Pro*C/C++
プログラマーズ・ガイド
19c

F16160-01(原本部品番号:E96467-01)

2019年1月

タイトルおよび著作権情報

Pro*C/C++プログラマーズ・ガイド, 19c

F16160-01

原本著者: Celin Cherian

原本主協力者: Jack Melnick、Neelam Singh、Tim Smith、Paul Lane

原本協力者: Bill Bailey、Subhranshu Banerjee、Julie Basu、Beethoven Chang、Michael Chiocca、Nancy Ikeda、Alex Keh、Thomas Kurian、Shiao-Yen Lin、Valarie Moore、Vidya Nagaraj、Ajay Popat、Ekkehard Rohwedder、Pamela Rothman、Gael Stevens

する情報を提供することがあります。適用されるお客様とOracle Corporationとの間の契約に別段の定めがある場合を除いて、Oracle Corporationおよびその関連会社は、第三者のコンテンツ、製品、サービスに関して一切の責任を負わず、いかなる保証もいたしません。適用されるお客様とOracle Corporationとの間の契約に定めがある場合を除いて、Oracle Corporationおよびその関連会社は、第三者のコンテンツ、製品、サービスへのアクセスまたは使用によって損失、費用、あるいは損害が発生しても一切の責任を負いかねます。

目次

- 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について
    - 関連ドキュメント
    - 表記規則
- 『Pro*C/C++プログラマーズ・ガイド』のこのリリースでの変更点
    - Pro*C/C++リリース18cバージョン18.1での変更点
        - 新機能
    - Pro*C/C++ 12cリリース2 (12.2)での変更点
        - 新機能
- 第I部 概要および概念
    - 1 概要
        - 1.1 Oracleプリコンパイラ
        - 1.2 Oracle Pro*C/C++プリコンパイラを使用する理由
        - 1.3 SQLを使用する理由
        - 1.4 PL/SQLを使用する理由
        - 1.5 Pro*C/C++プリコンパイラの利点
        - 1.6 ディレクトリ構造
            - 1.6.1 既知の問題、制限および回避策
        - 1.7 ライブラリ・ファイル
        - 1.8 よくある質問(FAQ)
            - 1.8.1 VARCHARについて説明してください。
            - 1.8.2 Pro*C/C++はOracle Call Interfaceのコールを生成しますか。
            - 1.8.3 Pro*C/C++を使用せず、SQLLIBコールを使用してコーディングできますか。
            - 1.8.4 PL/SQLのストアド・プロシージャをPro*C/C++プログラムからコールできますか。
            - 1.8.5 C++のコードを作成し、Pro*C/C++を使用してプリコンパイルできますか。
            - 1.8.6 SQL文の任意の場所でバインド変数を使用できますか。
            - 1.8.7 Pro*C/C++の文字処理がよくわかりません。
            - 1.8.8 文字ポインタについて何か特別なことはありますか。
            - 1.8.9 Pro*C/C++でSPOOLが動作しない理由を説明してください。
            - 1.8.10 サンプル・プログラムのオンライン版はどこにありますか。
            - 1.8.11 アプリケーションをコンパイルしてリンクする方法を教えてください。
            - 1.8.12 Pro*C/C++では構造体をホスト変数として使用できますか。
            - 1.8.13 再帰関数内で埋込みSQLを使用した場合、その再帰関数をPro*C/C++で使用できますか。
            - 1.8.14 Pro*C/C++のすべてのリリースを、Oracleサーバーのすべてのバージョンで使用できますか。
            - 1.8.15 アプリケーションを実行すると、必ずORA-01405エラー(フェッチした列の値がNULLです)が発生します。
            - 1.8.16 すべてのSQLLIB関数はプライベート関数ですか。

- 1.8.17 新しいオブジェクト型はOracleでどのようにサポートされていますか。
- 1.8.18 互換性、アップグレードおよび移行

- 2 プリコンパイラの概要
  - 2.1 埋込みSQLプログラムの主な概要
    - 2.1.1 埋込みSQL文
      - 2.1.1.1 実行文とディレクティブ
    - 2.1.2 埋込みSQLの構文
    - 2.1.3 静的SQL文と動的SQL文の対比
    - 2.1.4 埋込みPL/SQLブロック
    - 2.1.5 ホスト変数および標識変数
    - 2.1.6 Oracleのデータ型
    - 2.1.7 配列
    - 2.1.8 データ型の同値化
    - 2.1.9 プライベートSQL領域、カーソルおよびアクティブ・セット
    - 2.1.10 トランザクション
    - 2.1.11 エラーおよび警告
    - 2.1.12 SQL99構文サポート
  - 2.2 埋込みSQLアプリケーションの開発ステップ
  - 2.3 プログラミングのガイドライン
    - 2.3.1 コメント
    - 2.3.2 定数
    - 2.3.3 宣言部
    - 2.3.4 デリミタ
    - 2.3.5 ファイルの長さ
    - 2.3.6 関数プロトタイプ
      - 2.3.6.1 ANSI_C
      - 2.3.6.2 KR_C
      - 2.3.6.3 CPP
    - 2.3.7 ヒントの長さ
    - 2.3.8 ホスト変数名
    - 2.3.9 行の継続
    - 2.3.10 行の長さ
    - 2.3.11 MAXLITERALのデフォルト値
    - 2.3.12 演算子
    - 2.3.13 文の終了記号
  - 2.4 条件付きプリコンパイル
    - 2.4.1 記号の定義
    - 2.4.2 SELECT文の例
  - 2.5 分割プリコンパイル
    - 2.5.1 ガイドライン
      - 2.5.1.1 カーソルの参照
      - 2.5.1.2 MAXOPENCURSORSの指定
      - 2.5.1.3 単一のSQLCAの使用
  - 2.6 コンパイルおよびリンク
  - 2.7 サンプル表

        - 2.7.1 サンプル・データ
    - 2.8 サンプル・プログラム: 単純な問合せ
    - 2.9 サンプル・プログラム: SQL99構文を使用する単純な問合せ
- 3 データベースの概要
    - 3.1 データベースへの接続
        - 3.1.1 ALTER AUTHORIZATION句を使用したパスワードの変更
            - 3.1.1.1 標準CONNECT
            - 3.1.1.2 CONNECT文でのパスワードの変更
        - 3.1.2 Oracle Net Servicesを使用した接続
        - 3.1.3 自動接続
            - 3.1.3.1 AUTO_CONNECTプリコンパイラ・オプション
            - 3.1.3.2 SYSDBAまたはSYSOPERシステム権限
    - 3.2 高度な接続オプション
        - 3.2.1 予備知識
        - 3.2.2 同時ログイン
        - 3.2.3 デフォルトのデータベースおよび接続
        - 3.2.4 明示的接続
            - 3.2.4.1 単一の明示的接続
                - 3.2.4.1.1 SQL操作
                - 3.2.4.1.2 PL/SQLブロック
                - 3.2.4.1.3 カーソルの制御
                - 3.2.4.1.4 動的SQL
            - 3.2.4.2 複数の明示的接続
            - 3.2.4.3 データの整合性の確認
        - 3.2.5 暗黙的接続
            - 3.2.5.1 単一の暗黙的接続
            - 3.2.5.2 複数の暗黙的接続
    - 3.3 トランザクション用語の定義
    - 3.4 トランザクションがデータベースを保護する方法
    - 3.5 トランザクションの開始および終了方法
    - 3.6 COMMIT文の使用
        - 3.6.1 DECLARE CURSOR文のWITH HOLD句
        - 3.6.2 CLOSE_ON_COMMITプリコンパイラ・オプション
    - 3.7 SAVEPOINT文の使用
    - 3.8 ROLLBACK文
        - 3.8.1 文レベルのロールバック
    - 3.9 RELEASEオプション
    - 3.10 SET TRANSACTION文
    - 3.11 デフォルト・ロックの上書き
        - 3.11.1 FOR UPDATE OFの使用
            - 3.11.1.1 制限事項
        - 3.11.2 LOCK TABLEの使用
    - 3.12 コミットにまたがるフェッチ
    - 3.13 分散トランザクションの処理
    - 3.14 ガイドライン

- 3.14.1 アプリケーションの設計
- 3.14.2 ロックの取得
- 3.14.3 PL/SQLの使用

- 4 データ型とホスト変数
  - 4.1 Oracleのデータ型
    - 4.1.1 内部データ型
    - 4.1.2 外部データ型
      - 4.1.2.1 VARCHAR2
        - 4.1.2.1.1 入力時
        - 4.1.2.1.2 出力時
      - 4.1.2.2 NUMBER
      - 4.1.2.3 INTEGER
      - 4.1.2.4 FLOAT
      - 4.1.2.5 STRING
        - 4.1.2.5.1 入力時
        - 4.1.2.5.2 出力時
      - 4.1.2.6 VARNUM
      - 4.1.2.7 LONG
      - 4.1.2.8 VARCHAR
      - 4.1.2.9 ROWID
      - 4.1.2.10 DATE
      - 4.1.2.11 RAW
      - 4.1.2.12 VARRAW
      - 4.1.2.13 LONG RAW
      - 4.1.2.14 UNSIGNED
      - 4.1.2.15 LONG VARCHAR
      - 4.1.2.16 LONG VARRAW
      - 4.1.2.17 CHAR
        - 4.1.2.17.1 入力時
        - 4.1.2.17.2 出力時
      - 4.1.2.18 CHARZ
        - 4.1.2.18.1 入力時
        - 4.1.2.18.2 出力時
      - 4.1.2.19 CHARF
    - 4.1.3 その他の外部データ型
      - 4.1.3.1 日時および時間隔のデータ型
      - 4.1.3.2 ANSI DATE
      - 4.1.3.3 TIMESTAMP
      - 4.1.3.4 TIMESTAMP WITH TIME ZONE
      - 4.1.3.5 TIMESTAMP WITH LOCAL TIME ZONE
      - 4.1.3.6 INTERVAL YEAR TO MONTH
      - 4.1.3.7 INTERVAL DAY TO SECOND
      - 4.1.3.8 日時を使用した予期しない結果の回避
  - 4.2 ホスト変数
    - 4.2.1 ホスト変数の宣言

- 4.2.1.1 記憶域クラス指定子
- 4.2.1.2 型修飾子
- 4.2.2 ホスト変数の参照
- 4.2.2.1 制限事項
- 4.3 標識変数
- 4.3.1 INDICATORキーワード
- 4.3.2 標識変数の使用例
- 4.3.3 標識変数のガイドライン
- 4.3.4 Oracle制限事項
- 4.4 VARCHAR変数
- 4.4.1 VARCHAR変数の宣言
- 4.4.2 VARCHAR変数の参照
- 4.4.3 VARCHAR変数にNULLを戻す
- 4.4.4 VARCHAR変数を使用したNULLの挿入
- 4.4.5 関数へのVARCHAR変数の受渡し
- 4.4.6 VARCHAR配列コンポーネントの長さを調べる方法
- 4.4.7 サンプル・プログラム: sqlvcp()の使用
- 4.5 カーソル変数
- 4.5.1 カーソル変数の宣言
- 4.5.2 カーソル変数の割当て
- 4.5.3 カーソル変数のオープン
- 4.5.3.1 スタンドアロン・ストアド・プロシージャでのオープン
- 4.5.3.2 戻り型
- 4.5.4 カーソル変数のクローズと解放
- 4.5.5 OCIでのカーソル変数の使用(リリース7のみ)
- 4.5.6 制限(カーソル変数)
- 4.5.7 例: cv_demo.sqlおよびsample11.pc
- 4.5.7.1 cv_demo.sql
- 4.5.7.2 sample11.pc
- 4.6 CONTEXT変数
- 4.7 ユニバーサルROWID
- 4.7.1 SQLRowidGet()
- 4.8 ホスト構造体
- 4.8.1 ホスト構造体と配列
- 4.8.2 PL/SQLレコード
- 4.8.3 ネストした構造体と共用体
- 4.8.4 ホスト・インジケータ構造体
- 4.8.5 サンプル・プログラム: カーソルとホスト構造体
- 4.9 ポインタ変数
- 4.9.1 ポインタ変数の宣言
- 4.9.2 ポインタ変数の参照
- 4.9.3 構造体ポインタ
- 4.10 グローバリゼーション・サポート
- 4.11 NCHAR変数
- 4.11.1 CHARACTER SET [IS] NCHAR_CS

        - 4.11.2 環境変数NLS_NCHAR
        - 4.11.3 VAR文でのCONVBUFSZ句
        - 4.11.4 埋込みSQL内の文字列
        - 4.11.5 文字列の制限事項
        - 4.11.6 標識変数
- 5 高度なトピック
    - 5.1 文字データ
        - 5.1.1 CHAR_MAPプリコンパイラ・オプション
        - 5.1.2 CHAR_MAPオプションのインラインでの使用方法
        - 5.1.3 DBMSオプションおよびCHAR_MAPオプションの影響
            - 5.1.3.1 入力時
            - 5.1.3.2 入力時
            - 5.1.3.3 出力時
        - 5.1.4 VARCHAR変数およびポインタ
            - 5.1.4.1 入力時
            - 5.1.4.2 出力時
        - 5.1.5 Unicode変数
            - 5.1.5.1 Unicode変数の使用に関する制限事項
    - 5.2 データ型変換
    - 5.3 データ型の同値化
        - 5.3.1 ホスト変数の同値化
        - 5.3.2 ユーザー定義型同値化
            - 5.3.2.1 REFERENCE句
        - 5.3.3 CHARF外部データ型
        - 5.3.4 EXEC SQL VARおよびTYPEディレクティブ
        - 5.3.5 例: データ型の同値化(sample4.pc)
    - 5.4 Cプリプロセッサ
        - 5.4.1 Pro*C/C++プリプロセッサの機能
        - 5.4.2 プリプロセッサ・ディレクティブ
            - 5.4.2.1 無視されるディレクティブ
        - 5.4.3 ORA_PROCマクロ
        - 5.4.4 ヘッダー・ファイルの格納場所の指定
        - 5.4.5 プリプロセッサの例
            - 5.4.5.1 #defineの使用について
            - 5.4.5.2 他のプリプロセッサの制限
        - 5.4.6 #includeに使用できないSQL文
        - 5.4.7 SQLCA、ORACAおよびSQLDAの組込み
        - 5.4.8 EXEC SQL INCLUDEおよび#includeのまとめ
        - 5.4.9 定義済マクロ
        - 5.4.10 インクルード・ファイル
    - 5.5 プリコンパイル済ヘッダー・ファイル
        - 5.5.1 プリコンパイル済ヘッダー・ファイルの作成
        - 5.5.2 プリコンパイル済ヘッダー・ファイルの使用
        - 5.5.3 例
            - 5.5.3.1 冗長なファイル・インクルード

- 5.5.3.1.1 ケース1: 最上位のヘッダー・ファイルのインクルード
- 5.5.3.1.2 ケース2: ネストされたヘッダー・ファイルのインクルード
- 5.5.3.2 複数のプリコンパイル済ヘッダー・ファイル
- 5.5.4 ヘッダー・ファイルのリスト
- 5.5.5 オプションの効果
- 5.5.5.1 DEFINEおよびINCLUDEオプション
- 5.5.5.1.1 シングル・ユーザーの場合
- 5.5.5.1.2 マルチ・ユーザーの場合
- 5.5.5.2 CODEおよびPARSEオプション
- 5.5.6 使用上の注意
- 5.6 Oracleプリプロセッサ
- 5.6.1 記号の定義
- 5.6.2 Oracleプリプロセッサの例
- 5.7 数値定数の評価
- 5.7.1 Pro*C/C++での数値定数
- 5.7.2 数値定数の規則および例
- 5.8 OCIリリース8のSQLLIB拡張相互運用性
- 5.8.1 OCIリリース8環境でのランタイム・コンテキスト
- 5.8.2 OCIリリース8環境ハンドルのパラメータ
- 5.9 OCIリリース8へのインタフェース
- 5.9.1 SQLEnvGet()
- 5.9.2 SQLSvcCtxGet()
- 5.9.3 OCIリリース8コールの埋込み
- 5.10 OCIリリース7コールの埋込み
- 5.10.1 LDAの設定
- 5.10.2 リモートの複数接続
- 5.11 SQLLIBパブリック関数の新しい名前
- 5.12 X/Openアプリケーションの開発
- 5.12.1 Oracle固有の問題
- 5.12.1.1 Oracleへの接続
- 5.12.1.2 トランザクション制御
- 5.12.1.3 OCIコール(リリース7のみ)
- 5.12.1.4 リンク
- 6 埋込みSQL
- 6.1 ホスト変数
- 6.1.1 出力ホスト変数および入力ホスト変数
- 6.2 標識変数
- 6.2.1 NULL値の挿入
- 6.2.2 戻されたNULL値
- 6.2.3 NULL値のフェッチ
- 6.2.4 NULLのテスト
- 6.2.5 切り捨てられた値
- 6.3 基本的なSQL文
- 6.3.1 SELECT文
- 6.3.1.1 使用可能な句

        - 6.3.2 INSERT文
            - 6.3.2.1 副問合せの使用について
        - 6.3.3 UPDATE文
        - 6.3.4 DELETE文
        - 6.3.5 WHERE句
    - 6.4 DML RETURNING句
    - 6.5 カーソル
        - 6.5.1 DECLARE CURSOR文
        - 6.5.2 OPEN文
        - 6.5.3 FETCH文
        - 6.5.4 CLOSE文
    - 6.6 スクロール可能カーソル
        - 6.6.1 スクロール可能なカーソルの使用について
            - 6.6.1.1 DECLARE SCROLL CURSOR
            - 6.6.1.2 スクロール可能カーソル用のOPEN
            - 6.6.1.3 スクロール可能カーソル用のFETCH
            - 6.6.1.4 スクロール可能カーソル用のCLOSE
        - 6.6.2 CLOSE_ON_COMMITプリコンパイラ・オプション
        - 6.6.3 PREFETCHプリコンパイラ・オプション
    - 6.7 オプティマイザ・ヒント
        - 6.7.1 ヒントの発行
    - 6.8 実行計画の修正
        - 6.8.1 SQLファイル
            - 6.8.1.1 例
        - 6.8.2 LOGファイル
    - 6.9 CURRENT OF句
        - 6.9.1 制限事項(FOR UPDATE OF)
    - 6.10 カーソル文
    - 6.11 スクロール不可カーソルを使用する完全な例
    - 6.12 スクロール可能カーソルを使用する完全な例
    - 6.13 位置付け更新
- 7 埋込みPL/SQL
    - 7.1 PL/SQLの利点
        - 7.1.1 パフォーマンス向上
        - 7.1.2 Oracleとの統合
        - 7.1.3 カーソルFORループ
        - 7.1.4 プロシージャとファンクション
        - 7.1.5 パッケージ
        - 7.1.6 PL/SQL表
        - 7.1.7 ユーザー定義のレコード
    - 7.2 埋込みPL/SQLブロック
    - 7.3 ホスト変数
        - 7.3.1 例: PL/SQLでのホスト変数の使用
        - 7.3.2 複雑な例
        - 7.3.3 VARCHAR疑似型

- 7.3.4 制限事項
- 7.4 標識変数
  - 7.4.1 NULLの処理
  - 7.4.2 切り捨てられた値
- 7.5 ホスト配列
  - 7.5.1 ARRAYLEN文
  - 7.5.2 オプション・キーワードEXECUTE
- 7.6 埋込みPL/SQLのカーソルの使用方法
- 7.7 ストアドPL/SQLおよびJavaサブプログラム
  - 7.7.1 ストアド・サブプログラムの作成について
  - 7.7.2 ストアドPL/SQLまたはJavaサブプログラムのコールについて
    - 7.7.2.1 無名PL/SQLブロック
    - 7.7.2.2 リモート・アクセス
    - 7.7.2.3 CALL文
    - 7.7.2.4 CALLの例
  - 7.7.3 ストアド・サブプログラムに関する情報を得る方法について
- 7.8 外部プロシージャ
  - 7.8.1 外部プロシージャの制限
  - 7.8.2 外部プロシージャの作成について
  - 7.8.3 SQLExtProcError()
- 7.9 動的SQLの使用方法について

- 8 ホスト配列
  - 8.1 配列を使用する理由
  - 8.2 ホスト配列の宣言について
    - 8.2.1 制限事項(ホスト配列の宣言)
    - 8.2.2 配列の最大サイズ
  - 8.3 SQL文での配列の使用方法について
    - 8.3.1 ホスト配列の参照について
    - 8.3.2 インジケータ配列の使用方法について
    - 8.3.3 Oracle制限事項(ホスト配列用)
    - 8.3.4 ANSI制限事項および要件
  - 8.4 配列への選択について
    - 8.4.1 カーソルのフェッチ
    - 8.4.2 sqlca.sqlerrd[2]の使用方法について
    - 8.4.3 FETCHされる行数
    - 8.4.4 スクロール可能カーソルのフェッチ
    - 8.4.5 サンプル・プログラム3: ホスト配列
    - 8.4.6 サンプル・プログラム: スクロール可能カーソルを使用するホスト配列
      - 8.4.6.1 scdemo2.pc
    - 8.4.7 ホスト配列の制限
    - 8.4.8 NULL値のフェッチについて
    - 8.4.9 切り捨てられた値のフェッチについて
  - 8.5 配列での挿入について
    - 8.5.1 配列での挿入の制限について
  - 8.6 配列での更新について

- 8.6.1 配列での更新の制限について
- 8.7 配列での削除について
  - 8.7.1 配列での削除の制限について
- 8.8 FOR句の使用方法について
  - 8.8.1 FOR句の制限
    - 8.8.1.1 SELECT文中
    - 8.8.1.2 CURRENT OF句を使用する場合
- 8.9 WHERE句の使用方法について
- 8.10 構造体配列
  - 8.10.1 構造体の配列の使用方法
  - 8.10.2 構造体の配列の制限
  - 8.10.3 構造体の配列の宣言について
  - 8.10.4 変数のガイドライン
  - 8.10.5 構造体配列へのポインタの宣言について
  - 8.10.6 例
    - 8.10.6.1 例1: スカラー構造体の単純な配列
    - 8.10.6.2 例2: スカラーの配列と構造体の配列との組合せ使用
    - 8.10.6.3 例3: 複数の構造体配列と1つのカーソルとの組合せ使用
      - 8.10.6.3.1 FOR句の使用方法について
    - 8.10.6.4 例4: 個々の配列メンバーおよび構造体メンバーの参照
    - 8.10.6.5 例5: 標識変数の使用(特殊な場合)
    - 8.10.6.6 例6: 構造体配列へのポインタの使用
- 8.11 CURRENT OFの疑似実行について
- 8.12 追加の配列の挿入/選択構文の使用について
- 8.13 暗黙的なバッファ済INSERTの使用について
- 8.14 スクロール可能カーソル

- 9 ランタイム・エラーの処理
  - 9.1 エラー処理の必要性
  - 9.2 エラー処理の代替手段
    - 9.2.1 状態変数
    - 9.2.2 SQL通信領域
  - 9.3 SQLSTATE状態変数
    - 9.3.1 SQLSTATEの宣言について
    - 9.3.2 SQLSTATE値
    - 9.3.3 SQLSTATEの使用について
      - 9.3.3.1 SQLSTATEを宣言する場合
      - 9.3.3.2 SQLSTATEを宣言しない場合
  - 9.4 SQLCODEの宣言について
  - 9.5 SQLCAを使用したエラー・レポートの主要コンポーネント
    - 9.5.1 ステータス・コード
    - 9.5.2 警告フラグ
    - 9.5.3 処理済行数
    - 9.5.4 解析エラー・オフセット
    - 9.5.5 エラー・メッセージ・テキスト
  - 9.6 SQL通信領域(SQLCA)の使用

- 9.6.1 SQLCAの宣言について
- 9.6.2 SQLCAの内容
- 9.6.3 SQLCAの構造体
    - 9.6.3.1 sqlcaid
    - 9.6.3.2 sqlcabc
    - 9.6.3.3 sqlcode
    - 9.6.3.4 sqlerrm
    - 9.6.3.5 sqlerrp
    - 9.6.3.6 sqlerrd
    - 9.6.3.7 sqlwarn
    - 9.6.3.8 sqlext
- 9.6.4 PL/SQLの考慮事項
- 9.7 エラー・メッセージの全文の取得について
- 9.8 WHENEVERディレクティブの使用について
    - 9.8.1 WHENEVERの条件
        - 9.8.1.1 SQLWARNING
        - 9.8.1.2 SQLERROR
        - 9.8.1.3 NOT FOUND
    - 9.8.2 WHENEVERのアクション
        - 9.8.2.1 CONTINUE
        - 9.8.2.2 DO
        - 9.8.2.3 DO BREAK
        - 9.8.2.4 DO CONTINUE
        - 9.8.2.5 GOTO label_name
        - 9.8.2.6 STOP
    - 9.8.3 WHENEVERの例
    - 9.8.4 DO BREAKとDO CONTINUEの利用
    - 9.8.5 WHENEVER文の適用範囲
    - 9.8.6 WHENEVERのガイドライン
        - 9.8.6.1 文の位置
        - 9.8.6.2 データの終わり条件の処理
        - 9.8.6.3 無限ループの回避について
        - 9.8.6.4 アドレス指定可能度の維持について
        - 9.8.6.5 エラー後の戻りについて
- 9.9 SQL文のテキスト取得について
    - 9.9.1 制限(SQLStmtGetText()の使用)
    - 9.9.2 サンプル・プログラム
- 9.10 Oracle通信領域(ORACA)の使用について
    - 9.10.1 ORACAの宣言について
    - 9.10.2 ORACAの有効化について
    - 9.10.3 ORACAの内容
    - 9.10.4 ランタイム・オプションの選択について
    - 9.10.5 ORACAの構造体
        - 9.10.5.1 oracaid
        - 9.10.5.2 oracabc

- 9.10.5.3 oracchf
- 9.10.5.4 oradbgf
- 9.10.5.5 orahchf
- 9.10.5.6 orastxtf
- 9.10.5.7 診断情報
- 9.10.5.8 orastxt
- 9.10.5.9 orasfnm
- 9.10.5.10 oraslnr
- 9.10.5.11 カーソル・キャッシュ統計情報
- 9.10.5.12 orahoc
- 9.10.5.13 oramoc
- 9.10.5.14 oracoc
- 9.10.5.15 oranor
- 9.10.5.16 oranpr
- 9.10.5.17 oranex

- 9.10.6 ORACAの例

- 10 プリコンパイラのオプション
  - 10.1 プリコンパイラのコマンド
    - 10.1.1 大/小文字区別
  - 10.2 プリコンパイラのオプション
    - 10.2.1 環境変数
    - 10.2.2 構成ファイル
    - 10.2.3 オプション値の優先順位
    - 10.2.4 マクロ・オプションおよびマイクロ・オプション
    - 10.2.5 プリコンパイル中の状況
    - 10.2.6 オプションの適用範囲
    - 10.2.7 Windowsプラットフォーム用Pro*C/C++プリコンパイラの問題
      - 10.2.7.1 構成ファイル
      - 10.2.7.2 CODE
      - 10.2.7.3 DBMS
      - 10.2.7.4 INCLUDE
      - 10.2.7.5 PARSE
  - 10.3 クイック・リファレンス
  - 10.4 オプションの入力
    - 10.4.1 コマンドライン
    - 10.4.2 インライン
      - 10.4.2.1 EXEC ORACLEの用途
      - 10.4.2.2 EXEC ORACLEの適用範囲
    - 10.4.3 列プロパティのサポート
  - 10.5 プリコンパイラ・オプションの使用方法について
    - 10.5.1 AUTO_CONNECT
    - 10.5.2 CHAR_MAP
    - 10.5.3 CINCR
    - 10.5.4 CLOSE_ON_COMMIT
    - 10.5.5 CMAX

- 10.5.6 CMIN
- 10.5.7 CNOWAIT
- 10.5.8 CODE
- 10.5.9 COMMON_PARSER
- 10.5.10 COMP_CHARSET
- 10.5.11 CONFIG
- 10.5.12 CPOOL
- 10.5.13 CPP_SUFFIX
- 10.5.14 CTIMEOUT
- 10.5.15 DB2_ARRAY
- 10.5.16 DBMS
- 10.5.17 DEF_SQLCODE
- 10.5.18 DEFINE
- 10.5.19 DURATION
- 10.5.20 DYNAMIC
- 10.5.21 ERRORS
- 10.5.22 ERRTYPE
- 10.5.23 EVENTS
- 10.5.24 FIPS
- 10.5.25 HEADER
- 10.5.26 HOLD_CURSOR
- 10.5.27 IMPLICIT_SVPT
- 10.5.28 INAME
- 10.5.29 INCLUDE
- 10.5.30 INTYPE
- 10.5.31 LINES
- 10.5.32 LNAME
- 10.5.33 LTYPE
- 10.5.34 MAX_ROW_INSERT
- 10.5.35 MAXLITERAL
- 10.5.36 MAXOPENCURSORS
- 10.5.37 MODE
- 10.5.38 NATIVE_TYPES
- 10.5.39 NLS_CHAR
- 10.5.40 NLS_LOCAL
- 10.5.41 OBJECTS
- 10.5.42 ONAME
- 10.5.43 ORACA
- 10.5.44 OUTLINE
- 10.5.45 OUTLNPREFIX
- 10.5.46 PAGELEN
- 10.5.47 PARSE
- 10.5.48 PLAN_BASELINE
- 10.5.49 PLAN_PREFIX
- 10.5.50 PLAN_RUN

- 10.5.51 PLAN_FIXED
- 10.5.52 PLAN_ENABLED
- 10.5.53 MEMFORPREFETCH
- 10.5.54 PREFETCH
- 10.5.55 RELEASE_CURSOR
- 10.5.56 RUNOUTLINE
- 10.5.57 SELECT_ERROR
- 10.5.58 STMT_CACHE
- 10.5.59 SYS_INCLUDE
- 10.5.60 THREADS
- 10.5.61 TYPE_CODE
- 10.5.62 UNSAFE_NULL
- 10.5.63 USERID
- 10.5.64 UTF16_CHARSET
- 10.5.65 VARCHAR
- 10.5.66 VERSION

- 11 マルチスレッド・アプリケーション
  - 11.1 スレッド
  - 11.2 Pro*C/C++のランタイム・コンテキスト
  - 11.3 ランタイム・コンテキストの使用モデル
    - 11.3.1 単一のランタイム・コンテキストを共有する複数のスレッド
    - 11.3.2 複数のランタイム・コンテキストを共有する複数のスレッド
  - 11.4 マルチスレッド・アプリケーションのユーザー・インタフェース
    - 11.4.1 THREADSオプション
    - 11.4.2 埋込みSQL文およびディレクティブ
      - 11.4.2.1 EXEC SQL ENABLE THREADS
      - 11.4.2.2 EXEC SQL CONTEXT ALLOCATE
      - 11.4.2.3 EXEC SQL CONTEXT USE
      - 11.4.2.4 EXEC SQL CONTEXT FREE
    - 11.4.3 CONTEXT USEの例
    - 11.4.4 プログラミングの考慮事項
  - 11.5 マルチスレッドの例
  - 11.6 接続プーリング
    - 11.6.1 接続プーリング機能の使用方法について
      - 11.6.1.1 接続プーリングを使用可能にする方法
      - 11.6.1.2 接続プーリングのコマンドライン・オプション
      - 11.6.1.3 例
      - 11.6.1.4 パフォーマンス・チューニング
    - 11.6.2 デモ・プログラム:1
      - 11.6.2.1 例
    - 11.6.3 デモ・プログラム:2
      - 11.6.3.1 ケース1: CMINを変更
      - 11.6.3.2 ケース2: CMAXを変更
      - 11.6.3.3 例

- 第II部 アプリケーション

- 12 C++アプリケーション
  - 12.1 C++サポートの理解
    - 12.1.1 特殊なマクロ処理は不要
  - 12.2 C++のプリコンパイル
    - 12.2.1 コードの生成
    - 12.2.2 コードの解析について
    - 12.2.3 出力ファイル名の拡張子
    - 12.2.4 システム・ヘッダー・ファイル
  - 12.3 サンプル・プログラム
    - 12.3.1 cppdemo1.pc
    - 12.3.2 cppdemo2.pc
    - 12.3.3 cppdemo3.pc
- 13 Oracle動的SQL
  - 13.1 動的SQL
  - 13.2 動的SQLの長所と短所
  - 13.3 動的SQLの使用
  - 13.4 動的SQL文の要件
  - 13.5 動的SQL文の処理方法
  - 13.6 動的SQLの使用方法
    - 13.6.1 方法 1
    - 13.6.2 方法2
    - 13.6.3 方法3
    - 13.6.4 方法4
    - 13.6.5 ガイドライン
      - 13.6.5.1 一般的なエラーの回避について
  - 13.7 方法1の使用方法
    - 13.7.1 サンプル・プログラム: 動的SQL方法1
  - 13.8 方法2の使用方法
    - 13.8.1 USING句
    - 13.8.2 サンプル・プログラム: 動的SQL方法2
  - 13.9 方法3の使用方法
    - 13.9.1 PREPARE (動的SQL)
    - 13.9.2 DECLARE (動的SQL)
    - 13.9.3 OPEN (動的SQL)
    - 13.9.4 FETCH (動的SQL)
    - 13.9.5 CLOSE (動的SQL)
    - 13.9.6 サンプル・プログラム: 動的SQL方法3
  - 13.10 方法4の使用方法
    - 13.10.1 SQLDAの必要性
    - 13.10.2 DESCRIBE文
    - 13.10.3 SQLDA
    - 13.10.4 Oracle方法4の実装について
    - 13.10.5 制限事項
  - 13.11 DECLARE STATEMENT文の使用方法について
    - 13.11.1 ホスト配列の使用について

- 13.12 PL/SQLの使用について
  - 13.12.1 方法1の場合
  - 13.12.2 方法2の場合
  - 13.12.3 方法3の場合
  - 13.12.4 Oracle方法4の場合
- 13.13 動的SQL文のキャッシュ

- 14 ANSI動的SQL
  - 14.1 ANSI動的SQLの基本
    - 14.1.1 プリコンパイラのオプション
  - 14.2 ANSI SQL文の概要
    - 14.2.1 サンプル・コード
  - 14.3 Oracle拡張機能
    - 14.3.1 参照セマンティクス
    - 14.3.2 配列を使用したバルク操作について
    - 14.3.3 構造体配列のサポート
    - 14.3.4 オブジェクト型のサポート
  - 14.4 ANSI動的SQLプリコンパイラ・オプション
  - 14.5 動的SQL文の完全な構文
    - 14.5.1 ALLOCATE DESCRIPTOR
    - 14.5.2 DEALLOCATE DESCRIPTOR
    - 14.5.3 GET DESCRIPTOR
    - 14.5.4 SET DESCRIPTOR
    - 14.5.5 PREPAREの使用
    - 14.5.6 DESCRIBE INPUT
    - 14.5.7 DESCRIBE OUTPUT
    - 14.5.8 EXECUTE
    - 14.5.9 EXECUTE IMMEDIATEの使用
    - 14.5.10 DYNAMIC DECLARE CURSORの使用
    - 14.5.11 カーソルのOPEN
    - 14.5.12 FETCH
    - 14.5.13 動的カーソルのCLOSE
    - 14.5.14 旧バージョンのOracle動的方法4との相違点
    - 14.5.15 制限事項(ANSI動的SQL)
  - 14.6 サンプル・プログラム
    - 14.6.1 ansidyn1.pc
    - 14.6.2 ansidyn2.pc
- 15 Oracle動的SQL: 方法4
  - 15.1 方法4の特殊要件
    - 15.1.1 方法4が特別な理由
    - 15.1.2 Oracleに必要な情報
    - 15.1.3 情報の格納位置
    - 15.1.4 SQLDAの参照方法
    - 15.1.5 情報の取得方法
  - 15.2 SQLDAの説明
    - 15.2.1 SQLDAの用途

        - 15.2.2 複数のSQLDA
        - 15.2.3 SQLDAの宣言
        - 15.2.4 SQLDAの割当て
    - 15.3 SQLDA変数の使用について
        - 15.3.1 N変数
        - 15.3.2 V変数
        - 15.3.3 L変数
        - 15.3.4 T変数
        - 15.3.5 I変数
        - 15.3.6 F変数
        - 15.3.7 S変数
        - 15.3.8 M変数
        - 15.3.9 C変数
        - 15.3.10 X変数
        - 15.3.11 Y変数
        - 15.3.12 Z変数
    - 15.4 予備知識
        - 15.4.1 データの変換
            - 15.4.1.1 内部データ型
            - 15.4.1.2 外部データ型
        - 15.4.2 データ型の強制変換
            - 15.4.2.1 精度と位取りの抽出
        - 15.4.3 NULL/NOT NULLデータ型の処理
    - 15.5 基本ステップ
    - 15.6 各ステップの詳細
        - 15.6.1 ホスト文字列の宣言
        - 15.6.2 SQLDAの宣言
        - 15.6.3 記述子用の記憶域の割当て
        - 15.6.4 DESCRIBEへの最大数の設定
        - 15.6.5 ホスト文字列への問合せテキストの設定
        - 15.6.6 ホスト文字列からの問合せのPREPARE
        - 15.6.7 カーソルのDECLARE
        - 15.6.8 バインド変数のDESCRIBE
        - 15.6.9 プレースホルダの最大数の再設定
        - 15.6.10 バインド変数の値の取得と記憶域の割当て
        - 15.6.11 カーソルのOPEN
        - 15.6.12 選択リストのDESCRIBE
        - 15.6.13 選択リスト項目の最大数の再設定
        - 15.6.14 各選択リスト項目の長さとデータ型の再設定
        - 15.6.15 アクティブ・セットからの行のFETCH
        - 15.6.16 選択リストの値の取得と処理
        - 15.6.17 記憶域の割当て解除
        - 15.6.18 カーソルのCLOSE
        - 15.6.19 ホスト配列の使用について
        - 15.6.20 sample12.pc

- 15.7 サンプル・プログラム: 動的SQL方法4
- 15.8 サンプル・プログラム: スクロール可能カーソルを使用する動的SQL方法4

- 16 LOB
  - 16.1 LOB
    - 16.1.1 内部LOB
    - 16.1.2 外部LOB
    - 16.1.3 BFILEのセキュリティ
    - 16.1.4 LOBとLONGおよびLONG RAWの対比
    - 16.1.5 LOBロケータ
    - 16.1.6 テンポラリLOB
    - 16.1.7 LOBバッファリング・サブシステム
  - 16.2 プログラムでのLOBの使用方法
    - 16.2.1 LOBにアクセスする3種類の方法
    - 16.2.2 アプリケーションのLOBロケータ
    - 16.2.3 LOBの初期化
      - 16.2.3.1 内部LOB
      - 16.2.3.2 外部LOB
      - 16.2.3.3 テンポラリLOB
      - 16.2.3.4 LOBの解放
  - 16.3 LOB文のルール
    - 16.3.1 すべてのLOB文に対するルール
    - 16.3.2 LOBバッファリング・サブシステムに対するルール
    - 16.3.3 ホスト変数に対するルール
  - 16.4 LOB文
    - 16.4.1 APPEND
    - 16.4.2 ASSIGN
    - 16.4.3 CLOSE (LOBの場合)
    - 16.4.4 COPY
    - 16.4.5 CREATE TEMPORARY
    - 16.4.6 DISABLE BUFFERING
    - 16.4.7 ENABLE BUFFERING
    - 16.4.8 ERASE
    - 16.4.9 FILE CLOSE ALL
    - 16.4.10 FILE SET
    - 16.4.11 FLUSH BUFFER
    - 16.4.12 FREE TEMPORARY
    - 16.4.13 LOAD FROM FILE
    - 16.4.14 OPEN (LOBの場合)
    - 16.4.15 READ
    - 16.4.16 TRIM
    - 16.4.17 WRITE
    - 16.4.18 DESCRIBE
  - 16.5 LOBおよびナビゲーショナル・アクセス用インタフェース
    - 16.5.1 一時オブジェクト
    - 16.5.2 永続オブジェクト

- 16.5.3 ナビゲーショナル・アクセス用インタフェースの例
- 16.6 LOBプログラムの例
  - 16.6.1 BLOBのREADおよびファイル書込みの例
  - 16.6.2 ファイルの読込みおよびBLOBのWRITEの例
  - 16.6.3 lobdemo1.pc
- 17 オブジェクト
  - 17.1 オブジェクトの概要
    - 17.1.1 オブジェクト型
    - 17.1.2 オブジェクト型のREF
    - 17.1.3 型の継承
  - 17.2 Pro*C/C++でのオブジェクト型の使用について
    - 17.2.1 NULLインジケータ
  - 17.3 オブジェクト・キャッシュ
    - 17.3.1 永続オブジェクト対一時コピー
  - 17.4 連想アクセス用インタフェース
    - 17.4.1 連想アクセス用インタフェースを使用する場合
    - 17.4.2 ALLOCATE
    - 17.4.3 FREE
    - 17.4.4 CACHE FREE ALL
    - 17.4.5 連想アクセス用インタフェースを使用したオブジェクトへのアクセス
  - 17.5 ナビゲーショナル・アクセス用インタフェース
    - 17.5.1 ナビゲーショナル・アクセス用インタフェースを使用する場合
    - 17.5.2 ナビゲーション文に使用されるルール
    - 17.5.3 OBJECT CREATE
    - 17.5.4 OBJECT DEREF
    - 17.5.5 OBJECT RELEASE
    - 17.5.6 OBJECT DELETE
    - 17.5.7 OBJECT UPDATE
    - 17.5.8 OBJECT FLUSH
    - 17.5.9 オブジェクトへのナビゲーショナル・アクセス
  - 17.6 オブジェクト属性とC言語のデータ型の変換
    - 17.6.1 OBJECT SET
    - 17.6.2 OBJECT GET
  - 17.7 オブジェクト・オプションの設定/取得
    - 17.7.1 CONTEXT OBJECT OPTION SET
    - 17.7.2 CONTEXT OBJECT OPTION GET
  - 17.8 オブジェクトに対する新しいプリコンパイラ・オプション
    - 17.8.1 VERSION
    - 17.8.2 DURATION
    - 17.8.3 OBJECTS
    - 17.8.4 INTYPE
    - 17.8.5 ERRTYPE
    - 17.8.6 オブジェクトに対するSQLCHECKのサポート
    - 17.8.7 実行時のタイプ・チェック
  - 17.9 Pro*C/C++のオブジェクト例

        - 17.9.1 連想アクセス
        - 17.9.2 ナビゲーショナル・アクセス
    - 17.10 型の継承のサンプル・コード
    - 17.11 ナビゲーショナル・アクセスのサンプル・コード
    - 17.12 C言語の構造体の使用について
    - 17.13 REFの使用について
        - 17.13.1 REFのC言語の構造体の生成
        - 17.13.2 REFの宣言
        - 17.13.3 埋込みSQLでのREFの使用
    - 17.14 OCIDate、OCIString、OCINumberおよびOCIRawの使用について
        - 17.14.1 OCIDate、OCIString、OCINumber、OCIRawの宣言
        - 17.14.2 埋込みSQLでのOCI型の使用
        - 17.14.3 OCI型の操作
    - 17.15 Pro*C/C++の新しいデータベース型の概要
    - 17.16 動的SQLでのOracleデータ型使用の制限
- 18 コレクション
    - 18.1 コレクション
        - 18.1.1 ネストした表
        - 18.1.2 VARRAY
        - 18.1.3 C言語およびコレクション
    - 18.2 コレクションの記述子
        - 18.2.1 ホスト変数と標識変数の宣言
        - 18.2.2 コレクションの操作について
            - 18.2.2.1 自律型コレクション・アクセス
            - 18.2.2.2 コレクション要素アクセス
        - 18.2.3 アクセスのルール
            - 18.2.3.1 自律型アクセス
            - 18.2.3.2 要素アクセス
        - 18.2.4 標識変数
            - 18.2.4.1 自律型バインド
            - 18.2.4.2 要素バインド
    - 18.3 OBJECT GETおよびSET
    - 18.4 COLLECTION文
        - 18.4.1 COLLECTION GET
        - 18.4.2 COLLECTION SET
        - 18.4.3 COLLECTION RESET
        - 18.4.4 COLLECTION APPEND
        - 18.4.5 COLLECTION TRIM
        - 18.4.6 COLLECTION DESCRIBE
            - 18.4.6.1 表について
        - 18.4.7 コレクションを使用する場合の規則
    - 18.5 コレクション・サンプル・コード
        - 18.5.1 型および表の作成
        - 18.5.2 GETおよびSETの例
        - 18.5.3 DESCRIBEの例

- 18.5.4 RESETの例
- 18.5.5 サンプル・プログラム: coldemo1.pc

- 19 Object Type Translator
  - 19.1 OTT概要
  - 19.2 Object Type Translator(OTT)
    - 19.2.1 データベースでの型の作成について
    - 19.2.2 OTTの起動について
      - 19.2.2.1 コマンドライン
      - 19.2.2.2 構成ファイル
      - 19.2.2.3 INTYPEファイル
    - 19.2.3 OTTコマンドライン
      - 19.2.3.1 OTT
      - 19.2.3.2 Userid
      - 19.2.3.3 INTYPE
      - 19.2.3.4 OUTTYPE
      - 19.2.3.5 CODE
      - 19.2.3.6 HFILE
      - 19.2.3.7 INITFILE
      - 19.2.3.8 INITFUNC
    - 19.2.4 INTYPEファイル
    - 19.2.5 OTTデータ型のマッピング
      - 19.2.5.1 オブジェクト・データ型のC言語へのマッピング
      - 19.2.5.2 OTT型マッピングの例
    - 19.2.6 NULLインジケータ構造体
    - 19.2.7 OTTでの型の継承のサポート
      - 19.2.7.1 置換可能なオブジェクト属性
    - 19.2.8 OUTTYPEファイル
  - 19.3 OCIアプリケーションでのOTTの使用方法
    - 19.3.1 OCIでのオブジェクトへのアクセスおよび操作について
    - 19.3.2 初期化関数のコールについて
    - 19.3.3 初期化関数のタスク
  - 19.4 Pro*C/C++アプリケーションでのOTTの使用について
  - 19.5 OTT参照
    - 19.5.1 OTTコマンドライン構文
    - 19.5.2 OTTパラメータ
      - 19.5.2.1 USERID
      - 19.5.2.2 INTYPE
      - 19.5.2.3 OUTTYPE
      - 19.5.2.4 CODE
      - 19.5.2.5 INITFILE
      - 19.5.2.6 INITFUNC
      - 19.5.2.7 HFILE
      - 19.5.2.8 CONFIG
      - 19.5.2.9 ERRTYPE
      - 19.5.2.10 CASE

- 19.5.2.11 SCHEMA_NAMES
- 19.5.2.12 TRANSITIVE
- 19.5.3 OTTパラメータの使用場所
- 19.5.4 INTYPEファイルの構造
- 19.5.4.1 INTYPEファイルの型指定
- 19.5.5 ネストした#includeファイル生成
- 19.5.6 SCHEMA_NAMESの使用方法
- 19.5.7 デフォルト名のマッピング
- 19.5.8 制限
- 19.5.8.1 ファイル名比較
- 20 ユーザー・イグジット
- 20.1 ユーザー・イグジット
- 20.2 ユーザー・イグジットを作成する理由
- 20.3 ユーザー・イグジットの開発について
- 20.4 ユーザー・イグジットの作成について
- 20.4.1 変数の要件
- 20.5 EXEC TOOLS文
- 20.5.1 Toolsetユーザー・イグジットの作成について
- 20.5.2 EXEC TOOLS SET
- 20.5.3 EXEC TOOLS GET
- 20.5.4 EXEC TOOLS SET CONTEXT
- 20.5.5 EXEC TOOLS GET CONTEXT
- 20.5.6 EXEC TOOLS MESSAGE
- 20.6 ユーザー・イグジットのコールについて
- 20.7 ユーザー・イグジットへのパラメータの引渡しについて
- 20.8 フォームへの値の返却について
- 20.8.1 IAP定数
- 20.8.2 WHENEVERの使用について
- 20.9 ユーザー・イグジットの使用例
- 20.10 ユーザー・イグジットのプリコンパイルおよびコンパイルについて
- 20.11 サンプル・プログラム: ユーザー・イグジット
- 20.12 GENXTBユーティリティの使用について
- 20.13 SQL*Formsへのユーザー・イグジットのリンクについて
- 20.14 ガイドライン
- 20.14.1 イグジットの命名について
- 20.14.2 Oracleへの接続について
- 20.14.3 I/Oコールの発行について
- 20.14.4 ホスト変数の使用方法について
- 20.14.5 表の更新について
- 20.14.6 コマンドの発行について
- 付録
- A 予約語、キーワードおよびネームスペース
- A.1 予約語およびキーワード
- A.2 Oracleの予約済ネームスペース
- B パフォーマンス・チューニング

- B.1 パフォーマンスを低下させる原因
- B.2 パフォーマンスの改善方法
- B.3 ホスト配列の使用について
- B.4 埋込みPL/SQLの使用について
- B.5 SQL文の最適化について
  - B.5.1 オプティマイザ・ヒント
  - B.5.2 トレース機能
- B.6 文キャッシュについて
- B.7 索引の使用について
- B.8 行レベル・ロックの利用について
- B.9 不要な解析の排除について
  - B.9.1 明示カーソルの処理について
    - B.9.1.1 カーソルの制御
  - B.9.2 カーソル管理オプションの使用について
    - B.9.2.1 SQL領域とカーソル・キャッシュ
    - B.9.2.2 リソースの使用
    - B.9.2.3 実行回数が少ない場合
    - B.9.2.4 実行回数が多い場合
    - B.9.2.5 埋込みPL/SQLの考慮事項
    - B.9.2.6 パラメータの相互作用
- B.10 不要な再解析の回避について
- B.11 接続プーリングの使用方法について
- B.12 Traffic DirectorモードのOracle Connection Managerの使用について

- C 構文およびセマンティックのチェック
  - C.1 構文検査と意味検査
  - C.2 検査の種類および範囲の制御について
  - C.3 SQLCHECK=SEMANTICSの指定について
    - C.3.1 意味検査の使用許可について
      - C.3.1.1 Oracleサーバーへの接続について
      - C.3.1.2 DECLARE TABLEの使用について
      - C.3.1.3 DECLARE TYPEの使用について
  - C.4 SQLCHECK=SYNTAXの指定について
  - C.5 SQLCHECKオプションの入力について

- D システム固有の参照
  - D.1 システム固有の情報
    - D.1.1 標準ヘッダー・ファイルの位置
    - D.1.2 Cコンパイラ用インクルード・ファイルの位置指定について
    - D.1.3 ANSI Cサポート
    - D.1.4 構造体コンポーネントの位置合せ
    - D.1.5 整数とROWIDのサイズ
    - D.1.6 バイト順序
    - D.1.7 Oracleサーバーへの接続について
    - D.1.8 XAライブラリでのリンクについて
    - D.1.9 Pro*C/C++実行可能ファイルの位置
    - D.1.10 システム構成ファイル

        - D.1.11 INCLUDEオプションの構文
        - D.1.12 コンパイルとリンクについて
        - D.1.13 ユーザー・イグジット
- E 埋込みSQL文およびディレクティブ
    - E.1 プリコンパイラ・ディレクティブと埋込みSQL文の概要
    - E.2 文の説明
    - E.3 構文図の読み方
        - E.3.1 必須のキーワードとパラメータ
        - E.3.2 オプションのキーワードとパラメータ
        - E.3.3 構文ループ
        - E.3.4 複数パートの図
        - E.3.5 Oracleの名前
        - E.3.6 文の終了記号
    - E.4 ALLOCATE (実行可能埋込みSQL拡張機能)
    - E.5 ALLOCATE DESCRIPTOR (実行可能埋込みSQL)
    - E.6 CACHE FREE ALL (実行可能埋込みSQL拡張機能)
    - E.7 CALL (実行可能埋込みSQL)
    - E.8 CLOSE (実行可能埋込みSQL)
    - E.9 COLLECTION APPEND (実行可能埋込みSQL拡張機能)
    - E.10 COLLECTION DESCRIBE (実行可能埋込みSQL拡張機能)
    - E.11 COLLECTION GET (実行可能埋込みSQL拡張機能)
    - E.12 COLLECTION RESET (実行可能埋込みSQL拡張機能)
    - E.13 COLLECTION SET (実行可能埋込みSQL拡張機能)
    - E.14 COLLECTION TRIM (実行可能埋込みSQL拡張機能)
    - E.15 COMMIT (実行可能埋込みSQL)
    - E.16 CONNECT (実行可能埋込みSQL拡張機能)
    - E.17 CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)
    - E.18 CONTEXT FREE (実行可能埋込みSQL拡張機能)
    - E.19 CONTEXT OBJECT OPTION GET (実行可能埋込みSQL拡張機能)
    - E.20 CONTEXT OBJECT OPTION SET (実行可能埋込みSQL拡張機能)
    - E.21 CONTEXT USE (Oracle埋込みSQLディレクティブ)
    - E.22 DEALLOCATE DESCRIPTOR (埋込みSQL文)
    - E.23 DECLARE CURSOR (埋込みSQLディレクティブ)
    - E.24 DECLARE DATABASE (Oracle埋込みSQLディレクティブ)
    - E.25 DECLARE STATEMENT (埋込みSQLディレクティブ)
    - E.26 DECLARE TABLE (Oracle埋込みSQLディレクティブ)
    - E.27 DECLARE TYPE (Oracle埋込みSQLディレクティブ)
    - E.28 DELETE (実行可能埋込みSQL)
    - E.29 DESCRIBE (実行可能埋込みSQL拡張機能)
    - E.30 DESCRIBE DESCRIPTOR (実行可能埋込みSQL)
    - E.31 ENABLE THREADS (実行可能埋込みSQL拡張機能)
    - E.32 EXECUTE ... END-EXEC (実行可能埋込みSQL拡張機能)
    - E.33 EXECUTE (実行可能埋込みSQL)
    - E.34 EXECUTE DESCRIPTOR (実行可能埋込みSQL)
    - E.35 EXECUTE IMMEDIATE (実行可能埋込みSQL)

- E.36 FETCH (実行可能埋込みSQL)
- E.37 FETCH DESCRIPTOR (実行可能埋込みSQL)
- E.38 FREE (実行可能埋込みSQL拡張機能)
- E.39 GET DESCRIPTOR (実行可能埋込みSQL)
- E.40 INSERT (実行可能埋込みSQL)
- E.41 LOB APPEND (実行可能埋込みSQL拡張機能)
- E.42 LOB ASSIGN (実行可能埋込みSQL拡張機能)
- E.43 LOB CLOSE (実行可能埋込みSQL拡張機能)
- E.44 LOB COPY (実行可能埋込みSQL拡張機能)
- E.45 LOB CREATE TEMPORARY (実行可能埋込みSQL拡張機能)
- E.46 LOB DESCRIBE (実行可能埋込みSQL拡張機能)
- E.47 LOB DISABLE BUFFERING (実行可能埋込みSQL拡張機能)
- E.48 LOB ENABLE BUFFERING (実行可能埋込みSQL拡張機能)
- E.49 LOB ERASE (実行可能埋込みSQL拡張機能)
- E.50 LOB FILE CLOSE ALL (実行可能埋込みSQL拡張機能)
- E.51 LOB FILE SET (実行可能埋込みSQL拡張機能)
- E.52 LOB FLUSH BUFFER (実行可能埋込みSQL拡張機能)
- E.53 LOB FREE TEMPORARY (実行可能埋込みSQL拡張機能)
- E.54 LOB LOAD (実行可能埋込みSQL拡張機能)
- E.55 LOB OPEN (実行可能埋込みSQL拡張機能)
- E.56 LOB READ (実行可能埋込みSQL拡張機能)
- E.57 LOB TRIM (実行可能埋込みSQL拡張機能)
- E.58 LOB WRITE (実行可能埋込みSQL拡張機能)
- E.59 OBJECT CREATE (実行可能埋込みSQL拡張機能)
- E.60 OBJECT DELETE (実行可能埋込みSQL拡張機能)
- E.61 OBJECT DEREF (実行可能埋込みSQL拡張機能)
- E.62 OBJECT FLUSH (実行可能埋込みSQL拡張機能)
- E.63 OBJECT GET (実行可能埋込みSQL拡張機能)
- E.64 OBJECT RELEASE (実行可能埋込みSQL拡張機能)
- E.65 OBJECT SET (実行可能埋込みSQL拡張機能)
- E.66 OBJECT UPDATE (実行可能埋込みSQL拡張機能)
- E.67 OPEN (実行可能埋込みSQL)
- E.68 OPEN DESCRIPTOR (実行可能埋込みSQL)
- E.69 PREPARE (実行可能埋込みSQL)
- E.70 REGISTER CONNECT (実行可能埋込みSQL拡張機能)
- E.71 ROLLBACK (実行可能埋込みSQL)
- E.72 SAVEPOINT (実行可能埋込みSQL)
- E.73 SELECT (実行可能埋込みSQL)
- E.74 SET DESCRIPTOR (実行可能埋込みSQL)
- E.75 TYPE (Oracle埋込みSQLディレクティブ)
- E.76 UPDATE (実行可能埋込みSQL)
- E.77 VAR (Oracle埋込みSQLディレクティブ)
- E.78 WHENEVER (埋込みSQLディレクティブ)

- F サンプル・プログラム
  - F.1 サンプル・プログラムの説明

- F.2 サンプル表の作成
- F.3 サンプル・プログラムのビルドについて
  - F.3.1 pcmake.batを使用する方法
- F.4 Microsoft Visual Studioを使用する方法
- F.5 サンプル.preファイルのパスの設定
- G Microsoft Visual Studio .NETへのPro*C/C++の統合
  - G.1 Pro*C/C++のMicrosoft Visual Studio .NETプロジェクトへの統合
    - G.1.1 Pro*C/C++実行可能ファイルの位置の指定
    - G.1.2 Pro*C/C++ヘッダー・ファイルの位置の指定
  - G.2 プロジェクトへの.pcファイルの追加
    - G.2.1 プロジェクトへの.cファイルの参照の追加
    - G.2.2 プロジェクトへのPro*C/C++のライブラリの追加
    - G.2.3 カスタム・ビルド・オプションの指定
  - G.3 「ツール」メニューへのPro*C/C++の追加
- 索引

表一覧

- 1-1 precompディレクトリ構造
- 2-1 埋込みSQL文
- 2-2 **埋込みSQL文**
- 4-1 Oracle内部データ型
- 4-2 Oracle外部データ型
- 4-3 DATE書式
- 4-4 ホスト変数のC言語のデータ型
- 4-5 C言語とOracleのデータ型の互換性
- 4-6 **グローバリゼーション・サポート・パラメータ**
- 5-1 CHAR_MAPの設定
- 5-2 デフォルトの型割当て
- 5-3 ヘッダー・ファイル
- 5-4 SQLLIBパブリック関数 - 新しい名前
- 7-1 正当なデータ型変換
- 8-1 SELECT INTOで有効なホスト配列
- 8-2 UPDATE文で有効なホスト配列
- 8-3 DB2配列構文とOracleプリコンパイラ構文
- 9-1 事前定義済のクラス・コード
- 9-2 SQLSTATEステータス・コード
- 9-3 SQL機能コード
- 10-1 マクロ・オプション値によるマイクロ・オプション値の設定
- 10-2 プリコンパイラのオプション
- 10-3 DBMSとMODEの相互作用
- 11-1 接続プーリングのコマンドライン・オプション
- 12-1 PARSEオプションの値と効果
- 13-1 動的SQLの使用方法
- 14-1 ANSI SQLデータ型
- 14-2 DYNAMICオプションの設定
- 14-3 TYPE_CODEオプションの設定
- 14-4 GET DESCRIPTORの記述子項目名の定義
- 14-5 Oracle拡張機能により追加されたGET DESCRIPTORの記述子項目名の定義
- 14-6 SET DESCRIPTORの記述子項目名
- 14-7 Oracle拡張機能により追加されたSET DESCRIPTORの記述子項目名の定義
- 15-1 Oracle内部データ型
- 15-2 Oracle外部データ型とデータ型コード
- 15-3 SQLデータ型の精度と位取り
- 16-1 LOBアクセス方法
- 16-2 ソースLOBおよびプリコンパイラのデータ型
- 16-3 LOB属性
- 17-1 CONTEXT OBJECT OPTION値の有効な選択肢
- 17-2 Pro*C/C++での新しいデータベース型の使用
- 17-3 Pro*C/C++での新たなC言語のデータ型の使用

- 18-1　オブジェクト属性およびコレクション属性
- 18-2　コレクションおよびホスト配列で可能な型変換
- 18-3　COLLECTION DESCRIBEの属性
- 19-1　オブジェクト型属性のオブジェクト・データ型マッピング
- 19-2　コレクション型のオブジェクト・データ型マッピング
- A-1　Oracleの予約済ネームスペース
- B-1　HOLD_CURSORとRELEASE_CURSORの相互関係
- E-1　埋込みSQL文とディレクティブの機能概要
- E-2　プリコンパイラ・ディレクティブおよび埋込みSQL文および句
- F-1　サンプル・プログラム

はじめに

このドキュメントは総合的なユーザーズ・ガイドで、Pro*C/C++のリファレンスとして使用できます。Pro*C/C++とともにデータベース言語SQLおよびOracleのプロシージャ型拡張機能PL/SQLを使用して、Oracleデータベースでデータを操作する方法を説明します。基礎を形成する概念から高度なプログラミング技法まで、コード例を使用して解説しています。

この章の項目は、次のとおりです。

- 対象読者
- ドキュメントのアクセシビリティについて
- 関連ドキュメント
- 表記規則

対象読者

『*Pro*C/C++プログラマーズ・ガイド*』は、プログラマ、システム・アナリスト、プロジェクト・マネージャの他、次の作業を担当する、あるいはその習得に関心を持つOracleユーザーを対象としています。

- Oracle環境でのソフトウェア・アプリケーションの設計と開発
- Oracle環境での実行を目的とした既存のソフトウェア・アプリケーションの変換
- ソフトウェア・アプリケーション開発の管理

このマニュアルを使用するには、CおよびC++によるアプリケーション・プログラミングに関する作業上の知識を持ち、Structured Query Language(SQL)を十分に理解している必要があります。

ドキュメントのアクセシビリティについて

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、次のOracleリソースを参照してください。

- 『*Oracle Database SQL言語リファレンス*』
- 『*Oracle C++ Call Interfaceプログラマーズ・ガイド*』
- 『*Oracle Call Interfaceプログラマーズ・ガイド*』

このマニュアルに含まれる例の多くでは、Oracleのインストール時にデフォルトでインストールされるシード・データベースのサンプル・スキーマを使用しています。これらのスキーマがどのように作成されているか、およびその使用方法については、『*Oracle Databaseサンプル・スキーマ*』を参照してください。

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	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。

『Pro*C/C++プログラマーズ・ガイド』のこのリリースでの変更点

ここでは、『*Pro*C/C++プログラマーズ・ガイド*』の変更点を示します。

Pro*C/C++リリース18cバージョン18.1での変更点

新機能

このリリースの新機能は次のとおりです。

- Traffic DirectorモードのOracle Connection Managerのサポート

  Traffic DirectorモードのOracle Connection Managerは、改善された高可用性、接続多重化およびロード・バランシングのために、サポートされているデータベース・クライアントとデータベース・インスタンスの間に配置されるプロキシです。

  詳細は、Traffic DirectorモードのOracle Connection Managerの使用についてを参照してください。

Pro*C/C++ 12cリリース2 (12.2)での変更点

新機能

このリリースの新機能は次のとおりです。

- Pro*C/C++で128バイトの識別子長をサポートするようになりました。以前のリリースでは、識別子長の制限は30バイトでした。
- Pro*C/C++でOracle Instant Client - Basic Lightバージョンをサポートするようになりました。
- Pro*C/C++では新しいコマンドライン・オプションの"trim_password"が導入され、パスワード文字列に後続の空白が含まれることによる認証の問題を回避するようになりました。
- Transaction Guardなどの高可用性機能がOCI層に実装されました。詳細は、『Call Interfaceプログラマーズ・ガイド』を参照してください。

第I部 概要および概念

第I部は、次の章で構成されています。

- 概要
- プリコンパイラの概要
- データベースの概要
- データ型とホスト変数
- 高度なトピック
- 埋込みSQL
- 埋込みPL/SQL
- ホスト配列
- ランタイム・エラーの処理
- プリコンパイラのオプション
- マルチスレッド・アプリケーション

1 概要

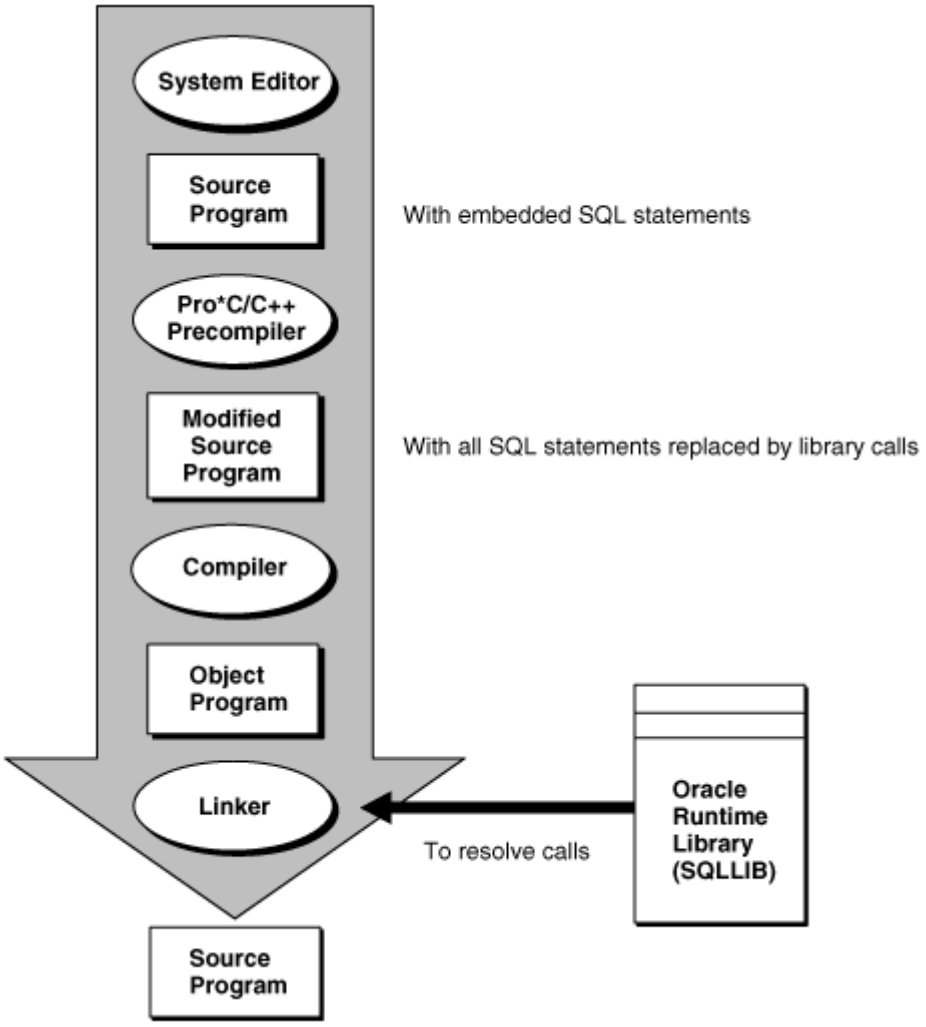
この章では、Oracle Pro*C/C++プリコンパイラについて説明します。Oracleデータを操作するアプリケーション・プログラムを開発するうえでのその役割と、このプリコンパイラによりアプリケーションで実行できる処理について説明します。この章のトピックは、次のとおりです:

- Oracleプリコンパイラ
- Oracle Pro*C/C++プリコンパイラを使用する理由
- SQLを使用する理由
- PL/SQLを使用する理由
- Pro*C/C++プリコンパイラの利点
- よくある質問(FAQ)

1.1 Oracleプリコンパイラ

Oracleプリコンパイラとは、高水準ソース・プログラムで埋込みSQL文を使用可能にするプログラミング・ツールです。図1-1のように、プリコンパイラはソース・プログラムを入力として受け入れ、埋込みSQL文を標準Oracleランタイム・ライブラリ・コールに変換して、通常の方法でコンパイル、リンクおよび実行できる変更済ソース・プログラムを生成します。

図1-1 埋込みSQLプログラムの開発



1.2 Oracle Pro*C/C++プリコンパイラを使用する理由

Oracle Pro*C/C++プリコンパイラを使用すると、アプリケーション・プログラムに強力かつ柔軟なSQLを使用できます。便利で使用しやすいインタフェースにより、アプリケーションからOracleに直接アクセスできます。

多くのアプリケーション開発ツールとは異なり、Pro*C/C++では、アプリケーションを高度にカスタマイズできます。たとえば、最新のウィンドウ機能およびマウス技術を取り込んだユーザー・インタフェースを作成できます。ユーザーとの対話なしに、バックグラウンドで実行するアプリケーションも作成できます。

さらに、Pro*C/C++はアプリケーションの微調整に役立ちます。リソースの使用状況、SQL文の実行状況および各種のランタイム・インジケータを綿密に監視できます。この情報に基づいて、最大のパフォーマンスが得られるようにプログラム・パラメータを変更できます。

プリコンパイルを行うと、アプリケーションの開発プロセスの工程が増えますが、時間の節約になります。Pro*C/C++プリコンパイラにより、埋込みSQL文はOracleランタイム・ライブラリ(SQLLIB)のコールに自動的に変換されます。また、Pro*C/C++によりホスト変数の分析、構造体から列へのマッピングの定義が行われ、SQLCHECK=FULLに設定することで埋込みSQL文の意味分析が実行されます。

1.3 SQLを使用する理由

Oracleデータにアクセスして操作するには、SQLが必要です。SQLをSQL*Plusで対話形式で使用するか、アプリケーション・プログラムに埋め込むかは、実行する作業によって決まります。ジョブにCまたはC++のプロシージャ型処理機能が必要な場合や、ジョブを定期的に実行する場合は、埋込みSQLを使用してください。

SQLは、その柔軟で強力な特性、および習得が容易であることから、最もすぐれたデータベース言語となりました。SQLは非プロシージャ言語であるため、目的とする処理を指定するときにその方法を指定する必要がありません。英文に似た少数の文を使用して、Oracleデータを一度に1行ずつまたは複数行ずつ容易に操作できます。

任意(SQL*Plus以外)のSQL文をアプリケーション・プログラムから実行できます。たとえば、次のようなSQL文です。

- データベースの表の動的なCREATE、ALTERおよびDROP
- データ行のSELECT、INSERT、UPDATEおよびDELETE
- トランザクションのCOMMITまたはROLLBACK

アプリケーション・プログラムにSQL文を埋め込む前に、SQL*Plusを使用して対話式にSQL文をテストできます。通常は、わずかな変更によって対話型SQLから埋込みSQLに切り替えることができます。

1.4 PL/SQLを使用する理由

SQLを拡張したPL/SQLは、プロシージャ構造、変数宣言および強力なエラー処理をサポートするトランザクション処理言語です。同一PL/SQLブロック内で、SQLおよびPL/SQLの拡張機能のすべてを使用できます。

埋込みPL/SQLの主な利点はパフォーマンスの向上です。SQLとは異なり、PL/SQLでは、SQL文を論理的にグループ化し、1文ずつではなくブロック単位でOracleに送ることができます。これにより、ネットワークの通信量と処理のオーバーヘッドが減少します。

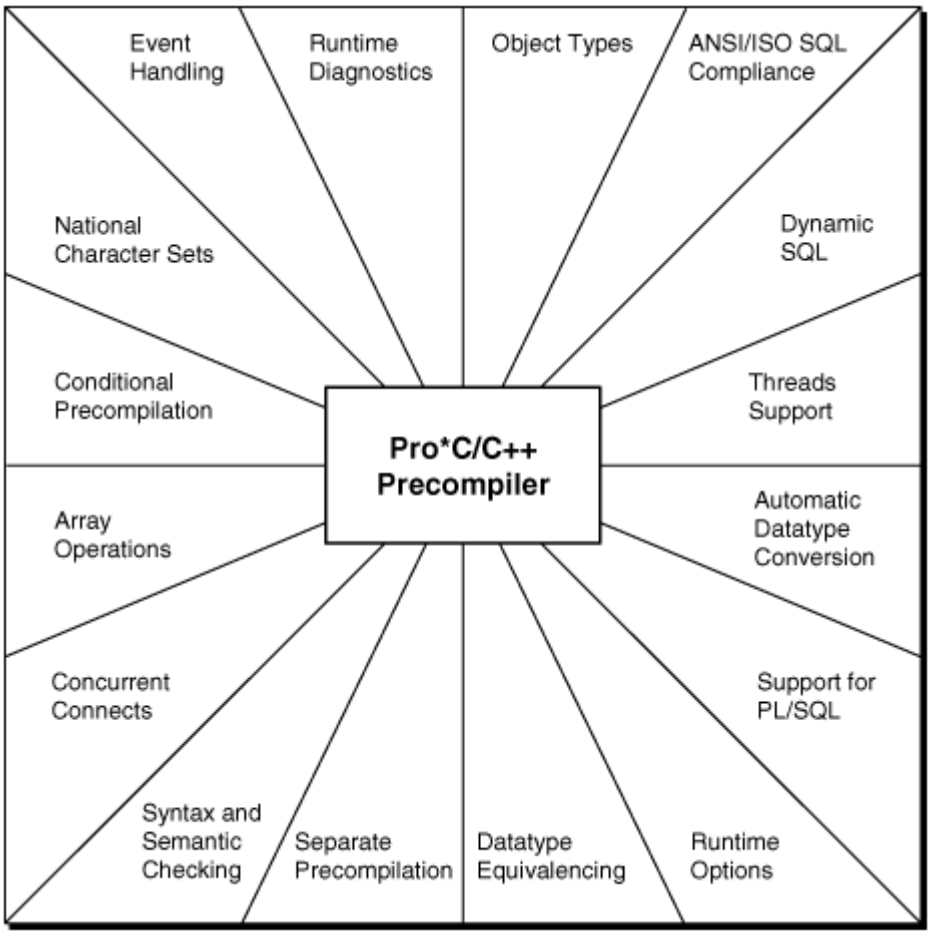
関連項目

- 埋込みPL/SQL

1.5 Pro*C/C++プリコンパイラの利点

図1-2のように、Pro*C/C++には多くの機能と利点があり、効果的で信頼性の高いアプリケーションの開発に役立ちます。

図1-2 機能と利点



Pro*C/C++によって次のことが可能となります。

- CまたはC++言語でアプリケーションを作成すること。
- ANSI/ISO規格に従って高水準言語にSQL文を埋め込むこと。
- 動的SQLを利用すること。動的SQLとは、プログラム実行時に適切なSQL文を受け入れるか作成する、高度なプログラミング技法です。
- 高度にカスタマイズしたアプリケーションを設計および開発すること。
- 共有サーバー・プロセス・アプリケーションを作成すること。
- Oracleの内部データ型と高水準言語のデータ型の間で自動的な変換を実行すること。
- アプリケーション・プログラムにPL/SQLトランザクション処理ブロックを埋め込むことで、パフォーマンスを向上すること。
- 役立つプリコンパイラ・オプションをインラインまたはコマンドラインで指定し、その値をプリコンパイル中に変更すること。
- データ型の同値化を使用して、Oracleで入力データを解釈し出力データをフォーマットする方法を制御すること。
- 複数のプログラム・モジュールを別々にプリコンパイルし、それらをリンクして1つの実行プログラムにすること。
- 埋め込まれたSQLデータ操作文とPL/SQLブロックの構文と意味を全面的にチェックすること。
- Oracle Netを使用して、複数のノード上のOracleデータベースに同時にアクセスすること。
- 配列を入力プログラム変数および出力プログラム変数として使用すること。

- 様々な環境で実行できるように、ホスト・プログラムのコード・セクションを条件付きでプリコンパイルすること。
- 高水準言語で作成されたユーザー・イグジットを使用した、SQL*Formsとの直接インタフェース。
- SQL通信領域(SQLCA)およびWHENEVER文またはDO文を使用して、エラーおよび警告を処理すること。
- Oracle通信領域(ORACA)により提供される強力な診断機能を使用すること。
- データベース内でユーザー定義のオブジェクト型を処理すること。
- データベースでコレクション(VARRAYおよびネストした表)を使用すること。
- データベースでラージ・オブジェクト(LOB)を使用すること。
- データベースに格納された各国語キャラクタ・セット・データを使用すること。
- プログラム内でOracle Call Interface(OCI)関数を使用すること。
- マルチスレッド・アプリケーションを使用すること。
- Microsoft Visual Studio .NET 2002/2003のサポート。

このように、Pro*C/C++は、充実した埋込みSQLプログラム技法をサポートする多機能ツールです。

注意:

Pro*C/C++では 16 ビット・コードの生成はサポートされません。

1.6 ディレクトリ構造

Oracleソフトウェアをインストールすると、ハード・ドライブ上にOracle製品のディレクトリ構造が作成されます。メインのOracleディレクトリには、Pro*C/C++の実行に必要なOracleサブディレクトリおよびファイルが含まれます。

Pro*C/C++をインストールすると、Oracle Universal Installerにより*ORACLE_BASE*¥*ORACLE_HOME*ディレクトリに¥precompというディレクトリが作成されます。このサブディレクトリには、表1-1に示したPro*C/C++の実行可能ファイル、ライブラリ・ファイルおよびサンプル・プログラム・ファイルが含まれています。

表1-1 precompディレクトリ構造

ディレクトリ名	目次
¥admin	構成ファイル
¥demo¥proc	Pro*C/C++のサンプル・プログラム
¥demo¥sql	サンプル・プログラムの SQL スクリプト
¥doc¥proc	Pro*C/C++の readme ファイル
¥lib¥msvc	Pro*C/C++のライブラリ・ファイル

ディレクトリ名	目次
¥mesg	メッセージ・ファイル
¥public	ヘッダー・ファイル

注意:

¥precomp ディレクトリには、その他の製品(Pro*COBOL など)用のファイルが含まれている可能性があります。

1.6.1 既知の問題、制限事項および対処方法

すべてのWindowsオペレーティング・システムでは、ファイル名やディレクトリ名での空白の使用が認められますが、Oracle Pro*C/C++およびOracle Pro*COBOLプリコンパイラでは、ファイル名またはディレクトリ名に空白が含まれているファイルはプリコンパイルされません。たとえば、次のような書式は使用しないでください。

- proc iname=test one.pc
- proc iname=d:¥dir1¥second dir¥sample1.pc

1.7 ライブラリ・ファイル

Pro*C/C++アプリケーションをリンクするとき、ライブラリ・ファイルを使用します。Pro*C/C++ライブラリ・ファイルは、次のようにインストールされます。

```
ORACLE_HOME¥precomp¥LIB¥orasql12.lib

ORACLE_HOME¥precomp¥LIB¥ottclasses.zip

ORACLE_HOME¥precomp¥LIB¥msvc¥orasqx12.lib
```

Pro*C/C++アプリケーション・プログラム・インタフェース(API)のコールは、使用するPro*C/C++ソフトウェアで提供されているDLLファイルに実装されます。DLLを使用するには、アプリケーションを、Pro*C/C++ DLLに対応するインポート・ライブラリ(.libファイル)とリンクする必要があります。また、DLLファイルが、Pro*C/C++アプリケーションを実行しているコンピュータ上にインストールされていることを確認する必要があります。

Microsoftでは、3つのライブラリ(libc.lib、libcmt.libおよびmsvcrt.lib)を提供しています。Oracle DLLは、msvcrt.libランタイム・ライブラリを使用します。アプリケーションを、他の2つのMicrosoftライブラリではなく、msvcrt.libとリンクする必要があります。

1.8 よくある質問(FAQ)

この項では、Pro*C/C++およびPro*C/C++に関連するOracleについての一般的な質問をいくつか示します。回答はこのマニュアルの他の部分に比べると簡単なものですが、該当項目の参照先がわかります。

1.8.1 VARCHARについて説明してください。

次の表は、VARCHARの簡単な説明を示しています。

VARCHAR	説明
VARCHAR2	データベースの列の一種で、可変長文字データが含まれています。列型として使用できるため、Oracle ではこれを「内部データ型」と呼びます。
VARCHAR	Oracle の「外部データ型」(データ型コード 9)です。このデータ型を使用するのは、動的 SQL 方法 4 またはデータ型同値化を使用する場合のみです。
VARCHAR[n] varchar[n]	これは Pro*C/C++プログラムでホスト変数として宣言できる Pro*C/C++の疑似型です。実際には、Pro*C/C++では、2 バイト長の要素と[n]バイトの文字配列からなる構造体として生成されます。

関連項目

- データ型とホスト変数
- ANSI動的SQL
- Oracle動的SQL: 方法4

1.8.2 Pro*C/C++はOracle Call Interfaceのコールを生成しますか。

生成しません。Pro*C/C++ではデータ構造体とSQLLIBランタイム・ライブラリ・コールが生成されます。

1.8.3 Pro*C/C++を使用せず、SQLLIBコールを使用してコーディングできますか。

SQLLIBはドキュメントが外部公開されておらず、サポートされておらず、さらにリリースごとに変更される可能性があります。一方、Pro*C/C++はANSI/ISOに準拠した製品であり、埋込みSQLの標準要件に従っています。

SQLLIBはAPIではありません。ユーザーがコール可能な関数も含まれていますが、主として言語のプリコンパイラ・パッケージ用のランタイム・ライブラリです。

データベース用のAPIコーディングが必要な場合は、Oracle Call InterfaceまたはOracle RDBMS用のクライアント側APIを使用するか、OCIとPro*C/C++を併用してください。

関連項目

- OCIリリース8のSQLLIB拡張相互運用性

1.8.4 PL/SQLのストアド・プロシージャをPro*C/C++プログラムからコールできますか。

コールできます。埋込みPL/SQLを参照してください。ストアドPL/SQLまたはJavaサブプログラムのコールについてに、デモ・プログラムがあります。

1.8.5 C++のコードを作成し、Pro*C/C++を使用してプリコンパイルできますか。

はい。C++アプリケーションを参照してください。

1.8.6 SQL文の任意の場所でバインド変数を使用できますか。

たとえば、実行時にSQL文に表の名前を入力できるようにするとします。しかし、ホスト変数を使用すると、プリコンパイラのエラーが発生します。

通常、SQL文またはPL/SQL文中の式を使用できるところであれば、任意の位置にホスト変数を使用できます。

ただし、次のSQL文は無効です(*table_name*はホスト変数です)。

```
EXEC SQL SELECT ename,sal INTO :name, :salary FROM :table_name;
```

問題を解決するには、動的SQLを使用する必要があります。これを実行するために応用できるデモ・プログラムとしてサンプル・プログラム: 動的SQL方法1があります。

関連項目

- ホスト変数の参照
- Oracle動的SQL

1.8.7 Pro*C/C++の文字処理がよくわかりません。

多数のオプションがありますが、簡単に説明します。第1に、従来のプリコンパイラおよびOracle7との互換性が必要な場合は、VARCHAR[n]ホスト変数を使用するのが最も安全な方法です。

Pro*C/C++では、他のすべての文字変数のデフォルト・データ型はCHARZです。つまり、入力時には文字列をヌル文字で終了する必要があります。出力時には空白文字で埋められ、ヌル文字で終了されます。

リリース8.0では、文字変数のデフォルト・マッピングを指定できるように、CHAR_MAPプリコンパイラ・オプションが追加されています。

アプリケーションでVARCHARもCHARZも不適切であり、全面的にC言語と同様の(ヌル文字で終了するが空白文字で埋められない)動作が必要な場合は、TYPEコマンドとC言語のtypedef文を使用し、データ型の同値化を使用して文字ホスト変数を文字列に変換してください。TYPEコマンドの使用方法を示すサンプル・プログラムについては、サンプル・プログラム: sqlvcp()の使用を参照してください。

関連項目

- VARCHAR変数の宣言
- CHARZ
- CHAR_MAPプリコンパイラ・オプション
- ユーザー定義型同値化

1.8.8 文字ポインタについて何か特別なことはありますか。

はい。Pro*C/C++では、入力ホスト変数または出力ホスト変数をバインドするときに、その長さを認識する必要があります。VARCHAR[n]を使用するか、char[n]型のホスト変数を宣言すると、Pro*C/C++では宣言から長さが認識されます。ただし、プログラム内で文字ポインタをホスト変数として使用し、malloc()を使用してバッファを定義すると、Pro*C/C++では長さが認識されません。

出力時には、バッファを割り当てるだけでなく、ヌル文字でない文字を埋め込んでからヌル文字で終了する必要があります。入力時または出力時、Pro*C/C++は長さを取得するために、バッファについてstrlen()をコールします。

関連項目

- ポインタ変数

1.8.9 Pro*C/C++でSPOOLが動作しない理由を説明してください。

SPOOLはSQL*Plusで使用される特殊なコマンドです。埋込みSQLコマンドではありません。

関連項目

- 埋込みSQLプログラミングの基本概念

1.8.10 サンプル・プログラムのオンライン版はどこにありますか。

Oracleのインストール時には、demoディレクトリが作成されます。このディレクトリがないか、あってもサンプル・プログラムが含まれていない場合は、システム管理者またはデータベース管理者に連絡してください。

1.8.11 アプリケーションをコンパイルしてリンクする方法を教えてください。

コンパイルとリンクの方法は、プラットフォームごとに異なります。Pro*C/C++アプリケーションをリンクする方法については、システム固有のOracleマニュアルを参照してください。UNIXシステムでは、demoディレクトリにdemo_proc.mkというMakeファイルがあります。たとえば、デモ・プログラムsample1.pcをリンクするには、次のコマンドラインを入力します。

```
make -f demo_proc.mk sample1
```

特殊なプリコンパイラ・オプションを使用する必要がある場合は、Pro*C/C++を別に実行してからmakeを実行します。または、独自のカスタムMakeファイルを作成することもできます。たとえば、プログラムに埋込みPL/SQLコードが含まれている場合は、次のように入力します。

```
proc cv_demo userid=username/password sqlcheck=semantics
make -f demo_proc.mk build OBJS=sample1.o EXE=sample1
```

VMSシステムには、Pro*C/C++アプリケーションをリンクできるように、LNPROCというスクリプトが用意されています。

1.8.12 Pro*C/C++では構造体をホスト変数として使用できますか。

配列インタフェースとともに使用した場合の動作について教えてください。

1つの構造体の内部で複数の配列を使用できます。また、構造体の配列を配列インタフェースとともに使用できます。

関連項目

- ホスト構造体
- ポインタ変数

1.8.13 再帰関数内で埋込みSQLを使用した場合、その再帰関数をPro*C/C++で使用できますか。

はい。ただし、埋込みSQLの場合はカーソル変数を使用する必要があります。

1.8.14 Pro*C/C++のすべてのリリースを、Oracleサーバーのすべてのバージョンで使用できますか。

データベース・サーバーに対してプリコンパイラまたはOCIアプリケーションを実行する場合、Oracleでは、クライアント・ソフトウェア・リリース以降のリリースのデータベース・サーバー・ソフトウェアを使用することをお勧めしますが、この構成が厳密に必要なわけではありません。たとえば、Oracle Databaseクライアント・ソフトウェアがリリース8.1.7である場合、サーバーに対してクライアント上でプリコンパイラ・アプリケーションを実行するには、リリース8.1.7以降のOracle Databaseサーバー・ソフトウェアを使用す

ることをお薦めします。

アプリケーションのアップグレードの詳細は、『*Oracle Databaseアップグレード・ガイド*』を参照してください。

1.8.15 アプリケーションを実行すると、必ずORA-01405エラー(フェッチした列の値がNULLです)が発生します。

標識変数が結合されていないホスト変数にNULLを入れています。これはANSI/ISO規格に準拠していないため、Oracle7からは変更されました。

可能であれば、標識変数を使用してプログラムを書きなおし、今後の開発にはインジケータを使用します。

または、MODE=ORACLEとDBMS=V7またはV8を指定してプリコンパイルしている場合は、コマンドラインでUNSAFE_NULL=YESを指定し、ORA-01405メッセージを無効にしてください。

関連項目

- 標識変数
- UNSAFE_NULL

1.8.16 すべてのSQLLIB関数はプライベート関数ですか。

いいえ。各自のプログラムまたはそのデータに関する情報を取得するためにコールできるSQLLIB関数がいくつかあります。SQLLIBパブリック関数は、次のとおりです。

SQLLIBパブリック関数	説明
SQLSQLDAAlloc()	動的 SQL 方法 4 で SQL 記述配列(SQLDA)を割り当てるために使用します。SQLDA の参照方法を参照してください。
SQLCDAFromResultSetCursor()	Pro*C/C++のカーソル変数を OCI カーソル・データ領域に変換するために使用します。SQLLIB パブリック関数の新しい名前を参照してください。
SQLSQLDAFree()	*SQLSQLDAAlloc()* を使用して割り当てた SQLDA を解放するために使用します。SQLLIB パブリック関数の新しい名前を参照してください。
SQLCDAToResultSetCursor()	OCI のカーソル・データ領域を Pro*C/C++のカーソル変数に変換するために使用します。SQLLIB パブリック関数の新しい名前を参照してください。
SQLErrorGetText()	長いエラー・メッセージを戻します。sqlerrm を参照してください。
SQLStmtGetText()	最後に実行された SQL 文のテキストを戻すために使用します。SQL 文のテキスト取得についてを参照してください。
SQLLDAGetNamed()	Pro*C/C++プログラムで OCI コールを使用するときに、指定された接続に有効なログイン・データ領域を取得するために使用します。SQLLIB パブリック関数の新しい名前を参照してください。

SQLLIBパブリック関数	説明
SQLLDAGetCurrent()	Pro*C/C++プログラムで OCI コールを使用するときに、最後の接続に有効なログイン・データ領域を取得するために使用します。SQLLIB パブリック関数の新しい名前を参照してください。
SQLColumnNullCheck()	動的 SQL 方法 4 の NULL 状態の表示を戻します。NULL/Not NULL データ型の処理を参照してください。
SQLNumberPrecV6()	数値の精度と位取りを戻します。精度と位取りの抽出を参照してください。
SQLNumberPrecV7()	*SQLNumberPrecV6()*の変形です。精度と位取りの抽出を参照してください。
SQLVarcharGetLength()	VARCHAR[n]の埋め込んだサイズを取得するために使用します。VARCHAR 配列コンポーネントの長さを調べる方法を参照してください。
SQLEnvGet()	特定の SQLLIB ランタイム・コンテキストの OCI 環境ハンドルを戻します。SQLEnvGet()を参照してください。
SQLSvcCtxGet()	データベース接続の OCI サービス・コンテキストを戻します。「SQLSvcCtxGet()」を参照してください。
SQLRowidGet()	最後に挿入された行のユニバーサル ROWID を戻します。SQLRowidGet()を参照してください。
SQLExtProcError()	外部 C プロシージャでエラーが発生した場合に、PL/SQL に制御を戻します。SQLExtProcError()を参照してください。

このリストの関数は、スレッド・セーフなSQLLIBパブリック関数です。すべての新規アプリケーションで、これらの関数を使用します。これらのスレッド・セーフなパブリック関数(およびその旧称)の詳細は、SQLLIBパブリック関数の新しい名前を参照してください。

1.8.17 新しいオブジェクト型はOracleでどのようにサポートされていますか。

Pro*C/C++アプリケーションでのオブジェクト型の使用方法は、オブジェクトおよびObject Type Translator (OTT)を参照してください。

1.8.18 互換性、アップグレードおよび移行

Pro*C/C++では、OCIベースのアプリケーションと同様の互換性規則を採用します。この場合、下位互換性に対するOCIの制限と同じ制限が適用されます。

追加の配列のINSERTおよびSELECT構文は、DB2プリコンパイラ・アプリケーションのPro*C/C++アプリケーションへの移行に役立ちます。これは、DB2の配列のINSERTおよびSELECT構文をOracle Pro*C/C++の構文に変更する必要がないためです。

Pro*C/C++でサポートされている「暗黙的バッファ挿入」機能は、パフォーマンス向上のためにPro*C/C++の配列構文を使

用せずに、DB2プリコンパイラ・アプリケーションをPro*C/C++アプリケーションに移行する場合に役立ちます。

2 プリコンパイラの概要

この章では埋込みSQLプログラムの動作を説明します。それらが動作する特殊な環境と、その環境がアプリケーション設計に及ぼす影響について調べます。埋込みSQLプログラミングの基本概念とアプリケーション開発の手順について説明した後、簡単なプログラムを使用して、要点を具体的に説明します。

この章のトピックは、次のとおりです:

- 埋込みSQLプログラミングの基本概念
- 埋込みSQLアプリケーションの開発ステップ
- プログラミングのガイドライン
- サンプル表
- サンプル・プログラム: 単純な問合せ
- サンプル・プログラム: SQL99構文を使用する単純な問合せ

2.1 埋込みSQLプログラミングの基本概念

この項では、後に続く章の内容の基本概念について説明します。この項の内容は次のとおりです。

- 埋込みSQL文
- 埋込みSQLの構文
- 静的SQL文と動的SQL文
- 埋込みPL/SQLブロック
- ホスト変数および標識変数
- Oracleのデータ型
- 配列
- データ型の同値化
- プライベートSQL領域、カーソルおよびアクティブ・セット
- トランザクション
- エラーおよび警告
- SQL99構文サポート

2.1.1 埋込みSQL文

埋込み*SQL*とは、アプリケーション・プログラムに記述されているSQL文のことです。SQL文を含むアプリケーション・プログラムはホスト・プログラムと呼ばれ、その記述言語はホスト言語と呼ばれます。たとえば、Pro*C/C++では、特定のSQL文をCまたはC++ホスト・プログラムに埋め込むことができます。

Oracleデータの操作および問合せには、INSERT文、UPDATE文、DELETE文およびSELECT文を使用します。INSERTではデータの行をデータベース表に追加し、UPDATEでは行を変更し、DELETEでは不要な行を削除し、SELECTでは検索条件と一致する行を取り出します。

強力なSET ROLE文を使用すると、データベース権限を動的に管理できます。ロールとは、関連するシステム権限やオブジェク

ト権限の名前付きグループ、あるいはユーザーまたは他のロールに付与された関連するシステム権限やオブジェクト権限の名前付きグループです。ロールの定義は、Oracleデータ・ディクショナリに格納されます。アプリケーションでは、必要に応じてSET ROLE文を使用し、ロールを有効または無効にできます。

アプリケーション・プログラムではSQL文のみ有効であり、SQL*Plus文は無効です。(SQL*Plusにはレポートの書式化、SQL文の編集、環境パラメータの設定のための文が追加されています。)

2.1.1.1 実行文とディレクティブ

埋込みSQL文には、すべての対話型SQL文に加えて、Oracleとホスト・プログラムの間でデータを転送できるその他の文があります。埋込みSQL文には、実行文およびディレクティブという2つのタイプがあります。実行文では、SQLLIBランタイム・ライブラリへのコールが発生します。実行文は、Oracleへの接続、Oracleデータの定義、問合せ、操作、Oracleデータへのアクセス制御およびトランザクション処理に使用します。CまたはC++言語の実行文を配置できる位置であれば、任意の位置に記述できます。

一方、ディレクティブではSQLLIBへのコールは発生せず、Oracleデータの操作も行われません。宣言文は、Oracleオブジェクト、通信領域およびSQL変数を宣言するために使用します。CまたはC++の変数宣言を配置できる位置であれば、任意の位置に記述できます。

表2-1は、各種の埋込みSQL文の一部をグループ化したものです。

表2-1 埋込みSQL文

デイレクティブ	**用途**
ARRAYLEN*	PL/SQL でのホスト配列の使用
BEGIN DECLARE SECTION* END DECLARE SECTION*	ホスト変数の宣言(オプション)
DECLARE*	Oracle スキーマ・オブジェクトの命名
INCLUDE*	ファイルへのコピー
TYPE*	データ型の同値化
VAR*	変数の同値化
WHENEVER*	ランタイム・エラーの処理

表2-2 **埋込み*SQL*文**

実行文	**用途**
ALLOCATE*	Oracle データの定義および制御
ALTER	-

実行文	用途
ANALYZE	-
DELETE	DML
INSERT	-
SELECT	-
UPDATE	-
COMMIT	トランザクションの処理
ROLLBACK	-
SAVEPOINT	-
SET TRANSACTION	-
DESCRIBE*	動的 SQL の使用
EXECUTE*	-
PREPARE*	-
ALTER SESSION	セッションの制御
SET ROLE	-

*対話形式はありません。

2.1.2 埋込みSQL構文

作成したアプリケーション・プログラムでは、完全なSQL文と完全なCプログラムを自由に混在させ、SQL文にC言語の変数や構造体を使用できます。SQL文をホスト・プログラム内に作成する場合の唯一の特殊要件は、SQL文をEXEC SQLキーワードで開始し、セミコロンで終了することです。Pro*C/C++では、すべてのEXEC SQL文がSQLLIBランタイム・ライブラリへのコールに変換されます。

多くの埋込みSQL文は、新しい句が追加される点、またはプログラム変数が使用される点のみが同等の対話型SQLと異なります。次の例は、対話型ROLLBACK文と埋込みROLLBACK文の対比を示しています。

```
ROLLBACK WORK:           -- interactive
EXEC SQL ROLLBACK WORK;  -- embedded
```

この2つの文の効果は同じですが、対話型SQL環境(SQL*Plusの実行時など)では前者を、Pro*C/C++プログラムでは後者を使用します。

2.1.3 静的SQL文と動的SQL文

ほとんどのアプリケーション・プログラムは、静的SQL文と固定的なトランザクションを処理するように設計されています。この場合、実行前にそれぞれのSQL文およびトランザクションの構成が認識できます。つまり、どのSQLコマンドが発行され、どのデータベースの表が変更され、どの列が更新されるかなどが事前にわかっています。

ただし、アプリケーションによっては、任意の有効なSQL文を実行時に受け入れて処理することを要求される場合もあります。したがって、関係するSQLコマンド、データベース表および列が実行時までわからないことがあります。

動的*SQL*は、実行時のプログラムでSQL文を受け入れるか作成し、データ型の変換を明示的に制御する高度なプログラミング技術です。

2.1.4 埋込みPL/SQLブロック

Pro*C/C++では、PL/SQLブロックが1つの埋込みSQL文と同様に扱われます。PL/SQLブロックは、アプリケーション・プログラム内でSQL文を記述できる位置であれば、任意の位置に記述できます。PL/SQLをホスト・プログラムに埋め込むには、単にPL/SQLと共有する変数を宣言し、PL/SQLブロックをEXEC SQL EXECUTEおよびEND-EXECキーワードで囲みます。

PL/SQLはすべてのSQLデータ操作コマンドおよびトランザクション処理コマンドをサポートしているため、埋込みPL/SQLブロックからOracleデータを柔軟かつ安全に操作できます。

2.1.5 ホスト変数および標識変数

ホスト変数は、Oracleとプログラムとの間で通信を行うための鍵です。ホスト変数とは、C言語で宣言され、Oracleで共有される(つまり、プログラムとOracleの両方がその値を参照できる)スカラー変数または集合体変数です。

プログラムでは、入力ホスト変数を使用してOracleにデータを渡します。Oracleでは、出力ホスト変数を使用してプログラムにデータおよびステータス情報を渡します。プログラムは入力ホスト変数に値を割り当て、Oracleは出力ホスト変数に値を割り当てます。

ホスト変数は、SQL式を使用できる位置であれば任意の位置に使用できます。SQL文では、SQLキーワードと区別するために、ホスト変数に接頭辞としてコロン(:)を付ける必要があります。

C構造体を使用して、複数のホスト変数を含めることもできます。接頭辞コロンを付けて埋込みSQL文の構造体を命名すると、Oracleでは構造体の各コンポーネントがホスト変数として使用されます。

任意のホスト変数に任意指定の標識変数を関連付けることができます。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すshort int型変数です。標識変数は、入力ホスト変数へのNULLの割当てと、出力ホスト変数に含まれるNULLまたは切捨て値の検出に使用されます。*NULL*値は、欠落している値、不明な値または適用不能な値です。

SQL文の場合、標識変数には接頭辞コロンを付けて、対応するホスト変数の直後に記述する必要があります。さらにわかりやすくするには、ホスト変数とその標識変数の間にINDICATORキーワードを記述します。

ホスト変数が構造体にパッケージされていて、標識変数を使用する場合は、ホスト構造体の各ホスト変数に対する標識変数を含む構造体を作成し、SQL文でインジケータ構造体を命名します。インジケータ構造体の名前の前にはコロンを付け、ホスト変数の構造体の直後に指定します。また、INDICATORキーワードを使用して、ホスト構造体とそれに対応するインジケータ構造体を分離することもできます。

2.1.6 Oracleのデータ型

通常、ホスト・プログラムからOracleにデータが入力され、Oracleからプログラムにデータが出力されます。Oracleではデータベース表に入力データが格納され、出力データはプログラム・ホスト変数に格納されます。データ項目を格納するために、Oracleではそのデータ型を認識する必要があり、データ型により値の記憶形式と有効範囲が指定されます。

Oracleでは、内部データ型と外部データ型という2種類のデータ型が認識されます。内部データ型は、Oracleでデータベース列にデータを格納する方法を指定します。また、Oracleでは、データベース疑似列を表す内部データ型も使用されます。データベース疑似列は特定のデータ項目を戻すものの、表には実際の列はありません。

外部データ型は、データがホスト変数にどのように格納されるかを指定します。ホスト・プログラムがOracleにデータを入力すると、Oracleは必要に応じて、入力ホスト変数の外部データ型とターゲット・データベース列の内部データ型の間で変換を行います。Oracleがホスト・プログラムにデータを出力すると、Oracleは必要に応じて、ソース・データベース列の内部データ型と出力ホスト変数の外部データ型の間で変換を行います。

2.1.7 配列

Pro*C/C++では、配列ホスト変数*(*ホスト配列と呼ばれます)および構造体の配列を定義して、1つのSQL文で操作できます。配列に対するSELECT、FETCH、DELETE、INSERTおよびUPDATE文を使用すると、大量のデータを容易に問合せおよび操作できます。また、ホスト配列をホスト変数の**構造体**の中で使用することもできます。

2.1.8 データ型の同値化

Pro*C/C++ではデータ型を同値化できるため、アプリケーションの柔軟性が向上します。つまり、Oracleが入力データを解釈し、出力データをフォーマットする方法をカスタマイズできます。

変数ごとに、サポートされているC言語のデータ型をOracleの外部データ型と同値化できます。また、ユーザー定義のデータ型をOracleの外部データ型と同値化することもできます。

2.1.9 プライベートSQL領域、カーソルおよびアクティブ・セット

Oracleでは、SQL文を処理するためにプライベート*SQL*領域と呼ばれる作業領域がオープンされます。このプライベートSQL領域にはSQL文の実行に必要な情報が格納されます。カーソルと呼ばれる識別子を使用すると、SQL文に名前を付け、そのプライベート領域内の情報にアクセスし、その処理をある程度制御できます。

静的SQL文には、明示的および暗黙的という2種類のカーソルがあります。Oracleでは、1行のみを戻すSELECT文(問合せ)などのすべてのデータ定義文とDML文に対しては、1つのカーソルが暗黙的に宣言されます。ただし、複数行を戻す問合せで2行目以降を処理する場合は、カーソルを明示的に宣言(またはホスト配列を使用)する必要があります。

戻された一連の行はアクティブ・セットと呼ばれ、そのサイズは問合せの検索条件と何行一致するかによって異なります。現在処理している行(カレント行と呼ばれる)を識別するには、明示カーソルを使用します。

たとえば、端末の画面に一連の行が戻されたとします。画面上のカーソルは、最初に処理される行、次に処理される行というように移動していきます。同様に、明示カーソルはアクティブ・セット内の現在行を指します。これを利用して、プログラムは行を1行ずつ処理できます。

2.1.10 トランザクション

トランザクションとは、論理的に関連のある一連のSQL文です(ある銀行勘定の貸方に記帳し、別の銀行勘定の借方に記帳する2つのUPDATEなど)。Oracleでは、トランザクションは1単位として扱われるため、それぞれの文による変更はすべてが同時に確定されるか、取り消されるかします。

最後のデータ定義、COMMITまたはROLLBACK文が実行された後に実行するDML文すべてが、現行のトランザクションを構成します。

データベースの整合性を維持するために、Pro*C/C++ではCOMMIT文、ROLLBACK文およびSAVEPOINT文を使用してトランザクションを定義できます。

COMMITでは、現行のトランザクション中にデータベースに加えられた変更が確定されます。ROLLBACKでは、現行のトランザクションを終了し、トランザクションの開始以降に加えられた変更がすべて取り消されます。SAVEPOINTでは、トランザクション処理の現在の位置にマークが付けられ、ROLLBACKと併用することで、トランザクションを部分的に取り消すことができます。

2.1.11 エラーおよび警告

埋込みSQL文を実行すると、エラーまたは警告が発生する場合があります。これらの結果を処理する方法が必要です。Pro*C/C++には、SQL通信領域(SQLCA)とWHENEVER文という2つのエラー処理方法が用意されています。

SQLCAは、ホスト・プログラムに組み込む(ハードコードする)データ構造体です。Oracleにより使用されるプログラム変数を定義して、ランタイム・ステータス情報をプログラムに渡します。SQLCAを使用すると、直前に試みた処理に関するOracleからのフィードバックに基づいて、異なる処理を実行できます。たとえば、DELETE文が成功したかチェックし、成功した場合は削除された行数をチェックできます。

WHENEVER文を使用すると、Oracleがエラーまたは警告状態を検出した際に自動的に行われるアクションを指定できます。これらの処理は、次の文の継続実行、関数のコール、ラベル付き文への分岐、停止です。

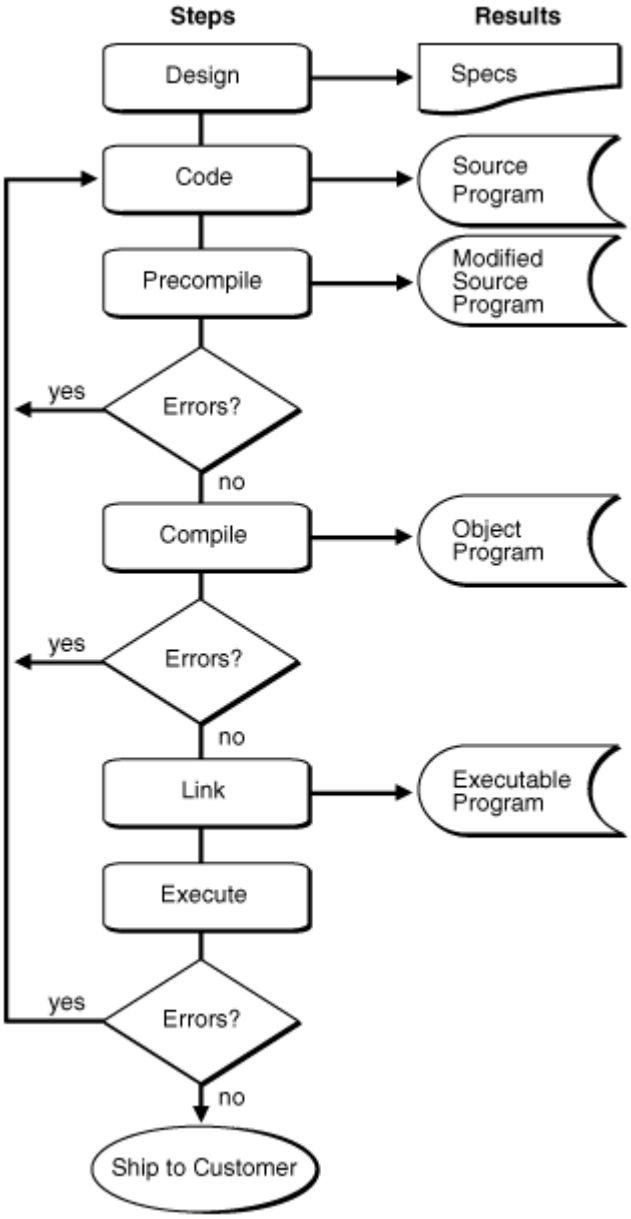
2.1.12 SQL99構文サポート

SQL規格により、規格に準拠するすべてのソフトウェア製品で、SQLアプリケーションを移植できます。Oracle機能は、ANSI/ISO SQL99規格(ANSI準拠の結合を含む)に準拠しています。Pro*C/C++では、Oracleデータベースでサポートされているすべての SQL99機能がサポートされており、SELECT文、INSERT文、DELETE文およびUPDATE文と、DECLARE CURSOR文でのカーソル本体のSQL99構文がサポートされています。

2.2 埋込みSQLアプリケーションの開発ステップ

図2-1は、埋込みSQLアプリケーションの開発プロセスを示しています。

図2-1 埋込みSQLアプリケーションの開発プロセス



図のように、プリコンパイルの結果、通常のコンパイルが可能な変更済ソース・ファイルが生成されます。従来の開発プロセスにプリコンパイル処理が追加されますが、このステップによりきわめて柔軟なアプリケーションを作成できます。

2.3 プログラミングのガイドライン

この項では、埋込みSQL構文、コーディング規則、C言語固有の機能および制限について説明します。簡単に参照できるように、項目をアルファベット順に記載してします。

2.3.1 コメント

SQL文には、空白を挿入できる位置(EXEC SQLキーワードの間以外)であれば、任意の位置にC言語の形式のコメント(/* ... */)を記述できます。また、SQL文の行末には、次の例のようにANSI形式のコメント(-- ...)も挿入できます。

```
EXEC SQL SELECT ENAME, SAL
    INTO :emp_name, :salary  -- output host variables
    FROM EMP
    WHERE DEPTNO = :dept_number;
```

CODE=CPPプリコンパイラ・オプションを使用してプリコンパイルする場合は、Pro*C/C++ソース内でC++形式のコメント(//)を使用できます。

2.3.2 定数

*L*または*l*を接尾辞として付けると、**long** int型定数に指定されます。*U*または*u*を接尾辞として付けると、**符号なし**整数定数に指定されます。*0X*または*0x*を接頭辞として付けると、16進整数定数に指定されます。*F*または*f*を接尾辞として付けると、**float**型浮動小数点定数に指定されます。これらの書式は、SQL文では使用できません。

2.3.3 宣言部

宣言部にはホスト変数宣言が含まれており、その形式は次のとおりです。

```
EXEC SQL BEGIN DECLARE SECTION;
/* Declare all host variables inside this section:  */
    char *uid = "username/password";
    ...
EXEC SQL END DECLARE SECTION;
```

宣言部は、次の文で開始します。

```
EXEC SQL BEGIN DECLARE SECTION;
```

次の文で終了します。

```
EXEC SQL END DECLARE SECTION;
```

この2つの文の間に指定できるのは、次の要素のみです。

- ホスト変数および標識変数の宣言
- 非ホストC/C++変数
- EXEC SQL DECLARE文
- EXEC SQL INCLUDE文
- EXEC SQL VAR文
- EXEC SQL TYPE文
- EXEC ORACLE文
- C/C++コメント

宣言部が必要になるのは、MODE=ANSIまたはCODE=CPP(C++アプリケーション内)、PARSE=NONEまたはPARTIALの場合です。

宣言部は複数使用できます。異なるコード・モジュールでもかまいません。

関連項目

- コードの解析について

2.3.4 デリミタ

Cでは1文字を区切る場合、一重引用符を次のように使用します。

```
ch = getchar();
switch (ch)
{
case 'U': update();  break;
case 'I': insert();  break;
...
```

SQLでは文字列を区切る場合、一重引用符を次のように使用します。

```
EXEC SQL SELECT ENAME, SAL FROM EMP WHERE JOB = 'MANAGER';
```

C言語では文字列を区切る場合、二重引用符を次のように使用します。

```
printf("¥nG'Day, mate!");
```

SQLでは特殊文字または小文字を含む識別子を区切る場合、二重引用符を次のように使用します。

```
EXEC SQL CREATE TABLE "Emp2" (empno  number(4), ...);
```

2.3.5 ファイルの長さ

Pro*C/C++で処理できるソース・ファイルの長さには制限があります。許容される行数には制限があります。ファイル・サイズの制約となる要因には、ソース・ファイルの次のような側面があります。

- 埋込みSQL文の複雑さ(バインド変数と定義変数の数など)
- データベース名の使用の有無(AT句を使用してデータベースに接続するなど)
- 埋込みSQL文の数

この制限に関連した問題を防ぐには、複数のプログラム単位を使用してソース・ファイルのサイズを小さくします。

2.3.6 関数プロトタイプ

ANSI C規格(X3.159-1989)は、関数プロトタイプを提供しています。関数プロトタイプでは、関数およびその引数のデータ型を宣言しており、Cコンパイラでは欠落している引数や一致しない引数を検出できます。

CODEオプションにより、プリコンパイラでCまたはC++コードが生成される方法が決まります。このオプションは、コマンドラインまたは構成ファイルに入力できます。

2.3.6.1 ANSI_C

CODE=ANSI_Cを指定してプログラムをプリコンパイルすると、プリコンパイラにより完全にプロトタイプ化された関数宣言が生成されます。次に例を示します。

```
extern void sqlora(long *, void *);
```

2.3.6.2 KR_C

CODE=KR_C(KRは「Kernighan and Ritchie」を表します)オプションを指定してプリコンパイルすると、関数のパラメータ・リストがコメント・アウトされていることを除き、ANSI_Cを指定した場合と同じように関数プロトタイプが生成されます。次に例を示します。

```
extern void sqlora(/*_ long *, void *  _*/);
```

したがって、ANSI Cがサポートされていないコンパイラを使用する場合は、必ずプリコンパイラ・オプションCODEをKR_Cに設定してください。CODEオプションをANSI_Cに設定すると、プリコンパイラでは**const**型の修飾子など、他のANSI固有の構造体も生成できます。

2.3.6.3 CPP

CODE=CPPを指定してコンパイルすると、C++互換の関数プロトタイプが生成されます。このオプション設定は、C++コンパイラで使用してください。

関連項目

- C++アプリケーション

2.3.7 ヒントの長さ

埋込みSQL文におけるSQLヒントの最大長は256文字に制限されます。この制限を超えるヒントはすべて切り捨てられます。

2.3.8 ホスト変数名

ホスト変数名には、大文字または小文字、数字およびアンダースコアを使用できますが、最初の文字は英字にする必要があります。長さは任意ですが、Pro*C/C++にとって重要なのは最初の31文字のみです。Cコンパイラやリンカーによっては最大長が短い場合があるため、使用するCコンパイラのユーザーズ・ガイドで確認してください。

移植性を考慮する場合は、ホスト変数名を18文字以下(SQL標準での長さの要件)に制限することもできます。

関連項目

- 予約語、キーワードおよびネームスペース

2.3.9 行の継続

SQL文は、ある行から次の行に続けることができます。文字列リテラルをある行から次の行に続ける場合は、次のようにバックスラッシュ(¥)を使用する必要があります。

```
EXEC SQL INSERT INTO dept (deptno, dname) VALUES (50, 'PURCHAS¥
ING');
```

このコンテキストでは、バックスラッシュはプリコンパイラで継続文字として扱われます。

2.3.10 行の長さ

行の最大長は、ASCII文字のみを含む行の場合は1299、マルチバイト文字の場合は324です。

2.3.11 MAXLITERALのデフォルト値

MAXLITERALプリコンパイラ・オプションを使用すると、プリコンパイラで生成される文字列リテラルの最大長を指定できます。MAXLITERALのデフォルト値は1024です。必要に応じて、より小さい値を指定してください。たとえば、Cコンパイラで513文字以上の文字列リテラルを処理できない場合は、MAXLITERAL=512と指定します。使用しているCコンパイラのユーザーズ・ガイドを参照してください。

2.3.12 演算子

論理演算子と関係演算子「equal to」は、CとSQLでは次のリストのように異なります。これらのC演算子は、SQL文では使用できません。

SQL演算子	C演算子
NOT	!
AND	&&

SQL演算子	C演算子
OR	||
=	==

次のC演算子も、SQL文では使用できません。

タイプ	C演算子
アドレス	&
ビット単位	&、|、^、~
コンパウンド代入	+=、-=、*=など
条件付き	?:
減分	--
増分	++
間接	*
モジュラス	%
シフト	>>、<<

2.3.13 文の終了記号

埋込みSQL文の終わりには、次のように常にセミコロン(;)を付けます。

```
EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
```

2.4 条件付きプリコンパイル

条件付きプリコンパイルとは、特定の条件に基づいてコード・セクションをホスト・プログラムに組み込む(または除外する)プリコンパイルの方法です。たとえば、UNIXでプリコンパイルするときにはあるコード・セクションを組み込み、VMSでプリコンパイルするときには別のコード・セクションを組み込むことができます。条件付きプリコンパイルの使用により、様々な環境で実行できるプログラムを作成できます。

環境および処理を定義する文によってコードの条件文が区切られます。これらのセクションには、CまたはC++の文とEXEC SQL文を記述できます。次の文でプリコンパイルの条件を制御します。

```
EXEC ORACLE DEFINE symbol;        -- define a symbol
EXEC ORACLE IFDEF symbol;         -- if symbol is defined
EXEC ORACLE IFNDEF symbol;        -- if symbol is not defined
```

```
EXEC ORACLE ELSE;                   -- otherwise
EXEC ORACLE ENDIF;                  -- end this control block
```

すべてのEXEC ORACLE文の終わりには、セミコロンを付ける必要があります。

2.4.1 記号の定義

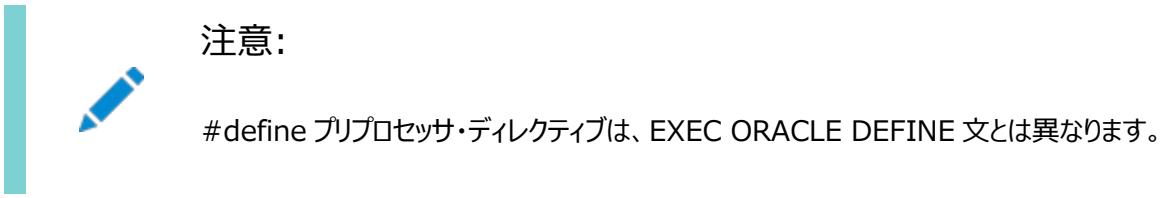
シンボルを定義するには2通りの方法があります。1つの方法では、ホスト・プログラムに次の文を含めます。

```
EXEC ORACLE DEFINE symbol;
```

もう1つの方法では、次の構文を使用してコマンドラインで記号を定義します。

```
... DEFINE=symbol ...
```

symbolには、大/小文字区別がありません。



注意:

#define プリプロセッサ・ディレクティブは、EXEC ORACLE DEFINE 文とは異なります。

Pro*C/C++をシステムにインストールするときに、ポート固有の記号がいくつか事前定義されます。たとえば、オペレーティング・システムの事前定義済記号には、CMS、MVS、MS-DOS、UNIXおよびVMSがあります。

2.4.2 SELECT文の例

次の例では、記号site2が定義されている場合にのみ、SELECT文がプリコンパイルされます。

```
EXEC ORACLE IFDEF site2;
   EXEC SQL SELECT DNAME
       INTO :dept_name
       FROM DEPT
       WHERE DEPTNO= :dept_number;
EXEC ORACLE ENDIF;
```

C、C++または埋込みSQLコードをIFDEFとENDIFの間に記述し、シンボルを**定義しない**ことで、そのコードをコメント行にすることができます。

2.5 分割プリコンパイル

複数のCまたはC++プログラム・モジュールを別々にプリコンパイルし、それらをリンクして1つの実行可能プログラムにできます。これにより、プログラムの機能コンポーネントの作成とデバッグを複数のプログラマが分担して行う場合に必要とされる、モジュラー・プログラミングが可能になります。個々のプログラム・モジュールを同じ言語で作成する必要はありません。

2.5.1 ガイドライン

次のガイドラインに従うと、いくつかの問題を回避できます。

2.5.1.1 カーソルの参照

カーソル名はSQL識別子であり、その有効範囲はプリコンパイル・ユニットです。このため、カーソルの動作が複数のプリコンパイ

ル・ユニット(ファイル)にまたがることはありません。つまり、あるファイル内で宣言したカーソルを、別のファイルからオープンまたはフェッチすることはできません。したがって、分割プリコンパイルを実行するときは、指定のカーソルに対する定義と参照がすべて1つのファイルに記述されているか確認してください。

2.5.1.2 MAXOPENCURSORSの指定

OracleにCONNECTするプログラム・モジュールをプリコンパイルする場合は、すべてのプログラム・モジュールに十分に対応できるように、MAXOPENCURSORSの値を指定してください。CONNECTしない別のプログラム・モジュールにMAXOPENCURSORSを使用しても、その値は無視されます。実行時には、CONNECTに有効な値のみが使用されます。

2.5.1.3 単一のSQLCAの使用

使用するSQLCAが1つのみの場合は、1つのプログラム・モジュール内でglobalとして宣言し、他のモジュール内ではexternalとして宣言する必要があります。*extern*記憶域クラスを使用して、コードに次の定義を追加します。

```
#define SQLCA_STORAGE_CLASS extern
```

この例は、プリコンパイラに対して、他のプログラム・モジュール内でSQLCAを検索するように指示しています。SQLCAをexternalとして宣言しないかぎり、各プログラム・モジュールでは専用のローカルSQLCAが使用されます。

注意:

アプリケーションのソース・ファイルはすべて、名前が一意である必要があります。一意でないとエラーが発生します。

2.6 コンパイルおよびリンク

実行可能プログラムを作成するには、プリコンパイラにより生成された出力である.cソース・ファイルをコンパイルし、生成されるオブジェクト・モジュールをSQLLIBおよびシステム固有のOracleライブラリに必要なモジュールとリンクさせる必要があります。プリコンパイラ・コードとOCIコールを併用している場合は、OCIランタイム・ライブラリ(UNIXシステムではliboci.a)にもリンクさせてください。

リンカーはオブジェクト・モジュール内のシンボリック参照を解決します。これらの参照で競合が発生すると、リンクは失敗します。このような失敗が起こるのは、サード・パーティ・ソフトウェアをプリコンパイル済プログラムにリンクする場合などです。すべてのサード・パーティ・ソフトウェアにOracleとの互換性があるとはかぎりません。したがって、プログラムをリンクさせて共有にすると、原因不明のエラーが発生することがあります。スタンドアロン型や2タスク型でリンクさせると、問題が解消する場合があります。

コンパイルとリンクはシステムに依存します。ほとんどのプラットフォームでは、Pro*C/C++アプリケーションのプリコンパイル、コンパイルおよびリンクに使用できるように、サンプルの*Make*ファイルまたはバッチ・ファイルが用意されています。システム固有のマニュアルを参照してください。

2.7 例の表

このマニュアルのほとんどのプログラミング例では、DEPTおよびEMPサンプル・データベース表を使用しています。これらの表の定義を次に示します。

```
CREATE TABLE DEPT
    (DEPTNO    NUMBER(2) NOT NULL,
     DNAME     VARCHAR2(14),
```

```
     LOC        VARCHAR2(13))

CREATE TABLE EMP
    (EMPNO      NUMBER(4) NOT NULL,
     ENAME      VARCHAR2(10),
     JOB        VARCHAR2(9),
     MGR        NUMBER(4),
     HIREDATE   DATE,
     SAL        NUMBER(7,2),
     COMM       NUMBER(7,2),
     DEPTNO     NUMBER(2))
```

2.7.1 サンプル・データ

DEPT表とEMP表には、それぞれ次のデータ行が含まれて

います。

```
DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

EMPNO ENAME   JOB           MGR  HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ------ --------- ------ ------ -------
 7369 SMITH   CLERK       7902 17-DEC-80    800              20
 7499 ALLEN   SALESMAN    7698 20-FEB-81   1600    300       30
 7521 WARD    SALESMAN    7698 22-FEB-81   1250    500       30
 7566 JONES   MANAGER     7839 02-APR-81   2975              20
 7654 MARTIN  SALESMAN    7698 28-SEP-81   1250   1400       30
 7698 BLAKE   MANAGER     7839 01-MAY-81   2850              30
 7782 CLARK   MANAGER     7839 09-JUN-81   2450              10
 7788 SCOTT   ANALYST     7566 19-APR-87   3000              20
 7839 KING    PRESIDENT         17-NOV-81   5000              10
 7844 TURNER  SALESMAN    7698 08-SEP-81   1500              30
 7876 ADAMS   CLERK       7788 23-MAY-87   1100              20
 7900 JAMES   CLERK       7698 03-DEC-81    950              30
 7902 FORD    ANALYST     7566 03-DEC-81   3000              20
 7934 MILLER  CLERK       7782 23-JAN-82   1300              10
```

2.8 サンプル・プログラム: 単純な問合せ

Pro*C/C++および埋込みSQLをよく理解する方法の1つは、サンプル・プログラムを学習することです。次のプログラムは、Pro*C/C++のdemoディレクトリにあるsample1.pcファイルにあり、オンラインでも使用可能です。

このプログラムはOracleに接続した後でループし、従業員番号の入力を求めるプロンプトを表示します。データベースに従業員名、給与およびコミッションを問い合せて、その情報を表示した後にループを継続します。この情報はホスト構造体に戻されます。また、SELECTで選択された出力値にNULLが含まれるかどうかを示すパラレル・インジケータ構造体もあります。

MODE=ORACLEプリコンパイラ・オプションを使用して、サンプル・プログラムをプリコンパイルします。

注意:

この機能を簡単に説明するために、この例では、デプロイされたシステムで通常使用されるパスワード管理手法を実行していません。本番環境では、Oracle Database のパスワード管理ガイドラインに従い、サンプル・アカウントを無効にしてください。パスワード管理ガイドラインおよびその他のセキュリティ上の推奨事項については、『*Oracle Database セキュリティ・ガイド*』を参照してください。

```
/*
```

* sample1.pc * * Prompts the user for an employee number, * then queries the emp table for the employee's * name, salary and commission. Uses indicator * variables (in an indicator struct) to determine * if the commission is NULL. * */ #include <stdio.h> #include <string.h> /* Define constants for VARCHAR lengths. */ #define UNAME_LEN 20 #define PWD_LEN 40 /* Declare variables. No declare section is needed if MODE=ORACLE.*/ VARCHAR username[UNAME_LEN]; /* VARCHAR is an Oracle-supplied struct */ varchar password[PWD_LEN]; /* varchar can be in lower case also. */ /* Define a host structure for the output values of a SELECT statement. */ struct { VARCHAR emp_name[UNAME_LEN]; float salary; float commission; } emprec; /* Define an indicator struct to correspond to the host output struct. */ struct { short emp_name_ind; short sal_ind; short comm_ind; } emprec_ind; /* Input host variable. */ int emp_number; int total_queried; /* Include the SQL Communications Area. You can use #include or EXEC SQL INCLUDE. */ #include <sqlca.h> /* Declare error handling function. */ void sql_error(); main() { char temp_char[32]; /* Connect to ORACLE-- * Copy the username into the VARCHAR. */ strncpy((char *) username.arr, "SCOTT", UNAME_LEN); /* Set the length component of the VARCHAR. */ username.len = strlen((char *) username.arr); /* Copy the password. */ strncpy((char *) password.arr, "TIGER", PWD_LEN); password.len = strlen((char *) password.arr); /* Register sql_error() as the error handler. */ EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--¥n"); /* Connect to ORACLE. Program will call sql_error() * if an error occurs when connecting to the default database. */ EXEC SQL CONNECT :username IDENTIFIED BY :password; printf("¥nConnected to ORACLE as user: %s¥n", username.arr); /* Loop, selecting individual employee's results */ total_queried = 0; for (;;) { /* Break out of the inner loop when a * 1403 ("No data found") condition occurs. */ EXEC SQL WHENEVER NOT FOUND DO break; for (;;) { emp_number = 0; printf("¥nEnter employee number (0 to quit): "); gets(temp_char); emp_number = atoi(temp_char); if (emp_number == 0) break; EXEC SQL SELECT ename, sal, NVL(comm, 0) INTO :emprec INDICATOR :emprec_ind FROM EMP WHERE EMPNO = :emp_number; /* Print data. */ printf("¥n¥nEmployee¥tSalary¥t¥tCommission¥n"); printf("--------¥t------¥t¥t----------¥n"); /* Null-terminate the output string data. */ emprec.emp_name.arr[emprec.emp_name.len] = '¥0'; printf("%-8s¥t%6.2f¥t¥t", emprec.emp_name.arr, emprec.salary); if (emprec_ind.comm_ind == -1) printf("NULL¥n"); else printf("%6.2f¥n", emprec.commission); total_queried++; } /* end inner for (;;) */ if (emp_number == 0) break; printf("¥nNot a valid employee number - try again.¥n"); } /* end outer for(;;) */ printf("¥n¥nTotal rows returned was %d.¥n", total_queried); printf("¥nG'day.¥n¥n¥n"); /* Disconnect from ORACLE. */ EXEC SQL COMMIT WORK RELEASE; exit(0); } void sql_error(msg) char *msg; { char err_msg[128]; int buf_len, msg_len; EXEC SQL WHENEVER SQLERROR CONTINUE; printf("¥n%s¥n", msg); buf_len = sizeof (err_msg); sqlglm(err_msg, &buf_len, &msg_len); if (msg_len > buf_len) msg_len = buf_len; printf("%.*s¥n", msg_len, err_msg); EXEC SQL ROLLBACK RELEASE; exit(1); }

2.9 サンプル・プログラム: SQL99構文を使用する単純な問合せ

このプログラムは前述のサンプルと似ていますが、SELECT、INSERT、DELETEおよびUPDATE文のSQL99構文を使用し、DECLARE CURSOR文のカーソル本体がサポートされています。

MODE=ORACLEプリコンパイラ・オプションを使用して、サンプル・プログラムをプリコンパイルします。

```
/*
 *  sql99.pc
 *
 *  Prompts the user for an employee number,
 *  then queries the emp table for the employee's
 *  name, salary and department.
 *
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
.
/* Define constants for VARCHAR lengths. */
#define     UNAME_LEN     30
#define     PWD_LEN        40
/* Declare variables.  No declare section is needed if MODE=ORACLE. */

VARCHAR     username[UNAME_LEN];
 /* VARCHAR is an Oracle-supplied struct */
 varchar     password[PWD_LEN];
 /* varchar can be in lower case also. */
/* Define a host structure for the output values of a SELECT statement.  */

struct{
   VARCHAR   emp_name[UNAME_LEN];
   float     salary;
   VARCHAR  dept_name[UNAME_LEN] ;
  } emprec;
/* Define an indicator struct to correspond to the host output struct. */
struct{
   short     emp_name_ind;
   short     sal_ind;
   short     dept_name;
  } emprec_ind;

/*  Input host variable. */
int         emp_number;
int         total_queried;
/* Include the SQL Communications Area. You can use #include or EXEC SQL
INCLUDE. */

#include <sqlca.h>
/* Declare error handling function. */
void sql_error(msg)
   char *msg;
   {
    char err_msg[128];
    size_t buf_len, msg_len;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("¥n%s¥n", msg);
    buf_len = sizeof (err_msg);
```

```
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s¥n", msg_len, err_msg);
    EXEC SQL ROLLBACK RELEASE;
  exit(EXIT_FAILURE);
  }

void main(){
   char temp_char[32];
  /* Connect to ORACLE-- * Copy the username into the VARCHAR. */
    strncpy((char *) username.arr, "scott", UNAME_LEN);
  /* Set the length component of the VARCHAR. */
    username.len = (unsigned short) strlen((char *) username.arr);
   /* Copy the password. */
     strncpy((char *) password.arr, "tiger", PWD_LEN);
     password.len = (unsigned short) strlen((char *) password.arr);
  /* Register sql_error() as the error handler. */
    EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--¥n");
  /* Connect to ORACLE.  Program will call sql_error() * if an error occurs
when connecting to the default database. */
     EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("¥nConnected to ORACLE as user: %s¥n", username.arr);
  /* Loop, selecting individual employee's results */
     total_queried = 0;
     for (;;)    {
                 emp_number = 0;
                 printf("¥nEnter employee number (0 to quit): ");
                 gets(temp_char);
                 emp_number = atoi(temp_char);
                 if (emp_number == 0)
                       break;
  /* Branch to the notfound label when the * 1403 ("No data found") condition
occurs. */
   EXEC SQL WHENEVER NOT FOUND GOTO notfound;

  /* The following query uses SQL99 syntax - RIGHT OUTER JOIN */
    EXEC SQL SELECT e.ename, e.sal, d.dname
    INTO :emprec INDICATOR :emprec_ind
    FROM EMP e RIGHT OUTER JOIN dept d
    ON e.deptno = d.deptno
    WHERE e.EMPNO = :emp_number;
  /* Print data. */
  printf("¥n¥nEmployee   Salary    Department Name¥n");
  printf("--------   -------   ------------------¥n");
  /* Null-terminate the output string data. */
    emprec.emp_name.arr[emprec.emp_name.len] = '¥0';
    emprec.dept_name.arr[emprec.dept_name.len]='¥0';
    printf("%s      %7.2f          %s ",             emprec.emp_name.arr,
  emprec.salary, emprec.dept_name.arr);
    total_queried++;
     continue;

  notfound:
     printf("¥nNot a valid employee number - try again.¥n");
  }

 printf("¥n¥nTotal rows returned was %d.¥n", total_queried);
 printf("¥nG'day.¥n¥n¥n");
/* Disconnect from ORACLE. */
   EXEC SQL ROLLBACK WORK RELEASE;
 exit(EXIT_SUCCESS);
```

```
}
```

3 データベースの概要

この章では、基本的なデータベースの概要と、トランザクション処理の実行方法について説明します。Oracleデータの変更内容の確定または取消しを制御する方法など、データベースの整合性を維持するための基本的な技術を学習します。

この章のトピックは、次のとおりです:

- データベースへの接続
- 高度な接続オプション
- トランザクション用語の定義
- トランザクションがデータベースを保護する方法
- トランザクションの開始および終了方法
- COMMIT文の使用
- SAVEPOINT文の使用
- ROLLBACK文
- RELEASEオプション
- SET TRANSACTION文
- デフォルト・ロックの上書き
- コミットにまたがるフェッチ
- 分散トランザクションの処理
- ガイドライン

3.1 データベースへの接続

各項で、CONNECT文の詳細な構文について説明します。構文は次のとおりです。

```
EXEC SQL CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
   [[ AT { dbname | :host_variable }] USING :connect_string ]
      [ {ALTER AUTHORIZATION :newpswd  |  IN { SYSDBA | SYSOPER } MODE} ] ;
```

データの問合せまたは操作をする前に、Pro*C/C++プログラムをデータベースに接続する必要があります。ログインするには、単にCONNECT文を使用します。

```
EXEC SQL CONNECT :username IDENTIFIED BY :password ;
```

*username*および*password*は、**char**またはVARCHARホスト変数です。

または、この文は次のようにも指定できます。

```
EXEC SQL CONNECT :usr_pwd;
```

ホスト変数*usr_pwd*には、スラッシュ文字(/)で区切られたユーザー名とパスワードが含まれます。

これらは、単純化したCONNECT文のサブセットです。

CONNECT文は、プログラムが実行する最初のSQL文であることが必要です。つまり、プリコンパイル・ユニット内では、他のSQL文をCONNECT文の前に物理的に置くことはできますが、論理的に置くことはできません。

Oracleのユーザー名とパスワードを別々に入力するには、2つのホスト変数を文字列またはVARCHARとして定義します。(ユーザー名とパスワードの両方を含むユーザー名を入力する場合、必要なホスト変数は1つのみです。)

CONNECTを実行する前に、ユーザー名とパスワードの変数を設定する必要があります。設定しない場合、CONNECTは失敗します。プログラムのプロンプトで値の入力を求めることも、次のように値をハードコードすることもできます。

```
char *username = "SCOTT";
char *password = "TIGER";
...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

ただし、ユーザー名とパスワードは、CONNECT文にはハードコードできません。また、引用符で囲んだリテラルは使用できません。たとえば、次の2つの文はどちらも無効です。

```
EXEC SQL CONNECT SCOTT IDENTIFIED BY TIGER;
EXEC SQL CONNECT 'SCOTT' IDENTIFIED BY 'TIGER';
```

ユーザー名およびパスワードをハードコードすることはお薦めしません。

関連項目

- CONNECT(実行可能埋込みSQL拡張機能)

3.1.1 ALTER AUTHORIZATION句を使用したパスワードの変更

Pro*C/C++のクライアント・アプリケーションでは、EXEC SQL CONNECT文を拡張し、実行時にユーザーのパスワードを変更できます。

この項では、様々なALTER AUTHORIZATION句の実行結果について説明します。

3.1.1.1 標準CONNECT

アプリケーションで次の文が発行されるとします。

```
EXEC SQL CONNECT ..;   /* No ALTER AUTHORIZATION clause */
```

通常の接続が実行されます。予想される結果は次のとおりです。

- アプリケーションが問題なく接続されます。
- アプリケーションは接続できるが、パスワードに関する警告を受ける。この警告は、パスワードの期限は切れているが、まだログインできる猶予期間であることを示しています。この時点で、ユーザーは、アカウントがロックされる前にパスワードを変更するように求められます。
- アプリケーションが接続に失敗します。次の原因が考えられます。
    - パスワードが間違っています。
    - アカウントが期限切れになっているか、またはロック状態です。

3.1.1.2 CONNECT文でのパスワードの変更

次のCONNECT文があるとします。

```
EXEC SQL CONNECT .. ALTER AUTHORIZATION :newpswd;
```

この文は、アプリケーションがアカウントのパスワードをnewpswdで指定した値に変更することを示します。変更後は、user/newpswdで接続試行が実行されます。次の結果が予想されます。

- アプリケーションは問題なく接続できる。
- アプリケーションが接続に失敗します。次のどちらかの原因が考えられます。
    - なんらかの理由でパスワードを認識できませんでした。パスワードは元のままです。
    - アカウントがロックされています。パスワードは変更できません。

3.1.2 Oracle Net Servicesを使用した接続

Oracle Net Servicesのドライバを使用して接続するには、tnsnames.ora構成ファイルまたはOracle Namesで定義されているサービス名を使用します。

Oracle Namesを使用する場合、ネーム・サーバーは、ネットワーク定義データベースからサービス名を取得します。

Oracle Net Servicesの詳細は、『*Oracle Net Services*管理者ガイド』を参照してください。

3.1.3 自動接続

ユーザー名を使用してOracleに自動的に接続できます。

```
CLUSTER$username
```

*username*は現行のオペレーティング・システムのユーザー名、CLUSTER$*username*は有効なOracleデータベース・ユーザー名です。(CLUSTER$の実際の値は、INIT.ORAパラメータ・ファイルに定義されています。)Pro*C/C++プリコンパイラに渡すのはスラッシュ文字のみです。次に例を示します。

```
...
char  *oracleid = "/";
...
EXEC SQL CONNECT :oracleid;
```

これにより、ユーザーCLUSTER$*username*で自動的に接続されます。たとえば、オペレーティング・システムのユーザー名がRHILLで、CLUSTER$RHILLが有効なOracleユーザー名の場合は、/を使用した接続により、ユーザーCLUSTER$RHILLでOracleに自動的にログインできます。

また、プリコンパイラには文字列内で/を渡すこともできます。ただし、その文字列の後続にブランクを入れないでください。たとえば、次のCONNECT文は失敗します。

```
...
char oracleid[10] = "/    ";
...
EXEC SQL CONNECT :oracleid;
```

3.1.3.1 AUTO_CONNECTプリコンパイラ・オプション

AUTO_CONNECT=YESで、最初の実行SQL文を処理するときにアプリケーションがまだデータベースに接続されていない場合、次のユーザーIDを使用して接続が試行されます。

```
CLUSTER$<username>
```

*username*は現行のオペレーティング・システムのユーザー名またはタスク名、CLUSTER$usernameは有効なOracleユーザーIDです。AUTO_CONNECTのデフォルト値はNOです。

AUTO_CONNECT=NOの場合、Oracleに接続するにはプログラムでCONNECT文を使用する必要があります。

3.1.3.2 SYSDBAまたはSYSOPERシステム権限

SYSDBAまたはSYSOPERシステム権限でログインするには、次のオプション文字列を他のすべての句の後に追加します。

```
[IN { SYSDBA | SYSOPER } MODE]
```

次に例を示します。

```
EXEC SQL CONNECT ... IN SYSDBA MODE ;
```

このオプションには次の制限があります。

- プリコンパイラのAUTO_CONNECT=YESオプション設定を使用する場合、このオプションは使用できません。
- CONNECT文にALTER AUTHORIZATIONキーワードを使用している場合、このオプションは使用できません。

関連項目

- ALTER AUTHORIZATION句を使用したパスワードの変更

3.2 高度な接続オプション

この項では、高度な接続で使用できるオプションについて説明します。

3.2.1 予備知識

ネットワーク上の通信ポイントは、ノードと呼ばれます。Oracle Netでは、ネットワーク上のノード間で情報(SQL文、データおよびステータス・コード)を送信できます。

プロトコルは、ネットワークへのアクセスに関する一連の規則です。この規則では、障害後のリカバリ手順、データの転送およびエラー検査のフォーマットなどが規定されます。

ローカル・ドメイン内のデフォルトのデータベースに接続するためにOracle Netの構文で使用するのは、そのデータベースのサービス名のみです。

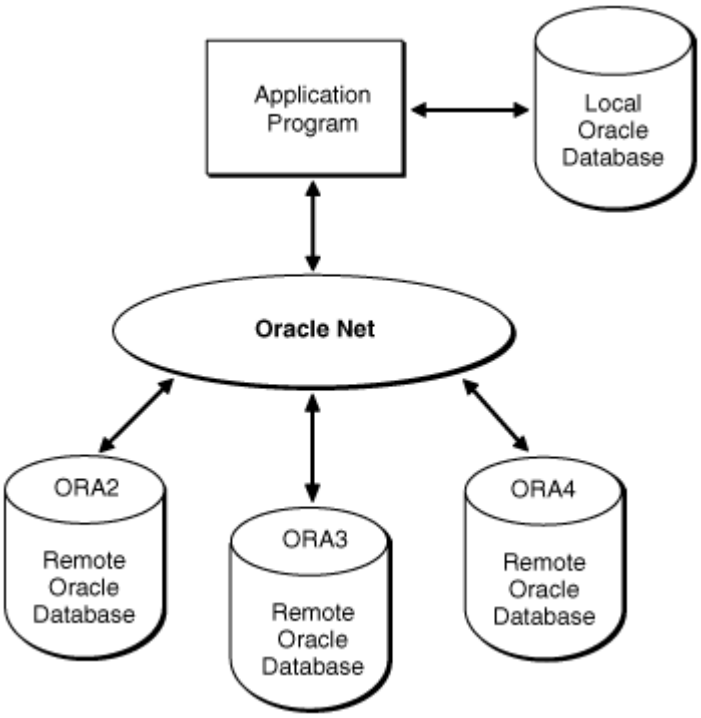
サービス名がデフォルト(ローカル)・ドメイン内にない場合は、グローバル指定(すべてのドメインの指定)を使用する必要があります。次に例を示します。

```
HR.XX.ORACLE.COM
```

3.2.2 同時ログイン

Pro*C/C++では、Oracle Net経由で分散処理がサポートされます。アプリケーションは、ローカル・データベースおよびリモート・データベースの任意の組合せに同時にアクセスしたり、同じデータベースへの複数の接続を確立できます。図3-1では、アプリケーション・プログラムはOracleの1つのローカル・データベースおよび3つのリモート・データベースと接続しています。ORA2、ORA3およびORA4は、CONNECT文で使用される論理名です。

図3-1 Oracle Net経由の接続



Oracle Netは、ネットワーク上の異なるマシン間およびオペレーティング・システム間に存在する境界を排除することによって、Oracleのツール製品に分散処理環境を提供します。この項では、Pro*C/C++によりOracle Net経由で分散処理がサポートされる方法について説明します。アプリケーションから可能な操作は、次のとおりです。

- 他のデータベースへの直接または間接アクセス
- ローカルおよびリモート・データベースの任意の組合せへの同時アクセス
- 同一のデータベースへの複数接続

Oracle Netのインストール方法および使用可能なデータベースの識別方法は、データベースの識別とアクセスおよびシステム固有のOracleドキュメントを参照してください。

3.2.3 デフォルトのデータベースおよび接続

各ノードにはデフォルトのデータベースがあります。CONNECT文でデータベース名のみを指定し、ドメインを指定しない場合、指定したローカル・ノードまたはリモート・ノード上のデフォルトのデータベースに接続されます。

デフォルトの接続は、AT句のないCONNECT文によって行われます。ローカルまたはリモートの任意のノード上のデフォルトまたは非デフォルトの任意のデータベースに接続できます。AT句のないSQL文は、デフォルトの接続に対して実行されます。逆に、非デフォルトの接続は、AT句があるCONNECT文により行われます。AT句を持つSQL文は、非デフォルトの接続に対して実行されます。

データベース名は一意にする必要がありますが、2つ以上のデータベース名で同じ接続を指定できます。したがって、任意のノード上のデータベースに対して複数の接続を確立できます。

3.2.4 明示的接続

通常は、Oracleへの接続を次のように確立します。

```
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

また、次の文も使用できます。

```
EXEC SQL CONNECT :usr_pwd;
```

*usr_pwd*には、*username*/*password*が含まれます。

次のユーザーIDを使用してOracleに自動的に接続できます。

```
CLUSTER$username
```

*username*は現行のオペレーティング・システムのユーザー名またはタスク名、CLUSTER$*username*は有効なOracleユーザーIDです。プリコンパイラに渡すのはスラッシュ文字(/)のみです。次に例を示します。

```
char oracleid = '/';
...
EXEC SQL CONNECT :oracleid;
```

これにより、ユーザーCLUSTER$*username*で自動的に接続されます。

データベースおよびノードを指定しない場合、現在のノード上のデフォルトのデータベースに接続されます。別のデータベースに接続する場合は、そのデータベースを明示的に指定する必要があります。

明示的接続では、SQL文で参照される接続名を指定して、別のデータベースに直接接続します。同時に複数のデータベースに接続することも、同じデータベースに複数回接続することもできます。

3.2.4.1 単一の明示的接続

次の例では、リモート・ノードにある単一の非デフォルトのデータベースに接続します。

```
/* declare needed host variables */
char  username[10]  = "scott";
char  password[10]  = "tiger";
char  db_string[20] = "NYNON";

/* give the database connection a unique name */
EXEC SQL DECLARE DB_NAME DATABASE;

/* connect to the nondefault database  */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME USING :db_string;
```

この例の識別子は、次の目的で使用されています。

- ホスト変数*username*および*password*は、有効なユーザーを識別します。
- ホスト変数*db_string*には、リモート・ノードにある非デフォルトのデータベースに接続するためのOracle Net構文が含まれています。
- 未宣言の識別子DB_NAMEは、非デフォルト接続の名前を指定します。これは、Oracleで使用される識別子で、ホスト変数でもプログラム変数でもありません。

USING句では、DB_NAMEに対応付けるネットワーク、マシンおよびデータベースを指定します。その後、AT句(DB_NAME付き)を使用しているSQL文は、*db_string*に指定したデータベースで実行されます。

もう1つの方法として、次の例に示すように、AT句で文字ホスト変数を使用できます。

```
/* declare needed host variables */
char  username[10]  = "scott";
char  password[10]  = "tiger";
char  db_name[10]   = "oracle1";
char  db_string[20] = "NYNON";

/* connect to the nondefault database using db_name */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT :db_name USING :db_string;
...
```

*db_name*がホスト変数の場合、DECLARE DATABASE文は不要です。DB_NAMEが未宣言の識別子の場合にのみ、CONNECT ... AT DB_NAME文を実行する前にDECLARE DB_NAME DATABASE文を実行する必要があります。

3.2.4.1.1 SQL操作

権限を付与されている場合は、非デフォルトの接続で任意のSQL DML文を実行できます。たとえば、次のように入力します。

```
EXEC SQL AT DB_NAME SELECT ...
EXEC SQL AT DB_NAME INSERT ...
EXEC SQL AT DB_NAME UPDATE ...
```

次の例では、*db_name*はホスト変数です。

```
EXEC SQL AT :db_name DELETE ...
```

*db_name*がホスト変数の場合、SQL文で参照されるすべてのデータベース表を、DECLARE TABLE文で定義する必要があります。定義しないと、プリコンパイラで警告が発行されます。

関連項目

- DECLARE TABLEの使用について
- DECLARE TABLE(Oracle埋込みSQLディレクティブ)

3.2.4.1.2 PL/SQLブロック

PL/SQLブロックは、AT句を使用して実行できます。次の例は構文を示しています。

```
EXEC SQL AT :db_name EXECUTE
    begin
        /* PL/SQL block here */
    end;
END-EXEC;
```

3.2.4.1.3 カーソルの制御

OPEN、FETCHおよびCLOSEなどのカーソルの制御文は例外で、AT句は使用しません。カーソルを明示的に識別されたデータベースに対応付ける場合は、次のようにDECLARE CURSOR文でAT句を使用します。

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor ...
EXEC SQL CLOSE emp_cursor;
```

*db_name*がホスト変数の場合は、DECLAREされたカーソルを参照するすべてのSQL文の適用範囲内で宣言する必要があります。たとえば、あるサブプログラム内でカーソルをOPENし、別のサブプログラムでそのカーソルからFETCHする場合は、*db_name*をグローバルに宣言する必要があります。

カーソルからOPEN、CLOSEおよびFETCHを実行する場合、AT句を使用しません。SQL文は、DECLARE CURSOR文のAT句で名前を付けられたデータベースか、カーソルの宣言でAT句が使用されていない場合はデフォルトのデータベースにおいて実行されます。

AT :*host_variable*句を使用すると、カーソルに対応付けられた接続を変更できます。ただし、カーソルがオープンされているときは対応付けを変更できません。次の例を考えてみます。

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
strcpy(db_name, "oracle1");
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
strcpy(db_name, "oracle2");
```

```
EXEC SQL OPEN emp_cursor;   /*  illegal, cursor still open */
EXEC SQL FETCH emp_cursor INTO ...
```

この例は、2番目のOPEN文を実行するときに*emp_cursor*がまだオープンされているため、無効となります。異なる接続に対して別々のカーソルが維持されることはありません。*emp_cursor*は1つのみ存在できます。別の接続のために再オープンするには、その前にクローズする必要があります。最後の例をデバッグするには、次のように、単にカーソルをクローズしてから再オープンします。

```
...
EXEC SQL CLOSE emp_cursor;  -- close cursor first
strcpy(db_name, "oracle2");
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
```

3.2.4.1.4 動的SQL

動的SQL文は、文中ではAT句が使用されないカーソル制御文に類似しています。

動的SQL方法1では、非デフォルトの接続で文を実行する場合は、AT句を使用する必要があります。次に例を示します。

```
EXEC SQL AT :db_name EXECUTE IMMEDIATE :sql_stmt;
```

方法2、3および4で非デフォルトの接続で文を実行する場合は、DECLARE STATEMENT文でのみAT句を使用します。PREPARE、DESCRIBE、OPEN、FETCHおよびCLOSEなど、その他の動的SQL文はAT句を使用しません。次の例に方法2を示します。

```
EXEC SQL AT :db_name DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL EXECUTE sql_stmt;
```

次の例は方法3を示します。

```
EXEC SQL AT :db_name DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor INTO ...
EXEC SQL CLOSE emp_cursor;
```

3.2.4.2 複数の明示的接続

単一の明示的接続の場合と同様に、複数の明示的接続にはAT *db_name*句を使用できます。次の例では、2つの非デフォルトのデータベースに同時に接続しています。

```
/* declare needed host variables */
char  username[10]   = "scott";
char  password[10]   = "tiger";
char  db_string1[20] = "NYNON1";
char  db_string2[20] = "CHINON";
...
/* give each database connection a unique name */
EXEC SQL DECLARE DB_NAME1 DATABASE;
EXEC SQL DECLARE DB_NAME2 DATABASE;
/* connect to the two nondefault databases */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT DB_NAME2 USING :db_string2;
```

識別子DB_NAME1およびDB_NAME2を宣言し、2つの非デフォルト・ノードのデフォルトのデータベースの名前を指定します。これにより、SQL文ではデータベースを名前で参照できます。

または、次の例のように、AT句でホスト変数を使用できます。

```
/* declare needed host variables */
char  username[10]   = "scott";
char  password[10]   = "tiger";
char  db_name[20];
char  db_string[20];
int   n_defs = 3;    /* number of connections to make */
...
for (i = 0; i < n_defs; i++)
{
    /* get next database name and OracleNet string */
    printf("Database name: ");
    gets(db_name);
    printf("OracleNet) string: ");
    gets(db_string);
    /* do the connect */
    EXEC SQL CONNECT :username IDENTIFIED BY :password
        AT :db_name USING :db_string;
}
```

この方法を使用すれば、次の例のように、同じデータベースに複数の接続を行うこともできます。

```
strcpy(db_string, "NYNON");
for (i = 0; i < ndefs; i++)
{
    /* connect to the nondefault database */
    printf("Database name: ");
    gets(db_name);
    EXEC SQL CONNECT :username IDENTIFIED BY :password
       AT :db_name USING :db_string;
}
...
```

複数の接続に同じOracle Net文字列を使用する場合も、接続ごとに異なるデータベース名を使用する必要があります。ただし、データベース名ではデフォルトおよび非デフォルトのデータベースの両方が識別されるため、1つのデータベース名で同じデータベースに2回接続できます。

3.2.4.3 データの整合性の確認

アプリケーション・プログラムでは、複数のリモート・データベースにあるデータを操作するトランザクションの整合性を確認する必要があります。つまり、プログラムはトランザクションのすべてのSQL文をコミットまたはロールバックする必要があります。これは、ネットワーク障害が発生した場合や、システムの1つがクラッシュした場合は不可能です。

たとえば、2つの会計データベースで作業しているとします。一方のデータベースで勘定の借方に記帳し、他方のデータベースで勘定の貸方に記帳してから、それぞれのデータベースでCOMMITを発行します。両方のトランザクションがコミットまたはロールバックされたかどうかは、プログラム側で確認する必要があります。

3.2.5 暗黙的接続

暗黙的接続は、Oracleの分散問合せ機能を通じてサポートされます。この機能には明示的接続は不要ですが、サポートされるのはSELECT文のみです。分散問合せを使用すると、単一のSELECT文で1つ以上の非デフォルトのデータベースにあるデータにアクセスできます。

分散問合せ機能はデータベース・リンクに依存しており、リンクにより接続事態ではなく、CONNECT文に名前が割り当てられます。 実行時には、指定したOracleサーバーにより埋込みSELECT文が実行され、非デフォルトのデータベースに暗黙的に接続されて、必要なデータが取得されます。

3.2.5.1 単一の暗黙的接続

次の例では、1つの非デフォルト・データベースに接続します。最初に、プログラムでは次の文が実行され、データベース・リンクが定義されます(通常、データベース・リンクは、DBAまたはユーザーが対話形式で確立します)。

```
EXEC SQL CREATE DATABASE LINK db_link
    CONNECT TO username IDENTIFIED BY password
    USING 'NYNON';
```

プログラムでは、次のようにデータベース・リンクを使用して非デフォルトのEMP表を問い合せできます。

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
    FROM emp@db_link
    WHERE DEPTNO = :dept_number;
```

データベース・リンクは、埋込みSQL文のAT句に使用されるデータベース名とは無関係です。単に、非デフォルトのデータベースの位置、データベースへのパス、使用するOracleユーザー名およびパスワードをOracleに対して指示します。データベース・リンクは、明示的に削除されるまで、データベース・ディクショナリに格納されます。

前述の例で、デフォルトのOracleサーバーは、データベース・リンク*db_link*を使用して、Oracle Net経由で非デフォルトのデータベースにログインします。問合せはデフォルトのサーバーに送られますが、非デフォルトのデータベースに転送されて実行されます。

データベース・リンクを簡単に参照できるように、次のようにシノニムを対話形式で作成できます。

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link;
```

その結果、プログラムで、次のように非デフォルトのEMP表に問合せができるようになります。

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
    FROM emp
    WHERE DEPTNO = :dept_number;
```

これにより、*emp*について位置の透過性が得られます。

3.2.5.2 複数の暗黙的接続

次の例では、2つの非デフォルト・データベースに同時に接続します。まず次の一連の文を実行し、2つのデータベース・リンクを定義して2つのシノニムを作成します。

```
EXEC SQL CREATE DATABASE LINK db_link1
    CONNECT TO username1 IDENTIFIED BY password1
        USING 'NYNON';
EXEC SQL CREATE DATABASE LINK db_link2
    CONNECT TO username2 IDENTIFIED BY password2
        USING 'CHINON';
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1;
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

その結果、プログラムで、次のように非デフォルトのEMP表とDEPT表に問合せができるようになります。

```
EXEC SQL SELECT ENAME, JOB, SAL, LOC
    FROM emp, dept
    WHERE emp.DEPTNO = dept.DEPTNO AND DEPTNO = :dept_number;
```

Oracleでは、*db_link1*にある非デフォルトのEMP表と*db_link2*にある非デフォルトのDEPT表を結合することにより、問合せが実行されます。

3.3 トランザクション用語の定義

トランザクションの説明に入る前に、この項で定義されている用語に慣れる必要があります。

Oracleが管理するジョブまたはタスクは、セッションと呼ばれます。アプリケーション・プログラムまたはSQL*Formsなどのツールを実行してデータベースに接続すると、ユーザー・セッションが開始されます。

Oracleでは、ユーザー・セッションを同時に機能させ、コンピュータ・リソースを共有できます。そのために、Oracleは同時実行性、つまり多数のユーザーによる同一データへのアクセスを制御する必要があります。同時実行性の制御が十分でないと、データの整合性が失われる可能性があります。つまり、データまたは構造への変更が誤った順序で行われるおそれがあります。

Oracleでは、ロック(エンキューとも呼ばれます)を使用してデータへの同時アクセスを制御します。ロックにより、データの表や行などのデータベース・リソースのユーザーに一時的な所有権が与えられます。そのため、ロックを使用しているユーザーが変更を終了するまで、他のユーザーはデータを変更できません。

デフォルトのロック機能がOracleのデータおよび構造を保護するため、明示的にリソースをロックする必要はありません。ただし、デフォルトのロックをオーバーライドした方が有利な場合は、表または行に対するデータ・ロックを要求できます。行の共有や排他など、数種類のロック・モードから選択できます。

複数のユーザーが同じデータベース・オブジェクトへのアクセスを試みると、デッドロックが発生する可能性があります。たとえば、同じ表を更新するユーザーが2人いる場合、それぞれ相手が現在ロックしている行の更新を試みると待機状態になります。それぞれのユーザーが、相手が使用中のリソースを待つことになるため、Oracleによりデッドロックが解除されるまで、どちらも処理を続行できません。Oracleでは、最低作業量を完了した関連するトランザクションにエラー信号が送られ、「リソース待機の間にデッドロックが検出されました。」というOracleエラー・コードがSQLCAの*sqlcode*に戻されます。

1人のユーザーによって問合せが行われている表を、同時に別のユーザーが更新すると、Oracleでは問合せ用の表データの読取り一貫性ビューが生成されます。つまり、ある問合せが開始され、進行していく間、その問合せによって読み込まれたデータは変更されません。更新アクティビティが続行している間、Oracleでは、表データのスナップショットを取り、変更内容をロールバック・セグメントに記録します。Oracleでは、ロールバック・セグメント内の情報に基づいて、読取り一貫性のある問合せ結果が作成され、必要に応じて変更内容が取り消されます。

3.4 トランザクションがデータベースを保護する方法

Oracleはトランザクション指向です。つまり、トランザクションを使用してデータの整合性が確保されます。トランザクションとは、あるタスクを完了するために定義する1つ以上の論理的に関連付けられたSQL文です。Oracleでは、一連のSQL文を一単位として扱い、それらの文によって実行されたすべての変更が、同時にコミット(確定)されるか、ロールバック(取消し)されます。トランザクションの途中でアプリケーション・プログラムに障害が発生すると、データベースは自動的にトランザクション前の状態にリストアされます。

次項では、トランザクションの設計および制御方法について説明します。特に、次の操作方法を学習します。

- データベースへの接続。
- 同時接続
- トランザクションの開始および終了
- COMMIT文を使用したトランザクションの確定
- ROLLBACK TO文とともにSAVEPOINT文を使用したトランザクションの部分的な取消し

- ROLLBACK文を使用したトランザクション全体の取消し
- RELEASEオプションの指定によるリソースの解放とデータベースのログオフ
- SET TRANSACTION文を使用した読取り専用トランザクションの設定
- FOR UPDATE句またはLOCK TABLE文を使用したデフォルト・ロックの上書き

この章で説明する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。

3.5 トランザクションの開始および終了方法

プログラムの最初の実行SQL文(CONNECT以外)によりトランザクションを開始します。1つのトランザクションが終了すると、次の実行SQL文により別のトランザクションが自動的に開始します。このように、すべての実行文はトランザクションの一部です。宣言SQL文はロールバックされません。また、コミットする必要もないため、トランザクションの一部とみなされません。

トランザクションは、次のいずれかの方法で終了します。

- COMMITまたはROLLBACK文を記述します。RELEASEオプションは、付けても付けなくてもかまいません。これにより、データベースへの変更を明示的に確定または取り消します。
- 実行の前と後に自動コミットを発行するデータ定義文(ALTER、CREATEまたはGRANTなど)を記述します。これにより、データベースへの変更を暗黙的に確定します。

システム障害が発生した場合や、ソフトウェアの問題、ハードウェアの問題または強制割込みなどが原因で、予期しないユーザー・セッション停止が発生した場合にも、トランザクションは終了します。そのトランザクションはOracleによりロールバックされます。

トランザクションの途中でプログラムに障害が発生すると、Oracleは発生したエラーを検出し、そのトランザクションをロールバックします。オペレーティング・システムに障害が発生すると、データベースがトランザクション前の状態にリストアされます。

3.6 COMMIT文の使用

プログラムをCOMMIT文またはROLLBACK文で分割しなければ、Oracleではそのプログラム全体が1つのトランザクションとみなされます(ただし、そのプログラムにデータ定義文が含まれている場合は、自動COMMITが発行されます)。

COMMIT文を使用すると、データベースへの変更を確定できます。変更をCOMMITするまで、他のユーザーは変更されたデータにアクセスできず、トランザクションの開始前の状態のデータが表示されます。特に、COMMIT文では次の処理が実行されます。

- 現行のトランザクション中にデータベースに対して行った変更をすべて確定します。
- これらの変更を他のユーザーが参照できるようにします。
- すべてのセーブポイントを消去します(次の項を参照)。
- 解析ロック以外の行および表のロックをすべて解除します。
- CURRENT OF句で参照されているカーソルをクローズします。MODE=ANSIの場合は、COMMIT文に指定されている接続の明示カーソルをすべてクローズします。
- トランザクションを終了します。

COMMIT文は、ホスト変数の値にも、プログラムの制御フローにも影響しません。

MODE=ORACLEの場合、CURRENT OF句で参照されていない明示カーソルは、COMMITの前後もオープンしたままです。これによってパフォーマンスが向上します。

これらの処理は通常の処理の一部分であるため、COMMIT文はプログラムのメイン・パスにインラインで設定する必要があります。プログラムの終了前に、保留中の変更を明示的にCOMMITする必要があります。コミットしない場合、保留中の変更はロールバックされます。次の例では、トランザクションをコミットしてOracleから切断します。

```
EXEC SQL COMMIT WORK RELEASE;
```

オプションのキーワードWORKには、ANSI互換性があります。RELEASEオプションを指定すると、プログラムで使用されているOracleリソース(ロックとカーソル)がすべて解放され、データベースからログオフされます。

データ定義文では、実行の前後に自動COMMITが発行されるため、データ定義文の後にCOMMIT文を記述する必要はありません。したがってデータ定義文が正常終了しても異常終了しても、その前のトランザクションがコミットされます。

関連項目

- コミットにまたがるフェッチ

3.6.1 DECLARE CURSOR文でのWITH HOLD句の使用

CURSORの後にWITH HOLD句を付けて宣言されているカーソルは、COMMIT後もオープン状態となります。この句の使用方法は、次の例のとおりです。

```
    EXEC SQL
        DECLARE C1 CURSOR WITH HOLD
        FOR SELECT ENAME FROM EMP
        WHERE EMPNO BETWEEN 7600 AND 7700
    END-EXEC.
```

UPDATEの場合は、カーソルを宣言しないでください。DB2では、デフォルト(コミット時に全カーソルをクローズする)を変更するためにWITH HOLD句が使用されます。Pro*COBOLでは、DB2からOracleへのアプリケーションの移行を簡単に行えるようにするために、この句が用意されています。MODE=ANSIと指定されているとき、OracleではDB2のデフォルトが使用されますが、ホスト変数はすべて宣言部で宣言する必要があります。宣言部を省略するには、次の項で説明するプリコンパイラ・オプション*CLOSE_ON_COMMIT*を使用します。

関連項目

- DECLARE CURSOR (埋込みSQLディレクティブ)

3.6.2 CLOSE_ON_COMMITプリコンパイラ・オプション

プリコンパイラ・オプションCLOSE_ON_COMMITを使用すると、MODE=ANSIのデフォルト動作をオーバーライドできます(コマンドラインにMODE=ANSIを指定した場合、WITH HOLD句で宣言されていないカーソルはコミット時にクローズされます)。

```
CLOSE_ON_COMMIT = {YES | NO}
```

デフォルトはNOです。このオプションは、コマンドラインまたは構成ファイルで入力する必要があります。

> 注意:
>
> このオプションは注意して使用してください。カーソルのオープン/クローズを何度も行うと、各 OPEN 文で再解析を行う必要があるためアプリケーションの処理速度が低下する可能性があります。

関連項目

- CLOSE_ON_COMMIT
- マクロ・オプションおよびマイクロ・オプション

3.7 SAVEPOINT文の使用

SAVEPOINT文を使用すると、トランザクション処理中にカレント・ポイントにマークを付けて名前を指定できます。マークを設定したそれぞれの点をセーブポイントと呼びます。たとえば、次の文により*start_delete*というセーブポイントを設定します。

```
EXEC SQL SAVEPOINT start_delete;
```

セーブポイントによってロング・トランザクションを分割できるため、より複雑なプロシージャを制御できるようになります。たとえば、単一のトランザクションが複数のファンクションを実行しているときに、それぞれのファンクションの前にセーブポイントを設定できます。これにより、ある関数が失敗した場合も、Oracleデータを前の状態に簡単にリストアし、リカバリしてから、その関数を再実行できます。

トランザクションの一部を取り消すには、ROLLBACK文とそのTO SAVEPOINT句を使用してセーブポイントを指定します。次の例は、MAIL_LIST表にアクセスして、新しいリストの挿入、古いリストの更新、(少数の)使用されていないリストの削除を行います。削除後に、SQLCA内の*sqlerrd*の3番目の要素で、削除された行数を調べます。削除された行数が予想以上に多い場合は、セーブポイント*start_delete*までロールバックして、その削除のみを取り消します。

```
...
for (;;)
{
   printf("Customer number? ");
   gets(temp);
   cust_number = atoi(temp);
   printf("Customer name? ");
   gets(cust_name);
  EXEC SQL INSERT INTO mail_list (custno, cname, stat)
     VALUES (:cust_number, :cust_name, 'ACTIVE');
...
}

for (;;)
{
   printf("Customer number? ");
   gets(temp);
   cust_number = atoi(temp);
   printf("New status? ");
   gets(new_status);
   EXEC SQL UPDATE mail_list
     SET stat = :new_status
     WHERE custno = :cust_number;
}
/* mark savepoint */
EXEC SQL SAVEPOINT start_delete;

EXEC SQL DELETE FROM mail_list
    WHERE stat = 'INACTIVE';
if (sqlca.sqlerrd[2] < 25)  /* check number of rows deleted */
    printf("Number of rows deleted is  %d¥n", sqlca.sqlerrd[2]);
else
{
    printf("Undoing deletion of %d rows¥n", sqlca.sqlerrd[2]);
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    EXEC SQL ROLLBACK TO SAVEPOINT start_delete;
}
```

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL COMMIT WORK RELEASE;
exit(0);
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error¥n");
exit(1);
```

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のすべてのセーブポイントが消去されます。ただし、ロールバックしたセーブポイントはそのまま残ります。たとえば、セーブポイントを5つマークし、3番目のセーブポイントまでロールバックすると、4番目と5番目のセーブポイントのみが消去されます。

2つのセーブポイントに同じ名前を付けると、最初のセーブポイントが消去されます。COMMIT文またはROLLBACK文では、すべてのセーブポイントが消去されます。

関連項目

- WHENEVERディレクティブの使用について

3.8 ROLLBACK文

ROLLBACK文を使用すると、保留状態のデータベースへの変更を取り消せます。たとえば表から行を誤って削除したときなどは、ROLLBACK文を使用して元のデータをリストアできます。TO SAVEPOINT句を使用すると、現行のトランザクションの途中の文までロールバックできます。したがって、変更をすべて取り消す必要はありません。

未完成のトランザクションを開始した場合(たとえば、SQL文が正常に実行されないなど)、ROLLBACKを使用すると起点まで戻ることができるため、データベースの整合性が維持されます。特に、ROLLBACK文では次の処理が実行されます。

- カレント・トランザクションで実行されたデータベースの変更を取り消します。
- すべてのセーブポイントを消去します。
- トランザクションを終了します。
- 解析ロック以外の行および表のロックをすべて解除します。
- CURRENT OF句で参照されているカーソルをクローズします。MODE=ANSIの場合は、すべての明示カーソルをクローズします。

ROLLBACK文は、ホスト変数の値やプログラム内の制御の流れには影響を与えません。

MODE=ORACLEの場合、CURRENT OF句で参照されていない明示カーソルは、ROLLBACKの前後もオープンしたままです。

特に、ROLLBACK TO SAVEPOINT文では次の処理が実行されます。

- 指定したセーブポイントがマークされた以降のデータベースへの変更を取り消します。
- 指定したセーブポイント以降のセーブポイントをすべて消去します。
- 指定したセーブポイントがマークされた以降に取得された行および表のロックをすべて解除します。

注意:

ROLLBACK TO SAVEPOINT 文では RELEASE オプションを指定できません。

ROLLBACK文は例外処理の一部になっているため、プログラムのメイン・パスではなくエラー処理ルーチン内に指定する必要があります。次の例では、トランザクションをロールバックして、Oracleとの接続を切断します。

```
EXEC SQL ROLLBACK WORK RELEASE;
```

オプションのキーワードWORKには、ANSI互換性があります。RELEASEオプションを指定すると、プログラムで使用されているリソースがすべて解放され、データベースから切断されます。

WHENEVER SQLERROR GOTO文からROLLBACK文が記述されているエラー処理ルーチンに分岐したときに、ROLLBACKでエラーが発生すると、プログラムが無限にループする可能性があります。無限ループを回避するには、次に示すようにROLLBACK文の前にWHENEVER SQLERROR CONTINUEを記述します。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

for (;;)
{
   printf("Employee number? ");
    gets(temp);
    emp_number = atoi(temp);
   printf("Employee name? ");
   gets(emp_name);
   EXEC SQL INSERT INTO emp (empno, ename)
       VALUES (:emp_number, :emp_name);
...
}
...
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error¥n");
exit(1);
```

プログラムが異常終了すると、Oracleによりトランザクションが自動的にロールバックされます。

関連項目

- RELEASEオプション

3.8.1 文レベルのロールバック

Oracleは、SQL文を実行する前に、暗黙的なセーブポイント(ユーザーは操作できません)を設定します。したがって、この文が失敗すると、自動的にロールバックされ、SQLCA内の*sqlcode*に適切なエラー・コードが戻されます。たとえば、INSERT文が一意の索引内に同じ値を挿入しようとしたためエラーが発生すると、この文はロールバックの対象になります。

Oracleは、デッドロックを解除するために単一のSQL文をロールバックすることもあります。Oracleは関係しているトランザクションの1つにエラーを通知し、そのトランザクション中の現在の文をロールバックします。

失われるのは失敗したSQL文で開始された作業のみです。つまり、現行のトランザクション内でこの文より前に行われた作業は保存されます。したがって、データ定義文が失敗しても、それ以前の自動コミットは取り消されません。

SQL文は実行前に必ず解析され、構文規則に従っているか、有効なデータベース・オブジェクトを参照しているかが検証されます。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。

3.9 RELEASEオプション

プログラムが異常終了すると、Oracleにより変更が自動的にロールバックされます。異常終了が発生するのは、プログラムが作業を明示的にコミットもロールバックもせずに、RELEASEオプションを使用してOracleから切断する場合です。正常終了が発生するのは、プログラムが正常に実行され、オープン状態のカーソルがクローズされ、作業が明示的にコミットまたはロールバックされ、Oracleから切断され、制御がユーザーに戻された場合です。

最後に実行されるSQL文が次のどちらかの場合、プログラムは正常終了します。

```
EXEC SQL COMMIT WORK RELEASE;
```

または

```
EXEC SQL ROLLBACK WORK RELEASE;
```

トークンWORKはオプションです。最後のSQL文が上のどちらでもない場合は、そのユーザー・セッションで取得したロックおよびカーソルはプログラムの終了後も解放されず、ユーザー・セッションがアクティブでなくなったことをOracleが認識するまで保持されます。この結果、マルチユーザー環境では、他のユーザーはロックされたリソースへのアクセスを必要以上に長く待たされる場合があります。

3.10 SET TRANSACTION文

SET TRANSACTION文を使用すると、読取り専用トランザクションを開始できます。読取り専用トランザクションでは反復可能読取りが行えるため、他のユーザーが更新中の1つ以上の表に対して、複数の問合せを実行する場合に便利です。次にSET TRANSACTION文の例を示します。

```
EXEC SQL SET TRANSACTION READ ONLY;
```

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。READ ONLYパラメータは必須です。これを使用しても、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文、COMMIT文およびROLLBACK文のみです。たとえば、INSERT文、DELETE文またはSELECT FOR UPDATE OF文を使用するとエラーが発生します。

読取り専用トランザクション中、複数の表と複数の問合せで構成された読取り一貫性ビューが作成され、すべての問合せがデータベースの同じ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。

読取り専用トランザクションは、COMMIT文またはROLLBACK文、データ定義文によって終了します。(データ定義文では暗黙的COMMITが発行されることを思い出してください。)

次の例では、店の管理者が読取り専用トランザクションを使用して、当日、先週および先月の売上を調べ、要約レポートを生成しています。このレポートは、このトランザクションの実行中にデータベースを更新する他のユーザーによる影響を受けません。

```
EXEC SQL SET TRANSACTION READ ONLY;
EXEC SQL SELECT sum(saleamt) INTO :daily FROM sales
    WHERE saledate = SYSDATE;
EXEC SQL SELECT sum(saleamt) INTO :weekly FROM sales
    WHERE saledate > SYSDATE - 7;
EXEC SQL SELECT sum(saleamt) INTO :monthly FROM sales
    WHERE saledate > SYSDATE - 30;
EXEC SQL COMMIT WORK;
    /* simply ends the transaction since there are no changes
       to make permanent */
/* format and print report */
```

3.11 デフォルト・ロックの上書き

デフォルトでは、多数のデータ構造がOracleにより自動的にロックされます。ただし、デフォルトのロックをオーバーライドして、別のロックを有効にする場合は、行または表を特定して、そこにデータ・ロックを要求できます。明示的なロックにより、トランザクション中に表に対するアクセスを共有または制限したり、複数の表および複数の問合せの読取り一貫性を保持したりできます。

SELECT FOR UPDATE OF文を使用すると、表の特定行を明示的にロックすることで、UPDATEまたはDELETEが実行されるまで、その行が変更されないようにできます。ただし、OracleではUPDATE時またはDELETE時に行レベルのロックが自動的に取得されます。したがって、UPDATEまたはDELETEの前に行をロックする場合にのみ、FOR UPDATE OF句を使用してください。

LOCK TABLE文を使用すると、表全体を明示的にロックできます。

3.11.1 FOR UPDATE OFの使用

UPDATE文またはDELETE文のCURRENT OF句で参照されるカーソルをDECLAREする場合は、FOR UPDATE OF句を使用すると行の排他ロックを取得できます。SELECT FOR UPDATE OF文では、更新または削除の対象となる行が識別され、アクティブ・セット内の各行がロックされます。これは、ある行内の既存の値に基づいて更新処理を行う場合に便利です。更新前に、その行が他のユーザーにより変更されないようにする必要があります。

FOR UPDATE OF句はオプションです。たとえば、次のようなコードがあるとします。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job, sal FROM emp WHERE deptno = 20
        FOR UPDATE OF sal;
```

ここでFOR UPDATE OF句を削除すると、次のようにコードが単純になります。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job, sal FROM emp WHERE deptno = 20;
```

CURRENT OF句は、必要に応じてFOR UPDATE句を追加するようにプリコンパイラに指示します。CURRENT OF句を使用して、カーソルから最後にFETCHされた行を参照します。

関連項目

- CURRENT OF句

3.11.1.1 制限事項

FOR UPDATE OF句を使用すると、複数の表を参照できません。

明示的なFOR UPDATE OFまたは暗黙的なFOR UPDATEによって行の排他ロックが取得されます。行はすべて、FETCH時ではなくOPEN時にロックされます。行ロックが解除されるのは、COMMIT時またはROLLBACK時です(セーブポイントまでROLLBACKする場合は、解除されません)。したがって、COMMIT後はFOR UPDATEカーソルからFETCHできません。

3.11.2 LOCK TABLEの使用

LOCK TABLE文を使用すると、指定したロック・モードで1つ以上の表をロックできます。たとえば、次の文は行共有モードでEMP表をロックします。行共有ロックでは、表への同時アクセスが可能です。他のユーザーが表全体をロックして排他使用することはできません。

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT;
```

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、同時に多数のユーザーが1つの表に対して行共

有ロックを取得できる一方で、排他ロックを取得できるのは一度に1ユーザーのみです。あるユーザーが表を排他ロックしている間は、他のユーザーはその表の行をINSERT、UPDATEまたはDELETEできません。

オプションのキーワードNOWAITを指定すると、他のユーザーが表をロックしている場合は、その表の解放を待機しないようにOracleに対して指示できます。制御はただちにプログラムに戻されるため、プログラムではロックの取得を再度試みるまでの間に別の作業ができます。(SQLCA内の*sqlcode*をチェックすると、LOCK TABLEが失敗したか確認できます。)NOWAITを省略すると、表が利用可能になるまで、Oracleは待機します。待機の時間制限は設定されていません。

表がロックされていても、他のユーザーは表に対して問い合せできますが、問合せを実行しても表のロックを取得できません。したがって、問合せが他の問合せや更新を妨げることはなく、更新が問合せを妨げることもありません。2つの異なるトランザクションで同じ行の更新が試みられる場合にのみ、一方のトランザクションが他方のトランザクションの完了まで待機の状態になります。

LOCK TABLE文は、暗黙的にすべてのカーソルをクローズします。

表のロックは、トランザクションがCOMMITまたはROLLBACKを発行すると解除されます。

関連項目

- ロック・モード

3.12 コミットにまたがるフェッチ

COMMITとFETCHを併用する場合は、CURRENT OF句を使用しないでください。かわりに、各行のROWIDをSELECTしてから、その値を使用して更新または削除中の現在行を識別します。次に例を示します。

```
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal, ROWID FROM emp WHERE job = 'CLERK';
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary, :row_id;
    ...
    EXEC SQL UPDATE emp SET sal = :new_salary
        WHERE ROWID = :row_id;
    EXEC SQL COMMIT;
...
}
```

ただし、FETCHされた行はロックされません。つまり、ある行を読み取っても、その行を更新または削除する前に別のユーザーがその行を変更すると、結果が一貫性のないものになる可能性があります。

3.13 分散トランザクションの処理

分散データベースとは、異なるノード上の複数の物理データベースで構成される単一の論理データベースです。分散型の文とは、データベース・リンクによってリモート・ノードにアクセスする任意のSQL文です。分散トランザクションには、分散データベースの複数のノードでデータを更新するための分散型の文が、1つ以上設定されています。その更新が1つのノードのみに影響するときは、そのトランザクションは分散型ではありません。

COMMITを発行すると、分散トランザクションによる影響を受ける各データベースの変更が確定されます。COMMITのかわりにROLLBACKを発行すると、すべての変更が取り消されます。ただし、コミットまたはロールバック中にネットワークやマシンで障害が発生すると、分散トランザクションの状態は不明またはインダウトになることがあります。そのような場合、FORCE TRANSACTIONシステム権限があれば、FORCE句を使用して、ローカル・データベースでトランザクションを手動でコミットまた

はロールバックできます。このトランザクションは、データ・ディクショナリ・ビューDBA_2PC_PENDINGにあるトランザクションIDを引用符で囲んだリテラルで指定する必要があります。次に例を示します。

```
EXEC SQL COMMIT FORCE '22.31.83';
...
EXEC SQL ROLLBACK FORCE '25.33.86';
```

FORCEは指定されたトランザクションのみコミットまたはロールバックするため、カレント・トランザクションには影響しません。インダウト・トランザクションは、手動でセーブポイントまでロールバックできません。

COMMIT文中のCOMMENT句を使用すると、分散トランザクションと対応付けるためのコメントを指定できます。トランザクションがインダウトになると、COMMENTで指定したテキストが、OracleによりトランザクションIDとともにデータ・ディクショナリ・ビューDBA_2PC_PENDINGに格納されます。長さ50文字以内の引用符付きリテラルを指定する必要があります。次に例を示します。

```
EXEC SQL COMMIT COMMENT 'In-doubt trans; notify Order Entry';
```

> **注意:**
>
> COMMENT 句は、将来のリリースでは廃止になる予定です。COMMENT 句ではなくトランザクション名を使用することをお薦めします。

関連項目

- トランザクション

3.14 ガイドライン

次のガイドラインに従うと、いくつかの問題を回避できます。

3.14.1 アプリケーションの設計

アプリケーションを設計するときは、論理的に関連する処理を1つのトランザクション内にグループ化してください。正しく設計されたトランザクションには、与えられた作業を完了するために必要なステップが、すべて過不足なく含まれています。

表を参照するデータは一貫している必要があります。したがって、トランザクション内のSQL文は、一貫した方法でデータを変更する必要があります。 たとえば、2つの銀行口座間の資金振替には、一方の口座の借方勘定と他方の口座の貸方勘定が含まれています。どちらの処理も、正常終了または失敗が同時であることが必要です。一方の口座への新規預金など、関連のない更新はトランザクションに含めないでください。

3.14.2 ロックの取得

アプリケーション・プログラムにSQLのロック文が含まれている場合は、ロックを要求するOracleユーザーに、そのロックを取得する権限があるか確認してください。データベース管理者(DBA)は、どの表でもロックできます。それ以外のユーザーは、自分が所有する表または権限を持つ表(ALTER、SELECT、INSERT、UPDATEおよびDELETEなど)のみロックできます。

3.14.3 PL/SQLの使用

PL/SQLブロックがトランザクションの一部になっている場合、そのブロック内のCOMMITとROLLBACKは、そのトランザクション

全体に影響します。次の例では、ROLLBACKはUPDATEおよびINSERTによる変更を取り消します。

```
EXEC SQL INSERT INTO EMP ...
EXEC SQL EXECUTE
    BEGIN
        UPDATE emp ...
        ...
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK;
        ...
    END;
END-EXEC;
...
```

4 データ型とホスト変数

この章では、Pro*C/C++プログラムの作成に必要な基本的な情報について説明します。この章のトピックは、次のとおりです:

- Oracleのデータ型
- ホスト変数
- インジケータ変数
- VARCHAR変数
- カーソル変数
- コンテキスト変数
- ユニバーサルROWID
- ホスト構造体
- ポインタ変数
- グローバリゼーション・サポート
- NCHAR変数

この章には、学習用の完全なデモンストレーション・プログラムもいくつか記載されています。これらのプログラムには、この章で説明する技法の使用例が示されています。これらはdemoディレクトリにあり、オンラインで使用できるため、コンパイル、実行および必要に応じた変更もできます。

4.1 Oracleのデータ型

Oracleでは、内部データ型と外部データ型という2種類のデータ型が認識されます。内部データ型は、Oracleでデータベース表に列値を格納する方法と、NULL、SYSDATE、USERなどの疑似列値の表現に使用する形式を指定します。外部データ型は、入力ホスト変数と出力ホスト変数に値を格納するための形式を指定します。

関連項目

- *Oracle*の組込みデータ型

4.1.1 内部データ型

表4-1は、Oracleでデータベースの列に格納される値に使用される内部データ型を示しています。

表4-1 Oracle内部データ型

名前	説明
VARCHAR2	可変長文字列(4000 バイト以下)。
NVARCHAR2 または NCHAR VARYING	可変長シングルバイト文字列または各国語キャラクタ文字列(4000 バイト以下)。
NUMBER	100 をベースとして表された、精度と位取りのある数値。

名前	説明
LONG	可変長文字列(2**31-1 バイト以下)。
BINARY_FLOAT	32 ビット浮動小数点数(4 バイト)。
BINARY_DOUBLE	64 ビット浮動小数点数(8 バイト)。
TIMESTAMP	日付の年、月、日の値、および、時刻の時、分、および秒の値(7 または 11 バイト)。
DATE	固定長の日付+時刻値(7 バイト)。
INTERVAL YEAR	年および月単位の期間を格納します(5 バイト)。
INTERVAL DAY	日、時、分および秒単位で期間を格納します(11 バイト)。
RAW	可変長バイナリ・データ(2000 バイト以下)。
LONG RAW	可変長バイナリ・データ(2**31-1 バイト以下)。
ROWID	バイナリ値
UROWID	バイナリ値(4000 バイト以下)
CHAR	固定長文字列(2000 バイト以下)。
NCHAR	固定長シングルバイト文字列または各国語キャラクタ文字列(2000 バイト以下)。
CLOB	文字データ(4GB 以下)
NCLOB	各国語キャラクタ・セット・データ(4GB 以下)。
BLOB	バイナリ・データ(4GB 以下)
BFILE	外部ファイル・バイナリ・データ(4GB 以下)。

これらの内部データ型は、C言語のデータ型とは大きく異なる場合があります。たとえば、C言語にはOracleのNUMBERデータ型と同等のデータ型はありません。ただし、NUMBERは、**float**や**double**などのC言語のデータ型との間で変換できますが、ある程度の制限があります。たとえば、OracleのNUMBERデータ型は小数点以下最大38桁の精度で指定できますが、現行のC言語の実装では**double**をそこまでの精度で表せるデータ型はありません。

OracleのNUMBERデータ型は値を正確に(精度の制限内で)表しますが、浮動小数点形式では10.0などの値を正確に表すことができません。

構造化されていないデータ(テキスト、グラフィック・イメージ、ビデオ・クリップまたはサウンド波形)を格納するには、LOBデータ型を使用します。BFILEデータは、データベース外部のオペレーティング・システム・ファイルに格納されます。LOB型には、データの位置を指定するロケータが格納されます。

NCHARとNVARCHAR2は、マルチバイト・キャラクタ・データの格納に使用されます。

関連項目

- LOB

4.1.2 外部データ型

表4-2のように、外部データ型にはすべての内部データ型と、C言語の構文とほぼ一致するいくつかのデータ型が含まれています。たとえば、STRING外部データ型は、C言語ではNULL終了記号文字列を指します。

表4-2 Oracle外部データ型

名前	説明
VARCHAR2	可変長文字列(65535 バイト以下)。
NUMBER	100 をベースとして表された 10 進数。
INTEGER	符号付き整数。
FLOAT	実数。
STRING	NULL で終了する可変長文字列。
VARNUM	10 進数(NUMBER と同様だが表現の長さのコンポーネントが含まれる)。
LONG	固定長文字列(2**31-1 バイトまで)。
VARCHAR	可変長文字列(65533 バイト以下)。
ROWID	バイナリ値(外部の長さはシステムに依存)。
DATE	固定長の日付/時刻値(7 バイト)。
VARRAW	可変長バイナリ・データ(65533 バイト以下)。
RAW	固定長バイナリ・データ(65535 バイト以下)。
LONG RAW	固定長バイナリ・データ(2**31-1 バイト以下)。
UNSIGNED	符号なし整数。

名前	説明
LONG VARCHAR	可変長文字列(2**31-5 バイト以下)。
LONG VARRAW	可変長バイナリ・データ(2**31-5 バイト以下)。
CHAR	固定長文字列(65535 バイト以下)。
CHARZ	固定長の NULL 終了文字列(65534 バイト以下)。
CHARF	CHAR のデフォルトを VARCHAR2 または CHARZ ではなく CHAR にするために、TYPE 文または VAR 文で使用する。

次にOracleデータ型について簡単に説明します。

4.1.2.1 VARCHAR2

VARCHAR2データ型を使用して、可変長文字列を格納します。VARCHAR2値の最大長は64KBです。

VARCHAR2(*n*)値の最大長は、文字数ではなくバイト数で指定します。したがって、VARCHAR2(*n*)変数にマルチバイト文字を格納すると、最大長は*n*文字より少なくなります。

Oracleでは、CHAR_MAP=VARCHAR2オプションを使用してプリコンパイルすると、**char[n]**または**char**として宣言しているすべてのホスト変数に、VARCHAR2データ型が割り当てられます。

4.1.2.1.1 入力時

Oracleは入力ホスト変数に指定されたバイト数を読み込み、後続の空白文字を取り除き、入力値をターゲット・データベース列に格納します。ここでは注意が必要です。未初期化ホスト変数には、一連のヌル文字が含まれている場合があります。したがって、常に文字入力ホスト変数の宣言長まで空白文字で埋め、ヌル文字で終了しないでください。

入力値がデータベース列の定義より長い場合は、エラーが発生します。入力値がすべて空白の場合は、1つのNULLと同様に扱われます。

文字値が有効な数値を表している場合は、Oracleはその文字値をNUMBER列値に変換できます。文字値が有効な数値を表していない場合は、エラーが発生します。

4.1.2.1.2 出力時

Oracleは出力ホスト変数に指定されたバイト数を、必要に応じて空白文字で埋めて戻します。続いて、出力値をターゲット・ホスト変数に割り当てます。NULLが戻されると、Oracleはホスト変数に空白文字を埋めます。

出力値がホスト変数の宣言長より長い場合、Oracleはホスト変数に割り当てる前に出力値を切り捨てます。そのホスト変数に標識変数が対応付けられている場合、Oracleは標識変数を出力値の元の長さに設定します。

Oracleでは、NUMBER列値を文字値に変換できます。文字ホスト変数の長さによって精度が決定します。ホスト変数の長さがその数に対して短すぎる場合は、科学表記法が使用されます。たとえば、列値123456789を長さ6の文字ホスト変数にSELECTすると、値1.2E08が戻されます。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.2.2 NUMBER

NUMBERデータ型を使用して、固定小数点数または浮動小数点数を格納します。精度および位取りを指定できます。

NUMBER値の最大精度は38です。強度範囲は1.0E-130から9.99...9E125(9が38個に続けて0が88個)です。位取りの範囲は-84から127です。

NUMBER値は、可変長形式(1バイトの指数部に19バイトの仮数部が続く)で格納されます。指数バイトの上位1ビットは符号ビットであり、正数の場合に設定します。下位7ビットは強度を表します。

仮数は38桁の数値を形成し、各バイトは100をベースとする2桁を表します。仮数の符号は、最初(左端)のバイトの値で指定されます。101より大きければ仮数は負であり、その1桁目は左端のバイトから101を差し引いた値と等しくなります。

出力時、ホスト変数にはOracleで内部的に表されたとおりの数値が含まれます。予想される最大の数に対応するには、出力ホスト変数を22バイトの長さにする必要があります。数値を表すために使用されるバイトのみが戻されます。Oracleでは出力値の空白を埋め込んだり、NULLで終了させたりしません。戻された値の長さを知る必要がある場合は、かわりにVARNUMデータ型を使用します。

この外部データ型を使用する必要はほとんどありません。

4.1.2.3 INTEGER

INTEGERデータ型を使用して、小数部のない数を格納します。整数は符号付きの2バイト、4バイトまたは8バイトの2進数です。ワード内のバイトの順序付けはシステムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時には、列値が実数であれば、小数部が切り捨てられます。

4.1.2.4 FLOAT

FLOATデータ型を使用して、小数部を持つ数、あるいはINTEGERデータ型の容量を超える数を格納します。数値はご使用のコンピュータの浮動小数点書式を使用して表され、通常、4または8バイトの記憶域を必要とします。入力および出力ホスト変数に長さを指定する必要があります。

Oracleでは、数の内部形式が10進数であるため、ほとんどの浮動小数点処理よりも高い精度で数を表現できます。したがって、FLOAT変数へのフェッチを行うと、精度が低下する可能性があります。

4.1.2.5 STRING

STRINGデータ型は、VARCHAR2データ型に似ていますが、STRING値は常にヌル文字で終了するという違いがあります。Oracleでは、CHAR_MAP=STRINGオプションを使用してプリコンパイルすると、char[n]またはcharとして宣言しているすべてのホスト変数に、STRINGデータ型が割り当てられます。

4.1.2.5.1 入力時

Oracleは指定された長さを使用して、ヌル終端文字のスキャンを制限します。ヌル終端文字が見つからなければ、エラーが生成されます。長さを指定しなければ、最大長は2000バイトと想定されます。STRINGの値の最小長は2バイトです。最初の文字がヌル終端文字で、指定した長さが2の場合、列がNOT NULLと定義されていなければOracleはNULLを挿入します。列がNOT NULLと定義されている場合はエラーが発生します。空白のみの値はそのまま格納されます。

4.1.2.5.2 出力時

Oracleは戻された最後の文字に1バイトのヌル文字を追加します。文字列長が指定された長さを超える場合は、Oracleは出力値を切り捨て、1バイトのヌル文字を追加します。NULLがSELECTされた場合は、Oracleは最初の文字位置に1バイトのヌル文字を入れて戻します。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.2.6 VARNUM

VARNUMデータ型はNUMBERデータ型に似ていますが、VARNUM変数の1バイト目にその表現の長さが格納されるという違いがあります。

入力時には、ホスト変数の1バイト目を値の長さに設定する必要があります。出力時には、ホスト変数に長さとそれに続いてOracleで内部的に表現された数が含まれます。この数が最大になっても対応できるように、ホスト変数の長さは22バイトにする必要があります。列値をVARNUMホスト変数にSELECTした後で、1バイト目をチェックして値の長さを取得できます。

通常、このデータ型を使用する理由はほとんどありません。

4.1.2.7 LONG

LONGデータ型を使用して、固定長文字列を格納します。

LONGデータ型はVARCHAR2データ型に似ていますが、LONG値の最大長は2147483647バイト、つまり2GBであるという違いがあります。

4.1.2.8 VARCHAR

VARCHARデータ型を使用して、可変長文字列を格納します。VARCHAR変数では、2バイト長のフィールドの後に65533バイト以下の文字列フィールドが続きます。ただし、VARCHAR配列要素では、文字列フィールドの最大長は65530バイトです。VARCHAR変数の長さを指定するときは、長さフィールド用に必ず2バイトを付加してください。さらに長い文字列には、LONG VARCHARデータ型を使用してください。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.2.9 ROWID

索引構成表の行には、永続物理アドレスは設定されていません。論理ROWIDには、物理ROWIDの場合と同じ構文を使用してアクセスします。このため、物理ROWIDは、データ・オブジェクト番号(同じセグメント内のスキーマ・オブジェクト)を格納します。

論理ROWIDと物理ROWIDの両方(およびOracle以外の表のROWID)をサポートするために、ユニバーサル*ROWID*が定義されました。

文字ホスト変数を使用すると、ROWIDを読取り可能な書式で格納できます。ROWIDを文字ホスト変数にSELECTまたはFETCHすると、その2進値は18バイトの文字列に変換され、次の形式で戻されます。

```
BBBBBBBB.RRRR.FFFF
```

ここで、BBBBBBBBはデータベース・ファイルのブロック、RRRRはブロック内の行(最初の行は0)、FFFFはデータベース・ファイルを示します。これらの値は16進数です。たとえば、次の行IDがあるとします。

```
0000000E.000A.0007
points to the 11th row in the 15th block in the 7th database file.
```

通常、ROWIDを文字ホスト変数にFETCHし、ホスト変数をUPDATE文またはDELETE文のWHERE句のROWID疑似列と比較します。そのようにして、カーソルによってフェッチされた最終行を識別できます。

> 注意:
>
> 完全な移植性が必要な場合、あるいはアプリケーションで Oracle Open Gateway テクノロジを使用して Oracle 以外のデータベースと通信する場合は、ホスト変数を宣言するときに最大長を(18 ではなく)256 に指定してください。ホスト変数の内容については予測できませんが、ホスト変数は SQL 文中で通常どおりに動作します。

関連項目

- ユニバーサルROWID
- CURRENT OFの疑似実行について

4.1.2.10 DATE

DATEデータ型を使用して、日付と時刻を7バイトの固定長フィールドに格納します。表4-3に示すように、世紀、年、月、日、時(24時間制)、分および秒は、左から右にこの順序で格納されます。

表4-3 DATE書式

日付データ型	世紀	年	月	日	時	分	秒
バイト	1	2	3	4	5	6	7
意味	世紀	年	月	日	時	分	秒
例	119	194	10	17	14	24	13

1994 年 10 月 17 日午後
1 時 23 分 12 秒

世紀と年を表すバイトは、100を加算した表記です。時間、分および秒は、1を加算した表記です。B.C.E.(西暦紀元前)の日付は99以下です。エポックは、紀元前4712年1月1日です。この日付の場合、世紀のバイトは53で、年のバイトは88です。時間のバイト範囲は1から24です。分と秒の範囲は、1から60です。時刻のデフォルトは、午前零時(1, 1, 1)です。

通常、DATEデータ型はほとんど使用されません。

4.1.2.11 RAW

RAWデータ型を使用して、バイナリ・データまたはバイト文字列を格納します。RAW値の最大長は65535バイトです。

RAWデータはCHARACTERデータに似ていますが、OracleではRAWデータは意味がないものと解釈され、システム間でRAWデータを転送するときにはキャラクタ・セットは変換されないという違いがあります。

4.1.2.12 VARRAW

VARRAWデータ型を使用して、可変長のバイナリ・データまたはバイト文字列を格納します。VARRAWデータ型はRAWデータ型に似ていますが、VARRAW変数は2バイトの長さフィールドの後に長さ65533バイト以下のデータ・フィールドが付いているという違いがあります。さらに長い文字列には、LONG VARRAWデータ型を使用してください。

VARRAW変数の長さを指定するときは、長さフィールド用の2バイトが含まれているかを確認してください。変数の最初の2バイトは、整数として解釈できる必要があります。

VARRAW変数の長さを取得するには、長さフィールドを参照してください。

4.1.2.13 LONG RAW

LONG RAWデータ型を使用して、バイナリ・データまたはバイト文字列を格納します。LONG RAW値の最大長は2147483647バイト、つまり2GBです。

LONG RAWデータはLONGデータと似ていますが、OracleではLONG RAWデータの意味は解釈されず、LONG RAWデータをあるシステムから別のシステムへ送信してもキャラクタ・セットは変換されません。

4.1.2.14 UNSIGNED

UNSIGNEDデータ型を使用して、符号なし整数を格納します。符号なし整数は、2バイトまたは4バイトの2進数です。ワード内のバイトの順序付けはシステムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時には、列値が浮動小数点数であれば、小数部が切り捨てられます。

4.1.2.15 LONG VARCHAR

LONG VARCHARデータ型を使用して、可変長文字列を格納します。LONG VARCHAR変数では、4バイトの長さフィールドに文字列フィールドが続きます。文字列フィールドの最大長は2147483643(2**31 - 5)バイトです。VAR文またはTYPE文に使用するLONG VARCHARの長さを指定する場合は、4バイトの長さフィールドを含めないでください。

4.1.2.16 LONG VARRAW

LONG VARRAWデータ型を使用して、可変長のバイナリ・データまたはバイト文字列を格納します。LONG VARRAW変数では、4バイトの長さフィールドにデータ・フィールドが続きます。データ・フィールドの最大長は2147483643バイトです。VAR文またはTYPE文に使用するLONG VARRAWの長さを指定する場合は、4バイトの長さフィールドを含めないでください。

4.1.2.17 CHAR

CHARデータ型は、固定長文字列の格納に使用します。CHAR値の最大長は65535バイトです。

4.1.2.17.1 入力時

Oracleでは、入力ホスト変数に指定されたバイト数を読み取り、後続の空白を切り捨てずに、ターゲット・データベース列に入力値を格納します。

入力値がデータベース列の定義より長い場合は、エラーが発生します。入力値がすべて空白の場合は、空白が文字値と同様に扱われます。

4.1.2.17.2 出力時

Oracleは出力ホスト変数に指定されたバイト数を戻し、必要に応じて空白埋込みを行ってから、出力値をターゲット・ホスト変数に割り当てます。NULLが戻されると、Oracleはホスト変数に空白文字を埋めます。

出力値がホスト変数の宣言長より長い場合、Oracleはホスト変数に割り当てる前に出力値を切り捨てます。標識変数が使用可能な場合、標識変数は出力値の元の長さに設定されます。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.2.18 CHARZ

DBMS=V7またはV8の場合、デフォルトでは、OracleはPro*C/C++プログラム内のすべての文字ホスト変数にCHARZデータ型を割り当てます。CHARZデータ型は、ヌル文字で終了する固定長文字列を示します。CHARZ値の最大長は65534バイトです。

4.1.2.18.1 入力時

CHARZデータ型とSTRINGデータ型の機能は同じです。入力値はNULL文字で終了する必要があります。ヌル終端文字は、文字列の区切り記号としての役割のみを果し、格納データの一部にはなりません。

4.1.2.18.2 出力時

CHARZホスト変数を必要に応じて空白文字で埋めて、ヌル文字で終了します。この出力値は、データが長すぎるために切捨てが必要な場合にも、常にヌル文字で終了します。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.2.19 CHARF

CHARFデータ型は、EXEC SQL TYPE文およびEXEC SQL VAR文で使用します。DBMSオプションをV7またはV8に設定してプリコンパイルするときに、TYPE文またはVAR文で外部データ型CHARを指定すると、C言語のデータ型、もしくは固定長でヌル文字で終了するデータ型であるCHARZに同値化されます。

ただし、これらの型に同値化するのではなく、固定長の外部型CHARに同値化することが必要な場合もあります。外部型CHARFを使用すると、C言語のデータ型または変数は、DBMS値に関係なく常に固定長のANSIデータ型CHARと同値化されます。CHARFによりC言語のデータ型がVARCHAR2またはCHARZと同値化されることはありません。そのかわり、CHAR_MAP=CHARFオプションを設定すると、char[n]またはcharとして宣言されたホスト変数はすべてCHAR文字列と同値化されます。明示的にNULLが選択された場合、ホスト変数の値は予測不能です。標識変数の値がNULLかどうかをチェックする必要があります。

4.1.3 その他の外部データ型

この項では、その他の外部データ型について説明します。

4.1.3.1 日時および時間隔のデータ型

ここでは、日時および時間隔のデータ型の概要を説明します。

関連項目

- Oracle Database SQL言語リファレンス

4.1.3.2 ANSI DATE

ANSI DATEはDATEに基づいていますが、時刻部分が含まれていません。(したがって、タイム・ゾーンも含まれていません。)ANSI DATEは、DATEデータ型のANSI仕様部の後に指定します。ANSI DATEをDATEまたはタイムスタンプ・データ型に割り当てると、Oracle DATEの時間部分およびタイムスタンプがゼロに設定されます。DATEまたはタイムスタンプをANSI DATEに割り当てると、時刻部分は無視されます。

このデータ型ではなく、日付および時刻が含まれているTIMESTAMPデータ型の使用をお薦めします。

4.1.3.3 TIMESTAMP

TIMESTAMPデータ型は、DATEデータ型の拡張です。DATEデータ型の年、月、日の他、時、分、秒の値が格納されます。タイム・ゾーンはありません。TIMESTAMPデータ型の書式は次のようになります。

```
TIMESTAMP(fractional_seconds_precision)
```

fractional_seconds_precision(オプション)では、SECOND日時フィールドの小数部の桁数を指定します。桁数の範囲は0から9です。デフォルトは6です。

4.1.3.4 TIMESTAMPWITHTIMEZONE

TIMESTAMP WITH TIME ZONE (TSTZ)データ型はTIMESTAMPの改良型で、その値に明示的なタイム・ゾーン置換が含まれています。タイムゾーンによる時差は、地方時とUTC(協定世界時、旧称はグリニッジ標準時)との時差(時および分単位)です。TIMESTAMP WITH TIME ZONEデータ型の書式は次のとおりです。

```
TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE
```

fractional_seconds_precision(オプション)では、SECOND日時フィールドの小数部の桁数を指定します。桁数の範囲は0から9です。デフォルトは6です。

2つのTIMESTAMP WITH TIME ZONE値がUTCで同じ時刻を表す場合は、データに格納されたTIME ZONEオフセットにかかわらず、

同一であるとみなされます。

4.1.3.5 TIMESTAMPWITHLOCALTIMEZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)データ型はTIMESTAMPの別の改良型で、その値にタイム・ゾーン置換が含まれています。格納される値の形式はTIMESTAMPと同じです。このデータ型とTIMESTAMP WITH TIME ZONEとの違いは、データベースに格納されるデータがデータベース・タイム・ゾーンに正規化されること、およびタイム・ゾーン置換が列データの一部として格納されないことです。ユーザーがデータを取り出すと、そのユーザーのローカル・セッション・タイム・ゾーンでデータが戻されます。

タイムゾーンによる時差は、地方時とUTC(協定世界時、旧称はグリニッジ標準時)との時差(時および分単位)です。
TIMESTAMP WITH LOCAL TIME ZONEデータ型の書式は次のとおりです。

```
TIMESTAMP(fractional_seconds_precision) WITH LOCAL TIME ZONE
```

fractional_seconds_precision(オプション)では、SECOND日時フィールドの小数部の桁数を指定します。桁数の範囲は0から9です。デフォルトは6です。

4.1.3.6 INTERVALYEARTOMONTH

INTERVAL YEAR TO MONTHは、YEARおよびMONTH日時フィールドを使用して期間を格納します。INTERVAL YEAR TO MONTHデータ型の書式は次のようになります。

```
INTERVAL YEAR(year_precision) TO MONTH
```

year_precision(オプション)には、YEAR日時フィールドの桁数を指定します。*year_precision*のデフォルト値は2です。

4.1.3.7 INTERVALDAYTOSECOND

INTERVAL DAY TO SECONDデータ型には、日、時間、分および秒による期間が格納されます。INTERVAL DAY TO SECONDデータ型の書式は次のようになります。

```
INTERVAL DAY (day_precision) TO SECOND(fractional_seconds_precision)
```

各パラメータの意味は次のとおりです。

- *day_precision*は、DAY日時フィールドの桁数です。これはオプションです。0から9までの値を使用できます。デフォルトは2です。

*fractional_seconds_precision*は、SECOND日時フィールドの小数部の桁数です。これはオプションです。0から9までの値を使用できます。デフォルトは6です。

4.1.3.8 日時を使用した予期しない結果の回避

注意:

日時データの DML 操作で正しい結果を得るには、組込み SQL ファンクション DBTIMEZONE および SESSIONTIMEZONE で問い合せることによって、データベースおよびセッションのタイムゾーンを確認します。タイム・ゾーンが手動で設定されていない場合、デフォルトではオペレーティング・システムのタイム・ゾーンが使用されます。オペレーティング・システムのタイム・ゾーンが Oracle で有効なタイム・ゾーンではない場合、UTC がデフォルト値として使用されます。

4.2 ホスト変数

ホスト変数は、ホスト・プログラムとOracle間の通信に重要な役割を果します。通常、プリコンパイラ・プログラムがホスト変数からOracleにデータを入力し、Oracleがプログラム内のホスト変数にデータを出力します。Oracleは入力データをデータベース列に格納し、出力データをプログラムのホスト変数に格納します。

ホスト変数には、スカラー型として解決される任意のC言語の式を指定してもかまいません。ただし、ホスト変数も左辺値であることが必要です。ほとんどのホスト変数のホスト配列もサポートされています。

4.2.1 ホスト変数の宣言

ホスト変数は、Oracleプログラム・インタフェースでサポートされているC言語のデータ型を指定して、Cプログラム言語の規則に従って宣言します。このC言語のデータ型には、ソースまたはターゲットのデータベース列のデータ型との互換性が必要です。

MODE=ORACLEの場合、特別な宣言部でホスト変数を宣言する必要はありません。ただし、宣言部がANSI SQL標準の一部である場合に宣言部を使用しないと、FIPSフラガーで警告が発行されます。 CODE=CPP(C++コードのコンパイル中)、PARSE=NONEまたはPARSE=PARTIALの場合は、宣言部を使用する必要があります。

表4-4は、C言語のデータ型と、ホスト変数の宣言時に使用できる疑似型を示しています。ホスト変数に使用できるのは、これらのデータ型のみです。

表4-4 ホスト変数のC言語のデータ型

C言語のデータ型または疑似型	説明
char	単一文字
char[n]	n 文字配列(文字列)
int	整数
short	小さい整数
long	大きい整数
long long	きわめて大きい整数(8 バイト)
float	浮動小数点数(通常は単精度)
double	浮動小数点数(通常は倍精度)
VARCHAR[n]	可変長文字列

表4-5は、互換性のあるOracleの内部データ型を示しています。

表4-5 C言語とOracleのデータ型の互換性

内部型	C言語のデータ型	説明

内部型	C言語のデータ型	説明
VARCHAR2(Y)	char	単一文字
(注意 1)		
CHAR(X)	char[n]	n バイトの文字配列
(注意 1)	VARCHAR[n]	n バイトの可変長文字配列
	int	整数
	short	小さい整数
	long	大きい整数
	long long	きわめて大きい整数(8 バイト)
	float	浮動小数点数
	double	倍精度浮動小数点
		NUMBER
NUMBER	int	整数
NUMBER(P,S)	short	小さい整数
(注意 2)	int	整数
	long	大きい整数
	float	浮動小数点数
	double	倍精度浮動小数点
	char	NUMBER
	char[n]	単一文字
	VARCHAR[n]	n バイトの文字配列
		n バイトの可変長文字配列
DATE	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの可変長文字配列

内部型	C言語のデータ型	説明
LONG	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの可変長文字配列
RAW(X)	unsigned char[n]	n バイトの文字配列
(注意 1)	VARCHAR[n]	n バイトの可変長文字配列
LONG RAW	unsigned char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの可変長文字配列
ROWID	unsigned char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの可変長文字配列

注意:

1. X の範囲は 1 から 2000 で、デフォルト値は 1 です。Y の範囲は 1 から 4000 です。

2. P の範囲は 1 から 38 です。S の範囲は-84 から 127 です。

単純なC言語のデータ型の1次元配列は、ホスト変数としての役割も果します。char[n]およびVARCHAR[n]の場合、*n*には文字列の最大長を指定します(配列内にある文字列の数ではありません)。2次元配列を指定できるのは、char[m][n]およびVARCHAR[m][n]の場合のみです。*m*には配列内の文字列の数を指定し、*n*には文字列の最大長を指定します。

単純なC言語のデータ型へのポインタがサポートされています。char[n]およびVARCHAR[n]変数へのポインタは、charまたはVARCHAR(長さの指定なし)へのポインタとして宣言する必要があります。ただし、ポインタの配列はサポートされていません。

4.2.1.1 記憶域クラス指定子

Pro*C/C++では、ホスト変数の宣言時に**auto**、**extern**および**static**記憶域クラス指定子を使用できます。ただし、プリコンパイラはホスト変数の前にアンパサンド(&)を挿入することでアドレスを取得するため、**register**記憶域クラス指定子を使用してホスト変数を格納することはできません。C言語の規則に従うと、**auto**記憶域クラス指定子を使用できるのはブロック内のみです。

ANSI C規格に準拠するために、Pro*C/C++プリコンパイラでは次のように、最大長の指定の有無にかかわらず、**extern char[n]**ホスト変数を宣言できます。

```
extern char  protocol[15];
extern char  msg[];
```

ただし、最大長は常に指定する必要があります。前述の例で、あるプリコンパイル・ユニットで宣言された出力ホスト変数*msg*が、他のプリコンパイル・ユニットで定義された場合、プリコンパイラではその最大長を認識する方法がありません。2番目のプリコンパイル・ユニットの*msg*に十分な記憶域を割り当てないと、メモリーが破損することがあります。(十分な記憶域とは通常、ホスト変数にSELECTまたはFETCHされる可能性のある最長の列値のバイト数に、ヌル終端文字が付く場合に備えた1バイトを加えた値です。)

extern char[ ]ホスト変数の最大長を指定しない場合、プリコンパイラから警告メッセージが発行されます。また、ホスト変数にはCHARACTER列値が格納されるとみなされますが、この値の長さは255文字以内にする必要があります。したがって、長さ256文字以上のVARCHAR2またはLONG列値をホスト変数にSELECTまたはFETCHする場合は、最大長を指定する必要があります。

4.2.1.2 型修飾子

ホスト変数の宣言時には、**const**および**volatile**型修飾子も使用できます。

constホスト変数は定数を持つ必要があります。つまり、プログラムではその初期値を変更できません。**volatile**ホスト変数の値は、プログラムでは認識されない方法で(システムに接続されたデバイスなどにより)変更されることがあります。

4.2.2 ホスト変数の参照

ホスト変数は、SQL DML文で使用します。SQL文中ではホスト変数の先頭にコロン(:)を付ける必要がありますが、C言語の文中では先頭にコロンを付けないでください。次に例を示します。

```
char    buf[15];
int     emp_number;
float   salary;
...
gets(buf);
emp_number = atoi(buf);

EXEC SQL SELECT sal INTO :salary FROM emp
    WHERE empno = :emp_number;
```

わかりにくくなる可能性もありますが、次のように、ホスト変数にOracleの表または列と同じ名前を付けることもできます。

```
int     empno;
char    ename[10];
float   sal;
...
EXEC SQL SELECT ename, sal INTO :ename, :sal FROM emp
    WHERE empno = :empno;
```

4.2.2.1 制限事項

ホスト変数名はC識別子であるため、宣言および参照時には大/小文字区別を一致させる必要があります。ホスト変数は、列、表またはSQL文中の他のOracleオブジェクトでは置換できません。また、Oracle予約語も使用できません。

ホスト変数は、プログラムのアドレスに設定する必要があります。このため、関数コールおよび数値式はホスト変数に指定できません。次のコードは無効です。

```
#define MAX_EMP_NUM    9000
...
int get_dept();
...
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (:MAX_EMP_NUM + 10, 'CHEN', :get_dept());
```

関連項目

- 予約語、キーワードおよびネームスペース

4.3 標識変数

ホスト変数はすべて、オプションのインジケータ変数に関連付けることができます。標識変数は、2バイトの整数として定義する必要があります。また、SQL文中では、標識変数の前にコロンを付けてホスト変数の直後に指定する必要があります(INDICATORキーワードを使用しない場合)。宣言部を使用する場合は、宣言部中でも標識変数を宣言する必要があります。

これはリレーショナル列に適用されるもので、オブジェクト型には適用されません。

関連項目

- オブジェクト

4.3.1 INDICATORキーワード

判読しやすくするため、それぞれの標識変数の前にINDICATORのオプションのキーワードを置くこともできます。その場合も、標識変数の前にはコロンを付ける必要があります。正しい構文は、次のとおりです。

```
:host_variable INDICATOR :indicator_variable
```

これは次の構文と同じ意味です。

```
:host_variable:indicator_variable
```

ホスト・プログラムでは、両方の形式の式を使用できます。

可能なインジケータの値とその意味は、次のとおりです。

インジケータの値	意味
0	操作は成功しました。
-1	NULL が戻されたか、挿入または更新されました。
-2	LONG 型の文字ホスト変数の出力は、長すぎる部分が切り捨てられましたが、元の列の長さは判別できません。
>0	SELECT または FETCH の結果として文字ホスト変数に入ったデータは、長すぎる文が切り捨てられました。この場合、そのホスト変数がマルチバイト・キャラクタ変数であれば、インジケータの値は元の列の長さを文字数で表したものになります。そのホスト変数がマルチバイト・キャラクタ変数でなければ、インジケータの長さは元の列の長さをバイト数で表したものになります。

4.3.2 標識変数の使用例

通常、標識変数は、入力ホスト変数へのNULLの割当てと、出力ホスト変数に含まれるNULLまたは切捨て値の検出に使用されます。次の例では、3つのホスト変数と1つの標識変数を宣言してから、SELECT文を使用して、ホスト変数

*emp_number*の値と一致する従業員番号をデータベース内で検索します。一致する行が見つかると、出力ホスト変数*salary*および*commission*が、その行の列SALおよびCOMMの値に設定され、リターン・コードが標識変数*ind_comm*に格納されます。後継の文では、*ind_comm*を使用してその後の処理を選択しています。

```
EXEC SQL BEGIN DECLARE SECTION;
    int    emp_number;
    float  salary, commission;
    short comm_ind;  /* indicator variable  */
EXEC SQL END DECLARE SECTION;
    char temp[16];
    float  pay;      /* not used in a SQL statement */
...
printf("Employee number? ");
gets(temp);
emp_number = atof(temp);
EXEC SQL SELECT SAL, COMM
    INTO :salary, :commission:ind_comm
    FROM EMP
    WHERE EMPNO = :emp_number;
if(ind_comm == -1)    /* commission is null */
    pay = salary;
else
    pay = salary + commission;
```

関連項目

- 標識変数

4.3.3 標識変数のガイドライン

標識変数の宣言および参照については、次のガイドラインに従ってください。標識変数は、次のようにする必要があります。

- 2バイトの整数として(宣言部がある場合はそこに)明示的に宣言します。
- SQL文中では前にコロン(:)を付けます。
- SQL文中とPL/SQLブロック内では、そのホスト変数の直後に指定します(INDICATORキーワードを前に付ける場合は除きます)。

標識変数では、次のことをしないでください。

- ホスト言語文中で、前にコロン(:)を付けないでください。
- ホスト言語文中で、そのホスト変数の直後に指定しないでください。
- Oracleの予約語にしないでください。

4.3.4 Oracle制限事項

DBMS=V7またはV8の場合は、標識変数に対応付けられていないホスト変数にNULLをSELECTまたはFETCHすると、次のエラー・メッセージが発行されます。

```
ORA-01405: fetched column value is NULL
```

MODE=ORACLEおよびDBMS=V7またはV8を指定してプリコンパイルする場合は、UNSAFE_NULL=YESを指定してORA-01405メッセージを使用禁止にできます。

関連項目

- UNSAFE_NULL

4.4 VARCHAR変数

VARCHAR疑似型を使用すると、可変長文字列を宣言できます。プログラムで扱う文字列がVARCHAR2列またはLONG列からの出力、あるいはその列への入力である場合、標準のC言語文字列のかわりにVARCHARホスト変数を使用する方が便利なこともあります。データ型名VARCHARは、すべて大文字でも、すべて小文字でもかまいませんが、大文字と小文字を混在させることはできません。このマニュアルでは、VARCHARがC言語固有のデータ型とは異なることを強調するために、大文字を使用しています。

4.4.1 VARCHAR変数の宣言

VARCHARは、C言語の拡張型または宣言済の**構造体**と考えてください。たとえば、次のVARCHAR宣言があると仮定します。

```
VARCHAR   username[20];
```

プリコンパイラは上の宣言を、配列メンバーおよび長さメンバーを持つ次の**構造体**に展開します。

```
struct
{
    unsigned short  len;
    unsigned char   arr[20];
} username;
```

VARCHAR変数の利点は、SELECTまたはFETCHの後にVARCHAR構造体の長さメンバーを明示的に参照できることです。Oracleでは、選択された文字列の長さが長さメンバーに置かれます。このメンバーを、ヌル終端文字('¥0')の追加などに使用できます。

```
username.arr[username.len] = '¥0';
```

また、次のように、長さは*strncpy*文または*printf*文でも指定できます。

```
printf("Username is %.*s¥n", username.len, username.arr);
```

VARCHAR変数の最大長は、その宣言内で指定します。長さの範囲は、1から65533にする必要があります。たとえば、次の宣言は、長さが指定されていないため無効です。

```
VARCHAR   null_string[];    /* invalid */
```

長さの指定では、配列メンバーに格納される値の現行の長さが保持されます。

次のように、1行に複数のVARCHARを宣言できます。

```
VARCHAR emp_name[ENAME_LEN], dept_loc[DEPT_NAME_LEN];
```

VARCHARの長さ指定子には、**#define**による定義済マクロか、プリコンパイル時に整数で解決できる任意の複合式を使用できます。

また、VARCHARデータ型へのポインタも宣言できます。

注意:

次のような typedef 文は使用しないでください。

```
typedef VARCHAR buf[64];
```

このような文を使用すると、C コンパイル・エラーが発生します。

関連項目

- VARCHAR変数およびポインタ

4.4.2 VARCHAR変数の参照

SQL文では、次のように、接頭辞としてコロンを付けた**構造体**名を使用してVARCHAR変数を参照します。

```
...
int       part_number;
VARCHAR   part_desc[40];
...
main()
{
    ...
    EXEC SQL SELECT pdesc INTO :part_desc
        FROM parts
        WHERE pnum = :part_number;
    ...
```

問合せが実行された後、データベースから取り出され、*part_desc.arr*に格納された文字列の実際の長さが*part_desc.len*に保持されます。

C言語の文では、次のように、コンポーネント名を使用してVARCHAR変数を参照します。

```
printf("¥n¥nEnter part description: ");
gets(part_desc.arr);
/* You must set the length of the string
   before using the VARCHAR in an INSERT or UPDATE */
part_desc.len = strlen(part_desc.arr);
```

4.4.3 VARCHAR変数にNULLを戻す

Oracleでは、VARCHAR出力ホスト変数の長さコンポーネントが自動的に設定されます。SELECTまたはFETCHでVARCHAR変数にNULLを入れた場合、サーバーでは長さメンバーも配列メンバーも変更されません。

注意:

NULL を VARCHAR ホスト変数に選択した場合に、対応する標識変数がないと、実行時に ORA-01405 エラーが発生します。これを回避するには、すべてのホスト変数に対応して標識変数を記述します。(一時修正としては、UNSAFE_NULL=YES プリコンパイラ・オプションを使用します。

関連項目

- DBMS

4.4.4 VARCHAR変数を使用したNULLの挿入

VARCHAR変数の長さを0に設定してからUPDATE文またはINSERT文を実行すると、列値はNULLに設定されます。列にNOT NULL制約がある場合は、エラーが戻されます。

4.4.5 関数へのVARCHAR変数の受渡し

VARCHARは構造体であり、ほとんどのCコンパイラでは、構造体を値で関数に渡すこと、および関数からコピーで構造体を戻すことが可能です。ただし、Pro*C/C++では、VARCHARを参照により関数に渡す必要があります。次の例は、VARCHAR変数を関数に渡す正しい方法を示しています。

```
VARCHAR emp_name[20];
...
emp_name.len = 20;
SELECT ename INTO :emp_name FROM emp
WHERE empno = 7499;
...
print_employee_name(&emp_name); /* pass by pointer */
...

print_employee_name(name)
VARCHAR *name;
{
    ...
    printf("name is %.*s¥n", name->len, name->arr);
    ...
}
```

4.4.6 VARCHAR配列コンポーネントの長さを調べる方法

プリコンパイラでVARCHAR宣言が処理されると、生成される構造体内の配列要素の実際の長さが、宣言された長さよりも長くなることがあります。たとえば、Sun Solarisシステムで次のようなPro*C/C++宣言があるとします。

```
VARCHAR my_varchar[12];
```

この宣言は、プリコンパイラにより次のように展開されます。

```
struct my_varchar
{
    unsigned short len;
    unsigned char  arr[12];
};
```

しかし、このシステムのプリコンパイラまたはCコンパイラは、配列コンポーネントの長さをパディングして14バイトにします。このアラインメント要件は、構造体全体の長さをパディングして16バイトにします。パディングされた配列が14バイト、長さが2バイトです。

SQLVarcharGetLength()(非スレッドsqlvcp()と置換)関数(SQLLIBランタイム・ライブラリの一部)により、配列メンバーの実際の(パディングされている場合もあります)長さが戻されます。

SQLVarcharGetLength()関数には、VARCHARホスト変数またはVARCHARポインタ・ホスト変数のデータの長さを渡します。SQLVarcharGetLength()からは、VARCHARの配列コンポーネント合計の長さが戻されます。合計の長さには、使用しているCコンパイラにより追加された可能性のあるパディングが含まれます。

SQLVarcharGetLength()の構文は、次のとおりです。

```
SQLVarcharGetLength (dvoid *context, unsigned long *datlen, unsigned long *totlen);
```

シングル・スレッド・アプリケーションの場合は、sqlvcp()を使用してください。VARCHARの長さをdatlenパラメータに指定してから、sqlvcp()をコールします。関数の戻り時には、totlenパラメータに配列要素の合計の長さが設定されています。どちらのパラメータもunsigned long整数へのポインタであるため、参照で渡す必要があります。

関連項目

- SQLLIBパブリック関数の新しい名前

4.4.7 サンプル・プログラム: sqlvcp()の使用

次のサンプル・プログラムは、Pro*C/C++アプリケーションでの関数の使用方法を示しています。また、このサンプル・プログラムではsqlgls()関数も使用しています。このサンプルでは、まずVARCHARポインタを宣言してから、sqlvcp()関数を使用してVARCHARバッファに必要なサイズを決定します。プログラムでは、EMP表から従業員名がFETCHされて出力されます。最後に、sqlgls()関数を使用してSQL文とその関数コード、長さの属性が出力されます。このプログラムは、demoディレクトリのsqlvcp.pcとしてオンラインで使用できます。

```
/*
 *  The sqlvcp.pc program demonstrates how you can use the
 *  sqlvcp() function to determine the actual size of a
 *  VARCHAR struct.  The size is then used as an offset to
 *  increment a pointer that steps through an array of
 *  VARCHARs.
 *
 *  This program also demonstrates the use of the sqlgls()
 *  function, to get the text of the last SQL statement executed.
 *  sqlgls() is described in the "Error Handling" chapter of
 *  The Programmer's Guide to the Oracle Pro*C/C++ Precompiler.
 */

#include <stdio.h>
#include <sqlca.h>
#include <sqlcpr.h>

/*  Fake a VARCHAR pointer type.  */

struct my_vc_ptr
{
    unsigned short len;
    unsigned char arr[32767];
};

/* Define a type for the VARCHAR pointer */
typedef struct my_vc_ptr my_vc_ptr;
my_vc_ptr *vc_ptr;


EXEC SQL BEGIN DECLARE SECTION;
VARCHAR *names;
int      limit;    /* for use in FETCH FOR clause  */
char    *username = "scott/tiger";
EXEC SQL END DECLARE SECTION;
void sql_error();
extern void sqlvcp(), sqlgls();

main()
{
    unsigned int vcplen, function_code, padlen, buflen;
    int i;
```

```
    char stmt_buf[120];

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    EXEC SQL CONNECT :username;
    printf("¥nConnected.¥n");

/*  Find number of rows in table. */
    EXEC SQL SELECT COUNT(*) INTO :limit FROM emp;


/*  Declare a cursor for the FETCH statement. */
    EXEC SQL DECLARE emp_name_cursor CURSOR FOR
    SELECT ename FROM emp;
    EXEC SQL FOR :limit OPEN emp_name_cursor;

/*  Set the desired DATA length for the VARCHAR. */
    vcplen = 10;

/*  Use SQLVCP to help find the length to malloc. */
    sqlvcp(&vcplen, &padlen);
    printf("Actual array length of VARCHAR is %ld¥n", padlen);

/*  Allocate the names buffer for names.
    Set the limit variable for the FOR clause. */
    names = (VARCHAR *) malloc((sizeof (short) +
    (int) padlen) * limit);
    if (names == 0)
    {
        printf("Memory allocation error.¥n");
        exit(1);
    }
/*  Set the maximum lengths before the FETCH.
 *  Note the "trick" to get an effective VARCHAR *.
 */
    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)
    {
        vc_ptr->len = (short) padlen;
        vc_ptr = (my_vc_ptr *)((char *) vc_ptr +
        padlen + sizeof (short));
    }
/*  Execute the FETCH. */
    EXEC SQL FOR :limit FETCH emp_name_cursor INTO :names;

/*  Print the results. */
    printf("Employee names--¥n");

    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)
    {
        printf
         ("%.*s¥t(%d)¥n", vc_ptr->len, vc_ptr->arr, vc_ptr->len);
        vc_ptr = (my_vc_ptr *)((char *) vc_ptr +
                  padlen + sizeof (short));
    }

/*  Get statistics about the most recent
 *  SQL statement using SQLGLS. Note that
 *  the most recent statement in this example
 *  is not a FETCH, but rather "SELECT ENAME FROM EMP"
 *  (the cursor).
```

```
 */
    buflen = (long) sizeof (stmt_buf);

/*  The returned value should be 1, indicating no error. */
    sqlgls(stmt_buf, &buflen, &function_code);
    if (buflen != 0)
    {
        /* Print out the SQL statement. */
        printf("The SQL statement was--¥n%.*s¥n", buflen, stmt_buf);

        /* Print the returned length. */
        printf("The statement length is %ld¥n", buflen);

        /* Print the attributes. */
        printf("The function code is %ld¥n", function_code);

        EXEC SQL COMMIT RELEASE;
        exit(0);
    }
    else
    {
        printf("The SQLGLS function returned an error.¥n");
        EXEC SQL ROLLBACK RELEASE;
        exit(1);
    }
}

void
sql_error()
{
    char err_msg[512];
    int buf_len, msg_len;


    EXEC SQL WHENEVER SQLERROR CONTINUE;

    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s¥n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

関連項目

- ランタイム・エラーの処理

4.5 カーソル変数

Pro*C/C++プログラムでは、問合せにカーソル変数を使用できます。カーソル変数は、OracleサーバーでPL/SQLを使用して定義し、オープンする必要のあるカーソルのハンドルです。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。

カーソル変数の利点は、次のとおりです。

- メンテナンスのしやすさ

  問合せは、カーソル変数をオープンするストアド・プロシージャに集中されます。カーソルを変更する必要がある場合は、ストアド・プロシージャの変更のみで済みます。各アプリケーションを変更する必要はありません。

- 便利なセキュリティ機能

  アプリケーションのユーザーは、Pro*C/C++アプリケーションからサーバーへの接続時に使用されるユーザー名です。ユーザーには、カーソルをオープンするストアド・プロシージャに対する実行権限が必要ですが、問合せに使用する表に対する読取り権限は不要です。このセキュリティ機能を使用して、表内の列や、他のストアド・プロシージャへのアクセスを制限できます。

4.5.1 カーソル変数の宣言

Pro*C/C++のSQL_CURSOR疑似型を使用して、Pro*C/C++プログラム内でカーソル変数を宣言します。次に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
    sql_cursor     emp_cursor;             /* a cursor variable */
    SQL_CURSOR     dept_cursor;      /* another cursor variable */
    sql_cursor     *ecp;       /* a pointer to a cursor variable */
    ...
EXEC SQL END DECLARE SECTION;
ecp = &emp_cursor;              /* assign a value to the pointer */
```

カーソル変数を宣言するときに、型指定にはすべて大文字のSQL_CURSOR、あるいはすべて小文字のsql_cursorを使用できます。大文字と小文字の組合せは使用できません。

カーソル変数は、Pro*C/C++プログラム内の他のホスト変数と同様です。カーソル変数にはスコープがあり、C言語のスコープ規則に従っています。カーソル変数は、他の関数にパラメータとして渡すことができ、カーソル変数を宣言しているソース・ファイルの外部にある関数にも渡すことができます。また、カーソル変数を戻す関数のみでなく、カーソル変数へのポインタを戻す関数も定義できます。

> 注意:
>
> SQL_CURSOR は、C 言語の**構造体**として Pro*C/C++で生成されるコードに実装されます。したがって、常にポインタを使用して他の関数に渡したり、関数からカーソル変数へのポインタを戻したりすることができます。ただし、SQL_CURSOR を値で渡す、あるいは戻すことができるのは、C コンパイラでその操作がサポートされている場合のみです。

4.5.2 カーソル変数の割当て

カーソル変数をオープンするか、カーソル変数を使用してFETCHする前に、カーソルを割り当てる必要があります。そのためには、新しいプリコンパイラ・コマンドALLOCATEを使用します。たとえば、前述の例で宣言したSQL_CURSOR *emp_cursor*を割り当てるには、次の文を記述します。

```
EXEC SQL ALLOCATE :emp_cursor;
```

カーソルの割当てには、プリコンパイル時も実行時もサーバーのコールは必要ありません。ALLOCATE文に誤り(未宣言のホスト変数など)があると、Pro*C/C++ではプリコンパイル時エラーが発行されます。カーソル変数を割り当てると、ヒープ・メモリーが使用されます。このため、プログラム・ループ内でカーソルを解放できます。カーソル変数に割り当てられるメモリーは、カーソルがクローズされるときには解放されず、明示的なCLOSEが実行されるか、接続がクローズされるときにのみ解放されます。

```
EXEC SQL CLOSE :emp_cursor;
```

関連項目

- カーソル変数のクローズと解放

4.5.3 カーソル変数のオープン

カーソル変数はOracleデータベース・サーバー上でオープンする必要があります。埋込みSQL OPENコマンドを使用しても、カーソル変数はオープンできません。カーソル変数をオープンするには、カーソルをオープン(および同じ文中で定義)するPL/SQLストアド・プロシージャをコールする方法があります。または、Pro*C/C++プログラム内で無名PL/SQLブロックを使用してカーソル変数をオープンし、定義する方法があります。

たとえば、次のPL/SQLパッケージがデータベースに格納されているとします。

```
CREATE PACKAGE demo_cur_pkg AS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE cur_type IS REF CURSOR RETURN EmpName;
    PROCEDURE open_emp_cur (
                curs     IN OUT cur_type,
                dept_num IN     NUMBER);
END;

CREATE PACKAGE BODY demo_cur_pkg AS
    CREATE PROCEDURE open_emp_cur (
                curs     IN OUT cur_type,
                dept_num IN     NUMBER) IS
    BEGIN
        OPEN curs FOR
            SELECT ename FROM emp
                WHERE deptno = dept_num
                ORDER BY ename ASC;
    END;
END;
```

このパッケージが格納された後で、Pro*C/C++プログラムから*open_emp_cur*ストアド・プロシージャをコールしてカーソル*curs*をオープンし、プログラム内のカーソルからFETCHできます。次に例を示します。

```
...
sql_cursor    emp_cursor;
char          emp_name[11];
...
EXEC SQL ALLOCATE :emp_cursor;  /* allocate the cursor variable */
...
/* Open the cursor on the server side. */
EXEC SQL EXECUTE
    begin
        demo_cur_pkg.open_emp_cur(:emp_cursor, :dept_num);
    end;
;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH :emp_cursor INTO :emp_name;
    printf("%s¥n", emp_name);
}
...
```

Pro*C/C++プログラム内で無名PL/SQLブロックを使用してカーソルをオープンするには、無名ブロック内でカーソルを定義します。次に例を示します。

```
sql_cursor emp_cursor;
int dept_num = 10;
...
EXEC SQL EXECUTE
    BEGIN
        OPEN :emp_cursor FOR SELECT ename FROM emp
             WHERE deptno = :dept_num;
    END;
END-EXEC;
...
```

前述の各例は、PL/SQLを使用してカーソル変数をオープンする方法を示しています。次のように、埋込みSQL文でCURSOR句を使用してカーソル変数をオープンすることもできます。

```
...
sql_cursor emp_cursor;
...
EXEC ORACLE OPTION(select_error=no);
EXEC SQL
    SELECT CURSOR(SELECT ename FROM emp WHERE deptno = :dept_num)
    INTO :emp_cursor FROM DUAL;
EXEC ORACLE OPTION(select_error=yes);
```

前述の文では、emp_cursorカーソル変数が最も外側のSELECTの最初の列にバインドされています。最初の列は、それ自体が問合せですが、CURSOR(...)変換句が指定されているため、sql_cursorホスト変数と互換性のある形式で表されています。

CURSOR句を含む問合せを使用する場合は、SELECT_ERRORオプションをNOに設定する必要があります。これにより、親カーソルの取消しが禁止され、プログラムをエラーなしで実行できます。

4.5.3.1 スタンドアロン・ストアド・プロシージャでのオープン

前述の例では、参照カーソルがパッケージ内で定義され、カーソルはそのパッケージ内のプロシージャ内でオープンされました。しかし、参照カーソルは、カーソルをオープンするプロシージャと同じパッケージ内で定義しなくてもかまいません。

スタンドアロン・ストアド・プロシージャ内でカーソルをオープンする必要がある場合は、別のパッケージ内にカーソルを定義し、カーソルをオープンするスタンドアロン・ストアド・プロシージャ内で、そのパッケージを参照できます。次はその例です。

```
PACKAGE dummy IS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE emp_cursor_type IS REF CURSOR RETURN EmpName;
END;
-- and then define a standalone procedure:
PROCEDURE open_emp_curs (
      emp_cursor IN OUT dummy.emp_cursor_type;
      dept_num   IN     NUMBER) IS
    BEGIN
        OPEN emp_cursor FOR
            SELECT ename FROM emp WHERE deptno = dept_num;
    END;
END;
```

4.5.3.2 戻り型

PL/SQLストアド・プロシージャ内に参照カーソルを定義する場合は、カーソルが戻す値の型を宣言する必要があります。

関連項目

- カーソル変数の宣言

4.5.4 カーソル変数のクローズと解放

カーソル変数をクローズするには、CLOSEコマンドを使用します。たとえば、前述の例でOPENした*emp_cursor*カーソル変数をクローズするには、埋込みSQL文を使用します。

```
EXEC SQL CLOSE :emp_cursor;
```

カーソル変数はホスト変数であるため、前にコロンを付ける必要があります。

ALLOCATEで割り当てたカーソル変数を再利用できます。アプリケーションで必要な回数だけオープン、FETCHおよびCLOSEできます。ただし、サーバーから切断してから再接続する場合は、カーソル変数をALLOCATEで再度割り当てる必要があります。

カーソルは、FREE埋込みSQL文で割当て解除されます。次に例を示します。

```
EXEC SQL FREE :emp_cursor;
```

オープンしているカーソルはクローズされ、割り当てられたメモリーは解放されます。

4.5.5 OCIでのカーソル変数の使用(リリース7のみ)

Pro*C/C++のカーソル変数をOCI関数と共有できます。そのためには、SQLLIB変換関数SQLCDAFromResultSetCursor() (以前のsqlcdat())およびSQLCDAToResultSetCursor(以前のsqlcurt())を使用する必要があります。これらの関数を使用して、OCIのカーソル・データ領域をPro*C/C++のカーソル変数に変換します。

SQLCDAFromResultSetCursor()関数では、割当て済のカーソル変数がOCIカーソル・データ領域に変換されます。次に構文を示します。

```
void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cur,
   sword *retval);
```

パラメータは次のとおりです。

パラメータ	説明
context	SQLLIB ランタイム・コンテキストへのポインタ
cda	変換先の OCI カーソル・データ領域へのポインタ
cur	変換元の Pro*C/C++カーソル変数へのポインタ
retval	エラーがなければ 0、それ以外の場合は SQLLIB(SQL)のエラー番号

注意:

エラーの場合には、CDA の *V2* および *rc* リターン・コード・フィールドもエラー・コードを受け取ります。CDA の処理済行数フィールドは設定されません。

非スレッドまたはデフォルト・コンテキスト・アプリケーションでは、定義した定数 SQL_SINGLE_RCTX をコンテキ

ストとして渡してください。

SQLCDAToResultSetCursor()関数では、OCIのカーソル・データ領域がPro*C/C++のカーソル変数に変換されます。次に構文を示します。

```
void SQLCDAToResultSetCursor(dvoid *context, void *cur, Cda_Def *cda,
   int *retval);
```

パラメータは次のとおりです。

パラメータ	説明
context	SQLLIB ランタイム・コンテキストへのポインタ
cur	変換先の Pro*C/C++カーソル変数へのポインタ
cda	変換元の OCI カーソル・データ領域へのポインタ
retval	エラーがなければ 0、それ以外の場合はエラー・コード

注意:

SQLCA 構造体は、このルーチンでは更新されません。SQLCA コンポーネントが設定されるのは、変換されたカーソルを使用してデータベース操作が実行された後のみです。

非スレッド・アプリケーションでは、定義した定数 SQL_SINGLE_RCTX をコンテキストとして渡してください。

これらの関数に対するANSIとK&Rのプロトタイプは、sql2oci.hヘッダー・ファイルに用意されています。これらの関数をコールする前に、*cda*および*cur*のメモリーを割り当てる必要があります。

関連項目

- SQLLIBパブリック関数の新しい名前

4.5.6 制限(カーソル変数)

カーソル変数の使用には、次の制限が適用されます。

- Pro*C/C++とOCI V7で同じカーソル変数を使用する場合は、接続直後にSQLLDAGetCurrent()またはSQLLDAGetName()を使用する必要があります。
- カーソル変数をOCIリリース8で対応するものには変換できません。
- カーソル変数は動的SQLでは使用できません。
- カーソル変数を使用できるコマンドは、ALLOCATE、FETCH、FREEおよびCLOSEのみです。
- DECLARE CURSORコマンドは、カーソル変数には適用されません。

- CLOSEでクローズしたカーソル変数からのFETCHはできません。
- ALLOCATEで割り当てられていないカーソル変数からのFETCHはできません。
- MODE=ANSIを指定してプリコンパイルする場合、すでにクローズ済のカーソル変数をクローズするとエラーになります。
- AT句は、ALLOCATEコマンド、カーソル変数を参照するFETCHコマンドおよびCLOSEコマンドには使用できません。
- カーソル変数はデータベースの列に格納できません。
- カーソル変数自体は、パッケージ指定内で宣言できません。パッケージ指定内で宣言できるのは、カーソル変数の型のみです。
- カーソル変数は、PL/SQLレコードのコンポーネントにはできません。

4.5.7 例: cv_demo.sqlおよびsample11.pc

次のサンプル・プログラム(PL/SQLスクリプトとPro*C/C++プログラム)は、カーソル変数の使用方法を示しています。これらのソースはdemoディレクトリにあり、オンラインで使用できます。demoディレクトリにある同じアプリケーションの別バージョンcv_demo.pcも参照してください。

4.5.7.1 cv_demo.sql

```
-- PL/SQL source for a package that declares and
-- opens a ref cursor
CONNECT SCOTT/TIGER;
CREATE OR REPLACE PACKAGE emp_demo_pkg as
   TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
     PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER);
END emp_demo_pkg;


CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS
    PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER) IS
    BEGIN
        OPEN curs FOR SELECT *
            FROM emp WHERE deptno = dno
            ORDER BY ename ASC;
    END;
END emp_demo_pkg;
```

4.5.7.2 sample11.pc

```
/*
 *  Fetch from the EMP table, using a cursor variable.
 *  The cursor is opened in the stored PL/SQL procedure
 *  open_cur, in the EMP_DEMO_PKG package.
 *
 *  This package is available on-line in the file
 *  sample11.sql, in the demo directory.
 *
 */

#include <stdio.h>
#include <sqlca.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
```

```
/* Error handling function. */
void sql_error(msg)
    char *msg;
{
    size_t clen, fc;
    char cbuf[128];

    clen = sizeof (cbuf);
    sqlgls((char *)cbuf, (size_t *)&clen, (size_t *)&fc);

    printf("¥n%s¥n", msg);
    printf("Statement is--¥n%s¥n", cbuf);
    printf("Function code is %ld¥n¥n", fc);

    sqlglm((char *)cbuf, (size_t *) &clen, (size_t *) &clen);
    printf ("¥n%.*s¥n", clen, cbuf);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
}

void main()
{
    char temp[32];

    EXEC SQL BEGIN DECLARE SECTION;
        char *uid = "scott/tiger";
        SQL_CURSOR emp_cursor;
        int dept_num;
        struct
        {
            int   emp_num;
            char  emp_name[11];
            char  job[10];
            int   manager;
            char  hire_date[10];
            float salary;
            float commission;
            int   dept_num;
        } emp_info;

        struct
        {
            short emp_num_ind;
            short emp_name_ind;
            short job_ind;
            short manager_ind;
            short hire_date_ind;
            short salary_ind;
            short commission_ind;
            short dept_num_ind;
        } emp_info_ind;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR do sql_error("Oracle error");

/* Connect to Oracle. */
    EXEC SQL CONNECT :uid;
```

```
/* Allocate the cursor variable. */
    EXEC SQL ALLOCATE :emp_cursor;

/* Exit the inner for (;;) loop when NO DATA FOUND. */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        printf("¥nEnter department number  (0 to exit): ");
        gets(temp);
        dept_num = atoi(temp);
        if (dept_num <= 0)
            break;

        EXEC SQL EXECUTE
            begin
                emp_demo_pkg.open_cur(:emp_cursor, :dept_num);
            end;
        END-EXEC;

        printf("¥nFor department %d--¥n", dept_num);
        printf("ENAME           SAL     COMM¥n");
        printf("-----           ---     ----¥n");

/* Fetch each row in the EMP table into the data struct.
   Note the use of a parallel indicator struct. */
        for (;;)
        {
             EXEC SQL FETCH :emp_cursor
                 INTO :emp_info INDICATOR :emp_info_ind;

             printf("%s ", emp_info.emp_name);
             printf("%8.2f ", emp_info.salary);
             if (emp_info_ind.commission_ind != 0)
                 printf("    NULL¥n");
             else
                 printf("%8.2f¥n", emp_info.commission);
        }

    }

/* Close the cursor. */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL CLOSE :emp_cursor;

/* Disconnect from Oracle. */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_SUCCESS);

}
```

4.6 コンテキスト変数

ランタイム・コンテキストは、通常は単にコンテキストと呼ばれ、クライアント・メモリーの領域へのハンドルです。このクライアント・メモリーには、0個以上の接続、0個以上のカーソル、そのインライン・オプション(MODE、HOLD_CURSOR、RELEASE_CURSOR、SELECT_ERRORなど)および他の状態情報が含まれます。

コンテキスト・ホスト変数を定義するには、*sql_context*疑似型を使用します。次に例を示します。

```
sql_context my_context ;
```

CONTEXT ALLOCATEプリコンパイラ・ディレクティブを使用し、コンテキスト用のメモリーを割り当てて初期化します。

```
EXEC SQL CONTEXT ALLOCATE :context ;
```

contextは、コンテキストへのハンドルであるホスト変数です。次に例を示します。

```
EXEC SQL CONTEXT ALLOCATE :my_context ;
```

CONTEXT USEプリコンパイラ・ディレクティブを使用して、プログラム・ロジックの流れではなく、ソース・ファイルのその点から埋込みSQL文(CONNECT、INSERT、DECLARE CURSORなど)で使用するコンテキストを定義します。このコンテキストは、別のCONTEXT USE文の検出時まで使用されます。次に構文を示します。

```
EXEC SQL CONTEXT USE {:context | DEFAULT} ;
```

DEFAULTキーワードでは、以降に実行されるすべての埋込みSQL文で使用されるデフォルト(またはグローバル)・コンテキストが指定されます。このコンテキストは、別のCONTEXT USEディレクティブが検出されるまで使用されます。単純な例を次に示します。

```
EXEC SQL CONTEXT USE :my_context ;
```

コンテキスト変数my_contextが定義および割り当てられていない場合、エラーが戻されます。

CONTEXT FREE文を使用すると、コンテキストに使用されたメモリーが不要になった場合に、メモリーを解放できます。

```
EXEC SQL CONTEXT FREE :context ;
```

次に例を示します。

```
EXEC SQL CONTEXT FREE :my_context ;
```

次の例は、ユーザー定義コンテキストと同じアプリケーションにおけるデフォルト・コンテキストの使用方法を示しています。

CONTEXT USEの例

```
#include <sqlca.h>
#include <ociextp.h>
main()
{
   sql_context ctx1;
   char *usr1 = "scott/tiger";
   char *usr2 = "system/manager";

   /* Establish connection to SCOTT in global runtime context */
   EXEC SQL CONNECT :usr1;

   /* Establish connection to SYSTEM in runtime context ctx1 */
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL CONNECT :usr2;

   /* Insert into the emp table from schema SCOTT */
   EXEC SQL CONTEXT USE DEFAULT;
   EXEC SQL INSERT INTO emp (empno, ename) VALUES (1234, 'WALKER');
   ...
}
```

4.7 ユニバーサルROWID

データベース・サーバーでは、ヒープ表および索引構成表という2種類の表編成が使用されています。

デフォルトでは、ヒープ表が使用されます。物理的な行アドレス(ROWID)は、ヒープ表の行を識別するためのパーマネント・プロパティです。物理ROWIDの外部文字書式は、ベース64でエンコーディングした18バイトの文字列です。

索引構成表には、永続識別子としての物理行アドレスはありません。これらの表には、論理ROWIDが定義されています。索引構成表からのSELECT ROWID...文を使用するとき、ROWIDは、表の主キー、制御情報、および物理的な推測を含む不透明な構造です。表から値を検索するために、「WHERE ROWID = ...」などの句を含むSQL文でこのROWIDを使用できます。

ユニバーサルROWIDは、物理ROWIDと論理ROWIDの両方で使用できます。表編成の変更はアプリケーションに影響しないため、ユニバーサルROWIDを使用すると、ヒープ表または索引構成表のデータにアクセスできます。ROWIDに使用される列データ型はUROWID(*length*)で、lengthはオプションです。

新しいアプリケーションでは、ユニバーサルROWIDを使用してください。

ユニバーサルROWID変数の使用方法は、次のとおりです。

- OCIROWIDへのタイプ・ポインタとして宣言します。
- ユニバーサルROWID変数用のメモリーを割り当てます。
- ユニバーサルROWIDをホスト・バインド変数として使用します。
- 終了時にメモリーを解放します。

次に例を示します。

```
OCIRowid *my_urowid ;
...
EXEC SQL ALLOCATE :my_urowid ;
/* Bind my_urowid as type SQLT_RDD -- no implicit conversion */
EXEC SQL SELECT rowid INTO :my_urowid FROM my_table WHERE ... ;
...
EXEC SQL UPDATE my_table SET ... WHERE rowid = :my_urowid ;
EXEC SQL FREE my_urpwid ;
...
```

また、19(18バイトとヌル終端文字)から4001の幅を持つ文字ホスト変数を、ユニバーサルROWIDのホスト・バインド変数として使用することもできます。文字ベースのユニバーサルROWIDはヒープ表でもサポートされていますが、下位互換性しかありません。ユニバーサルROWIDは可変長であるため、切り捨てられる場合があります。

文字変数の使用方法は、次のとおりです。

```
/* n is based on table characteristics  */
int n=4001 ;
char my_urowid_char[n] ;
...
EXEC SQL ALLOCATE :my_urowid_char ;
/* Bind my_urowid_char as SQLT_STR  */
EXEC SQL SELECT rowid INTO :my_urowid_char FROM my_table WHERE ... ;
EXEC ORACLE OPTION(CHAR_MAP=STRING);
EXEC SQL UPDATE my_table SET ... WHERE rowid = :my_urowid_char ;
EXEC SQL FREE :my_urowid_char ;
...
```

関連項目

- 位置付け更新
- 論理記憶域構造

4.7.1 SQLRowidGet()

SQLLIB関数SQLRowidGet()を使用すると、最後に挿入、更新または選択された行のユニバーサルROWIDへのポインタを取得できます。関数プロトタイプとその引数は、次のとおりです。

```
void SQLRowidGet (dvoid *rctx, OCIRowid **urid) ;
```

rctx (IN)

これは、ランタイム・コンテキストへのポインタです。デフォルト・コンテキストまたは非スレッドの場合は、SQL_SINGLE_RCTXを渡します。

urid (OUT)

これは、ユニバーサルROWIDポインタへのポインタです。通常の実行が終了すると、有効なROWIDをポイントします。エラーが発生した場合は、NULLが戻されます。

> 注意:
>
> SQLRowidGet()をコールするには、ユニバーサル ROWID ポインタを事前に割り当てる必要があります。その後、ユニバーサル ROWID に FREE を使用してください。

4.8 ホスト構造体

C言語の構造体を使用すると、ホスト変数を組み込むことができます。SELECT文またはFETCH文のINTO句と、INSERT文のVALUESリストに、ホスト変数を含んでいる構造体を参照します。ホスト構造体のすべてのコンポーネントは、表4-4の定義のように正当なPro*C/C++ホスト変数である必要があります。

構造体がホスト変数として使用されると、構造体の名前のみがSQL文で使用されます。ただし、構造体の各メンバーは、Oracleにデータを送信したり、問合せでOracleからデータを受信したりします。次の例は、EMP表に従業員を1人追加するときに使用されるホスト構造体を示しています。

```
typedef struct
{
    char  emp_name[11]; /* one greater than column length */
    int   emp_number;
    int   dept_number;
    float salary;
} emp_record;
...
/* define a new structure of type "emp_record" */
emp_record new_employee;

strcpy(new_employee.emp_name, "CHEN");
new_employee.emp_number = 9876;
new_employee.dept_number = 20;
new_employee.salary = 4250.00;

EXEC SQL INSERT INTO emp (ename, empno, deptno, sal)
```

```
    VALUES (:new_employee);
```

メンバーが構造体で宣言される順序は、SQL文中の対応する列の順序と一致する必要があります。また、INSERT文で列のリストが省略されている場合は、データベース表の列の順序と一致する必要があります。

たとえば、ホスト構造体を次のように使用すると無効になり、ランタイム・エラーが発生します。

```
struct
{
    int empno;
    float salary;          /* struct components in wrong order */
    char emp_name[10];
} emp_record;

...
SELECT empno, ename, sal
   INTO :emp_record FROM emp;
```

前述の例が無効となるのは、構造体のコンポーネントが選択リスト内の対応する列とは異なる順序で宣言されているためです。SELECT文の正しい書式は、次のとおりです。

```
SELECT empno, sal, ename   /* reverse order of sal and ename */
    INTO :emp_record FROM emp;
```

4.8.1 ホスト構造体と配列

配列とは、1つの変数名に関連付けられた要素と呼ばれる関連データ項目の集合です。ホスト変数として宣言されたとき、配列はホスト配列と呼ばれます。同様に、配列として宣言されたインジケータ変数はインジケータ配列と呼ばれます。インジケータ配列は、任意のホスト配列に対応付けることができます。

ホスト配列により、データ項目のコレクション全体を単一のSQL文で操作できるため、パフォーマンスを向上させることができます。いくつかの例外を除けば、スカラーのホスト変数が許可される場所であれば、任意の位置でホスト配列を使用できます。また、インジケータ配列は任意のホスト配列に対応付けることができます。

ホスト配列は、ホスト構造体のコンポーネントとして使用できます。次の例では、配列を含む構造体を使用して、EMP表に3つの新規項目をINSERTします。

```
struct
{
    char emp_name[3][10];
    int emp_number[3];
    int dept_number[3];
} emp_rec;
...
strcpy(emp_rec.emp_name[0], "ANQUETIL");
strcpy(emp_rec.emp_name[1], "MERCKX");
strcpy(emp_rec.emp_name[2], "HINAULT");
emp_rec.emp_number[0] = 1964; emp_rec.dept_number[0] = 5;
emp_rec.emp_number[1] = 1974; emp_rec.dept_number[1] = 5;
emp_rec.emp_number[2] = 1985; emp_rec.dept_number[2] = 5;

EXEC SQL INSERT INTO emp (ename, empno, deptno)
    VALUES (:emp_rec);
...
```

関連項目

- ホスト配列

4.8.2 PL/SQLレコード

C言語の**構造体**は、PL/SQLレコードにバインドできません。

4.8.3 ネストした構造体と共用体

ホスト構造体はネストできません。次の例は無効です。

```
struct
{
    int emp_number;
    struct
    {
        float salary;
        float commission;
    } sal_info;            /* INVALID */
    int dept_number;
} emp_record;
...
EXEC SQL SELECT empno, sal, comm, deptno
    INTO :emp_record
    FROM emp;
```

また、C言語の**共用体**をホスト構造体として使用することも、ホスト構造体として使用される構造体に**共用体**をネストすることもできません。

4.8.4 ホスト・インジケータ構造体

標識変数を使用する必要があっても、ホスト変数がホスト構造体に含まれている場合は、ホスト構造体内のホスト変数ごとに標識変数が含まれるように、2番目の構造体を設定します。

たとえば、ホスト構造体*student_record*を次のように宣言するとします。

```
struct
{
    char s_name[32];
    int s_id;
    char grad_date[9];
} student_record;
```

このホスト構造体を次のような問合せで使用するとします。

```
EXEC SQL SELECT student_name, student_idno, graduation_date
    INTO :student_record
    FROM college_enrollment
    WHERE student_idno = 7200;
```

また、卒業日がNULLでもよいかどうかを知る必要があります。さらに、個別のホスト・インジケータ構造体を宣言する必要があります。これは、次のように宣言します。

```
struct
{
    short s_name_ind;  /* indicator variables must be shorts */
    short s_id_ind;
    short grad_date_ind;
} student_record_ind;
```

SQL文中のインジケータ構造体は、ホスト・標識変数を参照する場合と同じ方法で参照してください。

```
EXEC SQL SELECT student_name, student_idno, graduation_date
    INTO :student_record INDICATOR :student_record_ind
    FROM college_enrollment
    WHERE student_idno = 7200;
```

問合せが完了すると、選択された各コンポーネントのNULL/NOT NULLステータスはホスト・インジケータ構造体で使用可能になります。

> **注意:**
>
> このマニュアルでは、従来どおりホスト変数または構造体の名前に*_ind* を追加して、標識変数とインジケータ構造体の名前にしています。ただし、標識変数の名前は任意です。異なる規則を使用しても、規則をまったく使用しなくてもかまいません。

4.8.5 サンプル・プログラム: カーソルとホスト構造体

この項のデモ・プログラムは、明示カーソルを使用し、データをホスト構造体に格納する問合せを示しています。このプログラムは、demoディレクトリのファイルsample2.pc内で使用できます。

```
/*
 *  sample2.pc
 *
 *  This program connects to ORACLE, declares and opens a cursor,
 *  fetches the names, salaries, and commissions of all
 *  salespeople, displays the results, then closes the cursor.
 */

#include <stdio.h>
#include <sqlca.h>

#define UNAME_LEN      20
#define PWD_LEN        40

/*
 * Use the precompiler typedef'ing capability to create
 * null-terminated strings for the authentication host
 * variables. (This isn't really necessary--plain char *'s
 * does work as well. This is just for illustration.)
 */
typedef char asciiz[PWD_LEN];

EXEC SQL TYPE asciiz IS STRING(PWD_LEN) REFERENCE;
asciiz     username;
asciiz     password;

struct emp_info
{
    asciiz     emp_name;
    float      salary;
    float      commission;
};
```

```
/* Declare function to handle unrecoverable errors. */
void sql_error();


main()
{
    struct emp_info *emp_rec_ptr;

/* Allocate memory for emp_info struct. */
    if ((emp_rec_ptr =
        (struct emp_info *) malloc(sizeof(struct emp_info))) == 0)
    {
        fprintf(stderr, "Memory allocation error.¥n");
        exit(1);
    }

/* Connect to ORACLE. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");

    EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--");

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("¥nConnected to ORACLE as user: %s¥n", username);

/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. 'salespeople' is a SQL identifier,
 * not a (C) host variable.
 */
    EXEC SQL DECLARE salespeople CURSOR FOR
        SELECT ENAME, SAL, COMM
            FROM EMP
            WHERE JOB LIKE 'SALES%';

/* Open the cursor. */
    EXEC SQL OPEN salespeople;

/* Get ready to print results. */
    printf("¥n¥nThe company's salespeople are--¥n¥n");
    printf("Salesperson   Salary   Commission¥n");
    printf("-----------   ------   ----------¥n");

/* Loop, fetching all salesperson's statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
 */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        EXEC SQL FETCH salespeople INTO :emp_rec_ptr;
        printf("%-11s%9.2f%13.2f¥n", emp_rec_ptr->emp_name,
                emp_rec_ptr->salary, emp_rec_ptr->commission);
    }

/* Close the cursor. */
    EXEC SQL CLOSE salespeople;

    printf("¥nArrivederci.¥n¥n");
```

```
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}



void
sql_error(msg)
char *msg;
{
    char err_msg[512];
    int buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("¥n%s¥n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s¥n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

4.9 ポインタ変数

C言語では、他の変数を指すポインタがサポートされています。ポインタには、変数の値ではなくアドレス(格納場所)が保持されます。

4.9.1 ポインタ変数の宣言

ポインタは、次のように、通常のC言語の形式に従ってホスト変数として定義します。

```
int   *int_ptr;
char  *char_ptr;
```

4.9.2 ポインタ変数の参照

SQL文では、次のようにポインタに接頭辞としてコロンを付けてください。

```
EXEC SQL SELECT intcol INTO :int_ptr FROM ...
```

文字列へのポインタを除き、参照される値のサイズは宣言で指定したベース型のサイズにより決定されます。文字列へのポインタの場合、参照される値はNULL終了文字列とみなされます。そのサイズは、strlen()関数のコールにより実行時に決定されます。

ポインタを使用すると、**構造体**のメンバーを参照できます。まずポインタ・ホスト変数を宣言してから、次の例に示すように、必要なメンバーのアドレスにポインタを設定します。**構造体**メンバーとポインタ変数のデータ型は同一にする必要があります。両者が一致しない場合、ほとんどのコンパイラでは警告が発行されます。

```
struct
{
    int  i;
```

```
    char c;
} structvar;
int   *i_ptr;
char  *c_ptr;
...
main()
{
    i_ptr = &structvar.i;
    c_ptr = &structvar.c;
/* Use i_ptr and c_ptr in SQL statements. */
...
```

関連項目

- グローバリゼーション・サポート

4.9.3 構造体ポインタ

構造体へのポインタをホスト変数として使用できます。次に例を示します

- 構造体を宣言します。
- 構造体へのポインタを宣言します。
- 構造体にメモリーを割り当てます。
- 構造体へのポインタを問合せでホスト変数として使用します。
- 構造体のコンポーネントの参照を解除して、結果を出力します。

```
struct EMP_REC
{
    int emp_number;
    float salary;
};
char *name = "HINAULT";
...
struct EMP_REC *sal_rec;
sal_rec = (struct EMP_REC *) malloc(sizeof (struct EMP_REC));
...
EXEC SQL SELECT empno, sal INTO :sal_rec
    FROM emp
    WHERE ename = :name;

printf("Employee number and salary for %s: ", name);
printf("%d, %g¥n", sal_rec->emp_number, sal_rec->salary);
```

SQL文では、ホスト構造体へのポインタは、ホスト構造体とまったく同じ方法で参照されます。「...のアドレス」を示す表記(**&**)は不要で、実際には使用するとエラーになります。

4.10 グローバリゼーション・サポート

広く使用されている7ビットまたは8ビットのASCIIおよびEBCDICのキャラクタ・セットは、英数字を表すには十分ですが、日本語のようなアジアの言語には数千もの文字が含まれる場合があります。これらの言語では、1文字を表すために16ビット(2バイト)以上が必要です。Oracleではこうした様々な言語がどのように処理されるかについて説明します。

Oracleには、シングルバイトおよびマルチバイトの文字データを処理し、キャラクタ・セット間で変換できるように、グローバリゼーション・サポートが用意されています。また、アプリケーションを異なる言語環境で実行することもできます。グローバリゼーション・サ

ポートでは、数値書式および日付書式はユーザー・セッション用に指定された言語規則に自動的に適応します。したがって、グローバリゼーション・サポートにより、世界中のユーザーがそれぞれの母国語でOracleとやりとりできます。

様々なグローバリゼーション・サポートまたはNLSパラメータを指定して、言語によって異なる機能の操作を制御できます。これらのパラメータのデフォルト値は、Oracle初期ファイルで設定できます。次の表では、それぞれのグローバリゼーション・サポート・パラメータの指定内容を示しています。

表4-6 **グローバリゼーション・サポート・パラメータ**

グローバリゼーション・サポート・パラメータ	**指定内容**
NLS_LANGUAGE	言語によって異なる表記規則
NLS_TERRITORY	地域によって異なる表記規則
NLS_DATE_FORMAT	日付書式
NLS_DATE_LANGUAGE	日および月の名前に使用する言語
NLS_NUMERIC_CHARACTERS	10 進数文字およびグループ・セパレータ
NLS_CURRENCY	各国通貨記号
NLS_ISO_CURRENCY	ISO 通貨記号
NLS_SORT	ソート基準

主なパラメータは、NLS_LANGUAGEおよびNLS_TERRITORYです。NLS_LANGUAGEでは、言語によって異なる次の機能のデフォルト値を指定します。

- サーバー・メッセージの言語
- 曜日と月の名前に使用する言語
- ソート基準

NLS_TERRITORYには、地域によって異なる機能のデフォルト値を指定します。この機能には次が含まれます。

- 日付書式
- 小数点文字
- グループ・セパレータ
- 各国通貨記号
- ISO通貨記号

パラメータNLS_LANGを次のように指定して、ユーザー・セッション用に言語ごとに異なるグローバリゼーション・サポート機能の操作を制御できます。

```
NLS_LANG = <language>_<territory>.<character set>
```

*language*にはユーザー・セッション用のNLS_LANGUAGEの値、*territory*にはNLS_TERRITORYの値、*character set*には端末に使用されるコード体系を指定します。コード体系(通常はキャラクタ・セットまたはコード・ページと呼ばれる)は、端末で表示可能なキャラクタ・セットに対応する数値コードの範囲です。これには、端末との通信を制御するコードも含まれています。

NLS_LANGは、環境変数(または使用しているシステムでこれに相当するもの)として定義します。たとえば、Cシェルを使用するUNIXでは、NLS_LANGを次のように定義できます。

```
setenv NLS_LANG French_France.WE8ISO8859P1
```

Oracleデータベース・セッション中に、グローバリゼーション・サポート・パラメータの値を変更できます。ALTER SESSION文を次のように使用してください。

```
ALTER SESSION SET <globalization support_parameter> = <value>
```

Pro*C/C++では、グローバリゼーション・サポート機能がすべてサポートされているため、アプリケーションではOracleデータベースに格納されている外国語データを処理できます。たとえば、外国語の文字変数を宣言し、それをINSTRB、LENGTHBおよびSUBSTRBなどの文字列関数に渡すことができます。これらの関数の構文は、それぞれINSTR、LENGTHおよびSUBSTR関数と同じですが、文字単位ではなくバイト単位で機能します。

関数NLS_INITCAP、NLS_LOWERおよびNLS_UPPERを使用して、大/小文字変換の特別なインスタンスを扱うことができます。さらに、関数NLSSORTを使用して、バイナリ順序ではなく言語上の順序でWHERE句の比較を指定できます。グローバリゼーション・サポート・パラメータをTO_CHAR、TO_DATEおよびTO_NUMBER関数に渡すこともできます。

4.11 NCHAR変数

3つの内部データベース・データ型に、各国語キャラクタ・セット・データを格納できます。この種のデータ型は、*NCHAR*、*NCLOB*および*NVARCHAR2*です(*NCHAR VARYING*とも呼ばれます)。これらのデータ型を使用できるのは、リレーショナル列のみです。

4.11.1 CHARACTER SET [IS] NCHAR_CS

各国語キャラクタ・セット・データを保持するホスト変数を指定するには、文字変数宣言に句*CHARACTER SET [IS] NCHAR_CS*を挿入します。これで、その変数に各国語キャラクタ・セット・データを格納できます。トークン*IS*は省略してもかまいません。NCHAR_CSは、各国語キャラクタ・セットの名前です。

次に例を示します。

```
char character set is nchar_cs *str = "<Japanese_string>";
```

この例で、<*Japanese_string*>は、変数NLS_NCHARで定義されたように、各国語キャラクタ・セット*AL16UTF16*にあるUnicode文字で構成されています。

また、コマンドラインにNLS_CHAR=strと入力し、アプリケーションのコードを変更しても、同じ操作を実行できます。

```
char *str = "<Japanese_string>"
```

Pro*C/C++では、このように宣言された変数は、環境変数*NLS_NCHAR*で指定されたキャラクタ・セットとして処理されます。NCHAR変数のサイズは、通常のCの変数と同様にバイト数で指定されます。

データをstrに入れるには、次の簡単な問合せを使用します。

```
EXEC SQL
   SELECT ENAME INTO :str FROM EMP WHERE DEPT = n'<Japanese_string1>';
```

また、*str*を次のSELECT文で使用することもできます。

```
EXEC SQL
   SELECT DEPT INTO :dept FROM DEPT_TAB WHERE ENAME = :str;
```

4.11.2 環境変数NLS_NCHAR

Pro*C/C++では、NLS_LOCAL=NOの場合に、データベース・サポートとともに各国語キャラクタ・セットがサポートされます。NLS_LOCAL=NOで、新規の環境変数NLS_NCHARが有効な各国語キャラクタ・セットに設定されていれば、データベース・サーバーでNCHARがサポートされます。

NLS_NCHARでは、各国語キャラクタ・セット・データ(NCHAR、NVARCHAR2、NCLOB)に使用するキャラクタ・セットを指定します。キャラクタ・セットを指定しない場合は、定義済、あるいはNLS_LANGで間接的に定義済のキャラクタ・セットが使用されます。

NLS_NCHARには、プリコンパイル時と実行時に、有効な各国語キャラクタ・セット(言語名ではなく、NLS_LANGで設定)を指定する必要があります。SQLLIBでは、最初のSQL文の実行時にランタイム・チェックが実行されます。プリコンパイル時と実行時のキャラクタ・セットが異なっていると、SQLLIBからエラー・コードが戻されます。

4.11.3 VAR文でのCONVBUFSZ句

EXEC SQL VAR文を使用すると、ホスト変数をOracleの外部データ型に同値化して、デフォルトの割当てを上書きできます。これはホスト変数の同値化と呼ばれます。

EXEC SQL VAR文には、オプション句CONVBUFSZ (<size>)を使用できます。Oracleランタイム・ライブラリ内で、指定したホスト変数をキャラクタ・セット間で変換するためのバッファのサイズ<size>をバイト単位で指定します。

新しい構文は、次のいずれかです。

```
EXEC SQL VAR host_variable IS datatype [CONVBUFSZ [IS] (size)] ;
```

または

```
EXEC SQL VAR host_variable [CONVBUFSZ [IS] (size)];
```

datatype は、次のとおりです。

```
type_name [ ( { length | precision, scale } ) ]
```

関連項目

- VAR(Oracle埋込みSQLディレクティブ)

4.11.4 埋込みSQL内の文字列

埋込みSQL文中のマルチバイト文字列は、その文字列をマルチバイトとして識別する文字リテラルと、後続の文字列で構成されます。文字列は、通常の一重引用符(')で囲みます。

たとえば、次のような埋込みSQL文があるとします。

```
EXEC SQL SELECT empno INTO :emp_num FROM emp
    WHERE ename = N'<Japanese_string>';
```

このSQL文には、マルチバイト文字列が含まれています(<Japanese_string>は、実際には漢字の可能性があります)。つまり、文字列の直前に文字リテラルNが付いているため、これはマルチバイト文字列として識別されます。Oracleでは大/小文字が区別されないため、この例ではnとNのどちらを使用してもかまいません。

4.11.5 文字列の制限事項

マルチバイト文字列では、データ型の同値化(TYPEまたはVARコマンド)は使用できません。

動的SQL方法4は、Pro*C/C++のマルチバイト文字列ホスト変数には使用できません。

4.11.6 標識変数

標識変数は、マルチバイト・キャラクタ(NLS_CHARオプションで指定)であるホスト文字変数とともに使用できます。

5 高度なトピック

この章では、Pro*C/C++の高度な技術について説明します。この章の項目は、次のとおりです。

- 文字データ
- データ型変換
- データ型の同値化
- Cプリプロセッサ
- プリコンパイル済ヘッダー・ファイル
- Oracleプリプロセッサ
- 数値定数の評価
- OCIリリース8のSQLLIB拡張相互運用性
- OCIリリース8へのインタフェース
- OCIリリース7コールの埋込み
- SQLLIBパブリック関数の新しい名前
- X/Openアプリケーションの開発

5.1 文字データ

この項では、Pro*C/C++プリコンパイラによる文字ホスト変数の処理方法について説明します。ホスト変数のキャラクタ・タイプには、次の4種類があります。

- 文字配列
- 文字列へのポインタ
- VARCHAR変数
- VARCHARへのポインタ

VARCHAR(プリコンパイラで提供されるホスト変数データ構造体)とVARCHAR2(可変長文字列に対応するOracleの内部データ型)を混同しないでください。

5.1.1 プリコンパイラ・オプションCHAR_MAP

CHAR_MAPプリコンパイラ・オプションを使用して、char[n]およびcharホスト変数のデフォルトのマッピングを指定できます。Oracleでは、CHARZにマップされます。CHARZにより、ANSI固定文字書式が実装されます。文字列は固定長で、空白文字で埋めてヌル文字で終了します。VARCHAR2値(ヌル文字を含む)は、常に固定長で空白文字で埋められます。表5-1は、CHAR_MAPの可能な設定を示しています。

表5-1 CHAR_MAPの設定

CHAR_MAPの設定	デフォルトとなる場合	説明
VARCHAR2	-	すべての値(ヌル文字を含む)が固定長で空白文字で埋め

CHAR_MAPの設定	デフォルトとなる場合	説明
		られます。
CHARZ	DBMS=V7、DBMS=V8	固定長で、空白文字で埋めてヌル文字で終了。ANSI 固定キャラクタ・タイプに準拠しています。
STRING	新しい書式	ヌル文字で終了。C プログラムで使用されている ASCII 形式に準拠しています。
CHARF	以前は、VAR 宣言または TYPE 宣言が行われた場合のみ。	固定長で空白文字で埋められ、ヌル文字は使用されません。

デフォルトのマッピングは、Pro*C/C++の旧リリースと同じCHAR_MAP=CHARZです。

廃止された従来のDBMS=V6_CHARのかわりにCHAR_MAP=VARCHAR2を使用してください。

5.1.2 CHAR_MAPオプションのインラインでの使用方法

char変数またはchar[n]変数を異なる方法で宣言していないかぎり、そのマッピングはインラインCHAR_MAPオプションにより決定されます。次のコード例は、このオプションをPro*C/C++でインライン設定した結果を示しています。

```
char ch_array[5];

strncpy(ch_array, "12345", 5);
/* char_map=charz is the default in Oracle7 and Oracle8 */
EXEC ORACLE OPTION (char_map=charz);
/* Select retrieves a string "AB" from the database */
SQL SELECT ... INTO :ch_array FROM ... WHERE ... ;
/* ch_array == { 'A', 'B', ' ', ' ', '¥0' } */

strncpy (ch_array, "12345", 5);
EXEC ORACLE OPTION (char_map=string) ;
/* Select retrieves a string "AB" from the database */
EXEC SQL SELECT ... INTO :ch_array FROM ... WHERE ... ;
/* ch_array == { 'A', 'B', '¥0', '4', '5' } */

strncpy( ch_array, "12345", 5);
EXEC ORACLE OPTION (char_map=charf);
/* Select retrieves a string "AB" from the database */
EXEC SQL SELECT ... INTO :ch_array FROM ... WHERE ... ;
/* ch_array == { 'A', 'B', ' ', ' ', ' ' } */
```

5.1.3 DBMSオプションおよびCHAR_MAPオプションの影響

DBMSオプションとCHAR_MAPオプションにより、Pro*C/C++で文字配列および文字列のデータを処理する方法が決定されます。これらのオプションにより、プログラムではANSI固定長文字列との互換性、あるいは可変長文字列を使用するOracleおよびPro*C/C++の旧リリースとの互換性を維持できます。

DBMSオプションは、入力(ホスト変数からOracle表へ)および出力(Oracle表からホスト変数へ)の両方で文字データに影響します。

文字配列およびCHAR_MAPオプション

文字配列のマッピングは、DBMSオプションとは関連しないCHAR_MAPオプションでも設定できます。DBMS=V7またはDBMS=V8の場合は、どちらもCHAR_MAP=CHARZが使用されます。これは、CHAR_MAP=VARCHAR2、STRINGまたはCHARFを指定することで上書きできます。

関連項目

- プリコンパイラ・オプション

5.1.3.1 入力時

文字配列

入力時に、DBMSオプションにより、プログラム内でホスト変数の文字配列に必要な形式が決定されます。CHAR_MAP=VARCHAR2の場合、ホスト変数の文字配列は空白で埋める必要があります。また、ヌル文字で終了しないでください。DBMS=V7またはV8の場合は、文字配列にヌル終端文字('¥0')を付ける必要があります。

CHAR_MAPオプションがVARCHAR2に設定されると、後続の空白文字が最初の非空白文字まで削除されてから、値がデータベースに送られます。未初期化文字配列には、ヌル文字が含まれている場合があります。NULLが表に挿入されるのを確実に防ぐには、長さに達するまで文字配列に空白文字を埋める必要があります。たとえば、次の文を実行するとします。

```
char emp_name[10];
...
strcpy(emp_name, "MILLER");      /* WRONG! Note no blank-padding */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

文字列MILLERがMILLER¥0¥0¥0¥0として挿入されていることがわかります(末尾に4個のNULLバイトが追加されています)。この値は、次の検索条件とは一致しません。

```
. . . WHERE ename = 'MILLER';
```

CHAR_MAPがVARCHAR2に設定されている場合に文字配列をINSERTするには、次の文を実行します。

```
strncpy(emp_name, "MILLER    ", 10); /* 4 trailing blanks */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

DBMS=V7またはV8の場合は、文字配列内の入力データにNULL終了記号を付ける必要があります。したがって、データがNULLで終わっているか確認してください。

```
char emp_name[11];  /* Note: one greater than column size of 10 */
...
strcpy(emp_name, "MILLER");        /* No blank-padding required */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

文字ポインタ

ポインタとして、入力データを保持できる大きさの、ヌル文字で終了するバッファをアドレス指定する必要があります。プログラムでは、そのために十分なメモリーを割り当てる必要があります。

5.1.3.2 入力時

次の例は、データベースから取り出して文字配列に格納する値にCHAR_MAPオプションの設定が及ぼす影響の可能な組合せをすべて示しています。

次のデータベースについて考えてみます。

```
TABLE strdbase ( ..., strval VARCHAR2(6));
```

strval列に次の文字列が入っているとします。

```
""        -- string of length 0
"AB"      -- string of length 2
"KING"    -- string of length 4
"QUEEN"   -- string of length 5
"MILLER"  -- string of length 6
```

Pro*C/C++プログラムでは、5文字のホスト配列*str*を文字Xで初期化し、strval列のすべての値の取出しに使用するとします。

```
char  str[5] = {'X', 'X', 'X','X', 'X'} ;
short str_ind;
...
EXEC SQL SELECT strval INTO :str:str_ind WHERE ... ;
```

CHAR_MAPがVARCHAR2、CHARF、CHARZおよびSTRINGに設定されると、配列strおよび標識変数str_indの結果は次のようになります。

```
strval = ""          "AB"       "KING"     "QUEEN"    "MILLER"
-------------------------------------------------------------
VARCHAR2 "     " -1 "AB   " 0  "KING "  0 "QUEEN"  0 "MILLE"  6
CHARF    "XXXXX" -1 "AB   " 0  "KING "  0 "QUEEN"  0 "MILLE"  6
CHARZ    "    0" -1 "AB  0" 0  "KING0"  0 "QUEE0"  5 "MILL0"  6
STRING   "0XXXX" -1 "AB0XX" 0  "KING0"  0 "QUEE0"  5 "MILL0"  6
```

0はヌル文字'¥0'を表します。

5.1.3.3 出力時

文字配列

出力時に、DBMSオプションとCHAR_MAPオプションにより、プログラム内でホスト変数の文字配列に使用される形式が決定されます。CHAR_MAP=VARCHAR2の場合、ホスト変数の文字配列は配列の長さに達するまで空白文字で埋められますが、ヌル文字で終了しません。DBMS=V7またはV8(あるいはCHAR_MAP=CHARZ)の場合、文字配列は空白文字で埋めて、配列の最終位置にヌル文字を入れて終了します。

次の文字出力の例について考えてみます。

```
CREATE TABLE test_char (C_col CHAR(10), V_col VARCHAR2(10));

INSERT INTO test_char VALUES ('MILLER', 'KING');
```

この表を選択するプリコンパイラ・プログラムには、次のような埋込みSQLが記述されています。

```
...
char name1[10];
char name2[10];
...
EXEC SQL SELECT C_col, V_col INTO :name1, :name2
    FROM test_char;
```

CHAR_MAP=VARCHAR2を指定してプログラムをプリコンパイルすると、*name1*には次の文字列が入ります。

```
"MILLER####"
```

つまり、名前MILLERの後に4個の空白が続き、ヌル文字で終了しません。(*name1*がサイズ15で宣言されている場合は、名前に続く空白が9個になります)。

*name2*には次の文字列が入ります。

```
"KING######"      /* 6 trailing blanks */
```

DBMS=V7またはV8を指定してプログラムをプリコンパイルすると、*name1*には次の文字列が入ります。

```
"MILLER###¥0" /* 3 trailing blanks, then a null-terminator */
```

つまり、名前を含み、列の長さに達するまで空白文字で埋められ、ヌル文字で終了する文字列です。*name2*には、次の文字列が入ります。

```
"KING#####¥0"
```

要約すると、CHAR_MAP=VARCHAR2の場合、CHARACTER列またはVARCHAR2列からの出力には、ホスト変数配列の長さに達するまで空白埋めが行われます。DBMS=V7またはV8の場合、出力文字列は常にヌル文字で終了します。

文字ポインタ

DBMSオプションとCHAR_MAPオプションを使用しても、文字データがポインタ・ホスト変数に出力される方法には影響しません。

データを文字ポインタ・ホスト変数に出力する場合、ポインタが指すバッファには、表からの出力に加えてヌル終端文字のための1バイトを保持できるサイズが必要です。

プリコンパイラ・ランタイム環境では、strlen()をコールして出力バッファのサイズを決定するため、バッファに埋込みヌル('¥0')が含まれていないことを確認してください。データをフェッチする前に'¥0'以外の値で割当て済バッファを満たし、最後のバイトにヌル文字を入れて終了してください。

注意:

C 言語のポインタは、DBMS=V7 または V8 と MODE=ANSI を指定してプリコンパイルした Pro*C/C++プログラムで使用できます。ただし、ポインタは、SQL 規格準拠のプログラムでは有効なホスト変数型ではありません。ポインタをホスト変数として使用すると、FIPS フラガーにより警告が発行されます。

次のコード例は、前項で定義した列と表を使用して、文字ポインタ・ホスト変数に宣言およびSELECTを行う方法を示しています。

```
...
char *p_name1;
char *p_name2;
...
p_name1 = (char *) malloc(11);
p_name2 = (char *) malloc(11);
strcpy(p_name1, "          ");
strcpy(p_name2, "0123456789");

EXEC SQL SELECT C_col, V_col INTO :p_name1, :p_name2
    FROM test_char;
```

前述のSELECT文がDBMSまたはCHAR_MAP設定で実行されると、フェッチされる値は次のようになります。

```
"MILLER####¥0"     /* 4 trailing blanks and a null terminator */

"KING######¥0"    /* 6 blanks and null */
```

5.1.4 VARCHAR変数およびポインタ

次の例は、VARCHARホスト変数の宣言方法を示しています。

```
VARCHAR   emp_name1[10];   /* VARCHAR variable   */
VARCHAR  *emp_name2;       /* pointer to VARCHAR */
```

5.1.4.1 入力時

VARCHAR変数

VARCHAR変数を入力ホスト変数として使用すると、プログラムに必要なのは、展開されたVARCHAR宣言(例では*emp_name1.arr*)の配列メンバーに必要な文字列を配置し、長さメンバー(*emp_name1.len*)を設定することのみです。配列に空白文字を埋める必要はありません。*emp_name1.len*の文字が正確にOracleに送られ、空白文字およびヌル文字があればカウントされます。次の例では、*emp_name1.len*を8に設定します。

```
strcpy((char *)emp_name1.arr, "VAN HORN");
emp_name1.len = strlen((char *)emp_name1.arr);
```

VARCHARへのポインタ

VARCHARへのポインタを入力ホスト変数として使用する場合は、展開されるVARCHAR宣言に十分なメモリーを割り当てる必要があります。その後、次のように、必要な文字列を配列メンバーに配置し、長さメンバーを設定します。

```
emp_name2 = malloc(sizeof(short) + 10)   /* len + arr */
strcpy((char *)emp_name2->arr, "MILLER");
emp_name2->len = strlen((char *)emp_name2->arr);
```

または、*emp_name2*が既存のVARCHAR(この場合は*emp_name1*)を指すように、割当てを次のように記述できます。

```
emp_name2 = &emp_name1;
```

その後、次のように通常の方法でVARCHARポインタを使用します。

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
    VALUES (:emp_number, :emp_name2, :dept_number);
```

5.1.4.2 出力時

VARCHAR変数

VARCHAR変数を出力ホスト変数として使用すると、プログラム・インタフェースにより長さメンバーが設定されますが、配列メンバーはヌル文字で終了しません。文字配列の場合と同様に、プログラムではVARCHAR変数の*arr*メンバーをヌル文字で終了してから、printf()またはstrlen()などの関数に渡すことができます。次に例を示します。

```
emp_name1.arr[emp_name1.len] = '¥0';
printf("%s", emp_name1.arr);
```

また、長さメンバーを使用して、次のように文字列の出力を制限できます。

```
printf("%.*s", emp_name1.len, emp_name1.arr);
```

VARCHAR変数が文字配列よりも優れている点は、Oracleによって戻される値の長さがすぐにわかることです。文字配列の場

合は、文字列の実際の長さを知るために、後続する空白を手動で削除する操作が必要になる場合もあります。

VARCHARポインタ

VARCHARへのポインタを出力ホスト変数として使用すると、プログラム・インタフェースでは長さメンバー(例では*emp_name2->len*)を調べることで、変数の最大長が決定されます。したがって、プログラムではフェッチの前ごとにこのメンバーを設定する必要があります。その後のフェッチ処理時に、次のように、長さメンバーは戻された実際の文字数に設定されます。

```
emp_name2->len = 10;  /* Set maximum length of buffer. */
EXEC SQL SELECT ENAME INTO :emp_name2 WHERE EMPNO = 7934;
printf("%d characters returned to emp_name2", emp_name2->len);
```

5.1.5 Unicode変数

Pro*C/C++では、ホストchar変数で固定幅のUnicodeデータ(キャラクタ・セットUnicode標準バージョン3.0、UCS-16とも呼ばれます)を使用できます。UCS-16では1文字に2バイトが使用されるため、データ型は符号なし2バイトです。UCS-16表記のSQL文は、まだサポートされていません。

次のサンプル・コードでは、Unicodeの*utext*型のホスト変数employeeが、20 Unicode文字長として宣言されています。emp表は、60バイト長の列enameを含むように作成されます。これにより、マルチバイト文字で3バイト長以内の、アジア言語のデータベース・キャラクタ・セットがサポートされます。

```
utext employee[20] ;                              /* Unicode host variable   */
EXEC SQL CREATE TABLE emp (ename CHAR(60));
/* ename is in the current database character set  */
EXEC SQL INSERT INTO emp (ename) VALUES ('test') ;
/* 'test' in NLS_LANG encoding converted to database character set */
EXEC SQL SELECT * INTO :employee FROM emp ;
/* Database character set converted to Unicode */
```

パブリック・ヘッダー・ファイルsqlucs2.hを、アプリケーション・コードにインクルードする必要があります。次を実行します。

- 次の文を含めます。

```
#include <oratypes.h>
```

- uvarchar、つまりUnicode varcharは、次のように定義します。

```
struct uvarchar
{
   ub2 len;
   utext arr[1] ;
};
typedef struct uvarchar uvarchar ;
```

- ulong_varchar、つまりUnicode long varcharは、次のように定義します。

```
struct ulong_varchar
{
   ub4 len ;
   utext arr[1] ;
}
typedef struct ulong_varchar ulong_varchar ;
```

utextのデフォルトのデータ型は、すべての文字変数のデフォルトであるCHARZと同じで、空白文字で埋められ、ヌル文字で終了します。

CHAR_MAPプリコンパイラ・オプションを使用して、次のようにデフォルト・データ型を変更してください。

```
#include <sqlca.h>
#include <sqlucs2.h>

main()
{
   utext employee1[20] ;

/* Change to STRING datatype:    */
   EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
   utext employee2[20] ;

   EXEC SQL CREATE TABLE emp (ename CHAR(60)) ;
   ...
/*************************************************************
  Initializing employee1 or employee2 is compiler-dependent.
 *************************************************************/
   EXEC SQL INSERT INTO emp (ename) VALUES (:employee1) ;
   ...
   EXEC SQL SELECT ename INTO :employee2 FROM emp;
/* employee2 is now not blank-padded and is null-terminated  */
   ...
```

5.1.5.1 Unicode変数の使用に関する制限事項

- SQL文の静的SQLおよび動的SQLにUnicodeを含めることはできません。次の記述は許可されません。

```
#include oratypes.h
utext sqlstmt[100] ;
...
/* If sqlstmt contains a SQL statement: */
EXEC SQL PREPARE s1 FROM :sqlstmt ;
EXEC SQL EXECUTE IMMEDIATE :sqlstmt ;
...
```

- utext変数には、データ型の同値化を使用できません。次のコードは使用できません。

```
    typedef utext utext_5 ;
    EXEC SQL TYPE utext_5 IS STRING ;
```

- CONVBUFSZは、変換バッファ・サイズとして使用できません。かわりにCHAR_MAPオプションを使用してください。
- Oracleの動的SQL方法4では、Unicodeはサポートされません。
- オブジェクト型では、Unicodeはサポートされません。

関連項目

- VAR文でのCONVBUFSZ句
- ANSI動的SQL
- オブジェクト

5.2 データ型変換

プリコンパイル時には、各ホスト変数にデフォルトの外部データ型が割り当てられます。たとえば、プリコンパイラでは、**short int**型および**int**型のホスト変数にINTEGER外部データ型が割り当てられます。

SQL文で使用するすべてのホスト変数のデータ型コードは、実行時にOracleに渡されます。Oracleは、コードを使用して内部データ型と外部データ型の間で変換します。

選択された列(または疑似列)の値を出力ホスト変数に割り当てる前に、Oracleではソース列の内部データ型をホスト変数のデータ型に変換する必要があります。同様に、入力ホスト変数の値の、列への割当てまたは比較を行う場合は、その前に必ずホスト変数の外部データ型をターゲット列の内部データ型に変換します。

内部データ型と外部データ型との変換は、通常のデータ変換規則に従って行われます。たとえば、CHAR値「1234」をC言語の*short*値に変換できます。CHAR値「65543」(大きすぎる値)または「10F」(10進数でない値)は、C言語の*short*値に変換できません。同様に、アルファベット文字が含まれるchar [n]値はNUMBER値に変換できません。

5.3 データ型の同値化

データ型の同値化により、Oracleで入力データを解釈する方法と出力データの書式を設定する方法を制御できます。データ型の同値化を使用すると、プリコンパイラにより割り当てられるデフォルトの外部データ型を上書きできます。サポートされているC言語のホスト変数データ型は、変数ごとにOracleの外部データ型にマップ(つまり同値化)できます。また、ユーザー定義のデータ型をOracleの外部データ型にマップすることもできます。

5.3.1 ホスト変数の同値化

Pro*C/C++プリコンパイラでは、デフォルトで、各ホスト変数に特定の外部データ型が割り当てられます。

表5-2は、デフォルトの割当てを示しています。

表5-2 デフォルトの型割当て

C言語のデータ型または疑似型	Oracleの外部型	注意
char	VARCHAR2	(CHAR_MAP=VARCHAR2)
char[n]	CHARZ	(DBMS=V7、V8 のデフォルト)
char*	STRING	(CHAR_MAP=STRING)
	CHARF	(CHAR_MAP=CHARF)
int、int*	INTEGER	-
short、short*	INTEGER	-
long、long*	INTEGER	-
long long、long long*	INTEGER	-
float、float*	FLOAT	-
double、double*	FLOAT	-
VARCHAR*、VARCHAR[n]	VARCHAR	-

VAR文を使用すると、ホスト変数をOracleの外部データ型に同値化して、デフォルトの割当てを上書きできます。使用する構

文は次のとおりです。

```
EXEC SQL VAR host_variable IS type_name [ (length) ];
```

*host_variable*はすでに宣言済の入力ホスト変数または出力ホスト変数(またはホスト配列)、*type_name*は有効な外部データ型の名前、*length*は有効な長さをバイト数で指定する整数リテラルです。

ホスト変数の同値化には、いくつかの利点があります。たとえば、EMP表から従業員名を選択して、ヌル文字で終了する文字列を前提としたルーチンに渡すとします。これらの名前に、明示的にヌル終端文字を付ける必要はありません。次のように、ホスト変数をSTRING外部データ型に同値化するのみです。

```
...
char  emp_name[11];
EXEC SQL VAR emp_name IS STRING(11);
```

EMP表のENAME列の長さは10文字のため、ヌル終端文字を含めるために、新しい*emp_name*に11文字を割り当てます。ENAME列から*emp_name*に入れる値をSELECTする場合、プログラム・インタフェースにより値はヌル文字で終了します。

NUMBERを除き、どの外部データ型(VARNUMなど)でも使用できます。

関連項目

- 外部データ型

5.3.2 ユーザー定義型同値化

また、ユーザー定義のデータ型をOracleの外部データ型にマップ(つまり同値化)することもできます。まず、要件を満たす外部データ型に似た構造の新しいデータ型を定義します。次に、TYPE文を使用して、新しいデータ型を外部データ型にマップします。

TYPE文を使用すると、ホスト変数のクラス全体にOracleの外部データ型を割り当てることができます。使用する構文は次のとおりです。

```
EXEC SQL TYPE user_type IS type_name [ (length) ] [REFERENCE];
```

図形文字を使用するために、可変長文字列データ型が必要であるとします。最初に、**short**型の長さコンポーネントと、それに続く65533バイトのデータ・コンポーネントからなる構造体を宣言します。次に、**typedef**を使用して、その構造体に基づく新規のデータ型を定義します。最後に、新しいユーザー定義データ型をVARRAW外部データ型に同値化します。次に例を示します。

```
struct  screen
{
    short  len;
    char   buff[4000];
};
typedef struct screen graphics;

EXEC SQL TYPE graphics IS VARRAW(4000);
graphics  crt;  - host variable of type graphics
    ...
```

新しい*graphics*型に長さ4000バイトを指定します。この構造体では、それがデータ・コンポーネントの最大長になるためです。プリコンパイラは、長さをOracleサーバーに送るときに、*len*コンポーネント(および必要なパディング)を考慮します。

5.3.2.1 REFERENCE句

ユーザー定義型はポインタとして宣言できます。明示的にスカラー型または構造体型へのポインタとして宣言することも、暗黙的に配列として宣言することもできます。また、この型をEXEC SQL TYPE文で使用できます。この場合は、文の終わりに

REFERENCE句を使用してください。次に例を示します。

```
typedef unsigned char *my_raw;

EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
my_raw    graphics_buffer;
...
graphics_buffer = (my_raw) malloc(4004);
```

この例では、型の長さ(4000)を超過したメモリーを割り当てています。プリコンパイラが長さ(*short*のサイズ)を戻し、システムのワード・アラインメント制限を考慮して長さの後に埋込みをパディングできるようにするため、このような割当てが必要になります。システムのワード・アラインメントの制限が不明な場合は、必ずその長さとパディングの分として余分に数バイトを割り当ててください(通常は9バイトで十分です)。

関連項目

- サンプル・プログラム: sqlvcp()の使用

5.3.3 CHARF外部データ型

CHARFは固定長文字列です。このデータ型をVAR文およびTYPE文で使用して、DBMSオプションまたはCHAR_MAPオプションの設定に関係なく、C言語のデータ型を固定長のSQL標準データ型CHARに同値化できます。

DBMS=V7またはDBMS=V8の場合は、VAR文またはTYPE文で外部データ型CHARACTERを指定すると、C言語のデータ型が固定長のCHARデータ型(データ型コード96)に同値化されます。ただし、CHAR_MAP=VARCHAR2の場合、C言語のデータ型は可変長のVARCHAR2データ型(コード1)に同値化されます。

現在では、VAR文またはTYPE文でCHARFデータ型を使用すれば、常にC言語のデータ型を固定長のSQL標準のCHAR型に同値化できます。CHARFを使用すると、DBMSオプションまたはCHAR_MAPオプションの設定に関係なく、常に固定長のキャラクタ・タイプになるように同値化されます。

5.3.4 EXEC SQL VARおよびTYPEディレクティブ

EXEC SQL VAR ...文またはEXEC SQL TYPE ...文は、プログラム内の任意の位置に記述できます。これらの文は、その影響を受ける変数のデータ型を、TYPE文またはVAR文が記述された位置から変数のスコープの終わりまでの範囲内で変更する実行文として扱われます。MODE=ANSIを指定してプリコンパイルする場合は、宣言部を使用する必要があります。この場合、TYPE文またはVAR文は宣言部中に記述してください。

関連項目

- TYPE(Oracle埋込みSQLディレクティブ)
- VAR(Oracle埋込みSQLディレクティブ)

5.3.5 例: データ型の同値化(sample4.pc)

この項のデモ・プログラムは、Pro*C/C++プログラムでデータ型の同値化を使用する方法を示しています。このプログラムは、demoディレクトリ内のsample4.pcとして使用でき、LONG VARRAW外部データ型を使用してデータ型の同値化方法を示しています。異なるシステム間で移植できる実用的な例を示すために、このプログラムではバイナリ・ファイルをデータベースに挿入し、そのファイルをデータベースから取り出します。

このプログラムでは、LOB埋込みSQL文を使用しています。

このプログラムの目的については、導入部分のコメントを参照してください。

```
/*************************************************************
```

```
sample4.pc
This program demonstrates the use of type equivalencing using the
LONG VARRAW external datatype. In order to provide a useful example
that is portable across different systems, the program inserts
binary files into and retrieves them from the database.  For
example, suppose you have a file called 'hello' in the current
directory.  You can create this file by compiling the following
source code:

#include <stdio.h>

int main()
{
  printf("Hello World!¥n");
}

When this program is run, we get:

$hello
Hello World!

Here is some sample output from a run of sample4:

$sample4
Connected.
Do you want to create (or re-create) the EXECUTABLES table (y/n)? y
EXECUTABLES table successfully dropped.  Now creating new table...
EXECUTABLES table created.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables stored in the database
(D)elete an executable from the database
(Q)uit the program

Enter i, r, l, or q: l

Executables           Length (bytes)
--------------------  --------------

Total Executables: 0

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables stored in the database
(D)elete an executable from the database
(Q)uit the program

Enter i, r, l, or q: i
Enter the key under which you will insert this executable: hello
Enter the filename to insert under key 'hello'.
If the file is not in the current directory, enter the full
path: hello
Inserting file 'hello' under key 'hello'...
Inserted.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
```

```
(R)etrieve an executable from the database
(L)ist the executables stored in the database
(D)elete an executable from the database
(Q)uit the program

Enter i, r, l, or q: l

Executables           Length (bytes)
--------------------  --------------
hello                           5508

Total Executables: 1

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables stored in the database
(D)elete an executable from the database
(Q)uit the program

Enter i, r, l, or q: r
Enter the key for the executable you wish to retrieve: hello
Enter the file to write the executable stored under key hello into.  If you
don't want the file in the current directory, enter the
full path: h1
Retrieving executable stored under key 'hello' to file 'h1'...
Retrieved.

Sample 4 Menu.  Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L)ist the executables stored in the database
(D)elete an executable from the database
(Q)uit the program

Enter i, r, l, or q: q

We now have the binary file 'h1' created, and we can run it:

$h1
Hello World!
**********************************************************************/

#include <oci.h>
#include <string.h>
#include <stdio.h>
#include <sqlca.h>
#include <stdlib.h>
#include <sqlcpr.h>

/* Oracle error code for 'table or view does not exist'. */
#define NON_EXISTENT  -942
#define NOT_FOUND     1403

/* This is the definition of the long varraw structure.
 * Note that the first field, len, is a long instead
 * of a short.  This is becuase the first 4
 * bytes contain the length, not the first 2 bytes.
 */
typedef struct long_varraw {
```

```
  ub4  len;
  text buf[1];
} long_varraw;


/* Type Equivalence long_varraw to LONG VARRAW.
 * All variables of type long_varraw from this point
 * on in the file will have external type 95 (LONG VARRAW)
 * associated with them.
 */
EXEC SQL TYPE long_varraw IS LONG VARRAW REFERENCE;


/* This program's functions declared. */
#if defined(__STDC__)
  void do_connect(void);
  void create_table(void);
  void sql_error(char *);
  void list_executables(void);
  void print_menu(void);
  void do_insert(varchar *, char *);
  void do_retrieve(varchar *, char *);
  void do_delete(varchar *);
  ub4  read_file(char *, OCIBlobLocator *);
  void write_file(char *, OCIBlobLocator *);
#else
  void do_connect(/*_ void _*/);
  void create_table(/*_ void _*/);
  void sql_error(/*_ char * _*/);
  void list_executables(/*_ void _*/);
  void print_menu(/*_ void _*/);
  void do_insert(/*_ varchar *, char * _*/);
  void do_retrieve(/*_ varchar *, char * _*/);
  void do_delete(/*_ varchar * _*/);
  ub4  read_file(/*_ char *, OCIBlobLocator * _*/);
  void write_file(/*_ char *, OCIBlobLocator * _*/);
#endif

void main()
{
  char reply[20], filename[100];
  varchar key[20];
  short ok = 1;

  /* Connect to the database. */
  do_connect();

  printf("Do you want to create (or re-create) the EXECUTABLES table (y/n)? ");
  gets(reply);

  if ((reply[0] == 'y') || (reply[0] == 'Y'))
    create_table();

  /* Print the menu, and read in the user's selection. */
  print_menu();
  gets(reply);

  while (ok)
  {
    switch(reply[0]) {
```

```
    case 'I': case 'i':
      /* User selected insert - get the key and file name. */
      printf("Enter the key under which you will insert this executable: ");
      key.len = strlen(gets((char *)key.arr));
      printf("Enter the filename to insert under key '%.*s'.¥n",
             key.len, key.arr);
      printf("If the file is not in the current directory, enter the full¥n");
      printf("path: ");
      gets(filename);
      do_insert((varchar *)&key, filename);
      break;
    case 'R': case 'r':
      /* User selected retrieve - get the key and file name. */
      printf("Enter the key for the executable you wish to retrieve: ");
      key.len = strlen(gets((char *)key.arr));
      printf("Enter the file to write the executable stored under key ");
      printf("%.*s into.  If you¥n", key.len, key.arr);
      printf("don't want the file in the current directory, enter the¥n");
      printf("full path: ");
      gets(filename);
      do_retrieve((varchar *)&key, filename);
      break;
    case 'L': case 'l':
      /* User selected list - just call the list routine. */
      list_executables();
      break;
    case 'D': case 'd':
      /* User selected delete - get the key for the executable to delete. */
      printf("Enter the key for the executable you wish to delete: ");
      key.len = strlen(gets((char *)key.arr));
      do_delete((varchar *)&key);
      break;
    case 'Q': case 'q':
      /* User selected quit - just end the loop. */
      ok = 0;
      break;
    default:
      /* Invalid selection. */
      printf("Invalid selection.¥n");
      break;
    }

    if (ok)
    {
      /* Print the menu again. */
      print_menu();
      gets(reply);
    }
  }

  EXEC SQL COMMIT WORK RELEASE;
}


/* Connect to the database. */
void do_connect()
{
  /* Note this declaration: uid is a char * pointer, so Oracle
     will do a strlen() on it at runtime to determine the length.
   */
```

```
  char *uid = "scott/tiger";

  EXEC SQL WHENEVER SQLERROR DO sql_error("do_connect():CONNECT");
  EXEC SQL CONNECT :uid;

  printf("Connected.¥n");
}


/* Creates the executables table. */
void create_table()
{
  /* We are going to check for errors ourselves for this statement. */
  EXEC SQL WHENEVER SQLERROR CONTINUE;

  EXEC SQL DROP TABLE EXECUTABLES;
  if (sqlca.sqlcode == 0)
    {
      printf("EXECUTABLES table successfully dropped.  ");
      printf("Now creating new table...¥n");
    }
  else if (sqlca.sqlcode == NON_EXISTENT)
    {
      printf("EXECUTABLES table does not exist.  ");
      printf("Now creating new table...¥n");
    }
  else
    sql_error("create_table()");

  /* Reset error handler. */
  EXEC SQL WHENEVER SQLERROR DO sql_error("create_table():CREATE TABLE");

  EXEC SQL CREATE TABLE EXECUTABLES
    ( name VARCHAR2(30), length NUMBER(10), binary BLOB ) ;

  printf("EXECUTABLES table created.¥n");
}

/* Opens the binary file identified by 'filename' for reading, and writes
   it into into a Binary LOB.  Returns the actual length of the file read.
 */
ub4 read_file(filename, blob)
  char *filename;
  OCIBlobLocator *blob;
{
  long_varraw *lvr;
  ub4      bufsize;
  ub4      amt;
  ub4      filelen, remainder, nbytes;
  ub4      offset = 1;
  boolean  last = FALSE;
  FILE    *in_fd;

  /* Open the file for reading. */
  in_fd = fopen(filename, "r");
  if (in_fd == (FILE *)0)
    return (ub4)0;

  /* Determine Total File Length - Total Amount to Write to BLOB */
  (void) fseek(in_fd, 0L, SEEK_END);
```

```
amt = filelen = (ub4)ftell(in_fd);

/* Determine the Buffer Size and Allocate the LONG VARRAW Object */
bufsize = 2048;
lvr = (long_varraw *)malloc(sizeof(ub4) + bufsize);

nbytes = (filelen > bufsize) ? bufsize : filelen;

/* Reset the File Pointer and Perform the Initial Read */
(void) fseek(in_fd, 0L, SEEK_SET);
lvr->len = fread((void *)lvr->buf, (size_t)1, (size_t)nbytes, in_fd);
remainder = filelen - nbytes;

EXEC SQL WHENEVER SQLERROR DO sql_error("read_file():WRITE");

if (remainder == 0)
  {
    /* Write the BLOB in a Single Piece */
    EXEC SQL LOB WRITE ONE :amt
       FROM :lvr WITH LENGTH :nbytes INTO :blob AT :offset;
  }
else
  {
    /* Write the BLOB in Multiple Pieces using Standard Polling */
    EXEC SQL LOB WRITE FIRST :amt
       FROM :lvr WITH LENGTH :nbytes INTO :blob AT :offset;

    do {

      if (remainder > bufsize)
        nbytes = bufsize;
      else
        {
          nbytes = remainder;
          last = TRUE;
        }

      if ((lvr->len = fread(
            (void *)lvr->buf, (size_t)1, (size_t)nbytes, in_fd)) != nbytes)
        last = TRUE;

      if (last)
        {
          /* Write the Final Piece */
          EXEC SQL LOB WRITE LAST :amt
             FROM :lvr WITH LENGTH :nbytes INTO :blob;
        }
      else
        {
          /* Write an Interim Piece - Still More to Write */
          EXEC SQL LOB WRITE NEXT :amt
             FROM :lvr WITH LENGTH :nbytes INTO :blob;
        }

      remainder -= nbytes;

    } while (!last && !feof(in_fd));
  }

/* Close the file, and return the total file size. */
```

```
  fclose(in_fd);
  free(lvr);
  return filelen;
}


/* Generic error handler.  The 'routine' parameter should contain the name
   of the routine executing when the error occured.  This would be specified
   in the 'EXEC SQL WHENEVER SQLERROR DO sql_error()' statement.
 */
void sql_error(routine)
  char *routine;
{
  char message_buffer[512];
  size_t buffer_size;
  size_t message_length;

  /* Turn off the call to sql_error() to avoid a possible infinite loop */
  EXEC SQL WHENEVER SQLERROR CONTINUE;

  printf("¥nOracle error while executing %s!¥n", routine);

  /* Use sqlglm() to get the full text of the error message. */
  buffer_size = sizeof(message_buffer);
  sqlglm(message_buffer, &buffer_size, &message_length);
  printf("%.*s¥n", message_length, message_buffer);

  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}


/* Opens the binary file identified by 'filename' for writing, and copies
   the contents of the Binary LOB into it.
 */
void write_file(filename, blob)
  char *filename;
  OCIBlobLocator *blob;
{
  FILE        *out_fd;       /* File descriptor for the output file */
  ub4          amt;
  ub4          bufsize;
  long_varraw *lvr;

  /* Determine the Buffer Size and Allocate the LONG VARRAW Object */
  bufsize = 2048;
  lvr = (long_varraw *)malloc(sizeof(ub4) + bufsize);

  /* Open the output file for Writing */
  out_fd = fopen(filename, "w");
  if (out_fd == (FILE *)0)
    return;

  amt = 0;             /* Initialize for Standard Polling (Possibly) */
  lvr->len = bufsize;                       /* Set the Buffer Length */

  EXEC SQL WHENEVER SQLERROR DO sql_error("write_file():READ");

  /* READ the BLOB using a Standard Polling Loop */
  EXEC SQL WHENEVER NOT FOUND DO break;
```

```
  while (TRUE)
    {
      EXEC SQL LOB READ :amt FROM :blob INTO :lvr WITH LENGTH :bufsize;
      (void) fwrite((void *)lvr->buf, (size_t)1, (size_t)lvr->len, out_fd);
    }

  EXEC SQL WHENEVER NOT FOUND CONTINUE;

  /* Write the Final Piece (or First and Only Piece if not Polling) */
  (void) fwrite((void *)lvr->buf, (size_t)lvr->len, (size_t)1, out_fd);

  /* Close the Output File and Return */
  fclose(out_fd);
  free(lvr);
  return;
}



/* Inserts the binary file identified by file into the
 * executables table identified by key.
 */
void do_insert(key, file)
  varchar *key;
  char *file;
{
  OCIBlobLocator *blob;
  ub4 loblen, fillen;

  EXEC SQL ALLOCATE :blob;

  EXEC SQL WHENEVER SQLERROR DO sql_error("do_insert():INSERT/SELECT");

  EXEC SQL SAVEPOINT PREINSERT;
  EXEC SQL INSERT
    INTO executables (name, length, binary) VALUES (:key, 0, empty_blob());

  EXEC SQL SELECT binary INTO :blob
             FROM executables WHERE name = :key FOR UPDATE;

  printf(
    "Inserting file '%s' under key '%.*s'...¥n", file, key->len, key->arr);

  fillen = read_file(file, blob);
  EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :loblen;

  if ((fillen == 0) || (fillen != loblen))
    {
      printf("Problem reading file '%s'¥n", file);
      EXEC SQL ROLLBACK TO SAVEPOINT PREINSERT;
      EXEC SQL FREE :blob;
      return;
    }

  EXEC SQL WHENEVER SQLERROR DO sql_error("do_insert():UPDATE");
  EXEC SQL UPDATE executables
    SET length = :loblen, binary = :blob WHERE name = :key;

  EXEC SQL COMMIT WORK;
```

```
  EXEC SQL FREE :blob;
  EXEC SQL COMMIT;
  printf("Inserted.¥n");
}


/* Retrieves the executable identified by key into file */
void do_retrieve(key, file)
  varchar *key;
  char *file;
{
  OCIBlobLocator *blob;

  printf("Retrieving executable stored under key '%.*s' to file '%s'...¥n",
         key->len, key->arr, file);

  EXEC SQL ALLOCATE :blob;

  EXEC SQL WHENEVER NOT FOUND continue;
  EXEC SQL SELECT binary INTO :blob FROM executables WHERE name = :key;

  if (sqlca.sqlcode == NOT_FOUND)
    printf("Key '%.*s' not found!¥n", key->len, key->arr);
  else
    {
      write_file(file, blob);
      printf("Retrieved.¥n");
    }

  EXEC SQL FREE :blob;
}


/* Delete an executable from the database */
void do_delete(key)
  varchar *key;
{
  EXEC SQL WHENEVER SQLERROR DO sql_error("do_delete():DELETE");
  EXEC SQL DELETE FROM executables WHERE name = :key;

  if (sqlca.sqlcode == NOT_FOUND)
    printf("Key '%.*s' not found!¥n", key->len, key->arr);
  else
    printf("Deleted.¥n");
}


/* List all executables currently stored in the database */
void list_executables()
{
  char key[21];
  ub4  length;

  EXEC SQL WHENEVER SQLERROR DO sql_error("list_executables");

  EXEC SQL DECLARE key_cursor CURSOR FOR
    SELECT name, length FROM executables;

  EXEC SQL OPEN key_cursor;
```

```
  printf("¥nExecutables           Length (bytes)¥n");
  printf("--------------------  --------------¥n");

  EXEC SQL WHENEVER NOT FOUND DO break;
  while (1)
  {
    EXEC SQL FETCH key_cursor INTO :key, :length;
    printf("%s      %10d¥n", key, length);
  }

  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  EXEC SQL CLOSE key_cursor;

  printf("¥nTotal Executables: %d¥n", sqlca.sqlerrd[2]);
}


/* Prints the menu selections. */
void print_menu()
{
  printf("¥nSample 4 Menu.  Would you like to:¥n");
  printf("(I)nsert a new executable into the database¥n");
  printf("(R)etrieve an executable from the database¥n");
  printf("(L)ist the executables stored in the database¥n");
  printf("(D)elete an executable from the database¥n");
  printf("(Q)uit the program¥n¥n");
  printf("Enter i, r, l, or q: ");
}
```

関連項目

- LOB

5.4 Cプリプロセッサ

Pro*C/C++では、Cプリプロセッサのほとんどのディレクティブがサポートされます。Pro*C/C++プリプロセッサを使用すると、次のような作業を実行できます。

- **#define**ディレクティブを使用した定数およびマクロの定義、VARCHARなどのPro*C/C++のデータ型宣言をパラメータ化する場合の定義済エンティティの使用。
- **#include**ディレクティブを使用した、プリコンパイラに必要なsqlca.hなどのファイルの読取り。
- 個別のファイルにある定数およびマクロの定義、プリコンパイラでの**#include**ディレクティブを使用したこのファイルの読取り。

5.4.1 Pro*C/C++プリプロセッサの機能

Pro*C/C++プリプロセッサでは、Cプリプロセッサのほとんどのコマンドが認識され、必要なマクロ置換、ファイルのインクルードおよび条件付きのソース・テキストのインクルードまたは除外が効率的に実行されます。Pro*C/C++プリプロセッサでは、この事前処理で取得した値を使用して、ソース出力テキスト(生成される.c出力ファイル)が変更されます。

この点について、次に例を示します。次のプログラムの一部分を記述したとします。

```
#include "my_header.h"
...
VARCHAR name[VC_LEN];              /* a Pro*C-supplied datatype */
char    another_name[VC_LEN];              /* a pure C datatype */
```

```
...
```

カレント・ディレクトリ内のmy_header.hファイルに、特に次の行が含まれているとします。

```
#define VC_LEN   20
```

プリコンパイラはファイルmy_header.hを読み込み、VC_LENの定義済の値(20)を使用し、*name*の構造体をVARCHAR[20]として宣言します。

charはネイティブな型です。プリコンパイラは、another_name[VC_LEN]の宣言時に20を代入しません。

プリコンパイラはC言語のデータ型の宣言を処理する必要がないため、そのデータ型がホスト変数として指定されてもかまいません。実際にファイルmy_header.hをインクルードして、*another_name*の宣言内でVC_LENに20を代入する処理は、Cコンパイラのプリプロセッサの役割です。

5.4.2 プリプロセッサ・ディレクティブ

Pro*C/C++でサポートされるプリプロセッサ・ディレクティブは、次のとおりです。

- **#define**。プリコンパイラとCまたはC++コンパイラで使用されるマクロを作成します。
- **#include**。プリコンパイラで使用される他のソース・ファイルを読み込みます。
- **#if**。定数式が0に評価される場合にのみ、ソース・テキストをプリコンパイルおよびコンパイルします。
- **#ifdef**。定義済定数の有無に応じてソース・テキストを条件付きでプリコンパイルおよびコンパイルします。
- **#ifndef**。ソース・テキストを条件付きで除外します。
- **#endif**。**#if**、**#ifdef**または**#ifndef**コマンドを終了します。
- **#else**。 **#if** 、**#ifdef**または**#ifndef**の条件が満たされない場合に、プリコンパイルおよびコンパイルされるソース・テキストの代替部を選択します。
- **#elif**。定数またはマクロ引数の値に応じて、プリコンパイルおよびコンパイルされるソース・テキストの代替部を選択します。

5.4.2.1 無視されるディレクティブ

Pro*C/C++プリプロセッサで使用されないCプリプロセッサ・ディレクティブもあります。これらのディレクティブのほとんどは、プリコンパイラとは無関係です。たとえば、**#pragma**はCコンパイラ用のディレクティブであり、プリコンパイラでは処理されません。プリコンパイラで処理されないCプリプロセッサ・ディレクティブは、次のとおりです。

- **#**。プリプロセッサのマクロ・パラメータを文字定数に変換します。
- **##**。2つのプリプロセッサ・トークンを1つのマクロ定義にマージします。
- **#error**。コンパイル時エラー・メッセージを生成します。
- **#pragma**。実装依存情報をCコンパイラに渡します。
- **#line**。Cコンパイラ・メッセージに行番号を提供します。

これらのディレクティブは、Cコンパイラのプリプロセッサでサポートされている場合でも、Pro*C/C++では使用されません。これらのディレクティブのほとんどは、プリコンパイラでは使用されません。コンパイラでサポートされている場合は、これらのディレクティブをPro*C/C++プログラムで使用できますが、CまたはC++コード以外の埋込みSQL文や、プリコンパイラが提供するVARCHARなどのデータ型を使用した変数の宣言では使用できません。

5.4.3 ORA_PROCマクロ

Pro*C/C++では、ORA_PROCというCプリプロセッサのマクロが事前定義されています。このマクロを使用すると、不要または無関係なコード・セクションがプリコンパイラで処理されるのを回避できます。アプリケーションには、プリコンパイル時に必要のない情報を提供する大きなヘッダー・ファイルがインクルードされている場合があります。このようなヘッダー・ファイルは、ORA_PROCマクロに基づいて条件付きで除外すれば、プリコンパイラで読み込まれることはありません。

次の例では、ORA_PROCマクロを使用してirrelevant.hファイルを除外しています。

```
#ifndef  ORA_PROC
#include <irrelevant.h>
#endif
```

プリコンパイル時にORA_PROCが定義されているため、irrelevant.hファイルはインクルードされません。

ORA_PROCマクロを使用できる対象は、**#ifdef**や**#ifndef**などのCプリプロセッサ・ディレクティブのみです。EXEC ORACLE条件文では、Cプリプロセッサ・マクロと同じネームスペースは共有されません。したがって、次の例の条件では、事前定義済のORA_PROCマクロを使用しません。

```
EXEC ORACLE IFNDEF ORA_PROC;
   <section of code to be ignored>
EXEC ORACLE ENDIF;
```

この場合、この条件コード部分が正常に処理されるように、DEFINEオプションまたはEXEC ORACLE DEFINE文を使用してORA_PROCを設定する必要があります。

5.4.4 ヘッダー・ファイルの格納場所の指定

各システムのPro*C/C++プリコンパイラでは、sqlca.h、oraca.hおよびsqlda.hなど、プリプロセッサで読み込まれるヘッダー・ファイルが標準的な場所にあるものとみなされます。たとえば、ほとんどのUNIXシステムでの標準的な場所は、$ORACLE_HOME/precomp/publicです。使用しているシステムのデフォルトの場所については、システム固有のOracleマニュアルを参照してください。インクルードする必要のあるヘッダー・ファイルがデフォルトの場所にない場合は、コマンドラインで、EXEC ORACLEオプションとして、INCLUDE=オプションを指定する必要があります。

stdio.hやiostream.hなどのシステム・ヘッダー・ファイルについて、Pro*C/C++にハードコードされているものと異なる格納場所を指定するには、SYS_INCLUDEプリコンパイラ・オプションを使用します。

関連項目

- プリコンパイラのオプション

5.4.5 プリプロセッサの例

#defineコマンドを使用すると、名前付きの定数を作成して、ソース・コード内でマジック番号のかわりに使用できます。VARCHAR[const]など、プリコンパイラで必要な宣言に対して、**#define**で指定した定数を使用できます。たとえば、次のコードは不具合を含んでいる可能性があります。

```
...
VARCHAR  emp_name[10];
VARCHAR  dept_loc[14];
...
...
/* much later in the code ... */
f42()
{
    /* did you remember the correct size? */
```

```
    VARCHAR new_dept_loc[10];
   ...
}
```

このコードのかわりに、次のように記述できます。

```
#define ENAME_LEN     10
#define LOCATION_LEN  14
VARCHAR  new_emp_name[ENAME_LEN];
   ...
/* much later in the code ... */
f42()
{
    VARCHAR new_dept_loc[LOCATION_LEN];
   ...
}
```

引数を持つプリプロセッサ・マクロは、Cオブジェクトで使用する場合と同様に、プリコンパイラで処理する必要のあるオブジェクトで使用できます。次に例を示します。

```
#define ENAME_LEN    10
#define LOCATION_LEN 14
#define MAX(A,B)  ((A) > (B) ? (A) : (B))


   ...
f43()
{
    /* need to declare a temporary variable to hold either an
       employee name or a department location */
    VARCHAR  name_loc_temp[MAX(ENAME_LEN, LOCATION_LEN)];
   ...
}
```

#include、**#ifdef**および**#endif**プリプロセッサ・ディレクティブを使用して、プリコンパイラに必要なファイルを条件付きでインクルードできます。次に例を示します。

```
#ifdef ORACLE_MODE
#include <sqlca.h>
#else
    long SQLCODE;
#endif
```

5.4.5.1 #defineの使用について

Pro*C/C++での**#define**プリプロセッサ・ディレクティブの使用には、制限があります。**#define**ディレクティブを使用して、実行SQL文に使用する記号定数を生成することはできません。次の無効な例を参照してください。

```
#define RESEARCH_DEPT   40
...
EXEC SQL SELECT empno, sal
    INTO :emp_number, :salary /* host arrays */
    FROM emp
    WHERE deptno = RESEARCH_DEPT;  /* INVALID! */
```

#defineしたマクロを有効に使用できる宣言SQL文は、TYPE文とVAR文のみです。したがって、次のマクロ使用例はPro*C/C++で有効です。

```
#define STR_LEN      40
...
```

```
typedef char asciiz[STR_LEN];
...
EXEC SQL TYPE asciiz IS STRING(STR_LEN) REFERENCE;
...
EXEC SQL VAR password IS STRING(STR_LEN);
```

5.4.5.2 他のプリプロセッサの制限

プリプロセッサでは、ディレクティブ**#**および **##** が無視され、プリコンパイラで認識する必要のあるトークンが作成されます。これらのコマンドは(コンパイラでサポートされている場合は)、純粋なCコードに使用できます。プリコンパイラでは、これらのコードは処理されません。次の例の場合、プリプロセッサ・コマンド**##**の使用は無効です。

```
#define MAKE_COL_NAME(A)    col ## A
...
EXEC SQL SELECT MAKE_COL_NAME(1), MAKE_COL_NAME(2)
    INTO :x, :y
    FROM table1;
```

プリコンパイラでは**##**は無視されるため、この例は正しくありません。

5.4.6 #includeに使用できないSQL文

Pro*C/C++プリプロセッサで**#include**ディレクティブが処理される方法については前項で説明しましたが、この方法のために、**#include**ディレクティブを使用して埋込みSQL文を含むファイルをインクルードできません。**#include**は、純粋に宣言文とディレクティブのみを含むファイルをインクルードする場合に使用します。**#define**や、プリコンパイラに必要な変数と構造体の宣言のみを含むsqlca.hファイルなどがその例です。

5.4.7 SQLCA、ORACAおよびSQLDAの組込み

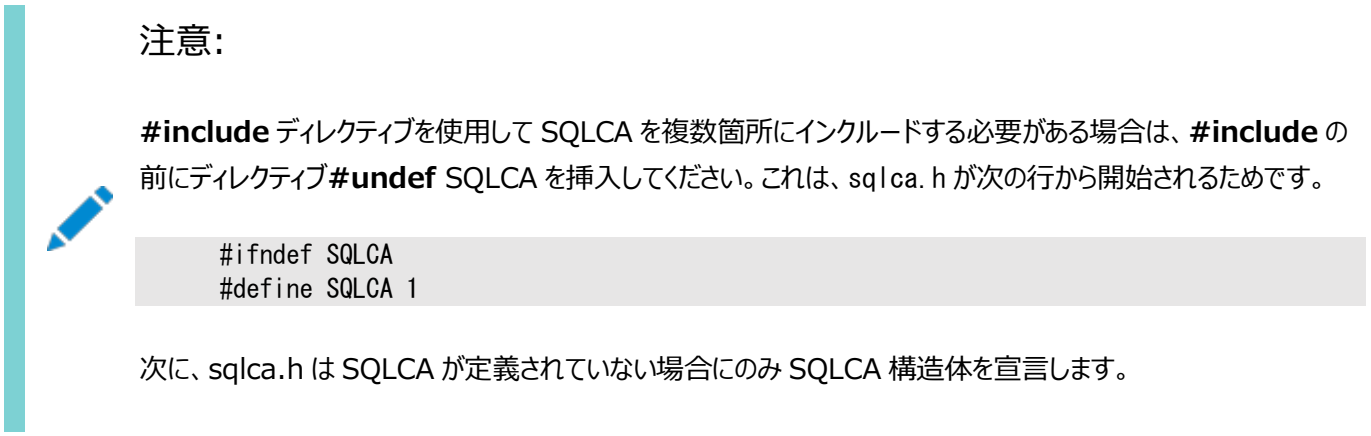
C/C++プリプロセッサの**#include**コマンド、あるいはプリコンパイラのEXEC SQL INCLUDEコマンドを使用すると、sqlca.h、oraca.hおよびsqlda.hの各宣言ヘッダー・ファイルをPro*C/C++プログラムにインクルードできます。たとえば、次の文のように、EXEC SQLオプションを指定して、SQL通信領域構造体(SQLCA)をプログラムにインクルードできます。

```
EXEC SQL INCLUDE sqlca;
```

C/C++プリプロセッサ・ディレクティブを使用してSQLCAをインクルードするには、次のコードを追加します。

```
#include <sqlca.h>
```

プリプロセッサの**#include**ディレクティブを使用する場合は、ファイル拡張子(.hなど)を指定する必要があります。



注意:

#include ディレクティブを使用して SQLCA を複数箇所にインクルードする必要がある場合は、**#include** の前にディレクティブ**#undef** SQLCA を挿入してください。これは、sqlca.h が次の行から開始されるためです。

```
#ifndef SQLCA
#define SQLCA 1
```

次に、sqlca.h は SQLCA が定義されていない場合にのみ SQLCA 構造体を宣言します。

#includeディレクティブまたはEXEC SQL INCLUDE文を含むファイルをプリコンパイルする場合は、インクルードするすべてのファイルの位置をプリコンパイラに対して指定する必要があります。INCLUDE=オプションは、コマンドライン、システム構成ファイルまたはユーザー構成ファイル内で使用できます。

sqlca.h、oraca.hおよびsqlda.hなど、標準的なプリプロセッサ・ヘッダー・ファイルのデフォルトの場所は、プリコンパイラ内で事前設定されています。この場所は、システムごとに異なります。使用しているシステムのデフォルトの場所については、システム固有のOracleマニュアルを参照してください。

Pro*C/C++で生成される.c出力ファイルをコンパイルする場合は、コンパイラおよびオペレーティング・システムに用意されているオプションを使用して、インクルード・ファイルの場所を識別する必要があります。

たとえば、ほとんどのUNIXシステムでは、次のコマンドを使用して、生成されるCソース・ファイルをコンパイルできます。

```
cc -o progname -I$ORACLE_HOME/sqllib/public ... filename.c ...
```

VAX/OPENVMSシステムでは、事前にインクルード・ディレクトリ・パスを論理VAXC$INCLUDE内の値に設定します。

関連項目

- ランタイム・エラーの処理
- プリコンパイラのオプション

5.4.8 EXEC SQL INCLUDEおよび#includeのまとめ

プログラム内でEXEC SQL INCLUDE文を使用すると、プリコンパイラによりソース・テキストが出力(.c)ファイルにインクルードされます。したがって、EXEC SQL INCLUDEを使用してインクルード・ファイルに、宣言文および埋込みSQLの実行文を入れることができます。

#includeを使用してファイルをインクルードする場合、プリコンパイラでは単にファイルが読み込まれ、**#define**で定義したマクロが追跡記録されるのみです。

> 注意:
>
> VARCHAR 宣言と SQL 文は、インクルード(#include)ファイルには使用できません。このため、Pro*C/C++プリプロセッサの#include ディレクティブを使用してインクルード・ファイルでは、SQL 文を使用できません。

5.4.9 定義済マクロ

Cコンパイラのコマンドラインでマクロを定義する場合、アプリケーションの要件によっては、そのマクロをプリコンパイラのコマンドラインでも定義する必要があります。たとえば、UNIXコマンドラインで次のようにコンパイルするとします。

```
cc -DDEBUG ...
```

この場合は、次のようにDEFINE=オプションを使用してプリコンパイルする必要があります。

```
proc DEFINE=DEBUG ...
```

5.4.10 インクルード・ファイル

プリコンパイルする必要のあるすべてのインクルード・ファイルの位置は、コマンドラインまたは構成ファイルで指定する必要があります。

たとえば、UNIX環境での開発中に、アプリケーションのインクルード・ファイルがディレクトリ/home/project42/includeにある場合は、このディレクトリをPro*C/C++のコマンドラインとccコマンドラインの両方で指定する必要があります。次のようなコマンドを使用します。

```
proc iname=my_app.pc include=/home/project42/include ...
cc -I/home/project42/include ... my_app.c
```

または、適切なマクロを*Make*ファイルに組み込みます。Pro*C/C++アプリケーションのコンパイルとリンクの詳細は、システム固有のOracleマニュアルを参照してください。

関連項目

- INCLUDE

5.5 プリコンパイル済ヘッダー・ファイル

プリコンパイル済ヘッダー・ファイルを使用すると、多数の#include文を含むヘッダー・ファイルをプリコンパイルすることで、時間とリソースを節約できます。この機能を使用する手順は次の2つです。

- まず、プリコンパイル済ヘッダー・ファイルを作成します。
- このプリコンパイル済のヘッダーが、次回以降のアプリケーションのプリコンパイル時に自動的に使用されます。

この機能は、多数のモジュールで構成される大型アプリケーションに使用してください。

プリコンパイラ・オプションをHEADER=hdrに設定すると、次のように指定されます。

- プリコンパイル済のヘッダーを使用します。
- 生成される出力ファイルのファイル拡張子はhdrです。

このオプションを入力できるのは、構成ファイルまたはコマンドラインのみです。HEADERにはデフォルト値はありませんが、入力ヘッダーの拡張子は.hである必要があります。

5.5.1 プリコンパイル済ヘッダー・ファイルの作成

ヘッダー・ファイルtop.hを作成するとします。このファイルは、HEADER=hdrを指定してプリコンパイルできます。

```
proc HEADER=hdr INAME=top.h
```

注意:

拡張子は.h にする必要があります。INAME 値には、「/」や「..」などの絶対パス要素や相対パス要素は使用できません。

Pro*C/C++により、指定した入力ファイルtop.hがプリコンパイルされ、同じディレクトリに新しいプリコンパイル済ヘッダー・ファイルtop.hdrが生成されます。出力ファイルtop.hdrは、#include文による検索対象となるディレクトリに移動できます。

注意:

出力ファイル名の指定には ONAME オプションを使用しないでください。HEADER とともに使用すると無視さ

れます。

5.5.2 プリコンパイル済ヘッダー・ファイルの使用

HEADERオプションには、プリコンパイル対象のアプリケーション・ファイルと同じ値を使用してください。simple.pcに次のファイルが含まれているとします。

```
#include <top.h>
...
```

また、top.hに次のファイルが含まれているとします。

```
#include <a.h>
#include <b.h>
#include <c.h>
...
```

次の方法でプリコンパイルします。

```
proc HEADER=hdr INAME=simple.pc
```

Pro*C/C++では、#include top.h文の読込み時に対応するtop.hdrファイルが検索され、top.hが再度プリコンパイルされるかわりに、そのファイルからデータがインスタンス化されます。

注意:

プリコンパイル済ヘッダー・ファイルは、常に入力ヘッダー・ファイルのかわりに使用されます。入力(.h)ファイルがインクルード・ディレクトリの標準検索階層の最初に表示されている場合も同様です。

5.5.3 例

この項の例では、いくつかの異なるケースを示します。

5.5.3.1 冗長なファイル・インクルード

次の2つのケースでは、考えられる2つの冗長なファイル・インクルードを示します。

5.5.3.1.1 ケース1: 最上位のヘッダー・ファイルのインクルード

プリコンパイル済ヘッダー・ファイルは、#includeディレクティブを使用してインクルードされた回数に関係なく、一度のみインスタンス化されます。

前述の例と同様に、HEADERの値をhdrに設定し、最上位のヘッダー・ファイルtop.hをプリコンパイルするとします。次に、そのヘッダー・ファイルに対して、複数の#includeディレクティブをプログラムに記述します。

```
#include <top.h>
#include <top.h>
main() {}
```

top.hの最初の#includeが発生すると、プリコンパイル済ヘッダー・ファイルtop.hdrがインスタンス化されます。同じヘッダー・

ファイルの2番目のインクルードは、冗長であるために無視されます。

5.5.3.1.2 ケース2: ネストされたヘッダー・ファイルのインクルード

ファイルa.hに次の文が含まれているとします。

```
#include <b.h>
```

前述の例と同様にHEADERを指定して、そのヘッダー・ファイルをプリコンパイルします。Pro*C/C++では、a.hとb.hの両方がプリコンパイルされ、a.hdrが生成されます。

次に、このPro*C/C++プログラムをプリコンパイルするとします。

```
#include <a.h>
#include <b.h>
main() {}
```

a.hの#includeが発生すると、a.hが再度プリコンパイルされるかわりに、プリコンパイル済ヘッダー・ファイルa.hdrがインスタンス化されます。このインスタンス化には、b.hの内容全体も含まれます。

b.hはa.hのプリコンパイルに含まれ、a.hdrはインスタンス化されているため、プログラムに指定されているb.hの後続の#includeは冗長になり、無視されます。

5.5.3.2 複数のプリコンパイル済ヘッダー・ファイル

Pro*C/C++では、1回のプリコンパイルで複数の異なるプリコンパイル済ヘッダー・ファイルをインスタンス化できます。ただし、複数のプリコンパイル済ヘッダー・ファイルが共通のヘッダー・ファイルを共有している場合は、次の点に注意してください。

たとえば、topA.hに次の行が含まれているとします。

```
#include <a.h>
#include <c.h>
```

また、topB.hには次の行が含まれているとします。

```
#include <b.h>
#include <c.h>
```

topA.hおよびtopB.hの両方で、同じ共通ヘッダー・ファイルc.hがインクルードされています。同じHEADER値を指定してtopA.hとtopB.hをプリコンパイルすると、topA.hdrおよびtopB.hdrが生成されます。しかし、両方にc.hの内容全体が含まれています。

次のようなPro*C/C++プログラムがあるとします。

```
#include <topA.h>
#include <topB.h>
main() {}
```

プリコンパイル済ヘッダー・ファイルtopA.hdrおよびtopB.hdrの両方が、前述の例と同様にインスタンス化されます。ただし、両方で共通ヘッダー・ファイルc.hが共有されるため、そのファイルの内容は2回インスタンス化されます。

Pro*C/C++では、プリコンパイル済ヘッダー・ファイル間のファイルの共有を判断できません。各プリコンパイル済ヘッダー・ファイルには、一意のヘッダー・セットをインクルードします。ヘッダーの共有はできるだけ避けてください。共有すると、プリコンパイルが低速になり、メモリー使用量が増加するため、プリコンパイル済ヘッダー・ファイルを使用する意味がなくなります。

5.5.4 ヘッダー・ファイルのリスト

ORACLE_BASE¥*ORACLE_HOME*¥precomp¥publicディレクトリには、Pro*C/C++ヘッダー・ファイルが含まれています。表5-3は、

ヘッダー・ファイルのリストと説明です。

表5-3 ヘッダー・ファイル

ヘッダー・ファイル	説明
oraca.h	Oracle 通信領域(ORACA)が含まれています。これは、ランタイム・エラーの診断と、プログラムでの様々な Oracle Database 10*g* リソースの使用の監視に役立ちます。
sql2oci.h	SQLLIB 関数が含まれています。これらの関数により、Oracle Call Interface(OCI)環境ハンドルと OCI サービス・コンテキストを Pro*C/C++アプリケーションで取得できます。
sqlapr.h	OCI とともに使用できる外部関数用の ANSI プロトタイプが含まれています。
sqlca.h	ランタイム・エラーの診断に役立つ SQL コミュニケーション領域(SQLCA)が含まれています。SQLCA は、実行可能な SQL 文が実行されるたびに更新されます。
sqlcpr.h	Pro*C/C++で生成される、SQLLIB 関数用のプラットフォーム固有の ANSI プロトタイプが含まれています。デフォルトでは、Pro*C/C++は、SQL プログラミング・コールの完全な関数プロトタイプをサポートしていません。この機能が必要な場合、アプリケーション・ソース・ファイル内のすべての EXEC SQL 文の前に sqlcpr.h を含める必要があります。
oraca.h	Oracle 通信領域(ORACA)が含まれています。これは、ランタイム・エラーの診断と、プログラムでの様々な Oracle Database 10*g* リソースの使用の監視に役立ちます。
sql2oci.h	SQLLIB 関数が含まれています。これらの関数により、Oracle Call Interface(OCI)環境ハンドルと OCI サービス・コンテキストを Pro*C/C++アプリケーションで取得できます。
sqlapr.h	OCI とともに使用できる外部関数用の ANSI プロトタイプが含まれています。

5.5.5 オプションの効果

アプリケーションのプリコンパイル時には、次のプリコンパイラ・オプションを使用できます。

5.5.5.1 DEFINEおよびINCLUDEオプション

プリコンパイル済ヘッダーを使用してプリコンパイルするときには、DEFINEとINCLUDEの値を、プリコンパイル済ヘッダー・ファイルの作成時と同じ値にする必要があります。DEFINEまたはINCLUDEの値を変更した場合は、プリコンパイル済ヘッダー・ファイルを再作成する必要があります。

開発環境を変更した場合も、プリコンパイル済ヘッダー・ファイルを再作成する必要があります。

5.5.5.1.1 シングル・ユーザーの場合

シングル・ユーザーの場合を考えてみます。DEFINEまたはINCLUDEオプションの値を変更した場合、プリコンパイル済ヘッダー・ファイルの内容は、その後に実行されるPro*C/C++プリコンパイルでは正常に使用できなくなります。

DEFINEおよびINCLUDE、DEFINEまたはINCLUDEオプションの値を変更しているため、プリコンパイル済ヘッダー・ファイルの内容は、#includeディレクティブの対応する.hファイルが正常に処理された場合の標準的な結果と一致しなくなります。

つまり、DEFINEおよびINCLUDE、DEFINEまたはINCLUDEオプションの値を変更した場合は、プリコンパイル済ヘッダー・ファイルを再作成し、それを使用するPro*C/C++プログラムを再度プリコンパイルする必要があります。

関連項目

- DEFINE
- INCLUDE

5.5.5.1.2 マルチ・ユーザーの場合

AおよびBという2人のユーザーがいる場合を考えてみます。AとBはまったく異なる環境で開発しているため、DEFINEおよびINCLUDEオプションの値もまったく異なっています。

ユーザーAは、共通ヘッダー・ファイルcommon.hをプリコンパイルし、プリコンパイル済ヘッダー・ファイルcommon.hdrAを作成します。ユーザーBも同じヘッダー・ファイルをプリコンパイルして、common.hdrBを作成します。ただし、両者の環境が異なるため、2人のユーザーが使用したDEFINEおよびINCLUDEオプションの値も異なります。そのため、ユーザーAとBが作成したcommon.hdrの内容が同じになるとはかぎりません。

コードは次のようになります。

```
A> proc HEADER=hdrA DEFINE=<A macros> INCLUDE=<A dirs> common.h

B> proc HEADER=hdrB DEFINE=<B macros> INCLUDE=<B dirs> common.h
```

異なる環境で作成されたため、生成されたプリコンパイル済ヘッダー・ファイルcommon.hdrAはcommon.hdrBと同等でない場合があります。つまり、ユーザーAとユーザーBが、相手の作成したcommon.hdrを使用しても、それぞれの開発環境でPro*C/C++プログラムが正常にプリコンパイルされるとはかぎりません。

したがって、プリコンパイル済ヘッダー・ファイルを、異なるユーザー間および異なる開発環境間で共有または交換する場合には、注意が必要です。

5.5.5.2 CODEおよびPARSEオプション

Pro*C/C++では、hppやh++などの拡張子が付いたC++ヘッダー・ファイルは検索されません。このため、ヘッダー・ファイルのプリコンパイル時には、CODE=CPPを使用しないでください。アプリケーションのプリコンパイル時にCPP値を使用できるのは、ソース・コードに.hヘッダー・ファイルのみが含まれる場合にかぎります。

プリコンパイル済ヘッダー・ファイルの作成時、あるいはモジュールのプリコンパイル時には、PARSEオプションに使用できる値はFULLまたはPARTIALのみです。値FULLは、PARTIALより高い値とみなされます。モジュールのプリコンパイル時に使用するPARSEの値は、プリコンパイル済ヘッダー・ファイルの作成時の値以下にする必要があります。

> 注意:
>
> PARSE=FULL を指定してプリコンパイル済ヘッダー・ファイルをプリコンパイルしてから、PARSE=PARTIAL を指定してモジュールをプリコンパイルする場合は、ホスト変数を宣言部中で宣言する必要があります。C++コードは、PARSE=PARTIAL を指定した場合にのみ認識されます。

PARTIALに次のPARSEオプションを指定してヘッダー・ファイルをプリコンパイルするとします。

```
proc HEADER=hdr PARSE=PARTIAL file.h
```

次に、PARSEをFULLに設定し、そのヘッダー・ファイルを含むプログラムをプリコンパイルします。

```
proc HEADER=hdr PARSE=FULL program.pc
```

ファイル.hはPARSEオプションにPARTIALを設定してプリコンパイルしたため、一部のヘッダー・ファイルは処理されていません。このため、未処理部分が参照された場合は、Pro*C/C++プログラムのプリコンパイル時にエラーが発生する可能性があります。

具体例として、ファイル.hに次のコードが含まれているとします。

```
#define LENGTH 10
typedef int myint;
```

program.pcに、次の小さいプログラムが含まれているとします。

```
#include <file.h>
main()
{
     VARCHAR ename[LENGTH];
     myint empno = ...;
     EXEC SQL SELECT ename INTO :ename WHERE JOB = :empno;
}
```

ファイル.hのプリコンパイル時にPARSEがPARTIALに設定されているため、typedefは処理されずにLENGTHマクロのみが処理されます。

VARCHARの宣言と宣言以後のホスト変数としての使用は、正常に行われます。ただし、Pro*C/C++ではmyint型宣言が処理されないため、empnoホスト変数を使用できません。

PARSEオプションをFULLに設定してヘッダー・ファイルをプリコンパイルしてから、PARSEオプションをPARTIALに設定してプログラムをプリコンパイルすると、正常に動作します。ただし、ホスト変数は明示的な宣言部中で宣言する必要があります。

関連項目

- コードの解析について
- CODE
- PARSE

5.5.6 使用上の注意

プリコンパイル済のヘッダーから生成された出力ファイルの形式は、リリースごとに異なる可能性があります。Pro*C/C++では、プリコンパイル済ヘッダー・ファイルの出力の生成に使用されたプリコンパイラのバージョンを判断できません。

このため、プリコンパイル済ヘッダー・ファイルを使用したプリコンパイル時に、エラーまたは他の予期しない動作を回避するために、Pro*C/C++のリリースのアップグレード時には、対応するヘッダー・ファイルを再度プリコンパイルしてファイルを再生成することをお薦めします。

ヘッダー・ファイルをプリコンパイルして生成された出力ファイルには、移植性がありません。つまり、ヘッダー・ファイルのプリコンパイルにより生成された出力ファイルは、プラットフォーム間で転送できず、別のヘッダー・ファイルまたはPro*C/C++プログラムのプリコンパイル時に使用できません。

5.6 Oracleプリプロセッサ

コードの条件文は、環境と実行する処理を定義するEXEC ORACLEディレクティブでマークされます。これらの条件文には、Cの文、埋込みSQL文およびディレクティブを記述できます。次のEXEC ORACLEディレクティブでは、プリコンパイルの条件を制御できます。

```
EXEC ORACLE DEFINE symbol;    -- define a symbol
EXEC ORACLE IFDEF symbol;     -- if symbol is defined
```

```
EXEC ORACLE IFNDEF symbol;    -- if symbol is not defined
EXEC ORACLE ELSE;             -- otherwise
EXEC ORACLE ENDIF;            -- end this block
```

すべてのEXEC ORACLE文の終わりには、セミコロンを付ける必要があります。

5.6.1 記号の定義

シンボルを定義するには2通りの方法があります。1つの方法では、ホスト・プログラムに次の文を含めます。

```
EXEC ORACLE DEFINE symbol;
```

もう1つは、次の構文を使用してコマンドラインでシンボルを定義する方法です。

```
... INAME=filename ... DEFINE=symbol
```

*symbol*の部分は大/小文字区別がありません。

注意:

#define プリプロセッサ・ディレクティブは、EXEC ORACLE DEFINE コマンドとは異なります。

Pro*C/C++プリコンパイラをシステムにインストールするときに、ポート固有の記号がいくつか事前定義されます。

5.6.2 Oracleプリプロセッサの例

次の例では、記号*site2*が定義されている場合にのみ、SELECT文がプリコンパイルされます。

```
EXEC ORACLE IFDEF site2;
    EXEC SQL SELECT DNAME
        INTO :dept_name
        FROM DEPT
        WHERE DEPTNO = :dept_number;
EXEC ORACLE ENDIF;
```

次の例に示すように条件ブロックはネストできます。

```
EXEC ORACLE IFDEF outer;
    EXEC ORACLE IFDEF inner;
    ...
    EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

Cまたは埋込みSQLコードをIFDEFとENDIFの間に記述し、シンボルを定義しないことで、そのコードをコメント行にすることができます。

5.7 数値定数の評価

以前のPro*C/C++では、ホスト変数(charまたはVARCHARなど)のサイズの宣言時に使用できるのは、数値リテラルや数値リテラルを含む単純な定数式のみでした。次に例を示します。

```
   #define LENGTH 10
   VARCHAR v[LENGTH];
   char c[LENGTH + 1];
```

現在は、次のような数値定数の宣言も使用できます。

```
const int length = 10;
VARCHAR v[length];
char c[length + 1];
```

このような定数宣言をサポートするANSIコンパイラやC++コンパイラを使用しているプログラマにとって、この機能は最適です。

従来のPro*C/C++では、評価可能な定数式の値の評価は常に実行されてきましたが、数値定数はどのような定数式にも宣言できませんでした。

Pro*C/C++では、マクロが数値リテラルに展開されていれば、通常の数値リテラルやマクロを使用する位置であれば、任意の位置で数値定数を宣言できます。

これは主に、SQL文で使用するバインド変数の配列のサイズを宣言する場合に使用されます。

5.7.1 Pro*C/C++での数値定数

Pro*C/C++では、数値定数が宣言された位置を検索する場合は、C言語の標準のスコープ規則が使用されます。

```
const int g = 30;     /* Global declaration to both function_1()
                                              and function_2() */
void function_1()
{
  const int a = 10;  /* Local declaration only to function_1() */
  char x[a];
  exec sql select ename into :x from emp where job = 'PRESIDENT';
}

void function_2()
{
  const int a = 20;  /* Local declaration only to function_2() */
  VARCHAR v[a];
  exec sql select ename into :v from emp where job = 'PRESIDENT';
}

void main()
{
  char m[g];                                  /* The global g */
  exec sql select ename into :m from emp where job = 'PRESIDENT';
}
```

5.7.2 数値定数の規則および例

特定の静的な型を持つ変数は、**static**で定義し、初期化する必要があります。Pro*C/C++で数値定数を宣言する場合は、必ず次の規則に従ってください。

- 定数の宣言時に**const**修飾子を指定します。
- 定数の値の初期化時に初期化機能を使用します。この初期化機能は、プリコンパイル時に評価可能である必要があります。

有効な初期化機能が指定された定数宣言で解決できない識別子を使用すると、エラーとみなされます。

次の例は、無効な指定方法とその指定が許可されない理由を示しています。

```
int a;
int b = 10;
volatile c;
```

```
volatile d = 10;
const e;
const f = b;

VARCHAR v1[a]; /* No const qualifier, missing initializer */
VARCHAR v2[b];                        /* No const qualifier */
VARCHAR v3[c];     /* Not a constant, missing initializer */
VARCHAR v4[d];                             /* Not a constant */
VARCHAR v5[e];                        /* Missing initializer */
VARCHAR v6[f];   /* Bad initializer.. b is not a constant */
```

5.8 OCIリリース8のSQLLIB拡張相互運用性

OCI環境ハンドルは、*sql_context*型のPro*C/C++ランタイム・コンテキストに関連付けられます。つまり、アプリケーション実行時にSQLLIBに保持されるPro*C/C++ランタイム・コンテキストは、複数のOCI環境ハンドルに関連付けることはできません。Pro*C/C++ランタイム・コンテキストごとに複数のデータベース接続を行うことができ、各ランタイム・コンテキストはそのOCI環境ハンドルに関連付けられます。

注意:

プリコンパイラ・アプリケーションは OCI ハンドルを抽出し、OCI 関数を直接呼び出すことができます。ただし、プリコンパイラは戻される可能性のある「まだ実行中」エラーを処理できないので、非ブロックモードはサポートされていません。

5.8.1 OCIリリース8環境でのランタイム・コンテキスト

EXEC SQL CONTEXT USE文は、Pro*C/C++プログラムで使用するランタイム・コンテキストを指定します。このコンテキストは、所定のPro*C/C++ファイル内でそのEXEC SQL CONTEXT USE文から次に指定されたEXEC SQL CONTEXT USE文までの、SQL実行文すべてに適用されます。ソース・ファイルにEXEC SQL CONTEXT USEがない場合、デフォルトのグローバル・コンテキストとみなされます。このため、カレント・ランタイム・コンテキストとそれに対応付けられているカレントOCI環境ハンドルは、プログラム内のどの場所にあっても認識されます。

Pro*C/C++では、EXEC SQL CONNECTを使用してデータベースへのログオンが実行されると、ランタイム・コンテキストとそれに対応付けられているOCI環境ハンドルが初期化されます。

EXEC SQL CONTEXT FREE文を使用してPro*C/C++ランタイム・コンテキストを解放すると、それに対応付けられているOCI環境ハンドルが終了し、各種OCIハンドルやLOBロケータに割り当てられている領域など、すべてのリソースが割当て解除されます。このコマンドにより、Pro*C/C++ランタイム・コンテキストに対応付けられた他のすべてのメモリーがすべて解放されます。デフォルトのグローバル・ランタイム・コンテキストに確立されるOCI環境ハンドルは、Pro*C/C++プログラムが終了するまで割り当てられたままになっています。

5.8.2 OCIリリース8環境ハンドルのパラメータ

Pro*C/C++で確立されるOCI環境は、次のパラメータを使用します。

- メモリーの割当て、メモリーの解放、テキスト・ファイルへの書込みおよび出力バッファのフラッシュについて、その環境で使用されるコールバック関数は、それぞれmalloc()、free()、fprintf(stderr, ...)およびfflush(stderr)をコールする通常の関数です。

- 言語は、グローバリゼーション・サポート変数NLS_LANGから取得されます。
- エラー・メッセージ・バッファは、スレッド固有の記憶域に割り当てられます。

5.9 OCIリリース8へのインタフェース

SQLLIBライブラリには、Pro*C/C++プログラムで確立したデータベース接続に対して、OCI環境ハンドルおよびサービス・コンテキスト・ハンドルを取得するためのルーチンが用意されています。OCIハンドルを取得すると、ユーザーは様々なOCIルーチンをコールできます。たとえば、クライアント側でDATE算術を実行したり、オブジェクト側でナビゲーション操作を実行したりできます。これらのSQLLIB関数については後述します。これらの関数のプロトタイプは、パブリック・ヘッダー・ファイルsql2oci.hに用意されています。

埋込みSQLと他のOracleプログラム・インタフェースのコールを混在させるPro*C/C++ユーザーは、十分に注意する必要があります。たとえば、ユーザーがOCIインタフェースを使用して直接接続を終了すると、SQLLIBが非同期状態となります。このような場合、Pro*C/C++プログラム内の後続のSQL文の動作は未定義になります。

注意:

Pro*C/C++、Oracle Call Interface(OCI)リリース 8 および XA には互換性がありません。

Oracle OCIとの相互運用性を提供する次の新しいSQLLIB関数がヘッダー・ファイルsql2oci.h内で宣言されています。

- SQLEnvGet()。所定のSQLLIBランタイム・コンテキストに関連付けられたOCI環境ハンドルへのポインタを戻します。単一サーバー環境と共有サーバー環境の両方で使用できます。
- SQLSvcCtxGet()。Pro*C/C++データベース接続用のOCIサービス・コンテキスト・ハンドルを戻します。単一サーバー環境と共有サーバー環境の両方で使用できます。
- シングル・スレッド・ランタイム・コンテキストを使用するときに、どちらかの関数の最初のパラメータとしてsql2oci.hをインクルードする場合は、(dvoid *)0として定義されている定数SQL_SINGLE_RCTXを渡します。

関連項目

- オブジェクト

5.9.1 SQLEnvGet()

SQLLIBライブラリ関数SQLEnvGet()(SQLLIBによるOCI環境の取得)を指定すると、所定のSQLLIBランタイム・コンテキストに関連付けられているOCI環境ハンドルへのポインタが戻されます。この関数のプロトタイプは、次のとおりです。

```
sword SQLEnvGet(dvoid *rctx, OCIEnv **oeh);
```

各パラメータの意味は次のとおりです。

項目	説明
説明	*oeh* をランタイム・コンテキストに対応する OCIEnv に設定します。
パラメータ	*rctx* (IN) = SQLLIB ランタイム・コンテキストへのポインタ。

項目	説明
	oeh (OUT) = OCIEnv へのポインタ。
戻り値	成功した場合は SQL_SUCCESS。 失敗した場合は SQL_ERROR。
注意	Pro*C/C++での通常のエラー状況変数(SQLCA、SQLSTATE など)は、この関数をコールしても影響を受けません。

5.9.2 SQLSvcCtxGet()

SQLLIBライブラリ関数SQLSvcCtxGet()(SQLLIBによるOCIサービス・コンテキストの取得)を指定すると、Pro*C/C++データベース接続用のOCIサービス・コンテキストが戻されます。OCIサービス・コンテキストを使用して、OCI関数を直接コールできます。この関数のプロトタイプは、次のとおりです。

```
sword SQLSvcCtxGet(dvoid *rctx, text *dbname,
       sb4 dbnamelen, OCISvcCtx **svc);
```

各パラメータの意味は次のとおりです。

項目	説明
説明	*svc* をランタイム・コンテキストに対応する OCI サービス・コンテキストに設定します。
パラメータ	*rctx* (IN) = SQLLIB ランタイム・コンテキストへのポインタ。 *dbname* (IN) = この接続の論理名を含むバッファ。 *dbnamelen* (IN) = dbname バッファの長さ。 *svc* (OUT) = OCISvcCtx ポインタのアドレス。
戻り値	成功した場合は SQL_SUCCESS。 失敗した場合は SQL_ERROR。
注意	1. Pro*C/C++での通常のエラー状況変数(SQLCA、SQLSTATE など)は、この関数をコールしても影響を受けません。 2. *dbname* は、埋込み SQL 文の AT 句に使用されている識別子と同じです。 3. *dbname* が NULL ポインタであるか、*dbnamelen* が 0 の場合は、SQL 文に AT 句が指定されていない場合と同様に、デフォルトのデータベース接続とみなされます。 4. *dbnamelen* の値が-1 の場合は、*dbname* が 0 で終了する文字列であることを示

項目	説明
	します。

5.9.3 OCIリリース8コールの埋込み

OCIリリース8コールをPro*C/C++プログラムに埋め込む手順は、次のとおりです。

1. パブリック・ヘッダーsql2oci.hを組み込みます。

2. Pro*C/C++プログラム内で環境ハンドル(OCIEnv *型)を次のように宣言します。

```
OCIEnv *oeh;
```

3. コールするOCI関数がサービス・コンテキスト・ハンドルを必要とする場合は、オプションとして、Pro*C/C++プログラム内でサービス・コンテキスト・ハンドル(OCISvcCtx *型)を宣言します。

```
OCISvcCtx *svc;
```

4. Pro*C/C++プログラム内でエラー・ハンドル(OCIError *型)を宣言します。

```
OCIError *err;
```

5. 埋込みSQL文CONNECTを使用してOracleに接続します。接続にはOCIを使用しないでください。

```
EXEC SQL CONNECT ...
```

6. SQLEnvGet関数を使用して、必要なランタイム・コンテキストに対応付けられているOCI環境ハンドルを取得します。

シングル・スレッド・アプリケーションの場合は、次のようにします。

```
retcode = SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
```

共有サーバー・アプリケーションの場合は、次のようにします。

```
sql_context ctx1;
...
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT USE :ctx1;
...
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
...
retcode = SQLEnvGet(ctx1, &oeh);
```

7. 取得した環境ハンドルを使用してOCIエラー・ハンドルを割り当てます。

```
retcode = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
                    (ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
```

8. 使用するOCIコールに必要な場合は、オプションとして、SQLSvcCtxGetコールを使用してOCIServiceContextハンドルを取得します。

シングル・スレッド・アプリケーションの場合は、次のようにします。

```
retcode = SQLSvcCtxGet(SQL_SINGLE_RCTX, (text *)dbname, (ub4)dbnlen, &svc);
```

共有サーバー環境アプリケーションの場合は、次のようにします。

```
sql_context ctx1;
```

```
...
EXEC SQL ALLOCATE :ctx1;
EXEC SQL CONTEXT USE :ctx1;
...
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd AT :dbname
     USING :hst;
...
retcode = SQLSvcCtxGet(ctx1, (text *)dbname, (ub4)strlen(dbname), &svc);
```

注意:

Pro*C/C++接続名がAT句で指定されていない場合は、NULLポインタが*dbname*として渡されることがあります。

5.10 OCIリリース7コールの埋込み

注意:

ログイン・データ領域(LDA)は、サポートされなくなりました。OCIリリース7コールをPro*C/C++プログラムに埋め込む機能は、サポートされていません。

OCIリリース7コールをPro*C/C++プログラムに埋め込む手順は、次のとおりです。

- OCIログイン・データ領域(LDA)をPro*C/C++プログラム内(MODE=ANSIを指定してプリコンパイルする場合は、宣言部の外)で宣言します。LDAは、OCIヘッダー・ファイルoci.h内で定義された構造体です。詳細は、リリース7対応の『Oracle Call Interfaceプログラマーズ・ガイド』を参照してください。
- OCIのorlon()またはonblon()コールではなく、埋込みSQL文CONNECTを使用してOracleに接続します。
- SQLLIBランタイム・ライブラリ関数sqllda()をコールしてLDA.SQLLIB関数を設定します。

これにより、Pro*C/C++プリコンパイラとOCIでは、両者が連動していることが認識されます。ただし、Oracleカーソルは共有されません。

Oracleランタイム・ライブラリにより接続が管理され、HDAがメンテナンスされるため、OCIホスト・データ領域(HDA)の宣言を意識する必要はありません。

5.10.1 LDAの設定

OCIコールを発行してLDAを設定します。

```
sqllda(&lda);
```

*lda*では、LDAデータ構造体を指定します。

設定が失敗すると、*lda*の*lda_rc*フィールドがエラーを示す1012に設定されます。

5.10.2 リモートの複数接続

sqllda()のコールにより、最後に実行されたSQL文で使用する接続用のLDAが設定されます。追加接続に必要な別のLDAを設定するには、各CONNECTの直後に別のLDAを指定してsqllda()をコールする必要があります。次の例では、2つの非デフォルト・データベースに同時に接続します。

```
#include <ocidfn.h>
Lda_Def lda1;
Lda_Def lda2;

char username[10], password[10], db_string1[20], dbstring2[20];
...
strcpy(username, "scott");
strcpy(password, "tiger");
strcpy(db_string1, "NYNON");
strcpy(db_string2, "CHINON");
/* give each database connection a unique name */
EXEC SQL DECLARE DB_NAME1 DATABASE;
EXEC SQL DECLARE DB_NAME2 DATABASE;
/* connect to first nondefault database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
    AT DB_NAME1 USING :db_string1;
/* set up first LDA */
sqllda(&lda1);
/* connect to second nondefault database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
    AT DB_NAME2 USING :db_string2;
/* set up second LDA */
sqllda(&lda2);
```

DB_NAME1およびDB_NAME2は、C言語の変数ではなくSQL識別子です。識別子DB_NAME1およびDB_NAME2は、2つの非デフォルト・ノードのデフォルトのデータベースの名前を指定するためにのみ使用します。これにより、後のSQL文ではデータベースを名前で参照できます。

5.11 SQLLIBパブリック関数の新しい名前

表5-4は、SQLLIB関数名を示しています。これらのSQLLIB関数は、スレッド・アプリケーションでも非スレッド・アプリケーションでも使用できます。たとえば、従来は、sqlglm()は、この関数の非スレッド・バージョンまたはデフォルト・コンテキスト・バージョンで、sqlglmt()は、スレッド・バージョンまたは非デフォルト・コンテキスト・バージョンで、最初の引数としてコンテキストを指定していました。sqlglm()およびsqlglmt()の名前は、引き続き使用可能です。新しい関数SQLErrorGetText()には、sqlglmt()と同じ引数が必要です。非スレッドまたはデフォルト・コンテキスト・アプリケーションでは、定義した定数SQL_SINGLE_RCTXをコンテキストとして渡してください。

標準SQLLIBパブリック関数はいずれもスレッド・セーフであり、ランタイム・コンテキストを最初の引数として受け入れます。たとえば、SQLErrorGetText()の構文は次のとおりです。

```
void SQLErrorGetText(dvoid *context,  char   *message_buffer,
            size_t *buffer_size,
            size_t *message_length);
```

つまり、古い関数名は既存のアプリケーションでそのまま使用できます。新しく作成するアプリケーションでは、新しい関数名を使用できます。

表5-4は、すべてのSQLLIBパブリック関数とそれに対応する構文を示しています。非スレッドまたはデフォルト・コンテキストの使用方法に関する相互参照も用意されているため、より詳細な説明が必要な場合に参照してください。

表5-4 SQLLIBパブリック関数 - 新しい名前

旧名称	新しい関数プロトタイプ	相互参照
sqlaldt()	struct SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int maximum_variables, unsigned int maximum_name_length, unsigned int maximum_ind_name_length);	SQLDA の割当ても参照してください。
sqlcdat()	void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cursor, sword *return_value);	OCI でのカーソル変数の使用(リリース 7 のみ)も参照してください。
sqlclut()	void SQLSQLDAFree(dvoid *context, struct SQLDA *descriptor_name);	記憶域の割当て解除も参照してください。
sqlcurt()	void SQLCDAToResultSetCursor(dvoid *context, void *cursor, Cda_Def *cda, sword *return_value)	OCI でのカーソル変数の使用(リリース 7 のみ)も参照してください。
sqlglmt()	void SQLErrorGetText(dvoid *context, unsigned char *message_buffer, size_t *buffer_size, size_t *message_length);	エラー・メッセージの全文の取得についても参照してください。
sqlglst()	void SQLStmtGetText(dvoid *context, char *statement_buffer, size_t *statement_length, size_t *sqlfc);	SQL 文のテキスト取得についても参照してください。
sqlld2t()	void SQLLDAGetName(dvoid *context, Lda_Def *lda, text *cname, int *cname_length);	OCI コール(リリース 7 のみ)も参照してください。
sqlldat()	void SQLLDAGetCurrent(dvoid *context, Lda_Def *lda);	リモートの複数接続も参照してください。
sqlnult()	void SQLColumnNullCheck(dvoid *context, unsigned short *value_type, unsigned short *type_code, int *null_status);	NULL/NOT NULL データ型の処理も参照してください。
sqlprct()	void SQLNumberPrecV6(dvoid *context, unsigned long *length, int *precision, int *scale);	精度と位取りの抽出も参照してください。
sqlpr2t()	void SQLNumberPrecV7(dvoid *context, unsigned long *length, int *precision, int *scale);	精度と位取りの抽出も参照してください。
sqlvcpt()	void SQLVarcharGetLength(dvoid *context, unsigned long *data_length, unsigned long *total_length);	VARCHAR 配列コンポーネントの長さを調べる方法も参照してください。
なし	sword SQLEnvGet(dvoid *context, OCIEnv **oeh);	SQLEnvGet()を参照してください。

旧名称	新しい関数プロトタイプ	相互参照
なし	sword SQLSvcCtxGet(dvoid *context, text *dbname, int dbnamelen, OCISvcCtx **svc);	SQLSvcCtxGet()を参照してください。
なし	void SQLRowidGet(dvoid *context, OCIRowid **urid);	SQLRowidGet()を参照してください。
なし	void SQLExtProcError(dvoid *context, char *msg, size_t msglen);	外部プロシージャで使用する場合は、SQLExtProcError()を参照してください。

注意:

これらの関数の引数リストで使用される特定のデータ型については、プラットフォーム固有のバージョンのsqlcpr.hヘッダー・ファイルを参照してください。

関連項目

- OCIリリース8へのインタフェース

5.12 X/Openアプリケーションの開発

X/Openアプリケーションは、分散トランザクション処理(DTP)環境で動作します。抽象モデルでは、X/Openアプリケーションはリソース・マネージャ(RM)に各種サービスの提供を要求します。たとえば、データベース・リソース・マネージャは、データベース内のデータにアクセスします。リソース・マネージャは、アプリケーションのすべてのトランザクションを制御するトランザクション・マネージャ(TM)と対話します。

図5-1 仮定されるDTPモデル

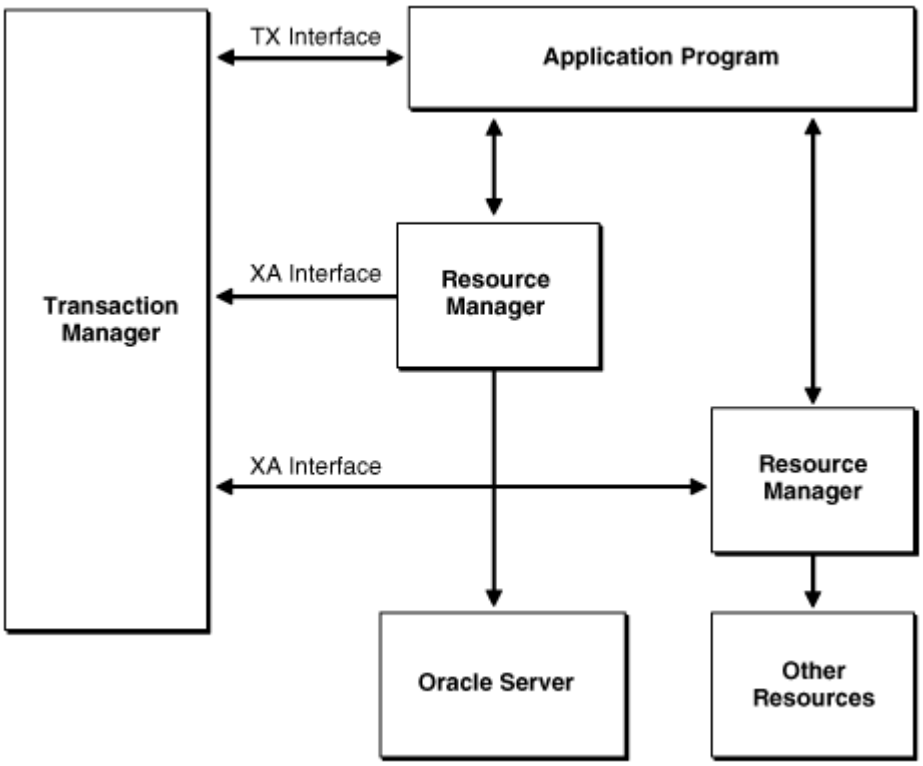


図5-1では、DTPモデルのコンポーネントで、Oracleデータベース内のデータに効率的にアクセスするために対話できる方法を

示しています。このDTPモデルでは、リソース・マネージャとトランザクション・マネージャの間に*XA*インタフェースが指定されています。Oracleでは、XA準拠のライブラリが提供され、このライブラリは、X/Openアプリケーションにリンクさせる必要があります。また、アプリケーション・プログラムとリソース・マネージャ間でネイティブ・インタフェースを指定する必要もあります。

DTPモデルは、トランザクション・マネージャとリソース・マネージャがアプリケーション・プログラムとやりとりする方法を指定します。これは、X/Openガイド『*Distributed Transaction Processing Reference Model*』と関連出版物に記載されています。これらは、次の宛先に書面で請求すれば入手できます。

- The Open Group
- 1010 El Camino Real, Suite 380
- Menlo Park, CA 94025-4345 USA
- http://www.opennc.org/
-

XAインタフェースの使用方法は、ご使用のトランザクション処理(TP)モニターのユーザー・ガイドを参照してください。

5.12.1 Oracle固有の問題

プリコンパイラを使用して、X/Open規格に準拠したアプリケーションを開発できます。ただし、次の要件を満たす必要があります。

5.12.1.1 Oracleへの接続

X/Openアプリケーションでは、データベースへの接続の確立およびメンテナンスは行われません。かわりに、Oracleにより提供されるトランザクション・マネージャとXAインタフェースにより、データベースの接続および切断が透過的に処理されます。したがって、通常、X/Open準拠のアプリケーションはCONNECT文を実行しません。

5.12.1.2 トランザクション制御

X/Openアプリケーションでは、グローバル・トランザクションに影響を与えるCOMMIT、ROLLBACK、SAVEPOINTおよびSET TRANSACTIONなどの文を実行しないでください。たとえば、コミットはトランザクション・マネージャで処理されるため、アプリケーションではCOMMIT文を実行しないでください。また、CREATE、ALTERおよびRENAMEなどのSQLデータ定義文では暗黙的なCOMMITが発行されるため、アプリケーションでこれらの文を実行しないでください。

アプリケーションで後続のSQL操作を妨げるエラーが検出された場合は、内部ROLLBACK文を実行できます。ただし、今後リリースされるXAインタフェースでは変更になる可能性があります。

5.12.1.3 OCIコール(リリース7のみ)

OCIコール・リリース7はサポートされなくなりますか。

> 注意:
>
> ログイン・データ領域(LDA)は、Oracle9*i* ではサポートされなくなりました。Oracle の次のバージョンでは、Pro*C/C++プログラムに OCI リリース 7 コールを埋め込む機能は廃止になります。

X/OpenアプリケーションでOCIコールを発行する場合は、ランタイム・ライブラリ・ルーチンsqllda2()を使用する必要があります。このルーチンにより、XAインタフェースを通じて確立された指定の接続のために、LDAが設定されます。sqllda2()コールの詳細は、リリース7対応の『Oracle Call Interfaceプログラマーズ・ガイド』を参照してください。

OCOM、OCON、OCOF、ONBLON、ORLON、OLON、OLOGOFなどのOCIコールは、X/Openアプリケーションからは発

行できません。

関連項目

- OCIリリース8へのインタフェース

5.12.1.4 リンク

XA機能を利用するには、XAライブラリをX/Openアプリケーション・オブジェクト・モジュールにリンクさせる必要があります。指示については、システム固有のOracleマニュアルを参照してください。

6 埋込みSQL

この章は、埋込みSQLプログラミングの基本技術を理解し、利用する上で役立ちます。この章のトピックは、次のとおりです:

- ホスト変数
- 標識変数
- 基本的なSQL文
- DML RETURNING句
- カーソル
- スクロール可能カーソル
- オプティマイザ・ヒント
- 実行計画の修正
- CURRENT OF句
- カーソル文
- スクロール不可カーソルを使用する完全な例
- スクロール可能カーソルを使用する完全な例

6.1 ホスト変数

Oracleでは、ホスト変数を使用してプログラムにデータやステータス情報を渡します。一方、プログラムでは、ホスト変数を使用してOracleにデータを渡します。

6.1.1 出力変数と入力変数

ホスト変数は、使用方法によって出力ホスト変数または入力ホスト変数と呼ばれます。

SELECT文またはFETCH文のINTO句内のホスト変数は、Oracleによって出力される列の値が入るため出力ホスト変数と呼ばれます。Oracleは、列の値をINTO句内の対応する出力ホスト変数に割り当てます。

SQL文のその他のホスト変数の値は、プログラムがそれをOracleに入力するため、すべて入力ホスト変数と呼ばれます。たとえば、INSERT文のVALUES句内およびUPDATE文のSET句内では入力ホスト変数を使用します。入力ホスト変数はWHERE句、HAVING句およびFOR句内でも使用されます。入力ホスト変数は、SQL文で値または式を使用できる位置であればどこにでも使用できます。

> 注意:
>
> ORDER BY 句ではホスト変数が使用できますが、定数またはリテラルとして扱われるのでホスト変数の内容は無効になります。たとえば、次のような SQL 文があるとします。
>
> ```
> EXEC SQL SELECT ename, empno INTO :name,:number FROM emp ORDER BY :ord;
> ```
>
> ここでは、入力ホスト変数*:ord* が含まれているように見えます。しかし、この句のホスト変数は、定数として扱わ

れ、*:ord* の値にかかわらず、順序付けは行われません。

入力ホスト変数は、SQLキーワードまたはデータベース・オブジェクトの名前を指定するためには使用できません。つまり、ALTER、CREATE、DROPなどのデータ定義文内で入力ホスト変数は使用できません。次の例のDROP TABLE文は無効です。

```
char table_name[30];

printf("Table name? ");
gets(table_name);

EXEC SQL DROP TABLE :table_name;  -- host variable not allowed
```

データベース・オブジェクト名を実行時に変更する必要があるときは、動的SQLを使用します。

入力ホスト変数を含むSQL文をOracleで実行する前に、それらの入力ホスト変数に値を割り当てる必要があります。次に例を示します。

```
int     emp_number;
char    temp[20];
VARCHAR emp_name[20];

/* get values for input host variables */
printf("Employee number? ");
gets(temp);
emp_number = atoi(temp);
printf("Employee name? ");
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
    VALUES (:emp_number, :emp_name);
```

INSERT文のVALUES句内で入力ホスト変数の前にコロンが付いていることに注意してください。

関連項目

- Oracle動的SQL

6.2 標識変数

任意のホスト変数に任意指定の標識変数を関連付けることができます。標識変数に関連付けたホスト変数をSQL文内で使用するたびに、結果コードが対応する標識変数内に格納されます。つまり、標識変数によってホスト変数を監視できます。

注意:

PL/SQL ブロックでは、単一のホスト変数に対して複数の標識変数を使用できません。このように使用すると、「バインドされていない変数があります。」というエラーが発生します。

標識変数をVALUESまたはSET句に使用して、入力ホスト変数にNULLを割り当てます。また、INTO句に使用すると、出力ホスト変数内のNULL値または切り捨てられた値を検出できます。

入力時

プログラムが標識変数に割り当てる値の意味は、次のとおりです。

変数	説明
-1	Oracle によって、その列に NULL が割り当てられます。このホスト変数の値は無視されます。
>=0	Oracle は、このホスト変数の値を列に割り当てます。

出力時

Oracleが標識変数に割り当てる値の意味は、次のとおりです。

変数	説明
-1	この列の値は NULL です。したがって、このホスト変数の値は予測不能です。
0	列の値がそのままこのホスト変数に割り当てられました。
>0	切り捨てられた列の値がこのホスト変数に割り当てられました。標識変数によって返される整数は、列値の元の長さです。SQLCA の SQLCODE が 0(ゼロ)に設定されます。
-2	Oracle によって切り捨てられた列値がこのホスト変数に割り当てられました。ただし、元の列値は決定できませんでした(LONG 列など)。

標識変数は2バイトの整数として定義する必要があります。また、SQL文中では、標識変数の前にコロンを付けてホスト変数の直後に置く必要があります。

6.2.1 NULL値の挿入

標識変数を使用して、NULLをINSERTできます。INSERTの前に、次に示すように、NULLにする列に対応する標識変数をそれぞれ-1に設定します。

```
set ind_comm = -1;

EXEC SQL INSERT INTO emp (empno, comm)
     VALUES (:emp_number, :commission:ind_comm);
```

標識変数*ind_comm*により、COMM列にNULLを入れるように指定されます。

次のように、NULLをハードコードにすることもできます。

```
EXEC SQL INSERT INTO emp (empno, comm)
     VALUES (:emp_number, NULL);
```

これは柔軟性が少なくなりますが、読みやすくなります。一般的には、次の例に示すように条件的にNULLを挿入します。

```
printf("Enter employee number or 0 if not available: ");
scanf("%d", &emp_number);

if (emp_number == 0)
```

```
    ind_empnum = -1;
else
    ind_empnum = 0;

EXEC SQL INSERT INTO emp (empno, sal)
     VALUES (:emp_number:ind_empnum, :salary);
```

6.2.2 戻されたNULL値

標識変数を使用すると、次の例に示すように、戻されたNULLを操作することもできます。

```
EXEC SQL SELECT ename, sal, comm
    INTO :emp_name, :salary, :commission:ind_comm
    FROM emp
    WHERE empno = :emp_number;
 if (ind_comm == -1)
    pay = salary;   /* commission is NULL; ignore it */
else
    pay = salary + commission;
```

6.2.3 NULL値のフェッチ

DBMS=V7またはDBMS=V8のとき、SELECTまたはFETCHしたNULL値を標識変数と関連付けられていないホスト変数に入れると、Oracleは次のエラー・メッセージを発行します。

```
ORA-01405: fetched column value is NULL
```

関連項目

- DBMS

6.2.4 NULLのテスト

次の例のようにWHERE句で標識変数を使用して、NULLをテストできます。

```
EXEC SQL SELECT ename, sal
INTO :emp_name, :salary
FROM emp
WHERE :commission INDICATOR :ind_comm IS NULL ...
```

しかし、関係演算子を使用してNULLとNULL、またはNULLと他の値を比較することはできません。たとえば、COMM列に1つ以上のNULLが含まれる場合、次のSELECT文ではエラーが出力されます。

```
EXEC SQL SELECT ename, sal
INTO :emp_name, :salary
FROM emp
WHERE comm = :commission;
```

次の例は、値のうちのいくつかがNULLである可能性がある場合の、値の等価性を比較する方法を示します。

```
EXEC SQL SELECT ename, sal
     INTO :emp_name, :salary
     FROM emp
     WHERE (comm = :commission) OR ((comm IS NULL) AND
          (:commission INDICATOR :ind_comm IS NULL));
```

6.2.5 切り捨てられた値

DBMS=V7またはV8のとき、切り捨てられた列の値をSELECTまたはFETCHして標識変数に関連付けられていないホスト変数に格納すると、エラーではなく警告が生成されます。

6.3 基本的なSQL文

実行SQL文を使用すると、Oracleデータの問合せ、操作および制御ができます。さらに、表、ビューおよび索引などのOracleオブジェクトの作成、定義およびメンテナンスができます。この章では、データの問合せおよび操作を行う文を重点的に説明しています。

INSERT、UPDATEまたはDELETEなどのデータ操作文を実行するときは、入力ホスト変数の値の設定以外に、文が成功するか失敗するかのみを考えます。これは、SQLCAを調べればわかります。(SQL文を実行すると、SQLCA変数が設定されます。)次の2つの方法でチェックできます。

- WHENEVER文による暗黙的なチェック
- SQLCA変数の明示的なチェック

ただし、SELECT文(問合せ)を実行している場合は、戻されたデータ行の処理もする必要があります。問合せは次のように分類されます。

- 行を戻さない問合せ(有無のみを調べる)
- 1行のみを戻す問合せ
- 複数の行を戻す問合せ

複数の行を戻す問合せの場合は、カーソルを明示的に宣言するか、ホスト配列(配列として宣言されたホスト変数)を使用する必要があります。

注意:

ホスト配列によって、行の一括処理が可能です。

この章ではスカラー・ホスト変数の使用を想定しています。

Oracleデータの問合せおよび操作は、次の埋込みSQL文で行います。

埋込みSQL文	説明
SELECT	1 つ以上の表から行を戻します。
INSERT	表に新しい行を追加します。
UPDATE	表内の行を変更します。
DELETE	表から不要な行を削除します。

明示カーソルの定義および操作は、次の埋込みSQL文で行います。

埋込みSQL文	説明
DECLARE	カーソルに名前を付け、問合せに関連付けます。
OPEN	問合せを実行してアクティブ・セットを決定します。
FETCH	カーソルを移動して、アクティブ・セット内の各行を 1 つずつ取り出します。
CLOSE	カーソルを使用禁止にします(アクティブ・セットは未定義)。

以降の項では、最初にINSERT、UPDATE、DELETEおよび単一行のSELECT文を記述する方法を説明します。次に、複数行のSELECT文の使用方法を説明します。

関連項目

- ランタイム・エラーの処理
- ホスト配列
- 埋込みSQL文およびディレクティブ
- Oracle Database SQL言語リファレンス

6.3.1 SELECT文

データベースへの問合せは日常的なSQL処理です。問合せを発行するには、SELECT文を使用します。次の例では、EMP表を問い合せています。

```
EXEC SQL SELECT ename, job, sal + 2000
INTO :emp_name, :job_title, :salary
FROM emp
WHERE empno = :emp_number;
```

キーワードSELECTの後の列名と式により、選択リストが作成されます。例の選択リストには、3つの項目が含まれています。WHERE句(および存在する場合はそれに続く句)で指定された条件のもと、OracleではINTO句のホスト変数に列値を戻します。

選択リスト内の項目数は、INTO句内のホスト変数と同数であり、すべての戻り値を格納する場所があります。

最も簡単な例として、問合せで1行のみ戻される場合の形式は前述の例のようになります。ただし、1つの問合せで複数の行を戻す場合は、カーソルを使用してそれらの行をFETCHするか、それらの行をSELECTしてホスト変数の配列に入れる必要があります。カーソルとFETCH文はこの章で後述します。

1行のみを戻すように作成した問合せが実際には複数行を戻す場合、SELECTの結果は予測不能です。これがエラーの原因かどうかは、SELECT_ERRORオプションの指定方法によって異なります。デフォルト値であるYESの場合は、複数行が戻されるとエラーが発生します。

関連項目

- ホスト配列

6.3.1.1 使用可能な句

以下の標準SQL句はすべて、次の文内で使用できます。

SELECT文:

- INTO
- FROM
- WHERE
- CONNECT BY
- START WITH
- GROUP BY
- HAVING
- ORDER BY
- FOR UPDATE OF

INTO句を除いて、埋込みSELECT文は、SQL*Plusを使用して対話形式で実行およびテストすることが可能です。SQL*Plusでは、入力ホスト変数のかわりに置換変数または定数を使用します。

6.3.2 INSERT文

INSERT文を使用すると、表またはビューに行を追加できます。次の例では、EMP表に1行追加します。

```
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno)
VALUES (:emp_number, :emp_name, :salary, :dept_number);
```

列リスト内に指定する各列は、INTO句で指定した表に含まれている必要があります。VALUES句には、挿入する行の値を指定します。これらの値は、定数、ホスト変数、SQL式、SQL関数(USER、SYSDATEなど)またはユーザー定義のPL/SQL関数のうち、どの値であってもかまいません。

VALUES句内の値の数は、列リスト内の名前の数と等しくする必要があります。ただし、表に定義されている順序で、VALUES句に表内の各列に対する値がすべて指定されている場合は、この列リストを省略できます。

関連項目

- INSERT(実行可能埋込みSQL)

6.3.2.1 副問合せの使用について

副問合せはネストされたSELECT文です。副問合せを使用すれば、マルチパート検索を実行できます。副問合せは、次の処理に使用できます。

- SELECT、UPDATEおよびDELETE文のWHERE、HAVINGおよびSTART WITH句内の比較のための値を指定します。
- CREATE TABLEまたはINSERT文によって挿入する行の集合を定義します。
- UPDATE文のSET句に対して値を定義します。

次の例では、INSERT文で副問合せを使用して、1つの表から別の表に行をコピーします。

```
EXEC SQL INSERT INTO emp2 (empno, ename, sal, deptno)
SELECT empno, ename, sal, deptno FROM emp
WHERE job= :job_title ;
```

このINSERT文では、中間結果を得るために副問合せを使用しています。

6.3.3 UPDATE文

UPDATE文を使用すると、表またはビュー内の指定した列の値を変更できます。次の例では、EMP表のSAL列とCOMM列を更新します。

```
EXEC SQL UPDATE emp
SET sal = :salary, comm = :commission
WHERE empno = :emp_number;
```

オプションのWHERE句を使用して、行を更新する条件を指定できます。

SET句には、値を指定する必要のある1つ以上の列の名前の並びを指定します。副問合せを使用すれば、次の例のように値を指定できます。

```
EXEC SQL UPDATE emp
SET sal = (SELECT AVG(sal)*1.1 FROM emp WHERE deptno = 20)
WHERE empno = :emp_number;
```

INSERTおよびDELETE文と同様に、UPDATE文ではオプションでRETURNING句を指定できます。オプションのWHERE条件の後にのみ指定できます。

関連項目

- WHERE句
- UPDATE (実行可能埋込みSQL)

6.3.4 DELETE文

DELETE文を使用すると、表またはビューから行を削除できます。次の例では、EMP表から指定した部内の全従業員を削除します。

```
EXEC SQL DELETE FROM emp
WHERE deptno = :dept_number ;
```

オプションのWHERE句を使用して、行を削除する条件を指定しています。

RETURNING句オプションはDELETE文でも使用できます。オプションのWHERE条件の後に指定できます。前述の例では、削除する各従業員のフィールド値を事前に記録することをお薦めします。

関連項目

- DELETE(実行可能埋込みSQL)

6.3.5 WHERE句

WHERE句を使用すると、表またはビュー内で検索条件を満たす行のみをSELECT、UPDATEまたはDELETEできます。WHERE句の検索条件はブール式であり、スカラー・ホスト変数、ホスト配列(SELECT文内を除く)、副問合せおよびユーザー定義のストアド・ファンクションを含めることができます。

WHERE句を省略した場合は、表またはビュー内のすべての行が処理されます。UPDATEまたはDELETE文でWHERE句を省略すると、OracleはSQLCAのsqlwarn[4]を「W」に設定して、すべての列が処理されたことを示します。

6.4 DML RETURNING句

INSERT、UPDATEおよびDELETE文では、オプションで*DML RETURNING*句を設定し、標識変数*iv*を付けて列値の式*expr*をホスト変数*hv*に戻すことができます。DML RETURNING句は次のように設定します。

```
{RETURNING | RETURN} {expr [,expr]}
    INTO {:hv [[INDICATOR]:iv] [, :hv [[INDICATOR]:iv]]}
```

式の個数は、ホスト変数の個数と等しくする必要があります。この句を使用すると、INSERTまたはUPDATEの後、およびアプリケーションに対して情報を記録する必要がある場合のDELETEの前に、行を選択する必要がなくなります。RETURNING句を使用すると、非効率的なネットワークのラウンドトリップや余分な処理を削減し、サーバーのメモリーを節約できます。

Oracle動的SQL方法4ではDML RETURNING句はサポートされませんが、ANSI動的SQL方法4ではサポートされます。複数行に影響するDML RETURNING句を含むDML文は、ANSI動的SQLではサポートされていません。

関連項目

- ANSI動的SQL

6.5 カーソル

問合せで複数行を戻す場合、明示的にカーソルを定義して次の処理を行うことができます。

- 問合せによって戻された最初の行以後の処理
- 現在どの行が処理されているかの追跡および記録

ホスト配列を使用することもできます。

カーソルは、問合せによって戻された行の集合内におけるカレント行を示します。これによって、プログラムは一度に1行ずつ処理できます。次の文を使用してカーソルを定義および操作します。

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

まず、DECLARE CURSOR文を使用して、カーソルに名前を付け、問合せに関連付けます。

OPEN文によって問合せが実行され、この問合せの検索条件を満たす行がすべて判別されます。これらの行は、カーソルのアクティブ・セットと呼ばれる集合を形成します。カーソルをOPENした後、対応する問合せによって戻された行を取り出すことができます。

アクティブ・セットの行は1行ずつ取り出されます(ホスト配列を使用していない場合)。FETCH文を使用してアクティブ・セット内のカレント行を取り出します。FETCHは、すべての行が取り出されるまで繰返し実行できます。

アクティブ・セットからの行のFETCHが終了した後、CLOSE文によってカーソルを使用禁止にします(アクティブ・セットは未定義になります)。

以降の項では、アプリケーション・プログラム内でのこれらのカーソル制御文の使用方法について説明します。

関連項目

- ホスト配列

6.5.1 DECLARE CURSOR文

次の例のように、DECLARE CURSOR文を使用して、カーソルに名前を付け、問合せに関連付けることで、カーソルを定義できます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
```

```
    SELECT ename, empno, sal
    FROM emp
    WHERE deptno = :dept_number;
```

カーソル名は、ホスト変数やプログラム変数ではなく、プリコンパイラで使用される識別子であるため、宣言部では定義しないでください。したがって、あるプリコンパイル単位から別のプリコンパイル・ユニットにカーソル名を渡すことはできません。カーソル名にハイフンは使用できません。長さは任意ですが、重要な意味があるのは先頭の31文字までです。ANSI互換性を維持するため、カーソル名は18文字までにしてください。

コマンドラインまたは構成ファイルでプリコンパイラ・オプションCLOSE_ON_COMMITが使用できます。CLOSE_ON_COMMIT=YESに設定すると、WITH HOLD句なしで宣言されたカーソルはすべてCOMMITまたはROLLBACKの後でクローズされます。

CLOSE_ON_COMMITより高いレベルでMODEが指定されていると、MODEが優先されます。デフォルトはMODE=ORACLEおよびCLOSE_ON_COMMIT=NOです。MODE=ANSIと指定した場合は、WITH HOLD句を使用していないカーソルはCOMMIT時にクローズされます。カーソルのクローズおよび再オープンの回数が多くなるため、アプリケーションの動作は遅くなります。MODE=ANSIのときは、CLOSE_ON_COMMIT=NOと設定するとパフォーマンスが向上します。MODEなどのマクロ・オプションがCLOSE_ON_COMMITなどのマイクロ・オプションに与える影響は、オプション値の優先順位を参照してください。

カーソルに関連付けられたSELECT文にINTO句を含めることはできません。INTO句および出力ホスト変数のリストはFETCH文の一部として指定します。

DECLARE CURSOR文は宣言部なので、そのカーソルを参照する他のすべてのSQL文よりも物理的に(論理的にというだけでなく)前にあることが必要です。つまり、カーソルの前方参照は許可されていません。次の例では、OPEN文の位置が誤っています。

```
...
EXEC SQL OPEN emp_cursor;
*    -- MISPLACED OPEN STATEMENT
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, empno, sal
    FROM emp
    WHERE ename = :emp_name;
```

カーソル制御文(DECLARE、OPEN、FETCH、CLOSE)はすべて同一のプリコンパイル・ユニット内で指定する必要があります。たとえば、ファイルAの中でカーソルをDECLAREしてファイルBでOPENすることはできません。

ホスト・プログラムでは、カーソルを必要な数だけDECLAREできます。ただし、指定されたファイル内ではそれぞれのDECLARE文は一意であることが必要です。つまり、カーソルの適用範囲は1つのファイル全体に及ぶため、1つのプリコンパイル・ユニット内には、たとえ別のブロックやプロシージャ内であっても、同じ名前のカーソルを2つDECLAREすることはできません。

MODE=ANSIまたはCLOSE_ON_COMMIT=YESを使用する場合は、DECLAREセクションにWITH HOLD句を使用して、2つのオプションで定義される動作をオーバーライドできます。これらのオプションを設定すると、COMMITが発行されたときにすべてのカーソルがクローズされます。この場合、処理を続行するのにカーソルを再びオープンする必要があるため、オーバーヘッドが生じてパフォーマンスが低下します。WITH HOLD句を適切に使用して処理を高速化するには、プリコンパイラがANSI規格に適合しているプログラムであることが必要です。

多数のカーソルを使用する場合は、MAXOPENCURSORSオプションの指定が必要になることがあります。

関連項目

- DECLARE CURSOR文でのWITH HOLD句の使用
- CLOSE_ON_COMMIT

- プリコンパイラのオプション
- パフォーマンス・チューニング

6.5.2 OPEN文

OPEN文を使用すると、問合せを実行してアクティブ・セットを決定できます。次の例では、*emp_cursor*という名前のカーソル変数をOPENします。

```
EXEC SQL OPEN emp_cursor;
```

OPENでは、SQLCA内のSQLERRDの第3要素に保存されている処理済行数が0(ゼロ)に設定されます。ただし、この時点ではアプリケーションから参照できる行はありません。これはFETCH文で処理されます。

OPENによって、カーソルはアクティブ・セットの最初の行の直前に位置付けられます。また、SQLCA内のSQLERRDの第3要素に保存されている処理済行数が0(ゼロ)に設定されます。ただし、この時点では実際に取り出される行はありません。行の取出しはFETCH文によって行われます。

カーソルをOPENすると、問合せの入力ホスト変数はカーソルを再度OPENするまで再検査されません。つまり、アクティブ・セットは変更されません。アクティブ・セットを変更するには、カーソルを再OPENします。

通常、カーソルは再OPENする前にCLOSEする必要があります。ただし、MODE=ORACLE(デフォルト)を指定すると、カーソルを再度OPENする前にCLOSEする必要はありません。これによりパフォーマンスが向上する場合があります。

OPENによって行われる作業量は、HOLD_CURSOR、RELEASE_CURSORおよびMAXOPENCURSORSの3つのプリコンパイラ・オプションの値によって決まります。

関連項目

- プリコンパイラ・オプションの使用について

6.5.3 FETCH文

FETCH文を使用すると、アクティブ・セットから行を取り出し、結果を格納する出力ホスト変数を指定できます。カーソルに関連付けられたSELECT文にはINTO句を組み込めないことを思い出してください。INTO句および出力ホスト変数のリストはFETCH文の一部として指定します。次の例では、フェッチした行を3つのホスト変数に対してFETCH INTOを実行します。

```
EXEC SQL FETCH emp_cursor
INTO :emp_name, :emp_number, :salary;
```

カーソルは、あらかじめDECLAREおよびOPENしておく必要があります。最初にFETCH文を実行すると、アクティブ・セットの最初の行より前にあるカーソルが最初の行に移動します。この行がカレント行になります。その後FETCHを実行するたびに、カレント行を変更しながら、カーソルをアクティブ・セットの次の行に進めます。カーソルはアクティブ・セット内を順方向にしか進みません。すでにFETCHを完了した行に戻るには、このカーソルを再OPENして、その後このアクティブ・セットの最初の行からもう一度始めます。

アクティブ・セットを変更する場合は、カーソルに対応する問合せの入力ホスト変数に新しい値を割り当て、カーソルを再OPENしてください。MODE=ANSIに設定されている場合は、再OPENする前にカーソルをCLOSEする必要があります。

次の例のように、異なるホスト変数のセットを使用して、同じカーソルからFETCHできます。しかし、各FETCH文のINTO句内の対応するホスト変数は、同じデータ型であることが必要です。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, sal FROM emp WHERE deptno = 20;
...
```

```
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name1, :salary1;
    EXEC SQL FETCH emp_cursor INTO :emp_name2, :salary2;
    EXEC SQL FETCH emp_cursor INTO :emp_name3, :salary3;
    ...
}
```

アクティブ・セットが空か、それ以上の行を含んでいない場合、FETCHによって「データが見つかりません。」というエラー・コードがSQLCAの*sqlcode*、またはSQLCODEまたはSQLSTATE状態変数に戻されます。出力ホスト変数のステータスは不確定です。(通常のプログラムでは、WHENEVER NOT FOUND文でこのエラーが検出されます。)カーソルを再利用するには、カーソルを再OPENします。

次の場合、カーソル上でのFETCHはエラーになります。

- カーソルをOPENする前
- 「データが見つかりません。」条件の後
- カーソルをCLOSEした後

6.5.4 CLOSE文

アクティブ・セットからの行のFETCHが終了したら、カーソルをCLOSEし、カーソルのOPENによって確保していたリソース(記憶域など)を解放します。カーソルがクローズされると、解析ロックが解放されます。どのリソースが解放されるかは、HOLD_CURSORオプションおよびRELEASE_CURSORオプションの指定によって異なります。次の例では、*emp_cursor*という名前のカーソルをCLOSEします。

```
EXEC SQL CLOSE emp_cursor;
```

クローズしたカーソルのアクティブ・セットは未定義になるため、クローズしたカーソルからはFETCHできません。必要であれば、(たとえば、入力ホスト変数に新しい値を指定して)カーソルを再OPENできます。

MODE=ORACLEの場合、COMMITまたはROLLBACKを発行すると、CURRENT OF句で参照されているカーソルがクローズされます。他のカーソルには、COMMITまたはROLLBACKによる影響はなく、オープンの場合は、オープンのままです。ただし、MODE=ANSIの場合は、COMMITまたはROLLBACKを発行すると、すべての明示カーソルがクローズされます。

関連項目

- データベースの概要

6.6 スクロール可能カーソル

スクロール可能カーソルは、SQL文が実行され、実行中に処理された情報が格納される作業領域です。

カーソルが実行されると、結果セットと呼ばれる一連の行に問合せ結果が入れられます。結果セットは、順番にフェッチすることも、順不同でフェッチすることもできます。順不同の結果セットをスクロール可能カーソルと呼びます。

スクロール可能カーソルを使用すると、ユーザーは前から、後ろからまたはランダムな方法でデータベース結果セットの行にアクセスできます。これにより、プログラムは結果セットの任意の行をフェッチできます。詳細は、リリース9.2.0の『*Oracle Call Interface*プログラマーズ・ガイド』を参照してください。

6.6.1 スクロール可能なカーソルの使用について

次の文を使用して、スクロール可能カーソルを定義および操作します。

6.6.1.1 DECLARE SCROLL CURSOR

DECLARE <cursor name> SCROLL CURSOR文を使用して、スクロール可能カーソルに名前を付け、問合せに対応付けます。

6.6.1.2 スクロール可能カーソル用のOPEN

OPEN文は、スクロール不可カーソルの場合と同じように使用できます。

6.6.1.3 スクロール可能カーソル用のFETCH

FETCH文を使用すると、必要な行をランダムな方法でフェッチできます。アプリケーションでは、行を上方向へまたは下方向へフェッチしたり、最初または最後の行を直接フェッチしたり、または任意の1行をランダムにフェッチできます。

FETCH文には次のオプションがあります。

1. FETCH FIRST

   結果セットの最初の行をフェッチします。

2. FETCH PRIOR

   カレント行の直前の行をフェッチします。

3. FETCH NEXT

   現在位置の次の行をフェッチします。これは、スクロール不可カーソルのFETCHに相当します。

4. FETCH LAST

   結果セットの最後の行をフェッチします。

5. FETCH CURRENT

   カレント行をフェッチします。

6. FETCH RELATIVE *n*

   カレント行を基準にして*n*番目の行をフェッチします。*n*はオフセットです。

7. FETCH ABSOLUTE *n*

   *n*番目の行をフェッチします。*n*は、結果セットの始まりからのオフセットです。

次の例は、結果セットの最後の行をFETCHする方法を示します。

```
EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR
SELECT ename, sal FROM emp WHERE deptno=20;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH LAST emp_cursor INTO :emp_name, :sal;
EXEC SQL CLOSE emp_cursor;
```

6.6.1.4 スクロール可能カーソル用のCLOSE

CLOSE文は、スクロール不可カーソルの場合と同じように使用できます。

注意:

スクロール可能カーソルを REF カーソルとして使用することはできません。

6.2.2 CLOSE_ON_COMMITプリコンパイラ・オプション

CLOSE_ON_COMMITマイクロ・プリコンパイラ・オプションを使用すると、マクロ・オプションMODE=ANSIでCOMMITが実行されるときにすべてのカーソルをクローズするかどうかを選択できます。MODE=ANSIのとき、CLOSE_ON_COMMITのデフォルト値はYESです。明示的にCLOSE_ON_COMMIT=NOと設定すると、COMMITが実行されてもカーソルはクローズされず、カーソルを再オープンして解析する必要がなくなるためパフォーマンスが向上します。

6.2.3 PREFETCHプリコンパイラ・オプション

プリコンパイラ・オプションPREFETCHを使用すると、一定の行数をプリフェッチすることによって問合せの効率を上げることができます。これにより、必要なサーバーのラウンドトリップ回数とメモリー使用量全体が減少します。PREFETCHオプションで設定した行数は、標準の優先順位規則に従って、明示カーソルを含むすべての問合せに使用されます。インラインで使用する場合は、次に示すカーソル文の前にPREFETCHオプションを指定する必要があります。

- EXEC SQL OPEN *cursor*
- EXEC SQL OPEN *cursor* USING *host_var_list*
- EXEC SQL OPEN *cursor* USING DESCRIPTOR *desc_name*

OPENを実行すると、問合せの実行時にプリフェッチする行数がPREFETCHの値によって指定されます。0 (プリフェッチなし)から65535までの値を設定できます。デフォルト値は1です。

注意:

PREFETCH オプションのデフォルト値は 1 で、ラウンドトリップごとに 1 つの行が戻されます。PREFETCH オプションを使用しないように選択する場合は、コマンドラインで PREFETCH オプションを 0 に設定して、明示的に無効にする必要があります。

LONG または LOB 列が取り出される場合は、PREFETCH が自動的に無効になります。

注意:

PREFETCH を使用すると、主に単一行フェッチのパフォーマンスが向上します。配列フェッチの使用時には、PREFETCH は無効です。

注意:

PREFETCH オプションは、状況に応じて広範囲に使用する必要があります。特定の FETCH 文のパフォーマンス

が最適化されるように、適切な PREFETCH 値を選択してください。そのためには、コマンドラインの PREFECTH オプションのかわりにインラインの PREFECTH オプションを使用します。

注意:

FETCH 文にホスト変数とともに標識変数を使用するだけで、多数の大型アプリケーションのパフォーマンスを改善できます。

プリコンパイラ・アプリケーションで、単一行フェッチに対する PREFETCH オプションの使用による最大の利点が得られるように、標識変数を使用することをお薦めします。

6.7 オプティマイザ・ヒント

Pro*C/C++プリコンパイラは、SQL文中のオプティマイザ・ヒントをサポートしています。オプティマイザ・ヒントとは、Oracle SQL オプティマイザへの提案機能であり、通常行われる最適化アプローチを上書きできます。ヒントを使用して、次の事項を指定できます。

- SQL文の最適化アプローチ
- 参照されているそれぞれの表へのアクセス・パス
- 結合のための結合順序
- 表を結合する方法

ヒントによって、ルールベースの最適化およびコストベースの最適化のどちらかを選択できます。コストベースの最適化を使用する場合は、この他にスループットまたは応答速度を最大にするためのヒントを使用できます。

6.7.1 ヒントの発行

オプティマイザ・ヒントは、SELECT、DELETE、UPDATEコマンドの直後に、C形式またはC++形式のコメントの中で発行できます。コメント開始記号の後に間にスペースを空けないでプラス記号(+)を入力し、コメントに1つまたは複数のヒントが含まれていることを示します。たとえば、次の文では最善のスループットを得るために文のコストベース・アプローチの最適化を行うALL_ROWSヒントを使用しています。

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ empno, ename, sal, job
    INTO :emp_rec FROM emp
    WHERE deptno = :dept_number;
```

この文で示されているように、コメントには、オプティマイザ・ヒントのみでなく、他のコメントも組み込めます。

6.8 実行計画の修正

ある環境で開発した複数のモジュールを統合して、別の環境にデプロイするアプリケーション開発環境では、アプリケーションのパフォーマンスが影響を受けます。プリコンパイラ・アプリケーションのパフォーマンスは、データベース環境の変更によって影響を受けることがあります。これらの変更には、オプティマイザ統計の変更、オプティマイザ設定の変更またはメモリー構造のサイズに影響するパラメータの変更が含まれます。

開発環境においてPro*C/C++で使用されるSQLの実行計画を修正するには、プリコンパイル時にOracleのアウトライン機能を使用する必要があります。アウトラインは、SQL文と関連付けられた一連のオプティマイザ・ヒントとして実装されます。SQL文のアウトラインの使用を有効にすると、Oracleでは、格納されたヒントが自動的に考慮され、それらのヒントに従って実行計画を生成しようとします。これにより、モジュールの統合時および異なる環境へのデプロイ時に、パフォーマンスは影響を受けません。

次のSQL文を使用して、Pro*C/C++でアウトラインを作成できます。

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATETABLE...ASSELECT

アウトライン・オプションが設定されている場合、プリコンパイルが正常に終了すると、2つのファイル(SQLファイルおよびLOGファイル)が生成されます。コマンドライン・オプションoutlineおよびoutlnprefixは、アウトラインの生成を制御します。

生成される各アウトライン名は一意です。アプリケーションで使用するファイル名が一意であるため、アウトライン名の生成時にこの情報が使用されます。また、カテゴリ名も接頭辞として使用されます。

> **注意:**
>
> アウトライン名の最大長は 128 バイトです。この制限を超えると、プリコンパイラではエラーが発生します。outlnprefix オプションを使用すると、アウトライン名の長さを制限できます。

例6-1 アウトラインを含むSQLファイルの生成

次のプログラムのアウトラインがサポートされているすべてのSQL文のアウトラインを含むSQLファイルを生成するには、アウトライン・オプションを使用して、次のプログラムをプリコンパイルする必要があります。

```
/*
 * outlndemo.pc
 *
 * Outlines will be created for the following SQL operations,
 * 1. CREATE ... SELECT
 * 2. INSERT ... SELECT
 * 3. UPDATE
 * 4. DELETE
 * 5. SELECT
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>

/* Error handling function. */
void sql_error(char *msg)
{
  exec sql whenever sqlerror continue;

  printf("¥n%s¥n", msg);
```

```
  printf("%.70s¥n", sqlca.sqlerrm.sqlerrmc);
  exec sql rollback release;

  exit(EXIT_FAILURE);
}

int main()
{
  varchar ename[10];
  varchar job[9];
  float sal, comm;

  exec sql begin declare section;
    char *uid = "scott/tiger";
  exec sql end declare section;

  exec sql whenever sqlerror do sql_error("ORACLE error--¥n");
  exec sql connect :uid;

  exec sql insert into bonus
           select ename, job, sal, comm from emp where job like 'SALESMAN';

  exec sql update bonus set sal = sal * 1.1 where sal < 1500;

  exec sql declare c1 cursor for
           select ename, job, sal, comm from bonus order by sal;
  exec sql open c1;
  printf ("Contents of updated BONUS table¥n¥n");
  printf ("ENAME       JOB        SALARY   COMMISSION¥n¥n");
  exec sql whenever not found do break;
  while (1)
  {
    exec sql fetch c1 into :ename, :job, :sal, :comm;
    ename.arr[ename.len]='¥0';
    job.arr[job.len]='¥0';
    printf ("%-9s %-9s   %8.2f    %8.2f¥n", ename.arr,
             job.arr, sal, comm);
  }
  exec sql close c1;
  exec sql whenever not found do sql_error("ORACLE error--¥n");

  exec sql delete from bonus;

  exec sql create table outlndemo_tab as
           select empno, ename, sal from emp where deptno = 10;

  /* Outline will not be created for this DDL statement */
  exec sql drop table outlndemo_tab;

  exec sql rollback work release;
  exit(EXIT_SUCCESS);
}
```

6.8.1 SQLファイル

生成されるファイル名の書式は次のとおりです。

```
<filename>_<filetype>.sql
```

Pro*Cでは、ファイル「abc.pc」の場合、生成されるSQLファイルはabc_pc.sqlになります。

生成されるファイル書式

outlnprefixオプションを使用しない場合は、アウトライン名およびコメントとして、次の一意識別子の書式が使用されます。

```
<category_name>_<filename>_<filetype>_<sequence no.>
```

outlnprefixオプションを使用する(outlnprefix=<prefix_name>)場合は、アウトライン名およびコメントとして、次の一意識別子の書式が使用されます。

```
<prefix_name>_<sequence no.>
```

outline=yes (デフォルトのカテゴリ)の場合、<category_name>はDEFAULTに、アウトライン名は次のいずれかになります。

```
DEFAULT_<filename>_<filetype>_<sequence no.>
```

または

```
<prefix_name>_<sequence no.>
```

<sequence no.>に使用できる値の範囲は、0000から9999です。

生成されたプリコンパイル済ファイルのSQLには、SQLのアウトラインで表示されるように、コメントが追加されます。

6.8.1.1 例

次の例を考えてみます。

例1

abc.pcには、次の文が含まれています。

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

outline=mycat1で、outlnprefixを使用しない場合は、次のようになります。

abc_pc.sqlの内容

select * from emp where empno=:b1 /* mycat1_abc_pc_0000 */;で、カテゴリmycat1のアウトラインmycat1_abc_pc_0000を作成または置換します。

select * from dept /* mycat1_abc_pc_0001 */;で、カテゴリmycat1のアウトラインmycat1_abc_pc_0001を作成または置換します。

abc.cの内容

```
sqlstm.stmt = select * from emp where empno=:b1 /* mycat1_abc_pc_0000 */;

sqlstm.stmt = select * from dept /* mycat1_abc_pc_0001 */;
```

例2

abc.pcには、次の文が含まれています。

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

outline=mycat1で、outlnprefix=myprefixの場合は、次のようになります。

abc_pc.sqlの内容

select * from emp where empno=:b1 /* myprefix_0000 */;で、カテゴリmycat1のアウトラインmyprefix_0000を作成または置換します。

select * from dept /* myprefix_0001 */;で、カテゴリmycat1のアウトラインmyprefix_0001を作成または置換します。

abc.cの内容

```
sqlstm.stmt = select * from emp where empno=:b1 /* myprefix_0000 */;

sqlstm.stmt = select * from dept /* myprefix_0001 */;
```

例3

abc.pcには、次の文が含まれています。

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

outline=yesで、outlnprefix=myprefixの場合は、次のようになります。

abc_pc.sqlの内容

select * from emp where empno=:b1 /* myprefix_0000 */;で、アウトラインmyprefix_0000を作成または置換します。

select * from dept /* myprefix_0001 */;で、アウトラインmyprefix_0001を作成または置換します。

abc.cの内容

```
sqlstm.stmt = "select * from emp where empno=:b1 /* myprefix_0000 */;

sqlstm.stmt = "select * from dept /* myprefix_0001 */";
```

6.8.2 LOGファイル

生成されるファイル名の書式は次のとおりです。

```
<filename>_<filetype>.log
```

Pro*Cでは、ファイル「abc.pc」の場合、生成されるLOGファイルはabc_pc.logになります。

次に例を示します。

例1

abc.pcには、次の文が含まれています。

```
EXEC SQL select * from emp;
```

abc_pc.logの内容

```
CATEGORY <Category_name>
     Source SQL_0
        SELECT * FROM emp
     OUTLINE NAME
        abc_pc_0000
     OUTLINE SQL_0
        Select * from emp /* abc_pc_0000 */
```

6.9 CURRENT OF句

DELETE文またはUPDATE文でCURRENT OF *cursor_name*句を使用すると、指定したカーソルから最後にFETCHした行を参照できます。カーソルをオープンし、行に位置付けておく必要があります。FETCHが一度も行われていない場合や、そのカーソルがオープンされていない場合には、CURRENT OF句を使用するとエラーが発生し、1行も処理されません。

UPDATE文またはDELETE文のCURRENT OF句で参照するカーソルをDECLAREするときに、FOR UPDATE OF句を任意で指定できます。CURRENT OF句は、必要に応じてFOR UPDATE句を追加するようにプリコンパイラに指示します。

次の例では、CURRENT OF句を使用して、*emp_cursor*という名前のカーソルから最後にFETCHされた行を参照します。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT ename, sal FROM emp WHERE job = 'CLERK'
     FOR UPDATE OF sal;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;) {
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
    EXEC SQL UPDATE emp SET sal = :new_salary
         WHERE CURRENT OF emp_cursor;
}
```

関連項目

- FOR UPDATE OFの使用

6.9.1 制限事項(FOR UPDATE OF)

CURRENT OF句は索引構成表では使用できません。

明示的なFOR UPDATE OF句または暗黙的なFOR UPDATE句では、行の排他ロックが取得されます。行はすべて、FETCHされるときではなく、OPEN時にロックされ、COMMITまたはROLLBACKを行うとロックが解除されます。したがって、COMMIT後はFOR UPDATEカーソルからFETCHできません。これを試みると、Oracleから1002エラー・コードが戻されます。

また、ホスト配列はCURRENT OF句と一緒には使用できません。代替方法については、CURRENT OF句の擬似実行についても参照してください。

さらに、関連するFOR UPDATE OF句で複数の表は参照できません。つまり、CURRENT OF句とは結合できないということです。

最後に、動的SQLはCURRENT OF句と一緒には使用できません。

6.10 カーソル文

次の例は、アプリケーション・プログラムでのカーソル制御文の一般的な順序を示しています。

```
   ...
/* define a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT ename, job
        FROM emp
        WHERE empno = :emp_number
        FOR UPDATE OF job;

/* open the cursor and identify the active set */
   EXEC SQL OPEN emp_cursor;
```

```
/* break if the last row was already fetched */
   EXEC SQL WHENEVER NOT FOUND DO break;

/* fetch and process data in a loop */
   for (;;)
   {
      EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;

/* optional host-language statements that operate on
   the FETCHed data */

      EXEC SQL UPDATE emp
           SET job = :new_job_title
           WHERE CURRENT OF emp_cursor;
   }
...
/* disable the cursor */
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT WORK RELEASE;
   ...
```

6.11 スクロール不可カーソルを使用する完全な例

次の完全なプログラム例では、カーソルとFETCH文の使用方法を示します。プログラムでは、部門番号の入力を要求してから、その部門の全従業員の名前を表示します。

最後の1つのFETCHを除くすべてのフェッチで1行ずつ戻され、FETCH中にエラーが検出されなければ、成功のステータス・コードが戻されます。最後のFETCHは失敗し、「データが見つかりません。」というOracleエラー・コードが*sqlca.sqlcode*に戻されます。実際にFETCHされた行の累積数は、SQLCAのsqlerrd[2]に示されます。

```
#include <stdio.h>

/* declare host variables */
char userid[12] = "SCOTT/TIGER";
char emp_name[10];
int  emp_number;
int  dept_number;
char temp[32];
void sql_error();

/* include the SQL Communications Area */
#include <sqlca.h>

main()
{  emp_number = 7499;
/* handle errors */
   EXEC SQL WHENEVER SQLERROR do sql_error("Oracle error");

/* connect to Oracle */
   EXEC SQL CONNECT :userid;
   printf("Connected.¥n");

/* declare a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename
      FROM emp
      WHERE deptno = :dept_number;
```

```
   printf("Department number? ");
   gets(temp);
   dept_number = atoi(temp);

/* open the cursor and identify the active set */
   EXEC SQL OPEN emp_cursor;

   printf("Employee Name¥n");
   printf("-------------¥n");
/* fetch and process data in a loop
   exit when no more data */
   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1)
   {
      EXEC SQL FETCH emp_cursor INTO :emp_name;
      printf("%s¥n", emp_name);
   }
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}

void
sql_error(msg)
char *msg;
{
   char buf[500];
   int buflen, msglen;

   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   buflen = sizeof (buf);
   sqlglm(buf, &buflen, &msglen);
   printf("%s¥n", msg);
   printf("%*.s¥n", msglen, buf);
   exit(1);
}
```

6.12 スクロール可能カーソルを使用する完全な例

次のプログラムでは、スクロール可能カーソルとFETCH文で使用される様々なオプションの使用方法を示します。

```
#include <stdio.h>

/* declare host variables */
char userid[12]="SCOTT/TIGER";
char emp_name[10];
void sql_error();

/*  include the SQL Communications Area */
#include<sqlca.h>

main()
{
/* handle errors */
    EXEC SQL WHENEVER SQLERROR do sql_error("Oracle error");

/* connect to Oracle */
```

```
    EXEC SQL CONNECT :userid;
    printf("Connected.¥n");

/* declare a scrollable cursor */
    EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR
    SELECT ename FROM emp;

/* open the cursor and identify the active set */
     EXEC SQL OPEN emp_cursor;

 /* Fetch the last row */
     EXEC SQL FETCH LAST emp_cursor INTO :emp_name;

 /* Fetch row number 5 */
      EXEC SQL FETCH ABSOLUTE 5 emp_cursor INTO :emp_name;

/*  Fetch row number 10 */
       EXEC SQL FETCH RELATIVE 5 emp_cursor INTO :emp_name;

/*  Fetch row number 7 */
       EXEC SQl FETCH RELATIVE -3 emp_cursor INTO :emp_name;

/*  Fetch the first row */
       EXEC SQL FETCH FIRST emp_cursor INTO :emp_name;

/*  Fetch row number 2*/
       EXEC SQL FETCH my_cursor INTO :emp_name;

/*  Fetch row number 3 */
       EXEC SQL FETCH NEXT my_cursor INTO :emp_name;

/*  Fetch row number 3 */
       EXEC SQL FETCH CURRENT my_cursor INTO :emp_name;

/*  Fetch row number 2 */
        EXEC SQL FETCH PRIOR my_cursor INTO :emp_name;
}

void
sql_error(msg)
char *msg;
{
     char buf[500];
     int buflen , msglen;

     EXEC SQL WHENEVER SQLERROR  CONTINUE;
     EXEC SQL ROLLBACK TRANSACTION;
     buflen = sizeof (buf);
     sqlglm(buf, &buflen, &mesglen);
     printf("%s¥n",msg);
     printf("%*.s¥n",msglen,buf);
     exit(1);
}
```

6.13 位置付け更新

次の抜粋例は、ユニバーサルROWIDを使用した位置付け更新を示しています。

```
#include <oci.h>
```

```
...
OCIRowid *urowid;
...
EXEC SQL ALLOCATE :urowid;
EXEC SQL DECLARE cur CURSOR FOR
    SELECT rowid, ... FROM my_table FOR UPDATE OF ...;
EXEC SQL OPEN cur;
EXEC SQL FETCH cur INTO :urowid, ...;
/* Process data */
...
EXEC SQL UPDATE my_table SET ... WHERE CURRENT OF cur;
EXEC SQL CLOSE cur;
EXEC SQL FREE :urowid;
...
```

関連項目

- ユニバーサルROWID

7 埋込みPL/SQL

PL/SQLトランザクション処理ブロックをプログラム内に埋め込むことにより、パフォーマンスを改善する方法を説明します。この章のトピックは、次のとおりです:

- PL/SQLの利点
- 埋込みPL/SQLブロック
- ホスト変数
- 標識変数
- ホスト配列
- 埋込みPL/SQLでのカーソルの使用
- ストアドPL/SQLおよびJavaサブプログラム
- 外部プロシージャ
- 動的SQLの使用方法について

関連項目

- 埋込みPL/SQL

7.1 PL/SQLの利点

この項では、PL/SQLによって提供される次のような機能および利点を説明します。

- パフォーマンス向上
- Oracleとの統合
- カーソルFORループ
- プロシージャおよびファンクション
- パッケージ
- PL/SQL表
- ユーザー定義のレコード

関連項目

- Oracle Database PL/SQL言語リファレンス

7.1.1 パフォーマンス向上

PL/SQLによって、オーバーヘッドの削減、パフォーマンスの改善および生産性の向上が図れます。たとえば、PL/SQLを使用しないと、Oracleは一度に1つずつSQL文を処理する必要があります。その結果、各SQL文によってサーバーへの別のコールが発生し、オーバーヘッドが増加します。しかし、PL/SQLを使用すると、SQL文のブロック全体をサーバーに送信できます。これにより、アプリケーションとOracleとの間の通信は最小限になります。

7.1.2 Oracleとの統合

PL/SQLは、Oracleサーバーと密接に統合されています。たとえば、PL/SQLデータ型の大部分は、Oracleデータ・ディクショナ

リにとっても固有のデータ型です。さらに、次の例に示すとおり、データ・ディクショナリ内に格納された列定義に基づいて変数を宣言するための%TYPE属性を指定できます。

```
job_title  emp.job%TYPE;
```

したがって、列の厳密なデータ型を知る必要はありません。しかも、列定義を変更すると、変数宣言もそれに応じて自動的に変更されます。これによって、データ独立性を提供し、メンテナンス・コストを削減し、データベース変更時にプログラムが順応できるようになります。

7.1.3 カーソルFORループ

PL/SQLを使用すれば、カーソルを定義して操作するために、DECLARE、OPEN、FETCHおよびCLOSE文を使用する必要はありません。かわりに、カーソルFORループを使用でき、ループ索引をレコードとして暗黙的に宣言し、指定された問合せに関連付けられているカーソルをオープンして、データを繰り返しカーソルからフェッチしてレコードに入れてから、カーソルをクローズします。次に例を示します。

```
DECLARE
...
BEGIN
   FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP
   IF emprec.comm / emprec.sal > 0.25 THEN ...
   ...
END LOOP;
END;
```

ドット表記法を使用すると、レコード内のコンポーネントを参照できます。

7.1.4 プロシージャおよびファンクション

PL/SQLにはプロシージャとファンクションと呼ばれる2種類のサブプログラムがあり、これらを使用すると、各動作を分離できるため、アプリケーションの開発が容易になります。一般に、プロシージャはアクションを実行するために使用し、ファンクションは値を計算するために使用します。

プロシージャおよびファンクションには拡張性があります。つまり、プロシージャとファンクションを使用することにより、PL/SQL言語を必要に応じて調整できます。たとえば、新しい部門を作成するプロシージャが必要な場合、次のように記述します。

```
PROCEDURE create_dept
  (new_dname  IN CHAR(14),
   new_loc    IN CHAR(13),
   new_deptno OUT NUMBER(2)) IS
BEGIN
   SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
   INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);
END create_dept;
```

このプロシージャをコールすると、プロシージャでは新しい部門名と場所を受け取り、部門番号データベース順序の次の値を選択し、その新しい番号、名前および場所を*dept*表に挿入してから、新しい番号をコール元に戻します。

仮パラメータの動作を定義するには、パラメータ・モードを使用します。パラメータ・モードにはIN (デフォルト)、OUTおよびIN OUTの3つがあります。INパラメータを使用すると、コールされるサブプログラムに値を渡せます。OUTパラメータを使用すると、サブプログラムのコール元に値を戻せます。IN OUTパラメータを使用すると、コールされるサブプログラムに初期値を渡し、更新された値をコール元に戻すことができます。

それぞれの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に変換可能であることが必要です。表7-1は、データ型間の正当な変換を示しています。

7.1.5 パッケージ

PL/SQLでは、論理的に関連する型、プログラム・オブジェクトおよびサブプログラムを1つのパッケージにまとめることができます。プロシージャ・データベース拡張機能がある場合、パッケージをコンパイルして、Oracleデータベースに格納でき、そのデータベースの内容は多くのアプリケーションで共有できます。

パッケージには通常、仕様部および本体の2つの部分があります。仕様部とは、アプリケーションへのインタフェースで、使用可能な型、定数、変数、例外、カーソルおよびサブプログラムが宣言されます。本体は、カーソルとサブプログラムを定義し、仕様を実行します。次の例では、2つの雇用プロシージャをパッケージ化しています。

```
PACKAGE emp_actions IS  -- package specification
  PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);

  PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

PACKAGE BODY emp_actions IS  -- package body
  PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
  BEGIN
    INSERT INTO emp VALUES (empno, ename, ...);
  END hire_employee;

  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM emp WHERE empno = emp_id;
  END fire_employee;
END emp_actions;
```

パッケージ仕様部内の宣言のみ参照可能で、アプリケーションからアクセスできます。パッケージ本体中の詳細な実装内容は非表示のためアクセスできません。

7.1.6 PL/SQL表

PL/SQLにはTABLEの名前の複合データ型が用意されています。TABLE型のオブジェクトは、*PL/SQL*表と呼ばれ、データベース表をモデルとしています(まったく同じではありません)。PL/SQL表は1列からなり、主キーを使用して、配列と同じ方法で行にアクセスします。列は、任意のスカラー型(CHAR、DATEまたはNUMBERなど)にできますが、主キーはBINARY_INTEGER型、PLS_INTEGER型またはVARCHAR2型にする必要があります。

ブロック、プロシージャ、ファンクションまたはパッケージのいずれかの宣言部でPL/SQL表型を宣言できます。次の例では、*NumTabTyp*と呼ばれるTABLE型を宣言しています。

```
...
DECLARE
   TYPE NumTabTyp IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
...
BEGIN
   ...
END;
...
```

次の例に示すように、一度*NumTabTyp*型を定義すると、その型のPL/SQL表を宣言できます。

```
num_tab  NumTabTyp;
```

識別子*num_tab*は、PL/SQL表全体を表しています。

配列に似た構文を使用してPL/SQL表の中の行を参照し、主キーの値を指定します。たとえば、*num_tab*の名前のPL/SQL表の中の9番目の行を参照するには次のように指定します。

```
num_tab(9) ...
```

7.1.7 ユーザー定義のレコード

%ROWTYPE属性を使用して、表の中の行を表すレコード、またはカーソルによってフェッチされる行を表すレコードを宣言できます。しかし、レコード内のコンポーネントのデータ型は指定できません。また、ユーザー独自のコンポーネントも定義できません。複合データ型RECORDを使用すると、これらの制限事項を取り除くことができます。

RECORD型のオブジェクトはレコードと呼ばれます。PL/SQL表とは異なり、レコードには一意の名前を持つコンポーネントがあります。コンポーネントのデータ型はそれぞれ異なっていてもかまいません。たとえば、ある従業員について異なる種類のデータ(名前、給与、雇用日など)があるとします。このデータは、型は異なりますが、論理的に関連しています。従業員の名前、給与および雇用日などのコンポーネントを持つレコードによって、1つの論理単位としてデータを処理できます。

ブロック、プロシージャ、ファンクションまたはパッケージのいずれかの宣言部で、レコード型およびレコード・オブジェクトを宣言できます。次の例では、*DeptRecTyp*と呼ばれるRECORD型を宣言しています。

```
DECLARE
TYPE DeptRecTyp IS RECORD
    (deptno  NUMBER(4) NOT NULL, -- default is NULL allowed
    dname   CHAR(9),
    loc     CHAR(14));
```

コンポーネント宣言は変数宣言に似ているため注意してください。各コンポーネントには一意の名前および固有のデータ型を指定します。コンポーネント宣言にNOT NULLオプションを追加すると、そのコンポーネントにはNULLを割り当てられません。

次の例に示すように、一度*DeptRecTyp*を定義すると、その型のレコードを宣言できます。

```
dept_rec  DeptRecTyp;
```

識別子*dept_rec*は、レコード全体を表しています。

ドット表記法を使用すると、レコード内の個別コンポーネントを参照できます。たとえば、次のように*dept_rec*レコード内で*dname*コンポーネントを参照します。

```
dept_rec.dname ...
```

7.2 埋込みPL/SQLブロック

Pro*C/C++プリコンパイラでは、PL/SQLブロックが1つの埋込みSQL文と同様に扱われます。したがって、PL/SQLブロックは、プログラム内のSQL文を記述できる位置であればどこにでも記述できます。

PL/SQLブロックをPro*C/C++プログラム内に埋め込むには、次のように、EXEC SQL EXECUTEおよびEND-EXECキーワードでPL/SQLブロックを囲むのみです。

```
EXEC SQL EXECUTE
DECLARE
...
BEGIN
   ...
END;
END-EXEC;
```

END-EXECキーワードの後には、セミコロンを付ける必要があります。

プログラムを作成した後に、通常の方法でソース・ファイルをプリコンパイルします。

プログラムに埋込みPL/SQLが含まれている場合、PL/SQLはOracleサーバーによって解析される必要があるため、必ずSQLCHECK=SEMANTICSコマンドライン・オプションを指定してください。サーバーに接続する場合には、SQLCHECK=SEMANTICSのみでなくUSERIDオプションも指定してください。

関連項目

- プリコンパイラ・オプションの使用について

7.3 ホスト変数

ホスト変数は、ホスト言語とPL/SQLブロック間の通信を仲介します。ホスト変数はPL/SQLと共有できるので、PL/SQLではホスト変数の設定および参照ができます。

たとえば、ユーザーに情報の提供を求め、この情報をPL/SQLブロックに渡すためのホスト変数を使用するようにユーザーに指示できます。これにより、PL/SQLを使用してデータベースにアクセスし、ホスト変数を介してその結果をホスト・プログラムに戻せるようになります。

PL/SQLブロック内ではホスト変数はブロック全体のグローバル変数として扱われ、PL/SQL変数を使用できる位置であればどこにでも使用できます。SQL文内におけるホスト変数と同様、PL/SQLブロック内のホスト変数も先頭にコロンを付ける必要があります。コロンは、ホスト変数とPL/SQL変数およびデータベース・オブジェクトとを区切ります。

> 注意:
>
> PL/SQL ブロックで VARCHAR、CHARZ または STRING 型を出力ホスト変数として使用するには、ブロックを入力する前に長さを初期化する必要があります。次の例のように、長さは VARCHAR、CHARZ または STRING の宣言された(最大の)長さに設定してください。

関連項目

- ポインタ変数

7.3.1 例: PL/SQLでのホスト変数の使用

次の例では、PL/SQLにおけるホスト変数の使用方法を示します。プログラムでは、ユーザーに従業員番号の入力を求めるプロンプトが表示され、その番号に応じて、従業員の役職名、雇用日、給与が表示されます。

```
char username[100], password[20];
char job_title[20], hire_date[9], temp[32];
int emp_number;
float salary;

#include <sqlca.h>

printf("Username? ¥n");
gets(username);
printf("Password? ¥n");
gets(password);

EXEC SQL WHENEVER SQLERROR GOTO sql_error;
```

```
EXEC SQL CONNECT :username IDENTIFIED BY :password;
printf("Connected to Oracle¥n");
for (;;)
{
   printf("Employee Number (0 to end)? ");
   gets(temp);
   emp_number = atoi(temp);

   if (emp_number == 0)
   {
      EXEC SQL COMMIT WORK RELEASE;
      printf("Exiting program¥n");
      break;
   }
/*-------------- begin PL/SQL block -----------------*/
   EXEC SQL EXECUTE
   BEGIN
      SELECT job, hiredate, sal
         INTO :job_title, :hire_date, :salary
         FROM emp
         WHERE empno = :emp_number;
   END;
   END-EXEC;
/*-------------- end PL/SQL block -----------------*/

   printf("Number  Job Title  Hire Date  Salary¥n");
   printf("------------------------------------¥n");
   printf("%6d  %8.8s  %9.9s  %6.2f¥n",
   emp_number, job_title, hire_date, salary);
}
...
exit(0);

sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error¥n");
exit(1);
```

ホスト変数*emp_number*がPL/SQLブロックが入力される前に設定され、ホスト変数*job_title*、*hire_date*および*salary*がブロック内で設定されていることに注意してください。

7.3.2 複雑な例

次の例では、ユーザーに銀行口座番号、取引の種類、取引金額の入力を求めるプロンプトが表示されてから、口座に記帳されます。口座が存在しない場合、例外が発生します。取引が完了すると、そのステータスが表示されます。

```
#include <stdio.h>
#include <sqlca.h>

char username[20];
char password[20];
char status[80];
char temp[32];
int  acct_num;
double trans_amt;
void sql_error();
```

```
main()
{
   char trans_type;

   strcpy(password, "TIGER");
   strcpy(username, "SCOTT");

   EXEC SQL WHENEVER SQLERROR DO sql_error();
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("Connected to Oracle¥n");

   for (;;)
   {
      printf("Account Number (0 to end)? ");
      gets(temp);
      acct_num = atoi(temp);

      if(acct_num == 0)
      {
         EXEC SQL COMMIT WORK RELEASE;
         printf("Exiting program¥n");
         break;
      }

      printf("Transaction Type - D)ebit or C)redit? ");
      gets(temp);
      trans_type = temp[0];

      printf("Transaction Amount? ");
      gets(temp);
      trans_amt = atof(temp);

/*----------------- begin PL/SQL block -------------------*/
      EXEC SQL EXECUTE
      DECLARE
         old_bal      NUMBER(9,2);
         err_msg      CHAR(70);
         nonexistent  EXCEPTION;

      BEGIN
         :trans_type := UPPER(:trans_type);
         IF :trans_type = 'C' THEN       -- credit the account
            UPDATE accts SET bal = bal + :trans_amt
            WHERE acctid = :acct_num;
            IF SQL%ROWCOUNT = 0 THEN    -- no rows affected
               RAISE nonexistent;
            ELSE
               :status := 'Credit applied';
            END IF;
         ELSIF :trans_type = 'D' THEN    -- debit the account
            SELECT bal INTO old_bal FROM accts
               WHERE acctid = :acct_num;
            IF old_bal >= :trans_amt THEN   -- enough funds
               UPDATE accts SET bal = bal - :trans_amt
                  WHERE acctid = :acct_num;
               :status := 'Debit applied';
            ELSE
               :status := 'Insufficient funds';
            END IF;
```

```
         ELSE
            :status := 'Invalid type: ' || :trans_type;
         END IF;
         COMMIT;
      EXCEPTION
         WHEN NO_DATA_FOUND OR nonexistent THEN
            :status := 'Nonexistent account';
         WHEN OTHERS THEN
            err_msg := SUBSTR(SQLERRM, 1, 70);
            :status := 'Error: ' || err_msg;
      END;
      END-EXEC;
/*----------------- end PL/SQL block ---------------------- */

      printf("¥nStatus: %s¥n", status);
   }
   exit(0);
}


void
sql_error()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    printf("Processing error¥n");
    exit(1);
}
```

7.3.3 VARCHAR疑似型

VARCHARデータ型は、可変長文字列の宣言に使用できます。VARCHARが入力ホスト変数の場合は、どのくらいの長さを前提にすればよいかをOracleに通知する必要があります。したがって、文字列コンポーネントに格納される値の実際の長さに、長さコンポーネントを設定してください。

VARCHARが出力ホスト変数の場合は、Oracleで自動的に長さコンポーネントが設定されます。ただし、PL/SQLブロックでVARCHAR(CHARZおよびSTRING)出力ホスト変数を使用するには、ブロックを入力する前に長さコンポーネントを初期化する必要があります。したがって、次の例に示すとおり、長さコンポーネントはVARCHARの宣言された(最大の)長さに設定してください。

```
int     emp_number;
varchar emp_name[10];
float   salary;
...
emp_name.len = 10;   /* initialize length component */

EXEC SQL EXECUTE
  BEGIN
    SELECT ename, sal INTO :emp_name, :salary
        FROM emp
        WHERE empno = :emp_number;
    ...
  END;
END-EXEC;
...
```

7.3.4 制限事項

PL/SQLブロックでは、Cポインタまたは配列構文を使用しないでください。PL/SQLコンパイラはCホスト変数の式を認識できないため、解析できません。たとえば、次の例は無効です。

```
EXEC SQL EXECUTE
    BEGIN
        :x[5].name := 'SCOTT';
        ...
    END;
END-EXEC;
```

構文エラーを回避するには、プレースホルダ(一時変数)を使用して、構造体の移入を行う構造体フィールドのアドレスを保持します。次の例は有効です。

```
name = x[5].name ;
EXEC SQL EXECUTE
    BEGIN
        :name := ...;
        ...
    END;
END-EXEC;
```

7.4 標識変数

PL/SQLでは、NULLを操作できるため、標識変数は必要ありません。たとえば、PL/SQL内では、次のようにIS NULL演算子を使用してNULLをテストできます。

```
IF variable IS NULL THEN ...
```

その後、次のように代入演算子(:=)を使用して、NULL値を指定できます。

```
variable := NULL;
```

ただし、C言語のようなホスト言語はNULL値を操作できないため、標識変数を必要とします。埋込みPL/SQLでは、次の目的でインジケータ変数を使用できるため、この要件を満たします。

- ホスト・プログラムからのNULL入力値の受入れ
- NULLまたは切り捨てられた値のホスト・プログラムへの出力

PL/SQLブロックでインジケータ変数を使用するときは、次の規則に従ってください。

- インジケータ変数は単独では参照できません。関連付けられたホスト変数に追加する必要があります。
- インジケータ変数を指定してホスト変数を参照する場合、同じブロックでは常に同じ方法で参照する必要があります。

次の例では、標識変数*ind_comm*がSELECT文でそのホスト変数*commission*とともに表示されているため、IF文でも同じように表示する必要があります。

```
...
EXEC SQL EXECUTE
BEGIN
    SELECT ename, comm
        INTO :emp_name, :commission :ind_comm
        FROM emp
        WHERE empno = :emp_number;
    IF :commission :ind_comm IS NULL THEN ...
    ...
```

```
END;
END-EXEC;
```

PL/SQLでは*:commission :ind_comm*はその他の単純な変数と同じように扱われるため注意してください。PL/SQLブロック内の標識変数は直接参照できませんが、PL/SQLでは、ブロックに入るときに標識変数の値がチェックされ、ブロックから出るときにその値が正しく設定されます。

7.4.1 NULLの処理

ブロックに入るとき、標識変数の値が-1であれば、PL/SQLによってNULLがホスト変数に自動的に割り当てられます。ブロックから出るとき、ホスト変数がNULLであれば、PL/SQLによって値-1が標識変数に自動的に割り当てられます。次の例では、PL/SQLブロックに入力される前に*ind_sal*の値が-1であった場合に、*salary_missing*例外が発生します。例外とは、名前が指定されたエラー状態のことです。

```
...
EXEC SQL EXECUTE
BEGIN
    IF :salary :ind_sal IS NULL THEN
    RAISE salary_missing;
END IF;
...
END;
END-EXEC;
...
```

7.4.2 切り捨てられた値

PL/SQLでは、切り捨てられた文字列の値がホスト変数に割り当てられても、例外とはみなされません。ただし、標識変数を指定している場合には、PL/SQLによってその標識変数が文字列の元の長さに設定されます。次の例では、ホスト・プログラムで*ind_name*の値をチェックして、切り捨てられた値が*emp_name*に割り当てられているかどうかを通知できます。

```
...
EXEC SQL EXECUTE
DECLARE
...
new_name  CHAR(10);
BEGIN
    ...
    :emp_name:ind_name := new_name;
    ...
END;
END-EXEC;
```

7.5 ホスト配列

入力ホスト配列およびインジケータ配列は、PL/SQLブロックに渡せます。これらは、BINARY_INTEGER型またはPLS_INTEGER型のPL/SQL変数を使用して索引付けができます。VARCHAR2型のキーは使用できません。通常は、ホスト配列全体がPL/SQLに渡されますが、ARRAYLEN文(後述)を使用すれば、より小さい配列サイズを指定できます。

さらに、ホスト配列のすべての値をPL/SQL表の複数の行に割り当てるために、プロシージャ・コールを使用できます。配列のサブスクリプトの範囲が*m*から*n*の場合、対応するPL/SQL表の索引範囲は常に1から*n* - *m* + 1になります。たとえば、配列サブスクリプト範囲が5から10の場合、対応するPL/SQL表の索引範囲は、1から(10 - 5 + 1)または1から6です。

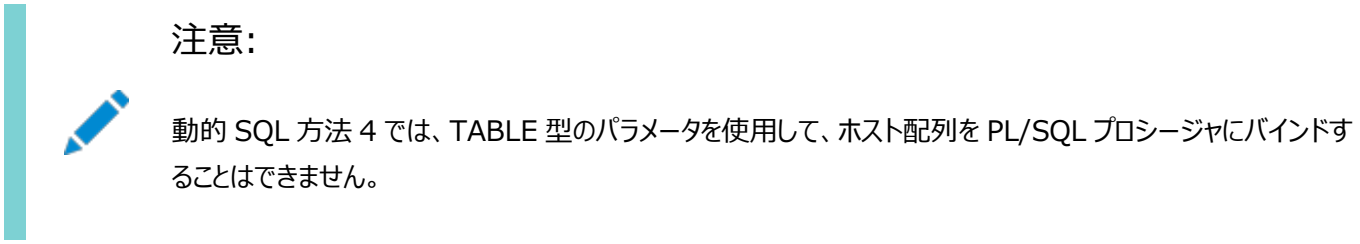
次の例では、*salary*という名前の配列をPL/SQLブロックに渡し、そのブロックのファンクション・コール内でその配列を使用してい

ます。このファンクションは、一連の数値の中央値を検出するため、*median*という名前が付いています。その仮パラメータには、*num_tab*というPL/SQL表が含まれています。このファンクション・コールにより、実パラメータ*salary*内のすべての値を仮パラメータ*num_tab*内の行に割り当てます。

```
...
float salary[100];

/* populate the host array */

EXEC SQL EXECUTE
  DECLARE
    TYPE NumTabTyp IS TABLE OF REAL
        INDEX BY BINARY_INTEGER;
    median_salary  REAL;
    n  BINARY_INTEGER;
...
  FUNCTION median (num_tab NumTabTyp, n INTEGER)
    RETURN REAL IS
  BEGIN
    -- compute median
  END;
  BEGIN
    n := 100;
    median_salary := median(:salary, n);
    ...
  END;
END-EXEC;
...
```



注意:

動的 SQL 方法 4 では、TABLE 型のパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。

PL/SQL表のすべての行の値をホスト配列の対応する要素に割り当てる場合にも、プロシージャ・コールを使用できます。

表7-1に、PL/SQL表の行の値とホスト配列のエレメント間での有効な変換を示します。たとえば、LONG型のホスト配列は、VARCHAR2、LONG、RAWまたはLONG RAW型のPL/SQL表と互換性があります。ただし、CHAR型のPL/SQL表とは互換性がないため注意してください。

表7-1 正当なデータ型変換

PL/SQL表→ホスト配列	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	X	-	-	-	-	-	-	-
CHARZ	X	-	-	-	-	-	-	-
DATE	-	X	-	-	-	-	-	-

PL/SQL表→ホスト配列	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
DECIMAL	-	-	-	-	X	-	-	-
DISPLAY	-	-	-	-	X	-	-	-
FLOAT	-	-	-	-	X	-	-	-
INTEGER	-	-	-	-	X	-	-	-
LONG	X	-	X	-	-	-	-	-
LONG VARCHAR	-	-	X	X	-	X	-	X
LONG VARRAW	-	-	-	X	-	X	-	-
NUMBER	-	-	-	-	X	-	-	-
RAW	-	-	-	X	-	X	-	-
ROWID	-	-	-	-	-	-	X	-
STRING	-	-	X	X	-	X	-	X
UNSIGNED	-	-	-	-	X	-	-	-
VARCHAR	-	-	X	X	-	X	-	X
VARCHAR2	-	-	X	X	-	X	-	X
VARNUM	-	-	-	-	X	-	-	-
VARRAW	-	-	-	X	-	X	-	-

注意:

Pro*C/C++プリコンパイラでは、ホスト配列の使用方法はチェックされません。たとえば、索引範囲チェックは実

行されません。

関連項目

- 方法4の使用方法
- ストアドPL/SQLおよびJavaサブプログラム

7.5.1 ARRAYLEN文

入力ホスト配列をPL/SQLブロックに渡して処理するとします。デフォルトでは、入力ホスト配列をバインドすると、Pro*C/C++プリコンパイラは、その宣言されたサイズを使用します。ただし、配列全体を処理する必要がない場合があります。この場合には、ARRAYLEN文を使用して、より小さい配列サイズを指定できます。ARRAYLEN文では、ホスト配列をホスト変数と対応付け、そのホスト変数がより小さいサイズを格納します。文の構文は次のとおりです。

```
EXEC SQL ARRAYLEN host_array (dimension) [EXECUTE];
```

*dimension*は4バイトの整数型ホスト変数です。リテラルや式ではありません。

EXECUTEはオプションのキーワードです。

ARRAYLEN文は*host_array*および*dimension*の宣言とともに(ただし、それらの宣言よりも後に)表示する必要があります。ホスト配列にオフセットは指定できません。しかし、この目的にC言語の機能が使用できる場合もあります。次の例では、ARRAYLENを使用して、*bonus*という名前のCホスト配列のデフォルトのサイズを上書きしています。

```
float bonus[100];
int dimension;
EXEC SQL ARRAYLEN bonus (dimension);
/* populate the host array */
...
dimension = 25;  /* set smaller array dimension */
EXEC SQL EXECUTE
DECLARE
    TYPE NumTabTyp IS TABLE OF REAL
    INDEX BY BINARY_INTEGER;
    median_bonus  REAL;
    FUNCTION median (num_tab NumTabTyp, n INTEGER)
        RETURN REAL IS
  BEGIN
    -- compute median
  END;
  BEGIN
    median_bonus := median(:bonus, :dimension);
    ...
  END;
END-EXEC;
```

ARRAYLENでホスト配列のサイズが100要素から25要素に減少するため、25の配列要素のみがPL/SQLブロックに渡されます。その結果、PL/SQLブロックが実行のためOracleに送信されるとき、一緒に送られるホスト配列はずっと小さくなります。これにより、時間を節約し、ネットワーク化された環境でネットワークの通信量を削減できます。

7.5.2 オプション・キーワードEXECUTE

動的SQL方法2のEXEC SQL EXECUTE文で使用されるホスト配列には、オプション・キーワードEXECUTEの有無によって、2つの異なる解釈があります。

デフォルト(ARRAYLEN文にEXECUTEキーワードがないとき):

- ホスト配列はPL/SQLブロックが実行される回数が決定されるときに考慮されます。(最小値の配列サイズが使用されます。)
- ホスト配列はPL/SQL索引表に結合されません。

EXECUTEキーワードが存在しているとき:

- ホスト配列は索引表に結合されます。
- PL/SQLブロックは1回のみ実行されます。
- EXEC SQL EXECUTE文で指定されているすべてのホスト変数は、次のいずれかです。
  - ARRAYLEN ... EXECUTE文で指定
  - スカラー

たとえば、次のPL/SQLプロシージャを仮定します。

```
   CREATE OR REPLACE PACKAGE pkg AS
          TYPE tab IS TABLE OF NUMBER(5) INDEX BY BINARY_INTEGER;
          PROCEDURE proc1 (parm1 tab, parm2 NUMBER, parm3 tab);
   END;
```

次のPro*C/C++ファンクションは、特定のPL/SQLブロックを実行する回数を決定するためにホスト配列を使用する方法を示しています。この場合、PL/SQLブロックはemp表に新しい行が3行あるために3回実行されます。

```
func1()
{
  int empno_arr[5] = {1111, 2222, 3333, 4444, 5555};
  char *ename_arr[3] = {"MICKEY", "MINNIE", "GOOFY"};
  char *stmt1 = "BEGIN INSERT INTO emp(empno, ename) VALUES :b1, :b2; END;";

  EXEC SQL PREPARE s1 FROM :stmt1;
  EXEC SQL EXECUTE s1 USING :empno_arr, :ename_arr;
}
```

次のPro*C/C++ファンクションは、動的方法2でホスト配列をPL/SQL索引表にバインドする方法を示しています。EXEC SQL EXECUTE文に指定されたすべてのホスト配列についてARRAYLEN...EXECUTE文があることに注意してください。

```
func2()
{
  int ii = 2;
  int int_tab[3] = {1,2,3};
  int dim = 3;
  EXEC SQL ARRAYLEN int_tab (dim) EXECUTE;

  char *stmt2 = "begin pkg.proc1(:v1, :v2, :v3); end; ";

  EXEC SQL PREPARE s2 FROM :stmt2;
  EXEC SQL EXECUTE s2 USING :int_tab, :ii, :int_tab;
}
```

次のPro*C/C++ファンクションは、int_arrのARRAYLEN...EXECUTE文がないために、プリコンパイル時警告を生じます。

```
func3()
```

```
{
  int int_arr[3];
  int int_tab[3] = {1,2,3};
  int dim = 3;
  EXEC SQL ARRAYLEN int_tab (dim) EXECUTE;

  char *stmt3 = "begin pkg.proc1(:v1, :v2, :v3); end; ";

  EXEC SQL PREPARE s3 FROM :stmt3;
  EXEC SQL EXECUTE s3 USING :int_tab, :int_arr, :int_tab;
}
```

関連項目

- 方法2の使用方法
- ホスト配列

7.6 埋込みPL/SQLでのカーソルの使用

プログラムで同時に使用できるカーソルの最大数は、データベース初期化パラメータOPEN_CURSORSによって設定されます。埋込みPL/SQLブロックの実行中に、1つのカーソル、親カーソルがブロック全体に対応付けられ、1つのカーソル、子カーソルが埋込みPL/SQLブロックの各SQL文に対応付けられます。親カーソルと子カーソルは、両方とも、OPEN_CURSORS制限にカウントされます。

次の計算は、使用するカーソルの最大数を決定する方法を示します。カーソル数の合計がOPEN_CURSORSを超えないようにします。

```
   SQL statement cursors
   PL/SQL parent cursors
   PL/SQL child cursors
+  6 cursors for overhead
---------------------------
   Sum of cursors in use
```

プログラムで、OPEN_CURSORSによって設定された制限を超える数のカーソルが使用された場合、エラーが発生します。

関連項目

- 埋込みPL/SQLに関する考慮事項

7.7 ストアドPL/SQLおよびJavaサブプログラム

無名ブロックとは異なり、PL/SQLサブプログラム(プロシージャおよびファンクション)およびJavaメソッドは別個にコンパイルされ、Oracleデータベースに格納されて起動されます。

SQL*PlusなどのOracleツールを使用して明示的に作成したサブプログラムを、ストアド・サブプログラムと呼びます。ストアド・サブプログラムは、一度コンパイルしてデータ・ディクショナリに格納しておくと、再コンパイルしなくても再実行できるデータベース・オブジェクトになります。

PL/SQLブロック内のサブプログラムまたはストアド・プロシージャがアプリケーションによってOracleに送られると、それはインライン・サブプログラムと呼ばれます。Oracleは、インライン・サブプログラムをコンパイルし、システム・グローバル領域(SGA)にキャッシュしますが、データ・ディクショナリへのソース・コードまたはオブジェクト・コードの格納はしません。

パッケージ内で定義されているサブプログラムは、そのパッケージの一部とみなされ、パッケージ・サブプログラムと呼ばれます。パッケージで定義されていないストアド・サブプログラムはスタンドアロン・サブプログラムと呼ばれます。

関連項目

- *Oracle DatabaseでのJavaアプリケーション*

7.7.1 ストアド・サブプログラムの作成について

次の例に示すように、SQL文CREATE FUNCTION、CREATE PROCEDUREおよびCREATE PACKAGEをホスト・プログラムに埋め込むことができます。

```
EXEC SQL CREATE
FUNCTION sal_ok (salary REAL, title CHAR)
RETURN BOOLEAN AS
min_sal  REAL;
max_sal  REAL;
  BEGIN
    SELECT losal, hisal INTO min_sal, max_sal
        FROM sals
        WHERE job = title;
    RETURN (salary >= min_sal) AND
           (salary <= max_sal);
  END sal_ok;
END-EXEC;
```

埋込みCREATE {FUNCTION | PROCEDURE | PACKAGE}文は混成であることに注意してください。他のすべてのCREATE埋込み文と同様、キーワードEXEC SQL (EXEC SQL EXECUTEではありません)で始まります。ただし、PL/SQL終了記号のEND-EXECで終了する点は、他のCREATE埋込み文とは異なります。

次の例では、EMP表からひとまとまりの行をフェッチする*get_employees*というプロシージャを含むパッケージを作成します。バッチ・サイズは、プロシージャのコール元(別のストアド・サブプログラムの場合もあれば、クライアント・アプリケーションの場合もある)によって決められます。

プロシージャでは、3つのPL/SQL表をOUT仮パラメータとして宣言し、その後雇用データのバッチをPL/SQL表にフェッチします。一致する実パラメータはホスト配列です。プロシージャの終了時には、PL/SQL表のすべての行の値が、ホスト配列の対応する要素に自動的に割り当てられます。

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
    TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    TYPE NumArrayTyp IS TABLE OF FLOAT
        INDEX BY BINARY_INTEGER;
  PROCEDURE get_employees(
    dept_number IN     INTEGER,
    batch_size  IN     INTEGER,
    found       IN OUT INTEGER,
    done_fetch  OUT    INTEGER,
    emp_name    OUT    CharArrayTyp,
    job_title   OUT    CharArrayTyp,
    salary      OUT    NumArrayTyp);
  END emp_actions;
END-EXEC;
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS

    CURSOR get_emp (dept_number IN INTEGER) IS
        SELECT ename, job, sal FROM emp
            WHERE deptno = dept_number;

  PROCEDURE get_employees(
    dept_number IN     INTEGER,
```

```
    batch_size  IN     INTEGER,
    found       IN OUT INTEGER,
    done_fetch  OUT    INTEGER,
    emp_name    OUT    CharArrayTyp,
    job_title   OUT    CharArrayTyp,
    salary      OUT    NumArrayTyp) IS

  BEGIN
    IF NOT get_emp%ISOPEN THEN
        OPEN get_emp(dept_number);
    END IF;
    done_fetch := 0;
    found := 0;
    FOR i IN 1..batch_size LOOP
        FETCH get_emp INTO emp_name(i),
        job_title(i), salary(i);
        IF get_emp%NOTFOUND THEN
            CLOSE get_emp;
            done_fetch := 1;
            EXIT;
        ELSE
            found := found + 1;
        END IF;
    END LOOP;
  END get_employees;
END emp_actions;
END-EXEC;
```

CREATE文でREPLACE句を指定すると、パッケージの削除、再作成および権限再付与を実行しなくても既存のパッケージを再定義できます。CREATE文の構文の詳細は、SQL文: COMMIT to CREATE JAVAを参照してください。

埋込みCREATE {FUNCTION | PROCEDURE | PACKAGE}文が失敗した場合、Oracleはエラーではなく警告を発行します。

7.7.2 ストアドPL/SQLまたはJavaサブプログラムのコールについて

ホスト・プログラムからストアド・サブプログラムをコールするには、無名PL/SQLブロックまたはCALL埋込みSQL文のいずれかを使用できます。

7.7.2.1 無名PL/SQLブロック

次の例では、*raise_salary*という名前のスタンドアロン・プロシージャをコールします。

```
EXEC SQL EXECUTE
  BEGIN
    raise_salary(:emp_id, :increase);
  END;
END-EXEC;
```

ストアド・サブプログラムにパラメータを組み込めることに注意してください。この例では、実パラメータ*emp_id*および*increase*はC言語のホスト変数です。

次の例では、プロシージャ*raise_salary*が*emp_actions*の名前のパッケージに格納されます。したがって、プロシージャ・コールを完全に修飾するにはドット表記法を使用する必要があります。

```
EXEC SQL EXECUTE
BEGIN
    emp_actions.raise_salary(:emp_id, :increase);
```

```
END;
END-EXEC;
```

IN実パラメータには、リテラル、スカラー・ホスト変数、ホスト配列、PL/SQL定数、PL/SQL変数、PL/SQL表、PL/SQLユーザー定義レコード、プロシージャ・コールまたは式を使用できます。しかし、OUT実パラメータは、リテラル、プロシージャ・コールまたは式にしないでください。

埋込みPL/SQLブロックとともにプリコンパイラ・オプションSQLCHECK=SEMANTICSを使用する必要があります。

次の例では、仮パラメータのうち3つがPL/SQL表であり、対応する実パラメータはホスト配列です。プログラムでは、ストアド・プロシージャ*get_employees*を繰り返しコールし、データがなくなるまで、従業員データの各バッチを表示します。このプログラムは、demoディレクトリのsample9.pcファイルに入っており、オンラインで利用できます。CALLDEMOストアド・パッケージ作成用のSQLスクリプトは、calldemo.sqlファイルに入っています。

```
/************************************************************
Sample Program 9:  Calling a stored procedure

This program connects to ORACLE using the SCOTT/TIGER
account.  The program declares several host arrays, then
calls a PL/SQL stored procedure (GET_EMPLOYEES in the
CALLDEMO package) that fills the table OUT parameters.  The
PL/SQL procedure returns up to ASIZE values.

Sample9 keeps calling GET_EMPLOYEES, getting ASIZE arrays
each time, and printing the values, until all rows have been
retrieved.  GET_EMPLOYEES sets the done_flag to indicate "no
more data."
************************************************************/
#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE sqlca.h;


typedef char asciz[20];
typedef char vc2_arr[11];

EXEC SQL BEGIN DECLARE SECTION;
/* User-defined type for null-terminated strings */
EXEC SQL TYPE asciz  IS STRING(20) REFERENCE;

/* User-defined type for a VARCHAR array element. */
EXEC SQL TYPE vc2_arr IS VARCHAR2(11) REFERENCE;

asciz     username;
asciz     password;
int       dept_no;              /* which department to query? */
vc2_arr   emp_name[10];            /* array of returned names */
vc2_arr   job[10];
float     salary[10];
int       done_flag;
int       array_size;
int       num_ret;                 /* number of rows returned */
EXEC SQL END DECLARE SECTION;

long      SQLCODE;
```

```
void print_rows();            /* produces program output      */
void sql_error();             /* handles unrecoverable errors */



main()
{
   int   i;
   char  temp_buf[32];

/* Connect to ORACLE. */
   EXEC SQL WHENEVER SQLERROR DO sql_error();
   strcpy(username, "scott");
   strcpy(password, "tiger");
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("¥nConnected to ORACLE as user: %s¥n¥n", username);
   printf("Enter department number: ");
   gets(temp_buf);
   dept_no = atoi(temp_buf);/* Print column headers. */
   printf("¥n¥n");
   printf("%-10.10s%-10.10s%s¥n", "Employee", "Job", "Salary");
   printf("%-10.10s%-10.10s%s¥n", "--------", "---", "------");

/* Set the array size. */
   array_size = 10;

   done_flag = 0;
   num_ret = 0;

/*  Array fetch loop.
 *  The loop continues until the OUT parameter done_flag is set.
 *  Pass in the department number, and the array size--
 *  get names, jobs, and salaries back.
 */
   for (;;)
   {
      EXEC SQL EXECUTE
         BEGIN calldemo.get_employees
            (:dept_no, :array_size, :num_ret, :done_flag,
             :emp_name, :job, :salary);
         END;
      END-EXEC;

      print_rows(num_ret);

      if (done_flag)
         break;
   }

/* Disconnect from the database. */
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}
void
print_rows(n)
int n;
{
   int i;

    if (n == 0)
```

```
    {
        printf("No rows retrieved.¥n");
        return;
    }

    for (i = 0; i < n; i++)
        printf("%10.10s%10.10s%6.2f¥n",
               emp_name[i], job[i], salary[i]);
}

/* Handle errors. Exit on any error. */
void
sql_error()
{
   char msg[512];
   int buf_len, msg_len;


   EXEC SQL WHENEVER SQLERROR CONTINUE;

   buf_len = sizeof(msg);
   sqlglm(msg, &buf_len, &msg_len);

   printf("¥nORACLE error detected:");
   printf("¥n%.*s ¥n", msg_len, msg);

   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

それぞれの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に変換可能であることが必要です。また、ストアド・プロシージャの終了前には、すべてのOUT仮パラメータは割り当てられた値である必要があります。そうしないと、対応する実パラメータの値が未確定になります。

無名PL/SQLブロックを使用する場合、SQLCHECK=SEMANTICSは必須です。

関連項目

- ストアド・サブプログラムの作成について

7.7.2.2 リモート・アクセス

PL/SQLを使用すると、データベース・リンクを使用してリモート・データベースにアクセスできます。一般的に、データベース・リンクは、データベース管理者(DBA)によって設定され、Oracleデータ・ディクショナリに格納されます。データベース・リンクは、リモート・データベースの位置、リモート・データベースへのパス、使用するOracleユーザー名およびパスワードをOracleに伝えます。次の例では、データベース・リンク*dallas*を使用して、*raise_salary*プロシージャをコールします。

```
EXEC SQL EXECUTE
BEGIN
    raise_salary@dallas(:emp_id, :increase);
END;
END-EXEC;
```

次の例に示すように、シノニムを作成して、リモート・サブプログラムに位置の透過性を与えることができます。

```
CREATE PUBLIC SYNONYM raise_salary
FOR raise_salary@dallas;
```

7.7.2.3 CALL文

前述の埋込みPL/SQLブロックの概念はCALL文にも当てはまります。CALL埋込みSQL文の書式は次のようになります。

```
EXEC SQL
   CALL [schema.] [package.]stored_proc[@db_link](arg1, ...)
   [INTO :ret_var [[INDICATOR]:ret_ind]] ;
```

パラメータは次のとおりです。

schema

プロシージャを含むスキーマ

package

プロシージャを含むパッケージ

stored_proc

コールするJavaまたはPL/SQLストアド・プロシージャ

db_link

オプションのリモート・データベース・リンク

arg1...

引き渡す一連の引数(変数、リテラル、式)

ret_var

結果を受け取るオプションのホスト変数

ind_var

ret_varのオプションの標識変数。

CALL文には、SQLCHECK=SYNTAXまたはSEMANTICSのいずれかを使用できます。

7.7.2.4 CALLの例

次に示すように、入力された整数を受け取り、その階乗を整数で戻すPL/SQLファンクションfact(パッケージmathpkgに格納されています)を作成済とします。

```
     EXEC SQL CREATE OR REPLACE PACKAGE BODY mathpkg as
       function fact(n IN INTEGER) RETURN INTEGER AS
         BEGIN
           IF (n <= 0) then return 1;
           ELSE return n * fact(n - 1);
           END IF;
         END fact;
       END mathpkge;
     END-EXEC.
```

次に、そのCALL文を使用しているPro*C/C++アプリケーションで、factを使用します。

```
 ...
int num, fact;
...
EXEC SQL CALL mathpkge.fact(:num) INTO :fact ;
...
```

関連項目:

- CALL文の詳細は、CALL(実行可能埋込みSQL)を参照してください。
- 引数の渡し方およびその他の問題の詳細は、『*Oracle Database*アドバンスト・アプリケーション開発者ガイド』を参照してください。

7.7.3 ストアド・サブプログラムに関する情報を得る方法について

> 注意:
>
> ログイン・データ領域(LDA)は、Oracle ではサポートされなくなりました。Oracle の次のバージョンでは、Pro*C/C++プログラムに OCI リリース 7 コールを埋め込む機能は廃止になります。

データ型とホスト変数では、ホスト・プログラムにOCIコールを埋め込む方法を説明しています。ライブラリ・ルーチンSQLLDAをコールしてLDAを設定すると、OCIコール*odessp*を使用して、ストアド・サブプログラムに関する有益な情報を取得できます。*odessp*をコールするときには、有効なLDAとサブプログラムの名前を渡す必要があります。パッケージ・サブプログラムの場合は、パッケージ名も渡す必要があります。*odessp*からは、各サブプログラム・パラメータに関する情報(データ型、サイズ、位置など)が戻されます。

DBMS_DESCRIBEパッケージでは、DESCRIBE_PROCEDUREストアド・プロシージャを使用できます。

7.8 外部プロシージャ

PL/SQLでは、外部プロシージャのC言語関数をコールできます。外部プロシージャは、Dynamic Link Library (DLL)またはSolarisの*.so*ライブラリなどに格納されています。

外部プロシージャをサーバーで実行する場合、同一トランザクション中でSQLおよびPL/SQLが実行されるようにサーバーにコールバックできます。サーバーで外部プロシージャを実行すると、クライアントで実行した場合よりも処理速度が速く、外部システムおよびデータ・ソースと、データベース・サーバーとのインタフェースとして使用できます。

サーバー側の外部C言語関数を実行する場合は、関数内でREGISTER CONNECT埋込みSQL文を使用する必要があります。文の構文は次のとおりです。

```
EXEC SQL REGISTER CONNECT USING :epctx [RETURNING :host_context] ;
```

epctxは、OCIExtProcContextへのタイプ・ポインタの外部プロシージャ・コンテキストです。epctxはPL/SQLによってプロシージャに渡されます。

host_contextは、外部プロシージャによって戻されるランタイム・コンテキストです。現在の設定は、デフォルト(グローバル)・コンテキストです。

REGISTER CONNECT文により、カレントのOracle接続およびトランザクションに対応付けられたOCIハンドル・セット(OCIEnv、OCISvcCtxおよびOCIError)が戻されます。これらのハンドルは、グローバルSQLLIBランタイム・コンテキストのPro*C/C++デフォルトである名前なし接続の定義に使用されます。このため、REGISTER CONNECTが、CONNECT文のかわりに使用されます。

後続の埋込みSQL文では、このOCIハンドル・セットを使用します。後続の埋込みSQL文は、グローバルSQLLIBランタイム・コンテキスト、名前なし接続に対して実行されます。これは、別々にプリコンパイルされたプログラム・ユニットにある場合でも同様です。コミットされていない変更は無効です。今後のバージョンでは、(非デフォルトの)ランタイム・コンテキストはオプションのRETURNING句で返されるようになります。

グローバル・ランタイム・コンテキストのアクティブなデフォルト接続はまだありません。すでに接続が確立しているときにREGISTER CONNECTを使用すると、ランタイム・エラーが戻されます。

実際の業務では、外部プロシージャは複数の異なるアプリケーションから再利用できるようにすることをお薦めします。

関連項目

- Oracle Call Interfaceプログラマーズ・ガイド

7.8.1 外部プロシージャの制限

外部プロシージャには次の規則があります。

- 外部プロシージャを使用できるのはCのみです。C++外部プロシージャはサポートされていません。
- 外部プロシージャ・コンテキストに接続した場合、接続を追加できません。ランタイム・エラーが発生します。
- マルチスレッドの外部プロシージャはサポートされていません。EXEC SQL ENABLE THREADS文は使用できません。ランタイム・エラーが発生します。Pro*C/C++では、ここで説明している外部プロシージャを使用しない場合は、アプリケーションでのマルチスレッドがサポートされます。
- DDL文を使用できません。ランタイム・エラーが発生します。
- EXEC SQL COMMITおよびEXEC SQL ROLLBACKなどのトランザクション制御文を使用できません。
- EXEC SQL OBJECTなどのオブジェクト・ナビゲーション文を使用できません。
- EXEC SQL LOB文のポーリングを行うことはできません。
- EXEC TOOLS文を使用できません。ランタイム・エラーが発生します。

7.8.2 外部プロシージャの作成について

外部プロシージャextp1を作成する場合の簡単な例を示します。

外部Cプロシージャを格納するには、コードのコンパイルおよびリンクを行い、DLLなどのライブラリに格納します。

NTの参照でユーザー・コメント9561が削除されました

次のSQLコマンドを1回実行して、外部プロシージャextp1を登録します。

```
CREATE OR REPLACE PROCEDURE extp1
AS EXTERNAL NAME "extp1"
LIBRARY mylib
WITH CONTEXT
PARAMETERS(CONTEXT) ;
```

*mylib*は、プロシージャ*extp1*が格納されるライブラリです。WITH CONTEXTが指定されているため、このプロシージャは、引数型OCIExtProcContext*で暗黙的にコールされます。このコールでは、コンテキストが省略されていますが、プロシージャには渡されます。ただし、CREATE文のCONTEXTキーワードは、プレース・マーカーとして指定されています。

このコンテキスト・パラメータは、extp1の内側のEXEC SQL REGISTER CONNECT文で参照されます。

外部プロシージャは、SQL*Plusから次のようにコールされます。

```
SQL>
BEGIN
  INSERT INTO emp VALUES(9999,'JOHNSON','SALESMAN',7782, sysdate, 1200,150,10);
  extp1;
END;
```

extp1.pcのリストです。

```
void extp1 (epctx)
OCIExtProcContext *epctx;
{
char name[15];
        EXEC SQL REGISTER CONNECT USING :epctx;
        EXEC SQL WHENEVER SQLERROR goto err;
        EXEC SQL SELECT ename INTO :name FROM emp WHERE empno = 9999;
        return;
err: SQLExtProcError(SQL_SINGLE_RCTX,sqlca.sqlerrm.sqlerrmc,sqlca.sqlerrm.sqlerrml);
        return;
}
```

関連項目

- 外部サブプログラム

7.8.3 SQLExtProcError()

SQLLIB関数SQLExtProcError()を使用すると、外部Cプロシージャでエラーが発生した場合にPL/SQLに制御を戻すことができます。関数とその引数は次のとおりです。

SQLExtProcError (ctx, msg, msglen)

各パラメータの意味は次のとおりです。

ctx (IN) sql_context *

この関数は、REGISTER CONNECT文のターゲットSQLLIBランタイム・コンテキストです。REGISTER CONNECT文は、この関数が起動される前に実行する必要があります。現在、グローバル・ランタイム・コンテキストのみがサポートされています。

msg (OUT) char *

エラー・メッセージのテキスト

msglen (OUT) size_t

メッセージのバイト単位の長さ

この関数が実行されると、SQLLIBによりOCIサービス関数OCIExtProcRaiseExcpWithMsgがコールされます。

メッセージは、SQLCAの構造体sqlerrmから出力されます。

SQLExtProcError()の使用方法の例です。

```
void extp1 (epctx)
OCIExtProcContext *epctx;
{
   char name[15];
   EXEC SQL REGISTER CONNECT USING :epctx;
   EXEC SQL WHENEVER SQLERROR goto err;
   EXEC SQL SELECT ename INTO :name FROM emp WHERE smpno = 9999;
   return;
 err:
```

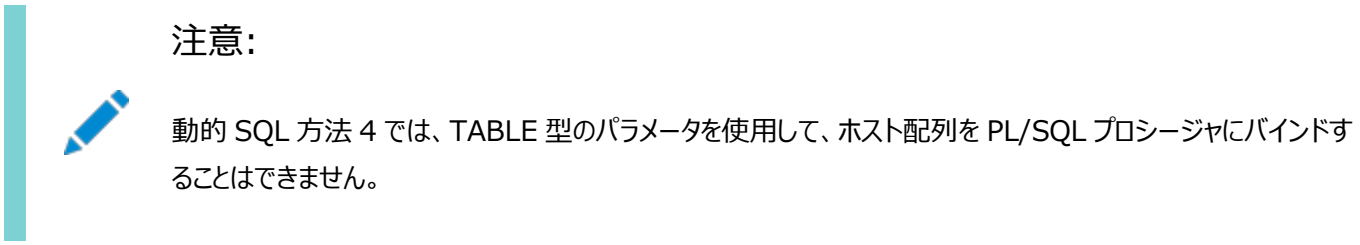
```
    SQLExtProcError (SQL_SINGLE_RCTX, sqlca.sqlerrm.sqlerrmc,
       sqlca.sqlerrm.sqlerrml);
    printf("¥n%*s¥n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    return;
}
```

関連項目

- SQLCAの構造体

7.9 動的SQLの使用方法について

プリコンパイラでは、PL/SQLブロック全体が1つのSQL文として処理されます。つまり、PL/SQLブロックをホスト変数の文字列に格納できることを意味します。この場合、ブロックにホスト変数が含まれない場合は、動的SQL方法1を使用して、PL/SQL文字列をEXECUTEできます。ブロックにホスト変数が含まれる場合、変数の数がわかっているときは、動的SQL方法2を使用してPL/SQL文字列をPREPAREおよびEXECUTEできます。ブロックに含まれるホスト変数の数がわからない場合は、動的SQL方法4を使用する必要があります。



注意:

動的 SQL 方法 4 では、TABLE 型のパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。

関連項目

- ANSI動的SQL
- Oracle動的SQL
- Oracle動的SQL: 方法4
- 方法4の使用方法

8 ホスト配列

この章では、配列を使用してコーディングを簡略化し、プログラムのパフォーマンスを改善する方法について説明します。配列を使用してOracleデータを操作する方法、単一のSQL文を使用して配列内のすべての要素を処理する方法および処理対象となる配列の要素数を制限する方法を説明します。この章の内容は次のとおりです。

- 配列を使用する理由
- ホスト配列の宣言について
- SQL文での配列の使用方法について
- 配列への選択について
- 配列での挿入について
- 配列での更新について
- 配列での削除について
- FOR句の使用方法について
- WHERE句の使用方法について
- 構造体配列
- CURRENT OFの疑似実行について
- sqlca.sqlerrd[2]の使用方法について
- 追加の配列の挿入/選択構文の使用について
- 暗黙的なバッファ済INSERTの使用について

8.1 配列を使用する理由

配列を使用すると、プログラミングの所要時間が短縮され、パフォーマンスを改善できます。

配列により、単一のSQL文で配列全体を操作できます。このため、特にネットワーク環境では、Oracleの通信オーバーヘッドが大幅に軽減されます。実行時間の大部分は、ネットワーク上でクライアント・プログラムとサーバー・データベース間のラウンドトリップとに費やされます。配列を使用すると、ラウンドトリップが減少します。

たとえば、およそ300人の従業員に関する情報をEMPという表に挿入する必要があるとします。配列がないと、プログラムは300の個々のINSERT(各従業員に1つ)を実行する必要があります。配列を使用すれば、INSERTの実行は1回で済みます。

8.2 ホスト配列の宣言について

次の例では3つのホスト配列を宣言するとともに、それぞれ要素の最大数を50に設定しています。

```
char  emp_name[50][10];
int   emp_number[50];
float salary[50];
```

VARCHARの配列も有効です。次の宣言は、有効なホスト言語宣言です。

```
VARCHAR v_array[10][30];
```

8.2.1 制限事項(ホスト配列の宣言)

オブジェクト型を除き、ポインタのホスト配列は宣言できません。

文字配列(文字列)を除き、SQL文内で参照できるホスト配列は1次元に限定されます。したがって、次の例で宣言されている2次元配列は無効です。

```
int hi_lo_scores[25][25];   /* not allowed */
```

8.2.2 配列の最大サイズ

1つのSQL文で、1回のフェッチ操作によってアクセスできる配列要素の最大数は32000です(使用するプラットフォームと使用可能なメモリーによっては、32000以上になります)。最大数を超える数のホスト表要素にアクセスすると、パラメータが範囲外であることを示すランタイム・エラーが表示されます。文が無名PL/SQLブロックの場合、アクセス可能な配列要素の数は、32512をデータ型のサイズで割った数に制限されます。

8.3 SQL文での配列の使用方法について

ホスト配列は、INSERT文、UPDATE文、DELETE文では入力変数として、またSELECT文およびFETCH文のINTO句では出力変数として使用できます。

ホスト配列に使用される埋込みSQL構文は、単純ホスト変数に使用される埋込みSQL構文とほとんど同じです。ただし、オプションのFOR句で配列処理が制御できるという点に違いがあります。また、ホスト配列と単純ホスト変数を1つのSQL文で併用するときにも制限があります。

後続の項では、データ操作文でのホスト配列の使用方法を説明します。

8.3.1 ホスト配列の参照について

単一のSQL文で複数のホスト配列を使用する場合、要素の数は同じにする必要があります。同じでない場合には、プリコンパイル時に「配列サイズが一致しません」という警告メッセージが出ます。この警告を無視すると、プリコンパイラではSQL操作で最小数の要素が使用されます。

次の例では、INSERTされるのは25行のみです。

```
int    emp_number[50];
char   emp_name[50][10];
int    dept_number[25];
/* Populate host arrays here. */

EXEC SQL INSERT INTO emp (empno, ename, deptno)
    VALUES (:emp_number, :emp_name, :dept_number);
```

SQL文のホスト配列に添字を付け、それをループで使用することでデータをINSERTまたはフェッチできます。たとえば、次のようなループを使用して、配列内の5番目の要素ごとにINSERTできます。

```
for (i = 0; i < 50; i += 5)
    EXEC SQL INSERT INTO emp (empno, deptno)
        VALUES (:emp_number[i], :dept_number[i]);
```

ただし、処理する必要のある配列要素が連続している場合は、ループでホスト配列を処理しないでください。単に、添字の付いていない配列名をSQL文で使用してください。要素数*n*のホスト配列を含むSQL文は、*n*個の異なるスカラー変数を持つ同じSQL文として*n*回実行するのと同様に扱われます。

8.3.2 インジケータ配列の使用方法について

インジケータ配列は、NULLを割り当ててホスト配列を入力し、出力ホスト配列でNULLまたは切り捨てられた値(文字列のみ)を検出する場合に使用できます。次の例は、インジケータ配列でINSERTを行う方法を示しています。

```
int    emp_number[50];
int    dept_number[50];
float  commission[50];
short  comm_ind[50];       /* indicator array */

/* Populate the host and indicator arrays.  To insert a null
   into the comm column, assign -1 to the appropriate
   element in the indicator array. */
    EXEC SQL INSERT INTO emp (empno, deptno, comm)
        VALUES (:emp_number, :dept_number,
        :commission INDICATOR :comm_ind);
```

8.3.3 Oracle制限事項(ホスト配列用)

VALUES、SET、INTOまたはWHERE句では、スカラーのホスト変数とホスト配列を併用できません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。

ホスト配列は、UPDATE文またはDELETE文でCURRENT OF句とともに使用できません。

8.3.4 ANSIでの制限および要件

配列インタフェースは、ANSI/ISOの埋込みSQL規格に対するOracle拡張機能です。ただし、MODE=ANSIでプリコンパイルすると、配列のSELECTおよびFETCHは使用できます。必要であれば、FIPSフラガー・プリコンパイラ・オプションによって、配列を使用していることをフラグで示すことができます。

配列をSELECTおよびFETCHするときは、必ずインジケータ配列を使用します。このようにして、関連する出力ホスト配列内にNULLがあるかどうかをテストできます。

DBMS=V7またはDBMS=v8のときに、インジケータ配列に対応付けられていないホスト配列にNULL列値をSELECTまたはFETCHすると、Oracleは処理を停止し、sqlerrd[2]に処理済行数を設定しエラー・メッセージを出します。DBMS=V7またはDBMS=v8の場合、Oracleは切捨てをエラーとみなしません。

また、SELECTまたはFETCHの結果、NULLの使用によるORA-24347などの警告が発生した場合や、列にインジケータ配列がない場合には、Oracleは処理を停止します。

注意:

SELECT または FETCH では、すべての列に標識変数を使用します。インジケータのない列がある場合は、プリコンパイラ・オプション unsafe_null=yes をかわりに使用できます。

8.4 配列への選択について

ホスト配列はSELECT文内の出力変数として使用できます。SELECTによって戻される最大行数がわかっている場合、その数の要素でホスト配列を宣言してください。次の例では、3つのホスト配列へのデータを直接選択します。このSELECTで戻される行が50行以下であることがわかっているため、配列は要素数50で宣言します。

```
char   emp_name[50][20];
int    emp_number[50];
float  salary[50];

EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO :emp_name, :emp_number, :salary
    FROM EMP
    WHERE SAL > 1000;
```

この例ではSELECT文は50行まで戻します。選択される行数が49行以下の場合、または50行のみを取り出す場合はこの方法を使用します。ただし、選択される行数が51行以上の場合は、この方法ではすべての行を取り出せません。このSELECT文を再実行しても、他に選択対象の行があるとしても、最初の50行のみがまた戻されます。この場合は大きな配列を宣言するか、FETCH文で使用するカーソルを宣言する必要があります。

宣言した要素数を超える行数がSELECT INTO文によって戻されると、SELECT_ERROR=NOを指定していないかぎりエラー・メッセージが出されます。

関連項目:

SELECT_ERRORオプションの詳細は、プリコンパイラのオプションを参照してください。

8.4.1 カーソルのフェッチ

SELECTが戻す最大行数がわからない場合には、カーソルを宣言して、一括でフェッチできます。

ループ内でバッチ・フェッチを実行すると、多数の行を簡単に取り出せます。FETCHを実行するたびに、次に続く行の集合がカレント・アクティブ・セットから戻されます。次の例では、20行ずつまとめて行をフェッチします。

```
int   emp_number[20];
float salary[20];

EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT empno, sal FROM emp;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_number, :salary;
    /* process batch of rows */
    ...
}
...
```

最後のフェッチで実際に戻された行数を必ずチェックして処理してください。

関連項目

- フェッチされる行数

8.4.2 sqlca.sqlerrd[2]の使用方法について

INSERT文、UPDATE文、DELETE文およびSELECT INTO文の場合は、sqlca.sqlerrd[2]に処理済行数が記録されます。FETCH文の場合は、処理した行の累積数が記録されます。

FETCHでホスト配列を使用しているときに、その時点での最後のループで戻された行数を確認するには、sqlca.sqlerrd[2]の現在の値と(別の変数内に保存した)前回の値との差分をとります。次の例では、最後のフェッチで戻された行数を確認します。

```
int  emp_number[100];
char emp_name[100][20];

int rows_to_fetch, rows_before, rows_this_time;
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT empno, ename
    FROM emp
    WHERE deptno = 30;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
/* initialize loop variables */
rows_to_fetch = 20;   /* number of rows in each "batch" */
rows_before = 0;      /* previous value of sqlerrd[2]  */
rows_this_time = 20;

while (rows_this_time == rows_to_fetch)
{
    EXEC SQL FOR :rows_to_fetch
    FETCH emp_cursor
        INTO :emp_number, :emp_name;
    rows_this_time = sqlca.sqlerrd[2] - rows_before;
    rows_before = sqlca.sqlerrd[2];
}
...
```

配列の処理中にエラーが発生したときにもsqlca.sqlerrd[2]が役立ちます。処理はエラーが発生した行で停止するため、sqlerrd[2]には正常に処理された行数が格納されます。

8.4.3 フェッチされる行数

各FETCHが戻すのは、最大でも配列の全行数分までです。次のような場合は最大行数より少ない行が戻ります。

- アクティブ・セットの最後に達したとき:「データが見つかりません。」というOracleエラー・コードがSQLCA内でSQLCODEに戻されます。たとえば、要素数100の配列に行をフェッチしたとき20行しか戻されなかった場合にこれが起こります。
- 残っているフェッチ対象の行が、一括フェッチの全行数より少ないとき。たとえば、要素数20の配列に70行をフェッチすると、3回目のFETCHの後にはフェッチ対象の行が10しか残っていないため、この状態が発生します。
- 行の処理中にエラーが検出されたとき。FETCHは失敗し、該当するOracleエラー・コードがSQLCODEに戻されます。

戻された行の累積数は、このガイドではsqlerrd[2]と記載しているSQLCA内の*sqlerrd*の3番目の要素に保存されます。これはオープン状態のすべてのカーソルに適用されます。次の例では、カーソルの状態がそれぞれ更新されている様子がわかります。

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 20 */
EXEC SQL FETCH cursor2 INTO :array_of_30;
```

```
/* now running total in sqlerrd[2] is 30, not 50 */
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 40 (20 + 20) */
EXEC SQL FETCH cursor2 INTO :array_of_30;
/* now running total in sqlerrd[2] is 60 (30 + 30) */
```

8.4.4 スクロール可能カーソルのフェッチ

ホスト配列はスクロール可能カーソルとともに使用することもできます。スクロール可能カーソルを使用する場合、sqlca.sqlerrd[2]は処理済最大(絶対)行数を表します。アプリケーションではスクロール可能モードでフェッチを任意の場所に配置できるため、この値が処理済行数の合計である必要はありません。

スクロール可能モードのFETCH文にホスト配列を使用している間は、sqlca.sqlerrd[2]の現在の値と前回の値との差分を取っても、その時点での最後のループで戻された行数を確認することはできません。アプリケーション・プログラムでは、FETCH LASTを実行して、結果セット内の合計行数を判断します。sqlca.sqlerrd[2]の値は、結果セット内の合計行数を示します。

関連項目

- サンプル・プログラム: スクロール可能カーソルを使用するホスト配列

8.4.5 サンプル・プログラム3: ホスト配列

この項のデモ・プログラムでは、Pro*C/C++で問合せを作成するときの配列の使用方法を示しています。特に、SQLCA(sqlca.sqlerrd[2])の処理済行数に注目してください。このプログラムは、demoディレクトリのファイルsample3.pcとして、オンラインで使用可能です。

```
/*
 *  sample3.pc
 *  Host Arrays
 *
 *  This program connects to ORACLE, declares and opens a cursor,
 *  fetches in batches using arrays, and prints the results using
 *  the function print_rows().
 */

#include <stdio.h>
#include <string.h>

#include <sqlca.h>

#define NAME_LENGTH   20
#define ARRAY_LENGTH   5
/* Another way to connect. */
char *username = "SCOTT";
char *password = "TIGER";

/* Declare a host structure tag. */
struct
{
    int    emp_number[ARRAY_LENGTH];
    char   emp_name[ARRAY_LENGTH][NAME_LENGTH];
    float  salary[ARRAY_LENGTH];
} emp_rec;

/* Declare this program's functions. */
void print_rows();              /* produces program output */
void sql_error();          /* handles unrecoverable errors */
```

```
main()
{
    int  num_ret;               /* number of rows returned */

/* Connect to ORACLE. */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Connect error:");

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("¥nConnected to ORACLE as user: %s¥n", username);


    EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error:");
/* Declare a cursor for the FETCH. */
    EXEC SQL DECLARE c1 CURSOR FOR
        SELECT empno, ename, sal FROM emp;

    EXEC SQL OPEN c1;

/* Initialize the number of rows. */
    num_ret = 0;

/* Array fetch loop - ends when NOT FOUND becomes true. */
    EXEC SQL WHENEVER NOT FOUND DO break;

    for (;;)
    {
        EXEC SQL FETCH c1 INTO :emp_rec;

/* Print however many rows were returned. */
        print_rows(sqlca.sqlerrd[2] - num_ret);
        num_ret = sqlca.sqlerrd[2];        /* Reset the number. */
    }
/* Print remaining rows from last fetch, if any. */
    if ((sqlca.sqlerrd[2] - num_ret) > 0)
        print_rows(sqlca.sqlerrd[2] - num_ret);

    EXEC SQL CLOSE c1;
    printf("¥nAu revoir.¥n¥n¥n");

/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}


void
print_rows(n)
int n;
{
    int i;

    printf("¥nNumber   Employee         Salary");
    printf("¥n------   --------         ------¥n");

    for (i = 0; i < n; i++)
        printf("%-9d%-15.15s%9.2f¥n", emp_rec.emp_number[i],
               emp_rec.emp_name[i], emp_rec.salary[i]);

}
```

```
void
sql_error(msg)
char *msg;
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("¥n%s", msg);
    printf("¥n% .70s ¥n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

関連項目:

SQLCAの詳細は、ランタイム・エラーの処理を参照してください。

8.4.6 サンプル・プログラム: スクロール可能カーソルを使用するホスト配列

このプログラムは、スクロール可能カーソルとともにホスト配列を使用する方法を示します。このプログラムは、demoディレクトリのファイルscdemo2.pcとして、オンラインで使用可能です。

注意:

結果セット内の行数の判断には FETCH LAST を実行していることに注意してください。

8.4.6.1 scdemo2.pc

```
/*
 *  A Sample program to demonstrate the use of scrollable
 *  cursors with host arrays.
 *
 *  This program uses the hr/hr schema.Make sure
 *  that this schema exists before executing this program
 */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

#define ARRAY_LENGTH   4

/* user and passwd */
char *username = "hr";
char *password = "hr";

/* Declare a host structure tag. */
```

```
struct emp_rec_array
{
    int    emp_number;
    char   emp_name[20];
    float  salary;
} emp_rec[ARRAY_LENGTH];


/* Print the result of the query */

void print_rows()
{
    int i;

    for (i=0; i<ARRAY_LENGTH; i++)
        printf("%d    %s  %8.2f¥n", emp_rec[i].emp_number,
             emp_rec[i].emp_name, emp_rec[i].salary);

}

/*  Oracle error handler */

void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("¥n%s", msg);
    printf("¥n% .70s ¥n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
}

void main()
{
    int noOfRows; /* Number of rows in the result set */

    /* Error handler */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Connect error:");

    /* Connect to the data base */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    /* Error handle */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error:");

    /* declare the cursor in scrollable mode */
    EXEC SQL DECLARE c1 SCROLL CURSOR FOR
        SELECT employee_id, first_name, salary FROM employees;

    EXEC SQL OPEN c1;

    EXEC SQL WHENEVER SQLERROR DO sql_error("Fetch Error:");

    /* This is a dummy fetch to find out the number of rows
       in the result set */
    EXEC SQL FETCH LAST c1 INTO :emp_rec;

    /* The number of rows in the result set is given by
       the value of sqlca.sqlerrd[2] */
```

```
    noOfRows = sqlca. sqlerrd[2];
    printf("Total number of rows in the result set %d:¥n",
             noOfRows);

    /* Fetch the first ARRAY_LENGTH number of rows */
    EXEC SQL FETCH FIRST c1 INTO :emp_rec;
    printf("******************** DEFAULT : ¥n");
    print_rows();

    /* Fetch the next set of ARRAY_LENGTH rows */
    EXEC SQL FETCH NEXT c1 INTO :emp_rec;
    printf("******************** NEXT  : ¥n");
    print_rows();

    /* Fetch a set of ARRAY_LENGTH rows from the 3rd row onwards */
    EXEC SQL FETCH ABSOLUTE 3 c1 INTO :emp_rec;
    printf("******************** ABSOLUTE 3 : ¥n");
    print_rows();

    /* Fetch the current ARRAY_LENGTH set of rows */
    EXEC SQL FETCH CURRENT c1 INTO :emp_rec;
    printf("******************** CURRENT : ¥n");
    print_rows();

    /* Fetch a set of ARRAY_LENGTH rows from the 2nd offset
       from the current cursor position */
    EXEC SQL FETCH RELATIVE 2 c1 INTO :emp_rec;
    printf("******************** RELATIVE 2 : ¥n");
    print_rows();

    /* Again Fetch the first ARRAY_LENGTH number of rows */
    EXEC SQL FETCH ABSOLUTE 0 c1 INTO :emp_rec;
    printf("******************** ABSOLUTE 0 : ¥n");
    print_rows();

    /* close the cursor */
    EXEC SQL CLOSE c1;

/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(EXIT_SUCCESS);
}
```

8.4.7 ホスト配列の制限

副問合せ文中を除き、SELECT文のWHERE句ではホスト配列を使用できません。例は、WHERE句の使用方法についてを参照してください。

またSELECT文またはFETCH文のINTO句では、単純ホスト変数とホスト配列を併用できません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。

表8-1は、SELECT INTO文で有効なホスト配列の使用を示しています。

表8-1 SELECT INTOで有効なホスト配列

INTO句	WHERE句	有効?

INTO句	WHERE句	有効?
配列	配列	いいえ
スカラー	スカラー	はい
配列	スカラー	はい
スカラー	配列	いいえ

8.4.8 NULL値のフェッチについて

配列をSELECTおよびFETCHするときは、必ずインジケータ配列を使用します。このようにして、関連する出力ホスト配列内にNULLがあるかどうかをテストできます。

DBMS = V7またはDBMS=v8のときに、インジケータ配列に対応付けられていないホスト配列にNULL列値をSELECTまたはFETCHすると、Oracleは処理を停止し、sqlerrd[2]に処理済行数を設定しエラー・メッセージを出します。

また、SELECTまたはFETCHの結果、NULLの使用による*ORA-24347*などの警告が発生した場合や、列にインジケータ配列がない場合には、Oracleは処理を停止します。SELECTまたはFETCHのすべての列で標識変数を使用します。インジケータのない列がある場合は、プリコンパイラ・オプションunsafe_null=yesをかわりに使用できます。

8.4.9 切り捨てられた値のフェッチについて

DBMS=V7のときは、切捨てによって警告メッセージが発行されますが、処理は継続されます。

また、配列をSELECTおよびFETCHするときは必ずインジケータ配列を使用します。そうすれば、Oracleで1つ以上の切り捨てられた列値が出力ホスト配列に割り当てられた場合に、関連付けられたインジケータ配列で列値の元の長さがわかります。

8.5 配列での挿入について

ホスト配列はINSERT文内の入力変数として使用できます。プログラムでINSERT文を実行する前に、プログラム内にデータが含まれている配列があるかどうかを確認してください。

配列内に不適切な要素があるときは、FOR句を使用してINSERT対象の行数を制御できます。

ホスト配列による挿入の例は次のとおりです。

```
char   emp_name[50][20];
int    emp_number[50];
float  salary[50];
/* populate the host arrays */
...
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:emp_name, :emp_number, :salary);
```

挿入された行の累積数は処理済行数sqlca.sqlerrd[2]に保存されます。

次の例では、一度に1行ずつINSERTされます。各行の挿入ごとにサーバーをコールする必要があるため、この方法は前の例に比べるとかなり効率は悪くなります。

```
for (i = 0; i < array_size; i++)
    EXEC SQL INSERT INTO emp (ename, empno, sal)
```

```
        VALUES (:emp_name[i], :emp_number[i], :salary[i]);
```

関連項目

- FOR句の使用について

8.5.1 配列での挿入の制限について

INSERT文のVALUES句内ではポインタ配列は使用できません。つまり配列要素はすべてデータ項目である必要があります。

INSERT文のVALUES句では、スカラー・ホスト変数とホスト配列を併用できません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。

8.6 配列での更新について

次の例に示すように、ホスト配列をUPDATE文内の入力変数として使用できます。

```
int   emp_number[50];
float salary[50];
/* populate the host arrays */
EXEC SQL UPDATE emp SET sal = :salary
    WHERE EMPNO = :emp_number;
```

更新された行の累積数はsqlerrd[2]に保存されます。この数には更新カスケードによって処理された行は含まれていません。

配列内に不適切な要素がある場合は、埋込みSQLのFOR句を使用して更新対象の行数を制御できます。

前の例では一意キー(EMP_NUMBER)を使用した一般的な更新を示しています。各配列要素で更新できる行は1行のみです。次の例では、各配列要素で複数の行を操作できます。

```
char  job_title [10][20];
float commission[10];

...

EXEC SQL UPDATE emp SET comm = :commission
    WHERE job = :job_title;
```

8.6.1 配列での更新の制限について

UPDATE文のSET句またはWHERE句では、単純ホスト変数とホスト配列の併用はお薦めしません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。 さらに、SET句でホスト配列を使用するときには、WHERE句の要素数と同じ数のものを使用してください。

UPDATE文のCURRENT OF句では、ホスト配列は使用できません。

表8-2は、UPDATE文で有効なホスト配列の使用方法を示しています。

表8-2 UPDATE文で有効なホスト配列

SET句	WHERE句	有効?
配列	配列	はい
スカラー	スカラー	はい

SET句	WHERE句	有効?
配列	スカラー	いいえ
スカラー	配列	いいえ

関連項目

- CURRENT OFの疑似実行について

8.7 配列での削除について

DELETE文でもホスト配列を入力変数として使用できます。これは、WHERE句内のホスト配列の連続した要素を使用して、DELETE文を繰り返し実行するのと同様です。つまり1回の実行で表から0行、1行または複数行が削除されます。

ホスト配列による削除の例は次のとおりです。

```
...
int emp_number[50];

/* populate the host array */
...
EXEC SQL DELETE FROM emp
    WHERE empno = :emp_number;
```

削除された行の累積数はsqlerrd[2]に保存されます。この数には削除カスケードによって処理された行は含まれていません。

この例では一意キー(EMP_NUMBER)を使用した一般的な削除を示しています。各配列要素で削除できる行は1行のみです。次の例では、各配列要素で複数の行を操作できます。

```
...
char job_title[10][20];

/* populate the host array  */
...
EXEC SQL DELETE FROM emp
    WHERE job = :job_title;
...
```

8.7.1 配列での削除の制限について

DELETE文のWHERE句では、単純ホスト変数とホスト配列を併用できません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。

DELETE文のCURRENT OF句ではホスト配列は使用できません。

関連項目:

代替方法については、CURRENT OFの擬似実行についてを参照してください。

8.8 FOR句の使用方法について

埋込みSQLでオプションのFOR句を使用すると、次に示すSQL文が処理する配列要素の数を設定できます。

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

特にUPDATE、INSERTおよびDELETE文内でFOR句を使用すると便利です。これらの文に配列全体を使用する必要がないときもあります。次の例に示すように、FOR句を使用すると、使用する要素数を任意の数に制限できます。

```
char  emp_name[100][20];
float salary[100];
int   rows_to_insert;

/* populate the host arrays */
rows_to_insert = 25;             /* set FOR-clause variable */
EXEC SQL FOR :rows_to_insert   /* will process only 25 rows */
    INSERT INTO emp (ename, sal)
    VALUES (:emp_name, :salary);
```

FOR句では、配列要素をカウントするための整数型のホスト変数や、整数リテラルも使用できます。整数値を取るC言語の複合式は使用できません。たとえば、整数式を使用する次の文は無効です。

```
EXEC SQL FOR :rows_to_insert + 5                 /* illegal */
    INSERT INTO emp (ename, empno, sal)
        VALUES (:emp_name, :emp_number, :salary);
```

FOR句の変数では、処理する配列の要素数を指定します。この数は、最小の配列次元を超えないように設定します。内部では、値は符号なしの数量として扱われます。付号付きのホスト変数を使用して負の値を渡すと、予期せぬ動作が発生します。

8.8.1 FOR句の制限

FOR句のセマンティクスを明確にするための制限が2つあります。FOR句は、SELECT文での使用またはCURRENT OF句との併用はできません。

8.8.1.1 SELECT文での使用

SELECT文中でFOR句を使用すると、エラー・メッセージが戻されます。

FOR句は意味があいまいなため、SELECT文中では使用できません。「このSELECT文を*n*回実行する」なのか、「このSELECT文を1回実行し、*n*行戻す」なのかはっきりしません。問題は、前者の場合、実行のたびに複数の行が戻される可能性があることです。後者の解釈では、次に示すように、カーソルを宣言してからFETCH文中でFOR句を使用することをお薦めします。

```
EXEC SQL FOR :limit FETCH emp_cursor INTO ...
```

8.8.1.2 CURRENT OF句との併用

次の例に示すように、UPDATEまたはDELETE文でCURRENT OF句を使用すると、FETCH文によって戻される最後の行を

参照できます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal FROM emp WHERE empno = :emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
EXEC SQL UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp_cursor;
```

ただし、CURRENT OF句とFOR句は併用できません。次の文は、*limit*の論理値が1に限定されているため無効です(つまり、現在の行を更新または削除できるのは1回のみです)。

```
EXEC SQL FOR :limit UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM emp
WHERE CURRENT OF emp_cursor;
```

8.9 WHERE句の使用方法について

Oracleでは、要素数*n*のホスト配列を含むSQL文を、同一のSQL文を*n*個の異なるスカラー変数(個々の配列要素)で*n*回実行するのと同様に扱います。このような扱いがあいまいなときにかぎり、プリコンパイラによりエラー・メッセージが発行されます。

たとえば、次のような宣言をしたとします。

```
int  mgr_number[50];
char job_title[50][20];
```

次の文の場合、あいまいになります。

```
EXEC SQL SELECT mgr INTO :mgr_number FROM emp
WHERE job = :job_title;
```

次の仮想の文のように処理されるためです。

```
for (i = 0; i < 50; i++)
    SELECT mgr INTO :mgr_number[i] FROM emp
        WHERE job = :job_title[i];
```

WHERE句の検索条件を満たす複数行があっても、データの受取りに使用できる出力変数が1つしかないためです。したがって、エラー・メッセージが出力されます。

一方、次の文を使用すると、不明瞭にならない場合があります。

```
EXEC SQL UPDATE emp SET mgr = :mgr_number
    WHERE empno IN (SELECT empno FROM emp
        WHERE job = :job_title);
```

次の仮想の文のように処理されるためです。

```
for (i = 0; i < 50; i++)
    UPDATE emp SET mgr = :mgr_number[i]
        WHERE empno IN (SELECT empno FROM emp
            WHERE job = :job_title[i]);
```

これは、それぞれの*job_title*が複数行に一致する場合でも、WHERE句内の*job_title*に一致する各行についてSET句内に

*mgr_number*が指定されているためです。それぞれの*job_title*に一致する行すべてに、同一の*mgr_number*をSETできます。したがってエラー・メッセージは発行されません。

8.10 構造体配列

スカラー配列を使用すると、1つの列での複数行操作が可能です。また、スカラー構造体を使用すると、1つの行での複数列操作が可能です。

しかし、複数列で複数行を操作する場合、これまでは複数のスカラー並列配列を個別にまたは1つの構造体中でカプセル化して割り当てる必要がありました。この方法よりも、このデータ構造を複数の構造体からなる1つの配列にしなおす方が便利です。

Pro*C/C++では構造体配列がサポートされています。アプリケーション・プログラマはC構造体の配列を使用して複数行および複数列の操作を実行できます。この機能拡張により、Pro*C/C++ではスカラー構造体の単純な配列を埋込みSQL文でバインド変数として処理できるため、データの処理がより簡単になります。これで、プログラミングがさらに直観的になり、データ編成もはるかに自由にできます。

Pro*C/C++では、構造体配列がバインド変数としてサポートされるのみでなく、インジケータ構造体の配列を構造体配列の宣言と併用できます。

> 注意:
>
> 構造体を PL/SQL レコードにバインドすることと、構造体の配列を PL/SQL レコードの表にバインドすることは、この新たな機能の一部ではありません。また、構造体の配列を埋込み PL/SQL ブロック内で使用することもできません。

構造体の配列は、複数の列で複数の行を操作するために使用され、通常次のように使用されると考えてください。

- SELECT文またはFETCH文での出力バインド変数として使用
- INSERT文のVALUES句での入力バインド変数として使用

関連項目

- 構造体の配列の制限

8.10.1 構造体の配列の使用方法

構造体の配列という概念は、C言語のプログラマにとって特に目新しいものではありません。しかし、複数の並列配列からなる1つの構造体と比較してみると、データの格納方法に考え方の違いがあります。

複数の並列配列からなる1つの構造体では、個々の列のデータが連続して格納されます。一方、構造体の配列では、列のデータはインタリーブされます。この場合、配列内の各列の間は、その構造体内の他の列に必要な空白で区切られます。この空白は**ストライド**と呼ばれます。

8.10.2 構造体の配列の制限

Pro*C/C++では、構造体配列の使用に次の制限事項があります。

- 構造体配列(通常の構造体による)を埋込みPL/SQLブロック内で使用できません。
- 構造体配列をWHERE句やFROM句で使用できません。

- 構造体配列はOracle動的SQL方法4では使用できません。ANSI動的SQLでは使用できます。
- 構造体配列をUPDATE文のSET句で使用できません。

構造体配列の宣言では構文に違いはありません。ただし、構造体配列を使用する場合には留意事項がいくつかあります。

関連項目

- ANSI動的SQL

8.10.3 構造体の配列の宣言について

Pro*C/C++アプリケーションで使用する構造体配列を宣言する場合、プログラマは必ず次の点に留意してください。

- 構造体には必ず構造体タグを付けてください。例は次のとおりです。

```
    struct person {
       char name[15];
       int  age;
    } people[10];
```

このコード・セグメントでは、person変数が構造体のタグです。このタグによって、プリコンパイラは構造体の名前を使用してストライドのサイズを計算できます。

- 構造体のメンバーは配列にしないでください。ただし、**char**や**VARCHAR**などの文字型の場合には、この規則は適用されません。このような型の変数の宣言で配列の構文が使用されるためです。
- **char**および**VARCHAR**型のメンバーは2次元配列にしないでください。
- ネストされた構造体は構造体配列のメンバーになることはできません。Pro*C/C++の旧リリースでは、ネストされた構造体はサポートされていなかったため、この制限は新しいものではありません。
- 構造体自体のサイズは、符号付き4バイト数で表すことのできる最大値を超えないようにしてください。通常、この最大値は2GBです。

構造体配列の使用に関する前述の制限事項を満たしているので、Pro*C/C++では次の宣言は有効です。

```
struct department {
   int deptno;
   char dname[15];
   char loc[14];
} dept[4];
```

一方、次の宣言は無効です。

```
struct {              /* the struct is missing a structure tag */
  int empno[15];      /* struct members may not be arrays */
  char ename[15][10]; /* character types may not be 2-dimensional */
  struct nested {
    int salary;   /* nested struct not permitted in array of structs */
  } sal_struct;
} bad[15];
```

データ型の同値化を、構造体配列自体や構造体内の個々のフィールドには適用できないことにも注意してください。たとえば、empnoが前述の無効な構造体内の配列として宣言されていない場合、次の文は無効です。

```
exec sql var bad[3].empno is integer(4);
```

プリコンパイラには、構造体配列中の個々の構造体要素を追跡し記録する機能はありません。ただし、次を実行すれば、期待した結果が得られます。

```
typedef int myint;
exec sql type myint is integer(4);

struct equiv {
  myint empno; /* now legally considered an integer(4) datatype */
   ...
} ok[15];
```

Pro*C/C++の以前のリリースでは、個別の配列項目の同値化がサポートされていなかったため、これは予測できたことといえます。たとえば、次のスカラー配列宣言は何が有効で、何が有効でないかを示しています。

```
int empno[15];
exec sql var empno[3] is integer(4); /* illegal */

myint empno[15]; /* legal */
```

基本的には、個々の配列項目を同値化することはできません。

8.10.4 変数のガイドライン

標識変数は、構造体の配列の宣言でも、通常の構造体の宣言の場合とほとんど同じはたらきをします。構造体のインジケータ配列の宣言は、構造体の配列についての次の規則にも従う必要があります。

- インジケータ構造体に含まれるフィールド数は、対応する構造体の配列に含まれるフィールド数以下である必要があります。
- フィールドの順序は、対応する構造体の配列のメンバーの順序に一致する必要があります。
- インジケータ構造体に含まれるすべての要素のデータ型は、**short**である必要があります。
- インジケータ配列のサイズは、ホスト変数で宣言されたサイズ以上である必要があります。ホスト変数で宣言されたサイズより大きい値は指定できますが、それより小さい値は指定できません。

これらの規則のほとんどには、Pro*C/C++の以前のリリースでの構造体使用規則が反映されています。配列制限も以前使用されたスカラーの配列に対するものと同じです。

これらの規則が適用されている場合に、次のように構造体を宣言するとします。

```
struct department {
   int deptno;
   char dname[15];
   char loc[14];
} dept[4];
```

次の標識変数の構造体の宣言は有効です。

```
struct department_ind {
   short deptno_ind;
   short dname_ind;
   short loc_ind;
} dept_ind[4];
```

一方、次は標識変数として無効です。

```
struct{               /* missing indicator structure tag */
  int deptno_ind;     /* indicator variable not of type short */
  short dname_ind[15];/* array element forbidden in indicator struct */
  short loc_ind[14];  /* array element forbidden in indicator struct */
} bad_ind[2];     /* indicator array size is smaller than host array */
```

関連項目

- ANSI動的SQL

8.10.5 構造体配列へのポインタの宣言について

構造体配列へのポインタを宣言する方が適切な場合があります。これにより、構造体配列へのポインタを他の関数に渡したり、埋込みSQL文で直接指定したりできます。

> 注意:
>
> 
>
> 構造体配列へのポインタが参照する配列の長さは、プリコンパイル中にはわかりません。このため、埋込み SQL 文で構造体の配列へのポインタ型になっているバインド変数を使用するときには、明示的な FOR 句を使用してください。

ただし、FOR句は埋込みSQL SELECT文では指定できません。したがって、データを取り出して構造体配列へのポインタに入れる場合には、必ずカーソルとFETCH文をFOR句とともに明示的に指定してください。

8.10.6 例

次の例は、Pro*C/C++での構造体配列の機能について、様々な使用方法を示しています。

8.10.6.1 例1: スカラー構造体の単純な配列

次の構造体の宣言を指定したとします。

```
struct department {
   int deptno;
   char dname[15];
   char loc[14];
} my_dept[4];
```

次のようにdeptデータを選択してmy_deptに入れることができます。

```
exec sql select * into :my_dept from dept;
```

また、最初にmy_deptにデータを入れてから、dept表に一括して挿入できます。

```
exec sql insert into dept values (:my_dept);
```

標識変数を指定するには、構造体のパラレル・インジケータ配列を宣言します。

```
struct deptartment_ind {
   short deptno_ind;
   short dname_ind;
   short loc_ind;
} my_dept_ind[4];
```

データの選択に使用される問合せは、標識変数の指定が追加されたこと以外は同じです。

```
exec sql select * into :my_dept indicator :my_dept_ind from dept;
```

同様に、データの挿入時にもインジケータを指定できます。

```
exec sql insert into dept values (:my_dept indicator :my_dept_ind);
```

8.10.6.2 例2: スカラーの配列と構造体の配列との組合せ使用

Pro*C/C++の旧リリースと同様に、ユーザー・データのバルク処理機能に複数の配列を使用する場合は、各配列のサイズを同じにする必要があります。同じにしないと、最も小さい配列のサイズが選択され、残りの配列は変更されません。

次の宣言を指定したとします。

```
struct employee {
   int empno;
   char ename[11];
} emp[14];

float sal[14];
float comm[14];
```

ただ1つの問合せで、すべての列に対して複数行を選択できます。

```
exec sql select empno, ename, sal, comm into :emp, :sal, :comm from emp;
```

また、コミッション列の値がNULLかどうかを確認する必要があります。次のように宣言すれば、1つのインジケータ配列を指定するのみで済みます。

```
short comm_ind[14];
...
exec sql select empno, ename, sal, comm
   into :emp, :sal, :comm indicator :comm_ind from emp;
```

問合せからのインジケータ情報をすべてカプセル化した構造体の単一インジケータ配列は宣言できません。次の例を考えます。

```
struct employee_ind {   /* example of illegal usage */
   short empno_ind;
   short ename_ind;
   short sal_ind;
   short comm_ind;
} illegal_ind[15];

exec sql select empno, ename, sal, comm
   into :emp, :sal, :comm indicator :illegal_ind from emp;
```

この列は無効です(またお薦めできません)。上の文にはSELECT...INTOリスト全体ではなく、comm列しかないインジケータ配列が対応付けられています。

構造体の配列とsal、commおよびcomm_ind配列にデータを挿入する場合、その挿入方法は非常に簡単です。

```
exec sql insert into emp (empno, ename, sal, comm)
   values (:emp, :sal, :comm indicator :comm_ind);
```

8.10.6.3 例3: 複数の構造体配列と1つのカーソルとの組合せ使用

次の宣言をこの例として使用します。

```
struct employee {
   int empno;
   char ename[11];
   char job[10];
} emp[14];

struct compensation {
```

```
   int sal;
   int comm;
} wage[14];

struct compensation_ind {
   short sal_ind;
   short comm_ind;
} wage_ind[14];
```

Oracleのプログラムでは、構造体配列を次のように指定できます。

```
exec sql declare c cursor for
   select empno, ename, job, sal, comm from emp;

exec sql open c;

exec sql whenever not found do break;
while(1)
{
  exec sql fetch c into :emp, :wage indicator :wage_ind;
  ... process batch rows returned by the fetch ...
}

printf("%d rows selected.¥n", sqlca.sqlerrd[2]);

exec sql close c;
```

8.10.6.3.1 FOR句の使用方法について

FOR句を使用しても、FETCHで取り出す行数を指定できます。FOR句は、SELECT文を使用している場合には指定できませんが、INSERT文やFETCH文の場合には指定できます。

元の宣言に次を追加します。

```
int limit = 10;
```

次は、それに対応したコードの例です。

```
   exec sql for :limit
      fetch c into :emp, :wage indicator :wage_ind;
```

8.10.6.4 例4: 個々の配列メンバーおよび構造体メンバーの参照

これまでのPro*C/C++リリースで、構造体配列内の1構造体に対する配列参照が可能となっています。これによりバインド式はスカラーの単純な構造体で解決できるため、次は有効です。

```
exec sql select * into :dept[3] from emp;
```

次の例に示すように、構造体配列内の特定の構造体のスカラー・メンバーを個別に参照することも可能です。

```
exec sql select dname into :dept[3].dname from dept where ...;
```

この場合には当然、問合せは単一行問合せにする必要があり、1行のみを選択してこのバインド式で表される変数に代入します。

8.10.6.5 例5: 標識変数の使用(特殊な場合)

これまでのPro*C/C++リリースでは、インジケータ構造体のフィールド数はそれに対応するバインド変数構造体と同じにする必要がありました。構造体を通常に使用する場合には、この制限は緩和されています。前述の構造体のインジケータ配列につい

てのガイドラインに従っているので、次の例が可能です。

```
struct employee {
    float comm;
    float sal;
    int empno;
    char ename[10];
} emp[14];

struct employee_ind {
    short comm;
} emp_ind[14];

exec sql select comm, sal, empno, ename
   into :emp indicator :emp_ind from emp;
```

標識変数はバインド値と1対1でマップされます。これらは、最初のフィールドから、対応した順番にマップされます。

ただし、他のフィールドでフェッチされた値がNULLであったり、インジケータが付いていなかったりすると、次のエラーが発生するため注意してください。

```
ORA-1405: fetched column value is NULL
```

たとえば、salがNULL値可能の場合にsalにはインジケータがないため、このエラーが発生します。

次のように構造体の配列を変更したとします。

```
struct employee {
   int empno;
   char ename[10];
   float sal;
   float comm;
} emp[15];
```

しかし、同じ構造体のインジケータ配列がまだ使用されているとします。

インジケータのマップは対応した順番に実行されるため、commインジケータがempnoフィールドにマップされて、commバインド変数にインジケータがないままとなり、再度ORA-1405エラーとなります。

対応するバインド変数の構造体よりもフィールド数の少ないインジケータ構造体を指定したときに、ORA-01405エラーが発生しないようにするには、NULL値可能の属性を最初から順番に並べる必要があります。

この例は、非配列の構造体を使用すれば複数列の単一行フェッチに簡単に変更でき、インジケータ構造体が次のように宣言されている場合と同等のはたらきをすると考えることができます。

```
struct employee_ind {
   short comm;
   short sal;
   short empno;
   short ename;
} emp_ind;
```

Pro*C/C++ではインジケータ構造体のフィールド数と対応する値の構造体のフィールド数が同じである必要がなくなったため、前述の例はこれまでPro*C/C++で無効でしたが現在は有効になりました。

Oracleのインジケータ構造体は、次のように簡単に指定できます。

```
struct employee_ind {
   short comm;
} emp_ind;
```

次のように配列なし構造体であるempおよびemp_indを使用すると、単一行フェッチを実行できます。

```
exec sql fetch comm, sal, empno, ename
   into :emp indicator :emp_ind from emp;
```

この場合にも、commインジケータがcommバインド変数にどのようにマップされるかに注意してください。

8.10.6.6 例6: 構造体配列へのポインタの使用

この例では、構造体配列へのポインタの使用方法を示します。

次の型の宣言を考えます。

```
typedef struct dept {
   int deptno;
   char dname[15];
   char loc[14];
} dept;
```

その型の構造体配列へのポインタを操作すると、様々な処理を実行できます。たとえば、構造体配列へのポインタを他の関数に渡すことができます。

```
void insert_data(d, n)
   dept *d;
   int n;
{
    exec sql for :n insert into dept values (:d);
}

void fetch_data(d, n)
   dept *d;
   int n;
{
   exec sql declare c cursor for select deptno, dname, loc from dept;
   exec sql open c;
   exec sql for :n fetch c into :d;
   exec sql close c;
}
```

このような関数をコールするには、例に示すように構造体配列のアドレスを渡します。

```
dept d[4];
dept *dptr = &d[0];
const int n = 4;

fetch_data(dptr, n);
insert_data(d, n); /* We are treating '&d[0]' as being equal to 'd' */
```

一部の埋込みSQL文では、構造体配列へのポインタを直接使用することができます。

```
exec sql for :n insert into dept values (:dptr);
```

FOR句の使用方法を間違えないように、十分に注意してください。

8.11 CURRENT OFの疑似実行について

DELETE文またはUPDATE文でCURRENT OF *cursor*句を使用すると、カーソルから最後にフェッチされた行を参照できます。ただし、CURRENT OF句とホスト配列の併用はできません。かわりに各行のROWIDを取得し、更新または削除するときにその値を使用して現在行を識別してください。

次に例を示します。

```
char  emp_name[20][10];
char  job_title[20][10];
char  old_title[20][10];
char  row_id[20][19];
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, job, rowid FROM emp FOR UPDATE;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_name, :job_title, :row_id;
    ...
    EXEC SQL DELETE FROM emp
        WHERE job = :old_title AND rowid = :row_id;
    EXEC SQL COMMIT WORK;
}
```

関連項目

- CURRENT OF句

8.12 追加の配列の挿入/選択構文の使用について

Oracleプリコンパイラでは、ホスト表に対するDB2のINSERTおよびFETCH構文もサポートされています。サポートされている追加の配列の挿入およびフェッチ構文は、次の図にそれぞれ示しています。

図8-1 追加の挿入構文

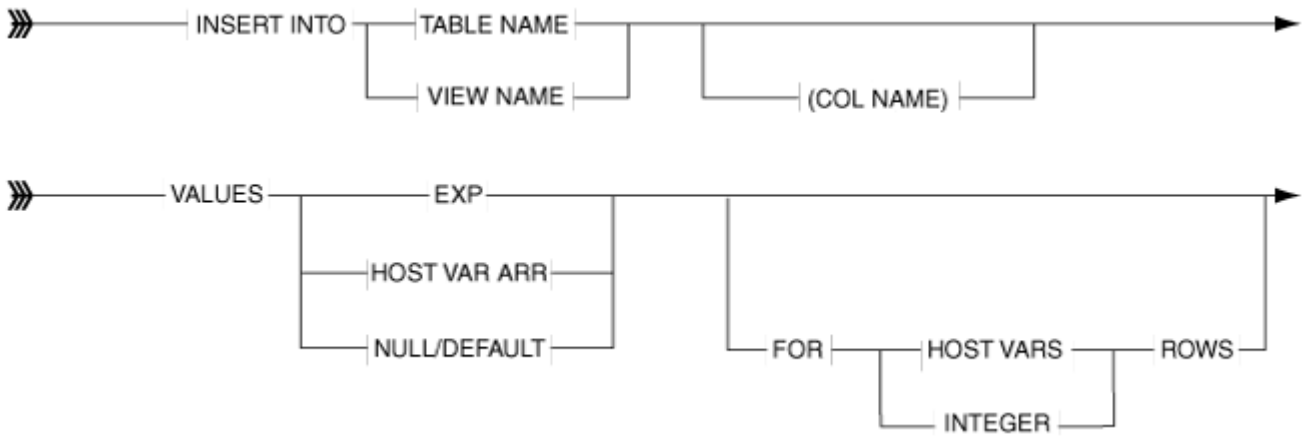
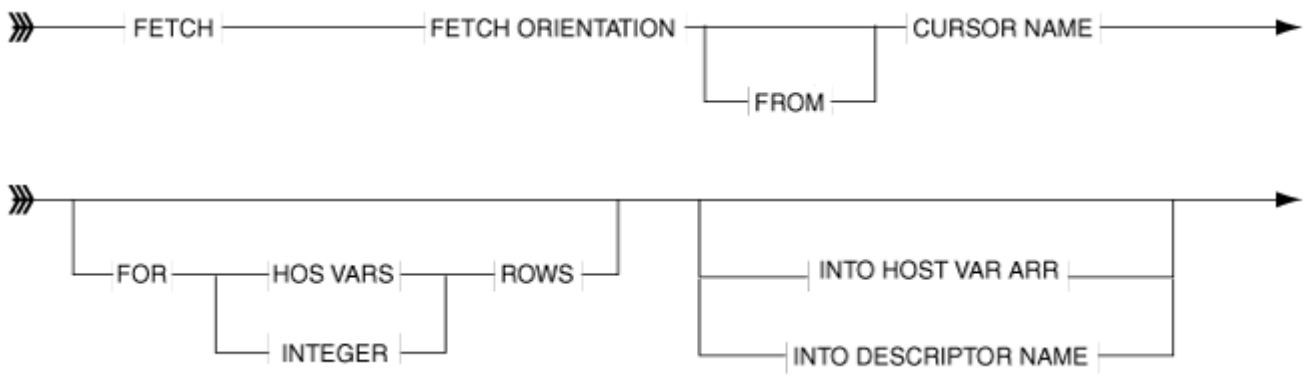


図8-2 追加のフェッチ構文



オプションでROWSET句とROWSET STARTING AT句がフェッチの方向(FIRST, PRIOR, NEXT, LAST, CURRENT, RELATIVEおよびABSOLUTE)で使用されます。次の例を参考にしてください。

- FIRST ROWSET
- PRIOR ROWSET
- NEXT ROWSET
- LAST ROWSET
- CURRENT ROWSET
- ROWSET STARTING AT RELATIVEn
- ROWSET STARTING AT ABSOLUTEn

DB2配列の挿入/フェッチ構文の例と、対応するOracleプリコンパイラ構文との比較を、表8-3に示します。

表8-3 DB2配列構文とOracleプリコンパイラ構文の比較

DB2配列構文	Oracleプリコンパイラの構文
EXEC SQL INSERT INTO dsn8810.act (actno, actkwd, actdesc) VALUES (:hva1, :hva2, :hva3) FOR :NUM_ROWS ROWS;	EXEC SQL FOR :num_rows INSERT INTO dsn8810.act (actno, actkwd, actdesc) VALUES (:hva1, :hva2, :hva3);
EXEC SQL FETCH NEXT ROWSET FROM c1 FOR 20 ROWS INTO :hva_empno, :hva_lastname, :hva_salary;	EXEC SQL FOR :twenty FETCH c1 INTO :hva_empno, :hva_lastname, :hva_salary;

DB2の構文では、行セットに位置付けられたカーソルは、データの行セットを取得する前に最初に宣言する必要があります。カーソルが行セットをフェッチできるようにするには、DECLARE CURSOR文で「WITH ROWSET POSITIONING」句を使用する必要があります。これは、次の表に示すように、Oracleプリコンパイラの構文では必要なく、関連もありません。

DB2配列構文	Oracleプリコンパイラの構文
EXEC SQL DECLARE c1 CURSOR WITH ROWSET POSITIONING FOR SELECT empno, lastname, salary FROM dsn8810.emp;	EXEC SQL DECLARE c1 CURSOR FOR SELECT empno, lastname, salary FROM dsn8810.emp;

このDB2配列構文のサポートは、プリコンパイラ・オプションdb2_arrayを指定して有効にできます。デフォルトのオプションはnoです。DB2配列構文のサポートは、Oracleプリコンパイラ構文と一緒に使用できません。一度にサポートされるのは、Oracleプリコンパイラ構文またはDB2構文のいずれか一方の構文のみです。

例8-1 DB2配列構文使用による行の挿入およびフェッチ

このプログラムでは、DB2の配列INSERT構文を使用して、EMP表にINSCNT行を挿入します。その後、DB2の配列FETCH構文を使用して、挿入された行をフェッチします。

```
/*
 * db2arrdemo.pc
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>
```

```
/* Number of rows to be inserted in one shot */
#define INSCNT 100
/* Number of rows to be fetched in one shot */
#define FETCHCNT 20

/* Define a host structure
   for inserting data into the table
   and for fetching data from the table */
struct emprec
{
  int     empno;
  varchar ename[10];
  varchar job[9];
  int mgr;
  char hiredate[10];
  float sal;
  float comm;
  int deptno;
};
typedef struct emprec empdata;

/* Function prototypes */
void sql_error(char *);
void insertdata();
void fetchdata();
void printempdata(empdata *);

void main()
{

  exec sql begin declare section;
    char *uid = "scott/tiger";
  exec sql end declare section;

  exec sql whenever sqlerror do sql_error("ORACLE error--¥n");
  exec sql connect :uid;

  printf("Inserting %d rows into EMP table using DB2 array insert syntax.¥n",
         INSCNT);
  insertdata();
  printf("¥nFetching data using DB2 array fetch syntax.¥n");
  fetchdata();

  exec sql rollback work release;
  exit(EXIT_SUCCESS);
}

/* Inserting data into the table using DB2 array insert syntax*/
void insertdata()
{
  int i, cnt;
  char *str;
  empdata emp_in[INSCNT];

  /* To store temporary strings */
  str = (char *)malloc (25 * sizeof(char));

  /* Fill the array elements to insert */
  for (i = 0; i < INSCNT; i++)
  {
```

```
    emp_in[i].empno = i+1;
    sprintf(str, "EMP_%03d", i+1);
    strcpy (emp_in[i].ename.arr, str);
    emp_in[i].ename.len = strlen (emp_in[i].ename.arr);
    sprintf(str, "JOB_%03d", i+1);
    strcpy (emp_in[i].job.arr, str);
    emp_in[i].job.len = strlen (emp_in[i].job.arr);
    emp_in[i].mgr = i+1001;
    sprintf(str, "%02d-MAY-06", (i%30)+1);
    strcpy (emp_in[i].hiredate, str);
    emp_in[i].sal = (i+1) * 10;
    emp_in[i].comm = (i+1) * 0.1;
    emp_in[i].deptno = 10;
  }

  free (str);

  /* Inserting data using DB2 array insert syntax */
  exec sql insert into emp values (:emp_in) FOR :INSCNT rows;

  exec sql select count(*) into :cnt from emp where ename like 'EMP_%';
  printf ("Number of rows successfully inserted into emp table: %d¥n", cnt);
}

/* Fetches data from the table using DB2 array fetch syntax*/
void fetchdata()
{
  empdata emp_out[FETCHCNT];

  /* Declares scrollable cursor to fetch data */
  exec sql declare c1 scroll cursor with rowset positioning for
         select empno, ename, job, mgr, hiredate, sal, comm, deptno
         from emp where ename like 'EMP_%' order by empno;

  exec sql open c1;

  exec sql whenever not found do break;
  while(1)
  {
    /* Fetches data using DB2 array fetch syntax */
    exec sql fetch next rowset from c1 for :FETCHCNT rows into :emp_out;
    printempdata(emp_out);
  }
  exec sql whenever not found do sql_error("ORACLE ERROR");

  exec sql close c1;
}

/* Prints the fetched employee data */
void printempdata(empdata *emp_out)
{
  int i;
  for (i=0; i<FETCHCNT; i++)
  {
    emp_out[i].ename.arr[emp_out[i].ename.len] = '¥0';
    emp_out[i].job.arr[emp_out[i].job.len] = '¥0';
    printf("Empno=%d, Ename=%s, Job=%s, Mgr=%d, Hiredate=%s, Sal=%6.2f,¥n"
           "Comm=%5.2f, Deptno=%d¥n", emp_out[i].empno, emp_out[i].ename.arr,
 emp_out[i].job.arr, emp_out[i].mgr, emp_out[i].hiredate,
 emp_out[i].sal, emp_out[i].comm, emp_out[i].deptno);
```

```
  }
}

/* Error handling function. */
void sql_error(char *msg)
{
  exec sql whenever sqlerror continue;

  printf("¥n%s¥n", msg);
  printf("%.70s¥n", sqlca.sqlerrm.sqlerrmc);
  exec sql rollback release;

  exit(EXIT_FAILURE);
}
```

8.13 暗黙的なバッファ済INSERTの使用について

パフォーマンスを高めるために、Pro*C/C++アプリケーション開発者は、埋め込んだSQL文のホスト配列を参照できます。これにより、データベースへの1回のラウンドトリップでSQL文の配列を実行できます。配列の実行によってパフォーマンスが大幅に向上するにもかかわらず、ANSI規格ではないため、この機能を使用しない開発者もいます。たとえば、Oracle製品で配列の実行を使用するように記述されたアプリケーションは、IBMのプリコンパイラを使用してプリコンパイルすることはできません。

対処方法として、バッファ済INSERT文を使用すると、ANSI規格の埋込みSQL構文を保持しながらパフォーマンスを向上させることができます。

コマンドライン・オプション「max_row_insert」は、INSERT文の実行前にバッファする行の数を制御します。このオプションのデフォルトは0で、機能は無効化されています。この機能を有効化するには、0よりも大きい任意の数を指定します。

挿入バッファを有効化すると、プリコンパイラのランタイムが対応するカーソルにフラグを付け、次を実行します。

- バインド変数を保持するための追加のメモリーを割当てまたは再割当てします(初回実行時のみ)。
- プログラムのホスト変数から内部ランタイム・バインド構造にバインド変数をコピーします。
- バッファ済の行カウントを増加させます。
- MAX_INSERT_ROWSがバッファされている場合、バッファされたINSERT文をフラッシュします。
- MAX_INSERT_ROWSに達していない場合は、フラッシュせずに、内部バインド・バッファに値をコピーした後、戻ります。

新しい埋込みSQL文が実行され、バッファ挿入文がフラッシュされる場合、次のことが実行されます。

- バッファをフラッシュします。
- フラッシュを求めたコールを続行します。

アプリケーションには、標準のプリコンパイラ・エラー・メカニズム(Pro*Cのsqlcaなど)を介して、エラーが知らされます。

「implicit_svpt」オプションは、新しくバッチ処理された挿入を開始する前に、暗黙的なセーブポイントを設定するかどうかを制御します。

- YESの場合は、新しい行のバッチを開始する前に、セーブポイントが設定されます。挿入時にエラーが発生すると、暗黙的な「セーブポイントへのロールバック」が実行されます。
- NOの場合は、暗黙的なセーブポイントが設定されません。バッファ済INSERTでエラーが発生すると、アプリケーションに通知されますが、ロールバックは実行されません。バッファ済INSERTのエラーは非同期に報告されます。アプリケーションでINSERT文が実行されると、挿入された行のエラーは報告されません。

- 挿入された行の一部のエラーは、INSERT以外の文が初めて実行されたときに後で報告されます。これには、DELETE、UPDATE、(別の表に対する)INSERT、COMMITおよびROLLBACKが含まれます。バッファ済INSERT文をクローズする文は、すべてエラーを報告します。この場合、エラーを報告する文は実行されません。エラーを処理した後で、バッファ済INSERTのエラーを報告した文を再実行する必要があります。それ以外の場合は、トランザクションをロールバックして再実行します。

  たとえば、COMMIT文を使用して、バッファ済INSERTのループをクローズする方法について考えてみます。COMMITでは、以前のINSERTの重複キーが原因でエラーが発生します。この場合、COMMITは実行されません。エラーを処理した後で、COMMITを再実行する必要があります。それ以外の場合は、トランザクションをロールバックして再実行します。

- 挿入自体についてもいくつかエラーが報告され、以前に挿入された行のエラーが反映されている可能性があります。そのような場合、それ以上挿入は実行されません。以前挿入された行のエラーを処理し、現行の挿入を継続する必要がありますが、それには時間がかかります。かわりに、トランザクションをロールバックし、再実行してもかまいません。

  たとえば、内部バッファの制限が10行で、アプリケーションがループで15行を挿入しているとします。8行目でエラーが発生したとします。エラーは、11行目が挿入されたときに報告され、これ以後にINSERTは実行されません。

バッファ済INSERT中に発生する可能性のあるエラーの一部を次に示します。

- ORA-00001: 索引キーが重複しています
- ORA-01400: 挿入時に必須列(NOT NULL)がないか、NULLになっています
- ORA-01401: 列に挿入した値が大きすぎます。
- ORA-01438: 指定した精度を超えた値が列に指定されています

例8-2 表へのバッファ行の挿入

このプログラムでは、EMP表に行のLOOPCNT数を挿入します。loop counter=5の場合、このプログラムは無効なempnoの挿入を試行します。max_row_insertオプションを使用しないと、プログラムでは無効な行を除くすべての行が挿入されます。max_row_insertオプションをLOOPCNTに設定すると、最初の4行のみが挿入されます。

max_row_insertオプションを使用すると、間違った文が削除されるときに、プログラムは配列INSERTプログラムと同様に動作します。

```
/*
 * bufinsdemo.pc
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>

/* Number of rows to be inserted into the table */
#define LOOPCNT 100

/* Define a host structure
   for inserting data into the table
   and for fetching data from the table */
struct emprec
{
```

```
  int empno;
  varchar ename[10];
  varchar job[9];
  int mgr;
  char hiredate[10];
  float sal;
  float comm;
  int deptno;
};
typedef struct emprec buffinstyp;

/* Function prototypes */
void sql_error();
void insertdata();
void fetchdata();
void printempdata(buffinstyp);

void main()
{

  exec sql begin declare section;
    char *uid = "scott/tiger";
  exec sql end declare section;

  exec sql whenever sqlerror do sql_error();
  exec sql connect :uid;

  printf("¥nInserting %d rows into EMP table.¥n", LOOPCNT);
  insertdata();
  printf("¥nFetching inserted data from EMP table.¥n");
  fetchdata();

  exec sql delete from emp where empno < 1000;

  exec sql commit work release;
  exit(EXIT_SUCCESS);
}

/* Inserting data into the table */
void insertdata()
{
  int i, cnt;
  char *str;
  buffinstyp emp_in;

  /* To store temporary strings */
  str = (char *)malloc (25 * sizeof(char));

  /*
   * When max_row_insert option is set to LOOPCNT and when the errorneous
   * statement is removed, all the rows will be inserted into the database in
   * one stretch and hence maximum performance gain will be achieved.
 */
  for (i = 1; i <= LOOPCNT; i++)
  {
    if (i != 5)
      emp_in.empno = i;
    else
    /* Errorneous statement. In emp table, empno is defined as number(4). */
      emp_in.empno = 10000;
```

```
    sprintf(str, "EMP_%03d", i);
    strcpy (emp_in.ename.arr, str);
    emp_in.ename.len = strlen (emp_in.ename.arr);
    sprintf(str, "JOB_%03d", i);
    strcpy (emp_in.job.arr, str);
    emp_in.job.len = strlen (emp_in.job.arr);
    emp_in.mgr = i+1001;
    sprintf(str, "%02d-MAY-06", (i%30));
    strcpy (emp_in.hiredate, str);
    emp_in.sal = (i) * 10;
    emp_in.comm = (i) * 0.1;
    emp_in.deptno = 10;

    exec sql insert into emp values (:emp_in);
  }

  free (str);

  exec sql commit;

  exec sql select count(*) into :cnt from emp where ename like 'EMP_%';
  printf ("Number of rows successfully inserted into emp table: %d¥n", cnt);
}

/* Fetches data from the table*/
void fetchdata()
{
  buffinstyp emp_out;

  /* Declares cursor to fetch only the rows that are inserted */
  exec sql declare c1 cursor for
         select empno, ename, job, mgr, hiredate, sal, comm, deptno
         from emp where ename like 'EMP_%' order by empno;

  exec sql open c1;

  exec sql whenever not found do break;
  while(1)
  {
    /* Fetches single row at each call */
    exec sql fetch c1 into :emp_out;
    printempdata(emp_out);
  }
  exec sql whenever not found do sql_error();

  exec sql close c1;
}

/* Prints the fetched employee data */
void printempdata(buffinstyp emp_out)
{
  emp_out.ename.arr[emp_out.ename.len] = '¥0';
  emp_out.job.arr[emp_out.job.len] = '¥0';
  printf("Empno=%d, Ename=%s, Job=%s, Mgr=%d, Hiredate=%s, Sal=%6.2f,¥n"
         "Comm=%5.2f, Deptno=%d¥n", emp_out.empno, emp_out.ename.arr,
 emp_out.job.arr, emp_out.mgr, emp_out.hiredate, emp_out.sal,
 emp_out.comm, emp_out.deptno);
}
```

```
/* Error handling function. */
void sql_error()
{
   printf("Error %s¥n", sqlca.sqlerrm.sqlerrmc);
   printf(" Rows Processed: %d¥n", sqlca.sqlerrd[2]);
   printf(" Rows Rolled Back: %d¥n", sqlca.sqlerrd[0]);
}
```

8.14 スクロール可能カーソル

スクロール可能カーソルは、SQL文が実行され、実行中に処理された情報が格納される作業領域です。カーソルを実行すると、問合せの結果は結果セットと呼ばれる一連の行に配置されます。結果セットは、順番にフェッチすることも、順不同でフェッチすることもできます。順不同の結果セットをスクロール可能カーソルと呼びます。スクロール可能カーソルを使用すると、ユーザーは前から、後ろからまたはランダムな方法でデータベース結果セットの行にアクセスできます。これにより、プログラムは結果セットの任意の行をフェッチできます。

9 ランタイム・エラーの処理

アプリケーション・プログラムでは、ランタイム・エラーを予測して、そのエラーをリカバリするように対処する必要があります。この章では、エラー・レポートおよびリカバリを詳しく説明します。SQL通信領域(SQLCA)、WHENEVERディレクティブおよびSQLSTATE状態変数を使用して、エラーおよび状態の変化を処理する方法を説明します。さらにOracle通信領域(ORACA)による問題点の診断方法も紹介します。この章のトピックは、次のとおりです:

- エラー処理の必要性
- エラー処理の代替手段
- SQLSTATE状態変数
- SQLCODEの宣言について
- SQLCAを使用したエラー・レポートの主要コンポーネント
- SQL通信領域(SQLCA)の使用
- エラー・メッセージの全文の取得について
- WHENEVERディレクティブの使用について
- SQL文のテキスト取得について
- Oracle通信領域(ORACA)の使用について

9.1 エラー処理の必要性

どのようなアプリケーション・プログラムでも、その大部分をエラー処理に当てる必要があります。エラー処理の主な目的は、エラーが発生してもプログラムの処理を続行できるようにすることです。エラーは設計ミス、コーディングの誤り、ハードウェア障害、誤ったユーザー入力をはじめ、様々な原因で発生します。

潜在的なエラーをすべて予測するのは無理ですが、プログラムにとって意味のある特定の種類のエラーについてアクションを考えることはできます。Pro*C/C++プリコンパイラにとっては、エラー処理とはSQL文の実行エラーの検出およびリカバリのことです。値が切り捨てられたことを示す警告やデータの終わりなどの状態の変更も処理できます。INSERT文、UPDATE文またはDELETE文では、表内の処理対象行すべてを処理する前に失敗することがあるため、SQLデータ操作文を実行するたびにエラー状態および警告状態がないか調べることが特に重要です。

9.2 エラー処理の代替手段

アプリケーションの状態の変化およびエラーを検出するにはいくつかの手段があります。この章ではそれらの手段について説明しますが、特定の手段の推奨は行っていません。最終的にどの手段を使用するかは、作成中のアプリケーション・プログラムまたはツールがどのように設計されているかによって決まります。

9.2.1 状態変数

状態変数SQLSTATEまたはSQLCODEを別に宣言し、実行SQL文の後の値をそれぞれ調べて、適切なアクションを取ることができます。アクションとして、エラー報告関数をコールし、エラーがリカバリ不能なときはプログラムを終了することができます。または、データを調整するか変数を制御して、アクションを再試行することもできます。

関連項目

- SQLSTATE状態変数

- SQLCODEの宣言について

9.2.2 SQL通信領域

もう1つの手段は、プログラムにSQL通信領域構造体(*sqlca*)を組み込むことです。実行時にSQL文がOracleによって処理されると、この構造体のコンポーネントに値が格納されます。

> 注意:
>
> このマニュアルでは、*sqlca* 構造体は一般に *SQL* 通信領域の頭字語(SQLCA)を使用して表記します。このマニュアルで C 言語の**構造体**の特定のコンポーネントに言及するときには、構造体の名称(*sqlca*)を使用します。

SQLCAはヘッダー・ファイルsqlca.hに定義されています。次の文のどちらかを使用してSQLCAをプログラムにインクルードします。

- EXEC SQL INCLUDE SQLCA;
- #include <sqlca.h>

Oracleでは、すべての実行SQL文の後でSQLCAが更新されます。(SQLCAの値は宣言文の後では変更されません。)SQLCAに格納されているOracleリターン・コードをチェックすることで、プログラムはSQL文の結果を判断できます。この判断には次の2通りの方法があります。

- WHENEVERディレクティブによる暗黙的なチェック
- SQLCAコンポーネントの明示的なチェック

WHENEVERディレクティブを使用するか、SQLCAコンポーネントの明示的なチェックを使用するか、またはその両方を同時に使用することが可能です。

SQLCAで最も頻繁に使用されるコンポーネントは、状態変数(*sqlca.sqlcode*)およびエラー・コード(*sqlca.sqlerrm.sqlerrmc*)に関連するテキストです。他のコンポーネントには、警告フラグおよびSQL文の処理に関する各種の情報が格納されます。

> 注意:
>
> SQLCODE(大文字)は常に個別の状態変数のことを指し、SQLCA のコンポーネントのことではありません。SQLCODE は、**整数**として宣言されます。SQLCA のコンポーネント *sqlcode* のことを指す場合は、常に完全修飾名 *sqlca.sqlcode* が使用されます。

ランタイム・エラーについて、SQLCAに格納されている情報よりも詳細な情報が必要なときにORACAを使用します。ORACAは、Oracleの通信を処理するC言語の**構造体**です。この中にはカーソル統計情報、現行のSQL文に関する情報、オプションの設定、およびシステムの統計情報が含まれます。

関連項目

- Oracle通信領域(ORACA)の使用について
- SQL通信領域(SQLCA)の使用

9.3 SQLSTATE状態変数

プリコンパイラのMODEコマンドライン・オプションは、ANSI/ISOへの準拠を制御します。MODE=ANSIのとき、SQLCAデータ構造体の宣言はオプションです。ただし、SQLCODEという状態変数は別に宣言する必要があります。SQL標準では、SQLSTATEという類似した状態変数が指定されています。SQLSTATEは、SQLCODEとともに使用しても別々に使用してもかまいません。

SQL文の実行後、Oracleサーバーは現在の適用範囲内のSQLSTATE変数にステータス・コードを戻します。ステータス・コードは、SQL文が正常に実行されたか例外(エラーまたは警告状態)が発生したかを示します。相互運用性(システム間で情報を簡単に交換する機能)を高めるために、共通のSQL例外がすべてSQL標準によってあらかじめ定義されています。

SQLCODEにはエラー・コードのみが格納されるのに対し、SQLSTATEにはエラー・コードおよび警告コードが格納されます。さらに、SQLSTATEの報告のメカニズムには、標準化されたコード化方式が採用されています。このため、SQLSTATEは状態変数として優先されます。SQLCODEは、SQL-89との互換性を保つためにのみ維持されていた、SQL-92の非推奨の機能です。SQL-92より後のすべてのSQL標準バージョンで、SQLCODEは削除されています。

9.3.1 SQLSTATEの宣言について

MODE=ANSIのときは、SQLSTATEまたはSQLCODEを宣言する必要があります。SQLCAの宣言はオプションです。MODE=ORACLEのときは、SQLSTATEを宣言しても無視されます。

SQLCODEでは符号付き整数を格納し、宣言部の外で宣言できますが、SQLSTATEではヌル文字で終了する5文字の文字列を格納し、宣言部の中で宣言する必要があります。次のようにSQLSTATEを宣言します。

```
char  SQLSTATE[6];  /* Upper case is required. */
```

注意:

SQLSTATE は正確に 6 文字のサイズで宣言する必要があります。

9.3.2 SQLSTATE値

SQLSTATEステータス・コードは、2文字のクラス・コードおよびその後に続く3文字のサブクラス・コードで構成されます。クラス・コード00(正常終了)以外のときには、クラス・コードは例外のカテゴリを示します。また、サブクラス・コード000(適用外)以外では、サブクラス・コードはそのカテゴリ内の特定の例外を示します。たとえば、SQLSTATEの値22012はクラス・コード22(データ例外)とサブクラス・コード012(ゼロ除算)を示します。

SQLSTATE値の5文字は、それぞれ数字(0から9)または大文字の英文字(AからZ)で構成されます。0から4の範囲の数字、またはAからHの範囲の文字で始まるクラス・コードは、事前定義済の状態(SQL標準で定義されている)用に確保されています。他のすべてのクラス・コードは実装定義の状態用に確保されています。事前定義クラスのうち、0から4の数字またはAからHの文字で始まるサブクラス・コードは、事前定義の副条件用に予約されています。他のすべてのサブクラス・コードは、実装時に定義される副条件用に予約されています。図9-1にコード化体系を示します。

図9-1 SQLSTATEコード化体系

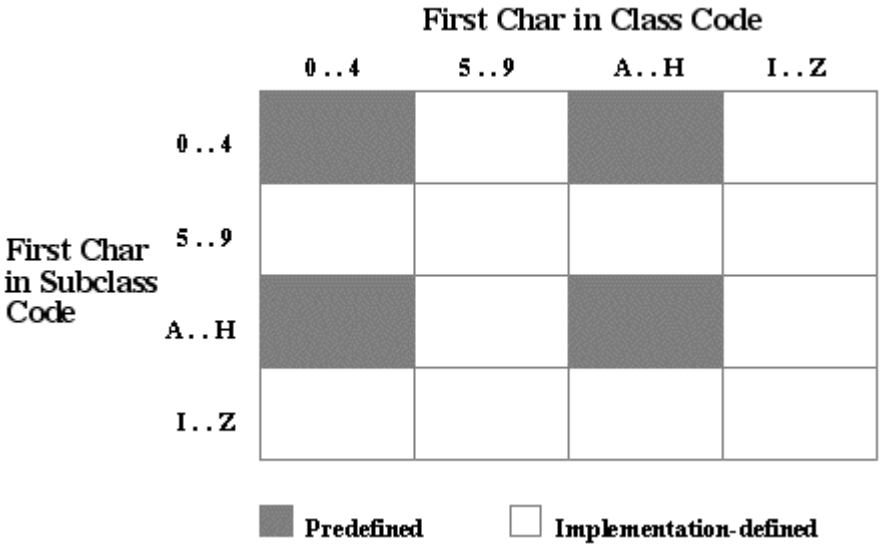


表9-1にSQL92で事前定義済のクラスを示します。

表9-1 事前定義済のクラス・コード

クラス	条件
00	正常終了
01	警告
02	データなし
07	動的 SQL エラー
08	接続例外
09	トリガー・アクション例外
0A	サポートされていない機能
0D	ターゲット・タイプの指定が無効
0E	スキーマ名リストの指定が無効
0F	ロケータ例外
0L	権限付与者が無効
0M	SQL が呼び出したプロシージャ参照が無効
0P	ロール指定が無効

クラス	条件
0S	変換グループ名の指定が無効
0T	ターゲット表がカーソル仕様と不一致
0U	更新可能でない列への割当て試行
0V	順序付け列への割当て試行
0W	トリガー実行中の禁止された文の発生
0Z	診断例外
21	制約違反
22	データ例外
23	整合性制約違反
24	カーソル状態が無効
25	トランザクション状態が無効
26	SQL 文名が無効
27	トリガー・データの変更違反
28	認証の指定が無効
2A	直接 SQL 構文エラーまたはアクセス規則違反
2B	依存権限記述子がまだ存在しています。
2C	キャラクタ・セット名が無効
2D	トランザクションの終了が無効
2E	接続名が無効
2F	SQL ルーチン例外

クラス	条件
2H	照合名が無効
30	SQL 文識別子が無効
33	SQL 記述子名が無効
34	カーソル名が無効
35	条件番号が無効
36	カーソル更新検出例外
37	動的 SQL 構文エラーまたはアクセス規則違反
38	外部ルーチン例外
39	外部ルーチン呼出しの例外
3B	セーブポイント例外
3C	カーソル名があいまい
3D	カタログ名が無効
3F	スキーマ名が無効
40	トランザクションのロールバック
42	構文エラーまたはアクセス規則違反
44	WITH_CHECK_OPTION 指定違反
HZ	リモート・データベース・アクセス

注意:

クラス・コード HZ は、国際標準規格 ISO/IEC DIS 9579-2 で定義された条件であるリモート・データベース・アクセス用に確保されています。

表9-2に、SQLSTATEステータス・コードと条件のOracleエラーとの対応を示します。60000から99999の範囲のステータス・コードは実装時に定義されます。

表9-2 SQLSTATEステータス・コード

コード	条件	Oracleエラー
00000	正常終了	ORA-00000
01000	警告	--
01001	カーソル操作の競合	--
01002	切断エラー	--
01003	集合関数で NULL 値が削除されている	--
01004	文字列データの右側切捨て	--
01005	項目記述子領域が不十分	--
01006	権限が取り消されていない	--
01007	権限が付与されていない	--
01008	暗黙的なゼロビットの埋込み	--
01009	情報スキーマの検索条件が長すぎます。	--
0100A	情報スキーマの問合せ式が長すぎます。	--
02000	データなし	ORA-01095 ORA-01403
07000	動的 SQL エラー	--
07001	USING 句がパラメータ指定と一致しません。	--
07002	USING 句が相手指定と一致しません。	--

コード	条件	Oracleエラー
07003	カーソル仕様を実行できません。	--
07004	動的パラメータには USING 句が必要です。	--
07005	プリコンパイルされた SQL 文がカーソル仕様ではありません。	--
07006	制限付きのデータ型属性違反	--
07007	結果コンポーネントには USING 句が必要、記述子の数が無効	--
07008	記述子の数が無効	SQL-02126
07009	記述子の索引が無効	--
08000	接続例外	--
08001	SQL クライアントは SQL 接続を確立できない	--
08002	接続名を使用中	--
08003	接続が存在しません。	SQL-02121
08004	SQL サーバーが SQL 接続を拒否した	--
08006	接続障害	--
08007	トランザクションの結果が不明	--
0A000	サポートされていない機能	ORA-03000 から 03099
0A001	複数サーバー・トランザクション	--
21000	制約違反	ORA-01427 SQL-02112
22000	データ例外	--
22001	文字列データの右側切捨て	ORA-01406

コード	条件	Oracleエラー
22002	NULL 値 - インジケータ・パラメータなし	SQL-02124
22003	数値が範囲外	ORA-01426
22005	割当てのエラー	--
22007	日時書式が無効	--
22008	日時フィールドのオーバーフロー	ORA-01800 から 01899
22009	タイム・ゾーンによる時差が無効	--
22011	部分文字列のエラー	--
22012	0 による除算	ORA-01476
22015	間隔フィールドのオーバーフロー	--
22018	キャストの文字値が無効	--
22019	エスケープ文字が無効	ORA-00911
22021	レパートリに文字がありません。	--
22022	インジケータのオーバーフロー	ORA-01411
22023	パラメータ値が無効	ORA-01025 ORA-04000 から 04019
22024	C 文字列が未終了	ORA-01479 ORA-01480
22025	エスケープ・シーケンスが無効	ORA-01424 ORA-01425
22026	文字列データの長さが不一致	ORA-01401

コード	条件	Oracleエラー
22027	切捨てエラー	-
23000	整合性制約違反	ORA-1400、ORA-02290 から 02299
24000	カーソル状態が無効	ORA-001002 ORA-001003 SQL-02114 SQL-02117
25000	トランザクション状態が無効	SQL-02118
26000	SQL 文名が無効	--
27000	トリガー・データの変更違反	--
28000	認証の指定が無効	--
2A000	直接 SQL 構文エラーまたはアクセス規則違反	--
2B000	依存権限記述子がまだ存在しています。	--
2C000	キャラクタ・セット名が無効	--
2D000	トランザクションの終了が無効	--
2E000	接続名が無効	--
33000	SQL 記述子名が無効	--
34000	カーソル名が無効	--
35000	条件番号が無効	--
37000	動的 SQL 構文エラーまたはアクセス規則違反	--
3C000	カーソル名があいまい	--

コード	条件	Oracleエラー
3D000	カタログ名が無効	--
3F000	スキーマ名が無効	--
40000	トランザクションのロールバック	ORA-02091 ORA-02092
40001	シリアライズの失敗	--
40002	整合性制約違反	--
40003	文の完了が不明	--
42000	構文エラーまたはアクセス規則違反	ORA-00022 ORA-00251 ORA-00900 から 00999 ORA-01031 ORA-01490 から 01493 ORA-01700 から 01799 ORA-01900 から 02099 ORA-02140 から 02289 ORA-02420 から 02424 ORA-02450 から 02499 ORA-03276 から 03299 ORA-04040 から 04059 ORA-04070 から 04099
44000	WITH_CHECK_OPTION 指定違反	ORA-01402
60000	システム・エラー	ORA-00370 から 00429

コード	条件	Oracleエラー
		ORA-00600 から 00899
		ORA-06430 から 06449
		ORA-07200 から 07999
		ORA-09700 から 09999
61000	共有サーバーおよび分離プロセスのエラー	ORA-00018 から 00035
		ORA-00050 から 00068
		ORA-02376 から 02399
		ORA-04020 から 04039
62000	共有サーバーおよび分離プロセスのエラー	ORA-00100 から 00120
		ORA-00440 から 00569
63000	Oracle*XA および 2 タスク・インタフェースのエラー	ORA-00150 から 00159
		ORA-02700 から 02899
		ORA-03100 から 03199
		ORA-06200 から 06249
		SQL-02128
64000	制御ファイル、データベース・ファイルおよび REDO ファイルのエラー、アーカイブおよびメディア・リカバリのエラー	ORA-00200 から 00369
		ORA-01100 から 01250
65000	PL/SQL のエラー	ORA-06500 から 06599
66000	Oracle Net ドライバのエラー	ORA-06000 から 06149
		ORA-06250 から 06429
		ORA-06600 から 06999
		ORA-12100 から 12299

コード	条件	Oracleエラー
		ORA-12500 から 12599
67000	ライセンス許可エラー	ORA-00430 から 00439
69000	SQL*Connect のエラー	ORA-00570 から 00599
		ORA-07000 から 07199
72000	SQL 実行フェーズのエラー	ORA-00001
		ORA-01000 から 01099
		ORA-01401 から 01489
		ORA-01495 から 01499
		ORA-01500 から 01699
		ORA-02400 から 02419
		ORA-02425 から 02449
		ORA-04060 から 04069
		ORA-08000 から 08190
		ORA-12000 から 12019
		ORA-12300 から 12499
		ORA-12700 から 21999
82100	メモリー不足のためメモリーが割り当てられません。	SQL-02100
82101	カーソル・キャッシュが矛盾(UCE/CUC が不一致)	SQL-02101
82102	カーソル・キャッシュが矛盾(UCE の CUC エントリがない)	SQL-02102
82103	カーソル・キャッシュが矛盾(CUC 参照の範囲外)	SQL-02103
82104	カーソル・キャッシュが矛盾(使用可能な CUC がない)	SQL-02104
82105	カーソル・キャッシュが矛盾(キャッシュに CUC エントリがない)	SQL-02105

コード	条件	Oracleエラー
82106	カーソル・キャッシュが矛盾(カーソル番号が無効)	SQL-02106
82107	ランタイム・ライブラリに対してプログラムが古すぎる; 再プリコンパイルが必要	SQL-02107
82108	ランタイム・ライブラリに無効な記述子が渡されました。	SQL-02108
82109	ホスト・キャッシュが矛盾(SIT 参照が範囲外)	SQL-02109
82110	ホスト・キャッシュが矛盾(SQL の型が無効)	SQL-02110
82111	ヒープ一貫性のエラー	SQL-02111
82113	コード生成の内部整合性の障害	SQL-02115
82114	リエントラント・コード・ジェネレータが無効なコンテキストを与えました。	SQL-02116
82117	データベースへの接続での OPEN または PREPARE が無効です。	SQL-02122
82118	アプリケーション・コンテキストが見つかりません。	SQL-02123
82119	エラー・メッセージのテキストを取り出せない	SQL-02125
82120	プリコンパイラと SQLLIB のバージョンが不一致	SQL-02127
82121	NCHAR エラー; フェッチされたバイト数が奇数	SQL-02129
82122	EXEC TOOLS インタフェースが使用できない	SQL-02130
82123	ランタイム・コンテキストは使用中です。	SQL-02131
82124	ランタイム・コンテキストを割り当てできません。	SQL-02132
82125	スレッドで使用するプロセスを初期化できません。	SQL-02133
82126	ランタイム・コンテキストが無効	SQL-02134
HZ000	リモート・データベース・アクセス	--

9.3.3 SQLSTATEの使用について

次の規則は、オプションの設定をMODE=ANSIにしてプリコンパイルするときに、SQLSTATEをSQLCODEまたはSQLCAと併用する場合に適用されます。SQLSTATEは宣言部内で宣言する必要があります。宣言部内で宣言しなければ無視されます。

9.3.3.1 SQLSTATEを宣言する場合

- SQLCODEの宣言はオプションです。宣言部の内部でSQLCODEを宣言すると、SQL処理を実行するたびにOracleサーバーはSQLSTATEおよびSQLCODEにステータス・コードを戻します。ただし、宣言部の外部でSQLCODEを宣言すると、OracleはSQLSTATEのみにステータス・コードを戻します。
- SQLCAの宣言はオプションです。SQLCAを宣言すると、OracleはSQLSTATEおよびSQLCAにステータス・コードを戻します。この場合は、コンパイル・エラーを防ぐために、SQLCODEを宣言しないでください。

9.3.3.2 SQLSTATEを宣言しない場合

- 宣言部の内部または外部で、SQLCODEを宣言する必要があります。SQL処理を実行するたびに、OracleサーバーはSQLCODEにステータス・コードを戻します。
- SQLCAの宣言はオプションです。SQLCAを宣言すると、OracleはSQLCODEおよびSQLCAにステータス・コードを戻します。

独自のコードを作成してSQLSTATEを明示的にチェックするか、WHENEVER SQLERRORディレクティブを使用してSQLSTATEを暗黙的にチェックすることで、最新の実行SQL文の結果が得られます。実行SQL文およびPL/SQL文の後にのみSQLSTATEをチェックしてください。

9.4 SQLCODEの宣言について

MODE=ANSIで、SQLSTATE状態変数を宣言していない場合は、宣言部の内側または外側でSQLCODEという**long**型の整数の変数を宣言する必要があります。次に例を示します。

```
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
int  emp_number, dept_number;
char emp_name[20];
EXEC SQL END DECLARE SECTION;

/* declare status variable--must be upper case */
long SQLCODE;
```

MODE=ORACLEの場合は、SQLCODEを宣言しても無視されます。

複数のSQLCODEを宣言することが可能です。ローカルなSQLCODEへのアクセスは、プログラム内の適用範囲によって制限されます。

SQLの動作が終わるたびに、Oracleから現在有効範囲にあるSQLCODEにステータス・コードが戻されます。したがって、プログラムでは、SQLCODEを明示的にチェックするか、WHENEVERディレクティブを暗黙的に指定することで、最新のSQL処理の結果を知ることができます。

特定のコンパイル・ユニットのSQLCAのかわりにSQLCODEを宣言すると、プリコンパイラでは、そのユニット用に内部SQLCAを1つ割り当てます。ホスト・プログラムでは、その内部SQLCAにアクセスできません。SQLCAおよびSQLCODEの両方を宣言すると、OracleはSQL操作を実行するたびに同じステータス・コードを両方に戻します。

9.5 SQLCAを使用したエラー・レポートの主要コンポーネント

エラー・レポートはSQLCA内の変数によって異なります。この項では、エラー・レポートの主要コンポーネントについて説明します。また、この次の項ではSQLCAを詳しく説明します。

9.5.1 ステータス・コード

すべての実行SQL文は、SQLCA変数*sqlcode*にステータス・コードを戻します。戻されたステータス・コードはWHENEVERディレクティブによって暗黙的に、あるいは独自のコードによって明示的にチェックできます。

0(ゼロ)のステータス・コードは、Oracleがエラーまたは例外を検出せずに文を実行したことを意味します。正のステータス・コードは、Oracleが例外を検出した上で文を実行したことを意味します。負のステータス・コードは、エラーが発生したためにOracleがSQL文を実行しなかったことを意味します。

9.5.2 警告フラグ

警告フラグはSQLCA変数のsqlwarn[0]からsqlwarn[7]に戻されます。これは暗黙的にも明示的にもチェックできます。警告フラグは、Oracleでエラーとみなされない実行時の条件をチェックするのに便利です。標識変数がなければ、Oracleではエラー・メッセージが発行されます。

9.5.3 処理済行数

最後に実行したSQL文で処理された行数は、SQLCA変数sqlca.sqlerrd[2]に戻されます。これは明示的にチェックできます。

厳密にはこの変数はエラー・レポート用ではなく、誤りを防止するためのものです。たとえば、表から約10行を削除するとします。削除処理後にsqlca.sqlerrd[2]をチェックすると、75行が削除されていました。このような場合は、念のため削除処理をロールバックしてWHERE句の検索条件を確認することが可能です。

9.5.4 解析エラー・オフセット

SQL文は実行前に必ず解析され、構文規則に従っているか、有効なデータベース・オブジェクトを参照しているかが検証されます。エラーが検出されると、SQLCA変数sqlca.sqlerrd[4]にオフセットが格納されます。これは明示的にチェックできます。このオフセットには、解析エラーの始まりを示すSQL文中の文字位置が示されています。通常のC言語の文字列と同様に、先頭の文字位置は0(ゼロ)です。たとえば、オフセットが9のとき、解析エラーは10番目の文字から始まっています。

解析エラー・オフセットは、準備と解析が別々に実行される状況で使用されます。その代表例は動的SQL文です。

解析エラーは、キーワードの欠落、キーワードの位置指定の誤り、キーワードのスペルミス、無効なオプションなどが原因で発生します。次に、動的SQL文の例を示します。

```
"UPDATE emp SET jib = :job_title WHERE empno = :emp_number"
```

これは解析エラーになります。

```
ORA-00904: invalid column name
```

原因は、列名JOBのスペルミスです。sqlca.sqlerrd[4] の値は15になりますが、これは誤った列名JIBが16番目の文字で始まっているためです。

SQL文に解析エラーがなければ、Oracleではsqlca.sqlerrd[4]が0(ゼロ)に設定されます。解析エラーが先頭の文字(文字位置は0(ゼロ))で始まっているときも、sqlca.sqlerrd[4]は0(ゼロ)に設定されます。このため、sqlca.sqlerrd[4]のチェックは、*sqlca.sqlcode*が負の値(エラーが発生したことを示す)の場合にのみ行ってください。

9.5.5 エラー・メッセージ・テキスト

Oracleエラーのエラー・コードおよびメッセージはSQLCA変数SQLERRMCに格納されます。テキストの最初の最大70文字が格納されます。70文字を超えるメッセージすべてを取得するには、sqlglm()関数を使用します。

関連項目

- エラー・メッセージの全文の取得について

9.6 SQL通信領域(SQLCA)の使用

SQLCAはデータ構造体です。そのコンポーネントには、SQL文を実行するたびにOracleによって更新されるエラー、警告およびステータス情報が格納されます。したがって、SQLCAには常に最新のSQLの動作結果が反映されます。この結果を判断するために、SQLCA内の変数をチェックすることが可能です。

プログラムでは複数のSQLCAを使用できます。たとえば、1つのグローバルSQLCAと複数のローカルSQLCAを設定できます。ローカルSQLCAへのアクセスは、プログラム内の有効範囲により制限されます。Oracleは、スコープ内にあるSQLCAにのみ情報を戻します。

> 注意:
>
> アプリケーションで Oracle Net を使用してローカル・データベースおよびリモート・データベースの組合せに同時にアクセスする場合、すべてのデータベースは 1 つの SQLCA に書き込みます。つまり、データベースごとに異なる SQLCA があるわけではありません。

関連項目

- 高度な接続オプション

9.6.1 SQLCAの宣言について

MODE=ORACLEのときは、SQLCAの宣言が必要です。SQLCAを宣言するには、次に示すようにINCLUDEまたは**#include**文を使用してSQLCAをプログラム内にコピーします。

```
EXEC SQL INCLUDE SQLCA;
```

または

```
#include <sqlca.h>
```

宣言部を使用するときは、SQLCAは宣言部の外側で宣言する必要があります。SQLCAを宣言しないとコンパイル時にエラーが発生します。

プログラムをプリコンパイルすると、INCLUDE SQLCA文はいくつかの変数宣言に置き換えられます。これらの変数宣言によって、Oracleとプログラムとの間の通信が可能になります。

MODE=ANSIのときは、SQLCAの宣言はオプションです。ただしこのとき、SQLCODEまたはSQLSTATE状態変数は宣言する必要があります。SQLCODE (必ず大文字)の型は**int**です。特定のコンパイル・ユニットでSQLCAのかわりにSQLCODEまたはSQLSTATEを宣言した場合、プリコンパイラではそのユニット用に内部SQLCAが割り当てられます。Pro*C/C++プログラムは、内部SQLCAにはアクセスできません。SQLCAおよびSQLCODEの両方を宣言すると、OracleはSQL操作を実行

するたびに同じステータス・コードを両方に戻します。

> **注意:**
>
> SQLCA の宣言は MODE=ANSI のときにはオプションですが、WHENEVER SQLWARNING ディレクティブは SQLCA がなければ使用できません。したがって、WHENEVER SQLWARNING ディレクティブを使用する場合は、SQLCA を宣言する必要があります。
>
> このマニュアルでは、SQLCODE 状態変数を指す場合に SQLCODE を使用しています。また、SQLCA 構造体のコンポーネントのことを明示する場合には、*sqlca.sqlcode* を使用しています。

9.6.2 SQLCAの内容

SQLCAにはSQL文の実行結果に関する次の情報が格納されます。

- Oracleエラー・コード
- 警告フラグ
- イベント情報
- 処理済行数
- 診断情報

sqlca.hヘッダー・ファイルは次のとおりです。

```
/*
NAME
  SQLCA : SQL Communications Area.
FUNCTION
  Contains no code. Oracle fills in the SQLCA with status info
  during the execution of a SQL stmt.
NOTES
  **************************************************************
  ***                                                        ***
  *** This file is SOSD.  Porters must change the data types ***
  *** appropriately on their platform.  See notes/pcport.doc ***
  *** for more information.                                  ***
  ***                                                        ***
  **************************************************************

  If the symbol SQLCA_STORAGE_CLASS is defined, then the SQLCA
  will be defined to have this storage class. For example:

    #define SQLCA_STORAGE_CLASS extern

  will define the SQLCA as an extern.

  If the symbol SQLCA_INIT is defined, then the SQLCA will be
  statically initialized. Although this is not necessary in order
  to use the SQLCA, it is a good programing practice not to have
  unitialized variables. However, some C compilers/operating systems
  don't allow automatic variables to be initialized in this manner.
  Therefore, if you are INCLUDE'ing the SQLCA in a place where it
```

```
  would be an automatic AND your C compiler/operating system doesn't
  allow this style of initialization, then SQLCA_INIT should be left
  undefined -- all others can define SQLCA_INIT if they wish.

  If the symbol SQLCA_NONE is defined, then the SQLCA
  variable will not be defined at all.  The symbol SQLCA_NONE
  should not be defined in source modules that have embedded SQL.
  However, source modules that have no embedded SQL, but need to
  manipulate a sqlca struct passed in as a parameter, can set the
  SQLCA_NONE symbol to avoid creation of an extraneous sqlca
  variable.
*/
#ifndef SQLCA
#define SQLCA 1
struct   sqlca
         {
         /* ub1 */ char    sqlcaid[8];
         /* b4  */ long    sqlabc;
         /* b4  */ long    sqlcode;
         struct
           {
           /* ub2 */ unsigned short sqlerrml;
           /* ub1 */ char           sqlerrmc[70];
           } sqlerrm;
         /* ub1 */ char    sqlerrp[8];
         /* b4  */ long    sqlerrd[6];
         /* ub1 */ char    sqlwarn[8];
         /* ub1 */ char    sqlext[8];
         };
#ifndef SQLCA_NONE
#ifdef   SQLCA_STORAGE_CLASS
SQLCA_STORAGE_CLASS struct sqlca sqlca
#else
         struct sqlca sqlca
#endif
#ifdef  SQLCA_INIT
         = {
         {'S', 'Q', 'L', 'C', 'A', ' ', ' ', ' '},
         sizeof(struct sqlca),
         0,
         { 0, {0}},
         {'N', 'O', 'T', ' ', 'S', 'E', 'T', ' '},
         {0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0, 0}
         }
#endif
         ;
#endif
#endif
```

9.6.3 SQLCAの構造

この項では、SQLCAの構造体、そのコンポーネントおよびコンポーネントに格納できる値について説明します。

9.6.3.1 *sqlcaid*

この文字列コンポーネントはSQLCAに初期化されて、SQL通信領域を示します。

9.6.3.2 *sqlcabc*

この整数コンポーネントにはSQLCA構造体の長さがバイト単位で格納されます。

9.6.3.3 *sqlcode*

この整数コンポーネントには最後に実行されたSQL文のステータス・コードが格納されます。SQLの動作の結果を示すステータス・コードは、次のいずれかの数値です。

ステータス・コード	説明
0	エラーまたは例外の検出なしで文が実行されたことを示します。
>0	文は実行されたが、例外が検出されたことを示します。この状態が発生するのは、WHERE 句の検索条件を満たす行がない場合、あるいは SELECT INTO または FETCH で 1 行も戻されなかった場合です。

MODE=ANSIのときは、どの行もINSERTできないと+100が*sqlcode*に戻されます。副問合せで処理に行が戻されなかったときにこの状態が発生します。

- <0はデータベース、システム、ネットワークまたはアプリケーションのエラーが原因で、文が実行されなかったことを示します。このようなエラーは致命的です。このようなエラーが発生すると、ほとんどの場合はカレント・トランザクションがロールバックされます。

エラー・コードに対応する負数のリターン・コードは、*Oracle Database*エラー・メッセージに記載されています

9.6.3.4 *sqlerrm*

この埋込み構造体には次の2つのコンポーネントがあります。

コンポーネント	説明
sqlerrml	この整数コンポーネントには、sqlerrmc 内に保存されているメッセージ・テキストの長さが格納されます。
sqlerrmc	この文字列コンポーネントには、sqlcode 内に保存されているエラー・コードに対応したメッセージ・テキストが格納されます。文字列はヌル文字で終了しません。長さを調べるには sqlerrml コンポーネントを使用します。

このコンポーネントには最大70文字まで格納できます。71文字以上のメッセージ全体を取得するには、この後で説明する`sqlglm`関数を使用します。

*sqlerrmc*を参照する前に、*sqlcode*が負であることを確認する必要があります。*sqlcode*が0(ゼロ)のときに*sqlerrmc*を参照すると、以前のSQL文に対応するメッセージが取得されることになります。

9.6.3.5 *sqlerrp*

この文字列コンポーネントは将来使用するために予約されています。

9.6.3.6 *sqlerrd*

この2進整数の配列には6つの要素があります。*sqlerrd*内のコンポーネントの説明を次に示します。

コンポーネント	説明
sqlerrd[0]	このコンポーネントは将来使用するために予約されています。
sqlerrd[1]	このコンポーネントは将来使用するために予約されています。
sqlerrd[2]	このコンポーネントには、その時点で最後に実行した SQL 文によって処理された行数が格納されます。ただし、SQL 文が失敗すると、1 つの例外を除き sqlca.sqlerrd[2]の値は未定義となります。配列処理中にエラーが発生すると、そのエラーの発生した行で処理は停止します。そのため、sqlca.sqlerrd[2]は正常に処理された行数を示します。

処理済行数はOPEN文の後に0 (ゼロ)に設定され、FETCH文の後に増分されます。処理済行数は、EXECUTE文、INSERT文、UPDATE文、DELETE文およびSELECT INTO文について、正常に処理された行数を反映します。この数には、UPDATEやDELETE_CASCADEで処理された行は含まれません。たとえばWHERE句の条件を満たす20行が削除された後で、列制約条件に違反する5行が削除されたときの処理済行数は、25ではなく20となります。

コンポーネント	説明
sqlerrd[3]	このコンポーネントは将来使用するために予約されています。
sqlerrd[4]	このコンポーネントは、最後に実行された SQL 文中で解析エラーの始まりを示す文字位置を指定するオフセットを保持します。先頭の文字位置は 0 (ゼロ)です。
sqlerrd[5]	このコンポーネントは将来使用するために予約されています。

9.6.3.7 *sqlwarn*

この1文字の配列には8つの要素があります。これらの要素は警告フラグとして使用されます。Oracleではそれに文字値W(警告)を割り当てることでフラグを設定します。

フラグは例外状態の発生を警告します。たとえば、切り捨てられた列値が出力ホスト変数に割り当てられると、警告フラグが設定されます。

*sqlwarn*のコンポーネントの説明を次に示します。

コンポーネント	説明
sqlwarn[0]	このフラグは別の警告フラグが設定されていることを示します。
sqlwarn[1]	このフラグは、切り捨てられた列値が出力ホスト変数に代入されたときに設定されます。これは文字データにのみ適用されます。つまり、Oracle は、警告で設定することも負の sqlcode を戻すこともなく、特定の数値データを切り捨てます。

列値が切り捨てられたかどうか、またどれだけ切り捨てられたかを調べるには、出力ホスト変数に対応する標識変数をチェックします。標識変数によって戻された値が正の整数のときは、その値は列値の元の長さを示します。その値に応じてホスト変数の長さを増やすことができます。

コンポーネント	説明
sqlwarn[2]	AVG()や SUM()などの SQL グループ関数の結果に NULL 列が使用されない場合、このフラグが設定されます。
sqlwarn[3]	問合せの選択リスト内の列の数が SELECT 文または FETCH 文の INTO 句内のホスト変数の数と一致しないときに、このフラグが設定されます。戻される項目の数は両者のうち少ない方の数となります。
sqlwarn[4]	このフラグは現在使用されていません。
sqlwarn[5]	PL/SQL のコンパイル・エラーが原因で EXEC SQL CREATE {PROCEDURE | FUNCTION | PACKAGE | PACKAGE BODY}文が失敗したときに、このフラグが設定されます。
sqlwarn[6]	このフラグは現在使用されていません。
sqlwarn[7]	このフラグは現在使用されていません。

9.6.3.8 *sqlext*

この文字列コンポーネントは将来使用するために予約されています。

9.6.4 PL/SQLの考慮事項

プリコンパイラ・アプリケーションで埋込みPL/SQLブロックを実行するときに、SQLCAのすべてのコンポーネントが設定されるわけではありません。たとえば、ブロックが複数の行をフェッチするときは、処理済行数(sqlerrd[2])には1しか設定されません。PL/SQLブロックの実行後は、SQLCAの*sqlcode*および*sqlerrm*コンポーネントのみを使用してください。

9.7 エラー・メッセージの全文の取得について

SQLCAには70文字までのエラー・メッセージを格納できます。71文字以上の(またはネストされた)エラー・メッセージ全体を取得するには、sqlglm()関数を使用する必要があります。次に構文を示します。

```
void sqlglm(unsigned char   *message_buffer,
            size_t *buffer_size,
            size_t *message_length);
```

各パラメータの意味は次のとおりです。

構文	説明
message_buffer	エラー・メッセージを格納するためのテキスト・バッファです(Oracle はバッファの最後まで空白文字で埋め込みます)。
buffer_size	バッファの最大サイズをバイト数で示したスカラー変数です。

構文	説明
message_length	Oracle によって格納されたエラー・メッセージの切り捨てられていない場合の、実際の長さを示すスカラー変数です。

注意:

sqlglm() 関数の最後の 2 つの引数の型は一般的な size_t ポインタとして示してあります。ただし、プラットフォームによっては型が異なることがあります。たとえば、多くの UNIX ワークステーション・ポートでは、unsigned int ***になります。**

これらのパラメータのデータ型を判別するには、システムの標準インクルード・ディレクトリにある sqlcpr.h ファイルをチェックしてください。

Oracleエラー・メッセージの最大長は、エラー・コード、ネストされたメッセージ、表や列の名前など、メッセージの挿入部分を含めて512文字です。sqlglmによって戻されるエラー・メッセージの最大長は、*buffer_size*に指定した値によって決まります。

次の例では、sqlglmをコールして、200文字以内の長さのエラー・メッセージを取得します。

```
EXEC SQL WHENEVER SQLERROR DO sql_error();
...
/* other statements */
...
sql_error()
{
    char msg[200];
    size_t buf_len, msg_len;

    buf_len = sizeof (msg);
    sqlglm(msg, &buf_len, &msg_len);    /* note use of pointers */
    if (msg_len > buf_len)
    msg_len = buf_len;
    printf("%.*s¥n¥n", msg_len, msg);
    exit(1);
}
```

sqlglm()は、SQLエラーが発生したときにのみコールできます。*sqlglm*をコールする前に、SQLCODE (または*sqlca.sqlcode*)の値が0(ゼロ)でないことを必ず確認してください。SQLCODEが0(ゼロ)のときにsqlglm()をコールすると、以前のSQL文に対応したメッセージが取得されることになります。

注意:

複数のランタイム・コンテキストが使用される場合は、コンテキストを持つバージョンの sqlglmt()を使用して、適切なエラー・メッセージを取得してください。

関連項目

- マルチスレッド・アプリケーション

9.8 WHENEVERディレクティブの使用について

デフォルトでは、プリコンパイルされたプログラムはOracleエラーおよび警告の状態を無視し、可能であれば処理を続行します。自動条件チェックおよびエラー処理を実行するにはWHENEVERディレクティブが必要です。

WHENEVERディレクティブによって、OracleでエラーやSQLERROR、SQLWARNINGまたはNOT FOUND条件が検出されたときのアクションを指定できます。これらのアクションには、次の文の継続実行、ルーチンのコール、ラベル付きの文への分岐、停止などがあります。

WHENEVERディレクティブの構文は次のとおりです。

```
EXEC SQL WHENEVER <condition> <action>;
```

9.8.1 WHENEVER条件

Oracleに自動的にSQLCAをチェックさせて、次の状態が存在しないかどうかを調べることができます。

9.8.1.1 SQLWARNING

sqlwarn[0]は、Oracleから警告(sqlwarn[1]からsqlwarn[7]までのいずれか1つも設定されます)が戻されたか、SQLCODEが+1403以外の正の値になっているために設定されます。たとえば、切り捨てられた列値が出力ホスト変数に割り当てられると、sqlwarn[0]が設定されます。

MODE=ANSIのときは、SQLCAの宣言はオプションです。ただし、WHENEVER SQLWARNINGを使用するには、必ずSQLCAを宣言してください。

9.8.1.2 SQLERROR

Oracleがエラーを戻したことにより、SQLCODEに負の値が設定されています。

9.8.1.3 NOT FOUND

OracleがWHERE句の検索条件を満たす行を検出できなかったか、SELECT INTOまたはFETCHが行を戻さなかったため、SQLCODEに+1403が設定されています(MODE=ANSIのときは+100)。

MODE=ANSIのときは、どの行もINSERTできないと+100がSQLCODEに戻されます。

9.8.2 WHENEVERアクション

Oracleで前述の状態のいずれかが検出されたときは、プログラムに次のいずれかのアクションを実行させることができます。

9.8.2.1 CONTINUE

可能であれば、プログラムは次の文からの実行を継続します。これはデフォルトの動作で、WHENEVERディレクティブを使用しない場合と同じです。このアクションを使用すると条件チェックを終了できます。

9.8.2.2 DO

制御をプログラム中のエラー処理関数に移します。ルーチンの最後に達すると、制御は失敗したSQL文の次の文に移ります。

関数の入力および終了について、通常の規則が適用されます。EXEC SQL WHENEVER ... DO ...ディレクティブで起動されるエラー・ハンドラにパラメータを渡し、関数によって値を戻すことができます。

9.8.2.3 DO BREAK

実際のbreak文はプログラム中に置かれます。このアクションはループ内で使用します。WHENEVER条件が成立すると、プログラムはそのループを抜けます。

9.8.2.4 DO CONTINUE

実際のcontinue文がプログラム中に置かれます。このアクションはループ内で使用します。WHENEVER条件が成立すると、プログラムはそのループの次の反復に移ります。

9.8.2.5 GOTO label_name

プログラムはラベル付き文に分岐します。ラベル名の長さに制限はありませんが、有効なのは最初の31文字のみです。異なる最大長を必要とするCコンパイラもあります。使用しているCコンパイラのユーザーズ・ガイドを参照してください。

9.8.2.6 STOP

プログラムは実行を停止し、COMMITされていない作業がロールバックされます。

STOPが実際に行うのは、WHENEVER条件発生時のexit()コールの生成のみです。ここでは注意が必要です。STOPアクションは、Oracleからの切断前に何もメッセージを表示しません。

9.8.3 WHENEVERの例

たとえば、プログラムで次のアクションが必要だとします。

- 「データが見つかりません」という条件が発生したときには、*close_cursor*に進みます。
- 警告が発生したときは、次の文を続行します。
- エラーが発生したときは、*error_handler*に進みます。

最初の実行SQL文の前に次のWHENEVERディレクティブを指定する必要があります。

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

次の例では、WHENEVER...DOディレクティブを使用して特定のエラーを処理します。

```
...
EXEC SQL WHENEVER SQLERROR DO handle_insert_error("INSERT error");
EXEC SQL INSERT INTO emp (empno, ename, deptno)
    VALUES (:emp_number, :emp_name, :dept_number);
EXEC SQL WHENEVER SQLERROR DO handle_delete_error("DELETE error");
EXEC SQL DELETE FROM dept WHERE deptno = :dept_number;
...
handle_insert_error(char *stmt)
{   switch(sqlca.sqlcode)
    {
    case -1:
    /* duplicate key value */
        ...
        break;
    case -1401:
    /* value too large */
        ...
        break;
    default:
    /* do something here too */
```

```
        ...
        break;
    }
}

handle_delete_error(char *stmt)
{
    printf("%s¥n¥n", stmt);
    if (sqlca.sqlerrd[2] == 0)
    {
        /* no rows deleted */
        ...
    }
    else
    {   ...
    }
    ...
}
```

SQLCAの変数をチェックしてアクションの過程を決定する手順に注意してください。

9.8.4 DO BREAKとDO CONTINUEの利用

次の例では、コミッションを受け取っている従業員の分のみ、従業員の名前、給料、コミッションを表示する方法を示しています。

```
#include <sqlca.h>
#include <stdio.h>

main()
{
    char *uid = "scott/tiger";
    struct { char ename[12]; float sal; float comm; } emp;

    /* Trap any connection error that might occur. */
    EXEC SQL WHENEVER SQLERROR GOTO whoops;
    EXEC SQL CONNECT :uid;

    EXEC SQL DECLARE c CURSOR FOR
        SELECT ename, sal, comm FROM EMP ORDER BY ENAME ASC;

    EXEC SQL OPEN c;

    /* Set up 'BREAK' condition to exit the loop. */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;
   /* The DO CONTINUE makes the loop start at the next iteration when an error occurs.*/
    EXEC SQL WHENEVER SQLERROR DO CONTINUE;

    while (1)
      {
          EXEC SQL FETCH c INTO :emp;
   /* An ORA-1405 would cause the 'continue' to occur. So only employees with */
   /* non-NULL commissions will be displayed. */
          printf("%s  %7.2f  %9.2f¥n", emp.ename, emp.sal, emp.comm);
       }

/* This 'CONTINUE' shuts off the 'DO CONTINUE' allowing the program to
   proceed if any further errors do occur, specifically, with the CLOSE */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    EXEC SQL CLOSE c;
```

```
    exit(EXIT_SUCCESS);

whoops:
    printf("%.*s¥n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    exit(EXIT_FAILURE);
}
```

9.8.5 WHENEVER文の適用範囲

WHENEVER文は宣言部のため、そのスコープは論理的なものではなく位置的なものになります。つまり、WHENEVER文はプログラム・ロジックの流れではなく、ソース・ファイル内で物理的にWHENEVER文に続く実行SQL文をすべてテストします。したがって、WHENEVERディレクティブはテストする最初の実行SQL文の前に指定する必要があります。

あるWHENEVERディレクティブは、同じ条件をチェックする別のWHENEVERディレクティブに置き換えられるまでの間は有効です。

次の例では、最初のWHENEVER SQLERRORディレクティブは2番目のものに置き換えられます。したがって、このディレクティブの制御はCONNECT文のみに適用されます。2番目のWHENEVER SQLERRORディレクティブは、*step1*から*step3*への制御の流れに関係なく、UPDATE文およびDROP文の両方に適用されます。

```
step1:
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    ...
    goto step3;
step2:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL UPDATE emp SET sal = sal * 1.10;
    ...
step3:
    EXEC SQL DROP INDEX emp_index;
    ...
```

9.8.6 WHENEVERのガイドライン

この項では、一般的な問題点を回避するためのガイドラインを示します。

9.8.6.1 文の位置

通常、WHENEVERディレクティブはプログラムの最初の実行SQL文の前に指定します。この位置に指定したWHENEVERディレクティブはファイルの最後まで有効になるため、発生するすべてのエラーを確実にトラップできます。

9.8.6.2 データの終わり条件の処理

カーソルを使用して行をフェッチするときは、プログラムでデータの終了条件を処理できるようにしておく必要があります。FETCHがデータを戻さないときは、プログラムは次のようにFETCHループを終了します。

```
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH...
}
EXEC SQL CLOSE my_cursor;
...
```

行が挿入されていない場合、INSERTはNOT FOUNDを戻します。この条件を取り上げない場合は、INSERTの前にEXEC SQL WHENEVER NOT FOUND CONTINUEを使用します。

```
EXEC SQL WHENEVER NOT FOUND DO break;
for(;;)
{
   EXEC SQL FETCH ...
   EXEC SQL WHENEVER NOT FOUND CONTINUE;
   EXEC SQL INSERT INTO ...
}
EXEC SQL CLOSE my_cursor;
...
```

9.8.6.3 無限ループの回避について

WHENEVER SQLERROR GOTOディレクティブが、実行SQL文を含むエラー処理ルーチンに分岐しているときに、そのSQL文にエラーが発生すると、プログラムが無限ループに陥るおそれがあります。無限ループを回避するには、次に示すようにSQL文の前にWHENEVER SQLERROR CONTINUEを記述します。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    ...
```

WHENEVER SQLERROR CONTINUE文を指定しなければ、ROLLBACKエラーが発生したときにこのルーチンが再び実行されるため、結果として無限ループに陥ります。

WHENEVER句は、注意して使用しないと問題が発生することがあります。たとえば、検索条件を満たす行がないためにDELETE文がNOT FOUNDを設定すると、次のコードは無限ループに陥ります。

```
/* improper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
}

no_more:
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
     ...
```

次の例では、GOTOのターゲットを再設定することで、NOT FOUND条件を適切に処理します。

```
/* proper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
}
no_more:
    EXEC SQL WHENEVER NOT FOUND GOTO no_match;
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
    ...
```

```
no_match:
    ...
```

9.8.6.4 アドレス指定可能度の維持について

WHENEVER GOTOディレクティブによって制御されるすべてのSQL文が、必ずGOTOラベルに分岐するようにしてください。次のコードは、*func1*内の*labelA*が*func2*内のINSERT文のスコープ内にないため、コンパイル時エラーが発生します。

```
func1()
{

    EXEC SQL WHENEVER SQLERROR GOTO labelA;
    EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
    ...
labelA:
...
}
func2()
{

    EXEC SQL INSERT INTO emp (job) VALUES (:job_title);
    ...
}
```

WHENEVER GOTOディレクティブの分岐先のラベルは、この文と同じプリコンパイル・ファイル内にする必要があります。

9.8.6.5 エラー後の戻りについて

エラー処理後にプログラムに戻る必要がある場合は、DO *routine_call*アクションを使用します。または、次の例に示すように、*sqlcode*の値をテストしてもかまいません。

```
...
EXEC SQL UPDATE emp SET sal = sal * 1.10;
if (sqlca.sqlcode < 0)
{  /* handle error  */

EXEC SQL DROP INDEX emp_index;
```

アクティブなWHENEVER GOTOディレクティブまたはWHENEVER STOPディレクティブがないことを確認してください。

9.9 SQL文のテキスト取得について

多くのプリコンパイラ・アプリケーションでは、処理中の文のテキスト、その長さ、指定されているSQLコマンド(INSERTやSELECTなど)を把握すると役に立ちます。これは、動的SQLを使用するアプリケーションについて特に当てはまります。

SQLStmtGetText()関数(旧称はsqlgls()関数)は、SQLLIBランタイム・ライブラリの一部であり、次の情報を戻します。

- 最後に解析されたSQL文のテキスト
- 文の有効な長さ
- 文で使用されているSQLコマンドの関数コード

SQLStmtGetText()はスレッド・セーフです。静的SQL文を発行した後にSQLStmtGetText()関数をコールすることができます。動的SQL方法1のときは、SQL文が実行された後にSQLStmtGetText()をコールします。動的SQL方法2、3および4のときは、文がPREPAREするとすぐにSQLStmtGetText()をコールすることができます。

すべてのSQLLIB関数の新しい名前については、SQLLIBパブリック関数の新しい名前を参照してください。

SQLStmtGetText()のプロトタイプは、次のとおりです。

```
void SQLStmtGetText(dvoid *context, char *sqlstm, size_t *stmlen, size_t *sqlfc);
```

コンテキスト・パラメータはランタイム・コンテキストです。コンテキストの定義と使用方法は、コンテキスト変数を参照してください。

*sqlstm*パラメータは文字バッファです。このバッファには、SQL文の戻されたテキストが格納されます。プログラムでは、静的にバッファを宣言するか、動的にバッファのメモリーを割り当てる必要があります。

*stmlen*パラメータはsize_t変数です。SQLStmtGetText()をコールする前に、このパラメータに*sqlstm*バッファの実際のサイズをバイト単位で設定してください。SQLStmtGetText()が戻ると、*sqlstm*バッファにはSQL文テキストが入り、その後はバッファ長まで空白文字で埋められます。*stmlen*パラメータは、戻された文の実際のバイト数を戻します。埋め込まれた空白文字のバイト数は含まれません。*stmlen*の最大値はポート固有で、通常は最大整数サイズになります。

*sqlfc*パラメータは、文のSQLコマンドのSQL機能コードを戻すsize_t変数です。表9-3は、各コマンドのSQL機能コードを示しています。

表9-3 SQL機能コード

コード	SQL機能	コード	SQL機能	コード	SQL機能
01	CREATE TABLE	26	ALTER TABLE	51	DROP TABLESPACE
02	SET ROLE	27	EXPLAIN	52	ALTER SESSION
03	INSERT	28	GRANT	53	ALTER USER
04	SELECT	29	REVOKE	54	COMMIT
05	UPDATE	30	CREATE SYNONYM	55	ROLLBACK
06	DROP ROLE	31	DROP SYNONYM	56	SAVEPOINT
07	DROP VIEW	32	ALTER SYSTEM SWITCH LOG	57	CREATE CONTROL FILE
08	DROP TABLE	33	SET TRANSACTION	58	ALTER TRACING
09	DELETE	34	PL/SQL EXECUTE	59	CREATE TRIGGER
10	CREATE VIEW	35	LOCK TABLE	60	ALTER TRIGGER
11	DROP USER	36	(使用されていない)	61	DROP TRIGGER
12	CREATE ROLE	37	RENAME	62	ANALYZE TABLE

コード	SQL機能	コード	SQL機能	コード	SQL機能
13	CREATE SEQUENCE	38	COMMENT	63	ANALYZE INDEX
14	ALTER SEQUENCE	39	AUDIT	64	ANALYZE CLUSTER
15	(使用されていない)	40	NOAUDIT	65	CREATE PROFILE
16	DROP SEQUENCE	41	ALTER INDEX	66	DROP PROFILE
17	CREATE SCHEMA	42	CREATE EXTERNAL DATABASE	67	ALTER PROFILE
18	CREATE CLUSTER	43	DROP EXTERNAL DATABASE	68	DROP PROCEDURE
19	CREATE USER	44	CREATE DATABASE	69	(使用されていない)
20	CREATE INDEX	45	ALTER DATABASE	70	ALTER RESOURCE COST
21	DROP INDEX	46	CREATE ROLLBACK SEGMENT	71	CREATE SNAPSHOT LOG
22	DROP CLUSTER	47	ALTER ROLLBACK SEGMENT	72	ALTER SNAPSHOT LOG
23	VALIDATE INDEX	48	DROP ROLLBACK SEGMENT	73	DROP SNAPSHOT LOG
24	CREATE PROCEDURE	49	CREATE TABLESPACE	74	CREATE SNAPSHOT
25	ALTER PROCEDURE	50	ALTER TABLESPACE	75	ALTER SNAPSHOT
--	--	--	--	76	DROP SNAPSHOT

エラーが発生すると、長さパラメータ(*stmlen*)はゼロを戻します。発生する可能性のあるエラー条件は、次のとおりです。

- SQL文が解析されていません。
- 無効なパラメータ(たとえば、負の長さのパラメータ)を渡しました。
- SQLLIBで内部例外が発生しました。

9.9.1 制限(SQLStmtGetText()の使用)

SQLStmtGetText()は、次のコマンドを含む文のテキストは戻しません。

- CONNECT
- COMMIT
- ROLLBACK
- FETCH

これらのコマンドのSQL機能コードはありません。

9.9.2 サンプル・プログラム

サンプル・プログラムsqlvcp.pcは、demoディレクトリにあります。このプログラムは、sqlgls()関数の使用方法を示します。

関連項目

- データ型とホスト変数

9.10 Oracle通信領域(ORACA)の使用について

SQLCAが標準的なSQL通信を処理するのに対し、ORACAはOracle通信を処理します。ランタイム・エラーおよび状態の変化について、SQLCAで提供されるより詳しい情報が必要な場合は、ORACAを使用してください。これには、豊富な診断ツールが用意されています。ただし、ORACAの使用はランタイム・オーバーヘッドを増加させるため、あくまでもオプションです。

ORACAは問題の診断に役立つ上に、プログラムによるOracleリソース(SQL文エグゼキュータやカーソル・キャッシュなど)の利用を監視できます。

プログラムでは複数のORACAを使用できます。たとえば、1つのグローバルORACAと複数のローカルORACAを設定できます。ローカルORACAへのアクセスは、プログラム内のその有効範囲によって制限されます。Oracleでは適用範囲内のORACAにのみ情報が戻されます。

9.10.1 ORACAの宣言について

ORACAを宣言するには、次に示すように、INCLUDE文または**#include**プリプロセッサ・ディレクティブを使用してORACAを自分のプログラムにコピーします。

```
EXEC SQL INCLUDE ORACA;
```

または

```
#include <oraca.h>
```

ORACAが**extern**記憶域クラスであることが必要な場合は、プログラムに次のようにORACA_STORAGE_CLASSを定義します。

```
#define ORACA_STORAGE_CLASS extern
```

プログラムで宣言部を使用するときは、ORACAを宣言部の外側で定義する必要があります。

9.10.2 ORACAの有効化について

ORACAを有効にするには、コマンドラインに次のようにORACAオプションを指定する必要があります。

```
ORACA=YES
```

またはインラインで次のように指定します。

```
EXEC ORACLE OPTION (ORACA=YES);
```

その後、ORACA内のフラグを設定することによって、適切なランタイム・オプションを選択する必要があります。

9.10.3 ORACAの内容

ORACAには、次のように、オプションの設定、システムの統計および高度な診断情報が保存されています。

- SQL文のテキスト(テキストの保存時に指定できます)
- エラーが発生したファイルの名称(サブルーチンの使用時に便利です)
- ファイル内のエラーの位置
- カーソル・キャッシュのエラーおよび統計情報

oraca.hの一部を次に示します。

```
/*
NAME
  ORACA : Oracle Communications Area.

  If the symbol ORACA_NONE is defined, then there will be no ORACA
  *variable*, although there will still be a struct defined.  This
  macro should not normally be defined in application code.

  If the symbol ORACA_INIT is defined, then the ORACA will be
  statically initialized. Although this is not necessary in order
  to use the ORACA, it is a good pgming practice not to have
  unitialized variables. However, some C compilers/operating systems
  don't allow automatic variables to be init'd in this manner. Therefore,
  if you are INCLUDE'ing the ORACA in a place where it would be
  an automatic AND your C compiler/operating system doesn't allow this style
  of initialization, then ORACA_INIT should be left undefined --
  all others can define ORACA_INIT if they wish.
*/

#ifndef  ORACA
#define  ORACA      1

struct    oraca
{
    char oracaid[8];   /* Reserved              */
    long oracabc;      /* Reserved              */

/*    Flags which are setable by User. */

   long  oracchf;      /* <> 0 if "check cur cache consistncy"*/
   long  oradbgf;      /* <> 0 if "do DEBUG mode checking"    */
   long  orahchf;      /* <> 0 if "do Heap consistency check" */
   long  orastxtf;     /* SQL stmt text flag            */
#define  ORASTFNON 0   /* = don't save text of SQL stmt       */
#define  ORASTFERR 1   /* = only save on SQLERROR         */
```

```
#define  ORASTFWRN 2   /* = only save on SQLWARNING/SQLERROR  */
#define  ORASTFANY 3      /* = always save             */
    struct
      {
  unsigned short orastxtl;
  char  orastxtc[70];
      } orastxt;         /* text of last SQL stmt          */
    struct
      {
  unsigned short orasfnml;
  char      orasfnmc[70];
      } orasfnm;        /* name of file containing SQL stmt    */
  long   oraslnr;        /* line nr-within-file of SQL stmt     */
  long   orahoc;         /* highest max open OraCurs requested  */
  long   oramoc;         /* max open OraCursors required         */
  long   oracoc;         /* current OraCursors open         */
  long   oranor;         /* nr of OraCursor re-assignments      */
  long   oranpr;         /* nr of parses               */
  long   oranex;         /* nr of executes            */
    };

#ifndef ORACA_NONE

#ifdef ORACA_STORAGE_CLASS
ORACA_STORAGE_CLASS struct oraca oraca
#else
struct oraca oraca
#endif
#ifdef ORACA_INIT
    =
    {
    {'O','R','A','C','A',' ',' ',' '},
    sizeof(struct oraca),
    0,0,0,0,
    {0,{0}},
    {0,{0}},
    0,
    0,0,0,0,0,0
    }
#endif
    ;

#endif

#endif
/* end oraca.h */
```

9.10.4 ランタイム・オプションの選択について

ORACAにはいくつかのオプション・フラグがあります。これらのフラグに0(ゼロ)以外の値を設定することで、次のことが可能になります。

- SQL文のテキストの保存
- DEBUG処理の有効化
- カーソル・キャッシュの一貫性チェック(カーソル・キャッシュとは、カーソル管理に使用されるメモリーで継続的に更新される領域)

- ヒープの一貫性チェック(ヒープは、動的変数のために予約されるメモリー領域)
- カーソルの統計情報の収集

次の説明はオプションを選択するときの参考になります。

9.10.5 ORACAの構造体

この項では、ORACAの構造体とそのコンポーネントおよび格納できる値について説明します。

9.10.5.1 *oracaid*

この文字列コンポーネントは「ORACA」に初期化されて、Oracle通信領域を示します。

9.10.5.2 *oracabc*

この整数コンポーネントには、ORACAデータ構造体の長さがバイト単位で格納されています。

9.10.5.3 *oracchf*

マスターDEBUGフラグ(*oradbgf*)が設定されていると、このフラグによってカーソル・キャッシュの統計情報の収集と、各カーソル操作前のカーソル・キャッシュの一貫性チェックができます。

Oracleランタイム・ライブラリでは一貫性チェックが行われ、エラー・メッセージが発行されることがあります(エラー・メッセージについては、『*Oracle Database*エラー・メッセージ』を参照してください)。これらは、Oracleエラー・メッセージと同様にSQLCAに戻されます。

このフラグは次のいずれかを設定します。

- キャッシュ一貫性チェックを使用禁止にします(デフォルト)。
- キャッシュ一貫性チェックを使用可能にします。

9.10.5.4 *oradbgf*

このマスター・フラグを使用すると、DEBUGオプションをすべて選択できます。これには次の設定があります。

すべてのDEBUG処理を使用禁止にします(デフォルト)。

すべてのDEBUG処理を有効にします。

9.10.5.5 *orahchf*

マスターDEBUGフラグ(*oradbgf*)が設定されていると、プリコンパイラによって動的にメモリーが割り当てまたは解放されるたびに、Oracleランタイム・ライブラリでヒープの一貫性がチェックされます。これはメモリー障害を起こすプログラムの不具合を検出するのに役立ちます。

このフラグはCONNECTコマンドを発行する前に設定する必要があります。また、このフラグは一度設定すると解除できなくなります。つまり、設定後にこのフラグの変更要求があっても無視されます。これには次の設定があります。

- ヒープ一貫性チェックを無効にします(デフォルト)。
- ヒープ一貫性チェックを有効にします。

9.10.5.6 *orastxtf*

このフラグを使用すると、現行のSQL文のテキストを保存するタイミングを指定できます。これには次の設定があります。

- SQL文のテキストを保存しません(デフォルト)。

- SQLERRORのSQL文のテキストのみ保存します。
- SQLERRORまたはSQLWARNINGのSQL文のテキストのみ保存します。
- 常にSQL文のテキストを保存します。

SQL文のテキストは、*orastxt*という名前のORACA埋込み構造体に保存されます。

9.10.5.7 診断情報

ORACAは高度な診断情報を提供します。次の変数によってエラーの位置をすばやく特定できます。

9.10.5.8 *orastxt*

この埋込み構造体は、問題のあるSQL文を見つけるために使用します。Oracleで解析された最後のSQL文のテキストを保存できます。これには次の2つのコンポーネントが格納されています。

コンポーネント	説明
orastxtl	この整数コンポーネントにはカレント SQL 文の長さが格納されます。
orastxtc	この文字列コンポーネントにはカレント SQL 文のテキストが格納されます。先頭から最大 70 文字までのテキストが保存されます。文字列はヌル文字で終了しません。文字列を印刷するときは、oratxtl 長さコンポーネントを使用します。

プリコンパイラによって解析された文(CONNECT、FETCHおよびCOMMITなど)は、ORACAには保存されません。

9.10.5.9 *orasfnm*

この埋込み構造体は、カレントSQL文が含まれているファイルを識別します。このため、1つのアプリケーション用に複数のファイルをプリコンパイルするときにエラーを検出できます。これには次の2つのコンポーネントが格納されています。

コンポーネント	説明
orasfnml	この整数コンポーネントには、orasfnmc に保存されているファイル名の長さが格納されます。
orasfnmc	この文字列コンポーネントにはファイル名が格納されます。先頭から最大 70 文字が格納されます。

9.10.5.10 *oraslnr*

この整数コンポーネントはカレントSQL文がある行またはその付近の行を識別します。

9.10.5.11 カーソル・キャッシュ統計情報

マスターDEBUGフラグ(*oradbgf*)およびカーソル・キャッシュ・フラグ(*oracchf*)が設定されているときは、次の変数を使用してカーソル・キャッシュ統計情報を収集できます。これらの変数は、プログラムがCOMMITコマンドまたはROLLBACKコマンドを発行するたびに自動的に設定されます。

内部的には、CONNECTされているデータベース別にこの変数のセットがあります。ORACA内の現在の設定値は、最後にCOMMITまたはROLLBACKが行われたデータベースに関係します。

9.10.5.12 *orahoc*

この整数コンポーネントは、プログラムの実行中にMAXOPENCURSORSに設定された最大値を記録します。

9.10.5.13 *oramoc*

この整数コンポーネントには、プログラムの要求によってオープンされたOracleカーソルの最大数が記録されます。MAXOPENCURSORSに設定されている値が小さすぎて、その結果プリコンパイラによってカーソル・キャッシュが拡張されると、この数は*orahoc*より大きくなることがあります。

9.10.5.14 *oracoc*

この整数コンポーネントは、プログラムの要求によってオープンされているOracleカーソルのカレント数を記録します。

9.10.5.15 *oranor*

この整数コンポーネントには、プログラムの要求によって再度割り当てられたカーソル・キャッシュの数が記録されます。この数値は、カーソル・キャッシュのスラッシングの程度を示すもので、できるだけ小さく保つ必要があります。

9.10.5.16 *oranpr*

この整数コンポーネントは、プログラムの要求によって解析されたSQL文の数を記録します。

9.10.5.17 *oranex*

この整数コンポーネントは、プログラムの要求によって実行されたSQL文の数を記録します。この数値の*oranpr*に対する割合は、できるかぎり高く保ってください。つまり、不要な再解析は回避する必要があります。

関連項目

- パフォーマンス・チューニング

9.10.6 ORACAの例

次のプログラムは部門番号の入力を要求し、その部内の各従業員の名前および給与を2つの表のどちらかに挿入してから、ORACAからの診断情報を表示します。このプログラムはoraca.pcとしてdemoディレクトリにあり、オンラインで利用できます。

```
/* oraca.pc
 * This sample program demonstrates how to
 * use the ORACA to determine various performance
 * parameters at runtime.
 */
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#include <oraca.h>

EXEC SQL BEGIN DECLARE SECTION;
char *userid = "SCOTT/TIGER";
char  emp_name[21];
int   dept_number;
float salary;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

void sql_error();

main()
{
```

```
    char temp_buf[32];

    EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");
    EXEC SQL CONNECT :userid;

    EXEC ORACLE OPTION (ORACA=YES);

    oraca.oradbgf  = 1;             /* enable debug operations */
    oraca.oracchf  = 1;      /* gather cursor cache statistics */
    oraca.orastxtf = 3;       /* always save the SQL statement */

    printf("Enter department number: ");
    gets(temp_buf);
    dept_number = atoi(temp_buf);


    EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT ename, sal + NVL(comm,0) AS sal_comm
        FROM emp
        WHERE deptno = :dept_number
        ORDER BY sal_comm DESC;
    EXEC SQL OPEN emp_cursor;
    EXEC SQL WHENEVER NOT FOUND DO sql_error("End of data");

    for (;;)
    {
        EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
        printf("%.10s¥n", emp_name);
        if (salary < 2500)
            EXEC SQL INSERT INTO pay1 VALUES (:emp_name, :salary);
        else
            EXEC SQL INSERT INTO pay2 VALUES (:emp_name, :salary);
    }
}

void
sql_error(errmsg)
char *errmsg;
{
    char buf[6];

    strcpy(buf, SQLSTATE);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;

    if (strncmp(errmsg, "Oracle error", 12) == 0)
        printf("¥n%s, sqlstate is %s¥n¥n", errmsg, buf);
    else
        printf("¥n%s¥n¥n", errmsg);

    printf("Last SQL statement: %.*s¥n",
    oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("¥nAt or near line number %d¥n", oraca.oraslnr);
    printf
("¥nCursor Cache Statistics¥n----------------------¥n");
    printf
("Maximum value of MAXOPENCURSORS:    %d¥n", oraca.orahoc);
    printf
("Maximum open cursors required:      %d¥n", oraca.oramoc);
    printf
```

```
("Current number of open cursors:     %d¥n", oraca.oracoc);
    printf
("Number of cache reassignments:      %d¥n", oraca.oranor);
    printf
("Number of SQL statement parses:     %d¥n", oraca.oranpr);
    printf
("Number of SQL statement executions: %d¥n", oraca.oranex);
    exit(1);
}
```

10 プリコンパイラのオプション

この章では、Pro*C/C++プリコンパイラの実行方法とプリコンパイラ・オプションの拡張セットについて詳しく説明します。この章のトピックは、次のとおりです:

- プリコンパイラのコマンド
- プリコンパイラ・オプション
- クイック・リファレンス
- オプションの入力
- プリコンパイラ・オプションの使用方法について

10.1 プリコンパイラのコマンド

プリコンパイラの場所はシステムごとに異なります。システム管理者またはデータベース管理者は、通常は論理名またはエイリアスを指定するか、その他のシステム固有の手段を使用して、Pro*C/C++実行ファイルをアクセス可能にします。

Pro*C/C++プリコンパイラを実行するには、次のコマンドを入力します。

```
proc option=value...
```

注意:

オプション値は必ずオプション名に続く等号(前後のスペースなし)の後に指定します。

たとえば次のコマンドを入力するとします。

```
proc INAME=test_proc
```

すると、カレント・ディレクトリのtest_proc.pcファイルがプリコンパイルされますが、これはプリコンパイラではファイル名の拡張子がpcとみなされるためです。INAME=引数には、プリコンパイル対象のソースファイルを指定します。INAMEオプションはコマンドラインの最初のオプションでなくてもかまいませんが、最初にくる場合はオプション指定を省略できます。したがって、次のコマンド

```
proc myfile
```

は、次のオプションに相当します。

```
proc INAME=myfile
```

注意:

特定の OS オブジェクト(たとえばファイル名)の名前を指定しないオプション名とオプション値には、大/小文字区別はありません。このマニュアル中の例では、オプション名は大文字で記述し、オプション値は通常は小文字で記述しています。Pro*C/C++プリコンパイラ実行ファイル自体も含めてファイル名を入力するときには、大/小文字区別に関してオペレーティング・システムの表記規則に従ってください。

UNIX など一部のプラットフォームでは、値の文字列の前に特定のエスケープ文字が必要です。プラットフォーム固有のマニュアルを参照してください。

10.1.1 大/小文字の区別

一般的に、プリコンパイラ・オプションの名前および値には、大文字と小文字のどちらを指定してもかまいません。ただし、UNIXのように大/小文字を区別するオペレーティング・システムの場合は、大文字および小文字を正しく組み合せて、Pro*C/C++実行ファイルの名前を含むファイル名を指定してください。

10.2 プリコンパイラのオプション

各オプションを使用すると、リソースの使用方法、エラーのレポート方法、入出力のフォーマット方法およびカーソルの管理方法を制御できます。

オプションの値はリテラルで、テキスト値または数値を表します。たとえば、次のオプションを指定するとします。

```
...   INAME=my_test
```

この値はファイル名を指定する文字列リテラルです。

次にMAXOPENCURSORSオプションの例を示します。

```
...MAXOPENCURSORS=20
```

この値は数値です。

一部のオプションはブール値をとり、文字列*yes*または*no*、*true*または*false*、あるいは整数リテラル1または0で表すことができます。たとえば、次のオプションを指定するとします。

```
...   SELECT_ERROR=yes
```

は、次のオプションに相当します。

```
...   SELECT_ERROR=true
```

または次のオプションに相当します。

```
...   SELECT_ERROR=1
```

これらはすべて、実行時にSELECTエラーが検出されることを意味しています。

10.2.1 環境変数

SYS_INCLUDEプリコンパイラ・オプションおよびINCLUDEプリコンパイラ・オプションでは、環境変数を使用できます。PROCアプリケーションをプリコンパイルするときは、SYS_INCLUDEディレクトリ・パスおよびINCLUDEディレクトリ・パスでORACLE_HOMEなどの環境変数を使用できます。また、CONFIGファイルpcscfg.cfgのSYS_INCLUDEオプション値およびINCLUDEオプション値を使用することもできます。環境変数の次の使用方法がサポートされています。

Linuxの場合

```
$ENV_VAR
sys_include=$ORACLE_HOME/precomp/public
include=$ORACLE_HOME/precomp/public
```

```
$(ENV_VAR)
sys_include=$(ORACLE_HOME)/precomp/public
include=$(ORACLE_HOME)/precomp/public

${ENV_VAR}
sys_include=${ORACLE_HOME}/precomp/public
include=${ORACLE_HOME}/precomp/public
```

Windowsの場合

```
%ENV_VAR%
sys_include=%ORACLE_HOME%¥precomp¥public
include=%ORACLE_HOME%¥precomp¥public
```

10.2.2 構成ファイル

構成ファイルは、プリコンパイラ・オプションを格納するテキスト・ファイルです。ファイル内の各レコード(行)には、オプション1つと、それに対応付けられた1つ以上の値が含まれます。*1*行に複数のオプションを入力すると、*2*番目以降のオプションは無視されます。たとえば、次の構成ファイルに次の行が含まれる場合があります。

```
FIPS=YES
MODE=ANSI
CODE=ANSI_C
```

これらの行は、FIPS、MODEおよびCODEオプションにデフォルト値を設定します。

pscfg.cfgでは、各エントリの1行当たりの文字数は最大300文字に制限されています。SYS_INCLUDEパスなど、この値を300文字より長く設定するには、エントリを複数行で作成します。次に例を示します。

```
sys_include=/ade/aime_rdbms_9819/oracle/precomp/public
sys_include=/usr/include,/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux7/2.96/include
sys_include=/usr/include
```

行末にはカッコを使用しないでください。行末にカッコを使用すると、前の行がすべて無効になります。たとえば、次のように3行目の行末にカッコがあると、次のエントリは、

```
sys_include=/ade/aime_rdbms_9819/oracle/precomp/public
sys_include=/usr/include,/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include)
sys_include=/usr/lib/gcc-lib/i386-redhat-linux7/2.96/include
sys_include=/usr/include
```

SYS_INCLUDEが次のように設定されます。

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include,  /usr/lib/gcc-lib/i386-redhat-
linux7/2.96/include,/usr/include
```

インストールごとにシステム構成ファイルが1つあります。システム構成ファイルの名前はpscfg.cfgです。このファイルの位置はシステムに依存します。

注意:

pscfg.cfg には、include や LIBPATH などの変数のデフォルトのパス設定が含まれています。これらのパスは、コ

ンピュータまたはオペレーティング・システムに依存します。デフォルトのパスが、ご使用のコンピュータおよびオペレーティング・システムで有効なことを確認する必要があります。有効でない場合は、そのパスを 8dot3 表記に置き換えてください。

pcscfg.cfg ファイルでは、空白を使用できません。たとえば、次のファイルに次の行が含まれるとします。

```
include="D:¥Program Files¥Microsoft Visual Studio¥VC98¥include"
```

プリコンパイルは失敗します。次の行に置き換えることができます。

```
include=D:¥Progra~1¥Microa~4¥VC98¥include
```

Pro*C/C++のユーザーはそれぞれ、1つ以上のプライベート構成ファイルを持つことができます。構成ファイルの名前は、必ずCONFIG=プリコンパイラ・オプションを使用して指定してください。

注意:

構成ファイルはネストできません。つまり、構成ファイル内では、CONFIG=は有効ではありません。

関連項目

- プリコンパイラ・オプションの使用について

10.2.3 オプション値の優先順位

オプションの値は、優先順位の低いものから順に、次のように決定されます。

- プリコンパイラに組み込まれた値
- Pro*C/C++システム構成ファイル内の値の集合
- Pro*C/C++ユーザー構成ファイル内の値の集合
- コマンドラインで設定される値
- インラインで設定される値

たとえば、MAXOPENCURSORSオプションでは、キャッシュ内のオープン・カーソルの最大数を指定します。このオプションのプリコンパイラに組み込まれたデフォルト値は10です。ただし、システム構成ファイルでMAXOPENCURSORS=32と指定されていると、デフォルトは32になります。ユーザー構成ファイルでは、これをさらに別の値に設定でき、これはシステム構成ファイルの値より優先されます。最終的には、インライン指定が前述のすべてのデフォルト値よりも優先されます。

PROCコマンドライン・オプションが複数回使用されている場合、PROCでは、最後に使用されているPROCコマンドライン・オプションに割り当てられている最後の値をプリコンパイルに使用します。次に例を示します。

```
$ proc iname=sample.pc ... oname=output1.c ... oname=output2.c ... oname=output3.c
```

この例では、output3.cはPROCが使用するONAME値であり、生成される出力ファイル名はoutput3.cです。

CONFIGファイル(システム・デフォルトまたはユーザー定義)とコマンドラインの両方にオプションが指定されている場合は、コマン

ドラインで指定されている値が優先されます。

SYS_INCLUDEオプションおよびINCLUDEオプションの場合、動作は環境変数で定義されます。行末にはカッコで中断されないかぎり、値は追加されます。

プライベート構成ファイルをCONFIG=*filename*で指定する場合、最初の値が優先され、コマンドラインの後続の値は無視されます。この場合は、例外的にコマンドラインの最後の値が優先されません。

USERIDなどの一部のオプションには、プリコンパイラ・デフォルト値がありません。オプションで組込みデフォルト値のあるものについては、表10-2を参照してください。

> **注意:**
>
> プリコンパイラのデフォルト値については、システム固有のマニュアルを参照してください。プラットフォーム上では、この章で示された値から変更されている可能性もあります。

現在の設定値の確認

コマンドラインで疑問符(?)を使用すると、複数のオプションのカレント値を対話形式で調べることができます。たとえば、次のコマンドを発行したとします。

```
proc ?
```

この場合、すべてのオプションが、現在の設定値とともに端末に出力(表示)されます。(UNIXシステムでCシェルを使用しているときには「?」をバックスラッシュでエスケープしてください。)この場合、値はプリコンパイラに組み込まれているもので、システム構成ファイルに値があれば、それが優先されます。ただし、次のコマンド

```
proc config=my_config_file.h ?
```

を入力したときに、カレント・ディレクトリにmy_config_file.hというファイルがあると、すべてのオプションがリスト形式で表示されます。ユーザー構成ファイルの値によって、不足している値が補われ、Pro*C/Cプリコンパイラに組み込まれている値またはシステム構成ファイルに指定されている値が置き換えられます。

オプション名を指定し、その後ろに=?を付けるだけで、どれか1つのオプションの現在の設定値を調べることもできます。次に例を示します。

```
proc maxopencursors=?
```

このように入力すると、MAXOPENCURSORSオプションの現在のデフォルト値が出力されます。

次のように入力するとします。

```
proc
```

すると、表10-2のような短いサマリーが表示されます。

関連項目

- プリコンパイル中の状況
- プリコンパイラ・オプションの使用について

10.2.4 マクロ・オプションおよびマイクロ・オプション

MODEオプションは複数のオプションを同時に制御します。MODEはマクロ・オプションとも呼ばれます。CLOSE_ON_COMMIT、DYNAMICおよびTYPE_CODEなどのより新しいオプションは1つの関数のみを制御し、マイクロ・オプションとして知られています。マクロ・オプションは、高い優先順位が付けられている場合にのみ、マイクロ・オプションより優先されます。

次の表は、マクロ・オプション値によって設定されるマイクロ・オプションの値を示しています。

表10-1 マクロ・オプション値によりマイクロ・オプション値が設定される方法

マクロ・オプション	マイクロ・オプション
MODE=ANSI | ISO	CLOSE_ON_COMMIT=YES DYNAMIC=ANSI TYPE_CODE=ANSI
MODE=ORACLE	CLOSE_ON_COMMIT=NO DYNAMIC=ORACLE TYPE_CODE=ORACLE

ユーザー構成ファイルでMODE=ANSIとCLOSE_ON_COMMIT=NOの両方を指定すると、COMMITしてもカーソルはクローズしません。構成ファイルでMODE=ORACLEを指定し、コマンドラインでCLOSE_ON_COMMIT=YESを指定すると、カーソルはクローズします。

関連項目

- オプション値の優先順位

10.2.5 プリコンパイル中の状況

プリコンパイル時に、ホスト・プログラムに埋め込まれているSQL文は、Pro*C/C++が生成するCまたはC++のコードに置換されます。生成されたコードには、データ型、データ長、ホスト変数のアドレスを示すデータ構造や、ランタイム・ライブラリであるSQLLIBに必要なその他の情報も含まれています。このコードには、埋込みSQLの動作を実行するSQLLIBルーチンに対するコールも含まれています。

注意:

プリコンパイラでは、Oracle Call Interface(OCI)に対するコールは生成されません。

表10-2は主なプリコンパイラ・オプションのクイック・リファレンスです。受入れ可能でも効力を持たないオプションは、この表には記載されていません。

関連項目

- Oracle Databaseエラー・メッセージ・リファレンス
- プリコンパイラ・オプションの使用について

10.2.6 オプションの適用範囲

プリコンパイル・ユニットは、Cコードと1つ以上の埋込みSQL文を含むファイルです。特定のプリコンパイル・ユニットに対して指定したオプションは、そのプリコンパイル・ユニットにのみ効力を持ちます。たとえば、ユニットAに対してHOLD_CURSOR=YESおよびRELEASE_CURSOR=YESを指定し、ユニットBには指定しなければ、ユニットAのSQL文はこれらのHOLD_CURSOR値およびRELEASE_CURSOR値を使用して実行されますが、ユニットBのSQL文はデフォルト値を使用して実行されます。

10.2.7 Windowsプラットフォーム用Pro*C/C++プリコンパイラの問題

この項では、Windowsプラットフォーム用Pro*C/C++関連の問題について説明します。

10.2.7.1 構成ファイル

このリリースでは、システム構成ファイルをpcscfg.cfgと呼びます。このファイルは、*ORACLE_HOME*¥precomp¥adminディレクトリにあります。

10.2.7.2 CODE

CODEオプションのデフォルト設定はANSI_Cです。他のオペレーティング・システム用のPro*C/C++では、デフォルト設定がKR_Cになっている場合があります。

10.2.7.3 DBMS

DBMS=V6_CHARは、CHAR_MAP=VARCHAR2使用時にサポートされません。かわりに、DBMS=V7を使用します。

10.2.7.4 INCLUDE

PARSE=PARTIALまたはPARSE=FULLでプリコンパイルするサンプル・プログラムの場合、c:¥program files¥devstudio¥vc¥includeのインクルード・パスが追加されています。Microsoft Visual Studioが別の場所にインストールされている場合、サンプル・プログラムで正しくプリコンパイルするために、「インクルード・ディレクトリ」フィールドを適宜変更します。

10.2.7.5 PARSE

PARSEオプションのデフォルト設定はNONEです。他のオペレーティング・システム用のPro*C/C++では、デフォルト設定がFULLになっている場合があります。

10.3 クイック・リファレンス

表10-2は、Pro*C/C++オプションのクイック・リファレンスです。アスタリスクでマークされたオプションはインラインで入力できます。

表10-2 プリコンパイラのオプション

構文	デフォルト値	指定
AUTO_CONNECT={YES | NO}	NO	最初の実行文の前の自動 CLUSTER$アカウント接続。
CHAR_MAP={VARCHAR2 | CHARZ |	CHARZ	文字配列および文字列のマッピング。

構文	デフォルト値	指定
STRING | CHARF} *		
CINCR	1	現在の物理接続数が CMAX 値より少ない場合、データベースに対してオープンされる物理接続数の次の増分をアプリケーションで設定できるようにします。
CLOSE_ON_COMMIT={YES | NO}	NO	COMMIT 時にすべてのカーソルをクローズします。
CODE={ANSI_C | KR_C | CPP}	KR_C	生成される C コードの種類。
COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}	MULTI_BYTE	C/C++コンパイラがサポートするキャラクタ・セットの型。
CONFIG=*filename*	なし	ユーザーのプライベート構成ファイル。
CMIN	2	接続プール内の最小物理接続数を指定します。
CMAX	100	データベースに対してオープンできる物理接続の最大数を指定します。
CNOWAIT	未設定を意味する 0。	この属性は、プール内の他のすべての物理接続が使用中で、物理接続の合計数がすでに最大値に達している場合に、アプリケーションで繰り返し物理接続を要求する必要があるかどうかを決定します。
CPOOL	NO	このオプションに基づき、プリコンパイラでは、SQLLIB に接続プール機能を有効または無効にするように指示する適切なコードを生成します。
CPP_SUFFIX=*extension*	なし	出力ファイルのデフォルトのファイルの拡張子を指定します。
CTIMEOUT	未設定を意味する 0。	指定された期間(秒単位)より長い間アイドル状態になっている物理接続を終了し、オープンされている物理接続数を最適に保ちます。

構文	デフォルト値	指定
DBMS={V7 | NATIVE | V8}	NATIVE	互換性(Oracle7、Oracle8、Oracle8*i*、Oracle9*i* またはプリコンパイル時に接続されていたデータベースのバージョン)。
DEF_SQLCODE={YES | NO}	NO	#define SQLCODE に対するマクロを生成します。
DEFINE=*name* *	なし	Pro*C/C++プリコンパイラで使用する名前を定義します。
DURATION={TRANSACTION | SESSION}	TRANSACTION	キャッシュ内のオブジェクトの確保継続時間を設定します。
DYNAMIC={ANSI | ORACLE}	ORACLE	Oracle または ANSI SQL の意味を指定します。
ERRORS={YES | NO}	YES	エラー・メッセージの送り先(NO を指定すると、リスト・ファイルにのみ送られ、端末には送られません)。
ERRTYPE=*filename*	なし	intype ファイル・エラー・メッセージのリスト・ファイル名。
FIPS={NO | SQL89 | SQL2 | YES} *	なし	ANSI/ISO 非準拠を切り替えるかどうか。
HEADER=*extension*	なし	プリコンパイルされたヘッダー・ファイルのファイル拡張子。
HOLD_CURSOR={YES | NO} *	NO	カーソル・キャッシュが SQL 文を処理する方法。
INAME=]*filename*	なし	入力ファイルの名前。
INCLUDE=*pathname* *	なし	EXEC SQL INCLUDE 文または#include 文のディレクトリ・パス。
INTYPE=*filename*	なし	型情報の入力ファイル名。
LINES={YES | NO}	NO	#line ディレクティブを生成するかどうか。

構文	デフォルト値	指定
LNAME=*filename*	なし	リスト・ファイルの名前。
LTYPE={NONE | SHORT | LONG}	なし	生成するリスト・ファイルの型(生成する場合)。
MAXLITERAL=10..1024	1024	生成される C コードの文字列リテラルの最大長(バイト)。
MAXOPENCURSORS=5 から 255 *	10	同時にキャッシュされるオープン・カーソルの最大数。
MODE={ANSI | ISO | ORACLE}	ORACLE	ANSI/ISO または Oracle の動作。
NATIVE_TYPES	NO	ネイティブ float/double をサポートします。
NLS_CHAR=(*var1*,..., *varn*)	なし	マルチバイトの文字変数を指定します。
NLS_LOCAL={YES | NO}	NO	マルチバイト文字の意味を制御します。
OBJECTS={YES | NO}	YES	オブジェクト型をサポートします。
ONAME=]*filename*	*iname*.c	出力(コード)ファイルの名前。
ORACA={YES | NO} *	NO	ORACA を使用するかどうか。
PAGELEN=30 から 256	80	リスト・ファイルのページ長。
PARSE={NONE | PARTIAL | FULL}	FULL	Pro*C/C++で(C パーサーで).pc ソース・コードが解析されるかどうか。
PLAN_BASELINE={*module_name* |YES |NO}	NO	モジュール名を指定して SQL 計画ベースラインを作成します。
PLAN_PREFIX={*prefix_name*| none}	なし	計画名が 128 バイト以内であることを確認します。
PLAN_RUN={YES | NO}	NO	生成された SQL ファイルを実行します。
PLAN_FIXED={YES | NO}	YES	作成した計画ベースラインが固定であるか、

構文	デフォルト値	指定
		非固定であるかを指定します。
PLAN_ENABLED={YES | NO}	YES	作成される計画ベースラインを有効にします。
MEMFORPREFETCH=0 から 4294967294	なし	指定メモリーに格納する行をプリフェッチすることで問合せを高速化します。
PREFETCH=0 から 65535	1	一定数の行をプリフェッチして、問合せを高速化します。
RELEASE_CURSOR={YES | NO} *	NO	カーソル・キャッシュからのカーソルの解放を制御します。
SELECT_ERROR={YES | NO} *	YES	SELECT エラーのフラグ付け。
SQLCHECK={SEMANTICS | SYNTAX} *	SYNTAX	プリコンパイル時の SQL チェック量。
SYS_INCLUDE=pathname	なし	iostream.h などのシステム・ヘッダー・ファイルがあるディレクトリ。
THREADS={YES | NO}	NO	マルチスレッド・アプリケーションを指定します。
TYPE_CODE={ORACLE | ANSI}	ORACLE	動的 SQL の Oracle または ANSI 型コードの使用方法。
UNSAFE_NULL={YES | NO}	NO	UNSAFE_NULL=YES と指定すると ORA-01405 メッセージが使用されなくなります。
USERID=*username/password[@dbname]*	なし	username/password[@dbname]接続文字列
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}	NCHAR_CHARSET	UNICODE(UTF16)で使用されるキャラクタ・セットの書式を指定します。
VARCHAR={YES | NO}	NO	暗黙的 VARCHAR 構造体の使用を許可するかどうか。
VERSION={ANY | LATEST | RECENT} *	RECENT	どのバージョンのオブジェクトを戻すか。

10.4 オプションの入力

どのプリコンパイラ・オプションも、コマンドラインに入力できます。また、その多くは、EXEC ORACLE OPTION文を使用してプリコンパイラ・プログラムのソース・ファイルにインライン入力できます。

10.4.1 コマンドライン

プリコンパイラ・オプションをコマンドラインに入力するには、次の構文を使用します。

```
... [OPTION_NAME=value] [OPTION_NAME=value] ...
```

それぞれのオプション=値の指定は、1つ以上の空白で区切ります。たとえば、次のように入力したとします。

```
... CODE=ANSI_C MODE=ANSI
```

10.4.2 インライン

次の構文を使用してEXEC ORACLE文を記述すると、オプションをインライン入力できます。

```
EXEC ORACLE OPTION (OPTION_NAME=value);
```

たとえば、次のように記述します。

```
EXEC ORACLE OPTION (RELEASE_CURSOR=yes);
```

10.4.2.1 EXEC ORACLEの用途

EXEC ORACLE機能は、プリコンパイル中にオプション値を変更する場合に特に便利です。たとえば、HOLD_CURSORとRELEASE_CURSORを1文単位で変更する場合があります。

また、コマンドラインで入力できる文字数が、使用しているオペレーティング・システムで制限されているときは、オプションをインラインまたは構成ファイルで指定すると便利です。

関連項目:

インライン・オプションを使用して実行時パフォーマンスを最適化する方法は、パフォーマンス・チューニングを参照してください。

10.4.2.2 EXEC ORACLEのスコープ

EXEC ORACLE文は、同一オプションを指定した別のEXEC ORACLE文によってオプション指定値(テキスト)が変更されるまで有効です。次の例では、HOLD_CURSOR=NOはHOLD_CURSOR=YESが指定されるまで有効です。

```
char emp_name[20];
int  emp_number, dept_number;
float salary;

EXEC SQL WHENEVER NOT FOUND DO break;
EXEC ORACLE OPTION (HOLD_CURSOR=NO);

EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT empno, deptno FROM emp;

EXEC SQL OPEN emp_cursor;
printf(
```

```
"Employee Number  Department¥n-------------------------¥n");
for (;;)
{
   EXEC SQL FETCH emp_cursor INTO :emp_number, :dept_number;
   printf("%d¥t%d¥n", emp_number, dept_number);
}

EXEC SQL WHENEVER NOT FOUND CONTINUE;
for (;;)
{
   printf("Employee number: ");
   scanf("%d", &emp_number);
   if (emp_number == 0)
      break;
   EXEC ORACLE OPTION (HOLD_CURSOR=YES);
   EXEC SQL SELECT ename, sal
      INTO :emp_name, :salary
      FROM emp WHERE empno = :emp_number;
   printf("Salary for %s is %6.2f.¥n", emp_name, salary);
```

10.4.3 列プロパティのサポート

列プロパティは8バイトの値で戻され、各ビットが1つの列プロパティを示します。3つの列プロパティがサポートされています。

```
  +-----------------------------------------------+
  ! 32 |.............| 10 | 9 | 8 |.....| 3 | 2 | 1 |
  +-----------------------------------------------+
                                        |   |   |
                                        |   |   |-> auto-increment column
                                        |   |-> auto value always generated
                                        |-> if generated by default when null
```

列プロパティは、新しいSQLDAメンバー(sqlda->CP[])を使用して動的文から取得できます。

```
struct SQLDA {
  /* ub4    */ int        N; /* Descriptor size in number of entries        */
  ..........
  ..........
  ..........
  ..........
  /* ub2*   */ short     *Z; /* Ptr to Arr of cur lengths of ind. var. names*/
  /* ub8*   */ long long *CP; /* Ptr to Arr of column properties            */
  };
```

このメンバーはメタデータDESCRIBEの一部として更新されます。

列プロパティは、新しい関数SQLGetColProp()を使用した静的文で取得できます(この関数は最後に実行された文から列プロパティを取得します)。

```
void SQLGetColProp(
   void  *uga,     --> IN -- run time context
   text *coln,     --> IN -- column name
   ub2  *colatr,   --> IN -- column attributes
   ub8  *colprop   --> IN/OUT -- column attribute/ub8 value that holds column properties
                    )
```

SQLGetColProp()は、列属性colatrで決定された値を戻します。

- SQL_ATTR_COL_PROPERTIES: 指定した列の列プロパティを含む、8バイトの値(colprop)を戻します。

- SQL_ATTR_COL_PROPERTY_IS_IDENTITY colpropは、指定した列がID列の場合にtrueになります。
- SQL_ATTR_COL_PROPERTY_IS_GEN_ALWAYS colpropは、指定した列が常に自動増分値を生成する場合にtrueになります。
- SQL_ATTR_COL_PROPERTY_IS_GEN_BY_DEF_ON_NULL colpropは、指定した列がデフォルトnull列制約である場合に自動増分値を生成するときは、trueになります。

10.5 プリコンパイラ・オプションの使用方法について

この項は、プリコンパイラ・オプションを簡単に参照できるように構成されています。プリコンパイラ・オプションをアルファベット順に並べ、オプションごとに用途、構文およびデフォルト値を示しています。さらに「使用上の注意」で、オプションについて説明します。

10.5.1 AUTO_CONNECT

用途

CLUSTER$アカウントへの自動接続を可能にします。

構文

AUTO_CONNECT={YES | NO}

デフォルト値

NO

使用上の注意

コマンドラインまたは構成ファイルからのみ入力できます。

AUTO_CONNECT=YESで、最初の実行SQL文を処理するときにアプリケーションがまだデータベースに接続されていない場合、次のユーザーIDを使用して接続が試行されます。

```
CLUSTER$username
```

*username*は現行のオペレーティング・システムのユーザー名またはタスク名、CLUSTER$usernameは有効なOracleユーザーIDです。

AUTO_CONNECT=NOの場合、Oracleに接続するにはプログラムでCONNECT文を使用する必要があります。

10.5.2 CHAR_MAP

用途

char型またはchar[n]型のCホスト変数およびそれらに対するポインタのSQLへのデフォルトのマッピングを指定します。

構文

CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF}

デフォルト値

CHARZ

使用上の注意

リリース8.0より前のリリースでは、SQL DECLARE文を使用してCHARなどのcharまたはchar[n]ホスト変数を宣言する必要がありました。外部データ型VARCHAR2およびCHARZが、Oracle7のデフォルト文字マッピングでした。

関連項目:

- CHAR_MAP設定の表、データ型の説明、それがデフォルトになる場所については、VARCHAR変数を参照してください。
- Pro*C/C++でのCHAR_MAPの使用例は、CHAR_MAPオプションのインラインでの使用方法を参照してください。

10.5.3 CINCR

用途

データベースに対してオープンされる物理接続数の次の増分をアプリケーションで設定できるようにします。

構文

CINCR = 範囲は1から(CMAX-CMIN)。

デフォルト値

1

使用上の注意

最初は、CMINにより指定されたとおりに物理接続がすべてサーバーに対してオープンされます。それ以降は、必要な場合にのみ物理接続がオープンされます。パフォーマンスを最適にするには、CMINを、アプリケーションによる実行が計画または予想される同時実行文の合計数に設定する必要があります。デフォルト値は2に設定されます。

10.5.4 CLOSE_ON_COMMIT

用途

WITH HOLD句なしで宣言されたカーソルを、コミット時にすべてクローズするかどうかを指定します。

構文

CLOSE_ON_COMMIT={YES | NO}

デフォルト値

NO

使用上の注意

コマンドラインまたは構成ファイルからのみ入力できます。

このオプションは、DECLARE CURSOR文でWITH HOLD句を使用せずに宣言されたカーソルがある場合にのみ有効です(WITH HOLD句が指定されていると、このオプションも、MODEオプションに対応するそれまでの動作も無効になります)。CLOSE_ON_COMMITより高いレベルでMODEが指定されていると、MODEが優先されます。たとえば、デフォルトはMODE=ORACLEおよびCLOSE_ON_COMMIT=NOです。コマンドラインでMODE=ANSIと指定した場合は、WITH HOLD句を使用せずに宣言されているカーソルはすべて、コミット時にクローズされます。

CLOSE_ON_COMMIT=NO (MODE=ORACLEの場合)の場合、COMMITまたはROLLBACKを発行しても、クローズされるのはFOR UPDATE句を使用して宣言されたカーソルまたはCURRENT OF句で参照されるカーソルのみです。その他のカーソルはCOMMITまたはROLLBACK文による影響を受けず、オープンされている場合はオープンされたままです。ただし、CLOSE_ON_COMMIT=YES (MODE=ANSIの場合)のときにCOMMITまたはROLLBACKを発行すると、すべてのカー

ソルがクローズします。

関連項目

- スクロール可能カーソル
- マクロ・オプションおよびマイクロ・オプション

10.5.5 CMAX

用途

データベースに対してオープンできる物理接続の最大数を指定します。

構文

CINCR = 範囲は1から65535

デフォルト値

100

使用上の注意

CMAX値は、CMIN+CINCRより大きく設定する必要があります。この値に達すると、物理接続をそれ以上オープンできません。通常のアプリケーションでは、同時データベース操作を100個実行するように設定すれば十分です。ユーザーが適切な値を設定できます。

10.5.6 CMIN

用途

データベースに対してオープンできる物理接続の最小数を指定します。

構文

CINCR = 範囲は1から(CMAX-CINCR)。

デフォルト値

2

使用上の注意

CMAX値は、CMIN+CINCRより大きく設定する必要があります。この値に達すると、物理接続をそれ以上オープンできません。通常のアプリケーションでは、同時データベース操作を100個実行するように設定すれば十分です。ユーザーが適切な値を設定できます。

10.5.7 CNOWAIT

用途

この属性は、プール内の他のすべての物理接続が使用中で、物理接続の合計数がすでに最大値に達している場合に、アプリケーションで繰り返し物理接続を要求する必要があるかどうかを決定します。

構文

CNOWAIT = 範囲は1から65535。

デフォルト値

未設定を意味する0。

使用上の注意

物理接続が使用できず、これ以上物理接続をオープンできない場合、この属性が設定されているとエラーが発生します。そうでない場合、コールは別の接続が取得されるまで待機します。デフォルトでは、CNOWAITは設定されないため、スレッドは、エラーを戻すかわりに、空いている接続を取得できるまで待機します。

10.5.8 CODE

用途

Pro*C/C++プリコンパイラによって生成されるC関数プロトタイプの書式を指定します。(関数プロトタイプは、関数とその引数のデータ型を宣言します。)プリコンパイラは、Cコンパイラが外部参照を解決できるよう、SQLライブラリ・ルーチンのために関数プロトタイプを生成します。CODEオプションで、プロトタイピングを制御できます。

構文

CODE={ANSI_C | KR_C | CPP}

デフォルト値

KR_C

使用上の注意

コマンドラインでは入力できますが、インラインではできません。

ANSI C規格X3.159-1989は、関数プロトタイプを規定しています。CODE=ANSI_Cのとき、Pro*C/C++では、ANSI C規格に準拠する完全な関数プロトタイプが生成されます。次に例を示します。

```
extern void sqlora(long *, void *);
```

プリコンパイラでは、他のANSI準拠の構造体(**const**型修飾子など)も生成できます。

CODE=KR_C(デフォルト)のとき、生成された関数プロトタイプの引数リストは、次のようなコメントになります。

```
extern void sqlora(/*_ long *, void * _*/);
```

CコンパイラがX3.159規格に準拠していなければ、CODE=KR_Cを指定します。

CODE=CPPのとき、プリコンパイラではC++互換コードが生成されます。

関連項目:

このオプション値を使用するすべての結果は、コードの生成を参照してください。

10.5.9 COMMON_PARSER

用途

SQL99構文のSELECT、INSERT、DELETE、UPDATEおよびDECLARE CURSOR文のカーソル本体がサポートされます。

構文

COMMON_PARSER={YES | NO}

デフォルト値

NO

使用上の注意

コマンドラインで入力できます。

10.5.10 COMP_CHARSET

用途

使用するコンパイラでマルチバイト・キャラクタ・セットがサポートされているかどうかを、Pro*C/C++プリコンパイラに指定します。これは、マルチバイトのクライアント環境(たとえば、NLS_LANGがマルチバイト・キャラクタ・セットに設定されているとき)で作業する開発者向けです。

構文

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

デフォルト値

MULTI_BYTE

使用上の注意

コマンドラインでのみ入力できます。

COMP_CHARSET=MULTI_BYTE(デフォルト)が指定されると、Pro*C/C++では、マルチバイト・キャラクタ・セットをサポートしているコンパイラによりコンパイルされるCコードを生成します。

COMP_CHARSET=SINGLE_BYTEを指定すると、Pro*C/C++では、シングルバイト系のコンパイラ用のCコードが生成され、マルチバイト文字列中のダブルバイト文字の2バイト目がバックスラッシュ(¥)に対応するASCII文字である場合に起きる可能性のある面倒な問題に、生成されたCコードにより対処します。この場合、バックスラッシュ(¥)は、前にもう1つバックスラッシュを置くことでエスケープされます。

注意:

この機能は、一般的に、古い C コンパイラを使用してシフト JIS 環境で開発をするときに必要になります。

シングルバイト・キャラクタ・セットにNLS_LANGが設定されていると、このオプションは効果がありません。

10.5.11 CONFIG

用途

ユーザー構成ファイルの名前を指定します。

構文

CONFIG=*filename*

デフォルト値

なし。

使用上の注意

コマンドラインでのみ入力できます。

ユーザー構成ファイルの名前および位置をPro*C/C++に通知する方法は、このオプション以外にありません。

10.5.12 CPOOL

用途

このオプションに基づき、プリコンパイラでは、SQLLIBに接続プール機能を有効または無効にするように指示する適切なコードを生成します。

構文

CPOOL = {YES|NO}

デフォルト値

NO

使用上の注意

このオプションがNOに設定されている場合、プリコンパイラではその他の接続プーリング・オプションを無視します。

10.5.13 CPP_SUFFIX

用途

CPP_SUFFIXオプションを使用すると、CODE=CPPオプションを指定した場合に生成されるC++出力ファイルに、プリコンパイラによって付けられるファイル拡張子を指定できます。

構文

CPP_SUFFIX=*filename_extension*

デフォルト値

システム固有。

使用上の注意

ほとんどのCコンパイラでは、入力ファイルのデフォルト拡張子は.cになります。しかし、C++コンパイラでは、ファイル名の拡張子がコンパイラごとに異なる場合があります。CPP_SUFFIXオプションを指定すると、プリコンパイラで生成されるファイル名拡張子を指定できます。このオプションの値は、引用符もピリオドも付けない文字列です。たとえば、CPP_SUFFIX=ccまたはCPP_SUFFIX=Cのように指定します。

10.5.14 CTIMEOUT

用途

指定した時間(秒単位)より長い間アイドル状態になっている物理接続を終了し、オープンされている物理接続を最適な数に保ちます。

構文

CTIMEOUT = 範囲は1から65535。

デフォルト値

未設定を意味する0。

使用上の注意

接続プールが終了されるまで、物理接続はクローズされません。新しい物理接続を作成すると、サーバーへのラウンド・トリップが必要になります。

10.5.15 DB2_ARRAY

用途

このオプションに基づいて、プリコンパイラは追加の配列INSERTおよび配列SELECT構文をアクティブにします。

構文

DB2_ARRAY={YES |NO}

デフォルト値

NO

使用上の注意

このオプションをNOに設定すると、Oracleプリコンパイラの構文がサポートされます。それ以外の場合は、DB2の配列INSERTおよび配列SELECT構文がサポートされます。

10.5.16 DBMS

用途

OracleがOracle9*i*、Oracle8*i*、Oracle8、Oracle7、あるいはOracleのネイティブ・バージョン(つまり、アプリケーションが接続しているバージョン)のうち、どの意味上および構文上の規則に従うかを指定します。

構文

DBMS=NATIVE | V7 | V8

デフォルト値

NATIVE

使用上の注意

コマンドラインまたは構成ファイルでのみ入力できます。

DBMSオプションの指定によりOracleのバージョンに固有の動作を制御できます。DBMS=NATIVE(デフォルト値)のとき、Oracleはアプリケーションが接続しているデータベース・バージョンの意味上および構文上の規則に従います。

DBMS=V8またはDBMS=V7のときは、OracleはそれぞれOracle9*i*の規則(Oracle7、Oracle8およびOracle8*i*の場合と同じ規則)に従います。

OracleではV6_CHARはサポートされておらず、この機能はCHAR_MAPプリコンパイラ・オプションに用意されています。

表10-3 DBMSとMODEの相互作用

状況	DBMS=V7 | V8MODE=ANSI	DBMS=V7 | V8MODE=ORACLE
「データが見つかりません」という警告コード	+100	+1403

状況	DBMS=V7 | V8MODE=ANSI	DBMS=V7 | V8MODE=ORACLE
標識変数を使用しない NULL のフェッチ	エラー-1405	エラー-1405
インジケータ変数を使用しない切捨て値のフェッチ	エラーはなし。sqlwarn[1]が設定されます。	エラーはなし。sqlwarn[1]が設定されます。
COMMIT または ROLLBACK によるカーソルのクローズ	すべて明示的に実行されます。	CURRENT OF のみ
すでにオープンしているカーソルのオープン	エラー-2117	エラーなし
すでにクローズしているカーソルのクローズ	エラー-2114	エラーなし
SQL グループ関数による NULL の無視	警告なし	警告なし
複数行の問合せで SQL グループ関数をコールするとき	FETCH 時	FETCH 時
SQLCA 構造体の宣言	オプション	必須
SQLCODE または SQLSTATE 状態変数の宣言	必須	指定できるが Oracle は無視
整合性制約	有効	有効
ロールバック・セグメント用の PCTINCREASE	使用不可	使用不可
MAXEXTENTS 記憶域パラメータ	使用不可	使用不可

関連項目

- CHAR_MAP

10.5.17 DEF_SQLCODE

用途

Pro*C/C++プリコンパイラによりSQLCODEの**#define**が生成されるかどうかを制御します。

構文

DEF_SQLCODE={NO | YES}

デフォルト値

NO

使用上の注意

コマンドラインまたは構成ファイルからのみ入力できます。

DEF_SQLCODE=YESの場合、プリコンパイラでは生成されるソース・コードでSQLCODEが次のように定義されます。

```
#define SQLCODE sqlca.sqlcode
```

この定義があれば、SQLCODEを使用して実行SQL文の結果をチェックできます。DEF_SQLCODEオプションは、SQLCODEの使用が必要な規格への準拠のために指定します。

また、次のいずれかを入力して、ソース・コードにSQLCAも組み込む必要があります。

```
#include <sqlca.h>
```

または

```
EXEC SQL INCLUDE SQLCA;
```

SQLCAを組み込まなければ、このオプションを使用するとプリコンパイル時にエラーが発生します。

10.5.18 DEFINE

用途

#ifdefおよび**#ifndef** Pro*C/C++プリコンパイラ・ディレクティブで使用できる名称を定義します。定義された名称はEXEC ORACLE IFDEFとEXEC ORACLE IFNDEF文でも使用できます。

構文

DEFINE=*name*

デフォルト値

なし。

使用上の注意

コマンドラインまたはインラインで入力できます。DEFINEでは名称しか定義できません。マクロは定義できません。たとえば、次のような定義は無効です。

```
proc my_prog DEFINE=LEN=20
```

DEFINE文を正しく使用すれば、次のような指定ができます。

```
proc my_prog DEFINE=XYZZY
```

すると*my_prog.pc*に次のコードを記述できるようになります。

```
#ifdef XYZZY
...
#else
...
#endif
```

または、単純にコーディングできます。

```
EXEC ORACLE IFDEF XYZZY;
...
EXEC ORACLE ELSE;
...
```

```
EXEC ORACLE ENDIF;
```

次の例は無効です。

```
#define XYZZY
...
EXEC ORACLE IFDEF XYZZY
...
EXEC ORACLE ENDIF;
```

EXEC ORACLE DEFINEまたはDEFINEオプションでマクロが定義されているときにかぎり、EXEC ORACLE条件文が有効となります。

DEFINE=を使用して名前を定義してから、Pro*C/C++プリコンパイラの**#ifdef**(または**#ifndef**)ディレクティブを使用して条件を含めた(または除外した)場合は、Cコンパイラを実行するときにその名前が定義されていることを確認します。たとえば、UNIXのccでは、-Dオプションを使用してCコンパイラの名前を定義する必要があります。

10.5.19 DURATION

用途

後続のEXEC SQL OBJECT CREATE文とEXEC SQL OBJECT DEREF文に使用される確保継続時間を設定します。キャッシュ内のオブジェクトは、保持期間の終わりに暗黙的に解放されます。

構文

DURATION={TRANSACTION | SESSION}

デフォルト値

TRANSACTION

使用上の注意

EXEC ORACLE OPTION文を使用してインライン入力できます。

TRANSACTIONは、オブジェクトがトランザクションの完了時に暗黙的に解放されることを意味します。

SESSIONは、オブジェクトが接続の終了時に暗黙的に解放されることを意味します。

10.5.20 DYNAMIC

用途

このマイクロ・オプションでは、動的SQL方法4の記述子の動作を指定します。MODEの設定によりDYNAMICの設定が決まります。

構文

DYNAMIC={ORACLE | ANSI}

デフォルト値

ORACLE

使用上の注意

EXEC ORACLE OPTION文を使用してインラインで入力することはできません。

DYNAMICオプションの設定については、「ANSI動的SQLのプリコンパイラ・オプション」を参照してください。

10.5.21 ERRORS

用途

エラー・メッセージを端末とリスト・ファイルの両方に送信(YES)するか、リスト・ファイルにのみ送信(NO)するかを指定します。

構文

ERRORS={YES | NO}

デフォルト値

YES

使用上の注意

コマンドラインまたは構成ファイルでのみ入力できます。

10.5.22 ERRTYPE

用途

型ファイルの処理中に生成されたエラーを書き込む出力ファイルを指定します。省略すると、エラーは画面に出力されます。

構文

ERRTYPE=*filename*

デフォルト値

なし。

使用上の注意

生成されるエラー・ファイルは1つのみです。複数の値を入力すると、最後の値がプリコンパイラで使用されます。

関連項目

- INTYPE

10.5.23 EVENTS

用途

アプリケーションが通知の登録および受信に対応しているかどうかを指定します。

構文

EVENTS={YES | NO}

デフォルト値

NO

使用上の注意

コマンドラインからのみ入力できます。

10.5.24 FIPS

用途

ANSI SQLの拡張機能に(FIPSフラガーで)フラグを立てるかどうかを指定します。拡張機能とは、ANSIの形式または構文規

則(権限適用規則は除く)に違反するSQL要素を指します。

構文

FIPS={SQL89 | SQL2 | YES | NO}

デフォルト値

なし。

使用上の注意

インラインまたはコマンドラインで入力できます。

FIPS=YESの場合、FIPSフラガーは使用可能になります。このときANSI SQLのOracle拡張機能または非準拠の方法でANSI SQL機能を使用すると、(エラーではなく)警告メッセージが発行されます。プリコンパイル時にフラグ付けされるANSI SQL拡張機能には次のものがあります。

- FOR句を含む配列インタフェース
- SQLCA、ORACAおよびSQLDAデータ構造体
- DESCRIBE文を含む動的SQL
- 埋込みPL/SQLブロック
- 自動データ型変換
- DATE、NUMBER、RAW、LONGRAW、VARRAW、ROWID、VARCHAR2およびVARCHARデータ型
- ポインタ・ホスト変数
- ランタイム・オプション指定用のOracle OPTION文
- ユーザー・イグジットのTOOLS文
- CONNECT文
- TYPEおよびVARデータ型の同等文
- AT *db_name*句
- DECLARE...DATABASE文、...STATEMENT文および...TABLE文
- WHENEVER文でのSQLWARNING条件
- WHENEVER文におけるDO *function_name()*、DO BREAKアクションおよびDO CONTINUEアクション
- COMMIT文のCOMMENT句およびFORCE TRANSACTION句
- ROLLBACK文でのFORCE TRANSACTION句およびTO SAVEPOINT句
- COMMIT文およびROLLBACK文でのRELEASEパラメータ
- WHENEVER...GOTOラベルおよびINTO句のホスト変数の前に任意指定で付加するコロン

10.5.25 HEADER

用途

プリコンパイル済ヘッダー・ファイルを許可します。プリコンパイル済ヘッダー・ファイルのファイル拡張子を指定します。

構文

HEADER=*extension*

デフォルト値

なし

使用上の注意

ヘッダー・ファイルをプリコンパイルする場合、このオプションは必須で、ヘッダー・ファイルのプリコンパイルによって生成される出力ファイルのファイル拡張子の指定に使用されます。

通常のPro*C/C++プログラムをプリコンパイルする場合、このオプションは任意です。指定すると、Pro*C/C++プログラムのプリコンパイル時に、プリコンパイル済ヘッダーのメカニズムを使用できます。

どちらの場合も、このオプションで#includeディレクティブの処理時に使用するファイル拡張子も指定できます。指定した拡張子の付いた#includeファイルが存在する場合、Pro*C/C++ではそのファイルをPro*C/C++によって以前に生成されたプリコンパイル済ヘッダー・ファイルとみなします。Pro*C/C++は、#includeディレクティブを処理してインクルードされるヘッダー・ファイルをプリコンパイルするかわりに、そのファイルからデータをインスタンス化します。

このオプションは、コマンドラインまたは構成ファイルでのみ使用できます。インラインでは使用できません。このオプションを使用する場合は、ファイル拡張子のみを指定します。ファイル・セパレータは含めないでください。たとえば、拡張子にピリオド(.)は含めないでください。

関連項目

- プリコンパイル済ヘッダー・ファイル

10.5.26 HOLD_CURSOR

用途

カーソル・キャッシュでのSQL文およびPL/SQLブロック用カーソルの処理方法を指定します。

構文

HOLD_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

HOLD_CURSORを使用すると、プログラムのパフォーマンスを改善できます。

SQLデータ操作文を実行すると、その文に関連付けられたカーソルが、カーソル・キャッシュ内のエントリにリンクされます。そのカーソル・キャッシュ・エントリは、文の処理に必要な情報が格納されるOracleプライベートSQL領域にリンクされます。
HOLD_CURSORは、カーソルとカーソル・キャッシュの間のリンクで発生する処理を制御します。

HOLD_CURSOR=NOのとき、OracleがSQL文を実行してカーソルがクローズされた後に、プリコンパイラはそのリンクに再利用可能とマークします。このリンクは、それが示すカーソル・キャッシュ・エントリが別のSQL文に必要になると、すぐに再利用されます。これにより、プライベートSQL領域に割り当てられたメモリーが解放され、解析ロックが解除されます。

HOLD_CURSOR=YESを指定した場合は、リンクは維持され、プリコンパイラはリンクを再利用しません。これによって、以降の実行をスピードアップし、文を再解析したり、OracleプライベートSQL領域にメモリーを割り当てる必要がなくなるため、実行頻度の高いSQL文に使用すると便利です。

暗黙カーソルをインラインで使用する場合は、SQL文の実行前にHOLD_CURSORを設定してください。明示カーソル用としてインラインで使用するときは、カーソルをクローズする前にHOLD_CURSORを設定してください。

RELEASE_CURSOR=YESはHOLD_CURSOR=YESをオーバーライドし、HOLD_CURSOR=NOはRELEASE_CURSOR=NOをオーバーライドします。この2つのオプションの相互作用方法を示す情報は、表B-1を参照してください。

関連項目

- パフォーマンス・チューニング

10.5.27 IMPLICIT_SVPT

用途

新しくバッチ処理された挿入を開始する前に、暗黙的なセーブポイントを設定するかどうかを制御します。

構文

implicit_svpt={YES|NO}

デフォルト値

NO

使用上の注意

implict_svpt=yesの場合、行の新しいバッチの開始前に、セーブポイントが設定されます。挿入時にエラーが発生すると、暗黙的な「セーブポイントへのロールバック」が実行されます。このオプションは、DB2との互換性のために用意されていますが、余分なラウンドトリップとなるためマイナス面であることは明らかです。

implict_svpt=noの場合、暗黙的セーブポイントは設定されません。バッファ済INSERTでエラーが発生すると、アプリケーションに通知されますが、ロールバックは実行されません。

10.5.28 INAME

用途

入力ファイル名を指定します。

構文

INAME=*path_and_filename*

デフォルト値

なし。

使用上の注意

コマンドラインでのみ入力できます。

プリコンパイル時は、すべての入力ファイル名を一意にする必要があります。

ファイル名拡張子が.pcの場合は省略できます。入力ファイル名がコマンドラインの先頭オプションになっているときは、オプションのINAME=の部分を省略できます。次に例を示します。

```
proc sample1 MODE=ansi
```

この例では、ANSIモードを使用してファイルsample1.pcをプリコンパイルします。このコマンドは次のコマンドに相当します。

```
proc INAME=sample1 MODE=ansi
```

> **注意:**
>
> sqlctx ハッシュ値は、Pro*C/C++コマンドに渡される INAME パラメータに基づいて生成されます。そのため、同じ名前の複数のファイルが、異なる関数を含む異なるディレクトリに格納されており、プログラムのプリコンパイルのためにビルド・スクリプトが物理ディレクトリに送られるようなアプリケーションでは、問題が発生する可能性があります。したがって、より高いレベルに Make ファイルを配置し、そのパス名を使用してファイルをプリコンパイルする必要はありません。

10.5.29 INCLUDE

用途

#includeまたはEXEC SQL INCLUDEディレクティブによって取り込まれるファイルのディレクトリ・パスを指定します。

構文

INCLUDE=*pathname*またはINCLUDE=(*path_1*,*path_2*,...,*path_n*)

デフォルト値

Pro*C/C++にインクルードするカレント・ディレクトリおよびパス

使用上の注意

インラインまたはコマンドラインで入力できます。

INCLUDEは、インクルードされたファイルに対するディレクトリ・パスを指定するために使用します。プリコンパイラでは、次の順序でディレクトリが検索されます。

1. カレント・ディレクトリ
2. SYS_INCLUDEプリコンパイラ・オプションに指定されているシステム・ディレクトリ
3. INCLUDEオプションで指定された入力順のディレクトリ
4. 標準ヘッダー・ファイル用の組込みディレクトリ

通常、Oracle固有のヘッダー・ファイルsqlca.hやsqlda.hなどのディレクトリ・パスを指定する必要はありません。

> **注意:**
>
> インクルードするファイルに Oracle 固有のファイル名を拡張子なしで指定すると、Pro*C/C++は拡張子.h とみなします。このため、インクルード・ファイルには、.h ではないにせよ、拡張子を付ける必要があります。

他のすべてのヘッダー・ファイルに対しては、プリコンパイラは拡張子.hを想定しません。

非標準ファイルの場合は、カレント・ディレクトリに格納されていないかぎり、INCLUDEを使用してディレクトリ・パスを指定する必要があります。次に示すように、コマンドラインに複数のパスを指定できます。

```
... INCLUDE=path_1 INCLUDE=path_2 ...
```

> **注意:**
>
> インクルードするファイルが別ディレクトリに存在している場合には、カレント・ディレクトリには同じ名前のファイルがないことを確認してください。

INCLUDEオプションを使用してディレクトリ・パスを指定する構文は、システムによって異なります。各オペレーティング・システムの規則に従ってください。

> **注意:**
>
> INCLUDE オプションの場合、オプション値の優先順位が逆になります。値が上書きされる他のオプションとは異なり、INCLUDE は次の場所で指定されたすべてのディレクトリ・ファイルを追加します。
>
> - プリコンパイラ
> - Pro*C/C++システム構成ファイル
> - Pro*C/C++ユーザー構成ファイル
> - コマンドライン
> - インライン
>
> ただし、カッコ内またはカッコなしでの値の渡し方には違いがあります。カッコ内で単一の値またはディレクトリ・リストを渡す場合、INCLUDE の既存の値は上書きされます。リストをカッコなしの単純な値として渡す場合、既存の値に追加されます。

10.5.30 INTYPE

用途

OTTで生成された型のファイルを1つ以上指定します(アプリケーションでオブジェクト型が使用される場合にのみ必要です)。

構文

INTYPE=(*file_1*,*file_2*,...,*file_n*)

デフォルト値

なし。

使用上の注意

Pro*C/C++コードには、オブジェクト型ごとに1つの型のファイルが存在します。

10.5.31 LINES

用途

Pro*C/C++プリコンパイラでその出力ファイルに**#line**プリプロセッサ・ディレクティブが追加されるかどうかを指定します。

構文

LINES={YES | NO}

デフォルト値

NO

使用上の注意

コマンドラインでのみ入力できます。

LINESオプションはデバッグに便利です。

LINES=YESの場合、Pro*C/C++プリコンパイラではその出力ファイルに**#line**プリプロセッサ・ディレクティブを追加します。

通常、Cコンパイラでは、それぞれの入力行が処理されるたびに行カウントを増やします。**#line**ディレクティブは、コンパイラの入力行カウントを強制的にリセットして、プリコンパイラで生成されたコードの行を数えないようにします。さらに、入力ファイルの名前が変わったときに、次の**#line**ディレクティブが新しいファイル名を指定します。

Cコンパイラでは、行番号とファイル名を使用して、エラーの発生場所を示します。したがって、Cコンパイラで発行されるエラー・メッセージは、変更済(プリコンパイル済)のソース・ファイルではなく、常に元のソース・ファイルを参照します。これにより、ほとんどのデバッガを使用して、元のソース・コードを1ステップずつ実行することもできます。

LINES=NO(デフォルト)の場合、プリコンパイラでは出力ファイルに**#line**ディレクティブは追加されません。

> 注意:
>
> Pro*C/C++プリコンパイラでは、**#line** ディレクティブはサポートされません。これは、プリコンパイラ・ソースでは **#line** ディレクティブを直接コーディングできないことを意味します。ただし、LINES=オプションを使用すると、プリコンパイラに**#line** ディレクティブを挿入させることができます。

関連項目

- 無視されるディレクティブ

10.5.32 LNAME

用途

リスト・ファイル名を指定します。

構文

LNAME=*filename*

デフォルト値

なし。

使用上の注意

コマンドラインでのみ入力できます。

リスト・ファイルのデフォルトのファイル名の拡張子は.lisです。

10.5.33 LTYPE

用途

生成されるリスト・ファイルの型を指定します。

構文

LTYPE={NONE | SHORT | LONG}

デフォルト値

SHORT

使用上の注意

コマンドラインまたは構成ファイルで入力できます。

リスト・ファイル生成時のデフォルトの形式はLONGです。LTYPE=LONGを指定すると、すべてのソース・コードが解析順に出力されます。また、メッセージも生成順に出力されます。さらに、現在有効なPro*C/C++オプションが出力されます。

LTYPE=SHORTを指定すると、生成されたメッセージのみが出力され、ソース・コードは出力されません。ソース・ファイルへの参照行はメッセージ状態を生成したコードの位置を特定するのに役立ちます。

LTYPE=NONEを指定すると、LNAMEオプションでリスト・ファイル名を明示的に指定しないかぎり、リスト・ファイルは作成されません。後者の場合、リスト・ファイルはLTYPE=LONGを想定して生成されます。

10.5.34 MAX_ROW_INSERT

用途

INSERT文の実行前にバッファする必要がある行の数を制御します。

構文

max_row_insert={0から1000}

デフォルト値

0

使用上の注意

0よりも大きい任意の値を指定すると、バッファ済INSERT機能が有効化され、INSERT文の実行前に多くの行がバッファされます。

10.5.35 MAXLITERAL

用途

プリコンパイラによって生成される文字列リテラルの最大長を指定します。これによってコンパイラの制限を超えないようにします。

構文

MAXLITERAL=*integer*(範囲は10から1024)

デフォルト値

1024

使用上の注意

インラインでは入力できません。

MAXLITERALに指定できる最大値はコンパイラによって異なります。たとえば、Cコンパイラには513文字以上の文字列リテラルを扱えないものがあるため、そのような場合はMAXLITERAL=512と指定します。

MAXLITERALで指定した長さを超える文字列はプリコンパイル中に分割され、実行時に再び結合(連結)されます。

インラインでMAXLITERALを入力することはできますが、プログラムで値を設定できるのは1回のみで、EXEC ORACLE文を最初のEXEC SQL文の前に指定する必要があります。指定しない場合、Pro*C/C++は警告メッセージを発行し、余分または誤って指定したEXEC ORACLE文を無視して、処理を続行します。

10.5.36 MAXOPENCURSORS

用途

同時にオープンされ、プリコンパイラによりキャッシュに保存されたままになるカーソルの数を指定します。

構文

MAXOPENCURSORS=*integer*

デフォルト値

10

使用上の注意

インラインまたはコマンドラインで入力できます。

MAXOPENCURSORSを使用すると、プログラムのパフォーマンスを改善できます。分割プリコンパイル時には、MAXOPENCURSORSを使用します。MAXOPENCURSORSオプションには、SQLLIBカーソル・キャッシュの初期サイズを指定します。

HOLD_CURSOR=NOのときに、暗黙的な文が実行されるか明示カーソルがクローズされると、カーソル・エントリは再利用可能とマークされます。この文が再び発行された時にカーソル・エントリが別の文に使用されていなければ、カーソルは再利用されます。

割当て済のカーソル数がMAXOPENCURSORSに満たない場合に新しいカーソルが必要になると、キャッシュに格納された次のカーソルが割り当てられます。MAXOPENCCURSORSを超えると、Oracleではまず以前のエントリの再利用を試みます。空いているエントリがない場合、追加キャッシュ・エントリが割り当てられます。Oracleは、プログラムがメモリーを消費するか、データベース・パラメータのOPEN_CURSORSを超過するまで継続してこれを行います。

通常の処理では、HOLD_CURSORS=NOでRELEASE_CURSOR=NO (デフォルト)を使用するときには、データ・ディクショナリで使用されるカーソルが文を処理できるように、MAXOPENCURSORSの値をOPEN_CURSORSデータベース・パラメータの値より6以上小さい値に設定しないことをお薦めします。

プログラムの同時オープン・カーソル数を増す必要がある場合には、MAXOPENCURSORSを必要な数まで増やして再度指定することがあります。45から50の値を指定することは珍しくありませんが、ユーザー・プロセスのメモリー領域にカーソル1つにつき、1つのプライベートSQL領域が必要なことに注意してください。デフォルト値の10は、大半のプログラムには適切な値です。

関連項目

- パフォーマンス・チューニング

- プログラミングのガイドライン

10.5.37 MODE

用途

プログラムがOracleの動作規則に従うかどうか、あるいは現行のANSI/ISO SQL規格に準拠するかどうかを指定します。

構文

MODE={ANSI | ISO | ORACLE}

デフォルト値

ORACLE

使用上の注意

コマンドラインまたは構成ファイルからのみ入力できます。

このオプションでは、ISOはANSIと同じ意味です。

MODE=ORACLE (デフォルト)のとき、埋込みSQLプログラムはOracleの動作規則に従います。たとえば、宣言部はオプションで、空白セルは削除されます。

MODE=ANSIの場合、プログラムは完全にANSI SQL規格に準拠し、次のような変更が有効になります。

- COMMITまたはROLLBACKを発行すると、明示カーソルがすべてクローズされます。
- すでにオープンされているカーソルのOPENや、クローズされているカーソルのCLOSEはできません。
  (MODE=ORACLEの場合は、再解析を避けるためにオープンされているカーソルを再オープンできます。)
- 各EXEC SQL文のスコープ内に*SQLCODE*という名前の**long**変数または**char**のSQLSTATE[6]変数(どちらの変数も大文字にする必要があります)のいずれかを宣言する必要があります。すべての場合に同一の*SQLCODE*または*SQLSTATE*変数を使用する必要はありません。つまり、変数はグローバル変数でなくてもかまいません。
- SQLCAの宣言はオプションです。SQLCAを挿入する必要はありません。
- SQLCODEに戻される「データが見つかりません」というOracle警告コードは、+1403から+100になります。メッセージ・テキストは変更されていません。
- ホスト変数に宣言部が必要です。

10.5.38 NATIVE_TYPES

用途

ネイティブfloat/doubleをサポートします。

構文

NATIVE_TYPES = {YES|NO}

デフォルト値

NO

使用上の注意

ネイティブfloatおよびネイティブdoubleデータ型は、単精度と倍精度の浮動小数点値を表します。これらはネイティブ、つまりホスト・システムの浮動小数点形式で表されます。

10.5.39 NLS_CHAR

用途

プリコンパイラでマルチバイト文字変数として扱われるCホスト文字変数を指定します。

構文

NLS_CHAR=*varname*またはNLS_CHAR=(*var_1*,*var_2*,...,*var_n*)

デフォルト値

なし。

使用上の注意

コマンドラインまたは構成ファイルでのみ入力できます。

このオプションを使用すると、プリコンパイラでマルチバイト文字変数として扱う必要のある1つ以上のホスト変数のリストを、プリコンパイル時に指定できます。このオプションでは、C言語の*char*変数またはPro*C/C++のVARCHAR変数のみを指定できます。

オプション・リストにプログラムで宣言していない変数を指定しても、プリコンパイラのエラーは発生しません。

10.5.40 NLS_LOCAL

用途

プリコンパイラのSQLLIBランタイム・ライブラリとデータベース・サーバーのうち、どちらでマルチバイト・キャラクタ・セット変換を実行するかを指定します。

構文

NLS_LOCAL={YES | NO}

デフォルト値

NO

使用上の注意

YESに設定すると、Pro*C/C++およびSQLLIBライブラリによって、ローカル・マルチバイト・サポートが提供されます。どのCホスト変数がマルチバイトかを指定するには、NLS_CHARオプションを使用する必要があります。

NOに設定すると、Pro*C/C++では、データベース・サーバーのマルチバイト・オブジェクトのサポートを使用します。新規アプリケーションにはすべて、NLS_LOCALをNOに設定してください。

環境変数NLS_NCHARには、有効な固定幅の各国語キャラクタ・セットを設定する必要があります。可変長幅の各国語キャラクタ・セットはサポートされていません。

コマンドラインまたは構成ファイルでのみ入力できます。

10.5.41 OBJECTS

用途

オブジェクト型のサポートを要求します。

構文

OBJECTS={YES | NO}

デフォルト値

YES

使用上の注意

コマンドラインからのみ入力できます。

10.5.42 ONAME

用途

出力ファイル名を指定します。出力ファイルは、プリコンパイラで生成されるCコードのファイルです。

構文

ONAME=*path_and_filename*

デフォルト値

.c拡張子付きのINAME

使用上の注意

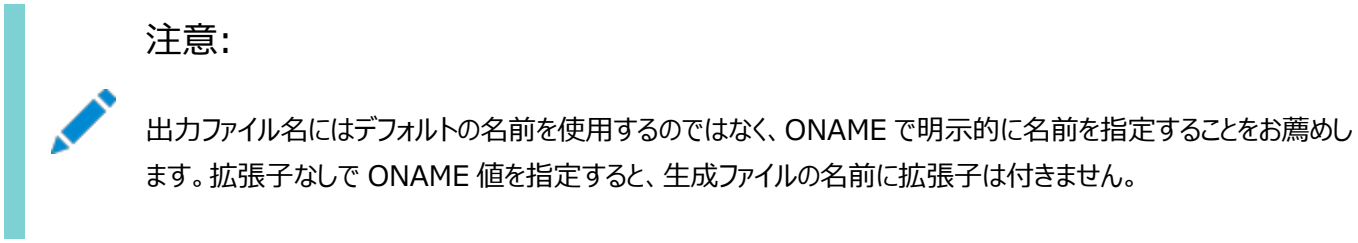
コマンドラインでのみ入力できます。このオプションを使用すると、入力(.pc)ファイルとは異なるパス名を使用して、出力ファイルのフルパス名を指定できます。たとえば、次のコマンドを発行する場合を考えます。

```
proc iname=my_test
```

デフォルト出力ファイル名はmy_test.cです。出力ファイル名をmy_test_1.cにする場合は、次のコマンドを発行します。

```
proc iname=my_test oname=my_test_1.c
```

ONAMEで指定するファイルにはデフォルトで拡張子が追加されないため、.cを付加する必要があります。



注意:

出力ファイル名にはデフォルトの名前を使用するのではなく、ONAME で明示的に名前を指定することをお薦めします。拡張子なしで ONAME 値を指定すると、生成ファイルの名前に拡張子は付きません。

10.5.43 ORACA

用途

プログラムでOracle通信領域(ORACA)を使用できるかどうかを指定します。

構文

ORACA={YES | NO}

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

ORACA=YESのときは、プログラムにEXEC SQL INCLUDE ORACA文または**#include** oraca.h文を記述する必要があります。

10.5.44 OUTLINE

用途

SQL文に対してアウトラインSQLファイルを生成する必要があることを示します。

構文

outline={yes | no | category_name}

デフォルト値

no

使用上の注意

値がyesの場合、アウトラインSQLファイルはDEFAULTカテゴリに含まれている必要があります。生成されるアウトライン書式は次のとおりです。

```
DEFAULT_<filename>_<filetype>_<sequence_no>
```

カテゴリ名が示されている場合は、そのカテゴリにSQLファイルを生成する必要があります。この場合、生成されるアウトライン書式は次のようになります。

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

値がnoの場合、アウトラインSQLファイルは生成されません。

このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneの場合は、エラーが生成されます。

10.5.45 OUTLNPREFIX

用途

アウトライン名の生成を制御します。

構文

outlnprefix={none | prefix_name}

デフォルト値

no

使用上の注意

outlnprefix=prefix_nameの場合、アウトライン書式は次のとおりです。

```
<category_name>_<filename>_<filetype>
```

この書式は、アウトライン名の<prefix_name>に置き換えられます。

アウトライン名が128バイトを超える場合、このオプションは接頭辞名を指定する際に有効です。

outlnprefix=noneの場合、アウトライン名はシステムによって生成されます。次の書式で生成されます。

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneまたはoutline=false(あるいはその両方)の場合、エラーが生成されます。

10.5.46 PAGELEN

用途

リスト・ファイルの物理ページごとの行数を指定します。

構文

PAGELEN=*integer*

デフォルト値

80

使用上の注意

インラインでは入力できません。値の許容範囲は30から256です。

10.5.47 PARSE

用途

Pro*C/C++プリコンパイラでソース・ファイルを解析する方法を指定します。

構文

PARSE={FULL | PARTIAL | NONE}

デフォルト値

FULL

使用上の注意

C++互換コードを生成するには、PARSEオプションにNONEまたはPARTIALのいずれかを指定する必要があります。

PARSE=NONEまたはPARSE=PARTIALの場合、すべてのホスト変数は宣言部内で宣言する必要があります。

変数SQLCODEを宣言部の内側で宣言しない場合、エラー検出の信頼性がなくなります。使用しているプラットフォームのPARSEのデフォルト値をチェックしてください。

PARSE=FULLの場合、Cパーサーが動作し、コード内のクラスなどのC++構造体は認識されません。

PARSE=FULLまたはPARSE=PARTIALを指定すると、Pro*C/C++は**#define**、**#ifdef**などのCプリプロセッサ・ディレクティブを完全にサポートします。ただし、PARSE=NONEを指定すると、EXEC ORACLE文により条件付きプリプロセッシングがサポートされます。

注意:

一部のプラットフォームでは、PARSE のデフォルト値が FULL 以外の値です。使用するシステム固有のマニュアルを参照してください。

関連項目

- コードの解析について
- 宣言部
- 条件付きプリコンパイル

10.5.48 PLAN_BASELINE

用途

モジュール名を指定してSQL計画ベースラインを作成します。

構文

PLAN_BASELINE={*module_name* | YES | NO}

デフォルト値

NO

使用上の注意

このモジュール名の後にファイル名、ファイル・タイプおよび順序番号が付いて、一意の計画名が生成されます。

10.5.49 PLAN_PREFIX

用途

計画名が128バイト以内であることを確認します。

構文

PLAN_PREFIX={*prefix_name*| none}

デフォルト値

なし

使用上の注意

これはオプションです。デフォルトはnoneで、接頭辞名が使用されないことを意味し、計画名が128バイトを超えるとエラー・メッセージが生成されます。

10.5.50 PLAN_RUN

用途

生成されたSQLファイルを実行します。

構文

PLAN_RUN={YES | NO}

デフォルト値

NO

使用上の注意

PLAN_RUNオプションを設定しない場合、生成されるSQLファイルは実行されません。

10.5.51 PLAN_FIXED

用途

作成した計画ベースラインが固定であるか、非固定であるかを指定します。

構文

PLAN_FIXED={ YES | NO }

デフォルト値

YES

使用上の注意

NOを設定すると、非固定の計画ベースラインが作成されます。

10.5.52 PLAN_ENABLED

用途

作成された計画ベースラインの使用を有効にします。

構文

PLAN_ENABLED={ YES | NO }

デフォルト値

YES

使用上の注意

デフォルトの設定では、計画の選択時に、作成された計画ベースラインが使用されます。NOを設定すると、計画ベースラインが作成されますが、手動で有効にするまで使用されません。

10.5.53 MEMFORPREFETCH

用途

このオプションを使用すると、指定したメモリーに格納可能な行数をプリフェッチすることによって、問合せが高速化します。

構文

MEMFORPREFETCH=*integer*

デフォルト値

値は設定されていません。

使用上の注意

このオプションは、構成ファイルまたはコマンドラインで使用できます。優先順位の規則に従い、明示カーソルを使用するすべての問合せの実行に、整数の値が使用されます。

インラインで使用するときは、明示カーソルを使用し、OPEN文の前に配置する必要があります。OPEN文が実行されたときにプリフェッチされる行数は、有効な最後のインラインMEMFORPREFETCHオプションによって決まります。

MEMFORPREFETCHは、デフォルトでは値が設定されていません。プリフェッチをオフにするには、コマンドラインでMEMFORPREFETCH=0を使用します。

LONG列およびLOB列へのアクセス中もプリフェッチはオフになります。MEMFORPREFETCHを使用すると、単一行フェッチのパフォーマンスが向上します。配列フェッチを実行する場合、MEMFORPREFETCHの値は割り当てた値に関係なく無効になります。

アプリケーションにおけるすべてのフェッチを支援できる完全なプリフェッチ・メモリー値はありません。

したがって、MEMFORPREFETCHオプションを使用する場合は、様々な値をテストし、プログラム内のすべての文にわたってパフォーマンスが向上する値を選択してください。いくつかの文を個別にチューニングする必要がある場合は、EXEC ORACLE OPTIONを使用してMEMFORPREFETCHオプションをインラインで指定します。この操作は、このコマンドの後のすべてのフェッチ文に影響を与えます。特定のFETCH文のパフォーマンスが向上するようにプリフェッチ・メモリーを選択してください。この個別プリフェッチ・カウントを実現するには、(コマンドラインからではなく)インラインのプリフェッチ・オプションを指定します。

10.5.54 PREFETCH

用途

任意の行数を事前に取得して問合せの実行速度を向上させます。

構文

PREFETCH=*integer*

デフォルト値

1

使用上の注意

構成ファイルまたはコマンドラインで入力できます。優先順位の規則に従い、明示カーソルを使用するすべての問合せの実行に、整数の値が使用されます。

インラインで使用する場合、明示カーソルのあるOPEN文の前に置く必要があります。OPEN文が実行されたときにプリフェッチされる行数は、有効な最後のインラインPREFETCHオプションによって決まります。

指定可能な値の範囲は0から65535です。

10.5.55 RELEASE_CURSOR

用途

カーソル・キャッシュでのSQL文およびPL/SQLブロック用カーソルの処理方法を指定します。

構文

RELEASE_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

RELEASE_CURSORを使用すると、プログラムのパフォーマンスを改善できます。

SQLデータ操作文を実行すると、その文に関連付けられたカーソルが、カーソル・キャッシュ内のエントリにリンクされます。そのカーソル・キャッシュ・エントリは、文の処理に必要な情報が格納されるOracleプライベートSQL領域にリンクされます。RELEASE_CURSORは、カーソル・キャッシュとプライベートSQL領域の間のリンクで発生する処理を制御します。

RELEASE_CURSOR=YESに指定すると、OracleがSQL文を実行し、カーソルがクローズされた後、プリコンパイラはこのリンクはただちに解除します。これにより、プライベートSQL領域に割り当てられたメモリーが解放され、解析ロックが解除されます。カーソルのCLOSE時に関連リソースが確実に解放されるようにするには、RELEASE_CURSOR=YESを指定する必要があります。

RELEASE_CURSOR=NOを指定すると、リンクは保持されます。オープン・カーソルの数がMAXOPENCURSORSの設定値を超えないかぎり、プリコンパイラではリンクは再利用されません。この設定によって後続の処理の実行速度が向上するため、これは実行頻度の高いSQL文には便利です。文を解析しなおしたり、OracleプライベートSQL領域にメモリーを割り当てる必要がないためです。

暗黙カーソルをインラインで使用する場合は、SQL文の実行前にRELEASE_CURSORを設定してください。明示カーソル用としてインラインで使用するときは、カーソルをCLOSEする前にRELEASE_CURSORを設定してください。

RELEASE_CURSOR=YESの場合は、HOLD_CURSOR=YESが上書きされます。

関連項目

- パフォーマンス・チューニング

10.5.56 RUNOUTLINE

用途

プリコンパイラを使用して、または後で開発者が手動で「CREATE OUTLINE」文を実行するオプションを提供します。

構文

runoutline={yes | no}

デフォルト値

no

使用上の注意

runoutline=yesの場合は、プリコンパイルが正常に完了した後で、プリコンパイラ/トランスレータによって、生成された「CREATE OUTLINE」文が実行されます。

RUNOUTLINEを使用する場合は、アウトライン・オプションをtrueまたはcategory_nameに設定する必要があります。このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneの場合は、エラーが生成されます。

10.5.57 SELECT_ERROR

用途

SELECT文が複数行を戻すとき、あるいはホスト配列の許容範囲を超える数の行を戻すときに、プログラムでエラーが生成されるかどうかを指定します。

構文

SELECT_ERROR={YES | NO}

デフォルト値

YES

使用上の注意

インラインまたはコマンドラインで入力できます。

SELECT_ERROR=YESのときは、行単位のSELECT文が戻した行数が多すぎたり、配列単位のSELECT文が戻した行がホスト配列に入りきらなかったりしたときに、エラーが生成されます。SELECT文の結果は未定義です。

SELECT_ERROR=NOのときは、行単位のSELECT文が戻した行数が多すぎても、配列単位のSELECT文が戻した行がホスト配列に入りきらなくても、エラーは生成されません。

YESを指定してもNOを指定しても、行は表から無作為に選択されます。選択する行の順序を特定するには、SELECT文にORDER BY句を指定する以外に方法はありません。SELECT_ERROR=NOのときORDER BY句を指定すると、配列からの選択時にOracleは先頭行または先頭の*n*行を戻します。SELECT_ERROR=YESを指定すると、ORDER BY句の有無を問わず、戻る行数が多すぎる場合にエラーが生成されます。

10.5.58 STMT_CACHE

用途

動的SQL文の文キャッシュ・サイズを指定します。

構文

STMT_CACHE = 0から65535

デフォルト値

0

使用上の注意

stmt_cacheオプションを設定すると、アプリケーションでそれぞれの動的SQL文の予測数を保持できます。

10.5.59 SYS_INCLUDE

用途

システム・ヘッダー・ファイルの位置を指定します。

構文

SYS_INCLUDE=*pathname* | (*path1*, ..., *pathn*)

デフォルト値

システム固有。

使用上の注意

Pro*C/C++では、プラットフォーム固有の標準的な位置でstdio.hなどの標準のシステム・ヘッダー・ファイルが検索されます。たとえば、ほとんどのUNIXシステムではstdio.hファイルのフルパス名は/usr/include/stdio.hです。

ただし、C++コンパイラには、stdio.hなどのシステム・ヘッダー・ファイルが標準的なシステム位置にないものがあります。SYS_INCLUDEコマンドライン・オプションを使用すると、Pro*C/C++でシステム・ヘッダー・ファイルを検索するためのディレクトリ・パスのリストを指定できます。次に例を示します。

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

SYS_INCLUDEを使用して指定した検索パスは、デフォルトのヘッダー位置より優先されます。

PARSE=NONEの場合、Pro*C/C++はシステム・ヘッダー・ファイルをプリコンパイルに含める必要がないため、SYS_INCLUDEで指定した値は無視されます。(ただし、Oracle特有のヘッダーsqlca.hやシステム・ヘッダー・ファイルなどは、

コンパイラによるプリプロセッシングのために#includeディレクティブを付けて、挿入する必要があります。)

プリコンパイラでは、次の順序でディレクトリが検索されます。

1. カレント・ディレクトリ
2. SYS_INCLUDEプリコンパイラ・オプションで指定されたシステム・ディレクトリ
3. INCLUDEオプションで指定されたディレクトリ(入力順)
4. 標準ヘッダー・ファイル用の組込みディレクトリ

手順3があるため、通常はsqlca.hとsqlda.hなどの標準ヘッダー・ファイルのディレクトリ・パスを指定する必要はありません。

SYS_INCLUDEオプションを使用してディレクトリ・パスを指定する構文は、システムによって異なります。各オペレーティング・システムの規則に従ってください。

注意:

SYS_INCLUDE オプションの場合、オプション値の優先順位が逆になります。値が上書きされる他のオプションとは異なり、SYS_INCLUDE は次の場所で指定されたすべてのディレクトリ・ファイルを追加します。

- プリコンパイラ
- Pro*C/C++システム構成ファイル
- Pro*C/C++ユーザー構成ファイル
- コマンドライン
- インライン

ただし、カッコ内またはカッコなしでの値の渡し方には違いがあります。カッコ内で単一の値またはディレクトリ・リストを渡す場合、SYS_INCLUDE の既存の値は上書きされます。リストをカッコなしの単純な値として渡す場合、既存の値に追加されます。

10.5.60 THREADS

用途

THREADS=YESの場合、プリコンパイラではコンテキスト宣言を検索します。

構文

THREADS={YES | NO}

デフォルト値

NO

使用上の注意

インラインでは入力できません。

マルチスレッド・サポートを必要とするプログラムには、すべてこのオプションを指定する必要があります。

THREADS=YESの場合、最初のコンテキストが現れ、実行SQL文が見つかる前にEXEC SQL CONTEXT USEディレクティブが検出されないと、プリコンパイラではエラーが発生します。

関連項目

- マルチスレッド・アプリケーション

10.5.61 TYPE_CODE

用途

このマイクロ・オプションは、動的SQL方法4でANSIまたはOracleのいずれのデータ型コードを使用するかを指定します。この設定は、MODEオプションの設定と同じです。

構文

TYPE_CODE={ORACLE | ANSI}

デフォルト値

ORACLE

使用上の注意

インラインでは入力できません。

設定できるオプションの詳細は、 表14-3を参照してください。

10.5.62 UNSAFE_NULL

用途

UNSAFE_NULL=YESを指定すると、標識変数を使用しないでNULLをフェッチしても、ORA-01405メッセージは生成されません。

構文

UNSAFE_NULL={YES | NO}

デフォルト値

NO

使用上の注意

インラインでは入力できません。

MODE=ORACLEの場合のみ、UNSAFE_NULL=YESを設定できます。

埋込みPL/SQLブロックのホスト変数ではUNSAFE_NULLオプションは何の効果もありません。ORA-01405エラーの発生を避けるために、必ず標識変数を使用してください。

10.5.63 USERID

用途

Oracleユーザー名およびパスワードを指定します。

構文

USERID=*username*/*password*[@*dbname*]

デフォルト値

なし。

使用上の注意

コマンドラインでのみ入力できます。

動接続機能を使用している場合は、このオプションを指定しないでください。自動接続機能では、先頭にCLUSTER$の付いたOracleユーザー名しか受け付けません。「CLUSTER$」文字列の実際の値は、INIT.ORAファイルのパラメータとして設定されています。

SQLCHECK=SEMANTICSのとき、Oracleに接続してデータ・ディクショナリにアクセスしてプリコンパイラに必要な情報を取得させるには、USERIDも同時に指定する必要があります。

10.5.64 UTF16_CHARSET

用途

UNICODE(UTF16)変数で使用されるキャラクタ・セットの形式を指定します。

構文

```
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}
```

デフォルト値

```
NCHAR_CHARSET
```

使用上の注意

コマンドラインまたは構成ファイルでのみ使用でき、インラインでは使用できません。

UTF16_CHARSET=NCHAR_CHARSET(デフォルト)の場合、UNICODE(UTF16)のバインドまたは定義バッファは、サーバー側の各国語キャラクタ・セットに従って変換されます。ターゲット列がCHARの場合は、パフォーマンスが低下することがあります。

UTF16_CHAR=DB_CHARSETの場合、UNICODE(UTF16)バインドまたは定義バッファは、データベースのキャラクタ・セットに従って変換されます。

警告:

ターゲット列が NCHAR の場合、データが失われることがあります。

10.5.65 VARCHAR

用途

ある構造体がVARCHARホスト変数として解釈されるように、Pro*C/C++プリコンパイラに対して指示します。

構文

VARCHAR={NO | YES}

デフォルト値

NO

使用上の注意

コマンドラインでのみ入力できます。

VARCHAR=YESのとき、次のようにC言語の構造体を記述します。

```
struct {
    short len;
    char  arr[n];
} name;
```

この場合、プリコンパイラによってVARCHAR[n]型のホスト変数として解釈されます。

VARCHARはNLS_CHARオプションとともに使用してマルチバイト文字変数を指定できます。

10.5.66 VERSION

用途

EXEC SQL OBJECT DEREF文によって戻されるオブジェクトのバージョンを指定します。

構文

VERSION={RECENT | LATEST | ANY}

デフォルト値

RECENT

使用上の注意

EXEC ORACLE OPTION文を使用してインライン入力できます。

RECENTは、現行のトランザクションでオブジェクトが選択されてオブジェクト・キャッシュに入っている場合、そのオブジェクトが戻されることを意味します。シリアライズ可能モードで実行中のトランザクションの場合、このオプションの効果はLATESTと同じですが、ネットワークのラウンドトリップはそれほど多くありません。ほとんどのアプリケーションには、RECENTが適切です。

LATESTは、オブジェクトがオブジェクト・キャッシュに存在しない場合、データベースから取得されることを意味します。オブジェクト・キャッシュに存在する場合は、サーバーからリフレッシュされます。LATESTを使用する場合、ネットワークのラウンドトリップ数が最大になるため注意してください。LATESTは、オブジェクト・キャッシュとサーバーのバッファ・キャッシュをできるかぎり一致させる必要がある場合にのみ使用してください。

ANYは、オブジェクトがすでにオブジェクト・キャッシュに存在している場合、そのオブジェクトが戻されることを意味します。キャッシュになければ、そのオブジェクトはサーバーから取得します。ANYを指定すると、ネットワークのラウンドトリップ数は最小になります。この値を使用するのは、アプリケーションが読取り専用オブジェクトにアクセスする場合や、ユーザーがオブジェクトに排他的にアクセスする場合です。

11 マルチスレッド・アプリケーション

使用中の開発プラットフォームでスレッドがサポートされない場合、この章は無視してください。この章のトピックは、次のとおりです:

- スレッド
- Pro*C/C++のランタイム・コンテキスト
- ランタイム・コンテキストの使用モデル
- マルチスレッド・アプリケーションのユーザー・インタフェース機能
- マルチスレッドの例
- 接続プーリング

> 注意:
>
> Pro*C/C++プリコンパイラと XA を併用する場合は、XA のマルチスレッドを使用する必要があります。EXEC SQL ENABLE THREADS 文を使用して Pro*C/C++のマルチスレッドを使用するとエラーになります。

11.1 スレッド

マルチスレッド・アプリケーションでは、共有のアドレス空間で複数のスレッドが実行されます。スレッドはプロセス内で実行される軽量のサブプロセスです。コードとデータ・セグメントは共有しますが、独自のプログラム・カウンタ、マシン・レジスタおよびスタックがあります。グローバル変数と静的変数はすべてのスレッドに共通であり、通常、アプリケーション内の複数のスレッドからこれらの変数へのアクセスを管理するには、相互排他メカニズムが必要です。 Mutexは、データの整合性が保たれることを保証する同期化メカニズムです。

mutexの詳細は、マルチスレッドの説明を参照してください。マルチスレッド・アプリケーションの詳細は、スレッド・ファンクションのマニュアルを参照してください。

Pro*C/C++プリコンパイラは、マルチスレッドのOracleサーバー・アプリケーションの開発を(マルチスレッド・アプリケーションをサポートするプラットフォームで)サポートします。サポートされている機能は次のとおりです。

- スレッド・セーフなコードを生成するコマンドライン・オプション
- マルチスレッド処理をサポートする埋込みSQL文およびディレクティブ
- スレッド・セーフなSQLLIBと、その他のクライアント側Oracleライブラリ

> 注意:
>
> プラットフォームによりサポートするスレッド・パッケージが異なるため、使用しているプラットフォーム固有の Oracle マニュアルを参照して、Oracle がスレッド・パッケージをサポートしているかを調べてください。

次の各項目では、前述の機能を使用してマルチスレッドのPro*C/C++アプリケーションを開発する方法を説明します。

- マルチスレッド・アプリケーションのランタイム・コンテキスト
- ランタイム・コンテキストを使用する2つのモデル
- マルチスレッド・アプリケーションのユーザー・インタフェース
- Pro*C/C++を使用してマルチスレッド・アプリケーションを作成する場合のプログラミング上の考慮事項
- マルチスレッドのPro*C/C++アプリケーションの例

11.2 Pro*C/C++のランタイム・コンテキスト

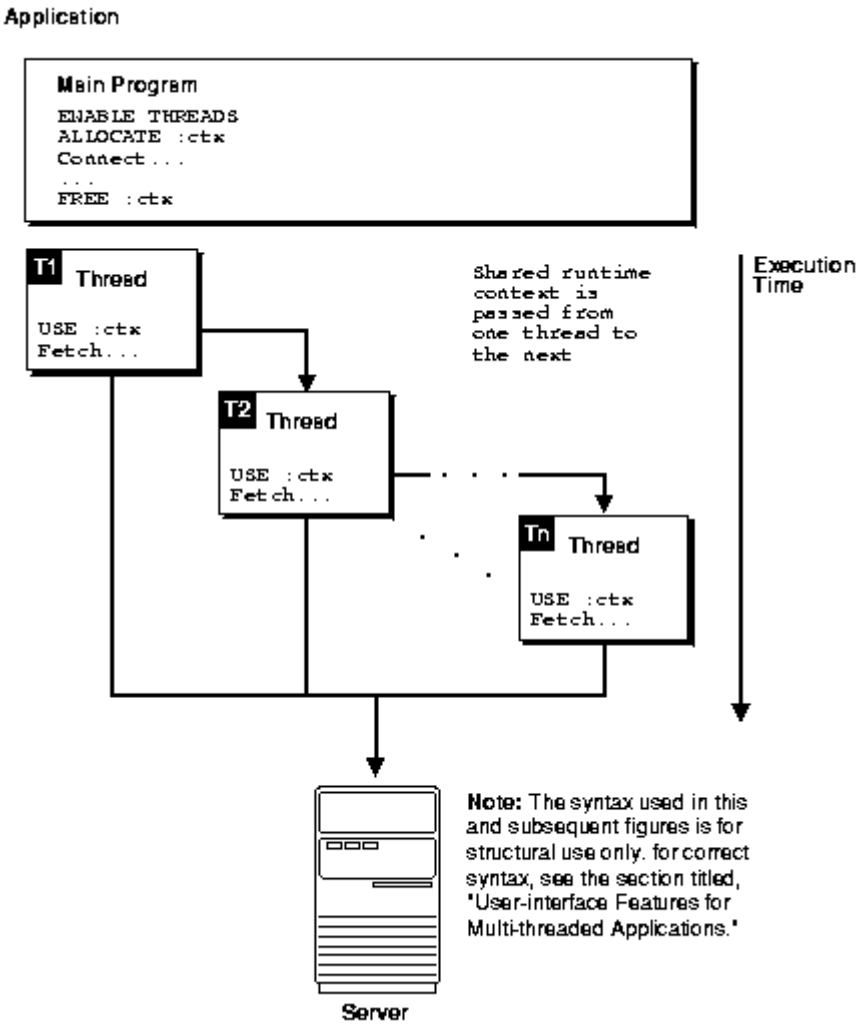
Pro*C/C++には、スレッドと接続を疎結合するために、ランタイム・コンテキストという概念が導入されています。ランタイム・コンテキストには、次のリソースおよびその現在の状態が含まれます。

- 1つ以上のOracleサーバーへの0個以上の接続
- サーバーへの接続に使用される0個以上のカーソル
- MODE、HOLD_CURSOR、RELEASE_CURSOR、SELECT_ERRORなどのインライン・オプション

Pro*C/C++プリコンパイラを使用すると、スレッドと接続を疎結合せずに、スレッドをランタイム・コンテキストに疎結合できます。また、Pro*C/C++では、アプリケーションでランタイム・コンテキストのハンドルを定義して、そのハンドルをあるスレッドから別のスレッドに渡すことができます。

たとえば、対話形式のアプリケーションで、スレッドT1を作成し、問合せを実行して先頭の10行をアプリケーションに戻します。その後、T1は終了します。必要なユーザー入力が取得されると、別のスレッドT2が作成され(または既存のスレッドが使用され)、T1のランタイム・コンテキストがT2に渡されます。T2は同じカーソルを処理して次の10行をフェッチできます。図11-1を参照してください。

図11-1 接続とスレッドの疎結合



11.3 ランタイム・コンテキストの使用モデル

マルチスレッドのPro*C/C++アプリケーションでランタイム・コンテキストを使用した2つの可能なモデルを次に示します。

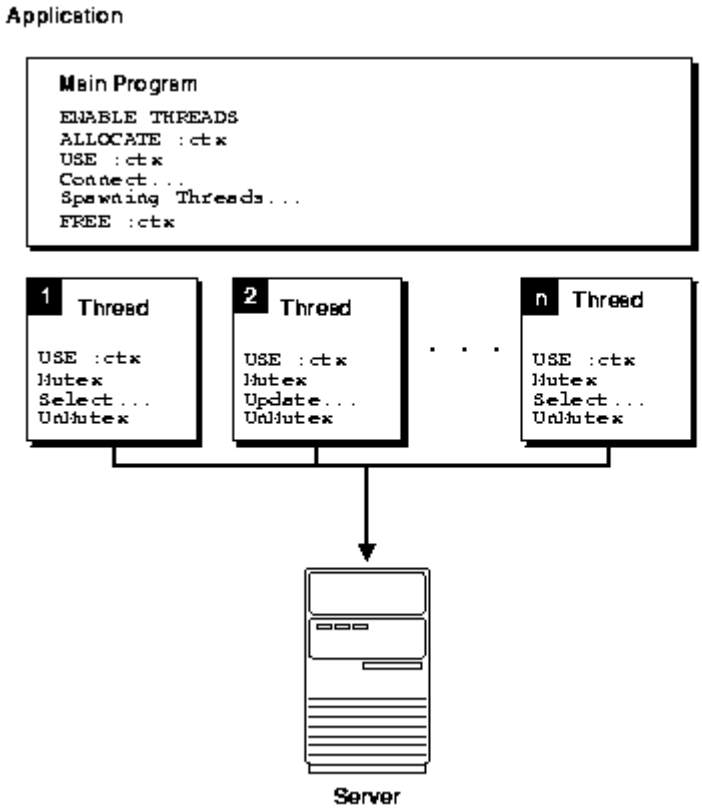
- 単一のランタイム・コンテキストを共有する複数のスレッド
- 複数のランタイム・コンテキストを使用する複数のスレッド

いずれのモデルを使用した場合も、複数のスレッドで同時に1つのランタイム・コンテキストを共有することはできません。複数のスレッドで同じランタイム・コンテキストを同時に使用すると、ランタイム・エラーが発生します。

11.3.1 複数のスレッドで1つのランタイム・コンテキストを共有

図11-2は、マルチスレッド環境で実行されるアプリケーションを示しています。1つ以上のSQL文を処理するために、様々なスレッドが1つのランタイム・コンテキストを共有します。この場合も、同時に複数のスレッドでランタイム・コンテキストを共有することはできません。図11-2のmutexは、同時使用を防ぐ方法を示しています。

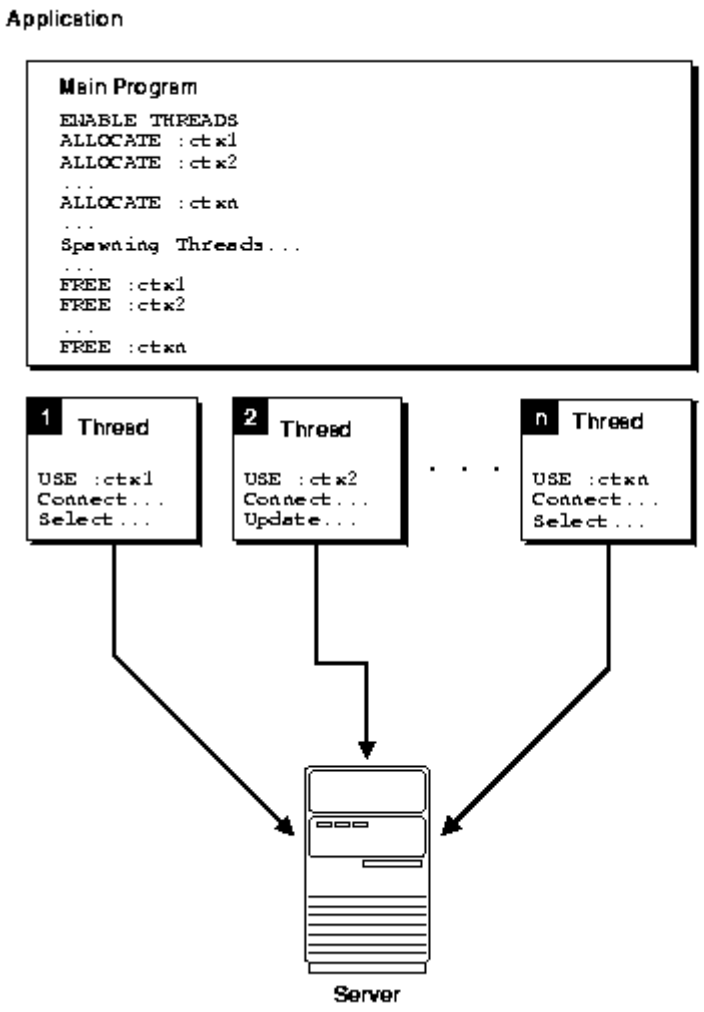
図11-2 スレッド間のコンテキスト共有



11.3.2 複数のスレッドで複数のランタイム・コンテキストを共有

図11-3に、複数のランタイム・コンテキストを使用して複数スレッドを実行するアプリケーションを示します。この場合、各スレッドは専用のランタイム・コンテキストを使用するため、アプリケーションにはmutexは必要ありません。

図11-3 スレッド間のコンテキスト非共有



11.4 マルチスレッド・アプリケーションのユーザー・インタフェース機能

Pro*C/C++プリコンパイラは、次に示すユーザー・インタフェース機能によってマルチスレッド・アプリケーションをサポートしています。

- コマンドライン・オプションTHREADS=YES|NO
- 埋込みSQL文とディレクティブ
- スレッド・セーフなSQLLIBパブリック関数

11.4.1 THREADSオプション

THREADS=YESをコマンドラインに指定すると、ガイドラインに従っている場合、生成されたコードがスレッド・セーフであることがPro*C/C++プリコンパイラにより保証されます。THREADS=YESと指定すると、Pro*C/C++ではすべてのSQL文がユーザー定義のランタイム・コンテキスト範囲内で実行されることが検証されます。プログラムがこの要件を満たしていないと、プリコンパイラ・エラーが戻されます。

関連項目

- プログラミングの考慮事項

11.4.2 埋込みSQL文およびディレクティブ

ランタイム・コンテキストおよびスレッドの定義と使用に対応した埋込みSQLおよびディレクティブは、次のとおりです。

- EXEC SQL ENABLE THREADS;

- EXEC SQL CONTEXT ALLOCATE :*context_var*;
- EXEC SQL CONTEXT USE { :*context_var* | DEFAULT};
- EXEC SQL CONTEXT FREE :*context_var*;

これらのEXEC SQL文では、*context_var*がランタイム・コンテキストへのハンドルで、次のように**sql_context**型として宣言する必要があります。

```
sql_context <context_variable>;
```

DEFAULTを使用すると、別のCONTEXT USE文でオーバーライドされるまで、以降のすべての埋込みSQL文にデフォルト(グローバル)ランタイム・コンテキストが使用されます。

11.4.2.1 EXEC SQL ENABLE THREADS

この実行SQL文は、複数のスレッドをサポートするプロセスを初期化します。このSQL文は、マルチスレッド・アプリケーション内の最初の実行SQL文にしてください。

> **注意:**
>
> Pro*C/C++プリコンパイラと XA を併用する場合は、XA のマルチスレッドを使用する必要があります。EXEC SQL ENABLE THREADS 文を使用して Pro*C のマルチスレッドを使用するとエラーになります。

関連項目

- ENABLE THREADS (実行可能埋込みSQL拡張機能)

11.4.2.2 EXEC SQL CONTEXT ALLOCATE

この実行SQL文は指定されたランタイム・コンテキストにメモリーを割り当てて初期化します。ランタイム・コンテキスト変数はsql_context型として宣言する必要があります。

関連項目

- CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)

11.4.2.3 EXEC SQL CONTEXT USE

このディレクティブはプリコンパイラに、後続の実行SQL文に対して指定したランタイム・コンテキストを使用するように指示します。ランタイム・コンテキストを指定するには、EXEC SQL CONTEXT ALLOCATE文で事前に割り当てる必要があります。

EXEC SQL CONTEXT USEディレクティブは、EXEC SQL WHENEVERディレクティブと同様に、指定したソース・ファイル内でこのディレクティブの後に続くすべての実行SQL文に影響し、C言語の標準のスコープ規則には従いません。次の例では、function2()のUPDATE文はグローバル・ランタイム・コンテキスト*ctx1*を使用しています。

```
sql_context ctx1;            /* declare global context ctx1     */

function1()
{
   sql_context :ctx1;         /* declare local context ctx1      */
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...   /* local ctx1 used for this stmt   */
   ...
```

```
}

function2()
{
   EXEC SQL UPDATE ...       /* global ctx1 used for this stmt */
}
```

ローカル・コンテキストを使用した後でグローバル・コンテキストを使用するには、次のコードをfunction1()に追加します。

```
function1()
{
   sql_context :ctx1;         /* declare local context ctx1      */
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...  /* local ctx1 used for this stmt   */
   EXEC SQL CONTEXT USE DEFAULT;
   EXEC SQL INSERT INTO ... /* global ctx1 used for this stmt   */
   ...
}
```

次の例に、グローバル・ランタイム・コンテキストはありません。プリコンパイラでは、UPDATE文に対して生成されたコードで*ctx1*ランタイム・コンテキストが参照されます。ただし、function2()のスコープにコンテキスト変数がないため、コンパイル時にエラーが発生します。

```
function1()
{
   sql_context ctx1;         /* local context variable declared */
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT USE :ctx1;
   EXEC SQL INSERT INTO ...     /* ctx1 used for this statement */
   ...
}
function2()
{
   EXEC SQL UPDATE ...    /* Error! No context variable in scope */
}
```

関連項目

- CONTEXT OBJECT OPTION GET(実行可能埋込みSQL拡張機能)
- CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)

11.4.2.4 EXEC SQL CONTEXT FREE

この実行SQL文は、指定したランタイム・コンテキストに関連付けられたメモリーを解放し、ホスト・プログラム変数にNULLポインタを入れます。

関連項目

- CONTEXT FREE (実行可能埋込みSQL拡張機能)

11.4.3 CONTEXT USEの例

次に示すコード例では、2つの典型的なプログラミング・モデルに埋込みSQL文とプリコンパイラ・ディレクティブを使用する方法を示しています。*thread_create()*を使用してスレッドを作成します。

最初の例では、複数のスレッドが複数のランタイム・コンテキストを使用する場合を示します。

```
main()
```

```
{
   sql_context ctx1,ctx2;           /* declare runtime contexts */
   EXEC SQL ENABLE THREADS;
   EXEC SQL CONTEXT ALLOCATE :ctx1;
   EXEC SQL CONTEXT ALLOCATE :ctx2;
   ...
/* spawn thread, execute function1 (in the thread) passing ctx1 */
   thread_create(..., function1, ctx1);
/* spawn thread, execute function2 (in the thread) passing ctx2 */
   thread_create(..., function2, ctx2);
   ...
   EXEC SQL CONTEXT FREE :ctx1;
   EXEC SQL CONTEXT FREE :ctx2;
   ...
}

void function1(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
/* execute executable SQL statements on runtime context ctx1!!! */
   ...
}

void function2(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
/* execute executable SQL statements on runtime context ctx2!!! */
   ...
}
```

次の例では、共通のランタイム・コンテキストを共有する複数のスレッドを使用する方法を示します。function1()およびfunction2()で実行されるSQL文は同時に実行される可能性があるため、すべての実行EXEC SQL文をmutexで囲むことで、データ操作を逐次的、つまり安全に行うことが必要です。

```
main()
{
   sql_context ctx;                  /* declare runtime context */
   EXEC SQL CONTEXT ALLOCATE :ctx;
   ...
/* spawn thread, execute function1 (in the thread) passing ctx  */
   thread_create(..., function1, ctx);
/* spawn thread, execute function2 (in the thread) passing ctx  */
   thread_create(..., function2, ctx);
   ...
}

void function1(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
/* Execute SQL statements on runtime context ctx.               */
   ...
}

void function2(sql_context ctx)
{
   EXEC SQL CONTEXT USE :ctx;
/* Execute SQL statements on runtime context ctx.               */
   ...
}
```

11.4.4 プログラミングの考慮事項

OracleはSQLLIBコードがスレッド・セーフであることを保証しますが、Pro*C/C++のソース・コードがスレッドで正しく動作するように設計する必要があります。たとえば、静的変数とグローバル変数は慎重に使用してください。

また、マルチスレッド・アプリケーションの設計時には次の点に注意してください。

- SQLCAをスレッド・セーフな構造体として宣言します。通常は自動変数として、ランタイム・コンテキストごとに1つずつ宣言します。
- SQLDAをスレッド・セーフな構造体として宣言します(SQLCAと同様)。通常は、自動変数としてランタイム・コンテキストごとに1つずつ宣言します。
- スレッド・セーフ方式でホスト変数を宣言するときは、静的ホスト変数およびグローバル・ホスト変数の使用に注意してください。
- 複数のスレッドでランタイム・コンテキストを同時に使用するのを避けます。
- デフォルトのデータベース接続を使用するか、あるいはAT句を使用して明示的に定義するかを判断します。

さらに、複数の埋込みSQLの実行文(EXEC SQL UPDATEなど)をランタイム・コンテキストで同時に未解決にしないでください。

プリコンパイル済アプリケーションの既存の要件も適用されます。たとえば、特定のカーソルへの参照はすべて同じソース・ファイルに含まれている必要があります。

11.5 マルチスレッドの例

次のプログラムは、マルチスレッドの埋込みSQLアプリケーションを作成する1つの方法です。このプログラムでは、スレッドと同じ数のセッションが作成されます。それぞれのスレッドは0個以上のトランザクションを実行します。これはレコードと呼ばれる一時的な構造体によって指定されます。

> 注意:
>
> このプログラムは Solaris が動作している Sun 社のワークステーション専用に開発されています。 このプログラムでは、DCE または Solaris のスレッド・パッケージを使用できます。スレッド・パッケージの可用性は、プラットフォーム固有のマニュアルを参照してください。

```
/*
```

* Name: Thread_example1.pc * * Description: This program illustrates how to use threading in * conjunction with precompilers. The program creates as many * sessions as there are threads. Each thread executes zero or * more transactions, that are specified in a transient * structure called 'records'. * Requirements: * The program requires a table 'ACCOUNTS' to be in the schema * scott/tiger. The description of ACCOUNTS is: * SQL> desc accounts * Name Null? Type * ---------- ------- ------ * ACCOUNT NUMBER(36) * BALANCE NUMBER(36,2) * * For proper execution, the table should be filled with the accounts * 10001 to 10008. * * */ #include <stdio.h> #include <stdlib.h> #include <string.h> #include <sqlca.h> #define _EXC_OS_ _EXC__UNIX #define _CMA_OS_ _CMA__UNIX #ifdef DCE_THREADS #include <pthread.h> #else #include <thread.h> #endif /* Function prototypes */ void err_report(); #ifdef DCE_THREADS void do_transaction(); #else void *do_transaction(); #endif void get_transaction(); void logon();

```
void logoff(); #define CONNINFO "scott/tiger" #define THREADS 3 struct parameters
{ sql_context * ctx; int thread_id; }; typedef struct parameters parameters; struct record_log
{ char action; unsigned int from_account; unsigned int to_account; float amount; }; typedef struct
record_log record_log; record_log records[]= { { 'M', 10001, 10002, 12.50 }, { 'M', 10001, 10003,
25.00 }, { 'M', 10001, 10003, 123.00 }, { 'M', 10001, 10003, 125.00 }, { 'M', 10002, 10006,
12.23 }, { 'M', 10007, 10008, 225.23 }, { 'M', 10002, 10008, 0.70 }, { 'M', 10001, 10003,
11.30 }, { 'M', 10003, 10002, 47.50 }, { 'M', 10002, 10006, 125.00 }, { 'M', 10007, 10008,
225.00 }, { 'M', 10002, 10008, 0.70 }, { 'M', 10001, 10003, 11.00 }, { 'M', 10003, 10002,
47.50 }, { 'M', 10002, 10006, 125.00 }, { 'M', 10007, 10008, 225.00 }, { 'M', 10002, 10008,
0.70 }, { 'M', 10001, 10003, 11.00 }, { 'M', 10003, 10002, 47.50 }, { 'M', 10008, 10001,
1034.54}}; static unsigned int trx_nr=0; #ifdef DCE_THREADS pthread_mutex_t mutex; #else
mutex_t mutex; #endif
/********************************************************************** * Main
**********************************************************************/ main()
{ sql_context ctx[THREADS]; #ifdef DCE_THREADS pthread_t thread_id[THREADS];
pthread_addr_t status; #else thread_t thread_id[THREADS]; int status; #endif parameters
params[THREADS]; int i; EXEC SQL ENABLE THREADS; EXEC SQL WHENEVER SQLERROR DO
err_report(sqlca); /* Create THREADS sessions by connecting THREADS times */
for(i=0;i<THREADS;i++) { printf("Start Session %d....",i); EXEC SQL CONTEXT ALLOCATE :ctx[i];
logon(ctx[i],CONNINFO); } /*Create mutex for transaction retrieval */ #ifdef DCE_THREADS if
(pthread_mutex_init(&mutex,pthread_mutexattr_default)) #else if (mutex_init(&mutex,
USYNC_THREAD, NULL)) #endif { printf("Can't initialize mutex¥n"); exit(1); } /*Spawn threads*/
for(i=0;i<THREADS;i++) { params[i].ctx=ctx[i]; params[i].thread_id=i; printf("Thread %d... ",i);
#ifdef DCE_THREADS if (pthread_create(&thread_id[i],pthread_attr_default,
(pthread_startroutine_t)do_transaction, (pthread_addr_t) &params[i])) #else if (status =
thr_create (NULL, 0, do_transaction, &params[i], 0, &thread_id[i])) #endif printf("Cant create
thread %d¥n",i); else printf("Created¥n"); } /* Logoff sessions....*/ for(i=0;i<THREADS;i++)
{ /*wait for thread to end */ printf("Thread %d ....",i); #ifdef DCE_THREADS if
(pthread_join(thread_id[i],&status)) printf("Error when waiting for thread % to terminate¥n", i);
else printf("stopped¥n"); printf("Detach thread..."); if (pthread_detach(&thread_id[i]))
printf("Error detaching thread! ¥n"); else printf("Detached!¥n"); #else if (thr_join(thread_id[i],
NULL, NULL)) printf("Error waiting for thread to terminate¥n"); #endif printf("Stop
Session %d....",i); logoff(ctx[i]); EXEC SQL CONTEXT FREE :ctx[i]; } /*Destroys mutex*/ #ifdef
DCE_THREADS if (pthread_mutex_destroy(&mutex)) #else if (mutex_destroy(&mutex)) #endif
{ printf("Can't destroy mutex¥n"); exit(1); } }
/********************************************************************** *
Function: do_transaction * * Description: This functions executes one transaction out of the *
records array. The records array is 'managed' by * the get_transaction function. * *
**********************************************************************/ #ifdef
DCE_THREADS void do_transaction(params) #else void *do_transaction(params) #endif
parameters *params; { struct sqlca sqlca; record_log *trx; sql_context ctx=params->ctx; /* Done
all transactions ? */ while (trx_nr < (sizeof(records)/sizeof(record_log))) { get_transaction(&trx);
EXEC SQL WHENEVER SQLERROR DO err_report(sqlca); EXEC SQL CONTEXT USE :ctx;
printf("Thread %d executing transaction¥n",params->thread_id); switch(trx->action) { case 'M':
EXEC SQL UPDATE ACCOUNTS SET BALANCE=BALANCE+:trx->amount WHERE ACCOUNT=:trx-
```

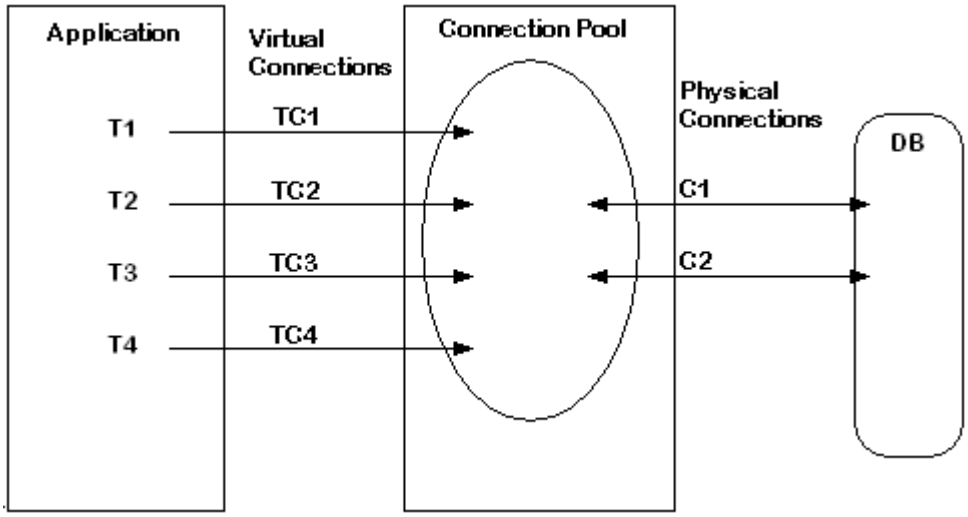
>to_account; EXEC SQL UPDATE ACCOUNTS SET BALANCE=BALANCE-:trx->amount WHERE ACCOUNT=:trx->from_account; break; default: break; } EXEC SQL COMMIT; } }
/** * Function: err_report * * Description: This routine prints out the most recent error *
**/ void err_report(sqlca) struct sqlca sqlca; { if (sqlca.sqlcode < 0) printf("¥n%.*s¥n¥n",sqlca.sqlerrm.sqlerrml,sqlca.sqlerrm.sqlerrmc); exit(1); }
/** * Function: logon * * Description: Logs on to the database as USERNAME/PASSWORD *
**/ void logon(ctx,connect_info) sql_context ctx; char * connect_info; { EXEC SQL WHENEVER SQLERROR DO err_report(sqlca); EXEC SQL CONTEXT USE :ctx; EXEC SQL CONNECT :connect_info; printf("Connected!¥n"); }
/** * Function: logoff * * Description: This routine logs off the database *
**/ void logoff(ctx) sql_context ctx; { EXEC SQL WHENEVER SQLERROR DO err_report(sqlca); EXEC SQL CONTEXT USE :ctx; EXEC SQL COMMIT WORK RELEASE; printf("Logged off!¥n"); }
/** * Function: get_transaction * * Description: This routine returns the next transaction to process *
**/ void get_transaction(trx) record_log ** trx; { #ifdef DCE_THREADS if (pthread_mutex_lock(&mutex)) #else if (mutex_lock(&mutex)) #endif printf("Can't lock mutex¥n"); *trx=&records[trx_nr]; trx_nr++; #ifdef DCE_THREADS if (pthread_mutex_unlock(&mutex)) #else if (mutex_unlock(&mutex)) #endif printf("Can't unlock mutex¥n"); }

11.6 接続プーリング

接続プールとは、いくつかの接続間で再使用できる、データベースへの物理接続グループです。接続プーリング機能の目的は、それぞれの接続に専用接続を使用させないことでパフォーマンスを改善し、リソースの使用を削減することです。

図11-4は、接続プーリングの機能を示します。この例では、アプリケーションの4つのスレッドが接続プールを使用してデータベースと対話しています。この接続プールには物理接続が2つあります。異なるランタイム・コンテキストを使用する4つのスレッドにより、接続プール・ハンドルが使用されます。

図11-4 接続プーリング




```
thread1()
{
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT USE:ctxl;
EXEC SQL CONNECT :uid  AT :TC1 USING :db_string;
...

}

thread2()
{
EXEC SQL CONTEXT ALLOCATE :ctx2;
EXEC SQL CONNECT :uid  AT :TC2 USING :db_string;
...

}

thread3()
{
EXEC SQL CONTEXT ALLOCATE :ctx3;
EXEC SQL CONNECT :uid  AT :TC3 USING :db_string;
EXEC SQL AT :TC3 SELECT count(*) into :count FROM emp;
...

}

thread4()
{
EXEC SQL CONTEXT ALLOCATE :ctx4;
EXEC SQL CONNECT :uid  AT :TC4 USING :db_string;
...

}
```

この例で、TC1、TC2、TC3およびTC4という4つの名前付き接続は、それぞれT1、T2、T3およびT4というスレッドにより作成された仮想接続です。異なるランタイム・コンテキストからの名前付き接続TC1、TC2、TC3およびTC4が1つの接続プールを共有し、接続プール内で使用可能な物理データベース接続を共有します。C1およびC2という2つの物理接続が、4つの名前付き接続にサービスを提供し、同じ1つのデータベースに接続します。

スレッドT1から最初の接続要求TC1を受け取ると、SQLLIBはデータベースへの物理接続C1を1つ含む接続プールを作成します。スレッドT2から別の接続要求TC2が同一データベースに送信されると、C1が空いている場合はデータベースへのTC2要求に対してC1がサービスを提供します。C1が空いていない場合は、要求にサービスを提供するために新しい物理接続C2が作

成されます。スレッドT3からTC3という別の接続要求が受信された場合、物理接続C1とC2がどちらも使用中の場合は、TC3は指定された時間待機するか、エラー・メッセージを返します。

スレッドT2が名前付き接続TC2を使用してデータを選択する必要があるときは、物理接続C1またはC2の空いている方を取得します。要求に対してサービスが提供された後、選択されていた接続は接続プール内で再び使用可能になるため、別の名前付き接続または仮想接続がその物理接続を利用できます。

11.6.1 接続プーリング機能の使用方法について

この項には、次の項目が含まれます。

- 接続プーリングを使用可能にする方法
- 接続プーリングのコマンドライン・オプション
- 例
- パフォーマンス・チューニング

11.6.1.1 接続プーリングを使用可能にする方法

アプリケーションのプリコンパイル中に接続プーリングを使用可能にするには、CPOOL=YESコマンドライン・オプションを設定する必要があります。CPOOL=YES/NOの設定に基づいて、接続プーリング機能が使用可能または使用禁止になります。

注意:

- デフォルトでは、CPOOL は NO に設定されます。つまり、接続プーリング機能は使用禁止です。この機能をインラインで使用可能または禁止にすることはできません。
- CPOOL が YES に設定されていても、外部オペレーティング・システム認証では接続プールは作成されません。

11.6.1.2 接続プーリングのコマンドライン・オプション

表11-1に、接続プーリングのコマンドライン・オプションの一覧を示します。

表11-1 接続プーリングのコマンドライン・オプション

オプション	有効な値	デフォルト値	備考
CPOOL	YES または NO	NO	このオプションに基づき、プリコンパイラでは、SQLLIB に接続プール機能を有効または無効にするように指示する適切なコードを生成します。 **注意:** このオプションが NO に設定されている場合、プリコンパイラはその他の接続プーリング・オプションを無視します。
CMAX	有効値は 1 から 65535 です。	100	データベースに対してオープンできる物理接続の最大数を指定します。CMAX 値は、CMIN+CINCR より大きく設定する必要があり

オプション	有効な値	デフォルト値	備考
			ます。 **注意:** この値に達すると、物理接続をそれ以上オープンできません。 通常のアプリケーションでは、同時データベース操作を 100 個実行するように設定すれば十分です。ユーザーが適切な値を設定できます。
CMIN	有効値は 1 から (CMAX-CINCR)です。	2	接続プール内の最小物理接続数を指定します。最初は、CMIN により指定されたとおりに物理接続がすべてサーバーに対してオープンされます。それ以降は、必要な場合にのみ物理接続がオープンされます。パフォーマンスを最適にするには、CMIN を、アプリケーションによる実行が計画または予想される同時実行文の合計数に設定する必要があります。デフォルト値は 2 に設定されます。
CINCR	有効値は 1 から (CMAX-CMIN) です。	1	現在の物理接続数が CMAX 値より少ない場合、データベースに対してオープンされる物理接続数の次の増分をアプリケーションで設定できるようにします。不要な数の接続が作成されないように、デフォルト値は 1 に設定されています。
CTIMEOUT	有効値は 1 から 65535 です。	0(未設定)。つまり、タイムアウトになりません。	指定された期間(秒単位)より長い間アイドル状態になっている物理接続を終了し、オープンされている物理接続数を最適に保ちます。この属性を設定しない場合、物理接続はタイムアウトしません。このため、接続プールが終了するまで物理接続はクローズしません。 **注意:** 物理接続を新しく作成すると、サーバーへのラウンドトリップが必要になります。
CNOWAIT	有効値は 1 から 65535 です。	0(未設定)。つまり、接続が空くのを待機します。	この属性は、プール内の他のすべての物理接続が使用中で、物理接続の合計数がすでに最大値に達している場合に、アプリケーションで繰り返し物理接続を要求する必要があるかどうかを決定します。物理接続が使用できず、これ以上物理接続をオープンできない場合、この属性が設定されているとエラーが発生します。そうでない場合、コールは別の接続が取得されるまで待機します。デフォルトでは、CNOWAIT は設定されないため、スレッドは、エラーを戻すかわりに、空いている接続を取得できるまで待機します。

通常のマルチスレッド・アプリケーションでは、n個の物理接続を持つプールを作成します。nの値は、プリコンパイル中にCMIN値で指定する必要があります。最初の接続コールでは、データベースへの最小数(CMIN)の物理接続が作成されます。新しい要求に対しては、仮想接続(名前付き接続)から物理接続へのマッピングが実行されます。これを次の項で説明します。

ケース1: 物理接続が(すでにオープンされている接続の中で)使用可能な場合、新しい要求にはこの接続がサービスを提供し

ます。

ケース2: 物理接続がすべて使用中の場合

ケース2a: オープンされている接続の数が最大数制限(CMAX)に達していない場合は、CINCR分の新しい接続が作成され、その中の1つが要求の処理に使用されます。

ケース2b: オープンされている物理接続数が最大数(CMAX)に達していて、CNOWAITが設定されていない場合、要求は接続を取得できるまで待機します。それ以外の場合は、「*ORA 24401:* これ以上の接続は開けません」というエラー・メッセージが表示されます。

11.6.1.3 例

次の例の説明は、図11-4を参照してください。

```
Let
CMIN be 1,
CMAX be 2, and
CINCR be 1.
```

次のシナリオについて考えます。最初の要求TC1を受け取ると、SQLLIBは物理接続C1を1つ含む接続プールを作成します。別の要求TC2を受け取ると、アプリケーションはC1が空いているかどうかをチェックします。C1は最初の要求の処理に使用されているため(ケース1)、要求にサービスを提供するために新しい物理接続C2が作成されます(ケース2a)。別の要求TC3を受け取ったときに、C1もC2も使用中の場合、TC3は指定された時間待機するか、エラー・メッセージ付きで戻されます(ケース2b)。

11.6.1.4 パフォーマンス・チューニング

パフォーマンスを向上するために、アプリケーションに応じて接続プーリングのパラメータを設定できます。図11-5のパフォーマンス・グラフは、Pro*C/C++のデモ・プログラム:1のCMIN値を変更することで、パフォーマンスが向上することを示しています。デモ・プログラム:2は、CMAXパラメータを変更することで、パフォーマンスが向上することを示しています。

11.6.2 デモ・プログラム: 1

デモ・プログラム: 1のプリコンパイル中には、次の接続プール・パラメータが使用されます。

```
CMAX = 40
CINCR = 3
CMIN = varying values between 1 to 40
CPOOL = YES
CTIMEOUT - Do not set
```

(物理接続がタイムアウトしないことを示します。)

```
CNOWAIT - Do not set
```

(空いている接続を取得するまでスレッドが待機することを示します。詳細は、表11-1を参照してください。)

この例に必要なその他のコマンドライン・オプションは、次の項で示します。

```
threads = yes
```

注意:

この例では、スレッド数は 40 で、データベース操作はローカル・データベースに対して実行されます。

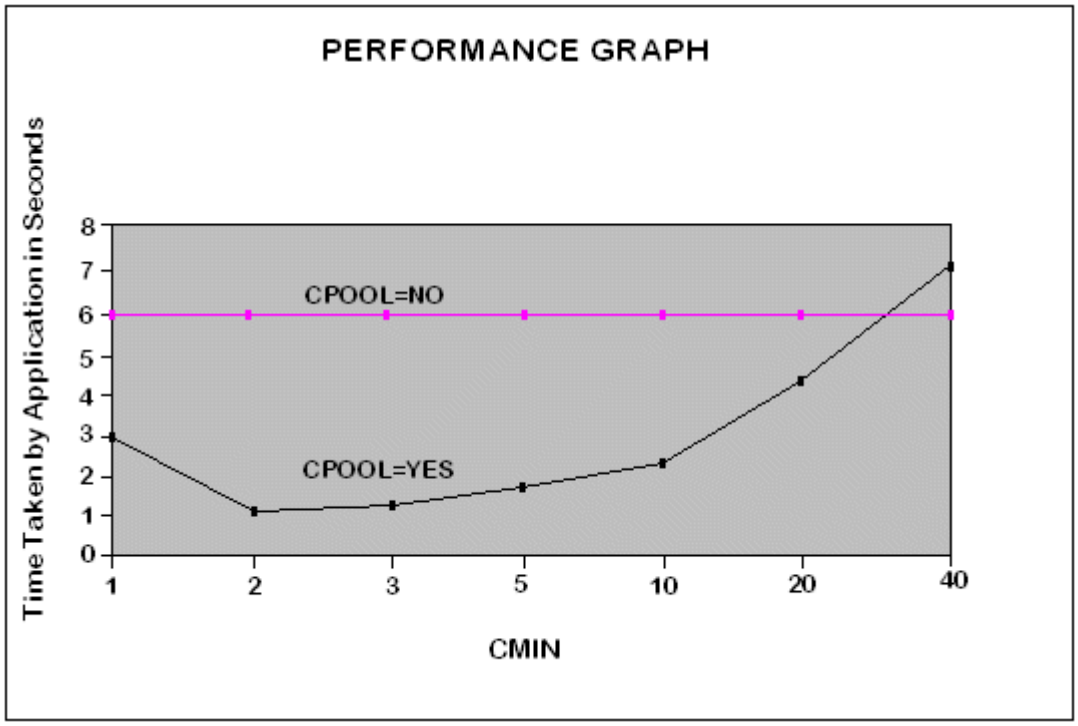
CPOOL=NO(接続プーリングを使用しない)を指定した場合、アプリケーションの所要時間は6.1秒でした。これに対し、CPOOL=YES(接続プーリングを使用する)を指定すると、アプリケーションの最小所要時間は1.3秒になりました(CMIN=2の場合)。

どちらの場合も、接続プールによって短縮されるのはCONNECT文の所要時間のみであるため、データベース問合せ操作に要する時間は変わりません。CPOOL=NOを指定した場合、アプリケーションは40個の専用接続を作成します。CPOOL=YESとCMIN=2を指定した場合は、最初に2つの接続を作成し、2つのスレッドが接続に同時にアクセスする場合にのみ、これ以上の接続を作成します。そうでない場合は、すべてのスレッドがこの2つの接続を共有します。このため、アプリケーションでは潜在的に38個の接続が回避され、これにより、接続確立のためのサーバーへのラウンドトリップが38個回避されます。その結果、パフォーマンスは3倍に向上します。

注意:

前述の結果は、Solaris 2.6 オペレーティング・システム上で Oracle サーバーが 1 つ稼働する、単一 CPU および 256MB RAM 搭載の Sparc Ultra60 マシンでの測定値です。サーバーとクライアントは同一マシン上で実行されています。

図11-5 パフォーマンス・グラフ



CPOOL=YESの線は、接続プーリングが使用可能になっているときのアプリケーションの所要時間を表します。CPOOL=NOの線は、接続プーリングが使用禁止になっているときのアプリケーションの所要時間を表します。

11.6.2.1 例

```
/*
*  cpdemo1.pc
*
* Description:
*      The program creates as many sessions as there are threads.
*      Each thread connects to the default database and executes the
*      SELECT statement 5 times.  Each thread has its own runtime context.
```

```
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlca.h>

#define      _EXC_OS_        _EXC__UNIX
#define      _CMA_OS_        _CMA__UNIX

#ifdef DCE_THREADS
#include <pthread.h>
#else
#include <pthread.h>
typedef void*       pthread_addr_t;
typedef void*      (*pthread_startroutine_t) (void*);
#define pthread_attr_default  (const pthread_attr_t *)NULL
#endif

/* Function prototypes */
void   err_report();
void   do_transaction();
void   get_transaction();
void   logon();
void   logoff();

#define CONNINFO "hr/hr"
#define THREADS  40

struct parameters
  {
   sql_context * ctx;
   int thread_id;
  };
typedef struct parameters parameters;

struct timeval tp1;
struct timeval tp2;

/*************************************
*  Main
*************************************/

main()
{
  sql_context ctx[THREADS];
  pthread_t thread_id[THREADS];
  pthread_addr_t status;
  parameters params[THREADS];
  int i;

  EXEC SQL ENABLE THREADS;
  EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);

  if(gettimeofday(&tp1, (void*)NULL) == -1)
  {
    perror("First: ");
    exit(0);
  }
```

```
  /* Create THREADS sessions by connecting THREADS times */
  for(i=0;i<THREADS;i++)
  {
    printf("Start Session %d....",i);
    EXEC SQL CONTEXT ALLOCATE :ctx[i];
    logon(ctx[i],CONNINFO);
  }

  /*Spawn threads*/
  for(i=0;i<THREADS;i++)
  {
    params[i].ctx=ctx[i];
    params[i].thread_id=i;

    if (pthread_create(&thread_id[i],pthread_attr_default,
      (pthread_startroutine_t)do_transaction,
      (pthread_addr_t) &params[i]))
      printf("Cant create thread %d¥n",i);
    else
      printf("Created Thread %d¥n", i);
  }

  /* Logoff sessions....*/
  for(i=0;i<THREADS;i++)
  {
    /*wait for thread to end */
    if (pthread_join(thread_id[i],&status))
      printf("Error when waiting for thread % to terminate¥n", i);
    else
      printf("stopped¥n");

    if(i==THREADS-1)
    {
      logoff(ctx[i]);
      EXEC SQL CONTEXT FREE :ctx[i];
    }
  }

  if(gettimeofday(&tp2, (void*)NULL) == -1)
  {
    perror("Second: ");
    exit(0);
  }

    printf(" ¥n¥nTHE TOTAL TIME TAKEN FOR THE PROGRAM EXECUTION = %f ¥n¥n",
    (float)(tp2.tv_sec - tp1.tv_sec) + ((float)(tp2.tv_usec -
    tp1.tv_usec)/1000000.0));


}

/*********************************************************************
* Function: do_transaction
* Description:  This functions executes SELECT 5 times and calls COMMIT.
*********************************************************************/
void do_transaction(params)
parameters *params;
{
  struct sqlca sqlca;
```

```
  int src_count;
  sql_context ctx=params->ctx;

  EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);
  EXEC SQL CONTEXT USE :ctx;
  printf("Thread %d executing transaction¥n",params->thread_id);
  EXEC SQL COMMIT;
  EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
  EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
  EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
  EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
  EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
}

/*************************************************************
* Function: err_report
* Description: This routine prints out the most recent error
*************************************************************/
void      err_report(sqlca)
struct sqlca sqlca;
{
  if (sqlca.sqlcode < 0)
    printf("¥n%.*s¥n¥n",sqlca.sqlerrm.sqlerrml,sqlca.sqlerrm.sqlerrmc);
  exit(1);
}

/*************************************************************
* Function: logon
* Description: Logs on to the database as USERNAME/PASSWORD
*************************************************************/
void      logon(ctx,connect_info)
sql_context ctx;
char * connect_info;
{
  EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);
  EXEC SQL CONTEXT USE :ctx;
  EXEC SQL CONNECT :connect_info;
  printf("Connected!¥n");
}

/*************************************************
* Function: logoff
* Description: This routine logs off the database
*************************************************/
void      logoff(ctx)
sql_context ctx;
{
  EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);
  EXEC SQL CONTEXT USE :ctx;
  EXEC SQL COMMIT WORK RELEASE;
  printf("Logged off!¥n");
}
```

11.6.3 デモ・プログラム: 2

デモ・プログラム: 2のプリコンパイル中に、次の接続プーリング・パラメータが使用されます。

CMAX = 5から40の可変値

CINCR = 3

CMIN = 1から40の可変値

CPOOL = YES

CTIMEOUT-設定しない

(物理接続がタイムアウトしないことを示します。)

CNOWAIT-設定しない

(空いている接続を取得するまでスレッドが待機することを示します。詳細は、表11-1を参照してください。)

この例に必要なその他のコマンドライン・オプションは、次の項で示します。

```
threads = yes
```

次の図は、cpdemo2のパフォーマンス・グラフを示します。

注意:

この例では、スレッド数は 40 で、データベース操作はローカル・データベースに対して実行されます。

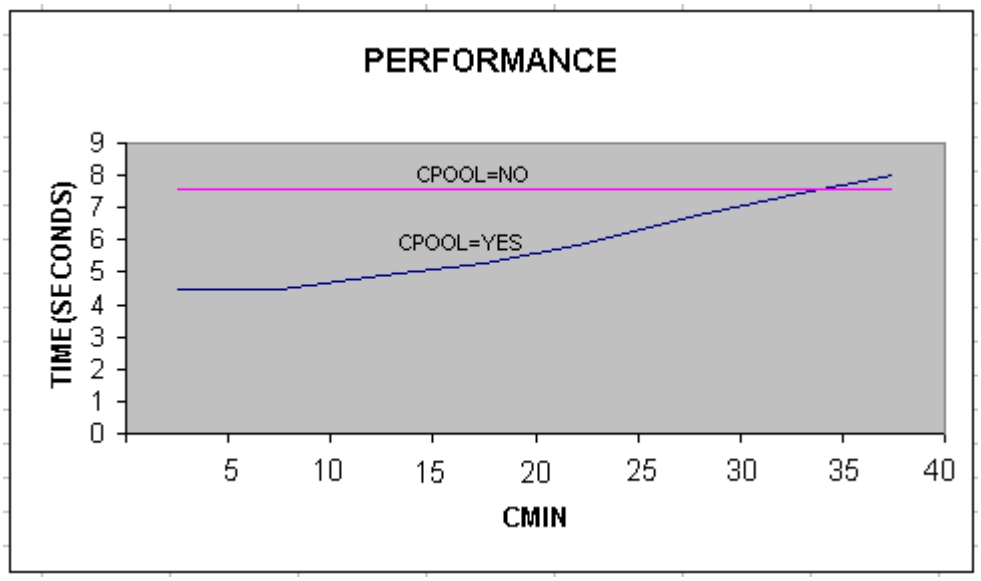
この例では、CMIN=5、CMAX=14のときに、CPOOL=NOを使用した場合と比べてプログラムの実行速度が約2.3倍に向上し、最善のパフォーマンスが得られます。ただし、接続プーリングを使用可能にすることで実行がさらに高速化する「cpdemo1」ほどの向上は見られません。これは、「cpdemo1」では単純なSELECT文のみを実行しているのに対し、「cpdemo2」ではUPDATE文とSELECT文の両方を実行しているためです。したがって、「cpdemo1」ではデータベース操作の実行よりも接続の作成に時間がかかっています。接続プーリングを使用可能にすると、作成される接続の数が減り、時間が短縮されます。その結果、全体的なパフォーマンスが向上します。「cpdemo2」では、データベース操作の実行に比べて接続の作成にかかる時間が少ないため、全体的なパフォーマンス向上の度合いは低くなります。

次のグラフでは、CPOOL=YESの線は、接続プーリングが使用可能になっているときのアプリケーションの所要時間を表します。CPOOL=NOの線は、接続プーリングが使用禁止になっているときのアプリケーションの所要時間を表します。デモ・プログラム「cpdemo2」では40個のスレッドを作成します。CPOOL=NOオプションの場合は、スレッドはそれぞれサーバーに対する専用接続を確立します。このため、40個の接続が作成されます。CPOOL=YESとCMAX=14を指定して同じデモ・プログラムをビルドすると、最大14の接続が作成されます。これらの接続は40個のスレッドで共有されるため、少なくとも26個の接続が節約され、サーバーへの26回のラウンドトリップが回避されます。

次の2つのグラフは、CMINとCMAXの値をそれぞれ変化させた場合のパフォーマンスを示しています。

11.6.3.1 ケース1: CMINを変更

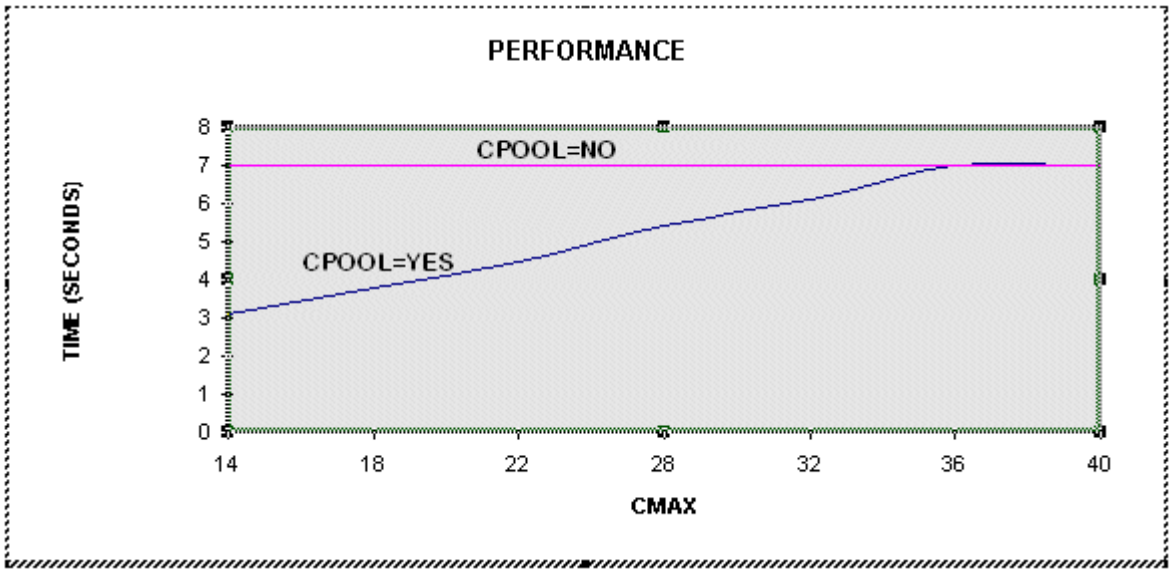
図11-6 ケース1のパフォーマンス・グラフ



CPOOL=NOの場合、アプリケーションでは実行に約7.5秒かかります。CPOOL=YESで、CMIN=8およびCMAX=14の場合、実行時間は4.5秒に短縮されます。このため、パフォーマンスの改善は約1.7倍になります。このパフォーマンスの相違は、データベース操作が違う(SELECT対UPDATE)のためです。これは純粋にサーバー側のアクティビティであり、クライアント側機能である接続プール機能の範囲外です。

11.6.3.2 ケース2: CMAXを変更

図11-7 ケース2のパフォーマンス・グラフ



前述のグラフの場合、デモ・プログラムはCMIN=5およびCINCR=3で実行されました。最善のパフォーマンスはCMAX=14の場合に得られます。CPOOL=NOの場合は実行に約7.4秒かかっています。CPOOL=YESでCMAX=14の場合、実行時間は約3.1秒まで短縮され、パフォーマンスは2.3倍向上しています。

パフォーマンス向上の度合いはCMAXによって異なります。したがって、特定のアプリケーションで最善のパフォーマンスを得るには、最適なパフォーマンスに達するまでCMINとCMAXを変化させる必要があります。

11.6.3.3 例

```
/*
*  cpdemo2.pc
  * Program to show the performance imcrement when the cpool option is used
  * Run this program with cpool=no. Record the time taken for the program to
  * execute
  *
```

```
  * Compare the execution time
  *
  * This program also demonstrates the impact of a properly tuned CMAX
  * parameter on the performance
  *
  * Run the program with the following parameter values
  *
  * CMIN=5
  * CINCR=2
  * CMAX=20
  *
  */

#include <stdio.h>
#include <sqlca.h>

#ifdef DCE_THREADS
#include <pthread.h>
#else
#include <sys/time.h>
#include <pthread.h>
typedef void*       pthread_addr_t;
typedef void*      (*pthread_startroutine_t) (void*);
#define pthread_attr_default  (const pthread_attr_t *)NULL
#endif


#define CONNINFO "hr/hr"
#define THREADS 40

/***** prototypes ************** */
void selectFunction();
void updateFunction();

void err_report(struct sqlca sqlca);
/* ************************ */

/***** parameter to the function selectFunction, updateFunction */
struct parameters
{
  sql_context ctx;
  char connName[20];
  char dbName[20];
  int thread_id;
};
typedef struct parameters parameters;
/*****************************************/

parameters params[THREADS];

struct timeval tp1;
struct timeval tp2;

int main()
{
  int i;
  pthread_t thread_id[THREADS];
  pthread_addr_t status;

  int thrNos[THREADS];
```

```
for(i=0; i<THREADS; i++)
  thrNos[i] = i;

EXEC SQL ENABLE THREADS;

/* Time before executing the program */
if(gettimeofday(&tp1, (void*)NULL) == -1){
  perror("First: ");
  exit(0);
}

EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);
/* connect THREADS times to the data base */
for(i=0; i<THREADS; i++)
{
  strcpy(params[i].dbName, "");
  sprintf(params[i].connName,"conn%d", i);
  params[i].thread_id = i;

  /* logon to the data base */
  EXEC SQL CONTEXT ALLOCATE :params[i].ctx;
  EXEC SQL CONTEXT USE :params[i].ctx;
  EXEC SQL CONNECT :CONNINFO
    AT :params[i].connName USING :params[i].dbName;
}


/* create THREADS number of threads */
for(i=0;i<THREADS;i++)
{
    printf("Creating thread %d ¥n", i);
    if(i%2)
    {
      /* do a select operation if the thread id is odd */
      if(pthread_create(&thread_id[i],pthread_attr_default,
          (pthread_startroutine_t)selectFunction,
          (pthread_addr_t) &params[i]))
      printf("Cant create thread %d ¥n", i);
    }
    else
    {
      /* otherwise do an update operation */
      if(pthread_create(&thread_id[i],pthread_attr_default,
          (pthread_startroutine_t)updateFunction,
          (pthread_addr_t) &params[i]))
      printf("Cant create thread %d ¥n", i);
    }
}

for(i=0; i<THREADS; i++)
{
  if(pthread_join(thread_id[i],&status))
    printf("Error when waiting for thread % to terminate¥n", i);

}

if(gettimeofday(&tp2, (void*)NULL) == -1){
perror("Second: ");
      exit(0);
```

```
  }

  printf(" ¥n¥nTHE TOTAL TIME TAKEN FOR THE PROGRAM EXECUTION = %f ¥n¥n",
      (float)(tp2.tv_sec - tp1.tv_sec) + ((float)(tp2.tv_usec -
      tp1.tv_usec)/1000000.0));

  /* free the context */
  for(i=0; i<THREADS; i++)
  {
    EXEC SQL CONTEXT USE :params[i].ctx;
    EXEC SQL AT :params[i].connName COMMIT WORK RELEASE;

    EXEC SQL CONTEXT FREE :params[i].ctx;
  }

  return 0;
}

void selectFunction(parameters *params)
{
  struct sqlca sqlca;
  char empName[110][21];
  printf("Thread %d selecting .... ¥n", params->thread_id);

  EXEC SQL CONTEXT USE :params->ctx;
  EXEC SQL AT : params->connName
    SELECT FIRST_NAME into empName from EMPLOYEES;
  printf("Thread %d selected ....¥n", params->thread_id);
  return 0;
}

void updateFunction(parameters *params)
{
  struct sqlca sqlca;
  printf(" Thread %d Updating ... ¥n", params->thread_id);

  EXEC SQL CONTEXT USE :params->ctx;
  EXEC SQL AT :params->connName update EMPLOYEES
    set SALARY = 4000 where DEPARTMENT_ID = 10;

  /* commit the changes */
  EXEC SQL AT :params->connName COMMIT;

  printf(" Thread %d Updated ... ¥n", params->thread_id);
  return 0;
}

/*********** Oracle error ***********/
void err_report(struct sqlca sqlca)
{
  if (sqlca.sqlcode < 0)
   printf("¥n%.*s¥n¥n",sqlca.sqlerrm.sqlerrml,sqlca.sqlerrm.sqlerrmc);

  exit(0);
}
```

第II部 アプリケーション

第II部は、次の章で構成されます。

- C++アプリケーション
- Oracle動的SQL
- ANSI動的SQL
- Oracle動的SQL: 方法4
- LOB
- オブジェクト
- コレクション
- Object Type Translator
- ユーザー・イグジット

12 C++アプリケーション

この章では、Pro*C/C++プリコンパイラを使用してC++の埋込みSQLアプリケーションをプリコンパイルする方法と、Pro*C/C++がC++互換コードを生成する仕組みについて説明します。この章のトピックは、次のとおりです:

- C++サポートの理解
- C++のプリコンパイル
- サンプル・プログラム

12.1 C++サポートの理解

Pro*C/C++でC++がどのようにサポートされているかを理解するには、Pro*C/C++の基本機能を理解する必要があります。特に、Pro*C/C++とPro*Cバージョン1との違いを認識する必要があります。

Pro*C/C++の基本機能は、次のとおりです。

- Cプリプロセッサの完全サポート。Pro*C/C++プログラム内で#define、#include、#ifdefおよびその他のプリプロセッサ・ディレクティブを使用して、プリコンパイラ自体で処理する必要のある構成体を扱うことができます。
- C言語の固有の構造体をホスト変数として使用可能。構造体(または構造体へのポインタ)をホスト変数として関数に渡す機能や、ホスト構造体または構造体ポインタを戻す書込み関数などがあります。

Cプリプロセッサの機能をサポートし、特殊な宣言部の外でホスト変数を宣言できるようにするために、Pro*C/C++には完全なCパーサーが組み込まれています。Pro*C/C++パーサーはCパーサーであり、C++コードは解析できません。

したがって、C++をサポートするには、Cパーサーを完全または部分的に使用禁止にする必要があります。Cパーサーを使用禁止にするために、Pro*C/C++プリコンパイラには、Pro*C/C++がソース・コードに対して行うC解析の範囲を制御できるコマンドライン・オプションが組み込まれています。

関連項目

- Oracleのデータ型
- C++のプリコンパイル

12.1.1 特殊なマクロ処理は不要

C++をPro*C/C++とともに使用する場合、特殊な事前処理や、Pro*C/C++外部の特殊なマクロ・プロセッサは必要ありません。プリコンパイラの出力に対してマクロ・プロセッサを実行しなくても、C++との互換性を実現できます。

Pro*C/C++プリコンパイラの旧リリースのユーザーで、プリコンパイラの出力にマクロ・プロセッサを使用していた場合は、コードを変更しないでPro*C/C++を使用してC++アプリケーションをプリコンパイルできます。

12.2 C++のプリコンパイル

C++に対応できるようにプリコンパイルを制御するには、次の4点を考慮する必要があります。

- プリコンパイラによるコード出力
- 解析機能
- 出力ファイル名の拡張子
- システム・ヘッダー・ファイルの位置

12.2.1 コードの生成

プリコンパイラで生成されるコードの種類(C互換コードまたはC++互換コード)を指定する必要があります。デフォルトでは、Pro*C/C++によってC言語のコードが生成されます。C++は、C言語の完全なスーパーセットではありません。生成されるコードをC++のコンパイラでコンパイルするには、コードを多少変更する必要があります。

たとえば、プリコンパイラでは、アプリケーション・コードが出力されるのみでなく、ランタイム・ライブラリSQLLIBに対するコールが挿入されます。SQLLIB内の関数は、C関数です。特殊なC++版のSQLLIBはありません。このため、C++コンパイラを使用して生成したコードをコンパイルするには、SQLLIB内でコールした関数をPro*C/C++によりC関数として宣言する必要があります。

C言語の出力では、プリコンパイラにより次のようなプロトタイプが生成されます。

```
void sqlora(unsigned long *, void *);
```

ただし、C++互換コードの場合には、プリコンパイラで次のようなコードを生成する必要があります。

```
extern "C" {
void sqlora(unsigned long *, void *);
};
```

Pro*C/C++によって生成されるコードの種類は、CODEというプリコンパイラ・オプションを使用して制御します。このオプションの値は、CPP、KR_CおよびANSI_Cの3つです。これらのオプションの違いは、SQLLIB関数*sqlora*の宣言方法がCODEオプションの3つの値で異なることを考えると説明できます。

```
void sqlora( /*_ unsigned long *, void * _*/);  /* K&R C */

void sqlora(unsigned long *, void *);           /* ANSI C */

extern "C" {                                    /* CPP */
void sqlora(unsigned long *, void *);
};
```

CODE=CPPを指定すると、プリコンパイラは次を行います。

- C++互換コードを生成します。
- 出力ファイルに、標準の「.c」拡張子ではなく、「.C」や「.cc」など、プラットフォーム固有のファイル拡張子(接尾辞)を付けます。(この設定は、CPP_SUFFIXオプションを使用して上書きできます。)
- PARSEオプションの値をデフォルトのPARTIALにします。また、PARSE=NONEも指定できます。PARSE=FULLを指定すると、プリコンパイル時にエラーが発生します。
- C++形式の//コメントをコード内で使用可能にします。CODE=CPPのときは、この形式のコメントをSQL文およびPL/SQLブロックの中でも使用できます。
- Pro*C/C++は//+で始まるSQLオプティマイザ・ヒントを認識できます。
- Object Type Translator(OTT)によって生成されるヘッダー・ファイルは、宣言部の中にインクルードする必要があります。

関連項目:

CODEオプションの値KR_CとANSI_Cの詳細は、CODEを参照してください。

12.2.2 コードの解析について

Pro*C/C++のCパーサーによるコードへの効果を制御する必要があります。この制御はPARSEプリコンパイラ・オプションを使用することで可能になります。このオプションでは、プリコンパイラのCパーサーがコードの取扱方法を制御できます。

PARSEオプションの値と効果を次に示します。

表12-1 PARSEオプションの値と効果

値	効果
PARSE=NONE	値 NONE の効果は次のとおりです。 ● C プリプロセッサ・ディレクティブは、宣言部の中にある場合のみ解釈されます。 ● ホスト変数はすべて、宣言部の中に宣言する必要があります。 ● プリコンパイラ・リリース 1.x の動作。
PARSE=PARTIAL	値 PARTIAL の効果は次のとおりです。 ● すべてのプリプロセッサ・ディレクティブが解釈されます。 ● ホスト変数はすべて、宣言部の中に宣言する必要があります。 このオプション値は、CODE=CPP のときデフォルトです。
PARSE=FULL	値 FULL の効果は次のとおりです。 ● プリコンパイラの C パーサーがコード上で動作します。 ● すべてのプリプロセッサ・ディレクティブが解釈されます。 ● ホスト変数は、C 言語で有効に宣言できる位置であればどこにでも宣言できます。

このオプション値は、CODEオプションの値がCPP以外のときのデフォルト値です。CODE=CPPのときにPARSE=FULLを指定すると、エラーになります。

C++互換コードを生成するには、PARSEオプションにNONEまたはPARTIALのいずれかを指定する必要があります。PARSE=FULLのときはCパーサーが動作し、コードにあるC++クラスなどの構文は認識されません。

12.2.3 出力ファイル名の拡張子

ほとんどのCコンパイラでは、入力ファイルのデフォルト拡張子は.cになります。しかし、C++コンパイラでは、ファイル名の拡張子がコンパイラごとに異なる場合があります。CPP_SUFFIXオプションを指定すると、プリコンパイラで生成されるファイル名拡張子を指定できます。このオプションの値は、引用符もピリオドも付けない文字列です。たとえば、CPP_SUFFIX=ccまたはCPP_SUFFIX=Cのように指定します。

12.2.4 システム・ヘッダー・ファイル

Pro*C/C++では、プラットフォーム固有の標準的な位置でstdio.hなどの標準のシステム・ヘッダー・ファイルが検索されます。Pro*C/C++では、hppまたはh++などの拡張子が付いたヘッダー・ファイルは検索されません。たとえば、ほとんどのUNIXシステムではstdio.hファイルのフルパス名は/usr/include/stdio.hです。

ただし、C++コンパイラは独自のバージョンのstdio.hを持っており、これはシステムの標準位置にはありません。C++でのプリコンパイル時には、Pro*C/C++でシステム・ヘッダー・ファイルを検索できるように、SYS_INCLUDEプリコンパイラ・オプションを使用してディレクトリ・パスを指定する必要があります。次に例を示します。

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

システム・ヘッダー・ファイル以外の位置を指定するには、INCLUDEプリコンパイラ・オプションを使用します。SYS_INCLUDEオプションで指定したディレクトリは、INCLUDEオプションで指定したディレクトリよりも前に検索されます。

PARSE=NONEのときは、Pro*C/C++はシステム・ヘッダー・ファイルをインクルードする必要がないため、システム・ファイルについてSYS_INCLUDEおよびINCLUDEで指定した値は無視されます。(ただし、当然ながら、sqlca.hなどのPro*C/C++固有のヘッダーは、EXEC SQL INCLUDE文を使用してインクルードできます。)

関連項目

- INCLUDE

12.3 サンプル・プログラム

この項には、C++構造体を含んでいるサンプルのPro*C/C++プログラムを3つ記載しています。これらのプログラムはそれぞれdemoディレクトリにオンラインで利用可能な形で入っています。

12.3.1 cppdemo1.pc

```
/*  cppdemo1.pc
 *
 *  Prompts the user for an employee number, then queries the
 *  emp table for the employee's name, salary and commission.
 *  Uses indicator variables (in an indicator struct) to
 *  determine if the commission is NULL.
 */

#include <iostream.h>
#include <stdio.h>
#include <string.h>

// Parse=partial by default when code=cpp,
// so preprocessor directives are recognized and parsed fully.
#define     UNAME_LEN      20
#define     PWD_LEN        40

// Declare section is required when CODE=CPP or
// PARSE={PARTIAL|NONE} or both.
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR username[UNAME_LEN];  // VARCHAR is an ORACLE pseudotype
  varchar password[PWD_LEN];    // can be in lower case also

  // Define a host structure for the output values
  // of a SELECT statement
  struct empdat {
      VARCHAR   emp_name[UNAME_LEN];
```

```
      float     salary;
      float     commission;
  } emprec;

  // Define an indicator struct to correspond to the
  // host output struct
  struct empind {
      short     emp_name_ind;
      short     sal_ind;
      short     comm_ind;
  } emprec_ind;


  // Input host variables
  int   emp_number;
  int   total_queried;
EXEC SQL END DECLARE SECTION;

// Define a C++ class object to match the desired
// struct from the preceding declare section.
class emp {
  char  ename[UNAME_LEN];
  float salary;
  float commission;
public:
  // Define a constructor for this C++ object that
  // takes ordinary C objects.
  emp(empdat&, empind&);
  friend ostream& operator<<(ostream&, emp&);
};

emp::emp(empdat& dat, empind& ind)
{
  strncpy(ename, (char *)dat.emp_name.arr, dat.emp_name.len);
  ename[dat.emp_name.len] = '¥0';
  this->salary = dat.salary;
  this->commission = (ind.comm_ind < 0) ? 0 : dat.commission;
}

ostream& operator<<(ostream& s, emp& e)
{
  return s << e.ename << " earns " << e.salary <<
              " plus " << e.commission << " commission."
           << endl << endl;
}

// Include the SQL Communications Area
// You can use #include or EXEC SQL INCLUDE
#include <sqlca.h>

// Declare error handling function
void sql_error(char *msg);

main()
{
  char temp_char[32];

  // Register sql_error() as the error handler
  EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error:");
```

```
  // Connect to ORACLE.  Program calls sql_error()
  // if an error occurs
  // when connecting to the default database.
  // Note the (char *) cast when
  // copying into the VARCHAR array buffer.
  username.len = strlen(strcpy((char *)username.arr, "SCOTT"));
  password.len = strlen(strcpy((char *)password.arr, "TIGER"));

  EXEC SQL CONNECT :username IDENTIFIED BY :password;

  // Here again, note the (char *) cast when using VARCHARs
  cout << "¥nConnected to ORACLE as user: "
       << (char *)username.arr << endl << endl;

  // Loop, selecting individual employee's results
  total_queried = 0;
  while (1)
  {
      emp_number = 0;
      printf("Enter employee number (0 to quit): ");
      gets(temp_char);
      emp_number = atoi(temp_char);
      if (emp_number == 0)
        break;

      // Branch to the notfound label when the
      // 1403 ("No data found") condition occurs
      EXEC SQL WHENEVER NOT FOUND GOTO notfound;

      EXEC SQL SELECT ename, sal, comm
         INTO :emprec INDICATOR :emprec_ind // You can also use
                                            // C++ style
         FROM EMP                   // Comments in SQL statemtents.
         WHERE EMPNO = :emp_number;

      {
        // Basic idea is to pass C objects to
        // C++ constructors thus
        // creating equivalent C++ objects used in the
        // usual C++ way
        emp e(emprec, emprec_ind);
        cout << e;
      }

      total_queried++;
      continue;
notfound:
      cout << "Not a valid employee number - try again."
           << endl << endl;
  } // end while(1)

  cout << endl << "Total rows returned was "
       << total_queried << endl;
  cout << "Have a nice day!" << endl << endl;

  // Disconnect from ORACLE
  EXEC SQL COMMIT WORK RELEASE;
  exit(0);
}
```

```
void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    cout << endl << msg << endl;
    cout << sqlca.sqlerrm.sqlerrmc << endl;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

12.3.2 cppdemo2.pc

次のアプリケーションは、簡単なモジュール化の例です。最初にSQL*Plusの次のSQLスクリプトcppdemo2.sqlを実行します。

```
Rem  This is the SQL script that accompanies the cppdemo2 C++ Demo
Rem  Program.  Run this prior to Precompiling the empclass.pc file.
/
CONNECT SCOTT/TIGER
/
CREATE OR REPLACE VIEW emp_view AS SELECT ename, empno FROM EMP
/
CREATE OR REPLACE PACKAGE emp_package AS
  TYPE emp_cursor_type IS REF CURSOR RETURN emp_view%ROWTYPE;
  PROCEDURE open_cursor(curs IN OUT emp_cursor_type);
END emp_package;
/
CREATE OR REPLACE PACKAGE BODY emp_package AS
  PROCEDURE open_cursor(curs IN OUT emp_cursor_type) IS
  BEGIN
    OPEN curs FOR SELECT ename, empno FROM emp_view ORDER BY ename ASC;
  END;
END emp_package;
/
EXIT
/
```

ヘッダー・ファイルempclass.hでは、empクラスが定義されます。

```
// This class definition may be included in a Pro*C/C++ application
// program using the EXEC SQL INCLUDE directive only.  Because it
// contains EXEC SQL syntax, it may not be included using a #include
// directive.  Any program that includes this header must be
// precompiled with the CODE=CPP option.  This emp class definition
// is used when building the cppdemo2 C++ Demo Program.

class emp
{
  public:
    emp();   // Constructor: ALLOCATE Cursor Variable
    ~emp();  // Desctructor: FREE Cursor Variable

    void open();              // Open Cursor
    void fetch() throw (int); // Fetch (throw NOT FOUND condition)
    void close();             // Close Cursor

    void emp_error();         // Error Handler

    EXEC SQL BEGIN DECLARE SECTION;
      // When included using EXEC SQL INCLUDE, class variables have
      // global scope and are thus basically treated as ordinary
```

```
      // global variables by Pro*C/C++ during precompilation.
      char ename[10];
      int empno;
    EXEC SQL END DECLARE SECTION;

  private:
    EXEC SQL BEGIN DECLARE SECTION;
      // Pro*C/C++ treats this as a simple global variable also.
      SQL_CURSOR emp_cursor;
    EXEC SQL END DECLARE SECTION;
};
```

empclass.pcのコードには、empメソッドが含まれています。

```
#include <stdio.h>
#include <stdlib.h>

// This example uses a single (global) SQLCA that is shared by the
// emp class implementation as well as the main program for this
// application.
#define SQLCA_STORAGE_CLASS extern
#include <sqlca.h>

// Include the emp class specification in the implementation of the
// class body as well as the application program that makes use of it.
EXEC SQL INCLUDE empclass.h;

emp::emp()
{
  // The scope of this WHENEVER statement spans the entire module.
  // Note that the error handler function is really a member function
  // of the emp class.
  EXEC SQL WHENEVER SQLERROR DO emp_error();
  EXEC SQL ALLOCATE :emp_cursor;  // Constructor - ALLOCATE Cursor.
}

emp::~emp()
{
  EXEC SQL FREE :emp_cursor;      // Destructor - FREE Cursor.
}

void emp::open()
{
  EXEC SQL EXECUTE
    BEGIN
      emp_package.open_cursor(:emp_cursor);
    END;
  END-EXEC;
}

void emp::close()
{
  EXEC SQL CLOSE :emp_cursor;
}

void emp::fetch() throw (int)
{
  EXEC SQL FETCH :emp_cursor INTO :ename, :empno;
  if (sqlca.sqlcode == 1403)
    throw sqlca.sqlcode;     // Like a WHENEVER NOT FOUND statement.
```

```
}

void emp::emp_error()
{
  printf("%.*s¥n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}
```

メイン・プログラムcppdemo2.pcでは、カーソル変数が使用されます。

```
// Pro*C/C++ sample program demonstrating a simple use of Cursor Variables
// implemented within a C++ class framework.  Build this program as follows
//
//   1. Execute the cppdemo2.sql script within SQL*Plus
//   2. Precompile the empclass.pc program as follows
//      > proc code=cpp sqlcheck=full user=scott/tiger lines=yes empclass
//   3. Precompile the cppdemo2.pc program as follows
//      > proc code=cpp lines=yes cppdemo2
//   4. Compile and Link
//
// Note that you may have to specify various include directories using the
// include option when precompiling.

#include <stdio.h>
#include <stdlib.h>
#include <sqlca.h>

static void sql_error()
{
  printf("%.*s¥n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(1);
}

// Physically include the emp class definition in this module.
EXEC SQL INCLUDE empclass.h;

int main()
{
  EXEC SQL BEGIN DECLARE SECTION;
    char *uid = "scott/tiger";
  EXEC SQL END DECLARE SECTION;

  EXEC SQL WHENEVER SQLERROR DO sql_error();
  EXEC SQL CONNECT :uid;

  emp *e = new emp(); // Invoke Constructor - ALLOCATE Cursor Variable.

  e->open();          // Open the Cursor.

  while (1)
    {
      // Fetch from the Cursor, catching the NOT FOUND condition
      // thrown by the fetch() member function.
      try { e->fetch(); } catch (int code)
        { if (code == 1403) break; }
      printf("Employee:  %s[%d]¥n", e->ename, e->empno);
```

```
    }

  e->close();         // Close the Cursor.

  delete e;           // Invoke Destructor - FREE Cursor Variable.

  EXEC SQL ROLLBACK WORK RELEASE;
  return (0);
}
```

12.3.3 cppdemo3.pc

```
/*
 * cppdemo3.pc : An example of C++ Inheritance
 *
 * This program finds all salesman and prints their names
 * followed by how much they earn in total (ie; including
 * any commissions).
 */

#include <iostream.h>
#include <stdio.h>
#include <sqlca.h>
#include <string.h>

#define NAMELEN 10

class employee {    // Base class is a simple employee
public:
  char ename[NAMELEN];
  int sal;
  employee(char *, int);
};

employee::employee(char *ename, int sal)
{
  strcpy(this->ename, ename);
  this->sal = sal;
}

// A salesman is a kind of employee
class salesman : public employee
{
  int comm;
public:
  salesman(char *, int, int);
  friend ostream& operator<<(ostream&, salesman&);
};

// Inherits employee attributes
salesman::salesman(char *ename, int sal, int comm)
  : employee(ename, sal), comm(comm) {}

ostream& operator<<(ostream& s, salesman& m)
{
  return s << m.ename << m.sal + m.comm << endl;
}

void print(char *ename, int sal, int comm)
```

```
{
  salesman man(ename, sal, comm);
  cout << man;
}

main()
{
  EXEC SQL BEGIN DECLARE SECTION;
    char *uid = "scott/tiger";
    char  ename[NAMELEN];
    int   sal, comm;
    short comm_ind;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL WHENEVER SQLERROR GOTO error;

  EXEC SQL CONNECT :uid;
  EXEC SQL DECLARE c CURSOR FOR
    SELECT ename, sal, comm FROM emp WHERE job = 'SALESMAN'
      ORDER BY ename;
  EXEC SQL OPEN c;

  cout << "Name    Salary" << endl << "------  ------" << endl;

  EXEC SQL WHENEVER NOT FOUND DO break;
  while(1)
   {
     EXEC SQL FETCH c INTO :ename, :sal, :comm:comm_ind;
     print(ename, sal, (comm_ind < 0) ? 0 : comm);
   }
  EXEC SQL CLOSE c;
  exit(0);

error:
  cout << endl << sqlca.sqlerrm.sqlerrmc << endl;
  exit(1);
}
```

13 Oracle動的SQL

この章では、アプリケーションに柔軟性と機能性を持たせる高度なプログラミング技法であるOracle動的SQLの使用方法について説明します。SQL文を実行時に受け入れて処理するプログラムについて、4つの作成方法を紹介します。この章のトピックは、次のとおりです:

> 注意:
>
> Oracle 動的 SQL は、オブジェクト型、カーソル変数、構造体の配列、DML RETURNING 句、Unicode 変数、および LOB をサポートしていません。かわりに ANSI 動的 SQL 方法 4 を使用してください。

- 動的SQL
- 動的SQLの長所および短所
- 動的SQLの使用
- 動的SQL文の要件
- 動的SQL文の処理方法
- 動的SQLの使用方法
- 方法1の使用方法
- 方法2の使用方法
- 方法3の使用方法
- 方法4の使用方法
- DECLARE STATEMENT文の使用方法について
- PL/SQLの使用について
- 動的SQL文のキャッシング

13.1 動的SQL

ほとんどのデータベース・アプリケーションでは、ある特定のジョブが実行されます。たとえば、ユーザーに従業員番号の入力を要求して、その後EMPおよびDEPTの表の行を更新するという単純なプログラムがあります。この場合は、プリコンパイル時にUPDATE文の構成がわかっています。つまり、変更する表、それぞれの表および列に定義されている制約、更新する列、それぞれの列のデータ型がわかっています。

しかし、アプリケーションによっては、様々なSQL文を実行時に受け入れ(または作成し)、処理する必要があります。たとえば汎用レポート・ライターでは、生成するレポートについてそれぞれ別のSELECT文を作成する必要があります。この場合、文の構成は実行時までわかりません。このような文は実行のたびに異なる可能性があります。このような文を動的SQL文と呼びます。

静的SQL文とは異なり、動的SQL文はソース・プログラム内には埋め込まれません。そのかわり、これらの文は実行時にプログラムに入力される(またはプログラムによって作成される)文字列に格納されます。動的SQL文は対話形式で入力できるのみでなく、ファイルから読み込むこともできます。

13.2 動的SQLの長所および短所

通常の埋込みSQLプログラムと比べると、動的に定義されたSQL文を受け入れて処理するホスト・プログラムの方が柔軟性は高くなります。動的SQL文は、SQLの知識がほとんどないユーザーでも対話形式で作成できます。

たとえば、SELECT文、UPDATE文またはDELETE文のWHERE句内で使用する検索条件の入力をユーザーに要求する単純なプログラムがあります。さらに複雑なプログラムでは、SQL処理、表およびビューの名前、列の名前などを表示されているメニューからユーザーが選択できるようになります。このように、動的SQLを使用すると柔軟性に富んだアプリケーションを記述できます。

ただし、動的問合せの中には複雑なコーディング、特殊なデータ構造体の使用および実行時の処理の増加が必要なものもあります。処理時間が増えることは支障がない場合もありますが、動的SQLの概念および方法を完全に理解するまではコーディングが難しく感じられることもあります。

13.3 動的SQLの使用

実際は静的SQLによって、プログラミング要件のほとんどを満たすことができます。動的SQLは、その高度な柔軟性が必要とされる場合にのみ使用してください。プリコンパイル時に次の項目のいずれかが不明の場合は、動的SQLの使用が適しています。

- SQL文のテキスト(コマンドまたは句など)
- ホスト変数の数
- ホスト変数のデータ型
- データベース・オブジェクトの参照(列、索引、順序、表、ユーザー名またはビューなど)

13.4 動的SQL文の要件

動的SQL文を記述するには、文字列に有効なSQL文を示すテキストが格納される必要がありますが、このときEXEC SQL句、文終了記号または次に示す埋込みSQLコマンドを含まないでください。

- ALLOCATE
- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- FREE
- GET
- INCLUDE
- OPEN
- PREPARE
- SET
- WHENEVER

ほとんどの場合、この文字列にはダミーのホスト変数が含まれます。これらはSQL文内に実際のホスト変数のための場所を確保

します。ダミーのホスト変数はプレースホルダにすぎないため宣言する必要はなく、しかも任意の名前を指定できます。たとえば、Oracleでは次の2つの文字列は区別されません。

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

13.5 動的SQL文の処理方法

一般にアプリケーション・プログラムでは、SQL文のテキストおよびその文で使用するホスト変数の値をユーザーが入力する必要があります。入力されたSQL文は、Oracleにより構文規則に従っているかどうかが解析されます。

次に、Oracleはホスト変数をSQL文にバインドします。つまり、Oracleがホスト変数のアドレスを取得するため、値の読込みと書込みができます。

この後、OracleでSQL文が実行されます。つまり、そのSQL文の要求(表からの行の削除など)をOracleが実行します。

これらのホスト変数に別の値を指定することで、このSQL文を繰り返し実行できます。

13.6 動的SQLの使用方法

この項では、動的SQL文の定義に使用できる4つの方法を紹介します。まずそれぞれの方法の機能および制限事項を簡単に説明した後、適切な方法を選択するためのガイドラインを示します。この後の項でこれらの方法の使用方法を説明します。また、学習用にサンプル・プログラムを示します。

この4つの方法は番号が大きくなるに従って対象が広がるようになっています。つまり方法2は方法1を包含し、方法3は方法1と方法2を包含するというようになります。ただし、表13-1のように、それぞれの方法は特定の種類のSQL文を処理するのに適しています。

表13-1 動的SQLの使用方法

方法	SQL文の種類
1	ホスト変数のない非問合せ
2	入力ホスト変数の数がわかっている非問合せ
3	選択リスト項目の数および入力ホスト変数の数がわかっている問合せ
4	選択リストの項目の数または入力ホスト変数の数が不明な問合せ

注意:

選択リスト項目には、SAL * 1.10 および MAX(SAL)などの列名や式が含まれます。

13.6.1 方法1

この方法では、動的SQL文を受け入れるかまたは作成し、EXECUTE IMMEDIATEコマンドを使用してその文をすぐに実行

できます。このSQL文では、問合せ(SELECT文)の使用や、入力ホスト変数に対するプレースホルダの組込みはできません。たとえば、次のホスト文字列は有効です。

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

方法1では、SQL文は実行のたびに解析されます。

13.6.2 方法2

この方法では、動的SQL文を受け入れるかまたは作成し、PREPAREおよびEXECUTEコマンドを使用してその文を処理できます。SQL文は問合せであってはなりません。プリコンパイル時に、入力ホスト変数のプレースホルダの数および入力ホスト変数のデータ型を明確にする必要があります。たとえば、次のホスト文字列はこのカテゴリに該当します。

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

方法2では、SQL文の解析は一度しか行われませんが、ホスト変数に異なる値を指定して、このSQL文を複数回実行できます。SQLデータ定義文(CREATEやGRANTなど)は、PREPAREの際に実行されます。

13.6.3 方法3

この方法を使用すると、プログラムは動的問合せを受け入れ(または作成し)、DECLARE、OPEN、FETCHおよびCLOSEカーソル・コマンドとともにPREPAREコマンドを使用してその問合せを処理します。プリコンパイル時に、選択リストの項目数、入力ホスト変数のプレース・ホルダ数および入力ホスト変数のデータ型がわかっている必要があります。たとえば、次のホスト文字列は有効です。

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number'
```

13.6.4 方法4

この方法を使用すると、プログラムは動的SQL文を受け入れ(または作成し)、記述子を使用してその文を処理します。選択リストの項目数、入力ホスト変数のプレース・ホルダー数および入力ホスト変数のデータ型は、実行時まで不明でもかまいません。たとえば、次のホスト文字列はこのカテゴリに該当します。

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

方法4は、選択リスト項目の数または入力ホスト変数の数が不明な動的SQL文を実行するときに必要です。

関連項目

- 方法4の使用方法

13.6.5 ガイドライン

4つの方法はどれも、動的SQL文を文字列に格納する必要があり、この文字列は、ホスト変数または引用符で囲んだリテラルであることが必要です。SQL文を文字列に格納する場合は、キーワードEXEC SQLおよび文の終了記号「;」は省略してください。

方法2および方法3のときは、入力ホスト変数のプレースホルダの数と入力ホスト変数のデータ型をプリコンパイル時には明確にしてください。

方法の番号が大きくなるほどアプリケーションへの制約は少なくなりますが、コードの記述が難しくなります。通常は、最も簡単な方法を使用してください。ただし動的SQL文を方法1で繰り返し実行する場合は、実行のたびにその文が再解析されるのを避けるために方法2を使用します。

方法4は最も柔軟性に富んでいますが、複雑なコード記述方法および動的SQLの概念の完全な理解が求められます。通常、方法4を使用するのは、方法1、2または3を使用できない場合のみです。

図13-1の決定論理を基に、適切な方法を選択できます。

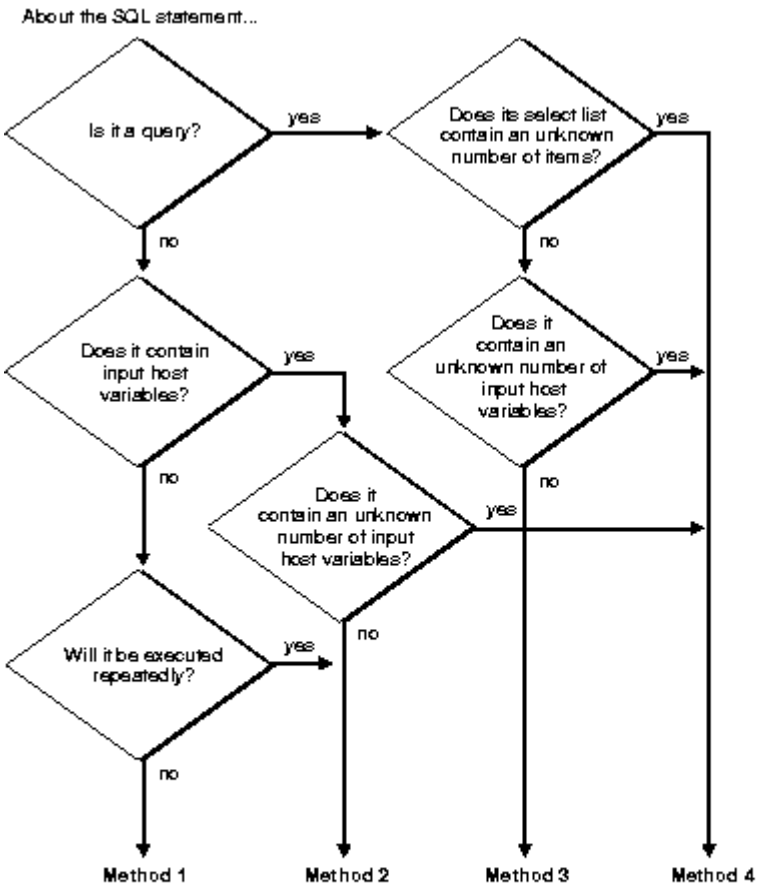
13.6.5.1 一般的なエラーの回避について

コマンドライン・オプションDBMS=V6_CHARと指定してプリコンパイルするときは、SQL文を格納する前に配列を空白文字で埋めてください。こうして余分な文字を消去します。別のSQL文を格納するために配列を再利用するときに、この処理が重要となります。原則として、SQL文を格納する前に必ずホスト文字列を初期化(または再初期化)してください。ホスト文字列にはヌル終端文字を使用しないでください。OracleではNULL終了文字は文字列の終了マーカーとはみなされません。SQL文の一部として扱われます。

コマンドライン・オプションDBMS=V8と指定してプリコンパイルするときは、PREPARE文またはEXECUTE IMMEDIATE文を実行する前に、文字列がヌル文字で終了していることを確認してください。

DBMSの値が何であっても、VARCHAR変数を使用して動的SQL文を格納するときは、PREPARE文またはEXECUTE IMMEDIATE文を実行する前に、VARCHARの長さが正しく設定(または再設定)されていることを確認してください。

図13-1 適切な方法の選択



13.7 方法1の使用方法

最も単純な動的SQL文では、結果が成功か失敗のどちらかで、ホスト変数は使用されません。次に例を示します。

```
'DELETE FROM table_name WHERE column_name = constant'
'CREATE TABLE table_name ...'
'DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
'REVOKE RESOURCE FROM username'
```

方法1では、SQL文を解析すると、EXECUTE IMMEDIATEコマンドを使用してすぐに実行します。コマンドの後には、実行するSQL文を含む文字列(ホスト変数またはリテラル)が続きますが、この文は問合せにしないでください。

EXECUTE IMMEDIATE文の構文は次のとおりです。

```
EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };
```

次の例では、ホスト変数*dyn_stmt*を使用して、ユーザーが入力するSQL文を格納します。

```
char dyn_stmt[132];
...
for (;;)
{
    printf("Enter SQL statement: ");
    gets(dyn_stmt);
    if (*dyn_stmt == '¥0')
        break;
    /* dyn_stmt now contains the text of a SQL statement */
    EXEC SQL EXECUTE IMMEDIATE :dyn_stmt;
}
...
```

次の例のように、文字列リテラルを使用することもできます。

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

EXECUTE IMMEDIATEは入力されているSQL文を実行するたびに解析するため、方法1は1回しか実行しない文に最も適しています。一般に、データ定義言語がこのカテゴリに該当します。

13.7.1 サンプル・プログラム: 動的SQL方法1

次のプログラムでは動的SQL方法1を使用して、表の作成、行の挿入、挿入のコミット、表の削除を実行します。このプログラムはdemoディレクトリのsample6.pcファイルにあり、オンラインで利用できます。

```
/*
 *  sample6.pc: Dynamic SQL Method 1
 *
 *  This program uses dynamic SQL Method 1 to create a table,
 *  insert a row, commit the insert, then drop the table.
 */

#include <stdio.h>
#include <string.h>

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
```

```
 * program.
 */
#include <sqlca.h>

/* Include the ORACLE Communications Area, a structure through
 * which ORACLE makes additional runtime status information
 * available to the program.
 */
#include <oraca.h>

/* The ORACA=YES option must be specified to enable you
 * to use the ORACA.
 */

EXEC ORACLE OPTION (ORACA=YES);

/* Specifying the RELEASE_CURSOR=YES option instructs Pro*C
 * to release resources associated with embedded SQL
 * statements after they are executed.  This ensures that
 * ORACLE does not keep parse locks on tables after data
 * manipulation operations, so that subsequent data definition
 * operations on those tables do not result in a parse-lock
 * error.
 */

EXEC ORACLE OPTION (RELEASE_CURSOR=YES);

void dyn_error();


main()
{
/* Declare the program host variables. */
    char    *username = "SCOTT";
    char    *password = "TIGER";
    char    *dynstmt1;
    char     dynstmt2[10];
    VARCHAR  dynstmt3[80];

/* Call routine dyn_error() if an ORACLE error occurs. */

    EXEC SQL WHENEVER SQLERROR DO dyn_error("Oracle error:");

/* Save text of current SQL statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("¥nConnected to ORACLE.¥n");

/* Execute a string literal to create the table.  This
 * usage is actually not dynamic because the program does
 * not determine the SQL statement at run time.
 */
    puts("CREATE TABLE dyn1 (col1 VARCHAR2(4))");

    EXEC SQL EXECUTE IMMEDIATE
```

```
        "CREATE TABLE dyn1 (col1 VARCHAR2(4))";

/* Execute a string to insert a row.  The string must
 * be null-terminated.  This usage is dynamic because the
 * SQL statement is a string variable whose contents the
 * program can determine at run time.
 */
    dynstmt1 = "INSERT INTO DYN1 values ('TEST')";
    puts(dynstmt1);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt1;

/* Execute a SQL statement in a string to commit the insert.
 * Pad the unused trailing portion of the array with spaces.
 * Do NOT null-terminate it.
 */
    strncpy(dynstmt2, "COMMIT    ", 10);
    printf("%.10s¥n", dynstmt2);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt2;

/* Execute a VARCHAR to drop the table.  Set the .len field
 * to the length of the .arr field.
 */
    strcpy(dynstmt3.arr, "DROP TABLE DYN1");
    dynstmt3.len = strlen(dynstmt3.arr);
    puts((char *) dynstmt3.arr);

    EXEC SQL EXECUTE IMMEDIATE :dynstmt3;

/* Commit any outstanding changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;

    puts("¥nHave a good day!¥n");

    return 0;
}


void
dyn_error(msg)
char *msg;
{
/* This is the Oracle error handler.
 * Print diagnostic text containing the error message,
 * current SQL statement, and location of error.
 */
    printf("¥n%.*s¥n",
       sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in ¥"%.*s...¥'¥n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.¥n¥n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

/* Disable Oracle error checking to avoid an infinite loop
 * should another error occur within this routine as a
 * result of the rollback.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
```

```
/* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK RELEASE;

    exit(1);
}
```

13.8 方法2の使用方法

方法1では1ステップで実行し、方法2では2ステップに分けて実行します。動的SQL文(問合せは不可)は、まずPREPARE(名前の指定と解析)され、次にEXECUTEされます。

方法2では、SQL文には入力ホスト変数と標識変数のプレースホルダを指定できます。このSQL文は一度PREPAREすると、ホスト変数に別の値を指定して繰り返しEXECUTEできます。したがって、(ログアウトして再接続しないかぎり)COMMITまたはROLLBACKの後に再度SQL文をPREPAREする必要はありません。

方法4では、非問合せにEXECUTEを使用できます。

PREPARE文の構文は次のとおりです。

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
```

PREPAREは、このSQL文を解析して名前を指定します。

*statement_name*は、ホスト変数やプログラム変数ではなく、プリコンパイラで使用される識別子であり、宣言部で宣言されません。これはEXECUTEの対象としてPREPARE済の文を示しているにすぎません。

EXECUTE文の構文は次のとおりです。

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

*host_variable_list*は、次の構文を表しています。

```
:host_variable1[:indicator1] [, host_variable2[:indicator2], ...]
```

解析したSQL文は、それぞれの入力ホスト変数に指定済の値を使用してEXECUTEによって実行されます。

次の例では、入力されたSQL文に、プレースホルダ*n*が含まれています。

```
...
int emp_number    INTEGER;
char delete_stmt[120], search_cond[40];;
...
strcpy(delete_stmt, "DELETE FROM EMP WHERE EMPNO = :n AND ");
printf("Complete the following statement's search condition--¥n");
printf("%s¥n", delete_stmt);
gets(search_cond);
strcat(delete_stmt, search_cond);

EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
for (;;)
{

    printf("Enter employee number: ");
    gets(temp);
    emp_number = atoi(temp);
    if (emp_number == 0)
```

```
        break;
    EXEC SQL EXECUTE sql_stmt USING :emp_number;
}
...
```

方法2では、プリコンパイル時に入力ホスト変数のデータ型がわかっている必要があります。最後の例では、*emp_number*が**int**として宣言されています。Oracleでは、**float**や**char**など、全データ型のOracle内部NUMBERデータ型への変換がサポートされるため、これをfloatまたはcharとして宣言することも可能です。

13.8.1 USING句

SQL文がEXECUTEされると、USING句の入力ホスト変数は、PREPAREされた動的SQL文内の該当するプレースホルダに置換されます。

PREPAREされた動的SQL文のプレースホルダは、それぞれが必ずUSING句の個別のホスト変数に対応している必要があります。つまり、PREPAREされた文に同じプレースホルダが2回以上現れるときは、それぞれがUSING句のホスト変数に対応している必要があります。

プレースホルダの名前は、ホスト変数の名前と一致する必要はありません。ただし、PREPARE済動的SQL文のプレースホルダの順序は、USING句の対応するホスト変数の順序と一致する必要があります。

USING句のホスト変数のうち1つでも配列がある場合は、すべてのホスト変数が配列であることが必要です。

NULLを指定するために、標識変数をUSING句のホスト変数と対応付けることができます。

関連項目

- 標識変数

13.8.2 サンプル・プログラム: 動的SQL方法2

次のプログラムでは、動的SQL方法2を使用して2つの行をEMP表に挿入し、その後でそれらの行を削除しています。このプログラムはdemoディレクトリのsample7.pcファイルにあり、オンラインで利用できます。

```
/*
 *  sample7.pc: Dynamic SQL Method 2
 *
 *  This program uses dynamic SQL Method 2 to insert two rows into
 *  the EMP table, then delete them.
 */

#include <stdio.h>
#include <string.h>

#define USERNAME "SCOTT"
#define PASSWORD "TIGER"

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program.
 */
#include <sqlca.h>

/* Include the ORACLE Communications Area, a structure through
 * which ORACLE makes additional runtime status information
 * available to the program.
 */
```

```
#include <oraca.h>

/* The ORACA=YES option must be specified to enable use of
 * the ORACA.
 */
EXEC ORACLE OPTION (ORACA=YES);

char    *username = USERNAME;
char    *password = PASSWORD;
VARCHAR  dynstmt[80];
int      empno   = 1234;
int      deptno1 = 97;
int      deptno2 = 99;


/* Handle SQL runtime errors. */
void dyn_error();


main()
{
/* Call dyn_error() whenever an error occurs
 * processing an embedded SQL statement.
 */
    EXEC SQL WHENEVER SQLERROR DO dyn_error("Oracle error");

/* Save text of current SQL statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("¥nConnected to Oracle.¥n");

/* Assign a SQL statement to the VARCHAR dynstmt.  Both
 * the array and the length parts must be set properly.
 * Note that the statement contains two host-variable
 * placeholders, v1 and v2, for which actual input
 * host variables must be supplied at EXECUTE time.
 */
    strcpy(dynstmt.arr,
        "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:v1, :v2)");
    dynstmt.len = strlen(dynstmt.arr);

/* Display the SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
    printf("   v1 = %d,  v2 = %d¥n", empno, deptno1);

/* The PREPARE statement associates a statement name with
 * a string containing a SQL statement.  The statement name
 * is a SQL identifier, not a host variable, and therefore
 * does not appear in the Declare Section.

 * A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
 */
```

```
    EXEC SQL PREPARE S FROM :dynstmt;

/* The EXECUTE statement executes a PREPAREd SQL statement
 * USING the specified input host variables, which are
 * substituted positionally for placeholders in the
 * PREPAREd statement.  For each occurrence of a
 * placeholder in the statement there must be a variable
 * in the USING clause.  That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.
 * The USING clause can be omitted only if the statement
 * contains no placeholders.
 *
 * A single PREPAREd statement can be EXECUTEd more
 * than once, optionally USING different input host
 * variables.
 */
    EXEC SQL EXECUTE S USING :empno, :deptno1;

/* Increment empno and display new input host variables. */

    empno++;
    printf("   v1 = %d,  v2 = %d¥n", empno, deptno2);

/* ReEXECUTE S to insert the new value of empno and a
 * different input host variable, deptno2.
 * A rePREPARE is unnecessary.
 */
    EXEC SQL EXECUTE S USING :empno, :deptno2;

/* Assign a new value to dynstmt. */

    strcpy(dynstmt.arr,
        "DELETE FROM EMP WHERE DEPTNO = :v1 OR DEPTNO = :v2");
    dynstmt.len = strlen(dynstmt.arr);

/* Display the new SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
    printf("   v1 = %d,    v2 = %d¥n", deptno1, deptno2);

/* RePREPARE S FROM the new dynstmt. */

    EXEC SQL PREPARE S FROM :dynstmt;

/* EXECUTE the new S to delete the two rows previously
 * inserted.
 */
    EXEC SQL EXECUTE S USING :deptno1, :deptno2;

/* Commit any pending changes and disconnect from Oracle. */

    EXEC SQL COMMIT RELEASE;
    puts("¥nHave a good day!¥n");
    exit(0);
}


void
```

```
dyn_error(msg)
char *msg;
{
/* This is the ORACLE error handler.
 * Print diagnostic text containing error message,
 * current SQL statement, and location of error.
 */
    printf("¥n%s", msg);
    printf("¥n%.*s¥n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in ¥"%.*s...¥"¥n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.¥n¥n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Roll back any pending changes and
 * disconnect from Oracle.
 */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

13.9 方法3の使用方法

方法3は方法2に似ていますが、PREPARE文をカーソルの定義および処理に必要な文と結合する点で異なります。これによって、プログラムで問合せを受け入れて処理できます。動的SQL文が問合せである場合は、方法3または方法4を必ず使用してください。

方法3では、プリコンパイル時に問合せ選択リストの列数と入力ホスト変数のプレースホルダの数を明確にする必要があります。ただし、表および列などのデータベース・オブジェクトの名前は実行時まで指定する必要はありません。データベース・オブジェクトの名前はホスト変数に指定できません。問合せ結果を限定、分類およびソートする句(WHERE、GROUP BY、ORDER BYなど)も実行時に指定できます。

方法3では、埋込みSQL文を次のような順序で使用します。

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

方法3では、スクロール可能カーソルも使用できます。スクロール可能カーソルには、埋込みSQL文を次の順序で使用する必要があります。

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE  cursor_name  SCROLL CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH  [ FIRST| PRIOR|NEXT|LAST|CURRENT | RELATIVE fetch_offset
               |ABSOLUTE fetch_offset ]  cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

各文の機能を次に説明します。

13.9.1 PREPARE (動的SQL)

PREPAREはこの動的SQL文を解析し、名前を指定します。次の例では、PREPAREは文字列*select_stmt*内の問合せを解析し、これに*sql_stmt*という名前を指定します。

```
char select_stmt[132] =
     "SELECT MGR, JOB FROM EMP WHERE SAL < :salary";
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

一般的には、この問合せのWHERE句は実行時に端末から入力するか、またはアプリケーションによって生成されます。

識別子*sql_stmt*は、ホスト変数でもプログラム変数でもありませんが、一意にする必要があります。sql_stmtは特定の動的SQL文を指定します。

次の文も有効です。

```
EXEC SQL PREPARE sql_stmt FROM SELECT MGR, JOB FROM EMP WHERE SAL < :salary;
```

'%'ワイルドカードを使用する次のPREPARE文も正しい文です。

```
EXEC SQL PREPARE S FROM select ename FROM test WHERE ename LIKE 'SMIT%';
```

13.9.2 DECLARE (動的SQL)

DECLAREは、カーソルに名前を指定し、これを特定の問合せに関連付けてカーソルを定義します。次の例では、DECLAREにより*emp_cursor*という名前のカーソルを定義し、それを*sql_stmt*に関連付けています。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

識別子*sql_stmt*および*emp_cursor*は、ホスト変数でもプログラム変数でもありませんが、一意であることが必要です。同じ文名を使用して2つのカーソルを宣言すると、プリコンパイラではこの2つのカーソル名を同義とみなします。

emp_cursorという名前のスクロール可能カーソルを定義して、これをsql_stmtに対応付けすることが可能です。

```
EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR sql_stmt;
```

たとえば次の文を実行したとします。

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept_cursor FOR sql_stmt;
```

この場合、*emp_cursor*をOPENすると、処理の対象となるのは*delete_stmt*に格納されている動的SQL文で、*select_stmt.*に格納されている動的SQL文ではありません。

13.9.3 OPEN (動的SQL)

OPENは、アクティブ・セットを識別して、Oracleカーソルを割り当て、入力ホスト変数にバインドし、問合せを実行します。OPENはさらに、アクティブ・セットの最初の行にカーソルを位置付け、SQLCA内の*sqlerrd*の3番目の要素に保存される処理済行数を0(ゼロ)に設定します。USING句の入力ホスト変数は、PREPARE済動的SQL文内の対応するプレースホルダに置き換わります。

前述の例に続いて、OPENは次に示すように*emp_cursor*を割り当て、ホスト変数*salary*をWHERE句に割り当てます。

```
EXEC SQL OPEN emp_cursor USING :salary;
```

13.9.4 FETCH (動的SQL)

FETCHは、アクティブ・セットから行を戻し、選択リスト内の列値をINTO句の対応するホスト変数に割り当てた後、カーソルを次の行に進めます。他に行がない場合は、FETCHにより「データが見つかりません。」というOracleエラー・コードが*sqlca.sqlcode*に戻されます。

次の例では、FETCHはアクティブ・セットから1行を戻して、MGRおよびJOBの列の値をホスト変数の*mgr_number*および*job_title*に割り当てます。

```
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
```

カーソルがSCROLLモードで宣言されている場合は、様々なFETCH方向モードを使用して結果セットにランダムにアクセスできます。

13.9.5 CLOSE (動的SQL)

CLOSEはカーソルを使用禁止にします。一度カーソルをCLOSEすると、それ以降はFETCHできなくなります。

次の例では、CLOSEにより*emp_cursor*が使用禁止になります。

```
EXEC SQL CLOSE emp_cursor;
```

13.9.6 サンプル・プログラム: 動的SQL方法3

次のプログラムは、動的SQL方法3を使用してEMP表から指定された部門のすべての従業員の名前を検索します。このプログラムはdemoディレクトリのsample8.pc ファイルにあり、オンラインで利用できます。

```
/*
 *  sample8.pc:  Dynamic SQL Method 3
 *
 *  This program uses dynamic SQL Method 3 to retrieve the names
 *  of all employees in a given department from the EMP table.
 */

#include <stdio.h>
#include <string.h>

#define USERNAME "SCOTT"
#define PASSWORD "TIGER"

/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program. Also include the ORACA.
 */
#include <sqlca.h>
#include <oraca.h>

/* The ORACA=YES option must be specified to enable use of
 * the ORACA.
 */
EXEC ORACLE OPTION (ORACA=YES);

char    *username = USERNAME;
char    *password = PASSWORD;
```

```
VARCHAR  dynstmt[80];
VARCHAR  ename[10];
int      deptno = 10;

void dyn_error();


main()
{
/* Call dyn_error() function on any error in
 * an embedded SQL statement.
 */
    EXEC SQL WHENEVER SQLERROR DO dyn_error("Oracle error");

/* Save text of SQL current statement in the ORACA if an
 * error occurs.
 */
    oraca.orastxtf = ORASTFERR;

/* Connect to Oracle. */

    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("¥nConnected to Oracle.¥n");

/* Assign a SQL query to the VARCHAR dynstmt.  Both the
 * array and the length parts must be set properly.  Note
 * that the query contains one host-variable placeholder,
 * v1, for which an actual input host variable must be
 * supplied at OPEN time.
 */
    strcpy(dynstmt.arr,
        "SELECT ename FROM emp WHERE deptno = :v1");
    dynstmt.len = strlen(dynstmt.arr);

/* Display the SQL statement and its current input host
 * variable.
 */
    puts((char *) dynstmt.arr);
    printf("   v1 = %d¥n", deptno);
    printf("¥nEmployee¥n");
    printf("--------¥n");

/* The PREPARE statement associates a statement name with
 * a string containing a SELECT statement.  The statement
 * name is a SQL identifier, not a host variable, and
 * therefore does not appear in the Declare Section.

 * A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
 */
    EXEC SQL PREPARE S FROM :dynstmt;

/* The DECLARE statement associates a cursor with a
 * PREPAREd statement.  The cursor name, like the statement
 * name, does not appear in the Declare Section.

 * A single cursor name cannot be DECLAREd more than once.
 */
    EXEC SQL DECLARE C CURSOR FOR S;
```

```
/* The OPEN statement evaluates the active set of the
 * PREPAREd query USING the specified input host variables,
 * which are substituted positionally for placeholders in
 * the PREPAREd query.  For each occurrence of a
 * placeholder in the statement there must be a variable
 * in the USING clause.  That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.

 * The USING clause can be omitted only if the statement
 * contains no placeholders.  OPEN places the cursor at the
 * first row of the active set in preparation for a FETCH.

 * A single DECLAREd cursor can be OPENed more than once,
 * optionally USING different input host variables.
 */
    EXEC SQL OPEN C USING :deptno;

/* Break the loop when all data have been retrieved. */

    EXEC SQL WHENEVER NOT FOUND DO break;

/* Loop until the NOT FOUND condition is detected. */

    for (;;)
    {
/* The FETCH statement places the select list of the
 * current row into the variables specified by the INTO
 * clause, then advances the cursor to the next row.  If
 * there are more select-list fields than output host
 * variables, the extra fields will not be returned.
 * Specifying more output host variables than select-list
 * fields results in an ORACLE error.
 */
        EXEC SQL FETCH C INTO :ename;

/* Null-terminate the array before output. */
        ename.arr[ename.len] = '¥0';
        puts((char *) ename.arr);
        }

/* Print the cumulative number of rows processed by the
 * current SQL statement.
 */
    printf("¥nQuery returned %d row%s.¥n¥n", sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2] == 1) ? "" : "s");

/* The CLOSE statement releases resources associated with
 * the cursor.
 */
    EXEC SQL CLOSE C;

/* Commit any pending changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;
    puts("Sayonara.¥n");
    exit(0);
}

void
dyn_error(msg)
```

```
char *msg;
{
    printf("¥n%s", msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '¥0';
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '¥0';
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '¥0';
    printf("¥n%s¥n", sqlca.sqlerrm.sqlerrmc);
    printf("in ¥"%s...¥"¥n", oraca.orastxt.orastxtc);
    printf("on line %d of %s.¥n¥n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Release resources associated with the cursor. */
    EXEC SQL CLOSE C;

/* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

13.10 方法4の使用方法

この項では、動的SQL方法4の概要を説明します。Oracle動的SQL方法4は、オブジェクト型、結果セット、構造体の配列およびLOBをサポートしていません。

ANSI SQLでは、すべてのデータ型がサポートされます。すべての新しいアプリケーションでは、ANSI SQLを使用してください。

方法3を使用したプログラムでも処理できない種類の動的SQL文があります。選択リストの項目数または入力ホスト変数のプレースホルダの数が実行時までわからない場合は、プログラムで記述子を使用する必要があります。記述子とは、プログラムおよびOracleが動的SQL文内の変数の完全な記述を保存するためのメモリー領域です。

複数行の問合せのときに、宣言済の出力ホスト変数のリスト内に列の値をFETCH INTOしたことを思い出してください。この選択リストがわからないときは、プリコンパイル時にINTO句でホスト変数リストを作成できません。たとえば、次の問合せでは2つの列値が戻されます。

```
SELECT ename, empno FROM emp WHERE deptno = :dept_number;
```

ただし、この選択リストをユーザーに定義させると、その問合せによって戻される列の数はわからなくなります。

関連項目

- ANSI動的SQL
- Oracle動的SQL: 方法4

13.10.1 SQLDAの必要性

このような種類の動的問合せを処理するには、プログラムでDESCRIBE SELECT LISTコマンドを発行するとともに、SQL記述子領域(SQLDA)というデータ構造体を宣言する必要があります。この構造体は問合せ選択リストの列の記述を保持しているため、選択記述子とも呼ばれます。

また、動的SQL文で入力ホスト変数のプレースホルダの数が明確でない場合、プリコンパイル時にUSING句でホスト変数リストを作成できません。

このような動的SQL文を処理するには、プログラムでDESCRIBE BIND VARIABLESコマンドを発行し、入力ホスト変数のプレースホルダの説明を保存するために、バインド記述子という別の種類のSQLDAを宣言する必要があります。(入力ホスト変数はバインド変数とも呼ばれます。)

プログラムにアクティブSQL文が複数ある(たとえば、複数のカーソルをOPENしている)場合、それぞれの文には専用のSQLDAが必要になります。ただし、非並行のカーソルではSQLDAを再利用できます。なお、1つのプログラム内のSQLDAの数に制限はありません。

13.10.2 DESCRIBE文

DESCRIBEは、選択リスト項目または入力ホスト変数の記述を保持するために記述子を初期化します。

選択記述子を指定すると、DESCRIBE SELECT LIST文によってPREPAREされた動的問合せ内の各選択リスト項目が調べられ、その名前、データ型、制約、長さ、位取りおよび精度が確認されます。その後、この情報は選択記述子に格納されます。

バインド記述子を指定すると、DESCRIBE BIND VARIABLES文によってPREPAREされた動的SQL文の各プレースホルダが調べられ、その名前および長さ、関連付けられた入力ホスト変数のデータ型が確認されます。続いて、この情報がそのバインド記述子に格納されます。たとえば、プレースホルダ名を使用して、ユーザーに入力ホスト変数の値の入力を要求できます。

13.10.3 SQLDA

SQLDAはホスト・プログラムのデータ構造体です。この構造体は選択リスト項目または入力ホスト変数の記述を保持します。

SQLDA変数は、宣言部で定義されません。

選択SQLDAには問合せ選択リストに関する次の情報が格納されています。

- DESCRIBEできる列の最大数
- DESCRIBEによって検出された列の実際の数
- 列値を格納するバッファのアドレス
- 列値の長さ
- 列値のデータ型
- 標識変数の値のアドレス
- 列名を格納するバッファのアドレス
- 列名を格納するバッファのサイズ
- 列名の現行の長さ

バインドSQLDAには、SQL文の入力ホスト変数に関する次の情報が格納されています。

- DESCRIBEできるプレースホルダの最大数
- 実際にDESCRIBEで検出されたプレースホルダの数
- 入力ホスト変数のアドレス
- 入力ホスト変数の長さ
- 入力ホスト変数のデータ型
- 標識変数のアドレス

- プレースホルダ名を格納するバッファのアドレス
- プレースホルダ名を格納するバッファのサイズ
- プレースホルダ名の現在の長さ
- 標識変数名を格納するバッファのアドレス
- 標識変数名を格納するバッファのサイズ
- 標識変数名の現行の長さ

関連項目:

SQLDA構造体と変数名については、Oracle動的SQL: 方法4を参照してください。

13.10.4 Oracle方法4の実装について

Oracle方法4では、一般に次の順序で埋込みSQL文を使用します。

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

ただし、選択記述子とバインド記述子が同時に動作する必要はありません。したがって、問合せ選択リストの列数が明確でも入力ホスト変数のプレースホルダの数が不明な場合は、方法4のOPEN文とともに次の方法3のFETCH文を使用できます。

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

逆に、入力ホスト変数のプレースホルダの数は明確でも問合せ選択リストの列数が不明な場合は、方法4のFETCH文とともに次の方法3のOPEN文を

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

使用できます。

方法4では、EXECUTEを非問合せにも使用できます。

13.10.5 制限事項

動的SQL方法4では、TABLE型のパラメータを使用して、ホスト配列をPL/SQLプロシージャにバインドすることはできません。

13.11 DECLARE STATEMENT文の使用方法について

方法2、3および4では、次の文を使用することが必要な場合があります。

```
EXEC SQL [AT db_name] DECLARE statement_name STATEMENT;
```

*db_name*および*statement_name*は、プリコンパイラで使用される識別子で、ホスト変数でもプログラム変数でもありません。

DECLARE STATEMENTによって動的SQL文の名前が宣言されます。すると、この動的SQL文はPREPARE、EXECUTE、DECLARE CURSORおよびDESCRIBEで参照できます。デフォルト以外のデータベースで動的SQL文を実行するときに、この文が必要になります。方法2での使用例を次に示します。

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL EXECUTE sql_stmt;
```

この例では、どこでSQL文をEXECUTEするかを*remote_db*によってOracleに指示します。

方法3および方法4では、次の例に示すようにDECLARE CURSOR文がPREPARE文の前にある場合にもDECLARE STATEMENTが必要です。

```
EXEC SQL DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
```

一般的な文の順序は次のとおりです。

```
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

13.11.1 ホスト配列の使用について

静的SQLおよび動的SQL内でのホスト配列の使用方法は似ています。たとえば、動的SQL方法2で入力ホスト配列を使用するには、次の構文を使用します。

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

*host_array_list*には1つ以上のホスト配列が含まれます。

同様に、方法3で入力ホスト配列を使用するには、次の構文を使用します。

```
OPEN cursor_name USING host_array_list;
```

方法3で出力ホスト配列を使用するには、次の構文を使用します。

```
FETCH cursor_name INTO host_array_list;
```

方法4では、オプションのFOR句を使用して、入力ホスト配列または出力ホスト配列のサイズをOracleに指示する必要があります。

関連項目

- Oracle動的SQL: 方法4

13.12 PL/SQLの使用について

Pro*C/C++プリコンパイラでは、PL/SQLブロックが単一のSQL文として取り扱われます。したがってSQL文と同様に、PL/SQLブロックを文字列のホスト変数またはリテラルに格納できます。文字列にPL/SQLブロックを格納する場合は、EXEC SQL EXECUTEキーワード、END-EXECキーワードおよび文の終了記号「;」を省略します。

ただし、プリコンパイラによるSQLとPL/SQLの処理方法には、次の2つの違いがあります。

- PL/SQLホスト変数のPL/SQLブロック内での役割が入力ホスト変数、出力ホスト変数、あるいはその両方のどれであっても、プリコンパイラではPL/SQLホスト変数はすべて入力ホスト変数として扱われます。

- PL/SQLブロックに格納できるSQL文の数には制限がないため、PL/SQLブロックからはFETCHできません。

13.12.1 方法1の場合

PL/SQLブロックにホスト変数が含まれていなければ、方法1で通常どおりPL/SQL文字列をEXECUTEできます。

13.12.2 方法2の場合

PL/SQLブロック内の入力ホスト変数および出力ホスト変数の数がわかっていれば、方法2で通常どおりPL/SQL文字列をPREPAREおよびEXECUTEできます。

USING句にはすべてのホスト変数を指定する必要があります。このPL/SQL文字列をEXECUTEすると、USING句内のホスト変数はPREPARE済の文字列内の対応するプレースホルダに置き換わります。プリコンパイラでPL/SQLホスト変数がすべて入力ホスト変数として扱われても、値は正しく割り当てられます。入力(プログラム)値は入力ホスト変数に割り当てられ、出力(列)値は出力ホスト変数に割り当てられます。

PREPAREされたPL/SQL文字列中のプレースホルダは、それぞれUSING句のホスト変数に対応している必要があります。したがって、PREPAREされた文に同じプレースホルダが2回以上現れるときは、それぞれがUSING句の個別のホスト変数に対応している必要があります。

13.12.3 方法3の場合

方法3は、FETCHが使用できることを除けば方法2と同じです。PL/SQLブロックからのFETCHはできないため、方法2を使用してください。

13.12.4 Oracle方法4の場合

PL/SQLブロックに数の不明な入力ホスト変数または出力ホスト変数が含まれている場合は、方法4を必ず使用します。

方法4を使用するには、すべての入力ホスト変数および出力ホスト変数について1つのバインド記述子を設定します。DESCRIBE BIND VARIABLESを実行すると、入力ホスト変数および出力ホスト変数に関する情報がそのバインド記述子に保存されます。プリコンパイラではPL/SQLホスト変数がすべて入力ホスト変数として扱われるため、DESCRIBE SELECT LISTを実行しても効果はありません。

警告:

動的 SQL 方法 4 では、TABLE 型のパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。

警告:

ANSI では行終了文字が無視されるため、動的に処理される PL/SQL ブロックでは ANSI 形式のコメント(- -)は使用しないでください。ANSI で記述すると、行の終わりではなくブロックの終わりまでコメントは続きます。ANSI 形式のコメントではなく、C 形式のコメント(/*... */)を使用してください。

関連項目

- Oracle動的SQL: 方法4

13.13 動的SQL文のキャッシング

文キャッシュは、セッションごとの文のキャッシュを提供および管理する機能です。サーバーでは、文を再び解析することなく、カーソルがいつでも使用できるようになっていることを意味します。文のキャッシングは、プリコンパイラ・アプリケーションで有効にでき、動的SQL文に依存するすべてのアプリケーションのパフォーマンス向上に役立ちます。パフォーマンスの向上は、動的文を再利用する際の解析のオーバーヘッドをなくすことで達成されます。

このパフォーマンスの改善は、動的文のキャッシングが可能になる新しいコマンドライン・オプションstmt_cache(文のキャッシュ・サイズ用)を使用することで実現します。この新しいオプションを有効にすると、セッション作成時に文のキャッシュが作成されます。キャッシングは動的文に対してのみ適用され、静的文用のカーソル・キャッシュとこの機能は共存します。

コマンドライン・オプションstmt_cacheには、0から65535の範囲で任意の値を指定できます。デフォルト(値0)で、文キャッシングは無効化されています。stmt_cacheオプションでは、アプリケーションにそれぞれの動的SQL文の予測数を保持するように設定できます。

例13-1 stmt_cacheオプションの使用方法

次の例は、stmt_cacheオプションの使用方法を示しています。このプログラムでは、表に行を挿入し、挿入した行をループ内のカーソルを使用して選択します。このプログラムのプリコンパイルにstmt_cacheオプションを使用すると、通常のプリコンパイルよりもパフォーマンスが向上します。

```
/*
 *  stmtcache.pc
 *
 *  NOTE:
 *  When this program is used to measure the performance with and without
 *  stmt_cache option, do the following changes in the program,
 *  1. Increase ROWSCNT to high value, say 10000.
 *  2. Remove all the print statements, usually which comsumes significant
 *     portion of the total program execution time.
 *
 *  HINT: In Linux, gettimeofday() can be used to measure time.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <oraca.h>

#define ROWSCNT 10

char    *username = "scott";
char    *password = "tiger";

/* Function prototypes */
void sql_error(char *msg);
void selectdata();
void insertdata();

int main()
{
  EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");

  /* Connect using the default schema scott/tiger */
```

```
  EXEC SQL CONNECT :username IDENTIFIED BY :password;

  /* core functions to insert and select the data */
  insertdata();
  selectdata();

/* Rollback pll the changes and disconnect from Oracle. */
  EXEC SQL ROLLBACK WORK RELEASE;

  exit(0);
}

/*Insert the data for ROWSCNT items into tpc2sc01 */
void insertdata()
{
  varchar dynstmt[80];
  int i;
  varchar ename[10];
  float comm;
  char *str;

  /* Allocates temporary buffer */
  str = (char *)malloc (11 * sizeof(char));

  strcpy ((char *)dynstmt.arr,
          "INSERT INTO bonus (ename, comm) VALUES (:ename, :comm)");
  dynstmt.len = strlen(dynstmt.arr);
  EXEC SQL PREPARE S FROM :dynstmt;

  printf ("Inserts %d rows into bonus table using dynamic SQL statement¥n",
          ROWSCNT);
  for (i=1; i<=ROWSCNT; i++)
  {
    sprintf (str, "EMP_%05d",i);
    strcpy (ename.arr, str);
    comm = i;
    ename.len = strlen (ename.arr);
    EXEC SQL EXECUTE S USING :ename, :comm;
  }

  free(str);
}

/* Select the data using the cursor */
void selectdata()
{
  varchar dynstmt[80];
  varchar ename[10];
  float comm;
  int i;

  strcpy((char *)dynstmt.arr,
         "SELECT ename, comm FROM bonus WHERE comm = :v1");
  dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);

  printf ("Fetches the inserted rows using using dynamic SQL statement¥n¥n");
  printf ("  ENAME      COMMISSION¥n¥n");

  for (i=1; i<=ROWSCNT; i++)
  {
```

```
    /* Do the prepare in the loop so that the advantage of stmt_caching
       is visible*/
    EXEC SQL PREPARE S FROM :dynstmt;

    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C USING :i;

    EXEC SQL WHENEVER NOT FOUND DO break;

    /* Loop until the NOT FOUND condition is detected. */
    for (;;)
    {
      EXEC SQL FETCH C INTO :ename, :comm;
      ename.arr[ename.len] = '¥0';
      printf ("%10s    %7.2f¥n", ename.arr, comm);
    }
    /* Close the cursor so that the reparsing is not required for stmt_cache */
    EXEC SQL CLOSE C;
  }
}

void sql_error(char *msg)
{
    printf("¥n%s", msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '¥0';
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '¥0';
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '¥0';
    printf("¥n%s¥n", sqlca.sqlerrm.sqlerrmc);
    printf("in ¥"%s...¥"¥n", oraca.orastxt.orastxtc);
    printf("on line %d of %s.¥n¥n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);

   /* Disable ORACLE error checking to avoid an infinite loop
    * should another error occur within this routine.
    */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Release resources associated with the cursor. */
    EXEC SQL CLOSE C;

    /* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

14 ANSI動的SQL

この章では、新しい方法4アプリケーションに使用するANSI動的SQL(SQL標準動的SQLとも呼ばれます)のOracleでの実装を説明します。これは、従来のOracle動的SQL方法4に拡張機能を加えたものです。詳細は、前の章の「Oracle動的SQL: 方法4」を参照してください。

ANSI方法4では、すべてのOracle型がサポートされます。従来のOracle方法4では、オブジェクト型、カーソル変数、構造体の配列、DML RETURNING句、Unicode変数およびLOBはサポートされませんでした。

従来のOracle動的SQL方法4では記述子はユーザーのPro*C/C++プログラムで定義されますが、ANSI動的SQLでは記述子はOracle内部で管理されます。どちらの場合でも、方法4では、様々な数のホスト変数を含むSQL文をPro*C/C++プログラムで受け取ったり作成したりできます。

この章のトピックは、次のとおりです:

- ANSI動的SQLの基礎
- ANSI SQL文の概要
- Oracle拡張機能
- ANSI動的SQLのプリコンパイラ・オプション
- 動的SQL文の構文
- サンプル・プログラム

14.1 ANSI動的SQLの基礎

次のSQL文について考えます。

```
SELECT ename, empno FROM emp WHERE deptno = :deptno_data
```

ANSI動的SQLを使用する手順は、次のとおりです。

- 実行する文を格納する文字列などの変数を宣言します。
- 入力変数および出力変数に記述子を割り当てます。
- 文を準備します。
- 入力記述子の入力を記述します。
- 入力記述子を設定します(この例では1つの入力ホスト・バインド変数deptno_data)。
- 動的カーソルを宣言およびオープンします。
- 出力記述子(上の例の出力ホスト変数は、enameおよびempno)を設定します。
- enameおよびempnoデータ・フィールドを各行から取り出すために、GET DESCRIPTORを使用してデータを繰り返しフェッチします。
- 取り出したデータを処理します(出力など)。
- 動的カーソルを閉じ、入力記述子および出力記述子への割当てを解除します。

14.1.1 プリコンパイラのオプション

マイクロ・プリコンパイラ・オプションDYNAMICをANSIに設定するか、マクロ・オプションMODEをANSIに設定してください。これにより、DYNAMICのデフォルト値がANSIに設定されます。DYNAMICのもう1つの設定はORACLEです。

ANSI型コードを使用するには、TYPE_CODEプリコンパイラ・マイクロ・オプションをANSIに設定するか、MODEマクロ・オプションをANSIに設定します。これにより、TYPE_CODEのデフォルト設定がANSIに変更されます。TYPE_CODEをANSIに設定するには、DYNAMICもANSIに設定する必要があります。

「ANSI SQL文の概要」に記載されているANSI SQL型のOracleによる実装は、ANSI規格と完全には一致していません。たとえば、INTEGERとして宣言された列の記述では、NUMERICのコードが戻されます。OracleをANSI規格に近づけると、動作にわずかな変更が必要になる場合があります。使用中のアプリケーションをデータベース・プラットフォーム間で移植できるようにして、可能なかぎりANSI準拠にする場合、TYPE_CODEプリコンパイラ・オプションを設定したANSI型を使用してください。このような変更ができない場合は、TYPE_CODEをANSIに設定しないでください。

14.2 ANSI SQL文の概要

動的SQL文で記述子を使用する前に、記述子領域を割り当てます。

ALLOCATE DESCRIPTOR文の構文は次のとおりです。

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
  [WITH MAX {:occurrences | numeric_literal}];
```

グローバル記述子は、プログラム内の任意のモジュールで使用できます。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。デフォルトはローカルです。

記述子名のdesc_namには、引用符で囲んだリテラルまたはホスト変数に格納した文字値を代入できます。

occurrencesは、記述子が保持できるバインド変数または列数の最大値です。数値リテラルを指定する必要があります。デフォルトは100です。

記述子が必要なくなった場合は、割当てを解除してメモリーを節約します。それ以外の場合には、アクティブなデータベース接続がなくなった時点で自動的に割当てが解除されます。

割当て解除文は次のとおりです。

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal};
```

準備済のSQL文に関する情報を取得するには、DESCRIBE文を使用します。DESCRIBE INPUTでは、準備済の動的文のバインド変数が記述されます。DESCRIBE OUTPUT (デフォルト)では、出力列の番号、型および長さが記述されます。単純化した構文は次のとおりです。

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] sql_statement
    USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal};
```

SQL文に入力値および出力値がある場合は、入力値および出力値に1つずつ記述子を割り当てる必要があります。入力値がない次の例のような場合があります。

```
SELECT ename, empno FROM emp ;
```

この場合、入力記述子は必要ありません。

SELECT文のINSERTS、UPDATES、DELETESおよびWHERE句の入力値を指定するには、SET DESCRIPTOR文を使用します。入力記述子内にDESCRIBEしていないときに入力バインド変数の数(COUNTに格納されています)を設定するに

は、SET DESCRIPTOR文を使用します。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   COUNT = {:kount | numeric_literal};
```

kountには、ホスト変数または数値リテラル(5など)を指定できます。SET DESCRIPTOR文を使用して、各ホスト変数に少なくともデータ・ソースを指定してください。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number DATA = :hv3;
```

また、次のように入力ホスト変数の型および長さを設定することもできます。

> **注意:**
>
> TYPE_CODE=ORACLE のとき、SET 文を使用して明示的に、または DESCRIBE OUTPUT によって暗黙的に TYPE および LENGTH を指定していない場合、プリコンパイラではホスト変数から導出された値が使用されます。TYPE_CODE=ANSI のときは、表 14-1 の値を使用して TYPE を設定する必要があります。また、ANSI デフォルト長はホスト変数に一致しないことがあるため、LENGTH も設定する必要があります。
>
> ```
> EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
> ```
>
> VALUE *item_number* TYPE = :*hv1*, LENGTH = :*hv2*, DATA = :*hv3*;

hv1、hv2およびhv3 などの識別子は、ホスト変数から値を供給する必要があることをユーザーが忘れないようにするために使用します。*item_number*は、入力変数のSQL文内での位置を表します。

TYPE_CODEをANSIに設定した場合は、次の表の型コードからTYPEを選択します。

表14-1 ANSI SQLデータ型

データ型	型コード
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2

データ型	型コード
REAL	7
SMALLINT	5

DATAは、入力されるホスト変数の値です。

インジケータ、精度、位取りなど、その他の入力値を設定することもできます。

SET DESCRIPTOR文の数値は、intまたはshort intで宣言する必要があります。ただし、インジケータおよび戻された長さの値はshort intとして宣言する必要があります。

たとえば、次の例でempnoを取得する場合、empnoは動的SQL文の2番目の出力ホスト変数であるため、値をVALUE = 2に設定します。ホスト変数empno_typは、3(Oracleタイプの整数値)に設定します。ホスト整数の長さを表すempno_lenは、4に設定します。この値はホスト変数のサイズです。DATAはホスト変数empno_dataと等しくなります。この変数は値をデータベース表から受け取ります。コード例は、次のようになります。

```
...
char *dyn_statement = "SELECT ename, empno FROM emp
   WHERE deptno = :deptno_number" ;
int empno_data ;
int empno_typ = 3 ;
int empno_len = 4 ;
...
EXEC SQL SET DESCRIPTOR 'out' VALUE 2  TYPE = :empno_typ, LENGTH = :empno_len,
   DATA = :empno_data ;
```

入力値を設定後に、入力記述子を使用して文を実行またはオープンします。文中に出力値がある場合、FETCHを行う前に出力値を設定してください。DESCRIBE OUTPUTを実行している場合は、ホスト変数の実際の型と長さのテストが必要になる場合があります。DESCRIBEを実行すると、ホスト変数の外部型および長さとは異なる内部型と長さが生成されます。

出力記述子をFETCHした後、戻されたデータにアクセスするには、GET DESCRIPTORを使用します。簡略化された構文は次のとおりです。構文の詳細は、この章の後半部分を参照してください。

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_litera/}
   VALUE item_number :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH ;
```

desc_namおよびitem_numberには、リテラルまたはホスト変数を指定できます。記述子名には、リテラル(outなど)を指定できます。項目番号には、数値リテラル(2など)を指定できます。

hv1、hv2およびhv3は、それぞれホスト変数です。ここにはホスト変数を指定する必要があり、リテラルは指定できません。例では、戻されるデータを3つのみ示しています。

数値すべてに**long、int**または**short**のいずれかを指定します。ただし、インジケータまたは戻された長さの値はshortにする必要があります。

関連項目:

- Oracle型コードは表15-2を参照してください
- 可能な記述子項目名の詳細は、SET DESCRIPTORを参照してください。
- 戻されたデータから取得できるすべての項目の一覧については、表14-4を参照してください。

14.2.1 サンプル･コード

次の例は、ANSI動的SQLの使用方法を示しています。ここでは入力記述子('in')および出力記述子('out')を割り当ててSELECT文を実行します。入力値はSET DESCRIPTOR文を使用して設定します。カーソルはオープンおよびフェッチされ、結果の出力値はGET DESCRIPTOR文を使用して取得されます。

```
...
char* dyn_statement = "SELECT ename, empno FROM emp WHERE deptno = :deptno_data" ;
int deptno_type = 3, deptno_len = 2, deptno_data = 10 ;
int ename_type = 97, ename_len = 30 ;
char ename_data[31] ;
int empno_type = 3, empno_len = 4 ;
int empno_data ;
long SQLCODE = 0 ;
...
main ()
{
/* Place preliminary code, including connection, here. */
...
EXEC SQL ALLOCATE DESCRIPTOR 'in' ;
EXEC SQL ALLOCATE DESCRIPTOR 'out' ;
EXEC SQL PREPARE s FROM :dyn_statement ;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' ;
EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE = :deptno_type,
   LENGTH = :deptno_len, DATA = :deptno_data ;
EXEC SQL DECLARE c CURSOR FOR s ;
EXEC SQL OPEN c USING DESCRIPTOR 'in' ;
EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out' ;
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
    LENGTH = :ename_len, DATA = :ename_data ;
EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE = :empno_type,
    LENGTH = :empno_len, DATA = :empno_data ;

EXEC SQL WHENEVER NOT FOUND DO BREAK ;
while (SQLCODE == 0)
{
   EXEC SQL FETCH c INTO DESCRIPTOR 'out' ;
   EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :ename_data = DATA ;
   EXEC SQL GET DESCRIPTOR 'out' VALUE 2 :empno_data = DATA ;
   printf("¥nEname = %s Empno = %s", ename_data, empno_data) ;
}
EXEC SQL CLOSE c ;
EXEC SQL DEALLOCATE DESCRIPTOR 'in' ;
EXEC SQL DEALLOCATE DESCRIPTOR 'out' ;
...
}
```

ANSI動的SQLとともにスクロール可能カーソルを使用することもできます。ANSI動的SQLをスクロール可能カーソルとともに使用するには、カーソルをSCROLLモードでDECLAREします。FETCH文に様々なフェッチ方向を使用して、結果セットにアクセスします。

14.3 Oracle拡張機能

この項では、次の拡張機能を説明します。

- SET文のデータ項目の参照セマンティクス。
- バルク操作のための配列。
- オブジェクト型、NCHAR列およびLOBのサポート。

14.3.1 参照セマンティクス

ANSI規格では、値構文が指定されています。パフォーマンス向上のために、Oracleではこの規格を拡張して参照セマンティクスを導入しています。

値構文では、ホスト変数データのコピーを作成します。参照セマンティクスでは、ホスト変数のアドレスを使用し、コピーは行いません。そのため、参照セマンティクスを使用すると、大量データ処理のパフォーマンスが向上します。

フェッチの速度向上には、データ句の前にREFキーワードを使用します。

```
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
   LENGTH = :ename_len, REF DATA = :ename_data ;
EXEC SQL DESCRIPTOR 'out' VALUE 2 TYPE = :empno_type,
   LENGTH = :empno_len, REF DATA = :empno_data ;
```

これにより、取り出された結果がホスト変数に渡されます。GET文は必要ありません。FETCHが実行されるたびに、取り出されたデータは、ename_dataおよびempno_dataに直接書き込まれます。

次のコード例に示すように、REFキーワードが使用できるのは、DATA、INDICATORおよびRETURNED_LENGTH項目(フェッチする行によって変わる可能性があります)の前に限られます。

```
int indi, returnLen ;
...
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
   LENGTH = :ename_len, REF DATA = :ename_data,
      REF INDICATOR = :indi, REF RETURNED_LENGTH = :returnLen ;
```

フェッチするたびに、returnLenによりenameフィールドの実際に取得した長さが保持されます。このフィールドはCHARまたはVARCHAR2データで使用すると便利です。

ename_lenには取り出された長さは渡されません。これは、FETCH文によって変更されません。データの行をフェッチする前に列の最大幅を調べるには、GET文の前にDESCRIBEを使用します。

REFキーワードは、SELECT以外のSQL文の処理速度向上のためにも使用します。参照セマンティクスの場合、記述子領域にコピーされた値ではなくホスト変数が使用されます。SQL文を実行する時点でのホスト変数データが使用されるのであって、SETの時点でのデータではありません。次はその例です。

```
int x = 1 ;
EXEC SQL SET DESCRIPTOR 'value' VALUE 1 DATA = :x ;
EXEC SQL SET DESCRIPTOR 'reference' VALUE 1 REF DATA = :x ;
x = 2 ;
EXEC SQL EXECUTE s USING  DESCRIPTOR 'value' ;    /* Will use  x = 1 */
EXEC SQL EXECUTE s USING DESCRIPTOR 'reference' ; /* Will use x = 2 */
```

関連項目

- SET DESCRIPTOR

14.3.2 配列を使用したバルク操作について

Oracleによりバルク操作機能が追加され、ANSI動的SQLが拡張されました。バルク操作を行うには、処理する入力データ量または行数を指定するために、FOR句で配列サイズを指定します。

FOR句は、ALLOCATE文で最大データ量または最大行数の指定に使用します。最大配列サイズ100を指定するには、次のように記述します。

```
EXEC SQL FOR 100 ALLOCATE DESCRIPTOR 'out' ;
```

または

```
int array_size = 100 ;
...
EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'out' ;
```

FOR句は、記述子にアクセスする後続の文で使用されます。次に示すように、出力記述子では、ALLOCATE文で指定した配列サイズと等しいか、それよりも小さい配列サイズをFETCH文に割り当てる必要があります。

```
EXEC SQL FOR 20 FETCH c1 USING DESCRIPTOR 'out' ;
```

後続の、同じ記述子のDATA、INDICATORまたはRETURNED_LENGTH値を取得するGET文では、FETCH文と同じ配列サイズを指定する必要があります。

```
int val_data[20] ;
short val_indi[20] ;
...
EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :val_data = DATA,
  :val_indi = INDICATOR ;
```

ただし、LENGTH、TYPE、COUNTなど、行によって変化しない項目を参照するGET文では、FOR句は使用しないでください。

```
int cnt, len ;
...
EXEC SQL GET DESCRIPTOR 'out' :cnt = COUNT ;
EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :len = LENGTH ;
```

これは、参照セマンティクスを使用したSET文でも同じです。FETCHの前にあり、DATA、INDICATORまたはRETURNED_LENGTHに対する参照セマンティクスを使用したSET文には、FETCHと同じ配列サイズを指定する必要があります。

```
int ref_data[20] ;
short ref_indi[20] ;
...
EXEC SQL FOR 20 SET DESCRIPTOR 'out' VALUE 1 REF DATA = :ref_data,
   REF INDICATOR = :ref_indi ;
```

同様に、行のバッチの挿入など、入力に使用する記述子でも、ALLOCATE文で使用した配列サイズと等しいか、それよりも小さいサイズの配列サイズをEXECUTEまたはOPEN文に使用する必要があります。値および参照セマンティクスのどちらも、DATA、INDICATORまたはRETURNED_LENGTHにアクセスするSET文では、EXECUTE文と同じ配列サイズを使用する必要があります。

FOR句は、DEALLOCATEまたはPREPARE文では使用しません。

次のコード例に、出力記述子のないバルク操作の例を示します(出力はなく、表empに挿入する入力のみあります)。COUNTの値は2です(INSERT文にename_arrおよびempno_arrの2つのホスト変数があることを示します)。データ配列ename_arrには、順番に「Tom」、「Dick」および「Harry」という3つの文字列が保持されています。インジケータ配列ename_indには2つ目の要素に-1の値を設定しているため、「Dick」ではなくNULLが挿入されます。データ配列empno_arrには、従業員番号が3つ含まれています。DML RETURNING句を使用すると、実際に挿入された名前を確認できます。

```
...
char* dyn_statement = "INSERT INTO emp (ename) VALUES (:ename_arr)" ;
```

```
char ename_arr[3][6] = {Tom","Dick","Harry"} ;
short ename_ind[3] = {0,-1,0} ;
int ename_len = 6, ename_type = 97, cnt = 2 ;
int empno_arr[3] = {8001, 8002, 8003} ;
int empno_len = 4 ;
int empno_type = 3 ;
int array_size = 3 ;
EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'in' ;
EXEC SQL SET DESCRIPTOR 'in' COUNT = :cnt ;
EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE = :ename_type, LENGTH = :ename_len ;
EXEC SQL SET DESCRIPTOR 'in' VALUE 2 TYPE = :empno_type, LENGTH = :empno_len ;
EXEC SQL FOR :array_size SET DESCRIPTOR 'in' VALUE 1
   DATA = :ename_arr, INDICATOR = :ename_ind ;
EXEC SQL FOR :array_size SET DESCRIPTOR 'in' VALUE 2
   DATA = :empno_arr ;
EXEC SQL PREPARE s FROM :dyn_statement ;
EXEC SQL FOR :array_size EXECUTE s USING DESCRIPTOR 'in' ;
...
```

上のコードを実行すると、次の値が挿入されます。

```
EMPNO   ENAME
 8001   Tom
 8002
 8003   Harry
```

関連項目

- DML RETURNING句
- FOR句の使用について

14.3.3 構造体配列のサポート

HOST_STRIDE_LENGTHを構造体のサイズに、INDICATOR_STRIDE_LENGTHをインジケータ構造体のサイズに、そしてRETURNED_LENGTH_STRIDEを戻された長さの構造体のサイズに設定する必要があります。

構造体の配列は、ANSI動的SQLによってサポートされていますが、従来のOracle動的SQLではサポートされていません。

14.3.4 オブジェクト型のサポート

独自に定義したオブジェクト型では、Oracle TYPEを108にして使用してください。オブジェクト型の列では、DESCRIBE文を使用してUSER_DEFINED_TYPE_VERSION、USER_DEFINED_TYPE_NAME、USER_DEFINED_TYPE_NAME_LENGTH、USER_DEFINED_TYPE_SCHEMAおよびUSER_DEFINED_TYPE_SCHEMA_LENGTHを取得します。

DESCRIBE文を使用しないでこれらの値を取得する場合は、SET DESCRIPTOR文を使用して自分で設定を行う必要があります。

14.4 ANSI動的SQLのプリコンパイラ・オプション

マクロ・オプションのMODEを使用すると、ANSIと互換性のある特性の設定や、多くの機能の制御ができます。値にはANSIまたはORACLEを設定できます。個々の機能には、MODE設定に優先するマイクロ・オプションがあります。

動的SQLでの記述子の動作を指定する場合は、プリコンパイラ・マイクロ・オプション*DYNAMIC*を使用します。ANSIとORACLEのどちらのデータ型を使用するかを指定する場合は、プリコンパイラ・マイクロ・オプション*TYPE_CODE*を使用します。

マクロ・オプションMODEをANSIに設定すると、マイクロ・オプションDYNAMICは自動的にANSIになります。MODEをORACLEに設定すると、DYNAMICはORACLEになります。

DYNAMICおよびTYPE_CODEはインラインでは使用できません。

次の表に機能とDYNAMICの設定がその機能に与える影響を示します。

表14-2 DYNAMICオプションの設定

機能	DYNAMIC = ANSI	DYNAMIC = ORACLE
記述子の作成。	ALLOCATE 文を使用する必要があります。	関数 SQLSQLDAAlloc()を使用する必要があります。
記述子の破壊。	DEALLOCATE 文を使用できます。	関数 SQLLDAFree()が使用可能です。
データの取得。	FETCH 文および GET 文のどちらも使用できます。	FETCH 文のみ使用できます。
入力データの設定。	DESCRIBE INPUT 文を使用できます。SET 文を使用する必要があります。	コードに記述子値を設定する必要があります。DESCRIBE BIND VARIABLES 文を使用する必要があります。
記述子の表現。	引用符付きのリテラル、または記述子名を含むホスト識別子。	ホスト変数、SQLDA を指すポインタ。
利用可能なデータ型。	BIT を除くすべての ANSI 型、およびすべての Oracle 型。	オブジェクト、LOB、構造体の配列およびカーソル変数以外の Oracle 型。

マイクロ・オプションTYPE_CODEは、プリコンパイラによってマクロ・オプションMODEと同じ値に設定されます。DYNAMICがANSIの場合、TYPE_CODEはANSI以外には設定できません。

TYPE_CODE設定に対応する機能は次のとおりです。

表14-3 TYPE_CODEオプションの設定

機能	TYPE_CODE = ANSI	TYPE_CODE = ORACLE
動的 SQL からの入出力に使用するデータ型コード番号。	ANSI 型があるときは ANSI コード番号を使用します。ない場合は Oracle コード番号の負の値を使用します。 DYNAMIC = ANSI のときにのみ有効です。	Oracle コード番号を使用します。 DYNAMIC の設定に関係なく使用できます。

関連項目

- MODE
- SQLLIBパブリック関数の新しい名前

14.5 動的SQL文の構文

ここで説明するすべての文の詳細は、埋込みSQL文およびディレクティブを参照してください。

14.5.1 ALLOCATE DESCRIPTOR

用途

この文は、SQL記述子領域を割り当てるために使用します。記述子、ホスト・バインド項目発生数の最大値および配列サイズを指定します。この文は、ANSI動的SQLでのみ使用できます。

構文

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_litera/} [WITH MAX occurrences] ;
```

変数

array_size

これは配列処理をサポートするオプション句(Oracle拡張機能)です。この句により、配列処理で記述子が使用可能であることがプリコンパイラに通知されます。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子の名前。ローカル記述子は、モジュール内で一意にしてください。前回の割当てを解除せずに記述子を割り当てた場合は、ランタイム・エラーが生成されます。グローバル記述子は、アプリケーション全体で一意にしてください。そうでない場合はランタイム・エラーが発生します。

occurrences

記述子で使用可能なホスト変数の最大数です。この値は、0から64KBの整数定数にする必要があります。それ以外の場合はエラーが戻されます。デフォルトは100です。この句はオプションです。これらの規則に違反するとプリコンパイラ・エラーが戻されます。

例

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 ;

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;
```

14.5.2 DEALLOCATE DESCRIPTOR

用途

以前に割り当てられたSQL記述子の割当てを解除してメモリーを解放する場合に、この文を使用します。この文は、ANSI動的SQLでのみ使用できます。

構文

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_litera/} ;
```

変数

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

同じ名前およびスコープの記述子が割り当てられていない場合、または割当てが解除されている場合は、ランタイム・エラーが発生します。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' ;

EXEC SQL DEALLOCATE DESCRIPTOR :binddes ;
```

14.5.3 GET DESCRIPTOR

用途

この文は、SQL記述子領域からの情報の取得に使用します。

構文

```
EXEC SQL [FOR [:]array_size] GET DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
   { :hv0  = COUNT | VALUE item_number
      :hv1 = item_name1 [ {, :hvN = item_nameN} ] } ;
```

変数

array_size

FOR array_sizeは、オプションのOracle拡張機能です。array_sizeは、FETCH文のarray_sizeフィールドと等しくする必要があります。

COUNT

バインド変数の合計数。

desc_nam

記述子の名前。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

VALUE item_number

SQL文内での項目の位置。item_numberには変数または定数を指定できます。item_numberの値がCOUNTより大きい場合、「データが見つかりません。」という条件が戻されます。item_numberには0より大きい値を指定する必要があります。

hv1 .. hvN

値の転送先のホスト変数。

item_name1 .. item_nameN

ホスト変数に対応付けられた記述子項目名。使用可能なANSI記述子項目名は次のとおりです。

表14-4 GET DESCRIPTORの記述子項目名の定義

記述子項目名	意味
TYPE	ANSI データ型が表にない場合および TYPE_CODE=ANSI の場合は、負の値の Oracle 型コードを使用してください。
LENGTH	列データの長さ。NCHAR では文字数、その他の場合はバイト数で表されます。DESCRIBE OUTPUT によって設定されます。
OCTET_LENGTH	バイト単位でのデータの長さ。
RETURNED_LENGTH	FETCH 後の実際のデータ長。
RETURNED_OCTET_LENGTH	戻されたデータのバイト単位での長さ。
PRECISION	桁数。
SCALE	真数値型での小数点の右側の桁数。
NULLABLE	1 のときは、列に NULL 値を使用できます。0 の場合、列には NULL 値を指定できません。
INDICATOR	対応付けられたインジケータ値。
DATA	データの値。
NAME	列の名前。
CHARACTER_SET_NAME	列のキャラクタ・セット。

追加されたOracle記述子項目の名前は次のとおりです。

表14-5 Oracle拡張機能により追加されたGET DESCRIPTORの記述子項目名の定義

記述子項目名	意味
NATIONAL_CHARACTER	2 は NCHAR または NVARCHAR2 を示します。1 の場合は文字です。0 の場合は文字以外の値です。
INTERNAL_LENGTH	内部でのバイト単位の長さ。
HOST_STRIDE_LENGTH	ホスト構造体のサイズ。バイト数で表します。

記述子項目名	意味
INDICATOR_STRIDE_LENGTH	インジケータ構造体のサイズ。バイト数で表します。
RETURNED_LENGTH_STRIDE	戻された長さの構造体のサイズ。バイト数で表します。
USER_DEFINED_TYPE_VERSION	オブジェクト型バージョンを表す文字。
USER_DEFINED_TYPE_NAME	オブジェクト型の名前。
USER_DEFINED_TYPE_NAME_LENGTH	オブジェクト型の名前の長さ。
USER_DEFINED_TYPE_SCHEMA	オブジェクト・スキーマを表す文字。
USER_DEFINED_TYPE_SCHEMA_LENGTH	USER_DEFINED_TYPE_SCHEMA の長さ。
NATIONAL_CHARACTER	2 は NCHAR または NVARCHAR2 を示します。1 の場合は文字です。0 の場合は文字以外の値です。

使用上の注意

FOR句は、DATA、INDICATORおよびRETURNED_LENGTH項目のみを含むGET DESCRIPTOR文で使用してください。

内部型は、DESCRIBE OUTPUT文によって設定されます。入力および出力のどちらでも、ホスト変数の外部型に使用する型を設定する必要があります。

TYPEはANSI SQLデータ型のコードです。ANSIの型が表に含まれていない場合は、負の値のOracle型コードを使用してください。

LENGTHには、固定幅の各国語キャラクタ・セットを持つフィールドの列の長さを表す文字数が含まれます。それ以外のキャラクタ列ではバイト単位になります。LENGTHはDESCRIBE OUTPUTによって設定されます。

RETURNED_LENGTHは、FETCH文によって設定される実際のデータ長です。LENGTHと同様にバイト単位または文字単位になります。フィールドOCTET_LENGTHおよびRETURNED_OCTET_LENGTHは、バイト単位の長さです。

NULLABLE = 1は、列にNULLを使用できることを示します。NULLABLE = 0は、列にNULLを使用できないことを示します。

CHARACTER_SET_NAMEは、キャラクタ列の場合にのみ意味があります。他の型では未定義になります。DESCRIBE OUTPUT文によって値が設定されます。

DATAおよびINDICATORは、その列のデータ値およびインジケータ・ステータスです。データがNULLでインジケータが要求されなかった場合は、実行時にエラーが発生します(DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER)。

Oracle固有の記述子項目名

列がNCHARまたはNVARCHAR2列の場合は、NATIONAL_CHARACTER = 2になります。列がキャラクタ(ただし、各国語キャラクタではない)列の場合、項目は1に設定されます。キャラクタ以外の列の場合、DESCRIBE OUTPUTの実行後にこ

の項目は0に設定されます。

INTERNAL_LENGTHは、Oracle動的方法4との互換性があるため、Oracle記述子領域の長さメンバーと同じ値に設定されます。この項目にはOracle SQL記述子領域の長さメンバーと同じ値が設定されています。

次の3つの項目はDESCRIBE OUTPUT文によって戻されません。

- ホスト変数構造体のサイズを示すHOST_STRIDE_LENGTH。
- 標識変数の構造体のサイズを示すINDICATOR_STRIDE_LENGTH。
- 戻された長さの変数構造体のサイズを示すRETURNED_LENGTH_STRIDE

次の項目は、プリコンパイラ・オプションOBJECTSがYESに設定されているときにのみオブジェクト型に適用されます。

- タイプ・バージョンを表す文字を含むUSER_DEFINED_TYPE_VERSION。
- 型の名前を表す文字を示すUSER_DEFINED_TYPE_NAME。
- 型の名前の長さをバイト数で示すUSER_DEFINED_TYPE_NAME_LENGTH。
- 型のスキーマ名を表す文字を示すUSER_DEFINED_TYPE_SCHEMA。
- 型のスキーマ名の長さを文字数で示すUSER_DEFINED_TYPE_SCHEMA_LENGTH。

例

```
EXEC SQL GET DESCRIPTOR :binddes :n = COUNT ;

EXEC SQL GET DESCRIPTOR 'SELDES' VALUE 1 :t = TYPE, :l = LENGTH ;

EXEC SQL FOR :batch GET DESCRIPTOR LOCAL 'SELDES'
   VALUE :sel_item_no :i = INDICATOR, :v = DATA ;
```

関連項目:

- ANSI型コードは表14-1を参照してください
- Oracle型コードは表15-2を参照してください
- Oracle動的SQL: 方法4

14.5.4 SET DESCRIPTOR

用途

この文は、ホスト変数からの記述子領域の情報を設定するために使用します。SET DESCRIPTOR文では、項目名のホスト変数のみサポートされます。

構文

```
EXEC SQL [FOR array_size] SET DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal} {COUNT = :hv0 |
   VALUE item_number
   [REF] item_name1 = :hv1
   [{, [REF] item_nameN = :hvN}]} ;
```

変数

array_size

このOracleオプション句で配列を使用できるのは、記述子項目DATA、INDICATORおよびRETURNED_LENGTHの設定時のみです。FOR句を含むSET DESCRIPTORでは他の項目を使用できません。ホスト変数配列サイズはすべて一致している必要があります。FETCH文で使用するのと同じ配列サイズをSET文で使用してください。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。ALLOCATE DESCRIPTORでの規則が適用されます。

COUNT

バインド(入力)変数または定義(出力)変数の数。

VALUE item_number

動的SQL文でのホスト変数の位置。

hv1 .. hvN

設定するホスト変数(定数ではありません)。

item_nameI

desc_item_nameでは、GET DESCRIPTOR構文と同様の方法でこれらの値が使用されます。

表14-6 SET DESCRIPTORの記述子項目名

記述子項目名	意味
TYPE	対応する ANSI 型がない場合は、負の値の Oracle 型を使用します。
LENGTH	列内のデータの最大長。
INDICATOR	対応付けられたインジケータ値。参照セマンティクスのために設定します。
DATA	設定するデータの値。参照セマンティクスのために設定します。
CHARACTER_SET_NAME	列のキャラクタ・セット。
TYPE	対応する ANSI 型がない場合は、負の値の Oracle 型を使用します。

ANSI型コードについては表14-1、Oracle型コードについては表15-2をそれぞれ参照してください。

Oracle拡張機能により追加された記述子項目の名前は次のとおりです。

表14-7 Oracle拡張機能により追加されたSET DESCRIPTORの記述子項目名の定義

記述子項目名	意味

記述子項目名	意味
RETURNED_LENGTH	FETCH 後に戻される長さ。参照セマンティクスを使用する場合に設定します。
NATIONAL_CHARACTER	入力ホスト変数が NCHAR または NVARCHAR2 型のときは、2 に設定します。 各国語キャラクタ設定が設定されていない場合、0 に設定します。
HOST_STRIDE_LENGTH	ホスト変数構造体のサイズ。バイト数で表します。
INDICATOR_STRIDE_LENGTH	標識変数のサイズ。バイト数で表します。
RETURNED_LENGTH_STRIDE	戻された長さの構造体のサイズ。バイト数で表します。
USER_DEFINED_TYPE_NAME	オブジェクト型の名前。
USER_DEFINED_TYPE_NAME_LENGTH	オブジェクト型の名前の長さ。
USER_DEFINED_TYPE_SCHEMA	オブジェクト・スキーマを表す文字。
USER_DEFINED_TYPE_SCHEMA_LENGTH	USER_DEFINED_TYPE_SCHEMA の長さ。

使用上の注意

参照セマンティクスは、パフォーマンス向上のために使用する別のオプションのOracle拡張機能です。記述子項目名がDATA、INDICATORおよびRETURNED_LENGTHの場合にのみ、それらの前にREFキーワードを指定します。REFキーワードを使用した場合は、GET文を使用する必要はありません。複合データ型(オブジェクト型、コレクション型、構造体の配列およびDML RETURNING句)はすべて、SET DESCRIPTORのREF形式を必要とします。

プログラムで別のSQLのDESCRIPTORを再利用する場合、DESCRIPTORの古い値は残ります。

REFを使用すると、対応付けられたホスト変数自体がSETで使用されます。この場合、GETは必要ありません。値構文ではなく、REFセマンティクスを使用するときにのみ、RETURNED_LENGTHを設定できます。

SETまたはGET文の配列サイズは、FETCHで使用する配列サイズと同じにしてください。

NCHARホスト変数には、NATIONAL_CHARフィールドを2に設定します。

古いSQLでNCHARホスト入力値用にDESCRIPTORが使用されるとき、NATIONAL_CHARACTERフィールドを0に設定します。

オブジェクト型の特性を設定するときは、USER_DEFINED_TYPE_NAMEおよびUSER_DEFINED_TYPE_NAME_LENGTHを設定する必要があります。

省略した場合は、USER_DEFINED_TYPE_SCHEMAおよびUSER_DEFINED_TYPE_SCHEMA_LENGTHはデフォルトで現行の接続に設定されます。

クライアント側でUnicodeをサポートするには、CHARACTER_SET_NAMEにUTF16を設定します。データはUCS2エンコーディングになり、RETURNED_LENGTHはCHARSです。

例

```
int bindno = 2 ;
short indi = -1 ;
char data = "ignore" ;
int batch = 1 ;

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR 'binddes' ;
EXEC SQL SET DESCRIPTOR GLOBAL :binddes COUNT = 3 ;
EXEC SQL FOR :batch SET DESCRIPTOR :bindes
   VALUE :bindno INDICATOR = :indi, DATA = :data ;
...
```

関連項目

- GET DESCRIPTOR
- DML RETURNING句
- 配列を使用したバルク操作について

関連項目:

14.5.5 PREPAREの使用

用途

この方法で使用されているPREPARE文は、他の動的SQL方法で使用されているPREPARE文と同じです。Oracle拡張機能によって、変数と同様にSQL文で引用符付き文字列を使用できます。

構文

```
EXEC SQL PREPARE statement_id FROM :sql_statement ;
```

変数

statement_id

これを宣言しないでください。これは未宣言のSQL識別子です。

sql_statement

埋込みSQL文を格納する文字列(定数または変数)。

例

```
char* statement = "SELECT ENAME FROM emp WHERE deptno = :d" ;
EXEC SQL PREPARE S1 FROM :statement ;
```

14.5.6 DESCRIBE INPUT

用途

この文はバインド変数についての情報を戻します。

構文

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam | string_literal} ;
```

変数

statement_id

PREPAREおよびDESCRIBE OUTPUTで使用するものと同じです。これを宣言しないでください。これは未宣言のSQL識別子です。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

DESCRIBE INPUTでは、COUNTおよびNAME項目のみが設定されます。

例

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :binddes ;
EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' ;
```

14.5.7 DESCRIBE OUTPUT

用途

PREPAREされた文の出力列についての情報を取得する場合は、この文を使用します。ANSI構文は、旧バージョンのOracle構文と異なります。SQL記述子領域に格納される情報は、戻された値の個数、および関連する情報(型、長さ、名前など)です。

構文

```
EXEC SQL DESCRIBE [OUTPUT] statement_id USING [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam | string_litera/} ;
```

変数

statement_id

PREPAREで使用するものと同じです。これを宣言しないでください。これは未宣言のSQL識別子です。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

OUTPUTがデフォルト設定で、これは省略できます。

例

```
char* desname = "SELDES" ;
EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES' ; /* Or, */
EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :desname ;
```

14.5.8 EXECUTE

用途

EXECUTEは、準備済のSQL文の入力変数および出力変数を照合し、文を実行します。EXECUTEのANSIバージョンは、1つの文中に2つの記述子を割り当てることでDML RETURNING句をサポートできる点で、旧バージョンのEXECUTE文とは異なります。

構文

```
EXEC SQL [FOR :array_size] EXECUTE statement_id
    [USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
        [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}] ;
```

変数

array_size

文が処理する行数。

statement_id

PREPAREで使用するものと同じです。これを宣言しないでください。これは未宣言のSQL識別子です。リテラルを指定できます。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

INTO句により、INSERT、UPDATEおよびDELETEのDML RETURNING句が実装されます。

例

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :binddes ;

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' ;
```

関連項目

- DML RETURNING句

14.5.9 EXECUTE IMMEDIATEの使用

用途

リテラルまたはSQL文を含むホスト変数文字列を実行します。この文のANSI SQL形式は、旧バージョンのOracle動的SQLと同じです。

構文

```
EXEC SQL EXECUTE IMMEDIATE {:sql_statement | string_literal}
```

変数

sql_statement

文字列内のSQL文またはPL/SQLブロック。

例

```
EXEC SQL EXECUTE IMMEDIATE :statement ;
```

14.5.10 DYNAMIC DECLARE CURSORの使用

用途

問合せ文に対応付けられたカーソルを宣言します。これは、一般的なカーソル宣言部です。

構文

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_id;
```

変数

cursor_name

カーソル変数(SQL識別子であり、ホスト変数ではありません)。

statement_id

未宣言のSQL識別子。

例

```
EXEC SQL DECLARE C1 CURSOR FOR S1 ;
```

14.5.11 OPENカーソル

用途

OPEN文は、入力パラメータとカーソルとを対応付け、カーソルをオープンします。

構文

```
EXEC SQL [FOR :array_size] OPEN dyn_cursor
    [[USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam1 | string_literal}]
    [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam2 | string_literal}]] ;
```

変数

array_size

この変数は、記述子を割り当てた時に指定した数と等しいか、それよりも小さい数に制限されます。

dyn_cursor

カーソル変数。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

カーソルに対応付けられた準備済の文にコロンまたは疑問符が含まれる場合は、USING句を指定する必要があります。指定しない場合は、ランタイム・エラーが発生します。DML RETURNING句がサポートされています。

例

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :binddes ;

EXEC SQL FOR :limit OPEN C2 USING DESCRIPTOR :b1, :b2
   INTO SQL DESCRIPTOR :seldes ;
```

関連項目

- DML RETURNING句

14.5.12 FETCH

用途

動的DECLARE文で宣言されたカーソルの行をフェッチします。

構文

```
EXEC SQL [FOR :array_size] FETCH cursor INTO [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam / string_litera/} ;
```

変数

array_size

文が処理する行数。

cursor

事前に宣言された動的カーソル。

GLOBAL | LOCAL

デフォルトでは、オプション句のスコープはLOCALに設定されています。ローカル記述子は、その記述子を割り当てたファイル内でのみアクセス可能です。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子の名前。

使用上の注意

FOR句のarray_sizeオプションは、ALLOCATE DESCRIPTOR文で指定された数以下にする必要があります。

例

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES' ;

EXEC SQL FOR :arsz FETCH C2 INTO DESCRIPTOR :desc ;
```

14.5.13 動的カーソルのCLOSE

用途

動的カーソルをクローズします。構文は、旧バージョンのOracle方法4から変更されていません。

構文

```
EXEC SQL CLOSE cursor ;
```

変数

cursor

事前に宣言された動的カーソル。

例

```
EXEC SQL CLOSE C1 ;
```

14.5.14 Oracle動的方法4との違い

ANSI動的SQLインタフェースは、Oracle動的方法4がサポートしたすべてのデータ型に加え、次の機能をサポートしています。

- ANSI動的SQLによるオブジェクト型、結果セットおよびLOB型を含むすべてのデータ型のサポート。
- ANSIモードでは、内部の*SQL*記述領域が使用されますが、これは、Oracleの以前の動的方法4で入力および出力情報の格納に使用される外部SQLDAを拡張したものです。
- ALLOCATE DESCRIPTOR、DEALLOCATE DESCRIPTOR、DESCRIBE、GET DESCRIPTORおよびSET DESCRIPTORという新しい埋込みSQL文が導入されています。
- ANSI動的SQLでは、DESCRIBE文は標識変数の名前を戻しません。
- ANSI動的SQLでは、戻される列名または式の最大サイズを指定できません。デフォルト・サイズは128に設定されます。
- 記述子名は、引用符で囲んだ識別子、または先頭にコロンをつけたホスト変数のどちらかであることが必要です。
- 出力時に、DESCRIBE文のSELECT LIST FOR句はオプション・キーワードOUTPUTに置き換えられます。INTO句はUSING DESCRIPTOR句に置き換えられます。この句には、オプション・キーワードSQLを含めることができます。
- 入力時に、DESCRIBE文のオプションのBIND VARIABLES FOR句をキーワードINPUTに置き換えることができます。INTO句はUSING DESCRIPTOR句に置き換えられます。この句には、オプション・キーワードSQLを含めることができます。
- EXECUTE、FETCHおよびOPEN文でUSING句のキーワードDESCRIPTORの前にオプション・キーワードSQLを指定できます。

14.5.15 制限事項(ANSI動的SQL)

ANSI動的SQLには次の制限が適用されます。

- 同じモジュール内でANSIおよびOracle動的SQLを組み合せて使用できません。
- プリコンパイラ・オプションDYNAMICをANSIに設定する必要があります。DYNAMICをANSIに設定したときにのみ、プリコンパイラ・オプションTYPE_CODEをANSIに設定できます。
- SET文は、項目名としてのホスト変数のみサポートします。

14.6 サンプル・プログラム

次の2つのプログラムは、demoディレクトリにあります。

14.6.1 ansidyn1.pc

このプログラムは、ANSI動的SQLを使用したSQL文の処理方法を示します。このSQL文は実行時まで不明です。これは、ANSI動的SQLを使用するための最も簡単な(ただし、最も効率的というわけではない)方法を紹介することを目的としています。ここでは、ANSIと互換性のある値セマンティクスおよびANSI型コードを使用しています。ANSI SQLSTATEは、エラー番号用に使用されています。記述子名はリテラルです。すべての入力および出力には、ANSI可変キャラクタ・タイプが使用されます。

このプログラムでは、自分のユーザー名とパスワードを使用してORACLEに接続した後、SQL文を入力します。埋込み型ではなく通常のSQL構文を使用して有効なSQLまたはPL/SQL文を入力し、各文の終わりにセミコロンを付けてください。入力した文が処理されます。問合せのときはフェッチされた行が表示されます。

複数行の文を入力できます。最大1023文字まで入力できます。変数のサイズには制限があり、MAX_VAR_LENが255に定義されています。このプログラムでは、バインド変数は40、選択リスト項目も40まで処理できます。DML RETURNING句およびユーザー定義型は、値セマンティクスではサポートされていません。

次のように、mode = ansiに設定してプログラムをプリコンパイルします。

```
proc mode=ansi ansidyn1
```

mode=ansiに指定すると、動的およびtype_codeがANSIに設定されます。

```
/*************************************************************************
ANSI Dynamic Demo 1:  ANSI Dynamic SQL with value semantics,
                                literal descriptor names
                                and ANSI type codes

This program demonstates using ANSI Dynamic SQL to process SQL
statements which are not known until runtime.  It is intended to
demonstrate the simplest (though not the most efficient) approach
to using ANSI Dynamic SQL.  It uses ANSI compatible value semantics
and ANSI type codes. ANSI Sqlstate is used for error numbers.
Descriptor names are literals. All input and output is through ANSI the
varying character type.

The program connects you to ORACLE using your username and password,
then prompts you for a SQL statement.  Enter legal SQL or PL/SQL
statements using regular, not embedded, SQL syntax and terminate each
statement with a seimcolon.  Your statement will be processed.  If it
is a query, the fetched rows are displayed.

You can enter multiline statements.  The limit is 1023 characters.
There is a limit on the size of the variables, MAX_VAR_LEN, defined as 255.
This program processes up to 40 bind variables and 40 select-list items.
DML returning statments and user defined types are not supported with
value semantics.

Precompile the program with mode=ansi, for example:

proc mode=ansi ansidyn1

Using mode=ansi will set dynamic and type_code to ansi.

*************************************************************************/

#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
```

```
#include <sqlcpr.h>

#define MAX_OCCURENCES 40
#define MAX_VAR_LEN    255
#define MAX_NAME_LEN   31

#ifndef NULL
#define NULL  0
#endif


/* Prototypes */
#if defined(__STDC__)
  void sql_error(void);
  int oracle_connect(void);
  int get_dyn_statement(void);
  int process_input(void);
  int process_output(void);
  void help(void);
#else
  void sql_error(/*_ void _*/);
  int oracle_connect(/*_ void _*/);
  int get_dyn_statement(/* void _*/);
  int process_input(/*_ void _*/);
  int process_output(/*_ void _*/);
  void help(/*_ void _*/);
#endif

EXEC SQL INCLUDE sqlca;

char SQLSTATE[6];

/* global variables */
EXEC SQL BEGIN DECLARE SECTION;
 char    dyn_statement[1024];
 char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;




/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;

/* A global flag for the error routine. */
int parse_flag = 0;
/* A global flag to indicate statement is a select */
int select_found;

void main()
{

    /* Connect to the database. */
    if (oracle_connect() != 0)
        exit(1);

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    /* Allocate the input and output descriptors. */
    EXEC SQL ALLOCATE DESCRIPTOR 'input_descriptor';
```

```
    EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor';

    /* Process SQL statements. */
    for (;;)
    {
        (void) setjmp(jmp_continue);

        /* Get the statement.  Break on "exit". */
        if (get_dyn_statement() != 0)
            break;

        /* Prepare the statement and declare a cursor. */
        parse_flag = 1;     /* Set a flag for sql_error(). */
        EXEC SQL PREPARE S FROM :dyn_statement;
        parse_flag = 0;     /* Unset the flag. */

        EXEC SQL DECLARE C CURSOR FOR S;

        /* Call the function that processes the input. */
        if (process_input())
            exit(1);

        /* Open the cursor and execute the statement. */
        EXEC SQL OPEN C USING DESCRIPTOR 'input_descriptor';

        /* Call the function that processes the output. */
        if (process_output())
            exit(1);

        /* Close the cursor. */
        EXEC SQL CLOSE C;

    }   /* end of for(;;) statement-processing loop */


    /* Deallocate the descriptors */
    EXEC SQL DEALLOCATE DESCRIPTOR 'input_descriptor';
    EXEC SQL DEALLOCATE DESCRIPTOR 'output_descriptor';

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK;
    puts("¥nHave a good day!¥n");

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    return;
}



int get_dyn_statement()
{
    char *cp, linebuf[256];
    int iter, plsql;

    for (plsql = 0, iter = 1; ;)
    {
        if (iter == 1)
        {
            printf("¥nSQL> ");
            dyn_statement[0] = '¥0';
```

```
            select_found = 0;
        }

        fgets(linebuf, sizeof linebuf, stdin);

        cp = strrchr(linebuf, '¥n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
            continue;

        if ((strncmp(linebuf, "SELECT", 6) == 0) ||
            (strncmp(linebuf, "select", 6) == 0))
        {
            select_found=1;;
        }

        if ((strncmp(linebuf, "EXIT", 4) == 0) ||
            (strncmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        }

        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn_statement, ';'))))
        {
            *cp = '¥0';
            break;
        }
        else
        {
            iter++;
            printf("%3d  ", iter);
        }
    }
    return 0;
}


int process_input()
{
    int i;
    EXEC SQL BEGIN DECLARE SECTION;
```

```
      char name[31];
      int  input_count, input_len, occurs, ANSI_varchar_type;
      char input_buf[MAX_VAR_LEN];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR 'input_descriptor';
    EXEC SQL GET DESCRIPTOR 'input_descriptor' :input_count = COUNT;

    ANSI_varchar_type=12;
    for (i=0; i < input_count; i++)
    {
        occurs = i +1;                        /* occurence is 1 based */
        EXEC SQL GET DESCRIPTOR 'input_descriptor'
                 VALUE :occurs :name = NAME;
        printf ("¥nEnter value for input variable %*.*s:  ", 10,31, name);
        fgets(input_buf, sizeof(input_buf), stdin);
        input_len = strlen(input_buf) - 1;  /* get rid of new line */
        input_buf[input_len] = '¥0';        /* null terminate */
        EXEC SQL SET DESCRIPTOR 'input_descriptor'
                 VALUE :occurs TYPE = :ANSI_varchar_type,
                               LENGTH = :input_len,
                               DATA = :input_buf;
    }
    return(sqlca.sqlcode);
}


int process_output()
{
   int i, j;
   EXEC SQL BEGIN DECLARE SECTION;
     int output_count, occurs, type, len, col_len;
     short indi;
     char data[MAX_VAR_LEN], name[MAX_NAME_LEN];
   EXEC SQL END DECLARE SECTION;
   if (!select_found)
       return(0);

   EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR 'output_descriptor';

   EXEC SQL GET DESCRIPTOR 'output_descriptor' :output_count = COUNT;


   printf ("¥n");
   type = 12;             /* ANSI VARYING character type */
   len = MAX_VAR_LEN;     /* use the max allocated length */
   for (i = 0; i < output_count; i++)
    {
        occurs = i + 1;
        EXEC SQL GET DESCRIPTOR 'output_descriptor' VALUE :occurs
                 :name = NAME;
        printf("%-*.*s ", 9,9, name);
        EXEC SQL SET DESCRIPTOR 'output_descriptor' VALUE :occurs
                 TYPE = :type, LENGTH = :len;
    }
    printf("¥n");

    /* FETCH each row selected and print the column values. */
    EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;
```

```
    for (;;)
    {
        EXEC SQL FETCH C INTO DESCRIPTOR 'output_descriptor';
        for (i=0; i < output_count; i++)
          {
            occurs = i + 1;
            EXEC SQL GET DESCRIPTOR 'output_descriptor' VALUE :occurs
                 :data = DATA, :indi = INDICATOR;
            if (indi == -1)
              printf("%-*.*s ", 9,9, "NULL");
            else
              printf("%-*.*s ", 9,9, data);  /* simplified output formatting */
                          /* truncation will occur, but columns will line up */
          }
         printf ("¥n");
    }
end_select_loop:
    return(0);
}



void help()
{
    puts("¥n¥nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
    puts("Statements can be continued over several lines, except");
    puts("within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
    puts("Terminate a PL/SQL block (which can contain embedded semicolons)");
    puts("with a slash (/).");
    puts("Typing ¥"exit¥" (no semicolon needed) exits the program.");
    puts("You typed ¥"?¥" or ¥"help¥" to get this message.¥n¥n");
}


void sql_error()
{
    /* ORACLE error handler */
    printf("¥n¥nANSI sqlstate: %s: ", SQLSTATE);
    printf ("¥n¥n%.70s¥n",sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
        printf
        ("Parse error at character offset %d in SQL statement.¥n",
           sqlca.sqlerrd[4]);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp_continue, 1);
}


int oracle_connect()
{
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR  username[128];
        VARCHAR  password[32];
    EXEC SQL END DECLARE SECTION;

    printf("¥nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
```

```
    username.arr[strlen((char *) username.arr)-1] = '¥0';
    username.len = (unsigned short)strlen((char *) username.arr);

    printf("password: ");
    fgets((char *) password.arr, sizeof password.arr, stdin);
    password.arr[strlen((char *) password.arr) - 1] = '¥0';
    password.len = (unsigned short)strlen((char *) password.arr);


    EXEC SQL WHENEVER SQLERROR GOTO connect_error;

    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    printf("¥nConnected to ORACLE as user %s.¥n", username.arr);

    return 0;

connect_error:
    fprintf(stderr, "Cannot connect to ORACLE as user %s¥n", username.arr);
    return -1;
}
```

14.6.2 ansidyn2.pc

このプログラムは、ANSI動的SQLを使用したSQL文の処理方法を示します。このSQL文は実行時まで不明です。バッチ処理および参照セマンティクスのOracle拡張機能が使用されます。

このプログラムでは、自分のユーザー名とパスワードを使用してORACLEに接続した後、SQL文を入力します。埋込み型ではなく対話型のSQL構文を使用して有効なSQLまたはPL/SQL文を入力し、各文の終わりにセミコロンを付けてください。入力した文が処理されます。問合せのときはフェッチされた行が表示されます。

複数行の文を入力できます。最大1023文字まで入力できます。変数のサイズには制限があり、MAX_VAR_LENが255に定義されています。このプログラムでは、バインド変数は40、選択リスト項目も40まで処理できます。

次のように、dynamic = ansiに設定してプログラムをプリコンパイルします。

```
proc dynamic=ansi ansidyn2

/*************************************************************************
ANSI Dynamic Demo 2:  ANSI Dynamic SQL with reference semantics,
                          batch processing and global descriptor
                          names in host variables

This program demonstates using ANSI Dynamic SQL to process SQL
statements which are not known until runtime.  It uses the Oracle
extensions for batch processing and reference semantics.

The program connects you to ORACLE using your username and password,
then prompts you for a SQL statement.  Enter legal SQL or PL/SQL
statement using interactive, not embedded, SQL syntax, terminating the
statement with a seimcolon.  Your statement will be processed.  If it
is a query, the fetched rows are displayed.

If your statement has input bind variables (other than in a where clause),
the program will ask for an input array size and then allow you to enter
that number of input values. If your statment has output, the program will
ask you for an output array size and will do array fetchng using that value.
It will also output the rows fetched in one batch together, so using a small
value for the output array size will improve the look of the output.
```

```
For example, connected as scott/tiger, try select empno, ename from emp
with an output array size of 4;

You can enter multiline statements.  The limit is 1023 characters.
There is a limit on the size of the variables, MAX_VAR_LEN, defined as 255.
This program processes up to 40 bind variables and 40 select-list items.

Precompile with program with dynamic=ansi, for example:

proc dynamic=ansi ansidyn2

**********************************************************************/

#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
#include <sqlcpr.h>


#define MAX_OCCURENCES  40
#define MAX_ARRSZ      100
#define MAX_VAR_LEN    255
#define MAX_NAME_LEN    31

#ifndef NULL
#define NULL  0
#endif


/* Prototypes */
#if defined(__STDC__)
  void sql_error(void);
  int oracle_connect(void);
  int get_dyn_statement(void);
  int process_input(void);
  int process_output(void);
  void rows_processed(void);
  void help(void);
#else
  void sql_error(/*_ void _*/);
  int oracle_connect(/*_ void _*/);
  int get_dyn_statement(/* void _*/);
  int process_input(/*_ void _*/);
  int process_output(/*_ void _*/);
  void rows_processed(/*_ void _*/);
  void help(/*_ void _*/);
#endif

EXEC SQL INCLUDE sqlca;

/* global variables */
char    dyn_statement[1024];                    /* statement variable     */
EXEC SQL VAR dyn_statement IS STRING(1024);

char  indesc[]="input_descriptor";              /* descriptor names       */
char outdesc[]="output_descriptor";
char   input[MAX_OCCURENCES][MAX_ARRSZ][MAX_VAR_LEN +1 ],    /* data areas */
      output[MAX_OCCURENCES][MAX_ARRSZ][MAX_VAR_LEN + 1];
```

```
short outindi[MAX_OCCURENCES][MAX_ARRSZ];        /* output indicators      */
short *iptr;

int   in_array_size;     /* size of input batch, that is, number of rows */
int   out_array_size;    /* size of input batch, that is, number of rows */
int   max_array_size=MAX_ARRSZ;   /* maximum arrays size used for allocates */

char *dml_commands[] = {"SELECT", "select", "INSERT", "insert",
                        "UPDATE", "update", "DELETE", "delete"};

int select_found, cursor_open = 0;

/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;

/* A global flag for the error routine. */
int parse_flag = 0;

void main()
{

    /* Connect to the database. */
    if (oracle_connect() != 0)
        exit(1);

    EXEC SQL WHENEVER SQLERROR DO sql_error();

    /* Allocate the input and output descriptors. */
    EXEC SQL FOR :max_array_size
             ALLOCATE DESCRIPTOR GLOBAL :indesc;
    EXEC SQL FOR :max_array_size
             ALLOCATE DESCRIPTOR GLOBAL :outdesc;

    /* Process SQL statements. */
    for (;;)
    {
        (void) setjmp(jmp_continue);

        /* Get the statement.  Break on "exit". */
        if (get_dyn_statement() != 0)
            break;

        /* Prepare the statement and declare a cursor. */
        parse_flag = 1;     /* Set a flag for sql_error(). */
        EXEC SQL PREPARE S FROM :dyn_statement;
        parse_flag = 0;     /* Unset the flag. */

        EXEC SQL DECLARE C CURSOR FOR S;

        /* Call the function that processes the input. */
        if (process_input())
            exit(1);

        /* Open the cursor and execute the statement. */
        EXEC SQL FOR :in_array_size
            OPEN C USING DESCRIPTOR GLOBAL :indesc;
        cursor_open = 1;

        /* Call the function that processes the output. */
        if (process_output())
```

```
            exit(1);

        /* Tell user how many rows were processed. */
        rows_processed();

    }   /* end of for(;;) statement-processing loop */


    /* Close the cursor. */
    if (cursor_open)
      EXEC SQL CLOSE C;

    /* Deallocate the descriptors */
    EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL :indesc;
    EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL :outdesc;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;
    puts("¥nHave a good day!¥n");

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    return;
}



int get_dyn_statement()
{
    char *cp, linebuf[256];
    int iter, plsql;

    for (plsql = 0, iter = 1; ;)
    {
        if (iter == 1)
        {
            printf("¥nSQL> ");
            dyn_statement[0] = '¥0';
            select_found = 0;
        }

        fgets(linebuf, sizeof linebuf, stdin);

        cp = strrchr(linebuf, '¥n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
            continue;

        if ((strncmp(linebuf, "SELECT", 6) == 0) ||
            (strncmp(linebuf, "select", 6) == 0))
        {
            select_found=1;;
        }

        if ((strncmp(linebuf, "EXIT", 4) == 0) ||
            (strncmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        }
```

```
        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn_statement, ';'))))
        {
            *cp = '¥0';
            break;
        }
        else
        {
            iter++;
            printf("%3d  ", iter);
        }
    }
    return 0;
}


int process_input()
{
    int i, j;
    char name[31];
    int  input_count, input_len= MAX_VAR_LEN;
    int  occurs, string_type = 5;
    int  string_len;
    char arr_size[3];

    EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR GLOBAL :indesc;
    EXEC SQL GET DESCRIPTOR GLOBAL :indesc :input_count = COUNT;

    if (input_count > 0 && !select_found )
       {     /* get input array size */
          printf ("¥nEnter value for input array size (max is %d) :  ",
                           max_array_size);
        fgets(arr_size, 4, stdin);
        in_array_size = atoi(arr_size);
       }
    else
       {
         in_array_size = 1;
       }
    for (i=0; i < input_count; i++)
    {
        occurs = i +1;                          /* occurence is 1 based */
        EXEC SQL GET DESCRIPTOR GLOBAL :indesc
```

```
                VALUE :occurs :name = NAME;

        for (j=0; j < in_array_size; j++)
        {
          if (in_array_size == 1)
            printf ("¥nEnter value for input variable %*.*s:  ",10,31, name);
          else
            printf ("¥nEnter %d%s value for input variable %*.*s:  ",
               j +1, ((j==0) ?  "st" :  (j==1) ? "nd" : (j==2) ? "rd" :"th"),
                     10,31, name);
          fgets(input[i][j], sizeof(input[i][j]), stdin);
          string_len = strlen(input[i][j]);
          input[i][j][string_len - 1 ] = '¥0';   /* change ¥n to ¥0 */
        }
        EXEC SQL SET DESCRIPTOR GLOBAL :indesc
                 VALUE :occurs TYPE = :string_type, LENGTH = :input_len;
        EXEC SQL FOR :in_array_size
                 SET DESCRIPTOR GLOBAL :indesc
                     VALUE :occurs  REF DATA = :input[i];
    }

    return(sqlca.sqlcode);
}


int process_output()
{
   int i, j;
   int output_count, occurs;
   int type, output_len= MAX_VAR_LEN;
   char name[MAX_OCCURENCES][MAX_NAME_LEN];
   int rows_this_fetch=0, cumulative_rows=0;
   char arr_size[3];
   if (!select_found)
      return(0);
   EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR GLOBAL :outdesc;

   EXEC SQL GET DESCRIPTOR GLOBAL :outdesc :output_count = COUNT;

   if (output_count > 0 )
      {
        printf ("¥nEnter value for output array size (max is %d) :  ",
                        max_array_size);
        fgets(arr_size, 4, stdin);
        out_array_size = atoi(arr_size);
      }
   if (out_array_size < 1)    /* must have at least one */
       out_array_size = 1;

   printf ("¥n");

   for (i = 0; i < output_count; i++)
   {
      occurs = i + 1;
      EXEC SQL GET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
               :type = TYPE, :name[i] = NAME;
      occurs = i + 1;                          /* occurence is one based */
      type = 5;  /* force all data to be null terminated character */
      EXEC SQL SET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
               TYPE = :type, LENGTH = :output_len;
```

```
      iptr = (short *)&outindi[i]; /* no mult-dimension non-char host vars */
      EXEC SQL FOR :out_array_size
               SET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
               REF DATA = :output[i], REF INDICATOR = :iptr;
   }



   EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;

   /* print the column headings */
   for (j=0; j < out_array_size; j++)
      for (i=0; i < output_count; i++)
         printf("%-*.*s ", 9,9, name[i]);
   printf("¥n");

   /* FETCH each row selected and print the column values. */
   for (;;)
   {
      EXEC SQL FOR :out_array_size
              FETCH C INTO DESCRIPTOR GLOBAL :outdesc;
      rows_this_fetch = sqlca.sqlerrd[2] - cumulative_rows;
      cumulative_rows = sqlca.sqlerrd[2];
      if (rows_this_fetch)
      for (j=0; j < out_array_size && j < rows_this_fetch; j++)
      {           /* output by columns using simplified formatting */
         for (i=0; i < output_count; i++)
           {
                if (outindi[i][j] == -1)
                   printf("%-*.*s ", 9, 9, "NULL");
               else
                  printf("%-*.*s ", 9, 9, output[i][j]);  /* simplified */
                              /* output formatting may cause truncation */
                              /* but columns will line up */
           }
      }
       printf ("¥n");
   }

end_select_loop:
   /* print any unprinted rows */
   rows_this_fetch = sqlca.sqlerrd[2] - cumulative_rows;
   cumulative_rows = sqlca.sqlerrd[2];
   if (rows_this_fetch)
     for (j=0; j < out_array_size && j < rows_this_fetch; j++)
       {           /* output by columns using simplified formatting */
         for (i=0; i < output_count; i++)
           {
              if (outindi[i][j] == -1)
                   printf("%-*.*s ",9, 9, "NULL");
               else
                  printf("%-*.*s ", 9, 9, output[i][j]);
           }
        }
   return(0);
}

void rows_processed()
{
```

```
   int i;
   for (i = 0; i < 8; i++)
     {
       if (strncmp(dyn_statement, dml_commands[i], 6) == 0)
         {
            printf("¥n¥n%d row%c processed.¥n", sqlca.sqlerrd[2],
                       sqlca.sqlerrd[2] == 1 ? ' ' : 's');
            break;
         }
     }
   return;
}


void help()
{
    puts("¥n¥nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
    puts("Statements can be continued over several lines, except");
    puts("within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
    puts("Terminate a PL/SQL block (which can contain embedded semicolons)");
    puts("with a slash (/).");
    puts("Typing ¥"exit¥" (no semicolon needed) exits the program.");
    puts("You typed ¥"?¥" or ¥"help¥" to get this message.¥n¥n");
}


void sql_error()
{
    /* ORACLE error handler */
    printf ("¥n¥n%.70s¥n",sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
        printf
        ("Parse error at character offset %d in SQL statement.¥n",
           sqlca.sqlerrd[4]);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp_continue, 1);
}


int oracle_connect()
{
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR  username[128];
        VARCHAR  password[32];
    EXEC SQL END DECLARE SECTION;

    printf("¥nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
    username.arr[strlen((char *) username.arr)-1] = '¥0';
    username.len = (unsigned short)strlen((char *) username.arr);

    printf("password: ");
    fgets((char *) password.arr, sizeof password.arr, stdin);
    password.arr[strlen((char *) password.arr) - 1] = '¥0';
    password.len = (unsigned short)strlen((char *) password.arr);
```

```
    EXEC SQL WHENEVER SQLERROR GOTO connect_error;

    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    printf("¥nConnected to ORACLE as user %s.¥n", username.arr);

    return 0;

connect_error:
    fprintf(stderr, "Cannot connect to ORACLE as user %s¥n", username.arr);
    return -1;
}
```

15 Oracle動的SQL: 方法4

この章ではOracle動的SQL方法4を実装する方法を説明します。この方法では、ホスト変数を含む動的SQL文を受け取ったり、作成できます。含まれるホスト変数の個数は様々です。既存のアプリケーションは、この方法を使用してサポートしてください。新しいアプリケーションでは、すべてANSI動的SQL方法4を使用してください。

Oracle動的SQL方法4は、オブジェクト型、カーソル変数、構造体の配列、DML RETURNING句、Unicode変数およびLOBをサポートしていません。かわりにANSI動的SQL方法4を使用してください。この章のトピックは、次のとおりです:

- 方法4の特殊要件
- SQLDAの説明
- SQLDA変数の使用について
- 予備知識
- 基本ステップ
- 各ステップの詳細
- サンプル・プログラム: 動的SQL方法4
- サンプル・プログラム: スクロール可能カーソルを使用する動的SQL方法4

関連項目:

- 動的SQL方法1、2、3の詳細および方法4の概要は、Oracle動的SQLを参照してください。
- ANSI動的SQL

15.1 方法4の特殊要件

方法4の必要条件を学習する前に、選択リスト項目とプレースホルダという用語を理解する必要があります。選択リスト項目とは、問合せ内でキーワードSELECTの後に続く列または式のことです。たとえば、次の動的問合せは3つの選択リスト項目を含んでいます。

```
SELECT ename, job, sal + comm FROM emp WHERE deptno = 20
```

プレースホルダとは、SQL文の中で実際のバインド変数用に場所を確保する、ダミーのバインド変数のことです。宣言する必要はなく、任意の名前を付けられます。

バインド変数のプレースホルダはSET句、VALUES句およびWHERE句で最もよく使用されます。たとえば、次の動的SQL文はそれぞれ2つのプレースホルダを含んでいます。

```
INSERT INTO emp (empno, deptno) VALUES (:e, :d)
DELETE FROM dept WHERE deptno = :num OR loc = :loc
```

15.1.1 方法4が特別な理由

方法1、2、3とは異なり、動的SQL方法4ではプログラムで次のことができます。

- 選択リスト項目数とプレースホルダ数が不明な動的SQL文の受取りまたは作成
- OracleとC言語の間のデータ型変換の明示的な制御

プログラムにこのような柔軟性を持たせるには、Oracleランタイム・ライブラリに補足情報を追加する必要があります。

15.1.2 Oracleに必要な情報

Pro*C/C++プリコンパイラでは、すべての実行可能な動的SQL文についてOracleコールが生成されます。動的SQL文に選択リスト項目またはプレースホルダが指定されていない場合、Oracleにはその文を実行するための補足情報は必要ありません。次のDELETE文がこのカテゴリに該当します。

```
DELETE FROM emp WHERE deptno = 30
```

ただし、ほとんどの動的SQL文には、次のUPDATE文のように、選択リスト項目、またはバインド変数のプレースホルダが含まれています。

UPDATE文:

```
UPDATE emp SET comm = :c WHERE empno = :e
```

バインド変数のためのプレースホルダまたは選択リスト項目を含む動的SQL文を実行するには、入力(バインド)値および問合せを実行するときにFETCHされた値を保持するプログラム変数についての情報が必要です。Oracleは次の情報を必要とします。

- バインド変数の数と選択リスト項目の数
- 各バインド変数と選択リスト項目の長さ
- 各バインド変数と選択リスト項目のデータ型
- 各バインド変数のアドレスと、各選択リスト項目の値が設定される出力変数のアドレス

15.1.3 情報の格納位置

選択リスト項目またはバインド変数のプレースホルダについてOracleで必要な情報は、その値以外はすべて、SQL記述子領域(SQLDA)というプログラム・データ構造体に格納されます。SQLDA構造体はsqlda.hヘッダー・ファイルに定義されています。

選択リスト項目の記述は選択記述子に格納されます。また、バインド変数に対するプレースホルダの記述はバインド記述子に格納されます。

選択リスト項目の値は出力変数に格納され、バインド変数の値は入力変数に格納されます。これらの変数のアドレスを選択SQLDAまたはバインドSQLDAに格納すると、出力値を書き込む位置および入力値を読み込む位置がOracleに認識されます。

値はどのようにしてこれらのデータ変数に格納されるのでしょうか。出力値は、カーソルを使用してFETCHされます。入力値は、通常はユーザーが対話形式で入力した情報をもとに、プログラムによって代入されます。

15.1.4 SQLDAの参照方法

バインド記述子および選択記述子は、通常はポインタによって参照されます。動的SQLプログラムでは、少なくとも1つのバインド記述子と1つの選択記述子のポインタを次のように宣言する必要があります。

```
#include <sqlda.h>
...
SQLDA *bind_dp;
SQLDA *select_dp;
```

この後にSQLSQLDAAlloc()関数を使用すると、次のように記述子を割り当てることができます。

```
bind_dp = SQLSQLDAAlloc(runtime_context, size, name_length, ind_name_length);
```

Oracle8以前のバージョンでは、SQLSQLDAAlloc()はsqlaldt()に相当します。

定数SQL_SINGLE_RCTX は、(dvoid*)0として定義されます。これは、アプリケーションがシングル・スレッドの場合に、*runtime_context*に使用します。

関連項目:

- これおよび他のSQLLIB関数の詳細は、表15-3を参照してください
- SQLSQLDAAlloc()およびそのパラメータの詳細は、SQLDAの割当てを参照してください。

15.1.5 情報の取得方法

DESCRIBE文を使用すると、Oracleに必要な情報が得られます。

DESCRIBE SELECT LIST文は、各選択リスト項目を検査して名前とその長さを判断します。次にこの情報を使用できるように選択SQLDAに格納します。たとえば、格納された情報は、後で印刷出力の列見出しとして選択リスト名を使用するときなどに使用できます。選択リスト項目の合計数も、DESCRIBE文によりSQLDAに格納されます。

DESCRIBE BIND VARIABLES文は、各プレースホルダを調べて、その名前および長さを確認した後、それらの情報を入力バッファおよびバインドSQLDAに格納します。格納された情報は、後でプレースホルダ名を使用してバインド変数の値の入力をユーザーに求めるときなどに使用できます。

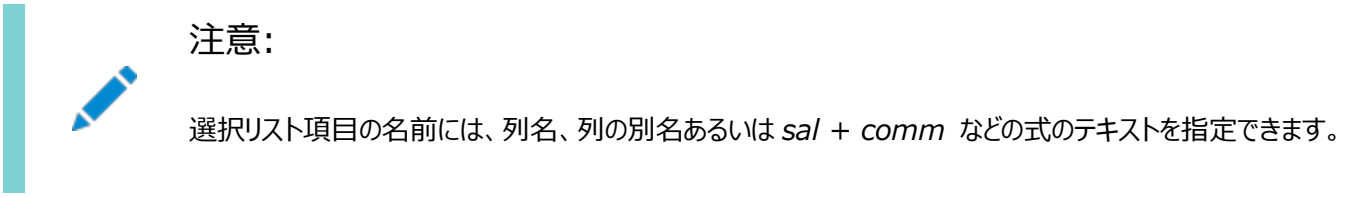
15.2 SQLDAの説明

この項ではSQLDAのデータ構造を詳しく説明します。SQLDAの宣言方法、格納されている変数、初期化の方法、プログラム内での使用方法を理解できます。

15.2.1 SQLDAの目的

選択リスト項目の数またはバインド変数のプレースホルダの数が不明の動的SQL文には、方法4を使用する必要があります。このような動的SQL文を処理するには、プログラムでSQLDA (記述子とも呼ばれます)を明示的に宣言する必要があります。記述子はそれぞれ**構造体**になっています。記述子はプログラムにコピーまたはハードコードする必要があります。

選択記述子には、選択リスト項目の記述、選択リスト項目の名前および値が格納されている出力バッファのアドレスが保持されます。



バインド記述子には、バインド変数と標識変数の記述、およびバインド変数と標識変数の名前と値が格納されている入力バッファのアドレスが保持されます。

15.2.2 複数のSQLDA

プログラムにアクティブな動的SQL文が2つ以上ある場合は、それぞれの文が専用のSQLDAを持つ必要があります。別の名前で任意の数のSQLDAを宣言できます。たとえば、*sel_desc1*、*sel_desc2*および*sel_desc3*という名前を付けて3つの選択SQLDAを宣言すると、同時にOPENされている3つのカーソルからFETCHできます。ただし、非並行のカーソルではSQLDAを再利用できます。

15.2.3 SQLDAの宣言

SQLDAを宣言するには、sqlda.hヘッダー・ファイルをインクルードします。SQLDAの内容は、次のとおりです。

```
struct SQLDA
{
    long    N;          /* Descriptor size in number of entries */
    char  **V;        Ptr to Arr of addresses of main variables */
    int   *L;             /* Ptr to Arr of lengths of buffers */
    short  *T;               /* Ptr to Arr of types of buffers */
    short **I;      * Ptr to Arr of addresses of indicator vars */
    long    F;         /* Number of variables found by DESCRIBE */
    char  **S;          /* Ptr to Arr of variable name pointers */
    short  *M;       /* Ptr to Arr of max lengths of var. names */
    short  *C;    * Ptr to Arr of current lengths of var. names */
    char  **X;         /* Ptr to Arr of ind. var. name pointers */
    short  *Y;  /* Ptr to Arr of max lengths of ind. var. names */
    short  *Z;  /* Ptr to Arr of cur lengths of ind. var. names */
};
```

15.2.4 SQLDAの割当て

SQLDAの宣言後に、次の構文のライブラリ関数SQLSQLDAAlloc()(Oracle8以前のバージョンではsqlaldt()に相当)を使用して、記憶域を割り当てます。

```
descriptor_name = SQLSQLDAAlloc (runtime_context, max_vars, max_name, max_ind_name);
```

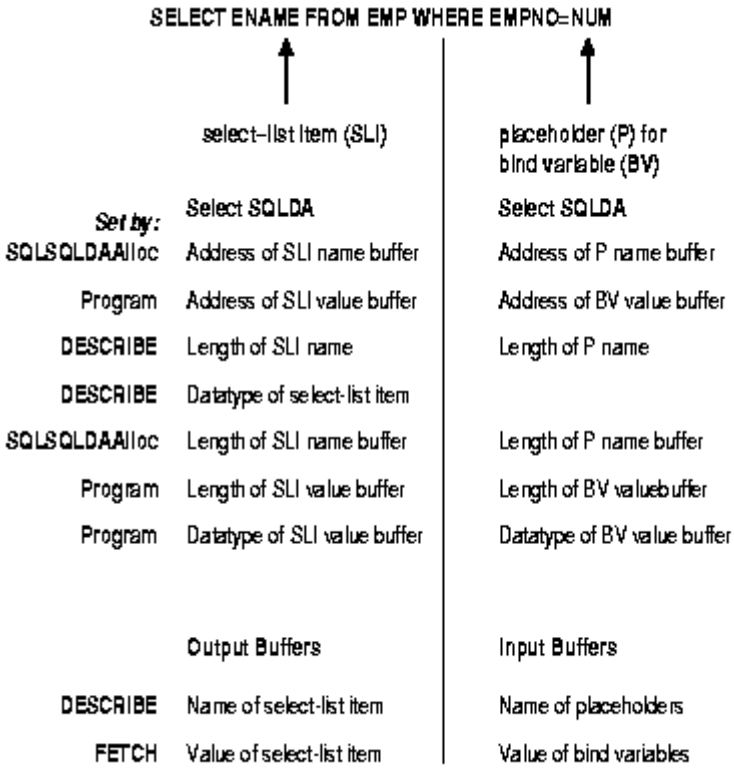
各パラメータの意味は次のとおりです。

構文	説明
runtime_context	ランタイム・コンテキストへのポインタ
max_vars	記述子が記述できる選択リスト項目またはプレースホルダの最大数。
max_name	選択リスト名またはプレースホルダ名の最大長。
max_ind_name	オプション指定でプレースホルダ名に付加される標識変数名の最大長。このパラメータはバインド記述子専用です。したがって、選択記述子を割り当てるときは、このパラメータを0(ゼロ)に設定します。

記述子の他に、SQLSQLDAAlloc()は記述子変数が指すデータ・バッファも割り当てます。

図15-1に、変数がSQLSQLDAAlloc()コール、DESCRIBEコマンド、FETCHコマンドまたはプログラム割当てのうち、どの方法で設定されるかを示します。

図15-1 変数の設定方法



関連項目

- SQLDA変数の使用について
- 記述子用の記憶域の割当て

15.3 SQLDA変数の使用について

この項では、SQLDA内の各変数の用途および使用方法を説明します。

15.3.1 *N*変数

Nは、DESCRIBE可能な選択リスト項目またはプレースホルダの最大数を指定します。つまり、*N*によって記述子配列に含まれる要素の数が決まります。

オプションのDESCRIBEコマンドを発行する前に、ライブラリ関数SQLSQLDAAlloc()を使用して*N*を記述子配列の次元に設定する必要があります。また、DESCRIBEの実行後に、DESCRIBEされた変数の実際の数に*N*を再設定する必要があります。この数は*F*変数に格納されています。

15.3.2 *V*変数

*V*は、選択リストまたはバインド変数の値を格納するデータ・バッファのアドレスからなる配列のポインタです。

記述子を割り当てると、SQLSQLDAAlloc()によってアドレスの配列にあるV[0]からV[N - 1]の要素が0(ゼロ)に設定されます。

選択記述子の場合は、FETCHコマンドを発行する前にデータ・バッファを割り当てて、この配列を設定する必要があります。文

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

この文は、FETCHされた選択リストの値を、V[0]からV[N - 1]が指しているデータ・バッファに格納するようにOracleに指示します。Oracleにより、*i*番目の選択リストの値が、V[i]が指しているデータ・バッファに格納されます。

バインド記述子の場合は、OPENコマンドを発行する前に、この配列を設定する必要があります。文

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

この文では、V[0]からV[N - 1]が指しているバインド変数の値を使用して動的SQL文を実行するようにOracleに指示します。Oracleでは、V[*i*]が指しているデータ・バッファで*i*番目のバインド変数の値が参照されます。

15.3.3 *L*変数

*L*は、データ・バッファに格納されている選択リストまたはバインド変数の値の長さからなる配列のポインタです。

選択記述子の場合、この配列は、DESCRIBE SELECT LISTによって、各選択リスト項目に予想される最大値に設定されます。しかし、FETCHコマンドを発行する前に長さを再設定する場合も考えられます。FETCHでは最大で*n*文字が戻されます(*n*は、FETCHを実行する前のL[*i*]の値です)。

長さの形式はOracleデータ型によって異なります。CHARまたはVARCHAR2の選択リスト項目については、DESCRIBE SELECT LISTはL[*i*]を選択リスト項目の最大長に設定します。NUMBER型の選択リスト項目については、位取りおよび精度が変数の下位バイトおよびその次の上位側バイトにそれぞれ戻されます。精度と位取りの値をL[*i*]から抽出するには、ライブラリ関数SQLNumberPrecV6()を使用できます。

FETCHする前に、L[*i*]を必要なデータ・バッファの長さに再設定する必要があります。たとえば、NUMBERをC言語の**char**文字列に強制変換する場合は、L[*i*]を精度の数値に符号と小数点のための2を加えたものに設定します。NUMBER型をC言語の**float**型に強制変換する場合は、L[*i*]をシステム上の**float**型の長さに設定します。

バインド記述子の場合は、OPENコマンドを発行する前に、配列の長さを設定する必要があります。たとえば、strlen()を使用してユーザーが入力したバインド変数文字列の長さを取得してから、適切な配列要素を設定します。

Oracleは、V[*i*]に格納されているアドレスを使用して間接的にデータ・バッファにアクセスするため、データ・バッファ内の値の長さは認識しません。*i*番目の選択リスト値またはバインド変数値に対してOracleが使用する長さを変更する場合は、L[*i*]を必要な長さに再設定してください。入力バッファまたは出力バッファにはそれぞれ異なる長さを指定できます。

関連項目

- 精度と位取りの抽出

15.3.4 *T*変数

*T*は、選択リストまたはバインド変数の値のデータ型コードからなる配列のポインタです。これらのデータ型コードは、*V*配列の要素が指示するデータ・バッファにOracleデータが格納されるときのデータの変換方法を決定します。

選択記述子の場合、DESCRIBE SELECT LISTはデータ型コードの配列を選択リスト内の項目の内部データ型(CHAR、NUMBER、DATEなど)に設定します。

Oracleデータ型の内部形式は処理が複雑なため、FETCHする前にデータ型をいくつか再設定する必要がある場合があります。表示用データのときは、一般には選択リストの値のデータ型をVARCHAR2またはSTRINGに強制変換することをお薦めします。計算用データのときは、Oracleの数値をC言語の形式に強制変換する必要がある場合があります。

T[*i*]の上位ビットの設定は、*i*番目の選択リスト項目のNULL/NOT NULLステータスを示します。OPENコマンドまたはFETCHコマンドを発行する前に、常にこのビットを消去する必要があります。データ型コードを取り出してNULL/NOT NULLビットを消去するには、ライブラリ関数SQLColumnNullCheck()を使用します。

OracleのNUMBER内部データ型は、V[*i*]が指すC言語のデータ・バッファと互換性のある外部データ型に変更する必要があります。

バインド記述子の場合、データ型コードの配列はDESCRIBE BIND VARIABLESによって0に設定されます。OPENコマンドを発行する前に、各要素に格納されたデータ型を設定する必要があります。コードは、V[*i*]が指すデータ・バッファの外部(C)

データ型を表します。バインド変数の値が文字列に格納され、データ型配列の要素が1(VARCHAR2データ型コード)に設定されることがよくあります。データ型コード5(STRING)を使用することもできます。

*i*番目の選択リストまたはバインド変数の値のデータ型を変更するには、T[*i*]を変更するデータ型に再設定してください。

関連項目

- データの変換
- データ型の強制変換
- NULLまたはNOT NULLデータ型の処理

15.3.5 *I*変数

*I*は、標識変数値を格納するデータ・バッファのアドレスの配列へのポインタです。

アドレスの配列のI[0]からI[N - 1]の要素を設定する必要があります。

選択記述子の場合は、FETCHコマンドを発行する前に、アドレスの配列を設定する必要があります。次の文を実行するとします。

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

この場合、戻された*i*番目の選択リスト値がNULLの場合は、I[*i*]が指す標識変数値が-1に設定されます。それ以外の場合は、(値がNULLでない) 0または(値が切り捨てられている)正の整数に設定されます。

バインド記述子の場合は、OPENコマンドを発行する前に、アドレスの配列とそれに対応付けられた標識変数を設定する必要があります。次の文を実行するとします。

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

この場合、I[*i*]が指しているデータ・バッファは*i*番目のバインド変数の値がNULLであるかどうかを決定します。標識変数の値が-1のとき、関連するバインド変数の値はNULLです。

15.3.6 *F*変数

*F*は、DESCRIBEによって検出される選択リスト項目またはプレースホルダの実際の個数です。

*F*はDESCRIBEによって設定されます。*F*の値が負のときは、割り当てられた記述子のサイズに対してDESCRIBEが検出した選択リスト項目またはプレースホルダが多すぎることを示しています。たとえば、*N*を10に設定したときに、DESCRIBEで11個の選択リスト項目またはプレースホルダが検出されると、*F*は-11に設定されます。この機能を使用すると、必要に応じて選択リスト項目またはプレースホルダに大きい記憶域を動的に再割当てすることができます。

15.3.7 *S*変数

*S*は、データ・バッファのアドレスの配列へのポインタです。動的SQL文で選択リストまたはプレースホルダの名前が見つかると、それをそのデータ・バッファに格納します。

SQLSQLDAAlloc()を使用してデータ・バッファを割り当ててから、*S*配列にそれらのアドレスを格納します。

DESCRIBEは、S[*i*]が指すデータ・バッファ内に*i*番目の選択リスト項目またはプレースホルダの名前を格納するように、Oracleに指示します。

15.3.8 *M*変数

*M*は、選択リストまたはプレースホルダの名前を格納するデータ・バッファの最大長からなる配列へのポインタです。このデータ・バッ

ファのアドレスは、S配列の要素によって指定されます。

記述子を割り当てると、SQLSQLDAAlloc()は要素M[0]からM[N - 1]までを最大長の配列に設定します。S[i]が指しているデータ・バッファに格納するとき、i番目の名前は必要に応じてM[i]の長さに切り捨てられます。

15.3.9 C変数

Cは、選択リストまたはプレースホルダの名前の現行の長さからなる配列へのポインタです。

DESCRIBEは、現行の長さの配列の中にC[0]からC[N - 1]の要素を設定します。DESCRIBE文を実行すると、配列には選択リスト名またはプレースホルダ名の文字数が格納されます。

15.3.10 X変数

Xは、標識変数の名前を格納するデータ・バッファのアドレスからなる配列へのポインタです。標識変数の値は選択リスト項目およびバインド変数に対応付けできます。ただし、標識変数の名前はバインド変数にのみ対応付けできます。したがって、Xはバインド記述子専用です。

データ・バッファを割り当て、そのアドレスをX配列に保存するには、SQLSQLDAAlloc()を使用します。

DESCRIBE BIND VARIABLESは、i番目の標識変数名をX[i]が指すデータ・バッファに格納するようにOracleに指示します。

15.3.11 Y変数

Yは、標識変数の名前を格納するデータ・バッファの最大長からなる配列へのポインタです。Xと同様に、Yもバインド記述子専用です。

SQLSQLDAAlloc()を使用して要素Y[0]からY[N - 1]までを最大長の配列に割り当てます。X[i]が指しているデータ・バッファに格納するとき、i番目の名前は必要に応じてY[i]の長さに切り捨てられます。

15.3.12 Z変数

Zは、標識変数の名前の現行の長さからなる配列へのポインタです。XおよびYと同様に、Zもバインド記述子専用です。

DESCRIBE BIND VARIABLESは、現行の長さの配列にZ[0]からZ[N - 1]の要素を設定します。DESCRIBEを実行すると、それぞれの標識変数名の文字数がこの配列に格納されます。

15.4 予備知識

動的SQL方法4を実装するには次の処理についての知識が必要です。

- データの変換
- データ型の強制変換
- NULL/NOT NULLデータ型の処理

15.4.1 データの変換

この項では、T(データ型)の記述子配列について詳しく説明します。データ型の同値化と動的SQL方法4のどちらも使用しないホスト・プログラムでは、Oracleの内部データ型と外部データ型との変換方法はプリコンパイル時に決定されます。デフォルトでは、プリコンパイラは宣言部内のそれぞれのホスト変数に特定の外部データ型を割り当てます。たとえば、プリコンパイラは *int*型のホスト変数にINTEGER外部データ型を割り当てます。

しかし方法4を使用すると、データの変換および形式を制御できます。データの変換方法を指定するには、*T*記述子配列にデータ型コードを設定します。

15.4.1.1 内部データ型

内部データ型は、Oracleがデータベース表に列値を格納するための形式と、疑似列値を表すための形式を指定します。

DESCRIBE SELECT LISTコマンドを発行すると、Oracleはそれぞれの選択リスト項目に対する内部データ型コードを*T*記述子配列に戻します。たとえば、*i*番目の選択リスト項目に対するデータ型コードはT[*i*]に戻されます。

表15-1に、Oracleの内部データ型とそのコードを示します。

表15-1 Oracle内部データ型

Oracle内部データ型	コード
VARCHAR2	1
NUMBER	2
LONG	8
BINARY_FLOAT	100
BINARY_DOUBLE	101
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHARACTER (または CHAR)	96
ユニバーサル ROWID	104

15.4.1.2 外部データ型

外部データ型は、入力ホスト変数と出力ホスト変数に値を格納するための形式を指定します。

DESCRIBE BIND VARIABLESコマンドはデータ型コードの*T*配列を0(ゼロ)に設定します。このため、OPENコマンドを発行する前に、それらのコードを再設定する必要があります。データ型コードは、様々なバインド変数にどの外部データ型が使用されるかをOracleに知らせます。*i*番目のバインド変数については、必要な外部データ型をT[*i*]に再設定してください。

表15-2に、Oracleの外部データ型とそのコード、および各外部データ型で通常使用するC言語のデータ型を示します。

表15-2 Oracle外部データ型とデータ型コード

外部データ型	コード	C言語のデータ型
VARCHAR2	1	char[n]
NUMBER	2	char[n] (n <= 22)
INTEGER	3	int
FLOAT	4	float
STRING	5	char[n+1]
VARNUM	6	char[n] (n <= 22)
DECIMAL	7	float
LONG	8	char[n]
SQLT_BFLOAT	21	float
SQLT_BDOUBLE	22	double
VARCHAR	9	char[n+2]
ROWID	11	char[n]
DATE	12	char[n]
VARRAW	15	char[n]
RAW	23	unsigned char[n]
LONG RAW	24	unsigned char[n]
UNSIGNED	68	unsigned int
DISPLAY	91	char[n]
LONG VARCHAR	94	char[n+4]
LONG VARRAW	95	unsigned char[n+4]

外部データ型	コード	C言語のデータ型
CHAR	96	char[n]
CHARF	96	char[n]
CHARZ	97	char[n+1]

関連項目

- Oracleのデータ型
- Oracle Database SQL言語リファレンス

15.4.2 データ型の強制変換

選択記述子の場合、DESCRIBE SELECT LISTはOracleの内部データ型をどれでも戻すことができます。文字データの場合など、ほとんどの場合内部データ型は適切な外部データ型と正確に対応しています。ただし、内部データ型には扱いにくい外部データ型にマップするものもあります。そのため、*T*記述子配列の一部の要素を再設定する必要がある場合があります。たとえば、NUMBER値をC言語の**float**値に対応するFLOAT値に再設定する場合があります。Oracleは、内部データ型と外部データ型の間の必要な変換をFETCH時に行います。このため、データ型の再設定は必ずDESCRIBE SELECT LISTの後、FETCHの前に行ってください。

バインド記述子の場合は、DESCRIBE BIND VARIABLESによってバインド変数のデータ型が戻されることはなく、バインド変数の数および名前のみ戻されます。したがって、データ型コードの*T*配列を明示的に設定することで、それぞれのバインド変数の外部データ型をOracleに通知する必要があります。Oracleは、内部データ型と外部データ型の間の必要な変換をOPEN時に行います。

*T*記述子配列でデータ型コードを再設定すると、データ型を強制変換することになります。たとえば、*i*番目の選択リスト値をSTRINGに強制変換するには、次の文を使用します。

```
/* Coerce select-list value to STRING. */
select_des->T[i] = 5;
```

データ表示用にNUMBERの選択リスト値をSTRINGに強制変換するときは、値の精度と位取りのバイトを抽出し、それらを使用して最大表示長を算出する必要もあります。FETCHの前に、*L* (長さ)記述子配列の該当する要素を再設定し、使用するバッファの長さをOracleに通知する必要があります。

たとえば、DESCRIBE SELECT LISTによって*i*番目の選択リスト項目のデータ型がNUMBER型であるとわかっているとします。このとき**float**型で宣言されているC変数に戻り値を格納する場合は、T[*i*]には4を、L[*i*]にはシステムが定める**float**の長さを設定するのみで済みます。

> 警告:
>
> DESCRIBE SELECT LIST によって戻される内部データ型が、目的に合わない場合もあります。DATE 型および NUMBER 型がその例です。DATE 型の選択リスト項目を DESCRIBE すると、Oracle ではデータ型コード 12 が *T* 記述子配列に戻されます。FETCH の前にコードを再設定しないかぎり、日付の値はその 7 バイト内部形式で戻されます。日付を文字形式(DD-MON-YY)で取得するには、12 に設定されているデータ型コードを 1(VARCHAR2)または 5(STRING)に変更し、7 に設定されている *L* 値を 9 または 10 に増やします。

NUMBER 型の選択リスト項目を同じ要領で DESCRIBE すると、Oracle ではデータ型コード 2 が *T* 配列に戻されます。FETCH の前にコードを再設定しないかぎり、数値はその内部形式で戻されるため、おそらく求めている値とは異なります。そのときは、2 に設定されているコードを 1(VARCHAR2)、3(INTEGER)、4(FLOAT)、5(STRING)またはその他の適切なデータ型に変更します。

関連項目

- 精度と位取りの抽出

15.4.2.1 精度および位取りの抽出

ライブラリ関数SQLNumberPrecV6()(従来のsqlprc()は、精度と位取りを抽出します。この関数は通常、DESCRIBE SELECT LISTの後に使用します。その最初の引数はL[*i*]です。次の構文で、SQLNumberPrecV6()をコールします。

注意:

プラットフォームの正しいプロトタイプは、プラットフォーム固有の *SQLNumberPrecV6* ヘッダー・ファイルを参照してください。

```
SQLNumberPrecV6(dvoid *runtime_context,  int *length,  int *precision,
```

int *scale);

各パラメータの意味は次のとおりです。

構文	説明
runtime_context	ランタイム・コンテキストへのポインタ
length	Oracle の NUMBER 値を格納する長い整変数へのポインタ。長さは L[i]に格納されます。値の位取りおよび精度はそれぞれ、下位バイトおよびその上のバイトに格納されます。
precision	NUMBER 値の精度を戻す整変数へのポインタ。精度とは有効桁数を指します。サイズが未指定の NUMBER が選択リスト項目によって参照される場合は、precision の値は 0 (ゼロ)に設定されます。この場合、サイズが未指定なので、最大精度の 38 とみなされます。
scale	NUMBER 値の位取りを戻す整変数へのポインタ。位取りには四捨五入する位置を指定します。たとえば位取りが 2 のときは、1/100 の倍数の近似値に値が四捨五入される(3.456 は 3.46 になる)ことを意味します。また位取りが-3 のときは、1000 の倍数の近似値に値が四捨五入される(3456 が 3000 になる)ことを意味します。

位取りが負の場合は、その絶対値を長さに追加してください。たとえば、精度に3、位取りに-2を指定すると、99900までの値が有効になります。

次の例に、SQLNumberPrecV6()を使用して、STRINGに強制変換するNUMBER値の最大値表示長を計算する方法を示

します。

```
/* Declare variables for the function call. */
sqlda         *select_des;  /* pointer to select descriptor */
int            prec;        /* precision                    */
int            scal;        /* scale                        */
extern void SQLNumberPrecV6();  /* Declare library function. */
/* Extract precision and scale. */
SQLNumberPrecV6(SQL_SINGLE_RCTX, &(select_des->L[i]), &prec, &scal);
/* Allow for maximum size of NUMBER. */
if (prec == 0)
    prec = 38;
/* Allow for possible decimal point and sign. */
select_des->L[i] = prec + 2;
/* Allow for negative scale. */
if (scal < 0)
    select_des->L[i] += -scal;
```

この関数コールの最初の引数は長さの配列の*i*番目の要素を指します。また、パラメータは3つともすべてアドレスであることに注意してください。

SQLNumberPrecV6()関数では、一部のSQLデータ型の精度と位取りの値に0(ゼロ)が戻されます。SQLNumberPrecV7()関数も同様で、次に示すSQLデータ型の場合を除けば引数リストも戻り値も同じです。

表15-3 SQLデータ型の精度と位取り

SQLデータ型	2進数精度	位取り
FLOAT	126	-127
FLOAT(N)	N(範囲は 1 から 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

15.4.3 NULLまたはNOT NULLデータ型の処理

すべての選択リスト列(式は不可)について、DESCRIBE SELECT LISTは選択記述子のデータ型配列*T*にNULL/NOT NULLインジケータを戻します。*i*番目の選択リスト列にNOT NULL制約が指定されていると、T[*i*]の上位ビットはオフにされます。それ以外の場合は上位ビットが設定されます。

OPEN文またはFETCH文でデータ型を使用する前に、すでにNULL/NOT NULLビットが設定されているときは、そのビットをオフにする必要があります。(このビットは絶対にオンにしないでください。)

列にNULLが有効かどうかを調べ、データ型のNULL/NOT NULLビットを消去するには、ライブラリ関数SQLColumnNullCheck()(従来のsqlnul())を使用します。次の構文で、SQLColumnNullCheck()をコールします。

```
SQLColumnNullCheck(dvoid *context, unsigned short *value_type,
      unsigned short *type_code, int *null_status);
```

各パラメータの意味は次のとおりです。

構文	説明
context	ランタイム・コンテキストへのポインタ
value_type	選択リスト列のデータ型コードを格納する符号なし short int 型変数へのポインタ。データ型は T[i]に格納されます。
type_code	選択リスト列のデータ型コードを戻す符号なし short int 型変数へのポインタ。上位ビットはオフにされています。
null_status	選択リスト列の NULL 状態を戻す int 型変数へのポインタ。1 は列が NULL を許可し、0 は許可しないことを意味します。

次の例に、SQLColumnNullCheck()の使用方法を示します。

```
/* Declare variables for the function call. */
sqlda  *select_des;      /* pointer to select descriptor */
unsigned short   dtype;  /* datatype without null bit    */
int   nullok;            /* 1 = null, 0 = not null       */
extern void SQLColumnNullCheck();    /* Declare library function.    */
/* Find out whether column is not null. */
SQLColumnNUllCheck(SQL_SINGLE_RCTX, (unsigned short *)&(select_des->T[i]), &dtype, &nullok);
if (nullok)
{
    /* Nulls are allowed. */
    ...
    /* Clear the null/not null bit. */
SQLColumnNullCheck(SQL_SINGLE_RCTX, &(select_des->T[i]), &(select_des->T[i]), &nullok);
}
```

SQLColumnNullCheck()関数の2回目のコールで指定されている1番目と2番目の引数は、データ型配列の*i*番目の要素を指します。また、パラメータは3つともアドレスであることに注意してください。

15.5 基本ステップ

方法4は任意の動的SQL文に使用できます。次の例には問合せの処理が示されているため、入力ホスト変数と出力ホスト変数の両方の処理方法が理解できます。

このサンプル・プログラムでは次の手順に従って動的問合せを処理します。

1. 問合せのテキストを保持するためのホスト文字列を宣言部で宣言します。
2. 選択SQLDAとバインドSQLDAを宣言します。
3. 選択記述子とバインド記述子に対する記憶域を割り当てます。
4. DESCRIBEできる選択リスト項目とプレースホルダの最大数を設定します。
5. 問合せのテキストをホスト文字列に設定します。
6. ホスト文字列から問合せをPREPAREします。
7. 問合せ用のカーソルをDECLAREします。
8. バインド記述子にバインド変数をDESCRIBEします。
9. プレースホルダの最大数をDESCRIBEによって実際に検出された数に再設定します。
10. DESCRIBEで検出されたバインド変数の値を取得し、それらの変数に対する記憶域を割り当てます。

11. バインド記述子を使用してカーソルをOPENします。
12. 選択記述子に選択リストをDESCRIBEします。
13. 選択リスト項目の最大数をDESCRIBEにより実際に検出された数に再設定します。
14. 表示用にそれぞれの選択リスト項目の長さおよびデータ型を再設定します。
15. 選択記述子が指している割当て済のデータ・バッファに(INTO)データベースの行をFETCHします。
16. FETCHにより戻された選択リストの値を処理します。
17. 選択リスト項目、プレースホルダ、標識変数および記述子に対する記憶域の割当てを解除します。
18. カーソルをCLOSEします。

注意:

動的 SQL 文に含まれる選択リスト項目またはプレースホルダの数が明確である場合、一部の手順は必要ありません。

15.6 各手順の詳細

この項では、それぞれのステップを詳しく説明します。章の終わりには、方法4を使用したコメント付きの完全なプログラム例を示します。

方法4では、埋込みSQL文を次のような順序で使用します。

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

方法4では、スクロール可能カーソルも使用できます。スクロール可能カーソルには、埋込みSQL文を次の順序で使用する必要があります。

```
 EXEC SQL PREPARE statement_name
           FROM { :host_string | string_literal };
 EXEC SQL DECLARE cursor_name SCROLL CURSOR FOR statement_name;
 EXEC SQL DESCRIBE BIND VARIABLES  FOR statement_name
           INTO bind_descriptor_name;
 EXEC SQL OPEN cusor_name
           [ USING DESCRIPTOR bind_descriptor_name];
 EXEC SQL DESCRIBE [ SELECT LIST FOR] statement_name
           INTO select_descriptor_name;
 EXEC SQL FETCH  [ FIRST| PRIOR|NEXT|LAST|CURRENT | RELATIVE fetch_offset
           |ABSOLUTE fetch_offset ]  cursor_name USING DESCRIPTOR
            select_descriptor_name;
 EXEC SQL CLOSE cursor_name;
```

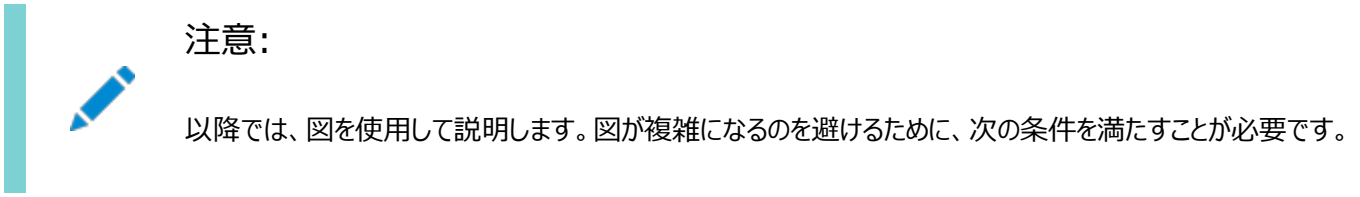
動的問合せの選択リスト項目の数がわかっているときは、DESCRIBE SELECT LISTを省略するとともに次の方法3のFETCH文を使用できます。

```
EXEC SQL FETCH cursor_name INTO host_variable_list;
```

または、動的SQL文のバインド変数に対するプレースホルダの数が明確な場合は、DESCRIBE BIND VARIABLESを省略するとともに次の方法3のOPEN文を使用できます。

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

次にこれらの文により、ホスト・プログラムで記述子を使用して動的SQL文を受け入れ、それを処理する方法を説明します。



注意:

以降では、図を使用して説明します。図が複雑になるのを避けるために、次の条件を満たすことが必要です。

- 記述子配列は3要素までに制限します。
- 名前の最大長は5文字以内に制限します。
- 値の最大長は10文字以内に制限します。

15.6.1 ホスト文字列の宣言

プログラムには、動的SQL文のテキストを格納するためのホスト変数が必要です。ホスト変数(ここでは*select_stmt*)は文字列として宣言する必要があります。

```
    ...
    int      emp_number;
    VARCHAR  emp_name[10];
    VARCHAR  select_stmt[120];
    float    bonus;
```

15.6.2 SQLDAの宣言

ここでは、SQLDAのデータ構造体をハードコードするかわりに、次のようにINCLUDEを使用してSQLDAをプログラムにコピーします。

```
#include <sqlda.h>
```

問合せに含まれる選択リスト項目の数またはバインド変数のプレースホルダの数が不明なため、次のように選択記述子とバインド記述子のポインタを宣言します。

```
sqlda  *select_des;
sqlda  *bind_des;
```

15.6.3 記述子用の記憶域の割当て

記述子用の記憶域を割り当てるには、SQLSQLDAAlloc() ライブラリ関数を使用することを思い出してください。ANSI C表記法による構文は次のとおりです。

```
SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int max_vars, unsigned int
```

```
max_name, unsigned int max_ind_name);
```

SQLSQLDAAlloc()関数は、記述子構造体およびポインタ変数*V*、*L*、*T*、*I*によってアドレス指定される配列を割り当てます。

*max_name*が0(ゼロ)以外のときは、ポインタ変数*S*、*M*、*C*によってアドレス指定される配列が割り当てられます。
*max_ind_name*が0(ゼロ)以外のときは、ポインタ変数*X*、*Y*、*Z*によってアドレス指定される配列が割り当てられます。
*max_name*と*max_ind_name*が0(ゼロ)のときは、領域は割り当てられません。

SQLSQLDAAlloc()は、成功すると構造体のポインタを戻します。SQLSQLDAAlloc()は、失敗すると0(ゼロ)を戻します。

この例では、選択記述子とバインド記述子を次のように割り当てます。

```
select_des = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t) 5, (size_t) 0);
bind_des = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t) 5, (size_t) 4);
```

選択記述子には、*X*によってアドレス指定される配列に領域が割り当てられないようにするため、常に*max_ind_name*を0(ゼロ)に設定します。

15.6.4 DESCRIBEへの最大数の設定

DESCRIBEできる選択リスト項目またはプレースホルダの最大数を次のように設定します。

```
select_des->N = 3;
bind_des->N = 3;
```

図15-2と図15-3に、結果として得られる記述子を示します。

注意:

選択記述子の場合(図 15-2)、標識変数名の選択は消されて使用されないことを示します。

図15-2 初期化された選択記述子

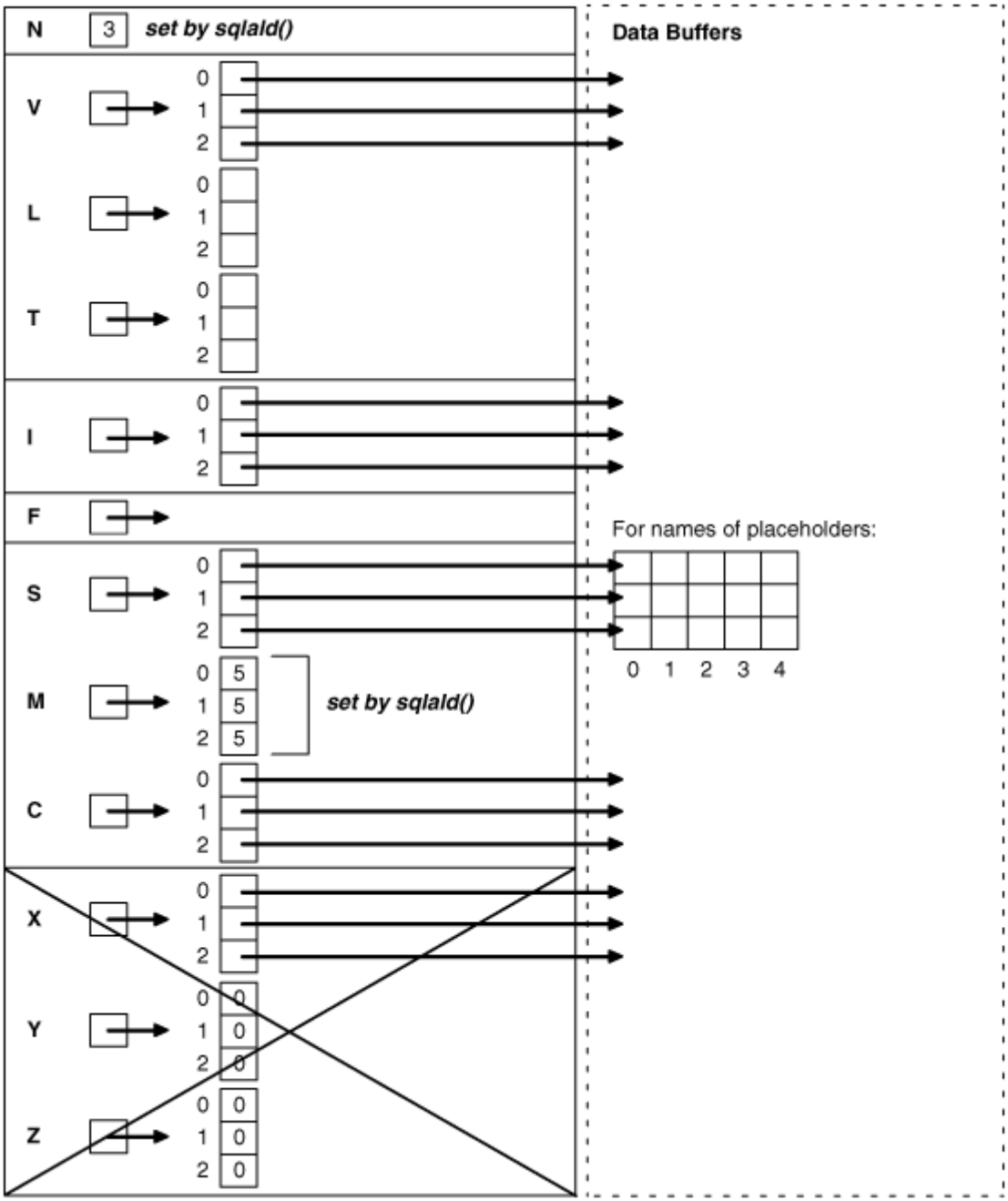
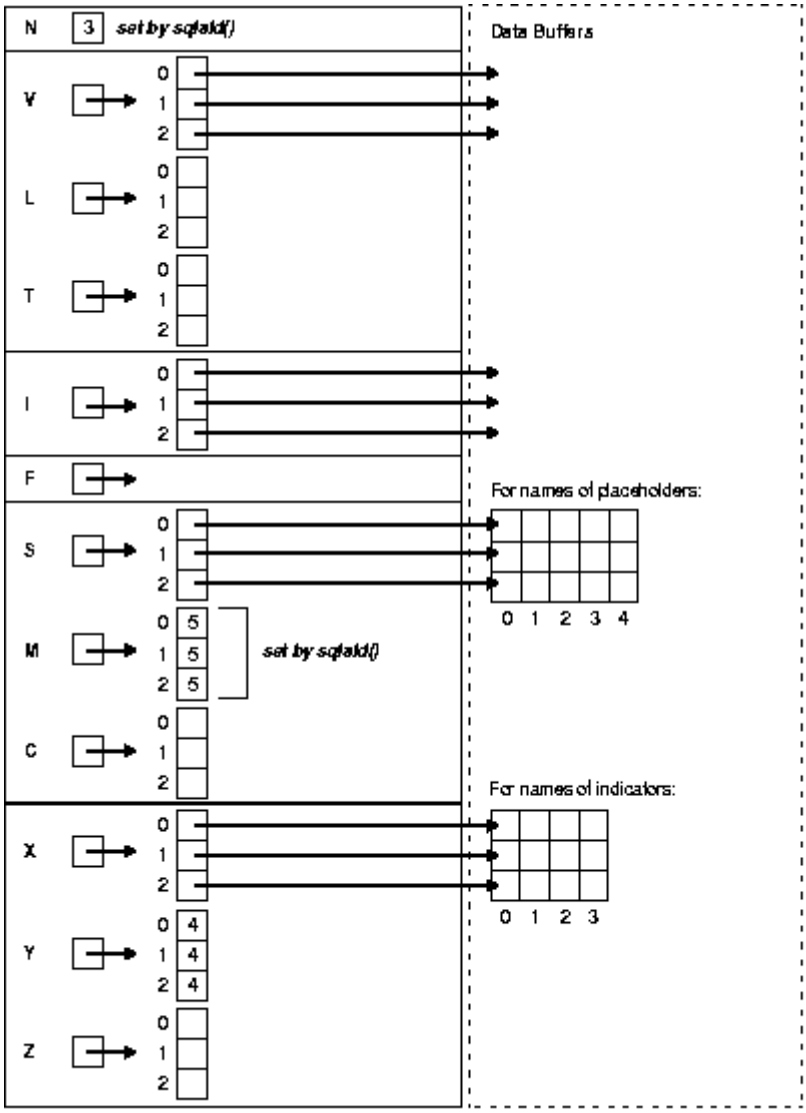


図15-3 初期化されたバインド記述子



15.6.5 ホスト文字列への問合せテキストの設定

次の例では、ユーザーにSQL文の入力を求め、入力された文字列を*select_stmt*に格納します。

```
printf("¥n¥nEnter SQL statement: ");
gets(select_stmt.arr);
select_stmt.len = strlen(select_stmt.arr);
```

このときユーザーが次の文字列を入力したと仮定します。

```
"SELECT ename, empno, comm FROM emp WHERE comm < :bonus"
```

15.6.6 ホスト文字列からの問合せのPREPARE

PREPAREは、このSQL文を解析して名前を指定します。例では、PREPAREはホスト文字列*select_stmt*を解析してから、それに*sql_stmt*という名前を指定します。

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

15.6.7 カーソルの宣言

DECLARE CURSORは名前を指定し、特定のSELECT文に対応付けることにより、カーソルを定義します。

静的問合せ用のカーソルを宣言するには、次の構文を使用します。

```
EXEC SQL DECLARE cursor_name CURSOR FOR SELECT ...
```

動的問合せ用にカーソルを宣言するには、PREPAREにより動的問合せに指定された文の名前で静的問合せを置換します。例では、DECLARE CURSORは*emp_cursor*という名前のカーソルを定義し、このカーソルを*sql_stmt*に対応付けます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

> 注意:
>
> 問合せのみでなく、すべての動的 SQL 文のカーソルを宣言できます。また、問合せ以外の場合も、カーソルの OPEN により動的 SQL 文を実行します。

15.6.8 バインド変数のDESCRIBE

DESCRIBE BIND VARIABLESは、バインド記述子にプレースホルダの記述を設定します。例では、DESCRIBEは次のように*bind_des*を準備します。

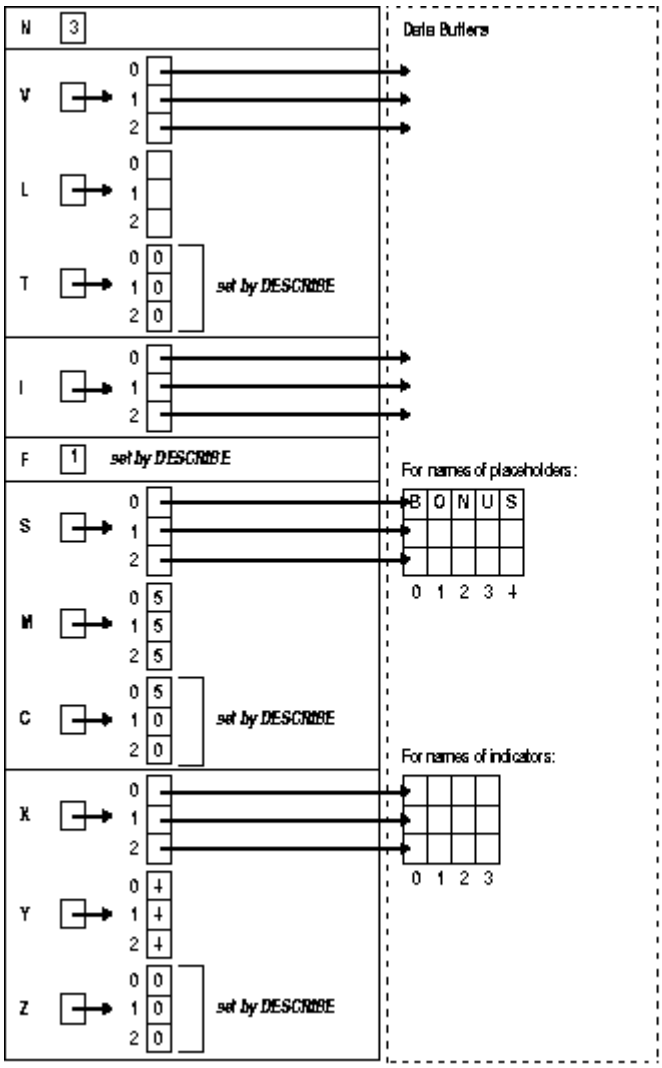
```
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
```

*bind_des*はコロンで始めることはできません。

DESCRIBE BIND VARIABLES文はPREPARE文の後で、かつOPEN文の前に指定する必要があります。

図15-4に、DESCRIBE実行後のバインド記述子を示します。SQL文の実行で検出されたプレースホルダの実際の数が、DESCRIBEによって*F*に設定されています。

図15-4 DESCRIBE後のバインド記述子



15.6.9 プレースホルダの最大数の再設定

次に、プレースホルダの最大数を、DESCRIBEによって実際に検出された数に再設定する必要があります。

```
bind_des->N = bind_des->F;
```

15.6.10 バインド変数の値の取得と記憶域の割当て

プログラムでは、SQL文で検出されたバインド変数に対する値を取得し、メモリーを割り当てる必要があります。値はどのように取得してもかまいません。たとえば値をハードコードしたり、ファイルから読み込んだり、または対話形式で入力することもできます。

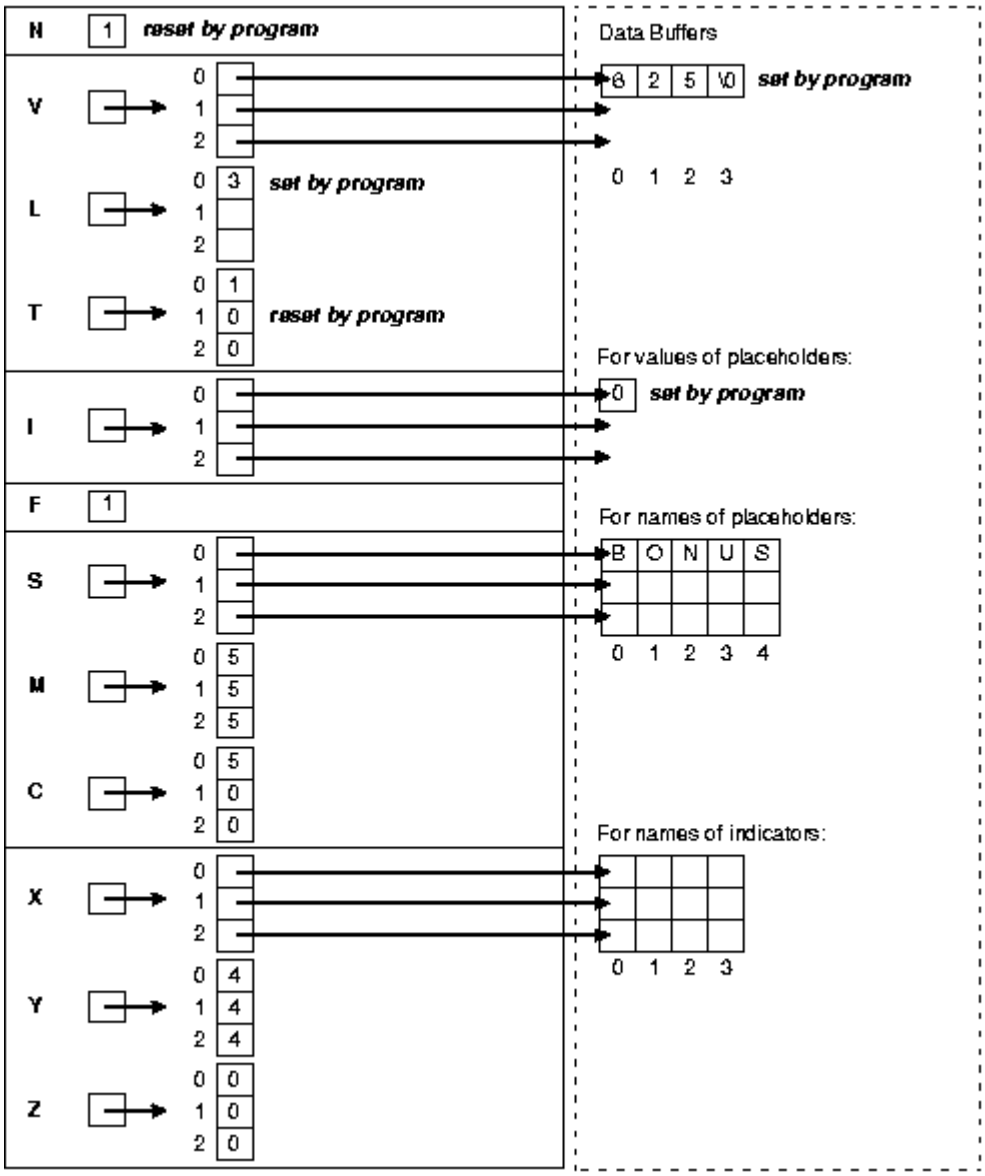
例では、問合せのWHERE句のプレースホルダ*bonus*に置換されるバインド変数に値を割り当てる必要があります。そこで、ユーザーに値の入力を求め、入力された値を次のように処理します。

```
for (i = 0; i < bind_des->F; i++)
{
    printf("¥nEnter value of bind variable %.*s:¥n? ",
        (int) bind_des->C[i], bind_des->S[i]);
    gets(hostval);
    /* Set length of value. */
    bind_des->L[i] = strlen(hostval);
    /* Allocate storage for value and null terminator. */
    bind_des->V[i] = malloc(bind_des->L[i] + 1);
    /* Allocate storage for indicator value. */
    bind_des->I[i] = (unsigned short *) malloc(sizeof(short));
    /* Store value in bind descriptor. */
    strcpy(bind_des->V[i], hostval);
```

```
    /* Set value of indicator variable. */
    *(bind_des->I[i]) = 0;  /* or -1 if "null" is the value */
    /* Set datatype to STRING. */
    bind_des->T[i] = 5;
}
```

ここでは、ユーザーが*bonus*の値として625を入力したと想定します。図15-5に、結果として得られるバインド記述子を示します。値はヌル文字で終了します。

図15-5 値を割り当てた後のバインド記述子



15.6.11 カーソルのOPEN

動的問合せに使用するOPEN文は、カーソルがバインド記述子に対応付けられることを除けば静的問合せに使用するものと同じです。実行時に決定され、バインド記述子表の要素でアドレス指定したバッファに格納された値を使用して、SQL文を評価します。問合せの場合は、アクティブ・セットの識別にも同じ値を使用します。

例では、OPENは次のように*emp_cursor*を*bind_des*に対応付けます。

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR bind_des;
```

*bind_des*はコロンで始めることはできません。

OPENはSQL文を実行します。問合せのときは、OPENはアクティブ・セットを決定するとともにカーソルを先頭行に位置づけます。

15.6.12 選択リストのDESCRIBE

動的SQL文が問合せのときは、DESCRIBE SELECT LIST文はOPEN文の後で、かつFETCH文の前に指定する必要があります。

DESCRIBE SELECT LISTは、選択記述子に選択リスト項目の記述を設定します。例では、DESCRIBEは次のように*select_des*を準備します。

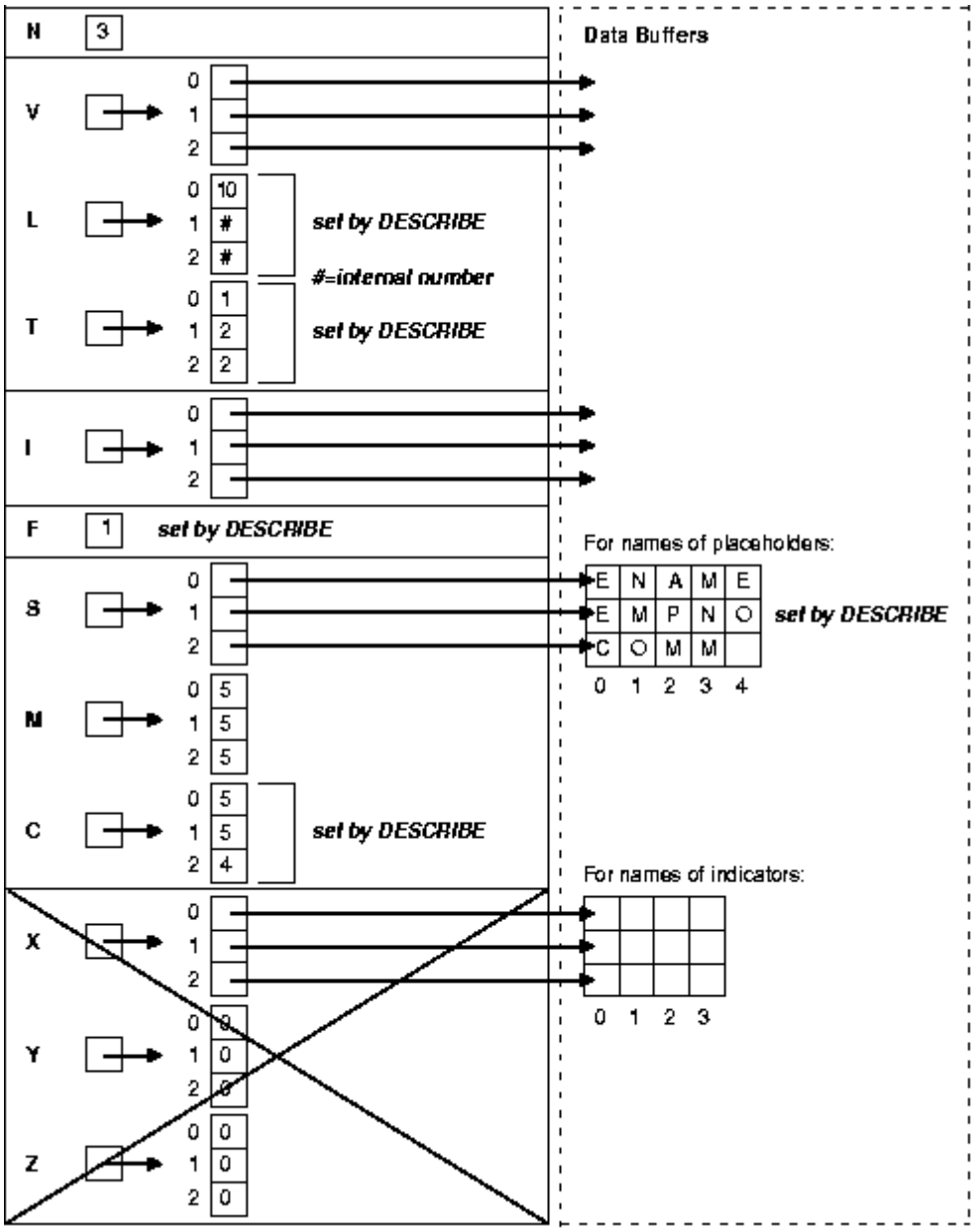
```
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
```

Oracleのデータ・ディクショナリにアクセスすることで、DESCRIBEは各選択リストの値の長さとデータ型を設定します。

図15-6に、DESCRIBE実行後の選択記述子を示します。問合せの選択リストで検出された項目の実際の数がDESCRIBEによって*F*に設定されています。SQL文が問合せでないときは、*F*は0(ゼロ)に設定されます。

また、NUMBER型の長さはまだ使用できないことに注意してください。NUMBERと定義した列には、ライブラリ関数SQLNumberPrecV6() を使用して精度と位取りを抽出する必要があります。

図15-6 DESCRIBE実行後の選択記述子



関連項目

- データ型の強制変換

15.6.13 選択リスト項目の最大数の再設定

次に選択リスト項目の最大数を、DESCRIBEにより実際に検出された数に再設定する必要があります。

```
select_des->N = select_des->F;
```

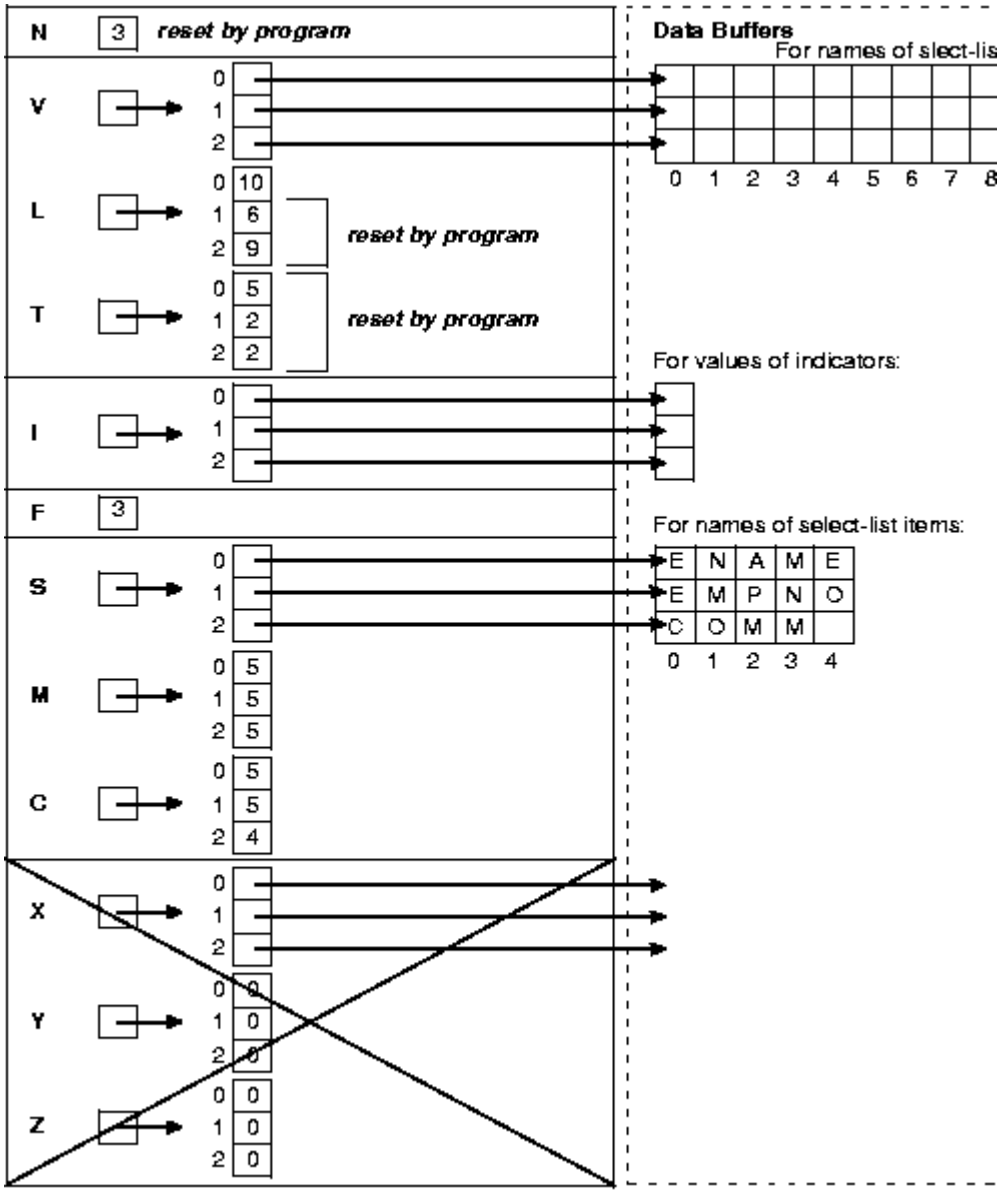
15.6.14 各選択リスト項目の長さとデータ型の再設定

例では、選択リストの値をFETCHする前に、ライブラリ関数malloc()を使用して、記憶域を割り当てます。また、表示用の長さとデータ型の配列の要素のいくつかを再設定します。

```
for (i=0; i<select_des->F; i++)
{
    /* Clear null bit. */
    SQLColumnNullCheck(SQL_SINGLE_RCTX, (unsigned short *)&(select_des->T[i]),
         (unsigned short *)&(select_des->T[i]), &nullok);
    /* Reset length if necessary. */
    switch(select_des->T[i])
    {
        case  1: break;
        case  2: SQLNumberPrecV6(SQL_SINGLE_RCTX, (unsigned long *)
                      &(select_des->L[i]), &prec, &scal);
                 if (prec == 0) prec = 40;
                 select_des->L[i] = prec + 2;
                 if (scal < 0) select_des->L[i] += -scal;
                 break;
        case  8: select_des->L[i] = 240;
                 break;
        case 11: select_des->L[i] = 18;
                 break;
        case 12: select_des->L[i] = 9;
                 break;
        case 23: break;
        case 24: select_des->L[i] = 240;
                 break;
    }
    /* Allocate storage for select-list value. */
    select_des->V[i] = malloc(select_des->L[i]+1);
    /* Allocate storage for indicator value. */
    select_des->I[i] = (short *)malloc(sizeof(short *));
    /* Coerce all datatypes except LONG RAW to STRING. */
    if (select_des->T[i] != 24) select_des->T[i] = 5;
}
```

図15-7に、結果として得られる選択記述子を示します。NUMBERの長さはこのとき使用可能となります。データ型はすべてSTRINGです。L[1]およびL[2]の長さはそれぞれ6と9になっています。これは、DESCRIBEされた長さ4と7にそれぞれ符号と小数点のための2を加算したためです。

図15-7 FETCH前の選択記述子



15.6.15 アクティブ・セットからの行のFETCH

FETCHはアクティブ・セットから1行を戻し、データ・バッファに選択リストの値を格納してから、カーソルをアクティブ・セットの次の行に進めます。行がなくなると、FETCHは「データが見つかりません。」のOracleエラー・コードを*sqlca.sqlcode*に設定します。例では、次に示すように、FETCHによってENAME、EMPNOおよびCOMMの列の値を*select_des*に戻します。

```
EXEC SQL FETCH emp_cursor USING DESCRIPTOR select_des;
```

図15-8に、FETCH実行後の選択記述子を示します。Oracleは選択リストの値とインジケータの値を、*V*と*I*の要素によってアドレス指定されるデータ・バッファに格納しています。

データ型1の出力バッファについては、Oracleは*L*配列に格納された長さを使用し、CHARまたはVARCHAR2のデータを左揃えにしてから、NUMBERデータを右揃えにします。データ型5(STRING)の出力バッファについては、値を左揃えにし、CHAR、VARCHAR2およびNUMBERのデータにヌル文字を付けます。

値'MARTIN'は、EMP表のVARCHAR2(10)列から取り出されました。L[0]の長さを使用して、Oracleは10バイトのフィールドの値を左揃えにしてバッファを埋めます。

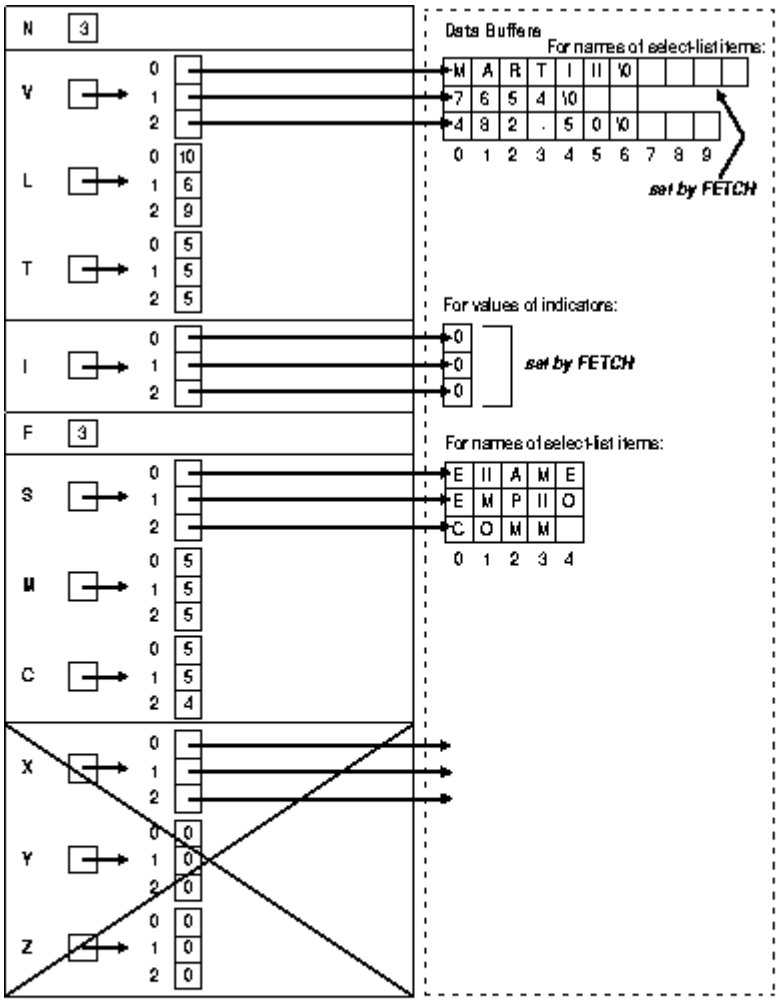
値7654はNUMBER(4)列から取り出され、'7654'に強制変換されています。しかし、符号と小数点を使用可能にするため、L[1]の長さが2のみ増えています。そこでOracleは6バイトのフィールドの値を左揃えにしてから、ヌル文字を付けます。

値482.50はNUMBER(7,2)列から取り出され、'482.50'に強制変換されています。L[2]の長さが2のみ増えています。Oracleは9バイトのフィールドの値を左揃えにしてから、ヌル文字を付けます。

15.6.16 選択リストの値の取得および処理

FETCH後、プログラムで戻り値を処理できます。例では、列ENAME、EMPNOおよびCOMMの値が処理されます。

図15-8 FETCH後の選択記述子



15.6.17 記憶域の割当て解除

malloc()によって割り当てられた記憶域を解除するには、free()ライブラリ関数を使用します。構文は次のとおりです。

```
free(char *pointer);
```

例では、選択リスト項目、バインド変数および標識変数の値に対する記憶域の割当てを次のように解除します。

```
for (i = 0; i < select_des->F; i++)   /* for select descriptor */
{
    free(select_des->V[i]);
    free(select_des->I[i]);
}
for (i = 0; i < bind_des->F; i++)   /* for bind descriptor */
{
```

```
    free(bind_des->V[i]);
    free(bind_des->I[i]);
}
```

記述子の記憶域の割当てを解除するには、次の構文のライブラリ関数SQLSQLDAFree()を使用します。

```
SQLSQLDAFree(context, descriptor_name);
```

記述子はSQLSQLDAAlloc()を使用して割り当ててください。そうしないと結果は予測できなくなります。

例では、選択記述子とバインド記述子に対する記憶域の割当てを次のように解除します。

```
SQLSQLDAFree(SQL_SINGLE_RCTX, select_des);
SQLSQLDAFree(SQL_SINGLE_RCTX, bind_des);
```

15.6.18 カーソルのCLOSE

CLOSEはカーソルを使用禁止にします。例では、CLOSEは次のように*emp_cursor*を使用禁止にします。

```
EXEC SQL CLOSE emp_cursor;
```

15.6.19 ホスト配列の使用について

方法4で入力ホスト配列または出力ホスト配列を使用するには、オプションのFOR句を使用してホスト配列のサイズをOracleに通知する必要があります。

次の構文を使用して、*i*番目の選択リスト項目またはバインド変数に記述子エントリを設定する必要があります。

```
V[i] = array_address;
L[i] = element_size;
```

*array_address*はホスト配列のアドレスで、*element_size*はある配列要素のサイズです。

EXECUTE文またはFETCH文(どちらか適切な方)にFOR句を指定して、処理対象の配列要素の数をOracleに通知する必要があります。Oracleがホスト配列のサイズを認識する方法は他にないので、この手順は必須です。

次の完全なプログラム例では、3つの入力ホスト配列を使用してEMP表に行をINSERTします。方法4による問合せ以外のDML文にもEXECUTEを使用できます。

```
#include <stdio.h>
#include <sqlcpr.h>
#include <sqlda.h>
#include <sqlca.h>

#define NAME_SIZE    10
#define INAME_SIZE   10
#define ARRAY_SIZE    5

/* connect string */
char *username = "scott/tiger";

char *sql_stmt =
"INSERT INTO emp (empno, ename, deptno) VALUES (:e, :n, :d)";
int  array_size = ARRAY_SIZE;  /* must have a host variable too */

SQLDA   *binda;

char    names[ARRAY_SIZE][NAME_SIZE];
int     numbers[ARRAY_SIZE], depts[ARRAY_SIZE];
```

```
/* Declare and initialize indicator vars. for empno and deptno columns */
short   ind_empno[ARRAY_SIZE] = {0,0,0,0,0};
short   ind_dept[ARRAY_SIZE] = {0,0,0,0,0};

main()
{
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;

/* Connect */
    EXEC SQL CONNECT :username;
    printf("Connected.¥n");

/* Allocate the descriptors and set the N component.
   This must be done before the DESCRIBE. */
    binda = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, NAME_SIZE, INAME_SIZE);
    binda->N = 3;

/* Prepare and describe the SQL statement. */
    EXEC SQL PREPARE stmt FROM :sql_stmt;
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO binda;

/* Initialize the descriptors. */
    binda->V[0] = (char *) numbers;
    binda->L[0] = (long) sizeof (int);
    binda->T[0] = 3;
    binda->I[0] = ind_empno;

    binda->V[1] = (char *) names;
    binda->L[1] = (long) NAME_SIZE;
    binda->T[1] = 1;
    binda->I[1] = (short *)0;

    binda->V[2] = (char *) depts;
    binda->L[2] = (long) sizeof (int);
    binda->T[2] = 3;
    binda->I[2] = ind_dept;

/* Initialize the data buffers. */
    strcpy(&names[0] [0], "ALLISON");
    numbers[0] = 1014;
    depts[0] = 30;

    strcpy(&names[1] [0], "TRUSDALE");
    numbers[1] = 1015;
    depts[1] = 30;

    strcpy(&names[2] [0], "FRAZIER");
    numbers[2] = 1016;
    depts[2] = 30;

    strcpy(&names[3] [0], "CARUSO");
    numbers[3] = 1017;
    ind_dept[3] = -1;       /* set indicator to -1 to insert NULL */
    depts[3] = 30;          /* value in depts[3] is ignored */

    strcpy(&names[4] [0], "WESTON");
    numbers[4] = 1018;
    depts[4] = 30;
```

```
/* Do the INSERT. */
    printf("Adding to the Sales force...¥n");

    EXEC SQL FOR :array_size
    EXECUTE stmt USING DESCRIPTOR binda;

/*  Print rows-processed count. */
    printf("%d rows inserted.¥n¥n", sqlca.sqlerrd[2]);
    EXEC SQL COMMIT RELEASE;
    exit(0);

sql_error:
/* Print Oracle error message. */
    printf("¥n%.70s", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

関連項目

- ホスト配列

15.6.20 sample12.pc

配列のフェッチを使用した簡単な動的SQLの例は、demoディレクトリのsample12.pcファイルにあります。

15.7 サンプル・プログラム: 動的SQL方法4

このプログラムでは、動的SQL方法4を使用するために必要な基本ステップを示します。Oracleへの接続後に、プログラムでは次のことが実行されます。

- SQLSQLDAAlloc()を使用して記述子にメモリーを割り当てます。
- ユーザーに対してSQL文の入力を求めるプロンプトを表示します。
- 文をPREPAREします。
- カーソルをDECLAREします。
- DESCRIBE BINDを使用してバインド変数をチェックします。
- カーソルをOPENします。
- 選択リスト項目をDESCRIBEします。

入力されたSQL文が問合せのときは、プログラムは各行のデータをFETCHしてからカーソルをCLOSEします。このプログラムはdemoディレクトリのファイルsample10.pcにあり、オンラインで利用できます。

```
/************************************************************************
Sample Program 10:  Dynamic SQL Method 4

This program connects you to ORACLE using your username and
password, then prompts you for a SQL statement.  You can enter
any legal SQL statement.  Use regular SQL syntax, not embedded SQL.
Your statement will be processed.  If it is a query, the rows
fetched are displayed.
You can enter multiline statements.  The limit is 1023 characters.
This sample program only processes up to MAX_ITEMS bind variables and
MAX_ITEMS select-list items.  MAX_ITEMS is #defined to be 40.
```

```
**********************************************************************/

#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <sqlda.h>
#include <stdlib.h>
#include <sqlcpr.h>

/* Maximum number of select-list items or bind variables. */
#define MAX_ITEMS         40

/* Maximum lengths of the _names_ of the
   select-list items or indicator variables. */
#define MAX_VNAME_LEN     30
#define MAX_INAME_LEN     30

#ifndef NULL
#define NULL  0
#endif

/* Prototypes */
#if defined(__STDC__)
  void sql_error(void);
  int oracle_connect(void);
  int alloc_descriptors(int, int, int);
  int get_dyn_statement(void);
  void set_bind_variables(void);
  void process_select_list(void);
  void help(void);
#else
  void sql_error(/*_ void _*/);
  int oracle_connect(/*_ void _*/);
  int alloc_descriptors(/*_ int, int, int _*/);
  int get_dyn_statement(/* void _*/);
  void set_bind_variables(/*_ void -*/);
  void process_select_list(/*_ void _*/);
  void help(/*_ void _*/);
#endif

char *dml_commands[] = {"SELECT", "select", "INSERT", "insert",
                        "UPDATE", "update", "DELETE", "delete"};

EXEC SQL INCLUDE sqlda;
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
    char    dyn_statement[1024];
    EXEC SQL VAR dyn_statement IS STRING(1024);
EXEC SQL END DECLARE SECTION;

SQLDA *bind_dp;
SQLDA *select_dp;

/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;

/* A global flag for the error routine. */
int parse_flag = 0;
```

```
void main()
{
    int i;

    /* Connect to the database. */
    if (oracle_connect() != 0)
        exit(1);

    /* Allocate memory for the select and bind descriptors. */
    if (alloc_descriptors(MAX_ITEMS, MAX_VNAME_LEN, MAX_INAME_LEN) != 0)
        exit(1);

    /* Process SQL statements. */
    for (;;)
    {
        (void) setjmp(jmp_continue);

        /* Get the statement.  Break on "exit". */
        if (get_dyn_statement() != 0)
            break;

        /* Prepare the statement and declare a cursor. */
        EXEC SQL WHENEVER SQLERROR DO sql_error();

        parse_flag = 1;     /* Set a flag for sql_error(). */
        EXEC SQL PREPARE S FROM :dyn_statement;
        parse_flag = 0;     /* Unset the flag. */

        EXEC SQL DECLARE C CURSOR FOR S;

        /* Set the bind variables for any placeholders in the
           SQL statement. */
        set_bind_variables();

        /* Open the cursor and execute the statement.
         * If the statement is not a query (SELECT), the
         * statement processing is completed after the
         * OPEN.
         */

        EXEC SQL OPEN C USING DESCRIPTOR bind_dp;

        /* Call the function that processes the select-list.
         * If the statement is not a query, this function
         * just returns, doing nothing.
         */
        process_select_list();

        /* Tell user how many rows processed. */
        for (i = 0; i < 8; i++)
        {
           if (strncmp(dyn_statement, dml_commands[i], 6) == 0)
           {
               printf("¥n¥n%d row%c processed.¥n", sqlca.sqlerrd[2],
                       sqlca.sqlerrd[2] == 1 ? '¥0' : 's');
               break;
           }
        }
    }       /* end of for(;;) statement-processing loop */
```

```
    /* When done, free the memory allocated for
       pointers in the bind and select descriptors. */
    for (i = 0; i < MAX_ITEMS; i++)
    {
        if (bind_dp->V[i] != (char *) 0)
            free(bind_dp->V[i]);
        free(bind_dp->I[i]);   /* MAX_ITEMS were allocated. */
        if (select_dp->V[i] != (char *) 0)
            free(select_dp->V[i]);
        free(select_dp->I[i]); /* MAX_ITEMS were allocated. */
    }

    /* Free space used by the descriptors themselves. */
    SQLSQLDAFree( SQL_SINGLE_RCTX, bind_dp);
    SQLSQLDAFree( SQL_SINGLE_RCTX, select_dp);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    /* Close the cursor. */
    EXEC SQL CLOSE C;

    EXEC SQL COMMIT WORK RELEASE;
    puts("¥nHave a good day!¥n");

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    return;
}


int oracle_connect()
{
    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR  username[128];
        VARCHAR  password[32];
    EXEC SQL END DECLARE SECTION;

    printf("¥nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
    username.arr[strlen((char *) username.arr)-1] = '¥0';
    username.len = (unsigned short)strlen((char *) username.arr);

    printf("password: ");
    fgets((char *) password.arr, sizeof password.arr, stdin);
    password.arr[strlen((char *) password.arr) - 1] = '¥0';
    password.len = (unsigned short)strlen((char *) password.arr);


    EXEC SQL WHENEVER SQLERROR GOTO connect_error;

    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    printf("¥nConnected to ORACLE as user %s.¥n", username.arr);

    return 0;

connect_error:
    fprintf(stderr, "Cannot connect to ORACLE as user %s¥n", username.arr);
    return -1;
}
```

```
/*
 *  Allocate the BIND and SELECT descriptors using SQLSQLDAAlloc().
 *  Also allocate the pointers to indicator variables
 *  in each descriptor.  The pointers to the actual bind
 *  variables and the select-list items are realloc'ed in
 *  the set_bind_variables() or process_select_list()
 *  routines.  This routine allocates 1 byte for select_dp->V[i]
 *  and bind_dp->V[i], so the realloc will work correctly.
 */

alloc_descriptors(size, max_vname_len, max_iname_len)
int size;
int max_vname_len;
int max_iname_len;
{
    int i;

    /*
     * The first SQLSQLDAAlloc parameter is the runtime context.

     * The second parameter determines the maximum number of
     * array elements in each variable in the descriptor. In
     * other words, it determines the maximum number of bind
     * variables or select-list items in the SQL statement.
     *
     * The third parameter determines the maximum length of
     * strings used to hold the names of select-list items
     * or placeholders.  The maximum length of column
     * names in ORACLE is 30, but you can allocate more or less
     * as needed.
     *
     * The fourth parameter determines the maximum length of
     * strings used to hold the names of any indicator
     * variables.  To follow ORACLE standards, the maximum
     * length of these should be 30.  But, you can allocate
     * more or less as needed.
     */

    if ((bind_dp =
       SQLSQLDAAlloc(SQL_SINGLE_RCTX, size, max_vname_len, max_iname_len)) ==
         (SQLDA *) 0)
    {
        fprintf(stderr,
            "Cannot allocate memory for bind descriptor.");
        return -1;  /* Have to exit in this case. */
    }

    if ((select_dp =
        SQLSQLDAAlloc (SQL_SINGLE_RCTX, size, max_vname_len, max_iname_len)) ==
           (SQLDA *) 0)
    {
        fprintf(stderr,
            "Cannot allocate memory for select descriptor.");
        return -1;
    }
    select_dp->N = MAX_ITEMS;

    /* Allocate the pointers to the indicator variables, and the
       actual data. */
    for (i = 0; i < MAX_ITEMS; i++) {
```

```
        bind_dp->I[i] = (short *) malloc(sizeof (short));
        select_dp->I[i] = (short *) malloc(sizeof(short));
        bind_dp->V[i] = (char *) malloc(1);
        select_dp->V[i] = (char *) malloc(1);
    }

    return 0;
}


int get_dyn_statement()
{
    char *cp, linebuf[256];
    int iter, plsql;


    for (plsql = 0, iter = 1; ;)
    {
        if (iter == 1)
        {
            printf("¥nSQL> ");
            dyn_statement[0] = '¥0';
        }

        fgets(linebuf, sizeof linebuf, stdin);

        cp = strrchr(linebuf, '¥n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
            continue;

        if ((strncmp(linebuf, "EXIT", 4) == 0) ||
            (strncmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        }

        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn_statement, ';'))))
        {
            *cp = '¥0';
            break;
```

```
        }
        else
        {
            iter++;
            printf("%3d  ", iter);
        }
    }
    return 0;
}



void set_bind_variables()
{
    int i, n;
    char bind_var[64];

    /* Describe any bind variables (input host variables) */
    EXEC SQL WHENEVER SQLERROR DO sql_error();

    bind_dp->N = MAX_ITEMS;  /* Initialize count of array elements. */
    EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_dp;

    /* If F is negative, there were more bind variables
       than originally allocated by SQLSQLDAAlloc(). */
    if (bind_dp->F < 0)
    {
        printf ("¥nToo many bind variables (%d), maximum is %d¥n.",
                    -bind_dp->F, MAX_ITEMS);
        return;
    }

    /* Set the maximum number of array elements in the
       descriptor to the number found. */
    bind_dp->N = bind_dp->F;

    /* Get the value of each bind variable as a
     * character string.
     *
     * C[i] contains the length of the bind variable
     *      name used in the SQL statement.
     * S[i] contains the actual name of the bind variable
     *      used in the SQL statement.
     *
     * L[i] will contain the length of the data value
     *      entered.
     *
     * V[i] will contain the address of the data value
     *      entered.
     *
     * T[i] is always set to 1 because in this sample program
     *      data values for all bind variables are entered
     *      as character strings.
     *      ORACLE converts to the table value from CHAR.
     *
     * I[i] will point to the indicator value, which is
     *      set to -1 when the bind variable value is "null".
     */
    for (i = 0; i < bind_dp->F; i++)
    {
```

```
        printf ("¥nEnter value for bind variable %.*s:  ",
               (int)bind_dp->C[i], bind_dp->S[i]);
        fgets(bind_var, sizeof bind_var, stdin);

        /* Get length and remove the new line character. */
        n = strlen(bind_var) - 1;

        /* Set it in the descriptor. */
        bind_dp->L[i] = n;

        /* (re-)allocate the buffer for the value.
           SQLSQLDAAlloc() reserves a pointer location for
           V[i] but does not allocate the full space for
           the pointer. */

         bind_dp->V[i] = (char *) realloc(bind_dp->V[i],
                         (bind_dp->L[i] + 1));

        /* And copy it in. */
        strncpy(bind_dp->V[i], bind_var, n);

        /* Set the indicator variable's value. */
        if ((strncmp(bind_dp->V[i], "NULL", 4) == 0) ||
                (strncmp(bind_dp->V[i], "null", 4) == 0))
            *bind_dp->I[i] = -1;
        else
            *bind_dp->I[i] = 0;

        /* Set the bind datatype to 1 for CHAR. */
        bind_dp->T[i] = 1;
    }
  return;
}



void process_select_list()
{
    int i, null_ok, precision, scale;

    if ((strncmp(dyn_statement, "SELECT", 6) != 0) &&
        (strncmp(dyn_statement, "select", 6) != 0))
    {
        select_dp->F = 0;
        return;
    }

    /* If the SQL statement is a SELECT, describe the
        select-list items.  The DESCRIBE function returns
        their names, datatypes, lengths (including precision
        and scale), and NULL/NOT NULL statuses. */

    select_dp->N = MAX_ITEMS;

    EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_dp;

    /* If F is negative, there were more select-list
       items than originally allocated by SQLSQLDAAlloc(). */
    if (select_dp->F < 0)
    {
```

```
    printf ("¥nToo many select-list items (%d), maximum is %d¥n",
            -(select_dp->F), MAX_ITEMS);
    return;
}

/* Set the maximum number of array elements in the
   descriptor to the number found. */
select_dp->N = select_dp->F;

/* Allocate storage for each select-list item.

   SQLNumberPrecV6() is used to extract precision and scale
   from the length (select_dp->L[i]).

   sqlcolumnNullCheck() is used to reset the high-order bit of
   the datatype and to check whether the column
   is NOT NULL.

   CHAR    datatypes have length, but zero precision and
           scale.  The length is defined at CREATE time.

   NUMBER  datatypes have precision and scale only if
           defined at CREATE time.  If the column
           definition was just NUMBER, the precision
           and scale are zero, and you must allocate
           the required maximum length.

   DATE    datatypes return a length of 7 if the default
           format is used.  This should be increased to
           9 to store the actual date character string.
           If you use the TO_CHAR function, the maximum
           length could be 75, but will probably be less
           (you can see the effects of this in SQL*Plus).

   ROWID   datatype always returns a fixed length of 18 if
           coerced to CHAR.

   LONG and
   LONG RAW datatypes return a length of 0 (zero),
           so you need to set a maximum.  In this example,
           it is 240 characters.

   */

printf ("¥n");
for (i = 0; i < select_dp->F; i++)
{
    char title[MAX_VNAME_LEN];
    /* Turn off high-order bit of datatype (in this example,
       it does not matter if the column is NOT NULL). */
    SQLColumnNullCheck ((unsigned short *)&(select_dp->T[i]),
         (unsigned short *)&(select_dp->T[i]), &null_ok);

    switch (select_dp->T[i])
    {
        case  1 : /* CHAR datatype: no change in length
                     needed, except possibly for TO_CHAR
                     conversions (not handled here). */
            break;
        case  2 : /* NUMBER datatype: use SQLNumberPrecV6() to
```

```
                    extract precision and scale. */
          SQLNumberPrecV6( SQL_SINGLE_RCTX,
                (unsigned long *)&(select_dp->L[i]), &precision, &scale);
                /* Allow for maximum size of NUMBER. */
          if (precision == 0) precision = 40;
                /* Also allow for decimal point and
                   possible sign. */
          /* convert NUMBER datatype to FLOAT if scale > 0,
             INT otherwise. */
          if (scale > 0)
              select_dp->L[i] = sizeof(float);
          else
              select_dp->L[i] = sizeof(int);
          break;

      case  8 : /* LONG datatype */
          select_dp->L[i] = 240;
          break;

      case 11 : /* ROWID datatype */
          select_dp->L[i] = 18;
          break;

      case 12 : /* DATE datatype */
          select_dp->L[i] = 9;
          break;

      case 23 : /* RAW datatype */
          break;

      case 24 : /* LONG RAW datatype */
          select_dp->L[i] = 240;
          break;
  }
  /* Allocate space for the select-list data values.
     SQLSQLDAAlloc() reserves a pointer location for
     V[i] but does not allocate the full space for
     the pointer.  */

   if (select_dp->T[i] != 2)
     select_dp->V[i] = (char *) realloc(select_dp->V[i],
                               select_dp->L[i] + 1);
   else
     select_dp->V[i] = (char *) realloc(select_dp->V[i],
                               select_dp->L[i]);

  /* Print column headings, right-justifying number
      column headings. */

  /* Copy to temporary buffer in case name is null-terminated */
  memset(title, ' ', MAX_VNAME_LEN);
  strncpy(title, select_dp->S[i], select_dp->C[i]);
  if (select_dp->T[i] == 2)
     if (scale > 0)
       printf ("%.*s ", select_dp->L[i]+3, title);
     else
       printf ("%.*s ", select_dp->L[i], title);
  else
    printf("%-.*s ", select_dp->L[i], title);
```

```
        /* Coerce ALL datatypes except for LONG RAW and NUMBER to
           character. */
        if (select_dp->T[i] != 24 && select_dp->T[i] != 2)
            select_dp->T[i] = 1;

        /* Coerce the datatypes of NUMBERs to float or int depending on
           the scale. */
        if (select_dp->T[i] == 2)
          if (scale > 0)
            select_dp->T[i] = 4;  /* float */
          else
            select_dp->T[i] = 3;  /* int */
    }
    printf ("¥n¥n");

    /* FETCH each row selected and print the column values. */
    EXEC SQL WHENEVER NOT FOUND GOTO end_select_loop;

    for (;;)
    {
        EXEC SQL FETCH C USING DESCRIPTOR select_dp;

        /* Since each variable returned has been coerced to a
           character string, int, or float very little processing
           is required here.  This routine just prints out the
           values on the terminal. */
        for (i = 0; i < select_dp->F; i++)
        {
            if (*select_dp->I[i] < 0)
                if (select_dp->T[i] == 4)
                  printf ("%-*c ",(int)select_dp->L[i]+3, ' ');
                else
                  printf ("%-*c ",(int)select_dp->L[i], ' ');
            else
                if (select_dp->T[i] == 3)     /* int datatype */
                  printf ("%*d ", (int)select_dp->L[i],
                                  *(int *)select_dp->V[i]);
                else if (select_dp->T[i] == 4)     /* float datatype */
                  printf ("%*.2f ", (int)select_dp->L[i],
                                  *(float *)select_dp->V[i]);
                else                          /* character string */
                  printf ("%-*.*s ", (int)select_dp->L[i],
                            (int)select_dp->L[i], select_dp->V[i]);
        }
        printf ("¥n");
    }
end_select_loop:
    return;
}



void help()
{
    puts("¥n¥nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
    puts("Statements can be continued over several lines, except");
    puts("within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
    puts("Terminate a PL/SQL block (which can contain embedded semicolons)");
    puts("with a slash (/).");
```

```
    puts("Typing ¥"exit¥" (no semicolon needed) exits the program.");
    puts("You typed ¥"?¥" or ¥"help¥" to get this message.¥n¥n");
}


void sql_error()
{
    /* ORACLE error handler */
    printf ("¥n¥n%.70s¥n",sqlca.sqlerrm.sqlerrmc);
    if (parse_flag)
        printf
        ("Parse error at character offset %d in SQL statement.¥n",
           sqlca.sqlerrd[4]);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp_continue, 1);
}
```

15.8 サンプル・プログラム: スクロール可能カーソルを使用する動的SQL方法4

次のデモ・プログラムには、Oracle動的SQL方法4を使用したスクロール可能カーソル機能が記述されています。このプログラムは、demoディレクトリのファイルscdemo1.pcとして、オンラインで使用可能です。

scdemo1.pc

```
/*
 * This demo program exhibits the scrollable cursor feature
 * used with oracle dynamic method 4. The scrollable cursor
 * feature can also be used with ANSI dynamic method 4.
 *
 * This program takes as argument the username/passwd. Once
 * logged in, it prompts for a select query. It then prompts
 * for the orientation and prints the results of the query.
 *
 * Before executing this example, make sure that the HR
 * schema exists.
 */

#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
#include <sqlda.h>
#include <string.h>
#include <ctype.h>

#include <sqlcpr.h>
#include <stdlib.h>

#include <setjmp.h>


#define MAX_SELECT_ITEMS    200
#define MAX_CHARS           20

/* Maximum size of a select-list item name */
#define MAX_NAME_SIZE       50
```

```
SQLDA   *selda;
SQLDA   *bind_des;
jmp_buf beginEnv;
jmp_buf loopEnv;

/* Data buffer */
char c_data[MAX_SELECT_ITEMS][MAX_CHARS];

char username[60];
char stmt[500];
char stmt2[500];

/* Print the generic error message & exit */

void sql_error()
{
    char msgbuf[512];
    size_t msgbuf_len, msg_len;

    msgbuf_len = sizeof(msgbuf);
    sqlglm(msgbuf, &msgbuf_len, &msg_len);

    printf ("¥n¥n%.*s¥n", msg_len, msgbuf);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
}

/* Print the error message and continue to query
   the user */
void sql_loop_error()
{
    char msgbuf[512];
    size_t msgbuf_len, msg_len;
    int code = sqlca.sqlcode;

    msgbuf_len = sizeof(msgbuf);
    sqlglm(msgbuf, &msgbuf_len, &msg_len);

    printf ("¥n%.*s¥n", msg_len, msgbuf);
    printf("The error code is %d¥n", sqlca.sqlcode);
    if(code!=0)
      longjmp(beginEnv, 1);

    longjmp(loopEnv, 1);
}

/* FETCH has returned the "no data found"  error code.
   This means that either we have reached the end of
   the active set or the offset refers to a row beyond the
   active set */
void no_data_found()
{
  printf("¥nNo Data available at the specified offset¥n");
  longjmp(loopEnv, 1);
}

void main(int argc, char *argv[])
```

```
{
    int i, n;
    int sli;     /* select-list item */
    int offset;
    int contFlag;
    char bindVar[20];
    char *u, temp[3];
    char choice;

    /* Error Handler  */
    EXEC SQL WHENEVER SQLERROR DO sql_error();


    if (argc == 1)
    {
      printf("Logging in as default user hr¥n");
      strcpy(username, "hr/hr");
    }
    else
      strcpy(username, argv[1]);

    /* Establish a connection to the data base */
    EXEC SQL CONNECT :username;

    u = username;
    while(*++u != '/');
    *u = '¥0';

    /* Error Handler */
    EXEC SQL WHENEVER SQLERROR DO sql_loop_error();
    for (;;)
    {

      setjmp(beginEnv);
      printf("[%s] SQL > ", username);
      gets(stmt);
      if (!strlen(stmt))
        continue;
      for (i=0 ; i < strlen(stmt) ; i++)
           stmt2[i] = tolower(stmt[i]) ;
      stmt2[i]=0 ;
      if(!strcmp(stmt2, "exit"))
        break;

      selda = SQLSQLDAAlloc(SQL_SINGLE_RCTX, MAX_SELECT_ITEMS, MAX_NAME_SIZE, 0);
      bind_des = SQLSQLDAAlloc(SQL_SINGLE_RCTX, MAX_SELECT_ITEMS,
                               MAX_NAME_SIZE, 30);

      /* prepare a sql statement for the query*/
      EXEC SQL PREPARE S FROM :stmt;

      /* Declare a cursor as scrollable */
      EXEC SQL DECLARE C SCROLL CURSOR FOR S;

      for (i=0; i<MAX_SELECT_ITEMS; i++)
      {
        bind_des->I[i] = (short *) malloc(sizeof (short));
        bind_des->V[i] = (char *) malloc(1);
      }
      bind_des->N = MAX_SELECT_ITEMS;
```

```
EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_des;

/* set up the bind variables */
if (bind_des->F < 0)
{
  printf("Bind descriptor, value exceeds the limit¥n");
  exit(-1);
}

bind_des->N = bind_des->F;
for (i=0; i<bind_des->F; i++)
{
  printf("Enter the value for bind variable %.*s: ",
         (int)bind_des->C[i], bind_des->S[i]);

  fgets(bindVar, sizeof(bindVar), stdin);
  n = strlen(bindVar) - 1;

  bind_des->L[i] = n;
  bind_des->V[i] = (char *) realloc(bind_des->V[i],
                   (bind_des->L[i] +1));

  strncpy(bind_des->V[i], bindVar, n);
  if ((strncmp(bind_des->V[i], "NULL", 4) == 0) ||
        (strncmp(bind_des->V[i], "null", 4) == 0))
    *bind_des ->I[i] = -1;
  else
    *bind_des ->I[i] = 0;

  bind_des->T[i] = 1;
}

/* open the cursor */
EXEC SQL OPEN C USING DESCRIPTOR bind_des;
EXEC SQL DESCRIBE SELECT LIST FOR S INTO selda;

if (selda->F < 0)
{
  printf("Select descriptor, value exceeds the limit¥n");
  exit(-1);
}

selda->N = selda->F;
for (sli = 0; sli < selda->N; sli++)
{
    /* Set addresses of heads of the arrays
       in the V element. */
    selda->V[sli] = c_data[sli];
    /* Convert everything to varchar on output. */
    selda->T[sli] = 1;
    /* Set the maximum lengths. */
    selda->L[sli] = MAX_CHARS;
}

contFlag = 'Y';
setjmp(loopEnv);

while(1)
{
```

```
        while(contFlag != 'Y' && contFlag != 'N')
        {
        printf("¥nContinue with the current fetch? [y/n] :");
        contFlag = toupper(getchar());
        /* To flush the input buffer */
        getchar();
        }

        if(contFlag != 'Y')
          break;

        contFlag = 'x';

        printf("¥n¥nEnter the row number to be fetched ¥n");
        printf("1.ABSOLUTE¥n");
        printf("2.RELATIVE¥n");
        printf("3.FIRST ¥n");
        printf("4.NEXT ¥n");
        printf("5.PREVIOUS ¥n");
        printf("6.LAST ¥n");
        printf("7.CURRENT ¥n");
        printf("Enter your choice --> ");
        scanf("%c",&choice);

        /* To flush the input buffer */        getchar();        EXEC SQL WHENEVER NOT FOUND DO
no_data_found();
        switch(choice)
        {
         case '1': printf("¥nEnter Offset :");
                   scanf("%d",&offset);
                   getchar();
                   EXEC SQL FETCH ABSOLUTE :offset C USING DESCRIPTOR selda;
                   break;
         case '2': printf("¥nEnter Offset :");
                   scanf("%d",&offset);
                   getchar();
                   EXEC SQL FETCH RELATIVE :offset C USING DESCRIPTOR selda;
                   break;
         case '3': EXEC SQL FETCH FIRST C USING DESCRIPTOR selda;
                   break;
         case '4': EXEC SQL FETCH NEXT C USING DESCRIPTOR selda;
                   break;
         case '5': EXEC SQL FETCH PRIOR C USING DESCRIPTOR selda;
                   break;
         case '6': EXEC SQL FETCH LAST C USING DESCRIPTOR selda;
                   break;
         case '7': EXEC SQL FETCH CURRENT C USING DESCRIPTOR selda;
                   break;
         default : printf("Invalid choice¥n");
                   contFlag = 'Y';
                   continue;
        }

        /* print the row */
        for(sli=0; sli<selda->N; sli++)
          printf("%.20s ", c_data[sli]);

        puts("");

      }
```

```
        EXEC SQL CLOSE C;
    }

    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_SUCCESS);
}
```

16 LOB

この章では、LOB(ラージ・オブジェクト)データ型の埋込みSQL文で提供されるサポートについて説明します。

4種類のLOB型を説明し、従来のLONGおよびLONG RAWデータ型と比較します。

Oracle Call Interface APIおよびPL/SQL言語と同様の機能を提供するPro*C/C++の埋込みSQLインタフェースを示します。

LOB文およびそのLOBオプションとホスト変数を説明します。

最後に、使用方法を簡単に示すLOBインタフェースを使用したPro*C/C++ プログラムの例をあげています。

この章のトピックは、次のとおりです:

- LOB
- プログラムでのLOBの使用方法
- LOB文のルール
- LOB文
- LOBおよびナビゲーショナル・アクセス用インタフェース
- LOBプログラムの例

16.1 LOB

LOB(ラージ・オブジェクト)列を使用するとASCIIテキスト、各国語キャラクタのテキスト、様々なグラフィック形式のファイルおよびサウンド波形などの大量のデータ(最大4GB)を格納できます。

16.1.1 内部LOB

内部LOB(BLOB、CLOB、NCLOB)はデータベース表領域に格納され、データベース・サーバーからトランザクション・サポート(コミット、ロールバックなどの処理)が有効です。

バイナリ・ラージ・オブジェクト(BLOB)には、ビデオ・クリップなどの非構造化バイナリ・データ(生データとも呼ばれます)が格納されます。

キャラクタ・ラージ・オブジェクト(CLOB)には、データベース・キャラクタ・セットの文字データの大きいブロックが格納されます。

各国語キャラクタ・ラージ・オブジェクト(NCLOB)には、各国語キャラクタ・セットの文字データの大きいブロックが格納されます。

16.1.2 外部LOB

外部LOBは、データベース表領域外のオペレーティング・システム・ファイルです。ただし、データベース・サーバーのトランザクション・サポートは無効です。

BFILE(Binary Files)では、データは外部バイナリ・ファイルの形式で格納されます。BFILEには、GIF、JPEG、MPEG、MPEG2、テキストなどの形式があります。

16.1.3 BFILEのセキュリティ

DIRECTORYオブジェクトは、BFILEにアクセスして操作するときに使用します。DIRECTORYは、ファイルが格納されているサーバー・ファイル・システムの実際の物理ディレクトリの(サーバーに格納されている)論理的な別名です。DIRECTORYオブ

ジェクトに対するアクセス権限が割り当てられているユーザー以外は、ファイルにアクセスできません。

- データ定義言語(DDL)のSQL文であるCREATE、REPLACE、ALTERおよびDROPは、DIRECTORYデータベース・オブジェクトに使用します。
- DIRECTORYオブジェクト上のシステムおよびオブジェクトに対するREAD権限のGRANTおよびREVOKEを行うデータ操作言語(DML)SQL文。

CREATE DIRECTORYディレクティブの例を次に示します。

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir' ;
```

ユーザーまたはロールは、GRANTなどのデータ操作言語(DML)文の権限が割り当てられている場合にのみ、ディレクトリを読み込むことができます。たとえば、ユーザーscottがディレクトリ/usr/home/mydirのBFILESを読み込めるようにするには次のようにします。

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott ;
```

1セッション内で、最大10個のBFILESを同時にオープンできます。SESSION_MAX_OPEN_FILESパラメータの設定を変更して、デフォルト値を変更できます。

関連項目

- Oracle Database開発ガイド
- *GRANT*

16.1.4 LOBとLONGおよびLONG RAWの対比

LOBは、従来のLONGおよびLONG RAWデータ型と多くの点で異なります。

- LOBの最大サイズは4GBです。LONGおよびLONG RAWの最大サイズは2GBです。
- LOBでは、ランダム・アクセス方法および順次アクセス方法を使用できます。LONGおよびLONG RAWでは、順次アクセス方法のみです。
- LOB (NCLOBは除きます)は、定義したオブジェクト型の属性になります。
- 表では複数のLOB列を作成できますが、複数のLONGまたはLONG RAW列は作成できません。

既存のLONGおよびLONG RAW属性は、LOBに移行することをお薦めします。今後のリリースでは、LONGおよびLONG RAWはサポートしない予定です。移行の詳細は、*『Oracle® Database移行ガイド』*を参照してください。

16.1.5 LOBロケータ

LOBロケータは、LOBの実際の内容を指しています。ロケータはLOBの内容ではなくLOBを取り出したときに戻されます。LOBロケータは、特定のトランザクションまたはセッションには保存されずに、後続のトランザクションまたはセッションで再使用されます。

16.1.6 一時LOB

ローカル変数のように使用できるテンポラリLOBを作成すると、データベースLOBが使用しやすくなります。テンポラリLOBは、表には対応付けられません。アクセスできるのは作成者のみです。また、ロケータを持っており(ロケータを使用してアクセスします)、セッション終了時には削除されます。

一時BFILEはサポートされていません。INSERT、UPDATEまたはDELETE文のWHERE句でのみ、テンポラリLOBを入力変数(IN値)として使用できます。テンポラリLOBは、INSERT文で挿入される値またはUPDATE文のSET句の値としても使用できます。テンポラリLOBではデータベース・サーバーのトランザクション・サポートが無効なため、COMMITまたはROLLBACK

を行うことはできません。

一時LOBロケータは、トランザクションをまたがって使用することができます。サーバーが異常終了したとき、およびデータベースSQL処理でエラーが発生したときは削除されます。

16.1.7 LOBバッファリング・サブシステム

LBS(LOBバッファリング・サブシステム)は、クライアント側のアドレス空間に、1つ以上のLOBのバッファとして提供されるユーザー・メモリーです。

バッファリングには次の利点があります。特に、LOBの特定領域に小規模な読取りおよび書込みを繰り返すクライアントのアプリケーションに有効です。

- LBSを使用すると、LOBに対して読込み/書込みが複数回行われ、バッファがいっぱいになってから、FLUSHディレクティブが実行されるときにサーバーに書き込まれるため、サーバー・ラウンドトリップが減少します。
- サーバー上でのLOBの合計更新数が減少します。この結果、LOBのパフォーマンスが向上し、ディスク領域が節約されます。

Oracleで提供しているバッファリングは、簡単なバッファ・サブシステムで、キャッシュではありません。バッファの内容が、サーバーのLOB値と常に同期しているとはかぎりません。サーバーのLOBに実際に更新を書き込むには、FLUSH文を使用します。

LOBのバッファへの読込み/書込みは、ロケータを使用して行われます。バッファリングが使用可能なロケータを使用すると、書込みを実行するまで、LOBを常に読み込むことができます。

ロケータは、WRITEのバッファに使用された後で更新され、バッファリング・サブシステムを介して表示できる最新のLOBに対するアクセス権限が割り当てられます。LOBに対するその後のWRITEは、この更新ロケータを介してのみバッファされます。バッファされたLOBの操作を行うトランザクションは、ユーザー・セッション間では移行できません。

LBSは、サーバーのLOB値をFLUSH文を使用して更新するユーザーが管理します。これは、シングル・ユーザーでシングル・スレッドです。サーバーLOBの妥当性を確保するには、ROLLBACKおよびSAVEPOINTアクションを使用します。バッファされたLOB操作のトランザクション・サポートは無効です。バッファされたLOB更新のトランザクション・セマンティクスを確保するには、エラー発生時にロールバックを行う論理セーブポイントをメンテナンスする必要があります。

16.2 プログラムでのLOBの使用方法

この項では、Pro*C/C++アプリケーションでのLOBの使用に関する、プログラミングの重要な問題をいくつか説明します。

16.2.1 LOBにアクセスする3種類の方法

Pro*C/C++のLOBにアクセスするには、次の3種類の方法があります。

- PL/SQLブロックのDBMS_LOBパッケージ。
- OCI(Oracle Call Interface)ファンクション・コール。
- 埋込みSQL文。

SQL文は、PL/SQLインタフェースと同等の機能を提供するように設計されており、OCIインタフェースほど複雑ではありません。

次の表では、Pro*C/C++内でのOCIファンクション・コール、PL/SQLおよびPro*C/C++の埋込みSQL文によるLOBアクセスを比較しています。空欄は機能がないことを示します。

表16-1 LOBアクセス方法

OCI (1)	PL/SQL(2)	Pro*C/C++埋込みSQL

OCI (1)	PL/SQL(2)	Pro*C/C++埋込みSQL
-	COMPARE()	-
-	INSTR()	-
-	SUBSTR()	-
OCILobAppend	APPEND()	APPEND
OCILobAssign	:=	ASSIGN
OCILobCharSetForm	-	-
OCICharSetId	-	-
OCILobClose	CLOSE()	CLOSE
OCILobCopy	COPY()	COPY
OCILobCreateTemporary	CREATETEMPORARY()	CREATE TEMPORARY
OCILobDisableBuffering	-	DISABLE BUFFERING
OCILobEnableBuffering	-	ENABLE BUFFERING
OCILobErase	ERASE()	ERASE
OCILobGetChunkSize	GETCHUNKSIZE()	DESCRIBE
OCILobIsOpen	ISOPEN()	DESCRIBE
OCILobFileClose	FILECLOSE()	CLOSE
OCILobFileCloseAll	FILECLOSEALL()	FILE CLOSE ALL
OCILobFileExists	FILEEXISTS()	DESCRIBE
OCILobFileGetName	FILEGETNAME()	DESCRIBE
OCILobFileIsOpen	FILEISOPEN()	DESCRIBE

OCI (1)	PL/SQL(2)	Pro*C/C++埋込みSQL
OCILobFileOpen	FILEOPEN()	OPEN
OCILobFileSetName	BFILENAME()	FILE SET 脚注 3
OCILobFlushBuffer	-	FLUSH BUFFER
OCILobFreeTemporary	FREETEMPORARY()	FREE TEMPORARY
OCILobGetLength	GETLENGTH()	DESCRIBE
OCILobIsEqual	=	-
OCILobIsTemporary	ISTEMPORARY()	DESCRIBE
OCILobLoadFromFile	LOADFROMFILE()	LOAD FROM FILE
OCILobLocatorIsInit	-	-
OCILobOpen	OPEN()	OPEN
OCILobRead	READ()	READ
OCILobTrim	TRIM()	TRIM
OCILobWrite	WRITE()	WRITE
OCILobWriteAppend	WRITEAPPEND()	WRITE

脚注1 C/C++ユーザーのみ。これらの関数のプロトタイプは、ociap.hにあります。

脚注2 dbmslob.sqlを参照してください。BFILENAMEを除くルーチンの前には、すべて'DBMS_LOB'を付ける必要があります。

脚注3

組込みSQL関数BFILENAME()も使用できます。

注意:

新しい文を使用する前に、LOB に対して修正または変更する行を明示的にロックする必要があります。LOB 値を修正する操作には、APPEND、COPY、ERASE、LOAD FROM FILE、TRIM および WRITE があります。

16.2.2 アプリケーションでのLOBロケータ

LOBロケータをPro*C/C++アプリケーションで使用する場合は、oci.hヘッダー・ファイルをインクルードし、BLOBに対してOCIBlobLocator型のポインタ、CLOBおよびNCLOBに対してOCIClobLocator、またはBFILEに対してOCIBFileLocatorを宣言します。

NCLOBの場合は、次のいずれかの操作を行う必要があります。

- C/C++宣言で'CHARACTER SET IS NCHAR_CS'句を使用します。
- コマンドラインまたは構成ファイルで、NLS_CHARプリコンパイラ・オプションをあらかじめ指定し、NLS_NCHAR環境変数を設定する必要があります。

設定方法は次のとおりです。

```
/* In your precompiler program */
#include <oci.h>
...
OCIClobLocator CHARACTER SET IS NCHAR_CS *a_nclob ;
```

または、Pro*C/C++をコールするときに、プリコンパイラ・オプションNLS_CHARを

```
NLS_CHAR=(a_nclob)
```

と設定している場合は、コードから、CHARACTER SET句を削除できます。

```
#include <oci.h>
...
OCIClobLocator *a_nclob ;
```

他に次のように簡潔に宣言します。

```
/* In your precompiler program */
#include <oci.h>
...
OCIBlobLocator  *a_blob ;
OCIClobLocator  *a_clob ;
OCIBFileLocator *a_bfile ;
```

関連項目

- NLS_CHAR

16.2.3 LOBの初期化

LOBの初期化方法は、LOBの種類によって異なります。この項では、種類ごとに初期化方法を説明します。

16.2.3.1 内部LOB

BLOBを初期化して空にするには、EMPTY_BLOB()関数を使用するか、ALLOCATE SQL文を使用します。CLOBおよびNCLOBの場合は、EMPTY_CLOB()関数を使用します。EMPTY_BLOB()およびEMPTY_CLOB()の詳細は、『*Oracle Database SQL言語リファレンス*』を参照してください。

これらの関数は、INSERT文のVALUES句内またはUPDATE文のSET句のソースでのみ使用できます。

次に例を示します。

```
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
   VALUES (EMPTY_BLOB(), EMPTY_CLOB()) ;
```

ALLOCATE文を実行すると、LOBロケータが割り当てられ、初期化後に空になります。つまり、次のコードを実行しても、前の例と同じ結果が得られます。

```
#include <oci.h>
...
OCIBlobLocator *blob ;
OCIClobLocator *clob ;
EXEC SQL ALLOCATE :blob ;
EXEC SQL ALLOCATE :clob ;
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
   VALUES (:blob, :clob) ;
```

16.2.3.2 外部LOB

BFILEおよびFILENAMEのDIRECTORY別名を初期化するには、LOB FILE SET文を次のように使用します。

```
#include <oci.h>
...
char *alias = "lob_dir" ;
char *filename = "image.gif" ;
OCIBFileLocator *bfile ;
EXEC SQL ALLOCATE :bfile ;
EXEC SQL LOB FILE SET :bfile
   DIRECTORY = :alias, FILENAME = :filename ;
EXEC SQL INSERT INTO file_table (a_bfile) VALUES (:bfile) ;
```

DIRECTORYオブジェクトのネーミング規則およびDIRECTORYオブジェクト権限の詳細は、『*Oracle Database*アドバンスト・アプリケーション開発者ガイド』を参照してください。

また、INSERTまたはUPDATE文のBFILENAME('ディレクトリ', 'ファイル名')機能を使用して、BFILE列または特定行の属性を初期化し、実際の物理ディレクトリおよびファイル名を取得できます。

```
EXEC SQL INSERT INTO file_table (a_bfile)
   VALUES (BFILENAME('lob_dir', 'image.gif'))
      RETURNING a_bfile INTO :bfile ;
```

> 注意:
>
> BFILENAME()では、ディレクトリまたはファイル名の権限、および物理ディレクトリの有無は確認されません。BFILE ロケータを使用してファイルにアクセスしたときに、これらが確認され、ファイルにアクセスできない場合にはエラーが戻されます。

16.2.3.3 一時LOB

埋込みSQLのLOB CREATE TEMPORARY文を使用して最初に一時LOBを作成したときに、一時LOBは初期化され空になります。EMPTY_BLOB()およびEMPTY_CLOB()関数は、一時LOBでは使用できません。

16.2.3.4 LOBの解放

FREE文は、ALLOCATE文によって確保されたメモリーを解放するときに使用します。

```
EXEC SQL FREE :a_blob;
```

16.3 LOB文のルール

LOB文を使用するときのルールを説明します。

16.3.1 すべてのLOB文に適用されるルール

次の一般的な制限および制約は、SQL LOB文を使用してLOBを操作するときに適用されます。

- EXEC SQL LOB文で使用できるLOBロケータは1つのみのため、EXEC SQL LOB文ではFOR句は使用できません。
- 分散LOBはサポートされていません。新しい埋込みSQL LOB文ではATデータベース句を使用できますが、同一のSQL LOB文で、異なるデータベース接続を使用して作成またはALLOCATEされたLOBロケータを混在させることはできません。
- LOB READおよびWRITE操作を行う場合、OCIでは、コールバック関数をクライアントから指定できるコールバック・メカニズムを提供しています。このコールバック関数は、LOB値ピースの読込みあるいは書込みが行われるたびに実行されます。埋込みSQL LOB文では、この機能はサポートされていません。
- OCIでは、テンポラリLOBを作成するときに使用可能な期間を、独自に作成または指定できるメカニズムを提供しています。テンポラリLOBのREADおよびWRITE操作に使用されるバッファ・キャッシュを指定するメカニズムもあります。このインタフェースでは、これらの機能はサポートされていません。

16.3.2 LOBバッファリング・サブシステムに適用されるルール

LBSでは、次のルールに従う必要があります。

- 読取りまたは書込みアクセス時のエラーは、次回のサーバーへのアクセス時にレポートされます。このため、エラー・リカバリのコードは、ユーザーが作成する必要があります。
- バッファへの書込みによりLOBを更新するときは、必ずLOBバッファリング・サブシステムを経由して更新を行ってください。
- バッファリングが可能な更新済LOBロケータは、INパラメータとしてPL/SQLプロシージャに渡せますが、IN OUTあるいはOUTパラメータとしては渡すことはできません。エラーが戻されます。更新されたロケータを戻そうとしたときも、エラーが戻されます。
- バッファリング可能な更新済ロケータを、別のロケータにはASSIGNできません。
- バッファへの書込みをLOB値に追加することができますが、その開始オフセットの1文字は、LOBの最後に続く必要があります。LBSでは、データベース・サーバーのLOBに、0バイトの充填文字または空白が入るAPPEND文は使用できません。
- ホスト・ロケータのバインド変数のキャラクタ・セットとデータベース・サーバーのCLOBは、同じであることが必要です。
- ASSIGN、READおよびWRITE文のみ、バッファリング可能なロケータとともに使用できます。
- バッファリングが使用可能なロケータで、APPEND、COPY、ERASE、DESCRIBE(LENGTHのみ)およびTRIM文を実行するとエラーが発生します。また、これらの文をバッファリング不可能なロケータとともに使用した場合でも、そのロケータによってポイントされたLOBが他のロケータによってバッファ・モードでアクセスされた場合には、エラーが戻されます。

注意:

次の処理の前に、LOB バッファリング・サブシステムが使用可能な LOB に対して、FLUSH 文を実行する必要があります。

- トランザクションをコミットするとき。
- カレント・トランザクションから他のトランザクションへの移行。
- LOB上でのバッファ操作の使用禁止。
- 外部プロシージャの実行からPL/SQLルーチンに戻るとき。

注意:

PL/SQL ブロックからロケータ・パラメータを引数にして外部コールアウトがコールされた場合、ENABLE 文を含むすべてのバッファリングは、外部プロシージャ内で行う必要があります。

次の手順に従います。

- 外部コールアウトを呼び出します。
- バッファリングのロケータをENABLEします。
- ロケータを使用してREADまたはWRITEします。
- LOBに対してFLUSHします(LOBを暗黙的にフラッシュできません)。
- バッファリングのロケータを使用禁止にします。
- PL/SQLのファンクション/プロシージャ/メソッドに戻ります。

LOBは、明示的にFLUSHする必要があります。

16.3.3 ホスト変数に対するルール

LOB文では、次のルールおよび注意事項に従ってください。

- *src*および*dst*を使用して、内部または外部LOBロケータを参照できますが、*file*では、外部ロケータ以外は参照できません。
- 数値のホスト値(amt、src_offset、dst_offsetなど)は、4バイトの符号なし整数の変数として宣言されます。値は、0から4GBに制限されています。
- NULLは、LOBロケータで使用します。LOB文では標識変数は必要ありません。NULLは、数値変数(amt、src_offsetなど)には使用できません。エラーが発生します。
- オフセット値src_offsetおよびdst_offsetのデフォルト値は1です。

16.4 LOB文

アルファベット順に文を説明します。*database*は、すべての文でデータベース接続を表しています。

16.4.1 APPEND

用途

この文はLOB値を別のLOBの最後に追加します。

構文

```
EXEC SQL [AT [:]database] LOB APPEND :src TO :dst ;
```

ホスト変数

src (IN):

一意にソースLOBを参照する内部LOBロケータです。

dst (IN OUT)

一意に宛先LOBを参照する内部LOBロケータです。

使用上の注意

ソースLOBのデータが宛先LOBの最後にコピーされると、宛先LOBが最大4GBまで拡張されます。LOBが4GBを超えて拡張される場合は、エラーが発生します。

ソースおよび宛先LOBは、すでに存在している必要があります。また、宛先LOBは初期化されている必要があります。

ソースおよび宛先LOBの両方が、同じ内部LOB型であることが必要です。どちらのロケータに対しても、LOBバッファリングを有効にすると、エラーとなります。

16.4.2 ASSIGN

用途

LOBまたはBFILEロケータを、別のロケータに割り当てます。

構文

```
EXEC SQL [AT [:]database] LOB ASSIGN :src to :dst ;
```

ホスト変数

src (IN):

コピー元のLOBまたはBFILEロケータ・ソース。

dst (IN OUT)

コピー先のLOBまたはBFILEロケータ。

使用上の注意

割当て後は、両方のロケータは同じLOB値を参照します。宛先LOBロケータは、初期化された有効な(ALLOCATEで割り当てられた)ロケータにしてください。

内部LOBでは、宛先ロケータが表に格納されている場合にのみ、ソース・ロケータのLOB値が宛先ロケータのLOB値にコピーされます。Pro*C/C++の場合は、宛先ロケータを含むオブジェクトに対してFLUSHを発行すると、LOB値がコピーされます。

BFILEロケータが内部LOBロケータに割り当てられている場合、またはその逆の場合には、エラーが戻されます。*src* LOBと*dst* LOBが同じ型でない場合にもエラーになります。

バッファリングが使用可能な内部LOBに対するソース・ロケータの場合、そのソース・ロケータがLOBバッファリング・サブシステム経

由でLOB値を修正するために使用され、WRITE後にバッファに対してFLUSHしていないときは、ソース・ロケータを宛先ロケータに割り当てることはできません。これは、LOBバッファリング・サブシステムを介してLOB値を修正する場合、1つのLOBに対してロケータは1つしか使用できないためです。

16.4.3 CLOSE (LOBの場合)

用途

オープンされているLOBまたはBFILEをクローズします。

構文

```
EXEC SQL [AT [:]database] LOB CLOSE :src ;
```

ホスト変数

src (IN OUT)

クローズされるLOBまたはBFILEのロケータ。

使用上の注意

異なるロケータまたは同一ロケータを使用した場合でも、同一LOBを2回クローズするとエラーになります。外部LOBの場合は、BFILEが存在して一度もオープンされていない状態であれば、エラーは発生しません。

オープンしていたLOBをすべてクローズする前に、トランザクションをCOMMITするとエラーになります。トランザクションのROLLBACK時にオープンしているLOBは、クローズされず、すべて破棄されます。

16.4.4 COPY

用途

LOB値の全部または一部を別のLOBにコピーします。

構文

```
EXEC SQL [AT [:]database] LOB COPY :amt FROM :src [AT :src_offset]
   TO :dst [AT :dst_offset] ;
```

ホスト変数

amt (IN)

コピーするBLOBの最大バイト数またはCLOBおよびNCLOBの最大文字数。

src (IN):

ソースLOBのロケータ。

src_offset (IN)

CLOBまたはNCLOBの場合は、文字数。BLOBの場合は、バイト数。LOBの先頭で1から始まります。

dst (IN)

宛先LOBのロケータ。

dst_offset (IN)

宛先オフセット。src_offsetと同じルールが適用されます。

使用上の注意

データが宛先のオフセット以降にあらかじめ存在する場合は、ソース・データで上書きされます。宛先のオフセットがカレント・データの最後を超えている場合は、宛先LOBのカレント・データの最後から新しく書き込まれたソース・データの先頭まで、ゼロバイト充填文字(BLOB)または空白(CLOB)が書き込まれます。

新規に書き込むデータが宛先LOBの現行の長さよりも大きい場合、宛先LOBは、そのデータにあわせて拡張されます。4GBを超えてLOBが拡張されると、ランタイム・エラーが発生します。

初期化されていないLOBからコピーすると、エラーになります。

ソースLOBおよび宛先LOBは、同じ型である必要があります。LOBバッファリングは、ロケータのどちらに対しても使用可能にしないでください。

amt変数は、コピーの最大量です。指定した量がコピーされる前にソースLOBの最後に到達した場合、操作はORA-22993エラーで終了します。

テンポラリLOBを永続LOBにするには、COPY文を使用して、テンポラリLOBを永続LOBに明示的にCOPYする必要があります。

16.4.5 CREATE TEMPORARY

用途

一時LOBを作成します。

構文

```
EXEC SQL [AT [:]database] LOB CREATE TEMPORARY :src ;
```

ホスト変数

src (IN OUT)

実行前はINで、srcは、以前にALLOCATEされたLOBロケータです。

実行後でOUTになったときは、srcは新しい空の一時LOBをポイントするLOBロケータです。

使用上の注意

実行が正常に終了すると、ロケータはデータベース・サーバーに新しく作成された、表に依存しない一時LOBをポイントします。一時LOBは空で、長さゼロです。

セッション終了時に、すべての一時LOBは解放されます。テンポラリLOBに対するREADおよびWRITEでは、バッファ・キャッシュは経由されません。

16.4.6 DISABLE BUFFERING

用途

LOBロケータのLOBバッファリングを使用禁止にします。

構文

```
EXEC SQL [AT [:]database] LOB DISABLE BUFFERING :src ;
```

ホスト変数

src (IN OUT)

内部LOBロケータ。

使用上の注意

この文は、BFILEをサポートしていません。後続の読取りまたは書込みは、LBS経由では行われません。

注意:

この文では、LOB バッファリング・サブシステムの変更は暗黙的にフラッシュされないため、変更を有効にするには FLUSH BUFFER コマンドを使用します。

16.4.7 ENABLE BUFFERING

用途

LOBロケータのLOBバッファリングを使用可能にします。

構文

```
EXEC SQL [AT [:]database] LOB ENABLE BUFFERING :src ;
```

ホスト変数

src (IN OUT)

内部LOBロケータ。

使用上の注意

この文は、BFILEをサポートしていません。後続の読取りおよび書込みは、LBSを経由して行われます。

16.4.8 ERASE

用途

指定されたオフセットから始まる、指定された量のLOBデータを消去します。

構文

```
EXEC SQL [AT [:]database] LOB ERASE :amt FROM :src [AT :src_offset] ;
```

ホスト変数

amt (IN OUT)

入力は、消去するバイト数または文字数です。戻される出力は、実際に消去されたバイト数または文字数です。

src (IN OUT)

内部LOBロケータ。

src_offset (IN)

LOBの先頭からのオフセット。1から始まります。

使用上の注意

この文は、BFILEをサポートしていません。

実行後に消去された実際の文字/バイト数がamtから戻されます。要求した文字数またはバイト数を消去する前にLOB値の最

後に到達した場合は、実際の消去数と要求した消去数は異なります。LOBが空の場合は、amtには、0文字/バイトが消去されたことを示します。

BLOBの場合は、消去とはゼロバイト充填文字で既存のLOB値を上書きすることです。CLOBの場合は、空白で既存のLOB値を上書きすることです。

16.4.9 FILE CLOSE ALL

用途

カレント・セッションでオープンしているBFILESをすべてクローズします。

構文

```
EXEC SQL [AT [:]database] LOB FILE CLOSE ALL ;
```

使用上の注意

クローズ処理が正常に終了しなかったためにセッションにオープンしているファイルが存在する場合は、FILE CLOSE ALL文を使用して、セッション内でオープンしているファイルをすべてクローズし、最初からファイル操作を再開できます。

16.4.10 FILE SET

用途

BFILEロケータのDIRECTORY別名およびFILENAMEを設定します。

構文

```
EXEC SQL [AT [:]database] LOB FILE SET :file
    DIRECTORY = :alias, FILENAME = :filename ;
```

ホスト変数

file (IN OUT)

DIRECTORY別名およびFILENAMEが設定されているBFILEロケータ。

alias (IN)

設定するDIRECTORY別名。

filename (IN)

設定するFILENAME。

使用上の注意

指定したBFILEロケータは、この文で使用する前に、ALLOCATEされている必要があります。

DIRECTORY別名およびFILENAMEは必須項目です。

DIRECTORY別名の最大長は128バイトです。FILENAMEの最大長は255バイトです。

DIRECTORY別名およびFILENAME属性は、CHARZ、STRING、VARCHAR、VARCHAR2およびCHARF以外の外部データ型ではサポートされません。

この文を外部LOBロケータ以外で使用すると、エラーになります。

16.4.11 FLUSH BUFFER

用途

LOBのバッファをデータベース・サーバーに書き込みます。

構文

```
EXEC SQL [AT [:]database] LOB FLUSH BUFFER :src [FREE] ;
```

ホスト変数

src (IN OUT)

内部LOBロケータ。

使用上の注意

入力ロケータが参照するLOBからサーバーのデータベースLOBに、バッファ・データを書き込みます。

LOBバッファリングは、入力LOBロケータに対してすでに有効になっている必要があります。

デフォルトでのFLUSH操作では、バッファ・リソースの解放および別のバッファされたLOB操作への再割当ては行われません。ただし、バッファを明示的に解放する場合は、オプションのFREEキーワードを使用して指定できます。

16.4.12 FREE TEMPORARY

用途

LOBロケータ用に一時領域を解放します。

構文

```
EXEC SQL [AT [:]database] LOB FREE TEMPORARY :src ;
```

ホスト変数

src (IN OUT)

テンポラリLOBをポイントしているLOBロケータ。

使用上の注意

入力ロケータは、一時LOBをポイントしている必要があります。出力ロケータは、初期化されずに、後続のLOB文で使用されます。

16.4.13 LOAD FROM FILE

用途

BFILEの一部または全部を、内部LOBにコピーします。

構文

```
EXEC SQL [AT [:]database] LOB LOAD :amt FROM FILE :file [AT :src_offset] INTO
    :dst [AT :dst_offset] ;
```

ホスト変数

amt (IN)

ロードされる最大バイト数。

file (IN OUT)

BFILEロケータのソース。

src_offset (IN)

ファイルの先頭からのオフセットのバイト数。1から始まります。

dst (IN OUT)

宛先LOBロケータ。BLOB、CLOBまたはNCLOBになります。

dst_offset (IN)

書込みが開始される宛先LOBの先頭からのバイト数(BLOBの場合)または文字数(CLOBおよびNCLOBの場合)。1から始まります。

使用上の注意

データは、ソースBFILEから宛先内部LOBにコピーされます。BFILEデータをCLOBまたはNCLOBにコピーする場合、キャラクタ・セットは変換されません。このため、BFILEデータは、あらかじめデータベース内のCLOBまたはNCLOBと同じキャラクタ・セットになっている必要があります。

ソースおよび宛先LOBは、すでに存在している必要があります。宛先の開始位置にすでにデータがある場合は、ソース・データで上書きされます。宛先の開始位置が現行のデータの最後を超える場合は、ゼロバイトの充填文字(BLOB)または空白(CLOBおよびNCLOB)が宛先LOBに書き込まれます。充填文字は、宛先LOBのデータの最後から、ソースから新しく書き込まれたデータの始まりまでに書き込まれます。

新規に書き込むデータが宛先LOBの現行の長さよりも大きい場合、宛先LOBは、そのデータにあわせて拡張されます。4GBを超えてLOBが拡張されると、エラーになります。

また、初期化されていないBFILEからコピーをすると、エラーになります。

amountパラメータは、ロードする最大量を示します。指定した量がロードされる前にソースBFILEの最後に到達した場合は、処理はORA-22993エラーで終了します。

16.4.14 OPEN (LOBの場合)

用途

読取り、または読取り/書込みアクセスのために、LOBまたはBFILEをオープンします。

構文

```
EXEC SQL [AT [:]database] LOB OPEN :src [ READ ONLY | READ WRITE ] ;
```

ホスト変数

src (IN OUT)

LOBまたはBFILEのLOBロケータ。

使用上の注意

LOBまたはBFILEをOPENできるデフォルト・モードは、READ ONLYアクセスです。

内部LOBの場合は、OPENはロケータではなくLOBに対応付けられます。すでにOPENされているロケータを別のロケータに割り当てても、新しいLOBをOPENしたとはみなされません。両方のロケータから同じLOBが参照されます。BFILEの場合は、OPENはロケータに対応付けられます。

同時にOPENできるLOBの最大数は32個です。33個目のLOBをOPENするとエラーが戻されます。

書込み可能なBFILEは、サポートしていません。このため、BFILEをREAD WRITEモードでOPENすると、エラーが戻されます。

LOBをREAD ONLYモードでオープンした後にWRITEすると、エラーになります。

16.4.15 READ

用途

LOBまたはBFILEの一部または全部をバッファに読み込みます。

構文

```
EXEC SQL [AT [:]database] LOB READ :amt FROM :src [AT :src_offset]
   INTO :buffer [WITH LENGTH :buflen] ;
```

ホスト変数

amt (IN OUT)

入力は、読み込まれる文字数またはバイト数です。出力は、読み込まれた実際の文字数またはバイト数です。

読み込まれたバイト数がバッファ長より大きい場合は、LOBはポーリング・モードで読み込まれているとみなされます。入力時にこの値がゼロの場合は、データは入力オフセットからLOBの最後までポーリング・モードで読み込まれます。

実際に読み込まれるバイト数または文字数はamtで戻されます。データがピース単位で読み込まれる場合、amtには常に最後に読み込まれたピースが含まれます。

LOBの最後に到達した場合は、「ORA-01403: データが見つかりません。」というエラーが発生します。

ポーリング・モードで読み込むときは、アプリケーションからLOB READを繰り返しコールし、データがなくなるまでLOBのピースを読み込む必要があります。ORA-01403エラーを捕捉するには、WHENEVERディレクティブのNOT FOUND条件を使用してポーリング・モードの使用を制御します。

src (IN):

LOBまたはBFILEロケータ。

src_offset (IN)

読込みを開始する、LOB値の先頭からの絶対オフセットです。キャラクタLOBにおいては、これはLOBの先頭からの文字数を意味します。バイナリLOBまたはBFILEにおいては、これはバイト数を意味します。先頭位置は1です。

buffer (IN/OUT)

LOBデータが読み込まれるバッファ。バッファの外部データ型は、ソースLOBの型によって一定の型に制限されます。バッファの最大長は、LOB値を格納するために使用される外部データ型によって決まります。次の表では、有効な外部データ型および対応する最大長を、ソースLOB型単位に分類したものです。

表16-2 ソースLOBおよびプリコンパイラ・データ型

外部LOB(4)	内部LOB	プリコンパイラ外部データ型	プリコンパイラ最大長(5)	PL/SQLデータ型	PL/SQL最大長
BFILE	BLOB	RAW	65535	RAW	32767
		VARRAW	65533		

外部LOB(4)	内部LOB	プリコンパイラ外部データ型	プリコンパイラ最大長(5)	PL/SQLデータ型	PL/SQL最大長
		LONG RAW	2147483647		
		LONG VARRAW	2147483643		
BFILE	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
BFILE	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

脚注 4 これらの外部データ型は、BFILEで使用できます。

脚注 5 長さは、文字単位ではなく、バイト単位で計算されています。

buflen (IN)

他の方法で指定できないときは、指定したバッファ長が指定されます。

使用上の注意

BFILEはデータベース・サーバーにあらかじめ存在し、入力ロケータを使用してオープンされている必要があります。データベースにはファイルを読み込む権限が、またユーザーにはディレクトリの読取り権限が必要です。

初期化されていないLOBまたはBFILEからの読込みは、エラーとなります。

バッファの長さは、次のように決定されます。

- WITH LENGTH句を指定した場合の、buflenの値。
- WITH LENGTH句がない場合は、OUTモードのバッファ・ホスト変数の指定によって決定されます。

関連項目

- グローバリゼーション・サポート
- BLOBのREADおよびファイル書込みの例

16.4.16 TRIM

用途

LOB値を切り捨てます。

構文

```
EXEC SQL [AT [:]database] LOB TRIM :src TO :newlen ;
```

ホスト変数

src (IN OUT)

内部LOBのLOBロケータ。

newlen (IN)

LOB値の新しい長さ。

使用上の注意

この文はBFILEには使用できません。新規の長さは、現行の長さより大きくは設定できません。大きく設定した場合は、エラーが戻されます。

16.4.17 WRITE

用途

バッファの内容をLOBに書き込みます。

構文

```
EXEC SQL [AT [:]database] LOB WRITE [APPEND] [ FIRST | NEXT | LAST | ONE ]
    :amt FROM :buffer [WITH LENGTH :buflen] INTO :dst [AT :dst_offset] ;
```

ホスト変数

amt (IN OUT)

入力は、書き込まれる文字数またはバイト数です。

出力は、書き込まれた実際の文字数またはバイト数です。

ポーリング・メソッドを使用して書込みをした場合は、WRITE LAST 文の実行後に、WRITE 文実行時に書き込まれた累積合計長が amt から戻されます。WRITE 文が中断された場合は、amt は定義されません。

buffer (IN)

LOB データが書き込まれるバッファ。

dst (IN OUT)

LOB ロケータ。

dst_offset (IN)

CLOB および NCLOB の場合は文字数単位、BLOB の場合はバイト数単位の、LOB の先頭からのオフセット(1 から始まります)。

buflen (IN)

他の方法で計算できないときのバッファ長。

使用上の注意

LOBデータがすでに存在する場合は、バッファに格納されているデータで上書きされます。指定されたオフセットが、現在LOB内にあるデータの最後を超える場合は、ゼロバイト充填文字または空白がLOBに挿入されます。

WRITE文にキーワードAPPENDを指定すると、LOBの最後にデータが自動的に書き込まれます。APPENDを指定すると、宛先オフセットがLOBの最後とみなされます。WRITE文にAPPENDオプションを指定したときに、宛先オフセットを指定するとエラーになります。

バッファは、LOBに対して1ピースで書き込まれるか(デフォルトのONE方向変換を使用します)、標準のポーリング・メソッドを使用してピース単位で書き込むことができます。

FIRSTを使用してポーリングを開始し、NEXTで後続のピースを書き込みます。LASTキーワードは、書込みを終了する最後のピースの書込みに使用します。

このピース単位の書込みモードを使用すると、各ピースのサイズおよび場所が異なる場合に、バッファおよびバッファ長をコール単位に指定できます。

すべての書込みの終了後に渡される合計データ量が、amtパラメータで指定した量よりも少ない場合は、エラーになります。

同じルールが、READ文のバッファ長の決定にも適用されます。

関連項目

- READ
- BLOBのREADおよびファイル書込みの例

16.4.18 DESCRIBE

用途

この文は複数のOCIおよびPL/SQL文に相当します(このため最後に保存します)。LOB DESCRIBE SQL文を使用してLOBから属性を取り出します。この機能は、OCIおよびPL/SQLプロシージャに似ています。LOB DESCRIBE文の書式は次のとおりです。

構文

```
EXEC SQL [AT [:]database] LOB DESCRIBE :src GET attribute1 [{, attributeM}]
   INTO :hv1 [[INDICATOR] :hv_ind1] [{, :hvN [[INDICATOR] :hv_indN] }] ;
```

次の属性を指定できます。

```
CHUNKSIZE | DIRECTORY | FILEEXISTS | FILENAME | ISOPEN | ISTEMPORARY | LENGTH
```

ホスト変数

src (IN):

内部または外部LOBのLOBロケータ。

hv1 ... hvN ... (OUT)

属性値を受け取るホスト変数。属性名リストで指定した順に指定します。

hv_ind1 ... hv_indN ... (OUT)

インジケータNULL状態を受け取るオプションのホスト変数。属性名リストで指定した順に指定します。

この表では、属性、対応付けられるLOBおよび読み込まれるC言語のデータ型を説明します。

表16-3 LOB属性

LOB属性	属性の説明	制限事項	C言語のデータ型
CHUNKSIZE	LOB 値を格納する LOB チャンクに使用される領域の量(BLOB の場合はバイト数単位、CLOB または NCLOB の場合は文字数単位)。このチャンク・サイズの倍数で READ または WRITE 要求を発行すると、パフォーマンスが向上します。すべての WRITE を 1 つのチャンクで行うと、余分なバージョン化が実行されることも重複されることもありません。同一 CHUNK に対して WRITE コールを複数回発行するかわりに、1 つのチャンク内で WRITE コールを十分な数だけまとめることができます。	BLOB、CLOB および NCLOB のみ	unsigned int
DIRECTORY	BFILE の場合は、DIRECTORY 別名。最大長は 128 バイトです。	FILE LOB のみ	char * 脚注 6
FILEEXISTS	サーバーのオペレーティング・システムのファイル・システム上に、BFILE が存在するかどうかを決定します。ゼロ以外の場合は、FILEEXIST は真、ゼロの場合は、偽です。	FILE LOB のみ	signed int
FILENAME	BFILE の名前。最大長は 255 バイトです。	FILE LOB のみ	char*
ISOPEN	BFILE の場合は、入力 BFILE ロケータが OPEN 文で使用されなかった場合は、このロケータからは OPEN されていないものとみなされます。ただし、別の BFILE ロケータによって、BFILE が OPEN されていることもあります。複数のロケータを使用して、同一 BFILE 上で複数の OPEN を実行することもできます。LOB の場合は、別のロケータにより LOB が OPEN された場合も、その入力ロケータによって OPEN されているとみなされます。ゼロ以外の場合は、ISOPEN は真、ゼロの場合は、偽です。	-	signed int
ISTEMPORARY	入力 LOB ロケータが一時 LOB を参照するかどうかを決定します。ゼロ以外の場合は、ISTEMPORARY は真、ゼロの場合は、偽です。	BLOB、CLOB および NCLOB のみ	signed int
LENGTH	BLOB および BFILE の長さはバイト数単位、CLOB および NCLOB の長さは文字数単位で表されます。BFILE の場合、EOF が存在するときは、EOF も長さに含まれます。空の内部 LOB は、ゼロ長になります。初期化されていない LOB および BFILE の長さは、定義されません。	-	unsigned int

脚注6

DIRECTORY属性およびFILENAME属性の場合は、CHARZ、STRING、VARCHAR、VARCHAR2およびCHARF以外の外部データ型はサポートされません。

使用上の注意

標識変数は、short型で宣言します。実行が完了すると、sqlca.sqlerrd[2]にはエラーなしで取り出された複数の属性が戻されます。実行エラーが発生した場合、エラーが発生した属性は、sqlca.sqlerrd[2]の内容より1つ多くなります。

DESCRIBEの例

ここで、任意のBFILEからDIRECTORYおよびFILENAME属性を抽出する、簡単なPro*C/C++の例を示します。

次のOCIBFileLocator宣言で型の解決およびコンパイルを正しく行うには、oci.hヘッダー・ファイルが必要です。

```
#include <oci.h>
...
OCIBFileLocator *bfile ;
char directory[31], filename[256] ;
short d_ind, f_ind ;
```

最後に、LOB表からBFILEロケータを選択し、DESCRIBEを実行します。

```
EXEC SQL ALLOCATE :bfile ;
EXEC SQL SELECT a_bfile INTO :bfile FROM lob_table WHERE ... ;
EXEC SQL LOB DESCRIBE :bfile
   GET DIRECTORY, FILENAME INTO :directory:d_ind, :filename:f_ind ;
```

標識変数は、DIRECTORYおよびFILENAME属性で使用するときにのみ有効です。属性値の保存に使用されるホスト変数バッファの大きさが不足している場合は、これらの属性値の文字列が切り捨てられることがあります。切捨てが発生した場合は、インジケータの値には属性の元の長さが設定されます。

16.5 LOBおよびナビゲーショナル・アクセス用インタフェース

ナビゲーショナル・アクセス用インタフェースは、LOBを属性として含むオブジェクト型の操作に使用することもできます。

関連項目

- オブジェクトへのナビゲーショナル・アクセス

16.5.1 一時オブジェクト

OBJECT CREATE文を使用して、LOB属性を持つ一時および永続オブジェクトを作成します。テンポラリLOBを一時オブジェクトのLOB属性にASSIGNし、永続LOBまたは永続オブジェクトのLOB属性に値をコピーしてデータを保存します。または、テンポラリLOBをLOB属性にASSIGNし、FLUSHを使用してデータベースに値を書き込みます。

BFILE属性の一時オブジェクトを作成して、ディスク上のBFILEからデータを読み込むことができます。テンポラリBFILEはサポートされていません。

16.5.2 永続オブジェクト

内部LOB属性が格納されたオブジェクト・キャッシュに永続オブジェクトを作成すると、LOB属性は暗黙的に空に設定されます。まず、OBJECT FLUSH文を使用してこのオブジェクトをフラッシュし、表に行を挿入して空のLOBを作成する必要があります。オブジェクト・キャッシュのオブジェクトを(VERSION=LATESTオプションを使用して)リフレッシュすると、実際のロケータが属性に読み込まれます。

BFILE属性のオブジェクトを作成すると、BFILEはNULLに設定されます。BFILEを読み込む前に、有効なディレクトリ別名およびファイル名で更新する必要があります。

テンポラリLOBは、永続オブジェクトのLOB属性にASSIGNされることがあります。オブジェクトがフラッシュされると、実際のLOB値がコピーされます。COPY文でテンポラリLOBロケータおよびLOB属性のロケータを使用して、テンポラリLOBの値を永続オブジェクトのLOB属性に明示的にコピーすることもできます。

16.5.3 ナビゲーショナル・アクセス用インタフェースの例

ナビゲーショナル・アクセス用インタフェースでLOBを処理する場合は、OBJECT GETおよびSET文を使用します。

オブジェクト型の属性のLOBロケータを取り出し、新しい埋込みSQL LOB文で使用できます。OBJECT SET文を使用して、LOBロケータをオブジェクト型の属性に戻します。

この場合、直接LOB ASSIGN操作を実行した場合と同じ結果になります。型の変更など、LOB ASSIGNが実行された場合に適用されるオブジェクト型に対して、LOB属性のOBJECT GETまたはSETを実行した場合にもこのルールが適用されます。

たとえば、次の簡単な型の定義を仮定します。

```
CREATE TYPE lob_type AS OBJECT (a_blob BLOB) ;
```

この例では、この型を有効な(初期化済の)BLOB属性を持つデータベースの列とみなします。

Pro*C/C++で使用できるOTT生成のC言語の構造体は、次のようになります。

```
struct lob_type
{
   OCIBlobLocator *a_blob ;
} ;
typedef struct lob_type lob_type ;
```

Pro*C/C++プログラムを作成して、DESCRIBE文でBLOB属性を抽出し、BLOBの現行の長さを取り出します。次に、TRIMでBLOBのサイズを半分に調整し、SET OBJECTで属性を元に戻してから、OBJECT FLUSHで変更を有効にします。

まず、oci.hをインクルードし、一部のローカル変数を宣言します。

```
#include <oci.h>
lob_type *lob_type_p ;
OCIBlobLocator *blob = (OCIBlobLocator *)0 ;
unsigned int length ;
```

オブジェクトからBLOB属性を選択し、OBJECT GETおよびDESCRIBEを行ってBLOBの現行の長さを取得します。

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL SELECT a_column
   INTO :lob_type_p FROM a_table WHERE ... FOR UPDATE ;
EXEC SQL OBJECT GET a_blob FROM :lob_type_p INTO :blob ;
EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :length ;
```

長さを半分にし、BLOBを新しい長さにTRIMします。

```
length = (unsigned int)(length / 2) ;
EXEC SQL LOB TRIM :blob TO :length ;
```

BLOBが変更されると、BLOB属性をオブジェクトに戻し、変更をサーバーにFLUSHし、コミットします。

```
EXEC SQL OBJECT SET a_blob OF :lob_type_p TO :blob ;
EXEC SQL OBJECT FLUSH :lob_type_p ;
EXEC SQL FREE :blob ;
```

```
EXEC SQL COMMIT WORK ;
```

関連項目:

OTTのINTYPEファイルの作成方法とOTTの実行方法については、Object Type Translator(OTT)を参照してください。

16.6 LOBプログラムの例

BFILEおよびBLOBの読込みおよび書込みの方法について、2つの例をあげます。

16.6.1 BLOBのREADおよびファイル書込みの例

この例では、長さが不明な任意の長さのBLOBからデータをバッファに読み込み、バッファから外部ファイルにそのデータを書き込みます。バッファが小さいため、読み込むBLOBのサイズに応じて、1つのREAD文でBLOB値をバッファに読み込める場合もありますが、標準ポーリング・モードを使用する必要がある場合もあります。

まず、oci.hおよび簡単なローカル変数を宣言します。

```
#include <oci.h>
OCIBlobLocator *blob ;
FILE *fp ;
unsigned int amt, offset = 1 ;
```

BLOB値を格納し、ファイルに書き込むバッファが必要です。

```
#define MAXBUFLEN 5000
unsigned char buffer[MAXBUFLEN] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLEN) ;
```

BLOBホスト変数を割り当て、READするBLOBを選択します。

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL SELECT a_blob INTO :blob FROM lob_table WHERE ... ;
```

BLOB値を書き込む外部ファイルをオープンします。

```
fp = fopen((const char *)"image.gif", (const char *)"w") ;
```

1回のREADですべてのLOB値をバッファに読み込める場合は、シグナルLOB READの終了に対してNOT FOUND条件を取得する必要があります。

```
EXEC SQL WHENEVER NOT FOUND GOTO end_of_lob ;
```

最初のREADを行います。量パラメータは、最大値の4GBに設定します。バッファより大きいため、LOBを読み込めない場合は、ポーリング・モードを使用してREADします。

```
amt = 4294967295 ;
EXEC SQL LOB READ :amt FROM :blob AT :offset INTO :buffer ;
```

この例の場合、バッファの大きさが不足しているためLOB値をすべて格納できないので、読込み済のデータはバイナリI/Oを使用して書込みおよび読込みを続行します。

```
(void) fwrite((void *)buffer, (size_t)MAXBUFLEN, (size_t)1, fp) ;
```

標準ポーリング・モードを使用して、無限ループ内でLOB READによって読込みを続行します。ループを終了するには、NOT FOUND条件を設定します。

```
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
  {
```

ポーリング中はオフセットが使用されないため、後続のLOB READでは省略できます。ただし、最後のREADのコールでREADされた量が通知されるようにするため、量パラメータを指定します。

```
    EXEC SQL LOB READ :amt FROM :blob INTO :buffer ;
    (void) fwrite((void *)buffer, (size_t)MAXBUFLEN, (size_t)1, fp) ;
  }
```

LOB値の最後に到達しました。量パラメータには、READされた最後のピースの量が保存されます。ポーリング中は、中間の各ピースの量が、MAXBUFLEN、つまりバッファの最大サイズに設定されます。

```
end_of_lob:
(void) fwrite((void *)buffer, (size_t)amt, (size_t)1, fp) ;
```

この基本的な構造のコードでは、任意の長さの内部LOBがローカル・バッファにREADされ、外部ファイルに書き込まれます。OCIおよびPL/SQLをモデルにしています。

16.6.2 ファイルの読込みおよびBLOBのWRITEの例

この例では、長さが明確な任意の長さのファイルからデータをバッファに読み込み、バッファからデータを内部BLOBに書き込みます。バッファが小さいため、読み込むファイルのサイズに応じて、1つのWRITE文でファイル・データをLOBに書き込める場合もありますが、標準ポーリング・モードを使用する必要がある場合もあります。

まず、oci.hおよび簡単なローカル変数を宣言します。

```
#include <oci.h>
OCIBlobLocator *blob ;
FILE *fp ;
unsigned int amt, offset = 1 ;
unsigned filelen, remainder, nbytes ;
boolean last ;
```

ファイル・データを格納し、LOBに書き込むバッファが必要です。

```
#define MAXBUFLEN 5000
unsigned char buffer[MAXBUFLEN] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLEN) ;
```

空の表の空のBLOBを初期化して、ALLOCATEされたロケータにそのBLOBを取り出し、ファイルからデータをコピーします。

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL INSERT INTO lob_table (a_blob) VALUES (EMPTY_BLOB())
   RETURNING a_blob INTO :blob ;
```

バイナリ・ファイルをオープンして長さを決定します。BLOBに書き込む合計量が、バイナリ・ファイルの実際の長さになります。

```
fp = fopen((const char *)"image.gif", (const char *)"r") ;
(void) fseek(fp, 0L, SEEK_END) ;
filelen = (unsigned int)ftell(fp) ;
amt = filelen ;
```

バッファ・サイズに基づいて読み込むバイト数を決定し、ファイルの初期読込みを設定します。

```
if (filelen > MAXBUFLEN)
    nbytes = MAXBUFLEN ;
else
    nbytes = filelen ;
```

ファイルI/O操作を発行してnバイトのデータをファイルfpからバッファに読み込み、残りの読込み量を決定します。ファイルの先頭から読込みを開始します。

```
(void) fseek(fp, 0L, SEEK_SET) ;
(void) fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) ;
remainder = filelen - nbytes ;
```

残りの読込み量に応じて、1ピースでバッファに書き込むか、ポーリングを開始し、複数の小さなピース単位でファイルのデータを書き込むポーリングを開始します。

```
     if (remainder == 0)
     {
```

この場合は、1ピースでデータを書き込みます。

```
        EXEC SQL LOB WRITE ONE :amt
           FROM :buffer INTO :blob AT :offset ;
     }
     else
      {
```

ポーリング・メソッドを開始し、ピース単位でデータをLOBに書き込みます。ポーリング・メソッドを開始するには、まず、最初のWRITEでFIRSTキーワードを使用します。

```
        EXEC SQL LOB WRITE FIRST :amt
           FROM :buffer INTO :blob AT :offset ;
```

簡単なループを設定し、ポーリング・モードを実装します。

```
        last = FALSE ;
        EXEC SQL WHENEVER SQLERROR DO break ;
        do
          {
```

ファイルから読み込み、宛先LOBにWRITEするバイト数を計算します。また、読み込んだピースがLASTピースかどうかを判断します。

```
            if (remainder > MAXBUFLEN)
                nbytes = MAXBUFLEN ;
            else
                {
                    nbytes = remainder ;
                    last = TRUE ;
                }
```

ファイル・システムのファイルから次のnbytesをバッファに読み込みます。ファイル読込み中にエラーが発生した場合は、自動的に次のWRITEがLASTになるように設定します。

```
            if  fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) != 1)
               last = TRUE ;
```

ここで、LASTピースをWRITEするか、中間のNEXTピースをWRITEします。NEXTピースは、ファイルから読み込まれるデータが残っていることを示しています。

```
          if (last)
            {
              EXEC SQL LOB WRITE LAST :amt
                 FROM :buffer INTO :blob  ;
            }

          else
            {
              EXEC SQL LOB WRITE NEXT :amt
                 FROM :buffer INTO :blob  ;
            }
          remainder -= nbytes ;
         }
while (!last && !feof(fp)) ;
```

このコード例では、任意の長さのファイルをローカル・バッファに読み込み、LOBに書き込みます。これは、OCIの例をモデルにしています。

16.6.3 lobdemo1.pc

このプログラムlobdemo1.pcは、LOB埋込みSQL文の例です。ソース・コードは、demoディレクトリにあります。このアプリケーションでは、社会保障番号列、名前列および交通違反の集計が格納されているCLOB列で構成されるlicense_tableという表を使用します。標準的な自動車部門の簡単なSQL操作をモデルにしています。

次の操作を指定できます。

- 新しいレコードを追加します。
- 社会保障番号別のレコード・リストを出力します。
- 特定の社会保障番号で指定して、レコード情報のリストを出力します。
- 既存のCLOBの内容に新しい交通違反を追加します。

```
/*****************************************************************************

  SCENARIO:

  We consider the example of a database used to store driver's
  licenses. The licenses are stored as rows of a table containing
  three columns: the sss number of a person, his name in text and the
  text summary of the info found in his license.

  The sss number is the driver's unique social security number.

  The name is the driver's given name as found on his ID card.

  The text summary is a summary of the information on the driver,
  including his driving record, which can be arbitrarily long and may
  contain comments and data regarding the person's driving ability.

  APPLICATION OVERVIEW:

  This example demonstrate how a Pro*C client can handle the new LOB
  datatypes through PL/SQL routines. Demonstrated are mechanisms for
  accessing and storing lobs to tables and manipulating LOBs through
  the stored procedures available through the dbms_lob package.

*****************************************************************************/
```

```
/*****************************************************************************

   To run the demo:

   1. Execute the script, lobdemo1c.sql in SQL*Plus
   2. Precompile using Pro*C/C++
        proc lobdemo1 user=scott/tiger sqlcheck=full
   3. Compile/Link (This step is platform specific)

*****************************************************************************/

/*** The following will be added to the creation script for this example ***
 *** This code can be found in lobdemo1c.sql                              ***

connect scott/tiger;

set serveroutput on;

Rem Make sure database has no license_table floating around

drop table license_table;

Rem ABSTRACTION:
Rem A license table reduces the notion of a driver's license into three
Rem distinct components - a unique social security number (sss),
Rem a name (name), and a text summary of miscellaneous information.

Rem IMPLEMENTATION:
Rem Our implementation follows this abstraction

create table license_table(
  sss char(9),
  name varchar2(50),
  txt_summary clob);

insert into license_table
        values('971517006', 'Dennis Kernighan',
        'Wearing a Bright Orange Shirt - 31 Oct 1996');

insert into license_table
        values('555001212', 'Eight H. Number',
        'Driving Under the Influence - 1 Jan 1997');

insert into license_table
        values('010101010', 'P. Doughboy',
        'Impersonating An Oracle Employee - 10 Jan 1997');

insert into license_table
        values('555377012', 'Calvin N. Hobbes',
        'Driving Under the Influence - 30 Nov 1996');

select count(*) from license_table;

Rem Commit to save
commit;

*****************************************************************************/

/*************************
 * Begin lobdemo1.pc code *
```

```
 *************************/

#define EX_SUCCESS       0
#define EX_FAILURE       1

#ifndef STDIO
# include <stdio.h>
#endif /* STDIO */

#ifndef SQLCA_ORACLE
# include <sqlca.h>
#endif /* SQLCA_ORACLE */

#ifndef OCI_ORACLE
# include <oci.h>
#endif /* OCI_ORACLE */

#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#ifndef LOBDEMO1_ORACLE
# include "lobdemo1.h"
#endif /* LOBDEMO1_ORACLE */

/***********
 * Defines *
 ***********/
#define SSS_LENGTH 12
#define NAME_LENGTH 50 /* corresponds with max length of name in table */
#define BUFLEN 1024
#define MAXCRIME 5
#define DATELENGTH 12

/***********
 * Globals *
 ***********/

char *CrimeList[MAXCRIME]={ "Driving Under the Influence",
                            "Grand Theft Auto",
                            "Driving Without a License",
                            "Impersonating an Oracle Employee",
                            "Wearing a Bright Orange Shirt" };

char curdate[DATELENGTH];

/************************
 * Function prototypes *
 ************************/

#if defined(__STDC__)
  void GetDate( void );
  void PrintSQLError( void );
  void Driver( void );
  void ListRecords( void );
  void PrintCrime( OCIClobLocator *a_clob );
  void GetRecord( void );
  void NewRecord( void );
  char *NewCrime( void );
```

```
  void GetName( char *name_holder );
  void AppendToClob( OCIClobLocator *a_clob, char *charbuf );
  void AddCrime( void );
  void ReadClob( OCIClobLocator *a_clob );
  boolean GetSSS( char *suggested_sss );
#else
  void GetDate();
  void PrintSQLError( );
  void Driver( );
  void ListRecords( );
  void PrintCrime(/* OCIClobLocator *a_clob */);
  void GetRecord( );
  void NewRecord( );
  char *NewCrime( );
  void GetName(/* char *name_holder */);
  void AppendToClob(/* OCIClobLocator *a_clob, char *charbuf */);
  void AddCrime();
  boolean GetSSS(/* char *suggested_sss */);
#endif

/*
 * NAME
 *   GetDate
 * DESCRIPTION
 *   Get date from user
 * LOB FEATURES
 *   none
 */

void GetDate()
{
  time_t now;

  now = time(NULL);
  strftime(curdate, 100, " - %d %b %Y", localtime(&now));
}

main()
{
  char * uid = "scott/tiger";

  EXEC SQL WHENEVER SQLERROR DO PrintSQLError();

  printf("Connecting to license database account: %s ¥n", uid);
  EXEC SQL CONNECT :uid;

  GetDate();

  printf("¥t**********************************¥n");
  printf("¥t* Welcome to the DMV Database *¥n");
  printf("¥t**********************************¥n¥n");
  printf("Today's Date is%s¥n", curdate);

  Driver();

  EXEC SQL COMMIT RELEASE;

  return (EX_SUCCESS);
}
```

```
/*
 * NAME
 *   Driver
 * DESCRIPTION
 *   Command Dispatch Routine
 * LOB FEATURES
 *   none
 */

void Driver()
{
  char choice[20];
  boolean done = FALSE;

  while (!done)
  {
    printf("¥nLicense Options:¥n");
    printf("¥t(L)ist available records by SSS number¥n");
    printf("¥t(G)et information on a particular record¥n");
    printf("¥t(A)dd crime to a record¥n");
    printf("¥t(I)nsert new record to database¥n");
    printf("¥t(Q)uit¥n");
    printf("Enter your choice: ");

    fgets(choice, 20, stdin);
    switch(toupper(choice[0]))
    {
    case 'L':
      ListRecords();
      break;
    case 'G':
      GetRecord();
      break;
    case 'A':
      AddCrime();
      break;
    case 'I':
      NewRecord();
      break;
    case 'Q':
      done = TRUE;
      break;
    default:
      break;
    }
  }
}

/*
 * NAME
 *   ListRecords
 * DESCRIPTION
 *   List available records by sss number
 * LOB FEATURES
 *   none
 */

void ListRecords()
{
  char *select_sss = "SELECT SSS FROM LICENSE_TABLE";
```

```
  char sss[10];

  EXEC SQL PREPARE sss_exec FROM :select_sss;
  EXEC SQL DECLARE sss_cursor CURSOR FOR sss_exec;
  EXEC SQL OPEN sss_cursor;

  printf("Available records:¥n");

  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      EXEC SQL FETCH sss_cursor INTO :sss;
      printf("¥t%s¥n", sss);
    }
  EXEC SQL WHENEVER NOT FOUND CONTINUE;

  EXEC SQL CLOSE sss_cursor;
}


/*
 * NAME
 *   PrintCrime
 * DESCRIPTION
 *   Tests correctness of clob
 * LOB FEATURES
 *   OCIlobRead and OCILobGetLength
 */

void PrintCrime(a_clob)
  OCIClobLocator *a_clob;
{
  ub4 lenp;

  printf("¥n");
  printf("====================¥n");
  printf(" CRIME SHEET SUMMARY ¥n");
  printf("====================¥n¥n");

  EXEC SQL LOB DESCRIBE :a_clob GET LENGTH INTO :lenp;

  if(lenp == 0) /* No crime on file */
    {
      printf("Record is clean¥n");
    }
  else
    {
      ub4 amt = lenp;
      varchar *the_string = (varchar *)malloc(2 + lenp);

      the_string->len = (ub2)lenp;

      EXEC SQL WHENEVER NOT FOUND CONTINUE;
      EXEC SQL LOB READ :amt
        FROM :a_clob INTO :the_string WITH LENGTH :lenp;

      printf("%.*s¥n", the_string->len, the_string->arr);
      free(the_string);
    }
}
```

```
/*
 * NAME
 *   GetRecord
 * DESCRIPTION
 *   Get license of single individual
 * LOB FEATURES
 *   allocate and select of blob and clob
 */

void GetRecord()
{
  char sss[SSS_LENGTH];

  if(GetSSS(sss) == TRUE)
    {
      OCIClobLocator *license_txt;
      char name[NAME_LENGTH]={'¥0'};

      EXEC SQL ALLOCATE :license_txt;

      EXEC SQL SELECT name, txt_summary INTO :name, :license_txt
        FROM license_table WHERE sss = :sss;

      printf("============================================================¥n¥n");
      printf("NAME: %s¥tSSS: %s¥n", name, sss);
      PrintCrime(license_txt);
      printf("¥n¥n============================================================¥n");

      EXEC SQL FREE :license_txt;
    }
  else
    {
      printf("SSS Number Not Found¥n");
    }
}

/*
 * NAME
 *   NewRecord
 * DESCRIPTION
 *   Create new record in database
 * LOB FEATURES
 *   EMPTY_CLOB() and OCILobWrite
 */

void NewRecord()
{
  char sss[SSS_LENGTH], name[NAME_LENGTH] = {'¥0'};

  if(GetSSS(sss) == TRUE)
    {
      printf("Record with that sss number already exists.¥n");
      return;
    }
  else
    {
      OCIClobLocator *license_txt;

      EXEC SQL ALLOCATE :license_txt;
```

```
      GetName(name);

      EXEC SQL INSERT INTO license_table
        VALUES (:sss, :name, empty_clob());

      EXEC SQL SELECT TXT_SUMMARY INTO :license_txt FROM LICENSE_TABLE
        WHERE SSS = :sss;

      printf("=========================================================¥n¥n");
      printf("NAME: %s¥tSSS: %s¥n", name, sss);
      PrintCrime(license_txt);
      printf("¥n¥n=========================================================¥n");

      EXEC SQL FREE :license_txt;
    }
}

/*
 * NAME
 *   NewCrime
 * DESCRIPTION
 *   Query user for new crime
 * LOB FEATURES
 *   None
 */

char *NewCrime()
{
  int  SuggestedCrimeNo;
  int  i;
  char crime[10];

  printf("Select from the following:¥n");
  for(i = 1; i <= MAXCRIME; i++)
    printf("(%d) %s¥n", i, CrimeList[i-1]);

  printf("Crime (1-5): ");
  fgets(crime, 10, stdin);
  SuggestedCrimeNo = atoi(crime);

  while((SuggestedCrimeNo < 1) || (SuggestedCrimeNo > MAXCRIME))
    {
      printf("Invalid selection¥n");
      printf("Crime (1-5): ");
      fgets(crime, 10, stdin);
      SuggestedCrimeNo = atoi(crime);
    }

  return CrimeList[SuggestedCrimeNo-1];
}

/*
 * NAME
 *   AppendToClob
 * DESCRIPTION
 *   Append String charbuf to a Clob in the following way:
 *   if the contents of the clob a_clob were <foo> and the
 *   contents of charbuf were <bar>, after the append a_clob
 *   will contain: <foo>¥n<bar> - <curdate>
```

```
 *   where <curdate> is today's date as obtained by the
 *   GetDate procedure.
 * LOB FEATURES
 *   OCILobWrite
 * NOTE
 *   Potentially, charbuf can be a very large string buffer.
 *   Furthermore, it should be noted that lobs and lob
 *   performance were designed for large data. Therefore,
 *   users are encouraged to read and write large chunks of
 *   data to lobs.
 */

void AppendToClob(a_clob, charbuf)
  OCIClobLocator *a_clob;
  char *charbuf;
{
  ub4 ClobLen, WriteAmt, Offset;
  int CharLen = strlen(charbuf);
  int NewCharbufLen = CharLen + DATELENGTH + 4;
  varchar *NewCharbuf;

  NewCharbuf = (varchar *)malloc(2 + NewCharbufLen);

  NewCharbuf->arr[0] = '¥n';
  NewCharbuf->arr[1] = '¥0';
  strcat((char *)NewCharbuf->arr, charbuf);
  NewCharbuf->arr[CharLen + 1] = '¥0';
  strcat((char *)NewCharbuf->arr, curdate);

  NewCharbuf->len = NewCharbufLen;

  EXEC SQL LOB DESCRIBE :a_clob GET LENGTH INTO :ClobLen;

  WriteAmt = NewCharbufLen;
  Offset = ClobLen + 1;

  EXEC SQL LOB WRITE ONE :WriteAmt FROM :NewCharbuf
    WITH LENGTH :NewCharbufLen INTO :a_clob AT :Offset;

  free(NewCharbuf);
}

/*
 * NAME
 *   AddCrime
 * DESCRIPTION
 *   Add a crime to a citizen's crime file
 * LOB FEATURES
 *   OCILobWrite
 */

void AddCrime()
{
  char sss[SSS_LENGTH];

  if (GetSSS(sss) == TRUE)
    {
      OCIClobLocator *license_txt;
      char *crimebuf;
      char  name[NAME_LENGTH] = {'¥0'};
```

```
      EXEC SQL ALLOCATE :license_txt;

      EXEC SQL SELECT txt_summary INTO :license_txt FROM license_table
        WHERE sss = :sss FOR UPDATE;

      crimebuf = NewCrime();

      printf("Added %s to CrimeList¥n", crimebuf);
      AppendToClob(license_txt, crimebuf);

      EXEC SQL SELECT name INTO :name FROM license_table WHERE sss = :sss;

      printf("NAME: %s SSS: %s¥n", name, sss);
      PrintCrime(license_txt);

      EXEC SQL COMMIT;
      EXEC SQL FREE :license_txt;
    }
  else
    {
      printf("SSS Number Not Found¥n");
    }
}

/*
 * NAME
 *   GetSSS
 * DESCRIPTION
 *   Fills the passed buffer with a client-supplied social security number
 *   Returns FALSE if sss does not correspond to any entry in the database,
 *   else returns TRUE
 * LOB FEATURES
 *   none
 */

boolean GetSSS(suggested_sss)
  char *suggested_sss;
{
  int count = 0;
  int i;

  printf("Social Security Number: ");
  fgets(suggested_sss, SSS_LENGTH, stdin);

  for(i = 0; ((suggested_sss[i] != '¥0') && (i < SSS_LENGTH)); i++)
    {
      if(suggested_sss[i] == '¥n')
        suggested_sss[i]='¥0';
    }

  EXEC SQL SELECT COUNT(*) INTO :count FROM license_table
    WHERE sss = :suggested_sss;

  return (count != 0);
}

/*
 * NAME
 *   GetName
```

```
 * DESCRIPTION
 *   Get name from user.
 *
 * LOB FEATURES
 *   none
 */

void GetName(name_holder)
  char *name_holder;
{
  int count=0;
  int i;

  printf("Enter Name: ");
  fgets(name_holder, NAME_LENGTH + 1, stdin);

  for(i = 0; name_holder[i] != '¥0'; i++)
    {
      if(name_holder[i] == '¥n')
        name_holder[i]='¥0';
    }

  return;
}

/*
 * NAME
 *   PrintSQLError
 * DESCRIPTION
 *   Prints an error message using info in sqlca and calls exit.
 * COLLECTION FEATURES
 *   none
 */

void PrintSQLError()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("SQL error occurred...¥n");
  printf("%.*s¥n", (int)sqlca.sqlerrm.sqlerrml,
          (CONST char *)sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK RELEASE;
  exit(EX_FAILURE);
}
```

17 オブジェクト

この章では、Pro*C/C++のユーザー定義オブジェクトのサポートについて説明します。この章のトピックは、次のとおりです:

- オブジェクトの概要
- Pro*C/C++でのオブジェクト型の使用について
- オブジェクト・キャッシュ
- 連想アクセス用インタフェース
- ナビゲーショナル・アクセス用インタフェース
- オブジェクト属性とC言語のデータ型の変換
- オブジェクト・オプションの設定/取得
- オブジェクトに対する新しいプリコンパイラ・オプション
- Pro*C/C++のオブジェクト例
- 型の継承のサンプル・コード
- ナビゲーショナル・アクセスのサンプル・コード
- C言語の構造体の使用について
- REFの使用について
- OCIDate、OCIString、OCINumberおよびOCIRawの使用について
- Pro*C/C++の新しいデータベース型の概要
- 動的SQLでのOracleデータ型使用の制限

17.1 オブジェクトの概要

Pro*C/C++では、Oracle8以降これまでサポートされていたOracleのリレーショナル・データ型に加えて、次のユーザー定義データ型がサポートされます。

- オブジェクト型
- オブジェクト型のREF
- コレクション・オブジェクト・タイプ
- 型の継承

関連項目

- コレクション

17.1.1 オブジェクト型

オブジェクト型は、ユーザー定義によるデータ型です。これには、CREATE TYPE SQL文で変数として定義されるデータ型の属性と、オブジェクト型に適用できる動作としての関数およびプロシージャからなるメソッドが含まれます。このマニュアルでは、属性のみを持つオブジェクト型を考えます。

次に例を示します。

```
--Defining an object type...
CREATE TYPE employee_type AS OBJECT(
    name    VARCHAR2(20),
    id      NUMBER,
    MEMBER FUNCTION get_id(name VARCHAR2) RETURN NUMBER);
/
--
--Creating an object table...
CREATE TABLE employees OF employee_type;
--Instantiating an object, using a constructor...
INSERT INTO employees VALUES (
        employee_type('JONES', 10042));
```

LONG、LONG RAW、NCLOB、NCHARおよびNCHAR可変幅データ型は、オブジェクト属性では使用できません。

17.1.2 オブジェクト型のREF

REF(「reference」の短縮形)はオブジェクト自体の参照ではなく、データベース表に格納されているオブジェクトを参照します。REF型は、リレーショナル列に指定できるのみでなく、オブジェクト型のデータ型としても指定できます。たとえば、次のように、表*employee_tab*にオブジェクト型*employee_t*自体のREFを表す列を組み込むことができます。

```
CREATE TYPE employee_t AS OBJECT(
   empname          CHAR(20),
   empno            INTEGER,
   manager          REF employee_t);
/
CREATE TABLE employee_tab OF employee_t;
```

17.1.3 型の継承

Oracleでは、オブジェクトの型の継承がサポートされます。これにより、類似するオブジェクト型の間で属性とメソッドを共有したり、オブジェクト型の特性を拡張したりできます。

Pro*C/C++では、次のSQL演算子によるオブジェクト型の継承がサポートされます。

- IS OF *type*
- TREAT

IS OF *type*演算子は、特定の*type*情報についてオブジェクト・インスタンスをテストするために使用します。

次のコード例では、pがEmployee_tおよびStudent_t型のすべてのpオブジェクトの参照が戻されます。

```
SELECT REF(p)
   FROM person_tab p
   WHERE VALUE(p) IS OF (Employee_t, Student_t);
```

次のコード例では、Student_t型のpのみを含むすべての行が戻されます。

```
SELECT VALUE(p)
   FROM person_tab p
   WHERE VALUE(p) IS OF (ONLY Student_t);
```

TREAT演算子は、式の宣言された型を変更するために使用します。

次のコード例では、Student_t型のpを含むすべての行が戻されます。pのうち、Student_t型でないすべてのインスタンスについては、NULLが戻されます。

```
SELECT TREAT(VALUE(p) AS Student_t)
```

```
   FROM person_tab p;
```

次のコード例では、Student_t型のすべてのpオブジェクトとサブタイプPartTimeStudent_tのオブジェクトの参照が戻されます。

```
SELECT TREAT(REP(p) AS REF Student_t)
   FROM person_tab p
   WHERE VALUE(p) IS OF (Student_t);
```

17.2 Pro*C/C++でのオブジェクト型の使用について

OTT(Object Type Translator)で生成されたC言語の構造体へのポインタを、Pro*C/C++アプリケーションでホスト変数と標識変数として宣言します。オブジェクト型の場合、標識変数はオプションですが、使用することをお薦めします。

Pro*C/C++プログラムでは、オブジェクト型を、OTTを使用してデータベース・オブジェクトから生成されたC言語の構造体として表現します。次の操作を必ず実行してください。

- OTTによって生成されるヘッダー・ファイルに、構造体定義およびそれに対応付けられたNULLインジケータ構造体、オブジェクト型のREFを表すC言語のデータ型を指定して、Pro*C/C++プログラムにインクルードします。
- OTT生成の型ファイルをPro*C/C++のINTYPEコマンドライン・オプションとして入力します。この型ファイルにより、OTTによって生成されるC言語の構造体とそれに対応するデータベース内のオブジェクト型の間およびスキーマと型のバージョン情報との間の対応関係がコード化されます。

関連項目

- Object Type Translator

17.2.1 NULLインジケータ

Object Type Translator(OTT)によって、オブジェクト型のNULLステータスを表すC言語の構造体が生成されます。生成されたこれらの構造体型を、オブジェクト型の標識変数の宣言で使用する必要があります。

その他のOracle型では、NULLインジケータに対して特別な処置は必要ありません。

オブジェクト型には内部構造があるため、オブジェクト型を表すNULLインジケータにも内部構造があります。コレクション・オブジェクト・タイプ以外のオブジェクト型を表すNULLインジケータ構造体は、オブジェクト型全体を表すアトミック(シングル)NULLステータスのみでなく、すべての属性のNULLステータスも提供します。OTTにより、オブジェクト型を表すNULLインジケータ構造体を示すC言語の構造体が生成されます。NULLインジケータ構造体の名前は、*Object_typename*_indです。*Object_typename*は、データベース内のユーザー定義型を表すC言語の構造体の名前です。

関連項目

- データ型とホスト変数

17.3 オブジェクト・キャッシュ

オブジェクト・キャッシュは、プログラムがデータベース・オブジェクトとのインタフェースに使用するために割り当てられたクライアントのメモリー領域です。オブジェクトには、2つのインタフェースが機能します。連想アクセス用インタフェースはオブジェクトの一時コピーを操作し、ナビゲーショナル・アクセス用インタフェースは永続オブジェクトを操作します。

17.3.1 永続オブジェクト対一時コピー

Pro*C/C++でEXEC SQL ALLOCATE文を使用してキャッシュに割り当てたオブジェクトは、Oracleデータベース内では永続オブジェクトの一時コピーになります。したがって、これらのコピーはフェッチした後にキャッシュ内で更新できますが、その変更を

データベース内で永続的なものにするには、明示的なSQLコマンドを使用する必要があります。この一時コピーまたは値ベースのオブジェクト・キャッシング・モデルはリレーショナル・モデルの拡張で、ここではリレーショナル表のスカラー列をホスト変数にフェッチすること、その場で更新することおよび更新結果をサーバーに通信することが可能です。

17.4 連想アクセス用インタフェース

連想アクセス用インタフェースは、オブジェクトの一時コピーを操作します。メモリーは、EXEC SQL ALLOCATE文を使用してオブジェクト・キャッシュ内で割り当てます。

SQLLIBランタイム・コンテキストごとに、オブジェクト・キャッシュが1つずつ作成されます。

各オブジェクトは、EXEC SQL SELECT文またはEXEC SQL FETCH文によって取り出されます。この2つの文では、ホスト変数の属性値が設定されます。NULLインジケータが与えられている場合は、それも設定されます。

オブジェクトの挿入、更新または削除には、EXEC SQL INSERT文、EXEC SQL UPDATE文およびEXEC SQL DELETE文を使用します。文が実行される前に、オブジェクトのホスト変数の属性を設定する必要があります。

トランザクション文EXEC SQL COMMITおよびEXEC SQL ROLLBACKは、変更をサーバーに永続的に書き込むときや、変更を取り消すときに使用します。

EXEC SQL FREE文を使用すると、オブジェクト・キャッシュ内のメモリーを明示的に解放できます。接続の終了時には、その割当て済メモリーが暗黙的に解放されます。

17.4.1 連想アクセス用インタフェースを使用する場合

次のような場合に使用します。

- 表の明示的な結合による処理上の負荷が大きくはないような、オブジェクトの大規模なコレクションにアクセスする場合。
- 参照できないオブジェクトにアクセスする場合。このようなオブジェクトには、識別性がありません。たとえば、リレーショナル列内のオブジェクト型などです。
- 一連のオブジェクトにUPDATEやINSERTなどの操作を適用する場合。たとえば、特定部門のすべての従業員に$1000のボーナスを追加する場合などです。

17.4.2 ALLOCATE

オブジェクト・キャッシュに領域を割り当てるには、次の文を使用します。次に構文を示します。

```
EXEC SQL [AT [:]database] ALLOCATE :host_ptr [[INDICATOR]:ind_ptr] ;
```

入力する変数は、次のとおりです。

database (IN)

データベース接続の名前を含むヌル文字で終了する文字列。データベース接続は次の文で事前に行われています。

```
EXEC SQL CONNECT :user [AT [:]database];
```

AT句のATを省略するか、*database*が空の文字列であれば、デフォルトのデータベース接続とみなされます。

host_ptr (IN)

オブジェクト型、コレクション・オブジェクト・タイプまたはREFに対してOTTにより生成されたホスト構造体へのポインタ、またはC言語のデータ型であるOCIDate、OCINumber、OCIRawまたはOCIStringのいずれかへのポインタ。

ind_ptr (IN)

標識変数*ind_ptr*とキーワードINDICATORはともにオプションです。構造体の型を持つインジケータへのポインタにかぎり、ALLOCATE文およびFREE文に指定できます。

*host_ptr*および*ind_ptr*には、ホスト配列を構成できます。

割当てはセッションが終了するまで有効です。どのインスタンスも、FREE文で明示的に解放されなくても、セッション(接続)が終了すると解放されます。

関連項目

- ALLOCATE (実行可能埋込みSQL拡張機能)
- FREE (実行可能埋込みSQL拡張機能)

17.4.3 FREE

```
EXEC SQL [AT[:]database] [OBJECT] FREE :host_ptr [[INDICATOR]:ind_ptr];
```

オブジェクト・キャッシュに格納されるオブジェクトの領域の割当てを解除するには、FREE文を使用します。この文に使用する変数は、ALLOCATE文の場合と同じです。

注意:

ホスト変数や標識変数へのポインタは、NULL には設定されません。

17.4.4 CACHE FREE ALL

```
EXEC SQL [AT [:]database] [OBJECT] CACHE FREE ALL;
```

前述の文を使用すると、指定したデータベース接続用のオブジェクト・キャッシュ・メモリーがすべて解放されます。

関連項目

- CACHE FREE ALL(実行可能埋込みSQL拡張機能)

17.4.5 連想アクセス用インタフェースを使用したオブジェクトへのアクセス

SQLを使用してオブジェクトにアクセスする場合、Pro*C/C++アプリケーションは永続オブジェクトの一時コピーを操作します。これは、SELECT、UPDATEおよびDELETEの各文を使用するリレーショナル・アクセス・インタフェースの直接の拡張です。

図17-1では、永続オブジェクトの一時コピーにALLOCATE文でキャッシュを割り当てます。割り当てられるオブジェクトにデータは含まれていませんが、OTTによって生成される構造体の形式をとります。

```
person *per_p;
...
EXEC SQL ALLOCATE :per_p;
```

SELECT文を実行してキャッシュに移入できます。また、FETCH文やC言語の代入を使用してキャッシュにデータを入れることもできます。

```
EXEC SQL SELECT ... INTO :per_p FROM person_tab WHERE ...
```

図のように、INSERT、UPDATEまたはDELETE文を使用して、サーバー・オブジェクトを変更します。データは、次のINSERT

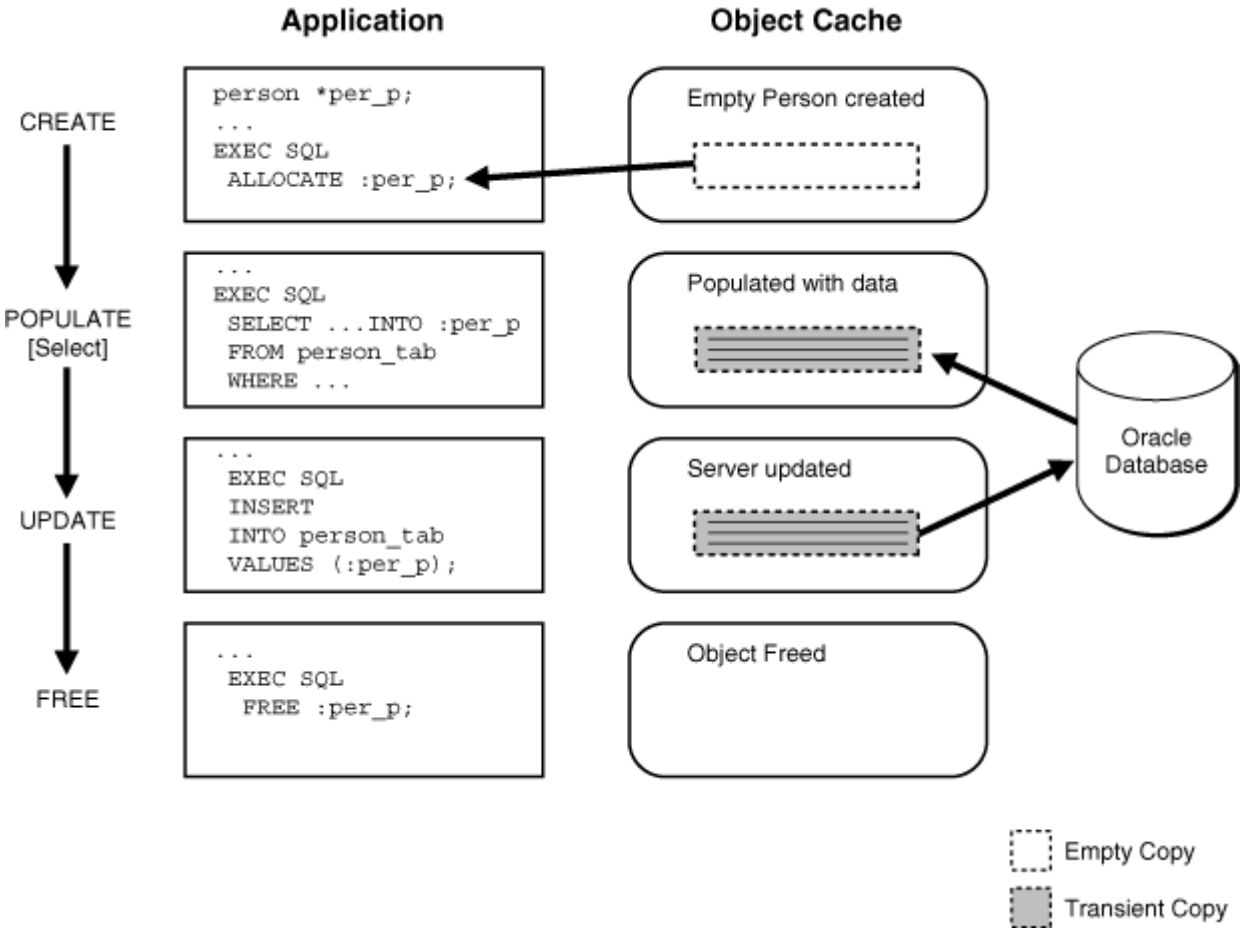
文を使用して表に挿入できます。

```
EXEC SQL INSERT INTO person_tab VALUES(:per_p);
```

最後に、FREE文を使用して、オブジェクトのコピーに対応するメモリーを解放します。

```
EXEC SQL FREE :per_p;
```

図17-1 SQLを使用したオブジェクトへのアクセス



17.5 ナビゲーショナル・アクセス用インタフェース

ナビゲーショナル・アクセス用インタフェースを使用すると、連想アクセス用インタフェースと同じスキーマにアクセスできます。ナビゲーショナル・アクセス用インタフェースはオブジェクトに対するREFを間接参照して、あるオブジェクトから他のオブジェクトへとたどる(ナビゲート)ことで(永続および一時)オブジェクトにアクセスします。次に、一部の定義を示します。

オブジェクトの確保とは、オブジェクトを間接参照し、プログラムがオブジェクトにアクセスできるようにするという意味の用語です。

オブジェクトの確保解除とは、オブジェクトが不要になったことをキャッシュに対して示すことです。

間接参照とは、サーバーがREFを使用してクライアント内にそのオブジェクトのある版を作成することであると定義できます。キャッシュ内では、各オブジェクトとそれに対応するサーバー・オブジェクトとの間の対応付けが保たれますが、自動的な一貫性は得られません。キャッシュ内の各オブジェクトの内容の正確さと一貫性を確認するのは、ユーザーの責任です。

オブジェクト・コピーのリリースとは、キャッシュに対してオブジェクトが現在使用されていないことを示すことです。メモリーを解放するために、不要になったオブジェクトをリリースして暗黙的にメモリーの解放ができるようにします。

オブジェクト・コピーを解放すると、オブジェクト・コピーはキャッシュから削除され、使用されていたメモリーが解放されます。

オブジェクトをマークすることで、キャッシュ内のオブジェクト・コピーが更新されており、オブジェクト・コピーをフラッシュするときに対応するサーバー・オブジェクトを更新する必要があることが、キャッシュに通知されます。

オブジェクトをマーク解除すると、オブジェクトが更新されたという指示が削除されます。

オブジェクトをフラッシュすると、キャッシュ内のマークされたコピーに対して行ったローカルな変更が、サーバー内の対応するオブジェクトに書き込まれます。この時点で、キャッシュ内のオブジェクト・コピーもマーク解除されます。

キャッシュ内のオブジェクト・コピーをリフレッシュすると、そのコピーは、サーバー内の対応するオブジェクトの最新値に置き換わります。

ナビゲーショナル・アクセス用インタフェースと連想アクセス用インタフェースは併用できます。

キャッシュ・コピーを更新、削除およびフラッシュする(キャッシュ内の変更をサーバーに書き込む)には、EXEC SQL OBJECT文、ナビゲーショナル・アクセス用インタフェースを使用します。

関連項目:

ナビゲーショナル・アクセス用インタフェースと連想アクセス用インタフェースの併用方法は、ナビゲーショナル・アクセスのサンプル・コードを参照してください。

17.5.1 ナビゲーショナル・アクセス用インタフェースを使用する場合

次の場合にナビゲーショナル・アクセス用インタフェースを使用します。

- 表の明示的な結合による処理上の負荷が大きい単一のオブジェクト、もしくは少数のオブジェクトへアクセスする場合。間接参照(DEREF)を使用してオブジェクト間でナビゲートする場合は、表全体の明示的な結合よりも処理負荷の少ない暗黙的な結合を行います。
- 多数の異なるオブジェクトに多数の少しの変更を加える場合。すべてのオブジェクトをクライアントにフェッチし、変更し、更新済としてマーク設定してから、変更後のすべてのオブジェクトをフラッシュしてサーバーに戻すほうが手軽です。

17.5.2 ナビゲーション文に使用されるルール

埋込みSQL OBJECT文は、次の仮定のもとに後述します。

- AT句がない場合、デフォルト(名前なし)接続とみなされます。
- 特に指定する場合を除き、ホスト変数は配列になります。
- 配列サイズを明示的に指定するには、FOR句を使用します。FOR句がない場合、永続ホスト変数の最小サイズが使用されます。
- 文の実行後に、SQLCAを状態変数として使用すると、処理された要素数はsqlca.sqlerrd[2]に戻されます。
- パラメータには、入力または出力を示すINまたはOUT(あるいはその両方)が指定されています。

関連項目

- 埋込みSQL文およびディレクティブ

17.5.3 OBJECT CREATE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT CREATE :obj [INDICATOR]:
obj_ind [TABLE tab] [RETURNING REF INTO :ref] ;
```

tabは次のとおりです。

{:hv | [schema.]table}

この文を使用して、オブジェクト・キャッシュ内で参照可能オブジェクトを作成します。オブジェクトの型は、ホスト変数*obj*に対応します。オプションの型ホスト変数(:obj_ind, :ref, :ref_ind)を指定する場合は、すべてが同じ型に対応している必要があります。

参照可能オブジェクトは、永続オブジェクト(TABLE句あり)でも一時オブジェクト(TABLE句なし)でもかまいません。永続オブジェクトは、暗黙的に保持され、更新済としてマークされます。一時オブジェクトは、暗黙的に保持されます。

ホスト変数は、次のとおりです。

obj (OUT)

オブジェクト・インスタンス・ホスト変数*obj*は、OTTが生成した構造体へのポインタである必要があります。この変数は、オブジェクト・キャッシュ内で作成される参照可能オブジェクトを判別するために使用されます。正常に実行されると、*obj*は新規作成されたオブジェクトを指します。

obj_ind (OUT)

この変数はOTTが生成したインジケータ構造体を指します。その型は、オブジェクト・インスタンスのホスト変数の型と対である必要があります。正常に実行されると、*obj_ind*は参照可能なオブジェクトに対する対になるインジケータ構造体へのポインタとなります。

tab (IN)

TABLE句を使用して永続オブジェクトを作成します。表名は、ホスト変数*hv*、または未宣言のSQL識別子として指定できます。スキーマ名で修飾することもできます。表名を含むホスト変数には、後続ブランクを使用しないでください。

hv (IN)

表を指定するホスト変数。ホスト変数を使用する場合、配列を使用しないでください 。また、空白文字で埋めないでください。大/小文字が区別されます。永続オブジェクトの配列を作成すると、すべてが同じ表に対応付けられます。

table (IN)

大/小文字が区別される未宣言のSQL識別子。

ref (OUT)

参照ホスト変数は、OTTによって生成される参照の型へのポインタである必要があります。*ref*の型は、オブジェクト・インスタンスのホスト変数の型と同じである必要があります。実行後の*ref*には、新規作成されたオブジェクトのrefへのポインタが含まれます。

属性は、NULLに初期設定されます。オブジェクト・ビューに対する新しいオブジェクトの作成は現在サポートされていません。

オブジェクト・ビューに対する新しいオブジェクトの作成は現在サポートされていません。

17.5.4 OBJECT DEREF

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT DEREF :ref INTO :obj [[INDICATOR]:obj_ind
] [FOR UPDATE [NOWAIT]] ;
```

オブジェクト参照*ref*を指定すると、OBJECT DEREF文は指定されたrefに対応するオブジェクトまたはオブジェクトの配列を、オブジェクト・キャッシュ内で保持します。これらのオブジェクトへのポインタは、変数*obj*および*obj_ind*内で戻されます。

ホスト変数は、次のとおりです。

ref (IN)

これはオブジェクト参照変数であり、OTTによって生成される参照の型へのポインタである必要があります。この変数(または変

数の配列)は間接参照され、キャッシュ内のそれに対応するオブジェクトへのポインタを戻します。

obj (OUT)

オブジェクト・インスタンスのホスト変数*obj*はOTTが生成した構造体へのポインタである必要があります。その型は、オブジェクト参照のホスト変数の型と同じである必要があります。正常に実行されると、*obj*にはオブジェクト・キャッシュ内で保持されたオブジェクトへのポインタが含まれます。

obj_ind (OUT)

オブジェクト・インスタンス標識変数*obj_ind*はOTTが生成したインジケータ構造体へのポインタである必要があります。その型は、オブジェクト参照の標識変数の型と対である必要があります。正常に実行されると、*obj_ind*には参照可能オブジェクトと対になるインジケータ構造体へのポインタが含まれます。

FOR UPDATE

この句を指定すると、サーバー内でそれに対応するオブジェクト用に排他ロックが取得されます。

NOWAIT

このオプション設定のキーワードを指定すると、別のユーザーがすでにオブジェクトをロックしていた場合、ただちにエラーが戻されます。

17.5.5 OBJECT RELEASE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT RELEASE :obj ;
```

この文では、オブジェクト・キャッシュ内のオブジェクトが確保解除されます。オブジェクトが確保されず、更新もされなければ、暗黙的な解放の対象になります。

オブジェクトがn回間接参照された場合は、オブジェクト・キャッシュから暗黙的に解放されるように、n回リリースする必要があります。不要になったオブジェクトは、すべてリリースすることをお薦めします。

17.5.6 OBJECT DELETE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT DELETE :obj ;
```

永続オブジェクトの場合、この文はオブジェクト・キャッシュ内のオブジェクトまたはオブジェクトの配列を削除済としてマークします。オブジェクトがサーバー内で削除されるのは、そのオブジェクトのフラッシュ時またはキャッシュのフラッシュ時です。オブジェクト・キャッシュ内で確保されているメモリーは解放されません。

一時オブジェクトの場合は、そのオブジェクトが削除済としてマーク設定されます。オブジェクト用のメモリーは解放されません。

17.5.7 OBJECT UPDATE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT UPDATE :obj ;
```

永続オブジェクトの場合、この文はオブジェクト・キャッシュ内でオブジェクトを更新済としてマークします。変更結果は、オブジェクトのフラッシュ時またはキャッシュのフラッシュ時に、サーバーに書き込まれます。

一時オブジェクトの場合、この文は何も操作しません。

17.5.8 OBJECT FLUSH

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT FLUSH :obj ;
```

この文は、更新済、削除済または作成済としてマークされた永続オブジェクトをサーバーにフラッシュします。

> **注意:**
>
> オブジェクトがフラッシュされると、排他ロックが暗黙的に取得されます。この文が正常終了すると、オブジェクトのマークが削除されます。オブジェクトのバージョンが LATEST(後続の項を参照)であれば、暗黙的にリフレッシュされます。

17.5.9 オブジェクトへのナビゲーショナル・アクセス

ナビゲーショナル・アクセス用インタフェースの実例は、図17-2を参照してください。

ALLOCATE文を使用して、*person*オブジェクトを指すREFのコピー用に、オブジェクト・キャッシュ内のメモリーを割り当てます。割り当てられたREFには、データは含まれません。

```
person *per_p;
person_ref *per_ref_p;
...
EXEC SQL ALLOCATE :per_p;
```

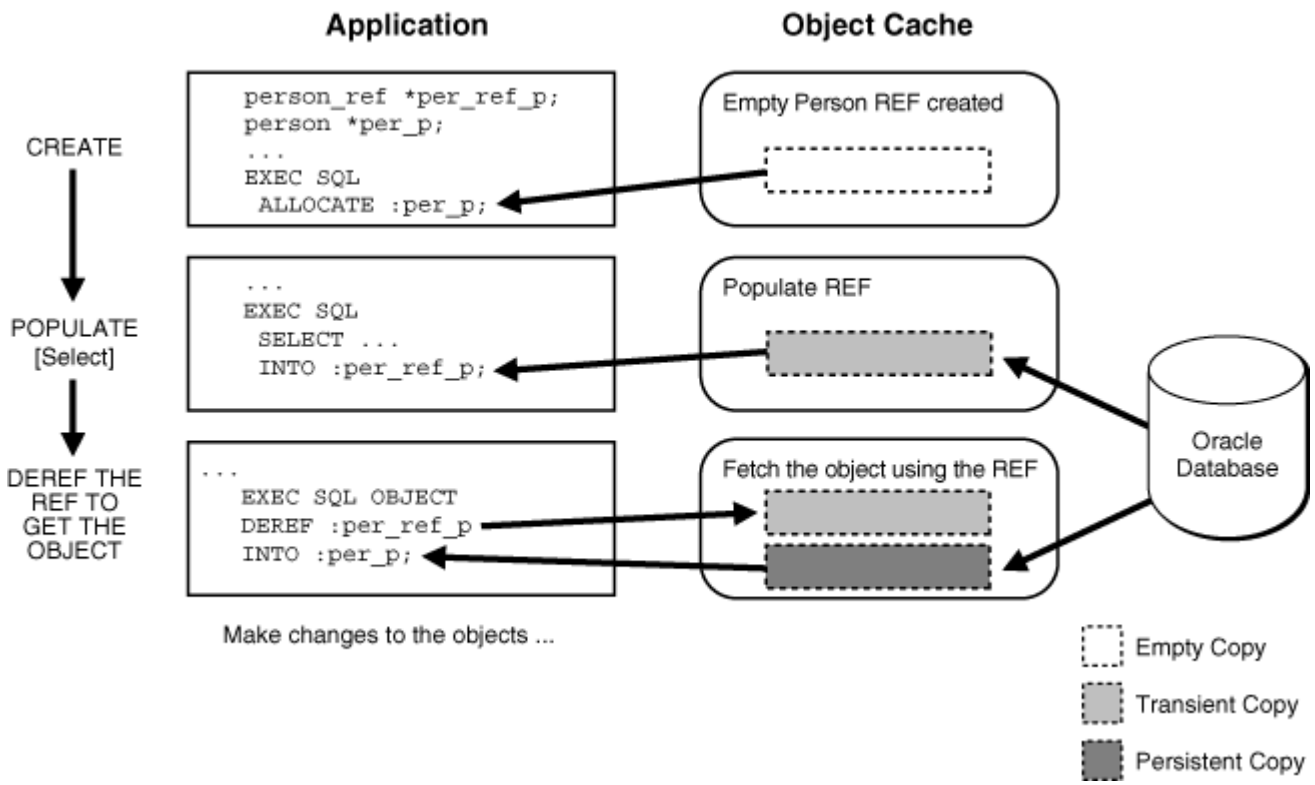
SELECT文を使用して*person*オブジェクトのREFを取り出すことで、割り当てたメモリーに移入します(正確な書式はアプリケーションによって異なります)。

```
EXEC SQL SELECT ... INTO :per_ref_p;
```

次に、オブジェクト内で変更できるように、DEREF文を使用してキャッシュ内でオブジェクトを確保します。DEREF文は、ポインタ*per_ref_p*を使用して、クライアント側キャッシュ内で*person*オブジェクトのインスタンスを作成します。*person*オブジェクトへのポインタ*per_p*が戻されます。

```
EXEC SQL OBJECT DEREF :per_ref_p INTO :per_p;
```

図17-2 ナビゲーショナル・アクセス



C言語の代入文を使用するか、またはOBJECT SET文によるデータ変換を使用して、キャッシュ内でオブジェクトを変更します。

次に、オブジェクトを更新済としてマーク設定する必要があります。図17-3を参照してください。キャッシュ内のオブジェクトを更新済としてマーク設定し、サーバーにフラッシュできるようにするには、次の文を使用します。

```
EXEC SQL OBJECT UPDATE :per_p;
```

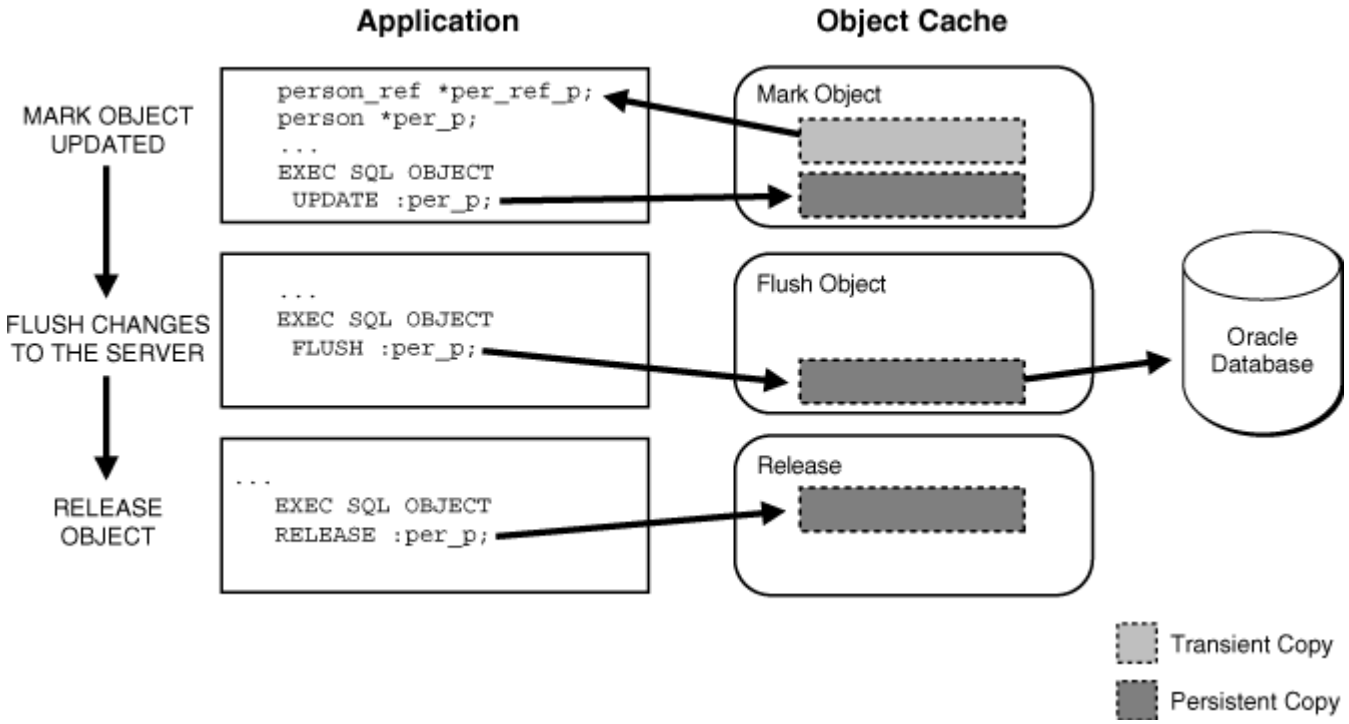
FLUSH文によって変更結果をサーバーに送ります。

```
EXEC SQL OBJECT FLUSH :per_p;
```

オブジェクトをリリースします。

```
EXEC SQL OBJECT RELEASE :per_p;
```

図17-3 ナビゲーショナル・アクセス(続き)



オブジェクト属性とC言語のデータ型の間で変換するには、次の項で説明する文を使用します。

17.6 オブジェクト属性とC言語のデータ型の変換

この項では、属性と型の変換に関する問題を説明します。

17.6.1 OBJECT SET

```
EXEC SQL [AT [:]database]
  OBJECT SET  [ {'*' | {attr[, attr]} } OF]
    :obj [[INDICATOR]:obj_ind]
       TO {:hv [[INDICATOR]:hv_ind]
          [, :hv [INDICATOR]:hv_ind]]} ;
```

この文は、連想アクセス用インタフェースとナビゲーショナル・アクセス用インタフェースのどちらで作成したオブジェクトにも使用します。この文によって、オブジェクトの属性が更新されます。永続オブジェクトの場合は、オブジェクトが更新され、フラッシュされたときに、変更がサーバーに書き込まれます。キャッシュのフラッシュは、更新済オブジェクトになされたすべての変更をサーバーに書き込みます。

OF句はオプション設定です。OF句を指定しなければ、*obj*の属性がすべて設定されます。次のように記述しても同じ結果が得られます。

```
... OBJECT SET * OF ...
```

ホスト変数リストには、属性の値を提供するように展開された構造体を組み込むことができます。ただし、*obj*内の属性の数は、展開する変数リスト内の要素数と同じである必要があります。

ホスト変数と属性は、次のとおりです。

attr

各属性はホスト変数ではなく、オブジェクトのどの属性が更新されるかを指定する識別子にすぎません。リスト内の最初の属性は、リスト内の最初の式と対になります。属性は、OCIString、OCINumber、OCIDateまたはOCIRefのいずれかにする必要があります。

obj (IN/OUT)

*obj*には、更新対象となるオブジェクトを指定します。バインド変数*obj*に配列を使用することはできません。これはOTTが生成した構造体へのポインタである必要があります。

obj_ind (IN/OUT)

更新対象となる対になるインジケータ構造体です。これはOTTが生成したインジケータ構造体へのポインタである必要があります。

hv (IN)

これは、OBJECT SET文への入力に使用されるバインド変数です。*hv*はint、float、OCIRef *、1次元文字配列またはこれらの型の構造体である必要があります。

hv_ind (IN)

これは、OBJECT SET文への入力に使用される対になるインジケータです。*hv_ind*は2バイト整数スカラーまたは2バイト整数スカラーの構造体である必要があります。

標識変数の使用方法:

ホスト変数のインジケータがある場合は、オブジェクト・インジケータも必要です。

hv_indを-1に設定すると、それに対応付けられたフィールドがobj_ind内で-1に設定されます。

次の暗黙的な変換が許されます。

- [OCIString | STRING | VARCHAR | CHARZ]からOCIStringへ
- OCIRefからOCIRefへ
- [OCINumber | int | float | double]からOCINumberへ
- [OCIDate | STRING | VARCHAR | CHARZ]からOCIDateへ

注意:

- ネストされた構造体は使用できません。
- この文を使用して、参照可能オブジェクトをアトミック NULL に設定することはできません。かわりに、NULL インジケータの適切なフィールドを設定してください。
- OCIDateTime または OCIInterval データ型と OCIString との間での変換はサポートされていません。

17.6.2 OBJECT GET

```
EXEC SQL [AT [:]database]
   OBJECT GET [ { '*' | {attr[, attr]} } FROM]
     :obj [[INDICATOR]:obj_ind]
        INTO {:hv [[INDICATOR]:hv_ind]
          [, :hv [[INDICATOR]:hv_ind]]} ;
```

この文では、オブジェクトの属性がネイティブなC言語のデータ型に変換されます。

FROM句はオプションです。FROM句を指定しなければ、*obj*の属性がすべて変換されます。次のように記述しても同じ結果が得られます。

```
... OBJECT GET * FROM ...
```

ホスト変数リストには、属性の値を受け取るように展開された構造体を組み込んでもかまいません。ただし、*obj*内の属性の数は、展開されるホスト変数リスト内の要素の数と同じである必要があります。

ホスト変数と属性は、次のとおりです。

attr

各属性はホスト変数ではなく、オブジェクトのどの属性が取り出されるかを指定する識別子にすぎません。リスト内の最初の属性は、リスト内の最初のホスト変数と対になります。属性はベース型を表す必要があります。OCIString、OCINumber、OCIRefまたはOCIDateのいずれかにする必要があります。

obj (IN)

この変数では、属性を取り出すときのソースとして機能するオブジェクトを指定します。バインド変数*obj*に配列を使用することはできません。

hv (OUT)

これは、OBJECT GET文からの出力を保持するためのバインド変数です。int、float、double、1次元文字配列またはこれらの型の構造体にできます。この文では、このホスト変数に変換済の属性値が戻されます。

hv_ind (OUT)

これは、属性値に対応付けられた標識変数です。また、2バイト整数スカラーまたは2バイト整数スカラーの構造体です。

標識変数の使用方法:

オブジェクト・インジケータを指定しなかった場合、属性は有効とみなされます。オブジェクトがアトミックNULLの場合または、要求した属性がNULLで、オブジェクト標識変数を指定しなかった場合は、オブジェクト属性がC言語のデータ型に変換されるプログラム・エラーになります。この状況では、*Oracle*エラーを呼び出せないことがあります。

オブジェクト変数がアトミックNULLの場合、または要求した属性がNULLで、ホスト変数インジケータ(*hv_ind*)が与えられている場合は、-1に設定されます。

オブジェクトがアトミックNULLの場合、または要求した属性がNULLで、ホスト変数インジケータが与えられていない場合は、エラーが発生します。

次の暗黙的な変換が許されます。

- OCIStringから[STRING | VARCHAR | CHARZ | OCIString]へ
- OCINumberから[int | float | double | OCINumber]へ
- OCIRefからOCIRefへ
- OCIDateから[STRING | VARCHAR | CHARZ | OCIDate]へ

注意:

- ネストされた構造体は使用できません。
- OCIDateTime または OCIInterval データ型と OCIString との間での変換はサポートされていません。

17.7 オブジェクト・オプションの設定/取得

ランタイム・コンテキストには、ランタイム・コンテキストが生成されて割り当てられたときにデフォルトの値が設定されるオプションがあります。これらのオプションは次の埋込みSQLディレクティブで設定します。

17.7.1 CONTEXT OBJECT OPTION SET

```
EXEC SQL CONTEXT OBJECT OPTION SET {option[, option]} TO {:hv[, :hv]} ;
```

変数は、次のとおりです。

:hv(IN) ...

入力バインド変数*hv ...*はSTRING型、VARCHAR型またはCHARZ型です。

option...

ランタイム・コンテキストのどのオプションを更新するかを指定する単純な識別子です。最初のオプションは、最初の入力バインド変数と対になります。現在サポートされている値を次に示します。

表17-1 CONTEXT OBJECT OPTION値の有効な選択肢

オプション値	指定内容
DATEFORMAT	Date 属性とコレクション要素の書式。
DATELANG	Date 型と Datetime 型すべてのグローバリゼーション・サポート言語。

次に例を示します。

```
char *new_format = "DD-MM-YYYY";
char *new_lang = "French";
char *new_date = "14-07-1789";
/* One of the attributes of the license type is dateofbirth */
license *aLicense;
...
/* Declaration and allocation of context ... */
EXEC SQL CONTEXT OBJECT OPTION SET DATEFORMAT, DATELANG TO :new_format,    :new_lang;
/* Navigational object obtained  */
...
EXEC SQL OBJECT SET dateofbirth OF :aLicense TO :new_date;
...
```

関連項目

- Oracle Database SQL言語リファレンス
- CONTEXT OBJECT OPTION SET(実行可能埋込みSQL拡張機能)

17.7.2 CONTEXT OBJECT OPTION GET

影響されるコンテキストは、その時点で使用されているコンテキストと解釈されます。これらのオプションの値を決定するには、次のディレクティブを使用します。

```
EXEC SQL CONTEXT OBJECT OPTION GET {option[, option]} INTO {:hv[, :hv]} ;
```

表17-1のオプションの値は次のとおりです。

出力に使用されるバインド変数*hv* ...はSTRING型、VARCHAR型またはCHARZ型です。影響されるコンテキストは、その時点で使用されているコンテキストと解釈されます。

関連項目

- Oracle Database SQL言語リファレンス
- CONTEXT OBJECT OPTION GET(実行可能埋込みSQL拡張機能)

17.8 オブジェクトに対する新しいプリコンパイラ・オプション

オブジェクトをサポートするには、次のプリコンパイラ・オプションを使用します。

17.8.1 VERSION

このオプションでは、EXEC SQL OBJECT DEREF文によってどのバージョンのオブジェクトが戻されるかが決まります。これにより、

キャッシュ・オブジェクトとサーバー・オブジェクトの間で、一貫性レベルを変更できます。

EXEC ORACLE OPTION文を使用してインラインで設定します。設定できる値は、次のとおりです。

RECENT (デフォルト)

現行のトランザクション内でオブジェクトがオブジェクト・キャッシュに選択されている場合は、そのオブジェクトが戻されます。オブジェクトが選択されていない場合は、サーバーから取り出されます。シリアライズされた状態で実行中のトランザクションの場合、このオプションの動作は VERSION=LATEST と同じですが、ネットワークのラウンドトリップはそれほど多くありません。この値は、ほとんどのPro*C/C++アプリケーションで問題なく使用できます。

LATEST

オブジェクトがオブジェクト・キャッシュに存在しない場合は、データベースから取り出されます。オブジェクト・キャッシュに存在している場合は、サーバーからリフレッシュされます。この値を指定すると、ネットワークのラウンドトリップが増大するため、慎重に使用してください。この値を使用するのは、オブジェクト・キャッシュでサーバー側バッファとできるかぎり一貫性のある状態を保つ必要がある場合です。

ANY

オブジェクトがすでにオブジェクト・キャッシュに存在している場合は、そのオブジェクトが戻されます。オブジェクトがオブジェクト・キャッシュに存在しなければ、サーバーから取り出されます。この値を指定すると、ネットワークのラウンドトリップは最小になります。この値を使用するのは、アプリケーションが読取り専用オブジェクトにアクセスする場合や、ユーザーがオブジェクトに排他的にアクセスする場合です。

17.8.2 DURATION

このプリコンパイラ・オプションは、後続のEXEC SQL OBJECT CREATE文とEXEC SQL OBJECT DEREF文に使用される確保継続時間を設定するときに使用します。キャッシュ内のオブジェクトは、保持期間の終わりに暗黙的に解放されます。

ナビゲーショナル・アクセス用インタフェースでのみ使用します。

このオプションは、EXEC ORACLE OPTION文で設定できます。設定できる値は、次のとおりです。

TRANSACTION (デフォルト)

オブジェクトは、トランザクションの完了時に暗黙的に解放されます。

SESSION

オブジェクトは、接続の終了時に暗黙的に解放されます。

17.8.3 OBJECT

このプリコンパイラ・オプションを指定すると、オブジェクト・キャッシュを使用できます。

DBMS=NATIVE | V8に対するOBJECTSのデフォルト値はYESです。オブジェクト・キャッシュのデフォルト・サイズは、OCIデフォルト・キャッシュ・サイズと同じく8MBです。

関連項目

- OBJECTS

17.8.4 INTYPE

プログラムでオブジェクト型、コレクション・オブジェクト・タイプまたはREF型を使用する場合は、このコマンドライン・オプションでINTYPEファイルを指定する必要があります。

次の構文で、INTYPEオプションを指定します。

```
INTYPE=filename1 INTYPE=filename2 ...
```

この場合、*filename1*、*filename2*...は、OTTで生成された型ファイルの名前です。これらのファイルはPro*C/C++への読取り専用入力ファイルになります。それに含まれる情報は単純なテキスト形式ですが、エンコードされている場合もあり、ユーザーが読める形式になっているとはかぎりません。

Pro*C/C++の1つのプリコンパイル単位に対する入力ファイルとして、複数のINTYPEファイルを指定できます。

このオプションは、EXEC ORACLE文内でインラインで使用することはできません。

OTTは、データベース内で作成されたオブジェクト型を表すC言語の構造体の宣言を生成し、型ファイルと呼ばれるファイルに型の名前とバージョン情報を書き込みます。

オブジェクト型の名前は、それを表すC言語の構造体の型やC++クラスの型と同じであるとはかぎりません。これには、次の理由が考えられます。

- サーバーに指定されたオブジェクト型の名前に、CまたはC++識別子で無効な文字が含まれている場合
- ユーザーがOTTに対して、構造体またはクラスに異なる名前を使用するように要求した場合
- ユーザーがOTTに対して、名前の大文字を小文字に、小文字を大文字に変更するように要求した場合

前述の状況では、構造体やクラスの宣言からは、その構造体やクラスがどのオブジェクト型と一致するかを推論できません。この情報はPro*C/C++に必要であり、OTTによって型ファイル内で生成されます。

17.8.5 ERRTYPE

```
ERRTYPE=filename
```

エラーは、画面のみでなく、指定したファイルにも書き込まれます。このオプションを省略すると、エラーは画面にのみ出力されます。ただし、ERRTYPEは1つしか指定できません。値が1つしかない他のコマンドライン・オプションと同様に、コマンドラインでERRTYPEに複数の値を入力すると、最後の値がすべてに優先します。

このオプションは、EXEC ORACLE文内でインラインで使用することはできません。

17.8.6 オブジェクトに対するSQLCHECKのサポート

各オブジェクト型とその属性は、Oracle型のCバインドに従ってCプログラムに表されます。プリコンパイラ・コマンドライン・オプションSQLCHECKをSEMANTICSまたはFULLに設定すると、Pro*C/C++はプリコンパイル中に、ホスト変数型がデータベース・

スキーマ内でその型に必須のCバインドに準拠しているかどうかを検証します。さらに、Oracle型がプログラム実行中に正しくマップされているかどうかを検証するために、常に実行時チェックが行われます。

リレーショナル・データ型は通常の方法でチェックされます。

リレーショナルSQLデータ型とホスト変数型は、両方の型が同一の場合または両方の型の間で変換が可能な場合に、互換性を持ちます。一方、オブジェクト型が互換性を持つのは、両方の型が同一の場合のみです。それらの型を次のように指定する必要があります。

- 同じ名前にします。
- 同じスキーマに含めます(スキーマが明示的に指定されている場合)。

オプションSQLCHECK=SEMANTICSまたはFULLを指定すると、Pro*C/C++はプリコンパイル中に、指定されたユーザーIDとパスワードを使用してデータベースにログインし、構造体の宣言が生成されたオブジェクト型と、埋込みSQL文に使用されたオブジェクト型が同じかどうかを検証します。

17.8.7 実行時のタイプ・チェック

Pro*C/C++は、ある型について、入力INTYPEファイルからオブジェクト、コレクション・オブジェクトおよびREFホスト変数の型の名前とバージョン、可能な場合にはスキーマ情報を収集し、これらの情報を生成したコードに格納します。これにより、実行時にオブジェクトおよびREFバインド変数の型情報にアクセスできます。型が同じでない場合は、固有のエラー・メッセージが戻されます。

17.9 Pro*C/C++のオブジェクト例

簡単なオブジェクトの例を検証します。型*person*と表*person_tab*を作成します。この表には同じくオブジェクト型の列*address*が含まれています。

```
create type person as object (
         lastname        varchar2(20),
         firstname       char(20),
         age             int,
         addr            address
)
/
create table person_tab of person;
```

表にデータを挿入し、処理します。

17.9.1 連想アクセス

Pro*C/C++を使用して、*lastname*の値を「Smith」から「Smythe」に変更する方法を考えてみます。

OTTを実行して、*person*にマップするC言語の構造体を生成します。Pro*C/C++プログラムに、OTTによって生成されるヘッダー・ファイルをインクルードする必要があります。

アプリケーション内で、クライアント側キャッシュ内の永続メモリーへのポインタ、*person_p*を宣言します。それからメモリーを割り当てて、戻されたポインタを使用します。

```
char *new_name = "Smythe";
person *person_p;
...
EXEC SQL ALLOCATE :person_p;
```

これで、永続オブジェクトのコピーにメモリーが割り当てられます。割当て済オブジェクトには、まだデータは含まれていません。

C言語の代入文か、SELECT文またはFETCH文を使用して既存のオブジェクトを取り出し、キャッシュにデータを入れます。

```
EXEC SQL SELECT VALUE(p) INTO :person_p FROM person_tab p WHERE lastname = 'Smith';
```

キャッシュ内のコピーに対する変更結果は、INSERT文、UPDATE文およびDELETE文を使用してサーバー・データベースに送信します。

```
EXEC SQL OBJECT SET lastname OF :person_p TO :new_name;
EXEC SQL INSERT INTO person_tab VALUES(:person_p);
```

次の文を使用してキャッシュ・メモリーを解放します。

```
EXEC SQL FREE :person_p;
```

17.9.2 ナビゲーショナル・アクセス

オブジェクト・キャッシュ内に、REFをオブジェクト*person*にコピーするためのメモリーを割り当てます。ALLOCATE文はREFへのポインタを戻します。

```
person *person_p;
person_ref *per_ref_p;
...
EXEC SQL ALLOCATE :per_ref_p;
```

割当て済のREFには、データが含まれていません。データを入れるには、オブジェクトのREFを取り出します。

```
EXEC SQL SELECT ... INTO :per_ref_p;
```

それからREFを間接参照して、オブジェクトのインスタンスをクライアント側のキャッシュに入れます。間接参照コマンドでは、*per_ref_p*を使用して、キャッシュ内で対応するオブジェクトのインスタンスを作成します。

```
EXEC SQL OBJECT DEREF :per_ref_p INTO :person_p;
```

C言語の代入を使用するか、またはOBJECT GET文を使用して、キャッシュ内のデータを変更します。

```
/* lname is a C variable to hold the result */
EXEC SQL OBJECT GET lastname FROM :person_p INTO :lname;
...
EXEC SQL OBJECT SET lastname OF :person_p TO :new_name;
/* Mark the changed object as changed with OBJECT UPDATE command */;
EXEC SQL OBJECT UPDATE :person_p;
EXEC SQL FREE :per_ref_p;
```

変更をデータベース内で永続的なものにするために、FLUSHを使用します。

```
EXEC SQL OBJECT FLUSH :person_p;
```

サーバーが変更されたため、オブジェクトをリリースできます。リリースされるオブジェクトが、オブジェクト・キャッシュ・メモリーからただちに解放されるとはかぎりません。最近の使用頻度が最も低いスタックに置かれます。キャッシュがいっぱいになると、オブジェクトはメモリーからスワップされます。

リリースされるのはオブジェクトのみで、そのオブジェクトを指すREFはキャッシュ内に残留します。REFをリリースするには、REF用のRELEASE文を使用します。person_pが指すオブジェクトをリリースするには次のようにします。

```
EXEC SQL OBJECT RELEASE :person_p;
```

または、確保継続時間が適切に設定されていれば、トランザクション・コミットを発行すると、キャッシュ内のオブジェクトがすべてリリースされます。

17.10 型の継承のサンプル・コード

次のコード例では、4つのオブジェクト型が作成されます。

- Person_t
- Person_tのサブタイプとしてEmployee_t
- Person_tのサブタイプとしてStudent_t
- Student_tのサブタイプとしてPartTimeStudent_t

また、次の表も作成されます。

- Person_tとそのサブタイプのオブジェクトを保持するためのperson_tab

次のSQLファイルinhdemo1.sqlでは、オブジェクト型と表が生成されてから、表に値が挿入されます。

```
connect scott/tiger;

rem ** Always drop your objects in reverse dependency order
drop table person_tab;
drop type PartTimeStudent_t;
drop type Student_t;
drop type Employee_t;
drop type Person_t;

rem ** Create the TYPES, TYPED TABLES and TABLES we need

rem ** Create a Person_t ADT
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;
/

rem ** Create a Person_t subtype Employee_t
CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr VARCHAR2(30));
/

rem ** Create a Person_t subtype Student_t
CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
   major VARCHAR2(30)) NOT FINAL;
/

rem ** Create a Student_t subtype PartTimeStudent_t
CREATE TYPE PartTimeStudent_t UNDER Student_t
( numhours NUMBER);
/

rem ** Create a typed table for person_t objects
CREATE table person_tab of person_t;

rem ** Insert 2 Employee_t objects into the person_t typed table
insert into person_tab values
  (Employee_t(123456, 'Alison Laurence', '100 Geary Street, San Francisco, CA 94013',
1001, 'CEO'));
insert into person_tab values
```

```
  (Employee_t(234567, 'William Bates', '123 Main Street, Anytown, WA 97818',
1002,'CFO'));

rem ** Insert 2 Student_t objects into the person_t typed table
insert into person_tab values
  (Student_t(20001, 'Van Gates', '1825 Aikido Way, Los Angeles, CA, 45300', 20,
'English'));
insert into person_tab values
  (Student_t(20002, 'Bill Wallace', '12 Shugyo Blvd, Los Angeles, CA, 95100', 30,
'Computer Science'));

rem ** Insert 1 PartTimeStudent_t object into the person_t typed table
insert into person_tab values
  (PartTimeStudent_t(20003, 'Jean Claude', '874 Richmond Street, New York, NY 45100',
40, 'Music',20));

commit;
```

例で使用したintypeファイルinhdemo1.typのリストを次に示します。

```
case=same
type person_t
type employee_t
type student_t
type parttimestudent_t
```

プリコンパイラ・ファイルinhdemo1.pcのリストを次に示します。

```
/*****************************************************************************
 *
 * This is a simple Pro*C/C++ program designed to illustrate how to
 * access type inheritance objects.
 *
 * To build the executable:
 *
 *   1. Execute the SQL script, inhdemo1.sql in SQL*Plus to create:
 *   - 4 object types person_t, employee_t as a subtype of person_t,
 *              student_t as a subtype of person_t and parttimestudent_t as
 *              a subtype of student_t.
 *   - 1 typed table person_tab to hold "person_t" and its subtype objects
 *
 *   2. Run OTT: (The following command should appear on one line)
 *        ott intype=inhdemo1.typ hfile=inhdemo1.h outtype=out.typ
 *            code=c userid=scott/tiger
 *
 *   3. Precompile using Pro*C/C++:
 *        proc inhdemo1 intype=out.typ
 *   4. Compile/Link (This step is platform specific)
 *
 *****************************************************************************/

/* Include files */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlda.h>

#include <sqlca.h>                                  /* SQL Communications Area */
#include <sql2oci.h>         /* SQLLIB interoperability routines for OCI8 */
```

```
#include "inhdemo1.h"        /* OTT-generated header with C typedefs for the */
                                    /* database types "person" and "address" */
/* Macros */
#define ARRAY_SIZE 10
#define NAME_LENGTH 31
#define ADDR_LENGTH 101

/* Global variables */

  char *uid="scott/tiger";
  int i;
  int count;
  VARCHAR  dynstmt[100];

main()
{

  printf("¥n*** STARTING OBJECT TYPE INHERITANCE DEMO ***¥n");

  EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--¥n");

  EXEC SQL connect :uid;
  printf("Connected successfully.¥n");

  exec sql select count(*) into :count from person_tab;
  printf("¥nThere are %d entries in table person_tab.¥n", count);

  do_fetch_all();                                  /* Fetch person_t objects */
  do_fetch_employee();                            /* Fetch employee_t objects */
  do_fetch_student();                           /* Fetch only student_t objects */
  do_fetch_parttimestudent();                 /* Fetch parttimestuden_t objects */
  do_fetch_student_employee();     /* Fetch student_t and employee_t objects */

  printf("¥nFetching only student_t objects with dynamic sql:¥n");
  strcpy((char *)dynstmt.arr,
   "SELECT value(p) from person_tab p where value(p) is of (only student_t)");
  do_dynamic_fetch();             /* Fetch student_t object with dynamic sql */

  printf("¥nFetching student_t and its subtype objects with dynamic sql:¥n");
  strcpy((char *)dynstmt.arr,
   "SELECT treat(value(p) as student_t) from person_tab p where value(p) is
of(student_t)");
  do_dynamic_fetch();             /* Fetch student_t object with dynamic sql */

  printf("¥n*** END OF OBJECT TYPE INHERITANCE DEMO ***¥n");
  exit(EXIT_SUCCESS);

}

void printPerson(person)
  person_t *person;
{
  int writtenSSN=-1;
  text writtenName[NAME_LENGTH];
  text writtenAddr[ADDR_LENGTH];

  EXEC SQL OBJECT GET SSN, NAME, ADDRESS FROM :person INTO
     :writtenSSN, :writtenName, :writtenAddr;
  printf("¥nSSN=%10d¥nNAME=%s¥nAddr=%s¥n", writtenSSN, writtenName,
          writtenAddr);
```

```
}

void printEmployee(employee)
  employee_t *employee;
{
  int writtenID=-1;
  text writtenMgr[NAME_LENGTH];

  printPerson(employee);
  EXEC SQL OBJECT GET EMPID, MGR FROM :employee INTO :writtenID, :writtenMgr;
  printf("EMPID=%10d¥nMGR=%s¥n", writtenID, writtenMgr);
}

void printStudent(student)
  student_t *student;
{
  int writtendeptid=-1;
  text writtenMajor[NAME_LENGTH];

  printPerson(student);
  EXEC SQL OBJECT GET DEPTID, MAJOR FROM :student INTO :writtendeptid, :writtenMajor;
  printf("DEPTID=%10d¥nMAJOR=%s¥n", writtendeptid, writtenMajor);
}

void printPartTimeStudent(parttimes)
  parttimestudent_t *parttimes;
{
  int written_numhours=-1;

  printStudent(parttimes);
  EXEC SQL OBJECT GET NUMHOURS FROM :parttimes INTO :written_numhours;
  printf("NUMHOURS=%10d¥n", written_numhours);
}

/* Declare error handling function. */
sql_error(msg)
    char *msg;
{
    char err_msg[128];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("¥n%s¥n", msg);
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s¥n", msg_len, err_msg);

    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_FAILURE);
}

/*****************************************************************************
 * The following function shows how to select person_t objects
 *****************************************************************************/

do_fetch_all()
{
person_t *personArray[ARRAY_SIZE];
person_t_ind *personArray_ind[ARRAY_SIZE];
```

```
  printf("¥nFetching person_t objects:¥n");

  exec sql declare c1 cursor for
    select value(p) from person_tab p;

  exec sql allocate :personArray:personArray_ind;

  exec sql open c1;

  exec sql whenever not found goto :done;
  while(sqlca.sqlcode==0)
    {
      exec sql fetch c1 into :personArray:personArray_ind;
      if (sqlca.sqlcode == 1403) goto done;
      for (i=0; i < ARRAY_SIZE; i++ )
        printPerson(personArray[i]);
    }

 done:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY_SIZE; i++)
    printPerson(personArray[i]);

  printf("Total number of person_t objects fetched: %d.¥n",
          sqlca.sqlerrd[2]);

  exec sql close c1;
  exec sql free :personArray:personArray_ind;
}

/*****************************************************************************
 * The following function shows how to select person_t subtype employee_t
 * objects
 *****************************************************************************/

do_fetch_employee()
{
employee_t *empArray[ARRAY_SIZE];
employee_t_ind *empArray_ind[ARRAY_SIZE];

  printf("¥nFetching employee_t objects:¥n");

  exec sql allocate :empArray:empArray_ind;

  exec sql declare c2 cursor for
    select value(p) from person_tab p
      where value(p) is of (employee_t);

  exec sql open c2;

  exec sql whenever not found goto :done_emp;
  while(sqlca.sqlcode==0)
    {
      exec sql fetch c2 into :empArray:empArray_ind;
      for (i=0; i < ARRAY_SIZE; i++ )
        printEmployee(empArray[i]);
    }

 done_emp:
   for (i=0; i < sqlca.sqlerrd[2] % ARRAY_SIZE; i++)
```

```
    printEmployee(empArray[i]);

  printf("Total number of employee_t objects fetched: %d.¥n",
         sqlca.sqlerrd[2]);

  exec sql close c2;
  exec sql free :empArray:empArray_ind;
}

/*****************************************************************************
 * The following function shows how to select person_t subtype student_t
 * objects
 *****************************************************************************/

do_fetch_student()
{
student_t *studentArray[ARRAY_SIZE];
student_t_ind *studentArray_ind[ARRAY_SIZE];

  printf("¥nFetching student_t objects:¥n");

  exec sql declare c3 cursor for
    select value(p) from person_tab p
      where value(p) is of (student_t);

  exec sql allocate :studentArray:studentArray_ind;

  exec sql open c3;

  exec sql whenever not found goto :done_student;
  for (;;)
    {
      exec sql fetch c3 into :studentArray:studentArray_ind;
      for (i=0; i < ARRAY_SIZE; i++ )
        printStudent(studentArray[i]);
    }

 done_student:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY_SIZE; i++)
    printStudent(studentArray[i]);

  printf("Total number of student_t objects fetched: %d.¥n",
         sqlca.sqlerrd[2]);

  exec sql close c3;
  exec sql free :studentArray:studentArray_ind;
}

/*****************************************************************************
 * The following function shows how to select student_t subtype
 * parttimestudent objects
 *****************************************************************************/

do_fetch_parttimestudent()
{
parttimestudent_t *parttimestudentArrayArray[ARRAY_SIZE];
parttimestudent_t_ind *parttimestudentArrayArray_ind[ARRAY_SIZE];

  printf("¥nFetching parttimestudent_t objects:¥n");
```

```
  exec sql declare c4 cursor for
    select value(p) from person_tab p
      where value(p) is of (parttimestudent_t);

  exec sql allocate :parttimestudentArrayArray:parttimestudentArrayArray_ind;

  exec sql open c4;

  exec sql whenever not found goto :done_parttimestudent;
  while(sqlca.sqlcode==0)
    {
      exec sql fetch c4 into :parttimestudentArrayArray:parttimestudentArrayArray_ind;
      for (i=0; i < ARRAY_SIZE; i++ )
        printPartTimeStudent(parttimestudentArrayArray[i]);
    }

 done_parttimestudent:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY_SIZE; i++)
    printPartTimeStudent(parttimestudentArrayArray[i]);

  printf("Total number of parttimestudent_t objects fetched: %d.¥n",
          sqlca.sqlerrd[2]);

  exec sql close c4;
  exec sql free :parttimestudentArrayArray:parttimestudentArrayArray_ind;
}

/*****************************************************************************
 * The following function shows how to select person_t subtypes student_t
 * and employee_t objects
 *****************************************************************************/

do_fetch_student_employee()
{
person_t *personArray[ARRAY_SIZE];
person_t_ind *personArray_ind[ARRAY_SIZE];

  printf("¥nFetching only student_t and employee_t objects:¥n");

  exec sql declare c5 cursor for
    select value(p) from person_tab p
      where value(p) is of (only student_t, employee_t);

  exec sql allocate :personArray:personArray_ind;

  exec sql open c5;

  exec sql whenever not found goto :done_student_employee;
  while(sqlca.sqlcode==0)
    {
      exec sql fetch c5 into :personArray:personArray_ind;
      for (i=0; i < ARRAY_SIZE; i++ )
        printPerson(personArray[i]);
    }

 done_student_employee:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY_SIZE; i++)
        printPerson(personArray[i]);

  printf("Total number of stuent_t and employee_t objects fetched: %d.¥n",
```

```
sqlca.sqlerrd[2]);

  exec sql close c5;
  exec sql free :personArray:personArray_ind;
}

/*****************************************************************************
 * The following function shows how to select person_t subtype student_t
 * objects using dynamic sql.
 *****************************************************************************/

do_dynamic_fetch()
{
student_t *student;
student_t_ind  *student_ind;

  exec sql allocate :student:student_ind;

  dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);
  EXEC SQL PREPARE S FROM :dynstmt;
  EXEC SQL DECLARE C CURSOR FOR S;
  EXEC SQL OPEN C;

  exec sql whenever not found do break;
  for (;;)
  {
     EXEC SQL FETCH C INTO :student:student_ind;
     printStudent(student);
  }

  printf("¥nQuery returned %d row%s.¥n", sqlca.sqlerrd[2],
         (sqlca.sqlerrd[2] == 1) ? "" : "s");

  EXEC SQL CLOSE C;
  exec sql free :student:student_ind;
}
```

17.11 ナビゲーショナル・アクセスのサンプル・コード

サンプル・コードでは3つのオブジェクト型が生成されます。*budoka*は武道家です。

- customer
- budoka
- location

さらに、次の2つの表が作成されます。

- person_tab
- customer_tab

次のSQLファイルnavdemo1.sqlでは、型と表が生成されてから、表に値が挿入されます。

```
connect scott/tiger

drop table customer_tab;
drop type customer;
drop table person_tab;
drop type budoka;
```

```
drop type location;

create type location as object (
        num     number,
        street  varchar2(60),
        city    varchar2(30),
        state   char(2),
        zip     char(10)
);
/

create type budoka as object (
        lastname        varchar2(20),
        firstname       varchar(20),
        birthdate       date,
        age             int,
        addr            location
);
/

create table person_tab of budoka;

create type customer as object (
        account_number varchar(20),
        aperson ref budoka
);
/

create table customer_tab of customer;

insert into person_tab values (
        budoka('Seagal', 'Steven', '14-FEB-1963', 34,
                location(1825, 'Aikido Way', 'Los Angeles', 'CA', 45300)));
insert into person_tab values (
        budoka('Norris', 'Chuck', '25-DEC-1952', 45,
                location(291, 'Grant Avenue', 'Hollywood', 'CA', 21003)));
insert into person_tab values (
        budoka('Wallace', 'Bill', '29-FEB-1944', 53,
                location(874, 'Richmond Street', 'New York', 'NY', 45100)));
insert into person_tab values (
        budoka('Van Damme', 'Jean Claude', '12-DEC-1964', 32,
                location(12, 'Shugyo Blvd', 'Los Angeles', 'CA', 95100)));

insert into customer_tab
        select 'AB123', ref(p)
          from person_tab p where p.lastname = 'Seagal';
insert into customer_tab
        select 'DD492', ref(p)
          from person_tab p where p.lastname = 'Norris';
insert into customer_tab
        select 'SM493', ref(p)
          from person_tab p where p.lastname = 'Wallace';
insert into customer_tab
        select 'AC493', ref(p)
          from person_tab p where p.lastname = 'Van Damme';

commit work;
```

例に使用したintypeファイルnavdemo1.typのリストを次に示します。

```
case=lower
type location
type budoka
type customer
```

OTTが生成するヘッダー・ファイルnavdemo1.hは、#includeプリプロセッサ・ディレクティブの付いたプリコンパイラのコードにインクルードされます。

プリコンパイラ・コード内のコメントを読み込みます。プログラムは新しいbudokaオブジェクト(Jackie Chanのもの)を追加してから、*customer_tab*表のすべての顧客を表示します。

プリコンパイラ・ファイルnavdemo1.pcのリストを次に示します。

```
/*************************************************************************
 *
 * This is a simple Pro*C/C++ program designed to illustrate the
 * Navigational access to objects in the object cache.
 *
 * To build the executable:
 *
 *   1. Execute the SQL script, navdemo1.sql in SQL*Plus
 *   2. Run OTT: (The following command should appear on one line)
 *        ott intype=navdemo1.typ hfile=navdemo1.h outtype=navdemo1_o.typ
 *            code=c user=scott/tiger
 *   3. Precompile using Pro*C/C++:
 *        proc navdemo1 intype=navdemo1_o.typ
 *   4. Compile/Link (This step is platform specific)
 *
 *************************************************************************/

#include "navdemo1.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlca.h>

void whoops(errcode, errtext, errtextlen)
  int   errcode;
  char *errtext;
  int   errtextlen;
{
  printf("ERROR! sqlcode=%d: text = %.*s", errcode, errtextlen, errtext);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(EXIT_FAILURE);
}

void main()
{
  char *uid = "scott/tiger";

       /* The following types are generated by OTT and defined in navdemo1.h */
  customer *cust_p;                            /* Pointer to customer object */
  customer_ind *cust_ind;        /* Pointer to indicator struct for customer */
  customer_ref *cust_ref;            /* Pointer to customer object reference */
  budoka *budo_p;                               /* Pointer to budoka object */
  budoka_ref *budo_ref;               /* Pointer to budoka object reference */
  budoka_ind *budo_ind;            /* Pointer to indicator struct for budoka */

    /* These are data declarations to be used to insert/retrieve object data */
```

```
VARCHAR acct[21];
struct { char lname[21], fname[21]; int age; } pers;
struct { int num; char street[61], city[31], state[3], zip[11]; } addr;

EXEC SQL WHENEVER SQLERROR DO whoops(
  sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);

EXEC SQL CONNECT :uid;

EXEC SQL ALLOCATE :budo_ref;

/* Create a new budoka object with an associated indicator
 * variable returning a REF to that budoka as well.
 */
EXEC SQL OBJECT CREATE :budo_p:budo_ind TABLE PERSON_TAB
                RETURNING REF INTO :budo_ref;

/* Create a new customer object with an associated indicator */
EXEC SQL OBJECT CREATE :cust_p:cust_ind TABLE CUSTOMER_TAB;

/* Set all budoka indicators to NOT NULL.  We
 * will be setting all attributes of the budoka.
 */
budo_ind->_atomic = budo_ind->lastname = budo_ind->firstname =
  budo_ind->age = OCI_IND_NOTNULL;

/* We will also set all address attributes of the budoka */
budo_ind->addr._atomic = budo_ind->addr.num = budo_ind->addr.street =
  budo_ind->addr.city = budo_ind->addr.state = budo_ind->addr.zip =
    OCI_IND_NOTNULL;

/* All customer attributes will likewise be set */
cust_ind->_atomic = cust_ind->account_number = cust_ind->aperson =
  OCI_IND_NOTNULL;

/* Set the default CHAR semantics to type 5 (STRING) */
EXEC ORACLE OPTION (char_map=string);

strcpy((char *)pers.lname, (char *)"Chan");
strcpy((char *)pers.fname, (char *)"Jackie");
pers.age = 38;

/* Convert native C types to OTS types */
EXEC SQL OBJECT SET lastname, firstname, age OF :budo_p TO :pers;

addr.num = 1893;
strcpy((char *)addr.street, (char *)"Rumble Street");
strcpy((char *)addr.city, (char *)"Bronx");
strcpy((char *)addr.state, (char *)"NY");
strcpy((char *)addr.zip, (char *)"92510");

/* Convert native C types to OTS types */
EXEC SQL OBJECT SET :budo_p->addr TO :addr;

acct.len = strlen(strcpy((char *)acct.arr, (char *)"FS926"));

/* Convert native C types to OTS types - Note also the REF type */
EXEC SQL OBJECT SET account_number, aperson OF :cust_p TO :acct, :budo_ref;

/* Mark as updated both the new customer and the budoka */
```

```
EXEC SQL OBJECT UPDATE :cust_p;
EXEC SQL OBJECT UPDATE :budo_p;

/* Now flush the changes to the server, effectively
 * inserting the data into the respective tables.
 */
EXEC SQL OBJECT FLUSH :budo_p;
EXEC SQL OBJECT FLUSH :cust_p;

/* Associative access to the REFs from CUSTOMER_TAB */
EXEC SQL DECLARE ref_cur CURSOR FOR
  SELECT REF(c) FROM customer_tab c;

EXEC SQL OPEN ref_cur;

printf("¥n");

/* Allocate a REF to a customer for use in the following */
EXEC SQL ALLOCATE :cust_ref;

EXEC SQL WHENEVER NOT FOUND DO break;
while (1)
{
  EXEC SQL FETCH ref_cur INTO :cust_ref;

  /* Pin the customer REF, returning a pointer to a customer object */
  EXEC SQL OBJECT DEREF :cust_ref INTO :cust_p:cust_ind;

  /* Convert the OTS types to native C types */
  EXEC SQL OBJECT GET account_number FROM :cust_p INTO :acct;
  printf("Customer Account is %.*s¥n", acct.len, (char *)acct.arr);

  /* Pin the budoka REF, returning a pointer to a budoka object */
  EXEC SQL OBJECT DEREF :cust_p->aperson INTO :budo_p:budo_ind;

  /* Convert the OTS types to native C types */
  EXEC SQL OBJECT GET lastname, firstname, age FROM :budo_p INTO :pers;
  printf("Last Name: %s¥nFirst Name: %s¥nAge: %d¥n",
         pers.lname, pers.fname, pers.age);

  /* Do the same for the address attributes as well */
  EXEC SQL OBJECT GET :budo_p->addr INTO :addr;
  printf("Address:¥n");
  printf("  Street: %d %s¥n  City: %s¥n  State: %s¥n  Zip: %s¥n¥n",
         addr.num, addr.street, addr.city, addr.state, addr.zip);

  /* Unpin the customer object and budoka objects */
  EXEC SQL OBJECT RELEASE :cust_p;
  EXEC SQL OBJECT RELEASE :budo_p;
}

EXEC SQL CLOSE ref_cur;

EXEC SQL WHENEVER NOT FOUND DO whoops(
  sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);

/* Associatively select the newly created customer object */
EXEC SQL SELECT VALUE(c) INTO :cust_p FROM customer_tab c
          WHERE c.account_number = 'FS926';
```

```
  /* Mark as deleted the new customer object */
  EXEC SQL OBJECT DELETE :cust_p;

  /* Flush the changes, effectively deleting the customer object */
  EXEC SQL OBJECT FLUSH :cust_p;

  /* Associatively select a REF to the newly created budoka object */
  EXEC SQL SELECT REF(p) INTO :budo_ref FROM person_tab p
            WHERE p.lastname = 'Chan';

  /* Pin the budoka REF, returning a pointer to the budoka object */
  EXEC SQL OBJECT DEREF :budo_ref INTO :budo_p;

  /* Mark the new budoka object as deleted in the object cache */
  EXEC SQL OBJECT DELETE :budo_p;

  /* Flush the changes, effectively deleting the budoka object */
  EXEC SQL OBJECT FLUSH :budo_p;

  /* Finally, free all object cache memory and log off */
  EXEC SQL OBJECT CACHE FREE ALL;

  EXEC SQL COMMIT WORK RELEASE;

  exit(EXIT_SUCCESS);
}
```

プログラムの実行結果は次のとおりです。

```
Customer Account is AB123
Last Name: Seagal
First Name: Steven
Birthdate: 02-14-1963
Age: 34
Address:
  Street: 1825 Aikido Way
  City: Los Angeles
  State: CA
  Zip: 45300

Customer Account is DD492
Last Name: Norris
First Name: Chuck
Birthdate: 12-25-1952
Age: 45
Address:
  Street: 291 Grant Avenue
  City: Hollywood
  State: CA
  Zip: 21003

Customer Account is SM493
Last Name: Wallace
First Name: Bill
Birthdate: 02-29-1944
Age: 53
Address:
  Street: 874 Richmond Street
  City: New York
  State: NY
```

```
  Zip: 45100

Customer Account is AC493
Last Name: Van Damme
First Name: Jean Claude
Birthdate: 12-12-1965
Age: 32
Address:
  Street: 12 Shugyo Blvd
  City: Los Angeles
  State: CA
  Zip: 95100

Customer Account is FS926
Last Name: Chan
First Name: Jackie
Birthdate: 10-10-1959
Age: 38
Address:
  Street: 1893 Rumble Street
  City: Bronx
  State: NY
  Zip: 92510
```

関連項目:

intypeファイルの書式の詳細は、OTTコマンドラインを参照してください。

17.12 C言語の構造体の使用について

Oracle8以前は、Pro*C/C++のSQL SELECT文では、C言語の構造体を1つのホスト変数として指定できました。その場合、構造体の各メンバーは、リレーショナル表の1つのデータベース列に対応付けられます。つまり、各メンバーは問合せによって戻される選択リスト内の1つの項目を表します。

Oracle8*i*以上のバージョンでは、データベース内のオブジェクト型は、1つのエンティティであり、1つの項目として選択できます。Oracle7の表記法では、構造体はスカラー変数のグループなのか、それともオブジェクトなのかがあいまいです。

Pro*C/C++では、次の規則を利用してこのあいまいさを解消しています。

OTTを使用してC宣言が生成された場合にかぎり、C言語の構造体のホスト変数がオブジェクト型を表すとみなされます。そのため、型記述はPro*C/C++へのINTYPEオプションで指定される型ファイルに表示されます。他のすべてのホスト構造体は、データベースに同じ名前のデータ型が存在する場合でも、Oracle7構文が使用されているものとみなされます。

したがって、既存の構造体ホスト変数の型と同じ名前を持つ新しいオブジェクト型を使用する場合は、Pro*C/C++ではINTYPEファイル内のオブジェクト型定義が使用されることに注意してください。これは、コンパイル・エラーの原因となる場合があります。この修正には、既存のホスト変数の型を名前変更するか、またはOTTを使用してオブジェクト型に新しい名前を付けます。

前述の規則は、OTT生成のデータ型に対して別名指定されるユーザー定義のデータ型にまで広く適用されます。例として、*emptype*がヘッダー・ファイルdbtypes.h内でOTTによって生成された構造体であり、Pro*C/C++プログラムに次の文を組み込んだ場合を考えます。

```
#include <dbtypes.h>
typedef emptype myemp;
```

```
myemp *employee;
```

変数*employee*を表す型の名前*myemp*は、データベース内で定義されたあるオブジェクト型を表すOTT生成の型の名前*emptype*に対して別名指定されます。これによって、Pro*C/C++では、変数*employee*はオブジェクト型を表すものとみなされます。

前述の規則は、OTT生成の型を持つC言語の構造体や、OTT生成の型に対して別名指定されているC言語の構造体を、オブジェクト型以外の型のデータのフェッチに使用できないという意味ではありません。単に、Pro*C/C++は自動的にそうした構造体を拡張しないということです。ユーザーは自由に一般的な構文を使用して、構造体の個々のフィールドを単一データベースの列の選択や更新に使用できます。

17.13 REFの使用について

REF型はオブジェクト自体ではなく、オブジェクトの参照を示します。REF型はリレーショナル列のみでなく、オブジェクト型の属性としても指定できます。

17.13.1 REFのC言語の構造体の生成

オブジェクト型のREFのC言語での表現は、OTTにより型の変換中に生成されます。たとえば、データベース内のユーザー定義型PERSONの参照は、C言語では「Person_ref」型で示されます。正確な型名は型変換時に有効なOTTオプションで決定されます。OTTにより生成された型ファイルは、Pro*C/C++のINTYPEプリコンパイラ・オプションで指定する必要があります。また、OTTにより生成されたヘッダーは、#includeを使用してPro*C/C++プログラムに組み込む必要があります。このスキームにより、REF型に対する適切なタイプ・チェックがPro*C/C++のプリコンパイル中に間違いなく実行されます。

REF型では、OTTで特殊なインジケータ構造体を生成する必要がありません。かわりに、2バイトの符号付きスカラー・インジケータが使用されます。

17.13.2 REFの宣言

Pro*C/C++でREF型を表すホスト変数は、該当するOTT生成の型へのポインタとして宣言する必要があります。

オブジェクト型とは異なり、REF型を表す標識変数は、2バイトの符号付きスカラー型OCIIndとして宣言されます。標識変数は本来オプションですが、Pro*C/C++で宣言された各ホスト変数に対してそれぞれ1つずつ指定するようにプログラミングしてください。

17.13.3 埋込みSQLでのREFの使用

REF型は、オブジェクト・キャッシュに格納されています。ただし、REF型を表すインジケータはスカラーであるため、キャッシュには割り当てられていません。このインジケータは通常ユーザー・スタックに格納されています。

REF型を表すホスト構造体を埋込みSQL文で指定する前に、EXEC SQL ALLOCATEコマンドを使用してオブジェクト・キャッシュ内の領域を割り当ててください。使用後は、EXEC SQL FREEまたはEXEC SQL CACHE FREE ALLコマンドを自由に使用できます。

スカラー標識変数用のメモリーはオブジェクト・キャッシュに割り当てられていないため、REF型を表すインジケータはALLOCATEコマンドおよびFREEコマンドには使用できません。OCIIndとして宣言されたスカラー・インジケータはプログラム・スタックに格納されています。ALLOCATE文を指定すれば、実行時に、指定されたホスト変数のための領域がオブジェクト・キャッシュに割り当てられます。ナビゲーショナル・アクセス用インタフェースでは、C割当てではなく、EXEC SQL GETとEXEC SQL SETを使用してください。

Pro*C/C++では、関連するSQL文および埋込みPL/SQLブロックでのREFホスト変数の指定がサポートされています。

関連項目

- ナビゲーショナル・アクセス用インタフェース

17.14 OCIDate、OCIString、OCINumberおよびOCIRawの使用について

これらのOCI型はそれぞれ、日付、可変長のヌルで終了する文字列、Oracleでの数、および可変長のバイナリ・データを表す新しいC言語の表現です。これらの型は、いくつかの面でこれまでのC言語の数量表現よりも機能的になっています。たとえば、OCIDate型はクライアント側のルーチンが日付演算を実行する準備をします。これは以前のリリースではサーバーでのSQL文を必要としていました。

17.14.1 OCIDate、OCIString、OCINumber、OCIRawの宣言

OCI*型は、OTT生成の構造体でオブジェクト型の属性として表示され、Pro*C/C++プログラムではオブジェクト型の一部として使用します。オブジェクト型として使用しない場合に、初心者レベルのC言語およびPro*C/C++ユーザーは、これらの型のホスト変数を単独で宣言しないでください。経験豊富なPro*C/C++ユーザーは、これらの型の高い機能性を活かすように、それぞれのCホスト変数を宣言してもかまいません。ホスト変数はこれらの型へのポインタ`OCIString *s`などとして宣言される必要があります。対応付けられた(オプションの)インジケータは、2バイトの符号付きスカラー、つまり`OCIInd s_ind`などとして宣言されます。

17.14.2 埋込みSQLでのOCI型の使用

これらの型のホスト変数のための領域は、EXEC SQL ALLOCATEを使用してオブジェクト・キャッシュに割り当てられます。これらの型を表すスカラー標識変数は、ALLOCATEおよびFREEコマンドには使用できません。このようなインジケータは、スタック上で静的に割り当てるか、またはヒープ上で動的に割り当てます。領域の割当ては、EXEC SQL FREE文またはEXEC SQL CACHE FREE ALL文を使用して解除できます。また、セッションの終わりには自動的に解除されます。

関連項目

- ナビゲーショナル・アクセス用インタフェース

17.14.3 OCI型の操作

年、月、日、時など、様々な日付コンポーネントごとに個別フィールドを持つ構造型`OCIDate`を除き、他のOCI型はカプセル化されているため外部ユーザーには見えません。現在、VARCHARのような既存のC言語のデータ型はPro*C/C++で処理されますが、この方法とは対照的にOCIヘッダー・ファイル`oci.h`をインクルードし、その関数を使用してDATE算術を実行したり、これらの型と*int*や *char*などのC言語固有の型の間で変換したりできます。

17.15 Pro*C/C++の新しいデータベース型の概要

表17-2に、オブジェクト・サポートのための新たなデータベース型を示します。

表17-2 Pro*C/C++での新しいデータベース型の使用

操作データベース型	DECLARE	ALLOCATE	FREE	MANIPULATE
オブジェクト型	**ホスト:** OTT で生成された C 言語の構造体への	連想アクセス用インタフェース:	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解	C 言語のポインタを間接参照(DEREF)して各属性を取得します。操作方

操作データベース型	DECLARE	ALLOCATE	FREE	MANIPULATE
	ポインタ **インジケータ:** OTT で生成されたインジケータ構造体へのポインタ	EXEC SQL ALLOCATE ナビゲーショナル・アクセス用インタフェース: EXEC SQL OBJECT CREATE ... EXEC SQL OBJECT DEREF ホスト変数およびインジケータのためのメモリーをオブジェクト・キャッシュに割り当てます。	放するか、セッションの終わりに自動的に解放します。	法は属性の型によって異なります(次を参照)。
コレクション・オブジェクト・タイプ (NESTED TABLE および可変長配列)	**ホスト:** OTT で生成された C 言語の構造体へのポインタ **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	OCIColl*関数(oci.h に定義)を使用して、各要素を取得または設定します。コレクションも参照してください。
REF	**ホスト:** OTT で生成された C 言語の構造体へのポインタ **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	EXEC SQL OBJECT DEREF を使用します。 ナビゲーショナル・アクセス用インタフェースで EXEC SQL OBJECT SET/GET を使用します。
LOB	**ホスト:** OCIBlobLocator *、OCIClobLocator *または OCIBfileLocator * **インジケータ:**	EXEC SQL ALLOCATE を使用してホスト変数用のメモリーをユーザー・ヒープ内で割り当てます。 malloc()	EXEC SQL FREE を使用して解放するか、すべての Pro*C/C++接続のクローズ時に自動的に解放します。EXEC SQL CACHE FREE ALL では、オブジェクトの LOB 属性のみ解放されます。	または、dbms_lob パッケージの埋込み PL/SQL ストアド・プロシージャを使用するか、あるいは oci.h に定義されている OCILob*関数を使用します。 LOB も参照してください。

操作データベース型	DECLARE	ALLOCATE	FREE	MANIPULATE
	OCIInd			
注意: Pro*C/C++では、これらの型のホスト配列は、SQL のバルク・フェッチまたはバルク挿入操作で宣言および使用できます。	-	-	-	-

表17-3に、Pro*C/C++での新たなC言語のデータ型の利用方法を示します。

表17-3 Pro*C/C++での新たなC言語のデータ型の使用

操作C言語のデータ型	DECLARE	ALLOCATE	FREE	MANIPULATE
OCIDate	**ホスト:** OCIDate * **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	(1)oci.h に定義されたOCIDate*関数を使用します。 (2)EXEC SQL OBJECT GET/SET を使用します。または (3)oci.h に定義されている OCINumber*関数を使用します。
OCINumber	**ホスト:** OCINumber * **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	(1)EXEC SQL OBJECT GET/SET を使用します。または (2)oci.h に定義されている OCINumber*関数を使用します。
OCIRaw	**ホスト:** OCIRaw * **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	oci.h に定義されている OCIRaw*関数を使用します。

操作C言語のデータ型	DECLARE	ALLOCATE	FREE	MANIPULATE
OCIString	**ホスト:** OCIString * **インジケータ:** OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放するか、セッションの終わりに自動的に解放します。	(1)EXEC SQL OBJECT GET/SET を使用します。または (2)oci.h に定義されている OCIString*関数を使用します。
注意: Pro*C/C++では、これらの型のホスト配列は、SQL のバルク・フェッチまたはバルク挿入操作で使用できない場合があります。	-	-	-	-

Oracle8での新しいデータ型は、REF、BLOB、NCLOB、CLOBおよびBFILEです。これらの型は、オブジェクト内またはリレーショナル列に使用することができます。どちらの場合も、Cバインドに従ってホスト変数にマップされます。

関連項目:

Cバインドについては、Pro*C/C++の新しいデータベース型の概要を参照してください。

17.16 動的SQLでのOracleデータ型使用の制限

現在Pro*C/C++では、動的SQL方法: 1、2、3および4(ANSIおよびOracle)の異なる型をサポートしています。

動的SQL方法1、2および3を使用すると、前述のPro*C/C++拡張機能、つまり新しいオブジェクト型、REF、NESTED TABLE、可変長配列、NCHAR、NCHARの可変長配列およびLOB型などをすべて処理できます。

従来のの動的SQL方法4は、基本的にリリース8.0より前のPro*C/C++でサポートされるOracle型に制限されています。NCHAR、NCHAR可変幅およびLOBデータ型のホスト変数はサポートされています。動的SQL方法4は、オブジェクト型、NESTED TABLE、可変長配列およびREF型の処理には使用できません。

そのかわり新しいアプリケーションにはすべて、Oracle8*i*で導入されたすべてのデータ型をサポートするANSI動的SQL方法4を使用してください。

関連項目

- Oracle動的SQL
- ANSI動的SQL
- Oracle動的SQL: 方法4

18 コレクション

この章では、コレクションと呼ばれる他の種類のオブジェクト型と、Pro*C/C++でのコレクションの使用方法を説明します。コレクションとその要素にアクセスする方法を示します。この章のトピックは、次のとおりです:

- コレクション
- コレクションの記述子
- OBJECT GETおよびSET
- COLLECTION文
- コレクション・サンプル・コード

18.1 コレクション

コレクション・オブジェクト・タイプには、*NESTED TABLE*および*VARRAY*の2種類があります。

コレクション型は、リレーショナル列に指定できるのみでなく、オブジェクト型の属性としても指定できます。すべてのコレクションは、データベース内では必ず名前付きのオブジェクト型である必要があります。VARRAYの場合は、最初にデータベース内に名前付きの型を作成して、希望する配列の要素型と最大配列サイズを指定する必要があります。

Oracleでは、ネスト・レベルを単一にするという制限がなくなっています。Oracleではオブジェクトの複数レベルのコレクションがサポートされるため、NESTED TABLEとVARRAYを複数レベルで使用できます。

18.1.1 ネストされた表

NESTED TABLEは、列内での要素と呼ばれる行の集まり(コレクション)です。データベース表の各行には、そのような要素が多数あります。簡単な例として、各従業員の作業リストがあります。この場合、多対1リレーションシップを1つの表に格納でき、従業員と作業の表を結合する必要はありません。

NESTED TABLEは、次の点でC言語の配列およびC++配列と異なります。

- 配列には一定の上限がありますが、NESTED TABLEには制限がありません(インデックス最大値がありません)。
- 配列には連続する添字があり、稠密です。NESTED TABLEは稠密または疎密です。NESTED TABLEがプログラムの配列に取り出されると、間隔はスキップされ、間隔のない稠密な配列が作成されます。

CREATE TYPE文を使用して表タイプを定義します。表タイプは、リレーショナル表の1つ以上の列でその他のオブジェクト型の中にネストできます。

たとえば、組織の各部門に複数のプロジェクトを格納する場合を考えます。

```
CREATE TYPE project_type AS OBJECT (
    pno             CHAR(5),
    pname           CHAR(20),
    budget          NUMBER(7,2)) ;

CREATE TYPE project_table AS TABLE OF project_type ;

CREATE TABLE depts (
    dno             CHAR(5),
    dname           CHAR(20),
    budgets_limit   NUMBER(15,2),
    projects        project_table)
    NESTED TABLE projects STORE AS depts_projects ;
```

18.1.2 VARRAY

VARRAY型を作成する場合、NESTED TABLEとは異なり、要素の最大数を指定する必要があります。NESTED TABLEでは稠密または疎密にできますが、VARRAYは疎密にはできません。VARRAYおよびNESTED TABLEの要素は、両方とも0から始まります。

次のようなCREATE TYPE文で、VARRAYを作成します。

```
CREATE TYPE employee AS OBJECT
(
  name   VARCHAR2(10),
  salary NUMBER(8,2)
) ;
CREATE TYPE employees AS VARRAY(15) OF employee ;
CREATE TYPE department AS OBJECT
(
  name VARCHAR2(15),
  team employees
) ;
```

VARRAYは、リレーショナル表の列またはオブジェクト型の属性として使用します。この場合、各チームについて最大15レコード(チームの名前を含みます)で構成されるリレーショナル表と比較して、記憶域を節約できます。

18.1.3 C言語およびコレクション

C言語またはC++プログラムでは、NESTED TABLEはコレクションのインデックス値0から読込みが開始されます。配列にNESTED TABLEを書き込むと、NESTED TABLEの要素は配列インデックス0で格納されます。配列に格納されたNESTED TABLEが疎密(インデックスに間隔があります)の場合、間隔はスキップされます。配列がNESTED TABLEに読み込まれると、間隔が再作成されます。

C言語またはC++プログラムでは、VARRAYは配列に書き込まれます(インデックス0から開始します)。VARRAYに再び読み込まれると、要素はインデックス0から同じ順序でリストアされます。このため、配列を使用するとコレクションに簡単にアクセスできます。

18.2 コレクションの記述子

NESTED TABLEのC言語のデータ型は、OCITableへのポインタです。VARRAYの場合は、OCIArrayへのポインタになります。(両方ともOCICollへのポインタのサブタイプです。)OTT(Object Type Translator)ユーティリティを使用して、アプリケーション・コードにインクルードするヘッダー・ファイルにtypedefを生成します。

コレクションのホスト構造体は記述子で、これを介してコレクションの要素にアクセスできます。この記述子には、実際のコレクション要素は保持されていませんが、その要素へのポインタが格納されています。記述子とその関連要素のためのメモリーは、オブジェクト・キャッシュで入手できます。

オブジェクト型における通常の手順に従って、OTT生成の型ファイルを、Pro*C/C++に対するINTYPEプリコンパイラ・オプション内と、#includeプリプロセッサ・ディレクティブを使用してPro*C/C++プログラムにインクルードされたOTT生成のヘッダー内に指定する必要があります。このスキーマにより、コレクション・オブジェクト・タイプに対する適切なタイプ・チェックがプリコンパイル中に間違いなく実行されます。

ただし、他のオブジェクト型とは異なり、コレクション・オブジェクト・タイプは、OTTによって生成される特別なインジケータ構造体を必要としません。かわりにスカラー・インジケータが使用されます。その理由は、アトミックNULLインジケータのみで十分にコレクション型全体がNULLかどうかがわかるからです。個々のコレクション要素のNULL状態は、(オプションで)各要素に対応する別個のインジケータで表すことができます。

関連項目

- Object Type Translator

18.2.1 ホスト変数と標識変数の宣言

コレクション・オブジェクト・タイプを表すホスト変数は、その他のオブジェクト型の場合と同様に、該当するOTT生成の型へのポインタとして宣言する必要があります。

ただし、コレクション・オブジェクト・タイプ全体を表す標識変数は、その他のオブジェクト型の場合とは異なり、2バイトの符号付きスカラー型OCIIndとして宣言されます。標識変数はオプションで設定しますが、Pro*C/C++で宣言された各ホスト変数に対してそれぞれ1つずつ指定するようにプログラミングすることをお薦めします。

18.2.2 コレクションの操作について

コレクションを操作する方法は2つあります。コレクションを自律型エンティティとして扱い、コレクションの要素へのアクセスが発生しない場合と、コレクションの要素に対するアクセス、追加および切捨てなどが発生する場合です。

18.2.2.1 自律型コレクション・アクセス

Cコレクション記述子(OCITableまたはOCIArray)を使用した場合、コレクション全体を1つのエンティティとして割り当てること以外はできません。OBJECT GET埋込みSQL文を使用すると、コレクションがCホスト変数記述子にバインドされます。OBJECT SET文では、逆にCホスト記述子がコレクションにバインドされます。

1つの文中で複数のコレクションを1つの互換Cコレクション記述子にバインドしたり、コレクションをCコレクション記述子にバインドしている文中で他のスカラーにバインドすることもできます。

関連項目

- オブジェクト属性とC言語のデータ型の変換

18.2.2.2 コレクション要素アクセス

Cコレクション記述子は、コレクションの要素にアクセスするために使用します。記述子には、始点およびエンドポイントなどのコレクションの内部属性およびその他の情報が含まれます。

要素のスライスは、互換データ型を持つホスト配列にバインドされます。コレクションのスライスとは、開始インデックスと終了インデックス間の内容のことです。スライスは配列にマップされます。配列の次元がスライス要素数よりも大きいことがあります。

スカラーのバインドは、ホスト配列の次元を1にすること、またはオプションのFOR句が1に評価されることと同じことです。

18.2.3 アクセスのルール

アクセスのルールは、自律型アクセスと要素アクセスで異なります。

18.2.3.1 自律型アクセス

- コレクションは1つの全体として扱われるため、FOR句は使用できません。
- NESTED TABLEとVARRAYの定義方法が異なるため、それらの間で代入を行うことはできません。
- 1つの文中で、複数のコレクションを様々に組み合せてCコレクション記述子に代入することができます。1つの文中で、コレクションをCコレクション記述子に代入したり、他のスカラー・データ型をバインドしたりできます。

18.2.3.2 要素アクセス

- FOR句を使用できます。省略すると、配列サイズの最小値が実行の繰返し回数になります。

- 一度に複数のコレクションにアクセスすることはできません。

> 注意:
>
> FOR 句の変数では、処理する配列の要素数を指定します。この数は、最小の配列次元を超えないように設定します。内部では、値は符号なしの数量として扱われます。符号付きのホスト変数を使用して負の値を渡すと、予測不能な動作が発生する原因となります。

18.2.4 インジケータ変数

アクセス方法ごとに標識変数の使用方法が異なります。

18.2.4.1 自律型バインド

1つの標識変数には、コレクションのNULL状態が1つのエンティティとして格納されます。各要素のNULL状態は格納されていません。

18.2.4.2 要素バインド

標識変数には、要素がNULLであるかどうかが格納されます。コレクション・データのスライスが、独自のインジケータ配列のホスト配列にバインドされている場合は、インジケータ配列には各要素のNULL状態がスライスに格納されます。

コレクション要素型がユーザー定義オブジェクト型の場合は、ホスト変数に対応付けられた標識変数に、オブジェクトおよびその属性のNULL状態が格納されます。

18.3 OBJECT GETおよびSET

OBJECT SETおよびOBJECT GETというナビゲーショナル文を使用すると、コレクション属性および定義したオブジェクト型の取出しおよび更新を行うことができます。

次の場合にオブジェクト型の要素に対してOBJECT GET文を実行すると、オブジェクトのすべての属性がホスト変数に取り出されます。

```
'*' | {attr [, attr]}  FROM
```

句が省略されるかOBJECT GET * FROMが使用される場合。

```
EXEC SQL [AT [:]database]
   OBJECT GET [  '*' | {attr [,attr]}  FROM]
      :object [[INDICATOR] :object_ind]
         INTO {:hv [[INDICATOR] :hv_ind]
            [,:hv [[INDICATOR] :hv_ind]]} ;
```

次の場合にOBJECT SET文を実行すると、オブジェクトのすべての属性に対してホスト変数を使用した更新が行われます。

```
'*' | {attr, [, attr]}  OF
```

句が省略されるかOBJECT SET * OFが使用される場合。

```
EXEC SQL [AT [:]database]
   OBJECT SET [  '*' | {attr [, attr]} OF]
     :object [INDICATOR] :object_ind]
        TO {:hv [[INDICATOR] :hv_ind]
```

```
        [,:hv [[INDICATOR] :hv_ind]]} ;
```

この表は、これら2つの文でオブジェクトとコレクション型をどのようにマッピングするかを示しています。

表18-1 オブジェクト属性およびコレクション属性

属性タイプ	表現	ホスト・データ型
オブジェクト	OTT 生成の構造体	OTT 構造体へのポインタ
コレクション	OCIArray、OCITable (OCIColl)	OCIArray *、OCITable * (OCIColl *)

オブジェクトまたはコレクションは、バインドされる属性と型互換性を持つ必要があります。コレクション属性の型互換性は両方がVARRAYまたはNESTED TABLEのいずれかで、その要素型に互換性がある場合にかぎり保たれます。

次の表は2つのコレクション型の要素でどのように互換性を保てるかを示しています。

表18-2 コレクションおよびホスト配列で可能な型変換

コレクション要素の型	表現	ホスト・データ型
CHAR、VARCHAR、VARCHAR2	OCIString	STRING、VARCHAR、CHARZ、OCIString
REF	OCIRef	OCIRef
INT、INTEGER、SMALLINT	OCINumber	INT、SHORT、OCINumber
NUMBER、NUMERIC、REAL、FLOAT、DOUBLE PRECISION	OCINumber	INT、FLOAT、DOUBLE、OCINumber
DATE	OCIDate	STRING、VARCHAR、CHARZ、OCIDate

REFが参照するオブジェクトは、バインドされるREFと型互換性を持つ必要があります。

これらの表の場合、OBJECT GETでは「記述」列に指定された形式が使用され、左端のデータベース型から「ホスト・データ型」列の形式を使用する内部データ型に変換されます。OBJECT SETでは逆の変換が行われます。

明示的タイプ・チェックはサポートされません

プリコンパイラでは、コレクション要素データ型とホスト変数データ型間のバインド時の、明示的タイプ・チェックはサポートされません。タイプ・チェックは実行時に行われます。

関連項目

- オブジェクト属性とC言語のデータ型の変換

18.4 COLLECTION文

この項では、COLLECTION文について説明します。

18.4.1 COLLECTION GET

用途

COLLECTION GET文は、OBJECT GET文に似ていますが、コレクションを対象としています。コレクション要素の取得、現行のスライスの設定、要素のC言語のデータ型への変換(必要な場合)を行います。

構文

```
EXEC SQL [AT [:]database] [FOR :num]
   COLLECTION GET :collect [[INDICATOR] :collect_ind]
      INTO :hv [[INDICATOR] :hv_ind] ;
```

変数

num (IN)

要求する要素の数。この句を省略すると、コレクションから取得する要素の数は、ホスト変数の配列サイズ(スカラーは1)により決定されます。

collect (IN)

ホスト変数Cコレクション記述子。

collect_ind (IN)

コレクションのNULL状態を戻すオプションの標識変数。

hv (OUT)

コレクション要素値を受け取るホスト変数。

hv_ind (OUT)

スカラーまたは配列にスライスの各要素の状態が格納されている場合、hvのNULL状態を戻すオプションの標識変数。

使用上の注意

最後のCOLLECTION GETから実際に戻される要素の数は、sqlca.sqlerrd[2]に設定されます(すべてのGET累計ではありません)。

スライスのエンドポイントのいずれかまたは両方がコレクションの境界を超過したときは、戻される要素数が要求した数よりも少ないことがあります。これは次の場合に発生します。

- コレクション記述子が、正しい構文のALLOCATE文で初期化されなかった場合、NULLの場合、その他の理由で無効な場合。
- コレクションがNULLの場合。対応付けられるインジケータは-1になります。
- コレクションが空(要素がない)の場合。
- コレクションに残っている要素数を超える要素が要求された場合。
- COLLECTION TRIM文が実行された結果、現行のスライスで終了インデックスが開始インデックスよりも前になっている場合。

Cコレクション記述子が適切に初期化されなかった場合は、エラーになります。前述のリスト以外の場合は、ORA-01403: no data foundというエラーが発生します。この場合、エラー発生前に正常に取得できた要素の合計数は、sqlca.sqlerrd[2]に格納されたままです。

最初のGETまたはRESET後の最初のGETにより、スライスは次のようになります。

- スライスの終了インデックスは、検出された最後の要素のインデックスになります。これは、要求する要素数によって変わります。コレクションに要求を満たせるだけの要素が残っていない場合、最後のインデックスはコレクションの最後のインデックスになります。

引き続きGETを実行すると、スライスのインデックスは次のようになります。

- 直前のスライスのエンドポイントの後に最初に検出された要素のインデックスが、開始インデックスになります。直前のスライスのエンドポイントの後に要素が残っていない場合、開始インデックスはコレクションの最後の要素のインデックスになります。
- 次のスライスの終了インデックスは、検出された最後の要素のインデックスになります。これは、要求する要素数によって変わります。直前のスライスで指定された位置で、要求を満たせるだけの要素がコレクションに残っていない場合、終了インデックスはコレクションの最後のインデックスになります。

関連項目

- sqlerrd

18.4.2 COLLECTION SET

用途

COLLECTION SET文は、OBJECT SET文に似ていますが、コレクションの要素値の更新に使用します。現行のスライスの要素は、C固有型からOracleデータ型に変換されます。

構文

```
EXEC SQL [AT [:]database] [FOR :num]
   COLLECTION SET :collect [[INDICATOR] :collect_ind]
      TO :hv [[INDICATOR] :hv_ind] ;
```

変数

num (IN)

このオプションのスカラー値は、スライス内で更新される要素の最大数です。この句を省略すると、コレクションから更新される要素数は、ホスト変数の配列サイズ(スカラーは1)により決定されます。

collect (OUT)

ホスト変数Cコレクション記述子。

collect_ind (OUT)

コレクションのNULL状態を決定するオプションの標識変数。

hv (IN)

コレクション内で更新される値を含むホスト変数。

hv_ind (IN)

ホスト変数のNULL状態を表す、対応付けられた標識変数。

使用上の注意

次の制限があります。

- COLLECTION GETは、COLLECTION SETの前に実行する必要があります。
- スライスの開始インデックスおよび終了インデックスは、常に変わりません。コレクションに格納されている要素数が、現行

のスライスに格納できる要素数より少ない場合も変わりません。SET文では、スライスのエンドポイントは変更されません 。現行のスライスの要素のみが変更されます。

- COLLECTION SETでは、現行のスライスの要素のみが更新されます。COLLECTION SET文を使用して、新しい要素をコレクションに追加することはできません。
- 現行のスライスに含まれる要素の数を超える要素をSET文で更新すると、既存のスライスに格納されている要素のみが更新されます。スライスのエンドポイント外の残りの要素は更新されず、ホスト変数で指定されたその他の値は使用されません。

オプションのFOR句で指定したホスト変数または値numの次元により、コレクションで更新を要求できる要素の最大数が決まります。

変数sqlca.sqlerrd[2]により、直前のSET文で正常に更新された要素の数が戻されます(累計ではありません)。次のような場合は、GET文と同様に、設定要求数よりも少ないことがあります。

- Cコレクション記述子が、構文の正しいALLOCATE文で正常に初期化されなかった場合、NULLの場合、またはその他の理由で無効の場合。
- コレクションが空の場合。
- コレクションの現行のスライスの位置で、コレクションの残りの部分の要素数が、設定要求数よりも少なかった場合。
- 現行のスライスのエンドポイントを超えた場合。既存のスライスの要素数以上の要素を設定すると、この状態になります。
- コレクションに対してTRIMが実行され、コレクションの終了インデックスの最大値が現行のスライスの開始インデックスを下回った場合。

COLLECTION GETまたはSETの直後にCOLLECTION SETを実行した場合は、既存のスライスの要素の値のみが更新されます。COLLECTION SETの直後にCOLLECTION GETを実行すると、すでに説明したように次のスライスに移ります。

18.4.3 COLLECTION RESET

用途

コレクション・スライスのエンドポイントをコレクションの最初にリセットします。

構文

```
EXEC SQL [AT [:]database]
   COLLECTION RESET :collect
      [ [INDICATOR] :collect_ind] ;
```

変数

collect (IN/OUT)

エンドポイントをリセットするコレクション。

collect_ind

コレクションのNULL状態を決定するオプションの標識変数。

使用上の注意

指定したコレクションがNULLまたは無効の場合は、エラーが発生します。

COLLETION RESETはコレクションのサイズまたは内容には影響しません。

18.4.4 COLLECTION APPEND

用途

コレクションの最後に要素のセット(1つ以上)を追加します。コレクションのサイズが増加します。

構文

```
EXEC SQL [AT [:]database] [FOR :num]
   COLLECTION APPEND :src [[INDICATOR] :src_ind]
      TO :collect [[INDICATOR] :collect_ind] ;
```

変数

num (IN)

追加する要素数が格納されたスカラー。指定しない場合は、配列サイズsrcが追加する要素数になります。

src (IN):

コレクションに追加する要素のスカラーまたは配列。

src_ind (IN)

追加する要素のNULL状態を決定するオプションの標識変数(スカラーまたは配列)。

collect (IN/OUT)

要素を追加するコレクション。

collect_ind (IN)

コレクションのNULL状態を決定するオプションの標識変数。

使用上の注意

要素は一度に1つずつ追加されます(コレクションのサイズは1ずつ増加し、データがその要素にコピーされます)。

変数sqlca.sqlerrd[2]により、最後のAPPENDで正常に追加された要素数が戻されます(累計ではありません)。コレクションの上限を超えて要素を追加したり、NULLコレクションを追加したりすると、エラーが発生します。上限以内の要素のみが追加されます。

18.4.5 COLLECTION TRIM

用途

コレクションの最後から要素を削除します。

構文

```
EXEC SQL [AT [:]database]
   COLLECTION TRIM :num
      FROM :collect [[INDICATOR] :collect_ind] ;
```

変数

num (IN)

削除する要素数を示すホスト・スカラー変数。最大許容値は2GBです。

collect (IN/OUT)

切り捨てるコレクション。

collect_ind (IN)

コレクションのNULL状態を決定するオプションの標識変数。

使用上の注意

次の制限が適用されます。

- FOR句は使用できません。
- numの最大値は2GBです(4バイト符号付きバイナリ変数の最大数)。
- numにインジケータを使用することはできません。

numがコレクションのサイズよりも大きい場合、エラーが戻されます。TRIM文で現行のスライスから要素を削除すると、警告が戻されます。

18.4.6 COLLECTION DESCRIBE

用途

コレクションについての情報が戻されます。

構文

```
EXEC SQL [AT [:]database]
   COLLECTION DESCRIBE :collect [[INDICATOR] :collect_ind]
      GET attribute1 [{, attributeM}]
         INTO :hv1 [[INDICATOR] :hv_ind1] [{, hvN [[INDICATOR] :hv_indN]}] ;
```

attributeNは次のとおりです。

```
DATA_SIZE | TYPECODE | DATA_TYPE | NUM_ELEMENTS
   | PRECISION | SCALE | TYPE_NAME | TYPE_SCHEMA | SIZE | TABLE_SIZE
```

変数

collect (IN)

ホスト変数Cコレクション記述子。

collect_ind (IN)

コレクションのNULL状態を含むオプションの標識変数。

hv1 .. hvN (OUT)

情報が格納される出力ホスト変数。

hv_ind1 .. hv_indN (OUT)

出力ホスト変数の標識変数。

使用上の注意

次の制限が適用されます。

- コレクションをNULLにはできません。
- ホスト変数型は、戻される属性の型と互換性を持つ必要があります。
- 属性の標識変数は、テキストの切捨てが行われるTYPE_NAMEおよびTYPE_SCHEMA属性値でのみ必要です。
- FOR句は使用できません。

- 変数sqlca.sqlerrd[2]には、エラーなしで正常に取り出された属性の数が戻されます。DESCRIBE文でエラーが発生した場合、sqlca.sqlqerrd[2]には、エラー発生前に戻された属性の数が格納されます。エラー発生時の属性数より1つ少なくなっています。

次の表は、属性、説明、および取り出される属性のC言語のデータ型を示しています。

表18-3 COLLECTION DESCRIBEの属性

属性	説明	C言語のデータ型	注意
DATA_SIZE	型属性の最大サイズ。戻される長さは、文字列のバイト長です。NUMBER の場合は 22 です。	unsigned short	オブジェクトまたはオブジェクト REF 要素では無効です。
TYPECODE	OCI 型コード。	OCITypeCode	-
DATA_TYPE	コレクション項目の内部数値型コード。	unsigned short	-
NUM_ELEMENTS	VARRAY の最大要素数。	unsigned int	VARRAY 型にのみ有効です。
PRECISION	数値型属性の精度。戻される値が 0 の場合、記述される項目は初期化されず、データ・ディクショナリは NULL になります。	unsigned char	NUMBER 型の要素にのみ有効です。
SCALE	数値型属性のスケール。戻される値が-127 の場合、記述される項目は初期化されず、データ・ディクショナリは NULL になります。	signed char	NUMBER 型の要素にのみ有効です。
TYPE_NAME	型の名前を含む文字列。オブジェクト型の場合、その名前が戻されます。REF の場合、REF によって参照されるデータ型の名前が戻されます。使用できる外部データ型は CHARZ、STRING および VARCHAR です。	char*	オブジェクトおよびオブジェクト REF 要素にのみ有効です。
TYPE_SCHEMA	型を作成するスキーマ名。使用できる外部データ型は CHARZ、STRING および VARCHAR です。	char*	オブジェクトおよびオブジェクト REF 要素にのみ有効です。
SIZE	コレクションに実際に格納されている要素数。NESTED TABLE の場合、SIZE により空の要素が格納されます。TRIM 文を実行すると、切り捨てられた要素数のみコレクションの SIZE が減少します。	signed int	-

属性	説明	C言語のデータ型	注意
TABLE_SIZE	NESTED TABLE の要素の数。間隔は含みません。	signed int	NESTED TABLE にのみ有効です。

18.4.6.1 表について

Pro*C/C++の外部データ型では、属性TYPE_NAMEおよびTYPE_SCHEMAのCHARZ、STRINGおよびVARCHARのみがサポートされます。

SIZEおよびTABLE_SIZEを除き、DESCRIBE属性は、すべてコレクションの要素型に依存し、コレクションの特定のインスタンスには依存しません。一方、SIZEおよびTABLE_SIZE属性は、値がコレクションの特定のインスタンスに完全に依存しています。割り当てられたコレクション記述子が再利用され、同じコレクションの異なるインスタンスが参照される場合、SIZEまたはTABLE_SIZEの値はコレクションのインスタンスごとに変わります。NUM_ELEMENTSはコレクション型(この例ではVARRAY)の属性で、コレクション要素型ではありません。また、コレクションの特定のインスタンスには依存しません。

18.4.7 コレクションを使用する場合の規則

- ホスト変数コレクション記述子は、常に明示的に割り当てる必要があります。
- メタデータ(コレクションおよびその要素型についての、データベースの内部Oracleデータ)は、ALLOCATEの実行中に収集されます。ALLOCATEが実行された接続がクローズしているとき、またはALLOCATEの実行後に型が変更されたときは、メタデータは無効になります。
- 各Cコレクション記述子の使用を開始または終了するには、ALLOCATEまたはFREE文を使用します。

18.5 コレクション・サンプル・コード

COLLECTION SQL文を使用するSQLおよびPro*C/C++コードの例を示します。

18.5.1 型および表の作成

scott/tigerで接続し、SQLを使用して次の型を作成するとします。

```
CREATE TYPE employee AS OBJECT
(
  name   VARCHAR2(10),
  salary NUMBER(8,2)
) ;
CREATE TYPE employees AS VARRAY(15) OF employee ;
CREATE TYPE department AS OBJECT
(
  name VARCHAR2(15),
  team employees
) ;
```

Object Type Translator(OTT)によって、ヘッダー・ファイルが生成されます。OTTの入力として、次の入力ファイル(ファイル名in.typ)が使用されます。

```
case=lower
type employee
type employees
type department
```

次のコマンドによりヘッダー・ファイルが生成されます。

```
ott intype=in.typ outtype=out.typ hfile=example.h user=scott/tiger code=c
```

このヘッダー・ファイルexample.hは、OTTにより生成されます。

```
#ifndef EXAMPLE_ORACLE
# define EXAMPLE_ORACLE

#ifndef OCI_ORACLE
# include <oci.h>
#endif

typedef OCIRef employee_ref ;
typedef OCIArray employees ;
typedef OCIRef department_ref ;

struct employee
{
   OCIString * name ;
   OCINumber salary ;
} ;
typedef struct employee employee ;

struct employee_ind
{
   OCIInd _atomic ;
   OCIInd name ;
   OCIInd salary ;
} ;
typedef struct employee_ind employee_ind ;
struct department_ind
{
   OCIInd _atomic ;
   OCIInd name ;
   OCIInd team ;
} ;
typedef struct department_ind department_ind ;

#endif
```

注意:

ファイル oci.h には、OCIArray を定義した typedef を持つ orl.h が格納されています。typedef は、次の「typedef OCIColl OCIArray;」のようになります。OCIColl は、汎用コレクションを表す不透明な構造体です。

次の1列が含まれる簡単な表を作成します。

```
CREATE TABLE division ( subdivision department ) ;
```

この表に複数の行を挿入します。

```
INSERT INTO division (subdivision) VALUES
(department('Accounting',
```

```
            employees(employee('John', 75000),
                      employee('Jane', 75000)))
);
INSERT INTO division (subdivision) VALUES
(department('Development',
            employees(employee('Peter', 80000),
                      employee('Paula', 80000)))
) ;
INSERT INTO division (subdivision) VALUES
(department('Research',
            employees(employee('Albert', 95000),
                      employee('Alison', 95000)))
);
```

これらの型定義および表の情報を、次の例で使用します。

18.5.2 GETおよびSETの例

この例では、オブジェクトのコレクション属性から値を取り出し、簡単な修正を加え、コレクションに戻します。

まず、example.hをインクルードし、オブジェクト型の変数を宣言する必要があります。

```
#include <example.h>
department *dept_p ;
```

「開発」部門を部署表から選択します。

```
       EXEC SQL ALLOCATE :dept_p ;
       EXEC SQL SELECT subdivision INTO :dept_p
          FROM division WHERE name = 'Development' ;
```

employeeオブジェクト型のチームVARRAYの変数および単一のemployeeオブジェクトを表す変数が必要です。また、開発部門のすべてのメンバーの給料を昇給するので、そのための変数が必要です。

```
       employees *emp_array ;
       employee *emp_p ;
       double salary ;
```

作成したVARRAY Cコレクションおよびemployeeオブジェクト記述子に、ALLOCATEを実行する必要があります。ナビゲーショナル・アクセス用インタフェースを使用して、オブジェクトから実際のコレクションを取り出します。

```
       EXEC SQL ALLOCATE :emp_array ;
       EXEC SQL ALLOCATE :emp_p ;
       EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

ループを使用し、VARRAY要素に対して処理を繰り返します。WHENEVERディレクティブを使用してループの終了を制御します。

```
       EXEC SQL WHENEVER NOT FOUND DO break ;
       while (TRUE)
         {
```

まず、コレクションから変更する要素を取り出します。実際の要素型は、employeeオブジェクトです。

```
            EXEC SQL COLLECTION GET :emp_array INTO :emp_p ;
```

実際のオブジェクト要素を取り出したため、既存のナビゲーショナル・アクセス用インタフェースを使用して属性の値を変更します。この例では、全員の給料を10%増やします。

```
            EXEC SQL OBJECT GET salary FROM :emp_p INTO :salary ;
            salary += (salary * .10) ;
            EXEC SQL OBJECT SET salary OF :emp_p TO :salary ;
```

変更が終わると、コレクションに現在含まれているオブジェクト要素の属性の値を更新できます。

```
            EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
        }
```

すべてのコレクション要素に対して処理を繰り返した後に、そのコレクションを含むオブジェクトが格納されている表の列を更新する必要があります。

```
    EXEC SQL UPDATE division SET subdivision = :dept_p ;
```

次に、FREEを実行してすべてのリソースを解放し、COMMITを実行してこの一連の操作を終了します。

```
    EXEC SQL FREE :emp_array ;
    EXEC SQL FREE :emp_p ;
    EXEC SQL FREE :dept_p ;
    EXEC SQL COMMIT WORK ;
```

簡単な例ですが、必要な処理はすべて含まれています。ナビゲーショナルOBJECT GET文を応用して、コレクション属性をオブジェクトから取り出し、Cコレクション記述子に格納する方法は、明確に説明されています。さらに、そのCコレクション記述子を使用して、実際のコレクションの要素を取出しおよび更新を行うための、新しいCOLLECTION GET文およびSET文の使用方法について説明しました。コレクション・オブジェクト要素型の属性値の変更には、ナビゲーショナル・アクセス用インタフェースを使用しています。

18.5.3 DESCRIBEの例

この例では、DESCRIBE SQL文の使用方法を示します。任意のコレクションについての基本情報を検索します。

まず、例で使用されているヘッダー・ファイル、オブジェクト・ポインタおよびSQLコレクション記述子が必要です。

```
#include <example.h>
department *dept_p ;
```

前と同様に、オブジェクト・ポインタのALLOCATEを実行し、表からオブジェクトを取り出します。

```
EXEC SQL ALLOCATE :dept_p ;
EXEC SQL SELECT subdivision INTO :dept_p
   FROM division WHERE name = 'Research' ;
```

検索するコレクション属性情報を格納するPro*C/C++変数を宣言します。

```
int size ;
char type_name[9] ;
employees *emp_array ;
```

コレクション記述子を割り当て、ナビゲーショナル・アクセス用インタフェースを使用して、オブジェクトからコレクション属性を取り出します。

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

最後に、新しいCOLLECTION DESCRIBE文を使用して目的のコレクション属性情報を抽出します。

```
EXEC SQL COLLECTION DESCRIBE :emp_array
   GET SIZE, TYPE_NAME INTO :size, :type_name ;
```

注意:

この例のように、目的のコレクション属性名と同じホスト変数名を使用できます。

型employeesは、オブジェクトemployeeのVARRAYのため、型名を抽出できます。

DESCRIBEが正常に実行されると、SIZEの値は2 (このコレクション・インスタンスResearchの場合、2つの要素、AlbertおよびAlisonが存在します)になります。type_name変数は「EMPLOYEE¥0」(デフォルトではCHARZ)になります。

SQL記述子およびオブジェクト・ポインタを使用した処理が終了した後に、FREEを実行してリソースを解放します。

```
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :dept_p ;
```

この例では、Cコレクション記述子の参照先の基礎となるコレクションについて、記述子から情報を抽出するために使用するDESCRIBEのメカニズムについて説明しました。

18.5.4 RESETの例

開発の従業員の給料を昇給するかわりに、GETおよびSETの例のように、部署全体の給料を昇給します。

前の例と同様に、Object Type Translator(OTT)で生成されたサンプル・ヘッダー・ファイルなどを処理します。ただし、今回は、カーソルを使用して部署の部門ごとに、一度に1部門ずつ繰り返し実行します。

```
#include <example.h>
EXEC SQL DECLARE c CURSOR FOR SELECT subdivision FROM division ;
```

データを操作するローカル変数が必要になります。

```
department *dept_p ;
employees *emp_array ;
employee *emp_p ;
double salary ;
int size ;
```

オブジェクト変数およびコレクション変数を使用する前に、次のALLOCATE文を使用して初期化する必要があります。

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL ALLOCATE :emp_p ;
```

カーソルを使用して、部署のすべての部門に対して繰り返し処理を行うことができるようになりました。

```
EXEC SQL OPEN c ;
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
    {
        EXEC SQL FETCH c INTO :dept_p ;
```

ここで、部門オブジェクトを使用します。ナビゲーショナル・アクセス用インタフェースを使用して、部門からVARRAY属性teamを抽出する必要があります。

```
        EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

コレクションへの参照を開始する前に、RESET文を使用して、スライスのエンドポイントが現行のコレクション・インスタンスの始点

(前のインスタンスの最後ではありません)に設定されていることを確認してください。

```
         EXEC SQL COLLECTION RESET :emp_array ;
```

VARRAYのすべての要素を繰り返し処理し、前と同様に給料を更新します。このループの場合も、既存のWHENEVERディレクティブは引き続き有効です。

```
        while (TRUE)
           {
            EXEC SQL COLLECTION GET :emp_array INTO :emp_p ;
            EXEC SQL OBJECT GET salary FROM :emp_p INTO :salary ;
            salary += (salary * .05) ;
            EXEC SQL OBJECT SET salary OF :emp_p TO :salary ;
```

処理が完了すると、コレクション属性を更新します。

```
            EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
           }
```

前の例と同様に、変更が完了したコレクションを含むオブジェクトが格納された表の列を更新する必要があります。

```
        EXEC SQL UPDATE division SET subdivision = :dept_p ;
    }
```

ループが終了すると、処理は終了です。FREEを実行してすべてのリソースを解放し、COMMITで作業内容を送信します。

```
EXEC SQL CLOSE c ;
EXEC SQL FREE :emp_p ;
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :dept_p ;
EXEC SQL COMMIT WORK ;
```

この例では、同じコレクション型の異なるインスタンスに対して、ALLOCATEで割り当てられたコレクション記述子の再利用方法について説明しています。COLLECTION RESET文により、スライスのエンドポイントが必ず現在のコレクション・インスタンスの最初にリセットされます。エンドポイントは、前のコレクション・インスタンスの参照中に移動された後は、それまでの位置に残りません。

COLLECTION RESET文をこのように使用すると、アプリケーション開発者は同じコレクション型の新しいインスタンスを作成するたびに、コレクション記述子に対して明示的にFREEおよびALLOCATEを実行する必要がなくなります。

18.5.5 サンプル・プログラム: coldemo1.pc

次のプログラムcoldemo1.pcは、demoディレクトリにあります。

この例では、Pro*Cクライアントからコレクション型データベース列を操作する3種類の方法について説明しています。この例では、NESTED TABLEを使用していますが、VARRAYにも適用できます。

この例では、SQL*Plusファイル、coldemo1.sqlを使用して、挿入データおよびcalidata.sqlに格納されているデータを使用する表をセットアップします。

```
REM ************************************************************************
REM ** This is a SQL*Plus script to demonstrate collection manipulation
REM ** in Pro*C/C++.
REM ** Run this script before executing OTT for the coldemo1.pc program
REM ************************************************************************

connect scott/tiger;

set serveroutput on;
```

```
REM Make sure database has no old version of the table and types

DROP TABLE county_tbl;
DROP TYPE citytbl_t;
DROP TYPE city_t;

REM ABSTRACTION:
REM The counties table contains census information about each of the
REM counties in a particular U.S. state (California is used here).
REM Each county has a name, and a collection of cities.
REM Each city has a name and a population.

REM IMPLEMENTATION:
REM Our implementation follows this abstraction
REM Each city is implemented as a "city" OBJECT, and the
REM collection of cities in the county is implemented using
REM a NESTED TABLE of "city" OBJECTS.

CREATE TYPE city_t AS OBJECT (name CHAR(30), population NUMBER);
/

CREATE TYPE citytbl_t AS TABLE OF city_t;
/

CREATE TABLE county_tbl (name CHAR(30), cities citytbl_t)
  NESTED TABLE cities STORE AS citytbl_t_tbl;

REM Load the counties table with data.  This example uses estimates of
REM California demographics from Janurary 1, 1996.

@calidata.sql;

REM Commit to save
COMMIT;
```

表の設定方法およびこのプログラムでデモンストレーションする機能の説明については、次のプログラムの最初のコメントを参照してください。

```
            /* ***************************************** */
            /*   Demo program for Collections in Pro*C   */
            /* ***************************************** */

/*************************************************************************

   In SQL*Plus, run the SQL script coldemo1.sql to create:
     - 2 types: city_t (OBJECT) and citytbl_t (NESTED TABLE)
     - 1 relational table county_tbl which contains a citytbl_t nested table

   Next, run the Object Type Translator (OTT) to generate typedefs of C structs
   corresponding to the city_t and citytbl_t types in the databases:
     ott int=coldemo1.typ outt=out.typ hfile=coldemo1.h code=c user=scott/tiger

   Then, run the Pro*C/C++ Precompiler as follows:
     proc coldemo1 intype=out.typ

   Finally, link the generated code using the Pro*C Makefile:
     (Compiling and Linking applications is a platform dependent step).

 *************************************************************************
```

```
  Scenario:
    We consider the example of a database used to store census
    information for the state of California.  The database has a table
    representing the counties of California.  Each county has a name
    and a collection of cities.  Each city has a name and a population.

  Application Overview:
    This example demonstrates three ways for the Pro*C client to
    navigate through collection-typed database columns.  Although the
    examples presented use nested tables, they also apply to varrays.
    Collections-specific functionality is demonstrated in three
    different functions, as described in the following section.

    PrintCounties shows examples of
    * Declaring collection-typed host variables and arrays
    * Allocating and freeing collection-typed host variables
    * Using SQL to load a collection-typed host variable
    * Using indicators for collection-typed host variables
    * Using OCI to examine a collection-typed host variables

    PrintCounty shows examples of
    * Binding a ref cursor host variable to a nested table column
    * Allocating and freeing a ref cursor
    * Using the SQL "CURSOR" clause

    CountyPopulation shows examples of
    * Binding a "DECLARED" cursor to a nested table column
    * Using the SQL "THE" clause

*****************************************************************************/

/* Include files */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>                                  /* SQL Communications Area */
#include <coldemo1.h>        /* OTT-generated header with C typedefs for the */
                                    /* database types city_t and citytbl_t */
#ifndef EXIT_SUCCESS
# define EXIT_SUCCESS   0
#endif
#ifndef EXIT_FAILURE
# define EXIT_FAILURE   1
#endif

#define CITY_NAME_LEN    30
#define COUNTY_NAME_LEN  30
#define MAX_COUNTIES     60

/* Function prototypes */

#if defined(__STDC__)
 void OptionLoop( void );
 boolean GetCountyName( char *countyName );
 void PrintCounties( void );
 long CountyPopulation( CONST char *countyName );
 void PrintCounty( CONST char *countyName );
 void PrintSQLError( void );
```

```
 void PrintCountyHeader( CONST char *county );
 void PrintCity( city_t *city );
#else
 void OptionLoop();
 boolean GetCountyName(/*_ char *countyName _*/);
 void PrintCounties();
 long CountyPopulation(/*_ CONST char *countyName _*/);
 void PrintCounty(/*_ CONST char *countyName _*/);
 void PrintSQLError(/*_ void _*/);
 void PrintCountyHeader(/*_ CONST char *county _*/);
 void PrintCity(/*_ city_t *city _*/);
#endif

/*
 * NAME
 *   main
 * COLLECTION FEATURES
 *   none
 */
int main()
{
  char * uid = "scott/tiger";

  EXEC SQL WHENEVER SQLERROR DO PrintSQLError();

  printf("¥nPro*Census: Release California - Jan 1 1996.¥n");
  EXEC SQL CONNECT :uid;

  OptionLoop();

  printf("¥nGoodbye¥n¥n");
  EXEC SQL ROLLBACK RELEASE;
  return(EXIT_SUCCESS);
}

/*
 * NAME
 *   OptionLoop
 * DESCRIPTION
 *   A command dispatch routine.
 * COLLECTION FEATURES
 *   none
 */
void OptionLoop()
{
  char choice[30];
  boolean done = FALSE;
  char countyName[COUNTY_NAME_LEN + 1];

  while (!done)
  {
    printf("¥nPro*Census options:¥n");
    printf("¥tlist information for (A)ll counties¥n");
    printf("¥tlist information for one (C)ounty¥n");
    printf("¥tlist (P)opulation total for one county¥n");
    printf("¥t(Q)uit¥n");
    printf("Choice? ");

    fgets(choice, 30, stdin);
    switch(toupper(choice[0]))
```

```
    {
    case 'A':
      PrintCounties();
      break;
    case 'C':
      if (GetCountyName(countyName))
        PrintCounty(countyName);
      break;
    case 'P':
      if (GetCountyName(countyName))
        printf("¥nPopulation for %s county: %ld¥n",
               countyName, CountyPopulation(countyName));
      break;
    case 'Q':
      done = TRUE;
      break;
    default:
      break;
    }
  }
}


/*
 * NAME
 *   GetCountyName
 * DESCRIPTION
 *   Fills the passed buffer with a client-supplied county name.
 *   Returns TRUE if the county is in the database, and FALSE otherwise.
 * COLLECTION FEATURES
 *   none
 */
boolean GetCountyName(countyName)
  char *countyName;
{
  int   count;
  int   i;

  printf("County name? ");
  fgets(countyName, COUNTY_NAME_LEN + 1, stdin);

  /* Convert the name to uppercase and remove the trailing '¥n' */
  for (i = 0; countyName[i] != '¥0'; i++)
    {
      countyName[i] = (char)toupper(countyName[i]);
      if (countyName[i] == '¥n') countyName[i] = '¥0';
    }

  EXEC SQL SELECT COUNT(*) INTO :count
    FROM county_tbl WHERE name = :countyName;

  if (count != 1)
    {
      printf("¥nUnable to find %s county.¥n", countyName);
      return FALSE;
    }
  else
    return TRUE;
}
```

```
/*
 * NAME
 *   PrintCounties
 * DESCRIPTION
 *   Prints the population and name of each city of every county
 *   in the database.
 * COLLECTION FEATURES
 *   The following features correspond to the inline commented numbers
 *   1) Host variables for collection-typed objects are declared using
 *      OTT-generated types.  Both array and scalar declarations are allowed.
 *      Scalar declarations must be of type pointer-to-collection-type, and
 *      array declarations must of type array-of-pointer-to-collection-type.
 *   2) SQL ALLOCATE should be used to allocate space for the collection.
 *      SQL FREE should be used to free the memory once the collection is
 *      no longer needed.  The host variable being allocated or free'd
 *      can be either array or scalar.
 *   3) SQL is used to load into or store from collection-typed host variables
 *      and arrays.  No special syntax is needed.
 *   4) The type of an indicator variable for a collection is OCIInd.
 *      An indicators for a collections is declared and used just like
 *      an indicator for an int or string.
 *   5) The COLLECTION GET Interface is used to access and manipulate the
 *      contents of collection-typed host variables.  Each member of the
 *      collection used here has type city_t, as generated by OTT.
 */
void PrintCounties()
{
  citytbl_t *cityTable[MAX_COUNTIES];                                   /* 1 */
  OCIInd     cityInd[MAX_COUNTIES];                                     /* 4 */
  char       county[MAX_COUNTIES][COUNTY_NAME_LEN + 1];
  int        i, numCounties;
  city_t    *city;

  EXEC SQL ALLOCATE :cityTable;                                         /* 2 */
  EXEC SQL ALLOCATE :city;

  EXEC SQL SELECT name, cities
    INTO :county, :cityTable:cityInd FROM county_tbl;                 /* 3, 4 */

  numCounties = sqlca.sqlerrd[2];

  for (i = 0; i < numCounties; i++)
  {
    if (cityInd[i] == OCI_IND_NULL)                                     /* 4 */
      {
        printf("Unexpected NULL city table for %s county¥n", county[i]);
      }
    else
      {                                                                 /* 5 */
        PrintCountyHeader(county[i]);
        EXEC SQL WHENEVER NOT FOUND DO break;
        while (TRUE)
          {
            EXEC SQL COLLECTION GET :cityTable[i] INTO :city;
            PrintCity(city);
          }
        EXEC SQL WHENEVER NOT FOUND CONTINUE;
      }
  }
```

```
  EXEC SQL FREE :city;
  EXEC SQL FREE :cityTable;                                          /* 2 */
}


/*
 * NAME
 *   PrintCountyHeader
 * COLLECTION FEATURES
 *   none
 */
void PrintCountyHeader(county)
  CONST char *county;
{
  printf("¥nCOUNTY: %s¥n", county);
}

/*
 * NAME
 *   PrintCity
 * COLLECTION FEATURES
 *   none
 */
void PrintCity(city)
  city_t *city;
{
  varchar newCITY[CITY_NAME_LEN];
  int newPOP;

  EXEC SQL OBJECT GET NAME, POPULATION from :city INTO :newCITY, :newPOP;
  printf("CITY: %.*s POP: %d¥n", CITY_NAME_LEN, newCITY.arr, newPOP);
}

/*
 * NAME
 *   PrintCounty
 * DESCRIPTION
 *   Prints the population and name of each city in a particular county.
 * COLLECTION FEATURES
 *   The following features correspond to the inline commented numbers
 *   1) A ref cursor host variable may be used to scroll through the
 *      rows of a collection.
 *   2) Use SQL ALLOCATE/FREE to create and destroy the ref cursor.
 *   3) The "CURSOR" clause in SQL can be used to load a ref cursor
 *      host variable.  In such a case, the SELECT ... INTO does an
 *      implicit "OPEN" of the ref cursor.
 * IMPLEMENTATION NOTES
 *   In the case of SQL SELECT statements which contain an embedded
 *   CURSOR(...) clause, the Pro*C "select_error" flag must be "no"
 *   to prevent cancellation of the parent cursor.
 */
void PrintCounty(countyName)
  CONST char *countyName;
{
  sql_cursor cityCursor;                                             /* 1 */
  city_t *city;

  EXEC SQL ALLOCATE :cityCursor;                                     /* 2 */
  EXEC SQL ALLOCATE :city;
```

```
  EXEC ORACLE OPTION(select_error=no);
  EXEC SQL SELECT
    CURSOR(SELECT VALUE(c) FROM TABLE(county_tbl.cities) c)
      INTO :cityCursor
      FROM county_tbl
      WHERE county_tbl.name = :countyName;                          /* 3 */
  EXEC ORACLE OPTION(select_error=yes);

  PrintCountyHeader(countyName);

  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      EXEC SQL FETCH :cityCursor INTO :city;
      PrintCity(city);
    }
  EXEC SQL WHENEVER NOT FOUND CONTINUE;

  EXEC SQL CLOSE :cityCursor;

  EXEC SQL FREE :cityCursor;                                        /* 2 */
  EXEC SQL FREE :city;
}


/*
 * NAME
 *   CountyPopulation
 * DESCRIPTION
 *   Returns the number of people living in a particular county.
 * COLLECTION FEATURES
 *   The following features correspond to the inline commented numbers
 *   1) A "DECLARED" cursor may be used to scroll through the
 *      rows of a collection.
 *   2) The "THE" clause in SQL is used to convert a single nested-table
 *      column into a table.
 */
long CountyPopulation(countyName)
  CONST char *countyName;
{
  long population;
  long populationTotal = 0;

  EXEC SQL DECLARE cityCursor CURSOR FOR
    SELECT c.population
    FROM THE(SELECT cities FROM county_tbl
               WHERE name = :countyName) AS c;                      /* 1, 2 */

  EXEC SQL OPEN cityCursor;

  EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
    {
      EXEC SQL FETCH cityCursor INTO :population;
      populationTotal += population;
    }
  EXEC SQL WHENEVER NOT FOUND CONTINUE;

  EXEC SQL CLOSE cityCursor;
```

```
  return populationTotal;
}


/*
 * NAME
 *   PrintSQLError
 * DESCRIPTION
 *   Prints an error message using info in sqlca and calls exit.
 * COLLECTION FEATURES
 *   none
 */
void PrintSQLError()
{
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  printf("SQL error occurred...¥n");
  printf("%.*s¥n", (int)sqlca.sqlerrm.sqlerrml,
          (CONST char *)sqlca.sqlerrm.sqlerrmc);
  EXEC SQL ROLLBACK RELEASE;
  exit(EXIT_FAILURE);
}
```

19 Object Type Translator

この章では、Pro*C/C++アプリケーションで使用するためにデータベース・オブジェクト・タイプ、LOB型およびコレクション型をC言語の構造体にマップする、Object Type Translator(OTT)について説明します。

この章の内容は以下のとおりです。

- OTT概要
- Object Type Translator(OTT)
- OCIアプリケーションでのOTTの使用方法
- Pro*C/C++アプリケーションでのOTTの使用について
- OTT参照

19.1 OTT概要

OTT(Object Type Translator)は、Oracleサーバーのユーザー定義型を利用するアプリケーションの開発に役立ちます。

SQL CREATE TYPE文を使用して、オブジェクト型を作成できます。これらの型の定義をデータベースに格納しておき、データベースの表を作成するときに使用できます。これらの表への移入後は、OCI、Pro*C/C++またはJavaのプログラマは表に格納されているオブジェクトにアクセスできます。

オブジェクト・データにアクセスするアプリケーションには、データをホスト言語形式で表現する機能が必要です。これは、オブジェクト型をC構造体として表現することによって実現できます。プログラマがデータベース・オブジェクト・タイプを表す構造体宣言を、手入力でコーディングすることは可能です。ただし、多くの型がある場合、この作業は時間がかかり、エラーを生む原因になりがちです。OTTを使用すると、必要な構造体の宣言が自動的に生成されるため、作業を軽減できます。Pro*C/C++の場合、アプリケーション側では、OTTによって生成されたヘッダー・ファイルをインクルードするのみで済みます。OCIの場合は、アプリケーション側ではOTTによって生成される初期化関数もコールする必要があります。

OTTは、格納されているデータ型を表す構造体を作成するのみでなく、オブジェクト型またはそのフィールドがNULLかどうかを示す対になるインジケータ構造体も生成します。

関連項目

- LOB
- オブジェクト
- コレクション

19.2 Object Type Translator(OTT)

Object Type Translator(OTT)は、オブジェクト型と名前付きコレクション型のデータベース定義を、OCIまたはPro*C/C++アプリケーションにインクルードできるC言語の構造体宣言に変換します。

OCIとPro*C/C++のプログラマは、OTTを明示的に起動してデータベース型をC言語の表現に変換する必要があります。また、OCIのプログラマは、タイプ・バージョン表と呼ばれるデータ構造を、プログラムに必要なユーザー定義型の情報で初期化する必要があります。この初期化を実行するためのコードは、OTTによって生成されます。Pro*C/C++では、型のバージョン情報は、Pro*C/C++にパラメータとして渡されるOUTTYPEファイルに記録されます。

ほとんどのオペレーティング・システムでは、コマンドラインからOTTを起動します。OTTは、INTYPEファイルを入力として受け取り、OUTTYPEファイル、1つ以上のC言語のヘッダー・ファイルおよびオプションの実装ファイル(OCIプログラマ用)を生成します。OTTを

起動するコマンドの例を次に示します。

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h
```

このコマンドを実行すると、OTTはユーザー名*scott*とパスワード*tiger*を使用してデータベースに接続され、INTYPEファイル*demoin.typ*内の指示に従ってデータベース型をC言語の構造体に変換します。その結果、構造体は*CODE* パラメータで指定したホスト言語(C)用に、ヘッダー・ファイルdemo.hに出力されます。OUTTYPEファイル*demoout.typ*には、この変換に関する情報が格納されます。

各パラメータについては、この章のこの後の項で詳しく説明します。

demoin.typファイルの例:

```
CASE=LOWER
TYPE employee
```

demoout.typファイルの例:

```
CASE = LOWER
TYPE SCOTT.EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
```

この例では、demoin.typファイルに、TYPEが前に付く変換対象の型(TYPE employeeなど)があります。OUTTYPEファイルの構造はINTYPEファイルに似ていますが、OTTが取得した情報が追加されています。

OTTによる変換が完了すると、ヘッダー・ファイルにはINTYPEファイルで指定した各型を表すC言語の構造体と、各型に対応するNULLインジケータ構造体が含まれています。たとえば、INTYPEファイルにリストされたemployee型が次のように定義されたとします。

```
CREATE TYPE employee AS OBJECT
(
    name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
    salary      NUMBER
);
```

この場合、OTTによって生成されるヘッダー・ファイル(demo.h)には、他の項目とともに次の宣言が含まれます。

```
struct employee
{
    OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary;
};
typedef struct emp_type emp_type;

struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary;
```

```
};
typedef struct employee_ind employee_ind;
```

構造体宣言で使用されるデータ型(**OCIString**、**OCIInd**など)はOracle 8の新しい特殊データ型です。

これ以降の項では、OTTの使用について次の事柄を説明します。

- データベースでの型の作成について
- OTTの起動について
- OTTコマンドライン
- INTYPEファイル
- OTTデータ型のマッピング
- NULLインジケータ構造体
- OTTによる型の継承のサポート

次の各項では、OTTをOCIおよびPro*C/C++とともに使用する方法について説明してから、参考のためにコマンドライン構文、パラメータ、INTYPEファイルの構造、ネストされた#includeファイルの生成、スキーマ名の使用方法、デフォルト名のマッピングおよび制限事項について説明します。

関連項目

- OTTデータ型のマッピング

19.2.1 データベースでの型の作成について

OTTを使用するときの最初のステップは、オブジェクト型または名前付きコレクション型を作成してデータベースに格納することです。そのためには、SQL CREATE TYPE文を使用します。

関連項目

- オブジェクト

19.2.2 OTTの起動について

OTTを起動するステップは、次のとおりです。

OTTのパラメータは、コマンドラインまたは構成ファイル内で指定できます。一部のパラメータはINTYPEファイルでも指定できます。

パラメータを数箇所で指定すると、優先順位が最も高いのはコマンドラインで指定した値で、次にINTYPEファイル内の値、ユーザー定義の構成ファイルの値、デフォルトの構成ファイル内の値の順になります。

グローバル・オプション、つまりコマンドラインのオプションまたはINTYPEファイルのTYPE文の前のオプションについては、INTYPEファイルの値が上書きされます。(INTYPEファイルでグローバルとして指定できるオプションには、CASE、CODE、INITFILE、OUTDIRおよびINITFUNCがあります。HFILEは含まれません。)TYPEを指定するINTYPEファイルのオプションは、特定の型にのみ適用されます。その型に適用される、コマンドラインで指定したそれ以外のオプションは上書きされます。TYPE person HFILE=p.hと入力した場合、オプションはpersonにのみ適用され、コマンドラインのHFILEは上書きされます。文はコマンドライン・パラメータとはみなされません。

19.2.2.1 コマンドライン

コマンドラインに設定されたパラメータ(オプションとも呼ばれます)は、他で設定されたパラメータを上書きします。

関連項目

- オブジェクト

19.2.2.2 構成ファイル

構成ファイルは、OTTパラメータが記述されているテキスト・ファイルです。ファイル内の非空白行には、1つのパラメータと、それに対応付けられた1つ以上の値が入っています。1行に2つ以上のパラメータを指定した場合は、最初のパラメータのみが使用されます。構成ファイルの非空白行では、空白は使用できません。

構成ファイルは、コマンドラインで名前を指定できます。また、デフォルトの構成ファイルは常に読み取られます。このデフォルトの構成ファイルは、常に存在している必要がありますが、空でも構いません。デフォルトの構成ファイルの名前はottcfg.cfgであり、構成ファイルの位置はシステム固有の設定です。詳細は、プラットフォーム固有のマニュアルを参照してください。

19.2.2.3 INTYPEファイル

INTYPEファイルは、OTTにより変換される型のリストを示します。

パラメータCASE、HFILE、INITFUNCおよびINITFILEは、INTYPEファイルに入れることができます。

関連項目

- INTYPEファイル

19.2.3 OTTコマンドライン

ほとんどのプラットフォームでは、OTTはコマンドラインから起動します。入力ファイル、出力ファイルおよびデータベース接続情報などを指定できます。プラットフォームでOTTを起動する方法は、プラットフォーム固有のマニュアルで確認してください。

次の例(例1)は、コマンドラインからOTTを起動する方法を示しています。

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h
```

注意:

等号(=)の前後には空白は入力できません。

次の各項では、この例で使用しているコマンドラインの要素を説明します。

関連項目

- OTT参照

19.2.3.1 OTT

OTTを起動します。必ずコマンドラインの先頭に置きます。

19.2.3.2 Userid

OTTで使用されるデータベース接続情報を指定します。例1では、OTTはユーザー名*scott*およびパスワード*tiger*を使用して接続しようとします。

19.2.3.3 INTYPE

使用するINTYPEファイルの名前を指定します。INTYPEファイルの名前を*demoin.typ*と指定しています。

19.2.3.4 OUTTYPE

OUTTYPEファイルの名前を指定します。OTTは、C言語のヘッダー・ファイルを生成するときに、変換対象の型の情報をOUTTYPEファイルにも書き込みます。このファイルには、変換された各型のエントリが、バージョン文字列およびC表現を書き込んだヘッダー・ファイルとともに含まれています。

例1では、OUTTYPEファイルの名前を*demoout. typ*と指定しています。

> 注意:
>
> OUTTYPE で既存のファイルを指定した場合は、OTT の実行時に上書きされますが、次のように例外が 1 つあります。OTT で生成されたファイルの内容がそのファイルの前の内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

19.2.3.5 CODE

変換のターゲット言語を指定します。次のオプションを使用できます。

- C (ANSI_Cと等価)
- ANSI_C (ANSI C対応)
- KR_C (Kernighan & Ritchie C対応)

現在、デフォルト値はないため、このパラメータは必須です。

構造体の宣言は、両方のC言語において同一です。INITFILEファイルで定義された初期化関数が定義されるスタイルは、KR_Cが使用されるかどうかによって決まります。INITFILE オプションが使用されていない場合、3つのオプションはすべて等価です。

19.2.3.6 HFILE

生成された構造体を書き込むCヘッダー・ファイルの名前を指定します。例1では、生成された構造体はファイルdemo. hに格納されます。

> 注意:
>
> HFILE で既存のファイルを指定した場合は、OTT の実行時に上書きされますが、次のように例外が 1 つあります。OTT で生成されたファイルの内容がそのファイルの前の内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

19.2.3.7 INITFILE

型初期化関数を書き込むCソース・ファイルの使用を指定します。

初期化関数は、OCIプログラム内でのみ必要です。Pro*C/C++プログラムでは、Pro*C/C++ランタイム・ライブラリによって

型が自動的に初期化されます。

> **注意:**
>
> OUTTYPE で既存のファイルを指定した場合は、OTT の実行時に上書きされますが、次のように例外が 1 つあります。OTT で生成されたファイルの内容がそのファイルの前の内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

19.2.3.8 INITFUNC

INITFILEに定義する初期化関数の名前を指定します。

このパラメータを使用せずに初期化関数を生成すると、初期化関数の名前は、INITFILEの基本名と同一になります。

この関数は、OCIプログラム内でのみ必要です。

19.2.4 INTYPEファイル

OTTの実行時に、INTYPEファイルはどのデータベース型の変換が必要であるかをOTTに指示します。また、このファイルを使用して、生成される構造体の命名方法も指定できます。INTYPEファイルは、新しく作成しても、前にOTTを起動したときのOUTTYPEファイルを使用してもかまいません。INTYPEパラメータが使用されていない場合は、OTTの接続先のスキーマにあるすべての型が変換されます。

簡単なユーザー作成INTYPEファイルの例を次に示します。

```
CASE=LOWER
TYPE employee
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

第1行では、CASEキーワードによって、生成されたC言語の識別子を小文字にすることを指示しています。ただし、このCASEオプションは、INTYPEファイルに明示的に記述していない識別子にのみ適用されます。そのため、常に*employee*はemployee、*ADDRESS*はADDRESSというC構造体になります。これらの構造体のメンバーは、小文字で名前が付けられます。

TYPEキーワードで始まる行では、データベース内のどの型を変換するかを指定します。この場合、*EMPLOYEE*、*ADDRESS*、*ITEM*、*PERSON*および*PURCHASE_ORDER*型です。

TRANSLATE...ASキーワードでは、オブジェクト型をC言語の構造体に変換するときに、オブジェクト属性の名前を変更するように指定しています。この場合は、employee型のSALARY$属性がsalaryに変換されます。

最終行のASキーワードでは、オブジェクト型を構造体に変換するときに、名前を変更するように指定しています。この例では、*purchase_order*というデータベース型を*p_o*という構造体に変換します。

ASを使用して型や属性名を変換しなければ、その型や属性のデータベース名がC言語の識別子名として使用されます。ただし、CASEオプションを指定した場合、有効なC言語の識別子文字にマップできない文字は、アンダースコアに置換されます。型または属性名を変換する理由は、次のとおりです。

- 名前に、アルファベット、数字およびアンダースコア以外の文字が含まれる場合
- 名前がC言語のキーワードと競合する場合
- 型名が、同じ有効範囲内の別の識別子と競合する。たとえば、プログラムで、異なるスキーマにある同じ名前の2つの型を使用する場合に発生します。
- プログラマが別の名前に変更する場合。

OTTでは、INTYPEファイル内で指定されていない型の変換が必要な場合があります。これは、OTTでは、変換の実行前にINTYPEファイル内の型相互の依存性が分析され、必要であれば他の型も変換されるためです。たとえば、*ADDRESS*型がINTYPEファイルにリストされていなくても、*Person*型が*ADDRESS*型の属性を所有している場合、*Person*型の定義は必須のため、OTTは、*ADDRESS*を変換します。

> **注意:**
>
> リリース 1(9.0.1)以降では、TRANSITIVE = FALSE を指定することにより、INTYPE ファイルに指定されていない型が生成されないようにできます。デフォルトでは、TRANSITIVE は TRUE に設定されます。

通常の大/小文字の区別がないSQL識別子は、INTYPEファイルで、大/小文字のどのような組合せのつづりでもかまいません。そして、引用符は付けません。

CREATE TYPE "*Person*"などの大/小文字を区別して作成したSQL識別子を参照するには、TYPE "*Person*"などの引用符を使用します。宣言されたときに引用符が付けられたSQL識別子では、大/小文字が区別されます。

また、引用符は、TYPE "CASE"のようにOTT予約語であるSQL識別子を参照するためにも使用できます。この場合、そのSQL識別子がCREATE TYPE Caseのように大/小文字を区別しないで作成されていれば、名前に引用符を付けるときは、その名前を大文字にする必要があります。SQL識別子名を参照するために使用するOTT予約語に引用符を付けなければ、INTYPEファイル内で構文エラーがレポートされます。

関連項目

- CASE
- INTYPEファイルの構造

19.2.5 OTTデータ型のマッピング

OTTでデータベース型から生成したC構造体には、オブジェクト型の属性ごとに1つずつ対応する要素が含まれています。属性のデータ型は、Oracleのオブジェクト型で使用される型にマップされます。オブジェクト型やコレクションなどのユーザー定義型の作成をサポートするために、Oracleのデータ型には、事前定義済のプリミティブ型の集合が組み込まれています。

一連の事前定義の型には、数値型や文字型を含む、プログラマによく知られている標準の型があります。また、Oracle8で新しく導入されたデータ型(BLOBやCLOBなど)もあります。

Oracleには、オブジェクト型の属性をC言語の構造体で表すための事前定義済の型のセットも用意されています。たとえば、次のようなオブジェクト型定義と、OTTで生成した対応する構造体宣言があるとします。

```
CREATE TYPE employee AS OBJECT
(   name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
```

```
    salary$     NUMBER);
```

CASE=LOWERで、型または属性名の明示的なマッピングがないと仮定すると、OTT出力は次のようになります。

```
struct employee
{   OCIString * name;
    OCINumber empno;
    OCINumber department;
    OCIDate   hiredate;
    OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd department;
    OCIInd hiredate;
    OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

インジケータ構造体(struct employee_ind)については、NULLインジケータ構造体を参照してください。

構造体宣言のデータ型OCIString、OCINumber、OCIDate、OCIIndはOracle8の新しいオブジェクト型のCマッピングです。オブジェクト型属性のデータ型をマップするためにここで使用されます。たとえば、empno属性のNUMBERデータ型は、新しいOCINumberデータ型にマップします。また、これらの新しいデータ型は、バインド変数および定義変数の型としても使用します。

関連項目

- *OCIデータ構造*

19.2.5.1 オブジェクト・データ型のC言語へのマッピング

この項では、オブジェクトの属性型をOTTによって生成されたC言語のデータ型にマップする方法について説明します。表19-1は、OTTによって生成されるオブジェクト・データ型の属性として使用できる型からのマッピングを示しています。

表19-1 オブジェクト型属性のオブジェクト・データ型マッピング

オブジェクト属性の型	Cマッピング
VARCHAR2(N)	OCIString *
VARCHAR(N)	OCIString *
CHAR(N)、CHARACTER(N)	OCIString *
NUMBER、NUMBER(N)、NUMBER(N,N)	OCINumber
NUMERIC、NUMERIC(N)、NUMERIC(N,N)	OCINumber
REAL	OCINumber

オブジェクト属性の型	Cマッピング
INT、INTEGER、SMALLINT	OCINumber
FLOAT、FLOAT(N)、DOUBLE PRECISION	OCINumber
DEC、DEC(N)、DEC(N,N)	OCINumber
DECIMAL、DECIMAL(N)、DECIMAL(N,N)	OCINumber
DATE	OCIDate *
BLOB	OCIBlobLocator *
CLOB	OCIClobLocator *
BFILE	OCIBFileLocator *
ネストされたオブジェクト型	ネストされたオブジェクト型の C 言語名。
REF	typedef を使用して宣言。OCIRef *に相当。
RAW(N)	OCIRaw *

表19-2は、名前付きのコレクション型の、OTTによって生成されるオブジェクト・データ型へのマッピングを示しています。

表19-2 コレクション型のオブジェクト・データ型マッピング

名前付きコレクション型	Cマッピング
VARRAY	typedef を使用して宣言。OCIArray *に相当。
NESTED TABLE	typedef を使用して宣言。OCITable *に相当。

注意:

REF、VARRAY および NESTED TABLE 型では、OTT により typedef が生成されます。この typedef で宣言された型は、構造体宣言でデータ・メンバーの型として使用されます。

オブジェクト型にREFまたはコレクション型の属性が含まれる場合は、最初にREFまたはコレクション型のtypedefが生成されます。次に、オブジェクト型に対応する構造体宣言が生成されます。構造体には、REFまたはコレクション型へのポインタを型に持つ要素が含まれます。

あるオブジェクト型に、別のオブジェクト型を持つ属性が含まれている場合、OTTではネストした型が最初に生成されます(TRANSITIVE=TRUEの場合。)次に、オブジェクト型属性が、ネストしたオブジェクト型である型のネストした構造体にマッピングされます。

OTTによってオブジェクトではないデータベース属性型がマップされるC言語のデータ型は、構造体です。ただし、OCIDateの場合は不明になります。

関連項目

- OTT型マッピングの例

19.2.5.2 OTT型マッピングの例

次の例は、OTTによって作成される各種の型のマッピングを示しています。

この例では、次のデータベース型を使用します。

```
CREATE TYPE my_varray AS VARRAY(5) of integer;

CREATE TYPE object_type AS OBJECT
(object_name    VARCHAR2(20));

CREATE TYPE my_table AS TABLE OF object_type;

CREATE TYPE many_types AS OBJECT
( the_varchar    VARCHAR2(30),
  the_char       CHAR(3),
  the_blob       BLOB,
  the_clob       CLOB,
  the_object     object_type,
  another_ref    REF other_type,
  the_ref        REF many_types,
  the_varray     my_varray,
  the_table      my_table,
  the_date       DATE,
  the_num        NUMBER,
  the_raw        RAW(255));
```

また、INTYPEファイルの内容は次のとおりです。

```
CASE = LOWER
TYPE many_types
```

OTTでは、次のC言語の構造体が生成されます。

注意:

構造体の説明の補足コメントを次に示します。これらのコメントは、実際の OTT 出力の一部ではありません。

```
#ifndef MYFILENAME_ORACLE
```

#define MYFILENAME_ORACLE #ifndef OCI_ORACLE #include <oci.h> #endif typedef OCIRef many_types_ref; typedef OCIRef object_type_ref; typedef OCIArray my_varray; /* part of many_types */ typedef OCITable my_table; /* part of many_types*/ typedef OCIRef other_type_ref; struct object_type /* part of many_types */ { OCIString * object_name; };

typedef struct object_type object_type; struct object_type_ind /*indicator struct for*/ { /*object_types*/ OCIInd _atomic; OCIInd object_name; }; typedef struct object_type_ind object_type_ind; struct many_types { OCIString * the_varchar; OCIString * the_char; OCIBlobLocator * the_blob; OCIClobLocator * the_clob; struct object_type the_object; other_type_ref * another_ref; many_types_ref * the_ref; my_varray * the_varray; my_table * the_table; OCIDate the_date; OCINumber the_num; OCIRaw * the_raw; }; typedef struct many_types many_types; struct many_types_ind /*indicator struct for*/ { /*many_types*/ OCIInd _atomic; OCIInd the_varchar; OCIInd the_char; OCIInd the_blob; OCIInd the_clob; struct object_type_ind the_object; /*nested*/ OCIInd another_ref; OCIInd the_ref; OCIInd the_varray; OCIInd the_table; OCIInd the_date; OCIInd the_num; OCIInd the_raw; }; typedef struct many_types_ind many_types_ind; #endif

INTYPEファイル内で指定した変換対象の項目は1つのみですが、2つのオブジェクト型と2つの名前付きコレクション型が変換されていることに注目してください。これは、OTTパラメータTRANSITIVEのデフォルト値がTRUEであるためです。該当する項の説明にあるとおり、TRANSITIVE=TRUEの場合、指定された型の変換を完了するために、OTTは変換対象の型の属性として使用される型をすべて自動的に変換します。

ただし、その型がオブジェクト型属性内でポインタまたはREFでしかアクセスされない場合は例外です。たとえば、*many_types*型には属性として*another_re REF other_type*がありますが、`struct other_type`は生成されません。

また、この例では、VARRAY、NESTED TABLEおよびREFの各型を宣言するためのtypedefの使用方法を示しています。

typedefは、始めに使用されます。

```
typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
typedef OCITable my_table;
typedef OCIRef other_type_ref;
```

構造体many_typesでは、次のようにVARRAY、NESTED TABLEおよびREF属性が宣言されています。

```
struct many_types
{
   ...
   other_type_ref *   another_ref;
   many_types_ref *   the_ref;
   my_varray *        the_varray;
   my_table *         the_table;
   ...
}
```

19.2.6 NULLインジケータ構造体

OTTによってデータベース・オブジェクト・タイプを表すC言語の構造体が生成されるたびに、それに対応するNULLインジケータ構造体も生成されます。あるオブジェクト型をC言語の構造体に変換すると、NULLインジケータ情報をパラレル構造体に変換できます。

たとえば、前の項の例では次のNULLインジケータ構造体が生成されました。

```
struct many_types_ind
{
OCIInd _atomic;
OCIInd the_varchar;
OCIInd the_char;
```

```
OCIInd the_blob;
OCIInd the_clob;
struct object_type_ind the_object;
OCIInd another_ref;
OCIInd the_ref;
OCIInd the_varray;
OCIInd the_table;
OCIInd the_date;
OCIInd the_num;
OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;
```

NULL構造体のレイアウトは重要です。構造体の第1要素(_atomic)は、アトミック*NULL*インジケータです。この値は、オブジェクト型全体のNULL状態を示します。このアトミックNULLインジケータの後に、OTTで生成した、オブジェクト型を表現する構造体の各要素に対応するインジケータ要素が続きます。

あるオブジェクト型の定義の一部として別のオブジェクト型が含まれている場合(この例では*object_type*属性)、その属性のインジケータ・エントリは、ネストしたオブジェクト型に対応しているNULLインジケータ構造体(object_type_ind)です。

VARRAYとNESTED TABLEには、要素に関するNULL情報が含まれます。NULLインジケータ構造体のその他の要素のデータ型は、すべてOCIIndです。

アトミックNULLの詳細は、データ型のオブジェクト型の説明を参照してください。

19.2.7 OTTでの型の継承のサポート

オブジェクトの型の継承をサポートするために、OTTは、新しい属性を宣言する前に、継承される属性を、特別な名前_superを持つカプセル化された構造体内で宣言することで、オブジェクトのサブタイプを表すC構造体を生成します。これによって、スーパータイプを継承するオブジェクトのサブタイプについては、その構造体の第1要素が'_super'と命名され、サブタイプの各属性に対応する要素がその後に続きます。名前'_super'が付いた要素の型は、スーパータイプの名前です。

たとえば、サブタイプStudent_tとEmployee_tを持つPerson_t型は、次のように作成されます。

```
CREATE TYPE Person_t AS OBJECT
( ssn     NUMBER,
  name    VARCHAR2(30),
  address VARCHAR2(100)) NOT FINAL;

CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
  major  VARCHAR2(30)) NOT FINAL;

CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr   VARCHAR2(30));
```

また、INTYPEファイルに次が含まれているとします。

```
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON_T
```

OTTでは、Person_t、Student_tおよびEmployee_tの次のC言語の構造体と、そのNULLインジケータ構造体が生成されます。

```
#ifndef MYFILENAME_ORACLE
```

```
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCIRef EMPLOYEE_T_ref;
typedef OCIRef STUDENT_T_ref;
typedef OCIRef PERSON_T_ref;

struct PERSON_T
{
   OCINumber SSN;
   OCIString * NAME;
   OCIString * ADDRESS;
};
typedef struct PERSON_T PERSON_T;

struct PERSON_T_ind
{
   OCIInd _atomic;
   OCIInd SSN;
   OCIInd NAME;
   OCIInd ADDRESS;
};
typedef struct PERSON_T_ind PERSON_T_ind;

struct EMPLOYEE_T
{
   PERSON_T _super;
   OCINumber EMPID;
   OCIString * MGR;
};
typedef struct EMPLOYEE_T EMPLOYEE_T;

struct EMPLOYEE_T_ind
{
   PERSON_T _super;
   OCIInd EMPID;
   OCIInd MGR;
};
typedef struct EMPLOYEE_T_ind EMPLOYEE_T_ind;

struct STUDENT_T
{
   PERSON_T _super;
   OCINumber DEPTID;
   OCIString * MAJOR;
};
typedef struct STUDENT_T STUDENT_T;

struct STUDENT_T_ind
{
   PERSON_T _super;
   OCIInd DEPTID;
   OCIInd MAJOR;
};
typedef struct STUDENT_T_ind STUDENT_T_ind;

#endif
```

前述のCマッピングの変換により、C言語でのサブタイプのインスタンスからスーパータイプのインスタンスへの単純な上位キャストが正常に機能します。次に例を示します。

```
STUDENT_T *stu_ptr = some_ptr;               /* some STUDENT_T instance  */
PERSON_T  *pers_ptr = (PERSON_T *)stu_ptr;   /* up-casting */
```

同様にNULLインジケータ構造体が生成されます。スーパータイプPerson_tのNULLインジケータ構造体の最初の要素は_atomicで、サブタイプEmployee_tおよびStudent_tのNULLインジケータ構造体の最初の要素は_superであることに注意してください(サブタイプの場合、アトミック要素は生成されません)。

19.2.7.1 置換可能なオブジェクト属性

NOT FINAL型の属性(したがって、潜在的に置換可能な属性)の場合、埋込み属性はポインタとして表されます。

Book_t型が次のように作成されるとします。

```
CREATE TYPE Book_t AS OBJECT
( title   VARCHAR2(30),
  author  Person_t     /* substitutable */);
```

OTTで生成された対応するC構造体には、Person_tへのポインタが含まれています。

```
struct Book_t
{
  OCIString  *title;
  Person_t   *author;    /* pointer to Person_t struct */
}
```

前述の型に対応するNULLインジケータ構造体は、次のようになります。

```
struct Book_t_ind
{
  OCIInd  _atomic;
  OCIInd  title;
  OCIInd  author;
}
```

author属性に対応するNULLインジケータ構造体は、authorオブジェクト自体から取得できることに注意してください。

型がFINALに定義されている場合、サブタイプは使用できません。したがって、FINAL型の属性は置換できません。その場合、マッピングは前述のとおりになり、属性構造体はインラインです。型が変更され、NOT FINALに定義される場合は、マッピングを変更する必要があります。新しいマッピングは、OTTを再実行すると生成されます。

関連項目

- OBJECT GET

19.2.8 OUTTYPEファイル

OUTTYPEファイルは、OTTコマンドラインで名前が指定されます。OTTは、C言語のヘッダー・ファイルを生成するときに、変換結果をOUTTYPEファイルにも書き込みます。このファイルには、変換された各型のエントリが、バージョン文字列およびC表現を書き込んだヘッダー・ファイルとともに含まれています。

OTTを一度実行して生成したOUTTYPEファイルは、それ以降にOTTを起動するときのINTYPEファイルとして使用できます。

たとえば、この章の前半の例で使用した単純なINTYPEファイルを考えてみます。

```
CASE=LOWER
TYPE employee
```

```
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE person
TYPE PURCHASE_ORDER AS p_o
```

ここで、ユーザーはOTTによって生成されるC言語の識別子に大/小文字のどちらを使用するかを選択し、変換する型のリストを指定しています。これらの型のうちの2つについては、ネーミング規則が指定されています。

次の例は、OTTの実行後のOUTTYPEファイルの内容を示しています。

```
CASE = LOWER
TYPE EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
  TRANSLATE SALARY$ AS salary
             DEPTNO AS department
TYPE ADDRESS AS ADDRESS
  VERSION = "$8.0"
  HFILE = demo.h
TYPE ITEM AS item
  VERSION = "$8.0"
  HFILE = demo.h
TYPE "Person" AS Person
  VERSION = "$8.0"
  HFILE = demo.h
TYPE PURCHASE_ORDER AS p_o
  VERSION = "$8.0"
  HFILE = demo.h
```

OUTTYPEファイルの内容を調べると、INTYPE仕様部で指定されていなかった型がリスト表示されている場合があります。たとえば、INTYPEファイルでは、次の内容による*person*型の変換のみを指定したとします。

```
CASE = LOWER
TYPE PERSON
```

このperson型の定義に*address*型の属性が含まれている場合、OUTTYPEファイルには、PERSONとADDRESSの両方のエントリが含まれます。person型を完全に変換するには、最初に*address*を変換する必要があります。

パラメータTRANSITIVEをTRUE(デフォルト)に設定していると、OTTでは、変換の実行前に、INTYPEファイル内の型相互の依存性が分析され、必要であれば他の型も変換されます。

19.3 OCIアプリケーションでのOTTの使用方法

OTTによって生成されるC言語のヘッダー・ファイルと実装ファイルは、データベース・サーバーのオブジェクトにアクセスするOCIアプリケーションで使用できます。ヘッダー・ファイルをOCIコードに取込むには、#include文を使用します。

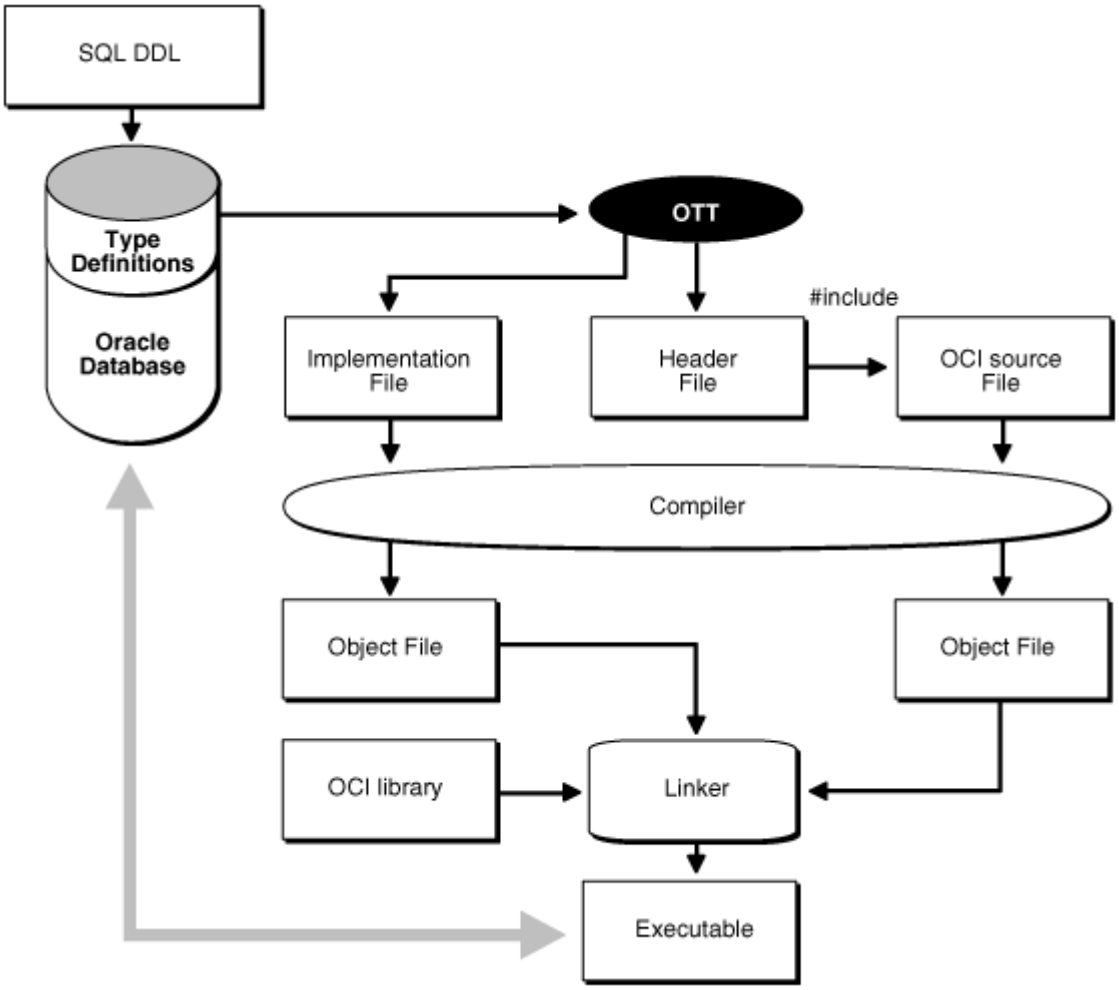
OCIアプリケーションでは、ヘッダー・ファイルを組み込んだ後、ホスト言語形式のオブジェクト・データにアクセスし、操作できます。

図19-1は、OCIでOTTを使用する場合に必要な手順を示しています。

1. SQLを使用してデータベースに型定義を作成します。
2. OTTで、オブジェクト型と名前付きコレクション型のC表現を含むヘッダー・ファイルを生成します。INITFILEオプションで命名された実装ファイルも生成します。
3. アプリケーションを記述します。OCIアプリケーションでユーザーが記述したコードで、INITFUNC関数を宣言してコールします。

4. ヘッダー・ファイルをOCIソース・コード・ファイルにインクルードします。
5. OTTで生成された実装ファイルを含めて、OCIアプリケーションがコンパイルされ、OCIライブラリにリンクされます。
6. OCI実行可能ファイルをOracleサーバーに対して実行します。

図19-1 OCIでのOTTの使用方法



19.3.1 OCIでのオブジェクトへのアクセスおよび操作について

アプリケーション内部では、OCIプログラムでバインド操作と定義操作を実行できます。そのためには、OTTで生成したヘッダー・ファイルに示されている型で宣言したプログラム変数を使用します。

たとえば、アプリケーションでSQLのSELECT文を使用してオブジェクトへのREFをフェッチし、適切なOCI関数を使用してそのオブジェクトを確保します。オブジェクトを確保した後、その他のOCI関数を使用してそのオブジェクトの属性データにアクセスし、操作できます。

OCIには、データ型のマッピングと操作のための一連の関数が組み込まれています。これらの関数は、オブジェクト型と名前付きコレクション型の属性を操作するために明確に設計されています。

次に、使用可能な関数の一部を示します。

- OCIStringSize()は、OCIString文字列のサイズを取得します。
- OCINumberAdd()は、2つのOCINumber数値を加算します。
- OCILobIsEqual()は、2つのLOBロケータが等しいかどうかを比較します。
- OCIRawPtr()は、OCIRawのRAWデータ型へのポインタを取得します。
- OCICollAppend()は、コレクション型(OCIArrayまたはOCITable)に要素を追加します。
- OCITableFirst()は、NESTED TABLE(OCITable)の最初の既存要素の索引を戻します。

- OCIRefIsNull()は、REF(OCIRef)がNULLかどうかをテストします。

これらの関数については、『*Oracle Call Interfaceプログラマーズ・ガイド*』の次の各章で詳しく説明しています。

- 第2章(バインドと定義付けなどのOCI概念の説明)
- 第6章(オブジェクトへのアクセスとナビゲーションの説明)
- 第7章(データ型のマップと操作の説明)
- 第12章(データ型のマップと操作の関数のリスト)

19.3.2 初期化関数のコールについて

OTTは、必要に応じてC初期化関数を生成します。初期化関数では、プログラムで使用されている各オブジェクト型について、どのバージョンの型が使用されているかを環境に通知します。INITFUNCオプションでOTTを起動する場合は、初期化関数の名前を指定するか、初期化関数が含まれている実装ファイル(INITFILE)の名前に基づいてデフォルト名を選択できます。

初期化関数には、環境ハンドル・ポインタとエラー・ハンドル・ポインタの2つの引数があります。一般的に、使用する初期化関数は1つですが、必ずしもそうである必要はありません。プログラムにコンパイル済の個別のピースがあり、異なる型を要求する場合は、各ピースに対して初期化関数が含まれた1つの初期化ファイルをそれぞれ要求して、OTTを個別に実行できます。

OCIEnvInit()をコールするなど、明示的なOCIオブジェクト・コールによって環境ハンドルを作成した後に、環境ハンドルごとに初期化関数も明示的にコールする必要があります。これによって、各ハンドルからプログラム全体で使用されるすべてのデータ型にアクセスできます。

EXEC SQL CONTEXT USEやEXEC SQL CONNECTなどの埋込みSQL文を使用して環境ハンドルを暗黙的に作成する場合、ハンドルは暗黙的に初期化され、初期化関数をコールする必要はありません。これは、Pro*C/C++アプリケーションまたはPro*C/C++とOCIアプリケーションが併用される場合に適用されます。

次に、初期化関数の例を示します。

INTYPEファイルex2c.typを指定します。このファイルには次の型が含まれています。

```
TYPE SCOTT.PERSON
TYPE SCOTT.ADDRESS
```

また、次のコマンドラインを含みます。

```
ott userid=scott/tiger intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c
```

OTTによって、次のようなファイルex2cv.cが生成されます。

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
   sword status = OCITypeVTInit(env, err);
   if (status == OCI_SUCCESS)
      status = OCITypeVTInsert(env, err,
          "SCOTT", 5,
          "PERSON", 6,
          "$8.0", 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
           "SCOTT", 5,
           "ADDRESS", 7,
```

```
            "$8.0", 4);
    return status;
}
```

関数*ex2cv*によってタイプ・バージョン表が作成され、SCOTT.PERSON型およびSCOTT.ADDRESS型が挿入されます。

プログラムで明示的に環境ハンドルを作成する場合、明示的に作成するハンドルごとに初期化関数をコールする必要があるので、すべての初期化関数を生成し、コンパイルし、リンクしてください。プログラムが明示的に環境ハンドルを作成しない場合、初期化関数は必要ありません。

OTTで生成したヘッダー・ファイルを使用するプログラムでは、同時に生成された初期化関数も使用する必要があります。具体例として、コンパイルによってプログラムPにリンクされるコードが生成される場合を考えます。OTTによって生成されるヘッダー・ファイルをコンパイル環境にインクルードし、プログラムPのどこかで環境ハンドルが明示的に作成される場合は、それと同時にOTTによって生成される実装ファイルもコンパイルしてプログラムPにリンクする必要があります。この操作は、プログラマが正しく実行する必要があります。

19.3.3 初期化関数のタスク

C初期化関数は、OTTで処理する型のバージョン情報を提供します。C初期化関数は、OTTで処理する各オブジェクト・データ型の名前とバージョン識別子を型バージョン表に追加します。

Open Type Manager(OTM)では、特定のプログラムで使用する型のバージョンを識別するためにタイプ・バージョン表を使用します。OTTによって様々な初期化関数が別々に生成されると、タイプ・バージョン表に同じ型がいくつも追加されることがあります。型が何度も追加されると、そのたびにOTMによって同じバージョンの型が登録されているかどうかが確認されます。

初期化関数に対して関数プロトタイプを宣言し、その関数をコールするのは、OCIプログラマの役割です。

> 注意:
>
> Oracle の現行のリリースでは、それぞれの型のバージョンは 1 つのみです。タイプ・バージョン表の初期化は、Oracle の将来のリリースとの互換性のためにのみ必要です。

19.4 Pro*C/C++アプリケーションでのOTTの使用について

Pro*C/C++アプリケーションの作成時の型の変換処理は、OCIベースのアプリケーションの作成時よりも簡単です。これは、プリコンパイラで生成されるコードによって自動的にタイプ・バージョン表が初期化されるためです。

Pro*C/C++アプリケーションでは、OTTによって生成されるC言語のヘッダー・ファイルを使用して、データベース・サーバーのオブジェクトにアクセスできます。このヘッダー・ファイルは、#include文によりコードに組み込まれます。ヘッダー・ファイルをインクルードした後は、Pro*C/C++アプリケーションからホスト言語形式でオブジェクト・データにアクセスしたり、オブジェクト・データを操作したりできます。

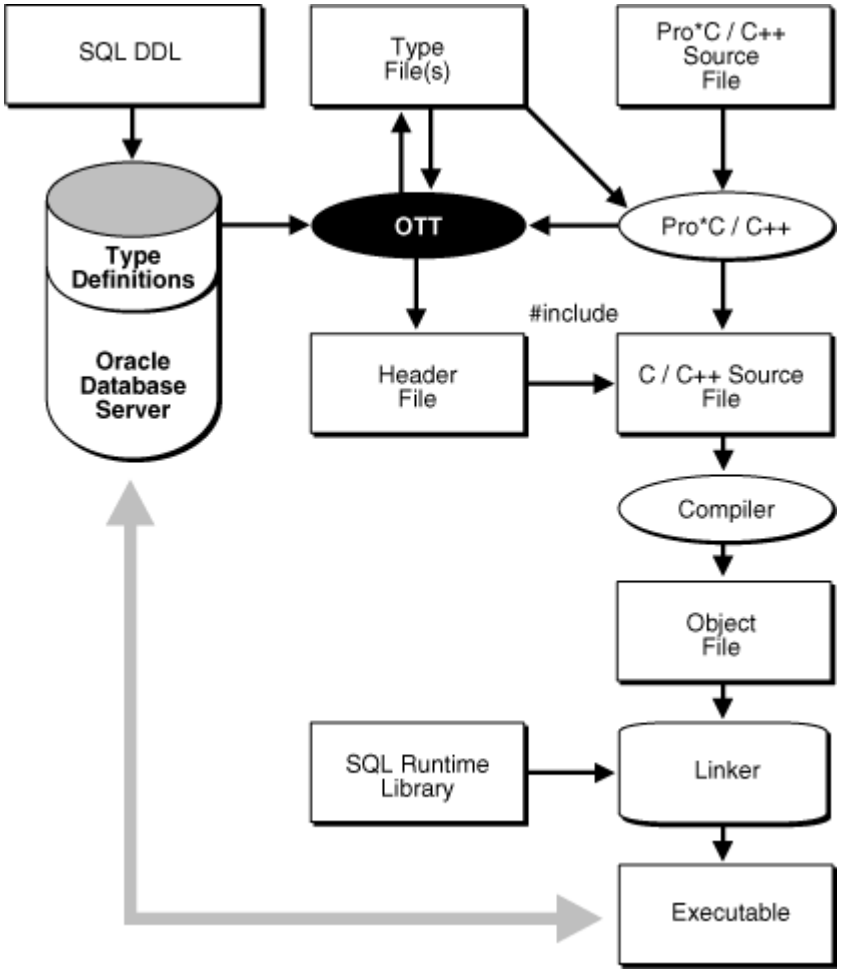
図19-2は、Pro*C/C++でOTTを使用する場合に必要な手順を示しています。

1. SQLを使用してデータベースに型定義を作成します。
2. OTTで、オブジェクト型、REF型および名前付きコレクション型のC表現を含むヘッダー・ファイルを生成します。INTYPEパラメータとしてPro*C/C++に渡されるOUTTYPEファイルも生成されます。
3. ヘッダー・ファイルをPro*C/C++ソース・コード・ファイルにインクルードします。

4. Pro*C/C++アプリケーションをコンパイルし、Pro*C/C++のランタイム・ライブラリSQLLIBにリンクします。

5. Pro*C/C++実行可能ファイルをOracleサーバーで実行します。

図19-2 オブジェクト指向Pro*C/C++アプリケーションの作成



前述の手順2が示すように、OTTによって生成されるOUTTYPEファイルは、Pro*C/C++プログラマにとって特別な用途があります。Pro*C/C++の起動時に、OUTTYPEファイルを新しいINTYPEコマンドライン・パラメータに渡します。このファイルの内容は、OTT生成の構造体に対応付けるデータベース型を決定するためにプリコンパイラで使用されます。OCIでプログラミングしている場合は、バインド、定義および型情報へのアクセスのための特別な関数を使用して、この対応付けを明示的に行う必要があります。

また、プリコンパイラにより、OTTのOUTTYPE(Pro*C/C++のINTYPE)ファイルで名前が付けられた型を使用して、タイプ・バージョン表を初期化するコードが生成されます。

> **注意:**
>
> 常に OTT からの OUTTYPE ファイルを Pro*C/C++の INTYPE ファイルとして使用することをお薦めします。Pro*C/C++の INTYPE ファイルを作成することも可能ですが、エラーが発生する可能性もあるためお薦めしません。

サーバーから取り出されたオブジェクトの属性を操作するには、OCIのデータ型のマッピングと操作のための関数をコールするのも1つの方法です。これを実行する前に、アプリケーションで、まずSQLEnvGet()をコールしてOCI関数に渡すOCI環境ハンドルを取得し、次にSQLSvcCtxGet()をコールしてOCI関数に渡すOCIサービス・コンテキストを取得する必要があります。
Pro*C/C++には、オブジェクト属性の操作に使用できる機能もあります。

関連項目

- オブジェクト
- OCIでのオブジェクトへのアクセスおよび操作について
- https://www.oracle.com/pls/topic/lookup?ctx=en/database/oracle/oracle-database/19/lnpcc&id=LNOCI030

19.5 OTT参照

OTTの動作を制御するパラメータは、OTTコマンドラインでもCONFIGファイル内でも指定できます。また、一部のパラメータは、INTYPEファイルにも指定できます。この項では、次の項目について詳しく説明します。

- OTTコマンドライン構文
- OTTパラメータ
- OTTパラメータの指定可能な場所
- INTYPEファイルの構造
- ネストした#includeファイル生成
- SCHEMA_NAMESの使用方法
- デフォルト名のマッピング
- 制限事項

この章では、次の規則を使用してOTTの構文を説明します。

- イタリック文字列はユーザーが入力する内容です。
- 大文字の文字列は、そのとおりに入力する文字列です。ただし、大/小文字は区別されないため、小文字で入力しても有効です。
- 大カッコ[...]で囲んだ項目は、オプション項目です。
- 1つの項目(あるいはカッコで囲まれた複数の項目)の直後の省略記号(...)は、その項目を何度も繰り返し指定できることを示します。
- これ以外の句読点記号は、示されているとおりに入力します。これには、'.'、'@'などが含まれます。

19.5.1 OTTコマンドライン構文

OTTコマンドライン・インタフェースは、OTTを明示的に起動してデータベース型をC構造体に変換するときに使用します。このインタフェースは、オブジェクトを活用するOCIアプリケーションまたはPro*C/C++アプリケーションの開発に必須です。

OTTコマンドライン文は、キーワードOTTと、その後に続くOTTパラメータのリストによって構成されます。

OTTコマンドラインの文で指定できるパラメータは、次のとおりです。

```
[USERID=username/password[@db_name]]
[INTYPE=in_filename]
OUTTYPE=out_filename
CODE={C|ANSI_C|KR_C}
[HFILE=filename]
[ERRTYPE=filename]
[CONFIG=filename]
[INITFILE=filename]
[INITFUNC=filename]
```

```
[CASE={SAME|LOWER|UPPER|OPPOSITE}]
[SCHEMA_NAMES={ALWAYS|IF_NEEDED|FROM_INTYPE}]
[TRANSITIVE=TRUE|FALSE]
```

注意:

通常、OTT コマンドの後に続くパラメータはどのような順序でもよく、常に必須となるのは OUTTYPE および CODE パラメータのみです。

HFILEパラメータは、ほとんどの場合に使用されます。省略すると、INTYPEファイルのそれぞれの型について、個別にHFILEを指定する必要があります。OTTがINTYPEファイルで指定されていない型を変換する必要があると判断すると、エラーがレポートされます。したがって、INTYPEファイルが以前にOTT OUTTYPEファイルとして生成されている場合のみ、HFILEパラメータを省略できます。

INTYPEファイルを省略すると、スキーマ全体が変換されます。詳細は、次の項のパラメータの説明を参照してください。

次にOTTコマンドライン文の例を示します(1行に入力します)。

```
OTT userid=scott/tiger intype=in.typ outtype=out.typ code=c hfile=demo.h errtype=demo.tls case=lower
```

OTTコマンドラインの各パラメータについて、この後の各項で説明します。

19.5.2 OTTパラメータ

OTTコマンドラインにパラメータを入力するときの書式は、次のとおりです。

parameter=value

*parameter*はリテラル・パラメータ文字列で、*value*は有効なパラメータ設定値です。リテラル・パラメータ文字列は大/小文字を区別しません。

コマンドラインのパラメータは、空白またはタブのいずれかを使用して区切ります。

また、パラメータは構成ファイル内でも指定できます。ただし、この場合、行の中に空白を入れることはできないため、各パラメータは独立した行に指定する必要があります。さらに、パラメータCASE、HFILE、INITFUNCおよびINITFILEは、INTYPEファイルに入れることができます。

19.5.2.1 USERID

USERIDパラメータでは、Oracleユーザー名およびパスワード、オプションのデータベース名(Oracle Netのデータベース指定文字列)を指定します。データベース名を省略すると、デフォルトのデータベースが使用されます。このパラメータの構文は、次のとおりです。

```
USERID=username/password[@db_name]
```

これが第1パラメータの場合は、「USERID=」を省略して、次のように指定できます。

```
OTT username/password...
```

USERIDパラメータはオプションです。省略した場合、OTTは自動的にユーザーCLUSTER$*username*でデフォルトのデータベースへの接続を試みます。*username*は、ユーザーのオペレーティング・システムのユーザー名です。

19.5.2.2 INTYPE

INTYPEパラメータでは、オブジェクト型指定のリストの読取り元ファイルの名前を指定します。読み取ったリストにある型が、OTTによって変換されます。このパラメータの構文は、次のとおりです。

```
INTYPE=filename
```

USERIDが第1パラメータ、INTYPEが第2パラメータで、USERID=を省略した場合は、INTYPE=も省略できます。INTYPEが指定されていない場合は、ユーザーのスキーマにおけるすべての型が変換されます。

```
OTT username/password filename...
```

INTYPEファイルは、型宣言に対するMakeファイルとみなすことができます。C構造体宣言の必要な型をリストします。

コマンドラインまたはINTYPEファイルのファイル名に拡張子が含まれていない場合は、「TYP」や「typ」などのプラットフォーム固有の拡張子が追加されます。

関連項目

- INTYPEファイルの構造

19.5.2.3 OUTTYPE

OTTによって処理されるすべてのオブジェクト・データ型の型情報が書き込まれるファイルの名前を指定します。これには、INTYPEファイルで明示的に指定したすべての型が含まれます。さらに、変換対象である他の型の宣言で使用しているために変換された型が含まれる場合もあります(TRANSITIVE=TRUEの場合)。このファイルは、それ以後にOTTを起動するときにINTYPEファイルとして使用できます。

```
OUTTYPE=filename
```

INTYPEパラメータとOUTTYPEパラメータが同一のファイルを参照している場合、INTYPEファイルの古い情報は、新しいINTYPEの情報に置き換えられます。これは、型の変更から始まり、型宣言の生成、ソースコードの編集、プリコンパイル、コンパイル、デバッグに至るサイクル内で、同一のINTYPEファイルを繰返し使用するときに便利です。

OUTTYPEは必ず指定します。

コマンドラインまたはINTYPEファイルのファイル名に拡張子が含まれていない場合は、「TYP」や「typ」などのプラットフォーム固有の拡張子が追加されます。

19.5.2.4 CODE

```
CODE= C|KR_C|ANSI_C
```

OTTの出力を表すホスト言語です。CODE=C、CODE=KR_CまたはCODE=ANSI_Cのいずれかを指定できます。「CODE=C」は、「CODE=ANSI_C」と等価です。

このパラメータは、デフォルト値がないので必ず指定する必要があります。

19.5.2.5 INITFILE

INITFILEパラメータでは、OTTで生成した初期化ファイルを書き込むファイルの名前を指定します。このパラメータを省略すると、OTTは初期化関数を生成しません。

Pro*C/C++プログラムの場合、必要な初期化はSQLLIBランタイム・ライブラリによって実行されるため、INITFILEは必要ありません。OCIプログラムのユーザーは、INITFILEファイルをコンパイルおよびリンクし、環境ハンドルの作成時に初期化関数をコールする必要があります。

コマンドラインまたはINTYPEファイルで指定したINITFILEファイル名に拡張子を付けなかった場合は、「C」または「.c」のようなプラットフォーム固有の拡張子が追加されます。

```
INITFILE=filename
```

19.5.2.6 INITFUNC

INITFUNCパラメータは、OCIプログラムのみで使用します。OTTで生成する初期化関数の名前を指定します。このパラメータを省略すると、INITFILEの名前から初期化関数の名前が付けられます。

```
INITFUNC=filename
```

19.5.2.7 HFILE

INTYPEファイルに記述した型の宣言でインクルード(.h)ファイルを指定しなかった場合に、OTTによって生成されるインクルード・ファイルの名前を指定します。INTYPEファイルで各型のインクルード・ファイルを個々に指定していない場合は、このパラメータが必要です。また、INTYPEファイルに記述していない型を、2つ以上の異なるファイルで宣言されている他の型で使用する場合(TRANSITIVE=TRUEの場合)は、INTYPEファイルに記述していない型も生成する必要があります。そのような場合もこのパラメータが必要です。

コマンドラインまたはINTYPEファイルで指定したHFILEファイル名に拡張子を付けなかった場合、「H」や「.h」のようなプラットフォーム固有の拡張子が追加されます。

```
HFILE=filename
```

19.5.2.8 CONFIG

CONFIGパラメータでは、共通に使用されるパラメータ指定を含むOTT構成ファイルの名前を指定します。また、パラメータ指定は、プラットフォームによって異なる位置にあるシステム構成ファイルから読み込まれます。残りのすべてのパラメータ指定は、コマンドラインまたはINTYPEファイルで指定する必要があります。

```
CONFIG=filename
```

注意: CONFIGパラメータは、CONFIGファイルでは使用できません。

19.5.2.9 ERRTYPE

このパラメータを指定すると、INTYPEファイルのリストが、すべての情報メッセージおよびエラー・メッセージとともにERRTYPEファイルに書き込まれます。情報メッセージおよびエラー・メッセージは、ERRTYPEを指定したかどうかに関係なく、標準出力に送信されます。

実質的に、ERRTYPEファイルはエラー・メッセージが追加されたINTYPEファイルのコピーです。ほとんどの場合、エラー・メッセージには、エラーの原因となったテキストへのポインタが含まれます。

コマンドラインまたはINTYPEファイルのERRTYPEファイル名に拡張子が付いていない場合は、「.TLS」または「.tls」など、プラットフォーム固有の拡張子が追加されます。

```
ERRTYPE=filename
```

19.5.2.10 CASE

このパラメータでは、OTT生成の特定のC言語の識別子を大文字で表記するか小文字で表記するかを指定します。CASEの可能な値は、SAME、LOWER、UPPERおよびOPPOSITEです。CASE = SAMEの場合は、データベース型と属性名をC識別子に変換するときに、大/小文字は変更されません。CASE=LOWERの場合、大文字はすべて小文字に変換されます。CASE=UPPERの場合、小文字はすべて大文字に変換されます。CASE=OPPOSITEとすると、大文字はすべて小文字に変換され、小文字はすべて

大文字に変換されます。

```
CASE=[SAME|LOWER|UPPER|OPPOSITE]
```

このパラメータは、INTYPEファイルに記述されていない識別子(明示的にリストされていない属性または型)のみに作用します。大/小文字の変換は、正当な識別子が生成された後で行われます。

> **注意:**
>
> INTYPE で特定された型の C 言語の構造体識別子の大/小文字の区別は、INTYPE ファイルにおける大/小文字と同じです。たとえば、INTYPE ファイルに次の行が含まれる場合、
>
> ```
> TYPE Worker
> ```

OTTによって次のように生成されます。

```
struct Worker {...};
```

一方で、INTYPEファイルに次のように記述したとします。

```
TYPE wOrKeR
```

OTTによって次のように生成されます。

```
struct wOrKeR {...};
```

これはINTYPEファイルの大/小文字区別どおりです。

INTYPEファイルに記述されていない、大/小文字の区別のないSQL識別子は、CASE=SAMEの場合は大文字で、CASE=OPPOSITEの場合は小文字で指定します。宣言時に引用符が付かなかったSQL識別子には、大/小文字の区別はありません。

19.5.2.11 SCHEMA_NAMES

このパラメータでは、デフォルトのスキーマからの型のデータベース名をOUTTYPEファイル内のスキーマ名で修飾する操作を制御できます。OTT生成のOUTTYPEファイルには、型の名前など、OTTの処理対象となる型の情報が含まれています。

関連項目

- SCHEMA_NAMESの使用方法

19.5.2.12 TRANSITIVE

値TRUE(デフォルト)またはFALSEをとります。INTYPEファイルに明示的に指定されていない型の依存性が変換されるかどうかを示します。TRANSITIVE=FALSEを指定すると、INTYPEファイルに指定されていない型は生成されません。これは、他に生成される型の属性の型として使用される場合も同じです。

19.5.3 OTTパラメータの指定可能な場所

OTTのパラメータは、コマンドライン、コマンドラインで指定するCONFIGファイル内、またはその両方で指定します。パラメータの一部は、INTYPEファイルにも指定できます。

OTTは、次のように起動します。

OTT *username/password* parameters

コマンドラインのパラメータの1つが、

CONFIG=*filename*

の場合、構成ファイル*filename*からその他のパラメータが読み込まれます。

さらに、パラメータは、プラットフォームによって異なる位置にあるデフォルトの構成ファイルから読み込まれます。このファイルの存在は必要ですが、空でもかまいません。構成ファイルには、各パラメータを1行に1つずつ、空白を使用せずに入力する必要があります。

引数を指定しないでOTTを実行すると、オンラインのパラメータ参照が表示されます。

変換されるOTTの型は、INTYPEパラメータで指定したファイルで命名されます。パラメータCASE、INITFILE、INITFUNCおよびHFILEは、INTYPEファイルに入れることもできます。OTT生成のOUTTYPEファイルにはCASEパラメータが含まれ、初期化ファイルが生成されている場合はINITFILEおよびINITFUNCパラメータが含まれます。OUTTYPEファイルでは、型ごとにそれぞれHFILEを指定します。

OTTコマンドの大/小文字区別は、プラットフォームによって異なります。

19.5.4 INTYPEファイルの構造

INTYPEおよびOUTTYPEファイルでは、OTTにより変換される型がリストされ、型または属性名の有効なC言語の識別子への変換方法を決定するときに必要なすべての情報が提供されます。これらのファイルには、1つ以上の型指定を記述します。また、次のオプションを指定する場合もあります。

- CASE
- HFILE
- INITFILE
- INITFUNC

CASE、INITFILEまたはINITFUNCオプションを指定する場合は、すべての型指定よりも前に指定する必要があります。これらのオプションをコマンドラインとINTYPEファイルの両方に指定した場合は、コマンドラインの値が使用されます。

単純なユーザー定義のINTYPEファイルと、そのファイルからOTTにより生成される完全なOUTTYPEファイルの例は、OTTでの型の継承のサポートを参照してください。

19.5.4.1 INTYPEファイルの型指定

INTYPEでの型指定によって、変換するオブジェクト・データ型の名前を指定します。次の記述は、ユーザー作成INTYPEファイルの例です。

```
TYPE employee
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

型指定の構造は、次のとおりです。

```
TYPE type_name [AS type_identifier]
[VERSION [=] version_string]
[HFILE [=] hfile_name]
[TRANSLATE{member_name [AS identifier]}...]
```

*type_name*の構文は、次のとおりです。

```
[schema_name.]type_name
```

この場合、*schema_name*は、指定されたオブジェクト・データ型を所有するスキーマの名前で、*type_name*はその型の名前です。デフォルト・スキーマは、OTTを実行するユーザーのスキーマです。デフォルト・データベースは、ローカル・データベースです。

型指定のコンポーネントは、次のとおりです。

- *type name*はオブジェクト・データ型の名前です。
- *type identifier*は型を表すために使用されるC言語の識別子です。省略すると、デフォルトの名前マッピング・アルゴリズムが使用されます。
- *version string*は、OTTの以前の起動によってコードが生成されたときに使用された型のバージョン文字列です。バージョン文字列はOTTによって生成され、OUTTYPEファイルに書き込まれます。このファイルは、後でOTTを実行するときにINTYPEファイルとして使用できます。バージョン文字列はOTTの操作には影響しませんが、最終的にこれを使用して、起動中のプログラムで使用されるオブジェクト型のバージョンが選択されます。
- *hfile name*は、該当する構造またはクラスの宣言が表れるヘッダー・ファイルの名前です。*hfile name*を省略すると、宣言の生成時にはコマンドラインのHFILEパラメータで指定したファイルが使用されます。
- *member name*は、次の*identifier*に変換される属性(データ・メンバー)の名前です。
- *identifier*はユーザー・プログラムの属性を表すために使用されるC言語の識別子です。この方法によって、任意の数の属性に対して識別子を指定できます。指定していない属性については、デフォルトの名前マッピング・アルゴリズムが使用されます。

オブジェクト・データ型は、次のいずれかの場合に変換する必要があります。

- INTYPEファイルに指定されている場合。
- 変換が必要な別の型を宣言する場合、およびTRANSITIVE=TRUEの場合は必須です。

明示的に記述していない型があり、その型が、正確に1つのファイルのみに宣言した型で必要だとします。この場合、明示的に記述していない型の変換結果は、それを必要とする明示的に宣言した型と同じファイルに書き込まれます。

明示的に記述していない型があり、その型が、複数の異なるファイルに宣言した型で必要だとします。この場合、要求された型の変換結果は、グローバルなHFILEファイルに書き込まれます。

関連項目

- デフォルト名のマッピング

19.5.5 ネストした#includeファイル生成

OTTによって生成されるすべてのHFILEで、他の必要なファイルが#includesを使用してインクルードされ、そのファイルの名前から組み立てられた記号が#definesを使用して定義されます。この記号は、HFILEがすでにインクルードされているかどうかを判断する場合に使用できます。たとえば、次の型を持つデータベースについて考えます。

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

INTYPEファイルの内容は次のとおりです。

```
CASE=lower
type pxl
  hfile tott95a.h
type px3
  hfile tott95b.h
```

OTTを次のように起動するとします。

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

すると、次の2つのヘッダー・ファイルが生成されます。

ファイルtott95b.hは次のとおりです。

```
#ifndef TOTT95B_ORACLE
#define TOTT95B_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#ifndef TOTT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCIRef px3_ref;
struct px3
{
   struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
   OCIInd _atomic;
   struct px1_ind col1
};
typedef struct px3_ind px3_ind;
#endif
```

ファイルtott95a.hは次のとおりです。

```
#ifndef TOTT95A_ORACLE
#define TOTT95A_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
typedef OCIRef px1_ref;
struct px1
{
   OCINumber col1;
   OCINumber col2;
}
typedef struct px1 px1;
struct px1_ind
{
   OCIInd _atomic;
   OCIInd col1;
   OCIInd col2;
}
typedef struct px1_ind px1_ind;
#endif
```

このファイルでは、TOTT95B_ORACLEという記号を最初に定義しています。そのため、プログラマは、次の構造体を使用してtott95b.hを条件付きでインクルードできます。その場合、tott95b.hがインクルード・ファイルに依存しているかどうかを考慮する必要はありません。

```
#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif
```

このテクニックを使用すると、「foo.h」などのファイルから「tott95b.h」をインクルードできます。この場合、「foo.h」からインクルードされる他のファイルに「tott95b.h」がインクルードされているかどうかを確認する必要はありません。

記号TOTT95B_ORACLEの定義の後に、ファイルoci.hが#includeによってインクルードされます。OTT生成のすべてのHFILEには、Pro*C/C++やOCIのプログラマに役立つ型と関数の宣言が含まれたoci.hが組み込まれます。OTTが#includeにかぎ括弧を使用するのは、この場合のみです。

次に、ファイルtott95a.hがインクルードされるのは、tott95a.hに必要な「struct px1」の宣言が含まれているためです。INTYPEファイルにより型宣言の複数ファイルへの書込みが要求される場合は、OTTにより、その他の各HFILEにインクルードする必要があるファイルが決定され、必要な#includeが生成されます。

この#includeでは、OTTにより引用符が使用されるため注意してください。tott95b.hがインクルードされたプログラムをコンパイルすると、tott95a.hの検索はソース・プログラムの検出位置から開始され、それ以降は実装定義の検索規則に従って実行されます。この方法でtott95a.hが検出されない場合は、INTYPEファイル内で完全ファイル名(/で始まるUNIXの絶対パス名など)を使用して、tott95a.hの位置を指定する必要があります。

19.5.6 SCHEMA_NAMESの使用方法

このパラメータでは、OTTの接続先となるデフォルト・スキーマに含まれる型の名前を、OUTTYPEファイル内のスキーマ名で修飾するかどうかを指定します。

デフォルト・スキーマ以外のスキーマに基づく型の名前は、OUTTYPEファイル内のスキーマ名で常に修飾されます。

スキーマ名で修飾するかしないかで、プログラム実行中に型がどのスキーマで検索されるかが決定します。

次の3通りの設定があります。

- SCHEMA_NAMES=ALWAYS(デフォルト)

  OUTTYPEファイル内のすべての型名をスキーマ名で修飾します。

- SCHEMA_NAMES=IF_NEEDED

  デフォルト・スキーマに属するOUTTYPEファイル内の型名はスキーマ名で修飾しません。デフォルト・スキーマ以外のスキーマに属する型名は、スキーマ名で修飾します。

- SCHEMA_NAMES=FROM_INTYPE

  INTYPEファイルに記述されている型は、INTYPEファイル内のスキーマ名で修飾されている場合にのみ、OUTTYPEファイル内のスキーマ名で修飾されます。デフォルト・スキーマ内の型は、INTYPEファイル内で記述されていなくても型の依存性によって生成する必要がある場合にはスキーマ名付きで記述されます。これは、その型に依存する型としてOTTで最初に検出された型がスキーマ名付きで記述されていた場合にのみ行われます。ただし、OTTが接続しているデフォルト・スキーマにない型は、常に、明示的に指定したスキーマ名付きで記述されます。

OTT生成のOUTTYPEファイルは、Pro*C/C++のINTYPEファイルになります。このファイルは、データベース型名をC言語の構造体名に対応付けます。この情報は、構造体内で正しいデータベース型が確実に選択されるようにするために、実行時に使用されます。OUTTYPEファイル(Pro*C/C++のINTYPEファイル)内のスキーマ名で型が指定される場合、その型はプログラムの実行中に名前付きスキーマ内で検索されます。型がスキーマ名なしで表示される場合、そのファイルはプログラムの接続先となるデフォルト・スキーマ内で検出されます。このデフォルト・スキーマは、OTTで使用されたデフォルト・スキーマと異なる場合があります。

例

SCHEMA_NAMESにFROM_INTYPEを設定し、INTYPEファイルで読み込みます。

```
TYPE Person
TYPE joe.Dept
TYPE sam.Company
```

OTTで生成した構造体を使用して、Pro*C/C++アプリケーションで*sam.Company*、*joe.Dept*および*Person*の各型を使用します。*Person*にはスキーマ名が付いていないので、アプリケーションが接続されているスキーマ内のPerson型を参照します。

OTTとアプリケーションの両方がスキーマjoeに接続すると、アプリケーションではOTTと同じ型(joe.Person)が使用されます。OTTがスキーマ*joe*に接続しても、アプリケーションがスキーマmaryに接続した場合、このアプリケーションでは型mary.Personが使用されます。この動作が有効なのは、スキーマjoeとスキーマmaryで同じCREATE TYPE Person文が実行された場合のみです。

一方、アプリケーションでは、どのスキーマに接続されているかに関係なく、joe.Dept型が使用されます。この動作のためには、INTYPEファイルに型名とともにスキーマ名を必ず記述する必要があります。

明示的に指定していない型が、OTTによって変換される場合があります。たとえば、次のSQL宣言があるとします。

```
CREATE TYPE Address AS OBJECT
(
street    VARCHAR2(40),
city      VARCHAR(30),
state     CHAR(2),
zip_code  CHAR(10)
);

CREATE TYPE Person AS OBJECT
(
name      CHAR(20),
age       NUMBER,
addr      ADDRESS
);
```

ここで、そのOTTがスキーマjoeに接続し、SCHEMA_NAMES=FROM_INTYPEが指定され、ユーザーのINTYPEファイルは次のいずれかを含むとします

```
TYPE Person or TYPE joe.Person
```

joe.Address型が指定されていない場合は、*joe.Person*型にネストされたオブジェクト型として使用されます。TYPE joe.PersonがINTYPEファイルに記述されている場合は、OUTTYPEファイルにはTYPE joe.PersonとTYPE joe.Addressが表示されます。INTYPEファイルにType Personが記述されている場合は、TYPE PersonおよびTYPE AddressがOUTTYPEファイルに表示されます。

*joe.Address*型がOTTによって変換された複数の型に埋め込まれていた場合にも、INTYPEファイル内で明示的に指定されていなければ、埋め込まれた*joe.Address*型が最初に検出されたときに、スキーマ名を使用するかどうかが決定されます。なんらかの理由で、*joe.Address*型にはスキーマ名を付け、*Person*型にはスキーマ名を付けない場合は、次のように明示的に要求する必要があります。

```
TYPE      joe.Address
```

これはINTYPE FILEで要求します。

各型を単一のスキーマ内で宣言する通常の場合は、すべての型名をINTYPEファイル内のスキーマ名で修飾することが最も安全です。

19.5.7 デフォルト名のマッピング

OTTでオブジェクト型または属性のC識別子名を作成する場合、OTTは、その名前をデータベース・キャラクタ・セットから有効なC識別子に変換します。最初に、名前はデータベース・キャラクタ・セットからOTTで使用するキャラクタ・セットに変換されます。次に、変換した結果の名前がINTYPEファイルに指定されている場合は、その変換内容が使用されます。それ以外の場合、名

前はCASEオプションで指定したコンパイラのキャラクタ・セットに1文字ずつOTTにより変換されます。この詳細を次に説明します。

OTTによってデータベース・エンティティ名が読み込まれると、データベース・キャラクタ・セットからOTTで使用するキャラクタ・セットに自動的に変換されます。OTTがデータベース・エンティティ名を正常に読み込むには、その名前のすべての文字がOTTのキャラクタ・セット内で検出される必要はあります。しかし、文字によっては2つのキャラクタ・セット間でエンコーディングが異なる場合があります。

必要な文字がOTTで使用するキャラクタ・セットにすべて含まれていることを保証するには、データベース・キャラクタ・セットと同じものを使用するのが最も簡単な方法です。ただし、OTTのキャラクタ・セットは、コンパイラのキャラクタ・セットのスーパーセットである必要があります。つまり、コンパイラのキャラクタ・セットが7ビットASCIIの場合、OTTのキャラクタ・セットはサブセットとして7ビットASCIIを含む必要があります。コンパイラのキャラクタ・セットが7ビットEBCDICの場合は、OTTのキャラクタ・セットはサブセットとして7ビットEBCDICを含む必要があります。OTTで使用されるキャラクタ・セットを指定するには、環境変数NLS_LANGを設定する方法と、プラットフォーム固有の他のメカニズムを使用する方法があります。

OTTがデータベース・エンティティの名前を読み取ると、その名前は、OTTで使用するキャラクタ・セットからコンパイラのキャラクタ・セットに変換されます。名前の変換がINTYPEファイルで指定されている場合は、OTTはその変換を使用します。

それ以外の場合、OTTは名前を次のように変換します。

1. 最初に、OTTのキャラクタ・セットがマルチバイト・キャラクタ・セットの場合、その名前にある、等価のシングルバイト文字を持つマルチバイト文字は、シングルバイト文字に変換されます。
2. 次に、その名前は、OTTのキャラクタ・セットからコンパイラのキャラクタ・セットに変換されます。コンパイラのキャラクタ・セットは、US7ASCIIのように、シングルバイト・キャラクタ・セットです。
3. 最後に、有効になっているCASEオプションに従って、文字の大/小文字が設定されます。そして、C言語の識別子で無効な文字、またはコンパイラのキャラクタ・セットに変換内容がない文字は、アンダースコアに置き換えられます。1文字でもアンダースコアに置き換えられると、OTTから警告メッセージが表示されます。名前に含まれる文字がすべてアンダースコアに置き換えられると、OTTによってエラー・メッセージが表示されます。

文字単位の名前の変換では、コンパイラのキャラクタ・セットにあるアンダースコア、数字またはシングルバイト文字は変更されません。したがって、有効なC言語の識別子は変更されません。

名前の変換ではたとえば、ウムラウトの付いた「o」、アクセント符号の付いた「a」などのシングルバイト文字をそれぞれ「o」や「a」に変換したり、またマルチバイト文字をシングルバイトと等価に変換できます。名前に等価のシングルバイトがないマルチバイト文字がある場合、通常、その名前の変換はエラーとなります。この場合、ユーザーは、INTYPEファイルで名前の変換を指定する必要があります。

OTTでは、同じC言語の名前に複数のデータベース識別子がマップされたために発生する名前の重複は検出されません。また、データベース識別子がC言語のキーワードにマップされる場合に発生する命名の問題も検出されません。

19.5.8 制限事項

OTTを使用する場合は、次の制限が適用されます。

19.5.8.1 ファイル名比較

現在、OTTでは、2つのファイルが同じかどうかは、コマンドラインまたはINTYPEファイルでユーザーが指定したファイル名を比較して判断しています。OTTで2つのファイル名が同一のファイルを参照しているかどうかを判別する必要がある場合は、1つの潜在的な問題が生じます。たとえば、OTT生成のファイルfoo.hに、foo1.hに書き込まれた型の宣言と、/private/smith/foo1.hに書き込まれた別の型の宣言が必要な場合、OTTは2つのファイルが同じであれば#includeを1つ、異なっていれば#includeを2つ生成する必要があります。ただし、実際には、OTTでは2つのファイルが異なるものとみなされ、次のように2つの#includeが生成されます。

```
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1_ORACLE
#include "/private/smith/foo1.h"
#endif
```

ファイルfoo1.hとファイル/private/smith/foo1.hが異なっていれば、最初のファイルのみがインクルードされます。ファイルfoo1.hとファイル/private/smith/foo1.hが同じであれば、#includeは重複して記述されます。

そのため、コマンドラインまたはINTYPEファイルでファイルを複数回記述するときは、各記述で正確に同じファイル名を使用する必要があります。

20 ユーザー・イグジット

この章では、Oracleツール・アプリケーション用のユーザー・イグジットの作成方法を説明します。C言語のサブルーチンを使用すると、SQL*FormsおよびOracle Formsよりも迅速で、しかも簡単に特定の作業を実行できることがわかります。この章のトピックは、次のとおりです:

- ユーザー・イグジット
- ユーザー・イグジットを作成する理由
- ユーザー・イグジットの開発について
- ユーザー・イグジットの作成について
- EXEC TOOLS文
- ユーザー・イグジットのコールについて
- ユーザー・イグジットへのパラメータの引渡しについて
- フォームへの値の返却について
- ユーザー・イグジットの使用例
- ユーザー・イグジットのプリコンパイルおよびコンパイルについて
- サンプル・プログラム: ユーザー・イグジット
- GENXTBユーティリティの使用について
- SQL*Formsへのユーザー・イグジットのリンクについて
- ガイドライン

この章の内容は補足説明です。ユーザー・イグジットの詳細は、*SQL*Forms*開発者リファレンス、*Oracle Forms*リファレンス・マニュアル、*Vol. 2*および各システム固有のOracleドキュメントを参照してください。

20.1 ユーザー・イグジット

ユーザー・イグジットとは、特別な目的の処理を実行するために作成するC言語のサブルーチンを指します。このサブルーチンはOracle Formsによってコールされます。ユーザー・イグジットにSQL文およびPL/SQLブロックを組み込み、その後ホスト・プログラムの場合と同様にこれをプリコンパイルできます。

Oracle Formsバージョン3のトリガーからユーザー・イグジットをコールすると、ユーザー・イグジットが実行され、その後ステータス・コードがOracle Formsに戻されます。ユーザー・イグジットでは、Oracle Formsステータス行へのメッセージ表示、フィールド値の取得と設定、高速計算と表参照およびOracleデータの操作ができます。

20.2 ユーザー・イグジットを作成する理由

SQL*Formsでは、トリガー内でPL/SQLブロックを使用できます。したがって、ほとんどの場合、ユーザー・イグジットをコールするかわりに、PL/SQLのプロシージャ機能を使用できます。ユーザー・イグジットが必要になったときは、USER_EXIT関数を使用するとPL/SQLブロックからユーザー・イグジットをコールできます。SQL、PL/SQLまたはSQL*Formsのコマンドに比べて、ユーザー・イグジットは記述も実装も複雑です。したがって、SQL、PL/SQLおよびSQL*Formsの範囲を超える処理を実行する場合にかぎり、ユーザー・イグジットを使用することになります。通常は次のような処理に使用します。

- C言語などの第3世代の言語内で実行すると迅速かつ簡単になる演算(数値積分など)

- リアルタイム・デバイスや処理の制御(プリンタまたはグラフィックス・デバイスへの命令の発行など)
- 拡張プロシージャ機能が必要なデータ操作(再帰ソートなど)
- 特殊なファイルI/O操作

20.3 ユーザー・イグジットの開発について

この項ではSQL*Formsユーザー・イグジットの開発方法の概要を示します。詳細はこの後の項で説明します。

ユーザー・イグジットをフォームに組み込むには、次の手順に従います。

- Pro*Cでユーザー・イグジットを記述します。
- ソース・コードをプリコンパイルします。
- 手順2で生成された.cファイルをコンパイルします。
- GENXTBユーティリティを使用してデータベース表IAPXTBを作成します。
- SQL*FormsのGENXTBフォームを使用して、ユーザー・イグジット情報を表に挿入します。
- GENXTBユーティリティを使用して表から情報を読み込み、IAPXITソース・コード・モジュールを作成します。次に、ソース・コード・モジュールをコンパイルします。
- 標準SQL*Formsモジュール、ユーザー・イグジットのオブジェクト、手順6で作成したIAPXITオブジェクトをリンクして、新しいSQL*Forms実行可能プログラムを作成します。
- このフォーム内に、ユーザー・イグジットをコールするためのトリガーを定義します。
- オペレータがフォームを実行する場合、新しいIAPを使用するように通知してください。新しいIAPが標準のものに置き換わる場合、これは不要です。詳細は、各システム固有のOracleインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

関連項目

- EXEC TOOLS文

20.4 ユーザー・イグジットの作成について

次の種類の文を使用すると、SQL*Formsユーザー・イグジットを記述できます。

- C言語のコード
- EXEC SQL
- EXEC ORACLE
- EXEC TOOLS

この項では、SQL*Formsとユーザー・イグジットの間での値の引渡しを可能にするEXEC TOOLS文を中心に説明します。

20.4.1 変数の要件

EXEC TOOLS文で使用される変数は、フォーム定義で使用されるフィールド名に対応している必要があります。

20.5 EXEC TOOLS文

EXEC TOOLS文は、ユーザー・イグジットからの読込み、設定および例外コールバックを処理する包括的な方法を提供するこ

とにより、基本的なOracle Toolsetをサポートします。次の説明はOracle Formsが中心になっていますが、Oracle Reportについても概念は同じです。

20.5.1 Toolsetユーザー・イグジットの作成について

EXEC SQL、EXEC ORACLEおよびホスト言語文の他にも、次のEXEC TOOLS文を使用してOracle Formsユーザー・イグジットを記述できます。

- SET
- GET
- SET CONTEXT
- GET CONTEXT
- MESSAGE

EXEC TOOLS GET文およびEXEC TOOLS SET文は、Oracle Formsの以前のバージョンで使用されていたEXEC IAF GET文およびEXEC IAF PUT文に相当します。ただし、IAF GETおよびIAF PUTとは異なり、TOOLS GETおよびTOOLS SETは標識変数を受け付けます。EXEC TOOLS MESSAGE文は、メッセージ処理関数*sqliem*に相当します。次に、すべてのEXEC TOOLS文について簡単に説明します。詳細は、*Oracle Forms*リファレンス・マニュアル、*Vol. 2*を参照してください。

20.5.2 EXEC TOOLS SET

EXEC TOOLS SET文はユーザー・イグジットからOracle Formsに値を渡します。具体的には、ホスト変数および定数の値をOracle Formsの変数および項目に割り当てます。フォーム項目に渡された値は、ユーザー・イグジットで制御がフォームに戻った後に表示されます。EXEC TOOLS SET文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET form_variable[, ...]
     VALUES ({:host_variable :indicator | constant}[, ...]);
```

*form_variable*は、Oracle Formsフィールド、block.field、システム変数、グローバル変数、またはこれらの項目のうち1つの値を含むホスト変数(先頭コロン付き)です。次の例では、ユーザー・イグジットによって従業員名をOracle Formsに渡します。

```
char ename[20];
short ename_ind;

...

    strcpy(ename, "MILLER");
    ename_ind = 0;
    EXEC TOOLS SET emp.ename VALUES (:ename :ename_ind);
```

この例で、*emp.ename*はOracle Formsのblock.fieldです。

20.5.3 EXEC TOOLS GET

EXEC TOOLS GET文は、Oracle Formsからユーザー・イグジットに値を渡します。具体的には、Oracle Formsの変数および項目の値を、ホスト変数に割り当てます。値が渡されるとすぐに、ユーザー・イグジットではそれらを任意の目的に使用できます。EXEC TOOLS GET文を記述するときは次の構文を使用してください。

```
EXEC TOOLS GET form_variable[, ...]
    INTO :host_variable:indicator[, ...];
```

*form_variable*は、Oracle Formsフィールド、block.field、システム変数、グローバル変数、またはこれらの項目のうち1つの値を含むホスト変数(先頭コロン付き)です。次の例では、Oracle Formsはブロックからユーザー・イグジットに項目名を渡します。

```
...
char     name_buff[20];
VARCHAR  name_fld[20];

strcpy(name_fld.arr, "EMP.NAME");
name_fld.len = strlen(name_fld.arr);
EXEC TOOLS GET :name_fld INTO :name_buff;
```

20.5.4 EXEC TOOLS SET CONTEXT

EXEC TOOLS SET CONTEXT文は、後で別のユーザー・イグジットで使用するために、ユーザー・イグジットからのコンテキスト情報を保存します。ポインタ変数は、コンテキスト情報が格納されているメモリーのブロックを指します。SET CONTEXT文と併用するときは、情報を保持するためのグローバル変数を宣言する必要はありません。EXEC TOOLS SET CONTEXT文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
    IDENTIFIED BY context_name;
```

*context_name*は、未宣言の識別子またはコンテキスト領域を命名する文字ホスト変数(先頭コロン付き)です。

```
...
char  *context_ptr;
char  context[20];

strcpy(context, "context1")
EXEC TOOLS SET CONTEXT :context IDENTIFIED BY application1;
```

20.5.5 EXEC TOOLS GET CONTEXT

EXEC TOOLS GET CONTEXT文は、(SET CONTEXTによって保存されている)コンテキスト情報をユーザー・イグジットに取り出します。ホスト言語ポインタ変数は、コンテキスト情報が格納されているメモリーのブロックを指します。EXEC TOOLS GET CONTEXT文を記述するには、次の構文を使用します。

```
EXEC TOOLS GET CONTEXT context_name
    INTO :host_pointer_variable;
```

*context_name*は、未宣言の識別子またはコンテキスト領域を命名する文字ホスト変数(先頭コロン付き)です。次の例では、ユーザー・イグジットは前に保管されたコンテキスト情報を取り出します。

```
...
char  *context_ptr;

EXEC TOOLS GET CONTEXT application1 INTO :context_ptr;
```

20.5.6 EXEC TOOLS MESSAGE

EXEC TOOLS MESSAGE文は、ユーザー・イグジットからOracle Formsにメッセージを渡します。このメッセージは、ユーザー・イグジットからフォームに制御が戻った後に、Oracle Formsのメッセージ行に表示されます。EXEC TOOLS MESSAGE文を記述するには、次の構文を使用します。

```
EXEC TOOLS MESSAGE message_text [severity_code];
```

*message_text*は引用文字列または文字ホスト変数(先頭コロン付き)で、オプションの*severity_code*は整数定数または整数ホスト変数(先頭コロン付き)です。MESSAGE文では、インジケータ変数は受け入れられません。次の例では、ユーザー・イグジットからエラー・メッセージがOracle Formsに渡されます。

```
EXEC TOOLS MESSAGE 'Bad field name! Please reenter.';
```

20.6 ユーザー・イグジットのコールについて

SQL*Formsトリガーからユーザー・イグジットをコールするには、USER_EXIT(SQL*Formsで提供されます)という名前のパッケージ・プロシージャを使用します。次の構文を使用します。

```
USER_EXIT(user_exit_string [, error_string]);
```

*user_exit_string*には、ユーザー・イグジットの名前とオプションのパラメータを指定し、*error_string*には、ユーザー・イグジットが異常終了したときにSQL*Formsにより発行されるエラー・メッセージを指定します。たとえば、次のトリガー・コマンドは、LOOKUPという名前のユーザー・イグジットをコールします。

```
USER_EXIT('LOOKUP');
```

ユーザー・イグジットの文字列は、一重(二重ではない)引用符で囲まれていることに注意してください。

20.7 ユーザー・イグジットへのパラメータの引渡しについて

ユーザー・イグジットをコールすると、SQL*Formsは自動的に次のパラメータをユーザー・イグジットに渡します。

パラメータ	説明
コマンドライン	ユーザー・イグジット文字列。
コマンドラインの長さ	ユーザー・イグジット文字列の長さ(文字数)。
エラー・メッセージ	定義済の場合、エラー文字列(障害メッセージ)。
エラー・メッセージの長さ	エラー文字列の長さ。
問合せモード	ブール値。ユーザー・イグジットのコールが通常モードと問合せモードのどちらで行われたかを示します。

しかし、ユーザー・イグジット文字列を使用すると、追加パラメータをユーザー・イグジットに渡せます。たとえば次のトリガー・コマンドを使用すると、2つのパラメータと1つのエラー・メッセージがユーザー・イグジットLOOKUPに渡されます。

ユーザー・イグジットの文字列は、一重(二重ではない)引用符で囲まれていることに注意してください。

```
USER_EXIT('LOOKUP 2025 A', 'Lookup failed');
```

この機能を使用すれば、次の例のように、フィールド名をユーザー・イグジットに渡せます。

```
USER_EXIT('CONCAT firstname, lastname, address');
```

ただし、ユーザー・イグジット文字列を解析は、SQL*Formsではなく、ユーザー・イグジットに依存します。

20.8 フォームへの値の返却について

ユーザー・イグジットでは、SQL*Formsに制御が戻るとき、成功したか、失敗したかまたは致命的エラーが発生したかのいずれの状態かを示すコードも必ず戻します。このリターン・コードはSQL*Formsによって定義される整数定数です(次の項を参照)。この3種類の結果はの意味は次のとおりです。

結果	意味
成功	ユーザー・イグジットでエラーが発生しませんでした。コール元の逆戻りリターン・コード・スイッチが設定されていなければ、SQL*Forms は成功ラベルまたは次のステップに進みます。
失敗	ユーザー・イグジットで、フィールド内の無効値などのエラーが検出されました。このイグジットによって渡された任意指定のメッセージが、SQL*Forms 画面下部のメッセージ行およびエラー表示画面に表示されます。SQL*Forms は行に影響を与えない SQL 文に対するときと同様に応答します。
致命的エラー	ユーザー・イグジットで、SQL 文中の実行エラーなど、それ以上処理を続行できない条件が検出されました。イグジットによって渡されたオプションのエラー・メッセージが、SQL*Forms のエラー表示画面に表示されます。SQL*Forms は SQL 文内の致命的エラーに対するときと同様に応答します。ユーザー・イグジットでフィールドの値が変更された後で、失敗または致命的エラーのコードが戻された場合、SQL*Forms はこの変更を廃棄しません。また、逆戻りリターン・コード・スイッチが設定されていて、成功コードが戻されたときにも、SQL*Forms は変更を廃棄しません。

20.8.1 IAP定数

リターン・コードとして使用する3つの記号定数がSQL*Formsによって定義されます。ホスト言語に応じて、これらの定数には文字IAPまたはSQLという接頭辞が付きます。たとえばIAPSUCC、IAPFAIL、IAPFTLなどと定義されます。

20.8.2 WHENEVERの使用について

イグジット内でWHENEVER文を指定すると、不当なデータ型の変換(SQLERROR)、フォーム・フィールドにPUTされた値の切捨て(SQLWARNING)および行を戻さない問合せ(NOT FOUND)を検出できます。

20.9 ユーザー・イグジットの使用例

次の例では、EXEC TOOLS GETルーチンとEXEC TOOLS PUTルーチン、および*EXEC TOOLS MESSAGE*関数を使用するユーザー・イグジットの記述方法を示します。

```
int
myexit()
{
    char field1[20], field2[20], value1[20], value2[20];
    char result_value[20];
    char errmsg[80];
    int errlen;

    #include sqlca.h
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    /* get field values into form */
```

```
    EXEC TOOLS GET :field1, :field2 INTO :value1, :value2;
    /* manipulate the values to obtain result_val */
    ...
    /* put result_val into form field result */
    EXEC TOOLS PUT result VALUES (:result_val);
    return IAPSUCC;   /* trigger step succeeded */

sql_error:
    strcpy(errmsg, CONCAT("MYEXIT", sqlca.sqlerrm.sqlerrmc);
    errlen = strlen(errmsg);
    EXEC TOOLS MESSAGE :errmsg ; /* send error msg to Forms */
    return IAPFAIL;
```

20.10 ユーザー・イグジットのプリコンパイルおよびコンパイルについて

ユーザー・イグジットは、スタンドアロン型のホスト・プログラムと同様にプリコンパイルされます。ユーザー・イグジットのコンパイル方法については、システム固有のOracleインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

関連項目

- プリコンパイラのオプション

20.11 サンプル・プログラム: ユーザー・イグジット

次の例にユーザー・イグジットを示します。

```
/*************************************************************
Sample Program 5:  SQL*Forms User Exit

This user exit concatenates form fields.  To call the user
exit from a SQL*Forms trigger, use the syntax

   user_exit('CONCAT field1, field2, ..., result_field');

where user_exit is a packaged procedure supplied with SQL*Forms
and CONCAT is the name of the user exit.  A sample form named
CONCAT invokes the user exit.
*************************************************************/

#define min(a, b) ((a < b) ? a : b)
#include <stdio.h>
#include <string.h>

/* Include the SQL Communications Area, a structure through which
 * Oracle makes runtime status information such as error
 * codes, warning flags, and diagnostic text available to the
 * program.
 */
#include <sqlca.h>

/* All host variables used in embedded SQL in this example
 * appear in the Declare Section.
 */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR   field[81];
    VARCHAR   value[81];
    VARCHAR   result[241];
EXEC SQL END DECLARE SECTION;
```

```
/* Define the user exit, called "concat". */
int concat(cmd, cmdlen, msg, msglen, query)
char *cmd;     /* command line in trigger step ("CONCAT...") */
int  *cmdlen;  /* length of command line */
char *msg;     /* trigger step failure message from form */
int  *msglen;  /* length of failure message */
int  *query;   /* TRUE if invoked by post-query trigger,
                  FALSE otherwise */
{
    char *cp = cmd + 7;    /* pointer to field list in
                              cmd string; 7 characters
                              are needed for "CONCAT " */
    char *fp = (char*)&field.arr[0];  /* pointer to a field name in
                                         cmd string */
    char  errmsg[81];      /* message returned to SQL*Forms
                              on error */
    int   errlen;          /* length of message returned
                              to SQL*Forms */

/* Branch to label sqlerror if an ORACLE error occurs. */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;

    result.arr[0] = '¥0';

/* Parse field names from cmd string. */
    for (; *cp != '¥0'; cp++)
    {
       if (*cp != ',' && *cp != ' ')
           /* Copy a field name into field.arr from cmd. */
       {
           *fp = *cp;
           fp++;
       }
       else
           if (*cp == ' ')
           {   /* Have whole field name now. */
               *fp = '¥0';
               field.len = strlen((char *) field.arr);
               /* Get field value from form. */
               EXEC TOOLS GET :field INTO :value;
               value.arr[value.len] = '¥0';
               strcat((char *) result.arr, (char *) value.arr);
               fp = (char *)&field.arr[0];  /* Reset field pointer. */
           }
    }

/* Have last field name now. */
    *fp = '¥0';
    field.len = strlen((char *) field.arr);
    result.len = strlen((char *) result.arr);

/* Put result into form. */
    EXEC TOOLS PUT :field VALUES (:result);

/* Trigger step succeeded. */
    return(IAPSUCC);

sqlerror:
```

```
    strcpy(errmsg, "CONCAT: ");
    strncat(errmsg, sqlca.sqlerrm.sqlerrmc, min(72,
        sqlca.sqlerrm.sqlerrml));
    errlen = strlen(errmsg);
/* Pass error message to SQL*Forms status line. */
     EXEC TOOLS MESSAGE :errmsg ;
    return(IAPFAIL);  /* Trigger step failed. */
}
```

20.12 GENXTBユーティリティの使用について

IAPXITモジュール内のIAPプログラム表IAPXTBには、IAP内にリンクされているそれぞれのユーザー・イグジット用のエントリが格納されています。IAPXTBはIAPに各ユーザー・イグジットの名前、位置およびホスト言語を指示します。新しいユーザー・イグジットをIAPに追加するときは、対応するエントリをIAPXTBに追加する必要があります。IAPXTBは、IAPXTBという同じ名前のデータベース表から導出されます。次のように、オペレーティング・システムのコマンドラインでGENXTBフォームを実行することで、データベースの表を修正できます。

```
RUNFORM GENXTB username/password
```

定義するそれぞれのユーザー・イグジットについて、次の情報を入力できるフォームが表示されます。

- イグジット名
- C言語コード
- 作成日
- 最終変更日
- コメント

IAPXTBデータベース表を変更してから、GENXTBユーティリティを使用して表を読み取り、IAPXITモジュールとそれに含まれるIAPXTBプログラム表を定義するアセンブラまたはC言語のソース・プログラムを作成します。使用するソース言語は、オペレーティング・システムによって異なります。GENXTBユーティリティを実行するための構文は次のとおりです。

```
GENXTB username/password outfile
```

*outfile*は、GENXTBが作成するアセンブラまたはC言語のソース・プログラムに指定する名前です。

関連項目

- ガイドライン

20.13 SQL*Formsへのユーザー・イグジットのリンクについて

ユーザー・イグジットをコールするフォームの実行前に、フォームを実行するSQL*FormsのコンポーネントであるIAPにこのユーザー・イグジットをリンクする必要があります。ユーザー・イグジットは標準的なバージョンのIAPにリンクしたり、そのイグジットをコールするフォーム用の特別なバージョンのIAPにリンクしたりできます。

IAPの実行可能コピーを新規に作成するには、OracleライブラリおよびC言語のリンク・ライブラリからユーザー・イグジット・オブジェクト・モジュール、標準IAPモジュール、IAPXITモジュール、その他必要なモジュールをすべてリンクします。

リンク方法はシステムによって異なります。システム固有のOracleインストレーション・ガイドまたはユーザーズ・ガイドで確認してください。

20.14 ガイドライン

この項では、一般的な問題を回避するためのガイドラインを示します。

20.14.1 イグジットの命名について

ユーザー・イグジットの名前をOracleの予約語にしないでください。また、SQL*Formsコマンド、関数コード、SQL*Formsで使用される外部定義の名前と競合する名前も使用しないでください。ソース・コード内のユーザー・イグジット・エントリ・ポイントの名前は、ユーザー・イグジット自体の名前になります。このイグジット名は有効なC言語の関数名であり、同時に使用しているオペレーティング・システムの規則に従った有効なファイル名である必要があります。

SQL*Formsでは、ユーザー・イグジットの検索前に、その名前が大文字に変換されます。したがって、イグジット名はソース・コード内では大文字になっている必要があります。

20.14.2 Oracleへの接続について

ユーザー・イグジットでは、SQL*Formsにより確立された接続を使用してOracleと通信します。ただし、ユーザー・イグジットでOracle Net Servicesを使用して任意のデータベースに追加の接続を確立できます。

関連項目

- 高度な接続オプション

20.14.3 I/Oコールの発行について

ファイルI/Oはサポートされていますが、画面I/Oはサポートされていません。

20.14.4 ホスト変数の使用方法について

スタンドアロン型のプログラムでの変数の使用に対する制限事項は、ユーザー・イグジットにも適用されます。EXEC SQLおよびEXEC TOOLS文内では、ホスト変数の前にコロン(:)が必要です。ただし、EXEC TOOLS文では、ホスト配列は使用できません。

20.14.5 表の更新について

一般に、ユーザー・イグジットではフォームに対応付けられたデータベースの表をUPDATEしないでください。たとえば、SQL*Forms作業領域でオペレータがレコードを更新した後で、対応付けられたデータベース表内の対応する行をUPDATEしたとします。このときトランザクションがCOMMITされると、SQL*Forms作業領域内のレコードがその表に適用され、ユーザー・イグジットのUPDATEは上書きされます。

20.14.6 コマンドの発行について

Oracleでは、ユーザー・イグジットで実行された作業に限らず、SQL*Formsのオペレータが開始した作業もコミットまたはロールバックされるため、ユーザー・イグジットからはCOMMITまたはROLLBACKコマンドを発行しないでください。かわりに、SQL*FormsトリガーからCOMMITまたはROLLBACKを発行してください。データ定義コマンド(ALTER、CREATE、GRANTなど)についても同様で、これらのコマンドでも、実行の前後に暗黙的なCOMMITが発行されるためです。

付録

ここには次の付録が含まれます。

- 予約語、キーワードおよびネームスペース
- パフォーマンス・チューニング
- 構文および意味検査
- システム固有の参照
- 埋込みSQL文およびディレクティブ
- サンプル・プログラム
- Pro*C/C++のMicrosoft Visual Studio .NETへの統合

A 予約語、キーワードおよびネームスペース

この付録の内容は次のとおりです。

- 予約語およびキーワード
- Oracleの予約済ネームスペース

A.1 予約語およびキーワード

一部の語はOracleで予約されています。つまり、それらのワードはOracleにとって特別な意味を持っており、再定義できません。このため、それらのワードを使用して、列、表、索引などのデータベース・オブジェクトに名前を付けることはできません。SQLおよびPL/SQLのOracle予約語リストは、『*Oracle Database SQL言語リファレンス*』および『*Oracle Database PL/SQL言語リファレンス*』を参照してください。

Pro*C/C++キーワードは、CまたはC++キーワードと同様、プログラムの中で変数として使用しないでください。変数として使用するとエラーが発生します。キーワードを列などのデータベース・オブジェクト名として使用すると、エラーが発生することがあります。Pro*C/C++で使用されるキーワードは次のとおりです。

キーワード	キーワード	キーワード
all	allocate	alter
analyze	and	any
append	arraylen	as
asc	assign	at
audit	authorization	avg
begin	between	bind
both	break	buffer
buffering	by	call
cast	char	character
character_set_name	charf	charz
check	chunksize	close
collection	comment	コミット

キーワード	キーワード	キーワード
connect	constraint	constraints
context	continue	convbufsz
copy	count	create
current	currval	cursor
data	database	date
dateformat	datelang	datetime_interval_code
datetime_interval_precision	day	deallocate
dec	decimal	declare
default	define	delete
deref	desc	describe
descriptor	directory	disable
display	distinct	do
double	drop	eject
else	enable	end
endif	end-exec	erase
escape	exec	execute
exists	explain	extract
fetch	file	fileexists
filename	first	float

キーワード	キーワード	キーワード
flush	for	force
found	free	from
関数	get	global
go	goto	grant
group	having	hold
host_stride_length	hour	iaf
identified	ifdef	ifndef
immediate	in	include
indicator	indicator_stride_length	input
insert	整数	internal_length
intersect	interval	into
is	isopen	istemporary
last	leading	length
level	like	list
load	lob	local
ロック	long	max
message	min	minus
minute	mode	month
multiset	name	national_character
nchar	nchar_cs	next

キーワード	キーワード	キーワード
nextval	noaudit	not
notfound	nowait	null
nullable	NUMBER	numeric
nvarchar	nvarchar2	object
ocibfilelocator	ocibloblocator	ocicloblocator
ocidate	ociextproccontext	ocinumber
ociraw	ocirowid	ocistring
octet_length	of	one
only	open	option
or	oracle	order
output	overlaps	overpunch
package	partition	perform
精度	prepare	prior
procedure	put	raw
read	real	ref
reference	register	release
rename	replace	return
returned_length	returned_octet_length	returning
revoke	role	rollback
rowid	rownum	savepoint

キーワード	キーワード	キーワード
位取り	second	section
select	set	skip1
skip2	skip3	smallint
some	sql	sql_context
sql_cursor	sqlerror	sqlwarning
start	statement	stddev
stop	string	sum
sysdate	sysdba	sysoper
table	temporary	スレッド
time	timestamp	timezone_hour
timezone_minute	to	tools
title	trailing	transaction
trigger	trim	truncate
type	uid	ulong_varchar
union	unique	unsigned
user_defined_type_name	user_defined_type_name_length	user_defined_type_schema
user_defined_type_schema_length	user_defined_type_version	update
使用	user	using
validate	value	values

キーワード	キーワード	キーワード
var	varchar	varchar2
変数	variance	varnum
varraw	view	whenever
パラメータは次のとおりです。	with	work
write	year	zone

A.2 Oracleの予約済ネームスペース

Oracleによって予約されているネームスペースのリストを次の表に示します。Oracleライブラリ内の関数名の最初の文字列は、このリストの文字列に制限されています。名前が競合する可能性があるため、これらの文字列で始まる関数名を付けないでください。たとえば、Oracle Net Transparent Network Service関数はすべてNSで始まるため、関数にはNSで始まる名前を付けないように注意する必要があります。

表A-1 Oracleの予約済ネームスペース

ネームスペース	ライブラリ
XA	XA アプリケーション専用の外部関数
SQ	Oracle プリコンパイラおよび SQL*モジュール・アプリケーションが使用する外部 SQLLIB 関数
O、OCI	外部 OCI 関数、内部 OCI 関数
UPI、KP	Oracle UPI レイヤーの関数名
NA	Oracle Net ネイティブ・サービス・プロダクト
NC	Oracle Net RPC プロジェクト
ND	Oracle Net ディレクトリ
NL	Oracle Net ネットワーク・ライブラリ・レイヤー
NM	Oracle Net 管理プロジェクト
NR	Oracle Net インターチェンジ
NS	Oracle Net トランスペアレント・ネットワーク・サービス

ネームスペース	ライブラリ
NT	Oracle Net ドライバ
NZ	Oracle Net セキュリティ・サービス
OSN	Oracle Net V1
TTC	Oracle Net 2 タスク
GEN、L、ORA	Core ライブラリ関数
LI、LM、LX	Oracle グローバリゼーション・サポート・レイヤーの関数名
S	システム依存ライブラリの関数名

表のリストはOracle予約済ネームスペースのすべての関数を包括的に示したものではありません。特定のネームスペースのすべての関数リストは、該当するOracleライブラリに対応したドキュメントを参照してください。

B パフォーマンス・チューニング

この付録では、アプリケーションのパフォーマンス(性能)が向上する簡単な適用方法をいくつか紹介します。これらの方法を使用すると、多くの場合、処理時間を25%以上短縮できます。この付録の内容は次のとおりです。

- パフォーマンスを低下させる原因
- パフォーマンスの改善方法
- ホスト配列の使用について
- 埋込みPL/SQLの使用について
- SQL文の最適化について
- 文キャッシュについて
- 索引の使用について
- 行レベル・ロックの利用について
- 不要な解析の排除について
- 接続プーリングの使用方法について
- Traffic DirectorモードのOracle Connection Managerの使用について

B.1 パフォーマンスを低下させる原因

パフォーマンスを低下させる原因の1つは通信オーバーヘッドが高いことです。サーバーでは、SQL文を一度に1つずつ処理する必要があります。つまり、文ごとに個別のコールが発生し、単一のオーバーヘッドが増加します。ネットワーク化された環境下では、ネットワークを介してSQL文を送信する必要があるため、ネットワークの通信量が増加することになります。ネットワークの通信量が多いと、アプリケーションの処理速度は著しく低下します。

パフォーマンスを低下させるもう1つの原因は、非効率的なSQL文です。SQLは非常に柔軟性があるため、2つの異なる文で同じ結果を得ることができますが、一方の文の効率が悪い場合があります。たとえば、次の2つのSELECT文は同じ行(少なくとも従業員が1人いる部門ごとの名称および番号)を戻します。

```
EXEC SQL SELECT dname, deptno
    FROM dept
    WHERE deptno IN (SELECT deptno FROM emp);

EXEC SQL SELECT dname, deptno
    FROM dept
    WHERE EXISTS
    (SELECT deptno FROM emp WHERE dept.deptno = emp.deptno);
```

ただし、この最初の文はDEPT表内のすべての部門番号を探してEMP表全体をスキャンするため、処理に時間がかかります。EMP表内のDEPTNO列に索引を付けていても、この副問合せにはDEPTNOを指定するWHERE句がないので、索引は使用されません。

パフォーマンス低下の3つ目の原因は、不要な解析およびバインドです。SQL文を実行する前に、サーバーでこのSQL文を解析してバインドする必要があることに注意してください。解析とは、SQL文を調べて、これが構文規則に従って正しいデータベース・オブジェクトを参照していることを確認する作業です。バインドとは、SQL文内のホスト変数をそれぞれのアドレスに対応付け、サーバーがその値に対して読込みまたは書込みができるようにする処理です。

大部分のアプリケーションにおいて、カーソルの管理を十分に行っているとはいえません。このため不要な解析またはバインドが発生し、結果的に処理のオーバーヘッドが著しく増加します。

B.2 パフォーマンスの改善方法

プリコンパイルしたプログラムのパフォーマンスがよくない場合でも、オーバーヘッドを減少させる方法はあります。

特にネットワーク化された環境下では、次の処理によって通信オーバーヘッドを大幅に削減できます。

- ホスト変数の使用
- 埋込みPL/SQLの使用

次の方法で、処理のオーバーヘッドを場合によっては大幅に削減できます。

- SQL文の最適化
- 索引の使用
- 行レベル・ロックの利用
- 不要な解析の排除
- 不要な再解析の回避

次の項目では、オーバーヘッドを削減するための方法を検討します。

B.3 ホスト配列の使用について

ホスト配列を使用すると、1つのSQL文でデータの集まり全体を操作できるため、パフォーマンスが向上します。たとえば、300人の従業員の給料をEMP表にINSERTする場合を考えてみます。配列がないと、プログラムは300の個々のINSERT(各従業員に1つ)を実行する必要があります。配列を使用すると、必要なINSERTは1回のみになります。次の文を考えてみます。

```
EXEC SQL INSERT INTO emp (sal) VALUES (:salary);
```

*salary*が単純なホスト変数の場合は、サーバーでこのINSERT文を1回実行すると、EMP表には1行のみが挿入されます。この行のSAL列には*salary*の値が格納されます。この方法で300行を挿入するには、このINSERT文を300回実行する必要があります。

しかし、*salary*がサイズ300のホスト配列の場合は、一度に300行すべてがEMP表に挿入されます。各行のSAL列には*salary*配列の要素の値が格納されます。

関連項目

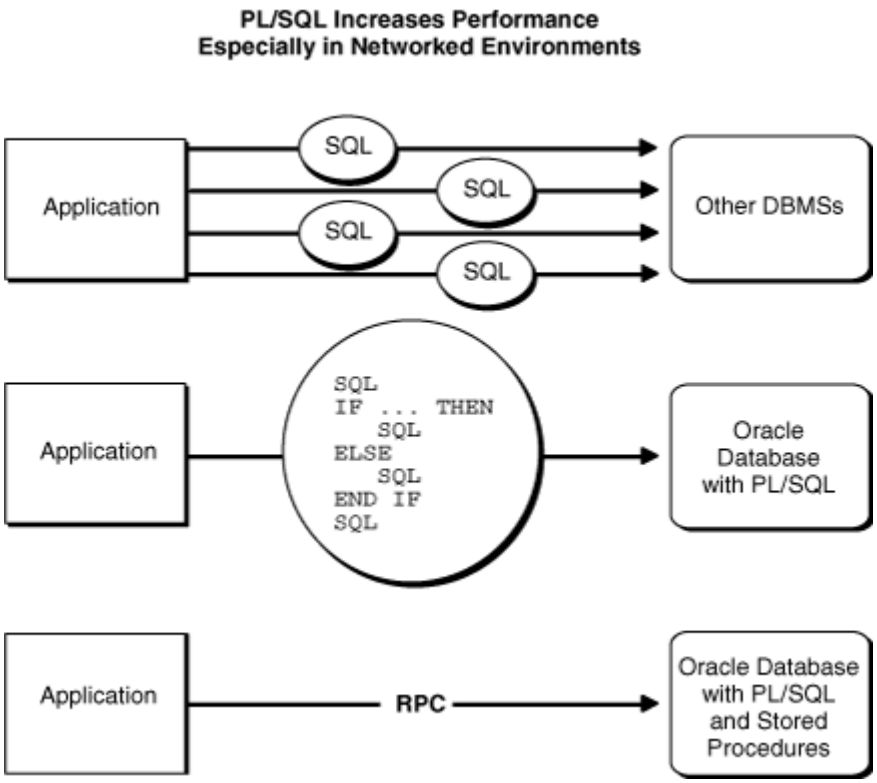
- ホスト配列

B.4 埋込みPL/SQLの使用について

図B-1のように、アプリケーションがデータベース集約型であれば、制御構造体を使用してPL/SQLブロック内でSQL文をグループ化し、ブロック全体をデータベース・サーバーに送ることができます。これによってアプリケーションとデータベース・サーバーとの間の通信量は大幅に減少します。

また、PL/SQLサブプログラムを使用してアプリケーションからサーバーへのコールを少なくすることもできます。たとえば、10個のSQL文を個々に実行するには、10回のコールが必要ですが、10個のSQL文を含む1つのサブプログラムを実行する場合、コールは1回で済みます。

図B-1 PL/SQLによるパフォーマンスの向上



PL/SQLは、Oracle FormsなどのOracleアプリケーション開発ツールでも使用できます。PL/SQLでこれらのツールに手続き型処理能力を追加することで、パフォーマンスが向上します。PL/SQLを使用すると、Oracleのツール製品ではデータベース・サーバーをコールせずに、すべての計算を迅速かつ効率的に処理できます。この結果、時間が節約され、ネットワークの通信量が減少します。

関連項目

- 埋込みPL/SQL
- Oracle Database PL/SQL言語リファレンス

B.5 SQL文の最適化について

Oracleオプティマイザにより、すべてのSQL文について実行計画(その文を実行するためにサーバーが行う一連の手順)が生成されます。これらの手順は、『Oracle Databaseアプリケーション開発者ガイド - 基礎編』に記載されたルールによって決まります。これらのルールに従うと、最適なSQL文を作成できます。

B.5.1 オプティマイザ・ヒント

場合によっては、サーバーに対してSQL文を最適化する方法を示すことができます。このようにして示す内容はヒントと呼ばれ、これによりオプティマイザによる決定に運用側から影響を与えることができます。

ヒントはディレクティブではなく、オプティマイザによるジョブの実行を助けるだけです。一部のヒントはSQL文を最適化するために使用される情報の範囲を制限し、他のヒントは全体的な戦略を提案します。

ヒントを使用して、次の事項を指定できます。

- SQL文の最適化アプローチ
- 参照されているそれぞれの表へのアクセス・パス
- 結合のための結合順序
- 表を結合する方法

つまり、ヒントは次の4つのカテゴリに分けられます。

- 最適化アプローチ
- アクセス・パス
- 結合順序
- 結合操作

たとえば、2つの最適化アプローチ・ヒントであるCOSTとNOCOSTは、コストベースのオプティマイザとルールベースのオプティマイザをそれぞれ起動します。

SELECT、UPDATE、INSERTあるいはDELETE文の動詞の直後にC言語のスタイルのコメントを記述して、オプティマイザにヒントを与えます。たとえば、オプティマイザは次の文でコストベースのアプローチを使用します。

```
SELECT /*+ COST */ ename, sal INTO ...
```

C++コードでは、//+という形式のオプティマイザ・ヒントも認識されます。

関連項目

- エディションベースの再定義の使用

B.5.2 トレース機能

SQLトレース機能とEXPLAIN PLAN文を使用すると、アプリケーションの処理速度を低下させるおそれのあるSQL文を特定できます。

SQLトレース機能は、実行された各SQL文についての統計情報を生成します。この統計表示で、最も処理時間のかかる文がどれか判断できます。このため、それらの文の処理効率のチューニングに専念できます。

EXPLAIN PLAN文はアプリケーション内の各SQL文に対する実行計画を示します。実行計画には、SQL文の実行に必要なデータベース処理が記述されています。実行計画を使用すると、非効率的なSQL文を特定できます。

関連項目

- パフォーマンスのためのツール

B.6 文キャッシュについて

これは、動的SQL文に依存するすべてのプリコンパイラ・アプリケーションのパフォーマンス向上に役立つ機能です。この新機能の実装により、動的文を再利用する際の解析のオーバーヘッドが削減されます。プリコンパイラ・アプリケーションのユーザーは、新しいコマンドライン・オプション(文のキャッシュ・サイズ用)を使用することで、パフォーマンスの向上を実現できます。このオプションにより、動的文の文のキャッシュが有効になります。この新しいオプションを有効にすると、セッション作成時に文のキャッシュが作成されます。キャッシングは動的文に対してのみ適用され、静的文用のカーソル・キャッシュとこの機能は共存します。

B.7 索引の使用について

索引は、ROWIDを使用して、表の列の各値をその値が格納されている行に関連付けます。索引はCREATE INDEX文で作成します。詳細は、CREATE INDEXを参照してください。

表の15%未満の行しか戻さない問合せでは、索引を使用することによりパフォーマンスが向上します。表の15%以上の行を戻す問合せは、全体スキャンによる方法、つまり、すべての行を順番に読み込む方法の方が速く処理されます。

WHERE句内で索引の付いた列を指定する問合せは、その索引を使用します。索引を付ける列を選択するためのガイドラインは、*Oracle Database*アドバンスト・アプリケーション開発者ガイドを参照してください。

B.8 行レベル・ロックの利用について

デフォルトでは、データは表レベルではなく行レベルでロックされます。行レベルでロックすると、複数のユーザーが同一の表の別の行に同時にアクセスできます。その結果、パフォーマンスが大幅に向上します。

表レベルでのロックも指定できますが、これはトランザクション処理オプションの効果を低下させます。

オンラインのトランザクション処理を実行するアプリケーションには、行レベル・ロックが最も有効です。アプリケーションで表レベル・ロックを指定している場合は、行レベル・ロックを利用できるように変更してください。通常、明示的な表レベル・ロックは使用しないようにします。

関連項目

- LOCK TABLEの使用

B.9 不要な解析の排除について

不要な解析をなくすには、カーソルを正しく操作することと、次に示すカーソル管理オプションを選択して使用する必要があります。

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

これらのオプションは、暗黙カーソルおよび明示カーソル、カーソル・キャッシュおよびプライベートSQL領域に影響します。

B.9.1 明示カーソルの処理について

カーソルには、暗黙カーソルと明示カーソルの2種類があることを思い出してください。カーソルは、データ定義文およびDML文のすべてについて暗黙的に宣言されます。ただし、複数行を戻す問合せでは、カーソルを明示的に宣言する(またはホスト配列を使用する)必要があります。DECLARE CURSOR文を使用すると、明示カーソルを宣言できます。明示カーソルのオープンおよびクローズを処理する方法はパフォーマンスに影響します。

アクティブ・セットを再評価する必要がある場合は、そのカーソルを再OPENするのみで済みます。OPENでは任意の新しいホスト変数値が使用されます。カーソルを再OPENする前にCLOSEしなければ、処理時間を節約できます。

> 注意:
>
> すでにオープンしているカーソルを再 OPEN すると、パフォーマンスの最適化が簡単になります。ただし、これは ANSI 拡張機能です。したがって、MODE=ANSI を指定している場合には、カーソルを再 OPEN する前に CLOSE する必要があります。

カーソルのOPENによって取得したリソース(メモリーおよびロック)を解放するときにのみ、そのカーソルをCLOSEします。たとえば、プログラムでは終了前にすべてのカーソルをCLOSEする必要があります。

B.9.1.1 カーソルの制御

一般に、明示的に宣言されたカーソルの制御には、次の3つの方法があります。

- DECLARE、OPENおよびCLOSEを使用します。

- PREPARE、DECLARE、OPENおよびCLOSEを使用します。
- MODE=ANSIの場合は、COMMITによってカーソルをクローズします。

最初の方法を使用する場合は、不要な解析に注意する必要があります。カーソルをCLOSEしたか、まだOPENしていないために、解析された文を使用できないときにかぎり、OPENで解析を実行します。プログラムはカーソルをDECLAREし、ホスト変数の値が変わるたびにこれを再OPENし、このSQL文が必要なくなったときにのみこれをCLOSEする必要があります。

2番目の方法(動的SQL方法3および方法4用の方法)を使用する場合は、PREPAREで解析が実行され、解析された文はCLOSEを実行するまで使用できます。プログラムで次のようにする必要があります。

- SQL文をPREPAREします。
- カーソルをDECLAREします。
- ホスト変数の値が変更されるたびに、カーソルを再OPENします。
- SQL文を再びPREPAREします。
- SQL文が変更された場合は、カーソルを再OPENします。
- SQL文が不要になった場合にのみカーソルをCLOSEします。

OPEN文およびCLOSE文をループの中に配置するのはできるだけ避けてください。SQL文の不要な再解析の原因になります。次の例では、OPEN文とCLOSE文がどちらも外側の*while*ループの中にあります。MODE=ANSIの場合は、CLOSE文は例に示す位置に配置する必要があります。ANSIでは、カーソルを再OPENする前にCLOSEする必要があります。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT ename, sal from emp where sal >  :salary and
                                      sal <= :salary + 1000;
salary = 0;
while (salary < 5000)
{
     EXEC SQL OPEN emp_cursor;
     while (SQLCODE==0)
     {
         EXEC SQL FETCH emp_cursor INTO ....
         ...
     }
     salary += 1000;
     EXEC SQL CLOSE emp_cursor;
}
```

一方、MODE=ORACLEのときは、カーソルを再OPENせずにCLOSE文を実行できます。CLOSE文を外側の*while*ループの外に配置すると、OPEN文が繰り返されるたびに再解析されるのを回避できます。

```
...
while (salary < 5000)
{
     EXEC SQL OPEN emp_cursor;
     while (sqlca.sqlcode==0)
     {
         EXEC SQL FETCH emp_cursor INTO ....
         ...
     }
     salary += 1000;
}
EXEC SQL CLOSE emp_cursor;
```

B.9.2 カーソル管理オプションの使用について

SQL文の解析は、構成を変更しないかぎり一度のみで十分です。たとえば、選択リストまたはWHERE句に列を追加して問合せの構成を変更するとします。HOLD_CURSORおよびRELEASE_CURSOR、MAXOPENCURSORSオプションによって、サーバーにおけるSQL文の解析および再解析の管理方法を制御できます。明示カーソルを宣言すると、解析を最大限に制御できます。

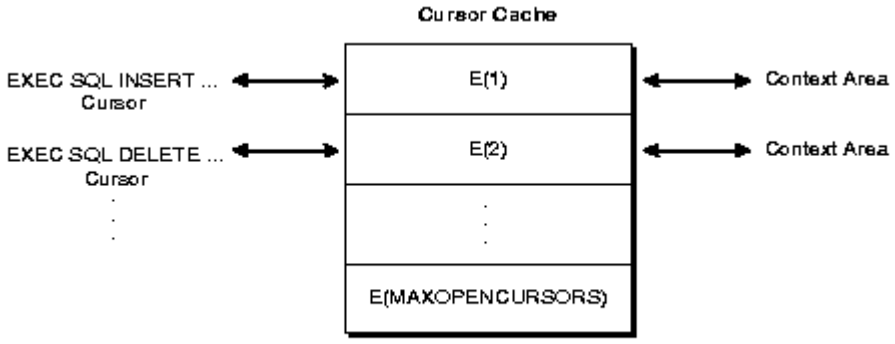
B.9.2.1 SQL領域とカーソル・キャッシュ

DML文を実行すると、その文に対応しているカーソルがPro*C/C++カーソル・キャッシュ内のエントリにリンクされます。カーソル・キャッシュとはカーソル管理のために使用されて連続的に更新されるメモリー領域です。カーソル・キャッシュ・エントリは、次々に1つのプライベートSQL領域にリンクされます。

プライベートSQL領域とは、実行時に動的に作成される作業領域で、ホスト変数のアドレスおよびその文の処理に必要なその他の情報が保存されます。明示カーソルを使用すると、SQL文に名前を付け、プライベートSQL領域に保存されている情報にアクセスし、この情報の処理をある程度制御できます。

図B-2は、プログラムでINSERTおよびDELETEを実行した後のカーソル・キャッシュを表しています。

図B-2 カーソル・キャッシュでリンクされたカーソル



B.9.2.2 リソースの使用

ユーザー・セッションごとのオープン・カーソルの最大数は、初期化パラメータOPEN_CURSORSによって設定します。

MAXOPENCURSORSは、カーソル・キャッシュの初期サイズを指定します。新しいカーソルが必要で、しかも空きのキャッシュ・エントリがない場合、サーバーではエントリの再利用が試行されます。再利用の可能性はHOLD_CURSORとRELEASE_CURSORの値によって決まり、また明示カーソルの場合には、カーソル自体の状態によって決まります。

MAXOPENCURSORSの値が実際に必要なキャッシュ・エントリの数より小さい場合、サーバーでは再利用可能とマークされている最初のキャッシュ・エントリが使用されます。たとえば、INSERT文のキャッシュ・エントリ*E*(1)が再利用可能とマークされていて、キャッシュ・エントリの数がすでにMAXOPENCURSORSに達しているとします。プログラムが新しい文を実行する場合、キャッシュ・エントリ*E*(1)とそのプライベートSQL領域は新しい文に再度割り当てられることがあります。INSERT文を再実行するために、サーバーではそれを再度解析しなおして、別のキャッシュ・エントリを再度割り当てる必要があります。

再利用できるキャッシュ・エントリが見つからない場合、サーバーでは追加のキャッシュ・エントリが割り当てられます。たとえば、MAXOPENCURSORS=8で、8エントリすべてがアクティブな場合、9番目のエントリが作成されます。サーバーでは、空きメモリーがなくなるかOPEN_CURSORSで設定された上限に達するまで、必要に応じてキャッシュ・エントリの割当てが続行されます。この動的割当ては、処理オーバーヘッドを増大させます。

したがって、MAXOPENCURSORSの値を小さく設定すると、メモリーの節約にはなりますが、新しいキャッシュ・エントリの動的割当ておよび割当て解除に資源を消耗する場合があります。MAXOPENCURSORSの値を大きく設定すると、実行は確実に速くなりますが、より大きなメモリーを使用することになります。

B.9.2.3 実行回数の少ない場合

実行回数の少ないSQL文とそのプライベートSQL領域間のリンクは、一時的なものにした方がよい場合もあります。

HOLD_CURSOR=NO(デフォルト値)と指定した場合は、サーバーによりそのSQL文が実行され、カーソルがクローズされた後に、プリコンパイラによりこのカーソルとカーソル・キャッシュ間のリンクが再利用可能としてマークされます。このリンクは、それが示すカーソル・キャッシュ・エントリが別のSQL文に必要になると、すぐに再利用されます。これにより、プライベートSQL領域に割り当てられたメモリーが解放され、解析ロックが解除されます。ただし、PREPAREしたカーソルは実行状態のままにする必要があるため、HOLD_CURSOR=NOと指定した場合でもそのリンクは維持されます。

RELEASE_CURSOR=YESと指定した場合は、サーバーによりそのSQL文が実行され、カーソルがクローズされた後に、プライベートSQL領域が自動的に解放され、解析した文は失われます。メモリーの節約のためにMAXOPENCURSORSを低い値に設定しているような場合には、この指定が必要です。

DML文がデータ定義文より前にあり、どちらの文も同じ表を参照する場合には、DML文にRELEASE_CURSOR=YESを指定してください。これにより、データ操作文で取得される解析ロックと、データ定義文で要求される排他ロックとの間の競合が回避できます。

RELEASE_CURSOR=YESを指定した場合は、プライベートSQL領域とキャッシュ・エントリ間のリンクはただちに削除され、このプライベートSQL領域は解放されます。HOLD_CURSOR=YESを指定している場合でも、RELEASE_CURSOR=YESによりHOLD_CURSOR=YESが上書きされるため、サーバーではSQL文を実行する前に、プライベートSQL領域にメモリーを再度割り当てて、このSQL文を再解析する必要があります。

ただし、RELEASE_CURSOR=YESを指定した場合、サーバーではSQL文とPL/SQLブロックの解析済の表現が共有*SQL*キャッシュに保存されるため、それ以上の再解析処理は不要になることがあります。カーソルをクローズしても、解析された表現はキャッシュの内容が書き換えられるまで効力を持ちます。

B.9.2.4 実行回数の多い場合

プライベートSQL領域には文の実行に必要なすべての情報が格納されているため、頻繁に実行されるSQL文とそのプライベートSQL領域のリンクを維持する必要があります。この情報へのアクセスを上手に管理すると、後続の文の実行速度をさらに向上させることができます。

HOLD_CURSOR=YESを指定した場合は、サーバーによりSQL文が実行された後に、カーソルとカーソル・キャッシュのリンクが維持されます。したがって、解析された文および割り当てられたメモリーが、利用可能なまま維持されます。これは、不必要な再解析を避けるためにアクティブにしておくSQL文で役に立ちます。

RELEASE_CURSOR=NO(デフォルト値)の場合、サーバーでSQL文を実行した後にキャッシュ・エントリとプライベートSQL領域間のリンクが維持され、オープンしたカーソルの数がMAXOPENCURSORSの値を超えないかぎり、そのリンクは再利用されません。これは、解析した文および割り当てたメモリーが使用可能な状態のままのため、頻繁に実行するSQL文の場合に有効です。

注意:

Oracle8*i* の以前のバージョンでは、SQL 文の実行後に RELEASE_CURSOR=NO および HOLD_CURSOR=YES となっていると、解析後の表現を継続して使用できました。ただし、Oracle の後続のバージョンでは、RELEASE_CURSOR=NO および HOLD_CURSOR=YES が指定されている場合、解析後の表現を使用できるのは共有 SQL キャッシュの内容が期限切れになるまでの間です。通常、これは問題にはなりませんが、その SQL 文が再解析される前に参照オブジェクトの定義が変わると、結果が予期せぬものになる場合があります。

B.9.2.5 埋込みPL/SQLの考慮事項

カーソルを管理するために、埋込みPL/SQLブロックはSQL文と同様に扱われます。埋込みPL/SQLブロックが実行されると、親カーソルがPL/SQLブロック全体に対応付けられ、埋込みPL/SQLブロック用にキャッシュ・エントリとPGAのプライベートSQL領域の間にリンクが作成されます。埋込みブロック内の各SQL文にも、PGAのプライベートSQL領域が必要なことに注意してください。これらのSQL文は、PL/SQLが管理する子カーソルを使用します。子カーソルの性質は、対応付けられた親カーソルによって決まります。つまり、子カーソルが使用するプライベートSQL領域は、親カーソルのプライベートSQL領域が解放された後解放されます。

B.9.2.6 パラメータの相互作用

図B-1に、HOLD_CURSORとRELEASE_CURSORの相互関係を示します。HOLD_CURSOR=NOを指定すると、RELEASE_CURSOR=NOはオーバーライドされ、RELEASE_CURSOR=YESを指定すると、HOLD_CURSOR=YESがオーバーライドされることに注意してください。

表B-1 HOLD_CURSORとRELEASE_CURSORの相互関係

HOLD_CURSOR	RELEASE_CURSOR	リンク
NO	NO	再利用可能とマークされます。
YES	NO	維持されます。
NO	YES	ただちに削除されます。
YES	YES	ただちに削除されます。

B.10 不要な再解析の回避について

不要な再解析を回避するには、ループ内のSQL文の実行フェーズ中に発生するエラーを排除する必要があります。ループ内の埋込みSQL文が実行されるときには、そのSQL文が一度のみ解析されます。ただし、実行結果がエラーになったSQL文は、通常は再解析されます。この場合、次のエラーを除き、発生したエラーすべてについて再解析が発生します。

- ORA-1403 (見つかりません)
- ORA-1405 (切捨て)
- ORA-1406 (NULL値)

他のすべてのエラーを排除すれば、不要な再解析を回避できます。

B.11 接続プーリングの使用方法について

この項では、接続プーリングを使用したパフォーマンス・チューニングを説明します。アプリケーションがマルチスレッドで、同一データベースに対して同時操作を実行している場合、接続プーリング機能を使用するとパフォーマンスを上げられます。接続プーリングに使用されるパラメータに適切な値を選択してアプリケーションのパフォーマンスをチューニングすると、既存のアプリケーション・パフォーマンスに比べて、パフォーマンスを最高3倍まで上げられます。

関連項目

- マルチスレッド・アプリケーション
- 接続プーリング
- Oracle Call Interfaceプログラマーズ・ガイド

B.12 Traffic DirectorモードのOracle Connection Managerの使用について

Traffic DirectorモードのOracle Connection Managerは、サポートされているデータベース・クライアントとデータベース・インスタンスの間に配置されるプロキシです。

Oracle Database 11*g*リリース2 (11.2)以上からサポートされるクライアントは、Traffic DirectorモードのOracle Connection Managerに接続できます。Traffic DirectorモードのOracle Connection Managerは、計画済および計画外のデータベース・サーバー停止に対する改善された高可用性(HA)、接続多重化のサポートおよびロード・バランシングを提供します。Traffic DirectorモードのOracle Connection Managerのサポートは、次の項で詳しく説明しています

- 操作のモード
- 主な機能

操作のモード

Traffic DirectorモードのOracle Connection Managerは、次のモードの操作をサポートします。

- プール接続モードでは、Traffic DirectorモードのOracle Connection Managerは、次のデータベース・クライアント・リリースを使用するすべてのアプリケーションをサポートします。
  - OCI、OCCIおよびOpen Source Driver (Oracle Database 11*g*リリース2 (11.2.0.4)以上)
  - JDBC (Oracle Database 12*c*リリース1 (12.1)以上)
  - ODP.NET (Oracle Database 12*c*リリース2 (12.2)以上)

  さらに、アプリケーションでDRCPを使用する必要があります。つまり、アプリケーションでは、接続文字列で(または`tnsnames.ora`別名で)DRCPを有効にする必要があります。
- 非プール接続(または専用)モードでは、Traffic DirectorモードのOracle Connection Managerは、データベース・クライアント・リリースOracle Database 11*g*リリース2 (11.2.0.4)以上を使用するすべてのアプリケーションをサポートします。このモードでは、接続多重化など一部の機能は使用できません。

主な機能

Traffic DirectorモードのOracle Connection Managerは次に対するサポートを提供しています。

- 次を含む透過的パフォーマンス拡張機能および接続多重化。
  - 文のキャッシング、行のプリフェッチ、および結果セットのキャッシングは、すべてのモードの操作に対して自動で有効化されます。
  - プロキシ常駐接続プール(PRCP)を使用するデータベース・セッション多重化(プール・モードのみ)。ここでPRCPは、データベース常駐接続プーリング(DRCP)のプロキシ・モードです。アプリケーションは、Traffic DirectorモードのOracle Connection Managerとデータベースの間で、透過的な接続時ロード・バランシングおよび実行時ロード・バランシングを取得します。
  - 複数のTraffic DirectorモードのOracle Connection Managerインスタンスがある場合、アプリケーションは、クライアント側の接続時ロード・バランシングまたはロード・バランサ(BIG-IP、NGINXなど)を通じてスケーラビリティが増加します

- アプリケーション停止時間がゼロ
  - 計画済データベース・メンテナンスまたはプラガブル・データベース(PDB)再配置
    - プール・モード

      Traffic DirectorモードのOracle Connection Managerは、計画済停止に対するOracle Notification Service (ONS)イベントに応答し、作業をリダイレクトします。要求が完了すると、Traffic DirectorモードのOracle Connection Managerのプールから接続が排出されます。サービスの再配置は、Oracle Database 11*g*リリース2 (11.2.0.4)以上に対してサポートされています。

      PDB再配置については、Traffic DirectorモードのOracle Connection Managerは、ONSが構成されていない場合でも、PDBが再配置されるときにインバンド通知に応答します(Oracle Databaseリリース18c、バージョン18.1以上のサーバーに対してのみ)

    - 非プールまたは専用モード

      クライアントからの要求境界情報がない場合、Traffic DirectorモードのOracle Connection Managerは、多くのアプリケーションに対して計画済停止をサポートします(要求/トランザクション境界を越えて保持する必要があるものが単純なセッション状態およびカーソル状態である場合のみ)。次の機能がサポートされます。
      - トランザクション境界でサービス/PDBを停止するか、Oracle Databaseリリース18cの連続アプリケーション可用性を利用して、要求境界でサービスを停止します
      - Traffic DirectorモードのOracle Connection Managerは、透過的アプリケーション・フェイルオーバー(TAF)のフェイルオーバー・リストアを利用し、再接続して単純な状態をリストアします。
  - ほとんどが読取りのワークロードに対する計画外データベース停止
- 単一点障害を回避するTraffic DirectorモードのOracle Connection Managerの高可用性。これは次によってサポートされます。
  - 接続文字列でロード・バランサまたはクライアント側のロード・バランシング/フェイルオーバーを使用する、複数インスタンスのTraffic DirectorモードのOracle Connection Manager
  - Traffic DirectorモードのOracle Connection Managerインスタンスのローリング・アップグレード
  - クライアントからTraffic DirectorモードのOracle Connection Managerへの既存の接続の、計画済停止に対する正常なクローズ
  - Oracle Databaseリリース18c以上のクライアントに対するインバンド通知
  - 古いクライアントに対しては、通知は現在の要求の応答とともに送信されます
- セキュリティおよび分離のために、Traffic DirectorモードのOracle Connection Managerは次の機能を提供します。
  - Transmission Control Protocol/Transmission Control Protocol Secure (TCP/TCPS)およびプロトコル変換をサポートするデータベース・プロキシ
  - IPアドレス、サービス名、およびSecure Socket Layer/Transport Layer Security (SSL/TLS)ウォレットに基づくファイアウォール
  - マルチテナント環境でのテナント分離
  - サービス妨害およびファジング攻撃からの保護

- オンプレミスのOracle DatabaseとOracle Cloudの間のデータベース・トラフィックの安全なトンネリング

関連項目:

- Traffic DirectorモードのOracle Connection Managerを設定するためのcman.ora構成ファイルの構成の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- Traffic DirectorモードのOracle Connection Managerプロキシ認証の構成の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- 計画外停止イベントに対するTraffic DirectorモードのOracle Connection Managerの構成の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- 計画済停止イベントに対するTraffic DirectorモードのOracle Connection Managerの構成の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- Traffic DirectorモードのOracle Connection Managerで使用するためのプロキシ常駐接続プールの構成の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- Traffic DirectorモードのOracle Connection Managerのすべてのドライバでサポートされていない機能の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- Oracle CMAN構成ファイルの概要は、Oracle Database Net Servicesリファレンスを参照してください

C 構文およびセマンティックのチェック

埋込みSQL文およびPL/SQLブロックの構文および意味を検査することで、Pro*C/C++プリコンパイラはコーディングの誤りをすみやかに発見し修正できるように支援します。この付録では、プリコンパイラ・オプションSQLCHECKを使用して検査の種類および範囲を制御する方法を説明します。この付録の内容は次のとおりです。

- 構文検査と意味検査

  検査の種類および範囲の制御について
- SQLCHECK=SEMANTICSの指定について
- SQLCHECK=SYNTAXの指定について
- SQLCHECKオプションの入力について

C.1 構文検査と意味検査

構文規則は、言語要素を並べて正しい文を作成する基準を示します。つまり、構文検査はキーワード、オブジェクト名、演算子、デリミタなどがSQL文に正しく配置されていることを検証します。たとえば、次の埋込みSQL文には構文上のエラーがあります。

```
EXEC SQL DELETE FROM EMP WHER DEPTNO = 20;
    -- misspelled keyword WHERE
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
    -- missing parentheses around column names COMM and SAL
```

意味上の規則は、有効な外部参照を行う方法を示しています。したがって、セマンティック・チェックでは、データベース・オブジェクトおよびホスト変数への参照が有効であることと、ホスト変数のデータ型が正しいことを検証します。たとえば、次の埋込みSQL文には意味上のエラーがあります。

```
EXEC SQL DELETE FROM empp WHERE deptno = 20;
    -- nonexistent table, EMPP
EXEC SQL SELECT * FROM emp WHERE ename = :emp_name;
    -- undeclared host variable, emp_name
```

SQL構文と意味に関する規則は、*Oracle Database SQL言語リファレンス*に定義されています。

C.2 チェックの種類および範囲の制御について

コマンドラインでプリコンパイラ・オプションのSQLCHECKを指定すると、検査の種類および範囲を制御します。SQLCHECKでは、チェックの種類に構文、セマンティック、あるいはその両方のいずれかを指定できます。検査の範囲には、次の要素を含めることができます。

- データ定義文(CREATEおよびGRANTなど)
- DML文(SELECTおよびINSERTなど)
- PL/SQLブロック

ただし、動的SQL文は実行時まで完全に定義されないため、SQLCHECKでは動的SQL文を検査できません。

SQLCHECKについて次の値を指定できます。

- SEMANTICSまたはFULL
- SYNTAX

デフォルト値はSYNTAXです。

SQLCHECKを使用しても、データ制御、カーソル制御、動的SQL文に対する通常の構文検査には影響しません。

C.3 SQLCHECK=SEMANTICSの指定について

SQLCHECK=SEMANTICSの場合、プリコンパイラは次の内容について構文および意味の検査を行います。

- DML文(INSERT、UPDATEなど)
- PL/SQLブロック
- ホスト変数のデータ型

次の構文も同様です。

- データ定義文(CREATE、ALTERなど)

ただし、AT *db_name*句を使用するDML文については、意味検査のみが行われます。

SQLCHECK=SEMANTICSの場合、プリコンパイラは、埋め込まれたDECLARE TABLE文を使用して、意味検査に必要な情報を取得します。ただし、コマンドラインでUSERIDオプションを指定した場合は、Oracleサーバーに接続し、データ・ディクショナリにアクセスしてこの情報を取得します。データ操作文またはPL/SQLブロックで参照される表がすべてDECLARE TABLE文で定義されている場合は、Oracleサーバーに接続する必要はありません。

Oracleサーバーに接続したときに、データ・ディクショナリに一部の必要な情報が見つからない場合は、DECLARE TABLE文を使用して、欠落している情報を補充する必要があります。DECLARE TABLE文とデータ・ディクショナリの定義が矛盾する場合は、DECLARE TABLEの定義が使用されます。

ホスト・プログラム内にPL/SQLブロックを埋め込むときは、必ずSQLCHECK=SEMANTICSを指定してください。

データ操作文を検査するときに、プリコンパイラは*Oracle Database SQL言語リファレンス*に記述されている構文規則を使用しますが、意味検査にはより厳密な規則を使用します。特に、データ型のチェックは厳密に行います。その結果、SQLCHECK=SEMANTICSのときは、Oracleの以前のバージョン用に作成した既存のアプリケーションを正常にプリコンパイルできない場合があります。

新規のプログラムをプリコンパイルする場合、またはデータ型のチェックを厳密に行う場合は、SQLCHECK=SEMANTICSと指定してください。

C.3.1 意味検査の使用許可について

SQLCHECK=SEMANTICSを指定すると、プリコンパイラは意味検査に必要な情報を、次の方法のどれかで入手できます。

- Oracleサーバーに接続してデータ・ディクショナリにアクセスします。
- 埋め込まれたDECLARE TABLE文とDECLARE TYPE文を使用します。

C.3.1.1 Oracleサーバーへの接続について

意味検査を行うには、表、データ型、ホスト・プログラムで参照されるビューをメンテナンスするOracleデータベースにプリコンパイラを接続します。

Oracleサーバーに接続した後、プリコンパイラはデータ・ディクショナリにアクセスして必要な情報を検索します。データ・ディクショナリには、表および列の名前、表および列の制約、列の長さ、列のデータ型などが格納されています。

プログラムが、まだ作成されていない表を参照しているなどの理由により、必要な情報の一部がデータ・ディクショナリ内で見つからない場合は、DECLARE TABLE文(この付録で後述されています)を使用して足りない情報を指定する必要があります。

Oracleサーバーに接続するには、次の構文を使用してコマンドラインでUSERIDオプションを指定します。

```
USERID=username/password
```

*username*および*password*は有効なOracleユーザーIDを構成します。パスワードを省略すると、パスワードの入力が求められます。

仮に、ユーザー名とパスワードのかわりに、次のように指定したとします。

```
USERID=/
```

プリコンパイラによりOracleサーバーへの接続が自動的に試行されます。この自動接続が成功するのは、既存のOracleユーザー名がオペレーティング・システムのIDの前に「CLUSTER$」を付けたもの、またはINIT.ORAファイル内のOS_AUTHENT_PREFIXパラメータの設定値と一致する場合のみです。たとえば、使用しているオペレーティング・システムのIDがMBLAKEであるときに、自動的に接続が成功するのはCLUSTER$MBLAKEが有効なOracleユーザー名である場合のみです。

USERIDオプションを省略した場合、プリコンパイラは埋込みDECLARE TABLE文から必要な情報を取得する必要があります。

データベースが使用不能であるなどの理由により、Oracleサーバーへの接続を試みて失敗した場合は、エラー・メッセージが発行され、プログラムはプリコンパイルされません。

C.3.1.2 DECLARE TABLEの使用方法について

プリコンパイラはOracleサーバーに接続せずに意味検査を実行できます。この検査を行うには、プリコンパイラは表およびビューに関する情報を埋込みDECLARE TABLE文から取得する必要があります。つまり、DML文またはPL/SQLブロック内で参照する表をすべてDECLARE TABLE文内で定義する必要があります。

DECLARE TABLE文の構文は、次のとおりです。

```
EXEC SQL DECLARE table_name TABLE
    (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

*expr*は、CREATE TABLE文で列のデフォルト値として使用できる整数です。

ユーザー定義のオブジェクト・データ型の場合、サイズは使用されないためオプションです。

DECLARE TABLEを使用して既存のデータベースの表を定義した場合、プリコンパイラはその定義に従います。このときデータ・ディクショナリの定義は無視されます。

C.3.1.3 DECLARE TYPEの使用について

同様に、TYPEの場合は、次の構文によるDECLARE TYPE文があります。

```
EXEC SQL DECLARE type TYPE
[AS OBJECT (col_name col_datatype, ...)] |
[AS VARRAY(size) OF element_type ]|
[AS TABLE OF object_type ] ;
```

この文を使用すると、プリコンパイル時にSQLCHECK=SEMANTICSと指定した場合に、ユーザー定義型のより適切なタイプ・チェックを実行できます。SQLCHECK=SYNTAXと指定すると、DECLARE TYPE文はドキュメントとしてのみ動作し、コメント・アウトされて無視されます。

C.4 SQLCHECK=SYNTAXの指定について

SQLCHECK=SYNTAXを指定すると、プリコンパイラでSQL文の構文がチェックされます。構文については、*Oracle Database SQL言語リファレンス*を参照してください。

- DML文
- ホスト変数表現

意味検査が実行されないため、次の制限事項が適用されます。

- Oracleサーバーへの接続は行われず、USERIDは無効なオプションとなります。USERIDを指定すると、警告メッセージが発行されます。
- DECLARE TABLE文とDECLARE TYPE文は無視され、ドキュメントとしてのみ機能します。
- PL/SQLブロックの埋込みはできません。プリコンパイラがPL/SQLブロックを検出すると、エラー・メッセージが発行されます。

DML文をチェックする場合、プリコンパイラはOracleの構文規則を使用します。これらの規則には下位互換性があるため、プリコンパイルしたプログラムを移行するときにはSQLCHECK=SYNTAXを指定してください。

C.5 SQLCHECKオプションの入力について

SQLCHECKオプションは、インラインまたはコマンドラインで入力できます。ただし、インラインで指定するチェックのレベルを、コマンドラインで指定する(またはデフォルトによって受け入れる)レベルよりも高くすることはできません。たとえば、SQLCHECK=SYNTAXをコマンドラインで指定した場合、インラインではSQLCHECK=SEMANTICSを指定できません。

D システム固有の参照

この付録では、このマニュアルで参照しているシステム固有の情報をすべてまとめて記載します。

この付録の項目は、次のとおりです。

- システム固有の情報

D.1 システム固有の情報

システム固有の情報は、使用しているOracleシステムのマニュアルで説明しています。

D.1.1 標準ヘッダー・ファイルの位置

標準Pro*C/C++ヘッダー・ファイル(sqlca.h、oraca.hおよびsqlda.h)の位置は、システムによって異なります。他のシステムについては、Oracleシステム固有のマニュアルを参照してください。

D.1.2 Cコンパイラ用インクルード・ファイルの位置指定について

Pro*C/C++コマンドライン・オプションINCLUDE=を使用して、インクルードされる非標準ファイルの位置を指定する場合は、Cコンパイラにも同じ位置を指定する必要があります。これを実行する方法はシステム固有です。インクルード・ファイルを参照してください。

D.1.3 ANSI Cサポート

CODE=オプションを使用すると、Pro*C/C++で生成されるC言語のコードと使用中のシステムのCコンパイラとの互換性を確保できます。関数プロトタイプを参照してください。

D.1.4 構造体コンポーネントの位置合せ

通常は、システムのハードウェアによって、Cコンパイラが構造体のコンポーネントの位置合せを行う方法が異なります。sqlvcp()関数を使用して、VARCHAR構造体の.arrコンポーネントに追加されるパディングを判別してください。VARCHAR配列コンポーネントの長さを調べる方法を参照してください。

D.1.5 整数とROWIDのサイズ

整数データ型のバイト数とROWIDデータ型のバイナリ外部サイズは、システムによって異なります。INTEGERおよびROWIDを参照してください。

D.1.6 バイト順序

1つのワード中のバイト順序は、プラットフォームによって異なります。詳細は、UNSIGNEDを参照してください。

D.1.7 Oracleサーバーへの接続について

Oracle Netドライバを使用してOracleサーバーに接続するには、システム固有のネットワーク・プロトコルが必要です。詳細は、OCIリリース8へのインタフェースを参照してください。

D.1.8 XAライブラリでのリンクについて

XAライブラリでのリンクは、システムによって異なります。詳細は、リンクおよびOracleのインストレーション・ガイドまたはユーザー

ズ・ガイドを参照してください。

D.1.9 Pro*C/C++実行可能ファイルの位置

Pro*C/C++プリコンパイラの位置は、システム固有です。詳細は、プリコンパイラのコマンドおよびインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

D.1.10 システム構成ファイル

各プリコンパイラのインストールには、システム構成ファイルがあります。このファイルはプリコンパイラに付属しているものではないため、システム管理者が作成する必要があります。Pro*C/C++がシステム構成ファイルを検索する位置(ディレクトリ・パス)は、システムによって異なります。詳細は、プリコンパイル中の状況を参照してください。

D.1.11 INCLUDEオプションの構文

INCLUDEコマンドライン・オプションの値に対応する構文は、システム固有です。INCLUDEを参照してください。

D.1.12 コンパイルとリンクについて

Pro*C/C++出力をコンパイルおよびリンクして実行可能なアプリケーションを得る方法は、常にシステムによって異なります。詳細は、コンパイルおよびリンクおよびこの後の各項を参照してください。

D.1.13 ユーザー・イグジット

Oracle Formsのユーザー・イグジットのコンパイルおよびリンクは、システム固有です。ユーザー・イグジットを参照してください。

E 埋込みSQL文およびディレクティブ

この付録では、SQL標準の埋込み文とディレクティブ、およびOracleの埋込みSQLの拡張機能について説明します。

> 注意:
>
> この付録では、非埋込み SQL と構文が異なる文のみを説明します。非埋込み SQL 文の詳細は、SQL 文の種類を参照してください。

この付録の内容は次のとおりです。

- プリコンパイラ・ディレクティブおよび埋込みSQL文の概要
- 文の説明
- 構文図の読み方
- ALLOCATE (実行可能埋込みSQL拡張機能)
- ALLOCATE DESCRIPTOR (実行可能埋込みSQL)
- CACHE FREE ALL(実行可能埋込みSQL拡張機能)
- CALL (実行可能埋込みSQL)
- CLOSE (実行可能埋込みSQL)
- COLLECTION APPEND(実行可能埋込みSQL拡張機能)
- COLLECTION DESCRIBE(実行可能埋込みSQL拡張機能)
- COLLECTION GET(実行可能埋込みSQL拡張機能)
- COLLECTION RESET(実行可能埋込みSQL拡張機能)
- COLLECTION SET(実行可能埋込みSQL拡張機能)
- COLLECTION TRIM(実行可能埋込みSQL拡張機能)
- COMMIT (実行可能埋込みSQL)
- CONNECT (実行可能埋込みSQL拡張機能)
- CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)
- CONTEXT FREE (実行可能埋込みSQL拡張機能)
- CONTEXT OBJECT OPTION GET(実行可能埋込みSQL拡張機能)
- CONTEXT OBJECT OPTION SET(実行可能埋込みSQL拡張機能)
- CONTEXT USE (Oracle埋込みSQLディレクティブ)
- DEALLOCATE DESCRIPTOR (埋込みSQL文)
- DECLARE CURSOR (埋込みSQLディレクティブ)
- DECLARE DATABASE (Oracle埋込みSQLディレクティブ)

- DECLARE STATEMENT (埋込みSQLディレクティブ)
- DECLARE TABLE (Oracle埋込みSQLディレクティブ)
- DECLARE TYPE(Oracle埋込みSQLディレクティブ)
- DELETE (実行可能埋込みSQL)
- DESCRIBE(実行可能埋込みSQL拡張機能)
- DESCRIBE DESCRIPTOR (実行可能埋込みSQL)
- ENABLE THREADS (実行可能埋込みSQL拡張機能)
- EXECUTE ...END-EXEC (実行可能埋込みSQL拡張機能)
- EXECUTE (実行可能埋込みSQL)
- EXECUTE DESCRIPTOR(実行可能埋込みSQL)
- EXECUTE IMMEDIATE (実行可能埋込みSQL)
- FETCH (実行可能埋込みSQL)
- FETCH DESCRIPTOR (実行可能埋込みSQL)
- FREE (実行可能埋込みSQL拡張機能)
- GET DESCRIPTOR (実行可能埋込みSQL)
- INSERT (実行可能埋込みSQL)
- LOB APPEND (実行可能埋込みSQL拡張機能)
- LOB ASSIGN (実行可能埋込みSQL拡張機能)
- LOB CLOSE (実行可能埋込みSQL拡張機能)
- LOB COPY (実行可能埋込みSQL拡張機能)
- LOB CREATE TEMPORARY (実行可能埋込みSQL拡張機能)
- LOB DESCRIBE (実行可能埋込みSQL拡張機能)
- LOB DISABLE BUFFERING (実行可能埋込みSQL拡張機能)
- LOB ENABLE BUFFERING (実行可能埋込みSQL拡張機能)
- LOB ERASE (実行可能埋込みSQL拡張機能)
- LOB FILE CLOSE ALL (実行可能埋込みSQL拡張機能)
- LOB FILE SET (実行可能埋込みSQL拡張機能)
- LOB FLUSH BUFFER (実行可能埋込みSQL拡張機能)
- LOB FREE TEMPORARY (実行可能埋込みSQL拡張機能)
- LOB LOAD (実行可能埋込みSQL拡張機能)
- LOB OPEN (実行可能埋込みSQL拡張機能)
- LOB READ (実行可能埋込みSQL拡張機能)
- LOB TRIM (実行可能埋込みSQL拡張機能)

- LOB WRITE (実行可能埋込みSQL拡張機能)
- OBJECT CREATE(実行可能埋込みSQL拡張機能)
- OBJECT DELETE(実行可能埋込みSQL拡張機能)
- OBJECT DEREF(実行可能埋込みSQL拡張機能)
- OBJECT FLUSH(実行可能埋込みSQL拡張機能)
- OBJECT GET(実行可能埋込みSQL拡張機能)
- OBJECT RELEASE(実行可能埋込みSQL拡張機能)
- OBJECT SET(実行可能埋込みSQL拡張機能)
- OBJECT UPDATE(実行可能埋込みSQL拡張機能)
- OPEN (実行可能埋込みSQL)
- OPEN DESCRIPTOR (実行可能埋込みSQL)
- PREPARE (実行可能埋込みSQL)
- REGISTER CONNECT(実行可能埋込みSQL拡張機能)
- ROLLBACK (実行可能埋込みSQL)
- SAVEPOINT (実行可能埋込みSQL)
- SELECT (実行可能埋込みSQL)
- SET DESCRIPTOR (実行可能埋込みSQL)
- TYPE(Oracle埋込みSQLディレクティブ)
- UPDATE (実行可能埋込みSQL)
- VAR (Oracle埋込みSQLディレクティブ)
- WHENEVER (埋込みSQLディレクティブ)

E.1 プリコンパイラのディレクティブと埋込みSQL文の概要

埋込みSQL文では、DDL、DMLおよびトランザクション制御文をPro*C/C++プログラム内に置きます。表E-1は、埋込みSQL文およびディレクティブの機能の概要です。

表E-2のソース/タイプの列は、次の形式で表記されています。

表E-1 埋込みSQL文とディレクティブの機能概要

ソース/タイプ	説明
ソース	標準 SQL (S)または Oracle の拡張機能(O)。
タイプ	実行文(E)またはディレクティブ(D)。

表E-2 プリコンパイラ・ディレクティブおよび埋込みSQL文および句

EXEC SQL文	ソース/タイプ	用途

EXEC SQL文	ソース/タイプ	用途
ALLOCATE	O/E	カーソル変数またはオブジェクト型にメモリーを割り当てます。
ALLOCATE DESCRIPTOR	S/E	ANSI 動的 SQL の記述子を割り当てます。
CACHE FREE ALL	O/E	割り当てられたオブジェクト・キャッシュ・メモリーをすべて解放します。
CALL	S/E	ストアド・プロシージャをコールします。
CLOSE	S/E	カーソルを無効にし、保持されているリソースを解放します。
COLLECTION APPEND	O/E	1 つのコレクションの要素を別のコレクションの最後に追加します。
COLLECTION DESCRIBE	O/E	コレクションについての情報を取得します。
COLLECTION GET	O/E	コレクションの要素を取得します。
COLLECTION RESET	O/E	コレクションのスライス・エンドポイントをコレクションの最初にリセットします。
COLLECTION SET	O/E	コレクションの値を更新します。
COLLECTION TRIM	O/E	コレクションの最後から要素を削除します。
COMMIT	S/E	データベースへの変更をすべて確定して、現行のトランザクションを終了します(オプションでリソースを解放し、データベースとの接続を切断します)。
CONNECT	O/E	インスタンスにログインします。
CONTEXT ALLOCATE	O/E	メモリーを SQLLIB ランタイム・コンテキストに割り当てます。
CONTEXT FREE	O/E	メモリーを SQLLIB ランタイム・コンテキストから解放します。
CONTEXT OBJECT OPTION GET	O/E	オプションの設定方法を判断します。
CONTEXT OBJECT OPTION SET	O/E	オプションを設定します。

EXEC SQL文	ソース/タイプ	用途
CONTEXT USE	O/D	後続の実行 SQL 文で使用する SQLLIB ランタイム・コンテキストを指定します。
DEALLOCATE DESCRIPTOR	S/E	記述子領域の割当てを解除し、メモリーを解放します。
DECLARE CURSOR	S/D	問合せに対応付けてカーソルを宣言します。
DECLARE DATABASE	O/D	後続の埋込み SQL 文でアクセスされるデフォルト以外のデータベースの識別子を宣言します。
DECLARE STATEMENT	S/D	SQL 文に SQL 変数名を割り当てます。
DECLARE TABLE	O/D	Pro*C/C++によって埋込み SQL 文の意味検査に使用される表構造を宣言します。
DECLARE TYPE	O/D	Pro*C/C++による埋込み SQL 文の意味検査に使用される型の構造体を宣言します。
DELETE	S/E	表またはビューの実表から行を削除します。
DESCRIBE	S/E	記述子(ホスト変数の説明を保持している構造体)を初期化します。
DESCRIBE DESCRIPTOR	S/E	ANSI SQL 文の変数についての情報を取得します。
ENABLE THREADS	O/E	複数のスレッドをサポートするプロセスを初期化します。
EXECUTE...END-EXEC	O/E	無名 PL/SQL ブロックを実行します。
EXECUTE	S/E	準備済の動的 SQL 文を実行します。
EXECUTE DESCRIPTOR	S/E	ANSI 方法 4 動的 SQL 文を実行します。
EXECUTE IMMEDIATE	S/E	ホスト変数を持たない SQL 文を準備して実行します。
FETCH	S/E	問合せで選択した行を取り出します。
FETCH DESCRIPTOR	S/E	ANSI 方法 4 動的 SQL を使用して選択された行を取得します。

EXEC SQL文	ソース/タイプ	用途
FREE	O/E	オブジェクト・キャッシュまたはカーソルに割り当てられているメモリーを解放します。
GET DESCRIPTOR	S/E	ANSI SQL の記述子領域の情報をホスト変数に移動します。
INSERT	S/E	表またはビューの実表に行を追加します。
LOB APPEND	O/E	LOB を別の LOB の最後に追加します。
LOB ASSIGN	O/E	LOB または BFILE ロケータを別のロケータに割り当てます。
LOB CLOSE	O/E	オープンされている LOB または BFILE をクローズします。
LOB COPY	O/E	LOB 値の全部または一部を別の LOB にコピーします。
LOB CREATE TEMPORARY	O/E	一時 LOB を作成します。
LOB DESCRIBE	O/E	LOB から属性を取り出します。
LOB DISABLE BUFFERING	O/E	LOB バッファリングを使用禁止にします。
LOB ENABLE BUFFERING	O/E	LOB バッファリングを有効にします。
LOB ERASE	O/E	指定されたオフセットから始まる指定された量の LOB データを消去します。
LOB FILE CLOSE ALL	O/E	オープンしているすべての BFILE をクローズします。
LOB FILE SET	O/E	BFILE ロケータの DIRECTORY および FILENAME を設定します。
LOB FLUSH BUFFER	O/E	LOB のバッファをデータベース・サーバーに書き込みます。
LOB FREE TEMPORARY	O/E	LOB ロケータ用に一時領域を解放します。
LOB LOAD	O/E	BFILE の全部または一部を、内部 LOB にコピーします。
LOB OPEN	O/E	読取りまたは読取り/書込みアクセスするために、LOB または BFILE をオープンします。

EXEC SQL文	ソース/タイプ	用途
LOB READ	O/E	LOB または BFILE の一部をバッファに読み込みます。
LOB TRIM	O/E	LOB 値を切り捨てます。
LOB WRITE	O/E	バッファの内容を LOB に書き込みます。
OBJECT CREATE	O/E	キャッシュ内で参照可能オブジェクトを作成します。
OBJECT DELETE	O/E	オブジェクトに削除マークを設定します。
OBJECT DEREF	O/E	オブジェクトを間接参照します。
OBJECT FLUSH	O/E	永続オブジェクトをサーバーに送信します。
OBJECT GET	O/E	オブジェクト属性を C 言語のデータ型に変換します。
OBJECT RELEASE	O/E	キャッシュ内のオブジェクトを確保解除します。
OBJECT SET	O/E	キャッシュ内のオブジェクト属性を更新します。
OBJECT UPDATE	O/E	キャッシュ内のオブジェクトに更新マークを設定します。
OPEN	S/E	カーソルに対応付けられた問合せを実行します。
OPEN DESCRIPTOR	S/E	カーソルに対応付けられた問合せを実行します(ANSI 動的 SQL 方法 4)。
PREPARE	S/E	動的 SQL 文を解析します。
REGISTER CONNECT	O/E	外部プロシージャのコールを使用可能にします。
ROLLBACK	S/E	現行のトランザクションを終了し、現行のトランザクションで加えられた変更をすべて破棄し、ロックをすべて解除します(オプションでリソースを解放し、データベースとの接続を切断します)。
SAVEPOINT	S/E	後でロールバックする位置をトランザクション内に指定します。
SELECT	S/E	選択した値をホスト変数に割り当てて、1 つ以上の表、ビューまたはスナップショットからデータを取り出します。

EXEC SQL文	ソース/タイプ	用途
SET DESCRIPTOR	S/E	ホスト変数からの情報を記述子領域に設定します。
TYPE	O/D	外部のデータ型をユーザー定義のデータ型と同値化して、外部データ型をホスト変数のクラス全体に割り当てます。
UPDATE	S/E	表またはビューの実表の既存の値を変更します。
VAR	O/D	デフォルトのデータ型を無効にして、特定の外部データ型をホスト変数に割り当てます。
WHENEVER	S/D	エラー状態および警告状態の処置を指定します。

E.2 文の説明

ディレクティブおよび文はアルファベット順に並べてあります。各コマンドの説明には、次の項目があります。

ディレクティブおよび文	説明
用途	コマンドの基本的な用途を示します。
前提条件	必要な権限、および文を使用する前に実行する必要のある手順を示します。特記していないかぎり、ほとんどの文では、データベースがユーザーのインスタンスによってオープンされている必要があります。
構文	文のキーワードとパラメータを示します。
キーワードおよびパラメータ	各キーワードおよびパラメータの用途を示します。
使用上の注意	文の使用方法および条件を示します。
例	文の例文を示します。
関連項目	関連する文、句およびこのマニュアルの関連項目を示します。

E.3 構文図の読み方

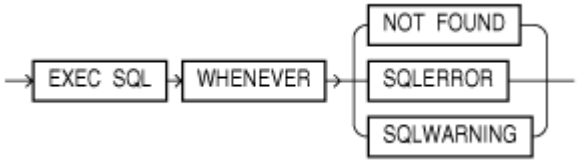
埋込みSQLの構文は、わかりやすいように構文図を使用して説明します。構文図は、正しい構文のパスを示す図です。

構文図は、左から右へと矢印が指す方向にたどってください。

文と他のキーワードは、四角形の中に大文字で表記されています。これらの文字は、四角形の中に表示されているとおり正確に入力してください。パラメータは、楕円形の中に小文字で表記されています。記述する文のパラメータを変数に置き換えてください。演算子、デリミタおよび終了記号は、円の中に表記されています。「はじめに」で定義されている表記規則に従い、セミコロ

ンで文を終了します。

構文図に複数のパスがある場合は、任意のパスを選択できます。キーワード、演算子またはパラメータの選択肢が複数ある場合は、オプションを縦に並べて示します。次の例では、まず縦方向を選択した後、横方向に進めます。

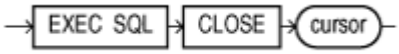


この図は、次の文がすべて有効であることを示しています。

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

E.3.1 必須のキーワードおよびパラメータ

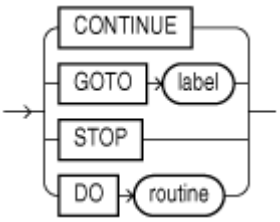
必須のキーワードおよびパラメータは、単一または代替の選択肢を縦に並べた状態で示します。単独の必須キーワードおよびパラメータはメイン・パス、つまり現在たどっている横線上に現れます。次の例では、*cursor*が必須パラメータです。



*emp_cursor*という名前のカーソルがある場合、この構文図によると、次の文は有効です。

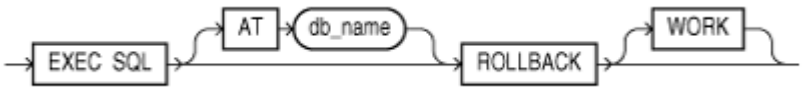
```
EXEC SQL CLOSE emp_cursor;
```

複数のキーワードまたはパラメータがメイン・パス上に縦に並んでいる場合は、その中のいずれかが必須になります。つまり、キーワードやパラメータを1つ選択する必要がありますが、それはメイン・パス上にあるものでなくてもかまいません。次の例では、4つのアクションのうち1つを選択する必要があります。



E.3.2 オプションのキーワードとパラメータ

キーワードとパラメータがメイン・パス内に縦に並べられている場合、それらはオプションです。次の例では、「AT : db_name」および「WORK」がオプション設定になります。

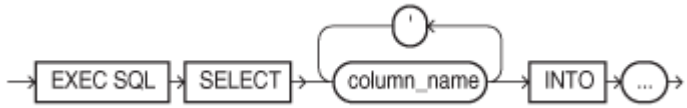


この図では、*oracle2*の名前のデータベースが存在する場合、次の文はすべて有効です。

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

E.3.3 構文ループ

ループは、その中の構文を何回でも繰り返せることを示します。次の例では、*column_name*がループの中にあります。したがって、列名を1つ選択した後に、他の列名をカンマで区切って繰り返し選択できます。



DEBIT、CREDITおよびBALANCEが列名の場合、この図では次の文がすべて有効です。

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

E.3.4 複数パーツの図

複数パーツの図では、メイン・パスがすべて端から端まで結合されていると考えます。次の例は2パーツの図です。

この図は、次の文が有効であることを示しています。

```
EXEC SQL PREPARE statement_name FROM string_literal;
```

E.3.5 Oracleの名前

表および列などのOracle Databaseオブジェクトの名前は、30文字以内であることが必要です。先頭文字は英文字であることが必要ですが、残りの文字には、英文字、数字、ドル記号($)、ポンド記号(#)およびアンダースコア(_)を任意に組み合せて使用できます。

ただし、名前を二重引用符(")で囲むと、有効な文字を任意に組み合せて使用できます。この場合、空白は有効な文字ですが、引用符は無効です。

Oracleの名前は、引用符で囲んだ場合を除いて大/小文字の区別がありません。

E.3.6 文の終了記号

どの埋込みSQL図の場合も、各文は文終了記号「;」で終わるものとみなされます。

E.4 ALLOCATE (実行可能埋込みSQL拡張機能)

用途

カーソル変数がPL/SQLブロックで参照されるように割り当てるか、オブジェクト・キャッシュに領域を割り当てます。

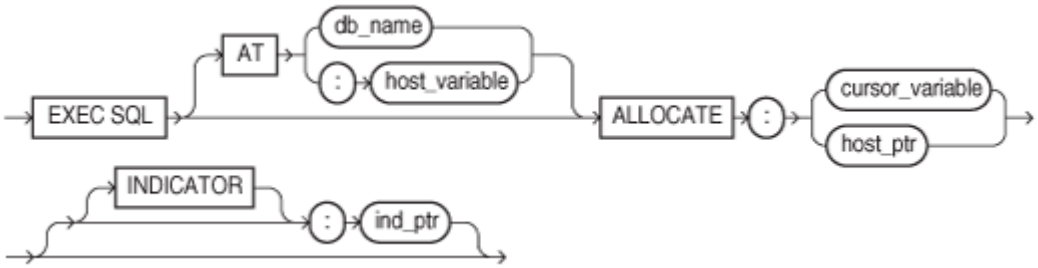
前提条件

カーソル変数にメモリーを割り当てる前に、sql_cursor型のカーソル変数を宣言する必要があります。

オブジェクト・キャッシュにメモリーを割り当てるには、その前にホスト構造体を指すポインタとオプションのインジケータ構造体を指すポインタを宣言する必要があります。

データベースへの接続が必ずアクティブである必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
db_name	CONNECT 文で先に設定されるデータベース接続名を含む NULL 終了文字列。省略した場合または空文字の場合は、デフォルトのデータベース接続とみなされます。
host_variable	データベース接続名を格納するホスト変数。
cursor_variable	割り当てるカーソル変数。
host_ptr	オブジェクト型に対して OTT により生成されるホスト構造体へのポインタ、sql_context 型のコンテキスト変数、OCIRowid へのタイプ・ポインタの ROWID 変数、または LOB の型に対応する LOB ロケータ変数。
ind_ptr	インジケータ構造体へのオプションのポインタ。

使用上の注意

カーソルは静的ですが、カーソル変数は特定の問合せに結び付けられていないため動的です。カーソル変数は、型の互換性のある任意の問合せに対してオープンできます。

例

この部分的な例では、Pro*C/C++プログラムでALLOCATE文を使用する方法を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
   SQL_CURSOR emp_cv;
   struct{ ... } emp_rec;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :emp_cv;
EXEC SQL EXECUTE
   BEGIN
      OPEN :emp_cv FOR SELECT * FROM emp;
   END;
END-EXEC;
for (;;)
   {
   EXEC SQL FETCH :emp_cv INTO :emp_rec;
   ...
   }
```

関連項目

「CACHE FREE ALL(実行可能埋込みSQL拡張機能)」

CLOSE (実行可能埋込みSQL) 。

「EXECUTE(実行可能埋込みSQL)」

FETCH (実行可能埋込みSQL) 。

FETCH DESCRIPTOR (実行可能埋込みSQL)

FREE (実行可能埋込みSQL拡張機能)。

E.5 ALLOCATE DESCRIPTOR (実行可能埋込みSQL)

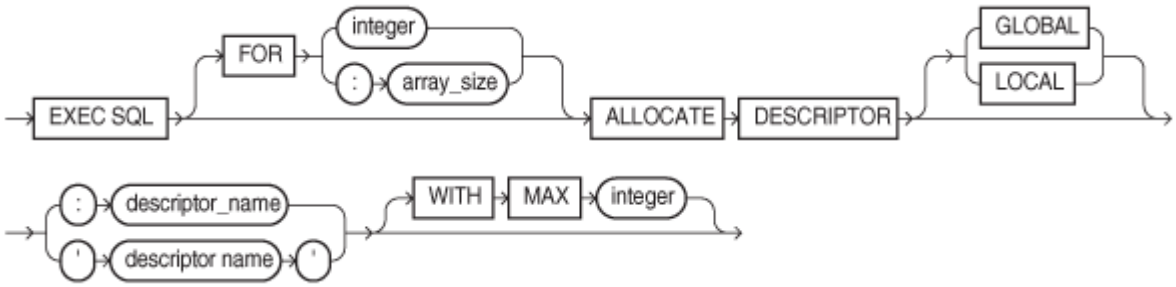
用途

記述子を割り当てるANSI動的SQL文です。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
array_size	処理される行数を格納するホスト変数。
integer	処理する行数。
descriptor_name	ANSI 記述子の名前を含むホスト変数。
descriptor name	ANSI 記述子の名前。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
WITH MAX *integer*	ホスト変数の最大数。デフォルトは 100 です。

使用上の注意

DYNAMIC=ANSIプリコンパイラ・オプションを使用してください。この文の使用の詳細は、ALLOCATE DESCRIPTORを参照してください。

例

```
EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;
```

関連項目

DESCRIBE DESCRIPTOR (実行可能埋込みSQL) 。

DEALLOCATE DESCRIPTOR (埋込みSQL文)

GET DESCRIPTOR (実行可能埋込みSQL)。

SET DESCRIPTOR (実行可能埋込みSQL)。

E.6 CACHE FREE ALL (実行可能埋込みSQL拡張機能)

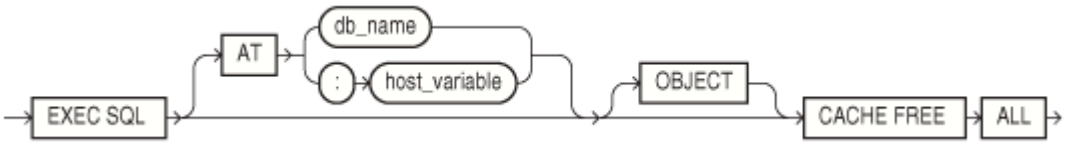
用途

オブジェクト・キャッシュ内のすべてのメモリーを解放します。

前提条件

アクティブなデータベース接続が存在している必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
db_name	CONNECT 文で先に設定されるデータベース接続名を含む NULL 終了文字列。省略した場合または空文字の場合は、デフォルトのデータベース接続とみなされます。
host_variable	データベース接続名を格納するホスト変数。

使用上の注意

接続カウントが0(ゼロ)になると、SQLLIBにより自動的にすべてのオブジェクト・キャッシュ・メモリーが解放されます。詳細は、CACHE FREE ALLを参照してください。

例

```
EXEC SQL AT mydb CACHE FREE ALL ;
```

関連項目

ALLOCATE (実行可能埋込みSQL拡張機能)。

FREE (実行可能埋込みSQL拡張機能)。

E.7 CALL (実行可能埋込みSQL)

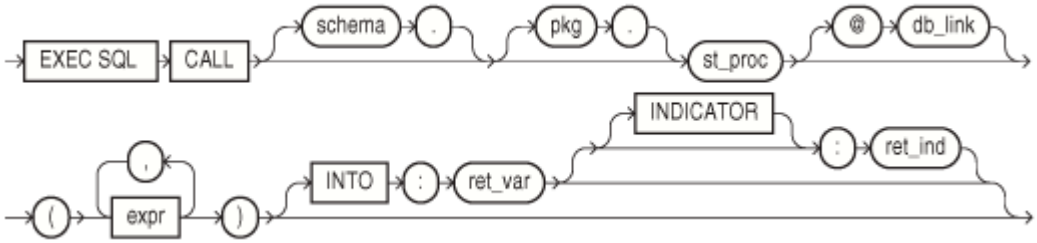
用途

ストアド・プロシージャをコールします。

前提条件

アクティブなデータベース接続が存在している必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
schema	プロシージャを格納するスキーマ。省略した場合、Oracle はプロシージャが固有のスキーマ内にあるとみなします。
pkg	プロシージャが格納されているパッケージ。
st_proc	コールするストアド・プロシージャ。
db_link	プロシージャが格納されているリモート・データベースへのデータベース・リンクの、完全または一部の名前。データベース・リンク参照の情報は、『*Oracle Database SQL 言語リファレンス*』を参照してください。
expr	プロシージャのパラメータ式のリスト。
ret_var	関数からの戻り値を受け取るホスト変数。
ret_ind	ret_var の標識変数。

使用上の注意

この文の詳細は、ストアドPL/SQLまたはJavaサブプログラムのコールを参照してください。

例

```
int emp_no;
char emp_name[10];
float salary;
char dept_name[20];
...
emp_no = 1325;
EXEC SQL CALL get_sal(:emp_no, :emp_name, :salary) INTO :dept_name ;
/* Print emp_name, salary, dept_name */
...
```

関連項目

なし。

E.8 CLOSE (実行可能埋込みSQL)

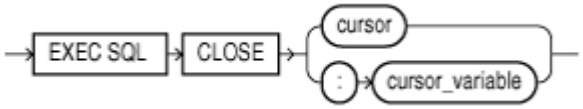
用途

カーソルのオープン時に取得したリソースを解放し、解析ロックを解除して、カーソルを使用禁止にします。

前提条件

MODE=ANSIの場合は、カーソルまたはカーソル変数はオープンである必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
cursor	クローズするカーソル。
cursor_variable	クローズするカーソル変数。

使用上の注意

クローズしたカーソルからは行をフェッチできません。カーソルを再オープンするには、そのカーソルがクローズされている必要はありません。HOLD_CURSORおよびRELEASE_CURSORのプリコンパイラ・オプションによって、CLOSE文の結果は異なります。これらのオプションの詳細は、プリコンパイラのオプションを参照してください。

例

この例では、CLOSE文の使用方法を示します。

```
EXEC SQL CLOSE emp_cursor;
```

関連項目

PREPARE (実行可能埋込みSQL)。

DECLARE CURSOR (埋込みSQLディレクティブ)。

OPEN (実行可能埋込みSQL) 。

E.9 COLLECTION APPEND (実行可能埋込みSQL拡張機能)

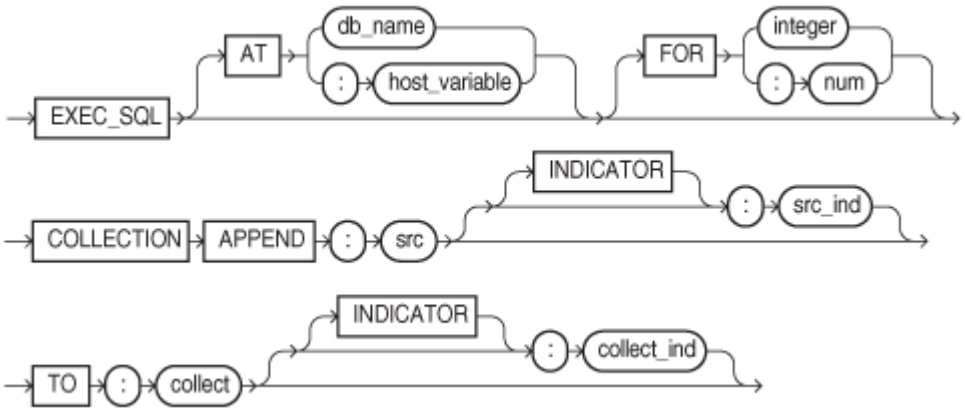
用途

1つのコレクションの要素を別のコレクションの最後に追加します。

前提条件

NULLコレクションに追加すること、またはコレクションの上限を超えて追加することはできません。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION APPENDを参照してください。

関連項目

他のCOLLECTION文を参照してください。

E.10 COLLECTION DESCRIBE (実行可能埋込みSQL拡張機能)

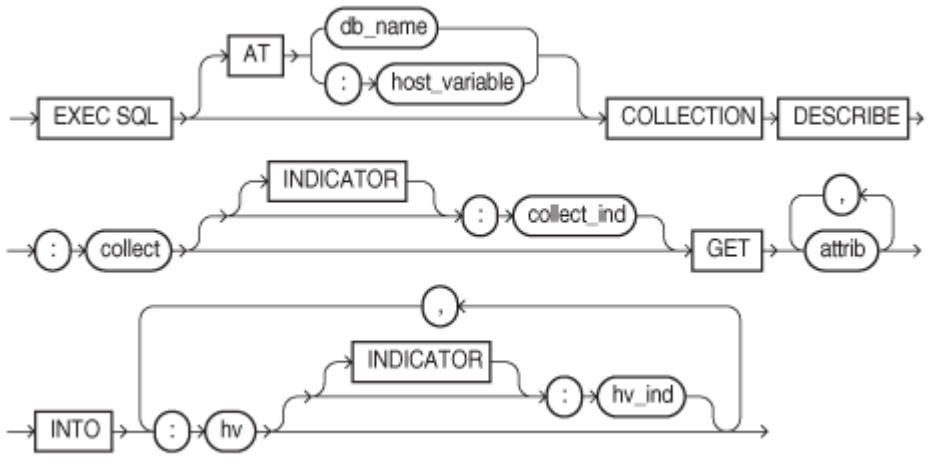
用途

コレクションについての情報を取得します。

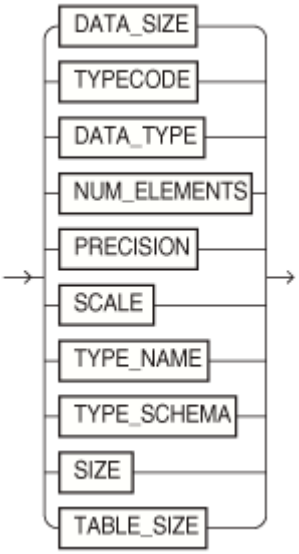
前提条件

ALLOCATEおよびOBJECT GET文を使用して記述子を割り当て、記述子にコレクション属性を格納します。

構文



attribは次のとおりです。



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION DESCRIBEを参照してください。

関連項目

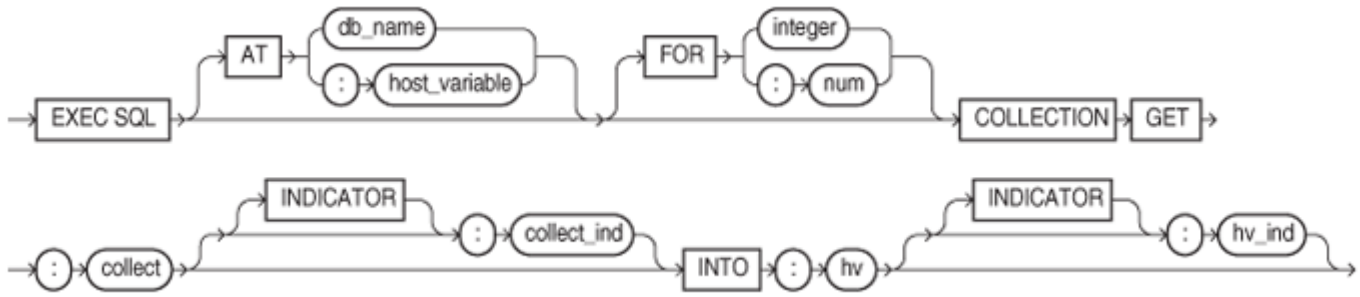
他のCOLLECTION文を参照してください。

E.11 COLLECTION GET (実行可能埋込みSQL拡張機能)

用途

コレクションの要素を取得します。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION文を参照してください。

関連項目

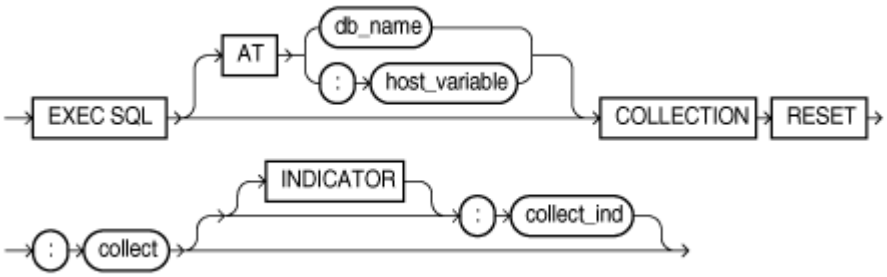
他のCOLLECTION文を参照してください。

E.12 COLLECTION RESET (実行可能埋込みSQL拡張機能)

用途

コレクションのスライス・エンドポイントをコレクションの最初にリセットします。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION RESETを参照してください。

関連項目

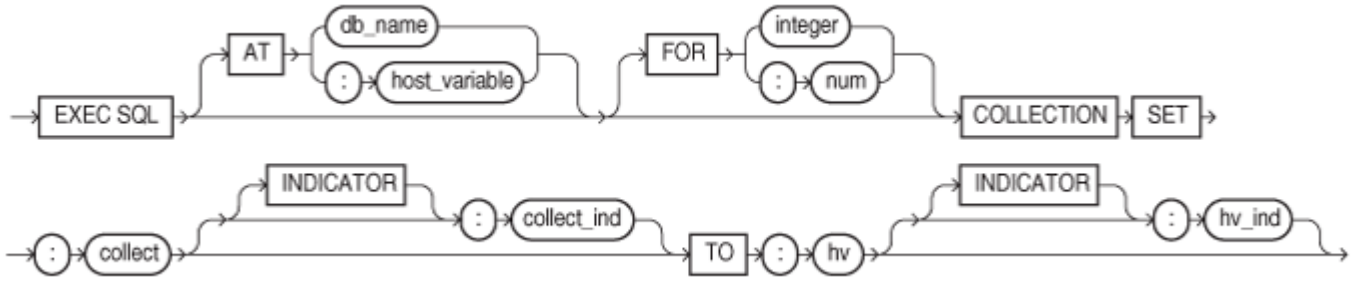
他のCOLLECTION文を参照してください。

E.13 COLLECTION SET (実行可能埋込みSQL拡張機能)

用途

コレクションの現行のスライスの要素値を更新します。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION SETを参照してください。

関連項目

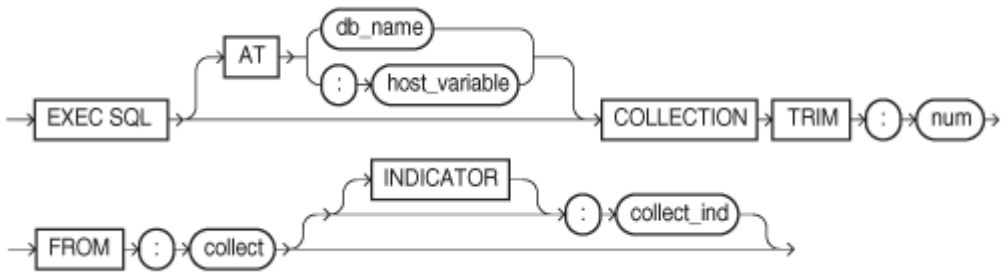
他のCOLLECTION文を参照してください。

E.14 COLLECTION TRIM (実行可能埋込みSQL拡張機能)

用途

コレクションの最後から要素を削除します。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、COLLECTION TRIMを参照してください。

関連項目

他のCOLLECTION文を参照してください。

E.15 COMMIT (実行可能埋込みSQL)

用途

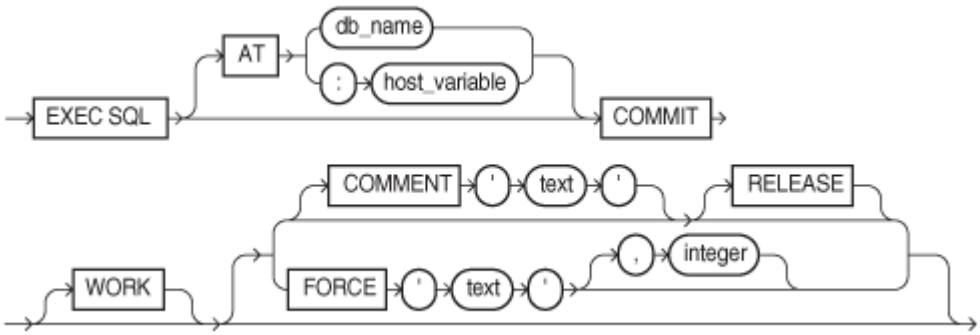
データベースの変更内容をすべて確定し、またオプションですべてのリソースを解放して切断し、現行のトランザクションを終了します。

前提条件

カレント・トランザクションをコミットするために必要な権限はありません。

ユーザーがコミットしたインダウトの分散トランザクションを手動でコミットするには、FORCE TRANSACTIONのシステム権限が必要です。他のユーザーがコミットしたインダウトの分散トランザクションを手動でコミットするには、FORCE ANY TRANSACTIONのシステム権限が必要です。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	どのデータベースに対して COMMIT 文を発行するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 値が db_name のホスト変数。この句を省略した場合、Oracle はデフォルトのデータベースに対して文を発行します。
WORK	標準 SQL への準拠のためにのみサポートされています。COMMIT 文と COMMIT WORK 文は同等です。
COMMENT	現行のトランザクションに対応付けるコメントを指定します。'text'は 50 文字以内の引用符付きリテラルで、トランザクションがインダウトになった場合に、Oracle によりデータ・ディクショナリ・ビューDBA_2PC_PENDING にトランザクション ID とともに格納されます。

キーワードおよびパラメータ	説明
RELEASE	リソースをすべて解放し、アプリケーションをサーバーから切断します。
FORCE	インダウトの分散トランザクションを手動でコミットします。トランザクションは、ローカル・トランザクション ID またはグローバル・トランザクション ID を含む'text'により指定します。このようなトランザクションの ID を検索するには、データ・ディクショナリ・ビュー DBA_2PC_PENDING に問合せをします。また、オプションの整数を使用してトランザクションにシステム変更番号(SCN)を明示的に割り当てることができます。integer を省略した場合、トランザクションはカレント SCN を使用してコミットされます。

使用上の注意

プログラムの最後のトランザクションは、COMMITコマンドまたはROLLBACK文およびRELEASEオプションを使用して、必ず明示的にコミットまたはロールバックしてください。プログラムが異常終了すると、Oracleは自動的に変更をロールバックします。

COMMIT文は、ホスト変数やプログラムの制御の流れには影響しません。この文の詳細は、データベースの概要を参照してください。

例

この例では、埋込みSQL COMMIT文の使用方法を示します。

```
EXEC SQL AT sales_db COMMIT RELEASE;
```

関連項目

ROLLBACK (実行可能埋込みSQL)

SAVEPOINT (実行可能埋込みSQL)

E.16 CONNECT (実行可能埋込みSQL拡張機能)

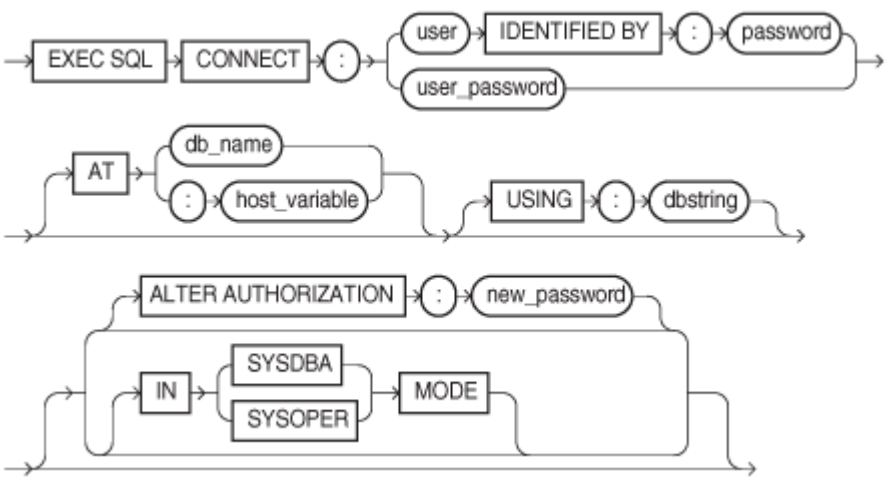
用途

データベースにログインすること。

前提条件

指定するデータベースに対してCREATE SESSIONのシステム権限が必要です。

構文



キーワードおよびパラメータ	説明
user password	ユーザー名およびパスワードを個別に指定します。
user_password	ユーザー名とパスワードをスラッシュ(/)で区切って格納した 1 つのホスト変数。 Oracle で、使用しているオペレーティング・システムとの接続を検証するには、「/」を *user_password* 値として指定します。
AT	接続先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 値が db_name のホスト変数。この句を省略した場合、Oracle はデフォルトのデータベースに対して文を発行します。
USING	非デフォルトのデータベースへの接続に使用される Oracle Net データベース指定の文字列を指定します。この句を省略した場合は、デフォルトのデータベースに接続します。
ALTER AUTHORIZATION	パスワードを次の文字列に変更します。
new_password	新しいパスワード。
IN SYSDBA MODE IN SYSOPER MODE	SYSDBA または SYSOPER システム権限で接続します。ALTER AUTHORIZATION が使用されているとき、またはプリコンパイラ・オプションの AUTO_CONNECT が YES に設定されているときは、接続が許可されません。

使用上の注意

プログラムは複数の接続を持つことができますが、デフォルト・データベースには1度しか接続できません。この文の詳細は、OCIリリース7コールの埋込みを参照してください。

例

次の例では、CONNECTの使用方法を示します。

```
EXEC SQL CONNECT :username
    IDENTIFIED BY :password ;
```

この文では、*userid*の値に、'SCOTT/TIGER'のように、*username*の値と*password*の値をスラッシュ(/)で区切って使用することもできます。

```
EXEC SQL CONNECT :userid ;
```

関連項目

COMMIT (実行可能埋込みSQL) 。

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

ROLLBACK (実行可能埋込みSQL)

E.17 CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)

用途

EXEC SQL CONTEXT USE文で参照されているSQLLIBランタイム・コンテキストを初期化します。

前提条件

ランタイム・コンテキストはsql_context型で宣言する必要があります。

構文

キーワードおよびパラメータ

キーワードおよびパラメータ	説明
context	メモリーを割り当てる SQLLIB ランタイム・コンテキスト。

使用上の注意

マルチスレッド・アプリケーションでは、ランタイム・コンテキストごとにこのファンクションを実行します。

この文の詳細は、OCIリリース8のSQLLIB拡張相互運用性を参照してください。

例

この例では、Pro*C/C++プログラムでCONTEXT ALLOCATE文を使用する方法を示します。

```
EXEC SQL CONTEXT ALLOCATE :ctx1;
```

関連項目

CONTEXT FREE (実行可能埋込みSQL拡張機能)。

CONTEXT USE (Oracle埋込みSQLディレクティブ)。

「ENABLE THREADS(実行可能埋込みSQL拡張機能)」

E.18 CONTEXT FREE (実行可能埋込みSQL拡張機能)

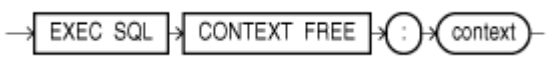
用途

ランタイム・コンテキストに関連付けられたすべてのメモリーを解放し、NULLポインタをホスト・プログラム変数に代入します。

前提条件

CONTEXT FREE文を使用してランタイム・コンテキストに割り当てられたメモリーを解放する前に、CONTEXT ALLOCATE文を使用して、指定されているランタイム・コンテキストにメモリーを割り当てる必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
:*context*	メモリーの割当てを解除する、割当て済ランタイム・コンテキスト。

使用上の注意

この文の詳細は、OCIリリース8のSQLLIB拡張相互運用性を参照してください。

例

この例では、Pro*C/C++プログラムでCONTEXT FREE文を使用する方法を示しています。

```
EXEC SQL CONTEXT FREE :ctx1;
```

関連項目

CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)。

CONTEXT USE (Oracle埋込みSQLディレクティブ)。

「ENABLE THREADS(実行可能埋込みSQL拡張機能)」

E.19 CONTEXT OBJECT OPTION GET (実行可能埋込みSQL拡張機能)

用途

CONTEXT OBJECT OPTION SETによって設定される使用中のコンテキストのオプションの値を決定します。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。

構文

キーワードおよびパラメータ

キーワードおよびパラメータ	説明
option	オプション値については、CONTEXT OBJECT OPTION SET を参照してください。
host_variable	*option* リストと同じ順序で表された STRING、VARCHAR、CHARZ 型の出力変数。

使用上の注意

CONTEXT OBJECT OPTION SETを参照してください。

例

```
char EuroFormat[50];
...
EXEC SQL CONTEXT OBJECT OPTION GET DATEFORMAT INTO :EuroFormat ;
printf("Date format is %s¥n", EuroFormat);
```

関連項目

CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)。

CONTEXT FREE (実行可能埋込みSQL拡張機能)。

「CONTEXT OBJECT OPTION SET(実行可能埋込みSQL拡張機能)」

CONTEXT USE (Oracle埋込みSQLディレクティブ)。

E.20 CONTEXT OBJECT OPTION SET (実行可能埋込みSQL拡張機能)

用途

使用中のコンテキストに対する指定済の日付属性DATEFORMATおよびDATELANGにオプションを設定します。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。

構文

キーワードおよびパラメータ

キーワードおよびパラメータ	説明
option	オプション値については、CONTEXT OBJECT OPTION SET を参照してください。
host_variable	STRING、VARCHAR、CHARZ 型の入力変数。*option* リストと同じ順序。

使用上の注意

CONTEXT OBJECT OPTION GETを参照してください。

例

```
char *new_format = "DD-MM-YYY";
char *new_lang = "French";
...
EXEC SQL CONTEXT OBJECT OPTION SET DATEFORMAT, DATELANG TO :new_format, :new_lang;
```

関連項目

CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)。

CONTEXT FREE (実行可能埋込みSQL拡張機能)。

CONTEXT USE (Oracle埋込みSQLディレクティブ)。

「CONTEXT OBJECT OPTION SET(実行可能埋込みSQL拡張機能)」

E.21 CONTEXT USE (Oracle埋込みSQLディレクティブ)

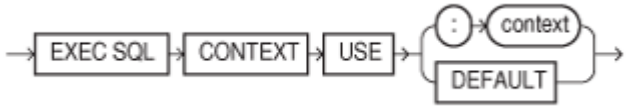
用途

後続の実行SQL文で指定のSQLLIBランタイム・コンテキストを使用するようにプリコンパイラに指示します。

前提条件

CONTEXT USEディレクティブによって指定されたランタイム・コンテキストは、事前に宣言されている必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
context	後続の実行 SQL 文によって使用される、割当て済ランタイム・コンテキストです。たとえば、使用するコンテキストをソース・コードに指定した後(複数のコンテキストを割り当てることができます)、Oracle サーバーに接続し、コンテキストのスコープ内でデータベースを操作できます。DEFAULT は作業したグローバル・コンテキストが使用されることを示します。
DEFAULT	グローバル・コンテキストが使用されることを示します。

使用上の注意

この文はEXEC SQL INCLUDEやEXEC ORACLE OPTIONなどの宣言文では無効です。この文は、EXEC SQL WHENEVERディレクティブと同様に、指定したソース・ファイル内でこのディレクティブの後に続くすべての実行SQL文に影響し、C言語の標準のスコープ規則には従いません。

この文の詳細は、OCIリリース8のSQLLIB拡張相互運用性を参照してください。

例

この例では、Pro*C/C++の埋込みSQLプログラムでCONTEXT USEディレクティブを使用する方法を示します。

```
EXEC SQL CONTEXT USE :ctx1;
```

関連項目

CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)。

CONTEXT FREE (実行可能埋込みSQL拡張機能)。

「ENABLE THREADS(実行可能埋込みSQL拡張機能)」

E.22 DEALLOCATE DESCRIPTOR (埋込みSQL文)

用途

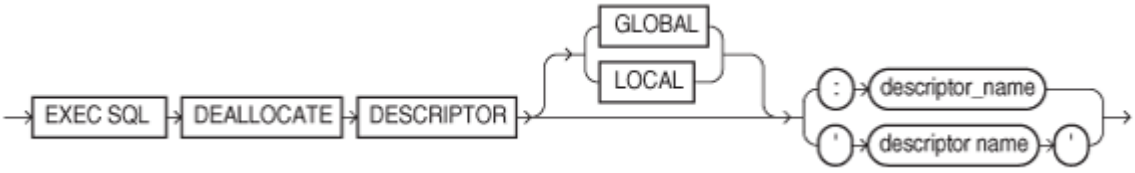
記述子領域の割当てを解除し、メモリーを解放するANSI動的SQL文です。

前提条件

DEALLOCATE DESCRIPTOR文で指定されている記述子は、ALLOCATE DESCRIPTOR文を使用して事前に割り当

てる必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
descriptor_name	割り当てられた ANSI 記述子名を格納するホスト変数。
'descriptor name'	割り当てられた ANSI 記述子の名前。

使用上の注意

DYNAMIC=ANSIプリコンパイラ・オプションを使用してください。

この文の詳細は、DEALLOCATE DESCRIPTORを参照してください。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' ;
```

関連項目

ALLOCATE DESCRIPTOR (実行可能埋込みSQL)。

DESCRIBE(実行可能埋込みSQL拡張機能)

GET DESCRIPTOR (実行可能埋込みSQL)。

PREPARE (実行可能埋込みSQL)。

SET DESCRIPTOR (実行可能埋込みSQL)。

E.23 DECLARE CURSOR (埋込みSQLディレクティブ)

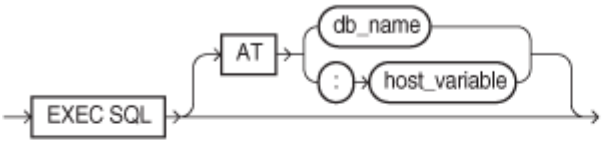
用途

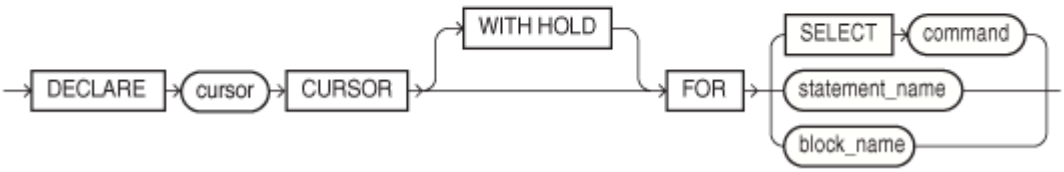
カーソルを宣言するためにカーソルに名前を付け、それをSQL文またはPL/SQLブロックに対応付けます。

前提条件

SQL文またはPL/SQLブロックの識別子を使用してカーソルを対応付けるには、DECLARE STATEMENT文を使用してこの識別子を事前に宣言する必要があります。

構文





キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	カーソルを宣言するデータベースを指定します。次のいずれかを使用してデータベースを指定します。
db_name	DECLARE DATABASE 文を使用してすでに宣言されているデータベース識別子。
host_variable	すでに宣言されている *db_name* 値のホスト変数。 この句を省略した場合、Oracle はデフォルトのデータベースに対してこのカーソルを宣言します。
cursor	宣言するカーソルの名前。
WITH HOLD	カーソルは COMMIT の実行後もオープン状態のままです。UPDATE の場合は、カーソルを宣言しないでください。
SELECT 文	カーソルに対応付ける SELECT 文。直後の文に INTO 句を含めないでください。
statement_name	カーソルに対応付ける SQL 文または PL/SQL ブロックを指定します。 *statement_name* または *block_name* は、DECLARE STATEMENT 文を使用して事前に宣言する必要があります。

使用上の注意

カーソルは、他の埋込みSQL文で参照する前に、宣言する必要があります。カーソル宣言のスコープはプリコンパイル・ユニット内全体になるため、各カーソルの名前はスコープ内で一意であることが必要です。1つのプリコンパイル・ユニット内で同じ名前のカーソルを複数宣言することはできません。

カーソルは、UPDATE文またはDELETE文のWHERE句内でCURRENT OF構文を使用して参照できます。このとき、カーソルはOPEN文を使用してオープンし、FETCH文を使用して行に位置付けられている必要があります。この文の詳細は、埋込みPL/SQLのカーソルの使用方法を参照してください。

例

この例では、DECLARE CURSOR文の使用方法を示します。

```
EXEC SQL DECLARE emp_cursor CURSOR
    FOR SELECT ename, empno, job, sal
```

```
        FROM emp
        WHERE deptno = :deptno
        FOR UPDATE OF sal;
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

DECLARE STATEMENT (埋込みSQLディレクティブ) 。

DELETE (実行可能埋込みSQL) 。

FETCH (実行可能埋込みSQL) 。

OPEN DESCRIPTOR (実行可能埋込みSQL)

PREPARE (実行可能埋込みSQL)。

SELECT (実行可能埋込みSQL)。

UPDATE (実行可能埋込みSQL)

E.24 DECLARE DATABASE (Oracle埋込みSQLディレクティブ)

用途

後続の埋込みSQL文でアクセスされるデフォルト以外のデータベースの識別子を宣言します。

前提条件

非デフォルト・データベースのユーザー名にアクセスできる必要があります。

構文

キーワードおよびパラメータ

キーワードおよびパラメータ	説明
db_name	非デフォルト・データベースに対して設定する識別子。

使用上の注意

デフォルト以外のデータベースに対して*db_name*を宣言するのは、他の埋込みSQL文がAT句を使用してそのデータベースを参照できるようにするためです。AT句を指定してCONNECT文を発行する前に、DECLARE DATABASE文を使用してデフォルト以外のデータベースに対して*db_name*を宣言する必要があります。

この文の詳細は、単一の明示的接続を参照してください。

例

この例では、DECLARE DATABASEディレクティブの使用方法を示します。

```
EXEC SQL DECLARE oracle3 DATABASE ;
```

関連項目

COMMIT (実行可能埋込みSQL) 。

CONNECT (実行可能埋込みSQL拡張機能)

DECLARE CURSOR (埋込みSQLディレクティブ)。

DECLARE STATEMENT (埋込みSQLディレクティブ) 。

DELETE (実行可能埋込みSQL) 。

EXECUTE ...END-EXEC (実行可能埋込みSQL拡張機能)

EXECUTE IMMEDIATE (実行可能埋込みSQL)。

INSERT (実行可能埋込みSQL) 。

SELECT (実行可能埋込みSQL)。

UPDATE (実行可能埋込みSQL)

E.25 DECLARE STATEMENT (埋込みSQLディレクティブ)

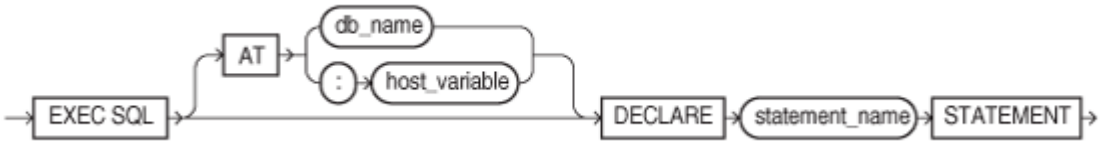
用途

SQL文またはPL/SQLブロックの識別子を宣言し、他の埋込みSQL文で使用できるようにします。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	SQL 文または PL/SQL ブロックが宣言されるデータベースを識別します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 値が db_name のホスト変数。この句を省略した場合、Oracle ではデフォルトのデータベースに対して SQL 文または PL/SQL ブロックが宣言されます。
statement_name	文に対して宣言する識別子。

使用上の注意

DECLARE STATEMENT文を使用してSQL文またはPL/SQLブロックの識別子を宣言する必要があるのは、その識別子を参照するDECLARE CURSOR文の埋込みSQLプログラム内での位置が、文またはブロックを解析して識別子と対応付けるPREPARE文よりも物理的に(論理的ではなく)前になっているときのみです。

文の宣言のスコープは、カーソルの宣言と同様に、プリコンパイル・ユニット内全体です。この文の詳細は、データ型とホスト変数

およびOracle動的SQLを参照してください。

例I

この例では、DECLARE STATEMENT文の使用方法を示します。

```
EXEC SQL AT remote_db DECLARE my_statement STATEMENT;
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

例II

このPro*C/C++の埋込みSQLプログラムからの例では、DECLARE CURSOR文がPREPARE文の前にあるため、DECLARE STATEMENT文が必要です。

```
EXEC SQL DECLARE my_statement STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;
EXEC SQL PREPARE my_statement FROM :my_string;
...
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

FETCH (実行可能埋込みSQL) 。

OPEN DESCRIPTOR (実行可能埋込みSQL)

PREPARE (実行可能埋込みSQL)。

E.26 DECLARE TABLE (Oracle埋込みSQLディレクティブ)

用途

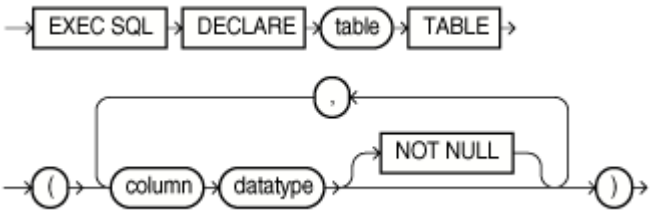
それぞれの列のデータ型、デフォルト値、Oracleプリコンパイラによる意味検査のためのNULLまたはNOT NULL仕様部など、表またはビューの構造を定義します。

前提条件

なし。

構文

リレーショナル表に使用する構文は、次のとおりです。



オブジェクト表に使用する構文は、次のとおりです。

キーワードおよびパラメータ

キーワードおよびパラメータ	説明
table	宣言した表の名前。
column	表の列。
datatype	列のデータ型。データ型の詳細は、Oracle のデータ型を参照してください。 データ型がユーザー定義オブジェクトの場合は、*size* をカッコで囲んで入力できます。size はマクロまたは複雑な C 言語の式にはできません。size は省略できます。例を参照してください。
NOT NULL	NULL を含めることのできない列を指定します。
obj_type	オブジェクト型を表します。

使用上の注意

この文の使用方法は、DECLARE TABLEの使用についてを参照してください。

例

次の文では、PARTNO、BINおよびQTYの列を含むPARTSという表を宣言しています。

```
EXEC SQL DECLARE parts TABLE
     (partno NUMBER  NOT NULL,
      bin    NUMBER,
      qty    NUMBER);
```

次のようにオブジェクト型を使用します。

```
EXEC SQL DECLARE person TYPE AS OBJECT (name VARCHAR2(20), age INT);
EXEC SQL DECLARE odjtab1 TABLE OF person;
```

関連項目

「DECLARE TYPE(Oracle埋込みSQLディレクティブ)」

E.27 DECLARE TYPE (Oracle埋込みSQLディレクティブ)

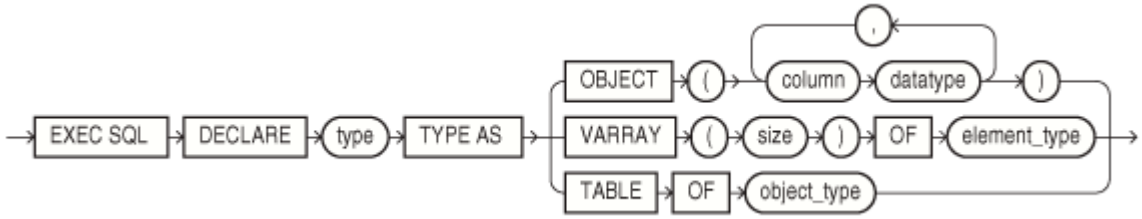
用途

プリコンパイラによる意味検査に使用される型の属性を定義します。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
column	列の名前。
datatype	列のデータ型。
size	VARRAY の要素数。
element_type	VARRAY の要素型。オブジェクトにすることもできます。
object_type	以前に宣言されていたオブジェクト型。

使用上の注意

この文の使用方法は、DECLARE TYPEの使用についてを参照してください。

例

```
EXEC SQL DECLARE project_type TYPE AS OBJECT(
               pno           CHAR(5),
               pname         CHAR(20),
               budget        NUMBER);
EXEC SQL DECLARE project_array TYPE as VARRAY(20) OF project_type ;
EXEC SQL DECLARE employees TYPE AS TABLE OF emp_objects ;
```

関連項目

DECLARE TABLE (Oracle埋込みSQLディレクティブ)

E.28 DELETE (実行可能埋込みSQL)

用途

表またはビューの実表から行を削除します。

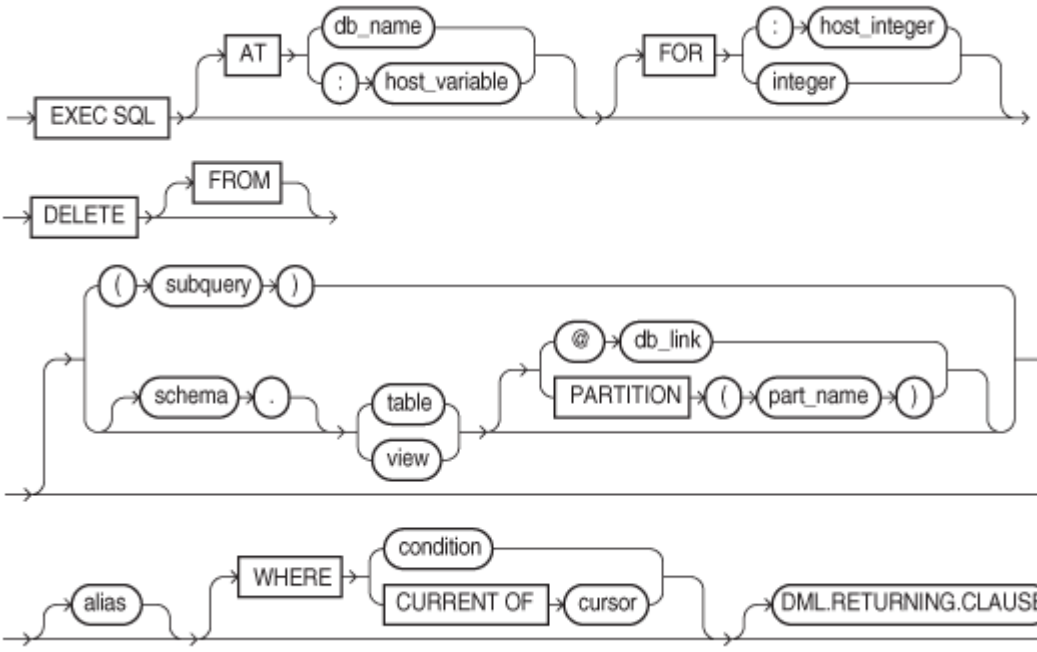
前提条件

表から行を削除するには、表がユーザーのスキーマ内にあるか、表に対してDELETEの権限を持っている必要があります。

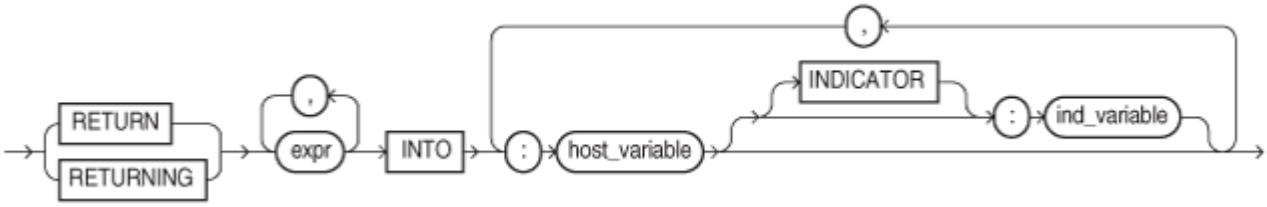
ビューの実表から行を削除するには、ビューが属するスキーマの所有者が、実表に対してDELETEの権限を持っている必要があります。また、ビューがユーザー所有のスキーマ以外のスキーマにある場合は、ビューに対するDELETEの権限を付与されている必要があります。

DELETE ANY TABLEのシステム権限では、どの表またはビューの実表からでも行を削除できます。

構文



DML RETURNING句は次のとおりです。



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	どのデータベースに対して DELETE 文を発行するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name:* DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した *db_name* の値を持つホスト変数。 この句を省略した場合、DELETE 文はデフォルトのデータベースに対して発行されます。
FOR *:host_integer*	WHERE 句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略した場合、データベースは最小の配列の各コンポーネントにつき 1 回ずつ文が実行されます。
subquery	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文は、SELECT を参照してください。
schema	表またはビューを含むスキーマ。schema を省略した場合、データベースでは表またはビューが自分のスキーマ内にあるとみなされます。

キーワードおよびパラメータ	説明
table	行を削除する表の名前。
view	ビューの名前。データベースはビューの実表から行を削除します。
FOR *:host_integer*	WHERE 句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略した場合、データベースは最小の配列の各コンポーネントにつき 1 回ずつ文が実行されます。
subquery	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文は、SELECT を参照してください。
schema	表またはビューを含むスキーマ。schema を省略した場合、データベースでは表またはビューが自分のスキーマ内にあるとみなされます。
table	行を削除する表の名前。
view	ビューの名前。データベースではビューの実表から行が削除されます。

使用上の注意

WHERE句のホスト変数は、すべてスカラーか、すべて配列である必要があります。変数がスカラーの場合、データベースはDELETE文を1回のみ実行します。変数が配列の場合は、データベースでは配列のコンポーネントのセットごとに文が1回実行されます。1回の実行で0行、1行または複数行を削除できます。

WHERE句の配列ホスト変数は、サイズが異なっていてもかまいません。この場合、データベースで文が実行される回数は、次の値のうち小さい方によって決まります。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値

この条件を満たす行が存在しない場合、行は削除されず、SQLCODEはNOT_FOUND条件を戻します。

削除された行の累積数はSQLCAを介して戻されます。WHERE句に配列ホスト変数が指定されていると、DELETE文によって処理された配列のすべてのコンポーネントにおよぶ削除行数の合計がこの値に設定されます。

条件を満たす行がない場合、データベースはSQLCAのSQLCODEを介してエラーを戻します。WHERE句を省略した場合、データベースはSQLCAのSQLWARNの第5コンポーネントに警告フラグを設定します。この文とSQLCAの詳細は、ランタイム・エラーの処理を参照してください。

DELETE文でコメントを使用して、指示(つまりヒント)をオプティマイザに渡すことができます。オプティマイザは、これらのヒントを使用して文の実行計画を選択します。

例

この例では、Pro*C/C++の埋込みSQLプログラムにおけるDELETE文の使用方法を示します。

```
EXEC SQL DELETE FROM emp
    WHERE deptno = :deptno
    AND job = :job;
```

```
EXEC SQL DECLARE emp_cursor CURSOR
    FOR SELECT empno, comm
        FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH c1
    INTO :emp_number, :commission;
EXEC SQL DELETE FROM emp
    WHERE CURRENT OF emp_cursor;
```

関連項目

「DECLARE DATABASE(Oracle埋込みSQLディレクティブ)」

DECLARE STATEMENT (埋込みSQLディレクティブ) 。

E.29 DESCRIBE (実行可能埋込みSQL拡張機能)

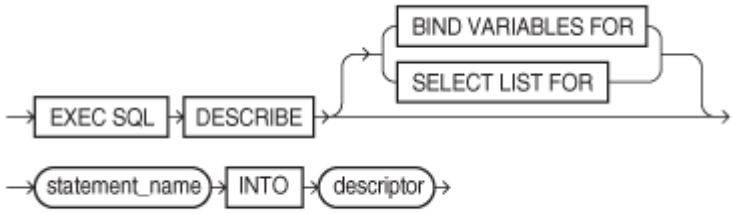
用途

Oracle記述子に動的SQL文またはPL/SQLブロックについての情報を入力します。

前提条件

埋込みSQLのPREPARE文を使用して、SQL文またはPL/SQLブロックを事前に準備しておく必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
BIND VARIABLES FOR	SQL 文または PL/SQL ブロックの入力変数に関する情報を保持する記述子を初期化します。
SELECT LIST FOR	SELECT 文の選択リストに関する情報を保持する記述子を初期化します。 デフォルトは SELECT LIST FOR です。
statement_name	PREPARE 文を使用して事前に準備した SQL 文または PL/SQL ブロックを指定します。
descriptor	入力する記述子の名前。

使用上の注意

埋込みSQLプログラム内のバインド記述子または選択記述子を操作するには、その前にDESCRIBE文を発行する必要があり

ます。

入力変数と出力変数の両方を同じ記述子に記述することはできません。

DESCRIBE文で検出される変数の数は、一意に名前が指定されたプレースホルダの合計数ではなく、準備するSQL文またはPL/SQLブロックのプレースホルダの合計数です。この文の詳細は、Oracle動的SQLを参照してください。

例

この例では、Pro*C/C++の埋込みSQLプログラムにおけるDESCRIBE文の使用方法を示します。

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL DECLARE emp_cursor
    FOR SELECT empno, ename, sal, comm
            FROM emp
            WHERE deptno = :dept_number;
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement
    INTO bind_descriptor;
EXEC SQL OPEN emp_cursor
    USING bind_descriptor;
EXEC SQL DESCRIBE SELECT LIST FOR my_statement
    INTO select_descriptor;
EXEC SQL FETCH emp_cursor
    INTO select_descriptor;
```

関連項目

PREPARE (実行可能埋込みSQL)。

E.30 DESCRIBE DESCRIPTOR (実行可能埋込みSQL)

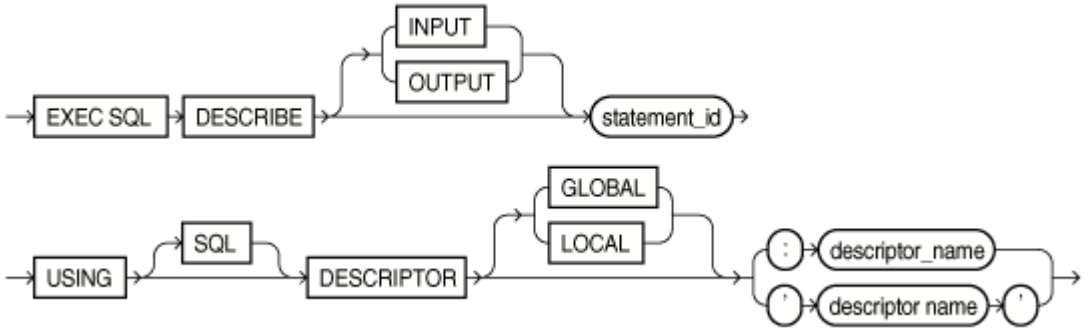
用途

SQL文についての情報を取得するために使用するANSI動的SQL文で、情報は記述子に格納されます。

前提条件

埋込みSQLのPREPARE文を使用して、SQL文を事前に準備しておく必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
statement_id	事前に準備されている SQL 文または PL/SQL ブロックの名前。OUTPUT はデフォルトです。

キーワードおよびパラメータ	説明
desc_name	SQL 文の情報を保持する記述子名を格納するホスト変数。
'descriptor name'	記述子の名前。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。

使用上の注意

DYNAMIC=ANSIプリコンパイラ・オプションを使用してください。

INPUT記述子では、COUNTおよびNAMEのみインプリメントされます。

DESCRIBE文で検出される変数の数は、一意に名前が指定されたプレースホルダの合計数ではなく、準備するSQL文またはPL/SQLブロックのプレースホルダの合計数です。この文の詳細は、DESCRIBE INPUTおよびDESCRIBE OUTPUTを参照してください。

例

```
EXEC SQL PREPARE s FROM :my_stament;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' ;
```

関連項目

ALLOCATE DESCRIPTOR (実行可能埋込みSQL)。

DEALLOCATE DESCRIPTOR (埋込みSQL文)。

GET DESCRIPTOR (実行可能埋込みSQL)。

PREPARE (実行可能埋込みSQL)。

SET DESCRIPTOR (実行可能埋込みSQL)。

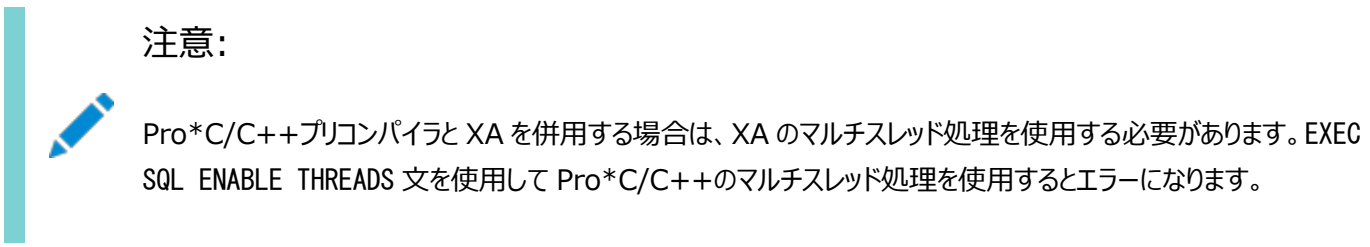
E.31 ENABLE THREADS (実行可能埋込みSQL拡張機能)

用途

複数のスレッドをサポートするプロセスを初期化します。

前提条件

マルチスレッド・アプリケーションをサポートするプラットフォーム用にプリコンパイラ・アプリケーションを開発し、このプラットフォームでコンパイルを実行して、コマンドラインにTHREADS=YESを指定する必要があります。



構文

キーワードおよびパラメータ

なし。

使用上の注意

ENABLE THREADS文は、他の実行SQL文の前、かつスレッドを作成する前に実行する必要があります。この文にはホスト変数を指定する必要はありません。

この文の詳細は、OCIリリース8のSQLLIB拡張相互運用性を参照してください。

例

この例では、Pro*C/C++プログラムでENABLE THREADS文を使用する方法を示しています。

```
EXEC SQL ENABLE THREADS;
```

関連項目

CONTEXT ALLOCATE (実行可能埋込みSQL拡張機能)。

CONTEXT FREE (実行可能埋込みSQL拡張機能)。

CONTEXT USE (Oracle埋込みSQLディレクティブ)。

E.32 EXECUTE ... END-EXEC (実行可能埋込みSQL拡張機能)

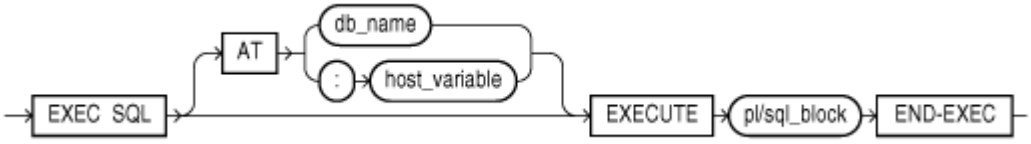
用途

Pro*C/C++プログラムに無名PL/SQLブロックを埋め込みます。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	PL/SQL ブロックをどのデータベースに対して実行するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した db_name の値を持つホスト変数。この句を省略した場合、PL/SQL ブロックはデフォルトのデータベースに対して実行されます。

キーワードおよびパラメータ	説明
pl/sql_block	PL/SQL ブロックの作成方法など、PL/SQL の詳細は『*Oracle Database PL/SQL 言語リファレンス*』を参照してください。
END-EXEC	このキーワードは、Oracle プリコンパイラ・プログラムが使用するプログラミング言語に関係なく、埋込み PL/SQL ブロックの後に配置する必要があります。キーワード END-EXEC の後には、C/C++の文終了記号「;」を付ける必要があります。

使用上の注意

Pro*C/C++では埋込みPL/SQLブロックが1つの埋込みSQL文のように扱われるため、PL/SQLブロックはプログラムでSQL文を埋め込める場所であればどこでも埋込みが可能です。Oracleプリコンパイラ・プログラムへのPL/SQLブロックの埋込みに関する詳細は、埋込みPL/SQLを参照してください。

例

このEXECUTE文をPro*C/C++プログラムに使用すると、PL/SQLブロックがプログラムに埋め込まれます。

```
EXEC SQL EXECUTE
    BEGIN
        SELECT ename, job, sal
            INTO :emp_name:ind_name, :job_title, :salary
            FROM emp
            WHERE empno = :emp_number;
        IF :emp_name:ind_name IS NULL
            THEN RAISE name_missing;
        END IF;
    END;
END-EXEC;
```

関連項目

EXECUTE IMMEDIATE (実行可能埋込みSQL)。

E.33 EXECUTE (実行可能埋込みSQL)

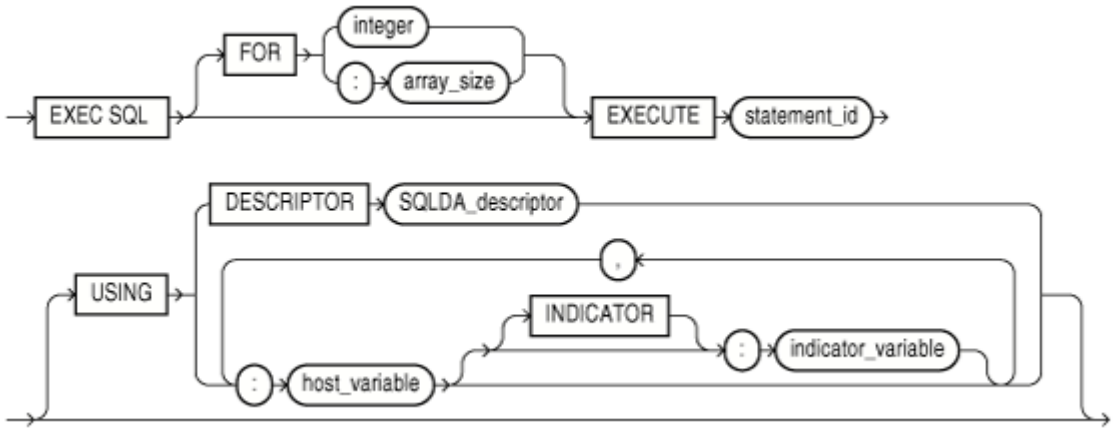
用途

Oracle動的SQLでは、埋込みSQLのPREPARE文によって準備済のDELETE文、INSERT文またはUPDATE文、あるいはPL/SQLブロックを実行します。ANSI動的SQL方法4については、EXECUTE DESCRIPTOR(実行可能埋込みSQL)を参照してください。

前提条件

埋込みSQLのPREPARE文を使用して、SQL文またはPL/SQLブロックを先に準備しておく必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
FOR *:array_size*	処理される行の数を格納するホスト変数。 処理される行数。
FOR *integer*	USING 句に配列ホスト変数が含まれる場合に文が実行される回数を制限します。この句を省略した場合、Oracle では最小の配列の各コンポーネントに対して 1 回ずつ文が実行されます。
statement_id	実行する SQL 文または PL/SQL ブロックに対応付けられているプリコンパイラ識別子。プリコンパイラ識別子を文または PL/SQL ブロックに対応付けるには、埋込み SQL の PREPARE 文を使用します。
USING DESCRIPTOR *SQLDA_descriptor*	Oracle 記述子を使用します。ANSI 記述子(INTO 句)と一緒には使用できません。
USING	オプションの標識変数を使用してホスト変数のリストを指定します(Oracle は実行する文にこれらの変数を入力変数として代入します)。ホスト変数および標識変数は、すべてスカラーか、すべて配列であることが必要です。
host_variable	ホスト変数
indicator_variable	標識変数。

使用上の注意

この文の詳細は、Oracleバージョンの[Oracle動的SQL]を参照してください。

例

この例では、Pro*C/C++プログラムでEXECUTE文を使用する方法を示しています。

```
EXEC SQL PREPARE my_statement
    FROM :my_string;
EXEC SQL EXECUTE my_statement
```

```
    USING :my_var;
```

関連項目

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

PREPARE (実行可能埋込みSQL)。

E.34 EXECUTE DESCRIPTOR (実行可能埋込みSQL)

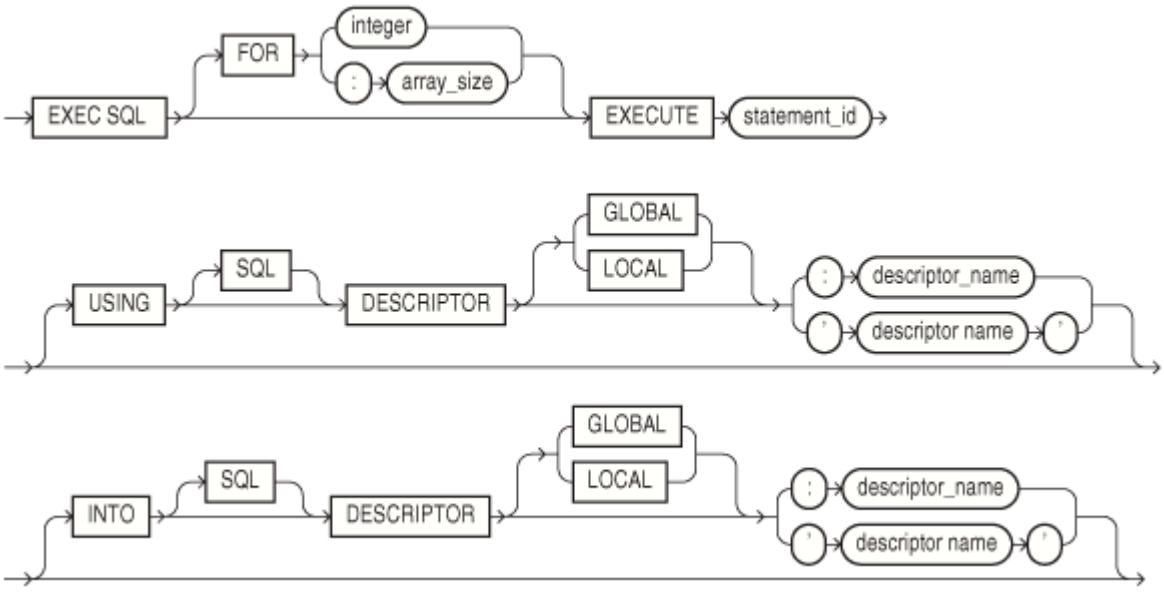
用途

ANSI SQL方法4では、埋込みSQLのPREPARE文によって準備済のDELETE文、INSERT文またはUPDATE文、あるいはPL/SQLブロックを実行します。

前提条件

埋込みSQLのPREPARE文を使用して、SQL文またはPL/SQLブロックを先に準備しておく必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
FOR *:array_size*	処理される行の数を格納するホスト変数。 処理される行数。
FOR *integer*	文が実行される回数を制限します。Oracle は最小の配列の各コンポーネントに対してこの文を 1 回ずつ実行します。
statement_id	実行する SQL 文または PL/SQL ブロックに対応付けられているプリコンパイラ識別子。プリコンパイラ識別子を文または PL/SQL ブロックに対応付けるには、埋込み SQL の PREPARE 文を使用します。

キーワードおよびパラメータ	説明
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
USING	ANSI 記述子。
descriptor_name	入力記述子名を格納するホスト変数。
'descriptor name'	入力記述子の名前。
INTO	ANSI 記述子。
descriptor_name	出力記述子の名前が含まれるホスト変数。
'descriptor name'	出力記述子の名前。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。

使用上の注意

この文の詳細は、ANSI動的SQLを参照してください。

例

ANSI動的SQL方法4では、EXECUTEのINTO句を使用して次のようにSELECTのDML RETURNING句がサポートされます。

```
EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' ;
```

関連項目

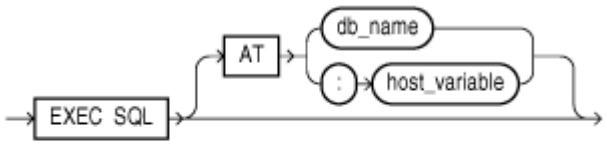
DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

PREPARE (実行可能埋込みSQL)。

E.35 EXECUTE IMMEDIATE (実行可能埋込みSQL)

用途

ホスト変数を含まないDELETE文、INSERT文またはUPDATE文、あるいはPL/SQLブロックを準備し、実行します。



前提条件

なし。

キーワードおよびパラメータ	説明
AT	SQL 文または PL/SQL ブロックをどのデータベースに対して宣言するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した db_name の値を持つホスト変数。この句を省略した場合、文またはブロックはデフォルトのデータベースに対して実行されます。
text	実行する SQL 文または PL/SQL ブロックが含まれる引用符付きのテキスト・リテラル(または引用符なしのテキスト・リテラル)。 SQL 文は、DELETE 文、INSERT 文または UPDATE 文のいずれかであることが必要です。
host_string	SQL 文を含むホスト変数。

使用上の注意

EXECUTE IMMEDIATE文を発行すると、Oracleは指定したSQL文またはPL/SQLブロックを解析してエラーをチェックし、実行します。見つかったエラーは、SQLCAのSQLCODEコンポーネントに戻されます。

この文の詳細は、Oracle動的SQLおよびANSI動的SQLを参照してください。

例

この例では、EXECUTE IMMEDIATE文の使用方法を示します。

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = 9460' ;
```

関連項目

EXECUTE (実行可能埋込みSQL)。

PREPARE (実行可能埋込みSQL)。

E.36 FETCH (実行可能埋込みSQL)

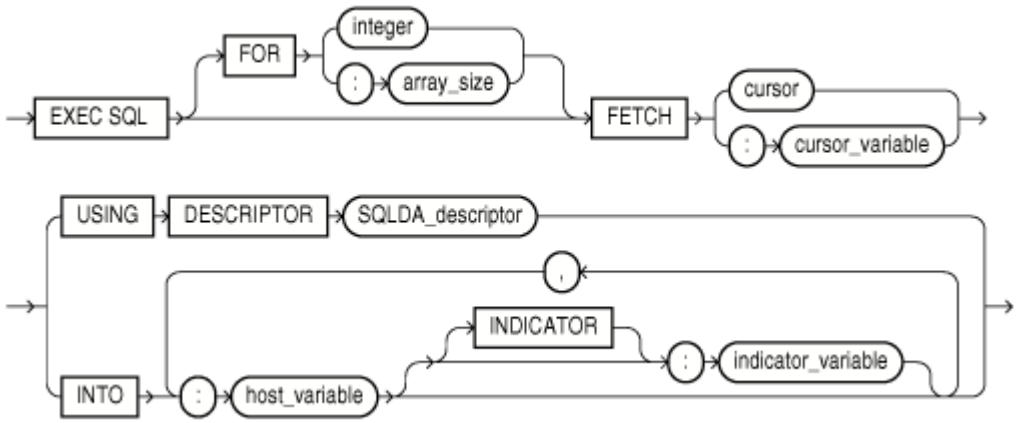
用途

Oracle動的SQLでは、選択リストの値がホスト変数に割り当てられ、問合せが戻した1行または複数の行が取り出されます。ANSI動的SQL方法4については、FETCH DESCRIPTOR(実行可能埋込みSQL)を参照してください。

前提条件

まず、OPEN文を使用してカーソルを先にオープンする必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
FOR :array_size	処理される行の数を格納するホスト変数。 処理される行数。
FOR integer	配列ホスト変数を使用する場合にフェッチする行数を制限します。この句を省略した場合、Oracle は最小の配列を満たすのに十分な数の行をフェッチします。
cursor	DECLARE CURSOR 文を使用して宣言したカーソル。FETCH 文は、カーソルに対応付けられた問合せが選択した行のうちの 1 行を戻します。
cursor_variable	ALLOCATE 文を使用して割り当てたカーソル変数。FETCH 文は、カーソル変数に対応付けられた問合せが選択した行のうちの 1 行を戻します。
INTO	データをフェッチするホスト変数およびオプションの標識変数のリストを指定します。これらのホスト変数および標識変数は、プログラム内で宣言されている必要があります。
host_variable	データを受け取るホスト変数。
i*ndicator_variables*	ホスト標識変数。
USING *SQLDA_variable*	DESCRIBE 文を使用して事前に参照している Oracle 記述子を指定します。この句は、動的埋込み SQL 方法 4 以外では使用しないでください。カーソル変数を使用している場合は、USING 句は適用されません。

使用上の注意

FETCH文はアクティブ・セットの行を読み取り、結果が含まれる出力変数の名前を示します。対応付けられたホスト変数がNULLの場合、インジケータの値は-1に設定されます。また、カーソルに対する最初のFETCH文は、必要に応じてアクティブ・セットの行をソートします。

出力ホスト変数のサイズは取り出された行数を示し、FOR句は値を示します。データを取得するホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。スカラーの場合は、Oracleは1行のみフェッチします。配列の場合、

Oracleは配列を満たすのに十分な数の行をフェッチします。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracleがフェッチする行数は、次の値のうち小さい方です。

- 最小の配列のサイズ
- オプションのFOR句の:*array_size*の値

フェッチする行数は、実際に問合せを満たす行数によってさらに限定できます。

FETCH文が、問合せで戻された行すべてを取得しなかった場合、カーソルは戻された次の行に配置されます。問合せで戻された最後の行を取得すると、次のFETCH文を実行すると、SQLCAのSQLCODE要素にエラー・コードが戻されることになります。

FETCH文にはAT句は含まれません。カーソルによってアクセスされるデータベースは、DECLARE CURSOR文で指定する必要があります。

FETCH文では、アクティブ・セット内を前方向にのみ進めます。すでにフェッチした行に戻る場合は、カーソルを再オープンして各行を順番に取り出す必要があります。アクティブ・セットを変更するには、新しい値をカーソルの問合せの入力ホスト変数に割り当て、カーソルを再オープンします。

Oracleの記述子の詳細は、FETCH文を参照してください。

例

この例では、FETCH文を示します。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT job, sal FROM emp WHERE deptno = 30;
EXEC SQL OPEN emp_cursor;
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
for(;;)
    {
    EXEC SQL FETCH emp_cursor INTO :job_title1, :salary1;
    ...
    }
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE CURSOR (埋込みSQLディレクティブ)。

OPEN (実行可能埋込みSQL) 。

PREPARE (実行可能埋込みSQL)。

E.37 FETCH DESCRIPTOR (実行可能埋込みSQL)

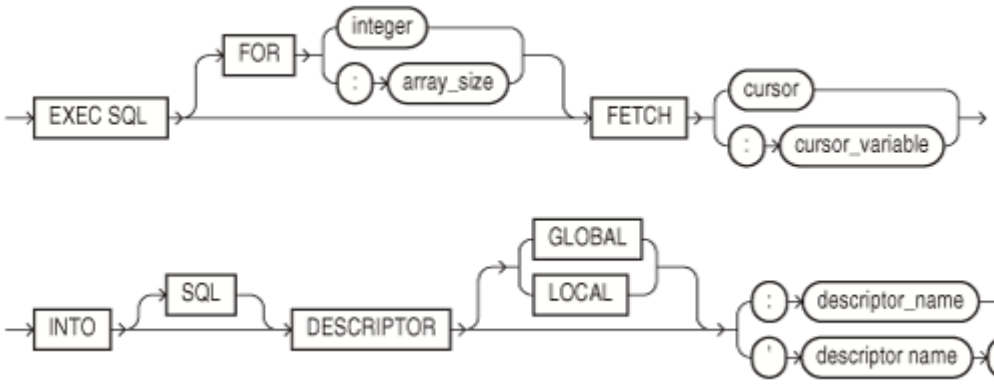
用途

選択リストの値をホスト変数に割り当てて、問合せが戻した1つまたは複数の行を取り出します。ANSI動的SQL方法4で使用します。

前提条件

OPEN文を使用してカーソルを先にオープンしておく必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
array_size	処理される行の数を格納するホスト変数。 処理される行数。
integer	配列ホスト変数を使用する場合にフェッチする行数を制限します。この句を省略した場合、Oracle は最小の配列を満たすのに十分な数の行をフェッチします。
cursor	DECLARE CURSOR 文を使用して宣言したカーソル。FETCH 文は、カーソルに対応付けられた問合せが選択した行のうちの 1 行を戻します。
cursor_variable	ALLOCATE 文を使用して割り当てたカーソル変数。FETCH 文は、カーソル変数に対応付けられた問合せが選択した行のうちの 1 行を戻します。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
INTO	データをフェッチするホスト変数およびオプションの標識変数のリストを指定します。これらのホスト変数および標識変数は、プログラム内で宣言されている必要があります。
descriptor name'	出力 ANSI 記述子の名前。
:descriptor_name	出力記述子名を格納するホスト変数。

使用上の注意

出力ホスト変数のサイズは取り出された行数を示し、FOR句は値を示します。データを取得するホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。スカラーの場合は、Oracleは1行のみフェッチします。配列の場合、Oracleは配列を満たすのに十分な数の行をフェッチします。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracleがフェッチする行数は、次の値のうち小さい方です。

- 最小の配列のサイズ
- オプションのFOR句の:*array_size*の値

フェッチする行数は、実際に問合せを満たす行数によってさらに限定できます。

FETCH文が、問合せで戻された行すべてを取得しなかった場合、カーソルは戻された次の行に配置されます。問合せで戻された最後の行を取得すると、次のFETCH文を実行すると、SQLCAのSQLCODE要素にエラー・コードが戻されることになります。

FETCH文にはAT句は含まれません。カーソルによってアクセスされるデータベースは、DECLARE CURSOR文で指定する必要があります。

FETCH文では、アクティブ・セット内を前方向にのみ進めます。すでにフェッチした行に戻る場合は、カーソルを再オープンして各行を順番に取り出す必要があります。アクティブ・セットを変更するには、新しい値をカーソルの問合せの入力ホスト変数に割り当て、カーソルを再オープンします。

ANSI SQL方法4アプリケーション用にDYNAMIC=ANSIプリコンパイラ・オプションを使用してください。ANSI SQL方法4アプリケーションの詳細は、FETCHを参照してください。

例

```
...
EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor' ;
...
EXEC SQL PREPARE S FROM :dyn_statement ;
EXEC SQL DECLARE mycursor CURSOR FOR S ;
...
EXEC SQL FETCH mycursor INTO DESCRIPTOR 'output_descriptor' ;
...
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE CURSOR (埋込みSQLディレクティブ)。

OPEN DESCRIPTOR (実行可能埋込みSQL)

PREPARE (実行可能埋込みSQL)。

E.38 FREE (実行可能埋込みSQL拡張機能)

用途

オブジェクト・キャッシュ内のメモリーを解放します。

前提条件

メモリーがすでに割り当てられている必要があります。

アクティブなデータベース接続が存在している必要があります。

構文

キーワードおよびパラメータ

キーワードおよびパラメータ	説明

キーワードおよびパラメータ	説明
dbname	CONNECT 文で先に設定されるデータベース接続名を含む NULL 終了文字列。これが省略される、または空の文字列である場合は、デフォルトのデータベース接続とみなされます。
host_ptr	事前に ALLOCATE で割り当てられていたホスト変数ポインタ。
ind_ptr	インジケータ・ポインタ。

使用上の注意

接続が切り離されると、オブジェクト・キャッシュに割り当てられているメモリーはすべて自動的に解放されます。詳細は、FREEを参照してください。

例

```
EXEC SQL FREE :ptr ;
```

関連項目

ALLOCATE (実行可能埋込みSQL拡張機能)。

「CACHE FREE ALL(実行可能埋込みSQL拡張機能)」

E.39 GET DESCRIPTOR (実行可能埋込みSQL)

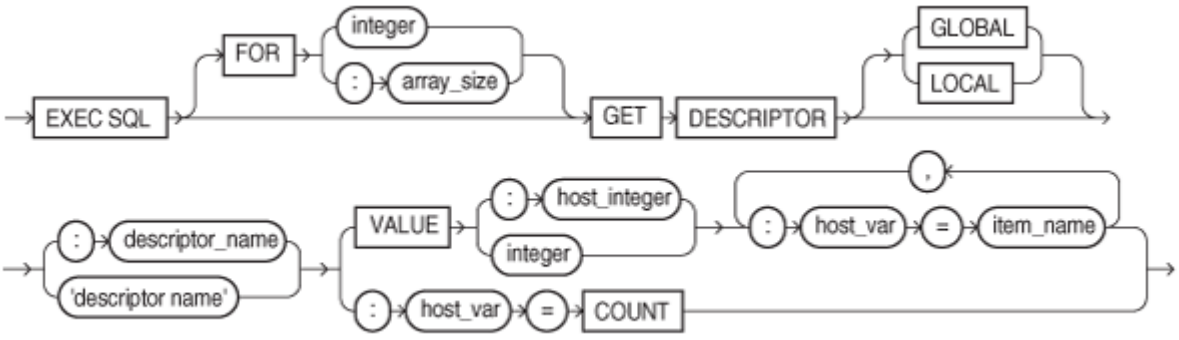
用途

SQL記述子領域のホスト変数の情報を取得します。

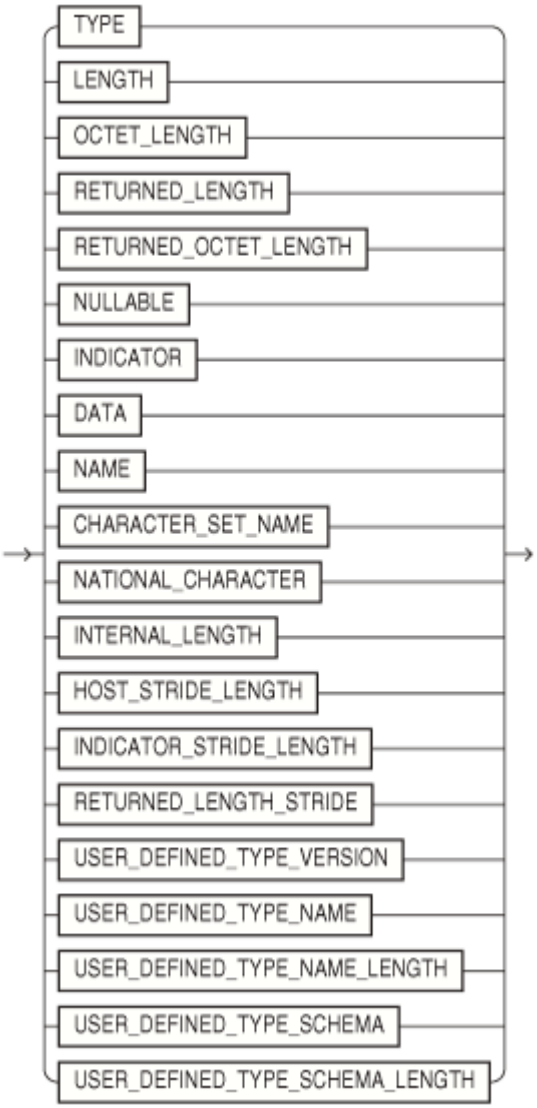
前提条件

値構文でのみ使用します。

構文



item_nameのみ次の中から選択できます。



キーワードおよびパラメータ

キーワードおよびパラメータ	**説明**
array_size	処理される行の数を格納するホスト変数。
integer	処理される行数。
descriptor_name *'descriptor name'*	割り当てられた ANSI 記述子名を格納するホスト変数。 割り当てられた ANSI 記述子の名前。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
host_var = COUNT	入力および出力変数の合計数を格納するホスト変数。

キーワードおよびパラメータ	説明
integer	入力および出力変数の合計数。
VALUE :*host_integer*	参照される入力または出力変数の位置を格納するホスト変数。
VALUE *integer*	参照される入力または出力変数の位置。
host_var	項目の値を受け取るホスト変数。
item_name	*item_name* に関しては、表 10-4 および表 10-5 の「記述子項目名」列を参照してください。

使用上の注意

DYNAMIC=ANSIプリコンパイラ・オプションを使用してください。配列サイズ句は、DATA、RETURNED_LENGTHおよびINDICATORの項目名で使用できます。「GET DESCRIPTOR」を参照してください。

例

```
EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc_num_vars = COUNT ;
```

関連項目

ALLOCATE DESCRIPTOR (実行可能埋込みSQL)。

DEALLOCATE DESCRIPTOR (埋込みSQL文)。

SET DESCRIPTOR (実行可能埋込みSQL)。

E.40 INSERT (実行可能埋込みSQL)

用途

表またはビューの実表に行を追加します。

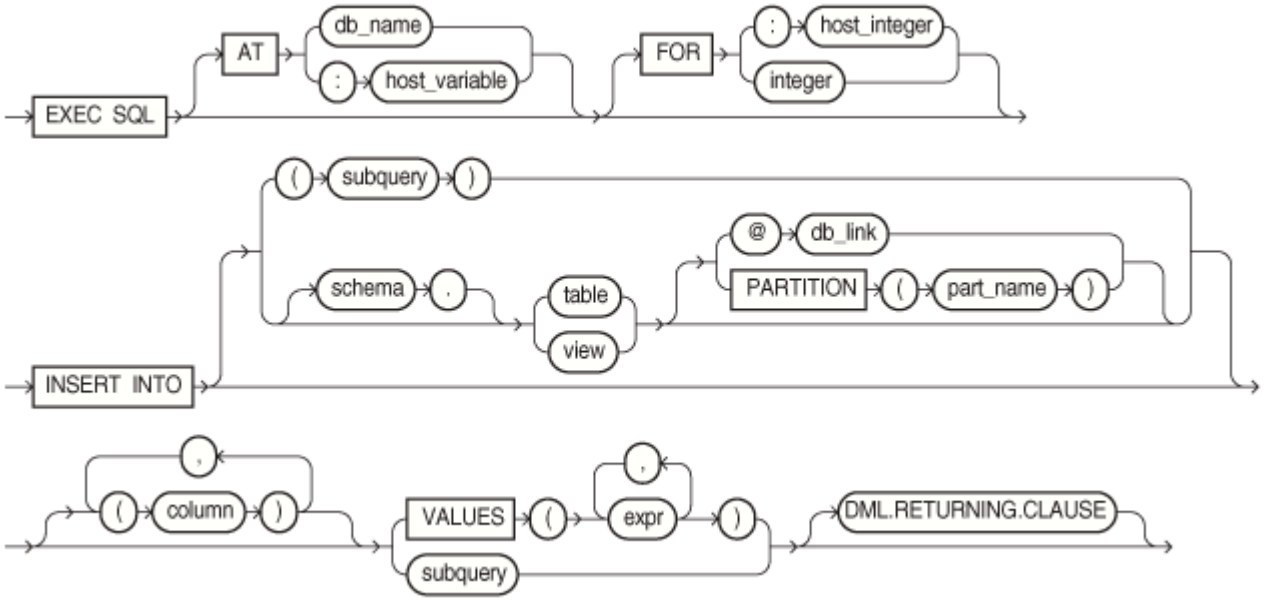
前提条件

表に行を挿入するには、その表が自分のスキーマ内にあるか、またはその表に対してINSERTの権限を持っている必要があります。

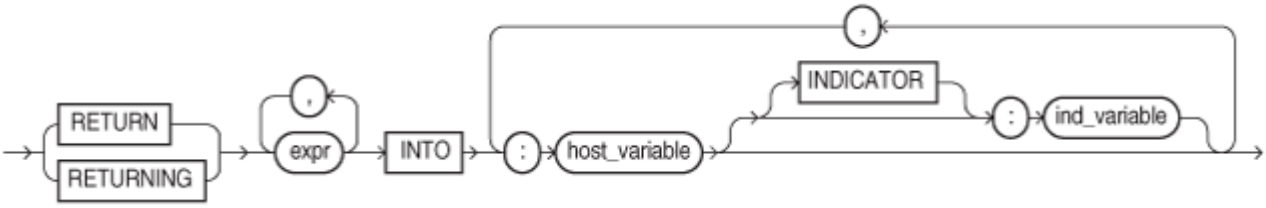
ビューの実表に行を挿入するには、ビューが属するスキーマの所有者が、その実表に対してINSERTの権限を持っている必要があります。また、ビューがユーザーのスキーマ以外のスキーマ内にある場合は、ビューに対してINSERTの権限を持っている必要があります。

INSERT ANY TABLEシステム権限を使用すると、どの表またはビューの実表にも行を挿入できます。

構文



DML RETURNING句は次のとおりです。



キーワードおよびパラメータ

キーワードおよびパラメータ	**説明**
AT	INSERT 文をどのデータベースについて実行するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 ***host_variable*:** 値が db_name のホスト変数。この句を省略した場合、INSERT 文はデフォルトのデータベースに対して実行されます。
FOR :host_integer 整数	VALUES 句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。 この句を省略した場合、Oracle は最小の配列の各コンポーネントにつき 1 回ずつ文が実行されます。
schema	表またはビューを含むスキーマ。schema を省略した場合、Oracle では表またはビューが自分のスキーマ内にあるとみなされます。
table *view*	行を挿入する表の名前。 ビューを指定すると、Oracle ではそのビューの実表に行が挿入されます。

キーワードおよびパラメータ	説明
db_link	表またはビューがあるリモート・データベースへのデータベース・リンクの完全名または部分名。 Oracle を分散オプションで使用している場合にのみ、リモートの表またはビューに行を挿入できます。 dblink を省略した場合、Oracle では表またはビューがローカル・データベース内にあるとみなされます。
part_name	表のパーティションの名前。
column	このリストから表の列を削除する場合、挿入された行の列値は、表の作成時に指定した列のデフォルト値となります。列のリストを完全に省略した場合は、VALUES 句または問合せによって、表のすべての列の値を指定する必要があります。
VALUES	表またはビューに挿入される値の行を指定します。構文については、『*Oracle Database SQL 言語リファレンス*』を参照してください。式には、ホスト変数とオプションの標識変数を使用できます。VALUES 句では、列のリストの各列に式を指定する必要があります。
subquery	表に挿入される行を戻す副問合せ。この副問合せの選択リストの列数は、INSERT 文の列のリストの列数と同じであることが必要です。副問合せの構文記述については、SELECT を参照してください。
DML RETURNING 句	DML RETURNING 句を参照してください。

使用上の注意

WHERE句内のホスト変数は、すべてスカラーか、すべて配列である必要があります。変数がスカラーの場合、OracleはINSERT文を1回実行します。変数が配列の場合、OracleはINSERT文を各配列コンポーネント・セットについて1回ずつ実行して、1行ずつ挿入します。

WHERE句の配列ホスト変数は、サイズが異なっていてもかまいません。この場合、Oracleが文を実行する回数は、次のうちの小さい方の値によって決定します。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値。

この文の詳細は、INSERT文を参照してください。

例I

この例では、埋込みSQL INSERT文の使用方法を示しています。

```
EXEC SQL
     INSERT INTO emp (ename, empno, sal)
     VALUES (:ename, :empno, :sal) ;
```

例II

この例では、副問合せを使用した埋込みSQLのINSERT文を示します。

```
EXEC SQL
   INSERT INTO new_emp (ename, empno, sal)
   SELECT ename, empno, sal FROM emp
   WHERE deptno = :deptno ;
```

関連項目

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

E.41 LOB APPEND (実行可能埋込みSQL拡張機能)

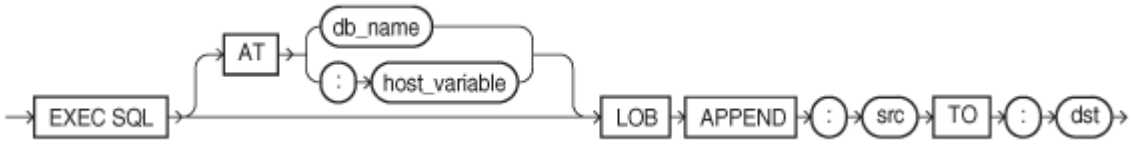
用途

LOBを別のLOBの最後に追加します。

前提条件

LOBバッファリングは使用可能にしないでください。宛先LOBが初期化されている必要があります。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、APPENDを参照してください。

関連項目

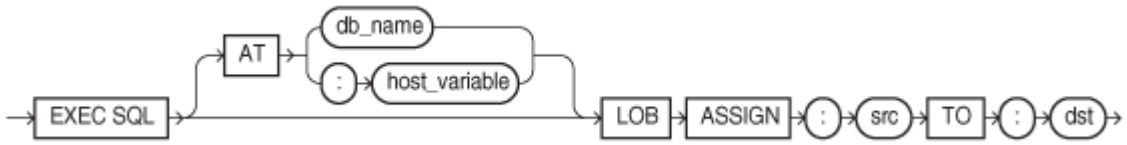
他のLOB文を参照してください。

E.42 LOB ASSIGN (実行可能埋込みSQL拡張機能)

用途

LOBまたはBFILEロケータを別のロケータに割り当てます。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、ASSIGNを参照してください。

関連項目

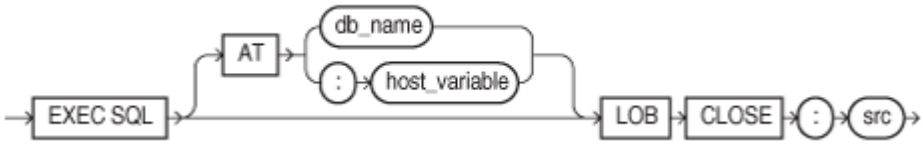
他のLOB文を参照してください。

E.43 LOB CLOSE (実行可能埋込みSQL拡張機能)

用途

オープンされているLOBまたはBFILEをクローズします。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、CLOSE (LOB用)を参照してください。

関連項目

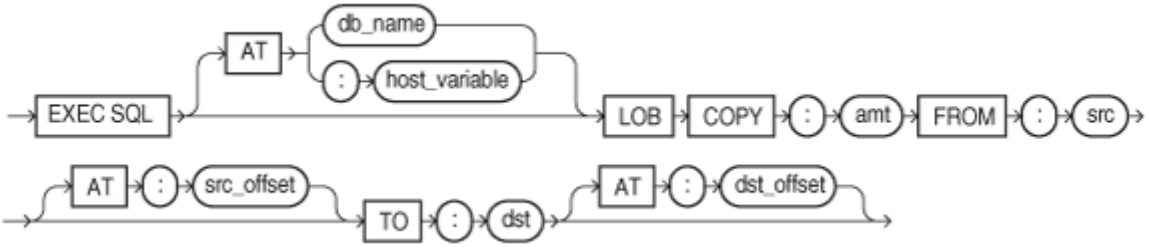
他のLOB文を参照してください。

E.44 LOB COPY (実行可能埋込みSQL拡張機能)

用途

LOB値の全部または一部を別のLOBにコピーします。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、COPYを参照してください。

関連項目

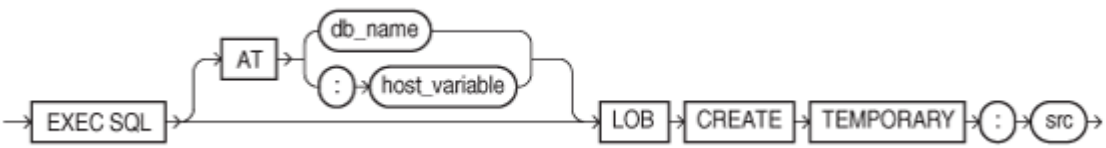
他のLOB文を参照してください。

E.45 LOB CREATE TEMPORARY (実行可能埋込みSQL拡張機能)

用途

一時LOBを作成します。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、CREATE TEMPORARYを参照してください。

関連項目

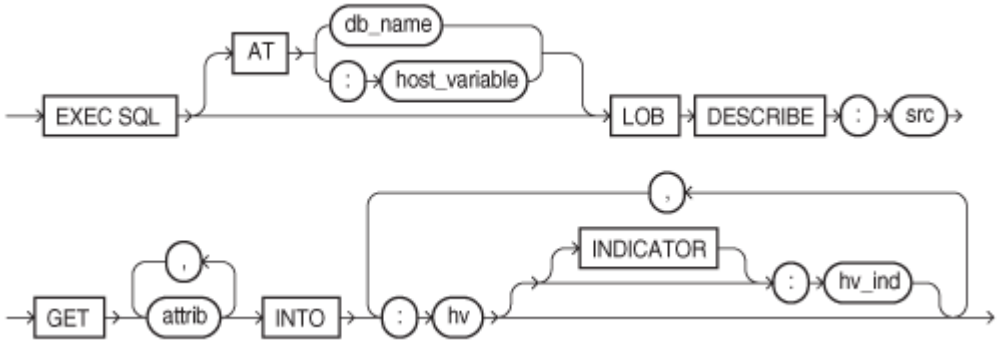
他のLOB文を参照してください。

E.46 LOB DESCRIBE (実行可能埋込みSQL拡張機能)

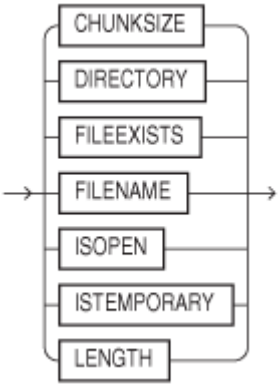
用途

LOBから属性を取り出します。

構文



attribは次のとおりです。



使用上の注意

使用方法、キーワード、パラメータおよび例は、DESCRIBEを参照してください。

関連項目

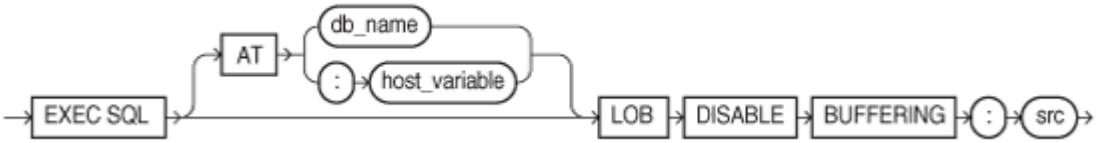
他のLOB文を参照してください。

E.47 LOB DISABLE BUFFERING (実行可能埋込みSQL拡張機能)

用途

LOBバッファリングを使用禁止にします。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、DISABLE BUFFERINGを参照してください。

関連項目

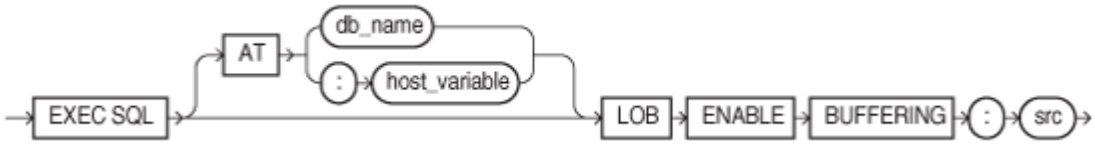
他のLOB文を参照してください。

E.48 LOB ENABLE BUFFERING (実行可能埋込みSQL拡張機能)

用途

LOBバッファリングを有効にします。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、ENABLE BUFFERINGを参照してください

関連項目

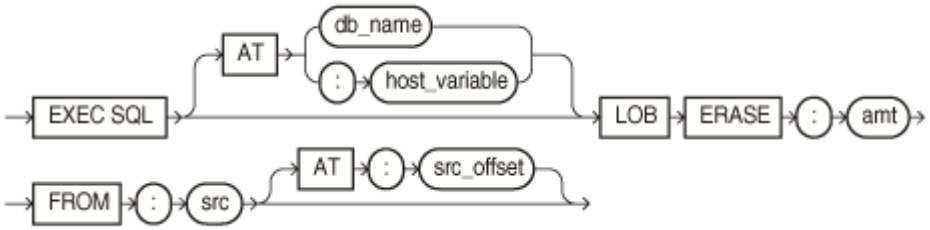
他のLOB文を参照してください。

E.49 LOB ERASE (実行可能埋込みSQL拡張機能)

用途

指定されたオフセットから始まる指定された量のLOBデータを消去します。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、ERASEを参照してください。

関連項目

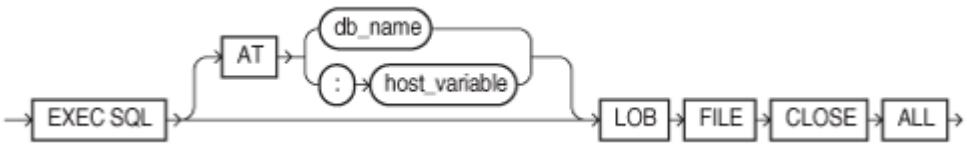
他のLOB文を参照してください。

E.50 LOB FILE CLOSE ALL (実行可能埋込みSQL拡張機能)

用途

カレント・セッションでオープンしているすべてのBFILEをクローズします。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、FILE CLOSE ALLを参照してください。

関連項目

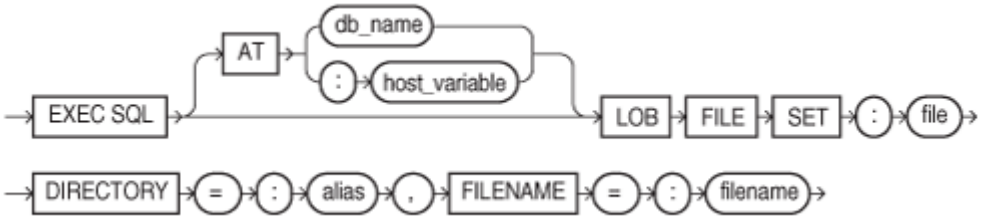
他のLOB文を参照してください。

E.51 LOB FILE SET (実行可能埋込みSQL拡張機能)

用途

BFILEロケータのDIRECTORYおよびFILENAMEを設定します。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、FILE SETを参照してください。

関連項目

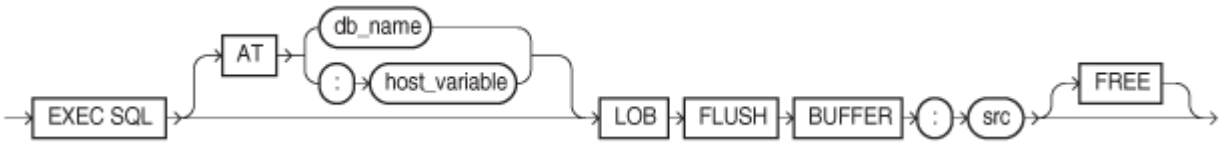
他のLOB文を参照してください。

E.52 LOB FLUSH BUFFER (実行可能埋込みSQL拡張機能)

用途

LOBのバッファをデータベース・サーバーに書き込みます。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、FLUSH BUFFERを参照してください。

関連項目

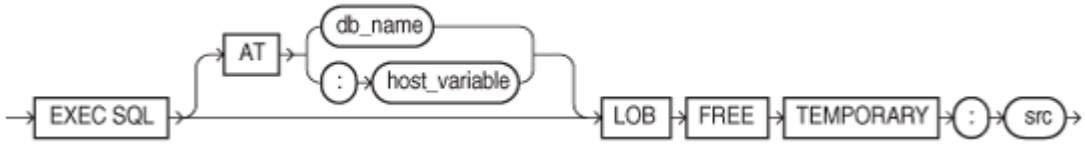
他のLOB文を参照してください。

E.53 LOB FREE TEMPORARY (実行可能埋込みSQL拡張機能)

用途

LOBロケータ用に一時領域を解放します。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、FREE TEMPORARYを参照してください。

関連項目

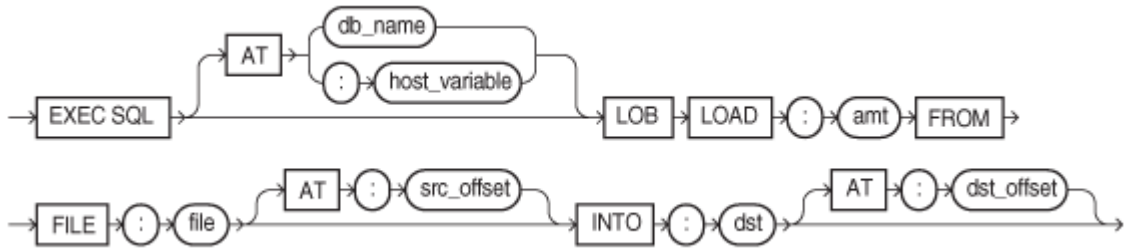
他のLOB文を参照してください。

E.54 LOB LOAD (実行可能埋込みSQL拡張機能)

用途

BFILEの全部または一部を、内部LOBにコピーします。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、LOAD FROM FILEを参照してください。

関連項目

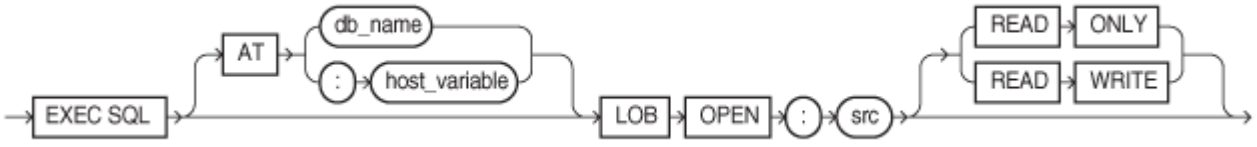
他のLOB文を参照してください。

E.55 LOB OPEN (実行可能埋込みSQL拡張機能)

用途

読取りまたは読取り/書込みを行うLOBまたはBFILEをオープンします。

構文



使用上の注意

使用上の注意、キーワード、パラメータおよび例は、OPEN (LOB用)を参照してください。

関連項目

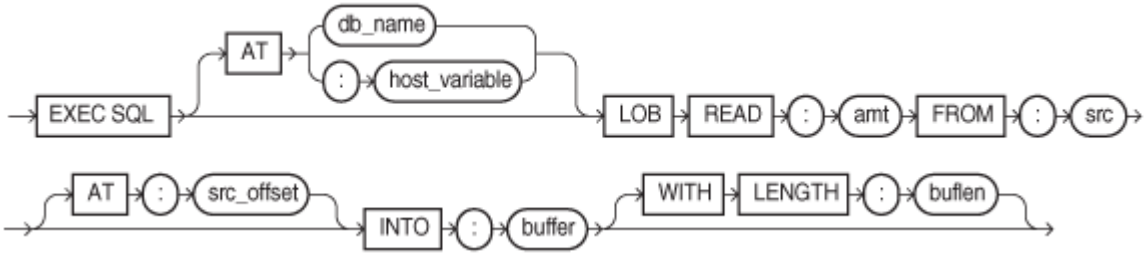
他のLOB文を参照してください。

E.56 LOB READ (実行可能埋込みSQL拡張機能)

用途

LOBまたはBFILEの一部をバッファに読み込みます。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、READを参照してください。

関連項目

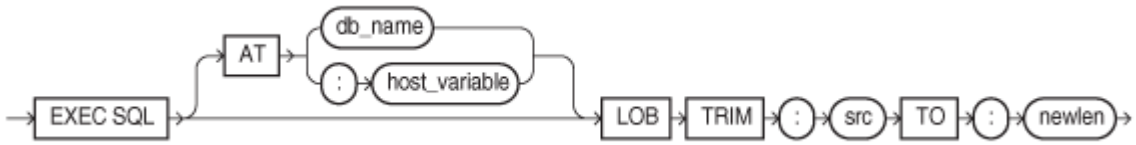
他のLOB文を参照してください。

E.57 LOB TRIM (実行可能埋込みSQL拡張機能)

用途

LOB値を切り捨てます。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、TRIMを参照してください。

関連項目

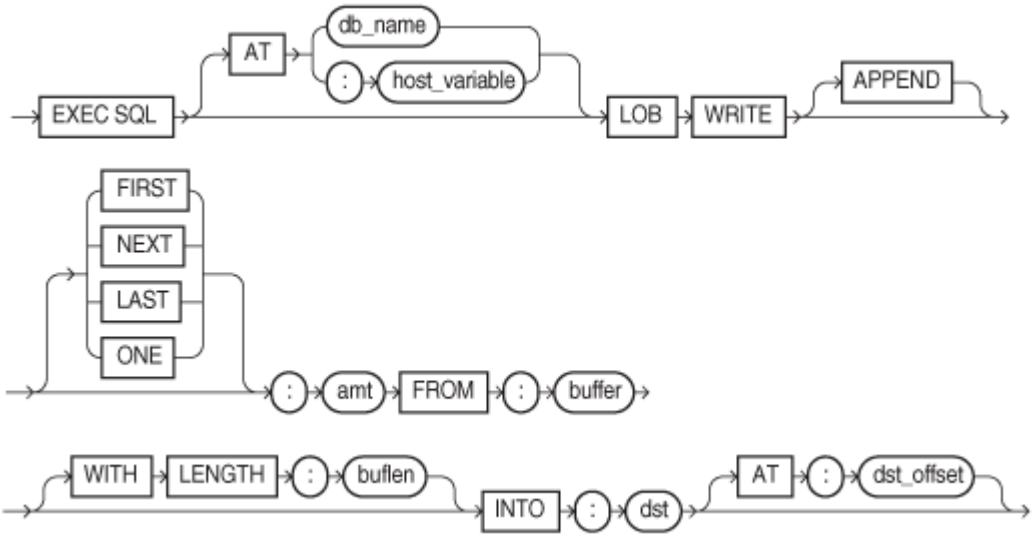
他のLOB文を参照してください。

E.58 LOB WRITE (実行可能埋込みSQL拡張機能)

用途

バッファの内容をLOBに書き込みます。

構文



使用上の注意

使用方法、キーワード、パラメータおよび例は、WRITEを参照してください。

関連項目

他のLOB文を参照してください。

E.59 OBJECT CREATE (実行可能埋込みSQL拡張機能)

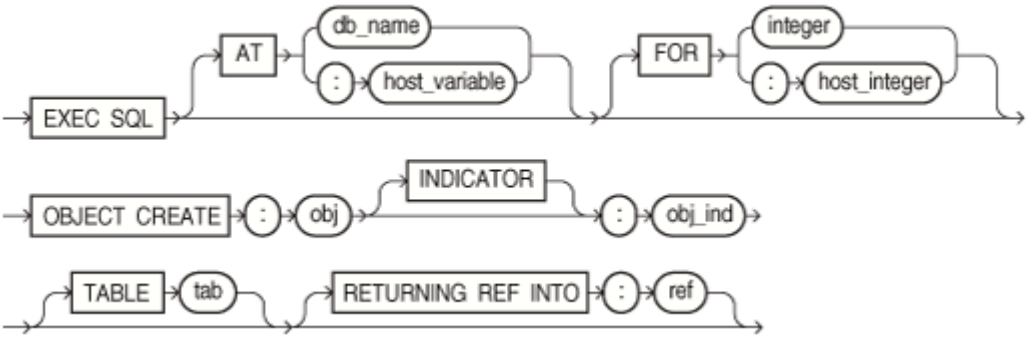
用途

オブジェクト・キャッシュ内に参照可能なオブジェクトを作成します。

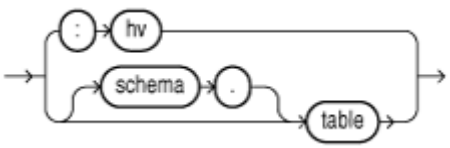
前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



tabは次のとおりです。



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT CREATEを参照してください。

例

```
person *pers_p;
```

```
person_ind *pers_ind;
person_ref *pers_ref;
...
EXEC SQL OBJECT CREATE :pers_p:pers_ind TABLE PERSON_TAB
          RETURNING REF INTO :pers_ref ;
```

関連項目

この付録における他のすべてのOBJECT文を参照してください。

E.60 OBJECT DELETE (実行可能埋込みSQL拡張機能)

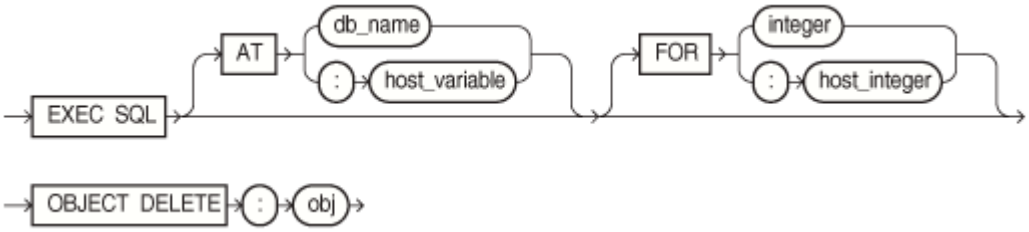
用途

オブジェクト・キャッシュ内の永続オブジェクトまたはオブジェクトの配列を削除済としてマークします。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT DELETEを参照してください。

例

```
customer *cust_p;
...
EXEC SQL OBJECT DELETE :cust_p;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。永続オブジェクトの場合、この文はオブジェクト・キャッシュ内のオブジェクトまたはオブジェクトの配列を削除済としてマーク設定します。

E.61 OBJECT DEREF (実行可能埋込みSQL拡張機能)

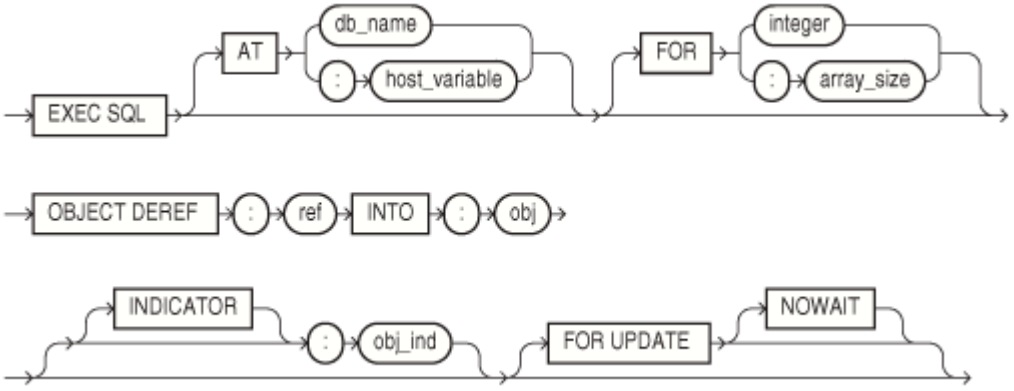
用途

オブジェクト・キャッシュ内にオブジェクトまたはオブジェクトの配列を保持します。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT DEREFを参照してください。

例

```
person *pers_p;
person_ref *pers_ref;
...
/* Pin the person REF, returning a pointer to the person object */
EXEC SQL OBJECT DEREF :pers_ref INTO :pers_p;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。ALLOCATE(実行可能埋込みSQL拡張機能)を参照してください。

E.62 OBJECT FLUSH (実行可能埋込みSQL拡張機能)

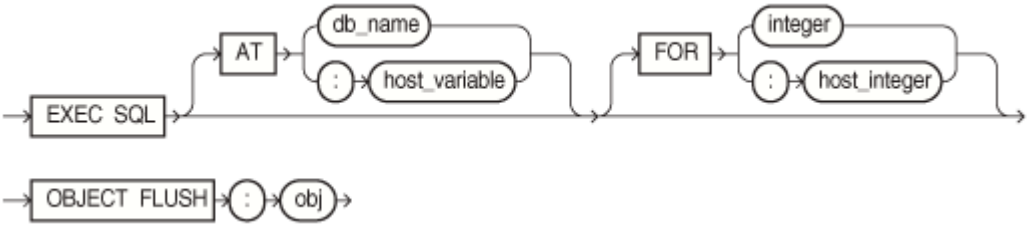
用途

更新済、削除済または作成済としてマーク設定された永続オブジェクトを、サーバーに反映します。この処理はフラッシュと呼ばれます。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT FLUSHを参照してください。

例

```
person *pers_p;
...
EXEC SQL OBJECT DELETE :pers_p;
/* Flush the changes, effectively deleting the person object */
```

```
EXEC SQL OBJECT FLUSH :pers_p;
/* Finally, free all object cache memory and logoff */
EXEC SQL OBJECT CACHE FREE ALL;
EXEC SQL COMMIT WORK RELEASE;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。

E.63 OBJECT GET (実行可能埋込みSQL拡張機能)

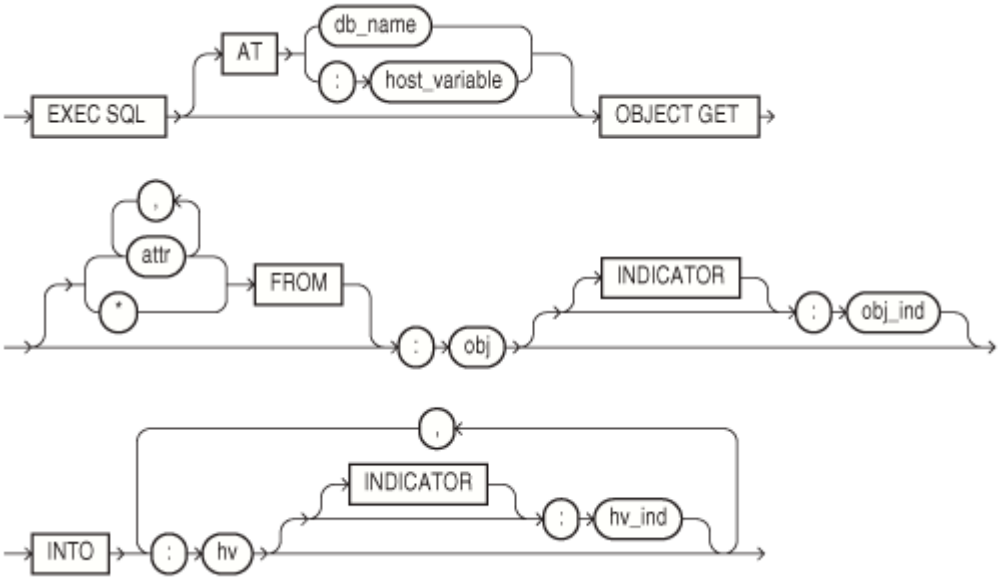
用途

オブジェクト型の属性をネイティブなC言語のデータ型に変換します。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT GETを参照してください。

例

```
person *pers_p;
struct { char lname[21], fname[21]; int age; } pers;
...
/* Convert object types to native C types */
EXEC SQL OBJECT GET lastname, firstname, age FROM :pers_p INTO :pers;
printf("Last Name: %s¥nFirstName: %s¥nAge: %d¥n",
        pers.lname, pers.fname, pers.age );
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。

E.64 OBJECT RELEASE (実行可能埋込みSQL拡張機能)

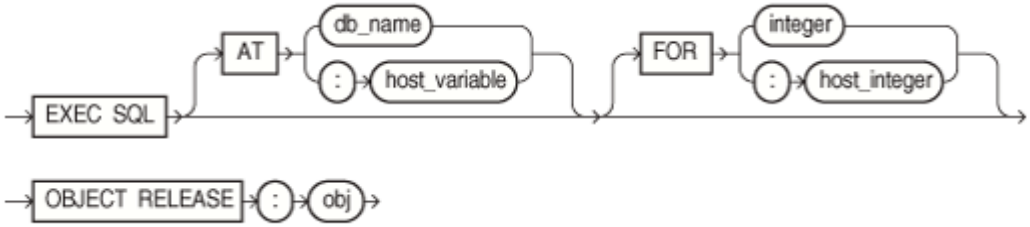
用途

オブジェクト・キャッシュ内のオブジェクトを解放します。オブジェクトが確保されず、更新もされなければ、暗黙的な解放の対象になります。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT RELEASEを参照してください。

例

```
person *pers_p;
...
EXEC SQL OBJECT RELEASE :pers_p;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。

E.65 OBJECT SET (実行可能埋込みSQL拡張機能)

用途

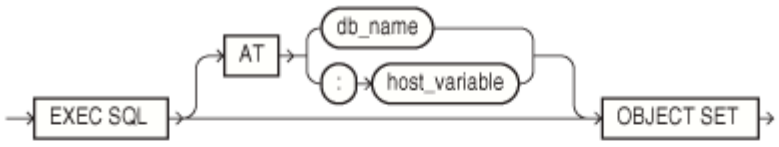
永続オブジェクトの属性を更新し、オブジェクトのフラッシュ時、またはキャッシュのフラッシュ時にサーバーへの書込み対象にします。

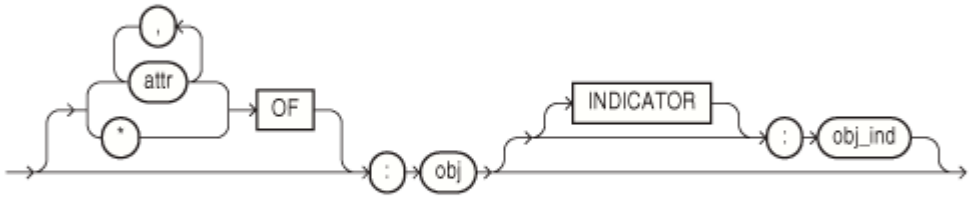
一時オブジェクトの属性を更新します。

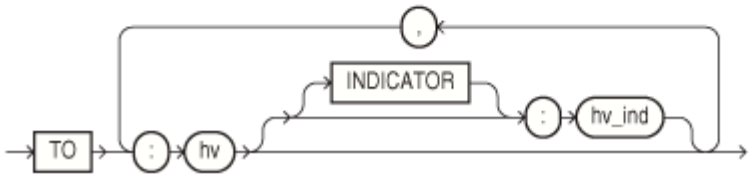
前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文







使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT FLUSHを参照してください。

例

```
person *pers_p;
struct {int num; char street[61], city[31], state[3], zip[11];} addr1;
...
addr1.num = 500;
strcpy((char *)addr1.street , (char *)"Oracle Parkway");
strcpy((char *)addr1.city,    (char *)"Redwood Shores");
strcpy((char *)addr1.state,   (char *)"CA");
strcpy((char *)addr1.zip,     (char *)"94065");

/* Convert native C types to object types */
EXEC SQL OBJECT SET :pers_p->addr TO :addr1;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。

E.66 OBJECT UPDATE (実行可能埋込みSQL拡張機能)

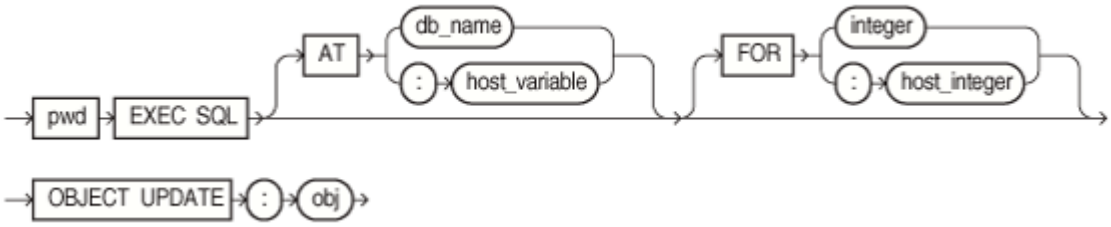
用途

オブジェクト・キャッシュ内の永続オブジェクトまたはオブジェクトの配列を更新済としてマーク設定します。

前提条件

プリコンパイラ・オプションOBJECTSをYESに設定する必要があります。INTYPEオプションでは、OTTによって生成される型ファイルを指定する必要があります。OTTによって生成されるヘッダー・ファイルをプログラムにインクルードしてください。

構文



使用上の注意

使用上の注意、キーワードおよびパラメータは、OBJECT UPDATEを参照してください。

例

```
person *pers_p;
...
/* Mark as updated */
EXEC SQL OBJECT UPDATE :pers_p;
```

関連項目

この付録にある他のすべてのOBJECT文を参照してください。

E.67 OPEN (実行可能埋込みSQL)

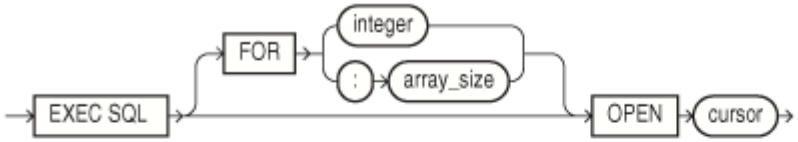
用途

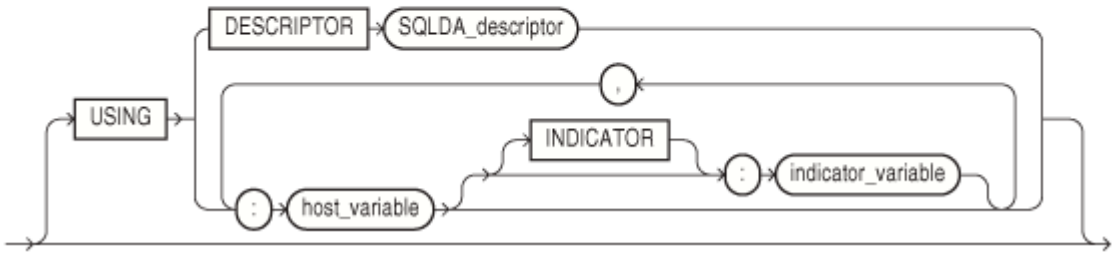
対応付けられた問合せを評価し、USING句が示すホスト変数名を問合せのWHERE句に代入して、カーソルをオープンします。ANSI動的SQL方法4バージョンについては、OPEN DESCRIPTOR(実行可能埋込みSQL)を参照してください。

前提条件

カーソルは、オープンする前に埋込みSQLのDECLARE CURSOR文を使用して宣言しておく必要があります。

構文





キーワードおよびパラメータ

キーワードおよびパラメータ	説明
array_size	処理される行の数を格納するホスト変数。
integer	処理される行数。
cursor	オープンする(すでに宣言されている)カーソル。
host_variable	カーソルに対応付けられた文に代入するホスト変数を指定します。 ANSI 記述子(INTO 句)と一緒には使用できません。

キーワードおよびパラメータ	説明
DESCRIPTOR *SQLDA_descriptor*	対応付けられた問合せの WHERE 句に代入するホスト変数を表す Oracle 記述子を指定します。記述子は、DESCRIBE 文を使用して事前に初期化されている必要があります。代入は、位置に基づきます。この文で指定するホスト変数名は、対応付けられた問合せの変数名と異なってもかまいません。 ANSI 記述子(INTO 句)と一緒には使用できません。

使用上の注意

OPEN文は、行のアクティブ・セットを定義し、アクティブ・セットの最初の行の直前でカーソルを初期化します。OPEN時のホスト変数の値が文に代入されます。この文は、実際には行を取り出しません。行はFETCH文を使用して取り出されます。

カーソルを一度オープンすると、入力ホスト変数はカーソルを再オープンするまで再テストされません。入力ホスト変数およびアクティブ・セットを変更するには、カーソルを再オープンする必要があります。

プログラム内のすべてのカーソルは、プログラムを開始する場合またはCLOSE文を使用してカーソルを明示的にクローズした後にクローズ状態になります。

カーソルは事前にクローズしなくても、再オープンできます。この文の詳細は、INSERT文を参照してください。

例

この例では、Pro*C/C++プログラムでOPEN文を使用する方法を示しています。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, job, sal
    FROM emp
    WHERE deptno = :deptno;
EXEC SQL OPEN emp_cursor;
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE STATEMENT (埋込みSQLディレクティブ) 。

FETCH (実行可能埋込みSQL) 。

PREPARE (実行可能埋込みSQL)。

E.68 OPEN DESCRIPTOR (実行可能埋込みSQL)

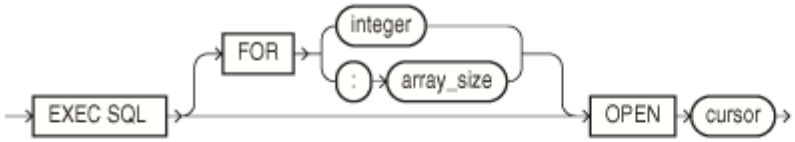
用途

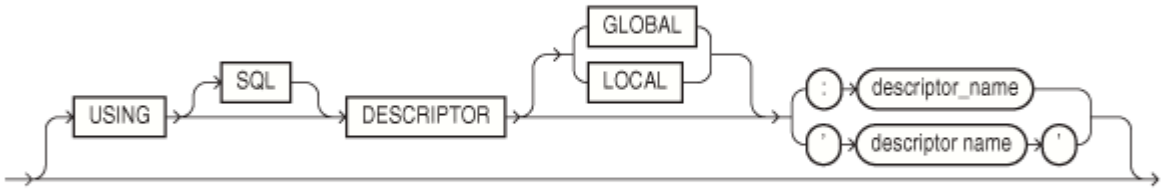
対応付けられた問合せを評価し、USING句が示す入力ホスト変数名を問合せのWHERE句に代入して、カーソル(ANSI動的SQL方法4用)をオープンします。INTO句は、出力記述子を示します。

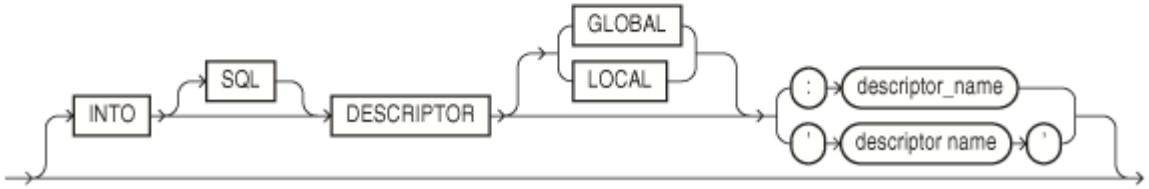
前提条件

カーソルは、オープンする前に埋込みSQLのDECLARE CURSOR文を使用して宣言しておく必要があります。

構文







キーワードおよびパラメータ

キーワードおよびパラメータ	**説明**
array_size	処理される行の数を格納するホスト変数。
integer	処理される行数。
cursor	オープンする(すでに宣言されている)カーソル。
USING DESCRIPTOR	ANSI 入力記述子を指定します。
descriptor_name	ANSI 記述子の名前が含まれるホスト変数。
'descriptor name'	ANSI 記述子の名前。
INTO DESCRIPTOR	ANSI 出力記述子を指定します。
descriptor_name	ANSI 記述子の名前が含まれるホスト変数。
'descriptor name'	ANSI 記述子の名前。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。

使用上の注意

プリコンパイラのオプションDYNAMICにANSIを設定します。

OPEN文は、行のアクティブ・セットを定義し、アクティブ・セットの最初の行の直前でカーソルを初期化します。OPEN時のホスト変数の値がSQL文に代入されます。この文は、実際には行を取り出しません。行はFETCH文を使用して取り出されます。

カーソルを一度オープンすると、入力ホスト変数はカーソルを再オープンするまで再テストされません。入力ホスト変数およびアクティブ・セットを変更するには、カーソルを再オープンする必要があります。

プログラム内のすべてのカーソルは、プログラムを開始する場合またはCLOSE文を使用してカーソルを明示的にクローズした後にクローズ状態になります。

カーソルは事前にクローズしなくても、再オープンできます。この文の詳細は、INSERT文を参照してください。

例

```
char dyn_statement[1024] ;
...
EXEC SQL ALLOCATE DESCRIPTOR 'input_descriptor' ;
EXEC SQL ALLOCATE DESCRIPTOR 'output descriptor'
...
EXEC SQL PREPARE S FROM :dyn_statement ;
EXEC SQL DECLARE C CURSOR FOR S ;
...
EXEC SQL OPEN C USING DESCRIPTOR 'input_descriptor' ;
...
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE CURSOR (埋込みSQLディレクティブ)。

FETCH DESCRIPTOR (実行可能埋込みSQL)

PREPARE (実行可能埋込みSQL)。

E.69 PREPARE (実行可能埋込みSQL)

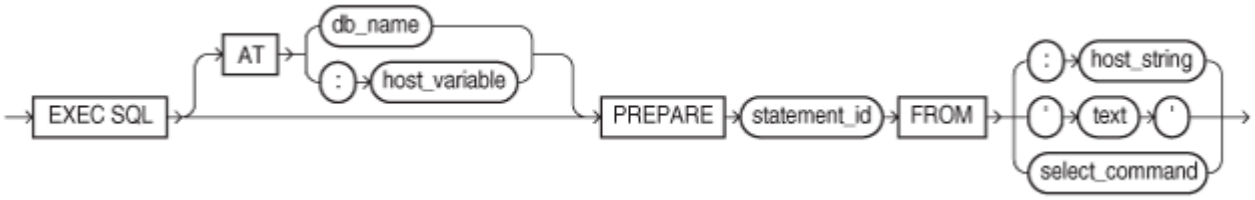
用途

ホスト変数で指定するSQL文またはPL/SQLブロックを解析し、識別子に対応付けます。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
statement_id	準備済の SQL 文または PL/SQL ブロックに対応付ける識別子。この識別子がすでに別の文またはブロックに割り当てられている場合は、以前の割当てが置き換えられます。
db_name	CONNECT 文で先に設定されるデータベース接続名を含む NULL 終了文字列。省略した場合または空文字の場合は、デフォルトのデータベース接続とみなされます。
host_variable	データベース接続名を格納するホスト変数。

キーワードおよびパラメータ	説明
text	準備する SQL 文または PL/SQL ブロックを含む文字列リテラル。
select_command	選択文。

使用上の注意

*host_string*または*text*の変数はすべてプレースホルダです。実際のホスト変数名は、OPEN文のUSING句(入力ホスト変数)またはFETCH文のINTO句(出力ホスト変数)で割り当てます。

SQL文は一度準備すると、何回でも実行できます。詳細は、PREPARE(動的SQL)を参照してください。

例

この例では、Pro*C/C++の埋込みSQLプログラムにおけるPREPARE文の使用方法を示します。

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

関連項目

CLOSE (実行可能埋込みSQL) 。

DECLARE CURSOR (埋込みSQLディレクティブ)。

FETCH DESCRIPTOR (実行可能埋込みSQL)

OPEN (実行可能埋込みSQL) 。

E.70 REGISTER CONNECT (実行可能埋込みSQL拡張機能)

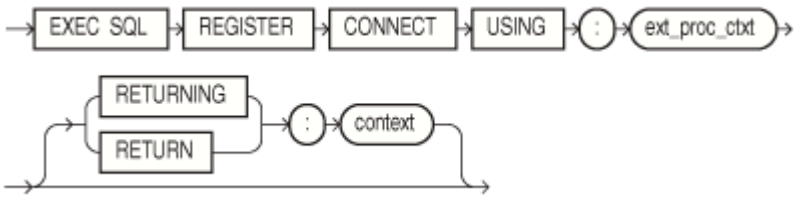
用途

外部Cプロシージャを Pro*C/C++ アプリケーションからコールできるようにします。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
ext_proc_ctxt	PL/SQL によってプロシージャに渡される外部プロシージャ・コンテキスト。 OCIExtProcContext へのタイプ・ポインタです。

キーワードおよびパラメータ	説明
context	戻されるランタイム・コンテキスト。sql_context 型になります。現在の設定は、デフォルト(グローバル)・コンテキストです。

使用上の注意

外部プロシージャを作成する方法および有効な制限の詳細は、「外部プロシージャ」を参照してください。

例

```
void myfunction(epctx)
OCIExtProcContext *epctx;
sql_context context;
...
{
EXEC SQL REGISTER CONNECT USING :epctx ;
EXEC SQL USE :context;
...
```

関連項目

なし。

E.71 ROLLBACK (実行可能埋込みSQL)

用途

現行のトランザクションで実行した作業を取り消します。

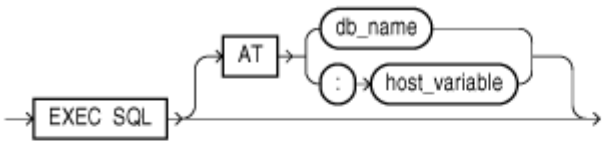
この文は、インダウトの分散トランザクションの処理を手動で取り消すときにも使用できます。

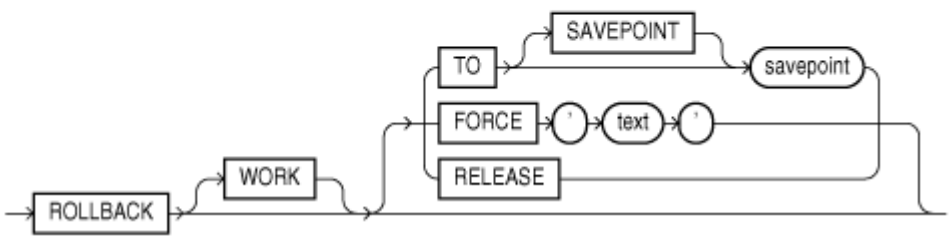
前提条件

現行のトランザクションをロールバックする場合、権限は不要です。

ユーザーがコミットしたインダウトの分散トランザクションを手動でロールバックするには、FORCE TRANSACTIONのシステム権限が必要です。他のユーザーがコミットしたインダウトの分散トランザクションを手動でロールバックするには、FORCE ANY TRANSACTIONのシステム権限が必要です。

構文





キーワードおよびパラメータ

キーワードおよびパラメータ	説明

キーワードおよびパラメータ	説明
AT	セーブポイントをどのデータベースに対して作成するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: 事前に CONNECT 文で確立されたデータベース接続の名前を含み、ヌル文字で終了する文字列。省略した場合または空文字の場合は、デフォルトのデータベース接続とみなされます。 *host_variable*: データベース接続の名前を含むホスト変数。この句を省略した場合、セーブポイントはデフォルトのデータベースに対して作成されます。
WORK	オプションで、ANSI との互換性のために用意されています。
TO	指定したセーブポイントまでカレント・トランザクションをロールバックします。この句を省略した場合、ROLLBACK 文はトランザクション全体をロールバックします。
FORCE	インダウトの分散トランザクションを手動でロールバックします。トランザクションは、ローカル・トランザクション ID またはグローバル・トランザクション ID を含むテキストにより指定します。このようなトランザクションの ID を検索するには、データ・ディクショナリ・ビュー DBA_2PC_PENDING に問合せをします。 ROLLBACK 文での FORCE 句の使用は PL/SQL ではサポートされていません。
RELEASE	リソースをすべて解放し、アプリケーションをデータベースから切断します。RELEASE 句は、SAVEPOINT 句および FORCE 句とは併用できません。
savepoint	ロールバックするセーブポイント

使用上の注意

トランザクション(または論理作業単位)は、Oracleが1つの単位として扱う一連のSQL文です。トランザクションは、COMMIT文、ROLLBACK文またはデータベースへの接続後の、最初の実行SQL文から始まります。トランザクションは、COMMIT文、ROLLBACK文またはデータベースからの切断(意図的かどうかに関係なく)で終了します。Oracleでは、データ定義言語文の処理前および処理後に暗黙的なCOMMIT文が発行されます。

TO SAVEPOINT句を指定せずにROLLBACK文を使用すると、次の処理が実行されます。

- トランザクションを終了します。
- 現行のトランザクションに対するすべての変更の取消し
- トランザクション内のすべてのセーブポイントが消去されます。
- トランザクションのロックが解除されます。

TO SAVEPOINT句を指定してROLLBACK文を使用すると、次の処理が実行されます。

- トランザクションのセーブポイント後の部分のみがロールバックされます。

- 指定したセーブポイントの後に作成したセーブポイントがすべて消去されます。名前付きのセーブポイントが保持されるため、複数回同じセーブポイントにロールバックできます。指定したセーブポイントより前に作成されたセーブポイントも残ります。
- 指定したセーブポイント以降に取得されたすべての表と行のロックの解除。セーブポイント後にロックされた行へのアクセスを要求した他のトランザクションは、コミットまたはロールバックされるまで待つ必要があります。行を要求していない他のトランザクションは、すぐに行の要求およびアクセスができます。

アプリケーション・プログラムでは、COMMIT文またはROLLBACK文を使用してトランザクションを明示的に終了することをお薦めします。トランザクションを明示的にコミットしなかった場合にプログラムが異常終了すると、Oracleはコミットされていない最後のトランザクションをロールバックします。

例1

次の文はカレント・トランザクション全体をロールバックします。

```
EXEC SQL ROLLBACK;
```

例2

次の文はカレント・トランザクションをセーブポイントSP5までロールバックします。

```
EXEC SQL ROLLBACK TO SAVEPOINT sp5;
```

分散トランザクション

Oracleで分散オプションを使用すると、分散トランザクション、または複数データベース上のデータを変更するトランザクションが可能になります。分散トランザクションをコミットまたはロールバックするには、他のトランザクションと同じようにCOMMIT文またはROLLBACK文を発行するだけで済みます。

分散トランザクションのコミット・プロセス中にネットワーク障害が発生すると、トランザクションの状態が不明、つまりインダウトになる可能性があります。そのトランザクションに関連する他のデータベースの管理者に問い合せて、ローカル・データベースのトランザクションを手動でコミットするか、ロールバックするかを決定できます。ローカル・データベースのトランザクションを手動でロールバックするには、FORCE句を指定してROLLBACK文を発行します。

インダウトのトランザクションを手動でセーブポイントまでロールバックすることはできません。

FORCE句を指定したROLLBACK文は、指定したトランザクションのみロールバックします。この文は、現行のトランザクションには影響しません。

例3

次の文は、インダウト分散トランザクションを手動でロールバックします。

```
EXEC SQL
    ROLLBACK WORK
    FORCE '25.32.87' ;
```

関連項目

COMMIT (実行可能埋込みSQL) 。

SAVEPOINT (実行可能埋込みSQL)

E.72 SAVEPOINT (実行可能埋込みSQL)

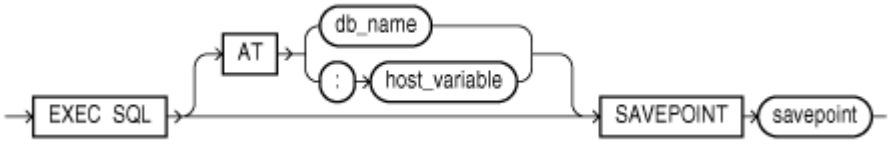
用途

後でロールバックする位置をトランザクション内に指定します。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	セーブポイントをどのデータベースに対して作成するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した db_name の値を持つホスト変数。 この句を省略した場合、セーブポイントはデフォルトのデータベースに対して作成されます。
savepoint	作成するセーブポイントの名前。

使用上の注意

この文の詳細は、SAVEPOINT文の使用を参照してください。

例

この例では、埋込みSQLのSAVEPOINT文の使用方法を示します。

```
EXEC SQL SAVEPOINT save3;
```

関連項目

COMMIT (実行可能埋込みSQL) 。

ROLLBACK (実行可能埋込みSQL)

E.73 SELECT (実行可能埋込みSQL)

用途

選択した値をホスト変数に割り当てて、1つ以上の表、ビューまたはスナップショットからデータを取り出します。

前提条件

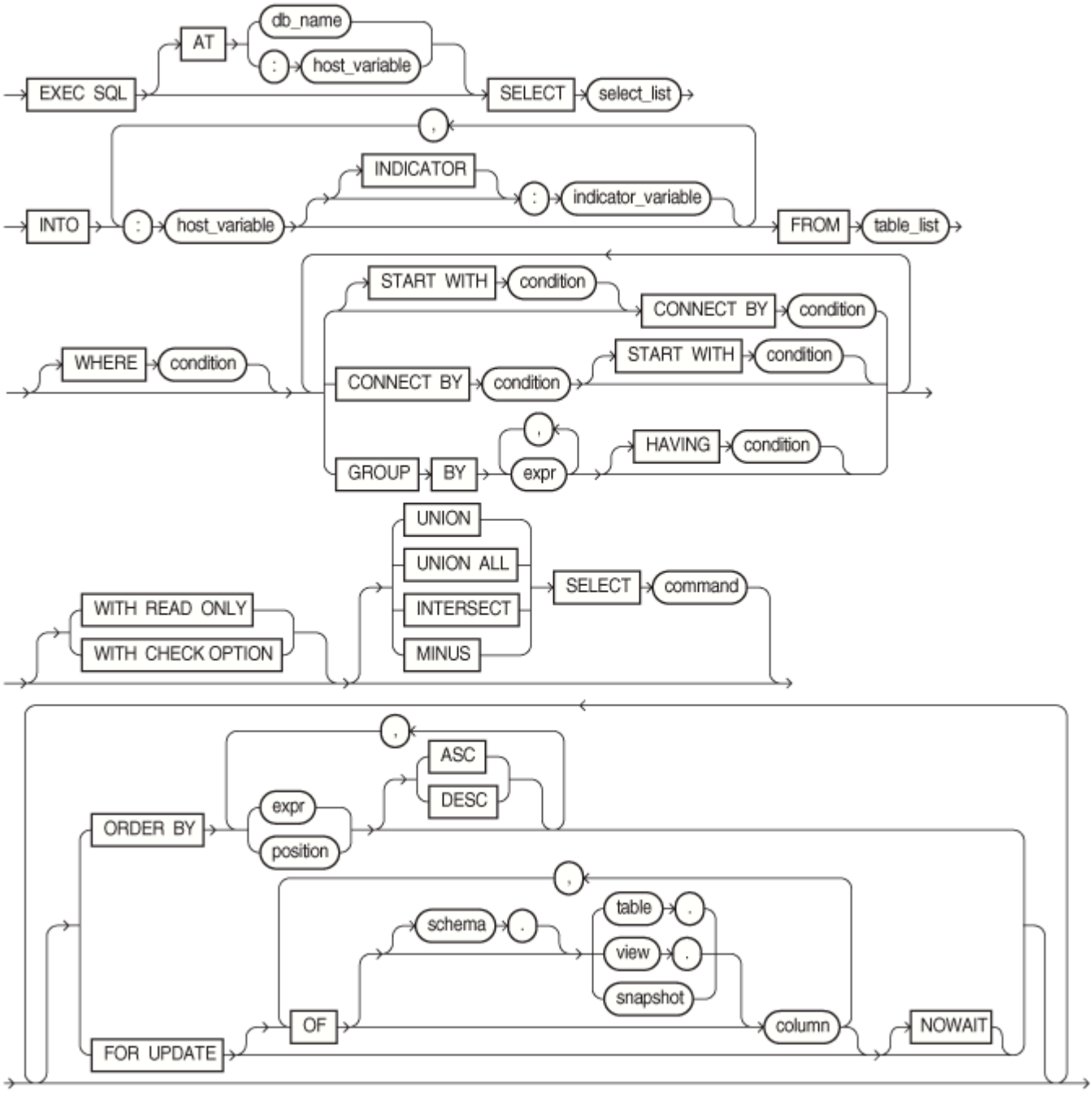
表またはスナップショットからデータを選択するには、表またはスナップショットがユーザーのスキーマ内にあるか、あるいは表またはスナップショットに対してSELECTの権限を持っている必要があります。

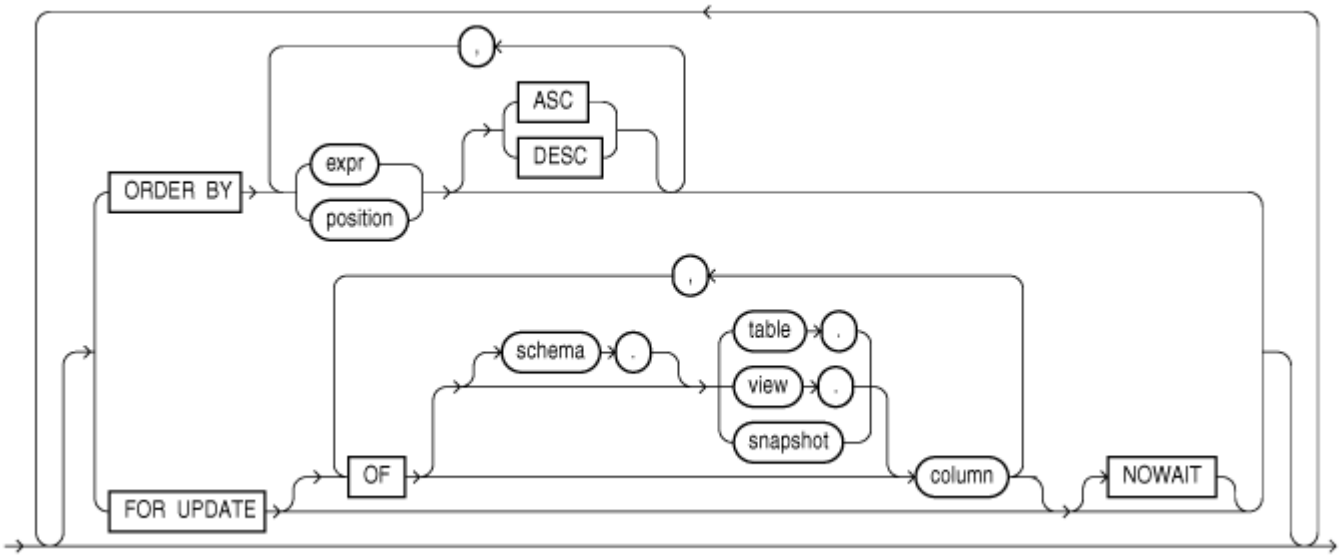
ビューの実表から行を選択するには、ビューが属するスキーマの所有者が、実表に対してSELECTの権限を持っている必要があ

ります。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対してSELECTの権限を持つ必要があります。

SELECT ANY TABLEのシステム権限を使用すると、すべての表、スナップショット、ビューの実表からデータを選択できます。

構文





キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	どのデータベースに対して SELECT 文を発行するかを指定します。次のいずれかを使用してデータベースを指定します。 *db_name*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した db_name の値を持つホスト変数。この句を省略した場合、SELECT 文はデフォルトのデータベースに対して発行されます。
select_list	非埋込み SELECT 文と同じですが、リテラルのかわりにホスト変数を使用できます。
INTO	SELECT 文が戻すデータを受け取る出力ホスト変数とオプションの標識変数を指定します。これらの変数は、すべてスカラーか、すべて配列である必要があります。ただし、配列は同じサイズでなくてもかまいません。
WHERE	戻される行を、条件が TRUE の行のみに制限します。condition の構文は、「条件」を参照してください。条件には、ホスト変数は使用できますが、標識変数は使用できません。これらのホスト変数はスカラーである必要があり、配列は使用できません。 その他のキーワードおよびパラメータは、非埋込み SQL の SELECT 文と同じです。ORDER BY 句のデフォルトは、昇順を示す ASC です。

使用上の注意

WHERE句の条件を満たす行が存在しない場合、行は取り出されず、OracleはSQLCAのSQLCODEコンポーネントを使用してエラー・コードを戻します。

SELECT文でコメントを使用して、指示(つまりヒント)をオプティマイザに渡すことができます。

例

この例では、埋込みSQLのSELECT文の使用方法を示します。

```
EXEC SQL SELECT ename, sal + 100, job
    INTO :ename, :sal, :job
    FROM emp
    WHERE empno = :empno;
```

関連項目

DECLARE CURSOR (埋込みSQLディレクティブ)。

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

EXECUTE (実行可能埋込みSQL)。

FETCH (実行可能埋込みSQL) 。

PREPARE (実行可能埋込みSQL)。

E.74 SET DESCRIPTOR (実行可能埋込みSQL)

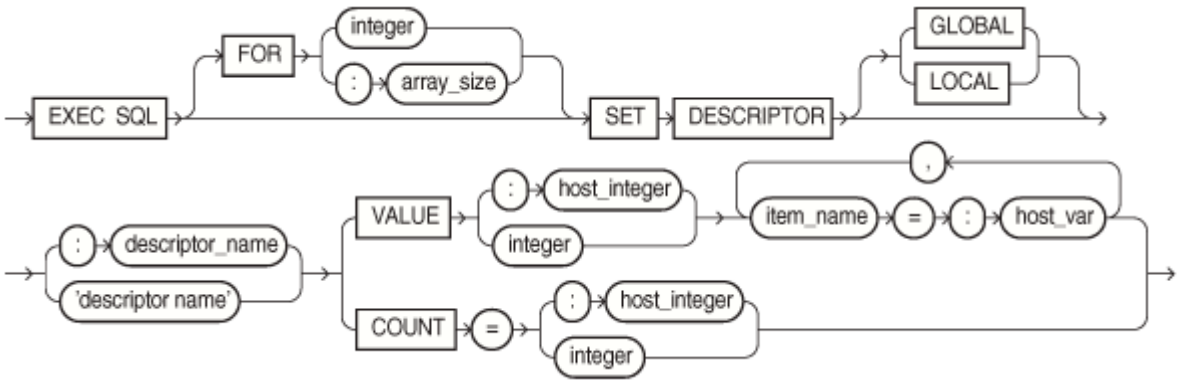
用途

ANSI動的SQL文を使用して、ホスト変数の記述子領域内の情報を設定します。

前提条件

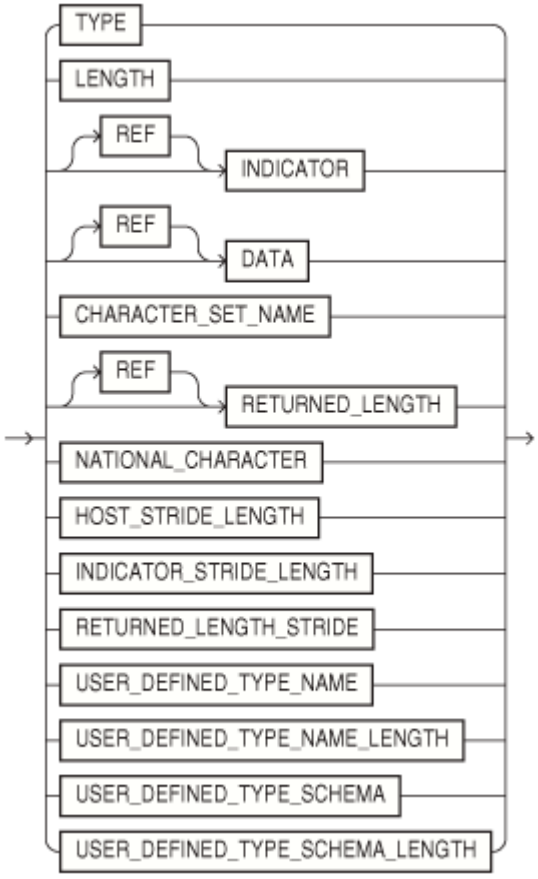
DESCRIBE DESCRIPTOR文の後に使用します。

構文



item_nameのみ次の中から選択できます。

item_nameのみ次の中から選択できます。



キーワードおよびパラメータ

キーワードおよびパラメータ	**説明**
array_size	処理される行の数を格納するホスト変数。
integer	処理される行数。配列サイズの句は、DATA、RETURNED_LENGTH および INDICATOR の項目名でのみ使用できます。
GLOBAL | LOCAL	LOCAL (デフォルト)はファイルのスコープで、GLOBAL はアプリケーションのスコープです。
descriptor_name	割り当てられた ANSI 記述子名を格納するホスト変数。
'descriptor name'	割り当てられた ANSI 記述子の名前。
COUNT	入力変数または出力変数の数。
VALUE	参照されるホスト変数の位置。
item_name	*item_names* のリストおよび説明は、表 10-6 および表 10-7 を参照してください。
host_integer	項目または COUNT または VALUE の設定に使用するホスト変数。
integer	COUNT または VALUE の設定に使用する整数。

キーワードおよびパラメータ	説明
host_var	記述子項目の設定に使用するホスト変数。
REF	参照セマンティクスが使用されます。RETURNED_LENGTH、DATA および INDICATOR の項目名以外では使用できません。 RETURNED_LENGTH を設定するときに使用する必要があります。

使用上の注意

DYNAMIC=ANSIプリコンパイラ・オプションを使用してください。

クライアント側でUnicodeをサポートするには、CHARACTER_SET_NAMEにUTF16を設定します。

記述子項目名の表などの詳細は、SET DESCRIPTORを参照してください。

例

```
EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 ;
```

関連項目

ALLOCATE DESCRIPTOR (実行可能埋込みSQL)。

DEALLOCATE DESCRIPTOR (埋込みSQL文)。

DESCRIBE(実行可能埋込みSQL拡張機能)

GET DESCRIPTOR (実行可能埋込みSQL)。

PREPARE (実行可能埋込みSQL)。

E.75 TYPE (Oracle埋込みSQLディレクティブ)

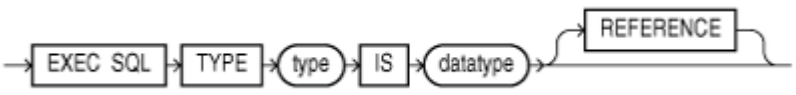
用途

ユーザー定義型の同値化を行うか、外部データ型をユーザー定義のデータ型に同値化することで、外部データ型をホスト変数のクラス全体に割り当てます。

前提条件

ユーザー定義のデータ型は、埋込みSQLプログラムで事前に宣言する必要があります。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
type	外部データ型と同値化するユーザー定義データ型。
datatype	プリコンパイラによって認識される内部データ型ではない外部データ型。データ型には、長さ、精度または位取りを含めることができます。この外部データ型は、ユーザー定義型と同

キーワードおよびパラメータ	説明
	値化された後に、型が割り当てられているホスト変数すべてに割り当てられます。外部データ型のリストは、Oracle のデータ型を参照してください。
REFERENCE	同値化した型をポインタ型にします。

使用上の注意

ユーザー定義型の同値化は、データ型の同値化の一種です。Pro*C/C++プログラムでは、ユーザー定義型を同値化するには、埋込みSQLのTYPE文を使用する必要があります。データ型の同値化は、次のいずれかの目的で使用します。

- 文字ホスト変数を自動的にヌル文字で終了します。
- プログラム・データをバイナリ・データとしてデータベースに格納します。
- デフォルトのデータ型のかわりに使用します。

Pro*C/C++では、VARCHARおよびVARRAW配列がワード整列されているものとみなされます。配列型をVARCHARまたはVARRAWデータ型に同値化する場合は、長さ+2が4で割り切れる数になっていることを確認してください。

Pro*C/C++では、ホスト変数の同値化のための埋込みSQL VAR文もサポートされています。詳細は、ユーザー定義型同値化を参照してください。

例I

この例では、Pro*C/C++のプリコンパイラ・プログラムにおける埋込みSQL TYPE文を示します。

```
struct screen {
    short len;
    char  buff[4002];
};

typedef struct screen graphics;

    EXEC SQL TYPE graphics IS VARRAW(4002);
    graphics crt;  -- host variable of type graphics
    ...
```

関連項目

VAR (Oracle埋込みSQLディレクティブ)

E.76 UPDATE (実行可能埋込みSQL)

用途

表またはビューの実表の既存の値を変更します。

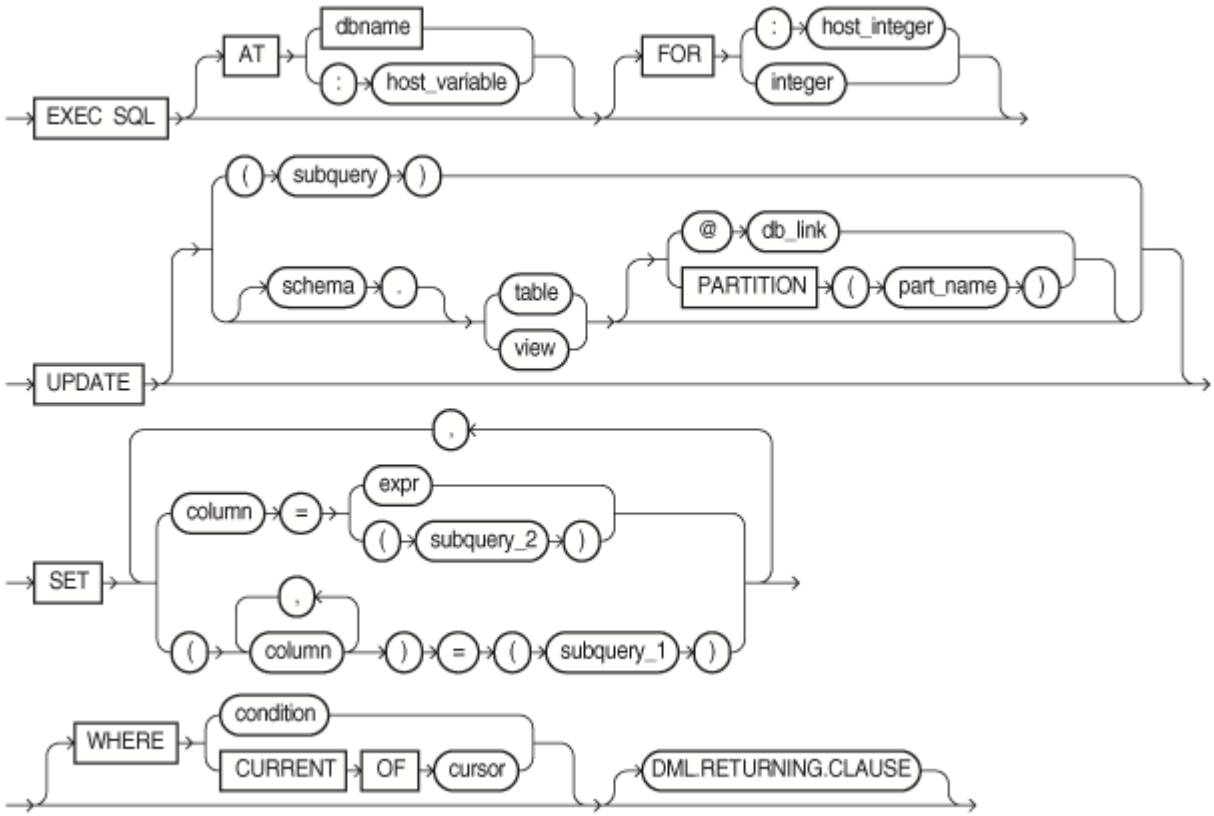
前提条件

表またはスナップショットの値を更新するには、表がユーザーのスキーマ内にあるか、または表に対してUPDATEの権限を持っている必要があります。

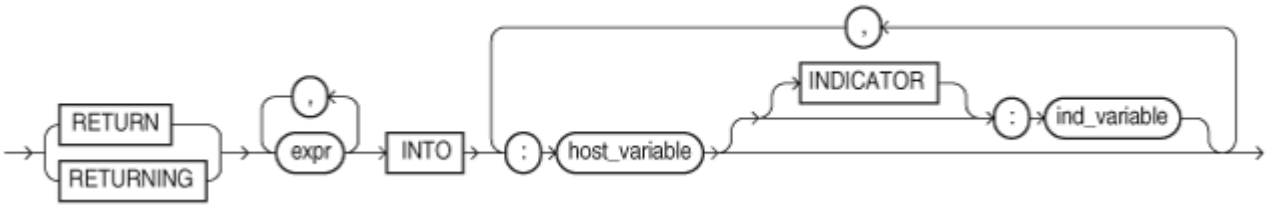
ビューの実表の値を更新するには、ビューが属するスキーマの所有者が、実表に対してUPDATEの権限を持っている必要があります。また、ビューがユーザーのスキーマ以外のスキーマ内にある場合は、ビューに対してUPDATEの権限を持っている必要があります。

UPDATE ANY TABLEのシステム権限により、すべての表またはビューの実表の値も更新できます。

構文



DML RETURNING句は次のとおりです。



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
AT	UPDATE 文の発行先のデータベースを識別します。次のいずれかを使用してデータベースを指定します。 *dbname*: DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 *host_variable*: 事前に宣言した db_name の値を持つホスト変数。 この句を省略した場合、UPDATE 文はデフォルトのデータベースに対して発行されます。
FOR *:host_integer* *integer*	SET 句および WHERE 句が配列ホスト変数を含む場合に、UPDATE 文を実行する回数を制限します。この句を省略した場合、Oracle は最小の配列の各コンポーネントにつき 1 回ずつ文が実行されます。

キーワードおよびパラメータ	説明
schema	表またはビューを含むスキーマ。schema を省略した場合、Oracle では表またはビューが自分のスキーマ内にあるとみなされます。
table、*view*	更新する表の名前。ビューを指定すると、Oracle ではそのビューの実表が更新されます。
dblink	表またはビューがあるリモート・データベースへのデータベース・リンクの完全または部分的な名前。データベース・リンクの参照の詳細は、*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・リンクを使用してリモートの表またはビューを更新できるのは、Oracle で分散オプションを使用している場合のみです。
part_name	表のパーティションの名前。
column	表またはビューで更新する列の名前。SET 句から表の列を削除する場合、その列の値は変更されません。
expr	対応する列に割り当てる新しい値。この式には、ホスト変数およびオプションの標識変数を含めることができます。
subquery_1	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文は、SELECT を参照してください。
subquery_2	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文は、SELECT を参照してください。
WHERE	更新される表またはビューの行を指定します。
condition CURRENT OF	この条件が TRUE の行のみを更新します。この条件には、ホスト変数およびオプションのインジケータ変数を含めることができます。条件の構文は、条件を参照してください。 カーソルによって最後にフェッチされた行のみを更新します。結合を実行する SELECT 文にカーソルを対応付けるには、FOR UPDATE 句で明示的に 1 つの表のみをロックするほかに方法はありません。 この句を完全に省略した場合、表またはビューのすべての行が更新されます。
DML RETURNING 句	DML RETURNING 句を参照してください。

使用上の注意

SET句およびWHERE句に含まれるホスト変数は、すべてスカラーか、またはすべて配列であることが必要です。変数がスカラーの場合、OracleはUPDATE文を1回のみ実行します。変数が配列の場合、Oracleは配列のコンポーネント・セットごとに1回ずつこの文を実行します。1回の実行で、0行または1行、複数行を更新できます。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracleが文を実行する回数は、次のうちの小さい方の値によって決定します。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値

更新された行の累積数は、SQLCAのSQLERRDコンポーネントの第3要素に設定されて戻されます。入力ホスト変数として配列を使用した場合、この数値はUPDATE文で処理された配列のすべてのコンポーネントにおよぶ更新数の合計を示します。この条件を満たす行が存在しない場合、行は更新されず、OracleはSQLCAのSQLCODE要素を通じてエラー・メッセージを戻します。WHERE句を省略した場合は、すべての行が更新され、OracleはSQLCAのSQLWARN要素の第5コンポーネントに警告フラグを設定します。

UPDATE文においてコメントを使用して、指示やヒントをオプティマイザに渡すことができます。オプティマイザは、これらのヒントを使用して文の実行計画を選択します。

この文の詳細は、埋込みSQLおよびデータベースの概要を参照してください。

例

次の例では、埋込みSQLのUPDATE文の使用方法を示します。

```
EXEC SQL UPDATE emp
    SET sal = :sal, comm = :comm INDICATOR :comm_ind
    WHERE ename = :ename;

EXEC SQL UPDATE emp
    SET (sal, comm) =
        (SELECT AVG(sal)*1.1, AVG(comm)*1.1
         FROM emp)
    WHERE ename = 'JONES';
```

関連項目

DECLARE DATABASE (Oracle埋込みSQLディレクティブ)。

E.77 VAR (Oracle埋込みSQLディレクティブ)

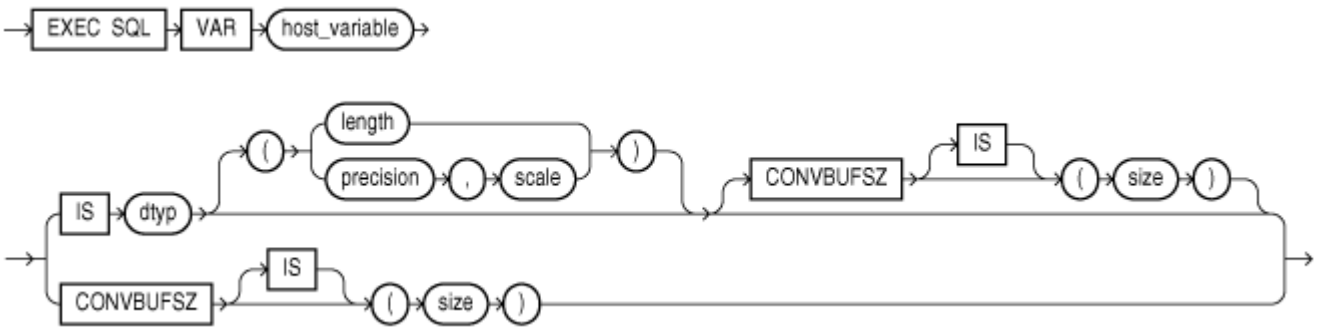
用途

ホスト変数の同値化を行うか、特定の外部データ型を個々のホスト変数に割り当て、デフォルトのデータ型割当てを上書きします。また、オプションのCONVBUFSZ句を使用して、キャラクタ・セットを変換するためのバッファ・サイズを指定します。

前提条件

ホスト変数がPro*C/C++プログラムで宣言済である必要があります。

構文



キーワードおよびパラメータ	説明
host_variable	前に宣言された入力または出力ホスト変数(あるいはホスト表)。 VARCHAR および VARRAW 外部データ型が 2 バイト長のフィールドで、*n* バイトのデータ・フィールドが続く場合、*n* の値の範囲は 1 から 65533 になります。そのため、*type_name* が VARCHAR または VARRAW の場合、*host_variable* には少なくとも 3 バイトの長さが必要です。 LONG VARCHAR および LONG VARRAW 外部データ型が 4 バイト長のフィールドで、*n* バイトのデータ・フィールドが続く場合、*n* の値の範囲は 1 から 2147483643 になります。そのため、*type_name* が LONG VARCHAR または LONG VARRAW の場合、*host_variable* には少なくとも 5 バイトの長さが必要です。
dtyp	Pro*C/C++によって認識される内部データ型ではない外部データ型。データ型には、長さ、精度または位取りを含めることができます。この外部データ型が host_variable に割り当てられます。外部データ型のリストは、外部データ型を参照してください。
length	データ型の長さ。有効な長さをバイト数で指定する定数式または整定数です。length の値は、外部データ型を指定するのに十分な長さにする必要があります。 *type_name* が ROWID または DATE の場合、*length* は事前に定義されているために指定できません。他の外部データ型の場合、*length* はオプションです。デフォルトで *host_variable* の長さに設定されます。 *length* を指定するときに、*type_name* が VARCHAR、VARRAW、LONG VARCHAR または LONG VARRAW の場合は、データ・フィールドの最大長を指定してください。この長さフィールドは、Pro*C/C++が指定します。*type_name* が LONG VARCHAR または LONG VARRAW で、データ・フィールドが 65534 バイト以上の場合は、*length* フィールドに「-1」を入れてください。
precision および *scale*	それぞれ有効桁数と四捨五入が実行される点を表す定数式または定数。たとえば位取りが 2 のときは、1/100 の倍数の近似値に値が四捨五入される(3.456 は 3.46 になる)ことを意味します。また位取りが-3 のときは、1000 の倍数の近似値に値が四捨五入される(3456 が 3000 になる)ことを意味します。 1 から 99 までの *precision* および-84 から 99 までの *scale* を指定できます。ただし、データベース列の精度および位取りの最大値は、それぞれ 38 と 127 です。したがって、*precision* が 38 を超えていると、*host_variable* の値はデータベース列に挿入できません。一方、列値の位取りが 99 を超えていると、*host_variable* に入れる値の選択もフェッチもできません。
size	指定した host_variable から他のキャラクタ・セットへの変換に使用されるバッファのバイ

キーワードおよびパラメータ	説明
	ト単位のサイズ。定数または定数式です。 ランタイム・ライブラリ内のバッファのバイト単位のサイズ。これを使用して、host_variableのキャラクタ・セットを変換します。

使用上の注意

length、precision、scaleおよびsizeは定数式になる場合があります。

ホスト変数の同値化は、一種のデータ型の同値化です。データ型の同値化は次の目的に有効です。

- 文字ホスト変数を自動的にヌル文字で終了します。
- プログラム・データをバイナリ・データとしてデータベースに格納します。
- デフォルトのデータ型のかわりに使用します。

size、*length*、*precision*、*scale*には、プリコンパイラの実行時に値が認識される複雑なC言語の定数式を任意に使用できることに注意してください。

次に例を示します。

```
#define LENGTH 10
...
char character set is nchar_cs ename[LENGTH+1];
exec sql var ename is string(LENGTH+1) convbufsz is (LENGTH*2);
```

また、この文ではマクロも使用できるため注意してください。

CONVBUFSZ句を指定しなければ、Oracleランタイム・ライブラリが、ホスト変数のキャラクタ・サイズ(NLS_LANGで判別)とデータベース・キャラクタ・セットのキャラクタ・サイズとの割合に基づいてバッファ・サイズを自動的に決定します。これによって、LONGサイズのバッファが作成されることがあります。データベース表では、LONG列は1列しか格納できません。複数のLONG値が指定されると、エラーとなります。

このようなエラーを回避するには、LONGのサイズよりも短い長さを使用します。キャラクタ・セットの変換によって値がCONVBUFSZで指定した長さを超える場合は、実行時にエラーが戻されます。Pro*C/C++プリコンパイラは、ユーザー定義型の同値化に使用できるプリコンパイラのTYPEディレクティブもサポートしています。ホスト変数の同値化も参照してください。

例

この例では、ホスト変数DEPT_NAMEがデータ型STRINGに同値化され、ホスト変数BUFFERがデータ型RAW(200)に同値化されます。

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
    char dept_name[15];               -- default datatype is CHAR
    EXEC SQL VAR dept_name IS STRING; -- reset to STRING
    ...
    char buffer[200];                 -- default datatype is CHAR
    EXEC SQL VAR buffer IS RAW(200); -- refer to RAW
    ...
EXEC SQL END DECLARE SECTION;
```

関連項目

E.78 WHENEVER(埋込みSQLディレクティブ)

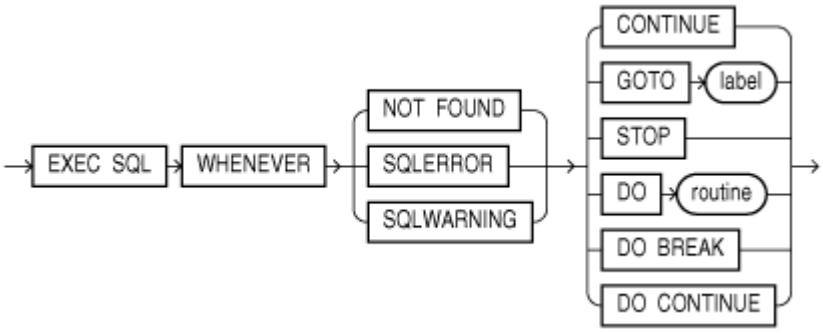
用途

埋込みSQLプログラムの実行時にエラーまたは警告が発生した場合の処置を指定します。

前提条件

なし。

構文



キーワードおよびパラメータ

キーワードおよびパラメータ	説明
NOT FOUND	エラー・コード+1403 (または、MODE=ANSI のときは+100 コード)を SQLCODE に戻す例外状態を示します。
SQLERROR	負のリターン・コードを戻す状態を示します。
SQLWARNING	致命的でない警告状態を示します。
CONTINUE	プログラムが次の文に進む必要があることを指示します。
GOTO *label*	プログラムに *label* で指定した文に分岐するように指示します。
STOP	プログラムの実行を停止します。
DO *routine*	プログラムが *routine* という名前のファンクションをコールすることを示します。
DO BREAK	条件が満たされると、ループから *break* 文が実行されます。
DO CONTINUE	条件が満たされると、ループから *continue* 文が実行されます。

使用上の注意

WHENEVERディレクティブを使用すると、プログラムからエラー処理のルーチンに制御を移すことができます。これは埋込みSQL文でエラーまたは警告が発生したときに可能です。

WHENEVERディレクティブのスコープは論理的にではなく、位置的に適用されます。WHENEVER文は、プログラム論理の流れではなく、ソース・ファイル内で物理的に後続するすべての埋込みSQL文に適用されます。WHENEVERディレクティブは、同じ条件をチェックする別のWHENEVERディレクティブに置換されるまで有効です。

このディレクティブの詳細は、WHENEVERディレクティブの使用についてを参照してください。

埋込みSQLのWHENEVERディレクティブとSQL*PlusコマンドのWHENEVERディレクティブを混同しないでください。

例

次の2つの例では、埋込みSQLプログラムにおけるWHENEVERディレクティブの使用方法を示しています。

例1:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
...
```

例2:

```
EXEC SQL WHENEVER SQLERROR GOTO connect_error;
...
connect_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    printf("¥nInvalid username/password¥n");
    exit(1);
```

関連項目

なし。

F サンプル・プログラム

この付録では、このリリースに含まれているサンプル・プログラムを使用して、Pro*C/C++でOracleデータベース・アプリケーションを作成する方法について説明します。

この章のトピックは、次のとおりです:

- サンプル・プログラムの説明
- サンプル表の作成
- サンプル・プログラムのビルドについて

F.1 サンプル・プログラムの説明

Pro*C/C++をインストールするときに、Oracle Universal Installerによって、Pro*C/C++の一連のサンプル・プログラムが*ORACLE_BASE*¥*ORACLE_HOME*¥precomp¥demo¥procディレクトリにコピーされます。表F-1に、これらのサンプル・プログラムの一覧を示し、後に続く項で説明します。

Oracleが提供するサンプル・プログラムでは、ビルド時に. exe実行可能ファイルが生成されます。

表の「注意」列に示しているように、一部のサンプル・プログラムについては、プリコンパイルして実行する前に、サンプル・ディレクトリにあるSQLスクリプトを実行する必要があります。SQLスクリプトは、サンプル・プログラムが正常に実行されるよう、適切な表およびデータを設定します。これらのSQLスクリプトは、*ORACLE_BASE*¥*ORACLE_HOME*¥precomp¥demo¥sqlディレクトリにあります。

Pro*C/C++が正常にインストールされ、正しく動作することを検証するために、これらのサンプル・プログラムをビルドし、実行することをお薦めします。プログラムは、使用後に削除できます。

サンプル・プログラムは、pcmake. batというバッチ・ファイルか、Microsoft Visual Studioを使用してビルドできます。

関連項目:

サンプル・プログラムのビルドについて

表F-1 サンプル・プログラム

サンプル・プログラム	ソース・ファイル	Pro*C/C++ GUI プロジェクト・ファイル	MSVCコンパイラ・プロジェクト・ファイル	注意
ANSIDYN1	ansidyn1.pc	ansidyn1.pre	ansidyn1.dsp	-
ANSIDYN2	ansidyn2.pc	ansidyn2.pre	ansidyn2.dsp	-
COLDEMO1	coldemo1.h coldemo1.pc coldemo1.sql coldemo1.typ	coldemo1.pre	coldemo1.dsp	coldemo1 のビルド前に、coldemo1.sql と Object Type Translator を実行します。

サンプル・プログラム	ソース・ファイル	Pro*C/C++ GUI プロジェクト・ファイル	MSVCコンパイラ・プロジェクト・ファイル	注意
CPDEMO1	cpdemo1.pc	cpdemo1.pre	cpdemo1.dsp	-
CPDEMO2	cpdemo2.pc	cpdemo2.pre	cpdemo2.dsp	-
CPPDEMO1	cppdemo1.pc	cppdemo1.pre	cppdemo1.dsp	-
CPPDEMO2	cppdemo2.pc empclass.pc cppdemo2.sql empclass.h	cppdemo2.pre	cppdemo2.dsp	cppdemo2 のビルド前に、cppdemo2.sql を実行します。
CPPDEMO3	cppdemo3.pc	cppdemo3.pre	cppdemo3.dsp	-
CVDEMO	cv_demo.pc cv_demo.sql	cv_demo.pre	cv_demo.dsp	cv_demo のビルド前に、cv_demo.sql を実行します。
EMPCLASS	cppdemo2.pc empclass.pc cppdemo2.sql empclass.h	empclass.pre	empclass.dsp	empclass のビルド前に、cppdemo2.sql を実行します。
LOBDEMO1	lobdemo1.h lobdemo1.pc lobdemo1.sql	lobdemo1.pre	lobdemo1.dsp	lobdemo1 のビルド前に、lobdemo1.sql を実行します。
MLTTHRD1	mltthrd1.pc mltthrd1.sql	mltthrd1.pre	mltthrd1.dsp	mltthrd1 のビルド前に、mltthrd1.sql を実行します。
NAVDEMO1	navdemo1.h navdemo1.pc navdemo1.sql navdemo1.typ	navdemo1.pre	navdemo1.dsp	navdemo1 のビルド前に、navdemo1.sql と Object Type Translator を実行します。
OBJDEMO1	objdemo1.h objdemo1.pc objdemo1.sql objdemo1.typ	objdemo1.pre	objdemo1.dsp	objdemo1 のビルド前に、objdemo1.sql と Object Type Translator を実行します。

サンプル・プログラム	ソース・ファイル	Pro*C/C++ GUI プロジェクト・ファイル	MSVCコンパイラ・プロジェクト・ファイル	注意
ORACA	oraca.pc oracatst.sql	oraca.pre	oraca.dsp	oraca のビルド前に、oracatst.sql を実行します。
PLSSAM	plssam.pc	plssam.pre	plssam.dsp	-
SAMPLE	sample.pc	sample.pre	sample.dsp	-
SAMPLE1	sample1.pc	sample1.pre	sample1.dsp	-
SAMPLE2	sample2.pc	sample2.pre	sample2.dsp	-
SAMPLE3	sample3.pc	sample3.pre	sample3.dsp	-
SAMPLE4	sample4.pc	sample4.pre	sample4.dsp	-
SAMPLE5	sample5.pc exampbld.sql examplod.sql	sample5.pre	sample5.dsp	sample5 のビルド前に、exampbld.sql を実行し、次に examplod.sql を実行します。
SAMPLE6	sample6.pc	sample6.pre	sample6.dsp	-
SAMPLE7	sample7.pc	sample7.pre	sample7.dsp	-
SAMPLE8	sample8.pc	sample8.pre	sample8.dsp	-
SAMPLE9	sample9.pc calldemo.sql	sample9.pre	sample9.dsp	sample9 のビルド前に、calldemo.sql を実行します。
SAMPLE10	sample10.pc	sample10.pre	sample10.dsp	-
SAMPLE11	sample11.pc sample11.sql	sample11.pre	sample11.dsp	sample11 のビルド前に、sample11.sql を実行します。
SAMPLE12	sample12.pc	sample12.pre	sample12.dsp	-

サンプル・プログラム	ソース・ファイル	Pro*C/C++ GUI プロジェクト・ファイル	MSVCコンパイラ・プロジェクト・ファイル	注意
SCDEMO1	scdemo1.pc	scdemo1.pre	scdemo1.dsp	-
SCDEMO2	scdemo2.pc	scdemo2.pre	scdemo2.dsp	-
SQLVCP	sqlvcp.pc	sqlvcp.pre	sqlvcp.dsp	-
WINSAM	resource.h winsam.h winsam.ico winsam.pc winsam.rc	winsam.pre	winsam.dsp	-

次の各項では、サンプル・プログラムの機能について説明します。

ANSIDYN1

実行時まで認識されない SQL 文を、ANSI 動的 SQL を使用して処理する方法を示します。このプログラムは、ANSI 動的 SQL を使用するための最も簡単な(ただし、最も効率的というわけではない)方法を紹介することを目的としています。

ANSIDYN2

実行時まで認識されない SQL 文を、ANSI 動的 SQL を使用して処理する方法を示します。このプログラムでは、バッチ処理および参照セマンティクスで Oracle 拡張機能が使用されます。

COLDEMO1

カリフォルニア州の複数の郡についての人口調査情報をフェッチします。このプログラムでは、コレクション型データベース列によりナビゲートする様々な方法を示します。

CPDEMO1

接続プール機能の使用方法を示します。各種接続プール・オプションを使用して、パフォーマンスを最適化する方法も示します。

CPDEMO2

比較的複雑な SQL 文のセットでの接続プーリング機能について示し、パフォーマンスの向上がプログラムで使用される SQL 文の種類によってどのように決まるかを示します。

CPPDEMO1

ユーザーに従業員番号の入力を求め、その従業員の名前、給与および歩合を emp 表に問い合せます。このプログラムでは、インジケータ構造体内の標識変数を使用して、歩合が NULL でないかどうかを判別します。

CPPDEMO2

emp 表から指定した部門の全従業員の名前を取得します(動的 SQL 方法 3)。

CPPDEMO3

すべての営業担当者を検索し、その名前と総収入(歩合を含む)を出力します。このプログラムは、C++の継承の例です。

CVDEMO

参照カーソルを宣言し、オープンします。

EMPCLASS

EMPCLASS および CPPDEMO2 ファイルは、C++フレームワーク内での Pro*C/C++プログラムを記述方法の例を示すためのものです。EMPCLASS は、emp 表に対する特定の問合せをカプセル化したもので、カーソル変数を使用して実装されます。EMPCLASS は、その問合せのインスタンスをインスタンス化し、emp クラスに属する C++メンバー関数により、カーソル変数の機能(open、fetch、close)を提供します。empclass.pc ファイルは、スタンドアロンのデモ・プログラムではありません。これは cppdemo2 デモ・プログラムで使用するために作成されたものです。emp クラスを使用するには、emp クラスのインスタンスを宣言し、そのクラスのメンバー関数をコールするドライバ(cppdemo2.pc)を作成する必要があります。

LOBDEMO1

個人の社会保障番号に基づいて、データベースに対して犯罪記録のフェッチおよび追加を行います。このプログラムは、ラージ・オブジェクト(LOB)にアクセスして表に格納するメカニズム、および DBMS_LOB パッケージにより使用可能になるストアド・プロシージャで LOB を操作するメカニズムを示します。

MLTTHRD1

スレッドをプリコンパイラとともに使用する方法を示します。このプログラムでは、スレッドと同じ数のセッ

ションが作成されます。

関連項目:

マルチスレッド・アプリケーション

NAVDEMO1

オブジェクト・キャッシュ内のオブジェクトへのナビゲーショナル・アクセスを示します。

OBJDEMO1

オブジェクトの使用方法を示します。このプログラムでは、オブジェクト型 *person* および *address* を操作します。

ORACA

実行時に様々なパフォーマンス・パラメータを判断するために ORACA を使用する方法を示します。

PLSSAM

埋込み PL/SQL ブロックの使用方法を示します。このプログラムでは、データベースに登録されている従業員名の入力が要求されます。次に、PL/SQL ブロックが実行され、4 つの SELECT 文の結果が戻されます。

SAMPLE

人事データベースに新しい従業員レコードを追加し、データベースの整合性をチェックします。データベース内の従業員番号には、現在の最大従業員番号+10 の値が自動的に選択されます。

SAMPLE1

Oracle データベースにログインし、ユーザーに従業員番号の入力を要求し、データベースにその従業員の名前、給与および歩合を問い合せて結果を表示します。ユーザーが従業員番号として 0 を入力するまでこの処理を続けます。

SAMPLE2

Oracle データベースにログインし、カーソルを宣言してオープンし、すべての営業担当者の名前、給与および歩合をフェッチして結果を表示し、カーソルをクローズします。

SAMPLE3

Oracle データベースにログインし、カーソルを宣言してオープンし、配列を使用して一括でフェッチし、print_rows()関数を使用して結果を出力します。

SAMPLE4

LONG VARRAW 外部データ型を使用して、型同値化の使用方法を示します。

SAMPLE5

ユーザーに口座番号および引落し金額の入力を要求します。プログラムは、口座から金額を引き落す前に、口座番号が正しいことと、その金額を引き落せるだけの預金があることを確認します。このプログラムは、埋込み SQL の使用方法を示します。

SAMPLE6

表を作成して行を挿入し、挿入をコミットして表を削除します(動的 SQL 方法 1)。

SAMPLE7

emp 表に 2 行挿入し、それらを削除します(動的 SQL 方法 2)。

SAMPLE8

emp 表から指定した部門の全従業員の名前を取得します(動的 SQL 方法 3)。

SAMPLE9

scott/tiger アカウントを使用して、Oracle データベースに接続します。このプログラムでは、複数のホスト配列を宣言し、PL/SQL ストアド・プロシージャ(CALLDEMO パッケージの GET_EMPLOYEES)をコールします。PL/SQL プロシージャは、ASIZE 値まで戻します。プログラムは、すべての行を取得するまで GET_EMPLOYEES をコールし、毎回 ASIZE 配列を取得して、値を出力し続けます。

SAMPLE10

ユーザー名とパスワードを使用して Oracle データベースに接続し、SQL 文の入力を要求します。任

意の有効な SQL 文を入力できますが、埋込み SQL ではなく、通常の SQL 構文を使用する必要があります。入力した文は処理されます。文が問合せの場合、フェッチされた行が表示されます(動的 SQL 方法 4)。

SAMPLE11

カーソル変数を使用して、emp 表からフェッチします。カーソルは、EMP_DEMO_PKG パッケージ内の PL/SQL ストアド・プロシージャ open_cur でオープンされます。

SAMPLE12

動的 SQL 方法 4 を使用して配列フェッチを実行する方法を示します。

SCDEMO1

Oracle 動的 SQL 方法 4 でスクロール可能カーソルを使用する方法を示します。スクロール可能カーソルは、ANSI の動的 SQL 方法 4 でも使用できます。

SCDEMO2

ホスト配列でスクロール可能カーソルを使用する方法を示します。

SQLVCP

sqlvcp()関数を使用して VARCHAR 構造体の実際のサイズを確認する方法を示します。サイズは、VARCHAR の配列を移動するポインタを増やすためのオフセットとして使用されます。

このプログラムは、SQLStmtGetText()関数を使用して、最後に実行された SQL 文のテキストを取得する方法も示します。

WINSAM

人事データベースに新しい従業員レコードを追加し、データベースの整合性をチェックします。人事データベースに新しい従業員レコードを追加し、データベースの整合性をチェックします。必要な数の従業員名を入力でき、「*Employee Record*」ダイアログ・ボックスで適切なボタンを選択して SQL コマンドを実行できます。これは、サンプル・プログラムの GUI バージョンです。

F.2 サンプル表の作成

サンプル・プログラムを実行するには、ユーザー名がscott、パスワードがtigerのデータベース・アカウントが必要です。また、サンプル表empおよびdept. が入っているデータベースが必要です。このアカウントは、Oracle Database 10*g*サーバーの初期デー

タベースに含まれています。このアカウントがデータベースにない場合は、サンプル・プログラムを実行する前にアカウントを作成します。データベースにemp表およびdept表がない場合は、demobld.sqlスクリプトを使用して作成します。

サンプル表を作成するには、次のようにします。

1. SQL*Plusを起動します。
2. ユーザー名scottとパスワードtigerを使用して接続します。
3. demobld.sqlスクリプトを実行します。

```
SQL> @ORACLE_BASE¥ORACLE_HOME¥sqlplus¥demo¥demobld.sql;
```

関連項目

- Oracle Database管理者リファレンスfor Microsoft Windows

F.3 サンプル・プログラムのビルドについて

サンプル・プログラムは、次の2つの方法でビルドできます。

- 用意されたpcmake.batファイルを使用する方法
- Microsoft Visual Studioを使用する方法

F.3.1 pcmake.batを使用する方法

Pro*C/C++デモをコンパイルするためのpcmake.batファイルは、次の場所にあります。

```
ORACLE_BASE¥ORACLE_HOME¥precomp¥demo¥proc
```

このバッチ・ファイルは、コマンド・プロンプトでPro*C/C++アプリケーションを作成する方法を説明するためのものです。

このバッチ・ファイルを使用するには、Microsoft Visual Studioがインストールされている必要があります。環境変数MSVCDirを設定します。Pro*C/C++コマンドライン・オプションおよびリンカー・オプションは、アプリケーションによって異なります。

このファイルを使用して、デモをビルドできます。たとえば、sample1をビルドするには、次のようにします。

1. デモ・ファイルの場所に移動し、コマンド・プロンプトで次のように入力します。

```
C:¥> CD ORACLE_BASE¥ORACLE_HOME¥precomp¥demo¥proc¥sample1
```

2. 次のように入力します。

```
% pcmake sample1
```

F.4 Microsoft Visual Studioを使用する方法

Microsoft Visual Studioプロジェクト・ファイルには、拡張子.dspが付いています。
ORACLE_BASE¥ORACLE_HOME¥precomp¥demo¥procディレクトリにある.dspファイルにより、サンプル・プログラムをプリコンパイル、コンパイルおよびリンクするために必要な手順が示され、制御されます。

Pro*C/C++、SQL*PlusおよびObject Type Translatorは、Microsoft Visual Studioサンプル・プロジェクト・ファイルに統合されています。コンパイル前に、Pro*C/C++、SQL*PlusおよびObject Type Translatorを別々に実行する必要はありません。

サンプル・プログラムをビルドするには、次のようにします。

1. Microsoft Visual Studioプロジェクト・ファイル(sample1.dspなど)を開きます。

2. プロジェクト・ファイル内のパスをチェックして、それらがシステムの構成に対応していることを確認します。対応していない場合、パスをそれに応じて変更します。コンポーネントのパスに正しくないものがある場合、システムでエラー・メッセージが表示される可能性があります。

注意:

サンプル・プログラムはすべて、デフォルト・ドライブを C:¥oracle¥ora92 として作成されています。

1. **「Build」→「Rebuild All」**を選択します。Microsoft Visual Studioは実行可能ファイルを作成します。

関連項目

- サンプル.preファイルのパスの設定
- Object Type Translator

F.5 サンプル.preファイルのパスの設定

デフォルトでは、サンプル. preファイルは、それぞれに対応する. pcファイルをC:¥oracle¥ora92ディレクトリで検索します。C:¥は使用しているドライブ、oracle¥ora92はOracleホームの場所を表します。Oracleベース・ディレクトリとOracleホーム・ディレクトリがコンピュータ上で異なる場合、ディレクトリ・パスを正しいパスに変更する必要があります。

サンプル.preファイルのディレクトリ・パスを変更するには、次のようにします。

1. Pro*C/C++で、. preファイルを開きます。
2. 「入力ファイル」領域でファイル名をダブルクリックし、「入力ファイル」ダイアログ・ボックスを表示します。
3. ディレクトリ・パスを正しいパスに変更します。
4. **「開く」**をクリックします。

G Microsoft Visual Studio .NETへのPro*C/C++の統合

この付録では、Pro*C/C++をMicrosoft Visual Studio .NET 2002/2003以降の統合開発環境に統合する方法について説明します。

この付録の内容は次のとおりです。

- Pro*C/C++のMicrosoft Visual Studio .NETプロジェクトへの統合
- 「ツール」メニューへのPro*C/C++の追加

G.1 Pro*C/C++のMicrosoft Visual Studio .NETプロジェクトへの統合

この項では、Microsoft Visual Studio .NETプロジェクトにPro*C/C++を完全に統合する方法について説明します。

プリコンパイラのすべてのエラーと警告は、Microsoft Visual Studio .NET 2002/2003がコンパイラおよびリンカーのメッセージを表示する出力ボックスに表示されます。Microsoft Visual Studio .NET 2002/2003のビルド環境とは別にファイルをプリコンパイルする必要はありません。さらに重要なことは、Microsoft Visual Studio .NET 2002/2003により、. cファイルと. pcファイルの間の依存性が保持されることです。必要に応じて、Microsoft Visual Studio .NET 2002/2003では、依存性とプリコンパイル・ファイルが保持されます。

この項で説明する手順はすべて、Microsoft Visual Studio .NET内で実行されます。

G.1.1 Pro*C/C++実行可能ファイルの位置の指定

Microsoft Visual Studio .NETからPro*C/C++を実行するには、Pro*C/C++実行可能ファイルの位置が認識されている必要があります。Microsoft Visual Studio .NETをインストールした後でOracle製品をインストールした場合は、ディレクトリ・パスを追加してください。

Pro*C/C++実行可能ファイルの位置を指定するには、次のようにします。

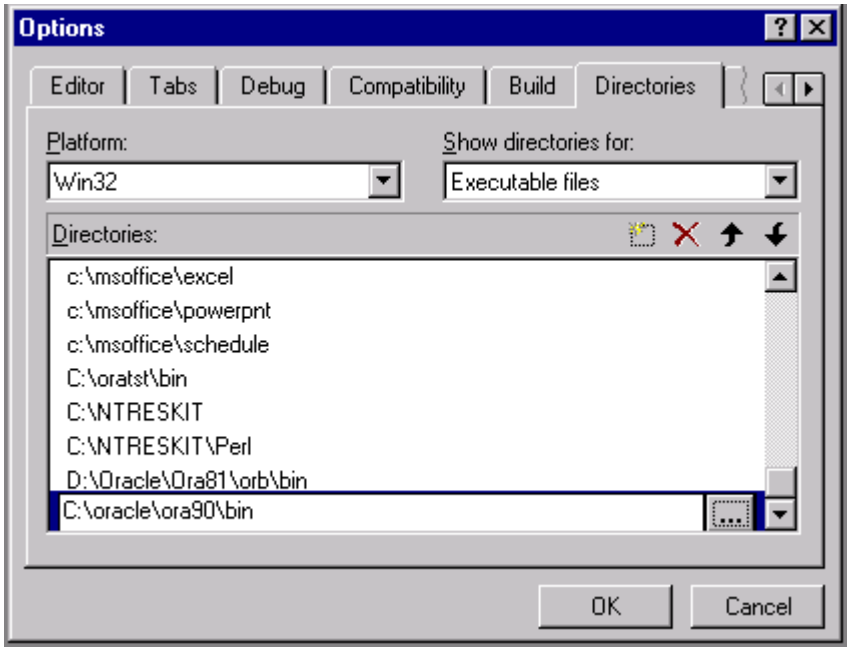
1. 「ツール」メニューから**「オプション」**を選択します。

   「オプション」ダイアログが表示されます。

2. **「ディレクトリ」**タブをクリックします。
3. **「ディレクトリを表示するプロジェクト」**から**「実行可能ファイル」**を選択します。
4. 「ディレクトリ」ボックスの一番下までスクロールし、点線の長方形をクリックします。
5. *ORACLE_BASE*¥*ORACLE_HOME*¥binディレクトリを入力します。次に例を示します。

   ```
   C:¥oracle¥ora121¥bin
   ```

6. **「OK」**をクリックします。



G.1.2 Pro*C/C++ヘッダー・ファイルの位置の指定

Pro*C/C++ヘッダー・ファイルの位置を指定するには、次のようにします。

1. 「ツール」メニューから**「オプション」**を選択します。「オプション」ダイアログが表示されます。
2. **「ディレクトリ」**タブをクリックします。
3. 「ディレクトリを表示するプロジェクト」リストから**「インクルード ファイル」**を選択します。
4. 「ディレクトリ」ボックスの一番下までスクロールし、**点線の長方形**をクリックします。
5. *ORACLE_BASE*¥*ORACLE_HOME*¥precomp¥publicディレクトリを入力します。次に例を示します。

   C:¥oracle¥ora121¥precomp¥public

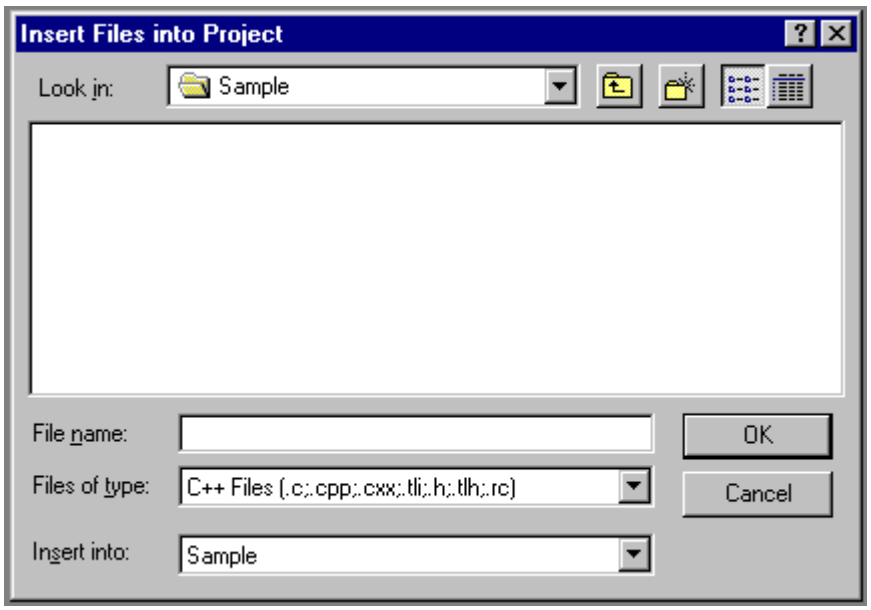
6. **「OK」**をクリックします。

G.2 プロジェクトへの.pcファイルの追加

プロジェクトの作成後、. pcファイルを追加する必要があります。

プロジェクトに. pcファイルを追加するには、次のようにします。

1. 「プロジェクト」メニューから**「プロジェクトへ追加」**→「ファイル」を選択します。「プロジェクトへファイルを追加」ダイアログが表示されます。



2. 「ファイル」リストから**「すべてのファイル」**を選択します。
3. . **pc**ファイルを選択します。
4. **「OK」**をクリックします。

G.2.1 プロジェクトへの.cファイルの参照の追加

各. pcファイルに対して、プリコンパイルの結果生成される. cファイルへの参照を追加する必要があります。

. cファイルへの参照をプロジェクトに追加するには、次のようにします。

1. 「プロジェクト」メニューから**「プロジェクトへ追加」**→「ファイル」を選択します。「プロジェクトへファイルを追加」ダイアログが表示されます。
2. 「ファイル名」ボックスに. cファイルの名前を入力します。
3. **「OK」**をクリックします。. cファイルがまだ作成されていないため、Microsoft Visual Studio .NETにより、「指定されたファイルは存在しません。プロジェクトに対する参照を追加しますか?」というメッセージが表示されます。
4. **「はい」**をクリックします。

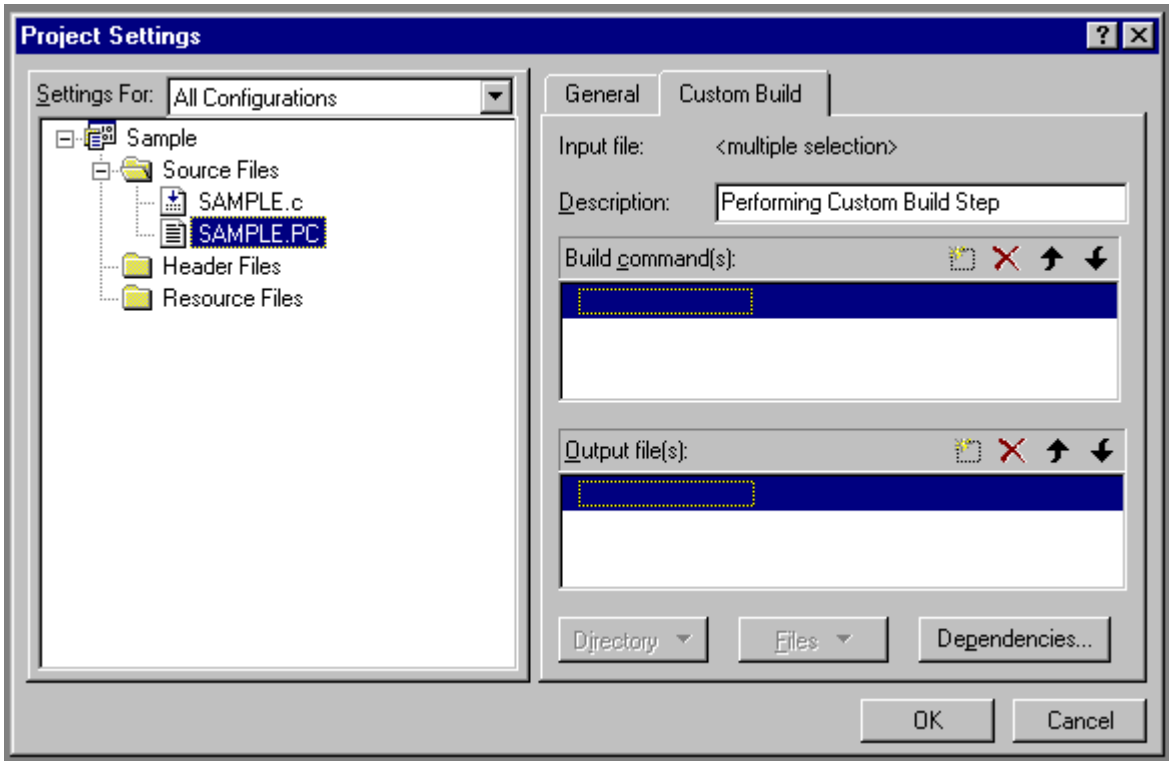
G.2.2 プロジェクトへのPro*C/C++のライブラリの追加

Pro*C/C++アプリケーションは、ライブラリ・ファイルorasql12. libとリンクする必要があります。

プロジェクトへPro*C/C++ライブラリを追加するには、次のようにします。

1. 「プロジェクト」メニューから**「プロジェクトへ追加」**→**「ファイル」**を選択します。「プロジェクトへファイルを追加」ダイアログが表示されます。
2. 「ファイル」リストから**「すべてのファイル」**を選択します。
3. *ORACLE_BASE*¥*ORACLE_HOME*¥precomp¥lib¥msvcディレクトリから**orasql12.lib**を選択します。
4. **「OK」**をクリックします。

G.2.3 カスタム・ビルド・オプションの指定



カスタム・ビルド・オプションを指定するには、次のようにします。

1. ファイルビューで. **pc**ファイルを右クリックし、「設定」を選択します。「プロジェクト設定」ダイアログが「カスタム ビルド」タブ

が表示された状態で表示されます。

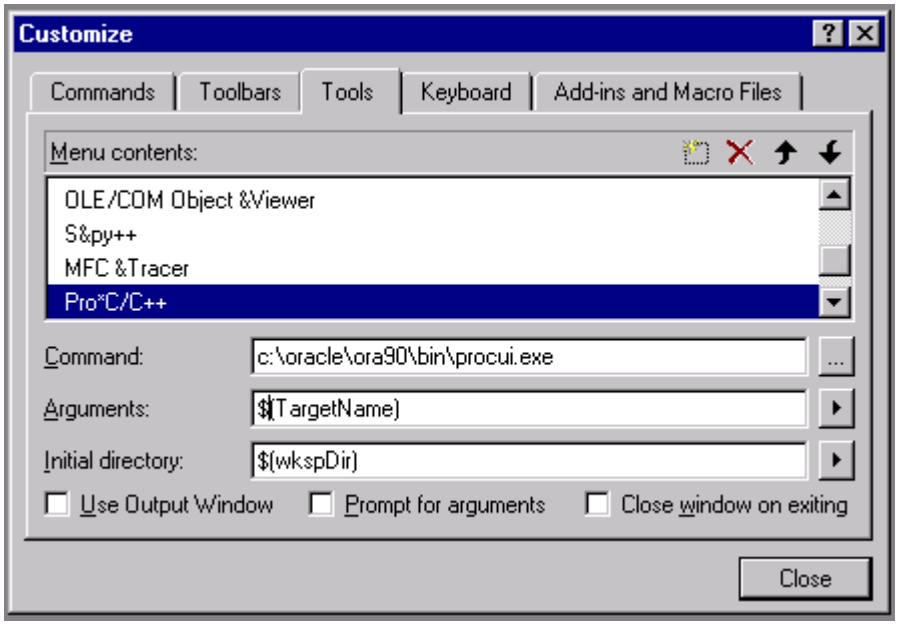
2. 「ビルドのコマンド」ボックスに、$ORACLE_HOME設定と同一のハードコードされたパスを使用するビルドを1行で設定します。
3. 「出力ファイル」ボックスに、次のいずれかを入力します。
   - .cファイルを生成する場合は、$(ProjDir)¥$(InputName).cと入力します。
   - .cppファイルを生成する場合は、$(ProjDir)¥$(InputName).cppと入力します。

   $(ProjDir)および$MSDEVDIRは、Microsoft Visual Studio .NETのカスタム・ビルド・コマンドのマクロです。プロジェクトの作成時、Microsoft Visual Studio .NETでは出力ファイルの日付をチェックして、ソース・コードへの新たな変更のためにプロジェクトを再作成する必要があるかどうかを判断します。
4. **「OK」**をクリックします。

関連項目:

Microsoft Visual Studio .NETのドキュメント

G.3 「ツール」メニューへのPro*C/C++の追加



Microsoft Visual Studio .NETの「ツール」メニューに選択項目として「Pro*C/C++」を追加できます。

「ツール」メニューに「Pro*C/C++」を追加するには、次のようにします。

1. Microsoft Visual Studio .NETで、「ツール」メニューから**「カスタマイズ」**を選択します。「カスタマイズ」ダイアログが表示されます。
2. **「ツール」**タブをクリックします。
3. 「メニューの内容」ボックスの一番下までスクロールして、点線の長方形をクリックします。
4. 次のように入力します。

   Pro*C/C++

5. 「コマンド」ボックスにPro*C/C++のグラフィカルな実行可能ファイルのパスとファイル名を入力するか、ボックスの右側の「参照」ボタンを使用して、ファイル名を選択します。次に例を示します。

   C:¥oracle¥ora121¥bin¥procui.exe

6. 「引数」ボックスに、次のように入力します。

   $(TargetName)

   「ツール」メニューから**「Pro*C/C++」**を選択すると、Microsoft Visual Studio .NETによって$(TargetName)引数が使用され、現行の開発プロジェクトの名前がPro*C/C++に渡されます。これにより、Pro*C/C++は、開いているプロジェクトと同じ名前で拡張子が.preの、プロジェクト・ディレクトリにあるプリコンパイル・プロジェクトを開きます。

7. 「初期ディレクトリ」ボックスに、次のように入力します。

   $(WkspDir)

   「カスタマイズ」ダイアログは、(Oracleホーム・ディレクトリはコンピュータ上では異なる可能性がありますが)次の図のように見えるようになります。

8. **「閉じる」**をクリックします。Microsoft Visual Studio .NETの「ツール」メニューに「Pro*C/C++」が追加されます。

索引

記号 A B C D E F G H I J L M N O P Q R S T U V W X Y Z

記号

- .dspファイル F.4
- .preファイル
  - パスのチェック F.5
- #include
  - ファイルのインクルード, Pro*CとCとの対比 5.4.1

A

- 異常終了
  - 自動ロールバック E.15
- アクティブ・セット
  - 変更 6.5.2, 6.5.3
  - カーソル移動 6.5.3
  - 定義 2.1.9
  - 識別方法 6.5
  - 空の場合 6.5.3
  - フェッチ 6.5.3
  - 未定義になった場合 6.5
- ALLOCATE
  - カーソル変数の割当て 4.5.2
- ALLOCATE DESCRIPTOR文 14.5.1, E.5
- ALLOCATE SQL文 17.4.2
- ALLOCATE SQL文 E.4
- 割当て
  - カーソル E.4
  - カーソル変数 4.5.2
  - スレッド・コンテキスト 11.4.2.2, E.17
- ANSI Cサポート D.1.3
- ANSI動的SQL F.1
  - 参照セマンティクス 14.3.1
  - 「動的SQL(ANSI)」も参照 14
- アプリケーション開発プロセス 2.2
- ARRAYLEN文 7.5.1
- 構造体の配列 8.10
- 配列
  - 一括フェッチ 8.4.1
  - バルク操作(ANSI動的SQL) 14.3.2

    - 使用方法を説明する章 8
    - 定義 4.8.1
    - ホスト配列 2.1.7
    - 操作 2.1.7
    - 可変長 18.1.2
- 連想アクセス用インタフェース 17.4
    - 使用する場合 17.4.1
- AT句
    - CONNECT文 3.2.4.1
    - DECLARE CURSOR文 3.2.4.1.3
    - DECLARE STATEMENT文 3.2.4.1.4
    - EXECUTE IMMEDIATE文 3.2.4.1.4
    - COMMIT文 E.15
    - DECLARE CURSORディレクティブ E.23
    - EXECUTE IMMEDIATE文 E.35
    - EXECUTE文 E.32
    - INSERT文 E.40
    - SAVEPOINT文 E.72
    - SELECT文 E.73
    - SROLLBACK文 E.71
    - UPDATE文 E.76
    - 制限 3.2.4.1.3
    - 使用 3.2.4.1.3
- コレクションの属性
    - 説明 18.4.6
- AUTO_CONNECT 10.5.1
    - プリコンパイラ・オプション 3.1.3.1
- AUTO_CONNECTプリコンパイラ・オプション 10.5.1
- 自動接続 3.1.3, 3.2.4

B

- 一括フェッチ
    - 利点 8.4.1
    - 例 8.4.1
    - 戻される行の数 8.4.3
- BFILE
    - セキュリティ 16.1.3
- バインド記述子 13.10.1, 15.2.1
    - 定義 13.10.1
    - 情報 13.10.2
- バインド
    - 定義 13.5
- バインドSQLDA

    - 用途 15.1.3
- バインド変数
    - 入力ホスト変数 13.10.1
- BREAKアクション
    - WHENEVER E.78の
- バイト順序 D.1.6

C

- C++ 1.8.5
- C++アプリケーション 12
- キャッシュ 17.3
- CACHE FREE ALL SQL文 17.4.4
- CACHE FREE ALL文 E.6
- CALL SQL文 E.7
- CALL文 7.7.2.3
    - 例 7.7.2.4
- CASE OTTパラメータ 19.5.2.10
- 大/小文字区別
    - プリコンパイラ・オプション 10.1.1
- CHAR_MAPプリコンパイラ・オプション 5.1.1, 10.5.2
- 文字データ 5.1
- 文字列
    - マルチバイト 4.11.4
- CHARデータ型 4.1.2.17
- CHARFデータ型 4.1.2.19, 5.3.3
- CHARZデータ型 4.1.2.18
- CLOSE_ON_COMMIT
    - プリコンパイラ・オプション 3.6.2, 6.5.1, 10.5.4
- CLOSE_ON_COMMITプリコンパイラ・オプション 10.5.4
- CLOSE CURSOR文 14.5.13
- CLOSE SQL文 E.8
- CLOSE文
    - プリコンパイラ・オプションへの依存 6.5.4
    - 例 6.5.4, E.8
    - 用途 6.5, 6.5.4
    - 動的SQL方法4での使用 15.6.18
- クローズ
    - カーソル E.8
- CODE
    - プリコンパイラ・オプション 12.2.1
- CODEオプション 10.2.7.2
- CODE OTTパラメータ 19.5.2.4
- コード・ページ 4.10

- CODEプリコンパイラ・オプション 10.5.8
- コーディング規則 2.3
- COLLECT GET文
  - 例 18.5.2
- COLLECTION APPEND E.9
- COLLECTION APPEND文 18.4.4
  - SQL文
    - COLLECTION APPEND E.9
- COLLECTION DESCRIBE
  - 例 18.5.3
- COLLECTION DESCRIBE文 18.4.6
  - SQL文
    - COLLECTION DESCRIBE E.10
- COLLECTION GET文 18.4.1
  - SQL文
    - COLLECTION GET E.11
- コレクション・オブジェクト・タイプ
  - 処理 18.2.2
- COLLECTION RESET文 18.4.3
  - 例 18.5.4
  - SQL文
    - COLLECTION RESET E.12
- コレクション
  - およびC 18.1.3
  - 自律型アクセス 18.2.2.1
  - 記述子 18.2
  - 要素アクセス 18.2.2.2
  - 操作 18.2.2
  - ネストした表 18.1.1
  - OBJECT GET文 18.3
  - OBJECT SET文 18.3
  - VARRAY 18.1.2
- COLLECTION SET文 18.4.2
  - 例 18.5.2
  - SQL文
    - COLLECTION SET E.13
- COLLECTION TRIM文 18.4.5
  - SQL文
    - COLLECTION TRIM E.14
- コレクション型
  - 構造体 18.2
- 列リスト
  - INSERT文 6.3.2
  - 省略が可能な場合 6.3.2
- COMMENT句

    - COMMIT文 E.15
- コメント
    - ANSI 2.3.1
    - 使用できる場合 2.3.1
- コメント
    - PL/SQLブロックの制限 13.12.4
- コミット
    - 自動 3.5
    - 明示的と暗黙的の対比 3.5
    - 機能 3.4
- COMMIT SQL文 E.15
- COMMIT文 3.6
    - 影響 3.6
    - トランザクションの終了 E.71
    - 例 3.6
    - 例 E.15
    - 用途 3.6
    - RELEASEオプション 3.6
    - PL/SQLブロックでの使用 3.14.3
    - 配置する場所 3.6
- コミット
    - トランザクション E.15
- ネットワークでの通信 3.2.1
- COMP_CHARSETプリコンパイラ・オプション 10.5.9, 10.5.10
- コンパイル 2.6
    - インクルード・ファイルの場所の指定 5.4.7
- 同時実行性
    - 定義 3.3
- 同時接続 3.2.2
- 条件付きプリコンパイル 2.4
    - 記号の定義 5.6.1
    - 例 2.4.2, 5.6.2
- CONFIG OTTパラメータ 19.5.2.8
- CONFIGプリコンパイラ・オプション 10.5.3, 10.5.12, 10.5.5, 10.5.6, 10.5.7, 10.5.11, 10.5.14
- 構成ファイル 10.2.2, 10.2.7.1
    - and the Object Type Translator 19.2.2
    - 場所 10.2.2, 10.2.7.1
    - システム 10.2.3
    - ユーザー 10.2.3
- Oracleへの接続 3.1
    - 自動接続 3.1.3
    - 同時 3.2.2
    - 例 3.1
    - Oracle Netを使用 3.2.2
- 接続プーリング 11.6

    - デモ・プログラム1 11.6.2
    - デモ・プログラム2 11.6.3
    - 例 11.6.3.3
    - 使用 11.6.1
- 接続
    - 同時 3.2.4.2
    - デフォルトと非デフォルトの対比 3.2.3
    - 明示的接続 3.2.4
    - 暗黙的 3.2.5
    - 命名 3.2.4
- CONNECT文 E.16
    - AT句 3.2.4.1
    - Oracleへの接続 3.1
    - 例 E.16
    - 要件 3.1
    - USING句 3.2.4.1
    - 意味検査の有効化に使用 C.3.1.1
- const
    - 定数の宣言 5.7.2
- CONTEXT ALLOCATE SQL文 E.17
- CONTEXT ALLOCATE文 11.4.2.2
- コンテキスト・ブロック
    - 定義 20.4.1
- CONTEXT FREE文 11.4.2.4, E.18
- CONTEXT OBJECT OPTION GET SQL文 17.7.2
- CONTEXT OBJECT OPTION SET SQL文 17.7.1
- CONTEXT USEディレクティブ 11.4.2.3
- CONTEXT USE SQLディレクティブ E.21
- CONTEXT USE SQL文 11.4.2.3
- CONTINUEアクション
    - WHENEVER文 9.8.2.1
    - WHENEVERディレクティブ E.78
    - 結果 9.8.2.1
- CONVBUFSZ句 4.11.3
- CPP_SUFFIX
    - プリコンパイラ・オプション 12.2.3
- CPP_SUFFIXプリコンパイラ・オプション 10.5.13
- CPP_SUFFIXプリコンパイラ・オプション 10.5.13
- Cプリプロセッサ
    - Pro*Cでサポートされるディレクティブ 5.4
    - Pro*Cでの使用方法 5.4
- CREATE PROCEDURE文
    - 埋込み 7.7.1
- 作成
    - セーブポイント E.72

- テンポラリLOBの作成 16.4.5
- C言語の構造体
  - REFに対して生成 17.13.1
  - 使用 17.12
- コレクション属性のC言語のデータ型 18.4.6
- CURRENT OF句 8.3.3
  - 例 6.9
  - ROWIDで疑似実行 3.12, 8.11
  - 用途 6.9
  - 制限 6.9.1
- 現在行
  - 定義 2.1.9
  - 検索用のFETCH使用 6.5
- カーソル・キャッシュ
  - 定義 9.10.4
  - 用途 B.9.2.1
- カーソル制御文
  - 一般的な順序の例 6.10
- カーソル操作
  - 概要 6.5
- カーソル 2.5.1.1, 4.5
  - 割当て E.4
  - カーソル変数の割当て 4.5.2
  - 類似点 2.1.9
  - 問合せとの対応付け 6.5
  - クローズ E.8
  - 宣言 6.5.1
  - 定義 2.1.9
  - 明示的と暗黙的の対比 2.1.9
  - 以降の行をフェッチ E.36, E.37
  - 複数行問合せ用 6.5
  - 処理が効率に及ぼす影響 B.9.1
  - アクティブ・セット内での移動 6.5.3
  - オープン E.67, E.68
  - 用途 6.5
  - 再オープン 6.5.2, 6.5.3
  - 宣言時の制限 6.5.1
  - ネーミングのルール 6.5.1
  - スコープ 6.5.1
  - スクロール可能カーソル 6.6
  - 操作用の文 6.5
  - タイプ 2.1.9
  - 複数使用 6.5.1
- カーソル変数 4.5, E.4
  - 割当て 4.5.2

    - 宣言 4.5.1
    - 制限 4.5.6
- SQLDAのC変数
    - 値の設定方法 15.3.9
    - 用途 15.3.9

D

- データベース・リンク
    - シノニムの作成 3.2.5.1
    - 定義 3.2.5.1
    - 使用例, 3.2.5.1
    - INSERT文で使用 E.40
    - 保管場所 3.2.5.1
- データベース
    - 命名 3.2.3
- データベース型
    - 新規 17.15
- データ定義言語
    - トランザクション 3.5
- データの整合性 3.2.4.3
    - 定義 3.3
- データ・ロック 3.3
- データ型の同値化 2.1.8
- データ型
    - ANSI DATE 4.1.3.2
    - 記述子内で使用するコード 15.4.2
    - 変換 5.2
    - ORACLE内部データ型の取扱い 15.4.2
    - 同値化 5.3
    - 同値化, 用途 2.1.8
    - 内部 4.1.1
    - 内部と外部の対比 2.1.6
    - INTERVALDAYTOSECOND 4.1.3.7
    - INTERVALYEARTOMONTH 4.1.3.6
    - 内部データ型のリスト 15.4.1.1
    - 強制変換の必要性 15.4.2
    - Oracle 2.1.6
    - OTTマッピング 19.2.5
    - 使用の制限 17.16
    - TIMESTAMP 4.1.3.3
    - TIMESTAMPWITHLOCALTIMEZONE 4.1.3.5
    - TIMESTAMPWITHTIMEZONE 4.1.3.4
    - ユーザー定義型の同値化 E.75

    - 再設定が必要な場合 15.4.2
- DATE, ANSI
    - データ型 4.1.3.2
- DATEデータ型 4.1.2.10
- 日時
    - 予期しない結果の回避 4.1.3.8
- DBMSとMODEの相互作用 10.5.16
- DBMSオプション 5.3.3, 10.2.7.3
- DBMSプリコンパイラ・オプション 10.5.15, 10.5.16
- デッドロック
    - 定義 3.3
    - トランザクションへの影響 3.8.1
    - 解消方法 3.8.1
- DEALLOCATE DESCRIPTOR文 14.5.2, E.22
- 宣言
    - カーソル 6.5.1
    - ホスト配列 8.2
    - ポインタ変数 4.9.1
    - SQLCA 9.6.1の
- 宣言SQL文
    - トランザクション 3.5
- DECLARE CURSORディレクティブ
    - 例 E.23
- DECLARE CURSOR文 14.5.10
    - AT句 3.2.4.1.3
    - 動的SQL方法4での使用 15.6.7
- DECLARE DATABASE SQLディレクティブ E.24
- 宣言部
    - 使用可能な文 2.3.3
    - 形式 2.3.3
    - 用途 2.3.3
    - MODE=ANSIの場合に必須 10.5.37
    - 要件 2.3.3
    - 定義のルール 2.3.3
    - MODE=ANSIの場合 5.3.4
    - 必要な場合 2.3.3, 4.2.1
- DECLARE文
    - 例 6.5.1
    - 用途 6.5
    - 必要な位置 6.5.1
    - スコープ E.25
    - 動的SQL方法3での使用 13.9.2
- DECLARE STATEMENTディレクティブ E.25
- DECLARE文
    - 例 E.25

- DECLARE STATEMENT文
  - AT句 3.2.4.1.4
  - 使用例 13.11
  - 動的SQLでの使用 13.11
  - 必要な場合 13.11
- DECLARE TABLEディレクティブ
  - 例 E.26
  - SQLCHECKオプションで使用 C.3.1.2
- DECLARE TABLE SQLディレクティブ E.26
- DECLARE TABLE文
  - AT句付きで必要な場合 3.2.4.1.1
- DECLARE TYPEディレクティブ E.27
- DEF_SQLCODEプリコンパイラ・オプション 10.5.17
- デフォルトの接続 3.2.3
- デフォルトのデータベース 3.2.3
- デフォルトのファイル名拡張子 19.5.7
- DEFINEプリコンパイラ・オプション 10.5.18
  - アプリケーションの移行に使用 5.4.9
- 記号の定義 2.4.1
- DELETE SQL文 E.28
- DELETE文
  - 埋込みSQLの例 E.28
  - 例 6.3.4
  - 用途 6.3.4
  - ホスト配列の使用 8.7
  - WHERE句を含む 6.3.4
- デリミタ
  - CとSQLの対比 2.3.4
- DEPT表 2.7
- DESCRIBE BIND VARIABLES文
  - 動的SQL方法4での使用 15.6.8
- DESCRIBEコマンド
  - PREPAREコマンドとともに使用 E.29
- DESCRIBE DESCRIPTOR文 E.30
- DESCRIBE INPUT文 14.5.6
- DESCRIBE OUTPUT文 14.5.7
- DESCRIBE SELECT LIST文
  - 動的SQL方法4での使用 15.6.12
- DESCRIBE SQL文 E.29
- DESCRIBE文
  - 例 E.29
  - 動的SQL方法4での使用 13.10.1
- コレクション属性の説明 18.4.6
- 記述子 15.2.1
  - バインド記述子 13.10.1

    - 定義 13.10
    - 必要性 15.2.1
    - 選択記述子 13.10.1
    - 割当てにsqlald()関数を使用 15.2.4
    - 割当ての解除にsqlclu()関数を使用 15.6.17
- プリコンパイラ・オプションの現在の設定値の確認 10.2.3
- ディレクトリ構造 1.6
- 分散処理
    - サポート 3.2.2
    - Oracle Netを使用 3.2.2
- 分散トランザクション E.71
- DML RETURNING句 6.4
- DOアクション
    - WHENEVER文 9.8.2.2
    - WHENEVERディレクティブ E.78
    - 結果 9.8.2.2
- DTPモデル 5.12
- ダミー・ホスト変数
    - プレースホルダ 13.4
- DURATIONプリコンパイラ・オプション 10.5.20, 17.8.2
- Dynamic Link Library(DLL) 1.7
- 動的PL/SQL
    - 規則 13.12
    - 動的SQLとの対比 13.12
- 動的SQL
    - 長所と短所 13.2
    - カーソル変数の同時使用の不可 4.5.6
    - 適切な方法の選択 13.6.5
    - 定義 2.1.3
    - ガイドライン 13.6.5
    - 方法1 F.1
    - 方法2 F.1
    - 方法3 F.1
    - 方法4 F.1
    - 概要 13.1
    - 制限 6.9.1
    - データ型使用の制限 17.16
    - データ型使用の制限 17.16
    - 使用 13.2
    - AT句の使用 3.2.4.1.4
    - 使用する場合 13.3
- 動的SQL (ANSI)
    - 基本 14.1
    - バルク操作 14.3.2
    - Oracle動的との違い 14.5.14

    - Oracle拡張機能 14.3
    - 概要 14.2
    - プリコンパイラ・オプション 14.4
    - プリコンパイラ・オプション 14.1.1
    - 参照セマンティクス 14.3.1
    - サンプル・プログラム 14.6.2
    - サンプル・プログラム 14.6
- 動的SQL方法1
    - 使用コマンド 13.6.1
    - 説明 13.7
    - 例 13.7.1
    - 使用方法 13.7
    - 要件 13.6.1
    - EXECUTE IMMEDIATEの使用 13.7
    - PL/SQLの使用 13.12.1
- 動的SQL方法2
    - 使用コマンド 13.6.2
    - 説明 13.8
    - 例 13.8.2
    - 要件 13.6.2
    - DECLARE STATEMENTの使用 13.11
    - EXECUTEの使用 13.8
    - PL/SQLの使用 13.12.2
    - PREPAREの使用 13.8
- 動的SQL方法3
    - 使用コマンド 13.6.3
    - 方法2との比較 13.9
    - サンプル・プログラム 13.9.6
    - 要件 13.6.3
    - 使用文の順序 13.9
    - DECLARE STATEMENTの使用 13.11
    - DECLAREの使用 13.9.2
    - FETCHの使用 13.9.4
    - OPENの使用 13.9.3
    - PL/SQLの使用 13.12.3
    - PREPAREの使用 13.9.1
- 動的SQL方法4
    - 記述子の必要性 15.2.1
    - 概要 13.10.1
    - 使用の前提条件 15.4
    - 要件 13.6.4
    - 要件 15.1.1
    - サンプル・プログラム 15.7
    - 使用文の順序 13.10.4, 15.6
    - 手順 15.5

        - CLOSE文の使用 15.6.18
        - DECLARE CURSOR文の使用 15.6.7
        - DECLARE STATEMENTの使用 13.11
        - DESCRの使用 13.10.1
        - DESCRIBE文の使用 15.6.8, 15.6.12
        - 記述子の使用 13.10
        - FETCH文の使用 15.6.15
        - OPEN文の使用 15.6.11
        - PL/SQLの使用 13.12.4
        - PREPARE文の使用 15.6.6
        - SQLDAの使用 13.10.1, 15.2.1
        - 必要な場合 13.10
    - 動的SQL方法
        - 概要 13.6
    - 動的SQL文
        - ホスト変数のバインド 13.5
        - 定義 13.1
        - 要件 13.4
        - プレースホルダの使用 13.4
        - ホスト配列の使用 13.11.1
        - 静的SQL文との対比 13.1

E

    - 埋込みPL/SQL
        - 利点 7.1.1
        - カーソルFORループ 7.1.3
        - 例 7.3.1, 7.3.2
        - 概要 2.1.4
        - パッケージ 7.1.5
        - PL/SQL表 7.1.6
        - プロシージャおよびファンクション 7.1.4
        - 要件 7.2
        - ユーザー定義のレコード 7.1.7
        - %TYPEの使用 7.1.2
        - VARCHAR疑似型の使用 7.3.3
        - パフォーマンスの向上 B.4
        - 使用できる場所 7.2
    - 埋込みSQL F.1
        - ALLOCATE文 E.4
        - CLOSE文 E.8
        - CONTEXT ALLOCATE文 11.4.2.2, E.17
        - CONTEXT FREE文 11.4.2.4
        - 定義 2.1.1

    - 対話型SQLとの違い 2.1.2
    - ENABLE THREADS文 11.4.2.1
    - EXEC SQL CACHE FREE ALL 17.4.4
    - EXECUTE文 E.32
    - 主な概要 2.1
    - ホスト言語文との混在 2.1.2
    - OPEN文 E.67
    - 概要 2.1.1
    - PREPARE文 E.69
    - 要件 2.1.2
    - SAVEPOINT文 E.72
    - SELECT文 E.73
    - 構文 2.1.2
    - SQL*Plusを使用したテスト 1.3
    - TYPEディレクティブ E.75
    - UPDATE文 E.76
    - OCI型の使用 17.14.2
    - REFの使用 17.13.3
    - VARディレクティブ E.77
    - WHENEVERディレクティブ E.78
    - 使用する場合 1.3
- 埋込みSQL文
    - ラベル 9.8.2.5
    - ホスト配列の参照 8.3.1
    - ホスト変数の参照 4.2.2
    - 使用できない接尾辞と接頭辞 2.3.2
    - 終了記号 2.3.13
    - アポストロフィの使用 2.3.4
    - 引用符の使用 2.3.4
- 埋込み
    - プリコンパイラ・プログラム中のPL/SQLブロック E.32
- EMP表 2.7
- ENABLE THREADS SQL文 E.31
- ENABLE THREADS文 11.4.2.1
- 有効化
    - スレッド 11.4.2.1
- コード体系(キャラクタ・セットまたはコード・ページ) 4.10
- エンキュー
    - ロック 3.3
- オプションの入力 10.4
- 環境変数 10.2.1
- 同値化
    - ホスト変数の同値化 E.77
    - ユーザー定義型の同値化 E.75
- データ型の同値化

    - データ型の同値化 2.1.8
- エラー検出
    - エラー・レポート E.78
- エラー処理 2.1.11
    - 代替手段 9.2
    - 必要性 9.1
    - 概要 2.1.11
    - SQLCAとWHENEVER文の対比 9.2.2
    - ROLLBACK文の使用 3.8
- エラー・メッセージ
    - 最大長 9.7
    - 取得用のsqlglm()関数使用 9.7
- エラー・レポート
    - 主要コンポーネント 9.5
    - 解析エラー・オフセットの使用 9.5.4
    - 処理済行数の使用 9.5.3
    - 警告フラグの使用 9.5.2
    - WHENEVERディレクティブ E.78
- ERRORSプリコンパイラ・オプション 10.5.21
- ERRTYPE
    - プリコンパイラ・オプション 10.5.22
- ERRTYPE OTTパラメータ 19.5.2.9
- ERRTYPEプリコンパイラ・オプション 17.8.5
- 例外, PL/SQL
    - 定義 7.4.1
- EXEC ORACLE DEFINE文 5.6
- EXEC ORACLE文 ELSE文 2.4, 5.6
- EXEC ORACLE ENDIF文 2.4, 5.6
- EXEC ORACLE IFDEF文 2.4, 5.6
- EXEC ORACLE IFNDEF文 2.4, 5.6
- EXEC ORACLE OPTION文
    - インラインでのオプション値の設定 10.4
- EXEC ORACLE文
    - スコープ 10.4.2.2
    - 構文 10.4.2
    - 使用 10.4.2.1
- EXEC ORACLE文 2.4
- EXEC SQL CACHE FREE文 17.4.4
- EXEC SQL句
    - SQLの埋込みに使用 2.1.2
- EXEC SQL INCLUDE
    - #includeとの対比 5.4.8
- EXEC SQL VAR文
    - CONVBUFSZ句 4.11.3
- EXEC TOOLS

        - GET CONTEXT文 20.5.5
        - GET文 20.5.3
        - MESSAGE文 20.5.6
        - SET CONTEXT文 20.5.4
        - SET文 20.5.2
- EXEC TOOLS文 20.5
- 実行SQL文
    - 用途 6.3
    - 使用 2.1.1.1
    - 使用できる場所 2.1.1.1
- EXECUTE... END-EXEC SQL文 E.32
- EXECUTE DESCRIPTOR文
    - SQL文
        - EXECUTE DESCRIPTOR E.34
- EXECUTE IMMEDIATE SQL文 E.35
- EXECUTE IMMEDIATE文 14.5.9
    - AT句 3.2.4.1.4
    - 例 E.35
    - 動的SQL方法1での使用 13.7
- ARRAYLEN文のEXECUTEオプション・キーワード 7.5.2
- EXECUTE SQL文 E.33
- EXECUTE文 14.5.8
    - 例 E.32, E.33
    - 動的SQL方法2での使用 13.8
- 文の実行 13.5
- 実行計画 B.5, B.5.2
- EXPLAIN PLAN文
    - 機能 B.5.2
    - パフォーマンスの向上 B.5.2
- 明示的接続 3.2.4
    - 説明 3.2.4
    - 複数 3.2.4.2
    - 単一 3.2.4.1
- 拡張子
    - デフォルト・ファイル名 19.5.7
- 外部データ型
    - 定義 2.1.6
    - FLOAT 4.1.2.4
    - INTEGER 4.1.2.3
    - STRING 4.1.2.5
- 外部プロシージャ
    - コールバック 7.8
    - PL/SQLからのコール 7.8
    - 作成 7.8.2
    - エラー処理 7.8.3

- 制限 7.8.1

F

- FAQ 1.8
- FETCH DESCRIPTOR SQL文 E.37
- フェッチ
  - カーソルからの行 E.36, E.37
- 一括でフェッチ
  - 一括フェッチ 8.4.1
- FETCH SQL文 E.36
- FETCH文 14.5.12
  - 例 6.5.3
  - 例 E.36
  - INTO句を含む 6.5.3
  - 用途 6.5, 6.5.3
  - 結果 6.5.3
  - OPENコマンドの後に使用 E.68
  - OPEN文の後に使用 E.67
  - 動的SQL方法3での使用 13.9.4
  - 動的SQL方法4での使用 15.6.15
- FIPSフラガー
  - 配列の使用方法に関する警告 8.3.4
  - 宣言部の欠落を警告 4.2.1
  - ポインタをホスト変数として使用した場合の警告 5.1.3.3
- FIPSプリコンパイラ・オプション 10.5.24
- フラグ
  - 警告フラグ 9.5.2
- FLOATデータ型 4.1.2.4
- FORCE句
  - COMMIT文 E.15
  - ROLLBACK文 E.71
- FOR句
  - 使用例 8.8
  - 埋込みSQL EXECUTE DESCRIPTOR文 E.34
  - 埋込みSQL EXECUTE文 E.33
  - 用途 8.8
  - 要件 8.8
  - 制限 8.8.1
  - ホスト配列と併用 8.8
  - 変数が負または0(ゼロ)の場合 8.8
- FOR UPDATE OF句
  - 行のロック 3.11
  - 用途 3.11.1

    - 使用する場合 3.11
- 前方参照
    - 許可されない理由 6.5.1
- free()関数 15.6.17
    - 使用例 15.6.17
- 解放
    - スレッド・コンテキスト 11.4.2.4, E.18
- FREE SQL文 17.4.3, E.38
- よくある質問 1.8
- 全体スキャン
    - 説明 B.7
- 関数プロトタイプ
    - 定義 10.5.8
- 関数
    - ホスト変数に指定できない 4.2.2.1
- SQLDAのF変数
    - 値の設定方法 15.3.6
    - 用途 15.3.6

G

- 一般ドキュメント参照
    - オプションのデフォルト値 10.2.7
    - ヘッダー・ファイル，場所 5.5.4
    - リンク 1.7
- GENXTBフォーム
    - 実行方法 20.12
    - ユーザー・イグジットでの使用方法 20.12
- GENXTBユーティリティ
    - 実行方法 20.12
    - ユーザー・イグジットでの使用方法 20.12
- GET DESCRIPTOR文 14.5.3
- グローバリゼーション・サポート 4.10
- GOTOアクション
    - WHENEVER文 9.8.2.5
    - WHENEVERディレクティブ E.78
    - 結果 9.8.2.5
- ガイドライン
    - 動的SQL 13.6.5
    - 分割プリコンパイル 2.5.1
    - WHENEVER文 9.8.6
    - トランザクション 3.14.1

H

- ヘッダー・ファイル
  - 場所 5.5.4
- HEADERプリコンパイラ・オプション 5.5, 10.5.25
- ヒープ
  - 定義 9.10.4
- HFILE OTTパラメータ 19.5.2.7
- ヒント
  - COST B.5.1
  - ORACLE SQL文オプティマイザ用 6.7
  - DELETE文 E.28
  - SELECT文 E.73
  - UPDATE文 E.76
- HOLD_CURSOR
  - プリコンパイラ・オプション
    - 効率改善のために使用 B.9.2.4
    - 影響される事項 B.9
- HOLD_CURSORオプション
  - Oracleプリコンパイラ E.8
- HOLD_CURSORプリコンパイラ・オプション 10.5.25
- ホスト配列
  - 利点 8.1
  - 宣言 8.2
  - 次元 8.2
  - DELETE文 8.7
  - INSERT文 8.5
  - SELECT文 8.4
  - UPDATE文 8.6
  - WHERE句 8.9
  - サイズの一致 8.3.1
  - 参照 8.2.1, 8.3.1
  - 制限 8.3.3, 8.4.7, 8.5.1, 8.6.1, 8.7.1
  - 入力ホスト変数としての使用 8.3
  - 出力ホスト変数としての使用 8.3
  - 動的SQL文での使用 13.11.1
  - FOR句の使用 8.8
  - パフォーマンスの向上 B.3
  - 有効でない場合 8.2.1
- ホスト言語
  - 定義 2.1.1, 2.1.1.1
- ホスト・プログラム
  - 定義 2.1.1
- ホスト構造体
  - 配列 4.8.1

    - 宣言 4.8
- ホスト変数 6.1.1
    - 値の割当て 2.1.5
    - 宣言 18.2.1
    - 宣言 2.3.3, 18.2.1
    - 定義 2.1.5
    - ダミー 13.4
    - ホスト変数の同値化 E.77
    - EXECUTE文 E.33
    - OPEN文 E.67
    - 入力と出力の対比 6.1.1
    - ユーザー・イグジット 20.4.1
    - アドレスへの解決が必要 4.2.2.1
    - 概要 2.1.5
    - 用途 6.1
    - ネーミングのルール 2.3.8
    - PL/SQLでの使用方法 7.3

I

- SQL*FormsのIAP
    - 用途 20.13
- 識別子, ORACLE
    - 形成の方法 E.3.5
- 暗黙的接続 3.2.5
    - 複数 3.2.5.2
    - 単一 3.2.5.1
- INAMEプリコンパイラ・オプション 10.5.28
- INCLUDE
    - SQLCAを組み込むために使用 9.6.1
- INCLUDE, SYS_INCLUDEプリコンパイラ・オプション 10.2.1
- INCLUDEオプション 10.2.7.4
- INCLUDEプリコンパイラ・オプション D.1.11
- 索引
    - パフォーマンスの向上 B.7
- インジケータ配列 8.3.2
    - 使用例 8.3.2
    - 使用 8.3.2
- INDICATORキーワード 4.3.1
- インジケータ変数
    - 値の割当て 6.2
    - ホスト変数との対応付け 6.2
    - 宣言 18.2.1
    - 宣言 4.3, 18.2.1

        - 定義 2.1.5
        - 機能 6.2
        - ガイドライン 4.3.3
        - 値の解釈 6.2
        - 命名 4.8.4
        - 参照 4.3
        - 要件 6.2
        - マルチバイト・キャラクタ文字列で使用 4.11.6
        - PL/SQLでの使用方法 7.4
        - NULLの検出に使用 6.2
        - 切り捨てられた値の検出に使用 6.2
        - NULLの挿入に使用 6.2.1
        - NULLを戻すための使用 6.2.2
        - NULLのテストに使用 6.2.4
        - 構造体 4.8.4
- インダウト・トランザクション 3.13
- INITFILE OTTパラメータ 19.5.2.5
- INITFUNC OTTパラメータ 19.5.2.6
- 初期化関数
    - コール 19.3.2
    - タスク 19.3.3
- IN OUTパラメータ・モード 7.1.4
- INパラメータ・モード 7.1.4
- 入力ホスト変数
    - 値の割当て 6.1.1
    - 定義 6.1.1
    - 制限 6.1.1
    - 使用 6.1.1
    - 使用できる場所 6.1.1
- 挿入
    - 行を表およびビューに E.40
- INSERT SQL文 E.40
    - 例 E.40
- INSERT文
    - 列リストを含む 6.3.2
    - 例 6.3.2
    - INTO句を含む 6.3.2
    - 用途 6.3.2
    - 要件 6.3.2
    - ホスト配列の使用 8.5
    - VALUES句を含む 6.3.2
- INTEGERデータ型 4.1.2.3
- インタフェース
    - ネイティブ 5.12
    - XA 5.12

- 内部データ型
  - 定義 2.1.6
- INTERVAL DAY TO SECONDデータ型 4.1.3.7
- INTERVAL YEAR TO MONTHデータ型 4.1.3.6
- INTO句
  - 出力ホスト変数用 6.1.1
  - FETCH文 6.5.3
  - INSERT文 6.3.2
  - SELECT文 6.3.1
  - FETCH DESCRIPTOR文 E.37
  - FETCH文 E.36
  - SELECT文 E.73
  - SELECTのかわりにFETCHで使用 6.5.1
- intypeファイル 19.5.4
  - OTTの実行時に指定 19.2.4
  - 構造 19.5.4
- INTYPE OTTパラメータ 19.5.2.2
- INTYPEプリコンパイラ・オプション 10.5.30
- 無効な使用
  - プリコンパイラ・プリプロセッサ 5.4.5.1
- SQLDAのI変数
  - 値の設定方法 15.3.5
  - 用途 15.3.5

J

- 結合
  - 制限 6.9.1

L

- ラベル名
  - 最大長 9.8.2.5
- ラージ・オブジェクト F.1
- LDA 5.10
  - リモートの複数接続 5.10.2
  - OCIバージョン8用の設定 5.10.1
- 行
  - 継続 2.3.9
  - 最大長 2.3.10
- LINESプリコンパイラ・オプション 10.5.31
- リンク
  - データベース・リンク 3.2.5.1
- リンク 1.7, 2.6

    - UNIXの場合 1.8.11
    - VMSの場合 1.8.11
    - 2タスク 2.6
- XAライブラリでのリンク D.1.8
- LNAMEプリコンパイラ・オプション 10.5.32
- LNPROC
    - VMSリンク・スクリプト 1.8.11
- LOB APPEND SQL文 E.41
- LOB APPEND文 16.4.1
- LOB ASSIGN SQL文 E.42
- LOB ASSIGN文 16.4.2
- LOB CLOSE SQL文 E.43
- LOB CLOSE文 16.4.3
- LOB COPY SQL文 E.44
- LOB COPY文 16.4.4
- LOB CREATE TEMPORARY SQL文 E.45
- LOB CREATE TEMPORARY文 16.4.5
- LOB DESCRIBE SQL文 E.46
- LOB DISABLE BUFFERING SQL文 E.47
- LOB DISABLE BUFFERING文 16.4.6
- LOB ENABLE BUFFERING SQL文 E.48
- LOB ENABLE BUFFERING文 16.4.7
- LOB ERASE SQL文 E.49
- LOB ERASE文 16.4.8
- LOB FILE CLOSE ALL SQL文 E.50
- LOB FILE CLOSE ALL文 16.4.9
- LOB FILE CLOSE ALL文 16.4.9
- LOB FILE SET SQL文 E.51
- LOB FILE SET文 16.4.10
- LOB FLUSH BUFFER SQL文 E.52
- LOB FLUSH BUFFER文 16.4.11
- LOB FREE TEMPORARY SQL文 E.53
- LOB FREE TEMPORARY文 16.4.12
- LOB LOAD FROM FILE文 16.4.13
- LOB LOAD SQL文 E.54
- LOB OPEN SQL文 E.55
- LOB OPEN文 16.4.14
- LOB READ SQL文 E.56
- LOB READ文 16.4.15
- LOB F.1
    - アクセス方法 16.2.1
    - バッファリング・システム 16.3.2
    - 外部 16.1.2
    - 初期化 16.2.3
    - 内部 16.1.1

    - ロケータ 16.1.5
    - C言語のロケータ 16.2.2
    - テンポラリ 16.1.6
- LOB TRIM SQL文 E.57
- LOB WRITE SQL文 E.58
- インクルード・ファイルの位置 D.1.2
- Pro*C/C++実行可能ファイルの位置 D.1.9
- 位置の透過性
    - 提供方法 3.2.5.1
- ロック,
    - ROLLBACK文による解除 E.71
- ロック 3.11
    - 定義 3.3
    - 明示的と暗黙的の対比 3.11
    - モード 3.3
    - デフォルトの上書き 3.11
    - 取得に必要な権限 3.14.2
    - 表と行の対比 3.11
    - 使用 3.11
    - FOR UPDATE OF 3.11
    - LOCK TABLE文 3.11.2
- LOCK TABLE文
    - すべてのカーソルのクローズ 3.11.2
    - 例 3.11.2
    - 表のロック 3.11.2
    - NOWAITパラメータ 3.11.2
    - 用途 3.11.2
- ログイン 3.1
- ログイン・データ領域 5.10
- LONGデータ型 4.1.2.7
- LONG RAWデータ型 4.1.2.13
- LONG VARCHAR
    - データ型 4.1.2.15
- LONG VARRAWデータ型 4.1.2.16
- LTYPEプリコンパイラ・オプション 10.5.33
- SQLDAのL変数
    - 値の設定方法 15.3.3
    - 用途 15.3.3

M

- マクロ・プリコンパイラ・オプション 10.2.4
- malloc()
    - 使用例 15.6.14

    - 用途 15.6.14
- MAXLITERAL
    - デフォルト値 2.3.11
- MAXLITERALプリコンパイラ・オプション 10.5.35
- MAXOPENCURSORS
    - プリコンパイラ・オプション
        - 効率への影響 B.9.2.2
        - 分割プリコンパイル用の指定 2.5.1.2
        - 影響される事項 B.9
- MAXOPENCURSORSプリコンパイラ・オプション 10.5.36
- MEMFORPREFETCHプリコンパイラ・オプション 10.5.53
- メタデータ 18.4.7
- マイクロ・プリコンパイラ・オプション 10.2.4
- Microsoft Visual Studio
    - Pro*C/C++を統合 G
- 移行
    - インクルード・ファイル 5.4.10
- MODEとDBMSの相互作用 10.5.16
- MODEプリコンパイラ・オプション 10.5.37
- モード, パラメータ 7.1.4
- msvcrt.libランタイム・ライブラリ 1.7
- マルチスレッド・アプリケーション F.1
    - サンプル・プログラム 11.5
    - ユーザー・インタフェースの特長
        - 埋込みSQL文およびディレクティブ 11.4.2
- SQLDAのM変数
    - 値の設定方法 15.3.8
    - 用途 15.3.8

N

- 命名
    - カーソル 6.5.1
    - データベース・オブジェクト E.3.5
    - 選択リスト項目 15.2.1
    - SQL*Formsユーザー・イグジット 20.14.1
- NATIVE
    - DBMSオプションの値 10.5.15, 10.5.16
- ネイティブ・インタフェース 5.12
- ナビゲーショナル・アクセス・サンプル・プログラム 17.11
- ネストした表 18.1.1
    - 作成 18.1.1
- ネットワーク
    - 通信 3.2.1

    - プロトコル 3.2.1
    - 通信量の低減 B.4
- NLS_CHARプリコンパイラ・オプション 10.5.38, 10.5.39
- NLS_LOCALプリコンパイラ・オプション 10.5.40
- NLSパラメータ 4.10
    - NLS_CURRENCY 4.10
    - NLS_DATE_FORMAT 4.10
    - NLS_DATE_LANGUAGE 4.10
    - NLS_ISO_CURRENCY 4.10
    - NLS_LANG 4.10
    - NLS_LANGUAGE 4.10
    - NLS_NUMERIC_CHARACTERS 4.10
    - NLS_TERRITORY 4.10
- ノード
    - カレント 3.2.3
    - 定義 3.2.1
- NOT FOUND条件
    - WHENEVER文 9.8.1.3
    - 意味 9.8.1.3
    - WHENEVERディレクティブ E.78
- NOWAITパラメータ
    - 影響 3.11.2
    - LOCK TABLE文 3.11.2
    - 省略 3.11.2
- NULL
    - 定義 2.1.5
    - 検出 6.2
    - 動的SQL方法4での取扱い 15.4.3
    - ハードコード 6.2.1
    - 挿入 6.2.1
    - 制限 6.2.4
    - 戻す 6.2.2
    - テスト 6.2.4
    - テストにsqlnul()関数を使用 15.4.3
- NULL終了文字列 4.1.2.5.2
- NUMBERデータ型 4.1.2.2
    - sqlprc()関数を使用 15.4.2.1
- 数式
    - ホスト変数に指定できない 4.2.2.1
- SQLDAのN変数
    - 値の設定方法 15.3.1
    - 用途 15.3.1

O

- オブジェクト・キャッシュ 17.3
- OBJECT CREATE SQL文 17.5.3, E.59
- OBJECT DELETE SQL文 17.5.6, E.60
- OBJECT DEREF SQL文 17.5.4, E.61
- OBJECT FLUSH SQL文 E.62
- OBJECT FLUSH SQL文 17.5.8
- OBJECT GET SQL文 17.6.2, E.63
- OBJECT RELEASE SQL文 E.64
- オブジェクト
    - OCIを使用したアクセス 19.3.1
    - デモ・プログラム F.1
    - 概要 17.1
    - OCIを使用した操作 19.3.1
    - 永続 17.3.1
    - 永続コピーと一時コピーの対比 17.3.1
    - 参照 17.1.2
    - サポート 17
    - 一時 17.3.1
    - 型 17.1.1
    - Pro*C/C++でのオブジェクト型の使用 17.2
- OBJECT SET SQL文 17.6.1, E.65
- OBJECTSプリコンパイラ・オプション 10.5.23, 10.5.41, 17.8.3
- Object Type Translator (OTT) F.4
    - コマンドライン 19.2.3
    - コマンドライン構文 19.5.1
    - データベースでの型の作成 19.2.1
    - デフォルト名のマッピング 19.5.7
    - Outtypeファイル 19.2.8
    - パラメータ 19.5.2
    - intypeファイルの指定 19.2.4
    - 参照 19.5
    - 制限 19.5.8
    - 使用 19, 19.2
    - Pro*C/C++での使用19.4
- OBJECT UPDATE SQL文 17.5.7, E.66
- OCIアプリケーション
    - OTTの使用 19.3
- OCIコール
    - 埋込み 5.10
- OCIDate 17.14
    - 宣言 17.14.1
- ocidfn.h 5.10
- OCINumber 17.14
    - 宣言 17.14.1
- OCI onblon()コール

    - 接続には使用されない 5.10
- OCI orlon()コール
    - 接続には使用されない 5.10
- OCIRaw 17.14
    - 宣言 17.14.1
- OCIリリース8 5.8
    - オブジェクトのアクセスおよび操作 19.3.1
    - Pro*C/C++への埋込み5.9.3
    - インタフェース 5.9
    - 環境ハンドルのパラメータ 5.8.2
    - SQLLIB拡張機能 5.8
- OCIString 17.14
    - 宣言 17.14.1
- OCI型
    - 宣言 17.14.1
    - 操作 17.14.3
    - OCIDate 17.14
    - OCINumber 17.14
    - OCIRaw 17.14
    - OCIString 17.14
    - 埋込みSQLでの使用 17.14.2
- ONAMEプリコンパイラ・オプション 10.5.42
- CURSOR文 をOPEN14.5.11
- OPEN DESCRIPTOR SQL文 E.68
- オープン
    - カーソル変数 4.5.3
    - カーソル E.67, E.68
- OPEN SQL文 E.67
- OPEN文
    - 影響 6.5.2
    - 例 6.5.2
    - 例 E.67
    - 用途 6.5, 6.5.2
    - 動的SQL方法3での使用 13.9.3
    - 動的SQL方法4での使用 15.6.11
- 演算子
    - CとSQLの対比 2.3.12
    - 制限 2.3.12
- 最適化のアプローチ B.5.1
- オプティマイザ・ヒント B.5.1
    - C 6.7
    - C++ 6.7.1, 12.2.1
- ORACA
    - 使用例 9.10.6
    - カーソル・キャッシュ統計情報の収集 9.10.5.11

- ORACAIDコンポーネント 9.10.5.1
- ORACAプリコンパイラ・オプション 10.5.27, 10.5.34, 10.5.43, 10.5.44, 10.5.45, 10.5.56
- Oracle
  - データ型 2.1.6
  - Formsのバージョン 4 20.5
  - Open Gateway
    - ROWIDデータ型の使用 4.1.2.9
  - Toolset 20.5
- Oracle Call Interfaceバージョン7 5.10
- Oracle通信領域 9.10
- Traffic DirectorモードのOracle Connection Manager B.12
- Oracleの名前
  - 形成の方法 E.3.5
- Oracle Net
  - Oracleへの接続 3.2.2
  - 接続構文 3.2.1
  - 同時接続 3.2.2
  - 機能 3.2.1
- Oracle Net Services
  - 接続 3.1.2
- orasql9.lib G.2.2
- orasql9.libライブラリ・ファイル 1.7
- OTT (Object Type Translator) F.4
- OTTパラメータ
  - CASE 19.5.2.10
  - CODE 19.5.2.4
  - CONFIG 19.5.2.8
  - ERRTYPE 19.5.2.9
  - HFILE 19.5.2.7
  - INITFILE 19.5.2.5
  - INITFUNC 19.5.2.6
  - INTYPE 19.5.2.2
  - OUTTYPE 19.5.2.3
  - SCHEMA_NAMES 19.5.2.11
  - USERID 19.5.2.1
  - 使用場所 19.5.3
- OTTパラメータTRANSITIVE 19.5.2.12
- OUTパラメータ・モード 7.1.4
- 出力ホスト変数
  - 値の割当て 6.1.1
  - 定義 6.1.1
- outtypeファイル 19.5.4
  - OTTの実行時 19.2.8
- OUTTYPE OTTパラメータ 19.5.2.3
- オーバーヘッド

- 削減 B.2

P

- PAGELEN
  - プリコンパイラ・オプション 10.5.46
- パラメータ・モード 7.1.4
- PARSE
  - プリコンパイラ・オプション 10.5.47
- 解析エラー・オフセット
  - 解釈方法 9.5.4
  - エラー・レポートでの使用 9.5.4
- PARSEオプション 10.2.7.5
- 動的文の解析
  - PREPARE文 E.69
- パスワード
  - 定義 3.1
- パス
  - チェック F.4
  - .preファイルのチェック F.5
- pcscfg.cfg構成ファイル 10.2.7.1
- パフォーマンス
  - 改善のために過剰な解析を排除 B.9
  - 改善のためにSQL文を最適化 B.5
  - 低下の理由 B.1
  - 改善のために埋込みPL/SQLを使用 B.4
  - 改善のためにHOLD_CURSORを使用 B.9.2.4
  - 改善のためにホスト配列を使用 B.3
  - 改善のために索引を使用 B.7
  - 改善のためにRELEASE_CURSORを使用 B.9.2.4
  - 改善のために行レベル・ロックを使用 B.8
- オブジェクトの永続コピー 17.3.1
- 永続オブジェクト 17.3.1
- PL/SQL 1.4
  - 無名ブロック
    - カーソル変数のオープンに使用 4.5.3
  - カーソルFORループ 7.1.3
  - 説明 1.4
  - SQLとの違い 1.4
  - AT句を使用したブロックの実行 3.2.4.1.2
  - データベース・サーバーとの統合 7.1.2
  - 主な利点 1.4
  - パッケージ 7.1.5
  - PL/SQL表 7.1.6

        - プロシージャおよびファンクション 7.1.4
        - RECORD型
            - Cの構造体にバインドできない 4.8.2
        - SQLとの関係 1.4
        - SQLCAの設定 9.6.4
        - ユーザー定義のレコード 7.1.7
- PL/SQLブロック
        - プリコンパイラ・プログラムに埋め込まれる E.32
- プレースホルダ
        - 複製 13.8.1, 13.12.2
        - 命名 13.8.1
        - 適切な順序 13.8.1
        - 動的SQL文での使用 13.4
- PLAN_BASELINE
        - プリコンパイラ・オプション 10.5.48
- PLAN_ENABLED
        - プリコンパイラ・オプション 10.5.52
- PLAN_FIXED
        - プリコンパイラ・オプション 10.5.51
- PLAN_PREFIX
        - プリコンパイラ・オプション 10.5.49
- PLAN_RUN
        - プリコンパイラ・オプション 10.5.50
- ポインタ
        - 定義 4.9
        - カーソル変数
            - 制限 4.5.1
- ポインタ変数
        - 宣言 4.9.1
        - 参照される値のサイズの決定 4.9.2
        - 参照 4.9.2
        - 構造体メンバーの参照 4.9.2
- プリコンパイラ・オプションの優先順位 10.2.3
- 精度
        - 定義 15.4.2.1
        - 抽出にsqlprc()関数を使用 15.4.2.1
        - 指定されていない場合 15.4.2.1
- プリコンパイル
        - 条件付き 2.4
        - 分割 2.5
- プリコンパイル・ユニット 3.1, 10.3
- プリコンパイル済ヘッダー・ファイル 5.5
        - C++制限 5.5.5.2
        - CODEオプション 5.5.5.2
        - PARSEオプション 5.5.5.2

- プリコンパイラ・オプション
  - アルファベット順のリスト 10.3, 10.5
  - AUTO_CONNECT 10.5.1
  - 大/小文字区別 10.1.1
  - CHAR_MAP 5.1.1, 10.5.2
  - CLOSE_ON_COMMIT 6.5.1, 6.6.2, 10.5.4
  - CODE 10.5.8
  - COMP_CHARSET 10.5.9, 10.5.10
  - CONFIG 10.5.3, 10.5.12, 10.5.5, 10.5.6, 10.5.7, 10.5.11, 10.5.14
  - 構成ファイル 10.2.2
  - CPP_SUFFIX 10.5.13
  - DBMS 10.5.15, 10.5.16
  - DEF_SQLCODE 10.5.17
  - DEFINE 10.5.18
  - 現在の設定値の確認 10.2.3
  - DURATION 10.5.20
  - DYNAMIC 14.4
  - 入力 10.4
  - コマンドラインに入力 10.4.1
  - ERRORS 10.5.21
  - ERRTYPE 10.5.22
  - FIPS 10.5.24
  - HEADER 10.5.25
  - HOLD_CURSOR 10.5.25, 10.5.26
  - INAME 10.5.28
  - INCLUDE 10.5.29
  - INTYPE 10.5.30
  - LINES 10.5.31
  - リスト 10.5
  - LNAME 10.5.32
  - LTYPE 10.5.33
  - MAXLITERAL 2.3.11, 10.5.35
  - MAXOPENCURSORS 10.5.36
  - MEMFORPREFETCH 10.5.53
  - マイクロおよびマクロ 10.2.4
  - MODE 10.5.37, 14.4
  - NLS_CHAR 10.5.38, 10.5.39
  - NLS_LOCAL 10.5.40
  - OBJECTS 10.5.23, 10.5.41
  - ONAME 10.5.42
  - ORACA 10.5.27, 10.5.34, 10.5.43, 10.5.44, 10.5.45, 10.5.56
  - PAGELEN 10.5.46
  - PARSE 10.5.47
  - PLAN_BASELINE 10.5.48
  - PLAN_ENABLED 10.5.52

    - PLAN_FIXED 10.5.51
    - PLAN_PREFIX 10.5.49
    - PLAN_RUN 10.5.50
    - 優先順位 10.2.3
    - PREFETCH 10.5.54
    - RELEASE_CURSOR 10.5.55
    - スコープ 10.2.6
    - スコープ 10.3
    - SELECT_ERROR 10.5.57, 10.5.58
    - 指定 10.4.1
    - SQLCHECK 17.8.6
    - 構文 10.4.1
    - SYS_INCLUDE 10.5.59
    - THREADS 10.5.60, 11.4.1
    - TYPE_CODE 10.5.61, 14.4
    - UNSAFE_NULL 10.5.62
    - USERID 10.5.63
    - 使用 10.5
    - VARCHAR 10.5.65
    - VERSION 10.5.66
- プリコンパイラ・オプション, SYS_INCLUDE, INCLUDE 10.2.1
- 事前定義済記号 2.4.1
- PREFETCHプリコンパイラ・オプション 6.6.3, 10.5.54
- PREPARE SQL文 E.69
- PREPARE文 14.5.5
    - データ定義文での効果 13.6.2
    - 例 E.69
    - 動的SQLでの使用 13.8, 13.9.1
    - 動的SQL方法4での使用 15.6.6
- プリプロセッサ
    - 例 5.6.2
    - EXEC ORACLEディレクティブ 5.6
- プリプロセッサ, サポート 4.1
- プリプロセッサ・ディレクティブ
    - Pro*Cでサポートされないディレクティブ 5.4.2.1
- プライベートSQL領域
    - カーソルとの対応付け 2.1.9
    - 定義 2.1.9
    - オープン 2.1.9
    - 用途 B.9.2.1
- Pro*C/C++
    - 構成ファイル 10.2.7.1
    - Microsoft Visual Studioへの統合 G
    - ライブラリ・ファイル G.2.2
    - リンク 1.7

- Pro*C/C++プリコンパイラ
  - 一般的な使用方法 1.2
  - グローバリゼーション・サポート 4.10
  - 新しいデータベース型 17.15
  - オブジェクト・サポート 17
  - ランタイム・コンテキスト 5.8.1
  - PL/SQLの使用 7.2
  - OTTの使用 19.4
- プロシージャ・データベース拡張機能 7.1.5
- プログラミングのガイドライン 2.3
- プログラムの終了
  - 正常と異常の対比 3.9
- プロジェクト・ファイル F.4

Q

- 問合せ
  - カーソルとの対応付け 6.5
  - 転送 3.2.5.1
  - 不正にコードされた 6.3.1
  - 種類 6.3
  - 要件 6.3
  - 複数行を戻す 6.3

R

- RAWデータ型 4.1.2.11
- READ ONLYパラメータ
  - SET TRANSACTION文 3.10
- 読取り専用トランザクション
  - 説明 3.10
  - 例 3.10
  - 終了方法 3.10
- レコード 7.1.7
- REF
  - 構造体 17.13.1
- REF(オブジェクトの参照) 17.1.2
- REFERENCE句
  - TYPE文 5.3.2.1
- 参照セマンティクス(ANSI動的SQL) 14.3.1
- オブジェクトの参照(REF)
  - 宣言 17.13.2
  - 使用 17.13
  - 埋込みSQLでの使用 17.13.3

- 参照
  - ホスト配列 8.2.1, 8.3.1
- REF
  - 宣言 17.13.2
  - 使用 17.13
  - 埋込みSQLでの使用 17.13.3
- REGISTER CONNECT SQL文 E.70
- RELEASE_CURSOR
  - プリコンパイラ・オプション
    - 影響される事項 B.9
- RELEASE_CURSORオプション
  - Oracleプリコンパイラ E.8
  - パフォーマンスの向上 B.9.2.4
- RELEASE_CURSORプリコンパイラ・オプション 10.5.55
- RELEASEオプション 3.9
  - 省略した場合 3.9
  - COMMIT文 3.6
  - ROLLBACK文 3.8
  - 用途 3.6
  - 制限 3.8
- リモート・データベース
  - 宣言 E.24
- 予約済ネームスペース A.2
- 予約語およびキーワード A.1
- リソース・マネージャ 5.12
- 制限事項
  - AT句 3.2.4.1.3
  - コメント 13.12.4
  - CURRENT OF句 6.9.1
  - カーソル宣言時 6.5.1
  - FOR句 8.8.1
  - ホスト配列 8.3.3, 8.4.7, 8.5.1, 8.6.1, 8.7.1
  - 入力ホスト変数 6.1.1
  - NULL上 6.2.4
  - 分割プリコンパイル 2.5.1.1
  - SET TRANSACTION文 3.10
  - CURRENT OF句の使用 8.3.3
- 表から行を取り出す
  - 埋込みSQL E.73
- リターン・コード
  - ユーザー・イグジット 20.8
- RETURNING句
  - DELETE 6.3.4
  - INSERT 6.4内
  - UPDATE 6.3.3内

- RETURNING句 6.4
- ロールバック
  - セーブポイントへ E.72
  - 同じセーブポイントへ複数回ロールバック E.71
- ロールバック
  - 機能 3.4
  - 文レベル 3.8.1
- ROLLBACK SQL文 E.71
- ROLLBACK文
  - 影響 3.8
  - トランザクションの終了 E.71
  - 例 3.8
  - 例 E.71
  - エラー処理ルーチン 3.8
  - 用途 3.8
  - RELEASEオプション 3.8
  - TO SAVEPOINT句 3.8
  - PL/SQLブロックでの使用 3.14.3
  - 配置する場所 3.8
- ROWID
  - 論理 4.1.2.9
  - 疑似列 3.12
    - CURRENT OFの疑似実行に使用 3.12, 8.11
  - ユニバーサル 4.1.2.9
- ROWIDデータ型 4.1.2.9
- 行ロック
  - FOR UPDATE OFで取得 3.11.1
  - 利点 B.8
  - パフォーマンスの向上 B.8
  - 取得される場合 3.11.1.1
  - 解除される場合 3.11.1.1
- 行
  - カーソル以降のフェッチ E.36, E.37
  - 表およびビューに挿入 E.40
  - 更新 E.76
- 処理済行数
  - エラー・レポートでの使用 9.5.3
- ランタイム・コンテキスト
  - 設定 5.8.1
  - 終了 5.8.1
- 実行時のタイプ・チェック 17.8.7

S

- サンプル・データベース表
  - DEPT表 2.7
  - EMP表 2.7
- サンプル・オブジェクト型コード 17.11
- サンプル・プログラム
  - ANSIDYN1 F.1
  - ansidyn1.pc 14.6.1
  - ANSIDYN2 F.1
  - ansidyn2.pc 14.6.2
  - ビルド F.3
  - calldemo.sqlとsample9.pc 7.7.2.1
  - COLDEMO1 F.1
  - coldemo1.pc 18.5.5
  - CPPDEMO1 F.1
  - cppdemo1.pc 12.3.1
  - CPPDEMO2 F.1
  - cppdemo2.pc 12.3.2
  - CPPDEMO3 F.1
  - cppdemo3.pc 12.3.3
  - カーソル変数のデモ 4.5.7
  - CV_DEMO F.1
  - cv_demo.pc 4.5.7.2
  - cv_demo.sql 4.5.7.1
  - デフォルト・ドライブ F.4
  - 説明 F.1
  - EMPCLASS F.1
  - extp1.pc 7.8.2
  - プリコンパイルする方法 2.8, 2.9
  - LOBDEMO1 F.1
  - lobdemo1.pc 16.6.3
  - 場所 1.6, F.1
  - MLTTHRD1 F.1
  - NAVDEMO1 F.1
  - navdemo1.pc 17.11
  - OBJDEMO1 F.1
  - ORACA F.1
  - oraca.pc 9.10.6
  - PLSSAM F.1
  - SAMPLE F.1
  - SAMPLE1 F.1
  - sample1.pc 2.8
  - SAMPLE10 F.1
  - sample10.pc 15.7
  - SAMPLE11 F.1
  - sample11.pc 4.5.7.2

    - SAMPLE12 F.1
    - sample12.pc 15.6.20
    - SAMPLE2 F.1
    - sample2.pc 4.8.5
    - SAMPLE3 F.1
    - sample3.pc 8.4.5
    - SAMPLE4 F.1
    - sample4.pc 5.3.5
    - SAMPLE5 F.1
    - sample5.pc 20.11
    - SAMPLE6 F.1
    - sample6.pc 13.7.1
    - SAMPLE7 F.1
    - sample7.pc 13.8.2
    - SAMPLE8 F.1
    - sample8.pc 13.9.6
    - SAMPLE9 F.1
    - sample9.pc 7.7.2.1
    - パスの設定 F.4
    - .preファイルのパスの設定 F.5
    - SQLVCP F.1
    - sqlvcp.pc 4.4.7
    - WINSAM F.1
- サンプル表
    - ビルド F.2
- セーブポイント
    - 作成 E.72
    - 定義 3.7
    - 使用 3.7
    - 消去される場合 3.7
- SAVEPOINT SQL文 E.72
- SAVEPOINT文 E.72
    - 例 3.7
    - 例 E.72
    - 用途 3.7
- 位取り
    - 定義 15.4.2.1, E.77
    - 抽出にsqlprc()関数を使用 15.4.2.1
    - 負の場合 15.4.2.1, E.77
- 位取り
    - 抽出のためのSQLPRCの使用 E.77
- SCHEMA_NAMES OTTパラメータ 19.5.2.11
    - 使用方法 19.5.6
- 適用範囲
    - カーソル変数 4.5.1

    - DECLARE STATEMENTディレクティブ E.25
    - プリコンパイラ・オプション 10.3
    - EXEC ORACLE文 10.4.2.2
    - WHENEVER文 9.8.5
- 検索条件
    - 定義 6.3.5
    - WHERE句 6.3.5
- SELECT_ERROR
    - プリコンパイラ・オプション 6.3.1, 10.5.57, 10.5.58
- 選択記述子 15.2.1
    - 情報 13.10.2
- 選択リスト
    - 定義 6.3.1
    - 項目数 6.3.1
    - free()関数の使用 15.6.17
    - malloc()関数を使用 15.6.14
- 選択SQLDA
    - 用途 15.1.3
- SELECT SQL文 E.73
- SELECT文 6.3.1
    - 使用可能な句 6.3.1.1
    - 埋込みSQLの例 E.73
    - 例 6.3.1
    - INTO句を含む 6.3.1
    - 用途 6.3.1
    - テスト 6.3.1.1
    - ホスト配列の使用 8.4
    - WHERE句を含む 6.3.1
- 意味検査
    - SQLCHECKオプションを使用した制御 C.2
    - 定義 C.1
    - 有効化 C.3.1
    - SQLCHECKオプション C.1
- 分割プリコンパイル
    - ガイドライン 2.5.1
    - カーソルの参照 2.5.1.1
    - 制限 2.5.1.1
    - MAXOPENCURSORSの指定 2.5.1.2
    - 単一のSQLCAの使用 2.5.1.3
- サーバー
    - PL/SQLとの統合 7.1.2
- セッション
    - 定義 3.3
- セッション
    - 開始 E.16

- SET句
  - UPDATE文 6.3.3
  - 用途 6.3.3
  - 副問合せの使用 6.3.3
- SET DESCRIPTOR文 14.5.4, E.74
- SET TRANSACTION文
  - 例 3.10
  - 用途 3.10
  - READ ONLYパラメータ 3.10
  - 要件 3.10
  - 制限 3.10
- 整数とROWIDのサイズ D.1.5
- SQL
  - 利点 1.3
  - 埋込みSQL 1.3
  - 性質 1.3
  - 必要性 1.3
- SQL_CURSOR E.4
- SQL_SINGLE_RCTX
  - 定義済の定数 5.11
  - 定義 5.9
- SQL, 動的 2.1.3
- SQL*Forms
  - エラー表示画面 20.8
  - IAP定数 20.8.1
  - 値を戻す 20.8
  - 逆戻りコード・スイッチ 20.8
- SQL*Plus 1.3
  - SELECT文のテストに使用 6.3.1.1
  - 埋込みSQLとの比較 1.3
- sqlald()関数
  - 使用例 15.6.3
  - 用途 15.2.4
  - 構文 15.2.4
- sqlaldt()関数
  - 「SQLSQLDAAlloc」を参照5.11
- SQLCA 9.2.2, 9.5
  - コンポーネント 9.6.3
  - PL/SQLブロック用コンポーネント・セット 9.6.4
  - 宣言 9.6.1
  - 説明 9.6
  - 明示的チェックと暗黙的チェックの対比 9.2.2
  - 複数回の組込み 5.4.7
  - 概要 2.1.11
  - SQLCABC component in 9.6.3.2

    - SQLCAID component in 9.6.3.1
    - sqlerrd 9.6.3.6
    - sqlerrd[2] component in 9.6.3.6
    - sqlerrmcコンポーネント 9.6.3.4
    - sqlerrmlコンポーネント 9.6.3.4
    - sqlwarn 9.6.3.7
    - 分割プリコンパイルに使用 2.5.1.3
    - 複数使用 9.6
- sqlca.h
    - リスト 9.6.2
    - SQLCA_STORAGE_CLASSの使用 2.5.1.3
- SQLCAIDコンポーネント 9.6.3.1
- SQLCDAFromResultSetCursor() 5.11
- SQLCDAGetCurrent 5.11
- sqlcdat()
    - 「SQLCDAFromResultSetCursor()」を参照 5.11
- SQLCHECKオプション
    - 制限 C.2
    - 影響される事項 C.2
- SQLCHECKプリコンパイラ・オプション 17.8.6, C.3.1.1, C.3.1.2
- オブジェクトに対するSQLCHECKのサポート 17.8.6
- sqlclu()関数
    - 使用例 15.6.17
    - 用途 15.6.17
    - 構文 15.6.17
- sqlclut()関数
    - 「SQLSQLDAFree()」を参照 5.11
- SQLCODE
    - SQLCAのコンポーネント 9.5.1
- SQLCODE
    - MODE=ANSIの使用 10.5.37
- SQLCODE状態変数
    - 宣言 9.4
    - SQLCAとともに宣言 9.4
    - 使用する場合 9.4
- SQLコミュニケーション領域 9.2.2
    - SQLCA 9.6
- sqlcpr.h 9.7
- sqlcurt()関数
    - 「SQLDAToResultSetCursor()」を参照 5.11
- SQLDA
    - バインドと選択の対比 13.10.3
    - C変数 15.3.9
    - 定義 13.10.3
    - F変数 15.3.6

    - 格納される情報 13.10.3
    - I変数 15.3.5
    - L変数 15.3.3
    - M変数 15.3.8
    - N変数 15.3.1
    - 用途 13.10.1
    - 構造体, 内容 15.2.3
    - 構造 15.3
    - S変数 15.3.7
    - 動的SQL方法4での使用 15.2.1
    - V変数 15.3.2
    - X変数 15.3.10
    - Y変数 15.3.11
    - Z変数 15.3.12
- sqlda.h 15.1.3
- SQLDAToResultSetCursor() 5.11
- SQL記述子領域
    - SQLDA 13.10.1, 15.2.1
- SQLディレクティブ
    - CONTEXT USE 11.4.2.3, E.21
    - DECLARE DATABASE E.24
    - DECLARE STATEMENT E.25
    - DECLARE TABLE E.26
    - DECLARE TYPE E.27
    - TYPE E.75
    - VAR E.77
    - WHENEVER E.78
- SQLEnvGet() 5.11
- SQLLIBでのSQLEnvGet関数 5.9.1
- sqlerrd
    - コンポーネント 9.5.4, 9.6.3.6
- sqlerrd[2]コンポーネント 9.5.3, 9.6.3.6
    - N行またはフェッチされた行を戻す 8.4.3
    - DML文との併用 8.4.2
- sqlerrmcコンポーネント 9.6.3.4
- sqlerrmlコンポーネント 9.6.3.4
- SQLERROR
    - WHENEVERディレクティブ条件 E.78
- SQLERROR条件
    - WHENEVER文 9.8.1.2
    - 意味 9.8.1.2
- SQLErrorGetText() 5.11
- SQLExtProcError() 5.11, 7.8.3
- sqlglm() 9.7
- sqlglm()関数 9.7

        - 使用例 9.7
        - パラメータ 9.7
- sqlglmt()
    - 「SQLErrorGetText」を参照 5.11
- sqlgls()関数 9.9
    - 使用例 4.4.7
    - サンプル・プログラム 9.9.2
    - 「SQLLIB」を参照
        - SQLStmGetText関数 4.4.7
- sqlglst()関数
    - 「SQLStmtGetText」を参照 5.11
- sqlld2t()関数
    - 「SQLLDAGetName」を参照 5.11
- SQLLDAGetName 5.11
- sqlldat()関数
    - 「SQLCDAGetCurrent」を参照 5.11
- SQLLIB
    - 埋込みSQL 2.1.2
    - OCIの拡張相互運用性 5.8
    - 関数
        - SQLCDAFromResultSetCursor 5.11
    - SQLCDAGetCurrent関数 5.11
    - SQLColumnNullCheck関数 5.11
    - SQLDAFree関数 5.11
    - SQLDAToResultSetCursor関数 5.11
    - SQLEnvGet関数 5.9.1, 5.11
    - SQLErrorGetText関数 5.11
    - SQLExtProcError関数 5.11, 7.8.3
    - SQLLDAGetName関数 5.11
    - SQLNumberPrecV6関数 5.11
    - SQLNumberPrecV7関数 5.11
    - SQLRowidGet関数 5.11
    - SQLStmtGetText()関数 5.11
    - SQLSvcCtxGet関数 5.9.2, 5.11
    - SQLVarcharGetLength関数 4.4.6
    - パブリック関数の新しい名前 5.11
- SQLLIB関数
    - SQLSQLDAAlloc 5.11
    - SQLVarcharGetLength 5.11
- sqlnul()関数
    - 使用例 15.4.3
    - 用途 15.4.3
    - 構文 15.4.3
- sqlnult()関数
    - 「SQLColumnNullCheck()」を参照 5.11

- SQLNumberPrecV6 5.11
- SQLNumberPrecV7 5.11
- sqlpr2()関数 15.4.2.1
- sqlpr2t()関数
    - 「SQLNumberPrecV7」を参照 5.11
- sqlprc()関数 15.4.2.1
- sqlprct()関数
    - 「SQLNumberPrecV6」を参照 5.11
- SQLRowidGet() 5.11
- SQLSQLDAAlloc 5.11
- SQLSQLDAFree() 5.11
- SQLSTATE
    - クラス・コード 9.3.2
    - 宣言 9.3.1
    - Oracleエラーへのマッピング 9.3.2
    - ステータス・コード 9.3.2
    - 状態変数 9.3
    - 使用 9.3.3
    - 値 9.3.2
    - MODE=ANSIの使用 10.5.37
- SQL文
    - ALLOCATE E.4
    - ALLOCATE DESCRIPTOR TYPE E.5
    - CACHE FREE ALL E.6
    - CALL 7.7.2.3, E.7
    - CLOSE E.8
    - COMMIT E.15
    - 実行時の注意点 6.3
    - CONNECT E.16
    - CONTEXT ALLOCATE E.17
    - CONTEXT FREE E.18
    - CONTEXT OBJECT OPTION GET E.19
    - CONTEXT OBJECT OPTION SET E.20
    - DESCRIPTOR E.22をDEALLOCATE
    - DELETE E.28
    - DESCRIBE E.29
    - DESCRIBE DESCRIPTOR E.30
    - ENABLE THREADS E.31
    - 実行文とディレクティブの対比 2.1.1.1
    - EXECUTE E.33
    - EXECUTE... END-EXEC E.32
    - EXECUTE IMMEDIATE E.35
    - FETCH E.36
    - FETCH DESCRIPTOR E.37
    - トランザクションの定義と制御 3.4

    - カーソル操作用 6.3, 6.5
    - FREE E.38
    - INSERT E.40
    - LOB APPEND E.41
    - LOB ASSIGN E.42
    - LOB CLOSE E.43
    - LOB E.44COPY
    - LOB CREATE E.45
    - LOB DESCRIBE E.46
    - LOB DISABLE BUFFERING E.47
    - LOB ENABLE BUFFERING E.48
    - LOB ERASE E.49
    - LOB FILE CLOSE E.50
    - LOB FILE SET E.51
    - LOB FLUSH BUFFER E.52
    - LOB FREE TEMPORARY E.53
    - LOB LOAD E.54
    - LOB OPEN E.55
    - LOB READ E.56
    - LOB TRIM E.57
    - LOB WRITE E.58
    - OBJECT CREATE E.59
    - OBJECT DELETE E.60
    - OBJECT DEREF E.61
    - OBJECT FLUSH E.62
    - OBJECT GET E.63
    - OBJECT RELEASE E.64
    - OBJECT SET E.65
    - OBJECT UPDATE E.66
    - OPEN E.67
    - OPEN DESCRIPTOR E.68
    - パフォーマンス向上のための最適化 B.5
    - PREPARE E.69
    - REGISTER CONNECT E.70
    - ROLLBACK E.71
    - 実行のルール B.5
    - SAVEPOINT E.72
    - SELECT E.73
    - SET DESCRIPTOR E.74
    - 概要 E.1
    - タイプ 2.1.1.1
    - UPDATE E.76
- SQLStmtGetText 5.11
- SQLStmtGetText()関数 F.1
- SQLSvcCtxGet() 5.11

- SQLLIBでのSQLSvcCtxGet関数 5.9.2
- SQLVarcharGetLength 5.11
- sqlvcp()関数 F.1
- sqlvcp()関数, 「SQLLIB」を参照
    - SQLVarcharGetLength関数 4.4.6
- sqlvcpt()関数
    - 「SQLVarcharGetLength」を参照 5.11
- sqlwarn
    - フラグ 9.6.3.7
- SQLWARNING
    - WHENEVERディレクティブ条件 E.78
- SQLWARNING条件
    - WHENEVER文 9.8.1.1
    - 意味 9.8.1.1
- 標準ヘッダー・ファイル D.1.1
- 文レベルのロールバック
    - 説明 3.8.1
    - デッドロックの解消 3.8.1
- ステータス・コード
    - 意味 9.5.1
- STOPアクション
    - WHENEVER文 9.8.2.6
    - WHENEVERディレクティブ E.78
    - 結果 9.8.2.6
- ストアド・プロシージャ
    - プログラム例 7.7.2.1
- ストアド・サブプログラム
    - コール 7.7.2.1
    - 作成 7.7.1
    - パッケージとスタンドアロンの対比 7.7
    - ストアドとインラインの対比 7.7
- STRINGデータ型 4.1.2.5
- 文字列ホスト変数
    - 宣言 5.1.4
- 構造体コンポーネントの位置合せ D.1.4
- 構造体
    - 配列 8.10
    - ホスト変数 4.8
    - C言語, 使用 17.12
    - コレクション・オブジェクト・タイプ 18.2
    - REFのC言語の構造体の生成 17.13.1
    - ホスト変数としてのポインタ 4.9.3
- 構造体
    - ネスト不可 4.8.3
- 構造体

    - ホストにネスト不可 4.8.3
- 副問合せ
    - 定義 6.3.2.1
    - 例 6.3.2.1, 6.3.3
    - 使用 6.3.2.1
    - SET句での使用 6.3.3
    - VALUES句での使用 6.3.2.1
- SQLDAのS変数
    - 値の設定方法 15.3.7
    - 用途 15.3.7
- 記号
    - 定義 2.4.1
- 構文, 埋込みSQL 2.1.2
- 構文検査
    - SQLCHECKオプションを使用した制御 C.2
    - 定義 C.1
- 構文図
    - 説明 E.3
    - 読み方 E.3
    - 使用方法 E.3
    - 使用される記号 E.3
- SYS_INCLUDE
    - C++内のシステム・ヘッダー・ファイル12.2.4
- SYS_INCLUDE, INCLUDEプリコンパイラ・オプション 10.2.1
- SYS_INCLUDEプリコンパイラ・オプション 10.5.59
- システム構成ファイル 10.2.3
- システム構成ファイル D.1.10
- システム障害
    - トランザクションへの影響 3.5
- システム・グローバル領域(SGA) 7.7
- システム・ヘッダー・ファイル
    - 場所の指定 12.2.4
- システム固有のOracleマニュアル 1.8.11, 2.6, 3.2.2, 5.12.1.4, 20
- システム固有参照 4.1.2.3, 10.1.1, 10.2.2, 10.5.29, 10.5.59

T

- 表ロック
    - LOCK TABLEで取得 3.11.2
    - 影響 3.11.2
    - 行共有 3.11.2
    - 解除される場合 3.11.2
- 表
    - 行の挿入 E.40

    - ネストした 18.1.1
    - 行の更新 E.76
- 端末
    - コード体系 4.10
- 終了, プログラム
    - 正常と異常の対比 3.9
- スレッド E.17
    - コンテキストの割当て 11.4.2.2, E.17
    - 有効化 11.4.2.1, E.31
    - コンテキストの解放 11.4.2.4, E.18
    - コンテキストの使用 11.4.2.3
- THREADS
    - プリコンパイラ・オプション 10.5.60, 11.4.1
- TIMESTAMPデータ型 4.1.3.3
- TIMESTAMP WITH LOCAL TIME ZONEデータ型 4.1.3.5
- TIMESTAMP WITH TIME ZONEデータ型 4.1.3.4
- TO句
    - ROLLBACK文 E.71
- Toolset
    - Oracle 20.5
- TO SAVEPOINT句
    - ROLLBACK文 3.8
    - 用途 3.8
    - 制限 3.8
- トレース機能
    - 機能 B.5.2
    - パフォーマンスの向上 B.5.2
- トランザクション処理
    - 概要 2.1.10
    - 使用する文 2.1.10
- トランザクション処理モニター 5.12
- トランザクション
    - コミット E.15
    - 目次 2.1.10, 3.5
    - 定義 2.1.10
    - 説明 3.4
    - 分散 E.71
    - 障害 3.5
    - データベースの保護 3.4
    - ガイドライン 3.14.1
    - 開始方法 3.5
    - 終了方法 3.5
    - 永続的なものにする 3.6
    - 読取り専用 3.10
    - ロールバック E.71

        - セーブポイントによる副分割 3.7
        - 終了 3.6
        - 取消し 3.8
        - 一部の取消し 3.7
        - 自動ロールバックされる場合 3.5
- オブジェクトの一時コピー 17.3.1
- 一時オブジェクト 17.3.1
- TRANSITIVE OTTパラメータ 19.5.2.12
- 切り捨てられた値
    - 検出 6.2, 7.4.2
- 切捨てエラー
    - 生成時 6.2.5
- チューニング, パフォーマンスB.1
- 2タスク
    - リンク 2.6
- TYPE_CODE
    - プリコンパイラ・オプション 10.5.61
- 実行時のタイプ・チェック 17.8.7
- TYPEディレクティブ
    - 例 E.75
- 型の継承 17.1.3
    - 例 17.10
    - IS OF type演算子 17.1.3
        - 例 17.1.3
    - TREAT演算子 17.1.3
        - 例 17.1.3
- TYPE SQLディレクティブ E.75

U

- トランザクションの取消し E.71
- Unicodeキャラクタ・セット 5.1.5
- 共用体
    - ホスト構造体内でネストできない 4.8.3
    - ホスト構造体として許されない 4.8.3
- ユニバーサルROWID 4.1.2.9
- UNIX
    - Pro*Cアプリケーションのリンク 1.8.11
- UNSAFE_NULLプリコンパイラ・オプション 10.5.62
- UNSIGNEDデータ型 4.1.2.14
- UPDATE SQL文 E.76
- UPDATE文
    - 埋込みSQLの例 E.76
    - 例 6.3.2.1

    - 用途 6.3.2.1
    - SET句を含む 6.3.3
    - ホスト配列の使用 8.6
    - WHERE句を含む 6.3.3
- 更新
    - 表およびビューの行 E.76
- 使用
    - スレッド・コンテキスト 11.4.2.3, E.21
- ユーザー構成ファイル
    - プリコンパイラ・オプションの設定 10.2.3
- ユーザー定義のレコード 7.1.7
- ユーザー定義のストアド・ファンクション
    - WHERE句で使用 6.3.5
- ユーザー定義型の同値化 E.75
- ユーザー・イグジット
    - SQL*Formsトリガーからのコール 20.6
    - 一般的な使用方法 20.2
    - 例 20.9
    - 使用可能な文の種類 20.4
    - IAPへのリンク 20.13
    - リターン・コードの意味 20.8
    - 命名 20.14.1
    - パラメータの引渡し 20.7
    - 変数の要件 20.4.1
    - GENXTBフォームの実行 20.12
    - GENXTBユーティリティの実行 20.12
    - WHENEVER文の使用方法 20.8.2
- ユーザー・イグジット D.1.13
- USERIDオプション
    - 必要な場合 10.5.63
- USERID OTTパラメータ 19.5.2.1
- USERIDプリコンパイラ・オプション 10.5.63
    - SQLCHECKオプションで使用 C.3.1.1
- ユーザー名
    - 定義 3.1
- ユーザー・セッション
    - 定義 3.3
- USING句
    - CONNECT文 3.2.4.1
    - EXECUTE文 13.8.1
    - FETCH文 E.36
    - OPEN文 E.67
    - 用途 13.8.1
- コレクション型の使用 17.13
- C言語の構造体の使用 17.12

- dbstringの使用
  - Oracle Netデータベース指定の文字列 E.16
- 埋込みSQLでのREFの使用 17.13.3

V

- V7
  - DBMSオプションの値 10.5.15, 10.5.16
- VALUES句
  - INS 6.3.2
  - 埋込みSQL INSERT文 E.40
  - INSERT文 E.40
  - 副問合せの使用 6.3.2.1
- VARCHAR
  - 配列 8.2
- VARCHAR2データ型 4.1.2.1, 5.3.3
- VARCHARデータ型 4.1.2.8
- VARCHAR プリコンパイラ・オプション 10.5.65
- VARCHAR疑似型
  - PL/SQLで使用するための要件 7.3.3
- VARCHAR変数
  - 利点 4.4.1
  - 宣言 4.4.1
  - 長さメンバー 4.4.1
  - 参照による関数への受渡しが必要 4.4.5
  - 長さの指定 4.4.1
  - 構造 4.4.1
  - 長さの定義にマクロを使用 5.4.1
  - 文字配列と対比 5.1.4.2
- VARディレクティブ
  - 例 E.77
- 変数 2.1.5
  - カーソル 4.5
  - ホスト 18.2.1
  - インジケータ 18.2.1
- VARNUMデータ型 4.1.2.6
- VARRAWデータ型 4.1.2.12
- VARRAY
  - 作成 18.1.2
- VAR SQLディレクティブ E.77
- VAR文
  - 構文 5.3.1, 5.3.2
- 可変長配列 18.1.2
- VERSIONプリコンパイラ・オプション 10.5.66, 17.8.1

- ビュー
  - 行の挿入 E.40
  - 行の更新 E.76
- VMS
  - プリコンパイラ・アプリケーションのリンク 1.8.11
- SQLDAのV変数
  - 値の設定方法 15.3.2
  - 用途 15.3.2

W

- 警告フラグ
  - エラー・レポートでの使用 9.5.2
- WHENEVERディレクティブ
  - 例 E.78
- WHENEVER SQLディレクティブ E.78
- WHENEVER文
  - SQLCAの自動チェック 9.8.1
  - CONTINUEアクション 9.8.2.1
  - DOアクション 9.8.2.2
  - DO BREAKアクション 9.8.2.3
  - DO CONTINUEアクション 9.8.2.4
  - 例 9.8.3
  - GOTOアクション 9.8.2.5
  - ガイドライン 9.8.6
  - アドレス指定可能度の維持 9.8.6.4
  - NOT FOUND条件 9.8.1.3
  - 概要 2.1.11
  - スコープ 9.8.5
  - SQLERROR条件 9.8.1.2
  - SQLWARNING条件 9.8.1.1
  - STOPアクション 9.8.2.6
  - ユーザー・イグジットでの使用方法 20.8.2
  - 無限ループの回避 9.8.6.3
  - データの終わり条件に対処 9.8.6.2
  - 配置する場所 9.8.6.1
- WHERE句
  - ホスト配列 8.9
  - 省略した場合 6.3.5
  - DELETE文 6.3.4
  - SELECT文 6.3.1
  - UPDATE文 6.3.3
  - 用途 6.3.5
  - 検索条件 6.3.5

- WHERE CURRENT OF句
  - CURRENT OF句 6.9
- WITH HOLD
  - DECLARE CURSOR文の句 6.5.1
- WORKオプション
  - COMMIT文 E.15
  - ROLLBACK文 E.71

X

- X/Open 5.12
  - アプリケーション開発 5.12
- XAインタフェース 5.12
- SQLDAのX変数
  - 値の設定方法 15.3.10
  - 用途 15.3.10

Y

- SQLDAのY変数
  - 値の設定方法 15.3.11
  - 用途 15.3.11

Z

- SQLDAのZ変数
  - 値の設定方法 15.3.12
  - 用途 15.3.12