Pro*COBOL プリコンパイラ

プログラマーズ・ガイド

リリース 8.1

ORACLE

Pro*COBOL プリコンパイラ・プログラマーズ・ガイド リリース 8.1

部品番号：A62771-1

第 1 版 1999 年 5 月（第 1 刷）

原本名：Pro*COBOL Precompiler Programmer’s Guide, Release 8.1.5

原本部品番号：A68023-01

原本著者：Jack Melnick, Tom Portfolio

グラフィック・デザイナ：Valarie Moore

原本協力者：Michael Chiocca, Nancy Ikeda, Maura Joglekar, Thomas Kurian, Shiao-yen Lin, Diana Lorentz, Lee Osborne, Jacqui Pons, Ajay Popat, Pamela Rothman, Gael Turk Stevens

目次

2 プリコンパイラの概念

3 データベースの概念

4 データ型とホスト変数

5 埋込み SQL

6 埋込み PL/SQL

7 ホスト表

8 エラー処理と診断

9 Oracle 動的 SQL

10 ANSI 動的 SQL

11 Oracle 動的 SQL: 方法 4

12 ユーザー・イグジット

13 ラージ・オブジェクト（LOB）

14 プリコンパイラのオプション

A 新機能

B　オペレーティング・システムの依存性

C　予約語、キーワードおよび名前領域

D　パフォーマンス・チューニング

E 構文および意味検査

F 埋込み SQL 文とプリコンパイラ・ディレクティブ

はじめに

このマニュアルは、Oracle Pro*COBOL プリコンパイラの総合的なユーザーズ・ガイドおよびリファレンスです。ここでは、データベース言語 SQL と PL/SQL を使って Oracle データへのアクセスと操作を行う COBOL プログラムの開発方法を説明します。SQL および PL/SQL の詳細は、『Oracle8*i* SQL リファレンス』および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

この「はじめに」の構成は、次のとおりです。

- このマニュアルの内容
- 対象読者
- このマニュアルの構成
- 表記上の規則
- サンプル・プログラム
- 規格への準拠

このマニュアルの内容

このマニュアルでは、Oracle Pro*COBOL プリコンパイラおよび埋込み SQL をアプリケーション開発のプロセスに利用する方法を紹介します。また、Oracle の機能を活用するアプリケーションの設計および開発方法を説明します。さらに、できるだけ短時間で埋込み SQL プログラムの作成方法を習得できるように援助します。

このマニュアルの大きな特徴の 1 つは、Pro*COBOL および埋込み SQL を最大限に活用することに重点を置いていることです。これらのツールを最大限に活用するために、このマニュアルではプログラムのパフォーマンスを向上する方法を含めた " 作業のこつ " をすべて掲載しています。また、埋込み SQL についての理解を深め、その有用性を確認できるように、多数のプログラム例を掲載しています。

> **注意** : このマニュアルでは、インストレーションの指示またはシステム固有の情報は説明していません。これらの詳細は、システム固有の Oracle マニュアルを参照してください。
>
> Oracle7 から Oracle8*i* へアプリケーションを移行する方法は、『Oracle8*i* 移行ガイド』を参照してください。

対象読者

このマニュアルは、Oracle8*i* 環境で動作する新規の COBOL アプリケーションを開発する場合や、既存のアプリケーションを Oracle8 環境用に変換する場合に役立ちます。また、このマニュアルは特にプログラマを対象として記述されていますが、Pro*COBOL について総合的に解説していますので、システム・アナリストやプロジェクト・マネージャ、埋込み SQL アプリケーションに関心のあるその他のユーザーにも役立つ内容となっています。

このマニュアルを有効に利用するには、次の知識が必要です。

- COBOL でのアプリケーションのプログラミング
- SQL データベース言語
- Oracle8*i* の概要と用語

このマニュアルの構成

各章および付録は、次のような内容になっています。

第 1 章「概要」

この章では、Pro*COBOL の概要を説明します。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上での Pro*COBOL の役割と、Pro*COBOL の主要な利点および機能について説明します。

第 2 章「プリコンパイラの概念」

この章では埋込み SQL プログラムの動作について説明します。さらに Pro*COBOL でのプログラミングの指針についても説明します。コンパイルの問題についても説明し、またこのマニュアルで使われる Oracle 表のサンプルおよび最初のデモ・プログラム SAMPLE1.PCO も紹介しています。

第 3 章「データベースの概念」

この章ではトランザクションの処理について説明します。データベースの整合性を維持するための基本的な技法を学びます。単一のデータベースおよび複数の分散データベースに接続する方法についても学びます。

第 4 章「データ型とホスト変数」

内部データ型および外部データ型の長さの定義について説明します。COBOL プログラムでそれらのデータ型を使う方法についても学びます。さらに、実行時コンテキストと ROWID、各国語サポート、データ型変換およびデータ型同値化について説明します。（サンプル・プログラム付き）

第 5 章「埋込み SQL」

埋込み SQL プログラミングの基本事項を説明します。ホスト変数および標識変数、カーソル、カーソル変数の使用方法と、Oracle データの insert および update、select、delete を行う基本的な SQL コマンドの使用方法を説明します。

第 6 章「埋込み PL/SQL」

この章では、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について説明します。ホスト変数、標識変数、カーソル、PL/SQL または Java のストアド・サブプログラム、ホスト表、動的 PL/SQL を使った PL/SQL の使用方法について学びます。

第 7 章「ホスト表」

この章では、ホスト（COBOL）表を使用してプログラムのパフォーマンスを改善する方法を説明します。ここで紹介するのは、配列を使って Oracle データを操作する方法、単一の SQL 文を使って配列内のすべての要素を処理する方法、処理対象となる配列の要素数を制限する方法です。

第 8 章「エラー処理と診断」

この章では、エラーの報告および回復ついて詳しく説明します。状態変数 SQLSTATE および SQLCA 構造体、WHENEVER 文を使用してエラーを検出、処理する方法を学びます。また、ORACA を使用して問題を診断する方法も紹介します。

第 9 章「Oracle 動的 SQL」

動的 SQL の利点を活用する方法について説明します。ユーザーが実行時に SQL 文を対話形式で構築できるように、単純なものから複雑なものまで、柔軟なプログラムを作成する 3 つの方法を説明します。

第 10 章「ANSI 動的 SQL」

ANSI 動的 SQL の方法 4 について説明します。この方法ではすべての Oracle データ型がサポートされています。以前の Oracle 方法 4 では、カーソル変数、グループ項目の表、DML 戻り句、LOB はサポートされていません。ANSI 方法 4 では、記述子領域をメモリーに設定する埋込み SQL 文を使います。新規のすべてのアプリケーションに対して ANSI SQL を使うようにしてください。

第 11 章「Oracle 動的 SQL: 方法 4」

この章では、動的 SQL 方法 4 を使う既存のアプリケーションをメンテナンスする方法について説明します。説明には多数の例を使用します。

第 12 章「ユーザー・イグジット」

この章では、SQL*Forms または Oracle Forms 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。まず、Forms アプリケーションとユーザー・イグジットとのインタフェースとなるコマンドについて説明します。次に、Forms ユーザー・イグジットを作成してリンクさせる方法を説明します。

第 13 章「ラージ・オブジェクト（LOB）」

この章では、ラージ・オブジェクト・データ型（BLOB、CLOB、NCLOB および BFILE）について説明します。OCI および PL/SQL に類似した機能を提供する埋込み SQL コマンドについて説明し、サンプル・コードで使います。

第 14 章「プリコンパイラのオプション」

この章では、Pro*COBOL プリコンパイラを実行するための要件と、プリコンパイラ・オプションの一覧について詳しく説明します。プリコンパイルの過程、Pro*COBOL コマンドの実行方法、各種プリコンパイラ・オプションの指定方法を学びます。

付録 A「新機能」

Pro*COBOL のリリース 8.1 および 8.0 の両方に導入された機能の改善点と新機能について説明します。

付録 B「オペレーティング・システムの依存性」

Pro*COBOL のプログラミングの詳細については、システムによって異なる部分があります。したがって、システム固有の情報は、必要に応じて他のマニュアルを参照してください。この付録には、外部の問題をすべてまとめてあるので活用してください。

付録 C「予約語、キーワードおよび名前領域」

この付録では、Pro*COBOL にとって特別な意味を持つ予約語の表を紹介します。Oracle ライブラリ用に予約されている名前領域を示します。

付録 D「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を改善させる手軽な方法をいくつか紹介します。

付録 E「構文および意味検査」

埋込み SQL 文および PL/SQL ブロックに対して行う構文および意味の検査の種類と範囲を、SQLCHECK オプションを使用して制御する方法を説明します。

付録 F「埋込み SQL 文とプリコンパイラ ・ディレクティブ」

この付録では、プリコンパイラ・ディレクティブ、埋込み SQL コマンド、Oracle 埋込み SQL の拡張要素について説明します。文およびディレクティブごとに、その目的、前提条件、構文図、キーワード、パラメータ、使用方法、例、関連項目を示します。

表記上の規則

説明の部分では、データベース・オブジェクト、プリコンパイラ・オプションおよび SQL キーワードには大文字のアルファベットが使われます。変数と定数は、サンプル・コードと同様、クーリエ・フォントで示されます。

表記上の規則

このマニュアルでは、次のような表記上の規則を使用しています。

< >	山カッコで囲まれたものは構文要素の名前を示します。イタリックが使われる場合もあります。
.	コンポーネント名とオブジェクト名を区切り、参照を修飾します。
..	2 つのドットは、ある範囲内の最低値と最高値を区切ります。
...	説明内容とは直接関係のない文または句が省略されていることを示します。
#	データベースの列の内容を参照する際に、テキスト内の空白を表します。

構文の説明

埋込み SQL の構文は、バッカス正規形（BNF）方式で表します。BNF では、次の記号を使用します。

[ ]	大カッコは、オプションの項目を囲みます。
{ }	中カッコで囲まれたものは、そのうちの 1 つだけを選択しなければならない項目を示します。
|	垂直バーは、大カッコまたは中カッコ内の項目を区切るために使います。
...	省略記号は、直前のパラメータを繰返し指定できることを示します。

サンプル・プログラム

このマニュアルには、読者が独自のプログラムを作成する際の参考用に、いくつかの Pro*COBOL プログラムが掲載してあります。これらのプログラムにより、Pro*COBOL プログラミングの基本的な概念と機能が理解でき、同時に SQL の機能と柔軟性を最大限に活用するための技法が習得できるようになっています。

このマニュアルに掲載されている各サンプル・プログラムの全体を、デモ・ディレクトリでオンラインで入手できます。ただし、厳密なファイル名はシステムによって異なります。厳密なファイル名は、使用しているシステム固有の Oracle マニュアルを参照してください。このマニュアルでは、Sun SPARC Solaris 用に開発されたサンプル・コードを掲載しています。

規格への準拠

SQL はリレーショナル・データベース管理システムの標準言語になりました。ここでは、次の機関が設定した最新の SQL 規格に対する Pro*COBOL プリコンパイラの準拠状況を説明します。

- 米国規格協会（ANSI）
- 国際標準化機構（ISO）
- 米国連邦情報・技術局（NIST）

これらの機関が承認する SQL の定義は、次の文書に記載されています。

- ANSI 規格 X3.135-1992、『データベース言語 SQL』
- ANSI 規格 X3.168-1992、『データベース言語埋込み SQL』
- ISO / IEC 規格 9075:1992、『データベース言語 SQL』
- NIST 規格 FIPS PUB 127-2、『データベース言語 SQL』

要件

ANSI X3.135-1992（SQL92 と略す）は、段階別に実現するための "SQL 言語の準拠 " を指定し、次の 3 種類の言語レベルを定義します。

- Full SQL
- Intermediate SQL（Full SQL のサブセット）
- Entry SQL（Intermediate SQL のサブセット）

SQL 準拠の処理系では、少なくとも Entry SQL がサポートされていなければなりません。

ANSI X3.168-1992 は、COBOL-74 や COBOL-85 などの標準プログラミング言語で作成されたアプリケーション・プログラムに SQL 文を埋め込むための構文および方法を指定しています。

ISO / IEC 9075-1992 は、ANSI 規格を完全に採用しています。

FIPS PUB 127-2 は、連邦政府が使用するために購入する RDBMS に適用されるもので、やはり ANSI 規格を採用しています。また、この規格では、データベース構文のための最小限のサイズ・パラメータを指定しています。さらに、"FIPS フラガー " により ANSI 拡張部分を識別することを定めています。

ANSI 規格書を入手するには、次の宛先に書面で問い合せてください。

American National Standards Institute

1430 Broadway

New York, NY 10018, USA

ISO 規格書を入手するには、ISO 加盟団体の国内事務所に書面で問い合せてください。NIST 規格書を入手するには、次の宛先に書面で問い合せてください。

National Technical Information Service

U.S. Department of Commerce

Springfield, VA 22161, USA

準拠性

Pro*COBOL プリコンパイラは ANSI および ISO、NIST 規格に 100% 準拠しています。必要に応じて、Entry SQL のサポートおよび FIPS フラガーの提供を行います。

FIPS フラガー

FIPS PUB 127-1 からの引用 :

" この規格で定められていない付加的な機能を提供するインプリメンテーションでは、非準拠の方法で処理される可能性のある非準拠 SQL 言語または準拠 SQL 言語にフラグを付けるオプションも提供するものとする。"

この要件を満たすため、Pro*COBOL プリコンパイラでは ANSI 拡張要素にフラグを付ける FIPS フラガーを提供しています。拡張要素とは、ANSI の形式または構文規則（権限付与規則は除く）に違反する SQL 要素を指します。標準 SQL への Oracle 拡張要素の一覧は、『Oracle8*i* SQL リファレンス』を参照してください。

FIPS フラガーを使うと、次の SQL 要素を識別できます。

- ANSI 準拠の環境にアプリケーションを移した場合に修正の必要が生じることになる、非準拠 SQL 要素

- 別の処理環境では異なる動作が予想される、準拠 SQL 要素

つまり、FIPS フラガーは移植性のあるアプリケーションを開発するのに役立ちます。

FIPS オプション

FIPS フラガーの制御には、FIPS というオプションを使用します。FIPS フラガーを使用可能にするには、インラインまたはコマンド行で FIPS=YES を指定します。コマンド行オプション FIPS の詳細は、14-21 ページの「FIPS」を参照してください。

準拠の証示

Pro*COBOL プリコンパイラの ANSI Entry SQL に対する準拠性のテストが、約 300 のテスト・プログラムからなる *SQL Test Suite* を使用して、NIST によって行われました。このテストでは、特に COBOL に組み込まれる SQL 規格に準拠しているかどうかが調べられました。その結果、Pro*COBOL プリコンパイラは ANSI 規格に 100% 準拠していることが検証されました。

テストの詳細は、次の宛先に書面で申請してください。

National Computer Systems Laboratory

Attn.: Software Standards Testing Program

National Institute of Standards

MIA/SPIRIT

Pro*COBOL プリコンパイラが提供するマルチバイト・キャラクタ・データの各国語サポート（NLS）は、Multivendor Integration Architecture（MIA）仕様バージョン 1.3 および Service Providers Integrated Requirements for Information Technology（SPIRIT）仕様第 2 号に準拠しています。

1

概要

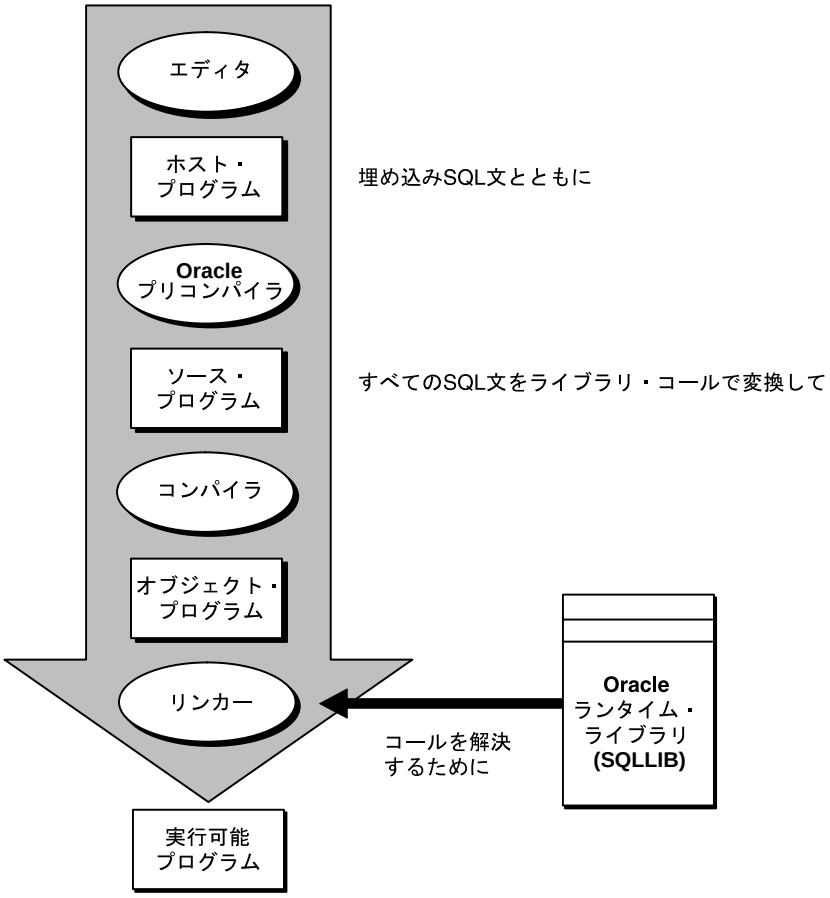
この章では、Pro*COBOL プリコンパイラの概要を説明します。Oracle データを操作するアプリケーション・プログラムを開発する際のその役割を理解すれば、Pro*COBOL プリコンパイラを使ってアプリケーションで何をできるかがわかります。

- Pro*COBOL とは
- Pro*COBOL プリコンパイラを使用する利点
- SQL を使用する利点
- PL/SQL を使用する利点
- Pro*COBOL の機能

Pro*COBOL とは

Pro*COBOL プリコンパイラは、SQL 文をホストの COBOL プログラムに埋め込むことを可能にするプログラミング・ツールです。図 1-1 に示すように、プリコンパイラはホスト・プログラムを入力として受け取り、埋込み SQL 文を標準 Oracle ランタイム・ライブラリ・コールに翻訳して、ソース・プログラムを生成します。このソース・プログラムは、通常の方法でコンパイル、リンクおよび実行が可能です。

図 1-1　埋込み SQL プログラムの開発



使用できる言語

Oracle プリコンパイラは次の言語で使用できます。（ただし、すべてのシステムで使用できるとは限りません。）

- C / C++
- COBOL
- FORTRAN

Pro*Pascal、Pro*ADA および Pro*PL / I が将来リリースされることはありません。ただし、Oracle では、Pro*FORTRAN のバグ・レポートとその修正が行われるごとに、パッチ・リリースの提供を継続して行います。

Pro*COBOL プリコンパイラを使用する利点

Pro*COBOL プリコンパイラを使うと、アプリケーション・プログラムに強力で柔軟な SQL を組み込めます。SQL 文を COBOL に埋め込むことができます。便利で使いやすいインタフェースによって、アプリケーションから直接 Oracle にアクセスできるようになります。

他の多くのアプリケーション開発ツールとは違い、Pro*COBOL 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、"1234" の CHAR 値を C の short 値に変換できます。ユーザーとの対話を介さずに、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*COBOL を使ってアプリケーションを微調整できます。リソースの使用、SQL 文の実行およびさまざまなランタイム標識の状況を詳細に監視できます。得られた情報に基づいて、最大のパフォーマンスが得られるようにプログラムのパラメータを調整できます。

SQL を使用する利点

Oracle データにアクセスし、それを操作するには、SQL が必要です。SQL を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んで使用するかは、作業の内容によって決まります。COBOL のプロシージャ処理能力を必要とする作業や、定期的に実行する作業の場合は、埋込み SQL を使用します。

SQL は、その柔軟かつ強力な特性と学習しやすさによって、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、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によって、Oracle データを一度に 1 行または複数行ずつ容易に操作できます。

任意の（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、次の操作が可能です。

- データベース表の動的な CREATE（作成）、ALTER（変更）および DROP（削除）。

- データ行の SELECT（選択）、INSERT（挿入）、UPDATE（更新）および DELETE（削除）。
- トランザクションの COMMIT（コミット）や ROLLBACK（ロールバック）。

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 を使って対話的に SQL 文をテストできます。通常、対話型 SQL から埋込み SQL に切り替えるためにはわずかの変更で済みます。

PL/SQL を使用する利点

SQL を拡張した PL/SQL は、プロシージャ構造および変数宣言、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 ブロック内では、SQL および PL/SQL の拡張機能のすべてを使用できます。

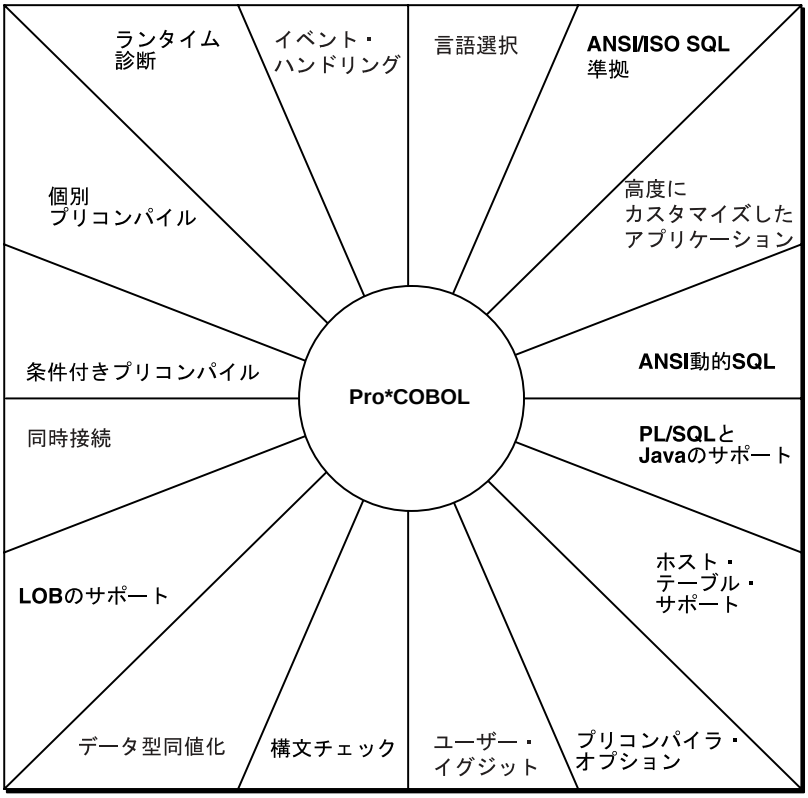
埋込み PL/SQL の主な利点はパフォーマンスの向上です。SQL と異なり、PL/SQL では、SQL 文を論理的にグループ化し、1 文ずつではなくブロック単位で Oracle に送ることができます。この結果、ネットワークの通信量および処理のオーバーヘッドが減少します。

アプリケーション・プログラムに PL/SQL を埋め込む方法を含む PL/SQL の詳細は、第 6 章の「埋込み PL/SQL」を参照してください。

Pro*COBOL の機能

図 1-2 に示すように、Pro*COBOL は効率と信頼性の高いアプリケーション開発を支援する多くの機能と利点を提供します。

図 1-2　機能および利点



たとえば、Pro*COBOL プリコンパイラを使って次のようなことができます。

- COBOL を使ったアプリケーション開発
- ANSI / ISO 埋込み SQL 規格への準拠
- 実行時に、COBOL プログラム内の有効な任意の SQL 文の受入れおよびビルドを可能にする、ANSI 動的 SQL 方法 4 の利用
- 高度にカスタマイズされたアプリケーションの設計と開発
- Oracle8*i* の内部データ型と COBOL データ型との間の自動相互変換
- COBOL 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による、パフォーマンスの改善
- 有用なプリコンパイラ・オプションの指定およびプリコンパイル中の値変更

- データ型の同値化機能を使った、Oracle8*i* による入力データの解釈と出力データの形式設定の制御
- 複数のプログラム・モジュールを個別にプリコンパイルしてからリンクすることにより、1 つの実行可能プログラムを作成する機能
- 埋込み SQL データを操作する文および PL/SQL ブロックの構文と意味の検査
- Net8 を使用した、複数ノード上の Oracle8*i* データベースへの同時アクセス
- 入力および出力プログラム変数での配列の使用
- 異なった環境下でのホスト・プログラムの実行を可能にする、コード・セクションの条件付きプリコンパイル
- 高級言語で記述されたユーザー・イグジットを介した、Oracle Forms や Oracle Reports などのツールとのインタフェース
- ANSI 準拠の状態変数 SQLSTATE と SQLCODE、または SQL コミュニケーション領域（SQLCA）と WHENEVER 文（あるいはその両方）を使用した、エラーと警告の処理
- Oracle コミュニケーション領域（ORACA）が提供する拡張セット診断プログラムの使用
- ラージ・オブジェクト（LOB）データベース型へのアクセス

2

プリコンパイラの概念

この章では埋込み SQL プログラムの動作について説明します。重要な用語の定義、基本概念の説明および使用上の規則を示します。

この章の構成は、次のとおりです。

- 埋込み SQL プログラミングの基本概念
- プログラミング・ガイドライン
- 宣言文
- ネストされたプログラム
- 条件付きプリコンパイル
- 分割プリコンパイル
- コンパイルとリンク
- サンプル表
- サンプル・プログラム : SAMPLE1.PCO

埋込み SQL プログラミングの基本概念

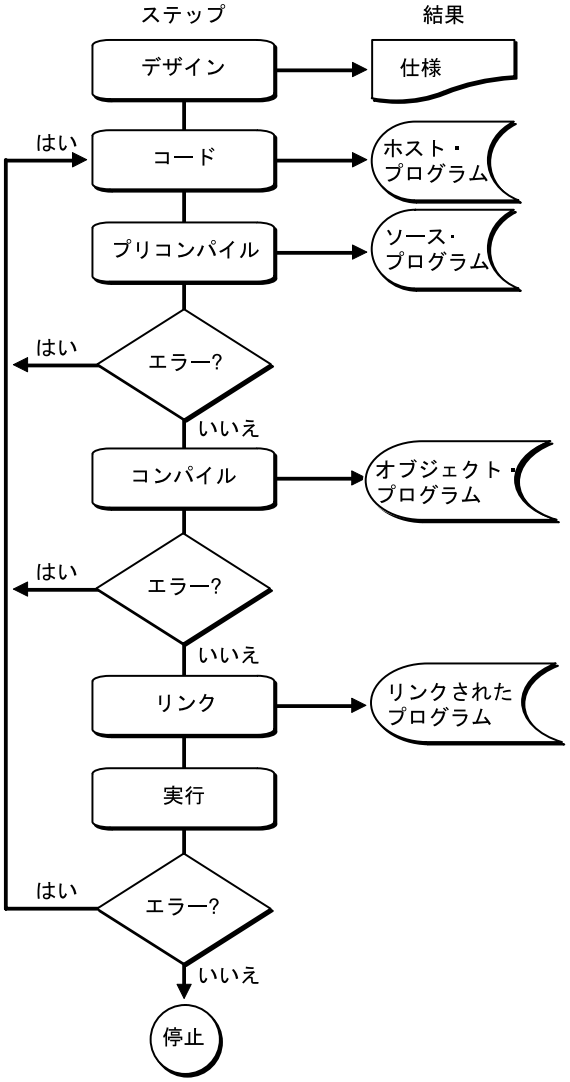
この項では、後の各章で説明する内容の基本概念について学習します。

埋込み SQL アプリケーションの開発ステップ

プリコンパイルを実行すると、通常どおりにコンパイルできるソース・ファイルが生成されます。プリコンパイルを行うと従来の開発過程より 1 ステップ処理が増えますが、これによって柔軟性に富んだアプリケーションを開発できるという大きな利点があります。

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 アプリケーションの開発プロセス



埋込み SQL 文

埋込み SQL という用語はアプリケーション・プログラム内に記述されている SQL 文を示します。アプリケーション・プログラムは、その中に SQL 文を包含していることからホスト・プログラムと呼ばれます。また、アプリケーション・プログラムの記述に使用される言語はホスト言語と呼ばれます。たとえば、Pro*COBOL では COBOL ホスト・プログラムに SQL 文を埋め込むことができます。

Oracle データの操作や問合せには、INSERT 文、UPDATE 文、DELETE 文および SELECT 文を使用します。INSERT はデータベースの表へのデータの行の追加、UPDATE は行の変更、DELETE は不要な行の削除、SELECT は検索条件を満たす行の検索を行います。

アプリケーション・プログラム内では、SQL 文だけが有効であり、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。）

実行文および宣言文

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、実行文と宣言文の 2 種類があります。

実行 SQL 文は、データベースのコールを生成します。実行 SQL 文には、ほとんどすべての問合せ、DML（データ操作言語）文、DDL（データ定義言語）文、DCL（データ制御言語）文が含まれます。

一方、宣言文では SQLLIB 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、ディレクティブとも呼ばれます。宣言文は Oracle オブジェクトおよび通信領域、SQL 変換を定義するために使います。宣言文は、COBOL の宣言を記述できる位置であればどこでも記述できます。

付録 F の「埋込み SQL 文とプリコンパイラ ・ディレクティブ」に、重要な文およびディレクティブを示します。表 2-1 に、いくつかの埋込み SQL 文の例を分類して示します。（すべてではありません）

表 2-1 埋込み SQL 文

宣言 SQL 文	
文	**用途**
ARRAYLEN*	PL/SQL でのホスト表の使用
BEGIN DECLARE SECTION*	ホスト変数の宣言
END DECLARE SECTION*	
DECLARE*	Oracle オブジェクトの命名
INCLUDE*	ファイルへの複写

表 2-1 埋込み SQL 文 (続き)

VAR*	変数の同値化
WHENEVER*	ランタイム・エラーの処理
実行可能 SQL 文	
文	用途
ALLOCATE*	Oracle データの定義および制御
ALTER	
ANALYZE	
AUDIT	
COMMENT	
CONNECT*	
CREATE	
DROP	
GRANT	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	
CLOSE*	Oracle データの問合せおよび操作
DELETE	
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	
UPDATE	
COMMIT	トランザクションの処理
ROLLBACK	
SAVEPOINT	
SET TRANSACTION	

表 2-1 埋込み SQL 文 (続き)

DESCRIBE*	動的 SQL の使用
EXECUTE*	
PREPARE*	
ALTER SESSION	セッションの制御
SET ROLE	

* には、対話形式のものはありません。

埋込み SQL の構文

アプリケーション・プログラムの中には、SQL 文とホスト言語の文を自由に混在させることができ、SQL 文でホスト言語の変数を使用できます。ホスト・プログラムに SQL 文を記述するために特に必要なことは、SQL 文をワード EXEC SQL で開始し、トークン END-EXEC で終了することだけです。Pro*COBOL が、実行 EXEC SQL 文をすべてランタイム・ライブラリ SQLLIB のコールに変換します。

大部分の埋込み SQL 文がそれに対応する対話型文と違う点は、新しい句が追加されていることか、プログラム変数が使用されていることだけです。次の対話型 ROLLBACK 文と埋込み ROLLBACK 文を比較してください。

```
ROLLBACK WORK;           -- interactive

* embedded
     EXEC SQL
         ROLLBACK WORK
     END-EXEC.
```

SQL 文の後にはピリオドやその他の終了記号を置くことができます。次は、どちらも正しい SQL 文です。

```
     EXEC SQL ... END-EXEC,
     EXEC SQL ... END-EXEC.
```

静的 SQL 文と動的 SQL 文

大部分のアプリケーション・プログラムは、静的 SQL 文および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、各 SQL 文およびトランザクションの構成が実行前にわかっています。つまり、どの SQL コマンドが発行されるか、どのデータベースの表が変更されるか、どの列が更新されるかなどが事前にわかっています。詳細は第 5 章の「埋込み SQL」を参照してください。

しかし、実行時に有効な SQL 文を受け入れて処理する必要のあるアプリケーションもあります。この場合は、関係する SQL コマンドおよびデータベースの表、列は実行時までわからない場合もあります。

動的 SQL は、プログラムの実行時に SQL 文を受け入れさせるか生成させて、データ型の変換を明示的に管理する高度なプログラミング技術です。詳細は第 9 章の「Oracle 動的 SQL」、第 10 章の「ANSI 動的 SQL」および第 11 章の「Oracle 動的 SQL: 方法 4」を参照してください。

埋込み PL/SQL ブロック

Pro*COBOL は、PL/SQL ブロックを単一の埋込み SQL 文のように扱います。したがって、PL/SQL ブロックは、アプリケーション・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 をホスト・プログラム内に埋め込むには、PL/SQL と共有する変数を宣言し、キーワード EXEC SQL EXECUTE および END-EXEC を使って PL/SQL ブロックを囲むだけです。

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ので、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。PL/SQL についての詳細は、第 6 章の「埋込み PL/SQL」を参照してください。

ホスト変数と標識変数

ホスト変数とは、ホスト言語で宣言され、Oracle との間で共有される（つまり、ユーザー・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表変数あるいはグループ項目です。ホスト変数は Oracle とプログラムとの間の通信を仲介します。

データベースにデータを渡すときは入力ホスト変数を使い、データベースからプログラムにデータおよび状態情報を渡すときは出力ホスト変数を使います。

ホスト変数は、式を使用できる位置であればどこに使用してもかまいません。しかし SQL 文では、ホスト変数にコロン（:）の接頭辞を付けなければ、データベース・スキーマ名からホスト変数を分離することはできません。

任意のホスト変数に任意指定の標識変数を対応付けることができます。標識変数とは、ホスト変数の値または条件を示す整変数です。NULL とは、値がないか、未知であるか、または適用不能な値です。標識変数を使うと、入力ホスト変数に NULL を割り当てたり、出力変数の NULL または出力文字ホスト変数の切り捨てられた値を検出したりできます。

ホスト変数の禁止事項は、次のとおりです。

- COBOL 文では、先頭にコロンを付けてはいけません。
- ALTER や CREATE などのデータ定義（DDL）文で使用してはいけません。

SQL 文中に標識変数を記述する場合は、前にコロンを付けて、対応付けられたホスト変数の直後に記述してください。（読みやすいように、標識変数の前にオプションのキーワード INDICATOR を付けてもかまいません）

SQL 文で使用するプログラム変数はすべて、COBOL 言語の規則に従って宣言しなければなりません。この場合、通常の有効範囲規則が適用されます。COBOL では変数名の長さは任

意ですが、Pro*COBOL では最初の 30 文字だけが有効です。数字で始まるものも含め、有効な COBOL 識別子ならどれでもホスト変数識別子として使えます。

ホスト変数の外部データ型とそのソース・データベースまたはターゲット・データベースの列の内部データ型は、同じでなくてもかまいませんが、互換性がなければなりません。必要に応じて Oracle8*i* で自動変換される互換性のあるデータ型を、表 4-9 の「内部データ型と外部データ型の変換」に示します。

Oracle データ型

一般的に、ホスト・プログラムはデータベースにデータを入力し、データベースはホスト・プログラムにデータを出力します。Oracle は、入力データをデータベースの表に挿入し、出力データを選択して プログラム・ホスト変数に格納します。データ項目を格納するために、Oracle はそのデータ型を認識しなければなりません。データ型によって、記憶形式および値の有効範囲が指定されます。

Oracle は、次の 2 種類のデータ型を認識します。内部データ型と外部データ型です。内部データ型は Oracle がデータベース列にどのようにデータを格納するかを指定します。また、Oracle はデータベース疑似列を表すときに内部データ型を使用します。

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・プログラムから Oracle にデータが入力されると、Oracle は必要に応じて、入力ホスト変数の外部データ型とデータベース列の内部データ型の変換を行います。また、Oracle からホスト・プログラムにデータを出力するときも、Oracle は必要に応じて、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変換を行います。

> **注意 :** 動的 SQL 方法 4 またはデータ型の同値化を使うと、デフォルトのデータ型変換を上書きできます。データ型の同値化は、4-43 ページの「データ型の同値化」を参照してください。

表

Pro*COBOL では、表ホスト変数（ホスト表と呼ばれます）を定義して、それを単一の SQL 文で操作できます。SELECT 文、FETCH 文、DELETE 文、INSERT 文および UPDATE 文を使用すると、大量のデータの問合せおよび操作を簡単に行うことができます。

ホスト表の詳細は、第 7 章の「ホスト表」を参照してください。

エラーおよび警告

埋込み SQL 文を実行すると、成功するか、もしくは失敗した場合にエラーまたは警告が発生します。これらの結果を処理する方法が必要です。Pro*COBOL は、次のエラー処理方法を備えています。

- SQLCODE 状態変数
- SQLSTATE 状態変数

- SQL 通信領域（SQLCA）
- WHENEVER 文
- Oracle 通信領域（ORACA）

SQLCODE/SQLSTATE 状態変数

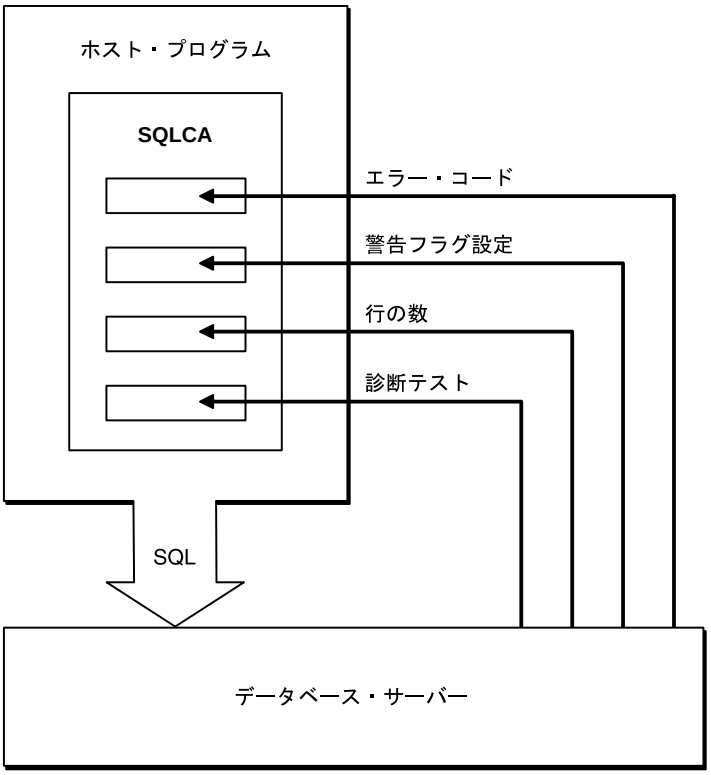
SQL 文の実行後、Oracle Server は SQLCODE または SQLSTATE という変数に状態コードを戻します。状態コードは、その SQL 文の実行に成功したか、エラーまたは警告状態が発生したかを示します。

SQLCA 状態変数

Oracle は、実行時の状態情報をプログラムに渡すためにプログラム変数を使用します。SQLCA は、このプログラム変数を定義するデータ構造です。SQLCA を使うと、直前に実行を試みた作業に関する Oracle からのフィードバックに基づいて、別の処理を行うことができます。たとえば、DELETE 文が成功したかどうかを調べ、成功した場合には、削除された行数を調べることができます。

SQLCA によって、診断チェックおよびイベント処理ができます。実行時には、Oracle8*i* からプログラムに渡される状態情報を SQLCA で保持します。SQL 文の実行後、Oracle8*i* により図 2-2 に示すように SQLCA 変数が設定されて、結果が示されます。

図 2-2 SQLCA の更新



INSERT 文、UPDATE 文または DELETE 文が成功したかどうかを調べることができ、成功した場合は、実行された行数を調べることができます。また、文が失敗した場合には、何が起こったかを詳しく調べることができます。

MODE={ANSI13 | ORACLE} のとき、SQLCA を宣言するには、SQLCA をハードコーディングするか、INCLUDE 文を使ってプログラムにコピーする必要があります。SQLCA の宣言方法と使用方法は、8-20 ページの「SQL 通信領域の使用」を参照してください。

WHENEVER 文

WHENEVER 文を使うと、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。アクションには、次の文から続行、サブプログラムのコール、ラベル付命令への分岐、段落の実行、停止などがあります。

ORACA

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 を使用できます。ORACA は、Oracle の通信を扱うデータ構造です。この中にはカーソルの統計情報、現在の 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。

プリコンパイラ・オプションとエラー処理

Oracle は、SQL 文が成功したか失敗したかを、状態変数 SQLSTATE および SQLCODE で戻します。プリコンパイラ・オプション MODE=ORACLE のときは、SQLCA を組み込むことによって SQLCODE を宣言できます。MODE=ANSI のときは、SQLSTATE または SQLCODE のどちらかを宣言する必要があります。詳細は第 8 章の「エラー処理と診断」を参照してください。

プログラミング・ガイドライン

ここでは、埋込み SQL の構文、コーディング規則、Pro*COBOL 固有の機能および制限事項について説明します。

略称

COBOL の標準略称を使用できます。たとえば、PICTURE IS を PIC としたり、USAGE IS COMPUTATIONAL を COMP と記述できます。

大 / 小文字の区別なし

Pro*COBOL プリコンパイラのオプションと値、EXEC SQL 文、インライン・コマンド、および COBOL 文には大文字と小文字の区別がありません。Pro*COBOL プリコンパイラは大文字のトークンも小文字のトークンも受け入れます。

COBOL のバージョンのサポート

Pro*COBOL では、使用しているオペレーティング・システムに標準の COBOL（通常は COBOL-85 または COBOL-74）がサポートされます。両方の COBOL をサポートするプラットフォームもあります。詳細は、使用しているシステム固有の Oracle マニュアルを参照してください。

コーディング領域

プリコンパイラ・オプション FORMAT は、ソース・コードの書式を指定します。FORMAT=ANSI（デフォルト）と指定すると、ANSI 規格にできる限り準拠したコードを作成できます。ANSI 規格では、1 ～ 6 桁目にオプションの順序番号を記述でき、7 桁目（標識領域）にはコメント行または継続行を記述できます。

分割ヘッダー、セクション・ヘッダー、段落の名前、FD、および 01 文は、8 ～ 11 桁目（A 領域）から始まります。EXEC SQL 文、EXEC ORACLE 文など、その他の文は、領域 B（12 ～ 72 桁）に記述する必要があります。ソース・コード書式のこれらのガイドラインは、コンパイラの規則によって変更できます。

FORMAT=TERMINAL を指定すると、COBOL 文を 1 桁目（一番左の桁）から始めるか、または 1 桁目を標識領域にすることができます。このガイドラインも、コンパイラの規則によって変更されます。

COBOL 文の書式が実際に受け付けられるかどうかを判断するには、使用しているプラットフォーム用の COBOL コンパイラのマニュアルを参照してください。

> **注意 :** このマニュアルのサンプル COBOL コードでは、FORMAT=TERMINAL になっています。デモ・ディレクトリのオンライン・サンプル・プログラムでは FORMAT=ANSI になっています。

カンマ

SQL では、次の例に示すように、カンマを使ってリスト項目を区切る必要があります。

```
EXEC SQL SELECT ENAME, JOB, SAL
    INTO :EMP-NAME, :JOB-TITLE, :SALARY
    FROM EMP
    WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

COBOL では、カンマまたは空白を使ってリスト項目を区切ることができます。たとえば、次の 2 つの文は等価です。

```
ADD AMT1, AMT2, AMT3 TO TOTAL-AMT.
ADD AMT1 AMT2 AMT3 TO TOTAL-AMT.
```

コメント

SQL 文の中に COBOL のコメント行を入れることができます。COBOL のコメント行は、標識領域内で先頭にアスタリスク（*）を付けて始めます。

SQL 文の中には、"--" で始まる ANSI SQL スタイルのコメントを行の最後に記述できます。（ただし、SQL 文の最終行の後には記述できません。）

行の残りの部分には、"*>" の 2 文字で始まる COBOL コメントを記述できます。

SQL 文には C スタイルのコメント（/* ... */）を記述できます。

次の例には、上記の 4 つのスタイルのコメントがすべて含まれています。

```
     MOVE 12 TO DEPT-NUMBER. *> This is the software development group.
     EXEC SQL SELECT ENAME, SAL
*    assign column values to output host variables
         INTO :EMP-NAME, :SALARY    -- output host variables
     /*  column values assigned to output host variables */
         FROM EMP
         WHERE DEPTNO = :DEPT-NUMBER
     END-EXEC.    -- illegal Comment
```

コメントをネストしたり、SQL 文の最終行の終了記号 END-EXEC の後にコメントを入れることはできません。

行の継続

次の例に示すように、COBOL の規則に従って SQL 文を次の行へ続けることができます。

```
     EXEC SQL SELECT ENAME, SAL INTO :EMP-NAME, :SALARY FROM EMP
          WHERE DEPTNO = :DEPT-NUMBER
     END-EXEC.
```

継続標識は必要ありません。

ある行から次の行に文字列リテラルを継続するには、72 桁までリテラルを記述します。次の行の 7 桁目にハイフン（-)、12 桁以降に引用符を記述して、残りのリテラルを記述します。次に例を示します。

```
 WORKING STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         ...
 01  UPDATE-STATEMENT  PIC X(80) VALUE "UPDATE EMP SET BON
-               "US = 500 WHERE DEPTNO = 20".
     EXEC SQL END DECLARE SECTION END-EXEC.
```

Copy 文

Copy 文は、Pro*COBOL ではサポートされません。かわりに、INCLUDE プリコンパイラ文を使います。詳細は、2-21 ページの「INCLUDE 文の使用」を参照してください。
INCLUDE および DECLARE_SECTION=YES を使う場合は注意が必要です。グループ項目は、宣言文の中または外のどちらかにすべてを記述する必要があります。

カンマ小数点

Pro*COBOL では、ENVIRONMENT DIVISION での DECIMAL-POINT IS COMMA 句がサポートされます。ソース・ファイルに DECIMAL-POINT IS COMMA 句が指定されていると、VALUE 句内で数値リテラルの小数部分の開始記号としてカンマを使うことができます。

たとえば、次の句は有効です。

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID.  FOO
        ENVIRONMENT DIVISION.
        CONFIGURATION   SECTION.
        SPECIAL-NAMES.
            DECIMAL-POINT IS COMMA.               *>  <-- **
        DATA DIVISION.
        WORKING-STORAGE SECTION.
    ...
        01  WDATA1          PIC      S9V999 VALUE  +,567. *>  <--- **
        01  WDATA2          PIC      S9V999 VALUE  -,234. *>  <--- **
    ...
```

デリミタ

COBOL の文字列定数およびリテラルのデリミタの指定には、LITDELIM オプションを使用します。LITDELIM=APOST と指定すると、Pro*COBOL では COBOL コードを生成する際にアポストロフィを使います。LITDELIM=QUOTE（デフォルト）と指定すると、次に示すように、引用符を使います。

```
CALL "SQLROL" USING SQL-TMP0.
```

SQL 文では、次の例に示すように、特殊文字または小文字を含んでいる識別子は引用符で区切らなければなりません。

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

また、文字列定数を区切る場合は、次の例のようにアポストロフィを使います。

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Pro*COBOL は、Pro*COBOL ソース・ファイルで使用されているデリミタに関係なく、LITDELIM の値で指定されたデリミタを生成します。

オプションの DIVISION

次の分割ヘッダーはオプションです。

- IDENTIFICATION DIVISION
- ENVIRONMENT DIVISION

- DATA DIVISION

PROCEDURE DIVISION ヘッダーはオプションではありません。たとえば、次のソースは、プリコンパイルできる状態になっています。

```
*IDENTIFICATION DIVISION header is optional
 PROGRAM-ID.     HELLO.
*ENVIRONMENT DIVISION header is optional
 CONFIGURATION SECTION.
*DATA DIVISION header is optional
 WORKING-STORAGE SECTION.
 PROCEDURE       DIVISION.
     DISPLAY "Hello World!".
     STOP RUN.
```

埋込み SQL の構文

Pro*COBOL プログラムで SQL 文を使用する場合は、SQL 文の前に EXEC SQL を指定し、SQL 文の最後に END-EXEC キーワードを指定します。埋込み SQL の構文については、『Oracle8*i* SQL リファレンス』を参照してください。

表意定数

HIGH-VALUE、ZERO、SPACE などの表意定数は、SQL 文では使用できません。たとえば、次の SQL 文は無効です。

```
EXEC SQL DELETE FROM EMP WHERE COMM = ZERO END-EXEC.
```

このかわりに、次のように指定します。

```
EXEC SQL DELETE FROM EMP WHERE COMM = 0 END-EXEC.
```

ファイルの長さ

Pro*COBOL が処理できるソース・ファイルの長さには制限があります。内部で使用する変数によって、生成されるファイルのサイズが制限される場合があります。許容される行数についての絶対的な制限はありませんが、次に示すようなソース・ファイルの状態によって、ファイル・サイズに制約が生じます。

- 埋込み SQL 文の複雑さ（たとえば、バインド変数と定義変数の数）
- データベース名の使用の有無（たとえば、AT 句を使ってデータベース名に接続する場合）
- 埋込み SQL 文の数

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

FILLER の使用

ホスト変数の宣言でワード FILLER を使用できます。ワード FILLER は、明示的に参照できないグループの基本項目を指定するときに使用します。次の宣言は有効です。

```
01  STOCK.
    05  DIVIDEND     PIC X(5).
    05  FILLER       PIC X.
    05  PRICE        PIC X(6).
```

ホスト変数名

ホスト変数には、有効な標準 COBOL 識別子であればどれでも使えます。変数名の長さに制限はありませんが、有効なのは最初の 30 文字だけです。COBOL コンパイラが認識する有効文字の最大数は 30 です。

SQL92 規格に準拠するには、ホスト変数名の長さを 18 文字以下に制限します。

アプリケーションでの使用制限があるワードの一覧は、付録 C の「予約語、キーワードおよび名前領域」を参照してください。

ハイフン付きの名前

静的 SQL 文ではハイフン付きのホスト変数名を使用できますが、動的 SQL では使用できません。たとえば次の指定は無効です。

```
MOVE "DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER" TO SQLSTMT.
EXEC SQL PREPARE STMT1 FROM SQLSTMT END-EXEC.
```

レベル番号

ホスト変数を宣言する際、レベル番号 01 ～ 49 および 77 を使用できます。Pro*COBOL では、VARYING 句を含む変数またはレベル 49 または 77 として宣言する疑似型の変数（先頭に "SQL-" が付くデータ型の変数）は使用できません。

MAXLITERAL のデフォルト

MAXLITERAL オプションを使って、Pro*COBOL で生成される文字列リテラルの最大長を指定して、コンパイラ制限の超過を防げます。Pro*COBOL の場合はデフォルト値は 256 ですが、これより小さい値の指定が必要な場合もあります。

マルチバイト・データ型

マルチバイト文字データを扱うために、ANSI 規格の各国キャラクタ・セットのデータ型がサポートされています。使用しているコンパイラで PIC N 句または PIC G 句がサポートされている場合、これらの句によって、固定長の NCHAR 文字列を格納する変数が定義されま

す。可変長のマルチバイト各国キャラクタ・セット文字列を格納するには、長さフィールドと文字列フィールドからなる COBOL グループ項目を使用します。詳細は 4-27 ページの「VARCHAR 変数」を参照してください。

クライアント側の各国キャラクタ・セットを指定するために環境変数 NLS_NCHAR を使うことができます。

NLS_LOCAL=YES

プリコンパイラ・オプション NLS_LOCAL=YES と設定した場合、動的 SQL 文はプリコンパイル時には処理されず、データベース・サーバー自体はマルチバイト NLS 文字列を処理しないため、マルチバイト NLS 文字列を動的 SQL 文に埋め込むことはできません。

また、NLS_LOCAL=YES のときは、マルチバイト NLS データを格納する列は埋込みデータ定義言語（DDL）文で使用できません。この制限はプリコンパイル時には適用されないため、このような列型を埋込み DDL 文で使用すると、プリコンパイル・エラーではなく実行エラーが発生します。

詳細は 14-31 ページの「NLS_LOCAL」を参照してください。

COBOL 文の NULL 文字

埋込み SQL 文または COBOL コードに NULL 文字を使用しないでください。NULL 文字はサポートされていません。

SQL の NULL

SQL では、NULL は欠落した列値、不明な列値または適用できない列値を表し、0（ゼロ）とも空白とも等価ではありません。NULL を NULL でない値に変換するには NVL 関数を使用し、NULL を検索するには IS [NOT] NULL 比較演算子を使用します。また、NULL の挿入およびテストを行うときは標識変数を使用します。

段落の名前

次の例に示すように、COBOL の標準的な段落の名前を SQL 文と対応付けできます。

```
LOAD-DATA.
    EXEC SQL
        INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
            VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
    END-EXEC.
```

また、次のように、WHENEVER... DO 文または WHENEVER... GOTO 文で段落の名前の参照もできます。

```
PROCEDURE DIVISION.
MAIN.
```

```
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
 SQL-ERROR.
     ...
```

段落の名前は、すべて領域 A から始めます。

REDEFINES 句

COBOL の REDEFINES 句を使って、グループ項目または基本項目を再定義できます。たとえば、次の宣言は有効です。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  REC-ID   PIC X(4).
 01  REC-NUM  REDEFINES  REC-ID  PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

および

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01  STOCK.
       05  DIVIDEND     PIC X(5).
       05  PRICE        PIC X(6).
   01  BOND  REDEFINES  STOCK.
       05  COUPON-RATE  PIC X(4).
       05  PRICE        PIC X(7).
     EXEC SQL END DECLARE SECTION END-EXEC.
```

1 つの INTO 句で、グループ項目ホスト変数とその再定義の両方の項目が使用されていても、Pro*COBOL は警告もエラーも発行しません。

関係演算子

表 2-2 のように、COBOL の関係演算子はそれに対応する SQL の関係演算子と異なります。また、COBOL では記号のかわりにワードを使用できますが、SQL では使用できません。

表 2-2 関係演算子

SQL 演算子	COBOL 演算子
=	=、EQUAL TO
< >、!=、^=	NOT=、NOT EQUAL TO
>	>、GREATER THAN
<	<、LESS THAN
>=	>=、GREATER THAN OR EQUAL TO
<=	<=、LESS THAN OR EQUAL TO

文終了記号

COBOL の文は COBOL 文または SQL 文、あるいはその両方を 1 つ以上含み、ピリオドで終わります。条件文では、次の例に示すように、ピリオドで終わる必要があるのは最後の文だけです。

```
IF EMP-NUMBER = ZERO
    MOVE FALSE TO VALID-DATA
    PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE
ELSE
    EXEC SQL DELETE FROM EMP
        WHERE EMPNO = :EMP-NUMBER
    END-EXEC
    ADD 1 TO DELETE-TOTAL.
END-IF.
```

SQL 文は、カンマ、ピリオドまたはその他の COBOL 文で終了することができます。

ただし、COBOL-74 で WHENEVER... GOTO または WHENEVER... STOP を使って SQL 文のエラーを処理する場合、SQL 文はピリオドで終わるか、直後に ELSE がなければなりません。

次の DELETE 文は、この必要条件を満たすために位置が変更されています。

```
EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    IF EMP-NUMBER = ZERO
        MOVE FALSE TO VALID-DATA
        PERFORM GET-EMP-NUM UNTIL VALID-DATA = TRUE
    ELSE
        ADD 1 TO DELETE-TOTAL
        EXEC SQL DELETE FROM EMP
            WHERE EMPNO = :EMP-NUMBER
```

```
            END-EXEC.
```

また、SQL 文を別の段落に入れ、その段落を PERFORM することもできます。

宣言文

データベース・サーバーとアプリケーション・プログラムの間のデータの受渡しには、ホスト変数とエラー処理が必要です。この項では、これらの必要条件を満たす方法を説明します。

宣言文とは

オプションの宣言文は、次の文で開始します。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

次の文で終了します。

```
     EXEC SQL END DECLARE SECTION END-EXEC.
```

この 2 つの文の間に指定できるのは、次の要素だけです。

- ホスト変数および標識変数の宣言
- ホスト変数以外の COBOL の変数
- EXEC SQL DECLARE 文
- EXEC SQL INCLUDE 文
- EXEC SQL VAR 文
- EXEC SQL TYPE 文
- EXEC ORACLE 文
- COBOL コメント

例

プログラムで使用する 4 つのホスト変数を宣言した例を次に示します。

```
 WORKING-STORAGE SECTION.
   ...
* The next line is optional
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         ...
 01 EMP-NUMBER     PIC 9(4)  COMP VALUE ZERO.
 01 EMP-NAME       PIC X(10) VARYING.
 01 SALARY         PIC S9(5)V99 COMP-3 VALUE ZERO.
```

```
 01 COMMISSION     PIC S9(5)V99 COMP-3 VALUE ZERO.
* The next line is optional
     EXEC SQL END DECLARE SECTION END-EXEC.
```

プリコンパイラ・オプション DECLARE_SECTION

宣言文はオプションです。Pro*COBOL には、8.0 より前のリリースとの下位互換性を保つために、宣言文内の宣言だけをホスト変数として使用するかどうかを明示的に制御できるコマンド行プリコンパイラ・オプションが用意されました。このオプションは次のとおりです。

DECLARE_SECTION={YES | NO }（デフォルトは NO）

DECLARE_SECTION オプションは、コマンド行または構成ファイル内で指定しなければなりません。

MODE=ORACLE、DECLARE_SECTION=YES と設定した場合は、宣言文の中で宣言した変数だけをホスト変数として使用できます。MODE=ANSI と設定すると、DECLARE_SETION は暗黙的に YES に設定されます。マクロ・オプションとマイクロ・オプションの詳細は、14-5 ページの「マクロ・オプションとマイクロ・オプション」を参照してください。

プリコンパイラ・オプション DECLARE_SECTION を NO（デフォルト）に設定すると宣言文は省略できます。このようなオプションとしての使用は、Pro*COBOL リリース 8.0 で加えられた変更です。DECLARE_SECTION を YES に設定した場合は、SQL 文中で使用するすべてのプログラム変数を宣言文で宣言する必要があります。

DECLARE_SECTION を NO に設定した場合は、宣言文を使うかどうかを選択できます。この場合、ホスト変数および標識変数は宣言文の中でも宣言文の外でも宣言できます。オプションの詳細は、14-17 ページの「DECLARE_SECTION」を参照してください。

1 つのプリコンパイル単位で複数の宣言文を使用できます。さらに、複数の独立したプリコンパイル単位を 1 つのホスト・プログラムに含めることもできます。

注意 : DECLARE_SECTION の値に関係なく、宣言は、すべて完全に解析されます。

INCLUDE 文の使用

INCLUDE 文を使用して、ホスト・プログラムにファイルをコピーできます。次に例を示します。

```
*    Copy in the SQL Communications Area (SQLCA)
     EXEC SQL INCLUDE SQLCA END-EXEC.
*    Copy in the Oracle Communications Area (ORACA)
     EXEC SQL INCLUDE ORACA END-EXEC.
```

INCLUDE は、どのファイルに対しても実行できます。Pro*COBOL プログラムをプリコンパイルすると、EXEC SQL INCLUDE 文はそれぞれ、その文で指定されたファイルのコピーに置き換えられます。

ファイルの拡張子

システムでファイル拡張子が使用される場合にファイル拡張子の指定を省略すると、Pro*COBOL ではソース・ファイルのデフォルトの拡張子（通常は COB）が使用されます。デフォルトの拡張子はシステムによって異なります。詳細は、使用しているシステム固有の Oracle マニュアルを参照してください。

検索パス

システムでディレクトリが使用される場合は、次のように INCLUDE オプションを使用して、INCLUDE するファイルの検索パスを設定できます。

```
INCLUDE=path
```

このとき、*path* のデフォルト値は現行のディレクトリです。

Pro*COBOL は、最初に現行のディレクトリを検索し、次に INCLUDE オプションで指定されたディレクトリを検索して、最後に標準 INCLUDE ファイル用のディレクトリを検索します。このため、SQLCA や ORACA などの標準ファイルのパスを指定する必要はありません。標準以外のファイルについては、現行のディレクトリに格納されている場合を除いてパスが必要です。

また次のように、コマンド行で複数のパスを指定することもできます。

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

複数のパスを指定すると、現行のディレクトリ、*path1* ディレクトリ、*path2* ディレクトリの順に検索されます。標準 INCLUDE ファイルが入っているディレクトリは最後に検索されます。パスの構文はシステムによって異なります。詳細は、使用しているシステム固有の Oracle マニュアルを参照してください。

注意 : Pro*COBOL では、検索パスの指定があっても、現行のディレクトリでファイルが先に検索されることに注意してください。INCLUDE するファイルが別のディレクトリにある場合は、現行のディレクトリに同じ名前のファイルがないことを確認してください。また、検索パスでそのディレクトリより前にあるディレクトリにも同じ名前のファイルがないことを確認してください。大 / 小文字が区別されるオペレーティング・システムを使用している場合は、ファイルを格納したときと同じように大 / 小文字を区別してファイル名を指定してください。

ネストされたプログラム

COBOL におけるプログラムのネストとは、あるプログラムを別のプログラムの中に入れることを意味します。中に含まれるプログラムは、それを含んでいるプログラムのリソースの一部を参照できます。また、ネストされたプログラムと上位のプログラムの中で同じ名前を使用できます。名前はそれぞれのプログラムの中だけで認識されるため、異なるデータ項目の記述に同じ名前を使用しても競合することはありません。ただし、上位プログラムの構成部に記述されている名前は、ネストされたプログラムから参照できます。

コンパイラによっては、GLOBAL 句を使うネストされたプログラムがサポートされないことがあります。Pro*COBOL ではネストされたプログラムがサポートされるので、生成されたコードには GLOBAL 句が含まれます。GLOBAL 句が無条件に生成されるのを回避にするには、プリコンパイラ・オプション NESTED を使用します。NESTED（=YES または NO）は、デフォルトで YES に設定されています。構成ファイルまたはコマンド行で使用できますが、インライン（EXEC ORACLE 文）では使用できません。GLOBAL 句の生成を避けるには、NESTED=NO を指定します。詳細は 14-31 ページの「NESTED」を参照してください。

プログラムでは、ネストされたプログラムを複数記述できます。同様に、ネストされたプログラムについても、その中にさらにネストされたプログラムを記述できます。ネストされたプログラムは、それを含んでいるプログラムの END PROGRAM ヘッダーの直前に記述する必要があります。

ネストされたプログラムをコールできるのは、そのプログラムを直接的または間接的に含んでいるプログラムだけです。ネストされたプログラムを別のプログラム（ネストのツリー構造の別の分岐にあるプログラムも含む）からコールする場合は、ネストされたプログラムの PROGRAM-ID 段落に COMMON 句を記述します。COMMON は、ネストされたプログラムにだけ使用できます。

```
PROGRAM-ID.  <nested-program-name> COMMON.
```

ファイル定義に GLOBAL 句を使用して、レベル 01 のデータ項目（自動的にグローバルになる従属項目）を記述できます。このように記述したデータ項目は、それらのデータ項目に直接的または間接的に含まれるすべてのサブプログラムで参照できます。上位プログラムには GLOBAL を使用します。上位プログラムで GLOBAL と宣言された名前と同じ名前がネストされたプログラムで定義されていると、COBOL ではネストされたプログラムの中の宣言が使用されます。データ項目に REDEFINES 句が使用されているときは、GLOBAL は REDEFINES の後に記述しなければなりません。

```
FD file-name GLOBAL ...
01 data-name1 GLOBAL ...
01 data-name2 REDEFINES data-name3 GLOBAL ...
```

ネストされたプログラムのサポート

Pro*COBOL では、1 つのソース・ファイルの中に埋込み SQL をもつネストされたプログラムを使用できます。上位プログラムでグローバルとしてマークされた、上位プログラム・レベルで有効なホスト変数である 01 レベルの項目はすべて、上位プログラムが直接的または間接的に含んでいるすべてのプログラムで、有効なホスト変数として使用できます。次の例を考えてみます。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 REC1  GLOBAL.
          05    VAR1   PIC X(10).
          05    VAR2   PIC X(10).
01 VAR1  PIC X(10) GLOBAL.
    EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
    ...
    <main program statements>
    ...
IDENTIFICATION DIVISION.
   PROGRAM-ID. NESTEDPROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 VAR1   PIC S9(4).

PROCEDURE DIVISION.
    ...
    EXEC SQL SELECT X, Y INTO :REC1 FROM ... END-EXEC.

    EXEC SQL SELECT X INTO :VAR1 FROM ... END-EXEC.

    EXEC SQL SELECT X INTO :REC1.VAR1 FROM ... END-EXEC.
    ...
END PROGRAM NESTEDPROG.
END PROGRAM MAINPROG.
```

メイン・プログラムでホスト変数 REC1 がグローバルとして宣言されているため、ネストされたプログラムは最初の SELECT 文で REC1 を使用できます。（REC1 を宣言する必要はありません。）VAR1 は、メイン・プログラムではグローバル変数として宣言され、ネストされたプログラムではローカル変数として宣言されています。このため、2 番目の SELECT 文では S9(4) として宣言された VAR1 が使用され、グローバル宣言は無効になります。3 番目の SELECT 文では、PIC X(10) として宣言された、REC1 のグローバルな VAR1 が使用されます。

上記では、DECLARE_SECTION=NO が使用されている場合の結果を説明しています。DECLARE_SECTION=YES の場合、Pro*COBOL は、宣言文の中で宣言されていないかぎりホスト変数を認識しません。DECLARE_SECTION=YES と指定して上記のプログラムをプリコンパイルすると、2 番目の SELECT 文の結果、あいまいなホスト変数エラーが発生します。最初の SELECT 文と 3 番目の SELECT 文は、DECLARE_SECTION=NO の場合と同じです。

注意 : 再帰的なネストされたプログラムは、サポートされていません。

SQLCA の宣言

ネストされたプログラムでは、提供される組込み SQLCA（2-9 ページの「SQLCA 状態変数」を参照）の定義はグローバルとして宣言されるため、SQLCA の宣言は上位プログラム内でだけ必要になります。SQLCA は、新しい SQL 文が実行されるたびに変更できます。提供される SQLCA はいつでも変更できるため、ネストされたプログラムで追加の SQLCA 領域を宣言したい場合にはグローバル指定を削除できます。これは、SQLDA および ORACA でも同じです。

ネストされたプログラムの例

デモ・ディレクトリの SAMPLE13.PCO を参照してください。

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、さまざま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。コード・セクションには、ホスト言語の文を記述することも、EXEC SQL 文を記述することもできます。次の文でプリコンパイルの条件を制御します。

```
*   -- define a symbol
     EXEC ORACLE DEFINE symbol END-EXEC.
*    -- if symbol is defined
     EXEC ORACLE IFDEF symbol  END-EXEC.
*   -- if symbol is not defined
     EXEC ORACLE IFNDEF symbol END-EXEC.
*           -- otherwise
     EXEC ORACLE ELSE END-EXEC.
*          -- end this control block
     EXEC ORACLE ENDIF END-EXEC.
```

条件文は END-EXEC で終了しなければなりません。

注意 : 使用しているコンパイラの条件付きコンパイル機能が、Pro*COBOL でサポートされない場合があります。

例

次の例では、記号 *SITE2* が定義されているときのみ SELECT 文がプリコンパイルされます。

```
EXEC ORACLE IFDEF SITE2 END-EXEC.
EXEC SQL SELECT DNAME
```

```
   INTO :DEPT-NAME
   FROM DEPT
   WHERE DEPTNO = :DEPT-NUMBER
EXEC ORACLE ENDIF END-EXEC.
```

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 END-EXEC.
EXEC ORACLE IFDEF INNER END-EXEC.
...
EXEC ORACLE ENDIF END-EXEC.
EXEC ORACLE ENDIF END-EXEC.
```

ホスト言語または埋込み SQL のコードを IFDEF と ENDIF の間に記述し、記号を定義しないことによって、そのコードをコメントとして扱うことができます。

記号の定義

記号を定義するには 2 通りの方法があります。次の文をホスト・プログラムに組み込む

```
EXEC ORACLE DEFINE symbol END-EXEC.
```

または、次の構文を使ってコマンド行で記号を定義します。

```
... INAME=filename ... DEFINE=symbol
```

このとき *symbol* の部分は大 / 小文字区別がありません。

Pro*COBOL をシステムにインストールした時点で、ポート固有のいくつかの記号が事前定義されています。たとえば、オペレーティング・システムの事前定義済み記号には、CMS および MVS、MS-DOS、UNIX、VMS があります。

分割プリコンパイル

複数の COBOL プログラム・モジュールを別々にプリコンパイルし、後でリンクして 1 つの実行可能プログラムを作成できます。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。

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。

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。このためカーソル操作が複数のプリコンパイル・ユニット（ファイル）に及ぶことはありません。つまり、あるファイルでカーソルを宣言し、そのカーソルを別のファイルでオープンしたりフェッチすることはできません。したがって、プリコンパイルを個別に実行するときは、指定のカーソルに対する定義および参照がすべて 1 つのファイル内に記述されていることを確認してください。

MAXOPENCURSORS の指定

Oracle に接続するプログラム・モジュールをプリコンパイルするときは、MAXOPENCURSORS に、どのプログラム・モジュールについても十分な大きさの値を指定してください。指定した MAXOPENCURSORS の値は、別のプログラム・モジュールに使用すると無視されます。実行時には、その接続に有効な値だけが使用されます。

単一の SQLCA の使用

SQLCA を 1 つだけ使用する場合は、プログラム・モジュールのどれかで SQLCA をグローバル宣言してください。

単一の DATE_FORMAT の使用

各プログラム・モジュールでは、DATE に同じ書式の文字列を使用しなければなりません。

制限

明示的なカーソルの参照は、すべて同じプログラム・ファイル内で行います。別のモジュールで DECLARE されたカーソルの操作はできません。カーソルの詳細は、第 4 章を参照してください。

また、SQL 文が記述されているプログラム・ファイルはすべて、ローカル SQL 文の有効範囲内にある SQLCA を必要とします。

コンパイルとリンク

実行可能プログラムを作成するには、Pro*COBOL によって生成されたソース・ファイルをコンパイルし、その結果得られたオブジェクト・モジュールを、SQLLIB およびシステム固有の Oracle ライブラリ内の必要なモジュールとリンクさせる必要があります。また、OCI コールを埋め込んでいる場合は、必ず OCI ランタイム・ライブラリ（OCILIB）にリンクしてください。

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。プリコンパイル済みのプログラムにサードパーティのソフトウェアをリンクしようとしたときに、このような現象が起こる可能性があります。こうした問題が生じるのは、サードパーティのソフトウェアの中には Oracle と互換性のないものがあるためです。Oracle カスタマ・サポートに問い合せて、使用するソフトウェアがサポートされているかどうかを確認してください。

コンパイルとリンクの方法はシステムによって異なります。たとえば、システムによっては、ホスト言語プログラムをコンパイルするときにコンパイラの最適化をオフにしなければならないものもあります。手順については、使用しているシステム固有の Oracle マニュアルを参照してください。

サンプル表

このマニュアルで紹介するほとんどのプログラム例では、DEPT および EMP という 2 つのサンプル・データベース表が使用されています。次に示すのはそれらの定義です。

```
CREATE TABLE DEPT
    (DEPTNO    NUMBER(2),
     DNAME     VARCHAR2(14),
     LOC       VARCHAR2(13))

CREATE TABLE EMP
    (EMPNO     NUMBER(4) primary key,
     ENAME     VARCHAR2(10),
     JOB       VARCHAR2(9),
     MGR       NUMBER(4),
     HIREDATE  DATE,
     SAL       NUMBER(7,2),
     COMM      NUMBER(7,2),
     DEPTNO    NUMBER(2))
```

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設定されています。

```
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
```

サンプル・プログラム : SAMPLE1.PCO

埋込み SQL がどういうものかは、プログラム例を見るとよくわかります。次のプログラムは、demo ディレクトリの SAMPLE1.PCO です。

プログラムでデータベースにログインし、従業員番号を入力すると、従業員名、給与およびコミッションを EMP 表に問合わせます。選択の結果はホスト変数 EMP-NAME および SALARY、COMMISSION に格納されます。また、ホスト標識変数 COMM-IND を使って列 COMMISSION 内の NULL 値を検出します。詳細は 4-23 ページの「標識変数」を参照してください。

次に、段落 DISPLAY-INFO によって結果が表示されます。

COBOL 変数 USERNAME および PASSWD、EMP-NUMBER は、VARYING 句によって宣言されます。VARYING 句では、VARCHAR という可変長文字列の Oracle 外部データ型を使用できます。このデータ型については、4-27 ページの「VARCHAR 変数」を参照してください。

エラー処理のために SQLCA 通信領域が組み込まれています。エラーが発生すると、段落 SQL-ERROR が実行されます。詳細は 8-20 ページの「SQL 通信領域の使用」を参照してください。

ここで使用されている BEGIN DECLARE SECTION 文および END DECLARE SECTION 文は、プリコンパイラ・オプション DECLARE_SECTION が YES に設定されているか、またはオプション MODE が ANSI に設定されていなければ、省略してもかまいません。詳細は 2-2 ページの「MODE」を参照してください。

エラー処理のために WHENEVER 文を使用しています。詳細は 8-28 ページの「WHENEVER ディレクティブ」を参照してください。

このプログラムは、従業員番号としてゼロを入力すると終了します。

```
*****************************************************************
* Sample Program 1:  Simple Query                               *
*                                                               *
* This program logs on to ORACLE, prompts the user for an       *
```

```
* employee number, queries the database for the employee's      *
* name, salary, and commission, then displays the result.       *
* The program terminates when the user enters a 0.              *
*****************************************************************

 ID DIVISION.

 PROGRAM-ID. QUERY.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME          PIC X(10) VARYING.
 01  PASSWD            PIC X(10) VARYING.
 01  EMP-REC-VARS.
     05  EMP-NAME      PIC X(10) VARYING.
     05  EMP-NUMBER    PIC S9(4) COMP VALUE ZERO.
     05  SALARY        PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMMISSION    PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMM-IND      PIC S9(4) COMP VALUE ZERO.
     EXEC SQL END DECLARE SECTION END-EXEC.

     EXEC SQL INCLUDE SQLCA END-EXEC.

 01  DISPLAY-VARIABLES.
     05  D-EMP-NAME    PIC X(10).
     05  D-SALARY      PIC Z(4)9.99.
     05  D-COMMISSION  PIC Z(4)9.99.
     05  D-EMP-NUMBER  PIC 9(4).

 01 D-TOTAL-QUERIED   PIC 9(4) VALUE ZERO.

 PROCEDURE DIVISION.
 BEGIN-PGM.
     EXEC SQL WHENEVER SQLERROR
        DO PERFORM SQL-ERROR END-EXEC.

     PERFORM LOGON.

 QUERY-LOOP.
     DISPLAY " ".
     DISPLAY "ENTER EMP NUMBER (0 TO QUIT): "
         WITH NO ADVANCING.

     ACCEPT D-EMP-NUMBER.
```

```
    MOVE D-EMP-NUMBER TO EMP-NUMBER.
    IF (EMP-NUMBER = 0)
        PERFORM SIGN-OFF.
    MOVE SPACES TO EMP-NAME-ARR.
    EXEC SQL WHENEVER NOT FOUND GOTO NO-EMP END-EXEC.
    EXEC SQL SELECT ENAME, SAL, COMM
        INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
        FROM EMP
        WHERE EMPNO = :EMP-NUMBER
    END-EXEC.
    PERFORM DISPLAY-INFO.
    ADD 1 TO D-TOTAL-QUERIED.
    GO TO QUERY-LOOP.

NO-EMP.
    DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
    GO TO QUERY-LOOP.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

DISPLAY-INFO.
    DISPLAY " ".
    DISPLAY "EMPLOYEE    SALARY    COMMISSION".
    DISPLAY "--------    ------    ----------".
    MOVE EMP-NAME-ARR TO D-EMP-NAME.
    MOVE SALARY TO D-SALARY.
    IF COMM-IND = -1
        DISPLAY D-EMP-NAME, D-SALARY, "          NULL"
    ELSE
        MOVE COMMISSION TO D-COMMISSION
        DISPLAY D-EMP-NAME, D-SALARY, "      ", D-COMMISSION
    END-IF.

SIGN-OFF.
    DISPLAY " ".
    DISPLAY "TOTAL NUMBER QUERIED WAS ",
        D-TOTAL-QUERIED, ".".
    DISPLAY " ".
```

```
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

3

データベースの概念

この章では、CONNECT 文とそのオプション、Net8 およびネットワーク接続に使う関連のある文について説明します。次に、トランザクション処理の方法を説明します。Oracle データに対する変更の確定または取消しを制御する方法を含めて、データベースの整合性を保つための基本的な技術を学習します。

- Oracle への接続
- デフォルトのデータベースおよび接続
- アドバンス・コネクション・オプション
- OCI（Oracle Call Interface）コールの埋込み
- 知っておくべき用語
- 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 文の使用方法
- ROLLBACK 文の使用方法
- SAVEPOINT 文の使用方法
- RELEASE オプションの使用方法
- SET TRANSACTION 文の使用方法
- デフォルトのロックの変更
- コミットを超えたフェッチ
- 分散トランザクションの処理
- トランザクション処理のガイドライン
- X/Open アプリケーションの開発

Oracle への接続

Pro*COBOL プログラムは、データの問合せや操作を行う前に、Oracle にログインしなければなりません。ログインするには、次に示すように CONNECT 文を使います。

```
EXEC SQL
    CONNECT :USERNAME IDENTIFIED BY :PASSWD
END-EXEC.
```

USERNAME および PASSWD は PIC X(*n*) または PIC X(*n*) VARYING ホスト変数です。また、次の文を使うこともできます。

```
EXEC SQL
    CONNECT :USR-PWD
END-EXEC.
```

この場合、ホスト変数 USR-PWD には、スラッシュ（/）で区切られたユーザー名とパスワードが入っています。

CONNECT 文の構文には、オプションの ALTER AUTHORIZATION 句を使うことができます。次が CONNECT 文の構文です。

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

ALTER AUTHORIZATION 句の詳細は、3-10 ページの「実行時のパスワード変更」を参照してください。SYSDBA および SYSOPER オプションの詳細は、3-11 ページの「SYSDBA 権限または SYSOPER 権限」を参照してください。

CONNECT 文は、プログラムが実行する最初の SQL 文でなければなりません。したがって、別の実行 SQL 文を、位置的には CONNECT 文の前に記述できますが、論理的には CONNECT 文の前に記述できません。プリコンパイラ・オプション AUTO_CONNECT=YES の場合は CONNECT 文は必要ありません。

Oracle ユーザー名とパスワードを別々に指定する場合は、2 つのホスト変数を文字列または VARCHAR 変数として定義します。ユーザー名とパスワードの両方を含んだユーザー ID を指定する場合は、必要なホスト変数は 1 つだけです。

ユーザー名とパスワードの変数は、CONNECT が実行される前に設定してください。この 2 つの変数が設定されていないと、CONNECT は失敗します。これらの変数は、プログラムで入力を要求することも、次のようにハードコードすることもできます。

```
WORKING STORAGE SECTION.
    ...
01  USERNAME  PIC X(10) VARYING.
01  PASSWD    PIC X(10) VARYING.
        ...
```

```
    ...
PROCEDURE DIVISION.
LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL WHENEVER SQLERROR GOTO LOGON-ERROR END-EXEC.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
```

ただし、ユーザー名およびパスワードは CONNECT 文にハードコードできません。また、引用符付きのリテラルも使用できません。たとえば、次の 2 つの文は無効です。

```
EXEC SQL
    CONNECT SCOTT IDENTIFIED BY TIGER
END-EXEC.

EXEC SQL
    CONNECT "SCOTT" IDENTIFIED BY "TIGER"
END-EXEC.
```

デフォルトのデータベースおよび接続

それぞれのノードにはデフォルトのデータベースがあります。CONNECT 文でノードだけを指定し、データベースを指定しないと、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。データベースもノードも指定しないと、現行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。デフォルトのデータベースおよび現行のノードも CONNECT 文に指定できますが、その必要はありません。

デフォルトの接続は、AT 句を指定しない CONNECT 文を使って行います。ローカルまたはリモート・ノードでデフォルトのデータベースにも非デフォルトのデータベースにも接続できます。AT 句のない SQL 文はデフォルト接続に対して実行されます。逆に、非デフォルト接続は AT 句をもつ CONNECT 文によって行われます。AT 句付きの 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でなければなりません。しかし、2 つ以上のデータベース名で同じ接続を指定できます。つまり任意のノード上のデータベースに複数の接続をもつことができます。

ユーザー名 / パスワードの使用方法

通常は、次のようにして Oracle への接続を確立します。

```
EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD END-EXEC.
```

また、次のような指定もできます。

```
EXEC SQL CONNECT :USR-PWD END-EXEC.
```

USR-PWD には *USERNAME/PASSWORD* が入ります。

3-9 ページの「自動ログイン」に示すように、自動的にログインすることもできます。

これらは、単純化した CONNECT 文のサブセットです。詳細は、この章の後の項およびF-17 ページの「CONNECT（実行可能な埋込み SQL 拡張要素）」を参照してください。

データベースおよびノードを指定しないと、現行ノードでデフォルトのデータベースに接続されます。異なるデータベースに接続したい場合、そのデータベースを明示的に識別する必要があります。

明示的なログインでは、別のデータベースに直接接続し、その接続に、SQL 文で参照される名前を付けます。同時にいくつか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。

単一の明示的ログイン

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

```
* --  Declare necessary host variables
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10) .
 01  PASSWORD  PIC X(10) .
 01  DB-STRING PIC X(20) .
        ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSSWORD.
     MOVE "nyremote" TO DB-STRING.
      ...
* --  Assign a unique name to the database connection.
     EXEC SQL DECLARE DBNAME DATABASE END-EXEC.
* --  Connect to the non-default database
     EXEC SQL
     CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT DBNAME USING :DB-STRING
     END-EXEC.
```

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* で有効なユーザーを識別します。

- ホスト変数 *DB-STRING* に、リモート・ノードの非デフォルト・データベースにログインするための Net8 の構文を入れます。
- 未宣言の識別子 *DBNAME* により、非デフォルトの接続に名前を付けます。これは Oracle が使用する識別子であって、ホスト変数でもプログラム変数でもありません。

USING 句は、*DBNAME* に対応付けるネットワークおよびマシン、データベースを指定します。その後、（*DBNAME* を指定した）AT 句を使った SQL 文が、*DB-STRING* で指定され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 necessary host variables
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10).
 01  PASSWORD  PIC X(10).
 01  DB-NAME   PIC X(10).
 01  DB-STRING PIC X(20).
        ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSSWORD.
     MOVE "oracle1" TO DB-NAME.
     MOVE "nyremote" TO DB-STRING.
     ...
* --  Connect to the non-default database
     EXEC SQL
     CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT :DB-NAME USING :DB-STRING
     END-EXEC.
```

DB-NAME がホスト変数の場合は、DECLARE DATABASE 文は必要ありません。*DBNAME* が未宣言の識別子の場合のみ、CONNECT... AT *DBNAME* 文を実行する前に DECLARE *DBNAME* DATABASE 文を実行する必要があります。

SQL 操作 : 権限を付与されている場合は、非デフォルト接続で任意の SQL データ操作文を実行できます。たとえば次のように入力します。

```
EXEC SQL AT DBNAME SELECT ...
EXEC SQL AT DBNAME INSERT ...
EXEC SQL AT DBNAME 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は、SQL 文で参照するデータベースの表をすべて、DECLARE TABLE 文で定義する必要があります。

カーソルの制御 : OPEN、FETCH、CLOSE などのカーソルの制御文は例外で、AT 句はいっさい使いません。カーソルと明示的に識別されたデータベースとを対応付けたい場合は、次に示すとおり、DECLARE CURSOR 文で AT 句を使ってください。

```
     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 END-EXEC.
```

DB-NAME がホスト変数の場合、その宣言は、宣言したカーソルを参照するすべての SQL 文の有効範囲内になければなりません。たとえば、あるサブプログラムでカーソルをオープンし、それを別のサブプログラムでフェッチする場合は、*DB-NAME* をグローバルに宣言するか、またはサブプログラムごとに渡す必要があります。

カーソルのオープンまたはクローズ、フェッチには、AT 句は使用しません。SQL 文が実行されるのは、DECLARE CURSOR 文の AT 句で名前を付けられたデータベースか、カーソル宣言で AT 句が使われていない場合はデフォルトのデータベースです。

AT :*host-variable* 句を使用して、カーソルに対応付けられた接続を変更できます。しかし、カーソルがオープンされているときは対応を変更できません。次の例を考えてみます。

```
     EXEC SQL AT :DB-NAME DECLARE EMP-CURSOR CURSOR FOR ...
     MOVE "oracle1" TO DB-NAME.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
     EXEC SQL FETCH EMP-CURSOR INTO ...
     MOVE "oracle2" TO DB-NAME.
* -- illegal, cursor still open
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
     EXEC SQL FETCH EMP-CURSOR INTO ...
```

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ので、これは無効です。複数の接続に別々のカーソルをもつことはできません。*EMP-CURSOR* は 1 つしかないので、別の接続について再オープンするには、いったんクローズする必要があります。この例をデバッグするには、次のように、カーソルを再オープンする前にクローズするだけです。

```
* -- close cursor first
     EXEC SQL CLOSE EMP-CURSOR END-EXEC.
     MOVE "oracle2" TO DB-NAME.
     EXEC SQL OPEN EMP-CUROR END-EXEC.
     EXEC SQL FETCH EMP-CURSOR INTO ...
```

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、非デフォルト接続で文を実行したい場合、AT 句を使う必要があります。次に例を示します。

```
EXEC SQL AT :DB-NAME EXECUTE IMMEDIATE :SQL-STMT END-EXEC.
```

方法 2、3、4 では、非デフォルト接続で文を実行したい場合、DECLARE STATEMENT 文でだけ AT 句を使います。PREPARE および DESCRIBE、OPEN、FETCH、CLOSE のようなその他の動的 SQL 文はすべて 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 END-EXEC.
EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC.
EXEC SQL EXECUTE SQL-STMT END-EXEC.
```

次の例は方法 3 を示します。

```
EXEC SQL AT :DB-NAME DECLARE SQL-STMT STATEMENT END-EXEC.
EXEC SQL PREPARE SQL-STMT FROM :SQL-STRING END-EXEC.
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR SQL-STMT END-EXEC.
EXEC SQL OPEN EMP-CURSOR ...
EXEC SQL FETCH EMP-CURSOR INTO ...
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

リモート・データベースに接続する場合は、AT 句を使用する必要はありません。ただし、複数の接続を同時にオープンする場合は例外です。（この場合は、アクティブな接続を識別するために AT 句が必要になります）リモート・データベースへのデフォルトの接続を行う場合は、次の構文を使用します。

```
EXEC SQL
   CONNECT :USERNAME IDENTIFIED BY :PASSWORD USING :DB-STRING
END-EXEC.
```

複数の明示的ログイン

複数の明示的ログインの場合も、単一の明示的ログインの場合と同じように、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。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME   PIC X(10) .
 01  PASSWORD   PIC X(10) .
 01  DB-STRING1 PIC X(20) .
 01  DB-STRING2 PIC X(20) .
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSWORD.
     MOVE "New-York" TO DB-STRING1.
     MOVE "Boston" TO DB-STRING2.

* --  give each database connection a unique name
     EXEC SQL DECLARE DBNAME1 DATABASE END-EXEC.
     EXEC SQL DECLARE DBNAME2 DATABASE;
```

```
* --  connect to the two non-default databases
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT DBNAME1 USING :DB-STRING1 END-EXEC.
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT DBNAME2 USING :DB-STRING2 END-EXEC.
```

未宣言の識別子 *DBNAME1* および *DBNAME2* 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に名前を付けるために使用されます。その結果、その後の SQL 文でそのデータベースを名前で参照できます。

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

```
 01  USERNAME   PIC X(10) .
 01  PASSWORD   PIC X(10) .
 01  DB-NAME    PIC X(10) .
 01  DB-STRING  PIC X(20) .
     ...
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSWORD.
     PERFORM GETDB 2 TIMES.
     ...
* --  get next database name and Net8 string
 GETDB.
     DISPLAY "Database Name? ".
     ACCEPT DB-NAME.
     DISPLAY "Net8 String? ".
     ACCEPT DB-STRING.
* --  connect to the non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
     AT :DB-NAME USING :DB-STRING
     END-EXEC.
     ...
```

また、次の例に示すように、この方法を使って同じデータベースに複数の接続が行えます。

```
     MOVE "scott" TO USERNAME.
     MOVE "tiger" TO PASSWORD.
     MOVE "nyremote" TO DB-STRING.
     PERFORM GETDB 2 TIMES
     ...
 GETDB.
* -- get next database name
     DISPLAY 'Database Name? '.
     ACCEPT DB-NAME.
* -- connect to the non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
        AT :DB-NAME USING :DB-STRING
     END-EXEC.
```

```
...
```

同じ Net8 文字列を使用する接続であっても、各接続には異なるデータベース名を使用しなければなりません。

自動ログイン

次のようなユーザー ID を使用すると、Oracle に自動的にログインできます。

```
<prefix><username>
```

prefix には Oracle 初期化パラメータ OS_AUTHENT_PREFIX の値（デフォルト値は OPS$）を指定し、*username* には使用しているオペレーティング・システムのユーザー名またはタスク名を指定します。たとえば、接頭辞が OPS$ でユーザー名が TBARNES の場合、OPS$TBARNES が Oracle の有効なユーザー ID であれば、ユーザー OPS$TBARNES として Oracle にログインします。

自動ログイン機能を利用するために必要なことは、次のようにして Pro*COBOL にスラッシュ（/）文字を渡すことだけです。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 ORACLEID   PIC X.
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
     MOVE '/' TO ORACLEID.
     EXEC SQL CONNECT :ORACLEID END-EXEC.
```

これによって自動的に OPS$*username* というユーザーとして接続します。たとえば、オペレーティング・システムのユーザー名が RHILL で、OPS$RHILL が Oracle の有効なユーザー名である場合、スラッシュ（/）を使って接続すると、ユーザー OPS$RHILL として自動的にログインします。

Pro*COBOL に文字列を渡すこともできます。ただし、その文字列に後続ブランクを入れてはいけません。たとえば、次の CONNECT 文は失敗します。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
           01 ORACLEID   PIC X(5).
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
     MOVE '/    ' TO ORACLEID.
     EXEC SQL CONNECT :ORACLEID END-EXEC.
```

AUTO_CONNECT プリコンパイラ・オプション

Pro*COBOL では、プログラムは CONNECT 文を使わずにデフォルトのデータベースにログインできます。このために必要なのは、コマンド行にプリコンパイラ・オプション AUTO_CONNECT を指定することだけです。

OS_AUTHENT_PREFIX のデフォルト値が OPS$、ユーザー名が TBARNES であり、OPS$TBARNES が Oracle の有効なユーザー ID であるとします。AUTO_CONNECT=YES の場合、Pro*COBOL が実行 SQL 文を検出すると、ユーザー・プログラムは OPS$TBARNES というユーザー ID で自動的に Oracle にログインします。

AUTO_CONNECT=NO（デフォルト）の場合は、Oracle にログインするには CONNECT 文を使用する必要があります。

実行時のパスワード変更

Pro*COBOL では、オプションの ALTER AUTHORIZATION 句によって、実行時のユーザー・パスワードをクライアント・アプリケーションで簡単に変更できます。

ALTER AUTHORIZATION 句の構文は次のとおりです。

```
EXEC SQL CONNECT .. ALTER AUTHORIZATION :newpswd END-EXEC.
```

この句を使うと、アカウントのパスワードが、`newpswd` で指定された値に変更されます。パスワードを変更すると、`user/newpswd` として接続試行が実行されます。次の結果が予想されます。

- アプリケーションが問題なく接続されます。
- アプリケーションが接続されません。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。

ALTER AUTHORIZATION を使用しない接続

この項では、別の種類の CONNECT 文に考えられる結果について説明します。

標準 CONNECT

次の文がアプリケーションから発行されると、

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

通常の接続が実行されます。次の結果が予想されます。

- アプリケーションが問題なく接続されます。

- アプリケーションは接続されますが、パスワードについての警告が出されます。この警告は、パスワードは期限切れになっているが、まだログインできる期間にあることを示します。この期間内にパスワードを変更するようにしてください。そうしないと、アカウントがロックされてしまいます。
- アプリケーションが接続されません。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、またはロック状態になっています。

SYSDBA 権限または SYSOPER 権限

Oracle8.1 より前のリリースでは、SYSOPER または SYSDBA システム権限をもつために次の句を使う必要はありませんでしたが、今回のリリースでは使用してください。

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

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

たとえば、次のとおりです。

```
EXEC SQL CONNECT ... IN SYSDBA MODE END-EXEC.
```

このオプションに適用される制限は、次のとおりです。

- AUTO_CONNECT=YES のプリコンパイラ・オプション設定を使用しているときは、このオプションはサポートされません。
- CONNECT 文で ALTER AUTHORIZATION キーワードを使用しているときは、このオプションは使用できません。

アドバンス・コネクション・オプション

ネットワーク内の通信ポイントをノードといいます。Net8 では、ネットワーク上のあるノードから別のノードへ情報（SQL 文およびデータ、状態コード）を送信できます。

プロトコルとはネットワークにアクセスするための一連の規則です。その規則は障害後の回復手順、データの転送およびエラーの検査のフォーマットなどを規定します。

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の Net8 の構文では、そのデータベースのサービス名を使うだけで済みます。

その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。たとえば、次のとおりです。

```
HR.US.ORACLE.COM
```

Net8 による接続

Net8 ドライバを使って接続するには、SQL*Net V1 接続文字列のかわりに、*tnsnames.ora* 構成ファイルまたは Oracle Names に定義されているサービス名を使用してください。

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。

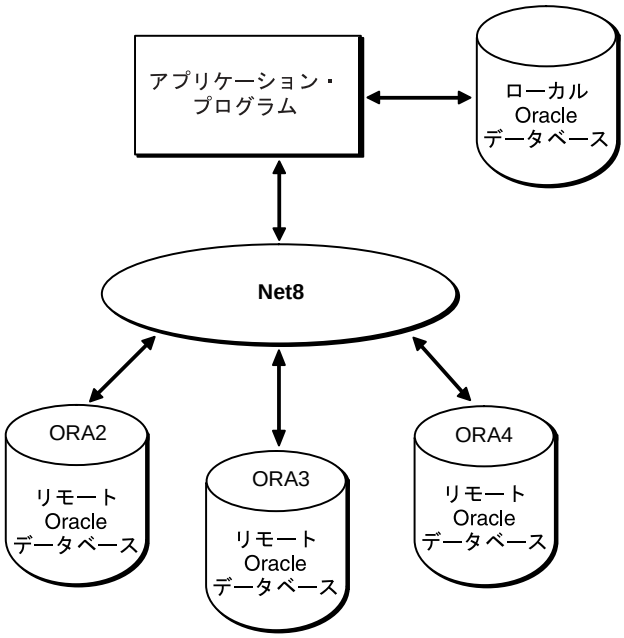
注意 : SQL*Net V1 は、Oracle8*i* で動作します。

Net8 の詳細は、『Net8 管理者ガイド』を参照してください。

同時ログイン

Pro*COBOL は、Net8 を介して分散処理をサポートします。アプリケーションは、ローカル・データベースとリモート・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 1 つのローカル Oracle8*i* データベースおよび 3 つのリモート Oracle8*i* データベースと通信しています。ORA2 および ORA3、ORA4 は CONNECT 文中で使用される論理名です。

図 3-1 Net8 を介した接続



Net8 は、ネットワーク上の異なるマシン間やオペレーティング・システム間に存在する境界を排除することによって、Oracle Tools に分散環境を提供します。この項では、Net8 を介した分散処理を Pro*COBOL がどのようにサポートするかについて説明します。アプリケーションで次の処理をどのように行うかを学びます。

- 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

リンクの使用方法

暗黙的ログインは、Oracle8*i* 分散データベース・オプションによってサポートされています。このオプションでは明示的ログインは必要とされません。たとえば、分散問合せでは、単一の SELECT 文で 1 つ以上の非デフォルト・データベース上のデータにアクセスできます。

分散問合せ機能ではデータベース・リンクを利用します。ここでは、接続そのものではなく CONNECT 文に名前を割り当てます。実行時には、指定されたデータベース・サーバーに

よって埋込み SELECT 文が実行されます。データベース・サーバーは非デフォルトのデータベースに暗黙的に接続し、必要なデータを取得します。

単一の暗黙的ログイン

次の例では、単一の非デフォルトのデータベースに接続します。最初に、プログラムは次の文を実行して、データベース・リンクを定義します。（データベース・リンクは通常、DBA またはユーザーによって対話的に確立されます。）

```
EXEC SQL CREATE DATABASE LINK db_link
   CONNECT TO username IDENTIFIED BY password USING 'nyremote'
END-EXEC.
```

その後、次に示すとおり、プログラムはデータベース・リンクを使って非デフォルトの EMP 表を問い合せることができます。

```
EXEC SQL SELECT ENAME, JOB INTO :EMP-NAME, :JOB-TITLE
    FROM emp@db_link
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

データベース・リンクは、埋込み SQL 文の AT 句で使われているデータベース名とは関係がありません。データベース・リンクは、非デフォルトのデータベースの場所、そのデータベースへのパス、使用するユーザー名およびパスワードを Oracle に伝えるだけです。データベース・リンクは、明示的に削除されるまでデータ・ディクショナリに格納されます。

上の例で、デフォルトの Oracle8*i* は、データベース・リンク *db_link* を使用し、Net8 を介して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、デフォルト以外のデータベースに先送りして実行されます。

データベース・リンクを簡単に参照するため、次のようにシノニムを作成できます（ここでも通常、対話的に作成します）。

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link END-EXEC.
```

さらに、プログラムは、次のようにして非デフォルト EMP 表を問い合せることができます。

```
EXEC SQL SELECT ENAME, JOB INTO :EMP-NAME, :JOB-TITLE
    FROM emp
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

これによって *emp* に位置が透過的になります。

複数の暗黙的ログイン

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。まず、2 つのデータベース・リンクを定義し、2 つのシノニムを作成するために次の順序の文を実行します。

```
EXEC SQL CREATE DATABASE LINK db_link1
    CONNECT TO username1 IDENTIFIED BY password1
    USING 'nyremote'
END-EXEC.
EXEC SQL CREATE DATABASE LINK db_link2
    CONNECT TO username2 IDENTIFIED BY password2
    USING 'chiremote'
END-EXEC.
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1 END-EXEC.
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2 END-EXEC.
```

プログラムは、次のようにして非デフォルトの EMP 表および DEPT 表の問合せができます。

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

Oracle8*i* は、*db_link1* の非デフォルトの EMP 表と *db_link2* の非デフォルトの DEPT 表とを結合して問合せを実行します。

OCI（Oracle Call Interface）コールの埋込み

Pro*COBOL では、プログラムに OCI コールを埋め込むことができます。手順は次のとおりです。

1. 宣言文がある場合は、宣言文の外で OCI ログイン・データ領域（LDA）を宣言します。詳細は『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』を参照してください。
2. OCI コール OLOG ではなく、埋込み SQL 文 CONNECT を使って Oracle に接続します。
3. Oracle8*i* ランタイム・ライブラリ・ルーチン SQLLDA をコールして、接続情報を LDA に格納します。

このようにすると、Pro*COBOL と OCI が共に動作することを認識します。ただし、Oracle8*i* のカーソルは共有されません。

Oracle8*i* ランタイム・ライブラリがユーザーにかわって接続の管理と OCI ホスト・データ領域（HDA）の維持を行うので、HDA の宣言を行う必要はありません。

LDA の設定

OCI コールを発行することによって、LDA を設定します。

```
CALL "SQLLDA" USING LDA.
```

このとき、*lda* は LDA データ構造を識別します。詳細は、『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』を参照してください。CONNECT 文が失敗すると、*lda* 内の *LDA-RC* フィールドがエラーを示す 1012 に設定されます。

リモートおよび複数接続

SQLLDA をコールすると、最後に実行した SQL 文で使用された接続の LDA が設定されます。追加の接続のために別の LDA を設定する場合は、それぞれの CONNECT の後に別の *lda* を指定して SQLLDA をコールします。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。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME   PIC X(10) .
 01  PASSWORD   PIC X(10) .
 01  DB-STRING1 PIC X(20) .
 01  DB-STRING2 PIC X(20) .
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
* -- Field sizes in LDA are system-dependent.
 01  LDA1.
     02 LDA1-V2RC  PIC S9(4) COMP.
     02 FILLER     PIC X(10).
     02 LDA1-RC    PIC S9(4) COMP.
     02 FILLER     PIC X(50).
 01  LDA2.
     02 LDA2-V2RC  PIC S9(4) COMP.
     02 FILLER     PIC X(10).
     02 LDA2-RC    PIC S9(4) COMP.
     02 FILLER     PIC X(50).
     ...
     MOVE 'SCOTT' TO USERNAME.
     MOVE 'TIGER' TO PASSWORD.
     MOVE 'nyremote' TO DB-STRING1.
     MOVE 'chiremote' TO DB-STRING2.
     ...
* --  give each database connection a unique name
     EXEC SQL DECLARE db_name1 DATABASE END-EXEC.
     EXEC SQL DECLARE db_name2 DATABASE END-EXEC.
     ...
* -- connect to first non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
         AT db_name1 USING :DB-STRING1
     END-EXEC.
* --  set up first LDA for OCI use
     CALL 'SQLLDA' USING LDA1.
* -- connect to second non-default database
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD
```

```
        AT db_name2 USING :DB-STRING2
    END-EXEC.
* --  set up second LDA for OCI use
    CALL 'SQLLDA' USING LDA2.
```

db_name1 および *db_name2* はホスト変数ではないので、宣言しないでください。db_name1 および db_name2 は、その後の SQL 文でデータベースを名前によって参照できるように、2 つの非デフォルト・ノードのデフォルト・データベースに名前を付けるためだけに使用します。

知っておくべき用語

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

データベースが管理するジョブおよびタスクをセッションといいます。ユーザー・セッションは、アプリケーション・プログラムまたは Oracle Forms などのツールを実行してデータベースに接続すると開始されます。Oracle8*i* では、複数のユーザー・セッションを同時に動作させ、コンピュータ・リソースを共用させることができます。このためには、並行性、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。並行性を適切に制御しないと、データの整合性が損なわれることがあります。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。

Oracle8*i* では、ロック機能を使用してデータへの同時アクセスを制御できます。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。つまり、このユーザーがデータの変更を終了するまで他のユーザーは同じデータの変更はできません。表のデータおよび構造はデフォルトのロック機構によって保護されるため、リソースを明示的に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変更した方が都合がよいときは、表または行単位でデータ・ロックを要求できます。行共有および行排他など、ロックのモードを選択できます。

複数のユーザーが同一のデータベース・オブジェクトにアクセスしようとすると、デッドロックが発生することがあります。たとえば、2 人のユーザーが同じ表を更新するときに、それぞれのユーザーがもう一方のユーザーによって現在ロックされている行を更新しようとしてお互いに待たされることがあります。それぞれのユーザーが相手側のロックしているリソースを待つことになるため、サーバーがデッドロックを解除するまでは両者とも処理を続行できません。最少量の作業を完了した関連トランザクションにサーバーからエラーが送られ、リソース待ちのときに検出されたデッドロックのエラー・コードが SQLCA の SQLCODE に返されます。

1 つの表をあるユーザーが問い合せ、同時に別のユーザーが更新している場合、データベースは問合せのためにその表のデータの読取り一貫性ビューを生成します。つまり一度問合せが開始されてから処理がそのまま続行しているときは、問合せによって読み込まれるデータは変化しません。更新アクティビティが続行している間、データベースは表のデータのスナップショットをとり、変更内容をロールバック・セグメントに記録します。データベースは、このロールバック・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を生成し、必要に応じて変更を取り消します。

トランザクションがデータベースを保護する方法

データベース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うことによってデータの整合性を保ちます。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、論理的に対応付けられた 1 つ以上の SQL 文です。データベースはこの一連の SQL 文を 1 つの単位として扱うため、これらの文による変更はすべて同時にコミット（確定）またはロールバック（取消し）されます。アプリケーション・プログラムがトランザクションの途中で異常終了すると、データベースは自動的に前の状態（トランザクション処理の前の状態）に復元されます。

以後の項では、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、次の方法を中心に説明します。

- トランザクションの開始および終了
- COMMIT 文を使ってのトランザクションの確定
- ROLLBACK TO 文とともに SAVEPOINT 文を使ってのトランザクションの部分的な取消し
- ROLLBACK 文を使ってのトランザクション全体の取消し
- RELEASE オプションを指定してのリソースの解放およびデータベースのログオフ
- SET TRANSACTION 文を使っての読取り専用トランザクションの設定
- FOR UPDATE 句または LOCK TABLE 文を使ってのデフォルトのロックの変更（上書き）

この章で説明する SQL 文の詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

トランザクションの開始および終了方法

プログラム内の最初の実行 SQL 文（CONNECT 以外）によって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 文が自動的に別のトランザクションを開始します。つまり、すべての実行文はトランザクションの一部です。宣言 SQL 文はロールバックできません。またこの文は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はみなされません。

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。これらの文は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。
- 実行の前後両方で自動コミットを発行するデータ定義文（ALTER または CREATE、GRANT など）を記述します。これは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、またはソフトウェア上の問題、ハードウェア上の問題、強制割込みなどが原因で予期しない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。Oracle8*i* は、終了したトランザクションをロールバックします。

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、Oracle8*i* 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した場合には、データベースは元の状態（トランザクション処理前の状態）に復元されます。

COMMIT 文の使用方法

COMMIT 文を使うとデータベースへの変更を確定できます。変更をコミットするまでは、他のユーザーは変更されたデータにアクセスできません。他のユーザーが参照すると、データはトランザクションが開始される前の状態で表示されます。COMMIT 文はホスト変数の値にもプログラム内の制御の流れにも影響しません。COMMIT 文は、次の操作を行います。

- 現在のトランザクション中に行ったデータベースに対する変更をすべて確定します。
- これらの変更が他のユーザーにわかるようにします。
- すべてのセーブポイントを消去します（次の項「SAVEPOINT 文の使用方法」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するか、または MODE={ANSI | ANSI14} の場合は、明示的な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。

MODE={ANSI13 | ORACLE} のときは、CURRENT OF 句で参照されていない明示的なカーソルはコミット後もオープン状態となります。これによってパフォーマンスが向上します。3-27 ページの「コミットを超えたフェッチ」の例を参照してください。

これらの処理は通常の処理の一部分になっているため、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。プログラムを終了する前に、保留中の変更を明示的にコミットしてください。コミットしなければ、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、切断します。

```
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

オプション指定のキーワード WORK は ANSI 互換性のために提供されています。RELEASE オプションを指定すると、プログラムが使用しているリソース（ロックおよびカーソル）がすべて解放され、データベースからログアウトされます。

データ定義文は実行の前後両方で自動コミットを発行するため、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。

DECLARE CURSOR 文での WITH HOLD 句の使用

CURSOR という語の後に WITH HOLD 句を付けて宣言されているカーソルは、COMMIT または ROLLBACK の後もオープン状態となります。次の例は、この句の使用方法を示したものです。

```
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、カーソルを宣言してはいけません。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 のデフォルトが使われますが、ホスト変数はすべて宣言文で宣言する必要があります。宣言文を省略するには、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。詳細は F-24 ページの「DECLARE CURSOR（埋込み SQL ディレクティブ）」を参照してください。

CLOSE_ON_COMMIT プリコンパイラ・オプション

DB2 との互換性を保つために、プリコンパイラ・オプション CLOSE_ON_COMMIT を使用できます。

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

デフォルトは NO です。このオプションは、コマンド行または構成ファイル内で指定しなければなりません。コマンド行で MODE=ANSI と指定した場合は、WITH HOLD 句を使わ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。

> **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およびクローズの回数が多いと、OPEN 文のたびに再解析が行われるため、アプリケーションの動作が遅くなることがあります。詳細は 14-14 ページの「CLOSE_ON_COMMIT」を参照してください。

ROLLBACK 文の使用方法

ROLLBACK 文を使うと、保留状態になっている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てしまったときなどは、ROLLBACK 文を使えば元のデータを復元できます。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。ROLLBACK 文は、次の操作を行います。

- 現在のトランザクション中に実行されたデータベースへ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するか、または MODE={ANSI | ANSI14} の場合は、明示的なカーソルをすべてクローズします。

MODE={ANSI13 | ORACLE} のときは、CURRENT OF 句で参照されていない明示的なカーソルはロールバック後もオープン状態となります。

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、切断します。

```
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

オプション指定のキーワード WORK は ANSI 互換性のために提供されています。RELEASE オプションを指定すると、プログラムが使用しているリソースがすべて解放され、データベースからログアウトされます。

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、ロールバックでエラーが発生すると、プログラムが無限ループに陥る可能性があります。このような事態を避けるには、ROLLBACK 文の前に WHENEVER SQLERROR CONTINUE を記述します。

次に例を示します。

```
     EXEC SQL
         WHENEVER SQLERROR GOTO SQL-ERROR
     END-EXEC.
     ...
     DISPLAY 'Employee number? '.
     ACCEPT EMP-NUMBER.
     DISPLAY 'Employee name? '.
     ACCEPT EMP-NAME.
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
        VALUES (:EMP-NUMBER, :EMP-NAME)
     END-EXEC.
     ...
 SQL-ERROR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     DISPLAY 'Processing error'.
* -- exit program with an error.
     STOP RUN.
```

プログラムが異常終了すると、Oracle8*i* はトランザクションをロールバックします。

文レベルのロールバック

Oracle8*i* 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。SQL 文でエラーが発生すると、Oracle8*i* は自動的にその文をロールバックし、該当するエラー・コードを SQLCA 内の SQLCODE に戻します。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。

ロールバックによって失われるのは、エラーとなった SQL 文から後の作業だけです。現行のトランザクションでその文より前に行われた作業は保存されます。データ定義文がエラーとなった場合でも、それ以前の自動コミットは取り消されません。

> **注意 :** Oracle8*i* は、SQL 文を解析してから実行します。つまり実行前に、SQL 文に正しい構文ルールが使われ有効なデータベース・オブジェクトを参照しているかを確認します。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、SQL 文の解析中にエラーが検出されても文はロールバックされません。

Oracle8*i* 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。デッドロックの原因となっているトランザクションの 1 つにエラーを通知して、そのトランザクションの現行の文をロールバックします。

SAVEPOINT 文の使用方法

SAVEPOINT 埋込み SQL 文を使うと、トランザクションの処理の現時点にマークを設定し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 END-EXEC.
```

セーブポイントによって長い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。この結果、あるファンクションでエラーが発生しても、簡単にデータを元の状態に復元し、回復して、そのファンクションを再実行できます。

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 句によってセーブポイントを指定します。TO SAVEPOINT 句を使うと、現在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。ROLLBACK TO SAVEPOINT 文は、次の操作を行います。

- 指定されたセーブポイント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位置設定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。

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、SQLCA の SQLERRD(3) をチェッ

クして、削除された行数を調べます。削除された行数が必要以上に大きいときは、セーブポイントの *start_delete* までロールバックしてこの削除を取り消します。

```
* -- For each new customer
     DISPLAY 'New customer number? '.
     ACCEPT CUST-NUMBER.
     IF CUST-NUMBER = 0
          GO TO REV-STATUS
     END-IF.
     DISPLAY 'New customer name? '.
          ACCEPT  CUST-NAME.
     EXEC SQL INSERT INTO MAIL-LIST (CUSTNO, CNAME, STAT)
         VALUES (:CUST-NUMBER, :CUST-NAME, 'ACTIVE').
     END-EXEC.
     ...
* -- For each revised status
 REV-STATUS.
     DISPLAY 'Customer number to revise status? '.
     ACCEPT CUST-NUMBER.
     IF CUST-NUMBER = 0
         GO TO SAVE-POINT
     END-IF.
     DISPLAY 'New status? '.
     ACCEPT NEW-STATUS.
     EXEC SQL UPDATE MAIL-LIST
        SET STAT = :NEW-STATUS WHERE CUSTNO = :CUST-NUMBER
     END-EXEC.
     ...
* -- mark savepoint
 SAVE-POINT.
     EXEC SQL SAVEPOINT START-DELETE END-EXEC.
     EXEC SQL DELETE FROM MAIL-LIST WHERE STAT = 'INACTIVE'
     END-EXEC.
     IF SQLERRD(3) < 25
* -- check number of rows deleted
         DISPLAY 'Number of rows deleted is ', SQLERRD(3)
     ELSE
         DISPLAY 'Undoing deletion of ', SQLERRD(3), ' rows'
         EXEC SQL
             WHENEVER SQLERROR GOTO SQL-ERROR
         END-EXEC
         EXEC SQL
             ROLLBACK TO SAVEPOINT START-DELETE
         END-EXEC
     END-IF.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

```
     STOP RUN.
* -- exit program.
     ...
 SQL-ERROR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     DISPLAY 'Processing error'.
* -- exit program with an error.
     STOP RUN.
```

ROLLBACK TO SAVEPOINT 文では RELEASE オプションを指定できないことに注意してください。

あるセーブポイントまでをロールバックすることによって、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、5 つのセーブポイントを設定しているときに 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だけが消去されます。COMMIT 文または ROLLBACK 文を実行すると、すべてのセーブポイントが消去されます。

RELEASE オプションの使用方法

プログラムが異常終了すると、Oracle8*i* は自動的に変更をロールバックします。異常終了は、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しないまま、RELEASE 埋込み SQL 文を使用して接続を切断した場合に発生します。

これに対し、プログラムが所定作業の実行、オープン・カーソルのクローズ、作業の明示的なコミットまたはロールバック、接続の切断を行って、制御をユーザーに戻した場合には、プログラムは正常に終了します。実行される最後の SQL 文が次のいずれかのときにプログラムは正常終了します。

```
EXEC SQL COMMIT RELEASE END-EXEC.
```

または

```
EXEC SQL ROLLBACK RELEASE END-EXEC.
```

最後の SQL 文が上のどちらでもない場合は、そのユーザー・セッションが使用している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 Oracle8*i* が認識するまで保持されたままになります。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ことがあります。

SET TRANSACTION 文の使用方法

SET TRANSACTION 文を使用すると、読込み専用または読み書き両用のトランザクションを開始したり、現行のトランザクションを特定のロールバック・セグメントに割り当てたりできます。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。

読込み専用トランザクションで " 繰返し可能読取り " ができます。これは別のユーザーが 1 つ以上の表を更新している間に、同じ表について複数の問合せを実行するときに適しています。読込み専用トランザクションでは、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するため、複数の表、複数の問合せの読取り一貫性ビューが生成されます。その他のユーザーは通常どおりデータの問合せまたは更新を続行できます。SET TRANSACTION 文の例を次に示します。

```
EXEC SQL SET TRANSACTION READ ONLY END-EXEC.
```

SET TRANSACTION 文は読込み専用トランザクション内の最初の SQL 文でなければなりません。また、1 つのトランザクション内では一度しか使うことができません。READ ONLY パラメータは必須です。READ ONLY パラメータを使っても他のトランザクションに影響はありません。読込み専用トランザクションで使用できる文は、SELECT（FOR UPDATE の場合を除く）および LOCK TABLE、SET ROLE、ALTER SESSION、ALTER SYSTEM、COMMIT、ROLLBACK だけです。

次に示すのは、ある商店の経営者が、読込み専用トランザクションを使ってその日の販売活動および過去 1 週間の販売活動、過去 1 か月間の販売活動をチェックして、要約レポートを作成する例です。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 END-EXEC.
     EXEC SQL SELECT SUM(SALEAMT) INTO :DAILY FROM SALES
         WHERE SALEDATE = SYSDATE END-EXEC.
     EXEC SQL SELECT SUM(SALEAMT) INTO :WEEKLY FROM SALES
         WHERE SALEDATE > SYSDATE - 7 END-EXEC.
     EXEC SQL SELECT SUM(SALEAMT) INTO :MONTHLY FROM SALES
         WHERE SALEDATE > SYSDATE - 30 END-EXEC.
     EXEC SQL COMMIT WORK END-EXEC.
* --  simply ends the transaction since there are no changes
* --  to make permanent
* --  format and print report
```

デフォルトのロックの変更

デフォルトでは、Oracle8*i* によって多数のデータ構造が暗黙的に（自動的に）ロックされます。ただし、デフォルトのロックを無効にして、別のロックを有効にしたいときは、行や表を特定して、そこにデータ・ロックを要求できます。明示的なロックを行うと、トランザクション中に表へのアクセスを共有したり拒絶したりでき、また、複数の表および複数の問合せの読取り一貫性を確保できます。

SELECT FOR UPDATE OF 文を使用すると、表の特定の行を明示的にロックして、更新または削除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、Oracle8*i* では、更新時または 削除時には自動的に行レベルのロックが行われます。したがって、UPDATE または DELETE の前に行をロックするときにかぎり FOR UPDATE OF 句を使用してください。

LOCK TABLE 文を使うと、表全体を明示的にロックできます。

FOR UPDATE OF 句の使用方法

UPDATE 文または DELETE 文の CURRENT OF 句内で参照されるカーソルを DECLARE するときに、FOR UPDATE OF 句を使うと行の排他ロックを取得できます。SELECT FOR UPDATE OF によって、まず更新または削除対象とする行が決定し、次にこのアクティブ・セット内のそれぞれの行がロックされます。（どの行も、フェッチの時点ではなくオープン時にロックされます。）この設定は、ある行内の既存の値に従って更新処理を行うときなどに便利です。更新前に別のユーザーによってその行が更新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、次のような文があるとします。

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
    SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20
    FOR UPDATE OF SAL
END-EXEC.
```

FOR UPDATE OF 句を削除すると、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
END-EXEC.
```

CURRENT OF 句は、必要に応じて FOR UPDATE 句を追加するようプリコンパイラに指示します。CURRENT OF 句を使用すると、最後にカーソルからフェッチした行を参照できます。5-15 ページの「CURRENT OF 句の使用方法」の例を参照してください。

制限

FOR UPDATE OF 句を使うと、5-15 ページの複数の表を参照できなくなります。また、明示的な FOR UPDATE OF でも暗黙的な FOR UPDATE でも、排他的な行ロックを獲得できます。行ロックは、コミットまたはロールバック（セーブポイントまでのロールバックは除

く）が行われると解除されます。コミットした後で FOR UPDATE カーソルからフェッチしようとすると、エラーが発生します。

LOCK TABLE 文の使用方法

LOCK TABLE 文を使うと、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。行共有ロックによって表への同時アクセスが可能となります。つまり、他のユーザーがその表全体をロックして排他使用することはできなくなります。

```
EXEC SQL
    LOCK TABLE EMP IN ROW SHARE MODE NOWAIT
END-EXEC.
```

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 ユーザーだけです。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内の行の挿入または更新、削除を行えません。ロック・モードの詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

表が別のユーザーによってロックされている場合に、オプションのキーワード NOWAIT を指定すると、Oracle8*i* ではその表の解放を待たずに作業を行なうことができます。制御はすぐにプログラムに戻されます。このため、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。（表のロックが成功したかどうかは、SQLCA の SQLCODE を調べればわかります。）NOWAIT を指定しないと、Oracle8*i* はその表が使用可能になるまで待ちます。待機時間に制限はありません。

表ロックによって他のユーザーからの表の問合せが禁止されることはありません。このため、問合せで表ロックが取得されることはありません。したがって、問合せが他の問合せまたは更新の妨げになることもありません。また、更新が問合せの妨げになることもありません。2 つの異なるトランザクションが同じ行を更新しようとしたときだけ、一方のトランザクションは他方のトランザクションが終了するまで待ち状態になります。トランザクションがコミットまたはロールバックを発行すると、表ロックは解除されます。

コミットを超えたフェッチ

コミットとフェッチを併用する場合は、CURRENT OF 句を使用しないでください。かわりに、各行の ROWID を選択し、その値によって更新または削除対象の現在行を識別します。次の例を考えてみます。

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
    SELECT ENAME, SAL, ROWID FROM EMP WHERE JOB = 'CLERK'
END-EXEC.
...
EXEC SQL OPEN EMP-CURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO ...
```

```
PERFORM
EXEC SQL
    FETCH EMP-CURSOR INTO :EMP_NAME, :SALARY, :ROW-ID
END-EXEC
...
    EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
        WHERE ROWID = :ROW-ID
    END-EXEC
    EXEC SQL COMMIT END-EXEC
END-PERFORM.
```

ただし、フェッチされた行はロックされないので注意してください。つまり、ある行を読み込んでも、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してしまうと、結果に矛盾が生じる可能性があります。

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文が少なくとも 1 つ設定されています。その更新が 1 つのノードにだけ影響するときは、トランザクションは分散型ではありません。

コミットを実行すると、分散トランザクションの対象となっている各データベースに対する変更が確定されます。ロールバックを実行すると、変更はすべて取り消されます。ただし、コミットまたはロールバック中にネットワークまたは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が不明またはインダウトになることがあります。そのようなときに FORCE TRANSACTION システム権限があれば、FORCE 句によってローカル・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、トランザクション ID を引用符付きリテラルで囲んで指定する必要があります。トランザクション ID はデータ・ディクショナリ・ビュー DBA_2PC_PENDING に収められています。次にいくつかの例を示します。

```
EXEC SQL COMMIT FORCE '22.31.83' END-EXEC.
...
EXEC SQL ROLLBACK FORCE '25.33.86'END-EXEC.
```

FORCE は指定されたトランザクションだけをコミットまたはロールバックするため、現行のトランザクションには影響しません。未確定のトランザクションはセーブポイントに手動でロールバックできないことに注意してください。

COMMIT 文中の COMMENT 句を使うと、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状態の場合、サーバーは、COMMENT で指定したテキストをトランザクション ID とともにデータ・ディクショナリ・ビュー DBA_2PC_PENDING に格納します。COMMENT で指定するテキストは、長さ 50 文字以下の引用符付きリテラルでなければなりません。次に例を示します。

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

分散トランザクションの詳細は、『Oracle8*i* 概要』を参照してください。

トランザクション処理のガイドライン

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。

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、すべて過不足なく盛り込まれています。

表内で参照するデータは一貫したものでなければなりません。したがって、トランザクション内の SQL 文は一貫した方法に従ってデータを変更する必要があります。たとえば 2 種類の銀行預金口座間の資金の送金取引の場合は、一方の口座に対する借方記帳と他方の口座に対する貸方記帳の処理がトランザクションに含まれていなければなりません。どちらの処理も同時に正常終了または失敗する必要があります。一方の口座への新規預金などといった、この取引とは無関係の更新取引はトランザクションには取り込まないでください。

ロックの取得

アプリケーション・プログラム内に SQL のロック文がある場合、ロックを要求するユーザーはそのロックの獲得に必要な権限をもっていなければなりません。データベース管理者（DBA）はどの表でもロックできます。DBA 以外のユーザーは、自分が所有する表または権限をもつ表（ALTER、SELECT、INSERT、UPDATE、DELETE など）だけをロックできます。

PL/SQL の使用方法

トランザクションに PL/SQL ブロックが記述されている場合、PL/SQL ブロック内で指定されたコミットおよびロールバックはトランザクション全体を対象に行われます。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;
```

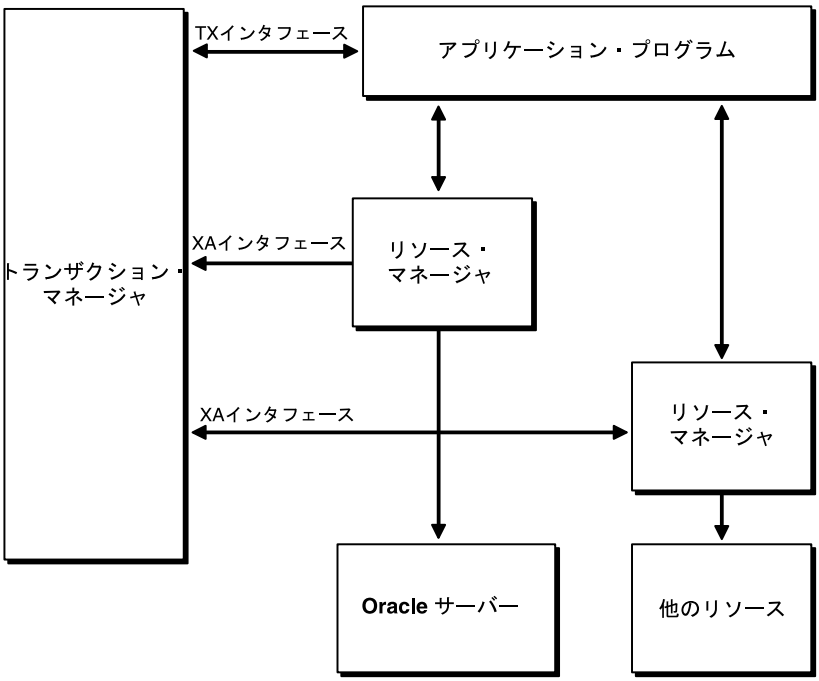
```
END;
END-EXEC.
...
```

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）に各種のサービスの提供を求めます。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ために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 3-2 に、Oracle8*i* データベース内のデータに効果的にアクセスするために DTP モデルのコンポーネントが行う対話の一例を示します。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・マネージャとの間に XA インタフェースが指定されています。Oracle は XA 準拠ライブラリを提供します。このライブラリは、X/Open アプリケーションにリンクする必要があります。また、アプリケーション・プログラムとリソース・マネージャ間で固有のインタフェースを指定する必要もあります。

図 3-2　仮定的な DTP モデル



トランザクション・マネージャおよびリソース・マネージャがアプリケーション・プログラムと対話する方法を指定している DTP モデルについては、X/Open のガイドである『Distributed Transaction Processing Reference Model』および関連する文書に説明があります。これらを入手するには、次の宛先に書面で問い合せてください。

X/Open Company Ltd.

1010 El Camino Real, Suite 380

Menlo Park, CA 94025

XA インタフェースの使用方法の詳細は、『Transaction Processing (TP)Monitor user's guide』を参照してください。

Oracle 固有の問題

Pro*COBOL を使って、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、次の必要条件を満たさなければなりません。

Oracle への接続

X/OPEN アプリケーションはデータベースとの接続の確立も維持もしません。かわりに、トランザクション・マネージャおよび XA インタフェースが Oracle によって供給され、データベースとの接続と接続の切離しを透過的に取り扱います。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。

トランザクション制御

X/OPEN アプリケーションで、グローバル・トランザクションに影響を与える COMMIT、ROLLBACK、SAVEPOINT、SET TRANSACTION などの文を実行してはいけません。たとえば、アプリケーションで COMMIT 文を実行してはいけません。トランザクション・マネージャがコミットを処理するためです。また、CREATE および ALTER、RENAME などの SQL データ定義文は暗黙的なコミットを発行するので、X/Open アプリケーションではこれらの文も実行してはいけません。

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れば、内部 ROLLBACK 文を実行できます。ただし、これは XA インタフェースの将来のバージョンでは変更になる可能性があります。

OCI コール

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 SQLLD2 を使用しなければなりません。このルーチンにより、XA インタフェースを介して確立された、指定された接続用の LDA が設定されます。SQLLD2 コールの説明は、『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』を参照してください。OCOM および OCON、OCOF、ORLON、OLON、OLOG、OLOGOF は X/Open アプリケーションでは発行できないことに注意してください。

リンク

XA の機能を使用するには、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する必要があります。手順については、使用しているシステム固有の Oracle8*i* マニュアルを参照してください。

4

データ型とホスト変数

この章では、Pro*COBOL プログラムを作成する際に必要となる基本的な情報を提供します。この章の構成は、次のとおりです。

Oracle8*i* のデータ型

Oracle8*i* は、内部データ型と外部データ型の 2 種類のデータ型を認識します。内部データ型は、Oracle8*i* がデータをデータベース列にどのように格納するかを指定します。Oracle8*i* は、データベース疑似列を表す場合にも内部データ型を使用します。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。

内部データ型

Oracle では、データベース列に値を格納するために次の内部データ型を使用します。

表 4-1 内部データ型

名前	コード	説明
CHAR	96	<= 2000 バイト、固定長文字列
NCHAR	96	<= 2000 バイト、固定長 1 バイトまたは固定幅マルチバイト文字列
DATE	12	7 バイト、固定長日付 / 時刻値
LONG	8	<= 2147483647 バイト、可変長文字列
LONG RAW	24	<= 2147483647 バイト、可変長バイナリ・データ
NUMBER	2	任意にコーディングされた 10 進書式で表される固定または浮動小数点数
RAW	23	<= 255 バイト、可変長バイナリ・データ
ROWID	11	固定長バイナリ値
VARCHAR2	1	<= 4000 バイト、可変長文字列
NVARCHAR2	1	<= 4000 バイト、可変長 1 バイトまたは固定幅マルチバイト文字列

CHAR

CHAR データ型は、固定長文字データの格納に使用します。データが内部でどのように表されるかは、データベースのキャラクタ・セットによって決まります。CHAR データ型は、2000 バイト以内の最大幅を指定するオプション・パラメータをとります。構文は次のとおりです。

```
CHAR[(maximum_width)]
```

最大幅を指定しない場合のデフォルト値は 1 です。CHAR(*n*) 列の最大幅は、文字数ではなくバイト数で指定するように注意してください。したがって、CHAR(*n*) 列にマルチバイト（2 バイト）文字を格納する場合、最大幅は *n* / 2 より小さくなります。

NCHAR

このデータ型を使って、NLS（各国語サポート）文字列を格納します。詳細は、4-36 ページの「各国語サポート」を参照してください。NCHAR 値は、内部データ型には変換できず、SQLCHECK=SEMANTICS（または FULL）を指定して意味検査を実行するときに、宣言表の中だけで使われます。意味検査の詳細は、E-3 ページの「SQLCHECK=SEMANTICS の指定」を参照してください。

この埋込み SQL ディレクティブの説明と構文図は、F-29 ページの「DECLARE TABLE（Oracle 埋込み SQL ディレクティブ）」を参照してください。NCHAR 列に CHAR 値を挿入することはできません。CHAR 列に NCHAR 値を挿入することもできません。データ型を同値化する VAR 文には、このデータ型は使用できません。

DATE

DATE データ型を使って、7 バイト固定長フィールドに日時を格納します。日付部分のデフォルトは現在の月の最初の日付、時刻部分のデフォルトは午前 0 時です。

内部的には、DATE はバイナリ形式で格納されます。プログラムで DATE 列値を文字列に変換すると、Oracle8*i* はそのセッションにデフォルトの書式マスクを使用します。ユリウス暦の日付など、別の日付 / 時刻情報が必要な場合は、TO_CHAR 関数と書式マスクを使用してください。DATE 列値から文字列への変換や、その逆の変換を行う場合は、必ず VARCHAR2 や STRING などの（外部）文字データ型を使います。

LONG

LONG データ型は、可変長文字列の格納に使用します。LONG 列にはテキストや文字の配列を格納でき、短いドキュメントの格納もできます。LONG データ型は VARCHAR2 データ型と似ていますが、LONG 列の最大幅は 2147483647 バイト（2 ギガバイト）である点が異なります。

LONG 列は、UPDATE 文および INSERT 文、（大部分の）SELECT 文で使用できますが、式やファンクション・コール、WHERE、GROUP BY、CONNECT BY などの SQL 句では使用できません。1 つのデータベース表で使用できる LONG 列は 1 つだけであり、その列に索引を付けることはできせん。

LONG RAW

LONG RAW データ型は、可変長バイナリ・データまたは可変長バイト列の格納に使用します。LONG RAW 列の最大幅は 2147483647 バイト（2 ギガバイト）です。

LONG RAW データは LONG データと似ていますが、Oracle8*i* 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。LONG データに適用される制限事項は LONG RAW データにも適用されます。

NUMBER

NUMBER データ型は、事実上任意のサイズの固定小数点数または浮動小数点数を格納するために使用します。精度に桁数の合計を指定し、位取りで丸めを行うかどうかを指定できます。

NUMBER 値の最大精度は 38、大きさの範囲は 1.0E-129 ～ 9.99E125 です。位取りの範囲は -84 ～ 127 です。たとえば、位取りに -3 を指定すると、値は千の位に丸められます。（3456 は 3000 になります。）位取りに 2 を指定すると、値は百分の一の位に丸められます。（3.456 は 3.46 になります。）

精度と位取りを指定すると、Oracle8*i* はデータを格納する前に特別な整合性チェックを行います。値が精度を超えているとエラー・メッセージが発行され、位取りを超えている値は丸められます。

RAW

RAW データ型は、バイナリ・データまたはバイト文字列（一連のグラフィック文字など）の格納に使用します。RAW データは Oracle8*i* では解釈されません。

RAW データ型は、255 バイト以内の最大幅を指定するための必須パラメータをとります。構文は次のとおりです。

```
RAW(maximum_width)
```

最大幅の指定に定数や変数は使用できません。必ず整数リテラルを使用してください。

RAW データは CHAR データと似ていますが、Oracle8*i* では RAW データの意味は解釈されず、RAW データをあるシステムから別のシステムへ（たとえば 7 ビット ASCII から EBCDIC コード・ページ 500 へ）送信しても、キャラクタ・セットは変換されません。

ROWID

内部的には、Oracle8*i* データベース内の表にはすべて ROWID という名前の疑似列があります。この疑似列には ROWID というバイナリ値が格納されます。ROWID によって行を一意に識別でき、目的の行に素早くアクセスできます。

ROWID の使用方法の詳細は、4-34 ページの「ユニバーサル ROWID」を参照してください。

VARCHAR2

VARCHAR2 データ型を使って、可変長文字列を格納します。文字列が内部でどのように表されるかは、データベースのキャラクタ・セット（7 ビットの ASCII や EBCDIC コード・ページ 500 など）によって決まります。

VARCHAR2 データベース列の最大幅は 4000 バイトです。VARCHAR2 列を定義するには次の構文を使用します。

```
VARCHAR2(maximum_width)
```

maximum_width は 1 ～ 2000 の範囲の整数リテラルです。

VARCHAR2(*n*) 列の最大幅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*) 列にマルチバイト（2 バイト）文字を格納する場合、最大幅は *n* / 2 文字より小さくなります。

NVARCHAR2

NVARCHAR2 データ型は、可変長 NLS 文字データの格納に使用します。固定幅キャラクタ・セットの場合は、最大長を文字単位で指定します。可変幅キャラクタ・セットの場合は、最大長をバイト単位で指定します。詳細は、4-36 ページの「各国語サポート」を参照してください。NVARCHAR2 値は、内部データ型には変換できず、SQLCHECK=SEMANTICS（または FULL）を指定して意味検査を実行するときに、宣言表の中でだけ使われます。意味検査の詳細は、E-3 ページの「SQLCHECK=SEMANTICS の指定」を参照してください。この埋込み SQL ディレクティブの説明と構文図は、F-29 ページの「DECLARE TABLE（Oracle 埋込み SQL ディレクティブ）」を参照してください。NVARCHAR2 列に VARCHAR2 値を挿入することはできません。また、VARCHAR2 列に NVARCHAR2 値を挿入することもできません。データ型を同値化する VAR 文には、このデータ型は使用できません。

SQL 疑似列と関数

SQL は、表 4-2 に示す疑似列を認識します。これらの疑似列は、特定のデータ項目を戻します。

表 4-2 疑似列と内部データ型

疑似列	内部データ型
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID
ROWNUM	NUMBER

疑似列は、表に実際に存在する列ではありませんが、列のように扱われます。そのため、疑似列の値は表から SELECT する必要があります。疑似列の値はダミー表から SELECT するのが便利なときもあります。

SQL は、表 4-3 に示すパラメータなしの関数も認識します。これらの関数も、特定のデータ項目を戻します。

表 4-3 関数と内部データ型

関数	内部データ型
SYSDATE	DATE
UID	NUMBER
USER	VARCHAR2

SQL の疑似列および関数は、SELECT 文、INSERT 文、UPDATE 文および DELETE 文で参照できます。次の例では、従業員の雇用後の月数の計算に SYSDATE を使用しています。

```
EXEC SQL SELECT MONTHS_BETWEEN(SYSDATE, HIREDATE)
    INTO :MONTHS-OF-SERVICE
    FROM EMP
    WHERE EMPNO = :EMP-NUMBER
END EXEC.
```

これ以降では、SQL の疑似列および関数について簡単に説明します。詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

CURRVAL は、指定された順序における現在の番号を戻します。CURRVAL を参照する前に、NEXTVAL を使って順序番号を作成する必要があります。

LEVEL は、ツリー構造におけるノードのレベル番号を戻します。ルートはレベル 1、ルートの子はレベル 2、孫はレベル 3 になります。

SELECT CONNECT BY 文で LEVEL を使用して、表の一部の行または全部の行をツリー構造に取り込むことができます。また、ORDER BY 句または GROUP BY 句で LEVEL を使用すると、ツリー内の各レベルでデータが分離されます。

問合せでツリーが検索される方向（ルートから下へ、または分岐から上へ）は、PRIOR 演算子で指定します。ツリーのルートを識別する条件は START WITH 句で指定します。

NEXTVAL は、指定された順序における次の番号を戻します。順序を作成したら、それを使用してトランザクション用に一意の順序番号を作成できます。次の例では、順序 *partno* を使って部品番号を割り当てます。

```
EXEC SQL INSERT INTO PARTS
    VALUES (PARTNO.NEXTVAL, :DESCRIPTION, :QUANTITY, :PRICE
END EXEC.
```

トランザクションで順序番号が生成された場合、そのトランザクションをコミットまたはロールバックすると順序番号が増分されます。NEXTVAL を参照すると、現在の順序番号が CURRVAL に格納されます。

ROWNUM は、表から行が選択された順序を示す番号を戻します。最初に選択された行の ROWNUM は 1 に、2 番目に戻された行の ROWNUM は 2 になります。SELECT 文に ORDER BY 句が含まれている場合は、ソート選択された行に ROWNUM が割り当てられた後に、ソートされます。

ROWNUM を使って、SELECT 文で戻される行数を制限できます。また、UPDATE 文で ROWNUM を使用して、表の各行に一意の値を割り当てることもできます。WHERE 句で ROWNUM を使用した場合、取り出される行数が制限されるだけで、SELECT 文の処理は停止されません。WHERE 句での ROWNUM の使用方法として適切なのは次の方法だけです。

```
... WHERE ROWNUM < constant END-EXEC.
```

これは、ROWNUM の値は行が取り出されるときにだけ増加するためです。次のように指定した場合、4 行目までは取り出されないので、この検索条件が満たされることはありません。

```
... WHERE ROWNUM = 5 END-EXEC.
```

SYSDATE は、現在の日付と時刻を戻します。

UID は、Oracle ユーザーに割り当てられた一意の ID 番号を戻します。

USER は、現在の Oracle ユーザーのユーザー名を戻します。

外部データ型

表 4-4 に示すように、外部データ型には内部データ型がすべて含まれ、さらに、サポートされる他のホスト言語で使用され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 では NULL 終了記号付きの文字列にあたります。データ型の同値化ではデータ型名を使用し、動的 SQL の方法 4 ではデータ型コードを使用します。

表 4-4 外部データ型

名前	コード	説明
CHAR	1	<= 65535 バイト、可変長文字列（1）
	96	<= 65535 バイト、固定長文字列（1）
CHARF	96	<= 65535 バイト、固定長文字列
CHARZ	97	<= 65535 バイト、固定長、NULL 終了文字列（2）
DATE	12	7 バイト、固定長日付 / 時刻値
DECIMAL	7	COBOL パック 10 進数
DISPLAY	91	COBOL 数値文字列
DISPLAY TRAILING	152	COBOL 後続符号付き数値
FLOAT	4	4 バイトまたは 8 バイトの浮動小数点数

表 4-4 外部データ型 (続き)

名前	コード	説明
INTEGER	3	2 バイトまたは 4 バイトの符号付き整数
LONG	8	<= 2147483647 バイト、固定長文字列
LONG RAW	24	<= 217483647 バイト、固定長バイナリ・データ
LONG VARCHAR	94	<= 217483643 バイト、可変長文字列
LONG VARRAW	95	<= 217483643 バイト、可変長バイナリ・データ
NUMBER	2	整数または浮動小数点数
OVER-PUNCH LEADING	172	埋込み先行符号付き数値
OVER-PUNCH TRAILING	154	埋込み後続符号付き数値
RAW	23	<= 65535 バイト、固定長バイナリ・データ（2）
ROWID	11	固定長バイナリ値（システム固有）
STRING	5	<= 65535 バイト、NULL 終了文字列（2）
UNSIGNED	68	2 バイトまたは 4 バイトの符号なし整数
UNSIGNED DISPLAY	153	COBOL 符号なし数値
VARCHAR	9	<= 65533 バイト、可変長文字列
VARCHAR2	1	<= 65535 バイト、可変長文字列（2）
VARNUM	6	可変長 2 進数
VARRAW	15	<= 65533 バイト、可変長バイナリ・データ

注意 :

1. CHAR のデータ型コードは、PICX=VARCHAR2 のときには 1、PICX=CHARF のときには 96 になります。
2. 一部のプラットフォームでは、最大サイズは 32767（32K）です。

CHAR

CHAR の動作は、オプション PICX の設定によって異なります。詳細は、14-34 ページの「PICX」を参照してください。

CHARF

デフォルトでは、Oracle8*i* はすべての非可変長文字ホスト変数に CHARF データ型を割り当てます。CHARF データ型は、固定長文字列の格納に使用します。ほとんどのプラットフォームでは、CHARF 値の最大長は 65535（64K）バイトです。詳細は、14-34 ページの「PICX」を参照してください。

入力時 : Oracle8*i* は、入力ホスト変数に指定したバイト数を読み込み、後続する空白を切り捨てないでターゲット・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。すべて空白の入力値は、Oracle8*i* では文字値として扱われます。

出力時 : Oracle8*i* は、出力ホスト変数について指定された数のバイトを戻し、必要に応じて空白を埋め込み、出力値をターゲット・ホスト変数に割り当てます。NULL が戻された場合、ホスト変数は空白で埋められます。

出力値がホスト変数の宣言長より長い場合、Oracle8*i* 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。

CHARZ

CHARZ データ型は、固定長の NULL 終了文字列の格納に使用します。ほとんどのプラットフォームでは、CHARZ 値の最大長は 65535 バイトです。Pro*COBOL では、この外部型は必要ありません。

入力時、CHARZ データ型および STRING データ型は同じように機能します。入力値には NULL 終了記号を付ける必要があります。NULL 終了記号は文字列を区切るためのもので、データの一部ではありません。

出力時には、CHARZ データ型は CHAR データ型と同じように動作します。Oracle8*i* は出力値に NULL 終了記号を追加し、必要であれば空白を埋め込みます。

DATE

DATE データ型は、日付と時刻を 7 バイトの固定長フィールドに格納するために使用します。表 4-5 に示すように、世紀、年、月、日、時（24 時間制）、分、秒は、左から右にこの順序で格納されます。

表 4-5 日付書式

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

17-OCT-1994 at 1:23:12 PM

世紀および年のバイトは 100 を加えた表記です。時刻、分、秒のバイトは 1 を加えた表記です。西暦紀元前（B.C.E.）の日付は 100 より小さくなります。新紀元は、西暦紀元前 4712 年 1 月 1 日から始まります。この日の世紀のバイトは 53、年のバイトは 88、時のバイト範囲は 1 ～ 24、分および秒のバイト範囲は 1 ～ 60 になります。時刻のデフォルトは真夜中（1, 1, 1）です。

DECIMAL

Pro*COBOL では、DECIMAL データ型は計算用にパック 10 進数を格納するために使用します。COBOL では、ホスト変数は暗黙的な小数点をもつ符号付き COMP-3 フィールドでなければなりません。データ変換中に有効数字が失われると、Oracle8*i* はホスト変数にアスタリスクを埋め込みます。

DISPLAY

Pro*COBOL では、DISPLAY データ型は数字データの格納に使用します。DISPLAY データ型は、COBOL の "DISPLAY SIGN LEADING SEPARATE" の数値を参照します。PIC S9(*n*) の場合には (*n*)+1 バイトの記憶域を、PIC S9(*n*)V9(*d*) の場合には (*n*)+(*d*)+1 バイトの記憶域を必要とします。

FLOAT

FLOAT データ型は、小数部分を持つ数値または INTEGER データ型の容量を超える数値の格納に使用します。数は使用しているコンピュータの浮動小数点形式を使って表示されます。一般的には、記憶域に 4 バイトまたは 8 バイトを必要とします。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。

Oracle8*i* では、数値の内部形式が 10 進数であるため、浮動小数点の場合よりも大きい精度で数を表すことができます。

> **注意 :** SQL 文で FLOAT 値を比較する場合、FLOAT データ型では数値がバイナリ（10 進数ではない）で格納されるので SQL 関数 ROUND を使います。小数部分は正確に変換されません。

INTEGER

INTEGER データ型は、小数部分のない数値の格納に使用します。整数は、2 バイトまたは 4 バイトの符号付きの 2 進数です。ワード内のバイトの並びはプラットフォー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。列値が浮動小数点数の場合、Oracle8*i* は出力時に小数部分を切り捨てます。

LONG

LONG データ型は、固定長文字列の格納に使用します。LONG データ型は VARCHAR2 データ型と似ていますが、LONG 値の最大長は 2147483647 バイト（2 ギガバイト）である点で異なります。

LONG RAW

LONG RAW データ型は、固定長バイナリ・データまたは固定長バイト列の格納に使用します。LONG RAW 値の最大長は 2147483647 バイト（2 ギガバイト）です。

LONG RAW データは LONG データと似ていますが、Oracle8*i* 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。

LONG VARCHAR

LONG VARCHAR データ型は、可変長文字列の格納に使用します。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 バイトです。EXEC SQL VAR 文では、4 バイトの長さフィールドは含めないでください。

LONG VARRAW

LONG VARRAW データ型は、バイナリ・データまたはバイト列の格納に使用します。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。EXEC SQL VAR 文では、4 バイトの長さフィールドは含めないでください。

NUMBER

NUMBER データ型は、固定小数点数または浮動小数点数の格納に使用します。精度および位取りを指定できます。NUMBER 値の最大精度は 38、大きさの範囲は 1.0E-129 ～ 9.99E125 です。位取りの範囲は -84 ～ 127 です。

NUMBER 値は可変長形式で格納されます。指数バイトで始まり、その後に最大 20 バイトの仮数バイトが続きます。指数部のバイトの上位 1 ビットは符号ビットであり、正数のときに設定します。下位 7 ビットは指数を表します。指数は、100 を基底としてオフセット 65 を加えた数字です。

各仮数のバイトは、1 ～ 100 の範囲の 100 を基底とする数値です。正数の場合は数値に 1 を加えた数になります。負数の場合は 101 から引いた数になります。仮数バイトが 20 バイトある場合を除いて、102 が入ったバイトがデータ・バイトに追加されます。各仮数バイトは 2 桁の 10 進数を表すことができます。仮数は正規化され、先行ゼロは格納されません。仮数には最大 20 データ・バイトを使用できますが、正確さが保証されるのは 19 バイトだけです。この 19 バイトはそれぞれ 100 を基底とする数字を表し、最大精度は 38 桁です。

出力では、Oracle8*i* で内部的に表現される数字がホスト変数に含まれます。最大値を表現するには、出力ホスト変数に 21 バイトが必要です。数を表現するのに使われるバイトだけが戻されます。Oracle8*i* は、出力値に空白を埋め込んだり、出力値を NULL で終了させることはありません。戻された値の長さを知る必要がある場合は、かわりに VARNUM データ型を使います。

通常、このデータ型はほとんど使用されません。

RAW

RAW データ型は、固定長バイナリ・データまたは固定長バイト列の格納に使用します。ほとんどのプラットフォームでは、RAW 値の最大長は 65535 バイトです。

RAW データは CHAR データと似ていますが、Oracle8*i* では RAW データの意味は解釈されず、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・セットの変換は行われません。

ROWID

Oracle8 より前のリリースでは、表内の行の物理アドレスを 16 進数として格納するために ROWID データ型を使用していました。ROWID に行の物理アドレスを格納することによって、1 回のブロック・アクセスで効果的に行を検索できます。

Oracle8 では、論理的な ROWID が導入されました。索引構成表の行にはパーマネント物理アドレスがありません。論理的な ROWID には、物理的な ROWID と同じ構文を使ってアクセスします。そのために、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）を格納するために物理的な ROWID のサイズが拡張されました。

論理的な ROWID と物理的な ROWID の両方（および Oracle 表以外の ROWID）をサポートするために、ユニバーサル ROWID が定義されています。

VARCHAR2 ホスト変数を使用すると、ROWID を読取り可能な形式で格納できます。ROWID を選択またはフェッチして VARCHAR2 ホスト変数に入れると、Oracle8*i* はそのバイナリ値を 18 バイトの文字列に変換し、次の書式で戻します。

```
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 ROWID を考えます。

```
0000000E.000A.0007
```

これは、7 番目のデータベース・ファイルの 15 番目のブロックの 11 行目を示します。

通常は、ROWID をフェッチして VARCHAR2 ホスト変数に入れてから、UPDATE 文または DELETE 文の WHERE 句に含まれる 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。7-18 ページの「CURRENT OF 句の疑似実行」の例を参照してください。

> **注意 :** 完全な移植性を必要とする場合、あるいはアプリケーションで Transparent Gateway を介して Oracle 以外のデータベースと通信する場合は、VARCHAR2 ホスト変数の宣言時に最大長の 256（18 ではない）バイトを指定します。また、アプリケーションが Oracle Open Gateway を介して Oracle 以外のデータ・ソースと通信する場合も、最大長として 256 バイトを指定します。ホスト変数の内容については何も類推できませんが、SQL 文では正常に動作します。

STRING

STRING データ型は、VARCHAR2 データ型に似ていますが、STRING 値が常に NULL 文字で終了する点で異なります。

入力時 : Oracle8*i* は、指定された長さを使用して NULL 終了記号の走査を制限します。NULL 終了記号が見つからなければ、エラーが発生します。長さを指定しなければ、Oracle8*i* は最大長を、ほとんどのプラットフォーム上では 65535 と見なします。

STRING 値の最小長は 2 バイトです。最初の文字が NULL 終了記号で、指定された長さが 2 の場合、Oracle8*i* は、その列が NOT NULL と定義されていない限り、NULL を挿入します。すべて空白の値または NULL で終了している値は、そのまま格納されます。

出力時 : Oracle8*i* は、戻された最後の文字に NULL バイトを追加します。文字列の長さが指定された長さを超えている場合は、出力値を切り捨てて NULL バイトを追加します。

UNSIGNED

UNSIGNED データ型は、符号なし整数の格納に使用します。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並び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。列値が浮動小数点数の場合、Oracle8*i* は出力時に小数部分を切り捨てます。

VARCHAR

VARCHAR データ型は、可変長文字列の格納に使用します。VARCHAR 変数には、2 バイトの長さフィールドと、その後に続く 65533 バイトの文字列フィールドがあります。しかし、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。もっと長い文字列には、LONG VARCHAR データ型を使ってください。EXEC SQL VAR 文では、2 バイトの長さフィールドは含めないでください。

VARCHAR2

VARCHAR2 データ型は、可変長文字列の格納に使用します。ほとんどのプラットフォームでは、VARCHAR2 値の最大長は 65535 バイトです。

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場合、最大長は *n* 文字より少なくなります。

入力時 : Oracle8*i* は、入力ホスト変数について指定された数のバイトを読み込み、後続する空白を削除し、入力値をターゲット・データベースの列に格納します。注意が必要です。未初期化変数には NULL が含まれている場合があります。したがって、文字ホスト変数は、宣言された長さになるように常に空白が埋め込まれます。（COBOL PIC X(*n*) 変数の場合は、これが自動的に実行されます。）

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。すべて空白の入力値は、Oracle8*i* では NULL として扱われます。

文字値が有効な数値を表している場合は、Oracle8*i* 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。

出力時 : Oracle8*i* は、出力ホスト変数について指定された数のバイトを戻し、必要に応じて空白を埋め込み、出力値をターゲット・ホスト変数に割り当てます。NULL が戻された場合、ホスト変数は空白で埋められます。

出力値がホスト変数の宣言長より長い場合、Oracle8*i* 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。

Oracle8*i* 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、科学表記法が使われます。たとえば、列値 123456789 を選択して長さ 6 のホスト変数に入れると、Oracle8*i* はホスト変数に値 "1.2E08" を戻します。

VARNUM

VARNUM データ型は NUMBER データ型と似ていますが、VARNUM 変数の最初のバイトには値の長さが格納される点が異なります。

入力では、ホスト変数の最初のバイトを値の長さに設定しなければなりません。出力では、長さの後に Oracle8*i* で内部的に表現される数字がホスト変数に含まれます。最大値を表現するには、ホスト変数に 22 バイトが必要です。列値を選択して VARNUM ホスト変数に格納した後、先頭バイトをチェックすることによって値の長さを調べることができます。

VARRAW

VARRAW データ型は、可変長バイナリ・データまたは可変長バイト列の格納に使用します。VARRAW データ型は RAW データ型に似ていますが、VARRAW 変数の場合は 2 バイトの長さフィールドに <= 65533 バイトのデータ・フィールドが続きます。もっと長い文字列には、LONG VARRAW データ型を使用してください。EXEC SQL VAR 文では、2 バイトの長さフィールドは含めないでください。VARRAW 変数の長さを得るには、長さフィールドを参照するだけで済みます。

ホスト変数

ホスト変数は、ホスト・プログラムとサーバーで通信する際のキーとなります。一般的に、ホスト・プログラムはサーバーにデータを入力し、サーバーはホスト・プログラムにデータを出力します。サーバー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。

ホスト変数の宣言

ホスト変数は、Pro*COBOL でサポートされている COBOL データ型を使い COBOL の規則に従って宣言します。COBOL データ型には、ソース / ターゲット・データベース列との互換性が必要です。

表 4-6 に、サポートされている COBOL のデータ型を示します。

表 4-6 ホスト変数の宣言

変数の宣言	説明
PIC X...X	1 バイト文字から成る固定長文字列（1）
PIC X(*n*)	1 バイト文字から成る長さ *n* の文字列
PIC X...X VARYING	1 バイト文字から成る可変長文字列（1、2）
PIC X(*n*) VARYING	1 バイト文字から成る可変長文字列（最大長 *n*）（2）
PIC N...N PIC G...G	マルチバイト NCHAR 文字から成る固定長文字列（1、3）
PIC N(*n*) PIC G(n)	マルチバイト NCHAR 文字から成る *n* 文字の文字列（3）
PIC N...N VARYING PIC N(n) VARYING PIC G...G VARYING PIC G(n) VARYING	マルチバイト文字から成る可変長文字列（2、3） マルチバイト文字から成る可変長文字列（最大長 *n*）（2、3）
PIC S9...9 BINARY PIC S9(*n*) BINARY PIC S9...9 COMP PIC S9(*n*) COMP PIC S9...9 COMP-4 PIC S9(*n*) COMP-4	整数（4、5、7）
COMP-1 COMP-2	浮動小数点数（5）
PIC S9...9V9...9 COMP-3 PIC S9(*n*)V9(*n*) COMP-3 PIC S9...9V9...9 PACKED-DECIMAL PIC S9(*n*)V9(*n*) PACKED-DECIMAL	パック 10 進数（4、5）

表 4-6 ホスト変数の宣言 (続き)

PIC S9...9 COMP-5 PIC S9(*n*) COMP-5	バイトスワップ整数（4、5、6、7）
PIC S9...9V9...9 DISPLAY SIGN LEADING SEPARATE PIC S9(*n*)V9(*m*) DISPLAY SIGN LEADING SEPARATE	先行符号表示（9、12）
PIC S9...9V9...9 DISPLAY SIGN TRAILING SEPARATE PIC S9(*n*)V9(*m*) DISPLAY SIGN TRAILING SEPARATE	後続符号表示（9）
PIC 9...9 DISPLAY PIC 9(n)V9(m) DISPLAY	符号なし表示（10）
PIC S9...9V9...9 DISPLAY SIGN TRAILING PIC S9(n)V9(m) DISPLAY SIGN TRAILING	埋込み後続符号付き（10、11）
PIC S9...9V9...9 DISPLAY SIGN LEADING PIC S9(n)V9(m) DISPLAY SIGN LEADING	埋込み先行符号付き（10）
SQL-CURSOR	カーソル変数
SQL-CONTEXT	実行時コンテキスト
SQL-ROWID	ユニバーサル ROWID

注意 :

1. X...X および 9...9 は、X または 9 の個数（*n*）をそれぞれ表します。可変長文字列の場合、*n* は最大長です。
2. キーワード VARYING は、外部データ型 VARCHAR を文字列に割り当てます。詳細は、4-27 ページの「VARCHAR 変数の宣言」を参照してください。
3. Pro*COBOL のソース・ファイルで PIC N または PIC G データ型を使用する場合は、そのデータ型が COBOL コンパイラでサポートされていることを事前に確認してください。

4. 符号付きの数（PIC S...）だけが使用できます。ただし、浮動小数点数の場合は PIC 文字列は受け入れられません。
5. すべての COBOL コンパイラがこれらのデータ型をすべてサポートしているわけではありません。
6. COMP または COMP-5 では、小数部分を持つ数値は受け入れられません。また、位取りされた 2 進数はサポートされません。
7. *n* の最大値の範囲は 9 ～ 18 で、システムによって異なります。
8. COBOL 型の 1 次元の表もサポートされます。
9. DISPLAY と SIGN はどちらも省略できます。
10. DISPLAY は省略できます。
11. TRAILING を省略した場合、埋め込まれる符号の位置はオペレーティング・システムによって異なります。
12. LEADING は省略できます。

詳細は、表 4-7 の「互換性のある Oracle の内部データ型」を参照してください。

表 4-7 互換性のある Oracle の内部データ型

内部データ型	注意	COBOL データ型	説明
CHAR(*x*)	(1)	PIC X...X	文字列
VARCHAR2(*y*)	(1)	PIC X(*n*)	*n* 文字の文字列
		PIC {X(*n*) | X(*n*) VARYING}	可変長文字列
		PIC S9...9 COMP	整数
		PIC S9(*n*) COMP	
		PIC S9...9 BINARY	整数
		PIC S9(*n*) BINARY	
		PIC S9...9 COMP-5	整数
		PIC S9(*n*) COMP-5	
		COMP-1	浮動小数点数
		COMP-2	
		PIC S9...9V9...9 COMP-3	パック 10 進数
		PIC S9(*n*)V9(*n*) COMP-3	
		PIC S9...9V9...9 DISPLAY	表示
		PIC S9(*n*)V9(*n*) DISPLAY	

表 4-7 互換性のある Oracle の内部データ型 (続き)

内部データ型	注意	COBOL データ型	説明
NCHAR(u)	(2)	PIC {N...N | G...G}	各国文字文字列
NVARCHAR2(v)	(2)	PIC { N(n) | G(n)}	n 文字の各国文字文字列
BLOB		SQL-BLOB	バイナリ LOB
CLOB		SQL-CLOB	文字 LOB
NCLOB		SQL-NCLOB	各国文字 LOB
BFILE		SQL-BFILE	外部バイナリ・ファイル
NUMBER		PIC S9...9 COMP	整数
NUMBER (*p*,*s*)	(3)	PIC S9(*n*) COMP	
		PIC S9...9 BINARY	整数
		PIC S9(*n*) BINARY	
		PIC S9...9 COMP-5	整数
		PIC S9(*n*) COMP-5	
		COMP-1	浮動小数点数
		COMP-2	
		PIC S9...9V9...9 COMP-3	パック 10 進数
		PIC S9(*n*)V9(*n*) COMP-3	
		PIC S9...9V9...9 DISPLAY	表示
		PIC S9(*n*)V9(*n*) DISPLAY	
		PIC [X...X | N...N | G...G]	文字列 (4)
		PIC [X(*n*) | N(*n*) | G(n)]	*n* 文字の文字列 (4)
		PIC X...X VARYING	可変長文字列
		PIC X(*n*) VARYING	*n* バイトの可変長文字列
DATE	(5)	PIC X(*n*)	*n* バイトの文字列
LONG			
RAW	(1)	PIC X...X VARYING	*n* バイトの可変長文字列
LONG RAW			
ROWID	(6)	SQL-ROWID	ユニバーサル ROWID

注意:

1. <= *x* <= 2000 バイト、デフォルト 1。1 <= *y* <= 4000 バイト、デフォルト 1。
2. 1 <=u <= 2000 バイト、デフォルト 1。1 <=v <= 4000 バイト、デフォルト 1。
3. *p* の範囲は 2 ～ 38 で、*s* の範囲は -84 ～ 127 です。
4. 変換可能な文字（0 ～ 9、ピリオド（.）、+、-、E、e）だけで文字列が構成されている場合は、文字列を数値に変換できます。システムの NLS の設定によっては、小数点がピリオド（.）からカンマ（,）に変わる場合もあります。
5. 文字列型に変換された場合の DATE のデフォルトのサイズは、システムで有効になっている NCHAR の設定によって決まります。2 進値に変換された場合の長さは 7 バイトです。
6. 文字列型に変換すると、ROWID には 18 ～ 4000 バイトが必要になります。

宣言の例

Pro*COBOL プログラムで使用する複数のホスト変数を宣言する例を次に示します。

```
    ...
01  STR1  PIC X(3).
01  STR2  PIC X(3) VARYING.
01  NUM1  PIC S9(5) COMP.
01  NUM2  COMP-1.
01  NUM3  COMP-2.
    ...
```

次の例に示すように、簡単な COBOL 型の 1 次元表の宣言もできます。

```
    ...
01  XMP-TABLES.
    05  TAB1  PIC XXX OCCURS 3 TIMES.
    05  TAB2  PIC XXX VARYING OCCURS 3 TIMES.
    05  TAB3  PIC S999 COMP-3 OCCURS 3 TIMES.
    ...
```

初期化

ホスト変数（疑似型のホスト変数を除く）は、次の例に示すように、VALUE 句を使って初期化できます。

```
01  USERNAME    PIC X(10) VALUE "SCOTT".
01  MAX-SALARY  PIC S9(4) COMP VALUE 5000.
```

文字変数に割り当てられた文字列値がその変数の宣言長より短い場合は、文字列の右側に空白が埋め込まれます。文字変数に割り当てられた文字列値が宣言長より長い場合は、文字列は切り捨てられます。

疑似型の変数について VALUES 句を指定しても、すべて無視され、廃棄されます。（エラーや警告は発行されません。）

制限

アルファベット文字（PIC A）変数および編集済みデータ項目はホスト変数として使用できません。このため、ホスト変数について次の変数宣言はできません。

```
     ....
01   AMOUNT-OF-CHECK  PIC ****9.99.
01   FIRST-NAME       PIC A(10).
01   BIRTH-DATE       PIC 99/99/99.

     ....
```

ホスト変数の参照

ホスト変数は、SQL データ操作文で使用します。次の例に示すように、SQL 文ではホスト変数の前にコロン（:）を付ける必要がありますが、COBOL 文ではコロンを付けません。

```
WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
01   EMP-NAME    PIC X(10) VALUE SPACE.
01   SALARY      PIC S9(5)V99 COMP-3.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
PROCEDURE DIVISION.
     ...
     DISPLAY "Employee number? " WITH NO ADVANCING.
     ACCEPT EMP-NUMBER.
     EXEC SQL SELECT ENAME, SAL
         INTO :EMP-NAME, :SALARY FROM EMP
         WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
     COMPUTE BONUS = SALARY / 10.
     ...
```

次の例に示すように、表または列と同じ名前をホスト変数に指定できます。ただし、このようにすると混乱を招く恐れがあります。

```
WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   EMPNO  PIC S9(4) COMP VALUE ZERO.
01   ENAME  PIC X(10) VALUE SPACE.
01   COMM   PIC S9(5)V99 COMP-3.
```

```
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    EXEC SQL SELECT ENAME, COMM
        INTO :ENAME, :COMM FROM EMP
        WHERE EMPNO = :EMPNO
    END-EXEC.
```

ホスト変数としてのグループ項目

Pro*COBOL では、埋込み SQL 文内でグループ項目を使用できます。（1 つのレベルだけを含んでいる）基本項目を持つグループ項目は、ホスト変数として使用できます。ホスト・グループ項目（ホスト構造体とも呼びます）は、SELECT 文または FETCH 文の INTO 句、および INSERT 文の VALUES リストで参照できます。グループ項目をホスト変数として使う場合は、SQL 文ではグループ名だけを使います。次に例を示します。

```
01  DEPARTURE.
    05 HOUR    PIC X(2).
    05 MINUTE  PIC X(2).
```

このように宣言した場合、次の文は有効です。

```
    EXEC SQL SELECT DHOUR, DMINUTE
        INTO :DEPARTURE
        FROM SCHEDULE
        WHERE ...
```

グループ項目内のメンバーを宣言する順序は、対応する列を SQL 文に記述する順序と一致していなければなりません。また、INSERT 文で列のリストを省略する場合は、データベースの表の中の対応する列の順序と一致している必要があります。グループ項目をホスト変数として使用することは、そのグループ項目を基本項目で置き換えることを意味します。上の例では、:DEPARTURE が :DEPARTURE.HOUR、:DEPARTURE.MINUTE に置き換えられます。

ホスト変数として使用するグループ項目には、ホスト表を含めることができます。次の例では、表を含んだグループ項目を使用して、SCHEDULE 表に 3 つのエントリを INSERT しています。

```
01  DEPARTURE.
    05  HOUR    PIC X(2) OCCURS 3 TIMES.
    05  MINUTE  PIC X(2) OCCURS 3 TIMES.
...
    EXEC SQL INSERT INTO SCHEDULE (DHOUR, DMINUTE)
         VALUES (:DEPARTURE) END-EXEC.
```

VARCHAR=YES と指定した場合、Pro*COBOL では暗黙的な VARCHAR が認識されます。ネストされたグループ項目宣言が VARCHAR ホスト変数と似ていると、そのグループ項目

全体が VARYING 型の 1 つの基本項目のように扱われます。詳細は、14-40 ページの「VARCHAR」を参照してください。

グループ項目ではなく基本項目をホスト変数として参照する場合、基本項目名は次の構文を使って修飾できるため、一意でなくてもかまいません。

```
<group_item>.<elementary_item>
```

このネーミング規則は SQL 文専用です。これは COBOL の IN（または OF）句と似ています。次に例を示します。

```
MOVE MINUTE IN DEPARTURE TO MINUTE-OUT.
DISPLAY HOUR OF DEPARTURE.
```

COBOL の IN（または OF）句は、SQL 文では使用できません。混乱しないように、基本項目名を修飾してください。たとえば、次のとおりです。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  DEPARTURE.
     05  HOUR    PIC X(2).
     05  MINUTE  PIC X(2).
 01  ARRIVAL.
     05  HOUR    PIC X(2).
     05  MINUTE  PIC X(2).
     EXEC SQL END DECLARE SECTION END-EXEC.
  ...
     EXEC SQL SELECT DHR, DMIN INTO :DEPARTURE.HOUR, :DEPARTURE.MINUTE
         FROM TIMETABLE
         WHERE ...
```

制限

SQL 文では、列または表、その他のオブジェクトのかわりにホスト変数を使用することはできません。また、Oracle8*i* の予約語をホスト変数としては使用できません。予約語とキーワードの一覧は、付録 C の「予約語、キーワードおよび名前領域」を参照してください。

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 文内で使うたびに、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。

VALUES 句または SET 句中の標識変数を使って、入力ホスト変数に NULL 値を割り当てたり、INTO 句中の標識変数を使って、出力ホスト変数内の NULL 値（または文字列の切り捨てられた値）を検出できます。

標識変数の使用方法

標識変数に割り当てることのできる変数は、次のとおりです。

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。

-1	Oracle によってその列に NULL 値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。

-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	Oracle によって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	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。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）。

標識変数の宣言

標識変数は PIC S9(4) COMP として明示的に宣言する必要があります。また、予約語は標識変数としては使用できません。次の例では、COMM-IND という名前の標識変数を宣言しています。（名前は任意に指定できます）

```
WORKING-STORAGE SECTION.
...
01  EMP-NAME    PIC X(10) VALUE SPACE.
01  SALARY      PIC S9(5)V99 COMP-3.
01  COMMISSION  PIC S9(5)V99 COMP-3.
01  COMM-IND    PIC S9(4) COMP.
...
```

標識変数の参照

SQL 文では、標識変数は前にコロンを付け、対応するホスト変数の直後に記述しなければなりません。COBOL 文では、標識変数の前にコロンを付けたり、対応するホスト変数の直後に記述したりしてはいけません。次に例を示します。

```
EXEC SQL SELECT SAL, COMM
    INTO :SALARY, :COMMISSION:COMM-IND FROM EMP
    WHERE EMPNO = :EMP-NUMBER
END-EXEC.
IF COMM-IND = -1
    COMPUTE PAY = SALARY
ELSE
    COMPUTE PAY = SALARY + COMMISSION.
```

判読しやすくするため、それぞれの標識変数の前に INDICATOR という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、次の構文と等価です。

```
:<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。

制限

WHERE 句では、標識変数による NULL の検索はできません。たとえば、次の DELETE 文を実行するとエラーが発生します。

```
*   Set indicator variable.
    COMM-IND = -1
    EXEC SQL
        DELETE FROM EMP WHERE COMM = :COMMISSION:COMM-IND
    END-EXEC.
```

正しい構文は次のとおりです。

```
EXEC SQL
    DELETE FROM EMP WHERE COMM IS NULL
END-EXEC.
```

Oracle8*i* での制限

標識を持たないホスト変数に NULL を SELECT または FETCH すると、Oracle8*i* はエラー・メッセージを発行します。

コマンド行に UNSAFE_NULL=YES も指定すると、エラー・メッセージは発行されなくなります。詳細は、第 14 章の「プリコンパイラのオプション」を参照してください。

ANSI の要件

MODE=ORACLE の場合、切り捨てられた列値を 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、Oracle8*i* はエラー・メッセージを発行します。

ただし、MODE={ANSI | ANSI14 | ANSI13} の場合にはエラーは発生しません。標識変数の値は、第 5 章の「埋込み SQL」を参照してください。

マルチバイト NCHAR 変数の標識変数

マルチバイト NCHAR 文字変数の標識変数は、他のホスト変数の場合と同様に使用できます。ただし、正の値（SELECT または FETCH の結果が切り捨てられた）は、1 バイト文字ではなくマルチバイト文字単位の文字列の長さを表します。

ホスト・グループ項目の標識変数

ホスト・グループ項目に標識変数を使用するには、別のグループ項目を設定するか、ハーフワードの整変数で構成される表を使用します。前者の場合、そのグループ項目にはホスト・グループ項目内の NULL 値指定が可能な変数に対するそれぞれの標識変数を指定します。グループ項目内の個々の変数に標識変数を関連付ける必要はありませんが、標識を使用しようとする、NULL 値が可能なフィールドは、データ・グループ項目の先頭に置かなければなりません。次の標識グループ項目は、DEPARTURE グループ項目に使用できます。

```
01  DEPARTURE-IND.
    05  HOUR-IND   PIC S9(4) COMP.
    05  MINUTE-IND PIC S9(4) COMP.
```

標識表を使用する場合、ホスト・グループ項目内のメンバー数と同じ数の要素を持つ表を宣言する必要はありません。次の標識表は、DEPARTURE グループ項目に使用できます。

```
01 DEPARTURE-IND PIC S9(4) COMP OCCURS 2 TIMES.
```

SQL 文で標識グループ項目を参照する方法は、ホスト標識変数を参照する方法と同じです。

```
EXEC SQL SELECT DHOUR, DMINUTE
   INTO :DEPARTURE:DEPARTURE-IND
     FROM SCHEDULE
       WHERE ...
```

問合せが完了すると、選択された各コンポーネントが NULL 状態か NOT NULL 状態かの情報がホスト標識グループ項目に設定されます。標識ホスト変数に関する制限事項および ANSI 必要条件は、ホスト標識グループ項目にも適用されます。

VARCHAR 変数

COBOL の文字列データ型は固定長ですが、Pro*COBOL では VARCHAR という可変長の文字列疑似型を宣言できます。

VARCHAR 変数の宣言

VARCHAR ホスト変数は、次の例に示すように、キーワード VARYING を宣言に追加することによって定義します。

```
01  ENAME  PIC X(15) VARYING.
```

注意 : PIC N および PIC G は、VARYING を使った定義では使用できません。VARYING 変数での PIC N および PIC G の正しい使用方法は、4-28 ページの「暗黙的な VARCHAR グループ項目」を参照してください。

COBOL の VARYING 句は、添字および索引を増分するために、PERFORM 文および SEARCH 文で使用するものです。上記の例の Pro*COBOL の VARYING 句と混同しないでください。

VARCHAR は、Pro*COBOL の拡張データ型または宣言済みグループ項目と考えられます。たとえば、Pro*COBOL では、

```
01  ENAME  PIC X(15) VARYING.
```

という VARCHAR 宣言が、次のような長さフィールドと文字列フィールドをもつグループ項目に展開されます。

```
01  ENAME.
    05  ENAME-LEN  PIC S9(4) COMP.
    05  ENAME-ARR  PIC X(15).
```

長さフィールド（接尾辞 -LEN）には、文字列フィールド（接尾辞 -ARR）に格納されている値の現在の長さが入れられます。VARCHAR ホスト変数宣言での最大長は 1 ～ 65533 バイトの範囲内でなければなりません。

VARCHAR 変数の利点は、長さフィールドを明示的に設定および参照できることです。Pro*COBOL は、入力ホスト変数を使用して長さフィールドの値を読み取り、そこに指定された数だけ、文字列フィールドの文字を使用します。また、出力ホスト変数では、長さの値を文字列フィールドに格納された文字列の長さに設定します。

暗黙的な VARCHAR グループ項目

Pro*COBOL は、プリコンパイラ・オプション VARCHAR=YES がコマンド行で指定された場合、一部のグループ項目を暗黙的に VARCHAR ホスト変数として認識します。可変長シングルバイト文字型の場合は、次の構造を使用してください。（長さはシングルバイト文字単位で表します。）

```
<nn>  DATA-NAME-1.
      49  DATA-NAME-2 PIC S9(4) COMP.
      49  DATA-NAME-3 PIC X(<length>).
```

nn には 01 ～ 48 を指定しなければなりません。

可変長マルチバイト NCHAR 文字列の場合は、次の書式を使用します。（長さはダブルバイト文字単位で表します）

```
<nn> DATA-NAME-1.
     49 DATA-NAME-2  PIC  S9(4)  COMP.
     49 DATA-NAME-3  PIC  N(<length>).

<nn> DATA-NAME-1.
     49 DATA-NAME-2  PIC  S9(4)  COMP.
     49 DATA-NAME-3  PIC  G(<length>).
```

これらのグループ項目の構造内の基本項目は、Pro*COBOL に VARCHAR ホスト変数であると認識されるように、レベル 49 として宣言されなければなりません。

Pro*COBOL に VARCHAR グループ項目の拡張書式を認識させるには、VARCHAR オプションをコマンド行で VARCHAR=YES と指定する必要があります。VARCHAR=NO と指定した場合は、上記の書式と似た宣言はどれも通常のグループ項目として解釈されます。VARCHAR=YES と指定しても、グループ項目の宣言書式が拡張 VARCHAR 書式と似ているが同じではない場合には、その項目は VARCHAR グループ項目ではなく通常のグループ項目と解釈されます。たとえば、VARCHAR=YES と指定して、次のように記述したとします。

```
01  lastname
    48 lastname-len  PIC S9(4) USAGE COMP.
    48 lastname-text PIC X(15).
```

このグループ項目の要素にはレベル 49 ではなくレベル 48 が使用されているため、この項目は VARCHAR グループ項目ではなく通常のグループ項目と解釈されます。

Pro*COBOL VARCHAR オプションの詳細は、第 14 章の「プリコンパイラのオプション」を参照してください。

VARCHAR 変数の参照

SQL 文で VARCHAR 変数を参照する場合は、次の例のように、グループ名の前にコロンを付けたものを使用します。

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
01  PART-NUMBER  PIC X(5).
01  PART-DESC    PIC X(20) VARYING.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...

    EXEC SQL
        SELECT PDESC INTO :PART-DESC FROM PARTS
        WHERE PNUM = :PART-NUMBER
    END-EXEC.
```

問合せの実行後、PART-DESC-LEN には、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入っています。

COBOL の文では、次の例に示すように、グループ名または基本項目を使って VARCHAR 変数を参照できます。

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        ...
01  EMP-TABLES.
    05  EMP-NAME  OCCURS 50 TIMES  PIC X(15) VARYING.
            ...
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    PERFORM DISPLAY-NAME
        VARYING J FROM 1 BY 1 UNTIL J > NAME-COUNT.
    ...
DISPLAY-NAME.
    DISPLAY EMP-NAME-ARR OF EMP-NAME(J).
```

文字データの処理

この項では、Pro*COBOL が文字ホスト変数をどのように処理するかを説明します。文字ホスト変数には、2 種類のシングルバイト文字ホスト変数と 2 種類のマルチバイト NLS 文字ホスト変数があります。

- PIC X(*n*)（または PIC X...X）
- PIC X(*n*) VARYING（または PIC X...X VARYING）
- PIC N(*n*)（または PIC N...N）または PIC G(n)（または PIC G...G）

  注意 : マルチバイト NCHAR データ型を使う前に、使用している COBOL コンパイラが PIC N データ型、または PIC G データ型をサポートしていることを確認してください。

PIC X のデフォルト

PCI X のデフォルトのデータ型は CHARF です。（リリース 8.0 より前は VARCHAR2 でした。）下位互換性を保つために、プリコンパイラ・コマンド行オプション PICX が用意されています。PICX はコマンド行または構成ファイルにだけ入力できます。詳細は、14-34 ページの「PICX」を参照してください。

PICX オプションの効果

PICX オプションによって、文字列内のデータを Pro*COBOL がどのように扱うかが決定されます。PICX オプションを使うことにより、プログラムで ANSI 固定長文字列を使用したり、データベース・サーバーおよび Pro*COBOL の旧バージョンとの互換性を維持できます。

リリース 8.0 より前の Pro*COBOL と同じ結果を得るには、PICX=VARCHAR2（デフォルトではない）を使用する必要があります。あるいは、変数ごとに次の文を使用します。

```
EXEC SQL <varname> IS VARCHAR2 END-EXEC.
```

固定長文字変数

固定長文字変数は、PIC X(*n*)、PIC G(*n*) および PIC N(*n*) データ型を使って宣言します。これらの変数の型では、文字データはそのロールに基づいて、入力変数または出力変数として扱われます。

入力時

PICX=VARCHAR2 の場合、プログラム・インタフェースは値をデータベースに送る前に、後続する空白を削除します。固定長 CHAR 列に挿入した場合、Pro*COBOL はそのデータベース列の長さに達するまで、後続する空白を再度追加します。これに対し、可変長 VARCHAR2 列に挿入した場合は、空白は追加されません。

PICX=CHARF の場合は、後続する空白は削除されません。

入力値に余分な文字が後続していないことを確認してください。たとえば、NULL は削除されずにデータベースに挿入されます。値が PIC X(*n*) 変数に ACCEPT または MOVE されると、COBOL によってその変数の長さまで空白が追加されるため、これは通常は問題になりません。

この点については、次の例で示します。

```
 WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMPLOYEES.
     05  EMP-NAME     PIC X(10).
     05  DEPT-NUMBER  PIC S9(4) VALUE 20 COMP.
     05  EMP-NUMBER   PIC S9(9) VALUE 9999 COMP.
     05  JOB-NAME     PIC X(8).
             ...
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
     DISPLAY "Employee name? " WITH NO ADVANCING.
     ACCEPT EMP-NAME.
*    Assume that the name MILLER was entered
*    EMP-NAME contains "MILLER    " (4 trailing blanks)
     MOVE "SALES" TO JOB-NAME.
*    JOB-NAME now contains "SALES   " (3 trailing blanks)
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO, JOB)
         VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER, :JOB-NAME
     END-EXEC.
     ...
```

最後の例で、PICX=VARCHAR2 を指定してプリコンパイルした場合に、ターゲット・データベース列が VARCHAR2 であれば、プログラム・インタフェースによって入力時に後続する空白が削除され、6 文字の文字列 "MILLER" と 5 文字の文字列 "SALES" だけがデータベースに挿入されます。これに対し、ターゲット・データベース列が CHAR の場合には、列の幅に達するまで文字列に空白が埋め込まれます。

最後の例で、PICX=CHARF を指定してコンパイルした場合に、JOB 列が CHAR(10) として定義されていると、JOB 列に挿入される値は "SALES#####"（後続する空白は 5 個）になります。ただし、JOB 列が VARCHAR2(10) として定義されている場合、ホスト変数は PIC X(8) として宣言されているので、挿入される値は "SALES###"（後続する空白は 3 個）になります。このように、期待したとおりの結果にならない場合があるので、注意してください。

出力時

PICX オプションは、固定長文字変数に対する出力には影響しません。PIC X(*n*) 変数を出力ホスト変数として使用すると、Pro*COBOL によって空白が埋め込まれます。たとえば、プログラムがデータベースから文字列 "MILLER" をフェッチした場合、EMP-NAME 内の値は "MILLER####"（後続する空白は 4 個）になります。この文字列は、そのまま別の SQL 文への入力として使用できます。

可変長変数

VARCHAR 変数では、文字データはそのロールに基づいて、入力変数または出力変数として扱われます。

入力時

VARCHAR 変数を入力ホスト変数として使用する場合は、次の例に示すように、拡張 VARCHAR 宣言の長さフィールドおよび文字列フィールドに値を割り当てる必要があります。

```
    IF ENAME-IND = -1
        MOVE "NOT AVAILABLE" TO ENAME-ARR
        MOVE 13 TO ENAME-LEN.
```

文字列変数に空白を埋め込む必要はありません。SQL 操作では、Pro*COBOL は空白も含めて、長さフィールドで指定されたとおりの文字数を使用します。

マルチバイト NLS データのホスト入力変数の場合、後続する 2 バイトの空白は削除されません。長さコンポーネントは、バイト単位ではなく文字単位のデータの長さと見なされます。

出力時

VARCHAR 変数を出力ホスト変数として使用すると、Pro*COBOL によって長さフィールドが設定されます。次に例を示します。

```
WORKING-STORAGE SECTION.
    ...
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  EMPNO  PIC S9(4) COMP.
01  ENAME  PIC X(15) VARYING.
        ...
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    EXEC SQL
        SELECT ENAME INTO :ENAME FROM EMP
        WHERE EMPNO = :EMPNO
    END-EXEC.
```

```
IF ENAME-LEN = 0
    MOVE FALSE TO VALID-DATA.
```

VARCHAR 変数が固定長文字列より優れている点は、Pro*COBOL によって戻された値の長さをそのまま使用できることです。固定長文字列を使用した場合は、値の長さを得るには文字数をカウントする必要があります。

マルチバイト NCHAR データのホスト出力変数には何も埋め込まれません。バッファの長さは、バイト単位ではなく文字単位の長さに設定されます。

ユーザー指定の実行時コンテキスト

リリース 8.1 から、Pro*COBOL では、マルチスレッド Pro*C / C++ プログラムによって、COBOL サブプログラムの連絡節で定義した引数を使ってサブプログラムをコールできるようになりました。この引数の 1 つをコンテキストにできます。

注意 : マルチスレッド・アプリケーションは、Pro*COBOL ではサポートされません。

実行時コンテキスト（通常はコンテキストと呼ばれます）は、0 個以上の接続および 0 個以上のカーソル、それらのインライン・オプション（MODE、HOLD_CURSOR、RELEASE_CURSOR、SELECT_CURSOR など）、その他の状態情報が格納されたクライアント・メモリー内の領域を指すハンドルです。

コンテキストのホスト変数を定義するには、疑似型の SQL-CONTEXT を使います。たとえば、次のとおりです。

```
01  MY-CONTEXT  SQL-CONTEXT.
```

CONTEXT ALLOCATE プリコンパイラ・ディレクティブを使って、コンテキストにメモリーを割り当て、初期化します。

```
EXEC SQL CONTEXT ALLOCATE :context END-EXEC.
```

`context` は、実行時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 END-EXEC.
```

CONTEXT USE プリコンパイラ・ディレクティブを使って、その位置以後の埋込み SQL 文で使用するコンテキストを定義します。位置は、プログラム論理の流れの中での位置はなく、ソース・ファイル内での位置です。このコンテキスト値は、別の CONTEXT USE 文が検出されるまで使用されます。構文は次のとおりです。

```
EXEC SQL CONTEXT USE {:context | DEFAULT} END-EXEC.
```

キーワード DEFAULT は、別の CONTEXT USE ディレクティブが検出されるまで、後に続いて実行されるすべての埋込み SQL 文でデフォルト（またはグローバル）・コンテキストを使用することを示します。次に簡単な例を示します。

```
EXEC SQL CONTEXT USE :MY-CONTEXT END_EXEC.
```

コンテキスト変数 MY-CONTEXT がまだ定義および割り当てられていない場合は、エラーが戻されます。

CONTEXT FREE 文によって、必要なくなったコンテキストが使用していたメモリーを解放します。

```
EXEC SQL CONTEXT FREE :context END-EXEC.
```

次に例を示します。

```
EXEC SQL CONTEXT FREE :MY-CONTEXT END-EXEC.
```

ユニバーサル ROWID

データベース・サーバーで使用される表編成には、ヒープ表と索引構成表の 2 種類があります。

ヒープ表はデフォルトです。これは、Oracle8 よりも前の表で使用される編成です。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。物理的な ROWID の外部文字書式は、基数 64 でコード化した 18 バイトの文字列です。

索引構成表には、パーマネント識別子としての物理的な行アドレスがありません。そのような表には、論理的な ROWID が定義されます。索引構成表からの SELECT ROWID... 文を使用するときは、ROWID は、表の主キー、制御情報、およびオプションの物理的な自動設定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、Oracle リリース 8.1 で導入されました。ユニバーサル ROWID は、物理的な ROWID と論理的な ROWID のどちらにも使用できます。ユニバーサル ROWID を使って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。索引構成表へのアクセスでは、アプリケーションに影響を与えずに表編成を変更できます。ROWID に使用される列データ型は、UROWID(*length*) です。length は省略できます。

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 の詳細は、『Oracle8*i* 概要』を参照してください。

疑似型 SQL-ROWID を使用したユニバーサル ROWID の宣言の例を次に示します。

```
01  MY-ROWID SQL-ROWID.
```

ユニバーサル ROWID のメモリーは、ALLOCATE 文を使って割り当てます。

```
EXEC SQL ALLOCATE :MY-ROWID END-EXEC.
```

SQL DML 文で MY-ROWID を次のように使用します。

```
EXEC SQL SELECT ROWID INTO :MY-ROWID FROM MYTABLE WHERE ... END-EXEC.
...
```

```
     EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID END-EXEC.
...
```

この ROWID が必要なくなったら、FREE ディレクティブを使ってメモリーを解放します。

```
     EXEC SQL FREE :MY-ROWID END-EXEC.
```

また、幅 1 ～ 4000 の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うこともできます。文字ベースのユニバーサル ROWID は、下位互換性のためだけにヒープ表でサポートされます。ユニバーサル ROWID は可変長にできます。そのため、選択したときに切り捨てられることがあります。この変数の詳細については、『Oracle8*i* 概要』を参照してください。

文字変数の使用例を次に示します。

```
 01  MY-ROWID-CHAR PIC X(4000) VARYING.
...
     EXEC SQL ALLOCATE :MY-ROWID-CHAR;
     EXEC SQL SELECT ROWID INTO :MY-ROWID-CHAR FROM MYTABLE WHERE ... END-EXEC.
...
     EXEC SQL UPDATE MYTABLE SET ... WHERE ROWID = :MY-ROWID-CHAR END-EXEC.
...
     EXEC SQL FREE :MY-ROWID-CHAR;
```

サブプログラム SQLROWIDGET

Oracle サブプログラム SQLROWIDGET を使って、最後に挿入、更新または選択した行の ROWID を取り出すことができます。SQLROWIDGET では、あらかじめ宣言されたコンテキストおよび ROWID が引数として必要です。デフォルトのコンテキストを使用するには、最初に型 SQL-CONTEXT の変数に ZERO を移します。

注意 : ユニバーサル ROWID は、コールの前に宣言および割当てを行わなければなりません。コンテキストは、コールの前に宣言および割当てを行わなければなりません。コールの構文例を次に示します。

```
     CALL "SQLROWIDGET" USING context rowid.
```

引数の

context（IN）

は、疑似型 SQL-CONTEXT の実行時コンテキスト変数です。

rowid（OUT）

は、疑似型 SQL-ROWID のユニバーサル ROWID 変数です。コールの実行が正常に終了すると、この変数は有効なユニバーサル ROWID を指します。エラーが発生した場合は、MY-ROWID は定義されません。

このサブプログラムの使用方法の例を次に示します。

```
 01  MY-ROWID   SQL-ROWID.
 01  MY-CONTEXT SQL-CONTEXT.
 ...
     EXEC SQL ALLOCATE :MY-ROWID END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :MY-CONTEXT END-EXEC.
     EXEC SQL CONTEXT USE :MY-CONTEXT END-EXEC.
* INSERT, or UPDATE or DELETE Goes here:
 ...
    CALL "SQLROWIDGET" USING MY-CONTEXT MY-ROWID.
* MY-ROWID now has the universal rowid descriptor for the last row
 ...
    EXEC SQL CONTEXT FREE :MY-CONTEXT END-EXEC.
    EXEC SQL FREE :MY-ROWID END-EXEC.
 ...
```

各国語サポート

広く使用されている 7 ビットまたは 8 ビットの ASCII キャラクタ・セットおよび EBCDIC キャラクタ・セットが英数字を表すのに十分であっても、日本語などのアジアの言語の中には、数千という文字があるものもあります。このような言語では、個々の文字を表すのに 16 ビット以上必要です。Oracle8*i* は、このように異なる言語をどのように扱っているでしょうか。

Oracle8*i* 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、各国語サポート（NLS）が用意されています。これによって、異なる言語環境でアプリケーションを実行できます。NLS では、数値書式と日付書式はユーザー・セッション用に指定された言語変換に自動的に適応します。したがって、NLS により、世界中のユーザーがそれぞれの母国語で Oracle8*i* と対話できます。

さまざまな 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。デフォルトのパラメータ値は、初期化ファイルに設定できます。表 4-8 に、各 NLS パラメータで指定する内容を示します。

表 4-8 NLS パラメータ

NLS パラメータ	指定内容
NLS_LANGUAGE	言語によって異なる表記法
NLS_TERRITORY	地域によって異なる表記法
NLS_DATE_FORMAT	日付書式
NLS_DATE_LANGUAGE	日と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	ローカル通貨記号
NLS_ISO_CURRENCY	ISO 通貨記号
NLS_SORT	ソート順序

主なパラメータは NLS_LANGUAGE および NLS_TERRITORY です。NLS_LANGUAGE には言語によって異なる機能のデフォルト値を指定します。この機能には次のものが含まれます。

- サーバー・メッセージに使用する言語
- 日と月の名前に使用する言語
- ソート順序

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。この機能には次のものが含まれます。

- 日付書式
- 10 進数文字
- グループ・セパレータ
- ローカル通貨記号
- 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 NLS 機能の操作を制御できます。

```
NLS_LANG = <language>_<territory>.<character set>
```

ここで、*language* はユーザー・セッション用の NLS_LANGUAGE の値、*territory* は NLS_TERRITORY の値、*character set* は端末に使用されるコード体系を指します。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、端末で表示できるキャラクタ・セットに対応する数値コードの範囲です。また、これには端末との通信を制御するコードも入っています。

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 シェルを使う UNIX では、NLS_LANG を次のように定義できます。

```
setenv NLS_LANG French_France.WE8ISO8859P1
```

セッション中に NLS パラメータの値を変更する場合は、次のような ALTER SESSION 文を使用します。

```
ALTER SESSION SET <nls_parameter> = <value>
```

Pro*COBOL は、Oracle8*i* データベースに格納されている多言語のデータをアプリケーションで処理するための NLS 機能をすべてサポートしています。たとえば、他国語の文字変数を宣言し、INSTRB、LENGTHB、SUBSTRB など、文字列関数にその変数を渡すことができます。これらの関数には、それぞれ INSTR、LENGTH および SUBSTR 関数と共通の構文がありますが、文字単位ではなく、バイト単位を基礎とする操作になります。

関数 NLS_INITCAP、NLS_LOWER および NLS_UPPER を使って、ケース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 を使って、バイナリ順序ではなく言語上の順序に基づいて WHERE 句の比較を指定できます。NLS 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。NLS の詳細は、『Oracle8*i* アプリケーション開発者ガイド基礎編』を参照してください。

マルチバイト NLS キャラクタ・セット

Pro*COBOL は、次の機能を通してマルチバイト NLS キャラクタ・セットを拡張サポートしています。

- Pro*COBOL による埋込み SQL 文中のマルチバイト文字列の認識。
- COBOL の PIC N および PIC G データ型宣言句。これは、ホスト文字変数をマルチバイト文字として解釈するように Pro*COBOL に指示します。
- NLS_NCHAR 環境変数。PIC N または PIC G で使用されるクライアント側のキャラクタ・セットと等価にします。

可変長の各国文字セットはサポートされません。

NLS_LOCAL=YES の場合の制限事項

プリコンパイラ・オプション NLS_LOCAL=YES と指定した場合、NLS マルチバイト・データ型の空白埋込みと空白削除はランタイム・ライブラリ（SQLLIB）によって実行されます。

NLS_LOCAL=YES の場合、PL/SQL ブロック内では、マルチバイト NCHAR 機能はサポートされません。これらの機能は、N 付き引用符で表された文字リテラルと固定長の文字変数を組み込みます。

そのため、次の制限が適用されます。

表は使用できません。PIC N または PIC G データ型を使って宣言するホスト変数として、表を使用できません。

奇数バイト幅はありません。マルチバイト NCHAR 文字を格納するときには、Oracle8*i* の CHAR 列は使わないでください。奇数バイト数のデータがシングルバイト列からマルチバイトの NCHAR ホスト変数に FETCH されると、ランタイム・エラーが生成されます。

ホスト変数の同値化は行えません。EXEC SQL VAR 文を使ってマルチバイト NCHAR 文字変数を同値化することはできません。

動的 SQL は使用できません。Pro*COBOL では、NCHAR マルチバイト文字列ホスト変数に動的 SQL を使用できません。

マルチバイト NLS データが格納されている列に対しては、関数を使用しないでください。

埋込み SQL 内の文字列

埋込み SQL 文内のマルチバイト NLS 文字列は、文字 *N* と、その後に続く引用符（'）で囲まれた文字列で構成されます。

たとえば、次のように指定します。

```
EXEC SQL
    SELECT EMPNO INTO :EMP-NUM FROM EMP
    WHERE ENAME=N'<NLS_string>'
END-EXEC.
```

埋込み DDL

プリコンパイラ・オプションが NLS_LOCAL=YES と設定されている場合、NCHAR データを格納する列は埋込みデータ定義言語（DDL）文では使用できません。この制限はプリコンパイル時には適用されないため、NCHAR などの拡張された列型を埋込み DDL 文で使用すると、プリコンパイル・エラーではなく実行エラーになります。

上記のオプションの詳細は、第 14 章の「プリコンパイラのオプション」内の各項目を参照してください。

空白埋込み

Pro*COBOL 文字変数をマルチバイト NLS 変数として定義すると、その変数の外部データ型に応じて、次の空白埋込みおよび空白削除の規則が適用されます。詳細は 4-30 ページの「文字データの処理」を参照してください。

CHARF。入力データから後続の 2 バイトの空白が削除されます。ただし、文字列がマルチバイトの空白だけで構成されている場合は、標識としてマルチバイトの空白が 1 つバッファに残されます。

出力ホスト変数には、マルチバイトの空白が埋め込まれます。

VARCHAR。入力時に、ホスト変数からは後続する 2 バイトの空白が削除されません。長さコンポーネントは、バイト単位ではなく文字単位のデータの長さと見なされます。

出力では、ホスト変数には空白は埋め込まれません。バッファの長さは、バイト単位ではなく文字単位のデータの長さに設定されます。

STRING / LONG VARCHAR。これらのホスト変数は、NLS データについてはサポートされていません。これらのホスト変数を指定するには動的 SQL またはデータ型の同値化を使用する必要がありますが、NLS データについてはそのどちらもサポートされていないためです。

標識変数

マルチバイト NLS 文字変数でも、他の変数の場合と同様に標識変数を使用できます。ただし、列の長さの値はバイト数ではなく文字数で表されます。可能な値の一覧は、5-3 ページの「標識変数の使用方法」を参照してください。

データ型の変換

プリコンパイル時に、各ホスト変数に外部データ型が割り当てられます。たとえば、Pro*COBOL は PIC S9(*n*) COMP 型のホスト変数に INTEGER 外部データ型を割り当てます。SQL 文で使うすべてのホスト変数のデータ型コードは、実行時に Oracle8*i* に渡されます。Oracle8*i* は、コードを使って内部データ型と外部データ型を変換します。

Oracle8*i* は、SELECT した列値を出力ホスト変数に割り当てる前に、必ず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します。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" は PIC S9(4) COMP 値に変換できます。ただし、CHAR 値 "65543"（大きすぎる数）や "10F"（10 進数でない数）を PIC S9(4) COMP 値に変換することはできません。同様に、アルファベット文字を含んでいる PIC X(*n*) 値は NUMBER 値に変換できません。

ホスト変数のデータ型は、データベース列のデータ型と互換性がなければなりません。必ず、変換可能な値を指定してください。たとえば、文字列値 "YESTERDAY" を DATE 列値に変換しようとすると、エラーが発生します。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 2 バイトの整数に変換できます。しかし、CHAR 値 "65543"（大きすぎる数）や "10F"（10 進数でない数）を 2 バイトの整数に変換することはできません。同様に、アルファベット文字を含んでいる文字列値は NUMBER 値に変換できません。

数値の変換は、Oracle8*i* 初期化ファイルの各国語サポート（NLS）パラメータで指定された規則に従って行われます。たとえば、ピリオド（.）ではなくカンマ（,）を小数点として認識するようにシステムが構成されている場合があります。NLS の詳細は、『Oracle8*i* アプリケーション開発者ガイド基礎編』を参照してください。

次の表は、サポートされている内部データ型と外部データ型の変換を示します。

表 4-9 内部データ型と外部データ型の変換

	内部							
外部	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHAR	I/O	I/O (2)	I/O	I (3)	I/O	I/O (3)	I/O (1)	I/O
CHARF	I/O	I/O (2)	I/O	I (3)	I/O	I/O (3)	I/O (1)	I/O
CHARZ	I/O	I/O (2)	I/O	I (3)	I/O	I/O (3)	I/O (1)	I/O
DATE	I/O	I/O	I					I/O
DECIMAL	I/O (4)		I		I/O			I/O (4)
DISPLAY	I/O (4)		I		I/O			I/O (4)
FLOAT	I/O (4)		I		I/O			I/O (4)
INTEGER	I/O (4)		I		I/O			I/O (4)
LONG	I/O	I/O (2)	I/O	I (3、5)	I/O	I/O (3)	I/O (1)	I/O
LONG RAW	O (6)		I (5、6)	I/O		I/O		O (6)
LONG VARCHAR	I/O	I/O (2)	I/O	I (3、5)	I/O	I/O (3)	I/O (1)	I/O
LONG VARRAW	I/O (6)		I (5、6)	I/O		I/O		I/O (6)
NUMBER	I/O (4)		I		I/O			I/O (4)
RAW	I/O (6)		I (5、6)	I/O		I/O		I/O (6)
ROWID	I		I				I/O	I
STRING	I/O	I/O (2)	I/O	I (3.5)	I/O	I/O (3)	I/O (1)	I/O
UNSIGNED	I/O (4)		I		I/O			I/O (4)
VARCHAR	I/O	I/O (2)	I/O	I (3、5)	I/O	I/O (3)		I/O

表 4-9 内部データ型と外部データ型の変換 (続き)

VARCHAR2	I/O	I/O（2）	I/O	I（3）	I/O	I/O（3）	I/O（1）	I/O
VARNUM	I/O（4）		I		I/O			I/O（4）
VARRAW	I/O（6）		I（5、6）	I/O		I/O		I/O（6）

注意 :

1. 入力時には、ホスト文字列を Oracle'BBBBBBBB.RRRR.FFFF' の形式にしなければなりません。
2. 出力時には、列値は同じ形式で戻されます。
3. 2. 入力時には、ホスト文字列をデフォルトの DATE 文字形式にしなければなりません。
4. 出力時には、列値は同じ形式で戻されます。
5. 3. 入力時には、ホスト文字列を 16 進数形式にしなければなりません。
6. 出力時には、列値は同じ形式で戻されます。
7. 4. 出力時には、列値は有効な数値を表していなければなりません。
8. 5. 入力時には、長さが 2000 以下でなければなりません。
9. 6. 入力時、列値は 16 進数形式で格納されます。
10. 出力時、列値は 16 進数形式でなければなりません。
11. 7. 入力時には、ホスト文字列はテキスト形式の有効な OS ラベルでなければなりません。
12. 出力時には、列値は同じ形式で戻されます。
13. 8. 入力時には、ホスト文字列はロー形式の有効な OS ラベルでなければなりません。
14. 出力時には、列値は同じ形式で戻されます。

凡例 :

I = 入力のみ

O = 出力のみ

I/O = 入出力

DATE 文字列書式の明示的な制御

Oracle8*i* では、DATE 列値を選択して文字ホスト変数に格納する場合、内部バイナリ値を外部文字値に変換しなければなりません。このため、デフォルトの日付書式で文字列を戻す SQL 関数 TO_CHAR が暗黙的にコールされます。デフォルトの日付書式は、Oracle8*i* の初期化パラメータ NLS_DATE_FORMAT で設定します。時刻やユリウス暦の日付などのその他の情報を得るには、書式マスクを指定して明示的に TO_CHAR をコールする必要があります。

文字ホスト値を DATE 列に挿入する際にも変換が必要です。Oracle8*i* は、デフォルトの日付書式を期待する SQL 関数 TO_DATE を暗黙的にコールします。その他の書式の日付を挿入する場合は、書式マスクを指定して明示的に TO_DATE をコールする必要があります。

Pro*COBOL には、他のバージョンの SQL との互換性を保証するために、次のような日付文字列を指定するためのプリコンパイラ・オプションが用意されています。

DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | '*fmt*'(default LOCAL)}

DATE_FORMAT オプションは、コマンド行または構成ファイル内で指定する必要があります。指定できる日付文字列を次の表に示します。

表 4-10 日付文字列用の書式

書式名	略称	日付書式
国際標準化機構規格	ISO	yyyy-mm-dd
USA 標準	USA	mm/dd/yyyy
ヨーロッパ標準	EUR	dd.mm.yyyy
日本工業規格	JIS	yyyy-mm-dd
導入時定義	LOCAL	導入時に定義した任意の書式

'*fmt*' は、'Month dd, yyyy などの日付書式モデルです。日付書式モデル要素の一覧は、『Oracle8*i* SQL リファレンス』を参照してください。

注意 : リンクする単位のうち別個にコンパイルしたものには、すべて同じ DATE_FORMAT 値を使う必要があります。

データ型の同値化

データ型を同値化することによって、Oracle8*i* による入力データの解釈方法および出力データのフォーマット方法を制御できます。変数単位で、サポートされる COBOL のデータ型を外部データ型に同値化できます。

データ型を同値化する理由

データ型の同値化には、いくつかの利点があります。たとえば、COBOL プログラムで NULL 終了ホスト文字列を使用するとします。この場合、PIC X ホスト変数を宣言し、それを、常に NULL で終了する外部データ型 STRING に同値化できます。

データを解釈せずに格納する場合にも、データ型の同値化を使用できます。たとえば、整数ホスト配列を LONG RAW データベース列に格納する場合、ホスト配列を外部データ型 LONG RAW に同値化できます。

また、デフォルトのデータ型変換を変更する場合に、データ型の同値化を使用できます。初期化ファイルの NLS パラメータで特に指定されていない限り、DATE 列値を選択して文字ホスト変数に入れると、Oracle8*i* は次のような書式の 9 バイトの文字列を戻します。

DD-MON-YY
文字ホスト変数を DATE 外部データ型に同値化すると、Oracle8*i* は内部形式の 7 バイトの値を戻します。

ホスト変数の同値化

デフォルトでは、Pro*COBOL はすべてのホスト変数に特定の外部データ型を割り当てます。デフォルトの割当ては、ホスト変数を外部データ型に同値化することによって変更できます。これをホスト変数の同値化といいます。

VAR 埋込み SQL 文の構文は、次のとおりです。

```
EXEC SQL
    VAR <host_variable> IS <datatype> [CONVBUFSZ [IS] (<size>)]
END-EXEC
```

または

```
EXEC SQL VAR <host_variable> [CONVBUFSZ [IS] (<size>)] END-EXEC
```

この場合、<datatype> は次のとおりです。

```
<SQL datatype> [ ( {<length> | <precision>, <scale> } ) ]
```

この 2 つの句のどちらか 1 つまたは両方を必ず指定しなければなりません。

パラメータは次のとおりです。

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）。

外部データ型 VARCHAR と VARRAW には、2 バイトの長さフィールドの後に *n* バイトのデータ・フィールドがあります。*n* の範囲は 1 ～ 65533 です。したがって、*type_name* が VARCHAR または VARRAW の場合は、*host_variable* に最低 3 バイトが必要です。

LONG VARCHAR と LONG VARRAW 外部データ型には、4 バイトの長さフィールドの後に *n* バイトのデータ・フィールドがあります。*n* の範囲は 1 ～ 2147483643 です。したがって、*type_name* が LONG VARCHAR または LONG VARRAW の場合は、*host_variable* に最低 5 バイトが必要です。

SQL datatype
RAW、STRING などの有効な外部データ型の名前。

length	有効な長さをバイト数で指定する整数リテラル。長さの値には、外部データ型を収容できるサイズを指定する必要があります。 *type_name* が DECIMAL または DISPLAY の場合は、*length* のかわりに *precision* および *scale* を指定しなければなりません。*type_name* が VARNUM、ROWID または DATE の場合、*length* は事前定義されているため指定できません。他の外部データ型では、*length* はオプションです。デフォルトは *host_variable* の長さです。 *length* を指定するとき、*type_name* が VARCHAR、VARRAW、LONG VARCHAR または LONG VARRAW の場合には、データ・フィールドの最大長を指定してください。Pro*COBOL は、長さフィールドの分を加算します。*type_name* が LONG VARCHAR または LONG VARRAW で、データ・フィールドが 65533 バイトを超える場合は、*length* フィールドに "-1" を入れてください。
precision および *scale*	有効桁数を示す整数リテラルと、丸めが発生する地点を示す整数リテラル。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。 *precision* に 1 ～ 99、*scale* に -84 ～ 99 を指定できます。ただし、データベース列の精度と位取りの最大値は、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。 *precision* および *scale* は、*type_name* が DECIMAL または DISPLAY の場合にだけ指定してください。
size	指定した host_variable から別のキャラクタ・セットへの変換に使用するバッファのサイズ（バイト数）を示す整数。

4-47 ページの表 4-11 に、各外部データ型に使用するパラメータを示します。

CONVBUFSZ 句の詳細は、4-46 ページの「VAR 文の CONVBUFSZ 句」を参照してください。

NCHAR ホスト変数（PIC G または PIC N 句を含む変数）には、EXEC SQL VAR は使用できません。

DECLARE_SECTION=TRUE の場合は宣言文が必要であり、また、宣言文に EXEC SQL VAR 文を記述する必要があります。

この文の構文図は、F-86 ページの「VAR（Oracle 埋込み SQL ディレクティブ）」を参照してください。

ext_type_name が FLOAT の場合は *length* を指定してください。*ext_type_name* が DECIMAL の場合は、*length* ではなく、*precision* および *scale* を指定してください。

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、Oracle8*i* にデータを格納したいが解釈はさせたくない場合に、ホスト変数の同値化を利用できます。4 バイトの整数から成るホスト表を RAW データベース列に格納するとします。この場合は、次に示すように、ホスト表を RAW 外部データ型に同値化します。

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-TABLES.
     05  EMP-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
             ...
*    Reset default datatype (INTEGER) to RAW.
     EXEC SQL VAR EMP-NUMBER IS RAW (200) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

ホスト表では、指定する長さは、その表の保持に必要なバッファ・サイズと一致していなければなりません。最後の例では長さに 200 を指定しています。これは、4 バイトの整数を 50 個格納するのに必要なバッファ・サイズです。

また、使用するグループ項目を LONG VARCHAR として宣言できます。

```
 01  MY-LONG-VARCHAR.
     05 UC-LEN PIC S9(9) COMP.
     05 UC-ARR PIC X(6000).
     EXEC SQL VAR MY-LONG-VARCHAR IS LONG VARCHAR(6000).
```

VAR 文の CONVBUFSZ 句

EXEC SQL VAR 文には、オプションの CONVBUFSZ 句を使用できます。CONVBUFSZ 句には、指定したホスト変数のキャラクタ・セット間での変換に使用する、ランタイム・ライブラリ内のバッファのサイズ（バイト数）を指定します。

CONVBUFSZ 句を指定していないと、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生成されることがときどきあります。データベースでは、LONG 列を 1 つしか指定できません。複数の LONG 値が指定されると、エラーとなります。

このようなエラーが発生しないように、LONG サイズ未満の長さを指定します。キャラクタ・セット変換の結果が CONVBUFSZ で指定された長さより長い値になると、Pro*COBOL はエラーを戻します。

例

EMP 表から従業員の名前を選択して、NULL 終了文字列を要求する C 言語のルーチンに渡したいとします。これらの名前に、明示的に NULL 終了記号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だけで済みます。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
```

```
01   EMP-NAME  PIC X(11).
     EXEC SQL VAR EMP-NAME IS STRING (11) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

ENAME 列の幅は 10 文字のため、NULL 終了記号を含めるために新しい EMP-NAME には 11 文字を割り当てます。（length のデフォルト値はホスト変数の長さのため、*length* の指定は任意です。）ENAME から値を選択して EMP-NAME に入れると、その値は Oracle8*i* によって NULL 終了にされます。

表 4-11 ホスト変数を同値化するためのパラメータ

外部データ型	長さ	精度	位取り	デフォルトの長さ
CHAR	オプション	該当なし	該当なし	変数の宣言長
CHARZ	オプション	該当なし	該当なし	変数の宣言長
DATE	該当なし	該当なし	該当なし	7 バイト
DECIMAL	該当なし	必須	必須	なし
DISPLAY	該当なし	必須	必須	なし
DISPLAY TRAILING	該当なし	必須	必須	なし
UNSIGNED DISPLAY	該当なし	必須	必須	なし
OVERPUNCH TRAILING	該当なし	必須	必須	なし
OVERPUNCH LEADING	該当なし	必須	必須	なし
FLOAT	オプション（4 または 8）	該当なし	該当なし	変数の宣言長
INTEGER	オプション（1、2 または 4）	該当なし	該当なし	変数の宣言長
LONG	オプション	該当なし	該当なし	変数の宣言長
LONG RAW	オプション	該当なし	該当なし	変数の宣言長
LONG VARCHAR	必須（注意 1）	該当なし	該当なし	なし
LONG VARRAW	必須（注意 1）	該当なし	該当なし	なし
NUMBER	該当なし	該当なし	該当なし	なし
STRING	オプション	該当なし	該当なし	変数の宣言長
RAW	オプション	該当なし	該当なし	変数の宣言長

表 4-11 ホスト変数を同値化するためのパラメータ (続き)

外部データ型	長さ	精度	位取り	デフォルトの長さ
ROWID	該当なし	該当なし	該当なし	18 バイト（注意 2）
UNSIGNED	オプション（1、2 または 4）	該当なし	該当なし	変数の宣言長
VARCHAR	必須	該当なし	該当なし	なし
VARCHAR2	オプション	該当なし	該当なし	変数の宣言長
VARNUM	該当なし	該当なし	該当なし	22 バイト
VARRAW	オプション	該当なし	該当なし	なし

1. データ・フィールドが 65533 バイトを超える場合は、-1 を渡します。
2. 一般的な値ですが、デフォルトはポートにより異なります。

CHARF データ型指定子の使用方法

VAR 文でデータ型指定子 CHARF を使用すると、COBOL のデータ型を固定長の ANSI データ型 CHAR に同値化できます。

PICX=CHARF の場合は、VAR 文でデータ型 CHAR を指定すると、ホスト言語のデータ型は固定長の ANSI データ型（Oracle8*i* の外部データ型コード 96）に同値化されます。PICX=VARCHAR2 の場合は、ホスト言語のデータ型は可変長のデータ型 VARCHAR2（コード 1）に同値化されます。

ホスト言語のデータ型をいつでも固定長の ANSI データ型 CHAR に同値化できます。これには、VAR 文でデータ型 CHARF を指定します。CHARF を使用すると、PICX=VARCHAR2 に設定されている場合でも、ホスト言語のデータ型は固定長の ANSI データ型 CHAR に同値化されます。

ガイドライン

VARNUM 値および DATE 値は、必ず Oracle8*i* の内部形式で入力してください。Oracle8*i* は、VARNUM 値および DATE 値の出力には内部形式を使用します。

列値を選択して VARNUM ホスト変数に格納した後、先頭バイトをチェックすることによって値の長さを調べることができます。表 4-1 に、戻された VARNUM 値の例を示します。

表 4-12 VARNUM の例

	VARNUM 値			
10 進数	**長さバイト**	**指数バイト**	**仮数バイト**	**終了文字バイト**
0	1	128	該当なし	該当なし
5	2	193	6	該当なし
-5	3	62	96	102
2767	3	194	28、68	該当なし
-2767	4	61	74、34	102
100000	2	195	11	該当なし
1234567	5	196	2、24、46、68	該当なし

DATE 値の変換は、4-42 ページの「DATE 文字列書式の明示的な制御」を参照してください。

ニーズに合った Oracle8*i* の外部データ型がない場合は、VARCHAR2 ベースまたは RAW ベースの外部データ型を使用してください。

RAW 値および LONG RAW 値

Oracle8*i* では、RAW 列値または LONG RAW 列値を選択して文字ホスト変数に格納する場合、内部バイナリ値を外部文字値に変換しなければなりません。この場合、Oracle8*i* は RAW データまたは LONG RAW データの各バイナリ・バイトを文字のペアとして戻します。個々の文字は、ニブル（1/2 バイト）の等価の 16 進値を表します。たとえば、バイナリ・バイト 11111111 は文字のペア "FF" として戻されます。SQL 関数 RAWTOHEX はこれと同じ変換を実行します。

文字ホスト値を RAW 列または LONG RAW 列に挿入する際にも変換が必要です。ホスト変数内の文字のペアはそれぞれ、バイナリ・バイトの等価の 16 進値を表していなければなりません。文字がニブルの 16 進値を表していないと、Oracle8*i* はエラー・メッセージを発行します。

データ型の変換の詳細は、4-51 ページの「サンプル・プログラム 4: データ型の同値化」を参照してください。

表 4-13 に、外部データ型と COBOL データ型のデフォルト割当てを示します。

表 4-13 ホスト変数の同値化

COBOL データ型	外部データ型	コード
PIC X...X PIC X(*n*)	CHARF	96
PIC X...X VARYING PIC X(*n*) VARYING	VARCHAR	9
PIC S9...9 COMP PIC S9(*n*) COMP PIC S9...9 COMP-5 PIC S9(*n*) COMP-5 PIC S9...9 COMP-4 PIC S9(*n*) COMP-4 PIC S9...9 BINARY PIC S9(*n*) BINARY	INTEGER	3
COMP-1 COMP-2	FLOAT	4
PIC S9...9V9...9 COMP-3 PIC S9(*n*)V9(*n*) COMP-3 PIC S9...9V9...9 PACKED-DECIMAL PIC S9(*n*)V9(*n*) PACKED-DECIMAL	DECIMAL	7
PIC 9(n) COMP PIC 9...9 COMP	UNSIGNED	68
PIC S9...9V9...9 LEADING SEPARATE PIC S9(*n*)V9(*n*) LEADING SEPARATE	DISPLAY	91
PIC 9(n)V9(9) PIC 9...9V9...9	UNSIGNED DISPLAY	153
PIC S9...9V9...9 TRAILING PIC S9(n)V9(n) TRAILING	OVERPUNCH TRAILING	154
PIC S9...9V9...9 LEADING PIC S9(n)V9(n) LEADING	OVERPUNCH LEADING	172
PIC S9...9V9...9 TRAILING SEPARATE PIC S9(n)V9(n) TRAILING SEPARATE	DISPLAY TRAILING	152

サンプル・プログラム 4: データ型の同値化

次のプログラムは、Oracle に接続した後、SCOTT アカウントに IMAGE という名前のデータベースの表を作成し、この表への従業員番号のビットマップ・イメージの挿入をシミュレートします。データ型の同値化により、このプログラムは Oracle 外部データ型 LONG RAW を使ってビットマップ・イメージを表現できます。後でユーザーが従業員番号を入力すると、その番号のビットマップが IMAGE 表から選択され、端末の画面に疑似表示されます。

```
*****************************************************************
* Sample Program 4:  Datatype Equivalencing                     *
*                                                               *
* This program simulates the storage and retrieval of bitmap    *
* images into table IMAGE, which is created in the SCOTT        *
* account after logging on to ORACLE.  Datatype equivalencing   *
* allows an ORACLE external type of LONG RAW to be specified    *
* for the programs representation of the images.                *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. DTY-EQUIV.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME          PIC X(10) VARYING.
 01  PASSWD            PIC X(10) VARYING.
 01  EMP-REC-VARS.
     05  EMP-NUMBER    PIC S9(4) COMP.
     05  EMP-NAME      PIC X(10) VARYING.
     05  SALARY        PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05  COMMISSION    PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05  COMM-IND      PIC S9(4) COMP.

     EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
     EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.

 01  BUFFER-VAR.
     05  BUFFER        PIC X(8192).
     EXEC SQL VAR BUFFER IS LONG RAW END-EXEC.

 01  INEMPNO           PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.
```

```
01  DISPLAY-VARIABLES.
    05  D-EMP-NAME    PIC X(10).
    05  D-SALARY      PIC $Z(4)9.99.
    05  D-COMMISSION  PIC $Z(4)9.99.
    05  D-INEMPNO     PIC 9(4).
01  REPLY             PIC X(10).
01  INDX              PIC S9(9) COMP.
01  PRT-QUOT          PIC S9(9) COMP.
01  PRT-MOD           PIC S9(9) COMP.

PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL WHENEVER SQLERROR
        DO PERFORM SQL-ERROR END-EXEC.

    PERFORM LOGON.
    DISPLAY "OK TO DROP THE IMAGE TABLE? (Y/N)  "
        WITH NO ADVANCING.

    ACCEPT REPLY.

    IF (REPLY NOT = "Y") AND (REPLY NOT = "y")
        GO TO SIGN-OFF-EXIT.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL DROP TABLE IMAGE END-EXEC.
    DISPLAY " ".
    IF (SQLCODE = 0) DISPLAY
        "TABLE IMAGE DROPPED - CREATING NEW TABLE."
    ELSE IF (SQLCODE = -942) DISPLAY
        "TABLE IMAGE DOES NOT EXIST - CREATING NEW TABLE."
    ELSE PERFORM SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR
        DO PERFORM SQL-ERROR END-EXEC.
    EXEC SQL CREATE TABLE IMAGE
         (EMPNO NUMBER(4) NOT NULL, BITMAP LONG RAW)
    END-EXEC.
    EXEC SQL DECLARE EMPCUR CURSOR FOR
        SELECT EMPNO, ENAME FROM EMP
    END-EXEC.
    EXEC SQL OPEN EMPCUR END-EXEC.
    DISPLAY " ".
    DISPLAY
      "INSERTING BITMAPS INTO IMAGE FOR ALL EMPLOYEES ...".
    DISPLAY " ".
```

```
INSERT-LOOP.
    EXEC SQL WHENEVER NOT FOUND GOTO NOT-FOUND END-EXEC.
    EXEC SQL FETCH EMPCUR
        INTO :EMP-NUMBER, :EMP-NAME
    END-EXEC.
    MOVE EMP-NAME-ARR TO D-EMP-NAME.
    DISPLAY "EMPLOYEE ", D-EMP-NAME WITH NO ADVANCING.
    PERFORM GET-IMAGE.
    EXEC SQL INSERT INTO IMAGE
        VALUES (:EMP-NUMBER, :BUFFER)
    END-EXEC.
    DISPLAY " IS DONE!".
    MOVE SPACES TO EMP-NAME-ARR.
    GO TO INSERT-LOOP.

NOT-FOUND.
    EXEC SQL CLOSE EMPCUR END-EXEC.
    EXEC SQL COMMIT WORK END-EXEC.
    DISPLAY " ".
    DISPLAY
      "DONE INSERTING BITMAPS.  NEXT, LET'S DISPLAY SOME.".

DISP-LOOP.
    MOVE 0 TO INEMPNO.
    DISPLAY " ".
    DISPLAY "ENTER EMPLOYEE NUMBER (0 TO QUIT):  "
        WITH NO ADVANCING.

    ACCEPT D-INEMPNO.

    MOVE D-INEMPNO TO INEMPNO.
    IF (INEMPNO = 0)
        GO TO SIGN-OFF.
    EXEC SQL WHENEVER NOT FOUND GOTO NO-EMP END-EXEC.
    EXEC SQL SELECT EMP.EMPNO, ENAME, SAL, COMM, BITMAP
        INTO :EMP-NUMBER, :EMP-NAME, :SALARY,
             :COMMISSION:COMM-IND, :BUFFER
        FROM EMP, IMAGE
        WHERE EMP.EMPNO = :INEMPNO
          AND EMP.EMPNO = IMAGE.EMPNO
    END-EXEC.
    DISPLAY " ".
    PERFORM SHOW-IMAGE.
    MOVE EMP-NAME-ARR TO D-EMP-NAME.
    MOVE SALARY TO D-SALARY.
    MOVE COMMISSION TO D-COMMISSION.
    DISPLAY "EMPLOYEE ", D-EMP-NAME, " HAS SALARY ", D-SALARY
```

```
        WITH NO ADVANCING.
    IF COMM-IND = -1
        DISPLAY " AND NO COMMISSION."
    ELSE
        DISPLAY " AND COMMISSION ", D-COMMISSION, "."
    END-IF.
    MOVE SPACES TO EMP-NAME-ARR.
    GO TO DISP-LOOP.

NO-EMP.
    DISPLAY "NOT A VALID EMPLOYEE NUMBER - TRY AGAIN.".
    GO TO DISP-LOOP.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.
    DISPLAY " ".

GET-IMAGE.
    PERFORM MOVE-IMAGE
        VARYING INDX FROM 1 BY 1 UNTIL INDX > 8192.

MOVE-IMAGE.
    STRING '*' DELIMITED BY SIZE
        INTO BUFFER
        WITH POINTER INDX.
    DIVIDE 256 INTO INDX
        GIVING PRT-QUOT REMAINDER PRT-MOD.
    IF (PRT-MOD = 0) DISPLAY "." WITH NO ADVANCING.

SHOW-IMAGE.
    PERFORM VARYING INDX FROM 1 BY 1 UNTIL INDX > 10
        DISPLAY "               *************************"
    END-PERFORM.
    DISPLAY " ".

SIGN-OFF.
    EXEC SQL DROP TABLE IMAGE END-EXEC.
SIGN-OFF-EXIT.
    DISPLAY " ".
```

```
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:  ".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

5

埋込み SQL

この章では、埋込み SQL プログラミングの基本的な技法およびその適用方法について説明します。この章の構成は、次のとおりです。

ホスト変数の使用方法

データベースからプログラムにデータおよび状態情報を渡すとき、またデータベースにデータを渡すときには、ホスト変数を使います。

出力ホスト変数および入力ホスト変数

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ので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います。入力ホスト変数は WHERE 句、HAVING 句、FOR 句内でも使用されます。実際、入力ホスト変数は、SQL 文内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 句ではホスト変数を使用できますが、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
END-EXEC.
```

この文では、入力ホスト変数 *ORD* が使われていますが、この場合はこのホスト変数は定数として扱われます。そのため、ORD の値が何であっても順序付けは行われません。

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、入力ホスト変数を使用できません。つまり、ALTER、CREATE、DROP などのデータ定義文（*DDL* とも呼ばれます）で入力ホスト変数を使うことはできません。次の例の DROP TABLE 文は無効です。

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01 TABLE-NAME    PIC X(30) VARYING.
         ...
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
    DISPLAY 'Table name? '.
    ACCEPT TABLE-NAME.
    EXEC SQL DROP TABLE :TABLE-NAME END-EXEC.
*  -- host variable not allowed
```

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。次の例を考えてみます。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01     EMP-NUMBER   PIC S9(4) COMP.
 01     EMP-NAME     PIC X(20) VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
* -- get values for input host variables
     DISPLAY 'Employee number? '.
     ACCEPT EMP-NUMBER.
     DISPLAY 'Employee name? '.
     ACCEPT EMP-NAME.
     EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
         VALUES (:EMP-NUMBER, :EMP-NAME)
     END-EXEC.
```

INSERT 文の VALUES 句内の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。

標識変数の使用方法

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 文内で使うたびに、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。

VALUES 句または SET 句中の標識変数を使って、入力ホスト変数に NULL を割り当てたり、INTO 句中の標識変数を使って、出力ホスト変数内の NULL または切り捨てられた値を検出できます。

入力変数

入力ホスト変数の場合、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。

-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。

出力変数

出力ホスト変数の場合、Oracle が標識変数に割り当てる値の意味は次のとおりです。

-2	Oracle は、列の値を切り捨ててホスト変数に割り当てましたが、数値が大きすぎるため、列の値の元の長さを標識変数に割り当てることはできませんでした。
-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	Oracle によって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	Oracle は、列の値を切り捨ててホスト変数に割り当て、列の元の長さ（マルチバイト NLS ホスト変数では、バイト数ではなく文字数で表される）を標識変数に割り当てて、SQLCA 内の SQLCODE を 0（ゼロ）に設定しました。

標識変数は、2 バイトの整数として宣言しなければなりません。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（キーワード INDICATOR を使用しない場合）。

NULL の挿入

標識変数を使用して、NULL を挿入できます。挿入の前に、次に示すように、NULL にしたい列に対応する標識変数をそれぞれ -1 に設定します。

```
MOVE -1 TO IND-COMM.
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
    VALUES (:EMP-NUMBER, :COMMISSION:IND-COMM)
END-EXEC.
```

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)
END-EXEC.
```

この方法は、柔軟性に欠けますが、非常に理解しやすい方法です。

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。

```
DISPLAY 'Enter employee number or 0 if not available: '
    WITH NO ADVANCING.
ACCEPT EMP-NUMBER.
IF EMP-NUMBER = 0
   MOVE -1 TO IND-EMPNUM
ELSE
    MOVE 0 TO IND-EMPNUM
```

```
      END-IF.
      EXEC SQL INSERT INTO EMP (EMPNO, SAL)
         VALUES (:EMP-NUMBER:IND-EMPNUM, :SALARY)
      END-EXEC.
```

戻された NULL の処理

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
      END-EXEC.
      IF IND-COMM = -1
          MOVE SALARY TO PAY.
*   -- commission is null; ignore it
      ELSE
          ADD SALARY TO COMMISSION GIVING PAY.
      END-IF.
```

NULL のフェッチ

プリコンパイラ・オプション UNSAFE_NULL=YES と設定されている場合は、次の例に示すように、標識変数を持たないホスト変数に対して NULL を選択またはフェッチできます。

```
* --  assume that commission is NULL
      EXEC SQL SELECT ENAME, SAL, COMM
         INTO :EMP-NAME, :SALARY, :COMMISSION
         FROM EMP
         WHERE EMPNO = :EMP-NUMBER
      END-EXEC.
```

SQLCA 内の SQLCODE が 0（ゼロ）に設定されます。これは、Oracle がエラーまたは例外を検出せずにその文を実行したことを示します。

NULL が戻されたかどうか、あるいは NULL が戻された場合のホスト変数の値を調べる方法はありません。したがって、このオプション名は避けるようにします。新しいアプリケーションでは、UNSAFE_NULL=YES を使わないでください。この設定は、下位互換性を維持するためにのみ使います。

ただし、UNSAFE_NULL=NO の場合は、標識変数を持たないホスト変数に対して NULL を選択またはフェッチすると、Oracle はエラー・メッセージを発行します。

詳細は、14-39 ページの「UNSAFE_NULL」を参照してください。

NULL のテスト

WHERE 句で次の例のように標識変数を使って、NULL をテストできます。

```
EXEC SQL SELECT ENAME, SAL
   INTO :EMP-NAME, :SALARY
   FROM EMP
   WHERE :COMMISSION:IND-COMM IS NULL ...
```

しかし、関係演算子を使って 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 列が 1 つ以上の NULL を持つ場合、次の SELECT 文は失敗します。

```
EXEC SQL SELECT ENAME, SAL
   INTO :EMP-NAME, :SALARY
    FROM EMP
    WHERE COMM = :COMMISSION:IND-COMM
END-EXEC.
```

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:IND-COMM IS NULL))
END-EXEC.
```

切り捨てられた値のフェッチ

ホスト変数に値をフェッチする際に値が切り捨てられると、エラー・メッセージは発行されません。

基本的な SQL 文

実行 SQL 文を使うと、Oracle データの問合せ、操作および制御ができ、表、ビュー、索引などの Oracle オブジェクトを作成、定義および維持できます。この章では、データ操作文（*DML* とも呼ばれます）およびカーソル制御文に重点を置いて説明します。

次の SQL 文は Oracle データの問合せおよび操作に使用します。

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	表から行を削除します。

INSERT、UPDATE、DELETE などのデータ操作文を実行する場合、入力ホスト変数の値を設定すること以外に考慮すべき事項は、その文が正常終了したか異常終了したかだけです。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 通りの方法で調べることができます。

- 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

MODE={ANSI | ANSI14} のときは、状態変数 SQLSTATE または SQLCODE をチェックすることもできます。詳細は、8-4 ページの「MODE={ANSI | ANSI14} の場合の状態変数の使用」を参照してください。

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しなければなりません。問合せは次のように分類されます。

- 行を戻さない問合せ（有無を調べるだけ）
- 1 行だけを戻す問合せ
- 複数の行を戻す問合せ

複数の行を戻す問合せでは、カーソルまたはカーソル変数を明示的に宣言する必要があります。（あるいは、第 7 章の「ホスト表」で説明するホスト配列を使用します。）明示的なカーソルの定義および制御は、次の埋込み SQL 文で行います。

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、複数行を戻す SELECT 文の使用方法を説明します。各文およびその句の詳しい説明は、『Oracle8*i* SQL リファレンス』を参照してください。

行の選択

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 文を使います。次の例では、EMP 表を問い合せます。

```
EXEC SQL SELECT ENAME, JOB, SAL + 2000
   INTO :emp_name, :JOB-TITLE, :SALARY
   FROM EMP
   WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

キーワード SELECT の後に続く列名および式が選択リストを構成します。この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後続する句）内で指定した条件に基づいて、Oracle は INTO 句内のホスト変数に列の値を戻します。選択リスト内の項目の数は INTO 句のホスト変数の数と一致していなければなりません。これにより、戻された値すべてに格納場所が確保されます。

問合せで 1 行だけが戻される最も簡単なケースでは、形式は上記の例のようになります。（EMPNO は一意のキーです。）これに対し、問合せで複数行が戻される可能性がある場合には、カーソルを使って行を FETCH するか、行を SELECT してホスト配列に入れる必要があります。

1 行だけを戻すように作成した問合せで、実際には複数の行が戻される可能性がある場合、エラーになるかどうかはオプション SELECT_ERROR の指定によって決まります。SELECT_ERROR=YES（デフォルト）の場合は、複数行が戻されると Oracle はエラー・メッセージを発行します。

SELECT_ERROR=NO の場合は、1 行だけが戻され、エラーにはなりません。

使用できる句

SELECT 文では、標準 SQL 句（INTO、FROM、WHERE、CONNECT BY、START WITH、GROUP BY、HAVING、ORDER BY、FOR UPDATE OF）をすべて使えます。

行の挿入

INSERT 文を使うと、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)
END-EXEC.
```

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ものでなければなりません。VALUES 句には、挿入する行の値を指定します。指定する値は、定数、ホスト変数、SQL 式または疑似列（USER、SYSDATE など）のどの値でもかまいません。

VALUES 句内の値の数は、列リスト内の名前の数に等しくなければなりません。ただし、CREATE TABLE で定義したときと同じ順序で表の各列の値を VALUES 句に指定する場合は、列リストを省略できます。

DML 戻し句

INSERT、UPDATE および DELETE 文には、オプションの DML 戻し句を入れることができます。この句は、ホスト標識変数 *iv* を使用して、ホスト変数 *hv* に列値の式 *expr* を戻します。戻し句の構文は次のとおりです。

```
{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なければなりません。この句を使うと、INSERT または UPDATE の後、およびアプリケーションに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。DML 戻し句によって、ネットワークの非効率的なラウンドトリップ、余分な処理およびサーバーの使用メモリーを低減できます。

戻し句は、副問合せでは使用できません。VALUES 句の後だけで使用できます。

たとえば、前の INSERT の例の最後に次の句を入れることができます。

```
RETURNING empno, ename, deptno INTO :new_emp_number, :new_emp_name, :dept
```

DELETE、INSERT および UPDATE のエントリについては、付録付録 F の「埋込み SQL 文とプリコンパイラ ・ディレクティブ」を参照してください。

副問合せの使用方法

副問合せはネストされた SELECT 文です。副問合せを使用すると、複数部分の検索を処理できます。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。

たとえば、ある表から別の表に複数の行をコピーする場合は、次の例に示すように、INSERT 文の VALUES 句を副問合せに置き換えます。

```
EXEC SQL INSERT INTO EMP2 (EMPNO, ENAME, SAL, DEPTNO)
    SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP
    WHERE JOB = :JOB-TITLE
END-EXEC.
```

INSERT 文が中間結果を得るためにどのように副問合せを使っているのかに注意してください。

行の更新

UPDATE 文を使うと、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 表内の SAL 列と COMM 列を更新します。

```
EXEC SQL UPDATE EMP
   SET SAL = :SALARY, COMM = :COMMISSION
   WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

任意指定の WHERE 句を使用して、行を更新する条件を指定できます。詳細は、5-10 ページの「WHERE 句の使用方法」を参照してください。

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。次の例に示すように、副問合せを使うと値を指定できます。

```
EXEC SQL UPDATE EMP
SET SAL = (SELECT AVG(SAL)*1.1 FROM EMP WHERE DEPTNO = 20)
WHERE EMPNO = :EMP-NUMBER
END-EXEC.
```

行の削除

DELETE 文を使うと、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
END-EXEC.
```

任意指定の WHERE 句を使用して、行を削除する条件を指定できます。

WHERE 句の使用方法

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だけを選択、更新または削除できます。WHERE 句の検索条件は論理式であり、スカラー・ホスト変数、ホスト配列（SELECT 文を除く）および副問合せを使用できます。

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(5) 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。

カーソル

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うと、SQL 文に名前を付け、そのプライベート SQL 領域に保存されている情報にアクセスし、その処理をある程度まで制御できます。

静的 SQL 文の場合は、暗黙的なカーソルと明示的なカーソルの 2 種類のカーソルがあります。Oracle では、1 行だけを戻す SELECT 文（問合せ）を含め、すべてのデータ定義文およびデータ操作文のためにカーソルが 1 つ暗黙的に宣言されます。ただし、複数の行を戻す問合せの場合、2 行目以降を処理するには明示的にカーソルを宣言する（またはホスト配列を使用する）必要があります。

取り出された一連の行を結果セットと呼びます。結果セットのサイズは、問合せの検索条件を満たす行の数によって変わります。現在処理されている行（現在行と呼びます）を識別するには、明示的なカーソルを使います。

問合せで複数の行を戻す場合、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の後を処理できます。
- 現在どの行が処理されているかを追跡し記録できます。

カーソルは、問合せによって戻される行の集合内の現在行を示します。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。次の文を使ってカーソルを定義および操作します。

- DECLARE
- OPEN
- FETCH
- CLOSE

最初に DECLARE 文（正確にはディレクティブ）を使用して、カーソルに名称を付け、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。カーソルをオープンした後、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（ホスト配列を使っていない場合）。FETCH 文を使ってアクティブ・セット内の現在の行を取り出します。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・セットからの行のフェッチが終了したら、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。）

カーソルの宣言

次の例に示すように、DECLARE 文でカーソルに名称を与え、問合せに関連付けることによって、カーソルを定義できます。

```
      EXEC SQL DECLARE EMPCURSOR CURSOR FOR
         SELECT ENAME, EMPNO, SAL
         FROM EMP
         WHERE DEPTNO = :DEPT_NUMBER
      END-EXEC.
```

カーソル名は、ホスト変数やプログラム変数ではなく、プリコンパイラが使用する識別子なので、COBOL 文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単位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。

DECLARE CURSOR 文で WITH HOLD 句を使用することにより、COMMIT または ROLLBACK の後もカーソルをオープンしたままにすることができます。

コマンド行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。詳細は、3-20 ページの「DECLARE CURSOR 文での WITH HOLD 句の使用」および 14-14 ページの「CLOSE_ON_COMMIT」を参照してください。

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。デフォルト値は、MODE=ORACLE および CLOSE_ON_COMMIT=NO です。MODE=ANSI と指定した場合は、WITH HOLD 句を使っ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については、14-4 ページの「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 文は宣言部分なので、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（論理的にではなく）前になければなりません。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 EMPCURSOR END-EXEC.
*    -- MISPLACED OPEN STATEMENT
      EXEC SQL DECLARE EMPCURSOR CURSOR FOR
         SELECT ENAME, EMPNO, SAL
         FROM EMP
         WHERE ENAME = :EMP-NAME
      END-EXEC.
```

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、ソース・ファイル A.PCO ではカーソルを宣言できませんが、ソース・ファイル B.PCO ではカーソルをオープンするという処理ができます。

ホスト・プログラムではカーソルを任意の数だけ宣言できます。ただし、指定されたファイル内ではそれぞれの DECLARE 文は一意でなければなりません。つまり、カーソルの有効範囲はファイル内でグローバルなので、1 つのプリコンパイル単位の中では、ブロックやプロシージャが異なっていたとしても同じ名前のカーソルを 2 つ宣言することはできません。カーソルを数多く使うときは、MAXOPENCURSORS オプションの指定が必要な場合があります。詳細は、14-29 ページの「MAXOPENCURSORS」を参照してください。

カーソルのオープン

OPEN 文を使用して、問合せを実行し、アクティブ・セットを決定します。次の例では、*EMPCURSOR* という名前のカーソルがオープンされます。

```
EXEC SQL OPEN EMPCURSOR END-EXEC.
```

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 の SQLERRD(3) に格納されている処理済みの行カウントが 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。

カーソルをオープンすると、問合せの入力ホスト変数はカーソルを再オープン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、カーソルを再オープンします。

通常、カーソルを再オープンするには、その前にそのカーソルをクローズする必要があります。ただし、CLOSE_ON_COMMIT=YES と設定した場合は、再オープンする前にカーソルをクローズする必要はありません。これによってパフォーマンスが向上します。詳細は、付録 D の「パフォーマンス・チューニング」を参照してください。

OPEN によって行われる作業量は、HOLD_CURSOR、RELEASE_CURSOR、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。詳細は、14-12 ページの「Pro*COBOL プリコンパイラ・オプションの使用」のアルファベット順の項目を参照してください。

カーソルからのフェッチ

FETCH 文を使うと、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 つのホスト変数に格納します。

```
EXEC SQL FETCH EMPCURSOR
   INTO :EMP-NAME, :EMP-NUMBER, :SALARY
```

```
END-EXEC.
```

カーソルは、あらかじめ宣言し、オープンしておかなければなりません。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。この行が現在行になります。その後 FETCH を実行するたびに、カーソルはアクティブ・セットの次の行に進みます。（現在行を変更します。）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。すでにフェッチした行に戻るには、カーソルを再オープンし、アクティブ・セットの最初の行からやり直す必要があります。

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、カーソルを再オープンしてください。CLOSE_ON_COMMIT=NO と設定されている場合は、再オープンする前にカーソルをクローズする必要があります。

次の例に示すように、異なる出力ホスト変数セットを使って同じカーソルからフェッチすることができます。しかし、各 FETCH 文の INTO 句内の対応するホスト変数は、同じデータ型でなければなりません。

```
     EXEC SQL DECLARE EMPCURSOR CURSOR FOR
         SELECT ENAME, SAL FROM EMP WHERE DEPTNO = 20
     END-EXEC.
     ...
     EXEC SQL OPEN EMPCURSOr END-EXEC.
     EXEC SQL WHENEVER NOT FOUND DO ...
 PERFORM
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME1, :SAL1 END-EXEC
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME2, :SAL2 END-EXEC
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME3, :SAL3 END-EXEC
     ...
 END-PERFORM.
```

アクティブ・セットが空か、または他に行がない場合、FETCH を実行すると " データがありません " という Oracle 警告コードが SQLCA の SQLCODE（MODE=ANSI の場合は状態変数 SQLSTATE）に戻されます。出力ホスト変数のステータスは予測不能です。（通常のプログラムでは、WHENEVER NOT FOUND 文でこのエラーを検出します。）そのカーソルを再使用するには、再オープンしなければなりません。

カーソルのクローズ

アクティブ・セットからの行のフェッチが終了したら、カーソルをクローズし、そのカーソルのオープンによって獲得していたリソース（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、オプション HOLD_CURSOR および RELEASE_CURSOR の指定によって異なります。次の例では、*EMPCURSOR* という名前のカーソルをクローズします。

```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

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ので、クローズしたカーソルからフェッチすることはできません。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）カーソルを再オープンできます。

CLOSE_ON_COMMIT=NO の場合、コミットまたはロールバックを行うと、CURRENT OF 句で参照されているカーソルがクローズされます。その他のカーソルは、コミットまたはロールバック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になります。これに対し、CLOSE_ON_COMMIT=YES の場合は、コミットまたはロールバックを行うと明示的カーソルがすべてクローズされます。詳細は、3-20 ページの「CLOSE_ON_COMMIT プリコンパイラ・オプション」を参照してください。

CURRENT OF 句の使用方法

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フェッチ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。フェッチが 1 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宣言するときに、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 句を追加するようプリコンパイラに指示します。詳細は、7-18 ページの「CURRENT OF 句の疑似実行」を参照してください。

次の例では、CURRENT OF 句を使用して、*EMPCURSOR* という名前のカーソルから最後にフェッチした行を参照します。

```
     EXEC SQL DECLARE EMPCURSOR CURSOR FOR
         SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK'
         FOR UPDATE OF SAL
     END-EXEC.
     ...
     EXEC SQL OPEN EMPCURSOR END-EXEC.
     EXEC SQL WHENEVER NOT FOUND DO ...
 PERFORM
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
     END-EXEC
     ...
     EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
         WHERE CURRENT OF EMPCURSOR
     END-EXEC
 END-PERFORM.
```

制限

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。いずれの行も、フェッチされるときではなくオープン時にロックされ、コミットまたはロールバックを行うとロックは解除されます。コミットした後で FOR UPDATE カーソルからフェッチしようとすると、エラーが発行されます。

CURRENT OF 句でホスト配列を使用することはできません。他の方法については、7-18 ページの「CURRENT OF 句の疑似実行」を参照してください。また、関連付けられた FOR UPDATE OF 句で複数の表を参照することはできません。つまり、CURRENT OF 句との結合は行えません。さらに、動的 SQL で CURRENT OF 句を使用することもできません。

一般的な文の順序

次の例に、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ます。

```
* --  Define a cursor.
     EXEC SQL DECLARE EMPCURSOR CURSOR FOR
         SELECT ENAME, JOB FROM EMP
         WHERE EMPNO = :EMP-NUMBER
         FOR UPDATE OF JOB
     END-EXEC.
* --  Open the cursor and identify the active set.
     EXEC SQL OPEN EMPCURSOR END-EXEC.
* --  Exit if the last row was already fetched.
     EXEC SQL
         WHENEVER NOT FOUND DO PERFORM NO-MORE
     END-EXEC.
* --  Fetch and process data in a loop.
 PERFORM
       EXEC SQL FETCH EMPCURSOR INTO :EMP-NAME, :JOB-TITLE
       END-EXEC
* -- host-language statements that operate on the fetched data
       EXEC SQL UPDATE EMP
          SET JOB = :NEW-JOB-TITLE
          WHERE CURRENT OF EMPCURSOR
       END-EXEC
 END-PERFORM.
     ...
 MO-MORE.
* --  Disable the cursor.
       EXEC SQL CLOSE EMPCURSOR END-EXEC.
       EXEC SQL COMMIT WORK RELEASE END-EXEC.
     STOP RUN.
```

サンプル・プログラム 2: カーソル操作

次のプログラムは、Oracle にログインし、カーソルを宣言およびオープンし、全販売員の名前、給料、歩合給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。

（最後のフェッチを除く）すべてのフェッチは 1 行を戻し、フェッチ中にエラーが検出されなかった場合には成功の状態コードを戻します。最後のフェッチは失敗し、" データがありません " という Oracle 警告コードが SQLCA の SQLCODE に戻されます。実際にフェッチされた行の累計数は、SQLCA の SQLERRD(3) に格納されます。

```
*****************************************************************
* Sample Program 2:  Cursor Operations                          *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches the names, salaries, and commissions of all           *
* salespeople, displays the results, then closes the cursor.    *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. CURSOR-OPS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME          PIC X(10) VARYING.
 01  PASSWD            PIC X(10) VARYING.
 01  EMP-REC-VARS.
     05  EMP-NAME      PIC X(10) VARYING.
     05  SALARY        PIC S9(6)V99
                       DISPLAY SIGN LEADING SEPARATE.
     05  COMMISSION    PIC S9(6)V99
                       DISPLAY SIGN LEADING SEPARATE.
     EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
     EXEC SQL VAR COMMISSION IS DISPLAY(8,2) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.

     EXEC SQL INCLUDE SQLCA END-EXEC.

 01  DISPLAY-VARIABLES.
     05  D-EMP-NAME    PIC X(10).
     05  D-SALARY      PIC Z(4)9.99.
     05  D-COMMISSION  PIC Z(4)9.99.

 PROCEDURE DIVISION.

 BEGIN-PGM.
     EXEC SQL WHENEVER SQLERROR
```

```
        DO PERFORM SQL-ERROR END-EXEC.
    PERFORM LOGON.
    EXEC SQL DECLARE SALESPEOPLE CURSOR FOR
        SELECT ENAME, SAL, COMM
        FROM EMP
        WHERE JOB LIKE 'SALES%'
    END-EXEC.
    EXEC SQL OPEN SALESPEOPLE END-EXEC.
    DISPLAY " ".
    DISPLAY "SALESPERSON  SALARY      COMMISSION".
    DISPLAY "-----------  ----------  ----------".

FETCH-LOOP.
    EXEC SQL WHENEVER NOT FOUND
        DO PERFORM SIGN-OFF END-EXEC.
    EXEC SQL FETCH SALESPEOPLE
        INTO :EMP-NAME, :SALARY, :COMMISSION
    END-EXEC.
    MOVE EMP-NAME-ARR TO D-EMP-NAME.
    MOVE SALARY TO D-SALARY.
    MOVE COMMISSION TO D-COMMISSION.
    DISPLAY D-EMP-NAME, "     ", D-SALARY, "    ", D-COMMISSION.
    MOVE SPACES TO EMP-NAME-ARR.
    GO TO FETCH-LOOP.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.

SIGN-OFF.
    EXEC SQL CLOSE SALESPEOPLE END-EXEC.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
```

```
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
STOP RUN.
```

PREFETCH オプション

プリコンパイラ・オプション PREFETCH を使えば、特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。これによって、サーバーのラウンドトリップの必要回数が増えます。構成ファイルまたはコマンド行の PREFETCH オプション値で設定した行数は、標準の優先順位規則に従って、明示的な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 オプションを指定しなければなりません。

- 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）～ 9999 までの値を設定できます。

6

埋込み PL/SQL

この章では、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について説明します。この章の構成は、次のとおりです。

PL/SQL の埋込み

Pro*COBOL は、PL/SQL ブロックを単一の埋込み SQL 文のように扱います。そのため、PL/SQL ブロックは、SQL 文を記述できる位置であればホスト・プログラム内のどこにでも記述できます。

ホスト・プログラムに PL/SQL ブロックを埋め込むには、PL/SQL との間で共有される変数を宣言し、PL/SQL ブロックを EXEC SQL EXECUTE キーワードと END-EXEC キーワードで囲みます。

ホスト変数

PL/SQL ブロックの中では、ホスト変数はブロック全体にわたるグローバルなものとして扱われ、PL/SQL 変数を記述できる位置であればどこにでも記述できます。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。

VARCHAR 変数

PL/SQL ブロックに入ると、Oracle8*i* は自動的に VARCHAR ホスト変数の長さフィールドをチェックします。このため、ブロックに入る前に長さフィールドを設定する必要があります。入力変数の長さフィールドは、文字列フィールドに格納される値の長さに設定します。出力変数の場合は、その文字列フィールドに許される最大の長さに設定します。

標識変数

PL/SQL ブロックでは、標識変数だけを参照することはできません。標識変数は、対応するホスト変数の後に記述する必要があります。また、標識変数によってホスト変数を参照する場合は、同じブロック内では常に標識変数を使ってそのホスト変数を参照しなければなりません。

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。

切捨て値の処理

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、例外とは見なされません。しかし、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。

SQLCHECK

埋込み PL/SQL ブロックを持つプログラムをプリコンパイルする際には、SQLCHECK=SEMANTICS を指定しなければなりません。また、USERID オプションも使用する必要があります。詳細は、第 14 章の「プリコンパイラのオプション」を参照してください。

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について説明します。

- パフォーマンスの向上
- Oracle8*i* との統合
- カーソル FOR ループ
- プロシージャとファンクション
- パッケージ
- PL/SQL 表
- ユーザー定義レコード

PL/SQL についての詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

パフォーマンスの向上

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、生産性の向上が図れます。たとえば、PL/SQL を使わないと、Oracle8*i* は一度に 1 つずつ SQL 文を処理しなければなりません。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 を使うと、サーバーに SQL 文のブロック全体を送ることができます。これによって、アプリケーションとサーバーとの間の通信は最小限になります。

Oracle8*i* との統合

PL/SQL はサーバーと密接に統合されます。たとえば、PL/SQL データ型の大部分は、データ・ディクショナリに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 属性が使用できます。

```
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、列定義を変更すれば、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。

カーソル FOR ループ

PL/SQL を使えば、カーソルを定義し操作するために DECLARE 文、OPEN 文、FETCH 文および CLOSE 文を指定する必要がありません。かわりに、カーソル FOR ループを指定できます。カーソル FOR ループは、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、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;
```

レコード中のフィールドの参照にはドット表記法を使用することに注意してください。

サブプログラム

PL/SQL にはプロシージャおよび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ます。これらを使うと、各処理を個別に行えるため、アプリケーション開発が容易になります。一般的には、プロシージャを使って処理を行い、ファンクションを使って値を計算します。

プロシージャと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うことにより、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、次のようにコーディングします。

```
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、新しい部門名および新しい位置が確立され、部門番号データベース順序内のその次の値が選択され、新しい番号、名前および位置が *dept* 表の中に挿入されます。次に、新しい番号がコール側に戻ります。

サブプログラムをその都度再コンパイルせずに、複数のアプリケーションからコールできます。（CREATE FUNCTION および CREATE PROCEDURE を使用してサブプログラムをデータベースに格納できます。）

パラメータ・モード

仮パラメータの動作を定義するには、パラメータ・モードを使います。パラメータ・モードには、IN（デフォルト）、OUT、IN OUT の 3 種類があります。IN パラメータを使うと、コールされるサブプログラムに値を渡すことができます。OUT パラメータを使うと、サブプログラムのコール側に値を戻すことができます。IN OUT パラメータを使うと、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側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、その対応する仮パラメータのデータ型に変換可能である必要があります。6-15 ページの表 6-1 にデータ型間の有効な変換を示します。

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。パッケージは、コンパイルしてデータベースに格納できます。これにより、パッケージの内容を複数のアプリケーションで共有できるようになります。

パッケージには通常、仕様部と本体の 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す。仕様部には、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、カーソルおよびサブプログラムを定義して、仕様部を実行します。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だけが参照可能で、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隠されていてアクセスできません。

PL/SQL 表

PL/SQL には TABLE という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 列から成り、主キーを使って、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、NUMBER など）にできますが、主キーは型 BINARY_INTEGER にしなければなりません。

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分で 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* 型を定義すれば、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って 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* という名前の 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) ...
```

ユーザー定義レコード

%ROWTYPE 属性を使って、データベース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、レコード内のフィールドのデータ型を指定することはできず、ユーザー独自のフィールドを定義することもできません。複合データ型 RECORD を使用することによって、これらの制限事項を取り除くことができます。

RECORD 型のオブジェクトをレコードといいます。PL/SQL 表とは異なり、レコードには一意の名前のフィールドがあります。フィールドのデータ型は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、給料、雇用日など）があるとします。これらのデータは、型は異なっていますが論理的に関連しています。従業員の名前、給料、雇用日などのフィールドを持つレコードによって、これらのデータを 1 つの論理単位として処理できます。

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分で、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 := 10,  -- must initialize
         dname   CHAR(9),
         loc     CHAR(14));
```

フィールド宣言は変数宣言と似ています。各フィールドには一意の名前と固有のデータ型を指定します。フィールド宣言に NOT NULL オプションを追加することによって、そのフィールドに NULL が割り当てられないようにできます。ただし、NOT NULL を指定したフィールドは初期化する必要があります。

次の例に示すように、一度 *DeptRecTyp* を定義すれば、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。

レコード内の個々のフィールドを参照するには、ドット表記法を使います。たとえば、*dept_rec* レコードの *dname* フィールドを参照する場合は、次のように記述します。

```
dept_rec.dname ...
```

PL/SQL ブロックの埋込み

Pro*COBOL は、PL/SQL ブロックを単一の埋込み SQL 文のように扱います。そのため、PL/SQL ブロックは、SQL 文を記述できる位置であればホスト・プログラム内のどこにでも記述できます。

PL/SQL ブロックをホスト・プログラム内に埋め込むには、次のように、キーワード EXEC SQL EXECUTE および END-EXEC で PL/SQL ブロックを囲むだけです。

```
EXEC SQL EXECUTE
    DECLARE
    ...
    BEGIN
    ...
    END;
END-EXEC.
```

プログラムで PL/SQL ブロックを埋め込む場合、Pro*COBOL で PL/SQL を解析する必要があるので、プリコンパイラ・オプション SQLCHECK=SEMANTICS を指定しなければなりません。サーバーに接続する場合も、オプション USERID を指定する必要があります。詳細は、14-12 ページの「Pro*COBOL プリコンパイラ・オプションの使用」を参照してください。

ホスト変数と PL/SQL

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 と共有できます。これにより、PL/SQL でのホスト変数の設定および参照が可能になります。

たとえば、外国語の文字変数を宣言し、それを文字列関数（INSTRB、LENGTHB、SUBSTRB など）に渡すことができます。これにより、PL/SQL を使っ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。ただし、文字ホスト変数の長さは 255 文字までです。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。

PL/SQL の例

次の例では、PL/SQL におけるホスト変数の使用方法について示します。プログラムはユーザーに従業員番号の入力を求め、その番号に応じて、従業員の役職名、雇用日、給与を表示します。

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 USERNAME   PIC X(20) VARYING.
 01 PASSWORD   PIC X(20) VARYING.
 01 EMP-NUMBER PIC S9(4) COMP.
 01 JOB-TITLE  PIC X(20) VARYING.
 01 HIRE-DATE  PIC X(9) VARYING.
 01 SALARY     PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
    ...
    DISPLAY 'Username? ' WITH NO ADVANCING.
    ACCEPT USERNAME.
    DISPLAY 'Password? ' WITH NO ADVANCING.
    ACCEPT PASSWORD.
    EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWORD
    END-EXEC.
    DISPLAY 'Connected to Oracle'.
 PERFORM
    DISPLAY 'Employee Number (0 to end)? 'WITH NO ADVANCING
    ACCEPT EMP-NUMBER
    IF EMP-NUMBER = 0
        EXEC SQL COMMIT WORK RELEASE END-EXEC
```

```
         DISPLAY 'Exiting program'
         STOP RUN
     END-IF.
*    ---------------- 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 -----------------
     DISPLAY 'Number  Job Title  Hire Date  Salary'.
     DISPLAY '-----------------------------------'.
     DISPLAY EMP-NUMBER, JOB-TITLE, HIRE-DATE, SALARY.
 END-PERFORM.
     ...
 SQL-ERROR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     DISPLAY 'Processing error'.
     STOP RUN.
```

ホスト変数 *EMP-NUMBER* は PL/SQL ブロックに入る前に設定され、ホスト変数 *JOB-TITLE*、*HIRE-DATE* および *SALARY* はブロックの中で設定されていることに注意してください。

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、次に口座への記帳を行います。口座が存在しない場合は、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。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 USERNAME   PIC X(20) VARYING.
 01 ACCT-NUM   PIC S9(4) COMP.
 01 TRANS-TYPE PIC X(1).
 01 TRANS-AMT  PIC PIC S9(6)V99
                      DISPLAY SIGN LEADING SEPARATE.
 01 STATUS     PIC X(80) VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.
     DISPLAY 'Username? 'WITH NO ADVANCING.
     ACCEPT USERNAME.
     DISPLAY 'Password? '.
     ACCEPT PASSWORD.
```

```
     EXEC SQL WHENEVER SQLERROR DO PERFORM SQL-ERROR.
     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD.
     PERFORM
     DISPLAY 'Account Number (0 to end)? '
         WITH NO ADVANCING
     ACCEPT ACCT_NUM
     IF ACCT-NUM = 0
         EXEC SQL COMMIT WORK RELEASE END-EXEC
         DISPLAY 'Exiting program' WITH NO ADVANCING
         STOP RUN
     END-IF.
     DISPLAY 'Transaction Type - D)ebit or C)redit? '
         WITH NO ADVANCING
     ACCEPT TRANS-TYPE
     DISPLAY 'Transaction Amount? '
     ACCEPT trans_amt
*   -------------------- begin PL/SQL block -------------------
     EXEC SQL EXECUTE
        DECLARE
            old_bal      NUMBER(9,2);
            err_msg      CHAR(70);
            nonexistent  EXCEPTION;
        BEGIN
           :TRANS-TYP-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 -----------------------
     DISPLAY 'Status: ', STATUS
 END-PERFORM.
     ...
 SQL-ERROR.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     DISPLAY 'Processing error'.
     STOP RUN.
```

VARCHAR 疑似型

VARCHAR 疑似型は、可変長の文字列を宣言するのに使用できます。VARCHAR が入力ホスト変数の場合は、どのくらいの長さを予期したらよいかを Pro*COBOL に指示しなければなりません。このため、長さフィールドは、文字列フィールドに格納される値の実際の長さに設定してください。

VARCHAR が出力ホスト変数の場合、Pro*COBOL は自動的に長さフィールドを設定します。しかし、PL/SQL ブロックで VARCHAR 出力ホスト変数を使うには、ブロックに入る前に長さフィールドを初期化する必要があります。したがって、次の例に示すように、宣言された（最大の）VARCHAR 長に、長さフィールドを設定してください。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01 EMP-NUM   PIC S9(4) COMP.
        01 EMP-NAME  PIC X(10) VARYING.
        01 SALARY    PIC S9(6)V99
                     DISPLAY SIGN LEADING SEPARATE.
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
 PROCEDURE DIVISION.
     ...
* -- initialize length field
     MOVE 10 TO EMP-NAME-LEN.
     EXEC SQL EXECUTE
     BEGIN
        SELECT ename, sal INTO :EMP-NAME, :SALARY
            FROM emp
            WHERE empno = :EMP-NUM;
        ...
     END;
```

```
END-EXEC.
```

標識変数と PL/SQL

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 演算子を使って NULL をテストできます。

```
IF variable IS NULL THEN ...
```

次のように、代入演算子（:=）を使って NULL を割り当てることができます。

```
variable := NULL;
```

しかし、ホスト言語は NULL を扱えないので、標識変数が必要です。埋込み PL/SQL でこの要件を満たすには、標識変数を次の用途に使用します。

- ホスト・プログラムからの NULL 入力値の受け入れ
- 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 ブロックで標識変数を使う場合は、次の規則に従ってください。

- 標識変数自体は参照できません。対応するホスト変数に追加する必要があります。
- 標識変数によってホスト変数を参照する場合、同じブロック内では常にホスト変数を使って標識変数を参照しなければなりません。

次の例では、標識変数 *IND-COMM* は、SELECT 文でホスト変数 *COMMISSION* とともに記述されているため、IF 文でもホスト変数とともに記述しなければなりません。

```
EXEC SQL EXECUTE
BEGIN
   SELECT ename, comm
       INTO :EMP-NAME, :COMMISSION:IND-COMM FROM emp
       WHERE empno = :EMP-NUM;
   IF :COMMISSION:IND-COMM IS NULL THEN ...
   ...
END;
END-EXEC.
```

:COMMISSION:IND-COMM は、PL/SQL では他の単純な変数と同じように扱われます。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。

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

ロックに入る前に *IND-SAL* の値が -1 になっていると、*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条件です。

```
EXEC SQL EXECUTE
BEGIN
    IF :SALARY:IND-SAL IS NULL THEN
        RAISE salary_missing;
    END IF;
    ...
END;
END-EXEC.
```

切捨て値の処理

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。しかし、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・プログラムは、*IND-NAME* の値をチェックすることによって、切り捨てられた値が *EMP-NAME* に割り当てられたかどうかを判別できます。

```
EXEC SQL EXECUTE
DECLARE
    ...
    new_name  CHAR(10);
BEGIN
    ...
    :EMP_NAME:IND-NAME := new_name;
    ...
END;
END-EXEC.
```

ホスト表と PL/SQL

入力ホスト表および標識表を PL/SQL ブロックに渡すことができます。入力ホスト表および標識表には、BINARY_INTEGER 型の PL/SQL 変数、またはその型と互換性のあるホスト変数を使って索引付けができます。通常はホスト表全体が PL/SQL に渡されますが、ARRAYLEN 文（後で説明）を使って、より小さい表ディメンションを指定できます。

また、サブプログラム・コールを使用して、ホスト表内のすべての値を PL/SQL 表内の行に割り当てることができます。表の添字範囲が *m* ～ *n* であると、対応する PL/SQL 表の索引範囲は常に 1 ～（*n*-*m*+ 1）になります。たとえば、表の添字範囲が 5 ～ 10 であると、対応する PL/SQL 表の索引範囲は 1 ～（10-5+1）つまり 1 ～ 6 になります。

注意: Pro*COBOL では、ホスト表の使用方法はチェックされません。たとえば、索引の範囲チェックは行われません。

次の例では、*salary* という名前のホスト表を PL/SQL ブロックに渡します。PL/SQL ブロックでは、このホスト表がファンクション・コールで使用されます。この関数は、一連の数値の中央値を検出するので、*median* という名前が付いています。この関数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・コールは、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。

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
01 SALARY OCCURS 100 TIMES PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
        01 MEDIAN-SALARY  PIC S9(6)V99
        DISPLAY SIGN LEADING SEPARATE.
     EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host table
     EXEC SQL EXECUTE
     DECLARE
        TYPE NumTabTyp IS TABLE OF REAL
            INDEX BY BINARY_INTEGER;
        n  BINARY_INTEGER;
        ...
        FUNCTION median (num_tab NumTabTyp, n INTEGER)
            RETURN REAL IS
        BEGIN
* -- compute median
        END;
     BEGIN
        n := 100;
        :MEDIAN-SALARY := median(:SALARY    END;
     END-EXEC.
```

また、サブプログラム・コールを使って、PL/SQL 表内のすべての行の値をホスト表内の対応する要素に割り当てることもできます。6-20 ページの「ストアド PL/SQL と Java サブプログラム」の例を参照してください。

表 6-1 に、PL/SQL 表内の行の値と、ホスト表内の要素との間の有効な変換を示します。たとえば、LONG 型のホスト表は VARCHAR2、LONG、RAW または LONG RAW 型の PL/SQL 表と互換性があります。しかし、CHAR 型の PL/SQL 表とは互換性がないので注意してください。

表 6-1 データ型の有効な変換

PL/SQL 表 ホスト表	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	X							
CHARZ	X							
DATE		X						
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		

ARRAYLEN 文

処理のために入力ホスト表を PL/SQL ブロックに渡す必要があるとします。デフォルトでは、このようなホスト表をバインドする際に、Pro*COBOL は宣言されたディメンションを使用します。しかし、表の一部だけを処理したい場合もあります。このような場合には、ARRAYLEN 文を使ってより小さい表ディメンションを指定できます。ARRAYLEN 文は、ホスト表をより小さいディメンションを格納するホスト変数と関連付けます。構文は次のとおりです。

```
     EXEC SQL ARRAYLEN host_array (dimension) EXECUTE END-EXEC.
```

dimension は 4 バイトの整数ホスト変数です。リテラルや式ではありません。

ARRAYLEN 文は、*host_array* および *dimension* の宣言の後に置く必要があります。ホスト表にオフセットを指定することはできません。ただし、そのために COBOL 機能を使用できる場合があります。

次の例では、ARRAYLEN を使って、*BONUS* という名前のホスト表のデフォルトのディメンションを指定変更しています。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 BONUS OCCURS 100 TIMES PIC S9(6)V99
         DISPLAY SIGN LEADING SEPARATE.
 01 MY-DIM  PIC S9(4) COMP.
     ...
     EXEC SQL ARRAYLEN BONUS (MY-DIM) END-EXEC.
     EXEC SQL END DECLARE SECTION END-EXEC.
* -- populate the host table
     ...
* -- set smaller table dimension
     MOVE 25 TO MY-DIM.
     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, :MY-DIM);
         ...
     END;
     END-EXEC.
```

ARRAYLEN によってホスト表の要素が 100 から 25 に減らされるので、25 の表要素だけが PL/SQL ブロックに渡されます。結果として、実行のために PL/SQL ブロックがサーバーに送られるときに、一緒に送られるホスト表ははるかに小さくなります。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通信量を削減できます。

オプションのキーワード EXECUTE

動的 SQL 方法 2 の EXEC SQL EXECUTE 文で使われるホスト表には、オプションのキーワード EXECUTE の有無によって 2 種類の解釈があります。詳細は、9-12 ページの「方法 2 の使用方法」を参照してください。

デフォルト（キーワード EXECUTE なし）では、次のようになります。

- ホスト配列は、PL/SQL ブロックの実行回数を決定する際に考慮されます。配列の最小ディメンションが使われます。
- ホスト配列は、PL/SQL 索引表にバインドできません。

キーワード EXECUTE がある場合は、次のようになります。

- ホスト表は、PL/SQL 索引表にバインドする必要があります。
- PL/SQL ブロックは、1 回だけ実行されます。
- EXEC SQL EXECUTE 文で指定するホスト変数は、次のいずれかになります。
  - ARRAYLEN...EXECUTE 文で指定
  - スカラー値

たとえば、次の PL/SQL プロシージャを使うとします。

```
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*COBOL の例に、ホスト表を使って PL/SQL ブロックの実行回数を決定する方法を示します。この例では、PL/SQL ブロックは 3 回実行され、*emp* 表に 3 行の新規の行が作成されます。

```
      ...
01  DYNSTMT   PIC X(80) VARYING.
01  EMPNOTAB  PIC S9(4) COMPUTATIONAL OCCURS 5 TIMES.
01  ENAMETAB  PIC X(10) OCCURS 3 TIMES.
01  DIM       PIC S9(9) COMP VALUE 2.
      ...
          MOVE 1111 TO EMPNOTAB(1).
          MOVE 2222 TO EMPNOTAB(2).
          MOVE 3333 TO EMPNOTAB(3).
          MOVE 4444 TO EMPNOTAB(4).
```

```
            MOVE 5555 TO EMPNOTAB(5).

            MOVE "MICKEY" TO ENAMETAB(1).
            MOVE "MINNIE" TO ENAMETAB(2).
            MOVE "GOOFY" TO ENAMETAB(3).

            MOVE "BEGIN INSERT INTO emp(empno, ename) VALUES :b1, :b2; END;"
              TO DYNSTMT-ARR.
            MOVE 57 TO DYNSTMT-LEN.

            EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
            EXEC SQL EXECUTE s1 USING :EMPNOTAB, :ENAMETAB END-EXEC.
        ...
```

次の Pro*COBOL の例に、動的方法 2 を使ってホスト表を PL/SQL 索引表にバインドする方法を示します。EXEC SQL EXECUTE 文で指定したすべてのホスト配列に ARRAYLEN...EXECUTE 文が存在することに注意してください。

```
        ...
 01  DYNSTMT   PIC X(80) VARYING.
 01  II        PIC S9(4) COMP VALUE 2.
 01  INTTAB    PIC S9(9) COMP OCCURS 3 TIMES.
 01  DIM       PIC S9(9) COMP VALUE 3.

            EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
        ...
            MOVE 1 TO INTTAB(1).
            MOVE 2 TO INTTAB(2).
            MOVE 3 TO INTTAB(3).

            MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";
              TO DYNSTMT-ARR.
            MOVE 37 TO DYNSTMT-LEN.

            EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
            EXEC SQL EXECUTE s1 USING :INTTAB, :II, :INTTAB END-EXEC.
        ...
```

ただし、次の Pro*COBOL の例では INTTAB2 に ARRAYLEN...EXECUTE 文が存在しないので、プリコンパイル時にエラーになります。

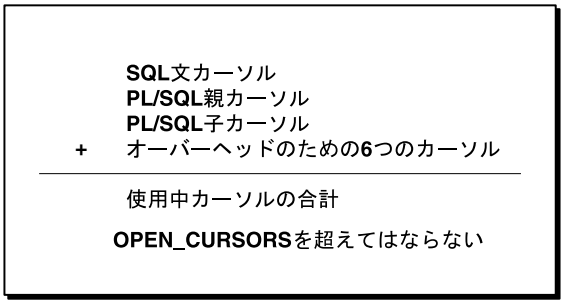
```
        ...
 01  DYNSTMT   PIC X(80) VARYING.
 01  INTTAB    PIC S9(9) COMP OCCURS 3 TIMES.
 01  INTTAB2   PIC S9(9) COMP OCCURS 3 TIMES.
 01  DIM       PIC S9(9) COMP VALUE 3.

            EXEC SQL ARRAYLEN INTTAB (DIM) EXECUTE END-EXEC.
```

```
...
    MOVE 1 TO INTTAB(1).
    MOVE 2 TO INTTAB(2).
    MOVE 3 TO INTTAB(3).

    MOVE "BEGIN pkg.proc1 (:v1, :v2, :v3); end;";
      TO DYNSTMT-ARR.
    MOVE 37 TO DYNSTMT-LEN.

    EXEC SQL PREPARE s1 FROM :DYNSTMT END-EXEC.
    EXEC SQL EXECUTE s1 USING :INTTAB, :INTTAB2, :INTTAB END-EXEC.
...
```

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、データベースの初期化パラメータ OPEN_CURSORS によって決まります。埋込み PL/SQL ブロックの実行中は、親カーソルと呼ばれる 1 つのカーソルがブロック全体に関連付けられ、子カーソルと呼ばれるカーソルが埋込み PL/SQL ブロック内の各 SQL 文に 1 つずつ関連付けられます。親と子どちらのカーソルも、OPEN_CURSORS に対してカウントされます。図 6-1 の「使用される最大カーソル数」に、使用される最大カーソル数を計算する方法を示します。

図 6-1　使用される最大カーソル数



プログラムが OPEN_CURSORS で設定された制限を超すと、エラーが発生します。このエラーを回避するには、RELEASE_CURSOR=YES オプションおよび HOLD_CURSOR=NO オプションを指定します。RELEASE_CURSOR を YES に設定してプログラム全体をプリコンパイルしたくない場合は、次に示すように、各 PL/SQL ブロックの後で NO にリセットしてください。

```
     EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.
* --  first embedded PL/SQL block
```

```
     EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.
* --  embedded SQL statements
     EXEC ORACLE OPTION (RELEASE_CURSOR=YES)END-EXEC.
* --  second embedded PL/SQL block
     EXEC ORACLE OPTION (RELEASE_CURSOR=NO)END-EXEC.
* --  embedded SQL statements
```

詳細は、D-12 ページの「埋込み PL/SQL に関する考慮事項」を参照してください。

ストアド PL/SQL と 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、別々にコンパイルし、データベースに格納して、起動することができます。

SQL*Plus などの Oracle ツールを使って明示的に作成したサブプログラムを、ストアド・サブプログラムといいます。コンパイルされ、データ・ディクショナリに格納されたストアド・サブプログラムは、データベース・オブジェクトとなり、再コンパイルせずに再実行できます。

PL/SQL ブロック内のサブプログラムまたはストアド・サブプログラムは、アプリケーションによってデータベースに送られ、インライン・サブプログラムになります。Pro*COBOL 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、ソース・コードまたはオブジェクト・コードはデータ・ディクショナリには格納しません。

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見なされ、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。

ストアド・サブプログラムの作成

次の例に示すように、SQL 文 CREATE FUNCTION、CREATE PROCEDURE および CREATE PACKAGE を COBOL 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 ではない）で始まります。しかし、PL/SQL 終結子 END-EXEC で終了する点が他の CREATE 埋込み文と異なります。

次の例では、*emp* 表から行のバッチを取り出す、*get_employees* という名前の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によって決まります。コール元は、別のストアド・サブプログラムである場合も、クライアント・アプリケーション・プログラムである場合もあります。

プロシージャは 3 つの PL/SQL 表を OUT 仮パラメータとして宣言し、その後従業員データのバッチを 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表です。このプロシージャは、終了すると、PL/SQL 表のすべての行の値をホスト表の対応する要素に自動的に割り当てます。

```
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 句を指定することによって、既存のパッケージを再定義できます。パッケージを削除して再作成し、権限を付与し直す必要はありません。CREATE 文の構文の詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

埋込み CREATE {FUNCTION | PROCEDURE | PACKAGE} 文が失敗した場合、Oracle8*i* はエラーではなく警告を発行します。

ストアド PL/SQL または Java サブプログラムのコール

ホスト・プログラムからストアド・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 文のどちらかを使用できます。

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* はホスト変数です。

次の例では、プロシージャ *raise_salary* が *emp_actions* という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しなければなりません。

```
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、ホスト変数、ホスト表、PL/SQL 定数、PL/SQL 変数、PL/SQL 表、PL/SQL ユーザー定義レコード、サブプログラム・コール、式を使用できます。これに対し、OUT 実パラメータには、リテラル、サブプログラム・コール、式は使用できません。

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 を使わなければなりません。

サンプル・プログラム 9: ストアド・プロシージャのコール

このサンプル・プログラムを実行する前に、次に示す CALLDEMO.SQL というスクリプトを実行して、*calldemo* という名前の PL/SQL パッケージを作成する必要があります。このスクリプトは Pro*COBOL に付属のスクリプトで、Pro*COBOL デモ・ライブラリに入っています。このスクリプトの正確なつづりについては、使用しているシステム固有の Oracle マニュアルを参照してください。

```
CREATE OR REPLACE PACKAGE calldemo AS

   TYPE name_array IS TABLE OF emp.ename%type
       INDEX BY BINARY_INTEGER;
   TYPE job_array IS TABLE OF emp.job%type
       INDEX BY BINARY_INTEGER;
   TYPE sal_array IS TABLE OF emp.sal%type
       INDEX BY BINARY_INTEGER;

   PROCEDURE get_employees(
     dept_number IN     number,    -- department to query
     batch_size  IN     INTEGER,   -- rows at a time
     found       IN OUT INTEGER,   -- rows actually returned
     done_fetch  OUT    INTEGER,   -- all done flag
     emp_name    OUT    name_array,
     job         OUT    job_array,
     sal         OUT    sal_array);

END calldemo;
/

CREATE OR REPLACE PACKAGE BODY calldemo AS

   CURSOR get_emp (dept_number IN number) IS
       SELECT ename, job, sal FROM emp
           WHERE deptno = dept_number;
```

```
    -- Procedure "get_employees" fetches a batch of employee
    -- rows (batch size is determined by the client/caller
    -- of the procedure).  It can be called from other
    -- stored procedures or client application programs.
    -- The procedure opens the cursor if it is not
    -- already open, fetches a batch of rows, and
    -- returns the number of rows actually retrieved. At
    -- end of fetch, the procedure closes the cursor.

    PROCEDURE get_employees(
      dept_number IN     number,
      batch_size  IN     INTEGER,
      found       IN OUT INTEGER,
      done_fetch  OUT    INTEGER,
      emp_name    OUT    name_array,
      job         OUT    job_array,
      sal         OUT    sal_array) IS

    BEGIN
        IF NOT get_emp%ISOPEN THEN      -- open the cursor if
            OPEN get_emp(dept_number);  -- not already open
        END IF;

        -- Fetch up to "batch_size" rows into PL/SQL table,
        -- tallying rows found as they are retrieved. When all
        -- rows have been fetched, close the cursor and exit
        -- the loop, returning only the last set of rows found.

        done_fetch := 0;  -- set the done flag FALSE
        found := 0;

        FOR i IN 1..batch_size LOOP
            FETCH get_emp INTO emp_name(i), job(i), sal(i);
            IF get_emp%NOTFOUND THEN    -- if no row was found
                CLOSE get_emp;
                done_fetch := 1;   -- indicate all done
                EXIT;
            ELSE
                found := found + 1;  -- count row
            END IF;
        END LOOP;
    END;
END;
/
```

次のサンプル・プログラムは、データベースに接続し、ユーザーに部門番号の入力を求め、*get_employees* という名前の PL/SQL プロシージャをコールします。この PL/SQL プロシージャはパッケージ *calldemo* に格納されています。プロシージャは 3 つの PL/SQL 表を OUT 仮パラメータとして宣言し、その後従業員データのバッチを 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表です。このプロシージャが終了すると、PL/SQL 表の行の値がホスト表の対応する要素に自動的に割り当てられます。プログラムはデータがなくなるまで繰り返しこのプロシージャをコールし、従業員データをバッチ単位で表示します。

```
*****************************************************************
* Sample Program 9: Calling a Stored Procedure
*
* This program connects to ORACLE, prompts the user for a
* department number, then calls a PL/SQL stored procedure named
* GET_EMPLOYEES, which is stored in package CALLDEMO.  The
* procedure declares three PL/SQL tables ast OUT formal
* parameters, then fetches a batch of employee data into the
* PL/SQL tables.  The matching actual parameters are host tables.
* When the procedure finishes, it automatically assigns all row
* values in the PL/SQL tables to corresponding elements in the
* host tables.  The program calls the procedure repeatedly,
* displaying each batch of employee data, until no more data
* is found.
* Use option picx=varchar2 when precompiling this sample program.
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. CALL-STORED-PROC.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME         PIC X(15) VARYING.
 01  PASSWD           PIC X(15) VARYING.
 01  DEPT-NUM         PIC S9(9) COMP.
 01  EMP-TABLES.
     05  EMP-NAME     OCCURS 10 TIMES PIC X(10).
     05  JOB-TITLE    OCCURS 10 TIMES PIC X(10).

     05  SALARY       OCCURS 10 TIMES COMP-2.

 01  DONE-FLAG        PIC S9(9) COMP.
 01  TABLE-SIZE       PIC S9(9) COMP VALUE 10.
 01  NUM-RET          PIC S9(9) COMP.
 01  SQLCODE          PIC S9(9) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

```
01  COUNTER          PIC S9(9) COMP.
01  DISPLAY-VARIABLES.
    05  D-EMP-NAME   PIC X(10).
    05  D-JOB-TITLE  PIC X(10).

    05  D-SALARY     PIC Z(5)9.

    05  D-DEPT-NUM   PIC 9(2).

    EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL WHENEVER SQLERROR DO
        PERFORM SQL-ERROR END-EXEC.

    PERFORM LOGON.
    PERFORM INIT-TABLES VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > 10.
    PERFORM GET-DEPT-NUM.
    PERFORM DISPLAY-HEADER.
    MOVE ZERO TO DONE-FLAG.
    MOVE ZERO TO NUM-RET.
    PERFORM FETCH-BATCH UNTIL DONE-FLAG = 1.
    PERFORM LOGOFF.

INIT-TABLES.
    MOVE SPACE TO EMP-NAME(COUNTER).
    MOVE SPACE TO JOB-TITLE(COUNTER).
    MOVE ZERO TO SALARY(COUNTER).

GET-DEPT-NUM.
    MOVE ZERO TO DEPT-NUM.
    DISPLAY " ".
    DISPLAY "ENTER DEPARTMENT NUMBER: "
        WITH NO ADVANCING.

    ACCEPT D-DEPT-NUM.

    MOVE D-DEPT-NUM TO DEPT-NUM.

DISPLAY-HEADER.
    DISPLAY " ".
    DISPLAY "EMPLOYEE    JOB TITLE    SALARY".
    DISPLAY "--------    ---------    ------".
```

```
FETCH-BATCH.
    EXEC SQL EXECUTE
        BEGIN
            CALLDEMO.GET_EMPLOYEES
                (:DEPT-NUM, :TABLE-SIZE,
                 :NUM-RET,  :DONE-FLAG,
                 :EMP-NAME, :JOB-TITLE, :SALARY);
        END;
    END-EXEC.
    PERFORM PRINT-ROWS VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > NUM-RET.

PRINT-ROWS.
    MOVE EMP-NAME(COUNTER) TO D-EMP-NAME.
    MOVE JOB-TITLE(COUNTER) TO D-JOB-TITLE.
    MOVE SALARY(COUNTER) TO D-SALARY.
    DISPLAY D-EMP-NAME, "  ",
            D-JOB-TITLE, "   ",
            D-SALARY.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

LOGOFF.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    STOP RUN.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    STOP RUN.
```

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なければなりません。また、ストアド・サブプログラムが終了する前にすべての OUT 仮パラメータに値を割り当てなければなりません。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。

リモート・アクセス

PL/SQL を使うと、データベース・リンクを経由してリモート・データベースにアクセスできます。通常、データベース・リンクは、DBA が作成し、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、データベースの位置、そのデータベースへのパス、および使用するユーザー名とパスワードをプログラムに示します。次の例では、データベース・リンク *dallas* を使って、*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提供できます。

```
CREATE PUBLIC SYNONYM raise_salary FOR raise_salary@dallas;
```

CALL 文

上記に示した埋込み PL/SQL ブロックに関する概念は、CALL 文にも当てはまります。CALL 埋込み PL/SQL 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]]
END-EXEC.
```

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 文とともに SQLCHECK=SYNTAX または SQLCHECK=SEMANTICS のどちらかを使用できます。

CALL の例

次に示すように、入力として整数を受け取り、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）を作成しておきます。

```
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.
```

Pro*COBOL アプリケーションで fact を使うには、次のように指定します。

```
...
        01 N      PIC S9(4) COMP.
        01 FACT   PIC S9(9) COMP.
...
    EXEC SQL CALL mathpkge.fact(:N) INTO :FACT END-EXEC.
...
```

CALL 文については、F-13 ページの「CALL（実行可能な埋込み SQL）」を参照してください。引数の引渡しなど、詳細については、『Oracle8*i* アプリケーション開発者ガイド 基礎編』の「外部ルーチン」の章を参照してください。

ストアド・サブプログラムに関する情報を得る方法

OCI コールをホスト・プログラムに埋め込む方法については、第 4 章の「データ型とホスト変数」で説明しました。ライブラリ・ルーチン SQLLDA をコールして LDA を設定した後、OCI コール ODESSP を使ってストアド・サブプログラムに関する有用な情報を得ることができます。ODESSP をコールする際には、有効な LDA および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・サブプログラムについては、パッケージの名前も渡す必要があります。ODESSP 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）を戻します。詳細は、『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』を参照してください。

Oracle8*i* に備わっているパッケージ DBMS_DESCRIBE の *describe_procedure* プロシージャも使用できます。詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

動的 PL/SQL の使用方法

Pro*COBOL は PL/SQL ブロック全体を 1 つの SQL 文のように扱うことを思い出してください。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。その場合、ブロックにホスト変数が含まれていなければ、動的 SQL 方法 1 を使って PL/SQL 文字列を実行できます。ブロックにホスト変数が含まれていて、その数がわかっている場合は、動的 SQL 方法 2 を使って PL/SQL 文字列を準備し、実行できます。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 を使う必要があります。詳細は、第 9 章の「Oracle 動的 SQL」および第 10 章の「ANSI 動的 SQL」、第 11 章の「Oracle 動的 SQL: 方法 4」を参照してください。

サブプログラムの制限事項

動的 SQL 方法 4 では、型が "table" のパラメータを指定してホスト表を PL/SQL プロシージャにバインドすることはできません。

カーソル変数

Pro*COBOL プログラムでカーソル変数を使用して、静的埋込み SQL によって複数行の問合せを処理できます。カーソル変数は、PL/SQL によってデータベース・サーバーで定義およびオープンされるカーソル参照を示します。カーソル変数の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

カーソル変数は、カーソルと同じように、複数行の問合せのアクティブ・セットの中の現在行を指します。カーソルとカーソル変数との違いは、定数と変数との違いと同じです。カーソルが静的であるのに対して、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。

カーソル変数に新しい値を割り当てて、サブプログラム（データベースに格納されているサブプログラムなど）にパラメータとして渡すことができます。これにより、データ検索の集中化が容易になります。

まず、カーソル変数を宣言します。カーソル変数を宣言したら、次の文を使ってカーソル変数を制御します。

- ALLOCATE
- OPEN...FOR

- FETCH
- CLOSE
- FREE

カーソル変数を宣言してメモリーを割り当てた後、そのカーソル変数を入力ホスト変数（バインド変数）として PL/SQL に渡します。次に、サーバー側で OPEN FOR を使って複数行の問合せ用にオープンし、クライアント側で FETCH してから、サーバー側かクライアント側のどちらかで CLOSE します。

カーソル変数の利点は、次のとおりです。

- メンテナンスの手軽さ。カーソル変数をオープンするストアド・プロシージャ内に問合せを集中できます。カーソルを変更する必要がある場合、変更しなければならないのは、ストアド・プロシージャだけです。アプリケーションごとに変更する必要はありません。
- セキュリティ。アプリケーションのユーザー（Pro*COBOL アプリケーションがデータベースに接続されている場合はユーザー名）は、カーソルをオープンするストアド・プロシージャの実行権限が必要です。また、問合せで使用される表の読込み権限は必要ありません。このセキュリティ機能を使って、表の列へのアクセスを制限できます。

カーソル変数の宣言

Pro*COBOL カーソル変数は SQL-CURSOR 疑似型を使って宣言します。たとえば、次のとおりです。

```
WORKING-STORAGE SECTION.
     ...
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         ...
         01  CUR-VAR  SQL-CURSOR.
         ...
     EXEC SQL END DECLARE SECTION END-EXEC.
```

SQL-CURSOR 変数は、Pro*COBOL が生成するコードに COBOL グループ項目としてインプリメントされます。カーソル変数は他の Pro*COBOL ホスト変数とまったく同じです。

カーソル変数の割当て

カーソル変数を OPEN するとき、またはその後で情報を FETCH するときには、Pro*COBOL の ALLOCATE コマンドを使ってカーソル変数を初期化する必要があります。たとえば、前の項で宣言したカーソル変数 CUR-VAR を初期化するには、次の文を使用します。

```
EXEC SQL ALLOCATE :CUR-VAR END-EXEC.
```

カーソル変数の割当てには、プリコンパイル時も実行時もサーバーをコールする必要はありません。

ALLOCATE 文では AT 句は使用できません。

警告 : 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。したがって、プログラム・ループではカーソル変数を割り当てないようにしてください。

カーソル変数のオープン

データベース・サーバーでカーソル変数をオープンするには、埋込み無名 PL/SQL ブロックを使う必要があります。無名 PL/SQL ブロックは、カーソルをオープンする（また、同じ文でカーソルを定義する）PL/SQL ストアド・プロシージャをコールすることにより間接的にカーソルをオープンするか、Pro*COBOL プログラムから直接カーソルをオープンします。

PL/SQL ストアド・プロシージャによる間接的なオープン

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 curtype,
               dept_num IN     number);
END;

CREATE PACKAGE BODY demo_cur_pkg AS
    CREATE PROCEDURE open_emp_cur (
               curs     IN OUT curtype,
               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、Pro*COBOL 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変数 EMP-CURSOR から FETCH することができます。たとえば、次のとおりです。

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-CURSOR      SQL-CURSOR.
 01  DEPT-NUM       PIC S9(4).
 01  EMP-NAME       PIC X(10) VARYING.
```

```
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...

 PROCEDURE DIVISION.
     ...
*    Allocate the cursor variable.
     EXEC SQL
         ALLOCATE :EMP-CURSOR
     END-EXEC.
     ...
     MOVE 30 TO DEPT_NUM.
*    Open the cursor on the Oracle Server.
     EXEC SQL EXECUTE
         BEGIN
             demo_cur_pkg.open_emp_cur(:EMP-CURSOR, :DEPT-NUM);
         END;
     END-EXEC.
     EXEC SQL
         WHENEVER NOT FOUND DO PERFORM SIGN-OFF
     END-EXEC.
 FETCH-LOOP.
     EXEC SQL
         FETCH :EMP-CURSOR INTO :EMP-NAME
     END-EXEC.
     DISPLAY "Employee Name: ",:EMP-NAME.
     GO TO FETCH-LOOP.
     ...
 SIGN-OFF.
     ...
```

Pro*COBOL アプリケーションからの直接的なオープン

Pro*COBOL プログラム内で PL/SQL 無名ブロックを使ってカーソルをオープンするには、無名ブロックの中でカーソルを定義します。次の例を考えてみます。

```
 PROCEDURE DIVISION.
     ...
     EXEC SQL EXECUTE
          BEGIN
             OPEN :EMP-CURSOR FOR SELECT ENAME FROM EMP
                 WHERE deptno = :DEPT-NUM;
         end;
     END-EXEC.
     ...
```

カーソル変数からのフェッチ

複数行の問合せ用にカーソル変数をオープンした後、FETCH 文を使って、アクティブ・セットから一度に 1 行ずつ行を取り出します。構文は次のとおりです。

```
     EXEC SQL FETCH cursor_variable_name
         INTO {record_name | variable_name[, variable_name, ...]}
     END-EXEC.
```

カーソル変数が戻す各列値は、データ型に互換性がある場合、INTO 句内の対応するフィールドまたは変数に割り当てられます。

FETCH 文はクライアント側で実行する必要があります。次の例では、*EMP-REC* という名前のホスト・レコードに行をフェッチします。

```
* -- exit loop when done fetching
     EXEC SQL
         WHENEVER NOT FOUND DO PERFORM NO-MORE
     END-EXEC.
     PERFORM
* -- fetch row into record
     EXEC SQL FETCH :EMP-CUR INTO :EMP-REC END-EXEC
* -- test for transfer out of loop
     ...
* -- process the data
     ...
     END-PERFORM.
     ...
 NO-MORE.
     ...
```

カーソル変数をオープンしたときに選択した行を取り出すには、埋込み SQL の FETCH.... INTO コマンドを使います。たとえば、次のとおりです。

```
 EXEC SQL
     FETCH :EMP-CURSOR INTO :EMP-INFO:EMP-INFO-IND
 END-EXEC.
```

カーソル変数から FETCH するには、そのカーソル変数を初期化し、オープンしておく必要があります。オープンされていないカーソル変数から FETCH することはできません。

カーソル変数のクローズ

カーソル変数をクローズするには、埋込み SQL の CLOSE 文を使います。その時点で、アクティブ・セットは未定義になります。構文は次のとおりです。

```
     EXEC SQL CLOSE cursor_variable_name END-EXEC.
```

CLOSE 文はクライアント側でもサーバー側でも実行できます。次の例では、最後の行が処理されたときにカーソル変数 *CUR-VAR* をクローズします。

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*    Declare the cursor variable.
         01  CUR-VAR       SQL-CURSOR.
         ...
     EXEC SQL END DECLARE SECTION END-EXEC.

 PROCEDURE DIVISION.
*    Allocate and open the cursor variable, then
*    Fetch one or more rows.
     ...
*    Close the cursor variable.
     EXEC SQL
         CLOSE :CUR-VAR
     END-EXEC.
```

カーソル変数の解放

カーソル変数 *CUR-VAR* に割り当てられたメモリーを解放するには、CLOSE の後で FREE 文を使います。

```
*    Free the cursor variable memory.
     EXEC SQL
         FREE :CUR-VAR
     END-EXEC.
```

カーソル変数の制限

カーソル変数の使用には、次の制限が適用されます。

- カーソル変数は、動的 SQL ではサポートされません。
- カーソル変数は、ALLOCATE、FETCH および CLOSE コマンドだけで使用できます。DECLARE CURSOR コマンドは、カーソル変数には適用されません。
- CLOSE_ON_COMMIT=NO を指定してプリコンパイルする場合、すでにクローズ済みのカーソル変数をクローズするとエラーになります。
- ALLOCATE コマンドで AT 句を使用することはできません。

エラー条件

次のような操作は行わないでください。

- クローズされたカーソル変数や割り当てられていないカーソル変数からのフェッチ
- 割り当てられていないカーソル変数の使用
- オープンされていないカーソル変数の CLOSE

カーソル変数に対してこれらの操作を行うと、エラーが発生します。

サンプル・プログラム 11: カーソル変数

次のサンプル・プログラム、SQL スクリプト（SAMPLE11.SQL）と Pro*COBOL プログラム（SAMPLE11.PCO）では、Pro*COBOL におけるカーソル変数の使用方法を示します。

SAMPLE11.SQL

このサンプル・プログラムは、カーソル変数を宣言してオープンするパッケージを作成するための PL/SQL ソース・コードです。

```
CONNECT SCOTT/TIGER
CREATE OR REPLACE PACKAGE emp_demo_pkg AS
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_cur (
        cursor   IN OUT emp_cur_type,
        dept_num IN     number);
END emp_demo_pkg;
/
CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS

    PROCEDURE open_cur (
        cursor   IN OUT emp_cur_type,
        dept_num IN     number) IS
    BEGIN
        OPEN cursor FOR SELECT * FROM emp
        WHERE deptno = dept_num
        ORDER BY ename ASC;
    END;
END emp_demo_pkg;
/
```

SAMPLE11.PCO

次に示すのは、前記の SAMPLE11.SQL で宣言したカーソルを使って、EMP 表から従業員名、給料、歩合給をフェッチする Pro*COBOL サンプル・プログラム SAMPLE11.PCO です。

```
*****************************************************************
* Sample Program 11:  Cursor Variable Operations                *
*                                                               *
* This program logs on to ORACLE, allocates and opens a cursor  *
* variable fetches the names, salaries, and commissions of all  *
* salespeople, displays the results, then closes the cursor.    *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. CURSOR-VARIABLES.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME          PIC X(15) VARYING.
 01  PASSWD            PIC X(15) VARYING.
 01  HOST              PIC X(15) VARYING.
 01  EMP-CUR           SQL-CURSOR.
 01  EMP-INFO.
     05  EMP-NUM       PIC S9(4) COMP.
     05  EMP-NAM       PIC X(10) VARYING.
     05  EMP-JOB       PIC X(10) VARYING.
     05  EMP-MGR       PIC S9(4) COMP.
     05  EMP-DAT       PIC X(10) VARYING.
     05  EMP-SAL       PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05  EMP-COM       PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
     05  EMP-DEP       PIC S9(4) COMP.
 01  EMP-INFO-IND.
     05  EMP-NUM-IND   PIC S9(4) COMP.
     05  EMP-NAM-IND   PIC S9(4) COMP.
     05  EMP-JOB-IND   PIC S9(4) COMP.
     05  EMP-MGR-IND   PIC S9(4) COMP.
     05  EMP-DAT-IND   PIC S9(4) COMP.
     05  EMP-SAL-IND   PIC S9(4) COMP.
     05  EMP-COM-IND   PIC S9(4) COMP.
     05  EMP-DEP-IND   PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
```

```
    EXEC SQL INCLUDE SQLCA END-EXEC.

01  DISPLAY-VARIABLES.
    05  D-DEP-NUM     PIC Z(3)9.
    05  D-EMP-NAM     PIC X(10).
    05  D-EMP-SAL     PIC Z(4)9.99.
    05  D-EMP-COM     PIC Z(4)9.99.
    05  D-EMP-DEP     PIC 9(2).


PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
    EXEC SQL
        ALLOCATE :EMP-CUR
    END-EXEC.
    DISPLAY "Enter department number (0 to exit):  "
        WITH NO ADVANCING.
    ACCEPT D-EMP-DEP.
    MOVE D-EMP-DEP TO EMP-DEP.
    IF EMP-DEP <= 0
        GO TO SIGN-OFF
    END-IF.
    MOVE EMP-DEP TO D-DEP-NUM.
    EXEC SQL EXECUTE
        BEGIN
            emp_demo_pkg.open_cur(:EMP-CUR, :EMP-DEP);
        END;
    END-EXEC.
    DISPLAY " ".
    DISPLAY "For department ", D-DEP-NUM, ":".
    DISPLAY " ".
    DISPLAY "EMPLOYEE   SALARY     COMMISSION".
    DISPLAY "---------- ---------- ----------".

FETCH-LOOP.
    EXEC SQL
        WHENEVER NOT FOUND GOTO CLOSE-UP
    END-EXEC.
    MOVE SPACES TO EMP-NAM-ARR.
    EXEC SQL FETCH :EMP-CUR
        INTO :EMP-NUM:EMP-NUM-IND,
             :EMP-NAM:EMP-NAM-IND,
```

```
            :EMP-JOB:EMP-JOB-IND,
            :EMP-MGR:EMP-MGR-IND,
            :EMP-DAT:EMP-DAT-IND,
            :EMP-SAL:EMP-SAL-IND,
            :EMP-COM:EMP-COM-IND,
            :EMP-DEP:EMP-DEP-IND
    END-EXEC.
    MOVE EMP-SAL TO D-EMP-SAL.
    IF EMP-COM-IND = 0
        MOVE EMP-COM TO D-EMP-COM
        DISPLAY EMP-NAM-ARR, "   ", D-EMP-SAL,
                "   ", D-EMP-COM
    ELSE
        DISPLAY EMP-NAM-ARR, "   ", D-EMP-SAL,
                "        N/A"
    END-IF.
    GO TO FETCH-LOOP.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    MOVE "INST1_ALIAS" TO HOST-ARR.
    MOVE 11 TO HOST-LEN.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME-ARR.

CLOSE-UP.
    EXEC SQL
        CLOSE :EMP-CUR
    END-EXEC.
    EXEC SQL
        FREE :EMP-CUR
    END-EXEC.
SIGN-OFF.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    STOP RUN.
```

```
SQL-ERROR.
    EXEC SQL
        WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.
    EXEC SQL
        ROLLBACK WORK RELEASE
    END-EXEC.
    STOP RUN.
```

7

ホスト表

この章では、表を使用してコーディングを単純化し、プログラムのパフォーマンスを向上させる方法を説明します。ここで紹介するのは、ホスト表を使って Oracle データを操作する方法、単一の SQL 文を使ってホスト表内のすべての要素を処理する方法、処理対象となる表の要素数を制限する方法、およびグループ項目の表の使用方法です。

この章の構成は、次のとおりです。

ホスト表とは

ホスト表とは、関連するデータ項目（これを要素と呼びます）の集まりを 1 つの変数名に関連付けたものです。標識変数を表として定義した場合、その標識変数を標識表と呼びます。標識表は、NULLABLE である任意のホスト表に関連付けできます。

表を使用する利点

表を使用することにより、プログラミングが容易になり、パフォーマンスが改善されます。アプリケーションを作成する際には、通常、大量のデータ集合の格納と操作という問題が出てきます。表を使用することで、各データ集合の個々の項目の命名および参照の作業を単純化できます。

表を使うと、単一の SQL 文でデータの集合全体を操作できます。このため、特にネットワーク化された環境では、通信オーバーヘッドが著しく低減されます。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。表を使わなければ、プログラムでは従業員ごとに 1 回づつ、合計 300 回 INSERT を実行しなければなりません。表を使用すれば、INSERT を 1 回実行するだけで済みます。

スカラー・ホスト変数を使用できる場合は、ほとんど例外なくホスト表を使用できます。また、ホスト表に標識表を関連付けることもできます。

ホスト表の宣言

ホスト表の宣言とディメンション指定はデータ部で行います。次の例では、3 つのホスト表を宣言し、各表のディメンションは 50 要素に指定されます。

```
     ....
 01  EMP-TABLES.
     05  EMP-NUMBER  OCCURS 50 TIMES PIC S9(4) COMP.
     05  EMP-NAME    OCCURS 50 TIMES PIC X(10.
     05  SALARY      OCCURS 50 TIMES PIC S9(5)V99 COMP-3.
     ....
```

次の例に示すように、OCCURS 句で INDEXED BY 句を使うと、索引を指定できます。

```
     ...
 01  EMP-TABLES.
     05  EMP-NUMBER  PIC X(10) OCCURS 50 TIMES
         INDEXED BY EMP-INDX.
             ...
     ...
```

この INDEXED BY 句によって、索引項目 EMP-INDX が暗黙的に宣言されます。

制限

多次元のホスト表は作成できません。したがって、次の例で宣言されている、2 次元のホスト表は無効です。

```
     ...
 01  NATION.
     05  STATE                OCCURS 50 TIMES.
         10  STATE-NAME       PIC X(25).
         10  COUNTY           OCCURS 25 TIMES.
             15  COUNTY-NAME  PIX X(25).
     ...
```

また、可変長のホスト表も使用できません。たとえば、次に示す EMP-REC の宣言はホスト変数としては無効です。

```
     ...
 01  EMP-FILE.
     05  REC-COUNT  PIC S9(3) COMP.
     05  EMP-REC    OCCURS 0 TO 250 TIMES
         DEPENDING ON REC-COUNT.
     ...
```

1 回のフェッチでホスト表によってアクセスできるバイトの最大数は、使用するリソースによって決まります。最大の大きさを超えるホスト表を定義すると、" パラメータが範囲外 " というランタイム・エラーが表示されます。また、単一の SQL 文で複数のホスト表を使用する場合、各表のエントリ数は同じでなければなりません。エントリ数が異なっていると、" 表のサイズが一致しません " という警告メッセージがプリコンパイル時に発行されます。この警告を無視すると、プリコンパイラは SQL 操作で最小のエントリ数を使用します。

ホスト表の参照

また、単一の SQL 文で複数のホスト表を使用する場合は、各表のディメンションは同じでなければなりません。ただし、これは Pro*COBOL では常に最小のディメンションが SQL 操作に使用されるためであり、必要条件ではありません。次の例では、INSERT されるのは 25 行だけです。

```
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-TABLES.
     05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
     05  EMP-NAME     PIC X(10) OCCURS 50 TIMES.
     05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 25 TIMES.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 PROCEDURE DIVISION.
     ...
*    Populate host tables here.
     ...
```

```
    EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
        VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
    END-EXEC.
```

SQL 文ではホスト表に添字を付けることはできません。たとえば、次の INSERT 文は無効です。

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  EMP-TABLES.
    05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
    05  EMP-NAME     PIC X(10) OCCURS 50 TIMES.
    05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
    PERFORM LOAD-EMP VARYING J FROM 1 BY 1 UNTIL J > 50.
    ...
LOAD-EMP.
    EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
        VALUES (:EMP-NUMBER(J), :EMP-NAME(J),
            :DEPT-NUMBER(J))
    END-EXEC.
```

PERFORM VARYING 文でホスト表を処理する必要はありません。SQL 文では添字なしの表名を使ってください。Pro*COBOL では、ディメンション *n* のホスト表を使った SQL 文は、*n* 個の異なるスカラー・ホスト変数を使ってその SQL 文を *n* 回実行した場合と同じように扱われます。（ただし、ホスト表を使用する方が効率的です。）

標識表の使用方法

標識表を使用すると、入力ホスト表の要素に NULL を割り当てたり、出力ホスト表の NULL または切り捨てられた値（文字型の列のみ）を検出できます。次の例は、標識表を使って INSERT する方法を示したものです。

```
WORKING-STORAGE SECTION.
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  EMP-TABLES.
    05  EMP-NUMBER   PIC S9(4) COMP OCCURS 50 TIMES.
    05  DEPT-NUMBER  PIC S9(4) COMP OCCURS 50 TIMES.
    05  COMMISSION   PIC S9(5)V99 COMP-3 OCCURS 50 TIMES.
    05  COMM-IND     PIC S9(4) COMP OCCURS 50 TIMES.
    EXEC SQL END DECLARE SECTION END-EXEC.
    ...
PROCEDURE DIVISION.
    ...
*   Populate the host and indicator tables.
```

```
*    Set indicator table to all zeros.
     ...
     EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
         VALUES (:EMP-NUMBER, :DEPT-NUMBER,
             :COMMISSION:COMM-IND)
     END-EXEC.
```

標識表のディメンションは、ホスト表のディメンションと同じか、それより大きくなければなりません。

Oracle での制限

VALUES、SET、INTO または WHERE 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。ホスト変数のうちのどれか 1 つでも配列の場合は、すべてのホスト変数が配列でなければなりません。

UPDATE または DELETE 文の CURRENT OF 句では、ホスト配列は使用できません。

ANSI での制限と要件

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れば、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。

配列の SELECT および FETCH を実行するときは、必ず標識配列を使います。標識配列を使用すると、対応する出力ホスト配列に NULL があるかどうかをテストできます。

プリコンパイラ・オプション DBMS=V7 または V8 でプリコンパイルする場合、関連付けられた標識変数のないホスト変数に NULL が選択またはフェッチされると、Oracle は処理を停止し、処理した行数を *sqlca.sqlerrd(3)* に設定して、エラーを戻します。

DBMS=V7 または V8 のときは、切捨てはエラーとみなされません。

表を含んでいるホスト・グループ項目

注意 : 表を含むホスト・グループ項目がある場合は、ハーフワードの整変数の表を標識に使用できません。標識には、表の対応するグループ項目を使う必要があります。たとえば、次のようなグループ項目を考えます。

```
01  DEPARTURE.
    05 HOUR    PIC X(2) OCCURS 3 TIMES.
    05 MINUTE  PIC X(2) OCCURS 3 TIMES.
```

この場合、次の標識変数は使用できません。

```
01  DEPARTURE-IND PIC S9(4) COMP OCCURS 6 TIMES.
```

表のグループ項目に使用する標識変数は、次のように、それ自体が表のグループ項目でなければなりません。

```
01  DEPARTURE-IND.
    05 HOUR-IND   PIC S9(4) COMP OCCURS 3 TIMES.
    05 MINUTE-IND PIC S9(4) COMP OCCURS 3 TIMES.
```

データ操作文の表

Pro*COBOL では、データ操作文でホスト表を使用できます。ホスト表は、INSERT 文、UPDATE 文および DELETE 文の入力変数として、また、SELECT 文および FETCH 文の INTO 句の出力変数として使用できます。

ホスト表に使用する構文は、通常のホスト変数に使用する構文とほとんど同じです。唯一の違いは、オプションの FOR 句を使って表の処理を制御できることです。また、ホスト表と通常のホスト変数を 1 つの SQL 文で併用する場合には制限がいくつかあります。

この後の各項では、データ操作文でのホスト表の使用方法を説明します。

表に対する選択

ホスト表を SELECT 文の出力変数として使用できます。選択で戻される行の最大数がわかっている場合は、それと同じ数の要素数でホスト表を定義してください。次の例では、選択結果を直接 3 つのホスト表に入れます。選択で戻される行の数が 50 以下であることがわかっているので、50 要素の表を定義します。

```
01  EMP-REC-TABLES.
    05  EMP-NUMBER   OCCURS 50 TIMES PIC S9(4) COMP.
    05  EMP-NAME     OCCURS 50 TIMES PIC X(10) VARYING.
    05  SALARY       OCCURS 50 TIMES PIC S9(6)V99
                     DISPLAY SIGN LEADING SEPARATE.
...
EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO :EMP-NAME, :EMP-NUMBER, :SALARY
    FROM EMP
    WHERE SAL > 1000
END-EXEC.
```

この例では SELECT 文は 50 行までの行を戻します。選択される行数が 50 行に満たないとき、または 50 行だけを取り出したいときはこの方法を使います。ただし、選択される行数が 50 行を超える場合は、この方法ではすべての行を取り出せません。この SELECT 文をもう 1 度実行しても、他に選択対象の行があるにもかかわらず最初の 50 行だけがまた戻されます。このような場合は、大きい表を定義するか、FETCH 文に使用するカーソルを宣言する必要があります。

SELECT_ERROR=NO と指定していない場合は、定義した表サイズよりも SELECT INTO 文で戻される行数が大きいと、Oracle8*i* はエラー・メッセージを発行します。オプションの詳細は、14-36 ページの「SELECT_ERROR」を参照してください。

バッチ・フェッチ

選択によって戻される行の最大数がわからない場合は、カーソルを宣言およびオープンして、そこからバッチ単位でフェッチすることができます。ループ内でバッチ・フェッチを実行すれば、多数の行を簡単に取り出せます。フェッチを行うたびに、現行のアクティブ・セットから次のバッチの行が戻されます。次の例では、20 行ずつまとめて行をフェッチします。

```
 ...
 01  EMP-REC-TABLES.
     05  EMP-NUMBER    OCCURS 20 TIMES PIC S9(4) COMP.
     05  EMP-NAME      OCCURS 20 TIMES PIC X(10) VARYING.
     05  SALARY        OCCURS 20 TIMES PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
     ...
     EXEC SQL DECLARE EMPCURSOR CURSOR FOR
     SELECT EMPNO, SAL FROM EMP
     END-EXEC.
     ...
     EXEC SQL OPEN EMPCURSOR END-EXEC.
     ...
     EXEC SQL WHENEVER NOT FOUND DO PERFORM END-IT.
 LOOP.
     EXEC SQL FETCH EMPCURSOR INTO :EMP-NUMBER, :SALARY END-EXEC.
* --  process batch of rows
     ...
     GO TO LOOP.
 END-IT.
...
```

最後のフェッチで実際に戻された行数を忘れずにチェックし、行を処理してください。

SQLERRD(3) の使用方法

INSERT 文、UPDATE 文および DELETE 文では、処理された行の数は SQLERRD(3) に格納されます。

SQLERRD(3) は、表の操作中にエラーが発生したときにも役に立ちます。処理はエラーを引き起こした行で停止するので、SQLERRD(3) を調べることによって正常に処理された行の数がわかります。

FETCH される行数

それぞれのフェッチで戻される行数は、最大でも表のエントリと同じ数です。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。" データがありません " という警告コードが SQLCA の SQLCODE に戻されます。たとえば、エントリ数が 100 の表に対してフェッチを行ったが 20 行しか戻らなかった場合です。
- フェッチ対象の行がバッチ行数よりも少ないとき。たとえば、エントリ数が 20 の表に対して 70 行をフェッチした場合、3 回目のフェッチの後ではフェッチ対象の行が 10 行しか残っていないため、このような状況が発生します。
- 行の処理中にエラーが検出されたとき。この場合、フェッチは失敗し、該当するエラー・コードが SQLCODE に戻されます。

戻された行の累計数は、SQLCA 内の SQLERRD の 3 番目の要素（このマニュアルでは SQLERRD(3) と呼びます）に格納されます。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 END-EXEC.
     EXEC SQL OPEN CURSOR2 END-EXEC.
     EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.
* --  now running total in SQLERRD(3) is 20
     EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.
* --  now running total in SQLERRD(3) is 30, not 50
     EXEC SQL FETCH CURSOR1 INTO :TABLE-OF-20 END-EXEC.
* --  now running total in SQLERRD(3) is 40 (20 + 20)
     EXEC SQL FETCH CURSOR2 INTO :TABLE-OF-30 END-EXEC.
* --  now running total in SQLERRD(3) is 60 (30 + 30)
```

ホスト表の使用制限

SELECT 文の WHERE 句でホスト表を使用できるのは、副問合せの場合だけです。（7-17 ページの「WHERE 句の使用方法」の例を参照してください。）また、SELECT 文や FETCH 文の INTO 句では、通常のホスト変数とホスト表は併用できません。ホスト変数のうちのどれか 1 つでも表の場合は、すべてのホスト変数が表でなければなりません。

表 7-1 に、SELECT INTO 文に有効なホスト表の使用方法を示します。

表 7-1 SELECT INTO 文に有効なホスト表

INTO 句	WHERE 句	有効 / 無効
表	表	無効
スカラー	スカラー	有効
表	スカラー	有効
スカラー	表	無効

NULL のフェッチ

UNSAFE_NULL=YES の場合は、標識表のないホスト表に対して NULL を選択またはフェッチしてもエラーが生成されません。このため、表の選択およびフェッチを行うときには、必ず標識表を使用してください。標識表を使用すると、対応する出力ホスト表に NULL があるかどうかを確認できます。（NULL および切捨て値の検出方法については、5-3 ページの「標識変数の使用方法」を参照してください。）

UNSAFE_NULL=NO の場合は、標識表のないホスト表に対して NULL を選択またはフェッチすると、Oracle8*i* は処理を停止し、処理した行数を SQLERRD(3) に設定して、エラー・メッセージを発行します。

切り捨てられた値のフェッチ

DBMS=V7 または V8 の場合は、標識表のないホスト表に対して切り捨てられた値を選択またはフェッチすると、Oracle8*i* は SQLWARN(2) を設定します。

SQLERRD(3) を調べると、切捨てが行われるまでに処理された行の数がわかります。処理済み行数には切捨てエラーが発生した行も含まれます。

表の選択およびフェッチを行うときには、必ず標識表を使用してください。標識表を使用すると、1 つ以上の切り捨てられた列値が出力ホスト表に割り当てられた場合に、対応する標識表を調べればその列値の元の長さがわかります。

サンプル・プログラム 3: バッチでのフェッチ

次のホスト表のサンプル・プログラムは、デモ・ディレクトリに入っています。

```
*****************************************************************
* Sample Program 3:  Host Tables                                *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches in batches using host tables, and prints the results. *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. HOST-TABLES.
```

```
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  USERNAME          PIC X(15) VARYING.
01  PASSWD            PIC X(15) VARYING.
01  EMP-REC-TABLES.
    05  EMP-NUMBER    OCCURS 5 TIMES PIC S9(4) COMP.
    05  EMP-NAME      OCCURS 5 TIMES PIC X(10) VARYING.
    05  SALARY        OCCURS 5 TIMES PIC S9(6)V99
                       DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
01  NUM-RET           PIC S9(9) COMP VALUE ZERO.
01  PRINT-NUM         PIC S9(9) COMP VALUE ZERO.
01  COUNTER           PIC S9(9) COMP.
01  DISPLAY-VARIABLES.
    05  D-EMP-NAME    PIC X(10).
    05  D-EMP-NUMBER  PIC 9(4).
    05  D-SALARY      PIC Z(4)9.99.

PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
    EXEC SQL
        DECLARE C1 CURSOR FOR
        SELECT EMPNO, SAL, ENAME
        FROM EMP
    END-EXEC.
    EXEC SQL
        OPEN C1
    END-EXEC.

FETCH-LOOP.
    EXEC SQL
        WHENEVER NOT FOUND DO PERFORM SIGN-OFF
    END-EXEC.
    EXEC SQL
        FETCH C1
        INTO :EMP-NUMBER, :SALARY, :EMP-NAME
    END-EXEC.
```

```
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
    PERFORM PRINT-IT.
    MOVE SQLERRD(3) TO NUM-RET.
    GO TO FETCH-LOOP.

LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
       CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.

PRINT-IT.
    DISPLAY " ".
    DISPLAY "EMPLOYEE NUMBER  SALARY   EMPLOYEE NAME".
    DISPLAY "---------------  -------  -------------".
    PERFORM PRINT-ROWS
        VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > PRINT-NUM.

PRINT-ROWS.
    MOVE EMP-NUMBER(COUNTER) TO D-EMP-NUMBER.
    MOVE SALARY(COUNTER) TO D-SALARY.
    DISPLAY "           ", D-EMP-NUMBER, " ", D-SALARY, "  ",
        EMP-NAME-ARR IN EMP-NAME(COUNTER).
    MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME(COUNTER).

SIGN-OFF.
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
    IF (PRINT-NUM > 0) PERFORM PRINT-IT.
    EXEC SQL
        CLOSE C1
    END-EXEC.
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    STOP RUN.

SQL-ERROR.
    EXEC SQL
```

```
    WHENEVER SQLERROR CONTINUE
END-EXEC.
DISPLAY " ".
DISPLAY "ORACLE ERROR DETECTED:".
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL
    ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
```

表に対する挿入

ホスト表は、INSERT 文の入力変数として使用できます。その場合、INSERT 文の実行前に表にデータが挿入されるようにプログラムを作成してください。表の中に不適切な要素がある場合は、FOR 句を使うことによって挿入する行数を制御できます。詳細は、7-15 ページの「FOR 句の使用方法」を参照してください。

次に示すのは、ホスト表に対する挿入の例です。

```
 01  EMP-REC-TABLES.
     05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
     05  EMP-NAME      OCCURS 50 TIMES PIC X(10) VARYING.
     05  SALARY        OCCURS 50 TIMES PIC S9(6)V99
                        DISPLAY SIGN LEADING SEPARATE.
* -- populate the host tables
     EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
         VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)
     END-EXEC.
```

挿入された行の累計数は、SQLERRD(3) を調べるとわかります。

SQL 文ではホスト表に添字を付けることはできません。たとえば、次の INSERT 文は無効です。

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = TABLE-DIMENSION.
   EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
       VALUES (:EMP-NAME(I), :EMP-NUMBER(I), :SALARY(I))
   END_EXEC
END-PERFORM.
```

ホスト表の使用制限

INSERT 文の VALUES 句では、通常のホスト変数とホスト表は併用できません。ホスト変数のうちのどれか 1 つでも表の場合は、すべてのホスト変数が表でなければなりません。

表に対する更新

次の例に示すように、UPDATE 文の入力変数としてもホスト表を使用できます。

```
 01  EMP-REC-TABLES.
     05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
     05  SALARY        OCCURS 50 TIMES PIC S9(6)V99
                            DISPLAY SIGN LEADING SEPARATE.
 ...
* --  populate the host tables
     EXEC SQL
         UPDATE EMP SET SAL = :SALARY WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
```

更新された行の累計数は、SQLERRD(3) を調べるとわかります。その累積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。

表の中に不適切な要素がある場合は、FOR 句を使うことによって更新する行数を制限できます。

上記の例は、一意キー（*EMP-NUMBER*）を使用した典型的な更新を示しています。この例では、表の各要素によって更新されるのは 1 行だけです。次の例では、表の 1 つの要素によって複数の行が更新されます。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
            05  JOB-TITLE      OCCURS 10 TIMES PIC X(10) VARYING.
            05  COMMISSION     OCCURS 50 TIMES PIC S9(6)V99
                              DISPLAY SIGN LEADING SEPARATE.
     EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
     EXEC SQL
         UPDATE EMP SET COMM = :COMMISSION WHERE JOB = :JOB-TITLE
     END-EXEC.
```

UPDATE での制限

UPDATE 文の SET 句や WHERE 句では、通常のホスト変数とホスト表は併用できません。ホスト変数のうちのどれか 1 つでも表の場合は、すべてのホスト変数が表でなければなりません。また、SET 句でホスト表を使用する場合は、WHERE 句でもホスト表を使用しなければなりません。エントリ数とデータ型は同じでなくてもかまいません。

UPDATE 文の CURRENT OF 句ではホスト表は使用できません。他の方法については、7-18 ページの「CURRENT OF 句の疑似実行」を参照してください。

表 7-2 に、UPDATE 文に有効なホスト表の使用方法を示します。

表 7-2 UPDATE 文に有効なホスト表

SET 句	WHERE 句	有効 / 無効
表	表	有効
スカラー	スカラー	有効
表	スカラー	無効
スカラー	表	無効

表に対する削除

ホスト表を DELETE 文の入力変数としても使用できます。これは、WHERE 句でそのホスト表の連続した要素を使用して DELETE 文を繰返し実行するのと同じことです。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。次に示すのは、ホスト表を使用した削除の例です。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
         05  EMP-NUMBER    OCCURS 50 TIMES PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host table
     EXEC SQL
         DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
```

削除された行の累計数は、SQLERRD(3) を調べるとわかります。この累計数には、削除カスケードで処理された行は含まれません。

上記の例は、一意キー（*EMP-NUMBER*）を使用した典型的な削除を示しています。この例では、表の各要素によって削除されるのは 1 行だけです。次の例では、表の 1 つの要素によって複数の行が削除されます。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
            05  JOB-TITLE    OCCURS 10 TIMES PIC X(10) VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host table
     EXEC SQL
        DELETE FROM EMP WHERE JOB = :JOB-TITLE
     END-EXEC.
```

DELETE での制限

DELETE 文の WHERE 句では、通常のホスト変数とホスト表は併用できません。ホスト変数のうちのどれか 1 つでも表の場合は、すべてのホスト変数が表でなければなりません。また、DELETE 文の CURRENT OF 句でもホスト表は使用できません。他の方法については、7-18 ページの「CURRENT OF 句の疑似実行」を参照してください。

標識表の使用方法

標識表は、ホスト表に NULL を割り当てる場合や、出力ホスト表の中の NULL または切り捨てられた値を検出するために使用します。次の例は、標識表を使用した挿入方法を示したものです。

```
 01  EMP-REC-VARS.
     05  EMP-NUMBER  OCCURS 50 TIMES PIC S9(4) COMP.
     05  DEPT-NUMBER OCCURS 50 TIMES PIC S9(4) COMP.
     05  COMMISSION  OCCURS 50 TIMES  PIC S9(6)V99
                           DISPLAY SIGN LEADING SEPARATE.
* -- indicator table:
     05  COMM-IND    OCCURS 50 TIMES  PIC S9(4) COMP.
* --  populate the host tables
* --  populate the indicator table; to insert a NULL into
* --  the COMM column, assign -1 to the appropriate element in
* --  the indicator table
     EXEC SQL
         INSERT INTO EMP (EMPNO, DEPTNO, COMM)
         VALUES (:EMP_NUMBER, :DEPT-NUMBER, :COMMISSION:COMM-IND)
     END-EXEC.
```

標識表のエントリ数をホスト表のエントリ数より少なくすることはできません。

FOR 句の使用方法

オプションの FOR 句を使用すると、次の SQL 文で処理する表要素の数を設定できます。

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

特に UPDATE 文、INSERT 文および DELETE 文内で FOR 句を使うと便利です。これらの文では、表全体を使用する必要がない場合があります。次の例に示すように、FOR 句を使うと、使用する要素数を任意の数に制限できます。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-REC-VARS.
     05  EMP-NAME  OCCURS 1000 TIMES  PIC X(20) VARYING.
     05  SALARY    OCCURS 100  TIMES  PIC S9(6)V99
                   DISPLAY SIGN LEADING SEPARATE.
 01 ROWS-TO-INSERT PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.
* --  populate the host tables
     MOVE 25 TO ROWS-TO-INSERT.
* -- set FOR-clause variable
* -- will process only 25 rows
     EXEC SQL FOR :ROWS-TO-INSERT
         INSERT INTO EMP (ENAME, SAL)
         VALUES (:EMP-NAME, :SALARY)
     END-EXEC.
```

FOR 句では、表の要素数の指定には整数のホスト変数を使用しなければなりません。たとえば、次の文は無効です。

```
* -- illegal
     EXEC SQL FOR 25
     INSERT INTO EMP (ENAME, EMPNO, SAL)
         VALUES (:EMP-NAME, :EMP-NUMBER, :SALARY)
     END-EXEC.
```

FOR 句の変数によって、処理する表要素の数を指定します。その数が、表の最小ディメンションよりも大きくならないようにしてください。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って負の値を渡すと、予測不能の動作が起こります。

制限

FOR 句のセマンティックスを明確にするための制限が 2 つあります。FOR 句は、SELECT 文の中でまたは CURRENT OF 句とともには使用できません。

SELECT 文中

SELECT 文で FOR 句を使うと、エラー・メッセージが戻されます。

FOR 句は意味があいまいであるため、SELECT 文では使えません。この SELECT 文を *n* 回実行するのか、この SELECT 文を 1 回実行して *n* 行戻すのかがはっきりしません。前者の場合は、SELECT 文を実行するたびに複数行が戻される可能性があります。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 句を使った方がよいでしょう。

```
EXEC SQL FOR :LIMIT FETCH EMPCURSOR INTO ...
```

CURRENT OF 句との併用

次の例に示すように、UPDATE 文または DELETE 文で CURRENT OF 句を使うと、FETCH 文によって戻される最後の行を参照できます。

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
    SELECT ENAME, SAL FROM EMP WHERE EMPNO = :EMP-NUMBER
END-EXEC.
...
EXEC SQL OPEN EMPCURSOR END-EXEC.
...
EXEC SQL FETCH emp_cursor INTO :EM-NAME, :SALARY END-EXEC.
...
EXEC SQL UPDATE EMP SET SAL = :NEW-SALARY
    WHERE CURRENT OF EMPCURSOR
END-EXEC.
```

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（現在行の更新または削除は 1 回しかできない）ため無効です。

```
EXEC SQL FOR :LIMIT UPDA-CURSOR END-EXEC.
...
EXEC SQL FOR :LIMIT DELETE FROM EMP
    WHERE CURRENT OF EMP-CURSOR
END-EXEC.
```

WHERE 句の使用方法

Pro*COBOL では、エントリ数 *n* のホスト表を使った SQL 文は、*n* 個の異なるスカラー変数（表の個々の要素）を使ってその SQL 文を *n* 回実行した場合と同じように扱われます。文意が不明瞭なためにこのような処理ができない場合だけ、プリコンパイラはエラー・メッセージを発行します。

たとえば次の宣言をしたとき、

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
    05  MGRP-NUMBER  OCCURS 50 TIMES  PIC S9(4) COMP.
    05  JOB-TITLE    OCCURS 50 TIMES  PIC X(20) VARYING.
01  I PIC S9(4) COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.
```

次の文を、

```
EXEC SQL SELECT MGR INTO :MGR-NUMBER FROM EMP
    WHERE JOB = :JOB-TITLE
```

```
END-EXEC.
```

次の架空の文のように処理した場合には、不明瞭となります。

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50
SELECT MGR INTO :MGR-NUMBER(I) FROM EMP
    WHERE JOB = :JOB_TITLE(I)
END-EXEC
END-PERFORM.
```

WHERE 句の検索条件を満たす行が複数あっても、データの受取りに使える出力変数が 1 つしかないためです。したがって、エラー・メッセージが発行されます。

一方、次の文を

```
 EXEC SQL
    UPDATE EMP SET MGR = :MGR_NUMBER
    WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE
    JOB = :JOB-TITLE)
END-EXEC.
```

次の架空の文のように処理した場合には、不明瞭にはなりません。

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 50
     UPDATE EMP SET MGR = :MGR_NUMBER(I)
        WHERE EMPNO IN
        (SELECT EMPNO FROM EMP WHERE JOB = :JOB-TITLE(I))
    END-EXEC
END-PERFORM.
```

これは、WHERE 句の各 *JOB-TITLE* に該当する行が複数ある場合でも、*JOB-TITLE* に該当する行のそれぞれに対して SET 句内に *MGR-NUMBER* が存在するためです。それぞれの *JOB-TITLE* に該当するすべての行を、同じ *MGR-NUMBER* に SET できます。このため、エラー・メッセージは発行されません。

CURRENT OF 句の疑似実行

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。CURRENT OF 句はホスト表には使用できませんが、かわりに各行の ROWID を選択しておいて、更新または削除するときにその値を使って現在行を識別してください。次に例を示します。

```
    05  EMP-NAME    OCCURS 25 TIMES PIC X(20) VARYING.
    05  JOB-TITLE   OCCURS 25 TIMES PIC X(15) VARYING.
    05  OLD-TITLE   OCCURS 25 TIMES PIC X(15) VARYING.
    05  ROW-ID      OCCURS 25 TIMES PIC X(18) VARYING.
...
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
```

```
    SELECT ENAME, JOB, ROWID FROM EMP
END-EXEC.
...
EXEC SQL OPEN EMPCURSOR END-EXEC.
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
...
PERFORM
    EXEC SQL
        FETCH EMPCURSOR
        INTO :EMP-NAME, :JOB-TITLE, :ROW-I
    END-EXEC
    ...
    EXEC SQL
        DELETE FROM EMP
        WHERE JOB = :OLD-TITLE AND ROWID = :ROW-ID
    END-EXEC
    EXEC SQL COMMIT WORK END-EXEC
 END-PERFORM.
```

ただし、ここでは FOR UPDATE OF 句が使用されていないため、フェッチされた行はロックされません。したがって、ある行を読み取ってからその行を削除する前に別のユーザーがその行を変更すると、結果に矛盾が生じることがあります。

ホスト変数としてのグループ項目の表

Pro*COBOL では、埋込み SQL 文でグループ項目（レコードとも呼ばれます）の表を使用できます。グループ項目の表は、SELECT 文または FETCH 文の INTO 句、および INSERT 文の VALUES リストで参照できます。

次に例を示します。

```
01    TABLES.
      05   EMP-TABLE          OCCURS 20 TIMES.
           10    EMP-NUMBER    PIC S9(4) COMP.
           10    EMP-NAME      PIC X(10).
           10    DEPT-NUMBER   PIC S9(4) COMP.
```

このように宣言した場合、次の文は有効です。

```
EXEC SQL INSERT INTO EMP(EMPNO, ENAME, DEPTNO)
     VALUES(:EMP-TABLE)
END-EXEC.
```

グループ項目にデータがすでに入っているものと見なして、この文によって、従業員番号、従業員名および部門番号で構成された行が 20 行一括して EMP 表に挿入されます。

グループ項目の順序が SQL 文内での順序と同じになっていることを確認してください。

グループ項目の表を使うときは、グループの基本項目を個別に指定することもできます。たとえば、次の文も有効です。従業員番号 20 行分が EMP 表の EMPNO 列に挿入されます。

```
EXEC SQL INSERT INTO EMP (EMPNO)
     VALUES (:EMP-TABLE.EMP-NUMBER)
END-EXEC.
```

VARCHAR=YES の場合、グループ項目の宣言が VARCHAR ホスト変数と似ていると、そのグループ項目が 1 つの基本項目のように扱われます。そのため、SQL 文でこのグループ項目を参照するには、基本項目名ではなく、グループ名を使う必要があります。

標識変数を使うには、グループ項目の各変数の標識変数を含むグループ項目表を別に設定します。

```
01     TABLES-IND.
       05   EMP-TABLE-IND  OCCURS 20 TIMES.
            10   EMP-NUMBER-IND       PIC S9(4) COMP.
            10   EMP-NAME-IND         PIC S9(4) COMP.
            10   DEPT-NUMBER_IND      PIC S9(4) COMP.
```

グループ項目のホスト標識表は、次のように使用できます。

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
    VALUES (:EMP-TABLE:EMP-TABLE-IND)
END-EXEC.
```

挿入のときに、グループ項目のコンポーネントに対応する標識グループ項目の要素を個別に使用できます。

```
EXEC SQL INSERT INTO EMP (EMPNO)
     VALUES (:EMP-TABLE.EMP-NUMBER:EMP-TABLE-IND.EMP-NUMBER-IND)
END-EXEC.
```

データの特徴が正確にわかっている場合は、基本項目の標識をグループ項目に指定すると便利です。

```
05    EMP-TABLE-IND     PIC S9(4) COMP
                        OCCURS 20 TIMES.
```

グループ項目のホスト表には、表であるグループ項目を入れることはできません。たとえば、次の場合です。

```
01   TABLES.
     05   EMP-TABLE              OCCURS 20 TIMES.
          10  EMP-NUMBER         PIC S9(4) COMP OCCURS 10 TIMES.
          10  EMP-NAME           PIC X(10).
          10  DEPT-NUMBER        PIC S9(4) COMP.
```

EMP-NUMBER は表であるため、EMP-TABLE をホスト変数として使うことはできません。

ネストされたグループ項目のホスト表は使用できません。たとえば、次の場合です。

```
01   TABLES.
     05   TEAM-TABLE                  OCCURS 20 TIMES
          10   EMP-TABLE
               15   EMP-NUMBER         PIC S9(4) COMP.
               15   EMP-NAME           PIC X(10).
          10   DEPT-TABLE.
               15   DEPT-NUMBER        PIC S9(4) COMP.
               15   DEPT-NAME          PIC X(10).
```

メンバー（EMP-TABLE および DEPT-TABLE）自体がグループ項目であるため、TEAM-TABLE をホスト変数として使うことはできません。

また、Pro*COBOL でホスト表に適用される制限は、グループ項目の表にも適用されます。

- 多次元のホスト表および可変長のホスト表は作成できません。
- 単一の SQL 文で複数の表を使用する場合、各表のディメンションは同じでなければなりません。
- SQL 文のホスト表に添字を付けてはいけません。

サンプル・プログラム 14: グループ項目の表

このプログラムで、ログインし、カーソルを宣言およびオープンし、グループ項目の表を使ってバッチ単位でフェッチを行います。詳細は、最初のコメントを参照してください。

```
*****************************************************************
* Sample Program 14:  Tables of group items                     *
*                                                               *
* This program logs on to ORACLE, declares and opens a cursor,  *
* fetches in batches using a table of group items , and prints  *
* the results.  This sample is identical to sample3 except that *
* instead of using three separate host tables of five elements  *
* each, it uses a five-element table of three group items.      *
* The output should be identical.                               *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID. TABLE-OF-GROUP-ITEMS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
```

```
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  USERNAME          PIC X(15) VARYING.
01  PASSWD            PIC X(15) VARYING.
01  EMP-REC-TABLE OCCURS 5 TIMES.
    05  EMP-NUMBER    PIC S9(4) COMP.
    05  SALARY        PIC S9(6)V99
                       DISPLAY SIGN LEADING SEPARATE.
    05  EMP-NAME      PIC X(10) VARYING.
    EXEC SQL VAR SALARY IS DISPLAY(8,2) END-EXEC.
    EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE SQLCA END-EXEC.
01  NUM-RET           PIC S9(9) COMP VALUE ZERO.
01  PRINT-NUM         PIC S9(9) COMP VALUE ZERO.
01  COUNTER           PIC S9(9) COMP.
01  DISPLAY-VARIABLES.
    05  D-EMP-NAME    PIC X(10).
    05  D-EMP-NUMBER  PIC 9(4).
    05  D-SALARY      PIC Z(4)9.99.

PROCEDURE DIVISION.

BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR DO PERFORM SQL-ERROR
    END-EXEC.
    PERFORM LOGON.
    EXEC SQL
        DECLARE C1 CURSOR FOR
        SELECT EMPNO, SAL, ENAME
        FROM EMP
    END-EXEC.
    EXEC SQL
        OPEN C1
    END-EXEC.

FETCH-LOOP.
    EXEC SQL
        WHENEVER NOT FOUND DO PERFORM SIGN-OFF
    END-EXEC.
    EXEC SQL
        FETCH C1
        INTO :EMP-REC-TABLE
    END-EXEC.
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
    PERFORM PRINT-IT.
    MOVE SQLERRD(3) TO NUM-RET.
    GO TO FETCH-LOOP.
```

```
LOGON.
    MOVE "SCOTT" TO USERNAME-ARR.
    MOVE 5 TO USERNAME-LEN.
    MOVE "TIGER" TO PASSWD-ARR.
    MOVE 5 TO PASSWD-LEN.
    EXEC SQL
       CONNECT :USERNAME IDENTIFIED BY :PASSWD
    END-EXEC.
    DISPLAY " ".
    DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.

PRINT-IT.
    DISPLAY " ".
    DISPLAY "EMPLOYEE NUMBER  SALARY   EMPLOYEE NAME".
    DISPLAY "---------------  -------  -------------".
    PERFORM PRINT-ROWS
        VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > PRINT-NUM.

PRINT-ROWS.
    MOVE EMP-NUMBER(COUNTER) TO D-EMP-NUMBER.
    MOVE SALARY(COUNTER) TO D-SALARY.
    DISPLAY "           ", D-EMP-NUMBER, " ", D-SALARY, "  ",
        EMP-NAME-ARR IN EMP-NAME(COUNTER).
    MOVE SPACES TO EMP-NAME-ARR IN EMP-NAME(COUNTER).

SIGN-OFF.
    SUBTRACT NUM-RET FROM SQLERRD(3) GIVING PRINT-NUM.
    IF (PRINT-NUM > 0) PERFORM PRINT-IT.
    EXEC SQL
        CLOSE C1
    END-EXEC.
    EXEC SQL
        COMMIT WORK RELEASE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "HAVE A GOOD DAY.".
    DISPLAY " ".
    STOP RUN.

SQL-ERROR.
    EXEC SQL
        WHENEVER SQLERROR CONTINUE
    END-EXEC.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
```

```
DISPLAY " ".
DISPLAY SQLERRMC.
EXEC SQL
    ROLLBACK WORK RELEASE
END-EXEC.
STOP RUN.
```

8

エラー処理と診断

アプリケーション・プログラムでは、ランタイム・エラーを見込んで、そのエラーを回復するように対処しなければなりません。この章では、エラーの報告および回復について詳しく説明します。具体的には、状態変数 SQLCODE、SQLSTATE、SQLCA（SQL 通信領域）および WHENEVER 文を使用して警告とエラーを処理する方法を説明します。また、状態変数 ORACA（Oracle 通信領域）を使用して問題を診断する方法も紹介します。この章の構成は、次のとおりです。

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なければなりません。エラー処理の主な利点は、エラーが存在していてもプログラムの処理を続行できることにあります。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、さまざま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ていくことはできます。Pro*COBOL では、エラー処理とは SQL 文の実行エラーを検出して回復を図ることを意味します。

" 切り捨てられた値 " などの警告や " データの終わり " などの状態の変更を処理することもできます。表中の該当する行をすべて処理する前に INSERT 文、UPDATE 文または DELETE 文が失敗する場合もあるので、各データ操作文の後にはエラーや警告の状態をチェックすることが非常に大切です。

エラー処理の代替手段

Pro*COBOL では、エラー処理機構として機能する次の 4 つの状態変数を使用できます。

- SQLCODE
- SQLSTATE
- SQLCA（WHENEVER 文を使用）
- ORACA

プリコンパイラ・オプション MODE は、ANSI/ISO に準拠するかどうかを制御します。SQLCODE および SQLSTATE、SQLCA の各変数を使用できるかどうかは、MODE の設定によって決まります。ORACA 変数は、MODE の設定を問わず宣言して使用できます。詳細は 8-35 ページの「Oracle 通信領域の使用」を参照してください。

MODE={ORACLE | ANSI13} のときは、SQLCA 状態変数を宣言する必要があります。SQLCODE と SQLSTATE の宣言も受け入れられますが（推奨はされていません）、状態変数としては認識されません。詳細は 8-20 ページの「SQL 通信領域の使用」を参照してください。

MODE={ANSI | ANSI14} のときは、SQLCODE、SQLSTATE および SQLCA の各変数のうち 1 つ、2 つまたは 3 つ全部を使用できます。どの変数（または変数の組合せ）がアプリケーションに最善であるかを判別するには、8-4 ページの「MODE={ANSI | ANSI14} の場合の状態変数の使用」を参照してください。

SQLCODE と SQLSTATE

SQLCODE 状態変数は、SQL89 規格の ANSI/ISO エラー報告機構として、Pro*COBOL リリース 1.5 で導入されました。SQL92 規格では、SQLCODE は推奨されない機能として挙げられ、新しい状態変数 SQLSTATE が望ましい ANSI/ISO エラー報告機構として定義されています。（SQLSTATE は Pro*COBOL リリース 1.6 で導入されました。）

SQLCODE には、エラー・コードおよび " 見つからない " という状態が格納されます。SQLCODE は SQL89 との互換性の維持だけを目的として残されたもので、将来的には規格からはずされる予定です。

SQLCODE とは異なり、SQLSTATE にはエラー・コードと警告コードが格納され、標準化されたコード体系が使用されます。データベース・サーバーは、SQL 文の実行後、その時点で有効範囲にある SQLSTATE 変数に状態コードを戻します。状態コードは、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。解釈能力（システム間の情報交換の容易さ）を向上させるため、SQL92 ではよく見られる SQL の例外がすべて事前定義されています。

SQLCA

SQLCA は、レコードに似た、ホスト言語のデータ構造です。Oracle8*i* は、実行 SQL 文を実行するたびに SQLCA を更新します。（宣言文の後では、SQLCA の値は未定義になります。）プログラムは、SQLCA に格納されたリターン・コードをチェックすることによって、SQL 文の結果を判別できます。SQLCA のチェックには、次の 2 つの方法があります。

- 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

WHENEVER 文を使用する方法と、SQLCA 変数の明示的なチェックをコーディングする方法があります。また、その両方を使用することもできます。WHENEVER 文を使用したほうが簡単で移植性が高く、ANSI にも準拠しているため、一般的に望ましいのは WHENEVER 文を使用する方法です。

ネストされたプログラム

ネストされたプログラムでは、提供される組込み SQLCA の定義はグローバルとして宣言されるため、SQLCA は上位のプログラムで宣言するだけで十分です。SQLCA は、新しい SQL 文が実行されるたびに変更される可能性があります。ネストされたプログラムで追加の SQLCA を宣言する場合には、提供される SQLCA のグローバル指定をいつでも削除できます。SQLDA および ORACA にも同じことが言えます。

ORACA

ランタイム・エラーについて SQLCA から得られる情報が十分でなければ、ORACA が使用できます。ORACA には、カーソル統計情報、SQL 文のデータ、オプションの設定およびシステム統計情報が格納されます。

ORACA はオプションであり、MODE の設定に関係なく宣言できます。ORACA 状態変数の詳細は、8-35 ページの「Oracle 通信領域の使用」を参照してください。

MODE={ANSI | ANSI14} の場合の状態変数の使用

MODE={ANSI | ANSI14} のときは、次の状態変数を最低 1 つは宣言する必要があります。（2 つ宣言することもできます。）

- SQLCODE
- SQLSTATE
- SQLCA

SQLCA が宣言されている場合は、SQLCODE は宣言できません。同様に、SQLCODE が宣言されている場合は SQLCA は宣言できません。SQLCA データ構造の中のエラー・コードを格納するフィールドも SQLCODE という名前になっているため、SQLCA 状態変数と SQLCODE 状態変数の両方を宣言するとエラーが発生します。

プログラムでは、実行 SQL 文および実行 PL/SQL 文の後に、SQLCODE または SQLSTATE、あるいはその両方の明示的なチェックをコーディングすることによって、一番最後に実行された実行 SQL 文の結果が得られます。また、（WHENEVER SQLERROR 文および WHENEVER SQLWARNING 文を使った）SQLCA の暗黙的なチェック、または SQLCA 変数の明示的なチェックもできます。

> **注意 :** MODE={ORACLE | ANSI13 | ANSI14} のときは、SQLCA 状態変数を宣言する必要があります。詳細は 8-20 ページの「SQL 通信領域の使用」を参照してください。

これまでの経緯

状態変数および変数の組合せを Pro*COBOL がどのように扱うかは、リリース 1.5 以降変化してきました。

リリース 1.5

Pro*COBOL リリース 1.5 では、状態変数 SQLCODE は、宣言されているかどうかに関係なく存在するものと想定されていました。つまり、SQLCODE が宣言されているかどうかを調べずに、単に存在するものと見なされていました。また、SQLCA が状態変数として使われるのは、SQLCA の INCLUDE が存在する場合だけでした。

リリース 1.6

Pro*COBOL リリース 1.6 からは、プリコンパイラは SQLCODE 状態変数が存在するものと想定しなくなり、SQLCODE 状態変数は必須ではなくなりました。必要条件は、SQLCODE と SQLSTATE のうち少なくとも一方が宣言されることに変わりました。

SQLCODE は、次の基準の少なくとも 1 つが満たされる場合だけに、状態変数として認識されます。

- 完全に正しいデータ型で宣言されている場合
- Pro*COBOL が他の状態変数を見つけられない場合

Pro*COBOL は、（完全に正しい型の）SQLSTATE 宣言を検出した場合や、SQLCA の INCLUDE を検出した場合には、SQLCODE が宣言されているとはみなしません。

リリース 1.7

Pro*COBOL リリース 1.5 では、SQLCA が宣言されている場合でも宣言文の外でならば SQLCODE 変数を宣言できたため、Pro*COBOL リリース 1.6 以降では互換性の問題が出てきました。リリース 1.6.7 ではこの問題に対処するために新しいオプション ASSUME_SQLCODE={YES | NO}（デフォルトは NO）が追加され、リリース 1.7 の新機能として記載されています。

リリース 8.0

リリース 8.0 からは、宣言文はオプションになりました。ASSUME_SQLCODE オプションの詳細は、14-13 ページの「ASSUME_SQLCODE」を参照してください。

状態変数の宣言

この項では、SQLCODE と SQLSTATE の宣言方法を説明します。ORACA 状態変数の宣言については、8-21 ページの「SQLCA の宣言」を参照してください。

SQLCODE の宣言

SQLCODE は、次の例に示すように、宣言文の内部または外部で 4 バイトの整変数として宣言する必要があります。

```
*     Declare host and indicator variables.
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
*     Declare the SQLCODE status variable.
 01   SQLCODE  PIC S9(9) COMP.
```

宣言文の外で宣言した場合、SQLCODE が状態変数として認識されるのは ASSUME_SQLCODE=YES のときだけです。MODE={ORACLE | ANSI13 | ANSI14} のときは、SQLCODE 状態変数の宣言は無視されます。

> **警告 :** SQLCA が宣言されている場合は、SQLCODE を宣言しないでください。同様に、SQLCODE が宣言されている場合は、SQLCA を宣言しないでください。SQLCA 構造体によって宣言される状態変数も SQLCODE という名前なので、両方のエラー・レポート・メカニズムを使うとエラーが発生します。

Oracle8*i* は、SQL 操作を実行するたびに SQLCODE 変数に状態コードを戻します。したがって、プログラムは、SQLCODE を明示的にチェックするか、WHENEVER 文で暗黙的にチェックすることによって、最新の SQL 処理の結果を知ることができます。

SQLCA ではなく SQLCODE をコンパイル・ユニットに宣言すると、Pro*COBOL はそのユニットに内部 SQLCA を割り当てます。ホスト・プログラムは内部 SQLCA にはアクセスできません。

SQLSTATE の宣言

SQLSTATE は、次の例に示すように、5 文字の英数文字列として宣言する必要があります。

```
*    Declare the SQLSTATE status variable.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       ...
 01 SQLSTATE PIC X(5).
       ...
     EXEC SQL END DECLARE SECTION END-EXEC.
```

MODE={ORACLE | ANSI13 | ANSI14} のときは、SQLSTATE の宣言は無視されます。SQLCA の宣言はオプションです。

状態変数の組合せ

MODE={ANSI | ANSI14} のときは、状態変数の動作は次の設定によって変わります。

- どの変数が宣言されているか
- 宣言の位置（宣言文の内部または外部）
- ASSUME_SQLCODE の設定

表 8-1 と表 8-2 に、ASSUME_SQLCODE=NO および ASSUME_SQLCODE=YES のときの各状態変数の組合わせの動作結果をそれぞれ示します。

どちらの表でも、DECLARE_SECTION=NO のときには、状態変数の宣言はすべて（宣言文の）内部にあるものとして扱われます。

ASSUME_SQLCODE=YES は、DECLARE_SECTION=NO と併用しないでください。

表 8-1 ASSUME_SQLCODE=NO、MODE=ANSI | ANSI14、DECLARE_SECTION=YES のときの状態変数の動作

宣言文（IN/OUT/—）			動作
SQLCODE	**SQLSTATE**	**SQLCA**	
OUT	—	—	SQLCODE が宣言され、状態変数であるとみなされます。
OUT	—	OUT	この状態変数の構成はサポートされていません。
OUT	—	IN	この状態変数の構成はサポートされていません。
OUT	OUT	—	SQLCODE が宣言され、状態変数とみなされます。SQLSTATE は宣言されますが、状態変数とは認識されません。
OUT	OUT	OUT	この状態変数の構成はサポートされていません。
OUT	OUT	IN	この状態変数の構成はサポートされていません。
OUT	IN	—	SQLSTATE が状態変数として宣言されます。SQLCODE は宣言されますが、状態変数とは認識されません。
OUT	IN	OUT	この状態変数の構成はサポートされていません。
OUT	IN	IN	この状態変数の構成はサポートされていません。
IN	—	—	SQLCODE が状態変数として宣言されます。
IN	—	OUT	この状態変数の構成はサポートされていません。
IN	—	IN	この状態変数の構成はサポートされていません。
IN	OUT	—	SQLCODE が状態変数として宣言されます。SQLSTATE は宣言されますが、状態変数とは認識されません。
IN	OUT	OUT	この状態変数の構成はサポートされていません。
IN	OUT	IN	この状態変数の構成はサポートされていません。
IN	IN	—	SQLCODE と SQLSTATE が状態変数として宣言されます。
IN	IN	OUT	この状態変数の構成はサポートされていません。
IN	IN	IN	この状態変数の構成はサポートされていません。
—	—	—	この状態変数の構成はサポートされていません。
—	—	OUT	SQLCA が状態変数として宣言されます。
—	—	IN	SQLCA が状態ホスト変数として宣言されます。
—	OUT	—	この状態変数の構成はサポートされていません。
—	OUT	OUT	SQLCA が状態変数として宣言されます。SQLSTATE は宣言されますが、状態変数とは認識されません。

表 8-1 ASSUME_SQLCODE=NO、MODE=ANSI | ANSI14、DECLARE_SECTION=YES のときの状態変数の動作 (続き)

宣言文（IN/OUT/—）			動作
SQLCODE	**SQLSTATE**	**SQLCA**	
—	OUT	IN	SQLCA が状態ホスト変数として宣言されます。SQLSTATE は宣言されますが、状態変数とは認識されません。
—	IN	—	SQLSTATE が状態変数として宣言されます。
—	IN	OUT	SQLSTATE と SQLCA が状態変数として宣言されます。
—	IN	IN	SQLSTATE と SQLCA が状態ホスト変数として宣言されます。

表 8-2 ASSUME_SQLCODE=YES、MODE=ANSI | ANSI14、DECLARE_SECTION=YES のときの状態変数の動作

宣言文（IN/OUT/—）			動作
SQLCODE	**SQLSTATE**	**SQLCA**	
OUT	—	—	SQLCODE が宣言され、状態変数であるとみなされます。
OUT	—	OUT	この状態変数の構成はサポートされていません。
OUT	—	IN	この状態変数の構成はサポートされていません。
OUT	OUT	—	SQLCODE が宣言され、状態変数とみなされます。SQLSTATE は宣言されますが、状態変数とは認識されません。
OUT	OUT	OUT	この状態変数の構成はサポートされていません。
OUT	OUT	IN	この状態変数の構成はサポートされていません。
OUT	IN	—	SQLSTATE が状態変数として宣言されます。SQLCODE は宣言されますが、状態変数とはみなされません。
OUT	IN	OUT	この状態変数の構成はサポートされていません。
OUT	IN	IN	この状態変数の構成はサポートされていません。
IN	—	—	SQLCODE が状態変数として宣言されます。
IN	—	OUT	この状態変数の構成はサポートされていません。
IN	—	IN	この状態変数の構成はサポートされていません。
IN	OUT	—	SQLCODE が状態変数として宣言されます。SQLSTATE は宣言されますが、状態変数とは認識されません。
IN	OUT	OUT	この状態変数の構成はサポートされていません。
IN	OUT	IN	この状態変数の構成はサポートされていません。

表 8-2 ASSUME_SQLCODE=YES、MODE=ANSI | ANSI14、DECLARE_SECTION=YES のときの状態変数の動作 (続き)

宣言文（IN/OUT/—）			動作
SQLCODE	**SQLSTATE**	**SQLCA**	
IN	IN	—	SQLCODE と SQLSTATE が状態変数として宣言されます。
IN	IN	OUT	この状態変数の構成はサポートされていません。
IN	IN	IN	この状態変数の構成はサポートされていません。
—	—	—	これらの状態変数の構成はサポートされていません。ASSUME_SQLCODE=YES のときは、SQLCODE を宣言する必要があります。
—	—	OUT	
—	—	IN	
—	OUT	—	
—	OUT	OUT	
—	OUT	IN	
—	IN	—	
—	IN	OUT	
—	IN	IN	

状態変数の値

この項では、SQLCODE 状態変数と SQLSTATE 状態変数の値について説明します。SQLCA 状態変数の詳細は、8-22 ページの「エラー報告の基本コンポーネント」を参照してください。

SQLCODE 値

Oracle8*i* は、SQL 操作を実行するたびに、その時点で有効範囲にある SQLCODE 変数に状態コードを戻します。状態コードは SQL 処理の結果を表します。コードの値は次のいずれかです。

最後に実行された SQL 操作の結果を確認するには、SQLCODE の明示的なチェックをコーディングするか、WHENEVER 文を使って SQLCODE を暗黙的にチェックします。

SQLCA のかわりに SQLCODE が宣言されているプリコンパイル・ユニットには、Pro*COBOL は内部 SQLCA を割り当てます。ホスト・プログラムは内部 SQLCA にはアクセスできません。

注意 : MODE={ORACLE | ANSI13} のときは、SQLCODE の宣言は無視されます。

SQLSTATE 値

SQLSTATE 状態コードは、2 文字のクラス・コードとそれに続く 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 は正常終了を示し、それ以外のクラス・コードは例外のカテゴリを示します。サブクラス・コード 000 は特定の例外を示しませんが、それ以外のサブクラス・コードは そのカテゴリの中の特定の例外を示します。たとえば、SQLSTATE 値 '222012' には、クラス・コード 22（データの例外）とサブクラス・コード 012（0 による除算）が含まれます。

SQLSTATE 値の 5 文字はそれぞれ数字（0 ～ 9）または大文字の英文字（A ～ Z）で構成されます。0 ～ 4 の範囲の数字、または A ～ H の範囲の文字で始まるクラス・コードは、事前定義済みの状態（SQL92 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・コードは処理系定義の状態用に確保されています。事前定義済みのクラス内では、0 ～ 4 の範囲の数字または A ～ H の範囲の英文字で始まるサブクラス・コードは、事前定義済みのサブコンディションを示すために確保されています。その他のサブクラス・コードはすべて、処理系定義のサブコンディションを示すために確保されています。図 8-1 に、このコード体系を示します。

図 8-1 SQLSTATE コード体系

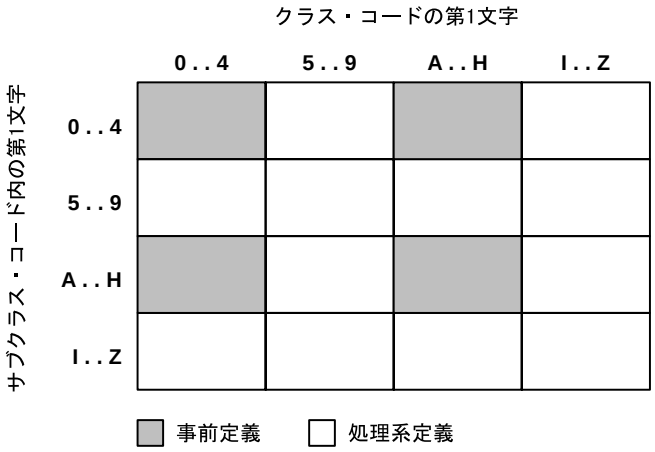


表 8-3 に、SQL92 によって事前定義されているクラスを示します。

表 8-3 事前定義のクラス

CLASS	状態
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
0A	サポートされていない機能
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
33	SQL 記述子名が無効
34	カーソル名が無効
35	状態番号が無効
37	動的 SQL 構文エラーまたはアクセス規則違反
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

表 8-3 事前定義のクラス (続き)

CLASS	状態
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 で定義された状態であるリモート・データベース・アクセス用に確保されています。

表 8-4 に、エラーと SQLSTATE 状態コードの対応関係を示します。1 つの状態コードに複数のエラーが対応する場合もあります。また、状態コードに対応するエラーがない場合もあります。（この場合は、表の右端の欄は空白になります。）60000 ～ 99999 の範囲の状態コードは、処理系定義です。

表 8-4 SQLSTATE コード

コード	状態	Oracle8*i* エラー
00000	正常終了	ORA-00000
01000	警告	
01001	カーソル操作の競合	
01002	切断のエラー	
01003	集合関数での NULL 値が排除	
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ません。	
01007	権限が付与されていません。	
01008	暗黙の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。	
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	準備された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	結果フィールドには USING 句が必要です。	
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 のクライアントが SQL 接続を確立できません。	

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
08002	接続名の重複	
08003	接続が存在しません。	SQL-02121
08004	SQL サーバーによる SQL 接続の拒絶	
08006	接続障害	
08007	トランザクションの結果が不明です。	
0A000	サポートされていない機能	ORA-03000 ～ 03099
0A001	複数のサーバー・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-01401 ORA-01406
22002	NULL 値（標識パラメータなし）	ORA-01405 SQL-02124
22003	数値が範囲外	ORA-01426 ORA-01438 ORA-01455 ORA-01457
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 ～ 01899
22009	時間帯の変位値が無効	
22011	副文字列のエラー	
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 ORA-01425
22021	レパートリに文字がありません。	

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
22022	標識のオーバーフロー	ORA-01411
22023	パラメータ値が無効	ORA-01025 ORA-01488 ORA-04000 ～ 04019
22024	C 文字列が未終了	ORA-01479 ～ 01480
22025	エスケープ・シーケンスが無効	ORA-01424
22026	文字列データの長さ不一致	
22027	切捨てエラー	
23000	整合性制約違反	ORA-00001 ORA-02290 ～ 02299
24000	カーソル状態が無効	ORA-01001 ～ 01003 ORA-01410 ORA-08006 SQL-02114 SQL-02117 SQL-02118 SQL-02122
25000	トランザクション状態が無効	
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	

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
35000	状態番号が無効	
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	
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 ～ 02092
40001	直列化の障害	
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 ～ 00999
		ORA-01031
		ORA-01490 ～ 01493
		ORA-01700 ～ 01799
		ORA-01900 ～ 02099
		ORA-02140 ～ 02289
		ORA-02420 ～ 02424
		ORA-02450 ～ 02499
		ORA-03276 ～ 03299
		ORA-04040 ～ 04059
		ORA-04070 ～ 04099
44000	WITH_CHECK_OPTION 指定違反	ORA-01402
60000	システム・エラー	ORA-00370 ～ 00429
		ORA-00600 ～ 00899
		ORA-06430 ～ 06449
		ORA-07200 ～ 07999
		ORA-09700 ～ 09999

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
61000	リソース・エラー	ORA-00018 ～ 00035 ORA-00050 ～ 00068 ORA-02376 ～ 02399 ORA-04020 ～ 04039
62000	マルチスレッド・サーバーおよび分離プロセスのエラー	ORA-00100 ～ 00120 ORA-00440 ～ 00569
63000	Oracle XA および 2 タスク・インタフェースのエラー	ORA-00150 ～ 00159 SQL-02128 ORA-02700 ～ 02899 ORA-03100 ～ 03199 ORA-06200 ～ 06249 SQL-02128
64000	制御ファイル、データベース・ファイル、REDO ファイルのエラー、ならびに アーカイブおよびメディア・リカバリのエラー	ORA-00200 ～ 00369 ORA-01100 ～ 01250
65000	PL / SQL のエラー	ORA-06500 ～ 06599
66000	Net8 ドライバのエラー	ORA-06000 ～ 06149 ORA-06250 ～ 06429 ORA-06600 ～ 06999 ORA-12100 ～ 12299 ORA-12500 ～ 12599
67000	ライセンス許可エラー	ora-00430 ～ 00439
69000	SQL*Connect のエラー	ora-00570 ～ 00599 ora-07000 ～ 07199

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
72000	SQL 実行フェーズのエラー	ora-01000 〜 01099
		ora-01400 〜 01489
		ora-01495 〜 01499
		ORA-01500 〜 01699
		ORA-02400 〜 02419
		ORA-02425 〜 02449
		ORA-04060 〜 04069
		ORA-08000 〜 08190
		ORA-12000 〜 12019
		ORA-12300 〜 12499
		ORA-12700 〜 21999
82100	メモリー不足（割り当てられない）	SQL-02100
82101	一貫性のないカーソル・キャッシュ : ユニット・カーソル / グローバル・カーソルの不一致	SQL-02101
82102	一貫性のないカーソル・キャッシュ : グローバル・カーソルのエントリではありません。	SQL-02102
82103	一貫性のないカーソル・キャッシュ : カーソル・キャッシュ参照の範囲外	SQL-02103
82104	一貫性のないカーソル・キャッシュ : カーソル・キャッシュを使用できません。	SQL-02104
82105	一貫性のないカーソル・キャッシュ : グローバル・カーソルが存在しません。	SQL-02105
82106	一貫性のないカーソル・キャッシュ（カーソル番号が無効）	SQL-02106
82107	実行時ライブラリに対してプログラムが古すぎます。	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	一貫性のないカーソル・キャッシュ : ホスト参照が範囲外	SQL-02109
82110	一貫性のないカーソル・キャッシュ : ホスト・キャッシュの入力タイプが無効	SQL-02110

表 8-4 SQLSTATE コード (続き)

コード	状態	Oracle8*i* エラー
82111	ヒープ一貫性のエラー	SQL-02111
82112	メッセージ・ファイルをオープンできません。	SQL-02113
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
82115	hstdef 引数が無効	SQL-02119
82116	sqlrcn の第 1 引数と第 2 引数がともに NULL	SQL-02120
82117	この接続での OPEN または PREPARE が無効	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	接続エラー。エラー・テキストを入手できません。	SQL-02125
82120	プリコンパイラと SQLLIB のバージョン不一致	SQL-02127
82121	FETCH したバイト数が奇数	SQL-02129
82122	EXEC TOOLS インタフェースが使用できません。	SQL-02130
82123	ランタイム・コンテキストが使用中	SQL-02131
82124	ランタイム・コンテキストが割り当てられていません。	SQL-02131
82125	スレッドで使用するための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
90000	デバッグ・イベント	ORA-10000 ～ 10999
99999	すべて捕捉	その他すべて
HZ000	リモート・データベース・アクセス	

SQL 通信領域の使用

Oracle8*i* では、実行時にプログラムに渡された状態情報は SQL 通信領域（SQLCA）に格納されます。SQLCA はレコードに似た COBOL データ構造で、実行 SQL 文が実行されるたびに更新されます。このため、SQLCA は常に、一番最後に実行された SQL 操作の結果を反映しています。SQLCA の各フィールドには、エラー、警告、状態の情報が格納されます。これらの情報は、SQL 文が実行されるたびに Oracle8*i* によって更新されます。SQL 文の実行結果を確認するには、SQLCA 内の変数を調べます。これには、COBOL コードを記述することによって明示的に調べる方法と、WHENEVER 文を使って暗黙的に調べる方法があります。

> **注意 :** アプリケーションで SQL*Net を使ってローカル・データベースとリモート・データベース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 に書込みを行います。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。詳細は 3-12 ページの「同時ログイン」を参照してください。

MODE={ORACLE | ANSI13} のときは SQLCA は必須です。SQLCA が宣言されていないと、コンパイル時エラーが発生します。MODE={ANSI | ANSI14} のときは SQLCA は必須ではありませんが、SQLCA がないと WHENEVER SQLWARNING 文は使用できません。したがって、WHENEVER SQLWARNING 文を使用する場合は、SQLCA を宣言する必要があります。

注意 : 特定のコンパイル・ユニットで SQLCA のかわりに SQLCODE を宣言すると、そのユニットに対して内部 SQLCA が割り当てられます。ホスト・プログラムは内部 SQLCA にはアクセスできません。

MODE={ANSI | ANSI14} の場合、SQLSTATE（8-6 ページの「SQLSTATE の宣言」を参照）または SQLCODE（8-5 ページの「SQLCODE の宣言」を参照）、あるいは両方を宣言する必要があります。SQLSTATE 状態変数は、SQL92 規格に規定されている SQLSTATE 状態変数をサポートしています。SQLSTATE 状態変数は、SQLCODE と併用することも単独で使用することもできます。

SQLCA 内の情報

SQLCA には、エラー・コード、警告フラグ、イベント情報、処理済み行数、診断情報など、SQL 文の実行に関する実行時情報が格納されます。

図 8-2 に、SQLCA 内のすべての変数を示します。ただし、SQLWARN2 および SQLWARN5、SQLWARN6、SQLWARN7、SQLEXT は現在は使用されていません。

図 8-2 Pro*COBOL の SQLCA 変数宣言

```
01  SQLCA.
    05  SQLCAID                PIC X(8).
    05  SQLCABC                PIC S9(9) COMPUTATIONAL.
    05  SQLCODE                PIC S9(9) COMPUTATIONAL.
    05  SQLERRM.
        49 SQLERRML            PIC S9(4) COMPUTATIONAL.
        49 SQLERRMC            PIC X(70)
    05  SQLERRP                PIC X(8).
    05  SQLERRD OCCURS 6 TIMES
                               PIC S9(9) COMPUTATIONAL.
    05  SQLWARN.
        10 SQLWARNO            PIC X(1).
        10 SQLWARN1            PIC X(1).
        10 SQLWARN2            PIC X(1).
        10 SQLWARN3            PIC X(1).
        10 SQLWARN4            PIC X(1).
        10 SQLWARN5            PIC X(1).
        10 SQLWARN6            PIC X(1).
        10 SQLWARN7            PIC X(1).
    05  SQLEXT                 PIC X(8).
```

SQLCA の宣言

SQLCA を宣言するには、次に示すように、（EXEC SQL INCLUDE 文を使って）Pro*COBOL ソース・ファイルの宣言文の外に組み込みます。

```
*    Include the SQL Communications Area (SQLCA).
     EXEC SQL INCLUDE SQLCA END-EXEC.
```

SQLCA は宣言文の外で宣言しなければなりません。

警告 : SQLCA が宣言されている場合は、SQLCODE を宣言しないでください。同様に、SQLCODE が宣言されている場合は、SQLCA を宣言しないでください。SQLCA 構造体によって宣言される状態変数も SQLCODE という名前なので、両方のエラー・レポート・メカニズムを使うと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、Oracle8*i* とプログラムとの間のやりとりが可能になります。

注意 : マルチバイトの NCHAR ホスト変数を使う場合は、SQLCA を組み込む必要があります。

エラー報告の基本コンポーネント

Pro*COBOL のエラー報告の基本的なコンポーネントは、SQLCA 内のいくつかのフィールドを使用します。

状態コード

実行 SQL 文はすべて、SQLCA 変数 SQLCODE に状態コードを戻します。戻された状態コードは、WHENEVER SQLERROR を使って暗黙的に、または COBOL コードを記述して明示的にチェックできます。

警告フラグ

警告フラグは、SQLCA 変数 SQLWARN0 ～ SQLWARN7 に戻されます。戻された警告フラグは、WHENEVER SQLWARNING を使用するか、COBOL コードを記述することによってチェックできます。警告フラグは、エラーとみなさない実行時の状態を検出する場合に役立ちます。

処理済み行数

一番最後に実行された SQL 文で処理された行数が、SQLCA 変数 SQLERRD(3) に戻されます。OPEN カーソルで繰り返される FETCH については、フェッチされた行数の実行合計が SQLERRD(3) に格納されます。

解析エラー・オフセット

Oracle8*i* は、SQL 文を解析してから実行します。つまり実行前に、SQL 文に正しい構文規則が使われ有効なデータベース・オブジェクトを参照しているかを確認します。エラーが見つかると、SQLCA 変数 SQLERRD(5) にオフセットを格納します。このオフセットは明示的にチェックできます。解析エラーの始まりを示す SQL 文中の文字位置がオフセットとして指定されます。先頭の文字位置は 0（ゼロ）です。たとえば、オフセットが 9 の場合、解析エラーの始まりは 10 番目の文字です。

SQL 文に解析エラーがない場合、SQLERRD(5) は 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 0（ゼロ））から始まっている場合にも、SQLERRD(5) は 0（ゼロ）に設定されます。このため、SQLERRD(5) のチェックは、SQLCODE が負の値（エラーが発生したことを示す）の場合にだけ行ってください。

エラー・メッセージ・テキスト

エラー・コードとメッセージは、SQLCA 変数 SQLERRMC に格納されます。たとえば、エラー処理ルーチンに次の文を記述します。

```
*    Handle SQL execution errors.
     MOVE SQLERRMC TO ERROR-MESSAGE.
     DISPLAY ERROR-MESSAGE.
```

格納されるのは、メッセージ・テキストの先頭から 70 文字までです。70 文字を超えるメッセージについては、SQLGLM サブルーチンをコールする必要があります。（SQLGLM サブルーチンについては 8-26 ページの「エラー・メッセージのテキスト全体の取得」で説明します。）

SQLCA の構造

この項では、SQLCA の構造およびそのフィールド、フィールドに格納できる値について説明します。

SQLCAID

この文字列フィールドは、SQL 通信領域を示す "SQLCA" に初期化されます。

SQLCABC

この整数フィールドには、SQLCA 構造の長さ（バイト数）が格納されます。

SQLCODE

この整数フィールドには、一番最後に実行された SQL 文の状態コードが格納されます。状態コードは SQL 処理の結果を表します。コードの値は次のいずれかです。

0	Oracle8*i* は文を実行し、エラーも例外も検出しませんでした。
> 0	Oracle8*i* は文を実行しましたが、例外を検出しました。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。
< 0	MODE={ANSI | ANSI14 | ANSI13} のときは、1 行も INSERT されなかった場合に SQLCODE に +100 が戻されます。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。 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いずれかにエラーが発生したため、Oracle8*i* は文を実行しませんでした。このようなエラーは致命的です。こうしたエラーが発生すると、ほとんどの場合は現行トランザクションがロールバックされます。 負のリターン・コードは、『Oracle8*i* エラー・メッセージ』に記載されているエラー・コードに対応しています。

SQLERRM

このサブレコードには、次の 2 つのフィールドがあります。

SQLERRML　この整数フィールドには、SQLERRMC に格納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　この文字列フィールドには、SQLCODE に格納されているエラー・コードに対応するメッセージ・テキストが格納されます。格納できるのは最大 70 文字です。70 文字を超えるメッセージのテキスト全体を調べる場合は、SQLGLM 関数を使用します。

SQLERRMC を参照するには、まず SQLCODE が負の値であることを確かめてください。SQLCODE が 0（ゼロ）の場合は、SQLERRMC を参照すると前の SQL 文に関連するメッセージ・テキストが戻されます。

SQLERRP

この文字列フィールドは、将来の使用に備えて確保されています。

SQLERRD

この 2 進整数の表には 6 つの要素があります。SQLERRD の各フィールドについて説明します。

SQLERRD(1)　このフィールドは、将来の使用に備えて確保されています。

SQLERRD(2)　このフィールドは、将来の使用に備えて確保されています。

SQLERRD(3)　このフィールドには、一番最後に実行された SQL 文で処理された行の数が格納されます。SQL 文が失敗した場合は、SQLERRD(3) の値は未定義になります。ただし、1 つだけ例外があります。表の操作中にエラーが発生した場合、処理はエラーの原因となった行で停止するため、SQLERRD(3) には正常に処理された行の数が格納されます。

処理済み行数は OPEN 文の後に 0（ゼロ）に設定され、FETCH 文の後にインクリメントされます。処理済み行数は、EXECUTE 文および INSERT 文、UPDATE 文、DELETE 文、SELECT INTO 文について、正常に処理された行の数を反映します。この数字には UPDATE または DELETE CASCADE によって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み行数は、25 ではなく 20 となります。

SQLERRD(4)　このフィールドは、将来の使用に備えて確保されています。

SQLERRD(5) このフィールドには、一番最後に実行された SQL 文中の、解析エラーが始まる文字位置を示すオフセットが格納されます。先頭の文字位置は 0（ゼロ）です。

SQLERRD(6) このフィールドは、将来の使用に備えて確保されています。

この表は、それぞれ 1 文字からなる 8 つの要素で構成されています。これらの要素は警告フラグとして使用されます。Oracle8*i* では、フラグに "W"（警告）文字値を割り当てて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、Oracle8*i* で列値を切り捨てて出力ホスト文字変数に割り当てると、警告フラグが設定されます。

注意 : 8-21 ページの図 8-2 の「Pro*COBOL の SQLCA 変数宣言」では SQLWARN を表で示していますが、Pro*COBOL では SQLWARN0 ～ SQLWARN7 という名前の基本的な PIC X 項目からなるグループ項目としてインプリメントされます。

SQLWARN の各フィールドについて説明します。

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だけ適用されます。Oracle8*i* が一部の数値データを切り捨てるときには、警告の設定も負の SQLCODE 値の戻しもありません。

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。

SQLWARN(2) AVG、COUNT、MAX などの SQL グループ関数の評価で 1 つ以上の NULL が無視されたときに、このフラグが設定されます。COUNT(*) 以外のすべてのグループ関数では NULL が無視されるので、このような動作になります。必要であれば、SQL 関数 NVL を使って、NULL の列項目に一時的に値（0（ゼロ）など）を割り当てることができます。

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 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。

SQLEXT

この文字列フィールドは、将来の使用に備えて確保されています。

PL/SQL に関する考慮事項

埋込み PL/SQL ブロックを実行する Pro*COBOL プログラムの場合、SQLCA のフィールドがすべて設定されるわけではありません。たとえば、PL/SQL ブロックで複数の行がフェッチされた場合、処理済み行数 SQLERRD(3) は、実際にフェッチされた行数ではなく、1 に設定されます。したがって、PL/SQL ブロックを実行した後は、信頼できる SQLCA のフィールドは SQLCODE フィールドおよび SQLERRM フィールドだけになります。

エラー・メッセージのテキスト全体の取得

SQLCA には 70 文字（バイト）までの長さのエラー・メッセージを格納できます。これより長い（またはネストされた）エラー・メッセージのテキスト全体を取得するには、SQLGLM サブルーチンを使う必要があります。

データベースに接続すると、次の構文を使って SQLGLM をコールできます。

```
CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH
```

パラメータは次のとおりです。

MSG-TEXT　エラー・メッセージを格納するフィールド（このフィールドには先頭から格納され、余った部分は空白で埋められます。）

MAX-SIZE　MSG-TEXT フィールドの最大サイズ（バイト数）を指定する整数

MSG-LENGTH　エラー・メッセージの実際の長さを格納する整変数

エラー・メッセージの最大長は、512 文字です。これには、エラー・コード、ネストされたメッセージ、表名や列名などのメッセージ挿入語句を含みます。SQLGLM で戻されるエラー・メッセージの最大長は、MAX-SIZE に指定した値によって決まります。

次の例では、SQLGLM を使って、エラー・メッセージの最大長を 200 文字に設定します。

```
     ...
*    Declare variables for the SQL-ERROR subroutine call.
 01  MSG-TEXT    PIC X(200).
 01  MAX-SIZE    PIC S9(9) COMP VALUE 200.
 01  MSG-LENGTH  PIC S9(9) COMP.
     ...
```

```
 PROCEDURE DIVISION.
 MAIN.
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
 SQL-ERROR.
*    Clear the previous message text.
     MOVE SPACES TO MSG-TEXT.
*    Get the full text of the error message.
     CALL "SQLGLM" USING MSG-TEXT, MAX-SIZE, MSG-LENGTH.
     DISPLAY MSG-TEXT.
```

この例では、SQLGLM は SQL エラーが発生したときだけにコールされます。SQLGLM をコールする前に、SQLCODE が負の値である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、前の SQL 文に対応するメッセージ・テキストが戻されます。

注意: アプリケーションで SQLGLM をコールしてメッセージ・テキストを取得したり Oracle*Forms ユーザー・イグジットで SQLIEM をコールして失敗時メッセージを表示する際には、メッセージの長さを渡す必要があります。SQLCA 変数 SQLERRML は使わないでください。SQLERRML は PIC S9(4) COMP 整数ですが、SQLGLM と SQLIEM では PIC S9(9) COMP 整数が予測されています。かわりに、PIC S9(9) COMP として宣言した別の変数を使います。

DSNTIAR

DB2 には、表示可能な形式の SQLCA を取得するための、DSNTIAR というアセンブラ・ルーチンがあります。DB2 から Oracle へ移行するユーザーのために、Pro*COBOL にも DSNTIAR が用意されています。DSNTIAR は、ラップ・アラウンドを行う SQLGLM として実現されています。DSNTIAR のインタフェースは次のとおりです。

```
     CALL 'DSNTIAR' USING SQLCA MESSAGE LRECL
```

MESSAGE はサイズ 240 以上の VARCHAR 形式の出力メッセージ領域で、*LRECL* は出力メッセージの長さ（72 ～ 240）が格納されるフル・ワードです。MESSAGE 引数の最初のハーフワードには、残りの領域の長さが含まれます。DSNTIAR が戻すエラー・コードは次のとおりです。

表 8-5 DNSTIAR エラー・コードと説明

0	正常に実行しました。
4	指定されたメッセージに入りきらないデータがあります。
8	論理レコード長（LRECL）が 72 ～ 240 の範囲外です。
12	メッセージ領域の大きさが十分ではありません（240 よりも大）。

WHENEVER ディレクティブ

デフォルトでは、Pro*COBOL は可能であればエラーおよび警告状態を無視して処理を続行します。自動状態チェックおよびエラー処理を実行するには WHENEVER 文が必要です。

WHENEVER 文を使用すると、Oracle8*i* がエラー、警告状態または " 見つからない " という状態を検出した場合の処置を指定できます。指定できる処置としては、次の文からの処理の続行、段落の PERFORM、段落への分岐、停止などがあります。

Oracle8*i* 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。

状態

SQLWARNING

Oracle8*i* から警告が戻されたため（SQLWARN(1) ～ SQLWARN(7) の警告フラグのうちの 1 つも設定されます）、あるいは SQLCODE の値が +1403 以外の正の値になっていたために、SQLWARN(0) が設定されている状態です。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(1) が設定されます。

SQLCA の宣言は、MODE={ANSI | ANSI14} のときにはオプションですが、WHENEVER SQLWARNING 文を使うには SQLCA を必ず宣言しなければなりません。

SQLERROR

Oracle8*i* からエラーが戻されたために、SQLCODE が負の値に設定されている状態です。

NOT FOUND または NOTFOUND

WHERE 句の検索条件を満たす行が見つからなかったため、あるいは SELECT INTO または FETCH で 1 行も戻されなかったために、SQLCODE の値が +1403（MODE={ANSI | ANSI14 | ANSI13} の場合は +100）に設定されている状態です。MODE={ANSI | ANSI14 | ANSI13} のときは、1 行も INSERT されなかった場合に SQLCODE に +100 が戻されます。

DB2 では、SQL 文の実行後に END-OF-FETCH 状態が発生した場合、SQLCODE の値として 100 が戻されます。このため、Pro*COBOL には、END-OF-FETCH 状態の発生時に戻される値を明示的に制御するための新しいコマンド行オプションが用意されています。このオプションは次のとおりです。

```
END_OF_FETCH = 100 | 1403 (default 1403)
```

END_OF_FETCH オプションは、コマンド行または構成ファイルで指定しなければなりません。詳細は 14-19 ページの「END_OF_FETCH」を参照してください。

構成ファイルで MODE=ANSI と指定した場合、END_OF_FETCH 状態が発生すると、Pro*COBOL は SQLCODE 値 100 を戻し、END_OF_FETCH のデフォルト値である 1403 を上書きします。構成ファイルで MODE=ANSI と END_OF_FETCH=1403 の両方を指定した

場合は、END_OF_FETCH 状態が発生すると、Pro*COBOL は SQLCODE 値 1403 を戻します。構成ファイルで MODE=ANSI と指定し、コマンド行で END_OF_FETCH=1403 と指定した場合も、END_OF_FETCH 状態が発生すると Pro*COBOL は 1403 を戻します。

Oracle8*i* が上記の状態のどれかを検出したとき、プログラムに次の処置のいずれかを実行させることができます。

アクション

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 文を使わない場合と同じです。状態のチェックを " オフ " にするのに使えます。

DO CALL

プログラムはネストされたサブプログラムをコールします。サブプログラムの最後に達すると、失敗した SQL 文の後の文に制御が渡されます。

DO PERFORM

プログラムは制御を COBOL 段落に渡します。段落の最後に達すると、失敗した SQL 文の後の文に制御が渡されます。

```
EXEC SQL
    WHENEVER <condition> DO PERFORM <paragraph_name>
END-EXEC.
```

GOTO または GO TO

プログラムはラベル付き文に分岐します。

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。

注意が必要です。STOP アクションでは、ログオフするまでメッセージは表示されません。

WHENEVER 文のコーディング

WHENEVER 文の構文は次のとおりです。

```
EXEC SQL
    WHENEVER <condition> <action>
END-EXEC.
```

DO PERFORM

WHENEVER...DO PERFORM 文の使用にあたっては、段落の PERFORM に関する通常の規則が適用されます。ただし、THRU 句、TIMES 句、UNTIL 句、VARYING 句は使用できません。

たとえば、次の WHENEVER...DO 文は無効です。

```
 PROCEDURE DIVISION.
*    Invalid statement
     EXEC SQL WHENEVER SQLERROR DO
         PERFORM DISPLAY-ERROR THRU LOG-OFF
     END-EXEC.
     ...
 DISPLAY-ERROR.
     ...
 LOG-OFF.
     ...
```

次に示すのは、WHENEVER SQLERROR DO PERFORM 文を使用して特定のエラーを処理する例です。

```
 PROCEDURE DIVISION.
 MAIN.
     ...
     EXEC SQL
         WHENEVER SQLERROR DO PERFORM INS-ERROR
     END-EXEC.
     EXEC SQL
         INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
         VALUES (:EMP-NUMBER, :EMP-NAME, :DEPT-NUMBER)
     END-EXEC.
     EXEC SQL
         WHENEVER SQLERROR DO PERFORM DEL-ERROR
     END-EXEC.
     EXEC SQL
         DELETE FROM DEPT
         WHERE DEPTNO = :DEPT-NUMBER
     END-EXEC.
     ...
*    Error-handling paragraphs.
 INS-ERROR.
*    Check for "duplicate key value" Oracle8 error
     IF SQLCA.SQLCODE = -1
     ...
*    Check for "value too large" Oracle8 error
     ELSE IF SQLCA.SQLCODE = -1401
     ...
     ELSE
```

```
     ...
     END-IF.
     ...
 DEL-ERROR.
*    Check for the number of rows processed.
     IF SQLCA.SQLERRD(3) = 0
     ...
     ELSE
     ...
     END-IF.
     ...
```

各段落でどのように SQLCA 内の変数をチェックし、実行する処理を決定しているかに注意してください。

DO CALL

この句はアクション・サブプログラムをコールします。句の構文を次に示します。

```
     EXEC SQL
         WHENEVER <condition> DO CALL <subprogram_name>
         [USING <param1> ...]
     END-EXEC.
```

次の制限または規則が適用されます。

- USING 句では、RETURNING、ON_EXCEPTION または OVER_FLOW 句を使用できません。
- COBOL ソース・コードの PROGRAM-ID 文で、キーワード COMMON の前にサブプログラム名を入力しなければならない場合があります。
- アクション・サブプログラムでは WHENEVER CONTINUE 文を使う必要があります。
- WHENEVER ディレクティブの DO CALL 句では、アクション・サブプログラム名を二重引用符で囲まなければならない場合があります。

次に示す例は、サブプログラム LOGON 内または MAIN プログラム内からエラー・サブプログラム SQL-ERROR をコールできるプログラムの例です。DO PERFORM 句を使用したときと同様、2 つの位置でコードが繰り返されることはありません。

```
IDENTIFICATION DIVISION.
            PROGRAM-ID. MAIN.
            ENVIRONMENT DIVISION.
        ...
            PROCEDURE DIVISION.
            BEGIN-PGM.
                EXEC SQL
                    WHENEVER SQLERROR DO CALL "SQL-ERROR"
```

```
        END-EXEC.
        CALL "LOGON".
...
    IDENTIFICATION DIVISION.
    PROGRAM-ID. LOGON.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01  USERNAME          PIC X(15) VARYING.
    01  PASSWD            PIC X(15) VARYING.
    PROCEDURE DIVISION.
        MOVE "SCOTT" TO USERNAME-ARR.
        MOVE 5 TO USERNAME-LEN.
        MOVE "TIGER" TO PASSWD-ARR.
        MOVE 5 TO PASSWD-LEN.
        EXEC SQL
           CONNECT :USERNAME IDENTIFIED BY :PASSWD
        END-EXEC.
        DISPLAY " ".
        DISPLAY "CONNECTED TO ORACLE AS USER:  ", USERNAME-ARR.
    END PROGRAM LOGON.
...
    IDENTIFICATION DIVISION.
    PROGRAM-ID. SQL-ERROR COMMON.
    PROCEDURE DIVISION.
        EXEC SQL
            WHENEVER SQLERROR CONTINUE
        END-EXEC.
        DISPLAY " ".
        DISPLAY SQLERRMC.
        EXEC SQL
            ROLLBACK WORK RELEASE
        END-EXEC.
        END PROGRAM SQL-ERROR.
    END PROGRAM MAIN.
```

有効範囲

WHENEVER 文は宣言文のため、その有効範囲は論理的なものではなく位置的なものになります。WHENEVER 文がテストするのは、ソース・ファイルの中でその WHENEVER 文より後に記述されているすべての実行 SQL 文であって、プログラム論理の流れの中でその WHENEVER 文それより後にくる実行 SQL 文ではありません。したがって WHENEVER 文は、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。

WHENEVER 文は、同じ状態をチェックする別の WHENEVER 文に置き換えられるまでの間は有効です。

提案 : SQL 文を含む各プログラム・ユニットの先頭に WHENEVER 文を記述してください。このようにすると、あるプログラム・ユニット内の SQL 文が別のプログラム・ユ

ニット内の WHENEVER の処置を参照することによって、コンパイル時や実行時に発生するエラーを回避できます。

不注意な使用方法 : 例

WHENEVER 文を不注意に使用すると、問題が発生することがあります。たとえば、次のコードは、検索条件を満たす行がないため、DELETE 文で NOT FOUND 状態を設定すると無限ループに陥ります。

```
*    Improper use of WHENEVER.
     EXEC SQL
         WHENEVER NOT FOUND GOTO NO-MORE
     END-EXEC.
     PERFORM GET-ROWS UNTIL DONE = "YES".
     ...
 GET-ROWS.
     EXEC SQL
         FETCH EMP-CURSOR INTO :EMP-NAME, :SALARY
     END-EXEC.
     ...
 NO-MORE.
     MOVE "YES" TO DONE.
     EXEC SQL
         DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
     ...
```

次の例では、GOTO のターゲットを設定しなおすことによって NOT FOUND 状態を適切に処理しています。

```
*    Proper use of WHENEVER.
     EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.
     PERFORM GET-ROWS UNTIL DONE = "YES".
     ...
 GET-ROWS.
     EXEC SQL
         FETCH EMP-CURSOR INTO :EMP-NAME, :SALARY
     END-EXEC.
     ...
 NO-MORE.
     MOVE "YES" TO DONE.
     EXEC SQL WHENEVER NOT FOUND GOTO NONE-FOUND END-EXEC.
     EXEC SQL
        DELETE FROM EMP WHERE EMPNO = :EMP-NUMBER
     END-EXEC.
     ...
 NONE-FOUND.
     ...
```

SQL 文のテキストの取得

多くの Pro*COBOL アプリケーションでは、処理している文のテキストおよび長さ、その文に記述されている SQL コマンド（INSERT や SELECT など）がわかると便利です。これは、動的 SQL を使うアプリケーションでは特に重要です。

ルーチン SQLGLS を使用すると、次の情報が戻されます。（このルーチンは SQLLIB ランタイム・ライブラリに入っています。）

- 最後に解析された SQL 文のテキスト
- その SQL 文の長さ
- 文に使われる SQL コマンドの関数コード（8-1 ページの表 8-1 の「ASSUME_SQLCODE=NO、MODE=ANSI | ANSI14、DECLARE_SECTION=YES のときの状態変数の動作」を参照）

SQLGLS は、静的 SQL 文の発行後にコールできます。動的 SQL 方法 1 では、SQL 文の実行後に SQLGLS をコールできます。動的 SQL 方法 2 または 3、4 では、SQL 文の作成後に SQLGLS をコールできます。

SQLGLS をコールするための構文は次のとおりです。

```
CALL "SQLGLS" USING SQLSTM STMLEN SQLFC.
```

表 8-6 に、SQLGLS 引数リストのパラメータに使用可能なホスト言語のデータ型を示します。

表 8-6 パラメータのデータ型

パラメータ	データ型
SQLSTM	PIC X(*n*)
STMLEN	PIC S9(9) COMP
SQLFC	PIC S9(9) COMP

パラメータはすべて、参照によって渡さなければなりません。通常、デフォルトのパラメータ引渡し規則は参照による引渡しになっているので、特別な処置は必要ありません。

パラメータ SQLSTM は、SQL 文の戻されたテキストを保持する空白埋め（NULL で終わるのではない）文字バッファです。プログラムでは、このバッファを静的に宣言するか、このバッファに動的にメモリーを割り当てる必要があります。

長さを指定するパラメータ STMLEN は、4 バイトの整数です。SQLGLS をコールする前に、このパラメータを SQLSTM バッファの実際のサイズ（バイト数）に設定してください。SQLGLS が戻されると、SQLSTM バッファには SQL 文のテキストが格納され、空き部分には空白が埋め込まれています。STMLEN は、戻された文のテキストでの実際のバイト数を

戻します。（埋め込まれた空白は数えません。）ただし、エラーが発生した場合は、STMLEN は 0（ゼロ）を戻します。

発生する可能性のあるエラーは次のとおりです。

- SQL 文が 1 つも解析されませんでした。
- 無効なパラメータを渡しました（たとえば、長さとして負の値を渡した場合）。
- SQLLIB で内部例外が発生しました。

パラメータ SQLFC は、文中の SQL コマンドに対応する SQL 関数コードを戻す 4 バイトの整数です。SQL コマンドの関数コードの詳細は、『Oracle8*i* コール・インタフェース・プログラマーズ・ガイド』の表を参照してください。

次の文には、対応する SQL 関数コードはありません。

- CONNECT
- COMMIT
- FETCH
- ROLLBACK
- RELEASE

Oracle 通信領域の使用

SQLCA は、標準的な SQL の通信を対象としています。Oracle 通信領域（ORACA）も SQLCA と同様の構造ですが、ORACA をプログラムに組み込むと Oracle8*i* 固有の通信を処理できるようになります。SQLCA で提供される情報よりも詳しい実行時情報が必要な場合に、ORACA を使用してください。

ORACA は、問題の診断に役立つだけでなく、SQL 文エグゼキュータやカーソル・キャッシュ（カーソル管理のために確保されているメモリー領域）などのリソースをプログラムがどのように使用しているかを監視する場合にも使えます。

ORACA 内の情報

ORACA には、オプションの設定、システム統計情報および拡張診断情報が格納されます。図 8-3 に、ORACA 内のすべての変数を示します。

図 8-3 Pro*COBOL の ORACA 変数宣言

```
ORACA

01   ORACA
     05      ORACAID PIC X(8).
     05      ORACABC PIC S9(9) COMP.
     05      ORACCHF PIC S9(9) COMP.
     05      ORADBGF PIC S9(9) COMP.
     05      ORAHCHF PIC S9(9) COMP.
     05      ORASTXTF                     PIC S9(9) COMP.
     05      ORASTXT.
             49      ORASTXTL             PIC S9(4) COMP.
             49      ORASTXTL             PIC X(70)
     05      ORASFNM.
             49      ORASFNML             PIC S9(4) COMP.
             49      ORASFNMC             PIC X(70)
     05      ORASLNR PIC X(8).
     05      ORAHOC  PIC S9(9) COMP.
     05      ORAMOC  PIC S9(9) COMP.
     05      ORACOC  PIC S9(9) COMP.
     05      ORANOR  PIC S9(9) COMP.
     05      ORANPR  PIC S9(9) COMP.
     05      ORANEX  PIC S9(9) COMP.
```

ORACA の宣言

ORACA を宣言するには、次に示すように、（EXEC SQL INCLUDE 文を使って）Pro*COBOL ソース・ファイルの宣言文の外に組み込みます。

```
*    Include the Oracle Communications Area (ORACA).
     EXEC SQL INCLUDE ORACA END-EXEC.
```

ORACA を使用可能にする

ORACA を使用可能にするには、ORACA プリコンパイラ・オプションを YES に設定しなければなりません。これには、コマンド行または構成ファイルに次のように入力します。

```
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) END-EXEC.
```

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。ORACA を使用可能にすると実行時のオーバーヘッドが増加することになるため、ORACA の使用は任意になっています。デフォルトの設定は ORACA=NO です。

ランタイム・オプションの選択

ORACA にはいくつかのオプション・フラグがあります。これらのフラグを設定するには、オプション・フラグに 0（ゼロ）以外の値を割り当てます。オプション・フラグの設定により、次の処理を行えます。

- 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を指します。）
- ヒープの一貫性チェック（ヒープとは動的変数のために予約されるメモリー領域を指します。）
- 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。

ORACA の構造

この項では、ORACA の構造およびフィールド、フィールドに格納できる値について説明します。

ORACAID

この文字列フィールドは、Oracle 通信領域を示す "ORACA" に初期化されます。

ORACABC

この整数フィールドには、ORACA データ構造の長さ（バイト数）が格納されています。

ORACCHF

マスター DEBUG フラグ（ORADBGF）が設定されているときにこのフラグを設定すると、カーソル操作のたびにカーソル・キャッシュの一貫性をあらかじめチェックできます。

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・メッセージの一覧は、『Oracle8*i* エラー・メッセージ』を参照してください。

このフラグは次のいずれかを設定します。

0　　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）。

1　　キャッシュ一貫性チェックを使用可能にします。

ORADBGF

このマスター・フラグを使うと、DEBUG オプションをすべて選択できます。このフラグには次のいずれかを設定します。

0　　すべての DEBUG 処理を使用禁止にします（デフォルト）。

1　　すべての DEBUG 処理を使用可能にします。

ORAHCHF

マスター DEBUG フラグ（ORADBGF）が設定されているときにこのフラグを設定することにより、Pro*COBOL が動的にメモリーの割当てや解放を行うたびにランタイム・ライブラリを使ってヒープの一貫性をチェックできます。これはメモリー障害を起こすプログラムのバグ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。このフラグには次のいずれかを設定します。

0　　すべての DEBUG 処理を使用禁止にします（デフォルト）。

1　　すべての DEBUG 処理を使用可能にします。

ORASTXTF

このフラグを使うと、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。このフラグには次のいずれかを設定します。

0　　SQL 文のテキストを保存しません（デフォルト）。

1　　SQLERROR の SQL 文のテキストだけを保存します。

2　　SQLERROR または SQLWARNING の SQL 文のテキストだけを保存します。

3　　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 サブレコードに保存されます。

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。

ORASTXT

このサブレコードは、問題のある SQL 文を見つける場合に役立ちます。Oracle8*i* で解析された最後の SQL 文のテキストを保存できます。このサブレコードには、次の 2 つのフィールドがあります。

ORASTXTL	この整数フィールドには、現行の SQL 文の長さが格納されます。
ORASTXTC	この文字列フィールドには、現行の SQL 文のテキストが格納されます。先頭から最長 70 文字分のテキストが保存されます。

Pro*COBOL によって解析される文（CONNECT、FETCH、COMMIT など）は、ORACA には保存されません。

ORASFNM

このサブレコードによって、現行の SQL 文が入っているファイルを識別できます。これにより、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した場合にエラーを見つけやすくなります。このサブレコードには、次の 2 つのフィールドがあります。

ORASFNML	この整数フィールドには、ORASFNMC に格納されているファイル名の長さが格納されます。
ORASFNMC	この文字列フィールドには、ファイル名が格納されます。先頭から最長 70 文字分が保存されます。

ORASLNR

この整数フィールドにより、現行の SQL 文が記述されている行（またはその近くの行）を識別できます。

カーソル・キャッシュ統計情報

次に説明する一連の変数を使用して、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 または ROLLBACK 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 の現在の設定値は、最後にコミットまたはロールバックされたデータベースに関する値です。

ORAHOC

この整数フィールドには、プログラムの実行中に MAXOPENCURSORS に設定された最大の値が記録されます。

ORAMOC

この整数フィールドには、プログラムの要求によってオープンされたカーソルの最大数が記録されます。MAXOPENCURSORS の設定が低すぎると、この値が ORAHOC の値を上回ることがあります。この場合、Pro*COBOL によってカーソル・キャッシュが強制的に拡張されます。

ORACOC

この整数フィールドには、プログラムの要求によって現在オープンしているカーソル数が記録されます。

ORANOR

この整数フィールドには、プログラムの要求によって再割当てされたカーソル・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し、できるだけ低く抑える必要があります。

ORANPR

この整数フィールドには、プログラムの要求によって解析された SQL 文の数が記録されます。

ORANEX

この整数フィールドには、プログラムの要求によって実行された SQL 文の数が記録されます。この数値の ORANPR に対する割合は、できる限り高く保たなければなりません。つまり、不要な再解析は回避しなければなりません。ヘルプについては、付録 D の「パフォーマンス・チューニング」を参照してください。

ORACA の例

次のプログラムは、部門番号の入力を要求し、その部門に所属している各従業員の名前と給料を 2 つの表のいずれかに挿入して、ORACA からの診断情報を表示します。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ORACAEX.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    EXEC SQL INCLUDE SQLCA END-EXEC.
    EXEC SQL INCLUDE ORACA END-EXEC.
```

```
    EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 USERNAME       PIC X(20).
 01 PASSWORD       PIC X(20).
 01 EMP-NAME       PIC X(10) VARYING.
 01 DEPT-NUMBER    PIC S9(4) COMP.
 01 SALARY         PIC S9(6)V99
                         DISPLAY SIGN LEADING SEPARATE.
    EXEC SQL END DECLARE SECTION END-EXEC.

 PROCEDURE DIVISION.
    DISPLAY "Username? " WITH NO ADVANCING.
    ACCEPT USERNAME.
    DISPLAY "Password? " WITH NO ADVANCING.
    ACCEPT PASSWORD.
    EXEC SQL
        WHENEVER SQLERROR GOTO SQL-ERROR
    END-EXEC.
    EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWORD
    END-EXEC.
    DISPLAY "Connected to Oracle".

* -- set flags in the ORACA
* -- enable debug operations
    MOVE 1 TO ORADBGF.
* -- enable cursor cache consistency check
    MOVE 1 TO ORACCHF.
* -- always save the SQL statement
    MOVE 3 TO ORASTXTF.
    DISPLAY "Department number? " WITH NO ADVANCING.
    ACCEPT DEPT-NUMBER.
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
        SELECT ENAME, SAL + NVL(COMM,0)
        FROM EMP
        WHERE DEPTNO = :DEPT-NUMBER
    END-EXEC.
    EXEC SQL OPEN EMPCURSOR END-EXEC.
    EXEC SQL
        WHENEVER NOT FOUND GOTO NO-MORE
    END-EXEC.
 LOOP.
    EXEC SQL
        FETCH EMPCURSOR INTO :EMP-NAME, :SALARY
    END-EXEC.
```

```
    IF SALARY < 2500
        EXEC SQL
            INSERT INTO PAY1 VALUES (:EMP-NAME, :SALARY)
        END-EXEC
    ELSE
        EXEC SQL
            INSERT INTO PAY2 VALUES (:EMP-NAME, :SALARY)
        END-EXEC
    END-IF.
    GO TO LOOP.

NO-MORE.
    EXEC SQL CLOSE EMPCURSOR END-EXEC.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL COMMIT WORK RELEASE END-EXEC.
    DISPLAY "(NO-MORE.) Last SQL statement: ", ORASTXTC.
    DISPLAY "... at or near line number:  ", ORASLNR.
    DISPLAY  " ".
    DISPLAY "          Cursor Cache Statistics".
    DISPLAY "-----------------------------------------".
    DISPLAY "Maximum value of MAXOPENCURSORS     ", ORAHOC.
    DISPLAY "Maximum open cursors required:      ", ORAMOC.
    DISPLAY "Current number of open cursors:     ", ORACOC.
    DISPLAY "Number of cache reassignments:      ", ORANOR.
    DISPLAY "Number of SQL statement parses:     ", ORANPR.
    DISPLAY "Number of SQL statement executions: ", ORANEX.
    STOP RUN.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
    DISPLAY "(SQL-ERROR.) Last SQL statement: ", ORASTXTC.
    DISPLAY "... at or near line number:  ", ORASLNR.
    DISPLAY " ".
    DISPLAY "          Cursor Cache Statistics".
    DISPLAY "-----------------------------------------".
    DISPLAY "MAXIMUM VALUE OF MAXOPENCURSORS     ", ORAHOC.
    DISPLAY "Maximum open cursors required:      ", ORAMOC.
    DISPLAY "Current number of open cursors:     ", ORACOC.
    DISPLAY "Number of cache reassignments:      ", ORANOR.
    DISPLAY "Number of SQL statement parses:     ", ORANPR.
    DISPLAY "Number of SQL statement executions: ", ORANEX.
    STOP RUN.
```

9

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動的 SQL の使用方法について説明します。動的 SQL の長所と短所を比較した後、実行時に SQL 文をその場で受け入れて処理するプログラムを記述する方法を、単純なものから複雑なものまで 4 つ紹介します。それぞれの方法の必要条件および制限事項、さらに実行するジョブに対する適切な方法の選択方法についても説明します。

この章の構成は、次のとおりです。

動的 SQL とは

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 という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、さまざまな SQL 文を実行時に受け入れ（または作成し）、処理しなければならないものもあります。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 文といいます。

静的 SQL 文と違って、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だけでなく、ファイルから読み込むこともできます。

動的 SQL の長所と短所

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 句内で使う検索条件の入力をユーザーに求めるという単純なプログラムがあります。さらにプログラムが複雑になると、SQL 処理、表およびビューの名前、列の名前などが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 を使うと柔軟性に富んだアプリケーションを記述できるようになります。

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、実行時の処理時間の増加が必要になるものもあります。処理時間が増えるのは特に気にならないかもしれませんが、動的 SQL の概念および技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。

動的 SQL を使用する場合

実際は静的 SQL によって、プログラミング要件のほど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だけ使用してください。動的 SQL の使用が望ましいのは、次の項目の中に、プリコンパイル時に不明なものが 1 つ以上ある場合です。

- SQL 文のテキスト（コマンド、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、ビューなど）

動的 SQL 文の要件

動的 SQL 文を表す文字列には、有効な DML SQL 文または DDL SQL 文のテキストを記述しなければなりませんが、EXEC SQL 句またはホスト言語のデリミタや文終了記号は記述しません。

ほとんどの場合、この文字列にはダミーのホスト変数を格納できます。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ただのプレースホルダ（場所を確保するもの）なので、宣言する必要はなく、任意の名前を付けられます （ハイフンは使用できません）。たとえば、Oracle8*i* では次の 2 つの文字列は区別されません。

```
'DELETE FROM EMP WHERE MGR = :MGRNUMBER AND JOB = :JOBTITLE'
'DELETE FROM EMP WHERE MGR = :M AND JOB = :J'
```

動的 SQL 文の処理

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うホスト変数の値をユーザーが入力する必要があります。作成された SQL 文は Oracle8*i* によって解析されます。解析では、SQL 文が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について調べられます。また、データベース・アクセス権限のチェック、必要なリソースの確保および最適なアクセス・パスの検索も行われます。

次に、Oracle8*i* はホスト変数を SQL 文にバインドします。これにより、Oracle8*i* はホスト変数のアドレスを取得し、その値の読込みや書込みを実行できるようになります。

この後、Oracle8*i* によって 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 Oracle8*i* 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によって、この SQL 文を繰り返し実行できます。

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と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。ここで紹介する方法の使用方法は、後の項で説明します。

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 9-1 に示すように、それぞれの方法は特定の種類の SQL 文を処理する場合に最も役立ちます。

表 9-1 適切な使用方法

方法	SQL 文の種類
1	入力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と入力ホスト変数の数がわかっている問合せ
4	選択リスト項目の数または入力ホスト変数の数が不明な問合せ

選択リスト項目には、列名や式を使用します。

方法 1

この方法を使うと、動的 SQL 文を受け入れ（または作成し）、EXECUTE IMMEDIATE コマンドを使っ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、入力ホスト変数の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（HOLD_CURSOR=YES と指定した場合は除く）。

方法 2

この方法を使うと、動的 SQL 文を受け入れ（または作成し）、PREPARE および EXECUTE コマンドを使っ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。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はプリコンパイル時にわかっていなければなりません。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 (ENAME, JOB) VALUES (:EMPNAME, :JOBTITLE)'
'DELETE FROM EMP WHERE EMPNO = :EMPNUMBER'
```

方法 2 では、（RELEASE_CURSOR=YES と指定した場合を除いて）SQL 文の解析は 1 回だけしか行われませんが、ホスト変数の値を変えれば同じ SQL 文を何回でも実行できます。CREATE などの SQL データ定義文は、PREPARE 時に実行されます。

方法 3

この方法を使うと、動的問合せを受け入れ（または作成し）、DECLARE、OPEN、FETCH、CLOSE カーソル・コマンドとともに PREPARE コマンドを使ってその問合せを処理できま

す。選択リスト項目の数、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は、プリコンパイル時にわかっていなければなりません。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 = :DEPTNUMBER'
```

方法 4

この方法では、プログラムは動的 SQL 文の受け入れまたは作成を行い、記述子（9-23 ページの「方法 4 の使用方法」を参照）を使って処理します。選択リスト項目の数、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。

ガイドライン

4 つの方法はいずれも、動的 SQL 文を文字列に格納する必要があります。このとき指定する文字列は、ホスト変数または引用符で囲んだリテラルでなければなりません。SQL 文を文字列に格納する際には、キーワード EXEC SQL および文終了記号は省略してください。

方法 2 と方法 3 では、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はプリコンパイル時にわかっていなければなりません。

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、コードの記述は難しくなります。原則として、できるだけ簡単な方法を使ってください。ただし、動的 SQL 文を繰り返し実行する場合は、方法 1 を使うと実行のたびに再解析されるので、これを避けるため方法 2 を使用してください。

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だけです。

9-7 ページの図 9-1 の「正しい方法の選択」を参考にして、適切な方法を選択してください。

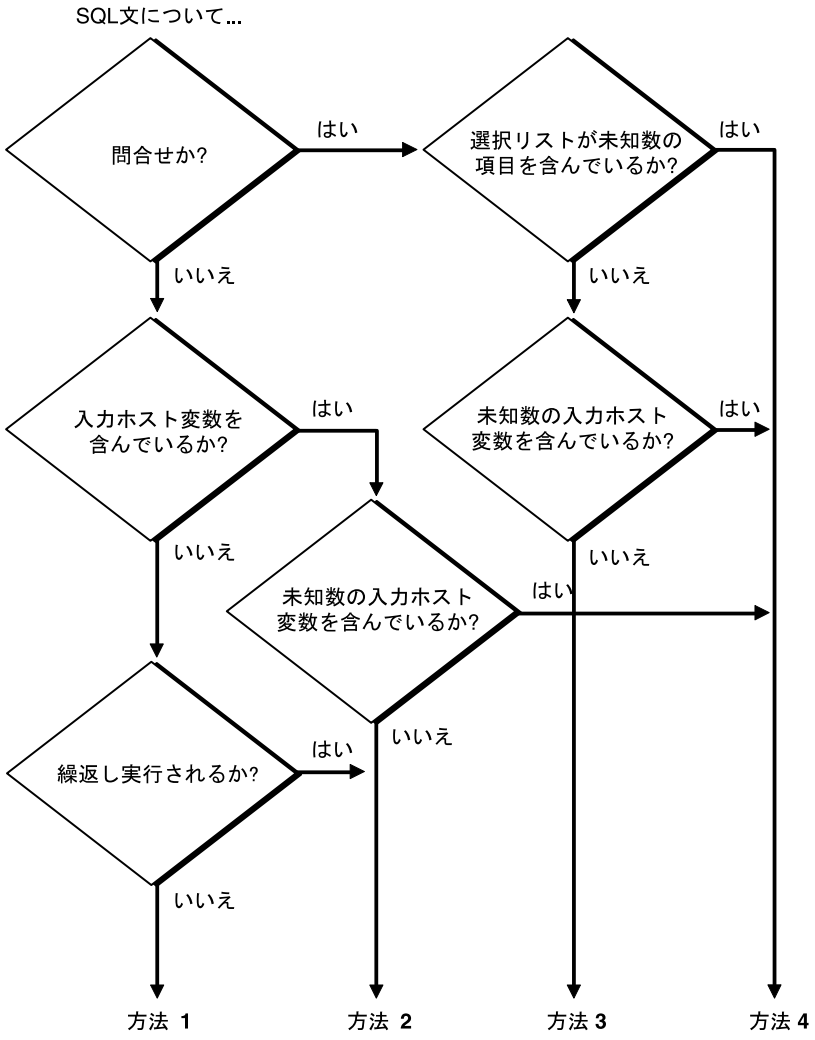
一般的なエラーの回避

動的 SQL 文を文字配列に格納する場合は、その配列に空白を埋め込んでから SQL 文を格納してください。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特に重要となります。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 NULL 終了記号を使用しないでください。Oracle8*i* では、NULL 終了記号は文字列の終了標識とはみなされず、SQL 文の一部として扱われます。

動的 SQL 文を VARCHAR 変数に格納する場合は、VARCHAR 変数の長さを正しく設定（または再設定）してから PREPARE 文および EXECUTE IMMEDIATE 文を実行してください。EXECUTE を実行すると、SQLCA の SQLWARN 警告フラグはリセットされます。したがって、無条件更新（これは WHERE 句を指定しなかった場合に発生します）などの誤りを検出するには、PREPARE 文を実行してから EXECUTE 文を実行するまでの間に SQLWARN フラグをチェックする必要があります。

図 9-1 正しい方法の選択



方法 1 の使用方法

最も単純な種類の動的 SQL 文の結果は " 成功 " または " 失敗 " だけです。このときホスト変数は使用されません。次にいくつかの例を示します。

```
'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'
```

EXECUTE IMMEDIATE 文

方法 1 では、SQL 文を解析すると、EXECUTE IMMEDIATE コマンドを使ってその文をすぐに実行します。コマンドには実行用の 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。この文は問合せであってはなりません。

EXECUTE IMMEDIATE 文の構文は次のとおりです。

```
EXEC SQL EXECUTE IMMEDIATE { :HOST-STRING | STRING-LITERAL }END-EXEC.
```

次の例では、ユーザーが入力する SQL 文をホスト変数 *SQL-STMT* に格納しています。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
 01   SQL-STMT  PIC X(120);
      EXEC SQL END DECLARE SECTION END-EXEC.
     ...
 LOOP.
     DISPLAY 'Enter SQL statement: ' WITH NO ADVANCING.
     ACCEPT SQL-STMT END-EXEC.
* --  sql_stmt now contains the text of a SQL statement
     EXEC SQL EXECUTE IMMEDIATE :SQL-STMT END-EXEC.
 NEXT.
     ...
```

次の例で示すように、文字列リテラルを使ってもかまいません。

```
     EXEC SQL
         EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER'
     END-EXEC.
```

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。通常、データ定義文は、このカテゴリに入ります。

例

次のプログラムは、UPDATE 文の WHERE 句で使用する検索条件の入力をユーザーに求め、方法 1 を使って UPDATE 文を実行します。

```
     ...
*    THE RELEASE_CURSOR=YES OPTION INSTRUCTS PRO*COBOL TO
*    RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
*    STATEMENTS.  THIS ENSURES THAT Oracle8 DOES NOT KEEP PARSE
*    LOCKS ON TABLES, SO THAT SUBSEQUENT DATA MANIPULATION
*    OPERATIONS ON THOSE TABLES DO NOT RESULT IN PARSE-LOCK
*    ERRORS.

     EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

*
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10) VALUE "SCOTT".
 01  PASSWD    PIC X(10) VALUE "TIGER".
 01  DYNSTMT   PIC X(80).
     EXEC SQL END DECLARE SECTION END-EXEC.
 01  UPDATESTMT PIC X(40).
 01  SEARCH-COND PIC X(40).
     ...
     DISPLAY "ENTER A SEARCH CONDITION FOR STATEMENT:".
     MOVE "UPDATE EMP SET COMM = 500 WHERE " TO UPDATESTMT.
     DISPLAY UPDATESTMT.
     ACCEPT SEARCH-COND.
*    Concatenate SEARCH-COND to UPDATESTMT and store result
*    in DYNSTMT.
     STRING UPDATESTMT DELIMITED BY SIZE
         SEARCH-COND DELIMITED BY SIZE INTO DYNSTMT.
     EXEC SQL EXECUTE IMMEDIATE :DYNSTMT END-EXEC.
```

サンプル・プログラム 6: 動的 SQL 方法 1

このプログラムは、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、表の削除を行います。

```
*****************************************************************
* Sample Program 6:  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.         *
*****************************************************************
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DYNSQL1.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

*    INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
*    THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
*    INFORMATION AVAILABLE TO THE PROGRAM.

     EXEC SQL INCLUDE SQLCA END-EXEC.

*    INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE
*    THROUGH WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS
*    INFORMATION AVAILABLE TO THE PROGRAM.

     EXEC SQL INCLUDE ORACA END-EXEC.

*    THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
*    THE ORACA.

     EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

*    THE RELEASE_CURSOR=YES OPTION INSTRUCTS PRO*COBOL TO
*    RELEASE IMPLICIT CURSORS ASSOCIATED WITH EMBEDDED SQL
*    STATEMENTS.  THIS ENSURES THAT ORACLE DOES NOT KEEP PARSE
*    LOCKS ON TABLES, SO THAT SUBSEQUENT DATA MANIPULATION
*    OPERATIONS ON THOSE TABLES DO NOT RESULT IN PARSE-LOCK
*    ERRORS.

     EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10) VALUE "SCOTT".
 01  PASSWD    PIC X(10) VALUE "TIGER".
 01  DYNSTMT   PIC X(80) VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.

*    DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
 01  ORASLNRD  PIC 9(9).

 PROCEDURE DIVISION.

 MAIN.

*    BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
     EXEC SQL WHENEVER SQLERROR GOTO SQLERROR END-EXEC.
```

```
*    SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
*    OCCURS.
     MOVE 1 TO ORASTXTF.

*    CONNECT TO ORACLE.
     EXEC SQL
         CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY " ".
     DISPLAY "CONNECTED TO ORACLE AS USER:  " WITH NO ADVANCING.
     DISPLAY USERNAME.
     DISPLAY " ".

*    EXECUTE A STRING LITERAL TO CREATE THE TABLE.  HERE, YOU
*    GENERALLY USE A STRING VARIABLE INSTEAD OF A LITERAL, AS IS
*    DONE LATER IN THIS PROGRAM.  BUT, YOU CAN USE A LITERAL IF
*    YOU WISH.
     DISPLAY "CREATE TABLE DYN1 (COL1 CHAR(4))".
     DISPLAY " ".
     EXEC SQL EXECUTE IMMEDIATE
         "CREATE TABLE DYN1 (COL1 CHAR(4))"
     END-EXEC.

*    ASSIGN A SQL STATEMENT TO THE VARYING STRING DYNSTMT.
*    SET THE -LEN PART TO THE LENGTH OF THE -ARR PART.
     MOVE "INSERT INTO DYN1 VALUES ('TEST')" TO DYNSTMT-ARR.
     MOVE 36 TO DYNSTMT-LEN.
     DISPLAY DYNSTMT-ARR.
     DISPLAY " ".

*    EXECUTE DYNSTMT TO INSERT A ROW.  THE SQL STATEMENT IS A
*    STRING VARIABLE WHOSE CONTENTS THE PROGRAM MAY DETERMINE
*    AT RUN TIME.
     EXEC SQL EXECUTE IMMEDIATE :DYNSTMT END-EXEC.

*    COMMIT THE INSERT.
     EXEC SQL COMMIT WORK END-EXEC.

*    CHANGE DYNSTMT AND EXECUTE IT TO DROP THE TABLE.
     MOVE "DROP TABLE DYN1" TO DYNSTMT-ARR.
     MOVE 19 TO DYNSTMT-LEN.
     DISPLAY DYNSTMT-ARR.
     DISPLAY " ".
     EXEC SQL EXECUTE IMMEDIATE :DYNSTMT END-EXEC.

*    COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
```

```
      EXEC SQL COMMIT RELEASE END-EXEC.
      DISPLAY "HAVE A GOOD DAY!".
      DISPLAY " ".
      STOP RUN.

  SQLERROR.

*     ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING
*     ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
      DISPLAY SQLERRMC.
      DISPLAY "IN ", ORASTXTC.
      MOVE ORASLNR TO ORASLNRD.
      DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*     DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*     SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

*     ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
      EXEC SQL ROLLBACK RELEASE END-EXEC.
      STOP RUN.
```

方法 2 の使用方法

方法 1 では 1 段階で実行し、方法 2 では 2 段階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 文の中にホスト変数および標識変数のプレースホルダを使用できます。この SQL 文は 1 度 PREPARE すれば、ホスト変数に別の値を指定して繰り返し EXECUTE できます。COMMIT または ROLLBACK の後で SQL 文を再度 PREPARE する必要はありません（ログオフして再接続する場合は除く）。

方法 4 では、非問合せに EXECUTE を使うことができるので注意してください。

PREPARE 文の構文は次のとおりです。

```
EXEC SQL PREPARE <STATEMENT-NAME>
    FROM { :<HOST-STRING> | <STRING-LITERAL> }
END-EXEC.
```

PREPARE は、この SQL 文を解析して名前を指定します。

STATEMENT-NAME はプリコンパイラが使用する識別子です。これはホスト変数でもプログラム変数でもないので、COBOL 文の中では宣言しないでください。これは EXECUTE の対象として PREPARE した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>]
    END-EXEC.
```

HOST-VARIABLE-LIST の構文を次に示します。

```
:<HOST-VAR1>[:<INDICATOR1>] [, <HOST-VAR2>[:<INDICATOR2>], ...]
```

解析した SQL 文は、それぞれの入力ホスト変数に指定済みの値を使って EXECUTE によって実行されます。次の例では、入力 SQL 文にプレースホルダ *n* が組み込まれています。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
 01  EMP-NUMBER   PIC S9(4) COMP VALUE ZERO.
     ...
 01  DELETE-STMT   PIC X(120) VALUE SPACES.
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
 01  WHERE-STMT    PIC X(40).
 01  SEARCH-COND   PIC X(40).
     ...
     MOVE 'DELETE FROM EMP WHERE EMPNO = :N AND ' TO WHERE-STMT.
     DISPLAY 'Complete this statement's search condition:'.
     DISPLAY WHERE-STMT.
     ACCEPT SEARCH-COND.
*    Concatenate SEARCH-COND to WHERE-STMT and store in DELETE-STMT
     STRING WHERE-STMT DELIMITED BY SIZE
         SEARCH-COND DELIMITED BY SIZE INTO
         DELETE-STMT.
     EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.
 LOOP.
     DISPLAY 'Enter employee number: ' WITH NO ADVANCING.
     ACCEPT EMP-NUMBER.
     IF EMP-NUMBER = 0
         GO TO NEXT.
     EXEC SQL EXECUTE SQLSTMT USING :EMP-NUMBER END-EXEC.
 NEXT.
```

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なければなりません。最後の例では、*EMP-NUMBER* が PIC S9(4) COMP 型で宣言されています Oracle8*i* では、PIC S9(4) COMP 型、PIC X(4) 型、PIC S9(4) COMP-1 型から内部 NUMBER 型へのデータ型変換がすべてサポートされているので、入力ホスト変数を PIC X(4) 型または PIC S9(4) COMP-1 型で宣言することもできます。

USING 句

SQL 文を EXECUTE すると、PREPARE された動的 SQL 文中のプレースホルダが USING 句の対応する入力ホスト変数に置き換えられます。

PREPARE された動的 SQL 文中のプレースホルダはすべて、USING 句のホスト変数に対応していなければなりません。このため、PREPARE 済みの文で同じプレースホルダが複数回使用されている場合は、それぞれが USING 句の中のホスト変数に対応している必要があります。USING 句のホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列でなければなりません。

プレースホルダの名前とホスト変数の名前が一致している必要はありません。ただし、PREPARE された動的 SQL 文中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していなければなりません。

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。詳細は 5-3 ページの「標識変数の使用方法」を参照してください。

サンプル・プログラム 7: 動的 SQL 方法 2

このプログラムは、動的 SQL 方法 2 を使用して、EMP 表に 2 行挿入し、挿入した行を削除します。

```
*****************************************************************
* Sample Program 7:  Dynamic SQL Method 2                       *
*                                                               *
* This program uses dynamic SQL Method 2 to insert two rows     *
* into the EMP table, then delete them.                         *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DYNSQL2.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

*    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.
     EXEC SQL INCLUDE SQLCA END-EXEC.

*    INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
*    WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
*    AVAILABLE TO THE PROGRAM.
     EXEC SQL INCLUDE ORACA END-EXEC.

*    THE OPTION ORACA=YES MUST BE SPECIFIED TO ENABLE USE OF
```

```
*    THE ORACA.
     EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10) VALUE "SCOTT".
 01  PASSWD    PIC X(10) VALUE "TIGER".
 01  DYNSTMT   PIC X(80) VARYING.
 01  EMPNO     PIC S9(4) COMPUTATIONAL VALUE 1234.
 01  DEPTNO1   PIC S9(4) COMPUTATIONAL VALUE 10.
 01  DEPTNO2   PIC S9(4) COMPUTATIONAL VALUE 20.
     EXEC SQL END DECLARE SECTION END-EXEC.

*    DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
 01  EMPNOD    PIC 9(4).
 01  DEPTNO1D  PIC 9(2).
 01  DEPTNO2D  PIC 9(2).
 01  ORASLNRD  PIC 9(9).

 PROCEDURE DIVISION.
 MAIN.

*    BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
     EXEC SQL WHENEVER SQLERROR GOTO SQLERROR END-EXEC.

*    SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
*    OCCURS.
     MOVE 1 TO ORASTXTF.

*    CONNECT TO ORACLE.
     EXEC SQL
         CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY " ".
     DISPLAY "CONNECTED TO ORACLE.".
     DISPLAY " ".

*    ASSIGN A SQL STATEMENT TO THE VARYING STRING 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.
     MOVE "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:V1, :V2)"
         TO DYNSTMT-ARR.
     MOVE 49 TO DYNSTMT-LEN.

*    DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
*    VARIABLES.
```

```
      DISPLAY DYNSTMT-ARR.
      MOVE EMPNO TO EMPNOD.
      MOVE DEPTNO1 TO DEPTNO1D.
      DISPLAY "    V1 = ", EMPNOD, "    V2 = ", DEPTNO1D.

*     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 MAY BE PREPARED MORE THAN ONCE,
*     OPTIONALLY FROM A DIFFERENT STRING VARIABLE.
      EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.

*     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 MAY
*     BE OMITTED ONLY IF THE STATEMENT CONTAINS NO PLACEHOLDERS.
*     A SINGLE PREPARED STATEMENT MAY BE EXECUTED MORE THAN ONCE,
*     OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES.
      EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO1 END-EXEC.

*     INCREMENT EMPNO AND DISPLAY NEW INPUT HOST VARIABLES.
      ADD 1 TO EMPNO.
      MOVE EMPNO TO EMPNOD.
      MOVE DEPTNO2 TO DEPTNO2D.
      DISPLAY "    V1 = ", EMPNOD, "    V2 = ", DEPTNO2D.

*     REEXECUTE S TO INSERT THE NEW VALUE OF EMPNO AND A
*     DIFFERENT INPUT HOST VARIABLE, DEPTNO2.  A REPREPARE IS NOT
*     NECESSARY.
      EXEC SQL EXECUTE S USING :EMPNO, :DEPTNO2 END-EXEC.

*     ASSIGN A NEW VALUE TO DYNSTMT.
      MOVE "DELETE FROM EMP WHERE DEPTNO = :V1 OR DEPTNO = :V2"
          TO DYNSTMT-ARR.
      MOVE 50 TO DYNSTMT-LEN.

*     DISPLAY THE NEW SQL STATEMENT AND ITS CURRENT INPUT HOST
*     VARIABLES.
      DISPLAY DYNSTMT-ARR.
      DISPLAY "    V1 = ", DEPTNO1D, "      V2 = ", DEPTNO2D.
```

```
*    REPREPARE S FROM THE NEW DYNSTMT.
     EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.

*    EXECUTE THE NEW S TO DELETE THE TWO ROWS PREVIOUSLY
*    INSERTED.
     EXEC SQL EXECUTE S USING :DEPTNO1, :DEPTNO2 END-EXEC.

*    ROLLBACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
     EXEC SQL ROLLBACK RELEASE END-EXEC.
     DISPLAY " ".
     DISPLAY "HAVE A GOOD DAY!".
     DISPLAY " ".
     STOP RUN.

  SQLERROR.
*    ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING
*    ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
     DISPLAY SQLERRMC.
     DISPLAY "IN ", ORASTXTC.
     MOVE ORASLNR TO ORASLNRD.
     DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*    DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*    SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

*    ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
     EXEC SQL ROLLBACK RELEASE END-EXEC.
     STOP RUN.
```

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。実際には、動的 SQL 文が問合せのときは方法 3 または方法 4 を必ず使います。

方法 3 では、問合せ選択リスト内の列の数と入力ホスト変数のプレースホルダの数がプリコンパイル時にわかっていなければなりません。ただし、表や列などのデータベース・オブジェクトの名前は、実行時に指定できます（ホスト変数と重複する名前は無効です）。問合せ結果を限定、分類、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います。

```
EXEC SQL
    PREPARE STATEMENTNAME FROM { :<HOST-STRING> | <STRING-LITERAL>
```

```
END-EXEC.
EXEC SQL DECLARE CURSORNAME CURSOR FOR STATEMENTNAME END-EXEC.
EXEC SQL OPEN CURSORNAME [USING <HOST-VARIABLE-LIST>] END-EXEC.
EXEC SQL FETCH CURSORNAME INTO <HOST-VARIABLE-LIST> END-EXEC.
EXEC SQL CLOSE CURSORNAME END-EXEC.
```

次に、それぞれの文の実行内容を説明します。

PREPARE

PREPARE はこの動的 SQL 文を解析するとともに名前を指定します。次の例では、PREPARE により文字列 *SELECT-STMT* に格納された問合せを解析し、その問合せに *SQLSTMT* という名前を付けます。

```
MOVE 'SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY'
    TO SELECT-STMT.
EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
```

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 *SQLSTMT* はホスト変数でもプログラム変数でもありませんが、一意でなければなりません。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 SQLSTMT FROM SELECT MGR, JOB FROM EMP WHERE SAL < :SALARY
END-EXEC.
```

'%' のワイルドカードを使った次の PREPARE 文も有効です。

```
EXEC SQL
    PREPARE S FROM SELECT ENAME FROM TEST WHERE ENAME LIKE 'SMIT%'
END-EXEC.
```

DECLARE

DECLARE は、カーソルに名前を指定するとともにこれを特定の問合せに対応付けてカーソルを定義します。カーソルの宣言は、そのプリコンパイル単位内でだけ有効です。上の例では、次に示すように、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* に対応付けます。

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR SQLSTMT END-EXEC.
```

識別子 *SQLSTMT* および *EMPCURSOR* はホスト変数でもプログラム変数でもありませんが、一意でなければなりません。同じ文名を使って 2 つのカーソルを宣言すると、

Pro*COBOL はその 2 つのカーソル名を同義とみなします。たとえば次の文を実行したとします。

```
EXEC SQL PREPARE SQLSTMT FROM :SELECT-STMT END-EXEC.
EXEC SQL DECLARE EMPCURSOR FOR SQLSTMT END-EXEC.
EXEC SQL PREPARE SQLSTMT FROM :DELETE-STMT END-EXEC.
EXEC SQL DECLARE DEPCURSOR FOR SQLSTMT END-EXEC.
```

この場合、*EMPCURSOR* を OPEN したときに処理対象となるのは *DELETE-STMT* に格納されている動的 SQL 文であって、*SELECT-STMT* に格納されている動的 SQL 文ではありません。

OPEN

OPEN により、カーソルの割当て、入力ホスト変数のバインド、問合せの実行、アクティブ・セットの決定を行います。さらに OPEN により、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み行数を 0（ゼロ）に設定します。PREPARE された動的 SQL 文中のプレースホルダは、USING 句の対応する入力ホスト変数に置き換えられます。

上の例では、次に示すように、OPEN により *EMPCURSOR* を割り当て、ホスト変数 *SALARY* を WHERE 句に割り当てます。

```
EXEC SQL OPEN EMPCURSOR USING :SALARY END-EXEC.
```

FETCH

FETCH はアクティブ・セットから行を戻し、選択リスト内の列の値を INTO 句内の対応するホスト変数に割り当ててから、カーソルを次の行に進めます。行がなくなると、" データがありません " というエラー・コードが SQLCA 内の SQLCODE に戻されます。

上の例では、次に示すように、FETCH によりアクティブ・セットから 1 行を戻し、列 MGR および JOB の値をホスト変数 *MGR-NUMBER* および *JOB-TITLE* に代入します。

```
EXEC SQL FETCH EMPCURSOR INTO :MGR-NUMBER,:JOB-TITLE END-EXEC.
```

CLOSE

CLOSE はカーソルを使用禁止にします。一度カーソルをクローズすると、それ以降は FETCH できなくなります。上の例では、次に示すように、CLOSE 文によって *EMPCURSOR* が無効になります。

```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

サンプル・プログラム 8: 動的 SQL 方法 3

このプログラムは、動的 SQL 方法 3 を使用して、指定された部門の全従業員の名前を EMP 表から取り出します。

```
*****************************************************************
* Sample Program 8:  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.    *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DYNSQL3.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

*    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.
     EXEC SQL INCLUDE SQLCA END-EXEC.

*    INCLUDE THE ORACLE COMMUNICATIONS AREA, A STRUCTURE THROUGH
*    WHICH ORACLE MAKES ADDITIONAL RUNTIME STATUS INFORMATION
*    AVAILABLE TO THE PROGRAM.
     EXEC SQL INCLUDE ORACA END-EXEC.

*    THE ORACA=YES OPTION MUST BE SPECIFIED TO ENABLE USE OF
*    THE ORACA.
     EXEC ORACLE OPTION (ORACA=YES) END-EXEC.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME  PIC X(10) VALUE "SCOTT".
 01  PASSWD    PIC X(10) VALUE "TIGER".
 01  DYNSTMT   PIC X(80) VARYING.
 01  ENAME     PIC X(10).
 01  DEPTNO    PIC S9999 COMPUTATIONAL VALUE 10.
     EXEC SQL END DECLARE SECTION END-EXEC.

*    DECLARE VARIABLES NEEDED TO DISPLAY COMPUTATIONALS.
 01  DEPTNOD   PIC 9(2).
 01  ENAMED    PIC X(10).
 01  SQLERRD3  PIC 9(2).
 01  ORASLNRD  PIC 9(4).
```

```
 PROCEDURE DIVISION.
 MAIN.

*    BRANCH TO PARAGRAPH SQLERROR IF AN ORACLE ERROR OCCURS.
     EXEC SQL WHENEVER SQLERROR GO TO SQLERROR END-EXEC.

*    SAVE TEXT OF CURRENT SQL STATEMENT IN THE ORACA IF AN ERROR
*    OCCURS.
     MOVE 1 TO ORASTXTF.

*    CONNECT TO ORACLE.
     EXEC SQL
         CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY " ".
     DISPLAY "CONNECTED TO ORACLE.".
     DISPLAY " ".

*    ASSIGN A SQL QUERY TO THE VARYING STRING DYNSTMT.  BOTH THE
*    ARRAY AND THE LENGTH PARTS MUST BE SET PROPERLY.  NOTE THAT
*    THE STATEMENT CONTAINS ONE HOST VARIABLE PLACEHOLDER, V1,
*    FOR WHICH AN ACTUAL INPUT HOST VARIABLE MUST BE SUPPLIED
*    AT OPEN TIME.
     MOVE "SELECT ENAME FROM EMP WHERE DEPTNO = :V1"
         TO DYNSTMT-ARR.
     MOVE 40 TO DYNSTMT-LEN.

*    DISPLAY THE SQL STATEMENT AND ITS CURRENT INPUT HOST
*    VARIABLE.
     DISPLAY DYNSTMT-ARR.
     MOVE DEPTNO TO DEPTNOD.
     DISPLAY "    V1 = ", DEPTNOD.
     DISPLAY " ".
     DISPLAY "EMPLOYEE".
     DISPLAY "--------".

*    THE PREPARE STATEMENT ASSOCIATES A STATEMENT NAME WITH A
*    STRING CONTAINING A SELECT STATEMENT.  THE STATEMENT NAME,
*    WHICH MUST BE UNIQUE, IS A SQL IDENTIFIER, NOT A HOST
*    VARIABLE, AND SO DOES NOT APPEAR IN THE DECLARE SECTION.
     EXEC SQL PREPARE S FROM :DYNSTMT END-EXEC.

*    THE DECLARE STATEMENT ASSOCIATES A CURSOR WITH A PREPARED
*    STATEMENT.  THE CURSOR NAME, LIKE THE STATEMENT NAME, DOES
*    NOT APPEAR IN THE DECLARE SECTION.
     EXEC SQL DECLARE C CURSOR FOR S END-EXEC.
```

```
*    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 MAY 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 MAY BE OPENED MORE THAN ONCE,
*    OPTIONALLY USING DIFFERENT INPUT HOST VARIABLES.
     EXEC SQL OPEN C USING :DEPTNO END-EXEC.

*    BRANCH TO PARAGRAPH NOTFOUND WHEN ALL ROWS HAVE BEEN
*    RETRIEVED.
     EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND END-EXEC.

 GETROWS.

*    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 ARE NOT RETURNED.  SPECIFYING MORE OUTPUT HOST
*    VARIABLES THAN SELECT-LIST FIELDS RESULTS IN AN ORACLE ERROR.
     EXEC SQL FETCH C INTO :ENAME END-EXEC.
     MOVE ENAME TO ENAMED.
     DISPLAY ENAMED.

*    LOOP UNTIL NOT FOUND CONDITION IS DETECTED.
     GO TO GETROWS.

 NOTFOUND.
     MOVE SQLERRD(3) TO SQLERRD3.
     DISPLAY " ".
     DISPLAY "QUERY RETURNED ", SQLERRD3, " ROW(S).".

*    THE CLOSE STATEMENT RELEASES RESOURCES ASSOCIATED WITH THE
*    CURSOR.
     EXEC SQL CLOSE C END-EXEC.

*    COMMIT ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
     EXEC SQL COMMIT RELEASE END-EXEC.
     DISPLAY " ".
```

```
      DISPLAY "HAVE A GOOD DAY!".
      DISPLAY " ".
      STOP RUN.

  SQLERROR.

*     ORACLE ERROR HANDLER.  PRINT DIAGNOSTIC TEXT CONTAINING
*     ERROR MESSAGE, CURRENT SQL STATEMENT, AND LOCATION OF ERROR.
      DISPLAY SQLERRMC.
      DISPLAY "IN ", ORASTXTC.
      MOVE ORASLNR TO ORASLNRD.
      DISPLAY "ON LINE ", ORASLNRD, " OF ", ORASFNMC.

*     DISABLE ORACLE ERROR CHECKING TO AVOID AN INFINITE LOOP
*     SHOULD ANOTHER ERROR OCCUR WITHIN THIS PARAGRAPH.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

*     RELEASE RESOURCES ASSOCIATED WITH THE CURSOR.
      EXEC SQL CLOSE C END-EXEC.

*     ROLL BACK ANY PENDING CHANGES AND DISCONNECT FROM ORACLE.
      EXEC SQL ROLLBACK RELEASE END-EXEC.
      STOP RUN.
```

方法 4 の使用方法

この項では、概要だけを説明します。詳細は第 11 章の「Oracle 動的 SQL: 方法 4」を参照してください

Oracle 方法 4 では LOB はサポートされません。LOB およびその他の新しいアプリケーションには、ANSI 動的 SQL を使用してください。

方法 3 を使用してプログラムで処理できない種類の動的 SQL 文があります。選択リスト項目の数や、入力ホスト変数のプレースホルダの数が実行時まで不明な場合、プログラムでは記述子を使用する必要があります。記述子とは、動的 SQL 文中の変数の完全な記述を保存するためにプログラムと Oracle8*i* が使用するメモリー領域です。

複数行を戻す問合せでは、選択された列値は宣言済みの出力ホスト変数のリストに FETCH INTO されます。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、次の問合わせでは 2 つの列値が戻されます。

```
EXEC SQL
    SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

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。

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。

同様に、動的 SQL 文に記述されている入力ホスト変数のプレースホルダの数が不明な場合には、プリコンパイル時に USING 句によるホスト変数リストの設定はできません。

このような動的 SQL 文を処理するには、プログラムは 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記述を格納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。）

プログラム内にアクティブな SQL 文が複数ある（たとえばプログラムが複数のカーソルを OPEN している）ときは、それぞれの文に専用の SQLDA が必要になります。ただし、カーソルが同時に実行されなければ 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。

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存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、PREPARE した動的問合せのそれぞれの選択リスト項目が DESCRIBE SELECT LIST 文によってチェックされます。これによって、選択リスト項目の名前、データ型、制約、長さ、位取りおよび精度が決定されます。続いて、この情報がその選択記述子に保存されます。

バインド記述子を指定すると、DESCRIBE BIND VARIABLES 文によって PREPARE 済みの動的 SQL 文内の各プレースホルダを調べ、その名前および長さ、対応する入力ホスト変数のデータ型を確認します。続いて、この情報がそのバインド記述子に保存されます。たとえば、プレースホルダ名を使用して、入力ホスト変数値の入力をユーザーに要求できます。

SQLDA とは

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 はホスト言語によって異なりますが、一般的な選択 SQLDA には、問合せ選択リストに関する次の情報が格納されます。

- DESCRIBE できる列の最大数

- 実際に DESCRIBE で検出された列数
- 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
- 列名の現在の長さ

一般的なバインド SQLDA には、SQL 文内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
- DESCRIBE によって検出されたプレースホルダの実際の数
- 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
- 標識変数名の現在の長さ

方法 4 の実行

方法 4 では、一般に次の順序で埋込み SQL 文を使います。

```
EXEC SQL
    PREPARE <STATEMENT-NAME>
    FROM { :<HOST-STRING> | <STRING-LITERAL> }
END-EXE
EXEC SQL
    DECLARE <CURSOR-NAME> CURSOR FOR <STATEMENT-NAME>
END-EXEC.
```

```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR <STATEMENT-NAME>
    INTO <BIND-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL
    OPEN <CURSOR-NAME>
    [USING DESCRIPTOR <BIND-DESCRIPTOR-NAME>]
END-EXEC.
EXEC SQL
    DESCRIBE [SELECT LIST FOR] <STATEMENT-NAME>
    INTO <SELECT-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL
    FETCH <CURSOR-NAME>
    USING DESCRIPTOR <SELECT-DESCRIPTOR-NAME>
END-EXEC.
EXEC SQL CLOSE <CURSOR-NAME> END-EXEC.
```

選択記述子とバインド記述子を両方とも使用する必要はありません。問合せ選択リストの列の数がわかっていて、入力ホスト変数のプレースホルダの数がわからない場合は、方法 4 の OPEN 文と次に示す方法 3 の FETCH 文を併用できます。

```
EXEC SQL FETCH <EMPCURSOR> INTO :<HOST-VARIABLE-LIST> END-EXEC.
```

逆に、入力ホスト変数のプレースホルダの数がわかっていて、選択リストの列の数がわからない場合は、次に示す方法 3 の OPEN 文と方法 4 の FETCH 文を併用できます。

```
EXEC SQL OPEN <CURSORNAME> [USING <HOST-VARIABLE-LIST>] END-EXEC.
```

方法 4 では問合せ以外の SQL 文に対して EXECUTE を使用できるので注意してください。

DECLARE STATEMENT 文の使用方法

方法 2、3、4 では、次の文を使わなければならない場合があります。

```
EXEC SQL [AT <dbname>] DECLARE <statementname> STATEMENT END-EXEC.
```

dbname と *statementname* は Pro*COBOL が使用する識別子であって、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、DESCRIBE で参照できるようになります。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 <remotedb> DECLARE <sqlstmt> STATEMENT END-EXEC.
EXEC SQL PREPARE <sqltmt> FROM :<SQL-STRING> END-EXEC.
EXEC SQL EXECUTE <sqlstmt> END-EXEC.
```

この例では、*remotedb* によって、SQL 文をどこで EXECUTE すべきか Oracle8*i* 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 文の前にあるときにも DECLARE STATEMENT が必要です。

```
EXEC SQL DECLARE <sqlstmt> STATEMENT END-EXEC.
EXEC SQL DECLARE <empcursor> CURSOR FOR <sqlstmt> END-EXEC.
EXEC SQL PREPARE <sqlstmt> FROM :<SQL-STRING> END-EXEC.
```

一般的な文の順序は次のとおりです。

```
EXEC SQL PREPARE <sqlstmt> FROM :<SQL-STRING> END-EXEC.
EXEC SQL DECLARE <empcursor> CURSOR FOR <sqlstmt> END-EXEC.
```

ホスト表の使用方法

ホスト表の使用方法は、静的 SQL でも動的 SQL でも同じです。たとえば、動的 SQL 方法 2 で入力ホスト表を使用する場合は、次の構文を使います。

```
EXEC SQL EXECUTE <statementname> USING :HOST-TABLE-LIST END-EXEC.
```

HOST-TABLE-LIST は 1 つ以上のホスト表で構成されています。方法 3 の場合は、次の構文を使用します。

```
OPEN <cursorname> USING :<HOST-TABLE-LIST> END-EXEC.
```

方法 3 で出力ホスト表を使用する場合は、次の構文を使います。

```
FETCH <cursorname> INTO :<HOST-TABLE-LIST> END-EXEC.
```

方法 4 では、オプションの FOR 句を使って、入力ホスト表または出力ホスト表のサイズを Oracle8*i* に認識させなければなりません。この方法については、ホスト言語の補足を参照してください。

PL/SQL の使用方法

Pro*COBOL は、PL/SQL ブロックを単一の SQL 文のように扱います。したがって SQL 文と同様に、PL/SQL ブロックを文字列ホスト変数またはリテラルに格納できます。PL/SQL ブロックを文字列に格納するときは、キーワード EXEC SQL EXECUTE およびキーワード END-EXEC、文終了記号は省略してください。

ただし、Pro*COBOL による SQL と PL/SQL の処理方法には、次の 2 つの相違点があります。

- PL/SQL ブロック内で入力ホスト変数として使用されているか出力ホスト変数として使用されているか（あるいはその両方として使用されているか）に関係なく、Pro*COBOL は PL/SQL のホスト変数をすべて入力ホスト変数として扱います。
- 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。

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。

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。PL/SQL 文字列を EXECUTE すると、PREPARE された文字列内のプレースホルダが USING 句内の対応するホスト変数に置き換えられます。Pro*COBOL 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が、値は正しく代入されます。入力（プログラム）値は入力ホスト変数に代入されます。また出力（列）値は出力ホスト変数に代入されます。

PREPARE された PL/SQL 文字列内のプレースホルダはすべて、USING 句内のホスト変数に対応していなければなりません。このため、PREPARE された文字列で同じプレースホルダが複数回使用されている場合は、それぞれが USING 句内のホスト変数に対応している必要があります。

方法 3 の場合

方法 3 は、FETCH が使えることを除けば方法 2 と同じです。PL/SQL ブロックから FETCH を行うことはできないので、方法 2 を使用してください。

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 を必ず使います。

方法 4 を使うには、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。Pro*COBOL 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るので、DESCRIBE SELECT LIST を実行しても効果はありません。

方法 4 でのバインド記述子の詳細は、ホスト言語の補足を参照してください。

注意

動的 SQL 方法 4 では、型が "table" のパラメータを指定してホスト配列を PL/SQL プロシージャにバインドすることはできません。

動的に処理される PL/SQL ブロックでは、行終了文字が無視されるので、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が続いてしまいます。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。

10

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（SQL92 動的 SQL とも呼ばれます）の Oracle でのインプリメンテーションについて説明します。インプリメンテーションは、11-1 ページの「Oracle 動的 SQL: 方法 4」で説明した従来の Oracle 動的 SQL 方法 4 に拡張機能を加えたものです。Oracle 方法 4 ではカーソル変数、グループ項目の表、DML 戻し句および LOB はサポートされませんが、ANSI 方法 4 では Oracle の型がすべてサポートされます。

従来の Oracle 動的 SQL 方法 4 では記述子はユーザーの Pro*COBOL プログラムで定義されますが、ANSI 動的 SQL では記述子は Oracle によって内部で管理されます。どちらの場合でも、方法 4 では、Pro*COBOL プログラムがさまざまな数のホスト変数を含む SQL 文の受け入れまたは作成を行います。

この章の構成は、次のとおりです。

- ANSI 動的 SQL の基礎
- ANSI SQL 文の概要
- Oracle 拡張要素
- 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 : SAMPLE12.PCO

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 を使用するには次の手順を行います。

- 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。
- 出力記述子を設定します（この例では出力ホスト変数 ename と empno）。
- 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 を使っ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放します。

プリコンパイラ・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 に設定します。MODE を ANSI に設定すると、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。マイクロ・オプションの詳細は 14-5 ページの「マクロ・オプションとマイクロ・オプション」を参照してください。

ANSI タイプ・コードを使用するには、プリコンパイラ・マイクロ・オプション TYPE_CODE を ANSI に設定するか、マクロ・オプション MODE を ANSI に設定します。MODE を ANSI に設定すると、TYPE_CODE のデフォルト設定が ANSI になります。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。

10-4 ページの表 10-1 に示す Oracle での ANSI SQL タイプのインプリメンテーションは、正確には ANSI 規格と一致しません。たとえば、INTEGER として宣言した列に DESCRIBE を実行すると、NUMERIC のコードが戻ります。Oracle を ANSI 規格に近づけるには、動作を多少変更する必要があることがあります。異なるデータベース・プラットフォームへの移植性があり、できる限り ANSI に準拠したアプリケーションを作成するのであれば、プリコンパイラ・オプション TYPE_CODE を ANSI に設定して ANSI タイプを使用してください。動作の変更が受け入れられない場合は、TYPE_CODE の設定を ANSI にしないでください。

ANSI SQL 文の概要

動的 SQL 文では、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}]
END-EXEC.
```

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だけアクセス可能です。デフォルトはローカルです。

記述子名 `desc_nam` は、ホスト変数です。かわりに文字列リテラルを使うこともできます。

`occurrences` は、記述子に格納できるバインド変数または列の最大数です。デフォルトは 100 です。

記述子が必要なくなったら、割当てを解放してメモリーを節約します。データベース接続がなくなった場合は、割当ての解放が自動的に行われます。

割当て解放文は次のとおりです。

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
END-EXEC.
```

準備済みの SQL 文に関する情報を取得するには、DESCRIBE 文を使います。DESCRIBE INPUT では、準備済み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}
END-EXEC.
```

SQL 文に入力値と出力値がある場合は、入力値用と出力値用の 2 つの記述子を割り当てる必要があります。次の例のように入力値がない場合は、入力記述子は必要ありません。

```
SELECT ename, empno FROM emp
```

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。入力記述子に DESCRIBE を行っていないときは、SET DESCRIPTOR を使って（*COUNT* に格納されている）入力バインド変数の数を設定します。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   COUNT = {:kount | numeric_literal}
END-EXEC.
```

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。次に、SET DESCRIPTOR を各ホスト変数に使って、少なくとも変数のデータ値だけは割り当てる必要があります。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number DATA = :hv3
END-EXEC.
```

また、次のように、入力ホスト変数の型と長さを設定することもできます。

注意 : TYPE_CODE=ORACLE の場合、SET DESCRIPTOR 文で明示的に、または DESCRIBE OUTPUT の実行を通して暗黙的に型と長さを設定しなかった場合は、ホスト変数自体から導出された型および長さの値がプリコンパイラで使用されます。TYPE_CODE=ANSI の場合は、表 10-1 の「ANSI SQL データ型」に示す値を使って型を設定しなければなりません。また、ANSI のデフォルトの長さがホスト変数のデフォルトの長さと一致しない場合があるので、長さ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
END-EXEC.
```

ホスト変数によって値が供給される必要があることがわかるように、ここでは識別子 hv1、hv2 および hv3 を使います。*item_number* は、SQL 文での入力変数の位置です。ここには、ホスト変数または整数を指定できます。

TYPE_CODE を ANSI に設定した場合は、次の表のタイプ・コードから TYPE を選択します。

表 10-1 ANSI SQL データ型

データ型	タイプ・コード
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

Oracle タイプ・コードは 11-14 ページの表 11-2 の「Oracle の外部データ型および関連する COBOL データ型」を参照してください。ANSI データ型が表にはなく、TYPE_CODE = ANSI である場合は、Oracle コードの負の値を使用してください。

DATA は、入力されるホスト変数値です。

標識、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。詳細は、10-17 ページの「SET DESCRIPTOR」の使用できる記述子項目名の一覧を参照してください。

SET DESCRIPTOR 文の数値は、`PIC S9(9) COMP` または `PIC S9(4) COMP` のどちらかとして宣言する必要があります。ただし、標識の値および戻される長さの値は、`PIC S9(9) COMP` として宣言しなければなりません。

次の例に、empno を取り出すときの値の設定を示します。empno は動的 SQL 文の 2 つ目の出力ホスト変数なので、VALUE=2 と設定します。ホスト変数 `EMPNO-TYP` は、3（整数の Oracle タイプ）に設定されます。ホスト変数の長さ `EMPNO-LEN` は、4 に設定されます。これは、ホスト変数のサイズを示します。DATA は、データベース表からの値を受け取るホスト変数 EMPNO-DATA と等しくなります。コードの一部を次に示します。

```
...
01  DYN-STATEMENT PIC X(58)
    VALUE "SELECT ename, empno FROM emp WHERE deptno =:deptno_number".
01  EMPNO-DATA PIC S9(9) COMP.
01  EMPNO-TYP  PIC S9(9) COMP   VALUE 3.
01  EMPNO-LEN  PIC S9(9) COMP   VALUE 4.
...
    EXEC SQL SET DESCRIPTOR 'out' VALUE 2  TYPE=:EMPNO-TYP, LENGTH=:EMPNO-LEN,
       DATA=:EMPNO-DATA END-EXEC.
```

入力値を設定した後で、入力記述子を使って文を実行またはオープンします。文に出力値がある場合は、FETCH を実行する前に出力値を設定します。DESCRIBE OUTPUT を実行した場合は、ホスト変数の実際の型と長さを再設定しなければならない場合があります。これは、DESCRIBE を実行すると、ホスト変数の外部型および長さとは別の内部型および長さが生成されるためです。

出力記述子を FETCH した後、GET DESCRIPTOR を使って、戻されたデータにアクセスします。単純化した構文を次に示します。詳しい構文については、この章の後の方に示します。

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number
   :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH
END-EXEC.
```

`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、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しなければなりません。リテラルは指定できません。例では、戻されるデータを 3 つだけ示しています。戻されるデータとして取得できる項目は、10-15 ページの表 10-4 の「記述子項目名の定義」の一覧を参照してください。

数値には、プラットフォーム依存の上限が n である PIC S9(n) COMP を使用するか、または PIC S9(9) COMP か *PIC S9(4) COMP* を使用します。ただし、標識変数および戻される長さの変数は *PIC S9(4) COMP* でなければなりません。

サンプル・コード

次の例に、ANSI 動的 SQL の使用方法を示します。この例では、SELECT 文を実行するための入力記述子（'in'）および出力記述子（'out'）を割り当てます。入力値は、SET DESCRIPTOR 文によって設定します。カーソルを開き、そこからフェッチを実行します。結果の出力値を GET DESCRIPTOR 文によって取り出します。

```
 ...
 01  DYN-STATEMENT PIC X(58)
     VALUE "SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO =:DEPTNO-DAT".
 01  EMPNO-DAT  PIC S9(9) COMP.
 01  EMPNO-TYP  PIC S9(9) COMP   VALUE 3.
 01  EMPNO-LEN  PIC S9(9) COMP   VALUE 4.
 01  DEPTNO-TYP PIC S9(9) COMP   VALUE 3.
 01  DEPTNO-LEN PIC S9(9) COMP   VALUE 2.
 01  DEPTNO-DAT PIC S9(9) COMP   VALUE 10.
 01  ENAME-TYP  PIC S9(9) COMP   VALUE 3.
 01  ENAME-LEN  PIC S9(9) COMP   VALUE 30.
 01  ENAME-DAT  PIC X(30).
 01  SQLCODE PIC S9(9)    COMP   VALUE 0.
 ...
* Place preliminary code, including connection, here
...
     EXEC SQL ALLOCATE DESCRIPTOR 'in' END-EXEC.
     EXEC SQL ALLOCATE DESCRIPTOR 'out' END-EXEC.
     EXEC SQL PREPARE s FROM :DYN-STATEMENT END-EXEC.
     EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE=:DEPTNO-TYP,
        LENGTH=:DEPTNO-LEN, DATA=:DEPTNO-DAT END-EXEC.
     EXEC SQL DECLARE c CURSOR FOR s END-EXEC.
     EXEC SQL OPEN c USING DESCRIPTOR 'in' END-EXEC.
     EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out' END-EXEC.
     EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
        LENGTH=:ENAME-LEN, DATA=:ENAME-DAT END-EXEC.
     EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP,
        LENGTH=:EMPNO-LEN, DATA=:EMPNO-DAT END-EXEC.

     EXEC SQL WHENEVER NOT FOUND GOTO BREAK END-EXEC.
```

```
LOOP.
    IF SQLCODE NOT = 0
       GOTO BREAK.
    EXEC SQL FETCH c INTO DESCRIPTOR 'out' END-EXEC.
    EXEC SQL GET DESCRIPTOR 'OUT' VALUE 1 :ENAME-DAT = DATA END-EXEC.
    EXEC SQL GET DESCRIPTOR 'OUT' VALUE 2 :EMPNO-DAT = DATA END-EXEC.
    DISPLAY "ENAME = " WITH NO ADVANCING
    DISPLAY ENAME-DAT  WITH NO ADVANCING
    DISPLAY "EMPNO = " WITH NO ADVANCING
    DISPLAY EMPNO-DAT.
    GOTO LOOP.
BREAK:
    EXEC SQL CLOSE c END-EXEC.
    EXEC SQL DEALLOCATE DESCRIPTOR 'in' END-EXEC.
    EXEC SQL DEALLOCATE DESCRIPTOR 'out' END-EXEC.
```

Oracle 拡張要素

この項では、次の拡張要素について説明します。

- SET 文のデータ項目のリファレンス・セマンティックス
- 一括操作のための配列
- オブジェクト型、NCHAR 列および LOB のサポート

リファレンス・セマンティックス

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リファレンス・セマンティッ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。リファレンス・セマンティックスでは、ホスト変数のアドレスを使い、コピーは行いません。そのため、リファレンス・セマンティックスを使うと、大量データ処理のパフォーマンスが向上します。

フェッチの速度を上げるには、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-TYP,
   LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT END-EXEC.
EXEC SQL DESCRIPTOR 'out' VALUE 2 TYPE=:EMPNO-TYP,
   LENGTH=:EMPNO-LEN, REF DATA=:EMPNO-DAT END-EXEC.
```

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化できます）の前だけです。

```
 01   INDI       PIC S9(4) COMP.
 01   RETRN-LEN  PIC S9(9) COMP.
 ...
      EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE=:ENAME-TYP,
         LENGTH=:ENAME-LEN, REF DATA=:ENAME-DAT,
         REF INDICATOR=:INDI, REF RETURNED_LENGTH =:RETRN-LEN END-EXEC.
```

フェッチするたびに、ename フィールドの実際に取り出された長さが RETRN-LEN に送られます。これは、CHAR または VARCHAR2 データの場合に便利です。

ENAME-LEN には取り出された長さは渡されません。ENAME-LEN 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います。リファレンス・セマンティックスでは、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ることに注意してください。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。次に例を示します。

```
 ...
     MOVE 1 to VAL.
 ...
     EXEC SQL SET DESCRIPTOR 'value' VALUE 1 DATA = :VAL END-EXEC.
     EXEC SQL SET DESCRIPTOR 'reference' VALUE 1 REF DATA = :VAL END-EXEC.
     MOVE 2 to VAL.
* Will use  VAL = 1
     EXEC SQL EXECUTE s USING  DESCRIPTOR 'value' END-EXEC.
* Will use VAL = 2
     EXEC SQL EXECUTE s USING DESCRIPTOR 'reference' END-EXEC.
```

この処理の違いについての詳細は 10-17 ページの「SET DESCRIPTOR」を参照してください。

一括操作での表の使用

Oracle では、SQL92 ANSI 動的規格を拡張した一括操作を提供しています。一括操作を行うには、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を指定するために使います。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' END-EXEC.
```

または

```
     MOVE 100 TO INT-ARR-SIZE.
     EXEC SQL FOR :INT-ARR-SIZE ALLOCATE DESCRIPTOR 'out' END-EXEC.
```

次に、記述子にアクセスする文で FOR 句を使います。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' END-EXEC.
```

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しなければなりません。

```
 01  VAL-DATA   OCCURS 20 TIMES  PIC S9(9) COMP.
 01  VAL-INDI   OCCURS 20 TIMES  PIC S9(4) COMP.
...
     EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :VAL-DATA = DATA,
       :VAL-INDI = INDICATOR
     END-EXEC.
```

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 句を使ってはいけません。

```
 01  CNT  PIC S9(9) COMP.
 01  LEN  PIC S9(9) COMP.
...
     EXEC SQL GET DESCRIPTOR 'out'   :CNT = COUNT END-EXEC.
     EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :LEN = LENGTH END-EXEC.
```

これは、リファレンス・セマンティックスを使った SET 文でも同じです。FETCH の前にあり、DATA、INDICATOR または RETURNED_LENGTH に対するリファレンス・セマンティックスを使用した SET 文には、FETCH と同じ配列サイズを指定しなければなりません。

```
 ...
 01  REF-DATA   OCCURS 20 TIMES  PIC S9(9) COMP.
 01  REF-INDI   OCCURS 20 TIMES  PIC S9(4) COMP.
...
     EXEC SQL FOR 20 SET DESCRIPTOR 'out' VALUE 1 REF DATA = :REF-DATA,
        REF INDICATOR = :REF-INDI END-EXEC.
```

同様に , 行のバッチの挿入など、入力に使う記述子でも、ALLOCATE 文で使っ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 文に使わなければなりません。値構文とリファレンス・セマンティックスのどちらでも、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う必要があります。

FOR 句は、DEALLOCATE または PREPARE 文では使用しません。

次のコード・サンプルに、出力記述子のない一括操作の例を示します（出力はなく、表 emp に挿入する入力だけがあります）。CNT の値は 2 です（INSERT 文に ENAME と EMPNO の 2 つのホスト変数があることを示します）。データ表 ENAME-TABLE に、"Tom "、"Dick " および "Harry " の 3 つ文字列をこの順序で格納します。彼らの雇用者番号を表 EMPNO-TABLE

に格納します。標識表 ENAME-IND には 2 つ目の要素に -1 の値を設定しているので、"Dick" ではなく NULL が挿入されます。挿入された実際の名前を確認するために RETURNING 句を使用できます。

```
 01  DYN-STATEMENT  PIC X(240) value
        "INSERT INTO EMP(ENAME, EMPNO) VALUES (:ENAME,:EMPNO)".
 01  ARRAY-SIZE PIC S9(9) COMP   VALUE 3.
 01  ENAME-VALUES.
      05 FILLER PIC X(6)   VALUE "Tom   ".
      05 FILLER PIC X(6)   VALUE "Dick  ".
      05 FILLER PIC X(6)   VALUE "Harry ".
 01  ENAME-TABLE REDEFINES ENAME-VALUES.
      05 ENAME  PIC X(6)OCCURS 3 TIMES.
 01  ENAME-IND  PIC S9(4) COMPOCCURS 3 TIMES.
 01  ENAME-LEN  PIC S9(9) COMP   VALUE   6.
 01  ENAME-TYP  PIC S9(9) COMP   VALUE  96.
 01  EMPNO-VALUES.
      05 FILLER PIC S9(9) COMP   VALUE 8001.
      05 FILLER PIC S9(9) COMP   VALUE 8002.
      05 FILLER PIC S9(9) COMP   VALUE 8003.
 01  EMPNO-TABLE REDEFINES EMPNO-VALUES.
      05 EMPNO  PIC S9(9) DISPLAY SIGN LEADING  OCCURS 3 TIMES.
 01  EMPNO-LEN  PIC S9(9) COMP   VALUE   4.
 01  EMPNO-TYP  PIC S9(9) COMP   VALUE   3.
 01  CNT        PIC S9(9) COMP   VALUE   2.
........
     EXEC SQL FOR :ARRAY-SIZE ALLOCATE DESCRIPTOR 'in' END-EXEC.
     EXEC SQL PREPARE S FROM :DYN-STATEMENT  END-EXEC.
     MOVE 0 TO ENAME-IND(1).
     MOVE -1 TO ENAME-IND(2).
     MOVE 0 TO ENAME-IND(3).
     EXEC SQL SET DESCRIPTOR 'in' COUNT = :CNT END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 1
        TYPE = :ENAME-TYP, LENGTH =:ENAME-LEN
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE  SET DESCRIPTOR 'in' VALUE 1
        DATA = :ENAME, INDICATOR = :ENAME-IND
     END-EXEC.
     EXEC SQL SET DESCRIPTOR 'in' VALUE 2
        TYPE = :EMPNO-TYP, LENGTH =:EMPNO-LEN
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE  SET DESCRIPTOR 'in' VALUE 2
        DATA = :EMPNO
     END-EXEC.
     EXEC SQL FOR :ARRAY-SIZE EXECUTE S
        USING DESCRIPTOR 'in' END-EXEC.
  ...
```

この例のコードによって、次の値が表に挿入されます。

```
EMPNO   ENAME
8001    Tom
8002
8003    Harry
```

制限および注意については、7-15 ページの「FOR 句の使用方法」を参照してください。

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 MODE（14-30 ページの「MODE」を参照）では、ANSI 互換特性を設定し、いくつかの機能を制御します。MODE の値には 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。

プリコンパイラ・マイクロ・オプション DYNAMIC では、動的 SQL の記述子の動作を指定します。プリコンパイラ・マイクロ・オプション TYPE_CODE では、ANSI と Oracle のどちらのデータ型コードを使うかを指定します。

マクロ・オプションを 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 と 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 設定による影響を示します。

表 10-2 DYNAMIC オプションの設定

機能	DYNAMIC=ANSI	DYNAMIC=ORACLE
記述子の作成	ALLOCATE 文を使用しなければなりません。	Oracle 形式の記述子を使用しなければなりません。
記述子の破棄	DEALLOCATE 文を使用できます。	使用不可。
データの取出し	FETCH 文と GET 文のどちらも使用できます。	FETCH 文だけ使用できます。
入力データの設定	DESCRIBE INPUT 文を使用できます。SET 文を使用しなければなりません。	コードに記述子値を設定しなければなりません。DESCRIBE BIND VARIABLES 文を使用しなければなりません。
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。

表 10-2 DYNAMIC オプションの設定

機能	DYNAMIC=ANSI	DYNAMIC=ORACLE
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 およびカーソル変数を除く 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 にしか設定できません。

TYPE_CODE の設定に対応する機能は次のとおりです。

表 10-3 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 のときだけ有効です。	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。

動的 SQL 文の構文

次の文の詳細は、付録 F の「埋込み SQL 文とプリコンパイラ ・ディレクティブ」のアルファベット順の一覧を参照してください。

ALLOCATE DESCRIPTOR

用途

この文は、SQL 記述子領域を割り当てるために使います。記述子、記述子のホスト・バインド項目のオカレンスの最大数、配列サイズを指定します。この文は ANSI 動的 SQL だけで使用できます。

構文

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal} [WITH MAX occurrences]
END-EXEC.
```

変数

array_size

これは、表の処理をサポートするオプションの句（Oracle 拡張要素）です。表の処理に記述子を使用できることをプリコンパイラに示します。

GLOBAL | LOCAL

オプションの有効範囲句です。入力しなかった場合のデフォルトは LOCAL です。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だけアクセス可能です。グローバル記述子は、コンパイル・ユニット内の任意のモジュールで使用できます。

desc_nam

ローカル記述子は、モジュール内で一意でなければなりません。記述子が以前に割り当てられていてまだ解放されていない場合は、ランタイム・エラーが生成されます。グローバル記述子は、アプリケーション内で一意でなければなりません。一意でない場合はランタイム・エラーが生成されます。

occurrences

記述子のホスト変数の予測最大数。0 ～ 64K の整定数を指定しなければなりません。これ以外の数を指定するとエラーが戻されます。デフォルトは 100 です。この句は省略できます。指定規則に反していた場合は、プリコンパイラ・エラーが生成されます。

例

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 END-EXEC.

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :BINDDES WITH MAX 25
   END-EXEC.
```

DEALLOCATE DESCRIPTOR

用途

この文は、割当て済みの SQL 記述子領域の割当てを解放して、メモリーを節約するために使います。この文は ANSI 動的 SQL だけで使用できます。

構文

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
   {:desc_nam | string_literal}
END-EXEC.
```

変数

desc_nam

記述子名。同じ名前と有効範囲の記述子が割り当てられていない場合、または割り当てられていてすでに解放されている場合は、ランタイム・エラーが生成されます。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' END-EXEC.

EXEC SQL DEALLOCATE DESCRIPTOR :BINDDES END-EXEC.
```

GET DESCRIPTOR

用途

この文は、SQL 記述子領域から情報を取得するために使います。

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 }]}
END-EXEC.
```

変数

array_size

`FOR array_size` は、オプションの Oracle 拡張要素です。`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。

desc_nam

記述子名。

GLOBAL | LOCAL

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。デフォルトは LOCAL です。

COUNT

バインド変数の合計数。

VALUE item_number

SQL 文での項目の位置。item_number には、変数または定数を指定できます。item_number が COUNT よりも大きいと、" データがありません " という状態が戻されます。item_number は 0（ゼロ）よりも大きくなければなりません。

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。ANSI 記述子項目名には次のものがあります。

表 10-4 記述子項目名の定義

記述子項目名	意味
TYPE	ANSI タイプ・コードは 10-4 ページの表 10-1 を参照してください。Oracle タイプ・コードは 11-14 ページの表 11-2 を参照してください。ANSI データ型が表になく、TYPE_CODE = ANSI である場合は、Oracle コードの負の値を使用してください。
LENGTH	列内のデータの長さ。NCHAR は文字単位、それ以外はバイト単位です。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	厳密な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 のときは、列に NULL 値は使用できません。
INDICATOR	対応付けられた標識値
DATA	データ値
NAME	列名
CHARACTER_SET_NAME	列の文字セット

Oracle に追加された記述子項目名は次のとおりです。

表 10-5 記述子項目名の定義の Oracle 拡張要素

記述子項目名	意味
NATIONAL_CHARACTER	2 は NCHAR または NVARCHAR2 を示します。1 は文字を示します。0 は非文字を示します。
INTERNAL_LENGTH	内部でのバイト単位の長さ。

使用上の注意

FOR 句は、DATA、INDICATOR および RETURNED_LENGTH 項目だけを含む 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、Oracle コードまたは ANSI コード（10-4 ページの表 10-1）です。ANSI 型が表にはない場合は、Oracle 型の負の値を使用してください。

LENGTH には、列の長さが格納され、固定幅の各国語キャラクタ・セットが設定されたフィールドでは文字単位、それ以外の文字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、文字列の場合にだけ意味があります。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標識ステータスです。データが NULL で標識が要求されなかった場合は、実行時にエラーが生成されます（"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。列が文字（各国文字以外）の場合は、この項目は 1 に設定されます。非文字列の場合は、DESCRIBE OUTPUT の実行後 0（ゼロ）に設定されます。

INTERNAL_LENGTH は、Oracle 動的方法 4 との互換性を保つための項目です。Oracle 記述子領域の長さメンバーと同じ値に設定されます。詳細は 11-1 ページの「Oracle 動的 SQL: 方法 4」を参照してください。

例

```
EXEC SQL GET DESCRIPTOR :BINDDES :COUNT = COUNT END-EXEC.

EXEC SQL GET DESCRIPTOR 'SELDES' VALUE 1 :TYP = TYPE, :LEN = LENGTH
   END-EXEC.

EXEC SQL FOR :batch GET DESCRIPTOR LOCAL 'SELDES'
   VALUE :SEL-ITEM-NO :IND = INDICATOR, :DAT = DATA END-EXEC.
```

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 では、項目名のホスト変数のみがサポートされます。

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}]}
END-EXEC.
```

変数

array_size

これは、オプションの Oracle 句です。この句によって、記述子項目の DATA、INDICATOR および RETURNED_LENGTH だけを設定するときに配列を使用できます。FOR 句を含む SET DESCRIPTOR では、他の項目は使用できません。すべてのホスト変数表のサイズが一致しなければなりません。SET 文の配列サイズは、FETCH 文に使う配列サイズと同じ設定にします。

desc_nam

記述子名。ALLOCATE DESCRIPTOR での規則が適用されます。

COUNT

バインド変数（入力）または定義変数（出力）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_name1 .. item_nameN

GET DESCRIPTOR 構文（10-14 ページの「GET DESCRIPTOR」を参照）と同様に、`item_name` に次の値を指定できます。

表 10-6 SET DESCRIPTOR の記述子項目名

記述子項目名	意味
TYPE	ANSI タイプ・コードは 10-4 ページの表 10-1 を参照してください。Oracle タイプ・コードは 11-14 ページの表 11-2 を参照してください。ANSI 型が表にはなく、TYPE_CODE = ANSI である場合は、Oracle タイプ・コードの負の値を使用してください。
LENGTH	列内のデータの最大長。
PRECISION	桁数。
SCALE	厳密な数値型での小数点の右側のバイト数。
INDICATOR	対応付けられた標識値。リファレンス・セマンティックスのために設定します。
DATA	設定するデータの値。リファレンス・セマンティックスのために設定します。
CHARACTER_SET_NAME	列の文字セット。

記述子項目名の Oracle 拡張要素は次のとおりです。

表 10-7 SET DESCRIPTOR の記述子項目名の Oracle 拡張要素

記述子項目名	意味
RETURNED_LENGTH	FETCH 後に戻される長さ。リファレンス・セマンティッ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。

使用上の注意

リファレンス・セマンティックスは、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 拡張要素です。DATA、INDICATOR および RETURNED_LENGTH にだけ、その記述子項目名の前に REF キーワードを指定します。REF キーワードを使った場合は、GET

文を使う必要はありません。複合データ型（DML の戻し句）には、SET DESCRIPTOR の REF 書式が必要です。詳細は 5-9 ページの「DML 戻し句」を参照してください。

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 セマンティックスを使用するときにだけ、RETURNED_LENGTH を設定できます。

SET または GET 文の配列サイズは、FETCH で使う配列サイズと同じにします。

NCHAR ホスト変数には、NATIONAL_CHAR フィールドを 2 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 を設定しなければなりません。

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。

例

一括表の例は 10-8 ページの「一括操作での表の使用」を参照してください。

```
 ...
 01  BINDNO  PIC S9(9) COMP  VALUE 2.
 01  INDI    PIC S9(4) COMP  VALUE -1.
 01  DATA    PIC X(6)  COMP  VALUE "ignore".
 01  BATCH   PIC S9(9) COMP  VALUE 1.
 ...
     EXEC SQL FOR :batch ALLOCATE DESCRIPTOR :BINDDES END-EXEC.
     EXEC SQL SET DESCRIPTOR GLOBAL :BINDDES COUNT = 3 END-EXEC.
     EXEC SQL FOR :batch SET DESCRIPTOR :BINDDES
        VALUE :BINDNO INDICATOR = :INDI, DATA = :DATA END-EXEC.
...
```

PREPARE の使用

用途

この方法で使用する PREPARE 文は、Oracle 動的 SQL 方法で使用する PREPARE 文と同じです。Oracle 拡張要素によって、変数と同様に SQL 文で引用符付きの文字列を使用できます。

構文

```
EXEC SQL PREPARE statement_id FROM :sql_statement END-EXEC.
```

変数

statement_id

この変数を宣言してはいけません。これは、準備済みの SQL 文に対応付けられた未宣言の 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

```
 ...
 01  STATEMENT    PIC X(255)
        VALUE "SELECT ENAME FROM emp WHERE deptno = :d".
 ...
     EXEC SQL PREPARE S1 FROM :STATEMENT END-EXEC.
```

DESCRIBE INPUT

用途

この文は、入力バインド変数に関する情報を戻します。

構文

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR
   [GLOBAL | LOCAL] {:desc_nam | string_literal}
END-EXEC.
```

変数

statement_id

PREPARE および DESCRIBE OUTPUT で使うものと同じです。この変数を宣言してはいけません。これは SQL 識別子です。

GLOBAL | LOCAL

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。デフォルトは LOCAL です。

desc_nam

記述子名。

使用上の注意

このバージョンでは、COUNT および NAME だけがバインド変数のためにインプリメントされています。

例

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC.
EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' END-EXEC.
```

DESCRIBE OUTPUT

用途

この文は、PREPARE 文の列に関する情報を取得するために使用します。ANSI 構文は以前の構文と違います。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_literal}
END-EXEC.
```

変数

statement_id

PREPARE で使うものと同じです。この変数を宣言してはいけません。これは SQL 識別子です。

GLOBAL | LOCAL

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。
LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。
デフォルトは LOCAL です。

desc_nam

記述子名。ホスト変数の前に ":" を付けたもの、または引用符付き文字列。

OUTPUT がデフォルト設定で、これは省略できます。

例

```
...
01  DESNAME   PIC X(10) VALUE "SELDES".
...
    EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES' END-EXEC.
```

```
* Or:
     EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :DESNAME END-EXEC.
```

EXECUTE

用途

EXECUTE は、準備済みの 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。ANSI バージョンの EXECUTE は、以前の EXECUTE とは違い、DML RETURNING をサポートするために 1 つの文に記述子を 2 つ使用できます。

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}]
END-EXEC.
```

変数

array_size

文が処理する行数。

statement_id

PREPARE で使うものと同じです。この変数を宣言してはいけません。これは SQL 識別子です。リテラルを指定できます。

GLOBAL | LOCAL

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。デフォルトは LOCAL です。

desc_nam

記述子名。ホスト変数の前に ":" を付けたもの、または引用符付き文字列。

使用上の注意

INTO 句は、INSERT、UPDATE および DELETE の RETURNING 句をインプリメントします（5-8 ページの「行の挿入」とその後のページを参照してください）。

例

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :BINDDES END-EXEC.

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES'
   END-EXEC.
```

EXECUTE IMMEDIATE の使用

用途

SQL 文を含むリテラル文字列またはホスト変数文字列を実行します。この文の ANSI SQL 書式は、以前の動的 SQL と同じです。

構文

EXEC SQL EXECUTE IMMEDIATE [:]*sql_statement END-EXEC.*

変数

sql_statement

文字列内の SQL 文または PL/SQL ブロック。ホスト変数またはリテラルを指定できます。

例

EXEC SQL EXECUTE IMMEDIATE :*statement* END-EXEC.

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文です。

構文

EXEC SQL DECLARE *cursor_name* CURSOR FOR *statement_id END-EXEC.*

変数

cursor_name

カーソル変数（SQL 識別子です。ホスト変数ではありません）。

statement_id

未定義の SQL 識別子（PREPARE 文で使うものと同じです）。

例

```
EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC.
```

カーソルの OPEN

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]
   [INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] desc_nam2] ]
END-EXEC.
```

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。

GLOBAL | LOCAL

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。デフォルトは LOCAL です。

dyn_cursor

カーソル変数。

desc_nam1、desc_nam2

オプションの記述子名。

使用上の注意

カーソルに対応付けられた準備済みの文にコロンまたは疑問符が含まれる場合は、USING 句を指定しなければなりません。指定しなかった場合は、実行時にエラーが生成されます。INTO 句は、DML RETURNING をサポートします（5-8 ページの「行の挿入」およびその後の DELETE と UPDATE に関する項を参照してください）。

例

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :BINDDES END-EXEC.

EXEC SQL FOR :limit OPEN C2 USING DESCRIPTOR :b1, :b2
  INTO SQL DESCRIPTOR :seldes
END-EXEC.
```

FETCH

用途

動的 DECLARE 文で宣言し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_literal}
END-EXEC.
```

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

GLOBAL は、すべてのプログラム・ファイルで記述子名が認識されることを示します。LOCAL は、記述子名を割り当てたファイル内でだけ記述子名が認識されることを示します。デフォルトは LOCAL です。

desc_nam

記述子名。

使用上の注意

FOR 句のオプションの `array_size` は、ALLOCATE DESCRIPTOR 文で指定した数と等しいか、それより小さくなければなりません。

例

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES' END-EXEC.
```

```
EXEC SQL FOR :arsz FETCH C2 INTO DESCRIPTOR :DESC END-EXEC.
```

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 Oracle 方法 4 と同じです。

構文

```
EXEC SQL CLOSE cursor END-EXEC.
```

変数

cursor

事前に宣言された動的カーソル。

例

```
EXEC SQL CLOSE C1 END-EXEC.
```

Oracle 動的方法 4 との違い

ANSI 動的 SQL インタフェースでは、Oracle 動的方法 4 でサポートされる機能がすべてサポートされます。それ以外に、次のサポートが追加されています。

- ANSI 動的 SQL では、すべてのデータ型（カーソル変数を含む）および LOB 型がサポートされます。
- ANSI モードでは、内部の SQL 記述領域が使用されます。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。
- 新しい埋込み SQL 文が導入されています。次のものがあります。ALLOCATE DESCRIPTOR、DEALLOCATE DESCRIPTOR、DESCRIBE、GET DESCRIPTOR、SET DESCRIPTOR。
- 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なければなりません。
- 出力では、DESCRIBE のオプションの SELECT LIST FOR 句がオプション・キーワード OUTPUT に変更されています。INTO 句は USING DESCRIPTOR 句に変更されています。USING DESCRIPTOR 句にはオプション・キーワード SQL を使用できます。

- 入力では、DESCRIBE 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 句に変更されています。USING DESCRIPTOR 句にはオプション・キーワード SQL を使用できます。
- 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。

制限

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 2 つの動的方法を併用することはできません。
- プリコンパイラ・オプション DYNAMIC を ANSI に設定しなければなりません。DYNAMIC を ANSI に設定したときだけ、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しかサポートしません。

サンプル・プログラム : SAMPLE12.PCO

次の ANSI SQL 動的方法 4 プログラム SAMPLE12.PCO は、デモ・ディレクトリに入っています。SAMPLE12 は、ユーザーが入力する SQL 文を要求することによって SQL*Plus を模造したプログラムです。プログラムの流れの詳細は、最初のコメントを参照してください。

```
*********************************************************************
* Sample Program 12: Dynamic SQL Method 4 using ANSI Dynamic SQL *
*                                                                   *
* This program shows the basic steps required to use dynamic     *
* SQL Method 4 with ANSI Dynamic SQL.  After logging on to        *
* ORACLE, the program prompts the user for a SQL statement,       *
* PREPAREs the statement, DECLAREs a cursor, checks for any       *
* bind variables using DESCRIBE INPUT, OPENs the cursor, and      *
* DESCRIBEs any select-list variables.  If the input SQL          *
* statement is a query, the program FETCHes each row of data,     *
* then CLOSEs the cursor.                                         *
* use option dynamic=ansi when precompiling this sample.          *
*********************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  ANSIDYNSQL4.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  USERNAME       PIC X(20).
 01  PASSWD         PIC X(20).
```

```
01  BDSC           PIC X(6) VALUE "BNDDSC".
01  SDSC           PIC X(6) VALUE "SELDSC".
01  BNDCNT         PIC S9(9) COMP.
01  SELCNT         PIC S9(9) COMP.
01  BNDNAME        PIC X(80).
01  BNDVAL         PIC X(80).
01  SELNAME        PIC X(80) VARYING.
01  SELDATA        PIC X(80).
01  SELTYP         PIC S9(4) COMP.
01  SELPREC        PIC S9(4) COMP.
01  SELLEN         PIC S9(4) COMP.
01  SELIND         PIC S9(4) COMP.
01  DYN-STATEMENT  PIC X(80).
01  BND-INDEX      PIC S9(9) COMP.
01  SEL-INDEX      PIC S9(9) COMP.
01  VARCHAR2-TYP   PIC S9(4) COMP VALUE 1.
01  VAR-COUNT      PIC 9(2).
01  ROW-COUNT      PIC 9(4).
01  NO-MORE-DATA   PIC X(1) VALUE "N".
01  TMPLEN         PIC S9(9) COMP.
01  MAX-LENGTH     PIC S9(9) COMP VALUE 80.

    EXEC SQL INCLUDE SQLCA         END-EXEC.

PROCEDURE DIVISION.
START-MAIN.

    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

    DISPLAY "USERNAME: " WITH NO ADVANCING.

    ACCEPT USERNAME.

    DISPLAY "PASSWORD: " WITH NO ADVANCING.

    ACCEPT PASSWD.

    EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWD END-EXEC.
    DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.

*   ALLOCATE THE BIND AND SELECT DESCRIPTORS.

    EXEC SQL ALLOCATE DESCRIPTOR :BDSC WITH MAX 20 END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR :SDSC WITH MAX 20 END-EXEC.

*   GET A SQL STATEMENT FROM THE OPERATOR.
```

```
     DISPLAY "ENTER SQL STATEMENT WITHOUT TERMINATOR:".
     DISPLAY ">" WITH NO ADVANCING.

     ACCEPT DYN-STATEMENT.

     DISPLAY " ".

*    PREPARE THE SQL STATEMENT AND DECLARE A CURSOR.

     EXEC SQL  PREPARE S1 FROM :DYN-STATEMENT  END-EXEC.
     EXEC SQL  DECLARE C1 CURSOR FOR S1        END-EXEC.

*    DESCRIBE BIND VARIABLES.

     EXEC SQL DESCRIBE INPUT S1 USING DESCRIPTOR :BDSC END-EXEC.

     EXEC SQL GET DESCRIPTOR :BDSC :BNDCNT = COUNT END-EXEC.

     IF BNDCNT < 0
         DISPLAY "TOO MANY BIND VARIABLES."
         GO TO END-SQL
     ELSE
         DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
         MOVE BNDCNT TO VAR-COUNT
         DISPLAY VAR-COUNT
*        EXEC SQL SET DESCRIPTOR :BDSC COUNT = :BNDCNT END-EXEC
     END-IF.

     IF BNDCNT = 0
         GO TO DESCRIBE-ITEMS.
     PERFORM SET-BND-DSC
             VARYING BND-INDEX FROM 1 BY 1
             UNTIL BND-INDEX > BNDCNT.

*    OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.

 DESCRIBE-ITEMS.
     EXEC SQL  OPEN C1 USING DESCRIPTOR :BDSC END-EXEC.

     EXEC SQL  DESCRIBE OUTPUT S1 USING DESCRIPTOR :SDSC  END-EXEC.

     EXEC SQL GET DESCRIPTOR :SDSC :SELCNT = COUNT END-EXEC.

     IF SELCNT < 0
         DISPLAY "TOO MANY SELECT-LIST ITEMS."
         GO TO END-SQL
     ELSE
```

```
            DISPLAY "NUMBER OF SELECT-LIST ITEMS: "
                WITH NO ADVANCING
            MOVE SELCNT TO VAR-COUNT
            DISPLAY VAR-COUNT
            DISPLAY " "
*           EXEC SQL SET DESCRIPTOR :SDSC COUNT = :SELCNT END-EXEC
        END-IF.

*       SET THE INPUT DESCRIPTOR

        IF SELCNT > 0
            PERFORM SET-SEL-DSC
                VARYING SEL-INDEX FROM 1 BY 1
                UNTIL SEL-INDEX > SELCNT
            DISPLAY " ".

*       FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE.

        IF SELCNT > 0
            PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y".

        DISPLAY " "
        DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING.
        MOVE SQLERRD(3) TO ROW-COUNT.
        DISPLAY ROW-COUNT.

*       CLEAN UP AND TERMINATE.

        EXEC SQL CLOSE C1 END-EXEC.
        EXEC SQL DEALLOCATE DESCRIPTOR :BDSC END-EXEC.
        EXEC SQL DEALLOCATE DESCRIPTOR :SDSC END-EXEC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        DISPLAY " ".
        DISPLAY "HAVE A GOOD DAY!".
        DISPLAY " ".
        STOP RUN.

*       DISPLAY ORACLE ERROR MESSAGE AND CODE.

    SQL-ERROR.
        DISPLAY " ".
        DISPLAY SQLERRMC.
    END-SQL.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        STOP RUN.
```

```
*    PERFORMED SUBROUTINES BEGIN HERE:

*    SET A BIND-LIST ELEMENT'S ATTRIBUTE
*    LET THE USER FILL IN THE BIND VARIABLES AND
*    REPLACE THE 0S DESCRIBED INTO THE DATATYPE FIELDS OF THE
*    BIND DESCRIPTOR WITH 1S TO AVOID AN "INVALID DATATYPE"
*    ORACLE ERROR
 SET-BND-DSC.
         EXEC SQL GET DESCRIPTOR :BDSC VALUE
                :BND-INDEX :BNDNAME = NAME END-EXEC.
         DISPLAY "ENTER VALUE FOR ", BNDNAME.

         ACCEPT BNDVAL.

         EXEC SQL SET DESCRIPTOR :BDSC VALUE :BND-INDEX
           TYPE = :VARCHAR2-TYP, LENGTH = :MAX-LENGTH,
           DATA = :BNDVAL END-EXEC.

* SET A SELECT-LIST ELEMENT'S ATTRIBUTES
 SET-SEL-DSC.
     MOVE SPACES TO SELNAME-ARR.
     EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
           :SELNAME = NAME, :SELTYP = TYPE,
           :SELPREC = PRECISION, :SELLEN = LENGTH END-EXEC.

*    DISPLAY COLUMN HEADING.
     DISPLAY SELNAME-ARR(1:SELNAME-LEN), "  " WITH NO ADVANCING.

*    IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
     IF SELTYP = 12
         MOVE 9 TO SELLEN.

*    IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
     MOVE 0 TO TMPLEN.
     IF SELTYP = 2 AND SELPREC = 0
         MOVE 40 TO TMPLEN.
     IF SELTYP = 2 AND SELPREC > 0
         ADD 2 TO SELPREC
         MOVE SELPREC TO TMPLEN.

     IF SELTYP = 2
         IF TMPLEN > MAX-LENGTH
             DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
             GO TO END-SQL
         ELSE
             MOVE TMPLEN TO SELLEN.
```

```
* COERCE DATATYPES TO VARCHAR2.
     MOVE 1 TO SELTYP.

     EXEC SQL SET DESCRIPTOR :SDSC VALUE :SEL-INDEX
        TYPE = :SELTYP, LENGTH = :SELLEN END-EXEC.

* FETCH A ROW AND PRINT THE SELECT-LIST VALUE.

 FETCH-ROWS.
     EXEC SQL  FETCH C1 INTO DESCRIPTOR :SDSC END-EXEC.
     IF SQLCODE NOT = 0
         MOVE "Y" TO NO-MORE-DATA.
     IF SQLCODE = 0
         PERFORM PRINT-COLUMN-VALUES
             VARYING SEL-INDEX FROM 1 BY 1
             UNTIL SEL-INDEX > SELCNT
         DISPLAY " ".

* PRINT A SELECT-LIST VALUE.

 PRINT-COLUMN-VALUES.
     MOVE SPACES TO SELDATA.
     EXEC SQL GET DESCRIPTOR :SDSC VALUE :SEL-INDEX
        :SELDATA = DATA, :SELIND = INDICATOR,
        :SELLEN = RETURNED_LENGTH END-EXEC.
     IF (SELIND = -1)
        DISPLAY "NULL  " WITH NO ADVANCING
     ELSE
        DISPLAY SELDATA(1:SELLEN), "  "
           WITH NO ADVANCING.
```

11

Oracle 動的 SQL: 方法 4

この章では Oracle 動的 SQL 方法 4 を実装する方法について説明します。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、作成したりできます。含まれるホスト変数の個数はさまざまです。

新しいアプリケーションは、第 10 章の「ANSI 動的 SQL」に示した最新の ANSI SQL 方法 4 を使用して開発してください。ANSI 方法 4 では、Oracle のすべての型をサポートしています。しかし、古い Oracle 方法 4 では、カーソル変数、グループ項目の表、DML 戻り句、LOB をサポートしていません。

この章では次の事項について説明します。

- 方法 4 の特殊要件
- SQL 記述子領域（SQLDA）の理解
- SQLDA 変数
- 予備知識
- 基本手順
- 各手順の詳細
- 方法 4 でのホスト配列の使用
- サンプル・プログラム 10: 動的 SQL 方法 4

注意 : 動的 SQL 方法 1、2、3 の詳細と方法 4 の概要は、第 9 章の「Oracle 動的 SQL」を参照してください。

方法 4 の特殊要件

方法 4 の要件について学ぶ前に、選択リスト項目およびプレースホルダという用語を十分に理解してください。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、実際のバインド変数用 SQL 文の場所を確保する、ダミーのバインド（入力）変数のことです。プレースホルダの宣言は行いません。また、プレースホルダには任意の名前を付けることができます。バインド変数のプレースホルダは、SET、VALUES および WHERE 句でよく使用されます。たとえば、次の動的 SQL 文にはそれぞれ 2 つのプレースホルダが記述されています。

```
INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:E, :D)
DELETE FROM DEPT WHERE DEPTNO = :DNUM AND LOC = :DLOC
```

プレースホルダでは、表や列の名前は参照できません。

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。

- 選択リスト項目の数またはプレースホルダの数が不明な動的 SQL 文の受け入れまたは作成をする
- Oracle8*i* と COBOL の間のデータ型の変換を明示的に制御する

このような柔軟性をプログラムに加えるには、ランタイム・ライブラリに情報を追加する必要があります。

データベースに必要な情報

Pro*COBOL は、実行可能な動的 SQL 文すべてについて Oracle8 のコールを生成します。データベースは、選択リスト項目もプレースホルダも組み込まれていない動的 SQL 文の実行には追加情報を必要としません。次の DELETE 文がこのカテゴリに該当します。

```
*    Dynamic SQL statement...
     MOVE 'DELETE FROM EMP WHERE DEPTNO = 30' TO STMT.
```

ところが、ほとんどの動的 SQL 文には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組み込まれています。次の UPDATE 文はその一例です。

```
*    Dynamic SQL statement with place-holders...
     MOVE 'UPDATE EMP SET COMM = :C WHERE EMPNO = :E' TO STMT.
```

選択リスト項目またはバインド変数のプレースホルダ（あるいはその両方）が組み込まれている動的 SQL 文を実行する場合、データベースは出力値または入力値を格納するプログラム変数についての情報を必要とします。データベースでは、特に次の情報が必要です。

- 選択リスト項目の数とバインド変数の数
- 各選択リスト項目の長さおよび各バインド変数の長さ
- 各選択リスト項目のデータ型および各バインド変数のデータ型
- 選択リスト項目の値を格納する各出力変数のメモリー・アドレスおよび各バインド変数のアドレス

たとえば、選択リスト項目の値を書き込むには、データベースは対応する出力変数のアドレスを必要とします。

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関してデータベースが必要とする情報は、（その値を除いて）すべて SQL 記述子領域（SQLDA）と呼ばれるプログラム・データ構造に格納されます。

選択リスト項目の記述は選択 SQLDA に格納され、バインド変数のプレースホルダの記述はバインド SQLDA に格納されます。

選択リスト項目の値は出力バッファに格納され、バインド変数の値は入力バッファに格納されます。これらのデータ・バッファのアドレスをライブラリ・ルーチン SQLADR を使って選択 SQLDA またはバインド SQLDA に格納すると、データベースは出力値の書込み先や入力値の読込み元を認識できるようになります。

値はどのようにしてこれらのデータ変数に代入されるのでしょうか。出力値はカーソルを使って 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。

情報の取得方法

データベースが必要とする情報を取得するには、DESCRIBE 文を使います。DESCRIBE SELECT LIST 文は、各選択リスト項目を調べて、その名前およびデータ型、制約、長さ、位取り、精度を確認後、それらの情報を選択 SQLDA に格納します。たとえば、プレースホルダ名を使って入力ホスト変数の値をユーザーに入力要求できます。DESCRIBE では、選択リスト項目の総数も 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、その名前と長さを確認後、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。

SQL 記述子領域（SQLDA）の理解

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。

SQLDA の目的

方法 4 は、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動的 SQL 文の処理に必要とされます。このような動的 SQL 文を処理するには、プログラムで SQLDA（記述子とも呼ばれる）を明示的に宣言する必要があります。個々の記述子は、プログラム内のグループ項目に対応します。

選択記述子には、選択リスト項目の記述、および選択リスト項目の名前と値を格納する出力バッファのアドレスが格納されます。

> **注意 :** 選択リスト項目の名前は、列名、列の別名、SAL+COMM などの表現テキストのどれでもかまいません。

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を格納します。

記述子変数の中には、値ではなくアドレスを格納するものがあります。このため、値を保存するためのデータ・バッファを宣言する必要があります。必要な入力バッファおよび出力バッファのサイズは自由に決められます。COBOL ではポインタはサポートされていないので、入力バッファおよび出力バッファのアドレスを取得するにはライブラリ・サブルーチン SQLADR を使用しなければなりません。SQLADR のコール方法については、11-12 ページの「SQLADR の使用」を参照してください。

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、同時にオープンされている 3 つのカーソルから FETCH するために、SELDSC1、SELDSC2、SELDSC3 という名前の 3 つの選択 SQLDA を宣言できます。ただし、カーソルが同時に実行されなければ SQLDA を再利用できます。

SQLDA の宣言

選択およびバインド SQLDA を宣言するには、図 11-1 に示す選択およびバインド SQLDA のサンプルを使用して、プログラムに記述する方法があります。配列の大きさは、必要に応じて変更できます。

注意 : バイトスワップ・プラットフォームの場合は、SQLDA の宣言時に COMP のかわりに COMP5 を使います。

図 11-1 Pro*COBOL SQLDA 記述子とデータ・バッファの例

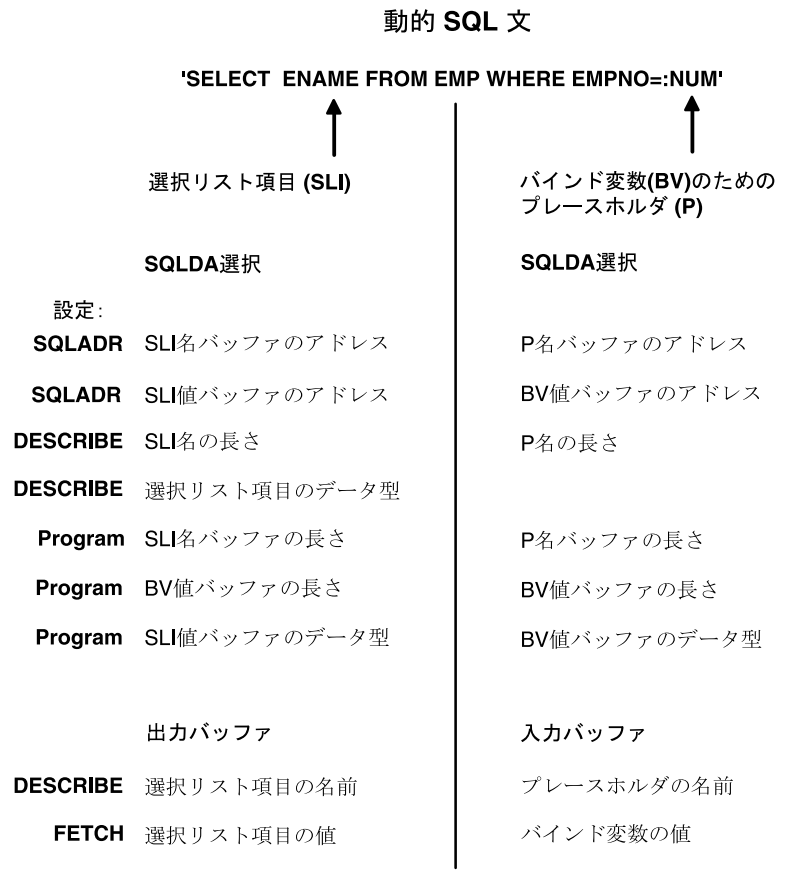
```
01 SELDSC.
   05   SQLDNUM                    PIC S9(9) COMP.
   05   SQLDFND                    PIC S9(9) COMP.
   05   SELDVAR                    OCCURS 20 TIMES.
        10   SELDV                 PIC S9(9) COMP.
        10   SELDFMT               PIC S9(9) COMP.
        10   SELDVLN               PIC S9(9) COMP.
        10   SELDFMTL              PIC S9(4) COMP.
        10   SELDVTYP              PIC S9(4) COMP.
        10   SELDI                 PIC S9(9) COMP.
        10   SELDH-VNAME           PIC S9(9) COMP.
        10   SELDH-MAX-VNAMEL      PIC S9(4) COMP.
        10   SELDH-CUR-VNAMEL      PIC S9(4) COMP.
        10   SELDI-VNAME           PIC S9(9) COMP.
        10   SELDI-MAX-VNAMEL      PIC S9(4) COMP.
        10   SELDI-CUR-VNAMEL      PIC S9(4) COMP.
        10   SELDFCLP              PIC S9(9) COMP.
        10   SELDFCRCP             PIC S9(9) COMP.
01  XSELDI.
    05 SEL-DI          OCCURS 20 TIMES PIC S9(4) COMP.
01  XSELDIVNAME.
    05 SEL-DI-VNAME    OCCURS 20 TIMES PIC X(80).
01  XSELDV.
    05 SEL-DV          OCCURS 20 TIMES PIC X(80).
01  XSELDHVNAME
    05 SEL-DH-VNAME    OCCURS 20 TIMES PIC X(80).
01  XSEL-DFMT          PIC X(6).
01 BNDDSC.
   05   SQLDNUM                    PIC S9(9) COMP.
   05   SQLDFND                    PIC S9(9) COMP.
   05   BNDDVAR                    OCCURS 20 TIMES.
        10   BNDDV                 PIC S9(9) COMP.
        10   BNDDFMT               PIC S9(9) COMP.
        10   BNDDVLN               PIC S9(9) COMP.
        10   BNDDFMTL              PIC S9(4) COMP.
        10   BNDDVTYP              PIC S9(4) COMP.
        10   BNDDI                 PIC S9(9) COMP.
        10   BNDDH-VNAME           PIC S9(9) COMP.
        10   BNDDH-MAX-VNAMEL      PIC S9(4) COMP.
        10   BNDDH-CUR-VNAMEL      PIC S9(4) COMP.
        10   BNDDI-VNAME           PIC S9(9) COMP.
        10   BNDDI-MAX-VNAMEL      PIC S9(4) COMP.
        10   BNDDI-CUR-VNAMEL      PIC S9(4) COMP.
        10   BNDDFCLP              PIC S9(9) COMP.
        10   BNDDFCRCP             PIC S9(9) COMP.
01  XBNDDI.
    05 BND-DI          OCCURS 20 TIMES PIC S9(4) COMP.
01  XBNDDINAME.
    05 BND-DI-VNAME    OCCURS 20 TIMES PIC X(80).
01  XBNDDV.
    05 BND-DV          OCCURS 20 TIMES PIC X(80).
01  XBNDDHVNAME
    05 BND-DH-VNAME    OCCURS 20 TIMES PIC X(80).
01  XBND-DFMT          PIC X(6).
```

次に示すように、SQLDA を（たとえば SELDSC および BNDDSC という名前の）ファイルに格納して、INCLUDE 文を使ってそのファイルをプログラムにコピーできます。

```
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

図 11-2 に、変数が SQLADR コール、DESCRIBE コマンド、FETCH コマンド、プログラム代入のどれによって設定されるかを示します。

図 11-2　変数の設定方法



SQLDA 変数

この項では、SQLDA 内の各変数の用途と使用方法を説明します。

SQLDNUM

この変数により、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が指定されます。したがって、SQLDNUM によって記述子表の要素の数（ディメンション）が決まります。

DESCRIBE コマンドを発行する前に、この変数を記述子表のサイズに設定しなければなりません。また、DESCRIBE の実行後に、この変数を実際に DESCRIBE された変数の数（これは SQLDFND に格納されます）に再設定する必要があります。

SQLDFND

これは、DESCRIBE コマンドで実際に検出された選択リスト項目またはプレースホルダの数です。

SQLDFND は DESCRIBE によって設定されます。SQLDFND が負の値の場合は、DESCRIBE コマンドが検出した選択リスト項目またはプレースホルダの数が、記述子のサイズに対して多すぎることを意味します。たとえば、SQLDNUM を 10 に設定した場合に DESCRIBE で 11 個の選択リスト項目またはプレースホルダが検索されると、SQLDFND は -11 に設定されます。この場合、記述子を再び割り当てないかぎり SQL 文の処理はできません。

DESCRIBE の後で、SQLDNUM を SQLDFND の値に設定してください。

SELDV | BNDDV

これは、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の表です。

SELDV および BNDDV の要素は、SQLADR を使って設定する必要があります。

選択記述子の場合

次の文は、

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

FETCH された選択リスト変数を、SELDV（1）から SELDV（SQLDNUM）が指しているデータ・バッファに格納するようにデータベースに指示します。したがって、データベースは J 番目の選択リスト値を SEL-DV(J) に格納します。

バインド記述子の場合

この表は、OPEN コマンドを発行する前に設定する必要があります。次の文は、

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

BNDDV（1）～ BNDDV（SQLDNUM）でアドレス指定されたバインド変数の値を使って動的 SQL 文を実行するように、Oracle8 に指示します。（通常、値はユーザーによって入力されます。）データベースは、J 番目のバインド変数値を BND-DV（J）から検索します。

SELDFMT | BNDDFMT

これは、選択リストまたはバインド変数の変換形式文字列を格納するデータ・バッファのアドレスの表です。現在は、COBOL のパック 10 進数にだけ使用できます。変換文字列の形式は PP.+SS または PP.-SS です。PP は精度を示し、SS は位取りを示します。精度と位取りの定義については、11-18 ページの「精度と位取りの抽出」を参照してください。

形式文字列を使用するかどうかは任意です。J 番目の選択リスト項目またはバインド変数に変換形式を使用する場合は、SQLADR を使って SELDFMT(J) または BNDDFMT(J) を設定し、パック 10 進形式（"07.+02" など）を SEL-DFMT または BND-DFMT に格納します。変換形式を使用しない場合は、SELDFMT(J) または BNDDFMT(J) を 0（ゼロ）に設定してください。

SELDVLN | BNDDVLN

これは、データ・バッファに格納される選択リスト値またはバインド変数値の長さの表です。

選択記述子の場合

この表は、DESCRIBE SELECT LIST によって、各選択リスト項目に期待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 コマンドを実行する前の SELDVLN(J) の値です。）

長さの形式はデータ型によって異なります。CHAR 型の選択リスト項目の場合は、DESCRIBE SELECT LIST は SELDVLN(J) をその選択リスト項目の最大長（バイト数）に設定します。NUMBER 型の選択リスト項目については、位取りと精度が変数の下位バイトとその次の上位側バイトにそれぞれ戻されます。ライブラリ・ルーチン SQLPRC を使って、SELDVLN から精度と位取りを抽出できます。詳細は 11-18 ページの「精度と位取りの抽出」を参照してください。

FETCH を実行する前に、SELDVLN(J) 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 型の数値を COBOL の文字列に強制変換するときには、SELDVLN(J) を、その数値の精度に符号と小数点のために 2 を加えた長さに設定してください。また、NUMBER 型の数値を COBOL の浮動小数点数に強制変換するときには、SELDVLN(J) を、使用しているシステムでの適切な浮動小数点型の長さに設定してくださ

い。強制変換するデータ型の長さの詳細は、11-13 ページの「データの変換」を参照してください。

バインド記述子の場合

OPEN コマンドを発行する前に、この長さの表を設定する必要があります。たとえば、次の文を使って、ユーザーが入力したバインド変数文字列の長さを設定できます。

```
PROCEDURE DIVISION.
     ...
     PERFORM GET-INPUT-VAR
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
     ...
GET-INPUT-VAR.
     DISPLAY "Enter value of ", BND-DH-VNAME(J).
     ACCEPT INPUT-STRING.
     UNSTRING INPUT-STRING DELIMITED BY "  "
         INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

Oracle8*i* は、SELDV(J) または BNDDV(J) に格納されているアドレスを使って間接的にデータ・バッファにアクセスするので、データ・バッファ内の値の長さは認識しません。J 番目の選択リスト値またはバインド変数値に対して Oracle8*i* が使用する長さを変更する場合は、SELDVLN(J) または BNDDVLN(J) 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。

SELDFMTL|BNDDFMTL

これは、選択リストまたはバインド変数の変換形式文字列の長さの表です。現在は、COBOL のパック 10 進数にだけ使用できます。

形式文字列を使用するかどうかは任意です。J 番目の選択リスト項目に変換形式を使用する場合は、FETCH の前に、SELDFMTL(J) を SEL-DFMT に格納されているパック 10 進形式の長さに設定します。J 番目のバインド変数に変換形式を使用したい場合は、OPEN の前に、BNDDFMTL(J) を BND-DFMT に格納されているパック 10 進形式の長さに設定します。変換形式を使用しない場合は、SELDFMTL(J) または BNDDFMTL(J) を 0（ゼロ）に設定してください。

SELDFMTL(J) の値が 0（ゼロ）の場合は、SELDFMTL(J) は使用されません。同様に、BNDDFMT(J) の値が 0（ゼロ）の場合は、BNDDFMT(J) は使用されません。

SELDVTYP | BNDDVTYP

これは、選択リスト値またはバインド変数値のデータ型コードの表です。データ型コードによって、SELDV の要素でアドレス指定されたデータ・バッファに格納されたときに Oracle8*i* データがどのように変換されるかが決まります。詳細は11-13 ページの「データの変換」を参照してください。

選択記述子の場合

DESCRIBE SELECT LIST により、この表は、選択リスト項目の内部データ型（VARCHAR2、CHAR、NUMBER、DATE など）に設定されます。

データ型の内部形式は処理が難しいので、FETCH を行う前にデータ型を再設定することをお薦めします。表示用には、選択リスト値のデータ型を VARCHAR2 に強制変換するのが一般的です。計算用には、数値を Oracle8*i* から COBOL 書式に強制変換することができます。詳細は 11-17 ページの「データ型の強制変換」を参照してください。

SELDVTYP(J) の上位ビットは、J 番目の選択リスト列の NULL / NOT NULL 状態を示します。OPEN コマンドまたは FETCH コマンドを発行する前に、常にこのビットをオフにする必要があります。ライブラリ・ルーチン SQLNUL を使って、データ型コードを取り出し、NULL / NOT NULL ビットをクリアできます。詳細は 11-19 ページの「NULL または NOT NULL データ型の処理」を参照してください。

NUMBER 内部データ型は、SELDV(J) でアドレス指定された COBOL データ・バッファの外部データ型と互換性のある外部データ型に変更してください。

バインド記述子の場合

DESCRIBE BIND VARIABLES は、この表を 0（ゼロ）に設定します。OPEN コマンドを発行する前に、このデータ型の表を再設定する必要があります。コードは、BNDDV(J) でアドレス指定されたバッファの外部（COBOL）データ型を表します。多くの場合、バインド変数値は文字列に格納されるので、データ型表の要素は 1（VARCHAR2 データ型のコード）に設定されます。

J 番目の選択リスト値またはバインド変数値のデータ型を変更するには、SELDVTYP(J) または BNDDVTYP(J) を希望するデータ型に再設定してください。

SELDI | BNDDI

これは、標識変数の値を格納するデータ・バッファのアドレスの表です。SELDI または BNDDI の要素は、SQLADR を使って設定する必要があります。

選択記述子の場合

この表は、FETCH コマンドを発行する前に設定する必要があります。Oracle8*i* が次の文を実行すると、

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

戻された選択リストの J 番目の値が NULL の場合は、SELDI(J) でアドレス指定されたバッファが -1 に設定されます。それ以外の場合は、0（値が NULL でない）または正の整数（値が切り捨てられている）に設定されます。

バインド記述子の場合

OPEN コマンドを実行する前にこの表を初期化し、対応する標識変数を設定する必要があります。Oracle8*i* が次の文を実行すると、

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

BNDDI(J) でアドレス指定されたバッファによって、J 番目のバインド変数が NULL かどうかがわかります。標識変数の値が -1 の場合、対応するバインド変数は NULL です。

SELDH-VNAME | BNDDH-VNAME

これは、動的 SQL 文に表示される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・バッファのアドレスの表です。SELDH-VNAME または BNDDH-VNAME の要素は、DESCRIBE コマンドを発行する前に SQLADR を使って設定する必要があります。

DESCRIBE は、J 番目の選択リスト項目またはプレースホルダの名前を SELDH-VNAME(J) または BNDDH-VNAME(J) でアドレス指定されたデータ・バッファに格納するように、Oracle8*i* に指示します。Oracle8*i* は、J 番目の選択リストまたはプレースホルダの名前を SEL-DH-VNAME(J) または BND-DH-VNAME(J) に格納します。

SELDH-MAX-VNAMEL | BNDDH-MAX-VNAMEL

これは、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・バッファの最大長の表です。バッファは SELDH-VNAME または BNDDH-VNAME の要素によってアドレス指定されます。

SELDH-MAX-VNAMEL または BNDDH-MAX-VNAMEL の要素は、DESCRIBE コマンドを発行する前に設定する必要があります。選択リスト名のバッファまたはプレースホルダ名のバッファは、それぞれ長さが異なっていてもかまいません。

SELDH-MAX-VNAMEL | BNDDH-MAX-VNAMEL

これは、選択リスト名またはプレースホルダ名の実際の長さの表です。DESCRIBE はこの表を、各選択リスト名またはプレースホルダ名の文字数に設定します。

SELDI-VNAME | BNDDI-VNAME

これは、標識変数名を格納するデータ・バッファのアドレスの表です。

標識変数の値は、選択リスト項目およびバインド変数と対応付けできますが、ただし標識変数の名前は、バインド変数だけに対応付けることができます。したがって、この表はバインド記述子でだけ使用できます。SELDH-VNAME または BNDDH-VNAME の要素は、DESCRIBE コマンドを発行する前に SQLADR を使って設定する必要があります。

DESCRIBE BIND VARIABLES は、標識変数名を BNDDI-VNAME(1) ～ BNDDI-VNAME(SQLDNUM) でアドレス指定されたデータ・バッファに格納するように、Oracle8*i* に指示します。Oracle8*i* は、J 番目の標識変数名を BND-DI-VNAME(J) に格納します。

SELDI-MAX-VNAMEL | BNDDI-MAX-VNAMEL

これは、標識変数名を格納するデータ・バッファの最大長の表です。バッファは、SELDI-VNAME または BNDDI-VNAME の要素によってアドレス指定されます。

標識変数名はバインド変数とだけ対応付けできます。したがって、この表はバインド記述子でだけ使用できます。

要素 BNDDI-MAX-VNAMEL(1) 〜 BNDDI-MAX-VNAMEL(SQLDNUM) は、DESCRIBE コマンドの発行前に設定する必要があります。標識変数名のバッファは、それぞれ長さが異なっていてもかまいません。

SELDI-CUR-VNAMEL | BNDDI-CUR-VNAMEL

これは、標識変数名の実際の長さを格納した表です。標識変数名はバインド変数とだけ対応付けできます。したがって、この表はバインド記述子でだけ使用できます。

DESCRIBE BIND VARIABLES は、この表を、各標識変数名の文字数に設定します。

SELDFCLP | BNDDFCLP

これは、将来の使用のために確保されている表です。Oracle8*i* はグループ項目 SELDSC または BNDDSC が特定のサイズであるとみなすため、この表が必要となります。SELDFCLP または BNDDFCLP の要素を 0（ゼロ）に設定する必要があります。

SELDFCRCP | BNDDFCRCP

これは、将来の使用のために確保されている表です。Oracle8*i* はグループ項目 SELDSC または BNDDSC が特定のサイズであるとみなすため、この表が必要となります。SELDFCRCP または BNDDFCRCP の要素を 0（ゼロ）に設定する必要があります。

予備知識

動的 SQL 方法 4 を実現するには次の処理についての知識が必要です。

- ライブラリ・ルーチン SQLADR の使用
- 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

SQLADR の使用

入力値および出力値を格納するデータ・バッファのアドレスを取得するには、ライブラリ・サブルーチン SQLADR をコールする必要があります。取得したアドレスをバインド SQLDA または選択 SQLDA に格納すると、Oracle8*i* はバインド変数値の読込み元や選択リスト値の書込み先を認識できるようになります。

次の構文で、SQLADR をコールします。

```
CALL "SQLADR" USING BUFFER, ADDRESS.
```

パラメータは次のとおりです。

BUFFER
選択リスト項目またはバインド変数、標識変数の値または名前を格納するデータ・バッファ。

アドレス
データ・バッファのアドレスを戻す整変数。

SQLADR をコールすると、BUFFER のアドレスが ADDRESS に格納されます。次の例では、SQLADR を使って、選択記述子表 SELDV、SELDH-VNAME、SELDI を初期化します。これらの表の要素によって、選択リスト値、選択リスト名、標識値のデータ・バッファのアドレスが指定されます。

```
PROCEDURE DIVISION.
    ...
    PERFORM INIT-SELDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    ...
INIT-SELDSC.
    CALL "SQLADR" USING SEL-DV(J), SELDV(J).
    CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).
    CALL "SQLADR" USING SEL-DI(J), SELDI(J).
```

データの変換

この項では、データ型記述子表について詳しく説明します。データ型の同値化も動的 SQL 方法 4 も使用しないホスト・プログラムでは、内部データ型と外部データ型の間の変換はプリコンパイル時に決定されます。デフォルトでは、Pro*COBOL によって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。たとえば、型 PIC S9(n) COMP のホスト変数には INTEGER 外部データ型が割り当てられます。

しかし方法 4 を使うと、データの変換および形式を制御できます。変換の指定は、データ型記述子表のデータ型コードを設定することによって行います。

内部データ型

内部データ型により、Oracle8*i* がデータベース表への列値の格納に使う形式および疑似列値の表現に使う形式が指定されます。

DESCRIBE SELECT LIST コマンドを発行すると、Oracle8*i* は各選択リスト項目の内部データ型コードを SELDVTYP（データ型）記述子表に戻します。たとえば、J 番目の選択リスト項目のデータ型コードは SELDVTYP(J) に戻されます。

表 11 に、内部のデータ型とそのコードを示します。

表 11-1 内部のデータ型と関連コード

内部データ型	コード
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR	96

外部データ型

外部データ型には、入力ホスト変数および出力ホスト変数に値を格納するのに使う形式を指定します。

DESCRIBE BIND VARIABLES コマンドは、データ型コードの BNDDVTYP 表を 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、さまざまなバインド変数にどの外部データ型が使用されるかを Oracle8*i* に知らせます。J 番目のバインド変数の外部データ型を指定するには、BNDDVTYP(J) を希望する外部データ型に再設定してください。

次の表は、外部データ型とそのコード、および対応する COBOL データ型を示したものです。

表 11-2 Oracle の外部データ型および関連する COBOL データ型

名前	コード	COBOL データ型
VARCHAR2	1	PIC X(*n*)（MODE=ANSI の場合）
NUMBER	2	PIC X(*n*)
INTEGER	3	PIC S9(*n*) COMP PIC S9(*n*) COMP5 （バイトスワップ・プラットフォームの場合は COMP5）
FLOAT	4	COMP-1 COMP-2
STRING (1)	5	PIC X(*n*)

表 11-2 Oracle の外部データ型および関連する COBOL データ型 (続き)

名前	コード	COBOL データ型
VARNUM	6	PIC X(*n*)
DECIMAL	7	PIC S9(n)V9(n) COMP-3
LONG	8	PIC X(*n*)
VARCHAR (2)	9	PIC X(*n*) VARYING PIC N(*n*) VARYING
ROWID	11	PIC X(*n*)
DATE	12	PIC X(*n*)
VARRAW (2)	15	PIC X(*n*)
RAW	23	PIC X(*n*)
LONG RAW	24	PIC X(*n*)
UNSIGNED	68	（サポートされていない）
DISPLAY	91	PIC S9...9V9...9 DISPLAY SIGN LEADING SEPARATE PIC S9(*n*)V9(*n*) DISPLAY SIGN LEADING SEPARATE
LONG VARCHAR (2)	94	PIC X(*n*)
LONG VARRAW (2)	95	PIC X(*n*)
CHARF	96	PIC X(*n*)（MODE=ANSI の場合） PIC N(*n*)（MODE=ANSI の場合）
CHARZ (1)	97	PIC X(*n*)
CURSOR	102	SQL-CURSOR

注意 :

1. EXEC SQL VAR 文でだけ使います。
2. *n* バイトの長さフィールドを含みます。

データ型とその書式の詳細は、4-2 ページの「Oracle8i のデータ型」を参照してください。

PL/SQL のデータ型

PL / SQL では、各種の事前定義済みスカラー・データ型および複合データ型を使用できます。スカラー型には内部コンポーネントはありません。複合型には、個々に操作できる内部コンポーネントがあります。表 11-3 に、事前定義済みの PL / SQL のスカラー・データ型とそれに

相当する内部データ型を示します。

表 11-3 PL/SQL のデータ型とそれに相当する内部データ型

PL/SQL データ型	Oracle 内部データ型
VARCHAR	VARCHAR2
VARCHAR2	
BINARY_INTEGER	NUMBER
DEC	
DECIMAL	
DOUBLE PRECISION	
FLOAT	
INT	
INTEGER	
NATURAL	
NUMBER	
NUMERIC	
POSITIVE	
REAL	
SMALLINT	
LONG	LONG
ROWID	ROWID
DATE	DATE
RAW	RAW
LONG RAW	LONG RAW
CHAR	CHAR
CHARACTER	
STRING	

データ型の強制変換

選択記述子の場合、DESCRIBE SELECT LIST は内部データ型をどれでも戻すことができます。文字データの場合のように、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が、しか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いくつかあります。このため、SELDVTYP 記述子表の要素の中には再設定したほうがよいものがあります。

たとえば、NUMBER 値を FLOAT 値（これは COBOL の PIC S9*n*V9(*n*) COMP-1 値に対応）に再設定します。Oracle8*i* 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ありません。バインド変数の数と名前だけが戻されます。したがって、データ型コードの BNDDVTYP 表を明示的に設定して、各バインド変数の外部データ型を Oracle8*i* に認識させる必要があります。Oracle8*i* 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。

SELDVTYP または BNDDVTYP の記述子表でデータ型コードをリセットするときに、データ型を強制変換します。たとえば、J 番目の選択リスト値を VARCHAR2 に強制変換するには、次の文を使います。

```
*    Coerce select-list value to VARCHAR2.
     MOVE 1 TO SELDVTYP(J).
```

表示を目的として NUMBER 選択リスト値を VARCHAR2 に強制変換する場合は、その値の精度および位取りのバイトを抽出し、その情報を使って最大表示長を算出する必要があります。その後、FETCH を行う前に SELDVLN（長さ）記述子表の該当する要素を再設定して、使用するバッファの長さを Oracle8*i* に認識させなければなりません。J 番目の選択リスト値の長さを指定するには、SELDVLN(J) を必要な長さに設定します。

たとえば、DESCRIBE SELECT LIST で J 番目の選択リスト項目の型が NUMBER であることがわかった場合、戻り値を PIC S9(*n*)V9(*n*) COMP-1 として宣言した COBOL 変数に格納するには、SELDVTYP(J) を 4 に設定し、SELDVLN(J) をシステムが定める COMP-1 数値の長さに設定します。

例外

DESCRIBE SELECT LIST によって戻される内部データ型が、期待する結果と合わない場合もあります。DATE 型と NUMBER 型がその例です。DATE 選択リスト項目を DESCRIBE すると、Oracle8*i* はデータ型 12 を SELDVTYP 表に戻します。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。デフォルトの文字形式で日付を取得するには、データ型コードを 12 から 1（VARCHAR2）に変更して、SELDVLN の値を 7 から 9 に増やす必要があります。

同様に、NUMBER 選択リスト項目を DESCRIBE すると、Oracle8*i* はデータ型コード 2 を SELDVTYP 表に戻します。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

式で戻されるので、求めている値と異なる可能性が高くなります。このような場合は、コードを 2 から 1（VARCHAR2）、3（INTEGER）、または 4（FLOAT）に変更するか、その他の適切なデータ型に変更してください。

精度と位取りの抽出

ライブラリ・サブルーチン SQLPRC により、精度と位取りが抽出されます。このサブルーチンは通常、DESCRIBE SELECT LIST の後で使用され、その最初のパラメータは SELDVLN(J) です。次の構文で、SQLPRC をコールします。

```
CALL "SQLPRC" USING LENGTH, PRECISION, SCALE.
```

パラメータは次のとおりです。

LENGTH	NUMBER 値の長さが格納される整変数。値の位取りと精度はそれぞれ、下位バイトと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。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。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 の場合は最も近い 100 分の 1 の位に値が四捨五入され（3.456 は 3.46 となります）、位取りが -3 の場合は最も近い 1000 の位に値が四捨五入されます（3.456 は 3000 となります）。

次の例に、SQLPRC を使用して、VARCHAR2 に強制変換される NUMBER 値の最大表示長を算出する方法を示します。

```
 WORKING-STORAGE SECTION.
 01  PRECISION       PIC S9(9) COMP.
 01  SCALE           PIC S9(9) COMP.
 01  DISPLAY-LENGTH  PIC S9(9) COMP.
 01  MAX-LENGTH      PIC S9(9) COMP VALUE 80.
     ...
 PROCEDURE DIVISION.
     ...
     PERFORM ADJUST-LENGTH
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
 ADJUST-LENGTH.
*    If datatype is NUMBER, extract precision and scale.
     IF SELDVTYP(J) = 2
         CALL "SQLPRC" USING SELDVLN(J), PRECISION, SCALE.
     MOVE 0 TO DISPLAY-LENGTH.
*    Precision is set to zero if the select-list item
```

```
*    refers to a NUMBER of unspecified size.  We allow for
*    a maximum precision of 10.
     IF SELDVTYP(J) = 2 AND PRECISION = 0
         MOVE 10 TO DISPLAY-LENGTH.
*    Allow for possible decimal point and sign.
     IF SELDVTYP(J) = 2 AND PRECISION > 0
         ADD 2 TO PRECISION
         MOVE PRECISION TO DISPLAY-LENGTH.
     ...
```

このサブルーチン・コールの最初のパラメータは、選択リストの長さの表の J 番目の要素になるので注意してください。

SQLPRC プロシージャ（これは SQLLIB ランタイム・ライブラリで定義されています）により、特定の SQL データ型については精度と位取りの値として 0（ゼロ）が戻されます。SQLPR2 プロシージャは、この表に示すデータ型を除けば、同じ構文で同じバイナリ値を戻すという点で SQLPRC に似ています。

表 11-4 SQLPR2 プロシージャのデータ型の例外

SQL データ型	2 進数精度	バイナリ位取り
FLOAT	126	-127
FLOAT(*n*)	*n*（範囲は 1 ～ 126）	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

NULL または NOT NULL データ型の処理

DESCRIBE SELECT LIST は、各選択リスト列（式は除く）の NULL / NOT NULL の標識を、選択記述子のデータ型表に戻します。J 番目の選択リスト列の制約が NOT NULL になっている場合、SELDVTYP(J) データ型変数の上位ビットは 0（ゼロ）です。NOT NULL 制約がない場合は、上位ビットは 1 に設定されています。

NULL 状態ビットがセットされていると、OPEN 文または FETCH 文でデータ型を使用する前にクリアしなければならないので、このビットはセットしないでください。

ライブラリ・ルーチン SQLNUL を使用して、列に NULL を使用できるかどうかを調べたり、そのデータ型の NULL 状態ビットをクリアできます。次の構文で、SQLNUL をコールします。

```
CALL "SQLNUL" USING VALUE-TYPE, TYPE-CODE, NULL-STATUS.
```

パラメータは次のとおりです。

VALUE-TYPE 選択リスト列のデータ型コードを戻す 2 バイトの整変数。

TYPE-CODE 選択リスト列のデータ型コードを戻す 2 バイトの整変数。上位ビットはクリアされます。

NULL-STATUS 選択リスト列の NULL 状態を戻す整変数。1 は列が NULL を許可し、0 は許可しないことを意味します。

次の例は、SQLNUL の使用方法を示したものです。

```
 WORKING-STORAGE SECTION.
     ...
*    Declare variable for subroutine call.
     01  NULL-STATUS  PIC S9(9) COMP.
     ...
 PROCEDURE DIVISION.
 MAIN.
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
     PERFORM HANDLE-NULLS
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
     ...
 HANDLE-NULLS.
*    Find out if column is NOT NULL, and clear high-order bit.
     CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.
*    If NULL-STATUS = 1, NULLs are allowed.
```

このサブルーチン・コールの最初のパラメータと 2 番目のパラメータは同じなので注意してください。この 2 つのパラメータはそれぞれ、NULL 状態ビットがクリアされる前と後のデータ型変数です。

基本手順

方法 4 はあらゆる動的 SQL 文に使うことができます。11-38 ページの「方法 4 でのホスト配列の使用」の例では、入力ホスト変数および出力ホスト変数をどのように扱うかがわかるように問合せが処理されています。

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. 問合せテキストを保存するためのホスト文字列を宣言する。
2. 選択記述子とバインド記述子を宣言する。
3. 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する。
4. 選択記述子とバインド記述子を初期化する。
5. 問合せテキストをホスト文字列に格納する。

6. ホスト文字列から問合せを PREPARE する。
7. 問合せ用の（FOR）カーソルを DECLARE する。
8. バインド記述子に（INTO）バインド変数を DESCRIBE する。
9. プレースホルダの数を、DESCRIBE で実際に検出された数に再設定する。
10. DESCRIBE で検出されたバインド変数の値を取得する。
11. バインド記述子を使って（USING）カーソルを OPEN する。
12. 選択記述子に（INTO）選択リストを DESCRIBE する。
13. 選択リスト項目の最大数を DESCRIBE によって実際に検出された数に再設定する。
14. 表示用にそれぞれの選択リスト項目の長さとデータ型を再設定する。
15. 選択記述子を使ってデータベースからデータ・バッファに行を FETCH INTO する。
16. FETCH によって戻された選択リストの値を処理する。
17. FETCH する行がなくなったらカーソルを CLOSE する。

注意 : 動的 SQL 文が問合せでない場合、あるいは個数がわかっている選択リスト項目またはプレースホルダを含む場合は、上記の一部のステップは必要ありません。

各手順の詳細

この項では、個々のステップを詳しく説明します。また、この章の終りには、方法 4 のコメント付きの完全なプログラム例が掲載されています。方法 4 では、次の一連の埋込み SQL 文を使用します。

```
EXEC SQL
    PREPARE <statement_name>
    FROM {:<host_string> | <string_literal>}
END-EXEC.
EXEC SQL
    DECLARE <cursor_name> CURSOR FOR <statement_name>
END-EXEC.
EXEC SQL
    DESCRIBE BIND VARIABLES FOR <statement_name>
    INTO <bind_descriptor_name>
END-EXEC.
EXEC SQL
    OPEN <cursor_name>
    [USING DESCRIPTOR <bind_descriptor_name>]
END-EXEC.
EXEC SQL
    DESCRIBE [SELECT LIST FOR] <statement_name>
    INTO <select_descriptor_name>
```

```
END-EXEC.
EXEC SQL
    FETCH <cursor_name> USING DESCRIPTOR <select_descriptor_name>
END-EXEC.
EXEC SQL
    CLOSE <cursor_name>
END-EXEC.
```

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 文を使えます。

```
EXEC SQL FETCH <cursor_name> INTO <host_variable_list> END-EXEC.
```

また、動的 SQL 文中のバインド変数のプレースホルダの数がわかっている場合は、DESCRIBE BIND VARIABLES を使わずに、次に示す方法 3 の OPEN 文を使用できます。

```
EXEC SQL OPEN <cursor_name> [USING <host_variable_list>] END-EXEC.
```

次にこれらの文によって、記述子を使ってホスト・プログラムで動的 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、図を使って説明します。図が複雑になるのを避けるため、記述子表の要素は 3 つまで、名前と値の最大長はそれぞれ 5 文字と 10 文字に制限しました。

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。このホスト変数（例では SELECT-STMT）は、文字列として宣言しなければなりません。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    ...
    01  SELECT-STMT  PIC X(120).
EXEC SQL END DECLARE SECTION END-EXEC.
```

SQLDA の宣言

例では、問合せの選択リスト項目またはプレースホルダの数が不明な場合があるため、選択記述子とバインド記述子を宣言する必要があります。SQLDA をハードコードするかわりに、次のように、INCLUDE を使って SQLDA をプログラムにコピーします。

```
EXEC SQL INCLUDE SELDSC END-EXEC.
EXEC SQL INCLUDE BNDDSC END-EXEC.
```

参考のため、INCLUDE される SELDSC の宣言を次に示します。

```
WORKING-STORAGE SECTION.
    ...
```

```
01  SELDSC.
        05  SQLDNUM              PIC S9(9) COMP.
        05  SQLDFND              PIC S9(9) COMP.
        05  SELDVAR              OCCURS 3 TIMES.
            10 SELDV             PIC S9(9) COMP.
            10 SELDFMT           PIC S9(9) COMP.
            10 SELDVLN           PIC S9(9) COMP.
            10 SELDFMTL          PIC S9(4) COMP.
            10 SELDVTYP          PIC S9(4) COMP.
            10 SELDI             PIC S9(9) COMP.
            10 SELDH-VNAME       PIC S9(9) COMP.
            10 SELDH-MAX-VNAMEL  PIC S9(4) COMP.
            10 SELDH-CUR-VNAMEL  PIC S9(4) COMP.
            10 SELDI-VNAME       PIC S9(9) COMP.
            10 SELDI-MAX-VNAMEL  PIC S9(4) COMP.
            10 SELDI-CUR-VNAMEL  PIC S9(4) COMP.
            10 SELDFCLP          PIC S9(9) COMP.
            10 SELDFCRCP         PIC S9(9) COMP.

01  XSELDI.
    05  SEL-DI        OCCURS 3 TIMES PIC S9(9) COMP.
01  XSELDIVNAME.
        05  SEL-DI-VNAME  OCCURS 3 TIMES PIC X(5).
01  XSELDV.
        05  SEL-DV        OCCURS 3 TIMES PIC X(10).
01  XSELDHVNAME.
        05  SEL-DH-VNAME  OCCURS 3 TIMES PIC X(5).
```

DESCRIBE への最大数の設定

次に、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設定します。

```
MOVE 3 TO SQLDNUM IN SELDSC.
MOVE 3 TO SQLDNUM IN BNDDSC.
```

記述子の初期化

初期化を必要とする記述子変数もあります。また、初期化にライブラリ・サブルーチン SQLADR が必要なものもあります。

例では、名前バッファの最大長が SELDH-MAX-VNAMEL 表、BNDDH-MAX-VNAMEL および BNDDI-MAX-VNAMEL 表に格納され、SQLADR を使って値バッファと名前バッファのアドレスが表 SELDV、SELDI、BNDDV、BNDDI、SELDH-VNAME、BNDDH-VNAME、BNDDI-VNAME に格納されています。

```
PROCEDURE DIVISION.
    ...
```

```
    PERFORM INIT-SELDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
    PERFORM INIT-BNDDSC
        VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
    ...
INIT-SELDSC.
    MOVE SPACES TO SEL-DV(J).
    MOVE SPACES TO SEL-DH-VNAME(J).
    MOVE 5 TO SELDH-MAX-VNAMEL(J).
    CALL "SQLADR" USING SEL-DV(J), SELDV(J).
    CALL "SQLADR" USING SEL-DH-VNAME(J), SELDH-VNAME(J).
    CALL "SQLADR" USING SEL-DI(J), SELDI(J).
    ...
INIT-BNDDSC.
    MOVE SPACES TO BND-DV(J).
    MOVE SPACES TO BND-DH-VNAME(J).
    MOVE SPACES TO BND-DI-VNAME(J).
    MOVE 5 TO BNDDH-MAX-VNAMEL(J).
    MOVE 5 TO BNDDI-MAX-VNAMEL(J).
    CALL "SQLADR" USING BND-DV(J), BNDDV(J).
    CALL "SQLADR" USING BND-DH-VNAME(J), BNDDH-VNAME(J).
    CALL "SQLADR" USING BND-DI(J), BNDDI(J).
    CALL "SQLADR" USING BND-DI-VNAME(J), BNDDI-VNAME(J).
    ...
```

図 11-3 と図 11-4 に、結果として得られる記述子を示します。

図 11-3　初期化した選択記述子

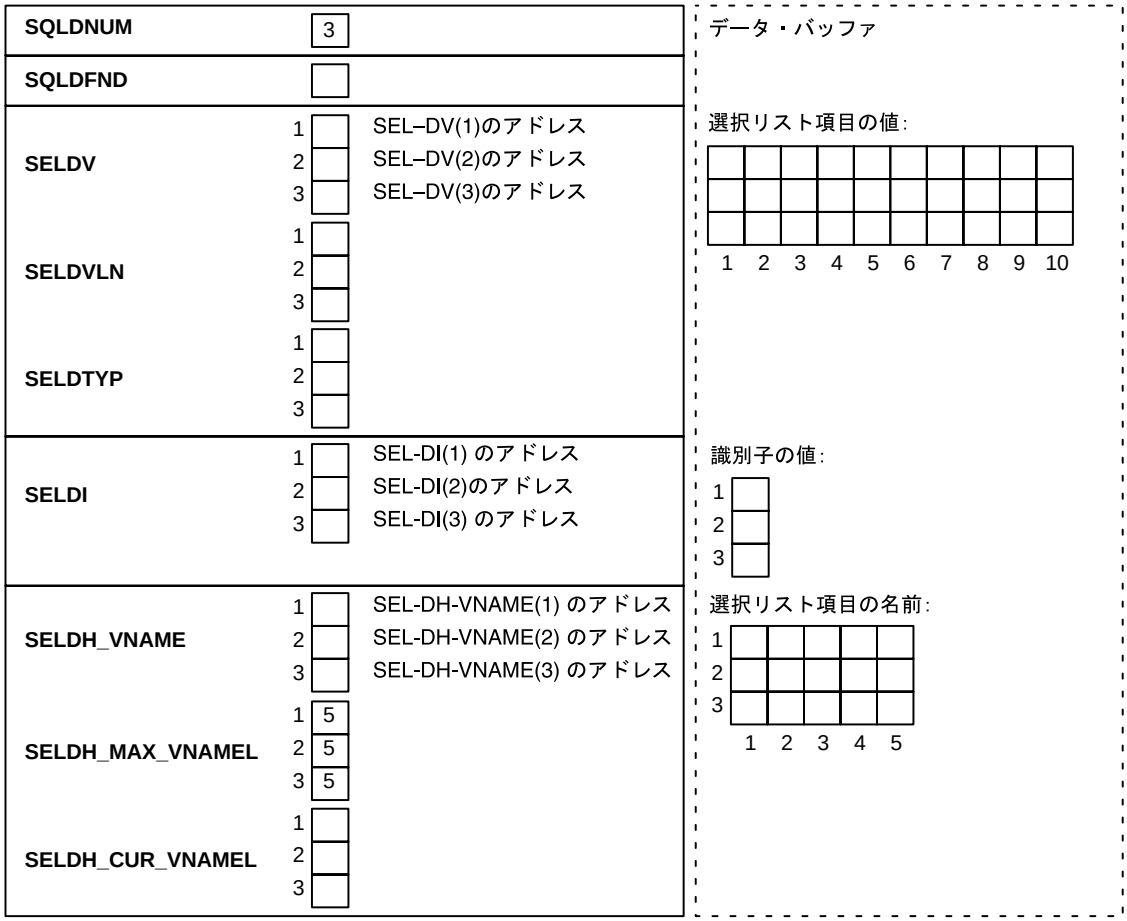
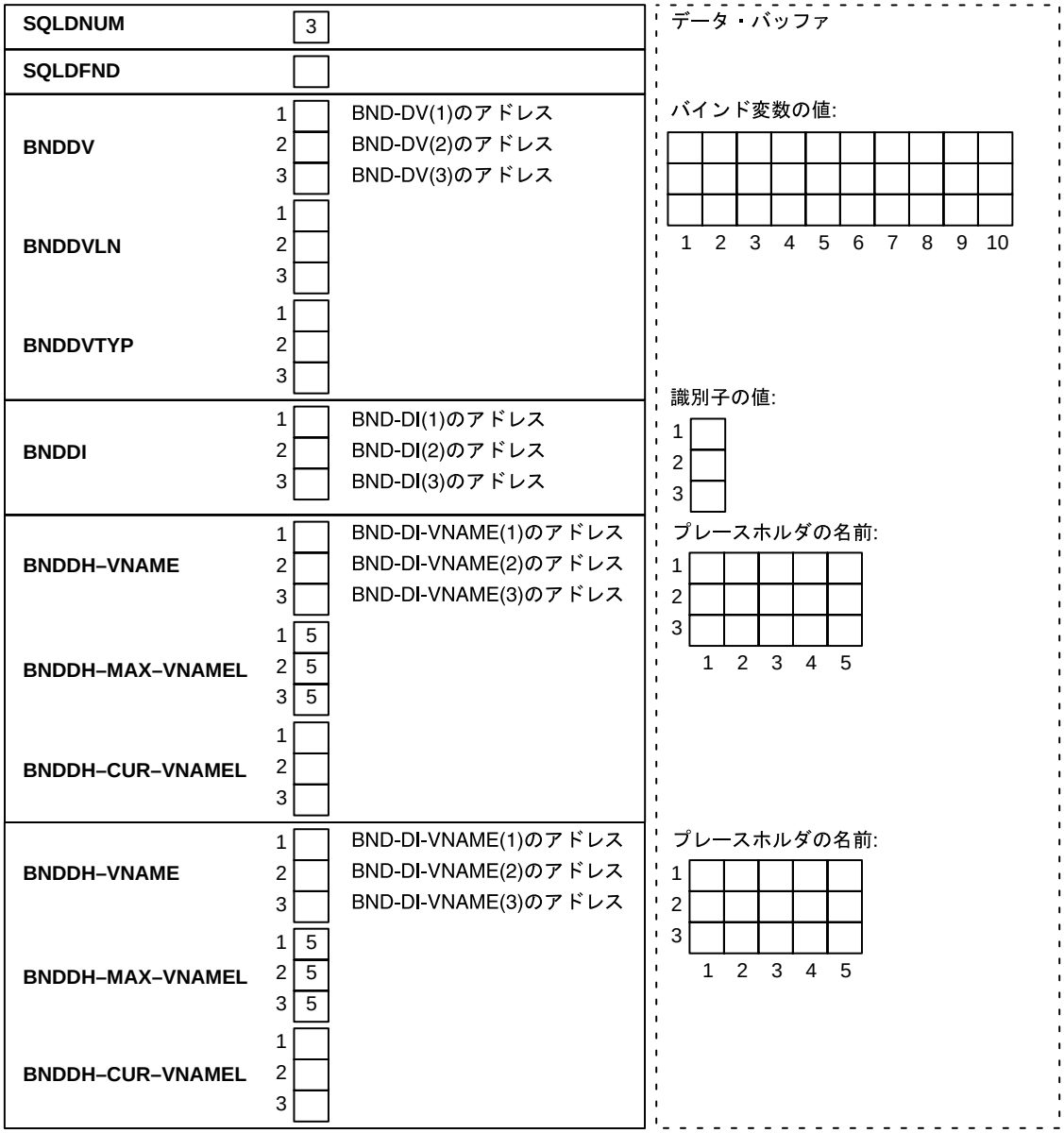


図 11-4　初期化したバインド記述子



ホスト文字列への問合せテキストの格納

次に、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。

```
DISPLAY "Enter a SELECT statement: " WITH NO ADVANCING.
ACCEPT SELECT-STMT.
```

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
```

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 でホスト文字列 SELECT-STMT を解析し、SQL-STMT という名前を付けます。

```
EXEC SQL PREPARE SQL-STMT FROM :SELECT-STMT END-EXEC.
```

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って、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います。

```
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 によって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、次に示すように、DECLARE CURSOR は EMP-CURSOR という名前のカーソルを定義し、それを SQL-STMT と対応付けます。

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR SQL-STMT END-EXEC.
```

注意 : 問合せ用だけでなく、すべての動的 SQL 文用にカーソルを宣言する必要があります。また、問合せ以外の場合も、カーソルの OPEN によって動的 SQL 文を実行します。

バインド変数の DESCRIBE

DESCRIBE BIND VARIABLES は、バインド変数の記述をバインド記述子に格納します。例では、DESCRIBE で BNDDSC を準備します。

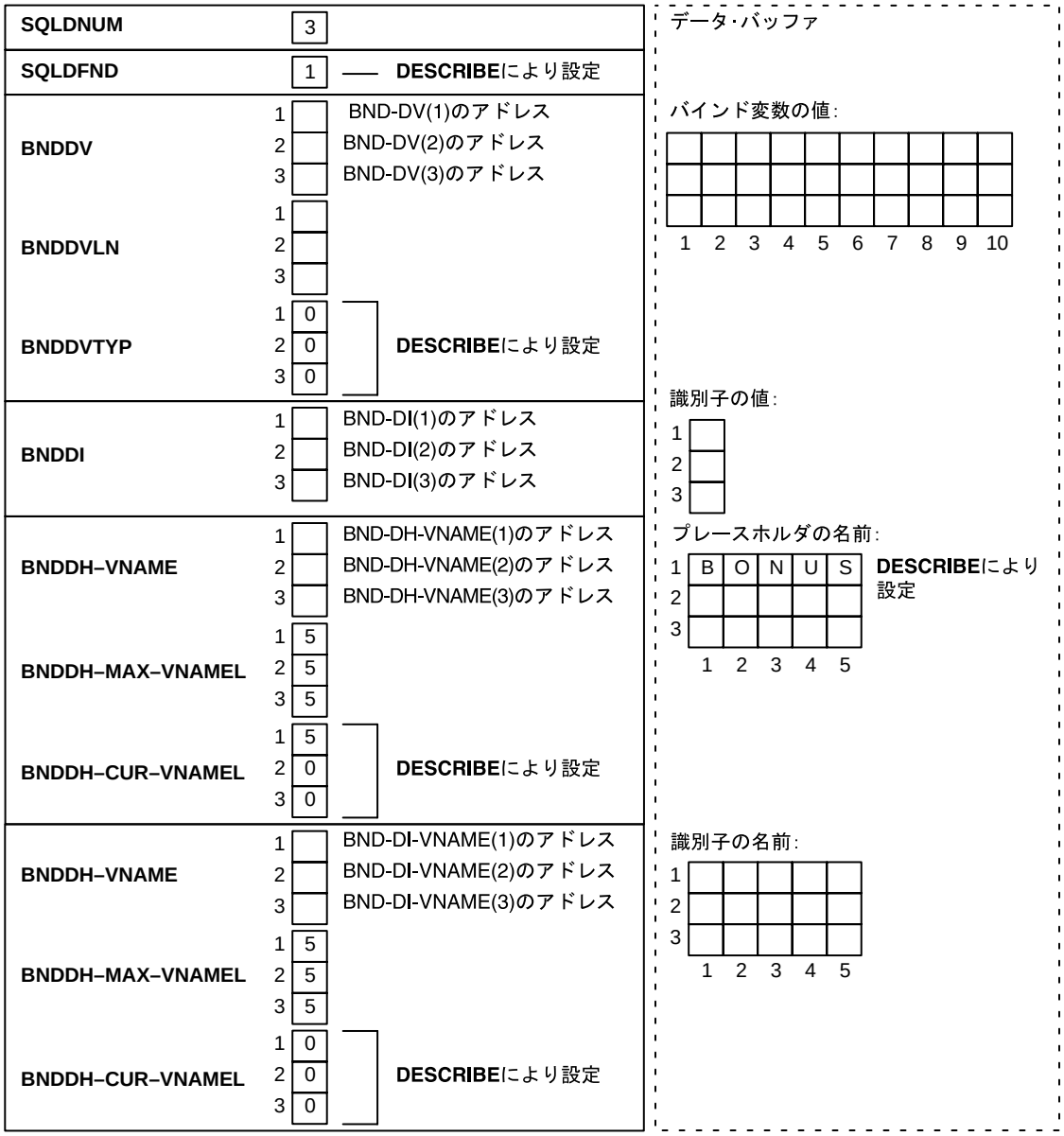
```
EXEC SQL
    DESCRIBE BIND VARIABLES FOR SQL-STMT
    INTO BNDDSC
END-EXEC.
```

BNDDSC の前にはコロンを付けないようにします。

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 11-5 に、DESCRIBE 実行後のバインド記述子を示します。DESCRIBE によって、処理対象の SQL 文で実際に検出されたプレースホルダの数に、SQLDFND が設定されたので注意してください。

図 11-5 DESCRIBE 実行後のバインド記述子



プレースホルダの数の再設定

次に、プレースホルダの最大数を、DESCRIBE で実際に検出された数に再設定する必要があります。

```
     IF SQLDFND IN BNDDSC < 0
         DISPLAY "Too many bind variables"
         GOTO ROLL-BACK
     ELSE
         MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC
     END-IF.
```

バインド変数の値の取得

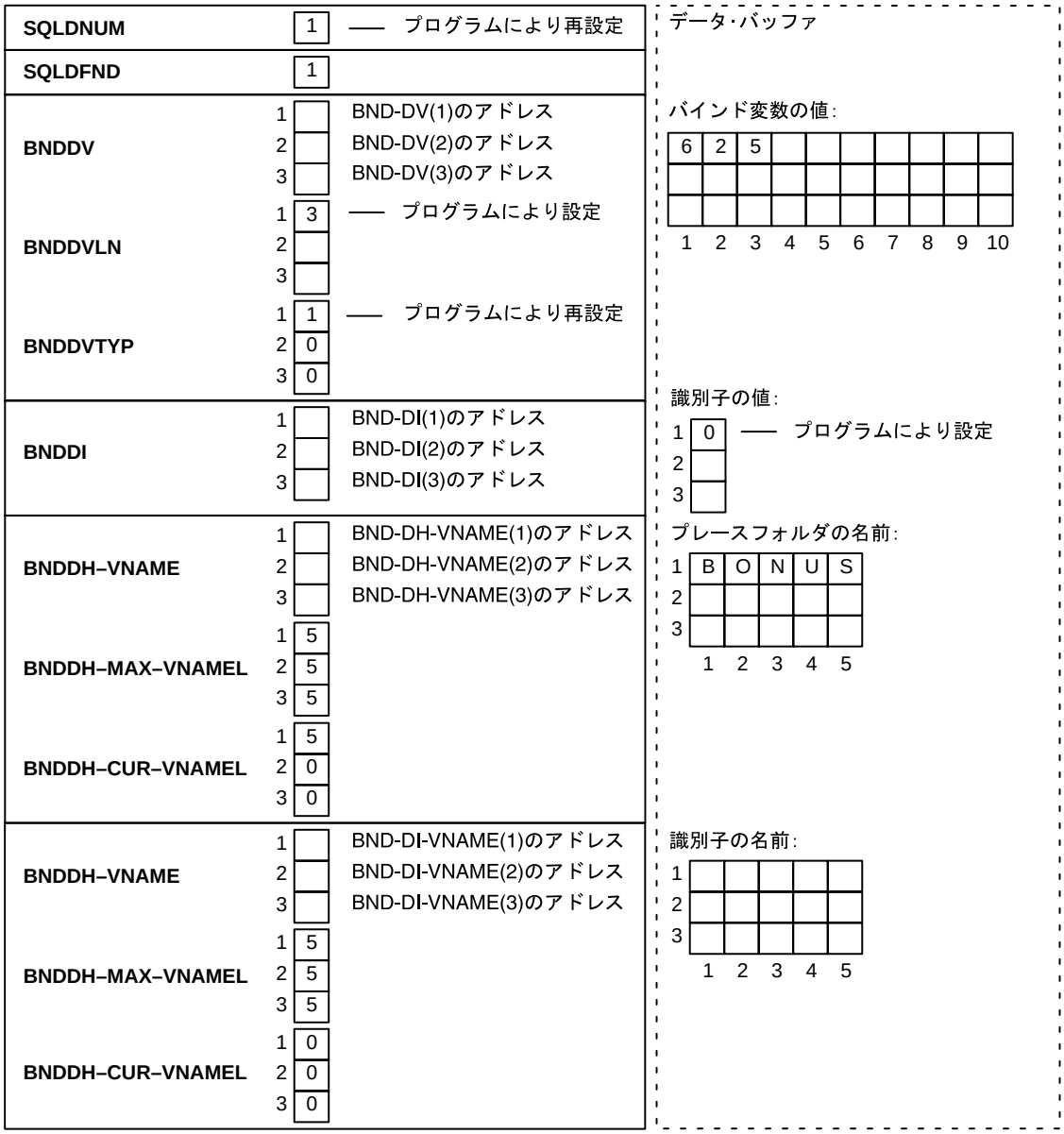
プログラムは SQL 文中のバインド変数の値を取得しなければなりません。値はどのように取得してもかまいません。たとえば値をハードコードしたり、ファイルから読込みしたり対話形式で入力することもできます。

例では、問合せの WHERE 句のプレースホルダ BONUS に置き換わるバインド変数に値を代入する必要があります。次のように、ユーザーに値の入力を求め、入力された値を処理します。

```
 PROCEDURE DIVISION.
     ...
     PERFORM GET-INPUT-VAR
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN BNDDSC.
     ...
 GET-INPUT-VAR.
     ...
*    Replace the 0 DESCRIBEd into the datatype table
*    with a 1 to avoid an "invalid datatype" Oracle error.
     MOVE 1 TO BNDDVTYP(J).
*    Get value of bind variable.
     DISPLAY "Enter value of ", BND-DH-VNAME(J).
     ACCEPT INPUT-STRING.
     UNSTRING INPUT-STRING DELIMITED BY "  "
         INTO BND-DV(J) COUNT IN BNDDVLN(J).
```

ここでは、ユーザーが BONUS の値として 625 を入力したとみなして、次の表に結果として得られるバインド記述子を示します。

図 11-6 値を割り当てた後のバインド記述子



カーソルの OPEN

動的問合せの 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、静的問合せの OPEN 文と同じです。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されたバッファに格納された値を使って、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 で EMP-CURSOR を BNDDSC に対応付けます。

```
EXEC SQL
    OPEN EMP-CUR USING DESCRIPTOR BNDDSC
END-EXEC.
```

BNDDSC の前にはコロンを付けないようにします。

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。

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。

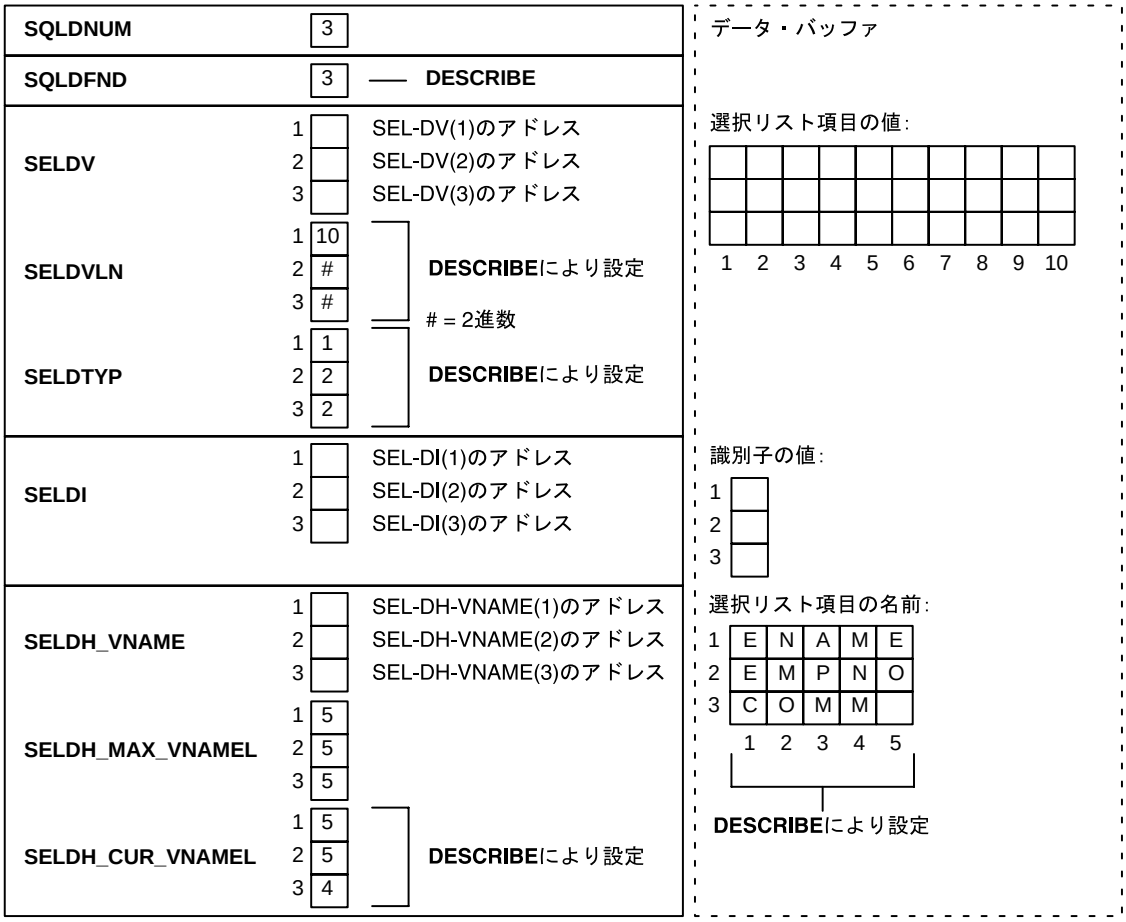
DESCRIBE SELECT LIST は、選択リスト項目の記述を選択記述子に格納します。例では、DESCRIBE で SELDSC を準備します。

```
EXEC SQL
    DESCRIBE SELECT LIST FOR SQL-STMT INTO SELDSC
END-EXEC.
```

DESCRIBE は、データ・ディクショナリにアクセスして、各選択リスト値の長さとデータ型を設定します。

図 11-7 に、DESCRIBE 実行後のバインド記述子を示します。DESCRIBE によって、問合せの選択リストで実際に検出された項目の数に、SQLDFND が設定されているので注意してください。SQL 文が問合せでない場合は、SQLDFND は 0（ゼロ）に設定されます。また、NUMBER 型の長さはまだ使えません。NUMBER として定義された列については、ライブラリ・サブルーチン SQLPRC を使って精度と位取りを抽出する必要があります。詳細は 11-17 ページの「データ型の強制変換」を参照してください。

図 11-7　DESCRIBE 実行後の選択記述子



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 によって実際に検出された数に再設定する必要があります。

```
     MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC.
```

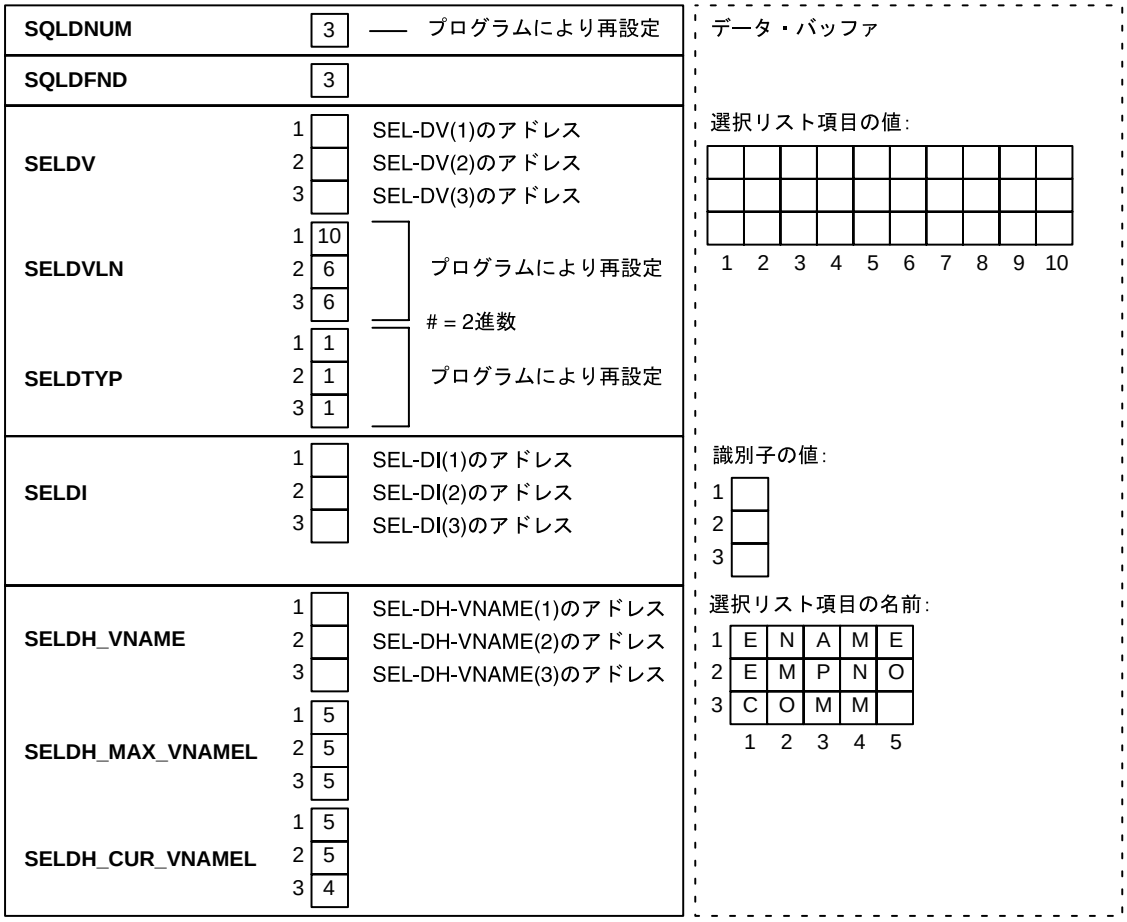
各選択リスト項目の長さとデータ型の再設定

例では、選択リストの値をフェッチする前に、長さの表とデータ型の表の要素の一部を表示するために再設定します。

```
 PROCEDURE DIVISION.
     ...
     PERFORM COERCE-COLUMN-TYPE
         VARYING J FROM 1 BY 1 UNTIL J > SQLDNUM IN SELDSC.
     ...
 COERCE-COLUMN-TYPE.
*    Clear NULL bit.
     CALL "SQLNUL" USING SELDVTYP(J), SELDVTYP(J), NULL-STATUS.

*    If datatype is DATE, lengthen to 9 characters.
     IF SELDVTYP(J) = 12
         MOVE 9 TO SELDVLN(J).

*    If datatype is NUMBER, extract precision and scale.
     MOVE 0 TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2 AND PRECISION = 0
         MOVE 10 TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2 AND PRECISION > 0
         ADD 2 TO PRECISION
         MOVE PRECISION TO DISPLAY-LENGTH.
     IF SELDVTYP(J) = 2
         IF DISPLAY-LENGTH > MAX-LENGTH
             DISPLAY "Column value too large for data buffer."
             GO TO END-PROGRAM
         ELSE
             MOVE DISPLAY-LENGTH TO SELDVLN(J).

*    Coerce datatypes to VARCHAR2.
     MOVE 1 TO SELDVTYP(J).
```

図 11-8 に、結果として得られる選択記述子を示します。NUMBER の長さが使用でき、すべてのデータ型が VARCHAR2 になっていることに注意してください。符号と小数点を使用できるよう、DESCRIBE した長さ 4 および 7 をそれぞれ 2 だけ増加させたため、SESLDVLN(2) と SELDVLN(3) の長さが 6 および 9 になっています。

図 11-8　FETCH 実行前の選択記述子



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、SQLCA の SQLCODE または SQLCODE 変数、SQLSTATE 変数を " データがありません " というエラー・コードに設定します。次の例では、FETCH は列 ENAME および EMPNO、COMM の値を SELDSC に戻します。

```
EXEC SQL
    FETCH EMP-CURSOR USING DESCRIPTOR SELDSC
END-EXEC.
```

図 11-9 に、FETCH 実行後の選択記述子を示します。Oracle8*i* によって、SELDV と SELDI の要素でアドレス指定されたデータ・バッファに選択リストと標識の値が格納されているので注意してください。

データ型 1 の出力バッファの場合、Oracle8*i* は SELDVLN に格納されている長さを使用して、CHAR データと VARCHAR2 データは左揃えにし、NUMBER データは右揃えにします。

値 "MARTIN" は、EMP 表の VARCHAR2(10) 列から取り出されました。Oracle8*i* は、SELDVLN(1) に格納された長さを使って、この値を 10 バイトのフィールドに左揃えにして入れ、バッファの残りの部分を埋めます。

値 7654 は NUMBER(4) から取り出され、"7654" に強制変換されています。しかし、SELDVLN(2) の長さが 2 だけ増えて符号と小数点を使用できるようになっているため、Oracle8*i* では 6 バイトのフィールド値で値が右揃えになります。

値 482.50 は NUMBER(7.2) から取り出され、"482.50" に強制変換されています。ここでも、SELDVLN(3) の長さが 2 だけ増えているため、Oracle8*i* では 9 バイトのフィールドで値が右揃えになります。

選択リストの値の取得と処理

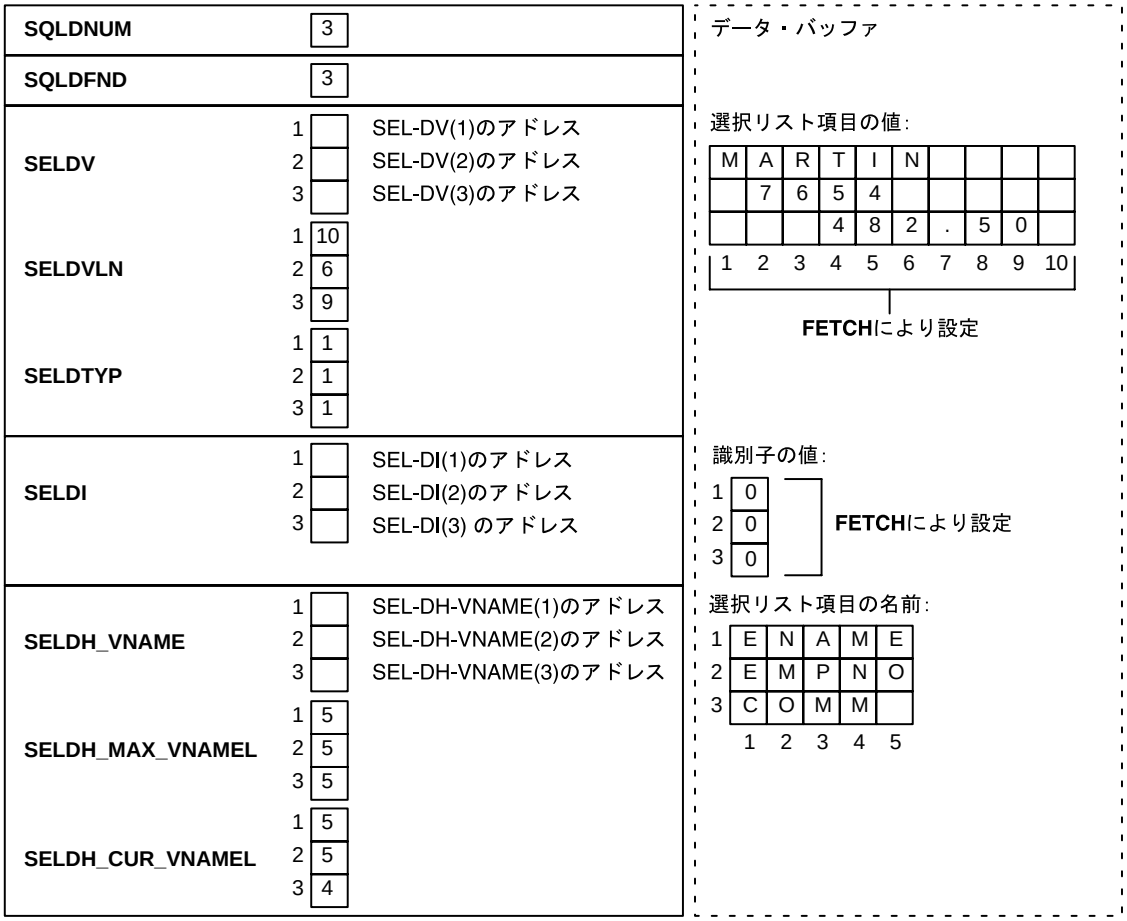
FETCH の後、プログラムは FETCH で戻された選択リストの値を処理できます。例では、列 ENAME および EMPNO、COMM の値が処理されます。

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 によって EMP-CURSOR が使用禁止になります。

```
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

図 11-9　FETCH 実行後の選択記述子



方法 4 でのホスト配列の使用

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 句を使ってホスト配列のサイズを Oracle8*i* に認識させる必要があります。FOR 句の詳細は、第 7 章の「ホスト表」を参照してください。

J 番目の選択リスト項目またはバインド変数の記述子エントリを設定します。この場合、SELDVLN(J) または BNDDVLN(J) により、1 つのデータ・バッファではなく、データ・バッファの配列がアドレス指定されます。次に、EXECUTE または FETCH 文で FOR 句を使用して、処理する配列要素の数を Oracle8*i* に認識させます。

Oracle8*i* がホスト配列のサイズを認識する方法は他にないため、このステップは必須です。

次の例では、2 つの入力ホスト配列を使って、EMPNO および DEPTNO の 8 組の値を EMP 表に挿入します。方法 4 では問合せ以外の SQL 文に対して EXECUTE を使用できるので注意してください。

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. DYN4INS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  BNDDSC.
     02  SQLDNUM              PIC S9(9) COMP VALUE 2.
     02  SQLDFND              PIC S9(9) COMP.
     02  BNDDVAR              OCCURS 2 TIMES.
         03 BNDDV             PIC S9(9) COMP.
         03 BNDDFMT           PIC S9(9) COMP.
         03 BNDDVLN           PIC S9(9) COMP.
         03 BNDDFMTL          PIC S9(4) COMP.
         03 BNDDVTYP          PIC S9(4) COMP.
         03 BNDDI             PIC S9(9) COMP.
         03 BNDDH-VNAME       PIC S9(9) COMP.
         03 BNDDH-MAX-VNAMEL  PIC S9(4) COMP.
         03 BNDDH-CUR-VNAMEL  PIC S9(4) COMP.
         03 BNDDI-VNAME       PIC S9(9) COMP.
         03 BNDDI-MAX-VNAMEL  PIC S9(4) COMP.
         03 BNDDI-CUR-VNAMEL  PIC S9(4) COMP.
         03 BNDDFCLP          PIC S9(9) COMP.
         03 BNDDFCRCP         PIC S9(9) COMP.
 01  XBNDDI.
     03  BND-DI               OCCURS 2 TIMES PIC S9(4) COMP.
 01  XBNDDIVNAME.
     03  BND-DI-VNAME         OCCURS 2 TIMES PIC X(80).
 01  XBNDDV.
*    Since you know what the SQL statement will be, you can set
*    up a two-dimensional table with a maximum of 2 columns and
*    8 rows.  Each element can be up to 10 characters long.  (You
```

```
*    can alter these values according to your needs.)
     03  BND-COLUMN           OCCURS 2 TIMES.
         05  BND-ELEMENT      OCCURS 8 TIMES PIC X(10).
 01  XBNDDHVNAME.
     03  BND-DH-VNAME         OCCURS 2 TIMES PIC X(80).
 01  COLUMN-INDEX             PIC 999.
 01  ROW-INDEX                PIC 999.
 01  DUMMY-INTEGER            PIC 9999.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
         01  USERNAME         PIC X(20).
         01  PASSWD           PIC X(20).
         01  DYN-STATEMENT    PIC X(80).
         01  NUMBER-OF-ROWS   PIC S9(4) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.

     EXEC SQL INCLUDE SQLCA END-EXEC.

 PROCEDURE DIVISION.
 START-MAIN.

     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

     MOVE "SCOTT" TO USERNAME.
     MOVE "TIGER" TO PASSWD.
     EXEC SQL
         CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY "Connected to Oracle".

*    Initialize bind and select descriptors.
     PERFORM INIT-BNDDSC THRU INIT-BNDDSC-EXIT
         VARYING COLUMN-INDEX FROM 1 BY 1
         UNTIL COLUMN-INDEX > 2.

*    Set up the SQL statement.
     MOVE SPACES TO DYN-STATEMENT.
     MOVE "INSERT INTO EMP(EMPNO, DEPTNO) VALUES(:EMPNO,:DEPTNO)"
         TO DYN-STATEMENT.
     DISPLAY DYN-STATEMENT.

*    Prepare the SQL statement.
     EXEC SQL
         PREPARE S1 FROM :DYN-STATEMENT
     END-EXEC.

*    Describe the bind variables.
     EXEC SQL
```

```
            DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
        END-EXEC.

        PERFORM Z-BIND-TYPE THRU Z-BIND-TYPE-EXIT
            VARYING COLUMN-INDEX FROM 1 BY 1
            UNTIL COLUMN-INDEX > 2.

        IF SQLDFND IN BNDDSC < 0
            DISPLAY "TOO MANY BIND VARIABLES."
            GO TO SQL-ERROR
        ELSE
            DISPLAY "BIND VARS = " WITH NO ADVANCING
            MOVE SQLDFND IN BNDDSC TO DUMMY-INTEGER
            DISPLAY DUMMY-INTEGER
            MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC.

            MOVE 8 TO NUMBER-OF-ROWS.
            PERFORM GET-ALL-VALUES THRU GET-ALL-VALUES-EXIT
                VARYING ROW-INDEX FROM 1 BY 1
                UNTIL ROW-INDEX > NUMBER-OF-ROWS.

*       Execute the SQL statement.
        EXEC SQL FOR :NUMBER-OF-ROWS
            EXECUTE S1 USING DESCRIPTOR BNDDSC
        END-EXEC.

        DISPLAY "INSERTED " WITH NO ADVANCING.
        MOVE SQLERRD(3) TO DUMMY-INTEGER.
        DISPLAY DUMMY-INTEGER WITH NO ADVANCING.
        DISPLAY " ROWS.".
        GO TO END-SQL.

    SQL-ERROR.
*       Display any SQL error message and code.
        DISPLAY SQLERRMC.
        EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
        STOP RUN.

    END-SQL.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL COMMIT WORK RELEASE END-EXEC.
        STOP RUN.

    INIT-BNDDSC.
*       Start of COBOL PERFORM procedures, initialize the bind
*       descriptor.
        MOVE 80 TO BNDDH-MAX-VNAMEL(COLUMN-INDEX).
```

```
     CALL "SQLADR" USING
         BND-DH-VNAME(COLUMN-INDEX)
         BNDDH-VNAME(COLUMN-INDEX).
     MOVE 80 TO BNDDI-MAX-VNAMEL(COLUMN-INDEX).
     CALL "SQLADR" USING
         BND-DI-VNAME(COLUMN-INDEX)
         BNDDI-VNAME (COLUMN-INDEX).
     MOVE 10 TO BNDDVLN(COLUMN-INDEX).
     CALL "SQLADR" USING
         BND-ELEMENT(COLUMN-INDEX,1)
         BNDDV(COLUMN-INDEX).
     MOVE ZERO TO BNDDI(COLUMN-INDEX).
     CALL "SQLADR" USING
         BND-DI(COLUMN-INDEX)
         BNDDI(COLUMN-INDEX).
     MOVE ZERO TO BNDDFMT(COLUMN-INDEX).
     MOVE ZERO TO BNDDFMTL(COLUMN-INDEX).
     MOVE ZERO TO BNDDFCLP(COLUMN-INDEX).
     MOVE ZERO TO BNDDFCRCP(COLUMN-INDEX).
 INIT-BNDDSC-EXIT.
     EXIT.

 Z-BIND-TYPE.
*    Replace the 0s DESCRIBEd into the datatype table with 1s to
*    avoid an "invalid datatype" Oracle error.
     MOVE 1 TO BNDDVTYP(COLUMN-INDEX).

 Z-BIND-TYPE-EXIT.
     EXIT.

 GET-ALL-VALUES.
*    Get the bind variables for each row.
     DISPLAY "ENTER VALUES FOR ROW NUMBER ",ROW-INDEX.
     PERFORM GET-BIND-VARS
         VARYING COLUMN-INDEX FROM 1 BY 1
         UNTIL COLUMN-INDEX > SQLDFND IN BNDDSC.
 GET-ALL-VALUES-EXIT.
     EXIT.

 GET-BIND-VARS.
*    Get the value of each bind variable.
     DISPLAY "    ENTER VALUE FOR ",BND-DH-VNAME(COLUMN-INDEX)
         WITH NO ADVANCING.
     ACCEPT BND-ELEMENT(COLUMN-INDEX,ROW-INDEX).
 GET-BIND-VARS-EXIT.
     EXIT.
```

サンプル・プログラム 10: 動的 SQL 方法 4

このプログラムでは、動的 SQL 方法 4 を使うために必要な基本手順を示します。ログインすると、ユーザーは SQL 文の入力を求められます。次にこのプログラムでは、文の PREPARE、カーソルの DECLARE に続いて、DESCRIBE BIND を使ってバインド変数をチェックします。最後にカーソルを OPEN して、選択リスト変数を 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。

```
****************************************************************
* Sample Program 10: Dynamic SQL Method 4                      *
*                                                              *
* This program shows the basic steps required to use dynamic   *
* SQL Method 4.  After logging on to ORACLE, the program       *
* prompts the user for a SQL statement, PREPAREs the           *
* statement, DECLAREs a cursor, checks for any bind variables *
* using DESCRIBE BIND, OPENs the cursor, and DESCRIBEs any     *
* select-list variables.  If the input SQL statement is a      *
* query, the program FETCHes each row of data, then CLOSEs     *
* the cursor.                                                  *
****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DYNSQL4.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 01  BNDDSC.

     02  SQLDNUM                PIC S9(9) COMP VALUE 20.
     02  SQLDFND                PIC S9(9) COMP.
     02  BNDDVAR                OCCURS 20 TIMES.
         03  BNDDV              PIC S9(9) COMP.
         03  BNDDFMT            PIC S9(9) COMP.
         03  BNDDVLN            PIC S9(9) COMP.
         03  BNDDFMTL           PIC S9(4) COMP.
         03  BNDDVTYP           PIC S9(4) COMP.
         03  BNDDI              PIC S9(9) COMP.
         03  BNDDH-VNAME        PIC S9(9) COMP.
         03  BNDDH-MAX-VNAMEL   PIC S9(4) COMP.
         03  BNDDH-CUR-VNAMEL   PIC S9(4) COMP.
         03  BNDDI-VNAME        PIC S9(9) COMP.
         03  BNDDI-MAX-VNAMEL   PIC S9(4) COMP.
         03  BNDDI-CUR-VNAMEL   PIC S9(4) COMP.
         03  BNDDFCLP           PIC S9(9) COMP.
         03  BNDDFCRCP          PIC S9(9) COMP.
```

```
01  XBNDDI.

    03  BND-DI                OCCURS 20 TIMES PIC S9(4) COMP.

01  XBNDDIVNAME.
    03  BND-DI-VNAME          OCCURS 20 TIMES PIC X(80).
01  XBNDDV.
    03  BND-DV                OCCURS 20 TIMES PIC X(80).
01  XBNDDHVNAME.
    03  BND-DH-VNAME          OCCURS 20 TIMES PIC X(80).

01  SELDSC.

    02  SQLDNUM               PIC S9(9) COMP VALUE 20.
    02  SQLDFND               PIC S9(9) COMP.
    02  SELDVAR               OCCURS 20 TIMES.
        03  SELDV             PIC S9(9) COMP.
        03  SELDFMT           PIC S9(9) COMP.
        03  SELDVLN           PIC S9(9) COMP.
        03  SELDFMTL          PIC S9(4) COMP.
        03  SELDVTYP          PIC S9(4) COMP.
        03  SELDI             PIC S9(9) COMP.
        03  SELDH-VNAME       PIC S9(9) COMP.
        03  SELDH-MAX-VNAMEL  PIC S9(4) COMP.
        03  SELDH-CUR-VNAMEL  PIC S9(4) COMP.
        03  SELDI-VNAME       PIC S9(9) COMP.
        03  SELDI-MAX-VNAMEL  PIC S9(4) COMP.
        03  SELDI-CUR-VNAMEL  PIC S9(4) COMP.
        03  SELDFCLP          PIC S9(9) COMP.
        03  SELDFCRCP         PIC S9(9) COMP.

01  XSELDI.

    03  SEL-DI                OCCURS 20 TIMES PIC S9(4) COMP.

01  XSELDIVNAME.
    03  SEL-DI-VNAME          OCCURS 20 TIMES PIC X(80).
01  XSELDV.
    03  SEL-DV                OCCURS 20 TIMES PIC X(80).
01  XSELDHVNAME.
    03  SEL-DH-VNAME          OCCURS 20 TIMES PIC X(80).

01  TABLE-INDEX               PIC 9(3).
01  VAR-COUNT                 PIC 9(2).
01  ROW-COUNT                 PIC 9(4).
01  NO-MORE-DATA              PIC X(1) VALUE "N".
```

```
 01  NULLS-ALLOWED            PIC S9(9) COMP.

 01  PRECISION                PIC S9(9) COMP.
 01  SCALE                    PIC S9(9) COMP.

 01  DISPLAY-LENGTH           PIC S9(9) COMP.
 01  MAX-LENGTH               PIC S9(9) COMP VALUE 80.
 01  COLUMN-NAME              PIC X(30).
 01  NULL-VAL                 PIC X(80) VALUE SPACES.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  USERNAME       PIC X(20).
 01  PASSWD         PIC X(20).
 01  DYN-STATEMENT  PIC X(80).
     EXEC SQL END DECLARE SECTION   END-EXEC.
     EXEC SQL INCLUDE SQLCA         END-EXEC.

 PROCEDURE DIVISION.
 START-MAIN.

     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.

     DISPLAY "USERNAME: " WITH NO ADVANCING.

     ACCEPT USERNAME.

     DISPLAY "PASSWORD: " WITH NO ADVANCING.

     ACCEPT PASSWD.

     EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWD END-EXEC.
     DISPLAY "CONNECTED TO ORACLE AS USER: ", USERNAME.

*    INITIALIZE THE BIND AND SELECT DESCRIPTORS.

     PERFORM INIT-BNDDSC
         VARYING TABLE-INDEX FROM 1 BY 1
         UNTIL TABLE-INDEX > 20.

     PERFORM INIT-SELDSC
         VARYING TABLE-INDEX FROM 1 BY 1
         UNTIL TABLE-INDEX > 20.

*    GET A SQL STATEMENT FROM THE OPERATOR.

     DISPLAY "ENTER SQL STATEMENT WITHOUT TERMINATOR:".
     DISPLAY ">" WITH NO ADVANCING.
```

```
     ACCEPT DYN-STATEMENT.

     DISPLAY " ".

*    PREPARE THE SQL STATEMENT AND DECLARE A CURSOR.

     EXEC SQL  PREPARE S1 FROM :DYN-STATEMENT  END-EXEC.
     EXEC SQL  DECLARE C1 CURSOR FOR S1        END-EXEC.

*    DESCRIBE ANY BIND VARIABLES.

     EXEC SQL  DESCRIBE BIND VARIABLES FOR S1 INTO BNDDSC
     END-EXEC.

     IF SQLDFND IN BNDDSC < 0
         DISPLAY "TOO MANY BIND VARIABLES."
         GO TO END-SQL
     ELSE
         DISPLAY "NUMBER OF BIND VARIABLES: " WITH NO ADVANCING
         MOVE SQLDFND IN BNDDSC TO VAR-COUNT
         DISPLAY VAR-COUNT
         MOVE SQLDFND IN BNDDSC TO SQLDNUM IN BNDDSC
     END-IF.

*    REPLACE THE 0S DESCRIBED INTO THE DATATYPE FIELDS OF THE
*    BIND DESCRIPTOR WITH 1S TO AVOID AN "INVALID DATATYPE"
*    ORACLE ERROR

     MOVE 1 TO TABLE-INDEX.
 FIX-BIND-TYPE.
         MOVE 1 TO BNDDVTYP(TABLE-INDEX)
         ADD 1 TO TABLE-INDEX
         IF TABLE-INDEX <= 20
             GO TO FIX-BIND-TYPE.

*    LET THE USER FILL IN THE BIND VARIABLES.

     IF SQLDFND IN BNDDSC = 0
         GO TO DESCRIBE-ITEMS.
     MOVE 1 TO TABLE-INDEX.
 GET-BIND-VAR.
         DISPLAY "ENTER VALUE FOR ", BND-DH-VNAME(TABLE-INDEX).

         ACCEPT BND-DV(TABLE-INDEX).

         ADD 1 TO TABLE-INDEX
         IF TABLE-INDEX <= SQLDFND IN BNDDSC
```

```
            GO TO GET-BIND-VAR.

*    OPEN THE CURSOR AND DESCRIBE THE SELECT-LIST ITEMS.

 DESCRIBE-ITEMS.

     EXEC SQL  OPEN C1 USING DESCRIPTOR BNDDSC          END-EXEC.
     EXEC SQL  DESCRIBE SELECT LIST FOR S1 INTO SELDSC  END-EXEC.

     IF SQLDFND IN SELDSC < 0
         DISPLAY "TOO MANY SELECT-LIST ITEMS."
         GO TO END-SQL
     ELSE
         DISPLAY "NUMBER OF SELECT-LIST ITEMS: "
             WITH NO ADVANCING
         MOVE SQLDFND IN SELDSC TO VAR-COUNT
         DISPLAY VAR-COUNT
         DISPLAY " "
         MOVE SQLDFND IN SELDSC TO SQLDNUM IN SELDSC
     END-IF.

*    COERCE THE DATATYPE OF ALL SELECT-LIST ITEMS TO VARCHAR2.

     IF SQLDNUM IN SELDSC > 0
         PERFORM COERCE-COLUMN-TYPE
             VARYING TABLE-INDEX FROM 1 BY 1
             UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
         DISPLAY " ".

*    FETCH EACH ROW AND PRINT EACH SELECT-LIST VALUE.

     IF SQLDNUM IN SELDSC > 0
         PERFORM FETCH-ROWS UNTIL NO-MORE-DATA = "Y".

     DISPLAY " "
     DISPLAY "NUMBER OF ROWS PROCESSED: " WITH NO ADVANCING.
     MOVE SQLERRD(3) TO ROW-COUNT.
     DISPLAY ROW-COUNT.

*    CLEAN UP AND TERMINATE.

     EXEC SQL  CLOSE C1             END-EXEC.
     EXEC SQL  COMMIT WORK RELEASE  END-EXEC.
     DISPLAY " ".
     DISPLAY "HAVE A GOOD DAY!".
     DISPLAY " ".
     STOP RUN.
```

```
*    DISPLAY ORACLE ERROR MESSAGE AND CODE.

 SQL-ERROR.
     DISPLAY " ".
     DISPLAY SQLERRMC.
 END-SQL.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
     STOP RUN.

*    PERFORMED SUBROUTINES BEGIN HERE:

*    INIT-BNDDSC: INITIALIZE THE BIND DESCRIPTOR.

 INIT-BNDDSC.

     MOVE SPACES TO BND-DH-VNAME(TABLE-INDEX).
     MOVE 80 TO BNDDH-MAX-VNAMEL(TABLE-INDEX).
     CALL "SQLADR" USING
         BND-DH-VNAME(TABLE-INDEX)
         BNDDH-VNAME(TABLE-INDEX).

     MOVE SPACES TO BND-DI-VNAME(TABLE-INDEX).
     MOVE 80 TO BNDDI-MAX-VNAMEL(TABLE-INDEX).
     CALL "SQLADR" USING
         BND-DI-VNAME(TABLE-INDEX)
         BNDDI-VNAME (TABLE-INDEX).

     MOVE SPACES TO BND-DV(TABLE-INDEX).
     MOVE 80 TO BNDDVLN(TABLE-INDEX).
     CALL "SQLADR" USING
         BND-DV(TABLE-INDEX)
         BNDDV(TABLE-INDEX).
     MOVE ZERO TO BND-DI(TABLE-INDEX).
     CALL "SQLADR" USING
         BND-DI(TABLE-INDEX)
         BNDDI(TABLE-INDEX).

     MOVE ZERO TO BNDDFMT(TABLE-INDEX).
     MOVE ZERO TO BNDDFMTL(TABLE-INDEX).
     MOVE ZERO TO BNDDFCLP(TABLE-INDEX).
     MOVE ZERO TO BNDDFCRCP(TABLE-INDEX).

*    INIT-SELDSC: INITIALIZE THE SELECT DESCRIPTOR.

 INIT-SELDSC.
```

```
     MOVE SPACES TO SEL-DH-VNAME(TABLE-INDEX).
     MOVE 80 TO SELDH-MAX-VNAMEL(TABLE-INDEX).
     CALL "SQLADR" USING
         SEL-DH-VNAME(TABLE-INDEX)
         SELDH-VNAME(TABLE-INDEX).

     MOVE SPACES TO SEL-DI-VNAME(TABLE-INDEX).
     MOVE 80 TO SELDI-MAX-VNAMEL(TABLE-INDEX).
     CALL "SQLADR" USING
         SEL-DI-VNAME(TABLE-INDEX)
         SELDI-VNAME  (TABLE-INDEX).

     MOVE SPACES TO SEL-DV(TABLE-INDEX).
     MOVE 80 TO SELDVLN(TABLE-INDEX).
     CALL "SQLADR" USING
         SEL-DV(TABLE-INDEX)
         SELDV(TABLE-INDEX).

     MOVE ZERO TO SEL-DI(TABLE-INDEX).
     CALL "SQLADR" USING
         SEL-DI(TABLE-INDEX)
         SELDI(TABLE-INDEX).

     MOVE ZERO TO SELDFMT(TABLE-INDEX).
     MOVE ZERO TO SELDFMTL(TABLE-INDEX).
     MOVE ZERO TO SELDFCLP(TABLE-INDEX).
     MOVE ZERO TO SELDFCRCP(TABLE-INDEX).

*    COERCE SELECT-LIST DATATYPES TO VARCHAR2.

 COERCE-COLUMN-TYPE.
     CALL "SQLNUL" USING
         SELDVTYP(TABLE-INDEX)
         SELDVTYP(TABLE-INDEX)
         NULLS-ALLOWED.

*    IF DATATYPE IS DATE, LENGTHEN TO 9 CHARACTERS.
     IF SELDVTYP(TABLE-INDEX) = 12
         MOVE 9 TO SELDVLN(TABLE-INDEX).

*    IF DATATYPE IS NUMBER, SET LENGTH TO PRECISION.
     IF SELDVTYP(TABLE-INDEX) = 2
         CALL "SQLPRC" USING
             SELDVLN(TABLE-INDEX)
             PRECISION
             SCALE.
```

```
     MOVE 0 TO DISPLAY-LENGTH.
     IF SELDVTYP(TABLE-INDEX) = 2 AND PRECISION = 0
         MOVE 40 TO DISPLAY-LENGTH.
     IF SELDVTYP(TABLE-INDEX) = 2 AND PRECISION > 0
         ADD 2 TO PRECISION
         MOVE PRECISION TO DISPLAY-LENGTH.

     IF SELDVTYP(TABLE-INDEX) = 2
         IF DISPLAY-LENGTH > MAX-LENGTH
             DISPLAY "COLUMN VALUE TOO LARGE FOR DATA BUFFER."
             GO TO END-SQL
         ELSE
             MOVE DISPLAY-LENGTH TO SELDVLN(TABLE-INDEX).

*    COERCE DATATYPES TO VARCHAR2.
     MOVE 1 TO SELDVTYP(TABLE-INDEX).

*    DISPLAY COLUMN HEADING.
     MOVE SEL-DH-VNAME(TABLE-INDEX) TO COLUMN-NAME.
     DISPLAY COLUMN-NAME(1:SELDVLN(TABLE-INDEX)), " "
         WITH NO ADVANCING.

*FETCH A ROW AND PRINT THE SELECT-LIST VALUE.

 FETCH-ROWS.
     EXEC SQL  FETCH C1 USING DESCRIPTOR SELDSC  END-EXEC.
     IF SQLCODE NOT = 0
         MOVE "Y" TO NO-MORE-DATA.
     IF SQLCODE = 0
         PERFORM PRINT-COLUMN-VALUES
             VARYING TABLE-INDEX FROM 1 BY 1
             UNTIL TABLE-INDEX > SQLDNUM IN SELDSC
         DISPLAY " ".

*PRINT A SELECT-LIST VALUE.

 PRINT-COLUMN-VALUES.
     IF SEL-DI(TABLE-INDEX) = -1
       DISPLAY NULL-VAL(1:SELDVLN(TABLE-INDEX)), " "
           WITH NO ADVANCING
     ELSE
       DISPLAY SEL-DV(TABLE-INDEX)(1:SELDVLN(TABLE-INDEX)), " "
           WITH NO ADVANCING
     END-IF.
```

12

ユーザー・イグジット

この章では、SQL*Forms アプリケーションおよび Oracle Forms 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。まず、SQL*Forms アプリケーションとユーザー・イグジットとの間でインタフェースをとるために使用する EXEC IAF 文について解説します。次に、SQL*Forms ユーザー・イグジットの作成方法とリンク方法を説明します。また、Oracle Forms で EXEC TOOLS 文を使用する方法も紹介します。（SQL*Forms では EXEC TOOLS はサポートされていません。）これらを理解すれば、EXEC IAF 文を使って既存のアプリケーションの機能を強化できる一方、EXEC TOOLS 文を使って新しいアプリケーションを作成できるようになります。

EXEC IAF は将来廃止される予定なので、新しいアプリケーションでは EXEC TOOLS を使用してください。

この章は、次のトピックで構成されています。

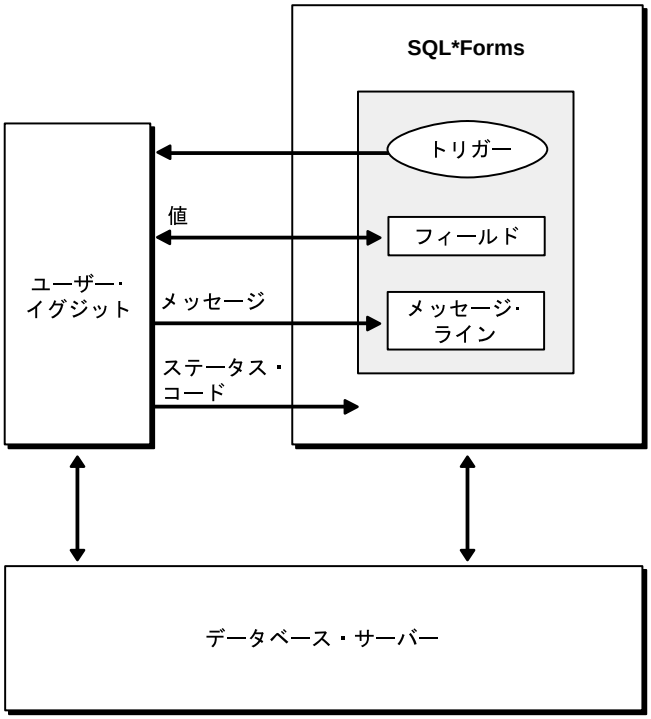
この章は補足です。ユーザー・イグジットの詳細は、『Oracle Forms デザイナーズ・リファレンス』、『Oracle Forms リファレンス・マニュアル、Vol 2』、およびシステム固有の Oracle マニュアルを参照してください。

ユーザー・イグジットとは

ユーザー・イグジットとは、特定の処理を実行するためにユーザーにより作成され、SQL*Forms によってコールされる、ホスト言語のサブルーチンです。ユーザー・イグジットに SQL コマンドおよび PL/SQL ブロックを埋め込んで、ホスト・プログラムと同じようにプリコンパイルできます。

SQL*Forms トリガーからコールされると、ユーザー・イグジットは実行後、SQL*Forms にステータス・コードを戻します（図 12-1 を参照）。ユーザー・イグジットでは、SQL*Forms のステータス行へのメッセージの表示、フィールド値の取得と設定、Oracle8*i* データの操作、高速計算、表参照を実行できます。さらに、異なるデータベースへのログインも実行できます。

図 12-1　ユーザー・イグジットとやり取りする SQL*Forms



ユーザー・イグジットを作成する利点

SQL*Forms バージョン 3 では、トリガー内で PL/SQL ブロックを使えます。したがってほとんどの場合は、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えます。ユーザー・イグジットが必要になったときは、USER_EXIT 関数を使えば 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、SQL*Forms コマンドに比べて、ユーザー・イグジットは記述方法もインプリメントの方法も複雑です。したがって、ユーザー・イグジットの使用は、SQL、PL/SQL、SQL*Forms の範囲を超える処理の実行に限定するのが一般的です。通常は次のような処理に使います。

- C や FORTRAN などの第 3 世代言語で実行する方が迅速または容易な演算（数値の統合など）
- デバイスおよびプロセスのリアルタイム制御（プリンタやグラフィックス・デバイスへの一連の命令の発行など）
- 拡張プロシージャ機能を必要とするデータ操作（再帰的ソートなど）
- 特殊なファイル I/O 処理

ユーザー・イグジットの開発

この項では、SQL*Forms ユーザー・イグジットを開発する方法の概要を紹介します。詳細は、後の項で説明します。Oracle Forms で使用可能な EXEC TOOLS 文の詳細は、「EXEC TOOLS 文」を参照してください。

ユーザー・イグジットをフォーム内に組み込むには、次の手順に従ってください。

1. サポートされているホスト言語でユーザー・イグジットを作成する。
2. ソース・コードをプリコンパイルする。
3. プリコンパイル後のソース・コードをコンパイルする。
4. GENXTB ユーティリティを使ってデータベース表 IAPXTB を作成する。
5. SQL*Forms の GENXTB フォームを使って、ユーザー・イグジットの情報をデータベース表 IAPXTB に挿入する。
6. GENXTB ユーティリティを使って、表から情報を読み込み、IAPXIT ソース・モジュールを作成する。続いて、IAPXIT ソース・モジュールをコンパイルする。
7. 標準の IAP オブジェクト・モジュール、ユーザー・イグジットのオブジェクト・モジュール、およびステップ 6 で作成した IAPXIT オブジェクト・モジュールをリンクすることにより、新しい IAP（フォームを実行する SQL*Forms コンポーネント）を作成する。
8. 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する。

9. フォームを実行するときには新しい IAP を使うようにオペレータに伝える。この新しい IAP が標準フォームを置換するときは、この伝達は必要ありません。詳細は、使用しているシステム固有の Oracle マニュアルを参照してください。

ユーザー・イグジットの作成

次の種類の文を使って、SQL*Forms ユーザー・イグジットを記述できます。

- ホスト言語
- EXEC SQL
- EXEC ORACLE
- EXEC IAF GET
- EXEC IAF PUT

この項では EXEC IAF GET および EXEC IAF PUT 文を中心に説明します。この 2 つの文によって、SQL*Forms とユーザー・イグジット間で値の受渡しができるようになります。

変数の要件

EXEC IAF 文内で使う変数は、フォーム定義内で使うフィールド名に対応していなければなりません。ブロック名を指定しなかったためにフィールド参照が一意でなくなると、エラーが発生します。フォーム・フィールドへの参照が無効またはあいまいなときはエラーが発生します。

ホスト変数は、ユーザー・イグジットの宣言文で命名し、EXEC IAF 文の中では前にコロン（:）を付けてください。

注意 : EXEC IAF GET 文と PUT 文では、標識変数は使用できません。

IAF GET 文

この文を使うと、ユーザー・イグジットでフォームのフィールドから値を " 取得 " し、それをホスト変数に割り当てることができます。その結果、ユーザー・イグジットでの計算、データ操作、更新などにこのデータを使えます。GET 文の構文は次のとおりです。

```
EXEC IAF GET field_name1, field_name2, ...
    INTO :host_variable1, :host_variable2, ... END-EXEC.
```

このとき *field_name* は次の SQL*Forms 変数のいずれかとなります。

- フィールド
- block.field
- システム変数

- グローバル変数
- フィールド、block.field、システム変数、グローバル変数のいずれかの値を格納するホスト変数（先頭コロン付き）

修飾しない場合、*field_name* には一意の値を指定してください。

ユーザー・イグジットでフィールド値を取得（GET）して、その値をホスト変数に割り当てる方法を次の例に示します。

```
EXEC IAF GET employee.job INTO :NEW-JOB END-EXEC.
```

フィールド値はすべて文字列です。可能であれば、GET はフィールド値を対応するホスト変数のデータ型に変換します。不当な変換またはサポートされていないデータ型を変換しようとすると、エラーが発生します。

前の例では、定数を使って *block.field* を指定しています。次のように、ホスト文字列をブロック名およびフィールド名の指定に使用することもできます。

```
MOVE "employee.job" TO BLKFLD.
EXEC IAF GET :BLKFLD INTO :NEW-JOB END-EXEC.
```

フィールドが一意の場合を除いて、ホスト文字列には *block.field* という形式でブロック名とフィールド名の両方をピリオドで区切って指定しなければなりません。たとえば次の指定は無効です。

```
MOVE "employee" TO BLK.
MOVE "job" TO FLD.
EXEC IAF GET :BLK.:FLD INTO :NEW-JOB END-EXEC.
```

GET 文のフィールド・リストには明示的なフィールド名と変数内に保存されているフィールド名をともに指定できます。ただし単一フィールドの参照では、これらを組み合わせて指定することはできません。たとえば次の指定は無効です。

```
MOVE "job" TO FLD.
EXEC IAF GET employee.:FLD INTO :NEW-JOB END-EXEC.
```

IAF PUT 文

この文を使うと、ユーザー・イグジットで定数とホスト変数の値をフォームのフィールドに " 入れる " ことができます。つまり、SQL*Forms 画面上に任意の値およびメッセージをユーザー・イグジットで表示できます。PUT 文の構文は次のとおりです。

```
EXEC IAF PUT field_name1, field_name2, ...
   VALUES (:host_variable1, :host_variable2, ...) END-EXEC.
```

このとき *field_name* は次の SQL*Forms 変数のいずれかとなります。

- フィールド

- block.field
- システム変数
- グローバル変数
- フィールド、block.field、システム変数、グローバル変数のいずれかの値を格納するホスト変数（先頭コロン付き）

ユーザー・イグジットで数値定数、文字列定数、およびホスト変数をフォームのフィールドに書き出す（PUT）方法を次の例に示します。

```
EXEC IAF PUT employee.number, employee.name, employee.job
    VALUES (7934, 'MILLER', :NEW-JOB) END-EXEC.
```

GET と同様に、PUT でもホスト文字列を使ってブロック名およびフィールド名を次のように指定できます。

```
MOVE "employee.job" TO BLKFLD.
EXEC IAF PUT :BLKFLD VALUES (:NEW-JOB) END-EXEC.
```

文字モード端末のとき、このフィールドが現在表示されているページ内にある場合は、フィールドに PUT される値は割当てが行われたときではなく、ユーザー・イグジットが戻ったときに表示されます。ブロックモード端末のときは、次にデバイスからフィールドを読み込むときにこの値が表示されます。

ユーザー・イグジットでフィールドの値が何度か変更されても、最後に変更された値だけが有効となります。

ユーザー・イグジットのコール

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 が提供する）という名前のパッケージ・プロシージャを使います。使う構文は次のとおりです。

```
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て、*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 が発行するエラー・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・イグジット文字列は引用符（二重引用符は不可）で囲まれます。

ユーザー・イグジットへのパラメータの引渡し

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。

しかしユーザー・イグジット文字列を使えば、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・コマンドを使うと、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。

```
USER_EXIT('LOOKUP 2025 A', 'Lookup failed');
```

次の例に示すように、この機能を使って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・イグジット文字列の解析は、SQL*Forms ではなくユーザー・イグジットによって実行されます。

フォームへの値の復帰

ユーザー・イグジットでは SQL*Forms に制御が戻るときに必ずコードが戻ります。このコードはユーザー・イグジットが成功したか、失敗したか、致命的エラーが発生したかどうかを示します。このリターン・コードはプリコンパイラが生成する整定数です（次の項を参照）。この 3 種類の結果は次の意味を持ちます。

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」コードが戻ったときは、SQL*Forms はこの変更を破棄しません。また、逆戻りコード・スイッチが設定されているときに「成功」コードが戻されたときにも、SQL*Forms は変更を破棄しません。

IAP 定数

プリコンパイラは、リターン・コードで使用するために 3 つの記号定数を生成します。この 3 つの記号定数には、接頭辞として IAP が付けられます。（たとえば、IAPSUCC、IAPFAIL、IAPFTL など。）

SQLIEM 関数の使用

関数 SQLIEM をコールすることにより、SQL*Forms が表示するエラー・メッセージをユーザー・イグジットから指定できます。このエラー・メッセージは、トリガー・ステップでエラーが発生したときはメッセージ行に、このステップで致命的エラーが発生したときはエラー表示画面に表示されます。このステップに対して定義されていた任意のメッセージが、ここで指定したメッセージに置き換えられます。

SQLIEM 関数の構文は次のとおりです。

```
     CALL "SQLIEM" USING ERROR-MESSAGE  ERROR-MESSAGE-LEN.
```

ERROR-MESSAGE は文字変数、*ERROR-MESSAGE-LEN* は整変数です。Oracle プリコンパイラによって適切な外部関数宣言が生成されます。どちらのパラメータも参照形式で渡してください。つまり、値ではなくアドレスを渡してください。SQLIEM は SQL*Forms の関数であり、他の Oracle Tools からはコールできません。

WHENEVER の使用

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。

サンプル・プログラム 5: Oracle Forms のユーザー・イグジット

このユーザー・イグジットは、フォーム・フィールドを連結します。このユーザー・イグジットを Oracle Forms トリガーからコールするには、次の構文を使用します。

```
<user_exit>('CONCAT <field1>, <field2>, ..., <result_field>');
```

user_exit は Oracle Forms に付属のパッケージ・プロシージャ、CONCAT はユーザー・イグジットの名前です。このユーザー・イグジットを起動する CONCAT フォームのサンプルを次に示します。

```
*****************************************************************
* 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.                      *
*****************************************************************

 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CONCAT.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  FIELD-NAME             PIC X(80)  VARYING.
```

```
01  FIELD-VALUE            PIC X(80)  VARYING.
01  RESULT                 PIC X(800) VARYING.
    EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL INCLUDE SQLCA END-EXEC.

01  EXIT-MESSAGE           PIC X(80).
01  EXIT-MESSAGE-LEN       PIC S9(9) COMP.
01  RTN-CODE               PIC S9(9) COMP.
77  INDX                   PIC S9(4) COMP.
01  DONE-FLAG              PIC X.
    88  DONE               VALUE 'Y'.
01  PTR                    PIC S9(4) COMP.
01  WS-CMD-LINE.
    05  WS-CMD-LINE-Y      PIC X(80).
    05  WS-CMD-LINE-X      REDEFINES WS-CMD-LINE-Y
                           PIC X OCCURS 80.
01  WS-FIELD-NAME-AREA.
    05  WS-FIELD-NAME      PIC X(80).
    05  WS-FIELD-NAME-X    REDEFINES WS-FIELD-NAME
                           PIC X OCCURS 80.
    05  WS-FIELD-NAME-LEN  PIC S9(4) COMP.

LINKAGE SECTION.
01  CMD-LINE               PIC X(80).
01  CMD-LINE-LEN           PIC S9(9) COMP.
01  ERR-MSG                PIC X(80).
01  ERR-MSG-LEN            PIC S9(9) COMP.
01  IN-QUERY               PIC S9(9) COMP.
01  RETURN-VALUE           PIC S9(9) COMP.

PROCEDURE DIVISION USING CMD-LINE, CMD-LINE-LEN,
                         ERR-MSG, ERR-MSG-LEN, IN-QUERY.

MAIN.
    MOVE 1 TO PTR.
    MOVE SPACE TO RESULT-ARR.
    MOVE ZERO TO RESULT-LEN.
    MOVE SPACE TO DONE-FLAG.
    MOVE 7 TO INDX.
    MOVE CMD-LINE TO WS-CMD-LINE-Y.
    PERFORM CMD-LINE-PARSE UNTIL DONE.
    EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
    MOVE WS-FIELD-NAME TO FIELD-NAME-ARR.
    MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN.
    EXEC IAF PUT :FIELD-NAME VALUES(:RESULT) END-EXEC.
    MOVE SQL-IAPXIT-SUCCESS TO RTN-CODE.
```

```
    EXIT PROGRAM GIVING RTN-CODE.

CMD-LINE-PARSE.
    MOVE ZERO TO WS-FIELD-NAME-LEN.
    MOVE SPACES TO WS-FIELD-NAME.
    MOVE SPACES TO FIELD-NAME-ARR.
    MOVE ZERO TO FIELD-NAME-LEN.
    PERFORM GET-FIELD-NAME
        UNTIL WS-CMD-LINE-X(INDX) = ',' OR DONE.
    IF WS-CMD-LINE-X(INDX) = ','
        MOVE SPACES TO FIELD-NAME-ARR
        MOVE WS-FIELD-NAME TO FIELD-NAME-ARR
        MOVE WS-FIELD-NAME-LEN TO FIELD-NAME-LEN
        MOVE SPACES TO FIELD-VALUE-ARR
        EXEC IAF GET :FIELD-NAME INTO :FIELD-VALUE END-EXEC
        STRING FIELD-VALUE-ARR
            DELIMITED BY SPACE
            INTO RESULT-ARR
            WITH POINTER PTR
        ADD FIELD-VALUE-LEN TO RESULT-LEN
        ADD 1 TO INDX.

GET-FIELD-NAME.
    IF WS-CMD-LINE-X(INDX) NOT EQUAL SPACE
        ADD 1 TO WS-FIELD-NAME-LEN
        MOVE WS-CMD-LINE-X(INDX) TO
            WS-FIELD-NAME-X(WS-FIELD-NAME-LEN).
    ADD 1 TO INDX.
    IF INDX > CMD-LINE-LEN
        MOVE 'Y' TO DONE-FLAG.

SQL-ERROR.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    MOVE SQLERRMC TO EXIT-MESSAGE.
    MOVE SQLERRML TO EXIT-MESSAGE-LEN.
    CALL "SQLIEM" USING EXIT-MESSAGE EXIT-MESSAGE-LEN.
    MOVE SQL-IAPXIT-FAILURE TO RTN-CODE.
    EXIT PROGRAM.
```

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

ユーザー・イグジットはスタンドアロン型のホスト・プログラムと同じ方法でプリコンパイルされます。第 14 章の「プリコンパイラのオプション」を参照してください。

ユーザー・イグジットのコンパイル手順については、使用しているシステム固有の Oracle マニュアルを参照してください。

GENXTB ユーティリティの使用

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。次に示すように、オペレーティング・システムのコマンド行で GENXTB フォームを実行することによって、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
- ホスト言語コード（COBOL または C）
- 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ってその表を読み込み、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 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 ユーティリティの構文は次のとおりです。

```
GENXTB username/password outfile
```

outfile は、GENXTB で作成されるアセンブラまたはソース・プログラムに付ける名前です。

SQL*Forms へのユーザー・イグジットのリンク

ユーザー・イグジットをコールするフォームを実行するには、まずユーザー・イグジットを IAP に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 にリンクできます。

IAP の新しい実行可能コピーを作成するには、ユーザー・イグジットのオブジェクト・モジュール、標準 IAP モジュール、IAPXIT モジュール、その他の必要なモジュールを Oracle およびホスト言語リンク・ライブラリからリンクします。リンクの方法はシステムによって異なりますので、使用しているシステム固有の Oracle マニュアルを参照してください。

SQL*Forms ユーザー・イグジットのためのガイドライン

この項では、よく見られる問題を回避するためのガイドラインを示します。

イグジットの命名

ユーザー・イグジットの名前は Oracle の予約語であってはなりません。また、SQL*Forms コマンドの名前、関数コードの名前、SQL*Forms に使われる外部定義済みの名前と競合を起こす名前は使わないでください。

SQL*Forms は検索前にそのユーザー・イグジットの名前を大文字に変換します。このため、ホスト言語で大文字と小文字が区別される場合は、ソース・コードのイグジット名は大文字でなければなりません。

ソース・コード内のユーザー・イグジットのエントリ・ポイントの名前はユーザー・イグジット自体の名前となります。イグジット名は、使用しているホスト言語およびオペレーティング・システムで有効なファイル名でなければなりません。

Oracle への接続

ユーザー・イグジットは、SQL*Forms で確立された接続を介して Oracle8*i* と通信します。ただし、ユーザー・イグジットで SQL*Net を使って任意のデータベースに追加の接続を確立できます。詳細は 3-12 ページの「同時ログイン」を参照してください。

I/O コールの発行

SQL*Forms の I/O ルーチンがホスト言語のプリンタ I/O ルーチンと競合することがあります。その場合、ユーザー・イグジットはプリンタ I/O コールを発行できません。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。

ホスト変数の使用

スタンドアロン型のプログラムに適用されるホスト変数の制限事項はユーザー・イグジットにも適用されます。ホスト変数はユーザー・イグジットの宣言文で命名しなければならず、EXEC SQL 文および EXEC IAF 文では前にコロンを付ける必要があります。ただし、EXEC IAF 文ではホスト配列は使用できません。

表の更新

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 は上書きされてしまいます。

コマンドの発行

ユーザー・イグジットからは、COMMIT コマンドまたは ROLLBACK コマンドを発行しないでください。ユーザー・イグジットから COMMIT または ROLLBACK コマンドを発行すると、Oracle8*i* は、ユーザー・イグジットが行った作業だけでなく 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します。かわりに SQL*Forms トリガーから COMMIT または ROLLBACK コマンドを発行してください。データ定義コマンド（ALTER や CREATE など）の場合も同様です。データ定義コマンドは実行の前後に暗黙的に COMMIT を発行するからです。

EXEC TOOLS 文

EXEC TOOLS 文は、ユーザー・イグジットからの GET コールバックおよび SET コールバック、例外コールバックを処理するための一般的な方法を提供することによって、基本的な Oracle Toolset （Oracle Forms、Oracle Reports、Oracle Graphics）をサポートします。ここでは Oracle Forms を取り上げて説明しますが、Oracle Reports および Oracle Graphics でも考え方は同じです。

EXEC SQL および EXEC ORACLE、ホスト言語文の他にも、次の EXEC TOOLS 文を使って Oracle Forms ユーザー・イグジットを記述できます。

- SET
- GET
- MESSAGE

EXEC TOOLS GET 文および EXEC TOOLS SET 文は、SQL*Forms で使用される EXEC IAF GET 文および EXEC IAF PUT 文に相当します。IAF GET 文および IAF PUT 文とは異なり、TOOLS GET 文および TOOLS SET 文では標識変数を使用できます。EXEC TOOLS MESSAGE 文は、メッセージ処理関数 SQLIEM に相当します。EXEC TOOLS SET CONTEXT 文と GET CONTEXT 文は新機能なので、SQL*Forms のバージョン 3 では使用できません。

> **注意 :** COBOL にはポインタ・データ型がないので、Pro*COBOL プログラムでは SET CONTEXT 文と GET CONTEXT 文は使えません。

EXEC TOOLS SET

EXEC TOOLS SET 文は、ユーザー・イグジットから Oracle Forms へ値を渡します。この文は、特にホスト変数および定数の値を Oracle Forms 変数および項目に割り当てます。代入された値は、ユーザー・イグジットがフォームに制御を戻した後で表示されます。

EXEC TOOLS SET 文を記述するには、次の構文を使います。

```
EXEC TOOLS SET form_variable[, ...]
    VALUES ({:host_variable[:indicator] | constant}[, ...])
END-EXEC.
```

form_variable には、Oracle Forms のフィールドまたはパラメータ、システム変数、グローバル変数、あるいはこれらの項目のどれかの名前が格納されているホスト変数（接頭辞はコロン）を指定します。

次に示す Pro*COBOL の例では、ユーザー・イグジットは従業員名を（オプションの標識とともに）Oracle Forms に渡します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
    01  ENAME      PIC X(20) VARYING.
    01  ENAME-IND  PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
MOVE "MILLER" TO ENAME-ARR.
MOVE 6 TO ENAME-LEN.
MOVE ZERO TO ENAME-IND.
EXEC TOOLS SET emp.ename VALUES (:ENAME:ENAME-IND) END-EXEC.
```

このとき *emp.ename* は Oracle Forms の block.field の 1 つです。

EXEC TOOLS GET

EXEC TOOLS GET 文は、Oracle Forms からユーザー・イグジットへ値を渡します。この文は、特に Oracle Forms 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・イグジットでそれらの値を任意の目的に使えます。

EXEC TOOLS GET 文を記述するときは次の構文を使ってください。

```
EXEC TOOLS GET form_variable[, ...]
    INTO :host_variable[:indicator][, ...] END-EXEC.
```

form_variable は Oracle Forms のフィールドまたはパラメータ、システム変数、グローバル変数、あるいはこれらの項目のどれかの名前が格納されているホスト変数です。

次の例では、Oracle Forms は block.field*emp.ename* から従業員名をユーザー・イグジットに渡します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
    01  ENAME      PIC X(20) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC TOOLS GET emp.ename INTO :ENAME END-EXEC.
...
```

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。ユーザー・イグジットによりフォームに制御が戻った後にメッセージが 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、次の構文を使います。

```
EXEC TOOLS MESSAGE message_text [severity_code] END-EXEC.
```

message_text には引用符で囲まれた文字列または文字ホスト変数を指定し、オプションの *severity_code* には整定数またはホスト変数を指定します。MESSAGE 文に標識変数は指定できません。

次に示す Pro*COBOL の例では、ユーザー・イグジットは Oracle Forms にエラー・メッセージと重大度コードを渡します。

```
EXEC TOOLS MESSAGE "Bad field name! Please reenter." 15
END-EXEC.
```

13

ラージ・オブジェクト（LOB）

この章では、埋込み SQL 文により提供される LOB（Large Object）データ型のサポートについて記述します。4 種類の LOB 型について説明し、従来の LONG および LONG RAW データ型と比較します。

Pro*COBOL の埋込み SQL インタフェースでは、PL/SQL 言語と同様の機能を提供します。

LOB 文およびその LOB オプションとホスト変数について説明します。

最後に、LOB インタフェースを使った Pro*COBOL によるプログラミング例を説明します。

主に次の項で構成されています。

LOB とは

LOB（large object）とは、ASCII テキスト、各国文字のテキスト、さまざまなグラフィック・フォーマットのファイルおよびサウンド波形などの多くのデータ（最大 4GB）を格納するために使われるデータベース型のことです。

内部 LOB

内部 LOB（BLOB、CLOB、NCLOB）は、データベースの表領域に格納されます。また、データベース・サーバーのトランザクション・サポート（コミット、ロールバックなど、内部 LOB で使う機能）が有効です。

BLOB（Binary LOB）には、ビデオ・クリップなどの非構造化バイナリ（" ロー " とも呼ばれます）データが格納されます。

CLOB（Character LOB）には、データベース・キャラクタ・セットのシングルバイト固定幅文字列データのラージ・ブロックが保存されます。

NCLOB（National Character LOB）には、各国キャラクタ・セットのシングルバイト、固定幅または可変幅マルチバイト文字列データのラージ・ブロックが格納されます。

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）には、データは外部バイナリ・ファイルに格納されます。BFILE では、GIF、JPEG、MPEG、MPEG2、テキストなどのフォーマットを扱うことができます。

BFILE のセキュリティ

DIRECTORY オブジェクトは、BFILE にアクセスして使うときに使います。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論理別名です。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。

次の 2 種類の SQL 文を BFILE で使うことができます。

- DDL（Data Definition Language）SQL 文の CREATE、REPLACE、ALTER および DROP。
- DIRECTORY オブジェクト上のシステムとオブジェクトに対する READ 特権の GRANT および REVOKE を行う DML（Data Management Language）SQL 文。

CREATE DIRECTORY ディレクティブの例です。

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir' END-EXEC.
```

ユーザーまたはロールは、GRANT などの DML（Data Manipulation Language）文の権限が割り当てを与えられている場合にだけ、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

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott END-EXEC.
```

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 パラメータの設定の変更によって、デフォルト値を変更できます。

DIRECTORY オブジェクトおよび BFILE セキュリティの詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。GRANT コマンドの詳細は、『Oracle SQL サーバー・リファレンス』を参照してください。

LOB と、LONG、LONG RAW との比較

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 4 ギガバイトです。LONG および LONG RAW の最大サイズは 2 ギガバイトです。
- 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うことができます。LONG および LONG RAW では、順次アクセス方法だけです。
- 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。
- 表には複数の 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 ROW はサポートしない予定です。移行の詳細は、『Oracle8*i* 移行ガイド』を参照してください。

LOB ロケータ

LOB ロケータによって、実際の LOB 内容がポイントされます。LOB を取り出したときにロケータが戻されます。LOB の内容を取り出したときは戻されません。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。

テンポラリ LOB

テンポラリ LOB を作成すると、LOB データベースが使いやすくなることがあります。テンポラリ LOB はローカル変数に似ていますが、表には関連付けられていません。LOB を作成したユーザーだけがロケータを使ってアクセスできます。セッションが終了すると削除されます。

テンポラリ BFILES はサポートしていません。テンポラリ LOB は、INSERT 文の WHERE 句、UPDATE 文の SET 句または DELETE 文の WHERE 句の入力変数（IN 値）だけで使うことができます。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・サポートは無効です。つまり、COMMITS または ROLLBACK することができません。

テンポラリ LOB ロケータは、トランザクションをまたがって使うことができます。テンポラリ LOB は、サーバーが異常終了したとき、およびデータベース SQL 操作からエラーが戻されたときも削除されます。

LOB バッファリング・サブシステム

LBS（LOB Buffering Subsystem）は、クライアントのアドレス空間で 1 つ以上の LOB バッファとして使うためのユーザー・メモリー領域です。

バッファリングには次の利点があります。特に、LOB の特定領域に小単位の読込み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。

- LOB に対する複数の読込み / 書込みをバッファに蓄積し、FLUSH ディレクティブが実行されたときにサーバーに書き込むため、サーバーへの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。

このバッファリングは、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限りません。サーバーの LOB に実際に更新を書き込むには、FLUSH 文を使います。

LOB にバッファされた読込み / 書込みは、ロケータを使って実行されます。バッファリングが使用可能なロケータを使うと、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、バッファされた WRITE に使われると更新済みロケータになります。最新の LOB バージョンには、バッファリング・サブシステムを介してアクセスします。これ以降のバッファされた LOB への WRITE は、更新済みロケータを使わないとできません。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・セッション間で移行することはできません。

LBS は、サーバーの LOB 値を FLUSH 文を使って更新するユーザーが管理します。LBS は、シングル・ユーザーで単一スレッドです。サーバー LOB の妥当性を確保するには、ROLLBACK および SAVEPOINT アクションを使います。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・セマンティッ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管理する必要があります。

LBS の詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

プログラムから LOB の使用方法

LOB への 2 つのアクセス方法

Pro*COBOL から LOB にアクセスするには、次の 2 つの方法があります。

- PL/SQL ブロックの DBMS_LOB パッケージ
- 埋込み SQL 文

SQL 文は、PL/SQL インタフェースと同じ機能を使えるように設計されています。

次の表では、PL/SQL からの LOB アクセスおよび Pro*COBOL の埋込み SQL 文を比較します。空欄は、機能がないことを示します。

表 13-1 LOB へのアクセス方法

PL/SQL[1]	Pro*COBOL の埋込み SQL
COMPARE()	
INSTR()	
SUBSTR()	
APPEND()	APPEND
:=	ASSIGN
CLOSE()	CLOSE
COPY()	COPY
CREATETEMPORARY()	CREATE TEMPORARY
	DISABLE BUFFERING
	ENABLE BUFFERING
ERASE()	ERASE
GETCHUNKSIZE()	DESCRIBE
ISOPEN()	DESCRIBE
FILECLOSE()	CLOSE
FILECLOSEALL()	FILE CLOSE ALL
FILEEXISTS()	DESCRIBE
FILEGETNAME()	DESCRIBE

表 13-1 LOB へのアクセス方法 (続き)

PL/SQL[1]	Pro*COBOL の埋込み SQL
FILEISOPEN()	DESCRIBE
FILEOPEN()	OPEN
BFILENAME()	FILE SET[2]
	FLUSH BUFFER
FREETEMPORARY()	FREE TEMPORARY
GETLENGTH()	DESCRIBE
=	
ISTEMPORARY()	DESCRIBE
LOADFROMFILE()	LOAD FROM FILE
OPEN()	OPEN
READ()	READ
TRIM()	TRIM
WRITE()	WRITE
WRITEAPPEND()	WRITE

[1] dbmslob.sql からコールします。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。

[2] SQL 関数に組み込まれている BFILENAME() も使えることがあります。

注意 : LOB の修正または変更を行う新しい文を使う前に、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。

アプリケーションでの LOB ロケータ

Pro*COBOL アプリケーションで LOB ロケータを使うには、次の擬似タイプを使います。

- SQL-BLOB
- SQL-CLOB
- SQL-NCLOB
- SQL-BFILE.

たとえば、MY-NCLOB と呼ばれる NCLOB 変数を宣言するには、次のようになります。

```
 01  MY-NCLOB     SQL-NCLOB.
```

LOB の初期化

内部 LOB

BLOB を初期化して空にするには、*EMPTY_BLOB()* 関数または ALLOCATE SQL 文を使います。CLOB および NCLOB の場合は、*EMPTY_CLOB()* 関数を使います。EMPTY_BLOB() および EMPTY_CLOB() の詳細は、『Oracle8*i* SQL 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 句のソースとして以外では使えません。

たとえば、次のとおりです。

```
     EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
        VALUES (EMPTY_BLOB(), EMPTY_CLOB()) END-EXEC.
```

ALLOCATE 文では、LOB ロケータを割り当ててから、初期化して空にします。次のコードは、前の例と同じです。

```
...
 01  A-BLOB       SQL-BLOB.
 01  A-CLOB       SQL-CLOB.
...
     EXEC SQL ALLOCATE :A-BLOB END-EXEC.
     EXEC SQL ALLOCATE :A-CLOB END-EXEC.
     EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
        VALUES (:A-BLOB, :A-CLOB) END-EXEC.
```

外部 LOB

BFILE および FILENAME の DIRECTORY 別名を初期化するには、LOB FILE SET 文を次のように使います。

```
...
 01  ALIAS         PIC X(14) VARYING.
```

```
 01  FILENAME     PIC X(14) VARYING.
 01  A-BFILE      SQL-BFILE.
...
     MOVE "lob_dir" TO ALIAS-ARR.
     MOVE 7 TO ALIAS-LEN.
     MOVE "image.gif" TO FILENAME-ARR
     MOVE 9 TO FILENAME-LEN..
     EXEC SQL ALLOCATE :A-BFILE END-EXEC.
     EXEC SQL LOB FILE SET :A-BFILE
        DIRECTORY = :ALIAS, FILENAME = :FILENAME END-EXEC.
     EXEC SQL INSERT INTO file_table (a_bfile) VALUES (:A-BFILE) END-EXEC.
```

DIRECTORY オブジェクトの命名規則および DIRECTORY オブジェクトの特権の詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』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 '）機能を使って、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 :A-BFILE END-EXEC.
```

注意 : BFILENAME() では、ディレクトリおよびファイル名の権限または物理ディレクトリが実際に存在するかどうかはチェックしません。BFILE ロケータを使った後続のファイル・アクセスによってそのチェックが行われます。これらのチェックをしてファイルにアクセスできない場合は、エラーが返されます。

テンポラリ LOB

埋込み SQL の LOB CREATE TEMPORARY 文を使って最初にテンポラリ LOB を作成したときに、テンポラリ LOB が初期化され空になります。EMPTY_BLOB() および EMPTY_CLOB() 関数は、テンポラリ LOB では使えません。

LOB 文のルール

LOB 文を使うときのルールについて説明します。

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 文を使って LOB を操作するときに適用されます。

- 埋込み SQL の LOB 文では、FOR 句は使えません。複数の LOB ロケータを使うことはできません。ただし、ALLOCATE および FREE 文では、FOR 句を使うことができます。

- 分散 LOB はサポートされていません。新規の SQL LOB 文では AT データベース句を使えますが、同一 SQL LOB 文で、複数のデータベース接続を使って作成または ALLOCATE された LOB ロケータは併用できません。

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。

- 読込みまたは書込みアクセスのエラーは、次にサーバーにアクセスしたときにレポートが出力されます。このため、エラー復旧のコードは、ユーザーが作成する必要があります。
- バッファされた書込みにより LOB を更新するときは、LOB バッファリング・サブシステムを経由せずに、その LOB を更新しないでください。
- バッファリングが使用可能な更新済み LOB ロケータは、IN パラメータとして、PL/SQL プロシージャに渡すことができますが、IN OUT または OUT パラメータとして渡すことはできません。エラーが返されます。更新済みロケータを返そうとしたときにも、エラーが返されます。
- バッファリングが使用可能な更新済みロケータを、別のロケータに ASSIGN することはできません。
- バッファされた書込みは LOB 値に追加することができますが、その LOB の最後の後に 1 文字の開始オフセットが必要です。LBS では、APPEND 文を使うと、データベース・サーバーの LOB の最後が 0 バイトの充填文字または空白になるので使うことができません。
- ホスト・ロケータのバインド変数の文字セットとデータベース・サーバーの CLOB は、同じでなければなりません。
- バッファリングが使用可能なロケータを使った場合に、ASSIGN、READ および WRITE 文が実行されます。
- バッファリングが使用可能なロケータを使うと、APPEND、COPY、ERASE、DESCRIBE（LENGTH だけ）、SELECT および TRIM 文ではエラーが発生します。また、別のロケータによってバッファされた方法で、ロケータによってポイントされた LOB にアクセスしている場合は、バッファリングが使用可能なロケータを使って、これらの文を使ったときはエラーが返されます。

注意 : FLUSH 文を LOB で使うときは、次の処理の前に、LOB バッファリング・システムで使用可能になっている必要があります。

- トランザクションのコミット
- 現行のトランザクションと別のトランザクション間の移行
- LOB 上でのバッファ操作の使用禁止

ホスト変数に適用されるルール

LOB 文の場合、次のルールおよび注意を使います。

- *src* および *dst* を使って、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 バイトの整数値 PIC S9(9) COMP として宣言されます。値は、0 ～ 4 ギガバイトに制限されています。
- LOB ロケータでは、NULL の概念が使われています。LOB 文では、標識変数は必要ありません。NULL は、`amt`、`src_offset` などの数値変数とともに使えません。使った場合はエラーになります。
- オフセット値 `src_offset` および `dst_offset` のデフォルト値は 1 です。

LOB 文

アルファベット順に文を説明します。すべての説明文で、*database* はデータベース接続を示します。

APPEND

用途

この文では、別の LOB の最後に LOB 値を追加します。

構文

```
EXEC SQL [AT [:]database] LOB APPEND :src TO :dst END-EXEC.
```

ホスト変数

src (IN)

ソース LOB を一意に参照する内部 LOB ロケータ。

dst (IN OUT)

宛先 LOB を一意に参照する内部 LOB ロケータ。

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ます。宛先 LOB が最大 4 ギガバイトまで拡張されます。LOB が 4 ギガバイト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、あらかじめ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 型になる必要があります。両方のロケータの LOB バッファが有効になっていない場合は、エラーになります。

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 END-EXEC.
```

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、両方のロケータが同じ LOB 値を参照します。宛先 LOB ロケータは、初期化された（ALLOCATE された）使用可能なロケータでなければなりません。

内部 LOB では、宛先ロケータが表に格納されている場合だけ、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*COBOL では、宛先ロケータが格納されているオブジェクトの 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た場合は、エラーが返されます。その逆も同様です。*src* および *dst* LOB が同じ型でない場合も、エラーが返されます。

ソース・ロケータがバッファリングが使用可能な内部 LOB 用であった場合、LOB バッファリング・サブシステムを介した LOB 値の修正にソース・ロケータを使っていた場合、および WRITE 後に FLUSH されていないバッファの場合は、ソース・ロケータは宛先ロケータに割り当てられません。LOB バッファリング・サブシステムを介して LOB 値を修正するときに、1 つの LOB で複数のロケータを使うことができないためです。

CLOSE

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 END-EXEC.
```

ホスト変数

src (IN OUT)

LOB または BFILE のロケータをクローズします。

使用上の注意

複数のロケータまたは同一ロケータを使って、同一 LOB を 2 回クローズすると、エラーになります。外部 LOB では、BFILE の場合には一度もオープンされていないときでも、エラーは発生しません。

事前にオープンされていた 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 ROLL BACK 時にオープンしている LOB は、クローズされずにすべて破棄されます。

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] END-EXEC.
```

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 の文字数。

src (IN)

ソース LOB のロケータ。

src_offset (IN)

CLOB または NCLOB の文字数、および BLOB のバイト数。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)

宛先のオフセット。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現行データの最後を超えている場合は、宛先 LOB の現行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。

宛先 LOB の現行の長さを超えて拡張される場合は、新しく書き込まれるデータを格納できるように宛先 LOB が拡張されます。4 ギガバイトを超えて LOB が拡張されると、ランタイム・エラーになります。

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、同じ型でなければなりません。LOB バッファリングは、両方のロケータに対して使用可能でなければなりません。

`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は、エラーを発行せずに操作が終了します。

テンポラリ LOB を永続 LOB にするには、COPY 文を使って、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。

CREATE TEMPORARY

用途

テンポラリ LOB を作成します。

構文

```
EXEC SQL [AT [:]database] LOB CREATE TEMPORARY :src END-EXEC.
```

ホスト変数

src (IN OUT)

実行前で IN のときは、`src` は事前に ALLOCATE された LOB ロケータです。

実行後で OUT になったときは、`src` は新しい空のテンポラリ 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・サーバー上に新しく作成された、表に依存しないテンポラリ LOB をポイントします。テンポラリ LOB は空で、長さゼロです。

セッション終了時には、すべてのテンポラリ LOB が解放されます。テンポラリ LOB への READ および WRITE は、バッファ・キャッシュを経由しません。

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 END-EXEC.
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

この文では、BFILE をサポートしていません。後続の読込みまたは書込みは、LBS 経由では行われません。

注意 : 変更を永続なものにするために FLUSH BUFFER コマンドを使ってください。この文は、LOB バッファリング・サブシステムが作成した変更を、暗黙的にフラッシュすることはありません。

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 END-EXEC.
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

この文では、BFILE をサポートしていません。後続の読込みおよび書込みは、LBS を経由して行われます。

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] END-EXEC.
```

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 の開始からのオフセットです。1 から指定できます。

使用上の注意

この文では、BFILE をサポートしていません。

実行後、消去された実際の文字数またはバイト数が *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* はゼロ文字またはゼロ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。

FILE CLOSE ALL

用途

現行の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 END-EXEC.
```

使用上の注意

適切にクローズされなかったオープン・ファイルがセッションに存在する場合は、FILE CLOSE ALL 文を使って、セッションでオープンしているファイルをすべてクローズし、ファイル操作を始めから再開できます。

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 END-EXEC.
```

ホスト変数

file (IN OUT)

DIRECTORY 別名および FILENAME を設定する 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、この文で使う前に、ALLOCATE されている必要があります。

DIRECTORY 別名および FILENAME は入力必須です。

DIRECTORY 別名の最大長は 30 バイトです。FILENAME の最大長は 255 バイトです。

DIRECTORY 別名および FILENAME 属性がサポートされている外部データ型は、CHARZ、STRING、VARCHAR、VARCHAR2 および CHARF です。

この文を外部 LOB ロケータ以外で使うと、エラーになります。

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] END-EXEC.
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

バッファ・データを、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 ロケータに対して事前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 キーワードを使って指定することができます。

FREE TEMPORARY

用途

LOB ロケータ用にテンポラリ領域を解放します。

構文

```
EXEC SQL [AT [:]database] LOB FREE TEMPORARY :src END-EXEC.
```

ホスト変数

src (IN OUT)

テンポラリ LOB をポイントする LOB ロケータ。

使用上の注意

入力ロケータは、テンポラリ 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 文で使われます。

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] END-EXEC.
```

ホスト変数

amt (IN)

ロードする最大バイト数。

file (IN OUT)

ソースの BFILE ロケータ。

src_offset (IN)

ファイルの先頭からのオフセット・バイト数。1 から指定できます。

dst (IN OUT)

宛先 LOB ロケータ。BLOB、CLOB および NCLOB が有効です。

dst_offset (IN)

書込みが開始する宛先 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）です。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 にコピーするときに、文字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 と同じ文字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、あらかじめ存在している必要があります。宛先の開始位置に既存のデータは、ソース・データで上書きされます。宛先の開始位置が現行データの最後を超えている場合は、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 および NCLOB）が宛先 LOB に書き込まれます。

宛先 LOB の現行の長さを超えて拡張される場合は、新しく書き込まれるデータを格納できるように宛先 LOB が拡張されます。4 ギガバイトを越えて 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 からコピーすると、エラーになります。

パラメータ量は、ロードする最大量です。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、エラーを発行せずに操作が終了します。

OPEN

用途

読込み、または読込み / 書込みアクセスとして、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 ] END-EXEC.
```

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 のデフォルト・モードでは、READ ONLY アクセスで OPEN されます。

内部 LOB の場合は、OPEN は LOB と関連付けられます。ロケータとは関連付けられません。すでに OPEN し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 したとは見なされません。そのかわり、両方のロケータが同じ LOB を参照します。BFILE の場合は、OPEN しているファイルはロケータと関連付けられます。

同時に最大 32 個の LOB を OPEN できます。33 個目の LOB を OPEN すると、エラーが返されます。

書込み可能な BFILES 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、エラーが返されます。

LOB を READ ONLY モードでオープンおよび書込みすると、エラーになります。

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] END-EXEC.
```

ホスト変数

amt (IN OUT)

入力は、読み込む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量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見なされます。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。

実際に読み込まれたバイト数または文字数は amt に返されます。データがピース単位で読み込まれた場合は、amt には、最後に読み込まれたピースの長さが常に格納されます。

LOB の最後に到達すると、「ORA-1403: データがありません」エラーが発行されます。

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-1403 エラーを捕捉するには、WHENEVER ディレクティブで NOT FOUND 条件を使ってポーリング・モードを制御してください。

src (IN)

LOB または BFILE ロケータ。

src_offset (IN)

読込みを開始した LOB 値の先頭からの絶対オフセット。文字 LOB では、LOB の先頭からの文字数です。バイナリ LOB または BFILE では、バイト数です。最初の位置は 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だけに制限されます。バッファの最大長は、LOB 値の格納に使われている外部データ型によって決まります。次の表は、有効な外部データ型、およびそのソース LOB 型ごとの最大長のリストです。

表 13-2 ソース LOB およびプリコンパイラのデータ型

外部 LOB[1]	内部 LOB	プリコンパイラの外部データ型	プリコンパイラの最大長[2]	PL/SQL データ型	PL/SQL 最大長
	BLOB	RAW	65535	RAW	32767
		VARRAW	65533		
		LONG RAW	2147483647		
BFILE		LONG VARRAW	2147483643		
	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

[1] これらの外部データ型は、BFILE で使えます。

[2] 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)

他の方法で長さがわからない場合は、指定したバッファの長さを指定します。

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、入力ロケータを使ってオープンされている必要があります。ファイルの読込み権限およびディレクトリの読込み権限も必要です。

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、次の方法で決まります。

- WITH LENGTH 句を指定した場合は、`buflen` で決まります。
- WITH LENGTH 句を指定しなかった場合は、4-1 ページの「文字データの処理」のルールによって、バッファ・ホスト変数が OUT モードで処理されて長さが決まります。

TRIM

用途

LOB 値を切り捨てます。

構文

```
EXEC SQL [AT [:]database] LOB TRIM :src TO :newlen END-EXEC.
```

ホスト変数

src (IN OUT)

内部 LOB 用の LOB ロケータ。

newlen (IN)

LOB 値の新規の長さ。

使用上の注意

この文は BFILES 用ではありません。新規の長さは、現行の長さより大きくできません。大きくした場合は、エラーが返されます。

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] END-EXEC.
```

ホスト変数

amt (IN OUT)

入力は、書き込む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。

ポーリング方法を使って書き込んだ場合は、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 データの書込み元のバッファ。データ型の長さについては、13-20 ページの「READ」を参照してください。

dst (IN OUT)

LOB ロケータ。

dst_offset (IN)

LOB の先頭からのオフセット（1 から始まります）。文字単位の場合は CLOB および NCLOB、バイト単位の場合は BLOB。

buflen (IN)

他の方法で計算できなかった場合のバッファ長。

使用上の注意

LOB データが既に存在する場合は、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 の最後と見なされます。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、1 ピースで LOB を書き込むか（デフォルトの ONE 方向変換を使います）、標準のポーリング方法を使ってピース単位で書き込むことができます。

FIRST を使っ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います。

ピース単位の書込みモードを使うと、各ピースのサイズ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することができます。

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。詳細は、13-20 ページの「READ」を参照してください。

DESCRIBE

用途

この文は、いくつかの OCI および PL/SQL 文と同等です（このため、最後に保存されます）。LOB DESCRIBE SQL 文を使って LOB から属性を取り出します。この機能は、OCI および PL/SQL プロシージャに類似しています。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 [{, attributeN}]
   INTO :hv1 [[INDICATOR] :hv_ind1] [{, :hvN [[INDICATOR] :hv_indN] }]
      END-EXEC.
```

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)

属性名リストで指定された順番で、属性値を受け取るホスト変数。

hv_ind1 ～ hv_indN (OUT)

属性名リストの順番で、NULL 状態の標識を受け取るオプションのホスト変数。

次の表では、関連する LOB の属性および読み込む必要のある COBOL 型について説明します。

表 13-3 LOB 属性

LOB 属性	属性の説明	制限	COBOL 型
CHUNKSIZE	LOB 値を格納する LOB チャンクで使われる領域の量（BLOB の場合はバイト単位、CLOB または NCLOB の場合は文字単位）。複数のチャンク・サイズを使って READ/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 をチャンク単位に行うと、不要または重複した世代管理が行われることがありません。同一 CHUNK に対して複数の WRITE コールを発行するかわりに、チャンクがいっぱいになるまで、WRITE を蓄積することができます。	BLOB、CLOB および NCLOB だけ	PIC S9(9) COMP
DIRECTORY	BFILE の場合は、DIRECTORY 別名。長さ n は、1 ～ 30 バイトです。実際の長さを使います。	FILE LOB だけ	PIC X(n) VARYING
FILEEXISTS	BFILE が、サーバーの OS ファイル・システム上に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 だけ	PIC S9(9) COMP
FILENAME	BFILE の名前。長さ n は、1 ～ 255 バイトです。実際の長さを使います。	FILE LOB だけ	PIC X(n) VARYING
ISOPEN	BFILE では、入力 BFILE ロケータが OPEN 文で使われなかった場合は、このロケータにより OPEN されていないものと見なされます。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って、同一 BFILE 上で複数の OPEN を実行することができます。LOB では、別のロケータを使って LOB を OPEN しても、LOB は入力ロケータによって OPEN されていると見なされます。ゼロ以外の場合は、ISOPEN は真で、ゼロの場合は、偽です。		PIC S9(9) COMP
ISTEMPORARY	入力 LOB ロケータがテンポラリ 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 は真で、ゼロの場合は、偽です。	BLOB、CLOB および NCLOB だけ	PIC S9(9) COMP
LENGTH	BLOB および BFILE の場合はバイト長。CLOB および NCLOB の場合は文字長。BFILE では、EOF が存在する場合は長さに含まれます。空の内部 LOB は、長さゼロです。初期化されていない LOB または BFILE の長さは定義されません。		PIC 9(9) COMP

使用上の注意

標識変数は、PIC S9(4) COMP として宣言しなければなりません。実行完了後、SQLERRD(3) には、エラーなしに取り出された属性の数が格納されます。実行エラーが発生した場合は、エラーが発生した属性は、SQLERRD(3) の内容より 1 だけ大きくなっています。

DESCRIBE の例

指定された BFILE から、DIRECTORY および FILENAME 属性を抽出する簡単な Pro*COBOL の例です。

```
...
 01  A-BFILE      SQL-BFILE.
 01  DIRECTORY    PIC X(30) VARYING.
 01  FILENAME     PIC X(30) VARYING.
 01  D-IND        PIC S9(4) COMP.
 01  F-IND        PIC S9(4) COMP.
 01  FEXISTS      PIC S9(9) COMP.
 01  ISOPN        PIC S9(9) COMP.
...
```

最後に、いくつかの LOB 表から BFILE ロケータを選択し、DESCRIBE します。

```
     EXEC SQL ALLOCATE :A-BFILE END-EXEC.
     EXEC SQL INSERT INTO lob_table (a_bfile) VALUES (BFILENAME ('lob.dir',
        'image.gif')) END-EXEC.
     EXEC SQL SELECT a_bfile INTO :A-BFILE FROM lob_table WHERE ... END-EXEC.
     EXEC SQL DESCRIBE :A-BFILE GET DIRECTORY, FILENAME, FILEEXISTS, ISOPEN
        INTO :DIRECTORY:D-IND, :FILENAME:F-IND, FEXISTS, ISOPN ND-EXEC.
```

標識変数は、DIRECTORY および FILENAME 属性を使ったときにだけ有効です。これらの属性は文字列なので、値が格納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、値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、標識の値には属性の元の長さが設定されます。

ポーリング・モードを使った READ および WRITE

ポーリング・モードで READ を行うときの例です。

最初の LOB READ で量にゼロを設定（または読み込まれる全データのサイズ量を設定）し、読込みポーリングを開始します。次の例では、この量にゼロが初期設定されます（詳細は省略してあります）。

```
      EXEC SQL ALLOCATE :CLOB1 END-EXEC.

      EXEC SQL WHENEVER NOT FOUND GOTO END-OF-CLOB END-EXEC.

      EXEC SQL SELECT A_CLOB INTO :CLOB1 FROM LOB_TABLE WHERE ... END-EXEC.

      MOVE 0 TO AMT.
      EXEC SQL LOB READ :AMT FROM :VLOB1 AT :OFFSET INTO :BUFFER END-EXEC.

 READ-LOOP.
      EXEC SQL LOB READ :AMT FROM :CLOB1 INTO BUFFER $END-EXEC.
```

```
     GO TO READ-LOOP.

END-OF-CLOB.
     EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

     EXEC SQL FREE :CLOB1 END-EXEC.
```

次のコードは、バッファから内部 CLOB へのデータの書込みの例です。初期書込み文の AMT（16 文字）値は、書き込むデータ全体の長さでなければなりません。バッファの長さは 5 文字です。

初期読込みで EOF が読み込まれた場合は、LOB WRITE ONE が行われます。EOF が読み込まれなかった場合は、バッファの LOB WRITE FIRST によってポーリングが開始されます。データが読み込まれ、出力として LOB WRITE NEXT が行われます。最後の書込みの後でデータが書き込まれるため、LOB WRITE NEXT にはオフセットは必要ありません。EOF が読み込まれると、読込みループが終了し、LOB WRITE LAST が行われます。戻された量は、量の初期値（16）と等しくなっていなければなりません。

```
    MOVE 16 TO AMT.
    PERFORM READ-NEXT-RECORD.
    MOVE INREC TO BUFFER-ARR.
    MOVE 5 TO BUFFER-LEN.
    IF (END-OF-FILE = "Y")
        EXEC SQL LOB WRITE ONE :AMT FROM :BUFFER INTO CLOB1
           AT :OFFSET END-EXEC.
        PERFORM DISPLAY-CLOB
    ELSE
        EXEC SQL LOB WRITE FIRST :AMT FROM :BUFFER INTO :CLOB1
           AT :OFFSET END-EXEC.
    PERFORM READ-NEXT-RECORD.
    PERFORM WRITE-TO-CLOB
       UNTIL END-OF-FILE = "Y".
    MOVE INREC TO BUFFER-ARR.
    MOVE 1 TO BUFFER-LEN.
    EXEC SQL LOB WRITE LAST :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM DISPLAY-CLOB.
    ...
WRITE-TO-CLOB.
    MOVE INREC TO BUFFER-ARR.
    MOVE 5 TO BUFFER-LEN.
    EXEC SQL LOB WRITE NEXT :AMT FROM :BUFFER INTO :CLOB1 END-EXEC.
    PERFORM READ-NEXT RECORD.

READ-NEXT-RECORD.
    MOVE SPACES TO INREC.
    READ INFILE NEXT RECORD
       AT END
```

```
        MOVE "Y" TO END-OF-FILE.
...
```

LOB サンプル・プログラム : LOB DEMO1.PCO

次のプログラム LOBDEMO1.PCO は、いくつかの埋込み SQL 文の例です。ソース・コードは、demo ディレクトリ内にあります。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 という名前の表を使います。標準的な自動車部門の簡単な SQL 操作を、いくつかモデル化します。

発生する可能性のアクションを示します。

- 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。

```
*********************************************************************
* LOB Demo 1: DMV Database                                          *
*                                                                   *
* 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/her name and the   *
* text summary of the info found in his license.                    *
*                                                                   *
* The sss number and the name are the unique social security number *
* and name of an individual.  The text summary is a summary of the  *
* information on the individual, 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*COBOL client can handle the    *
* new LOB datatypes.  Demonstrated are the mechanisms for accessing *
* and storing lobs to/from tables.                                  *
*                                                                   *
*  To run the demo:                                                 *
*                                                                   *
*  1. Execute the script, lobdemo1.sql in Server Manager            *
*  2. Precompile using Pro*COBOL                                    *
*       procob lobdemo1                                             *
```

```
*  3. Compile/Link (This step is platform specific)                *
*                                                                  *
* lobdemo1.sql contains the following SQL statements:              *
*                                                                  *
* connect scott/tiger;                                             *
*                                                                  *
* drop table license_table;                                        *
*                                                                  *
* 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');                               *
*                                                                  *
* insert into license_table                                        *
*  values('555001212', 'Eight H. Number',                          *
*  'Driving Under the Influence');                                 *
*                                                                  *
* insert into license_table                                        *
*  values('010101010', 'P. Doughboy',                              *
*  'Impersonating An Oracle Employee');                            *
*                                                                  *
* insert into license_table                                        *
*  values('555377012', 'Calvin N. Hobbes',                         *
*  'Driving Under the Influence');                                 *
*                                                                  *
* The main program provides the menu of actions that can be        *
* performed.  The program stops when the number 5 (Quit) option    *
* is entered.  Depending on the input, this main program calls     *
* the appropriate nested program to execute the chosen action.     *
*                                                                  *
*******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  LOBDEMO1.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  USERNAME       PIC X(5) VARYING.
 01  PASSWD         PIC X(5) VARYING.
 01  CHOICE         PIC 9.
 01  SSS            PIC X(9) GLOBAL.
 01  SSSEXISTS      PIC 9 VALUE ZERO GLOBAL.
 01  LICENSE-TXT    SQL-CLOB GLOBAL.
 01  NEWCRIME       PIC X(35) VARYING GLOBAL.
```

```
    EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE DIVISION.
BEGIN-PGM.
    EXEC SQL
        WHENEVER SQLERROR GO TO SQLERROR
    END-EXEC.
    PERFORM LOGON.

MAIN.
    DISPLAY '**************************************************'.
    DISPLAY '*            Welcome to the DMV Database         *'.
    DISPLAY '**************************************************'.

MENU.
    DISPLAY " ".
    DISPLAY "License Options:".
    DISPLAY "1. List available records by SSS number".
    DISPLAY "2. Get information on a particular record".
    DISPLAY "3. Add crime to a record".
    DISPLAY "4. Insert new record to database".
    DISPLAY "5. Quit".
    DISPLAY " ".
    DISPLAY "Your Selection (1-5)? " WITH NO ADVANCING.

 ENTER-CHOICE.
    ACCEPT CHOICE.
    DISPLAY " ".
    IF (CHOICE > 5 OR CHOICE < 1)
       DISPLAY "Invalid selection"
       DISPLAY "Please enter one of the given options: "
       GO TO ENTER-CHOICE
    ELSE IF (CHOICE = 5)
       GO TO FINISHED.

    IF (CHOICE = 1)
       CALL "LIST-RECORDS".
    IF (CHOICE = 2)
        CALL "GET-RECORD".
    IF (CHOICE = 3)
        CALL "ADD-CRIME".
    IF (CHOICE = 4)
        CALL "NEW-RECORD".

    GO TO MENU.

 FINISHED.
```

```
     EXEC SQL ROLLBACK RELEASE END-EXEC.
     DISPLAY " ".
     DISPLAY "HAVE A GOOD DAY!".
     DISPLAY " ".
     STOP RUN.

 LOGON.
     MOVE "scott" TO USERNAME-ARR.
     MOVE 5 TO USERNAME-LEN.
     MOVE "tiger" TO PASSWD-ARR.
     MOVE 5 TO PASSWD-LEN.
     EXEC SQL
        CONNECT :USERNAME IDENTIFIED BY :PASSWD
     END-EXEC.
     DISPLAY " ".
     DISPLAY "Connecting to license database account:  ",
        USERNAME-ARR, "/", PASSWD-ARR.
     DISPLAY " ".

  SQLERROR.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.
     EXEC SQL ROLLBACK RELEASE END-EXEC.
     STOP RUN.

*******************************************************************
*  LIST-RECORDS
*    Lists available records by sss number.
*
*******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  LIST-RECORDS.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  SELECT-SSS     PIC X(50) VARYING.
 01  SSS            PIC X(9).
 PROCEDURE DIVISION.

     MOVE "SELECT SSS FROM LICENSE_TABLE"
         TO SELECT-SSS-ARR.
     MOVE 29 TO SELECT-SSS-len.

     EXEC SQL PREPARE SSS_EXEC FROM :SELECT-SSS END-EXEC.
```

```
    EXEC SQL DECLARE SSS_CURSOR CURSOR FOR SSS_EXEC END-EXEC.

    EXEC SQL OPEN SSS_CURSOR END-EXEC.

    DISPLAY "Available records:".

    EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND END-EXEC.

GETROWS.
     EXEC SQL FETCH SSS_CURSOR INTO :SSS END-EXEC.
     DISPLAY SSS.
     GO TO GETROWS.

NOTFOUND.
    EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
    EXEC SQL CLOSE SSS_CURSOR END-EXEC.
    GO TO END-PROGRAM.

 SQLERROR.
    DISPLAY " ".
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.

END-PROGRAM.
END PROGRAM LIST-RECORDS.

*****************************************************************
*  GETSSS
*    Fills the global variable SSS with the client-supplied sss.
*    Sets the global variable SSSEXISTS to 0 if the sss does not
*    correspond to any entry in the database, else sets it to 1.
*****************************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID.  GETSSS COMMON.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SSSCOUNT       PIC S9(4) COMP.
PROCEDURE DIVISION.

    DISPLAY "Social Security Number? " WITH NO ADVANCING.
    ACCEPT SSS.
    DISPLAY " ".

    EXEC SQL SELECT COUNT(*) INTO :SSSCOUNT FROM LICENSE_TABLE
        WHERE SSS = :SSS END-EXEC.
```

```
     IF (SSSCOUNT = 0)
        MOVE 0 TO SSSEXISTS
     ELSE
        MOVE 1 TO SSSEXISTS.
     GO TO END-PROGRAM.

  SQLERROR.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.

 END-PROGRAM.
 END PROGRAM GETSSS.

*****************************************************************
*  PRINTCRIME
*    Obtains the length of the global clob LICENSE-TXT and
*    uses that in the LOB READ statement to read the clob
*    into a character buffer to display the contents of the clob.
*    The caller to this function must allocate, select and later
*    free the global clob LICENSE-TXT.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  PRINTCRIME COMMON.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  THE-STRING    PIC X(200) VARYING.
 01  TXT-LENGTH    PIC S9(9) COMP.
 PROCEDURE DIVISION.

     DISPLAY "====================".
     DISPLAY " CRIME SHEET SUMMARY ".
     DISPLAY "====================".

     MOVE SPACE TO THE-STRING-ARR.
     EXEC SQL LOB DESCRIBE :LICENSE-TXT GET LENGTH
         INTO :TXT-LENGTH END-EXEC.

     IF (TXT-LENGTH = 0)
        DISPLAY "Record is clean"
      ELSE
        EXEC SQL LOB READ :TXT-LENGTH FROM :LICENSE-TXT
           INTO :THE-STRING END-EXEC
        DISPLAY THE-STRING-ARR.

     GO TO END-PROGRAM.
```

```
  SQLERROR.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.

 END-PROGRAM.
 END PROGRAM PRINTCRIME.

*****************************************************************
*  GET-RECORD
*    Allocates the global clob LICENSE-TXT then selects
*    the name and text what corresponds to the client-supplied
*    sss.  It then calls PRINTCRIME to print the information and
*    frees the clob.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  GET-RECORD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  NAME1          PIC X(50) VARYING.
 PROCEDURE DIVISION.

     CALL "GETSSS".
     IF (SSSEXISTS = 1)
       EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
       EXEC SQL SELECT NAME, TXT_SUMMARY
            INTO :NAME1, :LICENSE-TXT FROM LICENSE_TABLE
            WHERE SSS = :SSS END-EXEC
       DISPLAY "==================================================
-         "========================"
       DISPLAY " "
       DISPLAY "NAME:  ", NAME1-ARR, "SSS:  ", SSS
       DISPLAY " "
       CALL "PRINTCRIME"
       DISPLAY " "
       DISPLAY "==================================================
-         "========================"
       EXEC SQL FREE :LICENSE-TXT END-EXEC
     ELSE
        DISPLAY "SSS Number Not Found".

     GO TO END-PROGRAM.

  SQLERROR.
```

```
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.
     GO TO END-PROGRAM.

 END-PROGRAM.
 END PROGRAM GET-RECORD.

*******************************************************************
*  GETNEWCRIME
*    Provides a list of the possible crimes to the user and
*    stores the user's correct response in the global variable
*    NEWCRIME.
*******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  GETNEWCRIME COMMON.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CRIMES.
     05 FILLER PIC X(35) VALUE "Driving Under the Influence".
     05 FILLER PIC X(35) VALUE "Grand Theft Auto".
     05 FILLER PIC X(35) VALUE "Driving Without a License".
     05 FILLER PIC X(35) VALUE
            "Impersonating an Oracle Employee".
     05 FILLER PIC X(35) VALUE "Wearing a Bright Orange Shirt".
 01  CRIMELIST REDEFINES CRIMES.
     05 CRIME  PIC X(35) OCCURS 5 TIMES.
 01  CRIME-INDEX   PIC 9.
 01  CHOICE        PIC 9.
 PROCEDURE DIVISION.

     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

 CRIMEMENU.
     DISPLAY " ".
     DISPLAY "Select from the following:".
     PERFORM DISPLAY-CRIME
        VARYING CRIME-INDEX FROM 1 BY 1
        UNTIL CRIME-INDEX > 5.
     DISPLAY "Crime (1-5) = " WITH NO ADVANCING.

  ENTER-CHOICE.
     ACCEPT CHOICE.
     DISPLAY " ".
     IF (CHOICE > 5 OR CHOICE < 1)
        DISPLAY "Invalid selection"
```

```
        DISPLAY "Crime (1-5) = " WITH NO ADVANCING
        GO TO ENTER-CHOICE.

      MOVE CRIME(CHOICE) TO NEWCRIME-ARR.
      MOVE 35 TO NEWCRIME-LEN.
      GO TO END-PROGRAM.

 DISPLAY-CRIME.
     DISPLAY "(", CRIME-INDEX, ") ", CRIME(CRIME-INDEX).

 END-PROGRAM.
 END PROGRAM GETNEWCRIME.

*****************************************************************
* APPENDTOCLOB
*    Obtains the length of the global clob LICENSE-TXT and
*    uses that in the LOB WRITE statement to append the NEWCRIME
*    character buffer to the global clob LICENSE-TXT.
*    The name corresponding the global SSS is then selected
*    and displayed to the screen along with value of LICENSE-TXT.
*    The caller to this function must allocate, select and later
*    free the global clob LICENSE-TXT.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  APPENDTOCLOB COMMON.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  TXT-LEN       PIC S9(9) COMP.
 01  CRIME-LEN     PIC S9(9) COMP.
 01  NAME1         PIC X(50) VARYING.
 PROCEDURE DIVISION.

     EXEC SQL
         WHENEVER SQLERROR GO TO SQLERROR
     END-EXEC.

     EXEC SQL LOB DESCRIBE :LICENSE-TXT GET LENGTH
         INTO :TXT-LEN END-EXEC.

     MOVE NEWCRIME-LEN TO CRIME-LEN.
     IF (TXT-LEN NOT = 0)
       COMPUTE TXT-LEN = TXT-LEN + 3
     ELSE
       COMPUTE TXT-LEN = TXT-LEN + 1.
     EXEC SQL LOB WRITE :CRIME-LEN FROM :NEWCRIME
           INTO :LICENSE-TXT AT :TXT-LEN END-EXEC.
```

```
     EXEC SQL SELECT NAME INTO :NAME1 FROM LICENSE_TABLE
         WHERE SSS = :SSS END-EXEC
     DISPLAY " "
     DISPLAY "NAME:  ", NAME1-ARR, "SSS:  ", SSS
     DISPLAY " "
     CALL "PRINTCRIME"
     DISPLAY " "
     GO TO END-PROGRAM.

  SQLERROR.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.

 END-PROGRAM.
 END PROGRAM APPENDTOCLOB.

*****************************************************************
* ADD-CRIME
*    Obtains a sss and crime from the user and appends
*    the crime to the list of crimes of the corresponding sss.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  ADD-CRIME.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 PROCEDURE DIVISION.

     EXEC SQL
         WHENEVER SQLERROR GO TO SQLERROR
     END-EXEC.

     CALL "GETSSS".
     IF (SSSEXISTS = 1)
       EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
       CALL "GETNEWCRIME"
       EXEC SQL SELECT TXT_SUMMARY INTO :LICENSE-TXT
            FROM LICENSE_TABLE WHERE SSS = :SSS
            FOR UPDATE END-EXEC
       CALL "APPENDTOCLOB"
       EXEC SQL FREE :LICENSE-TXT END-EXEC
     ELSE
        DISPLAY "SSS Number Not Found".

     GO TO END-PROGRAM.
```

```
  SQLERROR.
     DISPLAY " ".
     DISPLAY "ORACLE ERROR DETECTED:".
     DISPLAY " ".
     DISPLAY SQLERRMC.

 END-PROGRAM.
 END PROGRAM ADD-CRIME.

*****************************************************************
* NEW-RECORD
*    Obtains the sss and name of a new record and inserts them
*    along with an empty_clob() for the clob in the table.
*****************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  NEW-RECORD.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  NEWNAME         PIC X(50).
 PROCEDURE DIVISION.

     CALL "GETSSS".
     IF (SSSEXISTS = 1)
       DISPLAY "Record with that sss number already exists"
     ELSE
       DISPLAY "Name? " WITH NO ADVANCING
       ACCEPT NEWNAME
       DISPLAY " ".
       EXEC SQL ALLOCATE :LICENSE-TXT END-EXEC
       EXEC SQL INSERT INTO LICENSE_TABLE
            VALUES (:SSS, :NEWNAME, EMPTY_CLOB()) END-EXEC
       EXEC SQL SELECT TXT_SUMMARY INTO :LICENSE-TXT
            FROM LICENSE_TABLE WHERE SSS = :SSS END-EXEC
       DISPLAY "==================================================
-         "========================"
       DISPLAY "NAME: ", NEWNAME,"SSS: ", SSS
       CALL "PRINTCRIME"
       DISPLAY "==================================================
-         "========================"
       EXEC SQL FREE :LICENSE-TXT END-EXEC.

     GO TO END-PROGRAM.

  SQLERROR.
     DISPLAY " ".
```

```
    DISPLAY "ORACLE ERROR DETECTED:".
    DISPLAY " ".
    DISPLAY SQLERRMC.

END-PROGRAM.
END PROGRAM NEW-RECORD.
END PROGRAM  LOBDEMO1.
```

14

プリコンパイラのオプション

この章では Pro*COBOL のプリコンパイラ・オプションを説明します。内容は次のとおりです。

- Pro*COBOL コマンド
- プリコンパイル時の処理
- オプションについて
- マクロ・オプションとマイクロ・オプション
- オプションの入力
- オプションの有効範囲
- クイック・リファレンス
- Pro*COBOL プリコンパイラ・オプションの使用

Pro*COBOL コマンド

Oracle Pro*COBOL プリコンパイラを実行するには、次のコマンドを使用します。

```
procob [option_name=value] [option_name=value] ...
```

オプション名とオプション値の間には必ず等号を置きます。等号の前後には空白を入れないでください。

Pro*COBOL の位置はシステムによって異なります。通常は、システム管理者か DBA が、環境変数、論理または別名を定義するか、あるいはオペレーティング・システム固有のその他の方法を使用して、Pro*COBOL の実行可能ファイルをアクセス可能にします。

たとえば、プリコンパイルするソース・ファイルの指定には、INAME オプションを使用します。次のコマンドでは、

```
procob INAME=test
```

Pro*COBOL はファイル拡張子を *.pco* とみなすため、現行ディレクトリ内のファイル *test.pco* がプリコンパイルされます。

このように、INAME の指定でファイル拡張子を使用する必要はありません（ただし、ファイル拡張子が標準以外の場合には拡張子を指定する必要があります）。

入力ファイル名と出力ファイル名を、それぞれのオプション名 INAME および ONAME とともに指定する必要はありません。オプション名を指定しなければ、Pro*COBOL はコマンド行に指定された最初のファイル名を入力ファイル名とみなし、2 番目のファイル名を出力ファイル名とみなします。

したがって、

```
procob MODE=ANSI myfile myfile.cob
```

は次のオプションに相当します。

```
procob MODE=ANSI INAME=myfile.pco ONAME=myfile.cob
```

大 / 小文字の区別

一般に、コマンド行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でも使用できます。ただし、大文字と小文字を区別するオペレーティング・システム（UNIX など）を使用している場合は、Pro*COBOL の実行可能ファイルの名前も含めて、ファイル名は大文字と小文字を正しく組み合わせて指定してください。

> **注意 :** ファイル名などのように特定のオペレーティング・システムを指定しないオプション名とオプション値では、大文字と小文字は区別されません。このマニュアル中の例では、オプション名は大文字または小文字で記述し、オプション値は通常は小文字で記述しています。ファイル名を指定する場合は、Pro*COBOL の実行可能ファイルも含

めて、それを実行するオペレーティング・システムの大 / 小文字区別の規則に従ってください。

UNIX C シェルなどの一部のオペレーティングおよびユーザー・シェルでは、"?" の前にバックスラッシュ（\）などの "escape" 文字を指定する必要があります。たとえば、Pro*COBOL のオプション設定を示すには、"procob?," のかわりに "procob \?" を使う必要があります。

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。

プリコンパイル時の処理

Pro*COBOL は、プリコンパイル中に、ホスト・プログラムに埋め込まれた SQL 文に置き換わる COBOL のコードを生成します。生成されたコードには、各ホスト変数のデータ型、長さ、アドレスなどデータ構造のほか、Oracle ランタイム・ライブラリ SQLLIB が必要とするその他の情報が組み込まれています。またこのコードには、埋込み SQL の処理を実行する SQLLIB ルーチンのコールも入っています。

注意 : Pro*COBOL は、Oracle コール・インタフェース（OCI）ルーチンへのコールを生成しません。

Pro*COBOL は警告やエラー・メッセージを発行することがあります。これらのメッセージは、『Oracle8*i* エラー・メッセージ』で説明します。

オプションについて

プリコンパイル時には数多くの便利なオプションを使用できます。それらを使って、リソースの使用、エラーの報告、入出力の書式化、カーソルの管理を制御できます。

オプションの値はリテラルであり、テキストまたは数値を表します。たとえば次のオプションでは、

```
... INAME=my_test
```

値はファイル名を指定する文字列リテラルです。

次のオプションでは、

```
... MAXOPENCURSORS=20
```

値は数値です。

オプションの中にはブール値をとるものもあります。ブール値には、文字列 YES または NO、TRUE または FALSE、整数リテラル 1 または 0 を指定できます。たとえばオプション

```
... SELECT_ERROR=YES
```

は次のオプションに相当します。

```
... SELECT_ERROR=TRUE
```

または

```
... SELECT_ERROR=1
```

空白によって各オプションを区切るため、等号の前後には空白を入れないでください。たとえば、次のように、コマンド行にオプション AUTO_CONNECT を指定できます。

```
... AUTO_CONNECT=YES
```

オプション名には略称を使用できますが、他と区別がつかなくなる略称は使用しないでください。たとえば、MAX という略称は MAXLITERAL と MAXOPENCURSORS のどちらを表すのかわからないため、使用できません。

Pro*COBOL オプションの簡単な参照がオンラインで利用できます。オンライン表示するには、オペレーティング・システムのプロンプトから、引数を指定しないで Pro*COBOL コマンドを入力します。

```
procob
```

オンライン画面には、各オプションの名前および構文、デフォルト値、用途が表示されます。アスタリスク（*）が付いたオプションは、コマンド行でもインラインでも指定できます。

オプション値の優先順位

オプションの値は、次の値によって決まります（下にいくほど優先順位が高くなります）。

- Pro*COBOL に組み込まれているデフォルト値
- システム構成ファイルに設定されている値
- ユーザー構成ファイルに設定されている値
- コマンド行に入力された値
- インライン指定で設定された値

たとえば、オプション MAXOPENCURSORS 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。Pro*COBOL に組み込まれているこのオプションのデフォルト値は、10 です。システム構成ファイルに MAXOPENCURSORS=32 が指定されている場合は、値は 32 になります。ユーザー構成ファイルの設定値は変更できます。変更すると、システム構成値が上書きされます。

さらに、このオプションをコマンド行で設定した場合には、新しいコマンド行の値が優先されます。最終的には、インライン指定が上記のすべてのデフォルト値よりも優先されます。詳細は、14-6 ページの「構成ファイル」および 14-7 ページの「オプションの入力」を参照してください。

マクロ・オプションとマイクロ・オプション

Pro*COBOL には、DBMS および MODE という 2 つのオプションが用意されています。この 2 つのオプションは、リリース 8.0 以前からあるオプションで、複数の機能を一度に制御します。このため、マクロ・オプションと呼ばれます。新しいオプションの中には、END_OF_FETCH のように、1 つの機能だけを制御するものがあります。このようなオプションをマイクロ・オプションと呼びます。マクロ・オプションおよびマイクロ・オプションを設定する際 14-4 ページの「オプション値の優先順位」の項に記載されているように、マクロ・オプションの優先順位がマイクロ・オプションより高い場合にかぎり、マクロ・オプションが優先されることに注意してください。この点は、リリース 8.0 より前の Pro*COBOL とは動作が異なります。

たとえば、MODE のデフォルトは ORACLE です。また、END_OF_FETCH のデフォルトは 1403 です。ユーザー構成ファイルに MODE=ANSI を指定した場合は、Pro*COBOL からフェッチの最後に値 100 が戻され、END_OF_FETCH のデフォルト値 1403 が上書きされます。また、構成ファイルで MODE=ANSI と指定し、コマンド行で END_OF_FETCH=1403 と指定した場合も、1403 が戻されます。

次の表に、マクロ・オプションの値によって設定されるマイクロ・オプションの値を示します。

表 14-1 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

マクロ・オプション	マイクロ・オプション
MODE=ANSI | ISO	CLOSE_ON_COMMIT=YES
	DECLARE_SECTION=YES
	END_OF_FETCH=100
	DYNAMIC=ANSI
	TYPE_CODE=ANSI
MODE=ANSI14 | ANSI13 | ISO14 | ISO13	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=YES
	END_OF_FETCH=100
MODE=ORACLE	CLOSE_ON_COMMIT=NO
	DECLARE_SECTION=NO
	END_OF_FETCH=1403
	DYNAMIC=ORACLE
	TYPE_CODE=ORACLE
DBMS=NATIVE | V7 | V8	UNSAFE_NULL=NO

現在の設定値の決定

コマンド行で疑問符（?）を使うと、1 つ以上のオプションの現在の設定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すると、

```
procob ?
```

すべてのオプションの設定とその現在の設定値が端末に表示されます。この場合、表示される値は Pro*COBOL に組み込まれている値ですが、システム構成ファイルに値が指定されている場合は構成ファイル内の値が表示されます。一方、次のコマンドを発行した場合には、

```
procob CONFIG=my_config_file.cfg ?
```

現行のディレクトリに *my_config_file.cfg* という名前のファイルがあると、*my_config_file.cfg* ファイルに指定されているオプションが、その他のデフォルト値とともに表示されます。ユーザー構成ファイル内で指定された値はここでは表示されません。また、ユーザー構成ファイルに指定された値は、Pro*COBOL に組み込まれている値やシステム構成ファイルに指定されている値より優先されます。

次のようにオプション名の後に "=?" を指定して、シングル・オプションの現在の設定値を指定することもできます。

```
procob MAXOPENCURSORS=?
```

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。ファイルのそれぞれのレコード（行）には、1 つのオプションおよび対応付けられた値 （値は複数の場合もある）が入ります。たとえば、構成ファイルに次のような行が入っているとします。

```
FIPS=YES
MODE=ANSI
```

これらの行によって FIPS オプションと MODE オプションの値が設定されています。

システムにはそれぞれ 1 つのシステム構成ファイルがあります。システム構成ファイルの名前は次のとおりです。

```
pcbcfg.cfg
```

システム構成ファイルの位置はオペレーティング・システムによって異なります。ほとんどの UNIX システムでは、Pro*COBOL 構成ファイルは通常、$ORACLE_HOME/precomp/admin ディレクトリにあります（$ORACLE_HOME はデータベース・ソフトウェアの環境変数です）。

リリース 8.0 より前の Pro*COBOL では、構成ファイル名は pccob.cfg であることを注意してください。

Pro*COBOL のユーザーは 1 つ以上のユーザー構成ファイルを持つことができます。ユーザー構成ファイルの名前は CONFIG オプションを使ってコマンド行で指定します。詳細は 14-6 ページの「現在の設定値の決定」を参照してください。

注意 : 構成ファイルはネストできません。つまり、CONFIG は構成ファイル内では有効なオプションではありません。

オプションの入力

Pro*COBOL のオプションはすべて、コマンド行または構成ファイル（CONFIG を除いて）から入力できます。また、インラインで入力できるオプションも多数あります。
Pro*COBOL は実行時に、これら 3 つのソースのオプションをすべて受け入れます。コマンド行については、既に説明してありました。

コマンド行

次の構文を使って、コマンド行からプリコンパイラ・オプションを入力できます。

```
[option_name=value] [option_name=value] ...
```

オプションとオプションの間は、1 つ以上の空白で区切ります。たとえば、次のようにオプションを入力できます。

```
... ERRORS=no LTYPE=short
```

インライン

次の構文を使って 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) END-EXEC.
```

たとえば、次のような文をコーディングできます。

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES) END-EXEC.
```

インラインでオプションを入力すると、コマンド行から入力された同じオプションは無効になります。

長所

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、文ごとに HOLD_CURSOR 値および RELEASE_CURSOR 値を変更することがあります。この場合、付録 D の「パフォーマンス・チューニング」を参照してください。実行時パフォーマンスを最適化するための、インライン・オプションの使用方法が記載されています。

インラインでのオプションの指定は、使用しているオペレーティング・システムでコマンド行に入力できる文字数に制限がある場合にも便利です。また、インライン・オプションは構成ファイルに格納できます。これについては次の項で説明します。

EXEC ORACLE の有効範囲

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。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  EMP-NAME    PIC X(20) VARYING.
 01  EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
 01  SALARY      PIC S9(5)V99 COMP-3 VALUE ZERO.
 01  DEPT-NUMBER PIC S9(4) COMP VALUE ZERO.
     EXEC SQL END DECLARE SECTION END-EXEC.
     ...
     EXEC SQL WHENEVER NOT FOUND GOTO NO-MORE END-EXEC.
     ...
     EXEC ORACLE OPTION (HOLD_CURSOR=NO)END-EXEC.
     ...
     EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
          SELECT EMPNO, DEPTNO FROM EMP
     END-EXEC.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.

     DISPLAY 'Employee Number  Dept'.
     DISPLAY '---------------  ----'.
     PERFORM
         EXEC SQL
           FETCH EMP-CURSOR INTO :EMP-NUMBER, :DEPT-NUMBER
         END-EXEC
         DISPLAY EMP-NUMBER, DEPT-NUMBER END-EXEC
     END-PERFORM.

 NO-MORE.
         EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
     PERFORM
         DISPLAY 'Employee number? '
         ACCEPT EMP-NUMBER
         IF EMP-NUMBER IS NOT = 0
             EXEC ORACLE OPTION (HOLD_CURSOR=YES) END-EXEC
             EXEC SQL SELECT ENAME, SAL
                 INTO :EMP-NAME, :SALARY
                 FROM EMP
                 WHERE EMPNO = :EMP-NUMBER
                 DISPLAY 'Salary for ', EMP-NAME, ' is ', SALARY
```

```
            END-EXEC
        END-IF
    END-PERFORM.
NEXT-PARA.
    ...
```

オプションの有効範囲

プリコンパイル単位は、COBOL コードおよび 1 つ以上の埋込み SQL 文が入っている 1 つのファイルです。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・ユニットにだけ効力を持ちます。

たとえば、単位 A に HOLD_CURSOR=YES および RELEASE_CURSOR=YES を指定し、単位 B には指定しなかった場合、単位 A の SQL 文は指定した HOLD_CURSOR および RELEASE_CURSOR の値で実行されますが、単位 B の SQL 文はデフォルト値で実行されます。ただし、Oracle に接続した時点で有効だった MAXOPENCURSORS の設定は、その接続を切り離すまでずっと有効です。

インライン・オプションの有効範囲は論理的なものではなく、位置的なものです。つまり、インライン・オプションの影響を受けるのは、プログラム論理の流れの中でそのインライン・オプションの後にくる SQL 文ではなく、ソース・ファイル内でそのインライン・オプションの後に記述されている SQL 文です。オプションの設定は、そのオプションを再指定しないかぎり、ファイルの終りまで有効です。

クイック・リファレンス

表 14-2 は、Pro*COBOL オプションの早見表を示しています。アスタリスクが付いているオプションは、インラインで入力できます。

また、オンラインでも簡単な参照ができます。オンライン画面で Pro*COBOL オプションを表示するには、オペレーティング・システムのプロンプトに、オプションを指定せずに Pro*COBOL コマンドを入力します。オンライン画面には、各オプションの名前および構文、デフォルト値、用途が表示されます。

> **注意 :** プラットフォーム固有のオプションもいくつかあります。たとえば、バイトスワップ・プラットフォームでは、オプション COMP5 によって COMPUTATIONAL 項目の使用を管理します。使用しているシステム固有の Oracle マニュアルを参照してください。

表 14-2 オプション・リスト

構文	デフォルト値	指定値
ASACC={YES | NO}	NO	YES の場合、リスト・ファイルに ASA キャリッジ制御を使用します。

表 14-2 オプション・リスト (続き)

構文	デフォルト値	指定値
ASSUME_SQLCODE={YES | NO}	NO	YES の場合、SQLCODE 変数が存在するものとみなします。
AUTO_CONNECT={YES | NO}	NO	YES の場合、OPS$ アカウントへの自動接続を行います。
CLOSE_ON_COMMIT	NO	YES の場合、COMMIT 時にすべてのカーソルをクローズします。
CONFIG=*filename*		ユーザー定義構成ファイルの名前を指定します。
DATE_FORMAT	LOCAL	日付文字列書式を指定します。
DBMS={NATIVE | V7 | V8}	NATIVE	プリコンパイル時の Oracle のバージョン固有の動作。
DECLARE_SECTION	NO	YES の場合、DECLARE SECTION が必須となります。
DEFINE=*symbol* *		条件付きプリコンパイルで使用する記号を定義します。
DYNAMIC	ORACLE	SQL 方法 4 で、Oracle または ANSI 動的セマンティックスを指定します。
END_OF_FETCH	1403	フェッチ終了時の SQLCODE の値。
ERRORS={YES | NO} *	YES	YES の場合、端末にエラーを表示します。
FIPS={YES | NO}	NO	YES の場合、ANSI/ISO の拡張要素にフラグを付けます。
FORMAT={ANSI | TERMINAL}	ANSI	入力ファイルの COBOL 文の書式。
HOLD_CURSOR={YES | NO}*	NO	YES の場合、Oracle カーソルを保持します（再割当てしません）。
HOST={COBOL | COB74}	COBOL	入力ファイルで使用する COBOL のバージョン（COBOL 85 または COBOL 74）。
[INAME=]*filename*		入力ファイルの名前。
INCLUDE=*path**		EXEC SQL INCLUDE ファイルのパス名。
IRECLEN=*integer*	80	入力ファイルのレコード長。
LITDELIM={APOST | QUOTE}	QUOTE	COBOL 文字列のデリミタ。

表 14-2 オプション・リスト (続き)

構文	デフォルト値	指定値
LNAME=*filename*		リスト・ファイルの名前。
LRECLEN=*integer*	132	リスト・ファイルのレコード長。
LTYPE={LONG | SHORT | NONE} *	LONG	リスト・ファイルの型。
MAXLITERAL=*integer**	256	文字列の最大長。
MAXOPENCURSORS=*integer* *	10	キャッシュされる Oracle カーソルの最大数（1)。
MODE={ORACLE | ANSI}	ORACLE	ANSI の場合、ANSI/ISO SQL 規格に準拠します。
NESTED={YES | NO}	YES	YES の場合は、ネストしたプログラムがサポートされます。
NLS_LOCAL={YES | NO}	NO	YES の場合、Pro*COBOL の旧リリースの NCHAR 方法を使用します。
[ONAME=]*filename*	*iname*.cob	出力ファイルの名前。
ORACA={YES | NO}*	NO	YES の場合、ORACA 通信領域を使用します。
ORECLEN=*integer*	80	出力ファイルのレコード長。
PAGELEN=*integer*	66	リスト・ファイルの 1 ページ当たりの行数。
PICX	CHARF	PIC X COBOL 変数のデータ型。
PREFETCH	1	一定数の行をプリフェッチして、クエリーを高速化します。
RELEASE_CURSOR={YES | NO} *	NO	YES の場合、実行後に Oracle カーソルを解放します。
SELECT_ERROR={YES | NO}*	YES	YES の場合、SELECT 時に FOUND エラーが発生します。
SQLCHECK={SEMANTICS | SYNTAX}*	SYNTAX	SQL チェックのレベル。
TYPE_CODE	ORACLE	動的 SQL 方法 4 の場合は、Oracle または ANSI 型コードを使います。
UNSAFE_NULL={YES | NO}	NO	YES の場合、安全でない NULL のフェッチが可能（ORA-01405 メッセージが発行されなくなります)。

表 14-2 オプション・リスト (続き)

構文	デフォルト値	指定値
USERID=*username / password[@dbname]*		Oracle のユーザー名、パスワードおよびオプショナル・データベース。
VARCHAR={YES | NO}	NO	YES の場合、ユーザー定義の VARCHAR グループ項目を受け入れます。
XREF={YES | NO}*	YES	YES の場合、リスト・ファイル内に記号のクロス・リファレンスを生成します。

Pro*COBOL プリコンパイラ・オプションの使用

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。Pro*COBOL プリコンパイラ・オプションをアルファベット順に示し、各オプションごとに用途、構文、デフォルト値を記載してあります。さらに「使用上の注意」の欄で、オプションの働きを説明します。使用上の注意に特に記載がない場合、そのオプションはコマンド行、インライン、構成ファイルのどの方法でも入力できます。

ASACC

用途

リスト・ファイルが ASA 規則に従って、キャリッジ制御のために各行の最初の桁を使用するかどうかを指定します。

構文

ASACC={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。

ASSUME_SQLCODE

用途

SQLCODE がプログラムで宣言されているかどうか、また、型が正しいかどうかに関係なく、SQLCODE が宣言されているものとみなすように Pro*COBOL に指示します。

構文

ASSUME_SQLCODE={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。

DECLARE_SECTION=YES の場合に ASSUME_SQLCODE=YES と指定すると、SQLCODE を宣言文の外で宣言できます。

DECLARE_SECTION=YES の場合に ASSUME_SQLCODE=NO と指定すると、次の基準のうち少なくとも 1 つが満たされた場合にだけ、SQLCODE は状態変数として認識されます。

- 完全に正しいデータ型で宣言されている。
- Pro*COBOL が他の状態変数を見つけられない。Pro*COBOL は、（完全に正しい型の）SQLSTATE 宣言を検出した場合や、SQLCA の組込みを検出した場合には、SQLCODE が宣言されているとはみなしません。

ASSUME_SQLCODE=YES と指定した場合は、SQLSTATE または SQLCA あるいはその両方が状態変数として宣言されると、SQLCODE が宣言されているかどうか、また正しい型かどうかに関係なく、Pro*COBOL は SQLCODE が宣言されているものとみなします。

AUTO_CONNECT

用途

プログラムをデフォルトのユーザー・アカウントに自動的に接続するかどうかを指定します。

構文

AUTO_CONNECT={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。

AUTO_CONNECT=YES と指定した場合、Pro*COBOL が実行 SQL 文を検出すると同時に、ユーザー・プログラムは自動的に次のユーザー ID で Oracle にログインしようとします。

```
<prefix><username>
```

prefix には Oracle 初期化パラメータ OS_AUTHENT_PREFIX の値（デフォルト値は OPS$）を指定し、*username* には使用しているオペレーティング・システムのユーザー名またはタスク名を指定します。この場合、コマンド行に別の値を指定しても、MAXOPENCURORS (10) のデフォルト値を変更することはできません。

AUTO_CONNECT=NO（デフォルト）の場合は、Oracle にログインするには CONNECT 文を使用する必要があります。

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

入力できるのは、コマンド行または構成ファイルからだけです。

このオプションは、DECLARE CURSOR 文で WITH HOLD 句を使わずに宣言されたカーソルがある場合にだけ有効です（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。コマンド行で MODE=ORACLE と指定した場合は、WITH HOLD 句を使わ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。

COMMIT または ROLLBACK を発行すると、明示的なカーソルがすべてクローズされます。（MODE=ORACLE の場合は、コミットまたはロールバックを発行すると CURRENT OF 句で参照されているカーソルだけがクローズされます。）

このオプションの優先順位の詳細は、14-5 ページの「マクロ・オプションとマイクロ・オプション」を参照してください。

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

NO

使用上の注意

入力できるのは、コマンド行でだけです。

Pro*COBOL は、コマンド行オプションがあらかじめ設定されている構成ファイルを使用できます。ユーザー構成ファイルと呼ばれるいくつかの代替ファイルのいずれかを指定することもできます。詳細は 14-6 ページの「構成ファイル」を参照してください。

構成ファイルはネストできません。したがって、構成ファイルの中ではオプション CONFIG を指定できません。

DATE_FORMAT

用途

日付が戻される文字列の書式を指定します。

構文

DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | '*fmt*'（デフォルト LOCAL）

デフォルト値

LOCAL

使用上の注意

入力できるのは、コマンド行または構成ファイルからだけです。指定できる日付文字列を次の表に示します。

表 14-3 日付文字列用の書式

書式名	略称	日付書式
国際標準化機構規格	ISO	yyyy-mm-dd
USA 標準	USA	mm/dd/yyyy
ヨーロッパ標準	EUR	dd.mm.yyyy
日本工業規格	JIS	yyyy-mm-dd
導入時定義	LOCAL	導入時に定義した任意の書式

"*fmt*' は、'Month dd, yyyy' などの日付書式モデルです。日付書式モデル要素のリストは、『Oracle8*i* SQL リファレンス』を参照してください。

DATE_FORMAT オプションの使用方法には、制限が 1 つあります。コンパイルした単位を後でリンクする場合、すべての単位が同じ DATE_FORMAT 値を使用している必要があります。コンパイル単位間で DATE_FORMAT の値に不一致があると、エラーが発生します。

DBMS

用途

Oracle の意味上および構文上の規則を、Oracle バージョン 7、Oracle8*i*、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されているバージョン）のうちのどの規則に合わせるかを指定します。

構文

DBMS={NATIVE | V7 | V8}

デフォルト値

NATIVE

使用上の注意

インラインでは入力できません。

DBMS オプションを使って、Oracle のバージョン固有の動作を制御できます。
DBMS=NATIVE（デフォルト）の場合、Oracle は、Oracle のネイティブ・バージョンの意味上および構文上の規則に従います。

表 14-4 に、互換性のある DBMS と MODE の設定値がどのようにやりとりするかを示します。これ以外の組合せは、互換性がないか不適切です。

表 14-4 DBMS と MODE 間のやりとり

状況	DBMS=V7 または V8 MODE=ANSI	DBMS=V7 または V8 MODE=ORACLE
標識変数を使用せずに切捨て値をフェッチします。	エラーにならないが SQLWARN(2) が設定されます。	エラーにならないが SQLWARN(2) が設定されます。
既にオープンしているカーソルのオープン	エラー -2117	エラーになりません。
既にクローズしているカーソルのクローズ	エラー -2114	エラーになりません。
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 グループ関数がコールされるタイミング	FETCH 時	FETCH 時
SQLCA 構造体の宣言	オプション	必須
SQLCODE または SQLSTATE ステータス変数の宣言	必須	指定できるが Oracle は無視
DESCRIBE が戻す外部データ型コード（動的 SQL 方法 4）	96	96
整合性制約	有効	有効
ロールバック・セグメント用 PCTINCREASE	使用不可	使用不可
MAXEXTENTS 記憶領域パラメータ	使用不可	使用不可

注意 :

1. ANSI13 を含みます。

2. ANSI14 と ANSI13 を含みます。

DECLARE_SECTION

用途

宣言文内の宣言だけをホスト変数として使用するかどうかを指定します。

構文

DECLARE_SECTION={YES | NO}

デフォルト値

NO

使用上の注意

入力できるのは、コマンド行または構成ファイルからだけです。

Pro*COBOL リリース 8.0 以降では、MODE=ORACLE であれば、BEGIN DECLARE SECTION 文と END DECLARE SECTION 文は省略できます。DECLARE_SECTION オプションは、旧リリースとの下位互換性のために用意されているものです。DECLARE_SECTION は MODE のマイクロ・オプションです。

このオプションは、コマンド行または構成ファイル内だけで指定しなければなりません。このオプションは MODE オプションに関連するマイクロ・オプションです（マイクロ・オプションは 1 つの動作だけを制御します。これに対し、マクロ・オプションは複数の動作を制御します）。また、マクロ・オプションのレベルがマイクロ・オプションのレベルより高いときにだけマクロ・オプションがマイクロ・オプションに優先する、という一般規則がこのオプションにも適用されます。レベルを高い順に示すと次のようになります。

- インライン
- コマンド行
- ユーザー構成ファイル
- システム構成ファイル
- デフォルト値

このオプションを使うと、MODE=ORACLE と DECLARE_SECTION=YES を一緒に指定でき、旧リリースで MODE=ORACLE だけを指定した場合と同じ結果が得られます（ホスト変数として使用できるのは DECLARE 文で宣言した変数だけです）。このオプションの優先順位の説明は、14-5 ページの「マクロ・オプションとマイクロ・オプション」を参照してください。

DEFINE

用途

条件付きプリコンパイル時にソース・コードの一部組込みまたは除外を行うために使う、ユーザー定義の記号を指定します。詳細は2-25 ページの「条件付きプリコンパイル」を参照してください。

構文

DEFINE=*symbol*

デフォルト値

なし

使用上の注意

DEFINE をインラインで入力する場合の EXEC ORACLE 文の書式は次のとおりです。

```
EXEC ORACLE DEFINE <symbol> END-EXEC.
```

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 の設定によって、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 文を使ってインラインを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、10-11 ページの「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。

END_OF_FETCH

用途

SQL 文の実行後に END-OF-FETCH 条件が発生した場合に戻される SQLCODE 値を指定します。

構文

END_OF_FETCH={100 | 1403}

デフォルト値

1403

使用上の注意

入力できるのは、コマンド行または構成ファイルからだけです。

END_OF_FETCH は MODE のマイクロ・オプションです。詳細は、14-5 ページの「マクロ・オプションとマイクロ・オプション」を参照してください。

構成ファイルで MODE=ANSI と指定した場合、END_OF_FETCH 条件が発生すると、Pro*COBOL は SQLCODE 値 100 を戻し、END_OF_FETCH のデフォルト値である 1403 を上書きします。

構成ファイルで MODE=ANSI と END_OF_FETCH=1403 の両方を指定した場合は、END_OF_FETCH 条件が発生すると、Pro*COBOL は SQLCODE 値 1403 を戻します。

構成ファイルで MODE=ANSI と指定し、構成ファイルよりも優先順位の高いコマンド行で END_OF_FETCH=1403 と指定した場合も、END_OF_FETCH 条件が発生すると Pro*COBOL は SQLCODE 値 1403 を戻します。

ERRORS

用途

Pro*COBOL のエラー・メッセージを、端末とリスト・ファイルの両方に送るか、リスト・ファイルだけに送るかを指定します。

構文

ERRORS={YES | NO} *

デフォルト値

YES

使用上の注意

ERRORS=YES と指定すると、エラー・メッセージは端末とリスト・ファイルの両方に送られます。

ERRORS=NO のときは、エラー・メッセージはリスト・ファイルだけに送られます。

FIPS

用途

ANSI/ISO SQL の拡張要素に（FIPS フラガーによって）フラグを付けるかどうかを指定します。拡張要素とは、ANSI/ISO の形式または構文規則（権限付与規則は除く）に従っていない SQL 要素のことです。

構文

FIPS={YES | NO}

デフォルト値

NO

使用上の注意

FIPS=YES の場合は、ANSI/ISO に組み込まれた SQL 規格（SQL92）からの Oracle 拡張要素を使用したり、SQL92 の機能を規格に準拠しない方法で使用すると、FIPS フラガーは警告メッセージを発行します（エラー・メッセージではありません）。

プリコンパイル時に、次に示す ANSI/ISO SQL の拡張要素にフラグが付きます。

- FOR 句を含む配列インタフェース
- SQLCA および ORACA、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 および COMP-3、NUMBER、RAW、LONG RAW、VARRAW、ROWID、VARCHAR データ型
- 実行時オプションを指定するための ORACLE OPTION 文
- ユーザー・イグジットでの EXEC IAF 文および EXEC TOOLS 文
- CONNECT 文
- TYPE および VAR データ型の同値化文
- AT *db_name* 句
- DECLARE...DATABASE 文および ...STATEMENT 文、...TABLE 文
- WHENEVER 文での SQLWARNING 条件
- WHENEVER 文の DO 処置および STOP 処置

- 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
- INTO 句の WHENEVER...DO ラベルおよびホスト変数の前に付けるオプションのコロン

FORMAT

用途

COBOL 文の書式を指定します。

構文

FORMAT={ANSI | TERMINAL}

デフォルト値

ANSI

使用上の注意

インラインでは入力できません。

入力行の書式はシステムによって異なります。システム固有の Oracle のマニュアルまたは COBOL コンパイラをチェックしてください。

FORMAT=ANSI と指定した場合、入力行の書式は COBOL に関する現行の ANSI 規格にできる限り準拠しているものと解釈されます。FORMAT=TERMINAL と指定した場合、入力行は列 1 から始まります。このマニュアルのコード例は TERMINAL 書式内にあります。詳細は 2-12 ページの「コーディング領域」を参照してください。

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取扱い方法を指定します。

構文

HOLD_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

HOLD_CURSOR を使うと、プログラムのパフォーマンスを改善できます。詳細は付録 D の「パフォーマンス・チューニング」を参照してください。

SQL データ操作文を実行すると、その文に対応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。続いてカーソル・キャッシュ・エントリは Oracle プライベート SQL 領域にリンクされます。この領域には SQL 文の実行に必要な情報が格納されます。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 と指定した場合、Oracle が SQL 文を実行してカーソルがクローズされた後、Pro*COBOL はそのリンクを再使用可能としてマークします。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 および RELEASE_CURSOR=NO を指定した場合は、リンクは維持され、Pro*COBOL はリンクを再使用しません。この設定によって後に続く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。文を解析し直したり、Oracle プライベート SQL 領域にメモリーを割り当てたりする必要がないためです。

暗黙的カーソル用としてインラインで使うときは、SQL 文の実行前に HOLD_CURSOR を設定してください。明示的なカーソルについてインラインで指定する場合は、カーソルをオープンする前に HOLD_CURSOR を設定してください。

RELEASE_CURSOR ＝ YES と指定した場合、HOLD_CURSOR=YES が上書きされます。また、HOLD_CURSOR=NO と指定した場合は、HOLD_CURSOR=NO が上書きされます。これら 2 つのオプションの関係については、付録 D の「パフォーマンス・チューニング」の D-12 ページの表 D-1 の「HOLD_CURSOR および RELEASE _CURSOR の相互関係」を参照してください。

HOST

用途

使用するホスト言語を指定します。

構文

HOST={COB74 | COBOL}

デフォルト値

COBOL

使用上の注意

インラインでは入力できません。

COB74 は、ANSI 承認 COBOL の 1974 版を表します。COBOL は、1985 版を表します。プラットフォームによっては、これ以外の値も使用できます。

INAME

用途

入力ファイル名を指定します。

構文

INAME=*filename*

デフォルト値

なし

使用上の注意

インラインでは入力できません。

プリコンパイル時は、すべての入力ファイル名が一意でなければなりません。

コマンド行から入力ファイルの名前を指定する場合は、キーワード INAME は省略できます。たとえば、Pro*COBOL では、INAME=*myprog.pco* と指定するかわりに *myprog.pco* と指定できます。

Pro*COBOL では、ファイル名の最初の一文字が小文字の場合は、標準の入力ファイル拡張子を *pco* と見なします。また、最初の一文字が大文字の場合は、PCO と見なします。

このように、INAME の指定でファイル拡張子を使用する必要はありません（ただし、ファイル拡張子が標準以外の場合には拡張子を指定する必要があります）。INAME の指定で非標準の入力ファイル拡張子を使用する場合は、HOST も指定しなければなりません。

INCLUDE

用途

EXEC SQL INCLUDE ファイルのディレクトリ・パスを指定します。このオプションは、ディレクトリを使用するオペレーティング・システム専用です。

構文

INCLUDE=*path*

デフォルト値

現行のディレクトリ

使用上の注意

通常、INCLUDE は SQLCA ファイルおよび ORACA ファイルのディレクトリ・パスを指定するために使用します。Pro*COBOL は、最初に現行のディレクトリを検索し、次に INCLUDE で指定されたディレクトリを検索して、最後に標準 INCLUDE ファイル用のディレクトリを検索します。このため、SQLCA や ORACA などの標準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。

しかし、標準以外のファイルについては、現行のディレクトリに格納されている場合を除いて、INCLUDE を使ってディレクトリ・パスを指定する必要があります。次に示すように、コマンド行に複数のパスを指定できます。

```
... INCLUDE=<path1> INCLUDE=<path2> ...
```

Pro*COBOL は、最初に現行のディレクトリを検索し、次に *path1* で指定されたディレクトリを検索し、続いて *path2* で指定されたディレクトリを検索して、最後に標準 INCLUDE ファイル用のディレクトリを検索します。

注意 : ディレクトリ・パスを指定しても、Pro*COBOL は最初に現行のディレクトリを検索します。このため、INCLUDE したいファイルが現行のディレクトリ以外の場所に存在する場合は、同じ名前のファイルが現行のディレクトリにないことを確認してください。

ディレクトリ・パスを指定するための構文はシステムによって異なります。使用しているオペレーティング・システムの規則に従って指定してください。

IRECLEN

用途

入力ファイルのレコード長を指定します。

構文

IRECLEN= 整数

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。

IRECLEN には、ORECLEN の値より大きい値は指定できません。指定可能な最大値はシステムによって異なります。

LITDELIM

用途

LITDELIM オプションは、Pro*COBOL が生成する COBOL コード内の文字列定数およびリテラルのデリミタを指定します。

構文

LITDELIM={APOST | QUOTE}

デフォルト値

QUOTE

使用上の注意

LITDELIM=APOST と指定すると、Pro*COBOL は COBOL コードを生成するときにアポストロフィを使用します。LITDELIM=QUOTE を指定すると、次に示すように引用符が使われます。

```
CALL "SQLROL" USING SQL-TMP0.
```

SQL 文では、次の例に示すように、特殊文字または小文字を含んでいる識別子は引用符で区切らなければなりません。

```
EXEC SQL CREATE TABLE "Emp2" END-EXEC.
```

また、文字列定数を区切る場合は、次の例のようにアポストロフィを使います。

```
EXEC SQL SELECT ENAME FROM EMP WHERE JOB = 'CLERK' END-EXEC.
```

Pro*COBOL は、Pro*COBOL ソース・ファイルで使用されているデリミタに関係なく、LITDELIM の値で指定されたデリミタを使用します。

LNAME

用途

リスト・ファイルのデフォルト以外の名前を指定します。

構文

LNAME=*filename*

デフォルト値

input.LIS。*input* は入力ファイルの基本名です。

使用上の注意

インラインでは入力できません。

デフォルトでは、リスト・ファイルは現行のディレクトリに作成されます。

LRECLEN

用途

リスト・ファイルのレコード長を指定します。

構文

LRECLEN=*integer*

デフォルト値

132

使用上の注意

インラインでは入力できません。

LRCLEN の値の範囲は、80 ～ 255 です。80 未満の値を指定した場合は、80 がセットされます。また、255 より大きい値を指定すると、255 が採用されます。行番号を挿入できるように、LRECLEN には IRECLEN の値より少なくとも 8 大きい値を指定してください。

LTYPE

用途

リスト・ファイルの型を指定します。

構文

LTYPE={LONG | SHORT | NONE}

デフォルト値

LONG

使用上の注意

インラインでは入力できません。

表 14-5 リスト・ファイルの型

LTYPE=LONG	リスト・ファイルに入力行が表示されます。
LTYPE=SHORT	リスト・ファイルに入力行が表示されません。
LTYPE=NONE	リスト・ファイルが作成されません。

MAXLITERAL

用途

コンパイラの制限を超えないように、Pro*COBOL が生成する文字列リテラルの最大長を指定します。たとえば、コンパイラが 132 文字より長い文字列リテラルを処理できない場合は、コマンド行に MAXLITERAL=132 と指定します。

構文

MAXLITERAL=*integer**

デフォルト値

256

使用上の注意

MAXLITERAL に指定可能な最大値は、コンパイラによって異なります。言語ごとに異なるデフォルト値が適用されますが、このデフォルト値より小さい値を指定しなければならない場合もあります。たとえば、COBOL コンパイラの中には 132 文字より長い文字列リテラルを処理できないものもあります。その場合は、MAXLITERAL=132 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。

MAXLITERAL はインラインで入力できますが、プログラムで MAXLITERAL の値を設定できるのは 1 回だけです。また、その EXEC ORACLE 文は最初の EXEC SQL 文より前に記述しなければなりません。この条件に違反すると、Pro*COBOL は警告メッセージを発行し、余分な EXEC ORACLE 文あるいは誤った位置にある EXEC ORACLE 文を無視して、処理を続行します。

MAXOPENCURSORS

用途

同時にオープンされるカーソルの中で、Pro*COBOL がキャッシュに保存しておけるカーソルの最大数を指定します。

構文

MAXOPENCURSORS=*integer*

デフォルト値

10

使用上の注意

MAXOPENCURSORS を使うと、プログラムのパフォーマンスを改善できます。詳細は付録 D の「パフォーマンス・チューニング」を参照してください。

分割してプリコンパイルする場合は、2-26 ページの「分割プリコンパイル」の説明に従って MAXOPENCURSORS を指定してください。

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。新しいカーソルが必要で、空きのキャッシュ・エントリがない場合、Oracle はエントリを再利用しようとします。これが成功するかどうかは HOLD_CURSOR と RELEASE_CURSOR の設定値によって決まります。また明示的なカーソルの場合は、さらにカーソル自体のステータスにも左右されます。再利用できるキャッシュ・エントリが見つからない場合、Oracle は追加のキャッシュ・エントリを割り当てます。Oracle は、空きメモリーがなくなるか OPEN_CURSORS で設定された限界に達するまで、必要に応じてキャッシュ・エントリの割当

てを続行します。"maximum open cursors exceeded" という Oracle エラーを避けるには、MAXOPENCURSORS の値が OPEN_CURSORS の値より 6 以上小さくなければなりません。

プログラムが同時に必要とするオープン・カーソルの数が増えて、MAXOPENCURSORS を再指定しなければならない場合もあります。45 ～ 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つのプライベートな SQL 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。

MODE

用途

プログラムが Oracle の基準に従うか、現行の ANSI SQL 規格に準拠するかを指定します。

構文

MODE={ANSI | ISO | ANSI14 | ISO14 | ANSI13 | ISO13 | ORACLE}

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。

たとえば、次の 2 つの文は等価です。ANSI と ISO、ANSI14 と ISO14、ANSI13 と ISO13。

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。

MODE={ANSI14 | ANSI13} と指定した場合、プログラムはほぼ現行の ANSI SQL 規格に準拠します。

MODE=ANSI と指定した場合、プログラムは完全に ANSI 規格に準拠し、次のような変更が加えられます。

- 既にオープンされているカーソルを OPEN したり、既にクローズされているカーソルを CLOSE することはできません。（MODE=ORACLE の場合は、再解析をさけるためにオープンされているカーソルを再 OPEN できます。）
- Oracle が切り捨てられた列値を出力ホスト変数に割り当てた場合、エラー・メッセージは発行されません。

MODE={ANSI | ANSI14} と指定した場合、SQLCODE という 4 バイトの整変数または SQLSTATE という 5 バイトの文字変数を宣言しなければなりません。詳細は 8-2 ページの「エラー処理の代替手段」を参照してください。

表 14-4 の「DBMS と MODE 間のやりとり」に、互換性のある DBMS と MODE の設定値がどのようにやりとりするかを示します。これ以外の組合わせは、互換性がないか不適切です。

NESTED

用途

ネストしたプログラムの GLOBAL 句が生成されたかどうかを示します。コンパイラがネストしたプログラムをサポートしている場合は、NESTED 値として YES を使います。

構文

NESTED={YES | NO}

デフォルト値

YES

使用上の注意

インラインでは入力できません。

NLS_LOCAL

用途

NLS_LOCAL オプションは、NLS 文字変換が Pro*COBOL ランタイム・ライブラリによって実行されるか Oracle Server によって実行されるかを指定します。

構文

NLS_LOCAL={YES | NO}

デフォルト値

NO

使用上の注意

インラインでは入力できません。

NLS_LOCAL=YES と指定した場合、マルチバイト NLS データ型をもつホスト変数の空白埋込みと空白削除はランタイム・ライブラリ（SQLLIB）によってローカルに実行されます。

リリース 8.0 以前のリリース用に記述された Pro*COBOL アプリケーションの場合は、引き続きこの値を使います。

NLS_LOCAL=NO と指定した場合は、マルチバイト NLS データ型をもつホスト変数の空白埋込みおよび空白削除の操作は Oracle Server によって行われます。新しい Oracle8.0 以降のアプリケーションについては、すべてこの値を使用してください。

環境変数 NLS_NCCHAR は、有効な固定幅の各国キャラクタ・セットに設定する必要があります。可変幅の各国キャラクタ・セットはサポートされていません。

ONAME

用途

出力ファイル名を指定します。

構文

ONAME=*filename*

デフォルト値

システムによって異なります。

使用上の注意

インラインでは入力できません。

このオプションは、出力ファイルの名前が入力ファイルの名前と異なる場合に、出力ファイルの名前を指定するために使用します。たとえば、次のコマンドを発行したとします。

```
procob INAME=my_test
```

デフォルト出力ファイル名は、*my_test.cob* です。出力ファイル名を *my_test_1.cob* にする場合は、次のコマンドを発行します。

```
procob INAME=my_test ONAME=my_test_1.cob
```

ONAME を使って指定するファイルには、*.cob* 拡張子を付けてください。ONAME オプションにはデフォルトの拡張子はありません。

> **注意 :** 出力ファイルにはデフォルト名を使うのではなく、ONAME オプションで明示的に指定してください。

ORACA

用途

プログラムが 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

ORACA=YES と指定した場合は、プログラムに INCLUDE ORACA 文を記述しなければなりません。

ORECLEN

用途

出力ファイルのレコード長を指定します。

構文

ORECLEN=*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。

ORECLEN に指定する値は、IRECLEN の値と同じか、それより大きくなければなりません。指定可能な最大値はシステムによって異なります。

PAGELEN

用途

リスト・ファイルの物理ページ当たりの行数を指定します。

構文

PAGELEN=*integer*

デフォルト値

66

使用上の注意

インラインでは入力できません。指定可能な最大値はシステムによって異なります。

PICX

用途

PIC X 変数のデフォルトのデータ型を指定します。

構文

PICX={CHARF | VARCHAR2}

デフォルト値

CHARF

使用上の注意

入力できるのは、コマンド行または構成ファイルからだけです。

Pro*COBOL 8.0 からは、PIC X または N、G 変数のデフォルトのデータ型が VARCHAR2 から CHARF に変わりました。PICX は、下位互換性を維持するために用意されているオプションです。

新しいデフォルトの動作は、COBOL の標準の移動規則と整合がとれています。新しい方式により、PIC X 変数を（MODE=ORACLE と指定して）VARCHAR2 列に挿入した場合の動作が変わります。以前は後続ブランクは切り捨てられましたが、切り捨てられなくなります。また、新しいデフォルトにより、バインド変数の後続ブランクが比較の前に切り捨てられるので、WHERE 句で後続ブランク付きで初期化したバインド変数を使用すると、char 列に格納された後続ブランク数を持つ値とは一致しなくなるというような状態が少なくなります。

PICX=VARCHAR2 と指定した場合、Oracle は PL/SQL ブロック内のローカル CHAR 変数を可変長文字値のように扱います。PICX=CHARF と指定した場合は、Oracle は CHAR 変数を ANSI 準拠の固定長文字値のように扱います。詳細は、4-30 ページの「PIC X のデフォルト」を参照してください。

PREFETCH

用途

このオプションを使って一定の行数をプリフェッチすると、クエリーが高速化します。

構文

PREFETCH=*integer*

デフォルト値

1

使用上の注意

入力できるのは、構成ファイルまたはコマンド行からだけです。整数値は、明示的なカーソルを使うクエリーの実行にすべて適用されます。このとき、優先順位のルールも適用されます。

インラインで使うときは、明示的なカーソルを使い、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 PREFETCH オプションにより決まります。

最大値は 9999 です。詳細は 5-19 ページの「PREFETCH オプション」を参照してください。

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取扱い方法を指定します。

構文

RELEASE_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

RELEASE_CURSOR を使うと、プログラムのパフォーマンスを改善できます。詳細は付録 D の「パフォーマンス・チューニング」を参照してください。

SQL データ操作文を実行すると、その文に対応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。続いてカーソル・キャッシュ・エントリは Oracle プライベート SQL 領域にリンクされます。この領域には SQL 文の実行に必要な情報が格納されます。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 と指定した場合、Oracle が SQL 文を実行してカーソルがクローズされると、Pro*COBOL はただちにリンクを削除します。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 と HOLD_CURSOR=YES を指定すると、リンクは保持されます。オープンされているカーソルの数が MAXOPENCURSORS の値を超えないかぎり、Pro*COBOL はリンクを再使用しません。この設定によって後に続く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。文を解析し直したり、Oracle プライベート SQL 領域にメモリーを割り当てたりする必要がないためです。

暗黙的カーソル用としてインラインで使うときは、SQL 文の実行前に RELEASE_CURSOR を設定してください。明示的なカーソルについてインラインで指定する場合は、カーソルをオープンする前に RELEASE_CURSOR を設定してください。

RELEASE_CURSOR ＝ YES と指定した場合、HOLD_CURSOR=YES に上書きされます。また、HOLD_CURSOR=NO と指定した場合は、HOLD_CURSOR=NO に上書きされることに注意してください。これらの 2 つのオプションが、どのように相互干渉するかについては付録 D の「パフォーマンス・チューニング」を参照してください。特に、D-12 ページの表 D-1 の「HOLD_CURSOR および RELEASE _CURSOR の相互関係」を参照してください。

SELECT_ERROR

用途

1 行を戻す SELECT 文で複数の行が戻されたり、ホスト配列に入りきらない数の行が戻されたりした場合に、プログラムがエラーを生成す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

SELECT_ERROR=YES と指定すると、1 行を戻す選択で複数行が戻された場合や、配列の選択でホスト配列に入りきらない数の行が戻された場合にはエラーが発生します。

SELECT_ERROR=NO と指定すると、1 行を戻す選択で複数行が戻された場合や、配列の選択でホスト配列に入りきらない数の行が戻された場合でも、エラーは発生しません。

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。特定の順序で行を選択したい場合は、SELECT 文で ORDER BY 句を使用してください。SELECT_ERROR=NO と指定した場合、ORDER BY 句を使用すると、Oracle は最初の行を戻します。また、配列の選択の場合は最初の *n* 行を戻します。SELECT_ERROR=YES と指定した場合は、ORDER BY 句を使用してもしなくても、戻された行が多すぎる場合にはエラーが発生します。

SQLCHECK

用途

構文および意味チェックのタイプとレベルを指定します。

構文

SQLCHECK={SEMANTICS | FULL | SYNTAX | LIMITED}

デフォルト値

SYNTAX

使用上の注意

値 SEMANTICS と FULL は等価です。また、SYNTAX と LIMITED も等価です。

Pro*COBOL は、埋込み SQL 文および PL/SQL ブロックの構文と意味をチェックすることによって、プログラムのデバッグを支援します。検出されたエラーはプリコンパイル時にレポートされます。

チェックのレベルを制御するには、コマンド行もしくはインライン、またはその両方で SQLCHECK オプションを入力します。ただし、インラインで指定するチェックのレベルを、コマンド行で指定する（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。

SQLCHECK=SYNTAX|SEMANTICS と指定した場合、PL/SQL の予約語が SQL 文で使用されていると、その SQL 文が PL/SQL でない場合でも、Pro*COBOL はエラーを生成します。PL/SQL の予約語を識別子として使用しなければならない場合は、二重引用符で囲んでください。

SQLCHECK=SEMANTICS と指定した場合、Pro*COBOL は下記を対象として構文上および意味上のチェックを行います。

- INSERT 文や UPDATE 文などのデータ操作文
- PL/SQL ブロック

ただし、リモート・データ操作文（AT *db_name* 句を使用するデータ操作文）については、構文上のチェックだけが行われます。

Pro*COBOL は、意味上のチェックに必要な情報を、埋め込まれた DECLARE TABLE 文から入手します。また、オプション USERID が指定されている場合は、Oracle に接続してデータ・ディクショナリに アクセスすることによって入手します。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 文で定義されていれば、Oracle に接続する必要はありません。

Oracle に接続してデータ・ディクショナリにアクセスしても見つからない情報があった場合は、DECLARE TABLE 文を使用して欠けている情報を提供しなければなりません。プリコンパイル時に DECLARE TABLE 文の定義とデータ・ディクショナリの定義の内容が矛盾する場合は、DECLARE TABLE 文の定義が使用されます。

新しいプログラムをプリコンパイルするときは、SQLCHECK=SEMANTICS を指定してください。ホスト・プログラムに PL/SQL ブロックを埋め込む場合は、SQLCHECK=SEMANTICS と指定し、オプション USERID を指定する必要があります。

SQLCHECK=SYNTAX と指定した場合は、Pro*COBOL は下記を対象として構文上のチェックを行います。

- データ操作文
- PL/SQL ブロック

意味上のチェックは行われません。DECLARE TABLE 文は無視されます。また、PL/SQL ブロックは使用できません。データ操作文のチェックには、下位互換性のある Oracle8*i* 構文規則が使用されます。プリコンパイル済みのプログラムを移行する場合は、SQLCHECK=SYNTAX を指定してください。

表 14-6 に、SQLCHECK で行われる検査をまとめてあります。構成検査と意味検査の詳細は、付録 E の「構文および意味検査」を参照してください。

表 14-6 SQLCHECK による検査

	SQLCHECK=SEMANTICS の指定		SQLCHECK=SYNTAX の指定	
	構文	意味	構文	意味
DML	**X**	X	X	
リモート DML	X		X	
PL/SQL	X	X		

TYPE_CODE

用途

このマイクロ・オプションでは、動的 SQL 方法 4 で ANSI または Oracle データ型コードのどちらを使うかを決定します。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については、10-12 ページの表 10-3 を参照してください。

UNSAFE_NULL

用途

UNSAFE_NULL=YES を指定すると、標識変数を使わないで 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。

UNSAFE_NULL=YES と指定できるのは、MODE=ORACLE かつ DBMS=V7 または V8 の場合だけです。

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は、必ず標識変数を使ってください。

UNSAFE_NULL=YES と指定すると、SELECT 文または FETCH 文で NULL が選択され、出力ホスト変数に対応する標識変数がない場合でも、エラーは戻されません。UNSAFE_NULL=NO と指定した場合、対応する標識変数のないホスト変数に NULL の列または式を SELECT または FETCH すると、エラーが発生します（SQLSTATE は "22002" に、SQLCODE は ORA-01405 に設定されます）。

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

インラインでは入力できません。

自動ログイン機能を使用する場合は、このオプションは指定しないでください。自動ログインでは、Oracle 初期化パラメータ OS_AUTHENT_PREFIX の値を接頭辞とする Oracle ユーザー名が受け入れられます。

SQLCHECK=SEMANTICS と指定した場合に、Oracle に接続してデータ・ディクショナリにアクセスすることによって Pro*COBOL が必要な情報を得られるようにしたいときは、USERID も指定する必要があります。データベース別名はオプションです。大カッコは、入力しないでください。

VARCHAR

用途

VARCHAR オプションは、第 5 章の「埋込み SQL」で説明した COBOL のグループ項目を VARCHAR データ型として扱うように Pro*COBOL に指示します。

構文

VARCHAR={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。

VARCHAR=YES と指定した場合、第 5 章の「埋込み SQL」で説明した暗黙的なグループ項目を、長さのフィールドと文字列フィールドを持つ VARCHAR 外部データ型として受け入れます。

VARCHAR=NO と指定した場合、Pro*COBOL は暗黙的なグループ項目を VARCHAR 外部データ型として受け入れません。

XREF

用途

リスト・ファイルにクロス・リファレンスのセクションを組み込むかどうかを指定します。

構文

XREF={YES | NO}

デフォルト値

YES

使用上の注意

XREF=YES と指定すると、ホスト変数、カーソル名、文名についてクロス・リファレンスが組み込まれます。クロス・リファレンスは、個々のオブジェクトがプログラム内のどこで定義され、どこで参照されているかを示します。

XREF=NO と指定した場合は、クロス・リファレンスのセクションは組み込まれません。

A

新機能

この付録では、Oracle Pro*COBOL プリコンパイラで強化された点と新機能を紹介します。各項目について簡単に説明し、各章にある詳細な説明の参照先を示します。

この付録の構成は、次のとおりです。

- リリース 8.1 の新機能
- リリース 8.0 の DB2 互換性機能
- リリース 8.0 の他の新機能
- 以前のリリースからの移行

リリース 8.1 の新機能

CALL 文

CALL 埋込み SQL 文はストアド・プロシージャを起動します。CALL 文は、新しいアプリケーションで埋込み PL/SQL ブロックのかわりに使用できます。詳細は F-13 ページの「CALL（実行可能な埋込み SQL）」を参照してください。

Java メソッドのコール

Java で記述されたストアド・プロシージャ（メソッド）をアプリケーションからコールできます。Java で記述されたプロシージャをコールする方法は、6-20 ページの「ストアド PL/SQL と Java サブプログラム」を参照してください。

LOB サポート

埋込み SQL 文のインタフェースにより、LOB（large object）をプリコンパイラのアプリケーションで使えます。LOB の使用方法、内部 LOB と外部 LOB、LOB を処理する他の方法との比較について説明します。新しい SQL 文のおのおのについて説明します。LOB インタフェースの使用方法をサンプル・コードで示します。詳細は第 13 章の「ラージ・オブジェクト（LOB）」を参照してください。

ANSI 動的 SQL

埋込み SQL 文を使用した動的 SQL 方法 4 の完全な ANSI インプリメンテーションを第 10 章の「ANSI 動的 SQL」に示します。簡単な例で概要を示します。次に、新しい SQL 文を詳細に説明します。さらに、demo ディレクトリにあるサンプル・プログラムを示します。

PREFETCH オプション

このプリコンパイラ・オプションは、値を " あらかじめフェッチする " ことによりデータベース・アクセスの速度を向上させ、ネットワークのサーバー・ラウンドトリップ数を少なくします。詳細は 5-19 ページの「PREFETCH オプション」を参照してください。

DML 戻り句

この句によりデータベース・サーバーにラウンドトリップを保存することができ、現在は INSERT、DELETE および UPDATE 文で使用できます。詳細は 5-8 ページの「行の挿入」を参照してください。

ユニバーサル ROWID

ユニバーサル ROWID 型がサポートされます。索引構成表では、この概念を使用します。詳細は 4-34 ページのユニバーサル ROWID を参照してください。

ユーザー指定の実行時コンテキスト

ユーザー指定の実行時コンテキストについて説明します。C プログラムから COBOL サブプログラムをコールできるようになりました。コンテキストは、Pro*COBOL プログラムによって C サブプログラムに渡すことができます。詳細は 4-33 ページの「ユーザー指定の実行時コンテキスト」を参照してください。

CONNECT 文の SYSDBA/SYSOPER 権限

CONNECT 文を使用してこれらの権限を設定するには、3-2 ページの「Oracle への接続」を参照してください。

グループ項目の表

Pro*COBOL のホスト変数としてグループ項目の表を使用できるようになりました。詳細は 7-19 ページの「ホスト変数としてのグループ項目の表」を参照してください。

WHENEVER DO CALL 分岐

WHENEVER ディレクティブに DO CALL アクションが備わりました。サブプログラムがコールされます。詳細は 8-28 ページの「WHENEVER ディレクティブ」を参照してください。

DECIMAL-POINT IS COMMA

DECIMAL-POINT IS COMMA 句がサポートされています。これにより、数値リテラルの小数点のかわりにカンマを使用できます。詳細は 2-14 ページの「カンマ小数点」を参照してください。

ヘッダー部のオプション化

IDENTIFICATION、ENVIRONMENT、DATA の各部とその内容はオプションになりました。詳細は 2-14 ページの「オプションの DIVISION」を参照してください。

NESTED オプション

NESTED プリコンパイラ・オプションを NO に設定すると、ネストしていないプログラムのための GLOBAL 句は生成されません。詳細は 14-31 ページの「NESTED」を参照してください。

リリース 8.0 の DB2 互換性機能

Pro*COBOL リリース 8.0 の新機能は、DB2 から Oracle にアプリケーションを移行するのに役立ちますが、Pro*COBOL のユーザーはすべてそれらの機能を十分に検討する必要があります。

宣言文のオプション化

DECLARE_SECTION=NO（デフォルト）のときは、BEGIN DECLARE SECTION 文と END DECLARE SECTION 文の使用は任意になりました。この 2 つの DECLARE SECTION 文を使う場合は、同じ WORKING-STORAGE SECTION 内またはその他の COBOL 宣言単位内で対になっていなければなりません。詳細は 14-17 ページの「DECLARE_SECTION」を参照してください。

新しいデータ型のサポート

計算用のデータ型 COMP-4（COMPUTATIONAL-4）は、バイナリ・データ型として扱われます。IBM による計算用データ型 COMP-4（COMPUTATIONAL-4）も、バイナリ・データ型として扱われます。

また、次の表示用データ型がサポートされるようになりました。

- Over-Punch（ZONED-DECIMAL）。これは、COBOL 言語用のデフォルトの符号付き数値です。各桁は 10 を基数とした ASCII 形式または EBCDIC 形式で格納され、1 つの桁がコンピュータ記憶領域の 1 バイトを占めます。符号は、使用されるバイトの 1 つの上位ニブルに格納されます。このデータ型が overpunch と呼ばれるのは、符号が最初または最後のバイトのどちらかに格納された数字の " 上に埋め込まれる " からです。デフォルトの符号位置は、後続バイトです。OVER-PUNCH の指定には、PIC S9(n)V9(m) TRAILING または PIC S9(n)V9(m) LEADING を使います。
- DISPLAY-1 MULTIBYTE 型（PIC G）。このデータ型は PIC N と等価であり、マルチバイト文字に使われます。

詳細は 4-15 ページの「ホスト変数」を参照してください。

ホスト変数としてのグループ項目のサポート

Pro*COBOL 2.0 では、埋込み SQL 文内でグループ項目を使用できるようになりました。ホスト・グループ項目は、SELECT 文または FETCH 文の INTO 句および INSERT 文の VALUES リストで参照できます。グループ項目をホスト変数として使う場合は、SQL 文ではグループ名だけを使います。詳細は 4-22 ページの「ホスト変数としてのグループ項目」を参照してください。

VARCHAR グループ項目の暗黙書式

VARCHAR として認識される COBOL グループの宣言は、次のような書式をとります。

```
nn   <identifier-1>
     49   <identifier-2> PIC S9(4) <integer declaration>.
     49   <identifier-3> PIC X(nc).
```

レベル nn の範囲は 01 ～ 48 です。長さ nc の範囲は 1 ～ 65533 です。

Pro*COBOL に VARCHAR グループ項目の拡張書式を認識させるには、VARCHAR オプションをコマンド行で VARCHAR=YES と指定する必要があります。このオプションを指定しないと、上記の書式の宣言はすべて通常のグループ項目として解釈されます。詳細は 4-29 ページの「VARCHAR 変数の参照」を参照してください。

フェッチ終了時の SQLCODE 戻り値の明示的な制御

DB2 では、フェッチの終了条件が発生すると、SQLCODE 値として 100 が戻されます。Oracle が戻す値を明示的に制御できるように、次のオプションが追加されました。

```
END_OF_FETCH={100 | 1403 (default)}
```

このプリコンパイラ・オプションは、コマンド行または構成ファイル内で指定しなければなりません。詳細は 14-19 ページの「END_OF_FETCH」を参照してください。

DECLARE CURSOR 文での WITH HOLD 句のサポート

DB2 では、デフォルトでコミット時にカーソルがすべてクローズされます。（FOR UPDATE として宣言されている）カーソルの宣言で WITH HOLD 句を指定すれば、そのカーソルについてのこの動作を無効にできます。WITH HOLD 句を指定して宣言されたカーソルはすべて、コミットまたはロールバック後もオープンされたままになります。MODE=ANSI のときは DB2 のデフォルトが適用されますが、その場合はすべてのホスト変数を宣言文で宣言しなければなりません。詳細は 5-12 ページの「カーソルの宣言」を参照してください。

新しいプリコンパイラ・オプション CLOSE_ON_COMMIT

次のプリコンパイラ・オプションが新しく追加されています。

```
CLOSE_ON_COMMIT={YES | NO (default)}
```

このオプションは、コマンド行または構成ファイル内で指定しなければなりません。このオプションは、WITH HOLD 句を指定して記述されていないカーソルがあるときにだけ効力をもちます。WITH HOLD 句が指定されていると、CLOSE_ON_COMMIT 設定も、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。詳細は 5-12 ページの「カーソルの宣言」および 14-14 ページの「CLOSE_ON_COMMIT」を参照してください。

DSNTIAR のサポート

DB2 には、表示可能な形式の SQLCA を取得するためのルーチン DSNTIAR が用意されています。Pro*COBOL でも、DSNTIAR を使用できるようになりました。インタフェースは次のとおりです。

```
CALL "DSNTIAR" USING SQLCA MESSAGE LRECL.
```

SQLCA は SQL 通信領域、MESSAGE は出力メッセージ領域（サイズ 240 以上の VARCHAR 形式）です。LRECL は出力メッセージの長さ（72 ～ 240）が格納されるフルワードです。詳細は、8-27 ページの「DSNTIAR」を参照してください。

日付文字列書式のプリコンパイラ・オプション

Pro*COBOL では、DB2 との互換性のために、日付文字列を指定するための次のプリコンパイラ・オプションが追加されました。

```
DATE_FORMAT={ISO | USA | EUR | JIS | LOCAL | 'fmt' (default LOCAL)}
```

DATE_FORMAT オプションは、コマンド行または構成ファイル内で指定する必要があります。指定できる日付文字列を次の表に示します。

表 A-1 日付文字列用の書式

書式名	略称	日付書式
国際標準化機構規格	ISO	yyyy-mm-dd
USA 標準	USA	mm/dd/yyyy
ヨーロッパ標準	EUR	dd.mm.yyyy
日本工業規格	JIS	yyyy-mm-dd
導入時定義	LOCAL	導入時に定義した任意の書式

'*fmt*' は、'Month dd, yyyy' などの日付書式モデルです。日付書式モデル要素のリストは、『Oracle8*i* SQL リファレンス』を参照してください。詳細は 14-15 ページの「DATE_FORMAT」を参照してください。

SQL 文の後に指定できる終了記号

SQL 文は、カンマまたはピリオド、その他の COBOL 文によって終了できるようになりました。詳細は 2-19 ページの「文終了記号」を参照してください。

リリース 8.0 の他の新機能

構成ファイル名の変更

構成ファイルの名前は、*pccob.cfg* ではなく *pcbcfg.cfg* になりました。14-6 ページの「構成ファイル」を参照してください。

その他の追加データ型のサポート

計算用のデータ型 PACKED-DECIMAL は、ANSI との互換性のために COMP-3 データ型として扱われます。

データ型 SCALED DISPLAY（PIC 9(n) および PIC S9(n)）がサポートされています。各桁は 10 を基数とした ASCII 形式または EBCDIC 形式で格納され、1 つの桁がコンピュータ記憶領域の 1 バイトを占めます。符号がある場合、符号は独立したバイト（句 SIGN SEPARATE で 指定）に格納されます。位置は、デフォルトでは後続ですが、SIGN TRAILING 句を使って指定することもできます。

詳細は 4-15 ページの「ホスト変数」を参照してください。

ネストされたプログラムのサポート

Pro*COBOL では、1 つのソース・ファイルの中で、埋込み SQL によってネストされたプログラムを使用できるようになりました。ただし、再帰的なネストされたプログラムは使用できません。上位プログラムでグローバルとしてマークされた、上位プログラム・レベルで有効なホスト変数である 01 レベルの項目はすべて、上位プログラムが直接的または間接的に含んでいるすべてのプログラムで、有効なホスト変数として使用できます。詳細は 2-22 ページの「ネストされたプログラム」を参照してください。

REDEFINES および FILLER のサポート

REDEFINES 句を使ってグループ項目を再定義できます。詳細は 2-18 ページの「REDEFINES 句」を参照してください。

ホスト変数の宣言でワード FILLER を使用できるようになりました。詳細は 4-15 ページの「ホスト変数」を参照してください。

新しいプリコンパイラ・オプション PICX

PIC X 変数のデフォルトのデータ型が、VARCHAR2 から CHARF に変わりました。これに伴い、下位互換性を保つために次のプリコンパイラ・オプションが追加されました。

PICX={VARCHAR2 | CHARF（デフォルト）}

このオプションは、コマンド行または構成ファイル内でだけ指定できます。新しいデフォルトの動作は、標準の COBOL の移動方法と整合性があります。

詳細は 14-34 ページの「PICX」を参照してください。

VAR 文でのオプションの CONVBUFSZ 句

この句には、キャラクタ・セット間の変換に使うオプションのバッファが指定されます。

詳細は 4-46 ページの「VAR 文の CONVBUFSZ 句」を参照してください。

エラー報告の改善

エラーは、任意の行ファイルまたは端末出力の適切な行に対応付けられるようになりました。「無効なホスト変数」のエラーには、与えられた COBL 変数が埋め込み SQL で無効な理由が述べられています。

接続時のパスワード変更

実行可能な埋込み SQL 文、CONNECT には、パスワードを変更できるようにオプションの最後に新しい句が用意されています。

```
EXEC SQL CONNECT ... [ALTER AUTHORIZATION :new_password] END-EXEC.
```

3-10 ページの「実行時のパスワード変更」および F-17 ページの「CONNECT（実行可能な埋込み SQL 拡張要素）」を参照してください。

エラー・メッセージ・コード

エラー・コードおよび警告コードは、以前のリリースの Pro*COBOL と現行のリリースでは異なります。コードとメッセージの詳細リストは、『Oracle8 エラー・メッセージ』を参照してください。

SQLLIB が発行するランタイム・メッセージには、以前のリリースの Pro*COBOL で使用されていた接頭語 RTL- ではなく、接頭語 SQL- が付くようになりました。メッセージ・コードは以前のリリースのものと同じです。

SQLCHECK=SEMANTICS を使ってプリコンパイルする場合は、PL/SQL コンパイラは PLS を接頭語として使用します。この種のエラーは、Pro*COBOL によるものではありません。

以前のリリースからの移行

Pro*COBOL で記述された既存のアプリケーションは、変更を加えなくても Oracle8*i* サーバーで動作します。アプリケーションに新しい機能を追加する前に Oracle8*i* に移行するには、新しい SQLLIB ライブラリを使って再リンクしてください。

新しい機能を使用していない Pro*COBOL リリース 8.x のアプリケーションは、Oracle7 Server で動作します。

詳細は『Oracle8*i* 移行ガイド』を参照してください。

B

オペレーティング・システムの依存性

COBOL のプログラミングの詳細については、システムによって異なる部分があります。この付録は、Pro*COBOL に関するシステム固有の問題をまとめたものです。マニュアル・セット内の他の資料を参照できる場合は、適宜記載してあります。

このマニュアル内のシステム固有の参照元

この項の参照は、類似する順序およびヘッダーを使用している第 4 章の「データ型とホスト変数」に記載されています。

COBOL のバージョン

Pro*COBOL プリコンパイラは、オペレーティング・システムで使用される標準的な COBOL（通常は COBOL-85 または COBOL-74）の導入をサポートしています。両方の COBOL をサポートするプラットフォームもあります。使用しているシステム固有の Oracle マニュアルを参照してください。

ホスト変数

ホスト変数の宣言方法と命名方法は、使用している COBOL コンパイラによって異なります。ホスト変数の宣言および命名についての詳細は、使用している COBOL のユーザーズ・ガイドをチェックしてください。

宣言

ホスト変数の宣言は、Oracle がサポートする COBOL データ型を指定して、COBOL の規則に従って行います。4-16 ページの表 4-6 の「ホスト変数の宣言」に、指定できる COBOL データ型および擬似型を示します。ただし、使用している COBOL によっては、すべての型が指定可能とは限りません。

命名

ホスト変数名には英文字および数字、ハイフン以外使用できません。また、英文字で始めなければなりません。長さは任意ですが、重要な意味があるのは先頭の 30 文字までです。コンパイラによっては、最大長が異なるものもあります。

INCLUDE 文

INCLUDE は、どのファイルに対しても実行できます。Pro*COBOL プログラムをプリコンパイルすると、EXEC SQL INCLUDE 文はそれぞれ、その文で指定されたファイルのコピーに置き換えられます。

システムでファイル拡張子が参照される場合に拡張子の指定を省略すると、Pro*COBOL プリコンパイラはソース・ファイルのデフォルトの拡張子（通常は COB）を使います。デフォルトの拡張子はシステムによって異なります。使用しているシステム固有の Oracle マニュアルを参照してください。

システムでディレクトリが使われる場合は、プリコンパイラ・オプション INCLUDE=*path* を指定することによって、INCLUDE するファイルのディレクトリ・パスを設定できます。しかし、標準以外のファイルについては、現行のディレクトリに格納されている場合を除いて、INCLUDE を使ってディレクトリ・パスを指定する必要があります。ディレクトリ・パ

スを指定するための構文はシステムによって異なります。使用しているシステム固有の Oracle マニュアルを参照してください。

MAXLITERAL のデフォルト

MAXLITERAL プリコンパイラ・オプションを使って、プリコンパイラが生成する文字列リテラルの最大長を指定し、コンパイラの制限を超えないようにすることができます。MAXLITERAL のデフォルト値は 256 ですが、これより小さい値を指定しなければならないこともあります。

たとえば、お使いの COBOL コンパイラで、132 文字以上の文字列リテラルを扱えない場合は、"MAXLITERAL=132" を指定します。お使いの COBOL コンパイラのユーザーズ・ガイドをチェックしてください。MAXLITERAL オプションの詳細は、第 14 章の「プリコンパイラのオプション」を参照してください。

マルチバイト NLS 文字の PIC N 句または PIC G 句

COBOL コンパイラの中には、マルチバイト NLS 文字変数の宣言で PIC N または PIC G 句を使えないものがあります。マルチバイト NLS 文字変数の宣言用にこれらの句を使うソース・コードを書く前に、お使いの COBOL ユーザーズ・ガイドをチェックしてください。

RETURN-CODE 特別登録は予測できないことがあります。

SQL 文または SQLLIB 関数の後の RETURN-CODE 特別登録（システムがサポートしている場合）の内容は予測できません。

C

予約語、キーワードおよび名前領域

この章は次のトピックで構成されています。

- 予約語とキーワード
- 予約されている名前領域

予約語とキーワード

いくつかのワードが Oracle で予約されています。予約語は Oracle において特別な意味を持っているため、再定義はできません。したがって、予約語は列、表または索引などのデータベース・オブジェクト名としては使用できません。SQL および PL/SQL の Oracle 予約語リストは、『Oracle8*i* SQL リファレンス』および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Pro*COBOL キーワードは、COBOL キーワードと同様、プログラムで変数として使用することはできません。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*COBOL で使用されるキーワードを次に示します。

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	comment	commit
connect	constraint	constraints	context	continue
convbufsz	copy	count	crerate	current
curval	cursor	data	database	date
datetime_interval_code	datetime_interval_precision	day	deallocate	decimal
declare	default	define	delete	desc
describe	descriptor	directory	disable	display
distinct	do	drop	else	enable
end	end-exec	endif	erase	escape
exec	execute	exsits	explain	extract
fetch	file	fileexists	filename	first
float	flush	for	force	found
free	from	function	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	integer
internal_length	intersect	interval	into	is
isopen	istemporary	last	leading	length
level	like	list	load	lob
local	lock	long	max	message
min	minus	minute	mode	month
name	national_character	nchar	next	nextval
noaudit	not	notfound	nowait	NULL
nullable	number	nvachar2	octet_length	of
one	only	open	option	or
oracle	order	output	overlaps	verpunch
package	partition	perform	precision	prepare
prior	procedure	put	raw	read
ref	reference	release	rename	replace
return	returned_length	returned_octet_length	returning	revoke
role	rollback	rowid	rownum	savepoint
scale	second	section	select	set
some	sql	sql_context	sql_cursor	sqlerror
sqlwarning	start	statement	stddev	stop
string	sum	sysdate	sysdba	sysoper
table	temporary	threads	time	timestamp
timezone_hour	timezone_minute	to	tools	trailing
transaction	trigger	trim	truncate	type
uid	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
use	user	using	validate	value
values	var	varchar	varchar2	variables

variance	varnum	varraw	view	whenever
where	with	work	write	year
zone				

予約されている名前領域

表 C-1 に、Oracle8*i* により確保されている名前領域を示します。Oracle8*i* ライブラリでは、サブプログラム名の最初の文字は次に示す文字列に制限されます。名前が競合する可能性があるので、これらの文字で始まらない名前をサブプログラムに付けてください。

たとえば、Net8 透明ネットワーク・サービス・ファンクションはすべて、"NS" 文字で始まるため、名前 "NS" で始まるサブプログラムを書かないようにします。

表 C-1　予約されている名前領域

名前領域	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、OCT	外部 OCI 関数、内部 OCI 関数
UPI、KP	Oracle UPI レイヤーの関数名
NA	Net8 ネイティブ・サービス・プロダクト
NC	Net8 RPC プロジェクト
ND	Net8 ディレクトリ
NL	Net8 ネットワーク・ライブラリ・レイヤー
NM	Net8 ネット管理プロジェクト
NR	Net8 インタチェンジ
NS	Net8 透明ネットワーク・サービス
NT	Net8 ドライバ
NZ	Net8 セキュリティ・サービス
OSN	Net8 V1
TTC	Net8 2 タスク
GEN、L、ORA	Core ライブラリ関数
LI、LM、LX	Oracle NLS レイヤーの関数名
S	システム依存ライブラリの関数名

D

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）を向上させるための単純かつ簡単に適用できる方法をいくつか紹介します。これらの方法を使えば、多くの場合、処理時間を 25% 以上削減できます。この章は次のトピックで構成されています。

- パフォーマンスを低下させる原因
- パフォーマンスを改善する方法
- ホスト配列の使用
- PL/SQL および Java の使用
- 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

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 Oracle の通信オーバーヘッドが高いことです。Oracle は一度に SQL 文を 1 つずつ処理しなければなりません。つまり、各 SQL 文が Oracle をコールするので、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 文を送信しなければならないので、ネットワーク通信量が増加することになります。ネットワーク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 はたいへん柔軟性に富むため、2 つの異なる文から同一の結果を得ることもできますが、効率に差がある場合もあります。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)
END-EXEC.

EXEC SQL SELECT DNAME, DEPTNO
    FROM DEPT
    WHERE EXISTS
    (SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO)
END-EXEC.
```

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 表全体を走査するため、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、索引は使われません。

パフォーマンスを低下させる 3 番目の原因は、不要な解析とバインディングです。SQL 文を実行する前に Oracle がこの SQL 文を解析しバインドしなければならない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 Oracle がその値に対して読込みまたは書込みができるようにそれぞれのアドレスに対応付ける作業です。

大部分のアプリケーションでは、十分にカーソルを管理しているわけではありません。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。

パフォーマンスを改善する方法

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。

ネットワーク化された環境下では、次の方法で、Oracle の通信オーバーヘッドを大幅に削減できます。

- ホスト配列の使用
- 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

以降の項目では、オーバーヘッドを削減するための方法を検討します。

ホスト配列の使用

ホスト配列を使うと、単一の SQL 文でデータの集合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 表に挿入する場合を考えてみます。配列を使わなければ、プログラムでは従業員ごとに 1 回づつ、合計 300 回 INSERT を実行しなければなりません。配列を使えば、必要な INSERT は 1 回だけになります。次の文について考えてみましょう。

```
EXEC SQL INSERT INTO EMP (SAL) VALUES (:SALARY) END-EXEC.
```

SALARY が通常のホスト変数の場合は、Oracle は INSERT 文を 1 回実行し、EMP 表に 1 行を挿入します。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 回実行しなければなりません。

これに対して、SALARY がサイズ 300 のホスト配列の場合は、Oracle は 300 行全部を 1 度に EMP 表に挿入します。各行の SAL 列には SALARY 表の要素の値が格納されます。

詳細は、第 7 章の「ホスト表」を参照してください。

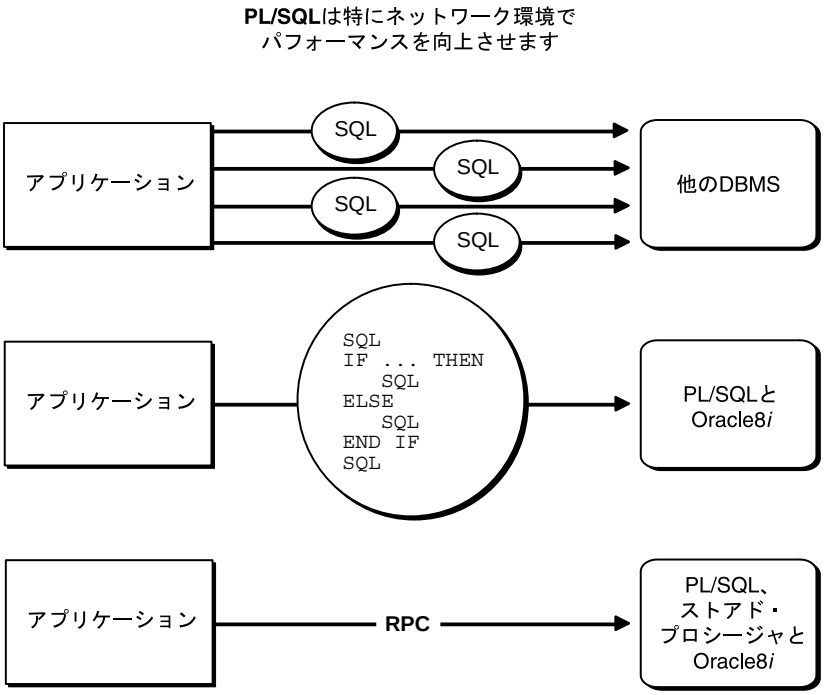
PL/SQL および Java の使用

図 E-1 に示すように、データベース処理が中心のアプリケーションの場合は、手続き型の構造を使用して複数の SQL 文を 1 つの PL/SQL ブロックにまとめ、そのブロック全体を Oracle に送信できます。これにより、使用しているアプリケーションとデータベースとの通信量を大幅に削減できます。

また、PL/SQL および Java サブプログラムを使用して、アプリケーションからデータベースへのコールを削減できます。たとえば、10 の SQL 文を個々に実行するには、10 のコールが必要になります。しかし、10 の SQL 文を含む 1 つのサブプログラムの実行は、1 つのコールだけで済みます。

無名ブロックとは異なり、PL/SQL および Java サブプログラムは別々にコンパイルし、データベースに格納できます。コールされた PL/SQL サブプログラムは、ただちに PL/SQL エンジンに渡されます。さらに、複数のユーザーが 1 つのサブプログラムを実行する場合でも、メモリーにロードするコピーは 1 つだけで済みます。

図 D-1　PL/SQL によるパフォーマンスの向上



また、PL/SQL と Oracle アプリケーション開発ツール（Oracle Forms や Oracle Reports など）と併用できます。PL/SQL によってツールにプロシージャ型の処理能力が加えられるため、パフォーマンスが向上します。PL/SQL を使用することにより、Oracle をコールしなくても、迅速かつ効率的にどのような計算でも行えます。その結果、時間の節約およびネットワーク通信量の削減に役立ちます。詳細は、第 6 章の「埋込み PL/SQL」および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

SQL 文の最適化

オプティマイザはすべての SQL 文に対して実行計画を生成します。実行計画とは、その SQL 文を実行するために Oracle が行う一連のステップです。この手順は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』に記載される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。

オプティマイザ・ヒント

オプティマイザはすべての SQL 文に対して実行計画を生成します。実行計画とは、その SQL 文を実行するために Oracle が行う一連のステップです。場合によっては、SQL 文を最適化する方法を提示することもできます。このような提示はヒントと呼ばれ、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。

ヒントは指示行ではありません。オプティマイザがジョブを実行するのを助けるためのものにすぎません。ヒントの中には、他のヒントが総体的な方針を提示しているときに、SQL 文を最適化するのに使われる情報の有効範囲を制限するものもあります。ヒントを使って、次の事項を指定できます。

- SQL 文のため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
- 表を結合するための方法

ヒントの渡し方

ヒントをオプティマイザに渡すには、SELECT または UPDATE、DELETE 文の動詞の直後の C 形式のコメントの中に入れます。ルール重視の最適化とコスト重視の最適化のどちらかを選択できます。コスト重視の最適化では、ヒントはスループットの最大化と応答時間に寄与します。次の例では、ALL_ROWS ヒントによって問合せのスループットが最大になります。

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ EMPNO, ENAME, SAL
    INTO :EMP-NUMBER, :EMP-NAME, :SALARY
    FROM EMP
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
```

コメント開始記号の直後に正符号 (+) を記述しなければなりません。正符号は、そのコメントに 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。同じコメントに、ヒントのほかに注釈も指定できます。

オプティマイザ・ヒントの詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

トレース機能

SQL トレース機能と EXPLAIN PLAN 文を使うと、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。トレース機能で、Oracle で実行するすべての SQL 文に対する統計表示を生成します。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調整に専念できます。

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。実行計画を使うと、非効率的な SQL 文を特定できます。

このようなツールの使用方法および出力の解析方法は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

索引の使用

索引は、ROWID を使って、表の列のそれぞれの値をその値が入っている行に対応付けます。索引は CREATE INDEX 文で作成します。詳細は『Oracle8*i* SQL リファレンス』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、索引を使う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、全体走査による方法、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、その索引を使います。どの列に索引を付けるかを選択するためのガイドラインは、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。

行レベル・ロックの利用

デフォルトでは、Oracle は表レベルではなく行レベルでデータをロックします。行レベルでロックすると、複数のユーザーが同一の表内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。表のロックの詳細は、3-27 ページの「LOCK TABLE 文の使用方法」の「LOCK TABLE 文の使用」を参照してください。

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。アプリケーションを表レベル・ロックで運用している場合は、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・ロックは使わないようにします。

不要な解析の排除

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う必要があります。

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

これらのオプションは、暗黙的なカーソルおよび明示的なカーソル、カーソル・キャッシュ、プライベート SQL 領域に影響します。

注意: カーソル・キャッシュの統計は、ORACA を使用して取得できます。詳細は 8-35 ページの「Oracle 通信領域の使用」を参照してください。

明示的なカーソルの操作

カーソルには暗黙的なカーソルおよび明示的なカーソルの 2 種類があることに注意してください（2-8 ページの「エラーおよび警告」を参照）。Oracle はデータ定義文およびデータ操作文のすべてについて暗黙的にカーソルを宣言します。ただし、複数の行を戻す問合せについては、ユーザーが明示的にカーソルを宣言（またはホスト配列を使用）しなければなりません。DECLARE CURSOR 文を使うと、明示的なカーソルを宣言できます。明示的なカーソルのオープンとクローズをどのように扱うかは、パフォーマンスに影響します。

アクティブ・セットの再評価が必要なときは、そのカーソルを再度 OPEN するだけでかまいません。OPEN 文では任意の新しいホスト変数値が使用されます。カーソルをオープンする前にクローズしなければ処理時間を節約できます。

注意: パフォーマンスをチューニングしやすいように、プリコンパイラではすでにオープンされているカーソルを再オープンできます。ただし、これは ANSI/ISO の埋込み SQL 規格に対する Oracle 拡張機能です。したがって、MODE=ANSI のときは、カーソルを再オープンする前にクローズする必要があります。

カーソルの OPEN によって取得したリソース（メモリーおよびロック）を解放するときにだけそのカーソルを 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クローズしなければなりません。

カーソルの制御

通常、明示的に宣言したカーソルを制御する方法は次の 3 つです。

- DECLARE および OPEN、CLOSE 文を使用する。
- PREPARE および DECLARE、OPEN、CLOSE 文を使用する。

- MODE=ANSI の場合、COMMIT によってカーソルをクローズする。

最初の方法を使う場合は、不要な解析に注意する必要があります。OPEN 文で解析を実行するのは、カーソルをクローズしたかまだ 1 度もオープンしていないために、解析された文を使用できないときだけです。したがって、プログラムはカーソルを宣言し、ホスト変数の値が変わるたびにこれを再度オープンし、この SQL 文が不要となったときにだけクローズするようにしなければなりません。

2 番目の方法（動的 SQL 方法 3 および方法 4）を使う場合は、PREPARE 文で解析を実行し、この解析された文は CLOSE 文を実行するまで使用できます。プログラムは、SQL 文を PREPARE してカーソルを宣言し、ホスト変数の値が変わるたびに、カーソルを再オープンします。また、SQL 文が変わった場合には SQL 文の再 PREPARE とカーソルの再オープンを行います。カーソルをクローズするのは、その SQL 文が不要となった場合だけです。

OPEN 文および CLOSE 文をループの中に指定するのは、できるかぎり避けてください。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指定しなければなりません。ANSI では、カーソルを再度オープンする前にクローズ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
     END-EXEC.
     MOVE 0 TO SALARY.
 TOP.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
 LOOP.
     EXEC SQL FETCH EMP-CURSOR INTO ....
     ...
         IF SQLCODE = 0
              GO TO LOOP
          ELSE
              ADD 1000 TO SALARY
          END-IF.
     EXEC SQL CLOSE EMP-CURSOR END-EXEC.
     IF SALARY < 5000
         GO TO TOP.
```

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 文を外側のループの外に指定することによって、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。

```
 TOP.
     EXEC SQL OPEN EMP-CURSOR END-EXEC.
 LOOP.
     EXEC SQL FETCH EMP-CURSOR INTO ....
     ...
```

```
    IF SQLCODE = 0
        GO TO LOOP
     ELSE
        ADD 1000 TO SALARY
     END-IF.
IF SALARY < 5000
    GO TO TOP.
EXEC SQL CLOSE EMP-CURSOR END-EXEC.
```

カーソル管理オプションの使用

SQL 文は、その構成を変更しないかぎり、一度だけ解析すれば十分です。たとえば、ファイル A 内でカーソルを DECLARE してから、ファイル B 内でそのカーソルを OPEN することはできません。HOLD_CURSOR、および RELEASE_CURSOR、MAXOPENCURSORS オプションによって、SQL 文の解析および再解析を Oracle がどのように管理するかを制御できます。明示的なカーソルを宣言すると、解析を最大限に制御できます。

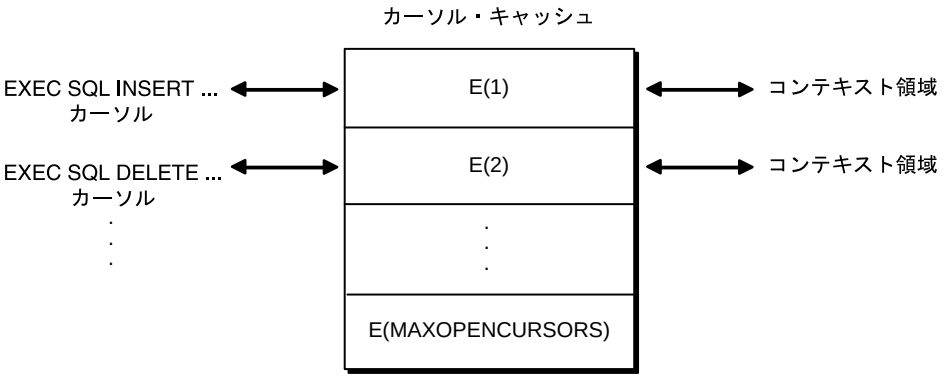
プライベート SQL 領域とカーソル・キャッシュ

データ操作文を実行すると、その文に対応している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。カーソル・キャッシュとはカーソル管理のために使わ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、Oracle が実行時に動的に作成する作業領域です。この領域には、解析された SQL 文、ホスト変数のアドレス、その文の処理に必要なその他の情報が保存されます。明示的なカーソルを使うと、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 D-2 は、プログラムで INSERT および DELETE が実行された後のカーソル・キャッシュを表しています。

図 D-2　カーソル・キャッシュでリンクされたカーソル



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、空きのキャッシュ・エントリがない場合、Oracle はエントリを再利用しようとします。再利用の可能性は HOLD_CURSOR と RELEASE_CURSOR の値によって決まり、また明示的カーソルの場合には、カーソル自身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、Oracle は再利用可能とマークされている最初のキャッシュ・エントリを使います。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は、Oracle はそれをもう一度解析し、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、Oracle は追加のキャッシュ・エントリを割り当てます。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。Oracle は、空きメモリーがなくなるか OPEN_CURSORS で設定された限界に達するまで、必要に応じてキャッシュ・エントリの割当てを続行します。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うことになります。

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、Oracle がその文を実行してカーソルがクローズされた後、プリコンパイラはこのカーソルとカーソル・キャッシュ間のリンクに使用可能の印を付けます。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。しかし、PREPARE したカーソルは実行状態のままでなければならないので、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、Oracle がその SQL 文を実行してカーソルがクローズされた後、プライベート SQL 領域が自動的に解放され、解析した文はなくなります。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。

データ操作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、データ操作文に RELEASE_CURSOR=YES を指定してください。これにより、データ操作文が得る解析ロックと、データ定義文が得る排他ロックとの間の対立が回避されます。

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 で HOLD_CURSOR=YES を上書きするため、Oracle では SQL 文を実行する前にプライベート SQL 領域のためにメモリーを再度割り当て、この SQL 文を再解析する必要があります。

ただし、RELEASE_CURSOR=YES を指定した場合は、Oracle は SQL 文と PL/SQL ブロックの解析した表現を共有 SQL キャッシュに入れておくため、これ以上再解析で処理する必要がないこともあります。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。

実行回数の多い場合

プライベート SQL 領域には SQL 文の実行に必要なすべての情報が格納されるため、頻繁に実行される SQL 文では、そのプライベート SQL 領域と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れば、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 の場合、Oracle が SQL 文を実行し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、解析された文と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、不要な再解析を避けるために、アクティブな状態にしておきたい SQL 文で有効です。

HOLD_CURSOR=YES で、RELEASE_CURSOR=NO（デフォルト値）の場合、Oracle が 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なので、頻繁に実行する SQL 文の場合に有効です。

埋込み PL/SQL に関する考慮事項

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。

注意：

デフォルトの HOLD_CURSOR=YES および RELEASE_CURSOR=NO を使用した場合、Oracle の旧バージョンで SQL 文を実行すると、解析された文は実行後も使用可能です。これと同じ条件で Oracle8*i* で SQL 文を実行すると、解析された文は、時間が経過して共有 SQL キャッシュから出されると使用できなくなります。通常、これが問題となることはありませんが、SQL 文が再解析される前に参照されたオブジェクトの定義が変更になると、予期しない結果をもたらす場合があります。

パラメータの相互作用

表 D-1 は、HOLD_CURSOR と RELEASE_CURSOR の相互関係を示しています。HOLD_CURSOR=NO を指定すると、RELEASE_CURSOR=NO は変更され、RELEASE_CURSOR=YES を指定すると、HOLD_CURSOR=YES が変更されることに注意してください。

表 D-1　HOLD_CURSOR および RELEASE _CURSOR の相互関係

HOLD_CURSOR	RELEASE_CURSOR	リンク
NO	NO	再利用可能の印
YES	NO	維持
NO	YES	ただちに削除
YES	YES	ただちに削除

E

構文および意味検査

埋込み SQL 文および PL/SQL ブロックの構文および意味をチェックすることによって、Oracle 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支援します。この付録では、プリコンパイラ・オプションの SQLCHECK を使って検査の種類および範囲を制御する方法を説明します。

この章は次のトピックで構成されています。

- 構文および意味検査とは？
- 検査の種類および範囲の制御
- SQLCHECK=SEMANTICS の指定

構文および意味検査とは？

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、構文の検査はキーワード、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。

```
* --  misspelled keyword WHERE
     EXEC SQL DELETE FROM EMP WERE DEPTNO = 20 END-EXEC.
* --  missing parentheses around column names COMM and SAL
     EXEC SQL
         INSERT INTO EMP COMM, SAL VALUES (NULL, 1500)
     END-EXEC.
```

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。つまり、意味検査では、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正しい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。

```
* --  nonexistent table, EMPP
     EXEC SQL DELETE FROM EMPP WHERE DEPTNO = 20 END-EXEC.
* --  undeclared host variable, EMP-NAME
     EXEC SQL SELECT * FROM EMP WHERE ENAME = :EMP-NAME END-EXEC.
```

SQL 構文と意味の規則の定義は、『Oracle8*i* SQL リファレンス』を参照してください。

検査の種類および範囲の制御

コマンド行でプリコンパイラ・オプションの SQLCHECK を指定することによって、検査の種類および範囲を制御します。SQLCHECK では、検査の種類は、構文、意味、その両方の 3 種類があります。チェックの範囲には、データ操作文と PL/SQL ブロックを含めることができます。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 で動的 SQL 文はチェックできません。

SQLCHECK について次の値を指定できます。

- SEMANTICS | FULL
- SYNTAX | LIMITED

値 SEMANTICS と FULL は等価です。また、SYNTAX と LIMITED も等価です。デフォルト値は SYNTAX です。

SQLCHECK=SEMANTICS の指定

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。

- INSERT 文や UPDATE 文などのデータ操作文
- PL/SQL ブロック

ただし、リモート・データ操作文（AT *db_name* 句を使用したデータ操作文）については、構文だけをチェックします。

プリコンパイラは、意味検査に必要な情報を、埋め込まれた DECLARE TABLE 文から取得します。また、オプション USERID が指定されている場合は、データベースに接続してデータ・ディクショナリにアクセスすることによっ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ている表がすべて DECLARE TABLE 文で定義されている場合は、データベースに接続する必要はありません。

データベースに接続してデータ・ディクショナリにアクセスしても見つからない情報があった場合は、DECLARE TABLE 文を使って不足している情報を提供しなければなりません。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。

データ操作文をチェックする場合、プリコンパイラは『Oracle8*i* SQL リファレンス』に掲載されている Oracle8*i* の構文規則を使いますが、意味検査にはさらに厳密な規則を使います。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。

新しいプログラムをプリコンパイルするときは、SQLCHECK=SEMANTICS を指定してください。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。

意味検査を使用可能にする

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 に接続し、そのデータ・ディクショナリにアクセスする
- 埋め込まれた DECLARE TABLE 文を使う

Oracle への接続

意味検査を行うために、プリコンパイラはホスト・プログラム内で参照される表およびビューの定義が保存されているデータベースに接続できます。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探します。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。

必要な情報の一部をデータ・ディクショナリ内から見つけられない場合（たとえば、プログラムが未作成の表を参照するときなど）は、DECLARE TABLE 文を使って足りない情報を指定する必要があります。

データベースに接続するには、次の構文を使って、コマンド行で USERID オプションを指定します。

```
USERID=username/password
```

この *username* および *password* は有効な Oracle8*i* 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=/
```

プリコンパイラは、自動的に次のユーザー ID を使ってデータベースに接続しようとします。

```
<prefix><username>
```

prefix は初期化パラメータ OS_AUTHENT_PREFIX の値（デフォルト値は OPS$）、*username* はオペレーティング・システムのユーザー名またはタスク名です。

（データベースが使用不能などの理由で）接続に失敗すると、プリコンパイラは処理を停止し、エラー・メッセージを発行します。オプション USERID が指定されないと、プリコンパイラは埋め込まれた DECLARE TABLE 文から必要な情報を取得することになります。

DECLARE TABLE の使用

プリコンパイラは、データベースに接続せずに意味検査を行います。プリコンパイラは、意味検査に必要な表やビューに関する情報を、埋込み DECLARE TABLE ディレクティブから取得する必要があります。つまり、データ操作文または PL/SQL ブロック内で参照する表をすべて DECLARE TABLE 文内で定義しなければりません。

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], ...)
END-EXEC.
```

この *expr* は CREATE TABLE 文で列のデフォルト値として使用できる任意の式です。*col_datatype* は、Oracle 列宣言です。使用できるのは整数だけで、式は使用できません。詳細は F-29 ページの「DECLARE TABLE（Oracle 埋込み SQL ディレクティブ）」を参照してください。

DECLARE TABLE を使っ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。

F

埋込み SQL 文とプリコンパイラ・ディレクティブ

この付録では、SQL92 の埋込み SQL 文とディレクティブ、および Oracle8*i* の埋込み SQL 拡張要素について説明します。これらの文とディレクティブをソース・コードで使うときは、キーワード EXEC SQL を前に付けます。

注意 : この付録で説明するのは、非埋込み SQL と構文が異なる文だけです。非埋込み SQL 文の詳細は、『Oracle8*i* SQL リファレンス』を参照してください。

この付録の構成は、次のとおりです。

- 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）
- DELETE（実行可能な埋込み SQL）
- DESCRIBE（実行可能な埋込み SQL）
- DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）
- EXECUTE ...END-EXEC（実行可能な埋込み SQL 拡張要素）
- EXECUTE（実行可能な埋込み SQL）
- EXECUTE DESCRIPTOR（実行可能な埋込み SQL）
- EXECUTE IMMEDIATE（実行可能な埋込み SQL）
- FETCH（実行可能な埋込み SQL）
- FETCH DESCRIPTOR（実行可能な埋込み SQL）
- FREE（実行可能な埋込み SQL 拡張要素）
- GET DESCRIPTOR（実行可能な埋込み SQL）
- INSERT（実行可能な埋込み SQL）
- LOB APPEND（実行可能な埋込み SQL 拡張要素）
- LOB ASSIGN（実行可能な埋込み SQL 拡張要素）
- LOB CLOSE（実行可能な埋込み SQL 拡張要素）
- LOB COPY（実行可能な埋込み SQL 拡張要素）
- LOB CREATE TEMPORARY（実行可能な埋込み SQL 拡張要素）
- LOB DESCRIBE（実行可能な埋込み SQL 拡張要素）
- LOB DISABLE BUFFERING（実行可能な埋込み SQL 拡張要素）
- LOB ENABLE BUFFERING（実行可能な埋込み SQL 拡張要素）
- LOB ERASE（実行可能な埋込み SQL 拡張要素）
- LOB FILE CLOSE ALL（実行可能な埋込み SQL 拡張要素）
- LOB FILE SET（実行可能な埋込み SQL 拡張要素）
- LOB FLUSH BUFFER（実行可能な埋込み SQL 拡張要素）
- LOB FREE TEMPORARY（実行可能な埋込み SQL 拡張要素）
- LOB LOAD（実行可能な埋込み SQL 拡張要素）
- LOB OPEN（実行可能な埋込み SQL 拡張要素）

- LOB READ（実行可能な埋込み SQL 拡張要素）
- LOB TRIM（実行可能な埋込み SQL 拡張要素）
- LOB WRITE（実行可能な埋込み SQL 拡張要素）
- OPEN（実行可能な埋込み SQL）
- OPEN DESCRIPTOR（実行可能な埋込み SQL）
- PREPARE（実行可能な埋込み SQL）
- ROLLBACK（実行可能な埋込み SQL）
- SAVEPOINT（実行可能な埋込み SQL）
- SET DESCRIPTOR（実行可能な埋込み SQL）
- SELECT（実行可能な埋込み SQL）
- UPDATE（実行可能な埋込み SQL）
- VAR（Oracle 埋込み SQL ディレクティブ）
- WHENEVER（埋込み SQL のディレクティブ）

プリコンパイラのディレクティブおよび埋込み SQL 文の概要

埋込み SQL 文は、DDL および DML、トランザクション制御文をプロシージャ型言語プログラムに挿入します。Oracle プリコンパイラでは、埋込み SQL をサポートしています。表 F-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 F-1 のタイプの欄は、ソース / 型という形式で、次のように記載されています。

ソース	SQL92 標準の SQL（S）または Oracle の拡張要素（O）
型	実行可能（E）文またはディレクティブ（D）

表 F-1　プリコンパイラのディレクティブおよび埋込み SQL コマンドと句

EXEC SQL 文	ソース / 型	用途
ALLOCATE	O / E	カーソル変数にメモリーを割り当てます。
ALLOCATE DESCRIPTOR	S / E	記述子を ANSI 動的 SQL に割り当てます。
CALL	S / E	ストアド・プロシージャをコールします。
CLOSE	S / E	保持されているリソースを解放し、カーソルを使用禁止にします。
COMMIT	S / E	データベースの変更をすべて確定します。
CONNECT	O / E	データベースのインスタンスにログインします。
CONTEXT ALLOCATE	O / E	メモリーを SQLLIB 実行時コンテキストに割り当てます。
CONTEXT FREE	O / E	メモリーを SQLLIB 実行時コンテキストから解放します。
CONTEXT USE	O / E	SQLLIB 実行時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	Oracle プリコンパイラが埋込み SQL 文の意味上のチェックに使う表の構造を宣言します。
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 文の情報を取得し、記述子に格納します。

表 F-1　プリコンパイラのディレクティブおよび埋込み SQL コマンドと句 (続き)

EXEC SQL 文	ソース / 型	用途
EXECUTE...END-EXEC	O/E	無名 PL/SQL ブロックを実行します。
EXECUTE	S/E	準備済みの動的 SQL 文を実行します。
EXECUTE DESCRIPTOR	S/E	ANSI 動的 SQL を使って準備済みの文を実行します。
EXECUTE IMMEDIATE	S/E	ホスト変数をもたない 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 動的 SQL を使う問合せにより、選択された行を取り出します。
FREE	S/E	カーソルが使う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	テンポラリ LOB を作成します。
LOB DESCRIBE	O/E	LOB の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 バッファリングを使用可能にします。
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 BFILE をすべてクローズします。
LOB FILE SET	O/E	BFILE ロケータに DIRECTORY および FILENAME を設定します。
LOB FLUSH BUFFER	O/E	LOB 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 ロケータの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	読込みまたは読込み / 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。

表 F-1　プリコンパイラのディレクティブおよび埋込み SQL コマンドと句 (続き)

EXEC SQL 文	ソース / 型	用途
LOB READ	O/E	LOB または BFILE の全部または一部をバッファにコピーします。
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。
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	ANSI 動的 SQL のカーソルに対応付けられた問合せを実行します。
PREPARE	S/E	動的 SQL 文を解析します。
ROLLBACK	S/E	現行のトランザクションを終了し、すべての変更を破棄します。
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 つまたは複数の表またはビュー、スナップショットからデータを取り出します。
SET DESCRIPTOR	S/E	ホスト変数から ANSI SQL 記述子領域に情報を設定します。
UPDATE	S/E	表またはビュー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 Oracle8*i* 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。

文記述子について

ディレクティブおよび句はアルファベット順で示します。各コマンドの説明には、次の項目があります。

用途	コマンドの基本的な用途を示します。
前提条件	必要な権限と、文を使う前に実行しなければならない手順を示します。特記していない限り、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	文のキーワードおよびパラメータの構文図を示します。
キーワードおよびパラメータ	各キーワードとパラメータの用途を示します。
使用上の注意	文の使用方法と条件を示します。
例	文の例文を示します。
関連項目	関連する文および句、このマニュアル内の項目を示します。

構文図の読みかた

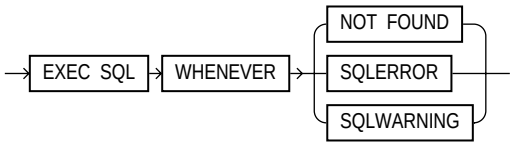
埋込み SQL の構文は、わかりやすいように構文図を使用して説明します。構文図とは、有効な構文を示す図です。

構文図は、左から右に矢印が指す方向にたどってください。

文のキーワードは、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。パラメータには変数が使われます。演算子およびデリミタ、終了記号は、円の中に表示されています。

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。

キーワードまたは演算子、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、縦方向に自由に進み、それから横線に戻ることができます。



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 ...
```

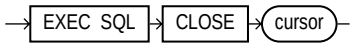
```
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

文終了記号

Pro*COBOL EXEC SQL の構文図はすべて、文がトークン *END-EXEC* で終了するものと解釈してください。

必須のキーワードとパラメータ

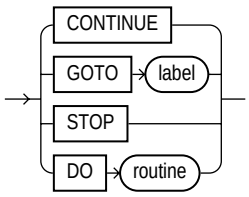
必須のキーワードとパラメータは単一で、あるいは代替の選択肢を縦に並べて示します。必須のキーワードまたはパラメータが 1 つしかない場合は、メイン・パス、つまり現在たどっている横線上に示します。次の例では、cursor は必須パラメータです。



したがって、*EMPCURSOR* 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。

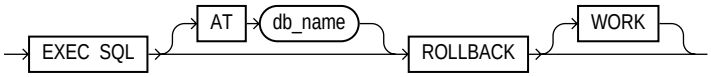
```
EXEC SQL CLOSE EMPCURSOR END-EXEC.
```

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 1 つが必須です。つまり、キーワードやパラメータを 1 つ選択しなければなりませんが、メイン・パスのちょうど上にあるものとは限りません。次の例では、4 つのアクションのうち 1 つを選択しなければなりません。



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・パスの上に縦に並べられている場合は、オプションです。次の例では、縦方向にたどらないで、メイン・パスを続けることができます。

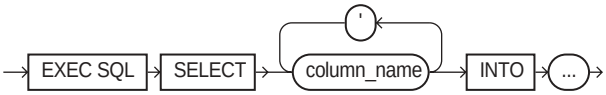


この図では、*oracle2* という名前のデータベースが存在する場合、次の文はすべて有効です。

```
EXEC SQL ROLLBACK END-EXEC.
EXEC SQL ROLLBACK WORK END-EXEC.
EXEC SQL AT ORACLE2 ROLLBACK END-EXEC.
```

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。このため、列名を 1 つ選択した後で、繰り返し戻って別の列名を選択できます。

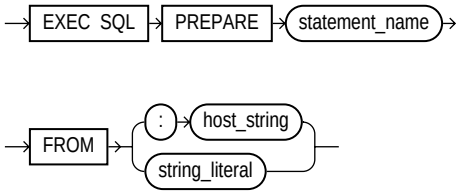


DEBIT および CREDIT、BALANCE が列名の場合、この図では次の文がすべて有効です。

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

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 :host_string END-EXEC.
```

データベース・オブジェクト

表や列などの Oracle の識別子の名前は、30 文字以内でなければなりません。先頭文字は英文字でなければなりませんが、残りの文字には、英文字、数字、ドル記号（$）、ポンド記号（#）、アンダースコア（_）を任意に組み合わせて使用できます。

ただし、Oracle 識別子を引用符（"）で囲むと、有効な文字を任意に組み合わせて使うことができます。この場合、空白は有効な文字ですが、引用符は無効です。

Oracle の識別子は、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。

ALLOCATE（実行可能な埋込み SQL 拡張要素）

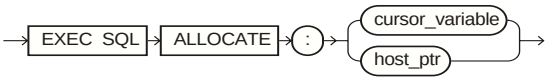
用途

PL/SQL ブロックで参照するカーソル変数を割り当てます。

前提条件

SQL-CURSOR 型のカーソル変数（第 6 章の「埋込み PL/SQL」を参照）は、カーソル変数のメモリーを割り当てる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

db_name	データベース接続名が格納される NULL 終了文字列です。データベース接続名は CONNECT 文で事前に設定されます。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見なされます。
cursor_variable	SQL_CURSOR 型のカーソル変数です。
host_ptr	LOB 用の ROWID、SQL-BLOB、SQL-CLOB または SQL-NCLOB の SQL_ROWID 型の変数です。

使用上の注意

カーソルが静的であるのに対して、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。

このコマンドの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』および『Oracle8*i* SQL リファレンス』を参照してください。

例

この例では、ALLOCATE 文の使用方法を示します。

```
      ...
 01   EMP-CUR        SQL-CURSOR.
 01   EMP-REC.
      ...
      EXEC SQL ALLOCATE :EMP-CUR END-EXEC.
      ...
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-14 ページの EXECUTE（実行可能な埋込み SQL）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-49 ページの FREE（実行可能な埋込み SQL 拡張要素）

ALLOCATE DESCRIPTOR（実行可能な埋込み SQL）

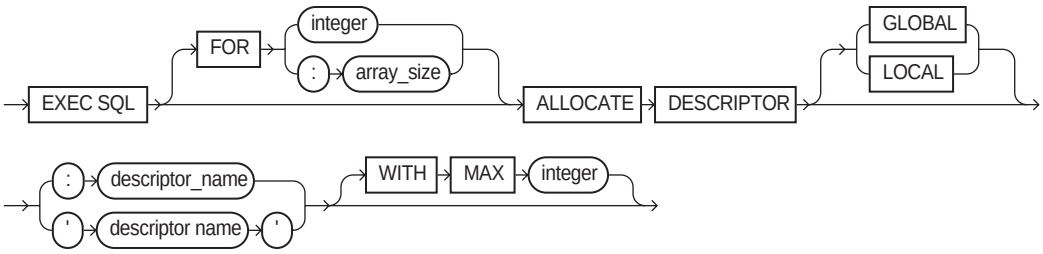
用途

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

array_size	処理される行数を格納するホスト変数です。
integer	処理される行数です。
descriptor_name	ANSI 記述子名を格納するホスト変数です。
descriptor name	ANSI 記述子の名前です。
GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。
WITH MAX *integer*	ホスト変数の最大数です。デフォルトは 100 です。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使います。このコマンドの詳細は、10-12 ページの「ALLOCATE DESCRIPTOR」を参照してください。

例

```
EXEC SQL
```

```
  FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25
END-EXEC.
```

関連項目

F-35 ページの DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）

F-50 ページの GET DESCRIPTOR（実行可能な埋込み SQL）

F-80 ページの SET DESCRIPTOR（実行可能な埋込み SQL）

CALL（実行可能な埋込み SQL）

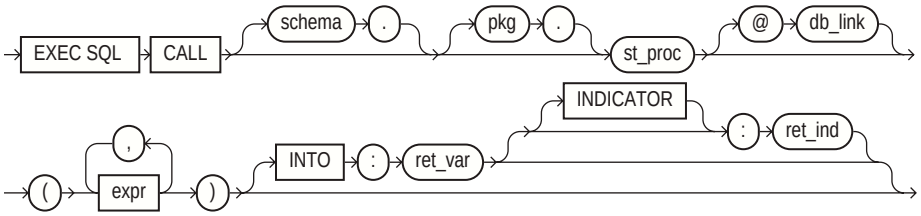
用途

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なければなりません。

構文



キーワードおよびパラメータ

schema	プロシージャを格納するスキーマです。schema を省略した場合、Oracle8*i* はプロシージャが自スキーマ内にあると見なします。
pkg	プロシージャが格納されているパッケージです。
st_proc	コールするストアド・プロシージャです。

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ーベース・リンクの、完全または一部の名前です。データベース・リンク参照の情報は、『Oracle8*i* SQL 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です。
ret_var	関数からの戻り値を受け取るホスト変数です。
ret_ind	ret_var 用の標識変数です。

使用上の注意

この文についての詳細は、6-22 ページのストアド 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。

ストアド・プロシージャの詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』（「外部ルーチン」の章）を参照してください。

例

```
 ...
     05  EMP-NAME      PIC X(10) VARYING.
     05  EMP-NUMBER    PIC S9(4) COMP VALUE ZERO.
     05  SALARY        PIC S9(5)V99 COMP-3 VALUE ZERO.
...
      05  D-EMP-NUMBER  PIC 9(4).
...
      ACCEPT D-EMP-NUMBER.
      EXEC SQL
        CALL mypkge.getsal(:EMP-NUMBER, :D-EMP-NUMBER, :EMP-NAME) INTO :SALARY
      END-EXEC.
...
```

関連項目

なし

CLOSE（実行可能な埋込み SQL）

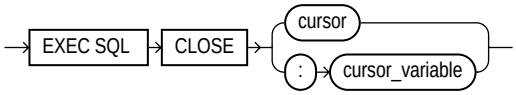
用途

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

カーソルまたはカーソル変数をオープンしていなければなりません。また MODE=ANSI でなければなりません。

構文



キーワードおよびパラメータ

cursor	クローズするカーソルです。
cusor_variable	クローズするカーソル変数です。

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、第 14 章の「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-CUR END-EXEC.
```

関連項目

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-67 ページの OPEN（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

COMMIT（実行可能な埋込み SQL）

用途

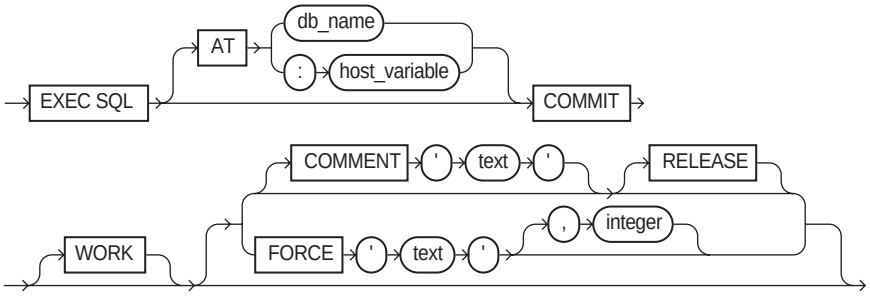
データベースの変更内容をすべて確定し、またオプションですべてのリソースを解放してデータベース・サーバーから切断し、現行トランザクションを終了します。

前提条件

現行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。

自分で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

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。次のいずれかを使いデータベースを指定します。

- *db_name*　以前の DECLARE DATABASE 文で宣言されているデータベース識別子または CONNECT 文で使用されるデータベース識別子です。
- *host_variable*　事前に宣言した *db_name* の値をもつホスト変数です。

この句を省略した場合、Oracle8*i* はデフォルトのデータベースに対して COMMIT 文を発行します。

WORK　標準 SQL への準拠のためだけにサポートされています。COMMIT 文と COMMIT WORK 文は同等です。

COMMENT　現行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 文字以内の引用リテラルで、トランザクションがインダウトになると、Oracle8*i* ではデータ・ディクショナリ・ビュー DBA_2PC_PENDING に、トランザクション ID と共に格納されます。

RELEASE　リソースをすべて解放し、アプリケーションを Oracle8*i* Server 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できます。トランザクションは、ローカルまたはグローバル・トランザクション ID を格納する '*text*' によって識別されます。インダウトの分散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 にお問い合わせください。また、オプションの *integer* を使い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現行の SCN を使いコミットされます。

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 文と RELEASE オプションを使い、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、Oracle8*i* は自動的に変更内容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、3-19 ページの「COMMIT 文の使用方法」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 SALESDB COMMIT RELEASE END-EXEC.
```

関連項目

F-73 ページの ROLLBACK（実行可能な埋込み SQL）

F-76 ページの SAVEPOINT（実行可能な埋込み SQL）

CONNECT（実行可能な埋込み SQL 拡張要素）

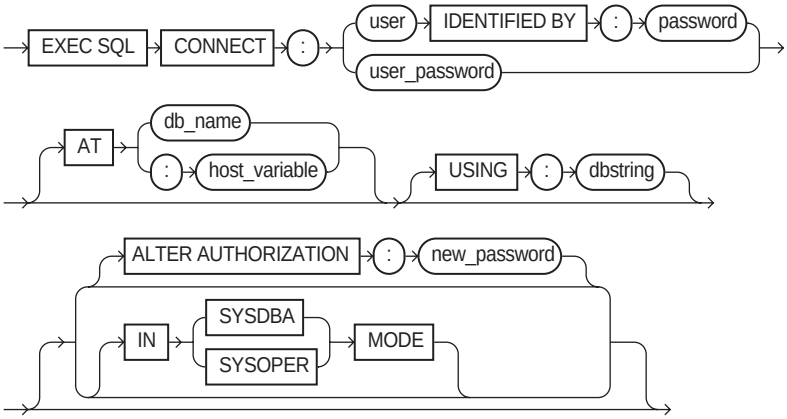
用途

Oracle8*i* のデータベースにログインします。

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

user
password
ユーザー名とパスワードを個別に指定します。

user_password
Oracle8*i* のユーザー名とパスワードをスラッシュ（/）で区切って指定する単一のホスト変数です。

Oracle8*i* でお使いのオペレーティング・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。次のいずれかを使いデータベースを指定します。

db_name
DECLARE DATABASE 文を使い事前に宣言したデータベース識別子です。

host_variable
事前に宣言した *db_name* の値をもつホスト変数です。

USING
デフォルト以外のデータベースへの接続に使います。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
新パスワードの文字列です。

IN SYSDBA MODE IN SYSOPER MODE	SYSDBA または SYSOPER システム権限で接続します。ALTER AUTHORIZATIO が使われているとき、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も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、3-11 ページの「アドバンス・コネクション・オプション」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
END-EXEC.
```

userid の値が *username* の値で、'SCOTT/TIGER' のように *password* が "/" で区分けされるこの文も使えます。

```
EXEC SQL CONNECT :USERID END-EXEC.
```

関連項目

F-15 ページの COMMIT（実行可能な埋込み SQL）

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-73 ページの ROLLBACK（実行可能な埋込み SQL）

CONTEXT ALLOCATE（実行可能な埋込み SQL 拡張要素）

用途

EXEC SQL CONTEXT USE 文で参照されている SQLLIB 実行時コンテキストを初期化します。

前提条件

実行時コンテキストは、SQL-CONTEXT 型で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

context メモリーが割り当てられる SQLLIB 実行時コンテキストです。

使用上の注意

この文の詳細は、4-33 ページの「ユーザー指定の実行時コンテキスト」を参照してください。

例

この例では、Pro*C / C++ 埋込み SQL プログラムで CONTEXT ALLOCATE 文の使用方法を示します。

```
EXEC SQL CONTEXT ALLOCATE :ctx1 END-EXEC.
```

関連項目

F-20 ページの CONTEXT FREE（実行可能な埋込み SQL 拡張要素）

F-21 ページの CONTEXT USE（Oracle 埋込み SQL ディレクティブ）

CONTEXT FREE（実行可能な埋込み SQL 拡張要素）

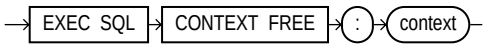
用途

実行時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 文を使って実行時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 文を使って、指定された実行時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

context メモリーの割当てを解除する、割当て済み実行時コンテキストです。

使用上の注意

この文の詳細は、4-33 ページの「ユーザー指定の実行時コンテキスト」を参照してください。

例

この例では、Pro*C / C++ 埋込み SQL プログラムで CONTEXT FREE 文を使用する方法を示します。

```
EXEC SQL CONTEXT FREE :ctx1 END-EXEC.
```

関連項目

F-19 ページの CONTEXT ALLOCATE（実行可能な埋込み SQL 拡張要素）

F-21 ページの CONTEXT USE（Oracle 埋込み SQL ディレクティブ）

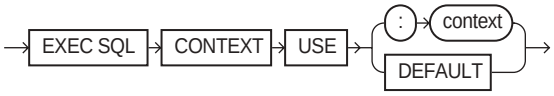
CONTEXT USE（Oracle 埋込み SQL ディレクティブ）

用途

後続の実行可能 SQL 文の前提条件で指定された SQLLIB 実行時コンテキストを使うように、プリコンパイラに指示します。

CONTEXT USE ディレクティブによって指定された実行時コンテキストは、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

context	後続の実行可能 SQL 文によって使われる、割当て済み実行時コンテキストです。たとえば、使用するコンテキストをソース・コードに指定したら（複数のコンテキストを割り当てることができます）、Oracle Server に接続し、コンテキストの有効範囲内でデータベースを操作できます。
DEFAULT	グローバル・コンテキストが使われるように指示します。

使用上の注意

この文は、EXEC SQL INCLUDE、EXEC ORACLE OPTION などの宣言文は無効です。EXEC SQL WHENEVER ディレクティブと同様に動作します。つまり、C の標準有効範囲ルールにかかわらず、指定されたソース・ファイル内でこの文とともに動作するすべての実行可能 SQL 文に対して有効です。

この文の詳細は、4-33 ページの「ユーザー指定の実行時コンテキスト」を参照してください。

例

この例では、Pro*C/C++ 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 END-EXEC.
```

関連項目

F-19 ページの CONTEXT ALLOCATE（実行可能な埋込み SQL 拡張要素）

F-20 ページの CONTEXT FREE（実行可能な埋込み SQL 拡張要素）

DEALLOCATE DESCRIPTOR（埋込み SQL 文）

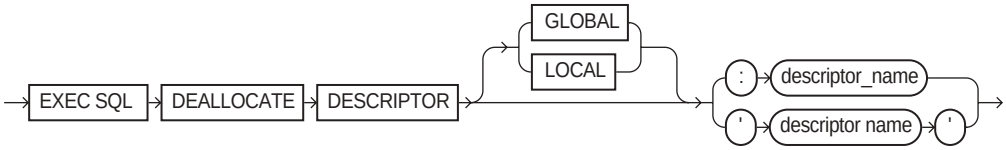
用途

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 文で指定された記述子は、ALLOCATE DESCRIPTOR 文を使っ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

GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。
descriptor_name	割当て済み ANSI 記述子名を格納するホスト変数です。
'descriptor name'	割当て済み ANSI 記述子の名前です。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。

この文の詳細は、10-13 ページの「DEALLOCATE DESCRIPTOR」を参照してください。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES'  END-EXEC.
```

関連項目

F-12 ページの ALLOCATE DESCRIPTOR（実行可能な埋込み SQL）

F-35 ページの DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）

F-50 ページの GET DESCRIPTOR（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

F-80 ページの SET DESCRIPTOR（実行可能な埋込み SQL）

DECLARE CURSOR（埋込み SQL ディレクティブ）

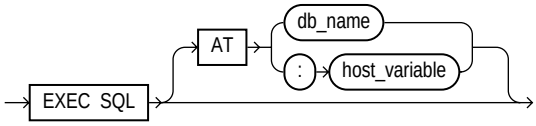
用途

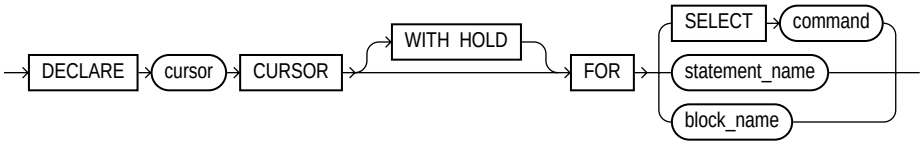
カーソルに名前を付け、それを SQL 文または PL/SQL ブロックに対応付けて宣言します。

前提条件

SQL 文または PL/SQL ブロックの識別子を使ってカーソルを対応付けるには、DECLARE STATEMENT 文を使ってこの識別子を事前に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

AT
: カーソルを宣言するデータベースを指定します。次のいずれかを使いデータベースを指定します。

- *db_name*: DECLARE DATABASE 文を使い事前に宣言したデータベース識別子です。
- *host_variable*: 事前に宣言された *db_name* 値のホスト変数です。

この句を省略した場合、Oracle8*i* 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です。

WITH HOLD
: カーソルは、COMMIT または ROLLBACK の実行後もオープンされたままです。UPDATE の場合は、カーソルを宣言してはいけません。

SELECT 文	カーソルに対応付ける SELECT 文です。直後の文に INTO 句を含めてはいけません。
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 文を使って事前に宣言しておく必要があります。

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。カーソル宣言の有効範囲はプリコンパイル・ユニット内全体になるため、各カーソルの名前は有効範囲内で一意でなければなりません。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 構文を使って参照できます。このとき、カーソルは OPEN 文を使ってオープンし、FETCH 文を使っ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、3-20 ページの「DECLARE CURSOR 文での WITH HOLD 句の使用」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 EMPCURSOR CURSOR
    FOR SELECT ENAME, EMPNO, JOB, SAL
    FROM EMP
    WHERE DEPTNO = :DEPTNO
    FOR UPDATE OF SAL
END-EXEC.
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-27 ページの DECLARE STATEMENT（埋込み SQL ディレクティブ）

F-30 ページの DELETE（実行可能な埋込み SQL）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-67 ページの OPEN（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

F-77 ページの SELECT（実行可能な埋込み SQL）

F-83 ページの UPDATE（実行可能な埋込み SQL）

DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

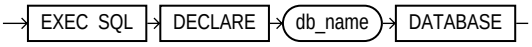
用途

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

デフォルト以外のデータベースのユーザー名にアクセスできなければなりません。

構文



キーワードおよびパラメータ

db_name　　デフォルト以外のデータベースに対して設定する識別子です。

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 句を使っ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、ECLARE DATABASE 文を使っ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、3-3 ページの「ユーザー名 / パスワードの使用方法」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 END-EXEC.
```

関連項目

F-15 ページの COMMIT（実行可能な埋込み SQL）

F-17 ページの CONNECT（実行可能な埋込み SQL 拡張要素）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-27 ページの DECLARE STATEMENT（埋込み SQL ディレクティブ）

F-30 ページの DELETE（実行可能な埋込み SQL）
F-38 ページの EXECUTE（実行可能な埋込み SQL）
F-42 ページの EXECUTE IMMEDIATE（実行可能な埋込み SQL）
F-52 ページの INSERT（実行可能な埋込み SQL）
F-77 ページの SELECT（実行可能な埋込み SQL）
F-83 ページの UPDATE（実行可能な埋込み SQL）

DECLARE STATEMENT（埋込み SQL ディレクティブ）

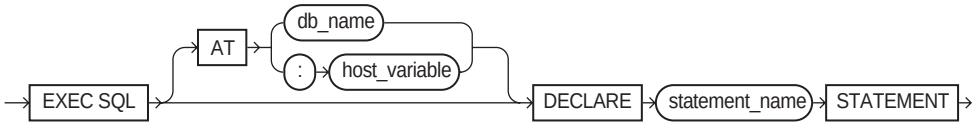
用途

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

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。次のいずれかを使ってデータベースを指定します。

db_name DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable 事前に宣言された *db_name* 値のホスト変数です。

この句を省略した場合、Oracle8*i* はデフォルトのデータベースに対して SQL 文または PL/SQL ブロックを宣言します。

statement_name

文または PL/SQL ブロックに対して宣言済みの識別子です。

使用上の注意

DECLARE STATEMENT 文を使って 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だけです。

文の宣言の有効範囲は、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、9-18 ページの「DECLARE」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 REMOTEDB
    DECLARE MYSTATEMENT STATEMENT
END-EXEC.
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING
END-EXEC.
EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

例 II

この例は、Pro*COBOL の埋込み SQL プログラムの一部です。ここでは DECLARE CURSOR 文が PREPARE 文の前にあるため、DECLARE STATEMENT 文が必要です。

```
EXEC SQL DECLARE MYSTATEMENT STATEMENT END-EXEC.
EXEC SQL DECLARE EMPCURSOR CURSOR FOR MYSTATEMENT END-EXEC.
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
...
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-67 ページの OPEN（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

DECLARE TABLE（Oracle 埋込み SQL ディレクティブ）

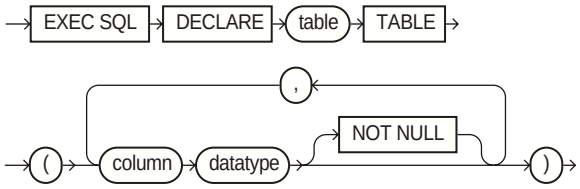
用途

オプションが SQLCHECK=SEMANTICS（または FULL）で、プリコンパイラで意味検査を行うため、データ型およびデフォルト値、NULL か NOT NULL 指定など、表またはビューの構造を定義します。

前提条件

なし

構文



キーワードおよびパラメータ

table	宣言した表の名前です。
column	*table* の列です。
datatype	*column* のデータ型です。Oracle8*i* データ型の詳細は、4-2 ページの「Oracle8i のデータ型」を参照してください。
NOT NULL	*column* に NULL を入れることができません。

使用上の注意

データ型の場合、長さおよび精度、スケールに使うことができるのは（式ではなく）整数だけです。この文の使用方法の詳細は、E-3 ページの「SQLCHECK=SEMANTICS の指定」を参照してください。

例

次の文では、PARTNO および BIN、QTY という列を含む 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)
END-EXEC.
```

関連項目

なし

DELETE（実行可能な埋込み SQL）

用途

表またはビューの実表から行を削除します。

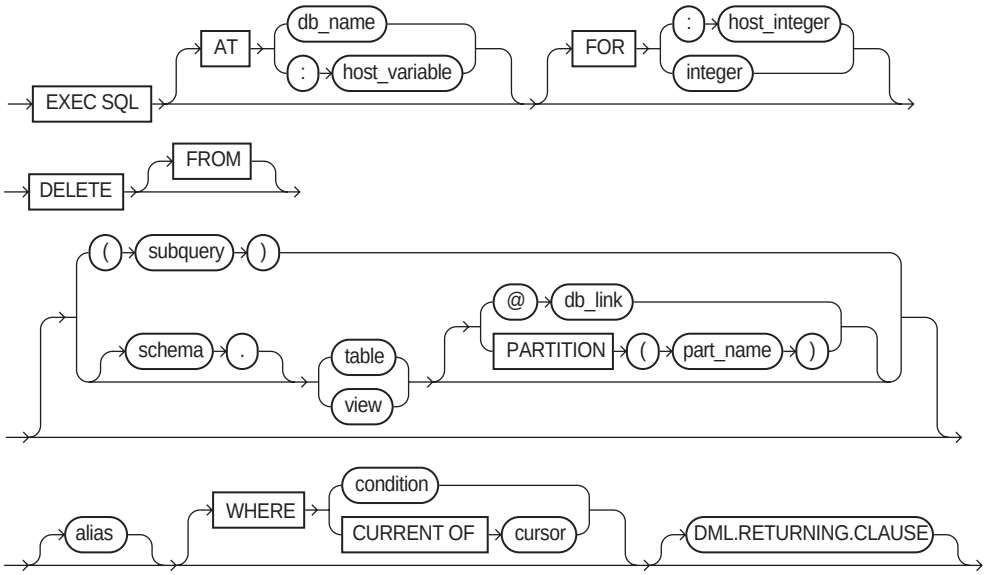
前提条件

表から行を削除するには、表が自分のスキーマ内にあるか、表に対して DELETE の権限を持っていなければなりません。

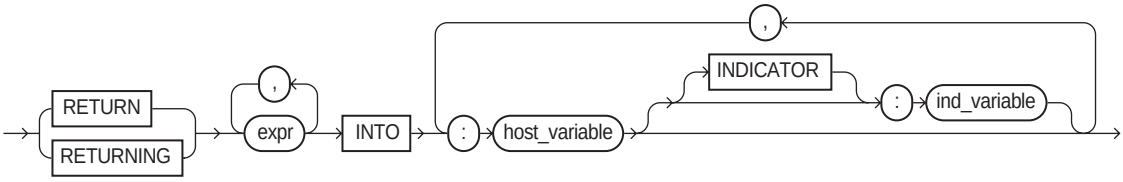
ビューの実表から行を削除するには、ビューが属するスキーマの所有者が、実表に対して DELETE の権限を持っていなければなりません。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対する DELETE の権限を付与されていなければなりません。

DELETE ANY TABLE システム権限を使って、表またはビューの実表ならどれでも行を削除できます。

構文



DML の Returning 句の構文を示します。



キーワードおよびパラメータ

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。次のいずれかを使ってデータベースを指定します。

db_name
DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable
事前に宣言された *db_name* 値のホスト変数です。

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。

host_integer
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。この句を省略すると、Oracle8*i* は最小の配列の各コンポーネントについて 1 回ずつ文を実行します。

schema

表またはビューを含むスキーマです。*schema* を省略した場合、Oracle8*i* は表またはビューが自スキーマ内にあるとみなします。

table view

行を削除する表の名前です。*view* を指定する場合、Oracle8*i* ではビューの実表から行を削除します。

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ーベース・リンクの完全または部分的な名前です。データベース・リンク参照の情報は、『Oracle8*i* SQL リファレンス』を参照してください。リモートの表またはビューから行を削除できるのは、Oracle8*i* を分散オプションで使っている場合だけです。

dblink を省略した場合、Oracle8 は表またはビューがローカル・データベース内にあると見なします。

part_name

表内のパーティションの名前です。

alias

表に割り当てられている別名です。別名は一般に、DELETE 文で相関問合せとともに使います。

WHERE

削除する行を指定します。

condition

条件を満たす行だけを削除します。条件には、ホスト変数およびオプションの標識変数を使用できます。『Oracle8*i* SQL リファレンス』の *condition* の構文説明を参照してください。

CURRENT OF

cursor によって最後にフェッチされた行だけを削除します。FOR UPDATE 句が明確に 1 つの表だけをロックしていない限り、*cursor* は結合を実行する SELECT 文に対応付けることができません。

この句を完全に省略した場合、Oracle8*i* は表またはビューからすべての行を削除します。

DML 戻し句

詳細は、5-9 ページの「DML 戻し句」を参照してください。

使用上の注意

WHERE 句のホスト変数は、すべてスカラーか、すべて配列でなければなりません。変数がスカラーの場合、Oracle8*i* は DELETE 文を 1 回だけ実行します。変数が配列の場合、Oracle8*i* 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、サイズが異なってもかまいません。この場合、Oracle8*i* が文を実行する回数は、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 *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、Oracle8*i* 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、Oracle8*i* 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、8-20 ページの「SQL 通信領域の使用」を参照してください。

DELETE 文ではコメントを使って指示またはヒントを Oracle8*i* のオプティマイザに渡すことができます。オプティマイザはヒントを使って文の実行計画を選択します。ヒントの詳細は、『Oracle8*i* チューニング』を参照してください。

例

この例では、Pro*COBOL プログラムで DELETE 文を使用する方法を示します。

```
 EXEC SQL DELETE FROM EMP
    WHERE DEPTNO = :DEPTNO
    AND JOB = :JOB
END-EXEC.
EXEC SQL DECLARE EMPCURSOR CURSOR
    FOR SELECT EMPNO, COMM
    FROM EMP
END-EXEC.
EXEC SQL OPEN EMPCURSOR END-EXEC.
EXEC SQL FETCH EMPCURSOR
    INTO :EMP-NUMBER, :COMMISSION
END-EXEC.
EXEC SQL DELETE FROM EMP
    WHERE CURRENT OF EMPCURSOR
END-EXEC.
```

関連項目

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-27 ページの DECLARE STATEMENT（埋込み SQL ディレクティブ）

DESCRIBE（実行可能な埋込み SQL）

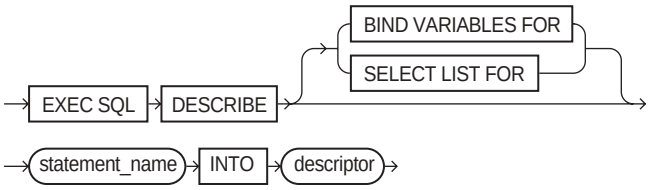
用途

Oracle 動的 SQL 文または PL/SQL ブロックのホスト変数の記述を保持する記述子を初期化します。

前提条件

埋込み SQL の PREPARE 文を使って、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

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 文を使って事前に準備した SQL 文または PL/SQL ブロックを指定します。
descriptor	初期化する記述子の名前です。

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 文が検出する変数の数は、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。一意に名前が付けられたプレースホルダの合計数ではありません。この文の詳細は、9-24 ページの「DESCRIBE 文」を参照してください。

例

この例では、Pro*COBOL 埋込み SQL プログラムで DESCRIBE 文を使用する方法を示します。

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
EXEC SQL DECLARE EMPCURSOR
    FOR SELECT EMPNO, ENAME, SAL, COMM
    FROM EMP
    WHERE DEPTNO = :DEPT-NUMBER
END-EXEC.
EXEC SQL DESCRIBE BIND VARIABLES FOR MYSTATEMENT
    INTO BINDDESCRIPTOR
END-EXEC.
EXEC SQL OPEN EMPCURSOR
    USING BINDDESCRIPTOR
END-EXEC.
EXEC SQL DESCRIBE SELECT LIST FOR MY-STATEMENT
    INTO SELECTDESCRIPTOR
END-EXEC.
EXEC SQL FETCH EMPCURSOR
    INTO SELECTDESCRIPTOR
END-EXEC.
```

関連項目

F-72 ページの PREPARE（実行可能な埋込み SQL）

DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）

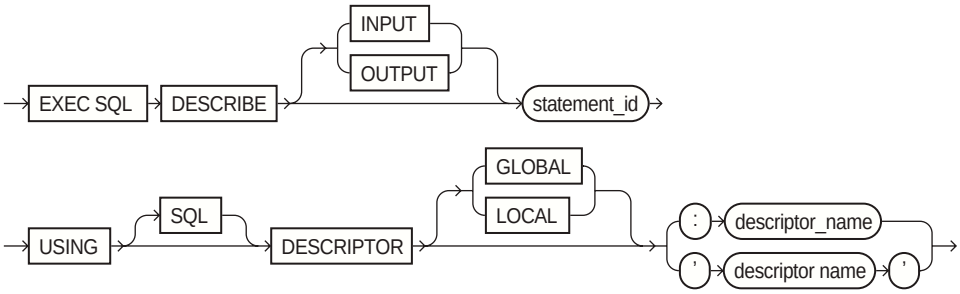
用途

ANSI SQL 文の情報を取得し、記述子に格納します。

前提条件

埋込み SQL の PREPARE 文を使って、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

statement_id	事前に準備されている SQL 文または PL/SQL ブロックの名前です。デフォルトは OUTPUT です。
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です。
'descriptor name'	記述子の名前です。
GLOBAL | LOCAL	LOCAL はデフォルトで、ファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。

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 だけがインプリメントされます。

DESCRIBE 文が検出する変数の数は、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。一意に名前が付けられたプレースホルダの合計数ではありません。この文の詳細は、第 9 章の「Oracle 動的 SQL」を参照してください。

例

```
EXEC SQL PREPARE s FROM :my_stament END-EXEC.
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' END-EXEC.
```

関連項目

F-12 ページの ALLOCATE DESCRIPTOR（実行可能な埋込み SQL）

F-22 ページの DEALLOCATE DESCRIPTOR（埋込み SQL 文）

F-50 ページの GET DESCRIPTOR（実行可能な埋込み SQL）

F-80 ページの PREPARE（実行可能な埋込み SQL）

F-80 ページの SET DESCRIPTOR（実行可能な埋込み SQL）

EXECUTE ...END-EXEC（実行可能な埋込み SQL 拡張要素）

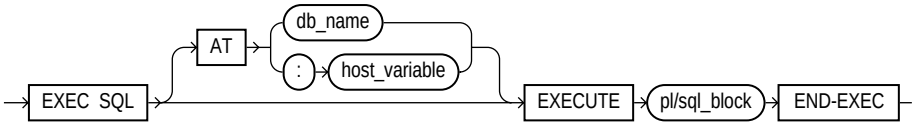
用途

Oracle Pro*COBOL 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

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。次のいずれかを使ってデータベースを指定します。

db_name DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable 事前に宣言された *db_name* 値のホスト変数です。

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。

pl/sql_block

PL/SQL ブロックの作成方法など、PL/SQL の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

END-EXEC

埋込み PL/SQL ブロックの後に配置しなければなりません。キーワード END-EXEC の後には、COBOL の文終了記号、"." を付ける必要があります。

使用上の注意

Oracle プリコンパイラは埋込み PL/SQL ブロックを 1 つの埋込み SQL 文のように扱うため、PL/SQL ブロックは Oracle プリコンパイラ・プログラムで SQL 文を埋め込める場所であればどこにでも埋め込めます。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、第 6 章の「埋込み PL/SQL」を参照してください。

例

Oracle プリコンパイラ・プログラムにこの EXECUTE 文を挿入すると、プログラムに PL/SQL ブロックが埋め込まれます。

```
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

F-42 ページの EXECUTE IMMEDIATE（実行可能な埋込み SQL）

EXECUTE（実行可能な埋込み SQL）

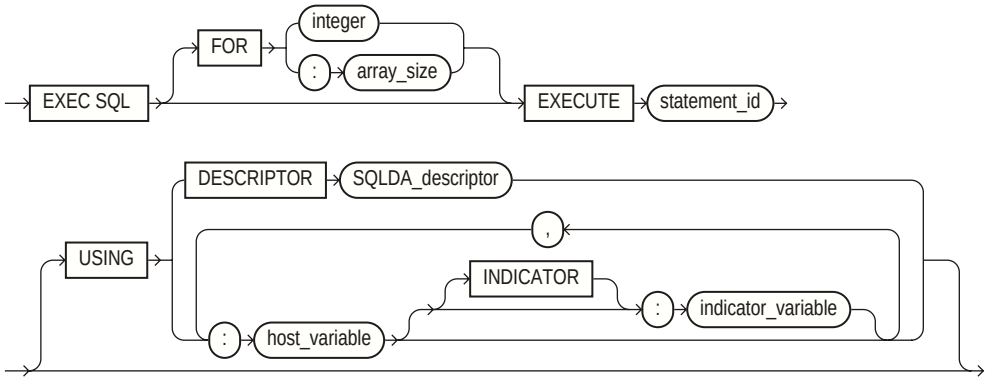
用途

Oracle 動的 SQL では、埋込み SQL の PREPARE 文によって準備済みの DELETE 文または INSERT 文、UPDATE 文、または PL/SQL ブロックを実行します。

前提条件

埋込み SQL の PREPARE 文を使って、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

FOR *:array_size*	処理される行の数を格納するホスト変数です。
FOR *integer*	処理される行数です。 USING 句が配列ホスト変数を含む場合に、文の実行回数を制限します。この句を省略した場合、Oracle8*i* は配列が最も小さい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です。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 文を使います。
USING DESCRIPTOR *SQLDA_descriptor*	Oracle 記述子を使います。 ANSI 記述子（INTO 句）とともには使えません。
USING	オプションの標識変数を使ってホスト変数のリストを指定します。Oracle8*i* 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、すべて配列でなければなりません。
host_variable	ホスト変数です。
indicator_variable	標識変数です。

使用上の注意

この文の詳細は、第 9 章の「Oracle 動的 SQL」を参照してください。

例

この例では、Pro*COBOL 埋込み SQL プログラムで DESCRIBE 文を使用する方法を示します。

```
EXEC SQL PREPARE MY-STATEMENT FROM MY-STRING END-EXEC.
EXEC SQL EXECUTE MY-STATEMENT USING :MY-VAR END-EXEC.
```

関連項目

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-72 ページの PREPARE（実行可能な埋込み SQL）

EXECUTE DESCRIPTOR（実行可能な埋込み SQL）

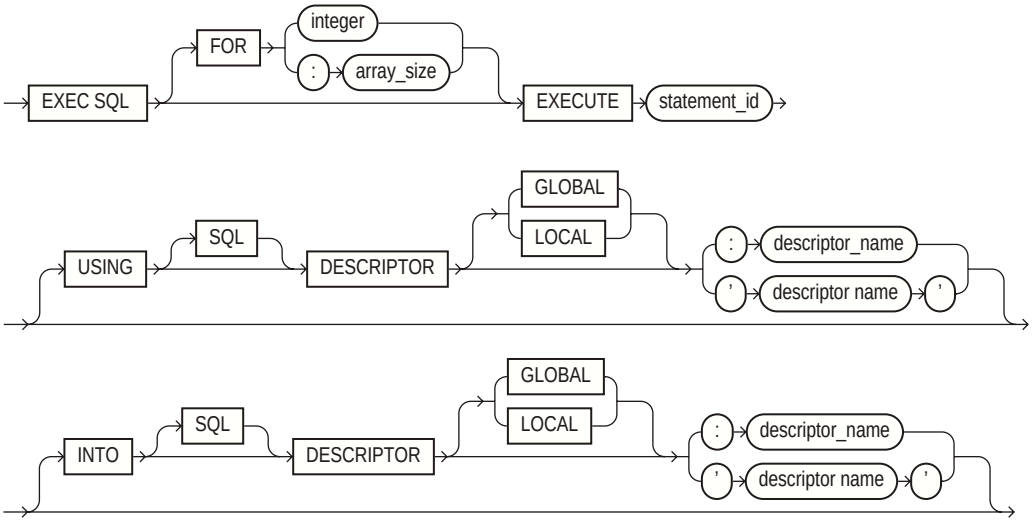
用途

ANSI SQL 方法 4 では、埋込み SQL の PREPARE 文によって準備済みの DELETE 文または INSERT 文、UPDATE 文、または PL/SQL ブロックを実行します。

前提条件

埋込み SQL の PREPARE 文を使って、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

FOR *:array_size*	処理される行の数を格納するホスト変数です。
FOR *integer*	処理される行数です。 文の実行回数を制限します。Oracle8*i* は配列が最も小さい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です。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 文を使います。
USING	ANSI 入力記述子です。
descriptor_name	入力記述子名を格納するホスト変数です。
descriptor name	入力記述子の名前です。
INTO	ANSI 出力記述子です。
descriptor_name	出力記述子名を格納するホスト変数です。
descriptor name	出力記述子の名前です。

GLOBAL | LOCAL　LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。

使用上の注意

この文の詳細は、10-22 ページの「EXECUTE」を参照してください。

例

ANSI 動的 SQL 方法 4 では、EXECUTE の INTO 句により SELECT の DML RETURNING を使うことができます。

```
EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' END-EXEC.
```

関連項目

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-72 ページの PREPARE（実行可能な埋込み SQL）

EXECUTE IMMEDIATE（実行可能な埋込み SQL）

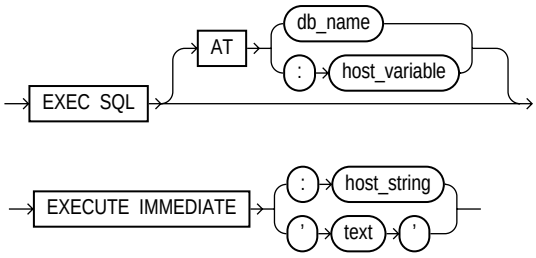
用途

ホスト変数を含まない DELETE 文または INSERT 文、UPDATE 文、または PL/SQL ブロックを準備し、実行します。

前提条件

なし

構文



キーワードおよびパラメータ

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。次のいずれかを使ってデータベースを指定します。

db_name DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable 事前に宣言された *db_name* 値のホスト変数です。

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。

host_string

実行する SQL 文または PL/SQL ブロックを値とするホスト変数です。

text

実行する SQL 文または PL/SQL ブロックを含むテキスト・リテラルです。引用符は省略できます。

SQL 文は、DELETE 文または INSERT 文、UPDATE 文のいずれかでなければなりません。

使用上の注意

EXECUTE IMMEDIATE 文を発行すると、Oracle8*i* 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、9-8 ページの「EXECUTE IMMEDIATE 文」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'
END-EXEC.
```

関連項目

F-72 ページの PREPARE（実行可能な埋込み SQL）

F-38 ページの EXECUTE（実行可能な埋込み SQL）

FETCH（実行可能な埋込み SQL）

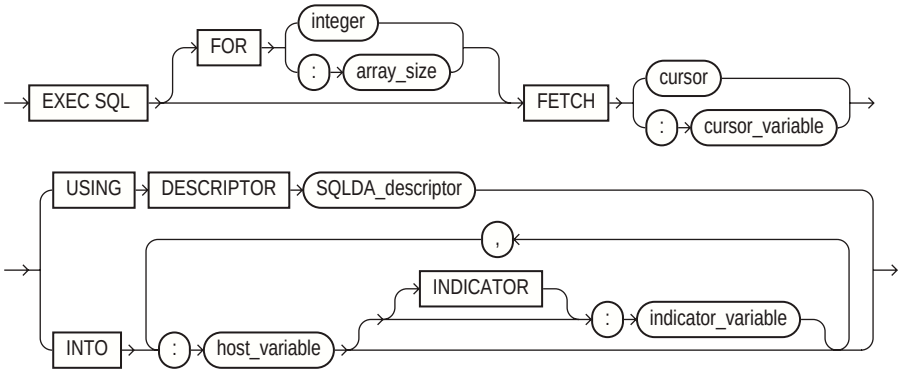
用途

Oracle 動的 SQL では、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 の詳細は、F-46 ページの「FETCH DESCRIPTOR（実行可能な埋込み SQL）」を参照してください。

前提条件

まず、OPEN 文を使っ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

FOR *:array_size*	処理される行の数を格納するホスト変数です。 処理される行数です。
FOR *integer*	配列ホスト変数を使う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、Oracle8*i* 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 文を使って宣言したカーソルです。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 文を使って割り当てたカーソル変数です。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なければなりません。
USING *SQLDA_variable*	DESCRIBE 文を使っ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 以外では使わないでください。カーソル変数を使っている場合は、USING 句は適用されません。
host_variable	データが戻されるホスト変数です。
i*ndicator_variable*	ホスト標識変数です。

使用上の注意

FETCH 文はアクティブ・セットの行を読み込み、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、標識変数の値は -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。データを受け取るホスト変数は、すべてスカラーか、すべて配列でなければなりません。スカラーの場合、Oracle8*i* は 1 行だけフェッチします。配列の場合、Oracle8*i* 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、Oracle8*i* がフェッチする行数は、次の値のうち低い方です。

- 最小の配列のサイズ
- オプションの FOR 句の *host_integer* の値

フェッチする行数は、実際に問合せを満たす行の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をすべて取り出したのではない場合、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り出すと、その次の FETCH ではエラー・コードが発生します。このエラー・コードは SQLCA の SQLCODE 要素に戻されます。

FETCH 文には AT 句がないので注意してください。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だけ進めます。すでにフェッチした行に戻りたい場合は、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。

例

この例では、Pro*COBOL 埋込み SQL プログラム内の FETCH 文を示します。

```
    EXEC SQL DECLARE EMPCURSOR CURSOR FOR
        SELECT JOB, SAL FROM EMP WHERE DEPTNO = 30
    END-EXEC.
    ...
    EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP.
    EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE1, :SALARY1 END-EXEC.
    EXEC SQL FETCH EMPCURSOR INTO :JOB-TITLE2, :SALARY2 END-EXEC.
    ...
    GO TO LOOP.
    ...
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-67 ページの OPEN（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

FETCH DESCRIPTOR（実行可能な埋込み SQL）

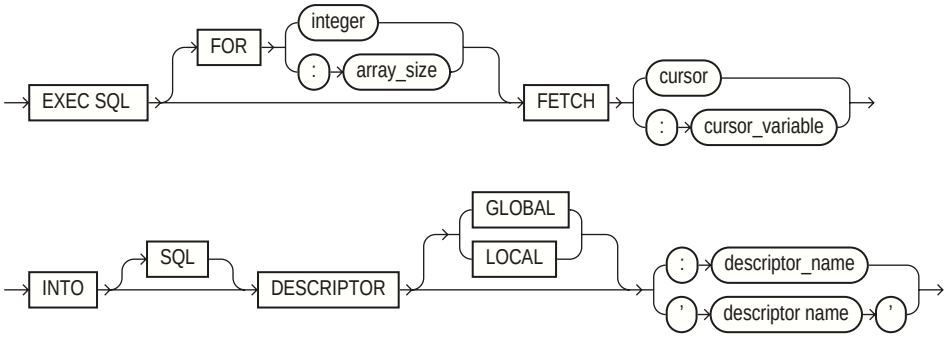
用途

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 で使います。

前提条件

OPEN 文を使っ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

FOR: array_size	処理される行の数を格納するホスト変数です。
FOR integer	処理される行数です。 配列ホスト変数を使う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、Oracle8*i* 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 文を使って宣言したカーソルです。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 文を使って割り当てたカーソル変数です。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なければなりません。
INTO *'descriptor name'*	ANSI 記述子の名前です。
INTO : *descriptor_name*	出力記述子名を格納するホスト変数です。
GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。

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。データを受け取るホスト変数は、すべてスカラーか、すべて配列でなければなりません。スカラーの場合、Oracle8*i* は 1 行だけフェッチします。配列の場合、Oracle8*i* 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、Oracle8*i* がフェッチする行数は、次の値のうち低い方です。

- 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の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をすべて取り出したのではない場合、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り出すと、その次の FETCH ではエラー・コードが発生します。このエラー・コードは SQLCA の SQLCODE 要素に戻されます。

FETCH 文には AT 句がないので注意してください。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だけ進めます。すでにフェッチした行に戻りたい場合は、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 アプリケーションについての詳細は、10-25 ページの「FETCH」を参照してください。

例

```
...
EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor' END-EXEC.
...
EXEC SQL PREPARE S FROM :dyn_statement END-EXEC.
EXEC SQL DECLARE mycursor CURSOR FOR S END-EXEC.
...
EXEC SQL FETCH mycursor INTO DESCRIPTOR 'output_descriptor' END-EXEC.
...
```

関連項目

F-72 ページの PREPARE 文

FREE（実行可能な埋込み SQL 拡張要素）

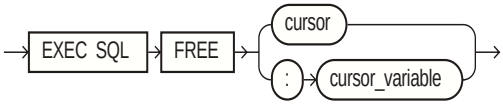
用途

カーソルが使用する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。

構文



キーワードおよびパラメータ

cursor
DECLARE CURSOR 文を使って宣言したカーソルです。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 文を使って割り当てたカーソル変数です。ROWID の場合は、SQL-CURSOR または SQL-ROWID です。LOB の場合は、SQL-BLOB、SQL-CLOB または SQL-NCLOB です。

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。

使用上の注意

5-11 ページの「カーソル」および 6-30 ページの「カーソル変数」を参照してください。

例

```
* CURSOR VARIABLE EXAMPLE
...
 01  CUR      SQL-CURSOR.
...
     EXEC SQL ALLOCATE :CUR END-EXEC.
...
     EXEC SQL CLOSE :CUR END-EXEC.
     EXEC SQL FREE  :CUR END-EXEC.
```

. . .

関連項目

F-10 ページの ALLOCATE（実行可能な埋込み SQL 拡張要素）

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

GET DESCRIPTOR（実行可能な埋込み SQL）

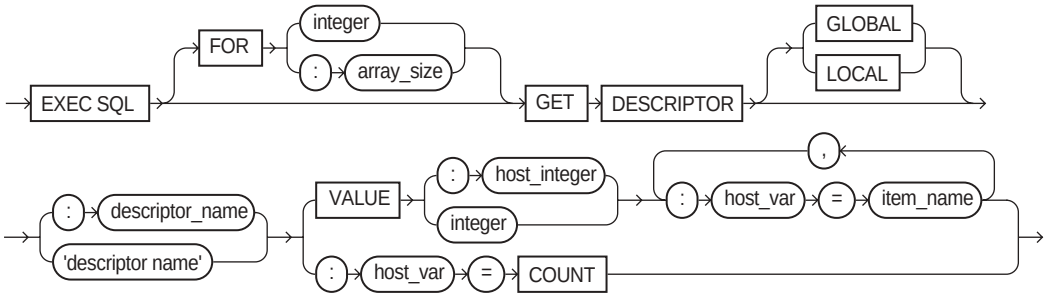
用途

SQL 記述子領域のホスト変数の情報を取得します。

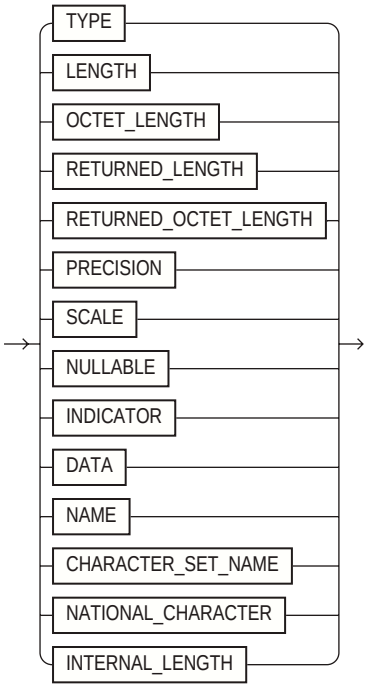
前提条件

値構文および ANSI 動的 SQL 方法 4 以外では使わないでください。

構文



`item_name` だけが以下から選択できます。



キーワードおよびパラメータ

array_size	処理される行の数を格納するホスト変数です。
integer	処理される行数です。
descriptor_name	割当てられた ANSI 記述子名を格納するホスト変数です。
'descriptor name'	割当てられた ANSI 記述子の名前です。
GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。
host_var=COUNT	入力または出力変数の合計数を格納するホスト変数です。
integer	入力または出力変数の合計数です。
VALUE :*host_integer*	参照される入力または出力変数の場所を格納するホスト変数です。
VALUE *integer*	参照される入力または出力変数の場所です。

host_var 項目の値を受け取るホスト変数です。

item_name *item_name* は、10-15 ページの表 10-4 および " 記述子項目名 " 列の見出しの下の 10-16 ページの表 10-5 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_LENGT および INDICATOR の項目名で使えます。詳細は 10-14 ページの「GET DESCRIPTOR」を参照してください。

例

```
EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc_num_vars = COUNT END-EXEC.
```

関連項目

F-12 ページの ALLOCATE DESCRIPTOR（実行可能な埋込み SQL）

F-35 ページの DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）

F-80 ページの SET DESCRIPTOR（実行可能な埋込み SQL）

INSERT（実行可能な埋込み SQL）

用途

表またはビューの実表に行を追加します。

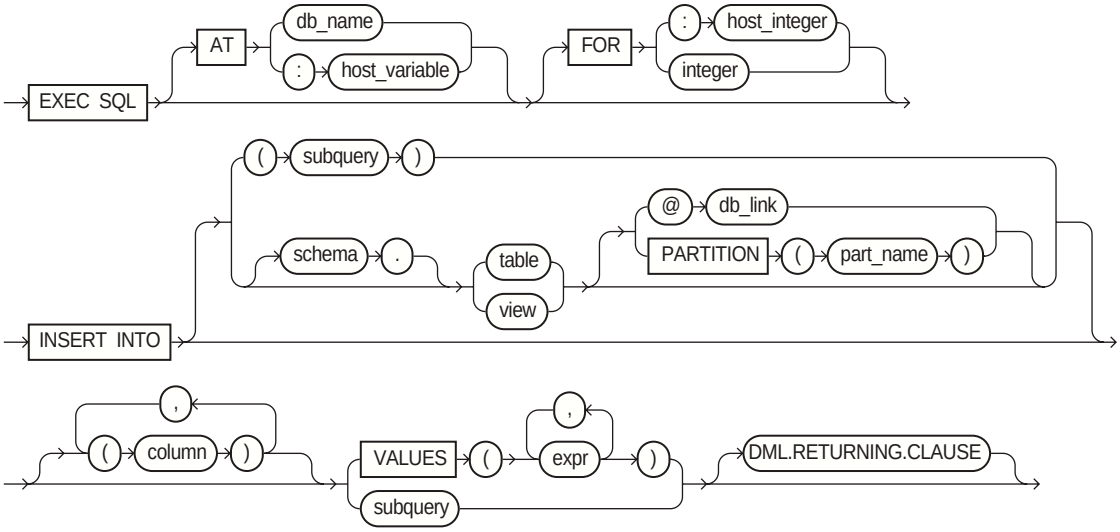
前提条件

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 の権限を持っていなければなりません。

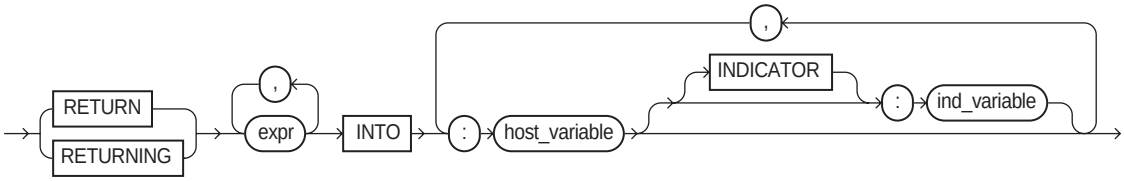
ビューの実表に行を挿入するには、ビューが属するスキーマの所有者が、その実表に対して INSERT の権限を持っていなければなりません。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対して INSERT の権限を持っていなければなりません。

INSERT ANY TABLE システム権限を使うと、表またはビューの実表ならどれにでも行を入力できます。

構文



DML の RETURNING 句の構文を示します。



キーワードおよびパラメータ

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。次のいずれかを使ってデータベースを指定します。

db_name

DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable

事前に宣言された *db_name* 値のホスト変数です。

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ついて実行されます。

FOR *:host_integer*	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。この句を省略すると、Oracle8*i* は最小の配列の各コンポーネントについて 1 回ずつ文を実行します。
schema	表またはビューを含むスキーマです。*schema* を省略した場合、Oracle8*i* は表またはビューが自スキーマ内にあると見なします。
table view	行を挿入する表の名前です。*view* を指定する場合、Oracle8*i* ではビューの実表に行を追加します。
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ーベース・リンクの完全または部分的な名前です。データベース・リンクの参照については、『Oracle8*i* SQL リファレンス』を参照してください。リモートの表またはビューに行を挿入できるのは、Oracle8*i* を分散オプションで使っている場合だけです。 *dblink* を省略した場合、Oracle8*i* は表またはビューがローカル・データベース内にあると見なします。
part_name	表内のパーティションの名前です。
column	表またはビューの列です。挿入した行では、このリストの各列に VALUES 句または問合せから値が割り当てられます。 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しなければなりません。
VALUES	表またはビューに挿入する値の行を指定します。『Oracle8*i* SQL リファレンス』の *expr* の構文説明を参照してください。ホスト変数にオプションの標識変数を合わせた表現も使えます。VALUES 句では、列のリストの各列に式を指定しなければなりません。
subquery	表に挿入される行を戻す副問合せです。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なければなりません。副問合せの構文説明は、『Oracle8*i* SQL リファレンス』の「SELECT」を参照してください。
DML 戻し句	詳細は、5-9 ページの「DML 戻し句」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、すべて配列でなければなりません。変数がスカラーの場合、Oracle8 は INSERT 文を 1 回実行します。変数が配列の場合、Oracle8*i* 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、サイズが異なってもかまいません。この場合、Oracle8*i* が文を実行する回数は、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 *host_integer* の値

この文の詳細は、5-6 ページの「基本的な SQL 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ます。

```
EXEC SQL
    INSERT INTO EMP (ENAME, EMPNO, SAL)
    VALUES (:ENAME, :EMPNO, :SAL)
END-EXEC.
```

例 II

この例では、副問合せを使った埋込み SQL の INSERT 文を示します。

```
EXEC SQL
    INSERT INTO NEWEMP (ENAME, EMPNO, SAL)
    SELECT ENAME, EMPNO, SAL FROM EMP
    WHERE DEPTNO = :DEPTNO
END-EXEC.
```

関連項目

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

LOB APPEND（実行可能な埋込み SQL 拡張要素）

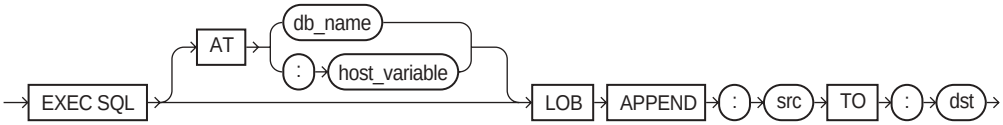
用途

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

使用上の注意、キーワード、パラメータおよび例については、13-10 ページの「APPEND」を参照してください。

関連項目

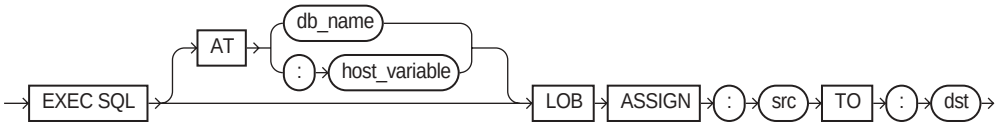
他の LOB 文を参照してください。

LOB ASSIGN（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-11 ページの「ASSIGN」を参照してください。

関連項目

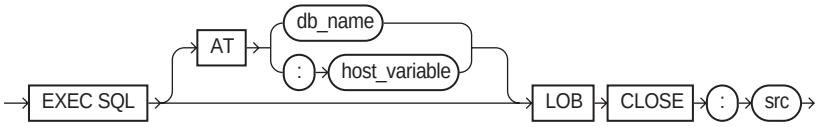
他の LOB 文を参照してください。

LOB CLOSE（実行可能な埋込み SQL 拡張要素）

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については、13-12 ページの「CLOSE」を参照してください。

関連項目

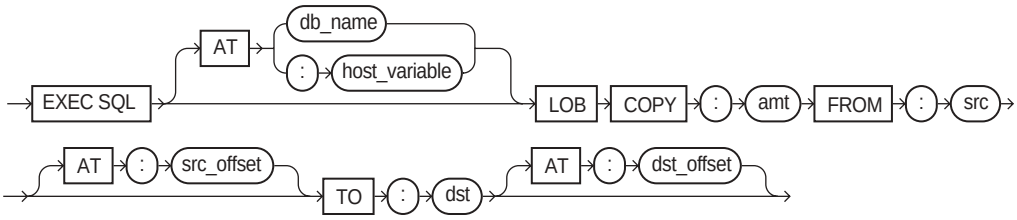
他の LOB 文を参照してください。

LOB COPY（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-12 ページの「COPY」を参照してください。

関連項目

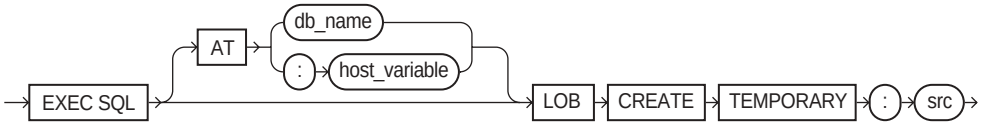
他の LOB 文を参照してください。

LOB CREATE TEMPORARY（実行可能な埋込み SQL 拡張要素）

用途

テンポラリ 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については、13-13 ページの「CREATE TEMPORARY」を参照してください。

関連項目

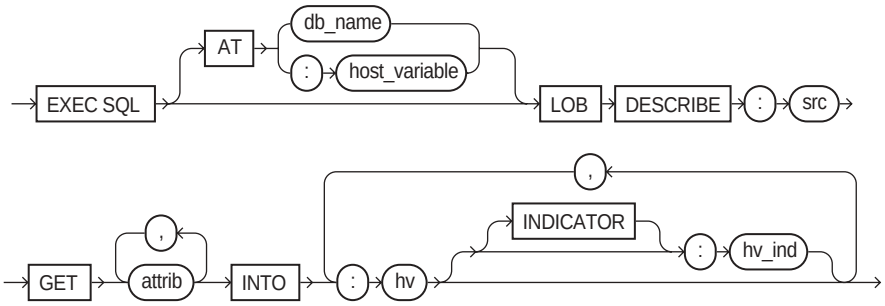
他の LOB 文を参照してください。

LOB DESCRIBE（実行可能な埋込み SQL 拡張要素）

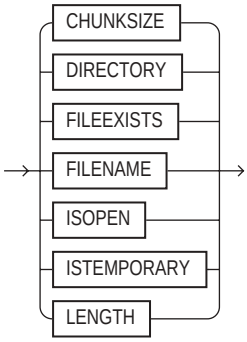
用途

LOB から属性を取り出します。

構文



属性を以下に示します。



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については、13-24 ページの「DESCRIBE」を参照してください。

関連項目

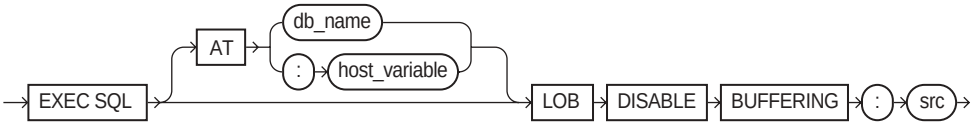
他の LOB 文を参照してください。

LOB DISABLE BUFFERING（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-14 ページの「DISABLE BUFFERING」を参照してください。

関連項目

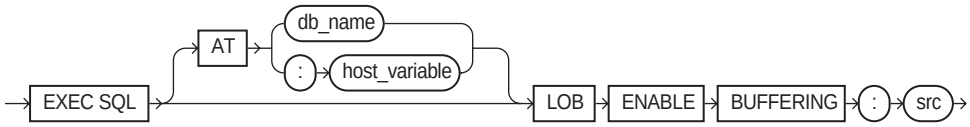
他の LOB 文を参照してください。

LOB ENABLE BUFFERING（実行可能な埋込み SQL 拡張要素）

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については、13-14 ページの ENABLE BUFFERING を参照してください。

関連項目

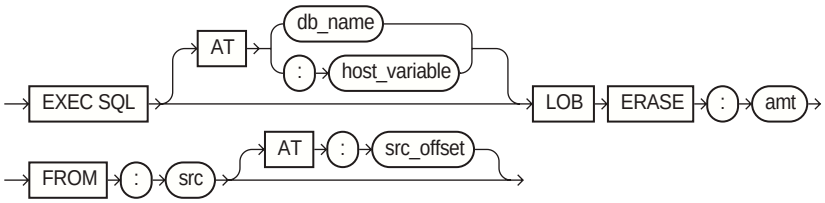
他の LOB 文を参照してください。

LOB ERASE（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-15 ページの「ERASE」を参照してください。

関連項目

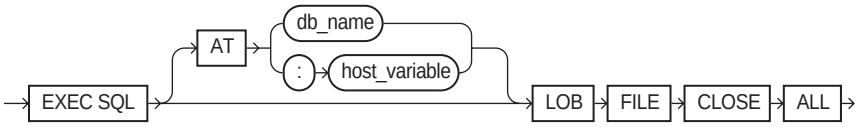
他の LOB 文を参照してください。

LOB FILE CLOSE ALL（実行可能な埋込み SQL 拡張要素）

用途

現行のセッションでオープンしているすべての BFILES 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については、13-16 ページの「FILE CLOSE ALL」を参照してください。

関連項目

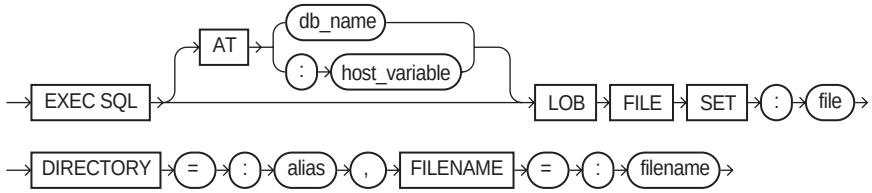
他の LOB 文を参照してください。

LOB FILE SET（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-16 ページの「FILE SET」を参照してください。

関連項目

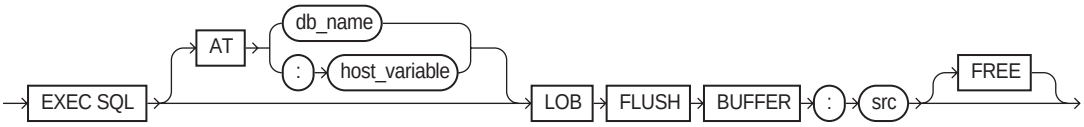
他の LOB 文を参照してください。

LOB FLUSH BUFFER（実行可能な埋込み SQL 拡張要素）

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については、13-17 ページの「FLUSH BUFFER」を参照してください。

関連項目

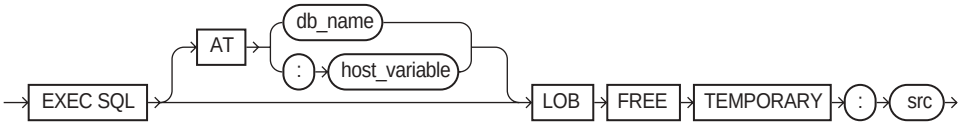
他の LOB 文を参照してください。

LOB FREE TEMPORARY（実行可能な埋込み SQL 拡張要素）

用途

LOB ロケータ用にテンポラリ領域を解放します。

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については、13-17 ページの「FREE TEMPORARY」を参照してください。

関連項目

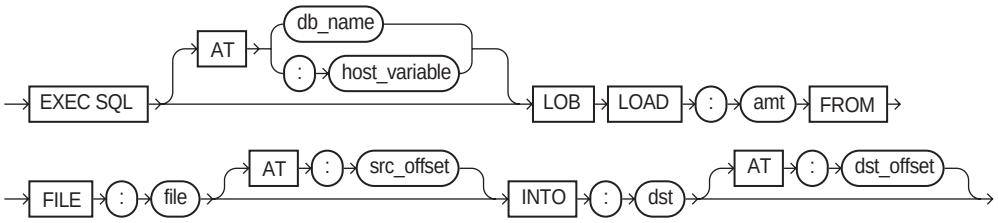
他の LOB 文を参照してください。

LOB LOAD（実行可能な埋込み SQL 拡張要素）

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については、13-18 ページの「LOAD FROM FILE」を参照してください。

関連項目

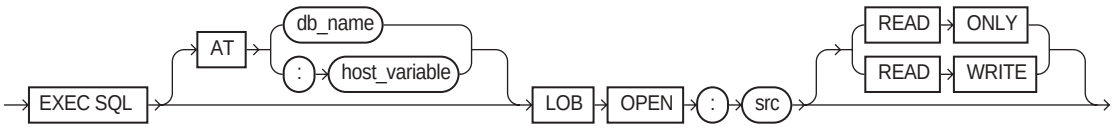
他の LOB 文を参照してください。

LOB OPEN（実行可能な埋込み SQL 拡張要素）

用途

読込みまたは読込み書込みを行う 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については、13-19 ページの「OPEN」を参照してください。

関連項目

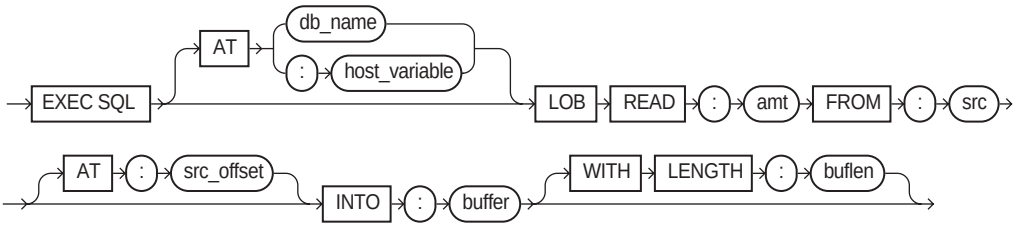
他の LOB 文を参照してください。

LOB READ（実行可能な埋込み SQL 拡張要素）

用途

LOB または BFILE の一部をバッファにコピーします。

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については、13-20 ページの「READ」を参照してください。

関連項目

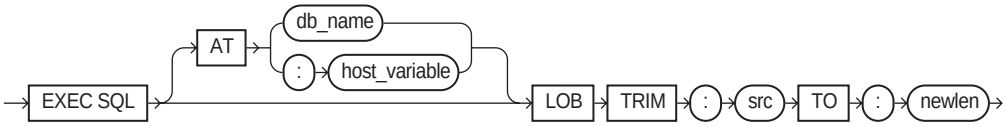
他の LOB 文を参照してください。

LOB TRIM（実行可能な埋込み SQL 拡張要素）

用途

LOB 値を切り捨てます。

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については、13-22 ページの「TRIM」を参照してください。

関連項目

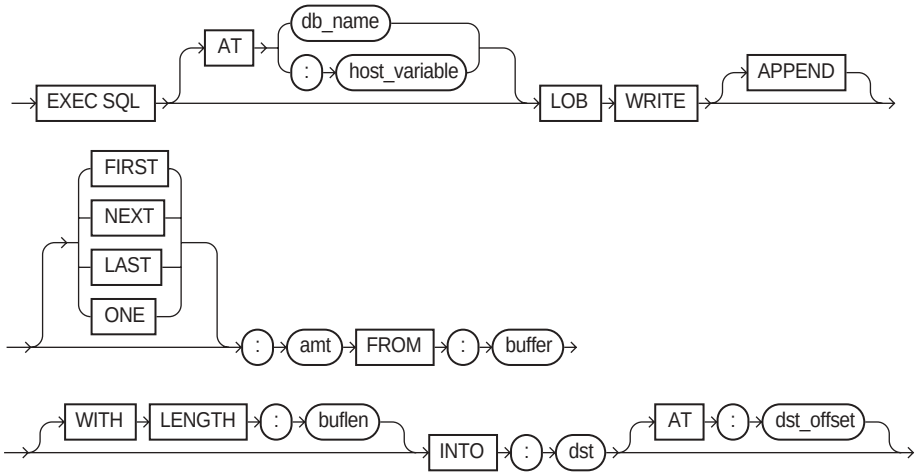
他の LOB 文を参照してください。

LOB WRITE（実行可能な埋込み SQL 拡張要素）

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

使用上の注意、キーワード、パラメータおよび例については、13-22 ページの「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。

OPEN（実行可能な埋込み SQL）

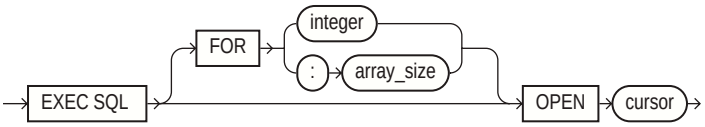
用途

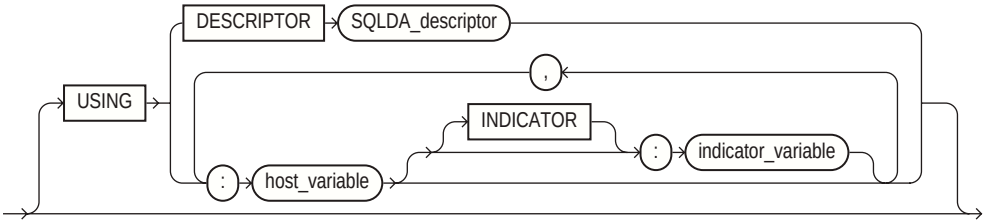
対応付けられた問合せを評価し、USING 句が示すホスト変数名を問合せの WHERE 句に代入して、カーソルをオープンします。ANSI 動的 SQL 構文については、F-69 ページの「OPEN DESCRIPTOR（実行可能な埋込み SQL）」を参照してください。

前提条件

カーソルは、オープンする前に埋込み SQL の DECLARE CURSOR 文を使っ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

array_size	処理される行の数を格納するホスト変数です。
integer	処理される行数です。
cursor	オープンする（事前に宣言された）カーソルです。
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。
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 文を使って事前に初期化されていなければなりません。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 文を使って取り出されます。

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しなければなりません。

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 文を使っ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。

カーソルはクローズせずに再オープンできます。この文の詳細は、5-13 ページの「カーソルのオープン」を参照してください。

例

この例では、Pro*COBOL プログラムで OPEN 文を使用する方法を示します。

```
EXEC SQL DECLARE EMPCURSOR CURSOR FOR
    SELECT ENAME, EMPNO, JOB, SAL
    FROM EMP
    WHERE DEPTNO = :DEPTNO
END-EXEC.
EXEC SQL OPEN EMPCURSOR END-EXEC.
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

OPEN DESCRIPTOR（実行可能な埋込み SQL）

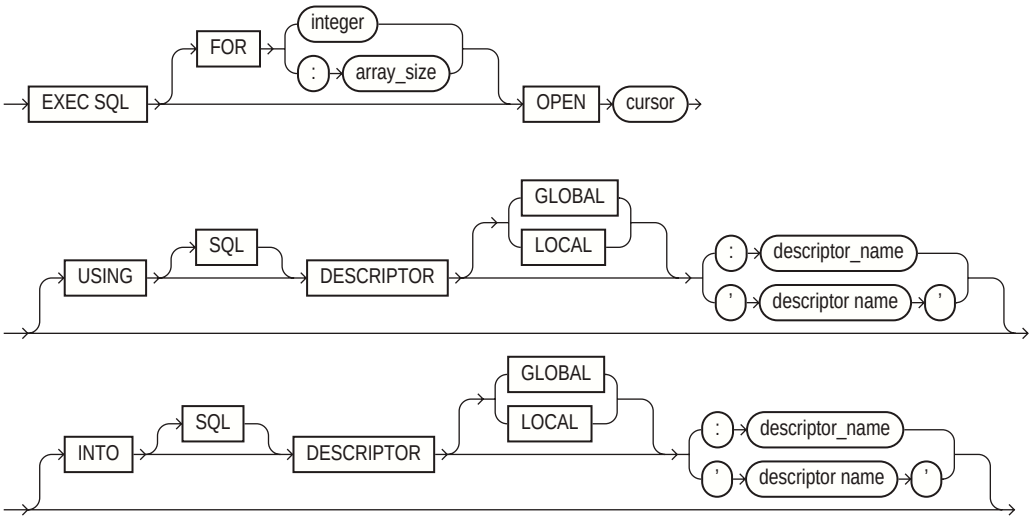
用途

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 文を使っ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

array_size	処理される行の数を格納するホスト変数です。
integer	処理される行数です。
cursor	オープンする（事前に宣言された）カーソルです。
USING DESCRIPTOR *descriptor_name* *'descriptor name'*	ANSI 入力記述子およびその名前を格納するホスト変数を指定するか、ANSI 記述子の名前を指定します。
INTO DESCRIPTOR *descriptor_name* *'descriptor name'*	ANSI 出力記述子およびその名前を格納するホスト変数を指定するか、ANSI 記述子の名前を指定します。
GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。

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 文を使って取り出されます。

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しなければなりません。

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 文を使っ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。

カーソルはクローズせずに再オープンできます。この文の詳細は、5-8 ページの「行の挿入」を参照してください。

例

```
01 DYN-STATEMENT PIC X(58) VALUE "SELECT ENAME, EMPNO FROM EMP WHERE
      DEPTNO =:DEPTNO-DAT".
01 DEPTNO-DAT PIC S9(9) COMP VALUE 10.
...
    EXEC SQL ALLOCATE DESCRIPTOR 'input-descriptor' END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR 'output-descriptor'
    ...
    EXEC SQL PREPARE S FROM :DYN-STATEMENT END-EXEC.
    EXEC SQL DECLARE C CURSOR FOR S END-EXEC.
    ...
    EXEC SQL OPEN C USING DESCRIPTOR 'input-descriptor' END-EXEC.
    ...
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-46 ページの FETCH DESCRIPTOR（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

PREPARE（実行可能な埋込み SQL）

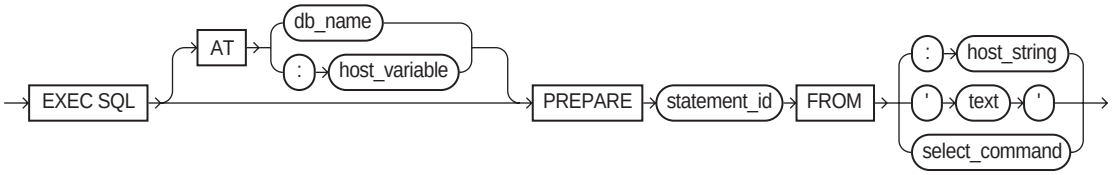
用途

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

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 終了文字列です。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見なされます。
host_variable	データベース接続名を格納するホスト変数です。
array_size	処理される行の数を格納するホスト変数です。
integer	処理される行数です。
statement_id	準備済みの SQL 文または PL/SQL ブロックに対応付ける識別子です。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。
host_string	準備する SQL 文または PL/SQL ブロックのテキストが値であるホスト変数です。
text	実行する SQL 文または PL/SQL ブロックを含むテキスト・リテラルです。引用符は省略できます。
select_command	SELECT 文です。

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 文は一度準備すれば、何回でも実行できます。

例

この例では、Pro*COBOL 埋込み SQL プログラムで PREPARE 文を使用する方法を示します。

```
EXEC SQL PREPARE MYSTATEMENT FROM :MY-STRING END-EXEC.
EXEC SQL EXECUTE MYSTATEMENT END-EXEC.
```

関連項目

F-14 ページの CLOSE（実行可能な埋込み SQL）

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-69 ページの OPEN（実行可能な埋込み SQL）

ROLLBACK（実行可能な埋込み SQL）

用途

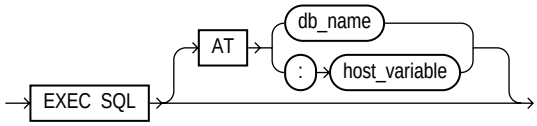
現行トランザクションで実行した作業を取り消します。この文は、インダウトの分散トランザクションの処理を手動で取り消すときにも使用できます。

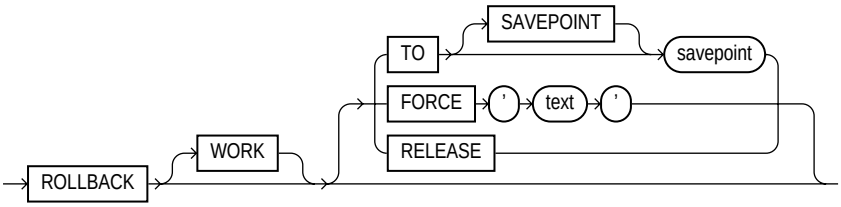
前提条件

現行トランザクションをロールバックするには、権限は必要ありません。

自分で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

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 終了文字列です。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見なされます。

host_variable
データベース接続名を格納するホスト変数です。

WORK
オプションです。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現行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。ローカルまたはグローバル・トランザクション ID を格納する *text* によりトランザクションを指定します。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・サーバーから切断します。RELEASE 句は、SAVEPOINT 句および FORCE 句とは併用できません。

savepoint
ロールバックするためセーブポイントの名前です。

使用上の注意

トランザクション（論理的な作業単位）は、Oracle8*i* 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 文または ROLLBACK 文、データベースへの接続の後の、最初の実行可能な SQL 文から始まります。トランザクションは、COMMIT 文または ROLLBACK 文、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。Oracle8*i* は、データ定義言語文の処理前および処理後に暗黙の COMMIT 文を発行します。

TO SAVEPOINT 句を指定せずに ROLLBACK 文を使うと、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。
- 現行トランザクションの変更内容がすべて取り消されます。
- トランザクションのセーブポイントがすべて消去されます。
- トランザクションのロックを解除します。

TO SAVEPOINT 句を指定して ROLLBACK 文を使うと、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だけ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。指定したセーブポイントは保持されるので、そのセーブポイントに複数回ロールバックできます。それ以前のセーブポイントも保持されます。
- 指定したセーブポイント後に取得した表と行のロックがすべて解除されます。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機しなければなりません。行をまだ要求していない他のトランザクションは、ただちに行を要求し、アクセスできます。

アプリケーション・プログラムでは、COMMIT 文または ROLLBACK 文を使ってトランザクションを明示的に終了することを推奨します。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、Oracle8*i* は最後のコミットされていないトランザクションをロールバックします。

例 I

次の文は現行トランザクション全体をロールバックします。

```
EXEC SQL ROLLBACK END-EXEC.
```

例 II

次の文は現行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 END-EXEC.
```

分散トランザクション

Oracle8*i* で分散オプションを使用すると、分散トランザクション、つまり複数のデータベースのデータを変更するトランザクションを実行できます。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わせて、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ロールバックする場合の詳細は、『Oracle8*i* 分散システム』を参照してください。

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だけをロールバックします。このような文は、現行トランザクションには影響しません。

例 III

次の文は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。

```
EXEC SQL ROLLBACK WORK FORCE '25.32.87' END-EXEC.
```

関連項目

F-15 ページの COMMIT（実行可能な埋込み SQL）

F-76 ページの SAVEPOINT（実行可能な埋込み SQL）

SAVEPOINT（実行可能な埋込み SQL）

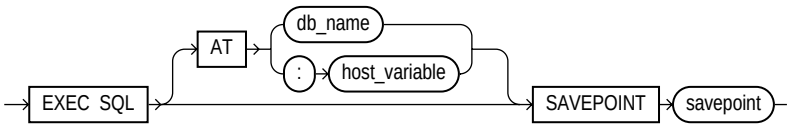
用途

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

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。次のいずれかを使ってデータベースを指定します。

- *db_name*　DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。
- *host_variable*　事前に宣言された *db_name* 値のホスト変数です。

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です。

使用上の注意

この文の詳細は、3-22 ページの「SAVEPOINT 文の使用方法」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 END-EXEC.
```

関連項目

F-15 ページの COMMIT（実行可能な埋込み SQL）

F-73 ページの ROLLBACK（実行可能な埋込み SQL）

SELECT（実行可能な埋込み SQL）

用途

選択した値をホスト変数に割り当てて、1 つまたは複数の表またはビュー、スナップショットからデータを取り出します。

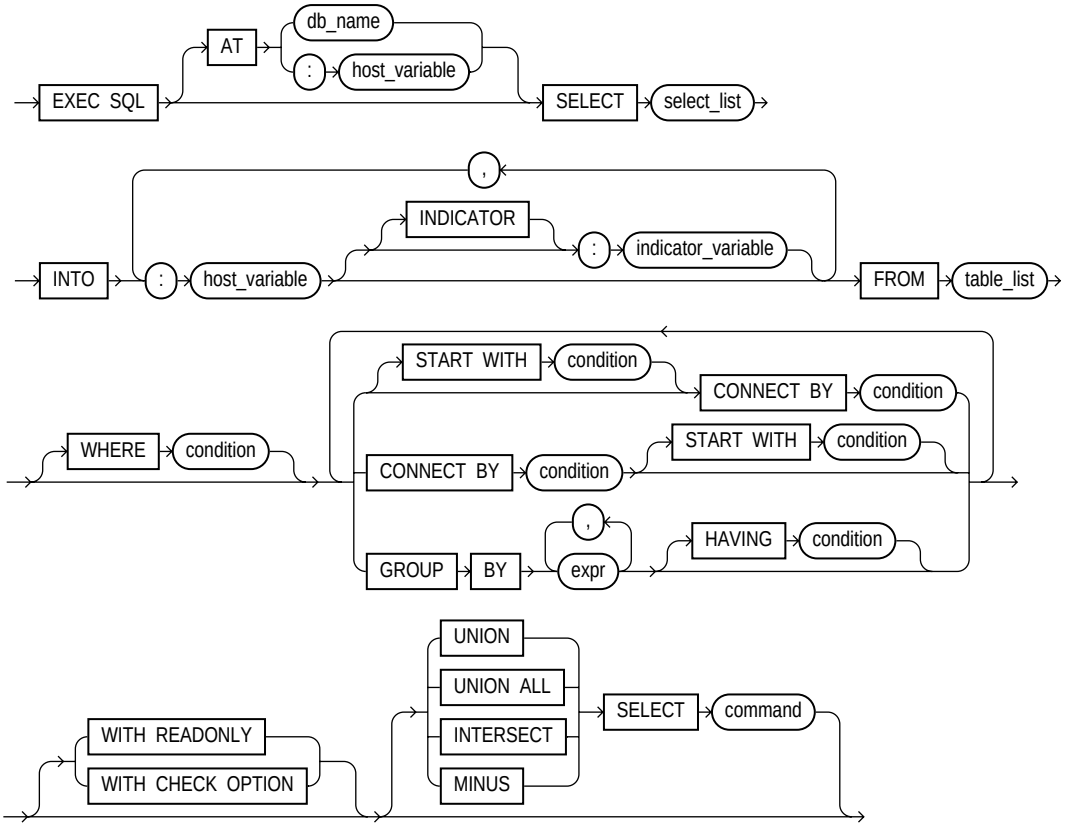
前提条件

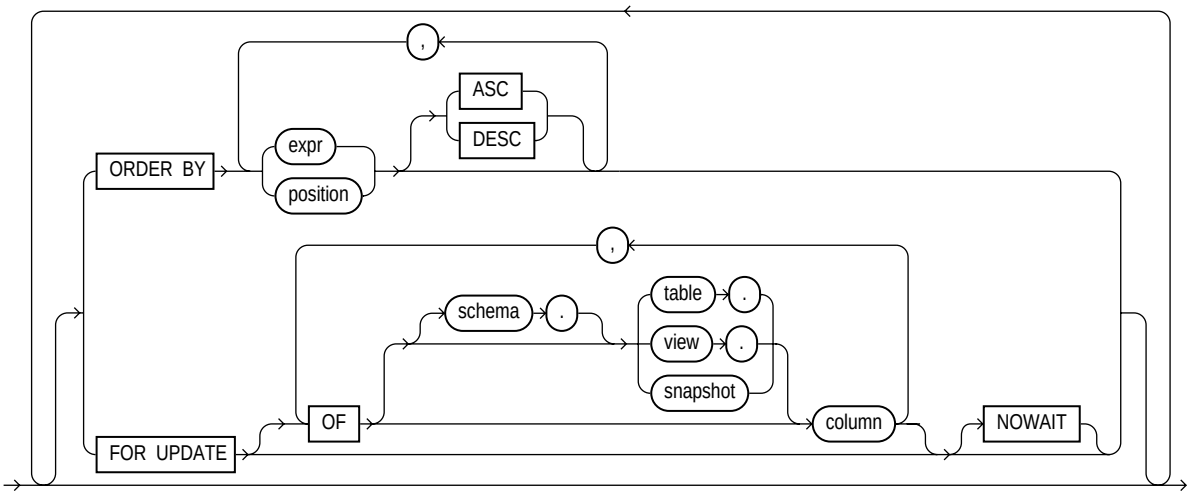
表またはスナップショットからデータを選択するには、表またはスナップショットが自分のスキーマ内にあるか、あるいは表またはスナップショットに対して SELECT の権限を持っている必要があります。

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 の権限を持っていなければなりません。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 の権限を持っている必要があります。

SELECT ANY TABLE システム権限を使うと、どのテーブル、スナップショット、あるいはビューの実表からでも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

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。次のいずれかを使ってデータベースを指定します。

- *db_name*　DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。
- *host_variable*　事前に宣言された *db_name* 値のホスト変数です。

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。ただし、リテラルのかわりにホスト変数を使えます。

INTO　SELECT 文が戻すデータを受け取る出力ホスト変数および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、すべて配列でなければなりません。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 の行だけに制限します。構文の詳細は、『Oracle8*i* SQL リファレンス』の *condition* を参照してください。*condition* では、ホスト変数は使えますが、標識変数は使えません。これらのホスト変数は、スカラーと配列のどちらでもかまいません。

その他のキーワードとパラメータは、非埋込み SQL の SELECT 文と同じです。

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、Oracle8*i* は SQLCA の SQLCODE コンポーネントを使ってエラー・コードを戻します。

SELECT 文ではコメントを使って指示またはヒントを Oracle8*i* のオプティマイザに渡すことができます。オプティマイザはヒントを使って文の実行計画を選択します。ヒントの詳細は、『Oracle8*i* チューニング』を参照してください。

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
END-EXEC.
```

関連項目

F-24 ページの DECLARE CURSOR（埋込み SQL ディレクティブ）

F-26 ページの DECLARE DATABASE（Oracle 埋込み SQL ディレクティブ）

F-38 ページの EXECUTE（実行可能な埋込み SQL）

F-44 ページの FETCH（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

SET DESCRIPTOR（実行可能な埋込み SQL）

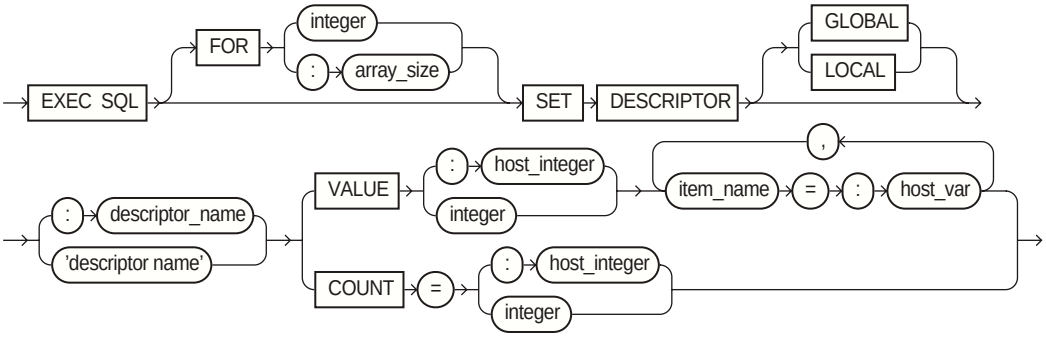
用途

ANSI 動的 SQL 文を使って、ホスト変数の記述子領域内の情報を設定します。

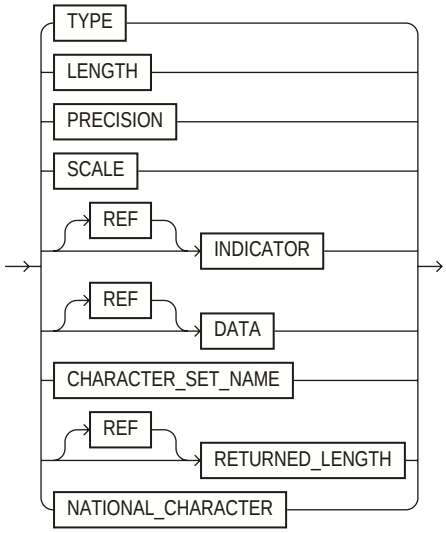
前提条件

DESCRIBE DESCRIPTOR のあとで使用します。

構文



`item_name` だけが以下から選択できます。



キーワードおよびパラメータ

array_size	処理される行の数を格納するホスト変数です。
integer	処理される行数です。配列サイズの句は、DATA、RETURNED_LENGT および INDICATOR の項目名だけで使えます。
GLOBAL | LOCAL	LOCAL（デフォルト）はファイルの有効範囲です。GLOBAL はアプリケーションの有効範囲です。
descriptor_name	割り当てられた ANSI 記述子名を格納するホスト変数です。
'descriptor name'	割り当てられた ANSI 記述子の名前です。
COUNT	入力および出力変数の合計数です。
VALUE	文中で参照されるホスト変数の場所です。
item_name	*item_names* およびその記述子のリストについては、10-18 ページの表 10-6 および 10-18 ページの表 10-7 を参照してください。
host_var	入力および出力変数の合計数を格納するホスト変数です。
integer	入力および出力変数の合計数です。
host_var	項目の設定に使うホスト変数です。
REF	リファレンス・セマンティックスが使われます。RETURNED_LENGTH、DATA および INDICATOR の項目名以外では使えません。 RETURNED_LENGTH を設定するときに使う必要があります。

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、10-17 ページの「SET DESCRIPTOR」を参照してください。

例

```
EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 END-EXEC.
```

関連項目

F-12 ページの ALLOCATE DESCRIPTOR（実行可能な埋込み SQL）

F-22 ページの DEALLOCATE DESCRIPTOR（埋込み SQL 文）

F-35 ページの DESCRIBE DESCRIPTOR（実行可能な埋込み SQL）

F-50 ページの GET DESCRIPTOR（実行可能な埋込み SQL）

F-72 ページの PREPARE（実行可能な埋込み SQL）

UPDATE（実行可能な埋込み SQL）

用途

表またはビュー実表の既存の値を変更します。

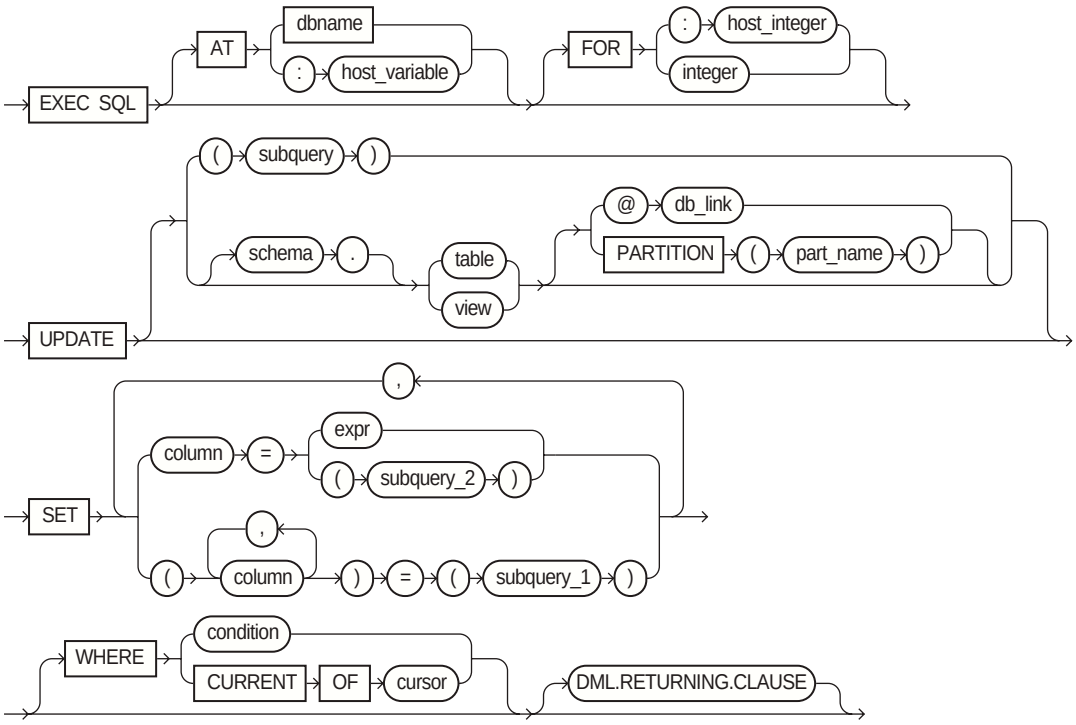
前提条件

表またはスナップショットの値を更新するには、表が自分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。

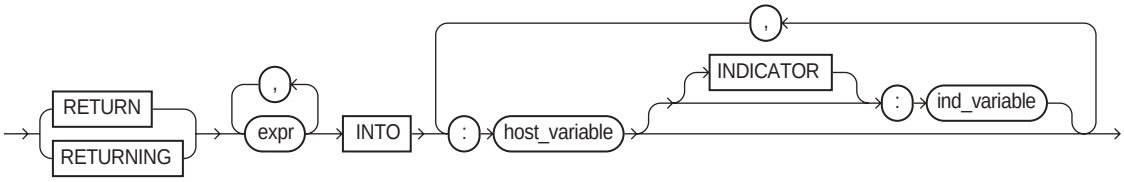
ビューの実表の値を更新するには、ビューが属するスキーマの所有者が、実表に対して UPDATE の権限を持っていなければなりません。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対して UPDATE の権限を持っている必要があります。

UPDATE ANY TABLE システム権限を使うと、どの表またはビュー実表の値でも更新できます。

構文



DML の RETURNING 句の構文を示します。



キーワードおよびパラメータ

AT

どのデータベースに対して UPDATE 文を発行するかを指定します。次のいずれかを使ってデータベースを指定します。

dbname DECLARE DATABASE 文を使って事前に宣言したデータベース識別子です。

host_variable 事前に宣言された *dbname* 値のホスト変数です。

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。

VALUE: *host_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。この句を省略すると、Oracle8*i* は最小の配列の各コンポーネントについて 1 回ずつ文を実行します。

schema

表またはビューを含むスキーマです。*schema* を省略した場合、Oracle8*i* は表またはビューが自スキーマ内にあると見なします。

table view

更新する表の名前です。*view* を指定する場合、Oracle8*i* ではビューの実表を更新します。

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ーベース・リンクの完全または部分的な名前です。データベース・リンク参照の情報は、『Oracle8*i* SQL リファレンス』を参照してください。データベース・リンクを使ってリモートの表またはビューを更新できるのは、Oracle8*i* を分散オプションで使っている場合だけです。

part_name

表内のパーティションの名前です。

alias

文の他の場所にある表、ビューまたは副問合せを参照するのに使う名前です。

column

表またはビューで更新する列の名前です。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です。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。『Oracle8*i* SQL リファレンス』の *expr* の構文を参照してください。

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です。副問合せの構文は、『Oracle8*i* SQL リファレンス』の「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です。副問合せの構文は、『Oracle8*i* SQL リファレンス』の「SELECT」を参照してください。

WHERE　表またはビューで更新する行を指定します。

condition　この条件が真の行だけを更新します。条件には、ホスト変数およびオプショの標識変数を使用できます。『Oracle8*i* SQL リファレンス』の *condition* の構文の説明を参照してください。

CURRENT OF　*cursor* によって最後にフェッチされた行だけを更新します。結合を実行する SELECT 文に *cursor* を対応付けるには、FOR UPDATE 句で明示的に 1 つの表だけ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、Oracle8*i* は表またはビューのすべての行を更新します。

DML 戻し句　詳細は、5-9 ページの「DML 戻し句」を参照してください。

使用上の注意

SET 句と 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なければなりません。変数がスカラーの場合、Oracle8*i* は UPDATE 文を 1 回だけ実行します。変数が配列の場合、Oracle8*i* 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、Oracle8*i* が文を実行する回数は、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

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った場合、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、Oracle8*i* は SQLCA の SQLCODE 要素を通じてエラー・メッセージを戻します。

the SQLCA.WHERE 句を省略した場合は、すべての行が更新され、Oracle8*i* 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 文ではコメントを使って指示またはヒントを Oracle8*i* のオプティマイザに渡すことができます。オプティマイザはヒントを使って文の実行計画を選択します。ヒントの詳細は、『Oracle8*i* チューニング』を参照してください。

このコマンドの詳細は、5-6 ページの「基本的な SQL 文」および第 3 章の「データベースの概念」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
END-EXEC.

EXEC SQL UPDATE EMP
    SET (SAL, COMM) =
        (SELECT AVG(SAL)*1.1, AVG(COMM)*1.1
        FROM EMP)
    WHERE ENAME = 'JONES'
END-EXEC.
```

関連項目

F-26 ページの DECLARE DATABASE（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）

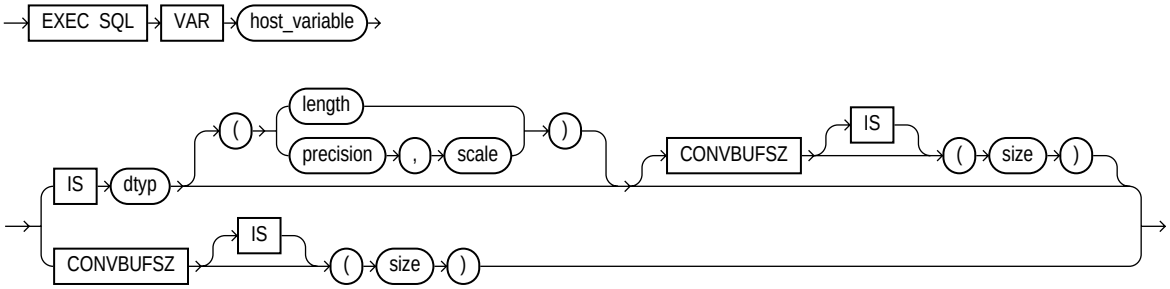
用途

ホスト変数の同値化を実行するか特定の Oracle8*i* 外部データ型をこ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無効にします。また、オプションの CONVBUFSZ 句を使って、キャラクタ・セット変換用のバッファ・サイズを指定します。

前提条件

ホスト変数は、埋込み SQL プログラム内であらかじめ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

host_variable Oracle8*i* の外部データ型を割り当てるホスト変数です。

dtyp Pro*COBOL によって認識される Oracle8*i* の外部データ型です。（Oracle8*i* の内部データ型ではありません。）データ型には、長さまたは精度、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、4-7 ページの「外部データ型」を参照してください。

size Oracle8*i* ランタイム・ライブラリ内のバッファのサイズ（バイト単位）です。これを使って、*host_variable* のキャラクタ・セットを変換します。

使用上の注意

ホスト変数の同値化は、データ型の同値化の 1 つです。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 NULL 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います。

Oracle データ型の変換の詳細は、4-51 ページの「サンプル・プログラム 4: データ型の同値化」を参照してください。

例

この例では、ホスト変数 DEPT_NAME をデータ型 STRING に、ホスト変数 BUFFER をデータ型 RAW(200) に同値化しています。

```
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     ...
     01 DEPT-NAME  PIC X(15).
* -- default datatype is CHAR
     EXEC SQL VAR DEPT-NAME IS STRING END-EXEC.
* -- reset to STRING
     ...
     01 BUFFER-VAR.
          05 BUFFER  PIC X(200).
* -- default datatype is CHAR
     EXEC SQL VAR BUFFER IS RAW(200) END-EXEC.
* -- refer to RAW
     ...
     EXEC SQL END DECLARE SECTION END-EXEC.
```

関連項目

なし

WHENEVER（埋込み SQL のディレクティブ）

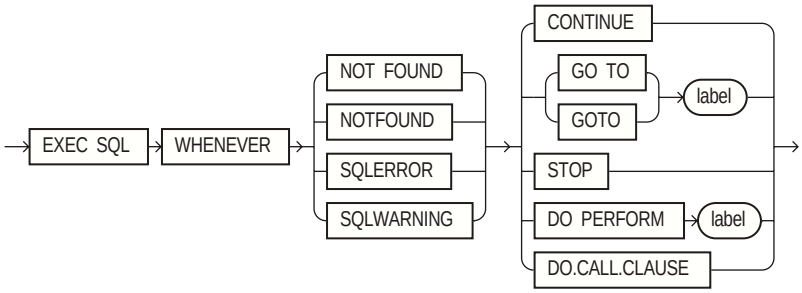
用途

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。

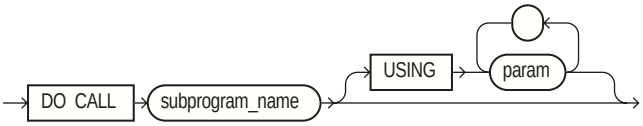
前提条件

なし

構文



DO CALL CAUSE の構文を以下に示します。



キーワードおよびパラメータ

NOT FOUND | NOTFOUND	エラー・コード +1403 を SQLCODE（または、MODE=ANSI のときは +100 コード）に戻す例外条件を指定します。
SQLERROR	負のリターン・コードを戻す条件を指定します。
SQLWARNING	致命的でない警告条件を指定します。
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 | GO TO	プログラムにラベル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 PERFORM	プログラムがラベルでパラグラフを実行する必要があることを示します。
DO CALL	プログラムがサブプログラムを実行する必要があることを示します。
subprogram_name	実行するサブプログラムです。引用符（"）で囲む必要があります。
USING	サブプログラムのパラメータです。
param	空白で区切られたサブプログラム・パラメータのリストです。

WHENEVER ディレクティブを使うと、埋込み SQL 文でエラーまたは警告が発生したときに、プログラムから指定したアクションの 1 つを実行できます。

WHENEVER 文の有効範囲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 文は、同じ条件をチェックする別の WHENEVER 文に置換されるまで有効です。

このディレクティブの条件およびアクションの例は、8-28 ページの「WHENEVER ディレクティブ」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

次の例では、Pro*COBOL 埋込み SQL プログラムで WHENEVER ディレクティブを使う方法を示します。

```
     EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
     ...
     EXEC SQL WHENEVER SQLERROR GOTO SQL-ERROR END-EXEC.
     ...
 SQL-ERROR.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     DISPLAY "ORACLE ERROR DETECTED.".
     EXEC SQL ROLLBACK RELEASE END-EXEC.
     STOP RUN.
```

関連項目

なし

索引

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

つ

て

と

な

に

ね

の

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

り

れ

ろ

わ