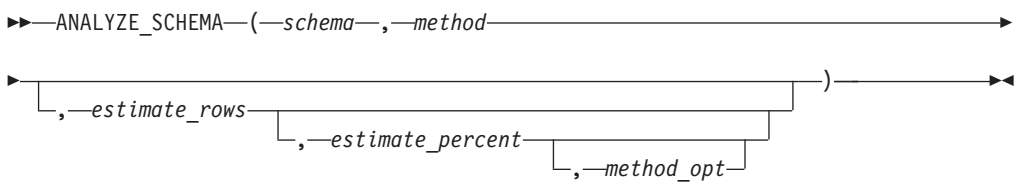
許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

ANALYZE_SCHEMA プロシージャー - スキーマの表、クラスター、および索引についての統計の収集

ANALYZE_SCHEMA プロシージャーは、指定したスキーマに含まれる表、クラスター、および索引の統計を収集する機能を提供します。

構文



パラメーター

schema
オブジェクトが分析されるスキーマのスキーマ名を指定する、タイプ VARCHAR(128) の入力引数。

メソッド
実行する分析機能のタイプを指定する、タイプ VARCHAR(128) の入力引数。有効な値は以下のとおりです。

- ESTIMATE - *estimate_rows* で指定された行数または *estimate_percent* で指定された行のパーセンテージのどちらかに基づいて、見積もり統計を収集します。
- COMPUTE - 正確な統計を計算します。
- DELETE - データ・ディクショナリーから統計を削除します。

estimate_rows
見積もり統計のベースとする行数を指定する、タイプ INTEGER のオプション

入力引数。 *method* を ESTIMATE にする場合は、*estimate_rows* か *estimate_percent* のどちらかを指定する必要があります。デフォルト値は NULL です。

estimate_percent
見積もり統計のベースとする行のパーセンテージを指定する、タイプ INTEGER のオプション入力引数。 *method* を ESTIMATE にする場合は、*estimate_rows* か *estimate_percent* のどちらかを指定する必要があります。デフォルト値は NULL です。

method_opt
分析するオブジェクト・タイプを指定する、タイプ VARCHAR(1024) のオプション入力引数。以下のキーワードの任意の組み合わせが有効です。

- [FOR TABLE]
- [FOR ALL [INDEXED] COLUMNS] [SIZE n]
- [FOR ALL INDEXES]

デフォルトは NULL です。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

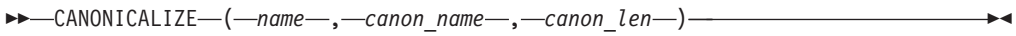
CANONICALIZE プロシージャー - ストリングの正規化

CANONICALIZE プロシージャーは、入力ストリングに対してさまざまな操作を実行します。

CANONICALIZE プロシージャーは、入力ストリングに対して以下の操作を実行します。

- ストリングが二重引用符で囲まれていない場合、許可されている ID の文字が使用されていることを検査します。正当な文字が使用されていない場合、例外がスローされます。ストリングが二重引用符で囲まれている場合、すべての文字が許可されます。
- ストリングが二重引用符で囲まれておらず、ピリオドが含まれていない場合、すべての英字が大文字に変換され、前後のスペースが除去されます。
- ストリングが二重引用符で囲まれていて、ピリオドが含まれていない場合、二重引用符が除去されます。
- ストリングにピリオドが含まれていて、ストリングのどの部分も二重引用符で囲まれていない場合、ストリングの各部分が大文字に変換され、各部分が二重引用符で囲まれます。
- ストリングにピリオドが含まれていて、ストリングのいくつかの部分が二重引用符で囲まれている場合、二重引用符の部分は二重引用符を含めて変更されずに返され、二重引用符で囲まれていない部分は大文字で返され、二重引用符で囲まれます。

構文



パラメーター

name
正規化されるストリングを指定する、タイプ VARCHAR(1024) の入力引数。

canon_name
正規化されたストリングを返す、タイプ VARCHAR(1024) の出力引数。

canon_len
name の中の先頭文字から何バイトを正規化するかを指定する、INTEGER のタイプの入力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下のプロシージャーは、その入力パラメーターに CANONICALIZE プロシージャーを適用し、結果を表示します。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END@

CALL canonicalize('Identifier')@
CALL canonicalize('"Identifier"')@
CALL canonicalize('"_+142%"')@
CALL canonicalize('abc.def.ghi')@
CALL canonicalize('"abc.def.ghi"')@
CALL canonicalize('"abc".def."ghi"')@
CALL canonicalize('"abc.def".ghi')@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END
DB20000I  The SQL command completed successfully.

CALL canonicalize('Identifier')

  Return Status = 0

Canonicalized name ==>IDENTIFIER<==
```

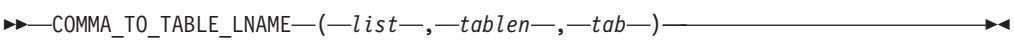
```
Length: 10

CALL canonicalize('"Identifier"')

  Return Status = 0

Canonicalized name ==>Identifier<==
Length: 10

CALL canonicalize('"_+142%"')

  Return Status = 0

Canonicalized name ==>_+142%<==
Length: 6

CALL canonicalize('abc.def.ghi')

  Return Status = 0

Canonicalized name ==>"ABC"."DEF"."GHI"<==
Length: 17

CALL canonicalize('"abc.def.ghi"')

  Return Status = 0

Canonicalized name ==>abc.def.ghi<==
Length: 11

CALL canonicalize('"abc".def."ghi"')

  Return Status = 0

Canonicalized name ==>"abc"."DEF"."ghi"<==
Length: 17

CALL canonicalize('"abc.def".ghi')

  Return Status = 0

Canonicalized name ==>"abc.def"."GHI"<==
Length: 15
```

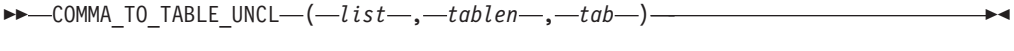
COMMA_TO_TABLE プロシージャー - 名前のコンマ区切りリストから名前の表への変換

COMMA_TO_TABLE プロシージャーは、名前のコンマ区切りリストを名前の配列に変換します。リスト内の各項目は、配列の要素になります。

注: 名前は、有効な ID としてフォーマット設定されている必要があります。

構文





パラメーター

list 名前のコンマ区切りリストを指定する、タイプ VARCHAR(32672) の入力引数。

tablen
tab 内の項目数を指定する、タイプ INTEGER の出力引数。

タブ
list 内に個々の名前の表が入る、タイプ LNAME_ARRAY または UNCL_ARRAY の出力引数。 *tab* の説明については、LNAME_ARRAY または UNCL_ARRAY を参照してください。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下のプロシージャーは、COMMA_TO_TABLE_LNAME プロシージャーを使用して名前のリストを表に変換します。その後、表の項目が表示されます。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
  END;
END@

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

この例では、以下の出力が結果として戻ります。

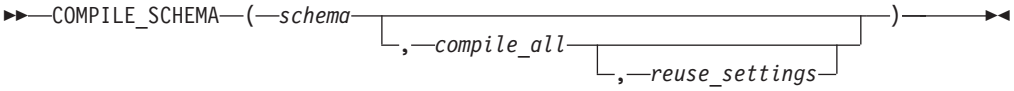
```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
```

```
  END;
END
DB20000I  The SQL command completed successfully.

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

sample_schema.dept
sample_schema.emp
sample_schema.jobhist
```

COMPILE_SCHEMA プロシージャー - スキーマ内のすべての関数、プロシージャー、トリガー、およびパッケージのコンパイル

COMPILE_SCHEMA プロシージャーは、スキーマ内のすべての関数、プロシージャー、トリガー、およびパッケージを再コンパイルする機能を提供します。

構文



パラメーター

schema
プログラムが再コンパイルされるスキーマを指定する、タイプ VARCHAR(128) の入力引数。

compile_all
オプションのタイプ BOOLEAN の入力引数。これを `false` (現在無効状態のプログラムだけをプロシージャーが再コンパイルするという意味) に設定する必要があります。

reuse_settings
オプションのタイプ BOOLEAN の入力引数。これを `false` (プロシージャーが現行セッション設定値を使用するという意味) に設定する必要があります。

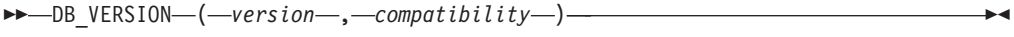
許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

DB_VERSION プロシージャー - データベース・バージョンの取得

DB_VERSION プロシージャーは、データベースのバージョン番号を返します。

構文



パラメーター

version
データベースのバージョン番号を返す、タイプ VARCHAR(1024) の出力引数。

互換性
データベースの互換性設定を返す、タイプ VARCHAR(1024) の出力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下の無名ブロックは、データベースのバージョン情報を表示します。

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END@
```

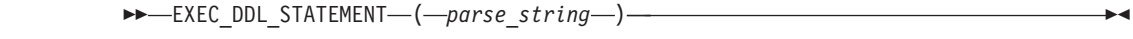
この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END
DB20000I  The SQL command completed successfully.

Version: DB2 v9.7.0.0
Compatibility: DB2 v9.7.0.0
```

EXEC_DDL_STATEMENT プロシージャー - DDL ステートメントの実行

EXEC_DDL_STATEMENT プロシージャーは、DDL コマンドを実行する機能を提供します。

構文



パラメーター

parse_string
実行する DDL コマンドを指定する、タイプ VARCHAR(1024) の入力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下の無名ブロックは、ジョブ表を作成します。

```
BEGIN
  CALL DBMS_UTILITY.EXEC_DDL_STATEMENT(
    'CREATE TABLE job (' ||
    'jobno DECIMAL(3),' ||
    'jname VARCHAR(9))' );
END@
```

GET_CPU_TIME 関数 - 現在 CPU 時間の取得

GET_CPU_TIME 関数は、任意の時点からの CPU 時間を、100 分の 1 秒単位で返します。

構文

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

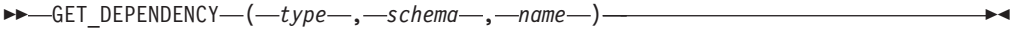
例 1: 以下の SELECT コマンドは、現在の CPU 時間を取得します。

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
-------------
          603
```

GET_DEPENDENCY プロシージャー - 特定のオブジェクトに依存するオブジェクトのリスト

GET_DEPENDENCY プロシージャーは、特定のオブジェクトに依存するすべてのオブジェクトをリストする機能を提供します。

構文



パラメーター

type
: *name* のオブジェクト・タイプを指定する、タイプ VARCHAR(128) の入力引数。有効な値は、FUNCTION、INDEX、LOB、PACKAGE、PACKAGE BODY、PROCEDURE、SEQUENCE、TABLE、TRIGGER、および VIEW です。

schema
name が存在するスキーマの名前を指定する、タイプ VARCHAR(128) の入力引数。

name
従属関係を取得する対象となるオブジェクトの名前を指定する、タイプ VARCHAR (128) の入力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下の無名ブロックは、表 T1 および関数 FUNC1 の従属関係を検出します。

```
SET SERVEROUTPUT ON@

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)@

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END@

CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END@

CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')@
CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END
DB20000I  The SQL command completed successfully.
```

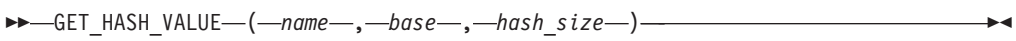
```
CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')

  Return Status = 0

DEPENDENCIES ON SCHEMA2.FUNC1
---------------------------------------------------------------------
*FUNCTION SCHEMA2.FUNC1()
*   FUNCTION SCHEMA3 .FUNC2()

CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')

  Return Status = 0

DEPENDENCIES ON SCHEMA1.T1
---------------------------------------------------------------------
*TABLE SCHEMA1.T1()
*   FUNCTION SCHEMA3 .FUNC2()
```

GET_HASH_VALUE 関数 - 特定のストリングについてのハッシュ値の計算

GET_HASH_VALUE 関数は、特定のストリングについてハッシュ値を計算する機能を提供します。

この関数は、タイプ INTEGER で生成されたハッシュ値を返します。この値はプラットフォームに依存します。

構文



パラメーター

name
ハッシュ値の計算対象となるストリングを指定する、タイプ VARCHAR(32672) の入力引数。

base
ハッシュ値が生成されるときの開始値を指定する、タイプ INTEGER の入力引数。

hash_size
対象となるハッシュ・テーブルにおけるハッシュ値の数を指定する、タイプ INTEGER の入力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下の例は、2 つのストリングのハッシュ値を戻します。ハッシュ値の開始値は 100 で、最大値は 1024 (重複しない固有の値) です。

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
```

この例では、以下の出力が結果として戻ります。

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 343

  1 record(s) selected.


SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 760

  1 record(s) selected.
```

GET_TIME 関数 - 現在時刻を返す

GET_TIME 関数は、現在時刻を 100 分の 1 秒単位で返す機能を提供します。

構文

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

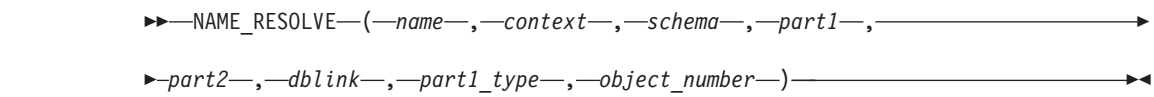
例 1: 以下の例は、GET_TIME 関数の呼び出しを示しています。

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1556037
```

NAME_RESOLVE プロシージャー - データベース・オブジェクトについてのスキーマおよびその他のメンバーシップ情報の取得

NAME_RESOLVE プロシージャーは、データベース・オブジェクトのスキーマおよびその他のメンバーシップ情報を取得する機能を提供します。同義語は、それらの基本オブジェクトに解決されます。

構文



パラメーター

name
解決するデータベース・オブジェクトの名前を指定する、タイプ VARCHAR (1024) の入力引数。次の形式で指定できます。

```
[[ a.]b.]c[@dblink ]
```

context
タイプ INTEGER の入力引数。以下の値のいずれかに設定されます。

- 1 - 関数、プロシージャー、またはモジュールの名前を解決
- 2 - 表、ビュー、シーケンス、または同義語の名前を解決
- 3 - トリガー名を解決

schema
name によって指定されたオブジェクトが入るスキーマの名前を指定する、タイプ VARCHAR(128) の出力引数。

part1
解決された表、ビュー、シーケンス、トリガー、またはモジュールの名前を指定する、タイプ VARCHAR(128) の出力引数。

part2
解決された関数またはプロシージャー (モジュール内の関数およびプロシージャーを含む) の名前を指定する、タイプ VARCHAR(128) の出力引数。

dblink
データベース・リンクの名前を指定する、タイプ VARCHAR(128) の出力引数 (`@dblink` が *name* の中で指定されている場合)。

part1_type
タイプ INTEGER の出力引数。以下の値を返します。

- 2 - 解決済みオブジェクトは表です
- 4 - 解決済みオブジェクトはビューです
- 6 - 解決済みオブジェクトはシーケンスです
- 7 - 解決済みオブジェクトはストアード・プロシージャーです
- 8 - 解決済みオブジェクトはストアード関数です
- 9 - 解決済みオブジェクトは、モジュール、またはモジュール内の関数またはプロシージャーです
- 12 - 解決済みオブジェクトはトリガーです

object_number
解決済みデータベース・オブジェクトのオブジェクト ID を指定する、タイプ INTEGER の出力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下のストアード・プロシージャーは、さまざまなデータベース・オブジェクトに対する NAME_RESOLVE プロシージャーの戻り値を表示するために使用されます。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END@

DROP TABLE S1.T1@
CREATE TABLE S1.T1 (C1 INT)@

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END@

CREATE OR REPLACE MODULE S3.M1@
ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END@

CALL NAME_RESOLVE( 'S1.T1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 1 )@
CALL NAME_RESOLVE( 'PROC1', 1 )@
CALL NAME_RESOLVE( 'M1', 1 )@
CALL NAME_RESOLVE( 'S3.M1.F1', 1 )@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END
DB20000I  The SQL command completed successfully.

DROP TABLE S1.T1
DB20000I  The SQL command completed successfully.

CREATE TABLE S1.T1 (C1 INT)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE MODULE S3.M1
DB20000I  The SQL command completed successfully.

ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END
DB20000I  The SQL command completed successfully.

CALL NAME_RESOLVE( 'S1.T1', 2 )

  Return Status = 0

name      : S1.T1
context   : 2
schema    : S1
part1     : T1
```

```
part2     : NULL
dblink    : NULL
part1 type: 2
object id : 8

CALL NAME_RESOLVE( 'S2.PROC1', 2 )
SQL0204N  "S2.PROC1" is an undefined name.  SQLSTATE=42704

CALL NAME_RESOLVE( 'S2.PROC1', 1 )

  Return Status = 0

name      : S2.PROC1
context   : 1
schema    : S2
part1     : PROC1
part2     : NULL
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'PROC1', 1 )

  Return Status = 0

name      : PROC1
context   : 1
schema    : S2
part1     : NULL
part2     : PROC1
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'M1', 1 )

  Return Status = 0

name      : M1
context   : 1
schema    : S3
part1     : NULL
part2     : M1
dblink    : NULL
part1 type: 9
object id : 16

CALL NAME_RESOLVE( 'S3.M1.F1', 1 )

  Return Status = 0

name      : S3.M1.F1
context   : 1
schema    : S3
part1     : M1
part2     : F1
dblink    : NULL
part1 type: 9
object id : 16
```

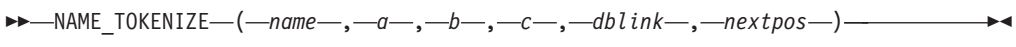
例 2: データベース・リンクによってアクセスされた表を解決します。
NAME_RESOLVE は、リモート・データベースにあるデータベース・オブジェクトの妥当性を検査しない点に注意してください。これは、*name* 引数で指定されたコンポーネントをエコー出力して戻すだけです。

```
BEGIN
    name_resolve('sample_schema.emp@sample_schema_link',2);
END;

name      : sample_schema.emp@sample_schema_link
context   : 2
schema    : SAMPLE_SCHEMA
part1     : EMP
part2     :
dblink    : SAMPLE_SCHEMA_LINK
part1 type: 0
object id : 0
```

NAME_TOKENIZE プロシージャー - 特定の名前をその構成要素のパーツに構文解析する

NAME_TOKENIZE プロシージャーは、名前をその構成要素のパーツに構文解析します。二重引用符がない名前は大文字に変換され、二重引用符がある名前からは二重引用符が除去されます。

構文



パラメーター

name
名前が入ったストリングを次の形式で指定する、タイプ VARCHAR(1024) の入力引数。

```
a[.b[.c]][@dblink ]
```

a 左端の構成要素を返す、タイプ VARCHAR(128) の出力引数。

b 2 番目の構成要素がある場合にはそれを返す、タイプ VARCHAR(128) の出力引数。

c 3 番目の構成要素がある場合にはそれを返す、タイプ VARCHAR(128) の出力引数。

dblink
データベース・リンク名を返す、タイプ VARCHAR(32672) の出力引数。

nextpos
name 内で構文解析された最後の文字の位置を指定する、タイプ INTEGER の出力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下のストアード・プロシージャーは、さまざまな名前に対する NAME_TOKENIZE プロシージャーの戻り値を表示するために使用されます。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_tokenize(
```

```
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END@

CALL name_tokenize( 'b' )@
CALL name_tokenize( 'a.b' )@
CALL name_tokenize( '"a".b.c' )@
CALL name_tokenize( 'a.b.c@d' )@
CALL name_tokenize( 'a.b."c"@"d"' )@
```

この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE name_tokenize(
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
```

```
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END
DB20000I  The SQL command completed successfully.

CALL name_tokenize( 'b' )

  Return Status = 0

name   : b
a      : B
b      : NULL
c      : NULL
dblink : NULL
nextpos: 1

CALL name_tokenize( 'a.b' )

  Return Status = 0

name   : a.b
a      : A
b      : B
c      : NULL
dblink : NULL
nextpos: 3

CALL name_tokenize( '"a".b.c' )

  Return Status = 0

name   : "a".b.c
a      : a
b      : B
c      : C
dblink : NULL
nextpos: 7

CALL name_tokenize( 'a.b.c@d' )

  Return Status = 0

name   : a.b.c@d
a      : A
b      : B
c      : C
dblink : D
nextpos: 7

CALL name_tokenize( 'a.b."c"@"d"' )

  Return Status = 0
```

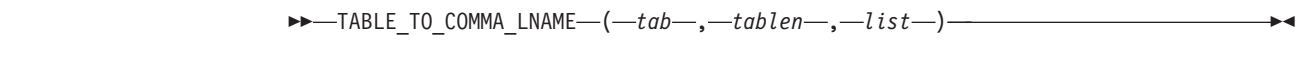
```
name   : a.b."c"@"d"
a      : A
b      : B
c      : c
dblink : d
nextpos: 11
```

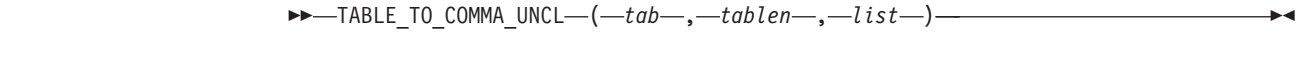
TABLE_TO_COMMA プロシージャー - 名前の表を名前のコンマ区切りリストに変換する

TABLE_TO_COMMA プロシージャーは、名前の配列を名前のコンマ区切りリストに変換します。各配列エレメントは、リスト項目になります。

注: 名前は、有効な ID としてフォーマット設定されている必要があります。

構文





パラメーター

タブ
名前が入った配列を指定する、タイプ LNAME_ARRAY または UNCL_ARRAY の入力引数。 *tab* の説明については、LNAME_ARRAY または UNCL_ARRAY を参照してください。

tablen
list 内の項目数を返す、タイプ INTEGER の出力引数。

list *tab* から名前のコンマ区切りリストを返す、タイプ VARCHAR(32672) の出力引数。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

例

例 1: 以下の例は、最初に COMMA_TO_TABLE_LNAME プロシージャーを使用してコンマ区切りリストを表に変換します。その後 TABLE_TO_COMMA_LNAME プロシージャーで表をコンマ区切りリストに変換して戻し、それを表示します。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
```

```
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END@

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

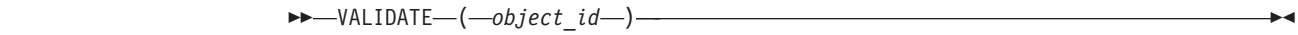
この例では、以下の出力が結果として戻ります。

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END
DB20000I  The SQL command completed successfully.

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

Table Entries
-------------
sample_schema.dept
sample_schema.emp
sample_schema.jobhist
-------------
Comma-Delimited List: sample_schema.dept,sample_schema.emp,sample_schema.jobhist
```

VALIDATE プロシージャー - 無効ルーチンの有効ルーチンへの変更

VALIDATE プロシージャーは、無効ルーチンの状態を有効に変更する機能を提供します。

構文



パラメーター

object_id
有効な状態に変更されるルーチンの ID を指定する、タイプ INTEGER の入力引数。 SYSCAT.ROUTINES ビューの ROUTINEID 列には、すべてのルーチン ID が入ります。

許可

DBMS_UTILITY モジュールに対する EXECUTE 特権。

MONREPORT モジュール

MONREPORT モジュールは、さまざまなモニター・データを取得してテキスト・レポートを生成するための一連のプロシージャーを備えています。

このモジュールのスキーマは SYSIBMADM です。

MONREPORT モジュールには、以下のシステム定義ルーチンが含まれています。

表 29. MONREPORT モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
341 ページの『CONNECTION プロシージャー - 接続メトリックに関するレポートの生成』	接続レポートは、それぞれの接続に関するモニター・データを表します。
342 ページの『CURRENTAPPS プロシージャー - ポイント・イン・タイムのアプリケーション処理メトリック・レポートの生成』	現行アプリケーション・レポートは、各接続に関連した作業単位、エージェント、アクティビティーの現在の瞬間的な処理状態を表します。レポートの冒頭ではすべての接続を合計した状態情報が示され、続いてそれぞれの接続の詳細セクションがあります。
342 ページの『CURRENTSQL プロシージャー - アクティビティーを要約するレポートの生成』	現行 SQL レポートは、現在実行中のアクティビティーをさまざまなメトリックで計測して、上位のアクティビティーをリストします。
343 ページの『DBSUMMARY プロシージャー - システムおよびアプリケーションのパフォーマンス・メトリックの要約レポートの生成』	要約レポートには、データベース全体に関する詳細なモニター・データと、それぞれの接続、ワークロード、サービス・クラス、データベース・メンバーに関する主要なパフォーマンス指標が含まれます。

表 29. *MONREPORT* モジュールで使用可能なシステム定義ルーチン *(続き)*

ルーチン名	説明
344 ページの『LOCKWAIT プロシージャー - 現在のロック待機のレポート生成』	ロック待機レポートには、現在進行中のそれぞれのロック待機についての情報が含まれます。ロックのホルダーとリクエスターに関する詳細、保持されているロックと要求されたロックの特性などの詳細が含まれています。
347 ページの『PKGCACHE プロシージャー - パッケージ・キャッシュ・メトリックの要約レポートの生成』	パッケージ・キャッシュ・レポートは、パッケージ・キャッシュに蓄積されたステートメントをさまざまメトリックで計測して、上位のステートメントをリストします。

使用上の注意

モニター・エレメント名は大文字で表示されます (例えば TOTAL_CPU_TIME)。モニター・エレメントについての詳細情報を調べるには、*DB2* インフォメーション・センターでモニター名を検索してください。

monitoring_interval 入力を使用するレポートの場合、レポート内の負の値は不正確です。ソース・データ・カウンターのロールオーバー中にこれが発生する可能性があります。正確な値を判別するには、ロールオーバーが完了した後でレポートを再実行してください。

注: レポートはモジュール内の SQL プロシージャーを使って実装されるため、パッケージ・キャッシュ構成の影響を受ける可能性があります。レポート実行時のパフォーマンスが遅い場合、パッケージ・キャッシュ構成を調べて、それがワークロードに対して十分であることを確認してください。詳細については、『pckcachesz - パッケージ・キャッシュ・サイズ構成パラメーター』を参照してください。

以下の例は MONREPORT ルーチンを呼び出すさまざまな方法を示しています。この例は MONREPORT.CONNECTION(*monitoring_interval*, *application_handle*) プロシージャーを示しています。オプション・パラメーターの値を入力したくない場合、次のような方法で扱うことができます。

- どんな場合でも NULL または DEFAULT を指定できます。
- 文字入力の場合、空ストリング ('') を指定できます。
- それが最後のパラメーターである場合は、省略できます。

デフォルト・モニター間隔 10 秒を使用して、接続ごとのセクションを含むレポートを生成するには、次のように MONREPORT.CONNECTION プロシージャーを呼び出します。

```
call monreport.connection()
```

デフォルト・モニター間隔 10 秒を使用して、アプリケーション・ハンドル 32 の接続のみに関するセクションを含むレポートを生成するには、以下のどちらの方法で MONREPORT.CONNECTION プロシージャーを呼び出すことも可能です。

```
call monreport.connection(DEFAULT, 32)
call monreport.connection(10, 32)
```

モニター間隔 60 秒を使用して、接続ごとのセクションを含むレポートを生成するには、以下のどちらの方法で MONREPORT.CONNECTION プロシージャーを呼び出すことも可能です。

```
call monreport.connection(60)

call monreport.connection(60, null)
```

デフォルトでは、このモジュール内のレポートは英語で生成されます。レポートを生成する際の言語を変更するには CURRENT LOCALE LC_MESSAGES 特殊レジスターを変更してください。例えば、CONNECTION レポートをフランス語で生成するには、以下のコマンドを発行します。

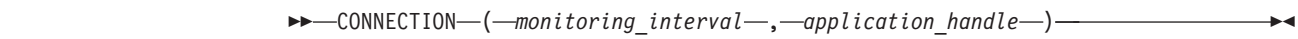
```
SET CURRENT LOCALE LC_MESSAGES = 'CLDR 1.5:fr_FR'
CALL MONREPORT.CONNECTION
```

CONNECTION プロシージャー - 接続メトリックに関するレポートの生成

CONNECTION プロシージャーは、各接続についてのモニター・データを収集して、テキスト形式のレポートを生成します。

CONNECTION プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

構文



パラメーター

monitoring_interval
オプションのタイプ INTEGER の入力引数。モニタリング・データが報告されるまでの収集期間 (秒) を指定します。例えば、モニタリング間隔 30 を指定した場合、ルーチンは表関数を呼び出して、30 秒待機した後、再び表関数を呼び出します。その後、このルーチンは (この期間中の変化を反映する) 差異を計算します。引数 *monitoring_interval* を指定しない場合、または NULL を指定した場合のデフォルト値は 10 です。有効な入力の範囲は 0 から 3600 (つまり最大で 1 時間) までの整数値です。

application_handle
接続を識別するアプリケーション・ハンドルを指定する、オプションのタイプ BIGINT の入力引数。引数 *application_handle* を指定しない場合、または NULL を指定した場合には、各接続についてのセクションがレポートに含まれます。デフォルトは NULL です。

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は CONNECTION プロシージャーを呼び出すさまざまな方法を示しています。最初の例は、10 秒間隔のデータを表す、すべての接続についてのレポートを生成します。

```
call monreport.connection;
```

次の例は、30 秒間隔のデータを表す、すべての接続についてのレポートを生成します。

```
call monreport.connection(30);
```

次の例は、アプリケーション・ハンドル 34 の接続に関するレポートを生成します。 (現在の間隔に基づく代わりに) ソース表関数で累積された絶対合計に基づいてデータが表示されます。

```
call monreport.connection(0, 34);
```

次の例は、アプリケーション・ハンドル 34 の接続に関するレポートを生成します。 10 秒間隔のデータが表示されます。

```
call monreport.connection(DEFAULT, 34);
```

CURRENTAPPS プロシージャー - ポイント・イン・タイムのアプリケーション処理メトリック・レポートの生成

CURRENTAPPS プロシージャーは、各接続に関連した作業単位、エージェント、アクティビティーの現在の瞬間的な処理状態についての情報を収集します。

CURRENTAPPS プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

構文

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は CURRENTAPPS プロシージャーを呼び出す方法を示しています。

```
call monreport.currentapps;
call monreport.currentapps();
```

CURRENTSQL プロシージャー - アクティビティーを要約するレポートの生成

CURRENTSQL プロシージャーは、現在実行中のアクティビティーを要約するテキスト形式のレポートを生成します。

CURRENTSQL プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

構文

パラメーター

member
タイプ SMALLINT の入力引数。特定のメンバーやパーティションに関するデータを表示するか、それとも全メンバーの合計データを表示するかを決定します。この引数を指定しない場合、または NULL を指定した場合には、全メンバーにわたって合計された値がレポートに表示されます。有効なメンバー番号を指定した場合、そのメンバーに関する値がレポートに表示されます。

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は CURRENTSQL プロシージャーを呼び出すさまざまな方法を示しています。最初の例は、すべてのメンバーにわたって集約されたアクティビティー・メトリックを示すレポートを生成します。

```
call monreport.currentsql;
```

次の例は、メンバー番号 4 のアクティビティー・パフォーマンスに関する特定のアクティビティー・メトリックを示すレポートを生成します。

```
call monreport.currentsql(4);
```

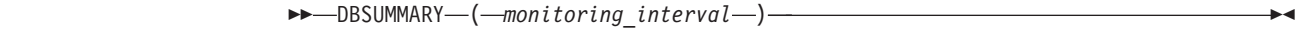
DBSUMMARY プロシージャー - システムおよびアプリケーションのパフォーマンス・メトリックの要約レポートの生成

DBSUMMARY プロシージャーは、システムおよびアプリケーションのパフォーマンス・メトリックを要約するテキスト形式のモニター・レポートを生成します。

DBSUMMARY プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

DB 要約レポートには、データベース全体に関する詳細なモニター・データと、それぞれの接続、ワークロード、サービス・クラス、データベース・メンバーに関する主要なパフォーマンス指標が含まれます。

構文



パラメーター

monitoring_interval
オプションのタイプ INTEGER の入力引数。モニタリング・データが報告されるまでの収集期間 (秒) を指定します。例えば、モニタリング間隔 30 を指定した場合、ルーチンは表関数を呼び出して、30 秒待機した後、再び表関数を呼び出します。その後、DBSUMMARY プロシージャーは (この期間中の変化を反映する) 差異を計算します。引数 *monitoring_interval* を指定しない場合、または NULL を指定した場合のデフォルト値は 10 です。有効な入力の範囲は 0 から 3600 (つまり最大で 1 時間) までの整数値です。

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は DBSUMMARY プロシージャーを呼び出すさまざまな方法を示しています。最初の例は、10 秒間隔のデータを表すレポートを生成します。

```
call monreport.dbsummary;
```

次の例は、30 秒間隔のデータを表すレポートを生成します。

```
call monreport.dbsummary(30);
```

LOCKWAIT プロシージャー - 現在のロック待機のレポート生成

ロック待機レポートには、現在進行中のそれぞれのロック待機についての情報が含まれます。ロックのホルダーとリクエスターについての情報、および保持されているロックと要求されたロックの特性などの詳細が含まれます。

LOCKWAIT プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

構文

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は LOCKWAIT プロシージャーを呼び出すさまざまな方法を示しています。

```
call monreport.lockwait;
call monreport.lockwait();
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Monitoring report - current lock waits
--------------------------------------------------------------------------------
Database:               SAMPLE
Generated:              08/28/2009 07:16:26

================================================================================
Part 1 - Summary of current lock waits

--------------------------------------------------------------------------------

      REQ_APPLICATION  LOCK_MODE  HLD_APPLICATION  LOCK_  LOCK_OBJECT_TYPE
#     HANDLE           REQUESTED  _HANDLE          MODE
----  ---------------  ---------  ---------------  -----  ----------------------
1     26               U          21               U      ROW
2     25               U          21               U      ROW
3     24               U          21               U      ROW
4     23               U          21               U      ROW
5     22               U          21               U      ROW
6     27               U          21               U      ROW

================================================================================

...


390 record(s) selected.

Return Status = 0
```

図 1. サンプル MONREPORT.LOCKWAIT 出力 - 要約セクション

```
================================================================================
Part 2: Details for each current lock wait

lock wait #:1
--------------------------------------------------------------------------------

-- Lock details --

LOCK_NAME             = 04000500040000000000000052
LOCK_WAIT_START_TIME = 2009-08-28-07.15.31.013802
LOCK_OBJECT_TYPE      = ROW
TABSCHEMA             = TRIPATHY
TABNAME               = INVENTORY
ROWID                 = 4
LOCK_STATUS           = W
LOCK_ATTRIBUTES       = 0000000000000000
ESCALATION            = N

-- Requestor and holder application details --

Attributes            Requestor                      Holder
-------------------   ----------------------------   ----------------------------
APPLICATION_HANDLE    26                             21
APPLICATION_ID        *LOCAL.tripathy.090828111531   *LOCAL.tripathy.090828111435
APPLICATION_NAME      java                           java
SESSION_AUTHID        TRIPATHY                       TRIPATHY
MEMBER                0                              0
LOCK_MODE             -                              U
LOCK_MODE_REQUESTED   U                              -

-- Lock holder current agents --

AGENT_TID             = 41
REQUEST_TYPE          = FETCH
EVENT_STATE           = IDLE
EVENT_OBJECT          = REQUEST
EVENT_TYPE            = WAIT
ACTIVITY_ID           =
UOW_ID                =

-- Lock holder current activities --

ACTIVITY_ID           = 1
UOW_ID                = 1
LOCAL_START_TIME      = 2009-08-28-07.14.31.079757
ACTIVITY_TYPE         = READ_DML
ACTIVITY_STATE        = IDLE

STMT_TEXT             =
select * from inventory for update

-- Lock requestor waiting agent and activity --

AGENT_TID             = 39
REQUEST_TYPE          = FETCH
ACTIVITY_ID           = 1
UOW_ID                = 1
LOCAL_START_TIME      = 2009-08-28-07.15.31.012935
ACTIVITY_TYPE         = READ_DML
ACTIVITY_STATE        = EXECUTING

STMT_TEXT             =
select * from inventory for update
```

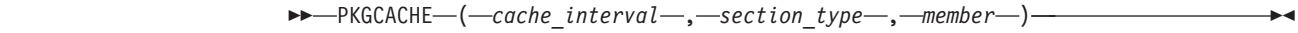
図 2. サンプル MONREPORT.LOCKWAIT 出力 - 詳細セクション

PKGCACHE プロシージャー - パッケージ・キャッシュ・メトリックの要約レポートの生成

パッケージ・キャッシュ要約レポートは、パッケージ・キャッシュに蓄積されたステートメントをさまざまメトリックで計測して、上位のステートメントをリストします。

PKGCACHE プロシージャーは DB2 バージョン 9.7 フィックスパック 1 以降で使用可能です。

構文



パラメーター

cache_interval
オプションのタイプ INTEGER の入力引数。*cache_interval* 値で指定される数値は、過去の何分間に更新されたパッケージ・キャッシュ項目に関するデータを限定的にレポートに含めるかを指定します。例えば *cache_interval* 値を 60 に設定すると、過去 60 分間に更新されたパッケージ・キャッシュ項目に基づくレポートが生成されます。有効な値は `0` から `10080` までの整数、つまり最大で 7 日間の間隔がサポートされます。この引数を指定しない場合、または NULL を指定した場合には、いつ追加または更新されたかに関わらず、パッケージ・キャッシュ項目のデータがレポートに含まれます。

section_type
オプションのタイプ CHAR(1) 入力引数。静的 SQL、動的 SQL、またはその両方のデータをレポートに含めることを指定します。この引数を指定しない場合、または NULL を指定した場合には、両方の種類の SQL データがレポートに含まれます。有効な値は `d` または `D` (動的)、および `s` または `S` (静的) です。

member
オプションのタイプ SMALLINT の入力引数。特定のメンバーやパーティションに関するデータを表示するか、それとも全メンバーの合計データを表示するかを決定します。この引数を指定しない場合、または NULL を指定した場合には、全メンバーにわたって合計された値がレポートに表示されます。有効なメンバー番号を指定した場合、そのメンバーに関する値がレポートに表示されます。

許可

以下の特権が必要です。

- MONREPORT モジュールに対する EXECUTE 特権

以下の例は PKGCACHE プロシージャーを呼び出すさまざまな方法を示しています。最初の例は、パッケージ・キャッシュ内のすべてのステートメントに基づき、すべてのメンバーにわたってデータが集約されたレポートを生成します。

```
call monreport.pkgcache;
```

次の例は、過去 30 分間にメトリックが更新されたパッケージ・キャッシュ内の動的ステートメントと静的ステートメントの両方に基づき、すべてのメンバーにわたってデータが集約されたレポートを生成します。

```
call monreport.pkgcache(30);
```

次の例は、パッケージ・キャッシュ内のすべての動的ステートメントに基づき、すべてのメンバーにわたってデータが集約されたレポートを生成します。

```
call monreport.pkgcache(DEFAULT, 'd');
```

次の例は、過去 30 分間にメトリックが更新されたパッケージ・キャッシュ内の動的ステートメントと静的ステートメントの両方に基づき、特定のメンバー番号 4 に関するデータが含まれるレポートを生成します。

```
call db2monreport.pkgcache(30, DEFAULT, 4);
```

UTL_DIR モジュール

UTL_DIR モジュールは、UTL_FILE モジュールで使用するディレクトリー別名を維持するための一連のルーチンを提供します。

注: UTL_DIR モジュールは、直接的なオペレーティング・システム呼び出し (例えば mkdir または rmdir コマンド) を発行しません。物理ディレクトリーの保守については、このモジュールでは扱われません。

このモジュールのスキーマは SYSIBMADM です。

UTL_DIR モジュールには、以下のシステム定義ルーチンが含まれています。

表 30. UTL_DIR モジュールで使用可能なシステム定義ルーチン

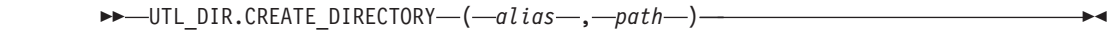
ルーチン名	説明
CREATE_DIRECTORY プロシージャー	指定されたパスのディレクトリー別名を作成します。
CREATE_OR_REPLACE_DIRECTORY プロシージャー	指定されたパスのディレクトリー別名を作成または置換します。
DROP_DIRECTORY プロシージャー	指定されたディレクトリー別名をドロップします。
GET_DIRECTORY_PATH プロシージャー	指定されたディレクトリー別名の対応パスを取得します。

CREATE_DIRECTORY プロシージャー - ディレクトリー別名の作成

CREATE_DIRECTORY プロシージャーは、指定されたパスのディレクトリー別名を作成します。

ディレクトリー情報は、最初にこのモジュールを参照したときにデータベースごとに SYSTOOLSPACE に作成される、SYSTOOLS.DIRECTORIES に保管されます。

構文



プロシージャー・パラメーター

alias
ディレクトリー別名を指定する、タイプ VARCHAR (128) の入力引数。

path
パスを指定する、タイプ VARCHAR(1024) の入力引数。

許可

UTL_DIR モジュールに対する EXECUTE 特権。

例

ディレクトリー別名を作成し、UTL_FILE.FOPEN 関数の呼び出しで使用します。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_DIRECTORY('mydir', '/home/user/temp/mydir');
  SET v_filehandle = UTL_FILE.FOPEN('mydir',v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Opened file: myfile.csv
```

CREATE_OR_REPLACE_DIRECTORY プロシージャー - ディレクトリー別名の作成または置換

CREATE_OR_REPLACE_DIRECTORY プロシージャーは、指定されたパスのディレクトリー別名を作成または置換します。

ディレクトリー情報は、最初にこのモジュールを参照したときにデータベースごとに SYSTOOLSPACE に作成される、SYSTOOLS.DIRECTORIES に保管されます。

構文

プロシージャー・パラメーター

alias
ディレクトリー別名を指定する、タイプ VARCHAR (128) の入力引数。

path
パスを指定する、タイプ VARCHAR(1024) の入力引数。

許可

UTL_DIR モジュールに対する EXECUTE 特権。

例

例 1: ディレクトリー別名を作成します。ディレクトリーがすでに存在するため、エラーが発生します。

```
CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

この例では、以下の出力が結果として戻ります。

```
SQL0438N  Application raised error or warning with diagnostic text: "directory
alias already defined".  SQLSTATE=23505
```

例 2: ディレクトリー別名を作成または置換します。

```
CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

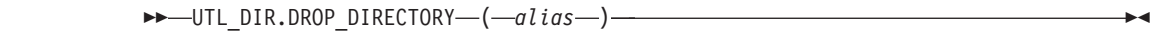
この例では、以下の出力が結果として戻ります。

```
  Return Status = 0
```

DROP_DIRECTORY プロシージャー - プロシージャー別名のドロップ

DROP_DIRECTORY プロシージャーは、指定されたディレクトリー別名をドロップします。

構文



プロシージャー・パラメーター

alias
ディレクトリー別名を指定する、タイプ VARCHAR (128) の入力引数。

許可

UTL_DIR モジュールに対する EXECUTE 特権。

例

指定されたディレクトリー別名をドロップします。

```
CALL UTL_DIR.DROP_DIRECTORY('mydir')@
```

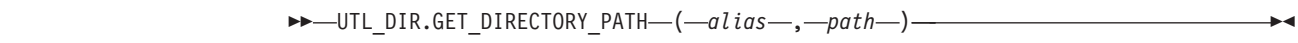
この例では、以下の出力が結果として戻ります。

```
  Return Status = 0
```

GET_DIRECTORY_PATH プロシージャー - ディレクトリー別名のパスの取得

GET_DIRECTORY_PATH プロシージャーは、ディレクトリー別名の対応パスを戻します。

構文



プロシージャー・パラメーター

alias
ディレクトリー別名を指定する、タイプ VARCHAR (128) の入力引数。

path
ディレクトリー別名に定義されるパスを指定する、タイプ VARCHAR(1024) の出力引数。

許可

UTL_DIR モジュールに対する EXECUTE 特権。

例

ディレクトリー別名に定義されるパスを取得します。

```
CALL UTL_DIR.GET_DIRECTORY_PATH('mydir', ? )@
```

この例では、以下の出力が結果として戻ります。

```
  Value of output parameters
  --------------------------
  Parameter Name  : PATH
  Parameter Value : home/rhoda/temp/mydir

  Return Status = 0
```

UTL_FILE モジュール

UTL_FILE モジュールは、データベース・サーバーのファイル・システム上のファイルとの間で読み取りおよび書き込みを行うための一連のルーチンを提供します。

このモジュールのスキーマは SYSIBMADM です。

UTL_FILE モジュールには、以下のシステム定義ルーチンおよびタイプが含まれています。

表 31. UTL_FILE モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
FCLOSE プロシージャー	指定されたファイルをクローズします。
FCLOSE_ALL プロシージャー	オープンしているファイルをすべてクローズします。

表 31. UTL_FILE モジュールで使用可能なシステム定義ルーチン (続き)

ルーチン名	説明
FCOPY プロシージャー	あるファイルから別のファイルへテキストをコピーします。
FFLUSH プロシージャー	書き込まれていないデータをファイルにフラッシュします。
FOPEN 関数	ファイルをオープンします。
FREMOVE プロシージャー	ファイルを除去します。
FRENAME プロシージャー	ファイルの名前を変更します。
GET_LINE プロシージャー	ファイルから行を取得します。
IS_OPEN 関数	指定されたファイルがオープンしているかどうかを判別します。
NEW_LINE プロシージャー	行末文字シーケンスをファイルに書き込みます。
PUT プロシージャー	ストリングをファイルに書き込みます。
PUT_LINE プロシージャー	単一の行をファイルに書き込みます。
PUTF プロシージャー	書式制御ストリングをファイルに書き込みます。
UTL_FILE.FILE_TYPE	ファイル・ハンドルを保管します。

以下のリストには、アプリケーションが受信できる名前付き条件 (Oracle では、「例外」と呼ばれます) が示されています。

表 32. アプリケーションの名前付き条件

条件名	説明
access_denied	ファイルへのアクセスがオペレーティング・システムによって拒否されました。
charsetmismatch	ファイルは、FOPEN_NCHAR を使用してオープンされましたが、その後の入出力操作が PUTF または GET_LINE などの非文字関数を使用しました。
delete_failed	ファイルを削除できません。
file_open	ファイルはすでにオープンされています。
internal_error	UTL_FILE モジュールの未処理の内部エラー。
invalid_filehandle	ファイル・ハンドルは存在しません。
invalid_filename	指定された名前のファイルはパスに存在しません。
invalid_maxlinesize	FOPEN の MAX_LINESIZE 値は無効です。値は 1 から 32672 の範囲でなければなりません。
invalid_mode	FOPEN の open_mode 引数は無効です。

表 32. アプリケーションの名前付き条件 (続き)

条件名	説明
invalid_offset	FSEEK の ABSOLUTE_OFFSET 引数は無効です。このパラメーターは、0 より大きく、ファイルの合計バイト数より少なくなければなりません。
invalid_operation	ファイルは要求どおりにオープンできなかったか、操作できませんでした。
invalid_path	指定されたパスが存在しないか、データベースに不可視です。
read_error	ファイルを読み取れません。
rename_failed	ファイルの名前を変更できません。
write_error	ファイルに書き込めません。

使用上の注意

ファイル・システム上のディレクトリーを参照するには、ディレクトリー別名を使用します。UTL_DIR.CREATE_DIRECTORY または UTL_DIR.CREATE_OR_REPLACE_DIRECTORY プロシージャーを呼び出すことによって、ディレクトリー別名を作成できます。例えば、`CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/mydir')@`。

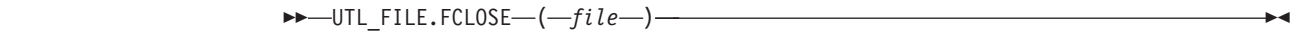
UTL_FILE モジュールは、DB2 インスタンス ID を使用することによって、ファイル操作を実行します。 したがって、ファイルをオープンしている場合は、DB2 インスタンス ID が適切なオペレーティング・システムの許可を持っていることを検証します。

UTL_FILE モジュールは、非パーティション・データベース環境でのみサポートされます。

FCLOSE プロシージャー - オープンしているファイルをクローズする

FCLOSE プロシージャーは、指定されたファイルをクローズします。

構文



プロシージャー・パラメーター

ファイル
ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力または出力引数。ファイルがクローズされている場合は、この値は 0 に設定されます。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

ファイルをオープンし、いくつかのテキストをファイルに書き込んだ後、ファイルをクローズします。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE('Closed file: ' || v_filename);
    END IF;
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Closed file: myfile.csv
```

FCLOSE_ALL プロシージャー - オープンしているファイルをすべてクローズする

FCLOSE_ALL プロシージャーは、オープンしているファイルをすべてクローズします。このプロシージャーは、クローズするオープン・ファイルがなくても、正常に実行されます。

構文

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

2 つのファイルをオープンし、いくつかのテキストをファイルに書き込んだ後、オープン・ファイルをすべてクローズします。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  v_filehandle2   UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
```

```
  DECLARE  v_filename2     VARCHAR(20) DEFAULT 'myfile2.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to a file.');
  SET v_filehandle2 = UTL_FILE.FOPEN(v_dirAlias,v_filename2,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle2,'Some text to write to another file.');
  CALL UTL_FILE.FCLOSE_ALL;
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename || ' is now closed.');
    END IF;
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename2 || ' is now closed.');
    END IF;
END@

CALL proc1@
```

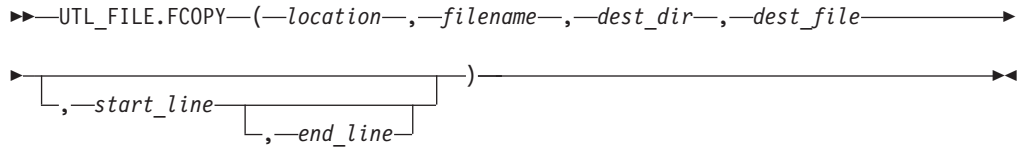
この例では、以下の出力が結果として戻ります。

```
myfile.csv is now closed.
myfile2.csv is now closed.
```

FCOPY プロシージャー - あるファイルから別のファイルへテキストをコピーする

FCOPY プロシージャーは、あるファイルから別のファイルへテキストをコピーします。

構文



プロシージャー・パラメーター

location
ソース・ファイルを含むディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

filename
ソース・ファイルの名前を指定する、タイプ VARCHAR (255) の入力引数。

dest_dir
宛先ディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

dest_file
宛先ファイルの名前を指定する、タイプ VARCHAR (255) の入力引数。

start_line
コピーするソース・ファイルのテキストの最初の行の行番号を指定する、タイプ INTEGER のオプション入力引数。デフォルトは 1 です。

end_line
コピーするソース・ファイルのテキストの最後の行の行番号を指定する、タイプ INTEGER のオプション入力引数。 この引数が省略されるか、NULL の場合、プロシージャーは、ファイルの終わりまですべてのテキストのコピーを続行します。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

emp 表からの従業員のコンマで区切られたリストを含むファイル、empfile.csv のコピーを作成します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'empcopy.csv';
  DECLARE    v_empline       VARCHAR(200);
  CALL UTL_FILE.FCOPY(v_dirAlias,v_src_file,v_dirAlias,v_dest_file);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
  Return Status = 0
```

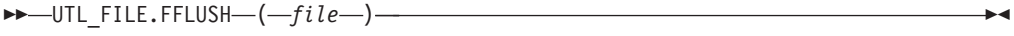
ファイル・コピー、empcopy.csv には、以下のデータが含まれます。

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220
20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300
30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060
50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214
60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580
70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893
90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380
100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

FFLUSH プロシージャー - 書き込まれていないデータをファイルにフラッシュする

FFLUSH プロシージャーは、書き込みバッファーに書き込まれていないデータを強制的にファイルに書き込みます。

構文



プロシージャー・パラメーター

ファイル
　ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

NEW_LINE プロシージャーを呼び出した後で各行をフラッシュします。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt);
    CALL UTL_FILE.FFLUSH(v_empfile_tgt);
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Updated file: empfilenew.csv
```

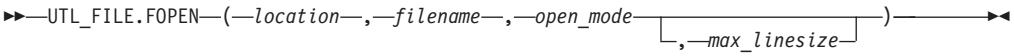
更新済みのファイル empfilenew.csv には、以下のデータが含まれます。

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

FOPEN 関数 - ファイルをオープンする

FOPEN 関数は、入出力用のファイルをオープンします。

構文



戻り値

この関数は、オープンされたファイルのファイル・ハンドルを示すタイプ UTL_FILE.FILE_TYPE の値を戻します。

関数のパラメーター

location
ファイルを含むディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

filename
ファイルの名前を指定する、タイプ VARCHAR (255) の入力引数。

open_mode
ファイルをオープンするモードを指定する、タイプ VARCHAR (10) の入力引数。

a ファイルへの追加

r ファイルからの読み取り

w ファイルへの書き込み

max_linesize
行の最大サイズを文字数で指定する、タイプ INTEGER のオプション入力引数。デフォルト値は 1024 バイトです。 読み取りモードでは、*max_linesize* を超える行を読み取ろうとすると、例外がスローされます。書き込みモードおよび付加モードでは、*max_linesize* を超える行を書き込もうとすると、例外がスローされます。行の終わり文字は、行サイズにカウントされません。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

ファイルをオープンし、いくつかのテキストをファイルに書き込んだ後、ファイルをクローズします。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
```

```
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

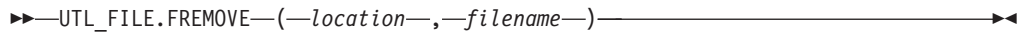
この例では、以下の出力が結果として戻ります。

```
Opened file: myfile.csv
```

FREMOVE プロシージャー - ファイルを除去する

FREMOVE プロシージャーは、指定されたファイルをシステムから除去します。ファイルが存在しない場合は、このプロシージャーは例外をスローします。

構文



プロシージャー・パラメーター

location
ファイルを含むディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

filename
ファイルの名前を指定する、タイプ VARCHAR (255) の入力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

ファイル myfile.csv をシステムから除去します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_FILE.FREMOVE(v_dirAlias,v_filename);
  CALL DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
END@

CALL proc1@
```

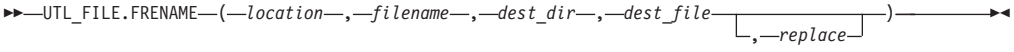
この例では、以下の出力が結果として戻ります。

```
Removed file: myfile.csv
```

FRENAME プロシージャー - ファイルの名前を変更する

FRENAME プロシージャーは、指定されたファイルの名前を変更します。ファイルの名前を変更することによって、実質的にファイルをある場所から別の場所へ移動します。

構文



プロシージャー・パラメーター

location
名前を変更するファイルを含むディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

filename
名前を変更するファイルの名前を指定する、タイプ VARCHAR (255) の入力引数。

dest_dir
宛先ディレクトリーの別名を指定する、タイプ VARCHAR (128) の入力引数。

dest_file
ファイルの新しい名前を指定する、タイプ VARCHAR (255) の入力引数。

replace
ファイルがすでに存在する場合、ディレクトリー *dest_dir* のファイル *dest_file* を置き換えるかどうかを指定する、タイプ INTEGER のオプション入力引数。

1
既存のファイルを置き換えます。

0
ファイルがすでに存在する場合、例外をスローします。*replace* に値が指定されていない場合、これがデフォルトです。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

emp 表からの従業員のコンマで区切られたリストを含むファイル、empfile.csv の名前を変更します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'oldemp.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'newemp.csv';
  DECLARE    v_replace       INTEGER DEFAULT 1;
  CALL UTL_FILE.FRENAME(v_dirAlias,v_src_file,v_dirAlias,
        v_dest_file,v_replace);
  CALL DBMS_OUTPUT.PUT_LINE('The file ' || v_src_file ||
    ' has been renamed to ' || v_dest_file);
END@

CALL proc1@
```

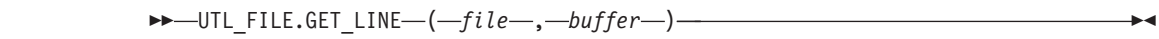
この例では、以下の出力が結果として戻ります。

```
The file oldemp.csv has been renamed to newemp.csv
```

GET_LINE プロシージャー - ファイルから行を取得する

GET_LINE プロシージャーは、指定されたファイルからテキスト行を取得します。テキスト行には、行の終了文字は含まれません。読み取る行がない場合、プロシージャーは、NO_DATA_FOUND 例外をスローします。

構文



プロシージャー・パラメーター

ファイル
オープンされたファイルのファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

buffer
ファイルからのテキストの行を含む、タイプ VARCHAR (32672) の出力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

ファイル empfile.csv のレコードを最後まで読み取って表示します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_filename      VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_empline       VARCHAR(200);
  DECLARE    v_count         INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE 'ORANF'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile = UTL_FILE.FOPEN(v_dirAlias,v_filename,'r');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile, v_empline);
    IF SQLSTATE1 = 'ORANF' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL DBMS_OUTPUT.PUT_LINE(v_empline);
    SET v_count = v_count + 1;
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' || v_count
        || ' records retrieved');
  CALL UTL_FILE.FCLOSE(v_empfile);
END@

CALL proc1@
```

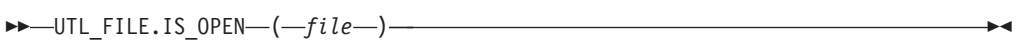
この例では、以下の出力が結果として戻ります。

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092

End of file empfile.csv - 8 records retrieved
```

IS_OPEN 関数 - 指定されたファイルがオープンしているかどうかを判別する

IS_OPEN 関数は、指定されたファイルがオープンしているかどうかを判別します。

構文



戻り値

この関数は、指定されたファイルがオープンされている (TRUE) か、クローズされている (FALSE) かを示すタイプ BOOLEAN の値を戻します。

関数のパラメーター

ファイル
ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

以下は、テキストをファイルに書き込む前に、IS_OPEN 関数を呼び出して、ファイルがオープンされているかどうかを確認する例を示しています。

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_filename);
```

```
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

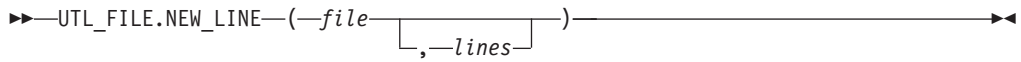
この例では、以下の出力が結果として戻ります。

```
Updated file: myfile.csv
```

NEW_LINE プロシージャー - 行末文字シーケンスをファイルに書き込む

NEW_LINE プロシージャーは、行末文字シーケンスを指定されたファイルに書き込みます。

構文



プロシージャー・パラメーター

ファイル
ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

lines
ファイルに書き込む行末文字シーケンスの数を指定する、タイプ INTEGER のオプション入力引数。 デフォルトは 1 です。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

2 行間隔の従業員レコードのリストを含むファイルを書き込みます。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
```

```
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Wrote to file: empfilenew.csv
```

更新済みのファイル、empfilenew.csv には、以下のデータが含まれます。

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

PUT プロシージャー - ストリングをファイルに書き込む

PUT プロシージャーは、ストリングを指定されたファイルに書き込みます。ストリングの末尾には、行末文字シーケンスは書き込まれません。

構文

プロシージャー・パラメーター

ファイル
ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

buffer
テキストをファイルに書き込むように指定する、タイプ VARCHAR(32672) の入力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

PUT プロシージャーを使用して、ストリングをファイルに追加し、次に NEW_LINE プロシージャーを使用して、行末文字シーケンスを追加します。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Wrote to file: empfilenew.csv
```

更新済みのファイル empfilenew.csv には、以下のデータが含まれます。

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220


20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300


30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060


50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214


60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580


70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893


90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380


100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

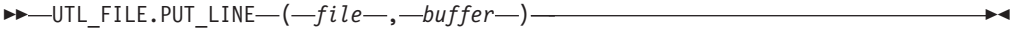
使用上の注意

PUT プロシージャーを使用して、ストリングをファイルに追加した後、NEW_LINE プロシージャーを使用して、行末文字シーケンスをファイルに追加します。

PUT_LINE プロシージャー - テキストの行をファイルに書き込む

PUT_LINE プロシージャーは、行末文字シーケンスを含むテキストの行を指定されたファイルに書き込みます。

構文



プロシージャー・パラメーター

ファイル
行が書き込まれるファイルのファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

buffer
テキストをファイルに書き込むように指定する、タイプ VARCHAR(32672) の入力引数。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

PUT_LINE プロシージャーを使用してテキストの行をファイルに書き込みます。

```
CALL proc1@
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew2.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE    v_count          INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    SET v_count = v_count + 1;
    CALL UTL_FILE.PUT(v_empfile_tgt,'Record ' || v_count || ': ');
    CALL UTL_FILE.PUT_LINE(v_empfile_tgt,v_empline);
```

```
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_src_file || ' - ' || v_count
    || ' records retrieved');
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
End of file empfile.csv - 8 records retrieved
```

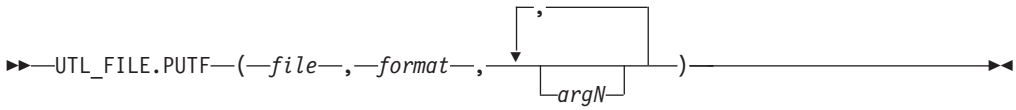
更新済みのファイル、empfilenew2.csv には、以下のデータが含まれます。

```
Record 1: 10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

Record 2: 20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

Record 3: 30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

Record 4: 50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

Record 5: 60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

Record 6: 70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

Record 7: 90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

Record 8: 100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

PUTF プロシージャー - 書式制御ストリングをファイルに書き込む

PUTF プロシージャーは、書式制御ストリングを指定されたファイルに書き込みます。

構文



プロシージャー・パラメーター

ファイル
ファイル・ハンドルを含むタイプ UTL_FILE.FILE_TYPE の入力引数。

フォーマット
テキストのフォーマット設定に使用するストリングを指定する、タイプ VARCHAR(1024) の入力引数。 特殊文字シーケンス %s は、*argN* の値によって置換されます。特殊文字シーケンス \n は、改行を示しています。

argN
特殊文字シーケンス %s の対応するオカレンスに対して、書式制御ストリングで置換する値を指定する、タイプ VARCHAR(1024) のオプション入力引数。最大 5 つの引数、*arg1* から *arg5* までを指定できます。*arg1* は、 %s の最初のオカレンス用に置換され、*arg2* は、%s の 2 番目のオカレンス用に置換されるという具合です。

許可

UTL_FILE モジュールに対する EXECUTE 特権。

例

従業員データをフォーマットします。

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  DECLARE v_format        VARCHAR(200);
  SET v_format = '%s %s, %s¥nSalary: $%s Commission: $%s¥n¥n';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  CALL UTL_FILE.PUTF(v_filehandle,v_format,'000030','SALLY','KWAN','40175','3214');
  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

この例では、以下の出力が結果として戻ります。

```
Wrote formatted text to file: myfile.csv
```

フォーマット済みのファイル、myfile.csv には、以下のデータが含まれます。

```
000030 SALLY, KWAN
Salary: $40175 Commission: $3214
```

UTL_FILE.FILE_TYPE

UTL_FILE.FILE_TYPE は、UTL_FILE モジュールのルーチンによって使用されるファイル・ハンドル・タイプです。

例

タイプ UTL_FILE.FILE_TYPE の変数を宣言します。

```
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
```

UTL_MAIL モジュール

UTL_MAIL モジュールは、E メールを送信する機能を提供します。

このモジュールのスキーマは SYSIBMADM です。

UTL_MAIL モジュールは、以下のルーチンで構成されます。

表 33. UTL_MAIL モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
SEND プロシージャー	E メールをパッケージして SMTP サーバーに送信します。
SEND_ATTACH_RAW プロシージャー	SEND プロシージャーと同じですが、BLOB 添付ファイルがあります。

表 33. UTL_MAIL モジュールで使用可能なシステム定義ルーチン (続き)

ルーチン名	説明
SEND_ATTACH_VARCHAR2	SEND プロシージャーと同じですが、VARCHAR 添付ファイルがあります。

使用上の注意

UTL_MAIL モジュールを使用して正常に E メールを送信するには、データベース構成パラメーター SMTP_SERVER に有効な SMTP サーバーのアドレスを 1 つ以上含める必要があります。

例

例 1: デフォルト・ポート 25 を使って単一の SMTP サーバーをセットアップするには、次のようにします。

```
db2 update db cfg using smtp_server 'smtp.ibm.com'
```

例 2: デフォルト・ポート 25 ではなく、ポート 2000 を使用する単一の SMTP サーバーをセットアップするには、次のようにします。

```
db2 update db cfg using smtp_server 'smtp2.ibm.com:2000'
```

例 3: SMTP サーバーのリストを設定するには、次のようにします。

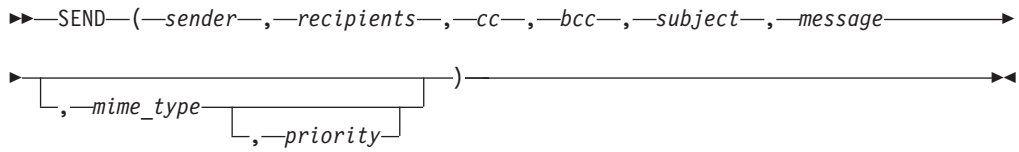
```
db2 update db cfg using smtp_server
   'smtp.example.com,smtp1.example.com:23,smtp2.example.com:2000'
```

注: E メールは、いずれかの SMTP サーバーから成功応答を受信するまで、リストの順序で各 SMTP サーバーに送信されます。

SEND プロシージャー - SMTP サーバーへの E メールの送信

SEND プロシージャーは、E メールを SMTP サーバーに送信する機能を提供します。

構文



パラメーター

sender
送信者の E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

recipients
受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

cc コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

bcc
ブラインド・コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

subject
E メールの件名行を指定する、タイプ VARCHAR(32672) の入力引数。

message
E メールの本文を指定する、タイプ VARCHAR(32672) の入力引数。

mime_type
メッセージの MIME タイプを指定する、タイプ VARCHAR(1024) のオプション入力引数。デフォルトは 'text/plain; charset=us-ascii' です。

priority
E メールの優先順位を指定する、タイプ INTEGER のオプション引数。デフォルト値は 3 です。

許可

UTL_MAIL モジュールに対する EXECUTE 特権。

例

例 1: 以下の無名ブロックは、簡単な E メール・メッセージを送信します。

```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END@
```

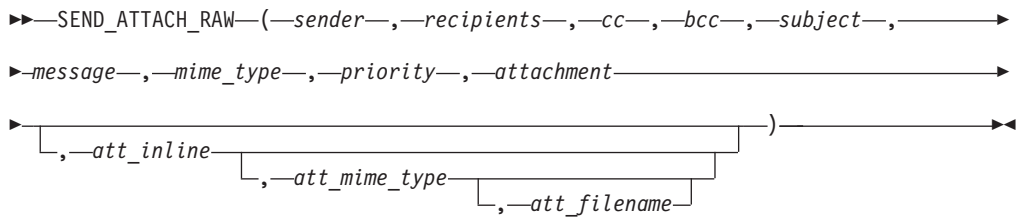
この例では、以下の出力が結果として戻ります。

```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END
DB20000I  The SQL command completed successfully.
```

SEND_ATTACH_RAW プロシージャー - BLOB 添付ファイル付き E メールの SMTP サーバーへの送信

SEND_ATTACH_RAW プロシージャーは、E メールをバイナリー添付ファイルと一緒に SMTP サーバーに送信する機能を提供します。

構文



パラメーター

sender
送信者の E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

recipients
受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

cc コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

bcc
ブラインド・コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

subject
E メールの件名行を指定する、タイプ VARCHAR(32672) の入力引数。

message
E メールの本文を指定する、タイプ VARCHAR(32672) の入力引数。

mime_type
メッセージの MIME タイプを指定する、タイプ VARCHAR(1024) の入力引数。デフォルトは `'text/plain; charset=us-ascii'` です。

priority
E メールの優先順位を指定する、タイプ INTEGER の入力引数。デフォルト値は 3 です。

attachment
添付ファイルが入った、タイプ BLOB(10M) の入力引数。

att_inline
添付ファイルが表示可能インラインであるかどうかを指定する、タイプ BOOLEAN のオプション入力引数。「true」に設定した場合には添付ファイルは表示可能インラインであり、「false」の場合は表示可能インラインではありません。デフォルト値は「true」です。

att_mime_type
添付ファイルの MIME タイプを指定する、タイプ VARCHAR(1024) のオプション入力引数。デフォルト値は `application/octet` です。

att_filename
添付ファイルが入っているファイルの名前を指定する、タイプ VARCHAR(512) のオプション入力引数。デフォルト値は NULL です。

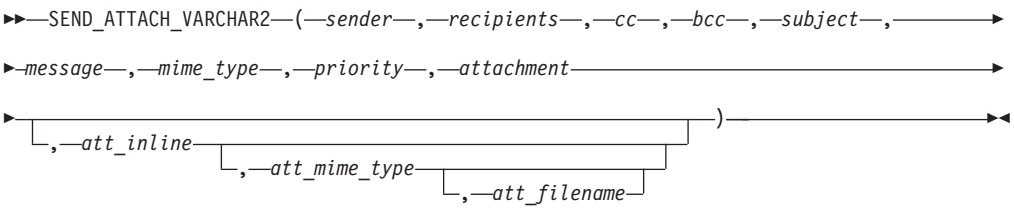
許可

UTL_MAIL モジュールに対する EXECUTE 特権。

SEND_ATTACH_VARCHAR2 プロシージャー - VARCHAR 添付ファイル付き E メールの SMTP サーバーへの送信

SEND_ATTACH_VARCHAR2 プロシージャーは、E メールをテキスト添付ファイルと一緒に SMTP サーバーに送信する機能を提供します。

構文



パラメーター

sender
送信者の E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

recipients
受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

cc コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

bcc
ブラインド・コピー受信者の E メール・アドレスをコンマ区切りで指定する、タイプ VARCHAR(32672) の入力引数。

subject
E メールの件名行を指定する、タイプ VARCHAR(32672) の入力引数。

message
E メールの本文を指定する、タイプ VARCHAR(32672) の入力引数。

mime_type
メッセージの MIME タイプを指定する、タイプ VARCHAR(1024) の入力引数。デフォルトは `'text/plain; charset=us-ascii'` です。

priority
E メールの優先順位を指定する、タイプ INTEGER の入力引数。デフォルト値は 3 です。

attachment
添付ファイルが入った、タイプ VARCHAR(32000) の入力引数。

att_inline
添付ファイルが表示可能インラインであるかどうかを指定する、タイプ BOOLEAN のオプション入力引数。「true」に設定した場合には添付ファイルは表示可能インラインであり、「false」の場合は表示可能インラインではありません。デフォルト値は「true」です。

att_mime_type
添付ファイルの MIME タイプを指定する、タイプ VARCHAR(1024) のオプション入力引数。デフォルト値は `'text/plain; charset=us-ascii'` です。

att_filename
添付ファイルが入っているファイルの名前を指定する、タイプ VARCHAR(512) のオプション入力引数。デフォルト値は NULL です。

許可

UTL_MAIL モジュールに対する EXECUTE 特権。

UTL_SMTP モジュール

UTL_SMTP モジュールは、SMTP (Simple Mail Transfer Protocol) を介して E メールを送信する機能を提供します。

UTL_SMTP モジュールは、以下のルーチンで構成されます。

表 34. UTL_SMTP モジュールで使用可能なシステム定義ルーチン

ルーチン名	説明
CLOSE_DATA プロシージャー	E メール・メッセージを終了します。
COMMAND プロシージャー	SMTP コマンドを実行します。
COMMAND_REPLIES プロシージャー	複数の応答行が予想される場合に SMTP コマンドを実行します。
DATA プロシージャー	E メール・メッセージの本文を指定します。
EHLO プロシージャー	SMTP サーバーとの初期ハンドシェークを実行し、拡張情報を返します。
HELO プロシージャー	SMTP サーバーとの初期ハンドシェークを実行します。
HELP プロシージャー	HELP コマンドを送信します。
MAIL プロシージャー	メール・トランザクションを開始します。
NOOP プロシージャー	ヌル・コマンドを送信します。
OPEN_CONNECTION 関数	接続を開きます。
OPEN_CONNECTION プロシージャー	接続を開きます。
OPEN_DATA プロシージャー	DATA コマンドを送信します。
QUIT プロシージャー	SMTP セッションを終了し、切断します。

表 34. UTL_SMTP モジュールで使用可能なシステム定義ルーチン (続き)

ルーチン名	説明
RCPT プロシージャー	E メール・メッセージの受信者を指定します。
RSET プロシージャー	現在のメール・トランザクションを終了します。
VRFY プロシージャー	E メール・アドレスを妥当性検査します。
WRITE_DATA プロシージャー	E メール・メッセージの一部分を書き込みます。
WRITE_RAW_DATA プロシージャー	RAW データで構成される E メール・メッセージの一部分を書き込みます。

次の表は、モジュールで使用可能なパブリック変数をリストしています。

表 35. UTL_SMTP モジュールで使用可能なシステム定義タイプ

パブリック変数	データ・タイプ	説明
connection	RECORD	SMTP 接続の説明。
reply	RECORD	SMTP 応答行。

CONNECTION レコード・タイプは、SMTP 接続の説明を記述します。

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE connection AS ROW
(
  /* name or IP address of the remote host running SMTP server */
    host VARCHAR(255),
  /* SMTP server port number */
    port INTEGER,
  /* transfer timeout in seconds */
    tx_timeout INTEGER,
);
```

REPLY レコード・タイプは、SMTP 応答行の説明を記述します。 REPLIES は、SMTP 応答行の配列です。

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE reply AS ROW
(
  /* 3 digit reply code received from the SMTP server */
    code INTEGER,
  /* the text of the message received from the SMTP server */
    text VARCHAR(508)
);
```

例

例 1: 以下のプロシージャーは、UTL_SMTP モジュールを使用してテキスト E メール・メッセージを構成し、送信します。

```
CREATE OR REPLACE PROCEDURE send_mail(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096))
SPECIFIC send_mail
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
```

```
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.DATA(
    v_conn,
    'Date: ' || TO_CHAR(SYSDATE, 'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf ||
    'From: ' || p_sender || v_crlf ||
    'To: ' || p_recipient || v_crlf ||
    'Subject: ' || p_subj || v_crlf ||
    p_msg);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

例 2: 以下の例では、DATA プロシージャーではなく、OPEN_DATA、WRITE_DATA、および CLOSE_DATA プロシージャーを使用します。

```
CREATE OR REPLACE PROCEDURE send_mail_2(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096)) SPECIFIC send_mail_2
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.OPEN_DATA(v_conn);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'From: ' || p_sender || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'To: ' || p_recipient || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'Subject: ' || p_subj || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, v_crlf || p_msg);
  CALL UTL_SMTP.CLOSE_DATA(v_conn);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail_2('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

CLOSE_DATA プロシージャー - E メール・メッセージの終了

CLOSE_DATA プロシージャーは、E メール・メッセージを終了します。

このプロシージャーは、以下のシーケンスを送信することによって E メール・メッセージを終了します。

```
<CR><LF>.<CR><LF>
```

これは、行の先頭の単一ピリオドです。

構文

パラメーター

c 閉じる SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

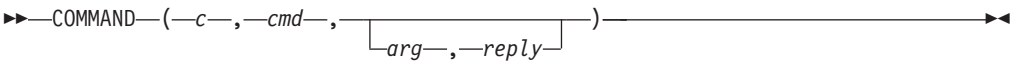
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

COMMAND プロシージャー - SMTP コマンドの実行

COMMAND プロシージャーは、SMTP コマンドを実行する機能を提供します。

注: 複数の応答行が返されることが予想される場合には、COMMAND_REPLIES を使用します。

構文



パラメーター

c コマンドの送信先 SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

cmd
処理する SMTP コマンドを指定する、タイプ VARCHAR(510) の入力引数。

arg
SMTP コマンドの引数を指定する、タイプ VARCHAR(32672) のオプション入力引数。デフォルトは NULL です。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

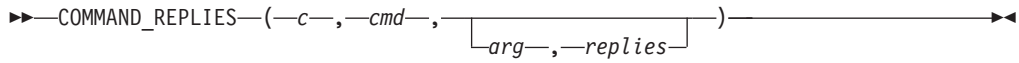
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

COMMAND_REPLIES プロシージャー - 複数の応答行が予想される場合の SMTP コマンドの実行

COMMAND_REPLIES 関数は、複数の応答行を返す SMTP コマンドを処理します。

注: 予想される応答行が 1 行だけの場合には、COMMAND を使用します。

構文



パラメーター

c コマンドの送信先 SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

cmd
処理する SMTP コマンドを指定する、タイプ VARCHAR(510) の入力引数。

arg
SMTP コマンドの引数を指定する、タイプ VARCHAR(32672) のオプション入力引数。デフォルトは NULL です。

replies
SMTP サーバーから複数の応答行を返す、タイプ REPLIES のオプション出力引数。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

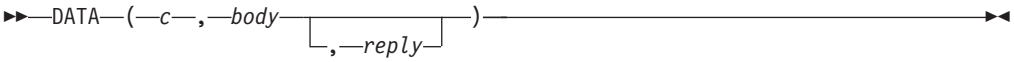
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

DATA プロシージャー - E メール・メッセージの本文の指定

DATA プロシージャーは、E メール・メッセージの本文を指定する機能を提供します。

メッセージは、<CR><LF>.<CR><LF> シーケンスで終了します。

構文



パラメーター

c コマンドの送信先 SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

本体
送信される E メール・メッセージの本文を指定する、タイプ VARCHAR(32000) の入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

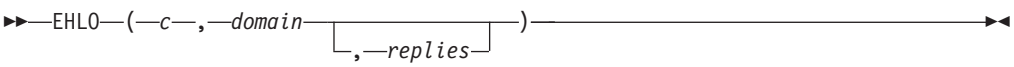
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

EHLO プロシージャー - SMTP サーバーとの初期ハンドシェークを実行し拡張情報を返す

EHLO プロシージャーは、接続を確立した後に、SMTP サーバーとの初期ハンドシェークを実行します。

EHLO プロシージャーにより、クライアントがそれ自体の実体を SMTP サーバーに示すことができます。 HELO プロシージャーは同等の機能を実行しますが、サーバーに関して返される情報は少なくなります。

構文



パラメーター

c ハンドシェークを実行するときの SMTP サーバーへの接続を指定する、タイプ CONNECTION の入力または出力引数。

domain
送信元ホストのドメイン・ネームを指定する、タイプ VARCHAR (255) の入力引数。

replies
SMTP サーバーから複数の応答行を返す、タイプ REPLIES のオプション出力引数。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

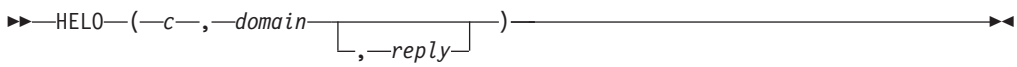
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

HELO プロシージャー - SMTP サーバーとの初期ハンドシェークの実行

HELO プロシージャーは、接続を確立した後に、SMTP サーバーとの初期ハンドシェークを実行します。

HELO プロシージャーにより、クライアントがそれ自体の実体を SMTP サーバーに示すことができます。 EHLO プロシージャーは同等の機能を実行しますが、サーバーに関して返される情報は多くなります。

構文



パラメーター

c ハンドシェークを実行するときの SMTP サーバーへの接続を指定する、タイプ CONNECTION の入力または出力引数。

domain
送信元ホストのドメイン・ネームを指定する、タイプ VARCHAR (255) の入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

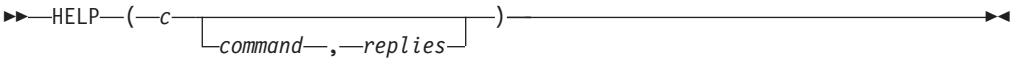
使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

HELP プロシージャー - HELP コマンドの送信

HELP 関数は、HELP コマンドを SMTP サーバーに送信する機能を提供します。

構文



パラメーター

c コマンドの送信先 SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

command
ヘルプを要求する対象となるコマンドを指定する、タイプ VARCHAR(510) のオプション入力引数。

replies
SMTP サーバーから複数の応答行を返す、タイプ REPLIES のオプション出力引数。

許可

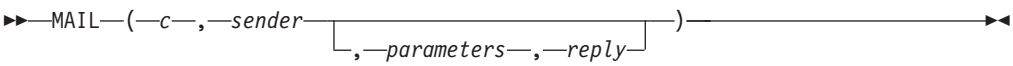
UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

MAIL プロシージャー - メール・トランザクションの開始

構文



パラメーター

c メール・トランザクションを開始するときの SMTP サーバーへの接続を指定する、タイプ CONNECTION の入力または出力引数。

sender
送信者の E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

パラメーター
オプションのメール・コマンド・パラメーターを `key=value` の形式で指定する、タイプ VARCHAR(32672) のオプション入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

NOOP プロシージャー - ヌル・コマンドの送信

NOOP プロシージャーは、ヌル・コマンドを SMTP サーバーに送信します。NOOP は、成功応答を取得する以外にサーバーへの影響はありません。

構文

パラメーター

c コマンドを送信する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

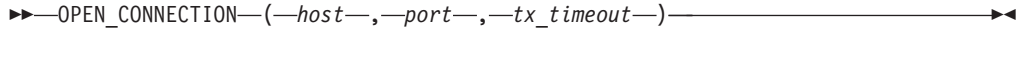
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

OPEN_CONNECTION 関数 - SMTP サーバーへ接続ハンドルを返す

OPEN_CONNECTION 関数は、SMTP サーバーへ接続ハンドルを返します。

この関数は、SMTP サーバーへ接続ハンドルを返します。

構文



パラメーター

ホスト
SMTP サーバーの名前を指定する、タイプ VARCHAR (255) の入力引数。

port
SMTP サーバーが listen するポート番号を指定する、タイプ INTEGER の入力引数。

tx_timeout
タイムアウト値を秒単位で指定する、タイプ INTEGER の入力引数。プロシージャーが待機しないように指示するには、この値を 0 に設定します。プロシージャーが無制限に待機するように指示するには、この値を NULL に設定します。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

OPEN_CONNECTION プロシージャー - SMTP サーバーへの接続のオープン

OPEN_CONNECTION プロシージャーは、SMTP サーバーへの接続を開きます。

構文

パラメーター

ホスト
SMTP サーバーの名前を指定する、タイプ VARCHAR (255) の入力引数。

port
SMTP サーバーが listen するポート番号を指定する、タイプ INTEGER の入力引数。

connection
SMTP サーバーへの接続ハンドルを返す、タイプ CONNECTION の出力引数。

tx_timeout
タイムアウト値を秒単位で指定する、タイプ INTEGER のオプション入力引数。プロシージャーが待機しないように指示するには、この値を 0 に設定します。プロシージャーが無制限に待機するように指示するには、この値を NULL に設定します。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY の出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

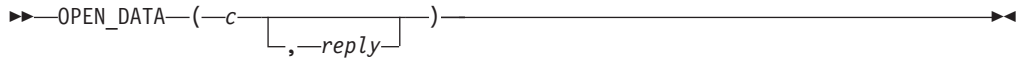
許可

UTL_SMTP モジュールに対する EXECUTE 特権。

OPEN_DATA プロシージャー - SMTP サーバーへの DATA コマンドの送信

OPEN_DATA プロシージャーは、DATA コマンドを SMTP サーバーに送信します。

構文



パラメーター

c コマンドを送信する SMTP 接続を指定する、タイプ CONNECTION の入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

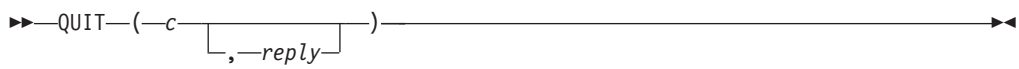
使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

QUIT プロシージャー - SMTP サーバーとのセッションのクローズ

QUIT プロシージャーは、SMTP サーバーとのセッションを閉じます。

構文



パラメーター

c 終了する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

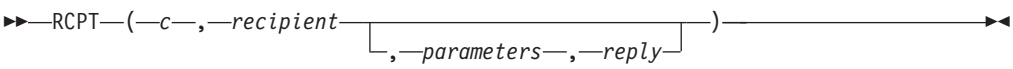
このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

RCPT プロシージャー - 受信者の E メール・アドレスの提供

RCPT プロシージャーは、受信者の E メール・アドレスを提供します。

注: 複数の受信者をスケジュールに入れるには、RCPT プロシージャーを複数回呼び出します。

構文



パラメーター

c 受信者を追加する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

recipient
受信者の E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

パラメーター
メール・コマンド・パラメーターを key=value の形式で指定する、タイプ VARCHAR(32672) のオプション入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

RSET プロシージャー - 現行メール・トランザクションの終了

RSET プロシージャーは、現行メール・トランザクションを終了する機能を提供します。

構文

パラメーター

c メール・トランザクションを取り消す SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY のオプション出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

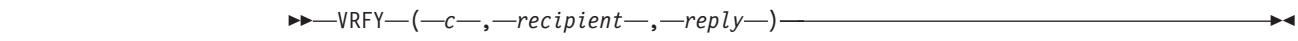
使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

VRFY プロシージャー - 受信者の E メール・アドレスの妥当性検査と検証

VRFY プロシージャーは、受信者の E メール・アドレスを妥当性検査および検証する機能を提供します。有効である場合、受信者の氏名および完全修飾メールボックスが返されます。

構文



パラメーター

c E メール・アドレスを検証する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

recipient
検証する E メール・アドレスを指定する、タイプ VARCHAR(256) の入力引数。

reply
SMTP サーバーから単一の応答行を返す、タイプ REPLY の出力引数。複数の応答行が SMTP サーバーによって返された場合には、最後の応答行です。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

使用上の注意

このプロシージャーは、PL/SQL 割り当てステートメントで関数呼び出し構文を使用して呼び出すことができます。

WRITE_DATA プロシージャー - E メール・メッセージの一部分の書き込み

WRITE_DATA プロシージャーは、データを E メール・メッセージに追加する機能を提供します。 WRITE_DATA プロシージャーは、繰り返し呼び出してデータを追加することができます。

構文

パラメーター

c データを追加する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

データ
E メール・メッセージに追加されるデータを指定する、タイプ VARCHAR(32000) の入力引数。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

WRITE_RAW_DATA プロシージャー - E メール・メッセージへの RAW データの追加

WRITE_RAW_DATA プロシージャーは、E メール・メッセージにデータを追加する機能を提供します。 WRITE_RAW_DATA プロシージャーは、繰り返し呼び出してデータを追加することができます。

構文

パラメーター

c データを追加する SMTP 接続を指定する、タイプ CONNECTION の入力または出力引数。

データ
E メール・メッセージに追加されるデータを指定する、タイプ BLOB(15M) の入力引数。

許可

UTL_SMTP モジュールに対する EXECUTE 特権。

第 4 章 DB2 互換性フィーチャー

DB2 互換性フィーチャーの概要

DB2 バージョン 9.5 では、別のリレーショナル・データベース製品のために作成されたいくつかのアプリケーションを DB2 上で実行できるようにするタスクを大幅に簡略化できる、多数のフィーチャーが導入されました。DB2 バージョン 9.7 では、既存のアプリケーションの有効化に伴うこの複雑性をさらに軽減して時間を短縮する、追加フィーチャーが導入されています。

以下のフィーチャーをはじめ、これらのフィーチャーの一部はデフォルトで有効になっています。

- 暗黙的キャスト (緩やかな型判定)。現在、他のデータ・サーバーで実行されているアプリケーションを DB2 で利用する場合に、変更する必要がある SQL の数を減らします。
- 新しいスカラー関数。詳しくは、『サポートされる関数、管理 SQL ルーチン、およびビュー』を参照してください。
- TIMESTAMP_FORMAT および VARCHAR_FORMAT スカラー関数に加えられた大幅な改善点。(TO_DATE と TO_TIMESTAMP は TIMESTAMP_FORMAT の同義語であり、また TO_CHAR は VARCHAR_FORMAT の同義語です)
  - TIMESTAMP_FORMAT – この関数は、指定された形式を使用した入力文字列の解釈に基づくタイム・スタンプを戻します。
  - VARCHAR_FORMAT – この関数は、指定された文字テンプレートに従ってフォーマットされた入力式の文字列表現を戻します。
- SQL 言語の一部の制限事項の解除。これにより、製品間で構文の互換性が実現しています。例えば、副照会および表関数での相関名の使用がオプションとなりました。
- 以下のような、他のデータベース製品で使用されている構文の同義語
  - UNIQUE は、列関数のコンテキストおよび照会の選択リストでは DISTINCT の同義語です。
  - MINUS は、EXCEPT セット演算子の同義語です。
  - *seqname*.NEXTVAL および *seqname*.CURRVAL は、SQL 標準構文 NEXT VALUE FOR *seqname* および PREVIOUS VALUE FOR *seqname* の代わりに使用できます。
- パッケージ変数を簡単にマップし、@@nested、@@level、または @errorlevel グローバル変数をエミュレートし、あるいは DB2 アプリケーションからトリガー、関数、またはプロシージャーに情報を渡すために使用できるグローバル変数
- SQL プロシージャー内の VARRAY 構成体へ簡単にマップするために使用できる ARRAY コレクション・データ・タイプ
- DB2 で他の DBMS ベンダーのアプリケーションを容易に有効にできるように、ID の長さ制限が緩和

- RID への参照に使用できる疑似列 ROWID。非修飾 ROWID 参照は RID_BIT() と同等であり、EMPLOYEE.ROWID などの修飾 ROWID は RID_BIT(EMPLOYEE) と同等です。

その他のフィーチャーは、**DB2_COMPATIBILITY_VECTOR** という DB2 レジストリー変数を設定することで個別に有効にすることができます。これらのフィーチャーはデフォルトで無効になっています。

- CONNECT BY PRIOR 構文を使用した階層照会の設定
- 外部結合演算子 (+) を使用した外部結合のサポート
- DATE データ・タイプを、日付と時間の値を結合した TIMESTAMP(0) として使用すること
- NUMBER データ・タイプをサポートする構文およびセマンティクス
- VARCHAR2 データ・タイプをサポートする構文およびセマンティクス
- ROWNUM という名前の疑似列は ROW_NUMBER() OVER() と同義ですが、ROWNUM は SELECT LIST および WHERE 節で許可されています。
- DUAL という名前のダミー表は、SYSIBM.SYSDUMMY1 と同様の機能を備えています。
- TRUNCATE TABLE ステートメント用の代替セマンティクス。この下では、IMMEDIATE がオプションのキーワードとなり、指定がない場合にデフォルトになります。TRUNCATE ステートメントが論理作業単位内の最初のステートメントではない場合、TRUNCATE ステートメントが実行される前に暗黙的なコミット操作が実行されます。
- バイト長が 254 以下である文字定数および GRAPHIC ストリング定数への、CHAR データ・タイプまたは GRAPHIC データ・タイプ (VARCHAR データ・タイプでも VARGRAPHIC データ・タイプでもない) の割り当てのサポート
- 配列で first、last、next、previous などの演算を実行するコレクション・メソッドの使用
- Oracle データ・ディクショナリーと互換性があるビューの作成のサポート
- PL/SQL ステートメントおよび言語要素のコンパイルおよび実行のサポート
- OPEN の際にカーソルをマテリアライズして、そのカーソルを後続のステートメントに対してインセンシティブにする操作のサポート
- プロシージャーにおける、デフォルトが定義された INOUT パラメーターのサポート。このパラメーターは、引数を指定しないでも呼び出しが可能です。

その他のリソース

詳しくは、DB2 Viper 2 の互換性フィーチャーを参照してください。

IBM Migration Toolkit (MTK) について詳しくは、Migrate Now! を参照してください。

Oracle データベースに関する DB2 互換性フィーチャーについては、「Oracle to DB2 Conversion Guide: Compatibility Made Easy」を参照してください。

DB2_COMPATIBILITY_VECTOR レジストリー変数

DB2_COMPATIBILITY_VECTOR レジストリー変数は、 DB2 バージョン 9.5 で導入された、1 つ以上の DB2 互換性フィーチャーを有効にするために使用します。

これらのフィーチャーによって、他のリレーショナル・データベース・ベンダー用に作成されたアプリケーションを DB2 バージョン 9.5 以降にマイグレーションするタスクが容易になります。

重要: こうしたフィーチャーは、互換性に関する特定の目的で必要な場合にのみ使用可能にしてください。DB2 互換性フィーチャーが使用可能な場合には、一部の SQL 動作は、SQL リファレンス情報に記されている動作とは異なります。ご使用の SQL アプリケーションにどのような影響が考えられるかを判別するには、それぞれの互換性設定に関連した資料を参照してください。

この DB2 レジストリー変数は 16 進値で表され、変数内の各ビットによって DB2 互換性フィーチャーの 1 つが有効になります。

- オペレーティング・システム: すべて
- デフォルト: NULL、値: NULL または 00 から 3FFF。これらの DB2 互換性フィーチャーを最大限に活用するには、Oracle アプリケーションの場合には値を ORA に、Sybase アプリケーションの場合には SYB に設定します。これらは推奨設定です。

レジストリー変数の設定

表 36. **DB2_COMPATIBILITY_VECTOR** *値*

ビット位置	互換性フィーチャー	説明
1 (0x01)	ROWNUM	ROWNUM を ROW_NUMBER() OVER() の同義語として使用することを可能にし、ROWNUM を SQL ステートメントの WHERE 節に含めることを許可します。
2 (0x02)	DUAL	「DUAL」への非修飾表参照を SYSIBM.DUAL として解決します。
3 (0x04)	外部結合演算子	外部結合演算子 (+) のサポートを有効にします。
4 (0x08)	階層照会	CONNECT BY 節を使用した階層照会のサポートを有効にします。
5 (0x10)	NUMBER データ・タイプ [1]	NUMBER データ・タイプおよび関連する数値処理を有効にします。
6 (0x20)	VARCHAR2 データ・タイプ [1]	VARCHAR2 と NVARCHAR2 データ・タイプ、および関連する文字ストリング処理を有効にします。
7 (0x40)	DATE データ・タイプ [1]	DATE データ・タイプを日付と時間の値を結合した TIMESTAMP(0) として使用することを可能にします。

表 36. **DB2_COMPATIBILITY_VECTOR** 値 (続き)

ビット位置	互換性フィーチャー	説明
8 (0x80)	TRUNCATE TABLE	TRUNCATE ステートメント用の代替セマンティクスを有効にします。IMMEDIATE がオプションのキーワードであり、指定がない場合にデフォルトになります。TRUNCATE ステートメントが論理作業単位内の最初のステートメントではない場合、TRUNCATE ステートメントが実行される前に暗黙的なコミット操作が実行されます。
9 (0x100)	文字リテラル	バイト長が 254 以下である文字定数および GRAPHIC ストリング定数に、CHAR データ・タイプまたは GRAPHIC データ・タイプ (VARCHAR データ・タイプでも VARGRAPHIC データ・タイプでもない) を割り当てることを可能にします。
10 (0x200)	コレクション・メソッド	配列で first、last、next、previous などの演算を実行するメソッドの使用を可能にします。また、配列内の特定の要素の参照に、大括弧の代わりに括弧を使用できるようにします。例えば、array1(*i*) は array1 の要素 *i* を参照します。
11 (0x400)	データ・ディクショナリー互換ビュー [1]	データ・ディクショナリー互換ビューの作成を可能にします。
12 (0x800)	PL/SQL のコンパイル [2]	PL/SQL ステートメントおよび言語要素のコンパイルと実行を可能にします。
13 (0x1000)	インセンシティブ・カーソル	SELECT ステートメントで FOR UPDATE が明示的に指定されていない場合には、WITH RETURN が定義されたカーソルをインセンシティブにできるようにします。
14 (0x2000)	416 ページの『INOUT パラメーター』	INOUT パラメーター宣言の DEFAULT 指定を可能にします。

1. データベースの作成中にのみ当てはまります。このフィーチャーを有効または無効にすると、それ以後に作成されたデータベースに対してのみ適用されます。
2. 『PL/SQL サポートに関する制約事項』を参照してください。

使用法

DB2_COMPATIBILITY_VECTOR レジストリー変数の設定と更新は、db2set コマンドを使用して以下のように行います。

- サポートされているすべての Oracle 互換性フィーチャーを有効にするには、レジストリー変数の値を ORA (16 進値 FFF と等価) に設定します。

- サポートされているすべての Sybase 互換性フィーチャーを有効にするには、レジストリー変数の値を SYB (16 進値 3004 と等価) に設定します。

DB2_COMPATIBILITY_VECTOR レジストリー変数を設定すると、作成されるすべてのデータベースが UNICODE データベースとして作成されることになります。

インスタンスを停止し、再始動するまで、レジストリー変数の新規設定は反映されません。変更を有効にするには、既存の DB2 パッケージを再バインドする必要があります。明示的に再バインドされていないパッケージは、次回の暗黙的な再バインドで変更が有効になります。

例 1

この例では、サポートされているすべての Oracle 互換性フィーチャーを有効にするようにレジストリー変数を設定しています。

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

例 2

この例は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数をリセットしてすべての互換性フィーチャーを無効にする方法を示しています。

```
db2set DB2_COMPATIBILITY_VECTOR=
db2stop
db2start
```

データベースの作成を、NUMBER データ・タイプまたは VARCHAR2 データ・タイプを有効にして、TIMESTAMP(0) としての DATE データ・タイプの使用を有効にして、または Oracle データ・ディクショナリー互換ビューの作成を有効にして行った場合、db2set コマンドの実行後もデータベースでこれらのフィーチャーが有効のままになることに注意してください。

Oracle アプリケーションを使用可能にするための DB2 のセットアップ

DB2 環境を適切にセットアップすることで、DB2 データ・サーバーで Oracle アプリケーションを使用できるようになります。

- DB2 データ・サーバー製品をインストールする必要があります。
- db2set コマンドを実行する権限が必要です。
- CREATE DATABASE コマンドを実行する権限が必要です。

注: コントロール・センター・ツール (DB2 バージョン 9.7 で非推奨になっており、将来のリリースで廃止される可能性がある) は、DB2 互換性フィーチャーが有効になっている環境ではサポートされません。

DB2 環境で Oracle アプリケーションを使用可能にするには、このタスクを実行します。DB2 環境では、他のデータベース・ベンダーが一般的に提供しているフィーチャーの多くがサポートされます。このタスクは、DB2 インターフェースから Oracle データ・タイプを参照する PL/SQL ステートメントまたは SQL ステートメント、あるいはその他の SQL 互換フィーチャーを実行する前提条件になります。

DB2 互換性フィーチャーはデータベース・レベルで有効になり、データベース作成後にこれを無効にすることはできません。

1. DB2 コマンド・ウィンドウを開きます。
2. DB2 データベース・マネージャーを開始します。

   ```
   db2start
   ```

3. **DB2_COMPATIBILITY_VECTOR** レジストリー変数を、使用する互換性フィーチャーを有効にする 16 進値に設定します。これらの DB2 互換性フィーチャーを最大限に活用するには、値を ORA に設定します。これは推奨される設定です。

   ```
   db2set DB2_COMPATIBILITY_VECTOR=ORA
   ```

4. **DB2_DEFERRED_PREPARE_SEMANTICS** レジストリー変数に YES を設定し、据え置き準備のサポートを使用可能にします。
   DB2_COMPATIBILITY_VECTOR レジストリー変数に ORA が設定され、また **DB2_DEFERRED_PREPARE_SEMANTICS** レジストリー変数が設定されていない場合、デフォルト値の YES が使用されます。しかし、**DB2_DEFERRED_PREPARE_SEMANTICS** レジストリー変数には YES を明示的に設定することをお勧めします。

   ```
   db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
   ```

5. db2stop コマンドおよび db2start コマンドを実行し、データベース・マネージャーを停止し、その後再始動します。

   ```
   db2stop
   db2start
   ```

6. CREATE DATABASE コマンドを実行して、DB2 データベースを作成します。データベースは、UNICODE データベースとして作成する必要があります。これが、デフォルトです。 例えば、DB という名前のデータベースを作成するには、以下のコマンドを実行します。

   ```
   db2 CREATE DATABASE DB
   ```

7. オプション: CLPPlus またはコマンド行プロセッサー (CLP) スクリプト (`script.sql`) を実行して、データベースで PL/SQL ステートメントとデータ・タイプがサポートされていることを確認します。 以下の CLPPlus スクリプトによって、シンプルなプロシージャーが作成され、呼び出されます。

   ```
   CONNECT user@hostname:port/dbname;

   CREATE TABLE t1 (c1 NUMBER);

   CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
   AS
   BEGIN
    INSERT INTO t1 VALUES (num);

    message := 'The number you passed is: ' || TO_CHAR(num);
   END;
   /

   CALL testdb(100, ?);

   DISCONNECT;
   EXIT;
   ```

   CLPPlus スクリプトを実行するには、以下のコマンドを実行します。

   ```
   clpplus @script.sql
   ```

以下の例は、同じスクリプトの CLP バージョンを示しています。このスクリプトでは、SET SQLCOMPAT PLSQL コマンドを使用して、新規の行でスラッシュ文字 (/) が PL/SQL ステートメント終了文字として認識されるようにします。

```
CONNECT TO DB;

SET SQLCOMPAT PLSQL;

-- Semicolon is used to terminate
-- the CREATE TABLE statement:
CREATE TABLE t1 (c1 NUMBER);

-- Forward slash on a new line is used to terminate
-- the CREATE PROCEDURE statement:
CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
AS
BEGIN
 INSERT INTO t1 VALUES (num);

 message := 'The number you passed is: ' || TO_CHAR(num);
END;
/

CALL testdb(100, ?);

SET SQLCOMPAT DB2;

CONNECT RESET;
```

CLP スクリプトを実行するには、以下のコマンドを実行します。

```
db2 -tvf script.sql
```

作成した DB2 データベースが Oracle アプリケーションで使用可能になりました。有効にした互換性フィーチャーを使用することができます。

- CLPPlus インターフェースの使用を開始します。
- PL/SQL スクリプトとステートメントを実行します。
- データベース・オブジェクト定義を転送します。
- データベース・アプリケーションを使用可能にします。

Sybase アプリケーションのマイグレーション

IBM DB2 SQL Skin Feature 1.0 for applications compatible with Sybase ASE (IBM DB2 SSacSA) を使用すると、Sybase Adaptive Server Enterprise (ASE) アプリケーションをマイグレーションして、DB2 データベースに対して実行できるようにするのに役立ちます。このマイグレーション操作を行っても、既存のソース・コードはほとんど、あるいはまったく変更されることはありません。

IBM DB2 SSacSA を使用すると、Sybase アプリケーションは、新しいターゲットが DB2 データベースになっていても、依然として Sybase データベースにアクセスしているかのように動作します。ご使用の実証済みのアプリケーション・コードは変更することなく、DB2 の保管データに対して実行されます。多くの場合、必要となるのは接続パラメーターを再設定して、アプリケーションが Sybase サーバーではなく IBM DB2 SSacSA に接続できるようにすることだけです。IBM DB2

SSacSA を使用すると、手動でのマイグレーションに比べて、貴重な時間と労力の節約になります。既存のアプリケーションに変更を加える必要はほとんどないので、以下の利点があります。

- 開発者がコードを書き換える必要がないので、必要な要員がわずかで済みます。
- コード変更が少ないため新しいバグが紛れ込む機会も減り、テスト時間を短くできます。
- データベースでほとんどのマイグレーション作業が行われるので、そこだけに集中できます。

また、データベース・スキーマとデータをご使用のソース Sybase データベースから DB2 データベースにマイグレーションするには、IBM Migration Toolkit を使用できます。すべての情報が同じデータベースに格納されている場合には、マイグレーション済みの Sybase アプリケーションと DB2 ネイティブ・アプリケーションは、DB2 インスタンス、ネイティブなトリガーとプロシージャー、および DB2 拡張セキュリティーを共有できます。

その他のリソース

IBM DB2 SSacSA 資料については、Sybase application migration guides を参照してください。

IBM Migration Toolkit (MTK) について詳しくは、Migrate Now! を参照してください。

データ・タイプ

TIMESTAMP(0) に基づく DATE データ・タイプ

DATE データ・タイプは、値に時刻情報 (例えば '2009-04-01-09.43.05') が含まれる Oracle DATE データ・タイプを使用するアプリケーションをサポートするように変更されています。

TIMESTAMP(0) に基づく DATE はデータベース・レベルでサポートされるため、サポートが必要なデータベースを作成する前に有効にする必要があります。その場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数に適切な値を設定します。TIMESTAMP(0) に基づく DATE を有効にしてデータベースを作成すると、データベース構成パラメーター **date_compat** が ON に設定されます。
TIMESTAMP(0) に基づく DATE のサポートを有効にしてデータベースを作成した場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数をリセットしても、そのデータベースについてサポートを無効にすることはできません。同様に、TIMESTAMP(0) に基づく DATE のサポートを無効にして作成されたデータベースでは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定しても、サポートを有効にすることはできません。

使用可能化

TIMESTAMP(0) に基づく DATE のサポートは、データベースを作成する前に、**DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 7 (0x40) を設定することで有効になります。インスタンスを停止し、再始動するまで、レジス

トリー変数の新規設定は反映されません。

使用法

date_compat データベース構成パラメーターが ON に設定されている DB2 データベースでは、以下のサポートが有効になります。

SQL ステートメント内で DATE データ・タイプが明示的に指定されていた場合は、TIMESTAMP(0) に暗黙的にマップされます。ただし、CREATE INDEX ステートメントの *xml-index-specification* 節で SQL DATE を指定した場合は例外です。この暗黙的なマップの結果、メッセージは DATE ではなく TIMESTAMP データ・タイプを参照し、列またはルーチンのデータ・タイプを記述する演算子は、DATE ではなく TIMESTAMP を戻します。

以下の場合を除き、**date_compat** データベース構成パラメーターが ON に設定されている DB2 データベースでは、Datetime リテラルのサポートは変更されません。

- 明示的な DATE リテラルの値は、時刻部分がすべてゼロである TIMESTAMP(0) の値を戻します。例えば、`DATE '2008-04-28'` は、実際はタイム・スタンプ値 '2008-04-28-00.00.00' を表します。
- データベース・マネージャーは、'DD-MON-YYYY' および 'DD-MON-RR' に対応する日付のストリング表記 (英語のみ) を使用する形式をサポートしています (形式要素については、『TIMESTAMP_FORMAT スカラー関数』を参照してください)。例えば、日付のストリング表記として '28-APR-2008' または '28-APR-08' を使用できますが、これらは実際は TIMESTAMP(0) 値 '2008-04-28-00.00.00' を表しています。

CURRENT_DATE (および CURRENT DATE) 特殊レジスターは、CURRENT_TIMESTAMP(0) と同じ TIMESTAMP(0) 値を戻します。

TIMESTAMP 値に数値を加算、または TIMESTAMP 値から数値を減算する場合、その数値は日数を表していると見なされます。数値はどのような数値データ・タイプをとることもでき、小数値は日付の小数部分であると見なされます。例えば、`TIMESTAMP '2008-03-28 12:00:00' + 1.3` は、TIMESTAMP 値に 1 日と 7 時間 12 分を追加することを表し、その結果は '2008-03-29 19:12:00' になります。1/24 (1 時間) または 1/24/60 (1 分) など、1 日未満の時間を表すのに式を使用する場合は、DECFLOAT 算術式を使用して除算が行われるように、**number_compat** データベース構成パラメーターを ON に設定する必要があります。

date_compat モードでは、一部の関数の結果が変わります。

- ストリング引数を含む ADD_MONTHS スカラー関数は、TIMESTAMP(0) を戻します。
- DATE スカラー関数は、すべての入力タイプに対して TIMESTAMP(0) を戻します。
- ストリング引数を含む LAST_DAY スカラー関数は、TIMESTAMP(0) を戻します。
- **date_compat** モードでは DATE が常に TIMESTAMP(0) になるため、DATE 入力引数 (ADD_MONTHS、LAST_DAY、NEXT_DAY、ROUND、TRUNCATE) に基づいて DATE を戻すその他のスカラー関数は、TIMESTAMP を戻します。

- 日付値は実際は TIMESTAMP(0) であるため、日付値を追加すると TIMESTAMP(0) が戻ります。
- タイム・スタンプ値を減算すると、差を日数として表す DECFLOAT(34) が戻ります日付値は実際は TIMESTAMP(0) であるため、同様に、日付値を減算すると、日数を表す DECFLOAT(34) が戻ります。

TIMESTAMP(0) としての DATE のサポートが有効になっているデータベースでは、インポート・ユーティリティーまたはロード・ユーティリティーを使用して DATE 列 (TIMESTAMP(0) に基づいている) にデータを入力する場合、DATEFORMAT 修飾子の代わりに TIMESTAMPFORMAT 修飾子を使用する必要があります。

NUMBER データ・タイプ

NUMBER データ・タイプは、Oracle NUMBER データ・タイプを使用するアプリケーションをサポートするために導入されました。

NUMBER はデータベース・レベルでサポートされるため、サポートが必要なデータベースを作成する前に有効にする必要があります。その場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数に適切な値を設定します。NUMBER を有効にしてデータベースを作成すると、データベース構成パラメーター **number_compat** が ON に設定されます。NUMBER のサポートを有効にしてデータベースを作成した場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数をリセットしても、そのデータベースについて NUMBER のサポートを無効にすることはできません。同様に、NUMBER のサポートを有効にせずに作成されたデータベースでは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定しても、NUMBER のサポートを有効にすることはできません。

使用可能化

NUMBER データ・タイプのサポートは、データベースを作成する前に **DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 5 (0x10) を設定することで有効になります。これらの DB2 互換性フィーチャーを最大限に活用するには、値を ORA に設定します。これは推奨される設定です。インスタンスを停止し、再始動するまで、レジストリー変数の新規設定は反映されません。

使用法

number_compat データベース構成パラメーターが ON に設定されている DB2 データベースでは、以下のサポートが有効になります。

SQL ステートメント内で NUMBER データ・タイプが明示的に指定されていた場合は、以下のように暗黙的にマップされます。

- NUMBER に精度属性とスケール属性が指定されていない場合は、DECFLOAT(16) にマップされます。
- NUMBER(*p*) が指定されている場合は、DECIMAL(*p*) にマップされます。
- NUMBER(*p*,*s*) が指定されている場合は、DECIMAL(*p*,*s*) にマップされます。

サポートされている最大の精度は 31 であり、スケールは精度の値以下の正の値である必要があります。この暗黙的なマップの結果、メッセージは NUMBER ではな

くデータ・タイプ DECFLOAT および DECIMAL を参照し、列またはルーチンのデータ・タイプを記述する演算子は、NUMBER ではなく DECIMAL または DECFLOAT のいずれかを戻します。DECFLOAT(16) は、Oracle NUMBER データ・タイプよりも最大精度の値が小さいことに注意してください。表に数値を保管するために 16 桁を超える精度が必要な場合は、該当する列を DECFLOAT(34) として明示的に定義する必要があります。

number_compat 構成パラメーターが ON に設定されている DB2 データベースでも、数値リテラルのサポートは変更されません。整数、10 進数、および浮動小数点定数の規則が引き続き適用されます。それによって、10 進リテラルが 31 桁に、浮動小数点リテラルがバイナリー倍精度浮動小数点値の範囲に制限されます。DECFLOAT(34) キャストに対するストリング (CAST 仕様または DECFLOAT 関数を使用) は、DECIMAL または DOUBLE の範囲を超えて、DECFLOAT(34) の範囲までの値に対して使用することができます。

NUMBER データ値が (CAST 仕様あるいは VARCHAR または CHAR スカラー関数を使用して) 文字ストリングにキャストされると、すべての先行ゼロが結果から削除されます。

それぞれ 64 ビットのバイナリー浮動小数点値および 32 ビットのバイナリー浮動小数点値を表す、末尾が D または F である数値リテラルは、現在サポートされていません。E を含む数値リテラルにはデータ・タイプ DOUBLE があり、CAST 仕様またはキャスト関数 REAL を使用して、REAL にキャストすることができます。

number_compat 構成パラメーターが ON に設定されている DB2 データベースでは、CREATE SEQUENCE ステートメントの順序値で使用されるデフォルトのデータ・タイプは、INTEGER ではなく DECIMAL(27) になります。

number_compat 構成パラメーターが ON に設定されている DB2 データベースでは、DECIMAL または DECFLOAT データ・タイプを含むすべての算術演算子と算術または数学関数は、10 進浮動小数点を使用して正しく実行され、データ・タイプ DECFLOAT(34) の値が戻されます。これは、両方のオペランドに DECIMAL または DECFLOAT(16) データ・タイプが含まれる算術演算子にも適用されます。これは、「DB2 SQL リファレンス」の 10 進演算の説明とは異なっています(『式』の『算術演算子がある式』を参照してください)。さらに、整数データ・タイプ (SMALLINT、INTEGER、BIGINT) しか含まないすべての除算演算子は、10 進浮動小数点を使用して正しく実行でき、整数データ・タイプではなく、データ・タイプが DECFLOAT(34) である値が戻されます (整数オペランドを使用してゼロによる除算を行うと、無限大が戻され、エラーではなく警告が表示されます)。

解決処理中にデータ・タイプ DECIMAL の引数が DECFLOAT 値と見なされるように、関数解決も変更されます。これにより、関数解決を目的として、NUMBER(*p*[,*s*]) データ・タイプに対応する引数を持つ関数が、引数のデータ・タイプが NUMBER である場合と同様に、正しく処理されます。

引数の数が可変であり、結果データ・タイプが引数のデータ・タイプ・セットに基づく関数のセットにはデータ・タイプが、このような関数解決の変更は適用されません。このセットには、以下の関数が含まれています。

- COALESCE
- DECODE

- GREATEST
- LEAST
- MAX (scalar)
- MIN (scalar)
- NVL
- VALUE

number_compat 構成パラメーターを ON に設定すると、結果データ・タイプの規則は、DECIMAL 結果データ・タイプの精度が 31 を超えた場合に、DECFLOAT(34) が結果データ・タイプになるように拡張されます。これらの規則は、セット演算子 (UNION、EXCEPT(MINUS)、INTERSECT)、IN 述部の IN リストの式値、および複数行の VALUES 節内の対応する式に含まれる、対応する列にも適用されます。

データベース・サーバー上の割り当ておよびキャストに使用される丸めモードは、関連するデータ・タイプによって異なります。場合によっては切り捨てが行われます。ターゲットがバイナリー浮動小数点値 (REAL または DOUBLE) である場合は、通常どおり最も近い偶数への丸めが使用されます。その他の場合 (一般的に DECIMAL または DECFLOAT 値を使用する場合) には、丸めは **decflt_rounding** データベース構成パラメーターの値に基づいて行われます。このパラメーターのデフォルトは最も近い偶数への丸めですが、Oracle の丸めモードに従って四捨五入に設定することもできます。以下の表に、さまざまな数値の割り当てとキャストに使用される丸め方法を示します。

表 37. 数値割り当ておよびキャストの丸め方法

ソース・データ・タイプ	ターゲット・データ・タイプ			
	整数タイプ	**DECIMAL**	**DECFLOAT**	**REAL/DOUBLE**
整数タイプ	適用外	適用外	decflt_rounding	最も近い偶数への丸め
DECIMAL	decflt_rounding	decflt_rounding	decflt_rounding	最も近い偶数への丸め
DECFLOAT	decflt_rounding	decflt_rounding	decflt_rounding	最も近い偶数への丸め
REAL/DOUBLE	切り捨て	切り捨て	decflt_rounding	最も近い偶数への丸め
ストリング (キャストのみ)	適用外	decflt_rounding	decflt_rounding	最も近い偶数への丸め

DB2 10 進浮動小数点値は、IEEE 754R 標準に基づいています。DECFLOAT データの取得、および DECFLOAT データの文字ストリングへのキャストを行うと、小数点以下のゼロがすべて削除されます。

クライアント/サーバーの互換性に関する考慮事項

- NUMBER データ・タイプがサポートされている DB2 データベース・サーバーと共に作動しているクライアント・アプリケーションには、サーバーから

NUMBER データ・タイプが表示されることはありません。Oracle サーバーから NUMBER を報告した列または式は、DB2 サーバーから DECIMAL または DECFLOAT を報告します。

- Oracle 環境では丸めモードに四捨五入を使用するため、クライアントの丸めモードがサーバーの丸めモードに一致していることが重要です。これは、db2cli.ini の設定が **decflt_rounding** データベース構成パラメーターの値に一致しなければならないことを意味します。Oracle の丸めモデルに可能な限り近づけるには、ROUND_HALF_UP を指定します。

制約事項

NUMBER データ・タイプのサポートには、以下の制限事項があります。

- 31 を超える精度属性、精度属性を示すアスタリスク (*)、精度属性を超えるスケール属性、または負のスケール属性を持つ NUMBER データ・タイプはサポートされません。このようなデータ・タイプを指定しても、対応する DECIMAL 精度およびスケールはサポートされません。
- 三角関数および DIGITS スカラー関数を、精度 (DECFLOAT) が指定されていない NUMBER データ・タイプの引数によって呼び出すことはできません。
- NUMBER という名前で特殊タイプを作成することはできません。

VARCHAR2 および NVARCHAR2 データ・タイプ

VARCHAR2 および NVARCHAR2 データ・タイプは、Oracle VARCHAR2 および NVARCHAR2 データ・タイプを使用するアプリケーションをサポートするために導入されました。

VARCHAR2 および NVARCHAR2 (以降共に VARCHAR2 と呼ぶ) のサポートは、データベース・レベルでのサポートであるため、サポートを必要とするデータベースを作成する前に有効にしておく必要があります。その場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数に適切な値を設定します。VARCHAR2 サポートを有効にしてデータベースを作成すると、データベース構成パラメーター **varchar2_compat** が ON に設定されます。VARCHAR2 のサポートを有効にしてデータベースを作成した場合は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数をリセットしても、そのデータベースについて VARCHAR2 のサポートを無効にすることはできません。同様に、VARCHAR2 のサポートを無効にして作成されたデータベースでは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定しても、VARCHAR2 のサポートを有効にすることはできません。

使用可能化

VARCHAR2 データ・タイプのサポートは、データベースを作成する前に **DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 6 (0x20) を設定することで有効になります。インスタンスを停止し、再始動するまで、レジストリー変数の新規設定は反映されません。

NVARCHAR2 データ・タイプを使用するには、データベースが Unicode データベースでなければなりません。

使用法

varchar2_compat データベース構成パラメーターが ON に設定されている DB2 データベースでは、以下のサポートが有効になります。

SQL ステートメント内で VARCHAR2 データ・タイプが明示的に指定されていた場合は、VARCHAR データ・タイプに暗黙的にマップされます。VARCHAR2 の最大長は、VARCHAR の最大長 (32 672) と同じです。

SQL ステートメント内で NVARCHAR2 データ・タイプが明示的に指定されていた場合は、VARGRAPHIC データ・タイプに暗黙的にマップされます。NVARCHAR2 の最大長は、VARGRAPHIC の最大長 (16 336) と同じです。

長さが最大 254 バイトまでの文字ストリング・リテラルのデータ・タイプは CHAR です。長さが 254 バイトを超える文字ストリング・リテラルのデータ・タイプは VARCHAR です。

可変長ストリング・タイプが関係する比較では非空白埋め比較セマンティクスが使用され、固定長ストリング・タイプのみの比較では、引き続き空白埋め比較セマンティクスが使用されます。ただし、以下の 2 つの例外があります。

- カタログ・ビューのストリング列情報が関連する比較では、データベース照合に関わらず、常に空白埋め比較セマンティクスによる IDENTITY 照合を使用します。
- FOR BIT DATA 属性を持つデータ・タイプが関連するストリング比較では、常に空白埋め比較セマンティクスによる IDENTITY 照合が使用されます。

IN 述部の IN リストに対する結果タイプが固定長ストリング・データ・タイプとして解決され、IN 述部の左オペランドが可変長ストリング・データ・タイプである場合、IN リスト式は可変長ストリング・データ・タイプがあるものとして処理されます。

長さがゼロである文字ストリング値 (LOB 値以外) は、通常は NULL 値として処理されます。CHAR、NCHAR、VARCHAR、または NVARCHAR に空ストリング値を割り当てるかキャストすると、NULL 値が生成されます。

文字ストリング引数を返す関数、または文字ストリング・データ・タイプのパラメーターに基づく関数も、CHAR、NCHAR、VARCHAR、または NVARCHAR の空ストリング値を NULL 値として扱います。 **varchar2_compat** データベース構成パラメーターが ON に設定されている場合には、以下に示す一部の関数について特別な考慮事項が適用されます。

- CONCAT および連結演算子。NULL または空ストリングの値は、連結結果では無視されます。以下の表に、連結結果のタイプを示します。

表 38. 連結されたオペランドのデータ・タイプおよび長さ

オペランド	長さ属性の組み合わせ	結果
CHAR(A) CHAR(B)	<255	CHAR(A+B)
CHAR(A) CHAR(B)	>254	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B,32672))
VARCHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B,32672))

表 38. 連結されたオペランドのデータ・タイプおよび長さ (続き)

オペランド	長さ属性の組み合わせ	結果
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) CLOB(B)		CLOB(MIN(A+B, 2G))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>128	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B,16336))
VARGRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B,16336))
DBCLOB(A) CHAR(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) VARCHAR(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) CLOB(B)		DBCLOB(MIN(A+B, 1G))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2G))

- INSERT。4 番目の引数で NULL 値または空ストリングを指定すると、1 番目の引数から、2 番目の引数で示されたバイト位置から 3 番目の引数で示されたバイト数分が削除されます。
- LENGTH。LENGTH 関数によって戻される値は、文字ストリング内のバイト数です。空ストリング値の場合は、NULL 値が戻ります。
- REPLACE。すべての引数の値のデータ・タイプが CHAR、VARCHAR、GRAPHIC、VARGRAPHIC のいずれかである場合は、以下のようになります。
  - NULL 値または空ストリングである 2 番目の引数は空ストリングとみなされ、1 番目の引数が結果として戻されます。
  - NULL 値または空ストリングである 3 番目の引数は空ストリングとみなされ、2 番目の引数によってソース・ストリングから削除されます。

  値のデータ・タイプが CLOB または BLOB である引数があり、いずれかの引数が NULL 値である場合、結果は NULL 値になります。REPLACE 関数の 3 つの引数すべてを指定する必要があります。
- SUBSTR。最初の引数の入力が文字ストリングである SUBSTR への参照は、SUBSTRB への呼び出しに置き換えられます。最初の引数の入力がグラフィック・ストリングである SUBSTR への参照は、変更されずに残されます。この場合、2 番目の引数の値を 1 より小さくすること、または 3 番目の引数の値を 0 より小さくすることはできません。
- TRANSLATE。*from-string-exp* は 2 番目の引数、*to-string-exp* は 3 番目の引数です。*to-string-exp* が *from-string-exp* より短い場合は、*char-string-exp* (1 番目の引数) にある *from-string-exp* の余分な文字が削除されます。つまり、異なる埋め込み文字が 4 番目の引数で指定されない限り、デフォルトの *pad-char* 引数が実質的に空ストリングになります。
- TRIM。TRIM 関数呼び出しのトリム文字引数が NULL 値または空ストリングである場合は、NULL 値が戻されます。

ALTER TABLE ステートメントまたは CREATE TABLE ステートメントで、VARCHAR または VARGRAPHIC データ・タイプで定義された列に、明示的な値がなく DEFAULT 節が指定された場合、デフォルト値はブランク文字になります。

データベース構成パラメーター **varchar2_compat** が ON に設定されている場合、空ストリングはブランク文字に変換されます。以下に例を示します。

- 可視データ・パーティションの場合、SYSCAT.DATAPARTITIONS.STATUS にはブランク文字が 1 つ含まれます
- パッケージ・バージョンが明示的に設定されていない場合、SYSCAT.PACKAGES.PKGVERSION にはブランク文字が 1 つ含まれます
- コンパイル・オプションが設定されていない場合、SYSCAT.ROUTINES.COMPILE_OPTIONS には NULL 値が 1 つ含まれます

注: SQL ステートメントでパラメーター・マーカーを使用すると、VARCHAR2 の使用に影響するデータ・タイプ変換が行われる場合があります。例えば、入力値が長さゼロの VARCHAR で、LOB に変換された場合、結果は NULL 値になります。ただし、入力値が長さゼロの LOB で、LOB に変換された場合には、結果は長さゼロの LOB になります。入力値のデータ・タイプは、据え置き準備に影響される場合があります。

制約事項

VARCHAR2 データ・タイプおよび関連する文字ストリングの処理のサポートには、以下の制約事項があります。

- VARCHAR2 の長さ属性修飾子 CHAR は受け入れられません。
- **varchar2_compat** データベース構成パラメーターを ON に設定した場合には、LONG VARCHAR および LONG VARGRAPHIC データ・タイプはサポートされません (ただし明示的にはブロックされません)。

文字定数および GRAPHIC 定数の処理

文字定数および GRAPHIC 定数に CHAR と GRAPHIC のデータ・タイプがそれぞれ割り当てられることが予期されるアプリケーションをサポートするために、それらの定数の構文解析を行う別の方法が導入されました。

使用可能化

このサポートは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 9 (0x100) を設定することで有効になります。
DB2_COMPATIBILITY_VECTOR の推奨設定は ORA で、すべての互換性ビットが設定されます。

このサポートを有効にすると、長さが 254 バイト以下の文字定数または GRAPHIC ストリング定数のデータ・タイプがそれぞれ CHAR または GRAPHIC になります。長さが 254 バイトより大きい文字定数または GRAPHIC ストリング定数のデータ・タイプは、それぞれ VARCHAR または VARGRAPHIC になります。このデータ・タイプ割り当てによって一部の SQL ステートメントのタイプが変わる場合があるため、データベースのこのレジストリー変数設定は切り替えないことを強くお勧めします。

外部結合演算子

DB2_COMPATIBILITY_VECTOR レジストリー変数が外部結合演算子 (+) をサポートするように設定されている場合、照会では WHERE 節の述部内でこの代替構文を使用することができます。

外部結合構文は可能な場所であればどこでも使用できますが、外部結合演算子は他のリレーショナル・データベース・ベンダーのアプリケーションを DB2 上で使用可能にする場合にのみ使用します。

外部結合演算子 (+) は、一般的に 2 つの表参照の列を参照する述部内の列名に続いて適用されます。

- 表 T1 および T2 の左外部結合を実行する照会を記述するには、コンマで区切られた FROM 節に両方の表を含め、T1 も参照する述部内の T2 のすべての列に外部結合演算子を適用します。以下に例を示します。

  ```
  SELECT * FROM T1
    LEFT OUTER JOIN T2 ON T1.PK1 = T2.FK1
      AND T1.PK2 = T2.FK2
  ```

  これは以下と等価です。

  ```
  SELECT * FROM T1, T2
    WHERE T1.PK1 = T2.FK1(+)
      AND T1.PK2 = T2.FK2(+)
  ```

- 表 T1 および T2 の右外部結合を実行する照会を記述するには、コンマで区切られた FROM 節に両方の表を含め、T2 も参照する述部内の T1 のすべての列に外部結合演算子を適用します。以下に例を示します。

  ```
  SELECT * FROM T1
    RIGHT OUTER JOIN T2 ON T1.FK1 = T2.PK1
      AND T1.FK2 = T2.PK2
  ```

  これは以下と等価です。

  ```
  SELECT * FROM T1, T2
    WHERE T1.FK1(+) = T2.PK1
      AND T1.FK2(+) = T2.PK2
  ```

外部結合演算子を含む列がある表参照は、*NULL プロデューサー* と呼ばれる場合があります。

AND 演算子で区切られる述部のセットは *AND ファクター* と呼ばれます。WHERE 節に AND 演算子がない場合は、WHERE 節内の一連の述部が唯一の AND ファクターであると見なされます。

外部結合演算子を使用する場合は、以下の規則が適用されます。

- WHERE 述部は、ANDed Boolean 値ファクターの細分単位と見なされます。
- 各 Boolean 値が参照できる表参照は 2 つまでです。つまり `T1.C11 + T2.C21 = T3.C3(+)` は許可されていません。
- 各表は、他の最大 1 つの表に関して NULL プロデューサーになることができます。表を 3 番目の表に結合する場合は、外部表にする必要があります。
- `T1.A(+) = 5` などのローカルの述部を指定することはできますが、結合と合わせて実行されます。(+) のないローカル述部は、結合の後に実行できます。
- 外部結合 Boolean 値項の相関は許可されていません。

- 外部結合演算子は、明示的な JOIN 構文と同じ副選択で指定することはできません。
- 外部結合演算子は、同じ副選択の FROM 節で指定されている表参照に関連付けられている列の WHERE 節でのみ指定することができます。
- AND ファクターは、NULL プロデューサーとして 1 つの表参照のみ指定することができます。外部結合演算子が付いた列参照は、同じ表参照を使用する必要があります。
- 外部結合演算子を含む AND ファクターは、最大 2 つの表参照を参照することができます。
- 2 つの表の間の外部結合で複数の AND ファクターが必要になる場合は、それらの AND ファクターのすべてで外部結合演算子を指定する必要があります。AND ファクターが外部結合演算子を指定しない場合は、外部結合の結果によって処理されます。
- 外部結合演算子は、式全体に適用することはできません。 AND ファクター内では、同じ表による列参照には、それぞれ外部結合演算子 (例えば `T1.COL1 (+) - T1.COL2 (+) = T2.COL1`) を付ける必要があります。
- 1 つの表参照だけを行う述部を持つ AND ファクターは、NULL プロデューサーと同じ表参照を含み、外部表として別の表参照がある他の AND ファクターが少なくとも 1 つ存在すれば、外部結合演算子を指定することができます。
- 1 つの表参照だけを行う述部を持ち、外部結合演算子がない AND ファクターは、結合の結果によって処理されます。
- 表参照を照会内の 1 つの他の表参照に対する NULL プロデューサーとして使用できるのは 1 回のみです。
- 循環を形成する別個の外部結合内では、同じ表参照を外部表と NULL プロデューサーの両方として使用することはできません。循環は、一連の述部が以前の表参照に戻された場合に、複数の結合にわたって形成される場合があります。以下に例を示します。

```
SELECT ... FROM T1,T2,T3
  WHERE T1.a1 = T2.b2(+)
    AND T2.b2 = T3.c3(+)
    AND T3.c3 = T1.a1(+)   -- 不正な循環
```

  この例では、最初の述部で T1 が外部表になっており、3 番目の述部で再び T1 に戻っています。T2 が最初の述部の NULL プロデューサーと 2 番目の述部の外部表の両方として使用されていますが、これ自体は循環にはならないことに注意してください。
- 外部結合演算子を含む AND ファクターは、結合表で定義されている ON 節の結合条件の規則に従う必要があります。
- 外部結合演算子は、同じ副選択の FROM 節で指定されている表参照に関連付けられている列の、WHERE 節でのみ指定できます。

階層照会

階層照会は再帰的照会の 1 つの形式であり、CONNECT BY 節を使用して、リレーショナル・データから部品表などの階層を検索することを可能にします。

階層照会のサポートは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定することで有効になります。これにより、疑似列 (『疑似列』 を参照)、単項演算子 (『単項演算子』 を参照)、および SYS_CONNECT_BY_PATH スカラー関数を使用した、CONNECT BY 構文を指定できるようになります。

CONNECT BY 再帰では、シード (開始) と再帰手順 (接続) で同じ副照会が使用されます。この組み合わせによって、bills-of-material、reports-to-chains、E メール・スレッドなどの再帰を表す簡潔な方式が得られます。

CONNECT BY 再帰は、循環が発生するとエラーを戻します。*循環*は、直接的または間接的に行が自身を作成すると発生します。オプションの CONNECT BY NOCYCLE 節を使用すると、再帰時に重複している行が無視され、循環およびエラーが回避されます。DB2の再帰に階層照会をマップする方法について詳しくは、Port CONNECT BY to DB2 を参照してください。

疑似列

*疑似列*は、特定のコンテキストで意味を持ち、同じネーム・スペースを列および変数として共有する、修飾または非修飾の ID です。非修飾の ID が列または変数を識別しない場合は、疑似列が識別されるかどうかをチェックします。

LEVEL は、階層照会で使用する疑似列です。LEVEL 疑似列は、行が作成された階層内の再帰手順を戻します。 START WITH 節によって作成されたすべての行は、値 1 を戻します。同様に、CONNECT BY 節の最初の反復を適用して作成された行は、値 2 を戻します。列のデータ・タイプは INTEGER NOT NULL です。

LEVEL は階層照会のコンテキストで指定する必要がありますが、START WITH 節で、CONNECT_BY_ROOT 演算子の引数として、または SYS_CONNECT_BY_PATH 関数 (SQLSTATE 428H4) の引数として指定することはできません。

単項演算子

階層照会をサポートする単項演算子には以下のものがあります。

- CONNECT_BY_ROOT
- PRIOR

関数

階層照会をサポートする新しい関数には、以下のものがあります。

- SYS_CONNECT_BY_PATH スカラー関数

副選択

階層照会のサポートには、副選択に対する以下の拡張が含まれています。

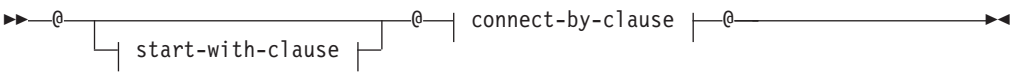
- この副選択には、新しい 406 ページの『hierarchical-query-clause』 が含まれています。
- 副選択の節は、以下の順序で処理されます。
  1. FROM 節
  2. *hierarchical-query-clause*

3. WHERE 節
4. GROUP BY 節
5. HAVING 節
6. SELECT 節
7. ORDER BY 節
8. FETCH FIRST 節

- 副選択に *hierarchical-query-clause* が含まれている場合には、WHERE 節内の述部を処理する順序に特殊な規則が適用されます。*search-condition* は、AND 条件 (結合) と共に述部に組み込まれます。述部が暗黙的な結合述部である場合 (FROM 節内の複数の表を参照する場合)、この述部は *hierarchical-query-clause* の前に適用されます。 FROM 節内で最大 1 つの表を参照する述部は、*hierarchical-query-clause* の中間結果表に適用されます。

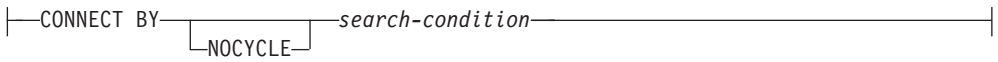
  結合を含む階層照会は、WHERE 節述部の適用に混乱が生じないように、ON 節を使用した明示的な結合表を使用して記述するようにします。
- 新しい ORDER SIBLINGS BY 節は、副選択に *hierarchical-query-clause* が含まれている場合に指定できます。この節は、順序が階層内の兄弟関係にのみ適用されることを指定します。

hierarchical-query-clause



start-with-clause:

connect-by-clause:



hierarchical-query-clause を含む副選択を階層照会と呼びます。最初の中間結果表 H_1 を生成すると、H_n と R を結合することで、後続の中間結果表 H_2、H_3 などが生成されます。その場合は、*connect-by-clause* を結合条件として使用して H_{n+1} を生成します。R は、副選択の FROM 節、および WHERE 節内の結合述部の結果です。H_{n+1} によって空の結果表が生成されると、プロセスは停止します。*hierarchical-query-clause* の結果表 H は、すべての H_i の UNION ALL です。

start-with-clause は、*search-condition* が真である R 行で構成された階層照会の中間結果表 H_1 を指定します。*start-with-clause* が指定されていない場合、H_1 は中間結果表 R 全体になります。

connect-by-clause は、検索条件を使用して H_n と R を結合することで、H_n から中間結果表 H_{n+1} を生成します。

単項演算子 PRIOR は、前回の再帰手順である H_n に対する列参照と、R に対する列参照を識別するために使用します。例えば次のように使用します。

```
CONNECT BY MGRID = PRIOR EMPID
```

MGRID は R によって解決され、EMPID は前の中間結果表 H_n によって解決されます。

start-with-clause および *connect-by-clause* 内の *search-condition* の規則は、*connect-by-clause* 内で OLAP 仕様を指定できないことを除き、WHERE 節内の規則と同じです (SQLSTATE 42903)。

中間結果表 H_{n+1} が、ある階層パスの R から、すでにその階層パスにある R からの行と同じ行を戻した場合は、エラーが戻されます (SQLSTATE 560CO)。

NOCYCLE キーワードが指定されている場合はエラーは戻されませんが、中間結果表 H_{n+1} には反復された行は含まれません。

DB2 は、最大 64 レベルの再帰をサポートしています (SQLSTATE 54066)。

階層照会である副選択は、選択リスト内で明示的な ORDER BY 節、GROUP BY 節または HAVING 節、あるいは DISTINCT キーワードを使用することで順序が無効にならない限り、中間結果表を部分順序で戻します。部分順序は、特定の階層について H_{n+1} で生成された行が、それらの行を生成した H_n の直後に配列されるように行を戻します。ORDER SIBLINGS BY 節は、同じ親から生成された行のセット内で順序を強制するために使用できます。

階層照会の使用に関する制限事項

- 階層照会は、マテリアライズ照会表ではサポートされていません (SQLSTATE 428EC)。
- CONNECT BY 節は、XML 関数または XQuery と合わせて使用することはできません (SQLSTATE 428H4)
- ある順序の NEXT VALUE 式は、以下のもので指定することはできません (SQLSTATE 428F9)
  - CONNECT_BY_ROOT 演算子または SYS_CONNECT_BY_PATH 関数のパラメーター・リスト
  - START WITH 節および CONNECT BY 節

例

- 以下の reports-to-chain の例は、connect-by 再帰を示しています。この例は、以下のデータを使用して作成および移植された MY_EMP という表に基づいています。

```
CREATE TABLE MY_EMP(
  EMPID  INTEGER NOT NULL PRIMARY KEY,
  NAME   VARCHAR(10),
  SALARY DECIMAL(9, 2),
  MGRID  INTEGER);

INSERT INTO MY_EMP VALUES ( 1, 'Jones',    30000, 10);
INSERT INTO MY_EMP VALUES ( 2, 'Hall',     35000, 10);
INSERT INTO MY_EMP VALUES ( 3, 'Kim',      40000, 10);
INSERT INTO MY_EMP VALUES ( 4, 'Lindsay',  38000, 10);
INSERT INTO MY_EMP VALUES ( 5, 'McKeough', 42000, 11);
```

```
INSERT INTO MY_EMP VALUES ( 6, 'Barnes',   41000, 11);
INSERT INTO MY_EMP VALUES ( 7, 'O''Neil',  36000, 12);
INSERT INTO MY_EMP VALUES ( 8, 'Smith',    34000, 12);
INSERT INTO MY_EMP VALUES ( 9, 'Shoeman',  33000, 12);
INSERT INTO MY_EMP VALUES (10, 'Monroe',   50000, 15);
INSERT INTO MY_EMP VALUES (11, 'Zander',   52000, 16);
INSERT INTO MY_EMP VALUES (12, 'Henry',    51000, 16);
INSERT INTO MY_EMP VALUES (13, 'Aaron',    54000, 15);
INSERT INTO MY_EMP VALUES (14, 'Scott',    53000, 16);
INSERT INTO MY_EMP VALUES (15, 'Mills',    70000, 17);
INSERT INTO MY_EMP VALUES (16, 'Goyal',    80000, 17);
INSERT INTO MY_EMP VALUES (17, 'Urbassek', 95000, NULL);
```

以下の照会によって、Goyal のすべての従業員名、および reports-to-chain などの追加情報が戻されます。

```
1 SELECT NAME,
2        LEVEL,
3        SALARY,
4        CONNECT_BY_ROOT NAME AS ROOT,
5        SUBSTR(SYS_CONNECT_BY_PATH(NAME, ':'), 1, 25) AS CHAIN
6   FROM MY_EMP
7   START WITH NAME = 'Goyal'
8   CONNECT BY PRIOR EMPID = MGRID
9   ORDER SIBLINGS BY SALARY;

NAME       LEVEL       SALARY      ROOT  CHAIN
---------- ----------- ----------- ----- ---------------
Goyal                1    80000.00 Goyal :Goyal
Henry                2    51000.00 Goyal :Goyal:Henry
Shoeman              3    33000.00 Goyal :Goyal:Henry:Shoeman
Smith                3    34000.00 Goyal :Goyal:Henry:Smith
O'Neil               3    36000.00 Goyal :Goyal:Henry:O'Neil
Zander               2    52000.00 Goyal :Goyal:Zander
Barnes               3    41000.00 Goyal :Goyal:Zander:Barnes
McKeough             3    42000.00 Goyal :Goyal:Zander:McKeough
Scott                2    53000.00 Goyal :Goyal:Scott
```

行 7 と 8 が、この再帰のコアになっています。オプションの START WITH 節は、ソース表で再帰のシードに使用される WHERE 節を記述します。この場合は、従業員 Goyal の行だけが選択されます。START WITH 節を省略すると、ソース表全体が再帰のシードに使用されます。CONNECT BY 節は、既存の各行の中で、次の行のセットを検索する方法を示します。単項演算子 PRIOR は、前の手順の値と現在の手順の値を区別するために使用します。PRIOR は、EMPID を前の再帰手順の従業員 ID として識別し、MGRID を現在の再帰手順から生成された ID として識別します。

行 2 の LEVEL は、現在の再帰レベルを示す疑似列です。

CONNECT_BY_ROOT は、最初の再帰手順でのように、常に自身の引数の値 (つまり、明示的または暗黙的な START WITH 節によって戻される値) を戻す単項演算子です。

SYS_CONNECT_BY_PATH() は、2 番目の引数を最初の引数の前に付加し、前の再帰手順で生成された値に結果を付加するバイナリー関数です。この引数は文字タイプである必要があります。

明示的に無効にしない限り、connect-by recursion は結果セットを部分順序で戻します。つまり、再帰手順によって生成された行は、常に生成元の行の後になりま

す。同じ再帰レベルにある兄弟間は順不同になります。第 9 行の ORDER SIBLINGS BY 節は、これらの兄弟間の順序を調整し、可能な場合は部分順序をさらに全順序にします。

- DEPARTMENT 表の組織構造を戻します。階層を視覚化するために、部門のレベルを使用します。

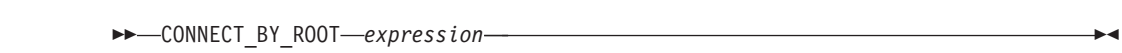
```
SELECT LEVEL, CAST(SPACE((LEVEL - 1) * 4) || '/' || DEPTNAME
    AS VARCHAR(40)) AS DEPTNAME
  FROM DEPARTMENT
  START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

この照会は以下のように戻ります。

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 /SPIFFY COMPUTER SERVICE DIV.
          2     /PLANNING
          2     /INFORMATION CENTER
          2     /DEVELOPMENT CENTER
          3         /MANUFACTURING SYSTEMS
          3         /ADMINISTRATION SYSTEMS
          2     /SUPPORT SERVICES
          3         /OPERATIONS
          3         /SOFTWARE SUPPORT
          3         /BRANCH OFFICE F2
          3         /BRANCH OFFICE G2
          3         /BRANCH OFFICE H2
          3         /BRANCH OFFICE I2
          3         /BRANCH OFFICE J2
```

CONNECT_BY_ROOT 単項演算子

CONNECT_BY_ROOT 単項演算子は、階層照会でのみ使用できます (SQLSTATE 428H4)。この演算子は、階層内のすべての行について、行のルート祖先の式を戻します。



expression
NEXT VALUE 式、階層照会構成体 (LEVEL 疑似列など)、SYS_CONNECT_BY_PATH 関数、または OLAP 関数を含まない式 (SQLSTATE 428H4)。

演算子の結果タイプは、式の結果タイプになります。

規則:

- CONNECT_BY_ROOT 演算子を階層照会の START WITH 節または CONNECT BY 節で指定することはできません (SQLSTATE 428H4)。
- CONNECT_BY_ROOT 演算子を SYS_CONNECT_BY_PATH 関数に対する引数として指定することはできません (SQLSTATE 428H4)。

注:

- CONNECT_BY_ROOT 演算子は、どの 2 項演算子よりも優先されます。したがって、2 項演算子 (+ や || など) を引数にして式を渡すには、括弧を使用する必要があります。以下に例を示します。

```
CONNECT_BY_ROOT FIRSTNME || LASTNAME
```

この式は以下のものと同等であるため、階層内にある実際の行の LASTNAME の値に連結された、ルート祖先行の FIRSTNME の値を戻します。

```
(CONNECT_BY_ROOT FIRSTNME) || LASTNAME
```

以下とは違います。

```
CONNECT_BY_ROOT (FIRSTNME || LASTNAME)
```

以下に例を示します。

- DEPARTMENT 表の部門の階層およびそれらのルートの部門を戻します。

```
SELECT CONNECT_BY_ROOT DEPTNAME AS ROOT, DEPTNAME
  FROM DEPARTMENT START WITH DEPTNO IN ('B01','C01','D01','E01')
  CONNECT BY PRIOR DEPTNO = ADMRDEPT
```

この照会は以下のように戻ります。

```
ROOT                DEPTNAME
------------------- ----------------------
PLANNING            PLANNING
INFORMATION CENTER  INFORMATION CENTER
DEVELOPMENT CENTER  DEVELOPMENT CENTER
DEVELOPMENT CENTER  MANUFACTURING SYSTEMS
DEVELOPMENT CENTER  ADMINISTRATION SYSTEMS
SUPPORT SERVICES    SUPPORT SERVICES
SUPPORT SERVICES    OPERATIONS
SUPPORT SERVICES    SOFTWARE SUPPORT
SUPPORT SERVICES    BRANCH OFFICE F2
SUPPORT SERVICES    BRANCH OFFICE G2
SUPPORT SERVICES    BRANCH OFFICE H2
SUPPORT SERVICES    BRANCH OFFICE I2
SUPPORT SERVICES    BRANCH OFFICE J2
```

PRIOR 単項演算子

PRIOR 単項演算子は、階層照会の CONNECT BY 節でのみ使用できます (SQLSTATE 428H4)。

CONNECT BY 節は、階層照会の中間結果表 H_n と、FROM 節で指定されたソース結果表との内部結合を実行します。FROM 節で参照される表に対するすべての列参照のうち、PRIOR 演算子に対する引数については、H_n の範囲を超えていると見なされます。

中間結果表 H_n の主キーは、一般的にソース結果表の外部キーに結合され、階層を再帰的にトラバースします。

```
CONNECT BY PRIOR T.PK = T.FK
```

主キーが複合キーである場合は、各列の先頭に PRIOR を付ける必要があります。

```
CONNECT BY PRIOR T.PK1 = T.FK1 AND PRIOR T.PK2 = T.FK2
```

expression
NEXT VALUE 式、階層照会構成体 (LEVEL 疑似列など)、SYS_CONNECT_BY_PATH 関数、または OLAP 関数を含まない式 (SQLSTATE 428H4)。

演算子の結果データ・タイプは、式の結果データ・タイプになります。

注:

- PRIOR 演算子は、どの 2 項演算子よりも優先されます。したがって、2 項演算子 (+ や || など) を引数にして式を渡すには、括弧を使用する必要があります。以下に例を示します。

```
PRIOR FIRSTNME || LASTNAME
```

この式は以下のものと同等であるため、階層内にある実際の行の LASTNAME の値に連結された、先行する行の FIRSTNME の値を戻します。

```
(PRIOR FIRSTNME) || LASTNAME
```

以下とは違います。

```
PRIOR (FIRSTNME || LASTNAME)
```

以下に例を示します。

- DEPARTMENT 表の部門階層が戻ります。

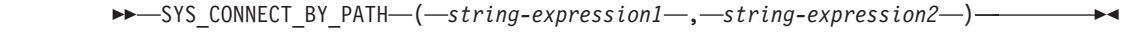
```
SELECT LEVEL, DEPTNAME
  FROM DEPARTMENT START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

この照会は以下のように戻ります。

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 SPIFFY COMPUTER SERVICE DIV.
          2 PLANNING
          2 INFORMATION CENTER
          2 DEVELOPMENT CENTER
          3 MANUFACTURING SYSTEMS
          3 ADMINISTRATION SYSTEMS
          2 SUPPORT SERVICES
          3 OPERATIONS
          3 SOFTWARE SUPPORT
          3 BRANCH OFFICE F2
          3 BRANCH OFFICE G2
          3 BRANCH OFFICE H2
          3 BRANCH OFFICE I2
          3 BRANCH OFFICE J2
```

SYS_CONNECT_BY_PATH

SYS_CONNECT_BY_PATH 関数 (SYSIBM スキーマ) は、階層照会でルート行からこの行までのパスを表すストリングを作成するために使用されます。



LEVEL *n* の任意の行のストリングは、以下のように作成します。

- ステップ 1 (最初の中間結果表 H_1 のルート行の値を使用):

$path_1$:= *string-expression2* || *string-expression1*

- ステップ *n* (中間結果表 H_n の行に基づく):

  $path_n$:= $path_{n-1}$ || *string-expression2* || *string-expression1*

string-expression1
行を特定する文字ストリング式。この式に、順序に対する NEXT VALUE 式 (SQLSTATE 428F9)、LEVEL 疑似列または CONNECT_BY_ROOT 演算子などの階層照会構成体 (SQLSTATE 428H4)、OLAP 関数、または集約関数 (SQLSTATE 428H4) を含めることはできません。

string-expression2
区切り文字として機能する定数ストリング。この式に、順序に対する NEXT VALUE 式 (SQLSTATE 428F9)、LEVEL 疑似列または CONNECT_BY_ROOT 演算子などの階層照会構成体 (SQLSTATE 428H4)、OLAP 関数、または集約関数 (SQLSTATE 428H4) を含めることはできません。

結果は可変長文字ストリングになります。結果のデータ・タイプの長さ属性は、1000 と、*string-expression1* の長さ属性のどちらか長いほうになります。

規則:

- SYS_CONTEXT_BY_PATH 関数を階層照会のコンテキスト以外で使用することはできません (SQLSTATE 428H4)。
- この関数を START WITH 節または CONNECT BY 節で使用することはできません (SQLSTATE 428H4)。

以下に例を示します。

- DEPARTMENT 表の部門階層が戻ります。

```
SELECT CAST(SYS_CONNECT_BY_PATH(DEPTNAME, '/')
    AS VARCHAR(76)) AS ORG
  FROM DEPARTMENT START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

この照会は以下のように戻ります。

```
ORG
------------------------------------------------------------------------
/SPIFFY COMPUTER SERVICE DIV.
/SPIFFY COMPUTER SERVICE DIV./PLANNING
/SPIFFY COMPUTER SERVICE DIV./INFORMATION CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/MANUFACTURING SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/ADMINISTRATION SYSTEMS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/OPERATIONS
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/SOFTWARE SUPPORT
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE F2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE G2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE H2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE I2
/SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE J2
```

データベース構成パラメーター

新しいデータベース構成パラメーターは、複数の互換性フィーチャーの状況を示します。

新しいパラメーターには以下のものがあります。

- **date_compat**。TIMESTAMP(0) データ・タイプに関連付けられている DATE 互換性セマンティクスが、接続されているデータベースに適用されているかどうかを示します。
- **number_compat**。NUMBER データ・タイプに関連付けられている互換性セマンティクスが、接続されているデータベースに適用されているかどうかを示します。
- **varchar2_compat**。VARCHAR2 データ・タイプに関連付けられている互換性セマンティクスが、接続されているデータベースに適用されているかどうかを示します。

各パラメーターの値は、**DB2_COMPATIBILITY_VECTOR** レジストリー変数の設定に基づいて、データベース作成時に決定されます。この値を変更することはできません。

ROWNUM 疑似列

DB2 は、ROWNUM に対する未解決および非修飾の列参照を OLAP 仕様の ROW_NUMBER() OVER() に変換します。

ROWNUM 疑似列のサポートは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定することで有効になります。

ROWNUM と ROW_NUMBER() OVER() はどちらも副選択の WHERE 節で許可されており、結果セットのサイズを制限するのに役立ちます。

ROWNUM が WHERE 節で使用されていて、同じ副選択内に ORDER BY 節がある場合、オーダーは ROWNUM 述語が評価される前に適用されます。これは、WHERE 節内の ROW_NUMBER() OVER() 関数についても同様です。

OLAP 仕様 ROW_NUMBER() OVER() が WHERE 節内で使用されている場合には、window-order-clause も window-partition-clause も指定することができません。

注

- 'ROWNUM' に対する非修飾参照を ROW_NUMBER() OVER() として解釈する前に、DB2 は以下のいずれかとして参照を解決します。
  - 現在の SQL 照会内の列
  - ローカル変数
  - ルーチン・パラメーター
  - グローバル変数
- ROWNUM 疑似列のサポートが有効になっている場合は、'ROWNUM' を列名または変数名として使用しないでください。

例 1

ROWNUM および外部結合演算子照会を使用するアプリケーションをサポートするには、**DB2_COMPATIBILITY_VECTOR** レジストリー変数を設定します。Oracle との最大の互換性を実現するには、値を ORA に設定します。これは推奨される設定です。

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

新しい設定は、このインスタンスが停止して再始動するまでは有効になりません。

例 2

接続しているデータベースについて ROWNUM 疑似列のサポートが有効になっているとして、一時表に保管されている結果セットの 20 ～ 40 行目を取得します。

```
SELECT TEXT FROM SESSION.SEARCHRESULTS
  WHERE ROWNUM BETWEEN 20 AND 40
  ORDER BY ID
```

ROWNUM は ORDER BY 節に影響を受けます。

DUAL 表

DB2 データ・サーバーは、「DUAL」へのすべての非修飾表参照を組み込みビューとして解決し、「DUMMY」という名前で値が「X」である 1 行 1 列を戻します。

DUAL 表への非修飾表参照は、*DB2_COMPATIBILITY_VECTOR* レジストリー変数を設定することによって SYSIBM.DUAL として解決されます。

DUAL という名前でユーザー定義表が存在する場合、参照が明示的に修飾されている場合に限り、DB2 サーバーはユーザー定義表への表参照を解決します。

例 1

DUAL から選択して乱数を生成します。

```
SELECT RAND() AS RANDOM_NUMBER FROM DUAL
```

例 2

CURRENT SCHEMA 特殊レジスターの値を取り出します。

```
SET SCHEMA = MYSCHEMA;
SELECT CURRENT SCHEMA AS CURRENT_SCHEMA FROM DUAL;
```

インセンシティブ・カーソル

バージョン 9.7 フィックスパック 2 以降、OPEN の際にカーソルをマテリアライズして、そのカーソルを後続のステートメントに対してインセンシティブにするオプションが使用できます。

他のデータ変更ステートメントに対してカーソルをインセンシティブにする場合、結果セットは OPEN 時にマテリアライズされ、カーソルは読み取り専用カーソルと

して動作します。インセンシティブ・カーソルのサポートがない場合、DB2 カーソルは OPEN 時に必ずマテリアライズされるわけではなくなります。そのため、照会の実行時に、すぐにカーソルをマテリアライズするようなリレーショナル・データベースとは対照的に、DB2 では結果セットが異なってしまう恐れがあります。例えば、Sybase TSQL には、呼び出し側の結果セットを生成するバッチ・コードまたはプロシージャー・コードから照会を発行する機能があります。照会はすぐにマテリアライズされます。それで、ブロック内の他のステートメントが結果に影響を与える可能性はなく、照会内で参照された同じ表に対して、削除などのステートメントの発行はできないものと想定されます。インセンシティブ・カーソルを使用しないで同じシナリオを実行する場合、カーソルからの結果セットは Sybase の結果と異なります。

インセンシティブ・カーソルのサポートは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 13 (0x1000) を設定することで有効になります。このビットが設定されると、WITH RETURN として定義されたすべてのカーソルは、FOR UPDATE として明示的に指定されない限りは INSENSITIVE になります。インスタンスを停止し、再始動するまで、レジストリー変数の新規設定は反映されません。

DECLARE CURSOR ステートメントが拡張され、カーソルを INSENSITIVE として定義できるようになりました。インセンシティブ・カーソルの宣言がサポートされているのは、コンパウンド SQL (コンパイル済み) ステートメントのコンテキストに限定されます。

BIND コマンドの STATICREADONLY オプションで、INSENSITIVE の指定ができるようになりました。 STATICREADONLY INSENSITIVE でバインドされるパッケージにより、読み取り専用で未確定のカーソルすべてはインセンシティブになります。このバインド・オプションは、レジストリー変数 DB2_SQLROUTINE_PREPOPTS とプロシージャー SET_ROUTINE_OPTS でもサポートされています。それにより、SQL ルーチンは、静的 SQL として発行される読み取り専用で未確定のカーソルすべてを OPEN 時にマテリアライズできます。

制約事項

INSENSITIVE キーワードを指定できるのは、コンパウンド SQL (コンパイル済み) ステートメントで使用される DECLARE CURSOR ステートメントのみです。プリコンパイラーではサポートされていません。インセンシティブ順方向カーソルを指定するための CLI および JDBC (カーソル属性と結果セット属性の両方) には変更はありません。

例

以下のコードでは、DELETE ステートメントの開始前にクライアントに対して、SELECT ステートメントの結果セット全体が戻ります。

```
BEGIN
 DECLARE res INSENSITIVE CURSOR WITH RETURN TO CLIENT FOR
  SELECT * FROM T;
  OPEN T;
  DELETE FROM T;
END
```

INOUT パラメーター

バージョン 9.7 フィックスパック 2 以降、プロシージャーではデフォルトが定義された INOUT パラメーターを使用でき、その後このプロシージャーをこのパラメーターの引数を指定しないでも呼び出すことができます。

プロシージャー宣言では、IN パラメーターに加えて、INOUT パラメーターでも DEFAULT 式を指定できます。デフォルトが定義された INOUT パラメーターが使用されているプロシージャーは、こうしたパラメーターに対応する引数を指定しないでも、呼び出しが可能です。INOUT パラメーターに対応する引数が指定されていない場合、または引数が DEFAULT キーワードの場合、提供されたデフォルト式 (何も指定されなかった場合には NULL) を使用してプロシージャー内のパラメーターが初期化され、プロシージャーの終了時にはこのパラメーターに対して何も値は返されません。

INOUT パラメーターのサポートは、**DB2_COMPATIBILITY_VECTOR** レジストリー変数のビット位置番号 14 (0x2000) を設定することで有効になります。インスタンスを停止し、再始動するまで、レジストリー変数の新規設定は反映されません。

制約事項

DEFAULT キーワードがサポートされているのは、プロシージャー内の INOUT パラメーターで、関数内ではサポートされていません。

例

オプションの INOUT パラメーターを指定したプロシージャーの作成

```
CREATE OR REPLACE PROCEDURE paybonus
  (IN empid INTEGER,
  IN percentbonus DECIMAL(2, 2),
  INOUT budget DECFLOAT DEFAULT NULL)
  ...
```

このプロシージャーは、従業員の給与から賞与額を算出し、賞与を支給し、部門別予算から賞与を差し引きます。予算が指定されていないと、その部分は無視されます。このプロシージャーは以下のように呼び出せます。

```
CALL paybonus(12, 0.05, 50000);
CALL paybonus(12, 0.05, DEFAULT);
CALL paybonus(12, 0.05);
```

コミットされているセマンティクスによる並行性の向上

ロック・タイムアウトとデッドロックが生じる可能性があるのは、行レベルのロックがある CS 分離レベルです。特に、そのような問題を回避するように設計されていないアプリケーションの場合に生じ得ます。一部の高スループットのデータベース・アプリケーションではトランザクション処理の際に発行されるロックの待機を許容できません。また、アプリケーションによっては、非コミット・データの処理を許容できないものの読み取りトランザクション用に非ブロッキング動作を依然として必要とするものがあります。

新しい *currently committed* セマンティクスでは、今までのようにコミットされたデータのみが返されますが、読み取り側は書き込み側が行ロックを解除するまで待機

しなくなりました。代わりに読み取り側は、現在コミット済みのバージョンに基づくデータ、つまり書き込み操作の開始前のデータを返します。

デフォルトでは、currently committed セマンティクスは新規データベースでオンになっています。これにより、すべてのアプリケーションが新しい動作を活用でき、アプリケーション自体に変更を加える必要はありません。新しいデータベース構成パラメーター **cur_commit** を使用すると、この動作をオーバーライドできます。これは、内部ロジックを同期するために書き込み側でブロッキングが必要なアプリケーションなどで、役立つ可能性があります。

同様に、内部ロジックを同期するために書き込み側でブロッキングが必要なアプリケーションで、デフォルトで **cur_commit** が使用不可にされているアップグレード・データベースでは、必要に応じてこのパラメーターを後ほどオンにできます。

currently committed セマンティクスは、カタログ表が関係しない読み取り専用スキャン、または制約の評価または強制に使用される内部スキャンにのみ適用されます。 currently committed はスキャン・レベルによって決定されるので、書き込み側のアクセス・プランには currently committed スキャンが組み込まれている場合があることに注意してください。例えば、読み取り専用副照会のスキャンには、currently committed セマンティクスが関係する可能性があります。 currently committed セマンティクスは分離レベル・セマンティクスに従うので、currently committed セマンティクスで実行されているアプリケーションは、引き続き分離レベルを考慮に入れます。

currently committed セマンティクスでは、書き込み側のログ・スペースを増やす必要があります。トランザクション中のデータ行の最初の更新をログに記録するために、追加スペースが必要です。行の現在コミット済みイメージを取得するためにこのデータが必要となります。ワークロードによっては、使用される合計ログ・スペースに関して、このことがかなりの影響を与えることもあれば、微々たる影響しかない場合もあります。追加のログ・スペースに関する要件は、**cur_commit** が使用不可の場合には当てはまりません。

制約事項

currently committed セマンティクスには、以下の制約事項が適用されます。

- データの更新または削除操作に使用されるセクション内のターゲット表オブジェクトは、currently committed セマンティクスを使用しません。変更される行は、更新操作の一部である照会述部を満たした後で変化することがないように、ロック保護する必要があります。
- 行に対して非コミット変更を行ったトランザクションは、行の現在コミット済みバージョンを判別するために、現在コミット済みリーダーに該当ログ・レコードに対して強制的にアクセスさせます。ログ・バッファーにもうないログ・レコードでも物理的に読み取ることは可能ですが、currently committed セマンティクスはログ・アーカイブからのログ・ファイルの取得をサポートしていません。これが影響するのは、無限ロギングを使用するように構成されたデータベースのみです。
- 以下のスキャンは、currently committed セマンティクスを使用しません。
  - カタログ表スキャン
  - 参照整合性制約を適用するために使用されるスキャン

– LONG VARCHAR または LONG VARGRAPHIC 列を参照するスキャン
– 範囲クラスター表 (RCT) スキャン
– 空間索引または拡張索引を使用するスキャン

例

currently committed セマンティクスでデッドロックを回避する、以下のシナリオについて考慮します。このシナリオでは、2 つのアプリケーションが 2 つの別々の表を更新しますが、まだコミットしていません。そのとき各アプリケーションは、もう一方のアプリケーションが更新した表を読み取ろうとします (読み取り専用カーソルを使用)。

ステップ	アプリケーション A	アプリケーション B
1	update T1 set col1 = ? where col2 = ?	update T2 set col1 = ? where col2 = ?
2	select col1, col3, col4 from T2 where col2 >= ?	select col1, col5, from T1 where col5 = ? and col2 = ?
3	コミット	コミット

currently committed セマンティクスを使用しない場合、カーソル固定分離レベルで実行されているこうしたアプリケーションではデッドロックが生じ、いずれかのアプリケーションが失敗することがあります。これは、各アプリケーションが、他のアプリケーションによって更新されているデータを読み取る必要がある場合に生じます。

currently committed セマンティクスでは、(いずれかのアプリケーションの) ステップ 2 における照会で、他のアプリケーションが現在更新しているデータを必要とする場合、そのアプリケーションはロックの解放を待機せず、デッドロックは生じません。代わりに、データのこれまでのコミット済みバージョンが探索されて使用されます。

Oracle データ・ディクショナリー互換ビュー

DB2_COMPATIBILITY_VECTOR レジストリー変数の設定が Oracle データ・ディクショナリー互換ビューをサポートするようになっている場合、データベースの作成時にビューが自動的に作成されます。

Oracle データ・ディクショナリー互換ビューのサポートはデータベース・レベルであり、サポートが必要なデータベースを作成する前に有効にする必要があります。データ・ディクショナリーの定義には、Oracle のデータ・ディクショナリーと互換性がある各ビューについての、CREATE VIEW、CREATE PUBLIC SYNONYM、および COMMENT の各ステートメントが含まれています。SYSIBMADM スキーマに作成されるこれらのビューを、表 39 にリストします。

表 39. Oracle データ・ディクショナリー互換ビュー

カテゴリー	定義されているビュー
一般	DICTIONARY、DICT_COLUMNS USER_CATALOG、DBA_CATALOG、ALL_CATALOG USER_DEPENDENCIES、DBA_DEPENDENCIES、ALL_DEPENDENCIES USER_OBJECTS、DBA_OBJECTS、ALL_OBJECTS USER_SEQUENCES、DBA_SEQUENCES、ALL_SEQUENCES USER_TABLESPACES、DBA_TABLESPACES

表 39. Oracle データ・ディクショナリー互換ビュー (続き)

カテゴリー	定義されているビュー
表またはビュー	USER_CONSTRAINTS、DBA_CONSTRAINTS、ALL_CONSTRAINTS USER_CONS_COLUMNS、DBA_CONS_COLUMNS、ALL_CONS_COLUMNS USER_INDEXES、DBA_INDEXES、ALL_INDEXES USER_IND_COLUMNS、DBA_IND_COLUMNS、ALL_IND_COLUMNS USER_TAB_PARTITIONS、DBA_TAB_PARTITIONS、ALL_TAB_PARTITIONS USER_PART_TABLES、DBA_PART_TABLES、ALL_PART_TABLES USER_PART_KEY_COLUMNS、DBA_PART_KEY_COLUMNS、ALL_PART_KEY_COLUMNS USER_SYNONYMS、DBA_SYNONYMS、ALL_SYNONYMS USER_TABLES、DBA_TABLES、ALL_TABLES USER_TAB_COMMENTS、DBA_TAB_COMMENTS、ALL_TAB_COMMENTS USER_TAB_COLUMNS、DBA_TAB_COLUMNS、ALL_TAB_COLUMNS USER_COL_COMMENTS、DBA_COL_COMMENTS、ALL_COL_COMMENTS USER_TAB_COL_STATISTICS、DBA_TAB_COL_STATISTICS、ALL_TAB_COL_STATISTICS USER_VIEWS、DBA_VIEWS、ALL_VIEWS USER_VIEW_COLUMNS、DBA_VIEW_COLUMNS、ALL_VIEW_COLUMNS
プログラミング・オブジェクト	USER_PROCEDURES、DBA_PROCEDURES、ALL_PROCEDURES USER_SOURCE、DBA_SOURCE、ALL_SOURCE USER_TRIGGERS、DBA_TRIGGERS、ALL_TRIGGERS USER_ERRORS、DBA_ERRORS、ALL_ERRORS USER_ARGUMENTS、DBA_ARGUMENTS、ALL_ARGUMENTS
セキュリティー	USER_ROLE_PRIVS、DBA_ROLE_PRIVS、ROLE_ROLE_PRIVS SESSION_ROLES USER_SYS_PRIVS、DBA_SYS_PRIVS、ROLE_SYS_PRIVS SESSION_PRIVS USER_TAB_PRIVS、DBA_TAB_PRIVS、ALL_TAB_PRIVS、ROLE_TAB_PRIVS USER_TAB_PRIVS_MADE、ALL_TAB_PRIVS_MADE USER_TAB_PRIVS_RECD、ALL_TAB_PRIVS_RECD DBA_ROLES

例

- MYDB というデータベース用のデータ・ディクショナリー互換ビューの作成を有効にします。

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 create db mydb
```

- どのデータ・ディクショナリー互換ビューが使用可能かを判別します。

```
connect to mydb
select * from dictionary
```

- いくつかのデータ・ディクショナリー互換ビューによって、特権に関する情報が提供されます。現在のユーザーに付与されているすべてのシステム特権を表示するには、USER_SYS_PRIVS ビューを使用します。

```
connect to mydb
select * from user_sys_privs
```

- DBA_TABLES ビューで列定義を確認します。

```
connect to mydb
describe select * from dba_tables
```

DB2 の用語と Oracle の用語のマッピング

DB2 環境が正しくセットアップされていれば、Oracle アプリケーションを DB2 データ・サーバーで使用することができます。このため、Oracle の特定の概念と DB2 の概念のマッピングを把握しておくことが重要です。

420 ページの表 40 は、一般的に使用される Oracle の用語とそれに相当する DB2 の用語に関する簡潔なサマリーです。

表 40. DB2 の概念に対応する一般的な Oracle の概念のマッピング

Oracle の概念	DB2 の概念	注
アクティブ・ログ	アクティブ・ログ	これは同じ概念です。
実パラメーター	引数	これは同じ概念です。
アラート・ログ	db2diag ログ・ファイルおよび管理通知ログ	db2diag ログ・ファイルは、主に IBM ソフトウェア・サポートがトラブルシューティングの目的で使用します。管理通知ログは、データベース管理者およびシステム管理者がトラブルシューティングの目的で使用します。管理通知ログ・メッセージも、標準化されたメッセージ・フォーマットを使用して db2diag ログ・ファイルに記録されます。
アーカイブ・ログ	オフライン・アーカイブ・ログ	これは同じ概念です。
アーカイブ・ログ・モード	ログ・アーカイブ	これは同じ概念です。
background_dump_dest	diagpath	これは同じ概念です。
作成されたグローバル一時表	作成されたグローバル一時表	これは同じ概念です。
カーソル共有	ステートメント・コンセントレーター	これは同じ概念です。
データ・ブロック	データ・ページ	これは同じ概念です。
データ・バッファー・キャッシュ	バッファー・プール	これは同じ概念です。ただし DB2 では、任意のページ・サイズのバッファー・プールをいくつでも設定することができます。
データ・ディクショナリー	システム・カタログ	DB2 システム・カタログには、メタデータが表やビューの形式で含まれています。データベース・マネージャーは、基本システム・カタログ表で定義されるシステム・カタログ・ビューを 2 セット作成し、維持します。 • 読み取り専用ビューである SYSCAT ビュー • 最適化プログラムが使用する統計情報を含む更新可能なビューである、SYSSTAT ビュー
データ・ディクショナリー・キャッシュ	カタログ・キャッシュ	これは同じ概念です。

表 40. DB2 の概念に対応する一般的な Oracle の概念のマッピング (続き)

Oracle の概念	DB2 の概念	注
データ・ファイル	コンテナー	DB2 データは、オブジェクトを含むコンテナーに物理的に保管されます。
データベース・リンク	ニックネーム	ニックネームは、リモート・データ・ソースにあるオブジェクト (フェデレーテッド・データベース・オブジェクト) を参照する ID です。
DUAL 表	DUAL 表	これは同じ概念です。
動的なパフォーマンス・ビュー	スナップショット・モニター SQL 管理ビュー	SYSIBMADM スキーマを使用するスナップショット・モニター SQL 管理ビューは、データベース・システムの特定の領域に関するモニター・データを戻します。例えば、SYSIBMADM.SNAPBP SQL 管理ビューは、バッファー・プール情報のスナップショットを提供します。
エクステント	エクステント	DB2 エクステントは、連続するデータ・ページで構成されています。
仮パラメーター	パラメーター	これは同じ概念です。
グローバル索引	非パーティション索引	これは同じ概念です。
非アクティブ・ログ	オンライン・アーカイブ・ログ	これは同じ概念です。
init.ora および Server Parameter File (SPFILE)	データベース・マネージャー構成ファイルおよびデータベース構成ファイル	DB2 インスタンスには、複数のデータベースが含まれる場合があります。したがって、構成パラメーターとそれらの値は、インスタンス・レベル (データベース構成ファイルに保管) とデータベース・レベル (データベース・マネージャー構成ファイルに保管) の両方で保管されます。これらのファイルは、GET コマンドまたは UPDATE DBM CFG コマンド、および GET コマンドまたは UPDATE DB CFG コマンドによってそれぞれ管理されます。

表 40. DB2 の概念に対応する一般的な Oracle の概念のマッピング (続き)

Oracle の概念	DB2 の概念	注
インスタンス	インスタンスまたはデータベース・マネージャー	インスタンスは、バックグラウンド・プロセスと共有メモリーを組み合わせたものです。DB2 インスタンスは、データベース・マネージャーとも呼ばれます。DB2 インスタンスには複数のデータベースが含まれる場合があるため、インスタンス・レベル (データベース・マネージャー構成ファイル) とデータベース・レベル (データベース構成ファイル) の両方に DB2 構成ファイルがあります。
ラージ・プール	ユーティリティー・ヒープ	ユーティリティー・ヒープは、バックアップ、リストア、およびロードの各ユーティリティーによって使用されます。
ライブラリー・キャッシュ	パッケージ・キャッシュ	データベース共有メモリーから割り振られるパッケージ・キャッシュは、データベース上で静的および動的な SQL ステートメントと XQuery ステートメント用のセクションのキャッシュに使用されます。
ローカル索引	パーティション索引	これは同じ概念です。
マテリアライズ・ビュー	マテリアライズ照会表 (MQT)	MQT は照会結果に基づいて定義される表で、パフォーマンス向上のために使用されます。 DB2 SQL コンパイラーは、ある照会を MQT に対して実行した場合のほうが、MQT の基本表に対して実行した場合よりも効率がよいかどうかを判別します。
非アーカイブ・ログ・モード	循環ロギング	これは同じ概念です。

表 40. *DB2* の概念に対応する一般的な *Oracle* の概念のマッピング (続き)

Oracle の概念	**DB2 の概念**	**注**
Oracle Call Interface (OCI)	DB2CI インターフェース	DB2CI は、関数呼び出しを使用して、DB2 バージョン 9.7 データベースに接続し、カーソルを管理し、また SQL ステートメントを実行する 'C' および 'C++' アプリケーション・プログラミング・インターフェースです。DB2CI ドライバーによってサポートされる OCI API のリストについては、 427 ページの『IBM Data Server Driver for DB2CI』を参照してください。
Oracle Call Interface (OCI)	コール・レベル・インターフェース (CLI)	CLI は、C および C++ アプリケーション・プログラミング・インターフェースで、関数呼び出しを使用して、動的 SQL ステートメントを関数引数として渡します。ほとんどの場合、OCI 関数は、CLI 関数と、サポートされているプログラム・コードに加えた変更内容に置き換えることができます。
ORACLE_SID 環境変数	DB2INSTANCE 環境変数	これは同じ概念です。
パーティション表	パーティション表	これは同じ概念です。
Procedural Language/Structured Query Language (PL/SQL)	SQL プロシージャー型言語 (SQL PL)	SQL PL は SQL の拡張言語で、ステートメントと言語要素で構成されています。SQL PL は、変数と条件処理ルーチンを宣言し、変数に値を割り当て、手続き型論理を設定するステートメントを提供します。SQL PL は SQL Persistent Stored Modules (SQL/PSM) 言語規格のサブセットです。 Oracle PL/SQL ステートメントは、DB2 インターフェースを使用してコンパイルし、実行することができます。

表 40. DB2 の概念に対応する一般的な Oracle の概念のマッピング (続き)

Oracle の概念	**DB2 の概念**	**注**
Program Global Area (PGA)	アプリケーション共有メモリーおよびエージェント専用メモリー	アプリケーション共有メモリーには、主にデータベースとの間でやりとりされるデータ行など、データベースと特定のアプリケーション間で共有される情報が保管されます。エージェント専用メモリーには、ソート・ヒープ、カーソル情報、セッション・コンテキストなど、特定のアプリケーションで使用される情報が保管されます。
REDO ログ	トランザクション・ログ	トランザクション・ログにはデータベース・トランザクションが記録され、リカバリーに使用することができます。
ロール	ロール	これは同じ概念です。
セグメント	ストレージ・オブジェクト	これは同じ概念です。
セッション	セッション、データベース接続	これは同じ概念です。
startup nomount	db2start	インスタンスを開始するコマンドです。
シノニム	別名	別名とは、表、ビュー、ニックネーム、またはその他の別名の代替名です。「シノニム」という用語が許可されており、「別名」の代わりに指定することができます。別名は、アプリケーションで使用される DB2 プロシージャーまたはユーザー定義関数のバージョンを管理するものではありません。バージョンを管理するには、SET PATH ステートメントを使用して、必要なスキーマを CURRENT PATH 特殊レジスターの値に追加します。

表 40. DB2 の概念に対応する一般的な Oracle の概念のマッピング (続き)

Oracle の概念	DB2 の概念	注
System Global Area (SGA)	インスタンス共有メモリーおよびデータベース共有メモリー	インスタンス共有メモリーには、すべてのアクティブな接続およびセキュリティー情報のリストなど、特定のインスタンスに関するすべての情報が保管されます。データベース共有メモリーには、パッケージ・キャッシュ、ログ・バッファー、バッファー・プールなど、データベース固有の情報が保管されます。
SYSTEM 表スペース	SYSCATSPACE 表スペース	SYSCATSPACE 表スペースにはシステム・カタログが含まれています。この表スペースは、データベースの作成時にデフォルトで作成されます。
表スペース	表スペース	これは同じ概念です。
User Global Area (UGA)	アプリケーション・グローバル・メモリー	アプリケーション・グローバル・メモリーは、アプリケーション共有メモリーとアプリケーション専用メモリーで構成されています。

第 5 章 DB2CI アプリケーション開発

DB2CI は、DB2 バージョン 9.7 データベース・サーバーに対する呼び出し可能 SQL インターフェースです。これは、関数呼び出しを使用して、データベースに接続し、カーソルを管理し、また SQL ステートメントを実行する、DB2 データベース・アクセス用の 'C' および 'C++' アプリケーション・プログラミング・インターフェースです。

バージョン 9.7 フィックスパック 1 以降では、DB2CI インターフェースを使用して、サポートされる任意のオペレーティング・システム上の DB2 バージョン 9.7 サーバーのデータベースにアクセスできます。

DB2CI インターフェースは、多数の Oracle Call Interface (OCI) API に対するサポートを提供します。このサポートにより、既存の OCI アプリケーションが DB2 データベースを使用した処理を実行できるように準備するという煩雑な作業が軽減されます。 IBM Data Server Driver for DB2CIは、DB2CI インターフェースのドライバーです。

IBM Data Server Driver for DB2CI

IBM Data Server Driver for DB2CIは、DB2CI アプリケーション開発のサポートを提供します。

IBM Data Server Client には DB2CI ドライバー が含まれます。 DB2CI ドライバーをインストールするには、このクライアントをインストールする必要があります。

DB2CI ドライバーは、以下の OCI API 呼び出しのサポートを提供します。

OCIAttrGet	OCILobGetLength	OCINumberTan
OCIAttrSet	OCILobIsEqual	OCINumberToInt
OCIBindArrayOfStruct	OCILobIsTemporary	OCINumberToReal
OCIBindByName	OCILobIsOpen	OCINumberToRealArray
OCIBindByPos	OCILobLocatorAssign	OCINumberToText
OCIBindDynamic	OCILobLocatorIsInit	OCINumberTrunc
OCIBreak	OCILobRead	OCIParamGet
OCIClientVersion	OCILobTrim	OCIParamSet
OCIDateAddDays	OCILobWrite	OCIPasswordChange
OCIDateAddMonths	OCILogoff	OCIPing
OCIDateAssign	OCILogon	OCIRawAllocSize
OCIDateCheck	OCILogon2	OCIRawAssignBytes
OCIDateCompare	OCINumberAbs	OCIRawAssignRaw
OCIDateDaysBetween	OCINumberAdd	OCIRawPtr
OCIDateFromText	OCINumberArcCos	OCIRawResize
OCIDateLastDay	OCINumberArcSin	OCIRawSize

OCIDateNextDay
OCIDateSysDate
OCIDateToText
OCIDefineArrayOfStruct
OCIDefineByPos
OCIDefineDynamic
OCIDescribeAny
OCIDescriptorAlloc
OCIDescriptorFree
OCIEnvCreate
OCIEnvInit
OCIErrorGet
OCIFileClose
OCIFileExists
OCIFileFlush
OCIFileGetLength
OCIFileInit
OCIFileOpen
OCIFileRead
OCIFileSeek
OCIFileTerm
OCIFileWrite
OCIHandleAlloc
OCIHandleFree
OCIInitialize
OCILobAppend
OCILobAssign
OCILobClose
OCILobCopy
OCILobCreateTemporary
OCILobDisableBuffering
OCILobEnableBuffering
OCILobErase
OCILobFreeTemporary
OCILobFlushBuffer
OCINumberArcTan
OCINumberArcTan2
OCINumberAssign
OCINumberCeil
OCINumberCmp
OCINumberCos
OCINumberDec
OCINumberDiv
OCINumberExp
OCINumberFloor
OCINumberFromInt
OCINumberFromReal
OCINumberFromText
OCINumberHypCos
OCINumberHypSin
OCINumberHypTan
OCINumberInc
OCINumberIntPower
OCINumberIsInt
OCINumberIsZero
OCINumberLn
OCINumberLog
OCINumberMod
OCINumberMul
OCINumberNeg
OCINumberPower
OCINumberPrec
OCINumberRound
OCINumberSetPi
OCINumberSetZero
OCINumberShift
OCINumberSign
OCINumberSin
OCINumberSqrt
OCINumberSub
OCIReset
OCIResultSetToStmt
OCIServerAttach
OCIServerDetach
OCIServerVersion
OCISessionBegin
OCISessionEnd
OCISessionGet
OCISessionRelease
OCIStmtExecute
OCIStmtFetch
OCIStmtFetch2
OCIStmtGetBindInfo
OCIStmtGetPieceInfo
OCIStmtPrepare
OCIStmtPrepare2
OCIStmtRelease
OCIStmtSetPieceInfo
OCIStringAllocSize
OCIStringAssign
OCIStringAssignText
OCIStringPtr
OCIStringResize
OCIStringSize
OCITerminate
OCITransCommit
OCITransDetach
OCITransForget
OCITransMultiPrepare
OCITransPrepare
OCITransRollback
OCITransStart
xaoEnv
xaosterr
xaoSvcCtx

DB2CI アプリケーションの構築

DB2CI アプリケーションは、既存の Oracle Call Interface (OCI) アプリケーションおよび bldapp スクリプト・ファイルを使用して構築できます。

- 既存の OCI アプリケーションにより使用される Oracle データベースと同じ構造を持つ DB2 データベースが必要です。

- IBM Data Server Client をインストールしている必要があります。

DB2 サンプルは、IBM Data Server Driver for DB2CI によってサポートされる OCI 関数を使用するアプリケーションをコンパイルおよびリンクするための bldapp というスクリプトを提供します。これは *DB2DIR*¥samples¥db2ci または *DB2DIR*/samples/db2ci ディレクトリーに、サンプル・プログラムと共に置かれています。 *DB2DIR* は DB2 コピーがインストールされているロケーションを表します。

bldapp スクリプト・ファイルは、最大 4 つのパラメーターを使用します。第 1 パラメーター $1 には、ソース・ファイルの名前を指定します。追加パラメーターが必要なのは、データベースへの接続を必要とする組み込み SQL プログラムを構築する場合に限られます。2 番目のパラメーターは $2 で、接続するデータベースの名前を指定します。 3 番目のパラメーターは $3 で、データベースのユーザー ID を指定します。そしてもう 1 つが $4 で、パスワードを指定します。プログラムに組み込み SQL が含まれる場合 (.sqc 拡張子によって識別される)、embprep スクリプトが呼び出されて、プログラムをプリコンパイルし、拡張子 .c を持つプログラム・ファイルを作成します。

制約事項

- 既存の OCI アプリケーションが、DB2CI ドライバー によってサポートされる OCI 関数の呼び出しのみを使用することを確認します。サポートされる OCI 関数の完全なリストについては、 427 ページの『IBM Data Server Driver for DB2CI』 を参照してください。

1. DB2CI アプリケーションを既存の OCI アプリケーションを使用して構築する場合には必ず、db2ci.h インクルード・ファイルを指定します。
2. DB2CI アプリケーションを bldapp スクリプト・ファイルを使用して構築します。 以下の例は、Linux および UNIX オペレーティング・システム上で、ソース・ファイル tbinfo.c からサンプル・プログラム tbinfo を作成する方法を示しています。

   ```
   cd $INSTHOME/sqllib/samples/db2ci
   bldapp tbinfo
   ```

   結果として、実行可能ファイル tbinfo が作成されます。
3. 以下のように実行可能ファイル名を入力して、前のステップで生成された実行可能ファイルを実行します。

   ```
   tbinfo
   ```

DB2CI アプリケーションのコンパイルとリンクのオプション (AIX)

このトピックのコンパイルとリンクのオプションは、DB2CI アプリケーションを AIX® IBM C コンパイラーを使用して構築する場合に推奨されます。

以下のオプションは *DB2DIR*/samples/cli/bldapp バッチ・ファイルにあります (*DB2DIR* は DB2 コピーがインストールされているロケーション)。

bldapp のコンパイルとリンクのオプション

コンパイル・オプション:

xlc IBM C コンパイラー。

$EXTRA_CFLAG
64 ビット環境の場合は、値 "-q64" が入ります。それ以外の場合は、値が入りません。

-I$DB2PATH/include
DB2 組み込みファイルのロケーションを指定します。例えば、$HOME/sqllib/include のように指定します。

-c コンパイルのみを実行し、リンクは実行しません。このスクリプトでは、コンパイルとリンクは別個のステップです。

リンク・オプション:

xlc コンパイラーをリンカーのフロントエンドとして使用します。

$EXTRA_CFLAG
64 ビット環境の場合は、値 "-q64" が入ります。それ以外の場合は、値が入りません。

-o $1 実行可能プログラムを指定します。

$1.o オブジェクト・ファイルを指定します。

utilci.o
エラー・チェック用のユーティリティー・オブジェクト・ファイルを組み込みます。

-L$DB2PATH/$LIB
DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、$HOME/sqllib/$LIB。 -L オプションを指定しないと、コンパイラーは次のパスを想定します。 /usr/lib:/lib。

-ldb2ci
DB2CI ライブラリーとリンクします。

DB2CI アプリケーションのコンパイルとリンクのオプション (HP-UX)

このトピックのコンパイルとリンクのオプションは、DB2CI アプリケーションを HP-UX C コンパイラーを使用して構築する場合に推奨されます。

以下のオプションは *DB2DIR*/samples/db2ci/bldapp バッチ・ファイルにあります (*DB2DIR* は DB2 コピーがインストールされているロケーション)。

bldapp のコンパイルとリンクのオプション

コンパイル・オプション:

cc　C コンパイラーを使用します。

$EXTRA_CFLAG

HP-UX プラットフォームが IA64 で、64 ビット・サポートが使用可能な場合は、このフラグには値 **+DD64** が入り、 32 ビット・サポートが使用可能な場合は、値 **+DD32** が入ります。 HP-UX プラットフォームが PA-RISC で、64 ビット・サポートが使用可能な場合は、これには値 **+DA2.0W** が入ります。 PA-RISC プラットフォームでの 32 ビット・サポートの場合は、このフラグには値 **+DA2.0N** が入ります。

+DD64　IA64 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DD32　IA64 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+DA2.0W

PA-RISC 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DA2.0N

PA-RISC 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

-Ae　HP ANSI 拡張モードを使用可能にします。

-I$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。例えば、`$HOME/sqllib/include` のように指定します。

-c　コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

cc　コンパイラーをリンカーのフロントエンドとして使用します。

$EXTRA_CFLAG
HP-UX プラットフォームが IA64 で、64 ビット・サポートが使用可能な場合は、このフラグには値 **+DD64** が入り、 32 ビット・サポートが使用可能な場合は、値 **+DD32** が入ります。 HP-UX プラットフォームが PA-RISC で、64 ビット・サポートが使用可能な場合は、これには値 **+DA2.0W** が入ります。 PA-RISC プラットフォームでの 32 ビット・サポートの場合は、このフラグには値 **+DA2.0N** が入ります。

+DD64　IA64 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DD32　IA64 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+DA2.0W
PA-RISC 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DA2.0N
PA-RISC 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

-o $1　実行可能プログラムを指定します。

$1.o　オブジェクト・ファイルを指定します。

utilci.o
エラー・チェック用のユーティリティー・オブジェクト・ファイルを組み込みます。

$EXTRA_LFLAG
ランタイム・パスを指定します。設定する場合、32 ビットの場合は値 `-Wl,+b$HOME/sqllib/lib32`、 64 ビットの場合は `-Wl,+b$HOME/sqllib/lib64` が入ります。設定しない場合は、これには値が入りません。

-L$DB2PATH/$LIB
DB2 ランタイム共用ライブラリーのロケーションを指定します。 32 ビットの場合は `$HOME/sqllib/lib32`、 64 ビットの場合は `$HOME/sqllib/lib64` です。

-ldb2ci
DB2CI ライブラリーとリンクします。

DB2CI アプリケーションのコンパイルとリンクのオプション (Linux)

このトピックのコンパイルとリンクのオプションは、DB2CI アプリケーションを GNU/Linux gcc コンパイラーを使用して構築する場合に推奨されます。

以下のオプションは *DB2DIR*`/samples/db2ci/bldapp` バッチ・ファイルにあります (*DB2DIR* は DB2 コピーがインストールされているロケーション)。

bldapp のコンパイルとリンクのオプション

コンパイル・オプション:

gcc　C コンパイラー。

$EXTRA_C_FLAGS
次のうちの 1 つが含まれています。

- Linux for zSeries® の場合のみ -m31 を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER® の場合には -m32 を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には -m64 を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

-I$DB2PATH/include
DB2 組み込みファイルのロケーションを指定します。例えば、$HOME/sqllib/include のように指定します。

-c　コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

gcc　コンパイラーをリンカーのフロントエンドとして使用します。

$EXTRA_C_FLAGS
次のうちの 1 つが含まれています。

- Linux for zSeries の場合のみ -m31 を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER の場合には -m32 を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には -m64 を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

-o $1　実行可能ファイルを指定します。

$1.o　プログラム・オブジェクト・ファイルを組み込みます。

utilci.o
エラー・チェック用のユーティリティー・オブジェクト・ファイルを組み込みます。

$EXTRA_LFLAG
32 ビットの場合は値 "-Wl,-rpath,$DB2PATH/lib32" が入り、 64 ビットの場合は値 "-Wl,-rpath,$DB2PATH/lib64" が入ります。

-L$DB2PATH/$LIB
リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は $HOME/sqllib/lib32、64 ビットの場合は $HOME/sqllib/lib64 です。

-ldb2ci
DB2CI ライブラリーとリンクします。

DB2CI アプリケーションのコンパイルとリンクのオプション (Solaris)

このトピックのコンパイルとリンクのオプションは、DB2CI アプリケーションを Solaris C コンパイラーを使用して構築する場合に推奨されます。

以下のオプションは *DB2DIR*/samples/db2ci/bldapp バッチ・ファイルにあります (*DB2DIR* は DB2 コピーがインストールされているロケーション)。

bldapp のコンパイルとリンクのオプション
コンパイル・オプション: **cc** C コンパイラーを使用します。 **-xarch=$CFLAG_ARCH** このオプションを使用すると、 libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。 $CFLAG_ARCH の値は、次のように設定されます。 • Solaris SPARC の 32 ビット・アプリケーションの場合は「v8plusa」 • Solaris SPARC の 64 ビット・アプリケーションの場合は「v9」 • Solaris x64 の 32 ビット・アプリケーションの場合は「sse2」 • Solaris x64 の 64 ビット・アプリケーションの場合は「amd64」 **-I$DB2PATH/include** DB2 組み込みファイルのロケーションを指定します。例えば、 $HOME/sqllib/include のように指定します。 **-c** コンパイルのみを実行し、リンクは実行しません。このスクリプトでは、コンパイルとリンクは別個のステップです。

リンク・オプション:

cc
コンパイラーをリンカーのフロントエンドとして使用します。

-xarch=$CFLAG_ARCH
このオプションを使用すると、 libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。 $CFLAG_ARCH の値は、32 ビットの場合は v8plusa に、64 ビットの場合は v9 に設定されます。

-mt
マルチスレッド・サポートにリンクし、fopen の呼び出し時に問題が起きないようにします。
注: POSIX スレッドを使用する場合、DB2 アプリケーションは、スレッド化されていてもいなくても -lpthread にリンクする必要もあります。

-o $1
実行可能プログラムを指定します。

$1.o
プログラム・オブジェクト・ファイルを組み込みます。

utilci.o
エラー・チェック用のユーティリティー・オブジェクト・ファイルを組み込みます。

-L$DB2PATH/$LIB
リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は $HOME/sqllib/lib32、64 ビットの場合は $HOME/sqllib/lib64 です。

$EXTRA_LFLAG
実行時の DB2 共用ライブラリーのロケーションを示します。 32 ビットの場合は値 "-R$DB2PATH/lib32" が入り、 64 ビットの場合は値 "-R$DB2PATH/lib64" が入ります。

-ldb2ci
DB2CI ライブラリーとリンクします。

DB2CI アプリケーションのコンパイルとリンクのオプション (Windows)

このトピックのコンパイルとリンクのオプションは、DB2CI アプリケーションを Microsoft® Visual C++ コンパイラーを使用して構築する場合に推奨されます。

以下のオプションは *DB2DIR*¥samples¥db2ci¥bldapp.bat バッチ・ファイルにあります (*DB2DIR* は DB2 コピーがインストールされているロケーション)。

bldapp のコンパイルとリンクのオプション

コンパイル・オプション:

%BLDCOMP%
コンパイラー用の変数です。デフォルトは、cl (Microsoft Visual C++ コンパイラー) です。またこれは、icl (32 ビットおよび 64 ビット・アプリケーション用の Intel® C++ コンパイラー)、または ecl (Itanium® 64 ビット・アプリケーション用の Intel C++ コンパイラー) に設定することもできます。

-Zi デバッグ情報を使用可能にします。

-Od 最適化なし。最適化をオフにしてデバッガーを使用する方が簡単です。

-c コンパイルのみを実行し、リンクは実行しません。

-W2 警告レベルを設定します。

-DWIN32
Windows オペレーティング・システムに必要なコンパイラー・オプション。

リンク・オプション:

link リンカーを使用します。

-debug デバッグ情報を組み込みます。

-out:%1.exe
実行可能ファイルを指定します。

%1.obj オブジェクト・ファイルを組み込みます。

db2ci.lib または db2ci64.lib
DB2CI ライブラリーとリンクします。 Windows 32 ビット・オペレーティング・システムの場合は、db2ci.lib を使用します。 Windows 64 ビット・オペレーティング・システムの場合は、db2ci64.lib を使用します。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

付録 A. DB2 技術情報の概説

DB2 技術情報は、以下のツールと方法を介して利用できます。

- DB2 インフォメーション・センター
  - トピック (タスク、概念、およびリファレンス・トピック)
  - DB2 ツールのヘルプ
  - サンプル・プログラム
  - チュートリアル
- DB2 資料
  - PDF ファイル (ダウンロード可能)
  - PDF ファイル (DB2 PDF DVD に含まれる)
  - 印刷資料
- コマンド行ヘルプ
  - コマンド・ヘルプ
  - メッセージ・ヘルプ

注: DB2 インフォメーション・センターのトピックは、PDF やハードコピー資料よりも頻繁に更新されます。最新の情報を入手するには、資料の更新が発行されたときにそれをインストールするか、ibm.com にある DB2 インフォメーション・センターを参照してください。

技術資料、ホワイト・ペーパー、IBM Redbooks® 資料などのその他の DB2 技術情報には、オンライン (ibm.com) でアクセスできます。 DB2 Information Management ソフトウェア・ライブラリー・サイト (http://www.ibm.com/software/data/sw-library/) にアクセスしてください。

資料についてのフィードバック

DB2 の資料についてのお客様からの貴重なご意見をお待ちしています。DB2 の資料を改善するための提案については、db2docs@ca.ibm.com まで E メールを送信してください。DB2 の資料チームは、お客様からのフィードバックすべてに目を通しますが、直接お客様に返答することはありません。お客様が関心をお持ちの内容について、可能な限り具体的な例を提供してください。特定のトピックまたはヘルプ・ファイルについてのフィードバックを提供する場合は、そのトピック・タイトルおよび URL を含めてください。

DB2 お客様サポートに連絡する場合には、この E メール・アドレスを使用しないでください。資料を参照しても、DB2 の技術的な問題が解決しない場合は、お近くの IBM サービス・センターにお問い合わせください。

DB2 テクニカル・ライブラリー (ハードコピーまたは PDF 形式)

以下の表は、IBM Publications Center (www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss) から利用できる DB2 ライブラリーについて説明しています。英語および翻訳された DB2 バージョン 9.7 のマニュアル (PDF 形式) は、www.ibm.com/support/docview.wss?rs=71&uid=swg2700947 からダウンロードできます。

この表には印刷資料が入手可能かどうかが示されていますが、国または地域によっては入手できない場合があります。

資料番号は、資料が更新される度に大きくなります。資料を参照する際は、以下にリストされている最新版であることを確認してください。

注: *DB2 インフォメーション・センター*は、PDF やハードコピー資料よりも頻繁に更新されます。

表 41. DB2 の技術情報

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
管理 API リファレンス	SC88-5883-02	入手可能	2010 年 9 月
管理ルーチンおよびビュー	SC88-5880-02	入手不可	2010 年 9 月
コール・レベル・インターフェース ガイドおよびリファレンス 第 1 巻	SC88-5885-02	入手可能	2010 年 9 月
コール・レベル・インターフェース ガイドおよびリファレンス 第 2 巻	SC88-5886-02	入手可能	2010 年 9 月
コマンド・リファレンス	SC88-5884-02	入手可能	2010 年 9 月
データ移動ユーティリティー ガイドおよびリファレンス	SC88-5903-00	入手可能	2009 年 8 月
データ・リカバリーと高可用性 ガイドおよびリファレンス	SC88-5904-02	入手可能	2010 年 9 月
データベース: 管理の概念および構成リファレンス	SC88-5870-02	入手可能	2010 年 9 月
データベースのモニタリング ガイドおよびリファレンス	SC88-5872-02	入手可能	2010 年 9 月
データベース・セキュリティー・ガイド	SC88-5905-01	入手可能	2009 年 11 月

表 41. DB2 の技術情報 (続き)

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
DB2 Text Search ガイド	SC88-5902-02	入手可能	2010 年 9 月
ADO.NET および OLE DB アプリケーションの開発	SC88-5874-01	入手可能	2009 年 11 月
組み込み SQL アプリケーションの開発	SC88-5875-01	入手可能	2009 年 11 月
Java アプリケーションの開発	SC88-5878-02	入手可能	2010 年 9 月
Perl、PHP、Python および Ruby on Rails アプリケーションの開発	SC88-5879-01	入手不可	2010 年 9 月
SQL および外部ルーチンの開発	SC88-5876-01	入手可能	2009 年 11 月
データベース・アプリケーション開発の基礎	GI88-4201-01	入手可能	2009 年 11 月
DB2 インストールおよび管理 概説 (Linux および Windows 版)	GI88-4202-00	入手可能	2009 年 8 月
グローバリゼーション・ガイド	SC88-5906-00	入手可能	2009 年 8 月
DB2 サーバー機能 インストール	GC88-5888-02	入手可能	2010 年 9 月
IBM データ・サーバー・クライアント機能 インストール	GC88-5889-01	入手不可	2010 年 9 月
メッセージ・リファレンス 第 1 巻	SC88-5897-00	入手不可	2009 年 8 月
メッセージ・リファレンス 第 2 巻	SC88-5898-00	入手不可	2009 年 8 月
Net Search Extender 管理およびユーザーズ・ガイド	SC88-5901-02	入手不可	2010 年 9 月
パーティションおよびクラスタリングのガイド	SC88-5907-01	入手可能	2009 年 11 月
pureXML ガイド	SC88-5895-01	入手可能	2009 年 11 月
Query Patroller 管理およびユーザーズ・ガイド	SC88-5908-00	入手不可	2009 年 8 月

表 41. DB2 の技術情報 (続き)

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
Spatial Extender および Geodetic Data Management Feature ユーザーズ・ガイドおよびリファレンス	SC88-5900-01	入手不可	2010 年 9 月
SQL プロシージャー言語: アプリケーションのイネーブルメントおよびサポート	SC88-5877-02	入手可能	2010 年 9 月
SQL リファレンス 第 1 巻	SC88-5881-02	入手可能	2010 年 9 月
SQL リファレンス 第 2 巻	SC88-5882-02	入手可能	2010 年 9 月
問題判別およびデータベース・パフォーマンスのチューニング	SC88-5871-02	入手可能	2010 年 9 月
DB2 バージョン 9.7 へのアップグレード	SC88-5887-02	入手可能	2010 年 9 月
Visual Explain チュートリアル	SC88-5899-00	入手不可	2009 年 8 月
DB2 バージョン 9.7 の新機能	SC88-5893-02	入手可能	2010 年 9 月
ワークロード・マネージャー ガイドおよびリファレンス	SC88-5894-02	入手可能	2010 年 9 月
XQuery リファレンス	SC88-5896-01	入手不可	2009 年 11 月

表 42. DB2 Connect 固有の技術情報

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
DB2 Connect Personal Edition インストールおよび構成	SC88-5891-02	入手可能	2010 年 9 月
DB2 Connect サーバー機能 インストールおよび構成	SC88-5892-02	入手可能	2010 年 9 月
DB2 Connect ユーザーズ・ガイド	SC88-5890-02	入手可能	2010 年 9 月

表 43. Information Integration の技術情報

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
Information Integration: フェデレーテッド・システム管理ガイド	SC88-4166-02	入手可能	2009 年 8 月
Information Integration: レプリケーションおよびイベント・パブリッシングのための ASNCLP プログラム・リファレンス	SC88-4167-04	入手可能	2009 年 8 月
Information Integration: フェデレーテッド・データ・ソース構成ガイド	SC88-4185-02	入手不可	2009 年 8 月
Information Integration: SQL レプリケーション ガイドとリファレンス	SC88-4168-02	入手可能	2009 年 8 月
Information Integration: レプリケーションとイベント・パブリッシング 概説	GC88-4187-02	入手可能	2009 年 8 月

DB2 の印刷資料の注文方法

DB2 の印刷資料が必要な場合、オンラインで購入することができますが、すべての国および地域で購入できるわけではありません。 DB2 の印刷資料については、IBM 営業担当員にお問い合わせください。 *DB2 PDF ドキュメンテーション* DVD の一部のソフトコピー・ブックは、印刷資料では入手できないことに留意してください。例えば、「*DB2 メッセージ・リファレンス*」はどちらの巻も印刷資料としては入手できません。

DB2 PDF ドキュメンテーション DVD で利用できる DB2 の印刷資料の大半は、IBM に有償で注文することができます。国または地域によっては、資料を IBM Publications Center からオンラインで注文することもできます。お客様の国または地域でオンライン注文が利用できない場合、DB2 の印刷資料については、IBM 営業担当員にお問い合わせください。 DB2 PDF ドキュメンテーション DVD に収録されている資料の中には、印刷資料として提供されていないものもあります。

注: 最新で完全な DB2 資料は、DB2 インフォメーション・センター (http://publib.boulder.ibm.com/infocenter/db2luw/v9r7) で参照することができます。

DB2 の印刷資料は以下の方法で注文することができます。

- 日本 IBM 発行のマニュアルはインターネット経由でご購入いただけます。詳しくは http://www.ibm.com/shop/publications/order をご覧ください。資料の注文情報にアクセスするには、お客様の国、地域、または言語を選択してください。その後、各ロケーションにおける注文についての指示に従ってください。

- DB2 の印刷資料を IBM 営業担当員に注文するには、以下のようにします。
  1. 以下の Web サイトのいずれかから、営業担当員の連絡先情報を見つけてください。
     - IBM Directory of world wide contacts (www.ibm.com/planetwide)
     - IBM Publications Web サイト (http://www.ibm.com/shop/publications/order)。国、地域、または言語を選択し、お客様の所在地に該当する Publications ホーム・ページにアクセスしてください。このページから、「このサイトについて」のリンクにアクセスしてください。
  2. 電話をご利用の場合は、DB2 資料の注文であることをご指定ください。
  3. 担当者に、注文する資料のタイトルと資料番号をお伝えください。タイトルと資料番号は、438 ページの『DB2 テクニカル・ライブラリー (ハードコピーまたは PDF 形式)』でご確認いただけます。

コマンド行プロセッサーから SQL 状態ヘルプを表示する

DB2 製品は、SQL ステートメントの結果の原因になったと考えられる条件の SQLSTATE 値を戻します。 SQLSTATE ヘルプは、SQL 状態および SQL 状態クラス・コードの意味を説明します。

SQL 状態ヘルプを開始するには、コマンド行プロセッサーを開いて以下のように入力します。

```
? sqlstate or ? class code
```

ここで、*sqlstate* は有効な 5 桁の SQL 状態を、*class code* は SQL 状態の最初の 2 桁を表します。
例えば、`? 08003` を指定すると SQL 状態 08003 のヘルプが表示され、 `? 08` を指定するとクラス・コード 08 のヘルプが表示されます。

異なるバージョンの DB2 インフォメーション・センターへのアクセス

DB2 バージョン 9.8 のトピックを扱っている *DB2* インフォメーション・センターの URL は、http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/ です。

DB2 バージョン 9.7 のトピックを扱っている *DB2* インフォメーション・センターの URL は、http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/ です。

DB2 バージョン 9.5 のトピックを扱っている *DB2* インフォメーション・センターの URL は、http://publib.boulder.ibm.com/infocenter/db2luw/v9r5 です。

DB2 バージョン 9.1 のトピックを扱っている *DB2* インフォメーション・センターの URL は、http://publib.boulder.ibm.com/infocenter/db2luw/v9/ です。

DB2 バージョン 8 のトピックについては、*DB2* インフォメーション・センター の URL http://publib.boulder.ibm.com/infocenter/db2luw/v8/ にアクセスしてください。

DB2 インフォメーション・センターでの希望する言語でのトピックの表示

DB2 インフォメーション・センターでは、ブラウザーの設定で指定した言語でのトピックの表示が試みられます。トピックがその指定言語に翻訳されていない場合は、 DB2 インフォメーション・センターでは英語でトピックが表示されます。

- Internet Explorer Web ブラウザーで、指定どおりの言語でトピックを表示するには、以下のようにします。
  1. Internet Explorer の「**ツール**」 -> 「**インターネット オプション**」 -> 「**言語...**」ボタンをクリックします。「言語の優先順位」ウィンドウがオープンします。
  2. 該当する言語が、言語リストの先頭の項目に指定されていることを確認します。
     - リストに新しい言語を追加するには、「**追加...**」ボタンをクリックします。

       注: 言語を追加しても、特定の言語でトピックを表示するのに必要なフォントがコンピューターに備えられているとはかぎりません。
     - リストの先頭に新しい言語を移動するには、その言語を選択してから、その言語が言語リストに先頭に行くまで「**上に移動**」ボタンをクリックします。
  3. ページを最新表示します。希望する言語で DB2 インフォメーション・センターが表示されます。
- Firefox または Mozilla Web ブラウザーの場合に、希望する言語でトピックを表示するには、以下のようにします。
  1. 「**ツール**」 -> 「**オプション**」 -> 「**詳細**」 ダイアログの「**言語**」セクションにあるボタンを選択します。「設定」ウィンドウに「言語」パネルが表示されます。
  2. 該当する言語が、言語リストの先頭の項目に指定されていることを確認します。
     - リストに新しい言語を追加するには、「**追加...**」ボタンをクリックしてから、「言語を追加」ウィンドウで言語を選択します。
     - リストの先頭に新しい言語を移動するには、その言語を選択してから、その言語が言語リストに先頭に行くまで「**上に移動**」ボタンをクリックします。
  3. ページを最新表示します。希望する言語で DB2 インフォメーション・センターが表示されます。

ブラウザーとオペレーティング・システムの組み合わせによっては、オペレーティング・システムの地域の設定も希望のロケールと言語に変更しなければなりません。

コンピューターまたはイントラネット・サーバーにインストールされた DB2 インフォメーション・センターの更新

ローカルにインストールされた DB2 インフォメーション・センターは、定期的に更新する必要があります。

DB2 バージョン 9.7 インフォメーション・センターが既にインストールされている必要があります。詳しくは、「*DB2 サーバー機能 インストール*」の『DB2 セットアップ・ウィザードによる DB2 インフォメーション・センターのインストール』のトピックを参照してください。インフォメーション・センターのインストールに適用されるすべての前提条件と制約事項は、インフォメーション・センターの更新にも適用されます。

既存の DB2 インフォメーション・センターは、自動で更新することも。手動で更新することもできます。

- 自動更新 - 既存のインフォメーション・センターのフィーチャーと言語を更新します。自動更新を使用すると、更新中にインフォメーション・センターが使用できなくなる時間が最小限で済むというメリットもあります。さらに、自動更新は、定期的に実行する他のバッチ・ジョブの一部として実行されるように設定することができます。
- 手動更新 - 更新処理中にフィーチャーまたは言語を追加する場合に使用する必要があります。例えば、ローカルのインフォメーション・センターが最初は英語とフランス語でインストールされており、その後ドイツ語もインストールすることにした場合、手動更新でドイツ語をインストールし、同時に、既存のインフォメーション・センターのフィーチャーおよび言語を更新できます。しかし、手動更新ではインフォメーション・センターを手動で停止、更新、再始動する必要があります。更新処理の間はずっと、インフォメーション・センターは使用できなくなります。

このトピックでは、自動更新のプロセスを詳しく説明しています。手動更新の手順については、『コンピューターまたはイントラネット・サーバーにインストールされた DB2 インフォメーション・センターの手動更新』のトピックを参照してください。

コンピューターまたはイントラネット・サーバーにインストールされている DB2 インフォメーション・センターを自動で更新するには、次のようにします。

1. Linux オペレーティング・システムの場合、次のようにします。
   a. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、DB2 インフォメーション・センターは、`/opt/ibm/db2ic/V9.7` ディレクトリーにインストールされています。
   b. インストール・ディレクトリーから `doc/bin` ディレクトリーにナビゲートします。
   c. 次のように `ic-update` スクリプトを実行します。

      ```
      ic-update
      ```
2. Windows オペレーティング・システムの場合、次のようにします。
   a. コマンド・ウィンドウを開きます。
   b. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、DB2 インフォメーション・センターは、`<Program Files>¥IBM¥DB2 Information Center¥Version 9.7` ディレクトリーにインストールされています (`<Program Files>` は「Program Files」ディレクトリーのロケーション)。
   c. インストール・ディレクトリーから `doc¥bin` ディレクトリーにナビゲートします。

d. 次のように ic-update.bat ファイルを実行します。

```
ic-update.bat
```

DB2 インフォメーション・センターが自動的に再始動します。更新が入手可能な場合、インフォメーション・センターに、更新された新しいトピックが表示されます。インフォメーション・センターの更新が入手可能でなかった場合、メッセージがログに追加されます。ログ・ファイルは、doc¥eclipse¥configuration ディレクトリーにあります。ログ・ファイル名はランダムに生成された名前です。例えば、1239053440785.log のようになります。

コンピューターまたはイントラネット・サーバーにインストールされた DB2 インフォメーション・センターの手動更新

DB2 インフォメーション・センターをローカルにインストールしている場合は、IBM から資料の更新を入手してインストールすることができます。

ローカルにインストールされた *DB2 インフォメーション・センター* を手動で更新するには、以下のことを行う必要があります。

1. コンピューター上の *DB2 インフォメーション・センター* を停止し、インフォメーション・センターをスタンドアロン・モードで再始動します。インフォメーション・センターをスタンドアロン・モードで実行すると、ネットワーク上の他のユーザーがそのインフォメーション・センターにアクセスできなくなります。これで、更新を適用できるようになります。DB2 インフォメーション・センターのワークステーション・バージョンは、常にスタンドアロン・モードで実行されます。
2. 「更新」機能を使用することにより、どんな更新が利用できるかを確認します。インストールしなければならない更新がある場合は、「更新」機能を使用してそれを入手およびインストールできます。

   注: ご使用の環境において、インターネットに接続されていないマシンに *DB2 インフォメーション・センター* の更新をインストールする必要がある場合、インターネットに接続されていて *DB2 インフォメーション・センター* がインストールされているマシンを使用して、更新サイトをローカル・ファイル・システムにミラーリングしてください。ネットワーク上の多数のユーザーが資料の更新をインストールする場合にも、更新サイトをローカルにミラーリングして、更新サイト用のプロキシーを作成することにより、個々のユーザーが更新を実行するのに要する時間を短縮できます。
   更新パッケージが入手可能な場合、「更新」機能を使用してパッケージを入手します。ただし、「更新」機能は、スタンドアロン・モードでのみ使用できます。
3. スタンドアロンのインフォメーション・センターを停止し、コンピューター上の *DB2 インフォメーション・センター* を再開します。

注: Windows 2008、Windows Vista (およびそれ以上) では、このセクションの後の部分でリストされているコマンドは管理者として実行する必要があります。完全な管理者特権でコマンド・プロンプトまたはグラフィカル・ツールを開くには、ショートカットを右クリックしてから、**「管理者として実行」**を選択します。

コンピューターまたはイントラネット・サーバーにインストール済みの *DB2* インフォメーション・センター を更新するには、以下のようにします。

1. *DB2* インフォメーション・センター を停止します。
   - Windows では、「**スタート**」 → 「**コントロール パネル**」 → 「**管理ツール**」 → 「**サービス**」をクリックします。次に、「**DB2 インフォメーション・センター**」サービスを右クリックして「**停止**」を選択します。
   - Linux では、以下のコマンドを入力します。

     ```
     /etc/init.d/db2icdv97 stop
     ```
2. インフォメーション・センターをスタンドアロン・モードで開始します。
   - Windows の場合:
     a. コマンド・ウィンドウを開きます。
     b. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、*DB2* インフォメーション・センター は、*Program_Files*¥IBM¥DB2 Information Center¥Version 9.7 ディレクトリーにインストールされています (*Program_Files* は Program Filesディレクトリーのロケーション)。
     c. インストール・ディレクトリーから doc¥bin ディレクトリーにナビゲートします。
     d. 次のように help_start.bat ファイルを実行します。

        ```
        help_start.bat
        ```
   - Linux の場合:
     a. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、*DB2* インフォメーション・センター は、/opt/ibm/db2ic/V9.7 ディレクトリーにインストールされています。
     b. インストール・ディレクトリーから doc/bin ディレクトリーにナビゲートします。
     c. 次のように help_start スクリプトを実行します。

        ```
        help_start
        ```

   システムのデフォルト Web ブラウザーが開き、スタンドアロンのインフォメーション・センターが表示されます。
3. 「**更新**」ボタン () をクリックします。(ブラウザーで JavaScript™ が有効になっている必要があります。) インフォメーション・センターの右側のパネルで、「**更新の検索 (Find Updates)**」をクリックします。 既存の文書に対する更新のリストが表示されます。
4. インストール・プロセスを開始するには、インストールする更新をチェックして選択し、「**更新のインストール**」をクリックします。
5. インストール・プロセスが完了したら、「**完了**」をクリックします。
6. 次のようにして、スタンドアロンのインフォメーション・センターを停止します。
   - Windows の場合は、インストール・ディレクトリーの doc¥bin ディレクトリーにナビゲートしてから、次のように help_end.bat ファイルを実行します。

     ```
     help_end.bat
     ```

注: help_end バッチ・ファイルには、help_start バッチ・ファイルを使用して開始したプロセスを安全に停止するのに必要なコマンドが含まれています。help_start.bat は、Ctrl-C や他の方法を使用して停止しないでください。

- Linux の場合は、インストール・ディレクトリーの doc/bin ディレクトリーにナビゲートしてから、次のように help_end スクリプトを実行します。

  ```
  help_end
  ```

  注: help_end スクリプトには、help_start スクリプトを使用して開始したプロセスを安全に停止するのに必要なコマンドが含まれています。他の方法を使用して、help_start スクリプトを停止しないでください。

7. *DB2 インフォメーション・センター* を再開します。
   - Windows では、「**スタート**」 → 「**コントロール パネル**」 → 「**管理ツール**」 → 「**サービス**」をクリックします。次に、「**DB2 インフォメーション・センター**」サービスを右クリックして「**開始**」を選択します。
   - Linux では、以下のコマンドを入力します。

     ```
     /etc/init.d/db2icdv97 start
     ```

更新された *DB2 インフォメーション・センター* に、更新された新しいトピックが表示されます。

DB2 チュートリアル

DB2 チュートリアルは、DB2 製品のさまざまな機能について学習するのを支援します。この演習をとおして段階的に学習することができます。

はじめに

インフォメーション・センター (http://publib.boulder.ibm.com/infocenter/db2help/) から、このチュートリアルの XHTML 版を表示できます。

演習の中で、サンプル・データまたはサンプル・コードを使用する場合があります。個々のタスクの前提条件については、チュートリアルを参照してください。

DB2 チュートリアル

チュートリアルを表示するには、タイトルをクリックします。

「*pureXML ガイド*」の『**pureXML®**』
: XML データを保管し、ネイティブ XML データ・ストアに対して基本的な操作を実行できるように、DB2 データベースをセットアップします。

「*Visual Explain チュートリアル*」の『**Visual Explain**』
: Visual Explain を使用して、パフォーマンスを向上させるために SQL ステートメントを分析し、最適化し、調整します。

DB2 トラブルシューティング情報

DB2 データベース製品を使用する際に役立つ、トラブルシューティングおよび問題判別に関する広範囲な情報を利用できます。

DB2 ドキュメンテーション

トラブルシューティング情報は、「*問題判別およびデータベース・パフォーマンスのチューニング*」または *DB2 インフォメーション・センター* の『データベースの基本』セクションにあります。ここには、DB2 診断ツールおよびユーティリティーを使用して、問題を切り分けて識別する方法、最も頻繁に起こる幾つかの問題に対するソリューションについての情報、および DB2 データベース製品を使用する際に発生する可能性のある問題の解決方法についての他のアドバイスがあります。

DB2 Technical Support の Web サイト

現在問題が発生していて、考えられる原因とソリューションを検索したい場合は、DB2 Technical Support の Web サイトを参照してください。Technical Support サイトには、最新の DB2 資料、TechNotes、プログラム診断依頼書 (APAR またはバグ修正)、フィックスパック、およびその他のリソースへのリンクが用意されています。この知識ベースを活用して、問題に対する有効なソリューションを探し出すことができます。

DB2 Technical Support の Web サイト (http://www.ibm.com/software/data/db2/support/db2_9/) にアクセスしてください。

ご利用条件

これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

個人使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

商業的使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。 ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任なしで提供されます。

付録 B. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。IBM 以外の製品に関する情報は、本書の最初の発行時点で入手可能な情報に基づいており、変更される場合があります。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。 IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、

利便性もしくは機能性があることをほのめかしたり、保証することはできません。サンプル・プログラムは、現存するままの状態で提供されるものであり、いかなる種類の保証も提供されません。 IBM は、これらのサンプル・プログラムの使用から生ずるいかなる損害に対しても責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物には、次のように、著作権表示を入れていただく必要があります。

© (*お客様の会社名*) (*西暦年*). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. _*年を入れる*_. All rights reserved.

商標

IBM、IBM ロゴおよび ibm.com® は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml をご覧ください。

以下は、それぞれ各社の商標または登録商標です。

- Linux は、Linus Torvalds の米国およびその他の国における商標です。
- Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。
- UNIX は The Open Group の米国およびその他の国における登録商標です。
- Intel、Intel ロゴ、Intel Inside®、 Intel Inside ロゴ、Intel® Centrino®、Intel Centrino ロゴ、Celeron®、Intel® Xeon®、Intel SpeedStep®、Itanium、Pentium® は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。
- Microsoft、 Windows、 Windows NT®、および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

［マ行］

［ヤ行］

［ラ行］

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Printed in Japan

SC88-5877-02

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21

Spine information:

IBM DB2 9.7 for Linux, UNIX, and Windows　バージョン 9 リリース 7　SQL プロシージャー言語 : アプリケーションのイネーブルメントおよびサポート