Oracle® Database
データベース PL/SQL 言語リファレンス
19c

F16151-04(原本部品番号:E96448-05)

2023年3月

タイトルおよび著作権情報

Oracle Databaseデータベース PL/SQL言語リファレンス, 19c

F16151-04

Copyright © 1996, 2023, Oracle and/or its affiliates.

原本著者: Louise Morin

原本協力著者: S.Moore、L.Jayapalan、P.Huey、E.Belden、K.Rich、V.Moore

原本協力者: D.Alpern、S.Agrawal、H.Baer、S.Castledine、T.Chang、B.Cheng、R.Dani、R.Decker、C.Iyer、A.Kruglikov、N.Le、W.Li、B.Llewellyn、T.Raney、R.Rajagopalan、I.Stocks、C.Wetherell、S.Wolicki、G.Viswanathan、M.Yang

目次

- 例一覧
- 図一覧
- 表一覧
- タイトルおよび著作権情報
- はじめに
  - 対象読者
  - ドキュメントのアクセシビリティ
  - 関連ドキュメント
  - 表記規則
  - 構文の説明
- 『Oracle Database PL/SQL言語リファレンス』のこのリリースの変更点
  - Oracle Databaseリリース19cにおける変更点
    - SQLマクロ
  - Oracle Databaseリリース18cにおける変更点
    - オブジェクト型の非存続性のサポート
    - 多相テーブル・ファンクション
    - 修飾式
  - 非推奨となった機能
  - サポート対象外機能
- 1 PL/SQLの概要
  - 1.1 PL/SQLのメリット
    - 1.1.1 SQLとの緊密な統合
    - 1.1.2 高いパフォーマンス
    - 1.1.3 高い生産性
    - 1.1.4 移植性
    - 1.1.5 スケーラビリティ
    - 1.1.6 管理性
    - 1.1.7 オブジェクト指向プログラミングのサポート
  - 1.2 PL/SQLの主な特長
    - 1.2.1 エラー処理
    - 1.2.2 ブロック
    - 1.2.3 変数および定数
    - 1.2.4 サブプログラム
    - 1.2.5 パッケージ
    - 1.2.6 トリガー
    - 1.2.7 入出力
    - 1.2.8 データの抽象化
      - 1.2.8.1 カーソル
      - 1.2.8.2 コンポジット変数
      - 1.2.8.3 %ROWTYPE属性の使用
      - 1.2.8.4 %TYPE属性の使用

- 1.2.8.5 抽象データ型
- 1.2.9 制御文
- 1.2.10 条件付きコンパイル
- 1.2.11 問合せ結果セットの一度に1行ずつの処理
- 1.3 PL/SQLのアーキテクチャ
  - 1.3.1 PL/SQLエンジン
  - 1.3.2 PL/SQLユニットおよびコンパイル・パラメータ
- 1.4 PL/SQLでの機密情報の保護
- 2 PL/SQL言語の基礎
  - 2.1 文字セット
    - 2.1.1 データベース文字セット
    - 2.1.2 各国語文字セット
    - 2.1.3 データ・バインドされた照合について
  - 2.2 字句単位
    - 2.2.1 デリミタ
    - 2.2.2 識別子
      - 2.2.2.1 予約語およびキーワード
      - 2.2.2.2 事前定義の識別子
      - 2.2.2.3 ユーザー定義の識別子
        - 2.2.2.3.1 通常のユーザー定義の識別子
        - 2.2.2.3.2 ユーザー定義の引用識別子
    - 2.2.3 リテラル
    - 2.2.4 プラグマ
    - 2.2.5 コメント
      - 2.2.5.1 単一行コメント
      - 2.2.5.2 複数行コメント
    - 2.2.6 字句単位の間の空白文字
  - 2.3 宣言
    - 2.3.1 NOT NULL制約
    - 2.3.2 変数の宣言
    - 2.3.3 定数の宣言
    - 2.3.4 変数および定数の初期値
    - 2.3.5 %TYPE属性を使用した項目の宣言
  - 2.4 識別子に対する参照
  - 2.5 識別子の有効範囲と可視性
  - 2.6 変数への値の代入
    - 2.6.1 代入文を使用した変数への値の代入
    - 2.6.2 SELECT INTO文を使用した変数への値の代入
    - 2.6.3 サブプログラムのパラメータとしての変数への値の代入
    - 2.6.4 ブール変数への値の代入
  - 2.7 式
    - 2.7.1 連結演算子
    - 2.7.2 演算子の優先順位

- 2.7.3 論理演算子
- 2.7.4 短絡評価
- 2.7.5 比較演算子
  - 2.7.5.1 IS [NOT] NULL演算子
  - 2.7.5.2 関係演算子
    - 2.7.5.2.1 算術比較
    - 2.7.5.2.2 BOOLEAN比較
    - 2.7.5.2.3 文字比較
    - 2.7.5.2.4 日付比較
  - 2.7.5.3 LIKE演算子
  - 2.7.5.4 BETWEEN演算子
  - 2.7.5.5 IN演算子
- 2.7.6 BOOLEAN式
- 2.7.7 CASE式
  - 2.7.7.1 単純なCASE式
  - 2.7.7.2 検索CASE式
- 2.7.8 PL/SQL式でのSQLファンクション
- 2.7.9 静的な式
  - 2.7.9.1 静的なPLS_INTEGER式
  - 2.7.9.2 静的なBOOLEAN式
  - 2.7.9.3 静的なVARCHAR2式
  - 2.7.9.4 静的定数
- 2.8 エラー・レポート・ファンクション
- 2.9 条件付きコンパイル
  - 2.9.1 条件付きコンパイルの動作方法
    - 2.9.1.1 プリプロセッサ制御トークン
    - 2.9.1.2 選択ディレクティブ
    - 2.9.1.3 エラー・ディレクティブ
    - 2.9.1.4 問合せディレクティブ
      - 2.9.1.4.1 事前定義の問合せディレクティブ
      - 2.9.1.4.2 問合せディレクティブへの値の代入
      - 2.9.1.4.3 解決できない問合せディレクティブ
    - 2.9.1.5 DBMS_DB_VERSIONパッケージ
  - 2.9.2 条件付きコンパイルの例
  - 2.9.3 処理後のソース・テキストの取得と出力
  - 2.9.4 条件付きコンパイル・ディレクティブの制限
- 3 PL/SQLデータ型
  - 3.1 SQLデータ型
    - 3.1.1 最大サイズの相違
    - 3.1.2 BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQL定数
    - 3.1.3 BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQLサブタイプ
    - 3.1.4 CHARおよびVARCHAR2変数
      - 3.1.4.1 長すぎる値の代入または挿入

- 3.1.4.2 マルチバイト文字の変数の宣言
- 3.1.4.3 CHARとVARCHAR2のデータ型の違い
    - 3.1.4.3.1 事前定義のサブタイプ
    - 3.1.4.3.2 空白埋めの動作方法
    - 3.1.4.3.3 値の比較
- 3.1.5 LONGおよびLONG RAW変数
- 3.1.6 ROWIDおよびUROWID変数
- 3.2 BOOLEANデータ型
- 3.3 PLS_INTEGERおよびBINARY_INTEGERデータ型
    - 3.3.1 PLS_INTEGERのオーバーフローの回避
    - 3.3.2 PLS_INTEGERの事前定義のサブタイプ
    - 3.3.3 PLS_INTEGERのSIMPLE_INTEGERサブタイプ
        - 3.3.3.1 SIMPLE_INTEGERのオーバーフローの方法
        - 3.3.3.2 SIMPLE_INTEGERと他のオペランドの両方を使用する式
        - 3.3.3.3 SIMPLE_INTEGERの範囲内の整数リテラル
- 3.4 ユーザー定義のPL/SQLサブタイプ
    - 3.4.1 無制約のサブタイプ
    - 3.4.2 制約付きサブタイプ
    - 3.4.3 同じデータ型ファミリ内のベース型を使用したサブタイプ
- 4 PL/SQLの制御文
    - 4.1 条件付き選択文
        - 4.1.1 IF THEN文
        - 4.1.2 IF THEN ELSE文
        - 4.1.3 IF THEN ELSIF文
        - 4.1.4 単純なCASE文
        - 4.1.5 検索CASE文
    - 4.2 LOOP文
        - 4.2.1 基本LOOP文
        - 4.2.2 EXIT文
        - 4.2.3 EXIT WHEN文
        - 4.2.4 CONTINUE文
        - 4.2.5 CONTINUE WHEN文
        - 4.2.6 FOR LOOP文
            - 4.2.6.1 FOR LOOP索引
            - 4.2.6.2 下限と上限
            - 4.2.6.3 FOR LOOP文内のEXIT WHENまたはCONTINUE WHEN文
        - 4.2.7 WHILE LOOP文
    - 4.3 順次制御文
        - 4.3.1 GOTO文
        - 4.3.2 NULL文
- 5 PL/SQLのコレクションとレコード
    - 5.1 コレクション型
    - 5.2 連想配列

- 5.2.1 連想配列の定数の宣言
- 5.2.2 文字列で索引付けされている結合配列に影響を与えるNLSパラメータ値
    - 5.2.2.1 結合配列に移入した後のNLSパラメータ値の変更
    - 5.2.2.2 VARCHAR2以外のデータ型の索引
    - 5.2.2.3 リモート・データベースへの結合配列の受渡し
- 5.2.3 結合配列の適切な使用方法
- 5.3 VARRAY (可変サイズの配列)
    - 5.3.1 VARRAYの適切な使用方法
- 5.4 ネストした表
    - 5.4.1 ネストした表と配列の重要な相違点
    - 5.4.2 ネストした表の適切な使用方法
- 5.5 コレクションのコンストラクタ
- 5.6 修飾式の概要
- 5.7 コレクション変数への値の代入
    - 5.7.1 データ型の互換性
    - 5.7.2 VARRAYまたはネストした表の変数へのNULL値の代入
    - 5.7.3 ネストした表の変数へのSET演算結果の代入
- 5.8 多次元コレクション
- 5.9 コレクションの比較
    - 5.9.1 VARRAYおよびネストした表の変数とNULLの比較
    - 5.9.2 ネストした表が等しいかどうかの比較
    - 5.9.3 ネストした表のSQL MULTISET条件による比較
- 5.10 コレクション・メソッド
    - 5.10.1 DELETEコレクション・メソッド
    - 5.10.2 TRIMコレクション・メソッド
    - 5.10.3 EXTENDコレクション・メソッド
    - 5.10.4 EXISTSコレクション・メソッド
    - 5.10.5 FIRSTおよびLASTコレクション・メソッド
        - 5.10.5.1 結合配列に対するFIRSTおよびLASTメソッド
        - 5.10.5.2 VARRAYに対するFIRSTおよびLASTメソッド
        - 5.10.5.3 ネストした表に対するFIRSTおよびLASTメソッド
    - 5.10.6 COUNTコレクション・メソッド
        - 5.10.6.1 VARRAYに対するCOUNTメソッド
        - 5.10.6.2 ネストした表に対するCOUNTメソッド
    - 5.10.7 LIMITコレクション・メソッド
    - 5.10.8 PRIORおよびNEXTコレクション・メソッド
- 5.11 パッケージ仕様部で定義されたコレクション型
- 5.12 レコード変数
    - 5.12.1 レコード変数の初期値
    - 5.12.2 レコード定数の宣言
    - 5.12.3 RECORD型
    - 5.12.4 %ROWTYPE属性を使用した項目の宣言
        - 5.12.4.1 常に行の全体を表すレコード変数の宣言

- 5.12.4.2 行の一部を表すことができるレコード変数の宣言
- 5.12.4.3 %ROWTYPE属性と仮想列
- 5.12.4.4 %ROWTYPE属性と非表示列
- 5.13 レコード変数への値の代入
  - 5.13.1 あるレコード変数から別のレコード変数への代入
  - 5.13.2 レコード変数への行の全体または一部の代入
    - 5.13.2.1 SELECT INTOを使用した行のレコード変数への代入
    - 5.13.2.2 FETCHを使用した行のレコード変数への代入
    - 5.13.2.3 PL/SQLレコード変数に行を戻すSQL文の使用
  - 5.13.3 レコード変数へのNULLの代入
- 5.14 レコードの比較
- 5.15 表へのレコードの挿入
- 5.16 レコードを使用した行の更新
- 5.17 レコードの挿入/更新に関する制限
- 6 PL/SQLの静的SQL
  - 6.1 静的SQLの説明
    - 6.1.1 文
    - 6.1.2 擬似列
      - 6.1.2.1 PL/SQLのCURRVALおよびNEXTVAL
  - 6.2 カーソルの概要
    - 6.2.1 暗黙カーソル
      - 6.2.1.1 SQL%ISOPEN属性: カーソルがオープンしているかどうか
      - 6.2.1.2 SQL%FOUND属性: 影響を受けた行があったかどうか
      - 6.2.1.3 SQL%NOTFOUND属性: どの行も影響を受けなかったかどうか
      - 6.2.1.4 SQL%ROWCOUNT属性: 影響を受けた行数
    - 6.2.2 明示カーソル
      - 6.2.2.1 明示カーソルの宣言および定義
      - 6.2.2.2 明示カーソルのオープンおよびクローズ
      - 6.2.2.3 明示カーソルによるデータのフェッチ
      - 6.2.2.4 明示カーソル問合せの変数
      - 6.2.2.5 明示カーソル問合せで列の別名が必要な場合
      - 6.2.2.6 パラメータを受け入れる明示カーソル
        - 6.2.2.6.1 デフォルト値を持つ仮カーソル・パラメータ
        - 6.2.2.6.2 デフォルト値を持つ仮カーソル・パラメータの追加
      - 6.2.2.7 明示カーソル属性
        - 6.2.2.7.1 %ISOPEN属性: カーソルがオープンしているかどうか
        - 6.2.2.7.2 %FOUND属性: 1行がフェッチされたかどうか
        - 6.2.2.7.3 %NOTFOUND属性: どの行もフェッチされていないかどうか
        - 6.2.2.7.4 %ROWCOUNT属性: フェッチされた行数
  - 6.3 問合せ結果セットの処理
    - 6.3.1 SELECT INTO文による問合せ結果セットの処理
      - 6.3.1.1 単一行の結果セットの処理
      - 6.3.1.2 多数の複数行の結果セットの処理

- 6.3.2 カーソルFOR LOOP文による問合せ結果セットの処理
- 6.3.3 明示カーソル、OPEN、FETCHおよびCLOSEによる問合せ結果セットの処理
- 6.3.4 副問合せによる問合せ結果セットの処理
- 6.4 カーソル変数
  - 6.4.1 カーソル変数の作成
  - 6.4.2 カーソル変数のオープンおよびクローズ
  - 6.4.3 カーソル変数によるデータのフェッチ
  - 6.4.4 カーソル変数への値の代入
  - 6.4.5 カーソル変数問合せの変数
  - 6.4.6 コレクションの問合せ
  - 6.4.7 カーソル変数の属性
  - 6.4.8 サブプログラム・パラメータとしてのカーソル変数
  - 6.4.9 ホスト変数としてのカーソル変数
- 6.5 CURSOR式
- 6.6 トランザクションの処理および制御
  - 6.6.1 COMMIT文
  - 6.6.2 ROLLBACK文
  - 6.6.3 SAVEPOINT文
  - 6.6.4 暗黙的なロールバック
  - 6.6.5 SET TRANSACTION文
  - 6.6.6 デフォルトのロックの上書き
    - 6.6.6.1 LOCK TABLE文
    - 6.6.6.2 SELECT FOR UPDATEカーソルとFOR UPDATEカーソル
    - 6.6.6.3 ROWID擬似列を使用したCURRENT OF句のシミュレート
- 6.7 自律型トランザクション
  - 6.7.1 自律型トランザクションのメリット
  - 6.7.2 トランザクション・コンテキスト
  - 6.7.3 トランザクションの可視性
  - 6.7.4 自律型ルーチンの宣言
  - 6.7.5 自律型トランザクションの制御
    - 6.7.5.1 自律型ルーチンの開始と終了
    - 6.7.5.2 自律型トランザクションのコミットおよびロールバック
    - 6.7.5.3 セーブポイント
    - 6.7.5.4 自律型トランザクションでのエラーの回避
  - 6.7.6 自律型トリガー
  - 6.7.7 SQLからの自律型ファンクションの起動
- 7 PL/SQLの動的SQL
  - 7.1 動的SQLが必要な場合
  - 7.2 システム固有の動的SQL
    - 7.2.1 EXECUTE IMMEDIATE文
    - 7.2.2 OPEN FOR文、FETCH文およびCLOSE文
    - 7.2.3 動的SQL文内でのプレースホルダ名の繰返し
      - 7.2.3.1 動的SQL文が無名ブロックまたはCALL文でない

- 7.2.3.2 動的SQL文が無名ブロックまたはCALL文
- 7.3 DBMS_SQLパッケージ
  - 7.3.1 DBMS_SQL.RETURN_RESULTプロシージャ
  - 7.3.2 DBMS_SQL.GET_NEXT_RESULTプロシージャ
  - 7.3.3 DBMS_SQL.TO_REFCURSORファンクション
  - 7.3.4 DBMS_SQL.TO_CURSOR_NUMBERファンクション
- 7.4 SQLインジェクション
  - 7.4.1 SQLインジェクション方法
    - 7.4.1.1 文の変更
    - 7.4.1.2 文のインジェクション
    - 7.4.1.3 データ型変換
  - 7.4.2 SQLインジェクションの回避
    - 7.4.2.1 バインド変数
    - 7.4.2.2 妥当性チェック
    - 7.4.2.3 明示的な書式モデル
- 8 PL/SQLサブプログラム
  - 8.1 サブプログラムを使用する理由
  - 8.2 ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラム
  - 8.3 サブプログラムの起動
  - 8.4 サブプログラムのプロパティ
  - 8.5 サブプログラムの各部
    - 8.5.1 ファンクションの追加部分
    - 8.5.2 RETURN文
      - 8.5.2.1 ファンクションのRETURN文
      - 8.5.2.2 プロシージャのRETURN文
      - 8.5.2.3 無名ブロックのRETURN文
  - 8.6 前方宣言
  - 8.7 サブプログラムのパラメータ
    - 8.7.1 サブプログラムの仮パラメータと実パラメータ
      - 8.7.1.1 制約付きサブタイプの仮パラメータ
    - 8.7.2 サブプログラムのパラメータの引渡し方法
    - 8.7.3 サブプログラムのパラメータ・モード
    - 8.7.4 サブプログラムのパラメータのエイリアシング
      - 8.7.4.1 参照渡しパラメータによるサブプログラムのパラメータのエイリアシング
      - 8.7.4.2 カーソル変数パラメータによるサブプログラムのパラメータのエイリアシング
    - 8.7.5 サブプログラムのINパラメータのデフォルト値
    - 8.7.6 実パラメータの位置表記法、名前表記法および混合表記法
  - 8.8 サブプログラムの起動の解決
  - 8.9 オーバーロードされたサブプログラム
    - 8.9.1 数値データ型のみが異なる仮パラメータ
    - 8.9.2 オーバーロードできないサブプログラム
    - 8.9.3 サブプログラムのオーバーロード・エラー
  - 8.10 再帰的サブプログラム

- 8.11 サブプログラムの副作用
- 8.12 PL/SQLファンクション結果キャッシュ
  - 8.12.1 ファンクションの結果キャッシュの有効化
  - 8.12.2 結果がキャッシュされるファンクションを使用するアプリケーションの開発
  - 8.12.3 結果がキャッシュされるファンクションの要件
  - 8.12.4 結果がキャッシュされるファンクションの例
    - 8.12.4.1 結果がキャッシュされるアプリケーション構成パラメータ
    - 8.12.4.2 結果がキャッシュされる再帰ファンクション
  - 8.12.5 結果がキャッシュされるファンクションの高度なトピック
    - 8.12.5.1 キャッシュ・ヒットの規則
    - 8.12.5.2 結果キャッシュのバイパス
    - 8.12.5.3 結果がキャッシュされるファンクションによるセッション固有の設定の処理
    - 8.12.5.4 結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理
    - 8.12.5.5 結果キャッシュの粒度の選択
    - 8.12.5.6 Oracle RAC環境での結果キャッシュ
    - 8.12.5.7 結果キャッシュの管理
    - 8.12.5.8 結果がキャッシュされるファンクションが依存するPL/SQLユニットへのホット・パッチの適用
- 8.13 SQL文によって起動できるPL/SQLファンクション
- 8.14 実行者権限および定義者権限(AUTHIDプロパティ)
  - 8.14.1 PL/SQLパッケージおよびスタンドアロン・サブプログラムへのロールの付与
  - 8.14.2 IRユニットにはテンプレート・オブジェクトが必要
  - 8.14.3 DRユニット内の接続ユーザー・データベース・リンク
- 8.15 外部サブプログラム

- 9 PL/SQLトリガー
  - 9.1 トリガーの概要
  - 9.2 トリガーを使用する理由
  - 9.3 DMLトリガー
    - 9.3.1 トリガーを起動するDML文を検出する条件述語
    - 9.3.2 INSTEAD OF DMLトリガー
    - 9.3.3 複合DMLトリガー
      - 9.3.3.1 複合DMLトリガーの構造
      - 9.3.3.2 複合DMLトリガーの制限
      - 9.3.3.3 複合DMLトリガーのパフォーマンス上のメリット
      - 9.3.3.4 バルク挿入での複合DMLトリガーの使用
      - 9.3.3.5 複合DMLトリガーを使用した変更表エラーの回避
    - 9.3.4 参照整合性を保証するトリガー
      - 9.3.4.1 子表に対する外部キー・トリガー
      - 9.3.4.2 親表に対するUPDATEおよびDELETE RESTRICTトリガー
      - 9.3.4.3 親表に対するUPDATEおよびDELETE SET NULLトリガー
      - 9.3.4.4 親表に対するDELETE CASCADEトリガー
      - 9.3.4.5 親表に対するUPDATE CASCADEトリガー

- 9.3.4.6 複雑な制約チェックに対するトリガー
- 9.3.4.7 複雑なセキュリティ認可に対するトリガー
- 9.3.4.8 透過的なイベント・ロギングに対するトリガー
- 9.3.4.9 導出列値に対するトリガー
- 9.3.4.10 複合更新可能ビューを作成するトリガー
- 9.3.4.11 ファイングレイン・アクセス制御に対するトリガー

- 9.4 相関名および疑似レコード
  - 9.4.1 OBJECT_VALUE擬似列
- 9.5 システム・トリガー
  - 9.5.1 SCHEMAトリガー
  - 9.5.2 DATABASEトリガー
  - 9.5.3 INSTEAD OF CREATEトリガー
- 9.6 トリガーによって起動されるサブプログラム
- 9.7 トリガーのコンパイル、無効化および再コンパイル
- 9.8 トリガーでの例外処理
- 9.9 トリガー設計のガイドライン
- 9.10 トリガーの制限
  - 9.10.1 トリガーのサイズ制限
  - 9.10.2 トリガーのLONGおよびLONG RAWデータ型の制約
  - 9.10.3 変更表の制限
- 9.11 トリガーの起動順序
- 9.12 トリガーの有効化および無効化
- 9.13 トリガーの変更およびデバッグ
- 9.14 トリガーおよびOracle Databaseデータ転送ユーティリティ
- 9.15 イベントをパブリッシュするトリガー
  - 9.15.1 イベント属性ファンクション
  - 9.15.2 データベース・イベント・トリガーのイベント属性ファンクション
  - 9.15.3 クライアント・イベント・トリガーのイベント属性ファンクション
- 9.16 トリガーに関する情報が表示されるビュー

- 10 PL/SQLパッケージ
  - 10.1 パッケージ
  - 10.2 パッケージを使用する理由
  - 10.3 パッケージ仕様部
    - 10.3.1 適切なパブリック項目
    - 10.3.2 パッケージ仕様部の作成
  - 10.4 パッケージ本体
  - 10.5 パッケージのインスタンス化および初期化
  - 10.6 パッケージの状態
  - 10.7 SERIALLY_REUSABLEパッケージ
    - 10.7.1 SERIALLY_REUSABLEパッケージの作成
    - 10.7.2 SERIALLY_REUSABLEパッケージの作業単位
    - 10.7.3 SERIALLY_REUSABLEパッケージの明示カーソル
  - 10.8 パッケージ作成のガイドライン

- 10.9 パッケージの例
- 10.10 STANDARDパッケージによるPL/SQL環境の定義
- 11 PL/SQLのエラー処理
  - 11.1 コンパイル時の警告
    - 11.1.1 DBMS_WARNINGパッケージ
  - 11.2 例外処理の概要
    - 11.2.1 例外のカテゴリ
    - 11.2.2 例外ハンドラのメリット
    - 11.2.3 例外の回避および処理のガイドライン
  - 11.3 内部的に定義された例外
  - 11.4 事前定義の例外
  - 11.5 ユーザー定義の例外
  - 11.6 再宣言された事前定義の例外
  - 11.7 例外の明示的な呼出し
    - 11.7.1 RAISE文
      - 11.7.1.1 ユーザー定義の例外のRAISE文による呼出し
      - 11.7.1.2 内部的に定義された例外のRAISE文による呼出し
      - 11.7.1.3 現行の例外のRAISE文による再呼出し
    - 11.7.2 RAISE_APPLICATION_ERRORプロシージャ
  - 11.8 例外の伝播
    - 11.8.1 宣言の中で呼び出された例外の伝播
    - 11.8.2 例外ハンドラの中で呼び出された例外の伝播
  - 11.9 未処理例外
  - 11.10 エラー・コードとエラー・メッセージの取得
  - 11.11 例外処理後に実行を続ける方法
  - 11.12 例外処理後のトランザクションの再試行
  - 11.13 分散問合せでのエラー処理
- 12 PL/SQLの最適化とチューニング
  - 12.1 PL/SQLオプティマイザ
    - 12.1.1 サブプログラムのインライン化
  - 12.2 チューニングの候補
  - 12.3 CPUオーバーヘッドの最小化
    - 12.3.1 SQL文のチューニング
    - 12.3.2 問合せにおけるファンクションの起動のチューニング
    - 12.3.3 サブプログラムの起動のチューニング
    - 12.3.4 ループのチューニング
    - 12.3.5 計算集中型PL/SQLコードのチューニング
      - 12.3.5.1 ハードウェア算術を使用するデータ型の使用
      - 12.3.5.2 パフォーマンスを重視するコードでの制約付きサブタイプの回避
      - 12.3.5.3 暗黙的なデータ型変換の最少化
    - 12.3.6 SQL文字関数の使用
    - 12.3.7 最低コストの条件テストの先頭への配置
  - 12.4 バルクSQLおよびバルク・バインド

- 12.4.1 FORALL文
  - 12.4.1.1 疎コレクションに対するFORALL文の使用
  - 12.4.1.2 FORALL文での未処理例外
  - 12.4.1.3 FORALL例外の即時処理
  - 12.4.1.4 FORALL文が完了した後のFORALL例外の処理
    - 12.4.1.4.1 疎コレクションおよびSQL%BULK_EXCEPTIONS
  - 12.4.1.5 FORALL文の影響を受ける行の数の取得
- 12.4.2 BULK COLLECT句
  - 12.4.2.1 BULK COLLECT句を使用したSELECT INTO文
    - 12.4.2.1.1 SELECT BULK COLLECT INTO文およびエイリアシング
    - 12.4.2.1.2 SELECT BULK COLLECT INTO文の行の制限
    - 12.4.2.1.3 コレクションのループのガイドライン
  - 12.4.2.2 BULK COLLECT句を使用したFETCH文
    - 12.4.2.2.1 FETCH BULK COLLECT文の行の制限
  - 12.4.2.3 BULK COLLECT句を使用したRETURNING INTO句
- 12.4.3 FORALL文とBULK COLLECT句の併用
- 12.4.4 ホスト配列のクライアント・バルク・バインド

- 12.5 複数変換用のパイプライン・テーブル・ファンクションの連鎖
  - 12.5.1 テーブル・ファンクションの概要
  - 12.5.2 パイプライン・テーブル・ファンクションの作成
  - 12.5.3 変換ファンクションとしてのパイプライン・テーブル・ファンクション
  - 12.5.4 パイプライン・テーブル・ファンクションの連鎖
  - 12.5.5 パイプライン・テーブル・ファンクションの結果からのフェッチ
  - 12.5.6 パイプライン・テーブル・ファンクションへのCURSOR式の引渡し
  - 12.5.7 パイプライン・テーブル・ファンクションの結果に対するDML文
  - 12.5.8 NO_DATA_NEEDED例外
- 12.6 多相テーブル・ファンクションの概要
  - 12.6.1 多相テーブル・ファンクションの定義
  - 12.6.2 多相テーブル・ファンクションの実装
  - 12.6.3 多相テーブル・ファンクションの起動
    - 12.6.3.1 可変個引数擬似演算子
    - 12.6.3.2 COLUMNS擬似演算子
  - 12.6.4 多相テーブル・ファンクションのコンパイルおよび実行
  - 12.6.5 多相テーブル・ファンクションの最適化
  - 12.6.6 Skip_col多相テーブル・ファンクションの例
  - 12.6.7 To_doc多相テーブル・ファンクションの例
  - 12.6.8 Implicit_echo多相表関数の例
- 12.7 大規模な表のパラレルでの更新
- 12.8 ユーザー定義の識別子に関するデータの収集
- 12.9 PL/SQLプログラムのプロファイルおよびトレース
- 12.10 システム固有の実行のためのPL/SQLユニットのコンパイル
  - 12.10.1 PL/SQLのネイティブ・コンパイルを使用するかどうかの判断
  - 12.10.2 PL/SQLのネイティブ・コンパイルの動作

- 12.10.3 依存性、無効化および再評価
- 12.10.4 PL/SQLのネイティブ・コンパイルで使用する新しいデータベースの設定
- 12.10.5 PL/SQLネイティブ・コンパイルまたは解釈コンパイルで使用するデータベース全体のコンパイル

- 13 PL/SQLの言語要素
  - 13.1 ACCESSIBLE BY句
  - 13.2 AGGREGATE句
  - 13.3 代入文
  - 13.4 AUTONOMOUS_TRANSACTIONプラグマ
  - 13.5 基本LOOP文
  - 13.6 ブロック
  - 13.7 コール仕様
  - 13.8 CASE文
  - 13.9 CLOSE文
  - 13.10 コレクション・メソッドの起動
  - 13.11 コレクション変数の宣言
  - 13.12 コメント
  - 13.13 COMPILE句
  - 13.14 定数の宣言
  - 13.15 CONTINUE文
  - 13.16 COVERAGEプラグマ
  - 13.17 カーソルFOR LOOP文
  - 13.18 カーソル変数の宣言
  - 13.19 データ型属性
  - 13.20 DEFAULT COLLATION句
  - 13.21 DELETE文の拡張機能
  - 13.22 DEPRECATEプラグマ
  - 13.23 DETERMINISTIC句
  - 13.24 要素の仕様
  - 13.25 EXCEPTION_INITプラグマ
  - 13.26 例外宣言
  - 13.27 例外ハンドラ
  - 13.28 EXECUTE IMMEDIATE文
  - 13.29 EXIT文
  - 13.30 明示カーソルの宣言および定義
  - 13.31 式
  - 13.32 FETCH文
  - 13.33 FOR LOOP文
  - 13.34 FORALL文
  - 13.35 仮パラメータの宣言
  - 13.36 ファンクションの宣言および定義
  - 13.37 GOTO文
  - 13.38 IF文

- 13.39 暗黙カーソル属性
- 13.40 INLINEプラグマ
- 13.41 実行者権限および定義者権限句
- 13.42 INSERT文の拡張機能
- 13.43 名前付きカーソル属性
- 13.44 NULL文
- 13.45 OPEN文
- 13.46 OPEN FOR文
- 13.47 PARALLEL_ENABLE句
- 13.48 PIPE ROW文
- 13.49 PIPELINED句
- 13.50 プロシージャの宣言および定義
- 13.51 RAISE文
- 13.52 レコード変数の宣言
- 13.53 RESTRICT_REFERENCESプラグマ
- 13.54 RETURN文
- 13.55 RETURNING INTO句
- 13.56 RESULT_CACHE句
- 13.57 %ROWTYPE属性
- 13.58 スカラー変数の宣言
- 13.59 SELECT INTO文
- 13.60 SERIALLY_REUSABLEプラグマ
- 13.61 SHARING句
- 13.62 SQLCODEファンクション
- 13.63 SQLERRMファンクション
- 13.64 SQL_MACRO句
- 13.65 %TYPE属性
- 13.66 UDFプラグマ
- 13.67 UPDATE文の拡張機能
- 13.68 WHILE LOOP文

- 14 ストアドPL/SQLユニット用のSQL文
  - 14.1 ALTER FUNCTION文
  - 14.2 ALTER LIBRARY文
  - 14.3 ALTER PACKAGE文
  - 14.4 ALTER PROCEDURE文
  - 14.5 ALTER TRIGGER文
  - 14.6 ALTER TYPE文
  - 14.7 CREATE FUNCTION文
  - 14.8 CREATE LIBRARY文
  - 14.9 CREATE PACKAGE文
  - 14.10 CREATE PACKAGE BODY文
  - 14.11 CREATE PROCEDURE文
  - 14.12 CREATE TRIGGER文

    - 14.13 CREATE TYPE文
    - 14.14 CREATE TYPE BODY文
    - 14.15 DROP FUNCTION文
    - 14.16 DROP LIBRARY文
    - 14.17 DROP PACKAGE文
    - 14.18 DROP PROCEDURE文
    - 14.19 DROP TRIGGER文
    - 14.20 DROP TYPE文
    - 14.21 DROP TYPE BODY文
- A PL/SQLのソース・テキストのラップ
    - A.1 PL/SQLソース・テキストのラップの制限
    - A.2 PL/SQLソース・テキストのラップのガイドライン
    - A.3 PL/SQLラッパー・ユーティリティによるPL/SQLソース・テキストのラップ
    - A.4 DBMS_DDLサブプログラムによるPL/SQLソース・テキストのラップ
- B PL/SQLの名前解決
    - B.1 修飾名およびドット表記法
    - B.2 列名の優先順位
    - B.3 PL/SQLとSQLでの名前解決ルールの相違点
    - B.4 静的SQL文における名前解決
    - B.5 取得
        - B.5.1 外部取得
        - B.5.2 同一有効範囲の取得
        - B.5.3 内部取得
    - B.6 SELECT文およびDML文の内部取得の回避
        - B.6.1 属性およびメソッドへの参照の修飾
        - B.6.2 行の式への参照の修飾
- C PL/SQLのプログラム上の制限
- D PL/SQLの予約語およびキーワード
- E PL/SQLの事前定義のデータ型
- 索引

例一覧

- 1-1　PL/SQLブロック構造
- 1-2　問合せ結果行の一度に1行ずつの処理
- 2-1　ユーザー定義の引用識別子に対する大/小文字を区別しない有効な参照
- 2-2　ユーザー定義の引用識別子に対する大/小文字を区別しない無効な参照
- 2-3　ユーザー定義の引用識別子としての予約語
- 2-4　二重引用符の省略
- 2-5　大/小文字の区別の無視
- 2-6　単一行コメント
- 2-7　複数行コメント
- 2-8　空白文字によるソース・テキストの読みやすさの向上
- 2-9　NOT NULL制約が指定された変数の宣言
- 2-10　NULL値に初期化される変数
- 2-11　スカラー変数の宣言
- 2-12　定数の宣言
- 2-13　初期値が指定された変数および定数の宣言
- 2-14　デフォルトでNULLに初期化される変数
- 2-15　列と同じ型の変数の宣言
- 2-16　他の変数と同じ型の変数の宣言
- 2-17　識別子の有効範囲と可視性
- 2-18　ブロック・ラベルによる、再宣言されたグローバル識別子の修飾
- 2-19　サブプログラム名による識別子の修飾
- 2-20　同じ有効範囲での重複する識別子
- 2-21　異なるユニットでの同じ識別子の宣言
- 2-22　同じ有効範囲内で同じ名前を持つラベルとサブプログラム
- 2-23　複数の重複したラベルを使用するブロック
- 2-24　代入文を使用した変数への値の代入
- 2-25　SELECT INTO文を使用した変数への値の代入
- 2-26　IN OUTサブプログラム・パラメータとしての変数への値の代入
- 2-27　BOOLEAN変数への値の代入
- 2-28　連結演算子
- 2-29　連結演算子とNULLオペランド
- 2-30　カッコを使用した評価順序の制御
- 2-31　ネストしたカッコを使用した式
- 2-32　カッコを使用した読みやすさの向上
- 2-33　演算子の優先順位
- 2-34　BOOLEAN変数を出力するプロシージャ
- 2-35　AND演算子
- 2-36　OR演算子
- 2-37　NOT演算子
- 2-38　等しくない比較でのNULL値
- 2-39　等しい比較でのNULL値

- 2-40　NULLと等しいNOT NULL
- 2-41　論理演算子の演算順序の変更
- 2-42　短絡評価
- 2-43　式での関係演算子
- 2-44　式でのLIKE演算子
- 2-45　パターン内のエスケープ文字
- 2-46　式でのBETWEEN演算子
- 2-47　式でのIN演算子
- 2-48　NULL値を含む集合でのIN演算子
- 2-49　等価なBOOLEAN式
- 2-50　単純なCASE式
- 2-51　WHEN NULLを使用した単純なCASE式
- 2-52　検索CASE式
- 2-53　WHEN ... IS NULLを使用した検索CASE式
- 2-54　静的定数
- 2-55　事前定義の問合せディレクティブ
- 2-56　PL/SQLコンパイル・パラメータの値の表示
- 2-57　PLSQL_CCFLAGSによるそれ自体への値の代入
- 2-58　データベース・バージョンを確認するコード
- 2-59　異なるデータベース・バージョンに対する異なるコードのコンパイル
- 2-60　処理後のソース・テキストの表示
- 2-61　パッケージ仕様の定義で使用する条件付きコンパイル・ディレクティブ
- 2-62　サブプログラムの仮パラメータ・リストで使用する条件付きコンパイル・ディレクティブ
- 3-1　CHARおよびVARCHAR2の空白埋めの相違点
- 3-2　BOOLEAN値の出力
- 3-3　SQL文からのBOOLEANパラメータを持つPL/SQLファンクションの起動
- 3-4　オーバーフロー例外が発生するPLS_INTEGERの計算
- 3-5　例3-4のオーバーフローの回避
- 3-6　SIMPLE_INTEGERサブタイプの制約違反
- 3-7　使用意図を示すユーザー定義の無制約のサブタイプ
- 3-8　範囲外の値を検出するユーザー定義の制約付きサブタイプ
- 3-9　同じベース型を使用する制約付きサブタイプ間の暗黙的な変換
- 3-10　同じファミリ内のベース型を使用するサブタイプ間の暗黙的な変換
- 4-1　IF THEN文
- 4-2　IF THEN ELSE文
- 4-3　ネストしたIF THEN ELSE文
- 4-4　IF THEN ELSIF文
- 4-5　単純なCASE文をシミュレートするIF THEN ELSIF文
- 4-6　単純なCASE文
- 4-7　検索CASE文
- 4-8　CASE文でのELSE句にかわるEXCEPTION
- 4-9　EXIT文が含まれている基本LOOP文
- 4-10　EXIT WHEN文が含まれている基本LOOP文

- 4-11　EXIT WHEN文が含まれているネストされたラベル付き基本LOOP文
- 4-12　EXIT WHEN文が含まれているネストされたラベルなしの基本LOOP文
- 4-13　基本LOOP文内のCONTINUE文
- 4-14　基本LOOP文内のCONTINUE WHEN文
- 4-15　FOR LOOP文
- 4-16　反転FOR LOOP文
- 4-17　FOR LOOP文でのSTEP句のシミュレート
- 4-18　索引値を変更しようとするFOR LOOP文
- 4-19　外側の文によるFOR LOOP文の索引の参照
- 4-20　変数と同じ名前を持つFOR LOOP文の索引
- 4-21　FOR LOOP文による索引と同じ名前を持つ変数の参照
- 4-22　同じ索引名のネストしたFOR LOOP文
- 4-23　FOR LOOP文の境界
- 4-24　実行時に境界が指定されるFOR LOOP文
- 4-25　FOR LOOP文内のEXIT WHEN文
- 4-26　内側のFOR LOOP文内のEXIT WHEN文
- 4-27　内側のFOR LOOP文内のCONTINUE WHEN文
- 4-28　WHILE LOOP文
- 4-29　GOTO文
- 4-30　不適切なラベル配置
- 4-31　GOTO文によるラベル付きのNULL文への移動
- 4-32　GOTO文による外側のブロックへの制御の移動
- 4-33　GOTO文によるIF文内への無効な制御の移動
- 4-34　アクションを実行しないことを明示するNULL文
- 4-35　サブプログラム作成時のプレースホルダとしてのNULL文
- 4-36　単純なCASE文のELSE句でのNULL文
- 5-1　文字列で索引付けされている結合配列
- 5-2　PLS_INTEGERで索引付けされている連想配列を戻すファンクション
- 5-3　連想配列の定数の宣言
- 5-4　VARRAY (可変サイズの配列)
- 5-5　ローカル型のネストした表
- 5-6　スタンドアロン型のネストした表
- 5-7　コレクション(VARRAY)変数の空への初期化
- 5-8　修飾式を使用したRECORD型変数への値の代入
- 5-9　修飾式を使用した連想配列型変数への値の代入
- 5-10　コレクション代入のデータ型の互換性
- 5-11　ネストした表の変数へのNULL値の代入
- 5-12　ネストした表の変数へのSET演算結果の代入
- 5-13　2次元のVARRAY(VARRAYのVARRAY)
- 5-14　ネストした表のネストした表と整数のVARRAYのネストした表
- 5-15　結合配列のネストした表と文字列のVARRAYのネストした表
- 5-16　VARRAYおよびネストした表の変数とNULLの比較
- 5-17　ネストした表が等しいかどうかの比較

- 5-18　ネストした表のSQL MULTISET条件による比較
- 5-19　ネストした表でのDELETEメソッド
- 5-20　文字列で索引付けされている結合配列でのDELETEメソッド
- 5-21　ネストした表でのTRIMメソッド
- 5-22　ネストした表でのEXTENDメソッド
- 5-23　ネストした表でのEXISTSメソッド
- 5-24　PLS_INTEGERで索引付けされている連想配列のFIRSTとLASTの値
- 5-25　文字列で索引付けされている結合配列のFIRSTとLASTの値
- 5-26　FOR LOOP内のFIRSTおよびLASTによるVARRAYの出力
- 5-27　FOR LOOP内のFIRSTおよびLASTによるネストした表の出力
- 5-28　VARRAYのCOUNTとLASTの値
- 5-29　ネストした表のCOUNTとLASTの値
- 5-30　様々なコレクション型のLIMITとCOUNTの値
- 5-31　PRIORおよびNEXTメソッド
- 5-32　疎であるネストした表の要素の出力
- 5-33　同一定義のパッケージ・コレクション型とローカル・コレクション型
- 5-34　同一定義のパッケージ・コレクション型とスタンドアロン・コレクション型
- 5-35　レコード定数の宣言
- 5-36　RECORD型の定義および変数の宣言
- 5-37　RECORDフィールドを持つRECORD型(ネストされたレコード)
- 5-38　VARRAYフィールド持つRECORD型
- 5-39　同一定義のパッケージRECORD型とローカルRECORD型
- 5-40　データベース表の行全体を表す%ROWTYPE変数
- 5-41　初期値または制約を継承しない%ROWTYPE変数
- 5-42　データベース表の行の一部を表す%ROWTYPE変数
- 5-43　結合行を表す%ROWTYPE変数
- 5-44　%ROWTYPEレコードの表への挿入(間違った例)
- 5-45　%ROWTYPEレコードの表への挿入(正しい例)
- 5-46　非表示列の表示列への変更により影響を受ける%ROWTYPE
- 5-47　同じRECORD型のレコードから別のレコードへの代入
- 5-48　RECORD型のレコードへの%ROWTYPEレコードの代入
- 5-49　同じRECORD型のネストしたレコードから別のレコードへの代入
- 5-50　SELECT INTOを使用したレコード変数への値の代入
- 5-51　ファンクションが戻すレコードへのFETCHによる値の代入
- 5-52　UPDATE文を使用したレコード変数への値の代入
- 5-53　レコード変数へのNULLの代入
- 5-54　デフォルト値のレコードの挿入による表の初期化
- 5-55　レコードを使用した行の更新
- 6-1　静的SQL文
- 6-2　擬似列CURRVALおよびNEXTVAL
- 6-3　SQL%FOUND暗黙カーソル属性
- 6-4　SQL%ROWCOUNT暗黙カーソル属性
- 6-5　明示カーソルの宣言および定義

- 6-6　LOOP文内部のFETCH文
- 6-7　異なる変数への同じ明示カーソルのフェッチ
- 6-8　明示カーソル問合せの変数(結果セットの変更なし)
- 6-9　明示カーソル問合せの変数(結果セットの変更あり)
- 6-10　仮想列に別名が必要な明示カーソル
- 6-11　パラメータを受け入れる明示カーソル
- 6-12　デフォルト値を持つカーソル・パラメータ
- 6-13　既存のカーソルへの仮パラメータの追加
- 6-14　%ISOPEN明示カーソル属性
- 6-15　%FOUND明示カーソル属性
- 6-16　%NOTFOUND明示カーソル属性
- 6-17　%ROWCOUNT明示カーソル属性
- 6-18　暗黙カーソルFOR LOOP文
- 6-19　明示カーソルFOR LOOP文
- 6-20　明示カーソルFOR LOOP文へのパラメータの引渡し
- 6-21　カーソルFOR LOOPによる仮想列の参照
- 6-22　親問合せのFROM句内の副問合せ
- 6-23　相関副問合せ
- 6-24　カーソル変数の宣言
- 6-25　ユーザー定義の戻り型を持つカーソル変数
- 6-26　カーソル変数によるデータのフェッチ
- 6-27　カーソル変数からコレクションへのフェッチ
- 6-28　カーソル変数問合せの変数(結果セットの変更なし)
- 6-29　カーソル変数問合せの変数(結果セットの変更あり)
- 6-30　静的SQLを使用したコレクションの問合せ
- 6-31　1つの問合せ用にカーソル変数をオープンするプロシージャ
- 6-32　選択された問合せ用のカーソル変数のオープン(同じ戻り型)
- 6-33　選択された問合せ用のカーソル変数のオープン(異なる戻り型)
- 6-34　Pro*Cクライアント・プログラムにおけるホスト変数としてのカーソル変数
- 6-35　CURSOR式
- 6-36　COMMENT句およびWRITE句を使用するCOMMIT文
- 6-37　ROLLBACK文
- 6-38　SAVEPOINT文およびROLLBACK文
- 6-39　ROLLBACKでのSAVEPOINTの再使用
- 6-40　読取り専用トランザクションでのSET TRANSACTION文
- 6-41　COMMIT文の後でのFOR UPDATEカーソルによるフェッチ
- 6-42　ROWID擬似列を使用したCURRENT OF句のシミュレート
- 6-43　パッケージでの自律型ファンクションの宣言
- 6-44　自律型スタンドアロン・プロシージャの宣言
- 6-45　自律型PL/SQLブロックの宣言
- 6-46　自律型トリガーによるINSERT文のログへの記録
- 6-47　自律型トリガーによるDDLに対するシステム固有の動的SQLの使用
- 6-48　自律型ファンクションの起動

- 7-1 動的PL/SQLブロックからのサブプログラムの起動
- 7-2 BOOLEAN仮パラメータを持つサブプログラムの動的起動
- 7-3 RECORD仮パラメータを持つサブプログラムの動的起動
- 7-4 連想配列仮パラメータを持つサブプログラムの動的起動
- 7-5 ネストした表仮パラメータを持つサブプログラムの動的起動
- 7-6 VARRAY仮パラメータを持つサブプログラムの動的起動
- 7-7 USING句でのNULLを表現する未初期化変数
- 7-8 OPEN FOR文、FETCH文およびCLOSE文を使用したシステム固有の動的SQL
- 7-9 システム固有の動的SQLを使用したコレクションの問合せ
- 7-10 動的PL/SQLブロックで繰り返されるプレースホルダ名
- 7-11 DBMS_SQL.RETURN_RESULTプロシージャ
- 7-12 DBMS_SQL.GET_NEXT_RESULTプロシージャ
- 7-13 DBMS_SQLパッケージからシステム固有の動的SQLへの切替え
- 7-14 システム固有の動的SQLからDBMS_SQLパッケージへの切替え
- 7-15 SQLインジェクションの例のための設定
- 7-16 文の変更に対して脆弱なプロシージャ
- 7-17 文のインジェクションに対して脆弱なプロシージャ
- 7-18 データ型変換によるSQLインジェクションに対して脆弱なプロシージャ
- 7-19 SQLインジェクションを回避するためのバインド変数
- 7-20 SQLインジェクションを回避するための妥当性チェック
- 7-21 SQLインジェクションを回避するための明示的な書式モデル
- 8-1 単純なPL/SQLプロシージャの宣言、定義および起動
- 8-2 単純なPL/SQLファンクションの宣言、定義および起動
- 8-3 ファンクション内にあるRETURN文の後での実行の再開
- 8-4 RETURN文へ導かれない実行パスを含むファンクション
- 8-5 すべての実行パスがRETURN文へ導かれるファンクション
- 8-6 プロシージャ内にあるRETURN文の後での実行の再開
- 8-7 無名ブロック内にあるRETURN文の後での実行の再開
- 8-8 相互に起動し合うネストしたサブプログラム
- 8-9 仮パラメータと実パラメータ
- 8-10 サブタイプからNOT NULLのみを継承する実パラメータ
- 8-11 サブタイプから範囲のみを継承する実パラメータと戻り値
- 8-12 ファンクションによる仮パラメータから制約付きサブタイプへの暗黙的な変換
- 8-13 実パラメータの暗黙的な変換の回避
- 8-14 プロシージャの起動前、起動中および起動後のパラメータ値
- 8-15 例外処理後のOUTおよびIN OUTパラメータの値
- 8-16 NULL以外のデフォルト値を持つレコード型のOUT仮パラメータ
- 8-17 実パラメータとしてのグローバル変数によるエイリアシング
- 8-18 複数の仮パラメータに対する同じ実パラメータによるエイリアシング
- 8-19 カーソル変数サブプログラム・パラメータによるエイリアシング
- 8-20 デフォルトのパラメータ値を使用したプロシージャ
- 8-21 ファンクションによるデフォルトのパラメータ値の提供
- 8-22 既存の起動を変更しないサブプログラム・パラメータの追加

- 8-23　無名ブロックでの異なる表記法を使用した同等な起動
- 8-24　SELECT文での異なる表記法を使用した同等な起動
- 8-25　PL/SQLプロシージャ名の解決
- 8-26　オーバーロードされたサブプログラム
- 8-27　コンパイル時エラーの原因となるオーバーロード・エラー
- 8-28　正常にコンパイルが行われるオーバーロード・エラー
- 8-29　コンパイル時エラーの原因となる例8-28のサブプログラムの起動
- 8-30　例8-28のオーバーロード・エラーの修正
- 8-31　例8-30のサブプログラムの起動
- 8-32　オーバーロード・エラーが含まれていないパッケージ仕様部
- 8-33　適切にオーバーロードされたサブプログラムの不適切な起動
- 8-34　パラメータの暗黙的な変換によるオーバーロード・エラー
- 8-35　nの階乗(n!)を戻す再帰ファンクション
- 8-36　n番目のフィボナッチ数を戻す再帰ファンクション
- 8-37　結果がキャッシュされるファンクションの宣言および定義
- 8-38　構成パラメータの設定を戻す、結果がキャッシュされるファンクション
- 8-39　結果がキャッシュされるファンクションによるセッション固有の設定の処理
- 8-40　結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理
- 8-41　一度に1つの名前のキャッシュ(粒度が細かい)
- 8-42　一度に1つの言語の変換された名前のキャッシュ(粒度が粗い)
- 8-43　DRユニット内のデータベース・リンク
- 8-44　PL/SQL無名ブロックによる外部プロシージャの起動
- 8-45　PL/SQLスタンドアロン・プロシージャによる外部プロシージャの起動
- 9-1　トリガーによる条件述語を使用したトリガーを起動する文の検出
- 9-2　INSTEAD OFトリガー
- 9-3　ビューのネストした表の列に対するINSTEAD OFトリガー
- 9-4　複合トリガーによる、別の表にある表に対する変更のログへの記録
- 9-5　複合トリガーによる変更表エラーの回避
- 9-6　子表に対する外部キー・トリガー
- 9-7　親表に対するUPDATEおよびDELETE RESTRICTトリガー
- 9-8　親表に対するUPDATEおよびDELETE SET NULLトリガー
- 9-9　親表に対するDELETE CASCADEトリガー
- 9-10　親表に対するUPDATE CASCADEトリガー
- 9-11　トリガーによる複雑な制約のチェック
- 9-12　トリガーによるセキュリティ認可の施行
- 9-13　トリガーによる新しい列値の導出
- 9-14　トリガーによるEMPLOYEES.SALARYの変更の記録
- 9-15　条件付きトリガーによる給与変更情報の出力
- 9-16　トリガーによるCLOB列の変更
- 9-17　REFERENCING句のあるトリガー
- 9-18　トリガーによるOBJECT_VALUE疑似列の参照
- 9-19　サンプル・スキーマHRに対するBEFORE文トリガー
- 9-20　データベースに対するAFTER文トリガー

- 9-21　トリガーによるログオンの監視
- 9-22　スキーマに対するINSTEAD OF CREATEトリガー
- 9-23　トリガーによるJavaサブプログラムの起動
- 9-24　リモート・データベースが使用できない場合に例外を処理できないトリガー
- 9-25　例9-24の回避策
- 9-26　トリガーによる変更表エラー
- 9-27　更新カスケード
- 9-28　トリガーに関する情報の表示
- 10-1　単純なパッケージ仕様部
- 10-2　スタンドアロン・サブプログラムへの結合配列の受渡し
- 10-3　パッケージの仕様部と本体の一致
- 10-4　SERIALLY_REUSABLEパッケージの作成
- 10-5　SERIALLY_REUSABLEプラグマの影響
- 10-6　コール境界でオープンされるSERIALLY_REUSABLEパッケージのカーソル
- 10-7　パッケージでのカーソル宣言と定義の分離
- 10-8　ACCESSIBLE BY句
- 10-9　emp_adminパッケージの作成
- 11-1　PLSQL_WARNINGSコンパイル・パラメータの値の設定
- 11-2　DBMS_WARNINGサブプログラムを使用したPLSQL_WARNINGSの表示と設定
- 11-3　複数の例外に対応した1つの例外ハンドラ
- 11-4　例外ハンドラを共有する文のロケータ変数
- 11-5　内部的に定義された例外への名前付け
- 11-6　ZERO_DIVIDEを処理する無名ブロック
- 11-7　ZERO_DIVIDEを回避する無名ブロック
- 11-8　ROWTYPE_MISMATCHを処理する無名ブロック
- 11-9　再宣言された事前定義の識別子
- 11-10　ユーザー定義の例外の宣言、呼出しおよび処理
- 11-11　事前定義の例外の明示的な呼出し
- 11-12　例外の再呼出し
- 11-13　RAISE_APPLICATION_ERRORによるユーザー定義の例外の呼出し
- 11-14　有効範囲を超えて伝播した例外が処理される場合
- 11-15　有効範囲を超えて伝播した例外が処理されない場合
- 11-16　宣言の中で呼び出された例外が処理されない場合
- 11-17　宣言の中で呼び出された例外が外側のブロックで処理される場合
- 11-18　例外ハンドラの中で呼び出された例外が処理されない場合
- 11-19　例外ハンドラの中で呼び出された例外が起動元で処理される場合
- 11-20　例外ハンドラの中で呼び出された例外が外側のブロックで処理される場合
- 11-21　例外ハンドラの中で呼び出された例外が処理されない場合
- 11-22　例外ハンドラの中で呼び出された例外が外側のブロックで処理される場合
- 11-23　SQLCODEおよびSQLERRMの値の表示
- 11-24　例外ハンドラの実行と実行の終了
- 11-25　例外ハンドラの実行と実行の継続
- 11-26　例外処理後のトランザクションの再試行

- 12-1　サブプログラムのインライン化の指定
- 12-2　オーバーロードされたサブプログラムのインライン化の指定
- 12-3　サブプログラムの非インライン化の指定
- 12-4　PRAGMA INLINE ... 'NO'によるPRAGMA INLINE ... 'YES'のオーバーライド
- 12-5　ネストした問合せによるパフォーマンスの改善
- 12-6　NOCOPYサブプログラム・パラメータ
- 12-7　FOR LOOP文のDELETE文
- 12-8　FORALL文のDELETE文
- 12-9　FOR LOOP文およびFORALL文におけるINSERT文の時間の違い
- 12-10　コレクションのサブセットに対するFORALL文
- 12-11　疎コレクションおよびそのサブセットに対するFORALL文
- 12-12　FORALL例外の即時処理
- 12-13　FORALL文が完了した後のFORALL例外の処理
- 12-14　FORALLの各DELETEで影響を受けた行の数の表示
- 12-15　FORALLの各INSERT SELECTで影響を受けた行の数の表示
- 12-16　2つのネストした表に対するデータベースの2つの列のバルク選択
- 12-17　レコードのネストした表に対するバルク選択
- 12-18　予期しない結果が発生するSELECT BULK COLLECT INTO文
- 12-19　カーソルによる例12-18の回避策
- 12-20　2番目のコレクションによる例12-18の回避策
- 12-21　ROWNUM、SAMPLEおよびFETCH FIRSTによるバルク選択の制限
- 12-22　2つのネストした表へのバルク・フェッチ
- 12-23　レコードのネストした表へのバルク・フェッチ
- 12-24　LIMITによるバルク・フェッチの制限
- 12-25　2つのネストした表に対する削除行の戻し
- 12-26　FORALL文でのRETURN BULK COLLECT INTO付きのDELETE
- 12-27　FOR LOOP文でのRETURN BULK COLLECT INTO付きのDELETE
- 12-28　無名ブロックによる入力ホスト配列のバルク・バインド
- 12-29　パイプライン・テーブル・ファンクションの作成および起動
- 12-30　パイプライン・テーブル・ファンクションによる各行の2つの行への変換
- 12-31　パイプライン・テーブル・ファンクションの結果からのフェッチ
- 12-32　2つのカーソル変数パラメータを使用するパイプライン・テーブル・ファンクション
- 12-33　集計ファンクションとしてのパイプライン・テーブル・ファンクション
- 12-34　パイプライン・テーブル・ファンクションで処理されないNO_DATA_NEEDED
- 12-35　パイプライン・テーブル・ファンクションで処理されるNO_DATA_NEEDED
- 12-36　Skip_col多相テーブル・ファンクションの例
- 12-37　To_doc多相テーブル・ファンクションの例
- 12-38　Implicit_echo多相表関数の例
- 13-1　同じスキーマの最上位プロシージャへのアクセスの制限
- 13-2　任意のユニット名へのアクセスの制限
- 13-3　ストアド・プロシージャへのアクセスの制限
- 13-4　外部ファンクションの例
- 13-5　単一の基本ブロックにカバレッジのテスト実行不可能のマークを付ける

- 13-6　行範囲にカバレッジのテスト実行不可能のマークを付ける
- 13-7　ユニット全体または個別のサブプログラムにカバレッジのテスト実行不可能のマークを付ける
- 13-8　内部プログラムにカバレッジのテスト実行不可能のマークを付ける
- 13-9　非推奨化の警告の有効化
- 13-10　PL/SQLパッケージの非推奨化
- 13-11　カスタム警告によるPL/SQLパッケージの非推奨化
- 13-12　PL/SQLプロシージャの非推奨化
- 13-13　オーバーロードされたプロシージャの非推奨化
- 13-14　定数と例外の非推奨化
- 13-15　条件付きコンパイルを使用した一部のデータベース・リリースでのエンティティの非推奨化
- 13-16　オブジェクト型の非推奨化
- 13-17　オブジェクト型仕様部のメンバー・ファンクションの非推奨化
- 13-18　継承されたオブジェクト型の非推奨化
- 13-19　最上位のサブプログラムのみに適用される非推奨化
- 13-20　DEPRECATEプラグマの置き間違い
- 13-21　要素名とDEPRECATEプラグマの引数の不一致
- 13-22　Budget: 表式でのSQLマクロの使用
- 13-23　Take: 多相ビューでのTableマクロの使用
- 14-1　ファンクションの再コンパイル
- 14-2　ライブラリの再コンパイル
- 14-3　パッケージの再コンパイル
- 14-4　プロシージャの再コンパイル
- 14-5　トリガーの無効化
- 14-6　トリガーの有効化
- 14-7　メンバー・ファンクションの追加
- 14-8　コレクション属性の追加
- 14-9　コレクション型の要素の数の増加
- 14-10　コレクション型の長さの増加
- 14-11　型の再コンパイル
- 14-12　型仕様部の再コンパイル
- 14-13　ADTの進化と再設定
- 14-14　ファンクションの作成
- 14-15　集計ファンクションの作成
- 14-16　ファンクションでのパッケージ・プロシージャ
- 14-17　ライブラリの作成
- 14-18　外部プロシージャ・エージェントの指定
- 14-19　emp_mgmtパッケージの仕様部の作成
- 14-20　emp_mgmtパッケージ本体の作成
- 14-21　プロシージャの作成
- 14-22　外部プロシージャの作成
- 14-23　ADTの例
- 14-24　サブタイプの作成
- 14-25　型階層の作成

- 14-26　VARRAY型の作成
- 14-27　ネストした表型の作成
- 14-28　VARRAYを含むネストした表型の作成
- 14-29　コンストラクタの例
- 14-30　メンバー・メソッドの作成
- 14-31　静的メソッドの作成
- 14-32　ファンクションの削除
- 14-33　ライブラリの削除
- 14-34　パッケージの削除
- 14-35　プロシージャの削除
- 14-36　トリガーの削除
- 14-37　ADTの削除
- 14-38　ADT本体の削除
- A-1　2つのラップ可能PL/SQLユニットを含むSQLファイル
- A-2　PL/SQLラッパー・ユーティリティによるファイルのラップ
- A-3　ラップされたファイルの実行およびラップされたPL/SQLユニットの表示
- A-4　CREATE_WRAPPEDプロシージャによるラップされたパッケージ本体の作成
- A-5　ラップされた本体を含むパッケージの表示およびパッケージ・プロシージャの起動
- B-1　修飾名
- B-2　列名に解釈されるために意図していない結果を招く変数名
- B-3　異なる変数名の使用による例B-2の修正
- B-4　ブロック・ラベルの使用による例B-2の修正
- B-5　名前解決でのサブプログラム名
- B-6　列の参照の内部取得
- B-7　属性の参照の内部取得
- B-8　ADT属性の参照の修飾
- B-9　行の式への参照の修飾

図一覧

- 1-1　PL/SQLエンジン
- 5-1　7個の要素を持つ最大サイズが10個のVARRAY
- 5-2　配列およびネストした表
- 6-1　トランザクション制御の流れ
- 8-1　PL/SQLコンパイラによる起動の解決方法
- 11-1　伝播しない例外
- 11-2　内側のブロックから外側のブロックに伝播する例外
- 11-3　ホスト環境に未処理例外エラーを戻すPL/SQL

表一覧

- 1-1 PL/SQL I/O処理パッケージ
- 1-2 PL/SQLコンパイル・パラメータ
- 2-1 すべてのデータベース文字セットに含まれている記号文字
- 2-2 PL/SQLのデリミタ
- 2-3 演算子の優先順位
- 2-4 論理真理値表
- 2-5 関係演算子
- 2-6 静的な式で許可される演算子
- 3-1 PL/SQLとSQLで最大サイズが異なるデータ型
- 3-2 事前定義のPL/SQLのBINARY_FLOATおよびBINARY_DOUBLEの定数
- 3-3 PLS_INTEGERデータ型の事前定義のサブタイプ
- 5-1 PL/SQLコレクション型
- 5-2 コレクション・メソッド
- 8-1 PL/SQLサブプログラムのパラメータ・モード
- 8-2 PL/SQLサブプログラムのパラメータ・モードの特性
- 8-3 PL/SQLの実パラメータの表記法
- 8-4 粒度が細かいキャッシュと粗いキャッシュ
- 9-1 条件述語
- 9-2 複合トリガーのタイミング部
- 9-3 参照整合性を保証する制約およびトリガー
- 9-4 OLDおよびNEW疑似レコードのフィールド値
- 9-5 システム定義のイベント属性
- 9-6 データベース・イベント・トリガー
- 9-7 クライアント・イベント・トリガー
- 11-1 コンパイル時の警告カテゴリ
- 11-2 例外のカテゴリ
- 11-3 PL/SQLの事前定義の例外
- 12-1 プロファイル・ツールおよびトレース・ツールの概要
- C-1 PL/SQLコンパイラの制限
- D-1 PL/SQLの予約語
- D-2 PL/SQLのキーワード

はじめに

『Oracle Database PL/SQL言語リファレンス』では、SQLを手続き型言語として機能拡張したオラクル社のPL/SQLの使用方法について説明します。

ここでのトピック

- 対象読者
- ドキュメントのアクセシビリティ
- 関連ドキュメント
- 表記規則
- 構文の説明

対象読者

『Oracle Database PL/SQL言語リファレンス』は、Oracle DatabaseまたはOracle TimesTen In-Memory DatabaseのいずれかでPL/SQLベースのアプリケーションの開発を行うすべてのユーザーを対象としています。次に例を示します。

- プログラマ
- システム・アナリスト
- プロジェクト・マネージャ
- データベース管理者

このマニュアルを使用するには、次に関する実践的な知識が必要です。

- Oracle Database
- Structured Query Language(SQL)
- IF-THEN文、ループ、プロシージャ、ファンクションなどの基本的なプログラミングの概念

ドキュメントのアクセシビリティ

Oracleのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWebサイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracleサポートへのアクセス

サポートを購入したオラクル社のお客様は、My Oracle Supportを介して電子的なサポートにアクセスできます。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info)か、聴覚に障害のあるお客様は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

関連ドキュメント

詳細は、Oracle Database ドキュメント・セットの次のドキュメントを参照してください。

- Oracle Database SQL言語リファレンス
- Oracle Database PL/SQLパッケージおよびタイプ・リファレンス
- Oracle Database JSON開発者ガイド
- Oracle Database PL/SQLのためのSODA開発者ガイド
- Oracle Database開発ガイド
- Oracle Database管理者ガイド
- Oracle Database SecureFilesおよびラージ・オブジェクト開発者ガイド
- Oracle Databaseオブジェクト・リレーショナル開発者ガイド
- Oracle Database概要

  Oracle Databaseサンプル・スキーマ

関連項目:

http://www.oracle.com/technetwork/database/features/plsql/index.html

表記規則

このマニュアルでは次の表記規則を使用します。

規則	意味
太字	太字は、操作に関連する Graphical User Interface 要素、または本文中で定義されている用語および用語集に記載されている用語を示します。
イタリック体	イタリックは、ユーザーが特定の値を指定するプレースホルダ変数を示します。
固定幅フォント	固定幅フォントは、段落内のコマンド、URL、サンプル内のコード、画面に表示されるテキスト、または入力するテキストを示します。
{A|B|C}	A、B または C のいずれかを選択します。

付記:

- *_viewは、名前がviewで終わるすべての静的データ・ディクショナリ・ビューを示します。たとえば、*_ERRORSは、ALL_ERRORS、DBA_ERRORSおよびUSER_ERRORSを示します。静的データ・ディクショナリ・ビューの詳細または概要は、『Oracle Databaseリファレンス』を参照してください。

- スキーマ名で修飾されていない表名は、サンプル・スキーマHRにあります。サンプル・スキーマの詳細は、『Oracle Databaseサンプル・スキーマ』を参照してください。

構文の説明

このマニュアルには、様々なSQL、PL/SQLまたはその他のコマンドライン構造体の構文の説明が、図形式またはバッカス正規形(BNF)で記載されています。これらの説明の解釈方法は、『Oracle Database SQL言語リファレンス』を参照してください。

『Oracle Database PL/SQL言語リファレンス』のこのリリースの変更点

『Oracle Database PL/SQL言語リファレンス』に対する変更は、リリースごとに記載されています。

この章の内容は次のとおりです。

- 『Oracle Database PL/SQL言語リファレンス』のリリース19cの変更点
- 『Oracle Database PL/SQL言語リファレンス』のリリース18cの変更点

Oracle Databaseリリース19cにおける変更点

Oracle Database 19c以降では、多相表関数とその実装メソッドが同じパッケージで定義されていると、USING句がオプションになります。

関連項目:

- 多相表関数、構文およびセマンティクスの詳細は、「PIPELINED句」を参照

SQLマクロ

SQLマクロ(SQM)を作成して、共通のSQL式および文を、他のSQL文で使用できる再利用可能なパラメータ化された構造体にくくり出すことができます。

Oracle Databaseリリース19c, バージョン19.7以降、SQL表マクロがサポートされています。SQL表マクロは、通常FROM句で使用される式で、多相(パラメータ化)ビューのように機能します。

SQL表マクロにより、開発者の生産性が高まり、コラボレーション開発が簡略化され、コード品質が向上します。

関連項目:

- 構文およびセマンティクスの詳細は、CREATE FUNCTION文を参照してください
- SQL_MACRO句の構文およびセマンティクス

Oracle Databaseリリース18cでの変更点

Oracle Databaseの18cのOracle Database PL/SQL言語リファレンスでは、次の新機能について説明します。

新機能

オブジェクト型の非存続性のサポート

この機能ではプログラム内の抽象データ型にマーキングをして、その使用を処理や格納またはそれら両方の用途に限定できます。

永続不可型のインスタンスは、ディスク上で保持することはできません。

関連項目:

- 構文およびセマンティクスの詳細は、CREATE TYPE文の[NOT] PERSISTABLE句を参照してください。

多相テーブル・ファンクション

多相テーブル・ファンクション(PTF)は新しいタイプのテーブル・ファンクションで、PTFに渡された引数によって戻り型が決定されます。

テーブル・ファンクションは行のコレクションを返すファンクションで、SQL問合せブロックのFROM句から呼び出せます。

この新しい多相テーブル・ファンクション(PTF)は、Oracle Databaseの分析機能を拡張するための効率的でスケーラブルなフレームワークを提供します。問合せの記述者はPTFの実装の詳細がわからなくても、これらのファンクションを呼び出すことができ、PTF側でも、ファンクションがどのように実行されるか(シリアルまたはパラレル)や、入力行がパーティション化または順序付けされているかどうかなどを把握する必要がありません。

したがってPTFは、SQL開発者やデータベース管理者が、任意的な入力表または問合せに対応できる汎用的な拡張機能を提供する場合に便利です。

詳細は、多相テーブル・ファンクションの概要を参照してください。

修飾式

集計およびそれらに必要な付加詞である修飾式によって、プログラムが明確になりプログラマの生産性が向上します。

Oracle Databaseの12cリリース2では、非スカラー・データ型の値を式を使用して提供することができました(ネストした表の型コンストラクタを使用するなど)。Oracle Databaseリリース18cからは、コンストラクタが抽象データ型の値を提供するのと同じように、どんなPL/SQL値も(たとえばレコードや連想配列用に)式を使用して提供できます。PL/SQLでは"修飾式"および"集計"と呼び、SQL用語では"型コンストラクタ"と呼びますが、両者の機能は同じです。

構文およびセマンティクスの詳細は、修飾式の概要およびqualified_expression ::=を参照してください。

非推奨となった機能

次の機能は今回のリリースで非推奨となり、今後のリリースではサポートされない可能性があります。

コマンド ALTER TYPE ... INVALIDATEは非推奨となりました。かわりにCASCADE句を使用してください。

ALTER TYPEのREPLACE句は非推奨となりました。かわりにalter_method_spec句を使用してください。または、CREATE OR REPLACE TYPE文を使用して型を再作成することもできます。

構文およびセマンティクスの詳細は、ALTER TYPE文を参照してください

Oracle Databaseの12cリリース1(12.1)以降、コンパイル・パラメータPLSQL_DEBUGは非推奨となります。

PL/SQLユニットをデバッグ用にコンパイルするには、PLSQL_OPTIMIZE_LEVEL=1を指定します。

コンパイル・パラメータの詳細は、PL/SQLユニットおよびコンパイル・パラメータを参照してください。

サポート対象外機能

19cでサポート対象外になったPL/SQL言語リファレンスの機能はありません。

ノート:

Oracle Database アドバンスト・キューイングは、Oracle Database 19c でサポートが終了した Oracle Streams とは無関係に機能強化が継続されています。

関連項目:

- Oracle Database19cのサポート対象外機能の詳細は、『Oracle Databaseアップグレード・ガイド』を参照してください

1 PL/SQLの概要

PL/SQLは、SQLの手続き型拡張機能としてオラクル社が提供する言語であり、移植性を備えた高性能のトランザクション処理言語です。ここでは、そのメリットと、主な特徴およびアーキテクチャについて説明します。

ここでのトピック

- PL/SQLの利点
- PL/SQLの主な特長
- PL/SQLのアーキテクチャ

1.1 PL/SQLの利点

PL/SQLには、その他のプログラミング言語と比較したときに、いくつかのメリットがあります。

PL/SQLのメリットは、次のとおりです。

- SQLとの緊密な統合
- 高いパフォーマンス
- 高い生産性
- 移植性
- スケーラビリティ
- 管理性
- オブジェクト指向プログラミングのサポート

1.1.1 SQLとの緊密な統合

PL/SQLは、SQLと緊密に統合された、最も広く使用されているデータベース操作言語です。

たとえば:

- PL/SQLでは、SQLのデータ操作、カーソル制御およびトランザクション制御のすべての文、SQLのすべてのファンクション、演算子および擬似列を使用できます。
- PL/SQLでは、SQLデータ型を完全にサポートしています。

  PL/SQLとSQLのデータ型間で変換を行う必要はありません。たとえば、PL/SQLプログラムがSQL型VARCHAR2の列から値を取得する場合は、その値をVARCHAR2型のPL/SQL変数に格納できます。

  明示的にデータ型を指定することなく、PL/SQLデータ項目を、データベース表の列や行のデータ型にできます(「%TYPE属性の使用」および「%ROWTYPE属性の使用」を参照)。
- PL/SQLでは、SQL問合せを実行し、結果セットの行を一度に1行ずつ処理することができます(「問合せ結果セットの一度に1行ずつの処理」を参照)。
- PL/SQLファンクションは、SQL SELECT文のWITH句で宣言および定義できます(『Oracle Database SQL言語リ

ファレンス』を参照)。

PL/SQLは静的SQLおよび動的SQLの両方をサポートしています。静的SQLとは、コンパイル時にテキスト全体がわかるSQLのことです。動的SQLとは、実行時までテキスト全体が不明なSQLのことです。動的SQLを使用すると、アプリケーションをより柔軟で多目的に使用できます。詳細は、「PL/SQLの静的SQL」および「PL/SQLの動的SQL」を参照してください。

1.1.2 高いパフォーマンス

PL/SQLでは、複数文のブロックをデータベースに送信して、アプリケーションとデータベースの間の通信量を大幅に削減できます。

バインド変数

PL/SQLコードにSQLのINSERT、UPDATE、DELETE、MERGEまたはSELECT文を直接埋め込むと、PL/SQLコンパイラは、WHERE句およびVALUES句内の変数をバインド変数に変換します(詳細は、「静的SQL文における名前解決」を参照)。これらのSQL文は、同じコードが実行されるたびにOracle Databaseによって再利用されるため、パフォーマンスが向上します。

バインド変数は、動的SQLの使用時にPL/SQLによって自動的には作成されませんが、明示的に指定することによって動的SQLとともに使用できます(詳細は、「EXECUTE IMMEDIATE文」を参照)。

サブプログラム

PL/SQLサブプログラムは実行可能な形式で格納され、繰り返し起動することができます。ストアド・サブプログラムはデータベース・サーバー内で実行されるため、ネットワークを介した1回の起動で大規模なジョブを開始できます。この作業の分割によってネットワークの通信量が軽減され、応答時間が短縮されます。ストアド・サブプログラムはキャッシュされ、ユーザー間で共有されるため、メモリー要件と起動のオーバーヘッドが低減します。サブプログラムの詳細は、「サブプログラム」を参照してください。

オプティマイザ

PL/SQLコンパイラには、コードのパフォーマンスが向上するように再調整する機能を持つオプティマイザがあります。オプティマイザの詳細は、「PL/SQLオプティマイザ」を参照してください。

1.1.3 高い生産性

PL/SQLは、設計およびデバッグ時間を削減するための多くの機能を備えていて、どの環境でも同様です。

PL/SQLでは、データを操作するコードをコンパクトにすることができます。Perlのようなスクリプト言語が、ファイルのデータを読み込み、変換して、書き込むのと同じように、PL/SQLは、データベースのデータを問い合せ、変換して、更新することができます。

あるOracleツールでPL/SQLの使用方法を学習すれば、その知識は他のOracleツールにも活用できます。PL/SQLの特長の概要は、「PL/SQLの主な特長」を参照してください。

1.1.4 移植性

PL/SQLは、移植性のあるOracle開発用の標準言語です。

PL/SQLアプリケーションは、Oracle Databaseが動作する任意のオペレーティング・システムおよびプラットフォームで実行できます。

1.1.5 スケーラビリティ

PL/SQLストアド・サブプログラムは、データベース・サーバー上のアプリケーション処理を集中化することでスケーラビリティを高めます。

共有サーバーの共有メモリー機能を使用すると、単一ノード上の何千もの同時ユーザーをOracle Databaseでサポートできるようになります。サブプログラムの詳細は、「サブプログラム」を参照してください

Oracle Connection Managerでネットワーク接続を多重化して、さらにスケーラビリティを高めることもできます。Oracle Connection Managerの詳細は、『Oracle Database Net Servicesリファレンス』を参照

1.1.6 管理性

PL/SQLストアド・サブプログラムでは、クライアント・システムごとにサブプログラムのコピーを1つずつ保持するのではなく、データベース・サーバー上にコピーを1つのみ保持することができるため、管理性が向上します。

サブプログラムは任意の数のアプリケーションで使用することができ、呼出し元のアプリケーションに影響を与えることなく、サブプログラムを変更できます。サブプログラムの詳細は、「サブプログラム」を参照してください。

1.1.7 オブジェクト指向プログラミングのサポート

PL/SQLでは、オブジェクト指向の設計に使用可能なオブジェクト型を定義できます。

PL/SQLは、「抽象データ型」を使用したオブジェクト指向のプログラミングをサポートします。

1.2 PL/SQLの主な特長

PL/SQLでは、SQLのデータ操作機能と手続き型言語の処理機能の両方が利用できます。

SQLを使用して問題が解決できる場合、PL/SQLプログラムからSQL文を発行できます。新しいAPIを学習する必要はありません。

その他の手続き型プログラム言語と同様に、PL/SQLでは、定数と変数の宣言、プログラム・フローの制御、サブプログラムの定義、およびランタイム・エラーのトラップを行うこともできます。

複雑な問題を容易に理解できるサブプログラムに分割し、複数のアプリケーションで再利用できます。

ここでのトピック

- エラー処理
- ブロック
- 変数および定数
- サブプログラム
- パッケージ
- トリガー

- 入出力
- データの抽象化
- 制御文
- 条件付きコンパイル
- 問合せ結果セットの一度に1行ずつの処理

1.2.1 エラー処理

PL/SQLでは、エラーを簡単に検出して処理できます。

エラーが発生するとPL/SQLによって例外が呼び出されます。通常の実行は中止され、PL/SQLブロックの例外処理部に制御が移ります。Cプログラムの場合のように、すべての操作をチェックして処理が成功したことを確認する必要はありません。

詳細は、「PL/SQLのエラー処理」を参照してください。

1.2.2 ブロック

PL/SQLソース・プログラムの基本単位はブロックで、関連する宣言および文をグループ化します。

PL/SQLブロックは、キーワードDECLARE、BEGIN、EXCEPTIONおよびENDで定義します。これらのキーワードは、ブロックを宣言部、実行部、例外処理部に分けます。このうち必ず存在する必要があるのは実行部のみです。ブロックにはラベルを付けられます。

宣言はブロックの中で局所的に有効で、そのブロックの実行が完了すると消滅します。これによって、変数およびサブプログラムの名前空間が一杯になることを回避できます。

ブロックはネストできます。ブロックは実行可能文であるため、実行可能文を配置できる場所であれば別のブロックでも配置できます。

ブロックは、(SQL*PlusやEnterprise Managerなどの)対話型ツールに送信するか、またはOracleプリコンパイラやOCIのプログラムに埋め込むことができます。対話型ツールまたはプログラムによって、ブロックが1回実行されます。ブロックはデータベースに格納されないため、(ラベルがある場合でも)無名ブロックと呼ばれます。

無名ブロックは、メモリーにロードされるたびにコンパイルされます。このコンパイルには、次の3段階があります。

1. 構文チェック: PL/SQL構文がチェックされ、解析ツリーが生成されます。
2. セマンティック・チェック: 型チェックおよび解析ツリーに対する追加処理が行われます。
3. コード生成

ノート:

無名ブロックは SQL 文です。

構文の詳細は、「ブロック」を参照してください。

例1-1 PL/SQLブロック構造

この例は、PL/SQLブロックの基本構造を示しています。

```
<< label >> (optional)
DECLARE    -- Declarative part (optional)
  -- Declarations of local types, variables, & subprograms
BEGIN      -- Executable part (required)
  -- Statements (which can use items declared in declarative part)
[EXCEPTION -- Exception-handling part (optional)
  -- Exception handlers for exceptions (errors) raised in executable part]
END;
```

1.2.3 変数および定数

PL/SQLでは変数と定数を宣言し、式を使用可能な任意の場所で使用できます。

プログラムの実行中に、変数の値を変更できますが、定数の値は変更できません。

詳細は、「宣言」および「変数への値の代入」を参照してください。

1.2.4 サブプログラム

PL/SQLのサブプログラムは、繰り返し起動できる名前付きPL/SQLブロックです。

サブプログラムにパラメータが含まれている場合は、起動のたびにパラメータ値を変えることができます。PL/SQLにはプロシージャとファンクションの2種類のサブプログラムがあります。ファンクションは結果を戻します。

PL/SQLサブプログラムの詳細は、「PL/SQLサブプログラム」を参照してください。

PL/SQLでは、他の言語で記述された外部プログラムも起動できます。

詳細は、「外部サブプログラム」を参照してください。

1.2.5 パッケージ

パッケージとは、論理的に関連するPL/SQLの型、変数、定数、サブプログラム、カーソルおよび例外をグループにまとめたスキーマ・オブジェクトのことです。

パッケージをコンパイルしてデータベースに格納し、その内容を複数のアプリケーションで共有できます。パッケージはアプリケーションと考えることができます。

ユーザー独自のパッケージを記述できます(詳細は、「PL/SQLパッケージ」を参照してください)。Oracle Databaseで提供されている製品固有の多数のパッケージを使用することもできます。詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

1.2.6 トリガー

トリガーとは、データベースに格納され、データベース内で発生したイベントに応答して実行される、名前付きPL/SQLユニットです。

トリガーをイベントの前に起動する場合でもイベントの後で起動する場合でも、またトリガーをイベントごとに実行する場合でもイ

ベントの影響を受ける行ごとに実行する場合でも、イベントを指定できます。たとえば、EMPLOYEES表がINSERT文の影響を受けるたびに実行されるトリガーを作成できます。

トリガーの詳細は、「PL/SQLトリガー」を参照してください。

1.2.7 入出力

PL/SQLのほとんどの入出力(I/O)は、SQL文によるデータベース表へのデータの格納や、それらの表の問合せによるものです。他のすべてのPL/SQL I/Oは、Oracle Databaseが提供するPL/SQLパッケージを使用して実行されます。

表1-1 PL/SQL I/O処理パッケージ

パッケージ	説明	参照
DBMS_OUTPUT	PL/SQL ブロック、サブプログラム、パッケージおよびトリガーからの出力を表示できます。PL/SQL デバッグ情報を表示する場合に特に役立ちます。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
HTF	HTML タグを生成するハイパーテキスト・ファンクションを含みます(たとえば、HTF.ANCHOR ファンクションでは HTML アンカー・タグ<A>が生成されます)。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
HTP	HTML タグを生成するハイパーテキスト・プロシージャを含みます。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
DBMS_PIPE	同じインスタンス内の複数のセッション間で通信できます。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
UTL_FILE	PL/SQL プログラムでオペレーティング・システムのファイルに対して読取りおよび書込みを実行できます。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
UTL_HTTP	PL/SQL プログラムで Hypertext Transfer Protocol(HTTP)コールアウトを実行し、HTTP 経由でインターネット上のデータにアクセスできます。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』
UTL_SMTP	RFC821 の仕様に従って、Simple Mail Transfer Protocol(SMTP)を介して電子メールを送信します。	『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』

DBMS_OUTPUTに渡される出力を表示するには、SQL*Plusなどの別のプログラムが必要です。SQL*PlusでDBMS_OUTPUTの出力を表示するには、事前に、SQL*PlusコマンドSET SERVEROUTPUT ONを発行しておく必要があります。

表1-1のパッケージに含まれるサブプログラムの一部は入力と出力表示の両方を受け入れますが、キーボードから入力されたデータを直接受け入れることはできません。キーボードから入力されたデータを直接受け入れるには、SQL*PlusコマンドPROMPTおよびACCEPTを使用します。

関連項目:

- SQL*PlusコマンドSET SERVEROUTPUT ONの詳細は、『SQL*Plusユーザーズ・ガイドおよびリファレンス』を参照してください。
- SQL*PlusコマンドPROMPTの詳細は、『SQL*Plusユーザーズ・ガイドおよびリファレンス』を参照してください。
- SQL*PlusコマンドACCEPTの詳細は、『SQL*Plusユーザーズ・ガイドおよびリファレンス』を参照してください。
- SQL文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

1.2.8 データの抽象化

データの抽象化によって、詳細な部分を必要以上に意識することなく、データの基本的なプロパティを操作できます。

初めにデータ構造を設計してから、データ構造を操作するアルゴリズムを設計できます。

ここでのトピック

- カーソル
- コンポジット変数
- %ROWTYPE属性の使用
- %TYPE属性の使用
- 抽象データ型

1.2.8.1 カーソル

カーソルとは、特定のSQL文やPL/SQLのSELECT INTO文を処理する情報を保存しておく、SQLのプライベート領域を指すポインタです。

カーソルを使用して、結果セットの行を一度に1行ずつ取り出すことができます。カーソル属性を使用して、カーソルの状態に関する情報(文によってこれまでに影響を受けた行数など)を取得できます。

カーソルの詳細は、「カーソルの概要」を参照してください。

1.2.8.2 コンポジット変数

コンポジット変数には内部コンポーネントがあり、個別にアクセスできます。

コンポジット変数全体をサブプログラムにパラメータとして渡すことができます。PL/SQLには、コレクションとレコードという、2種類のコンポジット変数があります。

コレクションの内部コンポーネントは、常に同じデータ型であり、要素と呼ばれます。各要素には、その一意の索引によってアクセスします。リストおよび配列は、コレクションの典型的な例です。

レコードの内部コンポーネントは、データ型が異なる場合があり、フィールドと呼ばれます。各フィールドには、その名前によってアクセスします。レコード変数には、表の1行または表の1行の列の一部を格納できます。

コンポジット変数の詳細は、「PL/SQLのコレクションおよびレコード」を参照してください。

1.2.8.3 %ROWTYPE属性の使用

%ROWTYPE属性を使用すると、データベースの表またはビュー内の行の全体または一部を表すレコードを宣言できます。

このレコードは、行の全体または一部のすべての列に対して、同じ名前とデータ型のフィールドを持ちます。行の構造が変更されると、それに応じてレコードの構造も変更されます。

%ROWTYPEの構文およびセマンティクスの詳細は、「%ROWTYPE属性」を参照してください。使用方法の詳細は、「%ROWTYPE属性を使用した項目の宣言」を参照してください。

1.2.8.4 %TYPE属性の使用

%TYPE属性を使用すると、事前に宣言されている変数または列と同じデータ型のデータ項目を(データ型を知らなくても)宣言できます。

参照先項目の宣言が変更されると、それに応じて参照元項目の宣言も変更されます。%TYPE属性は、データベースの値を保持する変数を宣言する場合に特に便利です。%TYPEの構文およびセマンティクスの詳細は、「%TYPE属性」を参照してください。使用方法の詳細は、「%TYPE属性を使用した項目の宣言」を参照してください。

1.2.8.5 抽象データ型

抽象データ型(ADT)は、データ構造と、データを操作するサブプログラムで構成されています。

データ構造を形成する変数は、属性と呼ばれます。属性を操作するサブプログラムは、メソッドと呼ばれます。

ADTはデータベースに格納されます。ADTのインスタンスは、表に格納したり、PL/SQL変数として使用することができます。

ADTにより、大規模なシステムを複数の再利用可能な論理コンポーネントに分割して、複雑さを軽減できます。

静的データ・ディクショナリ・ビュー*_OBJECTSでは、ADTのOBJECT_TYPEはTYPEです。静的データ・ディクショナリ・ビュー*_TYPESでは、ADTのTYPECODEはOBJECTです。

ADTの詳細は、「CREATE TYPE文」を参照してください。

ノート:

ADT は、ユーザー定義型およびオブジェクト型とも呼ばれます。

関連項目:

ADT(またはオブジェクト型)については、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

1.2.9 制御文

制御文は、SQLに対して加えられたPL/SQLの最も重要な機能拡張です。

PL/SQLには、制御文に3つのカテゴリがあります。

- 条件付き選択文。データ値に応じて、異なる文を実行できます。

  詳細は、「条件付き選択文」を参照してください。

- ループ文。一連の異なるデータ値を使用して、同じ文を繰り返し実行できます。

  詳細は、「LOOP文」を参照してください。

- 順次制御文。指定したラベル付きの文へ移動したり、または何もしないでおくことができます。

  詳細は、「順次制御文」を参照してください。

1.2.10 条件付きコンパイル

条件付きコンパイルによって、ソース・テキストを削除することなく、PL/SQLアプリケーションの機能をカスタマイズできます。

たとえば、次のことが可能です。

- 最新のデータベース・リリースで新機能を使用し、古いデータベース・リリースでアプリケーションを実行する場合にそれらの新機能を無効化することができます。
- 開発環境でデバッグ文またはトレース文をアクティブ化し、本番サイトでアプリケーションを実行する場合にそれらの文を隠ぺいすることができます。

詳細は、「条件付きコンパイル」を参照してください。

1.2.11 問合せ結果セットの一度に1行ずつの処理

PL/SQLでは、SQL問合せを発行し、結果セットの行を一度に1行ずつ処理できます。

基本ループを使用するか、または問合せの実行、結果の取出しおよび処理の終了を行う個別の文を使用して処理を正確に制御できます。

例1-2 問合せ結果行の一度に1行ずつの処理

この例では、基本ループを使用します。

```
BEGIN
  FOR someone IN (
    SELECT * FROM employees
    WHERE employee_id < 120
    ORDER BY employee_id
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('First name = ' || someone.first_name ||
                         ', Last name = ' || someone.last_name);
  END LOOP;
END;
/
```

結果:

```
First name = Steven, Last name = King
First name = Neena, Last name = Kochhar
```

```
First name = Lex, Last name = De Haan
First name = Alexander, Last name = Hunold
First name = Bruce, Last name = Ernst
First name = David, Last name = Austin
First name = Valli, Last name = Pataballa
First name = Diana, Last name = Lorentz
First name = Nancy, Last name = Greenberg
First name = Daniel, Last name = Faviet
First name = John, Last name = Chen
First name = Ismael, Last name = Sciarra
First name = Jose Manuel, Last name = Urman
First name = Luis, Last name = Popp
First name = Den, Last name = Raphaely
First name = Alexander, Last name = Khoo
First name = Shelli, Last name = Baida
First name = Sigal, Last name = Tobias
First name = Guy, Last name = Himuro
First name = Karen, Last name = Colmenares
```

1.3 PL/SQLのアーキテクチャ

PL/SQLのアーキテクチャに関して基本的に理解することは、PL/SQLのプログラマの役に立ちます。

ここでのトピック

- PL/SQLエンジン
- PL/SQLユニットおよびコンパイル・パラメータ

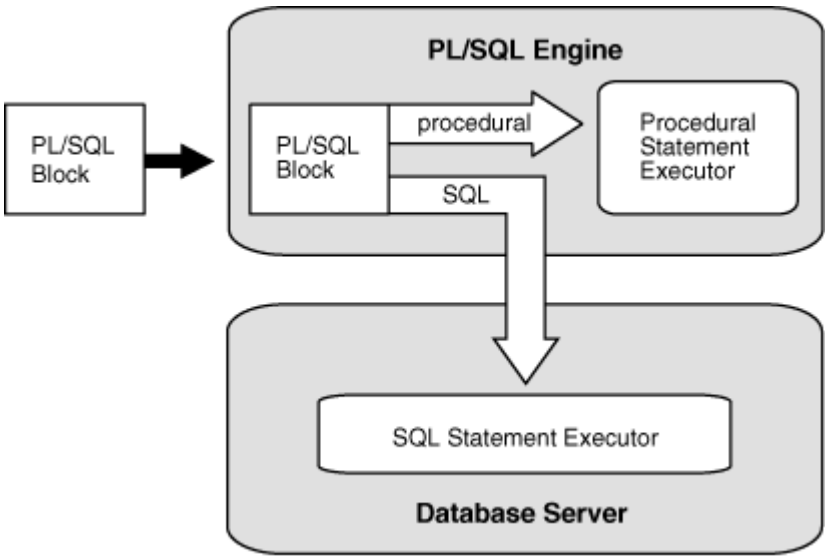
1.3.1 PL/SQLエンジン

PL/SQLコンパイルおよびランタイム・システムは、PL/SQLユニットをコンパイルして実行するエンジンです。

このエンジンは、データベースにインストールすることも、Oracle Formsのようなアプリケーション開発ツールにインストールすることもできます。

どちらの環境でも、PL/SQLエンジンは任意の適切なPL/SQLユニットを入力として受け入れます。図1-1に示すように、SQLエンジンはプロシージャ文のみを実行し、SQL文をデータベースのSQLエンジンに送信します。

図1-1 PL/SQLエンジン



通常、PL/SQLユニットはデータベースによって処理されます。

アプリケーション開発ツールは、PL/SQLユニットを処理する際に、それらをローカルのPL/SQLエンジンに渡します。PL/SQLユニットにSQL文がない場合、ローカルのエンジンがPL/SQLユニット全体を処理します。アプリケーション開発ツールが条件制御や反復制御を活用できる場合は、この機能が特に便利です。

たとえば、Oracle Formsアプリケーションは、フィールド・エントリの値のテストや単純な計算のためにSQL文を頻繁に使用します。SQLのかわりにPL/SQLを使用すると、これらのアプリケーションはデータベースへのコールを回避できます。

1.3.2 PL/SQLユニットおよびコンパイル・パラメータ

PL/SQLユニットは、PL/SQLコンパイル・パラメータ(データベース初期化パラメータのカテゴリ)によって影響を受けます。異なるPL/SQLユニット(パッケージの仕様部や本体など)に、異なるコンパイル・パラメータ設定を含めることができます。

PL/SQLユニットは、次のいずれかです。

- PL/SQLの無名ブロック
- FUNCTION
- LIBRARY
- PACKAGE
- PACKAGE BODY
- PROCEDURE
- TRIGGER
- TYPE
- TYPE BODY

表1-2に、PL/SQLコンパイル・パラメータの概要を示します。指定したPL/SQLユニットまたはすべてのPL/SQLユニットについて、これらのパラメータの値を表示するには、静的データ・ディクショナリ・ビューALL_PLSQL_OBJECT_SETTINGSを問い合せます。このビューの詳細は、『Oracle Databaseリファレンス』を参照してください。

表1-2 PL/SQLコンパイル・パラメータ

パラメータ	説明

パラメータ	説明
PLSCOPE_SETTINGS	コンパイル時のコレクション、相互参照、および PL/SQL ソース・テキストの識別子データの格納を制御します。PL/Scope ツールにより使用されます(『Oracle Database 開発ガイド』を参照)。 PLSCOPE_SETTINGS の詳細は、『Oracle Database リファレンス』を参照してください。
PLSQL_CCFLAGS	各 PL/SQL ユニットの条件付きコンパイルを独立して制御できます。 PLSQL_CCFLAGS の詳細は、「条件付きコンパイルの動作方法」および『Oracle Database リファレンス』を参照してください。
PLSQL_CODE_TYPE	PL/SQL ユニットのコンパイル・モード INTERPRETED(デフォルト)または NATIVE を指定します。どちらのモードを使用するかについては、「PL/SQL のネイティブ・コンパイルを使用するかどうかの判断」を参照してください。 (PLSQL_OPTIMIZE_LEVEL によって設定された)最適化レベルが 2 より小さい場合: ● コンパイラは、PLSQL_CODE_TYPE に関係なく、解釈済コードを生成します。 ● NATIVE を指定した場合、コンパイラは NATIVE が無視されたことを警告します。 PLSQL_CODE_TYPE の詳細は、『Oracle Database リファレンス』を参照してください。
PLSQL_OPTIMIZE_LEVEL	PL/SQL ユニットのコンパイルで使用される最適化レベルを指定します(レベルを高くすると、コンパイラでより多くの PL/SQL ユニットの最適化が試行されます)。 PLSQL_OPTIMIZE_LEVEL=1 で、PL/SQL デバッガで使用するためのコードを生成して格納するように PL/SQL コンパイラに指示できます。 PLSQL_OPTIMIZE_LEVEL の詳細は、「PL/SQL オプティマイザ」および『Oracle Database リファレンス』を参照してください。
PLSQL_WARNINGS	PL/SQL コンパイラによる警告メッセージのレポートを有効または無効にし、エラーとして表示する警告メッセージを指定します。 PLSQL_WARNINGS の詳細は、「コンパイル時の警告」および『Oracle Database リファレンス』を参照してください。
NLS_LENGTH_SEMANTICS	バイト長セマンティクスまたは文字長セマンティクスのいずれかを使用して、CHAR および

パラメータ	説明
	VARCHAR2 列を作成できます。 バイト長セマンティクスおよび文字長セマンティクスの詳細は、「CHAR および VARCHAR2 変数」を参照してください。 NLS_LENGTH_SEMANTICS の詳細は、『Oracle Database リファレンス』を参照してください。
PERMIT_92_WRAP_FORMAT	9.2 PL/SQL コンパイラでコンパイルされているラップされたパッケージを 12.1 PL/SQL コンパイラが使用できるかどうかを指定します。デフォルト値は TRUE です。 ラップされたパッケージの詳細は、「PL/SQL のソース・テキストのラップ」を参照してください。 PERMIT_92_WRAP_FORMAT の詳細は、『Oracle Database リファレンス』を参照してください。

ノート:

PL/SQL ユニットをデバッグ用にコンパイルするかどうかを指定するコンパイル・パラメータ PLSQL_DEBUG は、非推奨になりました。PL/SQL ユニットをデバッグ用にコンパイルするには、PLSQL_OPTIMIZE_LEVEL=1 を指定します。

表1-2のパラメータのコンパイル時の値が各ストアドPL/SQLユニットのメタデータとともに格納されており、ユニットを明示的に再コンパイルする際にその値を再利用できます。(ストアドPL/SQLユニットは、「CREATE [OR REPLACE]文」の1つを使用して作成します。無名ブロックは、ストアドPL/SQLユニットではありません。)

ストアドPL/SQLユニットを明示的に再コンパイルして、そのパラメータ値を再利用するには、ALTER文をCOMPILE句およびREUSE SETTINGS句とともに使用する必要があります。すべてのALTER文にこの句があります。ALTER文の一覧は、「ALTER文」を参照してください。

1.4 PL/SQLでの機密情報の保護

アプリケーション開発では、データ・セキュリティを最優先する必要があります。PL/SQLの使用中に脆弱性のリスクを軽減する方法はいくつかあります。

PL/SQLブロックのすべての内容が監査ログやトレース・ファイルなどの場所に書き込まれる可能性があることに注意してください。同様に、USER_SOURCEなどのディクショナリ・ビューからストアド・プロシージャにアクセスできます。このため、PL/SQLコードに表示されるリテラルには機密情報を含めないことをお薦めします。

バインド変数はSQLインジェクション攻撃から保護に役立つ場合がありますが、トレース・ファイル、監査、V$SQLおよび関連ビューなどの場所でバインド値を表示できます。アクセス権を厳密に管理して、必要なユーザーのみに、特に機密性が高いこの

情報を表示する権限を付与してください。バインド変数の使用の詳細は、バインド変数を参照してください。

2 PL/SQL言語の基礎

PL/SQL言語の基本コンポーネントについて説明します。

- 文字セット
- 字句単位
- 宣言
- 識別子に対する参照
- 識別子の有効範囲と可視性
- 変数への値の代入
- 式
- エラー・レポート・ファンクション
- 条件付きコンパイル

2.1 文字セット

PL/SQLで処理したりデータベースに格納する文字データはすべて、一連のバイトで表現する必要があります。単一の文字をバイトで表現したものを文字コードと呼びます。文字コードのセットを文字セットと呼びます。

すべてのOracle Databaseでは、データベース文字セットと各国語文字セットがサポートされています。また、PL/SQLでも同様に、これらの文字セットがサポートされています。このマニュアルでは、データベース文字セットおよび各国語文字セットをPL/SQLで使用する方法を説明します。

ここでのトピック

- データベース文字セット
- 各国語文字セット
- データ・バインドされた照合について

関連項目:

文字セットの概要は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

2.1.1 データベース文字セット

PL/SQLでは、データベース文字セットを使用して次のものを表現します。

- PL/SQLユニットの格納済ソース・テキスト

  PL/SQLユニットの詳細は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください。
- CHAR、VARCHAR2、CLOBおよびLONGの各データ型の文字値

これらのデータ型の詳細は、「SQLデータ型」を参照してください。

データベース文字セットは、サポートされている各文字を特定の1バイトにマッピングするシングルバイト、またはサポートされている各文字を一連の1、2、3または4バイトにマッピングする可変幅マルチバイトのいずれかで構成されます。1つの文字コードに含まれる最大バイト数は文字セットによって異なります。

すべてのデータベース文字セットには、次の基本文字が含まれます。

- ラテン文字(AからZおよびaからz)
- 10進数(0から9)
- 記号文字(表2-1を参照)
- 空白文字(空白、タブおよび改行およびキャリッジ・リターン)

基本文字のみを使用するPL/SQLソース・テキストは、すべてのデータベースで格納およびコンパイルできます。基本文字以外の文字を使用するPL/SQLソース・テキストは、その文字をサポートするデータベース文字セットが設定されたデータベースのみで格納およびコンパイルできます。

表2-1 すべてのデータベース文字セットに含まれている記号文字

記号	名前
(	左丸カッコ
)	右丸カッコ
<	左山カッコ
>	右山カッコ
+	プラス記号
-	ハイフンまたはマイナス記号
*	アスタリスク
/	スラッシュ
=	等号
,	カンマ
;	セミコロン

記号	名前
:	コロン
.	ピリオド
!	感嘆符
?	疑問符
'	アポストロフィ(一重引用符)
"	引用符(二重引用符)
@	アットマーク
%	パーセント記号
#	シャープ記号
$	ドル記号
_	アンダースコア
|	縦棒

関連項目:

データベース文字セットの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

2.1.2 各国語文字セット

PL/SQLでは、各国語文字セットを使用して、NCHAR、NVARCHAR2およびNCLOBの各データ型の文字値を表現します。

関連項目:

- これらのデータ型の詳細は、「SQLデータ型」を参照してください
- 各国語文字セットの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

2.1.3 データ・バインドされた照合について

照合(ソート順序付けとも呼ばれる)は、2つの文字列が比較およびソートされるときに、ある文字列が別の文字列と等しいか、別の文字列の前に来るか後に来るかを決定する一連のルールです。

異なる照合は、異なる話し言葉のルールに対応します。照合依存の演算は、テキストを比較し、比較ルールの制御に照合を必要とする演算です。等価演算子および組込みファンクションINSTRは、照合依存の演算の例です。

Oracle Database 12cリリース2 (12.2)では、文字データの演算に適用される照合を制御するためのアーキテクチャが追加されました。新しいアーキテクチャでは、照合は文字データの属性(データ型に類似したもの)になります。列に対して照合を宣言できるようになり、列を参照している照合依存のすべてのSQL操作によってこの照合が自動的に適用されます。データバインドされた照合の機能では、ISO/IECのSQL標準と互換性がある構文およびセマンティクスが使用されます。

PL/SQL言語は、データバインドされた照合のアーキテクチャに対して限定的なサポートを提供します。PL/SQL式で処理されるすべてのデータは、照合USING_NLS_COMPと互換性があるものと見なされます。この疑似照合は、照合依存の演算子に対して、以前のOracle Databaseリリースの場合と同じように動作するように指示します。つまり、セッション・パラメータNLS_COMPおよびNLS_SORTの値によって、使用する照合が決まります。ただし、PL/SQLに埋め込まれたSQL文、またはPL/SQLで動的に構成されたSQL文はすべて、新しいアーキテクチャを完全にサポートします。

デフォルトの照合と呼ばれる新しいプロパティが表、ビュー、マテリアライズド・ビュー、パッケージ、ストアド・プロシージャ、ストアド・ファンクション、トリガーおよび型に追加されました。ユニットのデフォルトの照合は、そのユニット内で明示的に照合の宣言を行っていないデータ・コンテナ(列、変数、パラメータ、リテラルおよび戻り値など)に対する照合を決定します。パッケージ、ストアド・プロシージャ、ストアド・ファンクション、トリガーおよび型に対するデフォルトの照合はUSING_NLS_COMPであることが必要です。

構文およびセマンティクスの詳細は、DEFAULT COLLATION句を参照してください。

デフォルトの照合がUSING_NLS_COMP以外のスキーマでPL/SQLユニットを容易に作成できるよう、次の文の構文およびセマンティクスを使用して、オブジェクトのデフォルトの照合をUSING_NLS_COMPとして明示的に宣言できます。

- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE PROCEDURE文
- CREATE TRIGGER文
- CREATE TYPE文

関連項目:

- PL/SQLユニットに対するデータバインドされた照合の指定の詳細は、Oracle Databaseグローバリゼーション・サポート・ガイドを参照してください
- 有効なスキーマのデフォルトの照合の詳細は、Oracle Databaseグローバリゼーション・サポート・ガイドを参照してください

2.2 字句単位

PL/SQLの字句単位(デリミタ、識別子、リテラル、プラグマおよびコメント)は、PL/SQLの最小の個別構成要素です。

ここでのトピック

- デリミタ
- 識別子
- リテラル
- プラグマ
- コメント
- 字句単位の間の空白文字

2.2.1 デリミタ

デリミタは、PL/SQLで特別な意味を持つ文字、または文字の組合せです。

デリミタの中にデリミタ以外の文字(空白文字を含む)を埋め込まないでください。

表2-2に、PL/SQLのデリミタの概要を示します。

表2-2 PL/SQLのデリミタ

デリミタ	意味
+	加算演算子
:=	代入演算子
=>	結合演算子
%	属性のインジケータ
'	文字列のデリミタ
.	構成要素のインジケータ
||	連結演算子
/	除算演算子
**	指数演算子

デリミタ	意味
(	式またはリストのデリミタ(開始)
)	式またはリストのデリミタ(終了)
:	ホスト変数のインジケータ
,	項目のセパレータ
<<	ラベルのデリミタ(開始)
>>	ラベルのデリミタ(終了)
/*	複数行コメントのデリミタ(開始)
*/	複数行コメントのデリミタ(終了)
*	乗算演算子
"	二重引用符で囲んだ識別子のデリミタ
..	範囲演算子
=	関係演算子(等しい)
<>	関係演算子(等しくない)
!=	関係演算子(等しくない)
~=	関係演算子(等しくない)
^=	関係演算子(等しくない)
<	関係演算子(より小さい)
>	関係演算子(より大きい)
<=	関係演算子(以下)

デリミタ	意味
>=	関係演算子(以上)
@	リモート・アクセスのインジケータ
--	単一行コメントのインジケータ
;	文の終了記号
-	減算演算子または否定演算子

2.2.2 識別子

識別子は、次に示すPL/SQL要素に名前を付けます。

- 定数
- カーソル
- 例外
- キーワード
- ラベル
- パッケージ
- 予約語
- サブプログラム
- 型
- 変数

英字かどうかに関係なく、識別子に含まれているすべての文字が意味を持ちます。たとえば、識別子lastnameとlast_nameは別のものとみなされます。

隣接する識別子は、1つ以上の空白またはデリミタ文字で区切る必要があります。

「ユーザー定義の引用識別子」で説明している例を除き、PL/SQLでは識別子の大/小文字が区別されません。たとえば、識別子lastname、LastNameおよびLASTNAMEは同じものとみなされます。

ここでのトピック

- 予約語およびキーワード
- 事前定義の識別子
- ユーザー定義の識別子

2.2.2.1 予約語およびキーワード

予約語とキーワードは、PL/SQLで特別な意味を持つ識別子です。

通常のユーザー定義の識別子として、予約語を使用することはできません。予約語は、ユーザー定義の引用識別子として使用できますが、使用しないことをお薦めします。詳細は、「ユーザー定義の引用識別子」を参照してください。

キーワードは、通常のユーザー定義の識別子として使用できますが、使用しないことをお薦めします。

PL/SQLの予約語のリストは表D-1、キーワードのリストは表D-2を参照してください。

2.2.2.2 事前定義の識別子

事前定義の識別子は、事前定義のパッケージSTANDARDで宣言されています。

事前定義の識別子の例としては、例外INVALID_NUMBERがあります。

事前定義の識別子のリストを取得するには、DBAロールを持つユーザーとしてOracle Databaseに接続し、次の問合せを使用します。

```
SELECT TYPE_NAME FROM ALL_TYPES WHERE PREDEFINED='YES';
```

事前定義の識別子は、ユーザー定義の識別子として使用できますが、使用しないことをお薦めします。ローカルの宣言はグローバルな宣言をオーバーライドします(「識別子の有効範囲と可視性」を参照)。

2.2.2.3 ユーザー定義の識別子

ユーザー定義の識別子の特性は次のとおりです。

- データベース文字セットの文字で構成されます。
- 通常の識別子と引用識別子があります。

ヒント:

ユーザー定義の識別子は、意味のあるものにします。たとえば、cost_per_thousand は意味が明白ですが、cpt は意味があいまいです。

ヒント:

スキーマとスキーマ・オブジェクトの両方に同じユーザー定義識別子を使用しないでください。コードの可読性とメンテナンス性が低下し、コーディングの誤りにつながる可能性があります。ローカル・オブジェクトの名前解決は、スキーマ修飾よりも優先されます。

データベース・オブジェクトのネーミング・ルールの詳細は、『Oracle Database SQL 言語リファレンス』を参照してください

PL/SQL 固有の名前解決ルールの詳細は、PL/SQL と SQL での名前解決ルールの相違点を参照してくださ

い。

2.2.2.3.1 通常のユーザー定義の識別子

通常のユーザー定義の識別子の特性は次のとおりです。

- 先頭は英字です。
- 文字、数字および次の記号を使用できます。
  - ドル記号($)
  - シャープ記号(#)
  - アンダースコア(_)
- 予約語(表D-1を参照)は使用できません。

データベース文字セットには、文字および数字として分類される文字が定義されています。COMPATIBLEが12.2以上の値に設定されている場合、データベース文字セットでの識別子表現は、128バイトを超えないように定義する必要があります。COMPATIBLEが12.1以下の値に設定されている場合、上限は30バイトです。

通常のユーザー定義の識別子として有効な例を次に示します。

```
X
t2
phone#
credit_limit
LastName
oracle$number
money$$$tree
SN##
try_again_
```

通常のユーザー定義の識別子として無効な例を次に示します。

```
mine&yours
debit-amount
on/off
user id
```

2.2.2.3.2 ユーザー定義の引用識別子

ユーザー定義の引用識別子は、二重引用符で囲まれています。

二重引用符の間には、二重引用符、改行文字およびNULL文字以外の任意の文字をデータベース文字セットから使用できます。たとえば、次の識別子を使用できます。

```
"X+Y"
"last name"
"on/off switch"
"employee(s)"
"*** header info ***"
```

COMPATIBLEが12.2以上の値に設定されている場合、データベース文字セットでの引用識別子表現は、(二重引用符を除いて)128バイトを超えないように定義する必要があります。COMPATIBLEが12.1以下の値に設定されている場合、上限

は30バイトです。

ユーザー定義の引用識別子では大/小文字が区別されますが、例外が1つあります。ユーザー定義の引用識別子から二重引用符を除いたものが有効な通常のユーザー定義の識別子である場合は、その識別子を参照するときの二重引用符は省略可能になり、省略した場合は識別子の大/小文字が区別されなくなります。

お薦めしませんが、予約語をユーザー定義の引用識別子として使用することができます。予約語は有効な通常のユーザー定義の識別子ではないため、識別子は二重引用符で囲む必要があり、大/小文字は常に区別されます。

例2-1 ユーザー定義の引用識別子に対する、大/小文字を区別しない有効な参照

この例では、ユーザー定義の引用識別子"HELLO"は、二重引用符がなくても有効な通常のユーザー定義の識別子です。したがって、Helloという参照は有効です。

```
DECLARE
  "HELLO" varchar2(10) := 'hello';
BEGIN
  DBMS_Output.Put_Line(Hello);
END;
/
```

結果:

```
hello
```

例2-2 ユーザー定義の引用識別子に対する、大/小文字を区別しない無効な参照

この例では、"Hello"という参照は無効です。これは、二重引用符を付けると識別子の大/小文字が区別されるためです。

```
DECLARE
  "HELLO" varchar2(10) := 'hello';
BEGIN
  DBMS_Output.Put_Line("Hello");
END;
/
```

結果:

```
  DBMS_Output.Put_Line("Hello");
                        *
ERROR at line 4:
ORA-06550: line 4, column 25:
PLS-00201: identifier 'Hello' must be declared
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

例2-3 ユーザー定義の引用識別子としての予約語

この例では、ユーザー定義の引用識別子"BEGIN"、"Begin"および"begin"が宣言されています。BEGIN、Beginおよびbeginは同じ予約語を表しますが、"BEGIN"、"Begin"および"begin"は異なる識別子を表します。

```
DECLARE
  "BEGIN" varchar2(15) := 'UPPERCASE';
  "Begin" varchar2(15) := 'Initial Capital';
```

```
  "begin" varchar2(15) := 'lowercase';
BEGIN
  DBMS_Output.Put_Line("BEGIN");
  DBMS_Output.Put_Line("Begin");
  DBMS_Output.Put_Line("begin");
END;
/
```

結果:

```
UPPERCASE
Initial Capital
lowercase

PL/SQL procedure successfully completed.
```

例2-4 二重引用符の省略

この例では、二重引用符で囲まずに、予約語であるユーザー定義の引用識別子を参照しています。

```
DECLARE
  "HELLO" varchar2(10) := 'hello';  -- HELLO is not a reserved word
  "BEGIN" varchar2(10) := 'begin';  -- BEGIN is a reserved word
BEGIN
  DBMS_Output.Put_Line(Hello);      -- Double quotation marks are optional
  DBMS_Output.Put_Line(BEGIN);      -- Double quotation marks are required
end;
/
```

結果:

```
  DBMS_Output.Put_Line(BEGIN);      -- Double quotation marks are required
                       *
ERROR at line 6:
ORA-06550: line 6, column 24:
PLS-00103: Encountered the symbol "BEGIN" when expecting one of the following:
( ) - + case mod new not null <an identifier>
<a double-quoted delimited-identifier> <a bind variable>
table continue avg count current exists max min prior sql
stddev sum variance execute multiset the both leading
trailing forall merge year month day hour minute second
timezone_hour timezone_minute timezone_region timezone_abbr
time timestamp interval date
<a string literal with character set specificat
```

例2-5 大/小文字の区別の無視

この例では、大/小文字の区別を無視して、予約語であるユーザー定義の引用識別子を参照しています。

```
DECLARE
  "HELLO" varchar2(10) := 'hello';  -- HELLO is not a reserved word
  "BEGIN" varchar2(10) := 'begin';  -- BEGIN is a reserved word
BEGIN
  DBMS_Output.Put_Line(Hello);      -- Identifier is case-insensitive
  DBMS_Output.Put_Line("Begin");    -- Identifier is case-sensitive
END;
/
```

結果:

```
  DBMS_Output.Put_Line("Begin");     -- Identifier is case-sensitive
                         *
ERROR at line 6:
ORA-06550: line 6, column 25:
PLS-00201: identifier 'Begin' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

2.2.3 リテラル

リテラルは、識別子で表現したり他の値から計算する必要がない値です。

たとえば、123は整数リテラルで'abc'は文字リテラルですが、1+2はリテラルではありません。

PL/SQLリテラルにはすべてのSQLリテラル(『Oracle Database SQL言語リファレンス』を参照)およびBOOLEANリテラル(SQLにはありません)が含まれます。BOOLEANリテラルとは、事前定義の論理値TRUE、FALSEまたはNULLのことです。NULLは、不明な値を表します。

> ノート:
>
> 『Oracle Database SQL 言語リファレンス』と同様に、このマニュアルでも文字リテラルおよび文字列という用語を同じ意味で使用しています。

PL/SQLで文字リテラルを使用する場合は、次のことに注意してください。

- 文字リテラルは大/小文字が区別されます。

  たとえば、'Z'と'z'は別のものとみなされます。

- 空白文字が意味を持ちます。

  たとえば、次のリテラルは別のものとみなされます。

  ```
  'abc'
  '  abc'
  'abc '
  '  abc '
  'a b c'
  ```

- PL/SQLには、「文字列が次のソース行へ続く」ことを意味する行継続文字がありません。文字列を次のソース行へ続けた場合、文字列に改行文字が含まれます。

  たとえば、次のPL/SQLコードの場合:

  ```
  BEGIN
    DBMS_OUTPUT.PUT_LINE('This string breaks
  here.');
  END;
  /
  ```

  次のように出力されます。

```
This string breaks
here.
```

文字列が1つのソース行に収まらない場合に改行文字を含まないようにするには、連結演算子(||)を使用して文字列を構築します。

たとえば、次のPL/SQLコードの場合:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('This string ' ||
                       'contains no line-break character.');
END;
/
```

次のように出力されます。

```
This string contains no line-break character.
```

連結演算子の詳細は、「連結演算子」を参照してください。

- '0' から '9' は、整数リテラルの0から9と同じではありません。

  ただし、PL/SQLによって整数に変換されるため、算術式内で使用できます。

- 0(ゼロ)文字の文字リテラルは値NULLを持っており、NULL文字列と呼ばれます。

  ただし、このNULL値はBOOLEAN値のNULLではありません。

- 通常の文字リテラルは、データベース文字セット内の文字で構成されます。

  データベース文字セットの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

- 各国語文字リテラルは、各国語文字セット内の文字で構成されます。

  各国語文字セットの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

- QまたはqをQ文字リテラル構文の一部として使用して、代替の引用符メカニズムが使用されることを示すことができます。このメカニズムでは、単に一重引用符ではなく、文字列に対して様々なデリミタを使用できます。

  代替引用符メカニズムの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

Live SQL:

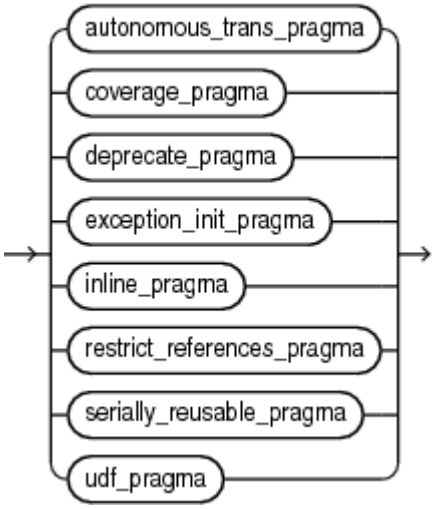
Q メカニズムの例は、文字列リテラルの代替引用符メカニズム("Q")で表示および実行できます。

2.2.4 プラグマ

プラグマは、コンパイラに対する命令で、コンパイル時に処理されます。

プラグマは予約語PRAGMAで始まり、プラグマの名前がその後に続きます。一部のプラグマには引数があります。プラグマは、宣言または文の前に記述できます。特定のプラグマにその他の制限が適用される場合もあります。プラグマの影響範囲はプラグマによって異なります。プラグマの名前または引数がコンパイラによって認識されない場合、そのプラグマによる影響はありません。

pragma ::=



プラグマの構文およびセマンティクスの詳細は、次を参照してください。

- AUTONOMOUS_TRANSACTIONプラグマ
- COVERAGEプラグマ
- DEPRECATEプラグマ
- EXCEPTION_INITプラグマ
- INLINEプラグマ
- RESTRICT_REFERENCESプラグマ
- SERIALLY_REUSABLEプラグマ
- UDFプラグマ

2.2.5 コメント

PL/SQLコンパイラは、コメントを無視します。コメントを付けておくと、他のアプリケーション開発者がソース・テキストを理解しやすくなります。

一般に、コメントは各コード・セグメントの目的や使用方法を説明するために使用します。また、廃止されたコード部分や作成途中のコード部分をコメントに入れて無効化することもできます。

ここでのトピック

- 単一行コメント
- 複数行コメント

関連項目:

「コメント」

2.2.5.1 単一行コメント

単一行コメントは--で始まり、その行の終わりまで続きます。

注意:

Oracle プリコンパイラ・プログラムが動的に処理する PL/SQL ブロックの中に単一行コメントを置かないでください。Oracle プリコンパイラ・プログラムでは行の終わりを示す文字が無視されるため、単一行コメントはブロックの末尾で終了します。

プログラムのテストまたはデバッグ中に、コード内の1行をコメントにすることによって無効にできます。たとえば:

```
-- DELETE FROM employees WHERE comm_pct IS NULL
```

例2-6 単一行コメント

この例には、3つの単一行コメントがあります。

```
DECLARE
  howmany     NUMBER;
  num_tables  NUMBER;
BEGIN
  -- Begin processing
  SELECT COUNT(*) INTO howmany
  FROM USER_OBJECTS
  WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
  num_tables := howmany;       -- Compute another value
END;
/
```

2.2.5.2 複数行コメント

複数行コメントは、/*で始まって*/で終わり、複数行にまたがることができます。

複数行コメントのデリミタを使用すると、コードの一部分をコメントにできます。その場合、複数行コメントをネストしないように注意してください。複数行コメントに別の複数行コメントを含めることはできません。ただし、複数行コメントに単一行コメントを含めることはできます。たとえば、次の場合、構文エラーが発生します。

```
/*
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  /* We expect this THEN to always be performed */
  END IF;
*/
```

次の場合、構文エラーは発生しません。

```
/*
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  -- We expect this THEN to always be performed
  END IF;
*/
```

例2-7 複数行コメント

この例には、2つの複数行コメントがあります。(SQLファンクションTO_CHARは、引数に相当する文字を戻します。TO_CHARの詳

細は、『Oracle Database SQL言語リファレンス』を参照してください。)

```
DECLARE
  some_condition  BOOLEAN;
  pi              NUMBER := 3.1415926;
  radius          NUMBER := 15;
  area            NUMBER;
BEGIN
  /* Perform some simple tests and assignments */

  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  /* We expect this THEN to always be performed */
  END IF;

  /* This line computes the area of a circle using pi,
  which is the ratio between the circumference and diameter.
  After the area is computed, the result is displayed. */

  area := pi * radius**2;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || TO_CHAR(area));
END;
/
```

結果:

```
The area is: 706.858335
```

2.2.6 字句単位の間の空白文字

字句単位の間に空白文字を挿入できます。多くの場合、これによってソース・テキストが読みやすくなります。

例2-8 空白文字によるソース・テキストの読みやすさの向上

```
DECLARE
  x    NUMBER := 10;
  y    NUMBER := 5;
  max  NUMBER;
BEGIN
  IF x>y THEN max:=x;ELSE max:=y;END IF;  -- correct but hard to read

  -- Easier to read:

  IF x > y THEN
    max:=x;
  ELSE
    max:=y;
  END IF;
END;
/
```

2.3 宣言

宣言によって、指定されたデータ型の値の記憶域を割り当て、値を参照できるように格納場所に名前を付けます。

オブジェクトは、参照する前に宣言しておく必要があります。宣言は、任意のブロック、サブプログラムまたはパッケージの宣言部で使用できます。

ここでのトピック

- 変数の宣言
- 定数の宣言
- 変数および定数の初期値
- NOT NULL制約
- %TYPE属性を使用した項目の宣言

変数と定数以外のオブジェクトの宣言の詳細は、「ブロック」のdeclare_sectionの構文を参照してください。

2.3.1 NOT NULL制約

スカラー変数または定数(またはコンポジット変数または定数のスカラー・コンポーネント)には、NOT NULL制約を指定できます。

NOT NULL制約を指定すると、その項目にNULL値を代入できなくなります。この制約は、(データ型から)暗黙的に指定するか、または明示的に指定することができます。

NOT NULLを指定するスカラー変数の宣言では、暗黙的か明示的かに関係なく、初期値を変数に代入する必要があります(スカラー変数のデフォルトの初期値はNULLであるため)。

PL/SQLは長さが0(ゼロ)の文字値をすべてNULL値とみなします。これには文字関数やブール式によって戻された値が含まれます。

NULL値かどうかをテストするには、「IS [NOT] NULL演算子」を使用します。

例

例2-9 NOT NULL制約が指定された変数の宣言

この例では、変数acct_idにNOT NULL制約を明示的に指定し、変数a、bおよびcにはデータ型から暗黙的に指定します。

```
DECLARE
  acct_id INTEGER(4) NOT NULL := 9999;
  a NATURALN                  := 9999;
  b POSITIVEN                 := 9999;
  c SIMPLE_INTEGER            := 9999;
BEGIN
  NULL;
END;
/
```

例2-10 NULL値に初期化される変数

この例では、すべての変数がNULLに初期化されます。

```
DECLARE
  null_string  VARCHAR2(80) := TO_CHAR('');
  address      VARCHAR2(80);
```

```
  zip_code      VARCHAR2(80) := SUBSTR(address, 25, 0);
  name          VARCHAR2(80);
  valid         BOOLEAN      := (name != '');
BEGIN
  NULL;
END;
/
```

2.3.2 変数の宣言

変数の宣言では、変数の名前とデータ型を常に指定します。

ほとんどのデータ型の場合、変数を宣言するときに初期値を指定することもできます。

変数名は、有効なユーザー定義の識別子である必要があります。

データ型は、任意のPL/SQLデータ型を選択できます。PL/SQLデータ型にはSQLデータ型が含まれます。データ型は、スカラー(内部コンポーネントなし)またはコンポジット(内部コンポーネントあり)のいずれかです。

例

例2-11 スカラー変数の宣言

この例では、いくつかの変数をスカラー・データ型で宣言しています。

```
DECLARE
  part_number       NUMBER(6);      -- SQL data type
  part_name         VARCHAR2(20);   -- SQL data type
  in_stock          BOOLEAN;        -- PL/SQL-only data type
  part_price        NUMBER(6,2);    -- SQL data type
  part_description  VARCHAR2(50);   -- SQL data type
BEGIN
  NULL;
END;
/
```

関連トピック

- ユーザー定義の識別子
- スカラー変数の宣言の構文は、「スカラー変数の宣言」を参照してください
- スカラーのデータ型の詳細は、「PL/SQLのデータ型」を参照してください
- コンポジット・データ型およびコンポジット変数の詳細は、「PL/SQLのコレクションおよびレコード」を参照してください

2.3.3 定数の宣言

定数は、変更されない値を保持します。

「変数の宣言」の内容は定数の宣言にも当てはまりますが、定数の宣言には必要なものがさらに2つ(キーワードCONSTANTと定数の初期値)あります。(定数の初期値は永続的な値になります。)

例2-12 定数の宣言

この例では、3つの定数をスカラー・データ型で宣言しています。

```
DECLARE
  credit_limit     CONSTANT REAL    := 5000.00;  -- SQL data type
  max_days_in_year CONSTANT INTEGER := 366;      -- SQL data type
  urban_legend     CONSTANT BOOLEAN := FALSE;    -- PL/SQL-only data type
BEGIN
  NULL;
END;
/
```

関連項目

- 定数の宣言の構文は、「定数の宣言」を参照してください

2.3.4 変数および定数の初期値

変数宣言では、NOT NULL制約を指定しないかぎり、初期値はオプションです。定数の宣言では、初期値は必須です。

ブロックまたはサブプログラムに宣言が含まれている場合は、ブロックまたはサブプログラムに制御が渡されるたびに、変数または定数に初期値が代入されます。宣言がパッケージ仕様部にある場合は、(変数または定数がパブリックかプライベートかには関係なく)セッションごとに初期値が変数または定数に代入されます。

初期値を指定するには、代入演算子(:=)またはキーワードDEFAULTを使用し、その後に式を指定します。式には、事前に宣言済の定数と、事前に初期化済の変数を使用できます。

変数に初期値を指定しない場合は、他のコンテキストで使用する前に、値を代入します。

例

例2-13 初期値が指定された変数および定数の宣言

この例では、宣言する定数および変数に初期値を代入します。areaの初期値は、事前に宣言済の定数piおよび事前に初期化済の変数radiusに応じて異なります。

```
DECLARE
  hours_worked    INTEGER := 40;
  employee_count  INTEGER := 0;
  pi      CONSTANT REAL := 3.14159;
  radius          REAL := 1;
  area            REAL := (pi * radius**2);
BEGIN
  NULL;
END;
/
```

例2-14 デフォルトでNULLに初期化される変数

この例では、変数counterの初期値はデフォルトのNULLです。この例では、「IS [NOT] NULL演算子」を使用して、NULLがゼロとは異なることを示しています。

```
DECLARE
  counter INTEGER;  -- initial value is NULL by default
BEGIN
```

```
  counter := counter + 1;  -- NULL + 1 is still NULL

  IF counter IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('counter is NULL.');
  END IF;
END;
/
```

結果:

```
counter is NULL.
```

関連トピック

- 定数の連想配列の宣言の詳細は、「連想配列の定数の宣言」を参照してください
- 定数レコードの宣言の詳細は、「レコード定数の宣言」を参照してください
- NOT NULL制約

2.3.5 %TYPE属性を使用した項目の宣言

%TYPE属性を使用すると、事前に宣言されている変数または列と同じデータ型のデータ項目を(データ型を知らなくても)宣言できます。参照先項目の宣言が変更されると、それに応じて参照元項目の宣言も変更されます。

次に宣言の構文を示します。

```
referencing_item referenced_item%TYPE;
```

参照元項目および参照先項目にすることができる項目の種類は、「%TYPE属性」を参照してください。

参照元項目は、参照先項目から次のものを継承します。

- データ型およびサイズ
- 制約(参照先項目が列である場合を除く)

参照元項目は、参照先項目の初期値は継承しません。そのため、参照元項目でNOT NULL制約が指定または継承される場合は、参照元項目の初期値を指定する必要があります。

%TYPE属性は、データベースの値を保持する変数を宣言する場合に特に便利です。列と同じ型の変数を宣言する構文は、次のようになります。

```
variable_name table_name.column_name%TYPE;
```

関連項目:

「%ROWTYPE属性を使用した項目の宣言」(この属性を使用すると、データベースの表またはビュー内の行の全体または一部を表すレコード変数を宣言できます)

例

例2-15 列と同じ型の変数の宣言

この例では、変数surnameは、NOT NULL制約が指定されている列employees.last_nameのデータ型とサイズを継承しています。surnameはNOT NULL制約を継承しないため、その宣言に初期値は不要です。

```
DECLARE
  surname  employees.last_name%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

結果:

```
surname=
```

例2-16 別の変数と同じ型の変数の宣言

この例では、変数surnameは変数nameのデータ型、サイズおよびNOT NULL制約を継承しています。surnameはnameの初期値を継承しないため、その宣言には初期値(25文字以内)が必要です。

```
DECLARE
  name     VARCHAR(25) NOT NULL := 'Smith';
  surname  name%TYPE := 'Jones';
BEGIN
  DBMS_OUTPUT.PUT_LINE('name=' || name);
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

結果:

```
name=Smith
surname=Jones
```

2.4 識別子に対する参照

識別子を参照するとき、名前には、単純名、修飾名、リモート名または修飾リモート名を使用します。

識別子の単純名は、その宣言に含まれる名前です。たとえば:

```
DECLARE
  a INTEGER;  -- Declaration
BEGIN
  a := 1;     -- Reference with simple name
END;
/
```

識別子が名前付きPL/SQLユニットで宣言されている場合、その参照に修飾名を使用できます(場合によっては必須)。構文(ドット表記法と呼ばれる)は次のとおりです。

```
unit_name.simple_identifier_name
```

たとえば、パッケージpで識別子aを宣言している場合、その識別子は修飾名p.aで参照できます。ユニット名も修飾できます(場

合によっては必須)。識別子を参照できない場合は、その識別子を修飾する必要があります(「識別子の有効範囲と可視性」を参照)。

リモート・データベース上のオブジェクトに識別子で名前が付けられている場合、その識別子はリモート名で参照する必要があります。構文は次のとおりです。

```
simple_identifier_name@link_to_remote_database
```

リモート・データベース上にあるPL/SQLユニットで識別子が宣言されている場合、その識別子は修飾リモート名で参照する必要があります。構文は次のとおりです。

```
unit_name.simple_identifier_name@link_to_remote_database
```

リモート・スキーマ・オブジェクトのシノニムは作成できますが、PL/SQLのサブプログラムまたはパッケージ内で宣言されたオブジェクトのシノニムは作成できません。シノニムを作成するには、SQL文CREATE SYNONYMを使用します(『Oracle Database SQL言語リファレンス』を参照)。

PL/SQLでのあいまいな名前の解決方法の詳細は、「PL/SQLの名前解決」を参照してください。

ノート:

パッケージ STANDARD および DBMS_STANDARD で宣言された識別子は、同じ名前のローカル識別子を宣言していないかぎり、パッケージ名で修飾しなくても参照できます(「識別子の有効範囲と可視性」を参照)。

2.5 識別子の有効範囲と可視性

識別子の有効範囲とは、その識別子の参照が可能な、PL/SQLユニットの領域です。識別子の可視性とは、修飾なしでその識別子の参照が可能な、PL/SQLユニットの領域です。識別子は、自身が宣言されているPL/SQLユニットに対してローカルです。そのユニットにサブユニットがある場合、識別子はサブユニットに対してグローバルです。

サブユニットでグローバル識別子が再宣言されると、そのサブユニット内では両方の識別子が有効範囲内にあることになりますが、ローカル識別子のみが表示されます。サブユニットでグローバル識別子を参照するには、グローバル識別子が宣言されているユニットの名前でグローバル識別子を修飾する必要があります。そのユニットに名前がない場合、サブユニットでグローバル識別子を参照することはできません。

PL/SQLユニットは、あるユニットから同じレベルの他のユニットで宣言されている識別子への参照はできません。そのような識別子は、そのブロックに対してローカルでもグローバルでもないためです。

同じPL/SQLユニット内で、同じ識別子を2回宣言することはできません。そうすると、重複した識別子を参照するときにエラーが発生します。

2つの異なるユニットであれば、同じ識別子を宣言できます。その識別子で表される2つのオブジェクトは区別されます。片方を変更しても、もう片方には影響しません。

同じ有効範囲内では、混乱や予期しない結果を回避するためにラベルとサブプログラムに一意の名前を付けます。

例

例2-17 識別子の有効範囲と可視性

この例では、いくつかの識別子の有効範囲と可視性を示します。最初のサブブロックで、グローバル識別子aが再宣言されています。グローバル変数aを参照するには、最初のサブブロックで、外側のブロックの名前を使用して変数を修飾する必要がありますが、外側のブロックには名前がありません。そのため、最初のサブブロックではグローバル変数aを参照することはできず、参照できるのはローカル変数aのみです。サブブロックは同じレベルにあるため、最初のサブブロックはdを参照できず、2番目のサブブロックはcを参照できません。

```
-- Outer block:
DECLARE
  a CHAR;  -- Scope of a (CHAR) begins
  b REAL;    -- Scope of b begins
BEGIN
  -- Visible: a (CHAR), b

  -- First sub-block:
  DECLARE
    a INTEGER;  -- Scope of a (INTEGER) begins
    c REAL;       -- Scope of c begins
  BEGIN
    -- Visible: a (INTEGER), b, c
    NULL;
  END;          -- Scopes of a (INTEGER) and c end
  -- Second sub-block:
  DECLARE
    d REAL;     -- Scope of d begins
  BEGIN
    -- Visible: a (CHAR), b, d
    NULL;
  END;          -- Scope of d ends
-- Visible: a (CHAR), b
END;            -- Scopes of a (CHAR) and b end
/
```

例2-18 ブロック・ラベルによる、再宣言されたグローバル識別子の修飾

この例では、外側のブロックにouterという名前のラベルを付けています。そのため、サブブロックでは、グローバル変数birthdateを再宣言した後、ブロックのラベルを使用して変数名を修飾することで、このグローバル変数を参照できます。サブブロックでは、単純名によってローカル変数birthdateも参照できます。

```
<<outer>>  -- label
DECLARE
  birthdate DATE := TO_DATE('09-AUG-70', 'DD-MON-YY');
BEGIN
  DECLARE
    birthdate DATE := TO_DATE('29-SEP-70', 'DD-MON-YY');
  BEGIN
    IF birthdate = outer.birthdate THEN
      DBMS_OUTPUT.PUT_LINE ('Same Birthday');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('Different Birthday');
    END IF;
```

```
  END;
END;
/
```

結果:

```
Different Birthday
```

例2-19 サブプログラム名による識別子の修飾

この例では、プロシージャcheck_creditで、変数ratingとファンクションcheck_ratingを宣言しています。ファンクションで変数を再宣言します。その後、ファンクションはグローバル変数をプロシージャ名で修飾して参照します。

```
CREATE OR REPLACE PROCEDURE check_credit (credit_limit NUMBER) AS
  rating NUMBER := 3;

  FUNCTION check_rating RETURN BOOLEAN IS
    rating  NUMBER := 1;
    over_limit  BOOLEAN;
  BEGIN
    IF check_credit.rating <= credit_limit THEN  -- reference global variable
      over_limit := FALSE;
    ELSE
      over_limit := TRUE;
      rating := credit_limit;                    -- reference local variable
    END IF;
    RETURN over_limit;
  END check_rating;
BEGIN
  IF check_rating THEN
    DBMS_OUTPUT.PUT_LINE
      ('Credit rating over limit (' || TO_CHAR(credit_limit) || ').  '
      || 'Rating: ' || TO_CHAR(rating));
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Credit rating OK.  ' || 'Rating: ' || TO_CHAR(rating));
  END IF;
END;
/

BEGIN
  check_credit(1);
END;
/
```

結果:

```
Credit rating over limit (1).  Rating: 3
```

例2-20 同じ有効範囲での重複する識別子

同じPL/SQLユニット内で、同じ識別子を2回宣言することはできません。この例に示すとおり、これを行うと、重複する識別子を参照したときにエラーが発生します。

```
DECLARE
  id  BOOLEAN;
  id  VARCHAR2(5);  -- duplicate identifier
BEGIN
  id := FALSE;
END;
/
```

結果:

```
  id := FALSE;
  *
ERROR at line 5:
ORA-06550: line 5, column 3:
PLS-00371: at most one declaration for 'ID' is permitted
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored
```

例2-21 異なるユニットでの同じ識別子の宣言

2つの異なるユニットであれば、同じ識別子を宣言できます。その識別子で表される2つのオブジェクトは区別されます。この例に示すとおり、1つを変更しても、もう1つに影響はありません。同じ有効範囲内では、混乱や予期しない結果を回避するためにラベルとサブプログラムに一意の名前を付けます。

```
DECLARE
  PROCEDURE p
  IS
    x VARCHAR2(1);
  BEGIN
    x := 'a';  -- Assign the value 'a' to x
    DBMS_OUTPUT.PUT_LINE('In procedure p, x = ' || x);
  END;

  PROCEDURE q
  IS
    x VARCHAR2(1);
  BEGIN
    x := 'b';  -- Assign the value 'b' to x
    DBMS_OUTPUT.PUT_LINE('In procedure q, x = ' || x);
  END;

BEGIN
  p;
  q;
END;
/
```

結果:

```
In procedure p, x = a
In procedure q, x = b
```

例2-22 同じ有効範囲内で同じ名前を持つラベルとサブプログラム

この例では、echoがブロックとサブプログラムの両方の名前になっています。ブロックおよびサブプログラムの両方で、xという変数を

宣言しています。サブプログラム内では、echo.xは、グローバル変数xではなく、ローカル変数xを参照します。

```
<<echo>>
DECLARE
  x  NUMBER := 5;

  PROCEDURE echo AS
    x  NUMBER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('x = ' || x);
    DBMS_OUTPUT.PUT_LINE('echo.x = ' || echo.x);
  END;

BEGIN
  echo;
END;
/
```

結果:

```
x = 0
echo.x = 0
```

例2-23 複数の重複したラベルを使用するブロック

この例では、外側のブロックに対して2つのラベルcompute_ratioとanother_labelが存在します。2番目のラベルは、内側のブロックに再度現れます。内側のブロック内では、another_label.denominatorはローカル変数denominatorを参照し、グローバル変数denominatorを参照しないため、ZERO_DIVIDEエラーが発生します。

```
<<compute_ratio>>
<<another_label>>
DECLARE
  numerator   NUMBER := 22;
  denominator NUMBER := 7;
BEGIN
  <<another_label>>
  DECLARE
    denominator NUMBER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Ratio with compute_ratio.denominator = ');
    DBMS_OUTPUT.PUT_LINE(numerator/compute_ratio.denominator);

    DBMS_OUTPUT.PUT_LINE('Ratio with another_label.denominator = ');
    DBMS_OUTPUT.PUT_LINE(numerator/another_label.denominator);

  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Divide-by-zero error: can''t divide '
        || numerator || ' by ' || denominator);
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unexpected error.');
  END another_label;
END compute_ratio;
/
```

結果:

```
Ratio with compute_ratio.denominator =
3.142857142857142857142857142857142857142857
Ratio with another_label.denominator =
Divide-by-zero error: cannot divide 22 by 0
```

2.6 変数への値の代入

変数を宣言した後、次の方法でその変数に値を代入できます。

- 代入文を使用して式の値を代入する方法。
- SELECT INTO文またはFETCH文を使用して、表の値を割り当てます。
- OUTパラメータまたはIN OUTパラメータとしてサブプログラムに渡し、サブプログラム内で値を代入する方法。

変数と値のデータ型には、互換性が必要です。あるデータ型を暗黙的に別のデータ型に変換できる場合、この2つのデータ型には互換性があります。暗黙的なデータ変換の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

ここでのトピック

- 代入文を使用した変数への値の代入
- SELECT INTO文を使用した変数への値の代入
- サブプログラムのパラメータとしての変数への値の代入
- ブール変数への値の代入

関連項目:

- 「コレクション変数への値の代入」
- 「レコード変数への値の代入」
- 「FETCH文」

2.6.1 代入文を使用した変数への値の代入

式の値を変数に代入するには、次の形式の代入文を使用します。

```
variable_name := expression;
```

代入文の構文の詳細は、「代入文」を参照してください。

式の構文は、「式」を参照してください。

例2-24 代入文を使用した変数への値の代入

この例では、複数の変数を宣言(一部には初期値も指定)してから、代入文を使用して式の値を変数に代入します。

```
DECLARE  -- You can assign initial values here
  wages          NUMBER;
  hours_worked   NUMBER := 40;
```

```
  hourly_salary  NUMBER := 22.50;
  bonus          NUMBER := 150;
  country        VARCHAR2(128);
  counter        NUMBER := 0;
  done           BOOLEAN;
  valid_id       BOOLEAN;
  emp_rec1       employees%ROWTYPE;
  emp_rec2       employees%ROWTYPE;
  TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  comm_tab       commissions;

BEGIN  -- You can assign values here too
  wages := (hours_worked * hourly_salary) + bonus;
  country := 'France';
  country := UPPER('Canada');
  done := (counter > 100);
  valid_id := TRUE;
  emp_rec1.first_name := 'Antonio';
  emp_rec1.last_name := 'Ortiz';
  emp_rec1 := emp_rec2;
  comm_tab(5) := 20000 * 0.15;
END;
/
```

2.6.2 SELECT INTO文を使用した変数への値の代入

SELECT INTO文の単純な形式は次のとおりです。

```
SELECT select_item [, select_item ]...
INTO variable_name [, variable_name ]...
FROM table_name;
```

それぞれのselect_itemに、対応する型互換性のあるvariable_nameが必要です。SQLにはBOOLEAN型がないため、variable_nameはBOOLEAN変数にはできません。

SELECT INTO文の構文の詳細は、「SELECT INTO文」を参照してください。

例2-25 SELECT INTO文を使用した変数への値の代入

この例では、SELECT INTO文を使用して、employee_idが100である従業員の給与の10%にあたる値を変数bonusに代入します。

```
DECLARE
  bonus   NUMBER(8,2);
BEGIN
  SELECT salary * 0.10 INTO bonus
  FROM employees
  WHERE employee_id = 100;
END;
DBMS_OUTPUT.PUT_LINE('bonus = ' || TO_CHAR(bonus));
/
```

結果:

```
bonus = 2400
```

2.6.3 サブプログラムのパラメータとしての変数への値の代入

変数をOUTパラメータまたはIN OUTパラメータとしてサブプログラムに渡し、サブプログラムによってこれらのパラメータに値が代入される場合は、サブプログラムの実行後、その値が変数で保持されます。詳細は、「サブプログラム・パラメータ」を参照してください。

例2-26 IN OUTサブプログラム・パラメータとしての変数への値の代入

この例では、変数new_salをプロシージャadjust_salaryに渡します。このプロシージャによって、対応する仮パラメータsalに値が代入されます。salはIN OUTパラメータであるため、プロシージャの実行終了後、代入された値が変数new_salで保持されます。

```
DECLARE
  emp_salary  NUMBER(8,2);

  PROCEDURE adjust_salary (
    emp        NUMBER,
    sal IN OUT NUMBER,
    adjustment NUMBER
  ) IS
  BEGIN
    sal := sal + adjustment;
  END;

BEGIN
  SELECT salary INTO emp_salary
  FROM employees
  WHERE employee_id = 100;

  DBMS_OUTPUT.PUT_LINE
   ('Before invoking procedure, emp_salary: ' || emp_salary);

  adjust_salary (100, emp_salary, 1000);

  DBMS_OUTPUT.PUT_LINE
   ('After invoking procedure, emp_salary: ' || emp_salary);
END;
/
```

結果:

```
Before invoking procedure, emp_salary: 24000
After invoking procedure, emp_salary: 25000
```

2.6.4 ブール変数への値の代入

BOOLEAN変数に代入できる値は、TRUE、FALSEおよびNULLのみです。

BOOLEANデータ型の詳細は、「ブール・データ型」を参照してください。

例2-27 BOOLEAN変数への値の代入

この例では、BOOLEAN変数doneをデフォルトでNULLに初期化し、リテラル値FALSEを代入して、リテラル値TRUEと比較し、

BOOLEAN式の値を代入します。

```
DECLARE
  done    BOOLEAN;              -- Initial value is NULL by default
  counter NUMBER := 0;
BEGIN
  done := FALSE;                -- Assign literal value
  WHILE done != TRUE            -- Compare to literal value
    LOOP
      counter := counter + 1;
      done := (counter > 500);  -- Assign value of BOOLEAN expression
    END LOOP;
END;
/
```

2.7 式

式は、1つ以上の値、演算子、および値に評価されるSQLファンクションの組合せです。

式は常に1つの値を戻します。単純な式には次のものがあります(複雑度の低い順に示しています)。

1. 1つの定数または変数(aなど)
2. 単項演算子とその1つのオペランド(-aなど)
3. バイナリ演算子とその2つのオペランド(a+bなど)

オペランドは、変数、定数、リテラル、演算子、ファンクション起動、プレースホルダ、またはその他の式にすることができます。そのため、式は複雑になる場合があります。式の構文の詳細は、「式」を参照してください。

オペランドのデータ型によって、式のデータ型が決まります。式が評価されるたびに、結果としてそのデータ型の値が1つ得られます。結果のデータ型は、式のデータ型です。

ここでのトピック

- 連結演算子
- 演算子の優先順位
- 論理演算子
- 短絡評価
- 比較演算子
- BOOLEAN式
- CASE式
- PL/SQL式でのSQLファンクション

2.7.1 連結演算子

連結演算子(||)は、ある文字列オペランドを別の文字列オペランドに追加します。

連結演算子は、NULLオペランドを無視します。

連結演算子の構文の詳細は、「character_expression ::=」を参照してください。

例2-28 連結演算子

```
DECLARE
  x VARCHAR2(4) := 'suit';
  y VARCHAR2(4) := 'case';
BEGIN
  DBMS_OUTPUT.PUT_LINE (x || y);
END;
/
```

結果:

```
suitcase
```

例2-29 連結演算子とNULLオペランド

連結演算子は、この例に示すとおり、NULLオペランドを無視します。

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('apple' || NULL || NULL || 'sauce');
END;
/
```

結果:

```
applesauce
```

2.7.2 演算子の優先順位

演算は、単項演算子とオペランド1つか、またはバイナリ演算子とオペランド2つのいずれかです。式内の演算は、演算子の優先順位に応じて評価されます。

表2-3に、優先順位の高い順に演算子を示します。同じ優先順位の演算子は、特に順序を考慮せずに評価されます。

表2-3 演算子の優先順位

演算子	演算
**	指数
+, -	恒等、否定
*, /	乗算、除算
+, -, ||	加算、減算、連結

演算子	演算
=、<、>、<=、>=、<>、!=、~=、 ^=、IS NULL、LIKE、BETWEEN、IN	比較
NOT	否定
AND	論理積
OR	論理和

評価の順序を制御するには、例2-30に示すとおり、演算をカッコで囲みます。

カッコがネストされている場合は、最も深くネストされた演算が最初に評価されます。

カッコが評価の順序に影響しない場合でも、カッコを使用してわかりやすくすることができます。

例2-30 カッコを使用した評価順序の制御

```
DECLARE
  a INTEGER := 1+2**2;
  b INTEGER := (1+2)**2;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
  DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/
```

結果:

```
a = 5
b = 9
```

例2-31 ネストしたカッコを使用した式

この例では、演算(1+2)および(3+4)が最初に評価され、それぞれ3および7という値が算出されます。次に、演算3*7が評価され、21という結果が算出されます。最後に21/7が評価され、最終的な値として3が算出されます。

```
DECLARE
  a INTEGER := ((1+2)*(3+4))/7;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
END;
/
```

結果:

```
a = 3
```

例2-32 カッコを使用した読みやすさの向上

この例では、カッコは評価の順序に影響しません。これらは、わかりやすくするためにのみ使用されます。

```
DECLARE
  a INTEGER := 2**2*3**2;
  b INTEGER := (2**2)*(3**2);
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
  DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/
```

結果:

```
a = 36
b = 36
```

例2-33 演算子の優先順位

この例では、より複雑ないくつかの式での演算子の優先順位とカッコの効果を示します。

```
DECLARE
  salary      NUMBER := 60000;
  commission  NUMBER := 0.10;
BEGIN
  -- Division has higher precedence than addition:

  DBMS_OUTPUT.PUT_LINE('5 + 12 / 4 = ' || TO_CHAR(5 + 12 / 4));
  DBMS_OUTPUT.PUT_LINE('12 / 4 + 5 = ' || TO_CHAR(12 / 4 + 5));

 -- Parentheses override default operator precedence:

  DBMS_OUTPUT.PUT_LINE('8 + 6 / 2 = ' || TO_CHAR(8 + 6 / 2));
  DBMS_OUTPUT.PUT_LINE('(8 + 6) / 2 = ' || TO_CHAR((8 + 6) / 2));

  -- Most deeply nested operation is evaluated first:

  DBMS_OUTPUT.PUT_LINE('100 + (20 / 5 + (7 - 3)) = '
                       || TO_CHAR(100 + (20 / 5 + (7 - 3))));

  -- Parentheses, even when unnecessary, improve readability:

  DBMS_OUTPUT.PUT_LINE('(salary * 0.05) + (commission * 0.25) = '
    || TO_CHAR((salary * 0.05) + (commission * 0.25))
  );

  DBMS_OUTPUT.PUT_LINE('salary * 0.05 + commission * 0.25 = '
    || TO_CHAR(salary * 0.05 + commission * 0.25)
  );
END;
/
```

結果:

```
5 + 12 / 4 = 8
12 / 4 + 5 = 8
8 + 6 / 2 = 11
(8 + 6) / 2 = 7
100 + (20 / 5 + (7 - 3)) = 108
```

```
(salary * 0.05) + (commission * 0.25) = 3000.025
salary * 0.05 + commission * 0.25 = 3000.025
```

2.7.3 論理演算子

論理演算子AND、OR、およびNOTは、3値論理に従います。

ANDとORはバイナリ演算子、NOTは単項演算子です。

表2-4 論理真理値表

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

ANDは、両方のオペランドがTRUEの場合のみ、TRUEを戻します。

ORは、どちらか一方のオペランドがTRUEの場合に、TRUEを戻します。

NOTは、オペランドがNULLの場合を除き、オペランドの反対の値を戻します。NULLは値を持たないため、NOT NULLはNULLを戻します。

例2-34 BOOLEAN変数を出力するプロシージャ

この例では、BOOLEAN変数の値を出力するprint_booleanというプロシージャを作成します。このプロシージャでは、「IS [NOT] NULL演算子」が使用されます。この章で紹介するいくつかの例で、print_booleanを起動しています。

```
CREATE OR REPLACE PROCEDURE print_boolean (
  b_name   VARCHAR2,
  b_value  BOOLEAN
) AUTHID DEFINER IS
BEGIN
  IF b_value IS NULL THEN
    DBMS_OUTPUT.PUT_LINE (b_name || ' = NULL');
```

```
  ELSIF b_value = TRUE THEN
    DBMS_OUTPUT.PUT_LINE (b_name || ' = TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE (b_name || ' = FALSE');
  END IF;
END;
/
```

例2-35 AND演算子

表2-4およびこの例に示すとおり、両方のオペランドがTRUEである場合にのみ、ANDはTRUEを戻します。

```
DECLARE
  PROCEDURE print_x_and_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
   print_boolean ('x', x);
   print_boolean ('y', y);
   print_boolean ('x AND y', x AND y);
 END print_x_and_y;

BEGIN
 print_x_and_y (FALSE, FALSE);
 print_x_and_y (TRUE, FALSE);
 print_x_and_y (FALSE, TRUE);
 print_x_and_y (TRUE, TRUE);

 print_x_and_y (TRUE, NULL);
 print_x_and_y (FALSE, NULL);
 print_x_and_y (NULL, TRUE);
 print_x_and_y (NULL, FALSE);
END;
/
```

結果:

```
x = FALSE
y = FALSE
x AND y = FALSE
x = TRUE
y = FALSE
x AND y = FALSE
x = FALSE
y = TRUE
x AND y = FALSE
x = TRUE
y = TRUE
x AND y = TRUE
x = TRUE
y = NULL
x AND y = NULL
x = FALSE
y = NULL
x AND y = FALSE
```

```
x = NULL
y = TRUE
x AND y = NULL
x = NULL
y = FALSE
x AND y = FALSE
```

例2-36 OR演算子

表2-4およびこの例に示すとおり、一方のオペランドがTRUEである場合、ORはTRUEを戻します。(この例は、例2-34のprint_booleanプロシージャを起動します)。

```
DECLARE
  PROCEDURE print_x_or_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('y', y);
    print_boolean ('x OR y', x OR y);
  END print_x_or_y;

BEGIN
  print_x_or_y (FALSE, FALSE);
  print_x_or_y (TRUE, FALSE);
  print_x_or_y (FALSE, TRUE);
  print_x_or_y (TRUE, TRUE);

  print_x_or_y (TRUE, NULL);
  print_x_or_y (FALSE, NULL);
  print_x_or_y (NULL, TRUE);
  print_x_or_y (NULL, FALSE);
END;
/
```

結果:

```
x = FALSE
y = FALSE
x OR y = FALSE
x = TRUE
y = FALSE
x OR y = TRUE
x = FALSE
y = TRUE
x OR y = TRUE
x = TRUE
y = TRUE
x OR y = TRUE
x = TRUE
y = NULL
x OR y = TRUE
x = FALSE
y = NULL
x OR y = NULL
```

```
x = NULL
y = TRUE
x OR y = TRUE
x = NULL
y = FALSE
x OR y = NULL
```

例2-37 NOT演算子

表2-4およびこの例に示すとおり、オペランドがNULLでないかぎり、NOTはオペランドの反対の値を戻します。NULLは値を持たないため、NOT NULLはNULLを戻します。(この例は、例2-34のprint_booleanプロシージャを起動します)。

```
DECLARE
  PROCEDURE print_not_x (
    x  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('NOT x', NOT x);
  END print_not_x;

BEGIN
  print_not_x (TRUE);
  print_not_x (FALSE);
  print_not_x (NULL);
END;
/
```

結果:

```
x = TRUE
NOT x = FALSE
x = FALSE
NOT x = TRUE
x = NULL
NOT x = NULL
```

例2-38 等しくない比較でのNULL値

この例では、xとyが等しくないため、一連の文が実行されることが予測されます。ただし、NULL値は予測不可能です。そのため、xとyが等しいかどうかは不明です。したがって、IF条件はNULLになり、一連の文は実行されずにバイパスされます。

```
DECLARE
  x NUMBER := 5;
  y NUMBER := NULL;
BEGIN
  IF x != y THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('x != y');  -- not run
  ELSIF x = y THEN -- also yields NULL
    DBMS_OUTPUT.PUT_LINE('x = y');
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Can''t tell if x and y are equal or not.');
  END IF;
END;
```

```
/
```

結果:

```
Can't tell if x and y are equal or not.
```

例2-39 等しい比較でのNULL値

この例では、aとbが等しいため、一連の文が実行されることが予測されます。ただし、等号条件が成立するかどうかは不明であるため、IF条件はNULLになり、一連の文は実行されずにバイパスされます。

```
DECLARE
  a NUMBER := NULL;
  b NUMBER := NULL;
BEGIN
  IF a = b THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('a = b');  -- not run
  ELSIF a != b THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('a != b');  -- not run
  ELSE
    DBMS_OUTPUT.PUT_LINE('Can''t tell if two NULLs are equal');
  END IF;
END;
/
```

結果:

```
Can't tell if two NULLs are equal
```

例2-40 NULLと等しいNOT NULL

この例では、2つのIF文は等しく見えます。ただし、xとyのどちらかがNULLの場合、1番目のIF文はyの値をhighに代入しますが、2番目のIF文はxの値をhighに代入します。

```
DECLARE
  x    INTEGER := 2;
  Y    INTEGER := 5;
  high INTEGER;
BEGIN
  IF (x > y)       -- If x or y is NULL, then (x > y) is NULL
    THEN high := x;  -- run if (x > y) is TRUE
    ELSE high := y;  -- run if (x > y) is FALSE or NULL
  END IF;

  IF NOT (x > y)   -- If x or y is NULL, then NOT (x > y) is NULL
    THEN high := y;  -- run if NOT (x > y) is TRUE
    ELSE high := x;  -- run if NOT (x > y) is FALSE or NULL
  END IF;
END;
/
```

例2-41 論理演算子の演算順序の変更

この例では、例2-34のprint_booleanプロシージャを3回起動します。3回目と1回目の起動は論理的に等しく、3回目の起

動は読みやすさを向上させただけです。2番目の起動では、カッコによって演算の順序が変更されています。

```
DECLARE
  x  BOOLEAN := FALSE;
  y  BOOLEAN := FALSE;

BEGIN
  print_boolean ('NOT x AND y', NOT x AND y);
  print_boolean ('NOT (x AND y)', NOT (x AND y));
  print_boolean ('(NOT x) AND y', (NOT x) AND y);
END;
/
```

結果:

```
NOT x AND y = FALSE
NOT (x AND y) = TRUE
(NOT x) AND y = FALSE
```

2.7.4 短絡評価

論理式を評価するときに、PL/SQLでは短絡評価を使用します。これによって、PL/SQLは結果が判別できた時点でただちに式の評価を停止します。

そのため、評価を続ければエラーになるような式でも書くことができます。

例2-42では、OR式で0(ゼロ)による除算エラーが発生するのを短絡評価によって回避しています。on_handの値が0(ゼロ)の場合、左のオペランドはTRUEになるため、PL/SQLは右のオペランドを評価しません。PL/SQLによりOR演算子を適用する前に両方のオペランドを評価した場合には、右のオペランドは0による除算エラーになります。

例2-42 短絡評価

```
DECLARE
  on_hand  INTEGER := 0;
  on_order INTEGER := 100;
BEGIN
  -- Does not cause divide-by-zero error;
  -- evaluation stops after first expression

  IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
    DBMS_OUTPUT.PUT_LINE('On hand quantity is zero.');
  END IF;
END;
/
```

結果:

```
On hand quantity is zero.
```

2.7.5 比較演算子

比較演算子は式と式を比較します。比較の結果は、常にTRUE、FALSEまたはNULLのいずれかです。

一方の式の値がNULLの場合、比較の結果もNULLになります。

比較演算子は、次のとおりです。

- IS [NOT] NULL演算子
- 関係演算子
- LIKE演算子
- BETWEEN演算子
- IN演算子

ノート:

文字の比較は、実行時に変更可能な NLS パラメータ設定に影響を受けます。したがって、文字の比較は実行時に評価されるので、同じ文字の比較を行っても、タイミングによっては異なる値になることがあります。文字の比較に影響する NLS パラメータの詳細は、『Oracle Database グローバリゼーション・サポート・ガイド』を参照してください。

ノート:

比較演算子に CLOB 値を使用すると、一時的な LOB 値が作成される場合があります。一時表領域がこのような値を処理できる大きさであることを確認してください。

2.7.5.1 IS [NOT] NULL演算子

IS NULL演算子は、オペランドがNULLの場合はBOOLEAN値TRUEを、NULLではない場合はFALSEを戻します。IS NOT NULL演算子の動作は逆です。

NULL値が関係する比較は、常に結果がNULLになります。

値がNULLかどうかをテストするには、IF value IS NULLを使用します。次の例を参照してください。

- 例2-14、デフォルトでNULLに初期化される変数
- 例2-34、BOOLEAN変数を出力するプロシージャ
- 例2-53、WHEN ... IS NULLを使用した検索CASE式

2.7.5.2 関係演算子

この表は、関係演算子の概要を示しています。

表2-5 関係演算子

演算子	意味
=	等しい

演算子	意味
<>, !=, ~=, ^=	等しくない
<	次より小さい
>	次より大きい
<=	<=
>=	以上または同等

ここでのトピック

- 算術比較
- BOOLEAN比較
- 文字比較
- 日付比較

2.7.5.2.1 算術比較

片方の数値がより大きな量を表す場合、その数値はより大きいとみなされます。

実数は近似値として格納されるため、等しいかどうかの比較をすることをお薦めします。

例2-43 式での関係演算子

この例では、例2-35のprint_booleanプロシージャを起動し、関係演算子を使用して算術値を比較する式の値を出力します。

```
BEGIN
  print_boolean ('(2 + 2 =  4)', 2 + 2 = 4);

  print_boolean ('(2 + 2 <> 4)', 2 + 2 <> 4);
  print_boolean ('(2 + 2 != 4)', 2 + 2 != 4);
  print_boolean ('(2 + 2 ~= 4)', 2 + 2 ~= 4);
  print_boolean ('(2 + 2 ^= 4)', 2 + 2  ^= 4);

  print_boolean ('(1 < 2)', 1 < 2);

  print_boolean ('(1 > 2)', 1 > 2);

  print_boolean ('(1 <= 2)', 1 <= 2);

  print_boolean ('(1 >= 1)', 1 >= 1);
END;
/
```

結果:

```
(2 + 2 =  4) = TRUE
(2 + 2 <> 4) = FALSE
(2 + 2 != 4) = FALSE
(2 + 2 ~= 4) = FALSE
(2 + 2 ^= 4) = FALSE
(1 < 2) = TRUE
(1 > 2) = FALSE
(1 <= 2) = TRUE
(1 >= 1) = TRUE
```

2.7.5.2.2 BOOLEAN比較

定義によれば、TRUEはFALSEよりも大きい値を持ちます。NULLとの比較では、NULLが戻されます。

2.7.5.2.3 文字比較

デフォルトでは、片方の文字のバイナリ値がより大きい場合、その文字はより大きいとみなされます。

たとえば、次の式はTRUEになります。

```
'y' > 'r'
```

文字列は文字ごとに比較されます。たとえば、次の式はTRUEになります。

```
'Kathy' > 'Kathryn'
```

初期化パラメータNLS_COMP=ANSIを設定すると、NLS_SORT初期化パラメータで指定された照合順番が文字列比較に使用されます。

照合順番とは、特定の範囲の数値コードが個々の文字に対応している文字セットの内部的な順序のことです。内部的な順番を表す数値が他方の文字より大きい場合、その文字値はより大きいとみなされます。この種の文字が照合順番に使用される場所については、言語ごとに規則が異なる場合があります。たとえば、アクセント記号が付いた文字のソート順序は、バイナリ値が同じであってもデータベース文字セットに応じて異なることがあります。

NLS_SORTパラメータの値を変更すると、大/小文字およびアクセント記号の有無を区別しない比較を実行できます。

大/小文字を区別しない比較では、対応する大文字と小文字が同じ文字として扱われます。たとえば、次の式はTRUEになります。

```
'a' = 'A'
'Alpha' = 'ALPHA'
```

大/小文字を区別せずに比較するには、NLS_SORTパラメータの値の末尾に_CIを付けます(BINARY_CIやXGERMAN_CIなど)。

アクセント記号の有無を区別しない比較では、大/小文字が区別されず、アクセント記号またはデリミタ文字のみが異なる文字も、同じ文字として扱われます。たとえば、次の式はTRUEになります。

```
'Cooperate' = 'Co-Operate'
'Co-Operate' = 'coöperate'
```

大/小文字およびアクセント記号の有無のいずれも区別せずに比較するには、NLS_SORTパラメータの値の末尾に_AIを付けます(BINARY_AIやFRENCH_M_AIなど)。

CHARデータ型とVARCHAR2データ型のセマンティックの違いは、文字の比較に影響します。

詳細は、「値の比較」を参照してください。

2.7.5.2.4 日付比較

片方の日付がより新しい場合、その日付はより大きいとみなされます。

たとえば、次の式はTRUEになります。

```
'01-JAN-91' > '31-DEC-90'
```

2.7.5.3 LIKE演算子

LIKE演算子は、文字、文字列またはCLOB値をパターンと比較し、値がパターンと一致した場合はTRUE、一致しなかった場合はFALSEを戻します。

大/小文字が区別されます。

パターンには、アンダースコア(_)およびパーセント記号(%)の2種類のワイルドカード文字を使用できます。

アンダースコアは1つの文字を表します。

パーセント記号(%)は0(ゼロ)個以上の文字を表します。

パーセント記号やアンダースコアを検索するには、エスケープ文字を定義して、パーセント記号またはアンダースコアの前にそのエスケープ文字を挿入します。

関連項目:

- LIKEの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- LIKEと類似したREGEXP_LIKEの使用の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例2-44 式でのLIKE演算子

この例に示すように、文字列'Johnson'はパターン'J%s_n'に一致しますが、'J%S_N'には一致しません。

```
DECLARE
  PROCEDURE compare (
    value   VARCHAR2,
    pattern VARCHAR2
  ) IS
  BEGIN
    IF value LIKE pattern THEN
      DBMS_OUTPUT.PUT_LINE ('TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('FALSE');
    END IF;
  END;
BEGIN
  compare('Johnson', 'J%s_n');
  compare('Johnson', 'J%S_N');
END;
```

```
/
```

結果:

```
TRUE
FALSE
```

例2-45 パターン内のエスケープ文字

この例では、バックスラッシュをエスケープ文字として使用しているため、文字列内のパーセント記号はワイルドカードとしては機能しません。

```
DECLARE
  PROCEDURE half_off (sale_sign VARCHAR2) IS
  BEGIN
    IF sale_sign LIKE '50¥% off!' ESCAPE '¥' THEN
      DBMS_OUTPUT.PUT_LINE ('TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('FALSE');
    END IF;
  END;
BEGIN
  half_off('Going out of business!');
  half_off('50% off!');
END;
/
```

結果:

```
FALSE
TRUE
```

2.7.5.4 BETWEEN演算子

BETWEEN演算子は、値が指定した範囲内にあるかどうかをテストします。

式x BETWEEN a AND bの値は、式(x>=a) AND (x<=b)の値と同じものとして定義されます。式xは1回のみ評価されます。

関連項目:

BETWEENの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例2-46 式でのBETWEEN演算子

この例では、例2-34のprint_booleanプロシージャを起動し、BETWEEN演算子を含む式の値を出力します。

```
BEGIN
  print_boolean ('2 BETWEEN 1 AND 3', 2 BETWEEN 1 AND 3);
  print_boolean ('2 BETWEEN 2 AND 3', 2 BETWEEN 2 AND 3);
  print_boolean ('2 BETWEEN 1 AND 2', 2 BETWEEN 1 AND 2);
  print_boolean ('2 BETWEEN 3 AND 4', 2 BETWEEN 3 AND 4);
END;
```

```
/
```

結果:

```
2 BETWEEN 1 AND 3 = TRUE
2 BETWEEN 2 AND 3 = TRUE
2 BETWEEN 1 AND 2 = TRUE
2 BETWEEN 3 AND 4 = FALSE
```

2.7.5.5 IN演算子

IN演算子は、セット・メンバーシップをテストします。

x IN (set)は、xがsetのメンバーに等しい場合にのみTRUEを戻します。

関連項目:

INの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例2-47 式でのIN演算子

この例では、例2-34のprint_booleanプロシージャを起動し、IN演算子を含む式の値を出力します。

```
DECLARE
  letter VARCHAR2(1) := 'm';
BEGIN
  print_boolean (
    'letter IN (''a'', ''b'', ''c'')',
    letter IN ('a', 'b', 'c')
  );
  print_boolean (
    'letter IN (''z'', ''m'', ''y'', ''p'')',
    letter IN ('z', 'm', 'y', 'p')
  );
END;
/
```

結果:

```
letter IN ('a', 'b', 'c') = FALSE
letter IN ('z', 'm', 'y', 'p') = TRUE
```

例2-48 NULL値を含む集合でのIN演算子

この例は、setにNULL値が含まれている場合に発生する状況を示しています。これは、例2-34のprint_booleanプロシージャを起動します。

```
DECLARE
  a INTEGER; -- Initialized to NULL by default
  b INTEGER := 10;
  c INTEGER := 100;
```

```
BEGIN
  print_boolean ('100 IN (a, b, c)', 100 IN (a, b, c));
  print_boolean ('100 NOT IN (a, b, c)', 100 NOT IN (a, b, c));

  print_boolean ('100 IN (a, b)', 100 IN (a, b));
  print_boolean ('100 NOT IN (a, b)', 100 NOT IN (a, b));

  print_boolean ('a IN (a, b)', a IN (a, b));
  print_boolean ('a NOT IN (a, b)', a NOT IN (a, b));
END;
/
```

結果:

```
100 IN (a, b, c) = TRUE
100 NOT IN (a, b, c) = FALSE
100 IN (a, b) = NULL
100 NOT IN (a, b) = NULL
a IN (a, b) = NULL
a NOT IN (a, b) = NULL
```

2.7.6 BOOLEAN式

BOOLEAN式とは、BOOLEAN値(TRUE、FALSEまたはNULL)を戻す式のことです。

最も単純なBOOLEAN式は、1個のBOOLEANリテラル、定数または変数です。次のものもBOOLEAN式です。

```
NOT boolean_expression
boolean_expression relational_operator boolean_expression
boolean_expression { AND | OR } boolean_expression
```

関係演算子のリストは、表2-5を参照してください。BOOLEAN式の構文の詳細は、「boolean_expression ::=」を参照してください。

通常、BOOLEAN式は、制御文(「PL/SQLの制御文」を参照)およびDML文のWHERE句で条件として使用します。

BOOLEAN変数自体を条件として使用できますが、TRUE値またはFALSE値と比較する必要はありません。

例2-49 等価なBOOLEAN式

この例では、ループの条件は等価です。

```
DECLARE
  done BOOLEAN;
BEGIN
  -- These WHILE loops are equivalent

  done := FALSE;
  WHILE done = FALSE
    LOOP
      done := TRUE;
    END LOOP;

  done := FALSE;
  WHILE NOT (done = TRUE)
```

```
    LOOP
      done := TRUE;
    END LOOP;

  done := FALSE;
  WHILE NOT done
    LOOP
      done := TRUE;
    END LOOP;
END;
/
```

2.7.7 CASE式

ここでのトピック

- 単純なCASE式
- 検索CASE式

2.7.7.1 単純なCASE式

この説明では、単純なCASE式の構文が次のような構文であるとします。

```
CASE selector
WHEN selector_value_1 THEN result_1
WHEN selector_value_2 THEN result_2
...
WHEN selector_value_n THEN result_n
[ ELSE
  else_result ]
END
```

selectorは式です(通常は1つの変数)。selector_valueとresultは、それぞれリテラルまたは式にすることができます。少なくとも1つのresultはリテラルのNULLではない必要があります。

単純なCASE式は、selector_valueがselectorと一致した最初のresultを戻します。残りの式は評価されません。selectorと一致するselector_valueがない場合、CASE式は、else_resultが存在するとそれを戻し、存在しないとNULLを戻します。

関連項目:

構文の詳細は、「simple_case_expression ::=」を参照してください

例2-50 単純なCASE式

この例では、単純なCASE式の値を変数appraisalに代入します。selectorはgradeです。

```
DECLARE
  grade CHAR(1) := 'B';
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
```

```
    CASE grade
      WHEN 'A' THEN 'Excellent'
      WHEN 'B' THEN 'Very Good'
      WHEN 'C' THEN 'Good'
      WHEN 'D' THEN 'Fair'
      WHEN 'F' THEN 'Poor'
      ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

結果:

```
Grade B is Very Good
```

例2-51 WHEN NULLを使用した単純なCASE式

この例に示すように、selectorの値がNULLの場合、WHEN NULLでは一致させることができません。

そのかわりに、例2-53に示すように、検索CASE式をWHEN boolean_expression IS NULLとともに使用します。

```
DECLARE
  grade CHAR(1); -- NULL by default
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
  CASE grade
    WHEN NULL THEN 'No grade assigned'
    WHEN 'A' THEN 'Excellent'
    WHEN 'B' THEN 'Very Good'
    WHEN 'C' THEN 'Good'
    WHEN 'D' THEN 'Fair'
    WHEN 'F' THEN 'Poor'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

結果:

```
Grade  is No such grade
```

2.7.7.2 検索CASE式

この説明では、検索CASE式の構文は次のような構文であるとします。

```
CASE
WHEN boolean_expression_1 THEN result_1
WHEN boolean_expression_2 THEN result_2
...
WHEN boolean_expression_n THEN result_n
[ ELSE
  else_result ]
END]
```

検索CASE式は、boolean_expressionがTRUEになる最初のresultを戻します。残りの式は評価されません。TRUEになるboolean_expressionがない場合、CASE式は、else_resultが存在するとそれを戻し、存在しないとNULLを戻します。

関連項目:

構文の詳細は、「searched_case_expression ::=」を参照してください

例2-52 検索CASE式

この例では、検索CASE式の値を変数appraisalに代入します。

```
DECLARE
  grade      CHAR(1) := 'B';
  appraisal  VARCHAR2(120);
  id         NUMBER  := 8429862;
  attendance NUMBER := 150;
  min_days   CONSTANT NUMBER := 200;

  FUNCTION attends_this_school (id NUMBER)
    RETURN BOOLEAN IS
  BEGIN
    RETURN TRUE;
  END;
BEGIN
  appraisal :=
  CASE
    WHEN attends_this_school(id) = FALSE
      THEN 'Student not enrolled'
    WHEN grade = 'F' OR attendance < min_days
      THEN 'Poor (poor performance or bad attendance)'
    WHEN grade = 'A' THEN 'Excellent'
    WHEN grade = 'B' THEN 'Very Good'
    WHEN grade = 'C' THEN 'Good'
    WHEN grade = 'D' THEN 'Fair'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE
    ('Result for student ' || id || ' is ' || appraisal);
END;
/
```

結果:

```
Result for student 8429862 is Poor (poor performance or bad attendance)
```

例2-53 WHEN ... IS NULLを使用した検索CASE式

この例では、検索CASE式を使用して、例2-51の問題を解決します。

```
DECLARE
  grade CHAR(1); -- NULL by default
  appraisal VARCHAR2(20);
BEGIN
```

```
  appraisal :=
    CASE
      WHEN grade IS NULL THEN 'No grade assigned'
      WHEN grade = 'A' THEN 'Excellent'
      WHEN grade = 'B' THEN 'Very Good'
      WHEN grade = 'C' THEN 'Good'
      WHEN grade = 'D' THEN 'Fair'
      WHEN grade = 'F' THEN 'Poor'
      ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

結果:

```
Grade  is No grade assigned
```

2.7.8 PL/SQL式でのSQLファンクション

PL/SQL式では、次のファンクションを除くすべてのSQLファンクションを使用できます。

- 集計ファンクション(AVG、COUNTなど)
- 分析ファンクション(LAG、RATIO_TO_REPORTなど)
- データ・マイニング・ファンクション(CLUSTER_ID、FEATURE_VALUEなど)
- エンコーディング・ファンクションおよびデコーディング・ファンクション(DECODE、DUMPなど)
- モデル・ファンクション(ITERATION_NUMBER、PREVIOUSなど)
- オブジェクト参照ファンクション(REF、VALUEなど)
- XMLファンクション
- 次の変換ファンクション:
    - BIN_TO_NUM
- 次のJSON SQL演算子:
    - JSON_ARRAYAGG
    - JSON_EQUAL
    - JSON_OBJECTAGG
    - JSON_TABLE
    - JSON_TEXTCONTAINS
- 次の照合SQL演算子およびファンクション:
    - COLLATE
    - COLLATION
    - NLS_COLLATION_ID
    - NLS_COLLATION_NAME
- その他の該当するファンクション:

- CUBE_TABLE
- DATAOBJ_TO_PARTITION
- LNNVL
- NVL2
- SYS_CONNECT_BY_PATH
- SYS_TYPEID
- WIDTH_BUCKET

PL/SQLでは、引数および結果がBINARY_INTEGERであるBITANDのオーバーロードがサポートされています。

PL/SQL式でRAWTOHEXファンクションを使用すると、データ型RAWの引数が受け入れられ、その引数の値を構成するバイトの16進表現でVARCHAR2値が戻されます。RAW以外の型の引数は、RAWに暗黙的に変換できる場合にのみ指定できます。この変換は、HEXTORAWファンクションの有効な引数であるCHAR、VARCHAR2およびLONGの値と、最大16380バイトのLONG RAWおよびBLOBの値に対して実行可能です。

2.7.9 静的な式

静的な式は、値がコンパイル時に決定される式です。文字の比較、変数またはファンクションの起動は含まれていません。静的な式は、条件付きコンパイル・ディレクティブで使用できる唯一の式です。

静的な式の定義

- 式がNULLリテラルである場合、その式は静的です。
- 式が文字、数値またはブールのリテラルである場合、その式は静的です。
- 式が静的定数への参照である場合、その式は静的です。
- 式が$$で始まる条件付きコンパイル変数への参照である場合、その式は静的です。
- 演算子が静的な式で許可されていて、そのオペランドがすべて静的であり、そのオペランドに対する評価時に演算子による例外が発生しない場合、その式は静的です。

表2-6 静的な式で許可される演算子

演算子	演算子のカテゴリ
()	式のデリミタ
**	指数
*, /,+, -	除算、乗算、加算(または正)、減算(または負)の算術演算子
=, !=, <, <=, >=, > IS [NOT] NULL	比較演算子
NOT	論理演算子

演算子	演算子のカテゴリ
[NOT] LIKE, [NOT] LIKE2, [NOT] LIKE4, [NOT] LIKEC	パターン一致演算子
XOR	二項演算子

このリストに、静的な式で許可されるファンクションを示します。

- ABS
- ACOS
- ASCII
- ASCIISTR
- ASIN
- ATAN
- ATAN2
- BITAND
- CEIL
- CHR
- COMPOSE
- CONVERT
- COS
- COSH
- DECOMPOSE
- EXP
- FLOOR
- HEXTORAW
- INSTR
- INSTRB
- INSTRC
- INSTR2
- INSTR4
- IS [NOT] INFINITE
- IS [NOT] NAN
- LENGTH

- LENGTH2
- LENGTH4
- LENGTHB
- LENGTHC
- LN
- LOG
- LOWER
- LPAD
- LTRIM
- MOD
- NVL
- POWER
- RAWTOHEX
- REM
- REMAINDER
- REPLACE
- ROUND
- RPAD
- RTRIM
- SIGN
- SIN
- SINH
- SQRT
- SUBSTR
- SUBSTR2
- SUBSTR4
- SUBSTRB
- SUBSTRC
- TAN
- TANH
- TO_BINARY_DOUBLE
- TO_BINARY_FLOAT

- TO_CHAR
- TO_NUMBER
- TRIM
- TRUNC
- UPPER

静的な式は、次のサブタイプの宣言で使用できます。

- 文字列型(VARCHAR2、NCHAR、CHAR、NVARCHAR2、RAW、およびANSIの同等の型)の長さ
- NUMBER型およびサブタイプ(FLOATなど)の位取りおよび精度
- 期間型の精度(年、月、秒)
- 時刻およびタイムスタンプの精度
- VARRAYの境界
- 型の宣言における範囲の境界

いずれの場合も、静的な式の結果の型は、宣言された項目のサブタイプと同じであることに加え、コンテキストに対して適切な範囲にある必要があります。

ここでのトピック

- 静的なPLS_INTEGER式
- 静的なBOOLEAN式
- 静的なVARCHAR2式
- 静的定数

関連項目:

式の一般情報は、「式」を参照してください

2.7.9.1 静的なPLS_INTEGER式

静的なPLS_INTEGER式には次のものがあります。

- PLS_INTEGERリテラル

  リテラルの詳細は、「リテラル」を参照してください。
- PLS_INTEGER静的定数

  静的定数の詳細は、「静的定数」を参照してください。
- NULL

関連項目:

PLS_INTEGERデータ型の詳細は、「PLS_INTEGERおよびBINARY_INTEGERデータ型」を参照してください

2.7.9.2 静的なBOOLEAN式

静的なBOOLEAN式には次のものがあります。

- BOOLEANリテラル(TRUE、FALSEまたはNULL)
- BOOLEAN静的定数

  静的定数の詳細は、「静的定数」を参照してください。
- xおよびyは、静的なPLS_INTEGER式です。
  - x > y
  - x < y
  - x >= y
  - x <= y
  - x = y
  - x <> y

  静的なPLS_INTEGER式の詳細は、「静的なPLS_INTEGER式」を参照してください。
- xおよびyはBOOLEAN式です。
  - NOT y
  - x AND y
  - x OR y
  - x > y
  - x >= y
  - x = y
  - x <= y
  - x <> y

  静的なBOOLEAN式の詳細は、「ブール式」を参照してください。
- xは静的な式です。
  - x IS NULL
  - x IS NOT NULL

  静的な式の詳細は、「静的な式」を参照してください。

関連項目:

BOOLEANデータ型の詳細は、「ブール・データ型」を参照してください

2.7.9.3 静的なVARCHAR2式

静的なVARCHAR2式には次のものがあります。

- 最大サイズが32,767バイトである文字列リテラル

  リテラルの詳細は、「リテラル」を参照してください。

- NULL
- TO_CHAR(x)(xは静的なPLS_INTEGER式)

  TO_CHARファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- TO_CHAR(x, f, n)(xは静的なPLS_INTEGER式で、fとnは静的なVARCHAR2式)

  TO_CHARファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- x || y(xおよびyは、静的なVARCHAR2式またはPLS_INTEGER式)

  静的なPLS_INTEGER式の詳細は、「静的なPLS_INTEGER式」を参照してください。

関連項目:

VARCHAR2データ型の詳細は、「CHARおよびVARCHAR2変数」を参照してください

2.7.9.4 静的定数

静的定数は、パッケージ仕様部で次の構文を使用して宣言します。

```
constant_name CONSTANT data_type := static_expression;
```

static_expressionの型は、data_type(BOOLEANまたはPLS_INTEGER)と同じである必要があります。

静的定数は、package_nameパッケージの本体内にある場合でも、常にpackage_name.constant_nameとして参照する必要があります。

PL/SQLユニットの条件付きコンパイル・ディレクティブのBOOLEAN式でconstant_nameを使用する場合、そのPL/SQLユニットはパッケージpackage_nameに依存します。パッケージ仕様部を変更すると、依存するPL/SQLユニットは無効になり、再コンパイルが必要になる場合があります(依存オブジェクトの無効化の詳細は、『Oracle Database開発ガイド』を参照)。

複数のPL/SQLユニットの条件付きコンパイルを制御する静的定数を持つパッケージを使用する場合は、パッケージ仕様部のみを作成し、そのパッケージ仕様部を条件付きコンパイルの制御専用にすることをお薦めします。この方法で、パッケージ仕様部の変更による無効化が最小限に抑えられます。

単一のPL/SQLユニットの条件付きコンパイルを制御する場合は、PLSQL_CCFLAGSコンパイル・パラメータにフラグを設定できます。このパラメータの詳細は、「問合せディレクティブへの値の代入」および『Oracle Databaseリファレンス』を参照してください。

関連項目:

- 定数の宣言の一般情報は、「定数の宣言」を参照してください
- パッケージの詳細は、「PL/SQLパッケージ」を参照してください
- スキーマ・オブジェクトの依存性の詳細は、『Oracle Database開発ガイド』を参照してください。

例2-54 静的定数

この例では、パッケージmy_debugで静的定数debugおよびtraceを定義し、複数のPL/SQLユニットでのデバッグおよびトレース

を制御します。プロシージャmy_proc1ではdebugのみが使用され、プロシージャmy_proc2ではtraceのみが使用されますが、いずれのプロシージャもこのパッケージに依存しています。ただし、再コンパイルされるコードが異ならない場合もあります。たとえば、debugの値をFALSEに変更しただけで2つのプロシージャを再コンパイルすると、my_proc1でコンパイルされるコードは変更されますが、my_proc2でコンパイルされるコードは変更されません。

```
CREATE PACKAGE my_debug IS
  debug CONSTANT BOOLEAN := TRUE;
  trace CONSTANT BOOLEAN := TRUE;
END my_debug;
/

CREATE PROCEDURE my_proc1 AUTHID DEFINER IS
BEGIN
  $IF my_debug.debug $THEN
    DBMS_OUTPUT.put_line('Debugging ON');
  $ELSE
    DBMS_OUTPUT.put_line('Debugging OFF');
  $END
END my_proc1;
/

CREATE PROCEDURE my_proc2 AUTHID DEFINER IS
BEGIN
  $IF my_debug.trace $THEN
    DBMS_OUTPUT.put_line('Tracing ON');
  $ELSE
    DBMS_OUTPUT.put_line('Tracing OFF');
  $END
END my_proc2;
/
```

2.8 エラー・レポート・ファンクション

PL/SQLには、PL/SQL例外処理コードで使用するためのSQLCODEとSQLERRMの2つのエラー・レポート・ファンクションがあります。

詳細は、「SQLCODEファンクション」および「SQLERRMファンクション」を参照してください。

SQLCODEまたはSQLERRMファンクションは、SQL文では使用できません。

2.9 条件付きコンパイル

条件付きコンパイルを使用すると、ソース・テキストを削除することなく、PL/SQLアプリケーションの機能をカスタマイズできます。

たとえば、次のことが可能です。

- 最新のデータベース・リリースで新機能を使用し、古いデータベース・リリースでアプリケーションを実行する場合にそれらの新機能を無効化することができます。
- 開発環境でデバッグ文またはトレース文をアクティブ化し、本番サイトでアプリケーションを実行する場合にそれらの文を隠ぺいすることができます。

ここでのトピック

- 条件付きコンパイルの動作方法
- 条件付きコンパイルの例
- 処理後のソース・テキストの取得と出力
- 条件付きコンパイル・ディレクティブの制限

2.9.1 条件付きコンパイルの動作方法

条件付きコンパイルでは、IF文に似た選択ディレクティブを使用して、コンパイルするソース・テキストを選択します。

通常、選択ディレクティブの条件には問合せディレクティブが含まれています。エラー・ディレクティブは、ユーザー定義のエラーを呼び出します。すべての条件付きコンパイル・ディレクティブは、プリプロセッサ制御トークンとPL/SQLテキストで構成されています。

ここでのトピック

- プリプロセッサ制御トークン
- 選択ディレクティブ
- エラー・ディレクティブ
- 問合せディレクティブ
- DBMS_DB_VERSIONパッケージ

関連項目:

静的な式

2.9.1.1 プリプロセッサ制御トークン

プリプロセッサ制御トークンによって、PL/SQLユニットがコンパイルされる前に処理されるコードが識別されます。

構文

```
$plsql_identifier
```

$とplsql_identifierの間に空白を入れることはできません。

文字$はplsql_identifier内でも使用できますが、そこでは特別な意味を持ちません。

次のプリプロセッサ制御トークンは予約されています。

- $IF
- $THEN
- $ELSE
- $ELSIF
- $ERROR

plsql_identifierの詳細は、「識別子」を参照してください。

2.9.1.2 選択ディレクティブ

選択ディレクティブは、コンパイルするソース・テキストを選択します。

構文

```
$IF boolean_static_expression $THEN
   text
[ $ELSIF boolean_static_expression $THEN
   text
]...
[ $ELSE
   text
$END
]
```

boolean_static_expressionの構文の詳細は、「静的なブール式」を参照してください。textは任意ですが、通常は文(「statement ::=」を参照)またはエラー・ディレクティブ(「エラー・ディレクティブ」を参照)です。

選択ディレクティブは、いずれかの静的なBOOLEAN式の値がTRUEであるか、またはすべての式が処理されるまで、使用されている順に式を評価します。いずれかの式の値がTRUEの場合、そのテキストがコンパイルされますが、残りの式は評価されず、そのテキストは分析されません。値がTRUEの式がなく、$ELSEが存在する場合は、そのテキストがコンパイルされますが、存在しない場合は、どのテキストもコンパイルされません。

選択ディレクティブの例は、「条件付きコンパイルの例」を参照してください。

関連項目:

選択ディレクティブと同じロジックを持つIF文の詳細は、「条件付き選択文」を参照してください

2.9.1.3 エラー・ディレクティブ

エラー・ディレクティブは、コンパイル時にユーザー定義のエラー・メッセージを生成します。

構文

```
$ERROR varchar2_static_expression $END
```

次のコンパイル時のエラー・メッセージを生成します。stringはvarchar2_static_expressionの値です。

```
PLS-00179: $ERROR: string
```

varchar2_static_expressionの構文の詳細は、「静的なVARCHAR2式」を参照してください。

エラー・ディレクティブの使用例は、例2-58を参照してください。

2.9.1.4 問合せディレクティブ

問合せディレクティブは、コンパイル環境の情報を提供します。

構文

```
$$name
```

name(引用符で囲まれていないPL/SQL識別子)の詳細は、「識別子」を参照してください。

通常、問合せディレクティブは、選択ディレクティブのboolean_static_expressionで使用しますが、その型の変数またはリテラルを使用できるすべての場所で使用できます。また、通常のPL/SQLでは(変数ではなく)リテラルのみ使用可能な場所でも使用できます(たとえば、VARCHAR2変数のサイズを指定するために使用できます)。

ここでのトピック

- 事前定義の問合せディレクティブ
- 問合せディレクティブへの値の代入
- 解決できない問合せディレクティブ

2.9.1.4.1 事前定義の問合せディレクティブ

事前定義の問合せディレクティブは、次のとおりです。

- $$PLSQL_LINE

  現行のPL/SQLユニットでディレクティブが使用されるソース行の番号を値として持つPLS_INTEGERリテラル。選択ディレクティブの$$PLSQL_LINEの例を次に示します。

  ```
  $IF $$PLSQL_LINE = 32 $THEN ...
  ```

- $$PLSQL_UNIT

  現行のPL/SQLユニットの名前を持つVARCHAR2リテラル。現行のPL/SQLユニットが無名ブロックの場合、$$PLSQL_UNITの値はNULL値になります。

- $$PLSQL_UNIT_OWNER

  現行のPL/SQLユニットの所有者を持つVARCHAR2リテラル。現行のPL/SQLユニットが無名ブロックの場合、$$PLSQL_UNIT_OWNERの値はNULL値になります。

- $$PLSQL_UNIT_TYPE

  現行のPL/SQLユニットのタイプを持つVARCHAR2リテラル。ANONYMOUS BLOCK、FUNCTION、PACKAGE、PACKAGE BODY、PROCEDURE、TRIGGER、TYPEまたはTYPE BODYのいずれか。無名ブロックまたは非DMLトリガー内では、$$PLSQL_UNIT_TYPEの値はANONYMOUS BLOCKになります。

- $$plsql_compilation_parameter

  名前plsql_compilation_parameterは、PL/SQLコンパイル・パラメータ(PLSCOPE_SETTINGSなど)です。これらのパラメータの詳細は、表1-2を参照してください。

選択ディレクティブには静的なBOOLEAN式が必要なため、VARCHAR2比較で次のような$$PLSQL_UNIT、$$PLSQL_UNIT_OWNERまたは$$PLSQL_UNIT_TYPEは使用できません。

```
$IF $$PLSQL_UNIT = 'AWARD_BONUS' $THEN ...
$IF $$PLSQL_UNIT_OWNER IS HR $THEN ...
$IF $$PLSQL_UNIT_TYPE IS FUNCTION $THEN ...
```

ただし、前述のディレクティブをNULLと比較することはできます。たとえば:

```
$IF $$PLSQL_UNIT IS NULL $THEN ...
```

```
$IF $$PLSQL_UNIT_OWNER IS NOT NULL $THEN ...
$IF $$PLSQL_UNIT_TYPE IS NULL $THEN ...
```

例2-55 事前定義の問合せディレクティブ

この例は、SQL*Plusスクリプトでは、複数の事前定義の問合せディレクティブをPLS_INTEGERおよびVARCHAR2リテラルとして使用し、値がどのように代入されるかを示しています。

```
SQL> CREATE OR REPLACE PROCEDURE p
  2  AUTHID DEFINER IS
  3    i PLS_INTEGER;
  4  BEGIN
  5    DBMS_OUTPUT.PUT_LINE('Inside p');
  6    i := $$PLSQL_LINE;
  7    DBMS_OUTPUT.PUT_LINE('i = ' || i);
  8    DBMS_OUTPUT.PUT_LINE('$$PLSQL_LINE = ' || $$PLSQL_LINE);
  9    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT = ' || $$PLSQL_UNIT);
 10    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_OWNER = ' || $$PLSQL_UNIT_OWNER);
 11    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_TYPE = ' || $$PLSQL_UNIT_TYPE);
 12  END;
 13  /

Procedure created.

SQL> BEGIN
  2    p;
  3    DBMS_OUTPUT.PUT_LINE('Outside p');
  4    DBMS_OUTPUT.PUT_LINE('$$PLSQL_LINE = ' || $$PLSQL_LINE);
  5    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT = ' || $$PLSQL_UNIT);
  6    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_OWNER = ' || $$PLSQL_UNIT_OWNER);
  7    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_TYPE = ' || $$PLSQL_UNIT_TYPE);
  8  END;
  9  /
```

結果:

```
Inside p
i = 6
$$PLSQL_LINE = 8
$$PLSQL_UNIT = P
$$PLSQL_UNIT_OWNER = HR
$$PLSQL_UNIT_TYPE = PROCEDURE
Outside p
$$PLSQL_LINE = 4
$$PLSQL_UNIT =
$$PLSQL_UNIT_OWNER =
$$PLSQL_UNIT_TYPE = ANONYMOUS BLOCK

PL/SQL procedure successfully completed.
```

例2-56 PL/SQLのコンパイル・パラメータの値の表示

この例では、PL/SQLコンパイル・パラメータの現在の値を表示します。

ノート:

SQL*Plus 環境では、SHOW PARAMETERS コマンドを使用して、PL/SQL コンパイル・パラメータを含む初期化パラメータの現在の値を表示できます。SHOW コマンドと PARAMETERS オプションの詳細は、『SQL*Plus ユーザーズ・ガイドおよびリファレンス』を参照してください。

```
BEGIN
DBMS_OUTPUT.PUT_LINE('$$PLSCOPE_SETTINGS = ' || $$PLSCOPE_SETTINGS);
DBMS_OUTPUT.PUT_LINE('$$PLSQL_CCFLAGS = ' || $$PLSQL_CCFLAGS);
DBMS_OUTPUT.PUT_LINE('$$PLSQL_CODE_TYPE = ' || $$PLSQL_CODE_TYPE);
DBMS_OUTPUT.PUT_LINE('$$PLSQL_OPTIMIZE_LEVEL = ' || $$PLSQL_OPTIMIZE_LEVEL);
DBMS_OUTPUT.PUT_LINE('$$PLSQL_WARNINGS = ' || $$PLSQL_WARNINGS);
DBMS_OUTPUT.PUT_LINE('$$NLS_LENGTH_SEMANTICS = ' || $$NLS_LENGTH_SEMANTICS);
END; /
```

結果:

```
$$PLSCOPE_SETTINGS = IDENTIFIERS:NONE
$$PLSQL_CCFLAGS =
$$PLSQL_CODE_TYPE = INTERPRETED
$$PLSQL_OPTIMIZE_LEVEL = 2
$$PLSQL_WARNINGS = ENABLE:ALL
$$NLS_LENGTH_SEMANTICS = BYTE
```

2.9.1.4.2 問合せディレクティブへの値の代入

PLSQL_CCFLAGSコンパイル・パラメータを使用して、問合せディレクティブに値を代入できます。

たとえば:

```
ALTER SESSION SET PLSQL_CCFLAGS =
  'name1:value1, name2:value2, ... namen:valuen'
```

それぞれのvalueは、BOOLEANリテラル(TRUE、FALSEまたはNULL)、あるいはPLS_INTEGERリテラルである必要があります。valueのデータ型によって、nameのデータ型が決まります。

同じデータ型または異なるデータ型の値を指定して同じnameを複数回使用できます。先に行った代入は、後で行った代入によってオーバーライドされます。たとえば、次のコマンドでは、$$flagの値は5、データ型はPLS_INTEGERに設定されます。

```
ALTER SESSION SET PLSQL_CCFLAGS = 'flag:TRUE, flag:5'
```

コンパイル・パラメータを含む事前定義の問合せディレクティブに値を代入する場合、PLSQL_CCFLAGSは使用しないことをお薦めします。コンパイル・パラメータに値を代入する場合は、ALTER SESSION文を使用することをお薦めします。

ALTER SESSION文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

ノート:

PLSQL_CCFLAGS のコンパイル時の値は、ストアド PL/SQL ユニットのメタデータとともに格納されているため、ユニットを明示的に再コンパイルするときにその値を再利用できます。詳細は、「PL/SQL ユニットおよびコンパイル・パラメータ」を参照してください。

PLSQL_CCFLAGSの詳細は、『Oracle Databaseリファレンス』を参照してください。

例2-57 PLSQL_CCFLAGSによるそれ自体への値の代入

この例では、PLSQL_CCFLAGSを使用して、ユーザー定義の問合せディレクティブ$$Some_Flagに値を代入し、さらに、(推奨されてはいませんが)それ自体にも値を代入します。先に行った代入は後で行った代入によってオーバーライドされるため、$$Some_Flagの結果の値は2になり、PLSQL_CCFLAGSの結果の値はALTER SESSION文によって代入される値('Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99')ではなく、それ自体に代入される値(99)になります。

```
ALTER SESSION SET
PLSQL_CCFlags = 'Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99'
/
BEGIN
  DBMS_OUTPUT.PUT_LINE($$Some_Flag);
  DBMS_OUTPUT.PUT_LINE($$PLSQL_CCFlags);
END;
/
```

結果:

```
2
99
```

2.9.1.4.3 解決できない問合せディレクティブ

ソース・テキストがラップされていない場合、問合せディレクティブの値を判別できないと、PL/SQLによって警告が発行されます。

問合せディレクティブ($$name)を解決できず、ソース・テキストがラップされていない場合は、PL/SQLにより警告PLW-6003が発行され、未解決の問合せディレクティブの値としてNULLが代入されます。ソース・テキストがラップされている場合は、未解決の問合せディレクティブが示されないように、警告メッセージが無効になります。

PL/SQLソース・テキストのラップの詳細は、「PL/SQLのソース・テキストのラップ」を参照してください。

2.9.1.5 DBMS_DB_VERSIONパッケージ

DBMS_DB_VERSIONパッケージでは、Oracleバージョン番号、およびOracleのバージョンに基づいて条件付きコンパイルを簡単に選択する場合に有効なその他の情報を指定します。

DBMS_DB_VERSIONパッケージでは、次の静的定数が提供されます。

- PLS_INTEGER定数VERSIONは、現行のOracle Databaseのバージョンを識別します。
- PLS_INTEGER定数RELEASEは、現行のOracle Databaseのリリース番号を識別します。
- VER_LE_v形式のBOOLEAN定数の値はそれぞれ、データベース・バージョンがv以下の場合にTRUEになり、それ以外の場合はFALSEになります。
- VER_LE_v_r形式のBOOLEAN定数の値はそれぞれ、データベース・バージョンがv以下でリリースがr以下の場合にTRUE

になり、それ以外の場合はFALSEになります。

DBMS_DB_VERSIONパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

2.9.2 条件付きコンパイルの例

選択およびユーザー定義の問合せディレクティブを使用する条件付きコンパイルの例。

例2-58 データベース・バージョンを確認するコード

この例では、データベース・バージョンおよびリリースがOracle Database 10gリリース2より前の場合はエラー・メッセージを生成し、それ以外の場合は、バージョンおよびリリースがサポート対象であるというメッセージを表示してOracle Database 10gリリース2から使用可能になったCOMMIT文を使用します。

```
BEGIN
  $IF DBMS_DB_VERSION.VER_LE_10_1 $THEN  -- selection directive begins
    $ERROR 'unsupported database release' $END  -- error directive
  $ELSE
    DBMS_OUTPUT.PUT_LINE (
      'Release ' || DBMS_DB_VERSION.VERSION || '.' ||
      DBMS_DB_VERSION.RELEASE || ' is supported.'
    );
  -- This COMMIT syntax is newly supported in 10.2:
  COMMIT WRITE IMMEDIATE NOWAIT;
  $END  -- selection directive ends
END;
/
```

結果:

```
Release 12.1 is supported.
```

例2-59 異なるデータベース・バージョンに対する異なるコードのコンパイル

この例では、ユーザー定義の問合せディレクティブ$$my_debugおよび$$my_tracingに値を設定し、条件付きコンパイルを使用します。

- パッケージmy_pkgの仕様部で、サブタイプmy_realのベース型を判断します(BINARY_DOUBLEは、Oracle Databaseバージョン10g以上でのみ使用できます)。
- パッケージmy_pkgの本体で、異なるデータベース・バージョンに対してmy_piおよびmy_eの値を別々に計算します
- プロシージャcircle_areaで、問合せディレクティブ$$my_debugの値がTRUEの場合のみに一部のコードをコンパイルします。

```
ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE, my_tracing:FALSE';

CREATE OR REPLACE PACKAGE my_pkg AUTHID DEFINER AS
  SUBTYPE my_real IS
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN
      NUMBER;
    $ELSE
      BINARY_DOUBLE;
```

```
    $END

  my_pi my_real;
  my_e  my_real;
END my_pkg;
/

CREATE OR REPLACE PACKAGE BODY my_pkg AS
BEGIN
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
    my_pi := 3.14159265358979323846264338327950288420;
    my_e  := 2.71828182845904523536028747135266249775;
  $ELSE
    my_pi := 3.14159265358979323846264338327950288420d;
    my_e  := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/

CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real) AUTHID DEFINER IS
  my_area       my_pkg.my_real;
  my_data_type  VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * (radius**2);

  DBMS_OUTPUT.PUT_LINE
    ('Radius: ' || TO_CHAR(radius) || ' Area: ' || TO_CHAR(my_area));

  $IF $$my_debug $THEN
    SELECT DATA_TYPE INTO my_data_type
    FROM USER_ARGUMENTS
    WHERE OBJECT_NAME = 'CIRCLE_AREA'
    AND ARGUMENT_NAME = 'RADIUS';

    DBMS_OUTPUT.PUT_LINE
      ('Data type of the RADIUS argument is: ' || my_data_type);
  $END
END;
/
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
 ('PACKAGE', 'HR', 'MY_PKG');
```

結果:

```
PACKAGE my_pkg AUTHID DEFINER AS
SUBTYPE my_real IS
BINARY_DOUBLE;
my_pi my_real;
my_e my_real;
END my_pkg;

Call completed.
```

2.9.3 処理後のソース・テキストの取得と出力

DBMS_PREPROCESSORパッケージでは、処理後の形式でPL/SQLのソース・テキストを取得して出力するサブプログラムが提供さ

れます。

DBMS_PREPROCESSORパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

例2-60 処理後のソース・テキストの表示

この例では、DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCEプロシージャを起動して、「例2-59」のmy_pkgの処理後の形式を印刷します。処理後のテキストに含まれない「例2-59」のコードの行は、空白行として表示されます。

```
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
  'PACKAGE', 'HR', 'MY_PKG'
);
```

結果:

```
PACKAGE my_pkg AUTHID DEFINERs AS
SUBTYPE my_real IS
BINARY_DOUBLE;
my_pi my_real;
my_e my_real;
END my_pkg;
```

2.9.4 条件付きコンパイル・ディレクティブの制限

条件付きコンパイル・ディレクティブは、これらのセマンティクス制限の対象となります。

条件付きコンパイル・ディレクティブは、(「CREATE TYPE文」で作成した)スキーマ・レベルのユーザー定義型では使用できません。次の型仕様部で、依存する型の属性構造と依存する表の列構造を決定する、型の属性構造を指定します。

注意:

条件付きコンパイル・ディレクティブを使用して型の属性構造を変更すると、依存オブジェクトが非同期になったり、依存表がアクセス不可になる場合があります。型の属性構造は、「ALTER TYPE 文」を使用してのみ変更することをお薦めします。ALTER TYPE 文は、変更を依存オブジェクトまで伝播します。

スキーマ・レベルの型の仕様部で条件付きコンパイル・ディレクティブが使用されていると、コンパイラはエラー「PLS-00180: プリプロセッサ・ディレクティブは、このコンテキストでサポートされていません」を報告します。

すべての条件付きコンパイラの構造はPL/SQLプリプロセッサによって処理されるため、SQLパーサーは、ストアドPL/SQLユニットまたは無名ブロック内の最初の条件付きコンパイル・ディレクティブの場所に対して次の制限を強制します。

- パッケージ仕様部、パッケージ本体、型本体、スキーマ・レベル・ファンクションおよびスキーマ・レベル・プロシージャには、条件コンパイル・ディレクティブが有効になる前に、ユニット名の識別子の後に少なくとも1つの空白でないPL/SQLトークンがあることが必要です。

ノート:

- PL/SQL のコメント"--"または"/*"は、空白トークンとしてカウントされます。
- トークンが PL/SQL で無効な場合は、PLS-00103 エラーが発行されます。ただし、このルールに違反した状態で条件付きコンパイル・ディレクティブが使用されていると、ORA エラーが生成されます。

例2-61および例2-62では、最初の条件付きコンパイル・ディレクティブが、定義しているユニットの識別子に続く最初のPL/SQLトークンの後にあることを示しています。

- トリガーまたは無名ブロックでは、キーワードDECLAREまたはBEGIN(いずれか先に使用されている方)の前に最初の条件付きコンパイル・ディレクティブを配置することはできません。

SQLパーサーでは、無名ブロックでプレースホルダが使用される場合、そのプレースホルダは条件付きコンパイル・ディレクティブで使用できないという制限もあります。たとえば:

```
BEGIN
  :n := 1; -- valid use of placeholder
  $IF ... $THEN
    :n := 1; -- invalid use of placeholder
$END
```

例2-61 パッケージ仕様の定義で使用する条件付きコンパイル・ディレクティブ

この例は、パッケージ仕様の定義で、AUTHID句の後のキーワードISの前に、最初の条件付きコンパイル・ディレクティブを配置する例を示しています。

```
CREATE OR REPLACE PACKAGE cc_pkg
AUTHID DEFINER
$IF $$XFLAG $THEN ACCESSIBLE BY(p1_pkg) $END
IS
    i NUMBER := 10;
    trace CONSTANT BOOLEAN := TRUE;
END cc_pkg;
```

結果:

```
Package created.
```

例2-62 サブプログラムの仮パラメータ・リストで使用する条件付きコンパイル・ディレクティブ

この例は、PL/SQLプロシージャ定義の仮パラメータ・リストで、左カッコの後に最初の条件付きコンパイル・ディレクティブが配置されることを示しています。

```
CREATE OR REPLACE PROCEDURE my_proc (
  $IF $$xxx $THEN i IN PLS_INTEGER $ELSE i IN INTEGER $END
) IS
BEGIN
      NULL;
END my_proc;
```

結果:

```
Procedure created.
```

3 PL/SQLデータ型

PL/SQLの定数、変数、パラメータ、ファンクションの戻り値はすべて、記憶形式および有効な値と演算を決定するデータ型を持っています。

この章では、スカラー・データ型という、内部コンポーネントのない値を格納するデータ型について説明します。

スカラー・データ型にはサブタイプを持たせることができます。サブタイプとは、別のデータ型のサブセットとなるデータ型のことで、その別のデータ型は、そのサブタイプのベース型となります。サブタイプには、そのベース型で有効な演算と同じ演算が含まれています。データ型とそのサブタイプでデータ型ファミリが構成されます。

PL/SQLには、多くの型やサブタイプがパッケージSTANDARDに事前定義されています。また、PL/SQLを使用して、独自のサブタイプを定義することもできます。

PL/SQLのスカラー・データ型は次のとおりです。

- SQLデータ型
- BOOLEAN
- PLS_INTEGER
- BINARY_INTEGER
- REF CURSOR
- ユーザー定義のサブタイプ

ここでのトピック

- SQLデータ型
- BOOLEANデータ型
- PLS_INTEGERおよびBINARY_INTEGERデータ型
- PLS_INTEGERのSIMPLE_INTEGERサブタイプ
- ユーザー定義のPL/SQLサブタイプ

関連項目:

- コンポジット・データ型の詳細は、「PL/SQLのコレクションおよびレコード」を参照してください
- REF CURSORの詳細は、「カーソル変数」を参照してください
- スキーマ・レベルのユーザー定義データ型の作成の詳細は、「CREATE TYPE文」を参照してください
- データ型ファミリごとにグループ化された、事前定義のPL/SQLデータ型とサブタイプは、「PL/SQLの事前定義のデータ型」を参照してください

3.1 SQLデータ型

PL/SQLデータ型にはSQLデータ型が含まれます。

SQLデータ型の詳細は、『Oracle Database SQL言語リファレンス』を参照してください(記載されているデータ型およびサブタイプ、データ型の比較のルール、データ変換、リテラル、および書式モデルに関する情報はすべて、次の項目を除き、SQLとPL/SQLの両方に適用されます)。

- 最大サイズの相違
- BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQL定数
- BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQLサブタイプ

SQLとは異なり、PL/SQLでは変数を宣言できます。変数には次のトピックが適用されます。

- CHARおよびVARCHAR2変数
- LONGおよびLONG RAW変数
- ROWIDおよびUROWID変数

3.1.1 最大サイズの相違

表3-1にリストしたSQLのデータ型は、PL/SQLとSQLで最大サイズが異なります。

表3-1 PL/SQLとSQLで最大サイズが異なるデータ型

データ型	PL/SQLでの最大サイズ	SQLでの最大サイズ
CHAR 脚注 1	32,767 バイト	2,000 バイト
NCHAR 脚注 1	32,767 バイト	2,000 バイト
RAW 脚注 1	32,767 バイト	2,000 バイト脚注 2
VARCHAR2 脚注 1	32,767 バイト	4,000 バイト脚注 2
NVARCHAR2 脚注 1	32,767 バイト	4,000 バイト脚注 2
LONG 脚注 3	32,760 バイト	2 ギガバイト(GB) - 1
LONG RAW 脚注 3	32,760 バイト	2GB
BLOB	128 テラバイト(TB)	(4GB - 1) * database_block_size
CLOB	128TB	(4GB - 1) * database_block_size
NCLOB	128TB	(4GB - 1) * database_block_size

脚注1

このデータ型の値の最大サイズをPL/SQLで指定する場合は、定数または変数ではなく、1から32,767の間の値を持つ整数リテラルを使用します。

脚注2

このサイズの違いをなくすには、『Oracle Database SQL言語リファレンス』の手順に従ってください。

脚注3

既存のアプリケーションとの下位互換性を保つためにのみサポートされています。

3.1.2 BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQL定数

SQLデータ型のBINARY_FLOATおよびBINARY_DOUBLEは、それぞれ単精度および倍精度のIEEE 754形式の浮動小数点数を表します。

BINARY_FLOATおよびBINARY_DOUBLEの計算では、例外は発生しないため、これらの計算によって生成された値がオーバーフローやアンダーフローなどの状態になっていないかどうか、事前定義の定数と比較して確認する必要があります(例は、『Oracle Database SQL言語リファレンス』を参照してください)。PL/SQLには、この型の定数がSQLよりも多数あります。

表3-2に、BINARY_FLOATおよびBINARY_DOUBLEとして事前定義されているPL/SQL定数を示します。SQLでも定義されているものには注を付けてあります。

表3-2 事前定義のPL/SQLのBINARY_FLOATおよびBINARY_DOUBLEの定数

定数	説明
BINARY_FLOAT_NAN (*)	条件 IS NAN(非数値)が TRUE になっている BINARY_FLOAT 値。
BINARY_FLOAT_INFINITY (*)	単精度の正の無限大
BINARY_FLOAT_MAX_NORMAL	最大正規 BINARY_FLOAT 値。
BINARY_FLOAT_MIN_NORMAL	最小正規 BINARY_FLOAT 値。
BINARY_FLOAT_MAX_SUBNORMAL	最大非正規 BINARY_FLOAT 値。
BINARY_FLOAT_MIN_SUBNORMAL	最小非正規 BINARY_FLOAT 値。
BINARY_DOUBLE_NAN (*)	条件 IS NAN(非数値)が TRUE になっている BINARY_DOUBLE 値。
BINARY_DOUBLE_INFINITY (*)	倍精度の正の無限大
BINARY_DOUBLE_MAX_NORMAL	最大正規 BINARY_DOUBLE 値。

定数	説明
BINARY_DOUBLE_MIN_NORMAL	最小正規 BINARY_DOUBLE 値。
BINARY_DOUBLE_MAX_SUBNORMAL	最大非正規 BINARY_DOUBLE 値。
BINARY_DOUBLE_MIN_SUBNORMAL	最小非正規 BINARY_DOUBLE 値。

(*) この定数は、SQLでも事前定義されています。

3.1.3 BINARY_FLOATおよびBINARY_DOUBLEの追加のPL/SQLサブタイプ

PL/SQLでは、次のサブタイプが事前定義されています。

- SIMPLE_FLOAT(SQLデータ型BINARY_FLOATのサブタイプ)
- SIMPLE_DOUBLE(SQLデータ型BINARY_DOUBLEのサブタイプ)

各サブタイプはそのベース型と同じ範囲を取り、NOT NULL制約(「NOT NULL制約」を参照)を含んでいます。

値NULLを取らないとわかっている変数は、BINARY_FLOATまたはBINARY_DOUBLEではなく、SIMPLE_FLOATまたはSIMPLE_DOUBLEとして宣言します。これらのサブタイプを使用すると、NULLかどうかのチェックのためのオーバーヘッドが発生せず、そのベース型を使用するよりパフォーマンスが大幅に向上します。PLSQL_CODE_TYPE='INTERPRETED'よりもPLSQL_CODE_TYPE='NATIVE'を使用する方が、パフォーマンスの向上幅が大きくなります(詳細は、「ハードウェア算術を使用するデータ型の使用」を参照してください)。

3.1.4 CHARおよびVARCHAR2変数

ここでのトピック

- 長すぎる値の代入または挿入
- マルチバイト文字の変数の宣言
- CHARとVARCHAR2のデータ型の違い

3.1.4.1 長すぎる値の代入または挿入

文字変数に変数の最大サイズより長い値を代入すると、エラーが発生します。たとえば:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := 'abc  ';
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
```

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4
```

同様に、列に挿入した文字変数の値が、定義されている列幅より長い場合も、エラーが発生します。たとえば:

```
DROP TABLE t;
CREATE TABLE t (c CHAR(3 CHAR));

DECLARE
  s VARCHAR2(5 CHAR) := 'abc  ';
BEGIN
  INSERT INTO t(c) VALUES(s);
END;
/
```

結果:

```
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."T"."C" (actual: 5, maximum: 3)
ORA-06512: at line 4
```

文字値を変数に代入したり、列に挿入する前に、その文字値から後続する空白を削除するには、RTRIMファンクション(『Oracle Database SQL言語リファレンス』を参照)を使用します。たとえば:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := RTRIM('abc  ');
  INSERT INTO t(c) VALUES(RTRIM('abc  '));
END;
/
```

結果:

```
PL/SQL procedure successfully completed.
```

3.1.4.2 マルチバイト文字の変数の宣言

CHARまたはVARCHAR2変数の最大サイズは、最大サイズを文字数で指定するかバイト数で指定するかに関係なく、32,767バイトです。変数の最大文字数は、文字セットのタイプによって、場合によっては文字自体によって異なります。

文字セットのタイプ	最大文字数
シングルバイト文字セット	32,767
n バイトの固定幅マルチバイト文字セット(AL16UTF16 など)	FLOOR(32,767/n)
文字幅が 1 と n バイトの間である、n バイトの可変幅マルチバイト文字セット(JA16SJIS や	文字自体によって異なり、32,767(1 バイト文字のみを含む文字列の場合)から FLOOR(32,767/n)(n バイト文字のみを含む文字列の場合)を指

文字セットのタイプ	最大文字数
AL32UTF8 など)	定できます。

任意のマルチバイト文字セットでn個の文字を常に保持できるように、CHARまたはVARCHAR2変数を宣言する場合は、CHAR(n CHAR)またはVARCHAR2(n CHAR)を使用して、文字数の長さを宣言します(nはFLOOR(32767/4) = 8191を超えることはできません)。

関連項目:

Oracle Databaseの文字セットのサポートの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

3.1.4.3 CHARとVARCHAR2のデータ型の違い

CHARとVARCHAR2のデータ型は、次の点で異なっています。

- 事前定義のサブタイプ
- 空白埋めの動作方法
- 値の比較

3.1.4.3.1 事前定義のサブタイプ

CHARデータ型には、PL/SQLとSQLの両方にCHARACTERというサブタイプが1つ事前定義されています。

VARCHAR2データ型には、PL/SQLとSQLの両方にVARCHARというサブタイプが1つ事前定義されており、PL/SQLには、STRINGというサブタイプがもう1つ事前定義されています。

各サブタイプの値の範囲はそのベース型と同じです。

ノート:

PL/SQL の今後のリリースでの VARCHAR は、新しい SQL 標準に従うために別のデータ型になり、VARCHAR2 と同義ではなくなる可能性があります。

3.1.4.3.2 空白埋めの動作方法

これは、空白埋めを使用する場合のCHARとVARCHAR2での相違点および考慮事項を説明しています。

次の状況について考えてみます。

- 変数に代入した値がこの変数の最大サイズより短い。
- 列に挿入した値が、定義されている列幅より短い。
- 列から取り出して変数に入れた値が、この変数の最大サイズより短い。

受信者のデータ型がCHARの場合、PL/SQLは最大サイズになるまで値を空白で埋めます。元の値の後続する空白に関する情

報は失われます。

受信者のデータ型がVARCHAR2の場合、PL/SQLは値の空白埋めも、後続する空白の削除もしません。文字値はそのまま代入され、情報は失われません。

例3-1 CHARおよびVARCHAR2の空白埋めの相違点

この例では、CHAR変数とVARCHAR2変数は、どちらも最大サイズが10文字です。それぞれの変数に、1つの空白が後続している5文字の値を代入します。CHAR変数に代入した値は空白が埋められて10文字になり、結果の値に含まれる後続の空白6つのうち、1つは元の値に含まれていたものだとは見分けられません。VARCHAR2変数に代入された値は変化しないため、後続の空白が1つあることを確認できます。

```
DECLARE
  first_name  CHAR(10 CHAR);
  last_name   VARCHAR2(10 CHAR);
BEGIN
  first_name := 'John ';
  last_name  := 'Chen ';

  DBMS_OUTPUT.PUT_LINE('*' || first_name || '*');
  DBMS_OUTPUT.PUT_LINE('*' || last_name || '*');
END;
/
```

結果:

```
*John      *
*Chen *
```

3.1.4.3.3 値の比較

SQLの文字値の比較ルールが、PL/SQLの文字変数に適用されます。

比較する値の1つまたは両方のデータ型がVARCHAR2またはNVARCHAR2の場合は、非空白埋め比較セマンティクスが適用されますが、それ以外の場合は、空白埋めセマンティクスが適用されます。詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

3.1.5 LONGおよびLONG RAW変数

ノート:

Oracle は、既存アプリケーションとの下位互換性のためにのみ、LONG および LONG RAW データ型をサポートしています。新しいアプリケーションでは次のようにしてください。

- LONG のかわりに、VARCHAR2(32760)、BLOB、CLOB または NCLOB を使用する。
- LONG RAW のかわりに、BLOB を使用する。

LONG列には、任意のLONG値を挿入できます。LONG RAW列には、任意のLONG RAW値を挿入できます。LONGまたはLONG RAW列

からは、32760バイトを超える長さの値を取り出して、LONGまたはLONG RAW変数に入れることはできません。

LONG列には、任意のCHARまたはVARCHAR2値を挿入できます。LONG列からは、32,767バイトを超える長さの値を取り出してCHAR変数またはVARCHAR2変数に入れることはできません。

LONG RAW列には、任意のRAW値を挿入できます。LONG RAW列からは、32767バイトを超える長さの値を取り出してRAW変数に入れることはできません。

関連項目:

トリガー内のLONGデータ型およびLONG RAWデータ型の制約については、「トリガーのLONGおよびLONG RAWデータ型の制約」を参照してください

3.1.6 ROWIDおよびUROWID変数

ROWIDを取り出してROWID変数に入れる場合は、バイナリ値を文字値に変換するROWIDTOCHARファンクションを使用します。このファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

ROWID変数の値をROWIDに変換するには、CHARTOROWIDファンクション(『Oracle Database SQL言語リファレンス』を参照)を使用します。値が有効なROWIDを表していない場合は、PL/SQLにより事前定義の例外SYS_INVALID_ROWIDが呼び出されます。

ROWIDを取り出してUROWID変数に入れる場合、またはUROWID変数の値をROWIDに変換する場合は、代入文を使用します。変換は暗黙的に実行されます。

ノート:

- UROWID は論理 ROWID と物理 ROWID の両方と互換性があるため、ROWID より汎用性の高いデータ型です。
- ハイブリッド列圧縮(HCC)で圧縮された表の行を更新すると、行の ROWID が変更されます。特定の Oracle ストレージ・システムの機能である HCC の詳細は、『Oracle Database 概要』を参照してください。

関連項目:

DBMS_ROWIDパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。このパッケージのサブプログラムを使用すると、ROWID値に関する情報を作成したり戻すことができます(ただし、UROWID値に関してはできません)。

3.2 BOOLEANデータ型

PL/SQLのデータ型BOOLEANには、論理値(ブール値のTRUEとFALSE、およびNULL値)が格納されます。NULLは、不明な値を表します。

BOOLEAN変数を宣言するための構文は、次のとおりです。

```
variable_name BOOLEAN
```

BOOLEAN変数に代入できる値はBOOLEAN式のみです。詳細は、「ブール式」を参照してください。

SQLにはBOOLEANに相当するデータ型がないため、次の操作は実行できません。

- データベース表の列にBOOLEAN値を割り当てる操作
- データベース表の列の値を選択またはフェッチしてBOOLEAN変数に入れる操作
- SQLファンクションでBOOLEAN値を使用する操作

  (ただし、「例3-3」に示すように、SQL問合せはBOOLEANパラメータを持つPL/SQLファンクションを起動できます。)
- SQL文(SQL問合せで起動されるPL/SQLファンクションの引数を除く)またはPL/SQL無名ブロックでのBOOLEAN式の使用

> ノート:
>
> 静的 SQL 問合せで起動される PL/SQL ファンクションの引数は、BOOLEAN リテラルにできません。これを回避するには、「例 3-3」に示すように、リテラルを変数に割り当ててからその変数をファンクションに渡します。

BOOLEAN値は、DBMS_OUTPUT.PUTサブプログラムまたはDBMS_OUTPUT.PUTLINEサブプログラムに渡せません。BOOLEAN値を出力するには、IF文またはCASE文を使用して値を文字値に変換します(これらの文の詳細は、「条件付き選択文」を参照してください)。

例3-2 BOOLEAN値の出力

この例では、プロシージャは、BOOLEANパラメータを受け取り、CASE文を使用して、値がNULLならUnknownを、TRUEならYesを、FALSEならNoを出力します。

関連項目:

例2-34を参照してください。この例では、IF文を使用してprint_booleanプロシージャを作成しています。

```
PROCEDURE print_boolean (b BOOLEAN)

AS
BEGIN
  DBMS_OUTPUT.PUT_LINE (
    CASE
      WHEN b IS NULL THEN 'Unknown'
      WHEN b THEN 'Yes'
```

```
      WHEN NOT b THEN 'No'
    END
  );
END;
BEGIN
  print_boolean(TRUE);
  print_boolean(FALSE);
  print_boolean(NULL);
END;
```

結果:

```
Yes
No
Unknown
```

例3-3 SQL文からのBOOLEANパラメータを持つPL/SQLファンクションの起動

この例では、SQL文により、BOOLEANパラメータを持つPL/SQLファンクションが起動されます。

```
FUNCTION f (x BOOLEAN, y PLS_INTEGER)
  RETURN employees.employee_id%TYPE
  AUTHID CURRENT_USER AS
BEGIN
  IF x THEN
    RETURN y;
  ELSE
    RETURN 2*y;
  END IF;
END;

DECLARE
  name employees.last_name%TYPE;
  b BOOLEAN := TRUE;
BEGIN
  SELECT last_name INTO name
  FROM employees
  WHERE employee_id = f(b, 100);

  DBMS_OUTPUT.PUT_LINE(name);

  b := FALSE;

  SELECT last_name INTO name
  FROM employees
  WHERE employee_id = f(b, 100);

  DBMS_OUTPUT.PUT_LINE(name);
END;
/
```

結果:

```
King
Whalen
```

3.3 PLS_INTEGERおよびBINARY_INTEGERデータ型

PL/SQLのデータ型であるPLS_INTEGERとBINARY_INTEGERは同じです。

わかりやすくするために、このマニュアルでは、PLS_INTEGERとBINARY_INTEGERの両方を表すためにPLS_INTEGERを使用します。

PLS_INTEGERデータ型は、32ビットで表される-2147483648から2147483647の範囲の符号付き整数を格納します。

PLS_INTEGERデータ型には、NUMBERデータ型およびNUMBERサブタイプと比べて、次のようなメリットがあります。

- PLS_INTEGER値の方が、必要な記憶域が少なくなります。
- PLS_INTEGER演算はハードウェア算術計算を使用するため、ライブラリ算術計算を使用するNUMBER演算より処理速度が速くなります。

効率のために、PLS_INTEGERの範囲内でのすべての計算にPLS_INTEGER値を使用してください。

ここでのトピック

- PLS_INTEGERのオーバーフローの回避
- PLS_INTEGERの事前定義のサブタイプ
- PLS_INTEGERのSIMPLE_INTEGERサブタイプ

3.3.1 PLS_INTEGERのオーバーフローの回避

PLS_INTEGER範囲をオーバーフローする2つのPLS_INTEGER値を使用した計算では、オーバーフロー例外が発生します。

PLS_INTEGERの範囲外の計算には、NUMBERデータ型の事前定義のサブタイプであるINTEGERを使用します。

例3-4 オーバーフロー例外が発生するPLS_INTEGERの計算

この例では、2つのPLS_INTEGER値を使用する計算がPLS_INTEGERの範囲をオーバーフローした場合は、結果をNUMBERデータ型に代入しても、オーバーフロー例外が発生することを示します。

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 PLS_INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

例3-5 例3-4のオーバーフローの防止

この例では、PLS_INTEGER範囲外の計算での事前定義のサブタイプINTEGERの正しい使用を示しています。

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

結果:

```
PL/SQL procedure successfully completed.
```

3.3.2 PLS_INTEGERの事前定義のサブタイプ

このサマリーは、PLS_INTEGERデータ型の事前定義済のサブタイプ、およびそれらが格納するデータを示します。

表3-3 PLS_INTEGERデータ型の事前定義のサブタイプ

データ型	データの説明
NATURAL	負でない PLS_INTEGER 値。
NATURALN	NOT NULL 制約が指定された負ではない PLS_INTEGER 値
POSITIVE	正の PLS_INTEGER 値。
POSITIVEN	NOT NULL 制約が指定された正の PLS_INTEGER 値
SIGNTYPE	-1、0 または 1 の PLS_INTEGER 値(3 値論理のプログラミングに有効です)。
SIMPLE_INTEGER	NOT NULL 制約が指定された PLS_INTEGER 値。

PLS_INTEGERおよびそのサブタイプは、次のデータ型に暗黙的に変換できます。

- CHAR
- VARCHAR2
- NUMBER
- LONG

LONGを除く前述のすべてのデータ型、およびすべてのPLS_INTEGERサブタイプは、PLS_INTEGERに暗黙的に変換できます。

PLS_INTEGER値は、サブタイプの制約に違反していない場合のみ、PLS_INTEGERサブタイプに暗黙的に変換できます。

関連項目:

- NOT NULL制約の詳細は、「NOT NULL制約」を参照してください
- SIMPLE_INTEGERの詳細は、「PLS_INTEGERのSIMPLE_INTEGERサブタイプ」を参照してください

例3-6 SIMPLE_INTEGERサブタイプの制約違反

この例は、PLS_INTEGER値のNULLをSIMPLE_INTEGERサブタイプにキャストすると例外が発生することを示しています。

```
DECLARE
  a SIMPLE_INTEGER := 1;
  b PLS_INTEGER := NULL;
BEGIN
  a := b;
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```

3.3.3 PLS_INTEGERのSIMPLE_INTEGERサブタイプ

SIMPLE_INTEGERは、PLS_INTEGERデータ型の事前定義済のサブタイプです。

SIMPLE_INTEGERはPLS_INTEGERと同じ範囲を持ち、NOT NULL制約があります。PLS_INTEGERとの大きな違いはオーバーフローの方法です。

変数の値がNULLになることがなく、オーバーフロー・チェックも不要であるとわかっている場合は、PLS_INTEGERではなくSIMPLE_INTEGERとして値を宣言します。SIMPLE_INTEGERの場合は、NULLかどうかのチェックおよびオーバーフロー・チェックのためのオーバーヘッドが発生しないため、PLS_INTEGERよりもパフォーマンスが大幅に向上します。

ここでのトピック

- SIMPLE_INTEGERのオーバーフローの方法
- SIMPLE_INTEGERと他のオペランドの両方を使用する式
- SIMPLE_INTEGERの範囲内の整数リテラル

関連項目:

NOT NULL制約

3.3.3.1 SIMPLE_INTEGERのオーバーフローの方法

式に含まれるすべてのオペランドのデータ型がSIMPLE_INTEGERの場合にのみ、PL/SQLで2の補数算術が使用され、オーバーフローが無視されます。

オーバーフローが無視されるため、値を正から負、または負から正にラップできます。たとえば:

$2^{30} + 2^{30}$ = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}

$-2^{31} + -2^{31}$ = 0x80000000 + 0x80000000 = 0x00000000 = 0

たとえば、次のブロックはエラーなしで実行されます。

```
DECLARE
  n SIMPLE_INTEGER := 2147483645;
BEGIN
  FOR j IN 1..4 LOOP
    n := n + 1;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
  FOR j IN 1..4 LOOP
   n := n - 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
END;
/
```

結果:

```
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645
PL/SQL procedure successfully completed.
```

3.3.3.2 SIMPLE_INTEGERと他のオペランドの両方を使用する式

SIMPLE_INTEGERと他のオペランドの両方が式に含まれている場合は、PL/SQLによりSIMPLE_INTEGER値がPLS_INTEGER NOT NULLに暗黙的に変換されます。

一部の最適化が抑制されてパフォーマンスに悪影響を与えるような状態でSIMPLE_INTEGER値と他の値が混在している場合は、PL/SQLコンパイラから警告が発行されます。

3.3.3.3 SIMPLE_INTEGERの範囲内の整数リテラル

SIMPLE_INTEGERの範囲内の整数リテラルのデータ型はSIMPLE_INTEGERになります。

ただし、算術式内のすべてのオペランドが整数リテラルである場合、PL/SQLでは、下位互換性を確保するために、PLS_INTEGERにキャストされた場合と同様にその整数リテラルが処理されます。

3.4 ユーザー定義のPL/SQLサブタイプ

PL/SQLではユーザー独自のサブタイプを定義できます。

ベース型は、CHAR、DATEまたはRECORDなどのスカラーまたはユーザー定義のPL/SQLデータ型指定子です(事前に定義したユーザー定義のサブタイプが含まれます)。

ノート:

このトピック内の情報は、ユーザー定義のサブタイプおよび「PL/SQL の事前定義のデータ型」にリストされている事前定義のサブタイプの両方に適用されます。

サブタイプの役割は次のとおりです。

- ANSI/ISOデータ型との互換性の提供
- その型のデータ項目の使用意図の提示
- 範囲外の値の検出

ここでのトピック

- 無制約のサブタイプ
- 制約付きサブタイプ
- 同じデータ型ファミリ内のベース型を使用したサブタイプ

3.4.1 無制約のサブタイプ

無制約のサブタイプには、そのベース型と同じ値セットが含まれているため、ベース型の別名にすぎません。

したがって、同じベース型の無制約のサブタイプは互換性があり、そのベース型との互換性もあります。データ型の変換は発生しません。

無制約のサブタイプを定義するには、次の構文を使用します。

```
SUBTYPE subtype_name IS base_type
```

subtype_nameおよびbase_typeの詳細は、subtypeを参照してください。

無制約のサブタイプの例を示します。これは、ANSIとの互換性のためにPL/SQLで事前定義されているものです。

```
SUBTYPE "DOUBLE PRECISION" IS FLOAT
```

例3-7 使用意図を示すユーザー定義の無制約のサブタイプ

この例では、制約なしのサブタイプであるBalanceおよびCounter型それぞれのデータ項目の使用意図を示しています。

```
DECLARE
  SUBTYPE Balance IS NUMBER;
  checking_account        Balance(6,2);
  savings_account         Balance(8,2);
  certificate_of_deposit  Balance(8,2);
  max_insured  CONSTANT   Balance(8,2) := 250000.00;
  SUBTYPE Counter IS NATURAL;
  accounts     Counter := 1;
  deposits     Counter := 0;
  withdrawals  Counter := 0;
  overdrafts   Counter := 0;
  PROCEDURE deposit (
```

```
    account  IN OUT Balance,
    amount   IN     Balance
  ) IS
  BEGIN
    account  := account + amount;
    deposits := deposits + 1;
  END;

BEGIN
  NULL;
END;
/
```

3.4.2 制約付きサブタイプ

制約付きサブタイプには、そのベース型の値のサブセットのみが含まれています。

サイズ、精度と位取り、または値の範囲を指定できるベース型の場合は、そのサブタイプにもこれらを指定できます。次に、サブタイプを定義する構文を示します。

```
SUBTYPE subtype_name IS base_type
  { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

この構文を使用しない場合、サブタイプに指定されるのはNOT NULL制約のみです。

```
SUBTYPE subtype_name IS base_type [ NOT NULL ]
```

ノート:

値の範囲を指定できるベース型は、PLS_INTEGER とそのサブタイプ(事前定義とユーザー定義の両方)のみです。

制約付きサブタイプは暗黙的にベース型に変換されますが、ベース型を制約付きサブタイプに暗黙的に変換できるのは、値がサブタイプの制約に違反していない場合のみです。

制約付きサブタイプは、変換元の値が変換先のサブタイプの制約に違反していない場合のみ、同じベース型を使用する別の制約付きサブタイプに暗黙的に変換できます。

関連項目:

- 「subtype_definition ::=」の構文図
- 「subtype」のセマンティクスの説明
- 「例3-6」、「SIMPLE_INTEGERサブタイプの制約違反」
- 「制約付きサブタイプの仮パラメータ」
- NOT NULL制約

例3-8 範囲外の値を検出するユーザー定義の制約付きサブタイプ

この例では、制約付きサブタイプBalanceを使用して範囲外の値を検出します。

```
DECLARE
  SUBTYPE Balance IS NUMBER(8,2);

  checking_account  Balance;
  savings_account   Balance;

BEGIN
  checking_account := 2000.00;
  savings_account  := 1000000.00;
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

例3-9 同じベース型を使用する制約付きサブタイプ間の暗黙的な変換

この例では、3つの制約付きサブタイプに同じベース型が使用されています。最初の2つのサブタイプは、3番目のサブタイプに暗黙的に変換できますが、相互には変換できません。

```
DECLARE
  SUBTYPE Digit        IS PLS_INTEGER RANGE 0..9;
  SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
  SUBTYPE Under_100    IS PLS_INTEGER RANGE 0..99;

  d   Digit        :=  4;
  dd  Double_digit := 35;
  u   Under_100;
BEGIN
  u := d;   -- Succeeds; Under_100 range includes Digit range
  u := dd;  -- Succeeds; Under_100 range includes Double_digit range
  dd := d;  -- Raises error; Double_digit range does not include Digit range
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 12
```

3.4.3 同じデータ型ファミリ内のベース型を使用したサブタイプ

同じデータ型ファミリ内の異なるベース型が2つのサブタイプに使用されている場合は、変換元の値が変換先のサブタイプの制約に違反していない場合のみ、一方から他方へ暗黙的に変換できます。

データ型ファミリでグループ化された事前定義のPL/SQLのデータ型およびサブタイプは、「PL/SQLの事前定義のデータ型」を参照してください。

例3-10 同じファミリ内のベース型を使用するサブタイプ間の暗黙的な変換

この例では、サブタイプのWordとTextに、同じデータ型ファミリ内の異なるベース型が使用されています。最初の代入文では、Wordの値が暗黙的にTextに変換されます。2番目の代入文では、Textの値が暗黙的にWordに変換されます。3番目の代入文では、Textの値が長すぎるため、値を暗黙的にWordに変換することができません。

```
DECLARE
  SUBTYPE Word IS CHAR(6);
  SUBTYPE Text IS VARCHAR2(15);

  verb       Word := 'run';
  sentence1  Text;
  sentence2  Text := 'Hurry!';
  sentence3  Text := 'See Tom run.';

BEGIN
  sentence1 := verb;  -- 3-character value, 15-character limit
  verb := sentence2;  -- 6-character value, 6-character limit
  verb := sentence3;  -- 12-character value, 6-character limit
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 13
```

4 PL/SQLの制御文

PL/SQLには、条件付き選択文、繰り返し文および順次制御文の3つのカテゴリの制御文があります。

PL/SQLの制御文のカテゴリは次のとおりです。

- 条件付き選択文: データ値に応じて、異なる文を実行します。

  条件選択文は、IFおよびCASEです。

- ループ文: 一連の異なるデータ値を使用して、同じ文を実行します。

  ループ文には、基本LOOP、FOR LOOPおよびWHILE LOOPがあります。

  EXIT文は、制御をループの終わりに転送します。 CONTINUE文は、現在のループの反復を終了し、制御を次の反復に転送します。EXITおよびCONTINUEのいずれにもオプションのWHEN句があり、これによって条件を指定できます。

- 順次制御文: PL/SQLのプログラミングでは重要ではありません。

  順次制御文には、指定された文に移動するGOTOおよび何も実行しないNULLがあります。

ここでのトピック

- 条件付き選択文
- LOOP文
- 順次制御文

4.1 条件付き選択文

条件付き選択文のIFおよびCASEは、データ値の違いに応じて異なる文を実行します。

IF文は、条件に応じて、一連の1つ以上の文を実行またはスキップします。IF文には次の形式があります。

- IF THEN
- IF THEN ELSE
- IF THEN ELSIF

CASE文を使用すると、一連の条件を基に、対応する文を選択して実行できます。CASE文には次の形式があります。

- 単純: 単一の式を評価して、可能性のある複数の値と比較します。
- 検索: 複数の条件を評価して、最初にTRUEである条件を選択します。

CASE文は、オプションごとにアクションが異なる場合に適しています。

ここでのトピック

- IF THEN文
- IF THEN ELSE文
- IF THEN ELSIF文
- 単純なCASE文

- 検索CASE文

4.1.1 IF THEN文

IF THEN文は、条件に応じて、一連の1つ以上の文を実行またはスキップします。

IF THEN文の構造は、次のとおりです。

```
IF condition THEN
  statements
END IF;
```

conditionがTRUEの場合、statementsが実行されますが、それ以外の場合、IF文は何も実行しません。

構文の詳細は、「IF文」を参照してください。

ヒント:

次のような IF 文の使用は避けてください。

```
IF new_balance < minimum_balance THEN
  overdrawn := TRUE;
ELSE
  overdrawn := FALSE;
END IF;
```

かわりに、BOOLEAN 式の値を BOOLEAN 変数に直接代入してください。

```
overdrawn := new_balance < minimum_balance;
```

BOOLEAN 変数は、TRUE、FALSE、NULL のいずれかです。次のようには記述しないでください。

```
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

かわりに、次のように記述してください。

```
IF overdrawn THEN
  RAISE insufficient_funds;
END IF;
```

例4-1 IF THEN文

この例では、salesの値がquota+200より大きい場合にのみ、THENとEND IFの間にある文が実行されます。

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
```

```
    bonus    NUMBER := 0;
    updated  VARCHAR2(3) := 'No';
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;

      UPDATE employees
      SET salary = salary + bonus
      WHERE employee_id = emp_id;

      updated := 'Yes';
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Table updated?  ' || updated || ', ' ||
      'bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
/
```

結果:

```
Table updated?  No, bonus = 0.
Table updated?  Yes, bonus = 125.
```

4.1.2 IF THEN ELSE文

IF THEN ELSE文の構造は、次のとおりです。

```
IF condition THEN
  statements
ELSE
  else_statements
END IF;
```

conditionの値がTRUEの場合、statementsが実行されますが、それ以外の場合は、else_statementsが実行されます。

例4-3に示すように、IF文はネストできます。

構文の詳細は、「IF文」を参照してください。

例4-2 IF THEN ELSE文

この例では、salesの値がquota+200より大きい場合にのみ、THENとELSEの間にある文が実行されます。それ以外の場合は、ELSEとEND IFの間にある文が実行されます。

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
```

```
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      bonus := 50;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
/
```

結果:

```
bonus = 50
bonus = 125
```

例4-3 ネストしたIF THEN ELSE文

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      IF sales > quota THEN
        bonus := 50;
      ELSE
        bonus := 0;
      END IF;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
  p(9500, 10000, 122);
```

```
END;
/
```

結果:

```
bonus = 50
bonus = 125
bonus = 0
```

4.1.3 IF THEN ELSIF文

IF THEN ELSIF文の構造は、次のとおりです。

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]...
[ ELSE
    else_statements
]
END IF;
```

IF THEN ELSIF文は、conditionがTRUEである最初のstatementsを実行します。残りの条件は評価されません。TRUEのconditionがない場合は、else_statements(存在する場合)が実行されますが、else_statementsも存在しない場合、IF THEN ELSIF文は何も実行しません。

単一のIF THEN ELSIF文は、論理的に等価なネストしたIF THEN ELSE文より理解しやすい文です。

```
-- IF THEN ELSIF statement
IF condition_1 THEN statements_1;
  ELSIF condition_2 THEN statements_2;
  ELSIF condition_3 THEN statement_3;
END IF;
-- Logically equivalent nested IF THEN ELSE statements
IF condition_1 THEN
  statements_1;
ELSE
  IF condition_2 THEN
    statements_2;
  ELSE
    IF condition_3 THEN
      statements_3;
    END IF;
  END IF;
END IF;
```

構文の詳細は、「IF文」を参照してください。

例4-4 IF THEN ELSIF文

この例では、salesの値が50000より大きい場合、1番目と2番目の条件のいずれもTRUEになります。ただし、1番目の条件

がTRUEであるため、bonusには1500という値が代入され、2番目の条件はテストされません。bonusに値1500が代入された後、制御はDBMS_OUTPUT.PUT_LINEの起動に移ります。

```
DECLARE
  PROCEDURE p (sales NUMBER)
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > 50000 THEN
      bonus := 1500;
    ELSIF sales > 35000 THEN
      bonus := 500;
    ELSE
      bonus := 100;
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Sales = ' || sales || ', bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(55000);
  p(40000);
  p(30000);
END;
/
```

結果:

```
Sales = 55000, bonus = 1500.
Sales = 40000, bonus = 500.
Sales = 30000, bonus = 100.
```

例4-5 単純なCASE文をシミュレートするIF THEN ELSIF文

この例では、IF THEN ELSIF文を多数のELSIF句とともに使用し、単一の値を多くの使用可能な値と比較しています。この目的では、単純なCASE文の方がより明確になります(例4-6を参照)。

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  IF grade = 'A' THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
  ELSIF grade = 'B' THEN
    DBMS_OUTPUT.PUT_LINE('Very Good');
  ELSIF grade = 'C' THEN
    DBMS_OUTPUT.PUT_LINE('Good');
  ELSIF grade = 'D' THEN
    DBMS_OUTPUT. PUT_LINE('Fair');
  ELSIF grade = 'F' THEN
    DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No such grade');
  END IF;
```

```
END;
/
```

結果:

```
Very Good
```

4.1.4 単純なCASE文

単純なCASE文の構造は、次のとおりです。

```
CASE selector
WHEN selector_value_1 THEN statements_1
WHEN selector_value_2 THEN statements_2
...
WHEN selector_value_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

selectorは式です(通常は1つの変数)。各selector_valueはリテラルまたは式のいずれかです。(構文の詳細は、「CASE文」を参照してください。)

単純なCASE文は、selector_valueがselectorと等しい最初のstatementsを実行します。残りの条件は評価されません。いずれのselector_valueもselectorと等しくない場合、CASE文はelse_statements(存在する場合)を実行しますが、else_statementsが存在しない場合は、事前定義の例外CASE_NOT_FOUNDを呼び出します。

例4-6では、単純なCASE文を使用して、単一の値を多くの使用可能な値と比較しています。例4-6のCASE文は、例4-5のIF THEN ELSIF文と論理的に等価です。

> ノート:
>
> 単純な CASE 式と同様に、単純な CASE 文の選択子の値が NULL の場合、その選択子が WHEN NULL によって一致することはありません(例 2-51 を参照)。かわりに、検索 CASE 文を WHEN condition IS NULL とともに使用します(例 2-53 を参照)。

例4-6 単純なCASE文

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
```

```
END;
/
```

結果:

```
Very Good
```

4.1.5 検索CASE文

検索CASE文の構造は、次のとおりです。

```
CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

検索CASE文は、conditionがTRUEである最初のstatementsを実行します。残りの条件は評価されません。TRUEのconditionがない場合、CASE文はelse_statements(存在する場合)を実行しますが、else_statementsも存在しない場合は、事前定義の例外CASE_NOT_FOUNDを呼び出します。(構文の詳細は、「CASE文」を参照してください。)

例4-7の検索CASE文は、例4-6の単純なCASE文と論理的に等価です。

例4-7と例4-6のいずれの場合も、ELSE句をEXCEPTION部に置き換えることができます。例4-8は、例4-7と論理的に等価です。

例4-7 検索CASE文

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

結果:

```
Very Good
```

例4-8 CASE文でのELSE句にかわるEXCEPTION

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
END;
/
```

結果:

```
Very Good
```

4.2 LOOP文

ループ文は、一連の異なる値を使用して、同じ文を実行します。ループ文には次のものがあります。

- 基本LOOP
- FOR LOOP
- カーソルFOR LOOP
- WHILE LOOP

ループを終了する文には次のものがあります。

- EXIT
- EXIT WHEN

現行のループの反復を終了する文には次のものがあります。

- CONTINUE
- CONTINUE WHEN

EXIT、EXIT WHEN、CONTINUEおよびCONTINUE WHENは、ループ内の任意の場所に置くことができますが、ループの外に置くことはできません。ループの外の文に制御を移してループまたは現行のループの反復を終了することができるGOTO文のかわりに、これらの文を使用することをお薦めします。(例外が呼び出された場合もループは終了します。例外の詳細は、「例外処理の概要」を参照してください。)

LOOP文にはラベルを付けることができ、LOOP文をネストすることもできます。わかりやすさを向上させるために、ネステッド・ループにはラベルを付けることをお薦めします。END LOOP文のラベルと同じループ文の先頭のラベルが一致していることを確認する必要があります(コンパイラはチェックしません)。

ここでのトピック

- 基本LOOP文
- EXIT文
- EXIT WHEN文
- CONTINUE文
- CONTINUE WHEN文
- FOR LOOP文
- WHILE LOOP文

カーソルFOR LOOPの詳細は、「カーソルFOR LOOP文による問合せ結果セットの処理」を参照してください。

4.2.1 基本LOOP文

基本LOOP文の構造は、次のとおりです。

```
[ label ] LOOP
  statements
END LOOP [ label ];
```

ループが反復されるたびにstatementsが実行され、制御がループの先頭に戻ります。無限ループが発生しないように、文または例外の呼び出しによってループを終了する必要があります。

関連項目:

「基本LOOP文」

4.2.2 EXIT文

EXIT文は、現行のループの反復を無条件で終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの最後に制御を移します。

例4-9では、基本LOOP文内のEXIT文が無条件でカレント・ループの最後に制御を移します。

関連項目:

「EXIT文」

例4-9 EXIT文が含まれている基本LOOP文

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
```

```
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
/
```

結果:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

4.2.3 EXIT WHEN文

EXIT WHEN文は、WHEN句内の条件がTRUEの場合、現行のループの反復を終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの最後に制御を移します。

制御がEXIT WHEN文に達するたびに、WHEN句内の条件が評価されます。条件がTRUEでない場合、EXIT WHEN文は何も実行しません。例4-10に示すように、無限ループが発生しないように、ループ内の文で条件がTRUEになるようにする必要があります。

例4-10では、xが3より大きい場合、基本LOOP文内のEXIT WHEN文がカレント・ループの最後に制御を移します。例4-10は例4-9と論理的に等価です。

関連項目:

「EXIT文」

例4-11では、基本LOOP文が別の基本LOOP文にネストされていて、両方にラベルが付けられています。内部ループにはEXIT WHEN文が2つあり、1つは内部ループを終了し、もう1つは外部ループを終了します。

内部ループ内のEXIT WHEN文から外部ループに制御を移すことができるのは、外部ループにラベルが付けられている場合のみです。

例4-12では、外部ループにはラベルが付いていないため、内部ループは制御を外部ループに移すことができません。

例4-10 EXIT WHEN文が含まれている基本LOOP文

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
```

```
  DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
/
```

結果:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

例4-11 EXIT WHEN文が含まれているネストされたラベル付き基本LOOP文

```
DECLARE
  s  PLS_INTEGER := 0;
  i  PLS_INTEGER := 0;
  j  PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
    <<inner_loop>>
    LOOP
      j := j + 1;
      s := s + i * j; -- Sum several products
      EXIT inner_loop WHEN (j > 5);
      EXIT outer_loop WHEN ((i * j) > 15);
    END LOOP inner_loop;
  END LOOP outer_loop;
  DBMS_OUTPUT.PUT_LINE
    ('The sum of products equals: ' || TO_CHAR(s));
END;
/
```

結果:

```
The sum of products equals: 166
```

例4-12 EXIT WHEN文が含まれているネストされたラベルなしの基本LOOP文

```
DECLARE
  i PLS_INTEGER := 0;
  j PLS_INTEGER := 0;

BEGIN
  LOOP
    i := i + 1;
    DBMS_OUTPUT.PUT_LINE ('i = ' || i);

    LOOP
      j := j + 1;
      DBMS_OUTPUT.PUT_LINE ('j = ' || j);
      EXIT WHEN (j > 3);
```

```
    END LOOP;

    DBMS_OUTPUT.PUT_LINE ('Exited inner loop');

    EXIT WHEN (i > 2);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Exited outer loop');
END;
/
```

結果:

```
i = 1
j = 1
j = 2
j = 3
j = 4
Exited inner loop
i = 2
j = 5
Exited inner loop
i = 3
j = 6
Exited inner loop
Exited outer loop

PL/SQL procedure successfully completed.
```

4.2.4 CONTINUE文

CONTINUE文は、現行のループの反復を無条件で終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの次の反復に制御を移します。

例4-13では、基本LOOP文内のCONTINUE文が無条件でカレント・ループの次の反復に制御を移します。

関連項目:

CONTINUE文

例4-13 基本LOOP文内のCONTINUE文

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
```

```
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

結果:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

4.2.5 CONTINUE WHEN文

CONTINUE WHEN文は、WHEN句内の条件がTRUEの場合、現行のループの反復を終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの次の反復に制御を移します。

制御がCONTINUE WHEN文に達するたびに、WHEN句内の条件が評価されます。条件がTRUEでない場合、CONTINUE WHEN文は何も実行しません。

例4-14では、xが3より小さい場合、基本LOOP文内のCONTINUE WHEN文がカレント・ループの次の反復に制御を移します。例4-14は例4-13と論理的に等価です。

関連項目:

CONTINUE文

例4-14 基本LOOP文内のCONTINUE WHEN文

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

結果:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

4.2.6 FOR LOOP文

FOR LOOP文は、指定された範囲にループ索引がある場合に1つ以上の文を実行します。この文の構造は、次のとおりです。

```
[ label ] FOR index IN [ REVERSE ] lower_bound..upper_bound LOOP
  statements
END LOOP [ label ];
```

REVERSEが指定されていない場合、indexの値はlower_boundから始まり、upper_boundに達するまで、ループを反復するたびに1ずつ増加します。lower_boundがupper_boundよりも大きい場合、statementsは実行されません。

REVERSEが指定されている場合、indexの値はupper_boundから始まり、lower_boundに達するまで、ループを反復するたびに1ずつ減少します。upper_boundがlower_boundよりも小さい場合、statementsは実行されません。

EXIT、EXIT WHEN、CONTINUEまたはCONTINUE WHENをstatements内に置くと、ループまたは現行のループの反復を途中で終了させることができます。

> ヒント:
>
> 問合せの結果セットの行を処理するには、整数の範囲のかわりに問合せが含まれているカーソル FOR LOOP を使用してください。詳細は、「 カーソル FOR LOOP 文による問合せ結果セットの処理」を参照してください。

関連項目:

「FOR LOOP文」

例4-15では、indexはi、lower_boundは1、upper_boundは3です。このループは1から3の数値を出力します。

例4-16のFOR LOOP文は、例4-15を反転したものであり、3から1の数値を出力します。

一部の言語のFOR LOOPには、ループ索引の増分値を1以外に指定することが可能なSTEP句があります。PL/SQLでSTEP句をシミュレートするには、ループ索引の参照のたびに、必要な増分値を乗じます。

例4-17では、FOR LOOPの索引が実質的に5ずつ増加します。

ここでのトピック

- FOR LOOP索引
- 下限と上限
- FOR LOOP文内のEXIT WHENまたはCONTINUE WHEN文

例4-15 FOR LOOP文

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');

  FOR i IN 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');

  FOR i IN 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
END;
/
```

結果:

```
lower_bound < upper_bound
1
2
3
lower_bound = upper_bound
2
lower_bound > upper_bound
```

例4-16 反転FOR LOOP文

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

  FOR i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound = lower_bound');

  FOR i IN REVERSE 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound < lower_bound');

  FOR i IN REVERSE 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
```

```
END;
/
```

結果:

```
upper_bound > lower_bound
3
2
1
upper_bound = lower_bound
2
upper_bound < lower_bound
```

例4-17 FOR LOOP文でのSTEP句のシミュレート

```
DECLARE
  step  PLS_INTEGER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i*step);
  END LOOP;
END;
/
```

結果:

```
5
10
15
```

4.2.6.1 FOR LOOP索引

FOR LOOP文の索引は、ループに対してローカルであるPLS_INTEGER型の変数として暗黙的に宣言されます。ループ内の文は索引の値を参照できますが、変更することはできません。ループの外側の文は、索引を参照できません。FOR LOOP文の実行後、索引は未定義となります。(ループ索引はループ・カウンタと呼ばれる場合があります。)

例4-18では、FOR LOOP文が索引の値を変更しようとしているため、エラーが発生します。

例4-19では、FOR LOOP文の外側の文がループ索引を参照しようとしているため、エラーが発生します。

例4-20に示すように、FOR LOOP文の索引の名前が、外側のブロックで宣言された変数の名前と同じ場合、ローカルの暗黙的な宣言によって他の宣言が隠されます。

例4-21は、Example 4-20を変更して、外側のブロックで宣言された変数をループ内の文で参照できるようにする方法を示しています。

例4-22では、ネストしたFOR LOOP文の索引で同じ名前を使用しています。内部ループでは、外部ループのラベルで参照を修飾して、外部ループの索引を参照しています。また、内部ループでは、意味を明確にするために、それ自体のラベルでそれ自体の索引への参照も修飾しています。

例4-18 索引値を変更しようとするFOR LOOP文

```
BEGIN
```

```
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```

結果:

```
       i := 2;
       *
ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored
```

例4-19 外側の文によるFOR LOOP文の索引の参照

```
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

結果:

```
  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
                                                          *
ERROR at line 6:
ORA-06550: line 6, column 58:
PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

例4-20 変数と同じ名前を持つFOR LOOP文の索引

```
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

結果:

```
Inside loop, i is 1
Inside loop, i is 2
Inside loop, i is 3
Outside loop, i is 5
```

例4-21 FOR LOOP文による索引と同じ名前を持つ変数の参照

```
<<main>>  -- Label block.
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (
      'local: ' || TO_CHAR(i) || ', global: ' ||
      TO_CHAR(main.i)  -- Qualify reference with block label.
    );
  END LOOP;
END main;
/
```

結果:

```
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5
```

例4-22 同じ索引名のネストしたFOR LOOP文

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3 LOOP
    <<inner_loop>>
    FOR i IN 1..3 LOOP
      IF outer_loop.i = 2 THEN
        DBMS_OUTPUT.PUT_LINE
          ('outer: ' || TO_CHAR(outer_loop.i) || ' inner: '
           || TO_CHAR(inner_loop.i));
      END IF;
    END LOOP inner_loop;
  END LOOP outer_loop;
END;
/
```

結果:

```
outer: 2 inner: 1
outer: 2 inner: 2
outer: 2 inner: 3
```

4.2.6.2 下限と上限

FOR LOOP文の下限と上限には、数値リテラル、数値変数または数式を使用できます。境界の数値が指定されていない場合、

PL/SQLは事前定義の例外VALUE_ERRORを呼び出します。

例4-24では、FOR LOOP文の上限が変数になっており、値は実行時に決定されます。

例4-23 FOR LOOP文の境界

```
DECLARE
  first  INTEGER := 1;
  last   INTEGER := 10;
  high   INTEGER := 100;
  low    INTEGER := 12;
BEGIN
  -- Bounds are numeric literals:
  FOR j IN -5..5 LOOP
    NULL;
  END LOOP;

  -- Bounds are numeric variables:
  FOR k IN REVERSE first..last LOOP
    NULL;
  END LOOP;

 -- Lower bound is numeric literal,
 -- Upper bound is numeric expression:
  FOR step IN 0..(TRUNC(high/low) * 2) LOOP
    NULL;
  END LOOP;
END;
/
```

例4-24 実行時に境界が指定されるFOR LOOP文

```
DROP TABLE temp;
CREATE TABLE temp (
  emp_no      NUMBER,
  email_addr  VARCHAR2(50)
);

DECLARE
  emp_count  NUMBER;
BEGIN
  SELECT COUNT(employee_id) INTO emp_count
  FROM employees;

  FOR i IN 1..emp_count LOOP
    INSERT INTO temp (emp_no, email_addr)
    VALUES(i, 'to be added later');
  END LOOP;
END;
/
```

4.2.6.3 FOR LOOP文内のEXIT WHENまたはCONTINUE WHEN文

特定の条件が発生した場合はただちにFOR LOOP文を終了する必要があるとします。FOR LOOP文内のEXIT WHEN文に、条件を指定できます。

例4-25では、FOR LOOP文が10回実行されますが、中のFETCH文から行が戻されなくなった場合は、ただちにループが終了します。

次は、途中で終了する必要のあるFOR LOOP文が、別のFOR LOOPの中にネストされているとします。内側のループが途中で終了した場合に外側のループも終了する必要があるなら、例4-26に示すように、外側のループにラベルを付け、EXIT WHEN文にラベル名を指定します。

内側のループが途中で終了しても外側のループの現行の反復を完了する必要があるなら、例4-27に示すように、外側のループにラベルを付け、CONTINUE WHEN文にラベル名を指定します。

関連項目:

例外の詳細は、「例外処理の概要」を参照してください。特定の条件が発生した場合は、例外を使用してただちにループを終了させることもできます

例4-25 FOR LOOP文内のEXIT WHEN文

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;
  -- Fetch entire row into v_employees record:
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_employees;
    EXIT WHEN c1%NOTFOUND;
    -- Process data here
  END LOOP;
  CLOSE c1;
END;
/
```

例4-26 内側のFOR LOOP文内のEXIT WHEN文

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      EXIT outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
```

```
END;
/
```

例4-27 内側のFOR LOOP文内のCONTINUE WHEN文

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

4.2.7 WHILE LOOP文

WHILE LOOP文は、条件がTRUEの場合に1つ以上の文を実行します。この文の構造は、次のとおりです。

```
[ label ] WHILE condition LOOP
  statements
END LOOP [ label ];
```

conditionがTRUEの場合は、statementsが実行され、ループの先頭に制御が戻り、そこでconditionが再度評価されます。conditionがTRUEでない場合は、WHILE LOOP文の後の文に制御が移ります。無限ループが発生しないように、ループ内の文で条件がFALSEまたはNULLになるようにする必要があります。構文の詳細は、「WHILE LOOP文」を参照してください。

EXIT、EXIT WHEN、CONTINUEまたはCONTINUE WHENをstatements内に置くと、ループまたは現行のループの反復を途中で終了させることができます。

一部の言語は、条件をループの先頭ではなく最後でテストするLOOP UNTILまたはREPEAT UNTIL構造を持っているため、一連の文は1回以上実行されます。この構造をPL/SQLでシミュレートするには、EXIT WHEN文とともに基本LOOP文を使用します。

```
LOOP
  statements
  EXIT WHEN condition;
END LOOP;
```

例4-28では、1番目のWHILE LOOP文内の文は実行されず、2番目のWHILE LOOP文内の文が1回実行されます。

例4-28 WHILE LOOP文

```
DECLARE
```

```
  done  BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print.');
    done := TRUE;  -- This assignment is not made.
  END LOOP;
  WHILE NOT done LOOP
    DBMS_OUTPUT.PUT_LINE ('Hello, world!');
    done := TRUE;
  END LOOP;
END;
/
```

結果:

```
Hello, world!
```

4.3 順次制御文

sequential control statementsであるGOTO文とNULL文は、PL/SQLプログラミングではIF文やLOOP文ほど重要ではありません。

指定した文に移動するGOTO文が必要となることはほとんどありません。ただし、これを使用すると論理を単純化できる場合もあります。

何も実行しないNULL文には、条件文の意味とアクションを明確にすることによって、コードをわかりやすくする効果があります。

ここでのトピック

- GOTO文
- NULL文

4.3.1 GOTO文

GOTO文は無条件に制御をラベルに移します。ラベルは有効範囲の中で他と重複しないもので、実行可能文かPL/SQLブロックの前に置かれている必要があります。GOTO文が実行されると、ラベルが付けられた文またはブロックに制御が移ります。GOTO文の制限については、「GOTO文」を参照してください。

GOTO文の使用は最小限にしてください(多用すると、コードの理解やメンテナンスが困難になります)。深いネスト構造から例外ハンドラに制御を移す場合、GOTO文は使用しないでください。かわりに、例外を呼び出してください。PL/SQLの例外処理メカニズムの詳細は、「PL/SQLのエラー処理」を参照してください。

ラベルは、ブロックの前(例4-21)または文の前(例4-29)にのみ置くことができます。例4-30のように文の途中に置くことはできません。

例4-30を修正する場合は、例4-31に示すように、NULL文を追加します。

GOTO文は、例4-32に示すように、カレント・ブロックから外側のブロックに制御を移すことができます。

このGOTO文は、参照されたラベルが置かれている最初の外側のブロックに制御を移しています。

例4-33では、GOTO文がIF文内に制御を移しているため、エラーが発生します。

例4-29 GOTO文

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;
  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

結果:

```
37 is a prime number
```

例4-30 不適切なラベル配置

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
       GOTO end_loop;
    END IF;
    <<end_loop>>
  END LOOP;
END;
/
```

結果:

```
  END LOOP;
  *
ERROR at line 9:
ORA-06550: line 9, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql run commit forall merge pipe purge
```

例4-31 GOTO文によるラベル付きのNULL文への移動

```
DECLARE
```

```
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    <<end_loop>>
    NULL;
  END LOOP;
END;
/
```

例4-32 GOTO文による外側のブロックへの制御の移動

```
DECLARE
  v_last_name  VARCHAR2(25);
  v_emp_id     NUMBER(6) := 120;
BEGIN
  <<get_name>>
  SELECT last_name INTO v_last_name
  FROM employees
  WHERE employee_id = v_emp_id;

  BEGIN
    DBMS_OUTPUT.PUT_LINE (v_last_name);
    v_emp_id := v_emp_id + 5;

    IF v_emp_id < 120 THEN
      GOTO get_name;
    END IF;
  END;
END;
/
```

結果:

```
Weiss
```

例4-33 GOTO文によるIF文内への無効な制御の移動

```
DECLARE
  valid BOOLEAN := TRUE;
BEGIN
  GOTO update_row;

  IF valid THEN
  <<update_row>>
    NULL;
  END IF;
END;
/
```

結果:

```
  GOTO update_row;
```

```
  *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00375: illegal GOTO statement; this GOTO cannot transfer control to label
'UPDATE_ROW'
ORA-06550: line 6, column 12:
PL/SQL: Statement ignored
```

4.3.2 NULL文

NULL文は、後続の文に制御を移すのみです。一部の言語では、このような命令をno-op(何もしない)と呼びます。

NULL文は、次のように使用できます。

- GOTO文に対するターゲットとして使用できます(例4-31を参照)。
- 条件文の意味とアクションを明確にすることによって、わかりやすさを向上できます(例4-34を参照)。
- プレースホルダおよびスタブ・サブプログラムを作成できます(例4-35を参照)。
- 考慮はするがアクションは必要ないことを示すことができます(例4-36を参照)。

例4-34では、NULL文によって、販売員のみがコミッションを受け取れることを明確にしています。

例4-35では、NULL文を使用してサブプログラムをコンパイルしています。実際の本体は、後で挿入できます。

> ノート:
>
> NULL 文を使用する場合に警告が有効になっていると、unreachable code という警告が発生する可能性があります。警告の詳細は、「コンパイル時の警告」を参照してください。

例4-36では、NULL文によって、A、B、C、D、F以外のgradeではアクションを起こさないことを示しています。

例4-34 アクションを実行しないことを明示するNULL文

```
DECLARE
  v_job_id  VARCHAR2(10);
   v_emp_id  NUMBER(6) := 110;
BEGIN
  SELECT job_id INTO v_job_id
  FROM employees
  WHERE employee_id = v_emp_id;

  IF v_job_id = 'SA_REP' THEN
    UPDATE employees
    SET commission_pct = commission_pct * 1.2;
  ELSE
    NULL;  -- Employee is not a sales rep
  END IF;
END;
/
```

例4-35 サブプログラム作成時のプレースホルダとしてのNULL文

```
CREATE OR REPLACE PROCEDURE award_bonus (
```

```
  emp_id NUMBER,
  bonus NUMBER
) AUTHID DEFINER AS
BEGIN    -- Executable part starts here
  NULL;  -- Placeholder
  -- (raises "unreachable code" if warnings enabled)
END award_bonus;
/
```

例4-36 単純なCASE文のELSE句でのNULL文

```
CREATE OR REPLACE PROCEDURE print_grade (
  grade CHAR
) AUTHID DEFINER AS
BEGIN
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE NULL;
  END CASE;
END;
/
BEGIN
  print_grade('A');
  print_grade('S');
END;
/
```

結果:

```
Excellent
```

5 PL/SQLのコレクションとレコード

PL/SQLでは、コレクションとレコードという、2種類のコンポジット・データ型を定義できます。

コンポジット・データ型には、内部コンポーネントを持つ値が格納されます。コンポジット変数全体をサブプログラムにパラメータとして渡すことや、コンポジット変数の内部コンポーネントに個別にアクセスすることができます。内部コンポーネントは、スカラーまたはコンポジットのいずれかです。スカラー・コンポーネントは、スカラー変数を使用できるすべての場所で使用できます。コンポジット・コンポーネントは、同じ型のコンポジット変数を使用できるすべての場所で使用できます。

> ノート:
>
> コンポジット変数をパラメータとしてリモート・サブプログラムに渡した場合、冗長なループバック DATABASE LINK を作成する必要がありますが、これにより、リモート・サブプログラムがコンパイルされたときに、ソースを検証するタイプ・チェッカが、実行者の使用しているものと同じユーザー定義のコンポジット変数型定義を使用するようになります。

collectionの内部コンポーネントは、常に同じデータ型であり、要素と呼ばれます。コレクション変数の各要素には、この variable_name(index)という構文を使用して、その一意の索引によってアクセスできます。コレクション変数を作成するには、コレクション型を定義してからその型の変数を作成するか、%TYPEを使用します。

レコードの内部コンポーネントは、データ型が異なる場合があり、フィールドと呼ばれます。レコード変数の各フィールドには、variable_name.field_nameという構文を使用して、その名前によってアクセスできます。レコード変数を作成するには、RECORD型を定義してからその型の変数を作成するか、%ROWTYPEまたは%TYPEを使用します。

レコードのコレクション、およびコレクションを含むレコードを作成できます。

コレクションに関するトピック

- コレクション型
- 連想配列
- VARRAY (可変サイズの配列)
- ネストした表
- コレクションのコンストラクタ
- 修飾式の概要
- コレクション変数への値の代入
- 多次元コレクション
- コレクションの比較
- コレクション・メソッド
- パッケージ仕様部で定義されたコレクション型

関連項目:

- CREATE DATABASE LINK文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- コレクションの問合せ
- コレクションへの問合せ結果の取出しの詳細は、「BULK COLLECT句」を参照してください
- コレクション型の定義およびコレクション変数の宣言の構文とセマンティクスは、「コレクション変数の宣言」を参照してください

レコードに関するトピック

- レコード変数
- レコード変数への値の代入
- レコードの比較
- 表へのレコードの挿入
- レコードを使用した行の更新
- レコードの挿入/更新に関する制限

ノート:

明示的にリストされたコンポジット・データ構造のコンポーネント(コレクション・コンストラクタまたはレコード・イニシャライザなど)はどの順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

5.1 コレクション型

PL/SQLには、連想配列、VARRAY (可変サイズの配列)およびネストした表の3つのコレクション型があります。

表5-1に、それぞれの類似点と相違点の概要を示します。

表5-1 PL/SQLコレクション型

コレクション型	要素の数	索引タイプ	密か疎か	未初期化時のステータス	定義される場所	ADT属性のデータ型として使用可能かどうか
連想配列(または索引付き表)	指定なし	文字列または PLS_INTEGER	密または疎のいずれか	なし	PL/SQL ブロック内またはパッケージ内	不可
VARRAY(可変サイズの配列)	指定されている	整数	常に密	Null	PL/SQL ブロック内、パッケージ内またはスキーマ・レベ	スキーマ・レベルで定義されている場合のみ

コレクション型	要素の数	索引タイプ	密か疎か	未初期化時のステータス	定義される場所	ADT属性のデータ型として使用可能かどうか
					ル	
ネストした表	指定なし	整数	密で始まり、疎になる可能性あり	Null	PL/SQL ブロック内、パッケージ内またはスキーマ・レベル	スキーマ・レベルで定義されている場合のみ

要素の数

要素の数が指定されている場合、その数がコレクション内の要素の最大数です。要素の数が指定されていない場合は、索引タイプの上限がコレクション内の要素の最大数になります。

密か疎か

dense collectionとは、要素間に欠損がないコレクションのことで、最初と最後の要素の間にあるすべての要素が定義され、値を含んでいます(要素にNOT NULL制約が指定されていない場合にかぎり、値にNULLを使用できます)。疎コレクションは、要素間に欠損があります。

未初期化時のステータス

空のコレクションは、要素を持たずに存在するコレクションです。空のコレクションに要素を追加するには、EXTENDメソッド(「EXTENDコレクション・メソッド」を参照)を起動します。

null collection(atomically null collectionとも呼ばれる)は存在しません。NULLのコレクションを存在するコレクションに変えるには、空にするかNULL以外の値を代入してコレクションを初期化する必要があります(詳細は「コレクションのコンストラクタ」および「コレクション変数への値の代入」を参照してください)。EXTENDメソッドを使用して、NULLのコレクションを初期化することはできません。

定義される場所

PL/SQLブロック内に定義されるコレクション型はローカル型です。ブロック内でのみ使用可能であり、スタンドアロン・サブプログラムまたはパッケージ・サブプログラム内にブロックがある場合にのみ、データベースに格納されます。(スタンドアロン・サブプログラムおよびパッケージ・サブプログラムの詳細は、ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラムを参照してください。)

パッケージ仕様部に定義されるコレクション型はパブリック項目です。パッケージ名(package_name. type_name)で修飾することで、パッケージの外から参照できます。パッケージを削除するまでデータベースに格納されます。(パッケージについては、「PL/SQLパッケージ」を参照してください。)

スキーマ・レベルで定義されるコレクション型は、スタンドアロン型です。「CREATE TYPE文」を使用して作成します。「DROP TYPE文」を使用して削除するまでデータベースに格納されます。

ノート:

パッケージ仕様部で定義されたコレクション型は、同一定義のローカル・コレクション型またはスタンドアロン・コレクション型と互換性がありません(例 5-33 および例 5-34 を参照)。

ADT属性のデータ型として使用可能かどうか

ADT属性のデータ型にするには、コレクション型がスタンドアロン・コレクション型である必要があります。その他の制限は、「datatypeの制限」を参照してください。

非PL/SQLコンポジット型からPL/SQLコンポジット型への変換

他の言語を使用するコードまたはビジネス・ロジックが存在する場合、通常は配列を変換して、その言語の型をPL/SQLのコレクション型に直接設定できます。たとえば:

非PL/SQLコンポジット型	等価なPL/SQLコンポジット型
ハッシュ表	連想配列
順序付けされていない表	連想配列
集合	ネストした表
バッグ	ネストした表
配列	VARRAY

関連項目:

あるSQLデータ型またはコレクション型の値を別のSQLデータ型またはコレクション型の値に変換するCASTファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

5.2 連想配列

連想配列(旧称はPL/SQL表または索引付き表)は、キーと値のペアのセットです。各キーは一意の索引であり、構文variable_name(index)を使用して、関連する値を検索するために使用されます。

索引のデータ型には、文字列型(VARCHAR2、VARCHAR、STRINGまたはLONG)またはPLS_INTEGERを使用できます。索引は、作成された順序ではなく、ソートされた順序で格納されます。文字列型の場合、ソートの順序は初期化パラメータNLS_SORTおよびNLS_COMPで決定されます。

データベース表と同様に、連想配列には次の特性があります。

- 移入するまで空である(ただし、NULLではない)
- 数が指定されていない要素を保持できる(要素の位置を知らなくてもアクセスできる)

連想配列には、データベース表とは異なる次の特性があります。

- ディスク領域またはネットワーク操作が不要
- DML文では操作できない

ここでのトピック

- 連想配列の定数の宣言
- 文字列で索引付けされている結合配列に影響を与えるNLSパラメータ値
- 結合配列の適切な使用方法

関連項目:

- 表5-1に、結合配列の特性の概要を示します。
- 結合配列型の定義の構文は、「assoc_array_type_def ::=」を参照してください

例5-1 文字列で索引付けされている連想配列

この例では、文字列で索引付けされる連想配列型を定義し、この型の変数を宣言して3つの要素を変数に移入し、1つの要素の値を変更した後に値を(作成された順序ではなくソートされた順序で)出力します。(FIRSTおよびNEXTはコレクション・メソッドです。詳細は「コレクション・メソッド」を参照してください。)

Live SQL:

この例は、Oracle Live SQL の「文字列で索引付けされている連想配列」で表示および実行できます

```
DECLARE
```

-- Associative array indexed by string: TYPE population IS TABLE OF NUMBER -- Associative array type INDEX BY VARCHAR2(64); -- indexed by string city_population population; -- Associative array variable i VARCHAR2(64); -- Scalar variable BEGIN -- Add elements (key-value pairs) to associative array: city_population('Smallville') := 2000; city_population('Midland') := 750000; city_population('Megalopolis') := 1000000; -- Change value associated with key 'Smallville': city_population('Smallville') := 2001; -- Print associative array: i := city_population.FIRST; -- Get first element of array WHILE i IS NOT NULL LOOP DBMS_Output.PUT_LINE ('Population of ' || i || ' is ' || city_population(i)); i := city_population.NEXT(i); -- Get next element of array END LOOP; END; /

結果:

```
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001
```

例5-2 PLS_INTEGERで索引付けされている連想配列を戻すファンクション

この例では、PLS_INTEGERで索引付けされている連想配列型と、その型の連想配列を戻すファンクションを定義します。

Live SQL:

この例は、Oracle Live SQL の「PLS_INTEGER で索引付けされている連想配列を戻すファンクション」で表示および実行できます

```
DECLARE
```

TYPE sum_multiples IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER; n PLS_INTEGER := 5; -- number of multiples to sum for display sn PLS_INTEGER := 10; -- number of multiples to sum m PLS_INTEGER := 3; -- multiple FUNCTION get_sum_multiples (multiple IN PLS_INTEGER, num IN PLS_INTEGER) RETURN sum_multiples IS s sum_multiples; BEGIN FOR i IN 1..num LOOP s(i) := multiple * ((i * (i + 1)) / 2); -- sum of multiples END LOOP; RETURN s; END get_sum_multiples; BEGIN DBMS_OUTPUT.PUT_LINE ('Sum of the first ' || TO_CHAR(n) || ' multiples of ' || TO_CHAR(m) || ' is ' || TO_CHAR(get_sum_multiples (m, sn)(n))); END; /

結果:

```
Sum of the first 5 multiples of 3 is 45
```

5.2.1 連想配列の定数の宣言

連想配列の定数を宣言する場合、初期値を連想配列に移入するファンクションを作成して、定数宣言でそのファンクションを起動する必要があります。

コンストラクタの詳細は、「コレクションのコンストラクタ」を参照してください。

例5-3 連想配列の定数の宣言

この例では、コンストラクタがVARRAYまたはネストした表に対して行う操作を、ファンクションが連想配列に対して行います。

Live SQL:

この例は、Oracle Live SQL の「連想配列の定数の宣言」で表示および実行できます

```
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
```

TYPE My_AA IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER; FUNCTION Init_My_AA RETURN My_AA; END My_Types; / CREATE OR REPLACE PACKAGE BODY My_Types IS FUNCTION Init_My_AA RETURN My_AA IS Ret My_AA; BEGIN Ret(-10) := '-ten'; Ret(0) := 'zero'; Ret(1) := 'one'; Ret(2) := 'two'; Ret(3) := 'three'; Ret(4) := 'four'; Ret(9) := 'nine'; RETURN Ret; END Init_My_AA; END My_Types; / DECLARE v CONSTANT My_Types.My_AA := My_Types.Init_My_AA(); BEGIN DECLARE Idx PLS_INTEGER := v.FIRST(); BEGIN WHILE Idx IS

NOT NULL LOOP DBMS_OUTPUT.PUT_LINE(TO_CHAR(Idx, '999')||LPAD(v(Idx), 7)); Idx := v.NEXT(Idx); END LOOP; END; END; /

結果:

```
-10   -ten
0   zero
1    one
2    two
3  three
4   four
9   nine

PL/SQL procedure successfully completed.
```

5.2.2 文字列で索引付けされている結合配列に影響を与えるNLSパラメータ値

文字列で索引付けされている連想配列は、NLS_SORT、NLS_COMP、NLS_DATE_FORMATなどの各国語サポート(NLS)のパラメータによって影響を受けます。

ここでのトピック

- 結合配列に移入した後のNLSパラメータ値の変更
- VARCHAR2以外のデータ型の索引
- リモート・データベースへの結合配列の受渡し

関連項目:

言語ソート・パラメータの詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください。

5.2.2.1 結合配列に移入した後のNLSパラメータ値の変更

連想配列の文字列索引の格納順序は、初期化パラメータNLS_SORTおよびNLS_COMPで決まります。

文字列で索引付けされている連想配列にデータを移入した後で、いずれかのパラメータ値を変更すると、コレクション・メソッドFIRST、LAST、NEXTおよびPRIORから予期しない値が戻されたり例外が呼び出される場合があります。セッション中にこれらのパラメータ値を変更する必要がある場合は、文字列で索引付けされている連想配列に対して操作を実行する前に元の値をリストアしてください。

関連項目:

FIRST、LAST、NEXTおよびPRIORの詳細は、「コレクション・メソッド」を参照してください

5.2.2.2 VARCHAR2以外のデータ型の索引

文字列で索引付けされている連想配列の宣言では、文字列型をVARCHAR2またはそのサブタイプの1つにする必要があります。

ただし、連想配列への移入では、TO_CHARファンクションでVARCHAR2に変換できる任意のデータ型の索引を使用できます。

VARCHAR2およびそのサブタイプ以外のデータ型の索引を使用する場合は、初期化パラメータの値が変更されても索引の一貫性および一意性が保持されることを確認してください。たとえば:

- TO_CHAR(SYSDATE)は索引として使用しないでください。

  NLS_DATE_FORMATの値が変更された場合は、(TO_CHAR(SYSDATE))も変更される可能性があります。

- それぞれ異なるNVARCHAR2の索引でも、同じVARCHAR2値に変換される可能性のある場合は使用しないでください。
- 大/小文字、アクセント記号またはデリミタ文字のみが異なるCHARまたはVARCHAR2の索引を使用しないでください。

  NLS_SORTの値の末尾が_CI(大/小文字を区別しない比較)または_AI(アクセント記号の有無および大/小文字を区別しない比較)である場合、大/小文字、アクセント記号またはデリミタ文字のみが異なる索引は同じ値に変換される可能性があります。

関連項目:

TO_CHARの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

5.2.2.3 リモート・データベースへの結合配列の受渡し

連想配列をパラメータとしてリモート・データベースに渡す場合に、ローカルとリモートのデータベースのNLS_SORT値またはNLS_COMP値が異なると、次のようになります。

- コレクション・メソッドFIRST、LAST、NEXTまたはPRIOR(「コレクション・メソッド」を参照)から予期しない値が戻されたり例外が呼び出される場合があります。
- ローカル・データベース上では一意の索引でも、リモート・データベース上では一意でない可能性があり、一意でない場合は事前定義の例外VALUE_ERRORが呼び出されます。

5.2.3 結合配列の適切な使用方法

連想配列は、次のような場合に適切です。

- 表が宣言されているサブプログラムの起動またはパッケージの初期化のたびにメモリー内に構成できる比較的小さい参照表
- データベース・サーバーとの間のコレクションの受渡し

  連想配列型の仮サブプログラム・パラメータを宣言します。Oracle Call Interface(OCI)またはOracleプリコンパイラを使用すると、対応する実パラメータにホスト配列がバインドされます。PL/SQLでは、PLS_INTEGERで索引付けされているホスト配列と連想配列との間で自動的に変換が行われます。

ノート:

VARCHAR で索引付けされている連想配列はバインドできません。

ノート:

スキーマ・レベルでは結合配列型を宣言できません。そのため、連想配列の変数をパラメータとしてスタンドアロン・サブプログラムに渡すには、その変数の型をパッケージ仕様部で宣言する必要があります。このようにすることで、起動されるサブプログラム(結合配列型の仮パラメータを宣言する側)と起動元のサブプログラムまたは無名ブロック(結合配列型の変数を宣言して受け渡す側)の両方で結合配列型を使用できます。例 10-2 を参照してください。

ヒント:

データベース・サーバーとの間でのコレクションの受渡しには、連想配列を FORALL 文または BULK COLLECT 句とともに使用することが、最も効率的な方法です。詳細は、「FORALL 文」および「BULK COLLECT 句」を参照してください。

連想配列は、一時的なデータの格納に使用されます。連想配列をデータベース・セッションの期間中持続させるには、パッケージ仕様部で連想配列を宣言し、パッケージ本体で移入します。

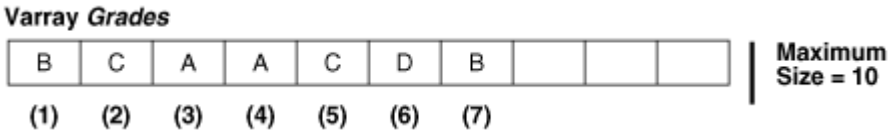
5.3 VARRAY (可変サイズの配列)

VARRAY(可変サイズの配列)とは、0個(空の場合)から宣言されている最大サイズまで要素の数を変更できる配列のことです。

VARRAY変数の要素にアクセスするには、構文variable_name(index)を使用します。indexの下限は1で、上限は現行の要素の数です。上限は、要素の追加または削除に合わせて変化しますが、最大サイズを超えることはできません。VARRAYをデータベースに格納したりデータベースから取り出しても、その索引と要素の順序は変わりません。

図5-1 は、Gradesという名前のVARRAY変数を示していて、最大サイズは10個で、7個の要素が格納されています。Grades(n)は、Gradesのn番目の要素を参照します。Gradesの上限は7で、10を超えることはできません。

図5-1 7個の要素を持つ最大サイズが10個のVARRAY



VARRAY変数は単一オブジェクトとしてデータベースに格納されます。VARRAY変数が4KB未満の場合は、そのVARRAY変数が列になっている表内に配置され、それ以外の場合は、表の外側に配置されますが、表領域は同じです。

初期化されていないVARRAY変数は、NULLのコレクションです。空にするかNULL以外の値を代入して初期化する必要があります。詳細は、「コレクションのコンストラクタ」および「コレクション変数への値の代入」を参照してください。

ここでのトピック

- VARRAYの適切な使用方法

関連項目:

- 表5-1に、VARRAYの特性の概要を示します。

- VARRAY型の定義の構文は、「varray_type_def ::=」を参照してください
- スタンドアロンVARRAY方の作成の詳細は、「CREATE TYPE文」を参照してください
- VARRAYの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例5-4 VARRAY(可変サイズの配列)

この例では、ローカルのVARRAY型を定義した後に、この型の変数を宣言(コンストラクタを使用して初期化)し、このVARRAYを出力するプロシージャを定義します。この例では、このプロシージャを3回起動します(変数を初期化した後、2つの要素の値をそれぞれ変更した後、およびコンストラクタを使用してすべての要素の値を変更した後に1回ずつ)。(NULLまたは空の可能性があるVARRAYを出力するプロシージャの例は、例5-26を参照してください。)

Live SQL:

この例は、Oracle Live SQL の「VARRAY(可変サイズの配列)」で表示および実行できます

```
DECLARE
TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type -- varray variable initialized with constructor: team Foursome := Foursome('John', 'Mary', 'Alberto', 'Juanita'); PROCEDURE print_team (heading VARCHAR2) IS BEGIN DBMS_OUTPUT.PUT_LINE(heading); FOR i IN 1..4 LOOP DBMS_OUTPUT.PUT_LINE(i || '.' || team(i)); END LOOP; DBMS_OUTPUT.PUT_LINE('---'); END; BEGIN print_team('2001 Team:'); team(3) := 'Pierre'; -- Change values of two elements team(4) := 'Yvonne'; print_team('2005 Team:'); -- Invoke constructor to assign new values to varray variable: team := Foursome('Arun', 'Amitha', 'Allan', 'Mae'); print_team('2009 Team:'); END; /
```

結果:

```
2001 Team:
1.John
2.Mary
3.Alberto
4.Juanita
---
2005 Team:
1.John
2.Mary
3.Pierre
4.Yvonne
---
2009 Team:
1.Arun
2.Amitha
3.Allan
4.Mae
---
```

5.3.1 VARRAYの適切な使用方法

VARRAYは、次のような場合に適切です。

- 要素の最大数がわかっている。
- 通常は要素に順番にアクセスする。

VARRAYはすべての要素を同時に格納または取得する必要があるため、要素の数が多い場合は現実的ではないことがあります。

5.4 ネストした表

データベース内では、ネストした表は、特に順序を付けずに数が指定されていない行を格納する列型になります。

ネストした表の値をデータベースから取り出してPL/SQLのネストした表の変数に入れると、PL/SQLにより1から始まる連続した索引が行に付けられます。これらの索引を使用して、ネストした表の変数の個々の行にアクセスできます。構文は variable_name(index)です。ネストした表の索引と行の順序は、ネストした表をデータベースに格納したりデータベースから取り出したときに変わる可能性があります。

ネストした表の変数に占有されるメモリー量は、要素の追加または削除に応じて動的に増減します。

初期化されていないネストした表の変数は、NULLのコレクションです。空にするかNULL以外の値を代入して初期化する必要があります。詳細は、「コレクションのコンストラクタ」および「コレクション変数への値の代入」を参照してください。

ノート:

nt_type および print_nt は、例 5-19、例 5-21 および例 5-22 で再利用します。

ここでのトピック

- ネストした表と配列の重要な相違点
- ネストした表の適切な使用方法

関連項目:

- 表5-1に、ネストした表の特性の概要を示します。
- ネストした表型の定義の構文は、「nested_table_type_def ::=」を参照してください
- スタンドアロン型のネストした表型の作成の詳細は、「CREATE TYPE文」を参照してください
- ネストした表のビューの列を更新するトリガーの詳細は、「INSTEAD OF DMLトリガー」を参照してください
- ネストした表の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例5-5 ローカル型のネストした表

この例では、ローカルのネストした表型を定義した後に、この型の変数を宣言(コンストラクタを使用して初期化)し、このネストし

た表を出力するプロシージャを定義します。(このプロシージャでは、コレクション・メソッドFIRSTおよびLASTを使用しています。詳細は「コレクション・メソッド」を参照してください。)この例では、このプロシージャを3回起動します(変数を初期化した後、1つの要素の値を変更した後、コンストラクタを使用してすべての要素の値を変更した後に1回ずつ)。2回目にコンストラクタを起動した後は、ネストした表の要素が2つのみになります。要素3を参照すると、エラーORA-06533が発生します。

Live SQL:

この例は、Oracle Live SQL の「ローカル型のネストした表」で表示および実行できます

```
DECLARE
```

TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type -- nested table variable initialized with constructor: names Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh'); PROCEDURE print_names (heading VARCHAR2) IS BEGIN DBMS_OUTPUT.PUT_LINE(heading); FOR i IN names.FIRST .. names.LAST LOOP -- For first to last element DBMS_OUTPUT.PUT_LINE(names(i)); END LOOP; DBMS_OUTPUT.PUT_LINE('---'); END; BEGIN print_names('Initial Values:'); names(3) := 'P Perez'; -- Change value of one element print_names('Current Values:'); names := Roster('A Jansen', 'B Gupta'); -- Change entire table print_names('Current Values:'); END; /

結果:

```
Initial Values:
D Caruso
J Hamil
D Piro
R Singh
---
Current Values:
D Caruso
J Hamil
P Perez
R Singh
---
Current Values:
A Jansen
B Gupta
```

例5-6 スタンドアロン型のネストした表

この例では、スタンドアロン型のネストした表型nt_typeを定義し、この型の変数を出力するスタンドアロン・プロシージャprint_ntを定義します。無名ブロックで型nt_typeの変数を宣言し、この変数をコンストラクタで空に初期化してから、print_ntを2回起動します(変数を初期化した後、およびコンストラクタですべての要素の値を変更した後に1回ずつ)。

Live SQL:

この例は、Oracle Live SQL の「スタンドアロン型のネストした表」で表示および実行できます

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
```

/ CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS i NUMBER; BEGIN i := nt.FIRST; IF i IS NULL THEN DBMS_OUTPUT.PUT_LINE('nt is empty'); ELSE WHILE i IS NOT NULL LOOP DBMS_OUTPUT.PUT('nt.(' || i || ') = '); DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL')); i := nt.NEXT(i); END LOOP; END IF; DBMS_OUTPUT.PUT_LINE('---'); END print_nt; / DECLARE nt nt_type := nt_type(); -- nested table variable initialized to empty BEGIN print_nt(nt); nt := nt_type(90, 9, 29, 58); print_nt(nt); END; /

結果:

```
nt is empty
---
nt.(1) = 90
nt.(2) = 9
nt.(3) = 29
nt.(4) = 58
---
```

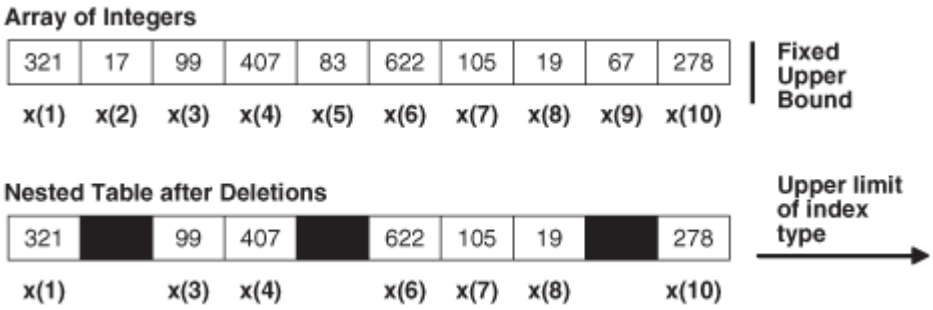
5.4.1 ネストした表と配列の重要な相違点

概念上、ネストした表は任意の数の要素がある1次元配列に似ています。ただし、ネストした表と配列には次の重要な相違点があります。

- 配列では要素の数が宣言されますが、ネストした表では宣言されません。ネストした表のサイズは動的に増やすことができます。
- 配列は常に密です。ネストした表は、最初は密であっても、要素を削除できるため、疎になる可能性があります。

図5-2は、ネストした表と配列の重要な相違点を示しています。

図5-2 配列およびネストした表



5.4.2 ネストした表の適切な使用方法

ネストした表は、次のような場合に適切です。

- 要素の数が設定されていない。
- 索引の値が連続していない。
- すべての要素を同時にではなく、一部の要素を削除または更新する必要がある。

ネストした表のデータは、別の記憶域表(システム生成によるデータベース表)に格納されます。ネストした表にアクセスすると、データベースによってその表がデータベースの記憶域表に結合されます。この記憶域表によって、ネストした表は、コレクションの一部の要素にのみ影響を与える問合せと更新に適した内容になります。

- メインの表の各行に複数のエントリを含む個別の参照表を作成し、結合問合せを介してその表にアクセスする。

5.5 コレクションのコンストラクタ

コレクションのコンストラクタ(コンストラクタ)は、コレクション型と同じ名前のシステム定義ファンクションです(この型のコレクションを戻します)。

ノート:

このトピックの内容は、VARRAY およびネストした表にのみ適用されます。このトピックでは、コレクションは VARRAY またはネストした表のことを意味します。連想配列は、修飾式と集計(修飾式の概要を参照)を使用します。

次にコンストラクタを起動する構文を示します。

```
collection_type ( [ value [, value ]... ] )
```

パラメータ・リストが空の場合は、コンストラクタから空のコレクションが戻されます。それ以外の場合は、指定した値を含むコレクションが戻されます。セマンティクスの詳細は、「collection_constructor」を参照してください。

戻されたコレクションは、変数の宣言およびブロックの実行部で(同じ型の)コレクション変数に代入できます。

例5-7 コレクション(VARRAY)変数の空への初期化

この例では、コンストラクタを2回起動します(変数の宣言でVARRAY変数teamを空に初期化するとき、およびブロックの実行部で新しい値を指定するときに1回ずつ)。プロシージャprint_teamは、teamの初期値と最終的な値を表示します。print_teamでは、teamが空であるかどうかを判断するために、コレクション・メソッドCOUNTを使用しています(詳細は「コレクション・メソッド」を参照)。(NULLの可能性があるVARRAYを出力するプロシージャの例は、例5-26を参照してください。)

Live SQL:

この例は、Oracle Live SQL の「コレクション(VARRAY)変数の空への初期化」で表示および実行できます

```
DECLARE
```

TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); team Foursome := Foursome(); -- initialize to empty PROCEDURE print_team (heading VARCHAR2) IS BEGIN DBMS_OUTPUT.PUT_LINE(heading); IF team.COUNT = 0 THEN DBMS_OUTPUT.PUT_LINE('Empty'); ELSE FOR i IN 1..4 LOOP DBMS_OUTPUT.PUT_LINE(i || '.' || team(i)); END LOOP; END IF; DBMS_OUTPUT.PUT_LINE('---'); END; BEGIN print_team('Team:'); team := Foursome('John', 'Mary', 'Alberto', 'Juanita'); print_team('Team:'); END; /

結果:

```
Team:
Empty
---
Team:
1. John
2. Mary
3. Alberto
4. Juanita
---
```

5.6 修飾式の概要

修飾式を使用すると、複雑な値を簡潔な形式で、値が必要になった箇所で宣言および定義できるため、プログラムが明確になり開発者の生産性も向上します。

修飾式は式の要素を結合して、RECORD型または連想配列型の値を作成します。

修飾式は明示的な型の指定を使用することで修飾項目の型を指定します。この明示的な指定はtypemarkと呼ばれるものです。

修飾式の構文は次のとおりです。

qualified_expression ::= typemark (aggregate)

aggregate ::= [positional_choice_list] [explicit_choice_list]

positional_choice_list ::= (expr)+

explicit_choice_list ::= named_choice_list | indexed_choice_list

named_choice_list ::= identifier => expr [,]+

indexed_choice_list ::= expr => expr [,] +

構文およびセマンティクスの詳細は、qualified_expression ::=を参照してください。

例5-8 修飾式を使用したRECORD型変数への値の代入

この例は、RECORD型変数の宣言、初期化および定義を示しています。パッケージpkg内でrec_t型が定義されていて、一部が初期化されています。その型を使用して変数v_rec1が宣言され、位置表記法を使用して初期値が代入されています。その型を使用して変数v_rec2が宣言され、名前関連付け構文を使用して初期値が代入されています。変数v_rec3にはNULL値が代入されています。プロシージャprint_recはローカル変数v_rec1の値を表示し、続いてプロシージャ・パラメータpi_rec変数値を表示します。プロシージャにパラメータが渡されない場合は、プロシージャ定義に設定されている初期値が表示されます。

Live SQL:

この例は、18c 修飾式を使用した RECORD 型変数への値の代入の Oracle Live SQL で表示および実行できます

```
CREATE PACKAGE pkg IS
```

TYPE rec_t IS RECORD (year PLS_INTEGER := 2, name VARCHAR2 (100)); END;

```
DECLARE
  v_rec1 pkg.rec_t := pkg.rec_t(1847,'ONE EIGHT FOUR SEVEN');
  v_rec2 pkg.rec_t := pkg.rec_t(year => 1, name => 'ONE');
  v_rec3 pkg.rec_t := pkg.rec_t(NULL,NULL);
PROCEDURE print_rec ( pi_rec pkg.rec_t := pkg.rec_t(1847+1,  'a'||'b')) IS
  v_rec1 pkg.rec_t := pkg.rec_t(2847,'TWO EIGHT FOUR SEVEN');
BEGIN
  DBMS_OUTPUT.PUT_LINE(NVL(v_rec1.year,0) ||' ' ||NVL(v_rec1.name,'N/A'));
  DBMS_OUTPUT.PUT_LINE(NVL(pi_rec.year,0) ||' ' ||NVL(pi_rec.name,'N/A'));
END;
BEGIN
  print_rec(v_rec1);
  print_rec(v_rec2);
  print_rec(v_rec3);
  print_rec();
END;

2847 TWO EIGHT FOUR SEVEN
1847 ONE EIGHT FOUR SEVEN
2847 TWO EIGHT FOUR SEVEN
1 ONE
2847 TWO EIGHT FOUR SEVEN
0 N/A
2847 TWO EIGHT FOUR SEVEN
1848 ab
```

例5-9 修飾式を使用した連想配列型変数への値の代入

この例では、ファンクションを使用してBOOLEANの表の値を表示します。

Live SQL:

この例は、18c 修飾式を使用した RECORD 型変数への値の代入の Oracle Live SQL で表示および実行できます

```
CREATE FUNCTION print_bool (v IN BOOLEAN)
```

RETURN VARCHAR2 IS v_rtn VARCHAR2(10); BEGIN CASE v WHEN TRUE THEN v_rtn := 'TRUE'; WHEN FALSE THEN v_rtn := 'FALSE'; ELSE v_rtn := 'NULL'; END CASE; RETURN v_rtn; END print_bool;

変数v_aa1は索引キーと値のペアを使用して初期化されます。

```
DECLARE
  TYPE t_aa IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
  v_aa1 t_aa := t_aa(1=>FALSE,
                     2=>TRUE,
```

```
                        3=>NULL);
BEGIN
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(1)));
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(2)));
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(3)));
END;

FALSE
TRUE
NULL
```

5.7 コレクション変数への値の代入

コレクション変数には次の方法で値を代入できます。

- コンストラクタを起動してコレクションを作成し、それをコレクション変数に代入します。
- 代入文を使用して、別の既存のコレクション変数の値を代入します。
- OUTパラメータまたはIN OUTパラメータとしてサブプログラムに渡し、サブプログラム内で値を代入する方法。
- 修飾式を使用して、連想配列に値を代入します(例5-9を参照)。

コレクション変数のスカラー要素に値を代入するには、collection_variable_name(index)として要素を参照し、値を代入します。

ここでのトピック

- データ型の互換性
- VARRAYまたはネストした表の変数へのNULL値の代入
- ネストした表の変数へのSET演算結果の代入

関連項目:

- コレクションのコンストラクタ
- 「代入文」の構文図
- コレクション変数のスカラー要素に値を代入する方法については、「変数への値の代入」を参照してください
- BULK COLLECT句

5.7.1 データ型の互換性

あるコレクションをコレクション変数に代入できるのは、両者のデータ型が同じ場合のみです。要素型が同じであることのみでは不十分です。

例5-10 コレクション代入のデータ型の互換性

この例では、VARRAY型のtripletとtrioが同じ要素型VARCHAR(15)になっています。コレクション変数group1とgroup2のデータ型は同じtripletですが、コレクション変数group3のデータ型はtrioです。group1からgroup2への代入は正常に行われますが、group1からgroup3への代入は失敗します。

Live SQL:

この例は、Oracle Live SQL の「コレクション代入のデータ型の互換性」で表示および実行できます

```
DECLARE
```

TYPE triplet IS VARRAY(3) OF VARCHAR2(15); TYPE trio IS VARRAY(3) OF VARCHAR2(15); group1 triplet := triplet('Jones', 'Wong', 'Marceau'); group2 triplet; group3 trio; BEGIN group2 := group1; -- succeeds group3 := group1; -- fails END; /

結果:

```
ORA-06550: line 10, column 13:
PLS-00382: expression is of wrong type
```

5.7.2 VARRAYまたはネストした表の変数へのNULL値の代入

VARRAYまたはネストした表の変数には、値NULLまたは同じデータ型のNULLのコレクションを代入できます。いずれの代入でも、変数はNULLになります。

例5-11では、ネストした表の変数dept_namesをNULL以外の値に初期化します。次に、NULLのコレクションを代入して変数をNULLにした後、別のNULL以外の値に再初期化します。

例5-11 ネストした表の変数へのNULL値の代入

Live SQL:

この例は、Oracle Live SQL の「ネストした表の変数への NULL 値の代入」で表示および実行できます

```
DECLARE
```

TYPE dnames_tab IS TABLE OF VARCHAR2(30); dept_names dnames_tab := dnames_tab('Shipping','Sales','Finance','Payroll'); -- Initialized to non-null value empty_set dnames_tab; -- Not initialized, therefore null PROCEDURE print_dept_names_status IS BEGIN IF dept_names IS NULL THEN DBMS_OUTPUT.PUT_LINE('dept_names is null.'); ELSE DBMS_OUTPUT.PUT_LINE('dept_names is not null.'); END IF; END print_dept_names_status; BEGIN print_dept_names_status; dept_names := empty_set; -- Assign null collection to dept_names. print_dept_names_status; dept_names := dnames_tab ('Shipping','Sales','Finance','Payroll'); -- Re-initialize dept_names print_dept_names_status; END; /

結果:

```
dept_names is not null.
dept_names is null.
dept_names is not null.
```

5.7.3 ネストした表の変数へのSET演算結果の代入

ネストした表の変数には、SQL MULTISET演算またはSQL SETファンクションの起動結果を代入できます。

SQLのMULTISET演算子は、2つのネストした表を1つのネストした表に統合します。2つのネストした表の要素は、同等のデータ型を持っている必要があります。MULTISET演算子の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

SQL SETファンクションはネストした表を引数に取り、別個の要素を含む同じデータ型のネストした表を戻します(重複する要素は結果から削除されます)。SETファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例5-12 ネストした表の変数へのSET演算結果の代入

この例では、複数のMULTISET演算と1つのSETファンクションの起動結果をネストした表の変数answerに代入し、代入するたびにプロシージャprint_nested_tableを使用してanswerを出力します。このプロシージャでは、コレクション・メソッドFIRSTおよびLASTを使用しています。詳細は「コレクション・メソッド」を参照してください。

Live SQL:

この例は、Oracle Live SQL の「ネストした表の変数への SET 演算結果の代入」で表示および実行できます

```
DECLARE
TYPE nested_typ IS TABLE OF NUMBER; nt1 nested_typ := nested_typ(1,2,3); nt2 nested_typ :=
nested_typ(3,2,1); nt3 nested_typ := nested_typ(2,3,1,3); nt4 nested_typ := nested_typ(1,2,4);
answer nested_typ; PROCEDURE print_nested_table (nt nested_typ) IS output VARCHAR2(128);
BEGIN IF nt IS NULL THEN DBMS_OUTPUT.PUT_LINE('Result: null set'); ELSIF nt.COUNT = 0
THEN DBMS_OUTPUT.PUT_LINE('Result: empty set'); ELSE FOR i IN nt.FIRST .. nt.LAST LOOP --
For first to last element output := output || nt(i) || ' '; END LOOP;
DBMS_OUTPUT.PUT_LINE('Result: ' || output); END IF; END print_nested_table; BEGIN
answer := nt1 MULTISET UNION nt4; print_nested_table(answer); answer := nt1 MULTISET
UNION nt3; print_nested_table(answer); answer := nt1 MULTISET UNION DISTINCT nt3;
print_nested_table(answer); answer := nt2 MULTISET INTERSECT nt3;
print_nested_table(answer); answer := nt2 MULTISET INTERSECT DISTINCT nt3;
print_nested_table(answer); answer := SET(nt3); print_nested_table(answer); answer := nt3
MULTISET EXCEPT nt2; print_nested_table(answer); answer := nt3 MULTISET EXCEPT DISTINCT
nt2; print_nested_table(answer); END; /
```

結果:

```
Result: 1 2 3 1 2 4
Result: 1 2 3 2 3 1 3
Result: 1 2 3
Result: 3 2 1
Result: 3 2 1
Result: 2 3 1
```

```
Result: 3
Result: empty set
```

5.8 多次元コレクション

コレクションは1次元のみですが、コレクションを要素に持つコレクションを使用して、多次元コレクションのモデルを作成できます。

例5-13 2次元のVARRAY (VARRAYのVARRAY)

この例のnvaは、2次元のVARRAY(整数のVARRAYのVARRAY)です。

Live SQL:

この例は、Oracle Live SQL の「2 次元の VARRAY(VARRAY の VARRAY)」で表示および実行できます

```
DECLARE
```

TYPE t1 IS VARRAY(10) OF INTEGER; -- varray of integer va t1 := t1(2,3,5); TYPE nt1 IS VARRAY(10) OF t1; -- varray of varray of integer nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va); i INTEGER; va1 t1; BEGIN i := nva(2)(3); DBMS_OUTPUT.PUT_LINE('i = ' || i); nva.EXTEND; nva(5) := t1(56, 32); -- replace inner varray elements nva(4) := t1(45,43,67,43345); -- replace an inner integer element nva(4)(4) := 1; -- replace 43345 with 1 nva(4).EXTEND; -- add element to 4th varray element nva(4)(5) := 89; -- store integer 89 there END; /

結果:

```
i = 73
```

例5-14 ネストした表のネストした表と整数のVARRAYのネストした表

この例のntb1は文字列のネストした表のネストした表で、ntb2は整数のVARRAYのネストした表です。

Live SQL:

この例は、Oracle Live SQL の「ネストした表のネストした表と整数の VARRAY のネストした表」で表示および実行できます

```
DECLARE
```

TYPE tb1 IS TABLE OF VARCHAR2(20); -- nested table of strings vtb1 tb1 := tb1('one', 'three'); TYPE ntb1 IS TABLE OF tb1; -- nested table of nested tables of strings vntb1 ntb1 := ntb1(vtb1); TYPE tv1 IS VARRAY(10) OF INTEGER; -- varray of integers TYPE ntb2 IS TABLE OF tv1; -- nested table of varrays of integers vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3)); BEGIN vntb1.EXTEND; vntb1(2) := vntb1(1); vntb1.DELETE(1); -- delete first element of vntb1 vntb1(2).DELETE(1); -- delete first string from second table in nested table END; /

例5-15 結合配列のネストした表と文字列のVARRAYのネストした表

この例のaa1は連想配列の連想配列で、ntb2は文字列のVARRAYのネストした表です。

Live SQL:

この例は、Oracle Live SQL の「連想配列のネストした表と文字列の VARRAY のネストした表」で表示および実行できます

```
DECLARE
TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER; -- associative arrays v4 tb1; v5 tb1;
TYPE aa1 IS TABLE OF tb1 INDEX BY PLS_INTEGER; -- associative array of v2 aa1; -- associative
arrays TYPE va1 IS VARRAY(10) OF VARCHAR2(20); -- varray of strings v1 va1 := va1('hello',
'world'); TYPE ntb2 IS TABLE OF va1 INDEX BY PLS_INTEGER; -- associative array of varrays v3
ntb2; BEGIN v4(1) := 34; -- populate associative array v4(2) := 46456; v4(456) := 343;
v2(23) := v4; -- populate associative array of associative arrays v3(34) := va1(33, 456, 656,
343); -- populate associative array varrays v2(35) := v5; -- assign empty associative array to
v2(35) v2(35)(2) := 78; END; /
```

5.9 コレクションの比較

たとえば、あるコレクション変数が別のコレクション変数より小さいかどうかを判別するには、そのコンテキストで「より小さい」の意味を定義して、TRUEまたはFALSEを戻すファンクションを記述する必要があります。

連想配列の変数と値NULLを比較したり、連想配列を互いに比較することはできません。

「ネストした表が等しいかどうかの比較」を除き、関係演算子を使用して2つのコレクション変数をそのまま比較はできません。この制限は、暗黙的な比較にも適用されます。たとえば、コレクション変数はDISTINCT、GROUP BYまたはORDER BY句には使用できません。

ここでのトピック

- VARRAYおよびネストした表の変数とNULLの比較
- ネストした表が等しいかどうかの比較
- ネストした表のSQL MULTISET条件による比較

関連項目:

- 表2-5
- 機能の記述の詳細は、「PL/SQLサブプログラム」を参照してください

5.9.1 VARRAYおよびネストした表の変数とNULLの比較

NULL値と比較するときは、IS[NOT] NULL演算子を使用します。

VARRAYおよびネストした表の変数は、「IS [NOT] NULL演算子」を使用して値NULLと比較できますが、関係演算子の等価演算子(=)および不等価演算子(<>、!=、~=または^=)を使用して比較できません。

例5-16 VARRAYおよびネストした表の変数とNULLの比較

この例では、VARRAYおよびネストした表の変数とNULLを正しく比較しています。

Live SQL:

この例は、Oracle Live SQL の「VARRAY およびネストした表の変数と NULL の比較」で表示および実行できます

```
DECLARE
```

TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type team Foursome; -- varray variable TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type names Roster := Roster('Adams', 'Patel'); -- nested table variable BEGIN IF team IS NULL THEN DBMS_OUTPUT.PUT_LINE('team IS NULL'); ELSE DBMS_OUTPUT.PUT_LINE('team IS NOT NULL'); END IF; IF names IS NOT NULL THEN DBMS_OUTPUT.PUT_LINE('names IS NOT NULL'); ELSE DBMS_OUTPUT.PUT_LINE('names IS NULL'); END IF; END; /

結果:

```
team IS NULL
names IS NOT NULL
```

5.9.2 ネストした表が等しいかどうかの比較

2つのネストした表の変数は、同じ要素のセット(順不同)を含む場合のみ等価です。

2つのネストした表の変数が同じネストした表型を持ち、ネストした表型にレコード型の要素が含まれていない場合は、関係演算子の等価演算子(=)および不等価演算子(<>、!=、~=、 ^=)を使用して、2つの変数が等しいかどうかを比較できます。

関連項目:

レコードの比較

例5-17 ネストした表が等しいかどうかの比較

この例では、ネストした表の変数が等しいか等しくないかを関係演算子で比較しています。

Live SQL:

この例は、Oracle Live SQL の「ネストした表が等しいかどうかの比較」で表示および実行できます

```
DECLARE
```

TYPE dnames_tab IS TABLE OF VARCHAR2(30); -- element type is not record type dept_names1 dnames_tab := dnames_tab('Shipping','Sales','Finance','Payroll'); dept_names2 dnames_tab := dnames_tab('Sales','Finance','Shipping','Payroll'); dept_names3 dnames_tab := dnames_tab('Sales','Finance','Payroll'); BEGIN IF dept_names1 = dept_names2 THEN DBMS_OUTPUT.PUT_LINE('dept_names1 = dept_names2'); END IF; IF dept_names2 != dept_names3 THEN DBMS_OUTPUT.PUT_LINE('dept_names2 != dept_names3'); END IF; END; /

結果:

```
dept_names1 = dept_names2
dept_names2 != dept_names3
```

5.9.3 ネストした表のSQL MULTISET条件による比較

ネストした表の変数を比較し、そのプロパティの一部をSQL MULTISET条件でテストできます。

関連項目:

- MULTISET条件の詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- CARDINALITY構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- SET構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください

例5-18 ネストした表のSQL MULTISET条件による比較

この例では、SQL MULTISET条件と、ネストした表の変数引数CARDINALITYおよびSETを使用する2つのSQLファンクションを使用します。

Live SQL:

この例は、Oracle Live SQL の「ネストした表の SQL MULTISET 条件による比較」で表示および実行できます

```
DECLARE
```

TYPE nested_typ IS TABLE OF NUMBER; nt1 nested_typ := nested_typ(1,2,3); nt2 nested_typ := nested_typ(3,2,1); nt3 nested_typ := nested_typ(2,3,1,3); nt4 nested_typ := nested_typ(1,2,4); PROCEDURE testify (truth BOOLEAN := NULL, quantity NUMBER := NULL) IS BEGIN IF truth IS NOT NULL THEN DBMS_OUTPUT.PUT_LINE (CASE truth WHEN TRUE THEN 'True' WHEN FALSE THEN 'False' END); END IF; IF quantity IS NOT NULL THEN DBMS_OUTPUT.PUT_LINE(quantity);

END IF; END; BEGIN testify(truth => (nt1 IN (nt2,nt3,nt4))); -- condition testify(truth => (nt1 SUBMULTISET OF nt3)); -- condition testify(truth => (nt1 NOT SUBMULTISET OF nt4)); -- condition testify(truth => (4 MEMBER OF nt1)); -- condition testify(truth => (nt3 IS A SET)); -- condition testify(truth => (nt3 IS NOT A SET)); -- condition testify(truth => (nt1 IS EMPTY)); -- condition testify(quantity => (CARDINALITY(nt3))); -- function testify(quantity => (CARDINALITY(SET(nt3)))); -- 2 functions END; /

結果:

```
True
True
True
False
False
True
False
4
3
```

5.10 コレクション・メソッド

コレクション・メソッドはPL/SQLサブプログラムであり、コレクションに関する情報を戻すファンクションまたはコレクションに対する操作を実行するプロシージャのいずれかです。コレクション・メソッドを使用すると、コレクションの使用およびアプリケーションのメンテナンスが簡単になります。

表5-2に、コレクション・メソッドの概要を示します。

ノート:

NULL のコレクションで使用する場合に、事前定義の例外 COLLECTION_IS_NULL が発生しないコレクション・メソッドは EXISTS のみです。

表5-2 コレクション・メソッド

メソッド	型	説明
DELETE	プロシージャ	コレクションから要素を削除します。
TRIM	プロシージャ	VARRAY またはネストした表の末尾から要素を削除します。
EXTEND	プロシージャ	VARRAY またはネストした表の末尾に要素を追加します。
EXISTS	ファンクション	VARRAY またはネストした表の指定された要素が存在する場合のみ TRUE を戻します。

メソッド	型	説明
FIRST	ファンクション	コレクションの最初の索引を戻します。
LAST	ファンクション	コレクションの最後の索引を戻します。
COUNT	ファンクション	コレクション内の要素の数を戻します。
LIMIT	ファンクション	コレクションに格納できる要素の最大数を戻します。
PRIOR	ファンクション	指定された索引の前の索引を戻します。
NEXT	ファンクション	指定された索引の後の索引を戻します。

コレクション・メソッドを起動する基本構文は、次のとおりです。

```
collection_name.method
```

構文の詳細は、「コレクション・メソッドの起動」を参照してください。

コレクション・メソッドの起動は、その型(ファンクションまたはプロシージャ)のPL/SQLサブプログラムの起動を指定できるすべての場所(SQL文を除く)で使用できます。(PL/SQLサブプログラムの概要は、「PL/SQLサブプログラム」を参照してください。)

サブプログラム内で、コレクション・パラメータは引数のプロパティがバインドされていることを前提にしています。コレクション・メソッドは、そのようなパラメータに適用できます。VARRAYパラメータの場合、パラメータ・モードに関係なく、LIMITの値は常にパラメータの型定義から導出されます。

ここでのトピック

- DELETEコレクション・メソッド
- TRIMコレクション・メソッド
- EXTENDコレクション・メソッド
- EXISTSコレクション・メソッド
- FIRSTおよびLASTコレクション・メソッド
- COUNTコレクション・メソッド
- LIMITコレクション・メソッド
- PRIORおよびNEXTコレクション・メソッド

5.10.1 DELETEコレクション・メソッド

DELETEは、コレクションから要素を削除するプロシージャです。

このメソッドには次の形式があります。

- DELETEはすべての型のコレクションからすべての要素を削除します。

  この操作を実行すると、削除された要素に割り当てられていたメモリーは即座に解放されます。

- 連想配列またはネストされた表(VARRAYではない)から削除する場合は次のとおりです。

  - DELETE(n)は、nの索引を持つ要素が存在する場合にその要素を削除します。それ以外の場合は何も実行しません。

  - DELETE(m,n)は、mとnの両方が存在し、m <= nの場合に、mからnの範囲の索引を持つすべての要素を削除します。それ以外の場合は何も実行しません。

  これら2つのDELETE形式を使用する場合、PL/SQLは削除された要素のプレースホルダを保持します。したがって、削除された要素はコレクションの内部サイズに含まれるため、削除された要素に有効な値を代入してリストアすることができます。

例5-19 ネストした表でのDELETEメソッド

この例では、ネストした表の変数を宣言し、6つの要素を使用して初期化した後で、2番目の要素を削除してからリストアし、次に、要素の範囲を削除し、削除した要素の1つをリストアしてから、すべての要素を削除します。リストアされた要素は、それに対応する削除された要素と同じメモリー量を占有します。プロシージャprint_ntは、初期化の後および各DELETE操作の後に、ネストした表の変数を出力します。型nt_typeおよびプロシージャprint_ntは、例5-6で定義しています。

```
DECLARE
  nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
  print_nt(nt);

  nt.DELETE(2);      -- Delete second element
  print_nt(nt);

  nt(2) := 2222;     -- Restore second element
  print_nt(nt);

  nt.DELETE(2, 4);   -- Delete range of elements
  print_nt(nt);

  nt(3) := 3333;     -- Restore third element
  print_nt(nt);

  nt.DELETE;         -- Delete all elements
  print_nt(nt);
END;
/
```

結果:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 33
```

```
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(2) = 2222
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 3333
nt.(5) = 55
nt.(6) = 66
---
nt is empty
---
```

例5-20 文字列で索引付けされている連想配列でのDELETEメソッド

この例では、文字列で索引付けされている連想配列に要素を移入してからすべての要素を削除し、これにより、要素に割り当てられていたメモリーは解放されます。次に、削除した要素を置き換えます(つまり、削除した要素と同じ索引を持つ新しい要素を追加します)。新たに置き換えた要素は、それに対応する削除された要素と同じメモリー量を占有しません。最後に、1つの要素を削除してから、要素の範囲を削除します。プロシージャprint_aa_strは、各操作の効果を表示します。

```
DECLARE
  TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
  aa_str  aa_type_str;

  PROCEDURE print_aa_str IS
    i  VARCHAR2(10);
  BEGIN
    i := aa_str.FIRST;

    IF i IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('aa_str is empty');
    ELSE
      WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT('aa_str.(' || i || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa_str(i)), 'NULL'));
        i := aa_str.NEXT(i);
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END print_aa_str;

BEGIN
  aa_str('M') := 13;
  aa_str('Z') := 26;
  aa_str('C') := 3;
  print_aa_str;
```

```
  aa_str.DELETE;  -- Delete all elements
  print_aa_str;

  aa_str('M') := 13;   -- Replace deleted element with same value
  aa_str('Z') := 260;  -- Replace deleted element with new value
  aa_str('C') := 30;   -- Replace deleted element with new value
  aa_str('W') := 23;   -- Add new element
  aa_str('J') := 10;   -- Add new element
  aa_str('N') := 14;   -- Add new element
  aa_str('P') := 16;   -- Add new element
  aa_str('W') := 23;   -- Add new element
  aa_str('J') := 10;   -- Add new element
  print_aa_str;

  aa_str.DELETE('C');      -- Delete one element
  print_aa_str;

  aa_str.DELETE('N','W');  -- Delete range of elements
  print_aa_str;

  aa_str.DELETE('Z','M');  -- Does nothing
  print_aa_str;
END;
/
```

結果:

```
aa_str.(C) = 3
aa_str.(M) = 13
aa_str.(Z) = 26
---
aa_str is empty
---
aa_str.(C) = 30
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(N) = 14
aa_str.(P) = 16
aa_str.(W) = 23
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(N) = 14
aa_str.(P) = 16
aa_str.(W) = 23
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(Z) = 260
---
```

5.10.2 TRIMコレクション・メソッド

TRIMは、VARRAYまたはネストした表の末尾から要素を削除するプロシージャです。

このメソッドには次の形式があります。

- TRIMは、コレクションに1つ以上の要素が含まれる場合に、末尾から1つの要素を削除します。それ以外の場合は、事前定義の例外SUBSCRIPT_BEYOND_COUNTを呼び出します。
- TRIM(n)は、コレクションの末尾にn個以上の要素が含まれる場合に、末尾からn個の要素を削除します。それ以外の場合は、事前定義の例外SUBSCRIPT_BEYOND_COUNTを呼び出します。

TRIMは、コレクションの内部サイズを操作します。つまり、DELETEで要素が削除されてもその要素のプレースホルダが保持されている場合、TRIMはその要素が存在するものとみなします。したがって、削除された要素をTRIMで削除することができます。

PL/SQLは切り捨てられた要素のプレースホルダを保持しません。したがって、切り捨てられた要素はコレクションの内部サイズに含まれないため、切り捨てられた要素に有効な値を代入して要素をリストアすることはできません。

注意:

TRIM と DELETE の間の相互作用には依存しないでください。ネストした表は、固定サイズの配列のように扱う(DELETE のみを使用する)か、またはスタックのように扱う(TRIM と EXTEND のみを使用する)ようにしてください。

例5-21 ネストした表でのTRIMメソッド

この例では、ネストした表の変数を宣言し、6つの要素で初期化した後で、末尾の要素を切り捨て、4番目の要素を削除してから、末尾の2つの要素を切り捨てます(切り捨てる要素の1つは、削除した4番目の要素です)。プロシージャprint_ntは、初期化の後とTRIM操作およびDELETE操作の後に、ネストした表の変数を出力します。型nt_typeおよびプロシージャprint_ntは、例5-6で定義しています。

```
DECLARE
  nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
  print_nt(nt);
  nt.TRIM;       -- Trim last element
  print_nt(nt);
  nt.DELETE(4);  -- Delete fourth element
  print_nt(nt);
  nt.TRIM(2);    -- Trim last two elements
  print_nt(nt);
END;
/
```

結果:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
```

```
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(5) = 55
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
---
```

5.10.3 EXTENDコレクション・メソッド

EXTENDは、VARRAYまたはネストした表の末尾に要素を追加するプロシージャです。

コレクションは空でもかまいませんが、NULLは許可されません。(コレクションを空にしたりNULLのコレクションに要素を追加するには、コンストラクタを使用します。詳細は、「コレクションのコンストラクタ」を参照してください。)

EXTENDメソッドには次の形式があります。

- EXTENDは、コレクションに1つのNULL要素を追加します。
- EXTEND(n)は、コレクションにn個のNULL要素を追加します。
- EXTEND(n,i)は、コレクションにi番目の要素のコピーをn個追加します。

ノート:

NOT NULL 制約が含まれている要素を持つコレクションに使用できる形式は、EXTEND(n,i)のみです。

EXTENDは、コレクションの内部サイズを操作します。つまり、DELETEで要素が削除されてもその要素のプレースホルダが保持されている場合、EXTENDはその要素が存在するものとみなします。

例5-22 ネストした表でのEXTENDメソッド

この例では、ネストした表の変数を宣言し、3つの要素を使用して初期化した後で、最初の要素のコピーを2つ追加し、次に、5番目(最後)の要素を削除してから、NULL要素を1つ追加します。EXTENDでは、削除された5番目の要素が存在するものとみなされるため、追加されたNULL要素は6番目の要素になります。プロシージャprint_ntは、初期化の後とEXTEND操作およびDELETE操作の後に、ネストした表の変数を出力します。型nt_typeおよびプロシージャprint_ntは、例5-6で定義しています。

```
DECLARE
  nt nt_type := nt_type(11, 22, 33);
BEGIN
  print_nt(nt);

  nt.EXTEND(2,1);  -- Append two copies of first element
  print_nt(nt);
```

```
  nt.DELETE(5);    -- Delete fifth element
  print_nt(nt);

  nt.EXTEND;       -- Append one null element
  print_nt(nt);
END;
/
```

結果:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
nt.(5) = 11
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
nt.(6) = NULL
---
```

5.10.4 EXISTSコレクション・メソッド

EXISTSは、VARRAYまたはネストした表の指定された要素が存在するかどうかを表示するファンクションです。

EXISTS(n)は、コレクションにn番目の要素が存在する場合にTRUEを戻し、それ以外の場合はFALSEを戻します。nが範囲外の場合、EXISTSは事前定義の例外SUBSCRIPT_OUTSIDE_LIMITを呼び出さずに、FALSEを戻します。

削除された要素の場合、DELETEによりその要素のプレースホルダが保持されていても、EXISTS(n)はFALSEを戻します。

例5-23 ネストした表でのEXISTSメソッド

この例では、ネストした表を4つの要素で初期化した後、2番目の要素を削除して、1番目から6番目の要素の値またはステータスのいずれかを出力します。

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(1,3,5,7);
BEGIN
  n.DELETE(2); -- Delete second element

  FOR i IN 1..6 LOOP
    IF n.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
```

```
    ELSE
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
    END IF;
  END LOOP;
END;
/
```

結果:

```
n(1) = 1
n(2) does not exist
n(3) = 5
n(4) = 7
n(5) does not exist
n(6) does not exist
```

5.10.5 FIRSTおよびLASTコレクション・メソッド

FIRSTおよびLASTはファンクションです。

1つ以上の要素を持つコレクションの場合、FIRSTおよびLASTはそれぞれ最初と最後の要素の索引を戻します(削除された要素のプレースホルダがDELETEにより保持されていても、それらの要素は無視されます)。コレクションの要素が1つのみの場合、FIRSTおよびLASTは同じ索引を戻します。コレクションが空の場合、FIRSTとLASTはNULLを戻します。

ここでのトピック

- 結合配列に対するFIRSTおよびLASTメソッド
- VARRAYに対するFIRSTおよびLASTメソッド
- ネストした表に対するFIRSTおよびLASTメソッド

5.10.5.1 結合配列に対するFIRSTおよびLASTメソッド

PLS_INTEGERで索引付けされている連想配列の場合、最初の要素および最後の要素は、それぞれ最小および最大の索引を持つ要素です。文字列で索引付けされている連想配列の場合、最初の要素および最後の要素は、それぞれ最小および最大のキー値を持つ要素です。

キー値の順序はソートされています(詳細は、「文字列で索引付けされている連想配列に影響を与えるNLSパラメータ値」を参照してください)。

例5-24 PLS_INTEGERで索引付けされている連想配列のFIRSTとLASTの値

この例では、PLS_INTEGERで索引付けされている連想配列のFIRSTとLASTの値を表示してから最初と最後の要素を削除し、再度、FIRSTとLASTの値を表示します。

```
DECLARE
  TYPE aa_type_int IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa_int  aa_type_int;

  PROCEDURE print_first_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_int.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_int.LAST);
```

```
  END print_first_and_last;

BEGIN
  aa_int(1) := 3;
  aa_int(2) := 6;
  aa_int(3) := 9;
  aa_int(4) := 12;

  DBMS_OUTPUT.PUT_LINE('Before deletions:');
  print_first_and_last;

  aa_int.DELETE(1);
  aa_int.DELETE(4);

  DBMS_OUTPUT.PUT_LINE('After deletions:');
  print_first_and_last;
END;
/
```

結果:

```
Before deletions:
FIRST = 1
LAST = 4
After deletions:
FIRST = 2
LAST = 3
```

例5-25 文字列で索引付けされている連想配列のFIRSTとLASTの値

この例では、文字列で索引付けされている連想配列のFIRSTとLASTの値を表示してから最初と最後の要素を削除し、再度、FIRSTとLASTの値を表示します。

```
DECLARE
  TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
  aa_str  aa_type_str;

  PROCEDURE print_first_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_str.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_str.LAST);
  END print_first_and_last;

BEGIN
  aa_str('Z') := 26;
  aa_str('A') := 1;
  aa_str('K') := 11;
  aa_str('R') := 18;

  DBMS_OUTPUT.PUT_LINE('Before deletions:');
  print_first_and_last;

  aa_str.DELETE('A');
  aa_str.DELETE('Z');

  DBMS_OUTPUT.PUT_LINE('After deletions:');
  print_first_and_last;
```

```
END;
/
```

結果:

```
Before deletions:
FIRST = A
LAST = Z
After deletions:
FIRST = K
LAST = R
```

5.10.5.2 VARRAYに対するFIRSTおよびLASTメソッド

空ではないVARRAYの場合、FIRSTは常に1を戻します。すべてのVARRAYで、LASTは常にCOUNTに等しくなります。

例5-26 FOR LOOP内のFIRSTおよびLASTによるVARRAYの出力

この例では、team.FIRSTおよびteam.LASTの境界を持つFOR LOOP文を使用して、VARRAYのteamを出力します。VARRAYは常に密であるため、ループの内側のteam(i)は常に存在します。

```
DECLARE
  TYPE team_type IS VARRAY(4) OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading VARCHAR2)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Does not exist');
    ELSIF team.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
      FOR i IN team.FIRST..team.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(i || '. ' || team(i));
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END;

BEGIN
  print_team('Team Status:');

  team := team_type();  -- Team is funded, but nobody is on it.
  print_team('Team Status:');

  team := team_type('John', 'Mary');  -- Put 2 members on team.
  print_team('Initial Team:');

  team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Change team.
  print_team('New Team:');
END;
/
```

結果:

```
Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. John
2. Mary
---
New Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---
```

関連項目

- 例5-28

5.10.5.3 ネストした表に対するFIRSTおよびLASTメソッド

ネストした表の場合、LASTがCOUNTより大きい場合は、中から要素を削除しないかぎり、LASTはCOUNTに等しくなります。

例5-27 FOR LOOP内のFIRSTおよびLASTによるネストした表の出力

この例では、team.FIRSTおよびteam.LASTの境界を持つFOR LOOP文を使用して、ネストした表のteamを出力します。ネストした表は疎の場合があるため、team.EXISTS(i)がTRUEの場合のみ、FOR LOOP文はteam(i)を出力します。

```
DECLARE
  TYPE team_type IS TABLE OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Does not exist');
    ELSIF team.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
      FOR i IN team.FIRST..team.LAST LOOP
        DBMS_OUTPUT.PUT(i || '. ');
        IF team.EXISTS(i) THEN
          DBMS_OUTPUT.PUT_LINE(team(i));
        ELSE
          DBMS_OUTPUT.PUT_LINE('(to be hired)');
        END IF;
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END;
```

```
BEGIN
  print_team('Team Status:');

  team := team_type();  -- Team is funded, but nobody is on it.
  print_team('Team Status:');

  team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Add members.
  print_team('Initial Team:');

  team.DELETE(2,3);  -- Remove 2nd and 3rd members.
  print_team('Current Team:');
END;
/
```

結果:

```
Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---
Current Team:
1. Arun
2. (to be hired)
3. (to be hired)
4. Mae
---
```

関連項目

- 例5-29

5.10.6 COUNTコレクション・メソッド

COUNTは、コレクション内の要素の数を戻すファンクションです(削除された要素がDELETEにより保持されていても、それらは無視されます)。

ここでのトピック

- VARRAYに対するCOUNTメソッド
- ネストした表に対するCOUNTメソッド

5.10.6.1 VARRAYに対するCOUNTメソッド

VARRAYの場合、COUNTは常にLASTと同じです。VARRAYのサイズを(EXTENDまたはTRIMメソッドを使用して)増減させると、COUNTの値が変更されます。

例5-28 VARRAYのCOUNTとLASTの値

この例では、4つの要素を使用して初期化し、EXTEND(3)に続いてTRIM(5)を実行してから、VARRAYのCOUNTとLASTの値を表示します。

```
DECLARE
  TYPE NumList IS VARRAY(10) OF INTEGER;
  n NumList := NumList(1,3,5,7);

  PROCEDURE print_count_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
    DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
  END  print_count_and_last;

BEGIN
  print_count_and_last;

  n.EXTEND(3);
  print_count_and_last;

  n.TRIM(5);
  print_count_and_last;
END;
/
```

結果:

```
n.COUNT = 4, n.LAST = 4
n.COUNT = 7, n.LAST = 7
n.COUNT = 2, n.LAST = 2
```

5.10.6.2 ネストした表に対するCOUNTメソッド

ネストした表の場合、COUNTはLASTと同じです。ただし、ネストした表の中から要素を削除すると、COUNTはLASTより小さくなります。

例5-29 ネストした表のCOUNTとLASTの値

この例では、4つの要素を使用して初期化し、3番目の要素を削除した後で末尾にNULL要素を2つ追加してから、ネストした表のCOUNTとLASTの値を表示します。最後に、1番目から8番目の要素のステータスを出力します。

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(1,3,5,7);

  PROCEDURE print_count_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
    DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
  END  print_count_and_last;

BEGIN
  print_count_and_last;
```

```
  n.DELETE(3);  -- Delete third element
  print_count_and_last;

  n.EXTEND(2);  -- Add two null elements to end
  print_count_and_last;

  FOR i IN 1..8 LOOP
    IF n.EXISTS(i) THEN
      IF n(i) IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
      ELSE
        DBMS_OUTPUT.PUT_LINE('n(' || i || ') = NULL');
      END IF;
    ELSE
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
    END IF;
  END LOOP;
END;
/
```

結果:

```
n.COUNT = 4, n.LAST = 4
n.COUNT = 3, n.LAST = 4
n.COUNT = 5, n.LAST = 6
n(1) = 1
n(2) = 3
n(3) does not exist
n(4) = 7
n(5) = NULL
n(6) = NULL
n(7) does not exist
n(8) does not exist
```

5.10.7 LIMITコレクション・メソッド

LIMITは、コレクションに格納可能な要素の最大数を戻すファンクションです。コレクションに要素の最大数がない場合、LIMITはNULLを戻します。最大サイズがあるのはVARRAYのみです。

例5-30 様々なコレクション型のLIMITとCOUNTの値

この例では、4つの要素を持つ連想配列、2つの要素を持つVARRAY、3つの要素を持つネストした表について、LIMITとCOUNTの値を表示します。

```
DECLARE
  TYPE aa_type IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa aa_type;                          -- associative array

  TYPE va_type IS VARRAY(4) OF INTEGER;
  va  va_type := va_type(2,4);   -- varray

  TYPE nt_type IS TABLE OF INTEGER;
  nt  nt_type := nt_type(1,3,5);  -- nested table

BEGIN
  aa(1):=3; aa(2):=6; aa(3):=9; aa(4):= 12;
```

```
  DBMS_OUTPUT.PUT('aa.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.COUNT), 'NULL'));

  DBMS_OUTPUT.PUT('aa.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.LIMIT), 'NULL'));

  DBMS_OUTPUT.PUT('va.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.COUNT), 'NULL'));

  DBMS_OUTPUT.PUT('va.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.LIMIT), 'NULL'));

  DBMS_OUTPUT.PUT('nt.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.COUNT), 'NULL'));

  DBMS_OUTPUT.PUT('nt.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.LIMIT), 'NULL'));
END;
/
```

結果:

```
aa.COUNT = 4
aa.LIMIT = NULL
va.COUNT = 2
va.LIMIT = 4
nt.COUNT = 3
nt.LIMIT = NULL
```

5.10.8 PRIORおよびNEXTコレクション・メソッド

PRIORおよびNEXTは、コレクション内を前後に移動できるファンクションです(削除された要素がDELETEにより保持されていても、それらは無視されます)。このメソッドは、疎コレクション内の移動に便利です。

索引を指定すると次のようになります。

- 先行する要素が存在する場合、PRIORはそのコレクション要素の索引を戻します。存在しない場合、PRIORはNULLを戻します。

  任意のコレクションcの場合、c.PRIOR(c.FIRST)はNULLを戻します。

- 後続の要素が存在する場合、NEXTはそのコレクション要素の索引を戻します。存在しない場合、NEXTはNULLを戻します。

  任意のコレクションcの場合、c.NEXT(c.LAST)はNULLを戻します。

指定された索引が存在していなくてもかまいません。ただし、コレクションcがVARRAYで、索引がc.LIMITより大きい場合は次のようになります。

- c.PRIOR(index)はc.LASTを戻します。
- c.NEXT(index)はNULLを戻します。

たとえば:

```
DECLARE
```

```
  TYPE Arr_Type IS VARRAY(10) OF NUMBER;
  v_Numbers Arr_Type := Arr_Type();
BEGIN
  v_Numbers.EXTEND(4);

  v_Numbers (1) := 10;
  v_Numbers (2) := 20;
  v_Numbers (3) := 30;
  v_Numbers (4) := 40;

  DBMS_OUTPUT.PUT_LINE(NVL(v_Numbers.prior (3400), -1));
  DBMS_OUTPUT.PUT_LINE(NVL(v_Numbers.next (3400), -1));
END;
/
```

結果:

```
4
-1
```

文字列で索引付けされている連想配列の場合、前の索引および次の索引はキー値で決まり、キー値の順序はソートされています(詳細は、「文字列で索引付けされている連想配列に影響を与えるNLSパラメータ値」を参照してください)。例5-1では、FIRST、NEXTおよびWHILE LOOP文を使用して連想配列の要素を出力しました。

例5-31 PRIORおよびNEXTメソッド

この例では、ネストした表を6つの要素で初期化した後、4番目の要素を削除して、1番目から7番目の要素のPRIORとNEXTの値を表示します。4番目と7番目の要素は存在しません。値はNULLですが、2番目の要素は存在します。

```
DECLARE
  TYPE nt_type IS TABLE OF NUMBER;
  nt nt_type := nt_type(18, NULL, 36, 45, 54, 63);

BEGIN
  nt.DELETE(4);
  DBMS_OUTPUT.PUT_LINE('nt(4) was deleted.');

  FOR i IN 1..7 LOOP
    DBMS_OUTPUT.PUT('nt.PRIOR(' || i || ') = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.PRIOR(i)), 'NULL'));
    DBMS_OUTPUT.PUT('nt.NEXT(' || i || ')  = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.NEXT(i)), 'NULL'));
  END LOOP;
END;
/
```

結果:

```
nt(4) was deleted.
nt.PRIOR(1) = NULL
nt.NEXT(1)  = 2
nt.PRIOR(2) = 1
nt.NEXT(2)  = 3
nt.PRIOR(3) = 2
nt.NEXT(3)  = 5
nt.PRIOR(4) = 3
```

```
nt.NEXT(4)  = 5
nt.PRIOR(5) = 3
nt.NEXT(5)  = 6
nt.PRIOR(6) = 5
nt.NEXT(6)  = NULL
nt.PRIOR(7) = 6
nt.NEXT(7)  = NULL
```

例5-32 疎であるネストした表の要素の出力

この例では、疎であるネストした表の要素を、FIRSTとNEXTを使用して最初から最後まで出力し、LASTとPRIORを使用して最後から最初まで出力します。

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  n NumList := NumList(1, 2, NULL, NULL, 5, NULL, 7, 8, 9, NULL);
  idx INTEGER;

BEGIN
  DBMS_OUTPUT.PUT_LINE('First to last:');
  idx := n.FIRST;
  WHILE idx IS NOT NULL LOOP
    DBMS_OUTPUT.PUT('n(' || idx || ') = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
    idx := n.NEXT(idx);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('--------------');

  DBMS_OUTPUT.PUT_LINE('Last to first:');
  idx := n.LAST;
  WHILE idx IS NOT NULL LOOP
    DBMS_OUTPUT.PUT('n(' || idx || ') = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
    idx := n.PRIOR(idx);
  END LOOP;
END;
/
```

結果:

```
First to last:
n(1) = 1
n(2) = 2
n(3) = NULL
n(4) = NULL
n(5) = 5
n(6) = NULL
n(7) = 7
n(8) = 8
n(9) = 9
n(10) = NULL
--------------
Last to first:
n(10) = NULL
n(9) = 9
n(8) = 8
```

```
n(7) = 7
n(6) = NULL
n(5) = 5
n(4) = NULL
n(3) = NULL
n(2) = 2
n(1) = 1
```

5.11 パッケージ仕様部で定義されたコレクション型

パッケージ仕様部で定義されたコレクション型は、同一定義のローカル・コレクション型またはスタンドアロン・コレクション型と互換性がありません。

ノート:

このトピックの例ではパッケージとプロシージャを定義しています。詳細は、「PL/SQL パッケージ」と「PL/SQL サブプログラム」をそれぞれ参照してください。

例5-33 同一定義のパッケージ・コレクション型とローカル・コレクション型

この例では、パッケージ仕様部と無名ブロックで、同じコレクション型NumListを定義します。パッケージでは、NumListのパラメータを取るプロシージャprint_numlistを定義します。また、無名ブロックでは、型pkg.NumList(パッケージで定義した型)の変数n1と、型NumList(このブロックで定義した型)の変数n2を宣言します。無名ブロックは、n1をprint_numlistに渡せますが、n2をprint_numlistに渡すことはできません。

Live SQL:

この例は、Oracle Live SQL の「同一定義のパッケージ・コレクション型とローカル・コレクション型」で表示および実行できます

```
CREATE OR REPLACE PACKAGE pkg AS
```

TYPE NumList IS TABLE OF NUMBER; PROCEDURE print_numlist (nums NumList); END pkg; / CREATE OR REPLACE PACKAGE BODY pkg AS PROCEDURE print_numlist (nums NumList) IS BEGIN FOR i IN nums.FIRST..nums.LAST LOOP DBMS_OUTPUT.PUT_LINE(nums(i)); END LOOP; END; END pkg; / DECLARE TYPE NumList IS TABLE OF NUMBER; -- local type identical to package type n1 pkg.NumList := pkg.NumList(2,4); -- package type n2 NumList := NumList(6,8); -- local type BEGIN pkg.print_numlist(n1); -- succeeds pkg.print_numlist(n2); -- fails END; /

結果:

```
  pkg.print_numlist(n2);  -- fails
  *
ERROR at line 7:
ORA-06550: line 7, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_NUMLIST'
ORA-06550: line 7, column 3:
```

```
PL/SQL: Statement ignored
```

例5-34 同一定義のパッケージ・コレクション型とスタンドアロン・コレクション型

この例では、例5-33のパッケージ仕様部で定義したコレクション型NumListと同じスタンドアロン・コレクション型NumListを定義します。また、無名ブロックでは、型pkg.NumList(パッケージで定義した型)の変数n1と、スタンドアロン型NumListのn2を宣言します。無名ブロックは、n1をprint_numlistに渡せますが、n2をprint_numlistに渡すことはできません。

Live SQL:

この例は、Oracle Live SQL の「同一定義のパッケージ・コレクション型とスタンドアロン・コレクション型」で表示および実行できます

```
CREATE OR REPLACE TYPE NumList IS TABLE OF NUMBER;
```

-- standalone collection type identical to package type / DECLARE n1 pkg.NumList := pkg.NumList(2,4); -- package type n2 NumList := NumList(6,8); -- standalone type BEGIN pkg.print_numlist(n1); -- succeeds pkg.print_numlist(n2); -- fails END; /

結果:

```
  pkg.print_numlist(n2);  -- fails
  *
ERROR at line 7:
ORA-06550: line 7, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_NUMLIST'
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```

5.12 レコード変数

次のいずれかの方法でレコード変数を作成できます。

- RECORD型を定義し、次にその型の変数を宣言します。
- データベースの表またはビュー内の行の全体または一部を表すレコード変数を宣言するには、%ROWTYPEを使用します。
- 事前に宣言されているレコード変数と同じ型のレコード変数を宣言するには、%TYPEを使用します。

(構文およびセマンティクスの詳細は、「レコード変数の宣言」を参照してください。)

ここでのトピック

- レコード変数の初期値
- レコード定数の宣言
- RECORD型
- %ROWTYPE属性を使用した項目の宣言

5.12.1 レコード変数の初期値

RECORD型のレコード変数の場合、型を定義するときに別の初期値を指定しないかぎり、各フィールドの初期値はNULLになります。

%ROWTYPEまたは%TYPEを使用して宣言したレコード変数の場合、各フィールドの初期値はNULLになります。この変数は、参照先項目の初期値を継承しません。

5.12.2 レコード定数の宣言

レコード定数を宣言する場合、初期値をレコードに移入するファンクションを作成して、定数宣言でそのファンクションを起動する必要があります。

例5-35 レコード定数の宣言

この例では、初期値をレコードに移入するファンクションを作成して、定数宣言でそのファンクションを起動します。

Live SQL:

この例は、Oracle Live SQL の「レコード定数の宣言」で表示および実行できます

```
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
```

TYPE My_Rec IS RECORD (a NUMBER, b NUMBER); FUNCTION Init_My_Rec RETURN My_Rec; END My_Types; / CREATE OR REPLACE PACKAGE BODY My_Types IS FUNCTION Init_My_Rec RETURN My_Rec IS Rec My_Rec; BEGIN Rec.a := 0; Rec.b := 1; RETURN Rec; END Init_My_Rec; END My_Types; / DECLARE r CONSTANT My_Types.My_Rec := My_Types.Init_My_Rec(); BEGIN DBMS_OUTPUT.PUT_LINE('r.a = ' || r.a); DBMS_OUTPUT.PUT_LINE('r.b = ' || r.b); END; /

結果:

```
r.a = 0
r.b = 1

PL/SQL procedure successfully completed.
```

5.12.3 RECORD型

PL/SQLブロック内に定義されるRECORD型はローカル型です。ブロック内でのみ使用可能であり、スタンドアロン・サブプログラムまたはパッケージ・サブプログラム内にブロックがある場合にのみ、データベースに格納されます。

パッケージ仕様部に定義されるRECORD型はパブリック項目です。パッケージ名(package_name.type_name)で修飾することで、パッケージの外から参照できます。パブリック項目は、DROP PACKAGE文を使用してパッケージを削除するまで、データベースに格納されます。

スキーマ・レベルではRECORD型を宣言できません。したがって、RECORD型はADT属性のデータ型にできません。

RECORD型を定義するには、型の名前とフィールドを指定します。フィールドを定義するには、フィールド名とデータ型を指定します。

デフォルトでは、フィールドの初期値はNULLです。フィールドにNOT NULL制約を指定できますが、指定する場合は、初期値にNULL以外の値を指定する必要があります。NOT NULL制約がない場合、NULL以外の初期値はオプションです。

パッケージ仕様部で定義されたRECORD型は、同一定義のローカルRECORD型と互換性がありません。

関連項目:

- PL/SQLパッケージ
- PL/SQLサブプログラム
- ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラム
- 例5-39, ""

例5-36 RECORD型の定義および変数の宣言

この例では、DeptRecTypという名前のRECORD型を定義し、各フィールドに初期値を指定します。その後、dept_recという名前でその型の変数を宣言し、変数のフィールドを出力します。

Live SQL:

この例は、Oracle Live SQL の「RECORD 型の定義および変数の宣言」で表示および実行できます

```
DECLARE
TYPE DeptRecTyp IS RECORD ( dept_id NUMBER(4) NOT NULL := 10, dept_name VARCHAR2(30) NOT NULL := 'Administration', mgr_id NUMBER(6) := 200, loc_id NUMBER(4) := 1700 ); dept_rec DeptRecTyp; BEGIN DBMS_OUTPUT.PUT_LINE('dept_id: ' || dept_rec.dept_id); DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.dept_name); DBMS_OUTPUT.PUT_LINE('mgr_id: ' || dept_rec.mgr_id); DBMS_OUTPUT.PUT_LINE('loc_id: ' || dept_rec.loc_id); END; /
```

結果:

```
dept_id:   10
dept_name: Administration
mgr_id:    200
loc_id:    1700
```

例5-37 RECORDフィールドを持つRECORD型(ネストされたレコード)

この例では、name_recおよびcontactという2つのRECORD型を定義します。型contactには型name_recのフィールドを含めます。

Live SQL:

この例は、Oracle Live SQL の「RECORD フィールドを持つ RECORD 型(ネストされたレコード)」で表示お

よび実行できます

```
DECLARE
```

TYPE name_rec IS RECORD (first employees.first_name%TYPE, last employees.last_name%TYPE); TYPE contact IS RECORD (name name_rec, -- nested record phone employees.phone_number%TYPE); friend contact; BEGIN friend.name.first := 'John'; friend.name.last := 'Smith'; friend.phone := '1-650-555-1234'; DBMS_OUTPUT.PUT_LINE (friend.name.first || ' ' || friend.name.last || ', ' || friend.phone); END; /

結果:

```
John Smith, 1-650-555-1234
```

例5-38 VARRAYフィールド持つRECORD型

この例では、full_nameというVARRAY型と、contactというRECORD型を定義します。型contactには型full_nameのフィールドを含めます。

Live SQL:

この例は、Oracle Live SQL の「VARRAY フィールド持つ RECORD 型」で表示および実行できます

```
DECLARE
```

TYPE full_name IS VARRAY(2) OF VARCHAR2(20); TYPE contact IS RECORD (name full_name := full_name('John', 'Smith'), -- varray field phone employees.phone_number%TYPE); friend contact; BEGIN friend.phone := '1-650-555-1234'; DBMS_OUTPUT.PUT_LINE (friend.name(1) || ' ' || friend.name(2) || ', ' || friend.phone); END; /

結果:

```
John Smith, 1-650-555-1234
```

例5-39 同一定義のパッケージRECORD型とローカルRECORD型

この例では、パッケージpkgと無名ブロックで、同じRECORD型rec_typeを定義します。パッケージでは、rec_typeのパラメータを取るプロシージャprint_rec_typeを定義します。無名ブロックでは、パッケージ型(pkg.rec_type)の変数r1と、ローカル型(rec_type)の変数r2を宣言します。無名ブロックは、r1をprint_rec_typeに渡せますが、r2をprint_rec_typeに渡すことはできません。

Live SQL:

この例は、Oracle Live SQL の「同一定義のパッケージ RECORD 型とローカル RECORD 型」で表示および実行できます

```
CREATE OR REPLACE PACKAGE pkg AS
```

TYPE rec_type IS RECORD (-- package RECORD type f1 INTEGER, f2 VARCHAR2(4));

PROCEDURE print_rec_type (rec rec_type); END pkg; / CREATE OR REPLACE PACKAGE BODY pkg AS PROCEDURE print_rec_type (rec rec_type) IS BEGIN DBMS_OUTPUT.PUT_LINE(rec.f1); DBMS_OUTPUT.PUT_LINE(rec.f2); END; END pkg; / DECLARE TYPE rec_type IS RECORD (-- local RECORD type f1 INTEGER, f2 VARCHAR2(4)); r1 pkg.rec_type; -- package type r2 rec_type; -- local type BEGIN r1.f1 := 10; r1.f2 := 'abcd'; r2.f1 := 25; r2.f2 := 'wxyz'; pkg.print_rec_type(r1); -- succeeds pkg.print_rec_type(r2); -- fails END; /

結果:

```
  pkg.print_rec_type(r2);  -- fails
  *
ERROR at line 14:
ORA-06550: line 14, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_REC_TYPE'
```

5.12.4 %ROWTYPE属性を使用した項目の宣言

%ROWTYPE属性を使用すると、データベースの表またはビュー内の行の全体または一部を表すレコード変数を宣言できます。

構文およびセマンティクスの詳細は、「%ROWTYPE属性」を参照してください。

ここでのトピック

- 常に行の全体を表すレコード変数の宣言
- 行の一部を表すことができるレコード変数の宣言
- %ROWTYPE属性と仮想列
- %ROWTYPE属性と非表示列

5.12.4.1 常に行の全体を表すレコード変数の宣言

常にデータベースの表またはビュー内の行の全体を表すレコード変数を宣言するには、次の構文を使用します。

```
variable_name table_or_view_name%ROWTYPE;
```

このレコードは、表またはビューのすべての列に対して、同じ名前とデータ型のフィールドを持ちます。

関連項目:

%ROWTYPEの詳細は、「%ROWTYPE属性」を参照してください

例5-40 データベース表の行全体を表す%ROWTYPE変数

この例では、表departmentsの行を表すレコード変数を宣言し、フィールドに値を代入してからフィールドを出力します。この例を例5-36と比較してください。

Live SQL:

この例は、Oracle Live SQL の「データベース表の行全体を表す%ROWTYPE 変数」で表示および実行できます

```
DECLARE
```

dept_rec departments%ROWTYPE; BEGIN -- Assign values to fields: dept_rec.department_id := 10; dept_rec.department_name := 'Administration'; dept_rec.manager_id := 200; dept_rec.location_id := 1700; -- Print fields: DBMS_OUTPUT.PUT_LINE('dept_id: ' || dept_rec.department_id); DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.department_name); DBMS_OUTPUT.PUT_LINE('mgr_id: ' || dept_rec.manager_id); DBMS_OUTPUT.PUT_LINE('loc_id: ' || dept_rec.location_id); END; /

結果:

```
dept_id:   10
dept_name: Administration
mgr_id:    200
loc_id:    1700
```

例5-41 初期値または制約を継承しない%ROWTYPE変数

この例では、2つの列を持つ表を作成し、それぞれの列に初期値とNOT NULL制約を指定します。次に、この表の行を表すレコード変数を宣言してからフィールドを出力し、初期値またはNOT NULL制約が変数に継承されなかったことを示します。

Live SQL:

この例は、Oracle Live SQL の「初期値または制約を継承しない%ROWTYPE 変数」で表示および実行できます

```
DROP TABLE t1;
```

CREATE TABLE t1 (c1 INTEGER DEFAULT 0 NOT NULL, c2 INTEGER DEFAULT 1 NOT NULL); DECLARE t1_row t1%ROWTYPE; BEGIN DBMS_OUTPUT.PUT('t1.c1 = '); DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c1), 'NULL')); DBMS_OUTPUT.PUT('t1.c2 = '); print(t1_row.c2); DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c2), 'NULL')); END; /

結果:

```
t1.c1 = NULL
t1.c2 = NULL
```

5.12.4.2 行の一部を表すことができるレコード変数の宣言

データベースの表またはビュー内の行の一部を表すことができるレコード変数を宣言するには、次の構文を使用します。

```
variable_name cursor%ROWTYPE;
```

カーソルは問合せに関連付けられています。問合せにより選択されるすべての列に対して、レコード変数の中には、対応する型

互換のフィールドが存在している必要があります。問合せにより表またはビュー内のすべての列が選択される場合、変数は行の全体を表しますが、それ以外の場合は、行の一部を表します。カーソルは、明示カーソルまたは強いカーソル変数のいずれかである必要があります。

関連項目:

- 構文の詳細は、「FETCH文」を参照してください
- カーソルの詳細は、「カーソルの概要」を参照してください
- 明示カーソルの詳細は、「明示カーソル」を参照してください
- カーソル変数の詳細は、「カーソル変数」を参照してください
- 結合の詳細は、『Oracle Database SQL言語リファレンス』を参照してください

例5-42 データベース表の行の一部を表す%ROWTYPE変数

この例では、サンプル・スキーマHRのemployees表から列first_name、last_nameおよびphone_numberのみを選択する問合せが指定された明示カーソルを宣言します。次に、カーソルにより選択される各列に対応するフィールドを持つレコード変数を宣言します。この変数はemployeesの行の一部を表します。この例を例5-37と比較してください。

Live SQL:

この例は、Oracle Live SQL の「データベース表の行の一部を表す%ROWTYPE 変数」で表示および実行できます

```
DECLARE
```

CURSOR c IS SELECT first_name, last_name, phone_number FROM employees; friend c%ROWTYPE; BEGIN friend.first_name := 'John'; friend.last_name := 'Smith'; friend.phone_number := '1-650-555-1234'; DBMS_OUTPUT.PUT_LINE (friend.first_name || ' ' || friend.last_name || ', ' || friend.phone_number); END; /

結果:

```
John Smith, 1-650-555-1234
```

例5-43 結合行を表す%ROWTYPE変数

この例では、結合問合せが指定された明示カーソルを定義した後、このカーソルにより選択される各列に対応するフィールドを持つレコード変数を宣言します。

Live SQL:

この例は、Oracle Live SQL の「結合行を表す%ROWTYPE 変数」で表示および実行できます

```
DECLARE
```

CURSOR c2 IS SELECT employee_id, email, employees.manager_id, location_id FROM employees, departments WHERE employees.department_id = departments.department_id; join_rec c2%ROWTYPE; -- includes columns from two tables BEGIN NULL; END; /

5.12.4.3 %ROWTYPE属性と仮想列

%ROWTYPE属性を使用して、仮想列を持つ表の行全体を表すレコード変数を定義する場合、そのレコードを表に挿入できません。かわりに、仮想列以外の各レコード・フィールドを表に挿入する必要があります。

例5-44 %ROWTYPEレコードの表への挿入(間違った例)

この例では、仮想列を持つ表の行全体を表すレコード変数を作成し、レコードに移入してから、そのレコードを表に挿入した結果、ORA-54013エラーが発生します。

```
DROP TABLE plch_departure;

CREATE TABLE plch_departure (
  destination    VARCHAR2(100),
  departure_time DATE,
  delay          NUMBER(10),
  expected       GENERATED ALWAYS AS (departure_time + delay/24/60/60)
);


DECLARE
 dep_rec plch_departure%ROWTYPE;
BEGIN
  dep_rec.destination := 'X';
  dep_rec.departure_time := SYSDATE;
  dep_rec.delay := 1500;

  INSERT INTO plch_departure VALUES dep_rec;
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-54013: INSERT operation disallowed on virtual columns
ORA-06512: at line 8
```

例5-45 %ROWTYPEレコードの表への挿入(正しい例)

これは、仮想列以外の各レコード・フィールドを表に挿入することで、例5-44の問題を解決しています。

```
DECLARE
  dep_rec plch_departure%rowtype;
BEGIN
  dep_rec.destination := 'X';
  dep_rec.departure_time := SYSDATE;
  dep_rec.delay := 1500;

  INSERT INTO plch_departure (destination, departure_time, delay)
```

```
  VALUES (dep_rec.destination, dep_rec.departure_time, dep_rec.delay);
end;
/
```

結果:

```
PL/SQL procedure successfully completed.
```

5.12.4.4 %ROWTYPE属性と非表示列

%ROWTYPE属性を使用して、非表示列を持つ表の行を表すレコード変数を定義しており、その後、非表示列を表示列に変更するとします。

「行の一部を表すことができるレコード変数の宣言」に示すように、カーソルとともにレコード変数を定義し、その後、非表示列を表示列に変更しても、レコード変数の構造は変わりません。

ただし、常に行の全体を表すレコード変数の宣言に示すようにレコード変数を定義し、SELECT * INTO文を使用してレコードに値を代入した後、非表示列を表示列に変更すると、レコードの構造が変わります。例5-46を参照してください。

関連項目:

非表示列の概要は、『Oracle Database SQL言語リファレンス』を参照してください。

例5-46 非表示列の表示列への変更により影響を受ける%ROWTYPE

```
CREATE TABLE t (a INT, b INT, c INT INVISIBLE);
INSERT INTO t (a, b, c) VALUES (1, 2, 3);
COMMIT;
DECLARE
  t_rec t%ROWTYPE;  -- t_rec has fields a and b, but not c
BEGIN
  SELECT * INTO t_rec FROM t WHERE ROWNUM < 2;  -- t_rec(a)=1, t_rec(b)=2
  DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
END;
/
```

結果:

```
  DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
                                       *
ERROR at line 5:
ORA-06550: line 5, column 40:
PLS-00302: component 'C' must be declared
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored
```

非表示列を表示列に変更:

```
ALTER TABLE t MODIFY (c VISIBLE);
```

結果:

```
Table altered.
```

前述の無名ブロックを繰り返す:

```
DECLARE
  t_rec t%ROWTYPE;  -- t_rec has fields a, b, and c
BEGIN
  SELECT * INTO t_rec FROM t WHERE ROWNUM < 2;  -- t_rec(a)=1, t_rec(b)=2,
                                                  -- t_rec(c)=3
  DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
END;
/
```

結果:

```
c = 3

PL/SQL procedure successfully completed.
```

5.13 レコード変数への値の代入

レコード変数とは、レコード変数またはコンポジット変数のレコード・コンポーネントのいずれかを意味します。

どのレコード変数にも、各フィールドには個別に値を代入できます。

修飾式(例5-8を参照)を使用して値を代入できます。

場合によっては、あるレコード変数の値を別のレコード変数に代入できます。

データベースの表またはビュー内の行の全体または一部を表すレコード変数の場合は、対象となる行をレコード変数に代入できます。

ここでのトピック

- あるレコード変数から別のレコード変数への代入
- レコード変数への行の全体または一部の代入
- レコード変数へのNULLの代入

5.13.1 あるレコード変数から別のレコード変数への代入

次の場合にかぎり、あるレコード変数の値を別のレコード変数に代入できます。

- 2つの変数は同じRECORD型を持ちます。
- 代入先の変数がRECORD型で、代入元の変数が%ROWTYPEでそれぞれ宣言され、両者のフィールドの数と順序が一致し、対応するフィールドのデータ型が同じである場合。

コンポジット変数のレコード・コンポーネントの場合は、コンポジット変数の型が一致している必要があります。

例5-47 同じRECORD型のレコードから別のレコードへの代入

この例では、name1とname2が同じRECORD型であるため、name1の値をname2に代入できます。

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe'
  );

  name1 name_rec;
  name2 name_rec;

BEGIN
  name1.first := 'Jane'; name1.last := 'Smith';
  DBMS_OUTPUT.PUT_LINE('name1: ' || name1.first || ' ' || name1.last);
  name2 := name1;
  DBMS_OUTPUT.PUT_LINE('name2: ' || name2.first || ' ' || name2.last);
END;
/
```

結果:

```
name1: Jane Smith
name2: Jane Smith
```

例5-48 RECORD型のレコードへの%ROWTYPEレコードの代入

この例では、代入先の変数がRECORD型、代入元の変数が%ROWTYPEでそれぞれ宣言され、両者のフィールドの数と順序が一致し、対応するフィールドのデータ型が同じです。

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe'
  );

  CURSOR c IS
    SELECT first_name, last_name
    FROM employees;

  target name_rec;
  source c%ROWTYPE;

BEGIN
  source.first_name := 'Jane'; source.last_name := 'Smith';

  DBMS_OUTPUT.PUT_LINE (
    'source: ' || source.first_name || ' ' || source.last_name
  );

 target := source;

 DBMS_OUTPUT.PUT_LINE (
   'target: ' || target.first || ' ' || target.last
 );
END;
/
```

結果:

```
source: Jane Smith
target: Jane Smith
```

例5-49 同じRECORD型のネストしたレコードから別のレコードへの代入

この例では、あるネストしたレコードの値を別のネストしたレコードに代入します。これらのネストしたレコードは同じRECORD型を持ちますが、これらがネストされているレコードは同じではありません。

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE,
    last   employees.last_name%TYPE
  );

  TYPE phone_rec IS RECORD (
    name  name_rec,                     -- nested record
    phone employees.phone_number%TYPE
  );

  TYPE email_rec IS RECORD (
    name  name_rec,                     -- nested record
    email employees.email%TYPE
  );

  phone_contact phone_rec;
  email_contact email_rec;

BEGIN
  phone_contact.name.first := 'John';
  phone_contact.name.last := 'Smith';
  phone_contact.phone := '1-650-555-1234';

  email_contact.name := phone_contact.name;
  email_contact.email := (
    email_contact.name.first || '.' ||
    email_contact.name.last  || '@' ||
    'example.com'
  );

  DBMS_OUTPUT.PUT_LINE (email_contact.email);
END;
/
```

結果:

```
John.Smith@example.com
```

5.13.2 レコード変数への行の全体または一部の代入

データベースの表またはビュー内の行の全体または一部を表すレコード変数の場合は、対象となる行をレコード変数に代入できます。

ここでのトピック

- SELECT INTOを使用した行のレコード変数への代入
- FETCHを使用した行のレコード変数への代入
- PL/SQLレコード変数に行を戻すSQL文の使用

5.13.2.1 SELECT INTOを使用した行のレコード変数への代入

単純なSELECT INTO文の構文は次のとおりです。

```
SELECT select_list INTO record_variable_name FROM table_or_view_name;
```

select_listの各列に対して、レコード変数の中には、対応する型互換のフィールドが存在している必要があります。select_list内の列は、レコード・フィールドと同じ順序で並んでいる必要があります。

関連項目:

構文の詳細は、「SELECT INTO文」を参照してください

例5-50 SELECT INTOを使用したレコード変数への値の代入

この例では、レコード変数rec1は、employees表の行の一部(列last_nameおよびemployee_id)を表しています。SELECT INTO文は、job_idが'AD_PRES'の行をemployeesから選択し、選択した行の列last_nameおよびemployee_idの値をrec1の対応するフィールドに代入しています。

```
DECLARE
  TYPE RecordTyp IS RECORD (
    last employees.last_name%TYPE,
    id   employees.employee_id%TYPE
  );
  rec1 RecordTyp;
BEGIN
  SELECT last_name, employee_id INTO rec1
  FROM employees
  WHERE job_id = 'AD_PRES';
  DBMS_OUTPUT.PUT_LINE ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

結果:

```
Employee #100 = King
```

5.13.2.2 FETCHを使用した行のレコード変数への代入

単純なFETCH文の構文は次のとおりです。

```
FETCH cursor INTO record_variable_name;
```

カーソルは問合せに関連付けられています。問合せにより選択されるすべての列に対して、レコード変数の中には、対応する型互換のフィールドが存在している必要があります。カーソルは、明示カーソルまたは強いカーソル変数のいずれかである必要があります。

関連項目:

- 構文の詳細は、「FETCH文」を参照してください
- すべてのカーソルの詳細は、「カーソルの概要」を参照してください
- 明示カーソルの詳細は、「明示カーソル」を参照してください
- カーソル変数の詳細は、「カーソル変数」を参照してください

例5-51 ファンクションが戻すレコードへのFETCHによる値の代入

この例では、RECORD型EmpRecTypの各変数は、employees表の行の一部(列employee_idおよびsalary)を表しています。カーソルおよびファンクションの両方は、型EmpRecTypの値を戻します。このファンクションでは、列employee_idおよびsalaryの値がFETCH文により型EmpRecTypのローカル変数の対応するフィールドに代入されます。

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    emp_id  employees.employee_id%TYPE,
    salary  employees.salary%TYPE
  );

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  highest_paid_emp       EmpRecTyp;
  next_highest_paid_emp  EmpRecTyp;

  FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
    emp_rec  EmpRecTyp;
  BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
      FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
  END nth_highest_salary;

BEGIN
  highest_paid_emp := nth_highest_salary(1);
  next_highest_paid_emp := nth_highest_salary(2);

  DBMS_OUTPUT.PUT_LINE(
    'Highest Paid: #' ||
    highest_paid_emp.emp_id || ', $' ||
    highest_paid_emp.salary
  );
  DBMS_OUTPUT.PUT_LINE(
    'Next Highest Paid: #' ||
    next_highest_paid_emp.emp_id || ', $' ||
    next_highest_paid_emp.salary
  );
END;
```

```
/
```

結果:

```
Highest Paid: #100, $24000
Next Highest Paid: #101, $17000
```

5.13.2.3 PL/SQLレコード変数に行を戻すSQL文の使用

SQLのINSERT文、UPDATE文およびDELETE文には、影響のある行をPL/SQLレコード変数に戻すことができる、オプションのRETURNING INTO句があります。

この句の詳細は、「RETURNING INTO句」を参照してください。

例5-52 UPDATE文を使用したレコード変数への値の代入

この例では、従業員の給与をUPDATE文で更新し、従業員の名前および新しい給与をレコード変数に戻します。

```
DECLARE
  TYPE EmpRec IS RECORD (
    last_name  employees.last_name%TYPE,
    salary     employees.salary%TYPE
  );
  emp_info    EmpRec;
  old_salary  employees.salary%TYPE;
BEGIN
  SELECT salary INTO old_salary
   FROM employees
   WHERE employee_id = 100;

  UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id = 100
    RETURNING last_name, salary INTO emp_info;

  DBMS_OUTPUT.PUT_LINE (
    'Salary of ' || emp_info.last_name || ' raised from ' ||
    old_salary || ' to ' || emp_info.salary
  );
END;
/
```

結果:

```
Salary of King raised from 24000 to 26400
```

5.13.3 レコード変数へのNULLの代入

値NULLをレコード変数に代入すると、その各フィールドに値NULLが代入されます。

この代入は再帰的です(つまり、フィールドがレコードの場合は、それに属するフィールドにも値NULLが代入されます)。

例5-53 レコード変数へのNULLの代入

この例では、レコード変数にNULLを代入する前後に、レコード変数のフィールド(そのうちの1つがレコード)を出力します。

```
DECLARE
  TYPE age_rec IS RECORD (
    years  INTEGER DEFAULT 35,
    months INTEGER DEFAULT 6
  );

  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe',
    age    age_rec
  );

  name name_rec;

  PROCEDURE print_name AS
  BEGIN
    DBMS_OUTPUT.PUT(NVL(name.first, 'NULL') || ' ');
    DBMS_OUTPUT.PUT(NVL(name.last,  'NULL') || ', ');
    DBMS_OUTPUT.PUT(NVL(TO_CHAR(name.age.years), 'NULL') || ' yrs ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(name.age.months), 'NULL') || ' mos');
  END;

BEGIN
  print_name;
  name := NULL;
  print_name;
END;
/
```

結果:

```
John Doe, 35 yrs 6 mos
NULL NULL, NULL yrs NULL mos
```

5.14 レコードの比較

レコードがNULLかどうか、または等しいかどうかは、そのままテストできません。

次のBOOLEAN式は違反となります。

- My_Record IS NULL
- My_Record_1 = My_Record_2
- My_Record_1 > My_Record_2

このようなテストは、独自のファンクションを記述して実装する必要があります。ファンクションの記述の詳細は、「PL/SQLサブプログラム」を参照してください。

5.15 表へのレコードの挿入

SQLのINSERT文に対するPL/SQLの拡張機能によって、レコードを表に挿入できます。

レコードは、表の行を表している必要があります。詳細は、「INSERT文の拡張機能」を参照してください。表へのレコードの挿入の制限については、「レコードの挿入/更新に関する制限」を参照してください。

レコードのコレクションを効率的に表に挿入するには、INSERT文をFORALL文内に置きます。FORALL文の詳細は、「FORALL文」を参照してください。

例5-54 デフォルト値のレコードの挿入による表の初期化

この例では、表scheduleを作成してから、デフォルト値をレコードに入力し、毎週そのレコードを表に挿入して表を初期化しています。(COLUMN書式設定コマンドはSQL*Plusのコマンドです。)

```
DROP TABLE schedule;
CREATE TABLE schedule (
  week  NUMBER,
  Mon   VARCHAR2(10),
  Tue   VARCHAR2(10),
  Wed   VARCHAR2(10),
  Thu   VARCHAR2(10),
  Fri   VARCHAR2(10),
  Sat   VARCHAR2(10),
  Sun   VARCHAR2(10)
);

DECLARE
  default_week  schedule%ROWTYPE;
  i             NUMBER;
BEGIN
  default_week.Mon := '0800-1700';
  default_week.Tue := '0800-1700';
  default_week.Wed := '0800-1700';
  default_week.Thu := '0800-1700';
  default_week.Fri := '0800-1700';
  default_week.Sat := 'Day Off';
  default_week.Sun := 'Day Off';

  FOR i IN 1..6 LOOP
    default_week.week    := i;

    INSERT INTO schedule VALUES default_week;
  END LOOP;
END;
/

COLUMN week FORMAT 99
COLUMN Mon  FORMAT A9
COLUMN Tue  FORMAT A9
COLUMN Wed  FORMAT A9
COLUMN Thu  FORMAT A9
COLUMN Fri  FORMAT A9
COLUMN Sat  FORMAT A9
COLUMN Sun  FORMAT A9

SELECT * FROM schedule;
```

結果:

```
WEEK MON       TUE       WED       THU       FRI       SAT       SUN
---- --------- --------- --------- --------- --------- --------- ---------
   1 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   2 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
```

```
   3 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   4 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   5 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   6 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
```

5.16 レコードを使用した行の更新

SQLのUPDATE文に対するPL/SQLの拡張機能によって、レコードを使用して1つ以上の表の行を更新できます。

レコードは、表の行を表している必要があります。詳細は、「UPDATE文の拡張機能」を参照してください。

レコードを使用した表の行の更新に関する制限については、「レコードの挿入/更新に関する制限」を参照してください。

レコードのコレクションを使用して行セットを更新するには、UPDATE文をFORALL文内に置きます。FORALL文の詳細は、「FORALL文」を参照してください。

例5-55 レコードを使用した行の更新

この例では、新しい値をレコードに入力し、そのレコードを使用して表の最初の3つの行を更新することによって、(例5-54で定義した)表scheduleの最初の3週間のデータを更新しています。

```
DECLARE
  default_week  schedule%ROWTYPE;
BEGIN
  default_week.Mon := 'Day Off';
  default_week.Tue := '0900-1800';
  default_week.Wed := '0900-1800';
  default_week.Thu := '0900-1800';
  default_week.Fri := '0900-1800';
  default_week.Sat := '0900-1800';
  default_week.Sun := 'Day Off';

  FOR i IN 1..3 LOOP
    default_week.week    := i;

    UPDATE schedule
    SET ROW = default_week
    WHERE week = i;
  END LOOP;
END;
/

SELECT * FROM schedule;
```

結果:

```
WEEK MON       TUE       WED       THU       FRI       SAT       SUN
---- --------- --------- --------- --------- --------- --------- ---------
   1 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   2 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   3 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   4 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   5 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   6 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
```

5.17 レコードの挿入/更新に関する制限

レコードの挿入および更新には、次の制限があります。

- レコード変数が使用できるのは、次の位置に限定されます。
  - UPDATE文のSET句の右側
  - INSERT文のVALUES句の中
  - RETURNING句のINTO副次句の中

  レコード変数は、SELECTリスト、WHERE句、GROUP BY句またはORDER BY句では使用できません。
- キーワードROWを指定できる位置は、SET句の左側のみです。また、ROWと副問合せは一緒に使用できません。
- UPDATE文では、ROWが使用されている場合、許可されるSET句は1つのみです。
- INSERT文のVALUES句にレコード変数が含まれている場合は、その句の中で他の変数または値を使用することはできません。
- RETURNING句のINTO副次句にレコード変数が含まれている場合は、その副次句の中で他の変数または値を使用することはできません。
- 次の内容はサポートされません。
  - ネストされたRECORD型
  - RECORD型を戻すファンクション
  - EXECUTE IMMEDIATE文を使用したレコードの挿入および更新

6 PL/SQLの静的SQL

静的SQLは、SQL構文をPL/SQL文中で直接使用できるようにするPL/SQLの機能です。

この章では、静的SQLおよびその使用方法について説明します。

ここでのトピック

- 静的SQLの説明
- カーソルの概要
- 問合せ結果セットの処理
- カーソル変数
- CURSOR式
- トランザクションの処理および制御
- 自律型トランザクション

関連項目:

「静的SQL文における名前解決」

6.1 静的SQLの説明

静的SQLの構文は、特に説明がないかぎり、SQLと同じです。

ここでのトピック

- 文
- 疑似列

6.1.1 文

PL/SQLの静的SQL文を次に示します。特に説明がないかぎり、これらの文の構文は、対応するSQL文と同じです。

- SELECT(この文は問合せとも呼ばれます)

  PL/SQL構文の詳細は、「SELECT INTO文」を参照してください。
- データ操作言語(DML)文:
  - INSERT

    PL/SQL構文の詳細は、「INSERT文の拡張機能」を参照してください。
  - UPDATE

    PL/SQL構文の詳細は、「UPDATE文の拡張機能」を参照してください。
  - DELETE

PL/SQL構文の詳細は、「DELETE文の拡張機能」を参照してください。

- MERGE(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)

ノート:

『Oracle Database SQL 言語リファレンス』における DML の定義とは異なります。

- トランザクション制御言語(TCL)文:
  - COMMIT(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)
  - ROLLBACK(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)
  - SAVEPOINT(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)
  - SET TRANSACTION(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)
- LOCK TABLE(構文の詳細は、『Oracle Database SQL言語リファレンス』を参照)

PL/SQLの静的SQL文では、対応するSQLでバインド変数のプレースホルダを使用できる場合は、PL/SQLの識別子を使用できます。PL/SQLの識別子は、変数または仮パラメータを示している必要があります。

表名や列名などに対してPL/SQL識別子を使用するには、「システム固有の動的SQL」で説明されているEXECUTE IMMEDIATE文を使用します

ノート:

PL/SQL コードによって DML 文が実行された後、変数の値が未定義になるものがあります。たとえば:

- FETCH 文または SELECT 文で例外が呼び出された場合、その文の後の定義変数値は未定義になります。
- 行が処理されない DML 文の後、その DML 文が BULK または複数行にわたる操作でなければ、OUT バインド変数の値は未定義になります。

例6-1 静的SQL文

この例では、PL/SQLの無名ブロックで3つのPL/SQL変数を宣言し、静的SQL文のINSERT、UPDATE、DELETEでそれらを使用しています。このブロックでは、静的SQL文COMMITも使用しています。

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  emp_id          employees_temp.employee_id%TYPE := 299;
  emp_first_name  employees_temp.first_name%TYPE  := 'Bob';
  emp_last_name   employees_temp.last_name%TYPE   := 'Henry';
BEGIN
  INSERT INTO employees_temp (employee_id, first_name, last_name)
```

```
  VALUES (emp_id, emp_first_name, emp_last_name);

  UPDATE employees_temp
  SET first_name = 'Robert'
  WHERE employee_id = emp_id;

  DELETE FROM employees_temp
  WHERE employee_id = emp_id
  RETURNING first_name, last_name
  INTO emp_first_name, emp_last_name;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE (emp_first_name || ' ' || emp_last_name);
END;
/
```

結果:

```
Robert Henry
```

6.1.2 疑似列

疑似列は、表の列のように動作しますが、表には格納されません。

制限などの擬似列の一般情報は、『Oracle Database SQL言語リファレンス』を参照してください。

静的SQLには、次のSQL疑似列が含まれます。

- CURRVALおよびNEXTVAL(詳細は、「PL/SQLのCURRVALおよびNEXTVAL」を参照)。
- LEVEL(詳細は、『Oracle Database SQL言語リファレンス』を参照)
- OBJECT_VALUE(詳細は、『Oracle Database SQL言語リファレンス』を参照)

  関連項目:

  トリガーにおけるOBJECT_VALUEの使用方法の詳細は、「OBJECT_VALUE疑似列」を参照してください

- ROWID(詳細は、『Oracle Database SQL言語リファレンス』を参照)

  関連項目:

  「ROWID擬似列を使用したCURRENT OF句のシミュレート」

- ROWNUM(詳細は、『Oracle Database SQL言語リファレンス』を参照)

6.1.2.1 PL/SQLのCURRVALおよびNEXTVAL

順序が作成されると、SQL文の中でCURRVAL疑似列を使用してその値にアクセスできます(この場合、その順序の現在の値が戻ります)。また、NEXTVAL疑似列を使用してもアクセスできます(この場合は、順序が増加され、新しい値が戻ります)。

これらの疑似列を参照するには、ドット表記法(sequence_name.CURRVALなど)を使用します。

ノート:

sequence_name.NEXTVAL を参照するたびに、トランザクションをコミットするかロールバックするかにかかわらず、順序はすぐに増分され、その変化は永続的になります。

PL/SQL式では、NUMBER式を使用できる場所であればどこでも、sequence_name.CURRVALおよびsequence_name.NEXTVALを使用できます。ただし、次のことに注意してください。

- sequence_name.CURRVALまたはsequence_name.NEXTVALを使用してADTメソッド・パラメータのデフォルト値を指定すると、コンパイル・エラーが発生します。
- PL/SQLは、出現するすべてのsequence_name.CURRVALおよびsequence_name.NEXTVALを評価します(これらが出現するすべての行に対して順序式を評価するSQLとは異なります)。

関連項目:

- 順序の概要は、『Oracle Database SQL言語リファレンス』を参照してください
- CURRVALおよびNEXTVALの構文の詳細は、『Oracle Database SQL言語リファレンス』

例6-2 擬似列CURRVALおよびNEXTVAL

この例では、順序HR.EMPLOYEES_SEQの順序番号を生成し、複数の文でその番号を参照しています。

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DROP TABLE employees_temp2;
CREATE TABLE employees_temp2 AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  seq_value NUMBER;
BEGIN
  -- Generate initial sequence number

  seq_value := employees_seq.NEXTVAL;

  -- Print initial sequence number:

  DBMS_OUTPUT.PUT_LINE (
    'Initial sequence value: ' || TO_CHAR(seq_value)
  );

  -- Use NEXTVAL to create unique number when inserting data:

     INSERT INTO employees_temp (employee_id, first_name, last_name)
     VALUES (employees_seq.NEXTVAL, 'Lynette', 'Smith');

  -- Use CURRVAL to store same value somewhere else:
```

```
    INSERT INTO employees_temp2 VALUES (employees_seq.CURRVAL,
                                        'Morgan', 'Smith');

  /* Because NEXTVAL values might be referenced
     by different users and applications,
     and some NEXTVAL values might not be stored in database,
     there might be gaps in sequence. */

  -- Use CURRVAL to specify record to delete:

     seq_value := employees_seq.CURRVAL;

     DELETE FROM employees_temp2
     WHERE employee_id = seq_value;

  -- Update employee_id with NEXTVAL for specified record:

     UPDATE employees_temp
     SET employee_id = employees_seq.NEXTVAL
     WHERE first_name = 'Lynette'
     AND last_name = 'Smith';

  -- Display final value of CURRVAL:

     seq_value := employees_seq.CURRVAL;

     DBMS_OUTPUT.PUT_LINE (
       'Ending sequence value: ' || TO_CHAR(seq_value)
     );
END;
/
```

6.2 カーソルの概要

カーソルとは、特定のSELECT文またはDML文の処理に関する情報を格納しておく、SQLのプライベート領域を指すポインタです。

> ノート:
>
> このトピックで説明するカーソルはセッション・カーソルです。セッション・カーソルはセッション・メモリーに存在し、セッションが終了すると消滅します。

PL/SQLで構築され管理されるカーソルは、暗黙カーソルです。ユーザーが構築および管理するカーソルは、明示カーソルです。

セッション・カーソルに関する情報は、その属性から取得できます(属性は、プロシージャ文では参照できますが、SQL文では参照できません)。

各ユーザー・セッションでオープンされて解析されているセッション・カーソルをリストするには、動的パフォーマンス・ビューV$OPEN_CURSORを問い合せます。

1つのセッションで同時にオープンできるカーソル数は、次のものにより決まります。

- セッションで使用可能なメモリーの量

- 初期化パラメータOPEN_CURSORSの値

ノート:

一般に、PL/SQL による明示カーソルは、それをセッションが最初にオープンするときにのみ解析されます。また、SQL 文の解析(およびそれによる暗黙カーソルの作成)は、その文が初めて実行されるときにのみ行われます。

解析された SQL 文は、すべてキャッシュに入れられます。SQL 文は、新しい SQL 文によってキャッシュからエージ・アウトされた場合にのみ再解析されます。したがって、明示カーソルを再オープンするには、まずクローズする必要がありますが、PL/SQL はカーソルに関連付けられた問合せを再解析する必要はありません。明示カーソルをクローズしてからすぐに再オープンした場合、PL/SQL は関連付けられた問合せを再解析しません。

ここでのトピック

- 暗黙カーソル
- 明示カーソル

関連項目:

- 動的パフォーマンス・ビューV$OPEN_CURSORの詳細は、『Oracle Databaseリファレンス』を参照してください
- 初期化パラメータOPEN_CURSORSの詳細は、『Oracle Databaseリファレンス』を参照してください

6.2.1 暗黙カーソル

暗黙カーソルは、PL/SQLで構築され管理されるセッション・カーソルです。PL/SQLでは、SELECT文またはDML文を実行するたびに、暗黙カーソルがオープンされます。暗黙カーソルを制御することはできませんが、その属性から情報を取得することはできます。

暗黙カーソルの属性値の構文は、SQLattributeです(このため、暗黙カーソルは、SQLカーソルとも呼ばれます)。SQLattributeは、直前に実行されたSELECT文またはDML文を常に参照します。そのような文が実行されていない場合は、SQLattributeの値はNULLです。

暗黙カーソルは、それが関連付けられている文の実行後、クローズされます。ただし、その属性値は、他のSELECT文またはDML文が実行されるまで、使用可能な状態に保たれます。

直前に実行されたSELECT文またはDML文が、別の有効範囲に存在する場合もあります。属性の値を保存して後で使用する場合は、値をローカル変数に直接代入してください。そうしないと、サブプログラム起動などの他の操作を実行した場合、属性値がテスト前に変更される可能性があります。

暗黙カーソルの属性には次のものがあります。

- SQL%ISOPEN属性: カーソルがオープンしているかどうか
- SQL%FOUND属性: 影響を受けた行があったかどうか
- SQL%NOTFOUND属性: どの行も影響を受けなかったかどうか

- SQL%ROWCOUNT属性: 影響を受けた行数
- SQL%BULK_ROWCOUNT(「FORALL文の影響を受ける行の数の取得」を参照)
- SQL%BULK_EXCEPTIONS(「FORALL文が完了した後のFORALL例外の処理」を参照

関連項目:

構文およびセマンティクスの詳細は、「暗黙カーソル属性」を参照してください

6.2.1.1 SQL%ISOPEN属性: カーソルがオープンしているかどうか

暗黙カーソルは、関連付けられている文の実行後、常にクローズされるため、SQL%ISOPENは常にFALSEを戻します。

6.2.1.2 SQL%FOUND属性: 影響を受けた行があったかどうか

SQL%FOUNDは次の値を戻します。

- SELECT文もDML文も実行されない場合は、NULL。
- SELECT文が1つ以上の行を戻した場合、またはDML文が1つ以上の行に影響を与えた場合は、TRUE。
- それ以外の場合は、FALSE

例6-3では、SQL%FOUNDを使用して、DELETE文によって影響を受けた行があるかどうかを判断しています。

例6-3 SQL%FOUND暗黙カーソル属性

```
DROP TABLE dept_temp;
CREATE TABLE dept_temp AS
  SELECT * FROM departments;

CREATE OR REPLACE PROCEDURE p (
  dept_no NUMBER
) AUTHID CURRENT_USER AS
BEGIN
  DELETE FROM dept_temp
  WHERE department_id = dept_no;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE (
      'Delete succeeded for department number ' || dept_no
    );
  ELSE
    DBMS_OUTPUT.PUT_LINE ('No department number ' || dept_no);
  END IF;
END;
/
BEGIN
  p(270);
  p(400);
END;
/
```

結果:

```
Delete succeeded for department number 270
```

```
No department number 400
```

6.2.1.3 SQL%NOTFOUND属性: どの行も影響を受けなかったかどうか

SQL%NOTFOUND(論理的にSQL%FOUNDの逆)は次の値を戻します。

- SELECT文もDML文も実行されない場合は、NULL。
- SELECT文が1つ以上の行を戻した場合、またはDML文が1つ以上の行に影響を与えた場合は、FALSE。
- それ以外の場合は、TRUE

SQL%NOTFOUND属性は、PL/SQLのSELECT INTO文と組み合せて使用しても効果がありません。これは、次のような理由からです。

- SELECT INTO文によって行が戻されない場合、SQL%NOTFOUNDをチェックする前に、PL/SQLによって事前定義例外NO_DATA_FOUNDがただちに呼び出されます。
- SQL集計ファンクションを起動するSELECT INTO文は、常にある値(多くはNULL)を戻します。このような文の後では、SQL%NOTFOUND属性の値は常にFALSEになるため、属性をチェックする必要はありません。

6.2.1.4 SQL%ROWCOUNT属性: 影響を受けた行数

SQL%ROWCOUNTは次の値を戻します。

- SELECT文もDML文も実行されない場合は、NULL。
- それ以外の場合は、SELECT文によって戻された行の数、またはDML文によって影響を受けた行の数(INTEGER)。

ノート:

サーバーが Oracle Database 12c 以降で、そのクライアントが Oracle Database 11g リリース 2 以前(またはその逆)の場合、SQL%ROWCOUNT が戻す最大値は 4,294,967,295 です。

例6-4では、SQL%ROWCOUNTを使用して、削除された行数を判断しています。

SELECT INTO文がBULK COLLECT句を使用せずに複数の行を戻した場合、PL/SQLによって事前定義の例外TOO_MANY_ROWSが呼び出され、SQL%ROWCOUNTは、問合せを満たす行の実数ではなく、1を戻します。

SQL%ROWCOUNT属性の値はトランザクションの状態には無関係です。したがって、次のようになります。

- トランザクションによってセーブポイントへロールバックされても、SQL%ROWCOUNTの値が、以前のセーブポイントの値に戻ることはありません。
- 自律型トランザクションが終了しても、SQL%ROWCOUNTが親トランザクション内の元の値に戻ることはありません。

例6-4 SQL%ROWCOUNT暗黙カーソル属性

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT * FROM employees;
DECLARE
  mgr_no NUMBER(6) := 122;
BEGIN
  DELETE FROM employees_temp WHERE manager_id = mgr_no;
```

```
  DBMS_OUTPUT.PUT_LINE
    ('Number of employees deleted: ' || TO_CHAR(SQL%ROWCOUNT));
END;
/
```

結果:

```
Number of employees deleted: 8
```

6.2.2 明示カーソル

明示カーソルは、ユーザーが構築および管理するセッション・カーソルです。明示カーソルを宣言および定義する際には、カーソルに名前を付け、問合せ(通常は複数の行を戻す問合せ)と関連付ける必要があります。その後、次のいずれかの方法で問合せ結果セットを処理できます。

- (OPEN文で)明示カーソルをオープンし、(FETCH文で)結果セットから行をフェッチし、(CLOSE文で)明示カーソルをクローズします。
- カーソルFOR LOOP文で明示カーソルを使用します(「カーソルFOR LOOP文による問合せ結果セットの処理」を参照)。

明示カーソルに値を代入したり、明示カーソルを式の中で使用したり、明示カーソルを仮サブプログラム・パラメータやホスト変数として使用することはできません。これらの操作は、カーソル変数を使用することで実行できます(「カーソル変数」を参照)。

明示カーソルまたはカーソル変数は、暗黙カーソルとは異なり、その名前で参照できます。そのため、明示カーソルまたはカーソル変数は、名前付きカーソルと呼ばれます。

ここでのトピック

- 明示カーソルの宣言および定義
- 明示カーソルのオープンおよびクローズ
- 明示カーソルによるデータのフェッチ
- 明示カーソル問合せの変数
- 明示カーソル問合せで列の別名が必要な場合
- パラメータを受け入れる明示カーソル
- 明示カーソル属性

6.2.2.1 明示カーソルの宣言および定義

最初に明示カーソルを宣言して、後で同じブロック、サブプログラムまたはパッケージ内で定義するか、または宣言と定義を同時に行うことができます。

カーソルの宣言のみを行う明示カーソル宣言の構文は、次のとおりです。

```
CURSOR cursor_name [ parameter_list ] RETURN return_type;
```

明示カーソル定義の構文は、次のとおりです。

```
CURSOR cursor_name [ parameter_list ] [ RETURN return_type ]
  IS select_statement;
```

先にカーソルを宣言した場合、後で明示カーソル定義によってカーソルを定義します。それ以外の場合は、明示カーソルの宣言と定義を同時に行います。

例6-5では、3つの明示カーソルを宣言および定義しています。

関連項目:

- 明示カーソルの宣言と定義の構文およびセマンティクスの詳細は、「明示カーソルの宣言および定義」を参照してください
- 「パラメータを受け入れる明示カーソル」

例6-5 明示カーソルの宣言および定義

```
DECLARE
  CURSOR c1 RETURN departments%ROWTYPE;    -- Declare c1

  CURSOR c2 IS                             -- Declare and define c2
    SELECT employee_id, job_id, salary FROM employees
    WHERE salary > 2000;

  CURSOR c1 RETURN departments%ROWTYPE IS  -- Define c1,
    SELECT * FROM departments              -- repeating return type
    WHERE department_id = 110;

  CURSOR c3 RETURN locations%ROWTYPE;      -- Declare c3

  CURSOR c3 IS                             -- Define c3,
    SELECT * FROM locations                -- omitting return type
    WHERE country_id = 'JP';
BEGIN
  NULL;
END;
/
```

6.2.2.2 明示カーソルのオープンおよびクローズ

明示カーソルを宣言および定義したら、OPEN文でそのカーソルをオープンし、次の操作を実行できます。

1. 問合せを処理するデータベース・リソースを割り当てます。
2. 次のように問合せを処理します。
   a. 結果セットを識別します。

      問合せが変数またはカーソル・パラメータを参照している場合、結果セットはそれらの値に影響を受けます。詳細は、「明示カーソル問合せの変数」および「パラメータを受け入れる明示カーソル」を参照してください。

   b. 問合せにFOR UPDATE句がある場合、結果セットの行をロックします。

      詳細は、「SELECT FOR UPDATEカーソルとFOR UPDATEカーソル」を参照してください。
3. カーソルを結果セットの最初の行の前に配置します。

オープンした明示カーソルは、CLOSE文を使用してクローズすることで、そのリソースを再利用できます。カーソルのクローズ後は、

その結果セットからレコードをフェッチしたり、その属性を参照することはできません。これらの操作が試行されると、PL/SQLは事前定義の例外INVALID_CURSORを呼び出します。

クローズしたカーソルは、再オープンできます。明示カーソルは、再オープンする前にクローズする必要があります。それ以外の場合、PL/SQLは事前定義の例外CURSOR_ALREADY_OPENを呼び出します。

関連項目:

- 構文およびセマンティクスの詳細は、「OPEN文」を参照してください
- CLOSE文の構文およびセマンティクスの詳細は、「CLOSE文」を参照してください

6.2.2.3 明示カーソルによるデータのフェッチ

明示カーソルをオープンしたら、FETCH文を使用して、問合せ結果セットの行をフェッチできます。1つの行を戻すFETCH文の基本構文は、次のとおりです。

```
FETCH cursor_name INTO into_clause
```

into_clauseは、変数のリストまたはシングル・レコード変数です。問合せが戻す各列に対して、変数のリストまたはレコードの中に、対応する型互換の変数またはフィールドが存在している必要があります。%TYPEおよび%ROWTYPE属性は、FETCH文で使用する変数およびレコードを宣言する場合に便利です。

FETCH文は、結果セットの現在の行を取り出し、その行の列値を変数またはレコードに格納して、カーソルを次の行に進めます。

通常、FETCH文はLOOP文の内部で使用し、FETCH文で取り出す行がなくなったら、ループを終了します。この終了条件を検出するには、カーソル属性%NOTFOUNDを使用します(「%NOTFOUND属性: どの行もフェッチされていないかどうか」を参照)。PL/SQLは、FETCH文が行を戻さない場合でも、例外を呼び出しません。

例6-6では、LOOP文の内部でFETCHおよび%NOTFOUNDを使用して、2つの明示カーソルの結果セットから一度に1行ずつ行をフェッチします。最初のFETCH文では、列値を変数に取り出しています。2番目のFETCH文では、列値をレコードに取り出しています。変数とレコードは、それぞれ%TYPEと%ROWTYPEで宣言されています。

例6-7では、それぞれが異なるレコード変数にフェッチする5つのFETCH文を使用して、結果セットの最初の5行を5つのレコードにフェッチします。レコード変数は、%ROWTYPEで宣言されています。

関連項目:

- FETCH文の構文およびセマンティクスの詳細は、「FETCH文」を参照してください
- 一度に複数の行を戻すFETCH文の詳細は、「BULK COLLECT句を使用したFETCH文」を参照してください

例6-6 LOOP文内部のFETCH文

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;
```

```
  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id
  CURSOR c2 IS
    SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]')
    ORDER BY job_id;
  v_employees employees%ROWTYPE;  -- record variable for row of table
BEGIN
  OPEN c1;
  LOOP  -- Fetches 2 columns into variables
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE( '-------------------------------------' );
  OPEN c2;
  LOOP  -- Fetches entire row into the v_employees record
    FETCH c2 INTO v_employees;
    EXIT WHEN c2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                              v_employees.job_id );
  END LOOP;
  CLOSE c2;
END;
/
```

結果:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-------------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
Zlotkey                   SA_MAN
```

例6-7 異なる変数への同じ明示カーソルのフェッチ

```
DECLARE
  CURSOR c IS
    SELECT e.job_id, j.job_title
    FROM employees e, jobs j
    WHERE e.job_id = j.job_id AND e.manager_id = 100
    ORDER BY last_name;

  -- Record variables for rows of cursor result set:

  job1 c%ROWTYPE;
  job2 c%ROWTYPE;
  job3 c%ROWTYPE;
  job4 c%ROWTYPE;
  job5 c%ROWTYPE;
```

```
BEGIN
  OPEN c;
  FETCH c INTO job1;  -- fetches first row
  FETCH c INTO job2;  -- fetches second row
  FETCH c INTO job3;  -- fetches third row
  FETCH c INTO job4;  -- fetches fourth row
  FETCH c INTO job5;  -- fetches fifth row
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE(job1.job_title || ' (' || job1.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job2.job_title || ' (' || job2.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job3.job_title || ' (' || job3.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job4.job_title || ' (' || job4.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job5.job_title || ' (' || job5.job_id || ')');
END;
/
```

結果:

```
Sales Manager (SA_MAN)
Administration Vice President (AD_VP)
Sales Manager (SA_MAN)
Stock Manager (ST_MAN)
Marketing Manager (MK_MAN)
PL/SQL procedure successfully completed.
```

6.2.2.4 明示カーソル問合せの変数

明示カーソル問合せでは、有効範囲内にある任意の変数を参照できます。明示カーソルをオープンすると、PL/SQLによって問合せ内のすべての変数が評価され、結果セットの識別時にそれらの値が使用されます。変数の値を後で変更しても、結果セットは変更されません。

例6-8では、明示カーソル問合せで変数factorを参照しています。カーソルがオープンしたとき、factorの値は2です。したがって、sal_multipleは、各フェッチ後にfactorが増分されても、常にsalの2倍になります。

結果セットを変更するには、例6-9に示すとおり、カーソルをクローズし、変数の値を変更してからカーソルを再度オープンする必要があります。

例6-8 明示カーソル問合せの変数(結果セットの変更なし)

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  OPEN c1;  -- PL/SQL evaluates factor

  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
```

```
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE c1;
END;
/
```

結果:

```
factor = 2
sal          = 4400
sal_multiple = 8800
factor = 3
sal          = 24000
sal_multiple = 48000
factor = 4
sal          = 17000
sal_multiple = 34000
factor = 5
sal          = 17000
sal_multiple = 34000
```

例6-9 明示カーソル問合せの変数(結果セットの変更あり)

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;
```

```
END;
/
```

結果:

```
factor = 2
sal          = 4400
sal_multiple = 8800
sal          = 24000
sal_multiple = 48000
sal          = 17000
sal_multiple = 34000
sal          = 17000
sal_multiple = 34000
factor = 3
sal          = 4400
sal_multiple = 13200
sal          = 24000
sal_multiple = 72000
sal          = 17000
sal_multiple = 51000
sal          = 17000
sal_multiple = 51000
```

6.2.2.5 明示カーソル問合せで列の別名が必要な場合

明示カーソル問合せに仮想列(式)が含まれる場合、次のいずれかに該当すると、その列には別名が必要です。

- カーソルを使用して、%ROWTYPEで宣言されたレコードにフェッチする場合。
- 仮想列をプログラム内で参照する場合。

例6-10では、前述の両方の理由のため、明示カーソル内の仮想列に別名が必要です。

関連項目:

例6-21

例6-10 仮想列に別名が必要な明示カーソル

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id,
           (salary * .05) raise
    FROM employees
    WHERE job_id LIKE '%_MAN'
    ORDER BY employee_id;
  emp_rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'Raise for employee #' || emp_rec.employee_id ||
      ' is $' || emp_rec.raise
```

```
    );
  END LOOP;
  CLOSE c1;
END;
/
```

結果:

```
Raise for employee #114 is $550
Raise for employee #120 is $400
Raise for employee #121 is $410
Raise for employee #122 is $395
Raise for employee #123 is $325
Raise for employee #124 is $368.445
Raise for employee #145 is $700
Raise for employee #146 is $675
Raise for employee #147 is $600
Raise for employee #148 is $550
Raise for employee #149 is $525
Raise for employee #201 is $650
```

6.2.2.6 パラメータを受け入れる明示カーソル

仮パラメータを含む明示カーソルを作成し、オープンするたびにそのカーソルに異なる実パラメータを渡すことができます。カーソル問合せ内では、定数を使用できる場所であればどこでも仮カーソル・パラメータを使用できます。カーソル問合せの外部では、仮カーソル・パラメータは参照できません。

ヒント:

混同を避けるため、仮カーソル・パラメータと実カーソル・パラメータには異なる名前を使用してください。

例6-11では、2つの仮パラメータで職種とその最大給与を表す明示カーソルを作成しています。カーソル問合せは、職種と最大給与を指定してオープンされると、その職種で給与が余分に支払われている従業員を選択します(問合せは、そのような従業員ごとに氏名と超過支払額を選択します)。次に、この例では、カーソル問合せの結果セットを出力するプロシージャを作成しています(プロシージャの詳細は、「PL/SQLサブプログラム」を参照してください)。最後に、この例では、実パラメータの1つのセットを使用したカーソルのオープン、その結果セットの出力、カーソルのクローズ、および異なる実パラメータを使用したカーソルのオープン、その結果セットの出力、カーソルのクローズを行っています。

ここでのトピック

- デフォルト値を持つ仮カーソル・パラメータ
- デフォルト値を持つ仮カーソル・パラメータの追加

関連項目:

- 仮カーソル・パラメータの詳細は、「明示カーソルの宣言および定義」を参照してください
- 実カーソル・パラメータの詳細は、「OPEN文」を参照してください

例6-11 パラメータを受け入れる明示カーソル

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('-----------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Stock Clerks:');
  DBMS_OUTPUT.PUT_LINE('-----------------------');
  OPEN c('ST_CLERK', 5000);
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c('SA_REP', 10000);
  print_overpaid;
  CLOSE c;
END;
/
```

結果:

```
-----------------------
Overpaid Stock Clerks:
-----------------------
--------------------------------
Overpaid Sales Representatives:
--------------------------------
Vishney, Clara (by 500)
Abel, Ellen (by 1000)
Ozer, Lisa (by 1500)

PL/SQL procedure successfully completed.
```

6.2.2.6.1 デフォルト値を持つ仮カーソル・パラメータ

仮パラメータを使用して明示カーソルを作成する場合、それらのカーソルにデフォルト値を指定できます。仮パラメータがデフォル

ト値を持つ場合、対応する実パラメータはオプションになります。実パラメータを指定せずにカーソルをオープンすると、仮パラメータにそのデフォルト値が割り当てられます。

例6-12では、場所IDを表す仮パラメータを持つ明示カーソルを作成しています。このパラメータのデフォルト値は、企業本社の場所IDです。

例6-12 デフォルト値を持つカーソル・パラメータ

```
DECLARE
  CURSOR c (location NUMBER DEFAULT 1700) IS
    SELECT d.department_name,
           e.last_name manager,
           l.city
    FROM departments d, employees e, locations l
    WHERE l.location_id = location
      AND l.location_id = d.location_id
      AND d.department_id = e.department_id
    ORDER BY d.department_id;

  PROCEDURE print_depts IS
    dept_name  departments.department_name%TYPE;
    mgr_name   employees.last_name%TYPE;
    city_name  locations.city%TYPE;
  BEGIN
    LOOP
      FETCH c INTO dept_name, mgr_name, city_name;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(dept_name || ' (Manager: ' || mgr_name || ')');
    END LOOP;
  END print_depts;

BEGIN
  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS AT HEADQUARTERS:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c;
  print_depts;
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS IN CANADA:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c(1800); -- Toronto
  print_depts;
  CLOSE c;
  OPEN c(1900); -- Whitehorse
  print_depts;
  CLOSE c;
END;
/
```

結果は次のようになります。

```
DEPARTMENTS AT HEADQUARTERS:
--------------------------------
Administration (Manager: Whalen)
Purchasing (Manager: Colmenares)
```

```
Purchasing (Manager: Baida)
Purchasing (Manager: Himuro)
Purchasing (Manager: Raphaely)
Purchasing (Manager: Khoo)
Purchasing (Manager: Tobias)
Executive (Manager: Kochhar)
Executive (Manager: De Haan)
Executive (Manager: King)
Finance (Manager: Popp)
Finance (Manager: Greenberg)
Finance (Manager: Faviet)
Finance (Manager: Chen)
Finance (Manager: Urman)
Finance (Manager: Sciarra)
Accounting (Manager: Gietz)
Accounting (Manager: Higgins)
--------------------------------
DEPARTMENTS IN CANADA:
--------------------------------
Marketing (Manager: Hartstein)
Marketing (Manager: Fay)

PL/SQL procedure successfully completed.
```

6.2.2.6.2 デフォルト値を持つ仮カーソル・パラメータの追加

カーソルに仮パラメータを追加し、その追加したパラメータにデフォルト値を指定する場合、カーソルに対する既存の参照を変更する必要はありません。例6-13を例6-11と比較してください。

例6-13 既存のカーソルへの仮パラメータの追加

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER,
            hired DATE DEFAULT TO_DATE('31-DEC-1999', 'DD-MON-YYYY')) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    AND hire_date > hired
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
```

```
  DBMS_OUTPUT.PUT_LINE('-------------------------------');
  OPEN c('SA_REP', 10000);  -- existing reference
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives Hired After 2004:');
  DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
  OPEN c('SA_REP', 10000, TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));
                          -- new reference
  print_overpaid;
  CLOSE c;
END;
/
```

結果:

```
-------------------------------
Overpaid Sales Representatives:
-------------------------------
Vishney, Clara (by 500)
Abel, Ellen (by 1000)
Ozer, Lisa (by 1500)
-----------------------------------------------
Overpaid Sales Representatives Hired After 2004:
-----------------------------------------------
Vishney, Clara (by 500)
Ozer, Lisa (by 1500)

PL/SQL procedure successfully completed.
```

6.2.2.7 明示カーソル属性

明示カーソル属性値の構文は、cursor_nameのすぐ後にattributeが続きます(たとえば、c1%ISOPEN)。

ノート:

明示カーソルとカーソル変数(名前付きカーソル)は、同じ属性を持ちます。このトピックは、特に説明がないかぎり、すべての名前付きカーソルに適用されます。

明示カーソルの属性には次のものがあります。

- %ISOPEN属性: カーソルがオープンしているかどうか
- %FOUND属性: 1行がフェッチされたかどうか
- %NOTFOUND属性: どの行もフェッチされていないかどうか
- %ROWCOUNT属性: フェッチされた行数

明示カーソルをオープンしていない場合、%ISOPEN以外の属性を参照すると、事前定義の例外INVALID_CURSORが呼び出されます。

関連項目:

名前付きカーソル(明示カーソルとカーソル変数)の属性の構文およびセマンティクスの詳細は、「名前付きカーソル属性」を参照してください

6.2.2.7.1 %ISOPEN属性: カーソルがオープンしているかどうか

%ISOPENは、明示カーソルをオープンしている場合はTRUEを、そうでない場合はFALSEを戻します。

%ISOPENは、次の場合に便利です。

- 明示カーソルをオープンする前にまだオープンしていないことを確認する場合。

  すでにオープンしている明示カーソルをオープンしようとすると、PL/SQLによって事前定義の例外CURSOR_ALREADY_OPENが呼び出されます。明示カーソルは、再オープンする前にクローズする必要があります。

  ノート:

  前述の内容は、カーソル変数には適用されません。

- 明示カーソルをクローズする前にオープンしていることを確認する場合。

例6-14では、明示カーソルc1がオープンしていない場合にのみオープンし、オープンしている場合にのみクローズします。

例6-14 %ISOPEN明示カーソル属性

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;
  the_name employees.last_name%TYPE;
  the_salary employees.salary%TYPE;
BEGIN
  IF NOT c1%ISOPEN THEN
    OPEN c1;
  END IF;
  FETCH c1 INTO the_name, the_salary;
  IF c1%ISOPEN THEN
    CLOSE c1;
  END IF;
END;
/
```

6.2.2.7.2 %FOUND属性: 1行がフェッチされたかどうか

%FOUNDは次の値を戻します。

- 明示カーソルのオープン後、最初のフェッチが実行されるまでは、NULL。
- 明示カーソルからの最新のフェッチで行が戻された場合は、TRUE。
- それ以外の場合は、FALSE

%FOUNDは、処理する必要のあるフェッチ行が存在するかどうかを確認する場合に便利です。

例6-15では、結果セットをループして各フェッチ行を出力し、フェッチする行がなくなると終了します。

例6-15 %FOUND明示カーソル属性

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;
  my_ename   employees.last_name%TYPE;
  my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%FOUND THEN  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
    ELSE  -- fetch failed
      EXIT;
    END IF;
  END LOOP;
END;
/
```

結果:

```
Name = Austin, salary = 4800
Name = De Haan, salary = 17000
Name = Ernst, salary = 6000
Name = Faviet, salary = 9000
Name = Greenberg, salary = 12008
Name = Hunold, salary = 9000
Name = King, salary = 24000
Name = Kochhar, salary = 17000
Name = Lorentz, salary = 4200
Name = Pataballa, salary = 4800
```

6.2.2.7.3 %NOTFOUND属性: どの行もフェッチされていないかどうか

%NOTFOUND(論理的に%FOUNDの逆)は次の値を戻します。

- 明示カーソルのオープン後、最初のフェッチが実行されるまでは、NULL。
- 明示カーソルからの最新のフェッチで行が戻された場合は、FALSE。
- それ以外の場合は、TRUE

例6-16に示すとおり、%NOTFOUNDは、FETCHが行を戻さなくなったときにループを終了する場合に便利です。

例6-16 %NOTFOUND明示カーソル属性

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;
   my_ename   employees.last_name%TYPE;
   my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
```

```
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%NOTFOUND THEN -- fetch failed
      EXIT;
    ELSE  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE
        ('Name = ' || my_ename || ', salary = ' || my_salary);
    END IF;
  END LOOP;
END;
/
```

結果:

```
Name = Austin, salary = 4800
Name = De Haan, salary = 17000
Name = Ernst, salary = 6000
Name = Faviet, salary = 9000
Name = Greenberg, salary = 12008
Name = Hunold, salary = 9000
Name = King, salary = 24000
Name = Kochhar, salary = 17000
Name = Lorentz, salary = 4200
Name = Pataballa, salary = 4800
```

6.2.2.7.4 %ROWCOUNT属性: フェッチされた行数

%ROWCOUNTは次の値を戻します。

- 明示カーソルのオープン後、最初のフェッチが実行されるまでは、0(ゼロ)。
- それ以外の場合は、フェッチされた行数(INTEGER)。

ノート:

サーバーが Oracle Database 12c 以降で、そのクライアントが Oracle Database 11g2 以前(またはその逆)の場合、SQL%ROWCOUNT が戻す最大値は 4,294,967,295 です。

例6-17では、フェッチした行の数をカウントして出力し、5行目のフェッチ後にメッセージを出力します。

例6-17 %ROWCOUNT明示カーソル属性

```
DECLARE
  CURSOR c1 IS
    SELECT last_name FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;
  name  employees.last_name%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO name;
    EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
    DBMS_OUTPUT.PUT_LINE(c1%ROWCOUNT || '. ' || name);
    IF c1%ROWCOUNT = 5 THEN
       DBMS_OUTPUT.PUT_LINE('--- Fetched 5th row ---');
```

```
    END IF;
  END LOOP;
  CLOSE c1;
END;
/
```

結果:

```
1. Abel
2. Ande
3. Atkinson
4. Austin
5. Baer
--- Fetched 5th row ---
6. Baida
7. Banda
8. Bates
9. Bell
10. Bernstein
```

6.3 問合せ結果セットの処理

PL/SQLでは、従来のデータベース・プログラミングと同様に、カーソルを使用して問合せ結果セットを処理できます。ただし、PL/SQLでは、暗黙カーソルまたは明示カーソルを使用できます。

前者ではコードが削減されますが、後者ではより柔軟な処理が可能です。たとえば、明示カーソルはパラメータを受け取ることができます。

次のPL/SQL文では、PL/SQLによって自動で定義および管理される暗黙カーソルを使用します。

- SELECT INTO
- 暗黙カーソルFOR LOOP

次のPL/SQL文では、明示カーソルを使用します。

- 明示カーソルFOR LOOP

  ユーザーが明示カーソルを定義しますが、文の実行中はPL/SQLが管理します。

- OPEN、FETCHおよびCLOSE

  ユーザーが明示カーソルを定義および管理します。

ノート:

問合せによって行が戻されない場合、PL/SQL によって例外 NO_DATA_FOUND が呼び出されます。

ここでのトピック

- SELECT INTO文による問合せ結果セットの処理
- カーソルFOR LOOP文による問合せ結果セットの処理
- 明示カーソル、OPEN、FETCHおよびCLOSEによる問合せ結果セットの処理

- 副問合せによる問合せ結果セットの処理

関連項目:

- 「パラメータを受け入れる明示カーソル」
- クライアントへの結果セットの戻しの詳細は、『Oracle Database開発ガイド』を参照してください。
- 例外処理の詳細は、「例外ハンドラ」を参照してください

6.3.1 SELECT INTO文による問合せ結果セットの処理

暗黙カーソルを使用して、SELECT INTO文は、1つ以上のデータベース表から値を取り出し(この処理はSQLのSELECT文と同様です)、それらの値を変数に格納します(この処理はSQLのSELECT文では実行されません)。

ここでのトピック

- 単一行の結果セットの処理
- 多数の複数行の結果セットの処理

関連項目:

SELECT INTO文の構文およびセマンティクスの詳細は、「SELECT INTO文」を参照してください

6.3.1.1 単一行の結果セットの処理

問合せで1つの行のみが戻されると予想される場合は、SELECT INTO文を使用して、その行の値を1つ以上のスカラー変数または1つのレコード変数に格納します。

問合せで複数の行が戻される可能性があるが、n番目の行のみを処理すればよい場合、WHERE ROWNUM=n句を使用して結果セットを目的の行に制限します。

関連項目:

- 「SELECT INTO文を使用した変数への値の代入」
- 「SELECT INTOを使用した行のレコード変数への代入」
- ROWNUM疑似列の詳細は、『Oracle Database SQL言語リファレンス』を参照してください

6.3.1.2 多数の複数行の結果セットの処理

大量の表データを変数に代入する必要がある場合、BULK COLLECT句付きでSELECT INTO文を使用することをお薦めします。

この文は、結果セット全体を1つ以上のコレクション変数に取り出します。

詳細は、「BULK COLLECT句を使用したSELECT INTO文」を参照してください。

6.3.2 カーソルFOR LOOP文による問合せ結果セットの処理

カーソルFOR LOOP文では、SELECT文を実行してから、すぐに結果セットの行をループできます。

この文では、暗黙カーソルまたは明示カーソルを使用できます(ただしカーソル変数は使用できません)。

SELECT文をカーソルFOR LOOP文でのみ使用する場合、例6-18に示すとおり、カーソルFOR LOOP文の内側にSELECT文を指定します。この形式のカーソルFOR LOOP文は、暗黙カーソルを使用するため、暗黙カーソルのFOR LOOP文と呼ばれます。暗黙カーソルは文に対して内部的であるため、SQLという名前で参照することはできません。

同じPL/SQLユニットでSELECT文を複数回使用する場合、例6-19に示すとおり、そのユニットで明示カーソルを定義し、カーソルFOR LOOP文にそのカーソルを指定します。この形式のカーソルFOR LOOP文は、明示カーソルのFOR LOOP文と呼ばれます。同じPL/SQLユニット内の別の場所でも同じ明示カーソルを使用できます。

カーソルFOR LOOP文は、そのカーソルが戻す型の%ROWTYPEレコード変数として暗黙的にループ索引を宣言します。このレコードは、ループに対してローカルで、ループの実行中のみ存在します。ループの内側の文は、レコードとそのフィールドを参照できます。例6-21に示すように、仮想列は別名でのみ参照できます。

ループ索引のレコード変数の宣言後、FOR LOOP文によって指定したカーソルがオープンされます。ループが反復されるたびに、FOR LOOP文によって結果セットから行がフェッチされ、レコードに格納されます。フェッチする行がなくなると、カーソルFOR LOOP文によってカーソルがクローズされます。また、ループの内側の文によって制御がループの外側に移された場合や、PL/SQLによって例外が呼び出された場合も、カーソルがクローズされます。

関連項目:

カーソルFOR LOOP文の構文およびセマンティクスの詳細は、「カーソルFOR LOOP文」を参照してください

ノート:

カーソル FOR LOOP 文の内側で例外が呼び出されると、カーソルは、例外ハンドラが実行される前にクローズされます。したがって、ハンドラでは明示カーソルの属性の値を参照できません。

例6-18 暗黙カーソルFOR LOOP文

この例では、暗黙カーソルFOR LOOP文で、担当マネージャのIDが120より大きいすべての事務員の名前および職種IDを出力します。

```
BEGIN
  FOR item IN (
    SELECT last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND manager_id > 120
    ORDER BY last_name
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE
```

```
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

結果:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...
Name = Walsh, Job = SH_CLERK
```

例6-19 明示カーソルFOR LOOP文

この例は例6-18と類似していますが、明示カーソルFOR LOOP文を使用しています。

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND manager_id > 120
    ORDER BY last_name;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

結果:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...
Name = Walsh, Job = SH_CLERK
```

例6-20 明示カーソルFOR LOOP文へのパラメータの引渡し

この例では、2つのパラメータを受け入れる明示カーソルを宣言および定義し、明示カーソルFOR LOOP文でそのカーソルを使用して、指定された部門内で給与が指定額を超える従業員の給与を示します。

```
DECLARE
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
    WHERE job_id = job
    AND salary > max_wage;
BEGIN
  FOR person IN c1('ST_CLERK', 3000)
  LOOP
     -- process data record
    DBMS_OUTPUT.PUT_LINE (
      'Name = ' || person.last_name || ', salary = ' ||
      person.salary || ', Job Id = ' || person.job_id
    );
```

```
  END LOOP;
END;
/
```

結果:

```
Name = Nayer, salary = 3200, Job Id = ST_CLERK
Name = Bissot, salary = 3300, Job Id = ST_CLERK
Name = Mallin, salary = 3300, Job Id = ST_CLERK
Name = Ladwig, salary = 3600, Job Id = ST_CLERK
Name = Stiles, salary = 3200, Job Id = ST_CLERK
Name = Rajs, salary = 3500, Job Id = ST_CLERK
Name = Davies, salary = 3100, Job Id = ST_CLERK
```

例6-21 カーソルFOR LOOPによる仮想列の参照

この例では、暗黙カーソルFOR LOOPで、仮想列をその別名full_nameおよびdream_salaryで参照しています。

```
BEGIN
  FOR item IN (
    SELECT first_name || ' ' || last_name AS full_name,
           salary * 10                    AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
    ORDER BY dream_salary DESC, last_name ASC
  ) LOOP
    DBMS_OUTPUT.PUT_LINE
      (item.full_name || ' dreams of making ' || item.dream_salary);
  END LOOP;
END;
/
```

結果:

```
Stephen King dreams of making 240000
Lex De Haan dreams of making 170000
Neena Kochhar dreams of making 170000
Alexander Hunold dreams of making 90000
Bruce Ernst dreams of making 60000
```

6.3.3 明示カーソル、OPEN、FETCHおよびCLOSEによる問合せ結果セットの処理

問合せ結果セットの処理を完全に制御するには、明示カーソルを宣言し、それらのカーソルをOPEN文、FETCH文およびCLOSE文を使用して管理します。

この結果セットの処理方法は、他の方法より複雑ですが、より柔軟な処理が可能です。たとえば、次のことが可能です。

- 複数のカーソルを使用して、複数の結果セットをパラレルに処理できます。
- 1回のループで複数の行を処理したり、行をスキップしたり、処理を複数のループに分割することができます。
- あるPL/SQLユニットで問合せを指定し、別のユニットで行を取り出すことができます。

詳細および例は、「明示カーソル」を参照してください。

6.3.4 副問合せによる問合せ結果セットの処理

問合せ結果セットを処理する際に、その結果セットをループして行ごとに別の問合せを実行する場合、ループの内側から2番目の問合せを削除し、その問合せを最初の問合せの副問合せにすると、パフォーマンスを向上できます。

通常の副問合せが表ごとに評価されるのに対し、相関副問合せは行ごとに評価されます。

副問合せの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例6-22 親問合せのFROM句内の副問合せ

この例では、FROM句に副問合せを含む問合せを使用して明示カーソルc1を定義しています。

```
DECLARE
  CURSOR c1 IS
    SELECT t1.department_id, department_name, staff
    FROM departments t1,
         ( SELECT department_id, COUNT(*) AS staff
           FROM employees
           GROUP BY department_id
         ) t2
    WHERE (t1.department_id = t2.department_id) AND staff >= 5
    ORDER BY staff;
BEGIN
   FOR dept IN c1
   LOOP
     DBMS_OUTPUT.PUT_LINE ('Department = '
       || dept.department_name || ', staff = ' || dept.staff);
   END LOOP;
END;
/
```

結果:

```
Department = IT, staff = 5
Department = Finance, staff = 6
Department = Purchasing, staff = 6
Department = Sales, staff = 34
Department = Shipping, staff = 45
```

例6-23 相関副問合せ

この例では、給与が部門平均を上回っている従業員の名前と給与を戻しています。相関副問合せでは、表の各行について、対応する部門の平均給与を計算します。

```
DECLARE
  CURSOR c1 IS
    SELECT department_id, last_name, salary
    FROM employees t
    WHERE salary > ( SELECT AVG(salary)
                     FROM employees
                     WHERE t.department_id = department_id
                   )
    ORDER BY department_id, last_name;
BEGIN
  FOR person IN c1
```

```
  LOOP
    DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
  END LOOP;
END;
/
```

結果:

```
Making above-average salary = Hartstein
Making above-average salary = Raphaely
Making above-average salary = Bell
...
Making above-average salary = Higgins
```

6.4 カーソル変数

カーソル変数は、明示カーソルと似ていますが、次の点が異なります。

- 1つの問合せに限定されません。

  カーソル変数を問合せに対してオープンし、結果セットを処理した後、カーソル変数を別の問合せのために使用できます。

- 値を代入できます。
- 式で使用できます。
- サブプログラム・パラメータとして使用できます。

  カーソル変数は、サブプログラム間で問合せ結果セットを渡すために使用できます。

- ホスト変数として使用できます。

  カーソル変数は、PL/SQLのストアド・サブプログラムとそれらのクライアントとの間で問合せ結果セットを渡すために使用できます。

- パラメータを受け入れることはできません。

  カーソル変数にパラメータを渡すことはできませんが、問合せ全体を渡すことができます。問合せには変数を組み込むことができます。

カーソル変数がこのような柔軟性を持つ理由は、それがポインタであるため(その値が項目自体ではなく項目のアドレスを示すため)です。

カーソル変数を参照する前に、カーソル変数をオープンするか、オープンされたPL/SQLカーソル変数またはホスト・カーソル変数の値をカーソル変数に代入することで、SQL作業領域を指すようにする必要があります。

ノート:

カーソル変数と明示カーソルには互換性がありません(一方の値が期待されている場所で、もう一方は使用できません)。たとえば、カーソル FOR LOOP 文の中ではカーソル変数を参照できません。

ここでのトピック

- カーソル変数の作成
- カーソル変数のオープンおよびクローズ
- カーソル変数によるデータのフェッチ
- カーソル変数への値の代入
- カーソル変数問合せの変数
- コレクションの問合せ
- カーソル変数の属性
- サブプログラム・パラメータとしてのカーソル変数
- ホスト変数としてのカーソル変数

関連項目:

- 明示カーソルの詳細は、「明示カーソル」を参照してください
- 「カーソル変数の制限」
- カーソル変数のその他のメリットの詳細は、『Oracle Database開発ガイド』を参照してください
- カーソル変数のデメリットの詳細は、『Oracle Database開発ガイド』を参照してください。

6.4.1 カーソル変数の作成

カーソル変数を作成するには、事前定義のSYS_REFCURSOR型の変数を宣言するか、REF CURSOR型を定義してからその型の変数を宣言します。

ノート:

カーソル変数は、通称で REF CURSOR と呼ばれることがあります。

REF CURSOR型の定義の基本構文は、次のとおりです。

```
TYPE type_name IS REF CURSOR [ RETURN return_type ]
```

構文およびセマンティクスの詳細は、「カーソル変数の宣言」を参照してください。

return_typeを指定する場合、REF CURSOR型とその型のカーソル変数は、強い型になりますが、指定しない場合、それらは弱い型になります。SYS_REFCURSORとその型のカーソル変数は、弱い型です。

強いカーソル変数には、指定した型を戻す問合せのみを関連付けることができます。弱いカーソル変数には、任意の問合せを関連付けることができます。

弱いカーソル変数は、強いカーソル変数よりエラーの発生する可能性が高まりますが、より柔軟な処理が可能です。弱いREF CURSOR型は、相互に互換性があり、事前定義のSYS_REFCURSOR型とも互換性があります。弱いカーソル変数の値は、他の弱いカーソル変数に代入できます。

強いカーソル変数の値を別の強いカーソル変数に代入できるのは、(戻り型が同じである以外に)両方のカーソル変数の型が同じである場合のみです。

> ノート:
>
> テーブル・ファンクションの弱いカーソル変数引数をパーティション化するには、PARTITION BY RANGE や PARTITION BY HASH ではなく、PARTITION BY ANY 句のみを使用します。
>
> 構文およびセマンティクスの詳細は、PARALLEL_ENABLE 句を参照してください。

例6-24 カーソル変数の宣言

この例では、強い型と弱い型のREF CURSOR、それらの型の変数、および事前定義のSYS_REFCURSOR型の変数を定義しています。

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;  -- strong type
  TYPE genericcurtyp IS REF CURSOR;                       -- weak type
  cursor1  empcurtyp;       -- strong cursor variable
  cursor2  genericcurtyp;   -- weak cursor variable
  my_cursor SYS_REFCURSOR;  -- weak cursor variable
  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;  -- strong type
  dept_cv deptcurtyp;  -- strong cursor variable
BEGIN
  NULL;
END;
/
```

例6-25 ユーザー定義の戻り型を持つカーソル変数

この例では、EmpRecTypは、ユーザー定義のRECORD型です。

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    employee_id NUMBER,
    last_name VARCHAR2(25),
    salary   NUMBER(8,2));
  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  emp_cv EmpCurTyp;
BEGIN
  NULL;
END;
/
```

6.4.2 カーソル変数のオープンおよびクローズ

カーソル変数を宣言したら、OPEN FOR文でその変数をオープンし、次の操作を実行できます。

1. カーソル変数を問合せに関連付けます(通常、この問合せで複数の行が戻されます)。

   問合せには、バインド変数用のプレースホルダを含め、その値をOPEN FOR文のUSING句で指定できます。

2. 問合せを処理するデータベース・リソースを割り当てます。

3. 次のように問合せを処理します。

   a. 結果セットを識別します。

      問合せが変数を参照している場合、結果セットはそれらの値に影響を受けます。詳細は、「カーソル変数問合せの変数」を参照してください。

   b. 問合せにFOR UPDATE句がある場合、結果セットの行をロックします。

      詳細は、「SELECT FOR UPDATEカーソルとFOR UPDATEカーソル」を参照してください。

4. カーソルを結果セットの最初の行の前に配置します。

カーソル変数は、再オープンする(別のOPEN FOR文で使用する)前にクローズする必要はありません。カーソル変数を再オープンすると、前にその変数に関連付けられていた問合せは失われます。

カーソル変数が必要なくなったら、CLOSE文を使用してクローズすることで、そのリソースを再利用できます。カーソル変数のクローズ後は、その結果セットからレコードをフェッチしたり、その属性を参照することはできません。これらの操作が試行されると、PL/SQLは事前定義の例外INVALID_CURSORを呼び出します。

クローズしたカーソル変数は、再オープンできます。

関連項目:

- 「OPEN FOR文の構文およびセマンティクス」の詳細は、「OPEN FOR文」を参照してください
- CLOSE文の構文およびセマンティクスの詳細は、「CLOSE文」を参照してください

6.4.3 カーソル変数によるデータのフェッチ

カーソル変数をオープンしたら、FETCH文を使用して、問合せ結果セットの行をフェッチできます。

カーソル変数の戻り型は、FETCH文のinto_clauseと互換性を持つ必要があります。カーソル変数が強い型の場合、PL/SQLはコンパイル時に互換性がないことを捕捉します。カーソル変数が弱い型の場合、PL/SQLは実行時に互換性がないことを捕捉し、最初のフェッチの前に事前定義の例外ROWTYPE_MISMATCHを呼び出します。

関連項目:

- 「明示カーソルによるデータのフェッチ」
- FETCH文の構文およびセマンティクスの詳細は、「FETCH文」を参照してください
- 一度に複数の行を戻すFETCH文の詳細は、「BULK COLLECT句を使用したFETCH文」を参照してください

例6-26 カーソル変数によるデータのフェッチ

この例では、例6-6で2つの明示カーソルを使用して処理していたことを、1つのカーソル変数を使用して処理しています。最初のOPEN FOR文には、問合せ自体が含まれます。2番目のOPEN FOR文は、その値が問合せである変数を参照します。

```
DECLARE
```

```
  cv SYS_REFCURSOR;  -- cursor variable

  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  query_2 VARCHAR2(200) :=
    'SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, ''[ACADFIMKSA]_M[ANGR]'')
    ORDER BY job_id';

  v_employees employees%ROWTYPE;  -- record variable row of table

BEGIN
  OPEN cv FOR
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  LOOP  -- Fetches 2 columns into variables
    FETCH cv INTO v_lastname, v_jobid;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( '-------------------------------------' );

  OPEN cv FOR query_2;

  LOOP  -- Fetches entire row into the v_employees record
    FETCH cv INTO v_employees;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                              v_employees.job_id );
  END LOOP;

  CLOSE cv;
END;
/
```

結果:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-------------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
Zlotkey                   SA_MAN
```

例6-27 カーソル変数からコレクションへのフェッチ

この例では、FETCH文のBULK COLLECT句を使用して、カーソル変数から2つのコレクション(ネストした表)にフェッチします。

```
DECLARE
```

```
  TYPE empcurtyp IS REF CURSOR;
  TYPE namelist IS TABLE OF employees.last_name%TYPE;
  TYPE sallist IS TABLE OF employees.salary%TYPE;
  emp_cv  empcurtyp;
  names   namelist;
  sals    sallist;
BEGIN
  OPEN emp_cv FOR
    SELECT last_name, salary FROM employees
    WHERE job_id = 'SA_REP'
    ORDER BY salary DESC;
  FETCH emp_cv BULK COLLECT INTO names, sals;
  CLOSE emp_cv;
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

結果:

```
Name = Ozer, salary = 11500
Name = Abel, salary = 11000
Name = Vishney, salary = 10500
...
Name = Kumar, salary = 6100
```

6.4.4 カーソル変数への値の代入

PL/SQLカーソル変数に、別のPL/SQLカーソル変数またはホスト・カーソル変数の値を代入できます。

構文は次のとおりです。

```
target_cursor_variable := source_cursor_variable;
```

source_cursor_variableがオープンしている場合、代入後にtarget_cursor_variableもオープンします。2つのカーソル変数は、同じSQL作業領域を指します。

source_cursor_variableがオープンしていない場合、代入後にtarget_cursor_variableがオープンしても、source_cursor_variableはオープンしません。

6.4.5 カーソル変数問合せの変数

カーソル変数に関連付けられた問合せでは、有効範囲内にある任意の変数を参照できます。

OPEN FOR文を使用してカーソル変数をオープンすると、PL/SQLによって問合せ内のすべての変数が評価され、結果セットの識別時にそれらの値が使用されます。変数の値を後で変更しても、結果セットは変更されません。

結果セットを変更するには、例6-29に示すとおり、変数の値を変更してから同じ問合せでカーソル変数を再度オープンする必要があります。

例6-28 カーソル変数問合せの変数(結果セットの変更なし)

この例では、値2を持つ変数factorを参照する問合せのカーソル変数をオープンしています。したがって、sal_multipleは、各フェッチ後にfactorが増分されても、常にsalの2倍になります。

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE cv;
END;
/
```

結果:

```
factor = 2
sal          = 4400
sal_multiple = 8800
factor = 3
sal          = 24000
sal_multiple = 48000
factor = 4
sal          = 17000
sal_multiple = 34000
factor = 5
sal          = 17000
sal_multiple = 34000
```

例6-29 カーソル変数問合せの変数(結果セットの変更あり)

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
```

```
  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  CLOSE cv;
END;
/
```

結果:

```
factor = 2
sal          = 4400
sal_multiple = 8800
sal          = 24000
sal_multiple = 48000
sal          = 17000
sal_multiple = 34000
sal          = 17000
sal_multiple = 34000
factor = 3
sal          = 4400
sal_multiple = 13200
sal          = 24000
sal_multiple = 72000
sal          = 17000
sal_multiple = 51000
sal          = 17000
sal_multiple = 51000
```

6.4.6 コレクションの問合せ

次のすべてが該当する場合、コレクションを問い合せることができます。

- コレクションのデータ型が、スキーマ・レベルで作成されているか、またはパッケージ仕様部で宣言されている。
- コレクション要素のデータ型が、スカラー・データ型、ユーザー定義のデータ型またはレコード型のいずれかである。

問合せのFROM句では、コレクションはtable_collection_expressionにTABLE演算子の引数として指定されます。

> ノート:
>
> SQL コンテキストでは、戻り型がパッケージ仕様部で宣言されているファンクションを使用できません。

関連項目:

- table_collection_expressionの詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- CREATE PACKAGE文の詳細は、「CREATE PACKAGE文」を参照してください
- コレクション型およびコレクション変数の詳細は、「PL/SQLのコレクションおよびレコード」を参照してください
- 例7-9、システム固有の動的SQLを使用したコレクションの問合せ

例6-30 静的SQLを使用したコレクションの問合せ

この例では、カーソル変数がレコードの連想配列の問合せに関連付けられています。ネストした表型mytabは、パッケージ仕様部で宣言されています。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;
DECLARE
  v1 pkg.mytab;  -- collection of records
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
  v1(1).f1 := 1;
  v1(1).f2 := 'one';
  OPEN c1 FOR SELECT * FROM TABLE(v1);
  FETCH c1 INTO v2;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE('Values in record are ' || v2.f1 || ' and ' || v2.f2);
END;
/
```

結果:

```
Values in record are 1 and one
```

6.4.7 カーソル変数の属性

カーソル変数は明示カーソルと同じ属性を持ちます(「明示カーソル属性」を参照)。カーソル変数の属性値の構文は、cursor_variable_nameのすぐ後にattributeが続きます(たとえば、cv%ISOPEN)。カーソル変数をオープンしていない場合、%ISOPEN以外の属性を参照すると、事前定義の例外INVALID_CURSORが呼び出されます。

6.4.8 サブプログラム・パラメータとしてのカーソル変数

カーソル変数は、サブプログラム・パラメータとして使用することができ、この使用方法は、サブプログラム間で問合せ結果を渡す場合に便利です。

たとえば:

- カーソル変数をあるサブプログラムでオープンし、別のサブプログラムで処理できます。
- 多言語アプリケーションでは、PL/SQLサブプログラムは、カーソル変数を使用して、異なる言語で記述されたサブプログラムに結果セットを戻すことができます。

ノート:

起動元と起動先のサブプログラムは、同じデータベース・インスタンス内に存在する必要があります。データベース・リンクを介して起動されたサブプログラムに、カーソル変数を渡したり、戻すことはできません。

注意:

カーソル変数はポインタであるため、カーソル変数をサブプログラム・パラメータとして使用すると、サブプログラム・パラメータのエイリアシングの可能性が高くなり、意図しない結果が発生することがあります。詳細は、「カーソル変数パラメータによるサブプログラムのパラメータのエイリアシング」を参照してください。

カーソル変数をサブプログラムの仮パラメータとして宣言する場合、次のことに注意してください。

- サブプログラムがカーソル変数をオープンするか、カーソル変数に値を代入する場合、パラメータ・モードはIN OUTである必要があります。
- サブプログラムがカーソル変数からフェッチするか、カーソル変数をクローズするのみである場合、パラメータ・モードはINまたはIN OUTのどちらでも可能です。

対応するカーソル変数の仮パラメータと実パラメータの戻り型には、互換性が必要です。それ以外の場合、PL/SQLは事前定義の例外ROWTYPE_MISMATCHを呼び出します。

異なるPL/SQLユニットのサブプログラム間でカーソル変数パラメータを渡すには、パッケージでパラメータのREF CURSOR型を定義します。型がパッケージに含まれる場合、複数のサブプログラムでその型を使用できます。あるサブプログラムでその型の仮パラメータを宣言し、別のサブプログラムでその型の変数を宣言して、それらの変数を最初のサブプログラムに渡すことができます。

関連項目:

- サブプログラムのパラメータの詳細は、「サブプログラムのパラメータ」を参照してください
- 仮カーソル変数パラメータで実パラメータとして使用できるCURSOR式の詳細は、「CURSOR式」を参照してください
- パッケージの詳細は、「PL/SQLパッケージ」を参照してください

例6-31 1つの問合せ用にカーソル変数をオープンするプロシージャ

この例では、REF CURSOR型と、その型のカーソル変数パラメータをオープンするプロシージャをパッケージで定義しています。

```
CREATE OR REPLACE PACKAGE emp_data AUTHID DEFINER AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

例6-32 選択された問合せ用のカーソル変数のオープン(同じ戻り型)

この例では、ストアド・プロシージャが選択された問合せ用にカーソル変数パラメータをオープンします。問合せには、同じ戻り型が含まれます。

```
CREATE OR REPLACE PACKAGE emp_data AUTHID DEFINER AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE commission_pct IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE salary > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE department_id = 100;
    END IF;
  END;
END emp_data;
/
```

例6-33 選択された問合せ用のカーソル変数のオープン(異なる戻り型)

この例では、ストアド・プロシージャが選択された問合せ用にカーソル変数パラメータをオープンします。問合せには、異なる戻り型が含まれます。

```
CREATE OR REPLACE PACKAGE admin_data AUTHID DEFINER AS
  TYPE gencurtyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT);
END admin_data;
/
CREATE OR REPLACE PACKAGE BODY admin_data AS
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT) IS
```

```
  BEGIN
    IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM employees;
    ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM departments;
    ELSIF choice = 3 THEN
      OPEN generic_cv FOR SELECT * FROM jobs;
    END IF;
  END;
END admin_data;
/
```

6.4.9 ホスト変数としてのカーソル変数

カーソル変数は、ホスト変数として使用することができ、この使用方法は、PL/SQLのストアド・サブプログラムとそれらのクライアントとの間で問合せ結果を渡す場合に便利です。

カーソル変数がホスト変数の場合、PL/SQLとクライアント(ホスト環境)は、結果セットを格納するSQL作業領域に対するポインタを共有します。

カーソル変数をホスト変数として使用するには、ホスト環境内でカーソル変数を宣言し、その変数を入力ホスト変数(バインド変数)としてPL/SQLに渡します。ホスト・カーソル変数は、(弱いPL/SQLカーソル変数と同様に)すべての問合せの戻り型と互換性があります。

カーソル変数の値を1つの有効範囲から別の有効範囲に渡したとしても、SQL作業領域は、それを指すカーソル変数が存在するかぎりアクセス可能です。たとえば、例6-34で、Pro*Cプログラムはホスト・カーソル変数を埋込みPL/SQL無名ブロックに渡しています。ブロックの実行後も、カーソル変数はSQL作業領域を指しています。

クライアント側にPL/SQLエンジンがあれば、クライアントからサーバーへのコールに課される制限はありません。たとえば、クライアント側でカーソル変数を宣言し、それをサーバー側でオープンしてフェッチした後で、クライアント側で引き続きフェッチすることができます。また、複数のホスト・カーソル変数を1回の往復でオープンまたはクローズするPL/SQL無名ブロックを使用することで、ネットワークの通信量を削減できます。たとえば:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM employees;
  OPEN :dept_cv FOR SELECT * FROM departments;
  OPEN :loc_cv FOR SELECT * FROM locations;
END;
/
```

PL/SQL無名ブロックの実行後もカーソル変数はSQL作業領域を指しているため、クライアント・プログラムではそれらの作業領域を使用できます。クライアント・プログラムでカーソルが必要なくなった場合、PL/SQL無名ブロックを使用してそれらのカーソルをクローズできます。たとえば:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  CLOSE :emp_cv;
  CLOSE :dept_cv;
  CLOSE :loc_cv;
END;
```

```
/
```

この方法は、Oracle Formsと同様に、マルチブロック・フォームにデータを移入する場合に便利です。たとえば、次のように1回の往復で複数のSQL作業領域をオープンできます。

```
/* PL/SQL anonymous block in host environment */
BEGIN
  OPEN :c1 FOR SELECT 1 FROM DUAL;
  OPEN :c2 FOR SELECT 1 FROM DUAL;
  OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

ノート:

ホスト・カーソル変数を Oracle Call Interface(OCI)クライアントから PL/SQL にバインドすると、同じサーバー・コールで変数をオープンしないかぎり、サーバー側の変数からフェッチすることはできません。

例6-34 Pro*Cクライアント・プログラムにおけるホスト変数としてのカーソル変数

この例では、Pro*Cクライアント・プログラムでカーソル変数と選択子を宣言し、それらをホスト変数としてPL/SQL無名ブロックに渡すことで、選択した問合せ用にカーソル変数をオープンしています。

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL_CURSOR  generic_cv;  -- Declare host cursor variable.
  int         choice;      -- Declare selector.
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :generic_cv;  -- Initialize host cursor variable.
-- Pass host cursor variable and selector to PL/SQL block.
/
EXEC SQL EXECUTE
BEGIN
  IF :choice = 1 THEN
    OPEN :generic_cv FOR SELECT * FROM employees;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM departments;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM jobs;
  END IF;
END;
END-EXEC;
```

6.5 CURSOR式

CURSOR式は、ネストしたカーソルを戻します。

構文は次のとおりです。

```
CURSOR ( subquery )
```

CURSOR式は、副問合せではないSELECT文で使用するか(例6-35のように)、カーソル変数パラメータを受け入れるファンクションに渡すことができます(「パイプライン・テーブル・ファンクションへのCURSOR式の引渡し」を参照)。カーソル式は、暗黙カーソル

とは併用できません。

関連項目:

制限などのCURSOR式の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例6-35 CURSOR式

この例では、カーソル式を含む問合せで明示カーソルを宣言および定義しています。departments表の部門ごとに、ネストしたカーソルはその部門の各従業員の名前を戻します(employees表から取得します)。

```
DECLARE
  TYPE emp_cur_typ IS REF CURSOR;

  emp_cur     emp_cur_typ;
  dept_name   departments.department_name%TYPE;
  emp_name    employees.last_name%TYPE;

  CURSOR c1 IS
    SELECT department_name,
      CURSOR ( SELECT e.last_name
                 FROM employees e
                 WHERE e.department_id = d.department_id
                 ORDER BY e.last_name
               ) employees
    FROM departments d
    WHERE department_name LIKE 'A%'
    ORDER BY department_name;
BEGIN
  OPEN c1;
  LOOP  -- Process each row of query result set
    FETCH c1 INTO dept_name, emp_cur;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);

    LOOP -- Process each row of subquery result set
      FETCH emp_cur INTO emp_name;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('-- Employee: ' || emp_name);
    END LOOP;
  END LOOP;
  CLOSE c1;
END;
/
```

結果:

```
Department: Accounting
-- Employee: Gietz
-- Employee: Higgins
Department: Administration
-- Employee: Whalen
```

6.6 トランザクションの処理および制御

トランザクション処理はOracle Databaseの機能で、複数のユーザーがデータベース上で同時に作業することができ、各ユーザーが参照するデータのバージョンに一貫性があり、すべての変更が正しい順序で適用されることが保証されます。

トランザクションは、Oracle Databaseが1つの単位として処理する連続した1つ以上のSQL文です(これらの文は、すべて実行されるか、1つも実行されないかのいずれかです)。

Oracle Databaseによってデータ構造が自動的にロックされるため、異なるユーザーが、相互のデータに悪影響を与えたりデータを同調させることなく、同じデータ構造に書込みできます。データの可用性を最大にするため、Oracle Databaseでは必要最小限のデータが最小限の時間のみロックされます。

複数のユーザーがデータに同時にアクセスすることによって発生する問題を防ぐために、追加のコードを記述する必要はほとんどありません。ただし、このレベルの制御が必要な場合は、Oracle Databaseのデフォルトのロック・メカニズムを手動で上書きできます。

ここでのトピック

- COMMIT文
- ROLLBACK文
- SAVEPOINT文
- 暗黙的なロールバック
- SET TRANSACTION文
- デフォルトのロックの上書き

関連項目:

- トランザクションの詳細は、『Oracle Database概要』を参照してください。
- トランザクション処理の詳細は、『Oracle Database概要』を参照してください
- Oracle Databaseのロック・メカニズムの詳細は、『Oracle Database概要』を参照してください
- 手動データ・ロックの詳細は、『Oracle Database概要』を参照してください

6.6.1 COMMIT文

COMMIT文は、カレント・トランザクションを終了し、その変更を永続的なものにして、他のユーザーから参照できるようにします。

ノート:

トランザクションは複数のブロックにまたがることができ、ブロックは複数のトランザクションを含むことができます。

COMMIT文のWRITE句では、コミット操作によって生成される情報をOracle DatabaseがREDOログに書き込む場合の優先順

位を指定します。

> ノート:
>
> データベース初期化パラメータ COMMIT_LOGGING および COMMIT_WAIT が設定されていない場合、非分散トランザクションに対する PL/SQL のデフォルトのコミット動作は BATCH NOWAIT です。

関連項目:

- トランザクションのコミットの詳細は、『Oracle Database概要』を参照してください。
- 分散トランザクションの詳細は、『Oracle Database概要』を参照してください。
- COMMIT文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- スタンバイ・データベースへのフェイルオーバー中にデータを消失させないようにする方法については、『Oracle Data Guard概要および管理』を参照してください。

例6-36 COMMENT句およびWRITE句を使用するCOMMIT文

この例では、トランザクションによって銀行口座の間で振替えを実行しています。一方の口座から現金が出金され、同時に他方の口座に現金が入金されることが重要であるため、COMMIT WRITE IMMEDIATE NOWAIT文を使用しています。

```
DROP TABLE accounts;
CREATE TABLE accounts (
  account_id  NUMBER(6),
  balance     NUMBER (10,2)
);

INSERT INTO accounts (account_id, balance)
VALUES (7715, 6350.00);

INSERT INTO accounts (account_id, balance)
VALUES (7720, 5100.50);

CREATE OR REPLACE PROCEDURE transfer (
  from_acct  NUMBER,
  to_acct    NUMBER,
  amount     NUMBER
) AUTHID CURRENT_USER AS
BEGIN
  UPDATE accounts
  SET balance = balance - amount
  WHERE account_id = from_acct;

  UPDATE accounts
  SET balance = balance + amount
  WHERE account_id = to_acct;

  COMMIT WRITE IMMEDIATE NOWAIT;
END;
/
```

振替え前の問合せ:

```
SELECT * FROM accounts;
```

結果:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6350
      7720     5100.5

BEGIN
  transfer(7715, 7720, 250);
END;
/
```

振替え後の問合せ:

```
SELECT * FROM accounts;
```

結果:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6100
      7720     5350.5
```

6.6.2 ROLLBACK文

ROLLBACK文は、カレント・トランザクションを終了し、トランザクション中に加えられたすべての変更を取り消します。

表からの間違った行の削除などの誤操作を行った場合に、ロールバックは元のデータをリストアできます。SQL文が失敗したりPL/SQLが例外を呼び出したためにトランザクションを終了できない場合は、ロールバックを使用すると、対処措置を行い、実行しなおすことができます。

関連項目:

ROLLBACK文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例6-37 ROLLBACK文

この例では、3つの異なる表に従業員に関する情報を挿入しています。INSERT文で重複する従業員番号を格納すると、PL/SQLによって事前定義の例外DUP_VAL_ON_INDEXが呼び出されます。3つの表すべてに対する変更が確実に取り消されるようにするため、例外ハンドラがROLLBACKを実行します。

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name
  FROM employees;

CREATE UNIQUE INDEX empname_ix
```

```
ON emp_name (employee_id);


DROP TABLE emp_sal;
CREATE TABLE emp_sal AS
  SELECT employee_id, salary
  FROM employees;

CREATE UNIQUE INDEX empsal_ix
ON emp_sal (employee_id);


DROP TABLE emp_job;
CREATE TABLE emp_job AS
  SELECT employee_id, job_id
  FROM employees;

CREATE UNIQUE INDEX empjobid_ix
ON emp_job (employee_id);


DECLARE
  emp_id        NUMBER(6);
  emp_lastname  VARCHAR2(25);
  emp_salary    NUMBER(8,2);
  emp_jobid     VARCHAR2(10);
BEGIN
  SELECT employee_id, last_name, salary, job_id
  INTO emp_id, emp_lastname, emp_salary, emp_jobid
  FROM employees
  WHERE employee_id = 120;

  INSERT INTO emp_name (employee_id, last_name)
  VALUES (emp_id, emp_lastname);

  INSERT INTO emp_sal (employee_id, salary)
  VALUES (emp_id, emp_salary);

  INSERT INTO emp_job (employee_id, job_id)
  VALUES (emp_id, emp_jobid);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');
END;
/
```

6.6.3 SAVEPOINT文

SAVEPOINT文は、トランザクション処理内の現在位置に名前とマークを付けます。

セーブポイントを使用すると、トランザクション全体をロールバックするのではなく、トランザクションの一部をロールバックできます。セッションごとのアクティブなセーブポイントの数には、制限がありません。

あるセーブポイントまでロールバックすると、そのセーブポイント以降にマークされたセーブポイントはすべて消去されます。ロールバッ

ク先のセーブポイントは消去されません。単純なロールバックまたはコミットではすべてのセーブポイントが消去されます。

再帰的サブプログラムの中でセーブポイントをマークすると、再帰しながら進む過程で、各レベルでSAVEPOINT文の新しいインスタンスが実行されます。ただし、ロールバックできるのは直前にマークされたセーブポイントまでです。

セーブポイント名は未宣言の識別子です。トランザクションでセーブポイント名を再利用すると、セーブポイントはトランザクションの中の古い位置から現在の位置に移動します(つまり、セーブポイントへのロールバックは、トランザクションの現在の部分のみに影響を与えます)。

関連項目:

SQLのSET TRANSACTION文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例6-38 SAVEPOINT文およびROLLBACK文

この例では、挿入する前にセーブポイントをマークしています。INSERT文でemployee_id列に重複した値を格納しようとすると、PL/SQLによって事前定義の例外DUP_VAL_ON_INDEXが呼び出され、トランザクションがセーブポイントまでロールバックして、INSERT文のみを取り消します。

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT do_insert;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO do_insert;
```

```
  DBMS_OUTPUT.PUT_LINE('Insert was rolled back');
END;
/
```

例6-39 ROLLBACKでのSAVEPOINTの再使用

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  SAVEPOINT my_savepoint;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT my_savepoint;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO my_savepoint;
    DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');
END;
/
```

6.6.4 暗黙的なロールバック

INSERT文、UPDATE文、DELETE文またはMERGE文の実行前に、データベースによって(ユーザーが利用できない)暗黙的なセーブポイントがマークされます。文が正常に実行されなかった場合は、データベースによってこのセーブポイントまでロールバックが行われます。

通常は、トランザクション全体ではなく、失敗したSQL文のみがロールバックされます。その文が原因で未処理例外が呼び出された場合は、ホスト環境によってロールバックの対象が決まります。

データベースでは、デッドロックを解消するためにSQL文を1文のみロールバックすることもできます。データベースによって、関係しているトランザクションにエラーが戻され、そのトランザクション中の現在の文がロールバックされます。

SQL文を実行する前に、データベースはその文を解析する必要があります(すなわち、その文が構文規則に従っているかどうかや、有効なスキーマ・オブジェクトを参照しているかどうかを確認する必要があります)。SQL文の実行中にエラーが検出されると、ロールバックが発生しますが、解析中にエラーが検出されても、ロールバックは発生しません。

ストアド・サブプログラムを未処理例外で終了すると、PL/SQLは値をOUTパラメータに代入せず、ロールバックを実行しません。

例外処理の詳細は、「PL/SQLのエラー処理」を参照してください

6.6.5 SET TRANSACTION文

SET TRANSACTION文を使用すると、読取り専用または読取り/書込みトランザクションの開始、分離レベルの確立、指定したロールバック・セグメントへのカレント・トランザクションの代入を実行できます。

読取り専用トランザクションは、他のユーザーが更新している表に対して複数の問合せを実行する場合に便利です。

読取り専用トランザクションでは、複数の表と複数の問合せで構成された読取り一貫性のあるビューが作成され、すべての問合せがデータベースの同一のスナップショットを参照します。他のユーザーは、通常の方法でデータの問合せや更新ができます。コミットまたはロールバックするとトランザクションが終了します。

SET TRANSACTION文は、読取り専用トランザクションの最初のSQL文である必要があり、1つのトランザクションで1回しか使用できません。トランザクションをREAD ONLYに設定すると、それ以降の問合せからはトランザクションの開始前にコミットされた変更内容しか見えません。READ ONLYを使用しても、他のユーザーや他のトランザクションには影響がありません。

読取り専用トランザクションに使用できるのは、SELECT、OPEN、FETCH、CLOSE、LOCK TABLE、COMMITおよびROLLBACK文のみです。問合せはFOR UPDATEにはできません。

関連項目:

SQLのSET TRANSACTION文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例6-40 読取り専用トランザクションでのSET TRANSACTION文

この例では、読取り専用トランザクションによって、当日、先週および先月の注文総数を収集しています。トランザクションの途中で他のユーザーがデータベースを更新しても、注文総数には影響がありません。orders表は、サンプル・スキーマOEにあります。

```
DECLARE
  daily_order_total     NUMBER(12,2);
  weekly_order_total    NUMBER(12,2);
  monthly_order_total   NUMBER(12,2);
BEGIN
   COMMIT; -- end previous transaction
   SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';
   SELECT SUM (order_total)
   INTO daily_order_total
```

```
   FROM orders
   WHERE order_date = SYSDATE;
   SELECT SUM (order_total)
   INTO weekly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 7;
   SELECT SUM (order_total)
   INTO monthly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 30;
   COMMIT; -- ends read-only transaction
END;
/
```

6.6.6 デフォルトのロックの上書き

デフォルトでは、Oracle Databaseによってデータ構造が自動的にロックされるため、異なるアプリケーションが、相互のデータに悪影響を与えたりデータを同調させることなく、同じデータ構造に書込みできます。

トランザクションの間にデータへの排他アクセスが必要な場合、次のSQL文を使用してデフォルトのロックを上書きできます。

- LOCK TABLEは、表全体を明示的にロックします。
- SELECTをFOR UPDATE句とともに使用すると(SELECT FOR UPDATE)、表の特定の行を明示的にロックできます。

ここでのトピック

- LOCK TABLE文
- SELECT FOR UPDATEカーソルとFOR UPDATEカーソル
- ROWID擬似列を使用したCURRENT OF句のシミュレート

6.6.6.1 LOCK TABLE文

LOCK TABLE文は、1つ以上の表を指定されたロック・モードで明示的にロックするため、それらの表へのアクセスを共有または拒否できます。

ロック・モードによって、その表に設定できる他のロックが決定されます。たとえば、1つの表に対して多くのユーザーが同時に行共用ロックを取得できますが、排他ロックを取得できるのは一度に1人のユーザーのみです。あるユーザーが表に対して排他ロックをかけていると、他のユーザーはその表に対して行の挿入、更新、削除を実行できません。

表がロックされていても、他のユーザーは表に対して問合せできますが、問合せを実行しても表のロックを取得できません。2つの異なるトランザクションが同じ行を変更した場合のみ、一方のトランザクションがもう一方のトランザクションの終了を待ちます。LOCK TABLE文を使用すると、別のトランザクションが完了するまでの待機時間を指定できます。

表ロックは、それらを取得したトランザクションがコミットまたはロールバックされる際に解除されます。

関連項目:

- 表の明示的なロックの詳細は、『Oracle Database開発ガイド』を参照してください。

- LOCK TABLE文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

6.6.6.2 SELECT FOR UPDATEカーソルとFOR UPDATEカーソル

FOR UPDATE句を伴うSELECT文(SELECT FOR UPDATE文)では、結果セットの行が選択され、それらがロックされます。SELECT FOR UPDATEを使用すると、行の中の既存の値に基づいて更新できます(これは、更新前に、他のユーザーがそれらの値を変更できないことが保証されるためです)。また、SELECT FOR UPDATEを使用すると、例9-6のように、更新する必要のない行をロックできます。

ノート:

ハイブリッド列圧縮(HCC)で圧縮された表では、DML 文によって行ではなく圧縮単位がロックされます。特定の Oracle ストレージ・システムの機能である HCC の詳細は、『Oracle Database 概要』を参照してください。

デフォルトでは、要求された行ロックが取得されるまでSELECT FOR UPDATE文は待機します。この動作を変更するには、SELECT FOR UPDATE文でNOWAIT、WAIT、またはSKIP LOCKED句を使用します。このような句の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

SELECT FOR UPDATEが明示カーソルと関連付けられる場合、そのカーソルはFOR UPDATEカーソルと呼ばれます。UPDATE文またはDELETE文のCURRENT OF句には、FOR UPDATEカーソルのみを使用できます。(CURRENT OF句は、SQL文UPDATEおよびDELETEのWHERE句に対するPL/SQLの拡張機能であり、文をカーソルの現在の行に制限します。)

SELECT FOR UPDATEで複数の表の問合せを実行する場合、列がFOR UPDATE句に現れる行のみがロックされます。

6.6.6.3 ROWID擬似列を使用したCURRENT OF句のシミュレート

結果セットの行は、FOR UPDATEカーソルをオープンするときにロックされるのであり、フェッチされるときにロックされるのではありません。また、トランザクションをコミットまたはロールバックすると、行のロックは解除されます。例6-41に示すように、行のロックが解除された後は、FOR UPDATEカーソルからはフェッチできません(ROLLBACKとCOMMITを入れ替えても結果は同じです)。

回避策は、ROWID擬似列を使用してCURRENT OF句をシミュレートすることです(『Oracle Database SQL言語リファレンス』を参照)。各行のROWIDを選択して、UROWID変数に入れ、更新や削除のときに、ROWIDを使用して現在行を識別します(例6-42を参照)。(UROWID変数の値を出力するには、『Oracle Database SQL言語リファレンス』で説明されているROWIDTOCHARファンクションを使用してその値をVARCHAR2に変換します。)

ノート:

ハイブリッド列圧縮(HCC)で圧縮された表の行を更新すると、行の ROWID が変更されます。特定の Oracle ストレージ・システムの機能である HCC の詳細は、『Oracle Database 概要』を参照してください。

注意:

フェッチされた行は、FOR UPDATE 句によってロックされないため、他のユーザーによって意識せずに変更内容が上

書きされる可能性があります。

ノート:

読取り一貫性のために必要な追加領域は、カーソルがクローズされるまで解放されません。そのため、大規模な更新では処理速度が低下する場合があります。

例6-41 COMMIT文の後でのFOR UPDATEカーソルによるフェッチ

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
    SELECT * FROM emp
    FOR UPDATE OF salary
    ORDER BY employee_id;

  emp_rec  emp%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;  -- fails on second iteration
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'emp_rec.employee_id = ' ||
      TO_CHAR(emp_rec.employee_id)
    );

    UPDATE emp
    SET salary = salary * 1.05
    WHERE employee_id = 105;

    COMMIT;  -- releases locks
  END LOOP;
END;
/
```

結果:

```
emp_rec.employee_id = 100
DECLARE
*
ERROR at line 1:
ORA-01002: fetch out of sequence
ORA-06512: at line 11
```

例6-42 ROWID擬似列を使用したCURRENT OF句のシミュレート

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
```

```
    SELECT last_name, job_id, rowid
    FROM emp;  -- no FOR UPDATE clause

  my_lastname   employees.last_name%TYPE;
  my_jobid      employees.job_id%TYPE;
  my_rowid      UROWID;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_lastname, my_jobid, my_rowid;
    EXIT WHEN c1%NOTFOUND;

    UPDATE emp
    SET salary = salary * 1.02
    WHERE rowid = my_rowid;  -- simulates WHERE CURRENT OF c1

    COMMIT;
  END LOOP;
  CLOSE c1;
END;
/
```

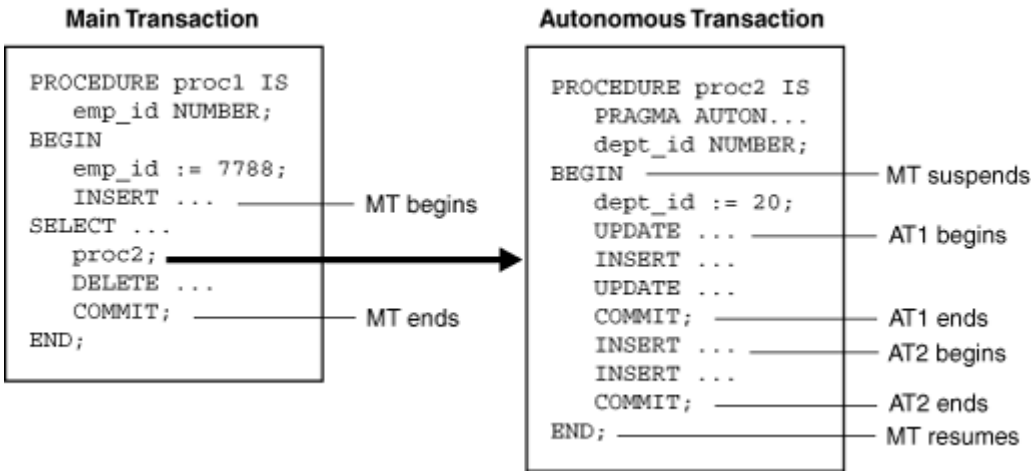
6.7 自律型トランザクション

自律型トランザクションは、別の、メイン・トランザクションによって開始される独立したトランザクションです。

自律型トランザクションは、メイン・トランザクションをコミットまたはロールバックせずに、SQL操作を実行してコミットまたはロールバックします。

図6-1に、メイン・トランザクション(MT)から自律型ルーチン(proc2)へ制御がどのように流れ、また戻るかを示します。自律型ルーチンは、2つの自立型トランザクション(AT1およびAT2)をコミットします。

図6-1 トランザクション制御の流れ



ノート:

自律型トランザクションは別のトランザクションによって開始されますが、これはネストしたトランザクションではありません。その理由は次のとおりです。

- ロックなどのトランザクション・リソースをメイン・トランザクションと共有しません。
- メイン・トランザクションに依存しません。

  たとえば、メイン・トランザクションがロールバックする場合は、ネストしたトランザクションがロールバックするのに対し、自律型トランザクションはロールバックしません。
- コミットされた変更を、他のトランザクションからすぐに参照できます。

  ネストしたトランザクションのコミットされた変更は、メイン・トランザクションがコミットするまで他のトランザクションからは参照できません。
- 自律型トランザクションで例外が呼び出されると、文レベルのロールバックではなくトランザクション・レベルのロールバックが発生します。

ここでのトピック

- 自律型トランザクションのメリット
- トランザクション・コンテキスト
- トランザクションの可視性
- 自律型ルーチンの宣言
- 自律型トランザクションの制御
- 自律型トリガー
- SQLからの自律型ファンクションの起動

関連項目:

自律型トランザクションの詳細は、『Oracle Database開発ガイド』を参照してください。

6.7.1 自律型トランザクションのメリット

自律型トランザクションは、開始すると完全に独立します。ロック、リソースまたはコミット依存関係をメイン・トランザクションと共有することはありません。メイン・トランザクションがロールバックする場合でも、イベントや増分再試行カウンタなどのログを取ることができます。

自律型トランザクションは再利用可能なソフトウェア・コンポーネントであるモジュール構造の作成に役立ちます。自律型トランザクションは、ストアド・サブプログラム内部でカプセル化できます。起動元のアプリケーションは、そのストアド・サブプログラムによって実行された操作が成功したか失敗したかを知る必要はありません。

6.7.2 トランザクション・コンテキスト

メイン・トランザクションはそのコンテキストをネストしたルーチンと共有しますが、自律型トランザクションとは共有しません。ある自律型ルーチンが別の自律型ルーチンを(または自身を再帰的に)起動する場合、ルーチンはトランザクション・コンテキストを共有

しません。ある自律型ルーチンが自律型ではないルーチンを起動する場合、ルーチンは同じトランザクション・コンテキストを共有します。

6.7.3 トランザクションの可視性

自律型トランザクションによって行われた変更は、自律型トランザクションがコミットすると、他のトランザクションから参照できるようになります。分離レベルがREAD COMMITTED(デフォルト)に設定されている場合、メイン・トランザクションが再開すると、これらの変更をメイン・トランザクションから参照できるようになります。

メイン・トランザクションの分離レベルをSERIALIZABLEに設定すると、その自律型トランザクションによって行われた変更は、再開してもメイン・トランザクションからは参照できません。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

ノート:

- トランザクションのプロパティは、それらが設定されたトランザクションにのみ適用されます。
- カーソル属性は自律型トランザクションの影響を受けません。

6.7.4 自律型ルーチンの宣言

自律型ルーチンを宣言するには、AUTONOMOUS_TRANSACTIONプラグマを使用します。

このプラグマの詳細は、「AUTONOMOUS_TRANSACTIONプラグマ」を参照してください。

ヒント:

読みやすくするために、AUTONOMOUS_TRANSACTION プラグマは宣言部の先頭に置きます。(プラグマは宣言部の任意の場所に置くことができます。)

AUTONOMOUS_TRANSACTIONプラグマはパッケージ全体またはADT全体に適用することはできませんが、パッケージ内の各サブプログラムまたはADTの各メソッドに適用できます。

例6-43 パッケージでの自律型ファンクションの宣言

この例では、パッケージ・ファンクションを自律型としてマークします。

```
CREATE OR REPLACE PACKAGE emp_actions AUTHID DEFINER AS  -- package specification
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER;
END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- package body
  -- code for function raise_salary
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER IS
```

```
    PRAGMA AUTONOMOUS_TRANSACTION;
    new_sal NUMBER(8,2);
  BEGIN
    UPDATE employees SET salary =
      salary + sal_raise WHERE employee_id = emp_id;
    COMMIT;
    SELECT salary INTO new_sal FROM employees
      WHERE employee_id = emp_id;
    RETURN new_sal;
  END raise_salary;
END emp_actions;
/
```

例6-44 自律型スタンドアロン・プロシージャの宣言

この例では、スタンドアロン・サブプログラムを自律型としてマークします。

```
CREATE OR REPLACE PROCEDURE lower_salary
  (emp_id NUMBER, amount NUMBER)
AUTHID DEFINER AS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;
  COMMIT;
END lower_salary;
/
```

例6-45 自律型PL/SQLブロックの宣言

この例では、スキーマ・レベルのPL/SQLブロックを自律型としてマークします。(ネストしたPL/SQLブロックは自律型にはできません。)

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  emp_id NUMBER(6)   := 200;
  amount NUMBER(6,2) := 200;
BEGIN
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;

  COMMIT;
END;
/
```

6.7.5 自律型トランザクションの制御

自律型ルーチンの最初のSQL文でトランザクションが開始されます。1つのトランザクションが終了すると、次のSQL文で別のトランザクションが開始されます。カレント・トランザクションは、最後のコミットまたはロールバックよりも後に実行されたすべてのSQL文で構成されます。自律型トランザクションを制御するには、次の文を使用します(これは現在の(アクティブな)トランザクションのみ

に適用されます)。

- COMMIT
- ROLLBACK [TO savepoint_name]
- SAVEPOINT savepoint_name
- SET TRANSACTION

ここでのトピック

- 自律型ルーチンの開始と終了
- 自律型トランザクションのコミットおよびロールバック
- セーブポイント
- 自律型トランザクションでのエラーの回避

6.7.5.1 自律型ルーチンの開始と終了

自律型ルーチンの実行部に入ると、メイン・トランザクションは停止します。ルーチンを終了すると、メイン・トランザクションは再開します。

コミットまたはロールバックせずにアクティブな自律型トランザクションを終了しようとすると、データベースによって例外が呼び出されます。例外が未処理になった場合、またはその他の未処理例外が発生してトランザクションが終了した場合、トランザクションはロールバックします。

正常に終了するには、ルーチンはすべての自律型トランザクションを明示的にコミットまたはロールバックする必要があります。ルーチン(またはそれによって起動されたルーチン)に保留中のトランザクションがある場合は、PL/SQLによって例外が呼び出され、保留中のトランザクションはロールバックします。

6.7.5.2 自律型トランザクションのコミットおよびロールバック

COMMITとROLLBACKはアクティブな自律型トランザクションを終了しますが、自律型ルーチンから抜けるわけではありません。1つのトランザクションが終了すると、次のSQL文で別のトランザクションが開始されます。1つの自律型ルーチンは、複数のCOMMIT文を発行する場合、複数の自律型トランザクションを含むことができます。

6.7.5.3 セーブポイント

セーブポイントの有効範囲は、それが定義されたトランザクションです。メイン・トランザクション内で定義されたセーブポイントは、その自律型トランザクション内で定義されたセーブポイントとは無関係です。実際、メイン・トランザクションと自律型トランザクションのセーブポイントには、同じ名前を使用できます。

ロールバックできるのは、カレント・トランザクション内でマークされたセーブポイントまでです。自律型トランザクション内では、メイン・トランザクション内でマークされたセーブポイントまではロールバックできません。メイン・トランザクションのセーブポイントまでロールバックするには、自律型ルーチンを抜けてメイン・トランザクションを再開する必要があります。

メイン・トランザクション内では、自律型トランザクションを開始する前にマークされたセーブポイントまでロールバックしても、自律型トランザクションはロールバックされません。自律型トランザクションは、メイン・トランザクションからは完全に独立していることに注意してください。

6.7.5.4 自律型トランザクションでのエラーの回避

一般的なエラーを回避するには、次のことに注意してください。

- メイン・トランザクションが保持するリソースに、自律型トランザクションがアクセスしようとすると、デッドロックが発生することがあります。この場合は、データベースによって自律型トランザクションで例外が呼び出され、例外が未処理になった場合、自律型トランザクションはロールバックします。
- データベース初期化パラメータTRANSACTIONSは、同時トランザクションの最大数を指定します。自律型トランザクションはメイン・トランザクションと同時に実行されるため、この最大数を超える場合があります。
- コミットまたはロールバックせずにアクティブな自律型トランザクションを終了しようとすると、データベースによって例外が呼び出されます。例外が未処理になった場合、トランザクションはロールバックします。
- 自律型トランザクションがオープンされている間は、自律型ルーチン内でPIPE ROW文を実行することはできません。PIPE ROW文を実行する前に、自律型トランザクションをクローズする必要があります。この処理は通常、PIPE ROW文を実行する前に自律型トランザクションをコミットまたはロールバックすることによって行います。

6.7.6 自律型トリガー

TCL文またはDDL文を実行する場合、トリガーは自律型である必要があります。

DDL文を実行する場合、トリガーは、システム固有の動的SQLを使用する必要があります。

関連項目:

- トリガーの一般情報は、「PL/SQLトリガー」を参照してください
- TCL文の一般情報は、「静的SQLの説明」を参照してください
- DDL文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- システム固有の動的SQLの詳細は、「システム固有の動的SQL」を参照してください

トリガーの使用例の1つは、イベントのログを透過的に取ることです(たとえば、ある表に対するすべての挿入を、ロールバックしたものも含めてログに記録できます)。

例6-46 自律型トリガーによるINSERT文のログへの記録

この例では、EMPLOYEES表に行が挿入されるたびに、トリガーによって同じ行がログ表に挿入されます。トリガーは自律型であるため、メイン表への変更をコミットするかどうかに関係なく、ログ表への変更をコミットできます。

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

-- Log table:

DROP TABLE log;
CREATE TABLE log (
  log_id   NUMBER(6),
  up_date  DATE,
  new_sal  NUMBER(8,2),
  old_sal  NUMBER(8,2)
);
```

```
-- Autonomous trigger on emp table:

CREATE OR REPLACE TRIGGER log_sal
  BEFORE UPDATE OF salary ON emp FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO log (
    log_id,
    up_date,
    new_sal,
    old_sal
  )
  VALUES (
    :old.employee_id,
    SYSDATE,
    :new.salary,
    :old.salary
  );
  COMMIT;
END;
/
UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 115;

COMMIT;

UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 116;

ROLLBACK;

-- Show that both committed and rolled-back updates
-- add rows to log table

SELECT * FROM log
WHERE log_id = 115 OR log_id = 116;
```

結果:

```
    LOG_ID UP_DATE      NEW_SAL    OLD_SAL
---------- --------- ---------- ----------
       115 02-OCT-12    3255          3100
       116 02-OCT-12    3045          2900

2 rows selected.
```

例6-47 自律型トリガーによるDDLに対するシステム固有の動的SQLの使用

この例では、自律型トリガーでシステム固有の動的SQL(EXECUTE IMMEDIATE文)を使用し、行が表logに挿入された後で一時表を削除しています。

```
DROP TABLE temp;
CREATE TABLE temp (
```

```
  temp_id NUMBER(6),
  up_date DATE
);
CREATE OR REPLACE TRIGGER drop_temp_table
  AFTER INSERT ON log
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE temp';
  COMMIT;
END;
/
-- Show how trigger works
SELECT * FROM temp;
```

結果:

```
no rows selected
INSERT INTO log (log_id, up_date, new_sal, old_sal)
VALUES (999, SYSDATE, 5000, 4500);

1 row created.

SELECT * FROM temp;
```

結果:

```
SELECT * FROM temp
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

6.7.7 SQLからの自律型ファンクションの起動

SQL文から呼び出されるファンクションは、副作用を制御するための規則に従う必要があります。

定義上は、自立型ルーチンによってデータベース状態の読取りおよび書込みは行われません(つまり、データベース表に対する問合せや変更が行われません)。

関連項目:

詳細は、「サブプログラムの副作用」を参照

例6-48 自律型ファンクションの起動

パッケージ・ファンクションlog_msgは自律型です。ファンクションが起動されると、データベース状態の書込み(データベース表の変更)禁止の規則に違反することなく、そのファンクションによってデータベース表debug_outputにメッセージが挿入されます。

```
DROP TABLE debug_output;
CREATE TABLE debug_output (message VARCHAR2(200));
```

```
CREATE OR REPLACE PACKAGE debugging AUTHID DEFINER AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
END debugging;
/
CREATE OR REPLACE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    INSERT INTO debug_output (message) VALUES (msg);
    COMMIT;
    RETURN msg;
  END;
END debugging;
/
-- Invoke package function from query
DECLARE
  my_emp_id    NUMBER(6);
  my_last_name VARCHAR2(25);
  my_count     NUMBER;
BEGIN
  my_emp_id := 120;

  SELECT debugging.log_msg(last_name)
  INTO my_last_name
  FROM employees
  WHERE employee_id = my_emp_id;

  /* Even if you roll back in this scope,
     the insert into 'debug_output' remains committed,
     because it is part of an autonomous transaction. */

  ROLLBACK;
END;
/
```

7 PL/SQLの動的SQL

動的SQLは、実行時にSQL文を生成して実行するためのプログラミング方法です。

この方法は、非定型の問合せシステムのような柔軟性がある汎用目的のプログラムを記述する場合、データベース定義言語(DDL)文を実行する必要があるプログラムを記述する場合、またはコンパイル時にSQL文のテキスト全体またはそのSQL文の入力変数および出力変数の数またはデータ型が不明な場合に有効です。

PL/SQLには、動的SQLを記述する次の2つの方法が用意されています。

- 動的SQL文を作成および実行するためのPL/SQL言語(つまり、システム固有の)機能であるシステム固有の動的SQL。
- 動的SQL文を作成、実行および記述するためのAPIであるDBMS_SQLパッケージ。

システム固有の動的SQLコードの方が、DBMS_SQLパッケージを使用する同等のコードより読み書きが簡単であり、大幅に高速で実行されます(特に、コンパイラによって最適化できる場合)。ただし、システム固有の動的SQLコードを記述するには、コンパイル時に動的SQL文の入力変数および出力変数の数およびデータ型がわかっている必要があります。コンパイル時にこの情報が不明な場合は、DBMS_SQLパッケージを使用する必要があります。また、ストアド・サブプログラムが問合せ結果を暗黙的に(OUT REF CURSORパラメータを使用するのではなく)戻すようにする場合にも、DBMS_SQLパッケージを使用する必要があります。

DBMS_SQLパッケージとシステム固有の動的SQLの両方が必要な場合は、「DBMS_SQL.TO_REFCURSORファンクション」と「DBMS_SQL.TO_CURSOR_NUMBERファンクション」を使用して、それらを切り替えることができます。

ここでのトピック

- 動的SQLが必要な場合
- システム固有の動的SQL
- DBMS_SQLパッケージ
- SQLインジェクション

7.1 動的SQLが必要な場合

PL/SQLでは、次のSQLを実行するために動的SQLが必要となります。

- コンパイル時に不明なテキストが含まれているSQL

  たとえば、コンパイル時に不明な識別子(表名など)が含まれているSELECT文、コンパイル時に副次句の数が不明なWHERE句などです。

- 静的SQLとしてサポートされていないSQL

  つまり、「静的SQLの説明」で示されていないSQL構文のことです。

動的SQLが必要ない場合は、静的SQLを使用します。静的SQLには、次のようなメリットがあります。

- コンパイルが正常に完了すると、静的SQL文が有効なデータベース・オブジェクトを参照していること、およびそれらのオブジェクトへのアクセスに必要な権限が存在していることが保証されます。
- コンパイルが正常に完了すると、スキーマ・オブジェクトの依存性が作成されます。

スキーマ・オブジェクトの依存性の詳細は、『Oracle Database開発ガイド』を参照してください。

PL/SQLでの静的SQL文の使用方法の詳細は、「PL/SQLの静的SQL」を参照してください。

7.2 システム固有の動的SQL

システム固有の動的SQLでは、EXECUTE IMMEDIATE文を使用してほとんどの動的SQL文を処理します。

動的SQL文が複数行を戻すSELECT文である場合、システム固有の動的SQLでは、次のいずれかを選択できます。

- BULK COLLECT INTO句を指定してEXECUTE IMMEDIATE文を使用します。
- OPEN FOR文、FETCH文およびCLOSE文を使用します。

SQLのカーソル属性は、システム固有の動的SQLのINSERT文、UPDATE文、DELETE文、MERGE文および単一行のSELECT文の後では、それらの文が静的SQL文に対して動作する場合と同様に動作します。SQLのカーソル属性の詳細は、「カーソルの概要」を参照してください。

ここでのトピック

- EXECUTE IMMEDIATE文
- OPEN FOR文、FETCH文およびCLOSE文
- 動的SQL文内でのプレースホルダ名の繰返し

7.2.1 EXECUTE IMMEDIATE文

EXECUTE IMMEDIATE文は、システム固有の動的SQLがほとんどの動的SQL文を処理する際に使用する手段です。

動的SQL文が自己完結型である場合(つまり、バインド変数のプレースホルダがなく、戻すことができる結果がエラーのみである場合)、EXECUTE IMMEDIATE文に句は不要です。

動的SQL文にバインド変数のプレースホルダが含まれている場合、各プレースホルダは、次のように、EXECUTE IMMEDIATE文の適切な句内に対応するバインド変数を持っている必要があります。

- 動的SQL文が最大で1行を戻すことができるSELECT文である場合、アウトバインド変数(定義)をINTO句に、インバインド変数をUSING句に含めます。
- 動的SQL文が複数行を戻すことができるSELECT文である場合、アウトバインド変数(定義)をBULK COLLECT INTO句に、インバインド変数をUSING句に含めます。
- 動的SQL文が、RETURNING INTO句が指定されていないDML文(SELECT以外)である場合、すべてのバインド変数をUSING句に含めます。
- 動的SQL文がRETURNING INTO句が指定されているDML文である場合、インバインド変数をUSING句に、アウトバインド変数をRETURNING INTO句に含めます。
- 動的SQL文が無名PL/SQLブロックまたはCALL文である場合、すべてのバインド変数をUSING句に含めます。

  動的SQL文がサブプログラムを起動する場合は、次のことを確認します。

  - サブプログラムが、スキーマ・レベルで作成されているか、またはパッケージ仕様部で宣言および定義されていること。

- サブプログラム・パラメータのプレースホルダに対応しているすべてのバインド変数のパラメータ・モードがそのサブプログラム・パラメータと同じであること、およびこのバインド変数のデータ型がそのサブプログラム・パラメータのデータ型と互換性があること。
- いずれのバインド変数も予約語NULLではないこと。

  この制限を回避するには、例7-7に示すように、NULLを使用する場所に未初期化変数を使用します。
- いずれのバインド変数も、SQLでサポートされないデータ型(文字列により索引付けされる連想配列など)を持たないこと。

  データ型がコレクションまたはレコード型である場合は、パッケージ仕様部で宣言されている必要があります。

ノート:

バインド変数はどのような順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

例7-4、例7-5および例7-6では、動的PL/SQLブロックは、PL/SQLコレクション型の仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。コレクション型はSQLデータ型ではありません。各例では、コレクション型はパッケージ仕様部で宣言され、サブプログラムはパッケージ仕様部で宣言され、パッケージ本体で定義されます。

関連項目:

- スキーマ・レベルでのファンクション作成の詳細は、「CREATE FUNCTION文」を参照してください
- スキーマ・レベルでのプロシージャ作成の詳細は、「CREATE PROCEDURE文」を参照してください
- パッケージの詳細は、「PL/SQLパッケージ」を参照してください
- パッケージ内のサブプログラムの宣言の詳細は、「CREATE PACKAGE文」を参照してください
- パッケージ内のサブプログラムの宣言および定義の詳細は、「CREATE PACKAGE BODY文」を参照してください
- パッケージ仕様での型の宣言の詳細は、「CREATE PACKAGE文」を参照してください
- EXECUTE IMMEDIATE文の構文の詳細は、「EXECUTE IMMEDIATE文」を参照してください
- コレクション型の詳細は、「PL/SQLのコレクションおよびレコード」を参照してください

例7-1 動的PL/SQLブロックからのサブプログラムの起動

この例では、動的PL/SQLブロックは、スキーマ・レベルで作成されたサブプログラムを起動する無名PL/SQLブロックです。

```
-- Subprogram that dynamic PL/SQL block invokes:
CREATE OR REPLACE PROCEDURE create_dept (
  deptid IN OUT NUMBER,
  dname  IN     VARCHAR2,
  mgrid  IN     NUMBER,
  locid  IN     NUMBER
) AUTHID DEFINER AS
BEGIN
  deptid := departments_seq.NEXTVAL;
  INSERT INTO departments (
```

```
    department_id,
    department_name,
    manager_id,
    location_id
  )
  VALUES (deptid, dname, mgrid, locid);
END;
/
DECLARE
  plsql_block VARCHAR2(500);
  new_deptid  NUMBER(4);
  new_dname   VARCHAR2(30) := 'Advertising';
  new_mgrid   NUMBER(6)    := 200;
  new_locid   NUMBER(4)    := 1700;
BEGIN
 -- Dynamic PL/SQL block invokes subprogram:
  plsql_block := 'BEGIN create_dept(:a, :b, :c, :d); END;';
 /* Specify bind variables in USING clause.
    Specify mode for first parameter.
    Modes of other parameters are correct by default. */
  EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptid, new_dname, new_mgrid, new_locid;
END;
/
```

例7-2 BOOLEAN仮パラメータを持つサブプログラムの動的起動

この例では、動的PL/SQLブロックは、PL/SQL(SQLではなく)データ型BOOLEANの仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。

```
CREATE OR REPLACE PROCEDURE p (x BOOLEAN) AUTHID DEFINER AS
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('x is true');
  END IF;
END;
/
DECLARE
  dyn_stmt VARCHAR2(200);
  b        BOOLEAN := TRUE;
BEGIN
  dyn_stmt := 'BEGIN p(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING b;
END;
/
```

結果:

```
x is true
```

例7-3 RECORD仮パラメータを持つサブプログラムの動的起動

この例では、動的PL/SQLブロックは、PL/SQL(SQLではなく)データ型RECORDの仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。レコード型はパッケージ仕様部で宣言され、サブプログラムはパッケージ仕様部で宣言され、パッケージ本体で定義されます。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

  TYPE rec IS RECORD (n1 NUMBER, n2 NUMBER);

  PROCEDURE p (x OUT rec, y NUMBER, z NUMBER);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS

  PROCEDURE p (x OUT rec, y NUMBER, z NUMBER) AS
  BEGIN
    x.n1 := y;
    x.n2 := z;
  END p;
END pkg;
/
DECLARE
  r       pkg.rec;
  dyn_str VARCHAR2(3000);
BEGIN
  dyn_str := 'BEGIN pkg.p(:x, 6, 8); END;';

  EXECUTE IMMEDIATE dyn_str USING OUT r;

  DBMS_OUTPUT.PUT_LINE('r.n1 = ' || r.n1);
  DBMS_OUTPUT.PUT_LINE('r.n2 = ' || r.n2);
END;
/
```

例7-4 連想を持つサブプログラムの動的起動配列仮パラメータ

この例では、動的PL/SQLブロックは、PLS_INTEGERで索引付けされる連想配列のPL/SQLコレクション型の仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。

ノート:

このコンテキストで使用される連想配列は、PLS_INTEGER を使用して索引付けする必要があります。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
```

TYPE number_names IS TABLE OF VARCHAR2(5) INDEX BY PLS_INTEGER; PROCEDURE print_number_names (x number_names); END pkg; / CREATE OR REPLACE PACKAGE BODY pkg AS PROCEDURE print_number_names (x number_names) IS BEGIN FOR i IN x.FIRST .. x.LAST LOOP DBMS_OUTPUT.PUT_LINE(x(i)); END LOOP; END; END pkg; / DECLARE digit_names pkg.number_names; dyn_stmt VARCHAR2(3000); BEGIN digit_names(0) := 'zero'; digit_names(1) := 'one'; digit_names(2) := 'two'; digit_names(3) := 'three'; digit_names(4) := 'four'; digit_names(5) := 'five'; digit_names(6) := 'six'; digit_names(7) := 'seven'; digit_names(8) := 'eight'; digit_names(9) := 'nine'; dyn_stmt := 'BEGIN pkg.print_number_names(:x); END;'; EXECUTE IMMEDIATE dyn_stmt USING digit_names; END; /

例7-5 ネストした表仮パラメータを持つサブプログラムの動的起動

この例では、動的PL/SQLブロックは、ネストした表のPL/SQLコレクション型の仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

  TYPE names IS TABLE OF VARCHAR2(10);

  PROCEDURE print_names (x names);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_names (x names) IS
  BEGIN
    FOR i IN x.FIRST .. x.LAST LOOP
      DBMS_OUTPUT.PUT_LINE(x(i));
    END LOOP;
  END;
END pkg;
/
DECLARE
  fruits   pkg.names;
  dyn_stmt VARCHAR2(3000);
BEGIN
  fruits := pkg.names('apple', 'banana', 'cherry');

  dyn_stmt := 'BEGIN pkg.print_names(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING fruits;
END;
/
```

例7-6 VARRAY仮パラメータを持つサブプログラムの動的起動

この例では、動的PL/SQLブロックは、PL/SQLコレクション型VARRAYの仮パラメータを持つサブプログラムを起動する無名PL/SQLブロックです。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

  TYPE foursome IS VARRAY(4) OF VARCHAR2(5);

  PROCEDURE print_foursome (x foursome);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_foursome (x foursome) IS
  BEGIN
    IF x.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Empty');
    ELSE
      FOR i IN x.FIRST .. x.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(x(i));
      END LOOP;
    END IF;
  END;
END pkg;
/
DECLARE
  directions pkg.foursome;
```

```
  dyn_stmt VARCHAR2(3000);
BEGIN
  directions := pkg.foursome('north', 'south', 'east', 'west');

  dyn_stmt := 'BEGIN pkg.print_foursome(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING directions;
END;
/
```

例7-7 USING句でのNULLを表現する未初期化変数

この例では、未初期化変数を使用して、USING句内で予約語NULLを表現しています。

```
CREATE TABLE employees_temp AS SELECT * FROM EMPLOYEES;
DECLARE
  a_null  CHAR(1);  -- Set to NULL automatically at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE employees_temp SET commission_pct = :x'
    USING a_null;
END;
/
```

7.2.2 OPEN FOR文、FETCH文およびCLOSE文

動的SQL文が複数行を戻すSELECT文である場合は、システム固有の動的SQLを使用してその文を次のように処理できます。

1. OPEN FOR文を使用して、カーソル変数を動的SQL文に関連付けます。OPEN FOR文のUSING句で、動的SQL文内の各プレースホルダにバインド変数を指定します。

   USING句に、リテラルNULLを含めることはできません。この制限を回避するには、例7-7に示すように、NULLを使用する場所に未初期化変数を使用します。

2. FETCH文を使用して、結果セットの行を一度に1行、複数行またはすべて取り出します。

3. CLOSE文を使用して、カーソル変数をクローズします。

コレクションが「コレクションの問合せ」に示す条件を満たしていれば、動的SQL文でコレクションを問い合せることができます。

関連項目:

- 構文の詳細は、「OPEN FOR文」を参照してください
- 構文の詳細は、「FETCH文」を参照してください
- 構文の詳細は、「CLOSE文」を参照してください

例7-8 OPEN FOR文、FETCH文およびCLOSE文を使用したシステム固有の動的SQL

この例では、結果セットの行を一度に1行ずつ取り出して、マネージャであるすべての従業員を表示しています。

```
DECLARE
  TYPE EmpCurTyp  IS REF CURSOR;
  v_emp_cursor    EmpCurTyp;
  emp_record      employees%ROWTYPE;
  v_stmt_str      VARCHAR2(200);
  v_e_job         employees.job%TYPE;
```

```
BEGIN
  -- Dynamic SQL statement with placeholder:
  v_stmt_str := 'SELECT * FROM employees WHERE job_id = :j';
  -- Open cursor & specify bind variable in USING clause:
  OPEN v_emp_cursor FOR v_stmt_str USING 'MANAGER';
  -- Fetch rows from result set one at a time:
  LOOP
    FETCH v_emp_cursor INTO emp_record;
    EXIT WHEN v_emp_cursor%NOTFOUND;
  END LOOP;
  -- Close cursor:
  CLOSE v_emp_cursor;
END;
/
```

例7-9 システム固有の動的SQLを使用したコレクションの問合せ

この例は例6-30と似ていますが、コレクション変数v1がバインド変数である点が異なります。

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;
/
DECLARE
  v1 pkg.mytab;  -- collection of records
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
  OPEN c1 FOR 'SELECT * FROM TABLE(:1)' USING v1;
  FETCH c1 INTO v2;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE('Values in record are ' || v2.f1 || ' and ' || v2.f2);
END;
/
```

7.2.3 動的SQL文内でのプレースホルダ名の繰返し

動的SQL文内でプレースホルダ名を繰り返す場合は、プレースホルダをバインド変数に関連付ける方法が動的SQL文の種類によって異なるということに注意してください。

ここでのトピック

- 動的SQL文が無名ブロックまたはCALL文でない
- 動的SQL文が無名ブロックまたはCALL文

7.2.3.1 動的SQL文が無名ブロックまたはCALL文でない

動的SQL文が無名PL/SQLブロックまたはCALL文を表していない場合、プレースホルダ名の繰返しは重要ではありません。

プレースホルダは、名前ではなく位置によって、USING句内のバインド変数に関連付けられます。

たとえば、次の動的SQL文では、:xという名前の繰返しは重要ではありません。

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

対応するUSING句では、4つのバインド変数を指定する必要があります。それらは異なっていてもかまいません。たとえば:

```
EXECUTE IMMEDIATE sql_stmt USING a, b, c, d;
```

前述のEXECUTE IMMEDIATE文は、次のSQL文を実行します。

```
INSERT INTO payroll VALUES (a, b, c, d)
```

:xが出現するたびに、それぞれに同じバインド変数を関連付けるには、そのバインド変数を繰り返す必要があります。たとえば:

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

前述のEXECUTE IMMEDIATE文は、次のSQL文を実行します。

```
INSERT INTO payroll VALUES (a, a, b, a)
```

7.2.3.2 動的SQL文が無名ブロックまたはCALL文

動的SQL文が無名PL/SQLブロックまたはCALL文を表している場合、プレースホルダ名の繰返しは重要です。

一意のプレースホルダ名が、それぞれUSING句内に対応するバインド変数を持っている必要があります。プレースホルダ名を繰り返す場合、それに対応するバインド変数を繰り返す必要はありません。そのプレースホルダ名に対するすべての参照がUSING句内の1つのバインド変数に対応します。

例7-10 動的PL/SQLブロックで繰り返されるプレースホルダ名

この例では、最初の一意のプレースホルダ名:xに対するすべての参照がUSING句内の最初のバインド変数aに関連付けられており、2番目の一意のプレースホルダ名:yがUSING句内の2番目のバインド変数bに関連付けられています。

```
CREATE PROCEDURE calc_stats (
  w NUMBER,
  x NUMBER,
  y NUMBER,
  z NUMBER )
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);
END;
/
DECLARE
  a NUMBER := 4;
  b NUMBER := 7;
  plsql_block VARCHAR2(100);
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
  EXECUTE IMMEDIATE plsql_block USING a, b;  -- calc_stats(a, a, b, a)
END;
/
```

7.3 DBMS_SQLパッケージ

DBMS_SQLパッケージは、SQLカーソル番号と呼ばれるエンティティを定義します。SQLカーソル番号は、PL/SQLの整数であるため、コール境界を越えて渡し、格納することができます。

次のいずれかが該当する場合は、DBMS_SQLパッケージを使用して、動的SQL文を実行する必要があります。

- 実行時までSELECTリストが不明。
- 実行時まで、SELECTまたはDML文でバインドする必要があるプレースホルダが不明。
- ストアド・サブプログラムが問合せ結果を暗黙的に(OUT REF CURSORパラメータを使用するのではなく)戻すようにする必要があり、それにはDBMS_SQL.RETURN_RESULTプロシージャが必要である。

次の状況では、DBMS_SQLパッケージではなく、システム固有の動的SQLを使用する必要があります。

- 動的SQL文が行を取り出してレコードに入れる場合。
- INSERT文、UPDATE文、DELETE文、MERGE文または単一行のSELECT文である動的SQL文の発行後に、SQLのカーソル属性%FOUND、%ISOPEN、%NOTFOUNDまたは%ROWCOUNTを使用する場合。

DBMS_SQLパッケージとシステム固有の動的SQLの両方が必要な場合は、DBMS_SQL.TO_REFCURSORファンクションとDBMS_SQL.TO_CURSOR_NUMBERファンクションを使用して、それらを切り替えることができます。

ここでのトピック

- DBMS_SQL.RETURN_RESULTプロシージャ
- DBMS_SQL.GET_NEXT_RESULTプロシージャ
- DBMS_SQL.TO_REFCURSORファンクション
- DBMS_SQL.TO_CURSOR_NUMBERファンクション

ノート:

DBMS_SQL サブプログラムはリモートで起動できます。

関連項目:

- システム固有の動的SQLの詳細は、「システム固有の動的SQL」を参照してください
- 入力変数または出力変数の数が不明な動的SQL文の実行方法(「メソッド4」)を含むDBMS_SQLパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください

7.3.1 DBMS_SQL.RETURN_RESULTプロシージャ

DBMS_SQL.RETURN_RESULTプロシージャは、ストアド・サブプログラムが問合せ結果をクライアント・プログラム(サブプログラムを間接的に起動する)またはサブプログラムの直接のコール元に暗黙的に戻すことを可能にします。DBMS_SQL.RETURN_RESULTによって結果が戻された後、受信者のみがこれにアクセスできます。

DBMS_SQL.RETURN_RESULTには次の2つのオーバーロードがあります。

```
PROCEDURE RETURN_RESULT (rc IN OUT SYS_REFCURSOR,
                         to_client IN BOOLEAN DEFAULT TRUE);
PROCEDURE RETURN_RESULT (rc IN OUT INTEGER,
                         to_client IN BOOLEAN DEFAULT TRUE);
```

rcパラメータは、オープンされているカーソル変数(SYS_REFCURSOR)またはオープン・カーソルのカーソル番号(INTEGER)です。カーソルをオープンしてのそのカーソル番号を取得するには、DBMS_SQL.OPEN_CURSORファンクション(『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照)を起動します。

to_clientパラメータがTRUEの場合(デフォルト)、DBMS_SQL.RETURN_RESULTプロシージャは問合せ結果を(サブプログラムを間接的に起動する)クライアント・プログラムに戻します。このパラメータがFALSEの場合、プロシージャは問合せ結果をサブプログラムの直接のコール元に戻します。

関連項目:

- DBMS_SQL.RETURN_RESULTの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- Cおよび.NETでの暗黙的問合せ結果のサポートの詳細は、Oracle Call Interfaceプログラマーズ・ガイドを参照してください。
- 暗黙的問合せ結果に対するSQL*Plusのサポートの詳細は、『SQL*Plusユーザーズ・ガイドおよびリファレンス』を参照してください

例7-11 DBMS_SQL.RETURN_RESULTプロシージャ

この例では、プロシージャpがオプションのto_clientパラメータ(デフォルトではTRUE)なしで、DBMS_SQL.RETURN_RESULTを起動します。したがって、DBMS_SQL.RETURN_RESULTは問合せ結果をサブプログラム・クライアント(pを起動する無名ブロック)に戻します。pから無名ブロックに戻された結果には、その無名ブロックのみがアクセスできます。

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  c1 SYS_REFCURSOR;
  c2 SYS_REFCURSOR;
BEGIN
  OPEN c1 FOR
    SELECT first_name, last_name
    FROM employees
    WHERE employee_id = 176;

  DBMS_SQL.RETURN_RESULT (c1);
  -- Now p cannot access the result.

  OPEN c2 FOR
    SELECT city, state_province
    FROM locations
    WHERE country_id = 'AU';

  DBMS_SQL.RETURN_RESULT (c2);
  -- Now p cannot access the result.
END;
/
BEGIN
  p;
END;
/
```

結果:

```
ResultSet #1
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Jonathon             Taylor
ResultSet #2
CITY                           STATE_PROVINCE
------------------------------ -------------------------
Sydney                         New South Wales
```

7.3.2 DBMS_SQL.GET_NEXT_RESULTプロシージャ

DBMS_SQL.GET_NEXT_RESULTプロシージャは、DBMS_SQL.RETURN_RESULTプロシージャにより受信者に戻された次の結果を取得します。2つのプロシージャは結果を同じ順序で戻します。

DBMS_SQL.GET_NEXT_RESULTには次の2つのオーバーロードがあります。

```
PROCEDURE GET_NEXT_RESULT (c IN INTEGER, rc OUT SYS_REFCURSOR);
PROCEDURE GET_NEXT_RESULT (c IN INTEGER, rc OUT INTEGER);
```

cパラメータは、DBMS_SQL.RETURN_RESULTプロシージャを使用して問合せ結果を暗黙的に戻すサブプログラムを直接的または間接的に起動するオープン・カーソルのカーソル番号です。

カーソルをオープンしてのそのカーソル番号を取得するには、DBMS_SQL.OPEN_CURSORファンクションを起動します。DBMS_SQL.OPEN_CURSORにはオプションのパラメータtreat_as_client_for_resultsがあります。このパラメータがFALSEの場合(デフォルト)、このカーソルをオープンする(サブプログラムを起動するために)コール元は、DBMS_SQL.RETURN_RESULTを使用するサブプログラムからクライアントに対する問合せ結果を受け取るクライアントとして扱われません。これらの問合せ結果はかわりに上位層のクライアントに戻されます。このパラメータがTRUEの場合、コール元はクライアントとして扱われます。DBMS_SQL.OPEN_CURSORファンクションの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

rcパラメータは、カーソル変数(SYS_REFCURSOR)またはオープン・カーソルのカーソル番号(INTEGER)です。

例7-12では、プロシージャget_employee_infoはDBMS_SQL.RETURN_RESULTを使用して2つの問合せ結果をクライアント・プログラムに戻し、また無名ブロック<<main>>により動的に起動されています。<<main>>はget_employee_infoが戻す2つの問合せ結果を受け取る必要があるため、<<main>>はパラメータtreat_as_client_for_resultsをTRUEに設定したDBMS_SQL.OPEN_CURSORを使用して、get_employee_infoを起動するためのカーソルをオープンします。したがって、DBMS_SQL.GET_NEXT_RESULTは結果を<<main>>に戻し、<<main>>はカーソルrcを使用してその結果をフェッチします。

例7-12 DBMS_SQL.GET_NEXT_RESULTプロシージャ

```
CREATE OR REPLACE PROCEDURE get_employee_info (id IN VARCHAR2) AUTHID DEFINER AS
  rc  SYS_REFCURSOR;
BEGIN
  -- Return employee info

  OPEN rc FOR SELECT first_name, last_name, email, phone_number
              FROM employees
              WHERE employee_id = id;
  DBMS_SQL.RETURN_RESULT(rc);
```

```
  -- Return employee job history

  OPEN RC FOR SELECT job_title, start_date, end_date
              FROM job_history jh, jobs j
              WHERE jh.employee_id = id AND
                    jh.job_id = j.job_id
              ORDER BY start_date DESC;
  DBMS_SQL.RETURN_RESULT(rc);
END;
/
<<main>>
DECLARE
  c            INTEGER;
  rc           SYS_REFCURSOR;
  n            NUMBER;

  first_name   VARCHAR2(20);
  last_name    VARCHAR2(25);
  email        VARCHAR2(25);
  phone_number VARCHAR2(20);

  job_title    VARCHAR2(35);
  start_date   DATE;
  end_date     DATE;

BEGIN

  c := DBMS_SQL.OPEN_CURSOR(true);
  DBMS_SQL.PARSE(c, 'BEGIN get_employee_info(:id); END;', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c, ':id', 176);
  n := DBMS_SQL.EXECUTE(c);

  -- Get employee info

  dbms_sql.get_next_result(c, rc);
  FETCH rc INTO first_name, last_name, email, phone_number;

  DBMS_OUTPUT.PUT_LINE('Employee: '||first_name || ' ' || last_name);
  DBMS_OUTPUT.PUT_LINE('Email: ' ||email);
  DBMS_OUTPUT.PUT_LINE('Phone: ' ||phone_number);

  -- Get employee job history

  DBMS_OUTPUT.PUT_LINE('Titles:');
  DBMS_SQL.GET_NEXT_RESULT(c, rc);
  LOOP
    FETCH rc INTO job_title, start_date, end_date;
    EXIT WHEN rc%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE
      ('- '||job_title||' ('||start_date||' - ' ||end_date||')');
  END LOOP;

  DBMS_SQL.CLOSE_CURSOR(c);
END main;
/
```

結果:

```
Employee: Jonathon Taylor
Email: JTAYLOR
Phone: 011.44.1644.429265
Titles:
- Sales Manager (01-JAN-07 - 31-DEC-07)
- Sales Representative (24-MAR-06 - 31-DEC-06)

PL/SQL procedure successfully completed.
```

7.3.3 DBMS_SQL.TO_REFCURSORファンクション

DBMS_SQL.TO_REFCURSORファンクションは、SQLカーソル番号を、システム固有の動的SQL文で使用できる弱いカーソル変数に変換します。

SQLカーソル番号をDBMS_SQL.TO_REFCURSORファンクションに渡す前に、この番号に対してOPEN、PARSEおよびEXECUTEを実行する必要があります(そうしない場合、エラーが発生します)。

SQLカーソル番号をREF CURSOR変数に変換した後、DBMS_SQLの操作では、SQLカーソル番号としてではなく、REF CURSOR変数としてのみこの値にアクセスできます。たとえば、DBMS_SQL.IS_OPENファンクションを使用して、変換されたSQLカーソル番号がまだオープンしているかどうかを確認すると、エラーが発生します。

例7-13では、DBMS_SQL.TO_REFCURSORファンクションを使用して、DBMS_SQLパッケージからシステム固有の動的SQLに切り替えています。

例7-13 DBMS_SQLパッケージからシステム固有の動的SQLへの切替え

```
CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
  placeholder vc_array,
  bindvars vc_array,
  sql_stmt VARCHAR2
) AUTHID DEFINER
IS
  TYPE curtype IS REF CURSOR;
  src_cur     curtype;
  curid       NUMBER;
  bindnames   vc_array;
  empnos      numlist;
  depts       numlist;
  ret         NUMBER;
  isopen      BOOLEAN;
BEGIN
  -- Open SQL cursor number:
  curid := DBMS_SQL.OPEN_CURSOR;
  -- Parse SQL cursor number:
  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);
  bindnames := placeholder;
  -- Bind variables:
  FOR i IN 1 .. bindnames.COUNT LOOP
    DBMS_SQL.BIND_VARIABLE(curid, bindnames(i), bindvars(i));
  END LOOP;
```

```
  -- Run SQL cursor number:
  ret := DBMS_SQL.EXECUTE(curid);
  -- Switch from DBMS_SQL to native dynamic SQL:
  src_cur := DBMS_SQL.TO_REFCURSOR(curid);
  FETCH src_cur BULK COLLECT INTO empnos, depts;
  -- This would cause an error because curid was converted to a REF CURSOR:
  -- isopen := DBMS_SQL.IS_OPEN(curid);
  CLOSE src_cur;
END;
/
```

7.3.4 DBMS_SQL.TO_CURSOR_NUMBERファンクション

DBMS_SQL.TO_CURSOR_NUMBERファンクションは、(強いか弱いかに関係なく)REF CURSOR変数を、DBMS_SQLサブプログラムに渡すことができるSQLカーソル番号に変換します。

REF CURSOR変数をDBMS_SQL.TO_CURSOR_NUMBERファンクションに渡す前に、この変数に対してOPENを実行する必要があります。

REF CURSOR変数をSQLカーソル番号に変換した後、システム固有の動的SQL操作では、この値にアクセスできなくなります。

例7-14では、DBMS_SQL.TO_CURSOR_NUMBERファンクションを使用して、システム固有の動的SQLからDBMS_SQLパッケージに切り替えています。

例7-14 システム固有の動的SQLからDBMS_SQLパッケージへの切替え

```
CREATE OR REPLACE PROCEDURE do_query_2 (
  sql_stmt VARCHAR2
) AUTHID DEFINER
IS
  TYPE curtype IS REF CURSOR;
  src_cur   curtype;
  curid     NUMBER;
  desctab   DBMS_SQL.DESC_TAB;
  colcnt    NUMBER;
  namevar   VARCHAR2(50);
  numvar    NUMBER;
  datevar   DATE;
  empno     NUMBER := 100;
BEGIN
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';
  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;
  -- Switch from native dynamic SQL to DBMS_SQL package:
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);
  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
      -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 50);
```

```
    END IF;
  END LOOP;
  -- Fetch rows with DBMS_SQL package:
  WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
    FOR i IN 1 .. colcnt LOOP
      IF (desctab(i).col_type = 1) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
      ELSIF (desctab(i).col_type = 2) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
      ELSIF (desctab(i).col_type = 12) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
        -- statements
      END IF;
    END LOOP;
  END LOOP;
  DBMS_SQL.CLOSE_CURSOR(curid);
END;
/
```

7.4 SQLインジェクション

SQLインジェクションでは、SQL文内でクライアントから提供されるデータを使用するアプリケーションを悪用することによってデータベースに不正にアクセスし、制限付きデータを表示または操作します。

この項では、PL/SQLでのSQLインジェクションの脆弱性、およびそれらの回避方法について説明します。

ここでのトピック

- SQLインジェクション方法
- SQLインジェクションの回避

例7-15 SQLインジェクションの例のための設定

例を試すには、次の文を実行します。

Live SQL:

この例は、Oracle Live SQL の「SQL インジェクションのデモ」で表示および実行できます

```
DROP TABLE secret_records;
```

CREATE TABLE secret_records (user_name VARCHAR2(9), service_type VARCHAR2(12), value VARCHAR2(30), date_created DATE); INSERT INTO secret_records (user_name, service_type, value, date_created) VALUES ('Andy', 'Waiter', 'Serve dinner at Cafe Pete', SYSDATE); INSERT INTO secret_records (user_name, service_type, value, date_created) VALUES ('Chuck', 'Merger', 'Buy company XYZ', SYSDATE);

7.4.1 SQLインジェクション方法

すべてのSQLインジェクション方法は、1つの脆弱性(文字列入力が正常に検証されずに動的SQL文に連結される状況)を悪

用しています。

ここでのトピック

- 文の変更
- 文のインジェクション
- データ型変換

7.4.1.1 文の変更

文の変更とは、アプリケーション開発者が意図していない方法で動的SQL文が実行されるように、その動的SQL文を故意に変更することを意味します。

通常、ユーザーは、SELECT文のWHERE句を変更するか、UNION ALL句を挿入することによって不正データを取り出します。この方法の典型的な例は、WHERE句が常にTRUEになるようにしてパスワード認証をバイパスする方法です。

例7-16 文の変更に対して脆弱なプロシージャ

この例では、文の変更に対して脆弱なプロシージャを作成してから、そのプロシージャを文の変更がある場合とない場合で起動します。文の変更がある場合、プロシージャはシークレット・レコードを戻します。

Live SQL:

この例は、Oracle Live SQL の「SQL インジェクションのデモ」で表示および実行できます

脆弱なプロシージャの作成:

```
CREATE OR REPLACE PROCEDURE get_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
) AUTHID DEFINER
IS
  query VARCHAR2(4000);
BEGIN
  -- Following SELECT statement is vulnerable to modification
  -- because it uses concatenation to build WHERE clause.
  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || '''  AND service_type='''
           || service_type
           || '''';
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec ;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec );
END;
/
```

SQLインジェクションが行われないプロシージャの例:

```
SET SERVEROUTPUT ON;
DECLARE
```

```
  record_value VARCHAR2(4000);
BEGIN
  get_record('Andy', 'Waiter', record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter'
Rec: Serve dinner at Cafe Pete
```

文の変更の例:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record(
  'Anybody '' OR service_type=''Merger''--',
  'Anything',
  record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody ' OR
service_type='Merger'--' AND service_type='Anything'
Rec: Buy company XYZ
PL/SQL procedure successfully completed.
```

7.4.1.2 文のインジェクション

文のインジェクションとは、動的SQL文に対してユーザーが1つ以上のSQL文を追加することを意味します。

無名PL/SQLブロックは、この方法に対して脆弱です。

例7-17 文のインジェクションに対して脆弱なプロシージャ

この例では、文のインジェクションに対して脆弱なプロシージャを作成してから、そのプロシージャを文のインジェクションがある場合とない場合で起動します。文のインジェクションがある場合、プロシージャは例7-16にあるシークレット・レコードを削除します。

Live SQL:

この例は、Oracle Live SQL の「SQL インジェクションのデモ」で表示および実行できます

脆弱なプロシージャの作成:

```
CREATE OR REPLACE PROCEDURE p (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2
) AUTHID DEFINER
IS
```

```
  block1 VARCHAR2(4000);
BEGIN
  -- Following block is vulnerable to statement injection
  -- because it is built by concatenation.
  block1 :=
    'BEGIN
    DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');'
    || 'DBMS_OUTPUT.PUT_LINE(''service_type: ' || service_type || ''');
    END;';
  DBMS_OUTPUT.PUT_LINE('Block1: ' || block1);

  EXECUTE IMMEDIATE block1;
END;
/
```

SQLインジェクションが行われないプロシージャの例:

```
SET SERVEROUTPUT ON;
BEGIN
  p('Andy', 'Waiter');
END;
/
```

結果:

```
Block1: BEGIN
         DBMS_OUTPUT.PUT_LINE('user_name: Andy');
         DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
       END;
user_name: Andy
service_type: Waiter
```

SQL*Plus書式設定コマンド:

```
COLUMN date_created FORMAT A12;
```

問合せ:

```
SELECT * FROM secret_records ORDER BY user_name;
```

結果:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-10
Chuck     Merger       Buy company XYZ                28-APR-10
```

文の変更の例:

```
BEGIN
  p('Anybody', 'Anything'');
  DELETE FROM secret_records WHERE service_type=INITCAP(''Merger');
END;
/
```

結果:

```
Block1: BEGIN
       DBMS_OUTPUT.PUT_LINE('user_name: Anybody');
       DBMS_OUTPUT.PUT_LINE('service_type: Anything');
       DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
     END;
user_name: Anybody
service_type: Anything
PL/SQL procedure successfully completed.
```

問合せ:

```
SELECT * FROM secret_records;
```

結果:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      18-MAR-09

1 row selected.
```

7.4.1.3 データ型変換

あまり知られていないSQLインジェクション方法として、NLSセッション・パラメータを使用してSQL文を変更またはインジェクトする方法があります。

動的SQL文のテキストに連結されている日時値または数値は、VARCHAR2データ型に変換する必要があります。この変換は、暗黙的(値が連結演算子のオペランドの場合)または明示的(値がTO_CHARファンクションの引数の場合)のいずれかで行われます。このデータ型変換は、動的SQL文を実行するデータベース・セッションのNLS設定によって異なります。日時値の変換では、特定の日時データ型に応じて、NLS_DATE_FORMATパラメータ、NLS_TIMESTAMP_FORMATパラメータまたはNLS_TIMESTAMP_TZ_FORMATパラメータで指定されている書式モデルが使用されます。数値の変換では、NLS_NUMERIC_CHARACTERSパラメータで指定されている小数点およびグループ・セパレータが適用されます。

日時書式モデルの1つとして、"text"があります。textは、変換結果にコピーされます。たとえば、NLS_DATE_FORMATの値が'"Month:" Month'の場合、6月にはTO_CHAR(SYSDATE)によって'Month: June'が戻されます。この日時書式モデルは、例7-18に示すように、悪用される可能性があります。

例7-18 データ型変換によるSQLインジェクションに対して脆弱なプロシージャ

```
SELECT * FROM secret_records;
```

結果:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATE
--------- ------------ ------------------------------ -----------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-2010
Chuck     Merger       Buy company XYZ                28-APR-2010
```

脆弱なプロシージャの作成:

```
-- Return records not older than a month
```

```
CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
) AUTHID DEFINER
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
     because it uses concatenation to build WHERE clause
     and because SYSDATE depends on the value of NLS_DATE_FORMAT. */
  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || ''' AND date_created>'''
           || (SYSDATE - 30)
           || '''';
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

SQLインジェクションが行われないプロシージャの例:

```
SET SERVEROUTPUT ON;
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Andy', 'Waiter', record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter' AND date_created>'29-MAR-2010'
Rec: Serve dinner at Cafe Pete
```

文の変更の例:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created>'' OR service_type='Merger'
Rec: Buy company XYZ
```

```
PL/SQL procedure successfully completed.
```

7.4.2 SQLインジェクションの回避

PL/SQLアプリケーションで動的SQLを使用する場合は、入力テキストをチェックして、それが意図したとおりのものであることを確認する必要があります。

次の方法を使用できます。

- バインド変数
- 妥当性チェック
- 明示的な書式モデル

7.4.2.1 バインド変数

PL/SQLコードをSQLインジェクション攻撃に対して強固にする最も効率的な方法は、バインド変数を使用することです。

データベースでは、バインド変数の値が排他的に使用され、その内容は解釈されません。(バインド変数を使用すると、パフォーマンスも向上します。)

例7-19 SQLインジェクションを回避するためのバインド変数

この例に示すプロシージャは、(例7-16の脆弱なプロシージャのように連結を使用するのではなく)バインド変数を使用して動的SQL文を作成するため、SQLインジェクションに対して強固です。これと同じバインド方法で、例7-17に示した脆弱なプロシージャを修正できます。

強固なプロシージャの作成:

```
CREATE OR REPLACE PROCEDURE get_record_2 (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
) AUTHID DEFINER
IS
  query VARCHAR2(4000);
BEGIN
  query := 'SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);

  EXECUTE IMMEDIATE query INTO rec USING user_name, service_type;

  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

SQLインジェクションが行われないプロシージャの例:

```
SET SERVEROUTPUT ON;
```

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record_2('Andy', 'Waiter', record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b
Rec: Serve dinner at Cafe Pete

PL/SQL procedure successfully completed.
```

文の変更の試行:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record_2('Anybody '' OR service_type=''Merger''--',
               'Anything',
               record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "HR.GET_RECORD_2", line 15
ORA-06512: at line 4
```

7.4.2.2 妥当性チェック

ユーザー入力が意図したとおりのものになっていることを確認するために、常にプログラムでユーザー入力を検証する必要があります。

たとえば、ユーザーがDELETE文に対して部門番号を渡した場合は、departments表から選択することによって、この部門番号の妥当性をチェックします。同様に、ユーザーが削除対象の表の名前を入力した場合は、静的データ・ディクショナリ・ビューALL_TABLESから選択することによって、この表が存在していることを確認します。

注意:

ユーザー名とそのパスワードの妥当性をチェックする場合は、無効な項目に関係なく、常に同じエラーを戻してください。そうしない場合、エラー・メッセージ「無効なパスワード」を受信し、「無効なユーザー名」は受信していない(また

はその逆の状況の)悪意のあるユーザーが、これらのうちの 1 つについては推測がうまく当たったことに気付く可能性があります。

妥当性チェック・コードでは、多くの場合、DBMS_ASSERTパッケージ内のサブプログラムが有効です。たとえば、例7-20のようにDBMS_ASSERT.ENQUOTE_LITERALファンクションを使用して、文字列リテラルを引用符で囲むことができます。これによって、悪意のあるユーザーが、開き引用符とそれに対応する閉じ引用符の間にテキストをインジェクトできなくなります。

注意:

DBMS_ASSERT のサブプログラムは妥当性コードで有効ですが、妥当性コードに置き換わるものではありません。たとえば、入力文字列は、(DBMS_ASSERT.QUALIFIED_SQL_NAME によって検証された)修飾 SQL 名であっても、不正なパスワードである可能性があります。

関連項目:

DBMS_ASSERTサブプログラムの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

例7-20 SQLインジェクションを回避するための妥当性チェック

この例では、プロシージャraise_emp_salaryはemployees表を更新する前に、渡された列名の妥当性をチェックします。その後、無名ブロックによって、動的PL/SQLブロックと動的SQL文の両方からこのプロシージャが起動されます。

```
CREATE OR REPLACE PROCEDURE raise_emp_salary (
  column_value  NUMBER,
  emp_column    VARCHAR2,
  amount NUMBER ) AUTHID DEFINER
IS
  v_column  VARCHAR2(30);
  sql_stmt  VARCHAR2(200);
BEGIN
  -- Check validity of column name that was given as input:
  SELECT column_name INTO v_column
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = 'EMPLOYEES'
  AND COLUMN_NAME = emp_column;
  sql_stmt := 'UPDATE employees SET salary = salary + :1 WHERE '
    || DBMS_ASSERT.ENQUOTE_NAME(v_column,FALSE) || ' = :2';
  EXECUTE IMMEDIATE sql_stmt USING amount, column_value;
  -- If column name is valid:
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Salaries were updated for: '
      || emp_column || ' = ' || column_value);
  END IF;
  -- If column name is not valid:
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Invalid Column: ' || emp_column);
END raise_emp_salary;
/
DECLARE
```

```
  plsql_block  VARCHAR2(500);
BEGIN
  -- Invoke raise_emp_salary from a dynamic PL/SQL block:
  plsql_block :=
    'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;';
  EXECUTE IMMEDIATE plsql_block
    USING 110, 'DEPARTMENT_ID', 10;
  -- Invoke raise_emp_salary from a dynamic SQL statement:
  EXECUTE IMMEDIATE 'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;'
    USING 112, 'EMPLOYEE_ID', 10;
END;
/
```

結果:

```
Salaries were updated for: DEPARTMENT_ID = 110
Salaries were updated for: EMPLOYEE_ID = 112
```

7.4.2.3 明示的な書式モデル

セキュリティ面からのみではなく、動的SQL文がすべてのグローバリゼーション環境で正常に実行されるようにするためにも、ロケールに依存しない明示的な書式モデルを使用してSQL文を構成することをお薦めします。

SQL文またはPL/SQL文のテキストに連結されている日時値または数値を使用しており、これらの値をバインド引数として渡すことができない場合は、実行中のセッションのNLSパラメータの値に依存しない明示的な書式モデルを使用して、これらの値をテキストに変換します。変換された値がSQLの日時リテラルまたは数値リテラルの書式になっているかどうかを確認します。

例7-21 SQLインジェクションを回避するための明示的な書式モデル

このプロシージャは、SQLインジェクションに対して強固です。例7-18の脆弱なプロシージャのように暗黙的ではなく、TO_CHARファンクションおよびロケールに依存しない書式モデルを使用して、日時パラメータ値のSYSDATE - 30をVARCHAR2値に明示的に変換するためです。

強固なプロシージャの作成:

```
-- Return records not older than a month
CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name     IN  VARCHAR2,
  service_type  IN  VARCHAR2,
  rec           OUT VARCHAR2
) AUTHID DEFINER
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
     because it uses concatenation to build WHERE clause. */
  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || '''  AND service_type='''
           || service_type
           || '''  AND date_created> DATE '''
           || TO_CHAR(SYSDATE - 30,'YYYY-MM-DD')
           || '''';
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
```

```
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

文の変更の試行:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

結果:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created> DATE '2010-03-29'
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SYS.GET_RECENT_RECORD", line 21
ORA-06512: at line 4
```

8 PL/SQLサブプログラム

PL/SQLのサブプログラムは、繰り返し起動できる名前付きPL/SQLブロックです。サブプログラムにパラメータが含まれている場合は、起動のたびにパラメータ値を変えることができます。

サブプログラムは、プロシージャまたはファンクションのいずれかです。通常、プロシージャはアクションを実行するために使用し、ファンクションは計算を行って値を戻すために使用します。

ここでのトピック

- サブプログラムを使用する理由
- ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラム
- サブプログラムの起動
- サブプログラムのプロパティ
- サブプログラムの各部
- 前方宣言
- サブプログラムのパラメータ
- サブプログラムの起動の解決
- オーバーロードされたサブプログラム
- 再帰的サブプログラム
- サブプログラムの副作用
- PL/SQLファンクション結果キャッシュ
- SQL文によって起動できるPL/SQLファンクション
- 実行者権限および定義者権限(AUTHIDプロパティ)
- 外部サブプログラム

8.1 サブプログラムを使用する理由

サブプログラムでは、次の特長に基づく信頼性の高い再利用可能なコードの開発およびメンテナンスがサポートされます。

- モジュール性

  サブプログラムを使用すると、プログラムを管理の容易な、正しく定義されたモジュールに分けることができます。

- アプリケーションの設計の容易さ

  アプリケーションを設計する場合、メイン・プログラムのテストが終了するまでサブプログラムの実装の詳細を遅らせ、段階的に各サブプログラムを改良できます。(実装の詳細なしでサブプログラムを定義するには、例4-35に示すとおり、NULL文を使用します。)

- メンテナンス性

  起動元を変更することなく、サブプログラムの実装の詳細を変更できます。

- パッケージ性

  サブプログラムは、パッケージにグループ化できます。そのメリットの詳細は、「パッケージを使用する理由」を参照してください。

- 再利用性

  多くの異なる環境において、任意の数のアプリケーションで、同じパッケージ・サブプログラムまたはスタンドアロン・サブプログラムを使用できます。

- より高いパフォーマンス

  各サブプログラムはコンパイルされてから実行可能な形式で格納され、繰り返し起動することができます。ストアド・サブプログラムはデータベース・サーバー内で実行されるため、ネットワークを介した1回の起動で大規模なジョブを開始できます。この作業の分割によってネットワークの通信量が軽減され、応答時間が短縮されます。ストアド・サブプログラムはキャッシュされ、ユーザー間で共有されるため、メモリー要件と起動のオーバーヘッドが低減します。

サブプログラムは、パッケージ(「PL/SQLパッケージ」を参照)や抽象データ型(「抽象データ型」を参照)などのメンテナンス性を向上させる他の機能の重要なコンポーネントです。

8.2 ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラム

サブプログラムは、PL/SQLブロック内(別のサブプログラムの可能性もあります)、パッケージ内またはスキーマ・レベルで作成できます。

PL/SQLブロック内で作成されるサブプログラムは、ネストしたサブプログラムです。サブプログラムは、宣言と定義を同時に行うか、または最初に宣言して、後で同じブロック内で定義することができます(「前方宣言」を参照)。ネストしたサブプログラムは、スタンドアロン・サブプログラムまたはパッケージ・サブプログラム内でネストした場合にのみ、データベースに格納されます。

パッケージ内で作成されるサブプログラムは、パッケージ・サブプログラムです。パッケージ仕様部で宣言し、パッケージ本体で定義します。パッケージを削除するまでデータベースに格納されます。(パッケージについては、「PL/SQLパッケージ」を参照してください。)

スキーマ・レベルで作成されるサブプログラムは、スタンドアロン・サブプログラムです。CREATE FUNCTION文またはCREATE PROCEDURE文を使用して作成できます。DROP FUNCTION文またはDROP PROCEDURE文を使用して削除するまでデータベースに格納されます。(これらの文の詳細は、「ストアドPL/SQLユニット用のSQL文」を参照してください。)

ストアド・サブプログラムは、パッケージ・サブプログラムまたはスタンドアロン・サブプログラムのいずれかです。ストアド・サブプログラムはAUTHIDおよびACCESSIBLE BY句の影響を受けます。これらはCREATE FUNCTION、CREATE PROCEDUREおよびCREATE PACKAGE文で使用されます。AUTHID句はサブプログラムが実行時に発行するSQL文の名前解決と権限チェックに影響します(詳細は「実行者権限および定義者権限(AUTHIDプロパティ)」)を参照してください。ACCESSIBLE BY句は、サブプログラムにアクセスできるPL/SQLユニットのホワイト・リストを指定します。

8.3 サブプログラムの起動

サブプログラムの起動の形式は次のとおりです。

```
subprogram_name [ ( [ parameter [, parameter]... ] ) ]
```

サブプログラムにパラメータがない場合、またはサブプログラムによってすべてのパラメータにデフォルト値が指定される場合は、パラメータ・リストを省略するかまたは空のパラメータ・リストを指定できます。

プロシージャの起動はPL/SQL文です。たとえば:

```
raise_salary(employee_id, amount);
```

ファンクションの起動は式です。たとえば:

```
new_salary := get_salary(employee_id);
IF salary_ok(new_salary, new_title) THEN ...
```

関連項目:

サブプログラムの起動でパラメータを指定する方法の詳細は、「サブプログラムのパラメータ」を参照してください

8.4 サブプログラムのプロパティ

サブプログラムの各プロパティは、サブプログラムの宣言で1回のみ記述できます。プロパティは任意の順序で記述できます。プロパティは、サブプログラムのヘッダー内のISまたはASキーワードの前に記述します。プロパティは、ネストしたサブプログラムには記述できません。

パッケージ・サブプログラムに記述できるのは、ACCESSIBLE BYプロパティのみです。スタンドアロン・サブプログラムの宣言には次のプロパティを使用できます。

- ACCESSIBLE BY句
- DEFAULT COLLATION句
- 実行者権限および定義者権限(AUTHIDプロパティ)

8.5 サブプログラムの各部

サブプログラムの先頭には、その名前およびオプションのパラメータ・リストを指定するサブプログラムのヘッダーがあります。

無名ブロックと同様に、サブプログラムには次の部分があります。

- 宣言部(オプション)

  この部分では、ローカル型、カーソル、定数、変数、例外およびネストしたサブプログラムを宣言および定義します。これらの項目は、サブプログラムの実行が完了すると消滅します。

  この部分は、プラグマを指定することもできます。

ノート:

無名ブロックの宣言部とは異なり、サブプログラムの宣言部は、キーワード DECLARE では始まりませ

ん。

- 実行部(必須)

  この部分には、値の代入、実行の制御およびデータの操作を実行する1つ以上の文が含まれています。(アプリケーション設計プロセスの初期段階では、例4-35に示すとおり、この部分にNULL文のみを含めることも可能です。)

- 例外処理部(オプション)

  この部分には、ランタイム・エラーを処理するコードが含まれています。

ここでのトピック

- ファンクションの追加部分
- RETURN文

関連項目:

- 「プラグマ」
- プロシージャ宣言と定義の構文は、「プロシージャの宣言および定義」を参照してください
- サブプログラムのパラメータの詳細は、「サブプログラムのパラメータ」を参照してください

例8-1 単純なPL/SQLプロシージャの宣言、定義および起動

この例では、無名ブロックでプロシージャの宣言と定義を同時に行い、そのプロシージャを3回起動しています。3回目の起動では、プロシージャの例外処理部で処理される例外が呼び出されています。

```
DECLARE
  first_name employees.first_name%TYPE;
  last_name  employees.last_name%TYPE;
  email      employees.email%TYPE;
  employer   VARCHAR2(8) := 'AcmeCorp';

  -- Declare and define procedure

  PROCEDURE create_email (  -- Subprogram heading begins
    name1   VARCHAR2,
    name2   VARCHAR2,
    company VARCHAR2
  )                         -- Subprogram heading ends
  IS
                            -- Declarative part begins
    error_message VARCHAR2(30) := 'Email address is too long.';
  BEGIN                     -- Executable part begins
    email := name1 || '.' || name2 || '@' || company;
  EXCEPTION                     -- Exception-handling part begins
    WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE(error_message);
  END create_email;

BEGIN
  first_name := 'John';
  last_name  := 'Doe';
```

```
  create_email(first_name, last_name, employer);  -- invocation
  DBMS_OUTPUT.PUT_LINE ('With first name first, email is: ' || email);

  create_email(last_name, first_name, employer);  -- invocation
  DBMS_OUTPUT.PUT_LINE ('With last name first, email is: ' || email);

  first_name := 'Elizabeth';
  last_name  := 'MacDonald';
  create_email(first_name, last_name, employer);  -- invocation
END;
/
```

結果:

```
With first name first, email is: John.Doe@AcmeCorp
With last name first, email is: Doe.John@AcmeCorp
Email address is too long.
```

8.5.1 ファンクションの追加部分

ファンクションの構造は、プロシージャと同じですが、次の点が異なります。

- ファンクションのヘッダーには、ファンクションが戻す値のデータ型を指定するRETURN句が含まれている必要があります。(プロシージャのヘッダーには、RETURN句を含めることはできません。)
- ファンクションの実行部では、すべての実行パスがRETURN文へ導かれる必要があります。そうではない場合、PL/SQLコンパイラによってコンパイル時に警告が発行されます。(プロシージャでは、RETURN文はオプションであり、推奨されません。詳細は、「RETURN文」を参照してください。)
- ファンクションの宣言には、次に示すオプションを含めることができます。

オプション	説明
DETERMINISTIC オプション	オプティマイザが冗長なファンクションの起動を回避するために役立ちます。
PARALLEL_ENABLE オプション	ファンクションのパラレル実行を可能にし、パラレル DML 評価の同時セッションで安全に使用できるようにします。
PIPELINED オプション	行ソースとして使用するためにテーブル・ファンクションをパイプライン化します。
RESULT_CACHE オプション	PL/SQL ファンクション結果キャッシュに、ファンクションの結果を格納します。

関連項目:

- 前述の表内の項目の説明を含む、ファンクション宣言と定義の構文は、「ファンクションの宣言および定義」を参照してください
- RESULT_CACHEオプションの詳細は、「PL/SQLファンクション結果キャッシュ」を参照してください

例8-2 単純なPL/SQLファンクションの宣言、定義および起動

この例では、無名ブロックでファンクションの宣言と定義を同時に行い、そのファンクションを起動しています。

```
DECLARE
  -- Declare and define function
  FUNCTION square (original NUMBER)   -- parameter list
    RETURN NUMBER                     -- RETURN clause
  AS
                                      -- Declarative part begins
    original_squared NUMBER;
  BEGIN                               -- Executable part begins
    original_squared := original * original;
    RETURN original_squared;          -- RETURN statement
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE(square(100));  -- invocation
END;
/
```

結果:

```
10000
```

8.5.2 RETURN文

RETURN文は、サブプログラムまたはこの文を含む無名ブロックの実行を即座に終了させます。サブプログラムまたは無名ブロックでは、複数のRETURN文を使用できます。

ここでのトピック

- ファンクションのRETURN文
- プロシージャのRETURN文
- 無名ブロックのRETURN文

関連項目:

RETURN文の構文は、「RETURN文」を参照してください

8.5.2.1 ファンクションのRETURN文

ファンクションでは、すべての実行パスがRETURN文に続き、すべてのRETURN文が式を指定している必要があります。RETURN文は、ファンクション識別子に式の値を代入し、起動元に制御を戻します(ここで、起動直後に実行が再開します)。

> ノート:
>
> パイプライン・テーブル・ファンクションでは、RETURN 文で式を指定する必要はありません。パイプライン・テーブル・ファンクションの各部の詳細は、「パイプライン・テーブル・ファンクションの作成」を参照してください。

例8-3では、無名ブロックで同じファンクションを2回起動しています。1回目では、RETURN文は起動元の文の内部に制御を戻

します。2回目では、RETURN文は起動元の文の直後の文に制御を戻します。

例8-4では、ファンクションに複数のRETURN文が含まれますが、パラメータが0または1でない場合、実行パスがRETURN文へ導かれません。このファンクションは、コンパイル時に「PLW-05005: サブプログラムFは、行11に値なしで戻ります」という警告が発生します。

例8-5は、例8-4と同様ですが、ELSE句が追加されています。すべての実行パスがRETURN文へ導かれるため、ファンクションのコンパイル時に警告PLW-05005は発生しません。

例8-3 ファンクション内にあるRETURN文の後での実行の再開

```
DECLARE
  x INTEGER;

  FUNCTION f (n INTEGER)
  RETURN INTEGER
  IS
  BEGIN
    RETURN (n*n);
  END;

BEGIN
  DBMS_OUTPUT.PUT_LINE (
    'f returns ' || f(2) || '. Execution returns here (1).'
  );

  x := f(2);
  DBMS_OUTPUT.PUT_LINE('Execution returns here (2).');
END;
/
```

結果:

```
f returns 4. Execution returns here (1).Execution returns here (2).
```

例8-4 RETURN文へ導かれない実行パスを含むファンクション

```
CREATE OR REPLACE FUNCTION f (n INTEGER)
  RETURN INTEGER
  AUTHID DEFINER
IS
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSIF n = 1 THEN
    RETURN n;
  END IF;
END;
/
```

例8-5 すべての実行パスがRETURN文へ導かれるファンクション

```
CREATE OR REPLACE FUNCTION f (n INTEGER)
  RETURN INTEGER
  AUTHID DEFINER
```

```
IS
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSIF n = 1 THEN
    RETURN n;
  ELSE
    RETURN n*n;
  END IF;
END;
/
BEGIN
  FOR i IN 0 .. 3 LOOP
    DBMS_OUTPUT.PUT_LINE('f(' || i || ') = ' || f(i));
  END LOOP;
END;
/
```

結果:

```
f(0) = 1
f(1) = 1
f(2) = 4
f(3) = 9
```

8.5.2.2 プロシージャのRETURN文

プロシージャでは、RETURN文は起動元に制御を戻します。ここで、起動すると実行が即座に再開します。RETURN文では、式を指定できません。

例8-6では、RETURN文は、起動元の文の直後の文に制御を戻しています。

例8-6 プロシージャ内にあるRETURN文の後での実行の再開

```
DECLARE
  PROCEDURE p IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside p');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('Unreachable statement.');
  END;
BEGIN
  p;
  DBMS_OUTPUT.PUT_LINE('Control returns here.');
END;
/
```

結果:

```
Inside p
Control returns here.
```

8.5.2.3 無名ブロックのRETURN文

無名ブロックでは、RETURN文によってそのブロックおよびすべての外側のブロックが終了されます。RETURN文では、式を指定でき

ません。

例8-7では、RETURN文によって、内側と外側の両方のブロックが終了されています。

例8-7 無名ブロック内にあるRETURN文の後での実行の再開

```
BEGIN
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside inner block.');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('Unreachable statement.');
  END;
  DBMS_OUTPUT.PUT_LINE('Inside outer block. Unreachable statement.');
END;
/
```

結果:

```
Inside inner block.
```

8.6 前方宣言

サブプログラムは起動の前に宣言する必要があるため、同じPL/SQLブロック内のネストしたサブプログラムが相互に起動し合う場合は、そのサブプログラムに前方宣言が必要です。

前方宣言では、ネストしたサブプログラムを宣言しますが、定義はしません。同じブロックで後から定義する必要があります。前方宣言および定義は、同じサブプログラム・ヘッダーを持つ必要があります。

例8-8では、無名ブロックによって、相互に起動し合う2つのプロシージャを作成しています。

例8-8 相互に起動し合うネストしたサブプログラム

```
DECLARE
  -- Declare proc1 (forward declaration):
  PROCEDURE proc1(number1 NUMBER);
  -- Declare and define proc2:
  PROCEDURE proc2(number2 NUMBER) IS
  BEGIN
    proc1(number2);
  END;
  -- Define proc 1:
  PROCEDURE proc1(number1 NUMBER) IS
  BEGIN
    proc2 (number1);
  END;
BEGIN
  NULL;
END;
/
```

8.7 サブプログラムのパラメータ

サブプログラムにパラメータが含まれている場合は、起動のたびにパラメータ値を変えることができます。

ここでのトピック

- サブプログラムの仮パラメータと実パラメータ
- サブプログラムのパラメータの引渡し方法
- サブプログラムのパラメータ・モード
- サブプログラムのパラメータのエイリアシング
- サブプログラムのINパラメータのデフォルト値
- 実パラメータの位置表記法、名前表記法および混合表記法

8.7.1 サブプログラムの仮パラメータと実パラメータ

サブプログラムにパラメータを保持する場合、サブプログラムのヘッダーで仮パラメータを宣言します。仮パラメータの宣言ごとに、パラメータの名前とデータ型を指定し、オプションでそのモードとデフォルト値を指定します。サブプログラムの実行部で、仮パラメータをその名前によって参照します。

サブプログラムを起動する際に、仮パラメータに代入する値が含まれる実パラメータを指定します。対応する実パラメータと仮パラメータのデータ型には、互換性が必要です。

ノート:

次のように、制約付きサブタイプの仮パラメータを宣言できます。

```
DECLARE
  SUBTYPE n1 IS NUMBER(1);
  SUBTYPE v1 IS VARCHAR2(1);

  PROCEDURE p (n n1, v v1) IS ...
```

ただし、次のように、仮パラメータの宣言内に制約を含めることはできません。

```
DECLARE
  PROCEDURE p (n NUMBER(1), v VARCHAR2(1)) IS ...
```

ヒント:

混同を避けるため、仮パラメータと実パラメータには異なる名前を使用してください。

ノート:

- 実際のパラメータ(仮パラメータのデフォルト値を含む)はどのような順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

- サーバー間リモート・プロシージャ・コール(RPC)では LOB パラメータは使用できません。

例8-9では、プロシージャに仮パラメータのemp_idおよびamountが含まれます。プロシージャの最初の起動では、対応する実パラメータはemp_num(値120)およびbonus(値100)です。プロシージャの2回目の起動では、実パラメータはemp_num(値120)およびmerit + bonus(値150)です。

トピック:

- 制約付きサブタイプの仮パラメータ

関連項目:

- 仮パラメータの宣言の構文およびセマンティクスは、「仮パラメータの宣言」を参照してください
- ファンクションの起動の構文およびセマンティクスは、「function_call ::=」および「function_call」を参照してください
- プロシージャの起動の構文およびセマンティクスは、procedure_call ::=およびprocedureを参照してください

例8-9 仮パラメータと実パラメータ

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6) := 100;
  merit   NUMBER(4) := 50;
  PROCEDURE raise_salary (
    emp_id NUMBER,  -- formal parameter
    amount NUMBER   -- formal parameter
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount  -- reference to formal parameter
    WHERE employee_id = emp_id;   -- reference to formal parameter
  END raise_salary;
BEGIN
  raise_salary(emp_num, bonus);          -- actual parameters
  /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 100
       WHERE employee_id = 120;       */
  raise_salary(emp_num, merit + bonus);  -- actual parameters
  /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 150
       WHERE employee_id = 120;       */
END;
/
```

8.7.1.1 制約付きサブタイプの仮パラメータ

仮パラメータのデータ型が制約付きサブタイプの場合、次のことに注意してください。

- サブタイプにNOT NULL制約が含まれる場合、実パラメータはその制約を継承します。
- サブタイプにベース型VARCHAR2が含まれる場合、実パラメータはサブタイプのサイズを継承しません。

- サブタイプに数値ベース型が含まれる場合、実パラメータはサブタイプの範囲を継承しますが、精度または位取りは継承しません。

> ノート:
>
> ファンクションでは、RETURN datatype 句で非表示仮パラメータを宣言し、RETURN value 文で対応する実パラメータを指定します。したがって、datatype が制約付きデータ型の場合は、前述の規則が value に適用されます(例 8-11 を参照)。

例8-10で、サブプログラムの実パラメータは、NOT NULL制約を継承していますが、VARCHAR2サブタイプのサイズは継承していません。

「PL/SQLの事前定義のデータ型」に示されているように、PL/SQLには、他のデータ型の制約付きサブタイプである多くの事前定義のデータ型が用意されています。たとえば、INTEGERはNUMBERの制約付きサブタイプです。

```
SUBTYPE INTEGER IS NUMBER(38,0);
```

例8-11では、ファンクションに仮パラメータINTEGERと戻り値INTEGERの両方が含まれています。無名ブロックは、整数でない実パラメータを使用してファンクションを起動します。実パラメータはINTEGERの範囲を継承しますが、精度と位取りは継承せず、また実パラメータはINTEGERの範囲内にあるため、起動が成功します。同じ理由で、RETURN文は正常に非整数値を戻します。

例8-12では、ファンクションは仮パラメータを戻す前に、その仮パラメータを制約付きサブタイプINTEGERに暗黙的に変換しています。

関連項目:

制約付きサブタイプの一般情報は、「制約付きサブタイプ」を参照してください

例8-10 サブタイプからNOT NULLのみを継承する実パラメータ

```
DECLARE
  SUBTYPE License IS VARCHAR2(7) NOT NULL;
  n  License := 'DLLLDDD';

  PROCEDURE p (x License) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(x);
  END;

BEGIN
  p('1ABC123456789');  -- Succeeds; size is not inherited
  p(NULL);             -- Raises error; NOT NULL is inherited
END;
/
```

結果:

```
  p(NULL);             -- Raises error; NOT NULL is inherited
    *
```

```
ERROR at line 12:
ORA-06550: line 12, column 5:
PLS-00567: cannot pass NULL to a NOT NULL constrained formal parameter
ORA-06550: line 12, column 3:
PL/SQL: Statement ignored
```

例8-11 サブタイプから範囲のみを継承する実パラメータと戻り値

```
DECLARE
  FUNCTION test (p INTEGER) RETURN INTEGER IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('p = ' || p);
    RETURN p;
  END test;

BEGIN
  DBMS_OUTPUT.PUT_LINE('test(p) = ' || test(0.66));
END;
/
```

結果:

```
p = .66
test(p) = .66

PL/SQL procedure successfully completed.
```

例8-12 ファンクションによる仮パラメータから制約付きサブタイプへの暗黙的な変換

```
DECLARE
  FUNCTION test (p NUMBER) RETURN NUMBER IS
    q INTEGER := p;  -- Implicitly converts p to INTEGER
  BEGIN
    DBMS_OUTPUT.PUT_LINE('p = ' || q);  -- Display q, not p
    RETURN q;                           -- Return q, not p
  END test;

BEGIN
  DBMS_OUTPUT.PUT_LINE('test(p) = ' || test(0.66));
END;
/
```

結果:

```
p = 1
test(p) = 1

PL/SQL procedure successfully completed.
```

8.7.2 サブプログラムのパラメータの引渡し方法

PL/SQLコンパイラには、実パラメータをサブプログラムに引き渡す方法として次の2つがあります。

- 参照渡し

  コンパイラは、実パラメータを指すポインタをサブプログラムに渡します。実パラメータと仮パラメータは、同じメモリー位置

を参照します。

- 値渡し

  コンパイラは、実パラメータの値を対応する仮パラメータに代入します。実パラメータと仮パラメータは、異なるメモリー位置を参照します。

  コンパイラは、必要に応じて実パラメータのデータ型を仮パラメータのデータ型に暗黙的に変換します。暗黙的なデータ変換の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

> ヒント:
>
> (『Oracle Database SQL 言語リファレンス』で説明されている理由のために)暗黙的なデータ変換を回避するには、次のいずれかの方法を使用します。
>
> - 実パラメータとして使用する予定の変数を、それに対応する仮パラメータと同じデータ型で宣言します(例 8-13 の変数 x の宣言を参照)。
> - 『Oracle Database SQL 言語リファレンス』で説明されている SQL 変換ファンクションを使用して、実パラメータをそれに対応する仮パラメータのデータ型に明示的に変換します(例 8-13 の 3 回目のプロシージャの起動を参照)。

例8-13では、プロシージャpに、値によって渡される1つのパラメータnが含まれています。無名ブロックは、pを3回起動していますが、そのうち2回は暗黙的な変換を回避しています。

コンパイラで特定の実パラメータが渡される方法は、そのモードによって異なります(「サブプログラムのパラメータ・モード」を参照)。

例8-13 実パラメータの暗黙的な変換の回避

```
CREATE OR REPLACE PROCEDURE p (
  n NUMBER
) AUTHID DEFINER IS
BEGIN
  NULL;
END;
/
DECLARE
  x NUMBER      :=  1;
  y VARCHAR2(1) := '1';
BEGIN
  p(x);              -- No conversion needed
  p(y);              -- z implicitly converted from VARCHAR2 to NUMBER
  p(TO_NUMBER(y));   -- z explicitly converted from VARCHAR2 to NUMBER
END;
/
```

8.7.3 サブプログラムのパラメータ・モード

仮パラメータのモードによって、その動作が決定されます。

表8-1に、サブプログラムのパラメータ・モードの特性の概要および比較を示します。

表8-1 PL/SQLサブプログラムのパラメータ・モード

パラメータ・モード	デフォルト	ロール
IN	デフォルト・モード	サブプログラムに値を渡します。
OUT	指定する必要があります。	起動元に値を戻します。
IN OUT	指定する必要があります。	サブプログラムに初期値を渡し、更新された値を起動元に戻します。

表8-2 PL/SQLサブプログラムのパラメータ・モードの特性

パラメータ・モード	仮パラメータ	実パラメータ	参照渡し
IN	仮パラメータは定数のように動作します。サブプログラムが起動すると、その値は実パラメータまたはデフォルト値のいずれかの値となり、サブプログラムではその値を変更できません。	実パラメータには定数、リテラル、初期化された変数または式が使用できます。	実パラメータは参照によって渡されます。
OUT	仮パラメータはその型のデフォルト値に初期化されます。NULL 以外のデフォルト値を持つレコード型を除いて、型のデフォルト値は NULL です(例 8-16 を参照)。 サブプログラムが起動すると、実パラメータの値にかかわらず、仮パラメータの値は初期値になります。サブプログラムで仮パラメータに値を割り当てることをお薦めします。	仮パラメータの型のデフォルト値が NULL である場合、実パラメータは、データ型が NOT NULL として定義されていない変数である必要があります。	デフォルトでは、実パラメータは値によって渡されます。NOCOPY を指定すると、参照によって渡される場合があります。
IN OUT	仮パラメータは初期化された変数のように動作します(サブプログラムが起動すると、その値は実パラメータの値になります)。サブプログラムで値を更新することをお薦めします。	実パラメータは、変数である必要があります(通常、文字列バッファまたは数値アキュムレータです)。	デフォルトでは、実パラメータは値によって渡されます(どちらの向きに対しても)。NOCOPY を指定すると、参照によって渡される場合があります。

ヒント:

ファンクションのパラメータには、OUT および IN OUT を使用しないでください。ファンクションは、0(ゼロ)個以上のパラメータを取り、単一の値を戻すことが理想です。IN OUT パラメータを持つファンクションは、複数の値を戻すため、副作用が発生します。

ノート:

Oracle Database 提供の多くのパッケージと型の仕様部では、次の表記法を使用して仮パラメータを宣言します。

```
i1 IN VARCHAR2 CHARACTER SET ANY_CS
i2 IN VARCHAR2 CHARACTER SET i1%CHARSET
```

独自の仮パラメータまたは実パラメータを宣言する場合は、この表記法を使用しないでください。これは、提供されているパッケージ・タイプの Oracle 実装のために予約されています。

OUTまたはIN OUTパラメータが渡される方法にかかわらず、次のようになります。

- サブプログラムが正常に終了した場合、実パラメータの値が、仮パラメータに代入される最終的な値になります。(仮パラメータには、少なくとも1つの値(初期値)が代入されます。)
- サブプログラムが例外によって終了した場合、実パラメータの値は定義されません。
- OUTおよびIN OUT仮パラメータは任意の順序で返されます。次の例では、xおよびyの最終的な値は定義されていません。

```
CREATE OR REPLACE PROCEDURE p (x OUT INTEGER, y OUT INTEGER) AS
BEGIN
  x := 17; y := 93;
END;
/
```

OUTまたはIN OUTパラメータが参照によって渡される場合、実パラメータと仮パラメータは同じメモリー位置を参照します。そのため、サブプログラムによって仮パラメータの値が変更されると、その変更は実パラメータに即座に反映されます(「参照渡しパラメータによるサブプログラムのパラメータのエイリアシング」を参照)。

例8-14では、プロシージャpに2つのINパラメータ、1つのOUTパラメータおよび1つのIN OUTパラメータが含まれます。OUTパラメータとIN OUTパラメータは、値によって渡されます(デフォルト)。無名ブロックは、異なる実パラメータを使用してpを2回起動しています。各起動の前に、無名ブロックは実パラメータの値を出力します。プロシージャpは、仮パラメータの初期値を出力します。各起動の後にも、無名ブロックは実パラメータの値を出力します。

例8-15では、無名ブロックがプロシージャp(例8-14を参照)を起動していますが、その実パラメータによってpで事前定義の例外ZERO_DIVIDEが呼び出されます(pはこの例外を処理しません)。例外は無名ブロックに伝播され、無名ブロックは、ZERO_DIVIDEを処理し、pのINおよびIN OUTパラメータの実パラメータが起動前に保持していた値を現在も維持していることを示します。(例外の伝播の詳細は、「例外の伝播」を参照してください。)

例8-16では、プロシージャpに3つのOUT仮パラメータがあります(NULL以外のデフォルト値を持つレコード型のx、NULL以外のデ

フォルト値を持たないレコード型のy、およびレコードではないz)。

x、yおよびzの対応する実パラメータはそれぞれ、r1、r2およびsです。sは初期値を使用して宣言されます。ただし、pが起動されると、sの値はNULLに初期化されます。r1およびr2の値は、それらのレコード型のデフォルト値(それぞれabcde、NULL)に初期化されます。

例8-14 プロシージャの起動前、起動中および起動後のパラメータ値

```
CREATE OR REPLACE PROCEDURE p (
  a        PLS_INTEGER,  -- IN by default
  b     IN PLS_INTEGER,
  c    OUT PLS_INTEGER,
  d IN OUT BINARY_FLOAT
) AUTHID DEFINER IS
BEGIN
  -- Print values of parameters:

  DBMS_OUTPUT.PUT_LINE('Inside procedure p:');

  DBMS_OUTPUT.PUT('IN a = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(a), 'NULL'));

  DBMS_OUTPUT.PUT('IN b = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(b), 'NULL'));

  DBMS_OUTPUT.PUT('OUT c = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(c), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('IN OUT d = ' || TO_CHAR(d));

  -- Can reference IN parameters a and b,
  -- but cannot assign values to them.

  c := a+10;  -- Assign value to OUT parameter
  d := 10/b;  -- Assign value to IN OUT parameter
END;
/
DECLARE
  aa  CONSTANT PLS_INTEGER := 1;
  bb  PLS_INTEGER  := 2;
  cc  PLS_INTEGER  := 3;
  dd  BINARY_FLOAT := 4;
  ee  PLS_INTEGER;
  ff  BINARY_FLOAT := 5;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');

  DBMS_OUTPUT.PUT('aa = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));

  DBMS_OUTPUT.PUT('bb = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));

  DBMS_OUTPUT.PUT('cc = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
```

```
  p (aa, -- constant
     bb, -- initialized variable
     cc, -- initialized variable
     dd  -- initialized variable
  );

  DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');

  DBMS_OUTPUT.PUT('aa = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));

  DBMS_OUTPUT.PUT('bb = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));

  DBMS_OUTPUT.PUT('cc = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');

  DBMS_OUTPUT.PUT('ee = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));

  p (1,        -- literal
     (bb+3)*4, -- expression
     ee,       -- uninitialized variable
     ff        -- initialized variable
   );

  DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');

  DBMS_OUTPUT.PUT('ee = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
END;
/
```

結果:

```
Before invoking procedure p:
aa = 1
bb = 2
cc = 3
dd = 4.0E+000
Inside procedure p:
IN a = 1
IN b = 2
OUT c = NULL
IN OUT d = 4.0E+000
After invoking procedure p:
aa = 1
bb = 2
cc = 11
```

```
dd = 5.0E+000
Before invoking procedure p:
ee = NULL
ff = 5.0E+000
Inside procedure p:
IN a = 1
IN b = 20
OUT c = NULL
IN OUT d = 5.0E+000
After invoking procedure p:
ee = 11
ff = 5.0E-001

PL/SQL procedure successfully completed.
```

例8-15 例外処理後のOUTおよびIN OUTパラメータの値

```
DECLARE
  j  PLS_INTEGER  := 10;
  k  BINARY_FLOAT := 15;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');

  DBMS_OUTPUT.PUT('j = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(j), 'NULL'));

  DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));

  p(4, 0, j, k);  -- causes p to exit with exception ZERO_DIVIDE

EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');

    DBMS_OUTPUT.PUT('j = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(j), 'NULL'));

    DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));
END;
/
```

結果:

```
Before invoking procedure p:
j = 10
k = 1.5E+001
Inside procedure p:
IN a = 4
IN b = 0
OUT c = NULL
IN OUT d = 1.5E+001
After invoking procedure p:
j = 10
k = 1.5E+001

PL/SQL procedure successfully completed.
```

例8-16 NULL以外のデフォルト値を持つレコード型のOUT仮パラメータ

```
CREATE OR REPLACE PACKAGE r_types AUTHID DEFINER IS
  TYPE r_type_1 IS RECORD (f VARCHAR2(5) := 'abcde');
  TYPE r_type_2 IS RECORD (f VARCHAR2(5));
END;
/

CREATE OR REPLACE PROCEDURE p (
  x OUT r_types.r_type_1,
  y OUT r_types.r_type_2,
  z OUT VARCHAR2)
AUTHID CURRENT_USER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('x.f is ' || NVL(x.f,'NULL'));
  DBMS_OUTPUT.PUT_LINE('y.f is ' || NVL(y.f,'NULL'));
  DBMS_OUTPUT.PUT_LINE('z is ' || NVL(z,'NULL'));
END;
/
DECLARE
  r1 r_types.r_type_1;
  r2 r_types.r_type_2;
  s  VARCHAR2(5) := 'fghij';
BEGIN
  p (r1, r2, s);
END;
/
```

結果:

```
x.f is abcde
y.f is NULL
z is NULL

PL/SQL procedure successfully completed.
```

8.7.4 サブプログラムのパラメータのエイリアシング

エイリアシングとは、同じメモリー位置に対して2つの異なる名前が存在することです。格納されている項目が複数のパスで表示可能な場合は、1つのパスでその項目を変更してからすべてのパスで変更を確認できます。

サブプログラムのパラメータのエイリアシングは、コンパイラが参照によって実パラメータを渡す場合に常に発生し、サブプログラムにカーソル変数パラメータが含まれる場合にも発生する可能性があります。

ここでのトピック

- 参照渡しパラメータによるサブプログラムのパラメータのエイリアシング
- カーソル変数パラメータによるサブプログラムのパラメータのエイリアシング

8.7.4.1 参照渡しパラメータによるサブプログラムのパラメータのエイリアシング

コンパイラが参照によって実パラメータを渡す場合、実パラメータと仮パラメータは同じメモリー位置を参照します。そのため、サブプログラムによって仮パラメータの値が変更されると、その変更は実パラメータに即座に反映されます。

コンパイラは常に参照によってINパラメータを渡しますが、サブプログラムはINパラメータに値を代入できないため、この場合のエイリアシングでは問題は発生しません。

コンパイラは、OUTまたはIN OUTパラメータにNOCOPYが指定されると、そのパラメータを参照によって渡す可能性があります。NOCOPYは単なるヒントで、コンパイラは、サブプログラムが起動されるたびに、NOCOPYに従うか無視するかを暗黙的に決定します。したがって、エイリアシングの発生の有無は起動ごとに異なる可能性があるため、サブプログラムの結果は予測不能です。たとえば:

- 実パラメータがグローバル変数である場合、仮パラメータに対する代入は、グローバル・パラメータに反映される可能性があります(例8-17を参照)。
- 同じ変数が2つの仮パラメータの実パラメータである場合、いずれかの仮パラメータに対する代入は、両方の仮パラメータに即座に反映される可能性があります(例8-18を参照)。
- 実パラメータがパッケージ変数である場合、仮パラメータまたはパッケージ変数に対する代入は、仮パラメータとパッケージ変数の両方に即座に反映される可能性があります。
- サブプログラムが未処理例外を戻して終了した場合、仮パラメータに対する代入は、実パラメータに反映される可能性があります。

関連項目:

コンパイラが常にNOCOPYを無視する事例は、「NOCOPY」を参照してください

例8-17では、プロシージャにIN OUT NOCOPY仮パラメータが含まれており、プロシージャはそのパラメータに値'aardvark'を代入します。無名ブロックは、グローバル変数に値'aardwolf'を代入し、そのグローバル変数をプロシージャに渡します。コンパイラがNOCOPYヒントに従う場合、グローバル変数の最終的な値は、'aardvark'です。コンパイラがNOCOPYヒントを無視する場合、グローバル変数の最終的な値は、'aardwolf'です。

例8-18では、プロシージャにINパラメータ、IN OUTパラメータおよびIN OUT NOCOPYパラメータが含まれます。無名ブロックは、3つのすべての仮パラメータに同じ実パラメータ(グローバル変数)を使用してプロシージャを起動します。プロシージャは、IN OUTパラメータの値を変更してからIN OUT NOCOPYパラメータの値を変更します。ただし、コンパイラがNOCOPYヒントに従う場合、後者の変更は、実パラメータに即座に反映されます。前者の変更は、プロシージャが正常に終了して制御が無名ブロックに戻された後に実パラメータに反映されます。

例8-17 実パラメータとしてのグローバル変数によるエイリアシング

```
DECLARE
  TYPE Definition IS RECORD (
    word      VARCHAR2(20),
    meaning   VARCHAR2(200)
  );
  TYPE Dictionary IS VARRAY(2000) OF Definition;
  lexicon  Dictionary := Dictionary();  -- global variable
  PROCEDURE add_entry (
    word_list IN OUT NOCOPY Dictionary  -- formal NOCOPY parameter
  ) IS
  BEGIN
    word_list(1).word := 'aardvark';
```

```
  END;
BEGIN
  lexicon.EXTEND;
  lexicon(1).word := 'aardwolf';
  add_entry(lexicon);  -- global variable is actual parameter
  DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
END;
/
```

結果:

```
aardvark
```

例8-18 複数の仮パラメータに対する同じ実パラメータによるエイリアシング

```
DECLARE
  n NUMBER := 10;
  PROCEDURE p (
    n1 IN NUMBER,
    n2 IN OUT NUMBER,
    n3 IN OUT NOCOPY NUMBER
  ) IS
  BEGIN
    n2 := 20;  -- actual parameter is 20 only after procedure succeeds
    DBMS_OUTPUT.put_line(n1);  -- actual parameter value is still 10
    n3 := 30;  -- might change actual parameter immediately
    DBMS_OUTPUT.put_line(n1);  -- actual parameter value is either 10 or 30
  END;
BEGIN
  p(n, n, n);
  DBMS_OUTPUT.put_line(n);
END;
/
```

コンパイラがNOCOPYヒントに従う場合の結果:

```
10
30
20
```

コンパイラがNOCOPYヒントを無視する場合の結果:

```
10
10
30
```

8.7.4.2 カーソル変数パラメータによるサブプログラムのパラメータのエイリアシング

カーソル変数パラメータは、ポインタです。したがって、サブプログラムがあるカーソル変数パラメータを別のカーソル変数パラメータに代入すると、それらのパラメータは同じメモリー位置を参照します。このエイリアシングでは、意図しない結果が発生することがあります。

例8-19では、プロシージャに2つのカーソル変数パラメータ(emp_cv1およびemp_cv2)が含まれます。プロシージャは、emp_cv1をオープンし、その値(ポインタ)をemp_cv2に代入します。この場合、emp_cv1とemp_cv2は、同じメモリー位置を参照します。プロ

シージャがemp_cv1をクローズすると、emp_cv2もクローズされます。そのため、プロシージャがemp_cv2からフェッチしようとすると、PL/SQLによって例外が呼び出されます。

例8-19 カーソル変数サブプログラム・パラメータによるエイリアシング

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR;
  c1 EmpCurTyp;
  c2 EmpCurTyp;
  PROCEDURE get_emp_data (
    emp_cv1 IN OUT EmpCurTyp,
    emp_cv2 IN OUT EmpCurTyp
  )
  IS
    emp_rec employees%ROWTYPE;
  BEGIN
    OPEN emp_cv1 FOR SELECT * FROM employees;
    emp_cv2 := emp_cv1;  -- now both variables refer to same location
    FETCH emp_cv1 INTO emp_rec;  -- fetches first row of employees
    FETCH emp_cv1 INTO emp_rec;  -- fetches second row of employees
    FETCH emp_cv2 INTO emp_rec;  -- fetches third row of employees
    CLOSE emp_cv1;  -- closes both variables
    FETCH emp_cv2 INTO emp_rec; -- causes error when get_emp_data is invoked
  END;
BEGIN
  get_emp_data(c1, c2);
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-01001: invalid cursor
ORA-06512: at line 19
ORA-06512: at line 22
```

8.7.5 サブプログラムのINパラメータのデフォルト値

IN仮パラメータを宣言する場合、そのパラメータにデフォルト値を指定できます。デフォルト値を持つ仮パラメータは、その対応する実パラメータがサブプログラムの起動においてオプションであるため、オプション・パラメータと呼ばれます。実パラメータが省略されると、起動によって仮パラメータにデフォルト値が代入されます。デフォルト値を持たない仮パラメータは、その対応する実パラメータがサブプログラムの起動において必須であるため、必須パラメータと呼ばれます。

実パラメータを省略しても、対応する仮パラメータの値はNULLになりません。仮パラメータの値をNULLにするには、デフォルト値または実パラメータとしてNULLを指定します。

例8-20では、プロシージャに1つの必須パラメータと2つのオプション・パラメータが含まれます。

例8-20では、プロシージャの起動によって、実パラメータが、それと対応する仮パラメータの宣言と同じ順序で指定されます(起動では位置表記法が使用されます)。位置表記法では、raise_salaryの2番目のパラメータを省略せずに、3番目のパラメータを指定します(途中のパラメータを省略するには、名前表記法または混合表記法を使用する必要があります)。詳細は、実パ

ラメータの位置表記法、名前表記法および混合表記法を参照してください。

仮パラメータのデフォルト値には、パラメータに値を代入できる任意の式を指定できます(値およびパラメータのデータ型には互換性が必要です)。サブプログラムの起動で仮パラメータに実パラメータが指定されると、その起動ではデフォルト値が評価されません。

例8-21では、プロシージャpに、ファンクションfを起動するデフォルト値を持つパラメータが含まれます。ファンクションfは、グローバル変数の値を増分します。pが実パラメータなしで起動されると、pはfを起動し、fはグローバル変数を増分します。pが実パラメータ付きで起動されると、pはfを起動せず、グローバル変数の値は変化しません。

例8-22では、2つの必須パラメータが含まれるプロシージャを作成してそれを起動し、その後、3番目のオプション・パラメータを追加しています。3番目のパラメータはオプションであるため、元の起動も引き続き有効です。

例8-20 デフォルトのパラメータ値を使用したプロシージャ

```
DECLARE
  PROCEDURE raise_salary (
    emp_id IN employees.employee_id%TYPE,
    amount IN employees.salary%TYPE := 100,
    extra  IN employees.salary%TYPE := 50
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount + extra
    WHERE employee_id = emp_id;
  END raise_salary;

BEGIN
  raise_salary(120);       -- same as raise_salary(120, 100, 50)
  raise_salary(121, 200);  -- same as raise_salary(121, 200, 50)
END;
/
```

例8-21 ファンクションによるデフォルトのパラメータ値の提供

```
DECLARE
  global PLS_INTEGER := 0;

  FUNCTION f RETURN PLS_INTEGER IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside f.');
    global := global + 1;
    RETURN global * 2;
  END f;

  PROCEDURE p (
    x IN PLS_INTEGER := f()
  ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (
      'Inside p. ' ||
      '  global = ' || global ||
      ', x = ' || x || '.'
    );
    DBMS_OUTPUT.PUT_LINE('--------------------------------');
```

```
  END p;

  PROCEDURE pre_p IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (
     'Before invoking p,  global = ' || global || '.'
    );
    DBMS_OUTPUT.PUT_LINE('Invoking p.');
  END pre_p;

BEGIN
  pre_p;
  p();     -- default expression is evaluated

  pre_p;
  p(100);  -- default expression is not evaluated

  pre_p;
  p();     -- default expression is evaluated
END;
/
```

結果:

```
Before invoking p,  global = 0.
Invoking p.
Inside f.
Inside p.   global = 1, x = 2.
--------------------------------
Before invoking p,  global = 1.
Invoking p.
Inside p.   global = 1, x = 100.
--------------------------------
Before invoking p,  global = 1.
Invoking p.
Inside f.
Inside p.   global = 2, x = 4.
--------------------------------
```

例8-22 既存の起動を変更しないサブプログラム・パラメータの追加

プロシージャの作成:

```
CREATE OR REPLACE PROCEDURE print_name (
  first VARCHAR2,
  last VARCHAR2
) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(first || ' ' || last);
END print_name;
/
```

プロシージャの起動:

```
BEGIN
  print_name('John', 'Doe');
END;
```

```
/
```

結果:

```
John Doe
```

デフォルト値を持つ3番目のパラメータの追加:

```
CREATE OR REPLACE PROCEDURE print_name (
  first VARCHAR2,
  last VARCHAR2,
  mi   VARCHAR2 := NULL
) AUTHID DEFINER IS
BEGIN
  IF mi IS NULL THEN
    DBMS_OUTPUT.PUT_LINE(first || ' ' || last);
  ELSE
    DBMS_OUTPUT.PUT_LINE(first || ' ' || mi || '. ' || last);
  END IF;
END print_name;
/
```

プロシージャの起動:

```
BEGIN
  print_name('John', 'Doe');          -- original invocation
  print_name('John', 'Public', 'Q');  -- new invocation
END;
/
```

結果:

```
John Doe
John Q. Public
```

8.7.6 実パラメータの位置表記法、名前表記法および混合表記法

サブプログラムの起動時に、位置表記法、名前表記法または混合表記法のいずれかを使用して実パラメータを指定できます。表8-3に、これらの表記法の概要および比較を示します。

表8-3 PL/SQLの実パラメータの表記法

表記法	構文	オプションのパラメータ	メリット	デメリット
位置	仮パラメータが宣言されている順序と同じ順序で実パラメータを指定します。	後続のオプション・パラメータを省略できます。		実パラメータを間違った順序で指定すると(特に実パラメータがリテラルの場合)、検出が困難な問題が発生する可能性があります。

表記法	構文	オプションのパラメータ	メリット	デメリット
				仮パラメータのリストが変更される場合、そのリストで新しい後続のオプション・パラメータのみが取得されるのでなければ、サブプログラムの起動を変更する必要があります(例 8-22 を参照)。 コードの明瞭性とメンテナンス性が下がります。サブプログラムに多数のパラメータがある場合はお薦めしません。
名前	次の構文を使用して、任意の順序で実パラメータを指定します。 `formal => actual` `formal` は仮パラメータの名前で、`actual` は実パラメータです。	任意のオプション・パラメータを省略できます。	実パラメータを指定する場合の間違った順序はありません。 仮パラメータのリストで新しい必須パラメータが取得される場合にのみ、サブプログラムの起動を変更する必要があります。 自分以外のユーザーによって定義または管理されているサブプログラムを起動する場合に推奨されます。	
混合	まず位置表記法を使用し、その後、残りのパラメータに名前表記法を使用します。	位置表記法では、後続のオプション・パラメータを省略できます。名前表記法では、任意のオプション・パラメータを省略できます。	必須パラメータの後にオプション・パラメータが続くサブプログラムを起動し、オプション・パラメータを数個のみ指定する必要がある場合に便利です。	位置表記法では、間違った順序で指定すると(特に実パラメータがリテラルの場合)、検出が困難な問題が発生する可能性があります。 仮パラメータのリストが変更される場合、位置表記法の変更が必要になる可能性があります。

例8-23では、プロシージャの起動で異なる表記法を使用していますが、同じ意味です。

例8-24では、異なる表記法による同等の起動を使用して、SQLのSELECT文でPL/SQLファンクションcompute_bonusを起動しています。

例8-23 無名ブロックでの異なる表記法を使用した同等な起動

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6) := 50;
  PROCEDURE raise_salary (
    emp_id NUMBER,
    amount NUMBER
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;
  END raise_salary;
BEGIN
  -- Equivalent invocations:
  raise_salary(emp_num, bonus);                      -- positional notation
  raise_salary(amount => bonus, emp_id => emp_num);  -- named notation
  raise_salary(emp_id => emp_num, amount => bonus);  -- named notation
  raise_salary(emp_num, amount => bonus);            -- mixed notation
END;
/
```

例8-24 SELECT文での異なる表記法を使用した同等な起動

```
CREATE OR REPLACE FUNCTION compute_bonus (
  emp_id NUMBER,
  bonus NUMBER
) RETURN NUMBER
  AUTHID DEFINER
IS
  emp_sal NUMBER;
BEGIN
  SELECT salary INTO emp_sal
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emp_sal + bonus;
END compute_bonus;
/
SELECT compute_bonus(120, 50) FROM DUAL;                  -- positional
SELECT compute_bonus(bonus => 50, emp_id => 120) FROM DUAL; -- named
SELECT compute_bonus(120, bonus => 50) FROM DUAL;           -- mixed
```

8.8 サブプログラムの起動の解決

PL/SQLコンパイラは、サブプログラムの起動を検出すると、まず現在の有効範囲内で一致するサブプログラムの宣言を検索し、次に必要に応じて連続する外側の有効範囲内を検索します。

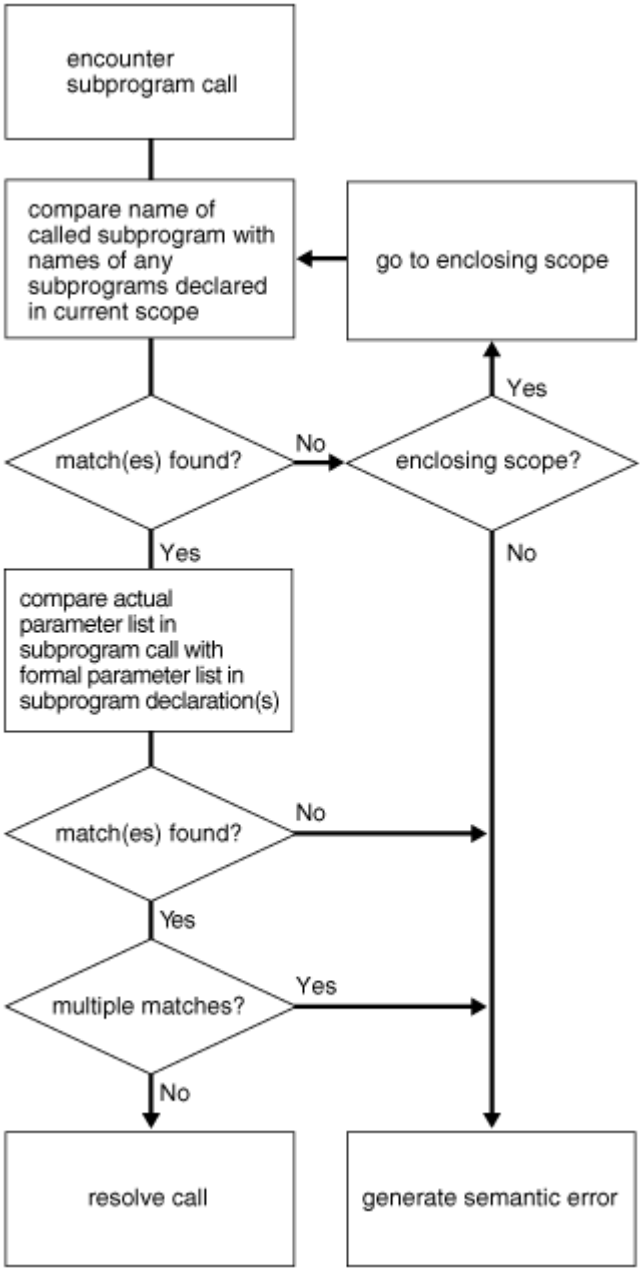
サブプログラムの名前およびパラメータ・リストが一致した場合に、宣言と起動は一致します。パラメータ・リストは、宣言内の必

須仮パラメータがそれぞれ起動内で対応する実パラメータを持つ場合に一致します。

コンパイラが起動に対して一致する宣言を見つけられない場合、セマンティック・エラーが生成されます。

図8-1に、PL/SQLコンパイラがサブプログラムの起動を解決する方法を示します。

図8-1 PL/SQLコンパイラによる起動の解決方法



例8-25では、ファンクションbalanceが、適切な実パラメータを使用して、外側のプロシージャswapを起動しようとしています。ただし、balanceには、swapという2つのネストしたプロシージャがありますが、どちらにも外側のプロシージャswapと同じ型のパラメータが存在しません。このため、この起動によって、コンパイル・エラーPLS-00306が発生します。

例8-25 PL/SQLプロシージャ名の解決

```
DECLARE
  PROCEDURE swap (
    n1 NUMBER,
    n2 NUMBER
  )
  IS
```

```
    num1 NUMBER;
    num2 NUMBER;
    FUNCTION balance
      (bal NUMBER)
      RETURN NUMBER
    IS
      x NUMBER := 10;
      PROCEDURE swap (
        d1 DATE,
        d2 DATE
      ) IS
      BEGIN
        NULL;
      END;
      PROCEDURE swap (
        b1 BOOLEAN,
        b2 BOOLEAN
      ) IS
      BEGIN
        NULL;
      END;
    BEGIN  -- balance
      swap(num1, num2);
      RETURN x;
    END balance;
  BEGIN  -- enclosing procedure swap
    NULL;
  END swap;
BEGIN  -- anonymous block
  NULL;
END;   -- anonymous block
/
```

結果:

```
      swap(num1, num2);
      *
ERROR at line 33:
ORA-06550: line 33, column 7:
PLS-00306: wrong number or types of arguments in call to 'SWAP'
ORA-06550: line 33, column 7:
PL/SQL: Statement ignored
```

8.9 オーバーロードされたサブプログラム

PL/SQLでは、ネストしたサブプログラム、パッケージ・サブプログラムおよび型のメソッドをオーバーロードできます。仮パラメータの名前、数、順序またはデータ型のファミリが異なっている場合、同じ名前を複数のサブプログラムで使用できます。(データ型ファミリは、データ型とそのサブタイプです。事前定義のPL/SQLデータ型のデータ型ファミリは、「PL/SQLの事前定義のデータ型」を参照してください。ユーザー定義のPL/SQLサブタイプの詳細は、「ユーザー定義のPL/SQLサブタイプ」を参照してください。)仮パラメータの違いが名前のみである場合、名前表記法を使用して対応する実パラメータを指定する必要があります。(名前表記法の詳細は、実パラメータの位置表記法、名前表記法および混合表記法を参照してください。)

例8-26では、同じ名前(initialize)を持つ2つのサブプログラムを定義しています。プロシージャでは様々な型のコレクションを

初期化します。これらのプロシージャは同じ処理を実行しているため、同じ名前を与えるのが論理的です。

この2つのinitializeプロシージャは、同じブロック、サブプログラム、パッケージまたは型本体の中に置くことができます。PL/SQLは仮パラメータをチェックして、どちらのプロシージャを起動するかを判断します。PL/SQLが使用するinitializeのバージョンは、プロシージャをdate_tab_typパラメータまたはnum_tab_typパラメータのどちらで起動するかによって異なります。

パッケージ内のオーバーロードされたプロシージャの例は、例10-9を参照してください。

ここでのトピック

- 数値データ型のみが異なる仮パラメータ
- オーバーロードできないサブプログラム
- サブプログラムのオーバーロード・エラー

例8-26 オーバーロードされたサブプログラム

```
DECLARE
  TYPE date_tab_typ IS TABLE OF DATE   INDEX BY PLS_INTEGER;
  TYPE num_tab_typ  IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  hiredate_tab  date_tab_typ;
  sal_tab       num_tab_typ;
  PROCEDURE initialize (tab OUT date_tab_typ, n INTEGER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Invoked first version');
    FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
    END LOOP;
  END initialize;
  PROCEDURE initialize (tab OUT num_tab_typ, n INTEGER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Invoked second version');
    FOR i IN 1..n LOOP
      tab(i) := 0.0;
    END LOOP;
  END initialize;
BEGIN
  initialize(hiredate_tab, 50);
  initialize(sal_tab, 100);
END;
/
```

結果:

```
Invoked first version
Invoked second version
```

8.9.1 数値データ型のみが異なる仮パラメータ

サブプログラムの仮パラメータの違いが数値データ型のみの場合、それらのサブプログラムはオーバーロードできます。ファンクションの複数のバージョンが同じ名前を使用でき、それぞれが異なる数値型を受け取ることができるため、この手法は数値演算Application Programming Interface(API)を記述する場合に有効です。たとえば、BINARY_FLOATを受け取るファンクションはより高速で、BINARY_DOUBLEを受け取るファンクションは精度がより高くなる場合があります。

オーバーロードされたサブプログラムにパラメータを渡す場合、次のことに注意して、問題または予期しない結果を回避します。

- 予期されるパラメータ・セットごとに、目的のバージョンのサブプログラムが起動されることを確認します。

  たとえば、オーバーロードされるファンクションがBINARY_FLOATおよびBINARY_DOUBLEを受け取る場合、「5.0」のようなVARCHAR2リテラルを渡すと、どちらのファンクションが起動されるかを確認します。

- 数値リテラルを修飾したり、変換ファンクションを使用して、目的のパラメータ型を明示します。

  たとえば、5.0f(BINARY_FLOAT)、5.0d(BINARY_DOUBLE)のようなリテラル、またはTO_BINARY_FLOAT、TO_BINARY_DOUBLE、TO_NUMBERのような変換ファンクションを使用します。

PL/SQLは、次の順序で、一致する数値パラメータを検索します。

1. PLS_INTEGER(または同じデータ型であるBINARY_INTEGER)
2. NUMBER
3. BINARY_FLOAT
4. BINARY_DOUBLE

VARCHAR2値は、NUMBER、BINARY_FLOATまたはBINARY_DOUBLEパラメータに一致します。

PL/SQLは、指定されたパラメータに最初に一致してオーバーロードされるサブプログラムを使用します。たとえば、SQRTファンクションは1つのパラメータを受け取ります。NUMBER、BINARY_FLOATまたはBINARY_DOUBLEパラメータを受け取るオーバーロードされるファンクションがあるとします。PLS_INTEGERパラメータを渡した場合、最初に一致してオーバーロードされるファンクションは、NUMBERパラメータを持つファンクションです。

NUMBERパラメータを受け取るSQRTファンクションは、最も低速となる可能性が高くなります。より高速なバージョンを使用するには、パラメータをSQRTファンクションに渡す前に、TO_BINARY_FLOATまたはTO_BINARY_DOUBLEファンクションを使用して、そのパラメータを別のデータ型に変換します。

PL/SQLは、パラメータを別のデータ型に変換する必要がある場合、まずそのパラメータをより上位のデータ型に変換しようとします。たとえば:

- ATAN2ファンクションは、同じ型の2つのパラメータを受け取ります。異なる型のパラメータを渡した場合(たとえば、1つがPLS_INTEGERで、1つがBINARY_FLOATの場合)、PL/SQLは、両方のパラメータでより高度な型が使用されている場合に一致するものを検索します。この例では、2つのBINARY_FLOATパラメータを受け取るバージョンのATAN2が使用され、PLS_INTEGERパラメータは、上位変換されます。
- あるファンクションは、異なる型の2つのパラメータを受け取ります。オーバーロードされるバージョンの1つは、PLS_INTEGERパラメータおよびBINARY_FLOATパラメータを受け取ります。別のオーバーロードされるバージョンは、NUMBERパラメータおよびBINARY_DOUBLEパラメータを受け取ります。このファンクションを起動して、2つのNUMBERパラメータを渡した場合、PL/SQLは、まず2つ目のパラメータがBINARY_FLOATのオーバーロードされるバージョンを検出します。このパラメータは、他方のオーバーロードされるバージョンのBINARY_DOUBLEパラメータよりも一致度が高いため、次にPL/SQLは下位検索し、1つ目のNUMBERパラメータをPLS_INTEGERに変換します。

8.9.2 オーバーロードできないサブプログラム

次のサブプログラムはオーバーロードできません。

- スタンドアロン・サブプログラム

- 仮パラメータの違いがモードのみのサブプログラム。たとえば:

```
PROCEDURE s (p IN  VARCHAR2) IS ...
PROCEDURE s (p OUT VARCHAR2) IS ...
```

- 仮パラメータの違いがサブタイプのみのサブプログラム。たとえば:

```
PROCEDURE s (p INTEGER) IS ...
PROCEDURE s (p REAL) IS ...
```

INTEGERおよびREALは、NUMBERのサブタイプであるため、同じデータ型のファミリに属しています。

- 戻り値のデータ型のみが異なるファンクション(そのデータ型のファミリが異なっている場合でも)。たとえば:

```
FUNCTION f (p INTEGER) RETURN BOOLEAN IS ...
FUNCTION f (p INTEGER) RETURN INTEGER IS ...
```

8.9.3 サブプログラムのオーバーロード・エラー

PL/SQLコンパイラは、起動されたサブプログラムを判別できないと判断するとすぐに、オーバーロード・エラーを捕捉します。複数のサブプログラムに同一のヘッダーがある場合、サブプログラム自体のコンパイル(サブプログラムがネストされている場合)、またはサブプログラムを宣言しているパッケージ仕様部のコンパイルを試行すると、コンパイラはオーバーロード・エラーを捕捉します。それ以外の場合、サブプログラムの曖昧な起動のコンパイルを試行すると、コンパイラはエラーを捕捉します。

例8-27に示す、同一のヘッダーを持つ複数のサブプログラムを宣言しているパッケージ仕様部をコンパイルしようとすると、コンパイル時エラーPLS-00305が発生します。

例8-28に示すパッケージ仕様部は、仮パラメータの違いがサブタイプのみのサブプログラムはオーバーロードできないという規則に違反していますが、エラーを生成せずにコンパイルできます。

ただし、例8-29に示すようにpkg2.sの起動をコンパイルしようとすると、コンパイル時エラーPLS-00307が発生します。

例8-30のように、オーバーロードされたサブプログラムの仮パラメータに別の名前を付けることによって、例8-28に示されているオーバーロード・エラーを修正するとします。

これによって、名前表記法を使用して実パラメータを指定する場合に、エラーを生成せずにpkg2.sの起動をコンパイルできます(例8-31を参照)。(例8-29に示されているように、位置表記法を使用して実パラメータを指定すると、コンパイル時エラーPLS-00307が発生します。)

例8-32に示すパッケージ仕様部は、オーバーロード規則に違反していないため、エラーを生成せずにコンパイルできます。ただし、オーバーロードされたプロシージャを起動すると、例8-33の2つ目の起動に示すように、コンパイル時エラーPLS-00307が発生します。

どのサブプログラムが起動されたかを特定しようとしたとき、PL/SQLコンパイラは、暗黙的にあるパラメータを一致するタイプに変換すると、暗黙的に一致するタイプに変換できる別のパラメータを検索します。複数の一致がある場合、例8-34に示すように、コンパイル時エラーPLS-00307が発生します。

例8-27 コンパイル時エラーの原因となるオーバーロード・エラー

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER IS
  PROCEDURE s (p VARCHAR2);
```

```
  PROCEDURE s (p VARCHAR2);
END pkg1;
/
```

例8-28 正常にコンパイルが行われるオーバーロード・エラー

```
CREATE OR REPLACE PACKAGE pkg2 AUTHID DEFINER IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p t1);
  PROCEDURE s (p t2);
END pkg2;
/
```

例8-29 コンパイル時エラーの原因となる例8-28のサブプログラムの起動

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(a);  -- Causes compile-time error PLS-00307
END p;
/
```

例8-30 例8-28のオーバーロード・エラーの修正

```
CREATE OR REPLACE PACKAGE pkg2 AUTHID DEFINER IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p1 t1);
  PROCEDURE s (p2 t2);
END pkg2;
/
```

例8-31 例8-30のサブプログラムの起動

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(p1=>a);  -- Compiles without error
END p;
/
```

例8-32 オーバーロード・エラーが含まれていないパッケージ仕様部

```
CREATE OR REPLACE PACKAGE pkg3 AUTHID DEFINER IS
  PROCEDURE s (p1 VARCHAR2);
  PROCEDURE s (p1 VARCHAR2, p2 VARCHAR2 := 'p2');
END pkg3;
/
```

例8-33 適切にオーバーロードされたサブプログラムの不適切な起動

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  a1 VARCHAR2(10) := 'a1';
  a2 VARCHAR2(10) := 'a2';
BEGIN
  pkg3.s(p1=>a1, p2=>a2);  -- Compiles without error
```

```
  pkg3.s(p1=>a1);          -- Causes compile-time error PLS-00307
END p;
/
```

例8-34 パラメータの暗黙的な変換によるオーバーロード・エラー

```
CREATE OR REPLACE PACKAGE pack1 AUTHID DEFINER AS
  PROCEDURE proc1 (a NUMBER, b VARCHAR2);
  PROCEDURE proc1 (a NUMBER, b NUMBER);
END;
/
CREATE OR REPLACE PACKAGE BODY pack1 AS
  PROCEDURE proc1 (a NUMBER, b VARCHAR2) IS BEGIN NULL; END;
  PROCEDURE proc1 (a NUMBER, b NUMBER) IS BEGIN NULL; END;
END;
/
BEGIN
  pack1.proc1(1,'2');     -- Compiles without error
  pack1.proc1(1,2);       -- Compiles without error
  pack1.proc1('1','2');   -- Causes compile-time error PLS-00307
  pack1.proc1('1',2);     -- Causes compile-time error PLS-00307
END;
/
```

8.10 再帰的サブプログラム

再帰的サブプログラムは、自分自身を起動します。再帰はアルゴリズムを単純化する強力な手法です。

再帰的サブプログラムには、2つ以上の実行パス(再帰的起動に導く1つのパスと終了条件に導く1つのパス)が必要です。後者が存在しない場合、メモリーが足りなくなり、PL/SQLによって事前定義の例外STORAGE_ERRORが呼び出されるまで、再帰が続きます。

例8-35では、ファンクションを使用してnの階乗(n!、1からnまでのすべての整数の積)に関する次の再帰定義を実装しています。

```
n! = n * (n - 1)!
```

例8-36では、ファンクションを使用してn番目のフィボナッチ数(n-1番目とn-2番目のフィボナッチ数の和)を戻しています。1番目と2番目のフィボナッチ数は、それぞれ0(ゼロ)と1です。

> ノート:
>
> 例 8-36 のファンクションは、結果キャッシュの対象に適したファンクションです。詳細は、「結果がキャッシュされる再帰ファンクション」を参照してください。

サブプログラムの再帰的起動が行われるたびに、サブプログラムで宣言されている各項目のインスタンスと、実行される各SQL文が作成されます。

カーソルFOR LOOP文の内側か、OPENまたはOPEN FOR文とCLOSE文の間にある再帰的起動は、起動ごとに新しいカーソルをオープンします。そのため、オープン・カーソルの数が、データベース初期化パラメータOPEN_CURSORSで設定されている制限を超える

可能性があります。

例8-35 nの階乗(n!)を戻す再帰ファンクション

```
CREATE OR REPLACE FUNCTION factorial (
  n POSITIVE
) RETURN POSITIVE
  AUTHID DEFINER
IS
BEGIN
  IF n = 1 THEN                 -- terminating condition
    RETURN n;
  ELSE
    RETURN n * factorial(n-1);  -- recursive invocation
  END IF;
END;
/
BEGIN
  FOR i IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i));
  END LOOP;
END;
/
```

結果:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

例8-36 n番目のフィボナッチ数を戻す再帰ファンクション

```
CREATE OR REPLACE FUNCTION fibonacci (
  n PLS_INTEGER
) RETURN PLS_INTEGER
  AUTHID DEFINER
IS
  fib_1 PLS_INTEGER := 0;
  fib_2 PLS_INTEGER := 1;
BEGIN
  IF n = 1 THEN                             -- terminating condition
    RETURN fib_1;
  ELSIF n = 2 THEN
    RETURN fib_2;                          -- terminating condition
  ELSE
    RETURN fibonacci(n-2) + fibonacci(n-1);  -- recursive invocations
  END IF;
END;
/
BEGIN
  FOR i IN 1..10 LOOP
    DBMS_OUTPUT.PUT(fibonacci(i));
    IF i < 10 THEN
      DBMS_OUTPUT.PUT(', ');
    END IF;
  END LOOP;
```

```
  DBMS_OUTPUT.PUT_LINE(' ...');
END;
/
```

結果:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...
```

8.11 サブプログラムの副作用

サブプログラムは、独自のローカル変数の値以外のものを変更する場合、副作用を伴います。たとえば、次のいずれかを変更するサブプログラムには、副作用があります。

- 独自のOUTまたはIN OUTパラメータ
- グローバル変数
- パッケージ内のパブリック変数
- データベース表
- データベース
- 外部の状態(たとえば、DBMS_OUTPUTの起動や電子メールの送信などによる変更)

副作用によって、問合せのパラレル処理が妨害されたり、処理順序に依存する(したがって、不確定な)結果が発生したり、ユーザー・セッションにまたがったパッケージ状態のメンテナンスが必要になります。

副作用を最小限に抑えることは、結果がキャッシュされるファンクションまたはストアド・ファンクションを定義してSQL文を起動する場合に特に重要です。

関連項目:

SQL文から起動されたPL/SQLファンクションでの副作用の制御の詳細は、『Oracle Database開発ガイド』を参照してください。

8.12 PL/SQLファンクション結果キャッシュ

PL/SQLファンクションにRESULT_CACHEオプションが指定されていると、その結果は共有グローバル領域(SGA)にキャッシュされるようになります。これにより、同じインスタンスに接続しているセッションは、キャッシュされた結果を再使用できるようになります(結果が利用可能な場合)。

Oracle Databaseでは、結果がキャッシュされるファンクションの実行中に問合せが行われるすべてのデータ・ソース(表およびビュー)が自動的に検出されます。これらのデータ・ソースのいずれかに対する変更がコミットされると、キャッシュされた結果は、すべてのインスタンスに渡って無効になります。結果キャッシュの対象として最良のファンクションは、頻繁に起動され、ほとんどまたはまったく変更されない情報に依存するファンクションです。

ここでのトピック

- ファンクションの結果キャッシュの有効化
- 結果がキャッシュされるファンクションを使用するアプリケーションの開発
- 結果がキャッシュされるファンクションの要件
- 結果がキャッシュされるファンクションの例
- 結果がキャッシュされるファンクションの高度なトピック

8.12.1 ファンクションの結果キャッシュの有効化

ファンクションの結果がキャッシュされるようにするには、ファンクションの宣言と定義にRESULT_CACHE句を含めます。構文の詳細は、「ファンクションの宣言および定義」を参照してください。

> ノート:
>
> データベース・サーバーの結果キャッシュの構成と管理の詳細は、『Oracle Database リファレンス』および『Oracle Database パフォーマンス・チューニング・ガイド』を参照してください。

例8-37では、パッケージdepartment_pkgが、結果がキャッシュされるファンクションget_dept_infoを宣言し、定義しています(このファンクションによって、指定した部門に関する情報のレコードが戻されます)。このファンクションは、データベース表のDEPARTMENTSおよびEMPLOYEESに依存します。

ファンクションget_dept_infoは、他のファンクションを起動する場合と同様に起動します。たとえば、次の起動では、部門番号10に関する情報のレコードが戻されます。

```
department_pkg.get_dept_info(10);
```

次の起動では、部門番号10の名前のみが戻されます。

```
department_pkg.get_dept_info(10).dept_name;
```

get_dept_info(10)の結果が結果キャッシュに含まれている場合、結果はこのキャッシュから戻されますが、そうでない場合は、結果は計算されてキャッシュに追加されます。get_dept_infoはDEPARTMENTS表およびEMPLOYEES表に依存するため、DEPARTMENTSまたはEMPLOYEESへの変更がコミットされると、get_dept_infoのキャッシュされた結果はすべて無効になり、DEPARTMENTSまたはEMPLOYEESが変更される可能性のあるすべての場所で、キャッシュ無効化ロジックをプログラミングする必要がなくなります。

例8-37 結果がキャッシュされるファンクションの宣言および定義

```
CREATE OR REPLACE PACKAGE department_pkg AUTHID DEFINER IS

  TYPE dept_info_record IS RECORD (
    dept_name   departments.department_name%TYPE,
    mgr_name    employees.last_name%TYPE,
    dept_size   PLS_INTEGER
  );

  -- Function declaration
```

```
  FUNCTION get_dept_info (dept_id NUMBER)
    RETURN dept_info_record
    RESULT_CACHE;

END department_pkg;
/
CREATE OR REPLACE PACKAGE BODY department_pkg IS
  -- Function definition
  FUNCTION get_dept_info (dept_id NUMBER)
    RETURN dept_info_record
    RESULT_CACHE
  IS
    rec  dept_info_record;
  BEGIN
    SELECT department_name INTO rec.dept_name
    FROM departments
    WHERE department_id = dept_id;

    SELECT e.last_name INTO rec.mgr_name
    FROM departments d, employees e
    WHERE d.department_id = dept_id
    AND d.manager_id = e.employee_id;

    SELECT COUNT(*) INTO rec.dept_size
    FROM EMPLOYEES
    WHERE department_id = dept_id;

    RETURN rec;
  END get_dept_info;
END department_pkg;
/
```

8.12.2 結果がキャッシュされるファンクションを使用するアプリケーションの開発

結果がキャッシュされるファンクションを使用するアプリケーションを開発する場合、指定したパラメータ値のセットに対してそのファンクションの本体が実行される回数については何も想定しないでください。

結果がキャッシュされるファンクションの本体が実行される状況をいくつか次に示します。

- このデータベース・インスタンスでのセッションが、これらのパラメータ値を使用してファンクションを初めて起動したとき
- これらのパラメータ値のキャッシュされた結果が無効である場合

  ファンクションが依存するいずれかのデータソースに対する変更がコミットされると、キャッシュされている結果が無効になります。
- これらのパラメータ値のキャッシュされた結果がエージ・アウトされた場合

  システムでは、必要なメモリーが不足すると、キャッシュされた値で最も古いものが破棄されます。
- ファンクションがキャッシュをバイパスする場合(「結果キャッシュのバイパス」を参照)

8.12.3 結果がキャッシュされるファンクションの要件

結果がキャッシュされるPL/SQLファンクションは、あらゆる入力に対して生成される出力が、RESULT_CACHEのマークが付けられていなかった場合に生成される出力と常に同じになるときに安全です。この安全性は、次に示す条件が満たされている場合に

のみ保証されます。

- ファンクションの実行時に、副作用がないこと。

  副作用の詳細は、「サブプログラムの副作用」を参照してください。

- ファンクションがアクセスするすべての表が、そのファンクションと同じデータベース内に存在する、通常の非SYS所有の永続的な表であること。

- ファンクションの結果は、常に、そのファンクションが参照する表の現在のSCNでコミットされた内容とあわせた、入力実績のベクトルによってのみ決定されること。

結果がキャッシュされるファンクションでは、次の条件も満たされていることがお薦めされています。

- セッション固有の設定に依存しない。

  詳細は、「結果がキャッシュされるファンクションによるセッション固有の設定の処理」を参照してください。

- セッション固有のアプリケーション・コンテキストに依存しない。

  詳細は、「結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理」を参照してください。

詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。

8.12.4 結果がキャッシュされるファンクションの例

結果キャッシュの対象として最良のファンクションは、(最初の例がこれに該当している可能性がありますが)頻繁に起動され、ほとんど変更されない情報に依存するファンクションです。結果キャッシュを行うことによって、再帰ファンクションでの冗長計算が回避されます。

例:

- 結果がキャッシュされるアプリケーション構成パラメータ
- 結果がキャッシュされる再帰ファンクション

8.12.4.1 結果がキャッシュされるアプリケーション構成パラメータ

グローバル・レベル、アプリケーション・レベルまたはロール・レベルのいずれのレベルで設定できる構成パラメータを持つアプリケーションについて考えてみます。このアプリケーションは、構成情報を次の表に格納します。

```
-- Global Configuration Settings
DROP TABLE global_config_params;
CREATE TABLE global_config_params
  (name  VARCHAR2(20), -- parameter NAME
   val   VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (name)
  );
-- Application-Level Configuration Settings
CREATE TABLE app_level_config_params
  (app_id  VARCHAR2(20), -- application ID
   name    VARCHAR2(20), -- parameter NAME
   val     VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (app_id, name)
  );
-- Role-Level Configuration Settings
```

```
CREATE TABLE role_level_config_params
  (role_id  VARCHAR2(20), -- application (role) ID
   name     VARCHAR2(20),  -- parameter NAME
   val      VARCHAR2(20),  -- parameter VALUE
   PRIMARY KEY (role_id, name)
  );
```

各構成パラメータで、ロール・レベルの設定はアプリケーション・レベルの設定をオーバーライドし、アプリケーション・レベルの設定はグローバル設定をオーバーライドします。パラメータに適用される設定を決定するために、このアプリケーションはPL/SQLファンクションget_valueを定義します。パラメータ名、アプリケーションIDおよびロールIDを指定すると、get_valueはそのパラメータに適用される設定を戻します。

ファンクションget_valueが頻繁に起動され、構成情報はほとんど変更されない場合、このファンクションは結果キャッシュの対象として最良のファンクションとなります。

例8-38に、get_valueに指定可能な定義を示します。あるパラメータ値のセットに対して、グローバル設定によってget_valueの結果が決まるとします。get_valueの実行中に、データベースでは3つの表role_level_config_params、app_level_config_paramsおよびglobal_config_paramsの問合せが検出されます。これらの3つの表のいずれに対する変更がコミットされても、このパラメータ値のセットについてキャッシュされた結果は無効となり、再計算する必要があります。

次に、2番目のパラメータ値セットに対して、ロール・レベルの設定によってget_valueの結果が決定されるとします。get_valueの実行中に、データベースでは表role_level_config_paramsの問合せのみが検出されます。role_level_config_paramsに対する変更がコミットされると、2番目のパラメータ値セットについてキャッシュされた結果は無効となりますが、app_level_config_paramsまたはglobal_config_paramsに対する変更がコミットされてもキャッシュされた結果には影響がありません。

例8-38 構成パラメータの設定を戻す、結果がキャッシュされるファンクション

```
CREATE OR REPLACE FUNCTION get_value
  (p_param VARCHAR2,
   p_app_id  NUMBER,
   p_role_id NUMBER
  )
  RETURN VARCHAR2
  RESULT_CACHE
  AUTHID DEFINER
IS
  answer VARCHAR2(20);
BEGIN
  -- Is parameter set at role level?
  BEGIN
    SELECT val INTO answer
      FROM role_level_config_params
        WHERE role_id = p_role_id
          AND name = p_param;
    RETURN answer;  -- Found
    EXCEPTION
      WHEN no_data_found THEN
        NULL;  -- Fall through to following code
  END;
  -- Is parameter set at application level?
  BEGIN
```

```
    SELECT val INTO answer
      FROM app_level_config_params
        WHERE app_id = p_app_id
          AND name = p_param;
    RETURN answer;  -- Found
    EXCEPTION
      WHEN no_data_found THEN
        NULL;  -- Fall through to following code
  END;
  -- Is parameter set at global level?
    SELECT val INTO answer
     FROM global_config_params
      WHERE name = p_param;
    RETURN answer;
END;
```

8.12.4.2 結果がキャッシュされる再帰ファンクション

フィボナッチ数列の数学的定義を模倣した、フィボナッチ数列のn番目の項を検索するための再帰ファンクションは、多くの冗長計算を実行する可能性があります。たとえば、fibonacci(7)を評価するために、このファンクションはfibonacci(6)およびfibonacci(5)を計算する必要があります。fibonacci(6)を計算するために、このファンクションはfibonacci(5)およびfibonacci(4)を計算する必要があります。このため、fibonacci(5)などのいくつかの項は、重複して計算されます。結果キャッシュを行うことによって、これらの冗長計算が回避されます。

ノート:

キャッシュされる再帰的起動の数は、最大で 128 です。

```
CREATE OR REPLACE FUNCTION fibonacci (n NUMBER)
```

RETURN NUMBER RESULT_CACHE AUTHID DEFINER IS BEGIN IF (n =0) OR (n =1) THEN RETURN 1; ELSE RETURN fibonacci(n - 1) + fibonacci(n - 2); END IF; END; /

8.12.5 結果がキャッシュされるファンクションの高度なトピック

ここでのトピック

- キャッシュ・ヒットの規則
- 結果キャッシュのバイパス
- 結果がキャッシュされるファンクションによるセッション固有の設定の処理
- 結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理
- 結果キャッシュの粒度の選択
- Oracle RAC環境での結果キャッシュ
- 結果キャッシュの管理
- 結果がキャッシュされるファンクションが依存するPL/SQLユニットへのホット・パッチの適用

8.12.5.1 キャッシュ・ヒットの規則

結果がキャッシュされるファンクションが異なるパラメータ値で起動されるたびに、それらのパラメータおよびそれぞれの結果がキャッシュに格納されます。それ以降、同じファンクションが同じパラメータ値で起動されると(つまり、キャッシュ・ヒットがある場合)、結果は再計算されるのではなく、キャッシュから取り出されます。

キャッシュ・ヒット用のパラメータ比較の規則は、次に示すように、PL/SQLの「等号」(=)演算子の規則とは異なります。

カテゴリ	キャッシュ・ヒットの規則	「等号」演算子の規則
NULL の比較	NULL は NULL に一致します	NULL = NULL の評価結果は NULL です。
NULL でないスカラーの比較	NULL でないスカラーは、それぞれの値が同一である場合にのみ同じとなります(つまり、指定されたプラットフォームでそれぞれの値が同一のビット・パターンを持っている場合にのみ同じとなります)。たとえば、CHAR 値'AA'と'AA 'は異なります。(この規則の方が、「等号」演算子の規則より厳密です。)	NULL でないスカラーは、指定されたプラットフォームでそれぞれの値が同一のビット・パターンを持っていない場合でも、等しくなる可能性があります。たとえば、CHAR 値'AA'と'AA'は等しくなります。

8.12.5.2 結果キャッシュのバイパス

場合によって、キャッシュはバイパスされます。キャッシュがバイパスされる場合を次に示します。

- ファンクションが結果をキャッシュから取り出すのではなく、計算する場合。
- ファンクションが計算する結果がキャッシュに追加されない場合。

キャッシュがバイパスされる場合の例を次にいくつか示します。

- すべてのセッションでキャッシュを使用できない場合。

  たとえば、データベース管理者がアプリケーションへのパッチの適用中に、結果キャッシュを使用できない状態にした場合などです(「結果がキャッシュされるファンクションが依存するPL/SQLユニットへのホット・パッチの適用」を参照)。

- 結果がキャッシュされるファンクションが依存する表またはビューに対して、セッションがDML文を実行している場合。

  このセッションは、そのDML文が完了するまで(コミットまたはロールバックされるまで)そのファンクションの結果キャッシュをバイパスします。その文がロールバックされると、このセッションは、そのファンクションのキャッシュの使用を再開します。

  キャッシュをバイパスすると、次のことが保証されます。

  - 各セッションのユーザーは、コミットされていないユーザー独自の変更を参照できます。
  - PL/SQLファンクションの結果キャッシュには、すべてのセッションで参照可能なコミットされた変更のみが含まれます。このため、あるセッションでコミットされていない変更は、他のセッションでは参照できません。

8.12.5.3 結果がキャッシュされるファンクションによるセッション固有の設定の処理

セッションによって異なる可能性がある設定(NLS_DATE_FORMATやTIME ZONEなど)にファンクションが依存している場合は、様々な設定を処理できるようにそのファンクションを変更できる場合にのみ、そのファンクションの結果がキャッシュされるようにします。

例8-39のファンクションget_hire_dateは、TO_CHARファンクションを使用してDATE項目をVARCHAR項目に変換しています。ファンクションget_hire_dateに書式マスクが指定されていないため、書式マスクは、デフォルトでNLS_DATE_FORMATに指定されている書式マスクになります。get_hire_dateを起動するセッションのNLS_DATE_FORMAT設定が異なっている場合、キャッシュされた結果の書式も異なる可能性があります。あるセッションで計算されてキャッシュされた結果がエージ・アウトされ、別のセッションで再計算された場合、同じパラメータ値に対する場合でも、書式が異なる可能性があります。キャッシュされた結果がセッションで取得され、結果の書式がセッションの書式とは異なる場合、その結果は不適切である可能性があります。

この問題の解決方法をいくつか次に示します。

- get_hire_dateの戻り型をDATEに変更し、各セッションがTO_CHARファンクションを起動するようにします。
- ある共通の書式がすべてのセッションで受入れ可能である場合は、書式マスクを指定して、NLS_DATE_FORMATへの依存性を削除します。たとえば:

  ```
  TO_CHAR(date_hired, 'mm/dd/yy');
  ```

- 書式マスクのパラメータをget_hire_dateに追加します。たとえば:

  ```
  CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER, fmt VARCHAR)
    RETURN VARCHAR
    RESULT_CACHE
    AUTHID DEFINER
  IS
    date_hired DATE;
  BEGIN
    SELECT hire_date INTO date_hired
      FROM HR.EMPLOYEES
        WHERE EMPLOYEE_ID = emp_id;
    RETURN TO_CHAR(date_hired, fmt);
  END;
  /
  ```

例8-39 結果がキャッシュされるファンクションによるセッション固有の設定の処理

```
CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER)
  RETURN VARCHAR
  RESULT_CACHE
  AUTHID DEFINER
IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
    FROM HR.EMPLOYEES
      WHERE EMPLOYEE_ID = emp_id;
  RETURN TO_CHAR(date_hired);
END;
/
```

8.12.5.4 結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理

アプリケーション・コンテキストは、グローバルまたはセッション固有のいずれかで、属性とそれらの値の集合のことです。PL/SQLファンクションは、次の1つ以上の項目を実行する場合、セッション固有のアプリケーション・コンテキストに依存します。

- 指定したコンテキストで指定した属性の値を戻すSQLファンクションSYS_CONTEXTの直接起動
- ファイングレイン・セキュリティのための仮想プライベート・データベース(VPD)・メカニズムを使用したSYS_CONTEXTの間接起動

  (VPDの詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください)

PL/SQLファンクションの結果キャッシュ機能は、セッション固有のアプリケーション・コンテキストへの依存性を自動的には処理しません。セッション固有のアプリケーション・コンテキストに依存しているファンクションの結果をキャッシュする必要がある場合は、アプリケーション・コンテキストをパラメータとしてファンクションに渡す必要があります。このパラメータにはデフォルト値を指定できるため、すべてのユーザーがこのパラメータを指定する必要があるわけではありません。

例8-40では、表config_tabに、次に示す問合せを変換するVPDポリシーがあると想定しています。

```
SELECT value FROM config_tab WHERE name = param_name;
```

この問合せへ:

```
SELECT value FROM config_tab
WHERE name = param_name
AND app_id = SYS_CONTEXT('Config', 'App_ID');
```

例8-40 結果がキャッシュされるファンクションによるセッション固有のアプリケーション・コンテキストの処理

```
CREATE OR REPLACE FUNCTION get_param_value (
  param_name VARCHAR,
  appctx     VARCHAR  DEFAULT SYS_CONTEXT('Config', 'App_ID')
) RETURN VARCHAR
  RESULT_CACHE
  AUTHID DEFINER
IS
  rec VARCHAR(2000);
BEGIN
  SELECT val INTO rec
  FROM config_tab
  WHERE name = param_name;

  RETURN rec;
END;
/
```

8.12.5.5 結果キャッシュの粒度の選択

PL/SQLにはファンクション結果キャッシュが用意されていますが、キャッシュの粒度はユーザーが選択します。粒度の概念を理解するために、Order Entry(OE)サンプル・スキーマ内のProduct_Descriptions表について考えてみます。

```
NAME                    NULL?     TYPE
----------------------  --------  ---------------
PRODUCT_ID              NOT NULL  NUMBER(6)
LANGUAGE_ID             NOT NULL  VARCHAR2(3)
TRANSLATED_NAME         NOT NULL  NVARCHAR2(50)
TRANSLATED_DESCRIPTION  NOT NULL  NVARCHAR2(2000)
```

この表には、各製品の名前と説明が複数の言語で記載されています。各行の一意のキーは、PRODUCT_ID, LANGUAGE_IDです。

PRODUCT_IDおよびLANGUAGE_IDを受け取って、関連付けられたTRANSLATED_NAMEを戻すファンクションを定義する必要があるとします。また、変換された名前をキャッシュする必要もあるとします。これらの名前をキャッシュする場合の粒度の選択肢の一部を次に示します。

- 一度に1つの名前(粒度が細かい)
- 一度に1つの言語(粒度が粗い)

表8-4 粒度が細かいキャッシュと粗いキャッシュ

粒度	メリット
細かい	各ファンクション結果は、1 つの論理結果に対応しています。 1 回以上必要とされるデータのみを格納します。 各データ項目は、個別にエージ・アウトされます。 バルク・ロードは最適化できません。
粗い	各ファンクション結果には、多数の論理的部分結果が含まれています。 使用されることのないデータを格納する場合もあります。 1 つのデータ項目がエージ・アウトされると、全体がエージ・アウトされます。 バルク・ロードを最適化できます。

例8-41および例8-42では、ファンクションproductNameがPRODUCT_IDおよびLANGUAGE_IDを取り、関連付けられたTRANSLATED_NAMEを戻しています。productNameの各バージョンは、変換された名前をキャッシュしますが、キャッシュする際の粒度はそれぞれ異なっています。

例8-41では、get_product_name_1は結果がキャッシュされるファンクションです。get_product_name_1は、別のPRODUCT_IDおよびLANGUAGE_IDで起動されると、常に関連付けられたTRANSLATED_NAMEをキャッシュします。get_product_name_1が起動されるたびに、最大1つのTRANSLATED_NAMEがキャッシュに追加されます。

例8-42では、get_product_name_2は、結果がキャッシュされるファンクションall_product_namesを定義します。get_product_name_2が別のLANGUAGE_IDでall_product_namesを起動すると、常にall_product_namesはそのLANGUAGE_IDに関連付けられたすべてのTRANSLATED_NAMEをキャッシュします。all_product_namesが起動されるたびに、最大1つのLANGUAGE_IDのすべてのTRANSLATED_NAMEがキャッシュに追加されます。

例8-41 一度に1つの名前のキャッシュ(粒度が細かい)

```
CREATE OR REPLACE FUNCTION get_product_name_1 (
  prod_id NUMBER,
  lang_id VARCHAR2
)
  RETURN NVARCHAR2
  RESULT_CACHE
```

```
  AUTHID DEFINER
IS
  result_ VARCHAR2(50);
BEGIN
  SELECT translated_name INTO result_
  FROM OE.Product_Descriptions
  WHERE PRODUCT_ID = prod_id
  AND LANGUAGE_ID = lang_id;
  RETURN result_;
END;
/
```

例8-42 一度に1つの言語の変換された名前のキャッシュ(粒度が粗い)

```
CREATE OR REPLACE FUNCTION get_product_name_2 (
  prod_id NUMBER,
  lang_id VARCHAR2
)
  RETURN NVARCHAR2
  AUTHID DEFINER
IS
  TYPE product_names IS TABLE OF NVARCHAR2(50) INDEX BY PLS_INTEGER;

  FUNCTION all_product_names (lang_id VARCHAR2)
    RETURN product_names
    RESULT_CACHE
  IS
    all_names product_names;
  BEGIN
    FOR c IN (SELECT * FROM OE.Product_Descriptions
              WHERE LANGUAGE_ID = lang_id) LOOP
      all_names(c.PRODUCT_ID) := c.TRANSLATED_NAME;
    END LOOP;
    RETURN all_names;
  END;
BEGIN
  RETURN all_product_names(lang_id)(prod_id);
END;
/
```

8.12.5.6 Oracle RAC環境での結果キャッシュ

キャッシュされた結果はシステム・グローバル領域(SGA)に格納されます。Oracle RAC環境では、各データベース・インスタンスによって、そのインスタンス独自のローカルなファンクション結果キャッシュが管理されます。ただし、ローカルな結果キャッシュの内容は、他のOracle RACインスタンスに付随するセッションからアクセスできます。必要な結果がローカル・インスタンスの結果キャッシュから欠落している場合、ローカルで計算するのではなく、他のインスタンスのローカル・キャッシュから結果が取り出される場合があります。

インスタンスのアクセス・パターンおよびワークロードによって、そのインスタンスのローカル・キャッシュに格納される結果セットが決まります。このため、インスタンスが異なると、そのローカル・キャッシュに格納される結果セットも異なります。

各データベース・インスタンスには、そのインスタンス独自のキャッシュされた結果セットが含まれている可能性がありますが、無効な結果を処理するメカニズムはOracle RAC環境全体にわたります。ローカル・インスタンスの結果キャッシュでのみ結果が無効

にされた場合、他のインスタンスで無効な結果が使用される可能性があります。たとえば、データベース表内のデータから計算される品目の価格の結果キャッシュについて考えてみます。ある品目の価格に影響を与える方法でこれらのデータベース表のいずれかが更新された場合、キャッシュされたその品目の価格をOracle RAC環境内のすべてのデータベース・インスタンスで無効にする必要があります。

8.12.5.7 結果キャッシュの管理

PL/SQLファンクション結果キャッシュは、その管理および管理性インフラストラクチャを結果キャッシュと共有します。

データベース管理者は、初期化パラメータのRESULT_CACHE_MAX_SIZE、RESULT_CACHE_MAX_RESULTおよびRESULT_CACHE_REMOTE_EXPIRATIONを指定することで、サーバーの結果キャッシュを管理します。

DBMS_RESULT_CACHEパッケージでは、DBAが、SQL結果キャッシュおよびPL/SQLファンクション結果キャッシュによって使用される共有プールのその部分を管理できるインタフェースが提供されます。

動的パフォーマンスのビューには、サーバーとクライアントの結果キャッシュを監視するための情報が示されます。

関連項目:

- RESULT_CACHE_MAX_SIZEの詳細は、『Oracle Databaseリファレンス』を参照してください。
- RESULT_CACHE_MAX_RESULTの詳細は、『Oracle Databaseリファレンス』を参照してください。
- 結果キャッシュの概念の詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。
- DBMS_RESULT_CACHEパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください
- V$RESULT_CACHE_STATISTICSの詳細は、『Oracle Databaseリファレンス』を参照してください
- V$RESULT_CACHE_MEMORYの詳細は、『Oracle Databaseリファレンス』を参照してください
- V$RESULT_CACHE_OBJECTSの詳細は、『Oracle Databaseリファレンス』を参照してください
- V$RESULT_CACHE_DEPENDENCYの詳細は、『Oracle Databaseリファレンス』を参照してください

8.12.5.8 結果がキャッシュされるファンクションが依存するPL/SQLユニットへのホット・パッチの適用

結果がキャッシュされるファンクションが依存するPL/SQLユニットに(直接または間接的に)ホット・パッチを適用する際、結果がキャッシュされるファンクションに関連付けられているキャッシュされた結果がすべての場合に自動的にフラッシュされるとはかぎりません。

たとえば、結果がキャッシュされるファンクションP1.foo()がパッケージ・サブプログラムP2.bar()に依存しているとします。パッケージP2の本体の新しいバージョンがロードされた場合、P1.foo()に関連付けられているキャッシュされた結果は、自動的にはフラッシュされません。

このため、PL/SQLユニットへのホット・パッチの適用には、次の手順を実行することをお薦めします。

ノート:

これらのステップに従うには、DBMS_RESULT_CACHE パッケージに対する EXECUTE 権限が必要です。

1. 結果キャッシュをバイパス・モードに設定し、既存の結果をフラッシュします。

```
BEGIN
  DBMS_RESULT_CACHE.Bypass(TRUE);
  DBMS_RESULT_CACHE.Flush;
END;
/
```

   Oracle RAC環境では、各データベース・インスタンスに対してこのステップを実行します。

2. PL/SQLコードにパッチを適用します。
3. 結果キャッシュの使用を再開します。

```
BEGIN
  DBMS_RESULT_CACHE.Bypass(FALSE);
END;
/
```

   Oracle RAC環境では、各データベース・インスタンスに対してこのステップを実行します。

8.13 SQL文によって起動できるPL/SQLファンクション

ストアド・ファンクション(およびそのファンクションによって起動されるサブプログラム)は、次に示す副作用を制御するための純正規則に従っている場合にのみ、SQL文から起動できます。

- SELECT文またはパラレル化INSERT文、UPDATE文、DELETE文またはMERGE文から起動された場合、サブプログラムはデータベース表を変更できません。
- INSERT文、UPDATE文、DELETE文またはMERGE文から起動された場合、サブプログラムは、その文によって変更されたデータベース表の問合せまたは変更を実行できません。

  ファンクションによって表の問合せまたは変更を実行する場合に、その表に対するDML文でファンクションを起動すると、ORA-04091(変更表エラー)が発生します。ただし、これには例外が1つあり、FORALL文に存在しない単一行のINSERT文によってVALUES句のファンクションを起動する場合、ORA-04091は発生しません。
- SELECT文、INSERT文、UPDATE文、DELETE文またはMERGE文から起動された場合、サブプログラムは、次のどのSQL文も実行できません(PRAGMA AUTONOMOUS_TRANSACTIONが指定されていない場合)。
  - トランザクション制御文(COMMITなど)
  - セッション制御文(SET ROLEなど)
  - システム制御文(ALTER SYSTEMなど)
  - データベース定義言語(DDL)文(CREATEなど)。これらの文は自動的にコミットされます。

  (PRAGMA AUTONOMOUS_TRANSACTIONの詳細は、「AUTONOMOUS_TRANSACTIONプラグマ」を参照してください。)

ファンクション実行部のいずれかのSQL文が規則に違反すると、文の解析時にランタイム・エラーが発生します。

特にファンクションがDETERMINISTICまたはPARALLEL_ENABLEオプション(これらのオプションの詳細は、DETERMINISTIC句およびPARALLEL_ENABLE句を参照)を使用して宣言されている場合、ファンクションの副作用が少なくなるほど、そのファンクションはSELECT文でより効率的に最適化されます。

関連項目:

- SQL文によって起動できるPL/SQLファンクションの制限の詳細は、『Oracle Database開発ガイド』を参照してください。
- 「問合せにおけるファンクションの起動のチューニング」

8.14 実行者権限および定義者権限(AUTHIDプロパティ)

ストアドPL/SQLユニットのAUTHIDプロパティは、実行時にユニットによって発行されるSQL文の名前解決および権限チェックに影響します。AUTHIDプロパティはコンパイルには影響せず、コレクション型などのコードのないユニットに対しては意味を持ちません。

AUTHIDプロパティ値は、静的データ・ディクショナリ・ビュー*_PROCEDURESに公開されます。AUTHIDが意味を持つユニットの場合はビューに値CURRENT_USERまたはDEFINERが表示され、他のユニットの場合はビューにNULLが表示されます。

次の文で作成または変更するストアドPL/SQLユニットについては、オプションのAUTHID句を使用して、DEFINER(デフォルト、下位互換性用)またはCURRENT_USER(優先用途)を指定できます。

- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE PROCEDURE文
- 「CREATE TYPE文」
- ALTER TYPE文

AUTHID値がCURRENT_USERのユニットは、実行者権限ユニットまたはIRユニットと呼ばれます。AUTHID値がDEFINER(デフォルト)のユニットは、定義者権限ユニットまたはDRユニットと呼ばれます。AUTHID値を指定できないPL/SQLユニットおよびスキーマ・オブジェクトは次のように動作します。

PL/SQLユニットまたはスキーマ・オブジェクト	動作
無名ブロック	IR ユニット
BEQUEATH CURRENT_USER ビュー	IR ユニットに類似(『Oracle Database セキュリティ・ガイド』を参照)
BEQUEATH DEFINER ビュー	DR ユニット
トリガー	DR ユニット

ユニットのAUTHIDプロパティによって、そのユニットがIRであるかDRであるかが決まります。このプロパティは実行時の名前解決および権限チェックの両方に影響します。

- 名前解決のコンテキストはCURRENT_SCHEMAです。
- 権限チェックのコンテキストはCURRENT_USERおよび有効になっているロールです。

セッションが開始されると、CURRENT_SCHEMAの値はSESSION_USERが所有するスキーマの値となり、CURRENT_USERの値はSESSION_USERの値と同じになります。(CURRENT_SCHEMA、CURRENT_USERまたはSESSION_USERの現在の値を取得するには、『Oracle Database SQL言語リファレンス』で説明されているSYS_CONTEXTファンクションを使用します。)

CURRENT_SCHEMAは、SQL文ALTER SESSION SET CURRENT_SCHEMAを使用してセッション中に変更できます。CURRENT_USERは、プログラムでは変更できませんが、PL/SQLユニットまたはビューがコール・スタックに格納されたり、コール・スタックから取り出されると、変更される場合があります。

ノート:

ALTER SESSION SET CURRENT_SCHEMA は、ストアド PL/SQL ユニット内から発行しないことをお薦めします。

サーバー・コール中にDRユニットがコール・スタックに格納されると、データベースによって、現在有効になっているロール、およびCURRENT_USERとCURRENT_SCHEMAの現在の値が格納されます。次に、CURRENT_USERとCURRENT_SCHEMAの両方がDRユニットの所有者に変更され、ロールPUBLICのみが有効にされます。(格納されたロールおよび値と新しいロールおよび値は、異なるとはかぎりません。)DRユニットがコール・スタックから取り出されると、データベースによって、格納されたロールおよび値がリストアされます。これに対し、IRユニットがコール・スタックに格納されたり、コール・スタックから取り出されても、CURRENT_USERとCURRENT_SCHEMAの値および現在有効になっているロールは変更されません(ロールがIRユニット自体に付与されていないかぎり。「PL/SQLパッケージおよびスタンドアロン・プログラムへのロールの付与」を参照)。

PL/SQLユニットによって発行される動的SQL文の場合、名前解決および権限チェックは実行時に1回行われます。静的SQL文の場合、名前解決および権限チェックは、PL/SQLユニットのコンパイル時、およびその後の実行時の2回行われます。コンパイル時に、AUTHIDプロパティは影響しません(DRユニットとIRユニットの両方がDRユニットのように処理されます)。ただし、実行時には、AUTHIDプロパティによってユニットがIRであるかDRであるかが決まり、その結果に従ってユニットは処理されます。

IRユニットの開始時に、初期化またはなんらかのコードの実行前に、ランタイム・システムにより権限がチェックされます。ユニット所有者に起動者に対するINHERIT PRIVILEGES権限またはINHERIT ANY PRIVILEGES権限のいずれもなければ、ランタイム・システムによりエラーORA-06598が発行されます。

ノート:

ユニット所有者に必要な権限がある場合、その権限は次のいずれかの文により付与されています。

```
GRANT INHERIT PRIVILEGES ON current_user TO PUBLIC
GRANT INHERIT PRIVILEGES ON current_user TO unit_owner
GRANT INHERIT ANY PRIVILEGES TO unit_owner
```

GRANT 文の詳細は『Oracle Database SQL 言語リファレンス』を参照してください。

関連項目:

- DRおよびIRユニットのセキュリティの管理の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。
- DRおよびIRプログラム・ユニットのコンパイルに必要な権限の取得の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照

ここでのトピック

- PL/SQLパッケージおよびスタンドアロン・サブプログラムへのロールの付与
- IRユニットにはテンプレート・オブジェクトが必要

8.14.1 PL/SQLパッケージおよびスタンドアロン・サブプログラムへのロールの付与

SQL GRANTコマンドを使用して、PL/SQLパッケージおよびスタンドアロン・サブプログラムにロールを付与できます。PL/SQLユニットに付与されたロールはコンパイルには影響しません。ロールはユニットが実行時に発行するSQL文の権限チェックに影響します。ユニットは、自身のロールと現在有効になっているその他のロールの両方の権限で実行されます。

通常、自分より低い権限のユーザーがユニットの実行に必要な権限でのみユニットを実行できるように、IRユニットにロールを付与します。DRユニット(起動者により定義者のすべての権限で実行される)にロールを付与するのは、そのDRユニットが実行時にのみチェックされる動的SQLを発行する場合のみです。

PL/SQLユニットにロールを付与するための基本構文は次のとおりです。

```
GRANT role [, role ]... TO unit [, unit ]...
```

たとえば、このコマンドはロールreadとexecuteをファンクションscott.funcとパッケージsys.pkgに付与します。

```
GRANT read, execute TO FUNCTION scott.func, PACKAGE sys.pkg
```

GRANTコマンドの構文およびセマンティクスの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

関連項目:

- PL/SQLユニットのロールを取り消すためのREVOKEコマンドの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- アプリケーション・ユーザーおよびアプリケーション・ロールの構成の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。

8.14.2 IRユニットにはテンプレート・オブジェクトが必要

1人のユーザー(つまり1つのスキーマ)がIRユニットを所有し、他のユーザーは自身のスキーマ内でそのユニットを実行します。IRユニットが静的SQL文を発行する場合、これらの文が影響するスキーマ・オブジェクトはコンパイル時には所有者のスキーマ内に存在する必要があり(コンパイラが参照を解決できるように)、また実行時には起動者のスキーマ内に存在する必要があります。

対応するスキーマ・オブジェクトの定義が一致する必要があります(たとえば、対応する表の名前と列が同じである必要があります)。一致していない場合は、エラーまたは予期しない結果が発生します。ただし、コンパイラで不要なため、所有者のスキーマ内のオブジェクトはデータを含む必要はありません。したがって、これらのオブジェクトはテンプレート・オブジェクトと呼ばれます。

8.14.3 DRユニット内の接続ユーザー・データベース・リンク

DRユニット(定義者権限ユニット)に接続ユーザー・データベース・リンクを含める場合、DRユニットを実行するユーザーにINHERIT REMOTE PRIVILEGES権限を付与する必要があります。

この権限をユーザーに付与すると、そのユーザーはDRユニットを実行できるようになります。付与していない場合、ORA-25433: User does not have INHERIT REMOTE PRIVILEGESエラーとともに実行は失敗します。定義者権限(DR)プロシージャ内から接続ユーザー・データベース・リンクを含めるには、プロシージャに@database_linkを含めます。

次の例は、DRユニットがdblinkというデータベース・リンクを使用してHR.EMPLOYEES表のEMPLOYEE_ID列にアクセスする方法を示しています。

例8-43 DRユニット内のデータベース・リンク

```
CREATE OR REPLACE PROCEDURE hr_remote_db_link
AS
v_employee_id VARCHAR(50);
BEGIN
    EXECUTE IMMEDIATE 'SELECT employee_id FROM employees@dblink' into v_employee_id;
    DBMS_OUTPUT.PUT_LINE('employee_id: ' || v_employee_id);
END ;
/
```

関連項目:

INHERIT REMOTE PRIVILEGES権限の使用に関する詳細(DRユニットでデータベース・リンクを使用する方法のチュートリアルを含む)は、Oracle Databaseセキュリティ・ガイドを参照してください

8.15 外部サブプログラム

CプロシージャまたはJavaメソッドがデータベースに格納されている場合、それを外部サブプログラムとして公開し、PL/SQLから起動できます。

外部サブプログラムを公開するには、コール仕様を使用して、ストアドPL/SQLサブプログラムを定義します。コール仕様は、外部サブプログラムの名前、パラメータ型および戻り型をPL/SQLの同等要素にマップします。公開した外部サブプログラムは、そのPL/SQL名によって起動します。

たとえば、Adjusterという次のJavaクラスがデータベースに格納されているとします。

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class Adjuster {
  public static void raiseSalary (int empNo, float percent)
  throws SQLException {
```

```
    Connection conn = new OracleDriver().defaultConnection();
    String sql = "UPDATE employees SET salary = salary * ?
                    WHERE employee_id = ?";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setFloat(1, (1 + percent / 100));
      pstmt.setInt(2, empNo);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e)
          {System.err.println(e.getMessage());}
    }
}
```

JavaクラスAdjusterには、指定の従業員の給与を指定のパーセンテージ分のみ増やすraiseSalaryというメソッドがあります。raiseSalaryは、voidメソッドであるため、(ファンクションではなく)PL/SQLプロシージャとして公開します。

例8-44では、格納されたJavaメソッドAdjuster.raiseSalaryをPL/SQLスタンドアロン・プロシージャとして公開するため、Javaメソッド名のAdjuster.raiseSalaryをPL/SQLプロシージャ名のraise_salaryに、Javaデータ型のintおよびfloatをPL/SQLデータ型のNUMBERにマップしています。その後、無名ブロックでraise_salaryを起動します。

例8-45では、格納されたJavaメソッドjava.lang.Thread.sleepをPL/SQLスタンドアロン・プロシージャとして公開するため、Javaメソッド名をPL/SQLプロシージャ名のjava_sleepに、Javaデータ型のlongをPL/SQLデータ型のNUMBERにマップしています。PL/SQLスタンドアロン・プロシージャのsleepで、java_sleepを起動します。

関連項目:

外部プログラムのコールの詳細は、『Oracle Database開発ガイド』を参照してください。

例8-44 PL/SQL無名ブロックによる外部プロシージャの起動

```
-- Publish Adjuster.raiseSalary as standalone PL/SQL procedure:
CREATE OR REPLACE PROCEDURE raise_salary (
  empid NUMBER,
  pct   NUMBER
) AS
  LANGUAGE JAVA NAME 'Adjuster.raiseSalary (int, float)';  -- call specification
/
BEGIN
  raise_salary(120, 10);  -- invoke Adjuster.raiseSalary by PL/SQL name
END;
/
```

例8-45 PL/SQLスタンドアロン・プロシージャによる外部プロシージャの起動

```
-- Java call specification:
CREATE PROCEDURE java_sleep (
  milli_seconds IN NUMBER
) AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
/
CREATE OR REPLACE PROCEDURE sleep (
  milli_seconds IN NUMBER
```

```
) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
  java_sleep (milli_seconds);
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
END;
/
```

9 PL/SQLトリガー

トリガーは、指定のイベントが発生するたびにOracle Databaseによって自動的に起動されるストアド・プロシージャに似ています。

> ノート:
>
> データベースは、システム定義のイベントのみ検出できます。独自のイベントは定義できません。

ここでのトピック

- トリガーの概要
- トリガーを使用する理由
- DMLトリガー
- 相関名および疑似レコード
- システム・トリガー
- トリガーによって起動されるサブプログラム
- トリガーのコンパイル、無効化および再コンパイル
- トリガーでの例外処理
- トリガー設計のガイドライン
- トリガーの制限
- トリガーの起動順序
- トリガーの有効化および無効化
- トリガーの変更およびデバッグ
- トリガーおよびOracle Databaseデータ転送ユーティリティ
- イベントをパブリッシュするトリガー
- トリガーに関する情報が表示されるビュー

9.1 トリガーの概要

トリガーは、ストアド・プロシージャのように、データベースに格納して繰り返し起動できる名前付きPL/SQLユニットです。ストアド・プロシージャとは異なり、トリガーは、有効にしたり無効にすることができますが、明示的に起動することはできません。

トリガーは、有効の場合、データベースによって自動的に起動されます(つまり、トリガーは、トリガーを起動するイベントが発生すると、常に起動されます)。トリガーは、無効の場合、起動されません。

トリガーは、CREATE TRIGGER文を使用して作成します。トリガーを起動する文およびトリガーの動作対象となる項目に関して、トリガーを起動するイベントを指定します。トリガーは、項目(表、ビュー、スキーマまたはデータベース)に対して作成または定義

するといいます。また、タイミングも指定します(タイミングによって、トリガーを起動する文の実行前または実行後のどちらにトリガーを起動するかと、トリガーを起動する文の影響を受ける行ごとにトリガーを起動するかどうかを決定します)。デフォルトでは、トリガーは有効な状態で作成されます。

トリガーが表またはビューに対して作成される場合、トリガーを起動するイベントはDML文で構成され、そのトリガーはDMLトリガーと呼ばれます。

crosseditionトリガーは、エディション・ベースの再定義でのみ使用されるDMLトリガーです。

トリガーがスキーマまたはデータベースに対して作成される場合、トリガーを起動するイベントは、DDLまたはデータベース操作文で構成され、そのトリガーはシステム・トリガーと呼ばれます。

条件付きトリガーは、トリガーを起動する文に影響を受ける行ごとにデータベースで評価されるSQL条件を指定するWHEN句を含む、DMLトリガーまたはシステム・トリガーです。

トリガーで参照される表は、トリガーの起動時に、他のユーザーのトランザクションで使用されたSQL文によって変更中である可能性があります。トリガー内で実行されるSQL文は、スタンドアロンのSQL文が従う規則と同じ規則に従います。具体的には次のとおりです。

- トリガーの問合せでは、参照される表の読取り一貫性のある現在のマテリアライズド・ビューおよび同一トランザクション内で変更されたデータが確認されます。
- トリガーの更新では、既存のデータ・ロックが解放されるまで待機してから、処理が続行されます。

INSTEAD OFトリガーは、次のいずれかです。

- 非エディショニング・ビューに対して、または非エディショニング・ビューのネストした表の列に対して作成されるDMLトリガー
- CREATE文で定義されたシステム・トリガー

データベースでは、トリガーを起動する文が実行されるかわりに、INSTEAD OFトリガーが起動されます。

> ノート:
>
> トリガーは、通常、トリガーを起動する文の名前(DELETE トリガーまたは LOGON トリガーなど)、トリガーの定義対象である項目の名前(DATABASE トリガーまたは SCHEMA トリガーなど)またはタイミング(BEFORE 文トリガーまたは AFTER 行単位トリガーなど)を使用して呼ばれます。

関連項目:

- 「CREATE TRIGGER文」の構文図
- DMLトリガー
- システム・トリガー
- crosseditionトリガーの詳細は、『Oracle Database開発ガイド』を参照してください

- WHEN句の詳細は、「CREATE TRIGGER文」を参照してください

9.2 トリガーを使用する理由

トリガーを使用して、データベース管理システムをカスタマイズできます。

たとえば、トリガーを使用すると、次の操作を行うことができます。

- 仮想列の値の自動的な生成
- イベントのログへの記録
- 表へのアクセスに関する統計情報の収集
- ビューに対してDML文が発行された場合の表データの変更
- 親表と子表が分散データベースの異なるノード上にあるときの参照整合性の実行
- サブスクライブ元のアプリケーションへのデータベース・イベント、ユーザー・イベントおよびSQL文に関する情報のパブリッシュ
- 通常の業務時間後における表に対するDML操作の防止
- 無効なトランザクションの防止
- 制約では定義できない複雑なビジネス・ルールまたは参照整合性規則の実行(「トリガーと制約の違い」を参照)

注意:

トリガーは、プログラムで構築されるものであり、無効になりやすいため、信頼性の高いセキュリティ・メカニズムではありません。高いセキュリティを確保するには、Oracle Database Vault を使用してください(『Oracle Database Vault 管理者ガイド』を参照)。

トリガーと制約の違い

トリガーと制約のどちらを使用してもデータ入力を制約できますが、両者は大きく異なります。

トリガーは、常に新規データのみに適用されます。たとえば、トリガーによって、DML文がデータベースの列にNULL値を挿入することを防止できますが、列には、トリガーが定義される前またはトリガーが無効な間に挿入されたNULL値が含まれている可能性があります。

制約は、トリガーのように新しいデータのみに適用するか、新しいデータと既存のデータの両方に適用できます。『Oracle Database SQL言語リファレンス』で説明されているように、制約の動作は制約の状態によって異なります。

制約は、同じルールを実行するトリガーに比べて作成しやすく、エラーが発生する可能性が低くなります。ただしトリガーは、制約では定義できない複雑なビジネス・ルールを実行できます。トリガーは、次の場合にのみデータ入力の制約に使用することをお薦めします。

- 親表と子表が分散データベースの異なるノード上にあるときに参照整合性を実行する場合
- 制約では定義できない複雑なビジネス・ルールまたは参照整合性規則を実行する場合

関連項目:

- 制約を使用してビジネス・ルールを実行したり、表への無効な情報の入力を防止する方法の詳細は、『Oracle Database開発ガイド』を参照してください。
- トリガーおよび制約を使用して親表と子表の間の参照整合性を保持する方法の詳細は、「参照整合性を保証するトリガー」を参照してください

9.3 DMLトリガー

DMLトリガーは、表またはビューに対して作成し、そのトリガーを起動するイベントは、DML文のDELETE、INSERTおよびUPDATEで構成されます。

MERGE文に応答して起動するトリガーを作成するには、MERGE操作の構成要素であるINSERT文およびUPDATE文に対してトリガーを作成します。

DMLトリガーは、単純なトリガーまたは複合トリガーです。

単純なDMLトリガーは、次のいずれかのタイミングで起動されます。

- トリガーを起動する文が実行される前

  (このトリガーは、BEFORE文トリガーまたは文レベルのBEFOREトリガーと呼ばれます。)
- トリガーを起動する文が実行された後

  (このトリガーは、AFTER文トリガーまたは文レベルのAFTERトリガーと呼ばれます。)
- トリガーを起動する文の影響を受ける各行の前

  (このトリガーは、BEFORE行単位トリガーまたは行レベルのBEFOREトリガーと呼ばれます。)
- トリガーを起動する文の影響を受ける各行の後

  (このトリガーは、AFTER行単位トリガーまたは行レベルのAFTERトリガーと呼ばれます。)

表またはエディショニング・ビューに対して作成された複合DMLトリガーは、前述のタイミングの1つ、一部または全部で起動できます。複合DMLトリガーを使用すると、様々なタイミングで実装したアクションで共通データを共有するアプローチを簡単にプログラムできます。

行レベルで起動される単純なDMLトリガーまたは複合DMLトリガーでは、処理中の行のデータにアクセスできます。詳細は、「相関名および疑似レコード」を参照してください。

INSTEAD OF DMLトリガーは、非エディショニング・ビューに対して、または非エディショニング・ビューのネストした表の列に対して作成されるDMLトリガーです。

INSTEAD OFトリガーを除き、トリガーを起動するUPDATE文には、列リストが含まれる場合があります。列リストが含まれていると、トリガーは、指定された列が更新される場合にのみ起動されます。列リストが省略されていると、トリガーは、関連付けられている表のいずれかの列が更新された場合に起動されます。

ここでのトピック

- トリガーを起動するDML文を検出する条件述語
- INSTEAD OF DMLトリガー
- 複合DMLトリガー
- 参照整合性を保証するトリガー

9.3.1 トリガーを起動するDML文を検出する条件述語

DMLトリガーのトリガーを起動するイベントは、トリガーを起動する複数の文で構成できます。複数の文のうちの1つによってトリガーが起動されると、トリガーは、次の条件述語を使用してその文を判断します。

表9-1 条件述語

条件述語	TRUEになる唯一の条件
INSERTING	INSERT 文がトリガーを起動した場合。
UPDATING	UPDATE 文がトリガーを起動した場合。
UPDATING ('column')	指定した列に影響する UPDATE 文がトリガーを起動した場合。
DELETING	DELETE 文がトリガーを起動した場合。

条件述語は、BOOLEAN式を指定できるすべての場所で使用できます。

例9-1 トリガーによる条件述語を使用したトリガーを起動する文の検出

この例では、条件述語を使用して、トリガーを起動する可能性のある4つの文のうちのどの文でトリガーが起動されたかを判断するDMLトリガーを作成します。

```
CREATE OR REPLACE TRIGGER t
  BEFORE
    INSERT OR
    UPDATE OF salary, department_id OR
    DELETE
  ON employees
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('salary') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary');
    WHEN UPDATING('department_id') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID');
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END CASE;
END;
/
```

9.3.2 INSTEAD OF DMLトリガー

INSTEAD OF DMLトリガーは、非エディショニング・ビューに対して、または非エディショニング・ビューのネストした表の列に対して作成されるDMLトリガーです。データベースでは、トリガーを起動するDML文が実行されるかわりに、INSTEAD OFトリガーが起動されます。

INSTEAD OFトリガーに条件を付けることはできません。

INSTEAD OFトリガーは、本質的に更新できないビューを更新する唯一の方法です。INSTEAD OFトリガーを設計して、対象となる操作を判断し、基礎となる表に対して適切なDML操作を実行します。

INSTEAD OFトリガーは、常に行レベルのトリガーです。INSTEAD OFトリガーは、OLD値およびNEW値を読み取ることができますが、変更することはできません。

NESTED TABLE句のあるINSTEAD OFトリガーは、トリガーを起動する文の操作対象がビューにおける特定のネストした表の列の要素である場合にのみ起動されます。このトリガーは、ネストした表の変更される各要素に対して起動されます。

関連項目:

- 更新可能なビューの詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- INSTEAD OF EACH ROWセクションを持つ複合DMLトリガーの詳細は、「複合DMLトリガーの構造」を参照してください

例9-2 INSTEAD OFトリガー

この例では、顧客情報および注文情報を表示するoe.order_infoビューを作成します。orders表の主キー(order_id)は結合ビューの結果セットで一意でないため、このビューは本質的に更新できません。この例では、ビューに送られるINSERT文を処理するINSTEAD OFトリガーを作成します。このトリガーは、ビューの実表(customersおよびorders)に行を挿入します。

```
CREATE OR REPLACE VIEW order_info AS
   SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
          o.order_id, o.order_date, o.order_status
   FROM customers c, orders o
   WHERE c.customer_id = o.customer_id;
CREATE OR REPLACE TRIGGER order_info_insert
   INSTEAD OF INSERT ON order_info
   DECLARE
     duplicate_info EXCEPTION;
     PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
   BEGIN
     INSERT INTO customers
       (customer_id, cust_last_name, cust_first_name)
     VALUES (
     :new.customer_id,
     :new.cust_last_name,
     :new.cust_first_name);
   INSERT INTO orders (order_id, order_date, customer_id)
   VALUES (
     :new.order_id,
     :new.order_date,
     :new.customer_id);
```

```
   EXCEPTION
     WHEN duplicate_info THEN
       RAISE_APPLICATION_ERROR (
         num=> -20107,
         msg=> 'Duplicate customer or order ID');
   END order_info_insert;
/
```

挿入行が存在しないことを示す問合せ:

```
SELECT COUNT(*) FROM order_info WHERE customer_id = 999;
```

結果:

```
  COUNT(*)
----------
         0

1 row selected.
```

ビューへの行の挿入:

```
INSERT INTO order_info VALUES
   (999, 'Smith', 'John', 2500, TO_DATE('13-MAR-2001', 'DD-MON-YYYY'), 0);
```

結果:

```
1 row created.
```

ビューに行が挿入されたことを示す問合せ:

```
SELECT COUNT(*) FROM order_info WHERE customer_id = 999;
```

結果:

```
  COUNT(*)
----------
         1

1 row selected.
```

customers表に行が挿入されたことを示す問合せ:

```
SELECT COUNT(*) FROM customers WHERE customer_id = 999;
```

結果:

```
  COUNT(*)
----------
         1

1 row selected.
```

orders表に行が挿入されたことを示す問合せ:

```
SELECT COUNT(*) FROM orders WHERE customer_id = 999;
```

結果:

```
  COUNT(*)
----------
         1

1 row selected.
```

例9-3 ビューのネストした表の列に対するINSTEAD OFトリガー

この例では、dept_viewビューに、CASTファンクション(『Oracle Database SQL言語リファレンス』を参照)によって作成された従業員のネストした表(emplist)が含まれます。この例では、emplist列を変更するため、その列に対するINSTEAD OFトリガーを作成します。

```
-- Create type of nested table element:

CREATE OR REPLACE TYPE nte
AUTHID DEFINER IS
OBJECT (
  emp_id     NUMBER(6),
  lastname   VARCHAR2(25),
  job        VARCHAR2(10),
  sal        NUMBER(8,2)
);
/

-- Created type of nested table:

CREATE OR REPLACE TYPE emp_list_ IS
  TABLE OF nte;
/

-- Create view:
CREATE OR REPLACE VIEW dept_view AS
  SELECT d.department_id,
         d.department_name,
         CAST (MULTISET (SELECT e.employee_id, e.last_name, e.job_id, e.salary
                          FROM employees e
                          WHERE e.department_id = d.department_id
                         )
                         AS emp_list_
               ) emplist
  FROM departments d;

-- Create trigger:

CREATE OR REPLACE TRIGGER dept_emplist_tr
  INSTEAD OF INSERT ON NESTED TABLE emplist OF dept_view
  REFERENCING NEW AS Employee
              PARENT AS Department
  FOR EACH ROW
BEGIN
  -- Insert on nested table translates to insert on base table:
  INSERT INTO employees (
    employee_id,
    last_name,
    email,
```

```
    hire_date,
    job_id,
    salary,
    department_id
  )
  VALUES (
    :Employee.emp_id,                       -- employee_id
    :Employee.lastname,                     -- last_name
    :Employee.lastname || '@example.com',   -- email
    SYSDATE,                                -- hire_date
    :Employee.job,                          -- job_id
    :Employee.sal,                          -- salary
    :Department.department_id               -- department_id
  );
END;
/
```

ネストした表に行を挿入する前のビューの問合せ:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

結果:

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
-----------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200))

1 row selected.
```

ネストした表に行を挿入する前の表の問合せ:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id = 10;
```

結果:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4200

1 row selected.
```

ネストした表への行の挿入:

```
INSERT INTO TABLE (
  SELECT d.emplist
  FROM dept_view d
  WHERE department_id = 10
)
VALUES (1001, 'Glenn', 'AC_MGR', 10000);
```

ネストした表に行を挿入した後のビューの問合せ:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

結果(ページに合わせて書式設定済):

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
--------------------------------------------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200),
          NTE(1001, 'Glenn', 'AC_MGR', 10000))

1 row selected.
```

ネストした表に行を挿入した後の表の問合せ:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id = 10;
```

結果:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4200
       1001 Glenn                     AC_MGR          10000

2 rows selected.
```

9.3.3 複合DMLトリガー

表またはエディショニング・ビューに対して作成された複合DMLトリガーは、複数のタイミングで起動できます。各タイミング部には、独自の実行部とオプションの例外処理部がありますが、それらのすべての部分から共通のPL/SQL状態にアクセスできます。共通の状態は、トリガーを起動する文でエラーが発生した場合でも、トリガーを起動する文の開始時に確立され、トリガーを起動する文の完了時に破棄されます。

非エディショニング・ビューに対して作成された複合DMLトリガーは、そのタイミング部が1つのみであるため、正確には複合トリガーではありません。

複合トリガーには条件を付けることができますが、自律型にはできません。

複合トリガーの一般的な2つの使用目的は次のとおりです。

- 定期的にバルク挿入できるように、2番目の表用の行を蓄積するため
- 変更表エラー(ORA-04091)を回避するため

ここでのトピック

- 複合DMLトリガーの構造
- 複合DMLトリガーの制限
- 複合DMLトリガーのパフォーマンス上のメリット
- バルク挿入での複合DMLトリガーの使用
- 複合DMLトリガーを使用した変更表エラーの回避

9.3.3.1 複合DMLトリガーの構造

複合トリガーのオプションの宣言部は、すべてのタイミング部で使用できる変数およびサブプログラムを宣言します。トリガーが起動されると、タイミング部が実行される前に宣言部が実行されます。変数およびサブプログラムは、トリガーを起動する文の継続期間中は存在します。

非エディショニング・ビューに対して作成された複合DMLトリガーは、そのタイミング部が1つのみであるため、正確には複合トリガーではありません。非エディショニング・ビューに対して最も単純な複合DMLトリガーを作成する構文は、次のとおりです。

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS BEGIN
  statement;
END INSTEAD OF EACH ROW;
```

表またはエディショニング・ビューに対して作成された複合DMLトリガーには、表9-2に示すタイミング部が1つ以上含まれます。トリガーに複数のタイミング部が含まれる場合、タイミング部は任意の順序で指定できますが、繰り返して使用することはできません。タイミング部がないと、そのタイミングでは何も実行されません。

表9-2 複合トリガーのタイミング部

タイミング	セクション
トリガーを起動する文が実行される前	BEFORE STATEMENT
トリガーを起動する文が実行された後	AFTER STATEMENT
トリガーを起動する文の影響を受ける各行の前	BEFORE EACH ROW
トリガーを起動する文の影響を受ける各行の後	AFTER EACH ROW

関連項目:

複合トリガーの構文の詳細は、「CREATE TRIGGER文」を参照してください

複合DMLトリガーに初期化部はありませんが、他のタイミング部の前に実行されるBEFORE STATEMENT部によって、必要な初期化をすべて実行できます。

複合DMLトリガーにBEFORE STATEMENT部もAFTER STATEMENT部もなく、トリガーを起動する文の影響を受ける行もない場合、トリガーは起動されません。

9.3.3.2 複合DMLトリガーの制限

複合DMLトリガーには、「トリガーの制限」の制限に加え、次の制限があります。

- OLD、NEWおよびPARENTは宣言部、BEFORE STATEMENT部またはAFTER STATEMENT部では使用できません。

- BEFORE EACH ROW部のみがNEWの値を変更できます。
- あるタイミング部は、別のタイミング部で呼び出される例外を処理できません。
- あるタイミング部にGOTO文が含まれている場合、GOTO文のターゲットは同じタイミング部に存在している必要があります。

9.3.3.3 複合DMLトリガーのパフォーマンス上のメリット

複合DMLトリガーには、トリガーを起動する文が多数の行に影響を与える場合にパフォーマンス上のメリットがあります。

たとえば、表9-2の4つのタイミング部をすべて含む複合DMLトリガーを、次の文が起動するとします。

```
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

トリガーのBEFORE EACH ROW部およびAFTER EACH ROW部は、列c1が0(ゼロ)より大きいSourceの各行に対して実行されますが、BEFORE STATEMENT部は、INSERT文の実行前にのみ実行され、AFTER STATEMENT部は、INSERT文の実行後にのみ実行されます。

複合DMLトリガーには、バルクSQL(「バルクSQLおよびバルク・バインド」を参照)を使用する場合により大きいパフォーマンス上のメリットがあります。

9.3.3.4 バルク挿入での複合DMLトリガーの使用

複合DMLトリガーは、定期的にバルク挿入できるように、2番目の表用の行を蓄積する場合に便利です。複合トリガーからパフォーマンス上のメリットを得るには、FORALL文にBULK COLLECT INTOを指定する必要があります(指定しないと、FORALL文は単一行のDML操作を複数回実行します)。FORALL文でのBULK COLLECT句の使用の詳細は、「FORALL文とBULK COLLECT句の併用」を参照してください。

関連項目:

FORALL文

使用例: 新しい表employee_salariesにhr.employees.salaryに対するすべての変更をログに記録する必要があります。単一のUPDATE文によってhr.employees表の多数の行が更新されるため、employee.salariesに行をバルク挿入する方が、行を個別に挿入するより効率的です。

解決方法: 例9-4に示すように、hr.employees表の更新に対して複合トリガーを定義します。idxまたはsalariesは、(トリガーを起動する文が中断され、再開された場合でも)トリガーが起動されるたびに初期化される状態変数であるため、これらを初期化するためにBEFORE STATEMENT部は必要ありません。

ノート:

例 9-4 を実行するには、DBMS_LOCK パッケージに対する EXECUTE 権限が必要です。

例9-4 複合トリガーによる、別の表にある表に対する変更のログへの記録

```
CREATE TABLE employee_salaries (
  employee_id NUMBER NOT NULL,
  change_date DATE   NOT NULL,
  salary NUMBER(8,2) NOT NULL,
  CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date),
  CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id)
    REFERENCES employees (employee_id)
      ON DELETE CASCADE)
/
CREATE OR REPLACE TRIGGER maintain_employee_salaries
  FOR UPDATE OF salary ON employees
    COMPOUND TRIGGER
-- Declarative Part:
-- Choose small threshhold value to show how example works:
  threshhold CONSTANT SIMPLE_INTEGER := 7;
  TYPE salaries_t IS TABLE OF employee_salaries%ROWTYPE INDEX BY SIMPLE_INTEGER;
  salaries  salaries_t;
  idx       SIMPLE_INTEGER := 0;
  PROCEDURE flush_array IS
    n CONSTANT SIMPLE_INTEGER := salaries.count();
  BEGIN
    FORALL j IN 1..n
      INSERT INTO employee_salaries VALUES salaries(j);
    salaries.delete();
    idx := 0;
    DBMS_OUTPUT.PUT_LINE('Flushed ' || n || ' rows');
  END flush_array;
  -- AFTER EACH ROW Section:
  AFTER EACH ROW IS
  BEGIN
    idx := idx + 1;
    salaries(idx).employee_id := :NEW.employee_id;
    salaries(idx).change_date := SYSTIMESTAMP;
    salaries(idx).salary := :NEW.salary;
    IF idx >= threshhold THEN
      flush_array();
    END IF;
  END AFTER EACH ROW;
  -- AFTER STATEMENT Section:
  AFTER STATEMENT IS
  BEGIN
    flush_array();
  END AFTER STATEMENT;
END maintain_employee_salaries;
/
```

部門50内のすべての従業員の給与を10%引き上げ:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 50
/
```

結果:

```
Flushed 7 rows
```

```
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows

45 rows updated.
```

2秒間待機:

```
BEGIN
  DBMS_LOCK.SLEEP(2);
END;
/
```

部門50内のすべての従業員の給与を5%引き上げ:

```
UPDATE employees
  SET salary = salary * 1.05
  WHERE department_id = 50
/
```

結果:

```
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows

45 rows updated.
```

従業員表への変更がemployee_salaries表に反映されていることを確認:

```
SELECT employee_id, count(*) c
  FROM employee_salaries
  GROUP BY employee_id
/
```

結果:

```
EMPLOYEE_ID          C
----------- ----------
        120          2
        121          2
        122          2
        123          2
        124          2
        125          2
...
        199          2

45 rows selected.
```

9.3.3.5 複合DMLトリガーを使用した変更表エラーの回避

複合DMLトリガーは、「変更表の制限」で説明されている変更表エラー(ORA-04091)を回避する場合に便利です。

使用例: ビジネス・ルールに、従業員の昇給はその従業員の部門の平均給与の10%を超えてはならないと記載されています。このルールを、トリガーによって実行する必要があります。

解決方法: 例9-5に示すように、hr.employees表の更新に対して複合トリガーを定義します。(トリガーを起動する文が中断され、再開された場合でも)トリガーが起動されるたびに、状態変数は初期化されます。

例9-5 複合トリガーによる変更表エラーの回避

```
CREATE OR REPLACE TRIGGER Check_Employee_Salary_Raise
  FOR UPDATE OF Salary ON Employees
COMPOUND TRIGGER
  Ten_Percent                 CONSTANT NUMBER := 0.1;
  TYPE Salaries_t             IS TABLE OF Employees.Salary%TYPE;
  Avg_Salaries                Salaries_t;
  TYPE Department_IDs_t       IS TABLE OF Employees.Department_ID%TYPE;
  Department_IDs              Department_IDs_t;
  -- Declare collection type and variable:
  TYPE Department_Salaries_t  IS TABLE OF Employees.Salary%TYPE
                                INDEX BY VARCHAR2(80);
  Department_Avg_Salaries     Department_Salaries_t;
  BEFORE STATEMENT IS
  BEGIN
    SELECT                AVG(e.Salary), NVL(e.Department_ID, -1)
      BULK COLLECT INTO   Avg_Salaries, Department_IDs
      FROM                Employees e
      GROUP BY            e.Department_ID;
    FOR j IN 1..Department_IDs.COUNT() LOOP
      Department_Avg_Salaries(Department_IDs(j)) := Avg_Salaries(j);
    END LOOP;
  END BEFORE STATEMENT;
  AFTER EACH ROW IS
  BEGIN
    IF :NEW.Salary - :Old.Salary >
      Ten_Percent*Department_Avg_Salaries(:NEW.Department_ID)
    THEN
      Raise_Application_Error(-20000, 'Raise too big');
    END IF;
  END AFTER EACH ROW;
END Check_Employee_Salary_Raise;
```

9.3.4 参照整合性を保証するトリガー

表9-3に示すとおり、トリガーおよび制約を使用して親表と子表の間の参照整合性を保持できます。(制約の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。)

表9-3 参照整合性を保証する制約およびトリガー

表	表で宣言する制約	表で作成するトリガー

表	表で宣言する制約	表で作成するトリガー
親	PRIMARY KEY または UNIQUE	PRIMARY KEY または UNIQUE の値が更新または削除されたときに、対応する FOREIGN KEY の値に対して目的のアクション(RESTRICT、CASCADE または SET NULL)が発生することを保証する 1 つ以上のトリガー。 親表への挿入にアクションは不要です(依存外部キーは存在しないため)。
子	FOREIGN KEY(親と子が同じデータベースに存在する場合)。(データベースでは、分散データベースの異なるノード上に存在する表の間の宣言参照制約はサポートされません。) この外部キー制約を無効化して、対応する PRIMARY KEY 制約または UNIQUE 制約が削除されることを防止します(CASCADE オプションを使用して明示的に削除する場合を除く)。	FOREIGN KEY で挿入または更新される値が親表の PRIMARY KEY または UNIQUE の値に対応することを保証する 1 つのトリガー。

ここでのトピック

- 子表に対する外部キー・トリガー
- 親表に対するUPDATEおよびDELETE RESTRICTトリガー
- 親表に対するUPDATEおよびDELETE SET NULLトリガー
- 親表に対するDELETE CASCADEトリガー
- 親表に対するUPDATE CASCADEトリガー
- 複雑な制約チェックに対するトリガー
- 複雑なセキュリティ認可に対するトリガー
- 透過的なイベント・ロギングに対するトリガー
- 導出列値に対するトリガー
- 複合更新可能ビューを作成するトリガー
- ファイングレイン・アクセス制御に対するトリガー

ノート:

後続のトピックの例では、列 Deptno を共有する次の表を使用します。

```
CREATE TABLE emp (
  Empno      NUMBER NOT NULL,
  Ename      VARCHAR2(10),
```

```
  Job        VARCHAR2(9),
  Mgr        NUMBER(4),
  Hiredate   DATE,
  Sal        NUMBER(7,2),
  Comm       NUMBER(7,2),
  Deptno     NUMBER(2) NOT NULL);
CREATE TABLE dept (
  Deptno     NUMBER(2) NOT NULL,
  Dname      VARCHAR2(14),
  Loc        VARCHAR2(13),
  Mgr_no     NUMBER,
  Dept_type NUMBER);
```

一部のトリガーには、行をロックする文(SELECT FOR UPDATE)が含まれています。この操作は、行の処理中に同時実行性を保持するために必要です。

これらの例は、記述されているとおりに使用する必要はありません。これらの例は、ユーザー独自のトリガーの設計を支援するために提供されています。

9.3.4.1 子表に対する外部キー・トリガー

例9-6のトリガーは、INSERT文またはUPDATE文が外部キー値に影響する前に、対応する値が親キー内に確実に存在することを保証します。例外ORA-04091(変更表エラー)によって、emp_dept_checkトリガーをUPDATE_SET_DEFAULTトリガーおよびUPDATE_CASCADEトリガーとともに使用できるようになります。emp_dept_checkトリガーを単独で使用する場合、この例外は必要ありません。

例9-6 子表に対する外部キー・トリガー

```
CREATE OR REPLACE TRIGGER emp_dept_check
  BEFORE INSERT OR UPDATE OF Deptno ON emp
  FOR EACH ROW WHEN (NEW.Deptno IS NOT NULL)
  -- Before row is inserted or DEPTNO is updated in emp table,
  -- fire this trigger to verify that new foreign key value (DEPTNO)
  -- is present in dept table.
DECLARE
  Dummy               INTEGER;  -- Use for cursor fetch
  Invalid_department  EXCEPTION;
  Valid_department    EXCEPTION;
  Mutating_table      EXCEPTION;
  PRAGMA EXCEPTION_INIT (Invalid_department, -4093);
  PRAGMA EXCEPTION_INIT (Valid_department, -4092);
  PRAGMA EXCEPTION_INIT (Mutating_table, -4091);
  -- Cursor used to verify parent key value exists.
  -- If present, lock parent key's row so it cannot be deleted
  -- by another transaction until this transaction is
  -- committed or rolled back.
  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM dept
    WHERE Deptno = Dn
    FOR UPDATE OF Deptno;
BEGIN
  OPEN Dummy_cursor (:NEW.Deptno);
  FETCH Dummy_cursor INTO Dummy;
  -- Verify parent key.
```

```
  -- If not found, raise user-specified error code and message.
  -- If found, close cursor before allowing triggering statement to complete:
  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_department;
  ELSE
    RAISE Valid_department;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_department THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20000, 'Invalid Department'
      || ' Number' || TO_CHAR(:NEW.deptno));
  WHEN Valid_department THEN
    CLOSE Dummy_cursor;
  WHEN Mutating_table THEN
    NULL;
END;
/
```

9.3.4.2 親表に対するUPDATEおよびDELETE RESTRICTトリガー

例9-7のトリガーは、dept表の主キーに対してUPDATEおよびDELETE RESTRICT参照アクションを実行します。

> 注意:
>
> 例 9-7 のトリガーは、自己参照型の表(主キーまたは一意キーおよび外部キーが存在する表)では機能しません。また、このトリガーでは、トリガーの循環(A が B を起動し、B が A を起動する場合など)は実行できません。

例9-7 親表に対するUPDATEおよびDELETE RESTRICTトリガー

```
CREATE OR REPLACE TRIGGER dept_restrict
  BEFORE DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW
  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- check for dependent foreign key values in emp;
  -- if any are found, roll back.
DECLARE
  Dummy                   INTEGER;  -- Use for cursor fetch
  employees_present       EXCEPTION;
  employees_not_present   EXCEPTION;
  PRAGMA EXCEPTION_INIT (employees_present, -4094);
  PRAGMA EXCEPTION_INIT (employees_not_present, -4095);
  -- Cursor used to check for dependent foreign key values.
  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM emp WHERE Deptno = Dn;
BEGIN
  OPEN Dummy_cursor (:OLD.Deptno);
  FETCH Dummy_cursor INTO Dummy;
  -- If dependent foreign key is found, raise user-specified
  -- error code and message. If not found, close cursor
  -- before allowing triggering statement to complete.
  IF Dummy_cursor%FOUND THEN
    RAISE employees_present;     -- Dependent rows exist
  ELSE
```

```
    RAISE employees_not_present; -- No dependent rows exist
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN employees_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Employees Present in'
      || ' Department ' || TO_CHAR(:OLD.DEPTNO));
  WHEN employees_not_present THEN
    CLOSE Dummy_cursor;
END;
```

9.3.4.3 親表に対するUPDATEおよびDELETE SET NULLトリガー

例9-8のトリガーは、dept表の主キーに対してUPDATEおよびDELETE SET NULL参照アクションを実行します。

例9-8 親表に対するUPDATEおよびDELETE SET NULLトリガー

```
CREATE OR REPLACE TRIGGER dept_set_null
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW
  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- set all corresponding dependent foreign key values in emp to NULL:
BEGIN
  IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
    UPDATE emp SET emp.Deptno = NULL
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
```

9.3.4.4 親表に対するDELETE CASCADEトリガー

例9-9のトリガーは、dept表の主キーに対してDELETE CASCADE参照アクションを実行します。

ノート:

通常、DELETE CASCADE のコードは、更新および削除の両方の可能性を考慮して、UPDATE SET NULL または UPDATE SET DEFAULT のコードと組み合されます。

例9-9 親表に対するDELETE CASCADEトリガー

```
CREATE OR REPLACE TRIGGER dept_del_cascade
  AFTER DELETE ON dept
  FOR EACH ROW
  -- Before row is deleted from dept,
  -- delete all rows from emp table whose DEPTNO is same as
  -- DEPTNO being deleted from dept table:
BEGIN
  DELETE FROM emp
  WHERE emp.Deptno = :OLD.Deptno;
END;
/
```

9.3.4.5 親表に対するUPDATE CASCADEトリガー

例9-10のトリガーは、dept表の部門番号が更新された場合に、その変更がemp表の依存外部キーに確実に伝播されることを保証します。

> ノート:
>
> dept_cascade2 トリガーによって emp 表が更新されるため、例 9-6 の emp_dept_check トリガー(有効になっている場合)も起動されます。結果として生成される変更表エラーは、emp_dept_check トリガーによってトラップされます。エラーのトラップが必要なトリガーは、使用環境で常に正常に動作することを保証できるように、慎重にテストしてください。

例9-10 親表に対するUPDATE CASCADEトリガー

```
-- Generate sequence number to be used as flag
-- for determining if update occurred on column:
CREATE SEQUENCE Update_sequence
  INCREMENT BY 1 MAXVALUE 5000 CYCLE;
CREATE OR REPLACE PACKAGE Integritypackage AUTHID DEFINER AS
  Updateseq NUMBER;
END Integritypackage;
/
CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
/
-- Create flag col:
ALTER TABLE emp ADD Update_id NUMBER;
CREATE OR REPLACE TRIGGER dept_cascade1
  BEFORE UPDATE OF Deptno ON dept
DECLARE
  -- Before updating dept table (this is a statement trigger),
  -- generate sequence number
  -- & assign it to public variable UPDATESEQ of
  -- user-defined package named INTEGRITYPACKAGE:
BEGIN
  Integritypackage.Updateseq := Update_sequence.NEXTVAL;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade2
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW
  -- For each department number in dept that is updated,
  -- cascade update to dependent foreign keys in emp table.
  -- Cascade update only if child row was not updated by this trigger:
BEGIN
  IF UPDATING THEN
    UPDATE emp
    SET Deptno = :NEW.Deptno,
        Update_id = Integritypackage.Updateseq   --from 1st
    WHERE emp.Deptno = :OLD.Deptno
    AND Update_id IS NULL;
    /* Only NULL if not updated by 3rd trigger
       fired by same triggering statement */
  END IF;
```

```
  IF DELETING THEN
    -- After row is deleted from dept,
    -- delete all rows from emp table whose DEPTNO is same as
    -- DEPTNO being deleted from dept table:
    DELETE FROM emp
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade3
  AFTER UPDATE OF Deptno ON dept
BEGIN UPDATE emp
  SET Update_id = NULL
  WHERE Update_id = Integritypackage.Updateseq;
END;
/
```

9.3.4.6 複雑な制約チェックに対するトリガー

トリガーは、参照整合性以外の整合性規則も実行できます。例9-11のトリガーは、トリガーを起動する文の実行を許可する前に、複雑なチェックを実行します。

ノート:

例 9-11 では、次のデータ構造が必要です。

```
CREATE TABLE Salgrade (
  Grade               NUMBER,
  Losal               NUMBER,
  Hisal               NUMBER,
  Job_classification  VARCHAR2(9));
```

例9-11 トリガーによる複雑な制約のチェック

```
CREATE OR REPLACE TRIGGER salary_check
  BEFORE INSERT OR UPDATE OF Sal, Job ON Emp
  FOR EACH ROW
DECLARE
  Minsal               NUMBER;
  Maxsal               NUMBER;
  Salary_out_of_range  EXCEPTION;
  PRAGMA EXCEPTION_INIT (Salary_out_of_range, -4096);
BEGIN
  /* Retrieve minimum & maximum salary for employee's new job classification
     from SALGRADE table into MINSAL and MAXSAL: */
  SELECT Losal, Hisal INTO Minsal, Maxsal
  FROM Salgrade
  WHERE Job_classification = :NEW.Job;
  /* If employee's new salary is less than or greater than
     job classification's limits, raise exception.
     Exception message is returned and pending INSERT or UPDATE statement
     that fired the trigger is rolled back: */
  IF (:NEW.Sal < Minsal OR :NEW.Sal > Maxsal) THEN
    RAISE Salary_out_of_range;
  END IF;
```

```
EXCEPTION
  WHEN Salary_out_of_range THEN
    Raise_application_error (
      -20300,
      'Salary '|| TO_CHAR(:NEW.Sal) ||' out of range for '
      || 'job classification ' ||:NEW.Job
      ||' for employee ' || :NEW.Ename
    );
  WHEN NO_DATA_FOUND THEN
    Raise_application_error(-20322, 'Invalid Job Classification');
END;
/
```

9.3.4.7 複雑なセキュリティ認可に対するトリガー

トリガーは、表データに対する複雑なセキュリティ認可の実行によく使用されます。トリガーは、データベースで提供されるデータベース・セキュリティ機能では定義できない複雑なセキュリティ認可の実行にのみ使用します。たとえば、トリガーを使用して、週末および休業時間にemployee表を更新できないようにすることができます。

複雑なセキュリティ認可の実行にトリガーを使用する場合は、BEFORE文トリガーを使用することをお薦めします。BEFORE文トリガーを使用すると、次のメリットがあります。

- トリガーを起動する文の実行が許可される前にセキュリティ・チェックが実行されるため、不正な文による無駄な作業は行われません。
- セキュリティ・チェックは、トリガーを起動する文に影響される各行に対してではなく、トリガーを起動する文に対してのみ実施されます。

例9-12のトリガーは、ユーザーが週末または休業時間にemployee表の更新を試行した場合に例外を呼び出すことで、セキュリティを施行します。

関連項目:

データベース・セキュリティ機能の詳細は『Oracle Databaseセキュリティ・ガイド』を参照してください。

例9-12 トリガーによるセキュリティ認可の施行

```
CREATE OR REPLACE TRIGGER Employee_permit_changes
  BEFORE INSERT OR DELETE OR UPDATE ON employees
DECLARE
  Dummy              INTEGER;
  Not_on_weekends    EXCEPTION;
  Nonworking_hours   EXCEPTION;
  PRAGMA EXCEPTION_INIT (Not_on_weekends, -4097);
  PRAGMA EXCEPTION_INIT (Nonworking_hours, -4099);
BEGIN
   -- Check for weekends:

   IF (TO_CHAR(Sysdate, 'DAY') = 'SAT' OR
     TO_CHAR(Sysdate, 'DAY') = 'SUN') THEN
       RAISE Not_on_weekends;
   END IF;
```

```
  -- Check for work hours (8am to 6pm):

  IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
    TO_CHAR(Sysdate, 'HH24') > 18) THEN
      RAISE Nonworking_hours;
  END IF;

EXCEPTION
  WHEN Not_on_weekends THEN
    Raise_application_error(-20324,'Might not change '
      ||'employee table during the weekend');
  WHEN Nonworking_hours THEN
    Raise_application_error(-20326,'Might not change '
     ||'emp table during Nonworking hours');
END;
/
```

9.3.4.8 透過的なイベント・ロギングに対するトリガー

特定のイベントに続いて、関連する変更をデータベースで透過的に実行する場合、トリガーは非常に有効です。

REORDERトリガーの例は、一定の条件が満たされると、必要に応じて部品を再注文するトリガーを示しています。(つまり、トリガーを起動する文が入力され、PARTS_ON_HAND値がREORDER_POINT値より小さい場合です。)

9.3.4.9 導出列値に対するトリガー

トリガーは、INSERT文またはUPDATE文で指定される値に基づいて、列の値を自動的に導出できます。このタイプのトリガーは、同じ行内の他の列値に依存する特定の列に値を強制的に設定する場合に有効です。次に示す理由から、このタイプの操作を実行するにはBEFORE行トリガーが必要です。

- トリガーを起動する文で導出値を使用できるようにするために、INSERTまたはUPDATEが発生する前に依存値を導出する必要があります。
- トリガーを起動するINSERT文またはUPDATE文によって影響される各行に対して、トリガーを起動する必要があります。

例9-13のトリガーは、行が挿入または更新されるたびに、表の新しい列値を導出します。

ノート:

例 9-13 では、データ構造に対する次のような変更が必要です。

```
ALTER TABLE Emp ADD(
   Uppername    VARCHAR2(20),
   Soundexname VARCHAR2(20));
```

例9-13 トリガーによる新しい列値の導出

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp
/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Restrict users
   from updating these fields directly: */
FOR EACH ROW
```

```
BEGIN
  :NEW.Uppername := UPPER(:NEW.Ename);
  :NEW.Soundexname := SOUNDEX(:NEW.Ename);
END;
/
```

9.3.4.10 複合更新可能ビューを作成するトリガー

ビューは、表データに対して論理ウィンドウを提供するための優れたメカニズムです。ただし、ビューの問合せが複雑になると、ビューに対するDMLから基礎となる表に対するDMLへの変換を、システムで暗黙的に実行できなくなります。この問題の解決には、INSTEAD OFトリガーが有効です。このトリガーは、ビューに対して定義でき、実際のDMLのかわりに起動されます。

書籍が書名の順に配置されているライブラリ・システムについて考えてみます。このライブラリは、次の一連の書籍型オブジェクトで構成されています。

```
CREATE OR REPLACE TYPE Book_t AS OBJECT (
  Booknum    NUMBER,
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);
/
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
/
```

次のように、表Book_tableが作成されデータが入れられます。

```
DROP TABLE Book_table;
CREATE TABLE Book_table (
  Booknum    NUMBER,
  Section    VARCHAR2(20),
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121001, 'Classic', 'Iliad', 'Homer', 'Y'
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121002, 'Novel', 'Gone with the Wind', 'Mitchell M', 'N'
);

SELECT * FROM Book_table ORDER BY Booknum;
```

結果:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
```

```
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N

2 rows selected.
```

次のように、表Library_tableが作成されデータが入れられます。

```
DROP TABLE Library_table;
CREATE TABLE Library_table (Section VARCHAR2(20));

INSERT INTO Library_table (Section)
VALUES ('Novel');

INSERT INTO Library_table (Section)
VALUES ('Classic');

SELECT * FROM Library_table ORDER BY Section;
```

結果:

```
SECTION
--------------------
Classic
Novel

2 rows selected.
```

次のように、表Book_tableおよびLibrary_tableに対して複合ビューを定義し、セクションおよび各セクション内の一連の書籍を示すライブラリの論理ビューを作成できます。

```
CREATE OR REPLACE VIEW Library_view AS
  SELECT i.Section, CAST (
    MULTISET (
      SELECT b.Booknum, b.Title, b.Author, b.Available
      FROM Book_table b
      WHERE b.Section = i.Section
    ) AS Book_list_t
  ) BOOKLIST
  FROM Library_table i;
```

(CASTファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。)

次のように、Library_viewに対してINSTEAD OFトリガーを定義することによって、このビューを更新可能にします。

```
CREATE OR REPLACE TRIGGER Library_trigger
  INSTEAD OF
  INSERT ON Library_view
  FOR EACH ROW
DECLARE
  Bookvar  Book_t;
  i        INTEGER;
BEGIN
  INSERT INTO Library_table
  VALUES (:NEW.Section);

  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := :NEW.Booklist(i);
```

```
    INSERT INTO Book_table (
      Booknum, Section, Title, Author, Available
    )
    VALUES (
      Bookvar.booknum, :NEW.Section, Bookvar.Title,
      Bookvar.Author, bookvar.Available
    );
  END LOOP;
END;
/
```

Library_viewへの新しい行の挿入:

```
INSERT INTO Library_view (Section, Booklist)
VALUES (
  'History',
  book_list_t (book_t (121330, 'Alexander', 'Mirth', 'Y'))
);
```

Library_viewに対する効果の確認:

```
SELECT * FROM Library_view ORDER BY Section;
```

結果:

```
SECTION
--------------------
BOOKLIST(BOOKNUM, TITLE, AUTHOR, AVAILABLE)
----------------------------------------------------------------------


Classic
BOOK_LIST_T(BOOK_T(121001, 'Iliad', 'Homer', 'Y'))

History
BOOK_LIST_T(BOOK_T(121330, 'Alexander', 'Mirth', 'Y'))

Novel
BOOK_LIST_T(BOOK_T(121002, 'Gone with the Wind', 'Mitchell M', 'N'))


3 rows selected.
```

Book_tableに対する効果の確認:

```
SELECT * FROM Book_table ORDER BY Booknum;
```

結果:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N
    121330 History              Alexander            Mirth                Y

3 rows selected.
```

Library_tableに対する効果の確認:

```
SELECT * FROM Library_table ORDER BY Section;
```

結果:

```
SECTION
--------------------
Classic
History
Novel

3 rows selected.
```

同様に、ネストした表booklistに対してトリガーを定義して、このネストした表の要素の変更を処理することもできます。

9.3.4.11 ファイングレイン・アクセス制御に対するトリガー

LOGONトリガーを使用すると、アプリケーション・コンテキストに関連付けられたパッケージを実行できます。アプリケーション・コンテキストによって、データベースにログインしているユーザーのセッション関連の情報が取得されます。アプリケーションから、ユーザーのセッション情報に基づいて、ユーザーが持つアクセス権を制御できます。

> ノート:
>
> ユーザーによるファイアウォールの外側からのログインや勤務時間外のログインの防止などの特別なログオン要件がある場合は、LOGON トリガーではなく Oracle Database Vault を使用することを検討してください。Oracle Database Vault を使用すると、ユーザーのアクセスを厳密に制御する独自の規則を作成できます。

関連項目:

- LOGONトリガーを作成してデータベース・セッションのアプリケーション・コンテキスト・パッケージを実行する方法の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。
- Oracle Database Vaultの詳細は、『Oracle Database Vault管理者ガイド』を参照してください。

9.4 相関名および疑似レコード

> ノート:
>
> このトピックの内容は、行レベルで起動されるトリガーにのみ適用されます。つまり、次のようになります。
>
> - 行レベルの単純な DML トリガー
> - 行レベルのタイミング・セクションを持つ複合 DML トリガー

行レベルで起動されるトリガーでは、相関名を使用して処理中の行のデータにアクセスできます。デフォルトの相関名はOLD、

NEWおよびPARENTです。相関名を変更するには、CREATE TRIGGER文のREFERENCING句を使用します(「referencing_clause ::=」を参照)。

ネストした表にトリガーが作成されている場合、OLDおよびNEWはネストした表の現在の行を参照し、PARENTは親表の現在の行を参照します。表またはビューに対してトリガーが作成される場合、OLDおよびNEWはその表またはビューの現在の行を参照し、PARENTは未定義になります。

OLD、NEWおよびPARENTは、レコード構造を保持しているものの、使用が許可されるコンテキストがレコードより少ないため、疑似レコードとも呼ばれます。疑似レコードの構造は、table_name%ROWTYPEであり、ここでtable_nameはトリガーが作成される表の名前(OLDおよびNEWの場合)または親表の名前(PARENTの場合)です。

単純なトリガーのtrigger_bodyまたは複合トリガーのtps_bodyでは、相関名はバインド変数のプレースホルダです。疑似レコードのフィールドを参照するには、次の構文を使用します。

```
:pseudorecord_name.field_name
```

条件付きトリガーのWHEN句では、相関名はバインド変数のプレースホルダではありません。そのため、前述の構文にあるコロンは省略してください。

表9-4に、トリガーを起動する文の処理対象となる行のOLDフィールドとNEWフィールドの値を示します。

表9-4 OLDおよびNEW疑似レコードのフィールド値

トリガーを起動する文	OLDフィールド値	NEWフィールド値
INSERT	NULL	挿入後の値
UPDATE	更新前の値	更新後の値
DELETE	削除前の値	NULL

疑似レコードの制限は次のとおりです。

- 疑似レコードは、レコードレベルの操作では使用できません。

  たとえば、トリガーに次の文を含めることはできません。

  ```
  :NEW := NULL;
  ```

- 疑似レコードは、サブプログラムの実パラメータとして使用できません。

  (疑似レコードのフィールドは、サブプログラムの実パラメータとして使用できます。)

- トリガーでは、OLDフィールドの値を変更できません。

  この操作を試行すると、ORA-04085が呼び出されます。

- トリガーを起動する文がDELETEの場合、トリガーでは、NEWフィールドの値を変更できません。

  この操作を試行すると、ORA-04084が呼び出されます。

- トリガーを起動する文は、トリガーの起動前に実行されるため、AFTERトリガーでは、NEWフィールドの値を変更できませ

ん。

この操作を試行すると、ORA-04084が呼び出されます。

BEFOREトリガーでは、トリガーを起動するINSERT文またはUPDATE文が表に値を設定する前に、NEWフィールドの値を変更できます。

文がBEFOREトリガーとAFTERトリガーの両方を起動する場合、BEFOREトリガーがNEWフィールドの値を変更すると、AFTERトリガーではその変更を認識できます。

例9-14 トリガーによるEMPLOYEES.SALARYの変更の記録

この例では、ログ表を作成し、UPDATE文がEMPLOYEES表のSALARY列に影響を与えた後にそのログ表に行を挿入するトリガーを作成します。その後、EMPLOYEES.SALARYを更新し、ログ表を表示します。

ログ表の作成:

```
DROP TABLE Emp_log;
CREATE TABLE Emp_log (
  Emp_id     NUMBER,
  Log_date   DATE,
  New_salary NUMBER,
  Action     VARCHAR2(20));
```

EMPLOYEES.SALARYの更新後にログ表に行を挿入するトリガーの作成:

```
CREATE OR REPLACE TRIGGER log_salary_increase
  AFTER UPDATE OF salary ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
  VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'New Salary');
END;
/
```

EMPLOYEES.SALARYの更新:

```
UPDATE employees
SET salary = salary + 1000.0
WHERE Department_id = 20;
```

結果:

```
2 rows updated.
```

ログ表の表示:

```
SELECT * FROM Emp_log;
```

結果:

```
    EMP_ID LOG_DATE  NEW_SALARY ACTION
```

```
---------- --------- ---------- --------------------
       201 28-APR-10      13650 New Salary
       202 28-APR-10       6300 New Salary

2 rows selected.
```

例9-15 条件付きトリガーによる給与変更情報の出力

この例では、DELETE文、INSERT文またはUPDATE文がEMPLOYEES表に影響するたびに給与変更情報を(それが社長の情報でなければ)出力する条件付きトリガーを作成します。データベースでは、影響を受ける行ごとにWHEN条件が評価されます。影響を受ける行に対してWHEN条件がTRUEになると、トリガーを起動する文が実行される前にその行でトリガーが起動されます。影響を受ける行に対してWHEN条件がTRUEにならない場合、トリガーはその行で起動されませんが、トリガーを起動する文はそのまま実行されます。

```
CREATE OR REPLACE TRIGGER print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')  -- do not print information about President
DECLARE
  sal_diff  NUMBER;
BEGIN
  sal_diff  := :NEW.salary  - :OLD.salary;
  DBMS_OUTPUT.PUT(:NEW.last_name || ': ');
  DBMS_OUTPUT.PUT('Old salary = ' || :OLD.salary || ', ');
  DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
  DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);
END;
/
```

問合せ:

```
SELECT last_name, department_id, salary, job_id
FROM employees
WHERE department_id IN (10, 20, 90)
ORDER BY department_id, last_name;
```

結果:

```
LAST_NAME                 DEPARTMENT_ID     SALARY JOB_ID
------------------------- ------------- ---------- ----------
Whalen                               10       4200 AD_ASST
Fay                                  20       6000 MK_REP
Hartstein                            20      13000 MK_MAN
De Haan                              90      17000 AD_VP
King                                 90      24000 AD_PRES
Kochhar                              90      17000 AD_VP

6 rows selected.
```

トリガーを起動する文:

```
UPDATE employees
SET salary = salary * 1.05
WHERE department_id IN (10, 20, 90);
```

結果:

```
Whalen: Old salary = 4200, New salary = 4410, Difference: 210
Hartstein: Old salary = 13000, New salary = 13650, Difference: 650
Fay: Old salary = 6000, New salary = 6300, Difference: 300
Kochhar: Old salary = 17000, New salary = 17850, Difference: 850
De Haan: Old salary = 17000, New salary = 17850, Difference: 850

6 rows updated.
```

問合せ:

```
SELECT salary FROM employees WHERE job_id = 'AD_PRES';
```

結果:

```
    SALARY
----------
     25200

1 row selected.
```

例9-16 トリガーによるCLOB列の変更

この例では、CLOB列を変更するUPDATEトリガーを作成します。

TO_CLOBおよび他の変換ファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

```
DROP TABLE tab1;
CREATE TABLE tab1 (c1 CLOB);
INSERT INTO tab1 VALUES ('<h1>HTML Document Fragment</h1><p>Some text.', 3);
CREATE OR REPLACE TRIGGER trg1
  BEFORE UPDATE ON tab1
  FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Old value of CLOB column: '||:OLD.c1);
  DBMS_OUTPUT.PUT_LINE('Proposed new value of CLOB column: '||:NEW.c1);
  :NEW.c1 := :NEW.c1 || TO_CLOB('<hr><p>Standard footer paragraph.');
  DBMS_OUTPUT.PUT_LINE('Final value of CLOB column: '||:NEW.c1);
END;
/
SET SERVEROUTPUT ON;
UPDATE tab1 SET c1 = '<h1>Different Document Fragment</h1><p>Different text.';
SELECT * FROM tab1;
```

例9-17 REFERENCING句のあるトリガー

この例では、相関名と同じ名前の表newを作成し、その表に対するトリガーを作成します。表名と相関名の競合を回避するため、トリガーは、相関名をNewestとして参照します。

```
CREATE TABLE new (
  field1  NUMBER,
  field2  VARCHAR2(20)
);
CREATE OR REPLACE TRIGGER Print_salary_changes
BEFORE UPDATE ON new
```

```
REFERENCING new AS Newest
FOR EACH ROW
BEGIN
  :Newest.Field2 := TO_CHAR (:newest.field1);
END;
/
```

9.4.1 OBJECT_VALUE疑似列

オブジェクト表に対するDMLトリガーでは、オブジェクト表の列のシステム生成名を戻す、SQL擬似列OBJECT_VALUEを参照できます。トリガーでは、OBJECT_VALUEのデータ型を持つIN仮パラメータを含むPL/SQLサブプログラムを起動することもできます。

関連項目:

- OBJECT_VALUEの一般情報は、『Oracle Database SQL言語リファレンス』を参照してください。
- 疑似列の概要は、『Oracle Database SQL言語リファレンス』を参照してください。

例9-18では、オブジェクト表tbl、tblの更新を記録するための表tbl_history、およびトリガーTbl_Trgを作成します。DML文の影響を受けるtb1の行ごとにトリガーが実行され、tbl内のオブジェクトtの古い値および新しい値がtbl_historyに書き込まれます。古い値と新しい値は、:OLD.OBJECT_VALUEおよび:NEW.OBJECT_VALUEです。

列nのすべての値が1増えました。mの値は0のままです。

例9-18 トリガーによるOBJECT_VALUE疑似列の参照

オブジェクト表の作成、データ入れ、および表示:

```
CREATE OR REPLACE TYPE t AUTHID DEFINER AS OBJECT (n NUMBER, m NUMBER)
/
CREATE TABLE tbl OF t
/
BEGIN
  FOR j IN 1..5 LOOP
    INSERT INTO tbl VALUES (t(j, 0));
  END LOOP;
END;
/
SELECT * FROM tbl ORDER BY n;
```

結果:

```
         N          M
---------- ----------
         1          0
         2          0
         3          0
         4          0
         5          0
5 rows selected.
```

履歴表とトリガーの作成:

```
CREATE TABLE tbl_history ( d DATE, old_obj t, new_obj t)
```

```
/
CREATE OR REPLACE TRIGGER Tbl_Trg
  AFTER UPDATE ON tbl
  FOR EACH ROW
BEGIN
  INSERT INTO tbl_history (d, old_obj, new_obj)
  VALUES (SYSDATE, :OLD.OBJECT_VALUE, :NEW.OBJECT_VALUE);
END Tbl_Trg;
/
```

オブジェクト表の更新:

```
UPDATE tbl SET tbl.n = tbl.n+1
/
```

結果:

```
5 rows updated.
```

古い値と新しい値の表示:

```
BEGIN
  FOR j IN (SELECT d, old_obj, new_obj FROM tbl_history) LOOP
    DBMS_OUTPUT.PUT_LINE (
      j.d ||
      ' -- old: ' || j.old_obj.n || ' ' || j.old_obj.m ||
      ' -- new: ' || j.new_obj.n || ' ' || j.new_obj.m
    );
  END LOOP;
END;
/
```

結果:

```
28-APR-10 -- old: 1 0 -- new: 2 0
28-APR-10 -- old: 2 0 -- new: 3 0
28-APR-10 -- old: 3 0 -- new: 4 0
28-APR-10 -- old: 4 0 -- new: 5 0
28-APR-10 -- old: 5 0 -- new: 6 0
```

9.5 システム・トリガー

システム・トリガーは、スキーマまたはデータベースのいずれかに対して作成されます。

このトリガーを起動するイベントは、DDL文(「ddl_event」のリストを参照)またはデータベース操作文(「database_event」のリストを参照)で構成されます。

システム・トリガーは、次のいずれかのタイミングで起動されます。

- トリガーを起動する文が実行される前

  (このトリガーは、BEFORE文トリガーまたは文レベルのBEFOREトリガーと呼ばれます。)

- トリガーを起動する文が実行された後

  (このトリガーは、AFTER文トリガーまたは文レベルのAFTERトリガーと呼ばれます。)

- CREATE文を起動するかわり

  (このトリガーは、INSTEAD OF CREATEトリガーと呼ばれます。)

ここでのトピック

- SCHEMAトリガー
- DATABASEトリガー
- INSTEAD OF CREATEトリガー

9.5.1 SCHEMAトリガー

SCHEMAトリガーは、スキーマに対して作成され、そのスキーマの所有者であるユーザーが現行ユーザーで、トリガー・イベントを開始するたびに起動されます。

user1とuser2の両方がSCHEMAトリガーを所有していて、user2が所有するDRユニットをuser1が起動するとします。DRユニット内では、user2が現行ユーザーになります。したがって、user2が所有するSCHEMAトリガーのトリガー・イベントをDRユニットが開始すると、そのトリガーが起動されます。ただし、user1が所有するSCHEMAトリガーのトリガー・イベントをDRユニットが開始しても、そのトリガーは起動されません。

例9-19では、サンプル・スキーマHRにBEFORE文トリガーを作成します。HRとして接続しているユーザーがデータベース・オブジェクトを削除しようとすると、オブジェクトの削除前に、データベースによってこのトリガーが起動されます。

例9-19 サンプル・スキーマHRに対するBEFORE文トリガー

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON hr.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR (
      num => -20000,
      msg => 'Cannot drop object');
  END;
/
```

9.5.2 DATABASEトリガー

DATABASEトリガーは、データベースに対して作成され、データベース・ユーザーがトリガー・イベントを開始するたびに起動されます。

例9-20では、エラーのログを記録するトリガーの基本構文を示します。このトリガーは、文の実行の失敗(ログオンの失敗など)の後で起動されます。

ノート:

AFTER SERVERERROR トリガーは、Oracle リレーショナル・データベース管理システム(RDBMS)が、エラー・トリガーを起動することが妥当であると判断した場合にのみ起動されます。AFTER SERVERERROR トリガーの詳細は、

「CREATE TRIGGER 文」を参照してください。

例9-21のトリガーは、ユーザーがデータベースにログオンした後にプロシージャcheck_userを実行します。

例9-20 データベースに対するAFTER文トリガー

```
CREATE TRIGGER log_errors
  AFTER SERVERERROR ON DATABASE
  BEGIN
    IF (IS_SERVERERROR (1017)) THEN
      NULL;  -- (substitute code that processes logon error)
    ELSE
      NULL;  -- (substitute code that logs error code)
    END IF;
  END;
/
```

例9-21 トリガーによるログオンの監視

```
CREATE OR REPLACE TRIGGER check_user
  AFTER LOGON ON DATABASE
  BEGIN
    check_user;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR
        (-20000, 'Unexpected error: '|| DBMS_Utility.Format_Error_Stack);
 END;
/
```

9.5.3 INSTEAD OF CREATEトリガー

INSTEAD OF CREATEトリガーは、トリガー・イベントがCREATE文であるSCHEMAトリガーです。データベースでは、トリガーを起動する文が実行されるかわりに、トリガー自体が起動されます。

例9-22は、現在のスキーマに対するINSTEAD OF CREATEトリガーの基本構文を示しています。このトリガーは、現在のスキーマの所有者が現在のスキーマでCREATE文を発行すると起動されます。

例9-22 スキーマに対するINSTEAD OF CREATEトリガー

```
CREATE OR REPLACE TRIGGER t
  INSTEAD OF CREATE ON SCHEMA
  BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE T (n NUMBER, m NUMBER)';
  END;
/
```

9.6 トリガーによって起動されるサブプログラム

トリガーでは、PL/SQL、CおよびJavaで記述されたサブプログラムを起動できます。例9-4のトリガーは、PL/SQLサブプログラムを起動します。例9-23のトリガーは、Javaサブプログラムを起動します。

トリガーによって起動されるサブプログラムは、トリガー本体のコンテキスト内で実行されるため、このようなサブプログラムではトラン

ザクション制御文は実行できません。

トリガーによって実行者権限(IR)サブプログラムが起動される場合は、トリガーを起動する文を実行したユーザーではなく、トリガーを作成したユーザーが現行ユーザーとみなされます。IRサブプログラムの詳細は、「実行者権限および定義者権限(AUTHIDプロパティ)」を参照してください。

トリガーがリモート・サブプログラムを起動し、トリガーの実行中にタイムスタンプまたはシグネチャの不一致が検出されると、リモート・サブプログラムは実行されず、トリガーは無効になります。

例9-23 トリガーによるJavaサブプログラムの起動

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS LANGUAGE Java
name 'thjvTriggers.beforeDelete (oracle.jdbc.NUMBER, oracle.jdbc.CHAR)';
CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:OLD.Id, :OLD.Ename)
/
```

対応するJavaファイルはthjvTriggers.javaです。

```
import java.sql.*
import java.io.*
import oracle.jdbc.*
import oracle.oracore.*
public class thjvTriggers
{
public static void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
   {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stmt = conn.CreateStatement();
   String sql = "insert into logtab values
   ("+ old_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
   stmt.executeUpdate (sql);
   stmt.close();
   return;
   }
}
```

9.7 トリガーのコンパイル、無効化および再コンパイル

CREATE TRIGGER文は、トリガーをコンパイルし、そのコードをデータベースに格納します。コンパイル・エラーが発生してもトリガーは作成されますが、トリガーを起動する文は、次の場合を除いて失敗します。

- トリガーが無効な状態で作成された場合。
- トリガーを起動するイベントがAFTER STARTUP ON DATABASEである場合。
- トリガーを起動するイベントがAFTER LOGON ON DATABASEまたはAFTER LOGON ON SCHEMAのいずれかで、ユーザーがSYSTEMとしてログオンしている場合。

トリガーのコンパイル・エラーを表示するには、SQL*PlusまたはEnterprise ManagerでSHOW ERRORSコマンドを使用するか、

または静的データ・ディクショナリ・ビュー*_ERRORSを問い合せます(『Oracle Databaseリファレンス』を参照)。

トリガーのコンパイルに失敗すると、その例外ハンドラは実行できません。例は、「リモート例外処理」を参照してください。

トリガーがサブプログラムやパッケージなどの別のオブジェクトを参照している場合、そのオブジェクトが変更または削除されると、トリガーは無効になります。次回トリガーを起動するイベントが発生すると、コンパイラはトリガーの再有効化を試みます(詳細は、『Oracle Database開発ガイド』を参照してください)。

> ノート:
>
> メッセージのエンキューには DBMS_AQ パッケージが使用されるため、トリガーとキューの間の依存性は維持できません。

トリガーを手動で再コンパイルするには、ALTER TRIGGER文を使用します(「ALTER TRIGGER文」を参照)。

9.8 トリガーでの例外処理

ほとんどの場合、例外を呼び出す文がトリガーによって実行され、その例外が例外ハンドラによって処理されないと、データベースは、トリガーとそのトリガーを呼び出す文の両方の影響をロールバックします。

次の場合、データベースは、トリガーを起動する文の影響ではなく、トリガーの影響のみをロールバックします(また、エラーをトレース・ファイルおよびアラート・ログに記録します)。

- トリガーを起動するイベントがAFTER STARTUP ON DATABASEまたはBEFORE SHUTDOWN ON DATABASEである場合。
- トリガーを起動するイベントがAFTER LOGON ON DATABASEで、ユーザーがADMINISTER DATABASE TRIGGER権限を持っている場合。
- トリガーを起動するイベントがAFTER LOGON ON SCHEMAで、ユーザーがスキーマを所有しているか、ALTER ANY TRIGGER権限を持っている場合。

複合DMLトリガーの場合、データベースは、トリガーの影響ではなくトリガーを起動する文の影響のみをロールバックします。ただし、トリガーで宣言された変数は再初期化され、トリガーを起動する文のロールバック前に計算された値は、失われます。

> ノート:
>
> 複雑なセキュリティ認可または制約を実行するトリガーは、通常、ユーザー定義の例外を呼び出します(「ユーザー定義の例外」を参照)。

関連項目:

例外処理の詳細は、「PL/SQLのエラー処理」を参照してください

リモート例外処理

リモート・データベースにアクセスするトリガーは、リモート・データベースが使用できる場合にのみ、リモート例外処理を実行できま

す。ローカル・データベースでトリガーをコンパイルする必要がある場合にリモート・データベースを使用できないと、ローカル・データベースでリモート・データベースにアクセスする文の妥当性チェックを実行できないため、コンパイルは正常に実行されません。トリガーをコンパイルできないと、その例外ハンドラは実行できません。

例9-24のトリガーには、リモート・データベースにアクセスするINSERT文が含まれます。このトリガーには、例外ハンドラも含まれます。ただし、ローカル・データベースでトリガーをコンパイルしようとした場合にリモート・データベースを使用できないと、コンパイルは正常に実行されず、例外ハンドラは実行できません。

例9-25では、例9-24の問題の回避策を示しています(つまり、リモートINSERT文および例外ハンドラをストアド・サブプログラムに配置し、トリガーでストアド・サブプログラムを起動します)。サブプログラムは、リモート・データベースにアクセスするための妥当性チェック済の文が含まれている、コンパイル済の形式でローカル・データベースに格納されます。したがって、リモート・データベースを使用できないため、リモートINSERT文が失敗すると、サブプログラムの例外ハンドラでその例外を処理できます。

例9-24 リモート・データベースが使用できない場合に例外を処理できないトリガー

```
CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  -- When remote database is unavailable, compilation fails here:
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
      VALUES (99, SYSDATE, NULL, 'Could not insert');
    RAISE;
END;
/
```

例9-25 例9-24の回避策

```
CREATE OR REPLACE PROCEDURE insert_row_proc AUTHID CURRENT_USER AS
  no_remote_db EXCEPTION;  -- declare exception
  PRAGMA EXCEPTION_INIT (no_remote_db, -20000);
                           -- assign error code to exception
BEGIN
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
      VALUES (99, SYSDATE, NULL, 'Could not insert row.');

  RAISE_APPLICATION_ERROR (-20000, 'Remote database is unavailable.');
END;
```

```
/

CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  insert_row_proc;
END;
/
```

9.9 トリガー設計のガイドライン

- トリガーを使用して、特定のイベントが発生するたびに(どのユーザーまたはアプリケーションがトリガーを起動する文を発行するかにかかわらず)必要なアクションが実行されることを保証します。

  たとえば、トリガーを使用して、いずれかのユーザーが表を更新するたびにそのログ・ファイルが更新されることを保証します。

- データベース機能が重複するトリガーは作成しないでください。

  たとえば、制約で同じ処理ができるときに、無効なデータを拒否するためのトリガーは作成しないでください(「トリガーと制約の違い」を参照)。

- SQL文が行を処理する順序(変化する可能性あり)に依存するトリガーは作成しないでください。

  たとえば、グローバル・パッケージ変数の現行の値が、行トリガーによって処理される行に依存する場合は、行トリガー内のその変数に値を代入しないでください。グローバル・パッケージ変数がトリガーによって更新される場合は、それらの変数をBEFORE文トリガー内で初期化してください。

- 行データがディスクに書き込まれる前に行を変更するために、BEFORE行トリガーを使用します。
- 行IDを取得して、それを操作で使用するために、AFTER行トリガーを使用します。

  トリガーを起動する文によってORA-02292エラーが発生すると、AFTER行トリガーが起動されます。

> ノート:
>
> AFTER 行トリガーを使用すると、BEFORE 行トリガーより少し効率が上がります。BEFORE 行トリガーでは、影響を受けるデータ・ブロックを最初にトリガーのために読み取り、次にトリガーを起動する文のために読み取ります。AFTER 行トリガーでは、影響を受けるデータ・ブロックをトリガーのためにのみ読み取ります。

- BEFORE文トリガーのトリガーを起動する文が、実行中のUPDATE文と競合するUPDATE文またはDELETE文である場合、データベースによって、SAVEPOINTまでの透過的ROLLBACKが実行され、トリガーを起動する文が再開されます。データベースでは、トリガーを起動する文が正常に完了するまでこの処理が何回も行われる可能性があります。データベースによってトリガーを起動する文が再開されるたびに、トリガーが起動されます。SAVEPOINTまでのROLLBACKでは、トリガーが参照するパッケージ変数への変更は取り消されません。この状況を検出するには、パッケージにカウンタ変数を含めてください。
- 再帰トリガーは作成しないでください。

  たとえば、トリガーが定義されている表に対してUPDATE文を発行するAFTER UPDATEトリガーは、作成しないでください。このトリガーは、メモリー不足になるまで再帰的に起動し続けます。

- リモート・データベースにアクセスする文が含まれるトリガーを作成する場合、その文の例外ハンドラをストアド・サブプログラムに配置し、そのサブプログラムをトリガーから起動してください。

  詳細は、「リモート例外処理」を参照してください。

- DATABASEトリガーは慎重に使用してください。このトリガーは、データベース・ユーザーがトリガー・イベントを開始するたびに起動されます。

- トリガーが次の文を実行すると、文によってトリガーの所有者は戻されますが、表を更新しているユーザーは戻されません。

```
SELECT Username FROM USER_USERS;
```

- コミットされたトリガーのみが起動されます。

  トリガーは、それを作成するCREATE TRIGGER文が成功した後に、暗黙的にコミットされます。したがって、トリガーを作成する次の文では、そのトリガーを起動できません。

```
CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
BEGIN
  NULL;
END;
/
```

- 同じ表に対するトリガーを持つアプリケーションのインストールをモジュール化するには、一連の操作を実行する単一のトリガーではなく、同じ型の複数のトリガーを作成します。

  各トリガーは、前に起動されたトリガーによって変更された内容を参照します。各トリガーは、OLDおよびNEWの値を参照できます。

9.10 トリガーの制限

すべてのPL/SQLユニットに適用される制限に加えて(表C-1を参照)、トリガーには次の制限があります。

- トリガーのサイズ制限
- トリガーのLONGおよびLONG RAWデータ型の制約
- 変更表の制限
- 自律型トリガーのみがTCL文またはDDL文を実行できます。

  自立型トリガーの詳細は、「自律型トリガー」を参照してください。

- サブプログラムはトリガー本体のコンテキスト内で実行されるため、トリガーでは、トランザクション制御文を実行するサブプログラムを起動できません。

  トリガーによって起動されるサブプログラムの詳細は、「トリガーによって起動されるサブプログラム」を参照してください。

- トリガーでは、SERIALLY_REUSABLEパッケージにアクセスできません。

  SERIALLY_REUSABLEパッケージの詳細は、「SERIALLY_REUSABLEパッケージ」を参照してください。

関連項目:

「複合DMLトリガーの制限」

9.10.1 トリガーのサイズ制限

トリガーのサイズは、32K以下にしてください。

トリガーのロジックで60行をはるかに超えるPL/SQLソース・テキストが必要な場合は、そのソース・テキストの大部分をストアド・サブプログラムに組み込んで、トリガーからそのサブプログラムを起動してください。トリガーによって起動されるサブプログラムの詳細は、「トリガーによって起動されるサブプログラム」を参照してください。

9.10.2 トリガーのLONGおよびLONG RAWデータ型の制約

ノート:

Oracle は、既存アプリケーションとの下位互換性のためにのみ、LONG および LONG RAW データ型をサポートしています。

すべてのPL/SQLユニットに適用される制限に加えて(「LONGおよびLONG RAW変数」を参照)、トリガーには次の制限があります。

- トリガーでは、LONGまたはLONG RAWデータ型の変数を宣言できません。
- トリガー内のSQL文では、列データをCHARまたはVARCHAR2データ型に変換できる場合にのみ、LONGまたはLONG RAW列を参照できます。
- トリガーでは、LONGまたはLONG RAW列で相関名のNEWまたはPARENTを使用できません。

9.10.3 変更表の制限

ノート:

このトピックの内容は、行レベルの単純な DML トリガーにのみ適用されます。

変更表とは、DML文で(状況によってはDELETE CASCADE制約の影響によって)現在変更されている表です。(INSTEAD OFトリガーによって変更中のビューは、変更ビューとはみなされません。)

変更表の制限のため、トリガーは、トリガーを起動する文が変更中の表を問い合せたり、変更することはできません。行レベルのトリガーで変更表が検出されると、ORA-04091が発生し、トリガーおよびトリガーを起動する文の影響がロールバックされ、トリガーを起動する文を発行したユーザーまたはアプリケーションに制御が戻されます(例9-26を参照)。

注意:

Oracle Database では、分散データベースの異なるノード上に存在する表の間の宣言参照制約はサポートされないため、リモート・ノードにアクセスするトリガーに関する変更表の制限は適用されません。

同様に、データベースでは、ループバック・データベース・リンクで接続されている同一データベース内の表に関する変更表の制限も適用されません。ループバック・データベース・リンクによって、リンクを含むデータベースに戻る Oracle Net パスが定義され、ローカル表がリモートで表示されます。

トリガーを使用して変更表を更新する必要がある場合、次のいずれかの方法で変更表エラーを回避できます。

- 複合DMLトリガーを使用します(「複合DMLトリガーを使用した変更表エラーの回避」を参照)。
- 一時表を使用します。

  たとえば、変更表を更新する1つのAFTER行単位トリガーを使用するかわりに、一時表を更新するAFTER行単位トリガーおよび一時表からの値を使用して変更表を更新するAFTER文トリガーの2つのトリガーを使用します。

変更表の制限の緩和

Oracle Database 8gリリース1以上では、親表に対して削除を行うと、BEFOREおよびAFTERトリガーが1回起動されます。したがって、親表および子表の問合せおよび変更を実行する行レベルと文レベルのトリガーを作成できます。制約が自己参照的でない場合、これによって、ほとんどの外部キー制約アクションはそれらのAFTER行トリガーを介して実装されます。更新カスケード、更新セットNULL、更新セット・デフォルト、削除セット・デフォルト、欠落した親の挿入および子のカウントの保持をすべて簡単に実装できます(「参照整合性を保証するトリガー」を参照)。

ただし、カスケードでは、複数行の外部キーを更新する場合に注意が必要です。外部キー制約によって、AFTER行トリガーが起動されるまでいずれの一致する外部キー行もロックされないことが保証されているため、トリガーは、別のトランザクションによってコミットされていない変更済の行を見逃すことはありません。

例9-27では、トリガーを起動する文によってpは正常に更新されますが、トリガーがfを更新するときに問題が発生します。まず、トリガーを起動する文はpの値(1)から(2)への変更を行い、トリガーはfに値(2)の行を2つ残したままfの値(1)から(2)への更新を行います。次に、トリガーを起動する文はpの値(2)から(3)への更新を行い、トリガーはfの2つの行の値を両方とも(2)から(3)へ更新します。最後に、この文はpの値(3)から(4)への更新を行い、トリガーはfの3つすべての行を(3)から(4)へ更新します。pとfのデータ項目間の関係は失われます。

この問題を回避するには、主キーを変更するpの複数行更新を禁止して既存の主キー値を再利用するか、または外部キー値の更新を追跡して行が2回更新されないようにトリガーを変更します。

例9-26 トリガーによる変更表エラー

```
-- Create log table

DROP TABLE log;
CREATE TABLE log (
  emp_id   NUMBER(6),
  l_name   VARCHAR2(25),
  f_name   VARCHAR2(20)
);

-- Create trigger that updates log and then reads employees

CREATE OR REPLACE TRIGGER log_deletions
  AFTER DELETE ON employees
  FOR EACH ROW
DECLARE
```

```
  n INTEGER;
BEGIN
  INSERT INTO log VALUES (
    :OLD.employee_id,
    :OLD.last_name,
    :OLD.first_name
  );

  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/

-- Issue triggering statement:

DELETE FROM employees WHERE employee_id = 197;
```

結果:

```
DELETE FROM employees WHERE employee_id = 197
            *
ERROR at line 1:
ORA-04091: table HR.EMPLOYEES is mutating, trigger/function might not see it
ORA-06512: at "HR.LOG_DELETIONS", line 10
ORA-04088: error during execution of trigger 'HR.LOG_DELETIONS'
```

トリガーの影響がロールバックされていることの確認:

```
SELECT count(*) FROM log;
```

結果:

```
  COUNT(*)
----------
         0

1 row selected.
```

トリガーを起動する文の影響がロールバックされていることの確認:

```
SELECT employee_id, last_name FROM employees WHERE employee_id = 197;
```

結果:

```
EMPLOYEE_ID LAST_NAME
----------- -------------------------
        197 Feeney

1 row selected.
```

例9-27 更新カスケード

```
DROP TABLE p;
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);
INSERT INTO p VALUES (1);
INSERT INTO p VALUES (2);
INSERT INTO p VALUES (3);
```

```
DROP TABLE f;
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);
INSERT INTO f VALUES (1);
INSERT INTO f VALUES (2);
INSERT INTO f VALUES (3);

CREATE TRIGGER pt
  AFTER UPDATE ON p
  FOR EACH ROW
BEGIN
  UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/
```

問合せ:

```
SELECT * FROM p ORDER BY p1;
```

結果:

```
        P1
----------
         1
         2
         3
```

問合せ:

```
SELECT * FROM f ORDER BY f1;
```

結果:

```
        F1
----------
         1
         2
         3
```

トリガーを起動する文の発行:

```
UPDATE p SET p1 = p1+1;
```

問合せ:

```
SELECT * FROM p ORDER BY p1;
```

結果:

```
        P1
----------
         2
```

```
         3
         4
```

問合せ:

```
SELECT * FROM f ORDER BY f1;
```

結果:

```
        F1
----------
         4
         4
         4
```

9.11 トリガーの起動順序

タイミングの異なる2つ以上のトリガーを同じ表の同じ文に定義する場合、それらのトリガーは次の順序で起動されます。

1. すべてのBEFORE STATEMENTトリガー
2. すべてのBEFORE EACH ROWトリガー
3. すべてのAFTER EACH ROWトリガー
4. すべてのAFTER STATEMENTトリガー

場合によっては、異なるタイミングを持つ個々のトリガー群を、必要な順序でアクションを明示的にコード化した単一の複合トリガーに置き換えます。複合トリガーの詳細は、「複合DMLトリガー」を参照してください。

タイミングが同じ2つ以上のトリガーを作成する際に、起動順序が重要である場合は、FOLLOWS句またはPRECEDES句(「FOLLOWS | PRECEDES」を参照)を使用してその起動順序を制御できます。

1つの表に複数の複合トリガーを作成する場合、次のことに注意してください。

- すべてのBEFORE STATEMENTセクションはBEFORE STATEMENTのタイミングで、BEFORE EACH ROWセクションはBEFORE EACH ROWのタイミングで、というように実行されます。

  トリガーの実行順序がFOLLOWS句で指定された場合、FOLLOWS句によって複合トリガーのセクションの実行順序が決定されます。FOLLOWSがすべてのトリガーではなく一部のトリガーに指定されている場合、トリガーの実行順序はFOLLOWS句によって関係付けられたトリガーについてのみ保証されます。

- すべてのAFTER STATEMENTセクションはAFTER STATEMENTのタイミングで、AFTER EACH ROWセクションはAFTER EACH ROWのタイミングで、というように実行されます。

  トリガーの実行順序がPRECEDES句で指定された場合、PRECEDES句によって複合トリガーのセクションの実行順序が決定されます。PRECEDESがすべてのトリガーではなく一部のトリガーに指定されている場合、トリガーの実行順序はPRECEDES句によって関係付けられたトリガーについてのみ保証されます。

ノート:

PRECEDES は、reverse crossedition トリガーにのみ適用されます。詳細は、『Oracle Database

開発ガイド』を参照してください。

複合トリガーの起動は、単純なトリガーの起動でインターリーブされる可能性があります。

あるトリガーによって他のトリガーが起動される場合、それらのトリガーはカスケードしていると言われます。データベースでは、一度に最大32個のトリガーをカスケードできます。カスケードするトリガーの数を制限する場合、トリガーを起動するたびにカーソルがオープンされるため、OPEN_CURSORS初期化パラメータ(『Oracle Databaseリファレンス』を参照)を使用します。

9.12 トリガーの有効化および無効化

デフォルトでは、トリガーは、CREATE TRIGGER文によって有効な状態で作成されます。無効な状態でトリガーを作成するには、DISABLEを指定します。無効な状態でトリガーを作成すると、有効化する前にトリガーがエラーなしでコンパイルされることを確認できます。

次の理由がある場合、トリガーを一時的に無効にできます。

- トリガーが、使用できないオブジェクトを参照している場合。
- 大規模なデータ・ロードを実行する必要があり、トリガーを起動せずに迅速に処理する場合。
- データをリロードする場合。

単一のトリガーを有効化または無効化するには、次の文を使用します。

```
ALTER TRIGGER [schema.]trigger_name { ENABLE | DISABLE };
```

特定の表に対して作成されたすべてのエディションのすべてのトリガーを有効化または無効化するには、次の文を使用します。

```
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
```

前述の2つの文で、schemaは、トリガーを含むスキーマの名前で、デフォルトは自分のスキーマです。

関連項目:

- ALTER TRIGGER文の詳細は、「ALTER TRIGGER文」を参照してください
- ALTER TABLE文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

9.13 トリガーの変更およびデバッグ

トリガーを変更するには、そのトリガーを置換または再作成する必要があります。(ALTER TRIGGER文は、トリガーの有効化、無効化、コンパイルまたは名前変更のみを実行します。)

トリガーを置換するには、OR REPLACE句を指定してCREATE TRIGGER文を使用します。

トリガーを再作成するには、最初にトリガーをDROP TRIGGER文を使用して削除してから、CREATE TRIGGER文を使用して再作成します。

トリガーをデバッグする場合、ストアド・サブプログラムで使用可能な機能を使用できます。これらの機能の詳細は、『Oracle

Database開発ガイド』を参照してください。

関連項目:

- CREATE TRIGGER文の詳細は、「CREATE TRIGGER文」を参照してください
- DROP TRIGGER文の詳細は、「DROP TRIGGER文」を参照してください
- ALTER TRIGGER文の詳細は、「ALTER TRIGGER文」を参照してください

9.14 トリガーおよびOracle Databaseデータ転送ユーティリティ

データベースにデータを転送する(トリガーを起動する可能性のある)Oracle Databaseユーティリティには、次のものがあります。

- SQL*Loader (sqlldr)

  SQL*Loaderは、外部ファイルからOracle Databaseの表にデータをロードします。

  SQL*Loaderによる従来型ロードの実行中に、INSERTトリガーが起動されます。

  SQL*Loaderによるダイレクト・ロードの前に、トリガーは無効になります。

  関連項目:

  SQL*Loaderの詳細は、『Oracle Databaseユーティリティ』を参照してください。

- データ・ポンプ・インポート(impdp)

  データ・ポンプ・インポート(impdp)は、データ・ポンプ・エクスポート(expdp)によって作成されたエクスポート・ダンプ・ファイルのセットを読み取り、その内容をOracle Databaseに書き込みます。

  インポート対象の表がターゲット・データベースに存在しない場合、またはTABLE_EXISTS_ACTION=REPLACEを指定する場合、impdpによってトリガーの作成前に表の作成およびロードが行われるため、トリガーは起動されません。

  インポート対象の表がターゲット・データベースに存在し、TABLE_EXISTS_ACTION=APPENDまたはTABLE_EXISTS_ACTION=TRUNCATEを指定する場合、impdpによって既存の表に行がロードされ、その表に対して作成されているINSERTトリガーが起動されます。

  関連項目:

  データ・ポンプ・インポートの詳細は、『Oracle Databaseユーティリティ』を参照してください。

- オリジナル・インポート(imp)

  オリジナル・インポート(オリジナル・インポート・ユーティリティimp)は、オリジナル・エクスポート(オリジナル・エクスポート・ユーティリティexp)によって作成されたダンプ・ファイルからオブジェクト定義および表データを読み取ります。

  ノート:

  オリジナル・エクスポートによって作成されたファイルをインポートするには、オリジナル・インポートを使用する

必要があります。それ以外のすべての場合、オリジナル・インポートのかわりにデータ・ポンプ・インポートを使用することをお薦めします。

インポート対象の表がターゲット・データベースに存在しない場合、impによってトリガーの作成前に表の作成およびロードが行われるため、トリガーは起動されません。

インポート対象の表がターゲット・データベースに存在する場合、インポートのIGNOREパラメータによってインポート操作中にトリガーが起動されるかどうかが決定されます。IGNOREパラメータでは、オブジェクト作成エラーを無視するかどうかを指定し、その結果、次のように動作します。

- IGNORE=n(デフォルト)の場合、impでは表が変更されず、トリガーは起動されません。
- IGNORE=yの場合、impによって既存の表に行がロードされ、その表に対して作成されているINSERTトリガーが起動されます。

関連項目:

- オリジナル・インポート・ユーティリティの詳細は、『Oracle Databaseユーティリティ』を参照してください。
- オリジナル・エクスポート・ユーティリティの詳細は、『Oracle Databaseユーティリティ』を参照してください。
- IGNOREの詳細は、『Oracle Databaseユーティリティ』を参照してください。

9.15 イベントをパブリッシュするトリガー

トリガーを使用してイベントをパブリッシュするには、次のようなトリガーを作成します。

- トリガーに、トリガーを起動するイベントとしてそのイベントを含めること。
- トリガーで、DBMS_AQパッケージの適切なサブプログラムを起動すること。このパッケージには、Oracleアドバンスト・キューイング(AQ)に対するインタフェースが含まれます。

  DBMS_AQパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

  AQの詳細は、『Oracle Databaseアドバンスト・キューイング・ユーザーズ・ガイド』を参照してください。

このようなトリガーを有効化および無効化することで、イベント通知のオンとオフを切り替えることができます。トリガーの有効化および無効化の詳細は、「トリガーの有効化および無効化」を参照してください。

トリガーによるイベントのパブリッシュ方法

データベースは、イベントを検出すると、そのイベントに対して定義されている有効なすべてのトリガーを起動します。これには次の例外があります。

- トリガーがトリガーを起動するイベントのターゲットである場合、このトリガーは起動されません。

  たとえば、すべてのDROPイベントのトリガーは、トリガー自体が削除される場合は起動されません。
- 変更されたトリガーが、トリガーを起動するイベントと同じトランザクション内でコミットされなかった場合、このトリガーは起動されません。

  たとえば、システム・トリガー内の再帰DDL文によって別のトリガーが変更されると、同じトランザクション内のイベントで

は、変更されたトリガーを起動できなくなります。

トリガーによってAQが起動されると、AQはイベントをパブリッシュして、トリガーにパブリケーション・コンテキストおよび指定された属性を渡します。トリガーでは、イベント属性ファンクションを起動することでこれらの属性にアクセスできます。

トリガーが(INパラメータとしてAQに渡すことで)AQに対して指定し、その後イベント属性ファンクションを使用してアクセスできる属性は、トリガーを起動するイベント(データベース・イベントまたはクライアント・イベント)によって異なります。

ノート:

- トリガーは、常に定義者権限(DR)ユニットのように動作します。イベントのトリガー・アクションは、アクションの定義者として(コールアウト内のパッケージまたはファンクションの定義者、またはキュー内のトリガーの所有者として)実行されます。トリガーの所有者には基礎となるキュー、パッケージまたはサブプログラムに対するEXECUTE 権限が必要なため、このアクションには一貫性があります。DR ユニットの詳細は、「実行者権限および定義者権限(AUTHID プロパティ)」を参照してください。
- データベースでは、すべてのイベントについてコールバック・ファンクションからの戻りステータスが無視されます。たとえば、データベースでは、SHUTDOWN イベントからの戻りステータスに対して何も実行できません。

ここでのトピック

- イベント属性ファンクション
- データベース・イベント・トリガーのイベント属性ファンクション
- クライアント・イベント・トリガーのイベント属性ファンクション

9.15.1 イベント属性ファンクション

トリガーでは、表9-5のシステム定義のイベント属性ファンクションを起動することで、トリガーを起動するイベントの特定の属性を取り出すことができます。すべてのトリガーがすべてのイベント属性ファンクションを起動できるわけではありません(詳細は、「データベース・イベント・トリガーのイベント属性ファンクション」および「クライアント・イベント・トリガーのイベント属性ファンクション」を参照してください)。

ノート:

- 以前のリリースでは、これらのファンクションには SYS パッケージを介してアクセスする必要がありました。現在、各ファンクションにアクセスするには、そのパブリック・シノニム(表 9-5 の最初の列の ora_で始まる名前)を使用することをお薦めします。
- ファンクション・パラメータ ora_name_list_t は、DBMS_STANDARD パッケージ内で次のように定義されます。

```
TYPE ora_name_list_t IS TABLE OF VARCHAR2(2*(ORA_MAX_NAME_LEN+2)+1);
```

表9-5 システム定義のイベント属性

属性	戻り型および値	例
ora_client_ip_address	VARCHAR2: 基礎となるプロトコルが TCP/IP の場合、LOGON イベントでのクライアントの IP アドレス	DECLARE v_addr VARCHAR2(11); BEGIN IF (ora_sysevent = 'LOGON') THEN v_addr := ora_client_ip_address; END IF; END; /
ora_database_name	VARCHAR2(50): データベース名	DECLARE v_db_name VARCHAR2(50); BEGIN v_db_name := ora_database_name; END; /
ora_des_encrypted_password	VARCHAR2: 作成または変更されるユーザーの DES 暗号化パスワード	IF (ora_dict_obj_type = 'USER') THEN INSERT INTO event_table VALUES (ora_des_encrypted_password); END IF;
ora_dict_obj_name	VARCHAR2(128): DDL 操作が発生したディクショナリ・オブジェクトの名前	INSERT INTO event_table VALUES ('Changed object is ' || ora_dict_obj_name);
ora_dict_obj_name_list (name_list OUT ora_name_list_t)	PLS_INTEGER: イベントで変更されたオブジェクト名の数 OUT パラメータ: イベントで変更されたオブジェクト名のリスト	DECLARE name_list ora_name_list_t; number_modified PLS_INTEGER; BEGIN IF (ora_sysevent='ASSOCIATE STATISTICS') THEN number_modified := ora_dict_obj_name_list(name_list); END IF; END;
ora_dict_obj_owner	VARCHAR2(128): DDL 操作が発生したディクショナリ・オブジェクトの所有者	INSERT INTO event_table VALUES ('object owner is' || ora_dict_obj_owner);
ora_dict_obj_owner_list (owner_list OUT ora_name_list_t)	PLS_INTEGER: イベントで変更されたオブジェクトの所有者の数 OUT パラメータ: イベントで変更されたオブジェクトの所有者のリスト	DECLARE owner_list ora_name_list_t; number_modified PLS_INTEGER; BEGIN IF (ora_sysevent='ASSOCIATE STATISTICS') THEN number_modified := ora_dict_obj_name_list(owner_list); END IF; END;
ora_dict_obj_type	VARCHAR2(20): DDL 操作が発生したディクショ	INSERT INTO event_table VALUES ('This object is a ' ||

属性	戻り型および値	例
	ナリ・オブジェクトの型	ora_dict_obj_type);
ora_grantee (user_list OUT ora_name_list_t)	PLS_INTEGER: 権限付与イベントでの権限受領者の数 OUT パラメータ: 権限付与イベントでの権限受領者のリスト	DECLARE user_list ora_name_list_t; number_of_grantees PLS_INTEGER; BEGIN IF (ora_sysevent = 'GRANT') THEN number_of_grantees := ora_grantee(user_list); END IF; END;
ora_instance_num	NUMBER: インスタンス番号	IF (ora_instance_num = 1) THEN INSERT INTO event_table VALUES ('1'); END IF;
ora_is_alter_column (column_name IN VARCHAR2)	BOOLEAN: 指定した列が変更されている場合は TRUE、それ以外の場合は FALSE	IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN alter_column := ora_is_alter_column('C'); END IF;
ora_is_creating_nested_table	BOOLEAN: 現行のイベントがネストした表を作成している場合は TRUE、それ以外の場合は FALSE	IF (ora_sysevent = 'CREATE' AND ora_dict_obj_type = 'TABLE' AND ora_is_creating_nested_table) THEN INSERT INTO event_table VALUES ('A nested table is created'); END IF;
ora_is_drop_column (column_name IN VARCHAR2)	BOOLEAN: 指定した列が削除されている場合は TRUE、それ以外の場合は FALSE	IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN drop_column := ora_is_drop_column('C'); END IF;
ora_is_servererror (error_number IN VARCHAR2)	BOOLEAN: 指定したエラーがエラー・スタック上にある場合は TRUE、それ以外の場合は FALSE	IF ora_is_servererror(error_number) THEN INSERT INTO event_table VALUES ('Server error!!'); END IF;
ora_login_user	VARCHAR2(128): ログイン・ユーザー名	SELECT ora_login_user FROM DUAL;
ora_partition_pos	PLS_INTEGER: CREATE TABLE の INSTEAD OF トリガーでは、PARTITION 句を挿入できる SQL テ	-- Retrieve ora_sql_txt into sql_text variable v_n := ora_partition_pos; v_new_stmt := SUBSTR(sql_text,1,v_n - 1) || ' ' || my_partition_clause || ' ' ||

属性	戻り型および値	例
	キスト内の位置	SUBSTR(sql_text, v_n));
ora_privilege_list (privilege_list OUT ora_name_list_t)	PLS_INTEGER: 権限付与イベントまたは取消しイベントでの権限の数 OUT パラメータ: イベントで付与または取り消された権限のリスト	DECLARE privilege_list ora_name_list_t; number_of_privileges PLS_INTEGER; BEGIN IF (ora_sysevent = 'GRANT' OR ora_sysevent = 'REVOKE') THEN number_of_privileges := ora_privilege_list(privilege_list); END IF; END;
ora_revokee (user_list OUT ora_name_list_t)	PLS_INTEGER: 権限取消しイベントでの取消し対象者の数 OUT パラメータ: イベントでの権限取消し対象者のリスト	DECLARE user_list ora_name_list_t; number_of_users PLS_INTEGER; BEGIN IF (ora_sysevent = 'REVOKE') THEN number_of_users := ora_revokee(user_list); END IF; END;
ora_server_error (position IN PLS_INTEGER)	NUMBER: エラー・スタック上の指定した位置にあるエラー・コード脚注 1	INSERT INTO event_table VALUES ('top stack error ' || ora_server_error(1));
ora_server_error_depth	PLS_INTEGER: エラー・スタック上のエラー・メッセージの数	n := ora_server_error_depth; -- Use n with functions such as ora_server_error
ora_server_error_msg (position IN PLS_INTEGER)	VARCHAR2: エラー・スタック上の指定した位置にあるエラー・メッセージ脚注 1	INSERT INTO event_table VALUES ('top stack error message' || ora_server_error_msg(1));
ora_server_error_num_params (position IN PLS_INTEGER)	PLS_INTEGER: エラー・スタック上の指定した位置にある(%s などの書式を使用して)エラー・メッセージに置き換えられた文字列の数脚注 1	n := ora_server_error_num_params(1);
ora_server_error_param (position IN PLS_INTEGER, param IN PLS_INTEGER)	VARCHAR2: 指定した位置およびパラメータ番号におけるエラー・メッセージ	-- Second %s in "Expected %s, found %s": param := ora_server_error_param(1,2);

属性	戻り型および値	例
	内の一致する置換値(%s、%d など)[脚注 1]	
ora_sql_txt (sql_text OUT ora_name_list_t)	PLS_INTEGER: PL/SQL 表での要素の数 OUT パラメータ: トリガーを起動する文の SQL テキスト(文が長い場合、複数のコレクション要素に分割)	CREATE TABLE event_table (col VARCHAR2(2030)); DECLARE sql_text ora_name_list_t; n PLS_INTEGER; v_stmt VARCHAR2(2000); BEGIN n := ora_sql_txt(sql_text); FOR i IN 1..n LOOP v_stmt := v_stmt || sql_text(i); END LOOP; INSERT INTO event_table VALUES ('text of triggering statement: ' || v_stmt); END;
ora_sysevent	VARCHAR2(20): 構文に指定されたトリガーを起動するイベントの名前	INSERT INTO event_table VALUES (ora_sysevent);
ora_with_grant_option	BOOLEAN: GRANT オプションとともに権限が付与されている場合は TRUE、それ以外の場合は FALSE	IF (ora_sysevent = 'GRANT' AND ora_with_grant_option = TRUE) THEN INSERT INTO event_table VALUES ('with grant option'); END IF;
ora_space_error_info (error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)	BOOLEAN: エラーが領域不足状態に関連している場合は TRUE、それ以外の場合は FALSE OUT パラメータ: エラーの原因となったオブジェクトに関する情報	IF (ora_space_error_info (eno, typ, owner, ts, obj, subobj) = TRUE) THEN DBMS_OUTPUT.PUT_LINE('The object '|| obj || ' owned by ' || owner || ' has run out of space.'); END IF;

脚注1

位置1は、スタックの先頭です。

9.15.2 データベース・イベント・トリガーのイベント属性ファンクション

表9-6に、イベント属性ファンクションを起動できるデータベース・イベント・トリガーの概要を示します。表9-6のトリガーを起動するイベントの詳細は、「database_event」を参照してください。

表9-6 データベース・イベント・トリガー

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
AFTER STARTUP	データベースのオープン時。	なし	トリガーではデータベース操作は実行できません。	トリガーの起動後、別のトランザクションを開始してコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE SHUTDOWN	サーバーでインスタンスの停止が開始される直前。 これによって、カートリッジを完全に停止できます。インスタンスの異常停止の場合、このトリガーは起動されない場合があります。	なし	トリガーではデータベース操作は実行できません。	トリガーの起動後、別のトランザクションを開始してコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER DB_ROLE_CHANGE	ロールの変更後最初のデータベースのオープン時。	なし	なし	トリガーの起動後、別のトランザクションを開始してコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER SERVERERROR	条件がある場合、指定したエラーが発生するたびに起動。条件がない場合、任意のエラーが発生するたびに起動。 トリガーは、「database_event」にリストされているエラーに対しては起動されません。	ERRNO = eno	エラーによって異なります。	トリガーの起動後、別のトランザクションを開始してコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror ora_space_error_info

9.15.3 クライアント・イベント・トリガーのイベント属性ファンクション

表9-7に、イベント属性ファンクションを起動できるクライアント・イベント・トリガーの概要を示します。表9-7のトリガーを起動するイベントの詳細は、「ddl_event」および「database_event」を参照してください。

ノート:

クライアント・イベント・トリガーが DDL 操作(CREATE OR REPLACE TRIGGER など)のターゲットになった場合、同じトランザクション中に後でこのトリガーを起動することはできません。

表9-7 クライアント・イベント・トリガー

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
BEFORE ALTER AFTER ALTER	カタログ・オブジェクトの変更時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column (for ALTER TABLE events) ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	カタログ・オブジェクトの削除時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
BEFORE ANALYZE AFTER ANALYZE	ANALYZE 文の発行時。	オブジェクトの型と名前、UID および USER に関する単	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
		純な条件	行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。		ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	ASSOCIATE STATISTICS 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	AUDIT または NOAUDIT 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
BEFORE COMMENT AFTER COMMENT	オブジェクトのコメント化時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	カタログ・オブジェクトの作成時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)
BEFORE DDL AFTER DDL	ほとんどの SQL DDL 文の発行時。PL/SQL サブプログラム・インタフェースを介して発行された ALTER DATABASE、CREATE CONTROLFILE、CREATE DATABASE および DDL(アドバンスト・キューの作	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、およ	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
	成など)に対しては起動されません。		び表の作成、変更、削除に制限されます。		
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	DISASSOCIATE STATISTICS 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE GRANT AFTER GRANT	GRANT 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privilege_list
BEFORE LOGOFF	ユーザー・ログオフの開始時。	UID および USER に関する単純な条件	他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
			更、削除に制限されます。		
AFTER LOGON	ユーザーが正常にログオンした後。	UID および USER に関する単純な条件	他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	トリガーの起動後、別のトランザクションを開始してコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	RENAME 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
BEFORE REVOKE AFTER REVOKE	REVOKE 文の発行時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privilege_list

トリガー・イベント	トリガーが起動されるタイミング	WHEN条件	制限事項	トランザクション	属性ファンクション
			されます。		
AFTER SUSPEND	領域不足状態のため SQL 文が一時停止された後。 (トリガーは、文を再開できるようにこの状態を修正する必要があります。)	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror ora_space_error_info
BEFORE TRUNCATE AFTER TRUNCATE	オブジェクトの切捨て時。	オブジェクトの型と名前、UID および USER に関する単純な条件	トリガーは、イベントを生成したオブジェクトに対して DDL 操作を実行できません。 他のオブジェクトに対する DDL は、オブジェクトのコンパイル、トリガーの作成、および表の作成、変更、削除に制限されます。	カレント・トランザクションでトリガーが起動されます。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

9.16 トリガーに関する情報が表示されるビュー

*_TRIGGERS静的データ・ディクショナリ・ビューには、トリガーに関する情報が表示されます。これらのビューの詳細は、『Oracle Databaseリファレンス』を参照してください。

例9-28 トリガーに関する情報の表示

この例では、トリガーを作成し、静的データ・ディクショナリ・ビューUSER_TRIGGERSを2回問い合せています。最初の問合せでは、トリガーのタイプ、トリガーを起動するイベント、およびトリガーが作成されている表の名前を表示し、次の問合せではトリガー本

体を表示します。

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/
```

これらのSQL*Plusコマンドは、問合せ結果の書式を設定します。

```
COLUMN Trigger_type FORMAT A15
COLUMN Triggering_event FORMAT A16
COLUMN Table_name FORMAT A11
COLUMN Trigger_body FORMAT A50
SET LONG 9999
```

問合せ:

```
SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

結果:

```
TRIGGER_TYPE    TRIGGERING_EVENT TABLE_NAME
--------------- ---------------- -----------
AFTER STATEMENT DELETE           EMPLOYEES
```

問合せ:

```
SELECT Trigger_body
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

結果:

```
TRIGGER_BODY
--------------------------------------------------
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || '
employees.');
END;
```

10 PL/SQLパッケージ

この章では、互いに関連するPL/SQLコードとデータを1つにまとめ、複数のアプリケーションで使用できる内容を持つパッケージにする方法について説明します。

ここでのトピック

- パッケージ
- パッケージを使用する理由
- パッケージ仕様部
- パッケージ本体
- パッケージのインスタンス化および初期化
- パッケージの状態
- SERIALLY_REUSABLEパッケージ
- パッケージ作成のガイドライン
- パッケージの例
- STANDARDパッケージによるPL/SQL環境の定義

10.1 パッケージ

パッケージとは、論理的に関連するPL/SQLの型、変数、定数、サブプログラム、カーソルおよび例外をグループにまとめたスキーマ・オブジェクトのことです。パッケージをコンパイルしてデータベースに格納し、その内容を複数のアプリケーションで共有できます。

パッケージには必ず仕様部があり、パッケージの外から参照できるパブリック項目はここで宣言します。

パブリック項目にカーソルまたはサブプログラムが含まれる場合、パッケージには本体も必要です。本体には、パブリック・カーソルの問合せとパブリック・サブプログラムのコードを定義する必要があります。本体では、パッケージの外からは参照できなくてもパッケージの内部動作に必要なプライベート項目を宣言および定義することもできます。本体の最後には初期化部(ここには、変数を初期化する文と、その他の1回のみの設定ステップを実行する文を入れます)、および例外処理部を配置することができます。本体は、仕様部またはパブリック項目への参照を変更せずに変更できるため、パッケージ本体はブラック・ボックスと考えることができます。

パッケージ仕様部またはパッケージ本体のいずれでも、パッケージ・サブプログラムを外部のJavaサブプログラムまたはCサブプログラムにマップできますが、その際にはコール仕様を使用して、外部サブプログラム名、パラメータ型および戻り型を対応するSQLにマップします。

パッケージ仕様部のAUTHID句は、パッケージのサブプログラムとカーソルをその定義者(デフォルト)と実行者のどちらの権限で実行するか、およびスキーマ・オブジェクトへの未修飾の参照が定義者と実行者のどちらのスキーマで解決されるかを決定します。

パッケージ仕様部のACCESSIBLE BY句を使用すると、パッケージにアクセスできるPL/SQLユニットのホワイト・リストを指定できます。この句は次のような状況で使用します。

- PL/SQLアプリケーションを複数のパッケージとして実装します。このうち、1つのパッケージがアプリケーション・プログラミング・インタフェース(API)を提供し、ヘルパー・パッケージが処理を実行します。このとき、クライアントがAPIにアクセスするが、ヘルパー・パッケージにはアクセスしないようにします。したがって、ACCESSIBLE BY句をAPIパッケージの仕様から省略し、ヘルパー・パッケージの仕様ごとに組み込み、ここで、APIパッケージのみがヘルパー・パッケージにアクセスできるように指定します。
- 同じスキーマ内のすべてではなく一部のPL/SQLユニットにサービスを提供するユーティリティ・パッケージを作成します。パッケージの使用対象を目的のユニットに制限するには、パッケージ仕様内のACCESSIBLE BY句にこれらをリストします。

関連項目:

- パッケージ仕様部の詳細は、「パッケージ仕様部」を参照してください
- パッケージ本体の詳細は、「パッケージ本体」を参照してください
- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」
- 「実行者権限および定義者権限(AUTHIDプロパティ)」

10.2 パッケージを使用する理由

パッケージは、信頼性のある再利用可能なコードの開発およびメンテナンスを次の機能で支援します。

- モジュール性

  パッケージを使用すると、論理的に関連した型、変数、定数、サブプログラム、カーソルおよび例外を、名前付きのPL/SQLモジュールにカプセル化できます。個々のパッケージを理解しやすくし、パッケージ間のインタフェースを単純かつ明快で、明確に定義することができます。この方法はアプリケーション開発に役立ちます。

- アプリケーションの設計の容易さ

  アプリケーション設計の最初の段階では、パッケージの仕様部に含まれるインタフェース情報のみが必要です。仕様部は本体がなくてもコーディングし、コンパイルできます。その後、このパッケージを参照するスタンドアロン・サブプログラムをコンパイルできます。アプリケーション作成の最終段階になるまで、パッケージ本体を完全に定義する必要はありません。

- 実装の詳細の隠ぺい

  パッケージを使用すると、パッケージ仕様部のインタフェース情報を共有でき、パッケージ本体の実装の細部を隠ぺいできます。本体の実装の細部を隠ぺいすることには次のメリットがあります。

  - アプリケーション・インタフェースに影響を与えずに実装の細部を変更できます。
  - アプリケーション・ユーザーは、開発者が変更する可能性のある実装の細部に依存するコードを開発できません。

- 機能の追加

  パッケージのパブリック変数およびカーソルは、セッションが存続する間維持されます。このため、同じ環境の中で実行するすべてのサブプログラムで共有できます。これによって、データベースに格納することなくトランザクション間でデータをメンテナンスできます。(パッケージのパブリック変数およびカーソルが、セッションの存続する間維持されない状況は、「パッケージの状態」を参照してください。)

- より高いパフォーマンス

  パッケージ・サブプログラムを初めて起動すると、パッケージ全体がOracle Databaseによってメモリーにロードされます。同じパッケージ内の他のサブプログラムの2度目以降の起動では、ディスクI/Oは必要ありません。

  パッケージ化すると互いに依存することがなくなるため、不要な再コンパイルを避けることができます。たとえば、パッケージ・ファンクションの本体を変更した場合、そのファンクションを起動する他のサブプログラムは、仕様部で宣言されたパラメータと戻り値にのみ依存するため、Oracle Databaseによって再コンパイルされません。

- ロール付与の容易さ

  パッケージ内の各オブジェクトにロールを付与するかわりに、パッケージ自体にロールを付与できます。

> ノート:
>
> パッケージ内からはホスト変数を参照できません。

10.3 パッケージ仕様部

パッケージ仕様部ではパブリック項目を宣言します。パブリック項目の有効範囲は、パッケージのスキーマです。パブリック項目はスキーマ内のあらゆる場所から参照できます。有効範囲内にあっても参照できないパブリック項目を参照するには、パッケージ名で修飾します。(有効範囲、可視性および修飾の詳細は、「識別子の有効範囲と可視性」を参照してください。)

各パブリック項目の宣言には、その項目を使用するうえで必要なすべての情報を含めます。たとえば、パッケージ仕様部でファンクションfactorialを次の方法で宣言するとします。

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

この宣言は、factorialという名前のファンクションがINTEGER型の引数を1つ取り、INTEGER型の値を戻すことを示しており、起動者がfactorialを起動するうえで知っておく必要のある情報です。起動者はfactorialがどのように実装されているか(たとえば、それが反復を利用しているのか、再帰を利用しているのかなど)を知る必要はありません。

> ノート:
>
> パッケージの使用を指定した PL/SQL ユニットに制限するために、パッケージ仕様部に ACCESSIBLE BY 句を含めます。

ここでのトピック

- 適切なパブリック項目
- パッケージ仕様部の作成

10.3.1 適切なパブリック項目

適切なパブリック項目は次のとおりです。

- 複数のサブプログラムで使用される型、変数、定数、サブプログラム、カーソルおよび例外

パッケージ仕様部で定義される型は、ユーザー定義のPL/SQLサブタイプ(「ユーザー定義のPL/SQLサブタイプ」を参照)またはPL/SQLコンポジット型(「PL/SQLのコレクションおよびレコード」を参照)のいずれかです。

ノート:

パッケージ仕様部で定義された PL/SQL コンポジット型は、同一定義のローカル型またはスタンドアロン型と互換性がありません(例 5-33、例 5-34 および例 5-39 を参照)。

- スタンドアロン・サブプログラム・パラメータの連想配列型

  スキーマ・レベルでは結合配列型を宣言できません。そのため、連想配列の変数をパラメータとしてスタンドアロン・サブプログラムに渡すには、その変数の型をパッケージ仕様部で宣言する必要があります。こうすると、起動されるサブプログラム(この型の仮パラメータを宣言する側)と起動元のサブプログラムまたは無名ブロック(この型の変数を宣言する側)の両方でこの型を使用できます。例10-2を参照してください。

- 同じセッション内のサブプログラム起動間で使用可能な状態に保つ必要がある変数
- パブリック変数の読取りおよび書込みを実行するサブプログラム("get"および"set"サブプログラム)

  これらのサブプログラムは、パッケージ・ユーザーにパブリック変数の読取りおよび書込みを直接実行させないために提供します。

- 相互に起動し合うサブプログラム

  互いに起動し合うスタンドアロン・サブプログラムの場合はコンパイルの順序を気にする必要がありますが、パッケージ・サブプログラムの場合はとらわれる必要はありません。

- オーバーロードされたサブプログラム

  オーバーロードされたサブプログラムとは、同じサブプログラムのバリエーションです。つまり、名前は同じですが、仮パラメータが異なります。詳細は、「オーバーロードされたサブプログラム」を参照してください。

ノート:

リモート・パッケージのパブリック変数は、間接的であっても参照できません。たとえば、サブプログラムがパッケージのパブリック変数を参照する場合、データベース・リンクを通じてサブプログラムを起動することはできません。

10.3.2 パッケージ仕様部の作成

パッケージ仕様部を作成するには、「 CREATE PACKAGE文」を使用します。

例10-1および例10-2のパッケージ仕様部ではカーソルまたはサブプログラムを宣言していないため、パッケージtrans_dataおよびaa_pkgに本体は不要です。

例10-1 単純なパッケージ仕様部

この例では、パッケージtrans_dataの仕様部で2つのパブリック型と3つのパブリック変数を宣言しています。

```
CREATE OR REPLACE PACKAGE trans_data AUTHID DEFINER AS
  TYPE TimeRec IS RECORD (
    minutes SMALLINT,
```

```
    hours   SMALLINT);
  TYPE TransRec IS RECORD (
    category VARCHAR2(10),
    account  INT,
    amount   REAL,
    time_of  TimeRec);
  minimum_balance     CONSTANT REAL := 10.00;
  number_processed    INT;
  insufficient_funds  EXCEPTION;
  PRAGMA EXCEPTION_INIT(insufficient_funds, -4097);
END trans_data;
/
```

例10-2 スタンドアロン・サブプログラムへの連想配列の受渡し

この例では、パッケージaa_pkgの仕様部で連想配列型のaa_typeを宣言しています。その後、スタンドアロン・プロシージャprint_aaで型aa_typeの仮パラメータを宣言しています。次に、無名ブロックで型aa_typeの変数を宣言してこれに移入し、この変数を出力するプロシージャprint_aaに変数を渡しています。

```
CREATE OR REPLACE PACKAGE aa_pkg AUTHID DEFINER IS
  TYPE aa_type IS TABLE OF INTEGER INDEX BY VARCHAR2(15);
END;
/
CREATE OR REPLACE PROCEDURE print_aa (
  aa aa_pkg.aa_type
) AUTHID DEFINER IS
  i  VARCHAR2(15);
BEGIN
  i := aa.FIRST;

  WHILE i IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE (aa(i) || '  ' || i);
    i := aa.NEXT(i);
  END LOOP;
END;
/
DECLARE
  aa_var  aa_pkg.aa_type;
BEGIN
  aa_var('zero') := 0;
  aa_var('one') := 1;
  aa_var('two') := 2;
  print_aa(aa_var);
END;
/
```

結果:

```
1  one
2  two
0  zero
```

10.4 パッケージ本体

パッケージ仕様部でカーソルまたはサブプログラムを宣言している場合は、パッケージ本体が必要ですが、それ以外の場合、本

体はオプションです。パッケージ本体とパッケージ仕様部は同じスキーマ内にある必要があります。

パッケージ仕様部内のカーソル宣言またはサブプログラム宣言の1つ1つに対応する定義がパッケージ本体にある必要があります。対応するサブプログラム宣言と定義のヘッダーは、空白以外の一語一語が一致している必要があります。

パッケージ本体を作成するには、「 CREATE PACKAGE BODY文」を使用します。

パッケージ仕様部で宣言し、パッケージ本体で定義したカーソルおよびサブプログラムは、パッケージの外から参照できるパブリック項目です。パッケージ本体では、パッケージの外からは参照できなくてもパッケージの内部動作に必要なプライベート項目を宣言および定義することもできます。

本体の最後には初期化部を配置することができ、ここには、パブリック変数を初期化する文と、その他の1回のみの設定ステップを実行する文を入れます。初期化部は、パッケージが初めて参照されたときにのみ実行されます。初期化部には例外ハンドラを含めることができます。

パッケージ本体は、仕様部またはパブリック項目への参照を変更せずに変更できます。

例10-3 パッケージの仕様部と本体の一致

この例では、対応するサブプログラム宣言と定義のヘッダーが完全一致していません。そのため、employees.hire_date%TYPEがDATEであっても、PL/SQLによって例外が生成されます。

```
CREATE PACKAGE emp_bonus AS
  PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);
END emp_bonus;
/
CREATE PACKAGE BODY emp_bonus AS
  -- DATE does not match employees.hire_date%TYPE
  PROCEDURE calc_bonus (date_hired DATE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

結果:

```
Warning: Package Body created with compilation errors.
```

SQL*Plusでのエラーの表示:

```
SHOW ERRORS
```

結果:

```
Errors for PACKAGE BODY EMP_BONUS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/13     PLS-00323: subprogram or cursor 'CALC_BONUS' is declared in a
         package specification and must be defined in the package body
```

問題の修正:

```
CREATE OR REPLACE PACKAGE BODY emp_bonus AS
  PROCEDURE calc_bonus
    (date_hired employees.hire_date%TYPE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

結果:

```
Package body created.
```

10.5 パッケージのインスタンス化および初期化

セッションがパッケージ項目を参照すると、Oracle Databaseはそのセッションのためにパッケージをインスタンス化します。パッケージを参照する各セッションは、そのパッケージの独自のインスタンス化を所有します。

Oracle Databaseがパッケージをインスタンス化すると、パッケージは初期化されます。初期化では、次の処理のうち適用できるものがすべて実行されます。

- パブリック定数への初期値の代入
- 宣言で初期値が指定されているパブリック変数への初期値の代入
- パッケージ本体の初期化部の実行

10.6 パッケージの状態

パッケージ(の仕様部または本体のいずれか)で宣言している変数、定数およびカーソルの値が、**パッケージの状態**を構成します。

PL/SQLパッケージに1つ以上の変数、定数またはカーソルが宣言されている場合、このパッケージはステートフルですが、それ以外の場合はステートレスです。

パッケージ項目を参照する各セッションは、そのパッケージの独自のインスタンス化を所有します。パッケージがステートフルの場合は、インスタンス化にパッケージの状態が含まれます。

パッケージの状態は、次の状況を除き、セッションが存続する間維持されます。

- パッケージがSERIALLY_REUSABLEの場合。
- パッケージ本体が再コンパイルされている場合。

  インスタンス化されたステートフル・パッケージの本体が再コンパイルされている場合は(「ALTER PACKAGE文」を使用した明示的な再コンパイルであっても、暗黙的な再コンパイルであっても)、このパッケージで次にサブプログラムを起動すると、Oracle Databaseは既存のパッケージ状態を破棄して例外ORA-04068を呼び出します。

  PL/SQLによって例外が呼び出された後にパッケージを参照すると、Oracle Databaseによりパッケージが再インスタンス化され、再初期化されます。したがって、パッケージ状態に対する以前の変更は失われます。

- セッションが所有するインスタンス化されたパッケージのいずれかが、無効化された後に再有効化されている場合。

  セッションが所有するインスタンス化されたパッケージのいずれかが無効化された後に再有効化されている場合、セッションのパッケージ・インスタンス化は(パッケージの状態を含め)すべて失われる可能性があります。

セッションが存続する間(またはそれより長い間)パッケージの状態が一定である場合、パッケージはOracle Databaseにステートレスとして処理されます。これは、パッケージの項目がすべてコンパイル時定数になっているパッケージの場合です。

コンパイル時定数とは、コンパイル時にPL/SQLコンパイラで値を決定できる定数のことです。初期値にリテラルが指定されている定数は、必ずコンパイル時定数になります。初期値にリテラル以外が指定されていても、オプティマイザによりリテラルに縮約される定数も、コンパイル時定数です。リテラル以外の式をPL/SQLオプティマイザでリテラルに縮約できるかどうかは、最適化レベルによって異なります。そのため、ある最適化レベルでコンパイルするとステートレスになるパッケージが、別の最適化レベルでコンパイルするとステートフルになることもあります。

関連項目:

- SERIALLY_REUSABLEパッケージ
- 初期化の詳細は、「パッケージのインスタンス化および初期化」を参照してください
- スキーマ・オブジェクトの無効化および再有効化の詳細は、『Oracle Database開発ガイド』を参照してください
- オプティマイザの詳細は、「PL/SQLオプティマイザ」を参照してください

10.7 SERIALLY_REUSABLEパッケージ

SERIALLY_REUSABLEパッケージを使用すると、スケーラビリティを向上させるためにメモリーをより効率的に管理するアプリケーションを設計できます。

パッケージがSERIALLY_REUSABLEでない場合、そのパッケージ状態は各ユーザーのユーザー・グローバル領域(UGA)に格納されます。したがって、UGAのメモリー量をユーザー数に比例して増加させる必要があり、スケーラビリティが制限されます。パッケージの状態はセッションが存続する間維持でき、UGAメモリーはセッションが終了するまでロックされます。Oracle Officeなどの一部のアプリケーションの標準的なセッションは数日間存続します。

パッケージがSERIALLY_REUSABLEである場合、そのパッケージ状態はシステム・グローバル領域(SGA)内の小規模なプールにある作業領域に格納されます。パッケージの状態は、サーバー・コールが存続する間のみ維持されます。サーバー・コールの後、作業領域はプールに戻されます。以降のサーバー・コールがこのパッケージを参照すると、Oracle Databaseはこのプールからインスタンス化を再利用します。インスタンス化を再利用すると、インスタンス化は再初期化されるため、それまでのサーバー・コールで実行されたパッケージ状態に対する変更は参照できなくなります。(初期化の詳細は、「パッケージのインスタンス化および初期化」を参照してください。)

ノート:

データベース・トリガー、または SQL 文から起動される他の PL/SQL サブプログラムから SERIALLY_REUSABLE

パッケージにアクセスしようとするとエラーが発生します。

ここでのトピック

- SERIALLY_REUSABLEパッケージの作成
- SERIALLY_REUSABLEパッケージの作業単位
- SERIALLY_REUSABLEパッケージの明示カーソル

10.7.1 SERIALLY_REUSABLEパッケージの作成

SERIALLY_REUSABLEパッケージを作成するには、SERIALLY_REUSABLEプラグマをパッケージ仕様部とパッケージ本体(存在する場合)に含めます。

例10-4では、きわめて単純なSERIALLY_REUSABLEパッケージを2つ作成します。1つは仕様部のみで、もう1つは仕様部と本体があります。

関連項目:

SERIALLY_REUSABLEプラグマ

例10-4 SERIALLY_REUSABLEパッケージの作成

```
-- Create bodiless SERIALLY_REUSABLE package:

CREATE OR REPLACE PACKAGE bodiless_pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

-- Create SERIALLY_REUSABLE package with specification and body:

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg IS
  PRAGMA SERIALLY_REUSABLE;
BEGIN
  n := 5;
END;
/
```

10.7.2 SERIALLY_REUSABLEパッケージの作業単位

SERIALLY_REUSABLEパッケージの作業単位はサーバー・コールです。

このパッケージのパブリック変数は、作業単位内のみで使用する必要があります。

ノート:

誤って以前の作業単位で設定されたパブリック変数の値に依存した場合、そのプログラムは失敗する可能性があります。PL/SQL ではこのようなケースはチェックできません。

SERIALLY_REUSABLEパッケージの作業単位(サーバー・コール)が完了した後、Oracle Databaseにより次の処理が実行されます。

- オープンされているすべてのカーソルのクローズ処理。
- 再使用不可のメモリーの一部を解放する処理(たとえば、コレクション変数用のメモリー、長いVARCHAR2用のメモリーなど)。
- このパッケージのインスタンス化を、このパッケージ用に保持された再使用可能インスタンス化のプールに戻す処理。

例10-5 SERIALLY_REUSABLEプラグマの影響

この例では、本体のないパッケージpkgとsr_pkgは、sr_pkgはSERIALLY_REUSABLEでpkgはそうでないという点以外同じです。各パッケージで、初期値に5が指定されたパブリック変数nを宣言します。その後、無名ブロックで各変数の値を10に変更します。次に、別の無名ブロックで各変数の値を出力します。pkgの状態はセッションが存続する間維持されるため、pkg.nの値は10のままです。sr_pkgの状態はサーバー・コールが存続する間のみ維持されるため、sr_pkg.nの値は5です。

```
CREATE OR REPLACE PACKAGE pkg IS
  n NUMBER := 5;
END pkg;
/
CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END sr_pkg;
/
BEGIN
  pkg.n := 10;
  sr_pkg.n := 10;
END;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('pkg.n: ' || pkg.n);
  DBMS_OUTPUT.PUT_LINE('sr_pkg.n: ' || sr_pkg.n);
END;
/
```

結果:

```
pkg.n: 10
sr_pkg.n: 5
```

10.7.3 SERIALLY_REUSABLEパッケージの明示カーソル

SERIALLY_REUSABLEパッケージの明示カーソルは、パッケージをクローズするかパッケージの作業単位(サーバー・コール)が終了するまでオープンされたままになります。カーソルを再度オープンするには、新しいサーバー・コールを作成する必要があります。

例10-6に示すとおり、サーバー・コールがサブプログラム起動と異なる場合があります。

対照的に、SERIALLY_REUSABLEでないパッケージの明示カーソルは、パッケージをクローズするかセッションから切断するまでオープンされたままになります。

例10-6 コール境界でオープンされるSERIALLY_REUSABLEパッケージのカーソル

```
DROP TABLE people;
CREATE TABLE people (name VARCHAR2(20));

INSERT INTO people (name) VALUES ('John Smith');
INSERT INTO people (name) VALUES ('Mary Jones');
INSERT INTO people (name) VALUES ('Joe Brown');
INSERT INTO people (name) VALUES ('Jane White');
CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  CURSOR c IS SELECT name FROM people;
END sr_pkg;
/

CREATE OR REPLACE PROCEDURE fetch_from_cursor IS
  v_name  people.name%TYPE;
BEGIN
  IF sr_pkg.c%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE('Cursor is open.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Cursor is closed; opening now.');
    OPEN sr_pkg.c;
  END IF;

  FETCH sr_pkg.c INTO v_name;
  DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);

  FETCH sr_pkg.c INTO v_name;
    DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);
  END fetch_from_cursor;
/
```

サーバーへの1回目のコール:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

結果:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
Fetched: Joe Brown
Fetched: Jane White
```

サーバーへの新たなコール:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

結果:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
Fetched: Joe Brown
Fetched: Jane White
```

10.8 パッケージ作成のガイドライン

- Oracle Databaseが提供するパッケージをよく理解して、その機能と重複する機能を持つパッケージを作成しないように注意してください。

  Oracle Databaseが提供するパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

- 別のアプリケーションで再利用できるように、パッケージに汎用性を持たせてください。
- パッケージ本体の前にパッケージ仕様部を設計および定義してください。
- パッケージ仕様部では、起動元のプログラムから見える必要のある項目のみを宣言してください。

  こうすることで、他の開発者が実装の細部に不適切に依存しないようにすることができ、再コンパイルの必要性を削減できます。

  パッケージ仕様部を変更する場合は、そのパッケージのパブリック・サブプログラムを起動するすべてのサブプログラムを再コンパイルする必要があります。パッケージ本体のみ変更する場合は、サブプログラムを再コンパイルする必要はありません。

- 例10-7に示すとおり、パブリック・カーソルはパッケージ仕様部で宣言し、パッケージ本体で定義してください。

  こうすると、カーソルの問合せをパッケージ・ユーザーに見えなくすることができ、カーソル宣言を変更せずにカーソル問合せを変更することができます。

- 初期値は、宣言部ではなくパッケージ本体の初期化部で代入してください。

  この方法には次のメリットがあります。

  - 初期値を計算するコードをより複雑にすることができ、詳しく説明できます。
  - 初期値の計算で例外が呼び出された場合、初期化部で独自の例外ハンドラを使用して例外を処理できます。

- データベース・アプリケーションを複数のPL/SQLパッケージ(APIを提供する1つのパッケージと機能を提供する複数のヘルパー・パッケージ)として実装する場合、例10-8に示すように、ヘルパー・パッケージをAPIパッケージに対してのみ使用可能にします。

例10-7では、カーソルc1の宣言および定義が、パッケージemp_stuffの仕様部と本体にそれぞれ置かれています。このカーソル宣言では戻り値のデータ型のみを指定し、問合せは指定せず、問合せはカーソルの定義に置かれています(構文およびセマンティクスの詳細は、「明示カーソルの宣言および定義」を参照してください)。

例10-8では、APIパッケージとヘルパー・パッケージを作成します。ヘルパー・パッケージ仕様部のACCESSIBLE BY句によって、APIパッケージのみがヘルパー・パッケージにアクセス可能となります。

例10-7 パッケージでのカーソル宣言と定義の分離

```
CREATE PACKAGE emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE;  -- Declare cursor
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE IS
    SELECT * FROM employees WHERE salary > 2500;  -- Define cursor
END emp_stuff;
/
```

例10-8 ACCESSIBLE BY句

```
CREATE OR REPLACE PACKAGE helper
  AUTHID DEFINER
  ACCESSIBLE BY (api)
IS
  PROCEDURE h1;
  PROCEDURE h2;
END;
/

CREATE OR REPLACE PACKAGE BODY helper
IS
  PROCEDURE h1 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Helper procedure h1');
  END;

  PROCEDURE h2 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Helper procedure h2');
  END;
END;
/

CREATE OR REPLACE PACKAGE api
  AUTHID DEFINER
IS
  PROCEDURE p1;
  PROCEDURE p2;
END;
/

CREATE OR REPLACE PACKAGE BODY api
IS
  PROCEDURE p1 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('API procedure p1');
```

```
    helper.h1;
  END;

  PROCEDURE p2 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('API procedure p2');
    helper.h2;
  END;
END;
/
```

APIパッケージのプロシージャを起動します。

```
BEGIN
  api.p1;
  api.p2;
END;
/
```

結果:

```
API procedure p1
Helper procedure h1
API procedure p2
Helper procedure h2
```

ヘルパー・パッケージのプロシージャを起動します。

```
BEGIN
  helper.h1;
END;
/
```

結果:

```
SQL> BEGIN
  2    helper.h1;
  3  END;
  4  /
  helper.h1;
  *
ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00904: insufficient privilege to access object HELPER
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

10.9 パッケージの例

例10-9では、表logとパッケージemp_adminを作成し、無名ブロックからパッケージ・サブプログラムを起動しています。パッケージには仕様部と本体の両方があります。

仕様部では、パブリック型、カーソルおよび例外と3つのパブリック・サブプログラムを宣言しています。1つのパブリック・サブプログラムはオーバーロードされたサブプログラムです(オーバーロードされたサブプログラムの詳細は、「オーバーロードされたサブプログラム」を参照してください)。

本体では、プライベート変数を宣言し、仕様部で宣言したパブリック・カーソルとサブプログラムを定義し、プライベート・ファンクションを宣言および定義しています。また、本体には初期化部が含まれています。

初期化部(無名ブロックがこのパッケージを初めて参照したときにのみ実行されます)では、表logに行を1行挿入し、プライベート変数number_hiredを0(ゼロ)に初期化しています。パッケージ・プロシージャhire_employeeが起動されるたびに、プライベート変数number_hiredが更新されます。

例10-9 emp_adminパッケージの作成

```
-- Log to track changes (not part of package):
DROP TABLE log;
CREATE TABLE log (
  date_of_action  DATE,
  user_id         VARCHAR2(20),
  package_name    VARCHAR2(30)
);
-- Package specification:
CREATE OR REPLACE PACKAGE emp_admin AUTHID DEFINER AS
  -- Declare public type, cursor, and exception:
  TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN EmpRecTyp;
  invalid_salary EXCEPTION;
  -- Declare public subprograms:
  FUNCTION hire_employee (
    last_name       VARCHAR2,
    first_name      VARCHAR2,
    email           VARCHAR2,
    phone_number    VARCHAR2,
    job_id          VARCHAR2,
    salary          NUMBER,
    commission_pct  NUMBER,
    manager_id      NUMBER,
    department_id   NUMBER
  ) RETURN NUMBER;
  -- Overload preceding public subprogram:
  PROCEDURE fire_employee (emp_id NUMBER);
  PROCEDURE fire_employee (emp_email VARCHAR2);
  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
  FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
/
-- Package body:
CREATE OR REPLACE PACKAGE BODY emp_admin AS
  number_hired  NUMBER;  -- private variable, visible only in this package
  -- Define cursor declared in package specification:
  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;
  -- Define subprograms declared in package specification:
  FUNCTION hire_employee (
```

```
  last_name       VARCHAR2,
  first_name      VARCHAR2,
  email           VARCHAR2,
  phone_number    VARCHAR2,
  job_id          VARCHAR2,
  salary          NUMBER,
  commission_pct  NUMBER,
  manager_id      NUMBER,
  department_id   NUMBER
) RETURN NUMBER
IS
  new_emp_id NUMBER;
BEGIN
  new_emp_id := employees_seq.NEXTVAL;
  INSERT INTO employees (
    employee_id,
    last_name,
    first_name,
    email,
    phone_number,
    hire_date,
    job_id,
    salary,
    commission_pct,
    manager_id,
    department_id
  )
  VALUES (
    new_emp_id,
    hire_employee.last_name,
    hire_employee.first_name,
    hire_employee.email,
    hire_employee.phone_number,
    SYSDATE,
    hire_employee.job_id,
    hire_employee.salary,
    hire_employee.commission_pct,
    hire_employee.manager_id,
    hire_employee.department_id
  );
  number_hired := number_hired + 1;
  DBMS_OUTPUT.PUT_LINE('The number of employees hired is '
                       || TO_CHAR(number_hired) );
  RETURN new_emp_id;
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
  DELETE FROM employees WHERE employee_id = emp_id;
END fire_employee;
PROCEDURE fire_employee (emp_email VARCHAR2) IS
BEGIN
  DELETE FROM employees WHERE email = emp_email;
END fire_employee;
-- Define private function, available only inside package:
FUNCTION sal_ok (
  jobid VARCHAR2,
  sal NUMBER
) RETURN BOOLEAN
```

```
  IS
    min_sal NUMBER;
    max_sal NUMBER;
  BEGIN
    SELECT MIN(salary), MAX(salary)
    INTO min_sal, max_sal
    FROM employees
    WHERE job_id = jobid;
    RETURN (sal >= min_sal) AND (sal <= max_sal);
  END sal_ok;
  PROCEDURE raise_salary (
    emp_id NUMBER,
    amount NUMBER
  )
  IS
    sal NUMBER(8,2);
    jobid VARCHAR2(10);
  BEGIN
    SELECT job_id, salary INTO jobid, sal
    FROM employees
    WHERE employee_id = emp_id;
    IF sal_ok(jobid, sal + amount) THEN  -- Invoke private function
      UPDATE employees
      SET salary = salary + amount
      WHERE employee_id = emp_id;
    ELSE
      RAISE invalid_salary;
    END IF;
  EXCEPTION
    WHEN invalid_salary THEN
      DBMS_OUTPUT.PUT_LINE ('The salary is out of the specified range.');
  END raise_salary;
  FUNCTION nth_highest_salary (
    n NUMBER
  ) RETURN EmpRecTyp
  IS
    emp_rec  EmpRecTyp;
  BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
      FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
  END nth_highest_salary;
BEGIN  -- initialization part of package body
   INSERT INTO log (date_of_action, user_id, package_name)
   VALUES (SYSDATE, USER, 'EMP_ADMIN');
   number_hired := 0;
END emp_admin;
/
-- Invoke packages subprograms in anonymous block:
DECLARE
  new_emp_id NUMBER(6);
BEGIN
  new_emp_id := emp_admin.hire_employee (
    'Belden',
    'Enrique',
```

```
    'EBELDEN',
    '555.111.2222',
    'ST_CLERK',
    2500,
    .1,
    101,
    110
  );
  DBMS_OUTPUT.PUT_LINE ('The employee id is ' || TO_CHAR(new_emp_id));
  emp_admin.raise_salary (new_emp_id, 100);
  DBMS_OUTPUT.PUT_LINE (
    'The 10th highest salary is '||
    TO_CHAR (emp_admin.nth_highest_salary(10).sal) ||
             ', belonging to employee: ' ||
             TO_CHAR (emp_admin.nth_highest_salary(10).emp_id)
  );
  emp_admin.fire_employee(new_emp_id);
  -- You can also delete the newly added employee as follows:
  -- emp_admin.fire_employee('EBELDEN');
END;
/
```

結果は次のようになります。

```
The number of employees hired is 1
The employee id is 210
The 10th highest salary is 11500, belonging to employee: 168
```

10.10 STANDARDパッケージによるPL/SQL環境の定義

STANDARDという名前のパッケージではPL/SQL環境を定義しています。このパッケージの仕様部では、パブリック型、変数、例外、サブプログラムを宣言し、それらは自動的にPL/SQLプログラムで使用可能になります。たとえば、パッケージSTANDARDでは、引数の絶対値を戻すファンクションABSを次のように宣言します。

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

パッケージSTANDARDの内容は、アプリケーションから直接見ることができます。その内容を参照する場合もパッケージ名に接頭辞を付けて修飾名にする必要はありません。たとえば、ABSはデータベース・トリガー、ストアド・サブプログラム、Oracleのツール製品または3GLアプリケーションから次のように起動できます。

```
abs_diff := ABS(x - y);
```

ユーザー独自のABSを宣言すると、ローカル宣言がパブリック宣言をオーバーライドします。ただし、次に示すとおり、完全な名前を指定して、SQLファンクションを起動できます。

```
abs_diff := STANDARD.ABS(x - y);
```

ほとんどのSQLファンクションはオーバーロードされています。たとえば、パッケージSTANDARDには次のような宣言があります。

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQLは、仮パラメータと実パラメータの数とデータ型を比較して、どのTO_CHARの起動かを判定します。

11 PL/SQLのエラー処理

この章では、PL/SQLのコンパイル時の警告およびPL/SQLランタイム・エラーの処理方法について説明します。後者は例外と呼ばれます。

ノート:

警告およびエラー・メッセージの言語は NLS_LANGUAGE パラメータに依存します。このパラメータの詳細は、『Oracle Database グローバリゼーション・サポート・ガイド』を参照してください。

ここでのトピック

- コンパイル時の警告
- 例外処理の概要
- 内部的に定義された例外
- 事前定義の例外
- ユーザー定義の例外
- 再宣言された事前定義の例外
- 例外の明示的な呼出し
- 例外の伝播
- 未処理の例外
- エラー・コードとエラー・メッセージの取得
- 例外処理後に実行を続ける方法
- 例外処理後のトランザクションの再試行
- 分散問合せでのエラー処理

関連項目:

- 「トリガーでの例外処理」
- 「FORALL文が完了した後のFORALL例外の処理」

ヒント:

PL/SQL コードの作成時または実行時に問題が発生した場合は、Oracle Database のトレース・ファイルを確認してください。DIAGNOSTIC_DEST 初期化パラメータにトレース・ファイルの現在の位置が示されます。このパラメータの値を見つけるには、SHOW PARAMETER DIAGNOSTIC_DEST を発行するか、V$DIAG_INFO ビューを問い合せます。診断データの詳細は、Oracle Database 管理者ガイドを参照してください。

11.1 コンパイル時の警告

PL/SQLコンパイラは、ストアドPL/SQLユニットのコンパイル中に、エラーを生成してコンパイルの妨げになるほど重大ではない条件(非推奨のPL/SQL機能を使用している場合など)に対して、警告を生成します。

コンパイル中に生成される警告(およびエラー)を確認するには、静的データ・ディクショナリ・ビュー*_ERRORSを問い合せるか、SQL*Plus環境でコマンドのSHOW ERRORSを使用します。

PL/SQL警告のメッセージ・コードは、PLW-nnnnnという形式です。

表11-1 コンパイル時の警告カテゴリ

カテゴリ	説明	例
SEVERE	条件によって、予期しないアクションまたは誤った結果が発生する可能性があります。	パラメータのエイリアシング問題
PERFORMANCE	条件によって、パフォーマンス問題が発生する可能性があります。	INSERT 文で NUMBER 列に VARCHAR2 値が渡した場合
INFORMATIONAL	条件は、パフォーマンスまたは正確性には影響しませんが、コードのメンテナンス性の向上のために変更する必要がある場合があります。	実行できないコード

コンパイル・パラメータPLSQL_WARNINGSを設定すると、次の操作を実行できます。

- すべての警告、1つ以上の警告カテゴリまたは特定の警告の有効化および無効化
- 特定の警告のエラーとしての処理(その条件が修正されるまで、PL/SQLユニットはコンパイルできません)

PLSQL_WARNINGSの値は、次のものに対して設定できます。

- Oracle Databaseインスタンス

  ALTER SYSTEM文を使用します。この文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

- セッション

  ALTER SESSION文を使用します。この文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

- ストアドPL/SQLユニット

  「ALTER文」のALTER文をcompiler_parameters_clauseとともに使用します。

前述のいずれかのALTER文で、PLSQL_WARNINGSの値を次の構文で設定します。

```
PLSQL_WARNINGS = 'value_clause' [, 'value_clause' ] ...
```

value_clauseの構文は、『Oracle Databaseリファレンス』を参照してください。

PLSQL_WARNINGSの現在の値を表示するには、静的データ・ディクショナリ・ビューALL_PLSQL_OBJECT_SETTINGSを問い合せます。

関連項目:

- 静的データ・ディクショナリ・ビューALL_PLSQL_OBJECT_SETTINGSの詳細は、『Oracle Databaseリファレンス』を参照してください
- すべてのPL/SQL警告のメッセージ・コードは、『Oracle Databaseエラー・メッセージ』を参照してください
- 静的データ・ディクショナリ・ビュー*_ERRORSの詳細は、『Oracle Databaseリファレンス』を参照してください
- PL/SQLユニットおよびコンパイラ・パラメータの詳細は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください

例11-1 PLSQL_WARNINGSコンパイル・パラメータの値の設定

この例では、PLSQL_WARNINGSの値を設定するALTER文をいくつか示します。

セッションで、すべての警告を有効にします(開発時に特にお薦めします)。

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

セッションで、PERFORMANCE警告を有効にします。

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
```

プロシージャloc_varで、PERFORMANCE警告を有効にし、設定を再利用します。

```
ALTER PROCEDURE loc_var
  COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'
  REUSE SETTINGS;
```

セッションで、SEVERE警告を有効にし、PERFORMANCE警告を無効にして、PLW-06002警告をエラーとして処理します。

```
ALTER SESSION
  SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE', 'ERROR:06002';
```

セッションで、すべての警告を無効にします。

```
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
```

11.1.1 DBMS_WARNINGパッケージ

PL/SQLユニットをコンパイルする開発環境(SQL*Plusなど)でPL/SQLユニットを作成している場合、DBMS_WARNINGパッケージ内のサブプログラムを起動することで、PLSQL_WARNINGSの値を表示および設定できます。

例11-2では、ALTER SESSION文を使用して、このセッションのすべての警告メッセージを無効化した後、使用不可コードを含むプロシージャをコンパイルしています。このプロシージャは警告なしでコンパイルされます。次に、DBMS_WARNING.set_warning_setting_stringを起動してこのセッションのすべての警告を有効化し、DBMS_WARNING.get_warning_setting_stringを起動してPLSQL_WARNINGSの値を表示しています。最後に、プロシージャを再コンパイルすると、コンパイラは使用不可コードに関する警告を生成します。

ノート:

使用不可コードは、間違いの可能性またはデバッグ・フラグで意図的に隠されている可能性があります。

複数のネストしたSQL*Plusスクリプトで構成され、サブプログラムごとに異なるPLSQL_WARNINGSを設定する必要がある複雑なアプリケーションをコンパイルする場合は、DBMS_WARNINGサブプログラムが便利です。DBMS_WARNINGサブプログラムを使用すると、現行のPLSQL_WARNINGS設定を保存し、設定を変更して特定のサブプログラム・セットをコンパイルしてから、設定を元の値にリストアすることができます。

関連項目:

DBMS_WARNINGパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください

例11-2 DBMS_WARNINGサブプログラムを使用したPLSQL_WARNINGSの表示と設定

このセッションで、すべての警告メッセージを無効にします。

```
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
```

警告を無効にすると、このプロシージャは警告なしでコンパイルされます。

```
CREATE OR REPLACE PROCEDURE unreachable_code AUTHID DEFINER AS
  x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
```

このセッションで、すべての警告メッセージを有効にします。

```
CALL DBMS_WARNING.set_warning_setting_string ('ENABLE:ALL', 'SESSION');
```

警告の設定を確認します。

```
SELECT DBMS_WARNING.get_warning_setting_string() FROM DUAL;
```

結果:

```
DBMS_WARNING.GET_WARNING_SETTING_STRING()
-----------------------------------------

ENABLE:ALL

1 row selected.
```

プロシージャを再コンパイルします。

```
ALTER PROCEDURE unreachable_code COMPILE;
```

結果:

```
SP2-0805: Procedure altered with compilation warnings
```

エラーを表示します。

```
SHOW ERRORS
```

結果:

```
Errors for PROCEDURE UNREACHABLE_CODE:

LINE/COL ERROR
-------- -----------------------------------------------------------------
7/5      PLW-06002: Unreachable code
```

11.2 例外処理の概要

例外(PL/SQLランタイム・エラー)は、設計の失敗、コーディングの間違い、ハードウェアの障害などの多くの原因で発生する可能性があります。発生する可能性があるすべての例外を予想することはできませんが、例外が発生してもプログラムで処理を継続できるように例外ハンドラを記述できます。

すべてのPL/SQLブロックに例外処理部を配置でき、そこに1つ以上の例外ハンドラを配置できます。たとえば、例外処理部に次の構文を使用することもできます。

```
EXCEPTION
  WHEN ex_name_1 THEN statements_1                  -- Exception handler
  WHEN ex_name_2 OR ex_name_3 THEN statements_2  -- Exception handler
  WHEN OTHERS THEN statements_3                       -- Exception handler
END;
```

前述の構文例で、ex_name_nは例外の名前、statements_nは1つ以上の文です。(構文およびセマンティクスの詳細は、「例外ハンドラ」を参照してください。)

ブロックの実行部で例外が呼び出されると、実行部は中止され、制御は例外処理部に移ります。ex_name_1が呼び出された場合は、statements_1が実行されます。ex_name_2またはex_name_3のいずれかが呼び出された場合は、statements_2が実行されます。それ以外の例外が呼び出された場合は、statements_3が実行されます。

例外ハンドラが実行されると、制御は外側のブロックの次の文に移ります。外側にブロックがない場合は、次のとおりです。

- 例外ハンドラがサブプログラム内にある場合、制御は起動元の起動後の文に戻ります。
- 例外ハンドラが無名ブロック内にある場合、制御はホスト環境(SQL*Plusなど)に移ります。

例外ハンドラを持たないブロックで例外が呼び出された場合は、例外が伝播します。つまり、その例外のハンドラを持つブロックに至るか外側のブロックがなくなるまで、1つずつ外側のブロックに進んで例外が再生されます(詳細は「例外の伝播」を参照してください)。その例外に対応するハンドラがない場合、PL/SQLは起動元またはホスト環境に未処理例外エラーを戻します。その結果は戻す場所によって異なります(詳細は「未処理例外」を参照してください)。

ここでのトピック

- 例外のカテゴリ
- 例外ハンドラのメリット
- 例外の回避および処理のガイドライン

11.2.1 例外のカテゴリ

例外のカテゴリは次のとおりです。

- 内部的定義

  内部的に定義された例外は、ランタイム・システムによって暗黙的(自動的)に呼び出されます。内部的に定義された例外の例には、ORA-00060(リソース待機の間にデッドロックが検出されました。)やORA-27102(メモリーが不足しています。)などがあります。

  内部的に定義された例外には必ずエラー・コードがありますが、PL/SQLで指定されているかユーザーが指定しないかぎり、名前はありません。

  詳細は、「内部的に定義された例外」を参照してください。

- 事前定義

  事前定義の例外とは、PL/SQLにより名前が指定されている内部的に定義された例外です。たとえば、ORA-06500(PL/SQL: 記憶域エラー)にはSTORAGE_ERRORという名前が事前に定義されています。

  詳細は、「事前定義の例外」を参照してください。

- ユーザー定義

  PL/SQLの無名ブロック、サブプログラムまたはパッケージの宣言部で、ユーザー独自の例外を宣言できます。たとえば、残高がマイナスになっている銀行口座にフラグを付けるために、insufficient_fundsという名前の例外を宣言できます。

  ユーザー定義の例外は明示的に呼び出す必要があります。

  詳細は、「ユーザー定義の例外」を参照してください。

表11-2に、例外のカテゴリの概要を示します。

表11-2 例外のカテゴリ

カテゴリ	定義者	エラー・コード	名前	暗黙的な呼出し	明示的な呼出し
内部的定義	ランタイム・システム	常に	ユーザーが割り当てた場合のみ	あり	オプション[脚注 1]
事前定義	ランタイム・システム	常に	常に	あり	オプション[脚注 1]

カテゴリ	定義者	エラー・コード	名前	暗黙的な呼出し	明示的な呼出し
ユーザー定義	ユーザー	ユーザーが割り当てた場合のみ	常に	なし	常に

脚注1

詳細は、「内部的に定義された例外のRAISE文による呼出し」を参照してください。

名前付き例外の場合は、OTHERS例外ハンドラを使用して処理するかわりに、専用の例外ハンドラを記述できます。OTHERS例外ハンドラの場合は、処理しようとしている例外をファンクションを起動して決定する必要があるため、専用の例外ハンドラの方が効率的です。詳細は、「エラー・コードとエラー・メッセージの取得」を参照してください。

11.2.2 例外ハンドラのメリット

エラー処理に例外ハンドラを使用すると、プログラムの作成および理解が容易になり、未処理例外が発生する可能性が減少します。

例外ハンドラを使用しない場合は、発生する可能性のあるあらゆるエラーを、発生する可能性のあるあらゆる場所でチェックし、処理する必要があります。発生する可能性のあるエラーまたはエラーが発生する可能性のある場所というのは、ただちに検出できないエラーの場合(たとえば、不正なデータは、計算で使用するまで検出できない可能性があります)は特に見落としやすいものです。エラー処理コードはプログラムのいたるところに存在することになります。

例外ハンドラを使用する場合は、発生する可能性のあるあらゆるエラー、またはエラーが発生する可能性のあるあらゆる場所を把握する必要がありません。エラーが発生する可能性のある各ブロックに、例外処理部を含めるだけでかまいません。例外処理部には、特定のエラーおよび不明なエラーの両方の例外ハンドラを含めることができます。ブロック(サブブロックを含む)のどこかでエラーが発生すると、例外ハンドラにより処理されます。エラー処理コードは、ブロックの例外処理部に分離されます。

例11-3のプロシージャでは、1つの例外ハンドラを使用してNO_DATA_FOUNDという事前定義の例外を処理していますが、この例外は2つのSELECT INTO文のいずれでも発生する可能性があります。

複数の文で同じ例外ハンドラを使用していて、失敗した文がどれなのかを知る必要がある場合は、例11-4に示すとおり、ロケータ変数を使用できます。

エラー処理コードの精度はユーザーが決定します。すべての0(ゼロ)による除算エラー、不正な配列の索引などに対応する1つの例外ハンドラを持つことができます。独自の例外ハンドラを持つブロック内に単一の文を配置することによって、その文に含まれるエラーをチェックすることもできます。

例11-3 複数の例外に対応した1つの例外ハンドラ

```
CREATE OR REPLACE PROCEDURE select_item (
  t_column VARCHAR2,
  t_name   VARCHAR2
) AUTHID DEFINER
IS
```

```
  temp VARCHAR2(30);
BEGIN
  temp := t_column;  -- For error message if next SELECT fails

  -- Fails if table t_name does not have column t_column:

  SELECT COLUMN_NAME INTO temp
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = UPPER(t_name)
  AND COLUMN_NAME = UPPER(t_column);

  temp := t_name;  -- For error message if next SELECT fails

  -- Fails if there is no table named t_name:

  SELECT OBJECT_NAME INTO temp
  FROM USER_OBJECTS
  WHERE OBJECT_NAME = UPPER(t_name)
  AND OBJECT_TYPE = 'TABLE';

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp);
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('Unexpected error');
    RAISE;
END;
/
```

プロシージャを起動します(DEPARTMENTS表は存在しますが、この表にLAST_NAMEという列はありません)。

```
BEGIN
  select_item('departments', 'last_name');
END;
/
```

結果:

```
No Data found for SELECT on departments
```

プロシージャを起動します(EMP表は存在しません)。

```
BEGIN
  select_item('emp', 'last_name');
END;
/
```

結果:

```
No Data found for SELECT on emp
```

例11-4 例外ハンドラを共有する文のロケータ変数

```
CREATE OR REPLACE PROCEDURE loc_var AUTHID DEFINER IS
  stmt_no  POSITIVE;
  name_    VARCHAR2(100);
BEGIN
```

```
  stmt_no := 1;
  SELECT table_name INTO name_
  FROM user_tables
  WHERE table_name LIKE 'ABC%';
  stmt_no := 2;
  SELECT table_name INTO name_
  FROM user_tables
  WHERE table_name LIKE 'XYZ%';
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Table name not found in query ' || stmt_no);
END;
/
CALL loc_var();
```

結果:

```
Table name not found in query 1
```

11.2.3 例外の回避および処理のガイドライン

可能なかぎり信頼性の高い安全なプログラムを作成するには:

- エラー・チェック・コードと例外ハンドラの両方を使用します。

  不正な入力データが原因でエラーが発生する可能性がある場所には常に、エラー・チェック・コードを使用します。不正な入力データの例には、不適切またはNULLである実パラメータ、行を戻さないまたは予想以上の行を戻す問合せがあります。不正な入力データの様々な組合せでコードをテストし、どのような潜在的なエラーが発生するかを調べます。

  エラー・チェック・コードを使用して、例外の発生を回避できる場合もあります(例11-7を参照)。

- エラーが発生する可能性がある場所には常に、例外ハンドラを追加します。

  算術演算、文字列操作、データベース操作の間は、エラーが発生する可能性が特に高くなります。コードとは関係のない問題(ディスク記憶域やメモリーのハードウェアの障害など)が原因でエラーが発生することもありますが、コードには対処措置が必要です。

- データベースが想定外の状態になった場合にもプログラムが動作するように設計します。

  たとえば、問い合せた表で列の追加や削除が行われていたり、列の型が変更されている場合があります。こうした問題を回避するには、スカラー変数を%TYPE修飾子で宣言し、問合せ結果を保持するレコード変数を%ROWTYPE修飾子で宣言します。

- 可能な場合は必ず、OTHERS例外ハンドラを使用するかわりに、名前付き例外用の例外ハンドラを記述します。

  事前定義の例外の名前と原因を調べておいてください。名前のない内部的に定義された特定の例外がデータベース操作によって呼び出される可能性があるとわかっている場合は、その例外に名前を付け、それ専用の例外ハンドラを記述できるようにします。

- 例外ハンドラからデバッグ情報が出力されるようにします。

  個別の表にデバッグ情報を格納する場合は、メイン・サブプログラムが実行した処理をロールバックする場合でもデバッグ情報をコミットできるように、自律型ルーチンを使用します。自立型ルーチンの詳細は、「AUTONOMOUS_TRANSACTIONプラグマ」を参照してください。

- 各例外ハンドラについて、トランザクションをコミットさせるのか、ロールバックさせるのか、継続させるのかを慎重に考慮し

ます。

エラーの重大さに関係なく、データベースを一貫性のある状態に保ち、不正なデータの格納を避けるようにしてください。

- すべてのPL/SQLプログラムの最上位にOTHERS例外ハンドラを配置し、未処理例外が発生しないようにします。

  OTHERS例外ハンドラの最後の文には、RAISEまたはRAISE_APPLICATION_ERRORプロシージャの起動のいずれかを使用してください。(この方法に従わない場合は、PL/SQL警告が有効になり、PLW-06009が表示されます。)RAISEまたはRAISE_APPLICATION_ERRORの起動の詳細は、「例外の明示的な呼出し」を参照してください。

11.3 内部的に定義された例外

内部的に定義された例外(ORA-nエラー)の詳細はOracle Databaseエラー・メッセージ・リファレンスを参照してください。これらの例外はランタイム・システムによって暗黙的(自動的)に呼び出されます。

内部的に定義された例外には、PL/SQLで指定されている(「事前定義の例外」を参照)かユーザーが指定しないかぎり、名前はありません。

名前のない内部的に定義された特定の例外がデータベース操作によって呼び出される可能性があるとわかっている場合は、その例外に名前を付け、それ専用の例外ハンドラを記述できるようにします。そうしない場合は、OTHERS例外ハンドラ以外では処理できません。

内部的に定義された例外に名前を指定するには、適切な無名ブロック、サブプログラムまたはパッケージの宣言部で次の処理を実行します。(適切なブロックを判定する方法は、「例外の伝播」を参照してください。)

1. 名前を宣言します。

   例外名の宣言には次の構文を使用します。

   ```
   exception_name EXCEPTION;
   ```

   セマンティクスの詳細は、「例外宣言」を参照してください。

2. 内部的に定義された例外のエラー・コードと名前を関連付けます。

   構文は次のとおりです。

   ```
   PRAGMA EXCEPTION_INIT (exception_name, error_code)
   ```

   セマンティクスの詳細は、「EXCEPTION_INITプラグマ」を参照してください。

> ノート:
>
> 内部的に定義された例外は、ユーザーが宣言した名前が付いていても内部的に定義された例外のままであり、ユーザー定義の例外ではありません。

例11-5では、内部的に定義された例外ORA-00060(リソース待機の間にデッドロックが検出されました。)にdeadlock_detectedという名前を指定し、この名前を例外ハンドラで使用しています。

関連項目:

「内部的に定義された例外のRAISE文による呼出し」

例11-5 内部的に定義された例外への名前付け

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
  ...
EXCEPTION
  WHEN deadlock_detected THEN
    ...
END;
/
```

11.4 事前定義の例外

事前定義の例外は、事前定義の名前を持つ内部的に定義された例外で、PL/SQLによりパッケージSTANDARDの中でグローバルに宣言されています。事前定義の例外はランタイム・システムによって暗黙的(自動的)に呼び出されます。事前定義の例外には名前が付いているため、それ専用の例外ハンドラを記述できます。

表11-3に、事前定義の例外の名前とエラー・コードを示します。

表11-3 PL/SQLの事前定義の例外

例外名	エラー・コード
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
COLLECTION_IS_NULL	-6531
CURSOR_ALREADY_OPEN	-6511
DUP_VAL_ON_INDEX	-1
INVALID_CURSOR	-1001
INVALID_NUMBER	-1722
LOGIN_DENIED	-1017
NO_DATA_FOUND	+100
NO_DATA_NEEDED	-6548

例外名	エラー・コード
NOT_LOGGED_ON	-1012
PROGRAM_ERROR	-6501
ROWTYPE_MISMATCH	-6504
SELF_IS_NULL	-30625
STORAGE_ERROR	-6500
SUBSCRIPT_BEYOND_COUNT	-6533
SUBSCRIPT_OUTSIDE_LIMIT	-6532
SYS_INVALID_ROWID	-1410
TIMEOUT_ON_RESOURCE	-51
TOO_MANY_ROWS	-1422
VALUE_ERROR	-6502
ZERO_DIVIDE	-1476

例11-6では、企業の株価収益率を計算しています。企業の収益が0(ゼロ)の場合に除算演算を実行すると、事前定義の例外ZERO_DIVIDEが呼び出され、ブロックの実行部から例外処理部に制御が移ります。

例11-7では、例11-6で処理されている例外を、エラー・チェック・コードを使用して回避しています。

例11-8では、プロシージャは、パラメータdiscrimの値に応じてEMPLOYEES表またはDEPARTMENTS表のカーソル変数をオープンします。無名ブロックは、プロシージャを起動してEMPLOYEES表のカーソル変数をオープンしますが、DEPARTMENTS表からフェッチするため、事前定義の例外ROWTYPE_MISMATCHが発生します。

関連項目:

「内部的に定義された例外のRAISE文による呼出し」

例11-6 ZERO_DIVIDEを処理する無名ブロック

```
DECLARE
  stock_price   NUMBER := 9.73;
  net_earnings  NUMBER := 0;
  pe_ratio      NUMBER;
BEGIN
```

```
  pe_ratio := stock_price / net_earnings;  -- raises ZERO_DIVIDE exception
  DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Company had zero earnings.');
    pe_ratio := NULL;
END;
/
```

結果:

```
Company had zero earnings.
```

例11-7 ZERO_DIVIDEを回避する無名ブロック

```
DECLARE
  stock_price   NUMBER := 9.73;
  net_earnings  NUMBER := 0;
  pe_ratio      NUMBER;
BEGIN
  pe_ratio :=
    CASE net_earnings
      WHEN 0 THEN NULL
      ELSE stock_price / net_earnings
    END;
END;
/
```

例11-8 ROWTYPE_MISMATCHを処理する無名ブロック

```
CREATE OR REPLACE PACKAGE emp_dept_data AUTHID DEFINER AS
  TYPE cv_type IS REF CURSOR;

  PROCEDURE open_cv (
    cv       IN OUT cv_type,
    discrim  IN     POSITIVE
  );
  END emp_dept_data;
/

CREATE OR REPLACE PACKAGE BODY emp_dept_data AS
  PROCEDURE open_cv (
    cv      IN OUT cv_type,
    discrim IN     POSITIVE) IS
  BEGIN
    IF discrim = 1 THEN
    OPEN cv FOR
      SELECT * FROM EMPLOYEES ORDER BY employee_id;
    ELSIF discrim = 2 THEN
      OPEN cv FOR
        SELECT * FROM DEPARTMENTS ORDER BY department_id;
    END IF;
  END open_cv;
END emp_dept_data;
/
```

無名ブロックからプロシージャopen_cvを起動します。

```
DECLARE
  emp_rec   EMPLOYEES%ROWTYPE;
  dept_rec  DEPARTMENTS%ROWTYPE;
  cv        Emp_dept_data.CV_TYPE;
BEGIN
  emp_dept_data.open_cv(cv, 1);  -- Open cv for EMPLOYEES fetch.
  FETCH cv INTO dept_rec;        -- Fetch from DEPARTMENTS.
  DBMS_OUTPUT.PUT(dept_rec.DEPARTMENT_ID);
  DBMS_OUTPUT.PUT_LINE('  ' || dept_rec.LOCATION_ID);
EXCEPTION
  WHEN ROWTYPE_MISMATCH THEN
     BEGIN
       DBMS_OUTPUT.PUT_LINE
         ('Row type mismatch, fetching EMPLOYEES data ...');
       FETCH cv INTO emp_rec;
       DBMS_OUTPUT.PUT(emp_rec.DEPARTMENT_ID);
       DBMS_OUTPUT.PUT_LINE('  ' || emp_rec.LAST_NAME);
     END;
END;
/
```

結果:

```
Row type mismatch, fetching EMPLOYEES data ...
90  King
```

11.5 ユーザー定義の例外

PL/SQLの無名ブロック、サブプログラムまたはパッケージの宣言部で、ユーザー独自の例外を宣言できます。

例外名の宣言には次の構文を使用します。

```
exception_name EXCEPTION;
```

セマンティクスの詳細は、「例外宣言」を参照してください。

ユーザー定義の例外は明示的に呼び出す必要があります。詳細は、「例外の明示的な呼出し」を参照してください。

11.6 再宣言された事前定義の例外

事前定義の例外の再宣言、つまり、事前定義の例外の名前と同じ名前でユーザー定義の例外を宣言することはお薦めしません。(事前定義の例外の名前のリストは、表11-3を参照してください。)

事前定義の例外を再宣言すると、ローカルな宣言がパッケージSTANDARDのグローバルな宣言をオーバーライドします。パッケージ名STANDARDで例外名を修飾しないかぎり、グローバルに宣言された例外のために記述された例外ハンドラでは例外を処理できなくなります。

例11-9に、これを示します。

例11-9 再宣言された事前定義の識別子

```
DROP TABLE t;
CREATE TABLE t (c NUMBER);
```

次のブロックでは、INSERT文から事前定義の例外INVALID_NUMBERが暗黙的に呼び出され、例外ハンドラにより処理されています。

```
DECLARE
  default_number NUMBER := 0;
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

結果:

```
Substituting default value for invalid number.
```

次のブロックでは、事前定義の例外INVALID_NUMBERを再宣言しています。INSERT文から事前定義の例外INVALID_NUMBERが暗黙的に呼び出されても、例外ハンドラでは処理されません。

```
DECLARE
  default_number NUMBER := 0;
  i NUMBER := 5;
  invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-01722: invalid number
ORA-06512: at line 6
```

例外ハンドラの例外名を修飾すると、前述のブロックの例外ハンドラにより事前定義の例外INVALID_NUMBERが処理されます。

```
DECLARE
  default_number NUMBER := 0;
  i NUMBER := 5;
  invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
```

```
  WHEN STANDARD.INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

結果:

```
Substituting default value for invalid number.
```

11.7 例外の明示的な呼出し

例外を明示的に呼び出すには、RAISE文またはRAISE_APPLICATION_ERRORプロシージャのいずれかを使用します。

ここでのトピック

- RAISE文
- RAISE_APPLICATION_ERRORプロシージャ

11.7.1 RAISE文

RAISE文は、例外を明示的に呼び出します。例外ハンドラの外側では、例外名を指定する必要があります。例外ハンドラの内側で、例外名を省略すると、RAISE文は現行の例外を再度呼び出します。

ここでのトピック

- ユーザー定義の例外のRAISE文による呼出し
- 内部的に定義された例外のRAISE文による呼出し
- 現行の例外のRAISE文による再呼出し

11.7.1.1 ユーザー定義の例外のRAISE文による呼出し

例11-10のプロシージャでは、past_dueという名前の例外を宣言し、この例外をRAISE文で明示的に呼び出し、例外ハンドラで処理しています。

例11-10 ユーザー定義の例外の宣言、呼出しおよび処理

```
CREATE PROCEDURE account_status (
  due_date DATE,
  today    DATE
) AUTHID DEFINER
IS
  past_due  EXCEPTION;  -- declare exception
BEGIN
  IF due_date < today THEN
    RAISE past_due;  -- explicitly raise exception
  END IF;
EXCEPTION
  WHEN past_due THEN  -- handle exception
    DBMS_OUTPUT.PUT_LINE ('Account past due.');
END;
```

```
/

BEGIN
  account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
                  TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));
END;
/
```

結果:

```
Account past due.
```

11.7.1.2 内部的に定義された例外のRAISE文による呼出し

内部的に定義された例外はランタイム・システムにより暗黙的に呼び出されますが、名前が付いている場合はRAISE文を使用して明示的に呼び出すことができます。表11-3に、事前定義の名前を持つ内部的に定義された例外を示します。内部的に定義された例外にユーザーが宣言した名前を指定する方法は、「内部的に定義された例外」を参照してください。

内部的に定義された名前付き例外の例外ハンドラは、その例外の呼出しが暗黙的か明示的かに関係なく、例外を処理します。

例11-11のプロシージャでは、事前定義の例外INVALID_NUMBERが明示的または暗黙的に呼び出され、必ずINVALID_NUMBER例外ハンドラによって処理されます。

例11-11 事前定義の例外の明示的な呼出し

```
DROP TABLE t;
CREATE TABLE t (c NUMBER);

CREATE PROCEDURE p (n NUMBER) AUTHID DEFINER IS
  default_number NUMBER := 0;
BEGIN
  IF n < 0 THEN
    RAISE INVALID_NUMBER;  -- raise explicitly
  ELSE
    INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));  -- raise implicitly
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/

BEGIN
  p(-1);
END;
/
```

結果:

```
Substituting default value for invalid number.

BEGIN
```

```
  p(1);
END;
/
```

結果:

```
Substituting default value for invalid number.
```

11.7.1.3 現行の例外のRAISE文による再呼出し

例外ハンドラでは、RAISE文を使用して処理中の例外を「再呼出し」できます。例外を再呼出しすると、外側のブロックに例外を渡してさらに処理することができます。(呼び出された例外を外側のブロックで処理できない場合は例外が伝播します。詳細は「例外の伝播」を参照してください。)現行の例外を再呼出しする場合、例外名を指定する必要はありません。

例11-12の例外処理は、内側のブロックから開始して外側のブロックで終了します。外側のブロックで例外を宣言しているため、両方のブロックに例外名が存在し、各ブロックにこの例外専用の例外ハンドラがあります。内部ブロックが例外を呼び出すと、内部ブロックの例外ハンドラは最初の処理を実行した後にこの例外を再度呼び出し、さらに処理するために外側のブロックに渡しています。

例11-12 例外の再呼出し

```
DECLARE
  salary_too_high   EXCEPTION;
  current_salary    NUMBER := 20000;
  max_salary        NUMBER := 10000;
  erroneous_salary  NUMBER;
BEGIN
  BEGIN
    IF current_salary > max_salary THEN
      RAISE salary_too_high;   -- raise exception
    END IF;
  EXCEPTION
    WHEN salary_too_high THEN  -- start handling exception
      erroneous_salary := current_salary;
      DBMS_OUTPUT.PUT_LINE('Salary ' || erroneous_salary ||' is out of range.');
      DBMS_OUTPUT.PUT_LINE ('Maximum salary is ' || max_salary || '.');
      RAISE;  -- reraise current exception (exception name is optional)
  END;
EXCEPTION
  WHEN salary_too_high THEN    -- finish handling exception
    current_salary := max_salary;
    DBMS_OUTPUT.PUT_LINE (
      'Revising salary from ' || erroneous_salary ||
      ' to ' || current_salary || '.'
    );
END;
/
```

結果:

```
Salary 20000 is out of range.
Maximum salary is 10000.
Revising salary from 20000 to 10000.
```

11.7.2 RAISE_APPLICATION_ERRORプロシージャ

(DBMS_STANDARDパッケージで定義されている)RAISE_APPLICATION_ERRORプロシージャは、ストアド・サブプログラムまたはメソッドからのみ起動できます。このプロシージャを起動する通常の目的は、ユーザー定義の例外を呼び出してそのエラー・コードとエラー・メッセージを起動元に戻すことです。

RAISE_APPLICATION_ERRORを起動するには、次の構文を使用します。

```
RAISE_APPLICATION_ERROR (error_code, message[, {TRUE | FALSE}]);
```

EXCEPTION_INITプラグマを使用してerror_codeをユーザー定義の例外に代入しておく必要があります。構文は次のとおりです。

```
PRAGMA EXCEPTION_INIT (exception_name, error_code)
```

error_codeは-20000から-20999の範囲の整数で、messageは長さが2048バイト以内の文字列です。

セマンティクスの詳細は、「EXCEPTION_INITプラグマ」を参照してください。

messageは長さが2048バイト以内の文字列です。

TRUEを指定すると、PL/SQLによりerror_codeがエラー・スタックの最上位に配置されます。それ以外の場合は、PL/SQLによりエラー・スタックがerror_codeに置き換えられます。

例11-13の無名ブロックでは、past_dueという名前の例外を宣言し、この例外にエラー・コード-20000を代入し、ストアド・プロシージャを起動しています。このストアド・プロシージャは、エラー・コード-20000とメッセージを使用してRAISE_APPLICATION_ERRORプロシージャを起動し、制御が無名ブロックに戻ったところで例外を処理します。例外に関連付けられているメッセージを取得するために、無名ブロックの例外ハンドラがSQLERRMファンクション(詳細は、「エラー・コードとエラー・メッセージの取得」を参照)を起動しています。

例11-13 RAISE_APPLICATION_ERRORによるユーザー定義の例外の呼出し

```
CREATE PROCEDURE account_status (
  due_date DATE,
  today    DATE
) AUTHID DEFINER
IS
BEGIN
  IF due_date < today THEN                    -- explicitly raise exception
    RAISE_APPLICATION_ERROR(-20000, 'Account past due.');
  END IF;
END;
/

DECLARE
  past_due  EXCEPTION;                        -- declare exception
  PRAGMA EXCEPTION_INIT (past_due, -20000);   -- assign error code to exception
BEGIN
  account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
                  TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));   -- invoke procedure
EXCEPTION
  WHEN past_due THEN                          -- handle exception
```

```
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQLERRM(-20000)));
END;
/
```

結果:

```
ORA-20000: Account past due.
```

11.8 例外の伝播

例外ハンドラを持たないブロックで例外が呼び出された場合は、例外が伝播します。つまり、その例外のハンドラを持つブロックに至るか外側のブロックがなくなるまで、1つずつ外側のブロックに進んで例外が再生されます。その例外に対応するハンドラがない場合、PL/SQLは起動元またはホスト環境に未処理例外エラーを戻します。その結果は戻す場所によって異なります(詳細は「未処理例外」を参照してください)。

図11-1では、1つのブロックが別のブロックの内側にネストされています。内側のブロックが例外Aを呼び出します。内側のブロックにはAの例外ハンドラがあるため、Aは伝播しません。例外ハンドラが実行されると、外側のブロックの次の文に制御が移ります。

図11-1 伝播しない例外

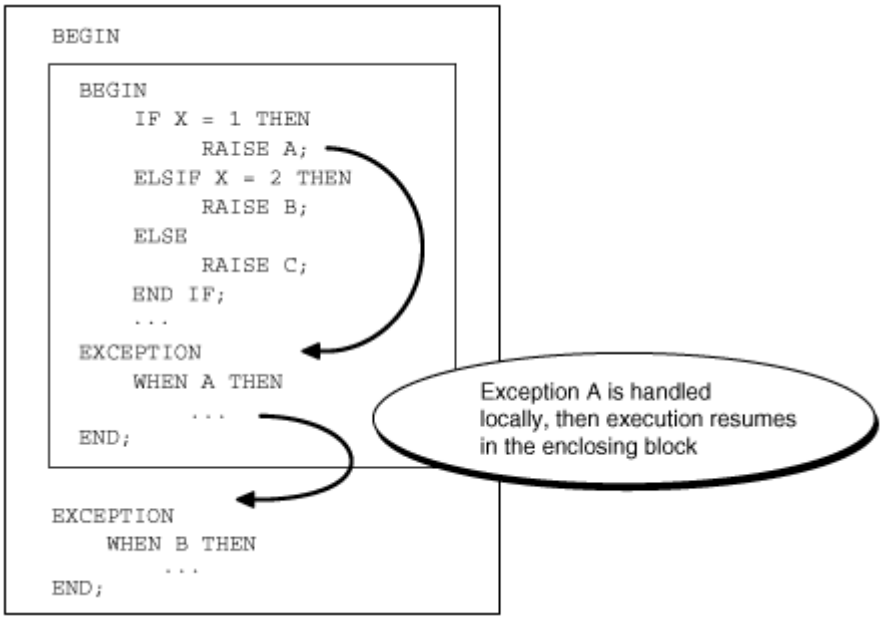


図11-2では、内側のブロックが例外Bを呼び出しています。内側のブロックには例外Bの例外ハンドラがないため、例外Bはその例外ハンドラを持つ外側のブロックに伝播します。例外ハンドラが実行されると、ホスト環境に制御が移ります。

図11-2 内側のブロックから外側のブロックに伝播する例外

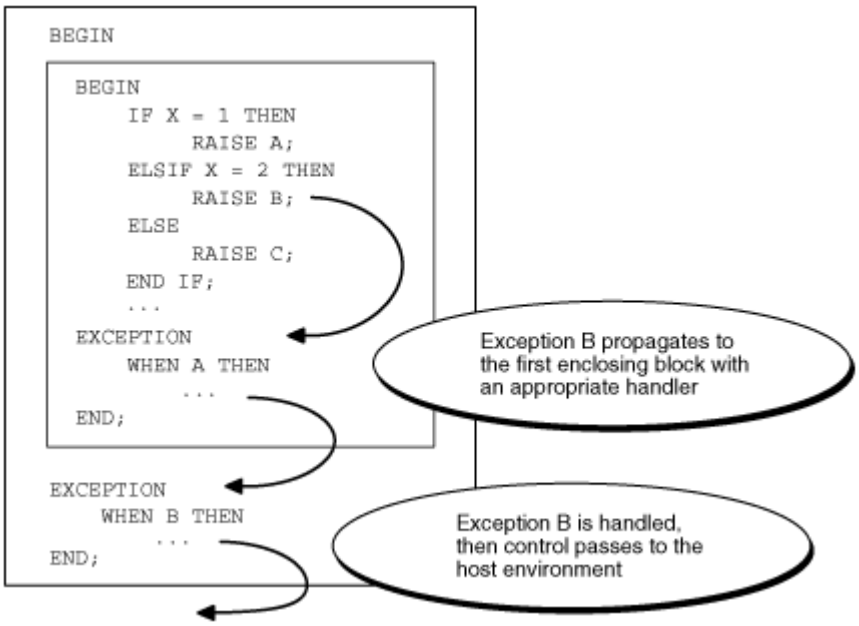
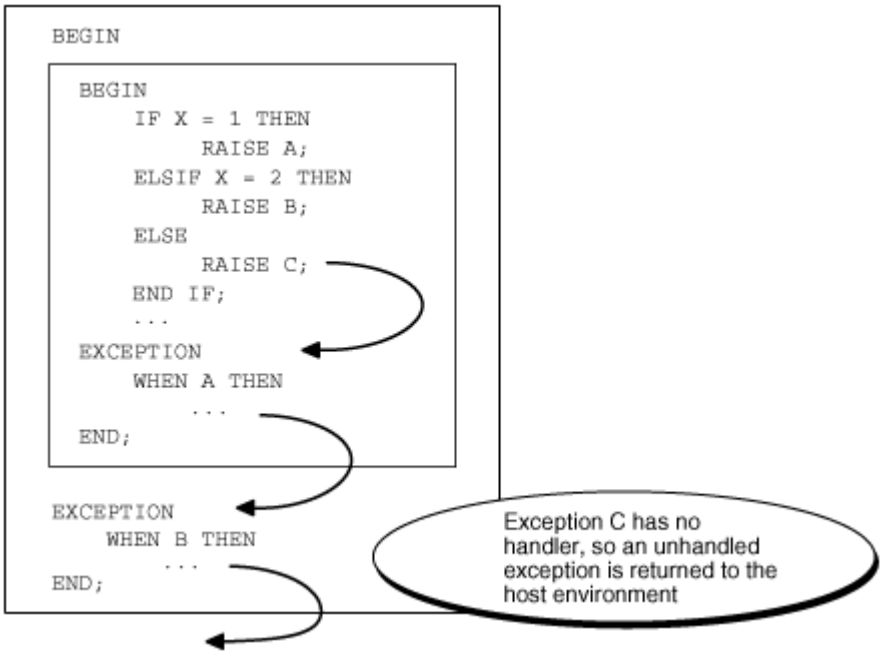


図11-3では、内側のブロックが例外Cを呼び出しています。内側のブロックには例外Cの例外ハンドラがないため、例外Cは外側のブロックに伝播します。外側のブロックにはCの例外ハンドラがないため、PL/SQLはホスト環境に未処理例外エラーを戻します。

図11-3 ホスト環境に未処理例外エラーを戻すPL/SQL



ユーザー定義の例外は有効範囲を超えて(つまり宣言されたブロックを超えたところまで)伝播することがありますが、有効範囲を超えたところには例外名が存在しません。そのため、有効範囲を超えたユーザー定義の例外は、OTHERS例外ハンドラ以外では処理できません。

例11-14の内側のブロックでは、past_dueという名前の例外を宣言していますが、この例外の例外ハンドラはありません。内側のブロックでpast_dueが呼び出されると、外側のブロックに例外が伝播しますが、そこにはpast_dueという名前が存在しません。外側のブロックはOTHERS例外ハンドラを使用して例外を処理します。

ユーザー定義の例外が外側のブロックで処理されない場合は、例11-15に示すとおり、エラーが発生します。

ノート:

例外はリモート・サブプログラム起動には伝播しません。そのため、PL/SQL ブロックは、リモート・サブプログラムによって呼び出された例外を処理できません。

ここでのトピック

- 宣言の中で呼び出された例外の伝播
- 例外ハンドラの中で呼び出された例外の伝播

例11-14 有効範囲を超えて伝播した例外が処理される場合

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
BEGIN
  DECLARE
    past_due     EXCEPTION;
    PRAGMA EXCEPTION_INIT (past_due, -4910);
    due_date     DATE := trunc(SYSDATE) - 1;
    todays_date  DATE := trunc(SYSDATE);
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;
    END IF;
  END;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END;
/
```

例11-15 有効範囲を超えて伝播した例外が処理されない場合

```
BEGIN
  DECLARE
    past_due     EXCEPTION;
    due_date     DATE := trunc(SYSDATE) - 1;
    todays_date  DATE := trunc(SYSDATE);
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;
    END IF;
  END;
END;
/
```

結果:

```
BEGIN
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 9
```

11.8.1 宣言の中で呼び出された例外の伝播

宣言の中で呼び出された例外はただちに外側のブロック(外側のブロックがない場合は起動元またはホスト環境)に伝播します。したがって、例外ハンドラは宣言と同じブロックの中ではなく、外側または起動元のブロックに存在する必要があります。

例11-16では、VALUE_ERRORが呼び出される宣言と同じブロックの中にVALUE_ERROR例外ハンドラが存在しています。例外はただちにホスト環境に伝播するため、この例外は例外ハンドラで処理されません。

例11-17は例11-16と似ていますが、内側のブロックの中の宣言で呼び出されるVALUE_ERROR例外が外側のブロックで処理される点が異なります。

例11-16 宣言の中で呼び出された例外が処理されない場合

```
DECLARE
  credit_limit CONSTANT NUMBER(3) := 5000;  -- Maximum value is 999
BEGIN
  NULL;
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

結果:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 2
```

例11-17 宣言の中で呼び出された例外が外側のブロックで処理される場合

```
BEGIN

  DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;
  BEGIN
    NULL;
  END;

EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

結果:

```
Exception raised in declaration.
```

11.8.2 例外ハンドラの中で呼び出された例外の伝播

例外ハンドラの中で呼び出された例外はただちに外側のブロック(外側のブロックがない場合は起動元またはホスト環境)に伝播

します。したがって、例外ハンドラは、外側または起動元のブロックに存在する必要があります。

例11-18では、nが0(ゼロ)の場合、計算1/nで事前定義の例外ZERO_DIVIDEが呼び出され、同じブロックの中のZERO_DIVIDE例外ハンドラに制御が移ります。例外ハンドラでZERO_DIVIDEが呼び出されると、例外はただちに起動元に伝播します。起動元で例外が処理されないため、PL/SQLはホスト環境に未処理例外エラーを戻します。

例11-19は例11-18に似ていますが、プロシージャが起動元に未処理例外エラーを戻すと起動元でエラーが処理される点が異なります。

例11-20は例11-18と似ていますが、内側のブロックの中の例外ハンドラで呼び出される例外が外側のブロックで処理される点が異なります。

例11-21では、ユーザー定義の例外i_is_oneと事前定義の例外ZERO_DIVIDEに対応する例外ハンドラがプロシージャの例外処理部にあります。i_is_one例外ハンドラでZERO_DIVIDEが呼び出されると、例外はただちに起動元に伝播します(そのため、ZERO_DIVIDE例外ハンドラでは例外が処理されません)。起動元で例外が処理されないため、PL/SQLはホスト環境に未処理例外エラーを戻します。

例11-22は例11-21と似ていますが、i_is_one例外ハンドラで呼び出されるZERO_DIVIDE例外が外側のブロックで処理される点が異なります。

例11-18 例外ハンドラの中で呼び出された例外が処理されない場合

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(1/n);  -- handled
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');  -- not handled
END;
/

BEGIN  -- invoking block
  print_reciprocal(0);
END;
```

結果:

```
Error:
BEGIN
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.PRINT_RECIPROCAL", line 7
ORA-01476: divisor is equal to zero
ORA-06512: at line 2
```

例11-19 例外ハンドラの中で呼び出された例外が起動元で処理される場合

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(1/n);
EXCEPTION
```

```
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN  -- invoking block
  print_reciprocal(0);
EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined.');
END;
/
```

結果:

```
Error:
1/0 is undefined.
```

例11-20 例外ハンドラの中で呼び出された例外が外側のブロックで処理される場合

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN

  BEGIN
    DBMS_OUTPUT.PUT_LINE(1/n);
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error in inner block:');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined.');
  END;

EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT('Error in outer block: ');
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined.');
END;
/

BEGIN
  print_reciprocal(0);
END;
/
```

結果:

```
Error in inner block:
Error in outer block: 1/0 is undefined.
```

例11-21 例外ハンドラの中で呼び出された例外が処理されない場合

```
CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
  i_is_one EXCEPTION;
BEGIN
  i := n;
```

```
  LOOP
    IF i = 1 THEN
      RAISE i_is_one;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
    END IF;

    i := i - 1;
  END LOOP;
EXCEPTION
  WHEN i_is_one THEN
    DBMS_OUTPUT.PUT_LINE('1 is its own reciprocal.');
    DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) ||
                         ' is ' || TO_CHAR(1/(i-1)));

  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN
  descending_reciprocals(3);
END;
/
```

結果:

```
Reciprocal of 3 is .3333333333333333333333333333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
BEGIN
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.DESCENDING_RECIPROCALS", line 19
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 2
```

例11-22 例外ハンドラの中で呼び出された例外が外側のブロックで処理される場合

```
CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
  i_is_one EXCEPTION;
BEGIN

  BEGIN
    i := n;

    LOOP
      IF i = 1 THEN
        RAISE i_is_one;
      ELSE
        DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
      END IF;

      i := i - 1;
    END LOOP;
```

```
  EXCEPTION
    WHEN i_is_one THEN
      DBMS_OUTPUT.PUT_LINE('1 is its own reciprocal.');
      DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) ||
                           ' is ' || TO_CHAR(1/(i-1)));

    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error:');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
  END;

EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined');
END;
/

BEGIN
  descending_reciprocals(3);
END;
/
```

結果:

```
Reciprocal of 3 is .3333333333333333333333333333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
Error:
1/0 is undefined
```

11.9 未処理の例外

発生した例外に対応するハンドラがない場合、PL/SQLは起動元またはホスト環境に未処理例外エラーを戻します。その結果は戻す場所によって異なります。

ストアド・サブプログラムが未処理例外で終了した場合には、PL/SQLはそのサブプログラムによって加えられたデータベースへの変更をロールバックしません。

FORALL文は、VALUES句およびWHERE句の異なる値を使用して、1つのDML文を複数回実行します。1つの値セットで未処理例外が呼び出されると、PL/SQLはこのFORALL文でそれまでに加えられたデータベースへのすべての変更をロールバックします。詳細は、「FORALL例外の即座の処理」および「FORALL文が完了した後のFORALL例外の処理」を参照してください。

ヒント:

すべての PL/SQL プログラムの最上位に OTHERS 例外ハンドラを配置し、未処理例外が発生しないようにします。

11.10 エラー・コードとエラー・メッセージの取得

例外ハンドラの中では、処理中の例外について次のことを実行できます。

- PL/SQLファンクションSQLCODEを使用してエラー・コードを取得できます。このファンクションの詳細は「SQLCODEファンクション」を参照してください。
- 次のいずれかを使用してエラー・メッセージを取得できます。
  - PL/SQLファンクションSQLERRM(詳細は「SQLERRMファンクション」を参照)

    このファンクションは最大512バイトを戻します。これは、Oracle Databaseエラー・メッセージ(エラー・コード、ネストされたメッセージおよび表や列の名前などのメッセージの挿入部分を含む)の最大長です。
  - パッケージ・ファンクションDBMS_UTILITY.FORMAT_ERROR_STACK(詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください)

    このファンクションは最大2000バイトの完全なエラー・スタックを戻します。

  FORALL文をSAVE EXCEPTIONS句とともに使用する場合以外は、例12-13に示すとおり、DBMS_UTILITY.FORMAT_ERROR_STACKを使用することをお薦めします。

SQL文では、SQLCODEまたはSQLERRMを起動できません。これらの値をSQL文で使用するには、例11-23に示すとおり、これらの値をまずローカル変数に代入します。

関連項目:

- サブプログラムがouterスコープの例外ハンドラから呼び出された場合でも、例外が発生した場所のコール・スタックを表示するDBMS_UTILITY.FORMAT_ERROR_BACKTRACE関数の詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。
- UTL_CALL_STACKパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。このパッケージのサブプログラムにより、現在実行中のサブプログラムのサブプログラム名などの情報が提供されます

例11-23 SQLCODEおよびSQLERRMの値の表示

```
DROP TABLE errors;
CREATE TABLE errors (
  code       NUMBER,
  message    VARCHAR2(64)
);
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  name     EMPLOYEES.LAST_NAME%TYPE;
  v_code   NUMBER;
  v_errm   VARCHAR2(64);
BEGIN
  SELECT last_name INTO name
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = -1;
EXCEPTION
  WHEN OTHERS THEN
    v_code := SQLCODE;
    v_errm := SUBSTR(SQLERRM, 1, 64);
    DBMS_OUTPUT.PUT_LINE
```

```
      ('Error code ' || v_code || ': ' || v_errm);

    /* Invoke another procedure,
       declared with PRAGMA AUTONOMOUS_TRANSACTION,
       to insert information about errors. */

    INSERT INTO errors (code, message)
    VALUES (v_code, v_errm);
    RAISE;
END;
/
```

11.11 例外処理後に実行を続ける方法

例外ハンドラが実行されると、外側のブロック(外側のブロックがない場合は起動元またはホスト環境)の次の文に制御が移ります。例外ハンドラは、そのハンドラが存在するブロックに制御を取り戻すことができません。

たとえば、例11-24では、SELECT INTO文でZERO_DIVIDEが呼び出され、例外ハンドラがこれを処理した後に、SELECT INTO文に続くINSERT文から続けて実行することができません。

SELECT INTO文に続くINSERT文から実行を再開する必要がある場合は、例11-25に示すとおり、独自のZERO_DIVIDE例外ハンドラを持つ内側のブロックにSELECT INTO文を配置します。

関連項目:

例12-13では、例外が発生してもバルクSQL操作が継続されます。

例11-24 例外ハンドラの実行と実行の終了

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, salary, commission_pct
  FROM employees;

DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (301, 2500, 0);

  SELECT (salary / commission_pct) INTO sal_calc
  FROM employees_temp
  WHERE employee_id = 301;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
  DBMS_OUTPUT.PUT_LINE('Row inserted.');
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero.');
END;
/
```

結果:

```
Division by zero.
```

例11-25 例外ハンドラの実行と実行の継続

```
DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (301, 2500, 0);

  BEGIN
    SELECT (salary / commission_pct) INTO sal_calc
    FROM employees_temp
    WHERE employee_id = 301;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Substituting 2500 for undefined number.');
      sal_calc := 2500;
  END;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
  DBMS_OUTPUT.PUT_LINE('Enclosing block: Row inserted.');
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Enclosing block: Division by zero.');
END;
/
```

結果:

```
Substituting 2500 for undefined number.
Enclosing block: Row inserted.
```

11.12 例外処理後のトランザクションの再試行

例外を呼び出したトランザクションを例外処理後に再試行するには、次の方法を使用します。

1. 例外処理部を持つサブブロックにトランザクションを入れます。
2. サブブロックの中で、トランザクションを開始する前にセーブポイントをマークします。
3. サブブロックの例外処理部の中で、セーブポイントまでロールバックする例外ハンドラを配置し、問題の修正を試行します。
4. LOOP文の中にサブブロックを配置します。
5. サブブロックの中で、トランザクションを終了させるCOMMIT文の後にEXIT文を配置します。

   トランザクションの実行に成功すると、COMMIT文とEXIT文が実行されます。

   トランザクションの実行に失敗すると、サブプログラムの例外処理部に制御が移り、例外ハンドラが実行されてからループが繰り返されます。

例11-26 例外処理後のトランザクションの再試行

```
DROP TABLE results;
CREATE TABLE results (
```

```
  res_name   VARCHAR(20),
  res_answer VARCHAR2(3)
);

CREATE UNIQUE INDEX res_name_ix ON results (res_name);
INSERT INTO results (res_name, res_answer) VALUES ('SMYTHE', 'YES');
INSERT INTO results (res_name, res_answer) VALUES ('JONES', 'NO');

DECLARE
  name    VARCHAR2(20) := 'SMYTHE';
  answer  VARCHAR2(3) := 'NO';
  suffix  NUMBER := 1;
BEGIN
  FOR i IN 1..5 LOOP  -- Try transaction at most 5 times.

    DBMS_OUTPUT.PUT('Try #' || i);

    BEGIN  -- sub-block begins

       SAVEPOINT start_transaction;

       -- transaction begins

       DELETE FROM results WHERE res_answer = 'NO';

       INSERT INTO results (res_name, res_answer) VALUES (name, answer);

       -- Nonunique name raises DUP_VAL_ON_INDEX.

       -- If transaction succeeded:

       COMMIT;
       DBMS_OUTPUT.PUT_LINE(' succeeded.');
       EXIT;

    EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE(' failed; trying again.');
        ROLLBACK TO start_transaction;    -- Undo changes.
        suffix := suffix + 1;             -- Try to fix problem.
        name := name || TO_CHAR(suffix);
    END;  -- sub-block ends

  END LOOP;
END;
/
```

結果:

```
Try #1 failed; trying again.
Try #2 succeeded.
```

例11-26では、res_nameの値が一意でない場合に事前定義の例外DUP_VAL_ON_INDEXがINSERT文により呼び出されるトランザクションを、前述の方法を使用して再試行しています。

11.13 分散問合せでのエラー処理

分散問合せは、トリガーまたはストアド・サブプログラムを使用して作成できます。この分散問合せは、ローカルのOracle Databaseインスタンスによって、対応する数のリモート問合せに分解されてリモート・ノードに送られます。リモート・ノードはその問合せを実行し、ローカル・ノードにその結果を送ります。その後、ローカル・ノードは必要な後処理を行い、ユーザーまたはアプリケーションに結果を戻します。

たとえば、制約違反のために分散問合せ文の一部でエラーが発生すると、Oracle DatabaseはORA-02055を戻します。後続する文またはサブプログラムの起動は、ロールバック、またはセーブポイントまでのロールバックが入力されるまで、ORA-02067を戻します。

分散更新の一部でエラーが発生したことを示すエラー・メッセージがチェックされるように、アプリケーションを設計してください。エラーを検出した場合、アプリケーションが処理を継続する前に、トランザクション全体をロールバック(またはセーブポイントまでロールバック)してください。

12 PL/SQLの最適化とチューニング

この章では、PL/SQLコンパイラでコードがどのように最適化されるかを説明し、効率的なPL/SQLコードを記述する方法と既存のPL/SQLコードを改善する方法について説明します。

ここでのトピック

- PL/SQLオプティマイザ
- チューニングの候補
- CPUオーバーヘッドの最小化
- バルクSQLおよびバルク・バインド
- 複数変換用のパイプライン・テーブル・ファンクションの連鎖
- 多相テーブル・ファンクションの概要
- 大規模な表のパラレルでの更新
- ユーザー定義の識別子に関するデータの収集
- PL/SQLプログラムのプロファイルおよびトレース
- システム固有の実行のためのPL/SQLユニットのコンパイル

関連項目:

カーソル変数のデメリットの詳細は、『Oracle Database開発ガイド』を参照してください。

12.1 PL/SQLオプティマイザ

Oracle Database 10gリリース1より前のPL/SQLコンパイラは、パフォーマンスの向上のための変更を適用せずに、作成されたソース・テキストをシステム・コードに変換していました。今回のリリースのPL/SQLでは、コードのパフォーマンスが向上するように再調整する機能を持つオプティマイザが使用されます。

このオプティマイザは、デフォルトで使用可能です。まれに、オプティマイザのオーバーヘッドによって、非常に大規模なアプリケーションをコンパイルするときに時間が長くかかる場合がありますが、この場合、(デフォルト値2のかわりに)コンパイル・パラメータPLSQL_OPTIMIZE_LEVEL=1を設定して、最適化レベルを低くすることができます。また、非常にまれですが、PL/SQLによって予測より早く例外が呼び出されたり、例外が1つも呼び出されなくなる場合があります。PLSQL_OPTIMIZE_LEVEL=1に設定すると、コードは再調整されません。

関連項目:

- PLSQL_OPTIMIZE_LEVELコンパイル・パラメータの詳細は、『Oracle Databaseリファレンス』を参照してください。
- PLSQL_OPTIMIZE_LEVELコンパイル・パラメータの変更例については、『Oracle Database開発ガイド』を参照してく

ださい。

- 静的ディクショナリ・ビューALL_PLSQL_OBJECT_SETTINGSの詳細は、『Oracle Databaseリファレンス』を参照してください。

12.1.1 サブプログラムのインライン化

コンパイラが実行できる最適化の1つとして、サブプログラムのインライン化があります。

サブプログラムのインライン化によって、サブプログラムの起動は、起動先サブプログラムのコピーに置き換えられます(起動先と起動元のサブプログラムが同じプログラム・ユニット内に存在する場合)。サブプログラムのインライン化を実行できるようにするには、PLSQL_OPTIMIZE_LEVELコンパイル・パラメータのデフォルト値(2)を受け入れるか、またはこのパラメータを3に設定します。

PLSQL_OPTIMIZE_LEVEL=2の場合、次のようにINLINEプラグマを使用して、インライン化する各サブプログラムを指定する必要があります。

```
PRAGMA INLINE (subprogram, 'YES')
```

subprogramがオーバーロードされた場合、前述のプラグマは、指定された名前を持つすべてのサブプログラムに適用されます。

PLSQL_OPTIMIZE_LEVEL=3の場合、PL/SQLコンパイラは、サブプログラムをインライン化する機会を探ります。インライン化するサブプログラムをユーザーが指定する必要はありません。ただし、(前述の構文に従って)INLINEプラグマを使用し、サブプログラムのインライン化に高い優先順位を指定することができ、その後、他の考慮事項または制限のためインライン化が望ましくない場合を除き、コンパイラによってサブプログラムがインライン化されます。

特定のサブプログラムがインライン化されている場合は、ほぼ常にパフォーマンスが向上します。ただし、コンパイラは最適化プロセスの初期の段階でサブプログラムをインライン化するため、サブプログラムのインライン化によって、後の段階でのより強力な最適化が妨げられる可能性があります。

サブプログラムのインライン化によって特定のPL/SQLプログラムのパフォーマンスが低下する場合は、PL/SQL階層型プロファイラを使用して、インライン化を無効にするサブプログラムを識別します。サブプログラムのインライン化を無効にするには、INLINEプラグマを次のように使用します。

```
PRAGMA INLINE (subprogram, 'NO')
```

INLINEプラグマは、直後にある宣言または文および一部の種類の文にのみ影響します。

INLINEプラグマが宣言の直前にある場合は、次のものに影響します。

- その宣言に指定されたサブプログラムのすべての起動
- その宣言内のすべての初期化値(レコードのデフォルトの初期化値を除く)

INLINEプラグマは、次に示すいずれかの文の直前にある場合、その文に指定されたサブプログラムのすべての起動に影響します。

- 代入
- CALL
- 条件付き
- CASE

- CONTINUE WHEN
- EXECUTE IMMEDIATE
- EXIT WHEN
- LOOP
- RETURN

INLINEプラグマは、前述のリストにない文には影響しません。

複数のプラグマが、同じ宣言または文に影響する場合があります。各プラグマは、文に対して独自の影響を及ぼします。PRAGMA INLINE(subprogram,'YES')およびPRAGMA INLINE(identifier,'NO')に同じsubprogramが含まれている場合、'NO'は'YES'をオーバーライドします。1つのPRAGMA INLINE(subprogram,'NO')は、任意の数のPRAGMA INLINE(subprogram,'YES')をオーバーライドしますが、これらのプラグマの順序は重要ではありません。

関連項目:

- PL/SQL階層プロファイラの詳細は、『Oracle Database開発ガイド』を参照してください
- PLSQL_OPTIMIZE_LEVELコンパイル・パラメータの詳細は、『Oracle Databaseリファレンス』を参照してください。
- 静的ディクショナリ・ビューALL_PLSQL_OBJECT_SETTINGSの詳細は、『Oracle Databaseリファレンス』を参照してください。

例12-1 サブプログラムのインライン化の指定

この例では、PLSQL_OPTIMIZE_LEVEL=2の場合、INLINEプラグマは、プロシージャの起動p1(1)およびp1(2)には影響しますが、プロシージャの起動p1(3)およびp1(4)には影響しません。

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 are not inlined
...
```

例12-2 オーバーロードされたサブプログラムのインライン化の指定

この例では、PLSQL_OPTIMIZE_LEVEL=2の場合、INLINEプラグマは、p2という名前の両方のファンクションに影響します。

```
FUNCTION p2 (p boolean) return PLS_INTEGER IS ...
FUNCTION p2 (x PLS_INTEGER) return PLS_INTEGER IS ...
...
PRAGMA INLINE(p2, 'YES');
x := p2(true) + p2(3);


...
```

例12-3 サブプログラムの非インライン化の指定

この例では、INLINEプラグマは、プロシージャの起動p1(1)およびp1(2)には影響しますが、プロシージャの起動p1(3)およびp1(4)には影響しません。

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'NO');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are not inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 might be inlined
...
```

例12-4 PRAGMA INLINE ... 'NO'によるPRAGMA INLINE ... 'YES'のオーバーライド

この例では、2番目のINLINEプラグマは1番目と3番目のINLINEプラグマをオーバーライドします。

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
PRAGMA INLINE (p1, 'NO');
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are not inlined
...
```

12.2 チューニングの候補

次のようなPL/SQLコードは、ほぼ確実にチューニングのメリットを得ることができます。

- 新しいPL/SQL言語機能を利用していない古いコード。

  ヒント:

  古いコードのチューニングに取りかかる前に、現在のシステムのベンチマークを取り、プログラムによって起動される古いサブプログラムをプロファイルします(「PL/SQL プログラムのプロファイルおよびトレース」を参照)。PL/SQL オプティマイザ(「PL/SQL オプティマイザ」を参照)の各種の自動最適化機能により、チューニングの実行前にパフォーマンスが改善する場合があります。

- DBMS_SQLパッケージを使用して記述された古い動的SQL文。

  コンパイル時に動的SQL文の入力変数と出力変数の数およびデータ型がわかる場合、その文をシステム固有の動的SQLで記述しなおすことができます(システム固有の動的SQLは、DBMS_SQLパッケージを使用する同等のコードより著しく高速に実行されます(特に、コードをコンパイラによって最適化できる場合))。詳細は、「PL/SQLの動的SQL」を参照してください。

- SQL文の処理に長時間を費やすコード。

  「SQL文のチューニング」を参照してください。

- 問合せで起動され、数百万回実行される可能性があるファンクション。

  「問合せにおけるファンクションの起動のチューニング」を参照してください。

- 問合せ結果のループに長時間を費やすコード。

  ループのチューニングを参照してください。

- 数値計算を大量に実行するコード。

「計算集中型PL/SQLコードのチューニング」を参照してください。

- PL/SQLからSQLに直接渡されるデータベース定義言語(DDL)文を発行するのではなく、PL/SQL文の処理に長時間を費やすコード。

  「システム固有の実行のためのPL/SQLユニットのコンパイル」を参照してください。

12.3 CPUオーバーヘッドの最小化

ここでのトピック

- SQL文のチューニング
- 問合せにおけるファンクションの起動のチューニング
- サブプログラムの起動のチューニング
- ループのチューニング
- 計算集中型PL/SQLコードのチューニング
- SQL文字関数の使用
- 最低コストの条件テストの先頭への配置

12.3.1 SQL文のチューニング

PL/SQLプログラムの速度が低下する最も一般的な原因は、速度の遅いSQL文です。PL/SQLプログラムのSQL文をできるだけ効率的にするには:

- 適切な索引を使用します。

  詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。

- 問合せヒントを使用して、不要な全表スキャンを回避します。

  詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

- DBMS_STATSパッケージのサブプログラムを使用して、すべての表について現在の統計を収集します。

  詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。

- 次のものを使用して、実行計画およびSQL文のパフォーマンスを分析します。

  - EXPLAIN PLAN文

    詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。

  - TKPROFユーティリティを持つSQLトレース機能

    詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。

- バルクSQL(PL/SQLとSQL間の通信のパフォーマンス・オーバーヘッドを最小化するPL/SQL機能のセット)を使用します。

  詳細は、「バルクSQLおよびバルク・バインド」を参照してください。

12.3.2 問合せにおけるファンクションの起動のチューニング

問合せで起動されるファンクションは、数百万回実行される可能性があります。問合せで不必要にファンクションを起動しないようにして、起動はできるだけ効率的にします。

問合せの表にファンクション索引を作成してください。CREATE INDEX文には少し時間がかかる場合もありますが、各行のファンクション値がキャッシュされるため、問合せを非常に高速に実行できます。

問合せによって列をファンクションに渡す場合、問合せではその列でユーザー作成索引を使用できないため、表(非常に大きい可能性がある)の行ごとにファンクションが起動される可能性があります。ファンクションの起動数を最小限に抑えるには、ネストした問合せを使用します。内側の問合せで結果セットが少ない行数になるようにフィルタリングし、外側の問合せでそれらの行のみを対象にファンクションを起動します。

関連項目:

- CREATE INDEX文の構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください
- PL/SQLファンクションの結果をキャッシュする方法の詳細は、「PL/SQLファンクション結果キャッシュ」を参照してください

例12-5 ネストした問合せによるパフォーマンスの改善

この例では、2つの問合せで同じ結果セットを生成しますが、2番目の問合せは1番目の問合せより効率的です。(この例では、EMPLOYEES表が非常に小さいため、回数および時間の違いはごくわずかです。非常に大きい表では、これらの違いが顕著になります。)

```
DECLARE
  starting_time  TIMESTAMP WITH TIME ZONE;
  ending_time    TIMESTAMP WITH TIME ZONE;
BEGIN
  -- Invokes SQRT for every row of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
    SELECT DISTINCT(SQRT(department_id)) col_alias
    FROM employees
    ORDER BY col_alias
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;

  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));

  -- Invokes SQRT for every distinct department_id of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
```

```
    SELECT SQRT(department_id) col_alias
    FROM (SELECT DISTINCT department_id FROM employees)
    ORDER BY col_alias
  )
  LOOP
    IF item.col_alias IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
    END IF;
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;

  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));
END;
/
```

結果は次のようになります。

```
Square root of dept. ID = 3.162277660168379331998893544432718533 72
Square root of dept. ID = 4.472135954999579392818347337462552470 88
Square root of dept. ID = 5.477225575051661134569697828008021339 53
Square root of dept. ID = 6.324555320336758663997787088865437067 44
Square root of dept. ID = 7.071067811865475244008443621048490392 85
Square root of dept. ID = 7.745966692414833770358530799564799221 67
Square root of dept. ID = 8.366600265340755479781720257851874893 93
Square root of dept. ID = 8.944271909999158785636694674925104941 76
Square root of dept. ID = 9.486832980505137995996680633298155601 16
Square root of dept. ID = 10
Square root of dept. ID = 10.488088481701515469914535136799375984 75
Time = +000000000 00:00:00.046000000
Square root of dept. ID = 3.162277660168379331998893544432718533 72
Square root of dept. ID = 4.472135954999579392818347337462552470 88
Square root of dept. ID = 5.477225575051661134569697828008021339 53
Square root of dept. ID = 6.324555320336758663997787088865437067 44
Square root of dept. ID = 7.071067811865475244008443621048490392 85
Square root of dept. ID = 7.745966692414833770358530799564799221 67
Square root of dept. ID = 8.366600265340755479781720257851874893 93
Square root of dept. ID = 8.944271909999158785636694674925104941 76
Square root of dept. ID = 9.486832980505137995996680633298155601 16
Square root of dept. ID = 10
Square root of dept. ID = 10.488088481701515469914535136799375984 75
Time = +000000000 00:00:00.000000000
```

12.3.3 サブプログラムの起動のチューニング

サブプログラムにOUTまたはIN OUTパラメータがある場合、NOCOPYヒントを使用してこれらのパラメータを宣言することによって、起動のオーバーヘッドを減らすことができます。

OUTパラメータまたはIN OUTパラメータが、コレクション、レコード、ADTのインスタンスなどの大規模なデータ構造を表す場合、それらのコピーによって実行速度が低下し、(特にADTのインスタンスで)メモリー使用量が増大します。

ADTメソッドが起動されるたびに、PL/SQLによってそのADTのすべての属性がコピーされます。メソッドが正常に終了すると、そのメソッドが属性に対して行った変更が、PL/SQLによって適用されます。メソッドが未処理例外を戻して終了すると、PL/SQLは、属性を変更しません。

プログラムで、サブプログラムが未処理例外を戻して終了したときにOUTパラメータまたはIN OUTパラメータに起動前の値を保持する必要がない場合、パラメータ宣言にNOCOPYヒントを含めます。NOCOPYヒントは、コンパイラが対応する実パラメータを値ではなく参照によって渡すように要求しますが、その保証はしません。

> 注意:
>
> サブプログラムが未処理例外を戻して終了したときに実パラメータまたは ADT 属性で起動前の値を確実に保持するためには、NOCOPY に依存しないでください(特定の起動について、コンパイラはこのヒントに従う場合と従わない場合があります)。かわりに、サブプログラムですべての例外が処理されるようにします。

関連項目:

- NOCOPYヒントの詳細は、「NOCOPY」を参照してください
- ADTのメンバー・メソッドとともにNOCOPYを使用する方法は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください

例12-6 NOCOPYサブプログラム・パラメータ

この例では、コンパイラがdo_nothing2の起動に関してNOCOPYヒントに従うと、do_nothing2の起動がdo_nothing1の起動より高速に実行されます。

```
DECLARE
  TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
  emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
  t1 NUMBER;
  t2 NUMBER;
  t3 NUMBER;
  PROCEDURE get_time (t OUT NUMBER) IS
  BEGIN
    t := DBMS_UTILITY.get_time;
  END;
  PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
  BEGIN
    NULL;
  END;
  PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
  BEGIN
    NULL;
  END;
BEGIN
  SELECT * INTO emp_tab(1)
  FROM employees
  WHERE employee_id = 100;
  emp_tab.EXTEND(49999, 1);  -- Copy element 1 into 2..50000
  get_time(t1);
  do_nothing1(emp_tab);  -- Pass IN OUT parameter
  get_time(t2);
  do_nothing2(emp_tab);  -- Pass IN OUT NOCOPY parameter
  get_time(t3);
  DBMS_OUTPUT.PUT_LINE ('Call Duration (secs)');
```

```
  DBMS_OUTPUT.PUT_LINE ('--------------------');
  DBMS_OUTPUT.PUT_LINE ('Just IN OUT: ' || TO_CHAR((t2 - t1)/100.0));
  DBMS_OUTPUT.PUT_LINE ('With NOCOPY: ' || TO_CHAR((t3 - t2))/100.0);
END;
/
```

12.3.4 ループのチューニング

PL/SQLアプリケーションはループの周囲に構築されることが多いため、ループ自体とループ内のコードの両方を最適化することが重要です。

結果セットを2回以上ループしたり、結果セットのループ時に他の問合せを発行する必要がある場合、元の問合せを変更して、目的の結果を正確に取得することが可能な場合があります。複数の問合せを結合できるSQLの集合演算子を調査してください(『Oracle Database SQL言語リファレンス』を参照)。

副問合せを使用して、複数の段階でフィルタリングおよびソートを行うこともできます(「副問合せによる問合せ結果セットの処理」を参照)。

関連項目:

「バルクSQLおよびバルク・バインド」

12.3.5 計算集中型PL/SQLコードのチューニング

ここでの推奨事項は、特に計算集中型PL/SQLコードに適用されます(ただし、それらのコードに限定されません)。

ここでのトピック

- ハードウェア算術を使用するデータ型の使用
- パフォーマンスを重視するコードでの制約付きサブタイプの回避
- 暗黙的なデータ型変換の最少化

12.3.5.1 ハードウェア算術を使用するデータ型の使用

NUMBERデータ型ファミリ(「NUMBERデータ型ファミリ」を参照)のデータ型を使用することは避けてください。これらのデータ型の内部的な表現形式は、パフォーマンスよりも移植性および任意の位取りと精度に重点を置いて設計されています。これらの型のデータに対する操作では、ライブラリ算術計算が使用されますが、PLS_INTEGER型、BINARY_FLOAT型およびBINARY_DOUBLE型のデータに対する操作では、ハードウェア算術計算が使用されます。

ローカル整変数には、PLS_INTEGERを使用します(「PLS_INTEGERおよびBINARY_INTEGERデータ型」を参照)。パフォーマンスを重視するコードで使用する変数(NULL値を持つことがなく、オーバーフロー・チェックを必要としないもの)には、SIMPLE_INTEGERを使用します(PLS_INTEGERのSIMPLE_INTEGERサブタイプを参照)。

浮動小数点変数には、BINARY_FLOATまたはBINARY_DOUBLEを使用します(『Oracle Database SQL言語リファレンス』を参照)。パフォーマンスを重視するコードで使用する変数(NULL値を持つことがなく、オーバーフロー・チェックを必要としないもの)には、SIMPLE_FLOATまたはSIMPLE_DOUBLEを使用します(「BINARY_FLOATおよびBINARY_DOUBLEの追加の

PL/SQLサブタイプ」を参照)。

ノート:

BINARY_FLOAT および BINARY_DOUBLE とそれらのサブタイプは、正確さが非常に重要な財務処理コードにはあまり適していません。これらの型は、常に正確に小数値を表すわけではなく、NUMBER 型とは異なり四捨五入処理が行われるためです。

多くのSQL数値ファンクション(『Oracle Database SQL言語リファレンス』を参照)は、BINARY_FLOATパラメータおよびBINARY_DOUBLEパラメータを受け入れるバージョンでオーバーロードされます。これらのデータ型の変数をこのようなファンクションに渡したり、このようなファンクションに式を渡すときに変換ファンクションのTO_BINARY_FLOAT(『Oracle Database SQL言語リファレンス』を参照)およびTO_BINARY_DOUBLE(『Oracle Database SQL言語リファレンス』を参照)を起動することによって、計算集中型コードをスピードアップできます。

12.3.5.2 パフォーマンスを重視するコードでの制約付きサブタイプの回避

パフォーマンスを重視するコードでは、制約付きサブタイプ(「制約付きサブタイプ」を参照)を使用することは避けてください。制約付きサブタイプの変数またはパラメータに代入を行うたびに、代入される値が制約に違反していないことを保証するために実行時に余分なチェックが必要になります。

関連項目:

「PL/SQLの事前定義のデータ型」(事前定義の制約付きサブタイプを含む)

12.3.5.3 暗黙的なデータ型変換の最少化

PL/SQLは実行時に、必要に応じて異なるデータ型を暗黙的(自動的)に変換します。たとえば、PLS_INTEGER変数をNUMBER変数に代入すると、PL/SQLによって、PLS_INTEGER値がNUMBER値に変換されます(各値の内部表現は異なるため)。

可能であれば、暗黙的な変換は常に最小限に抑えてください。たとえば:

- 表の列に変数を挿入する場合、または表の列から変数に値を代入する場合、その変数に表の列と同じデータ型を指定します。

  ヒント:

  %TYPE 属性を使用して変数を宣言します(「%TYPE 属性」を参照)。

- 各リテラルが、割当て先の変数または記述対象の式と同じデータ型になるようにします。
- SQLデータ型からPL/SQLデータ型に値を変換し、変換した値を式で使用します。

  たとえば、NUMBER値をPLS_INTEGER値に変換し、そのPLS_INTEGER値を式で使用します。PLS_INTEGER演算はハードウェア算術計算を使用するため、ライブラリ算術計算を使用するNUMBER演算より処理速度が速くなります。PLS_INTEGERデータ型の詳細は、「PLS_INTEGERおよびBINARY_INTEGERデータ型」を参照してください。

- あるSQLデータ型の値を別のSQLデータ型の変数に代入する前に、SQL変換ファンクションを使用してソース値をターゲット・データ型に明示的に変換します(SQL変換ファンクションの詳細は、『Oracle Database SQL言語リファレンス』を参照してください)。
- 異なるデータ型のパラメータを受け入れるバージョンでサブプログラムをオーバーロードし、そのパラメータ型の各バージョンを最適化します。オーバーロードされたサブプログラムの詳細は、「オーバーロードされたサブプログラム」を参照してください。

関連項目:

- SQLデータ型(PL/SQLデータ型にも相当)の暗黙的な変換の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- 「同じデータ型ファミリ内のベース型を使用したサブタイプ」

12.3.6 SQL文字関数の使用

SQLには、PL/SQLコードよりも効率的な低レベルのコードを使用する高度に最適化された文字ファンクションが多くあります。PL/SQLコードを記述するかわりにこれらのファンクションを使用することで、同じ処理を実行できます。

参照:

- 文字値を戻す SQL 文字関数の詳細は、『Oracle Database SQL 言語リファレンス』を参照してください。
- NLS 文字値を戻す SQL 文字関数の詳細は、『Oracle Database SQL 言語リファレンス』を参照してください。
- 数値を戻す SQL 文字関数の詳細は、『Oracle Database SQL 言語リファレンス』を参照してください。
- SQL 文字ファンクション REGEXP_LIKE を使用する PL/SQL コードの例は、例 6-6 を参照してください。

12.3.7 最低コストの条件テストの先頭への配置

PL/SQLは、結果が判別できた時点でただちに論理式の評価を停止します。可能であれば常に、最低コストの条件を論理式の最初に配置して評価することで、この短絡評価を利用してください。たとえば、変数テストに失敗した場合、PL/SQLでファンクションを起動する必要がなくなるように、ファンクションの戻り値をテストする前に、PL/SQL変数の値をテストします。

```
IF boolean_variable OR (number > 10) OR boolean_function(parameter) THEN ...
```

関連項目:

12.4 バルクSQLおよびバルク・バインド

バルクSQLは、PL/SQLとSQL間の通信のパフォーマンス・オーバーヘッドを最小化します。バルクSQLを構成するPL/SQL機能は、FORALL文およびBULK COLLECT句です。SQL文に指定されたPL/SQL変数に値を代入することを、バインドと呼びます。

PL/SQLとSQLは、次のように通信します。PL/SQLエンジンは、SELECT INTO文またはDML文を実行するために、SQLエンジンに問合せまたはDML文を送信します。SQLエンジンは、問合せまたはDML文を実行し、結果をPL/SQLエンジンに戻します。

FORALL文は、DML文をPL/SQLからSQLに1文ずつではなくバッチで送信します。BULK COLLECT句は、結果をSQLからPL/SQLに1つずつではなくバッチで戻します。問合せまたはDML文が4つ以上のデータベース行に影響する場合は、バルクSQLでパフォーマンスを大幅に向上できます。

ノート:

リモート表では、バルク SQL は実行できません。

PL/SQLバインド操作は、次のカテゴリに分類されます。

バインド・カテゴリ	このバインドが実行される場合
インバインド	INSERT 文、UPDATE 文または MERGE によって PL/SQL 変数またはホスト変数がデータベースに格納される場合
アウトバインド	INSERT 文、UPDATE 文または DELETE 文の RETURNING INTO 句によって PL/SQL 変数またはホスト変数にデータベースの値が代入される場合
DEFINE	SELECT 文または FETCH 文によって PL/SQL 変数またはホスト変数にデータベースの値が代入される場合

インバインドおよびアウトバインドでは、バルクSQLはバルク・バインドを使用します(つまり、値のコレクション全体を一度にバインドします)。n個の要素があるコレクションの場合、バルクSQLは単一の操作で、n回分のSELECT INTO文またはDML文に相当する処理を実行できます。バルクSQLを使用する問合せでは、行ごとにFETCH文を使用することなく、任意の数の行を戻すことができます。

ノート:

パラレル DML は、バルク SQL を使用すると無効になります。

ここでのトピック

- FORALL文

- BULK COLLECT句
- FORALL文とBULK COLLECT句の併用
- ホスト配列のクライアント・バルク・バインド

12.4.1 FORALL文

FORALL文は、バルクSQLの機能であり、DML文をPL/SQLからSQLに1文ずつではなくバッチで送信します。

FORALL文を理解するため、最初に例12-7のFOR LOOP文について考えてみます。これらのDML文は、PL/SQLからSQLに1文ずつ送信されます。

```
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

次に、例12-8のFORALL文について考えてみます。3つの同じDML文がPL/SQLからSQLにバッチとして送信されます。

通常、FORALL文は、同等のFOR LOOP文よりはるかに高速です。ただし、FOR LOOP文には複数のDML文を含めることができますが、FORALL文に含めることができるDML文は1つのみです。FORALL文によってSQLに送信されるDML文のバッチは、VALUES句とWHERE句のみが異なります。これらの句の値は、データが移入された既存のコレクションから取得される必要があります。

> ノート:
>
> FORALL 文の DML 文では、複数のコレクションを参照できますが、パフォーマンス上のメリットは、FORALL の索引変数を索引として使用するコレクション参照に対してのみ適用されます。

例12-9では、2つのデータベース表に同じコレクション要素を挿入します(1番目の表にはFOR LOOP文を使用し、2番目の表にはFORALL文を使用して、各文の実行にかかる時間を表示します)。(時間は実行ごとに異なります。)

例12-10では、FORALL文がコレクションのサブセットに適用されます。

ここでのトピック

- 疎コレクションに対するFORALL文の使用
- FORALL文での未処理例外
- FORALL例外の即時処理
- FORALL文が完了した後のFORALL例外の処理
- FORALL文の影響を受ける行の数の取得

関連項目:

- 制限などのFORALL文の構文およびセマンティクスの詳細は、「FORALL文」を参照してください
- 一般的な暗黙カーソル属性と、FORALL文で使用できる他の暗黙カーソル属性の詳細は、「暗黙カーソル」を参照してください

例12-7 FOR LOOP文のDELETE文

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FOR i IN depts.FIRST..depts.LAST LOOP
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
  END LOOP;
END;
/
```

例12-8 FORALL文のDELETE文

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FORALL i IN depts.FIRST..depts.LAST
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
END;
/
```

例12-9 FOR LOOP文およびFORALL文におけるINSERT文の時間の違い

```
DROP TABLE parts1;
CREATE TABLE parts1 (
  pnum INTEGER,
  pname VARCHAR2(15)
);

DROP TABLE parts2;
CREATE TABLE parts2 (
  pnum INTEGER,
  pname VARCHAR2(15)
);
DECLARE
  TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
  pnums   NumTab;
  pnames  NameTab;
  iterations  CONSTANT PLS_INTEGER := 50000;
  t1  INTEGER;
  t2  INTEGER;
  t3  INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- populate collections
    pnums(j) := j;
    pnames(j) := 'Part No. ' || TO_CHAR(j);
  END LOOP;
  t1 := DBMS_UTILITY.get_time;
  FOR i IN 1..iterations LOOP
    INSERT INTO parts1 (pnum, pname)
```

```
    VALUES (pnums(i), pnames(i));
  END LOOP;
  t2 := DBMS_UTILITY.get_time;
  FORALL i IN 1..iterations
    INSERT INTO parts2 (pnum, pname)
    VALUES (pnums(i), pnames(i));
  t3 := DBMS_UTILITY.get_time;
  DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
  DBMS_OUTPUT.PUT_LINE('---------------------');
  DBMS_OUTPUT.PUT_LINE('FOR LOOP: ' || TO_CHAR((t2 - t1)/100));
  DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR((t3 - t2)/100));
  COMMIT;
END;
/
```

結果は次のようになります。

```
Execution Time (secs)
---------------------
FOR LOOP: 5.97
FORALL:   .07

PL/SQL procedure successfully completed.
```

例12-10 コレクションのサブセットに対するFORALL文

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
  TYPE NumList IS VARRAY(10) OF NUMBER;
  depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
  FORALL j IN 4..7
    DELETE FROM employees_temp WHERE department_id = depts(j);
END;
/
```

12.4.1.1 疎コレクションに対するFORALL文の使用

FORALL文の境界句で疎コレクションを参照する場合、INDICES OF句またはVALUES OF句を使用して既存の索引値のみを指定します。

INDICES OFは、文字列で索引付けされている連想配列を除く任意のコレクションに使用できます。VALUES OFは、PLS_INTEGERで索引付けされているPLS_INTEGER要素のコレクションにのみ使用できます。

PLS_INTEGERで索引付けされているPLS_INTEGER要素のコレクションには、索引コレクションを使用できます。索引コレクションは、他のコレクションの要素に対するポインタのコレクションです(索引付けされたコレクション)。

索引コレクションは、異なるFORALL文で同じコレクションの異なるサブセットを処理する場合に便利です。時間とメモリーを大量に費やす可能性のあるサブセットを表す新しいコレクションに元のコレクションの要素をコピーするかわりに、索引コレクションを使用して各サブセットを表し、各索引コレクションを異なるFORALL文のVALUES OF句で使用します。

関連項目:

「疎コレクションおよびSQL%BULK_EXCEPTIONS」

例12-11 疎コレクションおよびそのサブセットに対するFORALL文

この例では、INDICES OF句付きでFORALL文を使用し、表に疎コレクションの要素を移入します。その後、VALUES OF句付きで2つのFORALL文を使用し、2つの表にコレクションのサブセットを入れます。

```
DROP TABLE valid_orders;
CREATE TABLE valid_orders (
  cust_name  VARCHAR2(32),
  amount     NUMBER(10,2)
);

DROP TABLE big_orders;
CREATE TABLE big_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DROP TABLE rejected_orders;
CREATE TABLE rejected_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DECLARE
  SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
  TYPE cust_typ IS TABLE OF cust_name;
  cust_tab  cust_typ;  -- Collection of customer names

  SUBTYPE order_amount IS valid_orders.amount%TYPE;
  TYPE amount_typ IS TABLE OF NUMBER;
  amount_tab  amount_typ;  -- Collection of order amounts

  TYPE index_pointer_t IS TABLE OF PLS_INTEGER;

  /* Collections for pointers to elements of cust_tab collection
     (to represent two subsets of cust_tab): */

  big_order_tab       index_pointer_t := index_pointer_t();
  rejected_order_tab  index_pointer_t := index_pointer_t();

  PROCEDURE populate_data_collections IS
  BEGIN
    cust_tab := cust_typ(
      'Company1','Company2','Company3','Company4','Company5'
    );

    amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
  END;

BEGIN
  populate_data_collections;

  DBMS_OUTPUT.PUT_LINE ('--- Original order data ---');

  FOR i IN 1..cust_tab.LAST LOOP
```

```
  DBMS_OUTPUT.PUT_LINE (
    'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
  );
END LOOP;

-- Delete invalid orders:

FOR i IN 1..cust_tab.LAST LOOP
  IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
    cust_tab.delete(i);
    amount_tab.delete(i);
  END IF;
END LOOP;

-- cust_tab is now a sparse collection.

DBMS_OUTPUT.PUT_LINE ('--- Order data with invalid orders deleted ---');

FOR i IN 1..cust_tab.LAST LOOP
  IF cust_tab.EXISTS(i) THEN
    DBMS_OUTPUT.PUT_LINE (
      'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
    );
  END IF;
END LOOP;

-- Using sparse collection, populate valid_orders table:

FORALL i IN INDICES OF cust_tab
  INSERT INTO valid_orders (cust_name, amount)
  VALUES (cust_tab(i), amount_tab(i));

populate_data_collections;  -- Restore original order data

-- cust_tab is a dense collection again.

/* Populate collections of pointers to elements of cust_tab collection
   (which represent two subsets of cust_tab): */

FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
  IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
    rejected_order_tab.EXTEND;
    rejected_order_tab(rejected_order_tab.LAST) := i;
  END IF;

  IF amount_tab(i) > 2000 THEN
    big_order_tab.EXTEND;
    big_order_tab(big_order_tab.LAST) := i;
  END IF;
END LOOP;

/* Using each subset in a different FORALL statement,
   populate rejected_orders and big_orders tables: */

FORALL i IN VALUES OF rejected_order_tab
  INSERT INTO rejected_orders (cust_name, amount)
  VALUES (cust_tab(i), amount_tab(i));
```

```
  FORALL i IN VALUES OF big_order_tab
    INSERT INTO big_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));
END;
/
```

結果:

```
--- Original order data ---
Customer #1, Company1: $5000.01
Customer #2, Company2: $0
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
Customer #5, Company5: $
--- Data with invalid orders deleted ---
Customer #1, Company1: $5000.01
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
```

正しい注文詳細が格納されたかどうかの検証:

```
SELECT cust_name "Customer", amount "Valid order amount"
FROM valid_orders
ORDER BY cust_name;
```

結果:

```
Customer                         Valid order amount
-------------------------------- ------------------
Company1                                    5000.01
Company3                                     150.25
Company4                                       4000

3 rows selected.
```

問合せ:

```
SELECT cust_name "Customer", amount "Big order amount"
FROM big_orders
ORDER BY cust_name;
```

結果:

```
Customer                         Big order amount
-------------------------------- ----------------
Company1                                 5000.01
Company4                                    4000

2 rows selected.
```

問合せ:

```
SELECT cust_name "Customer", amount "Rejected order amount"
FROM rejected_orders
ORDER BY cust_name;
```

結果:

```
Customer                         Rejected order amount
-------------------------------- ----------------------
Company2                                              0
Company5

2 rows selected.
```

12.4.1.2 FORALL文での未処理例外

SAVE EXCEPTIONS句なしのFORALL文で、いずれかのDML文によって未処理例外が呼び出されると、PL/SQLによってFORALL文が停止され、前のDML文で行われたすべての変更がロールバックされます。

たとえば、例12-8のFORALL文は、次のDML文を(そのいずれかによって未処理例外が呼び出されないかぎり)記載された順序で実行します。

```
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

3番目の文で未処理例外が呼び出されると、PL/SQLによって、1番目と2番目の文による変更がロールバックされます。2番目の文で未処理例外が呼び出されると、PL/SQLによって、1番目の文による変更がロールバックされ、3番目の文は実行されません。

FORALL文で呼び出される例外は、次のいずれかの方法で処理できます。

- 各例外が呼び出された時点で(「FORALL例外の即座の処理」を参照)
- FORALL文の実行が完了した後、SAVE EXCEPTIONS句を含めることによって(「FORALL文完了後のFORALL例外の処理」を参照)

12.4.1.3 FORALL例外の即時処理

FORALL文で呼び出された例外を即座に処理するには、SAVE EXCEPTIONS句を省略して適切な例外ハンドラを記述します。

1つのDML文で処理済例外が発生した場合、PL/SQLはその文によって行われた変更をロールバックしますが、それより前のDML文で行われた変更はロールバックしません。

例12-12では、FORALL文が3つのUPDATE文を実行するように設計されています。ただし、2番目の文によって例外が呼び出されます。例外は例外ハンドラによって処理され、エラー・メッセージの表示と1番目のUPDATE文による変更のコミットが行われます。3番目のUPDATE文は実行されません。

例外処理の詳細は、「PL/SQLのエラー処理」を参照してください。

例12-12 FORALL例外の即時処理

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp (
  deptno NUMBER(2),
  job VARCHAR2(18)
);

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;
```

```
  depts          NumList := NumList(10, 20, 30);
  error_message  VARCHAR2(100);

BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN OTHERS THEN
    error_message := SQLERRM;
    DBMS_OUTPUT.PUT_LINE (error_message);

    COMMIT;  -- Commit results of successful updates
    RAISE;
END;
/
```

結果:

```
Procedure created.
```

プロシージャの起動:

```
BEGIN
  p;
END;
/
```

結果:

```
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19,
maximum: 18)
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19,
maximum: 18)
ORA-06512: at "HR.P", line 27
ORA-06512: at line 2
```

問合せ:

```
SELECT * FROM emp_temp;
```

結果:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
        20 Bookkeeper
        30 Analyst

3 rows selected.
```

12.4.1.4 FORALL文が完了した後のFORALL例外の処理

一部のDML文が失敗した場合でもFORALL文の処理を継続するには、SAVE EXCEPTIONS句を含めます。DML文が失敗しても、PL/SQLによって例外は呼び出されず、かわりに、その失敗に関する情報が保存されます。FORALL文の完了後、PL/SQLによって、FORALL文に対して1つの例外(ORA-24381)が呼び出されます。

ORA-24381の例外ハンドラでは、暗黙カーソル属性SQL%BULK_EXCEPTIONSから個々のDML文の失敗に関する情報を取得できます。

SQL%BULK_EXCEPTIONSは、直前のFORALL文の実行中に失敗したDML文に関する情報の連想配列とほぼ同じです。

SQL%BULK_EXCEPTIONS.COUNTは、失敗したDML文の数です。SQL%BULK_EXCEPTIONS.COUNTが0(ゼロ)でない場合、1からSQL%BULK_EXCEPTIONS. COUNTまでの各索引値iに対して、次のようになります。

- SQL%BULK_EXCEPTIONS(i).ERROR_INDEXは、失敗したDML文の数です。
- SQL%BULK_EXCEPTIONS(i).ERROR_CODEは、失敗に関するOracle Databaseエラー・コードです。

たとえば、FORALL SAVE EXCEPTIONS文で100個のDML文を実行し、10番目と64番目の文がそれぞれエラー・コードORA-12899およびORA-19278を戻して失敗した場合、次のようになります。

- SQL%BULK_EXCEPTIONS.COUNT = 2
- SQL%BULK_EXCEPTIONS(1).ERROR_INDEX = 10
- SQL%BULK_EXCEPTIONS(1).ERROR_CODE = 12899
- SQL%BULK_EXCEPTIONS(2).ERROR_INDEX = 64
- SQL%BULK_EXCEPTIONS(2).ERROR_CODE = 19278

ノート:

SAVE EXCEPTIONS 句なしの FORALL 文によって例外が呼び出されると、SQL%BULK_EXCEPTIONS.COUNT = 1 になります。

エラー・コードを使用して、次のようにSQLERRMファンクション(「SQLERRMファンクション」を参照)で関連するエラー・メッセージを取得できます。

```
SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE))
```

ただし、SQLERRMによって戻されるエラー・メッセージには、置換引数は含まれません(例12-12と例12-13のエラー・メッセージを比較してください)。

例12-13は、例12-12とほぼ同じですが、次の点が異なります。

- FORALL文に、SAVE EXCEPTIONS句が含まれます。
- 例外処理部に、ORA-24381(バルク操作が例外を呼び出して保存したときにPL/SQLによって暗黙的に呼び出される内部的に定義された例外)に対応する例外ハンドラが含まれます。この例では、ORA-24381にユーザー定義の名前dml_errorsを割り当てています。
- dml_errorsの例外ハンドラは、SQL%BULK_EXCEPTIONSとSQLERRM(およびいくつかのローカル変数)を使用して、エラー・メッセージと、エラーの原因となった文、コレクション項目および文字列を表示します。

例12-13 FORALL文が完了した後のFORALL例外の処理

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;
  depts        NumList := NumList(10, 20, 30);

  error_message  VARCHAR2(100);
  bad_stmt_no    PLS_INTEGER;
  bad_deptno     emp_temp.deptno%TYPE;
  bad_job        emp_temp.job%TYPE;

  dml_errors  EXCEPTION;
  PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN dml_errors THEN
    FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      error_message := SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      DBMS_OUTPUT.PUT_LINE (error_message);

      bad_stmt_no := SQL%BULK_EXCEPTIONS(i).ERROR_INDEX;
      DBMS_OUTPUT.PUT_LINE('Bad statement #: ' || bad_stmt_no);

      bad_deptno := depts(bad_stmt_no);
      DBMS_OUTPUT.PUT_LINE('Bad department #: ' || bad_deptno);

      SELECT job INTO bad_job FROM emp_temp WHERE deptno = bad_deptno;

      DBMS_OUTPUT.PUT_LINE('Bad job: ' || bad_job);
    END LOOP;

    COMMIT;  -- Commit results of successful updates
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unrecognized error.');
```

```
      RAISE;
END;
/
```

結果:

```
Procedure created.
```

プロシージャの起動:

```
BEGIN
  p;
END;
/
```

結果:

```
ORA-12899: value too large for column  (actual: , maximum: )
Bad statement #: 2
Bad department #: 20
Bad job: Bookkeeper

PL/SQL procedure successfully completed.
```

問合せ:

```
SELECT * FROM emp_temp;
```

結果:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
        20 Bookkeeper
        30 Analyst (Senior)

3 rows selected.
```

12.4.1.4.1 疎コレクションおよびSQL%BULK_EXCEPTIONS

FORALL文の境界句で疎コレクションを参照する場合に、DML文が失敗する原因となったコレクション要素を検出するには、索引がSQL%BULK_EXCEPTIONS(i).ERROR_INDEXである要素を検出するまで、各要素を1つずつ確認する必要があります。

FORALL文で、VALUES OF句を使用して別のコレクションに対するポインタのコレクションを参照する場合、他のコレクションの要素で索引がSQL%BULK_EXCEPTIONS(i).ERROR_INDEXであるものを検出する必要があります。

12.4.1.5 FORALL文の影響を受ける行の数の取得

FORALL文の完了後、各DML文によって影響を受けた行の数を、暗黙カーソル属性SQL%BULK_ROWCOUNTから取得できます。

FORALL文の影響を受けた行の合計数を取得するには、「SQL%ROWCOUNT属性: 影響を受けた行数」で説明されている、暗黙カーソル属性のSQL%ROWCOUNTを使用します。

SQL%BULK_ROWCOUNTは、i番目の要素が、直前に完了したFORALL文のi番目のDML文によって影響を受けた行の数に相当

する連想配列とほぼ同じです。要素のデータ型はINTEGERです。

ノート:

サーバーが Oracle Database 12c 以降で、そのクライアントが Oracle Database 11g リリース 2 以前(またはその逆)の場合、SQL%BULK_ROWCOUNT が戻す最大値は 4,294,967,295 です。

例12-14では、SQL%BULK_ROWCOUNTを使用してFORALL文の各DELETE文で削除された行の数を表示し、SQL%ROWCOUNTを使用して削除された行の合計数を表示します。

例12-15では、SQL%BULK_ROWCOUNTを使用してFORALL文の各INSERT SELECT構文で挿入された行の数を表示し、SQL%ROWCOUNTを使用して挿入された行の合計数を表示します。

例12-14 FORALLの各DELETEで影響を受けた行の数の表示

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(30, 50, 60);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp WHERE department_id = depts(j);
  FOR i IN depts.FIRST..depts.LAST LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Statement #' || i || ' deleted ' ||
      SQL%BULK_ROWCOUNT(i) || ' rows.'
    );
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total rows deleted: ' || SQL%ROWCOUNT);
END;
/
```

結果:

```
Statement #1 deleted 6 rows.
Statement #2 deleted 45 rows.
Statement #3 deleted 5 rows.
Total rows deleted: 56
```

例12-15 FORALLの各INSERT SELECTで影響を受けた行の数の表示

```
DROP TABLE emp_by_dept;
CREATE TABLE emp_by_dept AS
  SELECT employee_id, department_id
  FROM employees
  WHERE 1 = 0;
DECLARE
  TYPE dept_tab IS TABLE OF departments.department_id%TYPE;
  deptnums  dept_tab;
BEGIN
  SELECT department_id BULK COLLECT INTO deptnums FROM departments;
  FORALL i IN 1..deptnums.COUNT
```

```
    INSERT INTO emp_by_dept (employee_id, department_id)
      SELECT employee_id, department_id
      FROM employees
      WHERE department_id = deptnums(i)
      ORDER BY department_id, employee_id;
  FOR i IN 1..deptnums.COUNT LOOP
    -- Count how many rows were inserted for each department; that is,
    -- how many employees are in each department.
    DBMS_OUTPUT.PUT_LINE (
      'Dept '||deptnums(i)||': inserted '||
      SQL%BULK_ROWCOUNT(i)||' records'
    );
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total records inserted: ' || SQL%ROWCOUNT);
END;
/
```

結果:

```
Dept 10: inserted 1 records
Dept 20: inserted 2 records
Dept 30: inserted 6 records
Dept 40: inserted 1 records
Dept 50: inserted 45 records
Dept 60: inserted 5 records
Dept 70: inserted 1 records
Dept 80: inserted 34 records
Dept 90: inserted 3 records
Dept 100: inserted 6 records
Dept 110: inserted 2 records
Dept 120: inserted 0 records
Dept 130: inserted 0 records
Dept 140: inserted 0 records
Dept 150: inserted 0 records
Dept 160: inserted 0 records
Dept 170: inserted 0 records
Dept 180: inserted 0 records
Dept 190: inserted 0 records
Dept 200: inserted 0 records
Dept 210: inserted 0 records
Dept 220: inserted 0 records
Dept 230: inserted 0 records
Dept 240: inserted 0 records
Dept 250: inserted 0 records
Dept 260: inserted 0 records
Dept 270: inserted 0 records
Dept 280: inserted 0 records
Total records inserted: 106
```

12.4.2 BULK COLLECT句

BULK COLLECT句は、バルクSQLの機能であり、結果をSQLからPL/SQLに1つずつではなくバッチで戻します。

BULK COLLECT句は、次の場所に使用できます。

- SELECT INTO文

- FETCH文
- 次の文のRETURNING INTO句:
    - DELETE文
    - INSERT文
    - UPDATE文
    - EXECUTE IMMEDIATE文

BULK COLLECT句を使用して、前述の各文で結果セット全体を取り出し、それを単一の操作で1つ以上のコレクション変数に格納します(一度に1つの結果行を取り出すループ文を使用するより効率的な操作です)。

ノート:

PL/SQL は、LOOP 文の内側で FETCH 文を処理する場合と同様の方法で、BULK COLLECT 句を処理します。PL/SQL は、BULK COLLECT 句付きの文が行を戻さない場合でも、例外を呼び出しません。例 12-22 に示すように、ターゲット・コレクションが空かどうかを確認する必要があります。

ここでのトピック

- BULK COLLECT句を使用したSELECT INTO文
- BULK COLLECT句を使用したFETCH文
- BULK COLLECT句を使用したRETURNING INTO句

12.4.2.1 BULK COLLECT句を使用したSELECT INTO文

BULK COLLECT句付きのSELECT INTO文(SELECT BULK COLLECT INTO文とも呼ばれます)は、結果セット全体を選択して1つ以上のコレクション変数に入れます。

詳細は、「SELECT INTO文」を参照してください。

注意:

SELECT BULK COLLECT INTO 文は、エイリアシングに対して不安定であるため、予期しない結果が発生する可能性があります。詳細は、「SELECT BULK COLLECT INTO 文およびエイリアシング」を参照してください。

例12-16では、SELECT BULK COLLECT INTO文を使用して、データベースの2つの列を選択して2つのコレクション(ネストした表)に入れています。

例12-17では、SELECT BULK COLLECT INTO文を使用して、結果セットを選択してレコードのネストした表に入れています。

ここでのトピック

- SELECT BULK COLLECT INTO文およびエイリアシング
- SELECT BULK COLLECT INTO文の行の制限

- コレクションのループのガイドライン

例12-16 2つのネストした表に対するデータベースの2つの列のバルク選択

```
DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;

  enums NumTab;
  names NameTab;

  PROCEDURE print_first_n (n POSITIVE) IS
  BEGIN
    IF enums.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE ('Collections are empty.');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('First ' || n || ' employees:');

      FOR i IN 1 .. n LOOP
        DBMS_OUTPUT.PUT_LINE (
          '  Employee #' || enums(i) || ': ' || names(i));
      END LOOP;
    END IF;
  END;

BEGIN
  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees
  ORDER BY employee_id;

  print_first_n(3);
  print_first_n(6);
END;
/
```

結果:

```
First 3 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
First 6 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
Employee #103: Hunold
Employee #104: Ernst
Employee #105: Austin
```

例12-17 レコードのネストした表に対するバルク選択

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;
```

```
  stock_managers  NameSet;  -- nested table of records

BEGIN
  -- Assign values to nested table of records:

  SELECT first_name, last_name, hire_date
    BULK COLLECT INTO stock_managers
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

結果:

```
01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin
```

12.4.2.1.1 SELECT BULK COLLECT INTO文およびエイリアシング

次の書式の文について説明します。

```
SELECT column BULK COLLECT INTO collection FROM table ...
```

columnおよびcollectionは、それぞれIN NOCOPYおよびOUT NOCOPYサブプログラム・パラメータに類似しており、PL/SQLによって参照方式で渡されます。参照方式で渡されるサブプログラム・パラメータと同様に、エイリアシングによって予期しない結果が発生する場合があります。

関連項目:

「参照渡しパラメータによるサブプログラムのパラメータのエイリアシング」

例12-18では、コレクションnumbers1から特定の値を選択し、その値を同じコレクションに格納することを意図しています。ところが、numbers1のすべての要素が削除されるという予期しない結果が発生します。この問題の回避策は、例12-19および例12-20を参照してください。

例12-19では、カーソルを使用して例12-18で意図されていた結果を実現します。

例12-20では、コレクションnumbers1から特定の値を選択し、その値を別のコレクションnumbers2に格納します。例12-20の

実行速度は例12-19の場合より速くなります。

例12-18 予期しない結果が発生するSELECT BULK COLLECT INTO文

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  --Self-selecting BULK COLLECT INTO clause:

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers1
  FROM TABLE(numbers1) a
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
END p;
/
```

pの起動:

```
BEGIN
  p(2);
END;
/
```

結果:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0

PL/SQL procedure successfully completed.
```

pの起動:

```
BEGIN
  p(10);
END;
/
```

結果:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0
```

例12-19 カーソルによる例12-18の回避策

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);

  CURSOR c IS
    SELECT a.COLUMN_VALUE
    FROM TABLE(numbers1) a
    WHERE a.COLUMN_VALUE > p.i
    ORDER BY a.COLUMN_VALUE;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Before FETCH statement');
    DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;

  OPEN c;
  FETCH c BULK COLLECT INTO numbers1;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('After FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;
END p;
/
```

pの起動:

```
BEGIN
  p(2);
END;
/
```

結果:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 3
numbers1(1) = 3
numbers1(2) = 4
numbers1(3) = 5
```

pの起動:

```
BEGIN
  p(10);
END;
/
```

結果:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 0
```

例12-20 2番目のコレクションによる例12-18の回避策

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
 numbers2  numbers_type := numbers_type(0,0,0,0,0);

BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');

  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers2      -- numbers2 appears here
```

```
  FROM TABLE(numbers1) a        -- numbers1 appears here
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  IF numbers2.COUNT() > 0 THEN
    FOR j IN 1..numbers2.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
    END LOOP;
  END IF;
END p;
/
```

pの起動:

```
BEGIN
  p(2);
 END;
/
```

結果:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 3
numbers2(1) = 3
numbers2(2) = 4
numbers2(3) = 5

PL/SQL procedure successfully completed.
```

pの起動:

```
BEGIN
  p(10);
END;
/
```

結果:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 0
```

12.4.2.1.2 SELECT BULK COLLECT INTO文の行の制限

大量の行を戻すSELECT BULK COLLECT INTO文では、大規模なコレクションが生成されます。行数およびコレクション・サイズを制限するには、次のいずれかを使用します。

- ROWNUM疑似列(『Oracle Database SQL言語リファレンス』を参照)
- SAMPLE句(『Oracle Database SQL言語リファレンス』を参照)
- FETCH FIRST句(『Oracle Database SQL言語リファレンス』を参照)

例12-21は、SELECT BULK COLLECT INTO文により戻される行数を制限するいくつかの方法を示しています。

例12-21 ROWNUM、SAMPLEおよびFETCH FIRSTによるバルク選択の制限

```
DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
  SELECT salary BULK COLLECT INTO sals FROM employees
    WHERE ROWNUM <= 50;

  SELECT salary BULK COLLECT INTO sals FROM employees
    SAMPLE (10);

  SELECT salary BULK COLLECT INTO sals FROM employees
    FETCH FIRST 50 ROWS ONLY;
```

```
END;
/
```

12.4.2.1.3 コレクションのループのガイドライン

結果セットをコレクションに格納する場合、行をループして異なる列を参照することは簡単です。この方法では非常に高速に処理されますが、メモリーも集中的に使用されます。この方法を頻繁に使用する場合、次のことに注意してください。

- 結果セットを1回ループするには、カーソルFOR LOOPを使用します(「カーソルFOR LOOP文による問合せ結果セットの処理」を参照)。

  この方法を使用すると、結果セットのコピーを格納する際のメモリーのオーバーヘッドを回避できます。

- 結果セットをループして特定の値を検索したり、結果をフィルタリングしてより小さい結果セットにするかわりに、SELECT INTO文の問合せで検索またはフィルタリングを行います。

  たとえば、単純な問合せではWHERE句を使用し、複数の結果セットを比較する問合せでは、INTERSECTやMINUSなどの集合演算子を使用します。集合演算子の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

- 結果セットをループして各結果行に対して別の問合せを実行するかわりに、SELECT INTO文の問合せで副問合せを使用します(「副問合せによる問合せ結果セットの処理」を参照)。

- 結果セットをループして各結果行に対して別のDML文を実行するかわりに、FORALL文を使用します(「FORALL文」を参照)。

12.4.2.2 BULK COLLECT句を使用したFETCH文

BULK COLLECT句付きのFETCH文(FETCH BULK COLLECT文とも呼ばれます)は、結果セット全体をフェッチして1つ以上のコレクション変数に入れます。

詳細は、「FETCH文」を参照してください。

例12-22では、FETCH BULK COLLECT文を使用し、結果セット全体をフェッチして2つのコレクション(ネストした表)に入れます。

例12-23では、FETCH BULK COLLECT文を使用し、結果セットをフェッチしてレコードのコレクション(ネストした表)に入れます。

例12-22 2つのネストした表へのバルク・フェッチ

```
DECLARE
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  CURSOR c1 IS
    SELECT last_name, salary
    FROM employees
    WHERE salary > 10000
    ORDER BY last_name;
  names  NameList;
  sals   SalList;
  TYPE RecList IS TABLE OF c1%ROWTYPE;
  recs RecList;
  v_limit PLS_INTEGER := 10;
  PROCEDURE print_results IS
  BEGIN
    -- Check if collections are empty:
```

```
    IF names IS NULL OR names.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('No results!');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Result: ');
      FOR i IN names.FIRST .. names.LAST
      LOOP
        DBMS_OUTPUT.PUT_LINE('  Employee ' || names(i) || ': $' || sals(i));
      END LOOP;
    END IF;
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('--- Processing all results simultaneously ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals;
  CLOSE c1;
  print_results();
  DBMS_OUTPUT.PUT_LINE ('--- Processing ' || v_limit || ' rows at a time ---');
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO names, sals LIMIT v_limit;
    EXIT WHEN names.COUNT = 0;
    print_results();
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('--- Fetching records rather than columns ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO recs;
  FOR i IN recs.FIRST .. recs.LAST
  LOOP
    -- Now all columns from result set come from one record
    DBMS_OUTPUT.PUT_LINE (
      '  Employee ' || recs(i).last_name || ': $' || recs(i).salary
    );
  END LOOP;
END;
/
```

結果:

```
--- Processing all results simultaneously ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $10418.1
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $10036.95
Employee King: $24000
Employee Kochhar: $17000
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
Employee Russell: $14000
Employee Vishney: $10500
Employee Weiss: $10418.1
```

```
Employee Zlotkey: $10500
--- Processing 10 rows at a time ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $10418.1
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $10036.95
Employee King: $24000
Result:
Employee Kochhar: $17000
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
Employee Russell: $14000
Employee Vishney: $10500
Employee Weiss: $10418.1
Employee Zlotkey: $10500
--- Fetching records rather than columns ---
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $10418.1
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $10036.95
Employee King: $24000
Employee Kochhar: $17000
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
Employee Russell: $14000
Employee Vishney: $10500
Employee Weiss: $10418.1
Employee Zlotkey: $10500
```

例12-23 レコードのネストした表へのバルク・フェッチ

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;
  stock_managers  NameSet;  -- nested table of records

  TYPE cursor_var_type is REF CURSOR;
  cv cursor_var_type;

BEGIN
  -- Assign values to nested table of records:
```

```
  OPEN cv FOR
    SELECT first_name, last_name, hire_date
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  FETCH cv BULK COLLECT INTO stock_managers;
  CLOSE cv;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

結果:

```
01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin
```

12.4.2.2.1 FETCH BULK COLLECT文の行の制限

大量の行を戻すFETCH BULK COLLECT文では、大規模なコレクションが生成されます。行数およびコレクション・サイズを制限するには、LIMIT句を使用します。

例12-24では、LOOP文が繰り返されるたびに、FETCH文によって10行(またはそれ以下)が連想配列empidsにフェッチされます(前の値は上書きされます)。LOOP文の終了条件に注意してください。

例12-24 LIMITによるバルクFETCHの制限

```
DECLARE
  TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  CURSOR c1 IS
    SELECT employee_id
    FROM employees
    WHERE department_id = 80
    ORDER BY employee_id;
  empids  numtab;
BEGIN
  OPEN c1;
  LOOP  -- Fetch 10 rows or fewer in each iteration
    FETCH c1 BULK COLLECT INTO empids LIMIT 10;
    DBMS_OUTPUT.PUT_LINE ('------- Results from One Bulk Fetch --------');
    FOR i IN 1..empids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Employee Id: ' || empids(i));
    END LOOP;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
```

```
END;
/
```

結果:

```
------- Results from One Bulk Fetch --------
Employee Id: 145
Employee Id: 146
Employee Id: 147
Employee Id: 148
Employee Id: 149
Employee Id: 150
Employee Id: 151
Employee Id: 152
Employee Id: 153
Employee Id: 154
------- Results from One Bulk Fetch --------
Employee Id: 155
Employee Id: 156
Employee Id: 157
Employee Id: 158
Employee Id: 159
Employee Id: 160
Employee Id: 161
Employee Id: 162
Employee Id: 163
Employee Id: 164
------- Results from One Bulk Fetch --------
Employee Id: 165
Employee Id: 166
Employee Id: 167
Employee Id: 168
Employee Id: 169
Employee Id: 170
Employee Id: 171
Employee Id: 172
Employee Id: 173
Employee Id: 174
------- Results from One Bulk Fetch --------
Employee Id: 175
Employee Id: 176
Employee Id: 177
Employee Id: 179
```

12.4.2.3 BULK COLLECT句を使用したRETURNING INTO句

BULK COLLECT句付きのRETURNING INTO句(RETURNING BULK COLLECT INTO句とも呼ばれます)は、INSERT文、UPDATE文、DELETE文またはEXECUTE IMMEDIATE文で使用できます。RETURNING BULK COLLECT INTO句を使用すると、文によってその結果セットが1つ以上のコレクションに格納されます。

詳細は、「RETURNING INTO句」を参照してください。

例12-25では、RETURNING BULK COLLECT INTO句付きのDELETE文を使用して、表から行を削除し、それらの行を2つのコレクション(ネストした表)に戻します。

例12-25 2つのネストした表に対する削除行の戻し

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id;
DECLARE
  TYPE NumList IS TABLE OF employees.employee_id%TYPE;
  enums  NumList;
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  names  NameList;
BEGIN
  DELETE FROM emp_temp
  WHERE department_id = 30
  RETURNING employee_id, last_name
  BULK COLLECT INTO enums, names;
  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN enums.FIRST .. enums.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
  END LOOP;
END;
/
```

結果:

```
Deleted 6 rows:
Employee #114: Raphaely
Employee #115: Khoo
Employee #116: Baida
Employee #117: Tobias
Employee #118: Himuro
Employee #119: Colmenares
```

12.4.3 FORALL文とBULK COLLECT句の併用

FORALL文では、DML文にRETURNING BULK COLLECT INTO句を含めることができます。FORALL文が繰り返されるたびに、DML文は前の値を上書きすることなく、指定された値を指定されたコレクションに格納します(この動作は、FOR LOOP文における同じDML文の処理と同様です)。

例12-26では、FORALL文で、RETURNING BULK COLLECT INTO句が含まれるDELETE文を実行します。FORALL文が繰り返されるたびに、DELETE文は、削除された行のemployee_idおよびdepartment_idの値をそれぞれコレクションe_idsおよびd_idsに格納します。

例12-27は、例12-26と類似していますが、FORALL文のかわりにFOR LOOP文を使用しています。

例12-26 FORALL文でのRETURN BULK COLLECT INTO付きのDELETE

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id, department_id;
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
```

```
  depts  NumList := NumList(10,20,30);
  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;
  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Employee #' || e_ids(i) || ' from dept #' || d_ids(i)
    );
  END LOOP;
END;
/
```

結果:

```
Deleted 9 rows:
Employee #200 from dept #10
Employee #201 from dept #20
Employee #202 from dept #20
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30
```

例12-27 FOR LOOP文でのRETURN BULK COLLECT INTO付きのDELETE

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts  NumList := NumList(10,20,30);

  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;

  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;

BEGIN
  FOR j IN depts.FIRST..depts.LAST LOOP
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');

  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
```

```
    DBMS_OUTPUT.PUT_LINE (
      'Employee #' || e_ids(i) || ' from dept #' || d_ids(i)
    );
  END LOOP;
END;
/
```

結果:

```
Deleted 6 rows:
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30
```

12.4.4 ホスト配列のクライアント・バルク・バインド

クライアント・プログラム(OCIプログラムやPro*Cプログラムなど)は、PL/SQLの無名ブロックを使用してホスト配列をバルク・バインド入出力できます。これは、コレクションをデータベース・サーバーとの間でやり取りするのに最も効率的な方法です。

クライアント・プログラムで、値を宣言し、無名ブロックで参照されるホスト変数にそれらの値を代入します。無名ブロックでは、各ホスト変数名にコロン(:)を接頭辞として付け、PL/SQLのコレクション変数名と区別します。クライアント・プログラムが実行されると、データベース・サーバーでPL/SQLの無名ブロックが実行されます。

例12-28では、無名ブロックでFORALL文を使用してホスト入力配列をバルク・バインドします。FORALL文において、DELETE文は、スカラーlower、upper、emp_idおよび配列deptsという4つのホスト変数を参照します。

例12-28 無名ブロックによる入力ホスト配列のバルク・バインド

```
BEGIN
  FORALL i IN :lower..:upper
    DELETE FROM employees
    WHERE department_id = :depts(i);
END;
/
```

12.5 複数変換用のパイプライン・テーブル・ファンクションの連鎖

パイプライン・テーブル・ファンクションの連鎖は、複数のデータ変換を実行する場合に効率的な方法です。

ノート:

パイプライン・テーブル・ファンクションは、データベース・リンクを介して実行することはできません。パイプライン・テーブル・ファンクションの戻り型がユーザー定義の SQL 型であり、単一データベース内でのみ使用可能であるためです(『Oracle Database オブジェクト・リレーショナル開発者ガイド』を参照)。パイプライン・テーブル・ファンクションの戻り型が PL/SQL 型のように見える場合でも、その PL/SQL 型は実際にはデータベースによって対応するユーザー定義の SQL 型に変換されます。

ここでのトピック

- テーブル・ファンクションの概要
- パイプライン・テーブル・ファンクションの作成
- 変換ファンクションとしてのパイプライン・テーブル・ファンクション
- パイプライン・テーブル・ファンクションの連鎖
- パイプライン・テーブル・ファンクションの結果からのフェッチ
- パイプライン・テーブル・ファンクションへのCURSOR式の引渡し
- パイプライン・テーブル・ファンクションの結果に対するDML文
- NO_DATA_NEEDED例外

12.5.1 テーブル・ファンクションの概要

テーブル・ファンクションは、行のコレクション(連想配列、ネストした表またはVARRAY)を戻すユーザー定義のPL/SQLファンクションです。

SELECT文のTABLE句の内部でテーブル・ファンクションを起動することで、データベース表のようにこのコレクションから要素を選択できます。TABLE演算子は省略可能です。

たとえば:

```
SELECT * FROM TABLE(table_function_name(parameter_list))
```

次のように、同等の問合せをTABLE演算子なしで記述できます。

```
SELECT * FROM table_function_name(parameter_list)
```

テーブル・ファンクションは、入力として行のコレクションを使用することができます(入力パラメータとしてネストした表、VARRAYまたはカーソル変数を含めることができます)。したがって、テーブル・ファンクションtf1からの出力をテーブル・ファンクションtf2に入力したり、tf2からの出力をテーブル・ファンクションtf3に入力することなどができます。

テーブル・ファンクションのパフォーマンスを向上するには、次の操作を実行します。

- PARALLEL_ENABLEオプションを使用して、ファンクションのパラレル実行を有効にします。

  パラレル実行が有効化されたファンクションは、同時に実行できます。

- Oracle Streamsを使用して、次のプロセスにファンクションの結果をストリーム化して直接送ります。

  ストリームによって、プロセス間の中間的なステージングが排除されます。

- PIPELINEDオプションを使用して、ファンクションの結果をパイプライン化します。

  パイプライン・テーブル・ファンクションは、行を処理した直後に起動元に行を戻し、行の処理を継続します。問合せで1つの結果行を戻す前にコレクション全体を組み立ててサーバーに戻す必要がないため、応答時間が短縮されます。(オブジェクト・キャッシュでコレクション全体をマテリアライズする必要がないため、ファンクションのメモリー消費量も減少します。)

注意:

パイプライン・テーブル・ファンクションは、常にデータの現在の状態を参照します。コレクションに対するカーソルのオープン後にコレクションのデータが変更されると、カーソルにその変更が反映されます。PL/SQL 変数は単一セッション内でのみ有効であり、トランザクション対応ではありません。このため、表データへの適用性がよく知られている読取り一貫性は、PL/SQL コレクション変数には適用されません。

関連項目:

- パイプライン・テーブル・ファンクションの連鎖
- SELECT文のTABLE句の詳細は、『Oracle Database SQL言語リファレンス』 を参照してください
- パイプライン・テーブル・ファンクションおよびパラレル・テーブル・ファンクションを使用する方法の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

12.5.2 パイプライン・テーブル・ファンクションの作成

パイプライン・テーブル・ファンクションは、スタンドアロン・ファンクションまたはパッケージ・ファンクションのいずれかである必要があります。

PIPELINEDオプション(必須)

スタンドアロン・ファンクションでは、CREATE FUNCTION文にPIPELINEDオプションを指定します(構文は、「CREATE FUNCTION文」を参照)。パッケージ・ファンクションでは、ファンクション宣言とファンクション定義の両方にPIPELINEDオプションを指定します(構文は、「ファンクションの宣言および定義」を参照してください)。

PARALLEL_ENABLEオプション(推奨)

パフォーマンスを向上するには、PARALLEL_ENABLEオプションを指定して、パイプライン・テーブル・ファンクションのパラレル実行を有効にします。

AUTONOMOUS_TRANSACTIONプラグマ

パイプライン・テーブル・ファンクションでDML文を実行する場合、AUTONOMOUS_TRANSACTIONプラグマ(「AUTONOMOUS_TRANSACTIONプラグマ」を参照)を使用すると、ファンクションが自律型になります。その後、パラレル実行中に、ファンクションの各インスタンスが独立したトランザクションを作成します。

DETERMINISTICオプション(推奨)

パイプライン・テーブル・ファンクションを同じ問合せまたは別の問合せで複数回起動すると、基礎となる実装が複数回実行されます。ファンクションが決定的である場合、DETERMINISTICオプションを指定します(DETERMINISTIC句を参照)。

パラメータ

通常、パイプライン・テーブル・ファンクションには、1つ以上のカーソル変数パラメータが含まれます。ファンクション・パラメータとしてのカーソル変数の詳細は、「サブプログラム・パラメータとしてのカーソル変数」を参照してください。

関連項目:

- カーソル変数の一般情報は、「カーソル変数」を参照してください
- サブプログラムのパラメータの詳細は、「サブプログラムのパラメータ」を参照してください

RETURNデータ型

パイプライン・テーブル・ファンクションが戻す値のデータ型は、スキーマ・レベルまたはパッケージ内で定義されたコレクション型である必要があります(そのため、結合配列型にはできません)。コレクション型の要素は、PL/SQLによってのみサポートされるデータ型(PLS_INTEGERやBOOLEANなど)ではなく、SQLデータ型にする必要があります。コレクション型の詳細は、「コレクション型」を参照してください。SQLデータ型の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

SQLデータ型のANYTYPE、ANYDATAおよびANYDATASETを使用して、オブジェクト型やコレクション型などの他のSQL型の型記述、データ・インスタンスおよびデータ・インスタンス・セットを動的にカプセル化してアクセスできます。また、これらの型を使用すると、匿名コレクション型などの名前を持たない型を作成できます。これらの型の詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

PIPE ROW文

パイプライン・テーブル・ファンクションの内部でPIPE ROW文を使用すると、制御を起動元に戻すことなく、起動元にコレクション要素を戻すことができます。構文およびセマンティクスの詳細は、「PIPE ROW文」を参照してください。

RETURN文

他のすべてのファンクションと同様に、パイプライン・テーブル・ファンクションのすべての実行パスは、RETURN文に導かれ、それによって制御が起動元に戻される必要があります。ただし、パイプライン・テーブル・ファンクションでは、RETURN文で起動元に値を戻す必要はありません。その構文およびセマンティクスは、「RETURN文」を参照してください。

例

例12-29 パイプライン・テーブル・ファンクションの作成および起動

この例では、パイプライン・テーブル・ファンクションf1を含むパッケージを作成し、f1が戻す行のコレクションから要素を選択します。

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/
```

要素(1,2,3,... x)のコレクションを戻すパイプライン・テーブル・ファンクションf1を作成します。

```
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
  BEGIN
    FOR i IN 1..x LOOP
      PIPE ROW(i);
    END LOOP;
    RETURN;
```

```
  END f1;
END pkg1;

SELECT * FROM TABLE(pkg1.f1(5));
```

結果:

```
COLUMN_VALUE
------------
           1
           2
           3
           4
           5

5 rows selected.

SELECT * FROM pkg1.f1(2);
```

結果:

```
COLUMN_VALUE
------------
           1
           2
```

12.5.3 変換ファンクションとしてのパイプライン・テーブル・ファンクション

カーソル変数パラメータを持つパイプライン・テーブル・ファンクションは、変換ファンクションとして使用できます。ファンクションは、カーソル変数を使用して入力行をフェッチします。ファンクションは、PIPE ROW文を使用して、変換された1つ以上の行を起動元にパイプします。FETCH文およびPIPE ROW文がLOOP文の内側にある場合、ファンクションで複数の入力行を変換できます。

例12-30では、パイプライン・テーブル・ファンクションによって、employees表の各選択行がネストした表の2つの行に変換され、起動元のSELECT文にパイプされます。仮カーソル変数パラメータに対応する実パラメータは、CURSOR式です(詳細は、「パイプライン・テーブル・ファンクションへのCURSOR式の引渡し」を参照してください)。

例12-30 パイプライン・テーブル・ファンクションによる各行の2つの行への変換

```
CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED;
END refcur_pkg;
/
CREATE OR REPLACE PACKAGE BODY refcur_pkg IS
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED IS
    out_rec outrec_typ;
    in_rec  p%ROWTYPE;
  BEGIN
```

```
    LOOP
      FETCH p INTO in_rec;  -- input row
      EXIT WHEN p%NOTFOUND;
      out_rec.var_num := in_rec.employee_id;
      out_rec.var_char1 := in_rec.first_name;
      out_rec.var_char2 := in_rec.last_name;
      PIPE ROW(out_rec);  -- first transformed output row
      out_rec.var_char1 := in_rec.email;
      out_rec.var_char2 := in_rec.phone_number;
      PIPE ROW(out_rec);  -- second transformed output row
    END LOOP;
    CLOSE p;
    RETURN;
  END f_trans;
END refcur_pkg;
/
SELECT * FROM TABLE (
  refcur_pkg.f_trans (
    CURSOR (SELECT * FROM employees WHERE department_id = 60)
  )
);
```

結果:

```
   VAR_NUM VAR_CHAR1                       VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      Hunold
       103 AHUNOLD                        590.423.4567
       104 Bruce                          Ernst
       104 BERNST                         590.423.4568
       105 David                          Austin
       105 DAUSTIN                        590.423.4569
       106 Valli                          Pataballa
       106 VPATABAL                       590.423.4560
       107 Diana                          Lorentz
       107 DLORENTZ                       590.423.5567
10 rows selected.
```

12.5.4 パイプライン・テーブル・ファンクションの連鎖

パイプライン・テーブル・ファンクションのtf1とtf2を連鎖するには、tf1の出力をtf2の入力にします。たとえば:

```
SELECT * FROM TABLE(tf2(CURSOR(SELECT * FROM TABLE(tf1()))));
```

tf1によってパイプ出力される行は、tf2の仮入力パラメータと互換性のある実パラメータである必要があります。

連鎖されたパイプライン・テーブル・ファンクションでパラレル実行が有効化されていると、各ファンクションは異なるプロセス(またはプロセスのセット)で実行されます。

関連項目:

「パイプライン・テーブル・ファンクションへのCURSOR式の引渡し」

12.5.5 パイプライン・テーブル・ファンクションの結果からのフェッチ

名前付きカーソルは、パイプライン・テーブル・ファンクションを起動する問合せに関連付けることができます。このようなカーソルに特別なフェッチ・セマンティクスはなく、また、このようなカーソル変数に特別な代入セマンティクスはありません。

ただし、SQLオプティマイザでは、PL/SQL文にまたがる最適化は行われません。したがって、例12-31で、2番目のPL/SQL文には2つのSQL文を実行するオーバーヘッドがありますが、1番目のPL/SQL文は2番目の文より遅くなり、この結果は、1番目のPL/SQL文の2つのSQL文の間でファンクション結果がパイプされる場合でも変わりません。

例12-31で、fおよびgがパイプライン・テーブル・ファンクションであり、各ファンクションでカーソル変数パラメータを受け入れるとします。1番目のPL/SQL文は、カーソル変数rを、fを起動する問合せに関連付けて、rをgに渡します。2番目のPL/SQL文は、CURSOR式をfとgの両方に渡します。

関連項目:

「サブプログラム・パラメータとしてのカーソル変数」

例12-31 パイプライン・テーブル・ファンクションの結果からのフェッチ

```
DECLARE
  r SYS_REFCURSOR;
  ...
  -- First PL/SQL statement (slower):
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
  SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
  -- NOTE: When g completes, it closes r.
END;
-- Second PL/SQL statement (faster):
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
/
```

12.5.6 パイプライン・テーブル・ファンクションへのCURSOR式の引渡し

例12-31のように、パイプライン・テーブル・ファンクションのカーソル変数パラメータの実パラメータには、カーソル変数またはCURSOR式を使用できますが、後者の方が効率的です。

ノート:

SQL の SELECT 文で CURSOR 式をファンクションに渡す場合、参照先のカーソルは、ファンクションの実行開始時にオープンされ、ファンクションの完了時にクローズされます。

関連項目:

CURSOR式の一般情報は、「CURSOR式」を参照してください

例12-32では、2つのカーソル変数パラメータがあるパイプライン・テーブル・ファンクションを含むパッケージを作成し、実パラメータにCURSOR式を使用してSELECT文のファンクションを起動します。

例12-33では、パイプライン・テーブル・ファンクションを集計ファンクションとして使用し、入力行のセットを取得して1つの結果を戻します。ファンクションの結果は、SELECT文で選択します。(疑似列COLUMN_VALUEの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。)

例12-32 2つのカーソル変数パラメータを使用するパイプライン・テーブル・ファンクション

```
CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t1 IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE refcur_t2 IS REF CURSOR RETURN departments%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION g_trans (p1 refcur_t1, p2 refcur_t2) RETURN outrecset PIPELINED;
END refcur_pkg;
/

CREATE PACKAGE BODY refcur_pkg IS
  FUNCTION g_trans (
    p1 refcur_t1,
    p2 refcur_t2
  ) RETURN outrecset PIPELINED
  IS
    out_rec outrec_typ;
    in_rec1 p1%ROWTYPE;
    in_rec2 p2%ROWTYPE;
  BEGIN
    LOOP
      FETCH p2 INTO in_rec2;
      EXIT WHEN p2%NOTFOUND;
    END LOOP;
    CLOSE p2;
    LOOP
      FETCH p1 INTO in_rec1;
      EXIT WHEN p1%NOTFOUND;
      -- first row
      out_rec.var_num := in_rec1.employee_id;
      out_rec.var_char1 := in_rec1.first_name;
      out_rec.var_char2 := in_rec1.last_name;
      PIPE ROW(out_rec);
      -- second row
      out_rec.var_num := in_rec2.department_id;
      out_rec.var_char1 := in_rec2.department_name;
      out_rec.var_char2 := TO_CHAR(in_rec2.location_id);
      PIPE ROW(out_rec);
    END LOOP;
    CLOSE p1;
    RETURN;
  END g_trans;
END refcur_pkg;
/
```

```
SELECT * FROM TABLE (
  refcur_pkg.g_trans (
    CURSOR (SELECT * FROM employees WHERE department_id = 60),
    CURSOR (SELECT * FROM departments WHERE department_id = 60)
  )
);
```

結果:

```
   VAR_NUM VAR_CHAR1                      VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      Hunold
        60 IT                             1400
       104 Bruce                          Ernst
        60 IT                             1400
       105 David                          Austin
        60 IT                             1400
       106 Valli                          Pataballa
        60 IT                             1400
       107 Diana                          Lorentz
        60 IT                             1400

10 rows selected.
```

例12-33 集計ファンクションとしてのパイプライン・テーブル・ファンクション

```
DROP TABLE gradereport;
CREATE TABLE gradereport (
  student VARCHAR2(30),
  subject VARCHAR2(30),
  weight NUMBER,
  grade NUMBER
);
INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Physics', 4, 4);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Chemistry', 4, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Maths', 3, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Economics', 3, 4);

CREATE PACKAGE pkg_gpa AUTHID DEFINER IS
  TYPE gpa IS TABLE OF NUMBER;
  FUNCTION weighted_average(input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED;
END pkg_gpa;
CREATE PACKAGE BODY pkg_gpa IS
  FUNCTION weighted_average (input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED
  IS
    grade         NUMBER;
    total         NUMBER := 0;
    total_weight  NUMBER := 0;
    weight        NUMBER := 0;
```

```
  BEGIN
    LOOP
      FETCH input_values INTO weight, grade;
      EXIT WHEN input_values%NOTFOUND;
      total_weight := total_weight + weight;  -- Accumulate weighted average
      total := total + grade*weight;
    END LOOP;
    PIPE ROW (total / total_weight);
    RETURN; -- returns single result
  END weighted_average;
END pkg_gpa;
```

この問合せは、テーブル・ファンクションをTABLE演算子なしで起動する方法を示しています。

```
SELECT w.column_value "weighted result"
FROM pkg_gpa.weighted_average (
    CURSOR (SELECT weight, grade FROM gradereport)
  ) w;
```

結果:

```
weighted result
---------------
            3.5

1 row selected.
```

12.5.7 パイプライン・テーブル・ファンクションの結果に対するDML文

パイプライン・テーブル・ファンクションが戻す表を、DELETE文、INSERT文、UPDATE文またはMERGEのターゲット表にすることはできません。ただし、このような表のビューを作成し、そのビューに対してINSTEAD OFトリガーを作成できます。INSTEAD OFトリガーの詳細は、「INSTEAD OF DMLトリガー」を参照してください。

関連項目:

CREATE VIEW文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

12.5.8 NO_DATA_NEEDED例外

事前定義の例外NO_DATA_NEEDEDを理解する必要があるのは、次の2つの場合です。

- PIPE ROW文が存在するブロックにOTHERS例外ハンドラを含める場合
- PIPE ROW文を使用するコードの次に、クリーンアップ・プロシージャを続ける必要がある場合

  通常、クリーンアップ・プロシージャは、コードで必要とされなくなったリソースを解放します。

パイプライン・テーブル・ファンクションの起動元がファンクションからの行をそれ以上必要としなくなると、PIPE ROW文によってNO_DATA_NEEDEDが呼び出されます。パイプライン・テーブル・ファンクションでNO_DATA_NEEDEDを処理しない場合、例12-34に示すとおり、ファンクションの起動は終了しますが、起動元の文は終了しません。パイプライン・テーブル・ファンクションでNO_DATA_NEEDEDを処理する場合、例12-35に示すとおり、例外ハンドラによって不要になったリソースを解放できます。

例12-34では、パイプライン・テーブル・ファンクションpipe_rowsでNO_DATA_NEEDED例外を処理しません。pipe_rowsを起動するSELECT文では、4つの行のみを必要とします。そのため、pipe_rowsの5回目の起動時に、PIPE ROW文によって例外NO_DATA_NEEDEDが呼び出されます。pipe_rowsの5回目の起動は終了しますが、SELECT文は終了しません。

PIPE ROW文が存在するブロックの例外処理部に、予期しない例外を処理するためのOTHERS例外ハンドラを含める場合、その例外処理部に予期されるNO_DATA_NEEDED例外のための例外ハンドラも含める必要があります。そうしない場合、NO_DATA_NEEDED例外は、OTHERS例外ハンドラで処理され、予期しないエラーとして扱われます。次の例外ハンドラでは、リカバリ不可能なエラーとして扱われるかわりに、NO_DATA_NEEDED例外が再度呼び出されます。

```
EXCEPTION
  WHEN NO_DATA_NEEDED THEN
    RAISE;
  WHEN OTHERS THEN
    -- (Put error-logging code here)
    RAISE_APPLICATION_ERROR(-20000, 'Fatal error.');
END;
```

例12-35では、External_Sourceパッケージに次のパブリック項目が含まれると想定します。

- プロシージャInit: Next_Rowが必要とするリソースの割当てと初期化を行います。
- ファンクションNext_Row: 特定の外部ソースのデータを戻し、外部ソースのデータがなくなるとユーザー定義の例外Done(同じくパッケージ内のパブリック項目)を呼び出します。
- プロシージャClean_Up: Initで割り当てたリソースを解放します。

パイプライン・テーブル・ファンクションget_external_source_dataは、External_Source.Next_Rowを起動して、次のいずれかの状態になるまで外部ソースから行をパイプします。

- 外部ソースの行がなくなった場合。

  この場合、External_Source.Next_Rowファンクションによってユーザー定義の例外External_Source.Doneが呼び出されます。

- get_external_source_dataで行を必要としなくなった場合

  この場合、get_external_source_dataのPIPE ROW文によってNO_DATA_NEEDED例外が呼び出されます。

どちらの場合でも、get_external_source_dataのブロックbにある例外ハンドラによって、Next_Rowが使用していたリソースを解放するExternal_Source.Clean_Upが起動されます。

例12-34 パイプライン・テーブル・ファンクションで処理されないNO_DATA_NEEDED

```
CREATE TYPE t IS TABLE OF NUMBER
/
CREATE OR REPLACE FUNCTION pipe_rows RETURN t PIPELINED AUTHID DEFINER IS
  n NUMBER := 0;
BEGIN
  LOOP
    n := n + 1;
    PIPE ROW (n);
  END LOOP;
END pipe_rows;
/
SELECT COLUMN_VALUE
```

```
  FROM TABLE(pipe_rows())
  WHERE ROWNUM < 5
/
```

結果:

```
COLUMN_VALUE
------------
           1
           2
           3
           4
4 rows selected.
```

例12-35 パイプライン・テーブル・ファンクションで処理されるNO_DATA_NEEDED

```
CREATE OR REPLACE FUNCTION get_external_source_data
  RETURN t PIPELINED AUTHID DEFINER IS
BEGIN
  External_Source.Init();           -- Initialize.
  <<b>> BEGIN
    LOOP                            -- Pipe rows from external source.
      PIPE ROW (External_Source.Next_Row());
    END LOOP;
  EXCEPTION
    WHEN External_Source.Done THEN  -- When no more rows are available,
      External_Source.Clean_Up();   --  clean up.
    WHEN NO_DATA_NEEDED THEN        -- When no more rows are needed,
      External_Source.Clean_Up();   --  clean up.
      RAISE NO_DATA_NEEDED;           -- Optional, equivalent to RETURN.
  END b;
END get_external_source_data;
/
```

12.6 多相テーブル・ファンクションの概要

多相テーブル・ファンクション(PTF)とは、オペランドに複数の型を指定できるテーブル・ファンクションです。戻り値の型はPTF起動の引数リストによって決定されます。通常は表の型への実際の引数によって行の出力形状が決まりますが、異なる場合もあります。

多相テーブル・ファンクションについて

多相テーブル・ファンクション(PTF)はユーザー定義のファンクションで、SQL問合せブロックのFROM句から起動できます。定義時に行の型が宣言されない表を処理したり、定義時に行の型が必ずしも宣言されない結果表を生成したりできます。多相テーブル・ファンクションは動的SQL機能を利用して、強力かつ複雑なカスタム・ファンクションを作成します。これが便利なのは、アプリケーションのインタフェースに、任意的な入力表または問合せに対応できる汎用的な拡張機能が求められる場合です。

PTF作成者は、表を定義する手続き型メカニズムへのインタフェースを作成します。PTF作成者はPTFの定義、ドキュメント化および実装を行います。

問合せ作成者は、公開されたインタフェースを記述して、問合せ内でPTFを起動することしかできません。

データベースはPTFの指揮者として機能します。これがPTFのコンパイルと実行状態を管理します。データベースおよびPTF作成

者は、関連するSQLで起動されるプロシージャのファミリや、呼び出されたPTFコンポーネント・プロシージャ、場合によってはプライベート・データ(変数やカーソルなど)を見ることができます。

多相テーブル・ファンクションの型

多相テーブル・ファンクションの型は仮引数リストのセマンティクスに基づいて指定されます。

- 入力TABLE引数がRow Semanticsである場合、入力は1行です。
- 入力TABLE引数がTable Semanticsである場合、入力は行のセットです。Table Semantics PTFを問合せから呼び出すときに、PARTITION BY句またはORDER BY句のいずれかまたは両方を使用して表引数を拡張できます。

12.6.1 多相テーブル・ファンクションの定義

PTF作成者は多相テーブル・ファンクション(PTF)の定義、ドキュメント化および実装を行います。

PTFは2つの部分で構成されます。

1. PTF実装のクライアント・インタフェースを含むPL/SQLパッケージ。
2. PTFおよび関連付けられている実装パッケージを指定するスタンドアロン・ファンクションまたはパッケージ・ファンクション。

12.6.2 多相テーブル・ファンクションの実装

多相テーブル・ファンクション(PTF)実装のクライアント・インタフェースは固定名を持つサブプログラムのセットであり、すべてのPTFがこれを提供する必要があります。

多相テーブル・ファンクションの実装ステップ

1. DESCRIBEファンクション(必須)と、OPEN、FETCH_ROWSおよびCLOSEプロシージャ(任意)を含む実装パッケージを作成します。
2. PTFを指定したファンクションの仕様部を作成します。パッケージ作成後の最上位レベルでファンクションを作成するか、または実装パッケージ(最初のステップで作成したパッケージ)内でパッケージ・ファンクションとしてファンクションを作成します。多相テーブル・ファンクションにはファンクション定義(FUNCTION BODY)がありません。定義は関連付けられている実装パッケージにカプセル化されます。

   ファンクション定義では次を指定します。

   - 多相テーブル・ファンクション(PTF)名
   - TABLE型の仮引数を1つ、および任意の数の非TABLE型の引数
   - PTFの戻り型TABLE
   - PTFファンクションの型(rowまたはtableセマンティクス)
   - PTF実装パッケージ名

関連項目:

- DESCRIBEのみの多相テーブル・ファンクションの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレ

ンス』

- PTF実装パッケージの指定方法とDBMS_TFユーティリティの使用方法の詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』
- スタンドアロンまたはパッケージ多相テーブル・ファンクションを作成する構文およびセマンティクの詳細は、PIPELINED句

12.6.3 多相テーブル・ファンクションの起動

多相テーブル・ファンクションを起動するには、SQL問合せブロックのFROM句でファンクション名を指定し、その後に引数リストを指定します。

PTF引数は通常のテーブル・ファンクションに渡すことができる標準スカラー引数として指定することもできますが、これに加えてPTFでは表引数を取ることができます。表引数は、WITH句問合せまたはFROM句で許可されているスキーマ・レベルのオブジェクト(表、ビューまたはテーブル・ファンクションなど)のいずれかです。

構文

table_argument ::= table [PARTITION BY column_list] [ORDER BY order_column_list]

column_list ::= identifier | (identifier[, identifier…])

order_column_list ::= order_column_name | (order_column_name [, order_column_name…])

order_column_name ::= identifier [ASC | DESC][NULLS FIRST | NULLS LAST]

セマンティクス

各識別子は対応する表内の1つの列です。

PTFにはTable Semanticsがあります。

必要に応じて、問合せでTable Semantics PTF入力をパーティション化したり順序付けしたりできます。これはRow Semantics PTF入力では許可されません。

多相テーブル・ファンクション(PTF)をDML文のターゲットにすることはできません。PTFの表引数は名前で渡されます。

たとえば、noop PTFは次のような問合せで使用できます。

```
SELECT *
FROM noop(emp);
```

または

```
WITH e AS
 (SELECT * FROM emp NATURAL JOIN dept)
SELECT t.* FROM noop(e) t;
```

入力表引数は、基本的な表名である必要があります。

表の識別子の名前解決ルールは次のとおりです(優先順)。

1. 識別子が列名(外側の問合せブロックからの相関列など)に解決されます。
2. 識別子が現在の問合せブロック内または一部の外側の問合せブロック内の共通表式(CTE)名に解決されます。CTEは一般にWITH句として知られています。
3. 識別子がスキーマ・レベルの表、ビューまたはテーブル・ファンクション(通常または多相で、スキーマ・レベルまたはパッケージ内で定義される)に解決されます。

表の式の多くのタイプは、FROM句で許可されるとしても、PTFの表引数として直接使用することはできません(ANSI結合、バインド変数、インライン・ビュー、CURSOR演算子、TABLE演算子など)。このような表の式をPTF引数として使用するには、これらの式をCTEでラップし、CTE名をPTFに渡す間接的な方法を使用します。

PTFはFROM句で表参照として使用できるため、ANSI結合やLATERAL構文の一部にすることができます。また、PTFはPIVOTまたはUNPIVOTおよびMATCH_RECOGNIZEのソース表にすることができます。表やビューを対象にした一部の表変更の句(SAMPLING、PARTITION、CONTAINERSなど)は、PTFでは許可されません。

PTFの直接的なファンクション合成は許可されます(ネストされたPTFカーソル式の起動やPTF(TF())ネスティングなど)。ただし、ネストされたPTFは許可されません(PTF(PTF())のようなネスティング)。

PTFのスカラー引数として任意のSQLスカラー式を指定できます。定数のスカラー値はDESCRIBEファンクションにそのまま渡されますが、その他の値はすべてNULLとして渡されます。これらの値が行の形状を決定するものでなければ、通常これがPTF実装で問題になることはありませんが、そうでない場合DESCRIBEファンクションでエラーが発生することがあります。通常、PTF付属のドキュメントに、どれがスカラー・パラメータで(存在する場合)、それゆえNULL以外の定数値を必要とするとの記載があります。問合せの実行中(OPEN、FETCH_ROWS、CLOSEの間)に式が評価され、これらのPTF実行プロシージャに実際の値が渡されます。戻り値の型はPTF起動の引数リストによって決定されます。

問合せ引数はWITH句を使用して、PTFに渡されます。

テーブル・ファンクションの引数リストまたは空のlist ()がある場合、TABLE演算子は省略できます。

12.6.3.1 可変個引数擬似演算子

可変個引数擬似演算子は、可変数のオペランドで動作します。

Oracle Databaseリリース18c以上では、可変個引数擬似演算子の概念をSQL式言語に導入し、多相テーブル・ファンクション(PTF)をサポートします。擬似演算子を使用すると、識別子のリスト(列名など)をPTFに渡すことができます。疑似演算子は、PTFへの引数としてのみ表示され、他のSQL演算子やPL/SQLファンクションの起動などと同様にSQLコンパイラによって解析されます。疑似演算子の引数は可変数ですが、少なくとも1つは必要です。疑似演算子には関連付けられている実行関数がなく、PTFコンパイルが終了した後にSQLカーソルから完全に削除されます。SQLのコンパイル中に、疑似演算子は対応するDBMS_TF型に変換されてからDESCRIBEメソッドに渡されます。これらの演算子に関連付けられた出力タイプはありません。汎用のSQL式に擬似演算子を埋め込むことはできません。

12.6.3.2 COLUMNS擬似演算子

COLUMNS擬似演算子を使用して、SQL問合せブロックのFROM句で、多相テーブル・ファンクション(PTF)の起動の引数を指定できます。

COLUMNS疑似演算子の引数は、列名のリスト、または関連付けられたタイプを使用して列名のリストを指定します。

構文

column_operator ::= COLUMNS (column_list)

column_list ::= column_name_list | column_type_list

column_name_list ::= identifier [, identifier ...]

column_type_list::= identifier column_type [, identifier column_type…]

セマンティクス

COLUMNS疑似演算子はPTFへの引数としてのみ指定できます。PTF式自体以外のその他のSQL式では使用できません。

column_typeはスカラー型である必要があります。

12.6.4 多相テーブル・ファンクションのコンパイルおよび実行

データベースは多相テーブル・ファンクション(PTF)の指揮者の役割を担います。そのため、PTFのコンパイルから実行、および関連する状態までを管理します。

データベースは次を管理します。

- コンパイル状態: DESCRIBEによって生成される不変の状態で、実行前に必要です。
- 実行状態: Table semantics PTFの実行プロシージャによって使用される状態です。

.

関連項目:

- データベースがPTFのコンパイルと実行状態を管理する方法の詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンス

12.6.5 多相テーブル・ファンクションの最適化

多相テーブル・ファンクション(PTF)は、データベースの分析機能を拡張するための効率的でスケーラブルなフレームワークを提供します。

主な利点は次のとおりです。

- 最小限のデータ移動: 関心のある列のみがPTFに渡されます。
- 述語/射影/パーティション化が基礎となる表/問合せ(意味的に可能な場合)にプッシュされます。
- PTF内外へのデータの一括移動
- 並列度は、PTFおよび問合せに指定されたパーティション(ある場合)のタイプに基づきます

12.6.6 Skip_col多相テーブル・ファンクションの例

このPTF例は、Row Semantics、Describeのみ、パッケージ・テーブル・ファンクション、およびオーバーロード機能を示しています。

関連項目:

その他の多相テーブル・ファンクションの(PTF)の例については、Oracle Database PL/SQLパッケージおよびタイプ・リファレンス

例12-36 Skip_col多相テーブル・ファンクションの例

skip_col多相テーブル・ファンクション(PTF)は、PTF入力引数で指定した列を除く、表内のすべての列を返します。skip_col PTFは列名(overload 1)または列データ型(overload 2)に基づいて列をスキップします。

> Live SQL:
>
> このサンプルは、18c Skip_col 多相テーブル・ファンクションの Oracle Live SQL で表示および実行できます

skip_col多相テーブル・ファンクション(PTF)に対して、DESCRIBEファンクションを含むskip_col_pkgという名前の実装パッケージを作成します。PTFによって生成された行の形状を決定するために、DESCRIBEファンクションが起動されます。DBMS_TF.DESCRIBE_T表を戻します。オーバーロードされています。FETCH_ROWSプロシージャは、 指定された行のサブセットに対して関連する新しい列値を生成する必要があるため、必須ではありません。

```
CREATE PACKAGE skip_col_pkg AS
  -- OVERLOAD 1: Skip by name --
  FUNCTION skip_col(tab TABLE,
                    col COLUMNS)
           RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
  FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T,
                    col        DBMS_TF.COLUMNS_T)
           RETURN DBMS_TF.DESCRIBE_T;
  -- OVERLOAD 2: Skip by type --
  FUNCTION skip_col(tab       TABLE,
                    type_name VARCHAR2,
                    flip      VARCHAR2 DEFAULT 'False')
           RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
  FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T,
                    type_name        VARCHAR2,
                    flip             VARCHAR2 DEFAULT 'False')
           RETURN DBMS_TF.DESCRIBE_T;
END skip_col_pkg;
```

多相テーブル・ファンクション定義を含む実装パッケージ本体を作成します。

```
CREATE PACKAGE BODY skip_col_pkg AS
/* OVERLOAD 1: Skip by name
 * Package PTF name:  skip_col_pkg.skip_col
 * Standalone PTF name: skip_col_by_name
 *
 * PARAMETERS:
 * tab - The input table
 * col - The name of the columns to drop from the output
 *
```

```
 * DESCRIPTION:
 *   This PTF removes all the input columns listed in col from the output
 *   of the PTF.
*/
 FUNCTION  describe(tab IN OUT DBMS_TF.TABLE_T,
                    col        DBMS_TF.COLUMNS_T)
            RETURN DBMS_TF.DESCRIBE_T
  AS
    new_cols DBMS_TF.COLUMNS_NEW_T;
    col_id   PLS_INTEGER := 1;
  BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
      FOR j IN 1 .. col.count() LOOP
        tab.column(i).PASS_THROUGH := tab.column(i).DESCRIPTION.NAME != col(j);
        EXIT WHEN NOT tab.column(i).PASS_THROUGH;
      END LOOP;
    END LOOP;
    RETURN NULL;
  END;
/* OVERLOAD 2: Skip by type
 * Package PTF name:  skip_col_pkg.skip_col
 * Standalone PTF name: skip_col_by_type
 *
 * PARAMETERS:
 *   tab       - Input table
 *   type_name - A string representing the type of columns to skip
 *   flip      - 'False' [default] => Match columns with given type_name
 *               otherwise         => Ignore columns with given type_name
 *
 * DESCRIPTION:
 *   This PTF removes the given type of columns from the given table.
*/
  FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T,
                    type_name        VARCHAR2,
                    flip             VARCHAR2 DEFAULT 'False')
           RETURN DBMS_TF.DESCRIBE_T
  AS
    typ CONSTANT VARCHAR2(1024) := UPPER(TRIM(type_name));
  BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
       tab.column(i).PASS_THROUGH :=
         CASE UPPER(SUBSTR(flip,1,1))
           WHEN 'F' THEN DBMS_TF.column_type_name(tab.column(i).DESCRIPTION)!=typ
           ELSE          DBMS_TF.column_type_name(tab.column(i).DESCRIPTION) =typ
         END /* case */;
    END LOOP;
    RETURN NULL;
  END;
END skip_col_pkg;
```

overload 1にskip_col_by_nameという名前のスタンドアロン多相テーブル・ファンクションを作成します。TABLE型の正式引数を1つ指定し、PTFの戻り型をTABLEとして指定し、行セマンティクスPTFタイプを指定し、使用するPTF実装パッケージをskip_col_pkgに指定します。

```
CREATE FUNCTION skip_col_by_name(tab TABLE,
                                 col COLUMNS)
                  RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
```

overload 2にskip_col_by_typeという名前のスタンドアロン多相テーブル・ファンクションを作成します。TABLE型の正式引数を1つ指定し、PTFの戻り型をTABLEとして指定し、行セマンティクスPTFタイプを指定し、使用するPTF実装パッケージをskip_col_pkgに指定します。

```
CREATE FUNCTION skip_col_by_type(tab TABLE,
                                 type_name VARCHAR2,
                                 flip VARCHAR2 DEFAULT 'False')
                 RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
```

型がNUMBERでない列のみをSCOTT.DEPT表からレポートするために、パッケージskip_col PTF (overload 1)を起動します。

```
SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number');

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

SCOTT.DEPT表からスタンドアロンskip_col_by_type PTFを起動して、型がNUMBERでない列のみをレポートすると、同じ結果が得られます。

```
SELECT * FROM skip_col_by_type(scott.dept, 'number');

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

SCOTT.DEPT表から型がNUMBERの列のみをレポートするために、パッケージskip_col PTF (overload 2)を起動します。

```
SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number', flip => 'True');

   DEPTNO
----------
        10
        20
        30
        40
```

SCOTT.DEPT表からスタンドアロンskip_col_by_type PTFを起動して、型がNUMBERの列のみをレポートすると、同じ結果が得られます。

```
SELECT * FROM skip_col_by_type(scott.dept, 'number', flip => 'True');

   DEPTNO
----------
        10
        20
        30
        40
```

パッケージskip_col PTFを起動して、SCOTT.EMP表から部門20のCOMM、HIREDATEおよびMGRを除くすべての従業員をレポートします。

```
SELECT *
FROM skip_col_pkg.skip_col(scott.emp, COLUMNS(comm, hiredate, mgr))
WHERE deptno = 20;

    EMPNO ENAME      JOB              SAL     DEPTNO
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            800         20
      7566 JONES      MANAGER         2975         20
      7788 SCOTT      ANALYST         3000         20
      7876 ADAMS      CLERK           1100         20
      7902 FORD       ANALYST         3000         20
```

12.6.7 To_doc多相テーブル・ファンクションの例

to_doc PTFの例では、指定された列のリストを1つのドキュメント列に結合します。

例12-37 To_doc多相テーブル・ファンクションの例

to_doc PTFは、列のリストをJSONオブジェクトのように構築されたドキュメント列に結合します。

Live SQL:

このサンプルは、18c To_doc 多相テーブル・ファンクションで Oracle Live SQL で表示および実行できます

多相テーブル・ファンクション(PTF)のDESCRIBEファンクションおよびFETCH_ROWSプロシージャを含む実装パッケージto_docを作成します。

PTFパラメータは、次のとおりです。

- tab : 入力表(tabパラメータは表記述子レコード型のDBMS_TF.TABLE_T型です)
- cols (オプション) : ドキュメントを変換する列のリスト。(colsパラメータは、列記述子レコード型のDBMS_TF.COLUMNS_Tです)

```
CREATE PACKAGE to_doc_p AS
   FUNCTION describe(tab      IN OUT DBMS_TF.TABLE_T,
                     cols     IN     DBMS_TF.COLUMNS_T DEFAULT NULL)
                       RETURN DBMS_TF.DESCRIBE_T;

   PROCEDURE fetch_rows;
END to_doc_p;
```

DESCRIBEファンクションおよびFETCH_ROWSプロシージャを含むパッケージを作成します。出力行セットにDOCUMENTという名前の新しい列を作成するには、FETCH_ROWSプロシージャが必要です。DESCRIBEファンクションは、入力表記述子TABLE_Tで注釈を付けることによって読取り列を指定します。指定された読取り列のみがフェッチされるため、FETCH_ROWS時の処理に使用できます。問合せでのPTFの起動は、COLUMNS疑似演算子を使用して、問合せがPTFに読み取らせる列を指定できます。この情報はDESCRIBEファンクションに渡され、このファンクションによってCOLUMN_T.FOR_READブール・フラグが設定されます。読取り

列には、スカラーSQLデータ型のみを使用できます。COLUMN_T.PASS_THROUGHブール・フラグは、変更なしでPTFの入力表から出力に渡される列を示します。

```
CREATE PACKAGE BODY to_doc_p AS

FUNCTION describe(tab      IN OUT DBMS_TF.TABLE_T,
                  cols     IN     DBMS_TF.COLUMNS_T DEFAULT NULL)
                    RETURN DBMS_TF.DESCRIBE_T AS
BEGIN
  FOR i IN 1 .. tab.column.count LOOP
         CONTINUE WHEN NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).DESCRIPTION.TYPE);

          IF cols IS NULL THEN
             tab.column(i).FOR_READ     := TRUE;
             tab.column(i).PASS_THROUGH := FALSE;
             CONTINUE;
           END IF;

          FOR j IN 1 .. cols.count LOOP
            IF (tab.column(i).DESCRIPTION.NAME = cols(j)) THEN
                tab.column(i).FOR_READ     := TRUE;
                tab.column(i).PASS_THROUGH := FALSE;
            END IF;
          END LOOP;

  END LOOP;

  RETURN DBMS_TF.describe_t(new_columns => DBMS_TF.COLUMNS_NEW_T(1 =>
                                  DBMS_TF.COLUMN_METADATA_T(name =>'DOCUMENT')));
END;

 PROCEDURE fetch_rows AS
      rst DBMS_TF.ROW_SET_T;
      col DBMS_TF.TAB_VARCHAR2_T;
      rct PLS_INTEGER;
 BEGIN
      DBMS_TF.GET_ROW_SET(rst, row_count => rct);
      FOR rid IN 1 .. rct LOOP
               col(rid) := DBMS_TF.ROW_TO_CHAR(rst, rid);
      END LOOP;
      DBMS_TF.PUT_COL(1, col);
 END;

END to_doc_p;
```

スタンドアロンto_doc PTFを作成します。TABLE型の仮引数を正確に1つ指定し、PTFの戻り型をTABLEとして指定し、行セマンティクスPTF型を指定し、PTF実装パッケージがto_doc_pを使用するように指定します。

```
CREATE FUNCTION to_doc(
                 tab  TABLE,
                            cols  COLUMNS DEFAULT NULL)
                            RETURN TABLE
    PIPELINED ROW POLYMORPHIC USING to_doc_p;
```

to_doc PTFを起動して、表SCOTT.DEPTのすべての列を結合された1つのDOCUMENT列として表示します。

```
SELECT * FROM to_doc(scott.dept);

DOCUMENT
--------------------------------------------------
{"DEPTNO":10, "DNAME":"ACCOUNTING", "LOC":"NEW YORK"}
{"DEPTNO":20, "DNAME":"RESEARCH", "LOC":"DALLAS"}
{"DEPTNO":30, "DNAME":"SALES", "LOC":"CHICAGO"}
{"DEPTNO":40, "DNAME":"OPERATIONS", "LOC":"BOSTON"}
```

部門10および30のすべての従業員について、DEPTNO、ENAMEおよびDOCUMENT列をDEPTNOおよびENAMEの順で表示します。to_doc PTFをCOLUMNS擬似演算子で起動し、表SCOTT.EMPの列EMPNO、JOB、MGR、HIREDATE、SALおよびCOMMを選択します。PTFは、これらの列をDOCUMENT列に結合します。

```
SELECT deptno, ename, document
FROM   to_doc(scott.emp, COLUMNS(empno,job,mgr,hiredate,sal,comm))
WHERE  deptno IN (10, 30)
ORDER BY 1, 2;

DEPTNO ENAME      DOCUMENT
------ ---------- ------------------------------------------------------------------------------------
    10 CLARK      {"EMPNO":7782, "JOB":"MANAGER", "MGR":7839, "HIREDATE":"09-JUN-81", "SAL":2450}
    10 KING       {"EMPNO":7839, "JOB":"PRESIDENT", "HIREDATE":"17-NOV-81", "SAL":5000}
    10 MILLER     {"EMPNO":7934, "JOB":"CLERK", "MGR":7782, "HIREDATE":"23-JAN-82", "SAL":1300}
    30 ALLEN      {"EMPNO":7499, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"20-FEB-81", "SAL":1600,
"COMM":300}
    30 BLAKE      {"EMPNO":7698, "JOB":"MANAGER", "MGR":7839, "HIREDATE":"01-MAY-81", "SAL":2850}
    30 JAMES      {"EMPNO":7900, "JOB":"CLERK", "MGR":7698, "HIREDATE":"03-DEC-81", "SAL":950}
    30 MARTIN     {"EMPNO":7654, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"28-SEP-81", "SAL":1250,
"COMM":1400}
    30 TURNER     {"EMPNO":7844, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"08-SEP-81", "SAL":1500,
"COMM":0}
    30 WARD       {"EMPNO":7521, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"22-FEB-81", "SAL":1250,
"COMM":500}
```

Eという名前の副問合せを使用して、DOC_ID列およびDOCUMENT列を表示します。すべての事務員従業員、給与、部門および部門の場所をレポートします。to_doc PTFを使用して、NAME、SAL、DEPTNOおよびLOC列をDOCUMENT列に結合します。

```
WITH e AS (
       SELECT ename name, sal, deptno, loc
             FROM scott.emp NATURAL JOIN scott.dept
        WHERE job = 'CLERK')
          SELECT ROWNUM doc_id, t.*
            FROM to_doc(e) t;

    DOC_ID DOCUMENT
---------- ----------------------------------------------------------
         1 {"NAME":"MILLER", "SAL":1300, "DEPTNO":10, "LOC":"NEW YORK"}
         2 {"NAME":"SMITH", "SAL":800, "DEPTNO":20, "LOC":"DALLAS"}
         3 {"NAME":"ADAMS", "SAL":1100, "DEPTNO":20, "LOC":"DALLAS"}
         4 {"NAME":"JAMES", "SAL":950, "DEPTNO":30, "LOC":"CHICAGO"}
```

副問合せブロックを使用して、DOCUMENT列に変換されたc1、c2、c3列の値を表示します。

```
WITH t(c1,c2,c3)  AS (
    SELECT NULL, NULL, NULL FROM dual
    UNION ALL
```

```
    SELECT    1, NULL, NULL FROM dual
    UNION ALL
    SELECT NULL,    2, NULL FROM dual
    UNION ALL
    SELECT    0, NULL,    3 FROM dual)
  SELECT *
    FROM to_doc(t);

DOCUMENT
---------------
{}
{"C1":1}
{"C2":2}
{"C1":0, "C3":3}
```

部門30のすべての従業員について、プロパティ名ENAMEおよびCOMMを持つメンバーの値を表示します。SCOTT.EMP表のPTF起動レポートにより、JSON_VALUEファンクションへの入力として使用できるDOCUMENT列が生成されます。このファンクションは、一部のJSONデータからスカラー値を選択します。

```
SELECT JSON_VALUE(document, '$.ENAME') ename,
       JSON_VALUE(document, '$.COMM')  comm
FROM   to_doc(scott.emp)
WHERE  JSON_VALUE(document, '$.DEPTNO') = 30;

ENAME      COMM
---------- ----
ALLEN      300
WARD       500
MARTIN     1400
BLAKE
TURNER     0
JAMES
```

12.6.8 Implicit_echo多相表関数の例

implicit_echo PTFの例では、多相表関数とDESCRIBE関数が同じパッケージ内で定義されている場合、USING句がオプションであることを示しています。

例12-38 Implicit_echo多相表関数の例

implicit_echo PTFは表と列を受け入れ、同じ値で新しい列を作成します。

このPTFは、入力表タブの列を返し、colsにリストされた、列名に"ECHO_"という接頭辞が付いた列を追加します。

DESCRIBEファンクション、implicit_echo多相表関数(PTF)およびFETCH_ROWSプロシージャが含まれている実装パッケージimplicit_echo_packageを作成します。

```
CREATE PACKAGE implicit_echo_package AS
  prefix   DBMS_ID := '"ECHO_';
  FUNCTION DESCRIBE(tab   IN OUT DBMS_TF.TABLE_T,
                    cols  IN     DBMS_TF.COLUMNS_T)
           RETURN DBMS_TF.DESCRIBE_T;

  PROCEDURE FETCH_ROWS;
```

```
  -- PTF FUNCTION: WITHOUT USING CLAUSE --
  FUNCTION implicit_echo(tab TABLE, cols COLUMNS)
           RETURN TABLE PIPELINED ROW POLYMORPHIC;
END implicit_echo_package;
```

読み取る入力表パラメータと列パラメータを含むDESCRIBEファンクションが含まれているパッケージを作成します。このファンクションは、多相表関数によって作成される行のタイプを決定する場合に呼び出されます。このファンクションは、表DBMS_TF.DESCRIBE_Tを戻します。FETCH_ROWSプロシージャは、出力行セット内に"ECHO_"という接頭辞付いた新しい列とともに指定された読取り列を生成するために必要になります。implicit_echoはPTF関数であり、2つの引数(tabとcols)が含まれています。それらの値は、問合せによって取得され、この情報がDESCRIBEファンクションに渡されます。RowセマンティクスではPTFタイプを指定しますが、USING句は指定されていません。このファンクションは、SQL問合せから起動されます。

PTF定義を含める実装パッケージ本体implicit_echo_packageを作成します。

```
CREATE PACKAGE BODY implicit_echo_package AS
FUNCTION DESCRIBE(tab  IN  OUT DBMS_TF.TABLE_T,
                  cols IN      DBMS_TF.COLUMNS_T)
          RETURN DBMS_TF.DESCRIBE_T
AS
  new_cols DBMS_TF.COLUMNS_NEW_T;
  col_id   PLS_INTEGER := 1;
BEGIN
 FOR i in 1 .. tab.column.COUNT LOOP

   FOR j in 1 .. cols.COUNT LOOP

     IF (tab.column(i).description.name = cols(j)) THEN

       IF (NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).description.type)) THEN
            RAISE_APPLICATION_ERROR(-20102, 'Unsupported column type['||
                                    tab.column(i).description.type||']');
       END IF;
       tab.column(i).for_read := TRUE;
       new_cols(col_id)       := tab.column(i).description;
       new_cols(col_id).name  := prefix||
                                 REGEXP_REPLACE(tab.column(i).description.name,
                                                                    '^"|"$');
       col_id                 := col_id + 1;
       EXIT;
     END IF;
    END LOOP;
 END LOOP;
/* VERIFY ALL COLUMNS WERE FOUND */
 IF (col_id - 1 != cols.COUNT) then
    RAISE_APPLICATION_ERROR(-20101,'Column mismatch['||col_id-1||'],
                                                ['||cols.COUNT||']');
 END IF;
 RETURN DBMS_TF.DESCRIBE_T(new_columns => new_cols);
END;
 PROCEDURE FETCH_ROWS AS
         rowset DBMS_TF.ROW_SET_T;
 BEGIN
         DBMS_TF.GET_ROW_SET(rowset);
         DBMS_TF.PUT_ROW_SET(rowset);
 END;
```

```
END implicit_echo_package;
```

PTFを起動すると、表SCOTT.EMPのENAME列が表示されます。また、同じ値を持つ別の列ECHO_ENAMEも表示されます。

```
SELECT ENAME, ECHO_ENAME
FROM implicit_echo_package.implicit_echo(SCOTT.EMP, COLUMNS(SCOTT.ENAME));

ENAME      ECHO_ENAME
---------- ----------
SMITH      SMITH
ALLEN      ALLEN
WARD       WARD
JONES      JONES
MARTIN     MARTIN
BLAKE      BLAKE
CLARK      CLARK
SCOTT      SCOTT
KING       KING
TURNER     TURNER
ADAMS      ADAMS
JAMES      JAMES
FORD       FORD
MILLER     MILLER
```

12.7 大規模な表のパラレルでの更新

DBMS_PARALLEL_EXECUTEパッケージを使用すると、次の2つのステップで、大規模な表のデータをパラレルで増分更新できます。

1. 表の行をグループ化して小さなチャンクに分けます。
2. 必要なUPDATE文を並列でチャンクに適用し、1チャンクの処理が終了するたびにコミットします。

大量のデータを更新する場合は、常にこの方法をお薦めします。この方法には次のメリットがあります。

- 表全体のロックではなく、相対的に短い時間で一度に1セットの行のみをロックする。
- 操作全体が完了する前に問題が発生してもそれまでに処理した内容が失われない。
- ロールバック用の領域の消費が削減されます。
- パフォーマンスが向上します。

関連項目:

DBMS_PARALLEL_EXECUTEパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

12.8 ユーザー定義の識別子に関するデータの収集

PL/Scopeは、PL/SQLソース・テキストからPL/SQLおよびSQLの識別子ならびにSQL文に関するデータを抽出、編成および格納します。静的データ・ディクショナリ・ビュー*_IDENTIFIERSおよび*_STATEMENTSを使用して、識別子と文のデータを取り出すことができます。

関連項目:

- PLSQL_SETTINGSパラメータの詳細は、PL/SQLユニットおよびコンパイル・パラメータを参照してください
- PL/Scopeの使用に関する詳細は、Oracle Database開発ガイドを参照してください

12.9 PL/SQLプログラムのプロファイルおよびトレース

大規模なPL/SQLプログラムでのパフォーマンスの問題を切り出すことができるように、PL/SQLには、PL/SQLパッケージとして実装される次のツールが用意されています。

例12-1 プロファイル・ツールおよびトレース・ツールの概要

ツール	パッケージ	説明
プロファイラ・インタフェース	DBMS_PROFILER	PL/SQL プログラムが各行および各サブプログラムに費やす時間を計算します。 プロファイル対象のユニットに対する CREATE 権限が必要です。 ランタイム統計をデータベース表に保存します。このデータベース表は、問い合せることができます。
トレース・インタフェース	DBMS_TRACE	サブプログラムの実行順序をトレースできます。 トレースするサブプログラムおよびトレース・レベルを指定できます。 ランタイム統計をデータベース表に保存します。このデータベース表は、問い合せることができます。
PL/SQL 階層型プロファイラ	DBMS_HPROF	PL/SQL プログラムの動的な実行プログラム・プロファイルをサブプログラムの起動ごとにまとめてレポートします。SQL 実行時間と PL/SQL 実行時間を別々に示します。 特別なソースまたはコンパイル時の準備は必要ありません。 HTML 形式でレポートを生成します。カスタム・レポートの生成用にプロファイラ・データと結果をリレーショナル形式でデータベース表に格納する(サードパーティ・ツールで提供されるような)オプションを提供します。
SQL トレース	DBMS_APPLICATION_INFO	Oracle Trace および SQL トレース機能を持つ

ツール	パッケージ	説明
		DBMS_APPLICATION_INFO パッケージを使用して、実行しているモジュール名またはトランザクションをデータベースに記録できます。この記録は、後で行う様々なモジュールおよびデバッグのパフォーマンスを追跡するときに使用されます。
PL/SQL 基本ブロック・カバレッジ	DBMS_PLSQL_CODE_COVERAGE	基本ブロック・カバレッジ・データを収集して分析します。
コール・スタック・ユーティリティ	UTL_CALL_STACK	現在実行中のサブプログラムに関する情報(サブプログラム名、ユニット名、所有者名、エディション名、エラー・スタック情報など)を提供するため、これらの情報からより露出度の高いエラー・ログやアプリケーション実行トレースを作成できます。

関連トピック

- DBMS_APPLICATION_INFOパッケージの詳細は、『Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス』を参照してください
- DBMS_HPROFパッケージの詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください
- DBMS_PLSQL_CODE_COVERAGEパッケージの詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください
- DBMS_PROFILERパッケージの詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください
- DBMS_TRACEパッケージの詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください

  UTL_CALL_STACKパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。
- COVERAGE PRAGMAの構文およびセマンティクスの詳細は、COVERAGEプラグマを参照してください
- PL/SQL基本ブロック・カバレッジの使用方法の詳細は、『Oracle Database開発ガイド』を参照してください
- PL/SQL階層プロファイラの詳細は、Oracle Database開発ガイドを参照してください
- ストアド・サブプログラムの分析およびデバッグの詳細は、Oracle Database開発ガイドを参照してください

12.10 システム固有の実行のためのPL/SQLユニットのコンパイル

通常、PL/SQLユニットをコンパイルして、システム固有のコード(プロセッサに依存するシステム・コード)にすると、PL/SQLユニットをスピードアップできます。システム固有のコードは、SYSTEM表領域に格納されます。

Oracle Databaseが提供するユニットを含め、すべてのタイプのPL/SQLユニットをネイティブ・コンパイルできます。

ネイティブ・コンパイルされたプログラム・ユニットは、共有サーバー構成(以前のマルチスレッド・サーバー)やOracle Real Application Clusters(Oracle RAC)などのすべてのサーバー環境で動作します。

ほとんどのプラットフォームで、PL/SQLネイティブ・コンパイルに特別な設定またはメンテナンスは必要ありません。一部のプラットフォームでは、DBAによるオプションの構成が必要な場合もあります。

関連項目:

- データベースの構成の詳細は、『Oracle Database管理者ガイド』を参照してください。
- ご使用のプラットフォームの詳細は、プラットフォーム固有の構成ドキュメントを参照してください。

PL/SQLのネイティブ・コンパイルを有効にしてパフォーマンスが改善される度合いは、テストによって確認できます。

PL/SQLのネイティブ・コンパイルを使用するとデータベース操作で大幅にパフォーマンスが改善されると判断した場合、ネイティブ・モードでデータベース全体をコンパイルすることをお薦めします(これを行うには、DBA権限が必要です)。これによって、独自のコードと、Oracle Database提供のPL/SQLパッケージへのコールの両方がスピードアップします。

ここでのトピック

- PL/SQLのネイティブ・コンパイルを使用するかどうかの判断
- PL/SQLのネイティブ・コンパイルの動作
- 依存性、無効化および再評価
- PL/SQLのネイティブ・コンパイルで使用する新しいデータベースの設定*
- PL/SQLネイティブ・コンパイルまたは解釈コンパイルで使用するデータベース全体のコンパイル*

* DBA権限が必要です。

12.10.1 PL/SQLのネイティブ・コンパイルを使用するかどうかの判断

PL/SQLユニットをネイティブ・モードでコンパイルするか、解釈済モードでコンパイルするかは、開発サイクルの段階およびプログラム・ユニットの内容によって決まります。

プログラム・ユニットをデバッグし、頻繁に再コンパイルしている場合は、解釈済モードに次のメリットがあります。

- (ネイティブ・モードでコンパイルされたプログラム・ユニットではなく)解釈済モードでコンパイルされたプログラム・ユニットに対しては、PL/SQLデバッグ・ツールを使用できます。
- 解釈済モードのコンパイルは、ネイティブ・モードでのコンパイルより高速です。

開発のデバッグ・フェーズ後、ネイティブ・モードでPL/SQLユニットをコンパイルするかどうかを判断する際は次のことを考慮してください。

- PL/SQLのネイティブ・コンパイルによって、計算集中型のプロシージャ操作のパフォーマンスは大幅に改善されます。この例には、データ・ウェアハウス・アプリケーションや、サーバー側でデータを大幅に変換して表示するアプリケーションなどがあります。
- PL/SQLのネイティブ・コンパイルによって、SQLの実行にほとんどの時間を費やすPL/SQLサブプログラムのパフォーマンスはほとんど改善されません。
- 多数のプログラム・ユニット(通常は15,000以上)をシステム固有の実行用にコンパイルし、同時にアクティブにすると、大量の共有メモリーが必要になるため、システムのパフォーマンスに影響することがあります。

12.10.2 PL/SQLのネイティブ・コンパイルの動作

ネイティブ・コンパイルを使用しない場合、PL/SQLユニットのPL/SQL文は、中間形式のシステム・コードにコンパイルされます。このコードは、カタログに格納され、実行時に解釈されます。

PL/SQLのネイティブ・コンパイルを使用する場合、PL/SQLユニットのPL/SQL文はシステム固有のコードにコンパイルされ、カタログに格納されます。システム固有のコードは、実行時に解釈する必要がないため、高速で実行されます。

ネイティブ・コンパイルはPL/SQL文にのみ適用されるため、SQL文のみを使用するPL/SQLユニットの場合、ネイティブ・コンパイルしても高速に実行されない可能性がありますが、対応する解釈済コードより低速になることはありません。コンパイルされたコードと解釈済コードは同じライブラリをコールするため、アクションは同じです。

ネイティブ・コンパイルされたPL/SQLユニットを初めて実行すると、このユニットは、SYSTEM表領域から共有メモリーにフェッチされます。プログラム・ユニットを起動するセッションの数に関係なく、共有メモリーにはそのコピーが1つのみ含まれています。プログラム・ユニットが使用されていない場合、そのプログラム・ユニットが使用している共有メモリーを解放してメモリー・ロードを軽減できます。

ネイティブ・コンパイルされたサブプログラムと解釈済サブプログラムは相互に起動し合うことができます。

PL/SQLのネイティブ・コンパイルは、Oracle Real Application Clusters(Oracle RAC)環境では透過的に動作します。

PLSQL_CODE_TYPEコンパイル・パラメータによって、PL/SQLコードをネイティブ・コンパイルするか解釈済にするかが決まります。このコンパイル・パラメータの詳細は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください。

12.10.3 依存性、無効化および再評価

無効化したPL/SQLモジュールでは、再コンパイルが自動的に行われます。たとえば、ネイティブ・コンパイルされたPL/SQLサブプログラムが依存するオブジェクトが変更されると、サブプログラムは無効になります。同じサブプログラムが次にコールされたとき、データベースはサブプログラムを自動的に再コンパイルします。PLSQL_CODE_TYPE設定はサブプログラムごとにライブラリ・ユニットに格納されるため、自動再コンパイルではこの格納された設定をコード型として使用します。

明示的な再コンパイルで、格納されているPLSQL_CODE_TYPE設定が使用されるとはかぎりません。格納されている設定を明示的な再コンパイルで使用する条件は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください。

12.10.4 PL/SQLのネイティブ・コンパイルで使用する新しいデータベースの設定

DBA権限を持っている場合、コンパイル・パラメータPLSQL_CODE_TYPEをNATIVEに設定して、PL/SQLネイティブ・コンパイル用に新しいデータベースを設定できます。多くのデータベース操作で使用されるOracle Database提供のPL/SQLパッケージで、パフォーマンスが改善します。

ノート:

データベース全体を NATIVE でコンパイルする場合は、システム・レベルで PLSQL_CODE_TYPE を設定することをお薦めします。

12.10.5 PL/SQLネイティブ・コンパイルまたは解釈コンパイルで使用するデータベース全体のコンパイル

DBA権限を持っている場合、この項で説明するプロセスで、dbmsupgnv.sqlおよびdbmsupgin.sqlのスクリプトを使用して、既存のデータベースのすべてのPL/SQLモジュールをそれぞれNATIVEまたはINTERPRETEDに再コンパイルできます。この変換を実行する前に、「PL/SQLのネイティブ・コンパイルを使用するかどうかの判断」の内容に目を通してください。

ノート:

- データベース全体を NATIVE でコンパイルする場合は、システム・レベルで PLSQL_CODE_TYPE を設定することをお薦めします。
- Database Vault が有効で、Database Vault の管理者から DV_PATCH_ADMIN ロールを付与された場合にのみ、dbmsupgnv.sql を実行できます。

ネイティブ・コンパイルへの変換で、dbmsupgnv.sqlによってTYPE仕様部をNATIVEに再コンパイルすることはできません。これらの仕様部には実行可能なコードが含まれていないためです。

パッケージ仕様部に実行可能なコードが含まれることはほとんどないため、NATIVEにコンパイルしても実行時の利点は得られません。dbmsupgnv.sqlスクリプトでTRUEコマンドライン・パラメータを使用すると、NATIVEへの再コンパイルからパッケージ仕様部を除外して、変換処理にかかる時間を節約できます。

dbmsupgin.sqlスクリプトで解釈コンパイルに変換するときは、パラメータは指定できず、PL/SQLユニットを除外することはできません。

ノート:

次の手順では、ネイティブ・コンパイルへの変換について説明します。すべての PL/SQL モジュールを解釈コンパイルに再コンパイルする必要がある場合は、ステップを次のように変更してください。

- 1 番目のステップをスキップします。
- PLSQL_CODE_TYPE コンパイル・パラメータを NATIVE ではなく INTERPRETED に設定します。
- dbmsupgnv.sql スクリプトを dbmsupgin.sql スクリプトに置き換えます。

1. PL/SQLテスト・ユニットをコンパイルできることを確認します。たとえば:

```
ALTER PROCEDURE my_proc COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
```

2. アプリケーション・サービス、リスナーおよびデータベースを停止します。
   - Forms Process、Web Server、Reports Server、Concurrent Manager Serverなど、すべてのアプリケーション・サービスを停止します。すべてのアプリケーション・サービスを停止した後で、データベースへのすべての接続が終了したことを確認します。

- データベースのTNSリスナーを停止して、新規接続が作成されないようにします。
- ユーザーSYSとして、通常モードまたは即時モードでデータベースを停止します。『Oracle Database管理者ガイド』を参照してください。

3. コンパイル・パラメータ・ファイルでPLSQL_CODE_TYPEをNATIVEに設定します。データベースでサーバー・パラメータ・ファイルを使用している場合は、データベースを起動してからこの設定を行ってください。

   PLSQL_CODE_TYPEの値は、このステップのPL/SQLユニットの変換には影響を与えません。ただし、この手順以降にコンパイルしたユニットは影響を受けるため、ここで必要なコンパイル・タイプを明示的に設定してください。

4. UPGRADEオプションを使用して、データベースをアップグレード・モードで起動します。SQL*PlusのSTARTUPの詳細は、『SQL*Plusユーザーズ・ガイドおよびリファレンス』を参照してください。
5. 次のコードを実行して、無効なPL/SQLユニットを一覧表示します。SQL SPOOL文を使用すると、問合せの出力を今後の参照のために保存しておくことができます。

```
-- To save the output of the query to a file:
  SPOOL pre_update_invalid.log
SELECT o.OWNER, o.OBJECT_NAME, o.OBJECT_TYPE
FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
WHERE o.OBJECT_NAME = s.NAME AND o.STATUS='INVALID';
-- To stop spooling the output: SPOOL OFF
```

   Oracleが提供するユニットが無効な場合は、それらのユニットを再コンパイルして有効にします。たとえば:

```
ALTER PACKAGE SYS.DBMS_OUTPUT COMPILE BODY REUSE SETTINGS;
```

   ユニットを有効にできない場合は、今後の解決のためにスプール・ログを保存してから処理を続行します。

6. 次の問合せを実行して、NATIVEおよびINTERPRETEDでコンパイルされるオブジェクトの数を確認します(出力を保存するには、SQLのSPOOL文を使用します)。

```
SELECT TYPE, PLSQL_CODE_TYPE, COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE;
```

   NULL plsql_code_typeのオブジェクトは特別な内部オブジェクトであり、無視してかまいません。

7. ユーザーSYSとして$ORACLE_HOME/rdbms/admin/dbmsupgnv.sqlスクリプトを実行して、すべてのPL/SQLユニットのディクショナリ表でplsql_code_type設定をNATIVEに更新します。この処理によってもユニットが無効化されます。パッケージ仕様部を除外するときは、スクリプトでTRUEを使用し、パッケージ仕様部を含めるときは、FALSEを使用します。

   この更新は、データベースがUPGRADEモードの場合に実行する必要があります。スクリプトを使用すると、更新を正常に完了したり、すべての変更をロールバックすることもできます。

8. データベースを停止し、NORMALモードで再起動します。
9. 問題の発生を回避するため、utlrp.sqlスクリプトを実行する前に他のセッションに接続しないことをお薦めします。このためには、次の文を実行します。

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
```

10. ユーザーSYSとして$ORACLE_HOME/rdbms/admin/utlrp.sqlスクリプトを実行します。このスクリプトでは、デフォルトの並列度を使用してすべてのPL/SQLモジュールを再コンパイルします。並列度を明示的に設定する方法に

ついては、スクリプトのコメントを参照してください。

スクリプトが異常終了した場合は、utlrp.sqlスクリプトを再実行して残りの無効なPL/SQLモジュールを再コンパイルします。

11. コンパイルが正常に完了したら、ステップ5の問合せを使用して、他に無効なPL/SQLユニットがないかどうかを確認します。問合せの出力をpost_upgrade_invalid.logファイルにスプールしておくと、以前に作成したpre_upgrade_invalid.logファイルがあれば、スプールした内容と比較できます。
12. ステップ6の問合せを再実行します。dbmsupgnv.sqlで再コンパイルした場合は、除外したTYPE仕様部およびパッケージ仕様部を除くすべてのPL/SQLユニットがNATIVEであることを確認します。dbmsupgin.sqlで再コンパイルした場合は、すべてのPL/SQLユニットがINTERPRETEDであることを確認します。
13. データベースの制限付きセッション・モードを無効にしてから、前に停止したサービスを開始します。制限付きセッション・モードを無効にするには、次の文を使用します。

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

13 PL/SQLの言語要素

PL/SQLの言語要素の構文と意味の概要、および例と関連トピックへのリンクを示します。

構文図の読み方は、『Oracle Database SQL言語リファレンス』を参照してください。

ここでのトピック

- ACCESSIBLE BY句
- AGGREGATE句
- 代入文
- AUTONOMOUS_TRANSACTIONプラグマ
- 基本LOOP文
- ブロック
- コール仕様
- CASE文
- CLOSE文
- コレクション・メソッドの起動
- コレクション変数の宣言
- コメント
- COMPILE句
- 定数の宣言
- CONTINUE文
- COVERAGEプラグマ
- カーソルFOR LOOP文
- カーソル変数の宣言
- データ型属性
- DEFAULT COLLATION句
- DELETE文の拡張機能
- DEPRECATEプラグマ
- DETERMINISTIC句
- 要素の仕様
- EXCEPTION_INITプラグマ
- 例外宣言
- 例外ハンドラ

- EXECUTE IMMEDIATE文
- EXIT文
- 明示カーソルの宣言および定義
- 式
- FETCH文
- FOR LOOP文
- FORALL文
- 仮パラメータの宣言
- ファンクションの宣言および定義
- GOTO文
- IF文
- 暗黙カーソル属性
- INLINEプラグマ
- 実行者権限および定義者権限句
- INSERT文の拡張機能
- 名前付きカーソル属性
- NULL文
- OPEN文
- OPEN FOR文
- PARALLEL_ENABLE句
- PIPE ROW文
- PIPELINED句
- プロシージャの宣言および定義
- RAISE文
- レコード変数の宣言
- RESTRICT_REFERENCESプラグマ(非推奨)
- RETURN文
- RETURNING INTO句
- RESULT_CACHE句
- %ROWTYPE属性
- スカラー変数の宣言
- SELECT INTO文

- SERIALLY_REUSABLEプラグマ
- SHARING句
- SQL_MACRO句
- SQLCODEファンクション
- SQLERRMファンクション
- %TYPE属性
- UDFプラグマ
- UPDATE文の拡張機能
- WHILE LOOP文

関連項目:

- PL/SQL言語の基礎

13.1 ACCESSIBLE BY句

ACCESSIBLE BY句は、他のユニットによるユニットまたはサブプログラムへのアクセスを制限します。

accessorリストは、アクセスできるユニットを明示的にリストします。accessorリストはパッケージ内の個々のサブプログラムで定義できます。パッケージ自体に定義されたaccessorリスト(ある場合)に加え、このリストがチェックされます。このリストは、サブプログラムへのアクセスを制限できるのみで、アクセスを広げることはできません。このコード管理機能は、内部プログラムの誤った使用を防止するために役立ちます。たとえば、1つのパッケージを再編成して、アクセス制限が必要な少数のプロシージャのパッケージと、パブリック・アクセスが必要な残りのユニットのパッケージという2つのパッケージにすることはできません。

ACCESSIBLE BY句は、オブジェクト型、オブジェクト型本体、パッケージ、およびサブプログラムの宣言で記述できます。

ACCESSIBLE BY句は、次のSQL文に指定できます。

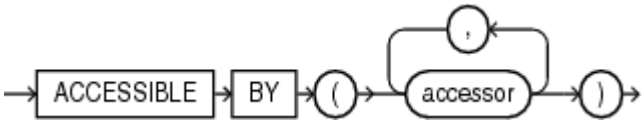
- ALTER TYPE文
- CREATE FUNCTION文
- CREATE PROCEDURE文
- CREATE PACKAGE文
- CREATE TYPE文
- CREATE TYPE BODY文

ここでのトピック

- 構文
- セマンティクス
- 使用上のノート
- 例
- 関連トピック

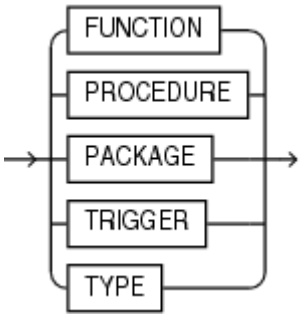
構文

accessible_by_clause ::=



accessor ::=

unit_kind ::=



セマンティクス

accessible_by_clause

accessor

[schema.]unit_name

エンティティを起動できるストアドPL/SQLユニットを指定します。

各accessorは、ACCESSIBLE BY句が含まれるエンティティにアクセスできる別のPL/SQLエンティティを指定します。

ACCESSIBLE BY句が出現した場合、その句が出現したエンティティにアクセスできるのは、その句で指定されたエンティティのみです。

accessorは、ACCESSIBLE BY句で複数回記述できます。

ACCESSIBLE BY句は、ユニットの宣言で1回のみ記述できます。

accessorで指定されたエンティティが存在する必要はありません。

ACCESSIBLE BY句のあるエンティティが起動されると、すべてのチェックが実行された後、追加のアクセス・チェックが課されます。これらのチェックには、次のものがあります。

- 起動されたユニットには、unit_nameとunit_kindが起動元のユニットと同じaccessorが含まれている必要があります。
- accessorにスキーマが含まれている場合、起動元ユニットはそのスキーマに含まれている必要があります。
- accessorにスキーマが含まれていない場合は、起動元は起動されたエンティティと同じスキーマからのものである必要があります。

unit_kind

ユニットがFUNCTION、PACKAGE、PROCEDURE、TRIGGERまたはTYPEのいずれであるかを指定します。

使用上のノート

unit_kindはオプションですが、ユニットの名前が同じ場合のあいまいさを避けるために指定することをお薦めします。たとえば、ファンクションと同じ名前でトリガーを定義することができます。

ACCESSIBLE BY句は、コールが直接行われた場合にのみアクセスを許可します。静的SQL、DBMS_SQLまたは動的SQLを介したアクセスの場合、チェックは失敗します。

パッケージ仕様部またはパッケージ本体の初期化プロシージャへのコールは、パッケージ仕様部のaccessorリストに対してチェッ

クされます。

ユニットはそれ自体に常にアクセスできます。ユニット内の項目は、同じユニット内の他の項目を参照できます。

保護されたサブプログラムへのRPCコールは、コールの有効性のチェックに使用できるコンテキストがないため、コンパイル時と実行時のいずれの場合でも常に失敗します。

保護されたサブプログラムへの条件付きコンパイル・ディレクティブからのコールは失敗します。

例

例13-1 同じスキーマの最上位プロシージャへのアクセスの制限

この例は、最上位プロシージャtop_protected_procをコールできるのは現在のスキーマのプロシージャtop_trusted_procのみであることを示しています。ユーザーがtop_proctected_procを直接コールすることはできません。

Live SQL:

この例は、Oracle Live SQL の「同じスキーマの最上位プロシージャへのアクセスの制限」で表示および実行できます

```
PROCEDURE top_protected_proc
```

ACCESSIBLE BY (PROCEDURE top_trusted_proc) AS BEGIN DBMS_OUTPUT.PUT_LINE('Executed top_protected_proc.'); END; PROCEDURE top_trusted_proc AS BEGIN DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls top_protected_proc'); top_protected_proc; END; EXEC top_trusted_proc; top_trusted_proc calls top_protected_proc Executed top_protected_proc. EXEC top_protected_proc; BEGIN top_protected_proc; END; PLS-00904: insufficient privilege to access object TOP_PROTECTED_PROC

例13-2 任意のユニット名へのアクセスの制限

この例は、PL/SQLのunit_kindがACCESSIBLE BY句で指定されていない場合、ユニット名が一致すれば、任意のunit kindからのコールが許可されることを示しています。ACCESSIBLE BY句で指定されているunit_kindが既存のオブジェクトに一致しない場合、コンパイル・エラーは発生しません。ファンクションと同じ名前でトリガーを定義することができます。unit_kindを指定することをお薦めします。

Live SQL:

この例は、Oracle Live SQL の「任意の種類のユニット名へのアクセスの制限」で表示および実行できます

```
PROCEDURE protected_proc2
```

ACCESSIBLE BY (top_trusted_f) AS BEGIN DBMS_OUTPUT.PUT_LINE('Executed protected_proc2.'); END; FUNCTION top_protected_f RETURN NUMBER ACCESSIBLE BY (TRIGGER top_trusted_f) AS BEGIN RETURN 0.5; END top_protected_f; FUNCTION top_trusted_f RETURN NUMBER AUTHID DEFINER IS FUNCTION g RETURN NUMBER DETERMINISTIC IS BEGIN RETURN

0.5; END g; BEGIN protected_proc2; RETURN g() - DBMS_RANDOM.VALUE(); END top_trusted_f; SELECT top_trusted_f FROM DUAL; .381773176 1 row selected. Executed protected_proc2.

例13-3 ストアド・プロシージャへのアクセスの制限

この例では、top_trusted_procプロシージャからのコールのみが可能なパッケージ・プロシージャを示します。サブプログラムの仕様部と本体のACCESSIBLE BY句が、一致している必要があります。accessorリストにこのプロシージャが含まれていないACCESSIBLE BY句のある既存のプロシージャに対してコールが行われた場合、エラーが発生します。

Live SQL:

この例は、Oracle Live SQL の「ストアド・プロシージャへのアクセスの制限」で表示および実行できます

```
CREATE OR REPLACE PACKAGE protected_pkg
```

AS PROCEDURE public_proc; PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc); END; CREATE OR REPLACE PACKAGE BODY protected_pkg AS PROCEDURE public_proc AS BEGIN DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.public_proc'); END; PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc) AS BEGIN DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.private_proc'); END; END; CREATE OR REPLACE PROCEDURE top_trusted_proc AS BEGIN DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls protected_pkg.private_proc '); protected_pkg.private_proc; END; Procedure created. EXEC top_trusted_proc; top_trusted_proc calls protected_pkg.private_proc Executed protected_pkg.private_proc EXEC protected_pkg.private_proc PLS-00904: insufficient privilege to access object PRIVATE_PROC

関連トピック

この章:

- ファンクションの宣言および定義
- プロシージャの宣言および定義

他の章:

- ネストしたサブプログラム、パッケージ・サブプログラムおよびスタンドアロン・サブプログラム
- サブプログラムのプロパティ
- パッケージ作成のガイドライン

13.2 AGGREGATE句

ファンクションを集計ファンクションまたは行のグループを評価して単一行を戻すファンクションとして指定します。

集計ファンクションは、選択リスト、HAVING句およびORDER BY句で指定できます。

問合せでユーザー定義の集計ファンクションを指定すると、分析ファンクション(問合せ結果セットで機能するファンクション)として処理できます。これを行うには、SQL分析ファンクションで使用可能なOVER analytic_clause構文を使用します。

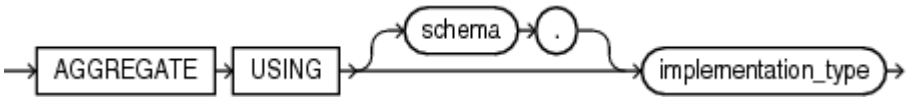
AGGREGATE句は、CREATE FUNCTION文に指定できます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

aggregate_clause ::=



セマンティクス

aggregate_clause

AGGREGATE USING

ファンクションの実装タイプの名前を指定します。

[schema.] implementation_type

この実装タイプは、ODCIAggregateサブプログラムの実装が含まれているADTにする必要があります。schemaを指定しない場合、実装タイプは自分のスキーマ内に存在するとみなされます。

AGGREGATE USINGの制限

ネストしたファンクションにaggregate_clauseを指定することはできません。

この句を指定する場合、ファンクションの入力引数は1つしか指定できません。

例

- 例12-33、集計ファンクションとしてのパイプライン・テーブル・ファンクション

関連トピック

この章:

- ファンクションの宣言および定義

その他のドキュメント:

- 分析ファンクションの構文およびセマンティクスは、『Oracle Database SQL言語リファレンス』を参照してください。
- ユーザー定義の集計関数の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。
- ODCIサブプログラムの詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

13.3 代入文

代入文は、データ項目の値を有効な値に設定します。

ここでのトピック

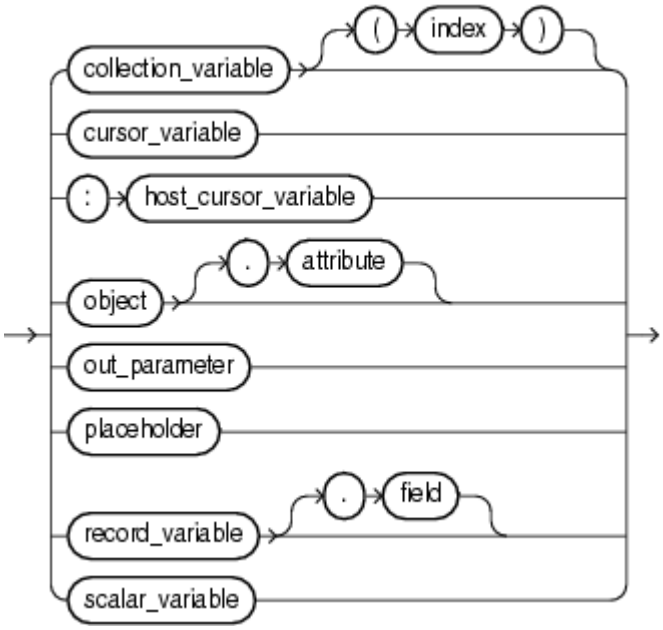
- 構文
- セマンティクス
- 例
- 関連トピック

構文

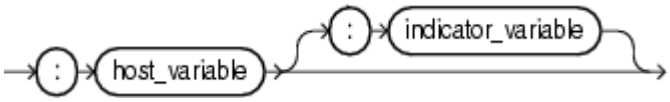
assignment_statement ::=

(expression ::=)

assignment_statement_target ::=



placeholder ::=



セマンティクス

assignment_statement

expression

assignment_statement_targetに代入される値を持つ式。

expressionとassignment_statement_targetのデータ型には互換性が必要です。

ノート:

同じ型の要素を持つコレクションは、同じデータ型ではない場合もあります。コレクション型の定義の構文は、「コレクション変数の宣言」を参照してください。

assignment_statement_target

expressionの値が代入されるデータ項目。

collection_variable

コレクション変数の名前。

index

collection_variableの要素の索引。indexを省略すると、コレクション変数全体が代入文のターゲットになります。

indexは、データ型PLS_INTEGERまたはPLS_INTEGERに暗黙的に変換可能なデータ型を持つ数式である必要があります(後者の詳細は「PLS_INTEGERの事前定義のサブタイプ」を参照してください)。

cursor_variable

カーソル変数の名前。

:host_cursor_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されるカーソル変数の名前。コロン(:)とhost_cursor_variableの間に空白は入れないでください。

ホスト・カーソル変数のデータ型は、PL/SQLカーソル変数の戻り型と互換性があります。

object

抽象データ型(ADT)のインスタンスの名前。

attribute

objectの属性の名前。attributeを省略すると、ADT全体が代入文のターゲットになります。

out_parameter

代入文が使用されているサブプログラムの仮パラメータOUTまたはIN OUTの名前。

record_variable

レコード変数の名前。

field

record_variableのフィールドの名前。fieldを省略すると、レコード変数全体が代入文のターゲットになります。

scalar_variable

PL/SQLスカラー変数の名前。

placeholder

:host_variable

PL/SQLホスト環境で宣言され、バインド引数としてPL/SQLに渡される変数の名前。コロン(:)とhost_variableの間に空白は入れないでください。

:indicator_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡される標識変数の名前。(標識変数は、関連付けられたホスト変数の値または条件を示します。たとえば、Oracleプリコンパイラ環境では、標識変数を使用して出力ホスト変数内のNULLや切り捨てられた値を検出できます。)host_variableとコロン(:)の間またはコロンとindicator_variableの間に空白は入れないでください。正しい場合を次に示します。

```
:host_variable:indicator_variable
```

例

- 例2-24、代入文を使用した変数への値の代入
- 例2-27、BOOLEAN変数への値の代入
- 例5-10、コレクション代入のデータ型の互換性

関連トピック

この章:

- 式
- FETCH文
- SELECT INTO文

他の章:

- 「変数への値の代入」
- 「コレクション変数への値の代入」
- 「レコード変数への値の代入」

13.4 AUTONOMOUS_TRANSACTIONプラグマ

AUTONOMOUS_TRANSACTIONプラグマは、ルーチンを自律型(メイン・トランザクションから独立している)とマークします。

このコンテキストでは、ルーチンは次のいずれかになります。

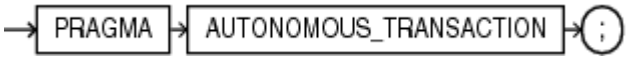
- スキーマレベル(ネストしていない)の無名PL/SQLブロック
- スタンドアロン・サブプログラム、パッケージ・サブプログラムまたはネストしたサブプログラム
- ADTのメソッド
- 非複合トリガー

ここでのトピック

- 構文
- 例
- 関連トピック

構文

autonomous_trans_pragma ::=



例

- 例6-43、パッケージでの自律型ファンクションの宣言
- 例6-44、自律型スタンドアロン・プロシージャの宣言
- 例6-45、自律型PL/SQLブロックの宣言
- 例6-46、自律型トリガーによるINSERT文のログへの記録
- 例6-47、自律型トリガーによるDDLに対するシステム固有の動的SQLの使用
- 例6-48、自律型ファンクションの起動

関連トピック

- プラグマ
- 自律型トランザクション

13.5 基本LOOP文

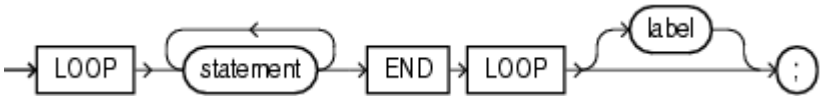
基本LOOP文が反復されるたびに、文が実行され、制御がループの先頭に戻ります。ループの内側の文によって制御がループの外側に移された場合や例外が呼び出された場合に、LOOP文は終了します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

basic_loop_statement ::=



(statement ::=)

セマンティクス

basic_loop_statement

statement

無限ループが発生しないように、1つ以上の文によって制御をループの外側に移す必要があります。制御をループの外側に移すことができる文には次のものがあります。

- 「CONTINUE文」(外側のラベル付きループの次の反復に制御を移す場合)
- EXIT文
- GOTO文
- RAISE文

label

basic_loop_statementを識別するラベル(「statement ::=」および「label」を参照)。CONTINUE、EXITおよびGOTO文でこのラベルを参照できます。

LOOP文がネストされている場合は特に、ラベルによって読みやすさが向上します。ただし、END LOOP文内のラベルが、同じLOOP文の先頭のラベルと一致している場合のみです(コンパイラではチェックしません)。

例

- 例1-2、問合せ結果行の一度に1行ずつの処理
- 例4-9、EXIT文が含まれている基本LOOP文
- 例4-10、EXIT WHEN文が含まれている基本LOOP文

- 例4-11、EXIT WHEN文が含まれているネストされたラベル付き基本LOOP文
- 例4-13、基本LOOP文内のCONTINUE文
- 例4-14、基本LOOP文内のCONTINUE WHEN文

関連トピック

この章:

- 「カーソルFOR LOOP文」
- FOR LOOP文
- WHILE LOOP文

他の章:

- 基本LOOP文

13.6 ブロック

関連する宣言および文をグループ化するブロックは、PL/SQLソース・プログラムの基本単位です。

オプションの宣言部、必須の実行部、オプションの例外処理部が含まれます。宣言はブロックに対してローカルで、そのブロックの実行が完了するとなくなります。ブロックはネストできます。

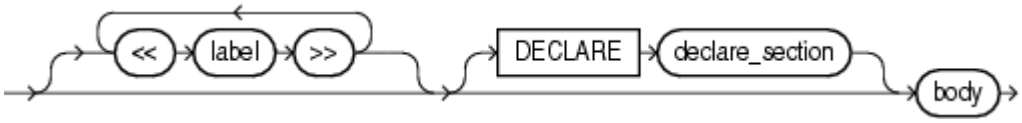
無名ブロックは実行可能文です。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

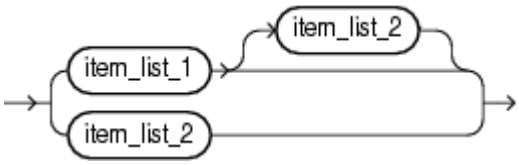
構文

plsql_block ::=



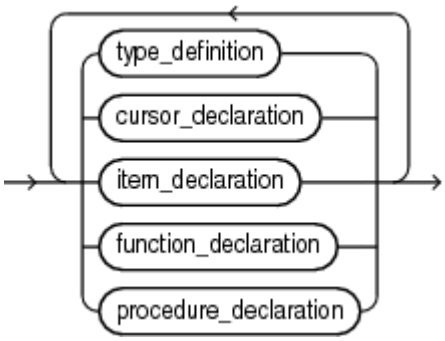
(body ::=)

declare_section ::=



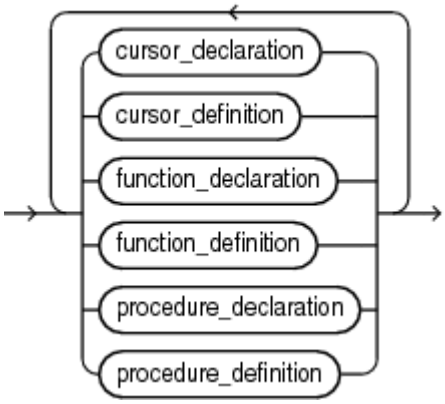
(item_list_2 ::=)

item_list_1 ::=



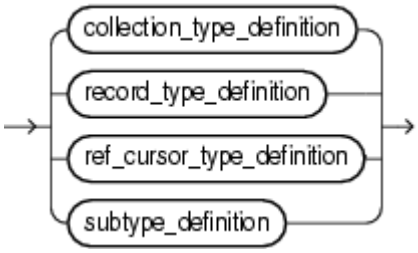
(cursor_declaration ::=、function_declaration ::=、item_declaration ::=、procedure_declaration ::=、type_definition ::=)

item_list_2 ::=



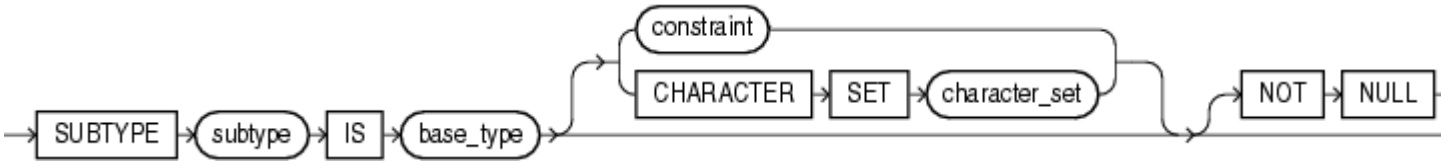
(cursor_declaration ::=、cursor_definition ::=、function_declaration ::=、function_definition ::=、procedure_declaration ::=、procedure_definition ::=)

type_definition ::=

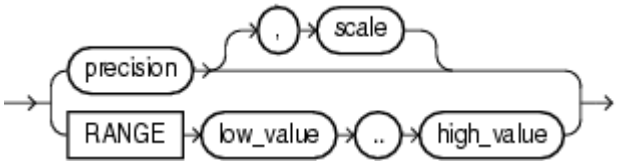


(collection_type_definition ::=、record_type_definition ::=、ref_cursor_type_definition ::=、subtype_definition ::=)

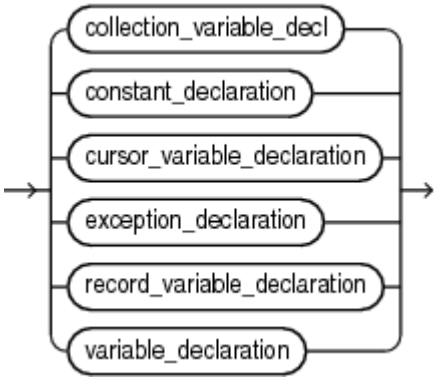
subtype_definition ::=



constraint ::=



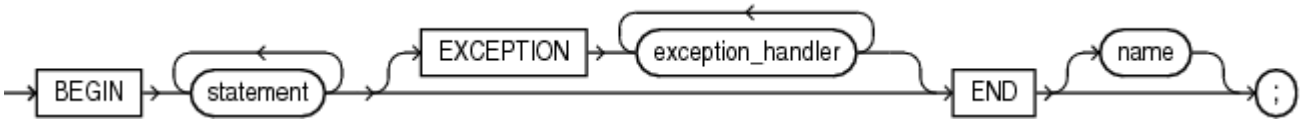
item_declaration ::=



(collection_variable_decl ::=, constant_declaration ::=, cursor_declaration ::=, cursor_variable_declaration ::=, exception_declaration ::=, record_variable_declaration ::=,

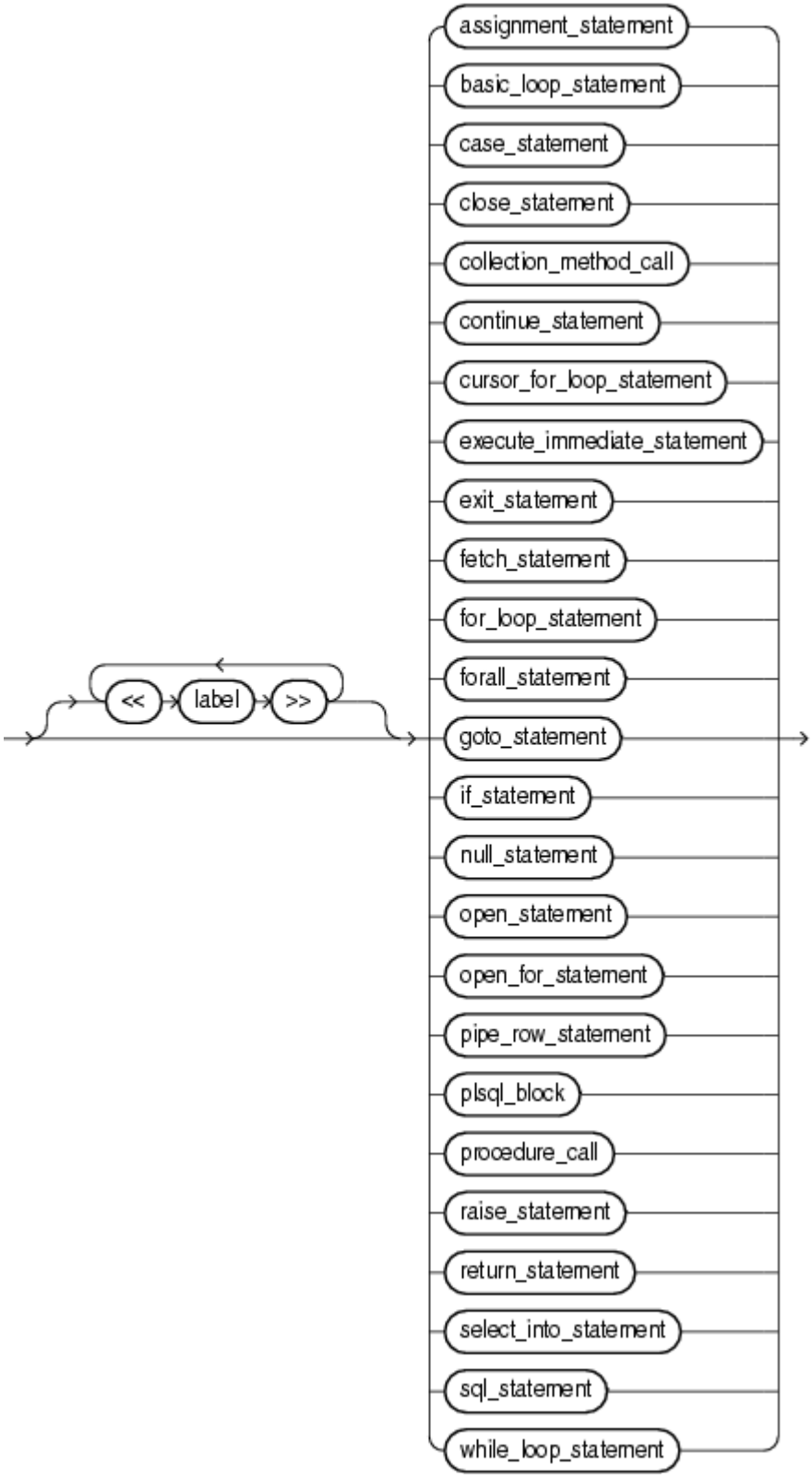
variable_declaration ::=)

body ::=



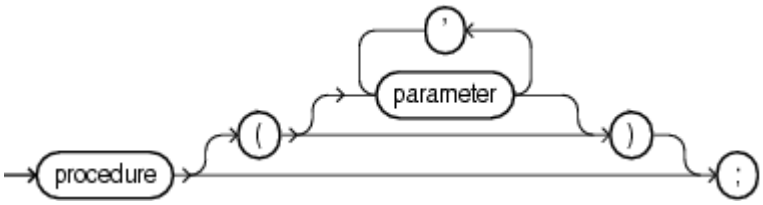
(exception_handler ::=)

statement ::=

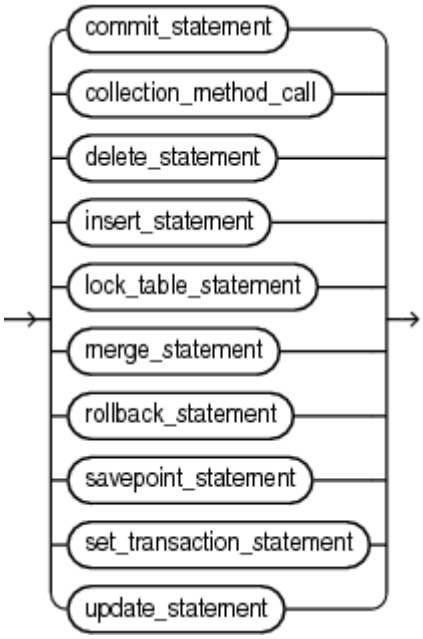


(plsql_block ::=、procedure_call ::=、sql_statement ::=)

procedure_call ::=



sql_statement ::=



セマンティクス

plsql_block

label

ブロックで一意である未宣言の識別子。

DECLARE

ブロックの宣言部を開始します。

declare_section

ローカル宣言を置きます。ローカル宣言はこのブロックとこのブロックのサブブロックにのみ存在し、外側のブロックからは見えません。

declare_sectionの制限

- create_package、create_package_bodyまたはcompound_trigger_blockのdeclare_sectionには、PRAGMA AUTONOMOUS_TRANSACTIONを含めることはできません。
- trigger_bodyまたはtps_bodyのdeclare_sectionでは、データ型LONGまたはLONG RAWの変数を宣言できません。

関連項目:

- create_packageの詳細は、「CREATE PACKAGE文」を参照してください
- create_package_bodyの詳細は、「CREATE PACKAGE BODY文」を参照してください
- compound_trigger_block、trigger_body、およびtps_bodyの詳細は、「CREATE TRIGGER文」を参照してください

subtype_definition

サブタイプの宣言では静的な式を使用できます。詳細は、静的な式を参照してください。

サブタイプ

定義するユーザー定義のサブタイプの名前。

base_type

定義しているサブタイプのベース型。base_typeは、CHAR、DATEまたはRECORDなどのスカラーまたはユーザー定義のPL/SQLデータ型指定子です。

CHARACTER SET character_set

文字データ型のサブタイプの文字セットを指定します。

CHARACTER SET character_setの制限

base_typeが文字データ型でない場合は、この句は指定しないでください。

NOT NULL

このサブタイプを使用して宣言されるデータ項目にNOT NULL制約を付けます。この制約の詳細は、「NOT NULL制約」を参照してください。

constraint

数値データ型のサブタイプの制約を指定します。

constraintの制限

base_typeが数値データ型でない場合は、constraintは指定しないでください。

precision

数値データ型の制約付きサブタイプの精度を指定します。

precisionの制限

base_typeに精度を指定できない場合は、precisionは指定しないでください。

scale

数値データ型の制約付きサブタイプの位取りを指定します。

scaleの制限

base_typeに位取りを指定できない場合は、scaleは指定しないでください。

RANGE low_value .. high_value

数値データ型の制約付きサブタイプの範囲を指定します。low_valueとhigh_valueは数値リテラルである必要があります。

RANGE high_value .. low_valueの制限

この句は、base_typeまたはPLS_INTEGERのサブタイプ(事前定義またはユーザー定義のいずれか)がPLS_INTEGERである場合にのみ指定します。(表3-3に、PLS_INTEGERの事前定義のサブタイプの概要を示しています。範囲を指定できるユーザー定義のサブタイプの詳細は、「制約付きサブタイプ」を参照してください。)

body

BEGIN

ブロックの実行部を開始します。実行部には実行可能文が置かれます。

EXCEPTION

ブロックの例外処理部を開始します。PL/SQLによって例外が呼び出されると、ブロックの通常の実行が停止され、制御が適切なexception_handlerに移ります。例外ハンドラが終了すると、ブロック直後の文から実行が再開されます。例外処理の詳細は、「PL/SQLのエラー処理」を参照してください。

exception_handler

例外ハンドラを参照してください。

END

ブロックを終了します。

name

ENDが適用されるブロックの名前(ラベル、ファンクション名、プロシージャ名またはパッケージ名)。

statement

label

文で一意である未宣言の識別子。

assignment_statement

代入文を参照してください。

basic_loop_statement

「基本LOOP文」を参照してください。

case_statement

CASE文を参照してください。

close_statement

CLOSE文を参照してください。

collection_method_call

次のいずれかのコレクション・メソッドの起動。これらコレクション・メソッドはプロシージャです。

- DELETE
- EXTEND
- TRIM

構文は、「コレクション・メソッドの起動」を参照してください。

continue_statement

CONTINUE文を参照してください。

cursor_for_loop_statement

「カーソルFOR LOOP文」を参照してください。

execute_immediate_statement

「EXECUTE IMMEDIATE文」を参照してください。

exit_statement

EXIT文を参照してください。

fetch_statement

FETCH文を参照してください。

for_loop_statement

「FOR LOOP文」を参照してください。

forall_statement

FORALL文を参照してください。

goto_statement

GOTO文を参照してください。

if_statement

IF文を参照してください。

null_statement

NULL文を参照してください。

open_statement

OPEN文を参照してください。

open_for_statement

「OPEN FOR文」を参照してください。

pipe_row_statement

「PIPE ROW文」を参照してください。

pipe_row_statementの制限

この文を使用できるのは、パイプライン・テーブル・ファンクションの本体内のみです。他の場所で使用すると、例外が発生します。

raise_statement

RAISE文を参照してください。

return_statement

RETURN文を参照してください。

select_into_statement

「SELECT INTO文」を参照してください。

while_loop_statement

「WHILE LOOP文」を参照してください。

procedure_call

procedure

起動するプロシージャの名前。

parameter [, parameter]...

起動するプロシージャの実パラメータのリスト。各実パラメータのデータ型は、対応する仮パラメータのデータ型と互換性がある必要があります。使用できる実パラメータは仮パラメータのモードで決まります。

仮パラメータのモード	実パラメータ
IN	定数、初期化された変数、リテラルまたは式
OUT	データ型が NOT NULL として定義されていない変数
IN OUT	変数(通常は、文字列バッファまたは数値アキュムレータ)

プロシージャによってパラメータのデフォルト値が指定される場合は、パラメータ・リストでそのパラメータを省略できます。プロシージャにパラメータがない場合、またはプロシージャによってすべてのパラメータにデフォルト値が指定される場合は、パラメータ・リスト

を省略するか、または空のパラメータ・リストを指定することができます。

関連項目:

実パラメータの位置表記法、名前表記法および混合表記法

sql_statement

commit_statement

SQL COMMIT文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

delete_statement

SQL DELETE文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。「DELETE文の拡張機能」も参照してください。

insert_statement

SQL INSERT文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。「INSERT文の拡張機能」も参照してください。

lock_table_statement

SQLのLOCK TABLE文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

merge_statement

SQL MERGE文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

rollback_statement

SQL ROLLBACK文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

savepoint_statement

SQL SAVEPOINT文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

set_transaction_statement

SQLのSET TRANSACTION文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

update_statement

SQL UPDATE文。構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。「UPDATE文の拡張機能」も参照してください。

例

- 例1-1、PL/SQLブロック構造
- 例2-23、複数の重複したラベルを使用するブロック

- 例4-30、不適切なラベル配置

関連トピック

この章:

- コメント

他の章:

- ブロック
- 識別子
- プラグマ
- PL/SQLデータ型
- 「ユーザー定義のPL/SQLサブタイプ」

13.7 コール仕様

コール仕様は、PL/SQLから起動できるようにJavaメソッドまたはC言語サブプログラムを宣言します。このようなメソッドまたはサブプログラムは、SQLのCALL文を使用して起動することもできます。コール仕様は、起動時に起動するJavaメソッドまたは共有ライブラリ内の名前付きサブプログラムをデータベースに指示します。また、引数および戻り値に対して実行する型変換もデータベースに指示します。

コール仕様は、次のSQL文に指定できます。

- ALTER TYPE文
- CREATE FUNCTION文
- CREATE PROCEDURE文
- CREATE TYPE文
- CREATE TYPE BODY文

ここでのトピック

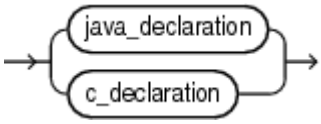
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

コール仕様を起動するには、追加の権限が必要になる場合があります。たとえば、Cコール仕様の場合には、Cライブラリに対するEXECUTE権限が必要です。

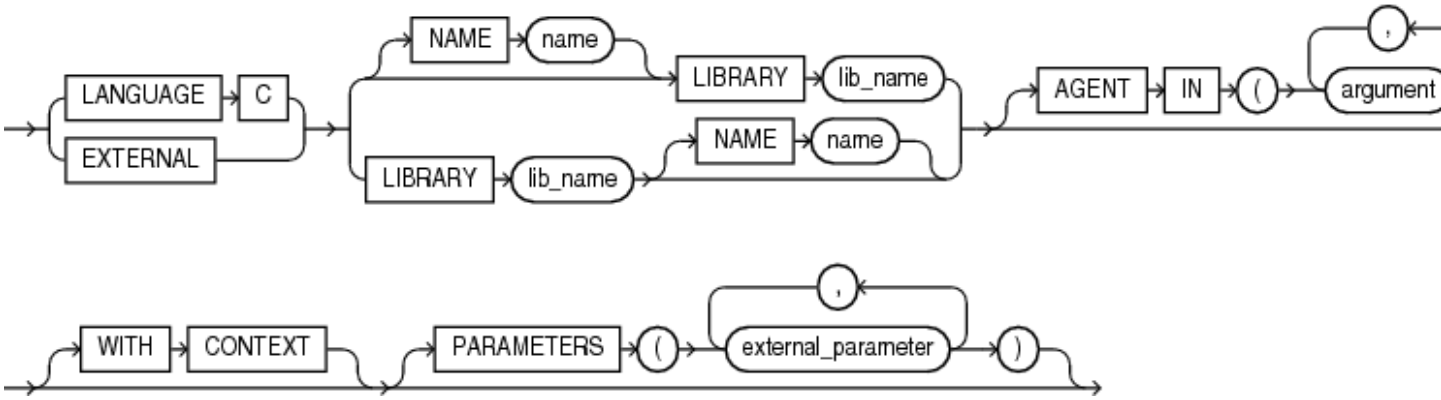
構文

call_spec ::=

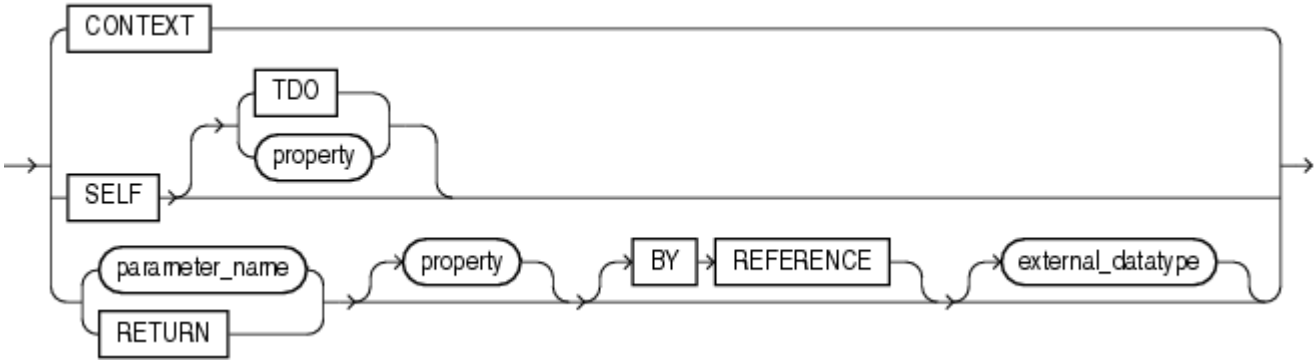


java_declaration ::=

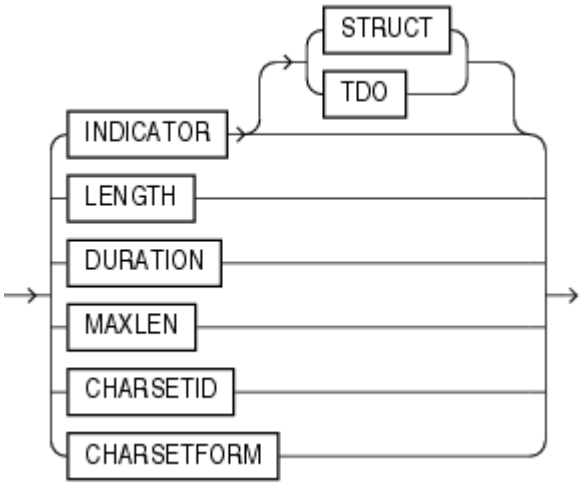
c_declaration ::=



external_parameter ::=



property ::=



セマンティクス

call_spec

Cプロシージャ名またはJavaメソッド名、パラメータ型および戻り型を、対応するSQLにマップします。

コール仕様は、PL/SQLスタンドアロン・サブプログラム、パッケージ仕様部とパッケージ本体、および型仕様部と型本体に記述できます。PL/SQLブロックの内部には記述できません。

java_declaration

string

メソッドのJava実装を識別します。

c_declaration

LIBRARY lib_name

CREATE LIBRARY文によって作成されたライブラリを識別します。

EXTERNAL

下位互換性のためにのみサポートされている、Cサブプログラムの非推奨の宣言方法。Cコール仕様にデフォルト設定された引数または制約付きPL/SQL型が含まれている場合は、その仕様でEXTERNALを使用します。それ以外の場合は、LANGUAGE C構文を使用します。

例

例13-4 外部ファンクションの例

次に示す仮定の文は、Cサブプログラムc_get_valを外部ファンクションとして登録するPL/SQLスタンドアロン・ファンクションget_valを作成します。(この例ではパラメータを省略しています。)

```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
    y_val IN NUMBER,
    image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

関連トピック

この章:

- ファンクションの宣言および定義
- プロシージャの宣言および定義

他の章:

- CREATE LIBRARY文
- 外部サブプログラム

その他のドキュメント:

- CALL文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- SQL文からコールされるユーザー定義ファンクションの制限の詳細は、『Oracle Database開発ガイド』を参照してください。
- Javaコール仕様を作成する方法を学習するには、『Oracle Database Java開発者ガイド』を参照してください。
- Cコール仕様を作成する方法を学習するには、『Oracle Database開発ガイド』を参照してください。

13.8 CASE文

CASE文を使用すると、一連の条件を基に、対応する文を選択して実行できます。

単純なCASE文は、単一の式を評価して、可能性のある複数の値と比較します。

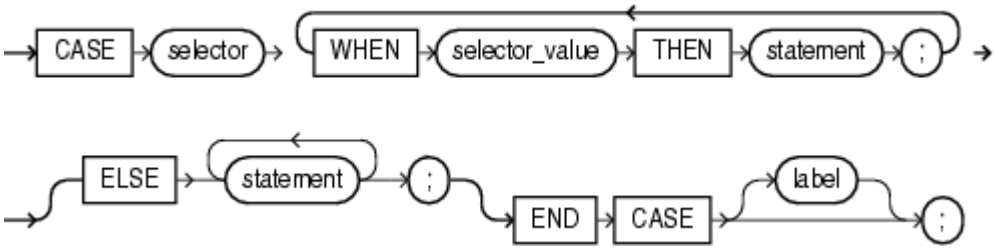
検索CASE文は、複数のブール式を評価して、値がTRUEである最初の式を選択します。

ここでのトピック

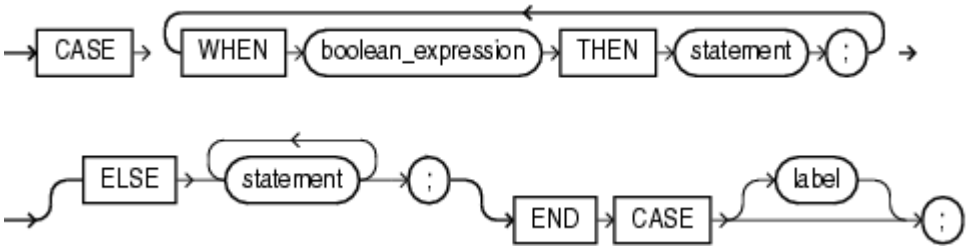
- 構文
- セマンティクス
- 例
- 関連トピック

構文

simple_case_statement ::=



searched_case_statement ::=



(boolean_expression ::=、statement ::=)

セマンティクス

simple_case_statement

selector

値が1回評価され、複数の選択肢から1つ選択するためにその値が使用される式。selectorの値は、BLOB、BFILEまたはユーザー定義型を除く任意のPL/SQL型になります。

WHEN selector_value THEN statement

selector_valueは、BLOB、BFILEまたはユーザー定義型を除く任意のPL/SQL型の式になります。

selector_valueは順番に評価されます。selector_valueの値がselectorの値と等しい場合、そのselector_valueに関

連付けられているstatementが実行され、CASE文は終了します。後続のselector_valueは評価されません。

> 注意:
>
> statement によってデータベースが変更され、非決定的なファンクションが起動される場合があります。C 言語の switch 文のような fall-through のメカニズムはありません。

ELSE statement [statement]...

statementは、selectorと同じ値を持つselector_valueが存在しない場合にのみ実行されます。

selectorと同じ値を持つselector_valueが存在しない場合にELSE句を指定しないと、システムによって事前定義の例外CASE_NOT_FOUNDが呼び出されます。

label

文を識別するラベル(「statement ::=」および「label」を参照)。

searched_case_statement

WHEN boolean_expression THEN statement

boolean_expressionは順番に評価されます。boolean_expressionの値がTRUEの場合、そのboolean_expressionに関連付けられているstatementが実行され、CASE文は終了します。後続のboolean_expressionは評価されません。

> 注意:
>
> statement によってデータベースが変更され、非決定的なファンクションが起動される場合があります。C 言語の switch 文のような fall-through のメカニズムはありません。

ELSE statement [statement]...

statementは、TRUEの値を持つboolean_expressionが存在しない場合にのみ実行されます。

TRUEの値を持つboolean_expressionが存在しない場合にELSE句を指定しないと、システムによって事前定義の例外CASE_NOT_FOUNDが呼び出されます。

label

文を識別するラベル(「statement ::=」および「label」を参照)。

例

- 例3-2、BOOLEAN値の出力
- 例4-6、単純なCASE文
- 例4-7、検索CASE文

関連トピック

この章:

- IF文

他の章:

- CASE式
- 条件付き選択文
- 単純なCASE文
- 検索CASE文

関連項目:

- NULLIFファンクションについては、『Oracle Database SQL言語リファレンス』を参照してください。
- COALESCEファンクションについては、『Oracle Database SQL言語リファレンス』を参照してください。

13.9 CLOSE文

CLOSE文は、名前付きカーソルをクローズし、そのリソースを再利用できるように解放します。

明示カーソルは、クローズした後、OPEN文を使用して再オープンできます。明示カーソルは、再オープンする前にクローズする必要があります。

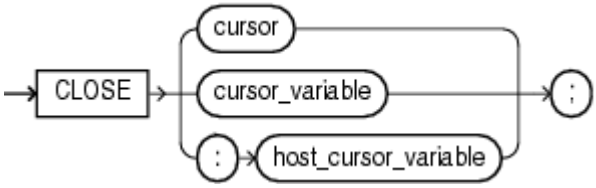
カーソル変数は、クローズした後、OPEN FOR文を使用して再オープンできます。カーソル変数を再オープンする場合、その前にクローズする必要はありません。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

close_statement ::=



セマンティクス

close_statement

cursor

オープンされている明示カーソルの名前。

cursor_variable

オープンされているカーソル変数の名前。

:host_cursor_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されるカーソル変数の名前。コロン(:)とhost_cursor_variableの間に空白は入れないでください。

例

- 例6-6、LOOP文内部のFETCH文

関連トピック

この章:

- 「FETCH文」
- OPEN文
- OPEN FOR文

他の章:

- 「明示カーソルのオープンおよびクローズ」
- 「カーソル変数のオープンおよびクローズ」

13.10 コレクション・メソッドの起動

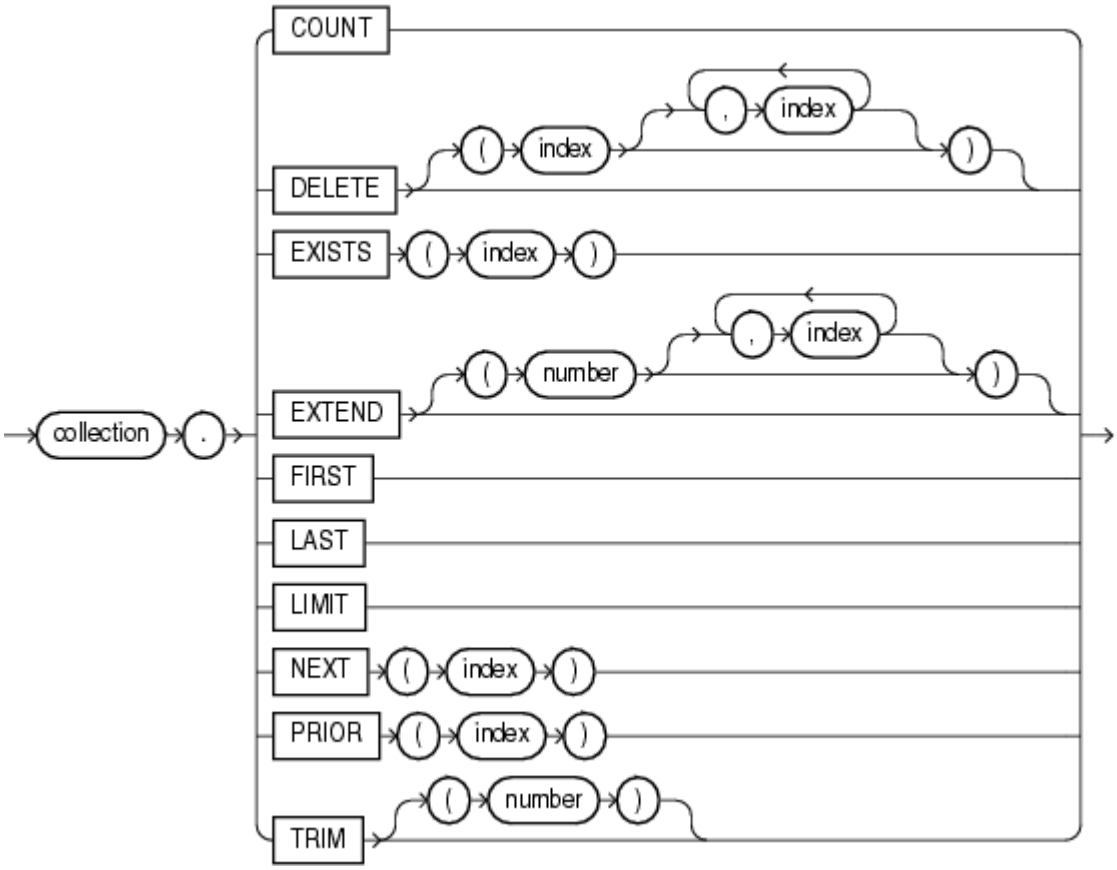
コレクション・メソッドは、コレクションに関する情報を戻したり、コレクションに対する操作を実行するPL/SQLサブプログラムです。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

collection_method_call ::=



セマンティクス

collection_method_call

collection

起動するメソッドを持つコレクションの名前。

COUNT

コレクション内の要素の数を戻すファンクション。詳細は、「COUNTコレクション・メソッド」を参照してください。

DELETE

コレクションから要素を削除するプロシージャ。詳細は、「DELETEコレクション・メソッド」を参照してください。

DELETEの制限

collectionがVARRAYの場合、DELETEを使用して索引を指定することはできません。

index

データ型PLS_INTEGERまたはPLS_INTEGERに暗黙的に変換可能なデータ型を持つ数式である必要があります(後者の詳細は「s」を参照してください)。

EXISTS

コレクションにindex番目の要素が存在する場合はTRUE、それ以外の場合はFALSEを戻すファンクション。詳細は、「EXISTSコレクション・メソッド」を参照してください。

EXTEND

コレクションの末尾に要素を追加するプロシージャ。詳細は、「EXTENDコレクション・メソッド」を参照してください。

EXTENDの制限

collectionが連想配列の場合は、EXTENDは使用できません。

FIRST

コレクション内の最初の索引を戻すファンクション(詳細は、「FIRSTおよびLASTコレクション・メソッド」を参照)。

LAST

コレクション内の最後の索引を戻すファンクション(詳細は、「FIRSTおよびLASTコレクション・メソッド」を参照)。

LIMIT

コレクションに格納できる要素の最大数を戻すファンクション。コレクションに最大サイズがない場合、LIMITはNULLを戻します。例は、「LIMITコレクション・メソッド」を参照してください。

NEXT

後続の要素が存在する場合、そのコレクション要素の索引を戻すファンクション。存在しない場合、NEXTはNULLを戻します。詳細は、「PRIORおよびNEXTコレクション・メソッド」を参照してください。

PRIOR

先行する要素が存在する場合、そのコレクション要素の索引を戻すファンクション。存在しない場合、NEXTはNULLを戻します。詳細は、「PRIORおよびNEXTコレクション・メソッド」を参照してください。

TRIM

コレクションの末尾から要素を削除するプロシージャ。詳細は、「TRIMコレクション・メソッド」を参照してください。

TRIMの制限

collectionが連想配列の場合は、TRIMは使用できません。

number

コレクションの末尾から削除する要素の数。デフォルトは1です。

例

- 例5-19、ネストした表でのDELETEメソッド
- 例5-20、文字列で索引付けされている結合配列でのDELETEメソッド
- 例5-21、ネストした表でのTRIMメソッド
- 例5-22、ネストした表でのEXTENDメソッド
- 例5-23、ネストした表でのEXISTSメソッド
- 例5-24、PLS_INTEGERで索引付けされている連想配列のFIRSTとLASTの値
- 例5-25、文字列で索引付けされている結合配列のFIRSTとLASTの値
- 例5-26、FOR LOOP内のFIRSTおよびLASTによるVARRAYの出力
- 例5-27、FOR LOOP内のFIRSTおよびLASTによるネストした表の出力
- 例5-28、VARRAYのCOUNTとLASTの値
- 例5-29、ネストした表のCOUNTとLASTの値
- 例5-30、様々なコレクション型のLIMITとCOUNTの値
- 例5-31、PRIORおよびNEXTメソッド
- 例5-32、疎であるネストした表の要素の出力

関連トピック

この章:

- コレクション変数の宣言

他の章:

- コレクション・メソッド

13.11 コレクション変数の宣言

コレクション変数はコンポジット変数であり、要素と呼ばれるその内部コンポーネントはデータ型が同じです。

コレクション変数の値およびその要素の値は変更可能です。

コレクション全体は、その名前によって参照します。コレクションの要素を参照するには、構文collection(index)を使用します。

PL/SQLには次の3種類のコレクション型があります。

- 連想配列(以前のPL/SQL表または索引付き表)
- 可変サイズの配列(VARRAY)
- ネストした表

連想配列は、文字列型またはPLS_INTEGERで索引付けできます。VARRAYおよびネストした表は、整数で索引付けできます。

コレクション変数は、次のいずれかの方法で作成できます。

- コレクション型を定義し、次にその型の変数を宣言します。
- 事前に宣言されているコレクション変数と同じ型のコレクション変数を宣言するには、%TYPEを使用します。

ノート:

このトピックの内容は、PL/SQL ブロックまたはパッケージ内で定義するコレクション型に適用されます。このコレクション型は、CREATE TYPE 文を使用して作成するスタンドアロン・コレクション型とは異なります。

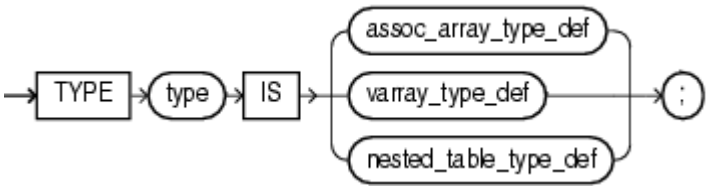
PL/SQL ブロックまたはパッケージでは、3 つすべてのコレクション型を定義できます。CREATE TYPE 文を使用した場合、ネストした表型および VARRAY 型は作成できますが、連想配列型は作成できません。

ここでのトピック

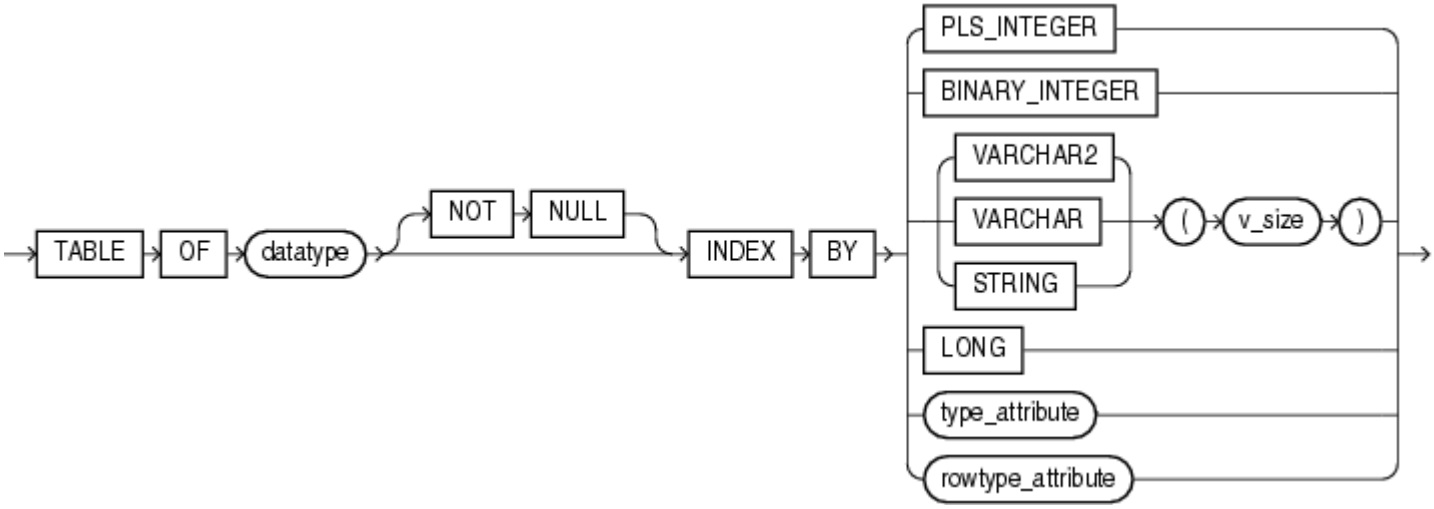
- 構文
- セマンティクス
- 例
- 関連トピック

構文

collection_type_definition ::=

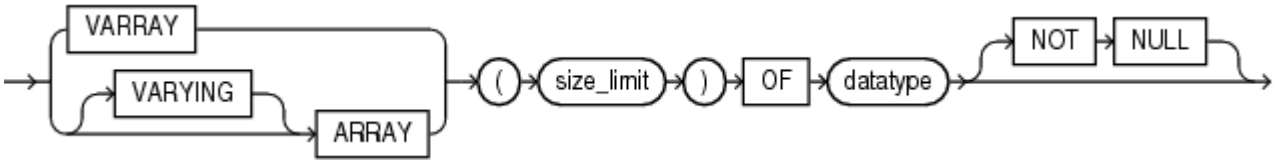


assoc_array_type_def ::=



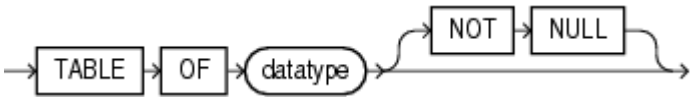
(datatype ::=, rowtype_attribute ::=, type_attribute ::=)

varray_type_def ::=



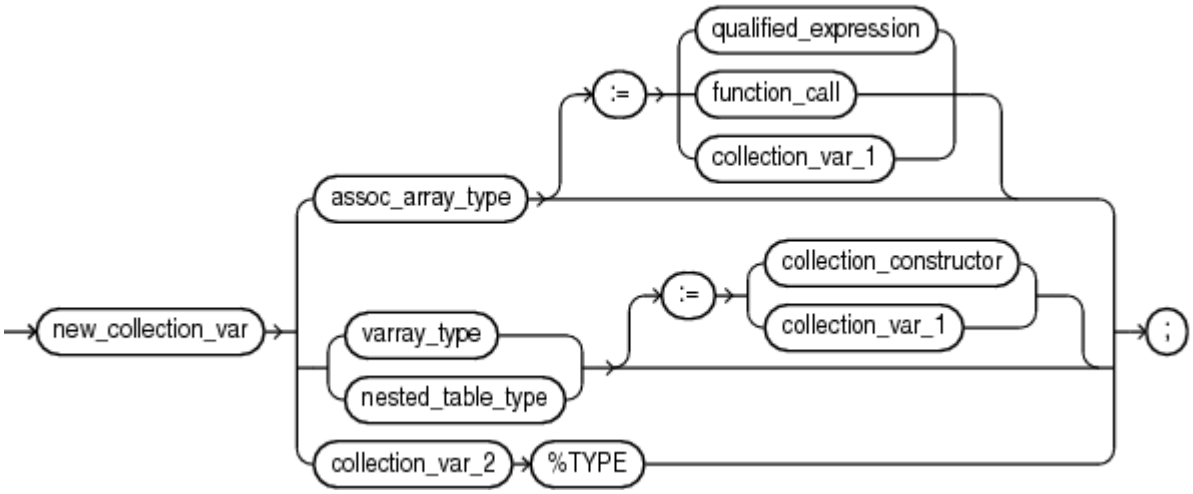
(datatype ::=)

nested_table_type_def ::=



(datatype ::=)

collection_variable_decl ::=



(collection_constructor ::=, function_call ::=, qualified_expression ::=)

セマンティクス

collection_type_definition

type

定義するコレクション型の名前。

assoc_array_type_def

連想配列の型定義。

assoc_array_type_defの制限

ブロック、サブプログラム、パッケージ仕様部、またはパッケージ本体の宣言部にのみ記述できます。

datatype

連想配列の要素のデータ型。datatypeには、REF CURSORを除く任意のPL/SQLデータ型を指定できます。

NOT NULL

連想配列のすべての要素にNOT NULL制約を指定します。この制約の詳細は、「NOT NULL制約」を参照してください。

{ PLS_INTEGER | BINARY_INTEGER }

連想配列の索引のデータ型がPLS_INTEGERであることを指定します。

{ VARCHAR2 | VARCHAR | STRING } (v_size)

連想配列の索引のデータ型が、v_sizeの長さのVARCHAR2(またはそのサブタイプのVARCHARまたはSTRING)であることを指定します。

連想配列の要素には、TO_CHARファンクション(『Oracle Database SQL言語リファレンス』を参照)でVARCHAR2に変換できる任意のデータ型の値を移入できます。

注意:

文字列で索引付けされている連想配列は、各国語サポート(NLS)パラメータの影響を受ける可能性があります。詳細は、「文字列で索引付けされている連想配列に影響を与える NLS パラメータ値」を参照してください。

LONG

連想配列の索引のデータ型がLONG(VARCHAR2(32760)と等価)であることを指定します。

ノート:

LONG は、既存のアプリケーションとの下位互換性のためにのみサポートされています。新しいアプリケーションでは VARCHAR2(32760)を使用してください。

type_attribute、rowtype_attribute

連想配列の索引のデータ型が、%ROWTYPEまたは%TYPEのいずれかで指定されたデータ型であることを指定します。このデータ型はPLS_INTEGER、BINARY_INTEGERまたはVARCHAR2(v_size)を表している必要があります。

varray_type_def

可変サイズの配列の型定義。

size_limit

VARRAYに格納できる要素の最大数。size_limitは、1から2147483647の範囲の整数リテラルである必要があります。

datatype

VARRAYの要素のデータ型。datatypeには、REF CURSORを除く任意のPL/SQLデータ型を指定できます。

NOT NULL

VARRAYのすべての要素にNOT NULL制約を指定します。この制約の詳細は、「NOT NULL制約」を参照してください。

nested_table_type_def

ネストした表の型定義。

datatype

ネストした表の要素のデータ型。datatypeには、REF CURSORまたはNCLOBを除く任意のPL/SQLデータ型を指定できます。

datatypeがスカラー型である場合、ネストした表にはCOLUMN_VALUEと呼ばれるスカラー型の列が1つ存在します。

datatypeがADTである場合、ネストした表の列はADTの名前および属性と一致します。

NOT NULL

ネストした表のすべての要素にNOT NULL制約を指定します。この制約の詳細は、「NOT NULL制約」を参照してください。

collection_variable_decl

new_collection_var

宣言するコレクション変数の名前。

assoc_array_type

事前に定義されている連想配列型の名前。new_collection_varのデータ型です。

varray_type

事前に定義されているVARRAY型の名前。new_collection_varのデータ型です。

nested_table_type

事前に定義されているネストした表型の名前。new_collection_varのデータ型です。

collection_constructor

new_collection_varのデータ型のコレクション・コンストラクタ。new_collection_varの初期値を指定します。

collection_var_1

new_collection_varと同じデータ型の事前に宣言されているコレクション変数の名前。new_collection_varの初期値を指定します。

> ノート:
>
> collection_var_1 と new_collection_var は、同じ型の要素を持っているだけでなく、同じデータ型を持っている必要があります。

collection_var_2

事前に宣言されているコレクション変数の名前。

%TYPE

%TYPE属性を参照してください。

例

- 例5-1、文字列で索引付けされている連想配列
- 例5-2、PLS_INTEGERで索引付けされている連想配列を戻すファンクション
- 例5-4、VARRAY (可変サイズの配列)
- 例5-5、ローカル型のネストした表
- 例5-13、2次元のVARRAY (VARRAYのVARRAY)
- 例5-14、ネストした表のネストした表と整数のVARRAYのネストした表

関連トピック

この章:

- コレクション・メソッドの起動
- FORALL文
- レコード変数の宣言
- %ROWTYPE属性
- %TYPE属性

他の章:

- コレクションに関するトピック
- BULK COLLECT句
- 「CREATE TYPE文」

13.12 コメント

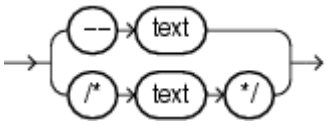
コメントは、PL/SQLコンパイラに無視されるソース・プログラム・テキストです。主な目的はコードに説明を付けることですが、廃止されたコード部分または作成途中のコード部分を無効にするために使用することもできます(つまり、コードをコメントにすることができます)。PL/SQLには単一行コメントと複数行コメントの両方があります。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

comment ::=



セマンティクス

comment

--

行の残りの部分が単一行コメントになります。次の行にラップされたテキストは、コメントになりません。

> 注意:
>
> Oracle プリコンパイラ・プログラムが動的に処理する PL/SQL ブロックの中に単一行コメントを置かないでください。Oracle プリコンパイラ・プログラムでは行の終わりを示す文字が無視されるため、単一行コメントはブロックの末尾で終了します。

/*

コメントを開始します。このコメントは複数行にまたがることができます。

*/

コメントを終了します。

text

任意のテキスト。

textの制限

複数行のコメントでは、textに複数行コメントのデリミタ/*または*/を含めることはできません。したがって、複数行コメントに別の複数行コメントを含めることはできません。ただし、複数行コメントに単一行コメントを含めることはできます。

例

- 例2-6、単一行コメント
- 例2-7、複数行コメント

関連トピック

- コメント

13.13 COMPILE句

COMPILE句は、無効になったストアド・ユニットを明示的に再コンパイルすることで、実行時の暗黙的な再コンパイルの必要をなくして、実行時コンパイル・エラーとパフォーマンスのオーバーヘッドを回避するために使用します。

COMPILE句は、次のSQL文に指定できます。

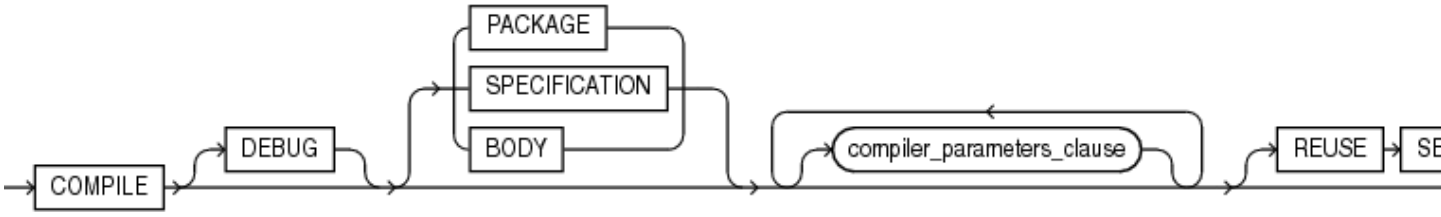
- ALTER FUNCTION文
- ALTER PACKAGE文
- ALTER PROCEDURE文
- ALTER LIBRARY文
- ALTER TYPE文
- ALTER TRIGGER文

ここでのトピック

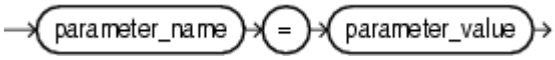
- 構文
- セマンティクス
- 関連トピック

構文

compile_clause ::=



compiler_parameters_clause ::=



セマンティクス

compile_clause

COMPILE

有効か無効かに関係なく、PL/SQLユニットが再コンパイルされます。PL/SQLユニットは、ライブラリ、パッケージ、パッケージ仕様部、パッケージ本体、トリガー、プロシージャ、ファンクション、型、型仕様部または型本体のいずれかになります。

まず、ユニットが依存するいずれかのオブジェクトが無効の場合は、そのオブジェクトがデータベースによって再コンパイルされます。

また、そのユニットに依存するすべてのローカル・オブジェクトがデータベースによって無効にされます。

ユニットは、データベースによって正常に再コンパイルされた場合に有効になります。それ以外の場合は、データベースによってエラーが戻され、ユニットは無効のままになります。関連付けられているコンパイラ・エラー・メッセージは、SQL*PlusコマンドSHOW

ERRORSを使用して確認できます。

再コンパイル中、データベースによってコンパイラの永続的なスイッチ設定はすべて削除され、セッションから再度取得されて、コンパイル後に格納されます。この処理を回避するには、REUSE SETTINGS句を指定します。

DEBUG

PLSQL_OPTIMIZE_LEVEL=1と同じ効果があり、PL/SQLデバッガで使用するためのコードを生成して格納するようにPL/SQLコンパイラに指示します。DEBUGではなく、PLSQL_OPTIMIZE_LEVEL=1を使用することをお薦めします。

PACKAGE

(デフォルト)有効か無効かに関係なくパッケージ仕様部とパッケージ本体(存在する場合)の両方が再コンパイルされます。パッケージ仕様部および本体を再コンパイルした場合、SPECIFICATIONおよびBODYで説明されているように、依存オブジェクトの無効化および再コンパイルが行われます。

PACKAGEの制限事項

PACKAGEは、パッケージをコンパイルする場合にのみ使用できます。

SPECIFICATION

有効か無効かに関係なく、型仕様部のみが再コンパイルされます。仕様部を変更した後で、パッケージ仕様部または型仕様部を再コンパイルすることで、コンパイル・エラーを確認できます。

仕様部を再コンパイルすると、その仕様部に依存するローカル・オブジェクト(パッケージ内のプロシージャまたはファンクションを起動するプロシージャなど)がデータベースによって無効にされます。パッケージの本体も仕様部に依存します。その後、最初に明示的に再コンパイルせずにこれらの依存オブジェクトのいずれかを参照すると、データベースによって実行時にそのオブジェクトが暗黙的に再コンパイルされます。

SPECIFICATIONの制限事項

SPECIFICATIONは、パッケージ仕様部または型仕様部をコンパイルする場合にのみ使用できます。

BODY

有効か無効かに関係なく、パッケージ本体または型本体のみが再コンパイルされます。パッケージ本体または型本体を変更後に再コンパイルできます。本体を再コンパイルしても、その本体の仕様部に依存するオブジェクトが無効になることはありません。

パッケージ本体または型本体の再コンパイル時に、その本体が依存するオブジェクトのいずれかが無効な場合は、データベースによってそれらのオブジェクトが最初に再コンパイルされます。本体は、データベースによって正常に再コンパイルされた場合に有効になります。

BODYの制限事項

BODYは、パッケージ本体または型本体をコンパイルする場合にのみ使用できます。

REUSE SETTINGS

Oracle Databaseによってコンパイラ・スイッチ設定が削除および再取得されないようにします。この句を指定すると、Oracleによって既存の設定が保持され、この文の他の場所に値が指定されていないパラメータの再コンパイルにその設定が使用されま

す。

DEFAULT COLLATION句のコンパイル・セマンティクスも参照してください。

compiler_parameters_clause

表1-2に示されているPL/SQLコンパイル・パラメータの値を指定します。これらの各パラメータのコンパイル時の値は、コンパイルされるPL/SQLユニットのメタデータとともに格納されます。

各パラメータは、それぞれの文で1回のみ指定できます。各設定は、コンパイルされているPL/SQLユニットに対してのみ有効であり、このセッションまたはシステムの他のコンパイルには影響しません。セッションまたはシステム全体に影響を与えるには、ALTER SESSIONまたはALTER SYSTEM文を使用してパラメータの値を設定する必要があります。

この句からいずれかのパラメータを省略してREUSE SETTINGSを指定した場合に、このPL/SQLユニットの以前のコンパイルでそのパラメータの値が指定されていると、データベースによって以前の値が使用されます。いずれかのパラメータを省略した場合に、REUSE SETTINGSを指定しないか、または以前のコンパイルでそのパラメータの値が指定されていないと、データベースによってセッション環境からそのパラメータの値が取得されます。

関連トピック

その他のドキュメント:

- プロシージャのデバッグの詳細は、『Oracle Database開発ガイド』を参照してください。
- ストアド・サブプログラムで使用可能な機能と同じ機能を使用したトリガーのデバッグの詳細は、『Oracle Database開発ガイド』を参照してください。

13.14 定数の宣言

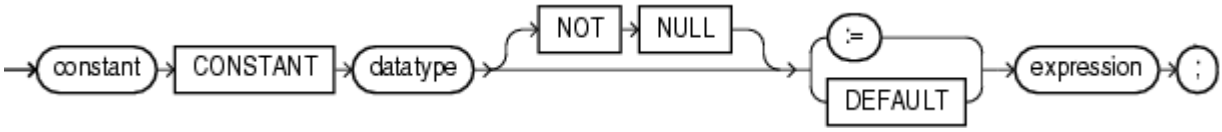
定数は、変更されない値を保持します。定数の宣言では、定数の名前、データ型および値を指定し、使用する記憶域を割り当てます。宣言でNOT NULL制約を課すこともできます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

constant_declaration ::=



(datatype ::=, expression ::=)

セマンティクス

constant_declaration

constant

宣言する定数の名前。

datatype

初期値を使用して変数を宣言できるデータ型。

NOT NULL

定数にNOT NULL制約を指定します。

この制約の詳細は、「NOT NULL制約」を参照してください。

expression

定数の初期値。expressionのデータ型はdatatypeと互換性がある必要があります。constant_declarationが処理されるとき、expressionの値がconstantに代入されます。

例

- 例2-12、定数の宣言
- 例2-13、初期値が指定された変数および定数の宣言

関連トピック

この章:

- コレクション変数の宣言
- レコード変数の宣言
- %ROWTYPE属性
- スカラー変数の宣言
- %TYPE属性

他の章:

- 定数の宣言
- 「連想配列の定数の宣言」
- レコード定数の宣言

13.15 CONTINUE文

CONTINUE文は、現行のループの反復を条件付きまたは無条件で終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの次の反復に制御を移します。

CONTINUE文でカーソルFORループを途中で終了すると(たとえば、内部ループを終了して外部ループの次の反復に制御を移すと)、カーソルはクローズします(このコンテキストでは、CONTINUEはGOTOと同様に動作します)。

> ノート:
>
> Oracle Database 11g リリース 1 の時点では、CONTINUE は PL/SQL のキーワードです。使用しているプログラムで CONTINUE というサブプログラムが起動されると、警告が表示されます。

CONTINUE文の制限

- CONTINUE文はLOOP文内にある必要があります。
- CONTINUE文はサブプログラムまたはメソッドの境界を越えることはできません。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

continue_statement ::=

(boolean_expression ::=)

セマンティクス

continue_statement

label

カレント・ループまたは外側のループのいずれかを識別する名前(「基本LOOP文」を参照)。

labelを指定しない場合、CONTINUE文はカレント・ループの次の反復に制御を移します。labelを指定すると、CONTINUE文は、labelで識別されるループの次の反復に制御を移します。

WHEN boolean_expression

この句を指定しない場合、CONTINUE文は現行のループの反復を無条件で終了します。この句を指定すると、

boolean_expressionの値がTRUEの場合にのみCONTINUE文は現行のループの反復を終了します。

例

- 例4-13、基本LOOP文内のCONTINUE文
- 例4-14、基本LOOP文内のCONTINUE WHEN文
- 例4-27、内側のFOR LOOP文内のCONTINUE WHEN文

関連トピック

この章:

- 「基本LOOP文」
- 「カーソルFOR LOOP文」
- 「EXIT文」
- 「式」
- 「FOR LOOP文」
- 「WHILE LOOP文」

他の章:

- LOOP文
- CONTINUE文
- CONTINUE WHEN文

13.16 COVERAGEプラグマ

COVERAGEプラグマは、テストできないPL/SQLコードをカバレッジ用にマークします。これらのマークは、カバレッジ・メトリックの精度を高めます。

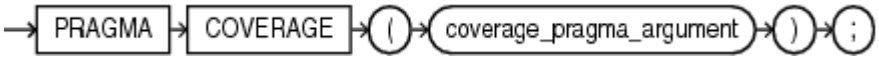
COVERAGEプラグマは、PL/SQLソース・コードをマークして、適切にテストできないコードをカバレッジ用に示します。プラグマは特定のコード・セクションをマークします。実行不可能なコードをマークすることで、テストが実施された程度を評価するために使用されるカバレッジ・メトリックの質を高めます。

ここでのトピック

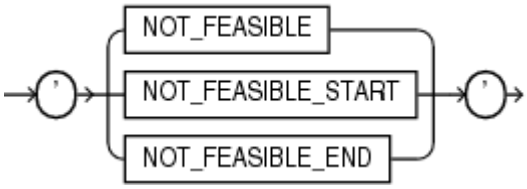
- 構文
- セマンティクス
- 例
- 関連トピック

構文

coverage_pragma ::=



coverage_pragma_argument ::=



セマンティクス

coverage_pragma

COVERAGEプラグマは、任意の宣言または文の前に記述できます。

coverage_pragma_argument

COVERAGEプラグマの引数には、次のいずれかの値が必要です。

- 'NOT_FEASIBLE'
- 'NOT_FEASIBLE_START'
- 'NOT_FEASIBLE_END'

COVERAGEが引数'NOT_FEASIBLE'とともに記述されている場合、プラグマに続く最初の宣言または文の先頭を含む基本ブロック全体をマークします。

引数'NOT_FEASIBLE_START'を指定したCOVERAGEプラグマを任意の宣言または文の前に記述できます。この後に、引数'NOT_FEASIBLE_END'を指定したCOVERAGEプラグマを続ける必要があります。2番目のプラグマは、任意の宣言または文の前に記述できます。これは最初のプラグマと同じPL/SQLブロックに記述する必要があり、ネストしたサブプログラム定義に記

述することはできません。

関連付けられたCOVERAGEプラグマのペアは、最初のプラグマに続く最初の宣言または文の先頭を含む基本ブロックの先頭から、2番目のプラグマに続く最初の宣言または文を含む基本ブロックの最後まで、実行不可能な基本ブロックをマークします。

内側のサブプログラムの定義または宣言を範囲に含むCOVERAGEプラグマは、そのサブプログラムのブロックを実行不可能としてマークしません。

例

例13-5 単一の基本ブロックにカバレッジのテスト実行不可能のマークを付ける

この例は、zおよびzl基本ブロックに対する代入に先行するプラグマCOVERAGEの配置を示しています。これら2つの基本ブロックはカバレッジ計算で無視されます。最初のCOVERAGEプラグマ(1としてマーク)はzに対する最初の代入を実行不可能としてマークし、2番目(2としてマーク)はzに対する3番目の代入をマークしています。いずれの場合も、影響を受ける基本ブロックは、識別子zから次のEND IFまでです。

```
IF (x>0) THEN
  y :=2;
ELSE
  PRAGMA COVERAGE (' NOT_FEASIBLE');  -- 1
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  PRAGMA COVERAGE (' NOT_FEASIBLE'); -- 2
  z :=3;
END IF;
```

例13-6 行範囲にカバレッジのテスト実行不可能のマークを付ける

この例では、行範囲全体を実行不可能としてマークしています。1つの行範囲に複数の基本ブロックを含めることができます。行範囲が、範囲の先頭で「NOT_FEASIBLE_START」引数を指定したプラグマCOVERAGEを使用し、範囲の末尾で「NOT_FEASIBLE_END」引数を指定したプラグマCOVERAGEを使用することにより、カバレッジの実行が可能ではないものとしてマークされています。ペアのCOVERAGEプラグマは、すべてのブロックを実行不可能としてマークしています。

```
PRAGMA COVERAGE (' NOT_FEASIBLE_START');
IF (x>0) THEN
  y :=2;
ELSE
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  z :=3;
END IF;
PRAGMA COVERAGE (' NOT_FEASIBLE_END');
```

例13-7 ユニット全体または個別のサブプログラムにカバレッジのテスト実行不可能のマークを付ける

この例では、プロシージャfoo全体を実行不可能としてカバレッジ用にマークしています。サブプログラムの本体をすべて実行不可能としてマークすることで、このサブプログラムを完全に実行不可能としてマークしていることになります。

```
CREATE PROCEDURE foo IS
PRAGMA COVERAGE ('NOT_FEASIBLE_START');
......
BEGIN
....
PRAGMA COVERAGE ('NOT_FEASIBLE_END');
END;
/
```

例13-8 内部プログラムにカバレッジのテスト実行不可能のマークを付ける

この例は、外側のCOVERAGEプラグマのペアがプロシージャinnerの内側のカバレッジに影響を与えないことを示しています。innerの本体の内側にあるCOVERAGEプラグマ(1としてマーク)は、xに対する2番目の代入を実行不可能としてマークしています。引数'NOT_FEASIBLE_END'を指定したプラグマが最後の行でなくても、プロシージャouterの本体全体が実行不可能としてマークされていることに注意してください。このプラグマは、プラグマに続く文を含む基本ブロックをマークし、そのブロックはプロシージャの最後まで続いています。

```
CREATE OR REPLACE PROCEDURE outer IS
   PRAGMA COVERAGE ('NOT_FEASIBLE_START');
   x NUMBER := 7;
   PROCEDURE inner IS
   BEGIN
     IF x < 6 THEN
        x := 19;
     ELSE
        PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 1
        x := 203;
     END IF;
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE ('X= ');
   PRAGMA COVERAGE ('NOT_FEASIBLE_END');
   DBMS_OUTPUT.PUT_LINE (x);
END;
/
```

関連トピック

このドキュメント:

- プラグマ
- PLSQL_OPTIMIZE_LEVELコンパイル・パラメータの詳細は、PL/SQLユニットおよびコンパイル・パラメータを参照してください

その他のドキュメント:

- PL/SQL基本ブロック・カバレッジを使用した品質維持の詳細は、Oracle Database開発ガイドを参照してください
- DBMS_PLSQL_CODE_COVERAGEパッケージの使用に関する詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください

13.17 カーソルFOR LOOP文

カーソルFOR LOOP文は、指定されたカーソルが戻す行の型のレコード変数として暗黙的にループ索引を宣言し、カーソルをオープンします。

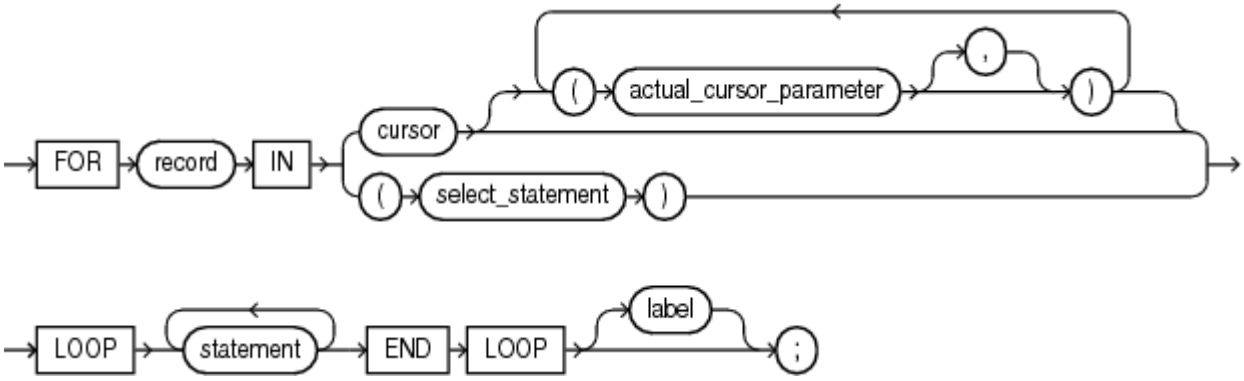
反復されるたびに、カーソルFOR LOOP文は、結果セットから行をフェッチしてレコードに入れます。フェッチする行がなくなると、カーソルFOR LOOP文によってカーソルがクローズされます。また、ループの内側の文によって制御がループの外側に移された場合や、例外が呼び出された場合も、カーソルがクローズされます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

cursor_for_loop_statement ::=



(statement ::=)

セマンティクス

cursor_for_loop_statement

record

cursorまたはselect_statementが戻す型の%ROWTYPEレコード変数としてカーソルFOR LOOP文が暗黙的に宣言するループ索引の名前。

recordは、カーソルFOR LOOP文に対してローカルです。ループの内側の文は、recordとそのフィールドを参照できます。仮想列は別名でのみ参照できます。ループの外側の文は、recordを参照できません。カーソルFOR LOOP文の実行後、recordは未定義となります。

cursor

カーソルFOR LOOPに入るときには、オープンされていない明示カーソル(カーソル変数ではなく)の名前。

actual_cursor_parameter

cursorの仮パラメータに対応する実パラメータ。

select_statement

SQLのSELECT文(PL/SQLのSELECT INTO文ではない)。select_statementに対して、PL/SQLは暗黙カーソルを宣言し、オープンし、データをフェッチしてクローズします。ただし、select_statementは独立した文ではないため、暗黙カーソルは内部的であり、SQLという名前で参照することはできません。

関連項目:

SELECT文の構文は、『Oracle Database SQL言語リファレンス』を参照してください。

label

cursor_for_loop_statementを識別するラベル(「statement ::=」および「label」を参照)。CONTINUE、EXITおよびGOTO文でこのラベルを参照できます。

LOOP文がネストされている場合は特に、ラベルによって読みやすさが向上します。ただし、END LOOP文内のラベルが、同じLOOP文の先頭のラベルと一致している場合のみです(コンパイラではチェックしません)。

例

- 例6-18、暗黙カーソルFOR LOOP文
- 例6-19、明示カーソルFOR LOOP文
- 例6-20、明示カーソルFOR LOOP文へのパラメータの引渡し
- 例6-21、カーソルFOR LOOPによる仮想列の参照

関連トピック

この章:

- 「基本LOOP文」
- CONTINUE文
- 「EXIT文」
- 「明示カーソルの宣言および定義」
- 「FETCH文」
- 「FOR LOOP文」
- FORALL文
- OPEN文
- 「WHILE LOOP文」

他の章:

- 「 カーソルFOR LOOP文による問合せ結果セットの処理」

13.18 カーソル変数の宣言

カーソル変数は、1つの問合せに限定されない明示カーソルと似ています。

カーソル変数を作成するには、事前定義のSYS_REFCURSOR型の変数を宣言するか、REF CURSOR型を定義してからその型の変数を宣言します。

カーソル変数の制限

- カーソルFOR LOOP文の中ではカーソル変数を使用できません。
- カーソル変数は、パッケージ仕様部では宣言できません。

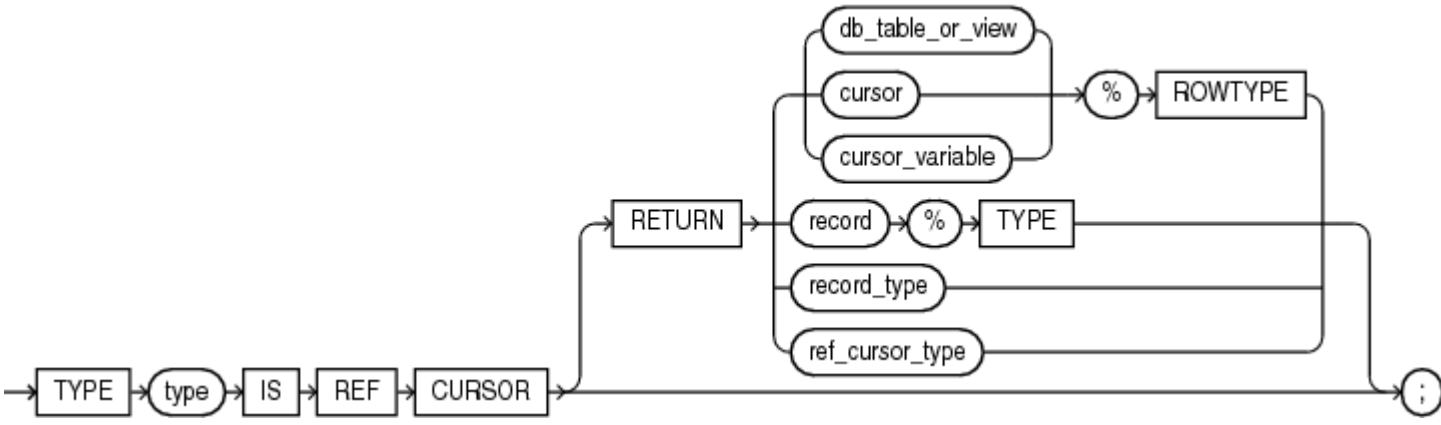
  つまり、パッケージにはパブリック・カーソル変数(パッケージの外から参照できるカーソル変数)を含めることはできません。
- カーソル変数の値をコレクションまたはデータベース列に格納することはできません。
- 比較演算子を使用して、カーソル変数が等しいかどうか、またはNULLかどうかをテストできません。
- サーバー間リモート・プロシージャ・コール(RPC)でカーソル変数を使用するとエラーが発生します。ただし、リモート・データベースがプロシージャ・ゲートウェイを介してアクセスされるOracle以外のデータベースである場合は、サーバー間RPCでカーソル変数を使用できます。

ここでのトピック

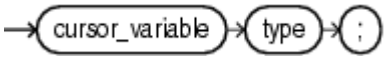
- 構文
- セマンティクス
- 例
- 関連トピック

構文

ref_cursor_type_definition ::=



cursor_variable_declaration ::=



セマンティクス

ref_cursor_type_definition

type

定義するREF CURSOR型の名前。

RETURN

カーソル変数が戻す値のデータ型を指定します。

強いREF CURSOR型を定義する場合はRETURNを指定します。弱いREF CURSOR型を定義する場合はRETURNを省略します。REF CURSORの強い型と弱い型の詳細は、「カーソル変数の作成」を参照してください。

db_table_or_view

宣言が処理されるときにアクセスできる必要があるデータベースの表またはビューの名前。

cursor

事前に宣言されている明示カーソルの名前。

cursor_variable

事前に宣言されているカーソル変数の名前。

record

ユーザー定義のレコードの名前。

record_type

データ型指定子RECORDを使用して定義されたユーザー定義型の名前。

ref_cursor_type

データ型指定子REF CURSORを使用して定義されたユーザー定義型の名前。

cursor_variable_declaration

cursor_variable

宣言するカーソル変数の名前。

type

宣言するカーソル変数の型(SYS_REFCURSOR、または事前に定義したREF CURSOR型の名前)。

SYS_REFCURSORは弱い型です。REF CURSORの強い型と弱い型の詳細は、「カーソル変数の作成」を参照してください。

例

- 例6-24、カーソル変数の宣言
- 例6-25、ユーザー定義の戻り型を持つカーソル変数
- 例6-28、カーソル変数問合せの変数(結果セットの変更なし)

- 例6-29、カーソル変数問合せの変数(結果セットの変更あり)
- 例6-30、静的SQLを使用したコレクションの問合せ
- 例6-31、1つの問合せ用にカーソル変数をオープンするプロシージャ
- 例6-32、選択された問合せ用のカーソル変数のオープン(同じ戻り型)
- 例6-33、選択された問合せ用のカーソル変数のオープン(異なる戻り型)
- 例6-34、Pro*Cクライアント・プログラムにおけるホスト変数としてのカーソル変数

関連トピック

この章:

- CLOSE文
- 名前付きカーソル属性
- 「明示カーソルの宣言および定義」
- 「FETCH文」
- OPEN FOR文
- %ROWTYPE属性
- %TYPE属性

他の章:

- カーソル変数
- 「パイプライン・テーブル・ファンクションへのCURSOR式の引渡し」

13.19 データ型属性

ADT要素のデータ型属性。

datatypeでは、レコード変数のフィールド、定数、ファンクションの戻り値、コレクション変数およびコレクション型の要素のデータ型を宣言できます。

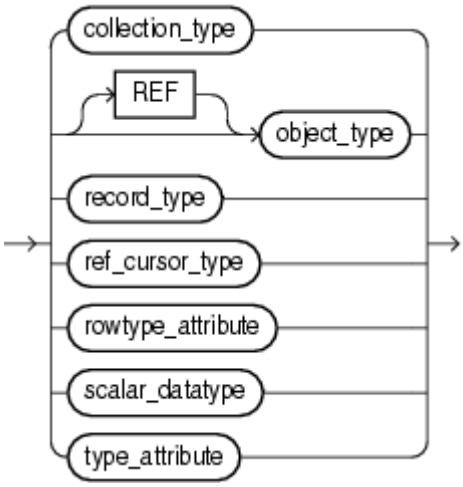
- レコード変数の宣言
- 定数の宣言
- ファンクションの宣言および定義
- コレクション変数の宣言
- CREATE FUNCTION文
- CREATE TYPE文

ここでのトピック

- 構文
- セマンティクス
- 関連トピック

構文

datatype ::=



(rowtype_attribute ::=, type_attribute ::=)

セマンティクス

datatype

collection_type

ユーザー定義のVARRAY型またはネストした表型の名前(連想配列型の名前ではありません)。

object_type

ユーザー定義型のインスタンス。

record_type

データ型指定子RECORDを使用して定義されたユーザー定義型の名前。

ref_cursor_type

データ型指定子REF CURSORを使用して定義されたユーザー定義型の名前。

scalar_datatype

サイズ、精度および文字またはバイトのセマンティクスの修飾子が含まれているスカラー・データ型の名前。

関連トピック

他の章:

- PL/SQLのデータ型

13.20 DEFAULT COLLATION句

照合(ソート順序付けとも呼ばれる)は、2つの文字列が比較およびソートされるときに、ある文字列が別の文字列と等しいか、別の文字列の前に来るか後に来るかを決定します。Oracle Databaseの照合は、異なる言語で使用される、テキストのソートのルールに従って、文字列を並べ替えます。

DEFAULT COLLATION句は、次のSQL文に指定できます。

- CREATE FUNCTION文
- CREATE PROCEDURE文
- CREATE PACKAGE文
- CREATE TRIGGER文
- CREATE TYPE文

ここでのトピック

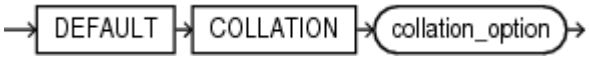
- 前提条件
- 構文
- セマンティクス
- コンパイル・セマンティクス
- 関連トピック

前提条件

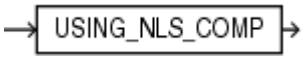
COMPATIBLE初期化パラメータは少なくとも12.2.0に設定する必要があり、これらのSQL文で照合の宣言を許可するためにMAX_STRING_SIZEはEXTENDEDに設定する必要があります。

構文

default_collation_clause ::=



collation_option ::=



セマンティクス

default_collation_clause

default_collation_clauseは、パッケージ仕様部、スタンドアロン型仕様部およびスタンドアロン・サブプログラムで記述できます。

collation_option

プロシージャ、ファンクション、パッケージ、型またはトリガーのデフォルトの照合はUSING_NLS_COMPであることが必要です。default_collation_clauseは、PL/SQLユニットのデフォルトの照合をUSING_NLS_COMPとして明示的に宣言します。この句

がない場合、ユニットのデフォルトの照合は有効なスキーマのデフォルトの照合から継承されます。有効なスキーマのデフォルトの照合がUSING_NLS_COMPでない場合、ユニットは無効です。

有効なスキーマのデフォルトの照合は次のようにして決定されます。

- セッション・パラメータDEFAULT_COLLATIONが設定されている場合、有効なスキーマのデフォルトの照合はこのパラメータの値です。パラメータの値は、SYS_CONTEXT('USERENV', 'SESSION_DEFAULT_COLLATION')を問い合せることでチェックできます。DEFAULT_COLLATIONが設定されていない場合、このファンクションはNULLを戻します。パラメータDEFAULT_COLLATIONの値は、次の文を使用して設定できます: ALTER SESSION SET DEFAULT_COLLATION = collation_option;
- セッション・パラメータDEFAULT_COLLATIONが設定されていない場合、有効なスキーマのデフォルトの照合は、PL/SQLユニットの作成先となるスキーマの宣言済のデフォルトの照合です。スキーマのデフォルトの照合は、静的データ・ディクショナリ*_USERSビューにあります。これは、DDL文CREATE USERおよびALTER USERを使用して設定できます。

セッション・パラメータDEFAULT_COLLATIONは、次の文を使用して未設定にできます: ALTER SESSION SET DEFAULT_COLLATION = NONE;

パッケージ本体および型本体は、対応する仕様部のデフォルトの照合を使用します。プロシージャ、ファンクションおよびメソッド(パラメータおよび戻り値を含む)の文字データ・コンテナおよび属性はすべて、そのデータバインドされた照合が疑似照合USING_NLS_COMPであるかのように動作します。

DEFAULT COLLATIONの制限

ネストしたサブプログラムもしくはパッケージ・サブプログラムまたは型のメソッドに対して指定することはできません。

コンパイル・セマンティクス

結果的にデフォルトのオブジェクト照合がUSING_NLS_COMPと異なる場合、データベース・オブジェクトはコンパイル・エラーのため無効として作成されます。

REUSE SETTINGS句を指定したALTER COMPILE文がPL/SQLユニットに対して発行されると、コンパイルするデータベース・オブジェクトについて、格納されたデフォルトの照合は変更されません。

REUSE SETTINGS句なしでALTER COMPILE文が発行されると、コンパイルするデータベース・オブジェクトについて、格納されたデフォルトの照合は破棄され、PL/SQLユニットにDEFAULT COLLATION句が含まれていないかぎり、文の実行時のオブジェクト所有者の有効なスキーマのデフォルトの照合がそのオブジェクトのデフォルトの照合として格納されます。結果的にデフォルトの照合がUSING_NLS_COMPでない場合、コンパイル・エラーが発生します。

パッケージ本体または型本体のALTER COMPILE文は、対応する仕様部の格納された照合を参照します。

関連トピック

他の章:

- ALTER FUNCTION文
- ALTER PACKAGE文
- ALTER PROCEDURE文

- ALTER TRIGGER文
- ALTER TYPE文

その他のドキュメント:

- PL/SQLユニットに対するデータバインドされた照合の指定の詳細は、Oracle Databaseグローバリゼーション・サポート・ガイドを参照してください
- 有効なスキーマのデフォルトの照合の詳細は、Oracle Databaseグローバリゼーション・サポート・ガイドを参照してください

13.21 DELETE文の拡張機能

SQLのDELETE文のwhere_clauseに対するPL/SQLの拡張機能によって、CURRENT OF句を指定することができ、この句を指定すると、DELETE文は指定したカーソルの現在の行に制限されます。

CURRENT OF句の詳細は、「UPDATE文の拡張機能」を参照してください。

関連項目:

SQLのDELETE文の構文は、『Oracle Database SQL言語リファレンス』を参照してください。

13.22 DEPRECATEプラグマ

DEPRECATEプラグマは、PL/SQL要素を非推奨としてマークします。コンパイラは、DEPRECATEプラグマまたは非推奨要素の使用に対して警告を発行します。

関連付けられた警告は非推奨要素のユーザーに対して、非推奨化が原因で他のコードの変更が必要になる可能性があることを示します。

ここでのトピック

- 構文
- セマンティクス
- DEPRECATEプラグマによるコンパイルの警告
- 例
- 関連トピック

構文

deprecate_pragma ::=

セマンティクス

deprecate_pragma

DEPRECATEプラグマは、パッケージ仕様部、オブジェクト仕様部、最上位プロシージャまたは最上位ファンクションの宣言セクションにのみ記述できます。

次の種類のPL/SQL要素を非推奨にすることができます。

- サブプログラム
- パッケージ
- 変数
- 定数
- 型
- サブタイプ
- 例外
- カーソル

DEPRECATEプラグマは、PL/SQLユニットの宣言セクションにのみ記述できます。非推奨になる項目の宣言の直後に記述する必要があります。

DEPRECATEプラグマは、プラグマに先行する宣言で指定されたPL/SQL要素に適用されます。

DEPRECATEプラグマがパッケージ仕様部、オブジェクト仕様部またはサブプログラムに適用される場合、定義の宣言部分を終了させるキーワードISまたはASの直後にプラグマを記述する必要があります。

DEPRECATEプラグマがパッケージ仕様部またはオブジェクト仕様部に適用される場合、仕様部で宣言されたすべての要素(非推奨化できる種類のもの)への参照も非推奨になります。

DEPRECATEプラグマがサブプログラムの宣言に適用される場合、そのサブプログラムのみが影響を受けます。同じ名前の他のオーバーロードは非推奨になりません。

DEPRECATEプラグマの使用においてオプションのカスタム・メッセージを記述すると、非推奨要素への参照に対して発行される警告にカスタム・メッセージが追加されます。

DEPRECATEプラグマ内の識別子は、そのプラグマが適用される宣言内の要素を指定する必要があります。

非推奨化は、型の導出中に継承されます。親が非推奨になった子オブジェクト型は非推奨になりません。継承された属性およびメソッドのみが非推奨になります。

ベース型が非推奨にならず、個々のメソッドまたは属性が非推奨になり、型がこの型から導出され、非推奨になった型またはメソッドが継承される場合、導出された型からこれらを参照すると、コンパイラによって警告が発行されます。

非推奨化プラグマまたはその本体があるユニット以外の場所に記述されている非推奨要素への参照があると、PL/SQLコンパイラは参照された要素に関する警告を発行します。無名ブロック内の非推奨要素への参照があっても、コンパイラは警告を発行しません。名前付きエンティティ内の参照にかぎり、警告が発生します。

非推奨になったエンティティが非推奨になった他のエンティティの定義内で参照された場合、警告は発生しません。

古いクライアント・コードが非推奨になった要素を参照すると、そのコードは無効になり、再コンパイルされます。警告は発生しません。

SQLコードが非推奨要素を直接参照しても影響はありません。

PL/SQLの静的SQL文内の非推奨要素への参照があると、PL/SQLコンパイラが警告を発行する場合があります。ただし、そのような参照は検出できない可能性があります。

pls_identifier

非推奨になるPL/SQL要素の識別子。

character_literal

オプションのコンパイル時の警告メッセージ。
DEPRECATEプラグマによるコンパイルの警告

PL/SQLコンパイラは、DEPRECATEプラグマが使用されている場合、かつ非推奨になった項目が参照された場合に警告を発行します。

- 6019 — このエンティティは非推奨になりました。今後のリリースで削除される可能性があります。非推奨になったエンティティは使用しないでください。
- 6020 — 参照されたエンティティは非推奨になりました。今後のリリースで削除される可能性があります。非推奨に

なったエンティティは使用しないでください。警告に特定の指示がある場合は、それに従ってください。

- 6021 — プラグマの置き間違い。プラグマDEPRECATEは、非推奨になるエンティティの宣言の直後に続ける必要があります。非推奨になるエンティティの宣言の直後にプラグマを置いてください。
- 6022 — このエンティティを非推奨にすることはできません。非推奨化が適用されるのは、パッケージ仕様部または型仕様部で宣言できるエンティティ、ならびに最上位のプロシージャおよびファンクション定義に限定されます。プラグマを削除してください。

DEPRECATEプラグマの警告は、PLSQL_WARNINGSパラメータまたはDBMS_WARNINGパッケージを使用して管理できます。

例

例13-9 非推奨化の警告の有効化

この例は、PLSQL_WARNINGSパラメータを設定して、セッションでこれらの警告を有効にする方法を示しています。

Live SQL:

この例は、Oracle Live SQL の「同じスキーマの最上位プロシージャへのアクセスの制限」で表示および実行できます

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:(6019,6020,6021,6022)';
```

例13-10 PL/SQLパッケージの非推奨化

この例は、PL/SQLパッケージ全体の非推奨化を示しています。警告は、パッケージpack1へのすべての参照に対して、ならびにパッケージおよびその本体の外部で使用されたプロシージャfooおよびbarに対して発行されます。

Live SQL:

この例は、Oracle Live SQL の「PL/SQL パッケージの非推奨化」で表示および実行できます

```
PACKAGE pack1 AS
PRAGMA DEPRECATE(pack1); PROCEDURE foo; PROCEDURE bar; END pack1;
```

例13-11 カスタム警告によるPL/SQLパッケージの非推奨化

この例は、PL/SQLパッケージの非推奨化を示します。非推奨になったプロシージャfooに対する参照が別のユニットでコンパイルされると、コンパイラはカスタム警告メッセージを発行します。

Live SQL:

この例は、Oracle Live SQL の「カスタム警告による PL/SQL パッケージの非推奨化」で表示および実行できます

```
PACKAGE pack5 AUTHID DEFINER AS
PRAGMA DEPRECATE(pack5 , 'Package pack5 has been deprecated, use new_pack5 instead.');
PROCEDURE foo; PROCEDURE bar; END pack5;
```

別のユニットでプロシージャpack5.fooへの参照があると、次のような警告が発生します。

```
SP2-0810: Package Body created with compilation warnings
Errors for PACKAGE BODY PACK6:
4/10     PLW-06020: reference to a deprecated entity: PACK5 declared in unit PACK5[1,9].
         Package pack5 has been deprecated, use new_pack5 instead
```

例13-12 PL/SQLプロシージャの非推奨化

この例は、パッケージpack7内の単一のPL/SQLプロシージャfooの非推奨化を示しています。

Live SQL:

この例は、Oracle Live SQL の「PL/SQL プロシージャの非推奨化」で表示および実行できます

```
PACKAGE pack7 AUTHID DEFINER AS
```

PROCEDURE foo; PRAGMA DEPRECATE (foo, 'pack7.foo is deprecated, use pack7.bar instead.');
PROCEDURE bar; END pack7;

例13-13 オーバーロードされたプロシージャの非推奨化

この例は、DEPRECATEプラグマがプロシージャ名の特定のオーバーロードのみに適用されることを示しています。proc1の2番目の宣言のみが非推奨になります。

Live SQL:

この例は、Oracle Live SQL の「オーバーロードされたプロシージャの非推奨化」で表示および実行できます

```
PACKAGE pack2 AS
```

PROCEDURE proc1(n1 NUMBER, n2 NUMBER, n3 NUMBER); -- Only the overloaded procedure with 2 arguments is deprecated PROCEDURE proc1(n1 NUMBER, n2 NUMBER); PRAGMA DEPRECATE(proc1); END pack2;

例13-14 定数と例外の非推奨化

Live SQL:

この例は、Oracle Live SQL の「定数と例外の非推奨化」で表示および実行できます

この例では、定数と例外の非推奨化を示します。

```
PACKAGE trans_data AUTHID DEFINER AS
   TYPE Transrec IS RECORD (
     accounttype VARCHAR2(30) ,
     ownername VARCHAR2(30) ,
     balance REAL
   );
   min_balance constant real := 10.0;
```

```
   PRAGMA DEPRECATE(min_balance , ' Minimum balance requirement has been removed.' );
   insufficient_funds EXCEPTION;
   PRAGMA DEPRECATE (insufficient_funds , ' Exception no longer raised.' );
END trans_data;
```

例13-15 条件付きコンパイルを使用した一部のデータベース・リリースでのエンティティの非推奨化

この例は、データベース・リリース・バージョンが11より後の場合のプロシージャproc1の非推奨化を示しています。

Live SQL:

この例は、Oracle Live SQL の「条件付きコンパイルを使用した一部のデータベース・リリースでのエンティティの非推奨化」で表示および実行できます

```
CREATE PACKAGE pack11 AUTHID DEFINER AS
```

$IF DBMS_DB_VERSION.VER_LE_11 $THEN PROCEDURE proc1; $ELSE PROCEDURE proc1; PRAGMA DEPRECATE(proc1); $END PROCEDURE proc2; PROCEDURE proc3; END pack11;

例13-16 オブジェクト型の非推奨化

この例は、オブジェクト型全体の非推奨化を示しています。

Live SQL:

この例は、Oracle Live SQL の「オブジェクト型の非推奨化」で表示および実行できます

```
TYPE type01 AS OBJECT(
```

PRAGMA DEPRECATE (type01), y NUMBER, MEMBER PROCEDURE proc(x NUMBER), MEMBER PROCEDURE proc2(x NUMBER));

例13-17 オブジェクト型仕様部のメンバー・ファンクションの非推奨化

この例は、オブジェクト型仕様部のメンバー・ファンクションadd2の非推奨化を示しています。

Live SQL:

この例は、Oracle Live SQL の「オブジェクト型仕様部のメンバー・ファンクションの非推奨化」で表示および実行できます

```
TYPE objdata AS OBJECT(
```

n1 NUMBER , n2 NUMBER , n3 NUMBER , MEMBER FUNCTION add2 RETURN NUMBER , PRAGMA DEPRECATE (add2), MEMBER FUNCTION add_all RETURN NUMBER);

例13-18 継承されたオブジェクト型の非推奨化

この例は、ユニットtype15_basetypeの型本体で宣言された、非推奨になったエンティティxへの参照があると、コンパイラが警告を発行することを示しています。

Live SQL:

この例は、Oracle Live SQL の「継承されたオブジェクト型の非推奨化」で表示および実行できます

```
TYPE type15_basetype AS OBJECT
```

(x1 NUMBER, x NUMBER, PRAGMA DEPRECATE (x), MEMBER PROCEDURE f0 , PRAGMA DEPRECATE (f0), MEMBER PROCEDURE f1 , PRAGMA DEPRECATE (f1), MEMBER PROCEDURE f2 , PRAGMA DEPRECATE (f2), MEMBER PROCEDURE f3) NOT FINAL;

```
TYPE BODY type15_basetype AS
   MEMBER PROCEDURE f0
   IS
   BEGIN
     x := 1;
   END;
   MEMBER PROCEDURE f1
   IS
   BEGIN
     x := 1;
   END;
   MEMBER PROCEDURE f2
   IS
   BEGIN
     x := 1;
   END;
   MEMBER PROCEDURE f3
   IS
   BEGIN
     x := 1;
   END;
END;
```

type15_basetypeの型本体の非推奨になったエンティティx、f0およびf2への参照があると、コンパイラが警告を発行します。

```
TYPE type15_subtype UNDER type15_basetype (
  y NUMBER ,
  MEMBER PROCEDURE f1(z NUMBER),
  MEMBER PROCEDURE f1(z NUMBER , m1 NUMBER),
  PRAGMA DEPRECATE(f1),
  OVERRIDING MEMBER PROCEDURE f2
);

TYPE BODY type15_subtype AS
   MEMBER PROCEDURE f1(z NUMBER)
IS
BEGIN
  -- deprecation attribute inherited in derived type.
  x := 1;
  x1:= 2;
  SELF.f0;
END;
   MEMBER PROCEDURE f1(z  NUMBER ,
                       m1 NUMBER)
   IS
```

```
  BEGIN
    NULL;
  END;
  OVERRIDING MEMBER PROCEDURE f2
  IS
  BEGIN
    /* refer to deprecated f2 in supertype */
    (SELF AS type15_basetype).f2;
    /* No warning for a reference to a not deprecated data member in the supertype */
    x1 := 1;
  END;
END;
```

ユニットtype15_basetypeの非推奨になったエンティティx、f1およびf0への参照があると、コンパイラが警告を発行します。

```
PROCEDURE test_types3
AS
  e type15_subtype ;
  d type15_basetype ;
BEGIN
  e := type15_subtype (1 ,1 ,1);
  d := type15_basetype (1, 1);
  d.x := 2; -- warning issued
  d.f1;     -- warning issued
  e.f1 (4); -- overloaded in derived type. no warning. not deprecated in the derived type.
  e.f1 (1); -- no warning
  e.f0;     -- f0 is deprecated in base type. deprecation is inherited. warning issued
            -- warning issued for deprecated x in d.x and e.x
  DBMS_OUTPUT.PUT_LINE(to_char(e.x) || to_char(' ') || to_char(d.x));
END;
```

例13-19 最上位のサブプログラムのみに適用される非推奨化

この例は、DEPRECATEプラグマを使用して、ネストしたプロシージャを非推奨化できないことを示しています。コンパイラは、エンティティでのプラグマの誤用について警告を発行します。プラグマによる影響はありません。

Live SQL:

この例は、Oracle Live SQL の「最上位のサブプログラムのみに適用される非推奨化」で表示および実行できます

```
PROCEDURE foo
```

IS PROCEDURE inner_foo IS PRAGMA DEPRECATE (inner_foo, 'procedure inner_foo is deprecated'); BEGIN DBMS_OUTPUT.PUT_LINE('Executing inner_foo'); END; BEGIN DBMS_OUTPUT.PUT_LINE('Executing foo'); END;

例13-20 DEPRECATEプラグマの置き間違い

DEPRECATEプラグマは、非推奨になる項目の宣言の直後に記述する必要があります。プラグマの置き間違いに関する警告が発行され、プラグマによる影響はありません。

Live SQL:

この例は、Oracle Live SQL の「DEPRECATE プラグマの置き間違い」で表示および実行できます

```
PROCEDURE bar
IS BEGIN PRAGMA DEPRECATE(bar); DBMS_OUTPUT.PUT_LINE('Executing bar.'); END;
```

例13-21 要素名とDEPRECATEプラグマの引数の不一致

この例は、プラグマの引数が宣言内の名前と一致しない場合、プラグマが無視され、コンパイラによる警告の発行がないことを示しています。

Live SQL:

この例は、Oracle Live SQL の「要素名と DEPRECATE プラグマの引数の不一致」で表示および実行できます

```
PACKAGE pkg13
AS PRAGMA DEPRECATE ('pkg13', 'Package pkg13 is deprecated, use pkg03'); Y NUMBER; END pkg13;
```

識別子が一致しない名前で適用されると、コンパイラは、プラグマが置き間違えられていることに関する警告を発行します。プラグマによる影響はありません。

```
CREATE PACKAGE pkg17
IS
  PRAGMA DEPRECATE ("pkg17");
END pkg17;
```

関連トピック

このドキュメント:

- プラグマ
- PLSQL_WARNINGSコンパイル・パラメータの設定の詳細は、PL/SQLユニットおよびコンパイル・パラメータを参照してください

その他のドキュメント:

- パッケージ、サブプログラムおよび型の非推奨化の詳細は、Oracle開発ガイドを参照してください
- DBMS_WARNING.ADD_WARNING_SETTING_NUMプロシージャを使用した非推奨化の警告の有効化に関する詳細は、Oracle Database PL/SQLパッケージおよびタイプ・リファレンスを参照してください
- コンパイルの警告の詳細は、コンパイル時の警告を参照してください。

13.23 DETERMINISTIC句

deterministicオプションは、予測可能な結果を戻し、かつ副作用をもたらさないファンクションをマークします。

ファンクション索引、PL/SQLファンクションを使用する仮想列定義、およびクエリーリライトが有効なマテリアライズド・ビューは、特別なファンクション・プロパティを必要とします。DETERMINISTIC句は、ファンクションにそれらのプロパティがあることを示します。

DETERMINISTICオプションは次の文に指定できます。

- ファンクションの宣言および定義
- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE TYPE BODY文

ここでのトピック

- 構文
- セマンティクス
- 使用上のノート
- 関連トピック

構文

deterministic_clause ::=

セマンティクス

deterministic_clause

DETERMINISTIC

ファンクションの宣言または定義にDETERMINISTIC句が指定されている場合、そのファンクションは決定的です。

DETERMINISTIC句は、ファンクションの宣言と定義でそれぞれ1回指定できます。

決定的ファンクションは、2つの異なる起動に対して、それらに指定された引数が同じであれば、同じ値を戻す必要があります。

DETERMINISTICファンクションは、副作用をもたらさないものとします。

DETERMINISTICファンクションは、未処理例外を生成しないものとします。

DETERMINISTIC句のあるファンクションがこれらのセマンティクス・ルールのいずれかに違反すると、その起動の結果、その値、およびその起動元への影響はすべて未定義になります。

使用上のノート

DETERMINISTIC句は、ファンクションがセマンティクス・ルールに従っていることを示すものです。ファンクションに違反があった場合、コンパイラ、SQL実行またはPL/SQL実行のいずれにおいても問題を診断できず、誤った結果が暗黙的に生成される可能性

があります。

ファンクション索引の式(仮想列定義内)でファンクションを起動する場合、あるいはREFRESH FASTまたはENABLE QUERY REWRITEとマークされたマテリアライズド・ビューの問合せからファンクションを起動する場合、このキーワードを指定する必要があります。データベースでは、決定的ファンクションが検出されると、ファンクションが再実行されるのではなく、以前に計算した結果の使用が可能なかぎり試行されます。このファンクションを変更する場合は、依存するすべてのファンクション索引およびマテリアライズド・ビューを手動で再構築する必要があります。

起動ごとに結果が変わる可能性があるため、結果がセッション変数またはスキーマ・オブジェクトの状態に依存するファンクションには、DETERMINISTICを指定しないでください。

パッケージ変数を使用するファンクション、またはファンクションから戻される結果に影響を与える可能性のある方法でデータベースにアクセスするファンクションを定義する場合は、この句を指定しないでください。

多相テーブル・ファンクションでこの句を指定することはできません。

DETERMINISTICオプションが記述されている場合、コンパイラはマークを使用して、ファンクション実行のパフォーマンスを高めることができます。

プログラミングにおいては、次のDETERMINISTICカテゴリに属するファンクションを作成するのが適切です。

- WHERE句、ORDER BY句またはGROUP BY句の中で使用されるファンクション
- SQL型のMAPメソッドまたはORDERメソッドであるファンクション
- 結果セットに行を入れるかどうか、またはどこに入れるかを決定するファンクション

関連トピック

他の章:

- サブプログラムの副作用

その他のドキュメント:

- 『Oracle Database SQL言語リファレンス』のCREATE INDEX文に関する項
- マテリアライズド・ビューの詳細は、『Oracle Databaseデータ・ウェアハウス・ガイド』を参照してください。
- ファンクション索引の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

13.24 要素の仕様

要素の仕様でADTの各属性を指定します。

要素の仕様は、次のSQL文に指定できます。

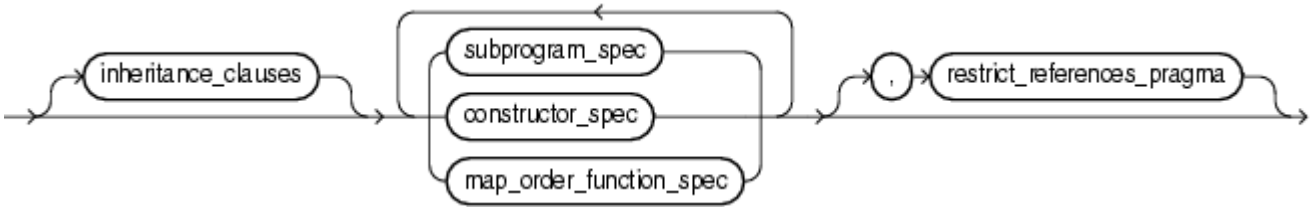
- ALTER TYPE文
- CREATE TYPE文

ここでのトピック

- 構文
- セマンティクス

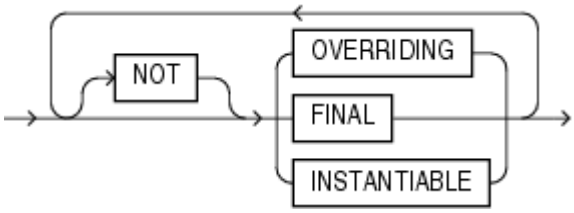
構文

element_spec ::=

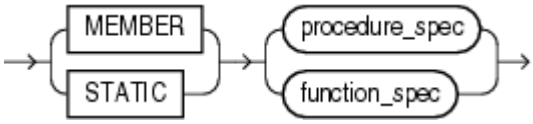


(subprogram_spec ::= , constructor_spec ::=, map_order_function_spec ::=, restrict_references_pragma ::=)

inheritance_clauses ::=

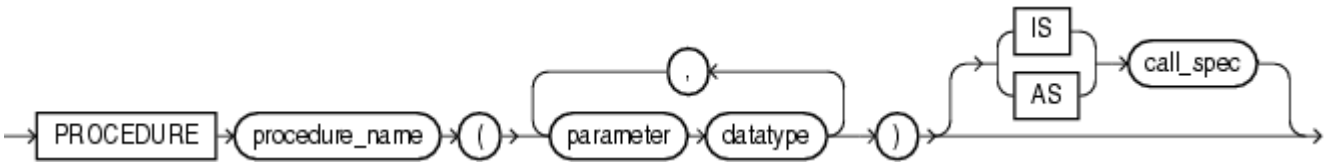


subprogram_spec ::=



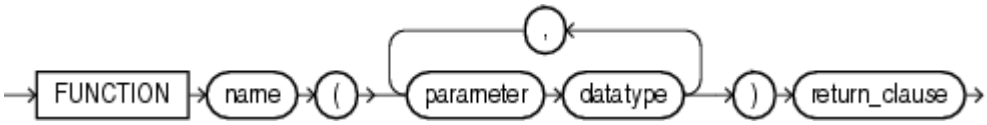
(procedure_spec ::=, function_spec ::=)

procedure_spec ::=

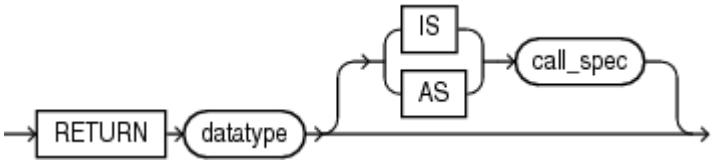


(call_spec ::=)

function_spec ::=

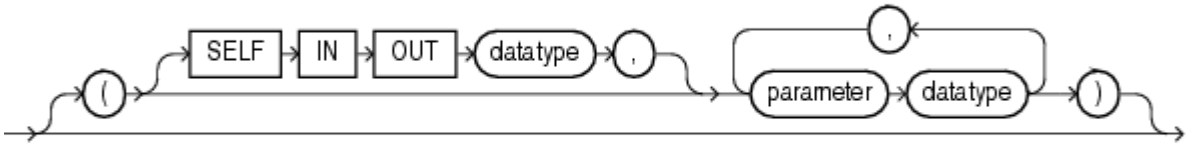


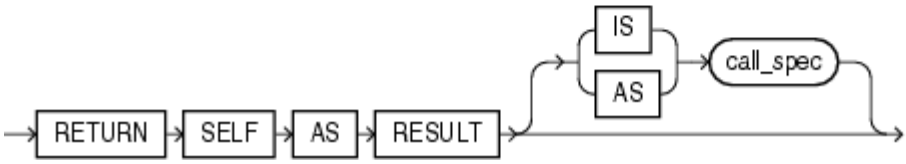
return_clause ::=



(call_spec ::=)

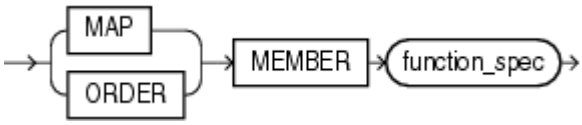
constructor_spec ::=





(call_spec ::=)

map_order_function_spec ::=



(function_spec ::=)

セマンティクス

element_spec

inheritance_clauses

スーパータイプとサブタイプの間の関係を指定します。

[NOT] OVERRIDING

スーパータイプで定義されているMEMBERメソッドがこのメソッドによってオーバーライドされるように指定します。このキーワードは、このメソッドでスーパータイプのメソッドを再定義する場合には必須です。デフォルト: NOT OVERRIDING。

[NOT] FINAL

このメソッドがこの型のサブタイプによってオーバーライドされないように指定します。デフォルト: NOT FINAL。

[NOT] INSTANTIABLE

このメソッドが型で実装されないように指定します。デフォルト: すべてのメソッドがINSTANTIABLEです。

NOT INSTANTIABLEの制限

NOT INSTANTIABLEを指定した場合は、FINALもSTATICも指定できません。

関連項目:

constructor_spec

subprogram_spec

ADT属性として参照されるサブプログラムを指定します。このようなサブプログラムには、それぞれの対応するメソッド本体をADT本体に指定する必要があります。

subprogram_specの制限

MEMBERメソッドをサブタイプ内で再定義するサブタイプに対してSTATICメソッドを定義することはできません。逆も同様です。

MEMBER

属性として参照されるADTに関連付けられたサブプログラム。通常、object_expression.method()などの自己参照的スタイルでMEMBERメソッドを起動します。このクラスのメソッドには、メソッド本体でSELFとして参照される暗黙的な最初の引数があり、この引数は、メソッドが起動されるオブジェクトを表します。

関連項目:

「例14-30」

STATIC

ADTに関連付けられたサブプログラムです。MEMBERメソッドとは異なり、STATICメソッドに暗黙的なパラメータはありません。本体でSELFを参照することはできません。通常、type_name.method()として起動されます。

STATICの制限

- JavaクラスのMEMBERメソッドをSQLJオブジェクト型のSTATICメソッドにマップすることはできません。
- MEMBERメソッドの場合も、STATICメソッドの場合も、プロシージャの仕様部またはファンクションの仕様部に対応するメソッド本体を型本体に指定する必要があります。

関連項目:

「例14-31」

procedure_specまたはfunction_spec

プロシージャまたはファンクションのパラメータおよびデータ型を指定します。このサブプログラムにプロシージャまたはファンクションの

宣言が含まれていない場合は、対応するCREATE TYPE BODY文を発行する必要があります。

procedure_specまたはfunction_specの制限

サブタイプを作成する場合、プロシージャまたはファンクションの名前は、継承された属性かどうかに関係なく、連鎖するスーパータイプで宣言されているどの属性とも同じ名前にできません。

return_clause

return_clauseの最初の形式はファンクションでのみ有効です。示されている構文は、省略形です。

関連項目:

メソッドおよびメソッドの起動の詳細は、「コレクション・メソッドの起動」を参照してください

constructor_spec

ADTの初期化インスタンスを戻すファンクションであるユーザー定義コンストラクタを作成します。各コンストラクタのパラメータの数、順序またはデータ型が異なる場合は、1つのADTに対して複数のコンストラクタを宣言できます。

- ユーザー定義コンストラクタ・ファンクションは常にFINALおよびINSTANTIABLEであるため、これらのキーワードはオプションです。
- ユーザー定義コンストラクタのパラメータの引渡しモードは常にSELF IN OUTです。そのため、意味を明確にするために指定する場合を除き、この句を指定する必要はありません。
- RETURN SELF AS RESULTは、コンストラクタによって戻される値の実行時型が、SELF引数の実行時型になるように指定します。

関連項目:

ユーザー定義コンストラクタの詳細および例は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』および「例14-29」を参照してください

map_order_function_spec

宣言するMEMBERまたはSTATICメソッドの数に関係なく、型仕様部には、1つのMAPメソッドまたは1つのORDERメソッドを宣言できます。いずれかのメソッドを宣言すると、SQLでオブジェクト・インスタンスを比較できます。

いずれのメソッドも宣言しない場合、オブジェクト・インスタンスに関して比較できるのは等しいか等しくないかについてのみです。同じ型定義のインスタンスは、それぞれの対応する属性の各ペアが等しい場合にのみ等しくなります。2つのADTが等しいかどうかを判断するために、比較メソッドを指定しないでください。

サブタイプにはMAPおよびORDERメソッドのいずれも宣言できません。ただし、スーパータイプでNOT FINAL MAPメソッドを定義する場合は、サブタイプでMAPメソッドをオーバーライドできます。サブタイプでは、ORDERメソッドはオーバーライドできません。

JavaクラスからSQL型へのマップ時に、MAPメソッドまたはORDERメソッドのいずれかを指定できます。ただし、MAPメソッドおよび

ORDERメソッドは、JavaクラスのMEMBERファンクションにマップする必要があります。

MAPメソッドもORDERメソッドも指定されていない場合、比較できるのは等しいか等しくないかについてのみとなります。したがって、オブジェクト・インスタンスに順序を付けることはできません。同じ型定義のインスタンスは、それぞれの対応する属性の各ペアが等しい場合にのみ等しくなります。2つのADTが等しいかどうかを判断するために、比較メソッドを指定する必要はありません。

オブジェクト・インスタンスで大規模のソートまたはハッシュ結合操作を実行する場合は、MAPメソッドを使用します。MAPメソッドを1回適用すると、オブジェクトがスカラー値にマップされ、ソート中およびマージ中にデータベースによってそのスカラーが使用されます。MAPメソッドは、各オブジェクトを比較するメソッドを起動する必要があるORDERメソッドより効率的です。ハッシュ結合ではMAPメソッドを使用する必要があります。ハッシュ・メカニズムはオブジェクト値でハッシュするため、ORDERメソッドを使用することはできません。

関連項目:

オブジェクト値の比較の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

MAP MEMBER

オブジェクトのすべてのインスタンスの順序における特定のインスタンスの相対的な位置を戻すMAPメンバー・ファンクション(MAPメソッド)を指定します。MAPメソッドは暗黙的にコールされ、オブジェクト・インスタンスを事前定義済のスカラー型の値にマップすることによって、それらのオブジェクト・インスタンスに順序を設定します。PL/SQLでは、順序付けを使用してブール式の評価と比較を行います。

MAPメソッドの引数がNULLの場合は、MAPメソッドによってNULLが戻され、メソッドは起動されません。

オブジェクトの仕様部には1つのMAPメソッドのみを含めることができ、このメソッドはファンクションである必要があります。結果として生成される型は、事前定義済のSQLスカラー型である必要があり、MAPメソッドに指定できる引数は、暗黙的なSELF引数のみです。

> ノート:
>
> ソート(ORDER BY 句、GROUP BY 句、DISTINCT 句または UNION 句を使用)または結合を含む問合せで type_name が参照される場合に、それらの問合せをパラレル化するには、MAP メンバー・ファンクションを指定する必要があります。

サブタイプでは、新しいMAPメソッドを定義することはできませんが、継承されたMAPメソッドをオーバーライドすることはできます。

ORDER MEMBER

オブジェクトのインスタンスを明示的な引数および暗黙的なSELF引数として取り、負の整数、0(ゼロ)または正の整数のいずれかを戻すORDERメンバー・ファンクション(ORDERメソッド)を指定します。負の整数、0(ゼロ)または正の整数は、暗黙的なSELF引数が明示的な引数より小さいか、等しいかまたは大きいことを示します。

ORDERメソッドのいずれかの引数がNULLの場合は、ORDERメソッドによってNULLが戻され、メソッドは起動されません。

同じADT定義のインスタンスをORDER BY句で比較すると、ORDERメソッド・ファンクションが起動されます。

オブジェクトの仕様部には1つのORDERメソッドのみを含めることができ、このメソッドは戻り型NUMBERを持つファンクションである必要があります。

サブタイプでは、ORDERメソッドを定義およびオーバーライドすることはできません。

map_order_function_specの制限

サブタイプにORDERメソッドを追加することはできません。

restrict_references_pragma

非推奨の句です。「RESTRICT_REFERENCESプラグマ」を参照してください。

restrict_references_pragmaの制限

この句は、メソッドを削除する場合には無効です。

13.25 EXCEPTION_INITプラグマ

EXCEPTION_INITプラグマは、ユーザー定義の例外名をエラー・コードに関連付けます。

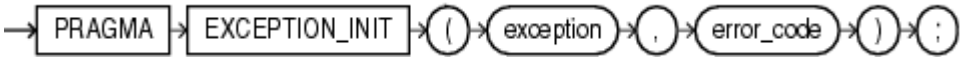
EXCEPTION_INITプラグマは、関連付けられた例外と同じ宣言部内の例外宣言の後の任意の場所に指定できます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

exception_init_pragma ::=



セマンティクス

exception_init_pragma

exception

事前に宣言されているユーザー定義の例外の名前。

error_code

exceptionに関連付けるエラー・コード。error_codeには、100(「SQLCODEファンクション」が戻す「データが見つかりません。」)か、または-1403(「データが見つかりません。」の別の数値コード)を除く-10000000より大きい負の整数のいずれかを使用できます。

ノート:

NO_DATA_FOUND は事前定義の例外です。

2つのEXCEPTION_INITプラグマが、同じユーザー定義例外に対して異なるエラー・コードを割り当てている場合、後のプラグマが前のプラグマをオーバーライドします。

例

- 例11-5、内部的に定義された例外への名前付け
- 例11-13、RAISE_APPLICATION_ERRORによるユーザー定義の例外の呼出し
- 例12-13、FORALL文が完了した後のFORALL例外の処理

関連トピック

この章:

- 例外宣言
- 例外ハンドラ
- SQLCODEファンクション
- SQLERRMファンクション

他の章:

- 内部的に定義された例外
- 「RAISE_APPLICATION_ERRORプロシージャ」

13.26 例外宣言

例外宣言は、ユーザー定義の例外の名前を宣言します。

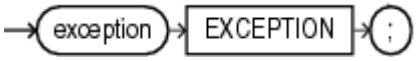
この名前は、EXCEPTION_INITプラグマを使用して、内部的に定義された例外に代入することができます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

exception_declaration ::=



セマンティクス

exception_declaration

exception

宣言する例外の名前。

exceptionの制限

exceptionは、EXCEPTION_INITプラグマ、RAISE文、RAISE_APPLICATION_ERRORの起動、または例外ハンドラでのみ使用できます。

注意:

exception に対しては、事前定義の例外の名前を付けないことをお薦めします。詳細は、「事前定義の例外の再宣言」を参照してください。事前定義の例外の名前のリストは、表 11-3 を参照してください。

例

- 例11-5、内部的に定義された例外への名前付け
- 例11-9、再宣言された事前定義の識別子
- 例11-10、ユーザー定義の例外の宣言、呼出しおよび処理

関連トピック

この章:

- EXCEPTION_INITプラグマ

- 例外ハンドラ
- 「RAISE文」

他の章:

- 内部的に定義された例外
- ユーザー定義の例外

13.27 例外ハンドラ

例外ハンドラは、呼び出された例外を処理します。

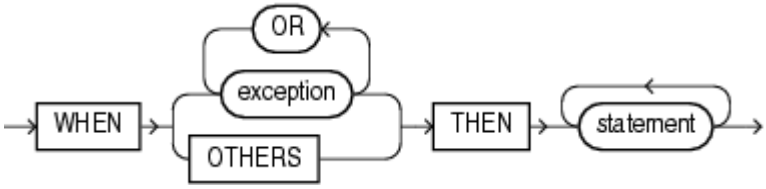
例外ハンドラは、無名ブロック、サブプログラム、トリガーおよびパッケージの例外処理部で使用できます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

exception_handler ::=



(statement ::=)

セマンティクス

exception_handler

exception

事前定義の例外(表11-3 を参照)またはユーザー定義の例外(例外宣言を参照)の名前。

指定した例外がPL/SQLによって呼び出されると、それに関連付けられた文が実行されます。

OTHERS

ブロックの例外処理部で明示的に指定していないすべての例外を指定します。このような例外がPL/SQLによって呼び出されると、それに関連付けられた文が実行されます。

> ノート:
>
> OTHERS 例外ハンドラの最後の文には、RAISE または RAISE_APPLICATION_ERROR プロシージャの起動のいずれかを使用することをお薦めします。
>
> この方法に従わない場合は、PL/SQL 警告が有効になり、PLW-06009 が表示されます。

ブロックの例外処理部では、WHEN OTHERS例外ハンドラはオプションです。これはブロックの例外処理部内の最後の例外ハンドラとして、1回のみ使用できます。

例

- 例11-3、複数の例外に対応した1つの例外ハンドラ
- 例11-4、例外ハンドラを共有する文のロケータ変数
- 例11-6、ZERO_DIVIDEを処理する無名ブロック
- 例11-7、ZERO_DIVIDEを回避する無名ブロック
- 例11-10、ユーザー定義の例外の宣言、呼出しおよび処理
- 例11-14、有効範囲を超えて伝播した例外が処理される場合
- 例11-24、例外ハンドラの実行と実行の終了
- 例11-25、例外ハンドラの実行と実行の継続
- 例12-12、FORALL例外の即時処理
- 例12-13、FORALL文が完了した後のFORALL例外の処理

関連トピック

この章:

- ブロック
- EXCEPTION_INITプラグマ
- 「例外宣言」
- 「RAISE文」
- 「SQLCODEファンクション」
- 「SQLERRMファンクション」

他の章:

- 「例外処理の概要」
- 「例外処理後に実行を続ける方法」
- 「例外処理後のトランザクションの再試行」
- 「CREATE PACKAGE BODY文」
- CREATE TRIGGER文

13.28 EXECUTE IMMEDIATE文

EXECUTE IMMEDIATE文は、動的SQL文を一度の操作で作成して実行します。

システム固有の動的SQLでは、EXECUTE IMMEDIATE文を使用してほとんどの動的SQL文を処理します。

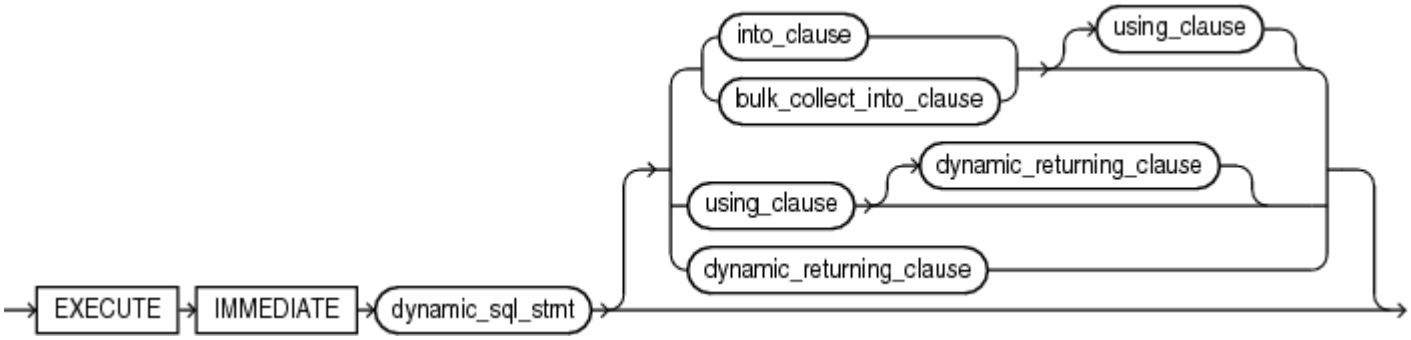
> 注意:
>
> 動的 SQL を使用する場合は、セキュリティ・リスクの SQL インジェクションに注意してください。SQL インジェクションの詳細は、「SQL インジェクション」を参照してください。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

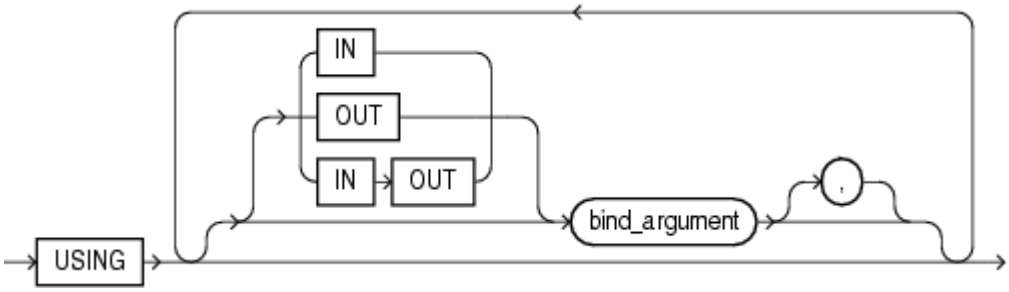
構文

execute_immediate_statement ::=



(bulk_collect_into_clause ::=、dynamic_returning_clause ::=、into_clause ::=)

using_clause ::=



セマンティクス

execute_immediate_statement

dynamic_sql_stmt

SQL文を表す文字列リテラル、文字列変数または文字列式です。その型は、CHAR型、VARCHAR2型またはCLOB型である必要があります。

> ノート:
>
> dynamic_sql_statement が SELECT 文の場合、into_clause と bulk_collect_into_clause の両方を省略すると、execute_immediate_statement は実行されません。
>
> たとえば、次の文では順序を増分しません。
>
> ```
> EXECUTE IMMEDIATE 'SELECT S.NEXTVAL FROM DUAL'
> ```

into_clause

文によって戻される列値を格納する変数またはレコードを指定します。この句の詳細は、「RETURNING INTO句」を参照してください。

into_clauseの制限

dynamic_sql_stmtが単一の行を返す場合にのみ使用します。

bulk_collect_into_clause

文によって戻される行を格納する1つ以上のコレクションを指定します。この句の詳細は、「RETURNING INTO句」を参照してください。

bulk_collect_into_clauseの制限

dynamic_sql_stmtが複数の行を返すことができる場合にのみ使用します。

dynamic_returning_clause

動的SQL文によって影響を受ける行の列値を個々の変数またはレコードのいずれかに戻します。この句の詳細は、「RETURNING INTO句」を参照してください。

dynamic_returning_clauseの制限

dynamic_sql_stmtがRETURNING INTO句を持つ場合にのみ使用します。

using_clause

位置表記法を使用して、バインド変数を指定します。

> ノート:
>
> dynamic_sql_statement 内でプレースホルダ名を繰り返す場合は、プレースホルダをバインド変数に関連付ける方法が動的 SQL 文の種類によって異なるということに注意してください。詳細は、「動的 SQL 文内でのプレース

ホルダ名の繰返し」を参照してください。

using_clauseの制限

- dynamic_sql_stmtにバインド変数用のプレースホルダが含まれている場合にのみ使用します。
- dynamic_sql_stmtにRETURNING INTO句(static_returning_clause)がある場合は、using_clauseにはINバインド変数のみを含めることができます。定義上、RETURNING INTO句のバインド変数はOUTバインド変数です。

IN、OUT、IN OUT

バインド変数のパラメータ・モード。INバインド変数は、その値をdynamic_sql_stmtに渡します。OUTバインド変数は、dynamic_sql_stmtによって戻される値を格納します。IN OUTバインド変数は、その初期値をdynamic_sql_stmtに渡し、dynamic_sql_stmtによって戻される値を格納します。デフォルト: IN。

RETURNING句を持つDML文の場合は、常にOUTであるパラメータ・モードを指定しなくても、OUTバインド変数をRETURNING INTO句に入れることができます。

bind_argument

実行時に、dynamic_sql_stmt内の対応するプレースホルダのかわりに置き換えられる値を持つ式。

dynamic_sql_stmt内のすべてのプレースホルダをUSING句内またはRETURNING INTO句内(あるいはその両方)のbind_argument、またはINTO句内の定義変数に関連付ける必要があります。

dynamic_sql_stmtは、バインド変数の異なる値を使用して繰り返し実行できます。ただし、EXECUTE IMMEDIATEは実行のたびに動的文字列を準備するため、オーバーヘッドが発生します。

ノート:

バインド変数はどのような順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

bind_argumentの制限

- bind_argumentに、文字列で索引付けされている連想配列は使用できません。
- bind_argumentに、予約語NULLは使用できません。

  値NULLを動的SQL文に渡すには、例7-7で示すように、NULLを使用する場所に未初期化変数を使用します。

例

- 例7-1、動的PL/SQLブロックからのサブプログラムの起動
- 例7-7、USING句でのNULLを表現する未初期化変数
- 例7-10、動的PL/SQLブロックで繰り返されるプレースホルダ名

関連トピック

この章:

- RETURNING INTO句

他の章:

- EXECUTE IMMEDIATE文
- DBMS_SQLパッケージ

13.29 EXIT文

EXIT文は、現行のループの反復を条件付きまたは無条件で終了し、カレント・ループまたはラベルが付けられている外側のループのいずれかの最後に制御を移します。

EXIT文の制限

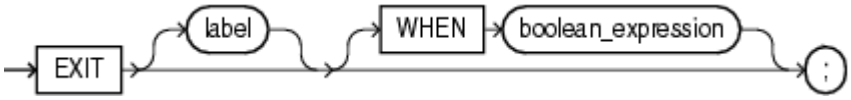
EXIT文はLOOP文内にある必要があります。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

exit_statement ::=



(boolean_expression ::=)

セマンティクス

exit_statement

label

現在のループまたは囲みループを識別する名前です。

labelを指定しない場合、EXIT文はカレント・ループの最後に制御を移します。labelを指定すると、EXIT文は、labelで識別されるループの最後に制御を移します。

WHEN boolean_expression

この句を指定しない場合、EXIT文は現行のループの反復を無条件で終了します。この句を指定すると、boolean_expressionの値がTRUEの場合にのみEXIT文は現行のループの反復を終了します。

例

- 例4-9、EXIT文が含まれている基本LOOP文
- 例4-10、EXIT WHEN文が含まれている基本LOOP文
- 例4-11、EXIT WHEN文が含まれているネストされたラベル付き基本LOOP文
- 例4-25、FOR LOOP文内のEXIT WHEN文
- 例4-26、内側のFOR LOOP文内のEXIT WHEN文

関連トピック

この章:

- 「基本LOOP文」
- CONTINUE文
- 「EXIT文」
- EXIT WHEN文

13.30 明示カーソルの宣言および定義

明示カーソルは、特定の問合せまたはDML文(通常は、複数の行を戻すか複数の行に影響を与える文)の処理に関する情報を格納しておくSQLのプライベート領域を指す名前付きのポインタです。

明示カーソルを使用すると、結果セットの行を一度に1行ずつ取り出すことができます。

明示カーソルは、使用する前に宣言および定義する必要があります。最初に宣言して(cursor_declarationを使用)、後で同じブロック、サブプログラムまたはパッケージ内で定義する(cursor_definitionを使用)か、または宣言と定義を同時に行う(cursor_definitionを使用)ことができます。

明示カーソルの宣言および定義は、それぞれカーソル仕様部およびカーソル本体とも呼ばれます。

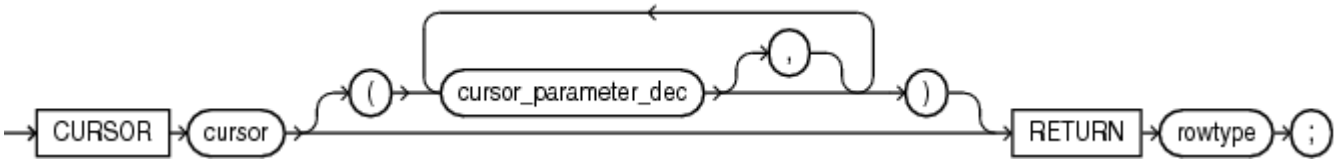
> ノート:
>
> パッケージ仕様部で宣言された明示カーソルは、パッケージの AUTHID 句の影響を受けます。詳細は、「CREATE PACKAGE 文」を参照してください。

ここでのトピック

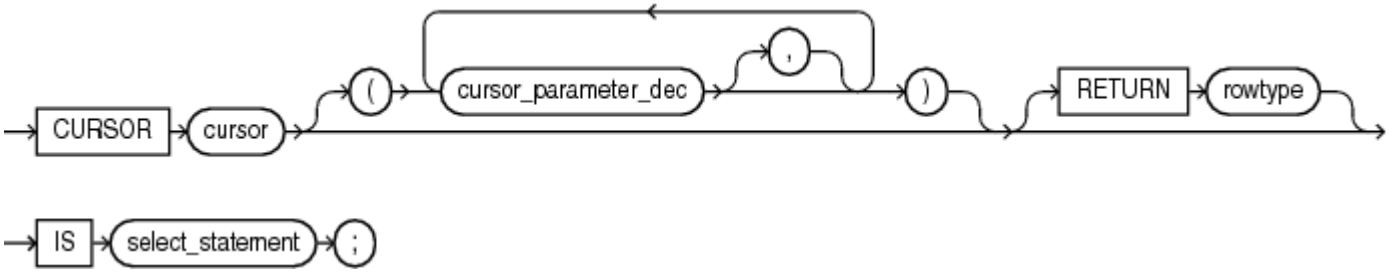
- 構文
- セマンティクス
- 例
- 関連トピック

構文

cursor_declaration ::=

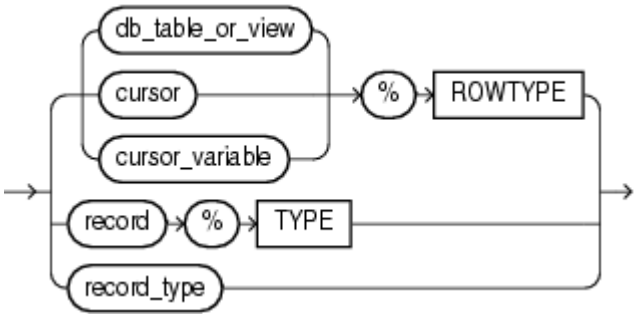


cursor_definition ::=



cursor_parameter_dec ::=

rowtype ::=



セマンティクス

cursor_declaration

cursor

ここで宣言し、後で同じブロック、サブプログラムまたはパッケージ内で定義する明示カーソルの名前。cursorは、予約語のSQLを除く任意の識別子にできます。カーソルにはデータベース表と同じ名前を付けないことをお薦めします。

明示カーソル名は、変数と同じ有効範囲規則に従います(「識別子の有効範囲と可視性」を参照)。

cursor_definition

すでに宣言済の明示カーソルを定義するか、または明示カーソルの宣言と定義の両方を行います。

cursor

ここで定義を行う事前宣言済の明示カーソルの名前、または宣言と定義の両方を行う明示カーソルの名前(cursorは、予約語のSQLを除く任意の識別子にできます)。カーソルにはデータベース表と同じ名前を付けないことをお薦めします。

select_statement

SQLのSELECT文(PL/SQLのSELECT INTO文ではない)。カーソルに仮パラメータが含まれている場合は、各パラメータをselect_statementで使用する必要があります。select_statementは、有効範囲内の他のPL/SQL変数を参照することもできます。

select_statementの制限

このselect_statementにWITH句を持たせることはできません。

参照:

SELECT 文の構文は、『Oracle Database SQL 言語リファレンス』を参照してください。

cursor_parameter_dec

カーソル・パラメータの宣言。

parameter

宣言する仮カーソル・パラメータの名前。select_statementでは、定数を使用できる場所であればどこでもこの名前を使用できます。

IN

INを指定するかどうかに関係なく、仮カーソル・パラメータはINサブプログラム・パラメータの特性(詳細は表8-1を参照)を持ちます。カーソルをオープンすると、仮パラメータの値は実パラメータの値またはデフォルト値のいずれかになります。

datatype

パラメータのデータ型。

datatypeの制限

このdatatypeには制約(NOT NULL、数値の精度と位取り、文字列の長さなど)を持たせることはできません。

expression

仮カーソル・パラメータのデフォルト値を指定します。expressionと仮カーソル・パラメータのデータ型には、互換性がある必要があります。

仮カーソル・パラメータに対応する実パラメータをOPEN文で指定しない場合は、expressionが評価され、その値が仮カーソル・パラメータに代入されます。

仮カーソル・パラメータに対応する実パラメータをOPEN文で指定すると、実パラメータの値が仮カーソル・パラメータに代入され、expressionは評価されません。

rowtype

カーソルが戻す行のデータ型。この行の列は、select_statementが戻す行の列と一致している必要があります。

db_table_or_view

宣言が処理されるときにアクセスできる必要があるデータベースの表またはビューの名前。

cursor

事前に宣言されている明示カーソルの名前。

cursor_variable

事前に宣言されているカーソル変数の名前。

record

事前に宣言されているレコード変数の名前。

record_type

データ型指定子RECORDを使用して定義されたユーザー定義型の名前。

例

- 例6-5、明示カーソルの宣言および定義
- 例6-8、明示カーソル問合せの変数(結果セットの変更なし)
- 例6-9、明示カーソル問合せの変数(結果セットの変更あり)
- 例6-10、仮想列に別名が必要な明示カーソル
- 例6-11、パラメータを受け入れる明示カーソル
- 例6-12、デフォルト値を持つカーソル・パラメータ
- 例6-13、既存のカーソルへの仮パラメータの追加
- 例6-22、親問合せのFROM句内の副問合せ
- 例6-23、相関副問合せ
- 例6-35、CURSOR式
- 例6-41、COMMIT文の後でのFOR UPDATEカーソルによるフェッチ

関連トピック

この章:

- CLOSE文
- 「カーソルFOR LOOP文」
- 「カーソル変数の宣言」
- 「FETCH文」
- 名前付きカーソル属性
- OPEN文
- %ROWTYPE属性
- %TYPE属性

他の章:

- 「明示カーソル」
- 「 問合せ結果セットの処理」
- 「SELECT FOR UPDATEカーソルとFOR UPDATEカーソル」

13.31 式

式は、オペランド(変数、定数、リテラル、演算子、ファンクション起動およびプレースホルダ)の任意の組合せです。

最も単純な式は、1個の変数で構成されています。

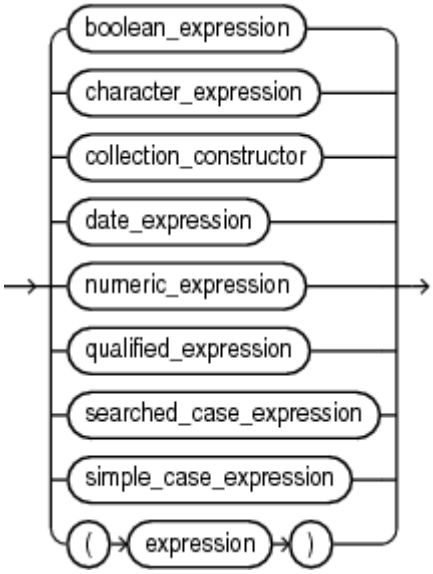
PL/SQLコンパイラは、式を構成するオペランドおよび演算子の型から、式のデータ型を決定します。式が評価されたときは、その型の1つの値が結果として得られます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

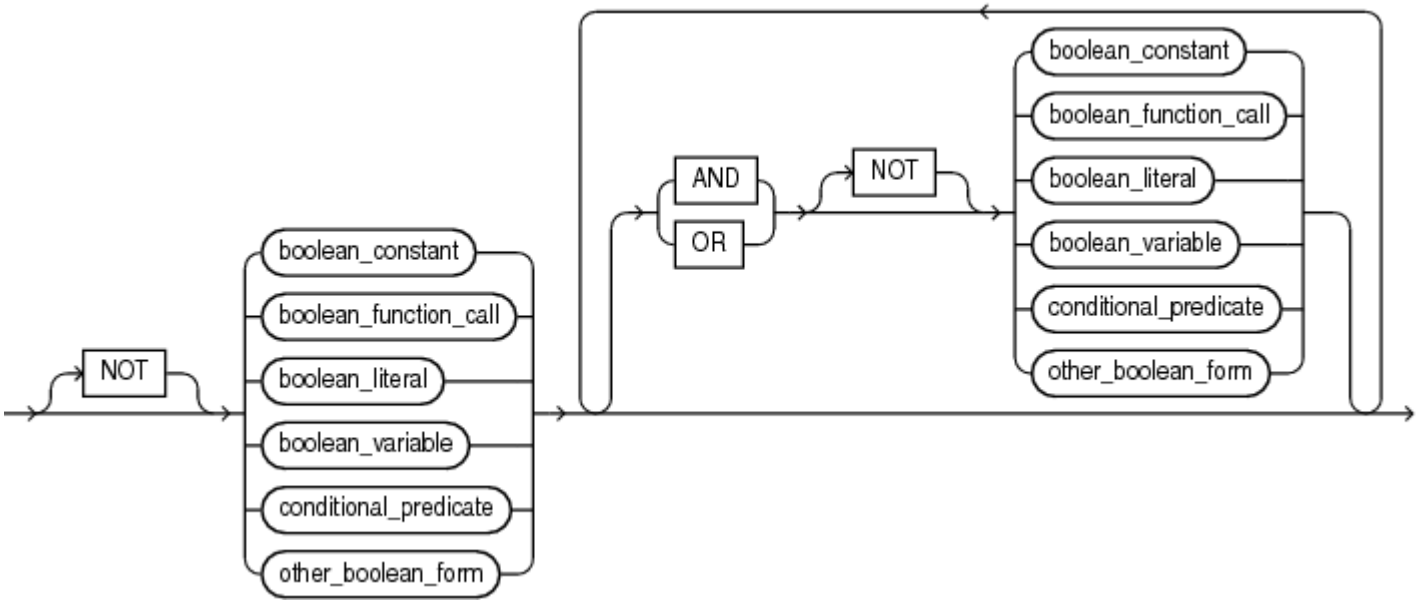
構文

expression ::=



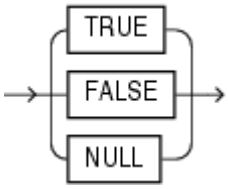
(boolean_expression ::=, character_expression ::=, collection_constructor ::=, date_expression ::=, numeric_expression ::=, qualified_expression ::= , searched_case_expression ::=, simple_case_expression ::=)

boolean_expression ::=

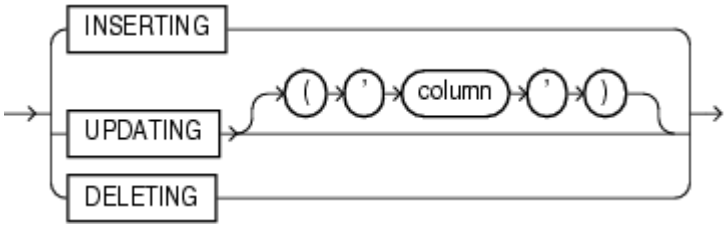


(function_call ::=)

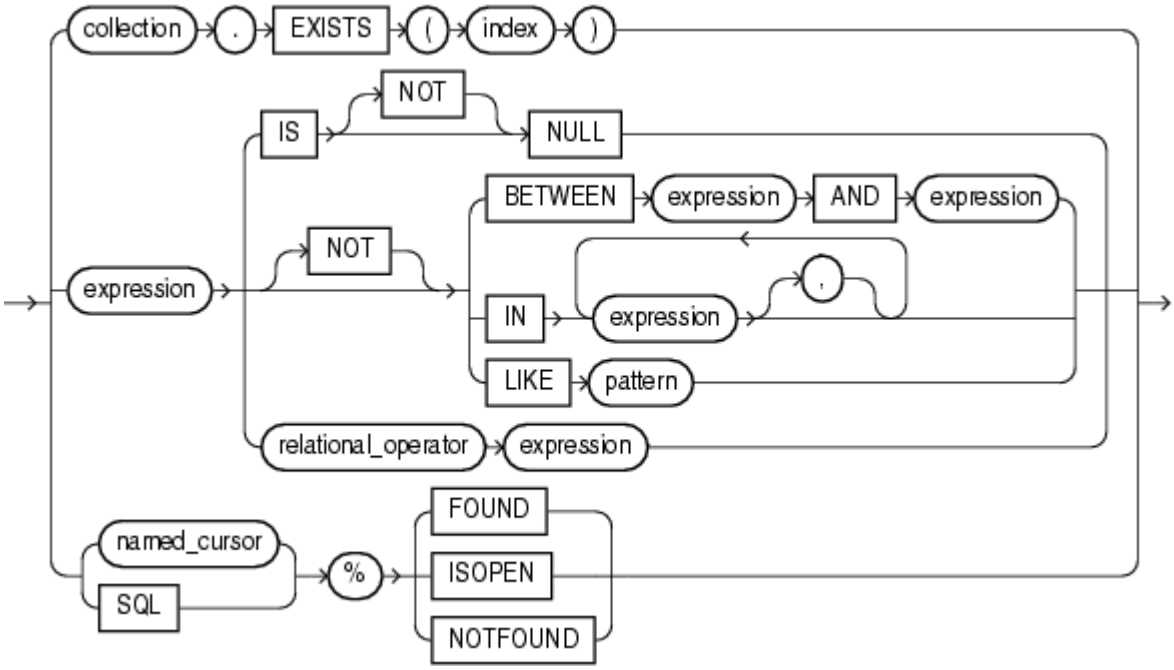
boolean_literal ::=



conditional_predicate ::=

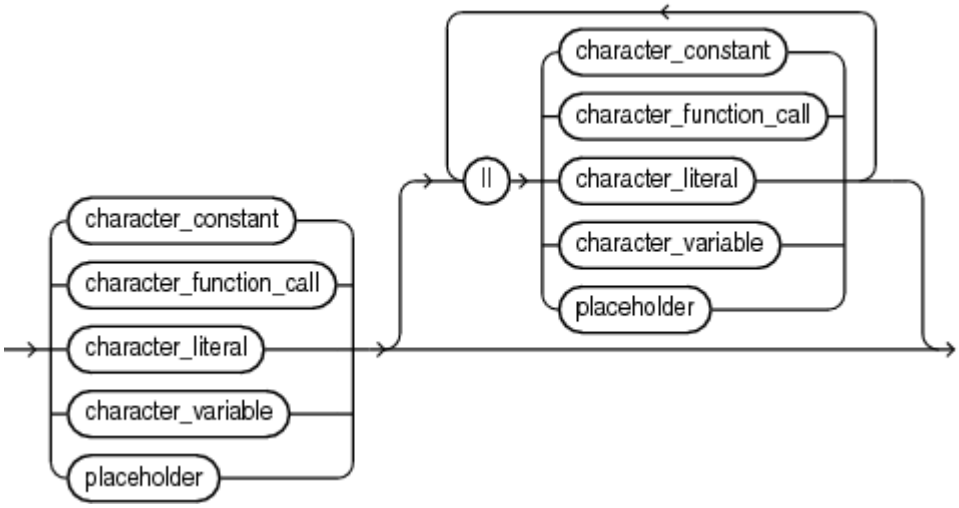


other_boolean_form ::=



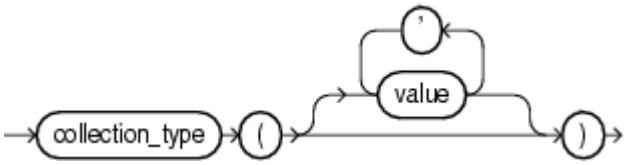
(expression ::=、named_cursor ::=)

character_expression ::=

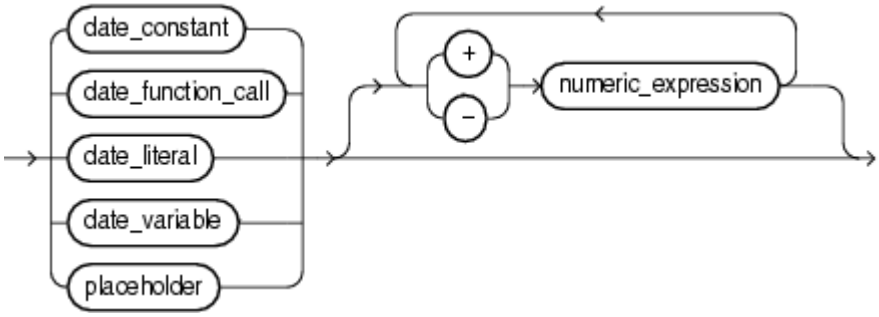


(function_call ::=、placeholder ::=)

collection_constructor ::=

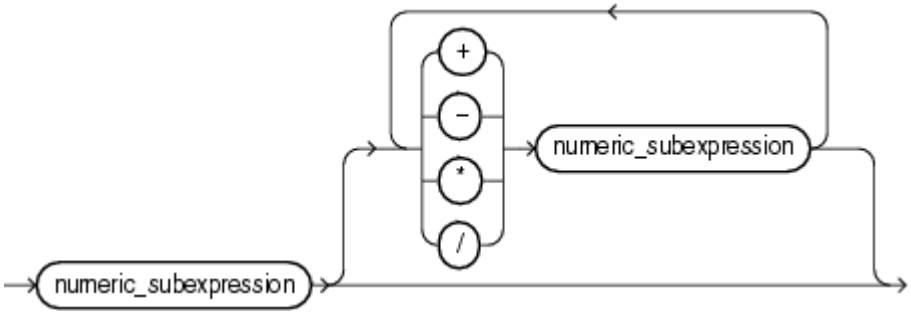


date_expression ::=

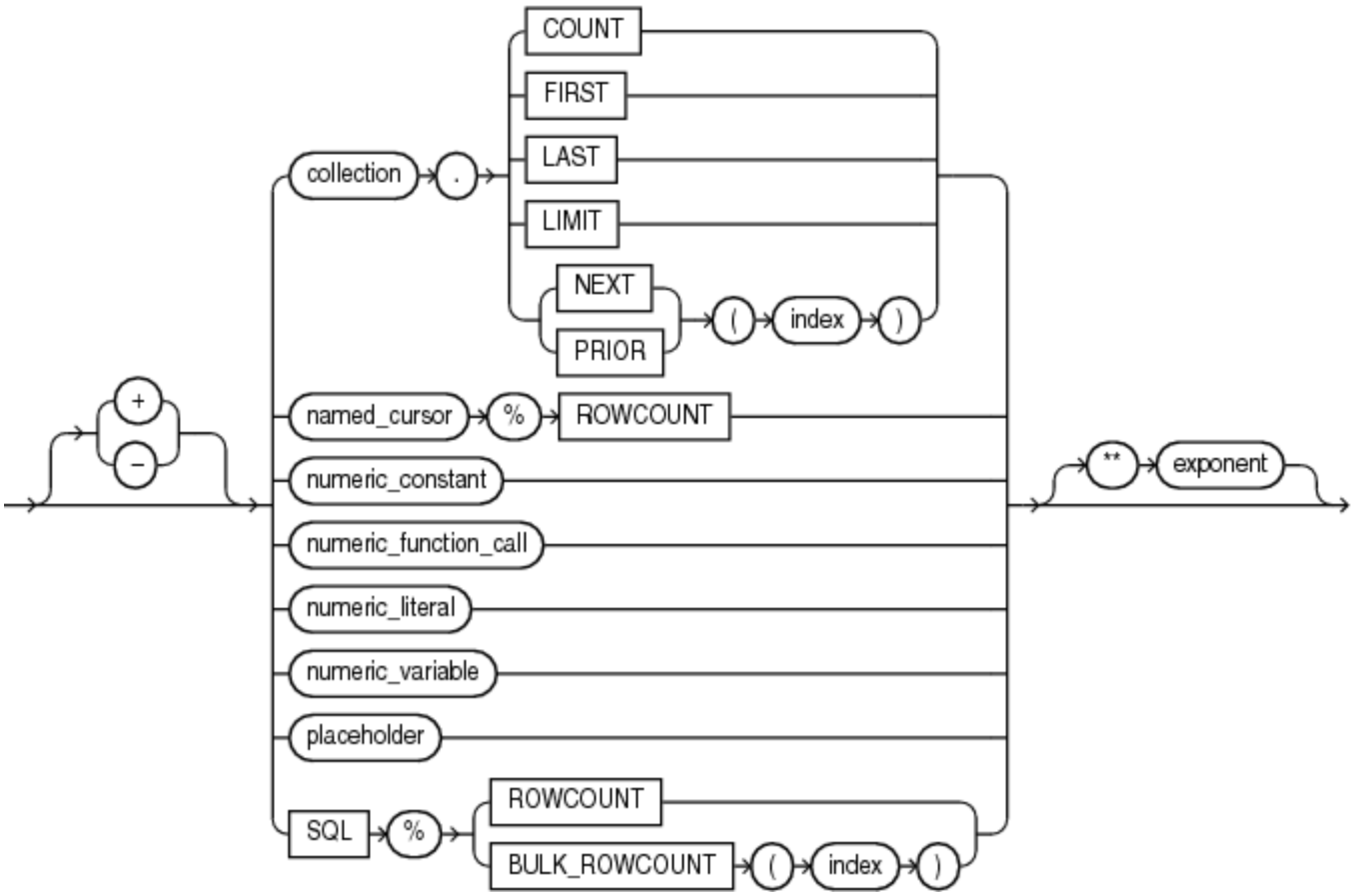


(function_call ::=, placeholder ::=)

numeric_expression ::=

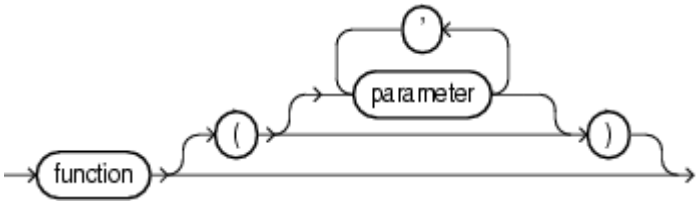


numeric_subexpression ::=

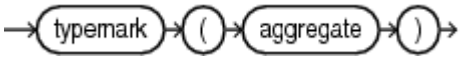


(function_call ::=、named_cursor ::=、placeholder ::=)

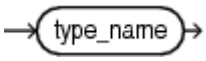
function_call ::=



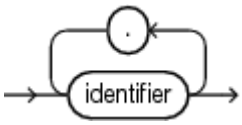
qualified_expression ::=



typemark ::=

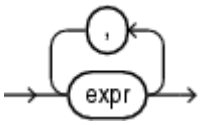


type_name ::=

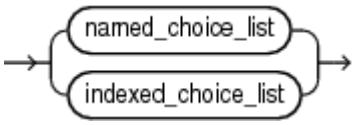


aggregate ::=

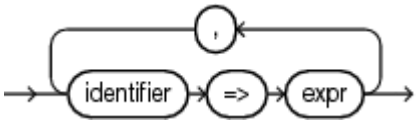
positional_choice_list ::=



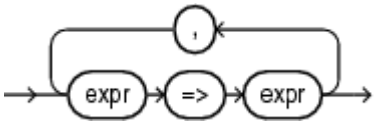
explicit_choice_list ::=



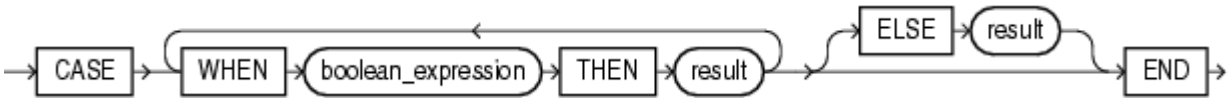
named_choice_list ::=



indexed_choice_list ::=

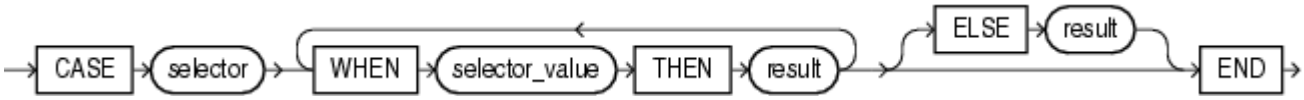


searched_case_expression ::=



(boolean_expression ::=)

simple_case_expression ::=



セマンティクス

boolean_expression

値がTRUE、FALSEまたはNULLの式です。詳細は、「ブール式」を参照してください。

boolean_expressionの制限

SQLにはBOOLEANに相当するデータ型がないため、次の操作は実行できません。

- データベース表の列にBOOLEAN値を割り当てる操作
- データベース表の列の値を選択またはフェッチしてBOOLEAN変数に入れる操作
- SQLファンクションでBOOLEAN値を使用する操作

  (ただし、例3-3に示すように、SQL問合せはBOOLEANパラメータを持つPL/SQLファンクションを起動できます。)
- SQL文(SQL問合せで起動されるPL/SQLファンクションの引数を除く)またはPL/SQL無名ブロックでのBOOLEAN式の使用

ノート:

静的 SQL 問合せで起動される PL/SQL ファンクションの引数は、BOOLEAN リテラルにできません。これを回避するには、例 3-3 に示すように、リテラルを変数に割り当ててからその変数をファンクションに渡します。

NOT、AND、OR

論理演算子を参照してください。

boolean_constant

BOOLEAN型の定数の名前。

boolean_function_call

BOOLEAN値を戻す、事前に定義されたファンクションの起動。セマンティクスの詳細は、「function_call」を参照してください。

boolean_variable

BOOLEAN型の変数の名前。

conditional_predicate

「トリガーを起動したDML文を検出する条件述語」を参照してください。

other_boolean_form

collection

コレクション変数の名前。

EXISTS

collectionのindex番目の要素が存在する場合はTRUE、それ以外の場合はFALSEを戻すコレクション・メソッド(ファンクション)。詳細は、「EXISTSコレクション・メソッド」を参照してください。

EXISTSの制限

collectionが連想配列の場合は、EXISTSは使用できません。

index

データ型PLS_INTEGERまたはPLS_INTEGERに暗黙的に変換可能なデータ型を持つ数式である必要があります(後者の詳細は「PLS_INTEGERの事前定義のサブタイプ」を参照してください)。

IS [NOT] NULL

「IS [NOT] NULL演算子」を参照してください。

BETWEEN expression AND expression

BETWEEN演算子を参照してください。

IN expression [, expression]...

IN演算子を参照してください。

LIKE pattern

LIKE演算子を参照してください。

relational_operator

関係演算子を参照してください。

SQL

直前に実行されたSELECT文またはDML文に関連付けられた暗黙カーソル。詳細は、「暗黙カーソル」を参照してください。

%FOUND、%ISOPEN、%NOTFOUND

「暗黙カーソル属性」および「名前付きカーソル属性」で説明されているカーソル属性。

character_expression

文字データ型(CHARファミリのデータ型)の値を持つ式(「CHARデータ型ファミリ」を参照)。

character_constant

文字データ型の定数の名前。

character_function_call

文字データ型または暗黙的に文字データ型に変換可能な値を戻す、事前に定義されたファンクションの起動。セマンティクスの詳細は、「function_call」を参照してください。

character_literal

文字データ型のリテラル。

character_variable

文字データ型の変数の名前。

||

連結演算子で、ある文字列オペランドを別の文字列オペランドに追加します。詳細は、「連結演算子」を参照してください。

collection_constructor

指定された値を持つ要素を使用して、指定された型のコレクションを構成します。

詳細は、「コレクションのコンストラクタ」を参照してください。

collection_type

事前に宣言されているネストした表型またはVARRAY型の名前(結合配列型ではありません)。

value

collection_typeのコレクション要素の有効な値。

collection_typeがVARRAY型の場合は最大サイズがあり、値の数はこのサイズを超えることができません。
collection_typeがネストした表型の場合は、最大サイズがありません。

値を指定しないと、構成されるコレクションはNULLではなく空になります(emptyとnullの違いは、「コレクション型」を参照してください)。

date_expression

日付データ型(DATEファミリのデータ型)の値を持つ式(「DATEデータ型ファミリ」を参照)。

date_constant

日付データ型の定数の名前。

date_function_call

日付データ型の値または暗黙的に日付データ型に変換可能な値を戻す、事前に定義されたファンクションの起動。セマンティクスの詳細は、「function_call」を参照してください。

date_literal

日付データ型の値または暗黙的に日付データ型に変換可能な値を持つリテラル。

date_variable

日付データ型の変数の名前。

+, -

加算と減算の演算子。

numeric_expression

日付数値型(DATEファミリのデータ型)の値を持つ式(「NUMBERデータ型ファミリ」を参照)。

+, -, /, *, **

加算、減算、除算、乗算、指数の演算子。

numeric_subexpression

collection

コレクション変数の名前。

COUNT、FIRST、LAST、LIMIT、NEXT、PRIOR

コレクション・メソッドの詳細は、「コレクション・メソッドの起動」を参照してください。

named_cursor%ROWCOUNT

「名前付きカーソル属性」を参照してください。

numeric_constant

数値データ型の定数の名前。

numeric_function_call

数値データ型の値または暗黙的に数値データ型に変換可能な値を戻す、事前に定義されたファンクションの起動。セマンティクスの詳細は、「function_call」を参照してください。

numeric_literal

数値データ型のリテラル。

numeric_variable

数値データ型の変数の名前。

SQL%ROWCOUNT

「暗黙カーソル属性」で説明されているカーソル属性。

SQL%BULK_ROWCOUNT]

「SQL%BULK_ROWCOUNT」で説明されているカーソル属性。

exponent

数式。

function_call

function

事前に定義されたファンクションの名前。

parameter [, parameter]...

コールされているファンクションの実パラメータのリスト。各実パラメータのデータ型は、対応する仮パラメータのデータ型と互換性がある必要があります。使用できる実パラメータは仮パラメータのモードで決まります。

仮パラメータのモード	実パラメータ
IN	定数、初期化された変数、リテラルまたは式
OUT	データ型が NOT NULL として定義されていない変数
IN OUT	変数(通常は、文字列バッファまたは数値アキュムレータ)

ファンクションによってパラメータのデフォルト値が指定される場合は、パラメータ・リストでそのパラメータを省略できます。ファンク

ションにパラメータがない場合、またはファンクションによってすべてのパラメータにデフォルト値が指定される場合は、パラメータ・リストを省略するか、または空のパラメータ・リストを指定することができます。

関連項目:

実パラメータの位置表記法、名前表記法および混合表記法

qualified_expression

RECORD型の修飾式は、RECORD型の式が許可されている任意のコンテキストで使用できます。

連想配列型の修飾式は、連想配列型の式が許可されている任意のコンテキストで使用できます。

typemark (aggregate)

集計(修飾項目)の型を明示的に指定します。

typemark

type_name

修飾式は明示的な型の指定を使用することで修飾項目の型を指定します。この明示的な指定はtypemarkと呼ばれるものです。

identifier [.]

修飾項目の型を示します。

aggregate

修飾式は式要素を組み合せて、RECORD型または連想配列型の値を作成します。

positional_choice_list

expr [,]

RECORD型の修飾式では位置関連付けを使用できます。

位置関連付けが同じ構造体内の名前関連付けに従っていない場合もあります(その逆も同様です)。

explicit_choice_list

named_choice_list | indexed_choice_list

名前関連付けでは修飾する構造型からのフィールドの名前を使用する必要があります。索引キーの値は修飾するベクトル型の索引の型と互換している必要があります。

named_choice_list

名前指定選択は構造型にのみ適用されます。

identifier => expr [,]

RECORD型の修飾式に名前関連付けを使用できます。

indexed_choice_list

索引選択は、ベクトル型にのみ適用されます。

expr => expr [,]

索引指定選択(キーと値のペア)を連想配列型の修飾式で使用できます。キーおよび値に式を使用することができます。

連想配列型の構造体で、NULLを索引キー値として使用することはできません。

searched_case_expression

WHEN boolean_expression THEN result

boolean_expressionは順番に評価されます。boolean_expressionの値がTRUEである場合は、boolean_expressionに関連付けられているresultが戻されます。後続のboolean_expressionは評価されません。

ELSE result

resultは、値がTRUEのboolean_expressionが存在しない場合にのみ戻されます。

ELSE句を省略すると、検索CASE式ではNULLが戻されます。

関連項目:

検索CASE文

simple_case_expression

selector

BLOB、BFILEまたはユーザー定義型を除くPL/SQL型の式。selectorは1回評価されます。

WHEN selector_value THEN result

selector_valueは順番に評価されます。selector_valueがselectorの値である場合は、selector_valueに関連付けられているresultが戻されます。後続のselector_valueは評価されません。

selector_valueは、BLOB、BFILE、ADT、PL/SQLレコード、結合配列、VARRAYまたはネストした表を除く任意のPL/SQL型にできます。

ELSE result

resultは、selectorと同じ値を持つselector_valueが存在しない場合にのみ戻されます。

ELSE句を省略すると、単純なcase式ではNULLが戻されます。

ノート:

すべての result (ELSE 句内の result を含む)に対してリテラル NULL を指定すると、エラーPLS-00617 が発生します。

関連項目:

単純なCASE文

例

- 例2-28、連結演算子の例
- 例2-30、カッコを使用した評価順序の制御
- 例2-31、ネストしたカッコを使用した式
- 例2-32、カッコを使用した読みやすさの向上
- 例2-33、演算子の優先順位
- 例2-43、式での関係演算子
- 例2-44、式でのLIKE演算子
- 例2-46、式でのBETWEEN演算子
- 例2-47、式でのIN演算子
- 例2-50、単純なCASE式
- 例2-52、検索CASE式
- 例9-1、トリガーによる条件述語を使用したトリガーを起動する文の検出

関連トピック

この章:

- コレクション・メソッドの起動
- 定数の宣言
- スカラー変数の宣言

他の章:

- 詳細および例は、修飾式の概要を参照してください。
- リテラル
- 式
- 演算子の優先順位
- PL/SQLデータ型
- サブプログラムのパラメータ

13.32 FETCH文

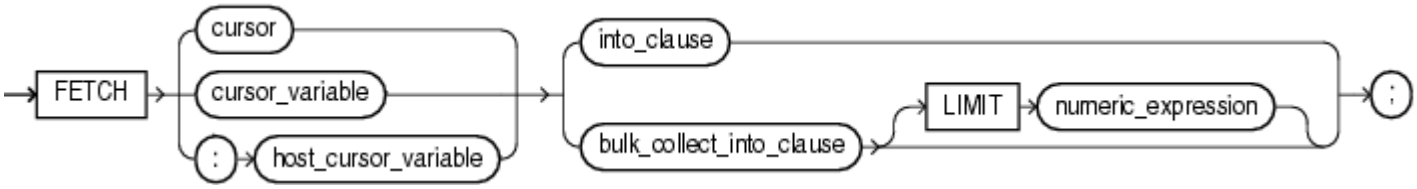
FETCH文は、複数行の問合せの結果セットからデータの行を一度に1行、複数行またはすべての行を取り出し、そのデータを変数、レコードまたはコレクションに格納します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

fetch_statement ::=



(bulk_collect_into_clause ::=、into_clause ::=、numeric_expression ::=)

セマンティクス

fetch_statement

cursor

オープンされている明示カーソルの名前。明示カーソルをオープンするには、「OPEN文」を使用します。

オープンする前またはクローズした後に明示カーソルからフェッチを実行すると、PL/SQLによって事前定義の例外INVALID_CURSORが呼び出されます。

cursor_variable

オープンされているカーソル変数の名前。カーソル変数をオープンするには、「OPEN FOR文」を使用します。カーソル変数は仮サブプログラム・パラメータにすることができます(「サブプログラム・パラメータとしてのカーソル変数」を参照)。

オープンする前またはクローズした後にカーソル変数からフェッチを実行すると、PL/SQLによって事前定義の例外INVALID_CURSORが呼び出されます。

:host_cursor_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されてからオープンされるカーソル変数の名前。ホスト・カーソル変数をオープンするには、「OPEN FOR文」を使用します。コロン(:)とhost_cursor_variableの間に空白は入れないでください。

ホスト・カーソル変数のデータ型は、PL/SQLカーソル変数の戻り型と互換性があります。

into_clause

FETCH文で一度に1行ずつ取り出すには、この句を使用して、カーソルが戻す行の列値を格納する変数またはレコードを指定します。into_clauseの詳細は、into_clause ::=を参照してください。

bulk_collect_into_clause [LIMIT numeric_expression]

FETCH文が戻す行を格納する1つ以上のコレクションを指定するには、bulk_collect_into_clauseを使用します。bulk_collect_into_clauseの詳細は、bulk_collect_into_clause ::=を参照してください。

FETCH文で一度にすべての行を取り出すには、LIMIT numeric_expressionを省略します。

FETCH文で一度に取り出す行の数を制限するには、LIMIT numeric_expressionを指定します。

bulk_collect_into_clauseの制限

- bulk_collect_into_clauseは、クライアント・プログラムでは使用できません。
- FETCH文で暗黙のデータ型変換が必要な場合、bulk_collect_into_clauseに含めることができるのはcollectionまたはhost_arrayを1つのみです。

例

- 例5-51、ファンクションが戻すレコードへのFETCHによる値の代入
- 例6-6、LOOP文内部のFETCH文
- 例6-7、異なる変数への同じ明示カーソルのフェッチ
- 例6-26、カーソル変数によるデータのフェッチ
- 例6-27、カーソル変数からコレクションへのフェッチ
- 例6-41、COMMIT文の後でのFOR UPDATEカーソルによるフェッチ
- 例7-8、OPEN FOR文、FETCH文およびCLOSE文を使用したシステム固有の動的SQL
- 例12-22、2つのネストした表へのバルク・フェッチ
- 例12-23、レコードのネストした表へのバルク・フェッチ
- 例12-24、LIMITによるバルクFETCHの制限

関連トピック

この章:

- 「代入文」
- CLOSE文
- カーソル変数の宣言
- 「明示カーソルの宣言および定義」
- OPEN文

- OPEN FOR文
- RETURNING INTO句
- %ROWTYPE属性
- 「SELECT INTO文」
- %TYPE属性

他の章:

- 「FETCHを使用した行のレコード変数への代入」
- 「明示カーソルによるデータのフェッチ」
- 「 カーソルFOR LOOP文による問合せ結果セットの処理」
- 「カーソル変数によるデータのフェッチ」
- OPEN FOR文、FETCH文およびCLOSE文
- 「BULK COLLECT句を使用したFETCH文」
- 「パイプライン・テーブル・ファンクションの結果からのフェッチ」

13.33 FOR LOOP文

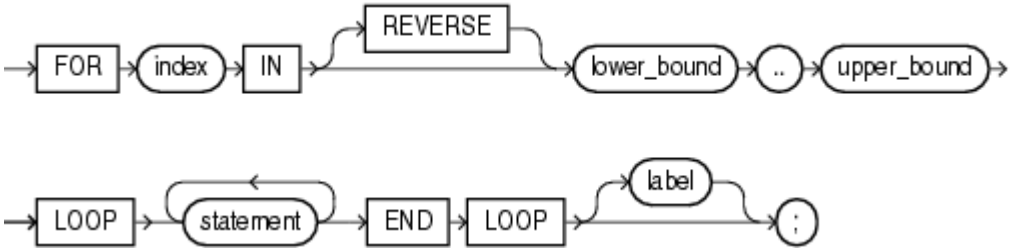
FOR LOOP文が反復されるたびに、文が実行され、索引が増分または減分され、制御がループの先頭に戻ります。指定された値に索引が達した場合、ループの内側の文によって制御がループの外側に移された場合、または例外が呼び出された場合は、FOR LOOP文が終了します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

for_loop_statement ::=



(statement ::=)

セマンティクス

for_loop_statement

index

FOR LOOP文に対してローカルな、暗黙に宣言された整変数の名前。ループの外側の文は、indexを参照できません。ループの内側の文はindexを参照できますが、値を変更することはできません。FOR LOOP文の実行後、indexは未定義となります。

関連項目:

FOR LOOP索引

[REVERSE] lower_bound ..upper_bound

lower_boundおよびupper_boundは、整数に評価されるものにする必要があります(「下限と上限」を参照)。PL/SQLでは、lower_boundとupper_boundはFOR LOOP文に入ったときに一度評価され、一時的なPLS_INTEGER値として格納されます(必要に応じて、最も近い整数に四捨五入されます)。

lower_boundとupper_boundが等しい場合、statementsは一度のみ実行されます。

FOR LOOP文の実行開始時にlower_boundとupper_boundが等しくない場合、次のようになります。

- REVERSEが省略されている場合:

  lower_boundがupper_boundよりも大きい場合、statementsは実行されず、制御はFOR LOOP文の後の文に移ります。

  それ以外の場合は、lower_boundがindexに代入され、statementsが実行され、制御がループの先頭に移り、indexがupper_boundと比較されます。indexがupper_boundよりも小さい場合、indexが1だけ増分され、statementsが再び実行され、制御がループの先頭に戻ります。indexがupper_boundよりも大きい場合、制御がFOR LOOP文の後の文に移ります。

- REVERSEが指定されている場合:

  upper_boundがlower_boundよりも小さい場合、statementsは実行されず、制御はFOR LOOP文の後の文に移ります。

  それ以外の場合は、upper_boundがindexに代入され、statementsが実行され、制御がループの先頭に移り、indexがlower_boundと比較されます。indexがlower_boundよりも大きい場合、indexが1だけ減分され、statementsが再び実行され、制御がループの先頭に戻ります。indexがlower_boundよりも小さい場合、制御がFOR LOOP文の後の文に移ります。

label

for_loop_statementを識別するラベル(「statement ::=」および「label」を参照)。CONTINUE、EXITおよびGOTO文でこのラベルを参照できます。

LOOP文がネストされている場合は特に、ラベルによって読みやすさが向上します。ただし、END LOOP文内のラベルが、同じLOOP文の先頭のラベルと一致している場合のみです(コンパイラではチェックしません)。

例

- 例4-15、FOR LOOP文
- 例4-16、反転FOR LOOP文
- 例4-17、FOR LOOP文でのSTEP句のシミュレート
- 例4-19、外側の文によるFOR LOOP文の索引の参照
- 例4-20、変数と同じ名前を持つFOR LOOP文の索引
- 例4-21、FOR LOOP文による索引と同じ名前を持つ変数の参照
- 例4-22、同じ索引名のネストしたFOR LOOP文
- 例4-23、FOR LOOP文の境界
- 例4-24、実行時に境界が指定されるFOR LOOP文

関連トピック

この章:

- 「基本LOOP文」
- CONTINUE文
- 「カーソルFOR LOOP文」

- 「EXIT文」
- 「FETCH文」
- FORALL文
- OPEN文
- 「WHILE LOOP文」

他の章:

- 「FOR LOOP文」

13.34 FORALL文

FORALL文は、VALUES句およびWHERE句の異なる値を使用して、1つのDML文を複数回実行します。

これらの異なる値は、データが移入された既存のコレクションまたはホスト配列から取得されます。通常、FORALL文は、同等のFOR LOOP文よりはるかに高速です。

ノート:

FORALL 文を使用できるのは、クライアント・プログラム内ではなく、サーバー・プログラム内のみです。

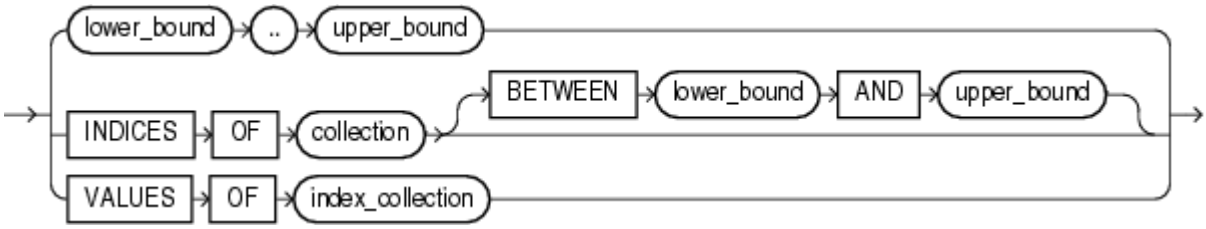
ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

forall_statement ::=

bounds_clause ::=



セマンティクス

forall_statement

index

FORALL文に対してローカルな、暗黙に宣言された整変数の名前。FORALL文の外側の文はindexを参照できません。FORALL文の内側の文はindexを索引変数として参照できますが、式の中で使用したり、その値を変更することはできません。FORALL文の実行後、indexは未定義になります。

dml_statement

VALUES句またはWHERE句内の1つ以上のコレクションを参照する、静的または動的なINSERT文、UPDATE文、DELETE文またはMERGE文。パフォーマンス上のメリットは、indexを索引として使用するコレクション参照に対してのみ適用されます。

dml_statementが参照するすべてのコレクションには、indexの値と一致する索引が必要です。DELETEメソッド、EXTENDメソッドまたはTRIMメソッドをあるコレクションに適用する場合、他のコレクションにも適用すると、すべてのコレクションに同じ索引のセットが含まれます。参照されている要素がないコレクションがある場合は、PL/SQLによって例外が呼び出されます。

dml_statementの制限

dml_statementが動的SQL文の場合、USING句の値(動的SQL文のバインド変数)は、式ではなくコレクションへの単純参照である必要があります。たとえば、collection(i)は有効ですが、UPPER(collection(i))は有効ではありません。

SAVE EXCEPTIONS

一部のDML文が失敗してもFORALL文を継続させることができます。詳細は、「FORALL文が完了した後のFORALL例外の処理」を参照してください。

bounds_clause

変数indexの値を提供する、コレクション要素の索引を指定します。それぞれの値に対して、SQLエンジンによってdml_statementが1回実行されます。

lower_bound .. upper_bound

lower_boundおよびupper_boundはいずれも、FORALL文が入力されるとPL/SQLによって1回評価され、必要に応じて最も近い整数に四捨五入される数式です。結果の整数は、連続した索引番号の有効範囲の下限と上限である必要があります。この範囲内の要素が存在しなかったり、削除されていた場合は、PL/SQLによって例外が呼び出されます。

INDICES OF collection [BETWEEN lower_bound AND upper_bound]

indexの値が、collectionの要素の索引に対応するように指定します。索引は連続していなくてもかまいません。

lower_boundおよびupper_boundはいずれも、FORALL文が入力されるとPL/SQLによって1回評価され、必要に応じて最も近い整数に四捨五入される数式です。結果の整数は索引番号の有効範囲の下限と上限ですが、連続していなくてもかまいません。

collectionの制限

collectionが連想配列の場合、PLS_INTEGERで索引付けする必要があります。

VALUES OF index_collection

indexの値が、PLS_INTEGERで索引付けされたPLS_INTEGER要素のコレクションindex_collectionの要素であると指定します。index_collectionの索引は連続していなくてもかまいません。index_collectionが空の場合、PL/SQLによって例外が呼び出され、FORALL文は実行されません。

例

- 例12-8、FORALL文のDELETE文
- 例12-9、FOR LOOP文およびFORALL文におけるINSERT文の時間の違い
- 例12-10、コレクションのサブセットに対するFORALL文

- 例12-11、疎コレクションおよびそのサブセットに対するFORALL文
- 例12-12、FORALL例外の即時処理
- 例12-13、FORALL文が完了した後のFORALL例外の処理
- 例12-26、FORALL文でのRETURN BULK COLLECT INTO付きのDELETE
- 例12-28、無名ブロックによる入力ホスト配列のバルク・バインド

関連トピック

この章:

- 「FOR LOOP文」
- 「暗黙カーソル属性」

他の章:

- FORALL文
- BULK COLLECT句
- 「FORALL文とBULK COLLECT句の併用」

13.35 仮パラメータの宣言

仮パラメータの宣言に、パラメータの名前とデータ型を指定し、オプションでそのモードとデフォルト値を指定します。

仮パラメータ宣言は、次の場所で使用できます。

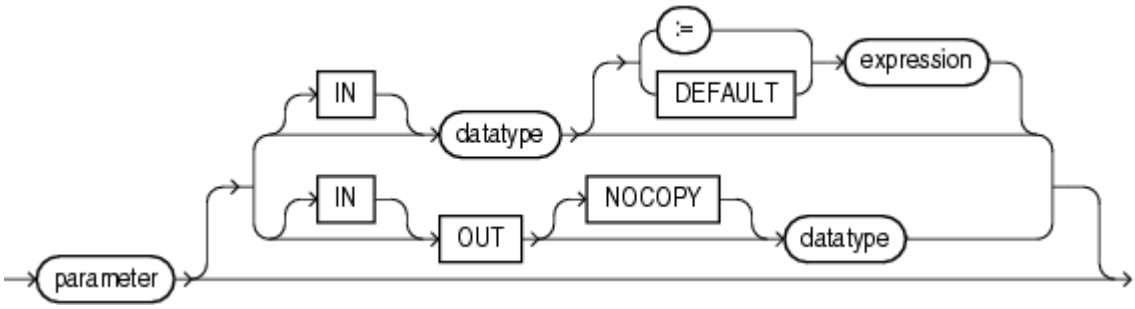
- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」
- CREATE FUNCTION文
- CREATE PROCEDURE文

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

parameter_declaration ::=



セマンティクス

parameter_declaration

parameter

宣言する仮パラメータの名前。この名前は、サブプログラムの実行部で参照できます。

IN、OUT、IN OUT

パラメータの動作を決定するモード(「サブプログラムのパラメータ・モード」を参照)。デフォルト: IN。

> ノート:
>
> ファンクション・パラメータには、OUT および IN OUT を使用しないでください。ファンクションの目的は、0(ゼロ)個以上のパラメータを取り、単一の値を戻すことです。サブプログラム専用ではない変数の値を変更するという副作用も避ける必要があります。

NOCOPY

対応する実パラメータを値によってではなく参照によって渡すように、コンパイラに指示します(この違いは、「サブプログラムのパラメータの引渡し方法」を参照してください)。サブプログラムが起動されるたびに、オプティマイザでは、NOCOPYに従うか無視するかが暗黙的に決定されます。

注意:

NOCOPY によって、エイリアシングの可能性が高くなります。詳細は、「参照渡しパラメータによるサブプログラムのパラメータのエイリアシング」を参照してください。

次の場合、コンパイラはNOCOPYを無視します。

- 実パラメータを仮パラメータのデータ型に暗黙的に変換する必要がある場合。
- 実パラメータがコレクションの要素である場合。
- 実パラメータがNOT NULL制約付きのスカラー変数である場合。
- 実パラメータが範囲、サイズ、位取りまたは精度の制約を伴うスカラー数値変数である場合。
- 実パラメータと仮パラメータがレコードであり、いずれかまたは両方が%ROWTYPEまたは%TYPEを使用して宣言されており、対応するフィールドの制約が異なる場合。
- 実パラメータと仮パラメータがレコードであり、実パラメータはカーソルFOR LOOP文の索引として(暗黙的に)宣言されており、対応するフィールドの制約が異なる場合。
- サブプログラムが、データベース・リンクによって、または外部サブプログラムとして起動される場合。

ノート:

前述のリストは、今後のリリースで変更される場合があります。

datatype

宣言する仮パラメータのデータ型。データ型は制約付きサブタイプにできますが、制約(NUMBER(2)またはVARCHAR2(20)など)を含めることはできません。

datatypeが制約付きサブタイプである場合、対応する実パラメータにサブタイプのNOT NULL制約は継承されますが(制約が付いている場合)、サイズは継承されません(例8-10を参照)。

注意:

データ型 REF CURSOR を使用すると、サブプログラム・パラメータのエイリアシングの可能性が高くなり、意図しない結果が発生することがあります。詳細は、「カーソル変数パラメータによるサブプログラムのパラメータのエイリアシング」を参照してください。

expression

宣言する仮パラメータのデフォルト値。expressionのデータ型は、datatypeと互換性がある必要があります。

仮パラメータに対応する実パラメータをサブプログラムの起動に指定しない場合は、起動によってexpressionが評価され、その値が仮パラメータに代入されます。

仮パラメータに対応する実パラメータをサブプログラムの起動に指定すると、起動によって実パラメータの値が仮パラメータに代入され、expressionは評価されません。

例

- 例2-26、IN OUTサブプログラム・パラメータとしての変数への値の代入
- 例8-9、仮パラメータと実パラメータ
- 例8-14、プロシージャの起動前、起動中および起動後のパラメータ値
- 例8-15、例外処理後のOUTおよびIN OUTパラメータの値
- 例8-20、デフォルトのパラメータ値を使用したプロシージャ
- 例8-21、ファンクションによるデフォルトのパラメータ値の提供
- 例8-22、既存の起動を変更しないサブプログラム・パラメータの追加

関連トピック

この章:

- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」

他の章:

- サブプログラムのパラメータ
- サブプログラムの起動のチューニング
- CREATE FUNCTION文
- CREATE PROCEDURE文

13.36 ファンクションの宣言および定義

ファンクションは、起動する前に定義および宣言する必要があります。最初に宣言して(function_declarationを使用)、後で同じブロック、サブプログラムまたはパッケージ内で定義する(function_declarationを使用)か、または宣言と定義を同時に行う(function_declarationを使用)ことができます。

ファンクションとは、値を戻すサブプログラムのことです。値のデータ型は、ファンクションのデータ型になります。ファンクションの起動(コール)は式の一部で、そのデータ型はファンクションのデータ型です。

ファンクション宣言は、ファンクション仕様部とも呼ばれます。

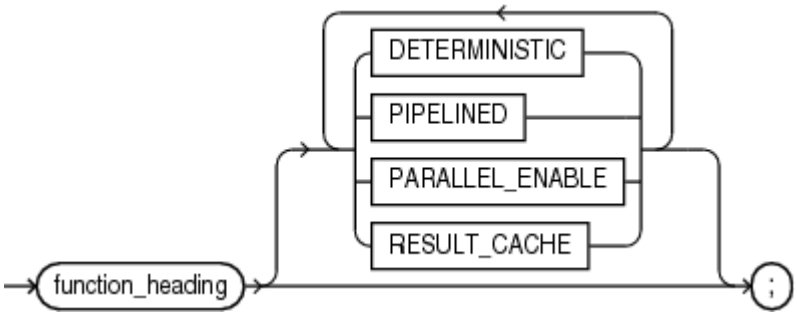
> ノート:
>
> このトピックの内容は、ネストされたファンクションに適用されます。
>
> スタンドアロン・ファンクションの詳細は、「CREATE FUNCTION 文」を参照してください。
>
> パッケージ・ファンクションの詳細は、「CREATE PACKAGE 文」を参照してください。

ここでのトピック

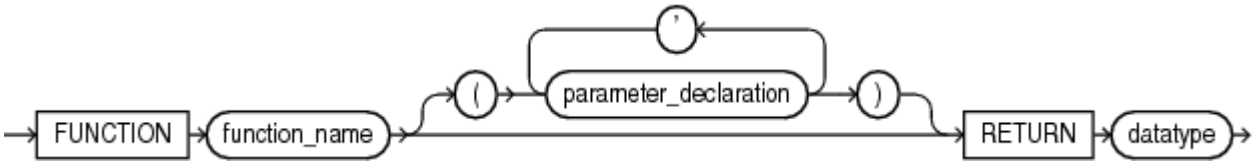
- 構文
- セマンティクス
- 例
- 関連トピック

構文

function_declaration ::=

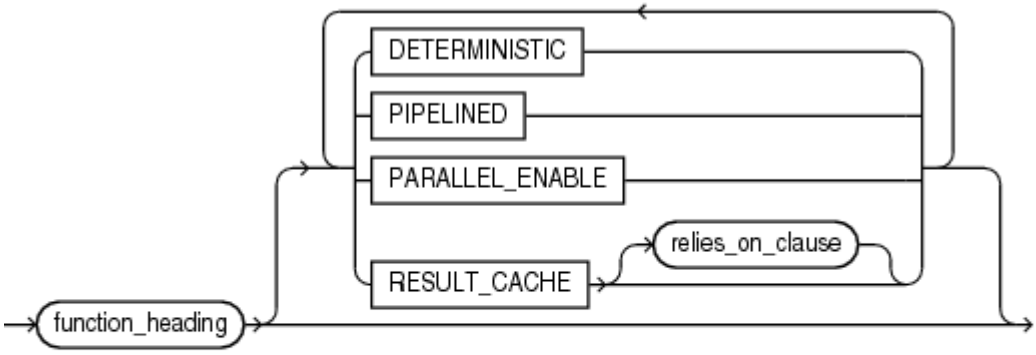


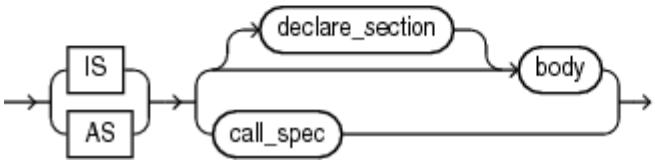
function_heading ::=



(datatype ::= , parameter_declaration ::=)

function_definition ::=





(body ::= , declare_section ::= , pipelined_clause ::= , deterministic_clause ::= , parallel_enable_clause ::= , result_cache_clause ::= , call_spec ::=)

セマンティクス

function_declaration

ファンクションを宣言しますが、定義はしません。定義は、同じブロック、サブプログラムまたはパッケージ内に後で宣言として存在している必要があります。

function_heading

ファンクションのヘッダーは、ファンクション名とそのパラメータ・リストを指定します。

function_name

宣言または定義するファンクションの名前。

RETURN datatype

ファンクションが返す値のデータ型を指定します。任意のPL/SQLデータ型を指定できます(「PL/SQLデータ型」を参照)。

datatypeの制限

このデータ型は、NOT NULLなどでは制約することができません。datatypeが制約付きサブタイプの場合、戻された値はサブタイプの制約を継承しません(詳細は、「制約付きサブタイプの仮パラメータ」を参照)。

function_definition

すでに宣言済のファンクションを定義するか、またはファンクションの宣言と定義の両方を行います。

declare_section

ファンクションに対してローカルであり、bodyで参照でき、ファンクションが実行を完了すると消滅する項目を宣言します。

body

ファンクションの必須の実行部およびオプションの例外処理部。実行部には、RETURN文へ導く1つ以上の実行パスが必要であり、

実行パスがないと、ランタイム・エラーが発生します。

例

- 例8-2、単純なPL/SQLファンクションの宣言、定義および起動

関連トピック

- 仮パラメータの宣言
- プロシージャの宣言および定義
- PL/SQLサブプログラム

13.37 GOTO文

GOTO文は、ラベルの付いたブロックまたは文に制御を移します。

GOTO文によってカーソルFOR LOOP文が途中で終了されると、カーソルはクローズされます。

GOTO文の制限

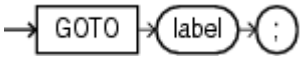
- GOTO文は、IF文、CASE文、LOOP文またはサブブロックには制御を移せません。
- GOTO文では、あるIF文の句から別の句へ制御を移したり、あるCASE文のWHEN句から別の句へ制御を移すことはできません。
- GOTO文では、サブプログラムの外に制御を移せません。
- GOTO文では、例外ハンドラに制御を移せません。
- GOTO文は、例外ハンドラからカレント・ブロックに制御を戻すことはできません(ただし、例外ハンドラから外側のブロックに制御を移すことはできます)。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

goto_statement ::=



セマンティクス

goto_statement

label

ブロックまたは文を識別します(「plsql_block ::=」、「statement ::=」、および「label」を参照)。

labelがカレント・ブロックにない場合、GOTO文は外側のブロックのうちlabelが存在する最初のものに制御を移します。

例

- 例4-29、GOTO文
- 例4-32、GOTO文による外側のブロックへの制御の移動
- 例4-33、GOTO文によるIF文内への無効な制御の移動

関連トピック

この章:

- ブロック

他の章:

- 「GOTO文」

13.38 IF文

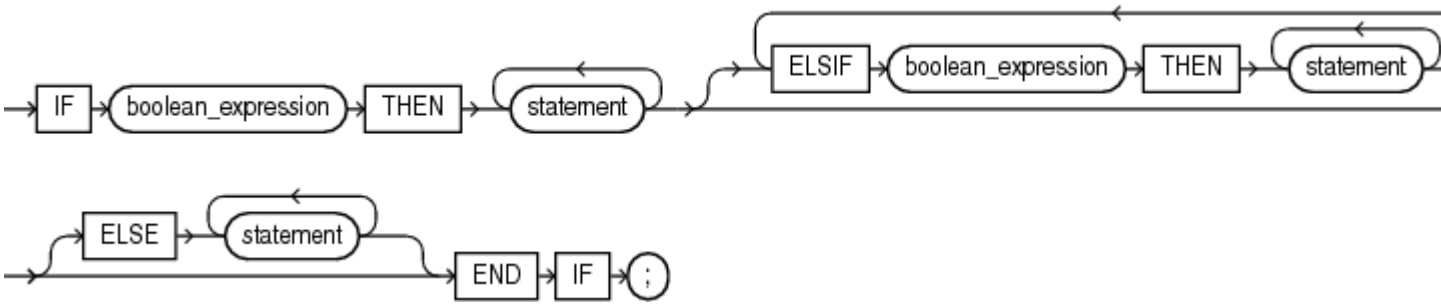
IF文を使用すると、BOOLEAN式の値に応じて、一連の1つ以上の文を実行するか、またはスキップできます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

if_statement ::=



(boolean_expression ::=、statement ::=)

セマンティクス

boolean_expression

値がTRUE、FALSEまたはNULLの式です。

1番目のboolean_expressionは常に評価されます。他の各boolean_expressionは、直前の式の値がFALSEの場合にのみ評価されます。

boolean_expressionが評価され、その値がTRUEだった場合は、対応するTHENの後の文が実行されます。それ以降の式は評価されず、それらに関連付けられた文は実行されません。

ELSE

値がTRUEになるboolean_expressionがない場合は、ELSEの後の文が実行されます。

例

- 例4-1、IF THEN文
- 例4-2、IF THEN ELSE文
- 例4-3、ネストしたIF THEN ELSE文
- 例4-4、IF THEN ELSIF文

関連トピック

この章:

- CASE文
- 「式」

他の章:

- 条件付き選択文

13.39 暗黙カーソル属性

暗黙カーソルには、名前付きカーソルに関連付けられていない、直前に実行されたSELECT文またはDML文の情報を戻す属性があります。

ノート:

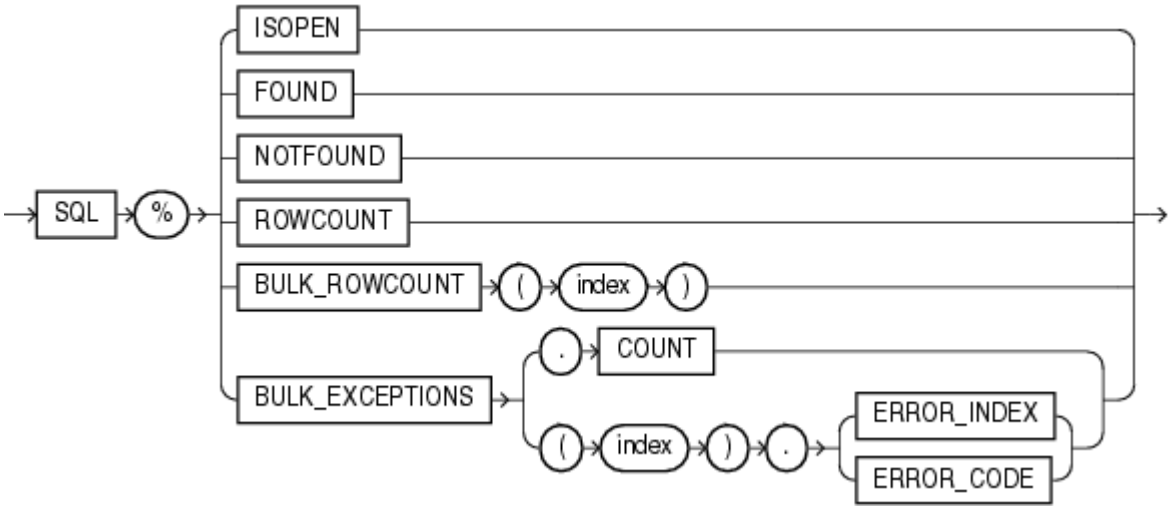
カーソル属性は、プロシージャ文でのみ使用でき、SQL 文では使用できません。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

implicit_cursor_attribute ::=



セマンティクス

%ISOPEN

SQL%ISOPENの値は常にFALSEです。

%FOUND

SQL%FOUNDの値は次のうちの1つです。

- SELECT文もDML文も実行されない場合は、NULL。
- 直前に実行されたSELECT文またはDML文が行を戻した場合は、TRUE。
- 直前に実行されたSELECT文またはDML文が行を戻さなかった場合は、FALSE。

%NOTFOUND

SQL%NOTFOUNDの値は次のうちの1つです。

- SELECT文もDML文も実行されない場合は、NULL。
- 直前に実行されたSELECT文またはDML文が行を戻した場合は、FALSE。
- 直前に実行されたSELECT文またはDML文が行を戻さなかった場合は、TRUE。

%ROWCOUNT

SQL%ROWCOUNTの値は次のうちの1つです。

- SELECT文もDML文も実行されない場合は、NULL。
- SELECT文またはDML文が実行されていれば、これまでにフェッチされた行数。

SQL%BULK_ROWCOUNT

連想配列のような複合属性で、そのi番目の要素には、直前に完了したFORALL文内のi番目のDML文によって影響を受けた行の数が含まれます。詳細は、「FORALLによる影響を受ける行の数」を参照してください。

SQL%BULK_ROWCOUNTの制限

SQL%BULK_ROWCOUNT(index)の値を他のコレクションに代入することはできません。

SQL%BULK_EXCEPTIONS

直前のFORALL文の実行中に失敗したDML文についての情報を示す連想配列のような複合属性です。SQL%BULK_EXCEPTIONS.COUNTは、失敗したDML文の数です。SQL%BULK_EXCEPTIONS.COUNTが0(ゼロ)でない場合、1からSQL%BULK_EXCEPTIONS. COUNTまでの各索引値iに対して、次のようになります。

- SQL%BULK_EXCEPTIONS(i).ERROR_INDEXは、失敗したDML文の数です。
- SQL%BULK_EXCEPTIONS(i).ERROR_CODEは、失敗に関するOracle Databaseエラー・コードです。

通常、この属性は、SAVE EXCEPTIONS句を使用するFORALL文の例外ハンドラで使用されます。詳細は、「FORALL文が完了した後のFORALL例外の処理」を参照してください。

例

- 例6-3、SQL%FOUND暗黙カーソル属性
- 例6-4、SQL%ROWCOUNT暗黙カーソル属性
- 例6-15、%FOUND明示カーソル属性
- 例6-14、%ISOPEN明示カーソル属性
- 例6-16、%NOTFOUND明示カーソル属性
- 例6-17、%ROWCOUNT明示カーソル属性
- 例12-13、FORALL文が完了した後のFORALL例外の処理
- 例12-14、FORALLの各DELETEで影響を受けた行の数の表示
- 例12-15、FORALLの各INSERT SELECTで影響を受けた行の数の表示

関連トピック

この章:

- FORALL文
- 名前付きカーソル属性

他の章:

- 暗黙カーソル
- 「 問合せ結果セットの処理」

13.40 INLINEプラグマ

INLINEプラグマは、サブプログラムの起動がインライン化されるかどうかを指定します。

インライン化によって、サブプログラムの起動は、起動先サブプログラムのコピーに置き換えられます(起動先サブプログラムと起動元サブプログラムが同じプログラム・ユニット内にある場合)。

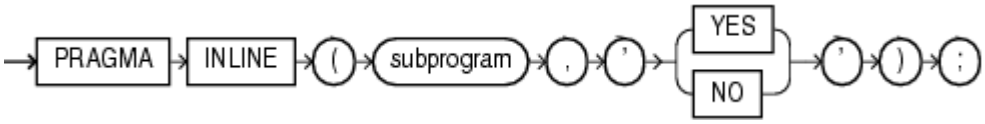
> ノート:
>
> INLINE プラグマは、直後にある宣言または文および一部の種類の文にのみ影響します。詳細は、「サブプログラムのインライン化」を参照してください。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

inline_pragma ::=



セマンティクス

subprogram

サブプログラムの名前。subprogramがオーバーロードされた場合、INLINEプラグマはその名前を持つすべてのサブプログラムに適用されます。

YES

PLSQL_OPTIMIZE_LEVEL=2の場合、'YES' はサブプログラムの起動のインライン化を指定します。

PLSQL_OPTIMIZE_LEVEL=3の場合、'YES' はサブプログラムの起動のインライン化に高い優先順位を指定します。

NO

サブプログラムの起動がインライン化されないように指定します。

例

- 例12-1、サブプログラムのインライン化の指定
- 例12-2、オーバーロードされたサブプログラムのインライン化の指定
- 例12-3、サブプログラムの非インライン化の指定

- 例12-4、PRAGMA INLINE ... 'NO'によるPRAGMA INLINE ... 'YES'のオーバーライド

関連トピック

- サブプログラムのインライン化

13.41 実行者権限および定義者権限句

ストアドPL/SQLサブプログラムのAUTHIDプロパティを指定します。AUTHIDプロパティは、実行時にユニットによって発行されるSQL文の名前解決および権限チェックに影響を与えます。

invoker_rights_clauseは、次のSQL文に指定できます。

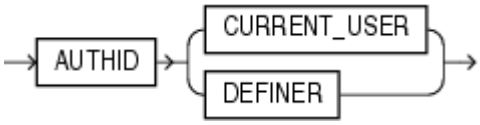
- ALTER TYPE文
- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE PROCEDURE文
- CREATE TYPE文
- CREATE TYPE BODY文

ここでのトピック

- 構文
- セマンティクス
- 関連トピック

構文

invoker_rights_clause ::=



セマンティクス

invoker_rights_clause

パッケージの宣言に記述されている場合、パッケージのファンクションとプロシージャおよびパッケージ仕様部で宣言された明示カーソルのAUTHIDプロパティを指定します。

スタンドアロン・ファンクションの宣言に記述されている場合、ファンクションのAUTHIDプロパティを指定します。

スタンドアロン・プロシージャの宣言に記述されている場合、プロシージャのAUTHIDプロパティを指定します。

invoker_rights_clauseは、サブプログラムの宣言で1回のみ記述できます。

ADTに記述されている場合、ADTのメンバー・ファンクションおよびプロシージャのAUTHIDプロパティを指定します。

invoker_rights_clauseの制限

型には次の制限が適用されます。

- この句は、ADTに対してのみ有効です。ネストした表やVARRAY型には使用できません。
- サブタイプを作成する場合は、意味を明確にするためにこの句を指定できます。ただし、サブタイプはスーパータイプの

AUTHIDプロパティを継承するため、スーパータイプに指定した値と異なる値は指定できません。

- スーパータイプがAUTHID DEFINERで作成された場合は、スーパータイプと同じスキーマでサブタイプを作成する必要があります。

関連トピック

このドキュメント:

- AUTHIDプロパティの詳細は、実行者権限および定義者権限(AUTHIDプロパティ)を参照してください
- サブプログラムのプロパティ

13.42 INSERT文の拡張機能

SQLのINSERT文に対するPL/SQLの拡張機能により、insert_into_clauseで列リストを指定するかわりに、single_table_insertのvalues_clauseでレコード名を指定できます

この形式のINSERT文は、結果的にはレコードを表に挿入しますが、実際には表に行を追加し、その行の各列へ対応するレコードの値を設定します。

関連項目:

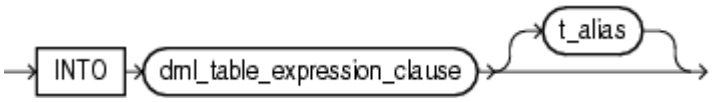
SQLのINSERT文の構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

ここでのトピック

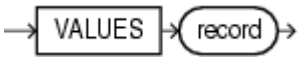
- 構文
- セマンティクス
- 例
- 関連トピック

構文

insert_into_clause ::=



values_clause ::=



セマンティクス

insert_into_clause

dml_table_expression_clause

通常は表名。詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

t_alias

dml_table_expression_clauseの別名。

values_clause

record

RECORD型または%ROWTYPE型のレコード変数の名前。recordは、dml_table_expression_clauseによって説明される項目の行を表している必要があります。つまり、レコードには、行のすべての列に対して、互換性のあるデータ型のフィールドが必要です。列にNOT NULL制約が含まれている場合、対応するフィールドにNULL値を含めることはできません。

関連項目:

INSERT文の構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例

- 例5-54、デフォルト値のレコードの挿入による表の初期化

関連トピック

この章:

- レコード変数の宣言
- %ROWTYPE属性

他の章:

- 「表へのレコードの挿入」
- 「レコードの挿入/更新に関する制限」

13.43 名前付きカーソル属性

すべての名前付きカーソル(明示カーソルまたはカーソル変数)には4つの属性があり、それぞれがDML文の実行に関する情報を戻します。

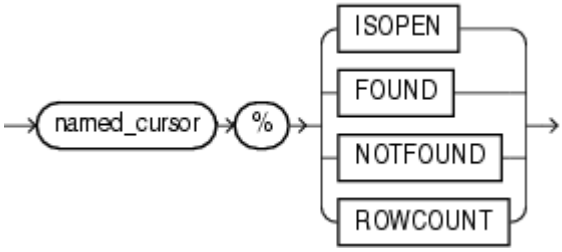
> ノート:
>
> カーソル属性は、プロシージャ文でのみ使用でき、SQL 文では使用できません。

ここでのトピック

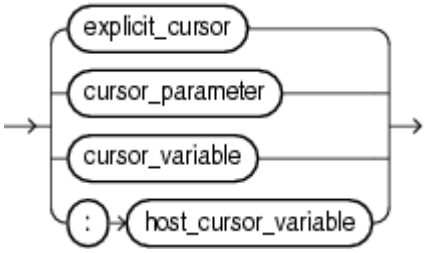
- 構文
- セマンティクス
- 例
- 関連トピック

構文

named_cursor_attribute ::=



named_cursor ::=



セマンティクス

named_cursor_attribute

%ISOPEN

named_cursor%ISOPENの値は、カーソルがオープンされている場合はTRUE、オープンされていない場合はFALSEとなります。

%FOUND

named_cursor%FOUNDは次のいずれかの値になります。

- INVALID_CURSOR(カーソルがオープンされていない場合)

- カーソルはオープンされているが、フェッチが試みられていない場合、NULL。
- 最後のフェッチが行を戻した場合、TRUE。
- 最後のフェッチが行を戻さなかった場合、FALSE。

%NOTFOUND

named_cursor%NOTFOUNDは次のいずれかの値になります。

- カーソルがオープンされていない場合、INVALID_CURSOR。
- カーソルはオープンされているが、フェッチが試みられていない場合、NULL。
- 最後のフェッチが行を戻した場合、FALSE。
- 最後のフェッチが行を戻さなかった場合、TRUE。

%ROWCOUNT

named_cursor%ROWCOUNTは次のいずれかの値になります。

- カーソルがオープンされていない場合、INVALID_CURSOR。
- それまでにフェッチした行の数(カーソルがオープンされている場合)。

named_cursor

explicit_cursor

明示カーソルの名前。

cursor_parameter

仮カーソル・パラメータの名前。

cursor_variable

カーソル変数の名前。

:host_cursor_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されるカーソル変数の名前。コロン(:)とhost_cursor_variableの間に空白は入れないでください。

例

- 例6-14、%ISOPEN明示カーソル属性
- 例6-15、%FOUND明示カーソル属性
- 例6-16、%NOTFOUND明示カーソル属性
- 例6-17、%ROWCOUNT明示カーソル属性

関連トピック

この章:

- カーソル変数の宣言
- 「明示カーソルの宣言および定義」
- 暗黙カーソル属性

他の章:

- 明示カーソル属性

13.44 NULL文

NULL文は、制御を次の文に渡す以外は何もしません。

> ノート:
>
> NULL 文と BOOLEAN 値 NULL は無関係です。

ここでのトピック

- 構文
- 例
- 関連トピック

構文

null_statement ::=

例

- 例4-31、GOTO文によるラベル付きのNULL文への移動
- 例4-34、アクションを実行しないことを明示するNULL文
- 例4-35、サブプログラム作成時のプレースホルダとしてのNULL文

関連トピック

- NULL文

13.45 OPEN文

OPEN文は、明示カーソルをオープンし、関連付けられた問合せを処理するデータベース・リソースを割り当て、結果セットを識別して、カーソルを結果セットの最初の行の前に配置します。

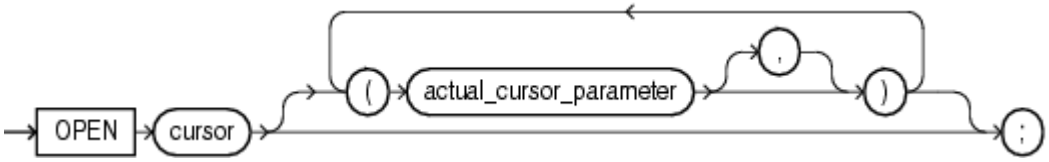
問合せにFOR UPDATE句がある場合、結果セットの行はOPEN文によってロックされます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

open_statement ::=



セマンティクス

cursor

オープンされていない明示カーソルの名前。

actual_cursor_parameter

オープンするカーソルの実パラメータのリスト。実パラメータには定数、リテラル、初期化された変数または式を使用できます。各実パラメータのデータ型は、対応する仮パラメータのデータ型と互換性がある必要があります。

実カーソル・パラメータには、位置表記法および名前表記法のいずれも指定できます。これらの表記法の詳細は、実パラメータの位置表記法、名前表記法および混合表記法を参照してください。

カーソルによってパラメータのデフォルト値が指定される場合は、パラメータ・リストでそのパラメータを省略できます。カーソルにパラメータがない場合、またはプロシージャによってすべてのパラメータにデフォルト値が指定される場合は、パラメータ・リストを省略するか、または空のパラメータ・リストを指定することができます。

例

- 例6-11、パラメータを受け入れる明示カーソル
- 例6-12、デフォルト値を持つカーソル・パラメータ

関連トピック

この章:

- CLOSE文
- 「明示カーソルの宣言および定義」
- 「FETCH文」
- OPEN FOR文

他の章:

- 「明示カーソルのオープンおよびクローズ」
- 「パラメータを受け入れる明示カーソル」

13.46 OPEN FOR文

OPEN FOR文は、カーソル変数を問合せに関連付け、問合せを処理するデータベース・リソースを割り当て、結果セットを識別して、カーソルを結果セットの最初の行の前に配置します。

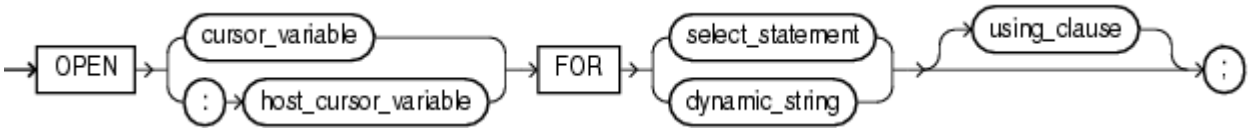
問合せにFOR UPDATE句がある場合、結果セットの行はOPEN FOR文によってロックされます。

ここでのトピック

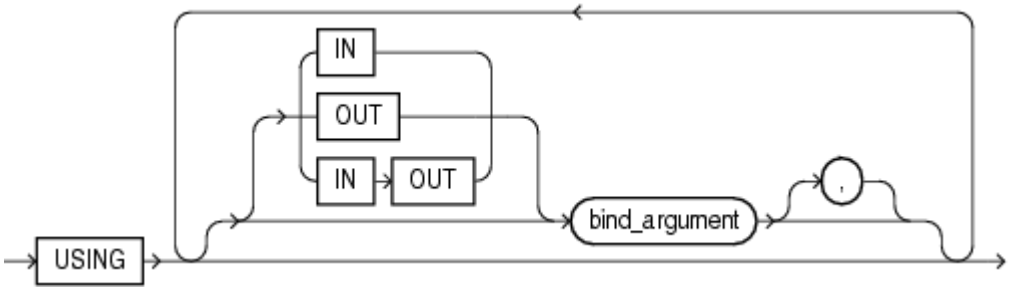
- 構文
- セマンティクス
- 例
- 関連トピック

構文

open_for_statement ::=



using_clause ::=



セマンティクス

open_for_statement

cursor_variable

カーソル変数の名前。cursor_variableがサブプログラムの仮パラメータの場合、戻り型は指定しないでください。サブプログラム・パラメータとしてのカーソル変数の詳細は、「サブプログラム・パラメータとしてのカーソル変数」を参照してください。

:host_cursor_variable

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されるカーソル変数の名前。コロン(:)とhost_cursor_variableの間に空白は入れないでください。

ホスト・カーソル変数のデータ型は、PL/SQLカーソル変数の戻り型と互換性があります。

select_statement

SQLのSELECT文(PL/SQLのSELECT INTO文ではない)。通常、select_statementは複数の行を戻します。

参照:

SELECT 文の構文は、『Oracle Database SQL 言語リファレンス』を参照してください。

dynamic_string

データ型がCHAR、VARCHAR2またはCLOBで、SQLのSELECT文を表す文字列リテラル、文字列変数または文字列式。通常、dynamic_statementは複数の行を戻すSQLのSELECT文を表します。

using_clause

位置表記法を使用して、バインド変数を指定します。

ノート:

dynamic_sql_statement 内でプレースホルダ名を繰り返す場合は、プレースホルダをバインド変数に関連付ける方法が動的 SQL 文の種類によって異なるということに注意してください。詳細は、「動的 SQL 文内でのプレースホルダ名の繰返し」を参照してください。

using_clauseの制限

select_statementまたはdynamic_sql_stmtにバインド変数のためのプレースホルダがある場合にのみ使用します。

IN、OUT、IN OUT

バインド変数のパラメータ・モード。INバインド変数は、その値をselect_statementまたはdynamic_stringに渡します。OUTバインド変数は、dynamic_stringによって戻される値を格納します。IN OUTバインド変数は、その初期値をdynamic_stringに渡し、dynamic_stringによって戻される値を格納します。デフォルト: IN。

bind_argument

実行時に、select_statementまたはdynamic_stringに含まれる対応するプレースホルダを置換する値を持つ式。すべてのプレースホルダにbind_argumentを指定する必要があります。

ノート:

バインド変数はどのような順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

bind_argumentの制限

- bind_argumentに、文字列で索引付けされている連想配列は使用できません。
- bind_argumentに、予約語NULLは使用できません。

  値NULLを動的SQL文に渡すには、例7-7で示すように、NULLを使用する場所に未初期化変数を使用します。

例

- 例6-26、カーソル変数によるデータのフェッチ
- 例6-30、静的SQLを使用したコレクションの問合せ
- 例6-31、1つの問合せ用にカーソル変数をオープンするプロシージャ
- 例6-32、選択された問合せ用のカーソル変数のオープン(同じ戻り型)
- 例6-33、選択された問合せ用のカーソル変数のオープン(異なる戻り型)
- 例7-8、OPEN FOR文、FETCH文およびCLOSE文を使用したシステム固有の動的SQL
- 例7-9、システム固有の動的SQLを使用したコレクションの問合せ

関連トピック

この章:

- CLOSE文
- カーソル変数の宣言
- EXECUTE IMMEDIATE文
- 「FETCH文」
- OPEN文

他の章:

- 「カーソル変数のオープンおよびクローズ」
- OPEN FOR文、FETCH文およびCLOSE文

13.47 PARALLEL_ENABLE句

ファンクションのパラレル実行を可能にし、パラレルDML評価の同時セッションで安全に使用できるようにします。

パラレル問合せ操作のパラレル実行サーバーからファンクションを実行できることを示します。

PARALLEL_ENABLE句は、次のSQL文に指定できます。

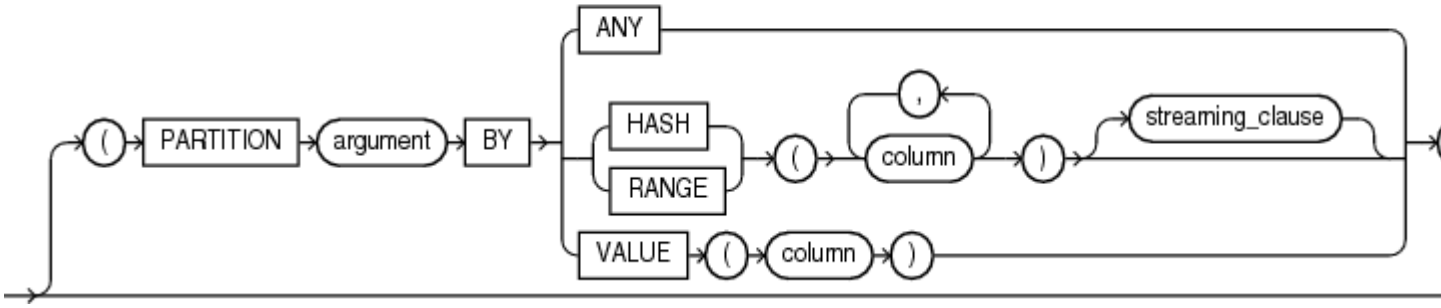
- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE TYPE BODY文

ここでのトピック

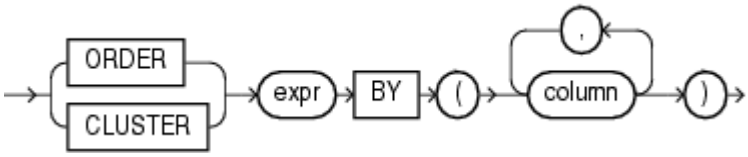
- 構文
- セマンティクス
- 関連トピック

構文

parallel_enable_clause ::=



streaming_clause ::=



セマンティクス

parallel_enable_clause

parallel_enable_clauseはファンクション内で1回のみ使用できます。

このファンクションでは、パッケージ変数などのセッション状態は使用しないでください(このような変数がパラレル実行サーバー間で共有されているとはかぎらないためです)。

オプションのPARTITION argument BY句は、REF CURSORデータ型のファンクションでのみ使用します。この句を使用すると、REF

CURSOR引数からファンクションへの入力のパーティション化を定義できます。ファンクションへの入力のパーティション化は、ファンクションが問合せのFROM句内のテーブル・ファンクションとして使用された場合の問合せのパラレル化の方法に影響します。

ANY

データをパラレル実行サーバー間でランダムにパーティション化できることを示します

> ノート:
>
> テーブル・ファンクションの弱いカーソル変数引数をパーティション化するには、RANGE、HASH や VALUE ではなく、ANY のみを使用します。

RANGEまたはHASH

ファンクションのREF CURSOR引数によって戻される指定済の列にデータをパーティション化します。

streaming_clause

オプションのstreaming_clauseを使用すると、パラレル処理を順序付けまたはクラスタ化できます。

ORDER BY | CLUSTER BY

ORDER BYまたはCLUSTER BYは、パラレル実行サーバー上の行はローカルに順序付けし、columnリストで指定されたキー値と同じキー値を持つことが必要であることを示します。

VALUE

ダイレクト・キー・パーティション化を指定します。これは、MapReduceワークロードを実行する際に使用するテーブル・ファンクション用です。columnは、データ型がNUMBERである必要があります。VALUEは、使用可能なリデューサに行処理を均等に分散します。

列内に使用可能なリデューサよりも多くのリデューサがある場合、PL/SQLでは、モジュロ演算を使用して、列内のリデューサ数を正しい範囲にマップします。

リデューサ数を計算して対応する行を処理する際に、PL/SQLでは、負の値が0(ゼロ)として処理され、正の小数値が最も近い整数に丸められます。

関連項目:

パラレル・テーブル・ファンクションを使用する方法の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

expr

exprは、パーティション化が指定されているテーブル・ファンクション、およびパラレル問合せの実行時に各同時セッションの順序付けまたはクラスタ化を指定する列を持つテーブル・ファンクションのREF CURSORパラメータの名前を示します。

parallel_enable_clauseの制限

ネストしたファンクションにparallel_enable_clauseを指定することはできません。

関連トピック

この章:

- ファンクションの宣言および定義

他の章:

- テーブル・ファンクションの概要
- パイプライン・テーブル・ファンクションの作成

13.48 PIPE ROW文

PIPE ROW文はパイプライン・テーブル・ファンクションの本体内でのみ使用できます。この文はファンクションの起動元に表の行を戻します(制御は戻しません)。

ノート:

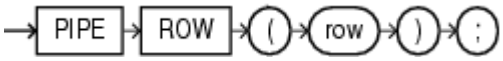
- パイプライン・テーブル・ファンクションが自律型トランザクションの一部である場合、コール元のサブプログラムでエラーが発生しないように、各 PIPE ROW 文の前に COMMIT または ROLLBACK を実行する必要があります。
- パフォーマンスを向上させるために、PL/SQL ランタイム・システムでは、パイプ化された行が起動元にバッチで送信されます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

pipe_row_statement ::=



セマンティクス

pipe_row_statement

row

ファンクションによって起動元に戻される行(表の要素)。表の要素の型を持つ式で表現します。

式がレコード変数の場合は、表の要素のデータ型を使用して明示的に宣言する必要があります。要素型と構造的にのみ一致するデータ型を使用して宣言することはできません。たとえば、要素型に名前が付いている場合は、%TYPEまたは%ROWTYPEを使用して明示的にレコード変数を宣言することや、カーソルFOR LOOP文内の%ROWTYPEを使用して暗黙的にレコード変数を宣言することはできません。

例

- 例12-29、パイプライン・テーブル・ファンクションの作成および起動
- 例12-30、パイプライン・テーブル・ファンクションによる各行の2つの行への変換
- 例12-32、2つのカーソル変数パラメータを使用するパイプライン・テーブル・ファンクション

- 例12-33、集計ファンクションとしてのパイプライン・テーブル・ファンクション
- 例12-34、パイプライン・テーブル・ファンクションで処理されないNO_DATA_NEEDED
- 例12-35、パイプライン・テーブル・ファンクションで処理されるNO_DATA_NEEDED

関連トピック

この章:

- 「ファンクションの宣言および定義」

他の章:

- 「パイプライン・テーブル・ファンクションの作成」

13.49 PIPELINED句

テーブル・ファンクションまたは多相テーブル・ファンクションの結果を繰り返し戻すようデータベースに指示します。

テーブル・ファンクションでのみ使用し、このファンクションがパイプラインであることを指定します。パイプライン・テーブル・ファンクションは、行を処理した直後に起動元に行を戻し、行の処理を継続します。起動元に(制御を戻さずに)行を戻すために、このファンクションではPIPE ROW文を使用します。

テーブル・ファンクションはコレクション型を返します。

多相テーブル・ファンクションは、戻り型が引数によって決定されるテーブル・ファンクションです。

問合せのFROM句内で、ファンクション名の前でTABLEキーワードを使用して、両方の種類のテーブル・ファンクションを問い合せます。たとえば:

```
SELECT * FROM TABLE(function_name(...))
```

テーブル・ファンクションの引数リストまたは空のlist ()がある場合、TABLE演算子は省略できます。たとえば:

```
SELECT * FROM function_name()
```

これによって、ファンクションで行が生成されると、データベースによってその行が戻されます。

PIPELINEDオプションは、次のSQL文に指定できます。

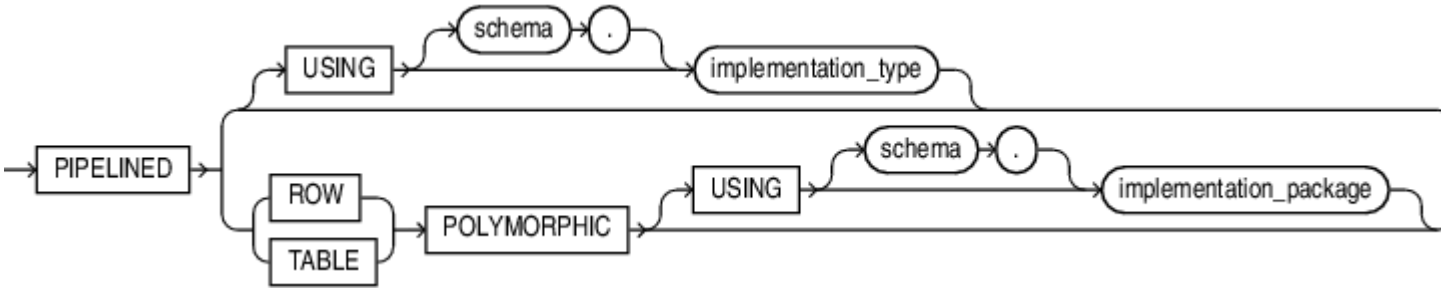
- CREATE FUNCTION文
- CREATE PACKAGE文
- CREATE PACKAGE BODY文

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

pipelined_clause ::=



セマンティクス

pipelined_clause

pipelined_clauseはファンクション内で1回のみ使用できます。

PIPELINED

パイプライン・ファンクションを作成するには、ファンクション定義にpipelinedを含めます。パイプライン・ファンクションを定義する前に宣言する場合、ファンクションの宣言でPIPELINEDオプションを指定する必要があります。

{ IS | USING }

- キーワードPIPELINEDを単独で指定する場合(PIPELINED IS ...)は、PL/SQLファンクション本体でPIPEキーワードを使用する必要があります。このキーワードは、コレクション全体を単一の値として戻すのではなく、コレクションの個々の要素をファンクションから戻すようにデータベースに指示します。
- 開始、フェッチおよび終了の操作が含まれるインタフェースを事前定義する場合は、PIPELINED USING implementation_type句を指定できます。実装タイプは、ODCITableインタフェースを実装する必要があり、テーブル・ファンクションの作成時に存在している必要があります。この句は、C++やJavaなどの外部言語で実装されるテーブル・ファンクションに便利です。

  ファンクションの戻り型がANYDATASETである場合は、ファンクションの実装タイプの一部としてdescribeメソッド(ODCITableDescribe)も定義する必要があります。

[schema.] implementation_type

この実装タイプは、ODCIAggregateサブプログラムの実装が含まれているADTにする必要があります。schemaを指定しない場合、実装タイプは自分のスキーマ内に存在するとみなされます。

PIPELINEDの制限

ネストしたファンクションにPIPELINEDを指定することはできません。

> ノート:
>
> パイプライン・テーブル・ファンクションは、データベース・リンクを介して実行することはできません。パイプライン・テーブル・ファンクションの戻り型がユーザー定義の SQL 型であり、単一データベース内でのみ使用可能であるためです(『Oracle Database オブジェクト・リレーショナル開発者ガイド』を参照)。パイプライン・テーブル・ファンクションの戻り型が PL/SQL 型のように見える場合でも、その PL/SQL 型は実際にはデータベースによって対応するユーザー定義の SQL 型に変換されます。

PIPELINED [ROW | TABLE] POLYMORPHIC [USING [schema.] implementation_package]

多相テーブル・ファンクションのエラボレータは、スタンドアロン・ファンクションの宣言またはパッケージ・ファンクションの宣言で指定できます。

PIPELINED

多相テーブル・ファンクションを定義するときに必要です。

ROW

TABLE型の単一の入力引数が、任意の単一行を使用して新しい列を判別する場合はROWを指定します。

TABLE

TABLE型の単一の入力引数が、現在の行を使用して新しい列を判別し、表全体または表の論理パーティションで動作する場合はTABLEを指定します。

POLYMORPHIC

POLYMORPHICの制限

次の句は、POLYMORPHICテーブル・ファンクションでは使用できません。

- PARALLEL_ENABLE句
- RESULT_CACHE句
- DETERMINISTICオプション
- AUTHIDプロパティ(実行者権限および定義者権限句)

[USING [schema.] implementation_package]

多相テーブル・ファンクション(PTF)実装パッケージを参照します。この仕様にはDESCRIBEメソッドを含める必要があります。OPEN、FETCH_ROWSおよびCLOSEメソッドの仕様は任意です。実装パッケージの仕様がすでに存在している必要があります(PTFとその実装が同じパッケージに作成される場合を除く)。

多相表関数とその実装メソッドが同じパッケージで定義されている場合、USING句はオプションです。

例

- 例: PIPE ROW文の例
- Skip_col多相テーブル・ファンクションの例
- To_doc多相テーブル・ファンクションの例
- Implicit_echo多相表関数の例
- DBMS_TFパッケージ・ユーティリティを使用したその他の例は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

関連トピック

この章:

- 「ファンクションの宣言および定義」

他の章:

- 「テーブル・ファンクションの概要」
- PTFの詳細は、多相テーブル・ファンクションの概要を参照してください。
- 「サブプログラムの各部」
- 「パイプライン・テーブル・ファンクションの作成」

- 「複数変換用のパイプライン・テーブル・ファンクションの連鎖」

その他のドキュメント:

- 多相テーブル・ファンクション(PTF)実装のユーティリティが含まれているDBMS_TFパッケージの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。
- パイプライン・テーブル・ファンクションを使用する方法の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

13.50 プロシージャの宣言および定義

プロシージャは、起動する前に宣言および定義する必要があります。最初に宣言して(procedure_declarationを使用)、後で同じブロック、サブプログラムまたはパッケージ内で定義する(procedure_declarationを使用)か、または宣言と定義を同時に行う(procedure_declarationを使用)ことができます。

プロシージャとは、特定のアクションを実行するサブプログラムのことです。プロシージャの起動(またはコール)は、式の一部です。

プロシージャ宣言は、プロシージャ仕様部とも呼ばれます。

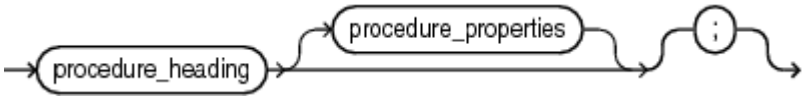
> ノート:
>
> スタンドアロン・プロシージャの詳細は、CREATE PROCEDURE 文を参照してください。パッケージ・プロシージャの詳細は、CREATE PACKAGE 文を参照してください。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

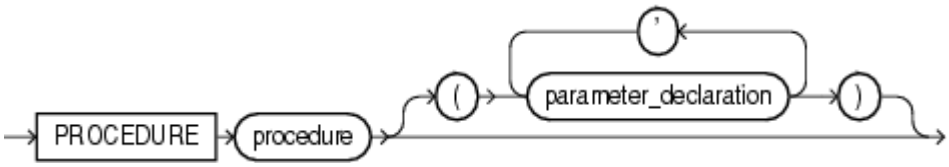
構文

procedure_declaration ::=



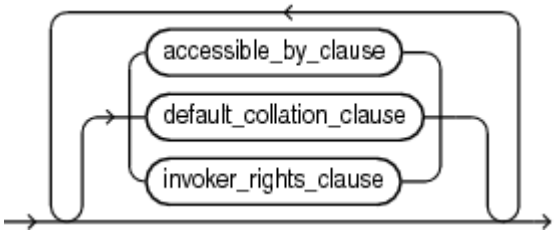
(procedure_properties ::=)

procedure_heading ::=



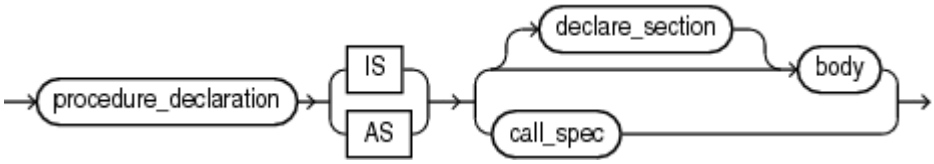
parameter_declaration ::=を参照してください。

procedure_properties ::=



(accessible_by_clause ::=、default_collation_clause ::=、invoker_rights_clause ::=)

procedure_definition ::=



(body ::=、declare_section ::=、call_spec ::=)

セマンティクス

procedure_declaration

プロシージャを宣言しますが、定義はしません。定義は、同じブロック、サブプログラムまたはパッケージ内に後で宣言として存在している必要があります。

procedure_heading

procedure

宣言または定義するプロシージャの名前。

procedure_properties

プロシージャの各プロパティは、プロシージャの宣言で1回のみ記述できます。プロパティは任意の順序で記述できます。プロパティは、ヘッダー内のISまたはASキーワードの前に記述します。プロパティは、ネストしたプロシージャには記述できません。パッケージ・プロシージャに記述できるのは、ACCESSIBLE BYプロパティのみです。

スタンドアロン・プロシージャの宣言には次のプロパティを使用できます。

- ACCESSIBLE BY句
- DEFAULT COLLATION句
- 実行者権限および定義者権限(AUTHIDプロパティ)

procedure_definition

すでに宣言済のプロシージャを定義するか、またはプロシージャの宣言と定義の両方を行います。

declare_section

プロシージャに対してローカルである項目を宣言し、bodyで参照でき、プロシージャが実行を完了すると消滅します。

body

プロシージャの必須の実行部およびオプションの例外処理部。

例

- 例8-1、単純なPL/SQLプロシージャの宣言、定義および起動

関連トピック

この章:

- 仮パラメータの宣言
- 「ファンクションの宣言および定義」

他の章:

- PL/SQLサブプログラム
- CREATE PROCEDURE文

13.51 RAISE文

RAISE文は、例外を明示的に呼び出します。

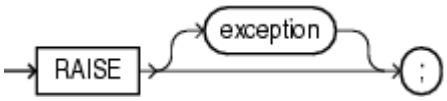
例外ハンドラの外側では、例外名を指定する必要があります。例外ハンドラの内側で、例外名を省略すると、RAISE文は現行の例外を再度呼び出します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

raise_statement ::=



セマンティクス

exception

事前定義の例外(表11-3を参照)またはユーザー宣言の例外(例外宣言を参照)の名前。

exceptionは、例外ハンドラのみでのオプションです。デフォルトは現行の例外です(「RAISE文を使用した現行の例外の再呼出し」を参照)。

例

- 例11-10、ユーザー定義の例外の宣言、呼出しおよび処理
- 例11-11、事前定義の例外の明示的な呼出し
- 例11-12、例外の再呼出し

関連トピック

この章:

- 例外宣言
- 例外ハンドラ

他の章:

- 例外の明示的な呼出し

13.52 レコード変数の宣言

レコード変数はコンポジット変数であり、フィールドと呼ばれるその内部コンポーネントは別々のデータ型にすることができます。レコード変数の値およびそのフィールドの値は変更可能です。

レコード変数全体はレコード変数の名前で参照します。レコード・フィールドを参照するには、構文record.fieldを使用します。

次のいずれかの方法でレコード変数を作成できます。

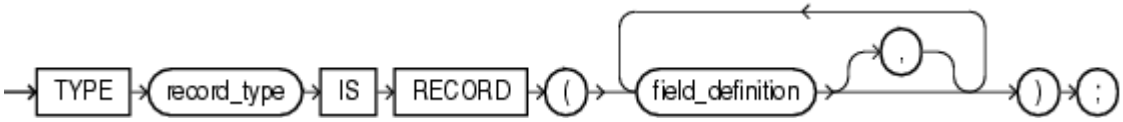
- レコード型を定義し、次にその型の変数を宣言します。
- データベースの表またはビュー内の行の全体または一部を表すレコード変数を宣言するには、%ROWTYPEを使用します。
- 事前に宣言されているレコード変数と同じ型のレコード変数を宣言するには、%TYPEを使用します。

ここでのトピック

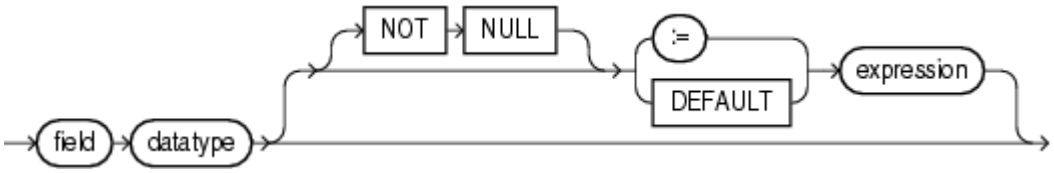
- 構文
- セマンティクス
- 例
- 関連トピック

構文

record_type_definition ::=

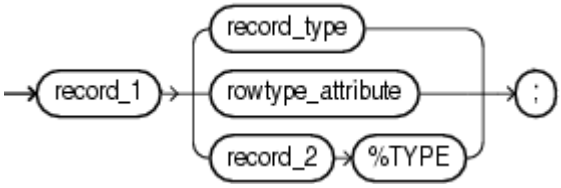


field_definition ::=



(datatype ::= , expression ::=)

record_variable_declaration ::=



(rowtype_attribute ::=)

セマンティクス

record_type_definition

record_type

定義するレコード型の名前。

field_definition

field

定義するフィールドの名前。

datatype

定義するフィールドのデータ型。

NOT NULL

定義するフィールドにNOT NULL制約を付けます。

この制約の詳細は、「NOT NULL制約」を参照してください。

expression

datatypeと互換性のあるデータ型の式。record_variable_declarationが処理されるとき、expressionの値がrecord.fieldに代入されます。この値が、フィールドの初期値です。

record_variable_declaration

record_1

宣言するレコード変数の名前。

record_type

事前に定義されたレコード型の名前。record_typeはrecord_1のデータ型です。

rowtype_attribute

%ROWTYPE属性を参照してください。

record_2

事前に宣言されているレコード変数の名前。

%TYPE

%TYPE属性を参照してください。

例

- 例5-36、RECORD型の定義および変数の宣言
- 例5-37、RECORDフィールドを持つRECORD型(ネストされたレコード)
- 例5-38、VARRAYフィールドを持つRECORD型

関連トピック

この章:

- コレクション変数の宣言
- %ROWTYPE属性

他の章:

- レコードに関するトピック

13.53 RESTRICT_REFERENCESプラグマ

RESTRICT_REFERENCESプラグマは、ユーザー定義のサブプログラムがデータベース表またはパッケージ変数に対する読込みまたは書込みを行っていないことを示します。

ノート:

RESTRICT_REFERENCES プラグマは非推奨となりました。RESTRICT_REFERENCES のかわりに DETERMINISTIC および PARALLEL_ENABLE を使用することをお薦めします。

データベース表またはパッケージ変数に対して読込みまたは書込み操作を行っているサブプログラムは、サブプログラムの起動によって結果が異なったり、エラーが発生する場合があるため、最適化は困難です。TRUSTを指定しないかぎり、ユーザー定義サブプログラム内の文がRESTRICT_REFERENCESにより行われたアサーションに違反する場合、PL/SQLコンパイラにより文の解析時にエラー・メッセージが発行されます。

通常、このプラグマはファンクションに対して指定します。ファンクションがプロシージャを起動する場合は、それらのプロシージャにも同様にこのプラグマを指定します。

RESTRICT_REFERENCESプラグマの制限

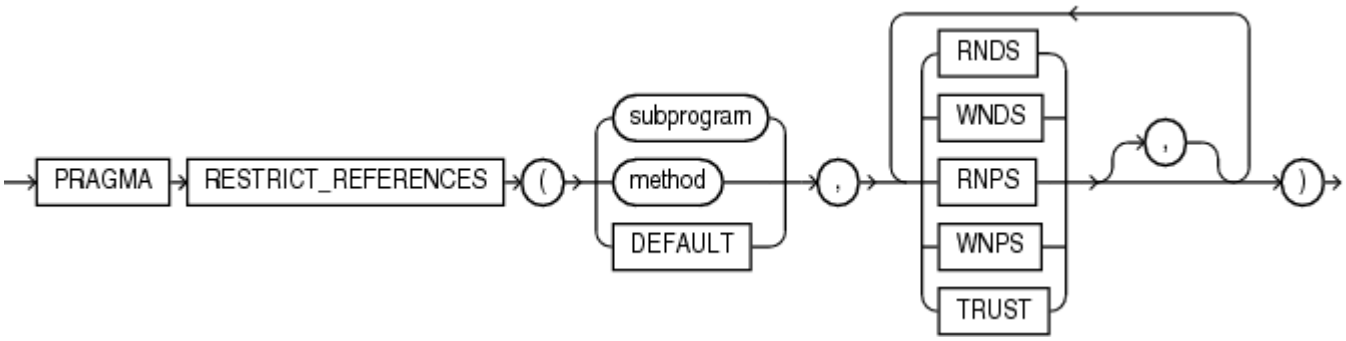
- このプラグマは、パッケージ仕様部またはADT仕様部でのみ使用できます。
- 所定のサブプログラム宣言を参照できるRESTRICT_REFERENCESプラグマは、1つのみです。

ここでのトピック

- 構文
- セマンティクス

構文

restrict_references_pragma ::=



セマンティクス

subprogram

ユーザー定義サブプログラム(通常はファンクション)の名前。subprogramがオーバーロードされた場合、プラグマは直前のサブプログラム宣言にのみ適用されます。

method

MEMBERサブプログラムの名前。詳細は、「CREATE TYPE文」を参照してください。

DEFAULT

プラグマをパッケージ仕様部またはADTの仕様部内のすべてのサブプログラム(ADTに対するシステム定義のコンストラクタを含む)に適用します。

個々のサブプログラムに対してもプラグマを宣言すると、そのサブプログラムのDEFAULTのプラグマはオーバーライドされます。

RNDS

サブプログラムがデータベースに読込み禁止状態である(データベース表を問合せできない)ことを示します。

WNDS

サブプログラムがデータベースへの書込み禁止状態である(表を変更できない)ことを示します。

RNPS

サブプログラムがパッケージに読込み禁止状態である(パッケージ変数の値を参照できない)ことを示します。

RNPSの制限

サブプログラムがSQLCODEファンクションまたはSQLERRMファンクションを起動する場合、RNPSを指定することはできません。

WNPS

サブプログラムがパッケージへの書込み禁止状態である(パッケージ変数の値を変更できない)ことを示します。

WNPSの制限

サブプログラムがSQLCODEファンクションまたはSQLERRMファンクションを起動する場合、WNPSを指定することはできません。

TRUST

指定された他のアサーションにサブプログラムが違反しないと信頼できることを示し、PL/SQLコンパイラがサブプログラム本体で違反の有無をチェックすることを防ぎます。これらのチェックをスキップすると、パフォーマンスが向上します。

PL/SQLサブプログラムがCまたはJavaサブプログラムを起動する場合、PL/SQLコンパイラは実行時にCまたはJavaサブプログラムで違反の有無をチェックできないため、PL/SQLサブプログラムあるいはCまたはJavaサブプログラムに対してTRUSTを指定する必要があります。

ノート:

パラレル化された DML 文からサブプログラムを起動するには、4 つの制約(RNDS、WNDS、RNPS および WNPS)すべてを指定する必要があります。他の制約を暗黙的に指定する制約はありません。

関連項目:

既存のアプリケーションでのPRAGMA RESTRICT_REFERENCESの使用の詳細は、『Oracle Database開発ガイド』を参照してください

13.54 RETURN文

RETURN文は、サブプログラムまたはこの文を含む無名ブロックの実行を即座に終了させます。

ファンクションの中のRETURN文は、指定された値をファンクション識別子に代入し、起動元に制御を戻します(ここで、起動すると(多くは起動する文の内側で)実行が即座に再開されます)。ファンクション内のすべての実行パスは、RETURN文へ導くものである必要があります(それ以外の場合、PL/SQLコンパイラはコンパイル時に警告PLW-05005を発行します)。

プロシージャでは、RETURN文は起動元に制御を戻します。ここで、起動すると実行が即座に再開します。

無名ブロックでは、RETURN文によってそのブロックおよびすべての外側のブロックが終了されます。

サブプログラムまたは無名ブロックでは、複数のRETURN文を使用できます。

ノート:

RETURN 文は、ファンクションのヘッダー内で戻り値のデータ型を指定する RETURN 句とは異なります。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

return_statement ::=

(expression ::=)

セマンティクス

expression

RETURN文がパイプライン・テーブル・ファンクションに含まれている場合はオプションです。RETURN文がその他のファンクションに含まれている場合は必須です。RETURN文がプロシージャまたは無名ブロックに内にある場合は使用できません。

RETURN文は、expressionの値をファンクション識別子に代入します。そのため、expressionのデータ型は、ファンクションのRETURN句内のデータ型と互換性がある必要があります。式の詳細は、「式」を参照してください。

例

- 例8-3、ファンクション内にあるRETURN文の後での実行の再開
- 例8-4、RETURN文へ導かれない実行パスを含むファンクション

- 例8-5、すべての実行パスがRETURN文へ導かれるファンクション
- 例8-6、プロシージャ内にあるRETURN文の後での実行の再開
- 例8-7、無名ブロック内にあるRETURN文の後での実行の再開

関連トピック

この章:

- ブロック
- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」

他の章:

- RETURN文

13.55 RETURNING INTO句

RETURNING INTO句は、この句が属する文によって戻された値を格納するための変数を指定します。

変数は、個々の変数またはコレクションのいずれかにできます。文を実行しても行に影響がない場合、変数の値は未定義になります。

静的RETURNING INTO句は、DELETE、INSERTまたはUPDATE文に属します。動的RETURNING INTO句は、EXECUTE IMMEDIATE文に属します。

ノート:

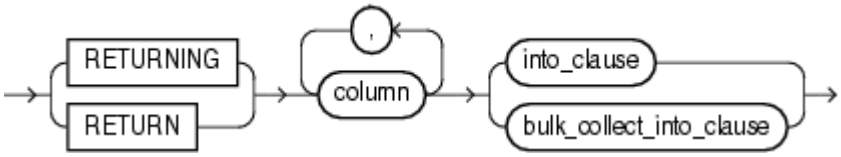
RETURNING INTO 句はリモートまたはパラレルでの削除には使用できません。

ここでのトピック

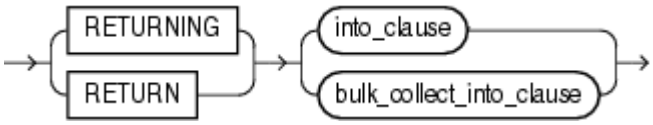
- 構文
- セマンティクス
- 例
- 関連トピック

構文

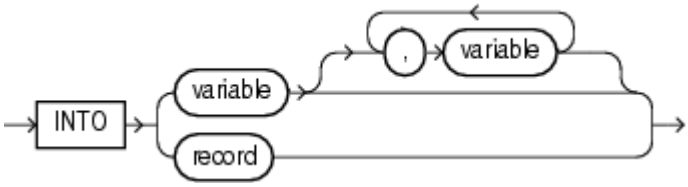
static_returning_clause ::=



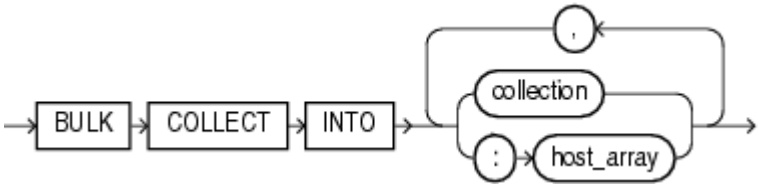
dynamic_returning_clause ::=



into_clause ::=



bulk_collect_into_clause ::=



セマンティクス

static_returning_clause

column

データベース表の列の名前の値を持つ式。

into_clause

文によって戻される列値を格納する変数またはレコードを指定します。

into_clauseの制限

dynamic_sql_stmt(「EXECUTE IMMEDIATE文」を参照)が単一行を戻すことができる場合にのみ、dynamic_returning_clause内でinto_clauseを使用します。

record

文によって戻される行を格納するレコード変数の名前。文内の各select_listに対して、レコードの中に、対応する型互換のフィールドが存在している必要があります。

variable

文によって戻された列を格納するスカラー変数の名前、またはPL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡されるホスト・カーソル変数の名前。文内の各select_list項目に対して、対応する型互換の変数が存在している必要があります。ホスト・カーソル変数のデータ型は、PL/SQLカーソル変数の戻り型と互換性があります。

variableの制限

変数のデータ型は、BOOLEANにすることはできません。

bulk_collect_into_clause

文によって戻された行を格納する1つ以上の既存のコレクションまたはホスト配列を指定します。文内の各select_list項目に対して、bulk_collect_into_clauseには、対応する型互換のcollectionまたはhost_arrayが存在している必要があります。

この句を使用する理由は、「バルクSQLおよびバルク・バインド」を参照してください。

bulk_collect_into_clauseの制限

- dynamic_sql_stmt(「EXECUTE IMMEDIATE文」を参照)が複数行を戻すことができる場合にのみ、dynamic_returning_clause内でbulk_collect_into_clause句を使用します。
- bulk_collect_into_clauseは、クライアント・プログラムでは使用できません。
- bulk_collect_into_clauseを含む文で暗黙のデータ型変換が必要な場合、bulk_collect_into_clauseに含めることができるのはcollectionまたはhost_arrayを1つのみです。

collection

文によって戻される行を格納するコレクション変数の名前。

collectionの制限

- collectionに、文字列で索引付けされた連想配列の名前は使用できません。
- 暗黙的なデータ型変換が文に必要となる場合、collectionにコンポジット型のコレクションの名前は使用できません。

:host_array

PL/SQLホスト環境で宣言され、バインド変数としてPL/SQLに渡される配列の名前。コロン(:)とhost_arrayの間に空白は入れないでください。

例

- 例5-52、UPDATE文を使用したレコード変数への値の代入
- 例6-1、静的SQL文
- 例12-25、2つのネストした表に対する削除行の戻し
- 例12-26、FORALL文でのRETURN BULK COLLECT INTO付きのDELETE

関連トピック

この章:

- 「DELETE文の拡張機能」
- EXECUTE IMMEDIATE文
- 「FETCH文」
- 「SELECT INTO文」
- 「UPDATE文の拡張機能」

他の章:

- 「PL/SQLレコード変数に行を戻すSQL文の使用」
- EXECUTE IMMEDIATE文
- 「BULK COLLECT句を使用したRETURNING INTO句」

13.56 RESULT_CACHE句

ファンクションの結果をサーバーの結果キャッシュに保存するように指示します。

RESULT_CACHE句は、次のSQL文に指定できます。

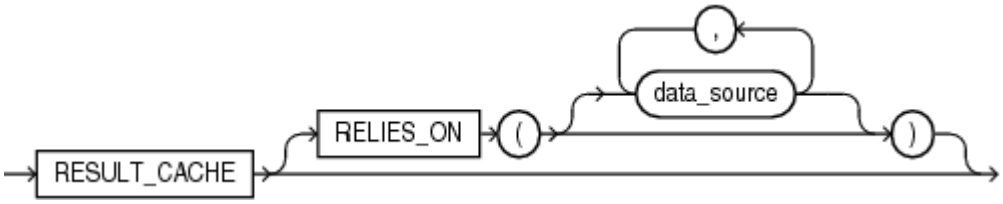
- CREATE FUNCTION文
- CREATE TYPE BODY文

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

result_cache_clause ::=



セマンティクス

result_cache_clause

result_cache_clauseは、ファンクションで1回のみ記述できます。

RESULT_CACHE

ファンクションの結果がキャッシュされるようにするには、RESULT_CACHE句をファンクション定義に含めます。そのファンクションを定義する前に宣言する場合は、そのファンクションの宣言にRESULT_CACHEオプションも含める必要があります。

RESULT_CACHEの制限

- RESULT_CACHEは、OUTパラメータまたはIN OUTパラメータを持つファンクションには使用できません。
- RESULT_CACHEは、次の型(または次の型を含む) INパラメータまたはRETURNパラメータを持つファンクションには使用できません。
  - BLOB
  - CLOB
  - NCLOB
  - REF CURSOR
  - コレクション
  - オブジェクト

- サポートされていない戻り型が含まれているレコードまたはPL/SQLコレクション
- RESULT_CACHEは、無名ブロック内のファンクションには使用できません。
- RESULT_CACHEは、パイプライン・テーブル・ファンクションとネストしたファンクションには使用できません。

RELIES_ON

ファンクションの結果が依存するデータ・ソースを指定します。それぞれのdata_sourceがデータベース表またはデータベース・ビューの名前です。

ノート:

- この句は非推奨です。Oracle Database 12c 以降は、結果がキャッシュされるファンクションの実行中に問い合せられるすべてのデータ・ソースがデータベースによって検出され、RELIES_ON 句は何も実行しません。
- 無名ブロック内で宣言されたファンクションで RELIES_ON 句を使用することはできません。

例

- 結果がキャッシュされるファンクションの例

関連トピック

この章:

- ファンクションの宣言および定義

他の章:

- PL/SQLファンクション結果キャッシュ

その他のドキュメント:

- 『Oracle Databaseパフォーマンス・チューニング・ガイド』

13.57 %ROWTYPE属性

%ROWTYPE属性を使用すると、データベースの表またはビュー内の行の全体または一部を表すレコードを宣言できます。

このレコードは、行の全体または一部のすべての表示列に対して、同じ名前とデータ型のフィールドを持ちます。行の構造が変更されると、それに応じてレコードの構造も変更されます。非表示列を表示列に変更すると、%ROWTYPE属性で宣言されているいくつかのレコードの構造が変わります。

このレコードのフィールドは、対応する列の制約または初期値を継承しません。

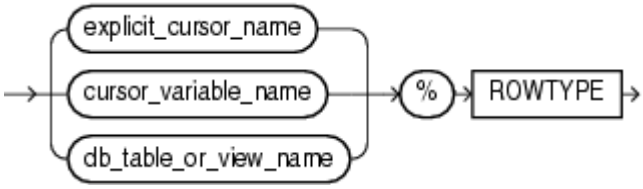
参照された文字の列にUSING_NLS_COMP以外の照合がある場合、%ROWTYPE属性は使用できません。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

rowtype_attribute ::=



セマンティクス

rowtype_attribute

explicit_cursor_name

明示カーソルの名前。このレコードは、explicit_cursor_nameと関連付けられた問合せによって選択されたすべての列に対して、同じ名前とデータ型のフィールドを持ちます。

cursor_variable_name

強いカーソル変数の名前。このレコードは、cursor_variable_nameと関連付けられた問合せによって選択されたすべての列に対して、同じ名前とデータ型のフィールドを持ちます。

db_table_or_view_name

宣言が処理されるときにアクセスできるデータベースの表またはビューの名前。このレコードは、db_table_or_view_nameのすべての列に対して、同じ名前とデータ型のフィールドを持ちます。

例

- 例5-40、データベース表の行全体を表す%ROWTYPE変数

- 例5-41、初期値または制約を継承しない%ROWTYPE変数
- 例5-42、データベース表の行の一部を表す%ROWTYPE変数
- 例5-43、結合行を表す%ROWTYPE変数
- 例5-46、非表示列の表示列への変更により影響を受ける%ROWTYPE
- 例5-48、RECORD型のレコードへの%ROWTYPEレコードの代入

関連トピック

この章:

- カーソル変数の宣言
- 「明示カーソルの宣言および定義」
- レコード変数の宣言
- %TYPE属性

他の章:

- データ・バインドされた照合について
- %ROWTYPE属性を使用した項目の宣言
- %ROWTYPE属性と非表示列

13.58 スカラー変数の宣言

スカラー変数は、内部コンポーネントがない値を格納します。値は変更できます。スカラー変数の宣言では、変数の名前およびデータ型を指定し、使用する記憶域を割り当てます。

また、初期値を代入したり、NOT NULL制約を付けることもできます。

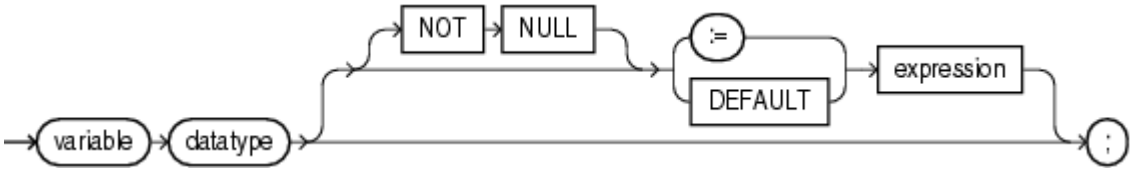
スカラー変数は、その名前を使用して参照します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

variable_declaration ::=



(expression ::=)

セマンティクス

variable_declaration

variable

宣言する変数の名前。

datatype

サイズ、精度および文字またはバイトのセマンティクスの修飾子が含まれているスカラー・データ型の名前。

スカラー・データ型の詳細は、「PL/SQLデータ型」を参照してください。

NOT NULL

変数にNOT NULL制約を付けます。この制約の詳細は、「NOT NULL制約」を参照してください。

expression

宣言が処理されるときに変数に代入される値です。expressionとvariableは互換性のあるデータ型にする必要があります。

例

- 例2-11、スカラー変数の宣言

- 例2-13、初期値が指定された変数および定数の宣言
- 例2-14、デフォルトでNULLに初期化される変数
- 例2-9、NOT NULL制約が指定された変数の宣言

関連トピック

この章:

- 代入文
- コレクション変数の宣言
- 定数の宣言
- 「式」
- レコード変数の宣言
- %ROWTYPE属性
- %TYPE属性

他の章:

- 変数の宣言
- PL/SQLデータ型

13.59 SELECT INTO文

SELECT INTO文は、(SQLのSELECT文と同様に)1つ以上のデータベース表から値を取り出し、それらの値を変数に格納します(SQLのSELECT文ではこの処理は実行されません)。

> 注意:
>
> SELECT INTO 文を BULK COLLECT 句とともに使用すると、エイリアシングに対して脆弱になり、予期しない結果が発生する場合があります。詳細は、「SELECT BULK COLLECT INTO 文およびエイリアシング」を参照してください。

関連項目:

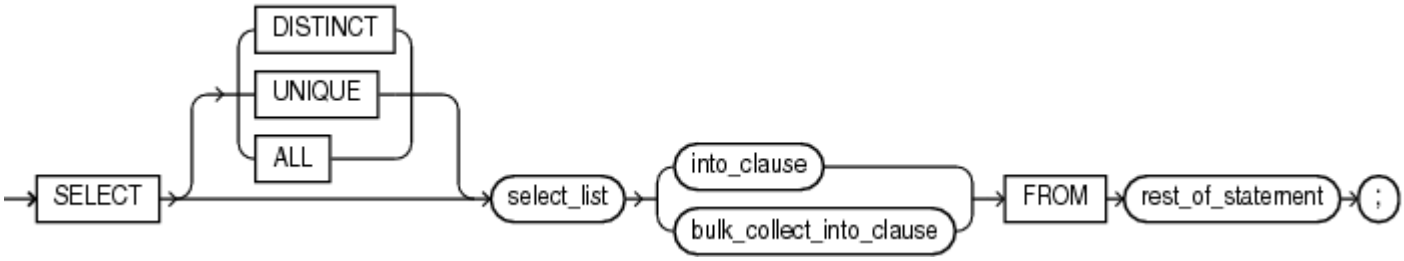
SQLのSELECT文の構文は、『Oracle Database SQL言語リファレンス』を参照してください。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

select_into_statement ::=



(bulk_collect_into_clause ::=, into_clause ::=、select_list構文の詳細はOracle Database SQL言語リファレンスを参照)

セマンティクス

select_into_statement

DISTINCT | UNIQUE

DISTINCTまたはUNIQUEを指定すると、選択された重複行の1行のみを戻すことができます。これらの2つのキーワードは同義です。重複行とは、SELECT構文のリスト中のそれぞれの式で一致する値を持つ行のことです。

DISTINCT問合せおよびUNIQUE問合せの制限事項

- すべてのselect list式の総バイト数は、データ・ブロックからオーバーヘッドを除いたサイズに制限されます。このサイズは、初期化パラメータDB_BLOCK_SIZEで指定されます。
- select_listにLOB列が含まれている場合、DISTINCTは指定できません。

ALL

(デフォルト)すべての重複行のコピーを含め、選択された行がすべて戻されるようにできます。

select_list

SELECT INTO文によって行が戻されない場合、PL/SQLによって事前定義の例外NO_DATA_FOUNDが呼び出されます。この例外を回避するには、条件に一致する行がない場合でも、単一の値を戻す集計ファンクションCOUNT(*)の結果を選択します。

into_clause

この句を使用すると、SELECT INTO文は、単一の行から1つ以上の列を取り出し、それらの値を1つ以上のスカラー変数または1つのレコード変数に格納します。詳細は、into_clause ::=を参照してください。

bulk_collect_into_clause

この句を使用すると、SELECT INTO文は結果セット全体を取り出し、それを1つ以上のコレクション変数に格納します。詳細は、bulk_collect_into_clause ::=を参照してください。

rest_of_statement

SQLのSELECT文のキーワードFROMに続けることができる任意の構造体(『Oracle Database SQL言語リファレンス』を参照)。

例

- 例2-25、SELECT INTO文を使用した変数への値の代入
- 例5-50、SELECT INTOを使用したレコード変数への値の代入
- 例6-37、ROLLBACK文
- 例6-38、SAVEPOINT文およびROLLBACK文
- 例6-43、パッケージでの自律型ファンクションの宣言
- 例7-20、SQLインジェクションを回避するための妥当性チェック
- 例12-16、2つのネストした表に対するデータベースの2つの列のバルク選択
- 例12-17、レコードのネストした表に対するバルク選択
- 例12-21、ROWNUM、SAMPLEおよびFETCH FIRSTによるバルク選択の制限

関連トピック

この章:

- 代入文
- 「FETCH文」

- %ROWTYPE属性

他の章:

- 「SELECT INTO文を使用した変数への値の代入」
- 「SELECT INTOを使用した行のレコード変数への代入」
- 「SELECT INTO文による問合せ結果セットの処理」
- 「BULK COLLECT句を使用したSELECT INTO文」

関連項目:

SQLのSELECT文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

13.60 SERIALLY_REUSABLEプラグマ

SERIALLY_REUSABLEプラグマは、サーバーへの1コール(データベースへのOCIコール、データベース・リンクを介したストアド・プロシージャの起動など)に対してのみ、そのパッケージ状態が必要であることを指定します。

このコールの後、パッケージ変数の記憶域の再利用が可能になり、長時間実行セッションのメモリー・オーバーヘッドが削減されます。

このプラグマは、同じセッションで1回使用される大規模な一時作業領域を宣言するパッケージに適しています。

SERIALLY_REUSABLEプラグマは、本体のないパッケージの仕様部のdeclare_section、またはパッケージの仕様部と本体の両方で使用できますが、パッケージの本体のみでは使用できません。

ここでのトピック

- 構文
- 例
- 関連トピック

構文

serially_reusable_pragma ::=

例

- 例10-4、SERIALLY_REUSABLEパッケージの作成
- 例10-5、SERIALLY_REUSABLEプラグマの影響
- 例10-6、コール境界でオープンされるSERIALLY_REUSABLEパッケージのカーソル

関連トピック

- SERIALLY_REUSABLEパッケージ
- 「プラグマ」

13.61 SHARING句

この句は、アプリケーション・ルートでオブジェクトを作成する場合にのみ適用されます。このタイプのオブジェクトはアプリケーション共通オブジェクトと呼ばれ、アプリケーション・ルートに属するアプリケーションPDBと共有できます。SHARING句を記述できるのは、このコンテキストにおけるライブラリ、スタンドアロン・オブジェクト型、パッケージおよびスタンドアロン・サブプログラムの宣言内のみです。

SHARING句は、次のSQL文でアプリケーション共通オブジェクトを作成する際のコンテキストでのみ使用できます。

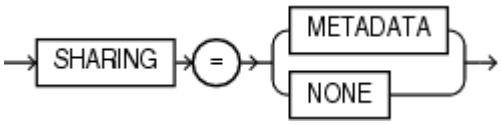
- CREATE FUNCTION文
- CREATE LIBRARY文
- CREATE PROCEDURE文
- CREATE PACKAGE文
- CREATE TRIGGER文
- CREATE TYPE文

ここでのトピック

- 構文
- セマンティクス
- 関連トピック

構文

sharing_clause ::=



セマンティクス

sharing_clause

次の共有属性のいずれかを使用してオブジェクトを共有する方法を指定します。

- METADATA - メタデータ・リンクはメタデータを共有しますが、そのデータは各コンテナに固有です。このタイプのオブジェクトは、メタデータリンク・アプリケーション共通オブジェクトと呼ばれます。
- NONE - オブジェクトは共有されません。

アプリケーションの操作中にこの句を省略すると、データベースではDEFAULT_SHARING初期化パラメータが使用され、オブジェクトの共有属性が決定されます。DEFAULT_SHARING初期化パラメータに値がない場合、デフォルトはMETADATAです。

SHARING句の制限

共有句は、アプリケーション・ルートでのアプリケーションのインストール、アップグレードまたはパッチ適用時にのみ使用できます。ALTER PLUGGABLE DATABASE APPLICATION ... BEGIN文を発行して操作を開始し、ALTER PLUGGABLE DATABASE APPLICATION ... END文を発行して操作を終了する必要があります。このコンテキスト以外のsharing_clauseは不正であ

り、オブジェクトが共有されないことを意味します。

オブジェクトの共有属性を作成後に変更することはできません。

関連トピック

その他のドキュメント:

- アプリケーションのメンテナンスの詳細は、Oracle Database概要を参照してください
- 自動化によるアプリケーションへのパッチ適用の例は、Oracle Database概要を参照してください
- DEFAULT_SHARING初期化パラメータの詳細は、Oracle Databaseリファレンスを参照してください
- アプリケーション共通オブジェクトの作成の詳細は、『Oracle Database管理者ガイド』を参照してください。

13.62 SQLCODEファンクション

例外ハンドラでは、SQLCODEファンクションは、処理する例外の数値コードを戻します。(例外ハンドラの外側では、SQLCODEは0を戻します。)

内部的に定義された例外の場合、数値コードは、関連するOracle Databaseエラーの番号になります。この番号は、エラー「データが見つかりません。」(数値コードは+100)を除き、負の値になります。

ユーザー定義の例外の場合、数値コードは+1(デフォルト)か、またはEXCEPTION_INITプラグマによってその例外に関連付けられているエラー・コードのいずれかになります。

SQL文では、SQLCODEを起動できません。

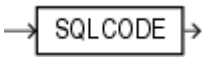
ファンクションによってSQLCODEが起動され、そのファンクションの純正度を示すためにRESTRICT_REFERENCESプラグマを使用する場合、WNPSおよびRNPS制約は指定できません。

ここでのトピック

- 構文
- 例
- 関連トピック

構文

sqlcode_function ::=



例

- 例11-23、SQLCODEおよびSQLERRM値の表示

関連トピック

この章:

- ブロック
- EXCEPTION_INITプラグマ
- 例外ハンドラ
- RESTRICT_REFERENCESプラグマ
- SQLERRMファンクション

他の章:

- 「 エラー・コードとエラー・メッセージの取得」

関連項目:

Oracle Databaseエラー・メッセージおよびそれらに関する情報(エラー番号など)のリストは、Oracle Databaseエラー・メッセージ・リファレンスを参照してください。

13.63 SQLERRMファンクション

SQLERRMファンクションは、エラー・コードに関連付けられているエラー・メッセージを戻します。

> ノート:
>
> エラー・メッセージの言語は、NLS_LANGUAGE パラメータによって決まります。このパラメータの詳細は、『Oracle Database グローバリゼーション・サポート・ガイド』を参照してください。

SQL文では、SQLERRMを起動できません。

ファンクションによってSQLERRMが起動され、そのファンクションの純正度を示すためにRESTRICT_REFERENCESプラグマを使用する場合、WNPSおよびRNPS制約は指定できません。

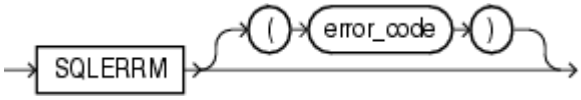
> ノート:
>
> FORALL 文を SAVE EXCEPTIONS 句とともに使用しないかぎり、SQLERRM より DBMS_UTILITY.FORMAT_ERROR_STACK を使用することをお薦めします。詳細は、「エラー・コードとエラー・メッセージの取得」を参照してください。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

sqlerrm_function ::=



セマンティクス

sqlerrm_function

error_code

Oracle Databaseエラー・コードの値を持つ式。

デフォルト: エラー・コードは、SQLCODEの現在の値に関連付けられています。

SQLCODEと同様に、error_codeのないSQLERRMは、例外ハンドラ内でのみ有効です。例外ハンドラの外側、またはerror_codeの値が0(ゼロ)の場合、SQLERRMはORA-0000を戻します。

error_codeの値が+100の場合、SQLERRMはORA-01403を戻します。

error_codeの値が+100以外の正の数値の場合、SQLERRMは次のメッセージを戻します。

```
-error_code: non-ORACLE exception
```

error_codeの値が負の数値で、その絶対値がOracle Databaseエラー・コードの場合、SQLERRMは、そのエラー・コードに関連付けられているエラー・メッセージを戻します。たとえば:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-6511): ' || TO_CHAR(SQLERRM(-6511)));
END;
/
```

結果:

```
SQLERRM(-6511): ORA-06511: PL/SQL: cursor already open
```

error_codeの値が負の数値で、その絶対値がOracle Databaseエラー・コードではない場合、SQLERRMは次のメッセージを戻します。

```
ORA-error_code: Message error_code not found; product=RDBMS;
facility=ORA
```

たとえば:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-50000): ' || TO_CHAR(SQLERRM(-50000)));
END;
/
```

結果:

```
SQLERRM(-50000): ORA-50000: Message 50000 not found; product=RDBMS;
facility=ORA
```

例

- 例11-23、SQLCODEおよびSQLERRM値の表示
- 例12-13、FORALL文が完了した後のFORALL例外の処理
- 例12-13、FORALL文が完了した後のFORALL例外の処理

関連トピック

この章:

- ブロック
- EXCEPTION_INITプラグマ
- RESTRICT_REFERENCESプラグマ
- SQLCODEファンクション

他の章:

- 「 エラー・コードとエラー・メッセージの取得」

関連項目:

Oracle Databaseエラー・メッセージおよびそれらに関する情報のリストは、Oracle Databaseエラー・メッセージ・リファレンスを参照してください。

13.64 SQL_MACRO句

SQL_MACRO句は、表式で使用できるSQLマクロとしてファンクションにマークを付けます。

ビューで参照されるSQLマクロは、常にビューの所有者の権限で実行されます。

AUTHIDプロパティは指定できません。SQLマクロは呼び出されると、実行者の正しいファンクションのように動作します。SQLマクロ所有者は、起動元のファンクションに継承権限を付与する必要があります。

マクロ注釈付きファンクションをPL/SQLで使用する場合は、通常のファンクション戻り文字またはCLOB型のようにマクロ拡張なしで動作します。

SQL_MACROの注釈は、次のSQL文で使用できます。

- CREATE FUNCTION文

SQLマクロの使用上の制限:

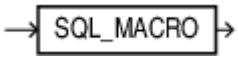
- SQLマクロは、問合せ表式のFROM句でのみ使用できます。
- SQLマクロは、仮想列式、ファンクション索引、エディショニング・ビューまたはマテリアライズド・ビューでは使用できません。
- タイプ・メソッドには、SQL_MACROで注釈を付けることはできません。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

sql_macro_clause ::=



セマンティクス

sql_macro_clause

sql_macro_clauseは、ファンクション内で1回のみ使用できます。SQLマクロ・ファンクションを作成するには、ファンクション定義にsql_macro_clauseを含めます。SQLマクロ・ファンクションを定義前に宣言する場合は、ファンクションの宣言でsql_macro_clauseを指定する必要があります。

sql_macro_clauseの制限

SQL_MACROの注釈は、RESULT_CACHE、PARALLEL_ENABLEおよびPIPELINEDには使用できません。DETERMINISTICプロパティは指定できませんが、SQLマクロは常に暗黙で決定的になります。

SQLマクロ・ファンクションには、VARCHAR2、CHARまたはCLOBの戻り型が必要です。

例

例13-22 Budget: 表式でのSQLマクロの使用

この例は、表式で使用するbudgetというSQLマクロを示しています。このマクロは、特定の役職の従業員に対する部署ごとの合計給与を返します。

```
CREATE FUNCTION budget(job VARCHAR2) RETURN VARCHAR2 SQL_MACRO IS
BEGIN
   RETURN q'{SELECT deptno, SUM(sal) budget
             FROM scott.emp
             WHERE job = budget.job
             GROUP BY deptno}';
END;
/
```

この問合せは、表式で使用されるSQLマクロbudgetを表示します。

```
SELECT * FROM budget('MANAGER');
```

結果:

```

DEPTNO     BUDGET
---------- ----------
        20       2975
        30       2850
        10       2450
```

例13-23 Take: 多相ビューでの表マクロの使用

この例では、表マクロを作成します。このマクロは、表tから最初のn行を返します。

```

CREATE FUNCTION take (n NUMBER, t DBMS_TF.TABLE_T)
                      RETURN VARCHAR2 SQL_MACRO IS
BEGIN
   RETURN 'SELECT * FROM t FETCH FIRST take.n ROWS ONLY';
END;
/
```

この問合せでは、表部門の最初の2行が戻されます。

```
SELECT * FROM take(2, dept);
```

結果:

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
```

SQLマクロは、バインド変数およびより複雑なSQL構文とともに使用できます。

```
VAR row_count NUMBER;
EXEC :row_count := 5
WITH t AS (SELECT * FROM emp NATURAL JOIN dept ORDER BY ename)
SELECT ename, dname FROM take(:row_count, t);
```

結果:

```
ENAME      DNAME
---------- --------------
ADAMS      RESEARCH
ALLEN      SALES
BLAKE      SALES
CLARK      ACCOUNTING
FORD       RESEARCH
```

関連トピック

- 多相テーブル・ファンクションの概要
- PTF実装パッケージの指定方法とDBMS_TFユーティリティの使用方法の詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』

13.65 %TYPE属性

%TYPE属性を使用すると、(データ型がわからなくても)事前に宣言されている変数または列と同じデータ型の定数、変数、コレクション要素、レコード・フィールドまたはサブプログラム・パラメータを宣言できます。

参照された文字の列にUSING_NLS_COMP以外の照合がある場合、%TYPE属性は使用できません。

%TYPEを使用して宣言された項目は参照元項目で、事前に宣言された項目は参照先項目です。

参照元項目は、参照先項目から次のものを継承します。

- データ型およびサイズ
- 制約(参照先項目が列である場合を除く)

参照元項目は、参照先項目の初期値は継承しません。

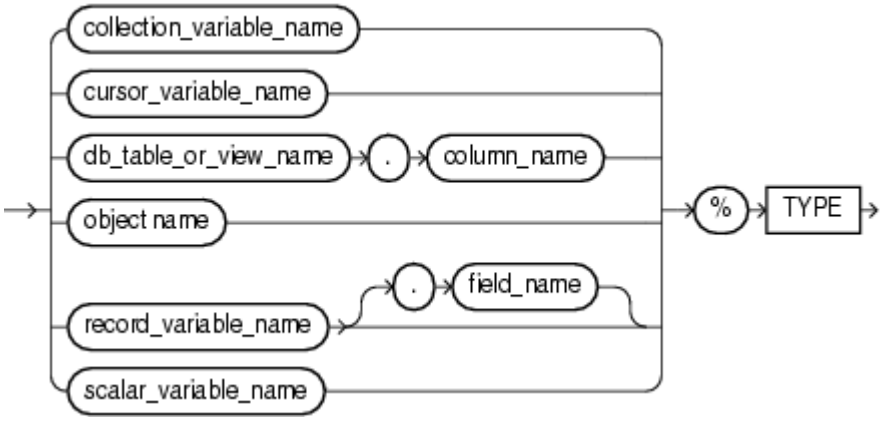
参照先項目の宣言が変更されると、それに応じて参照元項目の宣言も変更されます。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

type_attribute ::=



セマンティクス

type_attribute

collection_variable_name

コレクション変数の名前。

collection_variable_nameの制限

宣言と異なり、collection_variableを連想配列変数にすることはできません。

cursor_variable_name

カーソル変数の名前。

db_table_or_view_name

宣言が処理されるときにアクセスできるデータベースの表またはビューの名前。

column_name

db_table_or_viewの列の名前。

object_name

ADTのインスタンスの名前。

record_variable_name

レコード変数の名前。

field_name

record_variableのフィールドの名前。

scalar_variable_name

スカラー変数の名前。

例

- 例2-15、列と同じ型の変数の宣言
- 例2-16、別の変数と同じ型の変数の宣言

関連トピック

この章:

- 定数の宣言
- %ROWTYPE属性
- スカラー変数の宣言

他の章:

- データ・バインドされた照合について
- %TYPE属性を使用した項目の宣言

13.66 UDFプラグマ

UDFプラグマは、PL/SQLユニットが主にSQL文で使用され、パフォーマンスを向上させる可能性があるユーザー定義ファンクションであることをコンパイラに指示します。

構文

udf_pragma ::=

13.67 UPDATE文の拡張機能

PL/SQLでは、SQLのUPDATE文のupdate_set_clauseおよびwhere_clauseが次のように拡張されています。

- update_set_clauseでレコードを指定できます。選択した行ごとに、UPDATEによって、対応するレコード・フィールドの値で各列が更新されます。
- where_clauseでCURRENT OF句を指定できます。この句を指定すると、UPDATE文は、指定したカーソルの現在の行に制限されます。

関連項目:

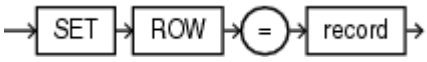
SQLのUPDATE文の構文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

ここでのトピック

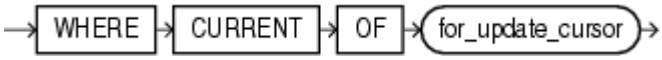
- 構文
- セマンティクス
- 例
- 関連トピック

構文

update_set_clause ::=



where_clause ::=



セマンティクス

update_set_clause

record

dml_table_expression_clauseによって記述される項目の行を表すレコード変数の名前。つまり、レコードには、行のすべての列に対して、互換性のあるデータ型のフィールドが必要です。列にNOT NULL制約が含まれている場合、対応するフィールドにNULL値を含めることはできません。

where_clause

for_update_cursor

FOR UPDATEカーソルの名前。つまり、FOR SELECT UPDATE文に関連付けられた明示カーソルのことです。

例

- 例5-55、レコードを使用した行の更新

関連トピック

この章:

- 「明示カーソルの宣言および定義」
- レコード変数の宣言
- %ROWTYPE属性

他の章:

- 「レコードを使用した行の更新」
- 「レコードの挿入/更新に関する制限」
- 「SELECT FOR UPDATEカーソルとFOR UPDATEカーソル」

13.68 WHILE LOOP文

WHILE LOOP文は、条件がTRUEの場合に1つ以上の文を実行します。条件がFALSEまたはNULLになった場合、ループの内側の文によって制御がループの外側に移されたかまたは例外が呼び出された場合に、WHILE LOOP文は終了します。

ここでのトピック

- 構文
- セマンティクス
- 例
- 関連トピック

構文

while_loop_statement ::=

(boolean_expression ::= , statement ::=)

セマンティクス

while_loop_statement

boolean_expression

値がTRUE、FALSEまたはNULLの式です。

boolean_expressionは、ループの各反復の先頭で評価されます。値がTRUEの場合は、LOOPの後の文が実行されます。それ以外の場合は、制御がWHILE LOOP文の後の文に移ります。

statement

無限ループが発生しないように、1つ以上の文によってboolean_expressionの値をFALSEまたはNULLに変更するか、制御をループの外側に移すか、または例外を呼び出す必要があります。制御をループの外側に移すことができる文には次のものがあります。

- 「CONTINUE文」(外側のラベル付きループの次の反復に制御を移す場合)
- 「EXIT文」
- 「GOTO文」
- 「RAISE文」

label

while_loop_statementを識別するラベル(「statement ::=」および「ラベル」を参照)。CONTINUE、EXITおよびGOTO文でこのラベルを参照できます。

LOOP文がネストされている場合は特に、ラベルによって読みやすさが向上します。ただし、END LOOP文内のラベルが、同じLOOP文の先頭のラベルと一致している場合のみです(コンパイラではチェックしません)。

例

- 例4-28、WHILE LOOP文

関連トピック

この章:

- 「基本LOOP文」
- CONTINUE文
- 「カーソルFOR LOOP文」
- 「EXIT文」
- 「明示カーソルの宣言および定義」
- 「FETCH文」
- 「FOR LOOP文」
- FORALL文
- OPEN文

他の章:

- 「WHILE LOOP文」

14 ストアドPL/SQLユニット用のSQL文

この章では、ストアドPL/SQLユニットの作成、変更および削除を行うSQL文の使用方法について説明します。

CREATE [OR REPLACE]文

次に示すSQL文は、いずれもPL/SQLユニットをスキーマ・レベルで作成してデータベースに格納します。

- CREATE FUNCTION文
- CREATE LIBRARY文
- CREATE PACKAGE文
- CREATE PACKAGE BODY文
- CREATE PROCEDURE文
- CREATE TRIGGER文
- CREATE TYPE文
- CREATE TYPE BODY文

これらのCREATE文には、それぞれオプションのOR REPLACE句を指定できます。OR REPLACE句は、既存のPL/SQLユニットを再作成する場合(つまり、既存のPL/SQLユニットの宣言または定義の削除または再作成およびそれらに事前に付与されているオブジェクト権限の再付与を行わずに、それらの宣言または定義を変更する場合)に指定します。PL/SQLユニットを再定義すると、そのPL/SQLユニットはデータベースによって再コンパイルされます。

注意:

CREATE OR REPLACE 文は、既存の PL/SQL ユニットを置き換える前に警告を発行しません。

これらのCREATE文は、いずれもPL/SQLブロックでは使用できません。

ALTER文

既存のPL/SQLユニットを(宣言または定義を変更することなく)再作成せずに再コンパイルする場合は、次のいずれかのSQL文を使用します。

- ALTER FUNCTION文
- ALTER LIBRARY文
- ALTER PACKAGE文
- ALTER PROCEDURE文
- ALTER TRIGGER文
- ALTER TYPE文

ALTER文を使用する理由は、次のとおりです。

- 無効になったストアド・ユニットを明示的に再コンパイルすることによって、暗黙的な実行時再コンパイルの必要をなくし、

実行時コンパイル・エラーとパフォーマンスのオーバーヘッドを回避するため。

- 異なるコンパイル・パラメータを使用してストアド・ユニットを再コンパイルするため。
- トリガーを有効または無効にするため。
- そのスキーマ内でスキーマ・オブジェクト型がまだエディション可能ではないストアド・ユニットのEDITIONABLEまたはNONEDITIONABLEプロパティを指定するため。

ALTER TYPE文には、他の用法もあります。

DROP文

既存のPL/SQLユニットをデータベースから削除するには、次のいずれかのSQL文を使用します。

- DROP FUNCTION文
- DROP LIBRARY文
- DROP PACKAGE文
- DROP PROCEDURE文
- DROP TRIGGER文
- DROP TYPE文
- DROP TYPE BODY文

関連トピック

- この章の構文図の読み方については、『Oracle Database SQL言語リファレンス』を参照してください。
- エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。
- コンパイル・パラメータの詳細は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください。

14.1 ALTER FUNCTION文

ALTER FUNCTION文は、スタンドアロン・ファンクションを明示的に再コンパイルします。

明示的に再コンパイルすることによって、実行時に暗黙的に再コンパイルする必要がなくなり、関連する実行時コンパイル・エラーおよびパフォーマンス・オーバーヘッドが発生しなくなります。

ノート:

この文では、既存のファンクションの宣言または定義は変更されません。スタンドアロン・ファンクションを再宣言または再定義するには、「CREATE FUNCTION 文」を OR REPLACE 句とともに使用します。

ここでのトピック

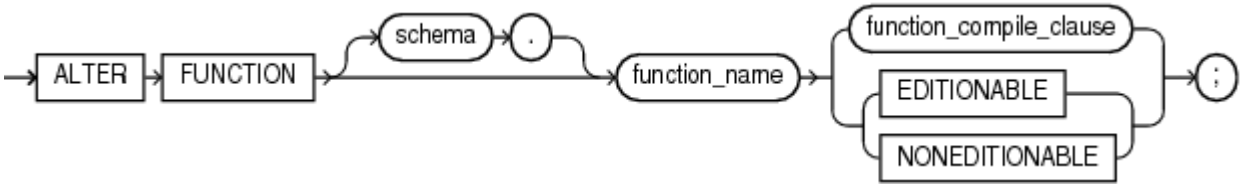
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

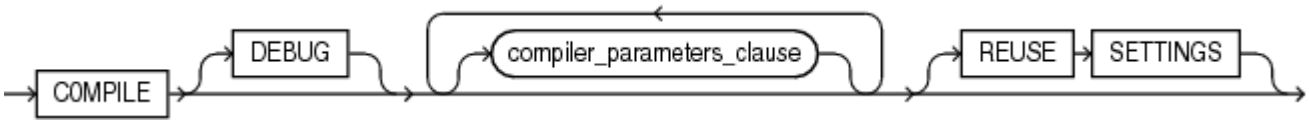
ファンクションがSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、ファンクションが自分のスキーマ内にあるか、ALTER ANY PROCEDUREシステム権限が必要です。

構文

alter_function ::=



function_compile_clause ::=



(compiler_parameters_clause ::=)

セマンティクス

alter_function

schema

ファンクションが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

function_name

再コンパイルするファンクションの名前。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプFUNCTIONに対して後からエディションが有効化された場合に、ファンクションがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

function_compile_clause

有効か無効かに関係なくファンクションが再コンパイルされます。

compile_clauseのセマンティクスを参照してください。

DEFAULT COLLATION句のコンパイル・セマンティクスも参照してください。

例

例14-1 ファンクションの再コンパイル

サンプル・ユーザーoeが所有しているファンクションget_balを明示的に再コンパイルするには、次の文を発行します。

```
ALTER FUNCTION oe.get_bal COMPILE;
```

get_balの再コンパイル中にデータベースでコンパイル・エラーが発生しなかった場合、get_balは有効になります。その後、実行時にデータベースによって再コンパイルされることなく、実行できます。get_balの再コンパイル時にコンパイル・エラーが発生した場合は、データベースによってエラーが戻され、get_balは無効なままになります。

また、データベースによって、get_balに依存するオブジェクトもすべて無効にされます。その後、最初に明示的に再コンパイルせずにこれらのオブジェクトのいずれかを参照すると、データベースによって実行時にそのオブジェクトが暗黙的に再コンパイルされます。

関連トピック

- CREATE FUNCTION文
- DROP FUNCTION文

14.2 ALTER LIBRARY文

ALTER LIBRARY文を使用すると、ライブラリを明示的に再コンパイルできます。

明示的に再コンパイルすることによって、実行時に暗黙的に再コンパイルする必要がなくなり、関連する実行時コンパイル・エラーおよびパフォーマンス・オーバーヘッドが発生しなくなります。

ノート:

この文では、既存のライブラリの宣言または定義は変更されません。ライブラリを再宣言または再定義するには、「CREATE LIBRARY 文」を OR REPLACE 句とともに使用します。

ここでのトピック

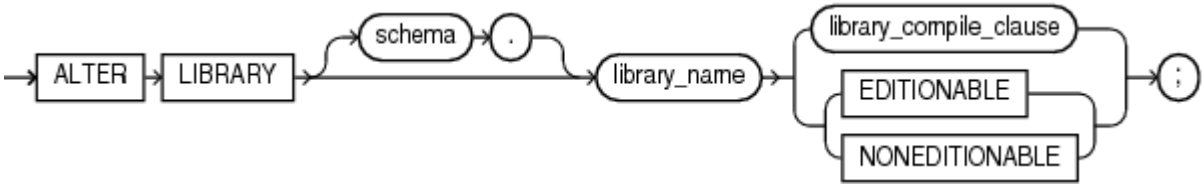
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

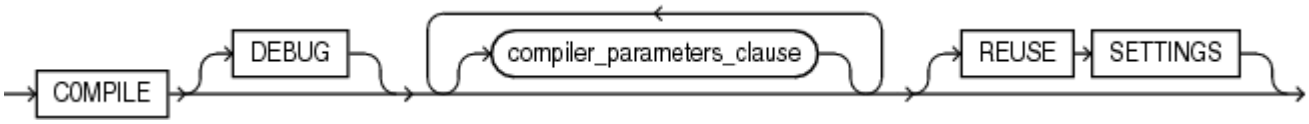
ライブラリがSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、ライブラリが自分のスキーマ内にあるか、ALTER ANY LIBRARYシステム権限が必要です。

構文

alter_library ::=



library_compile_clause ::=



(compiler_parameters_clause ::=)

セマンティクス

alter_library

library_name

再コンパイルされるライブラリの名前。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプLIBRARYに対して後からエディションが有効化された場合に、ライブラリがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

library_compile_clause

ライブラリを再コンパイルします。

compile_clauseおよびcompiler_parameters_clauseのセマンティクスを参照してください。

例

例14-2 ライブラリの再コンパイル

サンプル・ユーザーhrが所有しているライブラリmy_ext_libを明示的に再コンパイルするには、次の文を発行します。

```
ALTER LIBRARY hr.my_ext_lib COMPILE;
```

my_ext_libの再コンパイル中にデータベースでコンパイル・エラーが発生しなかった場合、my_ext_libは有効になります。その後、実行時にデータベースによって再コンパイルされることなく、実行できます。my_ext_libの再コンパイル時にコンパイル・エラーが発生した場合は、データベースによってエラーが戻され、my_ext_libは無効なままになります。

また、データベースによって、my_ext_libに依存するオブジェクトもすべて無効にされます。その後、最初に明示的に再コンパイルせずにこれらのオブジェクトのいずれかを参照すると、データベースによって実行時にそのオブジェクトが暗黙的に再コンパイルされます。

関連トピック

- CREATE LIBRARY文
- DROP LIBRARY文

14.3 ALTER PACKAGE文

ALTER PACKAGE文は、パッケージの仕様部、本体、またはその両方を明示的に再コンパイルします。明示的に再コンパイルすることによって、実行時に暗黙的に再コンパイルする必要がなくなり、関連する実行時コンパイル・エラーおよびパフォーマンス・オーバーヘッドが発生しなくなります。

パッケージ内のすべてのオブジェクトは1つの単位として格納されるため、ALTER PACKAGE文によって、すべてのパッケージ・オブジェクトが再コンパイルされます。ALTER PROCEDURE文またはALTER FUNCTION文を使用して、パッケージ中の一部のプロシージャまたはファンクションを再コンパイルすることはできません。

> ノート:
>
> この文では、既存のパッケージの宣言または定義は変更されません。パッケージを再宣言または再定義するには、CREATE PACKAGE 文を使用するか、または CREATE PACKAGE BODY 文を OR REPLACE 句とともに使用します。

ここでのトピック

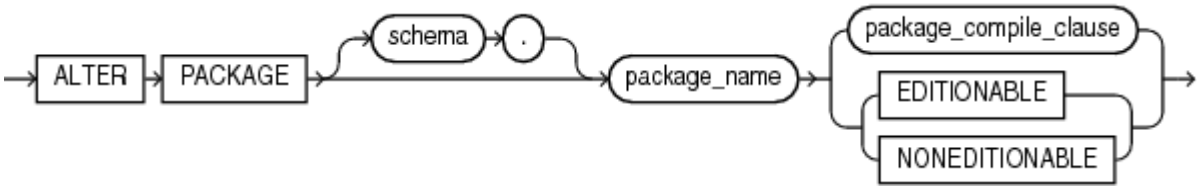
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

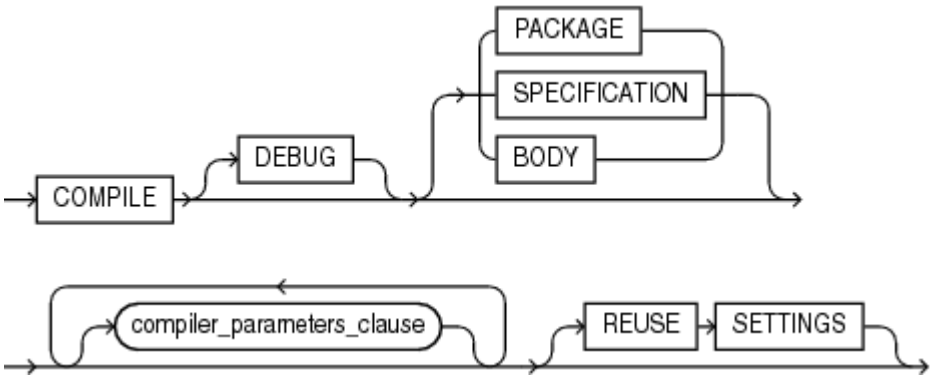
パッケージがSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、パッケージが自分のスキーマ内にあるか、ALTER ANY PROCEDUREシステム権限が必要です。

構文

alter_package ::=



package_compile_clause ::=



(compiler_parameters_clause ::=)

セマンティクス

alter_package

schema

パッケージが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

package_name

再コンパイルするパッケージの名前。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプPACKAGEに対して後からエディションが有効化された場合に、パッケージがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

package_compile_clause

パッケージ仕様部、パッケージ本体、またはその両方が再コンパイルされます。

compile_clauseおよびcompiler_parameters_clauseのセマンティクスを参照してください。

例

例14-3 パッケージの再コンパイル

この文は、hr.emp_mgmtパッケージの仕様部と本体を明示的に再コンパイルします。

このパッケージの作成例は、「CREATE PACKAGE文」を参照してください。

```
ALTER PACKAGE emp_mgmt COMPILE PACKAGE;
```

emp_mgmt仕様部および本体の再コンパイル中にデータベースでコンパイル・エラーが発生しなかった場合、emp_mgmtは有効となります。その後、ユーザーhrは、emp_mgmtの仕様部で宣言されたすべてのパッケージ・オブジェクトを実行時に再コンパイルせずに起動または参照できます。emp_mgmtの再コンパイル時にコンパイル・エラーが発生した場合は、データベースによってエラーが戻され、emp_mgmtは無効なままになります。

また、データベースによって、emp_mgmtに依存するオブジェクトもすべて無効にされます。その後、最初に明示的に再コンパイルせ

ずにこれらのオブジェクトのいずれかを参照すると、データベースによって実行時にそのオブジェクトが暗黙的に再コンパイルされます。

スキーマhr内のemp_mgmtパッケージの本体を再コンパイルするには、次の文を発行します。

```
ALTER PACKAGE hr.emp_mgmt COMPILE BODY;
```

パッケージ本体の再コンパイル中にデータベースでコンパイル・エラーが発生しなかった場合、本体は有効になります。その後、ユーザーhrは、emp_mgmtの仕様部で宣言されたすべてのパッケージ・オブジェクトを実行時に再コンパイルせずに起動または参照できます。本体の再コンパイル時にコンパイル・エラーが発生した場合は、データベースによってエラー・メッセージが戻され、本体は無効なままになります。

この文はemp_mgmtの仕様部ではなく本体を再コンパイルするため、依存オブジェクトはデータベースによって無効にされません。

関連トピック

- CREATE PACKAGE文
- DROP PACKAGE文

14.4 ALTER PROCEDURE文

ALTER PROCEDURE文は、スタンドアロン・プロシージャを明示的に再コンパイルします。

明示的に再コンパイルすることによって、実行時に暗黙的に再コンパイルする必要がなくなり、関連する実行時コンパイル・エラーおよびパフォーマンス・オーバーヘッドが発生しなくなります。

パッケージの一部であるプロシージャを再コンパイルするには、「ALTER PACKAGE文」を使用してパッケージ全体を再コンパイルします。

> ノート:
>
> この文では、既存のプロシージャの宣言または定義は変更されません。スタンドアロン・プロシージャを再宣言または再定義するには、「CREATE PROCEDURE 文」を OR REPLACE 句とともに使用します。

ALTER PROCEDURE文は、ALTER FUNCTION文に非常に似ています。詳細は、「ALTER FUNCTION文」を参照してください。

ここでのトピック

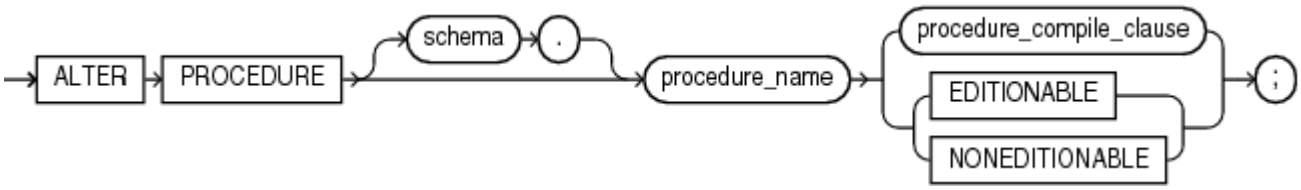
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

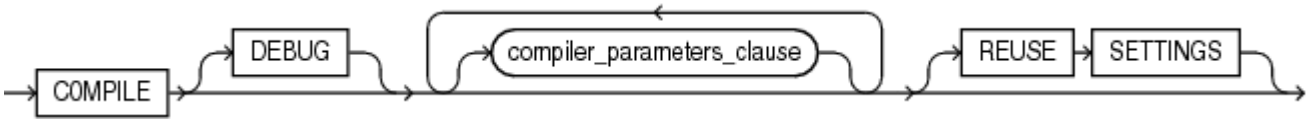
プロシージャがSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、プロシージャが自分のスキーマ内にあるか、ALTER ANY PROCEDUREシステム権限が必要です。

構文

alter_procedure ::=



procedure_compile_clause ::=



(compiler_parameters_clause ::=)

セマンティクス

alter_procedure

schema

プロシージャが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

procedure_name

再コンパイルするプロシージャの名前。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプPROCEDUREに対して後からエディションが有効化された場合に、プロシージャがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

procedure_compile_clause

compile_clauseおよびcompiler_parameters_clauseのセマンティクスを参照してください。

例

例14-4 プロシージャの再コンパイル

ユーザーhrが所有しているプロシージャremove_empを明示的に再コンパイルするには、次の文を発行します。

```
ALTER PROCEDURE hr.remove_emp COMPILE;
```

remove_empの再コンパイル中にデータベースでコンパイル・エラーが発生しなかった場合、remove_empは有効になります。その後、実行時にデータベースによって再コンパイルされることなく、実行できます。remove_empの再コンパイル時にコンパイル・エラーが発生した場合は、データベースによってエラーが戻され、remove_empは無効なままになります。

また、データベースによってすべての依存オブジェクトも無効にされます。これらのオブジェクトには、remove_empを起動するプロシージャ、ファンクションおよびパッケージ本体などがあります。その後、最初に明示的に再コンパイルせずにこれらのオブジェクトのいずれかを参照すると、データベースによって実行時にそのオブジェクトが暗黙的に再コンパイルされます。

関連トピック

- CREATE PROCEDURE文
- DROP PROCEDURE文

14.5 ALTER TRIGGER文

ALTER TRIGGER文は、データベース・トリガーの有効化、無効化、コンパイルまたは名前変更を実行します。

> ノート:
>
> この文では、既存のトリガーの宣言または定義は変更されません。トリガーを再宣言または再定義するには、「CREATE TRIGGER 文」を OR REPLACE 句とともに使用します。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

トリガーがSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、トリガーが自分のスキーマ内にあるか、ALTER ANY TRIGGERシステム権限が必要です。

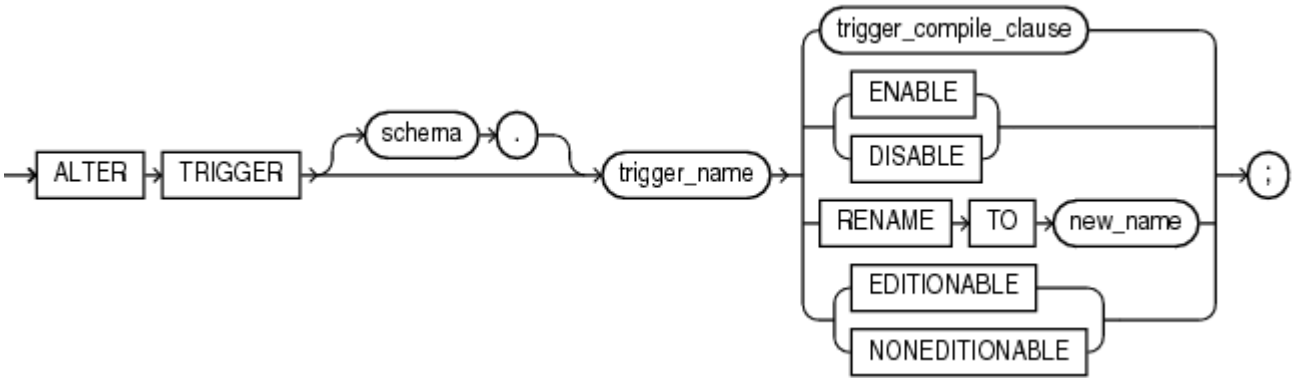
また、DATABASEに対するトリガーを変更するには、ADMINISTER DATABASE TRIGGERシステム権限を所有している必要があります。

関連項目:

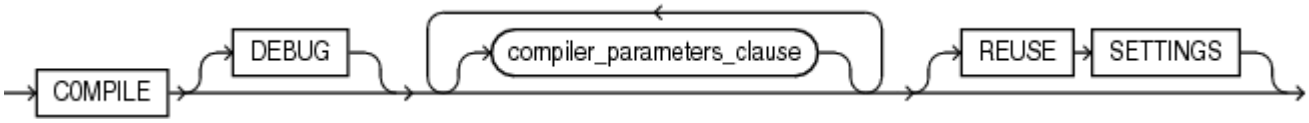
DATABASEトリガーに基づくトリガーの詳細は、「CREATE TRIGGER文」を参照してください

構文

alter_trigger ::=



trigger_compile_clause ::=



(compiler_parameters_clause ::=)

セマンティクス

alter_trigger

schema

トリガーが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

trigger_name

変更するトリガーの名前。

[ENABLE | DISABLE]

トリガーを有効または無効にします。

RENAME TO new_name

状態を変更せずにトリガーの名前を変更します。

トリガーの名前を変更すると、*_SOURCE静的データ・ディクショナリ・ビューに記憶されているトリガーのソースが再構築されます。その結果、トリガー・ソースが変更されていない場合でも、これらのビューのTEXT列のコメントおよび書式設定が変更されることがあります。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプTRIGGERに対して後からエディションが有効化された場合に、トリガーがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

NONEDITIONABLEの制限

crosseditionトリガーに対してNONEDITIONABLEを指定できません。

trigger_compile_clause

有効か無効かに関係なくトリガーが再コンパイルされます。

compile_clauseおよびcompiler_parameters_clauseのセマンティクスを参照してください。

例

例14-5 トリガーの無効化

サンプル・スキーマhrには、employees表に作成されたupdate_job_historyという名前のトリガーがあります。このトリガーは、UPDATE文によって従業員のjob_idが変更されるたびに起動されます。このトリガーによって、従業員のID、最後のジョブの開始日と終了日およびジョブのIDと部門を含む行がjob_history表に挿入されます。

このトリガーは、作成後、データベースによって自動的に有効にされます。その後、次の文を使用してこのトリガーを無効にすることができます。

```
ALTER TRIGGER update_job_history DISABLE;
```

トリガーが無効になっている場合は、UPDATE文によって従業員のジョブが変更されても、データベースによってトリガーは起動されません。

例14-6 トリガーの有効化

トリガーを無効化した後に、この文を使用してそれを有効化できます。

```
ALTER TRIGGER update_job_history ENABLE;
```

トリガーを再度有効にすると、UPDATE文によって従業員のジョブが変更されるたびに、データベースによってトリガーが起動されます。トリガーが無効になっている場合に従業員のジョブが更新されると、別のトランザクションによってjob_idが再度変更されるまで、この従業員のトリガーはデータベースによって自動的には起動されません。

関連トピック

この章:

- CREATE TRIGGER文
- DROP TRIGGER文

他の章:

- トリガーのコンパイル、無効化および再コンパイル
- 「トリガーの有効化および無効化」

14.6 ALTER TYPE文

ALTER TYPE文を使用すると、メンバー属性またはメソッドを追加または削除できます。オブジェクト型の既存のプロパティを変更できます。また、その型のスカラー属性を変更できます。この文を使用すると、新しいオブジェクト・メンバーのサブプログラム仕様を追加することによって、型の仕様部または本体を再コンパイルしたり、オブジェクト型の仕様を変更することができます。

ALTER TYPE文は、「CREATE TYPE文」および「CREATE TYPE BODY文」で作成された型に対して次のいずれかを実行します。

- 型を進化させます。つまり、メンバー属性またはメソッドを追加または削除します。

  型の進化の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。
- オブジェクト・メンバーのサブプログラム仕様部を追加することによって、型の仕様部を変更します。
- 型の仕様部または本体を再コンパイルします。
- 型のバージョンを1に再設定し、進化の対象とみなされないようにします。

ここでのトピック

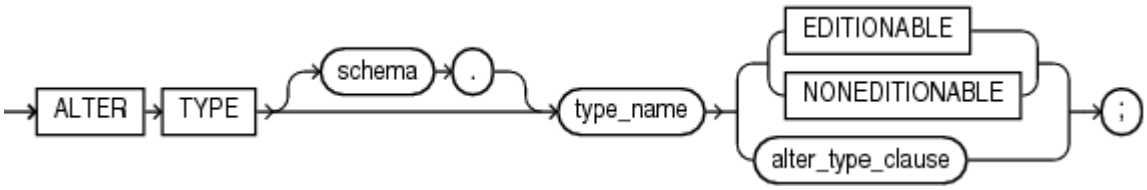
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

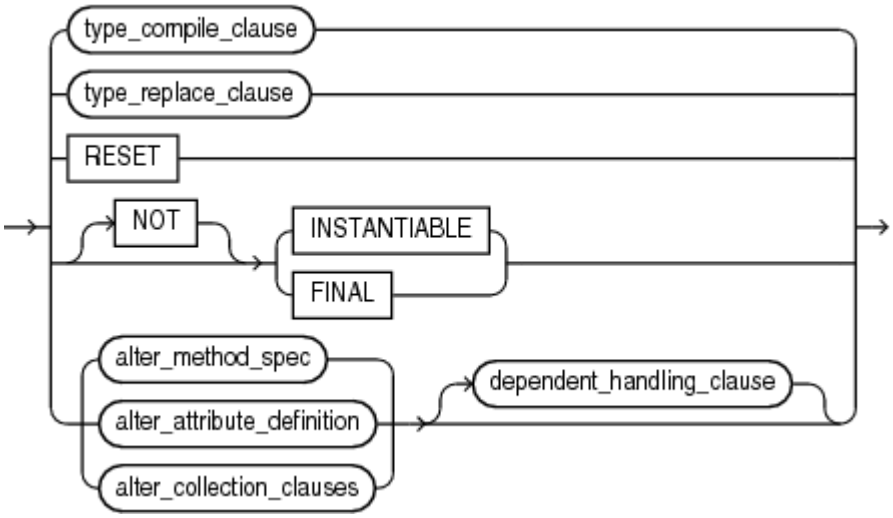
型がSYSスキーマ内にある場合、SYSDBAとして接続する必要があります。そうでない場合は、型が自分のスキーマ内にあり、CREATE TYPEまたはCREATE ANY TYPEシステム権限を所有しているか、またはALTER ANY TYPEシステム権限を所有している必要があります。

構文

alter_type ::=

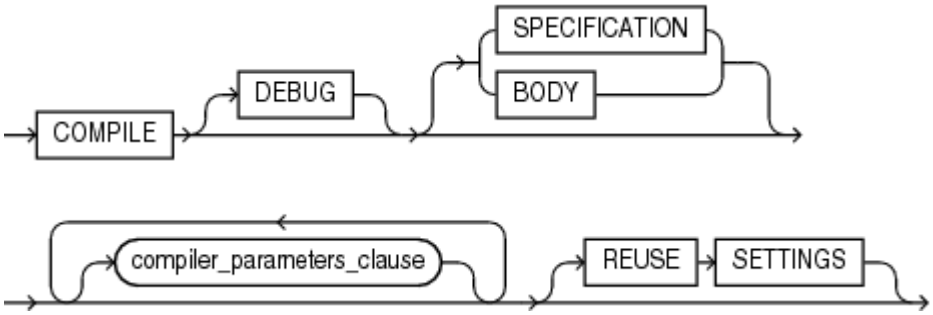


alter_type_clause ::=



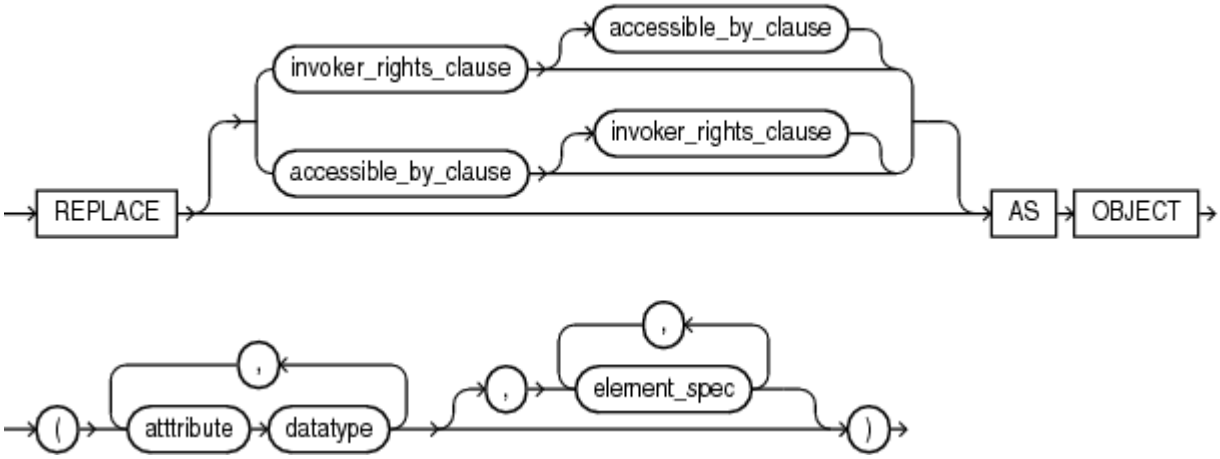
(type_compile_clause ::=, type_replace_clause ::=, alter_attribute_definition ::=, alter_method_spec ::=, alter_collections_clauses::=, dependent_handling_clause ::=)

type_compile_clause ::=



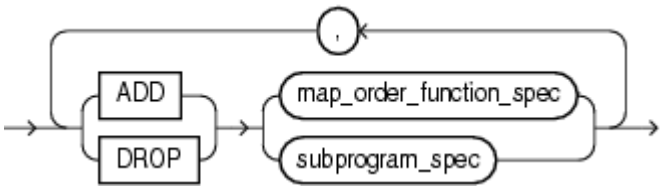
(compiler_parameters_clause ::=)

type_replace_clause ::=



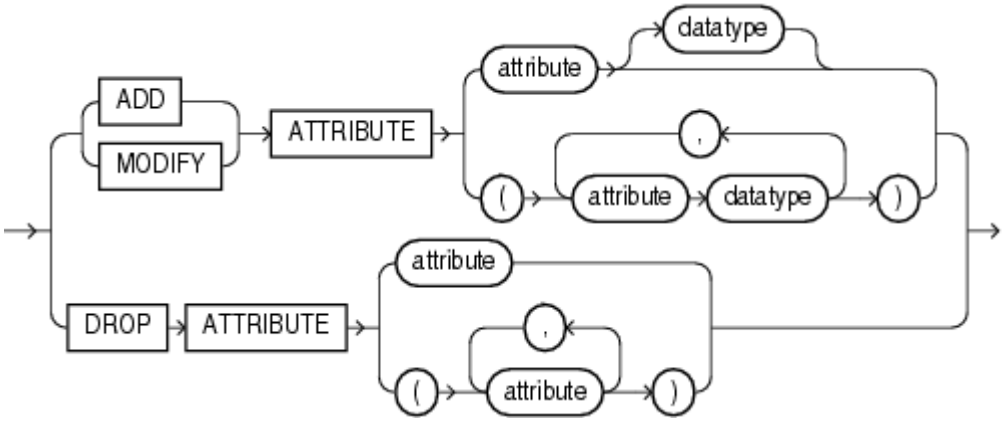
(accessible_by_clause ::=, invoker_rights_clause ::=, element_spec ::=)

alter_method_spec ::=

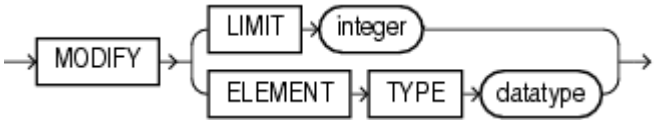


(map_order_function_spec ::=, subprogram_spec ::=)

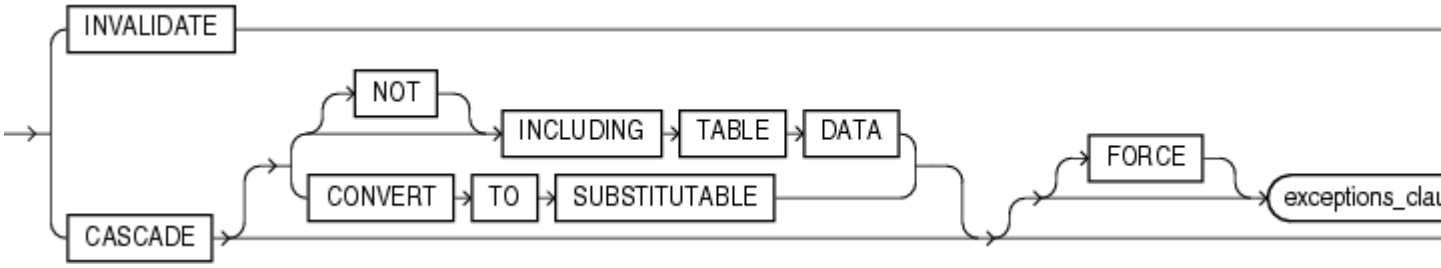
alter_attribute_definition ::=



alter_collections_clauses::=



dependent_handling_clause ::=



exceptions_clause::=

セマンティクス

alter_type

schema

型が含まれているスキーマの名前。デフォルト: 自分のスキーマ。

type_name

ADT、VARRAY型、またはネストした表型の名前。

type_nameの制限

エディション化したADTを進化させることはできません。

次のいずれかに該当する場合は、ORA-22348が発生してALTER TYPE文は失敗します。

- typeがエディションされたADTであり、ALTER TYPE文にtype_compile_clauseがない。

  (ALTER TYPE文は、エディションされたオブジェクト型の再コンパイルには使用できますが、他の目的には使用できませ

ん。)

- typeに依存するエディションされたADTが存在し、ALTER TYPE文にCASCADE句がある。

エディションされたオブジェクトは、エディション可能なオブジェクト型を持つスキーマ・オブジェクトであり、エディションが有効になっているユーザーによって作成されたものです。

{ EDITIONABLE | NONEDITIONABLE }

schemaでスキーマ・オブジェクト・タイプTYPEに対して後からエディションが有効化された場合に、型がエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの変更の詳細は、『Oracle Database開発ガイド』を参照してください。

alter_type_clause

RESET

この型のバージョンを1に再設定し、進化の対象とみなされないようにします。

> ノート:
>
> この型のバージョンを 1 に再設定すると、この型に依存するすべてのものが無効になります。

RESETは、所有者に対してエディションが有効にならないようにする進化したADT向けです。ユーザーに対するエディションの有効化の詳細は、『Oracle Database開発ガイド』を参照してください。

ADTのバージョン番号を確認するには、静的データ・ディクショナリ・ビュー*_TYPE_VERSIONSからVERSION#を選択します。たとえば:

```
SELECT Version#
FROM DBA_TYPE_VERSIONS
WHERE Owner = schema
AND Name = 'type_name'
AND Type = 'TYPE'
```

進化したADTでは、前述の問合せによってバージョン番号が異なる複数の行が戻されます。RESETは、バージョン番号が最大バージョン番号より小さい行をすべて削除して、その残りの行のバージョン番号を1に再設定します。

RESETの制限

型に表依存性がある場合は(直接か間接かにかかわらず)、RESETは指定できません。

[NOT] INSTANTIABLE

この型のオブジェクト・インスタンスを構成できる場合は、INSTANTIABLEを指定します。

この型に対してコンストラクタ(デフォルトまたはユーザー定義)が存在しない場合は、NOT INSTANTIABLEを指定します。インスタンス化ができないメソッドを持つ型、および(継承、またはこの句で指定された)属性を持たない型には、これらのキーワードを指定する必要があります。

NOT INSTANTIABLEの制限

ユーザー定義型に表依存性がある場合は、そのユーザー定義型をINSTANTIABLEからNOT INSTANTIABLEには変更できません。

[NOT] FINAL

この型に対してサブタイプをさらに作成できないようにする場合は、FINALを指定します。

この型に対してサブタイプをさらに作成できるようにする場合は、NOT FINALを指定します。

プロパティをFINALからNOT FINALに、またはその逆に変更する場合は、"dependent_handling_clause"のCASCADE句を指定して、依存する列および表のデータを変換する必要があります。具体的には次のとおりです。

- 型をNOT FINALからFINALに変更する場合は、CASCADE [INCLUDING TABLE DATA]を指定する必要があります。CASCADE NOT INCLUDING TABLE DATAを指定してデータ変換を遅延させることはできません。
- 型をFINALからNOT FINALに変更する場合は、次の操作を実行します。
  - その型の置換可能な表と列を作成する場合は、CASCADE INCLUDING TABLE DATAを指定します。ただし、既存の依存する表および列の置換可能性を考慮しない場合です。

    既存のすべての依存する列および表がNOT SUBSTITUTABLE AT ALL LEVELSとしてマークされるため、変更した型のサブタイプ・インスタンスをこれらの既存の列と表に挿入することはできません。
  - その型の置換可能な表と列を作成し、既存の依存する表および列に変更された型のサブタイプ・インスタンスを格納する場合は、CASCADE CONVERT TO SUBSTITUTABLEを指定します。

    明示的にNOT SUBSTITUTABLE AT ALL LEVELSとマークされたものを除いて、既存のすべての依存する列と表がSUBSTITUTABLE AT ALL LEVELSとしてマークされます。

    関連項目:

    ADTの進化の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

FINALの制限

型にサブタイプがある場合は、ユーザー定義型をNOT FINALからFINALには変更できません。

type_compile_clause

(デフォルト)型仕様部および型本体を再コンパイルします。

compile_clauseおよびcompiler_parameters_clauseのセマンティクスを参照してください。

type_replace_clause

Oracle Database 12cリリース2 (12.2)以降、type_replace_clauseは非推奨となりました。かわりにalter_method_spec句を使用してください。または、CREATE OR REPLACE TYPE文を使用して型を再作成することもできます。

メンバー・サブプログラム仕様部を追加します。

type_replace_clauseの制限

この句は、ADTに対してのみ有効です。ネストされた表やVARRAYには使用できません。

attribute

オブジェクト属性の名前。属性は、オブジェクトの構造を形成する、名前および型指定子を持つデータ項目です。

element_spec

再定義されたオブジェクトの要素を指定します。

alter_method_spec

型にメソッドを追加するか、型からメソッドを削除します。その型に依存するすべてのファンクション索引がデータベースによって無効にされます。

1つのALTER TYPE文で複数のメソッドを追加または削除できますが、各メソッドを参照できるのは1回のみです。

ADD

メソッドを追加する場合は、その名前が型階層内の既存の属性と競合しないようにする必要があります。

DROP

メソッドを削除すると、そのメソッドがターゲット型から削除されます。

DROPの制限

スーパータイプから継承されたメソッドをサブタイプから削除することはできません。かわりに、メソッドをスーパータイプから削除する必要があります。

alter_attribute_definition

ADTの属性を追加、削除または変更します。1つのALTER TYPE文で複数のメンバー属性またはメソッドを追加、削除または変更できますが、各属性またはメソッドを参照できるのは1回のみです。

ADD ATTRIBUTE

属性の名前は、型階層内の既存の属性またはメソッドと競合しないようにする必要があります。属性は、データベースによって、ローカルに定義された属性リストの最後に追加されます。

属性をスーパータイプに追加すると、すべてのサブタイプに継承されます。サブタイプでは、継承された属性は宣言された属性より常に優先されます。そのため、属性をスーパータイプに追加した後で、暗黙的に変更されたサブタイプのマッピングの更新が必要な場合があります。

DROP ATTRIBUTE

属性を型から削除すると、削除した属性に対応する列に加えて、削除した属性を参照している索引、統計および制約も削除されます。

削除する属性のデータ型を指定する必要はありません。

DROP ATTRIBUTEの制限

- スーパータイプから継承された属性は削除できません。かわりに、属性をスーパータイプから削除する必要があります。

- パーティション化キー、サブパーティション化キーまたはクラスタ・キーの一部となっている属性は削除できません。

> 注意:
>
> INVALIDATE オプションを使用する場合、コンパイラは依存をチェックしないため、このルールは施行されません。ただし、そのような属性を削除すると、表は使用できない状態のままとなります。

- オブジェクト表の主キー・ベースのオブジェクト識別子または索引構成表の主キーの属性は削除できません。
- ルート型のすべての属性を削除することはできません。かわりに、ルート型を削除する必要があります。ただし、サブタイプのローカルに宣言されたすべての属性を削除することはできます。

MODIFY ATTRIBUTE

既存のスカラー属性のデータ型を変更します。たとえば、VARCHAR2またはRAW属性の長さを増やしたり、数値属性の精度または位取りを増やすことができます。

MODIFY ATTRIBUTEの制限

ファンクション・ベースの索引内、ドメイン索引内、またはクラスタ・キー内で参照される属性のサイズを拡張することはできません。

alter_collection_clauses

この句は、コレクション型に対してのみ有効です。

MODIFY LIMIT integer

VARRAYの要素の数を増やします。この句は、ネストした表に対しては無効です。VARRAYの要素の現在の最大数より大きい整数を指定します。

MODIFY ELEMENT TYPE datatype

VARRAYまたはネストした表のスカラー・データ型の精度、サイズまたは長さを増やします。この句は、ADTのコレクションに対しては無効です。

- NUMBERのコレクションの場合は、精度または位取りを増やすことができます。
- RAWのコレクションの場合は、最大サイズを増やすことができます。
- VARCHAR2またはNVARCHAR2のコレクションの場合は、最大長を増やすことができます。

dependent_handling_clause

変更された型に依存するオブジェクトをデータベースで処理する方法を指定します。この句を省略すると、型に依存する型または表がある場合にALTER TYPE文が終了します。

INVALIDATE

チェック・メカニズムを使用しないで、すべての依存オブジェクトを無効にします。Oracle Database 12cリリース2 (12.2)以降、INVALIDATEコマンドは非推奨となりました。かわりにCASCADE句を使用することをお薦めします。

注意:

データベースによって型の変更は検証されないため、この句は注意して使用してください。たとえば、パーティション化またはクラスタ・キーである属性を削除すると、表は使用できなくなります。

CASCADE

型の変更を依存する型および表に伝播します。FORCEも指定しないかぎり、依存する型または表でエラーが検出された場合は文が終了します。

型のプロパティをFINALとNOT FINALの間で変更する場合は、この句を指定して、依存する列および表のデータを変換する必要があります。

INCLUDING TABLE DATA

(デフォルト)すべてのユーザー定義列に格納されているデータを最新バージョンの列型に変換します。

ノート:

列データが Oracle Database バージョン 8.0 のイメージ・フォーマットの場合は、この句を指定する必要があります。この句は、型のプロパティを FINAL と NOT FINAL の間で変更する場合も必須です。

- 列型に追加された属性ごとに、データベースによってデータに属性が追加され、NULLに初期化されます。
- 参照される型から削除された属性ごとに、対応する属性データが表の各行から削除されます。

INCLUDING TABLE DATAを指定する場合は、表データを含むすべての表領域が読取り/書込みモードである必要があります。

NOT INCLUDING TABLE DATAを指定すると、データベースによって列のメタデータがアップグレードされ、その型に対する変更が反映されますが、このALTER TYPE文の一部として依存する列はスキャンされず、データは更新されません。ただし、依存する列データはアクセス可能なままであり、後続のデータの問合せ結果には型の変更が反映されます。

CONVERT TO SUBSTITUTABLE

FINALからNOT FINALへの型の変更、その型の置換可能な表と列の作成、および既存の依存する表と列への変更した型のサブタイプ・インスタンスの格納も行う場合は、この句を指定します。

exceptions_clause

FORCE

依存する表および索引からのエラーを無視し、指定した例外表にすべてのエラーを記録する場合は、FORCEを指定します。DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLEプロシージャを実行して、例外表が作成されている必要があります。

例

これらの例で参照される型の作成例は、「CREATE TYPE文」を参照してください。

例14-7 メンバー・ファンクションの追加

この例では、ADT data_typ1を使用します。

メソッドがdata_typ1に追加され、その型本体がそれに対応するように変更されます。日付書式は、oe.ordersサンプル表のorder_date列と一致しています。

```
ALTER TYPE data_typ1
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;
CREATE OR REPLACE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
  BEGIN
  RETURN (year + invent);
  END;
     MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
     BEGIN
       IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
         RETURN 'FIRST';
       ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
         RETURN 'SECOND';
       ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
         RETURN 'THIRD';
       ELSE
         RETURN 'FOURTH';
       END IF;
     END;
   END;
/
```

例14-8 コレクション属性の追加

この例では、author属性をtext表のtextdoc_tabオブジェクト列に追加します。

```
CREATE TABLE text (
   doc_id       NUMBER,
   description  textdoc_tab)
   NESTED TABLE description STORE AS text_store;
ALTER TYPE textdoc_typ
   ADD ATTRIBUTE (author VARCHAR2) CASCADE;
```

textdoc_tab表とtext表の両方がtextdoc_typ型に依存しているため、CASCADEキーワードは必須です。

例14-9 コレクション型の要素の数の増加

この例では、VARRAYphone_list_typ_demo内の要素の最大数を増やします。

```
ALTER TYPE phone_list_typ_demo
  MODIFY LIMIT 10 CASCADE;
```

例14-10 コレクション型の長さの増加

この例では、VARRAY要素型phone_list_typの長さを長くします。

```
ALTER TYPE phone_list_typ
  MODIFY ELEMENT TYPE VARCHAR(64) CASCADE;
```

例14-11 型の再コンパイル

この例では、hrスキーマ内の型cust_address_typを再コンパイルします。

```
ALTER TYPE cust_address_typ2 COMPILE;
```

例14-12 型仕様部の再コンパイル

この例では、link2の型仕様部をコンパイルします。

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);
/
CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);
/
CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS
      BEGIN
         dbms_output.put_line(c1);
         RETURN c1;
      END;
   END;
/
```

次の例では、link2の属性であるlink1が変更されるため、link2の仕様部と本体の両方が無効になります。

```
ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;
```

仕様部と本体を別々の文で再コンパイルすることによって、型を再コンパイルする必要があります。

```
ALTER TYPE link2 COMPILE SPECIFICATION;

ALTER TYPE link2 COMPILE BODY;
```

また、仕様部と本体を同時にコンパイルすることもできます。

```
ALTER TYPE link2 COMPILE;
```

例14-13 ADTの進化と再設定

この例では、スキーマUsr内にADTを作成し、そのADTを進化させてからUsrに対してエディションを有効にしようとしますが、失敗します。

その後、ADTのバージョンを1に再設定すると、Usrに対するエディションが正常に有効になります。この例では、新しく作成し、進化させ、再設定したADTのバージョン番号を表示するために、静的データ・ディクショナリ・ビューDBA_TYPE_VERSIONSを使用します。

```
-- Create ADT in schema Usr:
create type Usr.My_ADT authid Definer is object(a1 number)
-- Show version number of ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
```

```
/
```

結果:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)
1 row selected.
-- Evolve ADT:
alter type Usr.My_ADT add attribute (a2 number)
/
-- Show version number of evolved ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

結果:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)
2
type     My_ADT authid Definer is object(a1 number)
2
 alter type     My_ADT add attribute (a2 number)
3 rows selected.
-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

結果:

```
alter user Usr enable editions
*
ERROR at line 1:
ORA-38820: user has evolved object type
-- Reset version of ADT to 1:
alter type Usr.My_ADT reset
/
-- Show version number of reset ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

結果:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)
1
```

```
 alter type     My_ADT add attribute (a2 number)
2 rows selected.
-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

結果:

```
User altered.
```

関連トピック

この章:

- CREATE TYPE文
- 「CREATE TYPE BODY文」
- DROP TYPE文

その他のドキュメント:

- エディションの詳細は、『Oracle Database開発ガイド』を参照してください。
- プラグマの詳細は、『Oracle Database開発ガイド』を参照してください。
- 型属性の変更時に表データを含めない場合の影響の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

14.7 CREATE FUNCTION文

CREATE FUNCTION文は、スタンドアロン・ファンクションまたはコール仕様を作成するか、または置き換えます。

スタンドアロン・ファンクションは、データベースに格納されるファンクション(単一の値を戻すサブプログラム)です。

> ノート:
>
> CREATE FUNCTION 文で作成するスタンドアロン・ファンクションは、PL/SQL ブロックまたはパッケージで宣言および定義するファンクションとは異なります。詳細は、ファンクションの宣言および定義および CREATE PACKAGE 文を参照してください。

コール仕様は、PL/SQLから起動できるようにJavaメソッドまたはCファンクションを宣言します。このようなメソッドまたはサブプログラムは、SQLのCALL文を使用して起動することもできます。コール仕様は、起動時に起動するJavaメソッドまたは共有ライブラリ内の名前付きファンクションをデータベースに指示します。また、引数および戻り値に対して実行する型変換もデータベースに指示します。

> ノート:
>
> ストアド・ファンクションは、副作用を制御する特定の規則に従っている場合にのみ、SQL 文からコールできます。「サブプログラムの副作用」を参照してください。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

自分のスキーマ内にスタンドアロン・ファンクションを作成するか、または自分のスキーマ内のスタンドアロン・ファンクションを置き換えるには、CREATE PROCEDUREシステム権限が必要です。

別のユーザーのスキーマ内にスタンドアロン・ファンクションを作成するか、または別のユーザーのスキーマ内のスタンドアロン・ファンクションを置き換えるには、CREATE ANY PROCEDUREシステム権限が必要です。

コール仕様を起動するには、追加の権限が必要になる場合があります。たとえば、Cコール仕様の場合には、Cライブラリに対するEXECUTE権限が必要です。

CREATE FUNCTION文をOracleプリコンパイラ・プログラム内に埋め込むには、キーワードEND-EXECの後に特定の言語の埋込みSQL文の終了記号を付けて文を終了する必要があります。

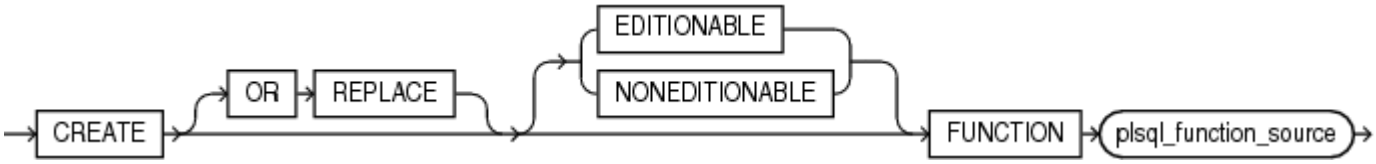
関連項目:

前述の前提条件の詳細は、次のマニュアルを参照してください。

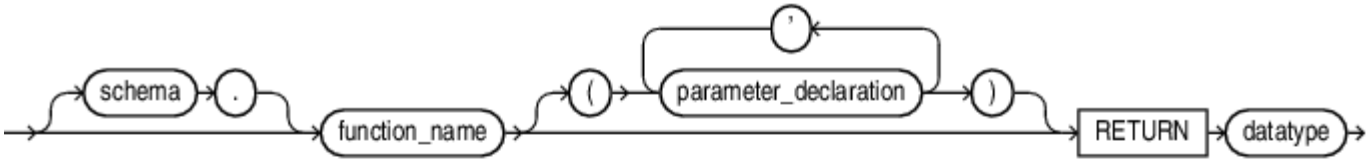
- 『Oracle Database開発ガイド』
- 『Oracle Database Java開発者ガイド』

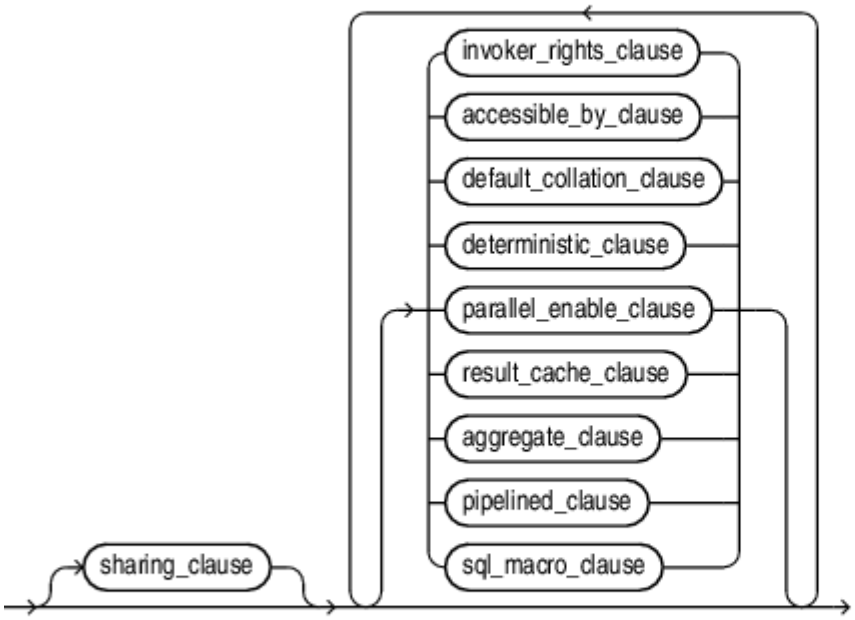
構文

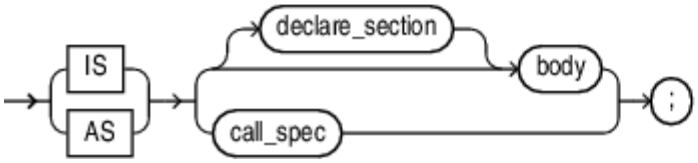
create_function ::=



plsql_function_source ::=







(sharing_clause ::= , invoker_rights_clause ::= , accessible_by_clause ::= , default_collation_clause ::= , deterministic_clause ::= , parallel_enable_clause ::= , result_cache_clause ::= , aggregate_clause ::= , pipelined_clause ::=, sql_macro_clause ::= ,

body ::= , call_spec ::= , datatype ::= , declare_section ::= , parameter_declaration ::=)

セマンティクス

create_function

OR REPLACE

ファンクションが存在する場合は、ファンクションを再作成し、再コンパイルします。

再定義する前のファンクションに対する権限を付与されていたユーザーは、権限を再付与される必要なく、ファンクションにアクセスできます。

ファンクション索引がファンクションに依存している場合は、データベースによってその索引にDISABLEDのマークが付けられます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプFUNCTIONに対してエディションが有効になっている場合に、ファンクションがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

plsql_function_source

schema

ファンクションが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

function_name

作成するファンクションの名前。

ノート:

SQL*Module によって生成されたスタブを使用してストアド・サブプログラムを起動する場合、ストアド・サブプログラム名は、起動側ホストの 3GL 言語(Ada や C など)の有効な識別子である必要もあります。

RETURN datatype

datatypeには、ファンクションの戻り値のデータ型を指定します。戻り値は、PL/SQLでサポートされているすべてのデータ型にすることができます。

ノート:

Oracle SQL では、BOOLEAN パラメータまたは戻り値を持つファンクションの起動はサポートされていません。したがって、ユーザー定義ファンクションを起動する SQL 文の場合は、数値(0 または 1)または文字列(TRUE または FALSE)を戻すように設計する必要があります。

データ型で長さ、精度または位取りを指定することはできません。戻り値の長さ、精度または位取りは、ファンクションがコールされ

た環境からデータベースによって導出されます。

戻り型がANYDATASETであり、問合せのFROM句でファンクションを使用する場合は、PIPELINED句を指定し、ファンクションの実装タイプの一部としてdescribeメソッド(ODCITableDescribe)を定義する必要もあります。

このデータ型は、NOT NULLなどでは制約することができません。

body

ファンクションの必須の実行部、およびオプションでファンクションの例外処理部。

declare_section

ファンクションのオプションの宣言部。宣言は、ファンクションに対してローカルであり、bodyで参照でき、ファンクションが実行を完了すると消滅します。

例

例14-14 ファンクションの作成

この文は、サンプル表oe.ordersにファンクションget_balを作成します。

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT order_total
      INTO acc_bal
      FROM orders
      WHERE customer_id = acc_no;
      RETURN(acc_bal);
    END;
/
```

get_balファンクションは、指定した口座の残高を戻します。

このファンクションを起動する場合は、引数acc_no(残高を確認する口座の番号)を指定する必要があります。acc_noのデータ型はNUMBERです。

このファンクションは口座の残高を戻します。CREATE FUNCTION文にRETURN句を指定すると、戻り値のデータ型がNUMBERになります。

このファンクションは、SELECT文を使用して、引数acc_noで識別される行のbalance列をorders表で選択します。次に、RETURN文を使用して、ファンクションがコールされた環境にこの値を戻します。

前述の例で作成したファンクションは、SQL文で使用できます。たとえば:

```
SELECT get_bal(165) FROM DUAL;
GET_BAL(165)
------------
        2519
```

例14-15 集計ファンクションの作成

次の文は、一連の数値を集計するためのSecondMaxという集権ファンクションを作成します。ADTのSecondMaxImplサブプログラムに、ODCIAggregateサブプログラムの実装が含まれていることを想定しています。

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
    PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

関連項目:

SecondMaxImplの型および型本体の実装の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

このような集計ファンクションは、サンプル表hr.employeesを問い合せる次の文のような問合せで使用します。

```
SELECT SecondMax(salary) "SecondMax", department_id
      FROM employees
      GROUP BY department_id
      HAVING SecondMax(salary) > 9000
      ORDER BY "SecondMax", department_id;
SecondMax DEPARTMENT_ID
--------- -------------
      9450           100
  13670.74            50
     14175            80
   18742.5            90
```

例14-16 ファンクションでのパッケージ・プロシージャ

この文は、DBMS_LOB.GETLENGTHプロシージャを使用してCLOB列の長さを戻すファンクションを作成します。

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

関連トピック

この章:

- ALTER FUNCTION文
- CREATE PROCEDURE文
- DROP FUNCTION文

他の章:

- 多相テーブル・ファンクションの概要
- PL/SQLブロック内でのファンクションの作成の詳細は、「ファンクションの宣言および定義」を参照してください
- 仮パラメータの宣言
- PL/SQLサブプログラム

その他のドキュメント:

- CALL文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- SQL文からコールされるユーザー定義ファンクションの制限の詳細は、『Oracle Database開発ガイド』を参照してください。
- コール仕様の詳細は、『Oracle Database開発ガイド』を参照してください。
- ODCITableDescribeファンクションの定義の詳細は、『Oracle Databaseデータ・カートリッジ開発者ガイド』を参照してください。

14.8 CREATE LIBRARY文

CREATE LIBRARY文を使用すると、ライブラリ、つまりオペレーティング・システムの共有ライブラリに関連付けられたスキーマ・オブジェクトが作成されます。

> ノート:
>
> CREATE LIBRARY 文は、共有ライブラリおよび動的リンクをサポートするプラットフォーム上でのみ有効です。

オペレーティング・システムの共有ライブラリ(DLL)の作成方法は、『Oracle Database開発ガイド』を参照してください。

このライブラリ・スキーマ・オブジェクトの名前は、SQLおよびPL/SQLによって第三世代言語(3GL)ファンクションおよびプロシージャを起動できるように、CREATE FUNCTION文またはCREATE PROCEDURE文のcall_specで使用したり、パッケージまたは型のファンクションまたはプロシージャを宣言する際に使用できます。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

自分のスキーマ内にライブラリを作成するには、CREATE LIBRARYシステム権限が必要です。他のユーザーのスキーマ内にライブラリを作成する場合は、CREATE ANY LIBRARYシステム権限が必要です。

ディレクトリ・オブジェクト内のDLLに関連付けられるライブラリを作成するには、そのディレクトリ・オブジェクトに対するEXECUTEオブジェクト権限が必要です。

資格証明名に関連付けられるライブラリを作成するには、その資格証明名に対するEXECUTEオブジェクト権限が必要です。

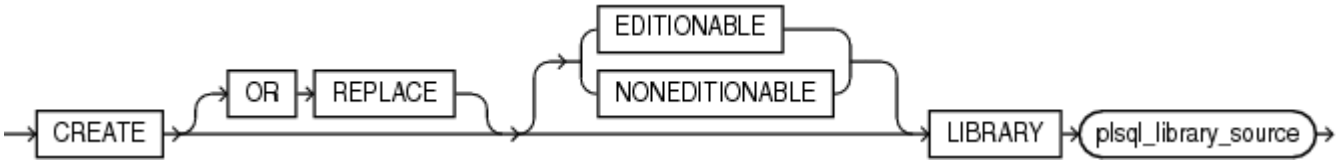
CREATE FUNCTION文のcall_specでライブラリを使用したり、パッケージまたは型のファンクションを宣言する場合には、そのライブラリに対するEXECUTEオブジェクト権限とCREATE FUNCTIONシステム権限が必要です。

CREATE PROCEDURE文のcall_specでライブラリを使用したり、パッケージまたは型のプロシージャを宣言する場合には、そのライブラリに対するEXECUTEオブジェクト権限とCREATE PROCEDUREシステム権限が必要です。

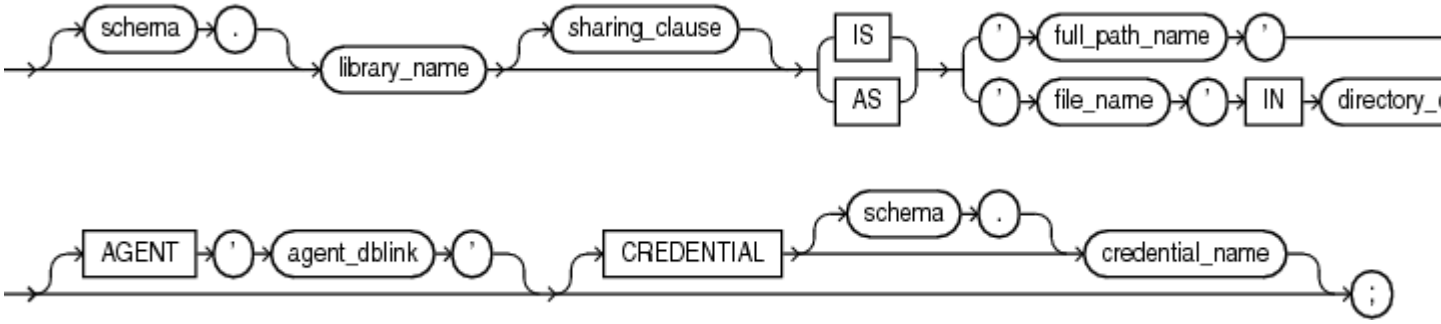
call_specで定義したプロシージャまたはファンクション(パッケージまたは型で定義したプロシージャまたはファンクションを含む)を実行するには、そのプロシージャまたはファンクションに対するEXECUTEオブジェクト権限が必要です(ただし、ライブラリに対するEXECUTEオブジェクト権限は不要です)。

構文

create_library ::=



plsql_library_source ::=



(sharing_clause ::=)

セマンティクス

create_library

OR REPLACE

ライブラリが存在する場合は、ライブラリを再作成し、再コンパイルします。

再定義する前のライブラリに対する権限を付与されていたユーザーは、権限を再付与される必要なく、ライブラリにアクセスできます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプLIBRARYに対してエディションが有効になっている場合に、ライブラリがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

plsql_library_source

schema

ライブラリが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

library_name

call_specとともにファンクションまたはプロシージャを宣言する場合に、このライブラリを表す名前。

'full_path_name'

一重引用符で囲まれた文字列リテラルであり、オペレーティング・システムにより共有ライブラリのフルパス名として認識されます。

CREATE LIBRARY文の実行中は、full_path_nameは解釈されません。共有ライブラリの存在は、ユーザーによりそのサブプログラムの1つが起動されるとチェックされます。

'file_name' IN directory_object

file_nameは一重引用符で囲まれた文字列リテラルであり、値はdirectory_object内のダイナミック・リンク・ライブラリ(DLL)の名前になります。文字列リテラルは2,000バイトを超えてはならず、またパス・デリミタを含むことはできません。コンパイラによりfile_nameは無視されますが、実行時にはfile_nameでパス・デリミタの有無がチェックされます。

directory_object

directory_objectは、CREATE DIRECTORY文を使用して作成されるディレクトリ・オブジェクトです(『Oracle Database SQL言語リファレンス』を参照)。directory_objectが存在しない場合、またはdirectory_objectに対するEXECUTEオブジェクト権限がない場合、ライブラリはエラーとともに作成されます。続けてdirectory_objectが作成されても、ライブラリは無効になります。ライブラリが無効になるその他の理由は次のとおりです。

- directory_objectが削除された。
- directory_objectが無効になった。
- directory_objectに対するEXECUTEオブジェクト権限が取り消された。

事前定義のPATH_PREFIXを含むPDB内にライブラリ・オブジェクトを作成する場合、そのライブラリでディレクトリ・オブジェクトを使用する必要があります。ディレクトリ・オブジェクトはPATH_PREFIXの規則をライブラリ・オブジェクトに強制します。ライブラリ・オブジェクトでディレクトリ・オブジェクトを使用しなかった場合、コンパイル・エラーが発生します。

事前定義のPATH_PREFIXを持つPDBとしてデータベースがCDBにプラグインされている場合、ディレクトリ・オブジェクトを使用していないライブラリを使用しようとすると、ORA-65394エラーが発生します。ライブラリ・オブジェクトは無効化されませんが、これを使用できるようにするには、ディレクトリ・オブジェクトを使用して再作成する必要があります。 CDB管理の詳細は、『Oracle Multitenant管理者ガイド』を参照してください。

AGENT 'agent_dblink'

サーバー以外のデータベース・リンクから外部プロシージャを実行できます。Oracle Databaseでは、agent_dblinkによって指定されたデータベース・リンクを使用して外部プロシージャを実行します。この句を省略すると、サーバー上のデフォルト・エージェント(extproc)によって外部プロシージャが実行されます。

CREDENTIAL [schema.]credential_name

ライブラリを指定する外部サブプログラムの実行時にextprocエージェントが偽装するオペレーティング・システム・ユーザーの資格証明を指定します。デフォルト: Oracle Databaseインストールの所有者。

credential_nameが存在しない場合、またはcredential_nameに対するEXECUTEオブジェクト権限がない場合、ライブラリはエラーとともに作成されます。続けてcredential_nameが作成されても、ライブラリは無効になります。ライブラリが無効になるその他の理由は次のとおりです。

- credential_nameが削除された。
- credential_nameが無効になった。
- credential_nameに対するEXECUTEオブジェクト権限が取り消された。

資格証明の使用の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。

例

例14-17 ライブラリの作成

次の文は、ディレクトリ・オブジェクトを使用して、ライブラリext_libを作成します。

```
CREATE LIBRARY ext_lib AS 'ddl_1' IN ddl_dir;
/
```

次の文によって、ディレクトリ・オブジェクトと資格証明を使用して、ライブラリext_libが再作成されます。

```
CREATE OR REPLACE LIBRARY ext_lib AS 'ddl_1' IN ddl_dir CREDENTIAL ddl_cred;
/
```

次の文によって、明示的なパスを使用して、ライブラリext_libが作成されます。

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
/
```

次の文によって、明示的なパスを使用して、ライブラリext_libが再作成されます。

```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
/
```

例14-18 外部プロシージャ・エージェントの指定

次の例は、ライブラリapp_libを作成し(明示的なパスを使用して)、外部プロシージャがパブリック・データベースsales.hq.example.comから実行されるよう指定します。

```
CREATE LIBRARY app_lib as '${ORACLE_HOME}/lib/app_lib.so'
   AGENT 'sales.hq.example.com';
/
```

関連項目:

データベース・リンクの作成の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

関連トピック

- ALTER LIBRARY文
- DROP LIBRARY文
- CREATE FUNCTION文
- CREATE PROCEDURE文

14.9 CREATE PACKAGE文

CREATE PACKAGE文は、データベースに1つの単位として格納される、関連するプロシージャやファンクション、その他のプログラム・オブジェクトをカプセル化したコレクションである、ストアド・パッケージの仕様部を作成するか、または置き換えます。

パッケージ仕様部でこれらのオブジェクトを宣言します。その後に指定するパッケージ本体で、これらのオブジェクトを定義します。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

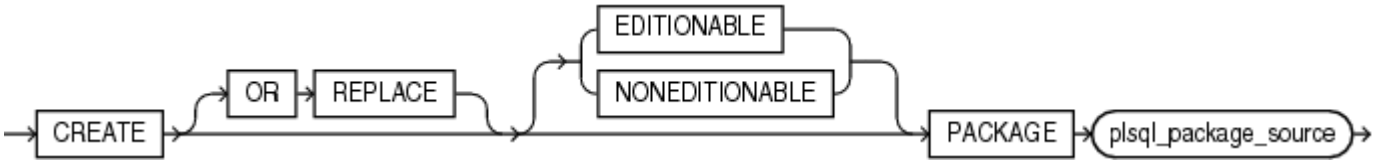
前提条件

自分のスキーマ内にパッケージを作成するか、または自分のスキーマ内のパッケージを置き換えるには、CREATE PROCEDUREシステム権限が必要です。他のユーザーのスキーマ内にパッケージを作成または再作成する場合は、CREATE ANY PROCEDUREシステム権限が必要です。

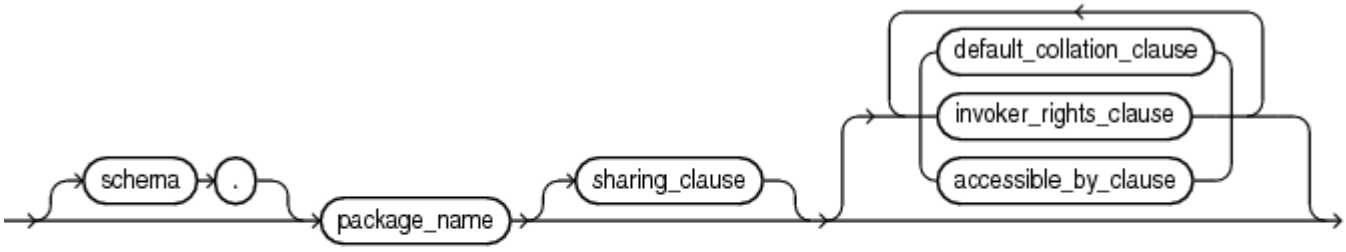
Oracleデータベースのプリコンパイラ・プログラム内にCREATE PACKAGE文を埋め込む場合は、キーワードEND-EXECに続けて、各言語の埋込みSQL文の終了記号を記述して文を終了する必要があります。

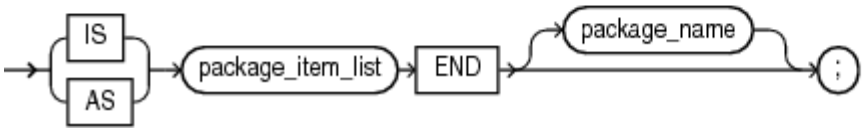
構文

create_package ::=



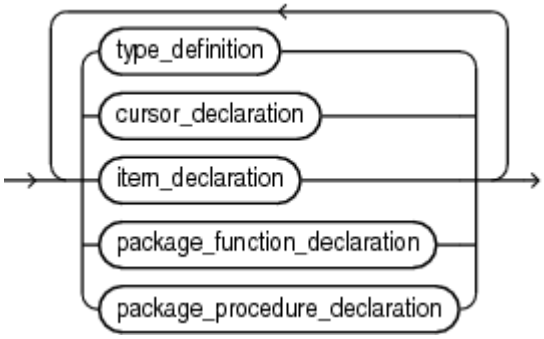
plsql_package_source ::=





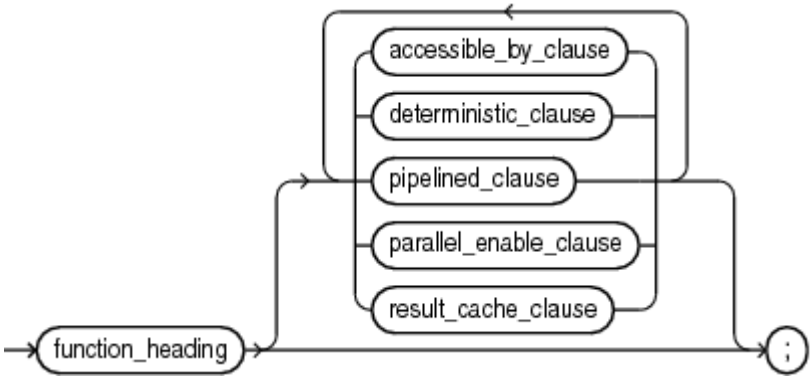
(sharing_clause ::= , default_collation_clause ::= , invoker_rights_clause ::= , accessible_by_clause ::= ,)

package_item_list ::=



(cursor_declaration ::=、item_declaration ::=、type_definition ::=)

package_function_declaration ::=



(function_heading ::= , accessible_by_clause ::= , deterministic_clause ::= , pipelined_clause ::=,parallel_enable_clause ::= ,result_cache_clause ::=)

package_procedure_declaration ::=

(procedure_heading ::=, accessible_by_clause ::=)

セマンティクス

create_package

OR REPLACE

パッケージが存在する場合は、パッケージを再作成し、再コンパイルします。

再定義する前のパッケージに対する権限を付与されていたユーザーは、権限を再付与される必要なく、パッケージにアクセスできます。

ファンクション索引がパッケージに依存している場合、索引にDISABLEDのマークが付きます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプPACKAGEに対してエディションが有効になっている場合に、パッケージがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非

エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

plsql_package_source

schema

パッケージが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

package_name

データベースに格納されたパッケージを識別します。ネーミング規則については、「Identifiers」を参照してください。

package_item_list

パッケージ内のすべてのタイプを定義し、パッケージ内のすべてのカーソルとサブプログラムを宣言します。多相テーブル・ファンクションを除いて、すべての宣言には、対応する定義がパッケージ本体に含まれている必要があります。対応する宣言と定義のヘッダーは、空白以外の一語一語が一致している必要があります。パッケージの多相テーブル・ファンクションは同じパッケージ内にその実装パッケージとして宣言する必要があります。

package_item_listの制限

PRAGMA AUTONOMOUS_TRANSACTIONは、ここには記述できません。

例

例14-19 emp_mgmtパッケージの仕様部の作成

この文は、emp_mgmtパッケージの仕様を作成します。

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
   FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
      manager_id NUMBER, salary NUMBER,
      commission_pct NUMBER, department_id NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
      RETURN NUMBER;
   PROCEDURE remove_emp(employee_id NUMBER);
   PROCEDURE remove_dept(department_id NUMBER);
   PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER);
   PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER);
   no_comm EXCEPTION;
   no_sal EXCEPTION;
END emp_mgmt;
```

emp_mgmtパッケージの仕様部では、次のようなパブリック・プログラム・オブジェクトを宣言します。

- ファンクションhireおよびcreate_dept
- プロシージャremove_emp、remove_dept、increase_salおよびincrease_comm
- 例外no_commおよびno_sal

パッケージにアクセスできるユーザーは、これらすべてのオブジェクトを使用できます。パッケージの作成後、これらのパブリック・プロシージャまたはパブリック・ファンクションのいずれかを起動したり、パッケージの任意のパブリック例外を呼び出すアプリケーションを

開発できます。

このパッケージのプロシージャおよびファンクションを起動するには、これらのプロシージャおよびファンクションをパッケージ本体で定義しておく必要があります。emp_mgmtパッケージの本体を作成するCREATE PACKAGE BODY文の例は、「CREATE PACKAGE BODY文」を参照してください。

関連トピック

この章:

- 「ALTER PACKAGE文」
- CREATE PACKAGE文
- 「CREATE PACKAGE BODY文」
- DROP PACKAGE文

他の章:

- PL/SQLパッケージ
- 「パッケージ仕様部」
- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」

14.10 CREATE PACKAGE BODY文

CREATE PACKAGE BODY文は、データベースに1つの単位として格納される、関連するプロシージャやストアド・ファンクション、その他のプログラム・オブジェクトをカプセル化したコレクションである、ストアド・パッケージの本体を作成するか、または置き換えます。

パッケージ本体でこれらのオブジェクトを定義します。前述のCREATE PACKAGE文で定義したパッケージ仕様部で、これらのオブジェクトを宣言します。

一連のプロシージャやファンクションをスタンドアロンのスキーマ・オブジェクトとして作成するかわりの方法としてパッケージを使用する方法があります。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

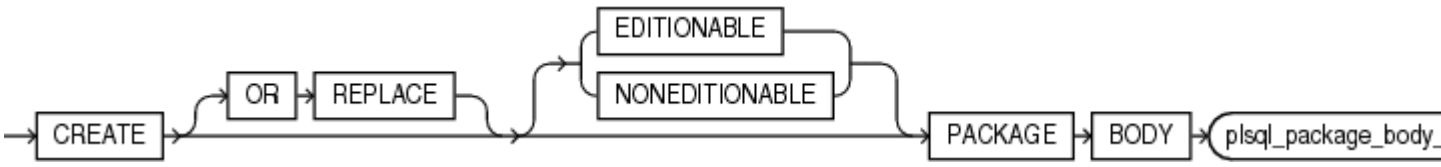
前提条件

自分のスキーマ内にパッケージを作成するか、または自分のスキーマ内のパッケージを置き換えるには、CREATE PROCEDUREシステム権限が必要です。他のユーザーのスキーマ内にパッケージを作成または再作成する場合は、CREATE ANY PROCEDUREシステム権限が必要です。いずれの場合も、パッケージ本体をパッケージと同じスキーマに作成する必要があります。

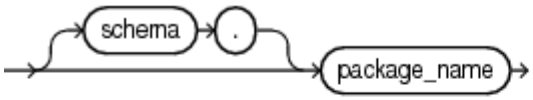
CREATE PACKAGE BODY文をデータベースのプリコンパイラ・プログラム内に埋め込むには、キーワードEND-EXECの後に特定の言語の埋込みSQL文の終了記号を付けて文を終了する必要があります。

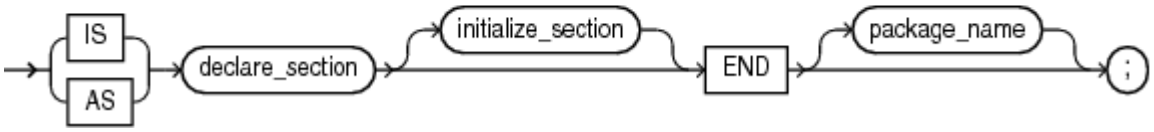
構文

create_package_body ::=



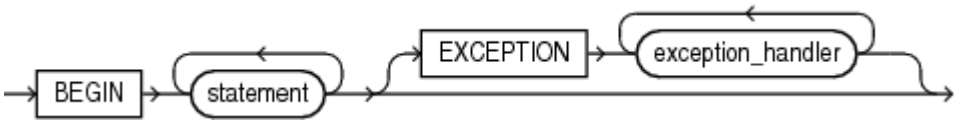
plsql_package_body_source ::=





(declare_section ::=)

initialize_section ::=



(statement ::=、exception_handler ::=)

セマンティクス

create_package_body

OR REPLACE

パッケージ本体が存在する場合は、パッケージ本体を再作成し、再コンパイルします。

再定義する前のパッケージ本体に対する権限を付与されていたユーザーは、権限を再付与される必要なく、パッケージにアクセスできます。

[EDITIONABLE | NONEDITIONABLE]

このプロパティを指定しないと、パッケージ本体はパッケージ仕様部からEDITIONABLEまたはNONEDITIONABLEを継承します。このプロパティを指定する場合は、パッケージ仕様部のプロパティと一致する必要があります。

plsql_package_body_source

schema

パッケージが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

package_name

作成するパッケージの名前。

declare_section

パッケージ仕様部のすべてのカーソル定義およびサブプログラム宣言を配置します。対応するサブプログラム宣言と定義のヘッダーは、空白以外の一語一語が一致している必要があります。

パッケージ内からのみ参照できるプライベート項目を宣言および定義することもできます。

declare_sectionの制限

PRAGMA AUTONOMOUS_TRANSACTIONは、ここには記述できません。

initialize_section

変数を初期化し、その他の1回のみの設定ステップをすべて実行します。

例

例14-20 emp_mgmtパッケージ本体の作成

この文では、emp_mgmtパッケージの本体を作成します。このパッケージは、「例14-19」で作成したものです。

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
   tot_emps NUMBER;
   tot_depts NUMBER;
FUNCTION hire
   (last_name VARCHAR2, job_id VARCHAR2,
    manager_id NUMBER, salary NUMBER,
    commission_pct NUMBER, department_id NUMBER)
   RETURN NUMBER IS new_empno NUMBER;
BEGIN
   SELECT employees_seq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO employees
      VALUES (new_empno, 'First', 'Last','first.example@example.com',
              '(415)555-0100',
              TO_DATE('18-JUN-2002','DD-MON-YYYY'),
              'IT_PROG',90000000,00, 100,110);
      tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;
FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
   RETURN NUMBER IS
      new_deptno NUMBER;
   BEGIN
      SELECT departments_seq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO departments
         VALUES (new_deptno, 'department name', 100, 1700);
      tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;
PROCEDURE remove_emp (employee_id NUMBER) IS
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
      tot_emps := tot_emps - 1;
   END;
PROCEDURE remove_dept(department_id NUMBER) IS
   BEGIN
      DELETE FROM departments
      WHERE departments.department_id = remove_dept.department_id;
      tot_depts := tot_depts - 1;
      SELECT COUNT(*) INTO tot_emps FROM employees;
   END;
PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER) IS
   curr_sal NUMBER;
   BEGIN
      SELECT salary INTO curr_sal FROM employees
      WHERE employees.employee_id = increase_sal.employee_id;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE employees
         SET salary = salary + salary_incr
         WHERE employee_id = employee_id;
      END IF;
   END;
```

```
PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER) IS
   curr_comm NUMBER;
   BEGIN
      SELECT commission_pct
      INTO curr_comm
      FROM employees
      WHERE employees.employee_id = increase_comm.employee_id;
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE employees
         SET commission_pct = commission_pct + comm_incr;
      END IF;
   END;
END emp_mgmt;
```

パッケージ本体では、パッケージ仕様部で宣言された次のパブリック・プログラム・オブジェクトを定義します。

- ファンクションhireおよびcreate_dept
- プロシージャremove_emp、remove_dept、increase_salおよびincrease_comm

これらのオブジェクトはパッケージ仕様部で宣言されているため、パッケージ外部のアプリケーション・プログラム、プロシージャおよびファンクションでコールできます。たとえば、パッケージにアクセスできる場合は、increase_commプロシージャを起動するemp_mgmtパッケージとは別にプロシージャincrease_all_commsを作成できます。

これらのオブジェクトはパッケージ本体で定義されているため、依存するスキーマ・オブジェクトをデータベースで無効にしなくても、オブジェクトの定義を変更できます。たとえば、この後にhireの定義を変更した場合、データベースで実行する前にincrease_all_commsを再コンパイルする必要はありません。

また、この例のパッケージ本体では、プライベート・プログラム・オブジェクトである変数tot_empsおよびtot_deptsも宣言しています。これらのオブジェクトは、パッケージ仕様部ではなくパッケージ本体で宣言されているため、パッケージ内の他のオブジェクトからはアクセスできますが、パッケージの外部からはアクセスできません。たとえば、変数tot_deptsの値を明示的に変更するアプリケーションを開発することはできません。ただし、ファンクションcreate_deptはパッケージの一部であるため、create_deptでtot_deptsの値を変更できます。

関連トピック

この章:

- CREATE PACKAGE文

他の章:

- 「PL/SQLパッケージ」
- 「パッケージ本体」
- 「ファンクションの宣言および定義」
- 「プロシージャの宣言および定義」

14.11 CREATE PROCEDURE文

CREATE PROCEDURE文は、スタンドアロン・プロシージャまたはコール仕様を作成するか、または置き換えます。

スタンドアロン・プロシージャとは、データベースに格納されるプロシージャ(特定のアクションを実行するサブプログラム)です。

> ノート:
>
> CREATE PROCEDURE 文で作成するスタンドアロン・プロシージャは、PL/SQL ブロックまたはパッケージで宣言および定義するプロシージャとは異なります。詳細は、プロシージャの宣言および定義または CREATE PACKAGE 文を参照してください。

コール仕様は、PL/SQLからコールできるようにJavaメソッドまたはC言語プロシージャを宣言します。このようなメソッドまたはサブプログラムは、SQLのCALL文を使用して起動することもできます。コール仕様は、起動時に起動するJavaメソッドまたは共有ライブラリ内の名前付きプロシージャをデータベースに指示します。また、引数および戻り値に対して実行する型変換もデータベースに指示します。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

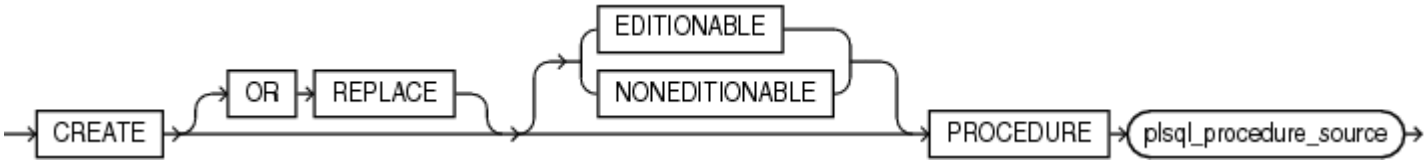
自分のスキーマ内にスタンドアロン・プロシージャを作成するか、または自分のスキーマ内のスタンドアロン・プロシージャを置き換えるには、CREATE PROCEDUREシステム権限が必要です。別のユーザーのスキーマ内にスタンドアロン・プロシージャを作成するか、または別のユーザーのスキーマ内のスタンドアロン・プロシージャを置き換えるには、CREATE ANY PROCEDUREシステム権限が必要です。

コール仕様を起動するには、追加の権限が必要になる場合があります。たとえば、Cコール仕様の場合には、Cライブラリに対するEXECUTEオブジェクト権限が必要です。

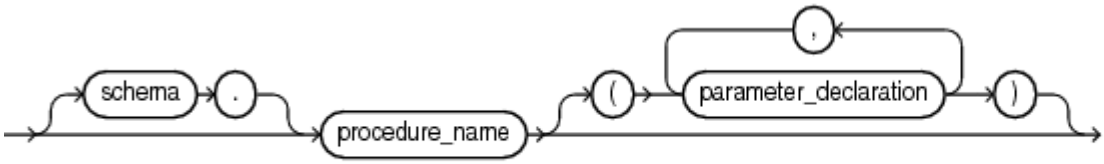
Oracleプリコンパイラ・プログラム内にCREATE PROCEDURE文を埋め込む場合、キーワードEND-EXECに続けて、各言語の埋込みSQL文の終了記号を記述して文を終了する必要があります。

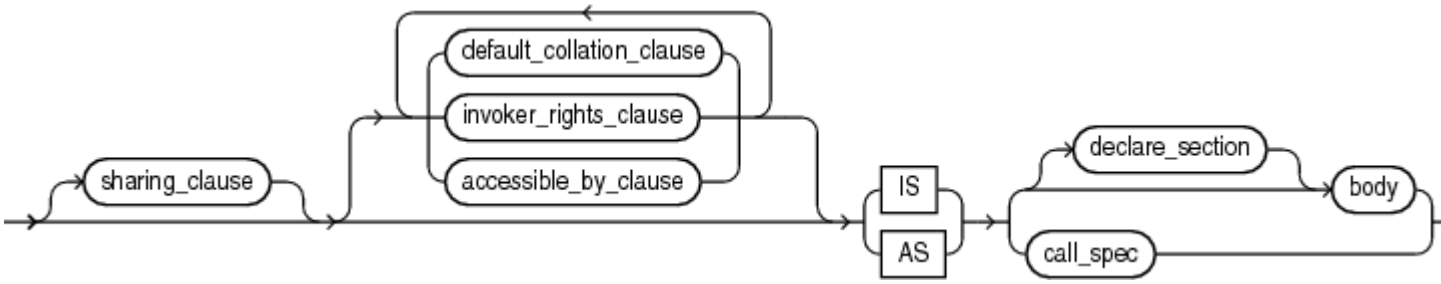
構文

create_procedure ::=



plsql_procedure_source ::=





(sharing_clause ::=、default_collation_clause ::=、invoker_rights_clause ::=、accessible_by_clause ::=、call_spec ::=、body ::=、declare_section ::=、parameter_declaration ::=)

セマンティクス

create_procedure

OR REPLACE

プロシージャが存在する場合は、プロシージャを再作成し、再コンパイルします。

再定義する前のプロシージャに対する権限を付与されていたユーザーは、権限を再付与される必要なく、プロシージャにアクセスできます。

ファンクション索引がプロシージャに依存している場合、データベースによって索引にDISABLEDのマークが付けられます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプPROCEDUREに対してエディションが有効になっている場合に、プロシージャがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

plsql_procedure_source

schema

プロシージャが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

procedure_name

作成するプロシージャの名前。

ノート:

SQL*Module によって生成されたスタブを使用してストアド・サブプログラムを起動する場合、ストアド・サブプログラム名は、起動側ホストの 3GL 言語(Ada や C など)の有効な識別子である必要もあります。

body

プロシージャの必須の実行部、およびオプションでプロシージャの例外処理部。

declare_section

プロシージャのオプションの宣言部。宣言は、プロシージャに対してローカルであり、bodyで参照でき、プロシージャが実行を完了すると消滅します。

例

例14-21 プロシージャの作成

この文は、スキーマhrにプロシージャremove_empを作成します。

```
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
   tot_emps NUMBER;
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
   tot_emps := tot_emps - 1;
   END;
/
```

remove_empプロシージャは、指定した従業員を削除します。このプロシージャを起動する場合は、削除する従業員のemployee_idを指定する必要があります。

このプロシージャは、DELETE文を使用して、employees表からemployee_idの行を削除します。

関連項目:

このプロシージャをパッケージに組み入れる方法は、「CREATE PACKAGE BODY文」を参照してください

例14-22 外部プロシージャの作成

次の例では、外部プロシージャc_find_rootはポインタをパラメータとして予測します。プロシージャfind_rootは、BY REFERENCE句を使用して、このパラメータを参照方式で渡します。

```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME c_find_root
      LIBRARY c_utils
```

```
      PARAMETERS ( x BY REFERENCE );
```

関連トピック

この章:

- ALTER PROCEDURE文
- CREATE FUNCTION文
- DROP PROCEDURE文

他の章:

- 仮パラメータの宣言
- 「プロシージャの宣言および定義」
- PL/SQLサブプログラム

その他のドキュメント:

- CALL文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- コール仕様の詳細は、『Oracle Database開発ガイド』を参照してください。
- ストアドPL/SQLサブプログラムの起動の詳細は、Oracle Database開発ガイド を参照してください

14.12 CREATE TRIGGER文

CREATE TRIGGER文は、次のいずれかのデータベース・トリガーを作成するか、または置き換えます。

- 表、ビュー、スキーマまたはデータベースに関連付けられているストアドPL/SQLブロック
- 無名PL/SQLブロック、あるいはPL/SQLまたはJavaで実装されているプロシージャの起動

指定された条件が発生した場合、トリガーは自動的に実行されます。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

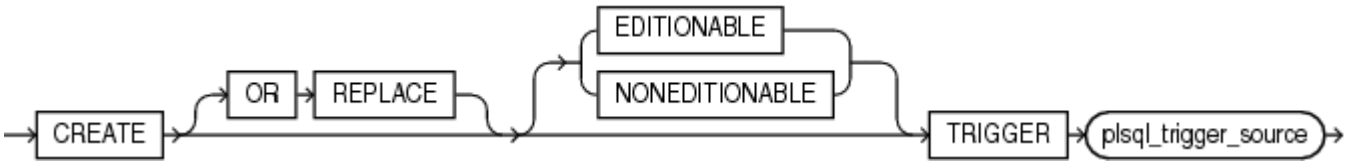
前提条件

- 自分のスキーマ内の表または自分のスキーマ(SCHEMA)に対するトリガーを自分のスキーマ内に作成する場合は、CREATE TRIGGERシステム権限が必要です。
- 任意のスキーマ内の表または別のユーザーのスキーマ(schema.SCHEMA)に対するトリガーを任意のスキーマ内に作成する場合は、CREATE ANY TRIGGERシステム権限が必要です。
- 前述の権限に加えて、DATABASEに対するトリガーを作成する場合は、ADMINISTER DATABASE TRIGGERシステム権限が必要です。
- プラガブル・データベース(PDB)に対するトリガーを作成する場合は、そのPDBに接続し、ADMINISTER DATABASE TRIGGERシステム権限を持っていることが必要です。PDBの詳細は、『Oracle Database管理者ガイド』を参照してください。
- 前述の権限に加えて、crosseditionトリガーを作成する場合は、エディションが有効になっている必要があります。ユーザーに対するエディションの有効化の詳細は、『Oracle Database開発ガイド』を参照してください。

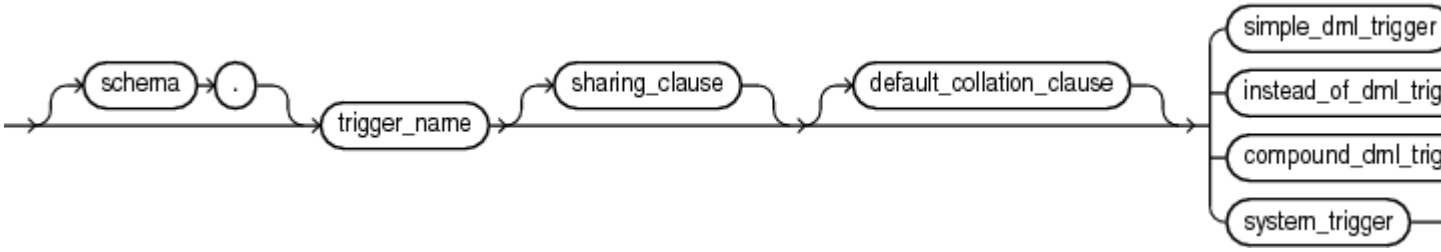
トリガーがSQL文を発行するか、あるいはプロシージャまたはファンクションを起動する場合、そのトリガーの所有者には、これらの操作を行うための権限が必要です。これらの権限は、ロールを介して付与するのではなく、所有者に直接付与する必要があります。

構文

create_trigger ::=

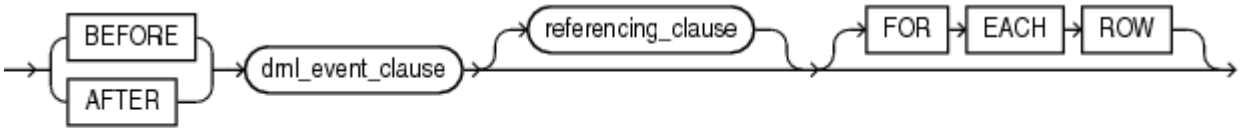


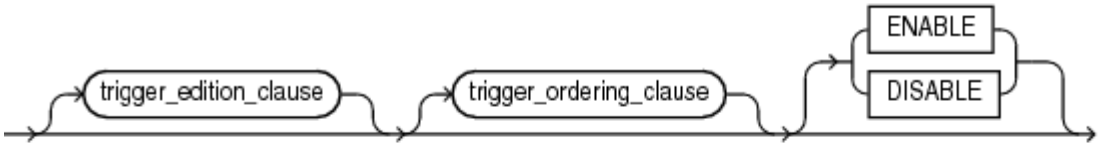
plsql_trigger_source ::=



(sharing_clause ::=、default_collation_clause ::=、compound_dml_trigger ::=、instead_of_dml_trigger ::=、system_trigger ::=)

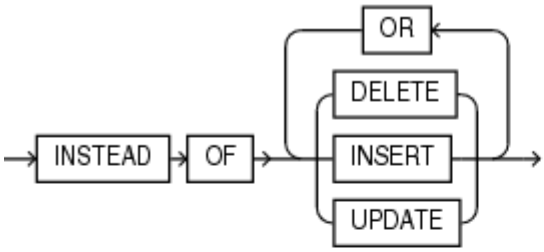
simple_dml_trigger ::=

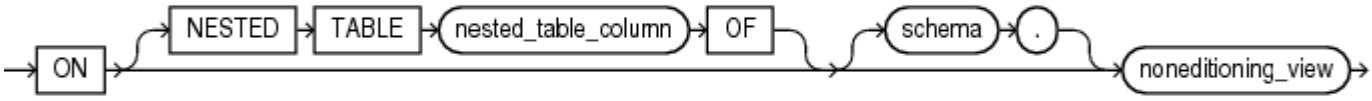


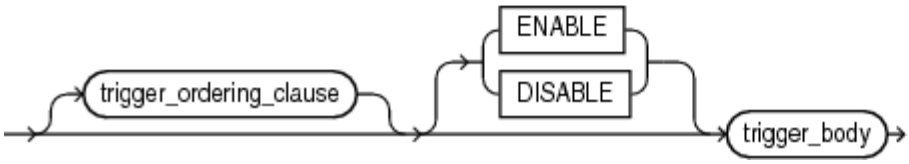


(dml_event_clause ::=、referencing_clause ::=、trigger_body ::=、trigger_edition_clause ::=、trigger_ordering_clause ::=)

instead_of_dml_trigger ::=



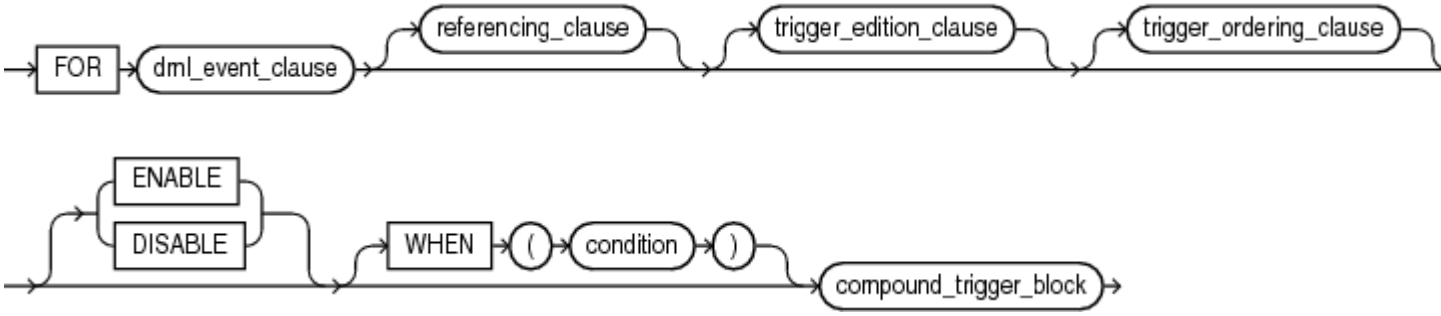




(referencing_clause ::=、trigger_body ::=、trigger_edition_clause ::=、

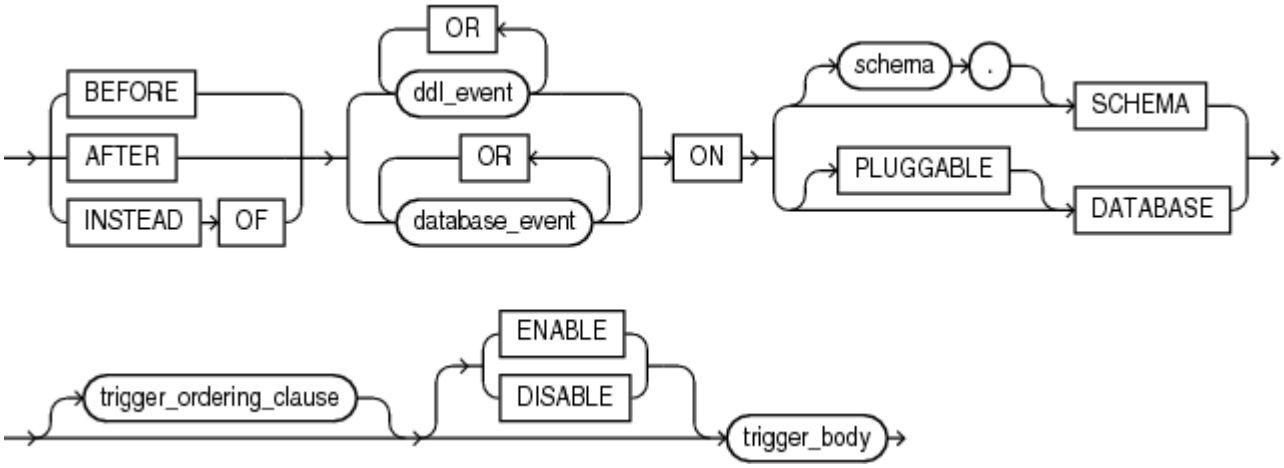
trigger_ordering_clause ::=)

compound_dml_trigger ::=



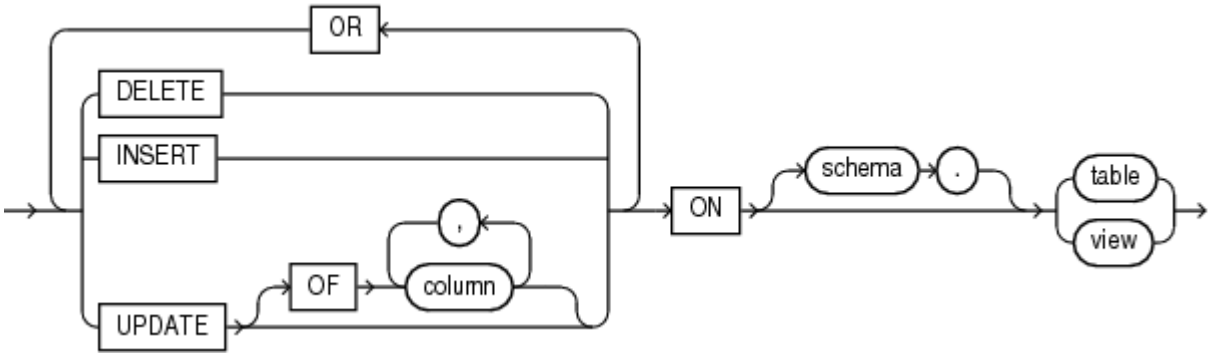
(compound_trigger_block ::=、dml_event_clause ::=、referencing_clause ::=、trigger_edition_clause ::=、trigger_ordering_clause ::=)

system_trigger ::=

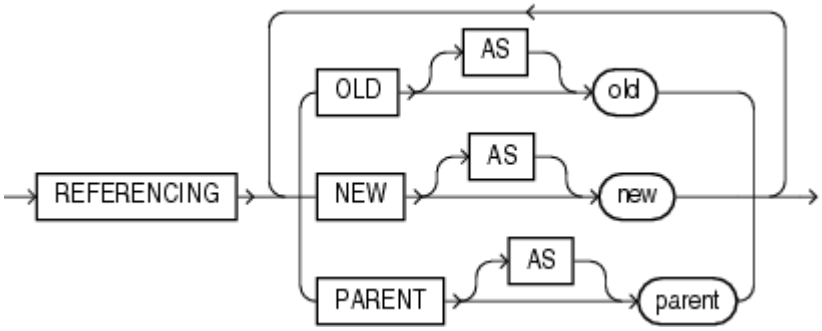


(trigger_body ::=、trigger_ordering_clause ::=)

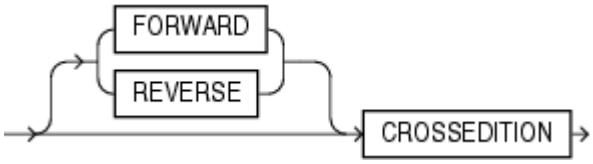
dml_event_clause ::=



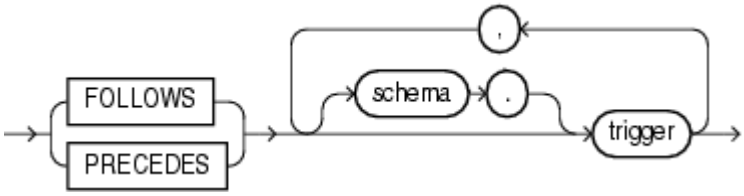
referencing_clause ::=



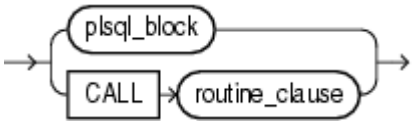
trigger_edition_clause ::=



trigger_ordering_clause ::=



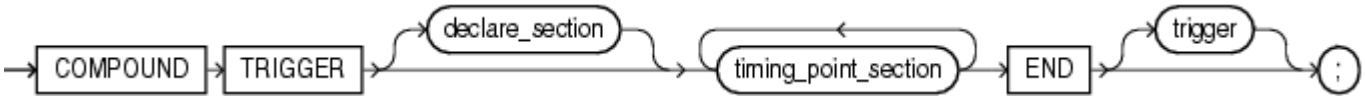
trigger_body ::=



(plsql_block ::=、

『Oracle Database SQL言語リファレンス』の routine_clause)

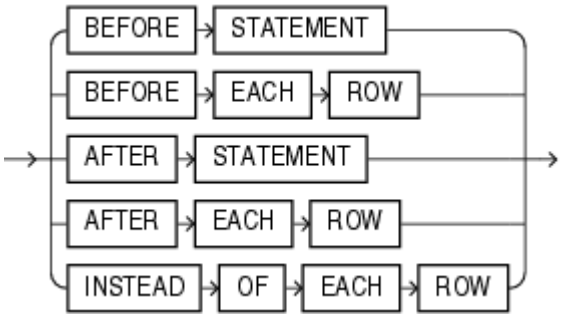
compound_trigger_block ::=



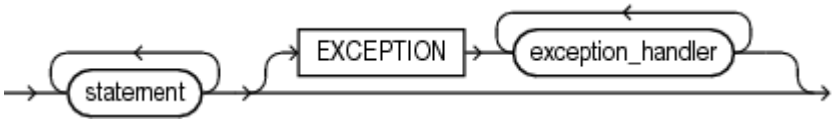
(declare_section ::=)

timing_point_section ::=

timing_point ::=



tps_body ::=



(exception_handler ::=、statement ::=)

セマンティクス

create_trigger

OR REPLACE

トリガーが存在する場合は、トリガーを再作成し、再コンパイルします。

再定義する前のトリガーに対する権限を付与されていたユーザーは、権限を再付与される必要なく、プロシージャにアクセスできます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプTRIGGERに対してエディションが有効になっている場合に、トリガーがエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

NONEDITIONABLEの制限

crosseditionトリガーに対してNONEDITIONABLEを指定できません。

create_triggerの制限

トリガーの制限を参照してください。

plsql_trigger_source

schema

作成するトリガーに対するスキーマの名前。デフォルト: 自分のスキーマ。

trigger

作成するトリガーの名前。

同じスキーマ内のトリガーには同じ名前を付けることができません。トリガーには、他のスキーマ・オブジェクトと同じ名前を付けることができます(たとえば、表とトリガーは同じ名前でもかまいませんが、混乱を避けるため、違う名前を付けることをお薦めします)。

トリガーでコンパイル・エラーが発生した場合、そのトリガーは作成されますが実行時に正常に処理されません。トリガーを正常に実行できない場合、トリガーが無効にされるか、コンパイル・エラーのないバージョンに置き換えられるか、または削除されるまで、トリガーを起動するすべてのDML文は実行できません。関連付けられているコンパイラ・エラー・メッセージは、SQL*PlusコマンドSHOW ERRORSを使用して確認できます。

ノート:

マテリアライズド・ビューの実表にトリガーを作成する場合は、トリガーがマテリアライズド・ビューのリフレッシュ中に起動されないようにする必要があります。リフレッシュ中、DBMS_MVIEW プロシージャ I_AM_A_REFRESH は TRUE を戻します。

simple_dml_trigger

単純なDMLトリガーを作成します(「DMLトリガー」を参照)。

BEFORE

トリガーを起動するイベントを実行する前にデータベースによってトリガーを起動できます。行トリガーの場合、影響を受ける各行が変更される前にトリガーが起動されます。

BEFOREの制限

- BEFOREトリガーは、エディショニング・ビュー以外のビューには指定できません。
- BEFORE文トリガーでは、トリガー本体は:NEWまたは:OLDを読み取ることはできません。(BEFORE行トリガーでは、トリガー本体は:OLDおよび:NEWフィールドの読取りと書込みを行えます。)

AFTER

トリガーを起動するイベントを実行した後にデータベースによってトリガーを起動できます。行トリガーの場合、影響を受ける各行が変更された後にトリガーが起動されます。

AFTERの制限

- AFTERトリガーは、エディショニング・ビュー以外のビューには指定できません。
- AFTER文トリガーでは、トリガー本体は:NEWまたは:OLDを読み取ることはできません。(AFTER行トリガーでは、トリガー本体は:OLDおよび:NEWフィールドを読み取れますが、書き込めません。)

ノート:

表に対するマテリアライズド・ビュー・ログを作成した場合は、その表に AFTER 行トリガーが暗黙的に作成されます。このトリガーは、INSERT 文、UPDATE 文または DELETE 文で関連付けられた表のデータが変更された場合、常にマテリアライズド・ビュー・ログに 1 行を挿入します。複数の行トリガーを起動する順序は制御できません。したがって、マテリアライズド・ビューの内容に影響するトリガーは記述しないでください。

関連項目:

- マテリアライズド・ビュー・ログの詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- エディショニング・ビューの詳細は、『Oracle Database開発ガイド』を参照してください。

FOR EACH ROW

トリガーを行トリガーとして作成します。トリガーを起動する文の影響を受け、WHEN条件で定義されているオプションのトリガー制約を満たす各行に対して、データベースによって行トリガーが起動されます。

この句を省略した場合、トリガーは文トリガーになります。オプションのトリガー制約が満たされている場合にトリガーを起動する文が発行されたときのみ、データベースによって文トリガーが起動されます。

[ENABLE | DISABLE]

トリガーを有効(デフォルト)または無効な状態で作成します。無効な状態でトリガーを作成すると、有効化する前にトリガーがエラーなしでコンパイルされることを確認できます。

> ノート:
>
> DISABLE は、crossedition トリガーを作成する場合に特に役立ちます。crossedition トリガーは、コンパイル・エラーが発生した場合にオンライン・アプリケーションの再定義に影響を与えるためです。

WHEN (condition)

トリガーを起動する文に影響を受ける行ごとにデータベースで評価されるSQL条件を指定します。影響を受ける行に対してconditionの値がTRUEの場合は、その行に対してtrigger_bodyが実行されますが、それ以外の場合は、その行に対してtrigger_bodyは実行されません。トリガーを起動する文は、conditionの値に関係なく実行されます。

conditionには相関名を指定できます(「referencing_clause ::=」を参照)。

conditionでは、相関名NEW、OLDまたはPARENTの前にコロン(:)を付けないでください(このコンテキストでは、相関名はバインド変数のプレースホルダではありません)。

関連項目:

SQL条件の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

WHEN (condition)の制限

- この句を指定する場合は、FOR EACH ROWも指定する必要があります。
- conditionには、副問合せまたはPL/SQL式(ユーザー定義のファンクションの起動など)を含めることはできません。

trigger_body

データベースがトリガーを起動するために実行するPL/SQLブロックまたはCALLサブプログラム。CALLサブプログラムは、PL/SQLサブプログラムまたはPL/SQLラッパー内のJavaサブプログラムのいずれかです。

trigger_bodyがPL/SQLブロックの場合、ブロック内にエラーがあると、CREATE [OR REPLACE]文は失敗します。

trigger_bodyの制限

declare_sectionでは、データ型LONGまたはLONG RAWの変数を宣言できません。

instead_of_dml_trigger

INSTEAD OF DMLトリガーを作成します(「INSTEAD OF DMLトリガー」を参照)。

INSTEAD OFの制限

INSTEAD OFトリガーは、:OLD値および:NEW値を読み取ることができますが、変更することはできません。

ノート:

- ビューが本質的に更新可能であり、そのビューに INSTEAD OF トリガーがある場合は、トリガーが優先されます。ビューで DML を実行するかわりに、トリガーが起動されます。
- ビューが階層に属する場合、サブビューはトリガーを継承しません。
- ビューの WITH CHECK OPTION は、INSTEAD OF トリガーを使用してビューに対する挿入または更新を行う場合は実行されません。INSTEAD OF トリガー本体でチェックを実行する必要があります。WITH CHECK OPTION の詳細は、『Oracle Database SQL 言語リファレンス』を参照してください。
- データベースのファイングレイン・アクセス制御を使用すると、ビューに行レベルのセキュリティ・ポリシーを定義できます。これらのポリシーによって、DML 操作に応じて指定されたルールが適用されます。ビューに INSTEAD OF トリガーも定義すると、データベースでは、ビューに対して DML が実行されるかわりに、INSTEAD OF トリガーが起動されるため、行レベルのセキュリティ・ポリシーは適用されません。

DELETE

トリガーが非エディショニング・ビューで作成される場合、DELETEにより、非エディショニング・ビューが定義されている表からDELETE文によって行が削除されるたびにデータベースによりトリガーが起動されるようになります。

トリガーが非エディショニング・ビューのネストした表の列で作成される場合、DELETEにより、ネストした表からDELETE文によって要素が削除されるたびにデータベースによりトリガーが起動されるようになります。

INSERT

トリガーが非エディショニング・ビューで作成される場合、INSERTにより、非エディショニング・ビューが定義されている表にINSERT文によって行が追加されるたびにデータベースによりトリガーが起動されるようになります。

トリガーが非エディショニング・ビューのネストした表の列で作成される場合、INSERTにより、ネストした表にINSERT文によって要素が追加されるたびにデータベースによりトリガーが起動されるようになります。

UPDATE

トリガーが非エディショニング・ビューで作成される場合、UPDATEにより、非エディショニング・ビューが定義されている表の列の値がUPDATE文によって変更されるたびにデータベースによりトリガーが起動されるようになります。

トリガーが非エディショニング・ビューのネストした表の列で作成される場合、UPDATEにより、ネストした表の列の値がUPDATE文によって変更されるたびにデータベースによりトリガーが起動されるようになります。

nested_table_column

トリガーを作成するnested_table_columnの名前。トリガーは、DMLの操作対象がネストした表の要素である場合にのみ起

動されます。ネストした表の列にDML操作を直接実行した場合、そのネストした表の列が含まれている表に定義されているトリガーはデータベースによって起動されません。詳細は、「INSTEAD OF DMLトリガー」を参照してください。

関連項目:

ビューに対する挿入、更新または削除を禁止する構造体の詳細は、『Oracle Database SQL言語リファレンス』の「CREATE VIEW」のAS subquery句に関する項を参照してください。

schema

非エディショニング・ビューが含まれるスキーマの名前。デフォルト: 自分のスキーマ。

noneditioning_view

nested_table_columnを指定する場合、noneditioning_viewはnested_table_columnを含む非エディショニング・ビューの名前です。それ以外の場合は、noneditioning_viewはトリガーを作成する非エディショニング・ビューの名前です。

FOR EACH ROW

INSTEAD OFトリガーは常に行トリガーであるため、ドキュメントでのみ使用します。

ENABLE

(デフォルト)有効な状態でトリガーを作成します。

DISABLE

無効な状態でトリガーを作成すると、有効化する前にトリガーがエラーなしでコンパイルされることを確認できます。

ノート:

DISABLE は、crossedition トリガーを作成する場合に特に役立ちます。crossedition トリガーは、コンパイル・エラーが発生した場合にオンライン・アプリケーションの再定義に影響を与えるためです。

trigger_body

データベースがトリガーを起動するために実行するPL/SQLブロックまたはCALLサブプログラム。CALLサブプログラムは、PL/SQLサブプログラムまたはPL/SQLラッパー内のJavaサブプログラムのいずれかです。

trigger_bodyがPL/SQLブロックの場合、ブロック内にエラーがあると、CREATE [OR REPLACE]文は失敗します。

trigger_bodyの制限

declare_sectionでは、データ型LONGまたはLONG RAWの変数を宣言できません。

compound_dml_trigger

複合DMLトリガーを作成します(「複合DMLトリガー」を参照)。

ENABLE

(デフォルト)有効な状態でトリガーを作成します。

DISABLE

無効な状態でトリガーを作成すると、有効化する前にトリガーがエラーなしでコンパイルされることを確認できます。

> ノート:
>
> DISABLE は、crossedition トリガーを作成する場合に特に役立ちます。crossedition トリガーは、コンパイル・エラーが発生した場合にオンライン・アプリケーションの再定義に影響を与えるためです。

WHEN (condition)

トリガーを起動する文に影響を受ける行ごとにデータベースで評価されるSQL条件を指定します。影響を受ける行に対してconditionの値がTRUEの場合は、その行に対してtps_bodyが実行されますが、それ以外の場合は、その行に対してtps_bodyは実行されません。トリガーを起動する文は、conditionの値に関係なく実行されます。

conditionには相関名を指定できます(「referencing_clause ::=」を参照)。conditionでは、相関名NEW、OLDまたはPARENTの前にコロン(:)を付けないでください(このコンテキストでは、相関名はバインド変数のプレースホルダではありません)。

関連項目:

SQL条件の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

WHEN (condition)の制限

- この句を指定する場合は、少なくとも次のタイミングのいずれかも指定する必要があります。
  - BEFORE EACH ROW
  - AFTER EACH ROW
  - INSTEAD OF EACH ROW
- conditionには、副問合せまたはPL/SQL式(ユーザー定義のファンクションの起動など)を含めることはできません。

system_trigger

システム・トリガーを作成します(「システム・トリガー」を参照)。

BEFORE

トリガーを起動するイベントを実行する前にデータベースによってトリガーを起動できます。

AFTER

トリガーを起動するイベントを実行した後にデータベースによってトリガーを起動できます。

INSTEAD OF

INSTEAD OFトリガーを作成します。

INSTEAD OFの制限

- トリガー・イベントは、CREATE文である必要があります。
- 最大1つのINSTEAD OF DDLトリガー(non_dml_trigger)を作成できます。

  たとえば、データベースまたはスキーマでINSTEAD OFトリガーを作成できますが、データベースとスキーマの両方でこれを作成できません。

ddl_event

トリガーを起動できる1つ以上の種類のDDL SQL文。

特に指定がないかぎり、DATABASEまたはSCHEMAに対してこれらのイベントのトリガーを作成できます。BEFOREおよびAFTERトリガーはこれらすべてのイベントに対して作成できますが、INSTEAD OFトリガーはCREATEイベントに対してのみ作成できます。データベースによって、既存のユーザー・トランザクションでトリガーが起動されます。

ノート:

一部のオブジェクトは、PL/SQL API を使用して作成、変更および削除されます(たとえば、スケジューラ・ジョブは DBMS_SCHEDULER パッケージ内のサブプログラムによって管理されます)。このような PL/SQL サブプログラムでは、DDL トリガーは起動されません。

次のddl_event値が有効です。

- ALTER

  ALTER文でデータ・ディクショナリのデータベース・オブジェクトを変更するたびにデータベースによってトリガーを起動できます。ALTER DATABASE文ではトリガーは起動されません。
- ANALYZE

  統計情報が収集または削除されるか、またはデータベース・オブジェクトの構造が検証されるたびにデータベースによってトリガーを起動できます。

  関連項目:

  SQL文のANALYZEを使用して統計情報を収集する方法の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- ASSOCIATE STATISTICS

  データベースで統計タイプがデータベース・オブジェクトに関連付けられるたびにデータベースによってトリガーを起動できます。
- AUDIT

  AUDIT文が発行されるたびにデータベースによってトリガーを起動できます。
- COMMENT

  データベース・オブジェクトに対するコメントがデータ・ディクショナリに追加されるたびにデータベースによってトリガーを起動

できます。

- CREATE

  CREATE文でデータ・ディクショナリにデータベース・オブジェクトを追加するたびにデータベースによってトリガーを起動できます。CREATE DATABASE文またはCREATE CONTROLFILE文ではトリガーは起動されません。

- DISASSOCIATE STATISTICS

  データベースで統計タイプがデータベース・オブジェクトとの関連付けを解除されるたびにデータベースによってトリガーを起動できます。

- DROP

  DROP文でデータ・ディクショナリからデータベース・オブジェクトが削除されるたびにデータベースによってトリガーを起動できます。

- GRANT

  ユーザーが別のユーザーまたはロールにシステム権限、ロールまたはオブジェクト権限を付与するたびにデータベースによってトリガーを起動できます。

- NOAUDIT

  NOAUDIT文が発行されるたびにデータベースによってトリガーを起動できます。

- RENAME

  RENAME文でデータベース・オブジェクトの名前を変更するたびにデータベースによってトリガーを起動できます。

- REVOKE

  REVOKE文でユーザーまたはロールからシステム権限、ロールまたはオブジェクト権限を取り消すたびにデータベースによってトリガーを起動できます。

- TRUNCATE

  TRUNCATE文で表またはクラスタから行を削除し、記憶特性を再設定するたびにデータベースによってトリガーを起動できます。

- DDL

  前述のDDL文のいずれかを発行するたびにデータベースによってトリガーを起動できます。

database_event

次のいずれかのデータベース・イベント。特に指定がないかぎり、DATABASEまたはSCHEMAに対してこれらのイベントのトリガーを作成できます。これらのトリガーを起動する各イベントに対して、データベースによって自律型トランザクションの有効範囲がオープンされ、トリガーが起動されて、(既存のユーザー・トランザクションには関係なく)別のトランザクションがコミットされます。

- AFTER STARTUP

  データベースがオープンされるたびにデータベースによってトリガーを起動できます。このイベントは、DATABASEでのみ有効であり、SCHEMAでは有効ではありません。

- BEFORE SHUTDOWN

  データベースのインスタンスが停止されるたびにデータベースによってトリガーを起動できます。このイベントは、DATABASEでのみ有効であり、SCHEMAでは有効ではありません。

- AFTER DB_ROLE_CHANGE

Oracle Data Guardの構成では、ロールがスタンバイからプライマリ、またはプライマリからスタンバイに変更されるたびにデータベースによってトリガーを起動できます。このイベントは、DATABASEでのみ有効であり、SCHEMAでは有効ではありません。

ノート:

PDB に対して AFTER DB_ROLE_CHANGE トリガーを作成することはできません。

- AFTER SERVERERROR

  次の条件の両方が該当するたびに、データベースによってトリガーを起動できます。

  - サーバー・エラー・メッセージがログに記録される。
  - Oracleリレーショナル・データベース管理システム(RDBMS)が、エラー・トリガーを起動することが妥当であると判断する。

    次のような場合は、エラー・トリガーを起動することが妥当ではありません。

    - RDBMSが起動中である。
    - クリティカル・エラーが発生した。

- AFTER LOGON

  クライアント・アプリケーションがデータベースにログオンするたびにデータベースによってトリガーを起動できます。

- BEFORE LOGOFF

  クライアント・アプリケーションがデータベースからログオフするたびにデータベースによってトリガーを起動できます。

- AFTER SUSPEND

  サーバー・エラーによってトランザクションが停止するたびにデータベースによってトリガーを起動できます。

- AFTER CLONE

  PLUGGABLE DATABASEが指定されている場合にのみ指定できます。PDBのコピー(クローン作成)後に、データベースが新しいPDB内でトリガーを起動した後、トリガーを削除します。トリガーが失敗すると、コピー操作は失敗します。

- BEFORE UNPLUG

  PLUGGABLE DATABASEが指定されている場合にのみ指定できます。PDBの切断前に、データベースがトリガーを起動した後、削除します。トリガーが失敗すると、切断操作は失敗します。

- [BEFORE | AFTER] SET CONTAINER

  ALTER SESSION SET CONTAINER文を実行する前または後のいずれかで、データベースによってトリガーを起動できます。

関連項目:

トリガーを介したデータベース・イベントに対する応答の詳細は、「イベントをパブリッシュするトリガー」を参照してください

[schema.]SCHEMA

指定したスキーマにトリガーを定義します。デフォルト: 現行のスキーマ。指定したスキーマとして接続しているユーザーがトリガー・イベントを開始するたびにトリガーが起動されます。

[PLUGGABLE] DATABASE

DATABASEはルートに対するトリガーを定義します。マルチテナント・コンテナ・データベース(CDB)では、ルートに接続している共通ユーザーのみがデータベース全体に対するトリガーを作成できます。

PLUGGABLE DATABASEは接続しているPDBに対するトリガーを定義します。

指定したデータベースまたはPDBのユーザーがトリガーを起動するイベントを開始するたびにトリガーが起動されます。

ノート:

PDB に接続している場合は、PDB にのみ適用されるオプション(CLONE または UNPLUG など)を指定しないかぎり、DATABASE を指定することは PLUGGABLE DATABASE を指定することと同じです。

ENABLE

(デフォルト)有効な状態でトリガーを作成します。

DISABLE

無効な状態でトリガーを作成すると、有効化する前にトリガーがエラーなしでコンパイルされることを確認できます。

WHEN (condition)

データベースにより評価されるSQL条件を指定します。conditionの値がTRUEの場合は、その行に対してtrigger_bodyが実行されますが、それ以外の場合は、その行に対してtrigger_bodyは実行されません。トリガーを起動する文は、conditionの値に関係なく実行されます。

関連項目:

SQL条件の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

WHEN (condition)の制限

- この句は、STARTUP、SHUTDOWNまたはDB_ROLE_CHANGEトリガーには指定できません。
- この句をSERVERERRORトリガーに指定する場合は、conditionをERRNO = error_codeにする必要があります。
- conditionには、副問合せ、PL/SQL式(ユーザー定義のファンクションの起動など)または相関名を含めることはできません。

trigger_body

データベースがトリガーを起動するために実行するPL/SQLブロックまたはCALLサブプログラム。CALLサブプログラムは、PL/SQLサブプログラムまたはPL/SQLラッパー内のJavaサブプログラムのいずれかです。

trigger_bodyがPL/SQLブロックの場合、ブロック内にエラーがあると、CREATE [OR REPLACE]文は失敗します。

trigger_bodyの制限

- declare_sectionでは、データ型LONGまたはLONG RAWの変数を宣言できません。
- トリガー本体は、:NEWまたは:OLDフィールドは指定できません。

dml_event_clause

simple_dml_triggerまたはcompound_dml_triggerに対して、トリガーを起動する文を指定します。データベースによって、既存のユーザー・トランザクションでトリガーが起動されます。

DELETE

DELETE文でtableから、またはviewが定義されている表から行を削除するたびに、データベースによってトリガーを起動できます。

INSERT

INSERT文でtableに、またはviewが定義されている表に行を追加するたびに、データベースによってトリガーを起動できます。

UPDATE [OF column [, column]]

指定した列の値をUPDATE文で変更するたびにデータベースによってトリガーを起動できます。デフォルト: UPDATE文でtable、またはviewが定義されている表のいずれかの列の値を変更するたびに、データベースによってトリガーを起動できます。

columnを指定する場合、トリガーの本体でその値を変更することはできません。

schema

トリガーが作成されるデータベース・オブジェクトを含むスキーマの名前。デフォルト: 自分のスキーマ。

table

トリガーが作成されるデータベース表またはオブジェクト表の名前。

schema.tableの制限

スキーマSYS内の表に対してトリガーを作成することはできません。

view

トリガーが作成されるデータベース・ビューまたはオブジェクト・ビューの名前。

ノート:

非エディショニング・ビューに対して作成された複合 DML トリガーは、そのタイミング部が 1 つのみであるため、正確には複合トリガーではありません。

referencing_clause

現在の行の新旧の値および親値を参照する相関名を指定します。デフォルト: OLD、NEWおよびPARENT。

トリガーをOLD、NEWまたはPARENTという表に関連付ける場合は、表名と相関名との混乱を回避するために、この句を使用して異なる相関名を指定します。

ネストした表にトリガーが定義されている場合、OLDおよびNEWはネストした表の現在の行を参照し、PARENTは親表の現在の行を参照します。データベースの表またはビューにトリガーが定義されている場合、OLDおよびNEWはデータベースの表またはビューの現在の行を参照し、PARENTは未定義になります。

referencing_clauseの制限

trigger_bodyがCALLルーチンの場合、referencing_clauseは有効ではありません。

DMLの行レベルのトリガーは、USING_NLS_COMP以外の宣言された照合が含まれる列に対応する、OLD/NEW/PARENT疑似レコード(相関名)のフィールドを参照できません。

trigger_edition_clause

トリガーをcrosseditionトリガーとして作成します。

オンライン・アプリケーションのエディション・ベースの再定義(EBR)時に発生したDML変更の処理には複数のステップが伴う可能性があります。したがって、crosseditionトリガーが複合トリガーでもある可能性が高いでしょう(必須ではありません)。

trigger_edition_clauseの制限

- crosseditionトリガーはビューに対しては定義できません。
- crosseditionトリガーに対してNONEDITIONABLEを指定できません。

FORWARD

(デフォルト)forward crosseditionトリガーとしてトリガーを作成します。forward crosseditionトリガーは、データベースを使用するオンライン・アプリケーションにEBRによりパッチ適用またはアップグレードが実行されている間にデータベースにDML変更が加えられた場合に起動するように意図されています。crosseditionトリガーの本体は、このようなDML変更を処理するように設計されているため、アプリケーション・コードに対する変更が完了した後でDML変更を適切に適用できます。

REVERSE

トリガーを、EBRによってパッチ適用またはアップグレードが実行された後のアプリケーションでDML変更が行われると起動される、reverse crosseditionトリガーとして作成します。このトリガーは、パッチ適用またはアップグレードが実行される前にアプリケーションで使用された列または表にデータを伝播します。

関連項目:

crosseditionトリガーの詳細は、『Oracle Database開発ガイド』を参照してください。

trigger_ordering_clause

FOLLOWS | PRECEDES

同じタイミングを持つ複数のトリガーに相対的な起動を指定します。これは、意図した機能を実現するために特定の順序で起動する必要がある、crosseditionトリガーを作成する場合に特に役立ちます。

FOLLOWSを使用すると、作成されるトリガーが、指定したトリガーの後に起動するように指定できます。FOLLOWSは、通常のトリ

ガーまたはforward crosseditionトリガーに指定できます。

PRECEDESを使用すると、作成中のトリガーが、指定したトリガーの前に起動するように指定できます。PRECEDESは、reverse crosseditionトリガーにのみ指定できます。

指定するトリガーが存在し、正常にコンパイルされている必要があります。有効になっている必要はありません。

crosseditionトリガー以外のトリガーを作成する場合、指定するトリガーは次のすべての条件を満たしている必要があります。

- crosseditionトリガー以外のトリガーである
- 作成中のトリガーと同じ表に定義されている。
- 作成中のトリガーと同じエディションで参照できる。

crosseditionトリガーを作成する場合、指定するトリガーは次のすべての条件を満たしている必要があります。

- crosseditionトリガーである
- FOLLOWSまたはPRECEDESを指定しない場合は、作成中のトリガーと同じ表または同じエディショニング・ビューに定義されている。

  FOLLOWSを指定する場合、指定するトリガーはforward crosseditionトリガーである必要があり、PRECEDESを指定する場合、指定するトリガーはreverse crosseditionトリガーである必要があります。ただし、指定するトリガーが、作成中のトリガーと同じ表または同じエディショニング・ビューにある必要はありません。
- 作成中のトリガーと同じエディションで参照できる。

次の定義では、A、B、CおよびDはcrosseditionトリガー以外のトリガーまたはforward crosseditionトリガーのいずれかです。

- BのFOLLOWS句にAが指定されている場合、BはAの直後に起動されます。
- Bの直後にCが起動され、Aの直後にBが起動される場合、CはAの後に間接的に起動されます。
- Cの直後にDが起動され、Aの後に間接的にCが起動される場合、DはAの後に間接的に起動されます。
- Aの直後またはAの後に間接的にBが起動される場合、BはAの後に明示的に起動されます(つまり、BおよびAが起動される順序は1つ以上のFOLLOWS句で明示的に指定されます)。

次の定義では、A、B、CおよびDはreverse crosseditionトリガーです。

- AのPRECEDES句にBが指定されている場合、AはBの直前に起動されます。
- Bの直前にAが起動され、Cの直前にBが起動される場合、AはCの前に間接的に起動されます。
- Bの直前にAが起動され、Dの前に間接的にBが起動される場合、AはDの前に間接的に起動されます。
- Bの直前またはBの前に間接的にAが起動される場合、AはBの前に明示的に起動されます(つまり、AおよびBが起動される順序は1つ以上のPRECEDES句で明示的に指定されます。)

compound_dml_triggerに属します。

compound_trigger_block

非エディショニング・ビューに対してトリガーを作成する場合は、compound_trigger_blockにはINSTEAD OF EACH ROWセクションのみを含める必要があります。

表またはエディショニング・ビューに対してトリガーを作成する場合は、タイミング部は任意の順序で指定できますが、繰り返して使用することはできません。compound_trigger_blockにはINSTEAD OF EACH ROWセクションを含めることはできません。

関連項目:

「複合DMLトリガーの構造」

compound_trigger_blockの制限

compound_trigger_blockのdeclare_sectionにPRAGMA AUTONOMOUS_TRANSACTIONを含めることはできません。

関連項目:

「複合DMLトリガーの制限」

timing_point

BEFORE STATEMENT

表またはエディショニング・ビューに対して、compound_dml_triggerのBEFORE STATEMENTセクションを指定します。このセクションにより、トリガーを起動するイベントを実行する前にデータベースによってトリガーを起動できます。

BEFORE STATEMENTの制限

ここでは、:NEWも:OLDも指定できません。

BEFORE EACH ROW

表またはエディショニング・ビューに対して、compound_dml_triggerのBEFORE EACH ROWセクションを指定します。このセクションにより、トリガーを起動するイベントを実行する前にデータベースによってトリガーを起動できます。影響を受ける各行が変更される前にトリガーが起動されます。

このセクションでは、:OLDおよび:NEWフィールドの読取りと書込みを行えます。

AFTER STATEMENT

表またはエディショニング・ビューに対して、compound_dml_triggerのAFTER STATEMENTセクションを指定します。このセクションにより、トリガーを起動するイベントを実行した後にデータベースによってトリガーを起動できます。

AFTER STATEMENTの制限

ここでは、:NEWも:OLDも指定できません。

AFTER EACH ROW

表またはエディショニング・ビューに対して、compound_dml_triggerのAFTER EACH ROWセクションを指定します。このセクションにより、トリガーを起動するイベントを実行した後にデータベースによってトリガーを起動できます。影響を受ける各行が変更された後にトリガーが起動されます。

このセクションでは、:OLDおよび:NEWフィールドを読み取れますが、書き込めません。

INSTEAD OF EACH ROW

非エディショニング・ビューでcompound_dml_triggerのINSTEAD OF EACH ROWセクション(唯一のタイミング部)を指定します。データベースにより、トリガーを起動するDML文のかわりにtps_bodyが実行されます。詳細は、「INSTEAD OF DMLトリガー」を参照してください。

INSTEAD OF EACH ROWの制限

- このセクションは、非エディショニング・ビューのcompound_dml_triggerにのみ指定できます。
- このセクションでは、:OLDおよび:NEW値を読み取れますが、書き込めません。

tps_body

データベースがトリガーを起動するために実行するPL/SQLブロックまたはCALLサブプログラム。CALLサブプログラムは、PL/SQLサブプログラムまたはPL/SQLラッパー内のJavaサブプログラムのいずれかです。

tps_bodyがPL/SQLブロックの場合、ブロック内にエラーがあると、CREATE [OR REPLACE]文は失敗します。

tps_bodyの制限

declare_sectionでは、データ型LONGまたはLONG RAWの変数を宣言できません。

例

DMLトリガー

- 例9-1、トリガーによる条件述語を使用したトリガーを起動する文の検出
- 例9-2、INSTEAD OFトリガー
- 例9-3、ビューのネストした表の列に対するINSTEAD OFトリガー
- 例9-4、複合トリガーによる、別の表にある表に対する変更のログへの記録
- 例9-5、複合トリガーによる変更表エラーの回避

参照整合性を保証するトリガー

- 例9-6、子表に対する外部キー・トリガー
- 例9-7、親表に対するUPDATEおよびDELETE RESTRICTトリガー
- 例9-8、親表に対するUPDATEおよびDELETE SET NULLトリガー
- 例9-9、親表に対するDELETE CASCADEトリガー
- 例9-10、親表に対するUPDATE CASCADEトリガー
- 例9-11、トリガーによる複雑な制約のチェック
- 例9-12、トリガーによるセキュリティ認可の施行
- 例9-13、トリガーによる新しい列値の導出

相関名および疑似レコードを使用するトリガー

- 例9-14、トリガーによるEMPLOYEES.SALARYの変更の記録
- 例9-15、条件付きトリガーによる給与変更情報の出力
- 例9-16、トリガーによるCLOB列の変更
- 例9-17、REFERENCING句のあるトリガー
- 例9-18、トリガーによるOBJECT_VALUE疑似列の参照

システム・トリガー

- 例9-19、サンプル・スキーマHRに対するBEFORE文トリガー
- 例9-20、データベースに対するAFTER文トリガー
- 例9-21、トリガーによるログオンの監視
- 例9-22、スキーマに対するINSTEAD OF CREATEトリガー

その他のトリガーの例

- 例9-23、トリガーによるJavaサブプログラムの起動
- 例9-24、リモート・データベースが使用できない場合に例外を処理できないトリガー
- 例9-25、リモート・データベースが使用できない場合に例外を処理できないトリガーに対する回避策
- 例9-26、トリガーによる変更表エラー
- 例9-27、更新カスケード
- 例9-28、トリガーに関する情報の表示

関連トピック

この章:

- ALTER TRIGGER文
- DROP TRIGGER文

他の章:

- PL/SQLトリガー

関連項目:

crosseditionトリガーの詳細は、『Oracle Database開発ガイド』を参照してください。

14.13 CREATE TYPE文

CREATE TYPE文は、型の名前、属性、メソッドおよびその他のプロパティを指定します。

CREATE TYPE文は、次のいずれかの仕様を作成するか、または置き換えます。

- 抽象データ型(ADT)
- スタンドアロン型の可変配列(VARRAY)型
- スタンドアロン型のネストした表型
- 不完全なオブジェクト型

  不完全型とは、フォワード型定義によって作成される型です。このオブジェクト型には名前はありますが、属性およびメソッドがないために不完全と呼ばれます。他の型からの参照が可能なため、互いに参照する型を定義できます。ただし、不完全オブジェクト型を使用して表やオブジェクト列またはネストした表型の列を作成する場合は、型を完全に指定しておく必要があります。

CREATE TYPE BODY文には、その型を実装するメソッドに対するコードが含まれます。

ノート:

- 仕様部で属性のみを宣言し、メソッドは宣言しない型を作成する場合、型本体を指定する必要はありません。
- CREATE TYPE 文で作成するスタンドアロン・コレクション型は、PL/SQL ブロックまたはパッケージで TYPE キーワードを使用して定義するコレクション型とは異なります。後者の詳細は、「コレクション変数の宣言」を参照してください。
- CREATE TYPE 文を使用した場合、ネストした表型および VARRAY 型は作成できますが、結合配列は作成できません。PL/SQL ブロックまたはパッケージでは、3 つすべてのコレクション型を定義できます。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

自分のスキーマ内に型を作成するには、CREATE TYPEシステム権限が必要です。他のユーザーのスキーマ内に型を作成する場合は、CREATE ANY TYPEシステム権限が必要です。これらの権限は、明示的に取得することもロールを介して取得することもできます。

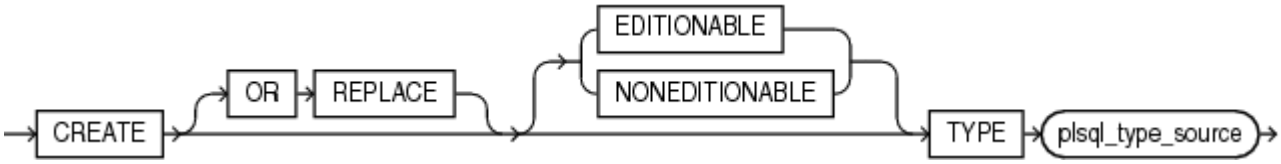
サブタイプを作成するには、UNDER ANY TYPEシステム権限またはスーパータイプに対するUNDERオブジェクト権限が必要です。

型の所有者には、型定義内で参照する他のすべての型にアクセスするためのEXECUTEオブジェクト権限、またはEXECUTE ANY TYPEシステム権限が必要です。所有者は、これらの権限をロールを介して取得することはできません。

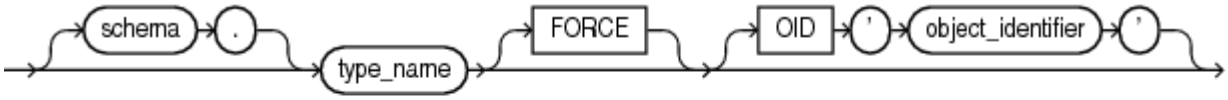
型の所有者が、型にアクセスする権限を他のユーザーに付与する場合、所有者には、参照型に対するGRANT OPTION付きのEXECUTEオブジェクト権限、またはADMIN OPTION付きのEXECUTE ANY TYPEシステム権限が必要です。これらの権限がない場合、型の所有者は、型にアクセスする権限を他のユーザーに付与できません。

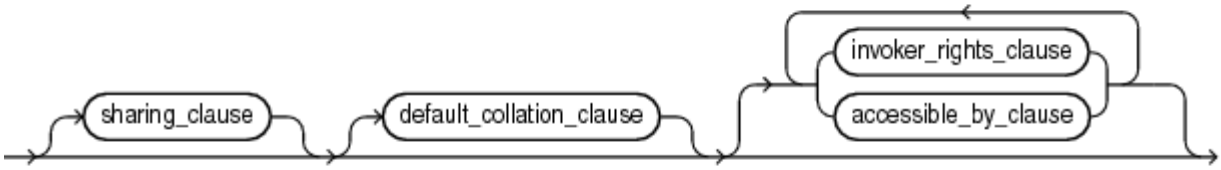
構文

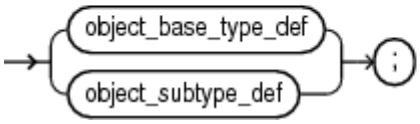
create_type ::=



plsql_type_source ::=

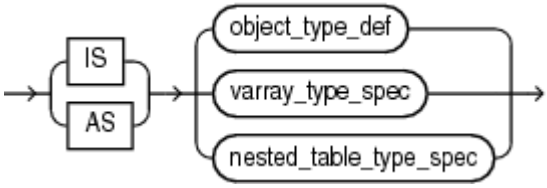






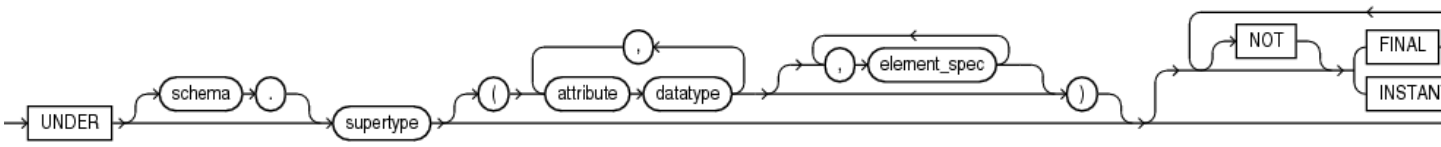
(sharing_clause ::=, default_collation_clause ::= , accessible_by_clause ::= , invoker_rights_clause ::= , object_base_type_def ::= , object_subtype_def ::=)

object_base_type_def ::=



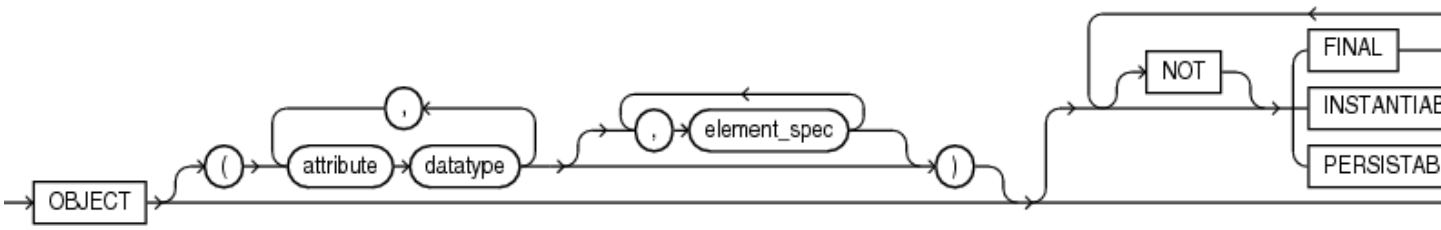
(object_type_def ::=, nested_table_type_spec ::=, varray_type_spec ::=)

object_subtype_def ::=



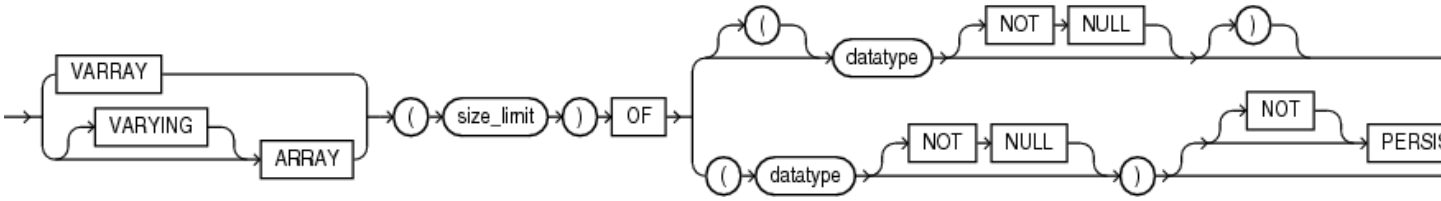
(datatype ::=, element_spec ::=)

object_type_def ::=



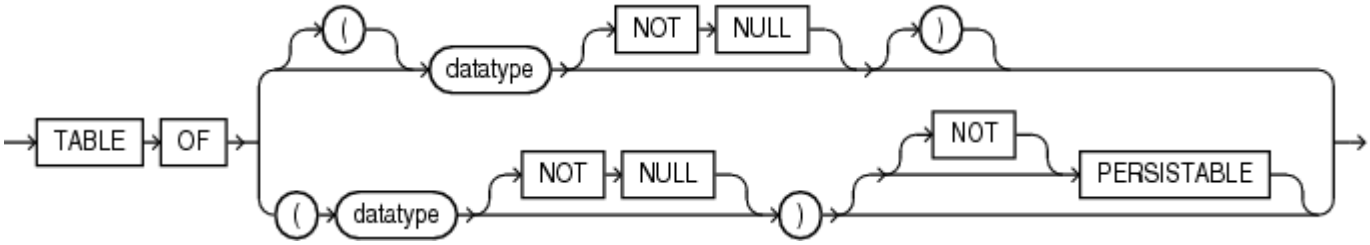
(datatype ::=, element_spec ::=)

varray_type_spec ::=



(datatype ::=)

nested_table_type_spec ::=



(datatype ::=)

セマンティクス

create_type

OR REPLACE

型が存在する場合は、型を再作成し、再コンパイルします。

再定義する前の型に対する権限を付与されていたユーザーは、権限を再付与される必要なく、型にアクセスできます。

ファンクション索引が型に依存している場合、データベースによって索引にDISABLEDのマークが付けられます。

[EDITIONABLE | NONEDITIONABLE]

schemaでスキーマ・オブジェクト・タイプTYPEに対してエディションが有効になっている場合に、型がエディション・オブジェクトまたは非エディション・オブジェクトのどちらになるかを指定します。デフォルト: EDITIONABLE。エディション・オブジェクトと非エディション・オブジェクトの詳細は、『Oracle Database開発ガイド』を参照してください。

plsql_type_source

schema

型が含まれているスキーマの名前。デフォルト: 自分のスキーマ。

type_name

ADT、ネストした表型またはVARRAY型の名前。

型の作成時にコンパイル・エラーが発生した場合は、エラーが戻されます。関連付けられているコンパイラ・エラー・メッセージは、SQL*PlusコマンドSHOW ERRORSを使用して確認できます。

作成した各ユーザー定義型に対して、データベースによって暗黙的にコンストラクタ・メソッドが定義されます。コンストラクタは、型の値のインスタンスを構成するためにSQL文またはPL/SQLコードで使用されるシステム提供のプロシージャです。コンストラクタ・メソッドの名前は、ユーザー定義型の名前です。また、constructor_spec構文を使用して、ユーザー定義コンストラクタを作成することもできます。

ADTのコンストラクタ・メソッドのパラメータは、ADTのデータ属性です。また、そのADT用の属性定義と同じ順序で定義されます。ネストした表またはVARRAYコンストラクタのパラメータは、ネストした表またはVARRAYの要素です。

FORCE

type_nameが存在し、型依存性はあり、表依存性はない場合は、FORCEによって文の型が置き換えられます。(type_nameに表依存性がある場合は、FORCEの有無にかかわらず文は失敗します。)

ノート:

型 t1 に型 t2 との依存性があり、型 t2 に表依存性がある場合は、型 t1 にも表依存性があります。

関連項目:

『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』

OID 'object_identifier'

複数のデータベースで同一オブジェクトの型の等価を設定します。この句の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

object_base_type_def

スキーマ・レベルのADTを作成します。このようなADTは、ルートADTと呼ばれる場合があります。

IS | AS

ADTの作成時には、キーワードISまたはASが必要です。

関連項目:

例14-23

object_subtype_def

既存の型のサブタイプを作成します。

UNDER supertype

既存のスーパータイプは、ADTである必要があります。この文で作成したサブタイプは、スーパータイプのプロパティを継承します。サブタイプではプロパティの一部をオーバーライドするか、またはプロパティを追加して、スーパータイプと区別する必要があります。

関連項目:

例14-24および例14-25

attribute

ADT属性の名前。ADT属性は、ADTの構造を形成する、名前および型指定子を持つデータ項目です。各ADTには、1つ以上の属性を指定する必要があります。名前はそのADT内で一意である必要がありますが、他のADT内では使用できます。

サブタイプを作成する場合、連鎖するスーパータイプで宣言されている属性名またはメソッド名と同じ属性名を指定することはできません。

datatype

ADT属性のデータ型。このデータ型は、データベースに格納されている必要があります(つまり、事前定義のデータ型またはユーザー定義のスタンドアロン・コレクション型のいずれかになります)。

datatypeの制限

- 属性にNOT NULL制約を課すことはできません。
- ROWID型、LONG型またはLONG RAW型の属性は指定できません。
- UROWIDというデータ型は、ADTには指定できません。
- REF型のオブジェクトを指定する場合、ターゲット・オブジェクトにはオブジェクト識別子が必要です。
- ネストした表または表のVARRAY列として使用するためのコレクション型を作成する場合、ANYTYPE型、ANYDATA型またはANYDATASET型の属性を指定することはできません。

object_type_def

ADTを作成します。データ構造を形成する変数は、属性と呼ばれます。ADTの動作を定義するメンバー・サブプログラムは、メソッドと呼ばれます。

OBJECT

キーワードOBJECTは必要です。

[NOT]FINAL、[NOT]INSTANTIABLE、[NOT]PERSISTABLE

これらの句を構文のスキーマ・レベルで指定すると、型の継承する属性を指定できます。

[NOT] FINAL

[NOT] FINAL句を使用すると、この型のサブタイプをさらに作成できるようにするかどうかを指定できます。

- (デフォルト)この型に対してサブタイプをさらに作成できないようにする場合は、FINALを指定します。
- この型に対してサブタイプをさらに作成できるようにする場合は、NOT FINALを指定します。

[NOT] INSTANTIABLE

[NOT] INSTANTIABLE句を使用すると、この型のオブジェクト・インスタンスを作成できるかどうかを構成できます。

- (デフォルト)この型のオブジェクト・インスタンスを構成できる場合は、INSTANTIABLEを指定します。
- このADTに対してデフォルトまたはユーザー定義のコンストラクタが存在しない場合は、NOT INSTANTIABLEを指定します。インスタンス化ができないメソッドを持つ型、および継承された属性またはこの句で指定された属性を持たない型には、これらのキーワードを指定する必要があります。

[NOT] PERSISTABLE

[NOT] PERSISTABLE句を使用して、このオブジェクト型のインスタンスが存続可能であるかどうかを指定します。

表に格納できるのはPERSISTABLE型のみです。

- (デフォルト)オブジェクト型のすべての属性が存続可能である場合、PERSISTABLEを指定できます。存続不可の属性を持つ存続可能オブジェクト型は作成できません。
- オブジェクト型の属性が存続可能または存続不可の場合、NOT PERSISTABLEを指定できます。

[NOT] PERSISTABLEの制限

サブタイプ定義に[NOT] PERSISTABLE句は指定できません。サブタイプの存続可否プロパティはスーパータイプから継承されます。

varray_type_spec

それぞれが同じデータ型を持つ要素の順序付け集合として型を作成します。

varray_type_specの制限

PL/SQLまたはビュー問合せなど、手順上の目的でXMLTypeまたはLOB型のVARRAY型を作成できます。ただし、そのようなVARRAY型のデータベースへの格納はサポートされていないため、そのようなVARRAY型のオブジェクト表または列は作成できません。

関連項目:

例14-26

[NOT] PERSISTABLE

(datatype [NOT NULL])

PERSISTABLEが指定されている場合、datatype [NOT NULL]句の前後にはカッコが必要です。PERSISTABLEが指定されていない場合、カッコの指定は任意です。

[NOT] PERSISTABLE句を使用して、このコレクション型(VARRAYまたはネストされた表)のインスタンスが存続可能であるかどうかを指定します。

- (デフォルト)コレクションの要素の型が存続可能である場合のみ、コレクションをPERSISTABLEにすることができます。存続不可の要素型を持つ存続可能コレクション型は作成できません。
- 存続不可の要素型がコレクションに1つでも含まれている場合は、NOT PERSISTABLEを指定します。NOT PERSISTABLEは、要素の型が存続可能であるかどうかに関係なくすべてのコレクションに指定できます。

nested_table_type_spec

datatype型で名前付きのネストした表を作成します。

[NOT] PERSISTABLE

VARRAYと同様、[NOT] PERSISTABLEを参照してください。

関連項目:

- 例14-27
- 例14-28

例

例14-23 ADTの例

この例は、サンプルの注文入力スキーマ(oe)用にサンプルのcustomer_typ型を作成する方法を示しています。テスト・データベースでこの例を複製できるように、表に仮定の名前が指定されています。

```
CREATE TYPE customer_typ_demo AS OBJECT
    ( customer_id        NUMBER(6)
    , cust_first_name    VARCHAR2(20)
    , cust_last_name     VARCHAR2(20)
    , cust_address       CUST_ADDRESS_TYP
    , phone_numbers      PHONE_LIST_TYP
    , nls_language       VARCHAR2(3)
    , nls_territory      VARCHAR2(30)
    , credit_limit       NUMBER(9,2)
    , cust_email         VARCHAR2(30)
    , cust_orders        ORDER_LIST_TYP
    ) ;
```

次の例では、CREATE TYPE BODY文で実装されるメンバー・ファンクションprodを持つdata_typ1 ADTを作成します。

```
CREATE TYPE data_typ1 AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );
```

```
/

CREATE TYPE BODY data_typ1 IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;
```

例14-24 サブタイプの作成

この文は、サンプル・スキーマoeにサブタイプcorporate_customer_typを作成する方法を示しています。

前述の例で作成したスーパータイプcustomer_typに基づき、account_mgr_id属性を追加しています。テスト・データベースでこの例を複製できるように、表に仮定の名前が指定されています。

```
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
    ( account_mgr_id     NUMBER(6)
    );
```

例14-25 型階層の作成

これらの文により、型階層が作成されます。

employee_t型は、person_t型からnameおよびssn属性を継承し、department_idおよびsalary属性を追加しています。part_time_emp_t型は、employee_tおよび(employee_tを介して)person_tからすべての属性を継承し、num_hrs属性を追加しています。part_time_emp_tはデフォルトで最終型であるため、そのサブタイプを作成できません。

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
```

この型階層を使用して、置換可能な表および置換可能な列を含む表を作成できます。

例14-26 VARRAY型の作成

この文は、サンプル・スキーマoeに、5つの要素を含むVARRAY型phone_list_typを作成する方法を示しています。

テスト・データベースでこの例を複製できるように、表に仮定の名前が指定されています。

```
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
```

例14-27 ネストした表型の作成

サンプル・スキーマpmからのこの例は、textdoc_typ型の表型textdoc_tabを作成します。

```
CREATE TYPE textdoc_typ AS OBJECT
    ( document_typ      VARCHAR2(32)
    , formatted_doc     BLOB
    ) ;
CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
```

例14-28 VARRAYを含むネストした表型の作成

マルチレベル・コレクションのこの例は、サンプルの表oe.customersのバリエーションです。

この例では、cust_addressオブジェクト列は、VARRAY列phone_list_typが埋め込まれたネストした表の列になります。このphone_list_typ_demo型は、「例14-26」で作成された型です。

```
CREATE TYPE cust_address_typ2 AS OBJECT
       ( street_address     VARCHAR2(40)
       , postal_code        VARCHAR2(10)
       , city               VARCHAR2(30)
       , state_province     VARCHAR2(10)
       , country_id         CHAR(2)
       , phone              phone_list_typ_demo
       );
CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ2;
```

例14-29 コンストラクタの例

この例では、システム定義のコンストラクタを起動して、demo_typオブジェクトを構成し、構成したオブジェクトをdemo_tab表に挿入します。

```
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);
INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
```

例14-30 メンバー・メソッドの作成

この例は、メソッド・コンストラクタcol.get_squareを起動します。

まず、型を作成します。

```
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
   MEMBER FUNCTION get_square RETURN NUMBER);
```

次に、ADT列を持つ表を作成し、その表にデータを挿入します。

```
CREATE TABLE demo_tab2(col demo_typ2);
INSERT INTO demo_tab2 VALUES (demo_typ2(2));
```

メンバー・ファンクションを定義する型本体を作成し、メンバー・メソッドを起動します。

```
CREATE TYPE BODY demo_typ2 IS
   MEMBER FUNCTION get_square
   RETURN NUMBER
   IS x NUMBER;
   BEGIN
      SELECT c.col.a1*c.col.a1 INTO x
      FROM demo_tab2 c;
      RETURN (x);
   END;
END;

SELECT t.col.get_square() FROM demo_tab2 t;
T.COL.GET_SQUARE()
------------------
                 4
```

ファンクションとは異なり、メソッドを起動する場合は、メソッドが他に引数を取らない場合でもカッコが必要です。

例14-31 静的メソッドの作成

この例は、employee_t型の定義を変更し、変更した定義をconstruct_empファンクションと関連付けます。

この例では、最初にADT department_tを作成し、次にdepartment_t型の属性を含むADT employee_tを作成します。

```
CREATE OR REPLACE TYPE department_t AS OBJECT (
   deptno number(10),
   dname CHAR(30));
CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept REF department_t,
      STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
      RETURN employee_t
);
```

この文には、次の型本体の文が必要です。

```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
   (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
END;
```

次に、オブジェクト表を作成し、表に挿入します。

```
CREATE TABLE emptab OF employee_t;
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
```

関連トピック

この章:

- ALTER TYPE文
- CREATE TYPE BODY文
- DROP TYPE文

他の章:

- 抽象データ型
- 条件付きコンパイル・ディレクティブの制限
- コレクション変数の宣言
- ユーザー定義のスタンドアロン・コレクション型の詳細は、コレクション型を参照してください。
- PL/SQLのデータ型

その他のドキュメント:

- オブジェクト、不完全型、VARRAYおよびネストした表の詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。
- コンストラクタの詳細は、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

14.14 CREATE TYPE BODY文

CREATE TYPE BODYは、CREATE TYPE文で作成された型の仕様部に定義されたメンバー・メソッドを定義または実装します。

call_specを指定しなかった型の仕様部に指定された各メソッドに対して、対応するメソッド本体を型本体で指定する必要があります。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

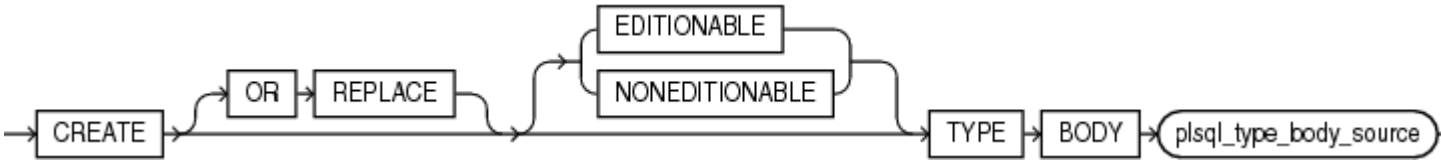
前提条件

ADTのCREATE TYPE仕様部のすべてのメンバー宣言には、対応する構成がCREATE TYPE文またはCREATE TYPE BODY文に必要です。

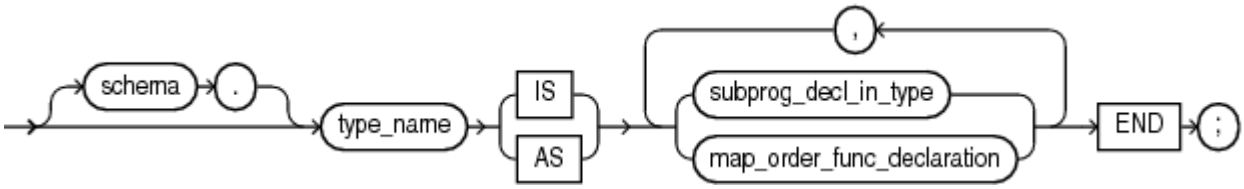
自分のスキーマ内に型本体を作成するか、または自分のスキーマ内の型本体を置き換えるには、CREATE TYPEまたはCREATE ANY TYPEのシステム権限が必要です。他のユーザーのスキーマ内に型を作成する場合は、CREATE ANY TYPEシステム権限が必要です。別のユーザーのスキーマ内の型を置き換えるには、DROP ANY TYPEシステム権限が必要です。

構文

create_type_body ::=

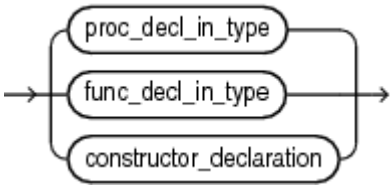


plsql_type_body_source ::=

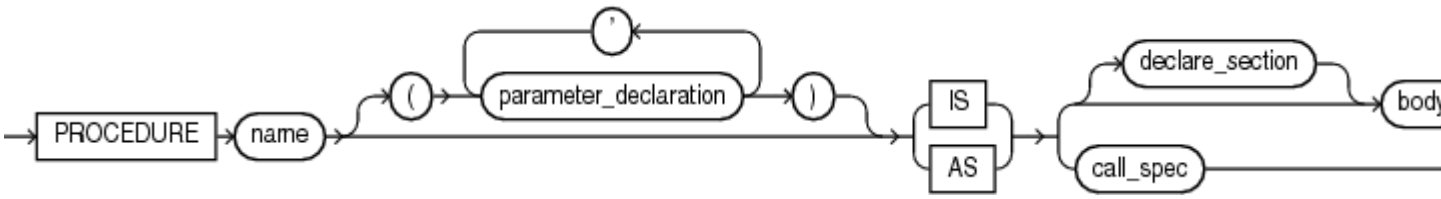


(map_order_func_declaration ::=、subprog_decl_in_type ::=)

subprog_decl_in_type ::=

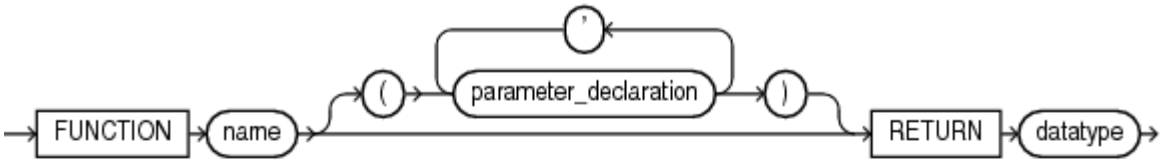


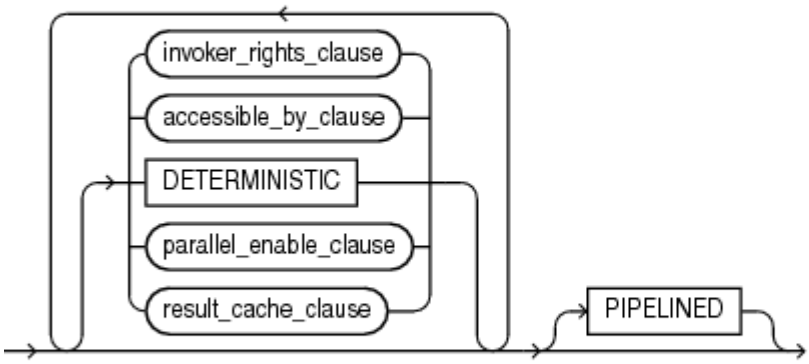
proc_decl_in_type ::=

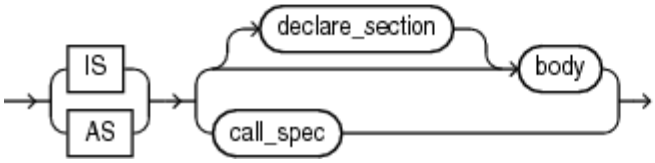


(body ::=、call_spec ::=、declare_section ::=、parameter_declaration ::=)

func_decl_in_type ::=

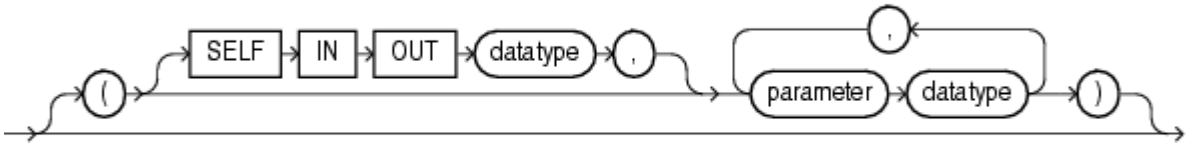


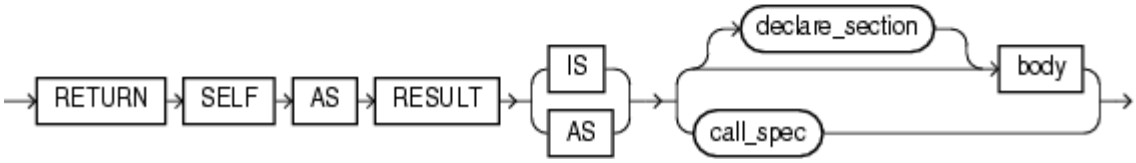




(body ::=、invoker_rights_clause ::=、accessible_by_clause ::=、deterministic_clause ::=、call_spec ::=、declare_section ::=、parameter_declaration ::=、parallel_enable_clause ::=、result_cache_clause ::=、pipelined_clause ::=)

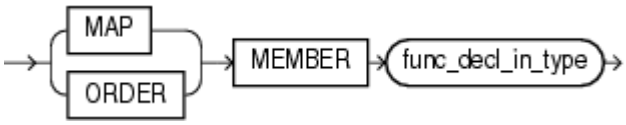
constructor_declaration ::=





(call_spec ::=)

map_order_func_declaration ::=



セマンティクス

create_type_body

OR REPLACE

型本体が存在する場合は、型本体を再作成し、再コンパイルします。

再定義する前の型本体に対する権限を付与されていたユーザーは、権限を再付与される必要なく、型本体にアクセスできます。

この句を使用すると、ALTER TYPE ... REPLACE文で追加された仕様部に、メンバー・サブプログラム定義を追加できます。

[EDITIONABLE | NONEDITIONABLE]

このプロパティを指定しないと、型本体は型仕様部からEDITIONABLEまたはNONEDITIONABLEを継承します。このプロパティを指定する場合は、型仕様部のプロパティと一致する必要があります。

plsql_type_body_source

schema

型本体が含まれているスキーマの名前。デフォルト: 自分のスキーマ。

type_name

ADTの名前。

subprog_decl_in_type

型の仕様部に関連付けられたファンクション・サブプログラムまたはプロシージャ・サブプログラムの型。

プロシージャまたはファンクションの宣言ごとに、対応するメソッド名とオプション・パラメータのリストを型の仕様部に定義する必要があります。ファンクションの場合は、戻り型も指定する必要があります。

map_order_func_declaration

宣言するMEMBERまたはSTATICメソッドの数に関係なく、1つのMAPメソッドまたは1つのORDERメソッドを宣言できます。MAPまたはORDERメソッドを宣言すると、オブジェクト・インスタンスをSQLで比較できます。

いずれのメソッドも宣言しない場合、オブジェクト・インスタンスに関して比較できるのは等しいか等しくないかについてのみです。同じ型定義のインスタンスは、それぞれの対応する属性の各ペアが等しい場合にのみ等しくなります。

MAP MEMBER

オブジェクトのすべてのインスタンスの順序における特定のインスタンスの相対的な位置を戻すMAPメンバー・ファンクションを宣言

または実装します。MAPメソッドは暗黙的にコールされ、事前定義済のスカラー型の値にマップすることによって、オブジェクト・インスタンスの順序を指定します。PL/SQLでは、順序付けを使用してブール式の評価と比較を行います。

MAPメソッドの引数がNULLの場合は、MAPメソッドによってNULLが戻され、メソッドは起動されません。

型本体には1つのMAPメソッドのみを含めることができ、このメソッドはファンクションである必要があります。MAPファンクションに指定できる引数は、暗黙的なSELF引数以外のみです。

ORDER MEMBER

オブジェクトのインスタンスを明示的な引数および暗黙的なSELF引数として取り、負の整数、0(ゼロ)または正の整数のいずれかを戻すORDERメンバー・ファンクションを指定します。戻される値は、暗黙的なSELF引数が明示的な引数より小さいか、等しいかまたは大きいことをそれぞれ示します。

ORDERメソッドのいずれかの引数がNULLの場合は、ORDERメソッドによってNULLが戻され、メソッドは起動されません。

同じADT定義のインスタンスをORDER BY句で比較すると、データベースによってORDER MEMBER func_decl_in_typeが起動されます。

オブジェクトの仕様部には1つのORDERメソッドのみを含めることができ、このメソッドは戻り型NUMBERを持つファンクションである必要があります。

proc_decl_in_type

プロシージャ・サブプログラムの宣言。

constructor_declaration

ユーザー定義のコンストラクタ・サブプログラムの宣言。コンストラクタ・ファンクションのRETURN句は、RETURN SELF AS RESULTにする必要があります。この設定は、コンストラクタ・ファンクションによって戻される値の最も固有の型が、コンストラクタ・ファンクションに渡されたSELF引数の最も固有の型であることを示します。

関連項目:

- ユーザー定義のファンクションの制限のリストは、「CREATE TYPE文」を参照してください
- パッケージ内のサブプログラム名のオーバーロードの詳細は、「オーバーロードされたサブプログラム」を参照してください
- ユーザー定義のコンストラクタの詳細と例については、『Oracle Databaseオブジェクト・リレーショナル開発者ガイド』を参照してください。

declare_section

プロシージャまたはファンクションに対してローカルな項目を宣言します。

body

プロシージャまたはファンクションの文。

func_decl_in_type

ファンクション・サブプログラムの宣言。

例

型本体を作成する例は、「CREATE TYPE文」の「例」を参照してください。型本体の再作成の例は、「例14-7」を参照してください。

関連トピック

- 「CREATE TYPE文」
- 「DROP TYPE BODY文」
- CREATE FUNCTION文
- CREATE PROCEDURE文

14.15 DROP FUNCTION文

DROP FUNCTION文は、データベースからスタンドアロン・ファンクションを削除します。

> ノート:
>
> この文を使用してパッケージの一部であるファンクションを削除しないでください。かわりに、 DROP PACKAGE 文を使用してパッケージ全体を削除するか、または OR REPLACE 句を指定した CREATE PACKAGE 文を使用して、そのファンクションを含めずにパッケージを再定義してください。

ここでのトピック

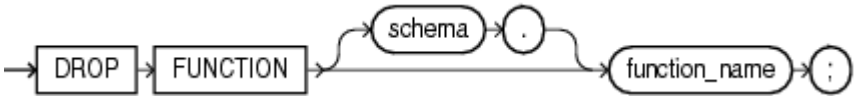
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

ファンクションが自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY PROCEDUREシステム権限が必要です。

構文

drop_function ::=



セマンティクス

drop_function

schema

ファンクションが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

function_name

削除するファンクションの名前。

データベースによって、削除したファンクションに依存するローカル・オブジェクト、または削除したファンクションを起動するローカル・オブジェクトが無効にされます。後でこれらのオブジェクトのいずれかを参照した場合、データベースは、そのオブジェクトを再コンパイルしようとします。削除したファンクションを再作成しないかぎり、エラーが戻されます。

任意の統計タイプがファンクションに関連付けられている場合、FORCE句によって統計タイプの関連付けが解除され、統計タイプ

を使用して収集したユーザー定義のすべての統計情報が削除されます。

関連項目:

- ASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- DISASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例

例14-32 ファンクションの削除

この文は、サンプル・スキーマoe内のファンクションSecondMaxを削除し、SecondMaxに依存していたすべてのオブジェクトを無効にします。

```
DROP FUNCTION oe.SecondMax;
```

関連項目:

SecondMaxファンクションの作成の詳細は、「例14-15」を参照してください

関連トピック

- ALTER FUNCTION文
- CREATE FUNCTION文

14.16 DROP LIBRARY文

DROP LIBRARY文は、データベースから外部プロシージャ・ライブラリを削除します。

ここでのトピック

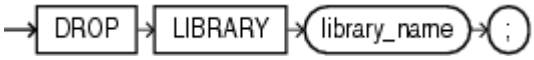
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

DROP ANY LIBRARYシステム権限が必要です。

構文

drop_library ::=



セマンティクス

library_name

削除される外部プロシージャ・ライブラリの名前。

例

例14-33 ライブラリの削除

次に示す文では、「CREATE LIBRARY文」で作成したext_libライブラリを削除します。

```
DROP LIBRARY ext_lib;
```

関連トピック

- ALTER LIBRARY文
- CREATE LIBRARY文

14.17 DROP PACKAGE文

DROP PACKAGE文は、データベースからストアド・パッケージを削除します。

この文は、パッケージの本体および仕様部を削除します。

> ノート:
>
> この文を使用してパッケージから単一のオブジェクトを削除しないでください。かわりに、 REATE PACKAGE 文および CREATE PACKAGE BODY 文に OR REPLACE 句を指定して、そのオブジェクトを含めずにパッケージを再作成してください。

ここでのトピック

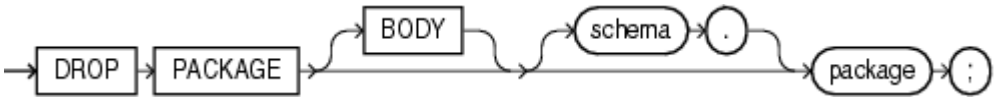
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

パッケージが自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY PROCEDUREシステム権限が必要です。

構文

drop_package ::=



セマンティクス

drop_package

BODY

パッケージ本体のみを削除します。この句を省略すると、パッケージの本体と仕様部の両方が削除されます。

パッケージ本体のみを削除して仕様部は削除しない場合、依存するオブジェクトは無効になりません。ただし、パッケージ本体を再作成するまで、パッケージ仕様部で宣言されたプロシージャまたはストアド・ファンクションは起動できません。

schema

パッケージが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

package

削除するパッケージの名前。

データベースによって、パッケージ仕様部に依存するすべてのローカル・オブジェクトが無効にされます。後でこれらのオブジェクトのいずれかを参照した場合、データベースは、そのオブジェクトを再コンパイルしようとします。削除したパッケージを再作成しないかぎり、エラーが戻されます。

任意の統計タイプがパッケージに関連付けられている場合、FORCE句によって統計タイプの関連付けが解除され、統計タイプを使用して収集されたユーザー定義のすべての統計情報が削除されます。

関連項目:

- ASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- DISASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

例

例14-34 パッケージの削除

この文では、「CREATE PACKAGE BODY文」で作成したemp_mgmtパッケージの仕様部と本体を削除します。その仕様部に依存しているオブジェクトは、すべて無効になります。

```
DROP PACKAGE emp_mgmt;
```

関連トピック

- ALTER PACKAGE文
- CREATE PACKAGE文
- 「CREATE PACKAGE BODY文」

14.18 DROP PROCEDURE文

DROP PROCEDURE文は、データベースからスタンドアロン・プロシージャを削除します。

> ノート:
>
> この文を使用してパッケージの一部であるプロシージャを削除しないでください。かわりに、「DROP PACKAGE 文」を使用してパッケージ全体を削除するか、または OR REPLACE 句を指定した「CREATE PACKAGE 文」を使用して、そのプロシージャを含めずにパッケージを再定義してください。

ここでのトピック

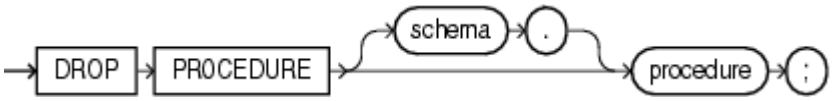
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

プロシージャが自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY PROCEDUREシステム権限が必要です。

構文

drop_procedure ::=



セマンティクス

schema

プロシージャが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

procedure

削除するプロシージャの名前。

プロシージャを削除すると、データベースによって、削除したプロシージャに依存するすべてのローカル・オブジェクトが無効にされます。後でこれらのオブジェクトのいずれかを参照した場合、データベースは、そのオブジェクトを再コンパイルしようとします。削除したプロシージャを再作成しないかぎり、エラー・メッセージが戻されます。

例

例14-35 プロシージャの削除

この文は、ユーザーhrが所有するプロシージャremove_empを削除し、remove_empに依存するすべてのオブジェクトを無効にします。

```
DROP PROCEDURE hr.remove_emp;
```

関連トピック

- ALTER PROCEDURE文
- CREATE PROCEDURE文

14.19 DROP TRIGGER文

DROP TRIGGER文は、データベースからデータベース・トリガーを削除します。

ここでのトピック

- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

削除するトリガーが自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY TRIGGERシステム権限が必要です。他のユーザーのスキーマ内のDATABASEにあるトリガーを削除する場合は、ADMINISTER DATABASE TRIGGERシステム権限も必要です。

構文

drop_trigger ::=

セマンティクス

schema

トリガーが含まれているスキーマの名前。デフォルト: 自分のスキーマ。

trigger

削除するトリガーの名前。

例

例14-36 トリガーの削除

この文は、スキーマhr内のsalary_checkトリガーを削除します。

```
DROP TRIGGER hr.salary_check;
```

関連トピック

- ALTER TRIGGER文
- CREATE TRIGGER文

14.20 DROP TYPE文

DROP TYPE文は、ADT、VARRAY型またはネストした表型の仕様部および本体を削除します。

ここでのトピック

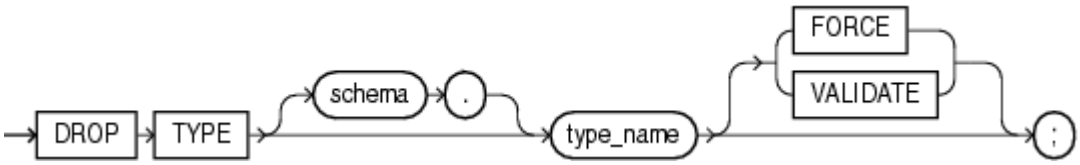
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

ADT、VARRAY型またはネストした表型が自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY TYPEシステム権限が必要です。

構文

drop_type ::=



セマンティクス

schema

型が含まれているスキーマの名前。デフォルト: 自分のスキーマ。

type_name

削除するオブジェクト型、VARRAY型またはネストした表型の名前。型依存性または表依存性のない型のみ削除できます。

type_nameがスーパータイプの場合、FORCEも指定しないかぎり、この文は正常に実行されません。FORCEを指定すると、そのスーパータイプに依存するすべてのサブタイプが無効になります。

type_nameが統計タイプの場合、FORCEも指定しないかぎり、この文は正常に実行されません。FORCEを指定した場合、最初にtype_nameに関連付けられたすべてのオブジェクトの関連付けが解除され、type_nameが削除されます。

関連項目:

- ASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。
- DISASSOCIATE STATISTICS文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

type_nameが統計タイプに関連付けられたADTの場合は、まずこの統計タイプからtype_nameの関連付けが解除され、次に

type_nameが削除されます。ただし、統計タイプを使用して統計情報が収集された場合は、データベースでこの統計タイプからtype_nameの関連付けを解除することはできないため、この文は正常に実行されません。

type_nameが索引タイプの実装タイプの場合は、索引タイプにINVALIDのマークが付けられます。

type_nameにパブリック・シノニムが定義されている場合は、このシノニムも削除されます。

FORCEを指定しないかぎり、削除できるのは、依存性のないスタンドアロンのスキーマ・オブジェクトとして定義されている型のみです。これはデフォルトの動作です。

関連項目:

CREATE INDEXTYPE文の詳細は、『Oracle Database SQL言語リファレンス』を参照してください。

FORCE

依存するデータベース・オブジェクトが型に含まれている場合でもその型を削除します。削除する型に依存するすべての列がデータベースによってUNUSEDとマークされ、それらの列へのアクセスができなくなります。

ノート:

FORCE を使用した、依存性のあるオブジェクト型の削除は行わないことをお薦めします。この操作はリカバリ不能であり、依存表または列のデータにアクセスできなくなる場合があります。

VALIDATE

この型のインスタンスがスーパータイプのいずれかの置換可能な列に格納されていることをデータベースによって確認できます。このようなインスタンスがない場合、削除操作が完了します。

この句はサブタイプのみに意味があります。明示的な型または表依存性のないサブタイプを安全に削除するために、このオプションの使用をお薦めします。

例

例14-37 ADTの削除

この文は、ADT person_tを削除します。このADTの作成例は、「CREATE TYPE文」を参照してください。person_tに依存するすべての列に、UNUSEDのマークが付けられ、アクセスできなくなります。

```
DROP TYPE person_t FORCE;
```

関連トピック

- ALTER TYPE文
- 「CREATE TYPE文」
- 「CREATE TYPE BODY文」

14.21 DROP TYPE BODY文

DROP TYPE BODY文は、ADT、VARRAY型またはネストした表型の本体を削除します。

型本体を削除しても、型の仕様部は残り、型本体は再作成できます。本体の再作成前でも、型は引き続き使用できますが、そのメンバー・ファンクションは起動できません。

ここでのトピック

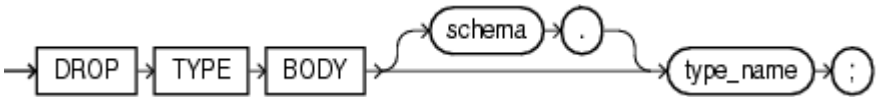
- 前提条件
- 構文
- セマンティクス
- 例
- 関連トピック

前提条件

型本体が自分のスキーマ内にある必要があります。自分のスキーマ内にない場合は、DROP ANY TYPEシステム権限が必要です。

構文

drop_type_body ::=



セマンティクス

schema

型が含まれているスキーマの名前。デフォルト: 自分のスキーマ。

type_name

削除する型本体の名前。

type_nameの制限

型依存性または表依存性のない型本体のみ削除できます。

例

例14-38 ADT本体の削除

この文は、ADT本体data_typ1を削除します。このADTの作成例は、「CREATE TYPE文」を参照してください。

```
DROP TYPE BODY data_typ1;
```

関連トピック

- ALTER TYPE文

- 「CREATE TYPE文」
- 「CREATE TYPE BODY文」

A PL/SQLのソース・テキストのラップ

次の任意のストアドPL/SQLユニットのPL/SQLソース・テキストをラップすることによって、他のユーザーが静的データ・ディクショナリ・ビュー*_SOURCEを使用して、他のユーザーがそのテキストを表示することを防止できます。

- パッケージ仕様部
- パッケージ本体
- 型仕様部
- 型本体
- ファンクション
- プロシージャ

> ノート:
>
> テキストのラップのセキュリティは高くありません。高いセキュリティを確保するには、Oracle Database Vault を使用してください(『Oracle Database Vault 管理者ガイド』を参照)。

ラップされたPL/SQLソース・テキストが含まれるファイルは、ラップされたファイルと呼ばれます。ラップされたファイルは、SQL*Plusによって、またはインポート・ユーティリティおよびエクスポート・ユーティリティによって移動、バックアップまたは処理できます。

ラップされたファイルを生成するには、PL/SQLラッパー・ユーティリティまたはDBMS_DDLサブプログラムを使用します。PL/SQLラッパー・ユーティリティは、指定のSQLファイルによって作成されたすべてのラップ可能PL/SQLユニットのソース・テキストをラップします。DBMS_DDLサブプログラムは、動的に生成された単一のラップ可能PL/SQLユニットのソース・テキストをラップします。

PL/SQLラッパー・ユーティリティおよびDBMS_DDLサブプログラムでは、トークン化エラー(文字列の欠落など)は検出されますが、構文エラーやセマンティック・エラー(表やビューが存在しないなど)は検出されません。

12.2 PL/SQLコンパイラは、デフォルトで、9.2 PL/SQLコンパイラでコンパイルされているラップされたパッケージを使用できます。12.2 PL/SQLコンパイラが9.2 PL/SQLコンパイラでコンパイルされているラップされたパッケージを使用できないようにするには、PL/SQLコンパイル・パラメータPERMIT_92_WRAP_FORMATをFALSEに設定します。PERMIT_92_WRAP_FORMATの詳細は、『Oracle Databaseリファレンス』を参照してください。PL/SQLのコンパイル・パラメータの詳細は、「PL/SQLユニットおよびコンパイル・パラメータ」を参照してください。

ここでのトピック

- PL/SQLソース・テキストのラップの制限
- PL/SQLソース・テキストのラップのガイドライン
- PL/SQLラッパー・ユーティリティによるPL/SQLソース・テキストのラップ
- DBMS_DDLサブプログラムによるPL/SQLソース・テキストのラップ

A.1 PL/SQLソース・テキストのラップの制限

- ラップされたファイルは、Oracle Databaseリリース間での下位互換性がありません。

  たとえば、バージョンn.1 PL/SQL Wrapperユーティリティにより生成されたファイルをバージョン(n-1).2 Oracle Databaseにロードすることはできません。また、バージョンn.2 PL/SQL Wrapperユーティリティにより生成されたファイルをバージョンn.1 Oracle Databaseにロードすることもできません。ラップされたファイルは、パッチセット間で上位および下位の両方の互換性があります。

- PL/SQLソース・テキストのラップは、パスワードや表名を隠すための安全な方法とはいえません。

  高いセキュリティを確保するには、Oracle Database Vaultを使用してください(『Oracle Database Vault管理者ガイド』を参照)。

- トリガーのPL/SQLソース・テキストは、ラップできません。

  トリガーの実装の詳細を隠すには、実装の詳細をストアド・サブプログラムに配置し、そのサブプログラムをラップして、そのサブプログラムを起動する1行トリガーを記述します。

A.2 PL/SQLソース・テキストのラップのガイドライン

- パッケージまたは型の仕様部ではなく、本体のみをラップします。

  仕様部をラップしないことで、他の開発者は、パッケージまたは型を使用するために必要な情報を見ることができます(例A-5を参照)。本体はラップされているため、パッケージまたは型の実装は、他の開発者から隠されます。

- ファイルの編集が完了した後にのみ、ファイルをラップします。

  ラップされたファイルは、編集できません。ラップされたファイルの変更が必要な場合、ラップされていない元のファイルを編集し、再度ラップする必要があります。

- ラップされたファイルを配布する前に、テキスト・エディタで表示して重要な部分がすべてラップされていることを確認します。

A.3 PL/SQLラッパー・ユーティリティによるPL/SQLソース・テキストのラップ

PL/SQLラッパー・ユーティリティは、単一のSQLファイル(SQL*Plusスクリプトなど)を取得して、各ラップ可能PL/SQLユニットのPL/SQLソース・テキストがラップされた同等のテキスト・ファイルを生成します。

> ノート:
>
> PL/SQL ラッパー・ユーティリティは、バージョン 10 以降を使用することをお薦めします。

ラップ可能PL/SQLユニットのリストは、「PL/SQLのソース・テキストのラップ」の概要を参照してください。

PL/SQLラッパー・ユーティリティは、Oracle Databaseに接続できません。PL/SQLラッパー・ユーティリティを実行するには、オペレーティング・システム・プロンプトで(等号の前後に空白を入れずに)次のコマンドを入力します。

```
wrap iname=input_file [ oname=output_file ] [ keep_comments=yes ]
```

input_fileは、複数のSQL文の任意の組合せを含む既存のファイルの名前です。output_fileは、PL/SQLラッパー・ユーティリティによって作成されるファイル(ラップされたファイル)の名前です。

ノート:

output_file は、SQL*Plus ではなく PL/SQL コンパイラによって解析されるため、input_file に SQL*Plus の DEFINE 表記法で指定された置換変数を含めることはできません。

PL/SQLラッパー・ユーティリティは、keep_comments=yesが指定されていないかぎり、ラップされたファイルからすべてのコメントを削除します。keep_comments=yesが指定されている場合、ソース外のコメントのみが維持されます。

ノート:

input_file がラップされたファイルの場合、input_file と output_file の内容は同じになります。

input_fileのデフォルトのファイル拡張子は、sqlです。output_fileのデフォルト名は、input_file.plbです。したがって、次のコマンドは同じ意味を持ちます。

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

次の例では、input_fileに異なるファイル拡張子を指定し、output_fileに異なる名前を指定します。

```
wrap iname=/mydir/myfile.src oname=/yourdir/yourfile.out keep_comments=yes
```

output_fileは、SQL*Plusのスクリプトとして実行できます。たとえば:

```
SQL> @myfile.plb;
```

例A-1 2つのラップ可能PL/SQLユニットを含むSQLファイル

この例では、2つのラップ可能PL/SQLユニット(プロシージャwraptestおよびファンクションfibonacci)を含むSQLファイルwraptest2.sqlのテキストを示します。このファイルには、コメントのほか、SQLのSELECT文も含まれます。

```
-- The following statement will not change.

SELECT COUNT(*) FROM EMPLOYEES
/

/* The PL/SQL source text of the following two CREATE statements will be wrapped. */

CREATE PROCEDURE wraptest AUTHID CURRENT_USER /* C style comment in procedure declaration */ IS
  TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
  all_emps  emp_tab;
BEGIN
  SELECT * BULK COLLECT INTO all_emps FROM employees;
  FOR i IN 1..10 LOOP /* C style in pl/sql source */
    DBMS_OUTPUT.PUT_LINE('Emp Id: ' || all_emps(i).employee_id);
  END LOOP;
END;
/

CREATE OR REPLACE FUNCTION fibonacci (
```

```
  n PLS_INTEGER
) RETURN PLS_INTEGER
AUTHID CURRENT_USER -- PL/SQL style comment inside fibonacci function spec
IS
  fib_1 PLS_INTEGER := 0;
  fib_2 PLS_INTEGER := 1;
BEGIN
  IF n = 1 THEN                              -- terminating condition
    RETURN fib_1;
  ELSIF n = 2 THEN
    RETURN fib_2;                           -- terminating condition
  ELSE
    RETURN fibonacci(n-2) + fibonacci(n-1);  -- recursive invocations
  END IF;
END;
/
```

例A-2 PL/SQLラッパー・ユーティリティによるファイルのラップ

この例では、PL/SQLラッパー・ユーティリティを使用してwraptest2.sqlをラップし、そのラップされたファイルwraptest2.plbを表示します。ラップされたファイルでは、ユーティリティによってコード内のコメントが削除され、プロシージャwraptestおよびファンクションfibonacciのPL/SQLソース・テキストがラップされた(判読不可能にされた)ことがわかりますが、ラップされたソース外のコメントは維持されています。

オペレーティング・システム・プロンプトが>であると仮定。ファイルwraptest.sqlのラップ:

```
> wrap keep_comments=yes iname=wraptest2.sql
```

結果:

```
 Processing wraptest2.sql to wraptest2.plb
```

wraptest.plbの内容:

```
-- The following statement will not change.

SELECT COUNT(*) FROM EMPLOYEES
/

/* The PL/SQL source text of the following two CREATE statements will be wrapped. */
CREATE OR REPLACE PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
```

```
abcd
abcd
7
129 138
qf4HggDBeNMPlWAsPn6pGf+2LGwwg+nwJK5qZ3SVWE4+GayDZaL1bF7RwYm2/zr1qjZY3FrN
48M1bKc/MG5aY9YB+DrtT4SJN37ORpq7ck5DOsc1D5sKAwTyX13HYvRmjwkdXaOvEZ4q/mCU
EQusX23UZbZjxha7CtlCDCx8guGw/M/oHZXc8wDHXL8V8OsqQMv/Hj7z68gINl7OstalRScr
uSZ/l/W1YaaA9Lj8Fbx5/nJw96ZNy1SCY8VsB/G6O5f/65+EDxdThpnfU4e1vrrE9iB3/IpI
+7fE1Tv29fwc+aZq3S7O
/

CREATE OR REPLACE FUNCTION fibonacci wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
8
150 ff
BFDvTL9ORO4SJbx+qOy5H/h8IcwwgxDcAJnWZ3TNz51mjAmegdQcpNJfq8hUuQtv1Y5xg7Wd
KqMH/HBANhnZ+E1mBWekavYjPxlqV9zIFqZAgB4SBqkqe42sai9VbOcLEUO2/ZCEyxDSfWf3
H1Lp6U9ztRXNy+oDZSNykWCUVLaZroOUmeFrNUBqzE6j9mI3AyRhPw1QbZX5oRMLgLOG3OtS
SGJsz7M+bnhnp+xP4ww+SIlxx5LhDtnyPw==
/
```

例A-3 ラップされたファイルの実行およびラップされたPL/SQLユニットの表示

この例では、ラップされたファイルwraptest.plbをSQL*Plusで実行し、プロシージャwraptestおよびファンクションfibonacciを作成します。各サブプログラムのテキスト(ラップされているため判読不可能)を選択し、その後、サブプログラムを起動します。

```
SQL> -- Run wrapped file:
SQL>
SQL> @wraptest2.plb
SQL> -- The following statement will not change.
SQL>
SQL> SELECT COUNT(*) FROM EMPLOYEES
  2  /

  COUNT(*)
----------
       107

1 row selected.
SQL> /* The PL/SQL source text of the following two CREATE statements will be wrapped. */
```

```
SQL> CREATE PROCEDURE wraptest wrapped
   2  a000000
  3  1
  4  abcd
  5  abcd
  6  abcd
  7  abcd
  8  abcd
  9  abcd
 10  abcd
 11  abcd
 12  abcd
 13  abcd
 14  abcd
 15  abcd
 16  abcd
 17  abcd
 18  abcd
 19  7
 20  129 138
 21  qf4HggDBeNMPlWAsPn6pGf+2LGwwg+nwJK5qZ3SVWE4+GayDZaL1bF7RwYm2/zr1qjZY3FrN
 22  48M1bKc/MG5aY9YB+DrtT4SJN37ORpq7ck5DOsc1D5sKAwTyX13HYvRmjwkdXaOvEZ4q/mCU
 23  EQusX23UZbZjxha7CtlCDCx8guGw/M/oHZXc8wDHXL8V8OsqQMv/Hj7z68gINl7OstalRScr
 24  uSZ/l/W1YaaA9Lj8Fbx5/nJw96ZNy1SCY8VsB/G6O5f/65+EDxdThpnfU4e1vrrE9iB3/IpI
 25  +7fE1Tv29fwc+aZq3S7O
 26
 27  /

Procedure created.

SQL> CREATE OR REPLACE FUNCTION fibonacci wrapped
 2  a000000
  3  1
  4  abcd
  5  abcd
  6  abcd
  7  abcd
  8  abcd
  9  abcd
 10  abcd
 11  abcd
 12  abcd
 13  abcd
 14  abcd
 15  abcd
 16  abcd
 17  abcd
 18  abcd
 19  8
 20  150 ff
 21  BFDvTL9ORO4SJbx+qOy5H/h8IcwwgxDcAJnWZ3TNz51mjAmegdQcpNJfq8hUuQtv1Y5xg7Wd
 22  KqMH/HBANhnZ+E1mBWekavYjPxlqV9zIFqZAgB4SBqkqe42sai9VbOcLEUO2/ZCEyxDSfWf3
 23  H1Lp6U9ztRXNy+oDZSNykWCUVLaZroOUmeFrNUBqzE6j9mI3AyRhPw1QbZX5oRMLgLOG3OtS
 24  SGJsz7M+bnhnp+xP4ww+SIlxx5LhDtnyPw==
 25
 26  /

Function created.
```

```
SQL>
SQL> -- Try to display procedure source text:
SQL>
SQL> SELECT text FROM USER_SOURCE WHERE name='WRAPTEST';

TEXT
--------------------------------------------------------------------------------
PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
129 138
qf4HggDBeNMPlWAsPn6pGf+2LGwwg+nwJK5qZ3SVWE4+GayDZaL1bF7RwYm2/zr1qjZY3FrN
48M1bKc/MG5aY9YB+DrtT4SJN37ORpq7ck5DOsc1D5sKAwTyX13HYvRmjwkdXaOvEZ4q/mCU
EQusX23UZbZjxha7CtlCDCx8guGw/M/oHZXc8wDHXL8V8OsqQMv/Hj7z68gINl7OstalRScr
uSZ/l/W1YaaA9Lj8Fbx5/nJw96ZNy1SCY8VsB/G6O5f/65+EDxdThpnfU4e1vrrE9iB3/IpI
+7fE1Tv29fwc+aZq3S7O


1 row selected.

SQL>
SQL> -- Try to display function source text:
SQL>
SQL> SELECT text FROM USER_SOURCE WHERE name='FIBONACCI';

TEXT
--------------------------------------------------------------------------------
FUNCTION fibonacci wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
```

```
abcd
abcd
abcd
abcd
8
150 ff
BFDvTL9ORO4SJbx+qOy5H/h8IcwwgxDcAJnWZ3TNz51mjAmegdQcpNJfq8hUuQtv1Y5xg7Wd
KqMH/HBANhnZ+E1mBWekavYjPxlqV9zIFqZAgB4SBqkqe42sai9VbOcLEUO2/ZCEyxDSfWf3
H1Lp6U9ztRXNy+oDZSNykWCUVLaZroOUmeFrNUBqzE6j9mI3AyRhPw1QbZX5oRMLgLOG3OtS
SGJsz7M+bnhnp+xP4ww+SIlxx5LhDtnyPw==


1 row selected.

SQL>
SQL> BEGIN
  2    wraptest;  -- invoke procedure
  3    DBMS_OUTPUT.PUT_LINE('fibonacci(5) = ' || fibonacci(5));
  4  END;
  5  /
Emp Id: 100
Emp Id: 101
Emp Id: 102
Emp Id: 103
Emp Id: 104
Emp Id: 105
Emp Id: 106
Emp Id: 107
Emp Id: 108
Emp Id: 109
fibonacci(5) = 3

PL/SQL procedure successfully completed.

SQL>
```

A.4 DBMS_DDLサブプログラムによるPL/SQLソース・テキストのラップ

DBMS_DDLパッケージには、WRAPファンクションおよびCREATE_WRAPPEDプロシージャが含まれていて、各ファンクションおよびプロシージャは、動的に生成された単一のラップ可能PL/SQLユニットのPL/SQLソース・テキストをラップします。また、DBMS_DDLパッケージには、WRAPまたはCREATE_WRAPPEDへの入力が有効なラップ可能PL/SQLユニットでない場合に発生する、例外MALFORMED_WRAP_INPUT(ORA-24230)も含まれます。(ラップ可能PL/SQLユニットのリストは、「PL/SQLのソース・テキストのラップ」の概要を参照してください。)

各WRAPファンクションは、ラップ可能PL/SQLユニットを作成する単一のCREATE文を入力として取得し、PL/SQLソース・テキストがラップされた同等のCREATE文を戻します。WRAPファンクションの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

注意:

DBMS_DDL.WRAP によって戻される文を、データ型が VARCHAR2A の仮パラメータ statement を含む

DBMS_SQL.PARSE プロシージャに渡す場合、DBMS_SQL.PARSE の lfflg パラメータを FALSE に設定する必要があります。設定しないと、DBMS_SQL.PARSE によって、ラップされた PL/SQL ユニットに行が追加され、ユニットが破損します。(DBMS_SQL.PARSE の構文は、『Oracle Database PL/SQL パッケージおよびタイプ・リファレンス』を参照してください。)

各CREATE_WRAPPEDプロシージャは、その対応するWRAPファンクションと同じ処理を行って、戻されたCREATE文を実行し、指定されたPL/SQLユニットを作成します。CREATE_WRAPPEDプロシージャの詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。

ヒント:

DBMS_DDL サブプログラムの起動時には、パッケージの完全修飾名 SYS.DBMS_DDL を使用し、他のユーザーが DBMS_DDL という名前のローカル・パッケージを作成していたり、パブリック・シノニム DBMS_DDL を定義している場合に発生する名前の競合を回避します。

ノート:

WRAP ファンクションまたは CREATE_WRAPPED プロシージャに入力される CREATE 文は、そのサブプログラムを起動したユーザーの権限で実行されます。

例A-4では、EXECUTE IMMEDIATE文を使用してパッケージ仕様部を、およびCREATE_WRAPPEDプロシージャを使用してラップされたパッケージ本体を動的に作成します。

例A-5では、例A-4で作成したパッケージemp_actionsのテキストを選択し、プロシージャemp_actions.raise_salaryを起動します。パッケージ仕様部がラップされている場合、プロシージャを起動するために必要な情報は、パッケージ本体のPL/SQLソース・テキストのように判読不可能になります。

例A-4 CREATE_WRAPPEDプロシージャによるラップされたパッケージ本体の作成

```
DECLARE
  package_text  VARCHAR2(32767); -- text for creating package spec and body

  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AUTHID CURRENT_USER AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
      PROCEDURE fire_employee (emp_id NUMBER);
      END ' || pkgname || ';';
  END generate_spec;

  FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
    RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
      BEGIN
        UPDATE employees
          SET salary = salary + amount WHERE employee_id = emp_id;
      END raise_salary;
```

```
      PROCEDURE fire_employee (emp_id NUMBER) IS
      BEGIN
        DELETE FROM employees WHERE employee_id = emp_id;
      END fire_employee;
    END ' || pkgname || ';';
  END generate_body;

BEGIN
  package_text := generate_spec('emp_actions');  -- Generate package spec
  EXECUTE IMMEDIATE package_text;                 -- Create package spec
  package_text := generate_body('emp_actions');  -- Generate package body
  SYS.DBMS_DDL.CREATE_WRAPPED(package_text);     -- Create wrapped package body
END;
/
```

例A-5 ラップされた本体を含むパッケージの表示およびパッケージ・プロシージャの起動

パッケージのテキストの選択:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

結果:

```
TEXT
------------------------------------------------------------------------

PACKAGE emp_actions AUTHID CURRENT_USER AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
      PROCEDURE fire_employee (emp_id NUMBER);
      END emp_actions;
PACKAGE BODY emp_actions wrapped
a000000
1f
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
b
180 113
1fOVodewm7j9dBOmBsiEQzOBKCgwg/BKoZ4VZy/pTBIYo8Uj1sjpbEzO8Ck3HMjYq/MfOXZn
u9DOKd+i89g9ZO61I6vZYjw2AuBidnLESyR63LHZpFD/7lyDTfF1eDY5vmNwLTXrFaxGy243
OlHKAzmOlwwfBWylkZZNi2UnpmSIe6z/BU2nhbwfpqd224p69FwYVXmFX2H5IMsdZ2/vWsK9
cDMCD1KEqOnPpbU2yXdpW3GIbGD8JFIbKAfpJLkoLfVxoRPXQfjOh1k=
```

raised_salaryの起動およびその影響の表示:

```
DECLARE
  s employees.salary%TYPE;
BEGIN
  SELECT salary INTO s FROM employees WHERE employee_id=130;
  DBMS_OUTPUT.PUT_LINE('Old salary: ' || s);
  emp_actions.raise_salary(130, 100);
  SELECT salary INTO s FROM employees WHERE employee_id=130;
  DBMS_OUTPUT.PUT_LINE('New salary: ' || s);
END;
/
```

結果:

```
Old salary: 3557.4
New salary: 3657.4

PL/SQL procedure successfully completed.
```

B PL/SQLの名前解決

この付録では、PL/SQLの名前解決(識別子に対するあいまいな参照を、PL/SQLコンパイラがどのように解決するか)について説明します。

コンパイル単位内で識別子を変更(識別子の追加、名前変更または削除)すると、あいまいでない識別子参照があいまいになる可能性があります。

> ノート:
>
> ストアド PL/SQL ユニットの AUTHID プロパティは、実行時にユニットによって発行される SQL 文の名前解決に影響します。詳細は、「実行者権限および定義者権限(AUTHID プロパティ)」を参照してください。

ここでのトピック

- 修飾名およびドット表記法
- 列名の優先順位
- PL/SQLとSQLでの名前解決ルールの相違点
- 静的SQL文における名前解決
- 取得
- SELECT文およびDML文の内部取得の回避

B.1 修飾名およびドット表記法

ある名前付き項目が別の名前付き項目に属する場合は、ドット表記法を使用して「子」項目の名前を「親」項目の名前で修飾できます(場合よっては必須)。たとえば:

参照する項目	修飾に使用する必要がある項目	使用する構文
レコードのフィールド	レコードの名前	record_name.field_name
コレクションのメソッド	コレクションの名前	collection_name.method
疑似列 CURRVAL	順序の名前	sequence_name.CURRVAL
疑似列 NEXTVAL	順序の名前	sequence_name.NEXTVAL

識別子が名前付きPL/SQLユニットで宣言されている場合、次の構文を使用して単純名(宣言に含まれる名前)をユニット(ブロック、サブプログラムまたはパッケージ)の名前で修飾できます。

```
unit_name.simple_identifier_name
```

識別子を参照できない場合は、識別子の名前を修飾する必要があります(「識別子の有効範囲と可視性」を参照)。

識別子が別のスキーマに属する場合は、次の構文を使用して識別子の名前をスキーマの名前で修飾する必要があります。

```
schema_name.package_name
```

例B-1に示すとおり、単純名は複数の名前で修飾できます。

あいまいになる可能性のある修飾名の例をいくつか次に示します。

- ファンクションの戻り値のフィールドまたは属性。たとえば:

  ```
  func_name().field_name
  func_name().attribute_name
  ```

- 別のスキーマが所有するスキーマ・オブジェクト。たとえば:

  ```
  schema_name.table_name
  schema_name.procedure_name()
  schema_name.type_name.member_name()
  ```

- 別のユーザーが所有するパッケージ・オブジェクト。たとえば:

  ```
  schema_name.package_name.procedure_name()
  schema_name.package_name.record_name.field_name
  ```

- ADTを含むレコード。たとえば:

  ```
  record_name.field_name.attribute_name
  record_name.field_name.member_name()
  ```

例B-1 修飾名

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  m NUMBER;
  TYPE t1 IS RECORD (a NUMBER);
  v1 t1;
  TYPE t2 IS TABLE OF t1 INDEX BY PLS_INTEGER;
  v2 t2;
  FUNCTION f1 (p1 NUMBER) RETURN t1;
  FUNCTION f2 (q1 NUMBER) RETURN t2;
END pkg1;
/
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1 (p1 NUMBER) RETURN t1 IS
    n NUMBER;
  BEGIN
     n := m;              -- Unqualified variable name
     n := pkg1.m;         -- Variable name qualified by package name
     n := pkg1.f1.p1;     -- Parameter name qualified by function name,
                          --  which is qualified by package name
     n := v1.a;           -- Variable name followed by component name
     n := pkg1.v1.a;      -- Variable name qualified by package name
                          --  and followed by component name
     n := v2(10).a;       -- Indexed name followed by component name
     n := f1(10).a;       -- Function invocation followed by component name
     n := f2(10)(10).a;   -- Function invocation followed by indexed name
                          --  and followed by component name
```

```
    n := hr.pkg1.f2(10)(10).a;  -- Schema name, package name,
                                -- function invocation, index, component name
    v1.a := p1;
    RETURN v1;
  END f1;
  FUNCTION f2 (q1 NUMBER) RETURN t2 IS
    v_t1 t1;
    v_t2 t2;
  BEGIN
    v_t1.a := q1;
    v_t2(1) := v_t1;
    RETURN v_t2;
  END f2;
END pkg1;
/
```

B.2 列名の優先順位

ローカル変数または仮パラメータのいずれかと列の両方に属する名前をSQL文で参照すると、列名が優先されます。

> 注意:
>
> 変数名またはパラメータ名が列名として解釈されると、データが誤って削除、変更または挿入されることがあります。

例B-2では、last_nameという名前がローカル変数と列の両方に属しています(名前は大/小文字が区別されません)。そのため、WHERE句に含まれるlast_nameの参照は両方とも列に解決され、すべての行が削除されます。

例B-3では、変数に異なる名前を付けることで、例B-2の問題を解決しています。

例B-4では、ブロックにラベルを付け、変数名をブロック名で修飾することで、例B-2の問題を解決しています。

例B-5のファンクションdept_nameには、表DEPARTMENTSの列名が名前になっている仮パラメータとローカル変数があります。パラメータおよび変数の名前をファンクション名で修飾して、列名と区別しています。

例B-2 列名に解釈されるために意図していない結果を招く変数名

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

結果:

```
Deleted 107 rows.
```

例B-3 異なる変数名の使用による例B-2の修正

```
DECLARE
  v_last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = v_last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

結果:

```
Deleted 2 rows.
```

例B-4 ブロック・ラベルの使用による例B-2の修正

```
<<main>>
DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE last_name = main.last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

結果:

```
Deleted 2 rows.
```

例B-5 名前解決でのサブプログラム名

```
DECLARE
  FUNCTION dept_name (department_id IN NUMBER)
    RETURN departments.department_name%TYPE
  IS
    department_name  departments.department_name%TYPE;
  BEGIN
    SELECT department_name INTO dept_name.department_name
      --    ^column               ^local variable
    FROM departments
    WHERE department_id = dept_name.department_id;
    --     ^column          ^formal parameter
    RETURN department_name;
  END dept_name;
BEGIN
  FOR item IN (
    SELECT department_id
    FROM departments
    ORDER BY department_name) LOOP

      DBMS_OUTPUT.PUT_LINE ('Department: ' || dept_name(item.department_id));
  END LOOP;
END;
```

```
/
```

結果:

```
Department: Accounting
Department: Administration
Department: Benefits
Department: Construction
Department: Contracting
Department: Control And Credit
Department: Corporate Tax
Department: Executive
Department: Finance
Department: Government Sales
Department: Human Resources
Department: IT
Department: IT Helpdesk
Department: IT Support
Department: Manufacturing
Department: Marketing
Department: NOC
Department: Operations
Department: Payroll
Department: Public Relations
Department: Purchasing
Department: Recruiting
Department: Retail Sales
Department: Sales
Department: Shareholder Services
Department: Shipping
Department: Treasury
```

B.3 PL/SQLとSQLでの名前解決ルールの相違点

PL/SQLとSQLの名前解決ルールはよく似ています。ただし、次のことに注意してください。

- PL/SQLルールはSQLルールと比較して、より許容性が低くなっています。

  ほとんどのSQLルールが状況依存であるため、PL/SQLルールで認識されるよりも多くの状況が有効なものと認識されます。

- PL/SQLとSQLでは、修飾名の解決方法が異なります。

  たとえば、HR.JOBSという表名を解決する場合は次のとおりです。

  - PL/SQLでは、まず現行スキーマでHRという名前のパッケージ、型、表およびビューが検索され、次にパブリック・シノニムが検索され、最後にHRスキーマでJOBSという名前のオブジェクトが検索されます。
  - SQLでは、まずHRスキーマでJOBSという名前のオブジェクトが検索され、次に現行スキーマでHRという名前のパッケージ、型、表およびビューが検索されます。

PL/SQLとSQLの名前解決ルールのわずかな相違点に起因する問題を回避するには、「SELECT文およびDML文の内部取得の回避」の推奨事項に従います。

ノート:

PL/SQL コンパイラで静的 SQL 文を処理する場合は、その文が SQL サブシステムに送信され、そこで文に含まれる名前が SQL ルールを使用して SQL サブシステムによって解決されます。詳細は、「静的 SQL 文における名前解決」を参照してください。

B.4 静的SQL文における名前解決

静的SQLの詳細は、「PL/SQLの静的SQL」を参照してください。

PL/SQLコンパイラが静的SQL文を見つけると、次の処理が行われます。

1. その文がSELECT文の場合、PL/SQLコンパイラはINTO句を削除します。
2. PL/SQLコンパイラは、その文をSQLサブシステムに送信します。
3. SQLサブシステムで、その文の構文がチェックされます。

   構文が正しくない場合、PL/SQLユニットのコンパイルは失敗します。構文が正しい場合は、SQLサブシステムによって表の名前が決定され、そのSQL文の有効範囲でその他の名前の解決が試行されます。
4. そのSQL文の有効範囲で名前を解決できない場合、SQLサブシステムは、その名前をPL/SQLコンパイラに送り返します。この名前は、エスケープ識別子と呼ばれます。
5. PL/SQLコンパイラは、エスケープ識別子を解決しようとします。

   まず、コンパイラは、PL/SQLユニットの有効範囲で識別子を解決しようとします。それが失敗すると、コンパイラは、スキーマの有効範囲で識別子を解決しようとします。それが失敗すると、PL/SQLユニットのコンパイルは失敗します。
6. PL/SQLユニットのコンパイルが成功すると、PL/SQLコンパイラは、静的SQL文と等価な通常のSQL文テキストを生成し、生成されたコンピュータ・コードとともにそのテキストを格納します。
7. 実行時に、PL/SQLランタイム・システムによって、通常のSQL文を解析、バインド、実行するルーチンが起動されます。

   バインド変数はエスケープ識別子です(ステップ 4を参照)。
8. この文がSELECT文の場合、PL/SQLランタイム・システムでは、ステップ 1でPL/SQLコンパイラによって削除されたINTO句に指定されたPL/SQLターゲットに結果が格納されます。

ノート:

バインド変数はどのような順序でも評価できます。プログラムにより評価の順序が決定される場合は、プログラムで決定が行われる時点で動作は定義されません。

B.5 取得

宣言または定義が、別の有効範囲における参照の正常な解決の妨げになる場合、その宣言または定義が参照を「取得する」と呼びます。通常、取得は移行またはスキーマのアップグレードの結果として行われます。

ここでのトピック

- 外部取得
- 同一有効範囲の取得
- 内部取得

> ノート:
>
> 同一有効範囲の取得および内部取得は、SQL スコープでのみ発生します。

B.5.1 外部取得

外部取得が発生するのは、内部有効範囲内の項目に解決されていた内部有効範囲内の名前が、外部有効範囲内の項目に解決される場合です。PL/SQLとSQLは両方とも、外部取得を防止する設計になっているので、外部取得の回避に慎重になる必要はありません。

B.5.2 同一有効範囲の取得

同一有効範囲の取得が発生するのは、結合に使用される2つの表のどちらかに列が追加され、この新しい列の名前が他の表の列名と同じである場合です。その名前の列が1つの表にのみ存在する場合は、名前が未修飾で結合に現れる可能性があります。同一有効範囲の取得を回避するには、その列名が結合に現れるすべての場所で、列名を適切な表名で修飾する必要があります。

B.5.3 内部取得

内部取得が発生するのは、外部有効範囲内の項目に解決されていた内部有効範囲内の名前が、内部有効範囲内の項目に解決される場合か解決できない場合のいずれかです。最初のケースでは結果が変わる可能性があります。2番目のケースではエラーが発生します。

例B-6では、新しい列は、同じ名前の古い列の参照を取得します。col2は、新しい列col2が表tab2に追加される前はtab1.col2に解決され、追加された後はtab2.col2に解決されます。

内部取得を防止するには、「SELECT文およびDML文の内部取得の回避」のルールに従います。

例B-6 列の参照の内部取得

表tab1にはcol2という名前の列がありますが、表tab2にはありません。

```
DROP TABLE tab1;
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
INSERT INTO tab1 (col1, col2) VALUES (100, 10);
DROP TABLE tab2;
CREATE TABLE tab2 (col1 NUMBER);
INSERT INTO tab2 (col1) VALUES (100);
```

そのため、内側のSELECT文では、col2への参照が列tab1.col2に解決されます。

```
CREATE OR REPLACE PROCEDURE proc AUTHID DEFINER AS
  CURSOR c1 IS
```

```
    SELECT * FROM tab1
    WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  OPEN c1;
  CLOSE c1;
END;
/
```

表tab2への列col2の追加:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

これでプロシージャprocは無効になりました。次に起動する際に、このプロシージャはデータベースによって自動的に再コンパイルされ、内側のSELECT文におけるcol2への参照は、列tab2.col2に解決されます。

B.6 SELECT文およびDML文の内部取得の回避

次の推奨事項を遵守することによって、SELECT文、SELECT INTO文およびDML文における参照の内部取得を防止できます。

- 文内の各表に対して一意の別名を指定します。
- 文で参照される項目を所有しているスキーマと同じ名前を表の別名に指定しないようにします。
- 文内の列の各参照を適切な表の別名で修飾します。

例B-7では、スキーマhrが表tab1およびtab2を所有しています。表tab1にはtab2という名前の列があり、その抽象データ型(ADT)は属性aを持っています。表tab2にはaという名前の列はありません。推奨事項に反して、この問合せでは、表tab1に別名hrを指定して表tab2を参照しています。そのため、この問合せ内の参照hr.tab2.aは、表tab1、列tab2、属性aに解決されます。その後、この例では、列aを表tab2に追加しています。今度は、問合せ内の参照hr.tab2.aは、スキーマhr、表tab2、列aに解決されます。表tab2の列aは、表tab1の列tab2の属性aへの参照を取得します。

ここでのトピック

- 属性およびメソッドへの参照の修飾
- 行の式への参照の修飾

例B-7 属性の参照の内部取得

```
CREATE OR REPLACE TYPE type1 AS OBJECT (a NUMBER);
/
DROP TABLE tab1;
CREATE TABLE tab1 (tab2 type1);
INSERT INTO tab1 (tab2) VALUES (type1(10));
DROP TABLE tab2;
CREATE TABLE tab2 (x NUMBER);
INSERT INTO tab2 (x) VALUES (10);
/* Alias tab1 with same name as schema name,
   a bad practice used here for illustration purpose.
   Note lack of alias in second SELECT statement. */
SELECT * FROM tab1 hr
WHERE EXISTS (SELECT * FROM hr.tab2 WHERE x = hr.tab2.a);
```

結果:

```
TAB2(A)
```

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

TYPE1(10)

1 row selected.
```

表tab2(スキーマhrに属する)への列aの追加:

```
ALTER TABLE tab2 ADD (a NUMBER);
```

ここでこの問合せを実行すると、hr.tab2.aは、スキーマhr、表tab2、列aに解決されます。この内部取得を回避するために、問合せに推奨事項を適用します。

```
SELECT * FROM hr.tab1 p1
WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

B.6.1 属性およびメソッドへの参照の修飾

表の要素の属性やメソッドを参照するには、表に別名を付け、その別名を使用して、属性やメソッドに対する参照を修飾します。

例B-8の表tbl1には、属性xを持つADTであるデータ型t1の列col1があります。この例は、tbl1.col1.xへの適切な参照と適切でない参照をいくつか示しています。

例B-8 ADT属性の参照の修飾

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x NUMBER);
/
DROP TABLE tb1;
CREATE TABLE tb1 (col1 t1);
```

次のINSERT文の参照では、列リストがないため、別名は不要です。

```
BEGIN
  INSERT INTO tb1 VALUES ( t1(10) );
  INSERT INTO tb1 VALUES ( t1(20) );
  INSERT INTO tb1 VALUES ( t1(30) );
END;
/
```

属性xへの次の参照では、エラーORA-00904が発生します。

```
UPDATE tb1 SET col1.x = 10 WHERE col1.x = 20;
UPDATE tb1 SET tb1.col1.x = 10 WHERE tb1.col1.x = 20;
UPDATE hr.tb1 SET hr.tb1.col1.x = 10 WHERE hr.tb1.col1.x = 20;
DELETE FROM tb1 WHERE tb1.col1.x = 10;
```

属性xへの表の別名を使用した次の参照は、適切です。

```
UPDATE hr.tb1 t SET t.col1.x = 10 WHERE t.col1.x = 20;
DECLARE
  y NUMBER;
BEGIN
  SELECT t.col1.x INTO y FROM tb1 t WHERE t.col1.x = 30;
END;
/
```

```
DELETE FROM tb1 t WHERE t.col1.x = 10;
```

B.6.2 行の式への参照の修飾

行の式は、表の別名への参照として解決する必要があります。行の式は、UPDATE文のSET句で使用したり、SQLファンクションREFまたはVALUEのパラメータにすることができます。

例B-9の表ot1は、属性xを持つADTであるデータ型t1の要素のスタンドアロンのネストした表です。この例は、行の式への適切な参照と適切でない参照をいくつか示しています。

例B-9 行の式への参照の修飾

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x number);
/
DROP TABLE ot1;
CREATE TABLE ot1 OF t1;
BEGIN
  INSERT INTO ot1 VALUES (t1(10));
  INSERT INTO ot1 VALUES (20);
  INSERT INTO ot1 VALUES (30);
END;
/
```

次の参照では、エラーORA-00904が発生します。

```
UPDATE ot1 SET VALUE(ot1.x) = t1(20) WHERE VALUE(ot1.x) = t1(10);
DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));
```

表の別名を使用した次の参照は、適切です。

```
UPDATE ot1 o SET o = (t1(20)) WHERE o.x = 10;
DECLARE
  n_ref  REF t1;
BEGIN
  SELECT REF(o) INTO n_ref FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/
DECLARE
  n t1;
BEGIN
  SELECT VALUE(o) INTO n FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/
DECLARE
  n NUMBER;
BEGIN
  SELECT o.x INTO n FROM ot1 o WHERE o.x = 30;
END;
/
DELETE FROM ot1 o WHERE VALUE(o) = (t1(20));
```

C PL/SQLのプログラム上の制限

この付録では、PL/SQL言語によって課されるプログラム上の制限について説明します。PL/SQLはプログラミング言語Adaをベースにしています。したがって、PL/SQLでは、ツリー構造の中間言語Descriptive Intermediate Attributed Notation for Ada(DIANA)を使用しています。この中間言語は、インタフェース定義言語(IDL)と呼ばれるメタ表記を使用して定義されます。DIANAは、コンパイラなどのツールによって内部的に使用されます。

PL/SQLのソース・テキストはコンパイル時にシステム・コードに変換されます。サブプログラムまたはパッケージのDIANAおよびシステム・コードは、データベースに格納されます。実行時、それらは共有メモリー・プール内にロードされます。DIANAは依存サブプログラムのコンパイルに使用され、システム・コードはそのまま実行されます。

共有メモリー・プールの場合、パッケージ仕様部、ADT仕様部、スタンドアロン・サブプログラムまたは無名ブロックは、67108864(2**26)個のDIANAノードに制限されています(ノードは、識別子、キーワード、演算子などのトークンに対応します)。PL/SQLコンパイラによる制限を超えないかぎり、6,000,000行以下のコードが許されます(PL/SQLコンパイラによる一部の制限を表C-1に示します)。

表C-1 PL/SQLコンパイラの制限

項目	制限
プログラム・ユニットに渡されるバインド変数	32768
プログラム・ユニット内の例外ハンドラ	65536
レコード内のフィールド	65536
ブロック・ネストのレベル	255
レコード・ネストのレベル	32
副問合せネストのレベル	254
ラベル・ネストのレベル	98
ネストしたコレクションのレベル	事前定義された制限なし
PLS_INTEGER 値または BINARY_INTEGER 値の絶対値	-2147483648..2147483647
明示カーソル、ファンクション、プロシージャでの仮パラメータの数	65536
プログラム・ユニットから参照されるオブジェクト	65536

項目	制限
FLOAT 値(2 進数)の精度	126
NUMBER 値(10 進数)の精度	38
REAL 値(2 進数)の精度	63
識別子のサイズ(バイト)	128
文字列リテラルのサイズ(バイト)	32767
CHAR 値(バイト)のサイズ	32767
LONG 値(バイト)のサイズ	32760
LONG RAW 値(バイト)のサイズ	32760
RAW 値(バイト)のサイズ	32767
VARCHAR2 値(バイト)のサイズ	32767
NCHAR 値(バイト)のサイズ	32767
NVARCHAR2 値(バイト)のサイズ	32767
BFILE 値(バイト)のサイズ	4G×DB_BLOCK_SIZE パラメータの値
BLOB 値(バイト)のサイズ	4G×DB_BLOCK_SIZE パラメータの値
CLOB 値(バイト)のサイズ	4G×DB_BLOCK_SIZE パラメータの値
NCLOB 値(バイト)のサイズ	4G×DB_BLOCK_SIZE パラメータの値
トリガーのサイズ	32K

プログラム・ユニットに必要なメモリー量を見積もるときには、データ・ディクショナリ・ビューUSER_OBJECT_SIZEを使用して静的データを問い合せることができます。PARSED_SIZEという列に、フラット化されたDIANAのバイト・サイズが戻ります。たとえば:

```
CREATE OR REPLACE PACKAGE pkg1 AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
```

```
END pkg1;
/
CREATE PACKAGE BODY pkg1 AS
  -- FUNCTION f1 returns a collection of elements (1,2,3,... x)
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
  BEGIN
    FOR i IN 1..x LOOP
      PIPE ROW(i);
    END LOOP;
    RETURN;
  END f1;
END pkg1;
/
```

次の問合せの結果を書式設定するためのSQL*Plusコマンド:

```
COLUMN name FORMAT A4
COLUMN type FORMAT A12
COLUMN source_size FORMAT 999
COLUMN parsed_size FORMAT 999
COLUMN code_size FORMAT 999
COLUMN error_size FORMAT 999
```

問合せ:

```
SELECT * FROM user_object_size WHERE name = 'PKG1' ORDER BY type;
```

結果:

```
NAME TYPE         SOURCE_SIZE PARSED_SIZE CODE_SIZE ERROR_SIZE
---- ------------ ----------- ----------- --------- ----------
PKG1 PACKAGE              112         498       310         79
PKG1 PACKAGE BODY         233         106       334          0
```

ただし、サイズの解析結果からDIANAノードの数を見積もることはできません。解析後のサイズが同じプログラム・ユニットが2つあり、一方に必要なDIANAノードの数が1500に対して、他方に必要なのが2000である(2番目のユニットに含まれるSQL文が複雑であるなどの理由で)場合もあるためです。

PL/SQLブロック、サブプログラム、パッケージまたはスキーマ・レベルのユーザー定義型のいずれかのサイズが上限を超えると、「PLS-00123: プログラムが大きすぎます。」などのエラーとなります。多くの場合、この問題はパッケージまたは無名ブロックで発生します。パッケージの場合、最良の方法は複数個のより小さなパッケージに分けることです。無名ブロックの場合、最良の方法はサブプログラムのグループとして再定義し、データベースに格納できるようにすることです。

データ型の制限の詳細は、「PL/SQLのデータ型」を参照してください。

D PL/SQLの予約語およびキーワード

予約語(表D-1を参照)とキーワード(表D-2を参照)は、PL/SQLで特別な意味を持つ識別子です。これらは、大/小文字を区別します。詳細は、「予約語およびキーワード」を参照してください。

ノート:

この付録の単語には、SQL の予約語もあります。これらは、動的パフォーマンス・ビューV$RESERVED_WORDS を使用して表示できます。このビューの詳細は、『Oracle Database リファレンス』を参照してください。

表D-1 PL/SQLの予約語

頭文字:	予約語
A	ALL、ALTER、AND、ANY、AS、ASC、AT
B	BEGIN、BETWEEN、BY
C	CASE、CHECK、CLUSTERS、CLUSTER、COLAUTH、COLUMNS、COMPRESS、CONNECT、CRASH、CREATE、CURSOR
D	DECLARE、DEFAULT、DESC、DISTINCT、DROP
E	ELSE、END、EXCEPTION、EXCLUSIVE
F	FETCH、FOR、FROM、FUNCTION
G	GOTO、GRANT、GROUP
H	HAVING
I	IDENTIFIED、IF、IN、INDEX、INDEXES、INSERT、INTERSECT、INTO、IS
L	LIKE、LOCK
M	MINUS、MODE
N	NOCOMPRESS、NOT、NOWAIT、NULL
O	OF、ON、OPTION、OR、ORDER、OVERLAPS

頭文字:	予約語
P	PROCEDURE、PUBLIC
R	RESOURCE、REVOKE
S	SELECT、SHARE、SIZE、SQL、START、SUBTYPE
T	TABAUTH、TABLE、THEN、TO、TYPE
U	UNION、UNIQUE、UPDATE
V	VALUES、VIEW、VIEWS
W	WHEN、WHERE、WITH

表D-2 PL/SQLのキーワード

頭文字:	キーワード
A	A、ADD、ACCESSIBLE、AGENT、AGGREGATE、ARRAY、ATTRIBUTE、AUTHID、AVG
B	BFILE_BASE、BINARY、BLOB_BASE、BLOCK、BODY、BOTH、BOUND、BULK、BYTE
C	C、CALL、CALLING、CASCADE、CHAR、CHAR_BASE、CHARACTER、CHARSET、CHARSETFORM、CHARSETID、CLOB_BASE、CLONE、CLOSE、COLLECT、COMMENT、COMMIT、COMMITTED、COMPILED、CONSTANT、CONSTRUCTOR、CONTEXT、CONTINUE、CONVERT、COUNT、CREDENTIAL、CURRENT、CUSTOMDATUM
D	DANGLING、DATA、DATE、DATE_BASE、DAY、DEFINE、DELETE、DETERMINISTIC、DIRECTORY、DOUBLE、DURATION
E	ELEMENT、ELSIF、EMPTY、ESCAPE、EXCEPT、EXCEPTIONS、EXECUTE、EXISTS、EXIT、EXTERNAL
F	FINAL、FIRST、FIXED、FLOAT、FORALL、FORCE
G	GENERAL
H	HASH、HEAP、HIDDEN、HOUR

頭文字:	キーワード
I	IMMEDIATE、INCLUDING、INDICATOR、INDICES、INFINITE、INSTANTIABLE、INT、INTERFACE、INTERVAL、INVALIDATE、ISOLATION
J	JAVA
L	LANGUAGE、LARGE、LEADING、LENGTH、LEVEL、LIBRARY、LIKE2、LIKE4、LIKEC、LIMIT、LIMITED、LOCAL、LONG、LOOP
M	MAP、MAX、MAXLEN、MEMBER、MERGE、MIN、MINUTE、MOD、MODIFY、MONTH、MULTISET
N	NAME、NAN、NATIONAL、NATIVE、NCHAR、NEW、NOCOPY、NUMBER_BASE
O	OBJECT、OCICOLL、OCIDATE、OCIDATETIME、OCIDURATION、OCIINTERVAL、OCILOBLOCATOR、OCINUMBER、OCIRAW、OCIREF、OCIREFCURSOR、OCIROWID、OCISTRING、OCITYPE、OLD、ONLY、OPAQUE、OPEN、OPERATOR、ORACLE、ORADATA、ORGANIZATION、ORLANY、ORLVARY、OTHERS、OUT、OVERRIDING
P	PACKAGE、PARALLEL_ENABLE、PARAMETER、PARAMETERS、PARENT、PARTITION、PASCAL、PERSISTABLE、PIPE、PIPELINED、PLUGGABLE、POLYMORPHIC、PRAGMA、PRECISION、PRIOR、PRIVATE
R	RAISE、RANGE、RAW、READ、RECORD、REF、REFERENCE、RELIES_ON、REM、REMAINDER、RENAME、RESULT、RESULT_CACHE、RETURN、RETURNING、REVERSE、ROLLBACK、ROW
S	SAMPLE、SAVE、SAVEPOINT、SB1、SB2、SB4、SECOND、SEGMENT、SELF、SEPARATE、SEQUENCE、SERIALIZABLE、SET、SHORT、SIZE_T、SOME、SPARSE、SQLCODE、SQLDATA、SQLNAME、SQLSTATE、STANDARD、STATIC、STDDEV、STORED、STRING、STRUCT、STYLE、SUBMULTISET、SUBPARTITION、SUBSTITUTABLE、SUM、SYNONYM
T	TDO、THE、TIME、TIMESTAMP、TIMEZONE_ABBR、TIMEZONE_HOUR、TIMEZONE_MINUTE、TIMEZONE_REGION、TRAILING、TRANSACTION、TRANSACTIONAL、TRUSTED
U	UB1、UB2、UB4、UNDER、UNPLUG、UNSIGNED、UNTRUSTED、USE、USING
V	VALIST、VALUE、VARIABLE、VARIANCE、VARRAY、VARYING、VOID

頭文字:	キーワード
W	WHILE、WORK、WRAPPED、WRITE
Y	YEAR
Z	ZONE

E PL/SQLの事前定義のデータ型

この付録は、パッケージSTANDARDに事前定義されたデータ型ファミリ(データ型とサブタイプ)でグループ分けされています。

定数

この定数は、名前の最大長を定義します。

```
ORA_MAX_NAME_LEN CONSTANT PLS_INTEGER := 128;
```

BFILEデータ型ファミリ

```
type BFILE is BFILE_BASE;
```

BLOBデータ型ファミリ

```
type BLOB is BLOB_BASE;
subtype "BINARY LARGE OBJECT" is BLOB;
```

BOOLEANデータ型ファミリ

```
type BOOLEAN is (FALSE, TRUE);
```

CHARデータ型ファミリ

```
type VARCHAR2 is new CHAR_BASE;
type MLSLABEL is new CHAR_BASE;
type UROWID   is new CHAR_BASE;
DBMS_ID and DBMS_QUOTED_ID define the length of identifiers in objects for SQL, PL/SQL and users.
subtype DBMS_ID              is VARCHAR2(ORA_MAX_NAME_LEN);
subtype DBMS_QUOTED_ID       is VARCHAR2(ORA_MAX_NAME_LEN+2);
DBMS_ID_30 and DBMS_QUOTED_ID_30 define the length of SQL objects whose limits is 30 bytes.
subtype DBMS_ID_30           is VARCHAR2(30);
subtype DBMS_QUOTED_ID_30    is VARCHAR2(32);
subtype VARCHAR              is VARCHAR2;
subtype STRING               is VARCHAR2;
subtype LONG                 is VARCHAR2(32760);
subtype RAW                  is VARCHAR2;
subtype "LONG RAW"           is RAW(32760);
subtype ROWID                is VARCHAR2(256);
subtype CHAR                 is VARCHAR2;
subtype CHARACTER            is CHAR;
subtype "CHARACTER VARYING"  is VARCHAR;
subtype "CHAR VARYING"       is VARCHAR;
subtype "NATIONAL CHARACTER" is CHAR CHARACTER SET NCHAR_CS;
subtype "NATIONAL CHAR"      is CHAR CHARACTER SET NCHAR_CS;
subtype "NCHAR"              is CHAR CHARACTER SET NCHAR_CS;
subtype "NVARCHAR2"          is VARCHAR2 CHARACTER SET NCHAR_CS;
```

CLOBデータ型ファミリ

```
type CLOB is CLOB_BASE;
subtype "CHARACTER LARGE OBJECT"          is CLOB;
subtype "CHAR LARGE OBJECT"               is CLOB;
subtype "NATIONAL CHARACTER LARGE OBJECT" is CLOB CHARACTER SET NCHAR_CS;
```

```
subtype "NCHAR LARGE OBJECT"              is CLOB CHARACTER SET NCHAR_CS;
subtype "NCLOB"                           is CLOB CHARACTER SET NCHAR_CS;
```

DATEデータ型ファミリ

```
type DATE                              is     DATE_BASE;

type TIMESTAMP                         is new DATE_BASE;

type "TIMESTAMP WITH TIME ZONE"        is new DATE_BASE;
type "INTERVAL YEAR TO MONTH"          is new DATE_BASE;
type "INTERVAL DAY TO SECOND"          is new DATE_BASE;
type "TIMESTAMP WITH LOCAL TIME ZONE" is new DATE_BASE;
subtype TIME_UNCONSTRAINED          is TIME(9);
subtype TIME_TZ_UNCONSTRAINED       is TIME(9) WITH TIME ZONE;
subtype TIMESTAMP_UNCONSTRAINED     is TIMESTAMP(9);
subtype TIMESTAMP_TZ_UNCONSTRAINED  is TIMESTAMP(9) WITH TIME ZONE;
subtype YMINTERVAL_UNCONSTRAINED    is INTERVAL YEAR(9) TO MONTH;
subtype DSINTERVAL_UNCONSTRAINED    is INTERVAL DAY(9) TO SECOND (9);
subtype TIMESTAMP_LTZ_UNCONSTRAINED is TIMESTAMP(9) WITH LOCAL TIME ZONE;
```

NUMBERデータ型ファミリ

```
type NUMBER is NUMBER_BASE;
subtype FLOAT              is NUMBER; -- NUMBER(126)
subtype REAL               is FLOAT;   -- FLOAT(63)
subtype "DOUBLE PRECISION" is FLOAT;
subtype INTEGER  is NUMBER(38,0);
subtype INT      is INTEGER;
subtype SMALLINT is NUMBER(38,0);
subtype DECIMAL is NUMBER(38,0);
subtype NUMERIC is DECIMAL;
subtype DEC     is DECIMAL;
subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
subtype NATURAL        is BINARY_INTEGER range 0..2147483647;
subtype NATURALN       is NATURAL not null;
subtype POSITIVE       is BINARY_INTEGER range 1..2147483647;
subtype POSITIVEN      is POSITIVE not null;
subtype SIGNTYPE       is BINARY_INTEGER range '-1'..1;  -- for SIGN functions
subtype PLS_INTEGER    is BINARY_INTEGER;
type BINARY_FLOAT  is NUMBER;
type BINARY_DOUBLE is NUMBER;
subtype SIMPLE_INTEGER is BINARY_INTEGER NOT NULL;
subtype SIMPLE_FLOAT   is BINARY_FLOAT   NOT NULL;
subtype SIMPLE_DOUBLE  is BINARY_DOUBLE  NOT NULL;
```

関連項目:

- PL/SQLのデータ型の詳細は、「PL/SQLのデータ型」を参照してください
- 事前定義のサブタイプにも適用される情報は、「ユーザー定義のPL/SQLサブタイプ」を参照してください

索引

記号

- _ワイルドカード文字 2.7.5.3
- $$PLSQL_LINE問合せディレクティブ 2.9.1.4.1
- $$PLSQL_UNIT_OWNER問合せディレクティブ 2.9.1.4.1
- $$PLSQL_UNIT_TYPE問合せディレクティブ 2.9.1.4.1
- $$PLSQL_UNIT問合せディレクティブ 2.9.1.4.1
- %BULK_EXCEPTIONSカーソル属性 12.4.1.4
- %BULK_ROWCOUNTカーソル属性 12.4.1.5
- %FOUNDカーソル属性
  - 暗黙カーソル 6.2.1.2
  - 名前付きカーソル 6.2.2.7.2
- %ISOPENカーソル属性
  - 暗黙カーソル 6.2.1.1
  - 名前付きカーソル 6.2.2.7.1
- %NOTFOUNDカーソル属性
  - 暗黙カーソル 6.2.1.3
  - 名前付きカーソル 6.2.2.7.3
- %ROWCOUNTカーソル属性
  - 暗黙カーソル 6.2.1.4
  - 名前付きカーソル 6.2.2.7.4
- %ROWTYPE属性 5.12.4
  - 列の別名 6.2.2.5
  - 明示カーソル 6.2.2.5
  - 非表示列 5.12.4.4
  - 構文図 13.57
  - 仮想列 5.12.4.3
- %TYPE属性 2.3.5
  - 初期値 2.3.5
  - NOT NULL制約 2.3.5
  - 構文図 13.65
- %ワイルドカード文字 2.7.5.3

A

- 抽象データ型(ADT) 1.2.8.5, 14.6
  - 作成 14.13

- アクセント記号の有無を区別しない比較 2.7.5.2.3
- ACCESS_INTO_NULL例外 11.4
- ACCESSIBLE BY句 13.1
    - CREATE FUNCTION文 14.7
    - CREATE PACKAGE文 14.9
    - CREATE PROCEDURE文 14.11
    - CREATE TYPE BODY文 14.14
    - CREATE TYPE文 14.13
    - パッケージ仕様部 10.1
    - サブプログラム 8.2
- accessor 13.1
- accessorリスト 13.1
- ADT
    - 「抽象データ型(ADT)」を参照:
- 集計 5.6
- AGGREGATE句 13.2
    - CREATE FUNCTION文 14.7
- 集計ファンクション 14.7
    - PL/SQL式 2.7.8
    - パイプライン・テーブル・ファンクション 12.5.6
    - SQL%NOTFOUND属性 6.2.1.3
- 別名
    - 列
        - カーソルFOR LOOP 6.3.2
        - 明示カーソル 6.2.2.5
    - 表
        - 内部取得の回避 B.6
        - 行の式 B.6.2
        - 表の要素の属性またはメソッド B.6.1
- エイリアシング(問題)
    - SELECT BULK COLLECT INTO文 12.4.2.1.1
    - サブプログラム・パラメータ 8.7.4
- ALTER FUNCTION文 14.1
- ALTER LIBRARY文 14.2
- ALTER PACKAGE文 14.3
- ALTER PROCEDURE文 14.4
- ALTER TRIGGER文 14.5
- ALTER TYPE文 14.6
- AND演算子 2.7.3
- 無名ブロック 1.2.2
    - AUTHIDプロパティ 8.14
- ANYDATAデータ型 12.5.2
- ANYDATASETデータ型 12.5.2

- ANYTYPEデータ型 12.5.2
- アプリケーション共通オブジェクト
    - メタデータリンク 13.61
- PL/SQLのアーキテクチャ 1.3
- 配列
    - 連想
        - 「連想配列」を参照 5.1
    - 非PL/SQL 5.1
- 値の代入
    - コンポジット変数
        - コレクション 5.7
        - レコード 5.13
    - スカラー変数 2.6
- 代入文 2.6.1
    - 構文図 13.3
- 連想配列 5.2
    - 「コレクション」も参照
    - 特性 5.1
    - 比較 5.9
    - 定数の宣言 5.2.1
    - FIRSTおよびLASTメソッド 5.10.5.1
    - FORALL文 13.34
    - NLSパラメータ 5.2.2
- 最小(字句)単位 2.2
- 基本構造的にNULLのコレクション
    - 「NULLのコレクション」を参照
- 属性
    - %ROWTYPE
        - 「%ROWTYPE属性」を参照 5.12.4
    - %TYPE
        - 「%TYPE属性」を参照 2.3.5
    - カーソル
        - 「カーソル属性」を参照 6.2
- AUTHIDプロパティ 8.14
    - 構文図 13.41
- AUTONOMOUS_TRANSACTIONプラグマ 6.7.4
    - パイプライン・テーブル・ファンクション 12.5.2
    - 構文図 13.4
- 自律型ルーチン 6.7
    - 宣言 6.7.4
- 自律型トランザクション 6.7
    - 制御 6.7.5
    - パイプライン・テーブル・ファンクション 13.48

- 自律型トリガー 6.7.6

B

- バッグ・データ構造 5.1
- ベース型 3
- 基本LOOP文 4.2.1
  - 構文図 13.5
- BETWEEN演算子 2.7.5.4
- BINARY_DOUBLEデータ型
  - 事前定義の定数 3.1.2
  - サブタイプ 3.1.3
  - コードのチューニング 12.3.5.1
- BINARY_FLOATデータ型
  - 事前定義の定数 3.1.2
  - サブタイプ 3.1.3
  - コードのチューニング 12.3.5.1
- BINARY_INTEGERデータ型
  - 「PLS_INTEGERデータ型」を参照:
- バインド変数
  - SQLインジェクションの回避 7.4.2.1
  - プレースホルダ
    - 「バインド変数のプレースホルダ」を参照 7.2.1
- 空白埋め
  - 代入 3.1.4.3.2
  - 比較 3.1.4.3.3
- ブロック 1.2.2
  - 構文図 13.6
- BOOLEANデータ型 3.2
- BOOLEAN式 2.7.6
- 静的なBOOLEAN式 2.7.9.2
- BOOLEAN変数 2.6.4
- 組込みファンクション
  - 「SQLファンクション」を参照
- バルク・バインド 12.4
- BULK COLLECT句 12.4.2, 12.4.2.3
  - エイリアシング 12.4.2.1.1
  - FETCH文 12.4.2.2
  - RETURNING INTO句
    - FORALL文 12.4.3
  - SELECT INTO文 12.4.2.1
  - 問合せ結果セットの処理 6.3.1.2
  - 行を戻さない 12.4.2

- バルクSQL 12.4
    - 複合DMLトリガー 9.3.3.3

C

- キャッシュ, ファンクション結果 8.12
- 計算された列
    - 「仮想列」を参照
- コール仕様 8.15
    - CREATE FUNCTION文 14.7
    - CREATE PROCEDURE文 14.11
    - CREATE TYPE文 14.13
    - パッケージ内 10.1
    - 構文図 13.7
- コール・スタック, AUTHIDプロパティ 8.14
- 取得 B.5
- カスケード・トリガー 9.11
- CASE_NOT_FOUND例外 11.4
- CASE式
    - 検索 2.7.7.2
    - 単純 2.7.7.1
- 大/小文字を区別しない比較 2.7.5.2.3
- 大/小文字の区別
    - 文字の比較 2.7.5.2.3
    - 文字リテラル 2.2.3
    - 識別子 2.2.2
        - ユーザー定義の引用識別子 2.2.2.3.2
    - キーワード D
    - LIKE演算子 2.7.5.3
    - 予約語 D
- CASE文 4.1
    - 検索 4.1.5
        - 構文図 13.8
    - 単純 4.1.4
        - IF THEN ELSIF文 4.1.3
        - 構文図 13.8
- C宣言 13.7
- 文字コード 2.1
- 文字リテラル 2.2.3
    - 「文字列」も参照
- 文字セット 2.1
- CHARデータ型 3.1.4
- CHARデータ型ファミリ E

- CLOBデータ型と比較演算子 2.7.5
- CLOBデータ型ファミリ E
- CLOSE文 13.9
- 照合順番 2.7.5.2.3
- 照合 13.20
- コレクション 5
    - パブリック・パッケージ項目 5.11
    - 他への代入 5.7
    - 他との比較 5.9
    - カーソル変数 13.18
    - 宣言の構文 13.11
    - 空 5.1
        - コンストラクタによる作成 5.5
    - 索引
        - 「索引コレクション」を参照 12.4.1.1
    - 内部サイズ
        - DELETEメソッド 5.10.1
        - EXTENDメソッド 5.10.3
        - TRIMメソッド 5.10.2
    - 多次元 5.8
    - null 5.1
        - コレクション変数への代入 5.7.2
    - パイプライン・テーブル・ファンクション 12.5.1
    - 問合せ
        - 動的SQLを使用 7.2.2
        - 静的SQLを使用 6.4.6
    - 問合せ結果の取出し 12.4.2
    - 型 5.1
- COLLECTION_IS_NULL例外 11.4
- コレクションのコンストラクタ 5.5
- コレクション・メソッド 5.10
    - サブプログラム・パラメータ 5.10
    - 起動の構文 13.10
    - NULLのコレクション 5.10
- 列の別名
    - カーソルFOR LOOP 6.3.2
    - 明示カーソル 6.2.2.5
- COLUMNS擬似演算子 12.6.3.2
- コメント2.2.5
    - ネスト 2.2.5.2
    - PL/SQLラッパー・ユーティリティ A.3
    - 構文図 13.12
- COMMIT文 6.6.1

    - FOR UPDATEカーソル 6.6.6.3
    - 自律型トランザクション 6.7.5.2
- 比較演算子 2.7.5
    - カーソル変数 13.18
- 互換性のあるデータ型
    - コレクション変数 5.7.1
    - スカラー変数 2.6
- コンパイル
    - 条件付き 2.9
    - システム固有の実行用 12.10
    - 解釈 12.10.5
- コンパイル・パラメータ 1.3.2
    - 値の表示 2.9.1.4.1
    - 事前定義問合せディレクティブ 2.9.1.4.1
- compile_clause
    - ALTER FUNCTION文 14.1
- compile_clause構文図 13.13
- COMPILE句 13.13
- コンパイラ・ディレクティブ
    - 「プラグマ」を参照
- コンパイル時の警告 11.1
- コンポジット・データ型 5
- コンポジット変数 5
- 複合トリガー 9.3.3
- 計算集中型コード 12.3.5
- 連結演算子(||) 2.7.1
- 同時トランザクション 6.7.5.4
- 条件, SQL MULTISET 5.9.3
- 条件付きコンパイル 2.9
- 条件付きコンパイル・ディレクティブ 2.9.1
    - エラー 2.9.1.3
    - 問合せ 2.9.1.4
    - 制限事項 2.9.4
    - 選択 2.9.1.2
- 条件述語 9.3.1
- 条件付き選択文 4.1
- 条件付きトリガー 9.1
- 定数
    - 宣言 2.3.3
        - 連想配列 5.2.1
        - レコード 5.12.2
        - 構文図 13.14
    - 初期値 2.3.4

    - 事前定義 3.1.2
    - 静的 2.7.9.4
        - DBMS_DB_VERSIONパッケージ内 2.9.1.5
- 制約付きサブタイプ 3.4.2
    - パフォーマンスを重視するコード 12.3.5.2
    - サブプログラム・パラメータ 8.7.1.1
- 制約
    - カーソル・パラメータ 13.30
    - NOT NULL
        - 「NOT NULL制約」を参照 2.3.1
    - 他とのトリガーの比較 9.2
- コンストラクタ
    - 「コレクションのコンストラクタ」を参照
- トランザクションのコンテキスト 6.7.2
- CONTINUE文 4.2.4
    - 構文図 13.15
- CONTINUE WHEN文 4.2.5
    - 構文図 13.15
- 制御文 4
- 制御トークン 2.9.1.1
- 相関副問合せ 6.3.4
- 相関名 9.4
    - 「疑似レコード」も参照
    - LONGまたはLONG RAW列 9.10.2
- COUNTコレクション・メソッド 5.10.6
- COVERAGEプラグマ 13.16
- Cプロシージャ, 起動 8.15
- CREATE_WRAPPEDプロシージャ A.4
- CREATE FUNCTION文 14.7
- CREATE LIBRARY文 14.8
- CREATE PACKAGE文 14.9
- CREATE TRIGGER文 14.12
- CREATE TYPE BODY文 14.14
- CREATE TYPE文 14.13
- crosseditionトリガー 9.1
- CURRENT OF句 6.6.6.2
    - FOR UPDATEカーソル 6.6.6.2
    - ROWIDプロシージャの代用 6.6.6.3
- CURRVAL疑似列 6.1.2.1
- カーソル 6.2
    - 明示的
        - 「明示カーソル」を参照 6.2.2
    - FOR UPDATE 6.6.6.2

        - COMMITまたはROLLBACKの後 6.6.6.3
    - 暗黙的
        - 「暗黙カーソル」を参照 6.2.1
    - SERIALLY_REUSABLEパッケージ内 10.7.2
    - 名前付き 6.2.2
        - パイプライン・テーブル・ファンクション 12.5.5
        - 「明示カーソル」および「カーソル変数」も参照 6.2.2
    - ネスト 6.5
- CURSOR_ALREADY_OPEN例外 11.4
- カーソル属性
    - カーソル変数 6.4.7
    - 明示カーソル 6.2.2.7
        - %FOUND 6.2.2.7.2
        - %ISOPEN 6.2.2.7.1
        - %NOTFOUND 6.2.2.7.3
        - %ROWCOUNT 6.2.2.7.4
    - 暗黙カーソル 6.2.1
        - DBMS_SQLパッケージ 7.3
        - ネイティブ動的SQL 7.2
        - SQL%BULK_EXCEPTIONS 12.4.1.4
        - SQL%BULK_ROWCOUNT 12.4.1.5
        - SQL%FOUND 6.2.1.2
        - SQL%ISOPEN 6.2.1.1
        - SQL%NOTFOUND 6.2.1.3
        - SQL%ROWCOUNT 6.2.1.4
    - 使用可能な場所 6.2
- CURSOR式 6.5
    - パイプライン・テーブル・ファンクションへの引渡し 12.5.6
- カーソルFOR LOOP文
    - 問合せ結果セットの処理 6.3.2
    - 再帰的起動 8.10
    - 構文図 13.17
- カーソル番号
    - へのカーソル変数の変換 7.3.4
    - カーソル変数への変換 7.3.3
    - DBMS_SQL.GET_NEXT_RESULTプロシージャ 7.3.2
    - DBMS_SQL.RETURN_RESULTプロシージャ 7.3.1
- カーソル・パラメータ 6.2.2.6
- カーソル仕様部 13.30
- カーソル変数 6.4
    - へのカーソル番号の変換 7.3.3
    - カーソル番号への変換 7.3.4
    - DBMS_SQL.GET_NEXT_RESULTプロシージャ 7.3.2

- DBMS_SQL.RETURN_RESULTプロシージャ 7.3.1
- 宣言の構文図 13.18

D

- データの抽象化 1.2.8
- データベース文字セット 2.1.1
- データベース・リンク
  - DRユニット 8.14.3
- DATABASEトリガー 9.5.2
- データバインドされた照合 13.20
- データ定義言語文
  - 「DDL文」を参照
- データ・ポンプ・インポートとトリガー 9.14
- データ型
  - CREATE TYPE文 14.13
- データ型 13.19
- データ型 3
  - 「サブタイプ」も参照
  - コレクション
    - 「コレクション」を参照 5
  - 互換性
    - コレクション変数 5.7.1
    - スカラー変数 2.6
  - コンポジット 5
  - オブジェクト
    - 「抽象データ型(ADT)」を参照 1.2.8.5
  - 式 2.7
  - 事前定義 E
  - RECORD
    - 「レコード」を参照 5
  - スカラー 3
  - SQL 3.1
  - ユーザー定義
    - 「抽象データ型(ADT)」を参照 1.2.8.5
  - 決定する内容 3
- データ型変換 3.1
  - 暗黙的
    - 「暗黙的なデータ型変換」を参照 12.3.5.3
  - SQLインジェクション 7.4.1.3
- データ型ファミリ 3
  - オーバーロードされたサブプログラム 8.9
  - グループ分けされた事前定義のデータ型 E

    - 同じベース型を使用したサブタイプ 3.4.3
- DATEデータ型ファミリ E
- DBMS_ASSERTパッケージ 7.4.2.2
- DBMS_DB_VERSIONパッケージ 2.9.1.5
- DBMS_DDLパッケージ A.4
- DBMS_PARALLEL_EXECUTEパッケージ 12.7
- DBMS_PREPROCESSORパッケージ 2.9.3
- DBMS_SQL.GET_NEXT_RESULTプロシージャ 7.3.2
- DBMS_SQL.RETURN_RESULTプロシージャ 7.3.1
- DBMS_SQL.TO_NUMBERファンクション 7.3.4
- DBMS_SQL.TO_REFCURSORファンクション 7.3.3
- DBMS_SQLパッケージ 7.3
    - システム固有の動的SQLへの切替え 7.3
- DBMS_STANDARDパッケージ 2.4
- DBMS_WARNINGパッケージ 11.1.1
- dbmsupgin.sqlスクリプト 12.10.5
- dbmsupgnv.sqlスクリプト 12.10.5
- DDL文
    - 動的SQL 7
    - トリガー 6.7.6
    - サブプログラムの副作用 8.13
- デッドロック
    - 自律型トランザクション 6.7.5.4
    - 暗黙的なロールバック 6.6.4
- 宣言 2.3
    - 例外の呼出し 11.8.1
- DEFAULT COLLATION句 13.20
    - CREATE FUNCTION文 14.7
    - CREATE PACKAGE文 14.9
    - CREATE PROCEDURE文 14.11
    - CREATE TRIGGER文 14.12
    - CREATE TYPE文 14.13
- デフォルト値 8.7.5
    - 「初期値」も参照
    - カーソル・パラメータ 6.2.2.6.1
    - サブプログラム・パラメータ 8.7.5
- DEFINE
    - バインド・カテゴリ 12.4
- 定義者権限句 13.41
- 定義者権限ユニット
    - 「DRユニット」を参照
- DELETEコレクション・メソッド 5.10.1
    - COUNTメソッド 5.10.6

    - EXISTSメソッド 5.10.4
    - EXTENDメソッド 5.10.3
    - FIRSTメソッド 5.10.5
    - LASTメソッド 5.10.5
    - NEXTメソッド 5.10.8
    - PRIORメソッド 5.10.8
    - TRIMメソッド 5.10.2
- DELETE文 13.21
    - 「DML文」も参照
    - BEFORE文トリガー 9.9
    - PL/SQLの拡張機能 13.21
- DELETING条件述語 9.3.1
- デリミタ 2.2.1
- 密コレクション 5.1
- DEPRECATEプラグマ 13.22
- Descriptive Intermediate Attributed Notation for Ada (DIANA) C
- DETERMINISTIC
    - CREATE FUNCTION文 14.7
- DETERMINISTIC句 13.23
- DETERMINISTICオプション 13.36
    - パイプライン・テーブル・ファンクション 12.5.2
- DIAGNOSTIC_DEST初期化パラメータ 11
- ディレクティブ 2.9.1
    - 「条件付きコンパイル・ディレクティブ」も参照
    - コンパイラ
        - 「プラグマ」を参照 2.2.4
    - エラー 2.9.1.3
    - 問合せ 2.9.1.4
    - 選択 2.9.1.2
- ダイレクト・キー・パーティション化 14.7, 14.14
- DML文
    - 内部取得の回避 B.6
    - FORALL文 12.4.1
    - パイプライン・テーブル・ファンクション内 12.5.2
    - パイプライン・テーブル・ファンクションの結果 12.5.7
    - PL/SQLの構文 6.1.1
    - 効率的な繰返し 12.4.1
- DMLトリガー 9.3
- ドット表記法
    - コレクション・メソッド 5.10
    - 名前付きPL/SQLユニット内の識別子 2.4
    - 擬似列 6.1.2.1
    - レコード・フィールド 5

    - 名前解決 B.1
- 二重引用符(") 2.1.1
- DROP FUNCTION文 14.15
- DROP LIBRARY文 14.16
- DROP PACKAGE文 14.17
- DROP PROCEDURE文 14.18
- DROP TRIGGER文 14.19
- DROP TYPE BODY文 14.21
- DRユニット 8.14
    - 「AUTHIDプロパティ」も参照
    - コール・スタック 8.14
    - データベース・リンク 8.14.3
    - 動的SQL 8.14
    - INHERIT REMOTE PRIVILEGES権限 8.14.3
    - 名前解決 8.14
    - 権限チェック 8.14
    - SCHEMAトリガー 9.5.1
    - 静的SQL 8.14
- DUP_VAL_ON_INDEX例外 11.4
- 動的SQL 7
    - AUTHIDプロパティ 8.14
    - システム固有 7.2
        - DBMS_SQLパッケージへの切替え 7.3
    - バインド変数のプレースホルダ
        - EXECUTE IMMEDIATE文 7.2.1
        - 繰返し 7.2.3
    - チューニング 12.2

E

- エディション化 14.6
- エディションされた抽象データ型(ADT) 14.6
- コレクションの要素 5
- 要素の仕様
    - CREATE TYPE文 14.13
    - 構文図 13.24
- 埋込みSQL
    - 「静的SQL」を参照
- 空のコレクション 5.1
    - コンストラクタによる作成 5.5
- エラー・ディレクティブ 2.9.1.3
- エラー処理 11
- エラー・レポート・ファンクション

    - SQLCODE 13.62
    - SQLERRM 13.63
        - SQL%BULK_EXCEPTIONS 12.4.1.4
- エスケープ文字 2.7.5.3
- エスケープ識別子 B.4
- 評価順序 2.7.2
- イベントのパブリケーション 9.15
- 型の進化 14.6
- 例外
    - 処理 11.2
        - FORALL文 12.4.1.2
        - トリガー 9.8
        - 「例外ハンドラ」も参照 11.2
    - 内部的定義
        - 「内部的に定義された例外」を参照 11.3
    - 事前定義
        - 「事前定義の例外」を参照 11.4
    - カーソルFOR LOOP文での呼出し 6.3.2
    - 宣言での呼出し 11.8.1
    - 例外ハンドラでの呼出し 11.8.2
    - 明示的な呼出し 11.7
    - 再呼出し 11.7.1.3
    - 未処理 11.9
        - FORALL文 12.4.1.2
    - ユーザー定義
        - 「ユーザー定義の例外」を参照 11.5
- EXCEPTION_INITプラグマ 13.25
    - ユーザー定義の例外へのエラー・コードの関連付け 11.7.2
    - 内部的に定義された例外への名前の指定 11.3
- 例外ハンドラ 11.2
    - 実行の続行 11.11
    - 例外の呼出し 11.8.2
    - NO_DATA_NEEDED 12.5.8
    - GOTO文 13.37
    - ロケータ変数 11.2.2
    - エラー・コードとメッセージの取得 11.10
    - トランザクションの再試行 11.12
    - 構文図 13.27
- EXECUTE IMMEDIATE文 7.2.1
    - 構文図 13.28
- EXISTSコレクション・メソッド 5.10.4
- ループの終了 4.2
- EXIT文 4.2.2

    - 構文図 13.29
- EXIT WHEN文 4.2.3
    - 基本LOOP文 4.2.3
    - 構文図 13.29
- 明示カーソル 6.2.2
    - 宣言の構文図 13.30
    - パッケージ内
        - 宣言 10.8
        - オープンとクローズ 10.7.3
    - 問合せ結果の処理
        - FOR LOOP文 6.3.2
        - OPEN文、FETCH文およびCLOSE文 6.3.3
- 明示的な書式モデル 7.4.2.3
- 式 2.7
    - CURSOR 6.5
        - パイプライン・テーブル・ファンクションへの引渡し 12.5.6
    - データ型 2.7
    - 明示カーソル 6.2.2.5
    - PL/SQLでのSQLファンクション 2.7.8
    - 静的 2.7.9
    - 構文図 13.31
- EXTENDコレクション・メソッド 5.10.3
- 外部サブプログラム 8.15
    - コール仕様 13.7

F

- FETCH FIRST句 12.4.2.1.2
- FETCH文
    - COMMITの間 6.6.6.3
    - レコード変数 5.13.2.2
    - 構文図 13.32
    - 行を戻さない 6.2.2.3
    - BULK COLLECT句 12.4.2.2
    - カーソル変数 6.4.3
    - 明示カーソル 6.2.2.3
- レコードのフィールド 5
- FIRSTコレクション・メソッド 5.10.5
- FORALL文 12.4.1
    - 連想配列 13.34
    - バルク・バインド 12.4
    - BULK COLLECT句 12.4.3
    - 疎コレクション 12.4.1.1

        - SQL%BULK_EXCEPTIONS 12.4.1.4.1
    - 呼び出された例外の処理
        - FORALL完了後 12.4.1.4
        - 即時 12.4.1.3
    - 影響を受ける行の数 12.4.1.5
    - 構文図 13.34
    - 未処理の例外 12.4.1.2
- FOR LOOP文 4.2.6
    - 「カーソルFOR LOOP文」も参照
    - 境界 4.2.6.2
    - FORALL文 12.4.1
    - STEP句 4.2.6
    - 構文図 13.33
- 書式モデル 7.4.2.3
- FOR UPDATEカーソル 6.6.6.2
    - COMMITまたはROLLBACKの後 6.6.6.3
- サブプログラムの前方宣言 8.6
- ファンクション 8
    - 「サブプログラム」も参照
    - 集計
        - 「集計ファンクション」を参照 12.5.6
    - 組込み
        - 「SQLファンクション」を参照 12.3.6
    - 宣言の構文図 13.36
    - エラー・レポート
        - SQLCODE 13.62
        - SQLERRM 12.4.1.4
    - 起動 8.3
        - SQL文 8.13
    - オプション 8.5.1
    - SQL
        - 「SQLファンクション」を参照 12.3.6
    - 構造 8.5.1
    - 表
        - 「テーブル・ファンクション」を参照 12.5.1
- ファンクション結果キャッシュ 8.12
- ファンクション仕様部 13.36

G

- 生成された列
    - 「仮想列」を参照:
- GET_NEXT_RESULTプロシージャ 7.3.2

- グローバル識別子 2.5
- GOTO文 4.3.1
    - 制限 13.37
    - 構文図 13.37
- PL/SQLユニットへのロールの付与 8.14.1

H

- ハードウェア算術 12.3.5.1
- ハッシュ表 5.1
- PL/SQLソース・テキストの隠ぺい
    - 「PL/SQLソース・テキストのラップ」を参照:
- ホスト変数
    - バルク・バインド 12.4.4
    - カーソル変数 6.4.9
    - パッケージ 10.2

I

- 識別子 2.2.2, 2.4
    - 「名前」も参照
    - あいまいな参照 B
    - エスケープ B.4
    - グローバル 2.5
    - 静的SQL 6.1.1
    - ローカル 2.5
    - 参照 2.4
    - 有効範囲 2.5
    - ユーザー定義 2.2.2.3
        - データの収集 12.8
    - 可視性 2.5
- IDL C
- IF文 4.1
    - IF THEN ELSE形式 4.1.2
    - IF THEN ELSIF形式 4.1.3
        - ネストしたIF THEN ELSE文 4.1.3
        - 単純なCASE文 4.1.3
    - IF THEN形式 4.1.1
    - ネスト 4.1.2
    - 構文図 13.38
- impとトリガー 9.14
- 暗黙カーソル 6.2.1

    - CURSOR式 6.5
    - 宣言の構文 13.39
    - 動的SQL 7.3
    - 問合せ結果の処理
        - カーソルFOR LOOP文 6.3.2
        - SELECT INTO文 6.3.1
- 暗黙的なデータ型変換
    - 最小化 12.3.5.3
    - サブプログラム・パラメータ 8.7.2
        - オーバーロード・エラーの発生 8.9.3
    - サブタイプ
        - 制約付き 3.4.2
        - 無制約 3.4.1
        - 同じファミリ内のベース型 3.4.3
- 問合せ結果の暗黙的な戻し 7.3.1
- 暗黙的なROLLBACK文 6.6.4
- インポートとトリガー 9.14
- インバインド 12.4
- 独立したトランザクション
    - 「自律型トランザクション」を参照
- 索引付き表
    - 「連想配列」を参照
- 索引コレクション 12.4.1.1
    - サブセットを表す 12.4.1.1
- コレクションの索引 5
- 無限ループ 4.2.1
- コンパイル時のINFORMATIONAL警告 11.1
- INHERIT ANY PRIVILEGES権限 8.14
- INHERIT PRIVILEGES権限 8.14
- INHERIT REMOTE PRIVILEGES権限 8.14.3
- 初期化パラメータ 1.3.2
- 初期値 8.7.5
    - 「デフォルト値」も参照
    - %TYPE属性 2.3.5
    - NOT NULL制約 2.3.1
    - 定数 2.3.4
    - 変数
        - ネストした表 5.4
        - レコード 5.12.1
        - スカラー 2.3.4
        - VARRAY 5.3
- INLINEプラグマ 12.1.1
    - 構文図 13.40

- 内部取得 B.5.3
    - 回避 B.6
- IN演算子 2.7.5.5
- IN OUTパラメータ・モード 8.7.3
- INパラメータ・モード 8.7.3
- 入力 1.2.7
- 問合せディレクティブ 2.9.1.4
- INSERTING条件述語 9.3.1
- INSERT文 13.42
    - 「DML文」も参照:
    - レコードの挿入 5.15
        - 制限 5.17
    - PL/SQLの拡張機能 13.42
- INSTEAD OFトリガー 9.1
    - CREATE文 9.5.3
    - DML文 9.3.2
        - 複合 9.3.3.1
        - パイプライン・テーブル・ファンクションの結果 12.5.7
        - ネストした表の列 9.3.2
- インタフェース定義言語(IDL) C
- 内部的に定義された例外 11.3
    - 名の指定 11.3
    - 明示的な呼出し 11.7.1.2
- インタプリタ型コンパイル 12.10.5
- INVALID_CURSOR例外 11.4
- INVALID_NUMBER例外 11.4
- 非表示列 5.12.4.4
- 実行者権限句 13.41
- 実行者権限ユニット
    - 「IRユニット」を参照
- IRユニット 8.14
    - 「AUTHIDプロパティ」も参照:
    - コール・スタック 8.14
    - 動的SQL 8.14
    - ロールの付与 8.14.1
    - 名前解決 8.14
    - 権限チェック 8.14
    - 静的SQL 8.14, 8.14.2
    - テンプレート・オブジェクト 8.14.2
- IS [NOT] NULL演算子 2.7.5.1
    - コレクション 5.9.1
- トランザクションの分離レベル 6.7.3

J

- Javaクラス・メソッドの起動 8.15
- Javaメソッドの宣言 13.7

K

- キーと値のペア
  - 「連想配列」を参照:
- キーワード 2.2.2.1
  - リスト D

L

- ラベル付きLOOP文 4.2.3
- LASTコレクション・メソッド 5.10.5
- LEVEL擬似列 6.1.2
- 字句単位 2.2
- ライブラリ
  - 作成 14.8
  - 削除 14.16
  - 明示的な再コンパイル 14.2
- ライブラリ算術 12.3.5.1
- LIKE演算子 2.7.5.3
- LIMIT句 12.4.2.2.1
- LIMITコレクション・メソッド 5.10.7
- 行継続文字 2.2.3
- リテラル 2.2.3
- ローカル識別子 2.5
- ロケータ変数 11.2.2
- ロック
  - デフォルトの上書き 6.6.6
  - 結果セットの行 6.6.6.2
  - 表 6.6.6.1
- ロック・モード 6.6.6.1
- LOCK TABLE文 6.6.6.1
- 論理演算子 2.7.3
- 論理値 3.2
- LOGIN_DENIED例外 11.4
- LONGデータ型 3.1.5
  - トリガー 9.10.2
- LONG RAWデータ型 3.1.5

    - トリガー 9.10.2
- LOOP文
    - 終了 4.2
    - 種類 4.2
    - ラベル付き 4.2, 4.2.3
    - ネスト 4.2.3
    - 最適化 12.3.4
- LOOP UNTIL構造 4.2.7

M

- MALFORMED_WRAP_INPUT例外 A.4
- 管理性 1.1.6
- MapReduceワークロード 14.7, 14.14
- マテリアライズド・ビューとトリガー 14.12
- メンバーシップ・テスト 2.7.5.5
- メソッド, コレクション
    - 「コレクション・メソッド」を参照
- メソッド4 7.3
- パラメータ混合表記法 8.7.6
- モード
    - ロック 6.6.6.1
    - サブプログラム・パラメータ 8.7.3
- マルチバイトの文字セット
    - データベース文字セット 2.1.1
    - 値の変数 3.1.4.2
- 多次元コレクション 5.8
- 複数行コメント 2.2.5.2
- 複数のデータ変換 12.5
- 多重集合条件 5.9.3
- 表の変更 9.10.3
- 変更表エラー
    - ファンクション 8.13
    - トリガー 9.10.3

N

- 名前 2.4
    - 「識別子」も参照
    - 修飾
        - 「ドット表記法」を参照 2.4
    - 修飾リモート 2.4

    - リモート 2.4
    - 単純 2.4
- 名前付きカーソル 6.2.2
    - 「明示カーソル」および「カーソル変数」も参照
    - パイプライン・テーブル・ファンクション 12.5.5
- 名前付きパラメータ表記法 8.7.6
- 名前解決 B
    - AUTHIDプロパティ 8.14
    - 静的SQL B.4
    - PL/SQLとSQLの相違点 B.3
- 各国語文字セット 2.1.2
- システム固有の動的SQL 7.2
    - DBMS_SQLパッケージへの切替え 7.3
- システム固有の実行，コンパイル 12.10
- NATURALNサブタイプ 3.3.2
- NATURALサブタイプ 3.3.2
- ネストしたコメント 2.2.5.2
- ネストしたカーソル 6.5
- ネストしたIF文 4.1.2
    - IF THEN ELSIF形式 4.1.3
- ネストしたLOOP文 4.2.3
- ネストしたレコード
    - 代入の例 5.13.1
    - 宣言の例 5.12.3
- ネストしたサブプログラム 8.2
    - 宣言および定義 8.2
    - 前方宣言 8.6
- ネストした表 5.4
    - 「コレクション」も参照:
    - NULL値の代入 5.7.2
    - SET演算子結果の代入 5.7.3
    - 特性 5.1
    - ビュー内の列，トリガー 9.3.2
    - NULLとの比較 5.9.1
    - 2つの比較 5.9.2
    - 相関名 9.4
    - COUNTメソッド 5.10.6.2
    - FIRSTおよびLASTメソッド 5.10.5.3
    - ファンクションによって戻される 12.5.1
    - SQL MULTISET条件 5.9.3
- ネストしたトランザクション 6.7
- NEW相関名 9.4
    - LONGまたはLONG RAW列 9.10.2

- NEXTコレクション・メソッド 5.10.8
- NEXTVAL疑似列 6.1.2.1
- NLSパラメータ
    - 連想配列 5.2.2
    - 文字の比較 2.7.5.2.3
    - SQLインジェクション 7.4.1.3
- NO_DATA_FOUND例外 11.4
- NO_DATA_NEEDED例外 12.5.8
    - SQLCODE 11.4
- NOCOPYヒント 13.35
    - サブプログラム・パラメータのエイリアシング 8.7.4.1
    - サブプログラムの起動のチューニング 12.3.3
- 非空白埋め比較セマンティクス 3.1.4.3.3
- no-op(何もしない)文 4.3.2
- NOT_LOGGED_ON例外 11.4
- NOT NULL制約 2.3.1
    - %TYPE属性 2.3.5
    - EXTENDメソッド 5.10.3
- NOT演算子 2.7.3
- NULLのコレクション 5.1
    - コレクション変数への代入 5.7.2
    - コレクション・メソッド 5.10
- NULL文
    - 構文図 13.44
    - 使用 4.3.2
- NULL文字列 2.2.3
- NULL値
    - レコード変数への代入 5.13.3
    - コレクションとの比較
        - 連想配列 5.9
        - ネストした表 5.9.1
        - VARRAY 5.9.1
    - 比較演算子 2.7.5
    - 連結演算子 2.7.1
    - $$PLSQL_UNIT_OWNER問合せディレクティブ 2.9.1.4.1
    - $$PLSQL_UNIT問合せディレクティブ 2.9.1.4.1
    - コレクション変数 5.7.2
    - サブプログラム・パラメータ 8.7.5
    - 解決できない問合せディレクティブ 2.9.1.4.3
    - 制御文 2.7.3
    - IN演算子 2.7.5.5
    - セット 2.7.5.5
    - USING句 7.2.1

    - 単純なCASE式 2.7.7.1
    - 単純なCASE文 4.1.4
- NUMBERデータ型ファミリ
    - 非効率 12.3.5.1
    - メンバー E

O

- PL/SQLソース・テキストの不明瞭化
    - 「PL/SQLソース・テキストのラップ」を参照:
- OBJECT_VALUE疑似列 9.4.1
- オブジェクト型
    - 「抽象データ型(ADT)」を参照:
- OCI
    - 連想配列 5.2.3
    - カーソル変数 6.4.9
- RETURNING INTO句 12.4.2.3
- OLD相関名 9.4
- OPEN FOR文 13.46
    - 再帰的起動 8.10
- OPEN文 13.45
    - 再帰的起動 8.10
- 操作 2.7.2
- 演算子
    - 比較 2.7.5
        - カーソル変数 13.18
    - 論理 2.7.3
    - 関係 2.7.5.2
        - コレクション 5.9
- 演算子の優先順位 2.7.2
- オプティマイザ
    - PL/SQL 12.1
    - SQL 12.5.5
- Oracle Call Interface(OCI)
    - 連想配列 5.2.3
    - カーソル変数 6.4.9
- Oracle RAC環境, 結果キャッシュ 8.12.5.6
- ORA-nエラー
    - 「内部的に定義された例外」を参照
- 通常のユーザー定義の識別子 2.2.2.3.1
- 元のインポートとトリガー 9.14
- OR演算子 2.7.3
- アウトバインド 12.4

- 外部取得 B.5.1
- OUTパラメータ・モード 8.7.3
- 出力 1.2.7
- オーバーロードされたサブプログラム 8.9
    - INLINEプラグマ 12.1.1

P

- パッケージ 10.1
    - 本体
        - 「パッケージ本体」を参照 10.4
    - DBMS_STANDARD 2.4
    - 明示的な再コンパイル 14.3
    - 機能 10.2
    - ロールの付与 8.14.1
    - 作成のガイドライン 10.8
    - 初期化 10.5
    - 静的定数 2.7.9.4
    - プライベート項目 10.4
    - 製品固有 10
    - パブリック項目
        - 「パブリック・パッケージ項目」を参照 10.3
    - 使用する理由 10.2
    - SERIALLY_REUSABLE 10.7
    - 仕様部
        - 「パッケージ仕様部」を参照 10.3
    - STANDARD
        - 「STANDARDパッケージ」を参照 10.10
    - 状態 10.6
    - Oracleが提供する 10
    - ラップ A
        - ガイドライン A.2
- パッケージ本体 10.1
    - 作成 14.10
    - 削除 14.17
    - 初期化部 10.4
        - 初期値の代入 10.8
    - 置換 14.10
- パッケージ仕様部 10.1, 10.3
    - 「パブリック・パッケージ項目」も参照
    - 作成 14.9
    - カーソル変数 13.18
    - 削除 14.17

    - 置換 14.9
- パッケージ・サブプログラム 8.2
- PARALLEL_ENABLE句 13.47
    - CREATE FUNCTION文 14.7
- PARALLEL_ENABLEオプション 13.36
    - パイプライン・テーブル・ファンクション 12.5.2
    - テーブル・ファンクション 12.5.1
- パラレルDML
    - バルク・バインド 12.4
    - 大規模な表 12.7
- パラメータ
    - コンパイル
        - 「コンパイル・パラメータ」を参照 1.3.2
    - 明示カーソル 6.2.2.6
    - 初期化 1.3.2
    - サブプログラム
        - 「サブプログラム・パラメータ」を参照 8.7
- パラメータ・モード 8.7.3
- PARENT相関名 9.4
    - LONGまたはLONG RAW列 9.10.2
- カッコ
    - ネスト 2.7.2
    - 評価順序の制御 2.7.2
    - わかりやすさの向上 2.7.2
- パターン一致 2.7.5.3
- パーセント記号(%)ワイルドカード文字 2.7.5.3
- コンパイル時のPERFORMANCE警告 11.1
- PIPELINEDファンクション
    - 構文図 13.49
- PIPELINEDオプション 12.5.1
    - 指定する場所 12.5.2
- パイプライン・テーブル・ファンクション 12.5, 12.5.1
    - 「テーブル・ファンクション」も参照
    - 集計ファンクション 12.5.6
    - 自律型トランザクション 13.48
- PIPE ROW文 12.5.2
    - 自律型ルーチン 6.7.5.4
- PL/Scopeツール 12.8
- PL/SQLアーキテクチャ 1.3
- PL/SQLブロック
    - 「ブロック」を参照
- PL/SQLエンジン 1.3.1
- PL/SQLファンクションの結果のキャッシュ 8.12

- PL/SQL言語
  - 利点 1.1
  - 高いパフォーマンス 1.1.2
  - 高い生産性 1.1.3
  - 字句単位 2.2
  - 制限 C
  - 主な特長 1.2
  - 管理性 1.1.6
  - 移植性 1.1.4
  - プログラム上の制限 C
  - スケーラビリティ 1.1.5
  - SQLとの統合 1.1.1
  - 構文およびセマンティクス 13
- PL/SQLオプティマイザ 12.1
- PL/SQL表
  - 「連想配列」を参照:
- PL/SQLユニット 1.3.2
  - ストアド
    - 「ストアドPL/SQLユニット」を参照 1.3.2
- PL/SQLラッパー・ユーティリティ A.3
- バインド変数のプレースホルダ
  - 条件付きコンパイル・ディレクティブ 2.9.4
  - 動的SQL
    - EXECUTE IMMEDIATE文 7.2.1
    - 繰返し 7.2.3
  - 静的SQL 6.1.1
    - OPEN FOR文 6.4.2
  - トリガー本体 9.4
- PLS_INTEGERデータ型 3.3
  - コードのチューニング 12.3.5.1
- 静的なPLS_INTEGER式 2.7.9.1
- PLSQL_CCFLAGSコンパイル・パラメータ 2.9.1.4.2
- PLSQL_OPTIMIZE_LEVELコンパイル・パラメータ 12.1
- PLSQL_WARNINGSコンパイル・パラメータ
  - 値の表示
    - ALL_PLSQL_OBJECT_SETTINGSビュー 11.1
    - DBMS_WARNINGサブプログラム 11.1.1
  - 値の設定
    - ALTER文 11.1
    - PLSQL_WARNINGSサブプログラム 11.1.1
- 多相テーブル・ファンクション 12.6
  - コンパイルおよび実行 12.6.4
  - 定義 12.6.1

    - 実装 12.6.2
    - 起動 12.6.3
        - COLUMNS擬似演算子 12.6.3.2
    - 最適化 12.6.5
- 移植性 1.1.4
- パラメータ位置表記法 8.7.6
- POSITIVENサブタイプ 3.3.2
- POSITIVEサブタイプ 3.3.2
- 処理後のソース・テキスト 2.9.3
- プラグマ 2.2.4
    - AUTONOMOUS_TRANSACTION 13.4
        - パイプライン・テーブル・ファンクション 12.5.2
    - COVERAGE 13.16
    - DEPRECATE 13.22
    - EXCEPTION_INIT 13.25
    - INLINE 12.1.1
        - 構文図 13.40
    - RESTRICT_REFERENCES 13.53
    - SERIALLY_REUSABLE 13.60
    - UDF 13.66
- 優先順位, 演算子 2.7.2
- 事前定義の定数 3.1.2
- 事前定義のデータ型 E
- 事前定義の例外 11.4
    - 明示的な呼出し 11.7.1.2
    - 再宣言 11.6
- 事前定義問合せディレクティブ 2.9.1.4.1
- 事前定義のサブタイプ E
- プリプロセッサ制御トークン 2.9.1.1
- PRIORコレクション・メソッド 5.10.8
- 権限チェックとAUTHIDプロパティ 8.14
- プロシージャ 8
    - 「サブプログラム」も参照:
    - 宣言の構文 13.50
    - 起動 8.3
    - 構造 8.5.1
- プロシージャ仕様部 13.50
- 製品固有のパッケージ 10
- プログラムのプロファイルおよびトレース 12.9
- PROGRAM_ERROR例外 11.4
- プログラム上の制限 C
- 疑似列 6.1.2
    - OBJECT_VALUE 9.4.1

- 擬似命令
  - 「プラグマ」を参照:
- 擬似レコード 9.4
  - 「相関名」も参照:
- パブリック・パッケージ項目
  - 適切 10.3.1
  - コレクション型 5.11
  - カーソル変数 13.18
  - 宣言 10.3
  - RECORD型 5.12.3
  - 参照 10.3
  - リモート変数 10.3.1
  - 有効範囲 10.3
  - 可視性 10.3
- イベントのパブリッシュ 9.15
- サブプログラムの純正規則 8.13

Q

- 修飾式 5.6
- 修飾名
  - 「ドット表記法」を参照
- 修飾リモート名 2.4
- 問合せ 6.1.1
  - 「SELECT INTO文」も参照
  - 結果の暗黙的な戻し 7.3.1
  - ファンクションの起動 12.3.2
  - 結果セットの処理 6.3
    - 複数行の動的問合せ 7.2.2
- 引用符，一重または二重 2.1.1
- ユーザー定義の引用識別子 2.2.2.3.2

R

- RAISE_APPLICATION_ERRORプロシージャ 11.7.2
- RAISE文 11.7.1
  - 構文図 13.51
- 例外の明示的な呼出し 11.7
- 範囲テスト 2.7.5.4
- 読取り専用トランザクション 6.6.5
- 読取り/書込みトランザクション 6.6.5
- ストアドPL/SQLユニットの再コンパイル 14

- レコード 5
  - パブリック・パッケージ項目 5.12.3
  - 値の代入 5.13
  - 他との比較 5.14
  - 作成 5.12
    - 構文図 13.52
  - 定数の宣言 5.12.2
  - ネスト
    - 「ネストしたレコード」を参照 5.12.3
  - 行を表す 5.12.4
  - 型 5.12.3
- 再帰的サブプログラム 8.10
  - 結果キャッシュ 8.12.4.2
- 再帰トリガー 9.9
- REF CURSOR
  - 「カーソル変数」を参照
- REF CURSOR型 6.4.1
- 関係演算子 2.7.5.2
  - コレクション 5.9
- RELEASE定数 2.9.1.5
- リモート例外処理
  - サブプログラム 11.8
  - トリガー 9.8
- リモート名 2.4
- リモート・パブリック・パッケージ変数 10.3.1
- リモート・サブプログラム
  - 例外 11.8
  - トリガーによる起動 9.6
  - コンポジット・パラメータ 5
- REPEAT UNTIL構造 4.2.7
- ストアドPL/SQLユニットの置換 14
- 例外の再呼出し 11.7.1.3
- 予約済のプリプロセッサ制御トークン 2.9.1.1
- 予約語
  - 情報 2.2.2.1
  - リスト D
- RESTRICT_REFERENCESプラグマ 13.53
- RESULT_CACHE句 8.12.1, 13.56
- ファンクションのRESULT_CACHEオプション 13.36, 14.7
- 結果キャッシュ 8.12
- RETURN_RESULTプロシージャ 7.3.1
- ファンクションのRETURN句 8.5.1
- RETURNING INTO句 13.55

    - BULK COLLECT句 12.4.2.3
        - FORALL文 12.4.3
- 問合せ結果の暗黙的な戻し 7.3.1
- RETURN INTO句
    - 「RETURNING INTO句」を参照
- RETURN文 8.5.2
- REUSE SETTINGS句 1.3.2
- ROLLBACK文 6.6.2
    - FOR UPDATEカーソル 6.6.6.3
    - 暗黙的 6.6.4
    - 自律型トランザクション 6.7.5.2
    - 透過的 9.9
- ROWID 3.1.6
- ROWIDデータ型 3.1.6
- ROWID疑似列 6.1.2
    - CURRENT OF句の代替 6.6.6.3
- 行レベルのトリガー 9.3
- ROWNUM擬似列
    - バルクSELECT操作 12.4.2.1.2
    - 単一行の結果セット 6.3.1.1
- ROWTYPE_MISMATCH例外
    - エラー・コード 11.4
    - 例 11.4
- ランタイム・エラー
    - 「例外」を参照

S

- 同一有効範囲の取得 B.5.2
- SAMPLE句 12.4.2.1.2
- SAVEPOINT文 6.6.3
    - 自律型トランザクション 6.7.5.3
- スケーラビリティ
    - SERIALLY_REUSABLEパッケージ 10.7
    - サブプログラム 1.1.5
- スカラー・データ型 3
- スカラー変数
    - 値の代入 2.6
    - 宣言 2.3.2
        - 構文図 13.58
    - 初期値 2.3.4
- スキーマ・オブジェクト
    - 「ストアドPL/SQLユニット」を参照:

- SCHEMAトリガー 9.5.1
- 識別子の有効範囲 2.5
- 検索CASE式 2.7.7.2
- 検索CASE文 4.1.5
    - 構文図 13.8
- セキュリティ・メカニズム
    - SQLインジェクションの回避 7.4
    - PL/SQLのソース・テキストのラップ
        - 利点 A
        - 制限 A.1
    - トリガー 9.2
- SELECT FOR UPDATE文 6.6.6.2
- SELECT INTO文 6.1.1
    - 「問合せ」も参照
    - 値の代入
        - レコード変数 5.13.2.1
        - スカラー変数 2.6.2
    - 内部取得の回避 B.6
    - 問合せ結果セットの処理 6.3.1
    - SQL%NOTFOUND属性 6.2.1.3
    - SQL%ROWCOUNT属性 6.2.1.4
    - 構文図 13.59
    - BULK COLLECT句 12.4.2.1
- 選択ディレクティブ 2.9.1.2
- 選択子
    - 単純なCASE式 2.7.7.1
    - 単純なCASE文 4.1.4
- SELF_IS_NULL例外 11.4
- 順序 6.1.2.1
- 順次制御文 4.3
- SERIALLY_REUSABLEパッケージ 10.7
- SERIALLY_REUSABLEプラグマ 13.60
- セッション・カーソル 6.2
- セット・データ構造 5.1
- セット・メンバーシップのテスト 2.7.5.5
- SET TRANSACTION文 6.6.5
- コンパイル時のSEVERE警告 11.1
- SHARING句
    - CREATE FUNCTION文 14.7
    - CREATE LIBRARY文 14.8
    - CREATE PACKAGE文 14.9
    - CREATE PROCEDURE文 14.11
    - CREATE TRIGGER文 14.12

    - CREATE TYPE文 14.13
    - sharing_clause構文図 13.61
- 短絡評価
    - 仕組み 2.7.4
    - コードのチューニング 12.3.7
- サブプログラムの副作用 8.11
- SIGNTYPEサブタイプ 3.3.2
- SIMPLE_DOUBLEサブタイプ 3.1.3
    - コードのチューニング 12.3.5.1
- SIMPLE_FLOATサブタイプ 3.1.3
    - コードのチューニング 12.3.5.1
- SIMPLE_INTEGERサブタイプ 3.3.3
    - コードのチューニング 12.3.5.1
- 単純なCASE式 2.7.7.1
- 単純なCASE文 4.1.4
    - IF THEN ELSIF文 4.1.3
    - 構文図 13.8
- 単純なDMLトリガー 9.3
- 単純名 2.4
- 単一行コメント 2.2.5.1
- 一重引用符(') 2.1.1
- ソート順序付け 13.20
- 疎コレクション 5.1
    - FORALL文 12.4.1.1
    - SQL%BULK_EXCEPTIONS 12.4.1.4.1
    - 移動 5.10.8
- 仕様
    - カーソル 13.30
    - ファンクション 13.36
    - パッケージ
        - 「パッケージ仕様部」を参照 10.1
    - プロシージャ 13.50
- SQL
    - バルク 12.4
        - 複合DMLトリガー 9.3.3.3
    - 動的
        - 「動的SQL」を参照 7
    - 静的
        - 「静的SQL」を参照 6
- SQL_MACRO 13.64
- SQL_MACRO句
    - CREATE FUNCTION文 14.7
- SQL*Loaderとトリガー 9.14

- SQL%BULK_EXCEPTIONSカーソル属性 12.4.1.4
- SQL%BULK_ROWCOUNTカーソル属性 12.4.1.5
- SQL%FOUNDカーソル属性 6.2.1.2
- SQL%NOTFOUNDカーソル属性 6.2.1.3
- SQL%ROWCOUNTカーソル属性 6.2.1.4
- SQLCODEファンクション 13.62
- SQLカーソル
  - 「暗黙カーソル」を参照
- SQLデータ型 3.1
- SQLERRMファンクション 13.63
  - SQL%BULK_EXCEPTIONS 12.4.1.4
- SQLファンクション
  - PL/SQL式 2.7.8
  - チューニング 12.3.6
- SQLインジェクション 7.4
- PL/SQLのSQLとの統合 1.1.1
- SQL MULTISET条件 5.9.3
- SQL MULTISET演算子 5.7.3
- SQLオプティマイザ 12.5.5
- SQL文 1.2.2
  - 「無名ブロック」も参照
  - ストアドPL/SQLユニット 14
  - トリガー 9.1
  - コレクション・メソッドの起動 5.10
  - PL/SQLファンクションの起動 8.13
  - チューニング 12.3.1
- スタンドアロン・サブプログラム 8.2
  - ファンクション
    - 作成 14.7
    - 削除 14.15
    - 明示的な再コンパイル 14.1
    - 置換 14.7
  - プロシージャ
    - 作成 14.11
    - 削除 14.18
    - 明示的な再コンパイル 14.4
    - 置換 14.11
- STANDARDパッケージ
  - 定義済みデータ型
    - 「事前定義のデータ型」を参照 E
  - 定義済例外
    - 「事前定義の例外」を参照 11.4
  - PL/SQL環境の定義方法 10.10

    - 定義済識別子の表示 2.2.2.2
    - 定義済項目の参照 2.4
- 文インジェクション 7.4.1.2
- 文レベルのトリガー 9.3
- 文の変更 7.4.1.1
- 静的定数 2.7.9.4
    - DBMS_DB_VERSIONパッケージ内 2.9.1.5
- 静的な式 2.7.9
- 静的SQL 6
    - AUTHIDプロパティ 8.14
    - 名前解決 B.4
    - PL/SQLの識別子 6.1.1
    - バインド変数のプレースホルダ 6.1.1
        - OPEN FOR文 6.4.2
- STORAGE_ERROR例外 11.4
    - 再帰的起動 8.10
- ストアドPL/SQLユニット 1.3.2
    - 作成 14
    - 再コンパイル 14
    - 置換 14
    - ラップ A
- ストアド・サブプログラム 8.2
    - 未処理の例外 11.9
    - ラップ A
- 記憶域表 5.4.2
- 文字列 2.2.3
    - 「文字リテラル」も参照
    - null 2.2.3
    - 長さが0 (ゼロ) 2.2.3
- STRINGサブタイプ 3.1.4.3.1
- 強いREF CURSOR型
    - 作成 6.4.1
    - FETCH文 6.4.3
- サブプログラム 8
    - インライン化 12.1.1
    - トリガーによる起動 9.6
    - リモート
        - 「リモート・サブプログラム」を参照 5
    - 未処理の例外 11.9
- サブプログラムの起動
    - 最適化 12.1.1
    - 解決 8.8
    - 構文 8.3

    - チューニング 12.3.3
- サブプログラム・パラメータ 8.7
    - コレクション 5.10
    - コンポジット変数 5
    - 実際のカーソル式 6.5
    - カーソル変数 6.4.8
    - オプション 8.7.5
    - 問合せ結果 6.4.8
    - 必須 8.7.5
- サブプログラムのプロパティ 8.4
- 副問合せ
    - 相関 6.3.4
    - 結果セットの処理 6.3.4
- SUBSCRIPT_BEYOND_COUNT例外 11.4
- SUBSCRIPT_OUTSIDE_LIMIT例外 11.4
- サブタイプ 3
    - 「データ型」も参照
    - 制約付き 3.4.2
        - サブプログラム・パラメータ 8.7.1.1
    - BINARY_DOUBLEデータ型 3.1.3
    - BINARY_FLOATデータ型 3.1.3
    - PLS_INTEGERデータ型 3.3.2
    - 事前定義 E
    - 無制約 3.4.1
    - ユーザー定義 3.4
- シノニム 2.4
- SYS_INVALID_ROWID例外 11.4
- SYS_REFCURSOR型 6.4.1
- システム・トリガー 9.5

T

- 表
    - ハッシュ 5.1
    - 索引付き
        - 「連想配列」を参照 5.2
    - 変更 9.10.3
    - ネスト 5.4
        - 特性 5.1
    - 大規模用のパラレルDML 12.7
    - PL/SQL
        - 「連想配列」を参照 5.2
    - 格納 5.4.2

    - 不規則 5.1
    - 大規模な表のパラレルでの更新 12.7
- 表の別名
    - 内部取得の回避 B.6
    - 行の式 B.6.2
    - 表の要素の属性またはメソッド B.6.1
- テーブル・ファンクション 12.5.1
    - パイプライン化
        - 「パイプライン・テーブル・ファンクション」を参照 12.5
    - 弱いカーソル変数の引数 6.4.1
- TABLE演算子 6.4.6
- TCL文 6.1.1
    - トリガーによるサブプログラムの起動 9.6
    - トリガー 6.7.6
- テンプレート・オブジェクト 8.14.2
- TIMEOUT_ON_RESOURCE例外 11.4
- タイミング
    - DMLトリガー
        - 複合 9.3.3.1
        - 単純 9.3
    - システム・トリガー 9.5
    - トリガーの起動順序 9.11
- TO_NUMBERファンクション 7.3.4
- TO_REFCURSORファンクション 7.3.3
- TOO_MANY_ROWS例外 11.4
- トレース・ファイル 11
- プログラムのトレースおよびプロファイル 12.9
- トランザクション
    - 自律型 6.7
        - パイプライン・テーブル・ファンクション 13.48
    - コンテキスト 6.7.2
    - 終了
        - COMMIT文の使用 6.6.1
        - ROLLBACK文の使用 6.6.2
    - 分離レベル 6.7.3
    - ネスト 6.7
    - 読取り専用 6.6.5
    - 読取り/書込み 6.6.5
    - 例外処理後の再試行 11.12
    - SQL%ROWCOUNTカーソル属性 6.2.1.4
    - 可視性 6.7.3
- トランザクション制御言語
    - 「TCL文」を参照

- TRANSACTIONS初期化パラメータ 6.7.5.4
- トリガー 9
  - セキュリティ・メカニズム 9.2
  - AUTHIDプロパティ 8.14
  - 自律型 6.7.6
  - カスケード 9.11
  - DDL文 6.7.6
  - 実装上の細部の隠ぺい A.1
  - マテリアライズド・ビュー 14.12
  - 再帰的 9.9
  - TCL文 6.7.6
- TRIMコレクション・メソッド 5.10.2
- 3値論理 2.7.3
- PL/SQLコードのチューニング 12
- 型
  - 「データ型」を参照
- 型互換のデータ型
  - コレクション変数 5.7.1
  - スカラー変数 2.6
- typemark 5.6

U

- UDFプラグマ 13.66
- 無制約サブタイプ 3.4.1
- アンダースコア(_)ワイルドカード文字 2.7.5.3
- 未処理例外 11.9
  - FORALL文 12.4.1.2
- 順序付けされていない表 5.1
- UPDATE文 13.67
  - 「DML文」も参照:
  - BEFORE文トリガー 9.9
  - PL/SQLの拡張機能 13.67
  - レコードの値の使用 5.16
    - 制限 5.17
- UPDATING条件述語 9.3.1
- UROWIDデータ型 3.1.6
- ユーザー定義の例外 11.5
  - エラー・コードの指定 11.7.2
  - 呼出し
    - RAISE_APPLICATION_ERRORプロシージャの使用 11.7.2
    - RAISE文 11.7.1.1
- ユーザー定義の識別子 2.2.2.3

    - データの収集 12.8
- ユーザー定義のサブタイプ 3.4
- ユーザー定義型
    - 「抽象データ型(ADT)」を参照:
- USING_NLS_COMP 13.20
- utlrp.sqlスクリプト 12.10.5

V

- V$RESERVED_WORDSビュー D
- SQLインジェクション回避のための妥当性チェック 7.4.2.2
- VALUE_ERROR例外 11.4
- VARCHAR2データ型 3.1.4
- 静的なVARCHAR2式 2.7.9.3
- VARCHARサブタイプ 3.1.4.3.1
- 変数
    - バインド 12.4
    - BOOLEAN 2.6.4
    - コレクション
        - 「コレクション」を参照 5
    - コンポジット 5
    - カーソル
        - 「カーソル変数」を参照 6.4
    - ホスト
        - カーソル変数 6.4.9
        - パッケージ 10.2
    - カーソル変数問合せ 6.4.5
    - 明示カーソル問合せ 6.2.2.4
    - ロケータ 11.2.2
    - レコード
        - 「レコード」を参照 5
    - リモート・パブリック・パッケージ 10.3.1
    - スカラー
        - 「スカラー変数」を参照 2.3.2
    - 未定義の値 6.1.1
- 可変サイズの配列
    - 「VARRAY」を参照
- 可変個引数疑似演算子 12.6.3.1
- VARRAY 5.3
    - 「コレクション」も参照:
    - NULL値の代入 5.7.2
    - 特性 5.1
    - NULLとの比較 5.9.1

    - COUNTメソッド 5.10.6.1
    - FIRSTおよびLASTメソッド 5.10.5.2
    - ファンクションによって戻される 12.5.1
- VERSION定数 2.9.1.5
- ビュー
    - AUTHIDプロパティ 8.14
    - INSTEAD OFトリガー 14.12
    - マテリアライズドとトリガー 14.12
- 仮想列
    - %ROWTYPE属性 5.12.4.3
    - 明示カーソル 6.2.2.5
- 可視性
    - 識別子 2.5
    - トランザクション 6.7.3

W

- 警告, コンパイル時 11.1
- 弱いREF CURSOR型
    - 作成 6.4.1
    - FETCH文 6.4.3
- WHILE LOOP文 4.2.7
    - 構文図 13.68
- ホワイト・リスト
    - 「ACCESSIBLE BY句」を参照
- 空白文字
    - 字句単位間 2.2.6
    - 文字リテラル 2.2.3
    - データベース文字セット 2.1.1
- ワイルドカード文字 2.7.5.3
- WRAPファンクション A.4
- PL/SQLソース・テキストのラップ A
    - 問合せディレクティブ 2.9.1.4.3
- wrapユーティリティ
    - 「PL/SQLラッパー・ユーティリティ」を参照:

Z

- ZERO_DIVIDE例外 11.4
- 長さが0 (ゼロ)の文字列 2.2.3