Oracle(R) Database
Oracle プリコンパイラのためのプログラマーズ・ガイド
19c

F16164-02(原本部品番号:E96474-02)

2020年10月

タイトルおよび著作権情報

Oracle Database Oracleプリコンパイラのためのプログラマーズ・ガイド 19c

F16164-02

原本著者: Celin Cherian

原本協力者: Denis Raphaely、Simon Watt、Radhakrishnan Hari、Nancy Ikeda、Ken Jacobs、Valarie Moore、Tim Smith、Scott Urman、Arun Desai、Mallikharjun Vemana、Subhranshu Banerjee

目次

表一覧

はじめに

この章の項目は次のとおりです。

- 対象読者
- ドキュメントのアクセシビリティについて
- このマニュアルの構成
- 関連ドキュメント
- 表記規則

このマニュアルは、Oracle Pro*COBOLおよびPro*FORTRANプリコンパイラの総合的なユーザーズ・ガイドであり、参考資料です。SQLを使用してデータにアクセスし、操作するアプリケーションの開発方法を順を追って説明します。明快な例を使用して、先進のプログラミング・テクニックの基礎にある概念を解明します。

対象読者

新しいアプリケーションの開発、または既存のアプリケーションをOracleデータベースで実行できるように変換するユーザーにとっては、このマニュアルを読むことが大いに役立ちます。このOracleプリコンパイラの総合的なマニュアルは、特にプログラマ向けに書かれたものですが、システム・アナリスト、プロジェクト・マネージャ、埋込みSQLアプリケーションに興味のあるその他のユーザーにとっても有益です。

このマニュアルを効果的に使用するには、次の問題についての実用的な知識が必要です。

- 高水準言語によるアプリケーション・プログラミング
- SQLデータベース言語
- Oracleの概念および専門用語

このマニュアルの構成

このマニュアルは、11の章と5つの付録で構成されています。第1章と第2章では、ユーザーにOracleプリコンパイラおよびSQLプログラムについての基礎知識を提供し、その後の第3、4、5および6章では、埋込みSQLプログラミングの要点を説明します。これらの章を読めば、実用的な埋込みSQLアプリケーションを作成し、実行できるようになります。第7、8、9、10および11章では、高度な問題を取り上げます。それぞれの章や付録で得られる知識について内容を要約すると、次のとおりです。

このサンプル・マニュアルには、1つのパート、2つの章、1つの付録が含まれます。(HTMLにリンクが表示されるように、この章、付録およびパートは、相互参照として挿入してください。)

概要

この章では、Oracleプリコンパイラについて説明します。Oracleデータを操作するアプリケーション・プログラムを開発する上でのプリコンパイラの役割と、プリコンパイラによりアプリケーションで実行できる処理について学習します。

基礎の学習

この章では、埋込みSQLプログラムがどのように機能するかを説明します。プログラムが動作する特殊な環境、アプリケーション設計に対するこの環境の影響、埋込みSQLプログラミングの基本概念、およびアプリケーション開発のステップについて調べます。

プログラム要件への対応

この章では、埋込みSQLプログラムの要件にどのように対応するかを説明します。変数の宣言、通信領域の宣言、および

Oracleデータベースへの接続を行う埋込みSQLコマンドについて学習します。Oracleデータ型、グローバリゼーション・サポート、データ変換、データ型の同値化の利用方法についても学習します。さらに、この章では、プログラムにOracle Call Interface(OCI)のコールを埋め込む方法や、X/Openアプリケーションの開発方法についても説明しています。

埋込みSQLの使用方法

この章では、埋込みSQLプログラミングの基本について説明します。ホスト変数、インジケータ変数、カーソル、カーソル変数、およびOracleデータの挿入、更新、選択、削除を行う基本的なSQLコマンドの使用方法について学習します。

埋込みPL/SQLの使用方法

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、インジケータ変数、カーソル、ストアド・サブプログラム、ホスト配列、動的SQLとともにPL/SQLを使用する方法について学習します。

Oracleプリコンパイラの実行

この章では、Oracleプリコンパイラを実行するための要件について詳しく説明します。プリコンパイル中に何が起こるか、プリコンパイラ・コマンドを発行する方法、多くの便利なプリコンパイラ・オプションを指定する方法、条件付きプリコンパイルと分割プリコンパイルの実行方法、ホスト・プログラムにOCIコールを埋め込む方法について学習します。

トランザクションの定義および制御

この章では、トランザクション処理について説明します。データベースの一貫性を保護する基本的なテクニックについて学習します。

エラー処理および診断

この章では、エラー・レポートおよびリカバリについて詳しく説明します。状態変数SQLSTATE、SQLCA構造体およびWHENEVER文を使用してエラーを検出し処理する方法について学習します。ORACAを使用してプログラムを診断する方法についても学習します。

ホスト配列の使用

この章では、プログラムのパフォーマンス改善のために配列を使用する方法について説明します。配列を使用したOracleデータの処理方法、1つのSQL文で配列のすべての要素を操作する方法、処理対象の配列要素の数を制限する方法について学習します。

動的SQLの使用方法

この章では、動的SQLの利用方法について説明します。柔軟性のあるプログラム、中でもユーザーが実行時に対話形式でSQL文を作成できるプログラムを作成する4つの方法を、簡単なものから順番に学習します。

ユーザー・イグジットの作成

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重点的に説明します。まず、Formsアプリケーションがユーザー・イグジットのインタフェースとなるようにするコマンドを学習します。その後、Formsユーザー・イグジットの作成およびリンクの方法について学習します。

新機能

この付録では、Oracleプリコンパイラのリリース1.8で導入された改善点および新機能について説明します。

Oracleの予約語、キーワードおよびネームスペース

この付録では、Oracleライブラリ用に予約されているOracleにとって特別な意味を持つ用語とネームスペースを一覧します。

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するための簡単で適用しやすい方法について説明します。

構文および意味検査

この付録では、埋込みSQL文およびPL/SQLブロックに対して行われる構文およびセマンティックのチェックの種類と範囲を制御するSQLCHECKオプションの使用方法について説明します。

埋込みSQLコマンドおよびディレクティブ

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拡張機能について説明します。ソース・コマンド内でこれらのコマンドは、先頭にキーワードEXEC SQLが付きます。

ドキュメントのアクセシビリティについて

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

Oracle Supportへのアクセス

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

関連ドキュメント

『Oracleプリコンパイラのためのプログラマーズ・ガイド』の詳細は、次の場所のOracle Technology Network(OTN)を参照してください。

http://otn.oracle.co.jp/document/

表記規則

このマニュアルでは次の表記規則を使用します。

規則	意味
. . .	例に含まれる垂直の省略記号は、直接関係のない情報が省略されていることを意味します。
. . .	文またはコマンド内の水平の省略記号は、例に直接関係のない文またはコマンドの一部が省略されていることを意味します。
太字テキスト	テキスト内の太字は、テキスト、用語集またはその両方で定義されている用語を表します。
< >	ユーザーが指定する名前は山カッコで囲みます。
[ ]	大カッコは、オプションの句を囲みます。そこから 1 つ選択しても、まったく選択しなくてもかまいません。

1 概要

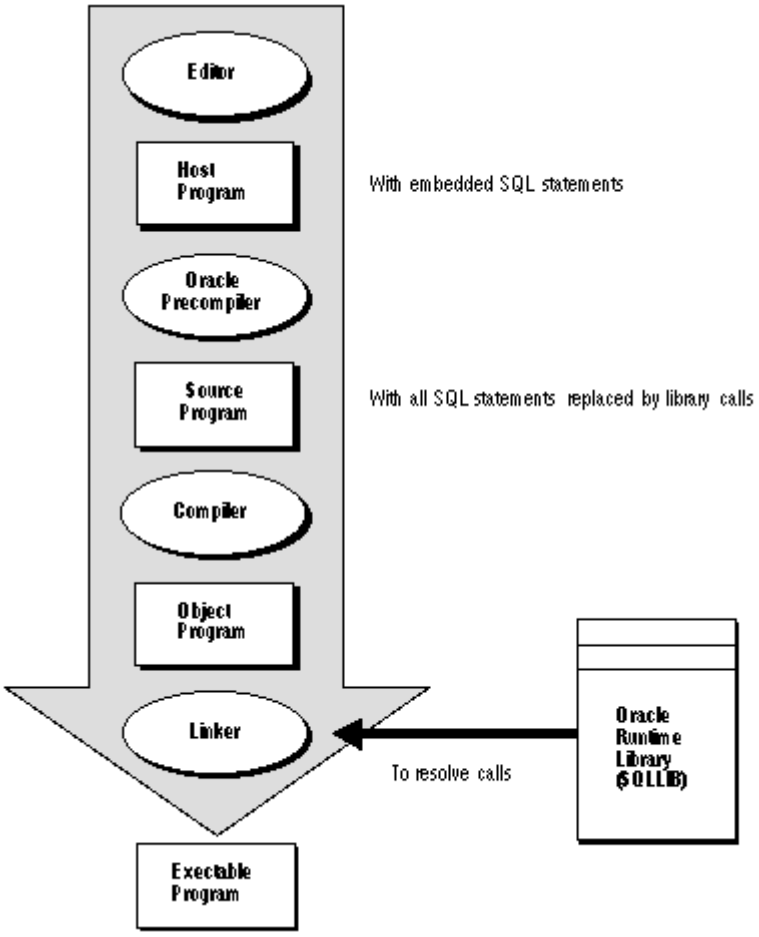
この章では、Oracleプリコンパイラについて説明します。Oracleデータを操作するアプリケーション・プログラムを開発する上でのプリコンパイラの役割と、プリコンパイラによりアプリケーションで実行できる処理について学習します。次の項目について説明します。

- Oracleプリコンパイラ
- Oracleプリコンパイラを使用する理由
- SQLを使用する理由
- PL/SQLを使用する理由
- Oracleプリコンパイラの機能と利点
- 業界標準を満たすOracleプリコンパイラ

1.1 Oracleプリコンパイラ

Oracleプリコンパイラとは、高水準ホスト・プログラムで埋込みSQL文を使用可能にするプログラミング・ツールです。図1-1のように、プリコンパイラはホスト・プログラムを入力として受け入れ、埋込みSQL文を標準Oracleランタイム・ライブラリ・コールに変換して、コンパイル、リンクおよび実行が可能なソース・プログラムを生成します。

図1-1 埋込みSQLプログラムの開発



1.1.1 代替言語

Oracleプリコンパイラには2種類あります(すべてのシステムで使用可能というわけではありません)。次の高水準言語がサポート

されています。

- C/C++
- COBOL

異なるアプリケーション分野用で、異なる設計理念を反映するこれらの言語は、様々なプログラミング上のソリューションを提供します。

ノート:

このマニュアルは、C/C++および COBOL でのプリコンパイラ使用の手引書と合せて参照してください。

Pro*FORTRANおよびSQL*Module for Adaはメンテナンス・モードであり、これらの製品のバージョン1は、リリース1.6以上の追加機能で強化されることはありません。ただし、オラクル社では、不具合が報告され、修正されるたびに、引き続きパッチ・リリースを発行します。

1.2 Oracleプリコンパイラを使用する理由

Oracleプリコンパイラを使用すると、アプリケーション・プログラムに柔軟なSQLを組み込むことができます。CやCOBOLなどの普及している高水準言語でSQLを使用できます。便利で使用しやすいインタフェースにより、アプリケーションからOracleに直接アクセスできます。

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

さらに、Oracleプリコンパイラは、アプリケーションの微調整に役立ちます。リソースの使用状況、SQL文の実行状況および各種ランタイム・インジケータを綿密に監視できます。得られた情報に基づいて、最大のパフォーマンスが実現できるようにプログラムのパラメータを調整できます。

1.3 SQLを使用する理由

Oracleデータにアクセスして操作するには、SQLが必要です。SQLを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異なります。ジョブにCまたはCOBOLの手続き型処理機能が必要な場合や、ジョブを定期的に実行する場合は、埋込みSQLを使用してください。

SQLは、その柔軟で強力な特性、および習得が容易であることから、最もすぐれたデータベース言語となりました。非手続き型言語であるため、処理方法を指定せずに要求内容を指定できます。英文に似た少数の文で、Oracleデータを一度に1行または複数行ずつ簡単に操作できます。

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、次のようなSQL文です。

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

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

1.4 PL/SQLを使用する理由

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

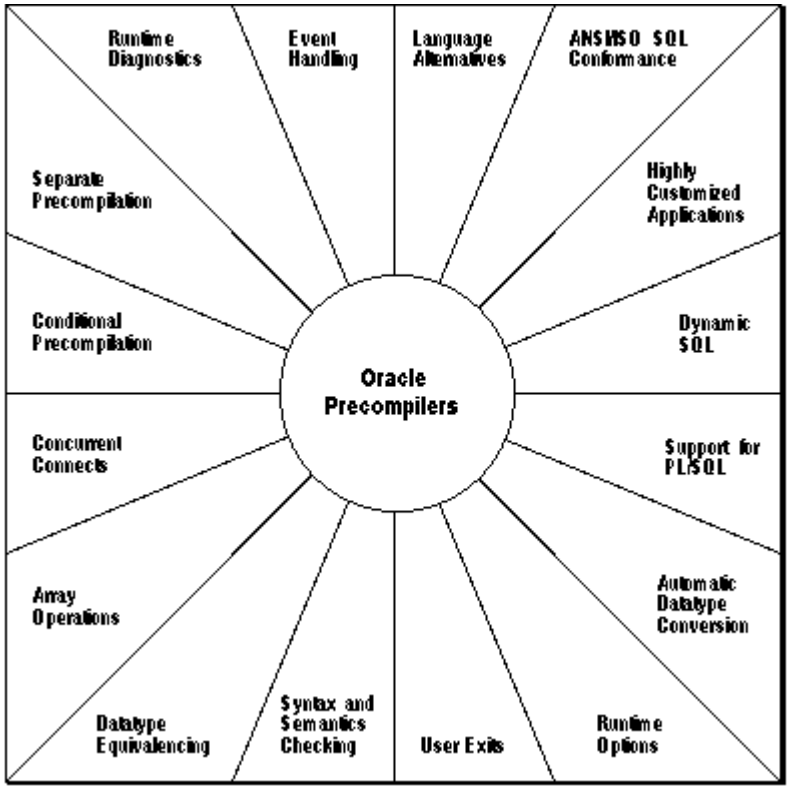
埋込みPL/SQLの主な利点は、パフォーマンスの向上です。SQLとは異なり、PL/SQLでは、SQL文を論理的にグループ化し、1文ずつではなくブロック単位でOracleに送ることができます。これにより、ネットワークの通信量と処理のオーバーヘッドが減少します。

アプリケーション・プログラムの埋込み方法を含め、PL/SQLの詳細は、Oracleプリコンパイラの実行を参照してください。

1.5 Oracleプリコンパイラの機能と利点

図1-2のように、Oracleプリコンパイラには多くの機能と利点があり、効果的で信頼性の高いアプリケーションの開発に役立ちます。

図1-2 機能と利点



たとえば、Oracleプリコンパイラでは次のことが可能です。

- 6種の高水準言語のいずれでもアプリケーションを作成
- ANSI/ISO規格に従ったSQLの埋込み
- 動的SQL(プログラム実行時に有効なSQL文の受入れや作成が可能な高度なプログラミング技術)の利用
- 高度にカスタマイズしたアプリケーションの設計および開発
- Oracleの内部のデータ型と高水準言語のデータ型との間の自動変換
- アプリケーション・プログラムへのPL/SQLトランザクション処理ブロック埋込みによるパフォーマンスの向上
- 便利なプリコンパイラ・オプションの指定およびプリコンパイル中の値の変更
- データ型の同値化を使用した、Oracleによる入力データの解析および出力データの書式設定方法の制御
- 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、それらをリンクして1つの実行可能プログラムにすること

- 埋め込まれたSQLデータ操作文とPL/SQLブロックの構文およびセマンティックのチェック
- 複数のノード上のOracleデータベースへのSQL*Netを使用した同時アクセス
- 入力プログラム変数および出力プログラム変数としての配列の使用
- ホスト・プログラムを異なる環境で実行できるように、コード・セクションを条件付きでプリコンパイル
- 高水準言語で作成されたユーザー・イグジットを使用した、Oracle FormsやOracle Reportsなどのツールとのインタフェース接続
- ANSI承認の状態変数SQLSTATEおよびSQLCODE、SQL通信領域(SQLCA)およびWHENEVER文を使用したエラーと警告の処理
- Oracle通信領域(ORACA)で提供される強力な診断機能の使用

要するに、Oracleプリコンパイラは、専門的な埋込みSQLプログラミング技法をサポートする多機能ツールです。

1.6 業界標準を満たすOracleプリコンパイラ

SQLは、リレーショナル・データベース管理システムにとっての標準言語となりました。この項では、次の機関によって規定された最新のSQL標準に対するOracleプリコンパイラの準拠の状況について説明します。

- ANSI(米国規格協会)
- 国際標準化機構(ISO)
- 米国標準技術局(NIST)

これらの機関では、SQLを次の出版物で定義されたものとして採用しています。

- ANSI Document ANSI X3.135-1992、『*Database Language SQL*』
- International Standard ISO/IEC 9075:1992、『*Database Language SQL*』
- ANSI Document ANSI X3.168-1992、『*Database Language Embedded SQL*』
- NIST Federal Information Processing Standard FIPS PUB 127-2、『*Database Language SQL*』

1.6.1 要件

ANSI X3.135-1992(非公式にはSQL92として知られる)では、適合するSQL言語が指定され、段階的に実装できるように、次の3つの言語レベルが定義されています。

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

適合するSQL実装は、最低でもEntry SQLをサポートする必要があります。

ANSI X3.168-1992では、COBOL、FORTRAN、PascalまたはPL/Iなどの標準的なプログラミング言語で書か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セマンティックが指定されています。

ISO/IEC 9075-1992では、ANSI規格が完全に採用されています。

FIPS PUB 127-2(連邦用に取得されたRDBMSソフトウェアに適用)では、ANSI規格も採用されています。さらに、データベース構造体用の最小サイズ設定パラメータを指定し、ANSI拡張機能を識別するためにFIPSフラガーを必要とします。

ANSI規格のコピーが必要な場合は、次の宛先にお問合せください。

American National Standards Institute 1430 Broadway New York, NY 10018, USA

ISO規格のコピーが必要な場合は、任意のISO加盟機関の国家規格局にお問合せください。NIST規格のコピーについては、次の宛先にお問合せください。

National Technical Information ServiceU.S. Department of Commerce Springfield, VA 22161, USA

1.6.2 準拠

Oracleプリコンパイラは、ANSI、 ISOおよびNIST規格に100%準拠しています。必要に応じて、Entry SQLをサポートし、FIPSフラガーを提供します。

1.6.3 FIPSフラガー

FIPS PUB 127-1によれば、「この標準によって指定されていない追加機能を提供する実装では、非準拠SQL言語または非準拠の方法で処理される可能性のある準拠SQL言語にフラグを付けるオプションも提供される」となっています。この要件を満たすために、Oracleプリコンパイラには*FIPS*フラガーが用意されており、ANSI拡張機能にフラグを付けます。拡張機能は、権限執行規則を除く、ANSI書式または構文規則に違反するSQL要素です。標準SQLに対するOracle拡張機能の一覧は、*『Oracle Database SQL言語リファレンス』*を参照してください。

FIPSフラガーは次のものを識別するために使用できます。

- アプリケーションを非準拠環境に移す場合に、変更が必要になる可能性のある非準拠SQL要素
- 別の処理環境で異なった動作をする可能性のある準拠SQL要素

したがって、FIPSフラガーは、移植性のあるアプリケーションの開発に役立ちます。

1.6.4 FIPSオプション

FIPSというオプションは、FIPSフラガーを制御します。FIPSフラガーを有効にするには、FIPS=YESをインラインまたはコマンドラインで指定します。コマンドライン・オプションFIPSの詳細は、FIPSを参照してください。

1.6.5 認定

NISTでは、約300のテスト・プログラムから成る*SQL*テスト・スイートを使用して、OracleプリコンパイラがANSI Entry SQLに準拠しているかどうかのテストを実施しました。具体的には、これらのプログラムにより、COBOLおよびFORTRAN埋込みSQL標準に対する適合性がテストされました。結果として、Oracleプリコンパイラは100% ANSIに準拠していることが保証されました。

テストの詳細は、次の宛先にお問合せください。:

National Computer Systems Laboratory

Attn: Software Standards Testing Program

National Institute of Standards and Technology

Gaithersburg, MD 20899, USA

2 基礎の学習

この章の項目は次のとおりです。

- 埋込みSQLプログラミングの基本概念
- 埋込みSQLアプリケーションの開発ステップ
- サンプル・プログラム
- サンプル表

この章では、埋込みSQLプログラムの機能について説明します。それらが動作する特殊な環境と、その環境がアプリケーション設計に及ぼす影響について調べます。

埋込みSQLプログラミングの基本概念とアプリケーション開発のステップについて説明した後、簡単なプログラムを使用して、要点を具体的に説明します。

2.1 埋込みSQLプログラミングの基本概念

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

- 埋込みSQL文
- 実行文および宣言部
- 埋込みSQL構文
- 静的SQL文と動的SQL文
- 埋込みPL/SQLブロック
- ホスト変数およびインジケータ変数
- Oracleデータ型
- 配列
- データ型の同値化
- プライベートSQL領域、カーソルおよびアクティブ・セット
- トランザクション
- エラーと警告

2.1.1 埋込みSQL文

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

たとえば、Oracleデータの操作や問合せを行うには、INSERT、UPDATE、DELETEおよびSELECT文を使用します。INSERTではデータの行をデータベース表に追加し、UPDATEでは行を変更し、DELETEでは不要な行を削除し、SELECTでは検索条件と一致する行を取得します。

Oracleプリコンパイラでは、すべてのOracle文がサポートされています。たとえば、強力なSET ROLE文を使用すると、データベース権限を動的に管理できます。ロールとは、関連するシステム権限やオブジェクト権限、あるいはユーザーまたは他のロールに付

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

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

2.1.2 実行文および宣言部

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。

実行文では、SQLLIBランタイム・ライブラリへのコールが発生します。実行文は、Oracleへの接続、Oracleデータの定義、問合せ、操作、Oracleデータへのアクセス制御およびトランザクション処理に使用します。他のホスト言語の実行文を配置できる位置であれば、どこにでも記述できます。

一方、宣言文ではSQLLIBへのコールは発生せず、Oracleデータの操作も行われません。宣言文は、Oracleオブジェクト、通信領域およびSQL変数を宣言するために使用します。ホスト言語の宣言を配置できる位置であれば、どこにでも記述できます。

表2-1は、各種埋込みSQL文をグループ化したもので、表2-2は、各種実行文をグループ化したものです。

表2-1 埋込みSQL文

宣言SQL	説明
STATEMENT	用途
ARRAYLEN*	PL/SQL でのホスト配列の使用
BEGIN DECLARE SECTION* END DECLARE SECTION*	ホスト変数の宣言
DECLARE*	Oracle オブジェクトの命名
INCLUDE*	ファイルへのコピー
TYPE*	データ型の同値化
VAR*	変数の同値化
WHENEVER*	ランタイム・エラーの処理

*対話形式はありません。

表2-2 実行SQL文とその説明

実行SQL	説明
STATEMENT	用途

実行SQL	説明
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*	
DELETE	Oracle データの問合せおよび操作
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	

実行SQL	説明
UPDATE	
COMMIT	トランザクションの処理
ROLLBACK	
SAVEPOINT	
SET TRANSACTION	
DESCRIBE*	動的 SQL の使用
EXECUTE*	
PREPARE*	
ALTER SESSION	セッションの制御
SET ROLE	

*対話形式はありません。

2.1.3 埋込みSQL構文

作成したアプリケーションでは、SQL文とホスト言語の文を自由に混在させ、SQL文にホスト言語の変数を使用できます。SQL文をホスト・プログラムに組み込むための唯一の特殊要件は、SQL文をキーワードのEXEC SQLで開始し、ホスト言語の終了文字で終了することです。プリコンパイラでは、すべての実行可能なEXEC SQL文がSQLLIBランタイム・ライブラリへのコールに変換されます。

大部分の埋込みSQL文と対話型のSQL文との違いは、新しい句が1つ追加されている、あるいはプログラム変数が使用されているという点のみです。次の対話型と埋込み型のROLLBACK文を比較してください。

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

埋込みSQL構文の概要は、『*Oracle Database SQL言語リファレンス*』を参照してください。

2.1.4 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、処理を実行する前に各SQL文およびトランザクションの構成がわかります。つまり、発行されるSQLコマンド、変更されるデータベースの表、更新される列などが事前にわかります。

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。したがって、関係するSQLコマンド、データベース表および列が、実行時までわからないことがあります。

動的SQLは、プログラムの実行時にSQL文を受け取るかまたは作成し、データ型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。

2.1.5 埋込みPL/SQLブロック

Oracleプリコンパイラでは、PL/SQLブロックが1つの埋込みSQL文と同様に扱われます。したがって、アプリケーション・プログラム内でSQL文を配置できる位置であれば、どこにでもPL/SQLブロックを記述できます。PL/SQLをホスト・プログラムに埋め込むには、単にPL/SQLと共有する変数を宣言し、PL/SQLブロックをEXEC SQL EXECUTEおよびEND-EXECキーワードで囲みます。

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

2.1.6 ホスト変数および標識変数

ホスト変数は、ホスト言語で宣言され、Oracleで共有される(つまり、プログラムとOracleの両方がその値を参照できる)スカラー変数または配列変数です。ホスト変数は、Oracleとプログラムとの間で通信を行うためのキーです。

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

ホスト変数は、式を使用できる場所であればどこでも使用できます。ただし、SQL文では、Oracleオブジェクトと区別するために、ホスト変数に接頭辞としてコロン(:)を付ける必要があります。

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。インジケータ変数は、関連付けられたホスト変数の値または条件を示す整変数です。インジケータ変数は、入力ホスト変数へのNULLの割当てと、出力ホスト変数に含まれるNULLまたは切捨て値の検出に使用されます。*NULL*値は、欠落している値、不明な値または適用不能な値です。

SQL文の場合、インジケータ変数には接頭辞のコロンを付けて、関連付けられたホスト変数の直後に記述する必要があります(そうしないで、さらに読みやすくするには、インジケータ変数の前にオプション・キーワードのINDICATORを付けます)。

2.1.7 Oracleのデータ型

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

Oracleでは、内部データ型と外部データ型という2種類のデータ型が認識されます。内部データ型は、Oracleでデータベース列にデータを格納する方法を指定します。Oracleでは、データベース擬似列を表す内部データ型も使用し、データベース擬似列からは、特定のデータ項目が戻されますが、表には実際の列はありません。

外部データ型は、データがホスト変数にどのように格納されるかを指定します。ホスト・プログラムからOracleにデータが入力されると、Oracleでは必要に応じて、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間で変換を行います。Oracleからホスト・プログラムにデータが出力されると、Oracleでは必要に応じて、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間で変換を行います。

2.1.8 配列

Oracleプリコンパイラでは、配列ホスト変数(ホスト配列と呼ばれる)を定義して、1つのSQL文で操作できます。配列に対するSELECT、FETCH、DELETE、INSERTおよびUPDATE文を使用すると、大量のデータの問合せおよび操作が容易にできます。

2.1.9 データ型の同値化

Oracleプリコンパイラではデータ型を同値化できるため、アプリケーションの柔軟性が向上します。つまり、Oracleで入力データを解釈し、出力データの書式を設定する方法をカスタマイズできます。

変数ごとに、サポートされているホスト言語のデータ型をOracleの外部データ型と同値化できます。

2.1.10 プライベートSQL領域、カーソルおよびアクティブ・セット

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

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

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

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

2.1.11 トランザクション

トランザクションとは、論理的に関連のある一連のSQL文です(ある銀行勘定貸方に記帳し、別の銀行の借方に記帳する2つのUPDATEなど)。Oracleでは、トランザクションは1単位として扱われるため、それぞれの文による変更はすべて同時に確定されるか、取り消されるかします。現行のトランザクションは、最後のデータ定義、COMMITまたはROLLBACK文が実行された後に実行されるすべてのデータ操作文で構成されます。

データベースの整合性を確保するために、Oracleプリコンパイラでは、COMMIT文、ROLLBACK文およびSAVEPOINT文を使用して、トランザクションを定義できます。COMMITでは、現行のトランザクション中にデータベースに加えられた変更が確定されます。ROLLBACKでは、現行のトランザクションを終了し、トランザクションの開始以降に加えられた変更がすべて取り消されます。SAVEPOINTでは、トランザクションの現在の位置にマークが付けられ、ROLLBACKと併用することで、トランザクションを部分的に取り消すことができます。

2.1.12 エラーと警告

埋込みSQL文を実行すると、エラーまたは警告が発生する場合があります。これらの結果を処理する方法が必要です。Oracleプリコンパイラには、4つのエラー処理方法が用意されています。

- SQLCODE状態変数
- SQLSTATE状態変数
- SQL通信領域(SQLCA)およびWHENEVER文
- Oracle通信領域(ORACA)

SQLCODE/SQLSTATE状態変数

SQL文の実行後、OracleサーバーからSQLCODEまたはSQLSTATEという変数にステータス・コードが戻されます。ステータス・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およびWHENEVER文

SQLCAは、Oracleでプログラムにランタイム・ステータス情報を渡すために使用されるプログラム変数を定義するデータ構造体です。SQLCAを使用すると、直前に試みた処理に関するOracleからのフィードバックに基づいて、異なる処理を実行できます。たとえば、DELETE文が成功したかどうかを確認し、成功した場合は削除された行数を確認できます。

WHENEVER文を使用すると、Oracleでエラーまたは警告の状態が検出されたときに自動的に実行されるアクションを指定できます。これらのアクションには、次の文の処理続行、サブルーチンのコール、ラベル付きの文への分岐、停止などがあります。

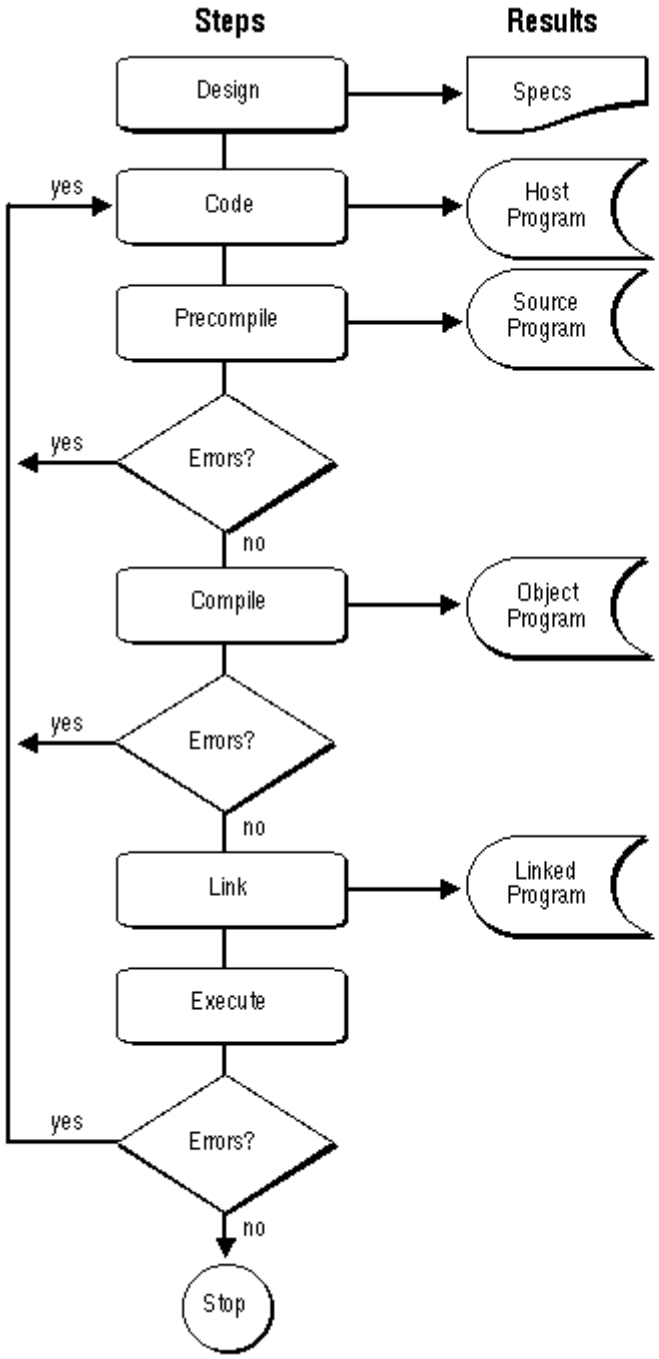
ORACA

ランタイム・エラーについてSQLCAで提供されるより多くの情報が必要な場合は、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統計情報、現行のSQL文に関する情報、オプションの設定、およびシステムの統計情報が含まれます。

2.2 埋込みSQLアプリケーションの開発ステップ

図2-1は、埋込みSQLアプリケーションの開発プロセスを示しています。

図2-1 アプリケーションの開発過程



図のように、プリコンパイルの結果、通常のコンパイルが可能なソース・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。

2.3 サンプル・プログラム

埋込みSQLをよく理解する方法の1つは、サンプル・プログラムの例を学習することです。

WHENEVER文によるエラーを処理するために、次のプログラムではOracleに接続し、ユーザーに従業員番号の入力を要求して、データベースに従業員の名前、給料、歩合を問合せ、その情報を表示して終了します。

```
-- declare host and indicator variables
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_number INTEGER;
 emp_name CHARACTER(10);
 salary REAL;
 commission REAL;
 ind_comm SMALLINT; -- indicator variable
EXEC SQL END DECLARE SECTION;
```

```
-- copy in the SQL Communications Area
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
-- handle processing errors
EXEC SQL WHENEVER SQLERROR DO sql_error;
-- log on to Oracle
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
display 'Employee number? ';
read emp_number;
-- query database for employee's name, salary, and commission
-- and assign values to host variables
EXEC SQL SELECT ENAME, SAL, COMM
 INTO :emp_name, :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
display 'Employee Salary Commission';
display '-------- ------ ----------';
-- display employee's name, salary, and commission (if not null)
IF ind_comm = -1 THEN -- commission is null
 display emp_name, salary, 'Not applicable';
ELSE
 display emp_name, salary, commission;
ENDIF;
-- release resources and log off the database
EXEC SQL COMMIT WORK RELEASE;
display 'Have a good day';
exit program;
ROUTINE sql_error
BEGIN
 -- avoid an infinite loop if the rollback results in an error
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 -- release resources and log off the database
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

2.4 サンプル表

このマニュアルの大部分のプログラミングの例では、DEPTとEMPの2つのサンプル・データベース表を使用しています。これらの定義は、次のとおりです。

```
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))
```

2.4.1 サンプル・データ

DEPT表とEMP表にはそれぞれ、次のデータの行が含まれています。

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

3 プログラム要件への対応

この章の項目は次のとおりです。

Oracle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、データ型変換、イベント処理およびOracleへのアクセスが必要になります。この章では、これらの要件にどのように対応するかを説明します。変数の宣言、通信領域の宣言、およびOracleデータベースへの接続を行う埋込みSQLコマンドについて学習します。Oracleデータ型、グローバリゼーション・サポート、データ変換、データ型の同値化の利用方法についても学習します。最後の2項では、プログラムにOCIコールを埋め込む方法と、X/Openアプリケーションの開発方法について説明します。

3.1 宣言部

SQL文で使用されるすべてのプログラム変数(つまり、すべてのホスト変数)は、宣言部で宣言する必要があります。SQL文で宣言されていないホスト変数を使用すると、プリコンパイラではエラー・メッセージが出ます。エラー・メッセージの完全なリストは、『*Oracle Database*エラー・メッセージ』を参照してください。

宣言部は、次の文で始まります。

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

そして、次の文で終了します。

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

COBOLでは、文の終了文字はEND-EXECです。FORTRANでは改行です。

これら2つの文の間に使用できるのは、次の項目のみです。

- ホスト変数とインジケータ変数の宣言
- EXEC SQL DECLARE文
- EXEC SQL INCLUDE文

- EXEC SQL VAR文
- EXEC ORACLE文
- ホスト言語のコメント

プリコンパイルの単位ごとに複数の宣言部を使用できます。さらに、1つのホスト・プログラムには、別々にプリコンパイルされる単位を複数含めることができます。

3.1.1 例

次の例では、プログラムで後から使用する4つのホスト変数を宣言しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name CHARACTER(10);
 salary REAL;
 commission REAL;
EXEC SQL END DECLARE SECTION;
```

ホスト変数の宣言の詳細は、ホスト変数の宣言および参照を参照してください。

3.2 INCLUDE文

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。これはCOBOLのCOPYコマンドに似ています。次に例を示します。

```
-- copy in the SQLCA file
EXEC SQL INCLUDE SQLCA;
```

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。

任意のファイルをインクルードできます。ファイルにSQLが埋め込まれている場合、インクルードされるファイルのみがプリコンパイルの対象となるため、そのファイルも必ずインクルードしてください。ファイル拡張子を指定しないと、プリコンパイラでは言語に依存するソース・ファイル用のデフォルトの拡張子であると想定されます(このマニュアルに対する使用ホスト言語の補足資料を参照してください)。

インクルードするファイルのディレクトリ・パスは、プリコンパイラ・オプションを指定することで設定できます。

```
INCLUDE=<path>
```

*path*のデフォルト値はカレント・ディレクトリです。(ここでは、ディレクトリはファイルの場所の索引です。)

プリコンパイラでは、最初にカレント・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、最後に標準のINCLUDEファイル用のディレクトリを検索します。したがって、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。標準以外のファイルについては、現在のディレクトリに格納されている場合を除いて、INCLUDEを使用してディレクトリ・パスを指定する必要があります。

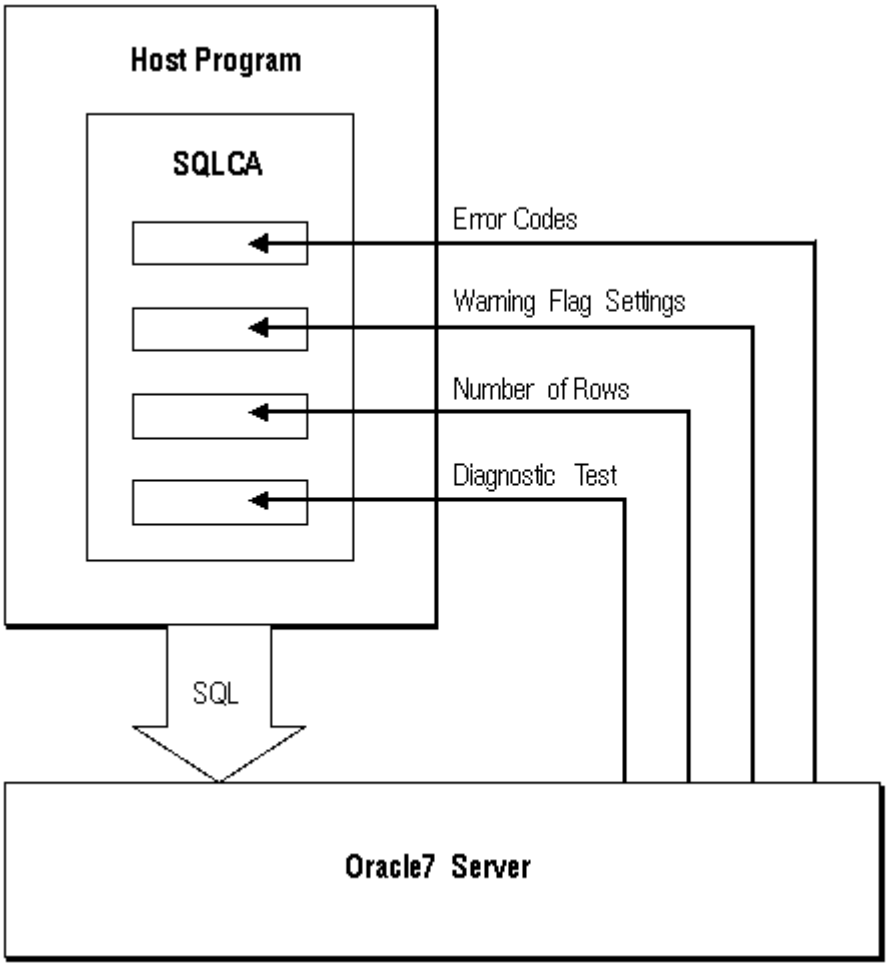
大文字と小文字が区別されるオペレーティング・システム(UNIXなど)を使用している場合は、ファイルを格納したときと同じファイル名を大文字と小文字を区別して指定してください。ディレクトリ・パスを指定するための構文はシステムによって異なります。使用しているシステム固有のOracleマニュアルを参照してください。

3.3 SQLCA

SQLCAは、診断チェックおよびイベント処理を行うためのデータ構造体です。実行時には、Oracleからプログラムに渡されるステータス情報がSQLCAで保持されます。SQL文の実行後、Oracleにより、図3-1に示すようにSQLCA変数が設定され、結

果が示されます。

図3-1 SQLCAの更新



このように、INSERT文、UPDATE文またはDELETE文の実行の成否を確認でき、成功した場合には、何行処理されたかを確認できます。文の実行に失敗した場合は、原因の詳細な情報が得られます。

MODE={ANSI13|ORACLE}のとき、SQLCAをハードコーディングするか、INCLUDE文を使用してプログラムにコピーするかにより、SQLCAを宣言する必要があります。SQLCAの宣言方法および使用方法は、SQL通信領域の使用についてを参照してください。

3.4 Oracleのデータ型

Oracleでは、内部データ型と外部データ型という2種類のデータ型が認識されます。内部データ型は、Oracleでデータベース列にデータを格納する方法を指定します。Oracleでは、データベース擬似列を表す内部データ型も使用されます。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。

プリコンパイル時に、宣言部の各ホスト変数は、外部データ型コードに関連付けられます。SQL文で使用するすべてのホスト変数のデータ型コードは、実行時にOracleに渡されます。Oracleは、コードを使用して内部データ型と外部データ型の間で変換します。

ノート:

動的 SQL 方法 4 またはデータ型の同値化を使用すると、デフォルトのデータ型変換をオーバーライドできます。動的 SQL の方法 4 の詳細は、方法 4 の使用方法を参照してください。データ型の同値化の詳細は、データ型の同値化を参照してください。

3.4.1 内部データ型

表3-1は、Oracleでデータベースの列および擬似列に使用される内部データ型の一覧です。

表3-1 列および擬似列のデータ型

名前	コード	説明
CHAR	96	固定長文字列(255 バイト以下)
DATE	12	7 バイト、固定長日付/時刻値
LONG	8	可変長文字列(2147483647 バイト以下)
LONG RAW	24	可変長バイナリ・データ(2147483647 バイト以下)
MLSLABEL	105	可変長バイナリ・ラベル(5 バイト以下)
NUMBER	2	固定数または浮動小数点数
RAW	23	可変長バイナリ・データ(255 バイト以下)
ROWID	11	固定長バイナリ値
VARCHAR2	1	可変長文字列(2000 バイト以下)

これらの内部データ型は、ホスト言語のデータ型とは大きく異なる場合があります。たとえば、NUMBERデータ型は、移植性、精度(四捨五入のエラーなし)および正しい照合のために設計されました。これに相当するデータ型を持つホスト言語はありません。

次に内部データ型について簡単に説明します。詳細は、「データ型」を参照してください。

3.4.2 CHAR

CHARデータ型は、固定長の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、データベースの文字セットによって決まります。CHARデータ型には、255バイトまでの最大幅を指定できるオプションのパラメータがあります。構文は次のとおりです。

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

最大幅の指定には、定数または変数は使用できません。整数リテラルを使用する必要があります。最大幅を指定しない場合、デフォルトで1になります。CHAR(*n*)列の最大幅は、文字単位ではなくバイト単位で指定することを思い出してください。そのため、CHAR(*n*)列にマルチバイト(2バイト)文字が格納される場合、その最大幅は*n*/2文字未満になります。

3.4.3 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。日付の部分はデフォルトで現在の月

の1日に、時刻の部分は午前0時に設定されます。

内部的に、DATEはバイナリ形式で保存されます。DATE列値をプログラムの文字列に変換するとき、Oracleではセッション用にデフォルトの書式マスクが使用されます。ユリウス日の日付など、別の日付/時刻情報が必要な場合は、TO_CHARファンクションに書式マスクを指定して使用してください。DATE列値と文字列との間の変換には、常にVARCHAR2またはSTRINGなどの(外部の)文字データ型を使用してください。

3.4.4 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、文字の配列、さらには短い文書まで格納できます。LONGデータ型は、VARCHAR2データ型と似ていますが、LONG列の最大幅が2147483647バイト(2GB)である点が異なります。

LONG列は、UPDATE文、INSERT文および(大部分の)SELECT文で使用できますが、式、ファンクション・コール、またはWHERE、GROUP BYおよびCONNECT BYなどのSQL句では使用できません。各データベース表で使用できるLONG列は1つのみで、その列には索引は付けられません。

3.4.5 LONG RAW

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

LONG RAWデータはLONGデータに似ていますが、OracleではLONG RAWデータの意味は解釈されず、あるシステムから別のシステムにLONG RAWデータを送信しても、文字セットの変換は行われない点が異なります。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。

3.4.6 MLSLABEL

MLSLABELデータ型は、可変長のバイナリ・オペレーティング・システム・ラベルの格納に使用します。Oracleでは、データのアクセス制御にラベルを使用します。

MLSLABELデータ型を使用すれば、データベース列を定義できます。データ型がMLSLABELの列には、任意の有効なオペレーティング・システム・ラベルを挿入できます。ラベルがテキスト形式の場合、Oracleではそのラベルをバイナリ値に自動的に変換します。テキスト文字列の長さは、最大255バイトです。ただし、MLSLABEL値の内部での長さは、2バイトから5バイトの間です。

MLSLABEL列から値を選択し、文字変数に代入することもできます。Oracleでは、内部バイナリ値はVARCHAR2値から自動的に変換されます。

3.4.7 NUMBER

NUMBERデータ型は、固定数または浮動小数点数の格納に使用し、事実上サイズに制限はありません。精度(合計桁数)と位取り(四捨五入が発生する位を決定)を指定できます。

NUMBER値の最大精度は38です。大きさの範囲は1.0E-129から9.99E125です。位取りの範囲は-84から127です。たとえば、位取りが-3とは、数が1000単位で近似されることを意味します(3456は3000になります)。位取りが2とは、値が1/100単位で近似されることを意味します(3.456は3.46になります)。

精度と位取りを指定すると、Oracleはデータを格納する前に追加の整合性チェックをします。値が精度を超えると、Oracleはエラー・メッセージを発行します。値が位取りを上回ると、Oracleは値を丸めます。

3.4.8 RAW

RAWデータ型は、バイナリ・データまたはバイト文字列の格納に使用します(グラフィックス文字のシーケンスなど)。RAWデータは、

Oracleでは解釈されません。

RAWデータ型には、255バイトまでの最大幅を指定できる必須パラメータを指定します。構文は次のとおりです。

```
RAW(maximum_width)
```

最大幅の指定には、定数または変数は使用できません。整数リテラルを使用する必要があります。

RAWデータはCHARデータと似ていますが、OracleではRAWデータの意味は解釈されず、あるシステムから別のシステムにRAWデータを送信しても、文字セットの変換(たとえば、7ビットASCIIからEBCDIC Code Page 500への変換)は行われない点が異なります。

3.4.9 ROWID

内部的には、Oracleデータベース内のすべての表にはROWIDという擬似列が1つあり、ここには行*ID*というバイナリ値が格納されます。ROWIDは行を一意に識別し、特定の行にアクセスする最速の方法です。

3.4.10 VARCHAR2

VARCHAR2データ型は、可変長文字列の格納に使用します。文字列が内部でどのように表されるかは、データベース文字セット(たとえば、7ビットASCIIまたはEBCDIC Code Page 500など)によって異なります。

VARCHAR2データベース列の最大幅は2000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から2000の範囲の整数リテラルです。

VARCHAR2(*n*)列の最大幅は、文字単位ではなくバイト単位で指定します。そのため、VARCHAR2(*n*)列にマルチバイト(2バイト)文字が格納される場合、その最大幅は*n*/2文字未満になります。

3.4.11 SQL疑似列および関数

SQLでは、表3-2の擬似列を認識し、擬似列からは特定のデータ項目が戻されます。

表3-2 擬似列のデータ型

擬似列	内部データ型
CURRVAL	NUMBER
LEVEL	NUMBER
NEXTVAL	NUMBER
ROWID	ROWID
ROWLABEL	MLSLABEL
ROWNUM	NUMBER

疑似列は、実際に表に存在する列ではありません。ただし、疑似列は列のように扱われるため、疑似列の値を表からSELECTする必要があります。ダミーの表から擬似列の値を選択すると便利な場合があります。

さらに、SQLでは、表3-3のパラメータがなく、特定のデータ項目を戻すファンクションを認識します。

表3-3 パラメータのないファンクションのデータ型

ファンクション	内部データ型
SYSDATE	DATE
UID	NUMBER
USER	VARCHAR2

SQLの擬似列およびファンクションは、SELECT文、INSERT文、UPDATE文およびDELETE文で参照できます。次の例では、SYSDATEを使用して、従業員が雇用されてからの月数を計算しています。

```
EXEC SQL SELECT MONTHS_BETWEEN(SYSDATE, HIREDATE)
 INTO :months_of_service
 FROM EMP
 WHERE EMPNO = :emp_number;
```

これから、SQLの疑似列および関数を簡単に説明します。詳細は、『*Oracle Database SQL言語リファレンス*』を参照してください。

CURRVALは、指定された順序における現行の番号を戻します。CURRVALを参照する前に、NEXTVALを使用して順序番号を生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。

LEVELをSELECT CONNECT BY文で使用して、表の一部または全部の行をツリー構造に組み込むことができます。ORDER BY句またはGROUP BY句でLEVELを使用すると、ツリー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、またはブランチから上へ)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。順序を作成した後、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、*partno*という順序で部品番号を割り当てます。

```
EXEC SQL INSERT INTO PARTS
 VALUES (partno.NEXTVAL, :description, :quantity, :price);
```

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やされます。NEXTVALを参照すると、現在の順序番号がCURRVALに格納されます。

ROWIDは、16進数で行アドレスを戻します。

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、選択された行にROWNUMが割り当てられた後にソートが実行されます。

ROWNUMを使用して、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を使用した場合は、取得する行数が制限されるのみで、SELECT文の処理は停止されません。WHERE句でのROWNUMの適切な使用方法は、次の方法のみです。

```
... WHERE ROWNUM < constant;
```

これは、ROWNUMの値は行が取得されたときにのみ増加するためです。次のように指定した場合、4行目までは取り出されないた

め、この検索条件が満たされません。

```
... WHERE ROWNUM = 5;
```

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、Oracleカレント・ユーザーの名前を戻します。

3.4.12 ROWLABEL列

SQLでは、Oracleでデータベースごとに作成される特殊な列ROWLABELも認識されます。他の列と同様に、ROWLABELは、SQL文で参照できます。ROWLABELは、行のオペレーティング・システムのラベルを戻します。

ROWLABELの一般的な使用方法、問合せ結果のフィルタ処理です。たとえば、次の文ではセキュリティ・レベルがunclassifiedより高い行のみが数えられます。

```
EXEC SQL SELECT COUNT(*) INTO :head_count FROM EMP
 WHERE ROWLABEL > 'UNCLASSIFIED';
```

3.4.13 外部データ型

表3-4のように、外部データ型にはすべての内部データ型と、サポートされている他のホスト言語で使用するいくつかのデータ型が含まれています。たとえば、STRING外部データ型は、C言語ではNULLで終了する文字列を指し、DECIMALデータ型は、COBOLパック10進数を指します。データ型の名前はデータ型の同値化で使用し、データ型のコードは動的SQLの方法4で使用します。

表3-4 外部データ型

名前	コード	説明
CHAR	1 96	可変長文字列(65535 バイト以下)、固定長文字列(65535 バイト以下)(ノート 1 を参照)
CHARF	96	<= 65535 バイト、固定長文字列
CHARZ	97	NULL で終了する固定長文字列(65535 バイト以下)(ノート 2 を参照)
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
FLOAT	4	4 バイトまたは 8 バイトの浮動小数点数
INTEGER	3	符号付き整数(2 バイトまたは 4 バイト)

名前	コード	説明
LONG	8	固定長文字列(2147483647 バイト以下)
LONG RAW	24	固定長バイナリ・データ(217483647 バイト以下)(ノート 3 を参照)
LONG VARCHAR	94	可変長文字列(217483643 バイト以下)(ノート 3 を参照)
LONG VARRAW	95	<= 217483643 バイト、可変長バイナリ・データ
MLSLABEL	106	可変長バイナリ・データ(2 バイトから 5 バイト)
NUMBER	2	整数または浮動小数点数
RAW	23	固定長バイナリ・データ(65535 バイト以下)(ノート 2 を参照)
ROWID	11	固定長バイナリ値(通常 13 バイト)
STRING	5	NULL で終了する可変長文字列(65535 バイト以下)(ノート 2 を参照)
UNSIGNED	68	2 バイトまたは 4 バイトの符号なし整数
VARCHAR	9	可変長文字列(65533 バイト以下)(ノート 3 を参照)
VARCHAR2	1	可変長文字列(65535 バイト以下)(ノート 2 を参照)
VARNUM	6	可変長バイナリ数
VARRAW	15	可変長バイナリ・データ(65533 バイト以下)(ノート 3 を参照)

ノート:

1. CHAR は、MODE={ORACLE|ANSI13|ANSI14} の場合はデータ型 1、MODE=ANSI の場合はデータ型 96 です。
2. 一部のプラットフォームでは、最大サイズが 32767 (32K)です。
3. EXEC SQL VAR 文には n バイト長のフィールドを含めないでください。

3.4.14 CHAR

CHARの動作は、DBMSオプションとMODEオプションの設定により異なります。

3.4.15 CHARF

MODE=ANSIの場合、OracleではCHARデータ型はすべての文字ホスト変数に割り当てられます。CHAR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、CHAR値の最大長は65535バイト(64KB)です。DBMSオプションとMODEオプションの関係の詳細は、表6-4を参照してください。

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

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

出力時。Oracleからは出力ホスト変数に指定したバイト数が戻され、必要に応じて空白が埋め込まれて、出力値がターゲット・ホスト変数に割り当てられます。NULLが戻される場合、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、Oracleはホスト変数に割り当てる前に出力値を切り捨てます。標識変数が使用可能な場合、標識変数は出力値の元の長さに設定されます。

3.4.16 CHARZ

CHARZデータ型は、NULLで終了する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、CHARZ値の最大長は65,535バイトです。Pro*COBOLまたはPro*FORTRANでは、この外部データ型は必要ありません。

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

出力時、CHARZデータ型とCHARデータ型の機能は同じです。出力値にはNULL終了文字が付けられ、必要に応じて空白文字も埋め込まれます。

3.4.17 DATE

DATEデータ型は、日付および時刻を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3-5に示すように、世紀、年、月、日、時(24時間形式)、分および秒は、左から右にこの順序で格納されます。

表3-5 DATEデータ型の例

バイト	1	2	3	4	5	6	7
意味	世紀	年	月	日	時	分	秒
例 1994 年 10 月 17 日午後 1 時 23 分 12 秒	119	194	10	17	14	24	13

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

3.4.18 DECIMAL

Pro*COBOLでは、DECIMALデータ型を使用して計算用のパック10進数の格納に使用します。COBOLでは、ホスト変数は暗黙的な小数点を持つ符号付きCOMP-3フィールドであることが必要です。データ変換中に有効な桁が失われると、ホスト変数はアスタリスクで埋められます。

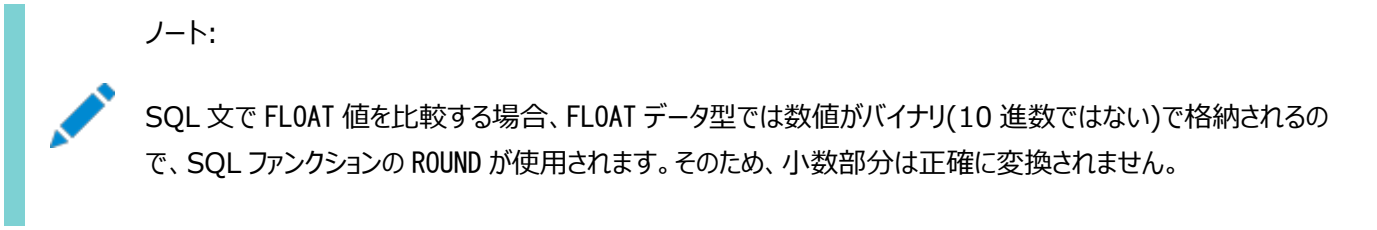
3.4.19 DISPLAY

Pro*COBOLでは、DISPLAYデータ型を使用して数値文字データを格納します。DISPLAYデータ型は、COBOLのDISPLAY SIGN LEADING SEPARATEの数値を参照し、通常、PIC S9(*n*)には*n* + 1バイトの記憶域が、PIC S9(*n*)V9(*d*)には*n* + *d* + 1バイトの記憶域が必要です。

3.4.20 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を格納するために使用します。数値はご使用のコンピュータの浮動小数点書式を使用して表され、通常、4または8バイトの記憶域を必要とします。入力および出力ホスト変数に長さを指定する必要があります。

Oracleでは、数値の内部形式が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現できます。



ノート:

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 ファンクションの ROUND が使用されます。そのため、小数部分は正確に変換されません。

3.4.21 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2進数です。1語内のバイトの順序は、システムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時には、列値が浮動小数点数であれば、小数部が切り捨てられます。

3.4.22 LONG

LONGデータ型は、固定長の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(2GB)である点で異なります。

3.4.23 LONG RAW

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

LONG RAWデータはLONGデータに似ていますが、OracleではLONG RAWデータの意味は解釈されず、あるシステムから別のシステムにLONG RAWデータを送信しても、文字セットの変換は行われない点が異なります。

3.4.24 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バイト長のフィールドの後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バイト長のフィール

ドを含めないでください。

3.4.25 LONG VARRAW

LONG VARRAWデータ型は、バイナリ・データまたはバイト文字列の格納に使用します。LONG VARRAW変数では、4バイト長の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バイト長のフィールドを含めないでください。

3.4.26 MLSLABEL

MLSLABELデータ型は、可変長のバイナリ・オペレーティング・システム・ラベルの格納に使用します。Oracleでは、データのアクセス制御にラベルを使用します。MLSLABELデータ型を使用して列を定義できます。データ型がMLSLABELの列には、任意の有効なオペレーティング・システム・ラベルを挿入できます。

入力時。Oracleでは、入力値は有効なオペレーティング・システム・ラベルであるバイナリ・ラベルに変換します。無効のラベルの場合は、エラー・メッセージが出ます。ラベルが有効ならば、ターゲット・データベース列に格納されます。

出力時。Oracleでは、バイナリ・ラベルを文字列に変換し、文字列はCHAR、CHARZ、STRING、VARCHARまたはVARCHAR2のいずれかのデータ型です。

3.4.27 NUMBER

NUMBERデータ型は、固定または浮動小数点のOracle数値を格納するために使用します。精度および位取りを指定できます。NUMBER値の最大精度は38です。大きさの範囲は1.0E-129から9.99E125です。位取りの範囲は-84から127です。

NUMBER値は、可変長形式(1バイトの指数部に最大20バイトの仮数部が続く)で格納されます。指数バイトの上位1ビットは符号ビットであり、正数の場合に設定します。下位7ビットは指数を表し、100の位を底とし、オフセットは65です。

各仮数バイトは基数100で範囲1から100の数字です。正数の場合は、それに1を加算した数字となります。負数の場合、101からそれが引かれ、20バイトの仮数部があるのでないかぎり、102を含んでいるバイトがデータ・バイトに追加されます。各仮数バイトは、2つの10進数を表すことができます。仮数は正規化され、先行する0(ゼロ)は格納されません。仮数部には最大20のデータ・バイトを使用できますが、正確さが保証されているのは19バイトのみです。19個のデータ・バイトは、それぞれ基数100の数字を表し、38桁の最大精度を提供します。

出力時、ホスト変数にはOracleで内部的に表されたとおりの数値が含まれます。考えられる最大の数値を含めるには、出力ホスト変数は21バイトの長さが必要です。数値を表すために使用されるバイトのみが戻されます。Oracleでは出力値の空白を埋め込んだり、NULLで終了させたりしません。戻り値の長さを知る必要がある場合は、かわりにVARNUMデータ型を使用してください。通常、このデータ型を使用する理由はほとんどありません。

3.4.28 RAW

RAWデータ型は、固定長のバイナリ・データまたはバイト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、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、Oracle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文字セットの変換は行われません。

3.4.29 ROWID

ROWIDデータ型は、バイナリ行IDを固定長(通常は13バイト)に格納するために使用します。フィールド・サイズは、ポート固有です。そのため、使用しているシステム固有のOracleマニュアルをチェックしてください。VARCHAR2ホスト変数を使用すると、読取り可能な形式で行IDを格納できます。行IDを選択またはフェッチしてVARCHAR2ホスト変数に入れると、Oracleで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、次の行IDがあるとします。

```
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、行IDをVARCHAR2ホスト変数にフェッチし、ホスト変数をUPDATE文またはDELETE文のWHERE句のROWID擬似列と比較します。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。

ノート:

完全な移植性が必要な場合、あるいはアプリケーションで Transparent Gateway を使用して Oracle 以外のデータベースと通信する場合は、VARCHAR2 ホスト変数を宣言するときに、最大長を(18 ではなく)256 に指定してください。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推測できませんが、ホスト変数は SQL 文中で正常に機能します。

3.4.30 STRING

STRINGデータ型は、VARCHAR2データ型に似ていますが、STRING値は常にヌル文字で終了するという違いがあります。

入力時。Oracleは指定された長さを使用して、ヌル終端文字のスキャンを制限します。ヌル終端文字が見つからなければ、エラーが生成されます。長さを指定しない場合、Oracleは最大長とみなします。大部分のプラットフォームでは65535です。

STRING値の最小長は2バイトです。最初の文字がNULL終了文字で、指定した長さが2の場合、列がNOT NULLと定義されていなければ、OracleではNULLを挿入します。空白のみの値またはNULLで終了する値はそのまま格納されます。

出力時。Oracleは戻された最後の文字に1バイトのヌル文字を追加します。文字列長が指定された長さを超える場合は、Oracleは出力値を切り捨て、1バイトのヌル文字を追加します。

3.4.31 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。1語内のバイトの順序は、システムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時には、列値が浮動小数点数であれば、小数部が切り捨てられます。Pro*COBOLまたはPro*FORTRANでは、この外部データ型は必要ありません。

3.4.32 VARCHAR

VARCHARデータ型は、可変長文字列の格納に使用します。VARCHAR変数では、2バイト長のフィールドの後に65533バイト以下の文字列フィールドが続きます。ただし、VARCHAR配列要素では、文字列フィールドの最大長は65530バイトです。VARCHAR変数の長さを指定するときには、長さフィールド用の2バイトが含まれていることを確認してください。これより長い文字列には、LONG VARCHARデータ型を使用してください。EXEC SQL VAR文では、2バイト長のフィールドを含めないでください。

3.4.33 VARCHAR2

MODE=ORACLEの場合、OracleではVARCHAR2データ型をすべての文字ホスト変数に割り当てます。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、文字単位ではなくバイト単位で指定します。したがって、VARCHAR2(*n*)変数にマルチバイト・キャラクタを格納する場合、最大長は*n*文字未満になります。

入力時。Oracleは入力ホスト変数に指定されたバイト数を読み込み、後続の空白文字を取り除き、入力値をターゲット・データベース列に格納します。ここでは注意が必要です。未初期化ホスト変数には、一連のヌル文字が含まれている場合があります。したがって、常に文字入力ホスト変数の宣言長まで空白文字で埋めてください。(COBOL PIC X(*n*)およびFORTRAN CHARACTER*nの各変数では、これは自動的に行われます。)

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。入力値がすべて空白の場合は、空白がNULLと同様に扱われます。

文字値が有効な数を表す場合、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。Oracleからは出力ホスト変数に指定したバイト数が戻され、必要に応じて空白が埋め込まれて、出力値がターゲット・ホスト変数に割り当てられます。NULLが戻される場合、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、Oracleはホスト変数に割り当てる前に出力値を切り捨てます。標識変数が使用可能な場合、標識変数は出力値の元の長さに設定されます。

NUMBER列値は文字値に変換できます。文字ホスト変数の長さによって精度が決定します。ホスト変数の長さがその数に対して短すぎる場合は、科学表記法が使用されます。たとえば、列値abcdefg89を長さ6のホスト変数に選択すると、ホスト変数に値1.2E08が戻されます。

3.4.34 VARNUM

VARNUMデータ型はNUMBERデータ型に似ていますが、VARNUM変数の1バイト目に値の長さが格納される点が異なります。入力時には、ホスト変数の1バイト目を値の長さに設定する必要があります。出力時には、ホスト変数に長さとそれに続いてOracleで内部的に表現された数が含まれます。この数が最大になっても対応できるように、ホスト変数の長さは22バイトにする必要があります。列値をVARNUMホスト変数に選択した後、1バイト目をチェックすれば値の長さがわかります。

3.4.35 VARRAW

VARRAWデータ型は、可変長のバイナリ・データまたはバイト文字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。それより長い文字列の場合は、LONG VARRAWデータ型を使用します。EXEC SQL VAR文では、2バイト長のフィールドを含めないでください。VARRAW変数の長さは、その長さフィールドを参照すればわかります。

3.5 データ型変換

プリコンパイル時に、宣言部の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、プリコンパイラは整数ホスト変数にINTEGER外部データ型を割り当てます。SQL文で使用するすべてのホスト変数のデータ型コードは、実行時にOracleに渡されます。Oracleは、コードを使用して内部データ型と外部データ型の間で変換します。

Oracleでは、選択した列(または擬似列)値を出力ホスト変数に割り当てる前に、必要に応じて、その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をデータベース列に割り当てたり、比較したりする前には、必要に応じて、ホスト変数の外部データ型を列の内部データ型に変換します。

ただし、ホスト変数のデータ型は、データベース列のデータ型との互換性が必要です。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値の1234を2バイトの整数に変換できます。しかし、CHAR値の65543(大きすぎる数)や10F(10進数ではない数)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む文字列値はNUMBER値に変換できません。

数値の変換は、Oracle初期化ファイルのグローバリゼーション・サポート・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。

表3-6は、サポートされている内部データ型と外部データ型間の変換を示しています。

ノート:

凡例:

- 入力時には、ホスト文字列を Oracle の BBBBBBBB.RRRR.FFFF 形式にする必要があります。

  I = 入力のみ。出力時には、列値が同じ形式で戻されます。

  O = 出力のみ

- 入力時には、ホスト文字列をデフォルトの DATE 文字形式にする必要があります。

  I/O = 入力または出力。出力時には、列値が同じ形式で戻されます。

- 入力時には、ホスト文字列を 16 進数の形式にする必要があります。出力時には、列値は同じ形式で戻されます。

- 出力時には、列値は有効な数値を表している必要があります。

- 入力時には、長さが 2000 以下であることが必要です。

- 入力時には、列値は 16 進数の形式で格納されます。出力時、列値は 16 進フォーマットであることが必要です。

- 入力時には、ホスト文字列をテキスト形式の有効なオペレーティング・システム・ラベルにする必要があります。出力時には、列値は同じ形式で戻されます。

- 入力時には、ホスト文字列を RAW 形式の有効なオペレーティング・システム・ラベルにする必要があります。出力時には、列値は同じ形式で戻されます。

表3-6 内部データ型と外部データ型間の変換

外部	内部: CHAR	内部: DATE	内部: LONG	内部: LONG RAW	内部: MLSLABEL	内部: NUMBER	内部: RAW	内部: ROWID	内部: VARCHAR2

外部	内部: CHAR	内部: DATE	内部: LONG	内部: LONG RAW	内部: MLSLABEL	内部: NUMBER	内部: RAW	内部: ROWID	内部: VARCHAR2
CHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
CHARF	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
CHARZ	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I						I/O
DECIMAL	I/O		I			I/O			I/O
DISPLAY	I/O		I			I/O			I/O
FLOAT	I/O		I			I/O			I/O
INTEGER	I/O		I			I/O			I/O
LONG	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
LONG RAW	O		I	I/O			I/O		O
LONG VARCHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
LONG VARRAW	I/O		I	I/O			I/O		I/O
MLSLABEL	I/O		I/O		I/O				I/O
NUMBER	I/O		I			I/O			I/O
RAW	I/O		I	I/O			I/O		I/O
ROWID	I		I					I/O	I
STRING	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
UNSIGNED	I/O		I			I/O			I/O
VARCHAR	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O

外部	内部: CHAR	内部: DATE	内部: LONG	内部: LONG RAW	内部: MLSLABEL	内部: NUMBER	内部: RAW	内部: ROWID	内部: VARCHAR2
VARCHAR2	I/O	I/O	I/O	I	I/O	I/O	I/O	I/O	I/O
VARNUM	I/O		I			I/O			I/O
VARRAW	I/O		I	I/O			I/O		I/O

3.5.1 DATE値

ホスト変数値にDATE列値を選択すると、Oracleでは内部バイナリ値を外部文字値に変換する必要があります。そのため、暗黙的にSQLファンクションTO_CHARがコールされ、これによって文字列がデフォルトの日付書式で戻されます。デフォルトはOracleの初期化パラメータ、グローバリゼーション・サポートのDATE_FORMATによって設定されます。時刻やユリウス日などの他の情報を取得するには、書式マスクを指定してTO_CHARを明示的にコールする必要があります。

文字ホスト変数をDATE列に挿入するときにも変換が必要です。Oracleでは暗黙的にSQLファンクションTO_DATEがコールされ、デフォルトの日付書式が予想されます。他の書式で日付を挿入するには、書式マスクを指定してTO_DATEを明示的にコールする必要があります。

3.5.2 RAWおよびLONG RAWの値

文字ホスト変数にRAW列値またはLONG RAW列値を選択すると、Oracleでは内部バイナリ値を外部文字値に変換する必要があります。この場合、OracleからはRAWまたはLONG RAWデータのそれぞれのバイナリ・バイトが文字のペアとして戻されます。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ファンクションRAWTOHEXは、同じ変換を実行します。

文字ホスト値をRAW列またはLONG RAW列に挿入するとき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進値を表している必要があります。文字がニブルの16進数等価値でなければ、次のエラー・メッセージが出ます。

```
ORA-01465: invalid hex number
```

3.6 ホスト変数の宣言および参照

SQL文で使用するプログラム変数はすべて、ホスト変数として宣言する必要があります。ホスト変数は、ホスト言語の規則に従って宣言部で宣言します。通常の有効範囲規則を適用します。ホスト変数名は、長さに制限はありませんが、有効な文字は最初の31文字のみです。ANSI/ISO準拠のため、ホスト変数名は長さを18文字以下にし、先頭が文字で、連続したアンダースコアや後続のアンダースコアを含まないことが必要です。

ホスト変数の外部データ型およびそのソース・データベースまたはターゲット・データベース列の内部データ型は、同じである必要はありませんが、互換性が必要です。表3-6では、必要に応じてOracleで自動的に変換される互換性のあるデータ型を示しています。

Oracle プリコンパイラでは、ほとんどの組込みホスト言語データ型をサポートしています。サポートされるデータ型のリストは、使用ホスト言語の補足資料を参照してください。ユーザー定義のデータ型はサポートされていません。データ型の同値化については、次の項で説明します。

ユーザー定義の構造体の参照はできませんが、Pro*COBOLプリコンパイラを使用すれば、構造体の個々の要素をホスト変数であるかのように参照できます。ホスト変数が使用可能な場所ならば、そのような参照を使用できます。

3.6.1 例

次の例では、3つのホスト変数を宣言してから、SELECT文を使用して、ホスト変数*emp_number*の値に一致する従業員番号をデータベースで検索します。一致する行が見つかると、Oracleでは出力ホスト変数*dept_number*と*emp_name*をその行のDEPTNO列とENAME列の値に設定します。

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name CHARACTER(10);
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
...
display 'Employee number? ';
read emp_number;
EXEC SQL SELECT DEPTNO, ENAME INTO :dept_number, :emp_name
 FROM EMP
 WHERE EMPNO = :emp_number;
```

ホスト変数の使用方法の詳細は、ホスト変数の使用についてを参照してください。

3.6.2 VARCHAR変数

VARCHAR擬似型を使用すると、可変長文字列を宣言できます。(擬似型は、使用するホスト言語に固有のデータ型ではありません。)VARCHAR変数には2バイト長のフィールドの後に文字列フィールドが1つ続くことを思い出してください。たとえば、Pro*COBOLプリコンパイラでは、次のVARCHAR宣言がどのように処理されるか見てみましょう。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 ENAME PIC X(20) VARYING.
 EXEC SQL END DECLARE SECTION END-EXEC.
```

宣言は、配列と長さのメンバーを指定した次のCOBOLグループ項目に拡張されます。

```
01 ENAME.
 05 ENAME-LEN PIC S9(4) COMP.
 05 ENAME-ARR PIC X(20).
```

VARCHARの長さは、長さフィールドを参照すればわかります。文字列ファンクションまたは文字カウント・アルゴリズムを使用する必要はありません。

VARCHARの詳細は、このマニュアルに対する使用ホスト言語の補足資料を参照してください。

3.6.3 ホスト変数のガイドライン

ホスト変数の宣言および参照については、次のガイドラインに従ってください。ホスト変数には次の必要条件があります。

- 宣言部で明示的に宣言します。
- SQL文およびPL/SQLブロックではコロン(:)の接頭辞を付けます。
- ホスト言語でサポートされているデータ型です。
- ソースまたはターゲットのデータベース列と互換性のあるデータ型です。

ホスト変数では、次のことはしないください。

- サブスクライブの対象
- ホスト言語文で先頭にコロン
- 列、表またはその他Oracleオブジェクトの識別に使用
- ALTERやCREATEなどのデータ定義文に使用
- Oracleの予約語(Oracleの予約語、キーワードおよびネームスペースを参照)

ホスト変数では次のことができます。

- SQL文で式が使用可能な場所での使用
- インジケータ変数と関連付けることができます。

3.7 インジケータ変数の宣言および参照

ホスト変数はすべて、オプションのインジケータ変数に関連付けることができます。インジケータ変数は、2バイトの整数として宣言部で定義し、SQL文では先頭にコロンを付け、INDICATORキーワードを使用しない場合には、ホスト変数の直後に指定する必要があります。

3.7.1 INDICATORキーワード

インジケータ変数の前にオプションのキーワードINDICATORを付けると読みやすくなります。その場合も、インディケータ変数の前にはコロンを付ける必要があります。正しい構文は次のとおりです。

```
:<host_variable> INDICATOR :<indicator_variable>
```

これは次の構文と同じ意味です。

```
:<host_variable>:<indicator_variable>
```

ホスト・プログラムでは、両方の形式の式を使用できます。

3.7.2 例

通常、インジケータ変数は、入力ホスト変数へのNULLの割当てと、出力ホスト変数に含まれるNULLまたは切り捨てられた値の検出に使用されます。次の例では、3つのホスト変数と1つのインジケータ変数を宣言してから、SELECT文を使用して、ホスト変数*emp_number*の値と一致する従業員番号をデータベースで検索します。一致する行が見つかると、出力ホスト変数*salary*と*commission*が、その行のSAL列とCOMM列の値に設定され、リターン・コードがインジケータ変数*ind_comm*に格納されます。次の文では、*ind_comm*を使用して、一連の処理を選択しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 salary REAL;
 commission REAL;
 ind_comm SMALLINT; -- indicator variable
EXEC SQL END DECLARE SECTION;
 pay REAL; -- not used in a SQL statement
display 'Employee number? ';
read emp_number;
EXEC SQL SELECT SAL, COMM
 INTO :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
IF ind_comm = -1 THEN -- commission is null
 set pay = salary;
```

```
ELSE
 set pay = salary + commission;
ENDIF;
```

詳細は、インジケータ変数の使用についてを参照してください。

3.7.3 インジケータ変数のガイドライン

インジケータ編集の宣言および参照については、次のガイドラインに従ってください。インジケータ変数には、次の必要条件があります。

- 宣言部で2バイト整数として明示的に宣言
- SQL文で先頭にコロン(:)
- SQL文およびPL/SQLブロックではホスト変数の直後に指定(INDICATORキーワードを前に付けない場合)

インジケータ変数では、次のことをしないでください。

- ホスト言語文で先頭にコロン
- ホスト言語の文中でホスト変数の直後に指定
- Oracleの予約語

3.8 データ型の同値化

データ型の同値化により、Oracleで入力データを解釈する方法と出力データの書式を設定する方法をカスタマイズできます。サポートされているホスト言語のデータ型は、変数ごとにOracleの外部データ型に同値化できます。

3.8.1 データ型の同値化を行う理由

データ型の同値化は、いくつかの点で役に立ちます。たとえば、COBOLプログラムでNULLで終了するホスト文字列を使用するとします。PIC Xホスト変数を宣言した後で、これを常にNULLで終了する外部データ型STRINGに同値化できます。

Oracleにデータを格納するが、解釈はしないという場合に、データ型の同値化を使用できます。たとえば、LONG RAWデータベース列に整数のホスト配列を格納する場合、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オーバーライドする場合にもデータ型の同値化を使用できます。Oracle初期化ファイルのグローバリゼーション・サポート・パラメータで特に指定されていない場合、DATE列値を選択して文字ホスト変数に入れると、Oracleからは次のような書式の9バイトの文字列が戻されます。

```
DD-MON-YY
```

ただし、文字ホスト変数をDATE外部データ型に同値化すると、内部書式の7バイトの値が戻されます。

3.8.2 ホスト変数の同値化

デフォルトでは、Oracleプリコンパイラは、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。(これらのデフォルトの割当ては、このマニュアルに対する補足資料で表にまとめられています。)デフォルトの割当ては、宣言部でホスト変数をOracleの外部データ型に同値化することによりオーバーライドできます。これをホスト変数の同値化と呼びます。

使用する構文は次のとおりです。

```
EXEC SQL VAR <host_variable>
 IS <ext_type_name> [({<length> | <precision>,<scale>})];
```

ここで、host_variableは、宣言部で先に宣言された入力または出力ホスト変数(あるいはホスト配列)です。VARCHARおよびVARRAW外部データ型が2バイト長のフィールドで、*n*バイトのデータ・フィールドが続く場合、*n*の値の範囲は1から65533になります。そのため、*type_name*がVARCHARまたはVARRAWの場合、*host_variable*には少なくとも3バイトの長さが必要です。

LONG VARCHARおよびLONG VARRAW外部データ型が4バイト長のフィールドで、*n*バイトのデータ・フィールドが続く場合、*n*の値の範囲は1から2147483643になります。そのため、*type_name*がLONG VARCHARまたはLONG VARRAWの場合、*host_variable*には少なくとも5バイトの長さが必要です。

ext_type_nameは、RAWやSTRINGなどの有効な外部データ型の名前です。

lengthは、有効な長さをバイトで指定する整数リテラルです。*length*の値は、外部データ型を指定するのに十分な長さにする必要があります。

*type_name*がDECIMALまたはDISPLAYの場合は、*length*ではなく、*precision*および*scale*を指定する必要があります。*type_name*がVARNUM、ROWIDまたはDATEの場合、*length*は事前に定義されているために指定できません。他の外部データ型の場合、*length*はオプションです。デフォルトで*host_variable*の長さに設定されます。

*length*を指定するときに、*type_name*がVARCHAR、VARRAW、LONG VARCHARまたはLONG VARRAWの場合は、データ・フィールドの最大長を指定してください。プリコンパイラでは、lengthフィールドの説明をします。*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を指定してください。

*precision*と*scale*は、それぞれ有効数字の数値と四捨五入が発生する位置を表します。たとえば、*scale*が2の場合、値は小数第2位に値が四捨五入されることを意味します(3.456は3.46になります)、scaleが-3のときは、1000の位に値が四捨五入されます(3456は3000になります)。

1から99までの*precision*および-84から99までの*scale*を指定できます。ただし、データベース列の精度および位取りの最大値は、それぞれ38と127です。したがって、*precision*が38を超えていると、*host_variable*の値はデータベース列に挿入できません。ただし、列値のscaleが99を超えると、*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の場合にのみ指定してください。

表3-7は、各外部データ型に使用されるパラメータを示しています。

3.8.3 例

EMP表から従業員名を選択し、NULLで終了する文字列が期待されるルーチンに渡すとします。これらの名前に、明示的にヌル終端文字を付ける必要はありません。次のように、ホスト変数をSTRING外部データ型に同値化するのみです。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 emp_name CHARACTER(11);
 EXEC SQL VAR emp_name IS STRING (11);
EXEC SQL END DECLARE SECTION;
```

ENAME列の幅は10文字のため、ヌル終端文字を含めるために、新しい*emp_name*に11文字を割り当てます。(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列から*emp_name*に入れる値をSELECTする場合、Oracleにより値はヌル文字で終了します。

表3-7 外部データ型のパラメータ

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

外部データ型	長さ	精度	位取り	デフォルトの長さ
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
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(ノート 1)	該当なし	該当なし	なし
LONG VARRAW	必須(ノート 1)	該当なし	該当なし	なし
MLSLABEL	必須	該当なし	該当なし	なし
NUMBER	該当なし	該当なし	該当なし	使用不可
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
ROWID	該当なし	該当なし	該当なし	13 バイト(ノート 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

ノート:

1. データ・フィールドが 65,533 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って異なります。

3.8.4 CHARFデータ型指定子の使用について

VAR文およびTYPE文でデータ型指定子CHARFを使用すると、DBMSの設定に関係なく、ホスト言語のデータ型を固定長のANSIデータ型CHARに同値化できます。

MODE=ANSIの場合、TYPE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CHAR(Oracleの外部データ型コード96)に同値化されます。ただし、MODE=ORACLEの場合、ホスト言語のデータ型は、ユーザーの意向に関係なく、可変長データ型VARCHAR2(コード1)に同値化されます。

ただし、ホスト言語のデータ型はいつでも固定長の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、MODE=ORACLE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。

3.8.5 ガイドライン

VARNUM値またはDATE値を入力するには、必ずOracleの内部形式で入力してください。Oracleでは、VARNUM値およびDATE値の出力には内部形式が使用されます。

列値をVARNUMホスト変数に選択した後、1バイト目をチェックすれば値の長さがわかります。表3-8では、戻されるVARNUM値の例を示しています。

表3-8 戻されるVARNUM値の例

10進数値	VARNUM値の長さバイト	VARNUM値の指数バイト	VARNUM値の仮数部バイト	VARNUM値の終了文字バイト
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
abcdefg	5	196	2, 24, 46, 68	該当なし

DATE値は、通常、プログラムによる値の出力(たとえば、表示)や入力に使用されめ、DD-MON-YYなどの文字書式に変換してください。

適したOracle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。

3.9 グローバリゼーション・サポート

広く使用されている7ビットまたは8ビットのASCIIおよびEBCDICの文字セットは、アルファベットを表すには十分ですが、日本語のようなアジアの言語には数千もの文字が含まれるものがあります。これらの言語では、1文字を表すために16ビット(2バイト)が必要です。Oracleではこうした様々な言語がどのように処理されるかについて説明します。

Oracleには、シングルバイトおよびマルチバイトの文字データを処理し、文字セット間で変換できるように、グローバリゼーション・サポートが用意されています。また、アプリケーションを異なる言語環境で実行することもできます。グローバリゼーション・サポートでは、数値書式および日付書式はユーザー・セッション用に指定された言語規則に自動的に適応します。したがって、グローバリゼーション・サポートにより、世界中のユーザーがそれぞれの母国語でOracleとやりとりできます。

様々なグローバリゼーション・サポート・パラメータを指定して、言語によって異なる機能の操作を制御できます。これらのパラメータのデフォルト値は、Oracleの初期化ファイルで設定できます。表3-9は、それぞれのグローバリゼーション・サポート・パラメータの指定内容を示しています。

表3-9 グローバリゼーション・サポート・パラメータ

グローバリゼーション・サポート・パラメータ	指定内容
Globalization Support_LANGUAGE	言語によって異なる表記規則
Globalization Support_TERRITORY	地域によって異なる表記規則
Globalization Support_DATE_FORMAT	日付書式
Globalization Support_DATE_LANGUAGE	日および月の名前に使用する言語
Globalization Support_NUMERIC_CHARACTERS	10 進数文字およびグループ・セパレータ
Globalization Support_CURRENCY	各国通貨記号
Globalization Support_ISO_CURRENCY	ISO 通貨記号
Globalization Support_SORT	ソートの順序

主なパラメータは、Globalization Support_LANGUAGEおよびGlobalization Support_TERRITORYです。
Globalization Support_LANGUAGEでは、言語によって異なる次の機能のデフォルト値を指定します。

- サーバー・メッセージに使用する言語
- 日および月の名前に使用する言語
- ソートの順序

Globalization Support_TERRITORYでは、地域によって異なる次の機能のデフォルト値を指定します。

- 日付書式
- 小数点文字
- グループ・セパレータ
- 各国通貨記号
- ISO通貨記号

Globalization Support_LANGパラメータを次のように指定して、ユーザー・セッション用に言語ごとに異なるグローバリゼーション・サポート機能の操作を制御できます。

```
Globalization Support_LANG = <language>_<territory>.<character set>
```

*language*はユーザー・セッション用のGlobalization Support_LANGUAGEの値、*territory*はGlobalization Support_TERRITORYの値、*character set*は端末に使用されるコード体系を指定します。コード体系(通常は文字セットまたはコード・ページと呼ばれる)は、端末で表示可能な文字セットに対応する数値コードの範囲です。これには、端末との通信を制御するコードも含まれています。

Globalization Support_LANGは、環境変数(またはシステムでそれに相当するもの)として定義します。たとえば、Cシェルを使用するUNIXでは、Globalization Support_LANGを次のように定義できます。

```
setenv Globalization Support_LANG French_France.WE8ISO8859P1
```

セッション中にグローバリゼーション・サポート・パラメータを変更するには、ALTER SESSION文を次のように使用します。

```
ALTER SESSION SET <Globalization Support_parameter> = <value>
```

Oracleプリコンパイラでは、グローバリゼーション・サポート機能がすべてサポートされているため、アプリケーションではOracleデータベースに格納されている多言語のデータを処理できます。たとえば、外国語の文字変数を宣言し、それをINSTRB、LENGTHBおよびSUBSTRBなどの文字列ファンクションに渡すことができます。これらのファンクションにはそれぞれ、INSTR、LENGTHおよびSUBSTRファンクションと同じ構文がありますが、文字単位ではなく、バイト単位で機能します。

Globalization Support_INITCAP、Globalization Support_LOWERおよびGlobalization Support_UPPERファンクションを使用して、大文字小文字変換の特別なインスタンスを処理できます。また、Globalization Support_SORTファンクションを使用して、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。グローバリゼーション・サポート・パラメータをTO_CHAR, TO_DATEおよびTO_NUMBERファンクションに渡すこともできます。

3.10 グローバリゼーション・サポートのマルチバイト文字セット

Pro*COBOLプリコンパイラでは、次の機能により、マルチバイトのグローバリゼーション・サポート文字セットのサポートが拡張されています。

- プリコンパイラによる埋込みSQL文中のマルチバイト・キャラクタ文字列の認識。
- ANSI標準COBOLのPIC Nデータ型宣言句で、ホスト文字変数を2バイト文字の文字列として解釈するようにプリコンパイラに指示します。

Oracleでは、プリコンパイラのランタイム・ライブラリSQLLIBにより、マルチバイト文字列がサポートされています。

3.10.1 埋込みSQL内の文字列

埋込みSQL文内のマルチバイトのグローバリゼーション・サポート文字列は、文字列をマルチバイト文字列として識別する文字リテラルと、その後に続く一重引用符で囲まれた文字列で構成されます。

たとえば、次のような埋込みSQL文があるとします。

```
EXEC SQL
 SELECT empno INTO :emp_num FROM emp
 WHERE ename=N'Kuroda'
 END-EXEC.
```

この文では、'Kuroda'という文字列の前に付いているN文字リテラルにより、この文字列がマルチバイト文字列であると認識されるため、マルチバイト文字の文字列が含まれていることがわかります。

3.10.2 動的SQL

動的SQL文はプリコンパイル時には処理されず、Oracleではマルチバイトのグローバリゼーション・サポート文字列自体は処理されないため、動的SQL文にはグローバリゼーション・サポートのマルチバイト文字列を埋め込むことはできません。

3.10.3 埋込みDDL

グローバリゼーション・サポートのマルチバイト・データを格納する列は、埋込みのデータ定義言語(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。

3.10.4 グローバリゼーション・サポートのマルチバイト・ホスト変数

Pro*COBOLプリコンパイラでは、ANSI規格PIC N句を使用して、マルチバイト文字データのホスト変数を宣言します。PIC N句を使用して宣言された変数は、2バイト文字の文字列変数として認識されます。

- Globalization Support_LOCAL
- VARCHAR

これらのオプションの詳細は、Oracleプリコンパイラの実行を参照してください。

3.10.5 制限事項

表は使用できません。

PIC Nデータ型を使用して宣言されたホスト変数は、表に使用できません。

奇数バイト幅はありません。マルチバイトのグローバリゼーション・サポート文字の格納に、Oracle CHAR列を使用しないでください。奇数バイトのデータがシングルバイト列からマルチバイトのグローバリゼーション・サポート(PIC N)・ホスト変数にフェッチされると、ランタイム・エラーが発生します。

ホスト変数の同値化は行えません。マルチバイトのグローバリゼーション・サポート文字変数は、EXEC SQL VAR文を使用して同値化できません。

動的SQLは使用できません。Pro*COBOLでは、グローバリゼーション・サポートのマルチバイト文字に動的SQLを使用できません。

3.10.6 空白埋込み

Pro*COBOL文字変数をマルチバイトのグローバリゼーション・サポート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。「外部データ型」を参照してください。

CHARF。これは、マルチバイト文字列が定義されるときのデフォルトの文字型です。入力データから、後続のダブルバイトの空白文字が削除されます。ただし、文字列がダブルバイト空白文字のみで構成されている場合は、標識としてダブルバイトの空白

文字が1つバッファに残されます。

出力ホスト変数では、ダブルバイトの空白文字で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のダブルバイト空白文字が削除されません。length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みなされます。

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、グローバリゼーション・サポート・データはそのどちらにも対応していないためです。

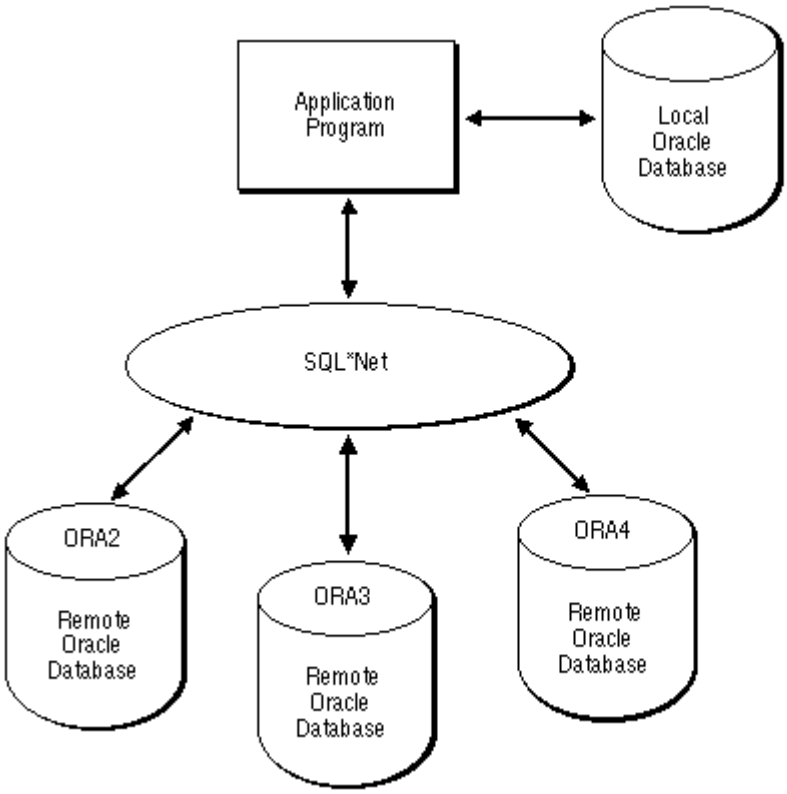
3.10.7 インジケータ変数

マルチバイトのグローバリゼーション・サポート文字変数でも、他の変数の場合と同様に、インディケータ変数を使用できますが、列の長さの値はバイト単位ではなく文字単位で表されます。使用可能な値の一覧は、インジケータ変数の使用についてを参照してください。

3.11 同時接続

Oracleプリコンパイラでは、SQL*Netを介して分散処理がサポートされます。アプリケーションは、ローカル・データベースおよびリモート・データベースの任意の組合せに同時にアクセスしたり、同じデータベースへの複数の接続を確立できます。図3-2では、アプリケーション・プログラムが1つのローカルOracleデータベースおよび3つのリモートOracleデータベースと通信しています。ORA2、ORA3およびORA4は、CONNECT文で使用される論理名です。

図3-2 SQL*Netを介した接続



SQL*Netは、異なるマシンやオペレーティング・システム間に存在するネットワーク上の境界を排除することにより、Oracleのツールに分散処理環境を提供します。この項では、SQL*Netを介した分散処理がOracleプリコンパイラでどのようにサポートされているかを説明します。さらに、アプリケーションで次の処理がどのように行われるかも学習します。

- 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
- 同じデータベースへの複数接続

3.11.1 予備知識

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

プロトコルは、ネットワークへのアクセスに関する一連の規則です。この規則では、障害発生後のリカバリ手順、データを送信およびエラー検査の形式などが規定されます。

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

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

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

3.11.2 デフォルトのデータベースおよび接続

各ノードにはデフォルトのデータベースがあります。CONNECT文でノードを指定し、データベースを指定しない場合、指定したローカルまたはリモート・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場合は、現在のノード上のデフォルト・データベースに接続されます。CONNECT文でデフォルト・データベースと現在のノードを指定することはできますが、必要ありません。

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

すべてのデータベース名は一意である必要があります。ただし、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、任意のノード上で、任意のデータベースに対する複数の接続を持てます。

3.11.3 明示的接続

通常、Oracleへの接続は次のように行います。

```
EXEC SQL CONNECT :userid IDENTIFIED BY :password
```

または、次の文でも接続できます。

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

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

次のようにして自動的に接続することもできます。データベースおよびノードを指定しない場合、現在のノード上のデフォルトのデータベースに接続されます。別のデータベースに接続する場合は、そのデータベースを明示的に指定する必要があります。

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

3.11.4 単一の明示的接続

次の例では、リモート・ノードにある1つの非デフォルト・データベースに接続します。

ノート:

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

```
-- Declare necessary host variables.
```

EXEC SQL BEGIN DECLARE SECTION; username CHARACTER(10); password CHARACTER(10); db_string CHARACTER(20); EXEC SQL END DECLARE SECTION; set username = 'scott'; set password = 'tiger'; set db_string = 'd:newyork-nondef'; -- Assign a unique name to the database connection. EXEC SQL DECLARE db_name DATABASE**;** -- Connect to the nondefault database EXEC SQL CONNECT :username IDENTIFIED BY :password AT db_name USING :db_string;

この例の識別子は、次の目的で使用されています。

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

USING句では、*db_name*に関連付けられるネットワーク、コンピュータおよびデータベースを指定します。その後、AT句(*db_name*付き)を使用したSQL文が、*db_string*に指定したデータベースで実行されます。

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

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_name CHARACTER(10);
 db_string CHARACTER(20);
EXEC SQL END DECLARE SECTION;
set username = 'scott';
set password = 'tiger';
set db_name = 'oracle1';
set db_string = 'd:newyork-nondef';
-- connect to the nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
...
```

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

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

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

次の例では、*db_name*はホスト変数です。

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

*db_name*がホスト変数の場合、SQL文で参照されるすべてのデータベース表を、DECLARE TABLE文で定義する必要があります。

カーソルの制御。OPEN, FETCHやCLOSEなどのカーソル制御文は例外で、AT句を使用しません。カーソルを明示的に指定したデータベースに関連付ける場合は、次のようにDECLARE CURSOR文でAT句を使用してください。

```
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;
```

*db_name*がホスト変数の場合、宣言されたカーソルを参照するすべてのSQL文の適用範囲内で宣言する必要があります。たとえば、あるサブプログラム内でカーソルをオープンし、別のサブプログラムでそのカーソルからフェッチする場合は、*db_name*をグローバルに宣言するか、それを各サブプログラムに渡す必要があります。

カーソルからのオープン、クローズまたはフェッチには、AT句は使用しません。SQL文は、DECLARE CURSOR文のAT句で名前を付けられたデータベースか、カーソルの宣言でAT句が使用されていない場合はデフォルトのデータベースにおいて実行されます。

AT :*host_variable*句を使用すると、カーソルに関連付けられた接続を変更できます。ただし、カーソルがオープンされているときは対応付けを変更できません。次の例を検討してください:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...
set db_name = 'oracle1';
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO ...
set db_name = 'oracle2';
EXEC SQL OPEN emp_cursor; -- illegal, cursor still open
EXEC SQL FETCH emp_cursor INTO ...
```

これは不適切です。2番目のOPEN文を実行しようとしても、 *emp_cursor* がまだオープン状態であるためです。異なる接続に対して別々のカーソルが維持されることはありません。*emp_cursor*は1つのみ存在できます。別の接続のために再オープンするには、その前にクローズする必要があります。最後の例をデバッグするには、次のように、単にカーソルをクローズしてから再オープンします。

```
EXEC SQL CLOSE emp_cursor; -- close cursor first
set db_name = 'oracle2';
EXEC SQL OPEN emp_cursor;
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、非デフォルトの接続で文を実行する場合は、AT句を使用する必要があります。次に例を示します。

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

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

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

次の例は方法3を示します。

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

複数の接続を同時にオープンする場合(アクティブな接続の識別にAT句が必要)でなければ、リモート・データベースに接続する際にAT句を使用する必要はありません。リモート・データベースにデフォルト接続を行うには、次の構文を使用します。

```
EXEC SQL CONNECT :username IDENTIFIED BY :password
 USING :db-string;
```

3.11.5 複数の明示的接続

単一の明示的接続の場合と同様に、複数の明示的接続にも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・データベースに同時に接続しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_string1 CHARACTER(20);
 db_string2 CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
set username = 'scott';
set password = 'tiger';
set db_string1 = 'New_York';
set db_string2 = 'Boston';
-- give each database connection a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;
-- connect to the two nondefault databases
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name2 USING :db_string2;
```

未宣言識別子*db_name1*および*db_name2*は、2つの非デフォルト・ノードにあるデフォルト・データベースの指名に使用され、これにより、SQL文では後からそれらのデータベースを名前で参照できます。

または、次の例のように、AT句でホスト変数を使用できます。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_name CHARACTER(10);
 db_string CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
set username = 'scott';
set password = 'tiger';
FOR EACH nondefault database
```

```
 -- get next database name and SQL*Net string
 display 'Database Name? ';
 read db_name;
 display 'SQL*Net String? ';
 read db_string;
 -- connect to the nondefault database
 EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
ENDFOR;
```

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

```
set username = 'scott';
set password = 'tiger';
set db_string = 'd:newyork-nondef';
FOR EACH nondefault database
 -- get next database name
 display 'Database Name? ';
 read db_name;
 -- connect to the nondefault database
 EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT :db_name USING :db_string;
ENDFOR;
```

複数の接続に同じSQL*Net文字列を使用する場合でも、接続ごとに異なるデータベース名を使用する必要があります。

3.11.6 暗黙的接続

暗黙的接続は、明示的接続の不要なOracleの分散データベース・データベース・オプションによりサポートされています。たとえば、分散問合せを使用すると、1つのSELECT文で1つ以上の非デフォルト・データベースにアクセスできます。

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

3.11.7 単一の暗黙的接続

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

```
EXEC SQL CREATE DATABASE LINK db_link
 CONNECT TO username IDENTIFIED BY password
 USING 'd:newyork-nondef';
```

これにより、プログラムでは、次のようにデータベース・リンクを使用して、非デフォルトのEMP表に対して問合せを実行できます。

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

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

例では、デフォルトのOracleサーバーは、データベース・リンク*db_link*を使用してSQL*Netで非デフォルトのデータベースに接続します。問合せはデフォルトのサーバーに送信されますが、非デフォルトのデータベースに転送されて実行されます。

データベース・リンクをさらに簡単に参照できるようにするには、次のようにシノニムを作成します(ここでも、通常は対話形式で行われます)。

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

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

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

これにより、*emp*の場所を意識する必要はなくなります。

3.11.8 複数の暗黙的接続

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

```
EXEC SQL CREATE DATABASE LINK db_link1
 CONNECT TO username1 IDENTIFIED BY password1
 USING 'd:newyork-nondef';
EXEC SQL CREATE DATABASE LINK db_link2
 CONNECT TO username2 IDENTIFIED BY password2
 USING 'd:chicago-nondef';
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1;
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

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

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

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

3.12 Oracle Call Interface(OCI)コールの埋込み

Oracleプリコンパイラを使用すれば、ホスト・プログラムにOCIコールを埋め込むことができます。それには次のステップを実行します。

1. OCIログイン・データ領域(LDA)を宣言部の外で宣言します。
2. OCIコールOLOGではなく、埋込みSQL文CONNECTを使用してOracleに接続します。
3. Oracleランタイム・ライブラリ・ルーチンSQLLDAをコールして、LDAに接続情報を格納します。

こうして、Oracleプリコンパイラと OCIでは、両者が連動していることが認識されます。ただし、Oracleカーソルは共有されません。

Oracleランタイム・ライブラリにより接続が管理され、HDAがメンテナンスされるため、OCIホスト・データ領域(HDA)の宣言を意識する必要はありません。

3.12.1 LDAの設定について

OCIコールを発行してLDAを設定します。

```
SQLLDA(lda);
```

*lda*では、LDAデータ構造体を指定します。このコールの書式は、言語によって異なります。CONNECT文が失敗した場合、*lda*

内の*lda_rc*フィールドはエラーを示す1012に設定されます。

3.12.2 リモートの複数接続

SQLLDAに対するコールにより、最後に実行されたSQL文で使用した接続のLDAが設定されます。追加の接続に必要な別のLDAを設定するには、各CONNECTの後に別の*lda*を指定してSQLLDAをコールしてください。次の例では、2つの非デフォルト・データベースに同時に接続します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(10);
 password CHARACTER(10);
 db_string1 CHARACTER(20);
 db_string2 CHARACTER(20);
EXEC SQL END DECLARE SECTION;
lda1 INTEGER(32);
lda2 INTEGER(32);
set username = 'SCOTT';
set password = 'TIGER';
set db_string1 = 'D:NEWYORK-NONDEF1';
set db_string2 = 'D:CHICAGO-NONDEF2';
-- give each database connection a unique name
EXEC SQL DECLARE db_name1 DATABASE;
EXEC SQL DECLARE db_name2 DATABASE;
-- connect to first nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name1 USING :db_string1;
-- set up first LDA for OCI use
SQLLDA(lda1);
-- connect to second nondefault database
EXEC SQL CONNECT :username IDENTIFIED BY :password
 AT db_name2 USING :db_string2;
-- set up second LDA for OCI use
SQLLDA(lda2);
```

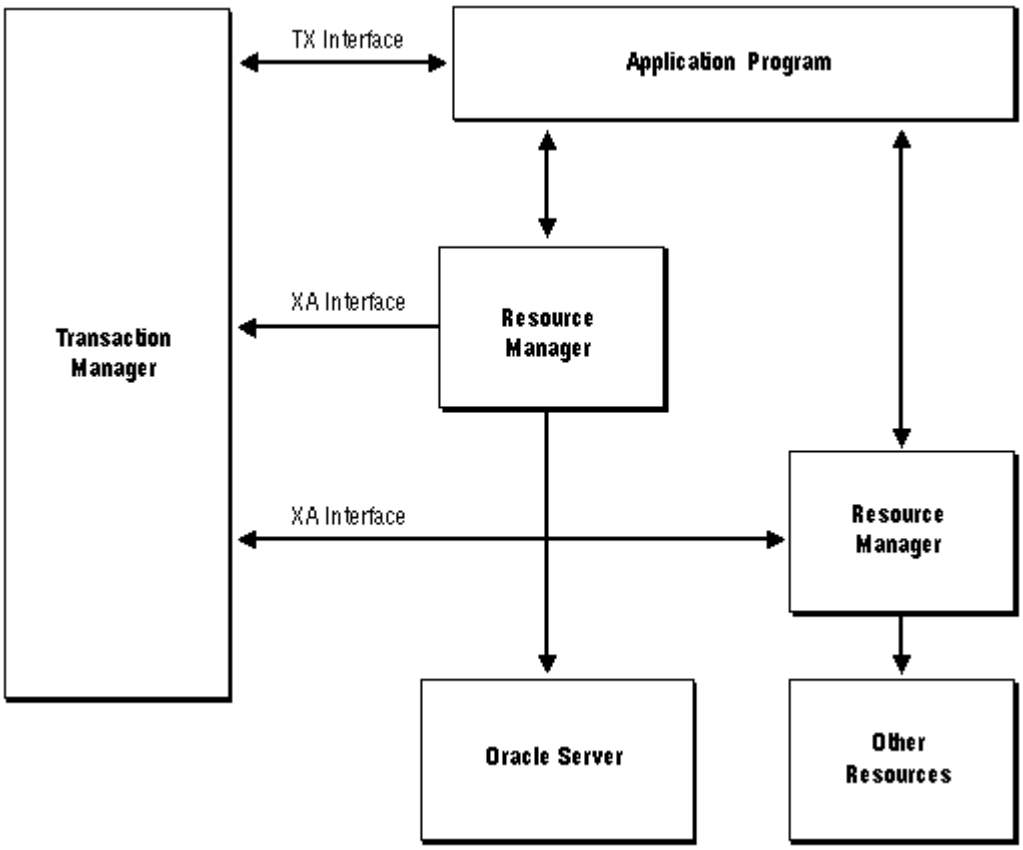
*db_name1*および*db_name2*は、ホスト変数ではないので、宣言部で宣言しないでください。これらは、後からSQL文で名前によりデータベースを参照できるように、2つの非デフォルト・ノードにあるデフォルト・データベースを指定する場合にのみ使用します。

3.13 X/Openアプリケーションの開発について

X/Openアプリケーションは、分散トランザクション処理(DTP)環境で動作します。抽象モデルでは、X/Openアプリケーションはリソース・マネージャ(RM)に各種サービスの提供を要求します。たとえば、データベース・リソース・マネージャは、データベース内のデータにアクセスします。リソース・マネージャは、アプリケーションのすべてのトランザクションを制御するトランザクション・マネージャ(TM)と対話します。

図3-3では、DTPモデルのコンポーネントで、Oracleデータベース内のデータに効率的にアクセスするために対話できる方法を示しています。このDTPモデルでは、リソース・マネージャとトランザクション・マネージャの間に*XA*インタフェースが指定されています。Oracleでは、XA準拠のライブラリが提供され、このライブラリは、X/Openアプリケーションにリンクさせる必要があります。また、アプリケーション・プログラムとリソース・マネージャ間でネイティブ・インタフェースを指定する必要もあります。

図3-3 DTPの仮想モデル



トランザクション・マネージャとリソース・マネージャがアプリケーション・プログラムと対話する方法を指定するこのDTPモデルについては、X/Openガイド『*Distributed Transaction Processing Reference Model*』および関連出版物にで説明されていますが、これらは次の宛先に書面で請求すれば入手できます。

X/Open Company Ltd.1010 El Camino Real, Suite 380Menlo Park, CA 94025

XAインタフェースの使用方法は、ご使用のトランザクション処理(TP)モニターのユーザー・ガイドを参照してください。

3.13.1 Oracle固有の問題

Oracleプリコンパイラを使用して、X/Open規格に準拠したアプリケーションを開発できます。ただし、次の要件を満たす必要があります。

3.13.2 Oracleへの接続について

X/Openアプリケーションでは、データベースへの接続の確立およびメンテナンスは行われません。かわりに、Oracleにより提供されるトランザクション・マネージャとXAインタフェースにより、データベースの接続および切断が透過的に処理されます。したがって、通常、X/Open準拠のアプリケーションでは、CONNECT文は実行されません。

3.13.3 トランザクション制御

X/Openアプリケーションでは、グローバル・トランザクションに影響を与えるCOMMIT、ROLLBACK、SAVEPOINTおよびSET TRANSACTIONなどの文を実行しないでください。たとえば、コミットはトランザクション・マネージャで処理されるため、アプリケーションではCOMMIT文を実行しないでください。また、CREATE、ALTERおよびRENAMEなどのSQLデータ定義文では暗黙的なコミットが発行されるため、アプリケーションでこれらの文を実行しないでください。

アプリケーションでは、さらなるSQL操作を妨げるエラーが検出された場合、内部のROLLBACK文を実行できます。ただし、今後リリースされるXAインタフェースでは、変更される可能性もあります。

3.13.4 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を使用する必要があります。このルーチンにより、XAインタフェースを通じて確立された指定の接続のために、LDAが設定されます。OCOM、OCON、OCOF、ORLON、OLON、OLOGおよびOLOGOFは、X/Openアプリケーションからは発行できません。

3.13.5 リンク

XA機能を利用するには、XAライブラリをX/Openアプリケーション・オブジェクト・モジュールにリンクさせる必要があります。手順は、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。

4 埋込みSQLの使用方法

この章は、次の項で構成されています。

- ホスト変数の使用方法について
- インジケータ変数の使用について
- 基本のSQL文
- カーソル
- カーソル変数

この章は、埋込みSQLプログラミングの基本技術を理解し、利用する上で役立ちます。ホスト変数、インジケータ変数、カーソル、カーソル変数、およびOracleデータの挿入、更新、選択、削除を行う基本的なSQLコマンドの使用方法について学習します。

4.1 ホスト変数の使用方法について

Oracleでは、ホスト変数を使用してプログラムにデータやステータス情報を渡します。一方、プログラムでは、ホスト変数を使用してOracleにデータを渡します。

4.1.1 出力変数と入力変数

その使用方法によって、ホスト変数は出力変数または入力変数と呼ばれます。SELECT文またはFETCH文のINTO句で使用されるホスト変数は、Oracleからの列値の出力を保持するので、出力ホスト変数と呼ばれます。Oracleでは、列値をINTO句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、入力ホスト変数は、INSERT文のVALUES句とUPDATE文のSET句で使用します。また、WHERE句、HAVING句およびFOR句でも使用します。実際に、値または式が使用可能であれば、1つのSQL文で複数の入力ホスト変数を使用できます。

ORDER BY句では、ホスト変数を使用使用できますが、定数またはリテラルとして扱われるため、ホスト変数のコンテンツには効果がありません。たとえば、次のようなSQL文があるとします。

```
EXEC SQL SELECT ename, empno INTO :name, :number
 FROM emp
 ORDER BY :ord;
```

ここでは、入力ホスト変数*ord*が含まれているように見えます。しかし、この句のホスト変数は、定数として扱われ、*ord*の値にかかわらず、順序付けは行われません。

入力ホスト変数を使用して、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ことはできません。したがって、ALTER、CREATEおよびDROPなどのデータ定義文(*DDL*とも呼ばれる)では、入力ホスト変数を使用できません。次の例では、DROP TABLE文は無効です。

```
EXEC SQL BEGIN DECLARE SECTION;
 table_name CHARACTER(30);
EXEC SQL END DECLARE SECTION;
display 'Table name? ';
read table_name;
EXEC SQL DROP TABLE :table_name; -- 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

を検討してください:

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name CHARACTER(20);
EXEC SQL END DECLARE SECTION;
-- get values for input host variables
display 'Employee number? ';
read emp_number;
display 'Employee name? ';
read emp_name;
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
 VALUES (:emp_number, :emp_name);
```

INSERT文のVALUES句にある入力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。

4.2 インジケータ変数の使用について

任意のホスト変数をオプションのインジケータ変数に関連付けることができます。標識変数に関連付けたホスト変数をSQL文内で使用するたびに、結果コードが対応する標識変数内に格納されます。したがって、インジケータ変数により、ホスト変数を監視できます。

インジケータ変数は、VALUES句またはSET句ではNULLを入力ホスト変数に割り当てるために、INTO句では出力ホスト変数で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。

4.2.1 入力変数

入力ホスト変数の場合、プログラムがインディケータ変数に割り当てる値の意味は次のとおりです。

- -1: Oracleでは、ホスト変数の値を無視して、NULLを列に割り当てます。
- >= 0: Oracleでは、ホスト変数の値を列に割り当てます。

4.2.2 出力変数

出力ホスト変数の場合、Oracleでインジケータ変数に割り当てられる値には、次の意味があります。

- -2: Oracleでは切り捨てられた列値をホスト変数に割り当てましたが、数値が大きすぎるため、元の長さの列値をインジケータ変数に割り当てることができませんでした。
- -1: 列値はNULLであるため、ホスト変数の値は不確定です。
- 0: Oracleでは列値をそのままの形でホスト変数に割り当てました。
- > 0: Oracleでは切り捨てられた列値をホスト変数に割り当て、元の列の長さ(マルチバイトのグローバリゼーション・サポート・ホスト変数の場合、バイト単位ではなく文字単位で表される)をインジケータ変数に割り当て、SQLCAのSQLCODEをゼロに設定しました。

インジケータ変数は、宣言部では2バイト整数として定義する必要があり、SQL文では(キーワードINDICATORを使用しない場合)前にコロンを付け、関連付けられたホスト変数に追加する必要があります。

4.2.3 NULLの挿入

インジケータ変数を使用して、NULLを挿入できます。挿入の前に、NULLにする列ごとに、該当するインジケータ変数を次の例に示すように-1に設定します。

```
set ind_comm = -1;
```

```
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
 VALUES (:emp_number, :commission:ind_comm);
```

インジケータ変数*ind_comm*は、NULLがCOMM列に格納されるように指定します。

かわりに、次のようにNULLをハードコードすることもできます。

```
EXEC SQL INSERT INTO EMP (EMPNO, COMM)
 VALUES (:emp_number, NULL);
```

これは柔軟性が少なくなりますが、読みやすくなります。

通常、NULLは、次の例のように条件付きで挿入します。

```
display 'Enter employee number or 0 if not available: ';
read emp_number;
IF emp_number = 0 THEN
 set ind_empnum = -1; ELSE
 set ind_empnum = 0;
ENDIF;
EXEC SQL INSERT INTO EMP (EMPNO, SAL)
 VALUES (:emp_number:ind_empnum, :salary);
```

4.2.4 戻されたNULLの処理

インジケータ変数を使用して、次の例のように、戻されたNULLを操作できます。

```
EXEC SQL SELECT ENAME, SAL, COMM
 INTO :emp_name, :salary, :commission:ind_comm
 FROM EMP
 WHERE EMPNO = :emp_number;
IF ind_comm = -1 THEN
 set pay = salary; -- commission is null; ignore it
ELSE
 set pay = salary + commission;
ENDIF;
```

4.2.5 NULLのフェッチ

ただし、DBMSがNATIVE、V7またはV8の場合、インジケータ変数のないホスト変数にNULLを選択またはフェッチにより入れると、次のエラー・メッセージが表示されます。

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

4.2.6 NULLの検査

WHERE句のインジケータ変数を使用して、次のようにNULLがないか検査できます。

```
EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary
 FROM EMP
 WHERE :commission:ind_comm IS NULL ...
```

ただし、NULLを互いに、あるいは他の値と比較するために比較演算子を使用することはできません。たとえば、次のSELECT文は、COMM列に1つ以上のNULLが含まれていれば失敗します。

```
EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary
```

```
 FROM EMP
 WHERE COMM = :commission:ind_comm;
```

次の例は、値のいくつかがNULLの可能性がある場合に、値を比較して等しいかどうかを調べる方法を示しています。

```
EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary
 FROM EMP
 WHERE (COMM = :commission) OR ((COMM IS NULL) AND
 (:commission:ind_comm IS NULL));
```

4.2.7 切り捨てられた値のフェッチ

切り捨てられた列値を、SELECT文またはFETCH文でインジケータ変数のないホスト変数に入れても、エラーは発生しません。

4.3 基本的なSQL文

実行可能なSQL文を使用すれば、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オブジェクトの作成、定義およびメンテナンスができます。この章では、データ操作文(*DML*)とカーソル制御文を中心に取り上げます。次のSQL文により、Oracleデータの問合せと操作ができます。

- SELECT: 1つ以上の表から行を戻します。
- INSERT: 新しい行を表に追加します。
- UPDATE: 表内の行を変更します。
- DELETE: 表から行を削除します。

INSERT、UPDATEまたはDELETEなどのデータ操作文を実行するときは、入力ホスト変数の値の設定以外に、文が成功するか失敗するかのみを考えます。これは、SQLCAを調べればわかります。(SQL文を実行すると、SQLCA変数が設定されます。)次の2通りの方法で調べることができます。

- WHENEVER文による暗黙的チェック
- SQLCA変数の明示的なチェック

または、MODE={ANSI|ANSI14}の場合、状態変数SQLSTATEまたはSQLCODEをチェックできます。詳細は、MODE={ANSI|ANSI14}の場合の状態変数の使用についてを参照してください。

ただし、SELECT文(問合せ)を実行する場合は、戻されるデータの行も処理する必要があります。問合せは次のように分類できます。

- 行を戻さない問合せ(有無のみを調べる)
- 1行のみ戻す問合せ
- 複数行を戻す問合せ

複数の行を戻す問合せには、明示的に宣言されたカーソルまたはカーソル変数(または、ホスト配列の使用方法で説明されているホスト配列の使用)が必要です。明示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: カーソルを指定し、問合せと関連付けます。
- OPEN: 問合せを実行し、アクティブ・セットを識別します。
- FETCH: カーソルを進め、アクティブ・セット内の行を1行ずつ取得します。
- CLOSE: カーソルを無効にします(アクティブ・セットは未定義になります)。

後続の項では、まずINSERT、UPDATE、DELETEと1行のSELECT文のコードの方法を学習します。その後、複数行のSELECT文に進みます。

4.3.1 行の選択

データベースに対する問合せは、一般的なSQL操作です。問合せを発行するには、SELECT文を使用します。次の例では、EMP表に問合せを行います。

```
EXEC SQL SELECT ENAME, JOB, SAL + 2000
 INTO :emp_name, :job_title, :salary
 FROM EMP
 WHERE EMPNO = :emp_number;
```

キーワードSELECTの後の列名と式により、選択リストが作成されます。例の選択リストには、3つの項目が含まれています。WHERE句(および存在する場合はそれに続く句)で指定された条件のもと、OracleではINTO句のホスト変数に列値を戻します。選択リスト内の項目数は、INTO句内のホスト変数と同数であり、すべての戻り値を格納する場所があります。

最も簡単な場合、1つの問合せで1行が戻されると、その形式は前述の例のようになります(EMPNOは一意キーです)。ただし、1つの問合せで複数行を戻すことができる場合、カーソルを使用してそれらの行をフェッチするか、行を選択してホスト配列に入れる必要があります。

1行のみを戻す問合せを作成したにもかかわらず、実際には複数行が戻される場合、その結果は、SELECT_ERRORオプションをどのように指定するかによって異なります。SELECT_ERROR=YES(デフォルト)の場合、複数行が戻されると、次のメッセージが表示されます。

```
ORA-01422: exact fetch returns more than requested number of rows
```

SELECT_ERROR=NOの場合、1行が戻され、エラーは発生しません。

4.3.2 使用可能な句

SELECT文では、次の標準的なSQL句をすべて使用できます。INTO、FROM、WHERE、CONNECT BY、START WITH、GROUP BY、HAVING、ORDER BYおよびFOR UPDATE OFです。

4.3.3 行の挿入

表またはビューに行を追加するには、INSERT文を使用します。次の例では、EMP表に1行追加します。

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, SAL, DEPTNO)
 VALUES (:emp_number, :emp_name, :salary, :dept_number);
```

列リストで指定する各列は、INTO句で指定した表のものである必要があります。VALUES句では、挿入される値の行を指定しています。値には、定数、ホスト変数、SQL式または擬似列(USERやSYSDATEなど)を指定できます。

VALUES句の値の数は、列リストにある名前の数と同じにする必要があります。ただし、VALUES句にCREATE TABLEで定義した表の各列の値が定義どおりの順序で含まれている場合は、列リストを省略してもかまいません。

4.3.4 副問合せの使用方法

副問合せは、ネストされたSELECT文です。副問合せを使用すれば、マルチパート検索を実行できます。これらは、次の目的で使用できます。

- SELECT、UPDATEおよびDELETE文のWHERE、HAVINGおよびSTART WITH句での比較の値を指定
- CREATE TABLEまたはINSERT文で挿入する行のセットを定義

- UPDATE文のSET句の値を定義

たとえば、ある表から別の表に行をコピーするには、INSERT文のVALUES句を次のように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;
```

中間結果を取得するために、INSERT文で副問合せがどのように使用されているかに注意してください。

4.3.5 行の更新

表またはビューの指定した列の値を変更するには、UPDATE文を使用します。次の例では、EMP表のSAL列とCOMM列を更新します。

```
EXEC SQL UPDATE EMP
 SET SAL = :salary, COMM = :commission
 WHERE EMPNO = :emp_number;
```

オプションのWHERE句を使用すれば、行が更新される条件を指定できます。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つ以上の行の名前をリストします。副問合せを使用すれば、次の例のように値を指定できます。

```
EXEC SQL UPDATE EMP
 SET SAL = (SELECT AVG(SAL)*1.1 FROM EMP WHERE DEPTNO = 20)
 WHERE EMPNO = :emp_number;
```

4.3.6 行の削除

表またはビューから行を削除するには、DELETE文を使用します。次の例では、EMP表から指定した部門の全従業員を削除します。

```
EXEC SQL DELETE FROM EMP
 WHERE DEPTNO = :dept_number;
```

オプションのWHERE句を使用すれば、行が削除される条件を指定できます。

4.3.7 WHERE句の使用

表またはビューで、検索条件を満たす行のみを選択、更新または削除するには、WHERE句を使用します。WHERE句の検索条件はブール式で、スカラー・ホスト変数、ホスト配列(SELECT文では不可)および副問合せを含めることができます。

WHEREを省略すると、表またはビュー内の行がすべて処理されます。UPDATEまたはDELETE文でWHERE句を省略すると、OracleではSQLCAのSQLWARN(5)を'W'に設定し、すべての行が処理されたことを警告します。

4.4 カーソル

1つの問合せで複数の行が戻される場合、カーソルを明示的に定義すれば、次のことができます。

- 問合せによって戻された最初の行以後の処理
- 現在どの行が処理されているかの追跡および記録

カーソルは、問合せによって戻された行の集合内で現在の行はどれかを示します。これによって、プログラムは一度に1行ずつ処理できます。次の文を使用して、カーソルの定義および操作を行います。

- DECLARE
- OPEN
- FETCH
- CLOSE

まず、DECLARE文を使用して、カーソルに名前を付け、問合せに関連付けます。

OPEN文により問合せが実行され、問合せの検索条件を満たす行がすべて識別されます。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。このカーソルをオープンした後、これを使用して、関連付けられた問合せによって戻された行を取得できます。

アクティブ・セットの行は1行ずつ取り出されます(ホスト配列を使用していない場合)。アクティブ・セット内の現在の行を取得するには、FETCH文を使用します。FETCH文は、すべての行が取得されるまで繰り返し実行できます。

アクティブ・セットからの行の取得が完了したら、CLOSE文でこのカーソルを無効にします。アクティブ・セットは未定義になります。

4.4.1 カーソルの宣言

次の例のように、DECLARE文を使用して、カーソルに名前を付け、問合せに関連付けることで、カーソルを定義できます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, EMPNO, SAL
 FROM EMP
 WHERE DEPTNO = :dept_number;
```

カーソル名は、ホスト変数やプログラム変数ではなく、プリコンパイラで使用される識別子であるため、宣言部では定義しないでください。したがって、あるプリコンパイル単位から別のプリコンパイル・ユニットにカーソル名を渡すことはできません。また、カーソル名には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字以内にしてください。

カーソルに関連付けられたSELECT文には、INTO句を指定できません。INTO句と出力ホスト変数のリストは、FETCH文で指定します。

DECLARE文は宣言文のため、カーソルを参照する他のすべてのSQL文より(論理的にではなく)物理的に前に配置する必要があります。つまり、カーソルの前の参照はできません。次の例では、OPEN文の位置が間違っています。

```
EXEC SQL OPEN emp_cursor; -- misplaced OPEN statement
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, EMPNO, SAL
 FROM EMP
 WHERE ENAME = :emp_name;
```

カーソル制御文はすべて、同じプリコンパイル・ユニット内で指定する必要があります。たとえば、Aファイルの中でカーソルを宣言して、それをBファイルでオープンすることはできません。

ホスト・プログラムでは、必要な数のカーソルを宣言できます。ただし、1つのファイルでは、すべてのDECLARE文は一意である必要があります。つまり、カーソルの適用範囲は1つのファイル全体に及ぶため、1つのプリコンパイル・ユニット内には、たとえ別のブロックやプロシージャ内であっても、同じ名前のカーソルを2つ宣言することはできません。多数のカーソルを使用する場合は、MAXOPENCURSORSオプションの指定が必要になることがあります。詳細は、MAXOPENCURSORSを参照してください。

4.4.2 カーソルのオープン

問合せを実行して、アクティブ・セットを識別するには、OPEN文を使用します。次の例では、*emp_cursor*という名前のカーソルをオープンします。

```
EXEC SQL OPEN emp_cursor;
```

OPENにより、カーソルはアクティブ・セットの最初の行の直前に位置付けられます。また、SQLCA内のSQLERRD(3)に保存されている処理済の行数も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。それは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する必要があります。

通常、カーソルはクローズしてから再オープンする必要があります。ただし、MODE=ORACLEを指定する場合(デフォルト)、カーソルをクローズしてから再オープン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する可能性があります。詳細は、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およびMAXOPENCURSORSの3つのプリコンパイラ・オプションの値によって決まります。詳細は、プリコンパイラ・オプションの使用についてを参照してください。

4.4.3 カーソルからのフェッチ

アクティブ・セットから行を取得して、結果を格納する出力ホスト変数を指定するには、FETCH文を使用します。カーソルに関連付けられたSELECT文には、INTO句を指定できないことを思い出してください。INTO句と出力ホスト変数のリストは、FETCH文で指定します。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 emp_cursor
 INTO :emp_name, :emp_number, :salary;
```

カーソルは、あらかじめ宣言し、オープンしておく必要があります。最初にFETCH文を実行すると、カーソルは、アクティブ・セットの最初の行の前から最初の行に移動します。この行がカレント行になります。その後FETCH文を実行するたびに、カーソルはアクティブ・セット内の次の行に進み、現在の行が変わっていきます。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、カーソルに関連付けられた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。MODE={ANSI | ANSI14 | ANSI13}に設定されている場合、カーソルを一度クローズしてから、再オープン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用して同じカーソルからフェッチできます。ただし、各FETCH文のINTO句の対応するホスト変数は、同じデータ型であることが必要です。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE DEPTNO = 20;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name1, :salary1;
 EXEC SQL FETCH emp_cursor INTO :emp_name2, :salary2;
 EXEC SQL FETCH emp_cursor INTO :emp_name3, :salary3;
 ...
ENDLOOP;
```

アクティブ・セットが空か、それ以上の列がない場合、FETCHは、データが見つからないことを示すOracle警告コードをSQLCAのSQLCODE (または、MODE=ANSIの場合は状態変数SQLSTATE)に戻します。出力ホスト変数のステータスは不確定です。(通常のプログラムでは、WHENEVER NOT FOUND文がこのエラーを検出します。)カーソルを再利用するには、再オープンする必要があります。

4.4.4 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、カーソルのオープンによって確保していたリソース(記憶域など)を解放します。カーソルがクローズされると、解析ロックが解放されます。どのリソースが解放されるかは、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、emp_cursorという名前のカーソルをクローズします。

```
EXEC SQL CLOSE emp_cursor;
```

クローズしたカーソルのアクティブ・セットは未定義になるため、クローズしたカーソルから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。

MODE={ANSI13|ORACLE}の場合、コミットまたはロールバックを発行すると、CURRENT OF句で参照されるカーソルがクローズされます。他のカーソルには、コミットまたはロールバックによる影響はなく、オープンの場合は、オープンのままです。ただし、MODE={ANSI|ANSI14}の場合は、コミットまたはロールバックを発行すると、すべての明示カーソルがクローズされます。

4.4.5 CURRENT OF句の使用方法

指定したカーソルからフェッチした最後の行を参照するには、DELETEまたはUPDATE文でCURRENT OF *cursor_name*句を使用します。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。フェッチが行われていない場合や、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またはDELETE文のCURRENT OF句で参照されるカーソルを宣言するとき、FOR UPDATE OF句をオプションとして指定できます。CURRENT OF句は、必要に応じてFOR UPDATE句を追加するようにプリコンパイラに指示します。詳細は、FOR UPDATE OF句の使用についてを参照してください。

次の例では、CURRENT OF句を使用して、*emp_cursor*という名前のカーソルから最後にフェッチされた行を参照します。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE JOB = 'CLERK'
 FOR UPDATE OF SAL;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
 EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
ENDLOOP;
```

4.4.6 制限事項

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。コミット後にFOR UPDATEカーソルからフェッチしようとすると、次のエラーが発生します。

```
ORA-01002: fetch out of sequence
```

CURRENT OF句ではホスト配列を使用できません。他の方法は、CURRENT OF句の擬似実行についてを参照してください。また、関連付けられた1つのFOR UPDATE OF句で複数の表を参照することもできず、つまり、CURRENT OF句との結合ができないということです。さらに、動的SQLではCURRENT OF句を使用できません。

4.4.7 一般的な文の順序

次の例は、アプリケーション・プログラムでのカーソル制御文の一般的な順序を示しています。

```
-- Define a cursor.
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB FROM EMP
 WHERE EMPNO = :emp_number
 FOR UPDATE OF JOB;

-- Open the cursor and identify the active set.
EXEC SQL OPEN emp_cursor;
-- Exit if the last row was already fetched.
EXEC SQL WHENEVER NOT FOUND DO no_more;

-- Fetch and process data in a loop.
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;
 -- host-language statements that operate on the fetched data
 EXEC SQL UPDATE EMP
 SET JOB = :new_job_title
 WHERE CURRENT OF emp_cursor;
ENDLOOP;
...
ROUTINE no_more
BEGIN
-- Disable the cursor.
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 exit program;
END no_more;
```

4.4.8 完全な例

次のプログラムは、カーソルとFETCH文の使用方法を説明するためのものです。プログラムでは、部門番号の入力を要求してから、その部門の全従業員の名前を表示します。

最後の1つのフェッチを除くすべてのフェッチで1行ずつ戻され、フェッチ中にエラーが検出されなければ、成功のステータス・コードが戻されます。最後のフェッチは失敗し、「データが見つかりません」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積数は、SQLCAのSQLERRD(3)に示されています。

```
-- declare host variables
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_name CHARACTER(10);
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
-- copy in the SQL Communications Area
EXEC SQL INCLUDE SQLCA;

display 'Username? ';
read username;
display 'Password? ';
read password;

-- handle processing errors
EXEC SQL WHENEVER SQLERROR DO sql_error;
```

```
-- log on to Oracle
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';

-- declare a cursor
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME FROM EMP WHERE DEPTNO = :dept_number;

display 'Department number? ';
read dept_number;

-- open the cursor and identify the active set
EXEC SQL OPEN emp_cursor;

-- exit if the last row was already fetched
EXEC SQL WHENEVER NOT FOUND DO no_more;

display 'Employee Name';
display '-------------';

-- fetch and process data in a loop
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name; display emp_name;
ENDLOOP;
ROUTINE no_more
BEGIN
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 display 'End of program';
 exit program;
END no_more;

ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

4.5 カーソル変数

この項では、カーソル変数の概要を簡単に説明します。詳細は、ホスト言語の補足資料および「カーソル変数」を参照してください。

Pro*COBOLおよびPro*FORTRANプリコンパイラで静的埋込みSQLを使用すると、カーソル変数を宣言できます。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・セットの中のカレント行を指します。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。カーソルは静的で、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。

また、カーソル変数に新しい値を割り当て、サブプログラム(Oracle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せます。これにより、データ検索を簡単に集中化できます。

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後、次の4つの文を使用してカーソル変数を制御します。

- ALLOCATE
- OPEN ... FOR

- FETCH
- CLOSE

カーソル変数を宣言して、メモリーを割り当てたら、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。サーバー側でOPEN、FORを使用して複数行の問合せのためにカーソル変数をオープンし、クライアント側ではその変数からFETCHを行い、サーバー側かクライアント側のいずれかでCLOSEします。

4.5.1 カーソル変数の宣言について

カーソル変数の宣言方法は、使用ホスト言語により異なります。カーソル変数の宣言に関する指示は、使用ホスト言語の補足資料を参照してください。

4.5.2 カーソル変数の割当て

カーソル変数にメモリーを割り当てるには、ALLOCATE文を使用します。構文は次のとおりです。

```
EXEC SQL ALLOCATE <cursor_variable>;
```

4.5.3 カーソル変数のオープン

OPEN ... FOR文を使用して、カーソル変数を複数行問合せに関連付け、問合せを実行し、アクティブ・セットを特定します。構文は次のとおりです。

```
EXEC SQL OPEN <cursor_variable> FOR <select_statement>;
```

SELECT文では、入力ホスト変数およびPL/SQL変数、パラメータ、ファンクションを参照できますが、FOR UPDATEには使用できません。次の例では、*emp_cv*という名前のカーソル変数をオープンします。

```
EXEC SQL OPEN emp_cv FOR SELECT * FROM EMP;
```

カーソル変数はサーバー側でオープンする必要があります。カーソル変数を入力ホスト変数として無名のPL/SQLブロックに渡すことでオープンします。実行時に、そのブロックはOracleサーバーに送られ、実行されます。次の例では、カーソル変数を宣言し、初期化してから、PL/SQLブロックに渡し、そこでカーソル変数がオープンされます。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
-- declare cursor variable
 emp_cur SQL_CURSOR;
EXEC SQL END DECLARE SECTION;

-- initialize cursor variable
EXEC SQL ALLOCATE :emp_cur;

EXEC SQL EXECUTE
 -- pass cursor variable to PL/SQL block
 BEGIN
 -- open cursor variable
 OPEN :emp_cur FOR SELECT * FROM EMP;
 END;
END-EXEC;
```

一般に、カーソル変数を仮パラメータの1つとして宣言するストアド・プロシージャをコールすることで、カーソル変数をPL/SQLに渡します。たとえば、次のパッケージ・プロシージャは、*emp_cv*という名前のカーソル変数をオープンします。

```
CREATE PACKAGE emp_data AS
 -- define REF CURSOR type
```

```
 TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
 -- declare formal paramter of that type
 PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp);
END emp_data;

CREATE PACKAGE BODY emp_data AS
 PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
 BEGIN
 -- open cursor variable
 OPEN emp_cv FOR SELECT * FROM emp;
 END open_emp_cv;
END emp_data;
```

このパッケージ・プロシージャは、次のように、どのアプリケーションからでもコールできます。

```
EXEC SQL EXECUTE
 BEGIN
 emp_data.open_emp_cv(:emp_cur);
 END;
END-EXEC;
```

4.5.4 カーソル変数からのフェッチ

複数行問合せのためにカーソル変数をオープンした後、FETCH文を使用して、アクティブ・セットから行を1行ずつ取得します。構文は次のとおりです。

```
EXEC SQL FETCH cursor_variable_name
 INTO {record_name | variable_name[, variable_name, ...]};
```

カーソル変数によって戻される各列値は、データ型に互換性がある場合には、INTO句で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、行をフェッチして、*emp_rec*という名前のホスト・レコードに入れます。

```
-- exit loop when done fetching
EXEC SQL WHENEVER NOT FOUND DO no_more;
LOOP
 -- fetch row into record
 EXEC SQL FETCH :emp_cur INTO :emp_rec;
 -- process the data
ENDLOOP;
```

4.5.5 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、CLOSE文を使用し、この時点で、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;
```

CLOSE文は、クライアント側またはサーバー側で実行できます。次の例では、最後の行が処理されてから、カーソル変数*emp_cur*をクローズします。

```
-- close cursor variable
EXEC SQL CLOSE :emp_cur;
```

5 埋込みPL/SQLの使用方法

この章の構成は、次のとおりです。

- PL/SQLの利点
- PL/SQLブロックの埋込みについて
- ホスト変数の使用方法について
- インジケータ変数の使用について
- ホスト配列の使用について
- カーソルの使用方法について
- ストアド・サブプログラム
- 動的PL/SQLの使用について

PL/SQLトランザクション処理ブロックをプログラム内に埋め込むことにより、パフォーマンスを改善する方法を説明します。

5.1 PL/SQLの利点

この項では、PL/SQLによって提供される次のような機能および利点について説明します。

- パフォーマンスの向上
- Oracleとの統合
- カーソルFORループ
- サブプログラム
- パラメータ・モード
- パッケージ
- PL/SQL表
- ユーザー定義のレコード

5.1.1 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および生産性の向上が図れます。たとえば、PL/SQLを使用しないと、Oracleは一度に1つずつSQL文を処理する必要があります。SQL文ごとにサーバーに対する別のコールが発生し、その結果、オーバーヘッドが増加します。しかし、PL/SQLを使用すると、SQL文のブロック全体をサーバーに送信できます。これにより、アプリケーションとOracle間の通信は最小限に抑えられます。

5.1.2 Oracleとの統合

PL/SQLは、Oracleサーバーと密接に統合されています。たとえば、PL/SQLデータ型の大部分は、Oracleデータ・ディクショナリにとっても固有のデータ型です。さらに、次の例に示すように、データ・ディクショナリに格納された列定義に基づいて変数を宣言するための%TYPE属性を指定できます。

```
job_title emp.job%TYPE;
```

したがって、列の厳密なデータ型を知る必要はありません。しかも、列定義を変更すると、変数宣言もそれに応じて自動的に変

更されます。これによって、データ独立性を提供し、メンテナンス・コストを削減し、データベース変更時にプログラムが順応できるようになります。

5.1.3 カーソルFORループ

PL/SQLを使用すれば、カーソルを定義して操作するために、DECLARE、OPEN、FETCHおよびCLOSE文を使用する必要はありません。かわりに、カーソルFORループを使用でき、ループ索引をレコードとして暗黙的に宣言し、指定された問合せに関連付けられているカーソルをオープンして、データを繰り返しカーソルからフェッチしてレコードに入れてから、カーソルをクローズします。次に例を示します。

```
DECLARE
 ...
BEGIN
 FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP
 IF emprec.comm / emprec.sal > 0.25 THEN ...
 ...
 END LOOP;
END;
```

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。

5.1.4 サブプログラム

PL/SQLにはプロシージャとファンクションと呼ばれる2種類のサブプログラムがあり、これらを使用すると、各動作を分離できるため、アプリケーションの開発が容易になります。一般的に、プロシージャは処理の実行に使用し、ファンクションは値の計算に使用します。

プロシージャおよびファンクションには拡張性があります。つまり、プロシージャとファンクションを使用することにより、PL/SQL言語を必要に応じて調整できます。たとえば、新しい部門を作成するプロシージャが必要な場合、次のように記述します。

```
PROCEDURE create_dept
 (new_dname IN CHAR(14),
 new_loc IN CHAR(13),
 new_deptno OUT NUMBER(2)) IS
BEGIN
 SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
 INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);
END create_dept;
```

このプロシージャをコールすると、プロシージャでは新しい部門名と場所を受け取り、部門番号データベース順序の次の値を選択し、その新しい番号、名前および場所を*dept*表に挿入してから、新しい番号をコール元に戻します。

サブプログラムは(CREATE FUNCTIONおよびCREATE PROCEDUREを使用して)データベースに格納できます。こうすることで、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。

5.1.5 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・モードを使用します。パラメータ・モードにはIN (デフォルト)、OUTおよびIN OUTの3つがあります。7INパラメータを使用すると、コールされるプログラムに値を渡せます。OUTパラメータを使用すると、サブプログラムのコール元に値を戻せます。IN OUTパラメータを使用すると、コールされるサブプログラムに初期値を渡し、コール元には更新された値を戻せます。

それぞれの実パラメータのデータ型は、対応する仮パラメータのデータ型に変換可能であることが必要です。表3-6は、データ型間の有効な変換を示しています。

5.1.6 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Oracleデータベースに格納でき、そこでそ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、仕様部および本体の2つの部分があります。仕様部とは、アプリケーションへのインタフェースで、使用可能な型、定数、変数、例外、カーソルおよびサブプログラムが宣言されます。本体は、カーソルおよびサブプログラムを定義して、仕様を実行します。次の例では、2つの雇用プロシージャをパッケージ化しています。

```
PACKAGE emp_actions IS -- package specification
 PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
 PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;
PACKAGE BODY emp_actions IS -- package body
 PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
 BEGIN
 INSERT INTO emp VALUES (empno, ename, ...);
 END hire_employee;
 PROCEDURE fire_employee (emp_id NUMBER) IS
 BEGIN
 DELETE FROM emp WHERE empno = emp_id;
 END fire_employee;
END emp_actions;
```

パッケージ仕様部内の宣言のみ参照可能で、アプリケーションからアクセスできます。パッケージ本体中の詳細な実装内容は非表示のためアクセスできません。

5.1.7 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(ただし、同じではありません)。PL/SQL表は1列からなり、主キーを使用して、配列と同じ方法で行にアクセスします。列は任意のスカラー型(CHAR、DATEまたはNUMBERなど)にできますが、主キーはBINARY_INTEGER型にする必要があります。

ブロック、プロシージャ、ファンクションまたはパッケージのいずれかの宣言部でPL/SQL表型を宣言できます。次の例では、*NumTabTyp*と呼ばれるTABLE型を宣言しています。

```
DECLARE
 TYPE NumTabTyp IS TABLE OF NUMBER
 INDEX BY BINARY_INTEGER;
 ...
BEGIN
 ...
END;
```

次の例に示すように、*NumTabTyp*型を定義すると、その型のPL/SQL表を宣言できます。

```
num_tab NumTabTyp;
```

識別子*num_tab*は、PL/SQL表全体を表しています。

配列に似た構文を使用してPL/SQL表の中の行を参照し、主キーの値を指定します。たとえば、*num_tab*の名前のPL/SQL表の中の9番目の行を参照するには次のように指定します。

```
num_tab(9) ...
```

5.1.8 ユーザー定義のレコード

%ROWTYPE属性を使用して、データベース表内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。ただし、レコード内のフィールドのデータ型は指定できず、ユーザー独自のフィールドも定義できません。複合データ型RECORDを使用すると、これらの制限を取り除くことができます。

RECORD型のオブジェクトはレコードと呼ばれます。PL/SQL表とは異なり、レコードには一意の名前のフィールドがあり、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、給与、雇用日など)があるとします。このデータは、型は異なりますが、論理的に関連しています。従業員の名前、給与および雇用日などのフィールドを持つレコードによって、1つの論理単位としてデータを処理できます。

ブロック、プロシージャ、ファンクションまたはパッケージのいずれかの宣言部で、レコード型およびレコード・オブジェクトを宣言できます。次の例では、*DeptRecTyp*という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、一意の名前と特定のデータ型があります。どのフィールド宣言にもNOT NULLオプションを追加でき、そのフィールドへのNULLの割当てを防止します。ただし、NOT NULL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、*DeptRecTyp*型を定義すると、その型のレコードを宣言できます。

```
dept_rec DeptRecTyp;
```

識別子*dept_rec*は、レコード全体を表しています。

レコード内の個々のフィールドを参照するには、ドット表記法を使用します。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 ...
```

5.2 PL/SQLブロックの埋込みについて

Oracleプリコンパイラでは、PL/SQLブロックが1つの埋込みSQL文と同様に扱われます。したがって、PL/SQLブロックは、ホスト・プログラム内でSQL文を記述できる場所であれば、どこにでも記述できます。

PL/SQLブロックをホスト・プログラムに埋め込むには、次のように、PL/SQLブロックをキーワードのEXEC SQL EXECUTEとEND-EXECで囲みます。

```
EXEC SQL EXECUTE
 DECLARE
 ...
 BEGIN
 ...
 END;
END-EXEC;
```

END-EXECキーワードの後には、ホスト言語の文終了文字を付ける必要があります。

プログラムにPL/SQLブロックを埋め込む場合、PL/SQLはOracleで解析する必要があるため、プリコンパイラ・オプションSQLCHECK=SEMANTICSを指定します。Oracleに接続するには、USERIDオプションも指定する必要があります。詳細は、プリコンパイラ・オプションの使用についてを参照してください。

5.3 ホスト変数の使用方法について

ホスト変数は、ホスト言語とPL/SQLブロック間の通信を仲介します。ホスト変数はPL/SQLと共有できるので、PL/SQLではホスト変数の設定および参照ができます。

たとえば、ユーザーに情報の提供を求め、この情報をPL/SQLブロックに渡すためのホスト変数を使用するようにユーザーに指示できます。これにより、PL/SQLではデータベースにアクセスし、ホスト変数を使用して結果をホスト・プログラムに戻すことができます。

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変数を使用できる場所であればそのブロックのどこにでも使用できます。ただし、文字ホスト変数は、長さが255文字以内です。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。

5.3.1 例

次の例では、PL/SQLにおけるホスト変数の使用方法を示します。プログラムでは、ユーザーに従業員番号の入力を要求し、その番号に応じて従業員の役職名、雇用日および給与を表示します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_number INTEGER;
 job_title CHARACTER(20);
 hire_date CHARACTER(9);
 salary REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
LOOP
 display 'Employee Number (0 to end)? ';
 read emp_number;
 IF emp_number = 0 THEN
 EXEC SQL COMMIT WORK RELEASE;
 display 'Exiting program';
 exit program;
 ENDIF;
 ---------------- 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;
ENDLOOP;
```

```
...
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

ホスト変数*emp_number*がPL/SQLブロックが入力される前に設定され、ホスト変数*job_title*、*hire_date*および*salary*がブロック内で設定されていることに注意してください。

5.3.2 複雑な例

次の例では、ユーザーに銀行口座番号、取引の種類、取引金額の入力を要求し、その後、口座の借方または貸方に記入します。口座が存在しない場合、例外が発生します。取引が完了すると、そのステータスを表示します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 acct_num INTEGER;
 trans_type CHARACTER(1);
 trans_amt REAL;
 status CHARACTER(80);
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
LOOP
 display 'Account Number (0 to end)? ';
 read acct_num;
 IF acct_num = 0 THEN
 EXEC SQL COMMIT WORK RELEASE;
 display 'Exiting program';
 exit program;
 ENDIF;
 display 'Transaction Type - D)ebit or C)redit? '
 read trans_type;
 display 'Transaction Amount? '
 read trans_amt;
 ---------------------- begin PL/SQL block -------------------
 EXEC SQL EXECUTE
 DECLARE
 old_bal NUMBER(9,2);
 err_msg CHAR(70);
 nonexistent EXCEPTION;
 BEGIN
 :trans_type := UPPER(:trans_type);
 IF :trans_type = 'C' THEN -- credit the account
 UPDATE accts SET bal = bal + :trans_amt
 WHERE acctid = :acct_num;
 IF SQL%ROWCOUNT = 0 THEN -- no rows affected
 RAISE nonexistent;
 ELSE
```

```
 :status := 'Credit applied';
 END IF;
 ELSIF :trans_type = 'D' THEN -- debit the account
 SELECT bal INTO old_bal FROM accts
 WHERE acctid = :acct_num;
 IF old_bal >= :trans_amt THEN -- enough funds
 UPDATE accts SET bal = bal - :trans_amt
 WHERE acctid = :acct_num;
 :status := 'Debit applied';
 ELSE
 :status := 'Insufficient funds';
 END IF;
 ELSE
 :status := 'Invalid type: ' || :trans_type;
 END IF;
 COMMIT;
 EXCEPTION
 WHEN NO_DATA_FOUND OR nonexistent THEN
 :status := 'Nonexistent account';
 WHEN OTHERS THEN
 err_msg := SUBSTR(SQLERRM, 1, 70);
 :status := 'Error: ' || err_msg;
 END;
 END-EXEC;
 ------------------- end PL/SQL block -----------------------
 display 'Status: ', status;
ENDLOOP;
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
END sql_error;
```

5.3.3 VARCHAR擬似型

プログラム要件への対応で、可変長文字列の宣言にVARCHAR擬似型が使用できると説明したことを思い出してください。VARCHARが入力ホスト変数の場合は、予想される長さをOracleに通知する必要があります。したがって、長さフィールドを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、Oracleでは自動的に長さフィールドが設定されます。しかし、PL/SQLブロックでVARCHAR出力ホスト変数を使用するには、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、長さフィールドを宣言されたVARCHARの(最大の)長さに設定してください。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number INTEGER;
 emp_name VARCHAR(10);
 salary REAL;
 ...
EXEC SQL END DECLARE SECTION;
...
set emp_name.len = 10; -- initialize length field
EXEC SQL EXECUTE
 BEGIN
 SELECT ename, sal INTO :emp_name, :salary
 FROM emp
 WHERE empno = :emp_number;
```

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

5.4 インジケータ変数の使用について

PL/SQLではNULLを操作できるため、インジケータ変数は必要ありません。たとえば、PL/SQL内では、次のようにIS NULL演算子を使用してNULLがないか検査できます。

```
IF variable IS NULL THEN ...
```

次のように、代入演算子(:=)を使用してNULLを割り当てることができます。

```
variable := NULL;
```

しかし、ホスト言語ではNULLを操作できないため、インジケータ変数が必要です。埋込みPL/SQLでは、次の目的でインジケータ変数を使用できるため、この要件を満たします。

- ホスト・プログラムからのNULL入力の受入れ
- NULLまたは切り捨てられた値のホスト・プログラムへの出力

PL/SQLブロックでインジケータ変数を使用するときは、次の規則に従ってください。

- インジケータ変数は単独では参照できません。関連付けられたホスト変数に追加する必要があります。
- インジケータ変数を指定してホスト変数を参照する場合、同じブロックでは常に同じ方法で参照する必要があります。

次の例では、SELECT文でインジケータ変数*ind_comm*がホスト変数*commission*とともに指定されているため、IF文でも同様に指定する必要があります。

```
EXEC SQL EXECUTE
 BEGIN
 SELECT ename, comm
 INTO :emp_name, :commission:ind_comm FROM emp
 WHERE empno = :emp_number;
 IF :commission:ind_comm IS NULL THEN ...
 ...
 END;
END-EXEC;
```

PL/SQLでは、*:commission:ind_comm*が他の単純な変数と同じように扱われることに注意してください。PL/SQLブロック内の標識変数は直接参照できませんが、PL/SQLでは、ブロックに入るときに標識変数の値がチェックされ、ブロックから出るときにその値が正しく設定されます。

5.4.1 NULLの処理

ブロックに入るとき、インジケータ変数の値が-1であれば、PL/SQLでは自動的にNULLがホスト変数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により自動的に-1の値がインジケータ変数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状態のことです。

```
EXEC SQL EXECUTE
 BEGIN
 IF :salary:ind_sal IS NULL THEN
 RAISE salary_missing;
 END IF;
```

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

5.4.2 切り捨てられた値の処理

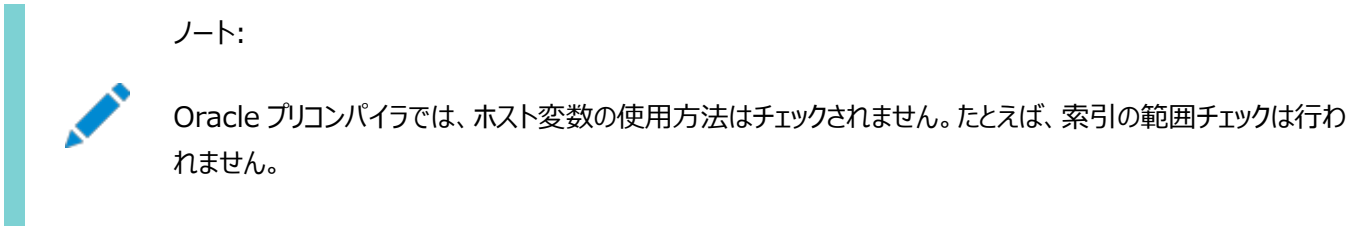
PL/SQLでは、切り捨てられた文字列の値がホスト変数に割り当てられても、例外とはみなされません。ただし、標識変数を指定している場合には、PL/SQLによってその標識変数が文字列の元の長さに設定されます。次の例では、ホスト・プログラムは、*ind_name*の値をチェックして、切り捨てられた値が*emp_name*に割り当てられたかどうかを判別できます。

```
EXEC SQL EXECUTE
 DECLARE
 ...
 new_name CHAR(10);
 BEGIN
 ...
 :emp_name:ind_name := new_name;
 ...
 END;
END-EXEC;
```

5.5 ホスト配列の使用について

入力ホスト配列およびインジケータ配列は、PL/SQLブロックに渡せます。これらには、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によって索引付けができます。通常は、ホスト配列全体がPL/SQLに渡されますが、ARRAYLEN文(後述)を使用すれば、より小さい配列サイズを指定できます。

さらに、サブプログラム・コールを使用して、ホスト配列のすべての値をPL/SQL表の複数の行に割り当てることができます。配列のサブスクリプトの範囲が*m*から*n*の場合、対応するPL/SQL表の索引範囲は常に1から*(n - m + 1)*になります。たとえば、配列サブスクリプト範囲が5から10の場合、対応するPL/SQL表の索引範囲は、1から(10 - 5 + 1)または1から6です。



ノート:

Oracle プリコンパイラでは、ホスト変数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*というホスト配列をPL/SQLブロックに渡し、ブロックではこのホスト配列がファンクション・コールで使用されます。このファンクションは、一連の数値の中央値を検出するため、*median*という名前が付いています。その仮パラメータには、*num_tab*というPL/SQL表が含まれています。このファンクション・コールにより、実パラメータ*salary*内のすべての値を仮パラメータ*num_tab*内の行に割り当てます。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 salary (100) REAL;
 median_salary REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host array
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, n);
 ...
 END;
END-EXEC;
```

また、サブプログラム・コールを使用して、PL/SQL表内のすべての行をホスト配列内の対応する要素に割り当てることもできます。

表5-1に、PL/SQL表の行の値とホスト配列の要素間での有効な変換を示しています。たとえば、LONG型のホスト配列は、VARCHAR2、LONG、RAWまたはLONG RAW型のPL/SQL表と互換性があります。ただし、CHAR型のPL/SQL表とは互換性がありません。

表5-1 PL/SQL表の行値とホスト配列の要素の有効な変換

PL/SQL表	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	_/							
CHARZ	_/							
DATE		_/						
DECIMAL					_/			
DISPLAY					_/			
FLOAT					_/			
INTEGER					_/			
LONG	_/		_/					
LONG VARCHAR			_/	_/		_/		_/
LONG VARRAW				_/		_/		
NUMBER					_/			

PL/SQL表	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
RAW				_/		_/		
ROWID							_/	
STRING			_/	_/		_/		_/
UNSIGNED					_/			
VARCHAR			_/	_/		_/		_/
VARCHAR2			_/	_/		_/		_/
VARNUM					_/			
VARRAW				_/		_/		

5.5.1 ARRAYLEN文

入力ホスト配列をPL/SQLブロックに渡して処理する必要があるとします。デフォルトでは、このようなホスト配列をバインドする際、Oracleプリコンパイラでは、宣言されたサイズが使用されます。ただし、配列全体を処理する必要がない場合があります。この場合、ARRAYLEN文を使用して、より小さい配列サイズを指定できます。ARRAYLEN文は、ホスト配列を格納サイズがより小さいホスト変数と関連付けます。文の構文は次のとおりです。

```
EXEC SQL ARRAYLEN host_array (dimension);
```

*dimension*は4バイトの整数ホスト変数であり、リテラルや式ではありません。

ARRAYLEN文は、宣言部で*host_array*および*dimension*宣言の後に指定する必要があります。ホスト配列にオフセットは指定できません。しかし、その目的にはホスト言語の機能を使用できる場合があります。

次の例では、ARRAYLENを使用して、*bonus*というホスト配列のデフォルトのサイズをオーバーライドします。

```
EXEC SQL BEGIN DECLARE SECTION;
 bonus (100) REAL;
 my_dim INTEGER;
 EXEC SQL ARRAYLEN bonus (my_dim);
EXEC SQL END DECLARE SECTION;
-- populate the host array
...
set my_dim = 25; -- set smaller array dimension
EXEC SQL EXECUTE
 DECLARE
 TYPE NumTabTyp IS TABLE OF REAL
 INDEX BY BINARY_INTEGER;
 median_bonus REAL;
 FUNCTION median (num_tab NumTabTyp, n INTEGER)
 RETURN REAL IS
 BEGIN
```

```
 -- compute median
 END;
 BEGIN
 median_bonus := median(:bonus, :my_dim);
 ...
 END;
END-EXEC;
```

ARRAYLENによってホスト配列のサイズが100要素から25要素に減るため、PL/SQLブロックには25の配列要素のみが渡されます。その結果、PL/SQLブロックが実行のためOracleに送信されるとき、一緒に送られるホスト配列はずっと小さくなります。これにより、時間を節約し、ネットワーク化された環境でネットワークの通信量を削減できます。

5.6 カーソルの使用方法について

すべての埋込みSQL文は、DECLARE CURSOR文で明示的に、またはプリコンパイラによって暗黙的に、カーソルを割り当てられます。内部的には、Oracle Precompilersはカーソル・キャッシュと呼ばれるキャッシュを維持して、埋込みSQL文の実行を制御します。すべてのSQL文は実行時に、カーソル・キャッシュ内にエントリを割り当てられます。このエントリは、Oracle内のプログラム・グローバル領域(PGA)にあるプライベートSQL領域にリンクされます。

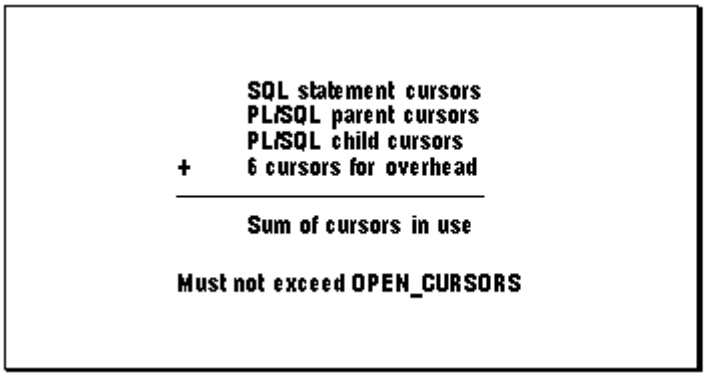
MAXOPENCURSORS、HOLD_CURSORおよびRELEASE_CURSORなどの各種プリコンパイラ・オプションを使用して、カーソル・キャッシュを管理することで、パフォーマンスが改善されます。たとえば、RELEASE_CURSORでは、カーソル・キャッシュとプライベートSQL領域間のリンクに起こることを制御します。RELEASE_CURSOR=YESを指定すると、OracleでSQL文が実行された後、リンクが削除されます。これにより、プライベートSQL領域に割り当てられたメモリーが解放され、解析ロックが解除されます。

カーソル・キャッシュ管理のために、埋込みPL/SQLブロックはSQL文と同様に扱われます。実行時に、親カーソルと呼ばれるカーソルが、PL/SQLブロック全体と関連付けられます。カーソル・キャッシュには対応するエントリが作成され、このエントリがPGA内のプライベートSQL領域にリンクします。

PL/SQLブロック内の各SQL文にも、PGAのプライベートSQL領域が必要です。したがって、これらのSQL文のために、PL/SQLでは子カーソル・キャッシュと呼ばれる個別のキャッシュが管理されます。このカーソルは、子カーソルと呼ばれます。子カーソル・キャッシュの管理はPL/SQLによって行われるため、ユーザーは子カーソルを直接制御できません。

プログラムで同時に使用できるカーソルの最大数は、Oracle初期化パラメータのOPEN_CURSORSによって設定されます。図5-1は、使用されるカーソルの最大数を計算する方法を示しています。

図5-1 使用される最大カーソル数



OPEN_CURSORSで設定された制限をプログラムが超えると、次のOracleエラーが発生します。

```
ORA-01000: maximum open cursors exceeded
```

このエラーは、RELEASE_CURSOR=YESオプションとHOLD_CURSOR=NOオプションを指定すれば回避できます。RELEASE_CURSORをYESに設定してプログラム全体をプリコンパイルする必要ない場合は、次のように、単に各PL/SQLブロックの後でオプションをNOにリセットします。

```
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
-- first embedded PL/SQL block
 EXEC ORACLE OPTION (RELEASE_CURSOR=NO);
-- embedded SQL statements
 EXEC ORACLE OPTION (RELEASE_CURSOR=YES);
-- second embedded PL/SQL block
 EXEC ORACLE OPTION (RELEASE_CURSOR=NO);
-- embedded SQL statements
```

5.6.1 代替方法

MAXOPENCURSORSオプションは、カーソル・キャッシュの初期サイズを指定します。たとえば、MAXOPENCURSORS=10の場合、カーソル・キャッシュでは最大10エントリを保持できます。新しいカーソルが必要なのに、空いているキャッシュ・エントリがなく、HOLD_CURSOR=NOに設定されている場合、プリコンパイラではエントリの再利用を試みます。MAXOPENCURSORSに非常に少ない値を指定すると、プリコンパイラは親カーソルを頻繁に再利用せざるを得なくなります。親カーソルが再利用されると同時に、子カーソルはすべて解放されます。

5.7 ストアド・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)は、別々にコンパイルしてOracleデータベースに格納し、起動できます。SQL*PlusなどのOracleツールを使用して明示的に作成したサブプログラムを、ストアド・サブプログラムと呼びます。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・サブプログラムがアプリケーションによってOracleに送られると、それはインライン・サブプログラムと呼ばれます。Oracleでは、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、ソースまたはオブジェクト・コードをデータ・ディクショナリに格納することはありません。

パッケージ内で定義されているサブプログラムは、そのパッケージの一部とみなされ、パッケージ・サブプログラムと呼ばれます。パッケージ内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・プログラムと呼ばれます。

5.7.1 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をホスト・プログラムに埋め込むことができます。

```
EXEC SQL CREATE
 FUNCTION sal_ok (salary REAL, title CHAR)
 RETURN BOOLEAN AS
 min_sal REAL;
 max_sal REAL;
 BEGIN
 SELECT losal, hisal INTO min_sal, max_sal
 FROM sals
 WHERE job = title;
 RETURN (salary >= min_sal) AND
 (salary <= max_sal);
 END sal_ok;
END-EXEC;
```

埋込みCREATE {FUNCTION | PROCEDURE | PACKAGE}文がハイブリッドであることに注意してください。他のすべての埋込みCREATE文と同様に、キーワードEXEC SQL(EXEC SQL EXECUTEではない)で始まります。ただし、他の埋込みCREATE文と異なり、PL/SQLの終了文字END-EXECで終わります。

次の例では、*emp*表からひとまとまりの行をフェッチする*get_employees*というプロシージャを含むパッケージを作成します。バッチ・サイズは、プロシージャのコール元(別のストアド・サブプログラムの場合もあれば、クライアント・アプリケーションの場合もある)によって決められます。

プロシージャでは、3つのPL/SQL表をOUT仮パラメータとして宣言し、その後、従業員のバッチ・データをPL/SQL表にフェッチします。一致する実パラメータはホスト配列です。プロシージャの終了時には、PL/SQL表のすべての行の値が、ホスト配列の対応する要素に自動的に割り当てられます。

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
 TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
 INDEX BY BINARY_INTEGER;
 TYPE NumArrayTyp IS TABLE OF FLOAT
 INDEX BY BINARY_INTEGER;
 PROCEDURE get_employees(
 dept_number IN INTEGER,
 batch_size IN INTEGER,
 found IN OUT INTEGER,
 done_fetch OUT INTEGER,
 emp_name OUT CharArrayTyp,
 job-title OUT CharArrayTyp,
 salary OUT NumArrayTyp);
 END emp_actions;
END-EXEC;
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS
 CURSOR get_emp (dept_number IN INTEGER) IS
 SELECT ename, job, sal FROM emp
 WHERE deptno = dept_number;
 PROCEDURE get_employees(
 dept_number IN INTEGER,
 batch_size IN INTEGER,
 found IN OUT INTEGER,
 done_fetch OUT INTEGER,
 emp_name OUT CharArrayTyp,
 job_title OUT CharArrayTyp,
 salary OUT NumArrayTyp) IS
 BEGIN
 IF NOT get_emp%ISOPEN THEN
 OPEN get_emp(dept_number);
 END IF;
 done_fetch := 0;
 found := 0;
 FOR i IN 1..batch_size LOOP
 FETCH get_emp INTO emp_name(i),
 job_title(i), salary(i);
 IF get_emp%NOTFOUND THEN
 CLOSE get_emp;
 done_fetch := 1;
 EXIT;
 ELSE
 found := found + 1;
 END IF;
 END LOOP;
 END get_employees;
 END emp_actions;
END-EXEC;
```

CREATE文でREPLACE句を指定すると、パッケージの削除、再作成および権限の再付与を行わなくても、既存のパッケージを再定義できます。CREATE文の完全な構文は、『*Oracle Database SQL言語リファレンス*』を参照してください。

埋込みCREATE {FUNCTION|PROCEDURE|PACKAGE}文が失敗した場合、Oracleではエラーではなく警告が出ます。

5.7.2 ストアド・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・サブプログラムを起動(コール)するには、無名PL/SQLブロックを使用する必要があります。次の例では、*raise_salary*というスタンドアロン・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*の名前のパッケージに格納されます。したがって、プロシージャ・コールを完全に修飾するにはドット表記法を使用する必要があります。

```
EXEC SQL EXECUTE
 BEGIN
 emp_actions.raise_salary(:emp_id, :increase);
 END;
END-EXEC;
```

IN実パラメータには、リテラル、ホスト変数、ホスト配列、PL/SQL定数または変数、PL/SQL表、PL/SQLユーザー定義レコード、サブプログラム・コールまたは式を使用できます。これに対してOUT実パラメータには、リテラル、サブプログラム・コールおよび式は使用できません。

次のPro*Cの例では、3つの仮パラメータがPL/SQL表で、対応する実パラメータはホスト配列です。プログラムでは、ストアド・プロシージャ*get_employees*を繰り返しコールし、データがなくなるまで、従業員データの各バッチを表示します。

```
#include <stdio.h>
#include <string.h>
typedef char asciz;
EXEC SQL BEGIN DECLARE SECTION;
 /* Define type for null-terminated strings */
 EXEC SQL TYPE asciz IS STRING(20);
 asciz username[20];
 asciz password[20];
 int dept_no; /* which department to query */
 char emp_name[10][21];
 char job[10][21];
 float salary[10];
 int done_flag;
 int array_size;
 int num_ret; /* number of rows returned */
 int SQLCODE;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE sqlca;
int print_rows(); /* produces program output */
int sql_error(); /* handles NOLOGGING errors */
main()
{
 int i;
 /* Connect to Oracle. */
 strcpy(username, "SCOTT");
 strcpy(password, "TIGER");
```

```
 EXEC SQL WHENEVER SQLERROR DO sql_error();
 EXEC SQL CONNECT :username IDENTIFIED BY :password;
 printf("¥nConnected to Oracle as user: %s¥n", username);
 printf("enter department number: ");
 scanf("%d", &dept_no);
 fflush(stdin);
 /* Set the array size. */
 array_size = 10;
 done_flag = 0;
 num_ret = 0;
 /* Array fetch loop - ends when done_flag is true. */
 for (;;)
 {
 EXEC SQL EXECUTE
 BEGIN emp_actions.get_employees
 (:dept_no, :array_size, :num_ret,
 :done_flag, :emp_name, :job, :salary);
 END;
 END-EXEC;
 print_rows(num_ret);
 if (done_flag)
 break;
 }
 /* Disconnect from the database. */
 EXEC SQL COMMIT WORK RELEASE;
 exit(0);
}
print_rows(n)
int n;
{
 int i;
 if (n == 0)
 {
 printf("No rows retrieved.¥n");
 return;
 }
 printf("¥n¥nGot %d row%c¥n", n, n == 1 ? '¥0' : 's');
 printf("%-20.20s%-20.20s%s¥n", "Ename", "Job", "Salary");
 for (i = 0; i < n; i++)
 printf("%20.20s%20.20s%6.2f¥n",
 emp_name[i], job[i], salary[i]);
}
sql_error()
{
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 printf("¥nOracle error detected:");
 printf("¥n% .70s ¥n", sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK WORK RELEASE;
 exit(1);
}
```

それぞれの実パラメータのデータ型は、対応する仮パラメータのデータ型に変換可能であることが必要です。また、ストアド・サブプログラムを終了する前には、すべてのOUT仮パラメータは割り当てられた値であることが必要です。そうしないと、対応する実パラメータの値が未確定になります。

5.7.3 リモート・アクセス

PL/SQLを使用すると、データベース・リンクを経由してリモート・データベースにアクセスできます。一般的に、データベース・リンク

は、データベース管理者(DBA)によって設定され、Oracleデータ・ディクショナリに格納されます。データベース・リンクは、リモート・データベースの位置、リモート・データベースへのパス、使用するOracleユーザー名およびパスワードをOracleに伝えます。次の例では、データベース・リンク*dallas*を使用して、*raise_salary*プロシージャをコールします。

```
EXEC SQL EXECUTE
 BEGIN
 raise_salary@dallas(:emp_id, :increase);
 END;
END-EXEC;
```

次の例に示すように、シノニムを作成して、リモート・サブプログラムに位置の透過性を与えることができます。

```
CREATE PUBLIC SYNONYM raise_salary FOR raise_salary@dallas;
```

5.7.4 ストアド・サブプログラムに関する情報の取得

プログラム要件への対応では、ホスト・プログラムにOCIコールを埋め込む方法を学習しました。ライブラリ・ルーチンSQLLDAをコールして、LDAを設定すると、OCIコールODESSPを使用して、ストアド・サブプログラムに関する有益な情報を取得できます。ODESSPをコールするときには、有効なLDAとサブプログラムの名前を渡す必要があります。パッケージ・プログラムの場合は、パッケージ名も渡す必要があります。ODESSPからは、各サブプログラム・パラメータについて、そのデータ型、サイズ、位置などの情報が戻されます。

Oracle付属のDBMS_DESCRIBEパッケージでは、*describe_procedure*プロシージャも使用できます。

5.8 動的PL/SQLの使用について

Oracleプリコンパイラでは、PL/SQLブロック全体が1つのSQL文のように扱われます。つまり、PL/SQLブロックをホスト変数の文字列に格納できることを意味します。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を使用して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を使用してPL/SQL文字列を準備し、実行します。ブロックに含まれるホスト変数の数がわからない場合は、動的SQL方法4を使用する必要があります。詳細は、動的SQLの使用方法を参照してください。

5.8.1 制限事項

動的SQL方法4では、TABLE型のパラメータを使用して、ホスト配列をPL/SQLプロシージャにバインドすることはできません。

6 Oracleプリコンパイラの実行

この章は、次の項で構成されています。

この章では、Oracleプリコンパイラを実行するための要件について詳しく説明します。プリコンパイル中に何が起こるか、プリコンパイラ・コマンドを発行する方法、多くの便利なプリコンパイラ・オプションを指定する方法、条件付きプリコンパイルと分割プリコンパイルの実行方法について学習します。

6.1 プリコンパイラのコマンド

Oracleプリコンパイラを実行するには、表6-1に示す言語固有のコマンドの1つを発行します。

表6-1 プリコンパイラの実行コマンド

ホスト言語	プリコンパイラのコマンド
COBOL	procob
FORTRAN	profor

プリコンパイラの場所はシステムごとに異なります。通常、システム管理者またはデータベース管理者は、環境変数、論理名または別名を定義するか、その他のシステム固有の方法を使用して、プリコンパイラ実行ファイルをアクセス可能にします。

INAMEオプションは、プリコンパイル対象のソースファイルを指定します。たとえば、次のPro*COBOLコマンドを入力します。

```
procob INAME=test
```

すると、カレント・ディレクトリの*test.pco*ファイルがプリコンパイルされますが、これはプリコンパイラではファイル名の拡張子が*.pco*とみなされるためです。非標準の拡張子でなければ、INAMEを指定する際にファイル拡張子を使用する必要はありません。

入力ファイル名および出力ファイル名を、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い場合、プリコンパイラではコマンドラインで最初に指定したファイル名を入力ファイル名、2番目のファイル名を出力ファイル名とみなします。

Pro*FORTRANコマンドを次のように入力したとします。

```
profor MODE=ANSI myfile.pfo DBMS=V7 myfile.f
```

これは次に相当します。

```
profor MODE=ANSI INAME=myfile.pfo DBMS=V7 ONAME=myfile.f
```

ノート:

特定のオペレーティング・システム・オブジェクト(ファイル名など)の名前を指定しないオプション名とオプション値には、大文字と小文字の区別はありません。このマニュアルの例では、オプション名は大文字で記述し、オプション値は通常小文字で記述しています。プリコンパイラ実行ファイル自体の名前も含め、ファイル名を入力するときには、大文字と小文字の区別は、オペレーティング・システムの表記規則に従ってください。

6.2 プリコンパイル中の状況

プリコンパイル中、ホスト・プログラムに埋め込まれたSQL文は、Oracleプリコンパイラが生成するホスト言語コードに置き換えられます。生成されたコードには、各ホスト変数のデータ型、長さおよびアドレスを含むデータ構造体と、Oracleランタイム・ライブラリのSQLLIBで必要なその他の情報が含まれています。このコードには、埋込みSQLの動作を実行するSQLLIBルーチンに対するコールも含まれています。

生成されたコードには、埋込みSQLの動作を実行するSQLLIBルーチンに対するコールも含まれています。プリコンパイラでは、Oracle Call Interface(OCI)ルーチンに対するコールは生成されないので注意してください。

プリコンパイラでは、Oracle Call Interface(OCI)に対するコールは生成されません。

プリコンパイラは警告やエラー・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には接頭辞PCC-が付いており、『*Oracle Database*エラー・メッセージ』で説明されています。

6.3 プリコンパイラのオプション

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。それらを使用して、リソースの使用、エラーの報告、入出力の書式化およびカーソルの管理を制御できます。コンパイラ・オプションを指定するには、次の構文を使用します。

```
<option_name>=<value>
```

オプションの*value*は文字列リテラルで、テキスト値または数値を表します。たとえば、次のオプションを指定するとします。

```
... INAME=my_test
```

ここでは、値はファイル名を指定する文字列リテラルです。しかし、次のオプションの場合は違います。

```
... MAXOPENCURSORS=20
```

この値は数値です。

オプションの中にはブール値をとるものもあり、文字列のYESまたはNO、TRUEまたはFALSE、整数リテラルの1または0を指定できます。たとえば、次のオプションを指定するとします。

```
... SELECT_ERROR=YES
```

これは次に相当します。

```
... SELECT_ERROR=TRUE
```

または

```
... SELECT_ERROR=1
```

オプション名とオプション値の間には必ず等号を入れ、空白により個々のオプションが区切られるため、等号の前後には空白を入れません。たとえば、コマンドラインで次のようにAUTO_CONNECTオプションを指定するとします。

```
... AUTO_CONNECT=YES
```

省略が一義的であるならば、オプション名を省略してもかまいません。たとえば、省略形のMAXは、MAXLITERALまたはMAXOPENCURSORSのいずれの略語とも考えられるので、使用できません。

プリコンパイラのオプションについては、オンラインで便利なリファレンスを利用できます。オンラインの表示を見るには、オペレーティング・システムのプロンプトで、プリコンパイラ・コマンドを引数を指定せずに入力します。オンライン画面には、各オプションの名前および、構文、デフォルト値および用途が表示されます。アスタリスク(*)の付いたオプションは、インラインでもコマンドラインでも指定できます。

6.3.1 デフォルト値

多くのオプションにはデフォルト値があり、次の値によって決まります。

- プリコンパイラに組み込まれた値
- システム構成ファイルで設定された値
- ユーザー構成ファイルで設定された値
- インライン指定で設定された値セット

たとえば、MAXOPENCURSORSオプションでは、キャッシュ内のオープン・カーソルの最大数を指定します。このオプションのプリコンパイラに組み込まれたデフォルト値は10です。ただし、システム構成ファイルでMAXOPENCURSORS=32と指定されていると、デフォルトは32になります。ユーザー構成ファイルでは、これをさらに別の値に設定でき、これはシステム構成ファイルの値より優先されます。

次に、このオプションがコマンドラインで設定された場合、新しいコマンドライン値が優先されます。最終的には、インライン指定が前述のすべてのデフォルト値よりも優先されます。詳細は、「構成ファイル」を参照してください。

6.3.2 現在の値の確認

コマンドラインで疑問符(?)を使用すると、複数のオプションのカレント値を対話形式で調べることができます。たとえば、次のPro*COBOLコマンドを発行するとします。

```
procob ?
```

すべてのオプションの設定およびそのカレント値が端末に表示されます。この場合、値はプリコンパイラに組み込まれているもので、システム構成ファイルに値があれば、それが優先されます。一方、次のコマンドを発行したとします。

```
procob CONFIG=my_config_file.cfg ?
```

これでカレント・ディレクトリに*my_config_file.cfg*というファイルがあると、*my_config_file.cfg*ファイルのオプションが、その他のデフォルト値とともに表示されます。不足している値はユーザー構成ファイル内の値によって補われ、これらの値はプリコンパイラに組み込まれている値や、システム構成ファイルで指定されている値に優先します。

オプション名を指定し、その後に次の例のように=?を付ければ、1つのオプションの現在の値を確認できます。

```
procob MAXOPENCURSORS=?
```

ノート:

一部のオペレーティング・システムでは、疑問符(?)の前にバックスラッシュなどのエスケープ文字を付ける必要がある場合があります。たとえば、Pro*COBOL オプションの設定値を表示するには、「procob ?」のかわりに、「procob ¥?」を使用する必要があります。

6.3.3 大文字と小文字の区別

一般に、コマンドライン・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も使用できます。しかし、UNIXのように大文字と小文字を区別するオペレーティング・システムの場合、プリコンパイラ実行ファイルの名前を含め、ファイル名の値は大文字と小文字の正しい組合せを使用して指定する必要があります。

6.3.4 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
```

これらにより、FIPSオプションとMODEオプションのデフォルト値が設定されます。

システムにはそれぞれ1つの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プリコンパイラ固有で、表6-2に示すとおりです。

表6-2 システム構成ファイル

プリコンパイラ	構成ファイル
Pro*COBOL	pcbcfg.cfg
Pro*FORTRAN	pccfor.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の構成ファイルは通常*$ORACLE_HOME/procob*ディレクトリにあり、Pro*FORTRANの構成ファイルは*$ORACLE_HOME/profor*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)。

プリコンパイラの各ユーザーは、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。構成ファイルの名前は、CONFIGコマンドライン・オプションを使用して指定します。詳細は、「現在の値の確認」を参照してください。

ノート:

構成ファイルはネストできません。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。

6.4 オプションの入力

どのプリコンパイラ・オプションも、コマンドラインや(CONFIG以外は)構成ファイルから入力できます。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。プリコンパイラでは実行時に、これら3つのソースすべてから入力されたオプションを受け入れます。

6.4.1 コマンドライン

プリコンパイラ・オプションをコマンドラインで入力するには、次の構文を使用します。

```
... [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
```

6.4.2 インライン

EXEC ORACLE文を記述して、インラインでオプションを入力するには、次の構文を使用します。

```
EXEC ORACLE OPTION (option_name=value);
```

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);
```

インラインでオプションを入力すると、コマンドラインから入力された同じオプションは無効になります。

6.4.3 長所

EXEC ORACLE機能は、プリコンパイル中にオプションを変更するときに特に便利です。たとえば、HOLD_CURSORとRELEASE_CURSORの値を文ごとに変更できます。インライン・オプションを使用して実行時のパフォーマンスを最適化する方法については、パフォーマンス・チューニングを参照してください。

インラインでのオプションの指定は、オペレーティング・システムでコマンドラインでの入力文字数が制限されている場合にも便利であり、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)。

6.4.4 EXEC ORACLEのスコープ

EXEC ORACLE文は、同じオプションを指定する別のEXEC ORACLE文によって指定値(テキスト)が変更されるまで有効です。たとえば、HOLD_CURSOR=NOは、HOLD_CURSOR=YES:が指定されるまで有効です。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name CHARACTER(20);
 emp_number INTEGER;
 salary REAL;
 dept_number INTEGER;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
EXEC ORACLE OPTION (HOLD_CURSOR=NO);
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, DEPTNO FROM EMP;
EXEC SQL OPEN emp_cursor;
display 'Employee Number Dept';
display '--------------- ----';
LOOP
```

```
 EXEC SQL FETCH emp_cursor INTO :emp_number, :dept_number;
 display emp_number, dept_number;
ENDLOOP;
no_more:
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 LOOP
 display 'Employee number? ';
 read emp_number;
 IF emp_number = 0 THEN
 exit loop;
 EXEC ORACLE OPTION (HOLD_CURSOR=YES);
 EXEC SQL SELECT ENAME, SAL
 INTO :emp_name, :salary
 FROM EMP
 WHERE EMPNO = :emp_number;
 display 'Salary for ', emp_name, ' is ', salary;
 ENDLOOP;
...
```

6.4.5 構成ファイル

Oracleプリコンパイラでは、コマンドライン・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。デフォルトでは、システム構成ファイルと呼ばれるテキスト・ファイルが使用されます。ただし、コマンドラインでは、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を指定できます。

6.4.6 長所

構成ファイルにはいくつかの利点があります。システム構成ファイルを使用すると、すべてのプロジェク用にオプション・セットを標準化できます。ユーザー構成ファイルを使用すると、プロジェクトごとにオプションのセットをカスタマイズできます。構成ファイルにより、コマンドラインでオプションの長い文字列を入力する必要がなくなります。また、システムでコマンドラインの長さが制限されている場合、構成ファイルを使用すれば、コマンドラインで入力できるより多くのオプションを指定できます。

6.4.7 構成ファイルの使用について

構成ファイル内の各レコード(行)には、コマンドライン・オプションが1つ入っています。たとえば、構成ファイルには、FIPS、MODEおよびSQLCHECKの各オプションのデフォルト値を設定する次の行が含まれている場合があります。

```
FIPS=YES
MODE=ANSI
SQLCHECK=SEMANTICS
```

Oracleプリコンパイラごとに、独自のシステム構成ファイルを設定できます。ファイルの名前や場所は、言語やシステム固有です。ファイルが見つからない場合、警告が表示されますが、プリコンパイラでは処理が続行されます。

1つの言語にはシステム構成ファイルが1つしかありませんが、ユーザー構成ファイルはいくつでも作成できます。特定のユーザー構成ファイルの名前と場所を指定するには、新しいコマンドライン・オプションCONFIGを次のように指定します。

```
... CONFIG=<filename>
```

構成ファイルはネストできません。したがって、構成ファイルでCONFIGオプションは指定できません。また、インラインでCONFIGを指定することもできません。

6.4.8 オプション値の設定について

多くのプリコンパイラのランタイム・オプションには、デフォルト値が組み込まれており、これらは構成ファイルまたはコマンドラインでリ

セットできます。コマンドラインの設定は、ユーザー構成ファイルの設定に優先し、ユーザー構成ファイルの設定は、システム構成ファイルの設定に優先します。

6.5 オプションの有効範囲

プリコンパイル・ユニットは、ホスト言語のコードと1つ以上の埋込みSQL文を含むファイルです。特定のプリコンパイル・ユニットに対して指定したオプションは、そのプリコンパイル・ユニットにのみ効力を持ちます。

たとえば、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に接続すると有効になるMAXOPENCURSORS設定は、その接続が続くかぎり有効です。

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・ロジックの流れでそのインライン・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、ファイルの終わりまで有効です。

6.6 クイック・リファレンス

表6-3は、プリコンパイラ・オプションのクイック・リファレンスです。アスタリスクの付いたオプションは、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・リファレンスを表示するには、オペレーティング・システムのプロンプトでオプションを指定せずにプリコンパイラ・コマンドを入力します。画面には、各オプションの名前、構文、デフォルト値および用途が表示されます。

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、MicroFocus COBOLを使用するバイトスワップ・プラットフォームでは、COMP5オプションによって、一部のCOMPUTATIONAL項目の使用が管理されます。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6-3 プリコンパイラのオプション・クイック・リファレンス

構文	デフォルト値	指定内容
ASACC={YES|NO}	NO	リストのキャリッジ制御
ASSUME_SQLCODE={YES|NO}	NO	プリコンパイラでは SQLCODE が宣言されているとみなします。
AUTO_CONNECT={YES|NO}	NO	自動接続
CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF} *	CHARZ	文字配列および文字列のマップ
CHARSET_PICN={NCHAR_CHARSET | DB_CHARSET }	NCHAR_CHARSET	PIC N 変数で使用される文字セットの形式
CHARSET_PICX={NCHAR_CHARSET | DB_CHARSET }	DB_CHARSET	PIC X 値で使用される文字セットの形式

構文	デフォルト値	指定内容
CINCR	1	接続プールの CINCR 値。現在の物理接続数が CMAX 値より少ない場合、データベースに対してオープンされる物理接続数の次の増分をアプリケーションで設定できるようにします。
CLOSE_ON_COMMIT={YES | NO}	NO	COMMIT 時にすべてのカーソルをクローズします。
CMAX	100	データベースに対してオープンできる物理接続の最大数を指定します。
CMIN	2	接続プール内の物理接続の最小数を指定します。
CNOWAIT	0(未設定)	プール内の他のすべての物理接続が使用中で、物理接続の合計数がすでに最大値に達している場合に、アプリケーションで繰り返し物理接続を要求する必要があるかどうかを決定します。接続プールの CNOWAIT 値です。
CODE={ANSI_C | KR_C | CPP}	KR_C	生成される C コードの種類
COMMON_NAME=*block_name* *		FORTRAN COMMON ブロックの名前
COMMON_PARSER	NO	共通の SQL・フロント・エンドを使用して解析します。
COMP5	YES	COMP 変数ではなく COMP-5 を生成します。
COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}	MULTI_BYTE	C/C++コンパイラでサポートされる文字セットの種類
CONFIG=*filename*		ユーザー構成ファイルの名前
CPOOL	NO	接続プールをサポート。このオプションに基づき、プリコンパイラでは、SQLLIB に接続プール機能を有効または無効にするように指示する適切なコードを生成します。
CPP_SUFFIX=extension	*なし*	デフォルトの C++ファイル名拡張子をオーバーライドします。
CTIMEOUT	0	指定した時間(秒単位)より長い間アイドル状態になっている物理接続を終了し、オープンされている物理接

構文	デフォルト値	指定内容
		続を最適な数に保ちます。
DB2_ARRAY={YES |NO}	NO	DB2 配列の挿入/選択構文。このオプションに基づき、プリコンパイラでは、追加配列の挿入/選択構文をアクティブ化します。
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 の場合は、DECLARE SECTION が必要です。
DEF_SQLCODE={NO | YES}	NO	Pro*C/C++プリコンパイラで#define SQLCODE を生成するかどうかを制御します。
DEFINE=*symbol* *		条件付きプリコンパイルで使用されるシンボル
DURATION={TRANSACTION | SESSION}	TRANSACTION	キャッシュ内のオブジェクトの確保継続時間を設定します。
DYNAMIC={ANSI | ORACLE}	ORACLE	Oracle または ANSI SQL の意味を指定します。
END_OF_FETCH	1403	フェッチ終了時の SQLCODE 値
ERRORS={YES|NO} *	YES	端末にエラーが送信されるかどうかを指定します。
ERRTYPE=*filename*	*なし*	intype ファイル・エラー用のリスト・ファイルの名前
EVENTS	NO	パブリッシュ・サブスクライブ・イベント通知をサポートします。
FILE_ID	0	生成された COBOL ファイル用の一意の数値識別子
FIPS={YES|NO}*	NO	ANSI/ISO 拡張機能にフラグを立てるかどうかを指定します。
FORMAT={ANSI|TERMINAL}	ANSI	COBOL または FORTRAN 入力行の書式
Globalization Support_LOCAL={YES|NO}	YES	SQLLIB により実行される空白埋込み動作

構文	デフォルト値	指定内容
HEADER=*extension*	*なし*	intype ファイル・エラー・メッセージ用のリスト・ファイルの名前
HOLD_CURSOR={YES|NO}*	NO	カーソル・キャッシュで SQL 文を処理する方法
HOST={COBOL|COB74}	COBOL	入力ファイルの COBOL バージョン
IMPLICIT_SVPT	NO	バッファされた挿入の前の暗黙的セーブポイント
[INAME=]*filename*		入力ファイルの名前
INCLUDE=*path**		INCLUDEd ファイルのディレクトリ・パス
INTYPE=*filename*	*なし*	型情報の入力ファイルの名前
IRECLEN=*integer*	80	入力ファイルのレコード長
LINES={YES | NO}	NO	#line ディレクティブを生成するかどうか
LITDELIM={APOST|QUOTE} *	QUOTE	COBOL 文字列のデリミタ
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	リストのタイプ
MAXLITERAL=*integer* *	プラットフォーム固有	文字列の最大長
MAXOPENCURSORS=*integer* *	10	キャッシュされるカーソルの最大数
MAX_ROW_INSERT	0	挿入時にバッファされる最大行数
MODE={ORACLE|ANSI|ANSI14|ANSI13}	ORACLE	ANSI/ISO SQL 規格への準拠
MULTISUBPROG={YES|NO}	YES	FORTRAN COMMON ブロックが生成されるかどうか

構文	デフォルト値	指定内容
NATIVE_TYPES	NO	ネイティブ float/double のサポート
NESTED={YES | NO}	YES	YES の場合は、ネストしたプログラムをサポート
NLS_CHAR=(*var1*,..., *varn*)	*なし*	マルチバイト文字変数の指定します。
NLS_LOCAL={YES | NO}	NO	YES の場合は、Pro*COBOL の旧リリースの NCHAR 方法を使用します。
OBJECTS={YES | NO}	YES	オブジェクト・タイプのサポート
OUTLINE	NO	アウトラインが作成されるカテゴリ
OUTLNPREFIX	*なし*	アウトライン名の接頭辞
[ONAME=]*filename*		出力ファイルの名前
ORACA={YES|NO}*	NO	ORACA を使用するかどうか
ORECLEN=*integer*	80	出力ファイルのレコード長
PAGELEN=*integer*	66	リストの各ページの行数
PARSE={NONE | PARTIAL | FULL}	FULL	Pro*C/C++で(C パーサーを使用して).pc ソースが解析されるかどうか
PICN_ENDIAN	BIG	PIC N ホスト変数のエンディアン
PICX	CHARF	PIC X COBOL 変数のデータ型。
PREFETCH=0..65535	1	一定の行数を事前にフェッチすることで問合せの実行速度を向上させます。
RELEASE_CURSOR={YES|NO} *	NO	カーソル・キャッシュで SQL 文を処理する方法
RUNOUTLINE	NO	データベースにアウトラインを作成します。
SELECT_ERROR={YES|NO}*	YES	SELECT エラーの処理の方法

構文	デフォルト値	指定内容
SQLCHECK={FULL|SYNTAX|LIMITED|NONE}*	SYNTAX	構文および意味のチェックの程度
STMT_CACHE	0	文キャッシュのサイズ
SYS_INCLUDE=pathname	なし	iostream.h などのシステム・ヘッダー・ファイルがあるディレクトリ
THREADS={YES | NO}	NO	共有サーバーアプリケーションを指定します。
TYPE_CODE={ORACLE | ANSI}	ORACLE	動的 SQL での Oracle または ANSI 型コードの使用
UNSAFE_NULL={YES|NO}	NO	ORA-01405 メッセージを無効にします。
USERID=*username/password*		有効な Oracle ユーザー名およびパスワード
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}	NCHAR_CHARSET	UNICODE(UTF16)で使用される文字セットの形式を指定します。
VARCHAR={YES|NO}	NO	COBOL で暗黙的 VARCHAR グループ項目を認識します。
VERSION={ANY | LATEST | RECENT}*	RECENT	どのバージョンのオブジェクトを戻すか。
XREF={YES|NO}*	YES	リスト内のクロス・リファレンス・セクション

6.7 プリコンパイラ・オプションの使用方法について

この項は、プリコンパイラ・オプションを簡単に参照できるように構成されています。プリコンパイラ・オプションをアルファベット順に並べ、オプションごとに用途、構文およびデフォルト値を示しています。さらに「使用上の注意」で、オプションについて説明します。使用上のノートに特に記載がない場合、そのオプションはコマンドライン、インラインまたは構成ファイルのどの方法でも入力できます。

6.7.1 ASACC

用途

リスト・ファイルが、キャリッジ制御のために各行の最初の列を使用するASA規則に従うかどうかを指定します。

構文

ASACC={YES|NO}

デフォルト値

NO

使用上のノート

インラインでは入力できません。

6.7.2 ASSUME_SQLCODE

用途

宣言部で宣言されているかどうか、あるいは型が正しいかどうかに関係なく、SQLCODEが宣言されているとみなすようにOracleプリコンパイラに指示します。ASSUME_SQLCODE=YESと指定すると、リリース1.6以降のOracleプリコンパイラは、この点ではリリース1.5と同様の動作をします。

構文

ASSUME_SQLCODE={YES|NO}

デフォルト値

NO

使用上のノート

インラインでは入力できません。

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かぎり、SQLCODEは状態変数として認識されます。

- 宣言部で完全に正しいデータ型で宣言されている場合。
- プリコンパイラで他の状態変数が見つからない場合。

プリコンパイラが宣言部で(完全に正しい型の)SQLSTATE宣言を検出した場合、または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。

ASSUME_SQLCODE=YESと指定した場合、SQLSTATEおよびSQLCA(Pro*FORTRANのみ)が状態変数として宣言されると、宣言部で宣言されているかどうか、あるいは正しい型かどうかに関係なく、プリコンパイラではSQLCODEが宣言されているものとみなします。このため、リリース1.6.7以降は、この点でリリース1.5と同様の動作をします。

6.7.3 AUTO_CONNECT

用途

プログラムが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

使用上のノート

インラインでは入力できません。

AUTO_CONNECT=YESと指定した場合、実行SQL文を検出する同時に、プログラムでは自動的に次のユーザーIDでOracleに接続を試みます。

```
<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NULL)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、コマンドラインに別の値を指定しても、MAXOPENCURORS(10)のデフォルト値は変更できません。

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。

6.7.4 CHAR_MAP

用途

char型またはchar[n]型のCホスト変数およびそれらに対するポインタのSQLへのデフォルトのマッピングを指定します。

構文

CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF}

デフォルト値

CHARZ

使用上のノート

旧リリースでは、SQL DECLARE文を使用してCHARなどのcharまたはchar[n]ホスト変数を宣言する必要がありました。外部データ型VARCHAR2およびCHARZが、Oracle7のデフォルト文字マッピングでした。

6.7.5 CINCR

用途

データベースに対してオープンされる物理接続数の次の増分をアプリケーションで設定できるようにします。

構文

CINCR = 範囲は1から(CMAX-CMIN)。

デフォルト値

1

使用上のノート

最初は、CMINにより指定されたとおりに物理接続がすべてサーバーに対してオープンされます。それ以降は、必要な場合にのみ物理接続がオープンされます。パフォーマンスを最適にするには、CMINを、アプリケーションによる実行が計画または予想される同時実行文の合計数に設定する必要があります。デフォルト値は2に設定されます。

6.7.6 CLOSE_ON_COMMIT

用途

コミット文でカーソルを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

使用上のノート

コマンドラインまたは構成ファイルからのみ入力できます。

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。ユーザーがコマンドラインでMODE=ANSIを指定すると、コミット時にすべてのカーソルがクローズされます。

CLOSE_ON_COMMIT=NO (MODE=ORACLEの場合)の場合、COMMITまたはROLLBACKを発行しても、クローズされるのはFOR UPDATE句を使用して宣言されたカーソルまたはCURRENT OF句で参照されるカーソルのみです。その他のカーソルはCOMMITまたはROLLBACK文による影響を受けず、オープンされている場合はオープンされたままです。ただし、CLOSE_ON_COMMIT=YES (MODE=ANSIの場合)のときにCOMMITまたはROLLBACKを発行すると、すべてのカーソルがクローズします。

6.7.7 CMAX

用途

データベースに対してオープンできる物理接続の最大数を指定します。

構文

CINCR = 範囲は1から65535

デフォルト値

100

使用上のノート

CMAXの値は、CMIN+CINCR以上である必要があります。この値に達したら、それ以上物理接続をオープンすることはできません。通常のアプリケーションでは、100のデータベース操作を同時実行できれば十分です。ユーザーが適切な値を設定できます。

6.7.8 CMIN

用途

データベースに対してオープンできる物理接続の最小数を指定します。

構文

CINCR = 範囲は1から(CMAX-CINCR)。

デフォルト値

2

使用上のノート

CMAXの値は、CMIN+CINCR以上である必要があります。この値に達したら、それ以上物理接続をオープンすることはできません。通常のアプリケーションでは、100のデータベース操作を同時実行できれば十分です。ユーザーが適切な値を設定できます。

6.7.9 CNOWAIT

用途

この属性は、プール内の他のすべての物理接続が使用中で、物理接続の合計数がすでに最大値に達している場合に、アプリケーションで繰り返し物理接続を要求する必要があるかどうかを決定します。

構文

CNOWAIT = 範囲は1から65535。

デフォルト値

未設定を意味する0。

使用上のノート

物理接続が使用できず、これ以上物理接続をオープンできない場合、この属性が設定されているとエラーが発生します。そうでない場合、コールは別の接続が取得されるまで待機します。デフォルトでは、CNOWAITは設定されないため、スレッドは、エラーを戻すかわりに、空いている接続を取得できるまで待機します。

6.7.10 CODE

用途

Pro*C/C++プリコンパイラによって生成されるC関数プロトタイプの書式を指定します。(関数プロトタイプは、関数とその引数のデータ型を宣言します。)プリコンパイラは、Cコンパイラが外部参照を解決できるよう、SQLライブラリ・ルーチンのために関数プロトタイプを生成します。CODEオプションで、プロトタイピングを制御できます。

構文

CODE={ANSI_C | KR_C | CPP}

デフォルト値

KR_C

使用上のノート

コマンドラインでは入力できますが、インラインではできません。

ANSI C規格X3.159-1989は、関数プロトタイプを規定しています。CODE=ANSI_Cのとき、Pro*C/C++では、ANSI C規格に準拠する完全な関数プロトタイプが生成されます。次に例を示します。

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

プリコンパイラでは、他のANSI準拠の構造体(**const**型修飾子など)も生成できます。

CODE=KR_C(デフォルト)のとき、生成された関数プロトタイプの引数リストは、次のようなコメントになります。

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

CコンパイラがX3.159規格に準拠していなければ、CODE=KR_Cを指定します。

CODE=CPPのとき、プリコンパイラではC++互換コードが生成されます。

6.7.11 COMMON_NAME

用途

Pro*FORTRANの場合のみ、COMMON_NAMEオプションは、内部FORTRAN COMMONブロックの命名に使用される接頭辞を指定します。ホスト・プログラムは、COMMONブロックに直接アクセスしません。しかし、このブロックでは、同じプリコンパイル・ユニット内の2つ以上のプログラム・ユニットにSQL文を含めることができます。

構文

COMMON_NAME=*blockname*

デフォルト値

入力ファイル名の最初の5文字

使用上のノート

Pro*FORTRANプリコンパイラでは、1つの入力ファイル内のすべてのSQL変数用にCOMMONブロックを設定するブロック・データ・サブプログラムという専用プログラム・ファイルを使用します。ブロック・データ・サブプログラムによって、2つのCOMMONブロック(1つはCHARACTER変数用、もう1つはCHARACTER以外の変数用)が定義され、DATA文を使用して変数が初期化されます。

ブロック・データ・サブプログラムの書式は次のとおりです。

```
BLOCK DATA <subprogram_name>
variable declarations
COMMON statements
DATA statements
END
```

ホスト・プログラムは、COMMONブロックに直接アクセスしません。しかし、これらでは、同じプリコンパイル・ユニット内の2つ以上のプログラム・ファイルにSQL文を含めることができます。

COMMONブロックの命名に、プリコンパイラでは入力ファイル名と接尾辞C、DおよびIを使用します。ファイル名の最初の多くても5文字までが使用されます。たとえば、入力ファイル名がACCTSPAYの場合、COMMONブロックには、ACCTSC、ACCTSDおよびACCTSIという名前が付けられます。

しかし、異なる出力ファイルで定義されたCOMMONブロックには、次の図に示すように同じ名前を付けることができます。

```
ACCTSPAY.PFO ===> ACCTSC, ACCTSD, ACCTSI in ACCTSPAY.FOR
ACCTSREC.PFO ===> ACCTSC, ACCTSD, ACCTSI in ACCTSREC.FOR
```

ACCTSPAYとACCTSRECを1つの実行可能プログラムにリンクさせる場合、リンカーで見えるCOMMONブロックは、6つではなく3つです。

この問題を解決するには、入力ファイル名を変更するか、COMMON_NAMEをインラインまたはコマンドラインで次のように指定することで、デフォルトのCOMMONブロック名をオーバーライドします。

```
COMMON_NAME=<block_name>
```

*block_name*は、適切なCOMMONブロック名です。たとえば、COMMON_NAME=PAYを指定した場合、COMMONブロックにはPAYCおよびPAYIという名前が付けられます。*block_name*の最初の多くても5文字までが使用されます。

たとえば、COMMON_NAME=PAYを指定した場合、COMMONブロックにはPAYCおよびPAYIという名前が付けられます。*block_name*の最初の多くても5文字までが使用されます。

COMMON_NAMEをインラインで指定する場合は、そのEXEC ORACLE OPTION文を必ずFORTRAN PROGRAM、SUBROUTINEまたはFUNCTION文の前に置く必要があります。

デフォルトのCOMMONブロック名が、ユーザー定義のCOMMONブロック名と競合する場合、デフォルトの名前はオーバーライドできます。ただし、ユーザー定義のCOMMONブロックの名前を変更することをお薦めします。

MULTISUBPROGを指定する場合、COMMON_NAMEは不要です。

6.7.12 COMMON_PARSER

用途

SQL99構文のSELECT、INSERT、DELETE、UPDATEおよびDECLARE CURSOR文のカーソル本体がサポートされます。

構文

COMMON_PARSER={YES | NO}

デフォルト値

NO

使用上のノート

コマンドラインで入力できます。

6.7.13 COMP_CHARSET

用途

使用するコンパイラでマルチバイト文字セットがサポートされているかどうかを、Pro*C/C++プリコンパイラに指定します。これは、マルチバイトのクライアント環境(たとえば、NLS_LANGがマルチバイト文字セットに設定されているとき)で作業する開発者向けです。

構文

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

デフォルト値

MULTI_BYTE

使用上のノート

コマンドラインでのみ入力できます。

COMP_CHARSET=MULTI_BYTE(デフォルト)が指定されると、Pro*C/C++では、マルチバイト文字セットをサポートしているコンパイラによりコンパイルされるCコードを生成します。

COMP_CHARSET=SINGLE_BYTEを指定すると、Pro*C/C++では、シングルバイト系のコンパイラ用のCコードが生成され、マルチバイト文字列中のダブルバイト文字の2バイト目がバックスラッシュ(¥)に対応するASCII文字である場合に起きる可能性のある面倒な問題に、生成されたCコードにより対処します。この場合、バックスラッシュ(¥)は、前にもう1つバックスラッシュを置くことでエスケープされます。

ノート:

この機能は、一般的に、古い C コンパイラを使用してシフト JIS 環境で開発をするときに必要になります。

シングルバイト文字セットにNLS_LANGが設定されていると、このオプションは効果がありません。

6.7.14 COMP_CHARSET

用途

使用するコンパイラでマルチバイト文字セットがサポートされているかどうかを、Pro*C/C++プリコンパイラに指定します。これは、マルチバイトのクライアント環境(たとえば、NLS_LANGがマルチバイト文字セットに設定されているとき)で作業する開発者向けです。

構文

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

デフォルト値

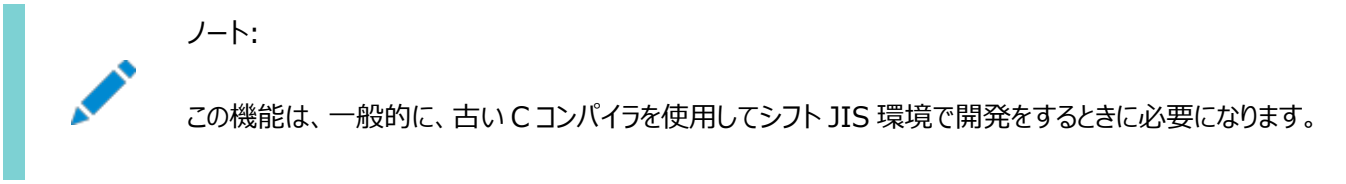
MULTI_BYTE

使用上のノート

コマンドラインでのみ入力できます。

COMP_CHARSET=MULTI_BYTE(デフォルト)が指定されると、Pro*C/C++では、マルチバイト文字セットをサポートしているコンパイラによりコンパイルされるCコードを生成します。

COMP_CHARSET=SINGLE_BYTEを指定すると、Pro*C/C++では、シングルバイト系のコンパイラ用のCコードが生成され、マルチバイト文字列中のダブルバイト文字の2バイト目がバックスラッシュ(¥)に対応するASCII文字である場合に起きる可能性のある面倒な問題に、生成されたCコードにより対処します。この場合、バックスラッシュ(¥)は、前にもう1つバックスラッシュを置くことでエスケープされます。



ノート:

この機能は、一般的に、古い C コンパイラを使用してシフト JIS 環境で開発をするときに必要になります。

シングルバイト文字セットにNLS_LANGが設定されていると、このオプションは効果がありません。

6.7.15 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=*file*

デフォルト値

None

使用上のノート

コマンドラインでのみ入力できます。

Oracleプリコンパイラでは、コマンドライン・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。デフォルトでは、システム構成ファイルと呼ばれるテキスト・ファイルが使用されます。ただし、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を指定できます。

構成ファイルはネストできません。したがて、構成ファイルではCONFIGオプションを指定できません。

6.7.16 CPOOL

用途

このオプションに基づき、プリコンパイラでは、SQLLIBに接続プール機能を有効または無効にするように指示する適切なコードを生成します。

構文

CPOOL = {YES|NO}

デフォルト値

NO

使用上のノート

このオプションがNOに設定されている場合、プリコンパイラではその他の接続プーリング・オプションを無視します。

6.7.17 CPP_SUFFIX

用途

CPP_SUFFIXオプションを使用すると、CODE=CPPオプションを指定した場合に生成されるC++出力ファイルに、プリコンパイラによって付けられるファイル拡張子を指定できます。

構文

CPP_SUFFIX=*filename_extension*

デフォルト値

システム固有。

使用上のノート

ほとんどのCコンパイラでは、入力ファイルのデフォルト拡張子は.cになります。しかし、C++コンパイラでは、ファイル名の拡張子がコンパイラごとに異なる場合があります。CPP_SUFFIXオプションを指定すると、プリコンパイラで生成されるファイル名拡張子を指定できます。このオプションの値は、引用符もピリオドも付けない文字列です。たとえば、CPP_SUFFIX=ccまたはCPP_SUFFIX=Cのように指定します。

6.7.18 CTIMEOUT

用途

指定した時間(秒単位)より長い間アイドル状態になっている物理接続を終了し、オープンされている物理接続を最適な数に保ちます。

構文

CTIMEOUT = 範囲は1から65535。

デフォルト値

未設定を意味する0。

使用上のノート

接続プールが終了されるまで、物理接続はクローズされません。新しい物理接続を作成すると、サーバーへのラウンド・トリップが必要になります。

6.7.19 DB2_ARRAY

用途

このオプションに基づいて、プリコンパイラは追加の配列INSERTおよび配列SELECT構文をアクティブにします。

構文

DB2_ARRAY={YES |NO}

デフォルト値

NO

使用上のノート

このオプションをNOに設定すると、Oracleプリコンパイラの構文がサポートされます。それ以外の場合は、DB2の配列INSERTおよび配列SELECT構文がサポートされます。

6.7.20 DBMS

用途

OracleがOracle9*i*、Oracle8*i*、Oracle8、Oracle7、あるいはOracleのネイティブ・バージョン(つまり、アプリケーションが接続しているバージョン)のうち、どの意味上および構文上の規則に従うかを指定します。

構文

DBMS=NATIVE|V7|V8

デフォルト値

NATIVE

使用上のノート

インラインでは入力できません。

DBMSオプションを使用すると、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・バージョンの意味および構文上の規則に従います。

DBMS=V8またはDBMS=V7のときは、OracleはそれぞれOracle9*i*の規則(Oracle7、Oracle8およびOracle8*i*の場合と同じ規則)に従います。

表6-4は、互換性のあるDBMSとMODEの設定がどのように相互に作用するかを示しています。その他の組合せはすべて互換性がなく、推奨できません。

表6-4 互換性のあるDBMSおよびMODE設定

状況	DBMS=V7/V8 MODE=ANSI	DBMS=V7/V8 MODE=ORACLE
「データが見つかりません」という警告コード	+100	+1403
インジケータ変数を使用しない NULL のフェッチ	エラー-1405	エラー-1405
インジケータ変数を使用しない切捨て値のフェッチ	エラーなしで SQLWARN(2) 設定	エラーなしで SQLWARN(2)設定
COMMIT または ROLLBACK によるカーソルのクローズ	すべて明示的	CURRENT OF のみ
すでにオープンしているカーソルのオープン	エラー-2117	エラーなし

状況	DBMS=V7/V8 MODE=ANSI	DBMS=V7/V8 MODE=ORACLE
すでにクローズしているカーソルのクローズ	エラー-2114	エラーなし
SQL グループ関数による NULL の無視	警告なし	警告なし
複数行の問合せで SQL グループ関数をコールするとき	FETCH 時	FETCH 時
SQLCA 構造体の宣言	オプション	必須
SQLCODE または SQLSTATE 状態変数の宣言	必須	指定できるが Oracle では無視
TYPE 文および VAR 文で CHAR に使用される文字ホスト変数のデフォルト外部データ型	CHARF	VARCHAR2
SQL 文における文字列リテラルのデフォルトの外部データ型	CHARF	CHARF
SQL 文における CHAR 変数のデフォルトの内部データ型	CHAR	CHAR
PL/SQL ブロックにおける CHAR 変数のデフォルトの外部データ型	CHARF	CHARF
USER ファンクションから戻される値のデフォルトの外部データ型	CHARF	CHARF
DESCRIBE コードから戻される外部データ型(動的 SQL 方法 4)	96	96
整合性制約	有効	有効
ロールバック・セグメント用の PCTINCREASE	使用不可	使用不可
MAXEXTENTS 記憶域パラメータ	使用不可	使用不可

6.7.21 DEF_SQLCODE

用途

Pro*C/C++プリコンパイラによりSQLCODEの**#define**が生成されるかどうかを制御します。

構文

DEF_SQLCODE={NO | YES}

デフォルト値

NO

使用上のノート

コマンドラインまたは構成ファイルからのみ入力できます。

DEF_SQLCODE=YESの場合、プリコンパイラでは生成されるソース・コードでSQLCODEが次のように定義されます。

```
#define SQLCODE sqlca.sqlcode
```

この定義があれば、SQLCODEを使用して実行SQL文の結果をチェックできます。DEF_SQLCODEオプションは、SQLCODEの使用が必要な規格への準拠のために指定します。

また、次のいずれかを入力して、ソース・コードにSQLCAも組み込む必要があります。

```
#include <sqlca.h>
```

または

```
EXEC SQL INCLUDE SQLCA;
```

SQLCAを組み込まなければ、このオプションを使用するとプリコンパイル時にエラーが発生します。

6.7.22 DEFINE

用途

条件付きのプリコンパイル時にソース・コードの一部の挿入または除外を行うために使用する、ユーザー定義の記号を指定します。

構文

DEFINE=*symbol*

デフォルト値

None

使用上のノート

DEFINEをインラインで入力する場合、EXEC ORACLE文の書式は次のとおりです。

```
EXEC ORACLE DEFINE <symbol>;
```

6.7.23 DURATION

用途

後続のEXEC SQL OBJECT CREATE文とEXEC SQL OBJECT DEREF文に使用される確保継続時間を設定します。キャッシュ内のオブジェクトは、保持期間の終わりに暗黙的に解放されます。

構文

DURATION={TRANSACTION | SESSION}

デフォルト値

TRANSACTION

使用上のノート

EXEC ORACLE OPTION文を使用してインライン入力できます。

TRANSACTIONは、オブジェクトがトランザクションの完了時に暗黙的に解放されることを意味します。

SESSIONは、オブジェクトが接続の終了時に暗黙的に解放されることを意味します。

6.7.24 DYNAMIC

用途

このマイクロ・オプションでは、動的SQL方法4の記述子の動作を指定します。MODEの設定によりDYNAMICの設定が決まります。

構文

DYNAMIC={ORACLE | ANSI}

デフォルト値

ORACLE

使用上のノート

EXEC ORACLE OPTION文を使用してインラインで入力することはできません。

6.7.25 ERRORS

用途

エラー・メッセージを端末とリスト・ファイルの両方に送信するか、リスト・ファイルにのみ送信するかを指定します。

構文

ERRORS={YES|NO}

デフォルト値

YES

使用上のノート

ERRORS=YESの場合、エラー・メッセージは端末とリスト・ファイルの両方に送信されます。

ERRORS=NOの場合、エラー・メッセージはリスト・ファイルにのみ送信されます。

6.7.26 ERRTYPE

用途

型ファイルの処理中に生成されたエラーを書き込む出力ファイルを指定します。省略すると、エラーは画面に出力されます。

構文

ERRTYPE=*filename*

デフォルト値

なし

使用上のノート

生成されるエラー・ファイルは1つのみです。複数の値を入力すると、最後の値がプリコンパイラで使用されます。

6.7.27 EVENTS

用途

アプリケーションが通知の登録および受信に対応しているかどうかを指定します。

構文

EVENTS={YES | NO}

デフォルト値

NO

使用上のノート

コマンドラインからのみ入力できます。

6.7.28 FIPS

用途

ANSI/ISO SQLの拡張機能に(FIPSフラガーで)フラグを立てるかどうかを指定します。拡張機能とは、ANSI/ISOの書式または構文規則(権限適用規則以外)に違反するSQL要素を指します。

構文

FIPS={YES|NO}

デフォルト値

NO

使用上のノート

FIPS=YESの場合は、ANSI/ISOに組み込まれたSQL規格(SQL92)からのOracle拡張機能を使用したり、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)。

次のANSI/ISO SQL拡張機能には、プリコンパイル時にフラグが立てられます。

- FOR句を含む配列インタフェース
- SQLCA、ORACAおよびSQLDAデータ構造体
- DESCRIBE文を含む動的SQL
- 埋込みPL/SQLブロック
- 自動データ型変換
- DATE、COMP-3(Pro*COBOLのみ)、NUMBER、RAW、LONG RAW、VARRAW、ROWIDおよびVARCHARデータ型
- ランタイム・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、...STATEMENT文および...TABLE文
- WHENEVER文でのSQLWARNING条件
- WHENEVER文でのDOおよびSTOPアクション
- COMMIT文での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

6.7.29 FORMAT

用途

COBOLまたはFORTRANで入力する行の書式を指定します。

構文

FORMAT={ANSI|TERMINAL}

デフォルト値

ANSI

使用上のノート

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。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。

FORMAT=ANSIの場合、入力行の書式は、できるかぎり現行のANSI規格に準拠します。

6.7.30 Globalization Support_LOCAL

用途

Pro*COBOLの場合のみ、Globalization Support_LOCALオプションにより、グローバリゼーション・サポートの文字変換が、プリコンパイラのランタイム・ライブラリまたはOracleサーバーにより実行されます。

構文

Globalization Support_LOCAL={YES|NO}

デフォルト値

NO

使用上のノート

インラインでは入力できません。

Globalization Support_LOCAL=YESの場合、ランタイム・ライブラリ(SQLLIB)により、マルチバイトのグローバリゼーション・サポート・データ型を持つホスト変数に対する空白の埋込みおよび削除がローカルに実行されます。

Globalization Support_LOCAL=NOの場合、マルチバイトのグローバリゼーション・サポート・データ型を持つホスト変数に対する空白の埋込みおよび削除の処理は、ローカルに実行されません。

Oracleでは、グローバリゼーション・サポート変数の空白の埋込みまたは削除は実行されません。Globalization Support_LOCAL=NOの場合、マルチバイトのグローバリゼーション・サポート・データを使用するSQL文を実行すると、Oracleサーバーからはエラーが戻されます。

6.7.31 HEADER

用途

プリコンパイル済ヘッダー・ファイルを許可します。プリコンパイル済ヘッダー・ファイルのファイル拡張子を指定します。

構文

HEADER=*extension*

デフォルト値

NONE

使用上のノート

ヘッダー・ファイルをプリコンパイルする場合、このオプションは必須で、ヘッダー・ファイルのプリコンパイルによって生成される出力ファイルのファイル拡張子の指定に使用されます。

通常のPro*C/C++プログラムをプリコンパイルする場合、このオプションは任意です。指定すると、Pro*C/C++プログラムのプリコンパイル時に、プリコンパイル済ヘッダーのメカニズムを使用できます。

どちらの場合も、このオプションで#includeディレクティブの処理時に使用するファイル拡張子も指定できます。指定した拡張子の付いた#includeファイルが存在する場合、Pro*C/C++ではそのファイルをPro*C/C++によって以前に生成されたプリコンパイル済ヘッダー・ファイルとみなします。Pro*C/C++は、#includeディレクティブを処理してインクルードされるヘッダー・ファイルをプリコンパイルするかわりに、そのファイルからデータをインスタンス化します。

このオプションは、コマンドラインまたは構成ファイルでのみ使用できます。インラインでは使用できません。このオプションを使用する場合は、ファイル拡張子のみを指定します。ファイル・セパレータは含めないでください。たとえば、拡張子にピリオド(.)は含めないでください。

6.7.32 HOLD_CURSOR

用途

カーソル・キャッシュでのSQL文およびPL/SQLブロック用カーソルの処理方法を指定します。

構文

HOLD_CURSOR={YES|NO}

デフォルト値

NO

使用上のノート

HOLD_CURSORを使用すると、プログラムのパフォーマンスを改善できます。詳細は、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、その文に関連付けられたカーソルが、カーソル・キャッシュ内のエントリにリンクされます。そのカーソル・キャッシュ・エントリは、文の処理に必要な情報が格納されるOracleプライベートSQL領域にリンクされます。HOLD_CURSORは、カーソルとカーソル・キャッシュの間のリンクで発生する処理を制御します。

HOLD_CURSOR=NOの場合、OracleでSQL文が実行され、カーソルがクローズされた後に、プリコンパイラではそのリンクに再利用可能のマークを付けます。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。

HOLD_CURSOR=YESで、RELEASE_CURSOR=NOの場合、リンクは維持されます。プリコンパイラはそれを再利用しません。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。文の再解析やOracleプライベートSQL領域用のメモリー割当ては不要です。

暗黙カーソルとともにインラインで使用する場合、SQL文の実行前にHOLD_CURSORを設定してください。明示カーソルとともにインラインで使用する場合は、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を指定するとHOLD_CURSOR=YESがオーバーライドされ、HOLD_CURSOR=NOを指定するとRELEASE_CURSOR=NOがオーバーライドされます。これら2つのオプションの相互作用の詳細は、表C-1を参照してください。

6.7.33 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

使用上のノート

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。

6.7.34 IMPLICIT_SVPT

用途

新しいバッチ挿入の開始前に、暗黙的セーブポイントを設定するかどうかを制御します。

構文

implicit_svpt={YES|NO}

デフォルト値

NO

使用上のノート

implict_svpt=yesの場合、新しい行のバッチを開始する前に、セーブポイントが設定されます。挿入時にエラーが発生すると、暗黙的な「セーブポイントへのロールバック」が実行されます。このオプションはDB/2との互換性のためのもので、余分なラウンドトリップが必要なことから、明らかにマイナスです。

implict_svpt=noの場合、暗黙的セーブポイントは設定されません。バッファ済INSERTでエラーが発生すると、アプリケーションに通知されますが、ロールバックは実行されません。

6.7.35 INAME

用途

入力ファイルの名前を指定します。

構文

INAME=*file name*

デフォルト値

None

使用上のノート

インラインでは入力できません。

コマンドラインで入力ファイルの名前を指定する場合、キーワードの**INAME**は省略可能です。たとえば、Pro*COBOLでは、INAME=myprog.pcoのかわりに、*myprog.pco*と指定できます。

プリコンパイラでは、標準の入力ファイル拡張子(表6-5を参照)とみなされます。したがって、拡張子が非標準の場合を除き、INAMEの指定にファイル拡張子を使用する必要はありません。たとえば、Pro*FORTRANでは、*myprog.pfo*のかわりに、*myprog*と指定できます。

表6-5 入力ファイルの拡張子

ホスト言語	標準のファイル拡張子
COBOL	pco
FORTRAN	pfo

Pro*COBOLの場合のみ、INAMEを指定する際に非標準の入力ファイル拡張子を使用すると、HOSTも指定する必要があります。

6.7.36 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。これは、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

現在のディレクトリ

使用上のノート

通常、INCLUDEは、SQLCAファイルおよびORACAファイルのディレクトリ・パスの指定に使用します。プリコンパイラでは、最初にカレント・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、最後に標準の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。

標準以外のファイルについては、現在のディレクトリに格納されている場合を除いて、INCLUDEを使用してディレクトリ・パスを指定する必要があります。次に示すように、コマンドラインに複数のパスを指定できます。

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

プリコンパイラでは、最初に現在のディレクトリを検索し、次に*path1*で指定したディレクトリを検索し、続いて*path2*で指定した

ディレクトリを検索して、最後に標準のINCLUDEファイル用のディレクトリを検索します。

ディレクトリ・パスを指定しても、プリコンパイラでは最初に現在のディレクトリでファイルを検索します。このため、INCLUDEするファイルが別のディレクトリにある場合は、同じ名前のファイルが現在のディレクトリに存在し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。

6.7.37 IRECLEN

用途

入力ファイルのレコード長を指定します。

構文

IRECLEN=*integer*

デフォルト値

80

使用上のノート

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。

6.7.38 INTYPE

用途

OTTで生成された型のファイルを1つ以上指定します(アプリケーションでオブジェクト型が使用される場合にのみ必要です)。

構文

INTYPE=(*file_1*,*file_2*,...,*file_n*)

デフォルト値

なし

使用上のノート

Pro*C/C++コードには、オブジェクト型ごとに1つの型のファイルが存在します。

6.7.39 LINES

用途

Pro*C/C++プリコンパイラでその出力ファイルに**#line**プリプロセッサ・ディレクティブが追加されるかどうかを指定します。

構文

LINES={YES | NO}

デフォルト値

NO

使用上のノート

コマンドラインでのみ入力できます。

LINESオプションはデバッグに便利です。

LINES=YESの場合、Pro*C/C++プリコンパイラではその出力ファイルに**#line**プリプロセッサ・ディレクティブを追加します。

通常、Cコンパイラでは、それぞれの入力行が処理されるたびに行カウントを増やします。**#line**ディレクティブは、コンパイラの入力行カウントを強制的にリセットして、プリコンパイラで生成されたコードの行を数えないようにします。さらに、入力ファイルの名前が変わったときに、次の**#line**ディレクティブが新しいファイル名を指定します。

Cコンパイラでは、行番号とファイル名を使用して、エラーの発生場所を示します。したがって、Cコンパイラで発行されるエラー・メッセージは、変更済(プリコンパイル済)のソース・ファイルではなく、常に元のソース・ファイルを参照します。これにより、ほとんどのデバッガを使用して、元のソース・コードを1ステップずつ実行することもできます。

LINES=NO(デフォルト)の場合、プリコンパイラでは出力ファイルに**#line**ディレクティブは追加されません。

ノート:

Pro*C/C++プリコンパイラでは、**#line** ディレクティブはサポートされません。つまり、プリコンパイラ・ソースでは、**#line** ディレクティブを直接コーディングできません。ただし、LINES=オプションを使用すると、プリコンパイラに **#line** ディレクティブを挿入させることができます。

6.7.40 LITDELIM

用途

Pro*COBOL専用のLITDELIMオプションは、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

使用上のノート

LITDELIM=APOSTの場合、プリコンパイラではCOBOLコードを生成するときにアポストロフィ(')が使用されます。
LITDELIM=QUOTEを指定すると、次のように二重引用符(")が使用されます。

```
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む識別子は二重引用符で区切る必要があります。

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

また、文字列定数を区切る場合は、次の例のように引用符を使用します。

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

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、プリコンパイラではLITDELIM値で指定したデリミタを生成します。

6.7.41 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底名)

使用上のノート

インラインでは入力できません。

デフォルトでは、リスト・ファイルはカレント・ディレクトリに作成されます。

6.7.42 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

使用上のノート

インラインでは入力できません。

LRECLENの値の範囲は80から255です。80未満の値を指定した場合は、かわりに80が使用されます。範囲を超える値を指定すると、かわりに255が使用されます。行番号を挿入できるように、LRECLENの値がIRECLENより8以上大きくなるように指定してください。

6.7.43 LTYPE

用途

リストのタイプを指定します。

構文

LTYPE={LONG|SHORT|NONE}

デフォルト値

LONG

使用上のノート

インラインでは入力できません。

LTYPE=LONGの場合、入力行はリスト・ファイルに表示されます。LTYPE=SHORTの場合、入力行はリスト・ファイルに表示されません。LTYPE=NONEの場合、リストは作成されません。

6.7.44 MAXLITERAL

用途

コンパイラの制限を超えないように、プリコンパイラで生成される文字列リテラルの最大長を指定します。たとえば、コンパイラで132文字より長い文字列リテラルを処理できない場合は、コマンドラインにMAXLITERAL=132と指定します。

構文

MAXLITERAL=*integer*

デフォルト値

デフォルト値は、次に示すように、プリコンパイラにより異なります。

プリコンパイラ	デフォルト値
Pro*COBOL	256
Pro*FORTRAN	1000

使用上のノート

MAXLITERALの最大値は、コンパイラによって異なります。デフォルト値は言語によって異なりますが、このデフォルト値より小さい値を指定する必要がある場合があります。たとえば、一部のCOBOLコンパイラでは、132文字より長い文字列リテラルを処理できないため、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。

インラインでMAXLITERALを入力することはできますが、プログラムで値を設定できるのは1回のみで、EXEC ORACLE文を最初のEXEC SQL文の前に指定する必要があります。指定しない場合、プリコンパイラは警告メッセージを発行し、余分または誤って指定したEXEC ORACLE文を無視して、処理を続行します。

6.7.45 MAXOPENCURSORS

用途

同時にオープンされ、プリコンパイラによりキャッシュに保存されたままになるカーソルの数を指定します。

構文

MAXOPENCURSORS=*integer*

デフォルト値

10

使用上のノート

MAXOPENCURSORSを使用すると、プログラムのパフォーマンスを改善できます。詳細は、パフォーマンス・チューニングを参照してください。

個別にプリコンパイルする場合は、「分割プリコンパイル」の説明のように、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、空いているキャッシュ・エントリがない場合、Oracleではエントリの再利用が試みられます。それが成功するどうかは、HOLD_CURSOR値およびRELEASE_CURSOR値によって決まり、明示カーソルの場合は、カーソル自体の状態によって決まります。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。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。「最大オープン・カーソル数を超えました」というOracleエラーを避けるには、MAXOPENCURSORSにOPEN_CURSORSより6以上小さい値を指定してください。

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を再指定する必要がある場合もあります。45から50の値を指定することは珍しくありませんが、ユーザー・プロセスのメモリー領域にカーソル1つにつき、1つのプライベートSQL領域が必要なことに注意してください。デフォルト値の10は、大半のプログラムには適切な値です。

6.7.46 MAX_ROW_INSERT

用途

INSERT文の実行前にバッファする必要がある行の数を制御します。

構文

max_row_insert={バッファに格納される行数}

デフォルト値

0

使用上のノート

0よりも大きい任意の値を指定すると、バッファ済INSERT機能が有効化され、INSERT文の実行前に多くの行がバッファされます。

6.7.47 MODE

用途

プログラムがOracleの動作規則に従うか、現行のANSI SQL規格に準拠するかどうかを指定します。

構文

MODE={ANSI|ISO|ANSI14|ISO14|ANSI13|ISO13|ORACLE}

デフォルト値

ORACLE

使用上のノート

インラインでは入力できません。

MODE値のANSIとISO、ANSI14とISO14、ANSI13とISO13は、それぞれ等価です。

MODE=ORACLE (デフォルト)の場合、埋込みSQLプログラムはOracleの動作規則に従います。

MODE={ANSI14|ANSI13}の場合、プログラムは現行のANSI SQL規格にほぼ準拠します。

MODE=ANSIの場合、プログラムは完全にANSI規格に準拠し、次のような変更が有効になります。

- CHAR列値、USER擬似列値、文字ホスト変数および引用符付きリテラルは、ANSI固定長文字列と同じように扱われます。そして、そのような値について割当て、比較、INSERT、UPDATE、SELECTまたはFETCHを実行するときに、ANSI準拠の空白埋込みが使用されます。
- COMMITまたはROLLBACKを発行すると、すべての明示カーソルがクローズされます。(MODE={ANSI13|ORACLE}の場合は、コミットまたはロールバックにより、CURRENT OF句で参照されるカーソルのみがクローズされます。)
- すでにオープンされているカーソルのOPENや、すでにクローズされているカーソルのCLOSEはできません。(MODE=ORACLEの場合は、再解析を避けるために、オープン状態のカーソルを再度OPENできます。)
- SQLCODEに戻される「データが見つかりません」というOracle警告コードは、+1403から+100になります。エラー・メッセージのテキストは変わりません。

- 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}の場合、SQLCODE(FORTRANのSQLCOD)という4バイトの整数変数、またはSQLSTATE(FORTRANのSQLSTA)という5バイトの文字変数を宣言する必要があります。詳細は、エラー処理の代替手段を参照してください。

表6-4では、MODE設定とDBMS設定の相互作用を示しています。その他の組合せは互換性がないか、お薦めできません。

6.7.48 MULTISUBPROG

用途

Pro*FORTRANの場合のみ、MULTISUBPROGオプションは、Pro*FORTRANプリコンパイラでCOMMON文およびBLOCK DATAサブプログラムを生成するかどうかを指定します。

> ノート:
>
> このオプションを使用すると、Pro*FORTRAN リリース 1.3 アプリケーションをその後のリリースに移行できます。Pro*FORTRAN リリース 1.3 ソース・コードを移行しない場合は、MUTISUBPROG オプションを無視してもかまいません。

構文

MULTISUBPROG={YES|NO}

デフォルト値

YES

使用上のノート

インラインでは入力できません。

MULTISUBPROG=YESの場合、プリコンパイラではCOMMON文およびBLOCK DATAサブプログラムが生成されます。ホスト・プログラムはCOMMONブロックに直接アクセスしませんが、同じプリコンパイル・ユニット内の複数のプログラム・ユニットにSQL文を含めることができます。

ただし、プリコンパイラでは、異なる出力ファイルで定義されているCOMMONブロックに、同じ名前を付けることができます。それらのファイルを実行可能プログラムにリンクさせると、リンク時または実行時にエラーが発生します。この問題を解決するには、入力ファイルの名前を変更するか、COMMON_NAMEオプションを指定してデフォルトのCOMMONブロック名をオーバーライドします。問題を避けるには、MULTISUBPROG=NOを指定してください。

Pro*FORTRANソース・コードの各ソース・ファイルにサブプログラムが1つしかない場合(これはリリース1.3での制限でした)は、MULTISUBPROG=NOを指定します。MULTISUBPROG=NOの場合、COMMON_BLOCKオプションは無視され、プリコンパイラではCOMMON文またはBLOCK DATAサブプログラムは生成されません。実行可能な複数のSQL文を含むプログラム・ユニットにはすべて、宣言部が必要です。それがないと、プリコンパイル・エラーが発生します。複数の埋込みSQLプログラム・ユニットが含まれる入力ファイルの場合、プリコンパイラではそれぞれのユニットで同じ宣言が生成されます。

6.7.49 NATIVE_TYPES

用途

ネイティブfloat/doubleをサポートします。

構文

NATIVE_TYPES = {YES|NO}

デフォルト値

NO

使用上のノート

ネイティブfloatおよびネイティブdoubleデータ型は、単精度と倍精度の浮動小数点値を表します。これらはネイティブ、つまりホスト・システムの浮動小数点形式で表されます。

6.7.50 NLS_CHAR

用途

プリコンパイラでマルチバイト文字変数として扱われるCホスト文字変数を指定します。

構文

NLS_CHAR=*varname*またはNLS_CHAR=(*var_1*,*var_2*,...,*var_n*)

デフォルト値

なし。

使用上のノート

コマンドラインまたは構成ファイルでのみ入力できます。

このオプションを使用すると、プリコンパイラでマルチバイト文字変数として扱う必要のある1つ以上のホスト変数のリストを、プリコンパイル時に指定できます。このオプションでは、C言語の*char*変数またはPro*C/C++のVARCHAR変数のみを指定できます。

オプション・リストにプログラムで宣言していない変数を指定しても、プリコンパイラのエラーは発生しません。

6.7.51 NLS_LOCAL

用途

プリコンパイラのSQLLIBランタイム・ライブラリとデータベース・サーバーのうち、どちらでマルチバイト文字セット変換が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

使用上のノート

YESに設定すると、Pro*C/C++およびSQLLIBライブラリによって、ローカル・マルチバイト・サポートが提供されます。どのCホスト変数がマルチバイトかを指定するには、NLS_CHARオプションを使用する必要があります。

NOに設定すると、Pro*C/C++では、データベース・サーバーのマルチバイト・オブジェクトのサポートを使用します。新規アプリケーションにはすべて、NLS_LOCALをNOに設定してください。

環境変数NLS_NCHARには、有効な固定幅の各国語文字セットを設定する必要があります。可変長幅の各国語文字セッ

トはサポートされていません。

コマンドラインまたは構成ファイルでのみ入力できます。

6.7.52 OBJECTS

用途

オブジェクト型のサポートを要求します。

構文

OBJECTS={YES | NO}

デフォルト値

YES

使用上のノート

コマンドラインからのみ入力できます。

6.7.53 ONAME

用途

出力ファイル名を指定します。

構文

ONAME=*fi/ename*

デフォルト値

System-dependent

使用上のノート

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
```

デフォルトの出力ファイル名は*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を使用し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。

出力ファイル名にはデフォルトの名前を使用するのではなく、ONAMEで明示的に名前を指定することをお薦めします。

6.7.54 ORACA

用途

プログラムでOracle通信領域(ORACA)を使用できるかどうかを指定します。

構文

ORACA={YES|NO}

デフォルト値

NO

使用上のノート

ORACA=YESの場合、プログラムにINCLUDE ORACA文を記述する必要があります。

6.7.55 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

使用上のノート

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、それより大きい値にする必要があります。指定可能な最大値はシステムによって異なります。

6.7.56 OUTLINE

用途

SQL文用にアウトラインSQLファイルを生成する必要があることを指定します。

構文

outline={yes | no | category_name}

デフォルト値

なし

使用上のノート

値がyesの場合、アウトラインSQLファイルはDEFAULTカテゴリに含まれている必要があります。生成されるアウトライン書式は次のとおりです。

```
DEFAULT_<filename>_<filetype>_<sequence_no>
```

カテゴリ名が示されている場合は、そのカテゴリにSQLファイルを生成する必要があります。この場合、生成されるアウトライン書式は次のようになります。

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

値がnoの場合、アウトラインSQLファイルは生成されません。

このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneの場合は、エラーが生成されます。

6.7.57 OUTLNPREFIX

用途

アウトライン名の生成を制御します。

構文

outlnprefix={none | prefix_name}

デフォルト値

なし

使用上のノート

outlnprefix=prefix_nameの場合、アウトライン書式は次のとおりです。

```
<category_name>_<filename>_<filetype>
```

この書式は、アウトライン名の<prefix_name>に置き換えられます。

アウトライン名が30バイトを超える場合、このオプションは接頭辞名を指定する際に有効です。

outlnprefix=noneの場合、アウトライン名はシステムによって生成されます。次の書式で生成されます。

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneまたはoutline=false(あるいはその両方)の場合、エラーが生成されます。

6.7.58 PAGELEN

用途

リスト・ファイルの1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

使用上のノート

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。

6.7.59 PARSE

用途

Pro*C/C++プリコンパイラでソース・ファイルを解析する方法を指定します。

構文

PARSE={FULL | PARTIAL | NONE}

デフォルト値

FULL

使用上のノート

C++互換コードを生成するには、PARSEオプションにNONEまたはPARTIALのいずれかを指定する必要があります。

PARSE=NONEまたはPARSE=PARTIALの場合、すべてのホスト変数は宣言部内で宣言する必要があります。

変数SQLCODEを宣言部の内側で宣言しない場合、エラー検出の信頼性がなくなります。使用しているプラットフォームのPARSEのデフォルト値をチェックしてください。

PARSE=FULLの場合、Cパーサーが動作し、コード内のクラスなどのC++構造体は認識されません。

PARSE=FULLまたはPARSE=PARTIALを指定すると、Pro*C/C++は**#define**、**#ifdef**などのCプリプロセッサ・ディレクティブを完全にサポートします。ただし、PARSE=NONEを指定すると、EXEC ORACLE文により条件付きプリプロセッシングがサポートされます。

ノート:

一部のプラットフォームでは、PARSE のデフォルト値が FULL 以外の値です。使用するシステム固有のマニュアルを参照してください。

6.7.60 PREFETCH

用途

数行を事前にフェッチすることで、問合せの実行速度を向上させます。

構文

PREFETCH=*integer*

デフォルト値

1

使用上のノート

構成ファイルまたはコマンドラインで入力できます。優先順位の規則に従い、明示カーソルを使用するすべての問合せの実行に、整数の値が使用されます。

インラインで使用する場合、明示カーソルのあるOPEN文の前に置く必要があります。次に、OPENが実行されるときに事前にフェッチされる行数は、有効な最後のインラインPREFETCHオプションによって決まります。

指定可能な値の範囲は0から65535です。

6.7.61 RELEASE_CURSOR

用途

カーソル・キャッシュでのSQL文およびPL/SQLブロック用カーソルの処理方法を指定します。

構文

RELEASE_CURSOR={YES|NO}

デフォルト値

NO

使用上のノート

RELEASE_CURSORを使用すると、プログラムのパフォーマンスを改善できます。詳細は、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、その文に関連付けられたカーソルが、カーソル・キャッシュ内のエントリにリンクされます。そのカーソル・キャッシュ・エントリは、文の処理に必要な情報が格納されるOracleプライベートSQL領域にリンクされます。RELEASE_CURSORは、カーソル・キャッシュとプライベートSQL領域の間のリンクで発生する処理を制御します。

RELEASE_CURSOR=YESの場合、OracleでSQL文が実行され、カーソルがクローズされると、プリコンパイラはただちにこの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、関連付けられた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およびHOLD_CURSOR=YESの場合、リンクは保持されます。オープン・カーソルの数がMAXOPENCURSORSの設定値を超えないかぎり、プリコンパイラではリンクは再利用されません。この設定によって後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。文の再解析やOracleプライベートSQL領域用のメモリー割当ては不要です。

暗黙カーソルとともにインラインで使用する場合、SQL文の実行前にRELEASE_CURSORを設定してください。明示カーソルとともにインラインで使用する場合は、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を指定するとHOLD_CURSOR=YESがオーバーライドされ、HOLD_CURSOR=NOを指定するとRELEASE_CURSOR=NOがオーバーライドされます。これら2つのオプションの相互作用の詳細は、表C-1を参照してください

6.7.62 RUNOUTLINE

用途

プリコンパイラを使用して、または後で開発者が手動で「CREATE OUTLINE」文を実行するオプションを提供します。

構文

runoutline={yes | no}

デフォルト値

なし

使用上のノート

runoutline=yesの場合は、プリコンパイルが正常に完了した後で、プリコンパイラ/トランスレータによって、生成された「CREATE OUTLINE」文が実行されます。

RUNOUTLINEを使用する場合は、アウトライン・オプションをtrueまたはcategory_nameに設定する必要があります。このオプションを有効にする場合は、意味検査をフルにする必要があり、つまりオプションsqlcheck=full/semanticsを意味します。sqlcheck=syntax/limited/noneの場合は、エラーが生成されます。

6.7.63 SELECT_ERROR

用途

1行のSELECT文が複数行を戻すとき、あるいはホスト配列の許容範囲を超える行数を戻すときに、プログラムでエラーが発生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

使用上のノート

SELECT_ERROR=YESの場合、1行のSELECT文で戻される行数が多すぎたり、配列のSELECT文でホスト配列に入りきらない行数が戻されたりすると、エラーが発生します。

SELECT_ERROR=NOの場合、1行のSELECT文で戻される行数が多すぎても、配列のSELECT文でホスト配列に入りきらない行数が戻されても、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。選択する行の順序を特定するには、SELECT文に**ORDER BY**句を指定します。SELECT_ERROR=NOのときORDER BY句を指定すると、配列からの選択時にOracleは先頭行または先頭の*n*行を戻します。SELECT_ERROR=YESを指定すると、ORDER BY句の有無を問わず、戻る行数が多すぎる場合にエラーが生成されます。

6.7.64 SQLCHECK

用途

構文およびセマンティック・チェックの種類と範囲を指定します。

構文

SQLCHECK={SEMANTICS|FULL|SYNTAX|LIMITED|NONE}

デフォルト値

SYNTAX

使用上のノート

SEMANTICS値とFULL値は等価です。同様に、SYNTAX値とLIMITED値も等価です。

Oracleプリコンパイラは、埋込みSQL文とPL/SQLブロックの構文およびセマンティックをチェックすることで、プログラムのデバッグに役立ちます。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。

チェックのレベルは、インラインまたはコマンドラインでSQLCHECKオプションを入力することで制御します。ただし、インラインで指定するチェックのレベルを、コマンドラインで指定する(またはデフォルトによって受け入れる)レベルよりも高くすることはできません。たとえば、コマンドラインでSQLCHECK=NONEを指定すると、インラインでSQLCHECK=SYNTAXを指定することはできません。

SQLCHECK=SYNTAX|SEMANTICSの場合、PL/SQLの予約語がSQL文で使用されると、そのSQL文がPL/SQLでなくても、プリコンパイラではエラーが発生します。PL/SQLの予約語を識別子として使用する必要がある場合は、二重引用符で囲んでください。

SQLCHECK=SEMANTICSを指定した場合、次のものを対象とした構文およびセマンティックのチェックが行われます。

- INSERTやUPDATEなどのデータ操作文
- PL/SQLブロック

ただし、リモートのデータ操作文(AT *db_name*句を使用する文)については、構文チェックのみが行われます。

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、Oracleに接続してデータ・ディクショナリにアクセスするとこの情報を取得します。データ操作文やPL/SQLブロックで参照する表がすべてDECLARE TABLE文で定義されている場合は、Oracleに接続する必要があります。

Oracleに接続してもデータ・ディクショナリで見つからない情報がある場合は、DECLARE TABLE文を使用して、欠けている情報

を提供する必要があります。プリコンパイル時には、DECLARE TABLE文の定義とデータ・ディクショナリの定義が矛盾する場合、前者が優先されます。

新しいプログラムをプリコンパイルするときには、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USERIDオプションを指定する必要があります。

SQLCHECK=SYNTAXの場合、プリコンパイラでは次のものの構文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いません。DECLARE TABLE文は無視され、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Oracleデータベース・バージョン7構文規則が使用されます。プリコンパイル済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。

SQLCHECK=NONEの場合、構文チェックもセマンティック・チェックも行われません。DECLARE TABLE文は無視され、PL/SQLブロックは使用できません。プログラムが次のような場合は、SQLCHECK=NONEを指定してください。

- Oracle以外のSQLが含まれる場合(たとえば、Oracle以外のサーバーにオープン・ゲートウェイを介して接続するため)
- まだ作成されておらず、それらのためのDECLARE TABLE文がない表を参照する場合

次の表に、SQLCHECKによって行われるチェックについてまとめます。構文およびセマンティックのチェックの詳細は、構文およびセマンティックのチェックを参照してください。

文	SQLCHECK=SEMANTIC: 構文	SQLCHECK=SEMANTIC: セマンティック	SQLCHECK=SYNTAX: 構文	SQLCHECK=SYNTAX: セマンティック	SQLCHECK=NONE: 構文	SQLCHECK=NONE: セマンティック
DML	◎	◎	◎	該当なし	該当なし	該当なし
リモートDML	◎	該当なし	◎	該当なし	該当なし	該当なし
PL/SQL	◎	◎	該当なし	該当なし	該当なし	該当なし

6.7.65 STMT_CACHE

用途

動的SQL文の文キャッシュ・サイズを指定します。

構文

STMT_CACHE = 0から65535

デフォルト値

0

使用上のノート

stmt_cacheオプションを設定すると、アプリケーションでそれぞれの動的SQL文の予測数を保持できます。

6.7.66 SQLCHECK

用途

構文およびセマンティック・チェックの種類と範囲を指定します。埋込みSQL文およびPL/SQLブロックの構文および意味を検査することで、Pro*C/C++プリコンパイラはコーディングの誤りをすみやかに発見し修正できるように支援します。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。これにより、キーワード、オブジェクト名、演算子、デリミタなどのオブジェクトが、SQL文に正しく配置されていることを確認できます。

構文

SQLCHECK={SEMANTICS | FULL | SYNTAX}

デフォルト値

SYNTAX

機能

次の場合

SQLCHECK=SYNTAX、

クライアント側SQLインタフェースを使用してSQL文の構文のみをチェックします。

次の場合

SQLCHECK=SEMANTICSまたはFULL

SQL文は、解析中に一般的な構文を使用してIDLオブジェクトにパッケージ化またはバンドルされます。一般的な構文では、SQL構文は解釈されず、ホスト変数、インジケータ変数および可能なSQL識別子のみが識別されます。意味フェーズ中に、現在SQLに対して行われている方法で、ホスト変数およびインジケータ変数の妥当性がチェックされます。同じ動作が、表名、列名、タイプなどのセマンティックに対して実行されます。

使用上のノート

SEMANTICSは、FULLと同じです。

インラインまたはコマンドラインで入力できます。

関連項目:

詳細は、ページD-1の「構文およびセマンティックのチェック」を参照してください。(Pro*Cプログラマーズ・ガイド)

6.7.67 THREADS

用途

THREADS=YESの場合、プリコンパイラではコンテキスト宣言を検索します。

構文

THREADS={YES | NO}

デフォルト値

NO

使用上のノート

インラインでは入力できません。

マルチスレッド・サポートを必要とするプログラムには、すべてこのオプションを指定する必要があります。

THREADS=YESの場合、最初のコンテキストが現れ、実行SQL文が見つかる前にEXEC SQL CONTEXT USEディレクティブが検出されないと、プリコンパイラではエラーが発生します。

6.7.68 TYPE_CODE

用途

このマイクロ・オプションは、動的SQL方法4でANSIまたはOracleのいずれのデータ型コードを使用するかを指定します。この設定は、MODEオプションの設定と同じです。

構文

TYPE_CODE={ORACLE | ANSI}

デフォルト値

ORACLE

使用上のノート

インラインでは入力できません。

6.7.69 UNSAFE_NULL

用途

UNSAFE_NULL=YESを指定すると、インジケータ変数を使用せずに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

使用上のノート

インラインでは入力できません。

MODE=ORACLEおよびDBMS=V7の場合のみ、UNSAFE_NULL=YESを指定できます。

埋込みPL/SQLブロックのホスト変数では、UNSAFE_NULLオプションには何の効果もありません。ORA-01405エラーを避けるには、インジケータ変数を使用する必要があります。

6.7.70 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*

デフォルト値

None

使用上のノート

インラインでは入力できません。

先頭にOracle初期化パラメータOS_AUTHENT_PREFIXの付いたOracleユーザー名しか受け入れない自動接続機能を使用している場合は、このオプションを使用しないでください。

SQLCHECK=SEMANTICSの場合、Oracleに接続し、データ・ディクショナリにアクセスして、プリコンパイラに必要な情報を取得させるには、USERIDも指定する必要があります。

6.7.71 UTF16_CHARSET

用途

UNICODE(UTF16)変数で使用される文字セットの形式を指定します。

構文

```
UTF16_CHARSET={NCHAR_CHARSET | DB_CHARSET}
```

デフォルト値

```
NCHAR_CHARSET
```

使用上のノート

コマンドラインまたは構成ファイルでのみ使用でき、インラインでは使用できません。

UTF16_CHARSET=NCHAR_CHARSET(デフォルト)の場合、UNICODE(UTF16)のバインドまたは定義バッファは、サーバー側の各国語文字セットに従って変換されます。ターゲット列がCHARの場合は、パフォーマンスが低下することがあります。

UTF16_CHAR=DB_CHARSETの場合、UNICODE(UTF16)バインドまたは定義バッファは、データベースの文字セットに従って変換されます。

警告:

ターゲット列が NCHAR の場合、データが失われることがあります。

6.7.72 VARCHAR

用途

Pro*COBOLの場合のみ、VARCHARオプションは、「概要」で説明されているCOBOLグループ項目をVARCHARデータ型として扱うようにプリコンパイラに指示します。

構文

```
VARCHAR={YES|NO}
```

デフォルト値

NO

使用上のノート

インラインでは入力できません。

VARCHAR=YESと指定した場合、「概要」で説明した暗黙的なグループ項目を、長さフィールドおよび文字列フィールドを持つOracle VARCHAR外部データ型として受け入れます。

VARCHAR=NOの場合、Pro*COBOLプリコンパイラでは、暗黙的なグループ項目はVARCHAR外部データ型としては受け入れられません。

6.7.73 VERSION

用途

EXEC SQL OBJECT DEREF文によって戻されるオブジェクトのバージョンを指定します。

構文

VERSION={RECENT | LATEST | ANY}

デフォルト値

RECENT

使用上のノート

EXEC ORACLE OPTION文を使用してインラインで入力できます。

RECENTは、現行のトランザクションでオブジェクトが選択されてオブジェクト・キャッシュに入っている場合、そのオブジェクトが戻されることを意味します。シリアライズ可能モードで実行中のトランザクションの場合、このオプションの効果はLATESTと同じですが、ネットワークのラウンドトリップはそれほど多くありません。ほとんどのアプリケーションには、RECENTが適切です。

LATESTは、オブジェクトがオブジェクト・キャッシュに存在しない場合、データベースから取得されることを意味します。オブジェクト・キャッシュに存在する場合は、サーバーからリフレッシュされます。LATESTを使用する場合、ネットワークのラウンドトリップ数が最大になるため注意してください。LATESTは、オブジェクト・キャッシュとサーバーのバッファ・キャッシュをできるかぎり一致させる必要がある場合にのみ使用してください。

ANYは、オブジェクトがすでにオブジェクト・キャッシュに存在している場合、そのオブジェクトが戻されることを意味します。キャッシュになければ、そのオブジェクトはサーバーから取得します。ANYを指定すると、ネットワークのラウンドトリップ数は最小になります。この値を使用するのは、アプリケーションが読取り専用オブジェクトにアクセスする場合や、ユーザーがオブジェクトに排他的にアクセスする場合です。

6.7.74 XREF

用途

リスト・ファイルに相互参照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

使用上のノート

XREF=YESの場合、ホスト変数、カーソル名および文名に相互参照が組み込まれます。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の場合、相互参照セクションは組み込まれません。

6.8 条件付きプリコンパイル

条件付きプリコンパイルでは、特定の条件に基づいて、ホスト・プログラム内のコード・セクションの組込み(または除外)を行います。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を使用すると、異なる環境で実行可能なプログラムを作成できます。

コードの条件付きセクションは、環境と実行するアクションを定義する文で示されます。これらのセクションには、ホスト言語の文とEXEC SQL文を記述できます。次の文により、プリコンパイルを条件付きで制御できます。

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

EXEC ORACLE文はすべて、ホスト言語の文の終了記号で終わる必要があります。たとえばPro*COBOLでは、条件文は「END-EXEC」で終了する必要があります。Pro*FORTRANでは改行記号で終了する必要があります。

6.8.1 例

次の例では、シンボル*site2*が定義されている場合のみ、SELECT文がプリコンパイルされます。

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

次の例に示すように条件ブロックはネストできます。

```
EXEC ORACLE IFDEF outer;
 EXEC ORACLE IFDEF inner;
 ...
 EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

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

6.8.2 シンボルの定義

シンボルを定義するには2通りの方法があります。1つは、次の文をホスト・プログラムに組み込む方法です。

```
EXEC ORACLE DEFINE symbol;
```

もう1つは、次の構文を使用してコマンドラインでシンボルを定義する方法です。

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

*symbol*の部分は大/小文字区別がありません。

Oracleプリコンパイラ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シンボルが事前に定義されます。たとえば、事前に定義されたオペレーティング・システムのシンボルには、CMS、MVS、MS-DOS、UNIXおよびVMSがあります。

6.9 分割プリコンパイル

Oracleプリコンパイルを使用すると、複数のホスト・プログラム・モジュールを別々にプリコンパイルし、それらをリンクして1つの実行可能プログラムを作成できます。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記述する必要はありません。

6.9.1 ガイドライン

次のガイドラインは、いくつかの一般的な問題を回避するのに役立ちます。

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

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、接続に有効な値のみが使用されます。

単一のSQLCAの使用

使用するSQLCAが1つのみの場合は、1つのプログラム・モジュールでそのSQLCAをグローバルに宣言する必要があります。

6.9.2 制限事項

1つの明示カーソルの参照はすべて、同じプログラム・ファイル内にあることが必要です。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、「埋込みSQLの使用方法」を参照してください。

また、SQL文が含まれるプログラム・ファイルにはすべて、ローカルSQL文の有効範囲内にあるSQLCAが1つ必要です。

6.10 コンパイルおよびリンク

実行可能プログラムを作成するには、プリコンパイラによって生成されたソース・ファイルをコンパイルし、その結果得られるオブジェクト・モジュールを、SQLLIBおよびシステム固有のOracleライブラリ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コールを埋め込む場合も、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するときに、このような競合が発生す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・サービスに問い合せて、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。

コンパイルおよびリンクの方法はシステムによって異なります。手順は、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。

OCIハンドルおよび記述子の属性の詳細は、「ハンドルおよび記述子の属性」を参照してください。

6.10.1 システム依存

コンパイルおよび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する必要があります。手順は、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。

6.10.2 マルチバイト・グローバリゼーション・サポートの互換性

マルチバイトのグローバリゼーション・サポート機能を使用する場合、オブジェクト・ファイルをSQLLIBランタイム・ライブラリの現行バージョンにリンクさせる必要があります。このリリースのマルチバイト・グローバリゼーション・サポート機能は、SQLLIBランタイム・ライブラリではサポートされていますが、Oracleサーバーではサポートされていません。この結果生まれたアプリケーションは、Oracleデータベースのどのリリースでも実行できます。

7 トランザクションの定義および制御

この章では、トランザクション処理の実行方法について説明します。Oracleデータ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。次の内容について説明します。

7.1 基本用語

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

Oracleが管理するジョブまたはタスクは、セッションと呼ばれます。アプリケーション・プログラムまたはOracle Formsなどのツールを実行してOracleに接続すると、ユーザー・セッションが開始されます。Oracleでは、ユーザー・セッションを同時に機能させ、コンピュータ・リソースを共有できます。そのために、Oracleでは同時実行性、つまり多数のユーザーによる同じデータベースへのアクセスを制御する必要があります。同時実行性の制御が十分でないと、データの整合性が失われる可能性があります。つまり、データや構造への変更が誤って行われる可能性があります。

Oracleでは、ロックを使用して、データへの同時アクセスを制御します。ロックにより、データの表や行などのデータベース・リソースのユーザーに一時的な所有権が与えられます。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を変更できません。デフォルトのロック機能がOracleのデータおよび構造を保護するため、明示的にリソースをロックする必要はありません。ただし、デフォルトのロックをオーバーライドするときは、表または行単位でデータ・ロックを要求できます。行の共有や排他など、数種類のロック・モードから選択できます。

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

1人のユーザーによって問合せが行われている表を、同時に別のユーザーが更新すると、Oracleでは問合せ用の表データの読取り一貫性ビューが生成されます。つまり、1つの問合せが開始され、処理が進む間、問合せによって読み取られるデータは変わりません。更新アクティビティが続行している間、Oracleでは、表データのスナップショットを取り、変更内容をロールバック・セグメントに記録します。Oracleは、ロールバック・セグメント内の情報を使用して、読取り一貫性のある問合せ結果を作成し、必

要に応じて変更を元に戻します。

7.2 トランザクションによるデータベースの保護

Oracleはトランザクション指向です。つまり、トランザクションを使用してデータの整合性を保証します。トランザクションとは、あるタスクを完了するために定義する1つ以上の論理的に関連付けられたSQL文です。Oracleでは、SQL文によるすべての変更がコミットされるか(確定)、ロールバックされるか(取消し)のどちらかになるように、一連の文が1単位とみなされます。トランザクションの途中でアプリケーション・プログラムに障害が発生すると、データベースは自動的にトランザクション前の状態にリストアされます。

以降の項では、トランザクションの定義および制御方法について説明します。特に、次の方法を学習します。

- トランザクションの開始および終了
- COMMIT文を使用したトランザクションの確定
- SAVEPOINT文をROLLBACK TO文と併用したトランザクションの部分的な取消し
- ROLLBACK文を使用したトランザクション全体の取消し
- RELEASEオプションの指定によるリソースの解放とデータベース接続の切断
- SET TRANSACTION文を使用した読取り専用トランザクションの設定
- FOR UPDATE句またはLOCK TABLE文を使用したデフォルト・ロックのオーバーライド

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

7.3 トランザクションの開始および終了方法

トランザクションは、プログラムの最初の実行SQL文(CONNECT以外)により開始します。1つのトランザクションが終了すると、次の実行SQL文により別のトランザクションが自動的に開始します。このように、すべての実行文はトランザクションの一部です。宣言SQL文は、ロールバックできず、コミットも必要ないため、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文またはROLLBACK文を記述し、RELEASEオプションは付けても付けなくてもかまいません。これにより、データベースへの変更を明示的に確定または取り消します。
- 実行の前と後に自動コミットを発行するデータ定義文(ALTER、CREATEまたはGRANTなど)を記述します。これにより、データベースへの変更を暗黙的に確定します。

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

トランザクションの途中でプログラムに障害が発生すると、Oracleによりエラーが検出され、トランザクションはロールバックされます。オペレーティング・システムに障害が発生すると、データベースがトランザクション前の状態にリストアされます。

7.4 COMMIT文の使用について

データベースへの変更を確定するには、COMMIT文を使用します。変更がコミットされるまで、他のユーザーは変更されたデータにアクセスできず、表示されるのはトランザクション開始前の状態のデータです。COMMIT文は、ホスト変数の値にも、プログラムの制御フローにも影響はありません。具体的には、COMMIT文により次の処理が実行されます。

- 現行のトランザクション中にデータベースに加えられた変更をすべて確定します。

- これらの変更を他のユーザーが参照できるようにします。
- すべてのセーブポイントを消去します(ROLLBACK文の使用についてを参照)。
- 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句で参照されるカーソルをクローズします。MODE={ANSI13|ORACLE}の場合は、すべての明示カーソルをクローズします。
- トランザクションを終了します。

MODE={ANSI13|ORACLE}の場合、CURRENT OF句で参照されていない明示カーソルは、複数のコミットにわたりオープン状態のままです。これによってパフォーマンスが向上します。たとえば、複数のコミットにわたるフェッチについてを参照してください。

これらは通常の処理の一部であるため、COMMIT文は、プログラムのメイン・パスにインラインで設定する必要があります。プログラムを終了する前に、保留中の変更を明示的にコミットしてください。コミットしない場合、保留中の変更はロールバックされます。次の例では、トランザクションをコミットして、Oracleとの接続を切断します。

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

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

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

7.5 ROLLBACK文の使用について

データベースに加えられ保留中の変更を取り消すには、ROLLBACK文を使用します。たとえば、表から間違った行を削除するなどのミスを犯した場合、ROLLBACKを使用すれば、元のデータをリストアできます。ROLLBACK文は、ホスト変数の値にも、プログラムのフロー制御にも影響がありません。具体的には、ROLLBACK文により次の処理が実行されます。

- カレント・トランザクションで実行されたデータベースの変更を取り消します。
- すべてのセーブポイントを消去します。
- トランザクションを終了します。
- 解析ロックを除き、すべての行と表のロックを解除します。
- CURRENT OF句で参照されるカーソルをクローズします。MODE={ANSI|ANSI14}の場合は、すべての明示カーソルをクローズします。

MODE={ANSI13|ORACLE}の場合、CURRENT OF句で参照されない明示カーソルは、複数のロールバックにわたりオープン状態のままです。

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

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

オプションのキーワードWORKを指定すると、ANSI互換になります。RELEASEオプションを指定すると、プログラムで使用されているすべてのリソースが解放され、データベースとの接続が切断されます。

WHENEVER SQLERROR GOTO文からエラー処理ルーチンに分岐するとき、そのルーチンにROLLBACK文が含まれている場合は、ロールバックがエラーで失敗すると、プログラムが無限ループに入るおそれがあります。このような無限ループは、ROLLBACK文の前にWHENEVER SQLERROR CONTINUEを記述することで回避できます。

次の例を考えてみます。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
FOR EACH new employee
 display 'Employee number? ';
 read emp_number;
 display 'Employee name? ';
 read emp_name;
 EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
 VALUES (:emp_number, :emp_name);
ENDFOR;
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
```

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

7.5.1 文レベルのロールバック

Oracleは、SQL文を実行する前に、暗黙的なセーブポイント(ユーザーは操作できません)を設定します。SQL文でエラーが発生すると、Oracle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文で一意の索引に重複する値を挿入しようとしてエラーが発生すると、その文はロールバックされます。

失敗したSQL文によって開始された作業のみが失われます。現在のトランザクションでその文の前に行われた作業は保持されます。このため、データ定義文が失敗した場合、それに先行する自動コミットは取り消されません。

SQL文を実行する前に、Oracleではその文を解析する、つまり、その文が構文規則に従っているか、あるいは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する要があります。SQL文の実行中にエラーが検出されると、ロールバックが発生しますが、解析中にエラーが検出されても、ロールバックは発生しません。

Oracleでは、デッドロックを解消するために、1つのSQL文のみをロールバックすることもできます。関連しているトランザクションの1つにエラーが通知され、そのトランザクション内の現在の文がロールバックされます。

7.6 SAVEPOINT文の使用について

トランザクションの処理中に現在のポイントにマークを付け、名前を指定するには、SAVEPOINT文を使用します。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、次の文では、*start_delete*というセーブポイントを設定しています。

```
EXEC SQL SAVEPOINT start_delete;
```

セーブポイントを設定すると、長いトランザクションを分割できるため、複雑なプロシージャでもうまく制御できます。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。その結果、1つのファンクションが失敗しても、簡単にOracleデータを前の状態にリストアし、リカバリしてから、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、セーブポイントをROLLBACK文とそのTO SAVEPOINT句とともに使用します。TO SAVEPOINT句を使用すると、現行のトランザクションの途中の文までロールバックできるため、変更をすべて取り消す必要はありません。具体的には、ROLLBACK TO SAVEPOINT文により次の処理が実行されます。

- 指定したセーブポイントがマークされた以降のデータベースへの変更を取り消します。
- 指定したセーブポイント以降のセーブポイントをすべて消去します。

- 指定したセーブポイントがマークされた以降に取得された行および表のロックをすべて解除します。

次の例では、MAIL_LIST表にアクセスして新しいリストを挿入し、古いリストを更新して、アクティブでない(少数の)リストを削除しています。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。行数が予想以上に多い場合は、セーブポイント*start_delete*までロールバックして、その削除のみを取り消します。

```
FOR EACH new customer
 display 'Customer number? ';
 read cust_number;
 display 'Customer name? ';
 read cust_name;
EXEC SQL INSERT INTO MAIL_LIST (CUSTNO, CNAME, STAT)
 VALUES (:cust_number, :cust_name, 'ACTIVE');
ENDFOR;
FOR EACH revised status
 display 'Customer number? ';
 read cust_number;
 display 'New status? ';
 read new_status;
 EXEC SQL UPDATE MAIL_LIST
 SET STAT = :new_status WHERE CUSTNO = :cust_number;
ENDFOR;
-- mark savepoint
EXEC SQL SAVEPOINT start_delete;
EXEC SQL DELETE FROM MAIL_LIST WHERE STAT = 'INACTIVE';
IF sqlca.sqlerrd(3) < 25 THEN -- check number of rows deleted
 display 'Number of rows deleted is ', sqlca.sqlerrd(3);
ELSE
 display 'Undoing deletion of ', sqlca.sqlerrd(3), ' rows';
 EXEC SQL WHENEVER SQLERROR GOTO sql_error;
 EXEC SQL ROLLBACK TO SAVEPOINT start_delete;
ENDIF;
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL COMMIT WORK RELEASE;
exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
```

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ロールバックすると、そのセーブポイント以降に設定されたセーブポイントはすべて消去されます。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のみ消去されます。COMMIT文またはROLLBACK文では、すべてのセーブポイントが消去されます。

デフォルトでは、ユーザー・セッションごとのアクティブなセーブポイントの数は、5つに制限されています。アクティブなセーブポイントとは、最後のコミットまたはロールバック以降に設定されたセーブポイントです。データベース管理者(DBA)は、Oracleの初期化パラメータSAVEPOINTSの値を増やして、この制限を引き上げることができます。2つのセーブポイントに同じ名前を付けると、最初のセーブポイントが消去されます。

7.7 RELEASEオプションの使用について

プログラムが異常終了すると、Oracleは自動的に変更をロールバックします。異常終了が発生するのは、プログラムが作業を明示的にコミットもロールバックもせずに、RELEASEオプションを使用してOracleから切断する場合です。

プログラムが正常に終了するのは、プログラムが正常に実行され、オープン状態のカーソルがクローズされ、作業が明示的にコミットまたはロールバックされて、Oracleとの接続が切断され、制御がユーザーに戻された場合です。最後に実行されるSQL文が次のどちらかの場合、プログラムは正常終了します。

```
EXEC SQL COMMIT RELEASE;
```

または

```
EXEC SQL ROLLBACK RELEASE;
```

それ以外の場合は、ユーザー・セッションで取得したロックおよびカーソルは、そのユーザー・セッションがアクティブでなくなったことがOracleで認識されるまで、プログラムの終了後も保持されます。このため、マルチユーザー環境では、他のユーザーがロックされたリソースが解除されるまで必要以上に長く待たされることになりかねません。

7.8 SET TRANSACTION文の使用について

SET TRANSACTION文を使用すると、読取り専用または読取り/書込みトランザクションを開始したり、現行のトランザクションを指定したロールバック・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、ROLLBACK文またはデータ定義文で終了します。

読取り専用トランザクションでは反復可能読取りが行えるため、他のユーザーが更新中の1つ以上の表に対して、複数の問合せを実行する場合に便利です。読取り専用トランザクション中、複数の表と複数の問合せで構成された読取り一貫性ビューが作成され、すべての問合せがデータベースの同じスナップショットを参照します。他のユーザーは、通常の方法でデータの問合せや更新ができます。次にSET TRANSACTION文の例を示します。

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

SET TRANSACTION文は、読取り専用トランザクションの最初のSQL文であることが必要で、1つのトランザクションで1回しか使用できません。READ ONLYパラメータは必須です。これを使用しても、他のトランザクションには影響がありません。読取り専用トランザクションで使用できるのは、SELECT文(FOR UPDATEなし)、LOCK TABLE文、 SET ROLE文、ALTER SESSION文、ALTER SYSTEM文、COMMIT文およびROLLBACK文のみです。

次の例では、店の管理者として、読取り専用トランザクションを使用して要約レポートを生成することで、当日、先週および先月の売り上げを調べます。トランザクションの途中で、他のユーザーがデータベースを更新しても、レポートには影響はありません。

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

7.9 デフォルト・ロックのオーバーライドについて

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

SELECT FOR UPDATE OF文を使用すると、表の特定行を明示的にロックすることで、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、Oracle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、更新または削除の前にロックする場合にのみ、FOR UPDATE OF句を使用してください。

LOCK TABLE文を使用すると、表全体をロックできます。

7.9.1 FOR UPDATE OF句の使用について

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

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

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20
 FOR UPDATE OF SAL;
```

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

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO = 20;
```

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

7.9.2 制限事項

FOR UPDATE OF句を使用すると、複数の表を参照できません。また、明示的なFOR UPDATE OFまたは暗黙的なFOR UPDATEでは、行の排他ロックが取得されます。行ロックが解除されるのは、コミット時またはロールバック時です(セーブポイントまでロールバックする場合は、解除されません)。コミット後にFOR UPDATEカーソルからフェッチしようとすると、次のエラーが発生します。

```
ORA-01002: fetch out of sequence
```

7.9.3 LOCK TABLE文の使用について

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

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

ロック・モードによって、表に対して他にどのようなロックを使用できるかが決まります。たとえば、多くのユーザーが同時に1つの表に対して行共有ロックを取得できる一方、排他ロックを取得できるのは一度に1ユーザーのみです。あるユーザーが表を排他ロックしている間は、他のユーザーはその表内の行の挿入、更新または削除を行えません。ロック・モードの詳細は、「アプリケーション開発者用のSQL処理」を参照してください。

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

表をロックしても、他のユーザーは表に対して問合せができますが、問合せを実行しても表ロックを取得できません。したがって、

問合せが他の問合せや更新を妨げることはなく、更新が問合せを妨げることもありません。2つの異なるトランザクションで同じ行の更新が試みられる場合にのみ、一方のトランザクションが他方のトランザクションの完了まで待機の状態になります。表のロックは、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解除されます。

7.10 複数のコミットにわたるフェッチについて

複数の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を選択してから、その値を使用して、更新または削除中の現在の行を識別します。次の例を検討してください:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL, ROWID FROM EMP WHERE JOB = 'CLERK';
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary, :row_id;
 ...
 EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE ROWID = :row_id;
 EXEC SQL COMMIT;
ENDLOOP;
```

ノート:

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

7.11 分散トランザクションの処理について

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

コミットを発行すると、分散トランザクションによる影響を受ける各データベースの変更が確定されます。コミットのかわりにロールバックを発行すると、すべての変更が取り消されます。ただし、コミットまたはロールバック中にネットワークやコンピュータで障害が発生すると、分散トランザクションの状態は不明またはインダウトになることがあります。そのような場合、FORCE TRANSACTIONシステム権限があれば、FORCE句を使用して、ローカル・データベースでトランザクションを手動でコミットまたはロールバックできます。このトランザクションは、データ・ディクショナリ・ビューDBA_2PC_PENDINGにあるトランザクションIDを引用符で囲んだリテラルで指定する必要があります。次に例を示します。

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

FORCEでは、指定したトランザクションのみがコミットまたはロールバックされるため、現行のトランザクションには影響がありません。インダウトのトランザクションは、手動ではセーブポイントにロールバックできないことに注意してください。

COMMIT文中でCOMMENT句を使用すると、分散トランザクションに関連付けるコメントを指定できます。トランザクションがインダウトになると、COMMENTで指定したテキストがトランザクションIDとともに、データ・ディクショナリ・ビューDBA_2PC_PENDINGに格納さ

れます。テキストには、長さ50文字以内の引用符付きリテラルを指定する必要があります。次に例を示します。

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

分散トランザクションの詳細は、トランザクションを参照してください。

7.12 ガイドライン

次のガイドラインは、いくつかの一般的な問題を回避するのに役立ちます。

7.12.1 アプリケーションの設計について

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。適切に設計されたトランザクションには、特定のタスクを完了するために必要なすべてのステップが過不足なく含まれています。

表を参照するデータは一貫している必要があります。したがって、トランザクション内のSQL文は、一貫した方法でデータを変更する必要があります。たとえば2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および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る必要があります。どちらの処理も、正常終了または失敗が同時であることが必要です。一方の口座への新規預金など、関連のない更新はトランザクションに含めないでください。

7.12.2 ロックの取得について

アプリケーション・プログラムにSQLのロック文が含まれている場合、ロックを要求するOracleユーザーに、そのロックの取得に必要な権限があるか確認してください。データベース管理者(DBA)は、どの表でもロックできます。その他のユーザーは、自分の所有する表か、ALTER、SELECT、INSERT、UPDATEまたはDELETEなどの権限を付与されている表をロックできます。

7.12.3 PL/SQLの使用について

PL/SQLブロックがトランザクションの一部になっている場合、そのロック内のコミットおよびロールバックは、トランザクション全体に影響を与えます。次の例では、ロールバックは更新および挿入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;
...
```

8 エラー処理および診断

アプリケーション・プログラムでは、ランタイム・エラーを予想し、そのエラーをリカバリするように対処する必要があります。この章では、エラー・レポートおよびリカバリを詳しく説明します。具体的には、状態変数SQLCODE、SQLSTATE、SQLCA(SQLコミュニケーション領域)およびWHENEVER文を使用して警告およびエラーを処理する方法を説明します。さらに状態変数ORACA(Oracle通信領域)による問題点の診断方法も紹介します。次の内容について説明します。

- エラー処理の必要性
- エラー処理の代替手段
- MODE={ANSI|ANSI14}の場合の状態変数の使用について
- SQL通信領域の使用について
- Oracle通信領域の使用について

8.1 エラー処理の必要性

どのようなアプリケーション・プログラムでも、その大部分をエラー処理に充てる必要があります。エラー処理の主な利点は、エラーが発生しても、プログラムの処理を続行できることです。エラーは、設計ミス、コーディングの誤り、ハードウェアの障害、無効なユーザー入力、その他様々な原因で発生します。

潜在的なエラーをすべて予測するのは無理ですが、プログラムにとって意味のある特定の種類のエラーについてアクションを考えることはできます。Oracleプリコンパイラにとって、エラー処理とはSQL文の実行エラーの検出およびリカバリのことです。

値が切り捨てられたことを示す警告やデータの終わりなどの状態の変更も処理できます。INSERT文、UPDATE文またはDELETE文は、表内の処理対象行すべてを処理する前に失敗することがあるため、データ操作文を実行するたびにエラー状態および警告状態がないか調べることが特に重要です。

8.2 エラー処理の代替手段

Oracleプリコンパイラには、エラー処理の手段として役立つ4種類の状態変数が用意されています。

- SQLCODE(Pro*FORTRANではSQLCOD)
- SQLSTATE(Pro*FORTRANではSQLSTA)
- SQLCA(WHENEVER文を使用)
- ORACA

MODEオプションは、ANSI/ISOの準拠を制御します。SQLCODE変数、SQLSTATE変数およびSQLCA変数の可用性は、MODE設定によって異なります。ORACA変数は、MODE設定に関係なく宣言して使用できます。詳細は、Oracle通信領域の使用についてを参照してください。

MODE={ORACLE|ANSI13}の場合、SQLCA状態変数を宣言する必要があります。SQLCODE宣言およびSQLSTATE宣言も指定できますが(お薦めはできません)、状態変数としては認識されません。詳細は、SQL通信領域の使用についてを参照してください。

MODE={ANSI|ANSI14}の場合、SQLCODE変数、SQLSTATE変数およびSQLCA変数のうちの任意の1つ、2つあるいは3つすべてを使用できます。アプリケーションに最適な変数(または変数の組合せ)を確認するには、MODE={ANSI|ANSI14}の場合の状態変数の使用についてを参照してください。

8.2.1 SQLCODEおよびSQLSTATE

Oracleプリコンパイラのリリース1.5では、SQLCODE状態変数が、SQL89標準ANSI/ISOエラー・レポート・メカニズムとして導入されました。SQL92標準では、SQLCODEが非推奨機能としてあげられ、新しい状態変数SQLSTATE(Oracleプリコンパイラのリリース1.6で導入)が推奨ANSI/ISOエラー・レポート・メカニズムとして定義されました。

SQLCODEには、エラー・コードおよび見つからない状態が格納されています。これは、SQL89との互換性のためのみに保持されており、標準の将来のバージョンからは削除される見込みです。

SQLCODEとは異なり、SQLSTATEにはエラーおよび警告のコードが格納され、標準化されたコード体系を使用します。SQL文を実行すると、Oracleサーバーからは、現在有効範囲内にあるSQLSTATE変数にステータス・コードが戻されます。ステータス・コードは、SQL文が正常に実行されたか、あるいは例外(エラーまたは警戒の状態)が発生したかを示します。相互運用性(システムが情報を容易に交換する能力)を高めるために、SQL92では、すべての一般的なSQL例外が事前に定義されています。

8.2.2 SQLCA

SQLCAは、レコードに似たホスト言語データ構造体です。Oracleでは、すべての実行SQL文の後でSQLCAが更新されます。(宣言部の後では、SQLCAの値は未定義になります。)SQLCAに格納されているOracleリターン・コードをチェックすることで、プログラムはSQL文の結果を判断できます。これには2通りの方法があります。

- 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、SQLCA変数に対するコードの明示的チェックを使用するか、あるいは両方を使用できます。一般的に、WHENEVER文を使用する方が簡単で、移植性も高く、ANSI準拠であるためお薦めします。

8.2.3 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い場合、カーソル統計、SQL文データ、オプション設定およびシステム統計が格納されているORACAを使用できます。

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、Oracle通信領域の使用についてを参照してください。

8.3 MODE={ANSI|ANSI14}の場合の状態変数の使用について

MODE={ANSI|ANSI14}の場合、次の状態変数の少なくとも1つを宣言する必要があります(2つあるいは3つ全部を宣言してもかまいません)。

- SQLCODE
- SQLSTATE
- SQLCA

Pro*COBOLでは、SQLCAが宣言されていれば、SQLCODEは宣言できません。同様に、SQLCODEが宣言されていれば、SQLCAは宣言できません。SQLCAデータ構造のフィールドにはPro*COBOL用のエラー・コードが格納されており、SQLCODEとも呼ばれ、そのためこれら2つの状態変数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PL/SQL文の後で、独自のコードを使用して明示的にSQLCODEおよびSQLSTATE、SQLCODEまたはSQLSTATEをチェックすることで、最新の実行SQL文の結果を取得できます。また、SQLCAを暗黙的にチェックすることも(WHENEVER SQLERROR文およびWHENEVER SQLWARNING文を使用)、SQLCA変数を明示的にチェック

することもできます。

ノート:

MODE={ORACLE|ANSI13}の場合、SQLCA 状態変数を宣言する必要があります。詳細は、SQL 通信領域の使用についてを参照してください。

8.3.1 履歴情報

Oracleプリコンパイラによる状態変数および変数の組合せの処理は、リリース1.5から進化してきました。

8.3.2 リリース1.5

Oracle Precompilerのリリース1.5では、宣言部で宣言されたかどうかに関係なく、ステータス変数SQLCODEが存在すると想定していました。実際、プリコンパイラは、SQLCODEのための宣言があるかどうかをチェックせず、ただ宣言が存在すると想定していました。SQLCAが状態変数として使用されるのは、SQLCAのINCLUDEが存在する場合のみでした。

8.3.3 リリース1.6

Oracleプリコンパイラ・リリース1.6から、プリコンパイラでは、SQLCODE状態変数があると仮定しなくなり、SQLCODE状態変数は必須ではありません。SQLCA、SQLCODEまたはSQLSTATEのうち少なくとも1つを宣言する必要があります。

SQLCODEは、少なくとも次の基準の1つが満たされたときにかぎり、状態変数として認識されます。

- 宣言部で完全に正しいデータ型で宣言されている場合。
- プリコンパイラで他の状態変数が見つからない場合。

プリコンパイラが宣言部で(完全に正しい型の)SQLSTATE宣言を検出した場合、または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。

8.3.4 リリース1.7

Oracleプリコンパイラのリリース1.5では、SQLCODE変数を宣言部の外部で宣言できたのに対し、リリース1.6以降のプリコンパイラでは、同時にSQLCAを宣言すると、互換性の問題が生じます。これを修正するために、リリース1.6.7では新しいオプションASSUME_SQLCODE={YES|NO} (デフォルトはNO)が追加され、リリース1.7で新機能として文書化されました。

ASSUME_SQLCODE=YESと指定した場合、SQLSTATEまたはSQLCA(Pro*FORTRANのみ)が状態変数として宣言されると、宣言部で宣言されているかどうか、あるいは正しい型かどうかに関係なく、プリコンパイラではSQLCODEが宣言されているものとみなします。このため、リリース1.6.7以降は、この点でリリース1.5と同様の動作をします。プリコンパイラ・オプションASSUME_SQLCODEの詳細は、「ASSUME_SQLCODE」を参照してください。

8.3.5 状態変数の宣言について

この項では、SQLCODEおよびSQLSTATEの宣言方法について説明します。SQLCA状態変数の宣言の詳細は、SQLCAの宣言を参照してください。

8.3.6 SQLCODEの宣言

SQLCODE(Pro*FORTRANではSQLCOD)は、表8-1に示すように、宣言部の内側または外側のいずれかで、4バイト整

数変数として宣言する必要があります。

表8-1 SQLCODEの宣言

言語	SQLCODEの宣言
COBOL	SQLCODE PIC S9(9) COMP.
FORTRAN	INTEGER*4 SQLCOD

宣言部の外部で宣言すると、SQLCODEは、ASSUME_SQLCODE=YESの場合にのみ状態変数として認識されます。MODE={ORACLE|ANSI13}の場合、SQLCODEの宣言は無視されます。

警告:

Pro*COBOL ソース・ファイルでは、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・メカニズムを使用するとエラーが発生します。

ローカルとグローバルの両方の宣言が可能なホスト言語を使用することにより、複数のSQLCODE変数を宣言できます。ローカルSQLCODEへのアクセスは、プログラム内の有効範囲により制限されます。SQLの動作が終わるたびに、Oracleから現在有効範囲にあるSQLCODEにステータス・コードが戻されます。したがって、プログラムでは、SQLCODEを明示的にチェックするか、WHENEVER文で暗黙的にチェックして、最近のSQL動作の結果を確認できます。

特定のコンパイル・ユニットのSQLCAのかわりにSQLCODEを宣言すると、プリコンパイラでは、そのユニット用に内部SQLCAを1つ割り当てます。ホスト・プログラムでは、その内部SQLCAにアクセスできません。SQLCAとSQLCODE(Pro*COBOLでは非対応)を宣言すると、OracleからはSQLの動作が終わるたびに、両方に同じステータス・コードが戻されます。

8.3.7 SQLSTATEの宣言

SQLSTATE(Pro*FORTRANではSQLSTA)は、表8-2に示すように、宣言部内5文字の英数字文字列として宣言する必要があります。SQLCAの宣言はオプションです。

表8-2 SQLSTATEの宣言

言語	SQLSTATEの宣言
COBOL	SQLSTATE PIC X(5).
FORTRAN	CHARACTER*5 SQLSTA

MODE={ORACLE|ANSI13}の場合、SQLSTATE変数の宣言は無視されます。

8.3.8 状態変数の組合せ

MODE={ANSI|ANSI14}の場合、状態変数の動作は、次の条件によって決まります。

- 宣言されている変数
- 宣言の配置(宣言部の内側か外側か)
- ASSUME_SQLCODE設定

表8-3および表8-4では、ASSUME_SQLCODE=NOおよびASSUME_SQLCODE=YESのそれぞれの場合、各状態変数の組合せがどのような動作になるかを説明しています。

表8-3 状態変数の組合せ - SQLCODE = NO

宣言部(内/外/ --)			**動作**
SQLCODE	**SQLSTATE**	**SQLCA**	
外	--	--	SQLCODE が宣言され、状態変数であるとみなされます。
外	--	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCA が状態変数として宣言され、SQLCODE は宣言されますが、状態変数とは認識されません。
外	--	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
外	外	--	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。
外	外	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCA が状態変数として宣言され、SQLCODE および SQLSTATE は宣言されますが、状態変数とは認識されません。
外	外	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
外	内	--	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。
外	内	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLSTATE および SQLCA が状態変数として宣言され、SQLCODE は宣言されますが、状態変数とは認識されません。
外	内	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。

宣言部(内/外/ --)			動作
内	--	--	SQLCODE が状態変数として宣言されます。
内	--	外	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、SQLCODE および SQLCA が状態変数として宣言されます。
内	--	内	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、この状態変数の構成はサポートされていません。
内	外	--	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。
内	外	外	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、SQLCODE および SQLCA が状態変数として宣言され、SQLSTATE は宣言されますが、状態変数とは認識されません。
内	外	内	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、この状態変数の構成はサポートされていません。
内	内	--	SQLCODE および SQLSTATE が状態変数として宣言されます。
内	内	外	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、SQLCODE、SQLSTATE および SQLCA が状態変数として宣言されます。
内	内	内	Pro*COBOL では、この状態変数の構成はサポートされていません。 Pro*FORTRAN では、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。
--	--	外	SQLCA が状態変数として宣言されます。
--	--	内	Pro*COBOL では、SQLCA が状態ホスト変数として宣言されます。 Pro*FORTRAN では、この状態変数の構成はサポートされていません。
--	外	--	この状態変数の構成はサポートされていません。
--	外	外	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。

宣言部(内/外/ --)			動作
--	外	内	Pro*COBOL では、SQLCA が状態ホスト変数として宣言され、SQLSTATE は宣言されますが、状態変数とは認識されません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
--	内	--	SQLSTATE が状態変数として宣言されます。
--	内	外	SQLSTATE および SQLCA が状態変数として宣言されます。
--	内	内	Pro*COBOL では、SQLSTATE および SQLCA が状態ホスト変数として宣言されます。Pro*FORTRAN では、この状態変数の構成はサポートされていません。

表8-4 状態変数の組合せ - SQLCODE = YES

宣言部(内/外/ --)			動作
SQLCODE	**SQLSTATE**	**SQLCA**	
外	--	--	SQLCODE が宣言され、状態変数であるとみなされます。
外	--	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCA が状態変数として宣言され、SQLCODE は宣言されますが、状態変数とはみなされません。
外	--	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
外	外	--	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。
外	外	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCA が状態変数として宣言され、SQLCODE は宣言されて状態変数とみなされ、SQLSTATE は宣言されますが、状態変数とは認識されません。
外	外	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
外	内	--	SQLSTATE が状態変数として宣言されます。SQLCODE は宣言さ

宣言部(内/外/ --)			動作
			れますが、状態変数とはみなされません。
外	内	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLSTATE および SQLCA が状態変数として宣言され、SQLCODE は宣言されて状態変数とみなされます。
外	内	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
内	--	--	SQLCODE が状態変数として宣言されます。
内	--	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCODE および SQLCA が状態変数として宣言されます。
内	--	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
内	外	--	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。
内	外	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCODE および SQLCA が状態変数として宣言され、SQLSTATE は宣言されますが、状態変数とは認識されません。
内	外	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていません。
内	内	--	SQLCODE および SQLSTATE が状態変数として宣言されます。
内	内	外	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、SQLCODE、SQLSTATE および SQLCA が状態変数として宣言されます。
内	内	内	Pro*COBOL では、この状態変数の構成はサポートされていません。Pro*FORTRAN では、この状態変数の構成はサポートされていませ

宣言部(内/外/ --)			動作
			ん。
-- -- -- -- -- -- -- -- -- -- -- --	-- -- -- 外 外 外 内 内 内	-- 外 内 -- 外 内 -- 外 内	これらの状態変数の構成はサポートされていません。SQLCODE は、ASSUME_SQLCODE=YES の場合、宣言部の内側または外側のいずれかで宣言する必要があります。

8.3.9 状態変数の値

この項では、SQLCODE状態変数およびSQLSTATE状態変数の値について説明します。SQLCA状態変数については、エラー・レポートの主要コンポーネントを参照してください。

8.3.10 SQLCODE値

SQLの動作が終わるたびに、Oracleから現在有効範囲にあるSQLCODE変数にステータス・コードが戻されます。SQLの動作の結果を示すステータス・コードは、次のいずれかの数値です。

0

OracleではSQL文が実行され、エラーも例外も検出されませんでした。

> 0

Oracleは文を実行しましたが、例外を検出しました。これは、WHERE句の条件を満たす行が見つからなかった場合、あるいはSELECT INTOまたはFETCHで行が戻されなかった場合に発生します。

MODE={ANSI|ANSI14|ANSI13}の場合、行のINSERTがなかったときに+100がSQLCODEに戻されます。副問合せで処理に行が戻されなかったときにこの状態が発生します。

< 0

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、Oracleは文を実行しませんでした。このようなエラーはリカバリ不能です。このようなエラーが発生すると、ほとんどの場合はカレント・トランザクションがロールバックされます。負のリターン・コードは、『*Oracle Database*エラー・メッセージ』に記載されているエラー・コードに対応しています。

最近のSQLの動作の結果は、SQLCODEを独自のコードで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で確認できます。

特定のプリコンパイル・ユニットのSQLCAのかわりにSQLCODEを宣言すると、プリコンパイラでは、そのユニット用に内部SQLCAを1つ割り当てます。ホスト・プログラムでは、その内部SQLCAにアクセスできません。SQLCAとSQLCODE(Pro*FORTRANのみ)を宣言すると、OracleからはSQLの動作が終わるたびに、両方に同じステータス・コードが戻されます。

ノート:

MODE={ORACLE|ANSI13}の場合、SQLCODE の宣言は無視されます。

8.3.11 SQLSTATE値

SQLSTATE状態コードは、2文字のクラス・コードおよびその後に続く3文字のサブクラス・コードで構成されます。クラス・コード00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22012は、クラス・コード22(データ例外)とサブクラス・コード012(ゼロ除算)で構成されています。

SQLSTATE値の5文字はそれぞれ、数字(0から9)または大文字の英字(AからZ)です。0から4の数字、あるいはAからHの文字で始まるクラス・コードは、事前定義の条件(SQL92で定義されている条件)用に予約されています。他のすべてのクラス・コードは実装定義の状態用に確保されています。事前定義クラスのうち、0から4の数字またはAからHの文字で始まるサブクラス・コードは、事前定義の副条件用に予約されています。他のすべてのサブクラス・コードは、実装時に定義される副条件用に予約されています。図8-1は、コード体系を示しています。

図8-1 SQLSTATEコード体系

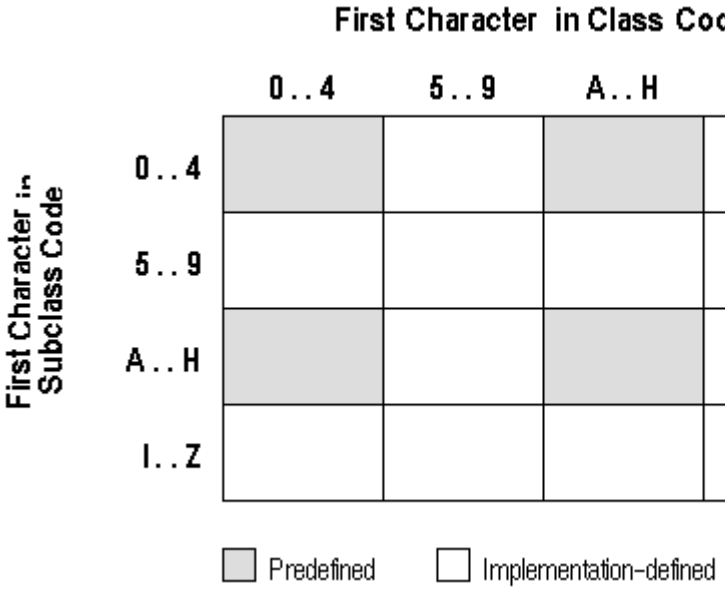


表8-5は、SQL92で事前に定義されているクラスを示しています。

表8-5 事前に定義されたSQL92のクラス

クラス	状態
00	正常終了
01	警告
02	データなし
07	動的 SQL エラー
08	接続の例外
0A	機能がサポートされていない
21	カーディナリティ違反

クラス	状態
22	データの例外
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	文字セット名が無効
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
40	トランザクション・ロールバック

クラス	状態
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

ノート:

クラス・コード HZ は、国際標準規格 ISO/IEC DIS 9579-2 で定義された条件であるリモート・データベース・アクセス用に確保されています。

表8-6 は、OracleのエラーとSQLSTATEステータス・コードの対応を示しています。1つのステータス・コードに複数のOracleエラーが対応する場合もあります。その他の場合、ステータス・コードに対応するOracleエラーはありません(したがって、最後の列は空)。60000から99999の範囲のステータス・コードは、実装定義です。

表8-6 OracleエラーとSQLSTATEステータス・コードの対応

コード	状態	Oracleエラー
00000	正常終了	ORA-00000
01000	警告	
01001	カーソル操作の競合	
01002	切断エラー	
01003	集合関数での 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	SQL-02142
01006	権限が取り消されていない	
01007	権限が付与されていない	
01008	暗黙的なゼロビットの埋込み	

コード	状態	Oracleエラー
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 ORA-0100
07000	動的 SQL エラー	SQL-02137 SQL-02139
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	プリコンパイルされた SQL 文がカーソル仕様ではありません。	
07006	制限付きのデータ型属性違反	
07007	結果フィールドには USING 句が必要です。	
07008	記述子の数が無効	SQL-02126 SQL-02141
07009	記述子の索引が無効	SQL-02140
08000	接続の例外	
08001	SQL のクライアントが SQL 接続を確立できません。	
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	

コード	状態	Oracleエラー
08006	接続の失敗	
08007	トランザクションの結果が不明	
0A000	機能がサポートされていない	ORA-03000～03099
0A001	複数のサーバー・トランザクション	
21000	カーディナリティ違反	ORA-01427 SQL-02112 ORA-01422
22000	データの例外	
22001	文字列データの右側切捨て	ORA-01401 ORA-01406 ORA-12899
22002	NULL 値(インジケータ・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	タイム・ゾーンによる時差が無効	
22011	部分文字列のエラー	
22012	ゼロによる除算	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	

コード	状態	Oracleエラー
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。	
22022	インジケータのオーバーフロー	ORA-01411
22023	パラメータ値が無効	ORA-01025 ORA-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-01400 ORA-02290～02299
24000	カーソル状態が無効	ORA-01001 から 01003 ORA-01410 ORA-06511 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	

コード	状態	Oracleエラー
2B000	依存権限記述子がまだ存在しています。	
2C000	文字セット名が無効	
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	SQL-02138
34000	カーソル名が無効	
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・ロールバック	ORA-02091～02092
40001	シリアライズの失敗	
40002	整合性制約違反	
40003	文の完了が不明	
42000	構文エラーまたはアクセス規則違反	ORA-00022 ORA-00251 ORA-00900 から 00999 ORA-01031 ORA-01490 から 01493 ORA-01700 から 01799 ORA-01900 から 02099 ORA-02140 から 02289 ORA-02420 から 02424 ORA-02450 から 02499 ORA-03276 から 03299

コード	状態	Oracleエラー
		ORA-04040 から 04059 ORA-04070 から 04099
44000	WITH_CHECK_OPTION 指定違反	ORA-01402
60000	システム・エラー	ORA-00370 から 00429 ORA-00600 から 00899 ORA-06430 から 06449 ORA-07200 から 07999 ORA-09700 から 09999
61000	リソース・エラー	ORA-00018 から 00035 ORA-00050 から 00068 ORA-02376 から 02399 ORA-04020 から 04039
62000	共有サーバーおよび分離プロセスのエラー	ORA-00101～00120 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 から 00159 SQL-02128 ORA-02700 から 02899 ORA-03100 から 03199 ORA-06200 から 06249 SQL-02128
64000	制御ファイル、データベース・ファイルおよび REDO ファイルのエラー、アーカイブおよびメディア・リカバリのエラー	ORA-00200～00369 ORA-01100～01250
65000	PL/SQL のエラー	ORA-06500～06599
66000	SQL*Net ドライバのエラー	ORA-06000 から 06149 ORA-06250 から 06429 ORA-06600 から 06999 ORA-12100 から 12299 ORA-12500 から 12599
67000	ライセンス許可のエラー	ORA-00430～00439
69000	SQL*Connect のエラー	ORA-00570～00599

コード	状態	Oracleエラー
		ORA-07000～07199
72000	SQL 実行フェーズのエラー	ORA-01000 から 01099 ORA-01400 から 01489 ORA-01495 から 01499 ORA-01500 から 01699 ORA-02400 から 02419 ORA-02425 から 02449 ORA-04060 から 04069 ORA-08000 から 08190 ORA-12000 から 12019 ORA-12300 から 12499 ORA-12700 から 21999
82100	メモリー不足のためメモリーが割り当てられません。	SQL-02100
82101	無効なカーソル・キャッシュです。ユニット・カーソル/グローバル・カーソルが一致しません。	SQL-02101
82102	無効なカーソル・キャッシュです。グローバル・カーソル・エントリがありません。	SQL-02102
82103	無効なカーソル・キャッシュです。カーソル・キャッシュ参照の範囲を超えています。	SQL-02103
82104	無効なホスト・キャッシュです。使用可能なカーソル・キャッシュがありません。	SQL-02104
82105	無効なカーソル・キャッシュです。グローバル・カーソルがありません。	SQL-02105
82106	無効なカーソル・キャッシュ(無効な Oracle カーソル番号)	SQL-02106
82107	ランタイム・ライブラリに対して古すぎるプログラム	SQL-02107
82108	ランタイム・ライブラリに渡された無効な記述子	SQL-02108
82109	無効なホスト・キャッシュです。ホスト参照が範囲外です。	SQL-02109
82110	無効なホスト・キャッシュです。ホスト・キャッシュ・エントリの型が	SQL-02110

コード	状態	Oracleエラー
	無効です。	
82111	ヒープ一貫性エラー	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 引数および第 2 引数が両方とも NULL です。	SQL-02120
82117	データベースへの接続での OPEN または PREPARE が無効です。	
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	接続エラーでメッセージを取り出せません。	SQL-02125
82120	プリコンパイラと SQLLIB のバージョンが一致しません。	SQL-02127
82121	FETCH されたバイト数が奇数です。	SQL-02129
82122	EXEC TOOLS インタフェースが使用できません。	SQL-02130
82123	使用中のランタイム・コンテキスト	SQL-02131
82124	ランタイム・コンテキストの割当て不能	ORA-01422 SQL-02132
82125	スレッドで使用するプロセスを初期化できません。	SQL-02133
82126	ランタイム・コンテキストが無効	SQL-02134
90000	デバッグ・イベント	ORA-10000～10999

コード	状態	Oracleエラー
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	

8.4 SQL通信領域の使用について

SQL通信領域(SQLCA)は、レコードに似たデータ構造体です。そのフィールドには、SQL文を実行するたびにOracleによって更新されるエラー、警告およびステータス情報が格納されます。したがって、SQLCAには常に最新のSQLの動作結果が反映されます。結果を確認するには、SQLCA内の変数をチェックします。

ローカルおよびグローバルの両方の宣言ができるホスト言語では、プログラムで複数のローカルSQLCAを使用できます。たとえば、1つのグローバルSQLCAと複数のローカルSQLCAを設定できます。ローカルSQLCAへのアクセスは、プログラム内の有効範囲により制限されます。Oracleからは、アクティブなSQLCAにのみ情報が戻されます。

また、アプリケーションでSQL*Netを使用してローカルとリモートのデータベースに同時にアクセスしている場合、すべてのデータベースが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、「同時接続」を参照してください。

MODE={ORACLE|ANSI13}の場合、SQLCAは必須です。SQLCAが宣言されないと、コンパイル時エラーが発生します。SQLCAはMODE={ANSI|ANSI14}のときにはオプションですが、WHENEVER SQLWARNING文はSQLCAの宣言がなければ使用できません。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。

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、プリコンパイラでは、そのユニット用に内部SQLCAを1つ割り当てます。ホスト・プログラムでは、その内部SQLCAにアクセスできません。SQLCAとSQLCODE(Pro*FORTRANのみ)を宣言すると、OracleからはSQLの動作が終わるたびに、両方に同じステータス・コードが戻されます。

MODE={ANSI|ANSI14}の場合、いずれかのSQLSTATEを宣言する必要があります(「SQLCODEおよびSQLSTATE」を参照)。SQLSTATE状態変数は、SQL92標準で指定されたSQLSTATE状態変数をサポートします。SQLSTATE状態変数は、SQLCODEの有無に関係なく使用できます。詳細は、表8-3と表8-4を参照してください。

8.4.1 SQLCAの宣言

SQLCAを宣言するには、次のようにホスト言語のソース・ファイルにEXEC SQL INCLUDE文を使用してSQLCAを含めます。

```
* Include the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE ORACA
EXEC SQL INCLUDE SQLCA;
```

SQLCAは、SQLCAのINCLUDEがある場合にかぎり使用されます。

プログラムをプリコンパイルすると、INCLUDE SQLCA文は複数の変数宣言に置換されます。この変数宣言により、Oracleはそのプログラムと通信できます。

8.4.2 Pro*COBOLでのSQLCAの宣言について

Pro*COBOLでは、INCLUDEが宣言部の内側であろうが外側であろうが違いはありません。Pro*COBOLでのSQLCAの宣言の詳細は、「SQL通信領域の使用方法」を参照してください。

8.4.3 Pro*FORTRANでのSQLCAの宣言について

Pro*FORTRANでは、SQLCAはCOMMONブロックなので宣言部の外側で宣言する必要があります。さらに、SQLCAはCONNECT文と最初の実行可能なFORTRAN文の前に置く必要があります。

SQLCAは、SQL文を含むサブルーチンやファンクションごとに宣言します。サブルーチンやファンクションの1つでSQL文が実行されるたびに、OracleによりCOMMONブロック内に保存されているSQLCAが更新されます。

通常、重要なのはCOMMONリストにある変数の順序とデータ型のみで、名前ではありません。ただし、プリコンパイラではそれらの名前を参照するコードが生成されるため、SQLCA変数の名前を変更することはできません。したがって、SQLCAの宣言はすべて同一であることが必要です。Pro*FORTRANでのSQLCAの宣言の詳細は、「SQL通信領域の使用方法」を参照してください。

8.4.4 SQLCAの内容

SQLCAにはSQL文の実行結果に関する次のランタイム情報が格納されます。

- Oracleエラー・コード
- 警告フラグ
- イベント情報
- 処理済行数
- 診断情報

図8-2に、SQLCA内のすべての変数を示します。特定のホスト言語におけるSQLCA構造体および変数の名前を調べるには、このマニュアルに対するその言語用の補足資料を参照してください。

図8-2 SQLCAの変数

SQLCAID	Character string "SQLCA"
SQLCABC	Length of SQLCA data structure in bytes
SQLCODE	Oracle error message code
SQLERRM	Subrecord for storing error message
SQLERRML	Length of error message
SQLERRMC	Text of error message
SQLERRP	Reserved for future use
SQLERRD	Array of six integer status codes
SQLERRD(1)	Reserved for future use
SQLERRD(2)	Reserved for future use
SQLERRD(3)	Number of rows processed
SQLERRD(4)	Reserved for future use
SQLERRD(5)	Parse error offset
SQLERRD(6)	Reserved for future use
SQLWARN	Array of eight warning flags
SQLWARN(0)	Another warning flag set
SQLWARN(1)	Character string truncated
SQLWARN(2)	No longer in use
SQLWARN(3)	SELECT list not equal to INTO list
SQLWARN(4)	DELETE or UPDATE without WHERE clause
SQLWARN(5)	Reserved for future use
SQLWARN(6)	No longer in use
SQLWARN(7)	No longer in use
SQLEXT	Reserved for future use

8.4.5 エラー・レポートの基本コンポーネント

エラー・レポートは、SQLCA内の変数によって異なります。この項では、エラー・レポートの主要コンポーネントについて説明します。次の項では、SQLCAについて詳しく説明します。

8.4.6 ステータス・コード

すべての実行SQL文は、SQLCA変数のSQLCODEにステータス・コードを戻し、このステータス・コードは、WHENEVER文によって暗黙的にチェックすることも、独自のコードによって明示的にチェックすることもできます。

ステータス・コードは、ゼロ、負数、正数のいずれかです。SQLCODEの詳細は、「SQLCODEの宣言」を参照してください。

8.4.7 警告フラグ

警告フラグは、SQLCA変数のSQLWARN(0)からSQLWARN(7)に戻され、これらは暗黙的にも明示的にもチェックできます。警告フラグは、Oracleでエラーとみなされない実行時の状態のチェックに便利です。

8.4.8 処理済行数

最後に実行したSQL文で処理された行数は、SQLCA変数のSQLERRD(3)に戻され、これは明示的にチェックできます。

厳密には、この変数はエラー・レポート用ではなく、誤りの防止に役立ちます。たとえば、表から約10行を削除するとします。削除後に、SQLERRD(3)をチェックすると、75行が処理されたことが判明します。念のために、削除処理をロールバックしてWHERE句の検索条件を確認します。

8.4.9 解析エラー・オフセット

SQL文は実行前に必ず解析され、構文規則に従っているか、有効なデータベース・オブジェクトを参照しているかが検証されます。エラーが検出されると、SQLCA変数のSQLERRD(5)にオフセットが格納され、これは明示的にチェックできます。このオフセットには、解析エラーの始まりを示すSQL文中の文字位置が示されています。先頭の文字位置は0 (ゼロ)です。たとえば、オフセットが9のとき、解析エラーは10番目の文字から始まっています。

デフォルトでは、静的SQL文は、プリコンパイル時に構文エラーがないかチェックされます。そのため、SQLERRD(5)は、プログラムの実行時に受け入れたり作成したりする動的SQL文のデバッグには最も便利です。

解析エラーは、キーワードの欠落、位置指定の誤りまたはスペルミス、無効なオプション、存在しない表などが原因で発生します。たとえば、次のような動的SQL文があるとします。

```
UPDATE EMP SET JIB = :job_title WHERE EMPNO = :emp_number
```

これは解析エラーになります。

```
ORA-00904: invalid column name
```

原因は、列名JOBのスペルミスです。SQLERRD(5)の値は15になりますが、これは誤った列名JIBが16番目の文字で始まっているためです。

SQL文に解析エラーがない場合、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)の場合にのみ行ってください。

8.4.10 エラー・メッセージ・テキスト

Oracleエラーのエラー・コードおよびメッセージは、SQLCA変数のSQLERRMCに格納されています。テキストの最初の最大70文字が格納されます。70文字を超えるメッセージの全文を取得するには、SQLGLMファンクションを使用します。エラー・メッセージの全文の取得を参照してください。

8.4.11 SQLCA構造体

この項では、SQLCAの構造体、そのフィールドおよびそこに格納可能な値について説明します。

8.4.12 SQLCAID

この文字列フィールドは、SQLCAに初期化され、SQL通信領域を識別します。

8.4.13 SQLCABC

この整数フィールドには、SQLCA構造体の長さがバイト単位で入ります。

8.4.14 SQLCODE

この整数フィールドには、最後に実行されたSQL文のステータス・コードが入ります。SQLの動作の結果を示すステータス・コードは、次のいずれかの数値です。

0

Oracleは文を実行し、エラーも例外も検出しませんでした。

> 0

Oracleは文を実行しましたが、例外を検出しました。これは、WHERE句の検索条件を満たす行が見つからなかった場合、あるいはSELECT INTOまたはFETCHで行が戻されなかった場合に発生します。

< 0

MODE={ANSI|ANSI14|ANSI13}の場合、行のINSERTがなかったときに+100がSQLCODEに戻されます。副問合せで処理に行が戻されなかったときにこの状態が発生します。

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、Oracleは文を実行しませんでした。このようなエラーはリカバリ不能です。このようなエラーが発生すると、ほとんどの場合はカレント・トランザクションがロールバックされます。

負のリターン・コードは、『*Oracle Database*エラー・メッセージ』に記載されているエラー・コードに対応しています。

8.4.15 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に格納されるメッセージ・テキストの長さが入ります。

SQLERRMC

この文字列フィールドには、SQLCODEに格納されたエラー・コードのメッセージテキストが保持され、最大70文字を格納できます。70文字を超えるメッセージの全文を取得するには、SQLGLMファンクションを使用します。

SQLCODEが負数であることを確認します

(SQLERRMCを参照する前に)。SQLCODEが0 (ゼロ)の場合は、SQLERRMCを参照するとそれ以前のSQL文に関連するメッセージ・テキストが戻されます。

8.4.16 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。

8.4.17 SQLERRD

この2進整数の配列には6つの要素があります。SQLERRD(FORTRANではSQLERDと呼ぶ)内のフィールドの説明は、次のとおりです。

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、最後に実行したSQL文によって処理された行数が入ります。ただし、SQL文が失敗すると、1つの例外を除き、SQLERRD(3)の値は未定義となります。配列の動作中にエラーが発生すると、エラーの原因となった行で処理が停止し、SQLERRD(3)は正常に処理された行数を示します。

処理済行数はOPEN文の後にゼロに設定され、FETCH文の後で増加します。EXECUTE、INSERT、UPDATE、DELETEおよびSELECT INTOの各文では、数は正常に処理された行数が反映されます。この数には、UPDATEやDELETE_CASCADEで処理された行は含まれません。たとえばWHERE句の条件を満たす20行が削除された後で、列制約条件に違反する5行が削除されたときの処理済行数は、25ではなく20となります。

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。

8.4.18 SQLWARN

この1文字の配列には8つの要素があります。これらの要素は警告フラグとして使用されます。Oracleではそれに文字値W(警告)を割り当てることでフラグを設定します。フラグは例外状態の発生を警告します。

たとえば、切り捨てられた列値が出力ホスト変数に割り当てられると、警告フラグが設定されます。

また、図8-2ではSQLWARNを配列として示していますが、Pro*COBOLでは、SQLWARN0からSQLWARN7までの基本のPIC X項目を持つグループ・アイテムとして実装されることにも注意してください。Pro*FORTRANの実装は、SQLWN0からSQLWN7までのLOGICAL変数で構成されます。

SQLWARNのフィールドの説明は次のとおりです。

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のみ適用されます。Oracle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)

このフラグは、AVG、COUNTまたはMAXなどのSQLグループ関数の評価で、1つ以上のNULLが無視された場合に設定されます。COUNT(*)を除き、すべてのグループ関数ではNULLが無視されるため、この動作になります。必要な場合は、SQL関数のNVLを使用して、NULLの列エントリに一時的に値(たとえばゼロ)を割り当てることができます。

SQLWARN(3)

このフラグは、問合せ選択リスト内の列数が、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場合に設定されます。戻される項目の数は、両者のうち少ない方です。

SQLWARN(4)

このフラグは、表内のすべての行がWHERE句のないUPDATE文またはDELETE文で処理された場合に設定されます。処理される行数を制限する検索条件がない場合、更新または削除は無条件と呼ばれます。このような更新や削除は例外的であるため、この警告フラグが設定されます。必要に応じて、トランザクションはロールバックできます。

SQLWARN(5)

このフラグは、EXEC SQL CREATE {PROCEDURE|FUNCTION|PACKAGE|PACKAGE BODY}文が、PL/SQLコンパイル・エラーのために失敗したときに設定されます。

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。

8.4.19 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。

8.4.20 PL/SQLの考慮事項

プリコンパイラ・プログラムで埋込みPL/SQLブロックを実行するときに、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、ブロックで複数の行がフェッチされた場合、処理済行数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フィールドのみになります。

8.4.21 エラー・メッセージのテキスト全体の取得

SQLCAには70文字までのエラー・メッセージを格納できます。それより長い(またはネストされた)エラー・メッセージの全文を取得するには、SQLGLMファンクションが必要です。Oracleに接続している場合、次の構文を使用してSQLGLMをコールできます。

```
SQLGLM(message_buffer, buffer_size, message_length);
```

説明:

message_buffer

エラー・メッセージを格納するためのテキスト・バッファです(Oracleではこのバッファの最後まで空白文字で埋め込みます)。

buffer_size

バッファの最大サイズをバイト単位で示した整数変数です。

message_length

エラー・メッセージの実際の長さが格納される整数変数です。

Oracleエラー・メッセージの最大長は、エラー・コード、ネストされたメッセージ、表や列の名前など、メッセージの挿入部分を含めて512文字です。SQLGLMによって戻されるエラー・メッセージの最大長は、*buffer_size*に指定した値によって決まります。

次の例では、SQLGLMをコールして、100文字以内の長さのエラー・メッセージを取得します。

```
-- declare variables for function call
msg_buffer CHARACTER(100);
buf_size INTEGER;
msg_length INTEGER;
set buf_size = 100;
EXEC SQL WHENEVER SQLERROR DO sql_error;
-- other statements
ROUTINE sql_error
BEGIN
 -- get full text of error message
 SQLGLM(msg_buffer, buf_size, msg_length);
 display contents of msg_buffer;
 exit program with an error
END sql_error;
```

SQLGLMは、SQLエラーが発生した場合にのみコールされます。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。

8.4.22 WHENEVER文の使用方法

デフォルトでは、プリコンパイルされたプログラムはOracleエラーおよび警告の状態を無視し、可能であれば処理を続行します。自動状態チェックおよびエラー処理を実行するには、WHENEVER文を使用します。

WHENEVER文を使用すると、Oracleでエラー、警告または「見つかりません」の状態が検出されたときのアクションを指定できます。これらのアクションには、次の文の継続実行、ルーチンのコール、ラベル付きの文への分岐、停止などがあります。

WHENEVER文は次の構文を使用して記述します。

```
EXEC SQL WHENEVER <condition> <action>;
```

次の状態がないか、Oracleに自動的にSQLCAをチェックさせることができます。

8.4.23 SQLWARNING

Oracleから警告が戻されたか(同時にSQLWARN(1)からSQLWARN(7)の警告フラグのうちの1つが設定されます)、または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、SQLWARN(1)は、Oracleにより切り捨てられた列値が出力ホスト変数に割り当てられた場合に設定されます。

MODE={ANSI|ANSI14}の場合、SQLCAの宣言はオプションです。ただし、WHENEVER SQLWARNINGを使用するには、必ずSQLCAを宣言してください。

8.4.24 SQLERROR

Oracleからエラーが戻されたので、SQLCODEには負の値が設定されています。

8.4.25 NOT FOUND

OracleでWHERE句の検索条件を満たす行を検出できなかったか、SELECT INTOまたはFETCHで行が戻されなかったため、SQLCODEには+1403が設定されています(MODE={ANSI|ANSI14| ANSI13}のときは+100)。
MODE={ANSI|ANSI14|ANSI13}の場合、行がINSERTされなければ、+100がSQLCODEに戻されます。

Oracleで前述の状態のいずれかが検出されたときは、プログラムに次のいずれかのアクションを実行させることができます。

8.4.26 CONTINUE

可能であれば、プログラムは次の文から実行を継続します。これはデフォルトの動作で、WHENEVER文を使用しない場合と同じです。これを使用すると、状態チェックを終了できます。

8.4.27 DO

プログラムでは、制御を内部ルーチンに移します。ルーチンの最後に達すると、制御は失敗したSQL文の次の文に移ります。

ルーチンとは、COBOLパラグラフまたはFORTRANサブルーチンなど、起動可能な関数プログラム・ユニットです。ここでは、COBOLのサブルーチンなど、別にコンパイルされたプログラムは、ルーチンではありません。

ルーチンの開始と終了の通常の規則が適用されます。ただし、ルーチンにパラーメータを渡すことはできません。さらに、ルーチンでは値を戻すことはできません。

パラーメータ*routine_call*は、次の例のように、ホスト言語を起動します。

```
EXEC SQL -- COBOL
 WHENEVER <condition> DO PERFORM <paragraph_name> -- COBOL
END-EXEC. -- COBOL
```

または

```
EXEC SQL -- FORTRAN
 WHENEVER <condition> DO CALL <subroutine_name> -- FORTRAN
```

8.4.28 GOTO

プログラムはラベル付きの文に分岐します。

8.4.29 STOP

プログラムは実行を停止し、コミットされていない作業がロールバックされます。

ここでは注意が必要です。STOPアクションでは、Oracleとの接続を切断する前に何もメッセージが表示されません。Pascalでは、同等のコマンドがないため、STOPアクションは無効です。

8.4.30 例

たとえば、プログラムで次のアクションが必要だとします。

- 「データが見つかりません」という状態が発生した場合、*close_cursor*に進みます。
- 警告が発生した場合、次の文の処理を続行します。
- エラーが発生した場合、*error_handler*に進みます。

最初の実行SQL文の前に、次のWHENEVER文を記述します。

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

次のPro*Cの例では、WHENEVER...DO文を使用して特定のエラーを処理します。

```
EXEC SQL WHENEVER SQLERROR DO handle_insert_error;
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
 VALUES (:emp_number, :emp_name, :dept_number);
EXEC SQL WHENEVER SQLERROR DO handle_delete_error;
EXEC SQL DELETE FROM DEPT WHERE DEPTNO = :dept_number;
...
ROUTINE handle_insert_error;
 BEGIN
 IF sqlca.sqlcode = -1 THEN -- duplicate key value
 ...
 ELSEIF sqlca.sqlcode = -1401 THEN -- value too large
 ...
 ENDIF;
 ...
 END;
ROUTINE handle_delete_error;
 BEGIN
 IF sqlca.sqlerrd(3) = 0 THEN -- no rows deleted
 ...
 ELSE
 ...
 ENDIF;
 ...
 END;
...
```

SQLCAの変数をチェックしてアクションの過程を決定する手順に注意してください。

8.4.31 有効範囲

WHENEVERは宣言文であるため、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。テストするのは、ソース・ファイルの中でその文より後に記述されているすべての実行SQL文であり、プログラム・ロジックの流れの中でその文より後にくる実行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文の前に記述する必要があります。

WHENEVER文は、同じ条件をチェックする別のWHENEVER文に置き換えられるまで有効です。

次の例では、最初のWHENEVER SQLERROR文は2番目の文に置き換えられるため、CONNECT文に対してのみ適用されます。2番目のWHENEVER SQLERROR文は、*step1*から*step3*への制御の流れに関係なく、UPDATE文とDROP文の両方に適用されます。

```
step1:
EXEC SQL WHENEVER SQLERROR STOP;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
 ...
 GOTO step3;
step2:
```

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;
 ...
step3:
 EXEC SQL DROP INDEX EMP_INDEX;
 ...
```

8.4.32 ガイドライン

次のガイドラインは、一般的な問題を回避するために役立ちます。

文の位置。通常、WHENEVER文はプログラムの最初の実行SQL文の前に記述してください。そうすればWHENEVER文はファイルの最後まで有効なので、発生するすべてのエラーを確実に捕捉できます。

データの終了条件の処理。カーソルを使用して行をフェッチするときは、プログラムでデータの終了条件を処理できるようにしておく必要があります。FETCHによりデータが戻されない場合、プログラムでは次のようにCLOSEコマンドが発行されるコードのラベル付きセクションに分岐します。

```
SQL WHENEVER NOT FOUND GOTO no_more;
...
no_more:
 ...
 EXEC SQL CLOSE my_cursor;
 ...
```

無限ループの回避。WHENEVER SQLERROR GOTO文が、実行SQL文を含むエラー処理ルーチンに分岐する場合、そのSQL文にエラーが発生すると、プログラムは無限ループに陥る恐れがあります。これを回避するには、次の例のように、そのSQL文の前にWHENEVER SQLERROR CONTINUEを記述します。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 ...
```

WHENEVER SQLERROR CONTINUE文を指定しなければ、ROLLBACKエラーが発生したときにこのルーチンが再び起動され、無限ルーチンに陥ります。

WHENEVERを不注意に使用すると、問題が発生する可能性があります。たとえば、DELETE文で検索条件を満たす行がないためNOT FOUNDを設定すると、次のコードは無限ループに陥ります。

```
-- improper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
ENDLOOP;
no_more:
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
 ...
```

次の例では、GOTOのターゲットを再設定することで、NOT FOUNDの状態を適切に処理します。

```
-- proper use of WHENEVER
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
```

```
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 ...
ENDLOOP;
no_more:
 EXEC SQL WHENEVER NOT FOUND GOTO no_match;
 EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
 ...
no_match:
 ...
```

アドレス指定能力の維持。ローカルおよびグローバルの識別子を使用できるホスト言語により、WHENEVER GOTO文に支配されるすべてのSQL文が必ずGOTOラベルに分岐できるようにします。次のコードでは、FUNC1内の*labelA*がFUNC2内のINSERT文の有効範囲内にないため、コンパイル時エラーが発生します。

```
FUNC1
 BEGIN
 EXEC SQL WHENEVER SQLERROR GOTO labelA;
 EXEC SQL DELETE FROM EMP WHERE DEPTNO = :dept_number;
 ...
 labelA:
 ...
 END;
FUNC2
 BEGIN
 EXEC SQL INSERT INTO EMP (JOB) VALUES (:job_title);
 ...
 END;
```

WHENEVER GOTO文の分岐先のラベルは、この文と同じプリコンパイル・ファイル内にする必要があります。

エラー後の戻り。エラーの処理後にプログラムに戻る必要がある場合、DO *routine_call*アクションを使用します。または、次の例のように、SQLCODEの値をテストしてもかまいません。

```
EXEC SQL UPDATE EMP SET SAL = SAL * 1.10;
IF sqlca.sqlcode < 0 THEN
 -- handle error
EXEC SQL DROP INDEX EMP_INDEX;
...
```

アクティブなWHENEVER GOTO文またはWHENEVER STOP文がないことを確認してください。

8.4.33 SQL文のテキストの取得

多くのプリコンパイラ・アプリケーションでは、処理中のテキスト文、その長さ、そこで指定されているSQLコマンド(INSERTまたはSELECTなど)を把握すると便利です。これは、動的SQLを使用するアプリケーションについて特に当てはまります。

ルーチンSQLGLSは、SQLLIBランタイム・ライブラリの一部で、次の情報を戻します。

- 最後に解析されたSQL文のテキスト
- 文の長さ
- 文で使用されているSQLコマンドの機能コード(表8-8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、3または4では、SQL文の作成後にSQLGLSをコールできます。

SQLGLSをコールするには、次の構文を使用します。

表8-7は、SQLGLS引数リストでパラメータに使用可能なホスト言語のデータ型を示しています。

表8-7 SQLGLSパラメータのデータ型

パラメータ	言語	データ型
SQLSTM	COBOL	PIC X(*n*)
SQLSTM	FORTRAN	CHARACTER*n
STMLEN、SQLFC	COBOL	PIC S9(9) COMP
STMLEN、SQLFC	FORTRAN	INTEGER*4

パラメータはすべて、参照によって渡す必要があります。通常、デフォルトでパラメータ引渡し規則は参照によって引渡されるため、特別な処置は必要ありません。

SQLSTMパラメータは、空白埋込み(NULLで終了しない)文字バッファで、SQL文の戻されたテキストが格納されます。プログラムでは、静的にバッファを宣言するか、動的にバッファのメモリーを割り当てる必要があります。

長さパラメータ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(単位はバイト)に設定します。SQLGLSが戻すときに、SQLSTMバッファには、SQL文のテキストにバッファの長さまで空白を足したものが含まれます。STMLENは戻された文テキストの実際のバイト数を戻します。足されている空白は数えません。ただし、エラーが発生した場合、STMLENはゼロを戻します。

次のエラーが考えられます。

- 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。

SQLFCパラメータは、文のSQLコマンドのSQL機能コードを戻す4バイトの整数です。表8-8は、各SQLコマンドの機能コードを示しています。

SQLGLSでは、次のコマンドを含む文は戻されません。

- CONNECT
- COMMIT
- ROLLBACK
- RELEASE
- FETCH

これらの文にはSQLファンクション・コードがありません。

表8-8 SQLコマンドの機能コード

コード	SQL機能	コード	SQL機能

コード	SQL機能	コード	SQL機能
01	CREATE TABLE	39	AUDIT
02	SET ROLE	40	NOAUDIT
03	INSERT	41	ALTER INDEX
04	SELECT	42	CREATE EXTERNAL DATABASE
05	UPDATE	43	DROP EXTERNAL DATABASE
06	DROP ROLE	44	CREATE DATABASE
07	DROP VIEW	45	ALTER DATABASE
08	DROP TABLE	46	CREATE ROLLBACK SEGMENT
09	DELETE	47	ALTER ROLLBACK SEGMENT
10	CREATE VIEW	48	DROP ROLLBACK SEGMENT
11	DROP USER	49	CREATE TABLESPACE
12	CREATE ROLE	50	ALTER TABLESPACE
13	CREATE SEQUENCE	51	DROP TABLESPACE
14	ALTER SEQUENCE	52	ALTER SESSION
15	(未使用)	53	ALTER USER
16	DROP SEQUENCE	54	COMMIT
17	CREATE SCHEMA	55	ROLLBACK
18	CREATE CLUSTER	56	SAVEPOINT
19	CREATE USER	57	CREATE CONTROL FILE
20	CREATE INDEX	58	ALTER TRACING

コード	SQL機能	コード	SQL機能
21	DROP INDEX	59	CREATE TRIGGER
22	DROP CLUSTER	60	ALTER TRIGGER
23	VALIDATE INDEX	61	DROP TRIGGER
24	CREATE PROCEDURE	62	ANALYZE TABLE
25	ALTER PROCEDURE	63	ANALYZE INDEX
26	ALTER TABLE	64	ANALYZE CLUSTER
27	EXPLAIN	65	CREATE PROFILE
28	GRANT	66	DROP PROFILE
29	REVOKE	67	ALTER PROFILE
30	CREATE SYNONYM	68	DROP PROCEDURE
31	DROP SYNONYM	69	(未使用)
32	ALTER SYSTEM SWITCH LOG	70	ALTER RESOURCE COST
33	SET TRANSACTION	71	CREATE SNAPSHOT LOG
34	PL/SQL EXECUTE	72	ALTER SNAPSHOT LOG
35	LOCK TABLE	73	DROP SNAPSHOT LOG
36	(未使用)	74	CREATE SNAPSHOT
37	RENAME	75	ALTER SNAPSHOT
38	COMMENT	76	DROP SNAPSHOT

8.5 Oracle通信領域の使用について

SQLCAで標準的なSQL通信が処理されるのと同様に、Oracle通信領域(ORACA)ではOracle通信が処理されます。ランタイム・エラーおよび状態の変化について、SQLCAで提供されるより詳しい情報が必要な場合は、ORACAを使用してください。

これには、豊富な診断ツールが用意されています。ただし、ORACAの使用はランタイム・オーバーヘッドを増加させるため、あくまでもオプションです。

ORACAは問題の診断に役立つ上に、プログラムによるOracleリソース(SQL文エグゼキュータやカーソル・キャッシュなど)の利用を監視できます。

ローカルおよびグローバルな宣言が可能なホスト言語では、プログラムで複数のORACAを使用できます。たとえば、1つのグローバルORACAと複数のローカルORACAを設定できます。ローカルORACAへのアクセスは、プログラム内のその有効範囲によって制限されます。Oracleでは、アクティブなORACAに対してのみ情報が戻されます。情報はコミットまたはロールバック後でなければ使用できません。

8.5.1 ORACAの宣言

ORACAを宣言するには、次のようにホスト言語のソース・ファイルにEXEC SQL INCLUDE文を使用してORACA含めます。

```
* Include the Oracle Communications Area (ORACA).
 EXEC SQL INCLUDE ORACA
```

ORACAは宣言部の外側で宣言する必要があります。

プログラムをプリコンパイルすると、INCLUDE ORACA文は複数の変数宣言に置換されます。この変数宣言により、Oracleはそのプログラムと通信できます。

8.5.2 ORACAの有効化

ORACAを有効にするには、コマンドラインに次のようにORACAオプションを指定する必要があります。

```
ORACA=YES
```

またはインラインで次のように指定します

```
EXEC ORACLE OPTION (ORACA=YES);
```

その後、ORACA内にフラグを設定することで、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。

8.5.3 ORACAの内容

ORACAには、次のように、オプションの設定、システムの統計および高度な診断情報が保存されています。

- SQL文のテキスト(テキストの保存時に指定できます)
- エラーが発生したファイルの名前
- ファイル内のエラーの位置
- カーソル・キャッシュのエラーおよび統計情報

図8-3は、ORACA内のすべての変数を示しています。特定のホスト言語におけるORACA構造体および変数の名前を調べるには、このマニュアルのその言語用の補足資料を参照してください。

図8-3 ORACAの変数

```
ORACAID         Character string "ORACA"
ORACABC         Length of ORACA data structure in bytes
ORACCHF         Cursor cache conssistency flag
ORADBGF         Master debug flag
ORAHCHF         Heap consistency flag
ORASTXTF        Save SQL statement flag
ORASTXT         Subrecord for storing SQL statement
    ORASTXTL    Length of current SQL statement
    ORASTXTC    Text of current SQL statement
ORASFNM         Subrecord for storing filename
    ORASFNML    Length of filename
    ORASFNMC    Name of file containing current SQL statement
ORASLNR         Line in file at or near current SQL statement
ORAHOC          Highest MAXOPENCURSORS requested
ORAMOC          Maximum open cursors required
ORACOC          Current number of cursors used
ORANOR          Number of cursor cache reassignments
ORANPR          Number of SQL statement parses
ORANEX          Number of SQL statement executions
```

8.5.4 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。これらのフラグにゼロ以外の値を割り当てて設定すると、次のことが可能になります。

- SQL文のテキストの保存
- DEBUG処理の有効化
- カーソル・キャッシュの一貫性チェック(カーソル・キャッシュとは、カーソル管理に使用されるメモリーで継続的に更新される領域)
- ヒープの一貫性チェック(ヒープとは、動的変数用に予約されているメモリー領域です)
- カーソルの統計情報の収集

次の説明は、必要なオプションを選択するときの参考になります。

8.5.5 ORACA構造体

この項では、ORACAの構造体、そのフィールドおよびそこに格納可能な値について説明します。

8.5.6 ORACAID

この文字列フィールドは、ORACAに初期化され、Oracle通信領域を識別します。

8.5.7 ORACABC

この整数フィールドには、ORACA構造体の長さがバイト単位で入ります。

8.5.8 ORACCHF

マスターDEBUGフラグ(ORADBGF)が設定されている場合、このフラグによって、各カーソルの動作の前にカーソ・キャッシュの一貫性をチェックできます。

Oracleランタイム・ライブラリでは一貫性チェックが行われ、エラー・メッセージが発行されることがあります(エラー・メッセージについては、『*Oracle Database*エラー・メッセージ』を参照してください)。これらは、Oracleエラー・メッセージと同様にSQLCAに戻さ

れます。

このフラグは次のいずれかを設定します。

- 0: キャッシュ一貫性チェックを無効にします(デフォルト)。
- 1: キャッシュ一貫性チェックを有効にします。

8.5.9 ORADBGF

このマスター・フラグを使用すると、DEBUGオプションをすべて選択できます。これには次の設定があります。

- 0: すべてのDEBUG処理を無効にします(デフォルト)。
- 1: すべてのDEBUG処理を有効にします。

8.5.10 ORAHCHF

マスターDEBUGフラグ(ORADBGF)が設定されている場合、プリコンパイラにより動的にメモリーが割り当てられたり解放されたりするたびに、このフラグがOracleランタイム・ライブラリにヒープの一貫性チェックを指示します。これはメモリー障害を起こすプログラムの不具合を検出するのに役立ちます。

このフラグはCONNECTコマンドを発行する前に設定する必要があります。また、このフラグは一度設定するとクリアできなくなります。つまり、設定後にこのフラグの変更要求があっても無視されます。これには次の設定があります。

- 0: ヒープ一貫性チェックを無効にします(デフォルト)。
- 1: ヒープ一貫性チェックを有効にします。

8.5.11 ORASTXTF

このフラグを使用すると、現行のSQL文のテキストを保存するタイミングを指定できます。これには次の設定があります。

- 0: SQL文のテキストを保存しません(デフォルト)。
- 1: SQLERRORのSQL文のテキストのみ保存します。
- 2: SQLERRORまたはSQLWARNINGのSQL文のテキストのみ保存します。
- 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。

8.5.12 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。

8.5.13 ORASTXT

このサブレコードは、問題のあるSQL文を見つけるのに役立ちます。Oracleで解析された最後のSQL文のテキストを保存できます。こ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SQL文の長さが入ります。

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。先頭から最大70文字までのテキストが保存されます。

プリコンパイラによって解析された文(CONNECT、FETCHおよびCOMMITなど)は、ORACAには保存されません。

8.5.14 ORASFNM

このサブレコードは、現行SQL文を含むファイルを識別し、1つのアプリケーションのために複数のファイルがプリコンパイルされるときのエラー検出に役立ちます。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入ります。

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。先頭から最大70文字が格納されます。

8.5.15 ORASLNR

この整数フィールドは、現行SQL文がある行またはその付近の行を識別します。

8.5.16 カーソル・キャッシュ統計情報

次の変数により、カーソル・キャッシュ統計情報を収集できます。これらは、プログラムでCOMMIT文またはROLLBACK文が発行されるたびに自動的に設定されます。内部的には、CONNECTされたデータベースごとにこれらの変数のセットがあります。ORACA内の現行値は、最後にコミットまたはロールバックが実行されたデータベースに関するものです。

8.5.17 ORAHOC

この整数フィールドには、プログラムの実行時にMAXOPENCURSORSに設定された最大値が記録されます。

8.5.18 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Oracleカーソルの最大数が記録されます。MAXOPENCURSORSに設定されている値が小さすぎて、その結果プリコンパイラによってカーソル・キャッシュが拡張されると、この数はORAHOCより大きくなることがあります。

8.5.19 ORACOC

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ているOracleカーソルの現在の数が記録されます。

8.5.20 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度割り当てられたカーソル・キャッシュの数が記録されます。この数値は、カーソル・キャッシュのスラッシングの程度を示すもので、できるだけ小さく保つ必要があります。

8.5.21 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。

8.5.22 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の数値に対する割合は、できるかぎり大きく保つ必要があります。つまり、不要な再解析は回避します。詳細は、パフォーマンス・チューニングを参照してください。

8.5.23 例

次のプログラムでは、部門番号の入力を要求し、その部門内の各従業員の名前および給与を2つの表のいずれかに挿入してから、ORACAからの診断情報を表示します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 emp_name INTEGER;
 dept_number INTEGER;
 salary REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE ORACA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR DO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
EXEC ORACLE OPTION (ORACA=YES);
-- set flags in the ORACA
set oraca.oradbgf = 1; -- enable debug operations
set oraca.oracchf = 1; -- enable cursor cache consistency check
set oraca.orastxtf = 3; -- always save the SQL statement
display 'Department number? ';
read dept_number;
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL + NVL(COMM,0)
 FROM EMP
 WHERE DEPTNO = :dept_number;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO no_more;
rLOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 IF salary < 2500 THEN
 EXEC SQL INSERT INTO PAY1 VALUES (:emp_name, :salary);
 ELSE
 EXEC SQL INSERT INTO PAY2 VALUES (:emp_name, :salary);
 ENDIF;
ENDLOOP;
ROUTINE no_more
BEGIN
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL COMMIT WORK RELEASE;
 display 'Last SQL statement: ', oraca.orastxt.orastxtc;
 display '... at or near line number: ', oraca.oraslnr;
 display
 display ' Cursor Cache Statistics';
 display '-----------------------------------------';
 display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
 display 'Maximum open cursors required: ', oraca.oramoc;
 display 'Current number of open cursors: ', oraca.oracoc;
 display 'Number of cache reassignments: ', oraca.oranor;
 display 'Number of SQL statement parses: ', oraca.oranpr;
 display 'Number of SQL statement executions: ', oraca.oranex;
 exit program;
END no_more;
```

```
ROUTINE sql_error
BEGIN
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Last SQL statement: ', oraca.orastxt.orastxtc;
 display '... at or near line number: ', oraca.oraslnr;
 display
 display ' Cursor Cache Statistics';
 display '-----------------------------------------';
 display 'Maximum value of MAXOPENCURSORS ', oraca.orahoc;
 display 'Maximum open cursors required: ', oraca.oramoc;
 display 'Current number of open cursors: ', oraca.oracoc;
 display 'Number of cache reassignments: ', oraca.oranor;
 display 'Number of SQL statement parses: ', oraca.oranpr;
 display 'Number of SQL statement executions: ', oraca.oranex;
 exit program with an error;
END sql_error;
```

9 ホスト配列の使用方法

この章の内容は次のとおりです。

この章では、コーディングの簡略化とプログラムのパフォーマンス改善のために配列を使用する方法について説明します。配列を使用したOracleデータの処理方法、1つのSQL文で配列のすべての要素を操作する方法、処理対象の配列要素の数を制限する方法について学習します。次の項目について説明します。

- ホスト配列
- 配列を使用する理由
- ホスト配列宣言方法
- SQL文での配列の使用方法

9.1 ホスト配列

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

9.2 配列を使用する理由

配列を使用すると、プログラミングを簡単にでき、パフォーマンスも改善されます。アプリケーションの作成時には、通常、大量のデータ集合の保存や操作の問題に直面します。配列は、各集合の個々の項目の命名および参照のタスクを簡略化します。

配列を使用すると、アプリケーションのパフォーマンスが向上します。配列により、1つのSQL文でデータ項目の集合全体を操作できます。このため、特にネットワーク環境では、Oracleの通信オーバーヘッドが大幅に軽減されます。たとえば、およそ300人の従業員に関する情報をEMPという表に挿入する必要があるとします。配列がないと、プログラムは300の個々のINSERT(各従業員に1つ)を実行する必要があります。配列を使用すれば、INSERTの実行は1回で済みます。

9.3 ホスト変数の宣言

ホスト配列は、単純なホスト変数と同様に宣言部で宣言します。ホスト配列の次元の設定(サイズの設定)も宣言部で行います。次の例では、3つのホスト配列を宣言するとともに、それぞれの次元を50要素に設定しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
```

9.3.1 配列の次元設定

ホスト配列の最大次元は32,767要素です。この最大次元を超えるホスト配列を使用すると、「パラメータの値が有効範囲外です」というランタイム・エラーが発生します。1つのSQL文で複数のホスト配列を使用する場合は、それらの次元を同じにする必要があります。同じでない場合には、プリコンパイル時に「配列サイズが一致しません」という警告メッセージが出ます。この警告を無視すると、プリコンパイラではSQLの操作で最小のサイズが使用されます。

9.3.2 制限事項

ポインタのホスト配列は宣言できません。また、1つのSQLで参照できるホスト配列は1次元(要素)に制限されます。したがって、次の例で宣言されている2次元配列は無効です。

```
EXEC SQL BEGIN DECLARE SECTION;
 hi_lo_scores (25, 25) INTEGER; -- not allowed
EXEC SQL END DECLARE SECTION;
```

9.4 SQL文での配列の使用方法について

Oracleプリコンパイラでは、データ操作文でホスト配列を使用できます。ホスト配列は、INSERT文、UPDATE文およびDELETE文では入力変数として、SELECT文およびFETCH文のINTO句では出力変数として使用できます。

MODE=ANSI14の場合、配列処理はできません。つまり、SQL文のホスト配列を参照できるのは、
MODE＝{ANSI|ANSI13|ORACLE}の場合のみです。

ホスト配列と単純ホスト変数に使用される構文はほとんど同じです。ただし、オプションのFOR句で配列処理が制御できるという点に違いがあります。また、ホスト配列と単純ホスト変数を1つのSQL文で併用するときにも制限があります。

後続の項では、データ操作文でのホスト配列の使用方法を説明します。

9.5 配列への選択について

ホスト配列は、SELECT文の出力変数として使用できます。選択によって戻される最大行数がわかっている場合、その数の要素でホスト配列の次元を設定してください。次の例では、3つのホスト配列へのデータを直接選択します。この選択で戻される行が50行以下であることがわかっているため、配列の次元を50要素に設定します。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT ENAME, EMPNO, SAL
 INTO :emp_name, :emp_number, :salary
 FROM EMP
 WHERE SAL > 1000;
```

この例では、SELECT文により最大50行が戻されます。選択される行数が49行以下の場合、または50行のみを取り出す場合はこの方法を使用します。ただし、選択される行数が51行以上の場合は、この方法ではすべての行を取り出せません。SELECT文を再実行すると、他に条件を満たす行があっても、最初の50行が再び戻されます。この場合は、より大きな配列を設定する

か、FETCH文で使用するカーソルを宣言する必要があります。

SELECT INTO文で設定した次元以上の行数が戻された場合、次のエラー・メッセージが表示されます。

```
SQL-02112: SELECT...INTO returns too many rows
```

これはSELECT_ERROR=NOを指定していないためです。SELECT_ERRORオプションの詳細は、SELECT_ERRORを参照してください。

9.5.1 一括フェッチ

選択で戻される最大行数がわからない場合には、cursor_nameを宣言してオープンし、そこから一括でフェッチできます。ループ内でバッチ・フェッチを実行すると、多数の行を簡単に取り出せます。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行ずつまとめてフェッチします。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (20) INTEGER;
 salary (20) REAL;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, SAL FROM EMP;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND DO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_number, :salary;
 -- process batch of rows
ENDLOOP;
```

9.5.2 フェッチされる行数

各フェッチで戻されるのは、最大でも配列の次元の行数までです。次のような場合、配列の次元より少ない行数が戻されます。

- アクティブ・セットの最後に達したとき:「データが見つかりません。」「データが見つかりません」というOracleの警告コードがSQLCA内のSQLCODEに戻されます。たとえば、100次元の配列に行をフェッチして、20行しか戻されなかった場合にこれが発生します。
- 残っているフェッチ対象の行が、一括フェッチの全行数より少ないとき。たとえば、20次元の配列に70行をフェッチすると、3回目のフェッチの後にはフェッチ対象の行が10しか残っていないため、この状態が発生します。
- 行の処理中にエラーが検出されたとき。フェッチは失敗し、該当するOracleエラー・コードがSQLCODEに戻されます。

戻された累積行数は、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)で確認できます。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、各カーソルの状態が個別に管理されている様子がわまります。

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
EXEC SQL FETCH cursor1 INTO :array_of_20;
 -- now running total in SQLERRD(3) is 20
EXEC SQL FETCH cursor2 INTO :array_of_30;
 -- now running total in SQLERRD(3) is 30, not 50
EXEC SQL FETCH cursor1 INTO :array_of_20;
 -- now running total in SQLERRD(3) is 40 (20 + 20)
EXEC SQL FETCH cursor2 INTO :array_of_30;
 -- now running total in SQLERRD(3) is 60 (30 + 30)
```

9.5.3 制限事項

SELECT文のWHERE句でホスト配列を使用できるのは、副問合においてのみです。(例はWHERE句の使用についてを参照してください。)また、SELECTまたはFETCH文のINTO句で、単純なホスト変数とホスト配列を混用することはできません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。表9-1では、SELECT INTO文でどのホスト配列の使用が有効かを示しています。

表9-1 SELECT INTOで有効なホスト配列

INTO句	WHERE句	有効?
配列	配列	いいえ
スカラー	スカラー	はい
配列	スカラー	はい
スカラー	配列	いいえ

9.5.4 NULLのフェッチについて

インジケータ配列のないホスト配列にNULLを選択またはフェッチすると、Oracleでは処理を停止し、SQLERRD(3)を処理済行数に設定して次のエラー・メッセージを発行します。

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

NULLや切り捨てられた値の見つけ方は、インジケータ変数の使用についてを参照してください。

9.5.5 切り捨てられた値のフェッチについて

DBMS=V7の場合、インジケータ配列のないホスト配列に切り捨てられた列値を選択またはフェッチすると、Oracleでは処理を停止し、SQLERRD(3)を処理済行数に設定して次のエラー・メッセージを発行します。

```
ORA-01406: fetched column value was truncated
```

SQLERRD(3)を調べると、切捨てが行われるまでに処理された行数がわかります。処理済行数には切捨てエラーが発生した行も含まれます。

MODE=ANSIの場合、切捨てはエラーとはみなされません。したがって、Oracleでは処理を続行します。

ここでも、配列の選択およびフェッチを行うときには、必ずインジケータ配列を使用してください。そうすれば、Oracleで1つ以上の切り捨てられた列値が出力ホスト配列に割り当てられた場合に、関連付けられたインジケータ配列で列値の元の長さがわかります。

9.5.6 配列での挿入について

ホスト配列は、INSERT文で入力変数として使用できます。INSERT文を実行する前に、プログラムで配列にデータが含まれているかどうかを確認してください。配列内に不適切な要素がある場合、FOR句を使用して挿入される行数を制御できます。FOR句の使用についてを参照してください。

ホスト配列による挿入の例は次のとおりです。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (50) CHARACTER(20);
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name, :emp_number, :salary);
```

挿入された累計行数は、SQLERRD(3)で確認できます。

機能的には次の文と同等ですが、前の例のINSERT文の方が、Oracleへのコールが1回のみであるためはるかに効率的です。

```
FOR i = 1 TO array_dimension
 EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name[i], :emp_number[i], :salary[i]);
ENDFOR;
```

この仮想の例(SQL文ではホスト変数に添字を付けられないため仮想)では、FORループを使用して、すべての配列要素に順次アクセスします。

制限事項

INSERT文のVALUES句内ではポインタ配列は使用できません。つまり配列要素はすべてデータ項目である必要があります。また、INSERT文のVALUES句で、単純なホスト変数とホスト配列を混用することはできません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。

9.5.7 配列での更新について

次の例に示すように、ホスト配列をUPDATE文の入力変数としても使用できます。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (50) INTEGER;
 salary (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL UPDATE EMP SET SAL = :salary WHERE EMPNO = :emp_number;
```

更新された累計行数は、SQLERRD(3)で確認できます。この数には更新カスケードによって処理された行は含まれていません。

配列の要素に不適切なものがある場合、FOR句を使用して更新される行数を制限できます。

前の例では、一意のキー(*emp_number*)を使用した一般的な更新を示しています。各配列要素で更新できる行は1行のみです。次の例では、各配列要素で複数の行を操作できます。

```
EXEC SQL BEGIN DECLARE SECTION;
 job_title (10) CHARACTER(10);
 commission (50) REAL;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
EXEC SQL UPDATE EMP SET COMM = :commission WHERE JOB = :job_title;
```

制限事項: UPDATE文のSET句またはWHERE句では、単純ホスト変数とホスト配列を併用できません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。さらに、SET句でホスト配列を使用する場合は、WHERE句のホスト配列を使用する必要があります。ただし、それらの次元やデータ型を一致させる必要はありません。

UPDATE文のCURRENTOF句では、ホスト配列は使用できません。かわりの方法については、CURRENT OF句の擬似実行についてを参照してください。

表9-2では、UPDATE文でどのホスト配列の使用が有効かを示しています。

表9-2 UPDATEで有効なホスト配列

SET句	WHERE句	有効?
配列	配列	はい
スカラー	スカラー	はい
配列	スカラー	いいえ
スカラー	配列	いいえ

9.5.8 配列での削除について

DELETE文でもホスト配列を入力変数として使用できます。これは、WHERE句内のホスト配列の連続した要素を使用して、DELETE文を繰り返し実行するのと同様です。つまり1回の実行で表から0行、1行または複数行が削除されます。ホスト配列による削除の例は次のとおりです。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 emp_number (50) INTEGER;
 EXEC SQL END DECLARE SECTION;
-- populate the host array
EXEC SQL DELETE FROM EMP WHERE EMPNO = :emp_number;
```

削除された行の累計数は、SQLERRD(3)を調べます。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。

前の例では、一意のキー(*emp_number*)を使用した一般的な削除を示しています。各配列要素で削除できる行は1行のみです。次の例では、各配列要素で複数の行を操作できます。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 job_title (10) CHARACTER(10);
EXEC SQL END DECLARE SECTION;
-- populate the host array
EXEC SQL DELETE FROM EMP WHERE JOB = :job_title;
```

9.5.9 制限事項

DELETE文のWHERE句で、単純なホスト変数とホスト配列を混用することはできません。ホスト変数のうち1つでも配列があれば、すべてのホスト変数を配列にする必要があります。また、DELETE文のCURRENT OF句ではホスト配列は使用できません。かわりの方法については、CURRENT OF句の擬似実行についてを参照してください。

9.6 インジケータ配列の使用方法について

入力ホスト配列へのNULLの割当て、および入力ホスト変数でのNULLまたは切り捨てられた値の検出には、インジケータ配列を使用します。次の例は、インジケータ配列による挿入の方法を示しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (50) INTEGER;
 dept_number (50) INTEGER;
```

```
 commission (50) REAL;
 ind_comm (50) SMALLINT; -- indicator array
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
-- populate the indicator array; to insert a null into
-- the COMM column, assign -1 to the appropriate element in
-- the indicator array
EXEC SQL INSERT INTO EMP (EMPNO, DEPTNO, COMM)
 VALUES (:emp_number, :dept_number, :commission:ind_comm);
```

インジケータ配列の次元は、ホスト配列の次元より大きくする必要があります。

9.7 FOR句の使用方法について

オプションのFOR句を使用すれば、次のSQL文で処理される配列要素の数を設定できます。

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

FOR句は特にUPDATE文、INSERT文およびDELETE文で役に立ちます。これらの文では、配列全体を使用する必要のない場合があります。FOR句を使用すると、次の例のように、使用する要素を必要な数に制限できます。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (100) CHARACTER(20);
 salary (100) REAL;
 rows_to_insert INTEGER;
EXEC SQL END DECLARE SECTION;
-- populate the host arrays
set rows_to_insert = 25; -- set FOR-clause variable
EXEC SQL FOR :rows_to_insert -- will process only 25 rows
 INSERT INTO EMP (ENAME, SAL)
 VALUES (:emp_name, :salary);
```

FOR句では、配列要素数をカウントするために整数型のホスト変数を使用する必要があります。たとえば、次の文は無効です。

```
EXEC SQL FOR 25 -- illegal
 INSERT INTO EMP (ENAME, EMPNO, SAL)
 VALUES (:emp_name, :emp_number, :salary);
```

FOR句の変数は、処理する配列要素数を指定します。この数は、最小の配列次元を超えないように設定します。また、この数は正数であることが必要です。負数またはゼロを指定すると、行は処理されません。

9.7.1 制限事項

2つの制限事項により、FOR句の意味が明確になります。FOR句はSELECT文では使用できません。また、CURRENT OF句とも使用することはできません。

9.7.2 SELECT文での使用

SELECT文でFOR句を使用すると、次のエラー・メッセージが表示されます。

```
PCC-E-0056: FOR clause not allowed on SELECT statement at ...
```

意味が不明確なため、SELECT文ではFOR句を使用できません。それは「このSELECT文を*n*回実行する」という意味でしょうか?それとも、「このSELECT文を実行するのは1度だが、*n*行戻す」という意味でしょうか。問題は、前者の場合、実行のたびに複数の行が戻される可能性があることです。後者の場合、カーソルを宣言し、次のように、FETCH文でFOR句を使用した方がよいでしょう。

```
EXEC SQL FOR :limit FETCH emp_cursor INTO ...
```

9.7.3 CURRENT OF句との併用

次の例のように、UPDATE文またはDELETE文でCURRENT OF句を使用すると、FETCH文によって戻される最後の行を参照できます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, SAL FROM EMP WHERE EMPNO = :emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
EXEC SQL UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
```

ただし、FOR句とCURRENT OF句は併用できません。次の文は、*limit*の論理値が1に限定されているため無効です(つまり、現在の行を更新または削除できるのは1回のみです)。

```
EXEC SQL FOR :limit UPDATE EMP SET SAL = :new_salary
 WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM EMP
 WHERE CURRENT OF emp_cursor;
```

9.8 WHERE句の使用方法について

Oracleでは、要素数*n*のホスト配列を含むSQL文を、同じSQL文を*n*個の異なるスカラー変数(個々の配列要素)で*n*回実行するのと同様に扱います。このように扱いがあいまいなときにかぎり、プリコンパイラから次のエラー・メッセージが発行されます。

```
PCC-S-0055: Array <name> not allowed as bind variable at ...
```

たとえば、次のような宣言をしたとします。

```
EXEC SQL BEGIN DECLARE SECTION;
 mgr_number (50) INTEGER;
 job_title (50) CHARACTER(20);
EXEC SQL END DECLARE SECTION;
```

次の文の場合、あいまいになります。

```
EXEC SQL SELECT MGR INTO :mgr_number FROM EMP
 WHERE JOB = :job_title;
```

次の仮想の文のように処理されるためです。

```
FOR i = 1 TO 50
 SELECT MGR INTO :mgr_number[i] FROM EMP
 WHERE JOB = :job_title[i];
ENDFOR;
```

これは、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用できる出力変数は1つしかないためです。したがって、エラー・メッセージが出力されます。

しかし、次の文の場合、あいまいになりません。

```
EXEC SQL UPDATE EMP SET MGR = :mgr_number
 WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE JOB = :job_title);
```

次の仮想の文のように処理されるためです。

```
FOR i = 1 TO 50
 UPDATE EMP SET MGR = :mgr_number[i]
 WHERE EMPNO IN
 (SELECT EMPNO FROM EMP WHERE JOB = :job_title[i]);
ENDFOR;
```

これは、各*job_title*が複数の行に一致する場合でも、WHERE句の*job_title*に一致する行ごとにSET句内で*mgr_number*が指定されているためです。各*job_title*に一致するすべての行に、同じ*mgr_number*をSETできます。したがって、エラー・メッセージは表示されません。

9.9 CURRENT OF句の疑似実行について

DELETE文またはUPDATE文でCURRENT OF *cursor*句を使用すると、カーソルから最後にフェッチされた行を参照できます。ただし、CURRENT OF句とホスト配列の併用はできません。かわりに各行のROWIDを取得し、更新または削除するときにその値を使用して現在行を識別してください。次に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_name (25) CHARACTER(20);
 job_title (25) CHARACTER(15);
 old_title (25) CHARACTER(15);
 row_id (25) CHARACTER(18);
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ENAME, JOB, ROWID FROM EMP;
...
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
...
LOOP
 EXEC SQL FETCH emp_cursor
 INTO :emp_name, :job_title, :row_id;
 ...
 EXEC SQL DELETE FROM EMP
 WHERE JOB = :old_title AND ROWID = :row_id;
 EXEC SQL COMMIT WORK;
ENDLOOP;
```

ただし、フェッチされた行は、FOR UPDATE OF句が使用されていないため、ロックされません。したがって、読み取った行を削除する前に、別のユーザーがその行を変更すると、結果が一貫性のないものになる可能性があります。

9.10 SQLERRD(3)の使用方法について

INSERT文、UPDATE文、DELETE文およびSELECT INTO文では、SQLERRD(3)は処理された行数を記録します。FETCH文では、処理された行の累積合計を記録します。

ホスト配列をFETCH文で使用するとき、最後の反復で戻された行数は、SQLERRD(3)の前の値(別の変数に格納)から現在の値を引けばわかります。次の例では、最後のフェッチで戻された行数を確認します。

```
EXEC SQL BEGIN DECLARE SECTION;
 emp_number (100) INTEGER;
 emp_name (100) CHARACTER(20);
EXEC SQL END DECLARE SECTION;
...
 rows_to fetch INTEGER;
 rows_before INTEGER;
 rows_this_time INTEGER;
...
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT EMPNO, ENAME
 FROM EMP
 WHERE DEPTNO = 30;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
-- initialize loop variables
set rows_to_fetch = 20; -- number of rows in each "batch"
set rows_before = 0; -- previous value of sqlerrd(3)
set rows_this_time = 20;
WHILE rows_this_time = rows_to_fetch
 LOOP
 EXEC SQL FOR :rows_to_fetch
 FETCH emp_cursor
 INTO :emp_number, :emp_name;
 set rows_this_time = sqlca.sqlerrd(3) - rows_before;
 set rows_before = sqlca.sqlerrd(3);
 ENDLOOP;
ENDWHILE;
```

SQLERRD(3)は、配列の操作中にエラーが発生した場合にも便利です。処理はエラーを引き起こした行で停止するため、SQLERRD(3)を調べることによって正常に処理された行数がわかります。

10 動的SQLの使用方法

この章の項目は次のとおりです。

この章では、アプリケーションに柔軟性や機能性をもたらす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。それぞれの方法の要件および制限事項、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。

10.1 動的SQL

ほとんどのデータベース・アプリケーションでは、特定のジョブが実行されます。たとえば、ユーザーに従業員番号の入力を要求して、その後EMPおよびDEPTの表の行を更新するという単純なプログラムがあります。この場合、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、変更される表、各表および列に定義されている制約、更新される列、各列のデータ型がわかっています。

しかし、アプリケーションによっては、様々なSQL文を実行時に受け入れ(または作成し)、処理する必要があります。たとえば、汎用レポート・ライターでは、生成する各種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。そこでこれらを動的SQL文と呼びます。

静的SQL文とは異なり、動的SQL文はソース・プログラム内には埋め込まれません。そのかわり、これらの文は実行時にプログラムに入力される(またはプログラムによって作成される)文字列に格納されます。対話形式で入力することも、ファイルから読み取ることもできます。

10.2 動的SQLの長所および短所

単純な埋込み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・プログラムは汎用性が高くなります。動的SQL文は、SQLの知識がほとんどないユーザーでも対話形式で作成できます。

たとえば、SELECT文、UPDATE文またはDELETE文のWHERE句内で使用する検索条件の入力をユーザーに求めるという単純なプ

ログラムがあります。さらに複雑なプログラムでは、SQL処理、表およびビューの名前、列の名前などを表示されているメニューからユーザーが選択できるようになります。このように、動的SQLを使用すると柔軟性に富んだアプリケーションを記述できます。

ただし、動的問合せの中には、複雑なコーディング、特殊なデータ構造体の使用および実行時の処理の増加が必要なものもあります。処理時間が増えることは支障がない場合もありますが、動的SQLの概念および方法を完全に理解するまではコーディングが難しく感じられることもあります。

10.3 動的SQLの使用

実際は静的SQLによって、プログラミング要件のほとんどを満たすことができます。動的SQLは、その高度な柔軟性が必要とされる場合にのみ使用してください。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(列、索引、順序、表、ユーザー名、ビューなど)の参照

10.4 動的SQL文の要件

動的SQL文を表すには、文字列に有効なSQL文のテキストを含める必要がありますが、EXEC SQL句、ホスト言語のデリミタまたは文の終了記号または次の埋込みコマンドは含めないでください。

- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- INCLUDE
- OPEN
- PREPARE
- WHENEVER

ほとんどの場合、この文字列にはダミーのホスト変数が含まれます。これらはSQL文内に実際のホスト変数のための場所を確保します。ダミーのホスト変数はプレースホルダにすぎないので、宣言する必要はなく、任意の名前を指定できます。たとえば、Oracleでは次の2つの文字列は区別されません。

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

10.5 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、ユーザーにSQL文のテキストおよびその文で使用するホスト変数の値の入力を要求します。そのSQL文はOracleにより解析されます。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。解析には、データベース・アクセス権のチェック、必要なリソースの予約、最適なアクセス・パスの確認も含まれます。

次に、このホスト変数がOracleによりSQL文にバインドされます。つまり、Oracleではホスト変数のアドレスを取得するため、値

の読取りまたは書込みができます。

この後、OracleでSQL文が実行されます。つまり、そのSQL文の要求(表からの行の削除など)をOracleが実行します。

これらのホスト変数に別の値を指定すると、このSQL文を繰り返し実行できます。

10.6 動的SQLの使用方法

この項では、動的SQL文の定義に使用できる4つの方法を紹介します。まずそれぞれの方法の機能および制限事項を簡単に説明した後、適切な方法を選択するためのガイドラインを示します。その後の項では、これらの方法の使用方法について説明します。また、このマニュアルの各ホスト言語の補足資料には、サンプル・プログラムが記載されています。

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10-1に示すように、それぞれの方法は、特定の種類のSQL文の処理に適しています。

表10-1 動的SQLの使用方法の適用性

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および入力ホスト変数の数がわかっている問合せ
4	選択リストの項目の数または入力ホスト変数の数が不明な問合せ

選択リスト項目には、列名や式を使用します。

10.6.1 方法1

この方法では、動的SQL文を受け入れるかまたは作成し、 EXECUTE IMMEDIATEコマンドを使用してその文をすぐに実行できます。このSQL文では、問合せ(SELECT文)の使用や、入力ホスト変数に対するプレースホルダの組込みはできません。たとえば、次のホスト文字列は有効です。

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

方法1では、SQL文が実行のたびに解析されます(HOLD_CURSOR=YESと指定した場合を除く)。

10.6.2 方法2

この方法を使用すると、プログラムは動的SQL文を受け入れ(または作成し)、PREPAREおよびEXECUTEコマンドを使用してその文を処理します。SQL文は、問合せにはできません。プリコンパイル時に、入力ホスト変数のプレースホルダの数および入力ホスト変数のデータ型を明確にする必要があります。たとえば、次のホスト文字列はこのカテゴリに入ります。

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

方法2では、SQL文の解析は1回しか行われませんが(RELEASE_CURSOR=YESと指定している場合を除く)、ホスト変数に異なる値を指定すれば、複数回実行できます。SQLデータ定義文(CREATEなど)は、PREPAREの際に実行されます。

10.6.3 方法3

この方法を使用すると、プログラムは動的問合せを受け入れるか、または作成し、DECLARE、 OPEN、FETCHおよびCLOSE カーソル・コマンドを使用してその文を処理します。プリコンパイル時に、選択リストの項目数、入力ホスト変数のプレース・ホルダ数および入力ホスト変数のデータ型がわかっている必要があります。たとえば、次のホスト文字列は有効です。

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number'
```

10.6.4 方法4

この方法を使用すると、プログラムは動的SQL文を受け入れるか、または作成し、記述子(「方法4の使用方法」を参照)を使用してその文を処理します。選択リストの項目数、入力ホスト変数のプレース・ホルダー数および入力ホスト変数のデータ型は、実行時まで不明でもかまいません。たとえば、次のホスト文字列はこのカテゴリに入ります。

```
'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。

10.6.5 ガイドライン

4つの方法はどれも、動的SQL文を文字列に格納する必要があり、この文字列は、ホスト変数または引用符で囲んだリテラルであることが必要です。SQL文を文字列に格納する場合は、キーワードEXEC SQLおよび文の終了記号は省略してください。

方法2および方法3では、プリコンパイル時に入力ホスト変数のプレース・ホルダ数および入力ホスト変数のデータ型がわかっている必要があります。

方法の番号が大きくなるほどアプリケーションへの制約は少なくなりますが、コードの記述が難しくなります。通常は、最も簡単な方法を使用してください。ただし、動的SQL文を繰り返し実行する場合、方法1では実行のたびに再解析されるので、これを避けるために方法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のみです。図10-1の決定論理を参考にして、適切な方法を選択してください。

10.6.6 一般的なエラーの回避

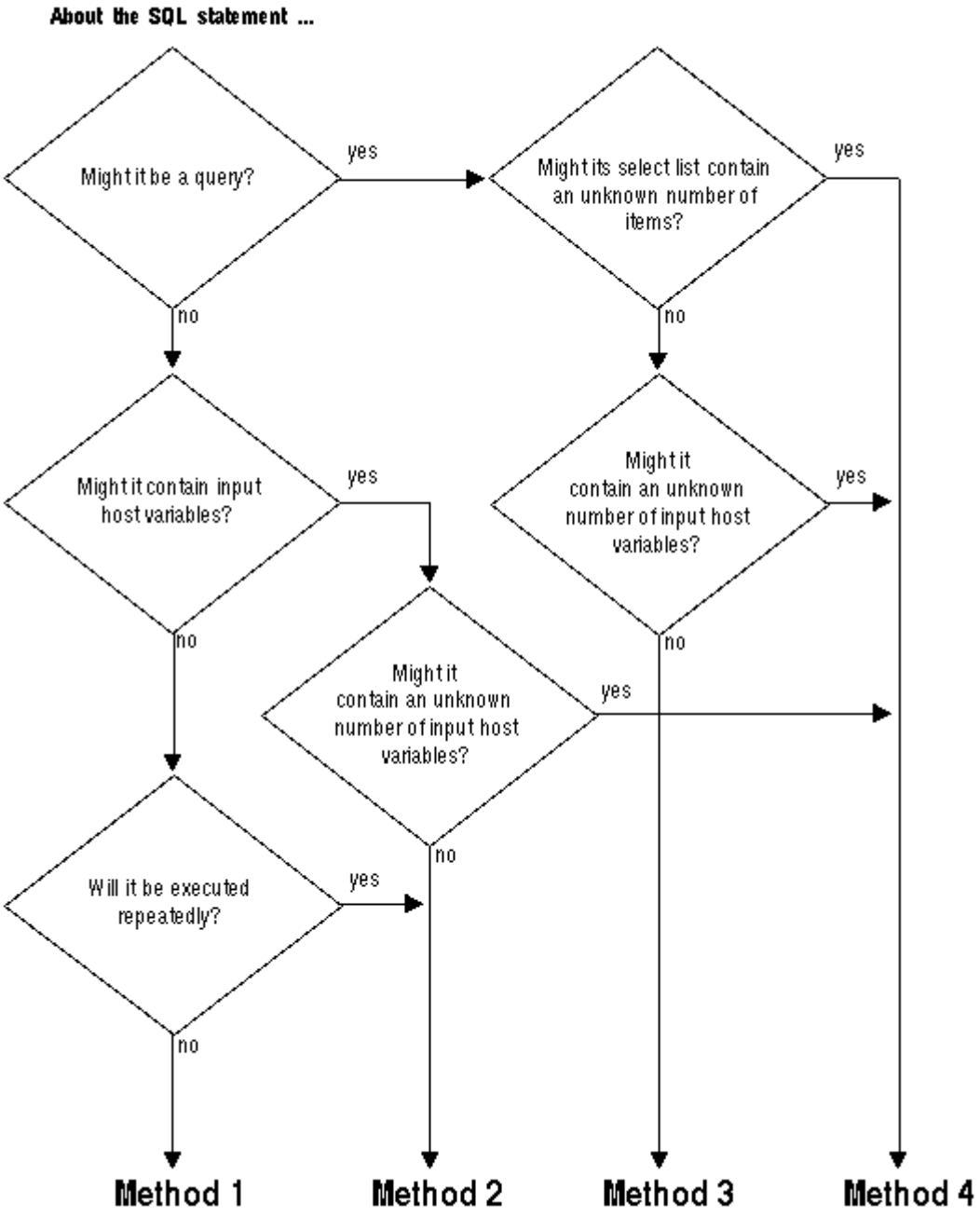
動的SQL文を文字配列に格納する場合は、その配列に空白を埋め込んでからSQL文を格納してください。こうして余分な文字をクリアします。別のSQL文を格納するために配列を再利用するときに、この処理が重要となります。一般に、SQL文を格納する前には必ずホスト文字列を初期化(または再初期化)してください。

ホスト文字列にはヌル終端文字を使用しないでください。OracleではNULL終了文字は文字列の終了マーカーとはみなされません。SQL文の一部として扱われます。

VARCHAR変数を使用して動的SQL文を格納する場合、VARCHARの長さが正しく設定(または再設定)されていること確認してから、PREPARE文またはEXECUTE IMMEDIATE文を実行してください。

EXECUTEを実行すると、SQLCA内のSQLWARN警告フラグはリセットされます。したがって、無条件の更新(WHERE句の省略により発生)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フラグをチェックしてください。

図10-1 正しい方法の選択



10.7 方法1の使用方法について

最も単純な種類の動的SQL文の結果は成功または失敗のみで、ホスト変数は使用されません。次に例を示します。

```
'DELETE FROM table_name WHERE column_name = constant'
'CREATE TABLE table_name ...'
'DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
'REVOKE RESOURCE FROM username'
```

10.7.1 EXECUTE IMMEDIATE文

方法1では、SQL文を解析すると、EXECUTE IMMEDIATEコマンドを使用してすぐに実行します。コマンドの後には、実行するSQL文を含む文字列(ホスト変数またはリテラル)が続きますが、この文は問合せにしないでください。

EXECUTE IMMEDIATE文の構文は次のとおりです。

```
EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };
```

次の例では、ホスト変数*sql_stmt*を使用してユーザーが入力するSQL文を格納しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 sql_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
...
LOOP
 display 'Enter SQL statement: ';
 read sql_stmt;
 IF sql_stmt is empty THEN
 exit loop;
 ENDIF;
 -- sql_stmt now contains the text of a SQL statement
 EXEC SQL EXECUTE IMMEDIATE :sql_stmt;
ENDLOOP;
```

次の例のように、文字列リテラルを使用することもできます。

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

EXECUTE IMMEDIATEは入力されているSQL文を実行するたびに解析するため、方法1は1回しか実行しない文に最も適しています。通常、データ定義文は、このカテゴリに入ります。

10.7.2 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を使用して文を実行します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 update_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
 search_cond CHARACTER(40);
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set update_stmt = 'UPDATE EMP SET COMM = 500 WHERE ';
display 'Enter a search condition for the following statement:';
display update_stmt;
read search_cond;
concatenate update_stmt, search_cond;
EXEC SQL EXECUTE IMMEDIATE :update_stmt;
EXEC SQL COMMIT WORK RELEASE;
exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 display 'Processing error';
 exit program with an error;
```

10.8 方法2の使用方法について

方法1では1ステップで実行することを、方法2では2ステップに分けて実行します。動的SQL文(問合せは不可)は、まずPREPARE(名前の指定および解析)され、その後実行されます。

方法2では、SQL文に入力ホスト変数およびインジケータ変数のプレースホルダを含めることができます。このSQL文は一度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。また、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を除く)。

方法4では非問合せにEXECUTEを使用できます。

PREPARE文の構文は次のとおりです。

```
EXEC SQL PREPARE statement_name
 FROM { :host_string | string_literal };
```

PREPAREはSQL文を解析して名前を指定します。

*statement_name*は、ホスト変数やプログラム変数ではなく、プリコンパイラで使用される識別子であり、宣言部で宣言されません。これはEXECUTEの対象としてPREPARE済の文を示しているにすぎません。

EXECUTE文の構文は次のとおりです。

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

*host_variable_list*は、次の構文を表しています。

```
:host_variable1[:indicator1] [, host_variable2[:indicator2], ...]
```

解析されたSQL文は、各入力変数に指定した値を使用してEXECUTEにより実行されます。次の例では、入力されたSQL文に、プレースホルダ*n*が含まれています。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 emp_number INTEGER;
 delete_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
 search_cond CHARACTER(40);
...
set delete_stmt = 'DELETE FROM EMP WHERE EMPNO = :n AND ';
display 'Complete the following statement's search condition:';
display delete_stmt;
read search_cond;
concatenate delete_stmt, search_cond;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
LOOP
 display 'Enter employee number: ';
 read emp_number;
 IF emp_number = 0 THEN
 exit loop;
 EXEC SQL EXECUTE sql_stmt USING :emp_number;
ENDLOOP;
```

方法2では、プリコンパイル時に入力ホスト変数のデータ型がわかっている必要があります。前の例では、*emp_number*がINTEGER型として宣言されています。これは、CHARACTER型やREAL型としても宣言できますが、これはOracleではこれらすべてのデータ型のNUMBERデータ型への変換がサポートされているためです。

10.8.1 USING句

SQL文がEXECUTEされると、USING句の入力ホスト変数は、PREPAREされた動的SQL文内の該当するプレースホルダに置換されます。

PREPAREされた動的SQL文内のプレースホルダはそれぞれ、USING句のホスト変数に対応している必要があります。したがって、PREPARE済の文に同じプレースホルダが2回以上現れる場合、それぞれがUSING句のホスト変数に対応している必要があります。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あることが必要です。

プレースホルダの名前は、ホスト変数の名前と一致する必要はありません。ただし、PREPARE済動的SQL文のプレースホルダの順序は、USING句の対応するホスト変数の順序と一致する必要があります。

NULLを指定するために、インジケータ変数をUSING句のホスト変数と関連付けることができます。詳細は、インジケータ変数の使用についてを参照してください

10.8.2 例

次のプログラムでは、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、その後方法2を使用してその文を準備し実行します。UPDATE文のSET句にはプレースホルダ(c)が含まれています。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 sql_stmt CHARACTER(80);
 empno INTEGER VALUE 1234;
 deptno1 INTEGER VALUE 97;
 deptno2 INTEGER VALUE 99;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
EXEC ORACLE OPTION (ORACA=YES);
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set sql_stmt =
 'INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:v1, :v2)';
display "V1 = ", empno, "V2 = ", deptno1;
EXEC SQL PREPARE S FROM :sql_stmt;
EXEC SQL EXECUTE S USING :empno, :deptno1;
set empno = empno + 1;
display "V1 = ", empno, "V2 = ", deptno2;
EXEC SQL EXECUTE S USING :empno, :deptno2;
set sql_stmt =
 'DELETE FROM EMP WHERE DEPTNO = :v1 OR DEPTNO = :v2")';
display "V1 = ", deptno1, "V2 = ", deptno2;
EXEC SQL PREPARE S FROM :sql_stmt;
EXEC SQL EXECUTE S USING :deptno1, :deptno2;
EXEC SQL COMMIT WORK RELEASE;
exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 display 'Processing error';
 EXEC SQL ROLLBACK WORK RELEASE;
 exit program with an error;
```

10.9 方法3の使用方法について

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操作に必要な文と結合する点が異なります。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、動的SQL文が問合せの場合、方法3または4を使用する必要があります。

方法3では、プリコンパイル時に、問合せ選択リストの列数と入力ホスト変数のプレースホルダの数がわかっている必要があります。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およびORDER BYなど)も、実行時に指定できます。

方法3では、埋込みSQL文を次のような順序で使用します。

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

次に、それぞれの文の実行内容を説明します。

10.9.1 PREPARE

PREPAREは動的SQL文を解析し、名前を指定します。次の例では、PREPAREは文字列*select_stmt*に格納されている問合せを解析し、これに*sql_stmt*という名前を指定します。

```
set select_stmt = 'SELECT MGR, JOB FROM EMP WHERE SAL < :salary';
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

通常、問合せのWHERE句は、実行時に端末から入力するか、アプリケーションによって生成されます。

識別子*sql_stmt*は、ホスト変数でもプログラム変数でもありませんが、一意にする必要があります。sql_stmtは特定の動的SQL文を指定します。

10.9.2 DECLARE

DECLAREは、カーソルに名前を指定し、これを特定の問合せに関連付けてカーソルを定義します。カーソルの宣言は、そのプリコンパイル・ユニット内でのみ有効です。次の例では、DECLAREにより*emp_cursor*という名前のカーソルを定義し、それを*sql_stmt*に関連付けています。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

識別子*sql_stmt*および*emp_cursor*は、ホスト変数でもプログラム変数でもありませんが、一意であることが必要です。同じ文名を使用して2つのカーソルを宣言すると、プリコンパイラではこの2つのカーソル名を同義とみなします。たとえば次の文を実行したとします。

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept_cursor FOR sql_stmt;
```

この場合、*emp_cursor*をOPENすると、処理の対象となるのは*delete_stmt*に格納されている動的SQL文で、*select_stmt.*に格納されている動的SQL文ではありません。

10.9.3 OPEN

OPENは、アクティブ・セットを識別して、Oracleカーソルを割り当て、入力ホスト変数にバインドし、問合せを実行します。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行数を0 (ゼロ)に設定します。USING句の入力ホスト変数は、PREPARE済動的SQL文内の対応するプレースホルダに置き換わります。

例では、次に示すように、OPENにより*emp_cursor*を割り当て、ホスト変数*salary*をWHERE句に割り当てます。

```
EXEC SQL OPEN emp_cursor USING :salary;
```

10.9.4 FETCH

FETCHは、アクティブ・セットから行を戻し、選択リスト内の列値をINTO句の対応するホスト変数に割り当てた後、カーソルを次の行に進めます。行がなくなると、FETCHは「データが見つかりません」というOracleエラー・コードをSQLCA内のSQLCODEに戻します。

例では、次に示すように、FETCHによってアクティブ・セットから1行戻し、MGR列とJOB列の値をホスト変数の*mgr_number*と*job_title*に割り当てます。

```
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
```

10.9.5 CLOSE

CLOSEは、カーソルを無効にします。カーソルをCLOSEすると、そこからのFETCHはできなくなります。例では次のように、CLOSE文により*emp_cursor*が無効になります。

```
EXEC SQL CLOSE emp_cursor;
```

10.9.6 例

次のプログラムでは、問合せのWHERE句で使用する検索条件の入力をユーザーに求めてから、方法3を使用して問合せを準備し、実行します。

```
EXEC SQL BEGIN DECLARE SECTION;
 username CHARACTER(20);
 password CHARACTER(20);
 dept_number INTEGER;
 emp_name CHARACTER(10);
 salary REAL;
 select_stmt CHARACTER(120);
EXEC SQL END DECLARE SECTION;
 search_cond CHARACTER(40);
EXEC SQL INCLUDE SQLCA;
display 'Username? ';
read username;
display 'Password? ';
read password;
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
display 'Connected to Oracle';
set select_stmt = 'SELECT ENAME,SAL FROM EMP WHERE ';
display 'Enter a search condition for the following statement:';
display select_stmt;
read search_cond;
concatenate select_stmt, search_cond;
```

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
display 'Employee Salary';
display '-------- ------';
LOOP
 EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
 display emp_name, salary;
ENDLOOP;
no_more:
 EXEC SQL CLOSE emp_cursor;
 EXEC SQL COMMIT WORK RELEASE;
 exit program;
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 exit program with an error;
```

10.10 方法4の使用方法

方法4の実装は、言語によって非常に異なります。したがって、この項では概要のみを説明します。詳細は、使用するホスト言語の補足資料を参照してください。

方法3を使用したプログラムでも処理できない種類の動的SQL文があります。選択リストの項目数または入力ホスト変数のプレースホルダの数が実行時までわからない場合は、プログラムで記述子を使用する必要があります。記述子とは、プログラムおよびOracleによって動的SQL文内の変数の完全な記述を保存するために使用されるメモリー領域です。

複数行の問合せの場合、宣言済の出力ホスト変数のリストに選択した列値をFETCH INTOしたことを思い出してください。この選択リストがわからない場合は、プリコンパイル時にINTO句でホスト変数リストを作成できません。たとえば、次の問合せでは2つの列値が戻されます。

```
SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number;
```

ただし、この選択リストをユーザーに定義させると、その問合せによって戻される列の数はわからなくなります。

10.10.1 SQLDAの必要性

この種の動的問合せを処理するには、プログラムでDESCRIBE SELECT LISTコマンドを発行し、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存するため、選択記述子とも呼ばれます。

同様に、動的SQL文に含まれる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には、プリコンパイル時にUSING句によってホスト変数リストを作成できません。

このような動的SQL文を処理するには、プログラムでDESCRIBE BIND VARIABLESコマンドを発行し、入力ホスト変数のプレースホルダの説明を保存するために、バインド記述子という別の種類のSQLDAを宣言する必要があります。(入力ホスト変数はバインド変数とも呼ばれます。)

プログラムにアクティブSQL文が複数ある(たとえば、複数のカーソルをOPENしている)場合、それぞれの文には専用のSQLDAが必要になります。ただし、非並行のカーソルではSQLDAを再利用できます。なお、1つのプログラム内のSQLDAの数に制限はありません。

10.10.2 DESCRIBE文

DESCRIBEは、選択リストまたは入力ホスト変数の説明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、DESCRIBE SELECT LIST文によってPREPAREされた動的問合せ内の各選択リスト項目が調べられ、その名前、データ型、制約、長さ、位取りおよび精度が確認されます。その後、この情報は選択記述子に格納されます。

バインド記述子を指定すると、DESCRIBE BIND VARIABLES文によってPREPAREされた動的SQL文の各プレースホルダが調べられ、その名前および長さ、関連付けられた入力ホスト変数のデータ型が確認されます。続いて、この情報がそのバインド記述子に格納されます。たとえば、プレースホルダ名を使用して、ユーザーに入力ホスト変数の値の入力を要求できます。

10.10.3 SQLDA

SQLDAとは、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ホストプログラムのデータ構造体です。

SQLDA変数は、宣言部で定義されません。

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ています。

- 記述できる列の最大数
- 記述により検出された列の実際の数
- 列値を格納するバッファのアドレス
- 列値の長さ
- 列値のデータ型
- インジケータ変数の値のアドレス
- 列名を格納するバッファのアドレス
- 列名を格納するバッファのサイズ
- 列名の現行の長さ

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。

- 記述できるプレースホルダの最大数
- 記述で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
- 標識変数名の現行の長さ

特定のホスト言語のSQLDA構造体および変数の名前を調べるには、使用するホスト言語の補足資料を参照してください。

10.10.4 方法4の実行

方法4では、一般に次の順序で埋込みSQL文を使用します。

```
EXEC SQL PREPARE statement_name
 FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
 INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
 [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
 INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
 USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

選択記述子およびバインド記述子の両方を使用する必要はありません。問合せ選択リスト内の列数がわかっていて、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、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 emp_cursor INTO host_variable_list;
```

逆に、入力ホスト変数のプレースホルダの数がわかっていて、選択リストの列数がわからない場合は、次の方法3のOPEN文と方法4のFETCH文を併用できます。

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

方法4では、問合せ以外の文にEXECUTEを使用できるので注意してください。

これらの文によりプログラムで記述子を使用して動的SQL文をどのように処理できるかについては、使用するホスト言語の補足資料を参照してください。

10.11 DECLARE STATEMENT文の使用方法について

方法2、3および4では、次の文を使用することが必要な場合があります。

```
EXEC SQL [AT db_name] DECLARE statement_name STATEMENT;
```

*db_name*および*statement_name*は、プリコンパイラで使用される識別子で、ホスト変数でもプログラム変数でもありません。

DECLARE STATEMENTは、PREPARE、EXECUTE、DECLARE CURSORおよびDESCRIBEが文を参照できるように、動的SQL文の名前を宣言します。デフォルト以外のデータベースで動的SQL文を実行する場合は必須です。方法2を使用する例を次に示します。

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

この例では、*remote_db*によって、SQL文をどこでEXECUTEするかをOracleに指示します。

方法3および方法4では、次の例に示すように、DECLARE CURSOR文がPREPARE文の前にある場合にもDECLARE STATEMENTが必要です。

```
EXEC SQL DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
```

一般的な文の順序は次のとおりです。

```
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

10.11.1 ホスト配列の使用方法

ホスト配列の使用方法は、静的SQLでも動的SQLでも同様です。たとえば、動的SQL方法2で入力ホスト配列を使用するには、次の構文を使用します。

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

*host_array_list*には1つ以上のホスト配列が含まれます。方法3の場合は、次の構文を使用します。

```
OPEN cursor_name USING host_array_list;
```

方法3で出力ホスト配列を使用するには、次の構文を使用します。

```
FETCH cursor_name INTO host_array_list;
```

方法4では、オプションのFOR句を使用して、入力ホスト配列または出力ホスト配列のサイズをOracleに指示する必要があります。この方法は、ホスト言語の補足資料を参照してください。

10.12 PL/SQLの使用について

Oracleプリコンパイラでは、PL/SQLブロックが1つのSQL文として扱われます。したがって、SQL文と同様に、PL/SQLブロックを文字列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場合、EXEC SQL EXECUTEキーワード、END-EXECキーワードおよび文の終了記号を省略します。

ただし、プリコンパイラによるSQLとPL/SQLの処理方法には、次の2つの違いがあります。

- PL/SQLホスト変数のPL/SQLブロック内での役割が入力ホスト変数、出力ホスト変数、あるいはその両方のどれであっても、プリコンパイラではPL/SQLホスト変数はすべて入力ホスト変数として扱われます。
- PL/SQLブロックに格納できるSQL文の数には制限がないため、PL/SQLブロックからはFETCHできません。

10.12.1 方法1の場合

PL/SQLブロックにホスト変数が含まれていない場合は、方法1で通常どおりPL/SQLをEXECUTEできます。

10.12.2 方法2の場合

PL/SQLブロック内の入力ホスト変数および出力ホスト変数の数がわかっている場合は、方法2で通常どおりPL/SQL文字列をPREPAREし、EXECUTEできます。

USING句にはすべてのホスト変数を指定する必要があります。このPL/SQL文字列をEXECUTEすると、USING句内のホスト変数はPREPARE済の文字列内の対応するプレースホルダに置き換わります。プリコンパイラでPL/SQLホスト変数がすべて入力ホスト変数として扱われても、値は正しく割り当てられます。入力(プログラム)値は入力ホスト変数に割り当てられ、出力(列)値は出力ホスト変数に割り当てられます。

PREPAREされたPL/SQL文字列中のプレースホルダは、それぞれUSING句のホスト変数に対応している必要があります。したがって、PREPARE済の文字列に同じプレースホルダが2回以上現れる場合、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。

10.12.3 方法3の場合

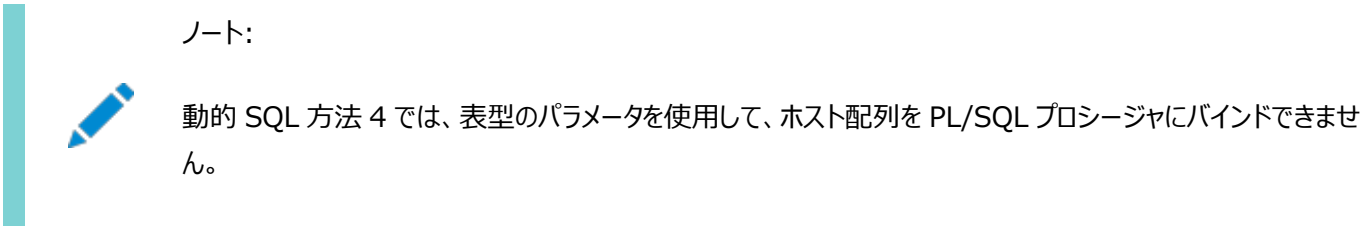
方法3は、FETCHができることを除けば、方法2と同じです。PL/SQLブロックからFETCHはできないので、かわりに方法2を使用してください。

10.12.4 方法4の場合

PL/SQLブロックに含まれる入力ホスト変数または出力ホスト変数の数がわからない場合は、方法4を使用してください。

方法4を使用するには、すべての入力ホスト変数および出力ホスト変数について1つのバインド記述子を設定します。DESCRIBE BIND VARIABLESを実行すると、入力ホスト変数および出力ホスト変数に関する情報がそのバインド記述子に格納されます。プリコンパイラではPL/SQLホスト変数がすべて入力ホスト変数として扱われるため、DESCRIBE SELECT LISTを実行しても効果はありません。

方法4でのバインド記述子の詳細は、ホスト言語補足を参照してください。



ノート:

動的 SQL 方法 4 では、表型のパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドできません。

10.12.5 注意

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ため、ANSI形式のコメント(- - ...)は使用しないでください。ANSI形式のコメントは、行の終わりではなく、ブロックの終わりまで続いてしまいます。かわりに、C形式のコメント(/* ... */)を使用してください。

11 ユーザー・イグジットの作成

この章は、次の項で構成されています。

この章では、SQL*FormsおよびOracle Formsアプリケーション用のユーザー・イグジットの作成について重点的に説明します。まず、SQL*Formsアプリケーションをユーザー・イグジットとのインタフェースにするの EXEC IAF文について学習します。次に、SQL*Formsユーザー・イグジットの作成方法とリンクの方法を学びます。また、EXEC TOOLS文をOracle Formsで使用する方法についても説明します。(SQL*FormsではEXEC TOOLSはサポートされていません。)このようにして、EXEC IAF文を使用することで、既存のアプリケーションやEXEC TOOLS文を強化し、新しいアプリケーションを作成できるようになります。内容は次のとおりです。

- ユーザー・イグジットの一般的な使用方法
- ユーザー・イグジットの作成
- SQL*Formsとユーザー・イグジット間の値の受渡し
- ユーザー・イグジットの実装
- ユーザー・イグジットのコール
- SQL*Formsユーザー・イグジットのガイドライン
- Oracle FormsでのEXEC TOOLS文の使用方法

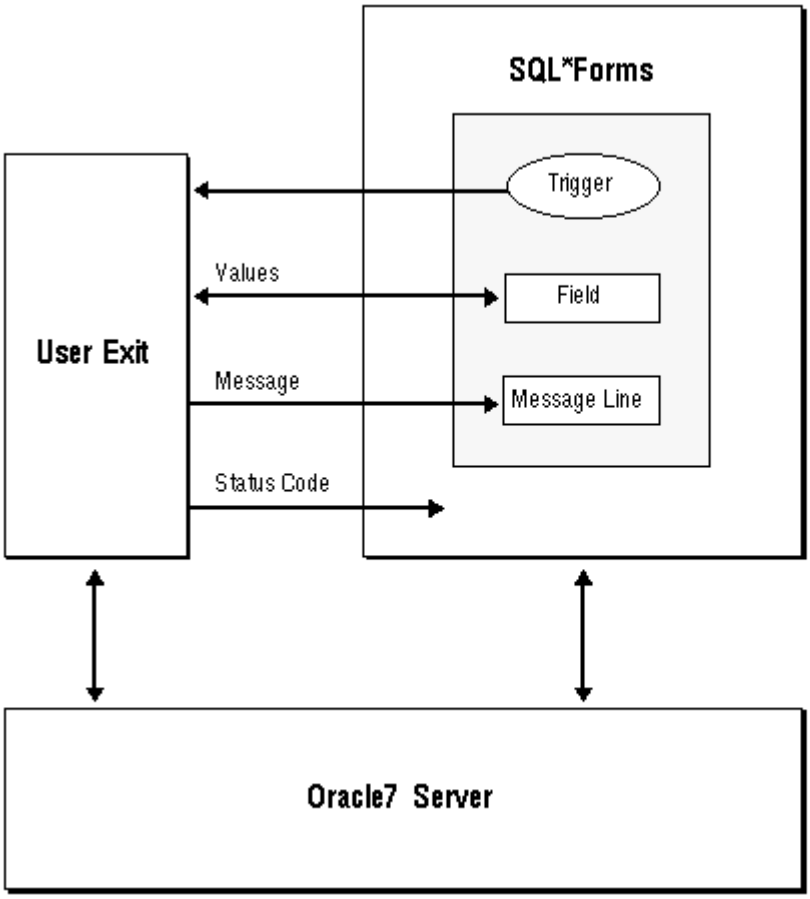
この章の内容は補足説明です。ユーザー・イグジットの詳細は、*Oracle Forms*リファレンス・マニュアル、*Vol. 2*の「*SQL*Forms*開発者リファレンス」および各システム固有のOracleマニュアルを参照してください。

11.1 ユーザー・イグジット

ユーザー・イグジットとは、特別な目的の処理を実行するために作成するホスト言語のサブルーチンのことで、SQL*Formsによってコールされます。ユーザー・イグジットにSQLコマンドやPL/SQLブロックを埋め込み、ホスト・プログラムの場合と同様にこれをプリコンパイルできます。

SQL*Formsトリガーからコールすると、ユーザー・イグジットが実行され、SQL*Formsにステータス・コードが戻されます(図11-1を参照)。ユーザー・イグジットでは、SQL*Forms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、Oracleデータの操作、高速計算および表参照、さらには別のデータベースへの接続もできます。

図11-1 SQL*Forms



11.2 ユーザー・イグジットを作成する理由

SQL*Formsバージョン3では、トリガーでPL/SQLブロックを使用できます。したがって、ほとんどの場合、ユーザー・イグジットをコールするかわりに、PL/SQLのプロシージャ機能を使用できます。ユーザー・イグジットが必要になった場合は、USER_EXIT関数を使用してPL/SQLブロックからコールできます。

SQL、PL/SQLまたはSQL*Formsのコマンドに比べて、ユーザー・イグジットは記述も実装も複雑です。したがって、SQL、PL/SQLおよびSQL*Formsの範囲を超える処理を実行する場合にかぎり、ユーザー・イグジットを使用することになります。一般的に次のような場合に使用します。

- CやFORTRANなどの第3世代の言語で実行するとより迅速または簡単になる演算(数値積分など)
- リアルタイムのデバイスまたは処理の制御(プリンタまたはグラフィックス・デバイスへの一連の命令の発行など)
- 拡張プロシージャ機能が必要なデータ操作(再帰ソートなど)
- 特殊なファイルI/O操作

11.3 ユーザー・イグジットの開発

この項ではSQL*Formsユーザー・イグジットの開発方法の概要を示します。詳細はこの後の項で説明します。Oracle Formsで利用できる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に組み込むには、次のステップに従います。

1. サポートされているホスト言語でユーザー・イグジットを記述します。

2. ソース・コードをプリコンパイルします。

3. プリコンパイルしたソース・コードをコンパイルします。

4. GENXTBユーティリティを使用してデータベース表IAPXTBを作成します。

5. SQL*FormsのGENXTBフォームを使用して、ユーザー・イグジット情報をデータベース表に挿入します。

6. GENXTBユーティリティを使用して表から情報を読み取り、IAPXITソース・モジュールを作成します。次に、ソース・モジュールをコンパイルします。

7. 標準IAPオブジェクト・モジュール、ユーザー・イグジット・オブジェクト・モジュール、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して、新しいIAP(フォームを実行するSQL*Formsコンポーネント)を作成します。

8. このフォームで、ユーザー・イグジットをコールするトリガーを定義します。

9. オペレータがフォームを実行する際には、新しいIAPを使用するように指示します。新しいIAPが標準のものに置き換わる場合、これは不要です。詳細は、各システム固有のOracleマニュアルを参照してください。

11.4 ユーザー・イグジットの作成

SQL*Formsユーザー・イグジットの作成には、次の文を使用できます。

- ホスト言語
- EXEC SQL
- EXEC ORACLE
- EXEC IAF GET
- EXEC IAF PUT

この項では、SQL*Formsとユーザー・イグジット間での値の受渡しを可能にするEXEC IAF GET文およびEXEC IAF EXEC IAF PUT文を中心に説明します。

11.4.1 変数の要件

EXEC IAF文で使用される変数は、フォーム定義で使用されるフィールド名に対応している必要があります。ブロック名を指定しなかったためにフィールド参照があいまいな場合は、エラーが発生します。フォーム・フィールドに無効な参照またはあいまいな参照を行うと、エラーが発生します。

ホスト変数の名前は、ユーザー・イグジットの宣言部で指定し、EXEC IAF文では前にコロン(:)を付ける必要があります。

ノート:

インジケータ変数は、EXEC IAF GET 文および EXEC IAF PUT 文では使用できません。

11.4.2 IAF GET文

この文を使用すると、ユーザー・イグジットでフォーム上のフィールドから値を取得し、ホスト変数に割り当てることができます。ユーザー・イグジットでは、それらの値を計算、データ操作、更新などに使用できます。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, ...;
```

*field_name*には、次のSQL*Forms変数のどれを指定してもかまいません。

- 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、システム変数またはグローバル変数の値を含むホスト変数(前にコロンが付く)

*field_name*は修飾されていない場合、一意であることが必要です。

次の例は、ユーザー・イグジットでどのようにフィールド値をGETし、ホスト変数に割り当てるかを示しています。

```
EXEC IAF GET employee.job INTO :new_job;
```

フィールド値はすべて文字列です。値を取得すると、GET文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。無効またはサポートされていないデータ型への変換が試みられると、エラーが発生します。

前の例では、*block.field*の指定に定数が使用されています。ブロック名およびフィールド名の指定には、次のようにホスト文字列も使用できます。

```
set blkfld = 'employee.job';
EXEC IAF GET :blkfld INTO :new_job;
```

フィールドが一意でない場合、ホスト文字列には間にピリオドの入った完全な*block.field*参照が含まれている必要があります。たとえば、次の使用方法は無効です。

```
set blk = 'employee';
set fld = 'job';
EXEC IAF GET :blk.:fld INTO :new_job;
```

1つのGET文のフィールド・リストに明示的フィールド名とストアド・フィールド名を混在させることはできますが、1つのフィールド参照ではできません。たとえば、次の使用方法は無効です。

```
set fld = 'job';
EXEC IAF GET employee.:fld INTO :new_job;
```

11.4.3 IAF PUT文

この文を使用すると、ユーザー・イグジットで定数およびホスト変数の値をフォームのフィールドに指定できます。このようにして、ユーザー・イグジットによりSQL*Formsの画面に任意の値やメッセージを表示できます。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, ...);
```

*field_name*には、次のSQL*Forms変数のどれを指定してもかまいません。

- 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、システム変数またはグローバル変数の値を含むホスト変数(前にコロンが付く)

次の例では、ユーザー・イグジットでどのように数字の定数、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PUTするかを示しています。

```
EXEC IAF PUT employee.number, employee.name, employee.job
 VALUES (7934, 'MILLER', :new_job);
```

GETと同様に、PUTでもホスト文字列を使用して、次のようにブロックやフィールドの名前を指定できます。

```
set blkfld = 'employee.job';
EXEC IAF PUT :blkfld VALUES (:new_job);
```

キャラクタモードの端末では、フィールドにPUTされた値は、フィールドが現在の表示ページ上にある場合、割り当てられたときではなく、ユーザー・イグジットから戻されたときに表示されます。ブロックモードの端末では、次回デバイスからフィールドが読み取られたときに値が表示されます。

ユーザー・イグジットで1つのフィールドの値が数回変更された場合、最後の変更のみが有効です。

11.5 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付属)というパッケージ・プロシージャを使用します。次の構文を使用します。

```
USER_EXIT(user_exit_string [, error_string]);
```

*user_exit_string*には、ユーザー・イグジットの名前とオプションのパラメータを指定し、*error_string*には、ユーザー・イグジットが異常終了したときにSQL*Formsにより発行されるエラー・メッセージを指定します。たとえば、次のトリガー・コマンドは、LOOKUPという名前のユーザー・イグジットをコールします。

```
USER_EXIT('LOOKUP');
```

ユーザー・イグジットの文字列は、一重(二重ではない)引用符で囲まれていることに注意してください。

11.6 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で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。

- コマンドラインは、ユーザー・イグジット文字列です。
- コマンドラインの長さは、ユーザー・イグジット文字列の長さ(文字数)です。
- エラー・メッセージは、定義されている場合、エラー文字列(障害メッセージ)です。
- エラー・メッセージの長さは、エラー文字列の長さです。
- 問合せモードは、ユーザー・イグジットのコールが通常モードと問合せモードのどちらで行われたかを示すブール値です。

ただし、ユーザー・イグジット文字列を使用すると、追加の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・コマンドでは、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');
```

この機能を使用すれば、次の例のように、フィールド名をユーザー・イグジットに渡せます。

```
USER_EXIT('CONCAT firstname, lastname, address');
```

ただし、ユーザー・イグジット文字列を解析は、SQL*Formsではなく、ユーザー・イグジットに依存します。

11.7 フォームへの値の返却

ユーザー・イグジットでは、SQL*Formsに制御が戻るとき、成功したか、失敗したかまたは致命的エラーが発生したかのいずれの状態かを示すコードも必ず戻します。このリターン・コーは、プリコンパイラによって生成される整数の定数です(この項の「例」を参照)。この3種類の結果はの意味は次のとおりです。

成功: ユーザー・イグジットでエラーが発生しませんでした。コール元の逆戻りリターン・コード・スイッチが設定されていなければ、SQL*Formsは成功ラベルまたは次のステップに進みます。

失敗: ユーザー・イグジットで、フィールド内の無効値などのエラーが検出されました。このイグジットによって渡されたオプションのメッセージが、SQL*Forms画面の下部のメッセージ行およびエラー表示画面に表示されます。SQL*Formsでは、行に影響を与えないSQL文に対する場合と同様に応答します。

致命的エラー: ユーザー・イグジットで、SQL文中の実行エラーなど、それ以上処理を続行できない状態が検出されました。イグジットによって渡されたオプションのエラー・メッセージが、SQL*Formsのエラー表示画面に表示されます。SQL*Formsでは、致命的なSQLエラーに対するときと同様に応答します。

ユーザー・イグジットでフィールドの値が変更された後で、失敗または致命的エラーのコードが戻された場合、SQL*Formsはこの変更を廃棄しません。また、逆戻りリターン・コード・スイッチが設定されていて、成功コードが戻されたときにも、SQL*Formsは変更を廃棄しません。

11.7.1 IAP定数

プリコンパイラでは、リターン・コードとして使用する3つの記号定数が生成されます。これらにはIAPの接頭辞が付きます。たとえば、3つの定数はIAPSUCC、IAPFAIL、IAPFTLのようになります。

11.7.2 SQLIEM関数

関数SQLIEMをコールすると、トリガー・ステップが、失敗するか、表示画面で回復不能のエラーを引き起こした場合に、SQL*Formsがメッセージ行に表示するエラー・メッセージをユーザー・イグジットで指定できます。指定したメッセージは、そのステップのために定義されたいかなるメッセージよりも優先されます。

SQLIEM関数コールの構文は次のとおりです。

```
SQLIEM (error_message, message_length);
```

ここで、*error_message*と*message_length*は、それぞれ文字変数と整数の変数です。Oracle Precompilersでは、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によって渡します。つまり、値でなくアドレスを渡します。SQLIEMはSQL*Forms関数です。他のOracleツールからはコールできません。

11.7.3 WHENEVERの使用方法

イグジット内でWHENEVER文を使用すると、無効なデータ型変換(SQLERROR)、フォーム・フィールドにPUTされた切捨て値(SQLWARNING)および行を戻さない問合せ(NOT FOUND)を検出できます。

11.8 例

次の例では、一般的なユーザー・イグジットの記述方法を示します。ホスト・プログラムと同様、ユーザー・イグジットにも宣言部とSQLCAがあることに注意してください。

```
-- subroutine MYEXIT
EXEC SQL BEGIN DECLARE SECTION;
 field1 CHARACTER(20);
```

```
 field2 CHARACTER(20);
 value1 CHARACTER(20);
 value2 CHARACTER(20);
 result_val CHARACTER(20);
EXEC SQL END DECLARE SECTION;
 errmsg CHARACTER(80);
 errlen INTEGER;
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
-- get field values from form
EXEC IAF GET :field1, :field2 INTO :value1, :value2;
-- manipulate values to obtain result_val
-- put result_val into form field
EXEC IAF PUT result VALUES (:result_val);
return(IAPSUCC); -- trigger step succeeded
sqlerror:
 set errmsg = CONCAT('MYEXIT: ', sqlca.sqlerrm.sqlerrmc);
 set errlen = LENGTH(errmsg);
 sqliem(errmsg, errlen); -- pass error message to SQL*Forms
 return(IAPFAIL); -- trigger step failed
```

使用するホスト言語の完全な例は、各ホスト言語の補足資料を参照してください。

11.9 ユーザー・イグジットのプリコンパイルおよびコンパイルについて

ユーザー・イグジットは、スタンドアロン型のホスト・プログラムと同様にプリコンパイルされます。Oracleプリコンパイラの実行を参照してください。

ユーザー・イグジットのコンパイルの手順は、システム固有のOracleマニュアルを参照してください。

11.10 GENXTBユーティリティの使用について

IAPXITモジュール内のIAPプログラム表IAPXTBには、IAPにリンクされているユーザー・イグジットごとに1つのエントリが格納されています。IAPXTBはIAPに各ユーザー・イグジットの名前、位置およびホスト言語を指示します。新しいユーザー・イグジットをIAPに追加するときは、対応するエントリをIAPXTBに追加する必要があります。

IAPXTBは、IAPXTBという同じ名前のデータベース表から導出されます。次のように、オペレーティング・システムのコマンドラインでGENXTBフォームを実行することで、データベース表を変更できます。

```
RUNFORM GENXTB username/password
```

定義するユーザー・イグジットごとに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(COB、FOR、PASまたはPLI)
- 作成日
- 最終変更日
- コメント

IAPXTBデータベース表を変更してから、GENXTBユーティリティを使用して表を読み取り、IAPXITモジュールとそれに含まれるIAPXTBプログラム表を定義するアセンブラまたはC言語のソース・プログラムを作成します。使用するソース言語は、オペレーティング・システムによって異なります。GENXTBユーティリティを実行するための構文は次のとおりです。

```
GENXTB username/password outfile
```

*outfile*は、GENXTBが作成するアセンブラまたはソース・プログラムに指定する名前です。

11.11 SQL*Formsへのユーザー・イグジットのリンクについて

ユーザー・イグジットをコールするフォームの実行前に、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にも、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にもリンクできます。

IAPの実行可能コピーを新規に作成するには、Oracleライブラリおよびホスト言語のリンク・ライブラリから、ユーザー・イグジット・オブジェクト・モジュール、標準IAPモジュール、IAPXITモジュール、その他必要なモジュールをリンクします。リンク方法の詳細はシステムによって異なりますので、システム固有のOracleマニュアルで確認してください。

11.12 SQL*Formsユーザー・イグジットのガイドライン

この項のガイドラインは、一般的な問題の回避に役立ちます。

11.12.1 イグジットの命名

ユーザー・イグジットの名前にはOracleの予約語を指定しないでください。また、SQL*Formsコマンド、関数コード、SQL*Formsで使用される外部定義の名前と競合する名前も使用しないでください。

SQL*Formsでは、ユーザー・イグジットの検索前に、その名前が大文字に変換されます。したがって、イグジット名は、大文字と小文字を区別するホスト言語の場合、ソース・コードで大文字になっている必要があります。

ソース・コード内のユーザー・イグジット・エントリ・ポイントの名前は、ユーザー・イグジット自体の名前になります。このイグジット名は、ホスト言語とオペレーティング・システムにとって有効なファイル名であることが必要です。

11.12.2 Oracleへの接続

ユーザー・イグジットでは、SQL*Formsによって確立された接続を使用してOracleと通信します。ただし、ユーザー・イグジットでSQL*Netを使用して任意のデータベースに追加の接続を確立できます。詳細は、「同時接続」を参照してください。

11.12.3 I/Oコールの発行

SQL*Forms I/Oルーチンは、ホスト言語のプリンタI/Oルーチンと競合する可能性があります。その場合、ユーザー・イグジットではプリンタ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。

11.12.4 ホスト変数の使用

スタンドアロン型のプログラムでの変数の使用に対する制限事項は、ユーザー・イグジットにも適用されます。EXEC SQL文およびEXEC IAF文では、ホスト変数はユーザー・イグジットの宣言部で名前を指定し、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。

11.12.5 表の更新

一般に、ユーザー・イグジットでは、フォームに関連付けられたデータベース表を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、ユーザー・イグジットが関連付けられたデータベース表内の対応する行をUPDATEしたとします。このときトランザクションがCOMMITされると、SQL*Forms作業領域内のレコードがその表に適用され、ユーザー・イグジットのUPDATEは上書きされます。

11.12.6 コマンドの発行

Oracleでは、ユーザー・イグジットで実行された作業に限らず、SQL*Formsのオペレータが開始した作業もコミットまたはロールバックされるため、ユーザー・イグジットからはCOMMITまたはROLLBACKコマンドを発行しないでください。かわりに、SQL*Formsトリガーから COMMITまたはROLLBACKを発行してください。データ定義コマンド(ALTER 、CREATEなど)についても同様で、これらのコマンドでも、実行の前後に暗黙的なCOMMITが発行されるためです。

11.13 EXEC TOOLS文

EXEC TOOLS文では、ユーザー・イグジットからの取得、設定および例外コールバックを処理する包括的な方法を提供することにより、基本的なOracle Toolset(Oracle Forms、Oracle ReportsおよびOracle Graphics)をサポートします。この後の説明はOracle Formsが中心になっていますが、Oracle ReportsおよびOracle Graphicsについても概念は同じです。

EXEC SQL、EXEC ORACLEおよびホスト言語文の他にも、次のEXEC TOOLS文を使用してOracle Formsユーザー・イグジットを記述できます。

- SET
- GET
- SET CONTEXT
- GET CONTEXT
- MESSAGE

EXEC TOOLS GET文およびEXEC TOOLS SET文は、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およびIAF PUTとは異なり、TOOLS GETおよびTOOLS SETはインジケータ変数を受け入れます。EXEC TOOLS MESSAGE文は、メッセージ処理関数のSQLIEMに相当します。EXEC TOOLS SET CONTEXT文およびEXEC TOOLS GET CONTEXT文は新しく、SQL*Formsバージョン3にはありません。

ノート:

COBOL および FORTRAN にはポインタ・データ型がないため、Pro*COBOL または Pro*FORTRAN のプログラムでは、SET CONTEXT 文および GET CONTEXT 文は使用できません。

11.13.1 EXEC TOOLS SET

EXEC TOOLS SET文は、ユーザー・イグジットからOracle Formsに値を渡します。具体的には、ホスト変数および定数の値をOracle Formsの変数および項目に割り当てます。これらの値は、ユーザー・イグジットからフォームに制御が戻った後に表示されます。

EXEC TOOLS SET文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET form_variable[, ...]
 VALUES ({:host_variable[:indicator] | constant}[, ...]);
```

*form_variable*は、Oracle Formsのフィールド、パラメータ、システム変数、グローバル変数、またはこれらのうちの1つの名前を含むホスト変数(接頭辞としてコロンが付く)です。

次のPro*Cの例では、ユーザー・イグジットによって従業員名(オプションのインジケータ付き)がOracle Formsに渡されます。

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

```
 ...
 char ename[20];
 short ename_ind;
EXEC SQL END DECLARE SECTION;
...
strcpy(ename, "MILLER");
ename_ind = 0;
EXEC TOOLS SET emp.ename VALUES (:ename:ename_ind);
```

この例で、*emp.ename*はOracle Formsのblock.fieldです。

11.13.2 EXEC TOOLS GET

EXEC TOOLS GET文は、Oracle Formsからユーザー・イグジットに値を渡します。具体的には、Oracle Formsの変数および項目の値を、ホスト変数に割り当てます。値が渡されるとすぐに、ユーザー・イグジットではそれらを任意の目的に使用できます。

EXEC TOOLS GET文を記述するには、次の構文を使用します。

```
EXEC TOOLS GET form_variable[, ...]
 INTO :host_variable[:indicator][, ...];
```

*form_variable*は、Oracle Formsのフィールド、パラメータ、システム変数、グローバル変数、またはこれらのうちの1つの名前を含むホスト変数です。

次の例では、Oracle Formsがblock.fieldの*emp.ename*からユーザー・イグジットに従業員名を渡します。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char ename[20];
EXEC SQL END DECLARE SECTION;
...
EXEC TOOLS GET emp.ename INTO :ename;
```

11.13.3 EXEC TOOLS SET CONTEXT

EXEC TOOLS SET CONTEXT文を使用すると、あるユーザー・イグジット・コールのコンテキスト情報を、別のユーザー・イグジット・コールのために保存できます。SET CONTEXTは、EXEC TOOLS GET CONTEXT文で後から参照できるホスト言語のポインタ変数の名前を指定します。ポインタ変数は、コンテキスト情報が格納されているメモリーのブロックを指します。SET CONTEXT文を使用すれば、情報を保存するためにグローバル変数を宣言する必要はありません。

EXEC TOOLS SET CONTEXT文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
 [IDENTIFIED] BY context_name;
```

オプションのIDENTIFIEDキーワードは読みやすさを改善するために使用でき、*context_name*は未宣言の識別子、またはコンテキスト領域を命名する文字ホスト変数です。

次の例では、ユーザー・イグジットで、後から使用するためのコンテキスト情報を保存します。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char context1[30];
EXEC SQL END DECLARE SECTION;
...
strcpy(context1, "This is context1");
EXEC TOOLS SET CONTEXT :context1 BY my_app1;
```

この例では、コンテキスト名*my_app1*は未宣言の識別子です。C言語の場合、引数としてCHAR型の配列が使用されているとき、配列名はその配列に対するポインタと同じになります。

11.13.4 EXEC TOOLS GET CONTEXT

EXEC TOOLS GET CONTEXT文は、ホスト言語のポインタ変数の値を取得して、ユーザー・イグジットに渡します。ポインタ変数は、コンテキスト情報が格納されているメモリーのブロックを指します。

EXEC TOOLS GET CONTEXT文を記述するには、次の構文を使用します。

```
EXEC TOOLS GET CONTEXT context_name INTO :host_pointer_variable;
```

*context_name*は、未宣言の識別子、またはコンテキスト領域を命名する文字ホスト変数です。

次のPro*Cの例では、ユーザー・イグジットは、前に保存されたコンテキスト情報に対するポインタを取得します。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 char *ctx_ptr;
EXEC SQL END DECLARE SECTION;
...
EXEC TOOLS GET CONTEXT my_app1 INTO :ctx_ptr;
```

この例では、コンテキスト名*my_app1*は未宣言の識別子です。

11.13.5 EXEC TOOLS MESSAGE

EXEC TOOLS MESSAGE文は、ユーザー・イグジットからOracle Formsにメッセージを渡します。このメッセージは、ユーザー・イグジットからフォームに制御が戻った後に、Oracle Formsのメッセージ行に表示されます。

EXEC TOOLS MESSAGE文を記述するには、次の構文を使用します。

```
EXEC TOOLS MESSAGE message_text [severity_code];
```

*message_text*は引用符付き文字列または文字ホスト変数、オプションの*severity_code*は整定数またはホスト変数です。MESSAGE文では、インジケータ変数は受け入れられません。

次のPro*Cの例では、ユーザー・イグジットからOracle Formsに、エラー・メッセージと重大度コードが渡されます。

```
EXEC TOOLS MESSAGE 'Bad field name! Please reenter.' 15;
```

A 新機能

この付録では、Oracleプリコンパイラ、リリース1.8の改良点と新機能について説明します。プロのソフトウェア開発者の実際的なニーズを満たすように設計されたこれらの機能は、効率的かつ信頼できるアプリケーションの作成に役立ちます。

A.1 インジケータ変数を使用しないNULLのフェッチについて

Oracleプリコンパイラのリリース1.5、1.6および1.7では、インジケータ変数が関連付けられていないホスト変数にデータをFETCHするソース・ファイルの場合、実行時にホスト変数にNULLが戻されると、ORA-01405のメッセージが表示されます。リリース1.8では、MODE=ORACLEおよびDBMS=V7を指定するとき、UNSAFE_NULL=YESも指定すると、ORA-01405のメッセージが表示されなくなります。

Oracleデータベース用のアプリケーションを開発する場合、NULLが戻される可能性のあるホスト変数には、インジケータ変数を組み込むことをお薦めします。ただし、Oracleバージョン6からOracleデータベース・バージョン7へアプリケーションを移行する際には、UNSAFE_NULLオプションを使用すると、移行のプロセスが非常に容易になります。

詳細は、UNSAFE_NULLおよびインジケータ変数の使用についてを参照してください。

A.1.1 DBMS=V7およびMODE=ORACLEの使用方法について

MODE=ORACLEおよびDBMS=V7を指定してプリコンパイルしたアプリケーションでは、実行時にインジケータ変数が関係付けられていないホスト変数にNULLが戻されると、ORA-01405エラーが発生します。これらのオプションを指定してOracleデータベース・バージョン7にアップグレードするときには、次の2つの方法のいずれかでアプリケーションを移行する必要があります。

- 必要なインジケータ変数を組み込むようにソース・コードを変更
- コマンドラインでUNSAFE_NULL=YESを指定

Oracleデータベース・バージョン7にアップグレード中で、プリコンパイル時にDBMS=V7を使用するとき、またはOracleバージョン6と異なるOracleデータベース・バージョン7の新機能を使用するとき、ほとんどの場合、ソース・ファイルに対する変更は最小限で済みます。ただし、アプリケーションでインジケータ変数が関連付けられていないホスト変数にNULL値をFETCHする可能性がある場合は、ORA-01405メッセージが表示されないようにUNSAFE_NULL=YESを指定し、ソース・ファイルに関連するインジケータ変数を追加しないでください。

A.1.2 関連のエラー・メッセージ

UNSAFE_NULLオプションに関連するプリコンパイル時のメッセージの詳細は、『*Oracle Database*エラー・メッセージ』を参照してください。

A.2 配列INSERTおよび配列SELECT構文の追加

DB2プリコンパイラの配列INSERTおよび配列SELECT構文が、Oracleプリコンパイラでサポートされるようになりました。オプションでROWSET句とROWSET STARTING AT句がフェッチの方向(FIRST, PRIOR, NEXT, LAST, CURRENT, RELATIVEおよびABSOLUTE)で使用されます。新しいINSERT/SELECT構文の詳細は、『*Pro*COBOL*プログラマーズ・ガイド』および『*Pro*C/C++*プログラマーズ・ガイド』を参照してください。

A.3 SQL99構文サポート

SQL規格により、規格に準拠するすべてのソフトウェア製品で、SQLアプリケーションを移植できます。Oracle機能は、ANSI/ISO SQL99規格(ANSI準拠の結合を含む)に準拠しています。Pro*Cobolでは、Oracle Databaseでサポートさ

れているすべてのSQL99機能がサポートされており、SELECT文、INSERT文、DELETE文およびUPDATE文と、DECLARE CURSOR文でのカーソル本体のSQL99構文がサポートされています。

A.4 実行計画の修正について

Pro*C/C++またはPro*Cobol開発環境で使用されるSQLの実行計画を修正するには、プリコンパイル時にOracleのアウトライン機能を使用する必要があります。アウトラインは、SQL文と関連付けられた一連のオプティマイザ・ヒントとして実装されます。SQL文のアウトラインの使用を有効にすると、Oracleでは、格納されたヒントが自動的に考慮され、それらのヒントに従って実行計画を生成しようとします。これにより、モジュールの統合時および異なる環境へのデプロイ時に、パフォーマンスは影響を受けません。

Pro*C/C++およびPro*Cobolでアウトラインを作成するときには、次のSQL文を使用できます。

- SELECT
- DELETE
- UPDATE
- INSERT... SELECT
- CREATE TABLE... AS SELECT

アウトライン・オプションが設定されている場合、プリコンパイルが正常に終了すると、2つのファイル(SQLファイルおよびLOGファイル)が生成されます。コマンドライン・オプションoutlineおよびoutlnprefixは、アウトラインの生成を制御します。生成された各アウトライン名は一意です。アプリケーションで使用するファイル名が一意であるため、アウトライン名の生成時にこの情報が使用されます。また、カテゴリ名も接頭辞として使用されます。

ノート:

アウトライン名の最大長は 30 バイトです。この制限を超えると、プリコンパイラではエラーが発生します。outlnprefix オプションを使用すると、アウトライン名の長さを制限できます。

関連項目:

- 『*Pro*COBOLプログラマーズ・ガイド*』
- 『*Pro*C/C++プログラマーズ・ガイド*』

A.5 暗黙的バッファ挿入の使用について

パフォーマンス向上のために、アプリケーション開発者は、埋込みSQL文内のホスト配列を参照できます。これにより、データベースへの1回のラウンドトリップでSQL文の配列を実行できます。配列の実行によってパフォーマンスが大幅に向上するにもかかわらず、ANSI規格ではないため、この機能を使用しない開発者もいます。たとえば、Oracle製品で配列の実行を使用するように記述されたアプリケーションは、IBMのプリコンパイラを使用してプリコンパイルすることはできません。

対処方法として、バッファ済INSERT文を使用すると、ANSI規格の埋込みSQL構文を保持しながらパフォーマンスを向上させることができます。

コマンドライン・オプション「max_row_insert」は、INSERT文の実行前にバッファする行の数を制御します。このオプションのデフォルトは0で、機能は無効化されています。この機能を有効化するには、0よりも大きい任意の数を指定します。

関連項目:

暗黙的バッファ挿入機能の使用方法の詳細は、次のマニュアルを参照してください。

- 『*Pro*COBOLプログラマーズ・ガイド*』
- 『*Pro*C/C++プログラマーズ・ガイド*』

A.6 動的SQL文のキャッシュ

文キャッシュは、セッションごとの文のキャッシュを提供および管理する機能です。サーバーでは、文を再び解析することなく、カーソルがいつでも使用できるようになっていることを意味します。文のキャッシングは、プリコンパイラ・アプリケーションで有効にでき、動的SQL文に依存するすべてのアプリケーションのパフォーマンス向上に役立ちます。パフォーマンスの向上は、動的文を再利用する際の解析のオーバーヘッドをなくすことで達成されます。

このパフォーマンスの改善は、動的文のキャッシングが可能になる新しいコマンドライン・オプションstmt_cache(文のキャッシュ・サイズ用)を使用することで実現します。この新しいオプションを有効にすると、セッション作成時に文のキャッシュが作成されます。キャッシングは動的文に対してのみ適用され、静的文用のカーソル・キャッシュとこの機能は共存します。

コマンドライン・オプションstmt_cacheには、0から65535の範囲で任意の値を指定できます。デフォルト(値0)で、文キャッシングは無効化されています。stmt_cacheオプションでは、アプリケーションにそれぞれの動的SQL文の予測数を保持するように設定できます。

例A-1 stmt_cacheオプションの使用方法

次の例は、stmt_cacheオプションの使用方法を示しています。このプログラムでは、表に行を挿入し、挿入した行をループ内のカーソルを使用して選択します。このプログラムのプリコンパイルにstmt_cacheオプションを使用すると、通常のプリコンパイルよりもパフォーマンスが向上します。

```
/*
 *  stmtcache.pc
 *
 *  NOTE:
 *  When this program is used to measure the performance with and without
 *  stmt_cache option, do the following changes in the program,
 *  1. Increase ROWSCNT to high value, say 10000.
 *  2. Remove all the print statements, usually which comsumes significant
 *     portion of the total program execution time.
 *
 *  HINT: In Linux, gettimeofday() can be used to measure time.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <oraca.h>

#define ROWSCNT 10

char    *username = "aaaaa";
char    *password = "bbbbb";

/* Function prototypes */
```

```
void sql_error(char *msg);
void selectdata();
void insertdata();

int main()
{
  EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");

  /* Connect using the default schema aaaaa/bbbbb */
  EXEC SQL CONNECT :username IDENTIFIED BY :password;

  /* core functions to insert and select the data */
  insertdata();
  selectdata();

/* Rollback pll the changes and disconnect from Oracle. */
  EXEC SQL ROLLBACK WORK RELEASE;

  exit(0);
}

/*Insert the data for ROWSCNT items into tpc2sc01 */
void insertdata()
{
  varchar dynstmt[80];
  int i;
  varchar ename[10];
  float comm;
  char *str;

  /* Allocates temporary buffer */
  str = (char *)malloc (11 * sizeof(char));

  strcpy ((char *)dynstmt.arr,
          "INSERT INTO bonus (ename, comm) VALUES (:ename, :comm)");
  dynstmt.len = strlen(dynstmt.arr);
  EXEC SQL PREPARE S FROM :dynstmt;

  printf ("Inserts %d rows into bonus table using dynamic SQL statement¥n",
          ROWSCNT);
  for (i=1; i<=ROWSCNT; i++)
  {
    sprintf (str, "EMP_%05d",i);
    strcpy (ename.arr, str);
    comm = i;
    ename.len = strlen (ename.arr);
    EXEC SQL EXECUTE S USING :ename, :comm;
  }

  free(str);
}

/* Select the data using the cursor */
void selectdata()
{
  varchar dynstmt[80];
  varchar ename[10];
  float comm;
  int i;
```

```
  strcpy((char *)dynstmt.arr,
         "SELECT ename, comm FROM bonus WHERE comm = :v1");
  dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);

  printf ("Fetches the inserted rows using using dynamic SQL statement¥n¥n");
  printf ("  ENAME      COMMISSION¥n¥n");

  for (i=1; i<=ROWSCNT; i++)
  {
    /* Do the prepare in the loop so that the advantage of stmt_caching
       is visible*/
    EXEC SQL PREPARE S FROM :dynstmt;

    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C USING :i;

    EXEC SQL WHENEVER NOT FOUND DO break;

    /* Loop until the NOT FOUND condition is detected. */
    for (;;)
    {
      EXEC SQL FETCH C INTO :ename, :comm;
      ename.arr[ename.len] = '¥0';
      printf ("%10s    %7.2f¥n", ename.arr, comm);
    }
    /* Close the cursor so that the reparsing is not required for stmt_cache */
    EXEC SQL CLOSE C;
  }
}

void sql_error(char *msg)
{
    printf("¥n%s", msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '¥0';
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '¥0';
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '¥0';
    printf("¥n%s¥n", sqlca.sqlerrm.sqlerrmc);
    printf("in ¥"%s...¥"¥n", oraca.orastxt.orastxtc);
    printf("on line %d of %s.¥n¥n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);

   /* Disable ORACLE error checking to avoid an infinite loop
    * should another error occur within this routine.
    */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Release resources associated with the cursor. */
    EXEC SQL CLOSE C;

    /* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

A.7スクロール可能なカーソル

スクロール可能カーソルは、SQL文が実行され、実行中に処理された情報が格納される作業領域です。カーソルを実行すると、問合せの結果は結果セットと呼ばれる一連の行に配置されます。結果セットは、順番にフェッチすることも、順不同でフェッチす

ることもできます。順不同の結果セットをスクロール可能カーソルと呼びます。スクロール可能カーソルを使用すると、ユーザーは前から、後ろからまたはランダムな方法でデータベース結果セットの行にアクセスできます。このスクロール可能なカーソルにより、プログラムは結果セットの任意の行をフェッチできます。スクロール可能なカーソルの詳細は、「スクロール可能なカーソル」を参照してください。

A.8 プラットフォームのエンディアン形式のサポート

Oracleで格納されるユニコード・データ(UTF16)は、常にビッグエンディアン方式です。現在、クライアント・アプリケーションは様々なプラットフォームで実行されます。LinuxおよびWindowsではリトルエンディアン方式で表され、Solarisではビッグエンディアン方式で表されます。Pro*Cobolでは、UTF16のデータが挿入または選択されると、サーバーとクライアント間のエンディアン形式が変換されません。このため、PIC N変数のUTF16 (UCS2)の文字列が破損します。

PIC N変数でのプラットフォーム・エンディアン(LinuxおよびWindowsの場合はリトルエンディアン方式、Solarisの場合はビッグエンディアン方式)は、コマンドライン・オプションのpicn_endianを使用すれば維持できます。

新しいコマンドライン・オプション

```
picn_endian={BIG|OS}
```

picn_endian=bigの場合、PIC N変数は文字セットID AL16UTF16にバインドされます。

picn_endian=osの場合、PIC N変数は文字セットID UCS2にバインドされます。

このオプションのデフォルト値は「big」で、現行の動作が保持されます。NLS_NCHARがAL16UTF16でない場合、このオプションは無視されます。

PIC N変数の文字セットの形式は、既存のPro*Cobolコマンドライン・オプションを使用して設定できます。

```
charset_picn={nchar_charset|db_charset}
```

A.9 B領域の長さの柔軟性

形式がANSIに設定されている場合、Pro*CobolプログラムのB領域の長さは72列に制限されていました。COBOLコンパイラでは、B領域の長さが最大253列までサポートされるようになりました。これにより、プログラマは72列より長い行を柔軟に入力できます。Pro*Cobolでは、Pro*Cobolアプリケーションが次のオプションでプリコンパイルされる場合にB領域の長さが最大253列までサポートされます。

FORMAT=VARIABLE

オプション。

B Oracleの予約語、キーワードおよびネームスペース

この付録の内容は次のとおりです。

- Oracleの予約語
- Oracleのキーワード
- PL/SQLの予約語
- Oracleの予約済ネームスペース

この付録では、Oracleで特別な意味を持つ語を示しています。それぞれの語は、出現するコンテキストで特別な役割を果します。たとえば、INSERT文では、予約語のINTOは、行が追加される表を示します。しかし、FETCH文またはSELECT文では、予約語のINTOは、列値が割り当てられる出力ホスト変数を示します。

B.1 Oracleの予約語

次の語は、Oracleで予約されています。つまり、予約語はOracleで特別な意味を持っているため、再定義はできません。このため、それらのワードを使用して、列、表、索引などのデータベース・オブジェクトに名前を付けることはできません。

Oracleの予約語	Oracleの予約語	Oracleの予約語	Oracleの予約語
ACCESS	ELSE	MODIFY	START
ADD	EXCLUSIVE	NOAUDIT	SELECT
ALL	EXISTS	NOCOMPRESS	SESSION
ALTER	FILE	NOT	SET
AND	FLOAT	NOTFOUND	SHARE
ANY	FOR	NOWAIT	SIZE
ARRAYLEN	FROM	NULL	SMALLINT
AS	GRANT	NUMBER	SQLBUF
ASC	GROUP	OF	SUCCESSFUL
AUDIT	HAVING	OFFLINE	SYNONYM
BETWEEN	IDENTIFIED	ON	SYSDATE
BY	IMMEDIATE	ONLINE	TABLE

Oracleの予約語	Oracleの予約語	Oracleの予約語	Oracleの予約語
CHAR	IN	OPTION	THEN
CHECK	INCREMENT	OR	TO
CLUSTER	INDEX	ORDER	TRIGGER
COLUMN	INITIAL	PCTFREE	UID
COMMENT	INSERT	PRIOR	UNION
COMPRESS	INTEGER	PRIVILEGES	UNIQUE
CONNECT	INTERSECT	PUBLIC	UPDATE
CREATE	INTO	RAW	USER
CURRENT	IS	RENAME	VALIDATE
DATE	LEVEL	RESOURCE	VALUES
DECIMAL	LIKE	REVOKE	VARCHAR
DEFAULT	LOCK	ROW	VARCHAR2
DELETE	LONG	ROWID	VIEW
DESC	MAXEXTENTS	ROWLABEL	WHENEVER
DISTINCT	MINUS	ROWNUM	WHERE
DROP	MODE	ROWS	WITH

B.2 Oracleキーワード

次の語もOracleで特別な意味を持っていますが、予約語ではないため再定義ができます。ただし、一部は最終的に予約語になる場合があります。

Oracleのキーワード	Oracleのキーワード	Oracleのキーワード	Oracleのキーワード
ADMIN	CURSOR	FOUND	MOUNT

Oracleのキーワード	Oracleのキーワード	Oracleのキーワード	Oracleのキーワード
AFTER	CYCLE	FUNCTION	NEXT
ALLOCATE	DATABASE	GO	NEW
ANALYZE	DATAFILE	GOTO	NOARCHIVELOG
ARCHIVE	DBA	GROUPS	NOCACHE
ARCHIVELOG	DEC	INCLUDING	NOCYCLE
AUTHORIZATION	DECLARE	INDICATOR	NOMAXVALUE
AVG	DISABLE	INITRANS	NOMINVALUE
BACKUP	DISMOUNT	INSTANCE	NONE
BEGIN	DOUBLE	INT	NOORDER
BECOME	DUMP	KEY	NORESETLOGS
BEFORE	EACH	LANGUAGE	NORMAL
BLOCK	ENABLE	LAYER	NOSORT
BODY	END	LINK	NUMERIC
CACHE	ESCAPE	LISTS	OFF
CANCEL	EVENTS	LOGFILE	OLD
CASCADE	EXCEPT	MANAGE	ONLY
CHANGE	EXCEPTIONS	MANUAL	OPEN
CHARACTER	EXEC	MAX	OPTIMAL
CHECKPOINT	EXPLAIN	MAXDATAFILES	OWN
CLOSE	EXECUTE	MAXINSTANCES	PACKAGE

Oracleのキーワード	Oracleのキーワード	Oracleのキーワード	Oracleのキーワード
COBOL	EXTENT	MAXLOGFILES	PARALLEL
COMMIT	EXTERNALLY	MAXLOGHISTORY	PCTINCREASE
COMPILE	FETCH	MAXLOGMEMBERS	PCTUSED
CONSTRAINT	FLUSH	MAXTRANS	PLAN
CONSTRAINTS	FREELIST	MAXVALUE	PLI
CONTENTS	FREELISTS	MIN	PRECISION
CONTINUE	FORCE	MINEXTENTS	PRIMARY
CONTROLFILE	FOREIGN	MINVALUE	PRIVATE
COUNT	FORTRAN	MODULE	PROCEDURE
PROFILE	SAVEPOINT	SQLSTATE	TRACING
QUOTA	SCHEMA	STATEMENT_ID	TRANSACTION
READ	SCN	STATISTICS	TRIGGERS
REAL	SECTION	STOP	TRUNCATE
RECOVER	SEGMENT	STORAGE	UNDER
REFERENCES	SEQUENCE	SUM	UNLIMITED
REFERENCING	SHARED	SWITCH	UNTIL
RESETLOGS	SNAPSHOT	SYSTEM	USE
RESTRICTED	SOME	TABLES	USING
REUSE	SORT	TABLESPACE	WHEN
ROLE	SQL	TEMPORARY	WRITE

Oracleのキーワード	Oracleのキーワード	Oracleのキーワード	Oracleのキーワード
ROLES	SQLCODE	THREAD	WORK
ROLLBACK	SQLERROR	TIME	

B.3 PL/SQLの予約語

次のPL/SQLのキーワードは、埋込みSQL文で使用する際に特別な処理が必要になる場合があります。

- ABORT
- ACCEPT
- ACCESS
- ADD
- ALL
- ALTER
- AND
- ANY
- ARRAY
- ARRAYLEN
- AS
- ASC
- ASSERT
- ASSIGN
- AT
- AUTHORIZATION
- AVG
- BASE_TABLE
- BEGIN
- BETWEEN
- BINARY_INTEGER
- BODY
- BOOLEAN
- BY
- CASE
- CHAR_BASE
- CHAR
- CHECK
- CLOSE
- CLUSTER
- CLUSTERS
- COLAUTH
- COLUMNS
- COMMIT
- COMPRESS

- CONNECT
- CONSTANT
- COUNT
- CRASH
- CREATE
- CURRENT
- CURRVAL
- CURSOR
- DATA_BASE
- DATABASE
- DATE
- DBA
- DEBUGOFF
- DEBUGON
- DECLARE
- DEFAULT
- DEFINITION
- DELAY
- DELETE
- DELTA
- DESC
- DIGITS
- DISPOSE
- DISTINCT
- DO
- DROP
- ELSE
- ELSIF
- END
- ENTRY
- EXCEPTION_INIT
- EXCEPTION
- EXISTS
- EXIT
- FALSE
- FETCH
- FLOAT
- FOR
- FORM
- FROM
- FUNCTION
- GENERIC
- GOTO
- GRANT
- GROUP

- HAVING
- IDENTIFIED
- IF
- IN
- INDEX
- INDEXES
- INDICATOR
- INSERT
- INTEGER
- INTERSECT
- INTO
- IS
- LEVEL
- LIKE
- LIMITED
- LOOP
- MAX
- MIN
- MINUS
- MLSLABEL
- MOD
- MODE
- NATURAL
- NEW
- NEXTVAL
- NOCOMPRESS
- NOT
- NULL
- NUMBER_BASE
- NUMBER
- OF
- ON
- OPEN
- OPTION
- OR
- ORDER
- OTHERS
- 外
- PACKAGE
- PARTITION
- PCTFREE
- POSITIVE
- PRAGMA
- PRIOR
- PRIVATE

- PROCEDURE
- PUBLIC
- RAISE
- RANGE
- REAL
- RECORD
- RELEASE
- REMR
- RENAME
- RESOURCE
- RETURN
- REVERSE
- REVOKE
- ROLLBACK
- ROWID
- ROWLABEL
- ROWNUM
- ROWTYPE
- RUN
- SAVEPOINT
- SCHEMA
- SEPARATE
- SET
- SIZE
- SMALLINT
- SELECT
- SPACE
- SQL
- SQLCODE
- SQLERRM
- START
- STATEMENT
- STDDEV
- SUBTYPE
- SUM
- TABAUTH
- TABLE
- TABLES
- TASK
- TERMINATE
- THEN
- TO
- TRUE
- TYPE
- UNION

- UNIQUE
- UPDATE
- USE
- VALUES
- VARCHAR
- VARCHAR2
- VARIANCE
- VIEW
- VIEWS
- WHEN
- WHERE
- WHILE
- WITH
- WORK
- XOR

B.4 Oracleの予約済ネームスペース

表B-1に、Oracleにより予約されているネームスペースを示します。Oracleライブラリ内の関数名の最初の文字列は、このリストの文字列に制限されています。名前が競合する可能性があるため、関数名にはこれらの文字で始まらない名前を使用してください。

たとえば、SQL*Net Transparent Network Serviceの機能はすべて「NS」の文字で始まるため、「NS」で始まる関数名は避ける必要があります。

表B-1 Oracleの予約済ネームスペース

ネームスペース	ライブラリ
O	OCI 関数
S	SQLLIB およびシステム依存ライブラリの関数名
XA	XA アプリケーション専用の外部関数
GEN KP L NA NC ND NL NM NR NS NT NZ TTC UPI	内部関数

C パフォーマンス・チューニング

この付録の内容は次のとおりです。

- パフォーマンス低下の原因
- パフォーマンスの改善方法
- ホスト配列の使用
- 埋込みPL/SQLの使用
- SQL文の最適化
- 索引の使用方法について
- 行レベル・ロックの利用
- 不要な解析の排除について

この付録では、アプリケーションのパフォーマンス(性能)が向上する簡単な適用方法をいくつか紹介します。これらの方法を使用すると、多くの場合、処理時間を25%以上短縮できます。

C.1 パフォーマンス低下の原因

パフォーマンス低下の原因の1つは、Oracleの通信オーバーヘッドが多いことです。Oracleは一度にSQL文を1つずつ処理する必要があります。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文を送信する必要があるため、ネットワークの通信量が増加することになります。ネットワークの通信量が多いと、アプリケーションの処理速度が著しく低下する可能性があります。

パフォーマンスを低下させるもう1つの原因は、非効率的なSQL文です。SQLは非常に柔軟性があるため、2つの異なる文で同じ結果を得ることができますが、一方の文の効率が悪い場合があります。たとえば、次の2つのSELECT文では、同じ行(最低1人の従業員がいる部門の名前と番号)を戻します。

```
EXEC SQL SELECT DNAME, DEPTNO
 FROM DEPT
 WHERE DEPTNO IN (SELECT DEPTNO FROM EMP);
EXEC SQL SELECT DNAME, DEPTNO
 FROM DEPT
 WHERE EXISTS
 (SELECT DEPTNO FROM EMP WHERE DEPT.DEPTNO = EMP.DEPTNO);
```

ただし、最初の文では、DEPT表内のすべての部門番号を探して時間のかかるEMP表全体のスキャンを行うため、処理速度が低下します。EMP表内のDEPTNO列に索引を付けていても、この副問合せにはDEPTNOを指定するWHERE句がないので、索引は使用されません。

パフォーマンス低下の3つ目の原因は、不要な解析およびバインドです。SQL文を実行する前にOracleがこのSQL文を解析しバインドする必要があることに注意してください。解析とは、SQL文を調べて、これが構文規則に従って正しいデータベース・オブジェクトを参照していることを確認する作業です。バインドとは、SQL文内のホスト変数をOracleがその値に対して読取り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において、カーソルの管理を十分に行っているとはいえません。このため不要な解析またはバインドが発生し、結果的に処理のオーバーヘッドが著しく増加します。

C.2 パフォーマンスの改善方法

プリコンパイルしたプログラムのパフォーマンスがよくない場合でも、オーバーヘッドを減少させる方法はあります。

特にネットワーク環境では、次の方法でOracleの通信オーバーヘッドを大幅に減らせます。

- ホスト配列の使用
- 埋込みPL/SQLの使用

処理のオーバーヘッドは、次の方法で大幅に減らせる場合があります。

- 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

C.3 ホスト配列の使用方法

ホスト配列を使用すると、データの集合全体を1つのSQL文で操作できるため、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表に挿入するとします。配列がないと、プログラムは300の個々の挿入(従業員ごとに1つ)を実行する必要があります。配列を使用すると、必要なINSERTは1回のみです。次の文を考えてみます。

```
EXEC SQL INSERT INTO EMP (SAL) VALUES (:salary);
```

*salary*が単純なホスト変数の場合、OracleではINSERT文を1回実行し、EMP表に1行挿入します。その行のSAL列に*salary*の値が格納されます。この方法で300行を挿入するには、INSERT文を300回実行する必要があります。

しかし、*salary*がサイズが300のホスト配列の場合は、300行すべてが一度にEMP表に挿入されます。各行のSAL列には、*salary*配列の要素の値が格納されます。

詳細は、ホスト配列の使用方法を参照してください。

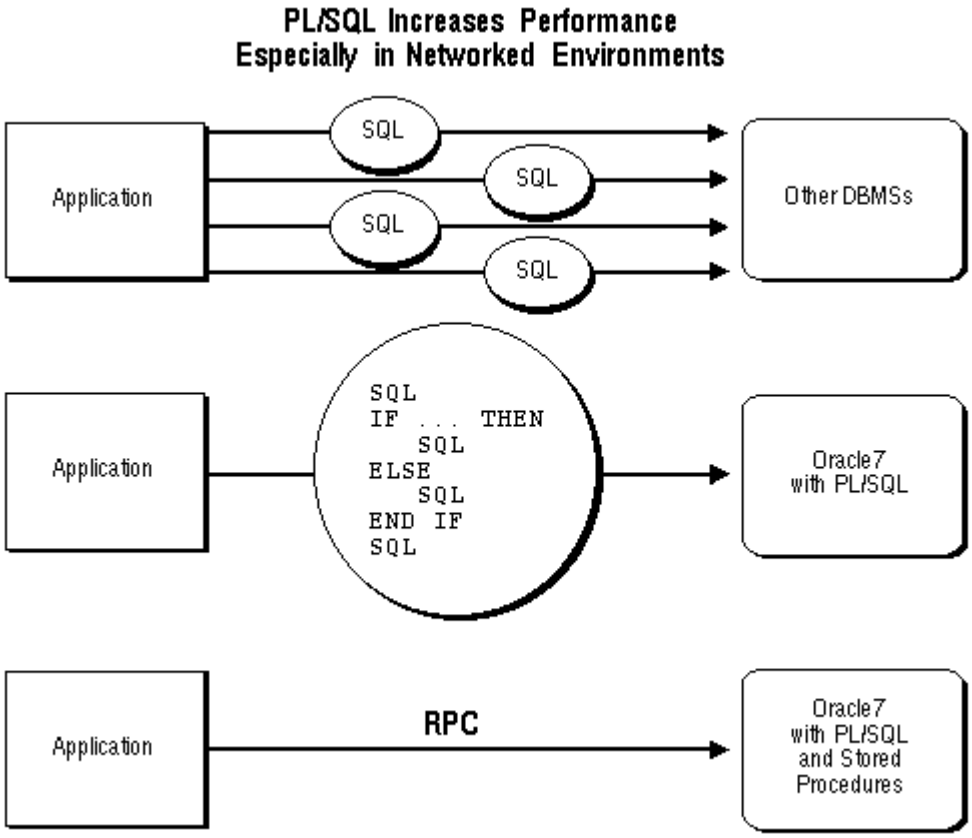
C.4 埋込みPL/SQLの使用方法

図C-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、制御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。これによりアプリケーションとOracle間の通信量は大幅に減少します。

また、PL/SQLサブプログラムを使用して、アプリケーションからOracleへのコールを減らすこともできます。たとえば、10個のSQL文を個々に実行するには、10回のコールが必要ですが、10個のSQL文を含む1つのサブプログラムを実行する場合、コールは1回で済みます。

無名ブロックとは異なり、PL/SQLサブプログラムは別々にコンパイルし、Oracle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つで済みます。

図C-1 PL/SQLによるパフォーマンスの向上



PL/SQLは、Oracleアプリケーション開発ツール(Oracle FormsやOracle Reportsなど)と一緒に使用できます。PL/SQLでこれらのツールに手続き型処理能力を追加することで、パフォーマンスが向上します。PL/SQLを使用すれば、ツールはOracleをコールせずに迅速かつ効率的に計算ができます。その結果、時間が節約され、ネットワーク通信量も減らせます。詳細は、埋込みPL/SQLの使用方法および*Oracle Database PL/SQL言語リファレンス*を参照してください。

C.5 SQL文の最適化

Oracleオプティマイザにより、すべてのSQL文について実行計画(その文を実行するためにOracleが行う一連のステップ)が生成されます。これらのステップは、*Oracle Database*アドバンスト・アプリケーション開発者ガイドに記載されてい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。

C.5.1 オプティマイザ・ヒント

Oracleオプティマイザにより、すべてのSQL文について実行計画(その文を実行するためにOracleが行う一連のステップ)が生成されます。場合によっては、OracleにSQL文を最適化する方法を提案できます。このような提案はヒントと呼ばれ、これによりオプティマイザによる決定に開発者が影響を与えることができます。

ヒントはディレクティブではなく、オプティマイザによるジョブの実行を助けるだけです。一部のヒントはSQL文を最適化するために使用される情報の範囲を制限し、他のヒントは全体的な戦略を提案します。ヒントを使用して、次の事項を指定できます。

- SQL文の最適化アプローチ
- 参照されているそれぞれの表へのアクセス・パス
- 結合のための結合順序
- 表を結合する方法

C.5.2 ヒントの与え方

ヒントをオプティマイザに与えるには、SELECT文、UPDATE文またはDELETE文の動詞の直後に、C言語形式のコメントとしてヒントを入れます。ルール重視の最適化または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または応答時間に寄与します。次の例では、ALL_ROWSヒントにより、問合せ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 -- host arrays
 FROM EMP
 WHERE DEPTNO = :dept_number;
```

プラス記号(+)はコメント先頭の直後に置く必要があり、コメントが1つ以上のヒントを含むことを示します。コメントには注釈とヒントが含まれることがあることに注意してください。

オプティマイザ・ヒントの詳細は、「パフォーマンスとスケーラビリティ」を参照してください。

C.5.3 トレース機能

SQLトレース機能とEXPLAIN PLAN文を使用すると、アプリケーションの処理速度を低下させる恐れのある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。その結果、これらの文のチューニングに専念できます。

EXPLAIN PLAN文は、アプリケーション内のSQL文ごとに実行計画を示します。実行計画を使用すると、非効率的なSQL文を特定できます。

C.6 索引の使用について

索引は、ROWIDを使用して、表の列の各値をその値が格納されている行に関連付けます。索引はCREATE INDEX文で作成します。

表の15%未満の行しか戻さない問合せでは、索引を使用することによりパフォーマンスが向上します。表の15%以上の行を戻す問合せは、全体スキャン(つまり、すべての行を順番に読み取る方法)の方が速く実行されます。WHERE句で索引付きの列を指定する問合せでは、その索引を使用できます。索引付けする列の選択に役立つガイドラインは、「パフォーマンスとスケーラビリティ」を参照してください。

C.7 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。表ロックの詳細は、LOCK TABLE文の使用について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。アプリケーションで表レベル・ロックを指定している場合は、行レベル・ロックを利用できるように変更してください。通常、明示的な表レベル・ロックは使用しないようにします。

C.8 不要な解析の排除について

不要な解析を排除するには、カーソルを正しく操作すること、および次に示すカーソル管理オプションを選択して使用することが必要です。

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

これらのオプションは、暗黙カーソルと明示カーソル、カーソル・キャッシュおよびプライベートSQL領域に影響を与えます。

カーソル・キャッシュの統計情報は、ORACAを使用して取得できます。Oracle通信領域の使用についてを参照してください。

C.8.1 明示カーソルの処理について

カーソルには、暗黙カーソルと明示カーソルの2種類があることを思い出してください。Oracleはデータ定義文およびDML文のすべてを暗黙的にカーソルを宣言します。ただし、複数行を戻す問合せでは、カーソルを明示的に宣言する(またはホスト配列を使用する)必要があります。DECLARE CURSOR文を使用すると、明示カーソルを宣言できます。明示カーソルのオープンおよびクローズの方法は、パフォーマンスに影響を与えます。

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オープン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の値が使用されます。最初にカーソルをクローズせずにオープンのままにしておくと、処理時間を節約できます。

パフォーマンス・チューニングを容易にするために、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の場合、カーソルは再オープンする前にクローズする必要があります。

カーソルのオープンによって取得したリソース(メモリーおよびロック)を解放する場合のみ、そのカーソルをCLOSEします。たとえば、プログラムでは終了前にすべてのカーソルをクローズする必要があります。

C.8.2 カーソルの制御

一般に、明示的に宣言されたカーソルの制御には、次の3つの方法があります。

- DECLARE、OPENおよびCLOSE文の使用
- PREPARE、DECLARE、OPENおよびCLOSE文の使用
- MODE=ANSIの場合のCOMMITによるカーソルのクローズ

最初の方法を使用する場合は、不要な解析に注意する必要があります。OPEN文は、カーソルをクローズしたか、まだ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にかぎり解析を実行します。プログラムはカーソルをDECLAREし、ホスト変数の値が変わるたびにこれを再オープンし、このSQL文が必要なくなったときにのみこれをCLOSEする必要があります。

2番目の方法(動的SQLの方法3および4)を使用する場合は、PREPAREで解析が実行され、解析された文はCLOSEを実行するまで使用できます。プログラムはSQL文を用意してカーソルをDECLAREし、ホスト変数の値が変わるたびにこれを再オープンし、このSQL文が必要なくなったときにのみこれをCLOSEする必要があります。

可能な場合、OPEN文とCLOSE文を1つのループに配置しないようにします。このSQL文が不必要に再解析される原因になる可能性があります。次の例では、OPENとCLOSE文のどちらも、外側のwhileループ内にあります。MODE=ANSIの場合、示されているようにCLOSE文を配置する必要があります。ANSIでは、カーソルをクローズしてから再オープンする必要があるためです。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT ename, sal from emp where sal > :salary and
 sal <= :salary + 1000;
salary = 0;
while (salary < 5000)
{
 EXEC SQL OPEN emp_cursor;
 while (SQLCODE==0)
 {
 EXEC SQL FETCH emp_cursor INTO ....
 ...
 }
 salary += 1000;
 EXEC SQL CLOSE emp_cursor;
}
```

しかし、MODE=ORACLEでは、CLOSE文はオープンされているカーソルがなくても実行できます。CLOSE文を外側のwhileループの外に指定すると、OPEN文が繰り戻されるたびに再解析されるのを回避できます。

```
...
while (salary < 5000)
{
 EXEC SQL OPEN emp_cursor;
 while (sqlca.sqlcode==0)
 {
 EXEC SQL FETCH emp_cursor INTO ....
 ...
 }
 salary += 1000;
}
EXEC SQL CLOSE emp_cursor;
```

C.8.3 カーソル管理オプションの使用について

SQL文の解析は、構成を変更しないかぎり一度のみで十分です。たとえば、選択リストまたはWHERE句に列を追加して、問合せの構成を変更します。HOLD_CURSOR、RELEASE_CURSORおよびMAXOPENCURSORSオプションにより、OracleでのSQL文の解析および再解析の管理方法を制御できます。明示カーソルを宣言すると、解析を最大限に制御できます。

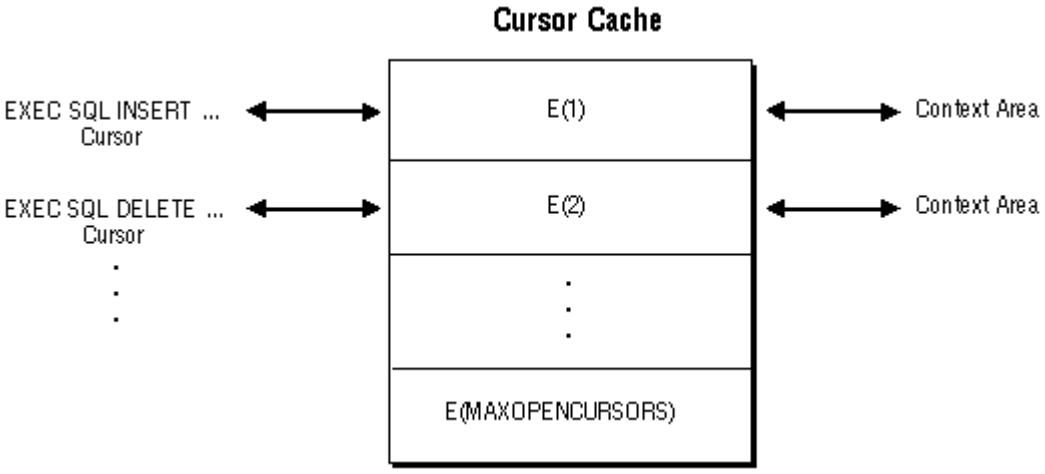
C.8.4 プライベートSQL領域およびカーソル・キャッシュ

データ操作文を実行すると、その文に関連付けられたカーソルがカーソル・キャッシュ内のエントリにリンクされます。カーソル・キャッシュとはカーソル管理のために使用されて連続的に更新されるメモリー領域です。そのカーソル・キャッシュのエントリは、次にプライベートSQL領域にリンクされます。

プライベートSQL領域は、実行時にOracleによって動的に作成される作業領域で、解析済のSQL文、ホスト変数のアドレス、その他文の処理に必要な情報が保存されます。明示カーソルを使用すると、SQL文に名前を付け、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、この情報の処理をある程度制御できます。

図C-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C-2 カーソル・キャッシュでリンクされたカーソル



C.8.5 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、Oracleの初期化パラメータ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。再利用できるかどうかは、HOLD_CURSORとRELEASE_CURSORの値によって決ま

り、明示カーソルの場合は、カーソル自体の状態によって決まります。

MAXOPENCURSORSの値が実際に必要なキャッシュ・エントリ数より少ない場合、Oracleでは再利用可能のマークが付いている最初のキャッシュ・エントリが使用されます。たとえば、INSERT文のキャッシュ・エントリ*E*(1)に再利用可能のマークが付いていて、キャッシュ・エントリの数はすでに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文を再実行するには、Oracleではその文を解析しなおし、別のカーソル・キャッシュ・エントリを再び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。空きメモリーがなくなるか、OPEN_CURSORSで設定した上限に達するまで、Oracleでは必要に応じて追加のキャッシュ・エントリの割当てを続けます。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に小さい値を指定するとメモリーは節約できますが、新しいキャッシュ・エントリの動的割当ておよび解除によりリソースの消費が大きくなる可能性があります。MAXOPENCURSORSの値を大きく設定すると、実行速度は確実に速くなりますが、メモリーの使用量は大きくなります。

C.8.6 実行回数が少ない場合

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)の場合、OracleでSQL文が実行され、カーソルがクローズされた後に、プリコンパイラではカーソルとカーソル・キャッシュ間のリンクに再利用可能のマークを付けます。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。ただし、準備されたカーソルはアクティブ状態のままにしておく必要があるため、HOLD_CURSOR=NOを指定した場合も、そのリンクは維持されます。

RELEASE_CURSOR=YESと指定した場合、OracleでSQL文が実行され、カーソルがクローズされると、プライベートSQL領域は自動的に解放され、解析済の文は失われます。たとえば、サイトでメモリーの節約のためにMAXOPENCURSORSを小さい値に設定しているような場合には、これが必要になります。

データ操作文がデータ定義文より前にあり、どちら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、データ操作文で取得される解析ロックと、データ定義文で要求される排他ロックとの間の競合が回避できます。

RELEASE_CURSOR=YESを指定した場合、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、プライベートSQL領域は解放されます。HOLD_CURSOR=YESを指定しても、RELEASE_CURSOR=YESによりHOLD_CURSOR=YESがオーバーライドされるため、OracleではSQL文を実行する前に、プライベートSQL領域用のメモリーを再度割り当て、SQL文の再解析を続ける必要があります。

それにもかかわらず、RELEASE_CURSOR=YESの場合、OracleではSQL文とPL/SQLブロックの解析済の表現が共有*SQL*キャッシュに保存されるため、それ以上の再解析処理は不要になることがあります。カーソルをクローズしても、解析された表現はキャッシュの内容が書き換えられるまで効力を持ちます。

C.8.7 実行回数が多い場合

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ると、後続の文の実行速度をさらに向上させることができます。

HOLD_CURSOR=YESの場合、OracleでSQL文が実行された後もカーソルとカーソル・キャッシュ間のリンクが維持されます。したがって、解析された文および割り当てられたメモリーが、利用可能なまま維持されます。これは、不要な再解析を避けるためにア

クティブの状態にしておくSQL文で役立ちます。

HOLD_CURSOR=YESおよびRELEASE_CURSOR=NO(デフォルト)の場合、キャッシュ・エントリとプライベートSQL領域間のリンクは、OracleでSQL文が実行された後も維持され、オープン・カーソル数がMAXOPENCURSORSの値を超えないかぎり、再利用されることはありません。これは、解析済の文と割り当てられたメモリーが使用可能な状態のままなので、頻繁に実行されるSQL文に役立ちます。

デフォルト値のHOLD_CURSOR=YESおよびRELEASE_CURSOR=NOを使用すると、Oracleの以前のバージョンでは、SQL文の実行後に、解析済の表現はそのまま使用可能でした。Oracleデータベース・バージョン7では、同様の状況で、解析済の表現を使用できるのは、共有SQLキャッシュで期限切れになって削除されるまでです。通常、これは問題にはなりませんが、そのSQL文が再解析される前に参照オブジェクトの定義が変わると、結果が予期せぬものになる場合があります。

C.8.8 パラメータの相互作用

表C-1は、HOLD_CURSORとRELEASE_CURSORの相互作用を示しています。HOLD_CURSOR=NOを指定すると、RELEASE_CURSOR=NOはオーバーライドされ、RELEASE_CURSOR=YESを指定すると、HOLD_CURSOR=YESがオーバーライドされることに注意してください。

表C-1 HOLD_CURSORとRELEASE_CURSORの相互作用

HOLD_CURSOR	RELEASE_CURSOR	リンク
NO	NO	再利用可能としてマーク
YES	NO	維持
NO	YES	ただちに削除
YES	YES	ただちに削除

D 構文および意味検査

この付録の内容は次のとおりです。

- 構文およびセマンティックのチェック
- チェックの種類および範囲の制御について
- SQLCHECK=SEMANTICSの指定について

埋込みSQL文およびPL/SQLブロックの構文および意味を検査すると、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、SQLCHECKオプションを使用してチェックの種類と範囲を制御する方法を説明します。

D.1 構文検査と意味検査

構文規則は、言語要素を並べて正しい文を作成する基準を示します。つまり、構文検査はキーワード、オブジェクト名、演算子、デリミタなどがSQL文に正しく配置されていることを検証します。たとえば、次の埋込みSQL文には構文上のエラーがあります。

```
-- misspelled keyword WHERE
EXEC SQL DELETE FROM EMP WERE DEPTNO = 20;
-- missing parentheses around column names COMM and SAL
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
```

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文にはセマンティック・エラーがあります。

```
-- nonexistent table, EMPP
EXEC SQL DELETE FROM EMPP WHERE DEPTNO = 20;
-- undeclared host variable, emp_name
EXEC SQL SELECT * FROM EMP WHERE ENAME = :emp_name;
```

SQL構文および意味の規則の定義は、『*Oracle Database SQL言語リファレンス*』を参照してください。

D.2 チェックの種類および範囲の制御について

チェックの種類と範囲は、コマンドラインでSQLCHECKオプションを指定することで制御します。SQLCHECKでは、チェックの種類に構文、セマンティック、あるいはその両方のいずれかを指定できます。検査の範囲には、DML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ではチェックできません。

SQLCHECKには次の値を指定できます。

- SEMANTICS|FULL
- SYNTAX|LIMITED|NONE

SEMANTICS値とFULL値は等価です。同様に、SYNTAX値とLIMITED値も等価ですデフォルト値SYNTAXです。

D.3 SQLCHECK=SEMANTICSの指定について

SQLCHECK=SEMANTICSを指定した場合、プリコンパイラでは次のものを対象とした構文およびセマンティックのチェックが行われます。

- INSERTやUPDATEなどのデータ操作文

- PL/SQLブロック

ただし、プリコンパイラでチェックするのは、リモートのデータ操作文(AT *db_name*句を使用する文)のみです。

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、Oracleに接続してデータ・ディクショナリにアクセスするとこの情報を取得します。データ操作文やPL/SQLブロックで参照する表がすべてDECLARE TABLE文で定義されている場合は、Oracleに接続する必要があります。

Oracleに接続してもデータ・ディクショナリで見つからない情報がある場合は、DECLARE TABLE文を使用して、欠けている情報を提供する必要があります。DECLARE TABLE文の定義とデータ・ディクショナリの定義が矛盾する場合、前者が優先されます。

データ操作文のチェックの際、プリコンパイラでは『*Oracle Database SQL言語リファレンス*』に記載されているOracleデータベース・バージョン7の構文規則が使用されますが、セマンティック・チェックにはさらに厳密な規則が使用されます。その結果、SQLCHECK=SEMANTICSのとき、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可能性があります。

新しいプログラムをプリコンパイルするときには、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を指定する必要があります。

D.3.1 意味検査の使用許可について

SQLCHECK=SEMANTICSを指定すると、プリコンパイラではセマンティック・チェックに必要な情報を、次のいずれかの方法で取得できます。

- Oracleに接続し、そのデータ・ディクショナリにアクセス
- 埋込みDECLARE TABLE文を使用

D.3.2 Oracleへの接続について

セマンティック・チェックを行うために、プリコンパイラではホスト・プログラムで参照される表およびビューの定義が保存されているOracleデータベースに接続できます。Oracleに接続した後、プリコンパイラはデータ・ディクショナリにアクセスして必要な情報を探します。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。

必要な情報の一部がデータ・ディクショナリで見つからない場合(たとえば、プログラムがまだ作成されていない表を参照する場合など)、DECLARE TABLE文を使用して足りない情報を指定する必要があります。

Oracleに接続するには、次の構文を使用して、コマンドラインでUSERIDオプションを指定します。

```
USERID=username
```

*username*は有効なOracleユーザーIDです。パスワードの入力を求められます。ユーザー名のかわりに、次のように指定したとします。

```
USERID=/
```

プリコンパイラでは、このユーザーIDを使用してOracleへの接続が自動的に試みられます。

```
<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NULL)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。

Oracleへの接続を試みて失敗した場合(たとえば、データベースが使用できない場合など)、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。USERIDオプションの指定を省略すると、プリコンパイラでは埋込みDECLARE TABLE文から必要な情報を取得する必要があります。

D.3.3 DECLARE TABLEの使用方法について

プリコンパイラでは、Oracleに接続しなくても、セマンティック・チェックを実行できます。チェックを実行するには、埋込みDECLARE TABLE文から表やビューに関する情報を取得する必要があります。そのため、データ操作文やPL/SQLブロックで参照する表はすべてDECLARE TABLE文で定義する必要があります。

DECLARE TABLE文の構文は次のとおりです。

```
EXEC SQL DECLARE table_name TABLE
 (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

*expr*は、CREATE TABLE文でデフォルトの列値として使用できる任意の式です。

DECLARE TABLEを使用して既存のデータベース表を定義すると、プリコンパイラではその定義を使用し、データ・ディクショナリの定義は無視します。

E 埋込みSQLコマンドおよびディレクティブ

この付録の内容は次のとおりです。

この付録では、SQL92の埋込みSQLコマンドとディレクティブ、およびOracleの埋込みSQL拡張機能について説明します。これらのコマンドおよびディレクティブをソース・コードで使用するときは、キーワードのEXEC SQLを前に付けます。SQL構文をすべて暗記しようとせずに、次のものを含め、この付録を参照してください。

- 埋込みSQLコマンドおよびディレクティブの概要
- コマンド記述に関する項

- 構文図の読み方
- コマンドおよびディレクティブのアルファベット順のリスト

E.1 プリコンパイラのディレクティブと埋込みSQLコマンドの概要

埋込みSQLコマンドは、DDL、DMLおよびトランザクション制御文を、手続き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E-1では、埋込みSQLコマンドおよびディレクティブの機能の概要を示しています。

表E-1の「タイプ」列は、ソース/タイプの形式で表記され、ソースはSQL92標準SQL(S)またはOracle拡張機能(O)のいずれか、タイプは実行可能(E)文またはディレクティブ(D)のいずれかです。

表E-1 埋込みSQLコマンドとディレクティブの概要

EXEC SQL文	タイプ	用途
ALLOCATE	O/E	カーソル変数にメモリーを割り当てます。
CLOSE	S/E	カーソルを無効にし、保持されているリソースを解放します。
COMMIT	S/E	データベースへの変更をすべて確定して、現行のトランザクションを終了します(オプションでリソースを解放し、データベースとの接続を切断します)。
CONNECT	O/E	Oracle インスタンスに接続します。
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 プリコンパイラで埋込み SQL 文の意味検査に使用される表構造を宣言します。
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。
EXECUTE...END-EXEC	O/E	無名 PL/SQL ブロックを実行します。
EXECUTE	S/E	準備済の動的 SQL 文を実行します。
EXECUTE IMMEDIATE	S/E	ホスト変数を持たない SQL 文を準備して実行します。

EXEC SQL文	タイプ	用途
FETCH	S/E	問合せで選択した行を取り出します。
INSERT	S/E	表またはビューの実表に行を追加します。
OPEN	S/E	カーソルに対応付けられた問合せを実行します。
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、現行のトランザクションで加えられた変更をすべて破棄し、ロックをすべて解除します(オプションでリソースを解放し、データベースとの接続を切断します)。
SAVEPOINT	S/E	後でロールバックする位置をトランザクション内に指定します。
SELECT	S/E	選択した値をホスト変数に割り当てて、1 つ以上の表、ビューまたはスナップショットからデータを取り出します。
UPDATE	S/E	表またはビューの実表の既存の値を変更します。
VAR	O/D	デフォルトのデータ型をオーバーライドし、特定の Oracle データ型をホスト変数に割り当てます。
WHENEVER	S/D	エラー状態および警告状態の処置を指定します。

E.2 コマンドの説明について

ディレクティブ、コマンドおよび句がアルファベット順に並んでいます。各コマンドの説明には、次の項目があります。

見出し	意味
用途	コマンドの基本的な用途を説明します。
前提条件	必要な権限と、コマンドを使用する前に実行する必要のあるステップを示します。特記されていないかぎり、ほとんどのコマンドではインスタンスでデータベースがオープンされている必要があります。
構文	コマンドのキーワードとパラメータを示します。
キーワードとパラメータ	各キーワードとパラメータの用途を説明します。

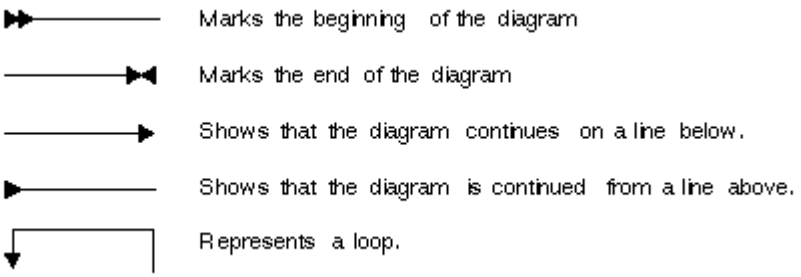
見出し	意味
使用上のノート	コマンドの使用方法と使用条件を説明します。
例	コマンドの例文を示します。
関連項目	関連するコマンド、句およびこのマニュアルの関連項目を示します。

E.3 構文図の読み方

埋込みSQLの構文の説明には、わかりやすい構文図を使用しています。これらは、正しい構文を示す線と矢印の図です。使用した経験がなくても、心配しないでください。この項では必要な知識を説明します。

構文図の論理的な流れを理解すれば、役に立つガイドになります。構文図をたどれば、どんな埋込みSQL文も検証または作成できます。

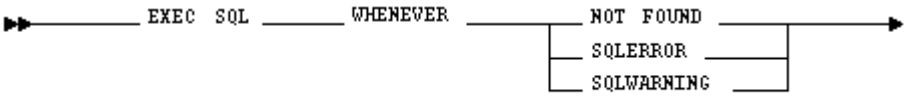
構文図は、線と矢印を使用して、文を作成するためのコマンド、パラメータおよびその他の言語要素の並べ方を示します。各図を左から右に矢印が指す方向にたどってください。次の記号が使用されています。



コマンドおよびその他のキーワードは、大文字で表記されています。パラメータは小文字で表記されています。演算子、デリミタおよび終了記号は普通に表記されています。「はじめに」で定義されている表記規則に従い、セミコロンで文を終了します。

構文図に複数のパスがある場合は、任意のパスを選択できます。

キーワード、演算子またはパラメータの選択肢が複数ある場合は、オプションを縦に並べて示します。次の例では、まず縦方向を選択した後、横方向に進めます。

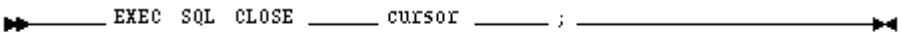


この図は、次の文がすべて有効であることを示しています。

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

E.3.1 必須のキーワードおよびパラメータ

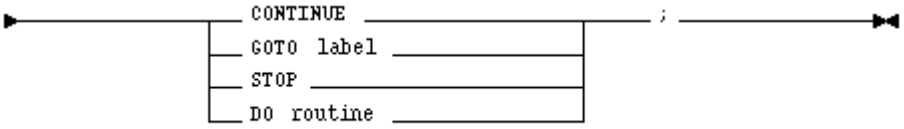
必須のキーワードおよびパラメータは、単一または代替の選択肢を縦に並べた状態で示します。単独の必須キーワードおよびパラメータはメイン・パス、つまり現在たどっている横線上に現れます。次の例では、*cursor*が必須パラメータです。



*emp_cursor*という名前のカーソルがある場合、この構文図によると、次の文は有効です。

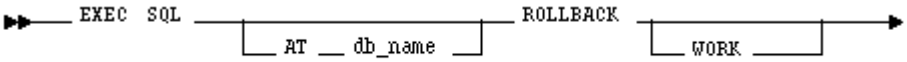
```
EXEC SQL CLOSE emp_cursor;
```

複数のキーワードまたはパラメータがメイン・パス上に縦に並んでいる場合は、その中のいずれかが必須になります。つまり、キーワードやパラメータを1つ選択する必要がありますが、それはメイン・パス上にあるものでなくてもかまいません。次の例では、4つのアクションのうち1つを選択する必要があります。



E.3.2 オプションのキーワードとパラメータ

キーワードおよびパラメータがメイン・パスの下に並べられている場合は、オプションです。つまり、それらの1つを選択する必要はありません。次の例では、上方向にたどらずに、メイン・パスを続けることができます。

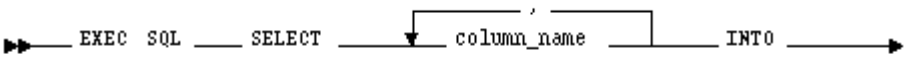


この図では、*oracle2*の名前のデータベースが存在する場合、次の文はすべて有効です。

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

E.3.3 構文ループ

ループは、その中の構文を何回でも繰り返せることを示します。次の例では、*column_name*がループの中にあります。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。

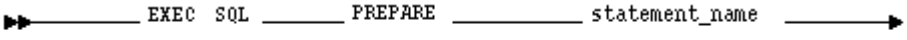


DEBIT、CREDITおよびBALANCEが列名の場合、この図によれば、次の文はすべて有効です。

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

E.3.4 マルチパート図

複数パーツの図では、メイン・パスがすべて端から端まで続いていると考えます。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 sql_statement FROM :sql_string;
```

E.3.5 データベース・オブジェクト

表や列などのOracleオブジェクトの名前の長さは、30文字以内にする必要があります。先頭文字は必ず英文字を使用します。残りは英文字、数字、ドル記号($)、ポンド記号(#)およびアンダースコア(_)の任意の組合せを使用できます。

ただし、Oracleオブジェクトの識別名を二重引用符(")で囲むと、有効な文字を任意に組み合せて使用できます。この場合空白は有効ですが、引用符は無効です。

Oracle識別名は、引用符で囲まれている場合を除き、大文字と小文字の区別がありません。

E.4 ALLOCATE (実行可能埋込みSQL拡張機能)

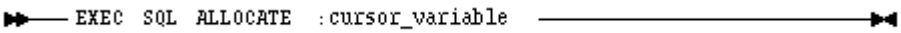
E.4.1 Allocateの用途

カーソル変数がPL/SQLブロックで参照されるように割り当てます。

E.4.2 Allocateの前提条件

カーソル変数にメモリーを割り当てる前に、SQL_CURSOR型のカーソル変数を宣言する必要があります。

E.4.3 Allocateの構文



E.4.4 Allocateのキーワードおよびパラメータ

:cursor_variable

割り当てるカーソル変数。

E.4.5 Allocateの使用上のノート

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。

例

この部分的な例では、Pro*C/C++の埋込みSQLプログラムでのALLOCATEコマンドの使用方法を示しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 SQL_CURSOR emp_cv;
 struct{ ... } emp_rec;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE emp_cv;
EXEC SQL EXECUTE
 BEGIN
 OPEN :emp_cv FOR SELECT * FROM emp;
 END;
END-EXEC;
for (;;)
{ EXEC SQL FETCH :emp_cv INTO :emp_rec;
}
```

E.4.6 Allocateの関連トピック

CLOSE(実行可能埋込みSQL)、EXECUTE(実行可能埋込みSQL)およびFETCH(実行可能埋込みSQL)

E.5 CLOSE (実行可能埋込みSQL)

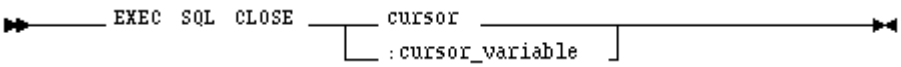
E.5.1 CLOSEの用途

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。

E.5.2 CLOSEの前提条件

カーソルまたはカーソル変数がオープンの状態で、MODE=ANSIであることが必要です。

E.5.3 CLOSEの構文



E.5.4 CLOSEのキーワードとパラメータ

cursor

クローズするカーソル。

cursor_variable

クローズするカーソル変数。

E.5.5 CLOSEの使用上のノート

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コマンドの効果が変わります。これらのオプションの詳細は、Oracleプリコンパイラの実行を参照してください。

E.5.6 CLOSEの例

この例では、CLOSEコマンドの使用方法を示しています。

```
EXEC SQL CLOSE emp_cursor;
```

E.5.7 CLOSEの関連項目

DECLARE CURSOR(埋込みSQLディレクティブ)、OPEN(実行可能埋込みSQL)およびPREPARE(実行可能埋込みSQL)

E.6 COMMIT (実行可能埋込みSQL)

E.6.1 COMMITの用途

データベースへの変更をすべて確定し、オプションですべてのリソースを解放して、Oracleデータベースとの接続を切断し、現行のトランザクションを終了します。

E.6.2 COMMITの前提条件

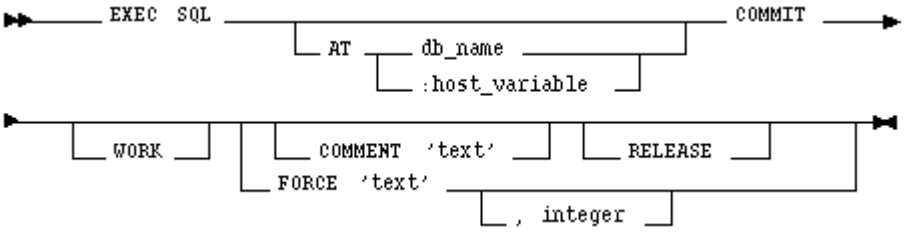
現行のトランザクションをコミットするために必要な権限はありません。

自分がコミットしたインダウト分散トランザクションを手動でコミットする場合は、FORCE TRANSACTIONシステム権限が必要です。別のユーザーがコミットしたインダウト分散トランザクションを手動でコミットする場合は、FORCE ANY TRANSACTIONシステム権限が必要です。

DBMS MACモードでOracleを使用しているとき、インダウト・トランザクションをコミットできるのは、DBMSラベルがトランザクションのラベル、およびこのトランザクションをコミットしたユーザーの作成ラベルと一致するか、次の条件のいずれかを満たしている場合のみです。

- トランザクションのラベルまたはユーザーの作成ラベルが、BMSラベルより上位である場合は、READUPおよびWRITEUPシステム権限が必要です。
- トランザクションのラベルまたはユーザーの作成ラベルがDBMSラベルより下位の場合は、WRITEDOWNシステム権限が必要です。
- トランザクションのラベルまたはユーザーの作成ラベルがDBMSラベルと同等でない場合は、READUP、WRITEUPおよびWRITEDOWNシステム権限が必要です。

E.6.3 COMMITの構文



E.6.4 COMMITのキーワードとパラメータ

AT

COMMIT文の発行先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。

- *db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。
- :*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、Oracleではデフォルトのデータベースに文が発行されます。

WORK

標準SQLに準拠している場合にのみサポートされます。COMMIT文とCOMMIT WORK文は同じです。

COMMENT

カレント・トランザクションに対応付けるコメントを指定します。*text*'は、50文字以内の引用符付きリテラルで、トランザクションがインダウトになると、OracleではトランザクションIDとともにデータ・ディクショナリ・ビューDBA_2PC_PENDINGに格納されます。

RELEASE

リソースをすべて解放し、アプリケーションのOracleデータベースとの接続を切断します。

FORCE

インダウトの分散トランザクションを手動でコミットします。トランザクションは、ローカルまたはグローバル・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。このトランザクションのIDを確認する場合は、データ・ディクショナリ・ビューDBA_2PC_PENDINGを問い合せます。また、オプションの*integer*を使用してトランザクションにシステム変更番号(SCN)を明示的に割り当てることができます。整数を省略すると、トランザクションは現行のSCNを使用してコミットされます。

E.6.5 COMMITの使用上のノート

プログラムの最後のトランザクションは、COMMITまたはROLLBACKコマンドとRELEASEオプションを使用して、必ず明示的にコミットまたはロールバックしてください。プログラムが異常終了すると、Oracleによって変更は自動的にロールバックされます。

COMMITコマンドは、ホスト変数またはプログラムの制御の流れには影響を与えません。このコマンドの詳細は、トランザクションの定義および制御を参照してください。

例

この例では、埋込みSQL COMMITコマンドの使用方法を示しています。

```
EXEC SQL AT sales_db COMMIT RELEASE;
```

E.6.6 COMMITの関連項目

ROLLBACK(実行可能埋込みSQL)およびSAVEPOINT(実行可能埋込みSQL)

E.7 CONNECT(実行可能埋込みSQL拡張機能)

E.7.1 CONNECTの用途

Oracleデータベースに接続します。

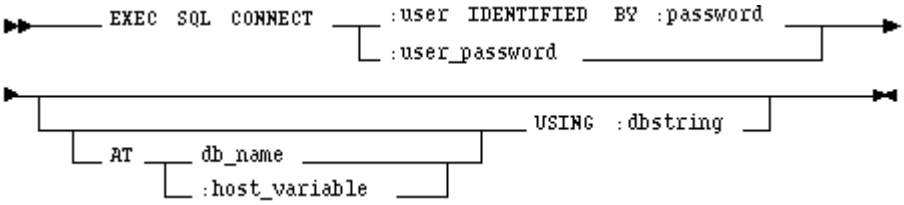
E.7.2 CONNECTの前提条件

指定するデータベースでのCREATE SESSIONシステム権限が必要です。

DBMS MACモードでOracleを使用している場合、オペレーティング・システム・ラベルが、作成ラベルおよびCREATE SESSIONシステム権限を付与されたラベルのどちらよりも上位である必要があります。また、オペレーティング・システム・ラベルは、オペレーティング・システムの同等ラベルDBHIGHとDBLOWの間に位置することも必要です。

Oracleをオペレーティング・システムMACモードで使用している場合、オペレーティング・システムのラベルは接続先のデータベースのラベルと一致する必要があります。

E.7.3 CONNECTの構文



E.7.4 CONNECTのキーワードとパラメータ

:user :password

ユーザー名とパスワードを個別に指定します。

:user_password

スラッシュ(/)で区切られたOracleのユーザー名とパスワードを含む1つのホスト変数。

Oracleで使用しているオペレーティング・システムを介した接続を検証するには、*:user_password*値として「/」を指定します。

AT

接続先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

*:host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

USING

デフォルト以外のデータベースへの接続に使用するSQL*Netデータベース指定文字列を指定します。この句を省略した場合は、デフォルトのデータベースに接続します。

E.7.5 CONNECTの使用上のノート

プログラムは複数の接続を持つことができますが、デフォルト・データベースには1度しか接続できません。このコマンドの詳細は、プログラム要件への対応を参照してください。

例

次の例では、CONNECTの使用方法を説明しています。

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

さらにこの文を使用して、*userid*の値を*username*の値にしたり、AAAAA/BBBBBのように*password*を「/」で区切ったものを設定できます。

```
EXEC SQL CONNECT :userid
```

E.7.6 CONNECTの関連項目

COMMIT(実行可能埋込みSQL)、DECLARE DATABASE(Oracle埋込みSQLディレクティブ)およびROLLBACK(実行可能埋込みSQL)

E.8 DECLARE CURSOR (埋込みSQLディレクティブ)

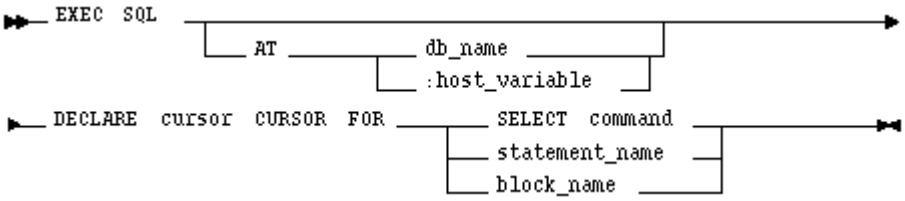
E.8.1 DECLARE CURSORの用途

カーソルに名前を付け、SQL文またはPL/SQLブロックに関連付けて宣言します。

E.8.2 DECLARE CURSORの前提条件

SQL文またはPL/SQLブロックの識別子にカーソルを関連付ける場合、DECLARE STATEMENT文でこの識別子を事前に宣言しておく必要があります。

E.8.3 DECLARE CURSORの構文



E.8.4 DECLARE CURSORのキーワードとパラメータ

AT

カーソルを宣言するデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、事前にDECLARE DATABASE文で宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、Oracleはデフォルトのデータベースに対してこのカーソルを宣言します。

cursor

宣言するカーソルの名前。

SELECTコマンド

カーソルに関連付けるSELECT文。直後の文にはINTO句を含めないでください。

statement_name block_name

カーソルと関連付ける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文で事前に宣言する必要があります。

E.8.5 DECLARE CURSORの使用上のノート

カーソルは、他の埋込みSQL文で参照する前に、宣言する必要があります。カーソル宣言のスコープはプリコンパイル・ユニット内全体になるため、各カーソルの名前はスコープ内で一意であることが必要です。1つのプリコンパイル・ユニット内で同じ名前のカーソルを複数宣言することはできません。

CURRENT OF構文を使用しているUPDATEまたはDELETE文のWHERE句ではカーソルを参照することができます。その場合カーソルはOPEN文でオープンされ、FETCH文で行に配置されます。このコマンドの詳細は、プログラム要件への対応を参照してください。

E.8.6 DECLARE CURSORの例

この例では、DECLARE CURSOR文の使用方法を示しています。

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

E.8.7 DECLARE CURSORの関連項目

CLOSE(実行可能埋込みSQL)、DECLARE DATABASE(Oracle埋込みSQLディレクティブ)、DECLARE STATEMENT(埋込みSQLディレクティブ)、DELETE(実行可能埋込みSQL)、FETCH(実行可能埋込みSQL)、OPEN(実行可能埋込みSQL)、PREPARE(実行可能埋込みSQL)、SELECT(実行可能埋込みSQL)およびUPDATE(実行可能埋込みSQL)

E.9 DECLARE DATABASE (Oracle埋込みSQLディレクティブ)

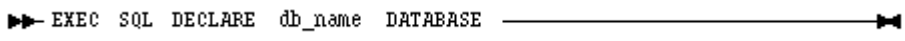
E.9.1 DECLARE DATABASEの用途

後続の埋込みSQL文でアクセスす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。

E.9.2 DECLARE DATABASEの前提条件

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る必要があります。

E.9.3 DECLARE DATABASEの構文



E.9.4 DECLARE DATABASEのキーワードとパラメータ

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。

E.9.5 DECLARE DATABASEの使用上のノート

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文でAT句を使用してそのデータベースを参照できるようにするためです。AT句を指定してCONNECT文を発行する前に、DECLARE DATABASE文で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。

このコマンドの詳細は、プログラム要件への対応を参照してください。

E.9.6 DECLARE DATABASEの例

この例では、DECLARE DATABASEディレクティブの使用方法を示しています。

```
EXEC SQL DECLARE oracle3 DATABASE
```

E.9.7 DECLARE DATABASEの関連項目

COMMIT(実行可能埋込みSQL)、CONNECT(実行可能埋込みSQL拡張機能)、DECLARE CURSOR(埋込みSQLディレクティブ)、DECLARE STATEMENT(埋込みSQLディレクティブ)、DELETE(実行可能埋込みSQL)、EXECUTE(実行可能埋込みSQL)、EXECUTE IMMEDIATE(実行可能埋込みSQL)、INSERT(実行可能埋込みSQL)、SELECT(実行可能埋込みSQL)およびUPDATE(実行可能埋込みSQL)

E.10 DECLARE STATEMENT (埋込みSQLディレクティブ)

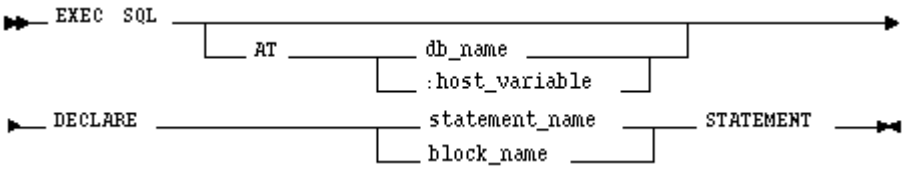
E.10.1 DECLARE STATEMENTの用途

他の埋込みSQL文で使用するSQL文またはPL/SQLブロックの識別子を宣言します。

E.10.2 DECLARE STATEMENTの前提条件

なし。

E.10.3 DECLARE STATEMENTの構文



E.10.4 DECLARE STATEMENTのキーワードとパラメータ

AT

SQL文またはPL/SQLブロックが宣言されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、Oracleではデフォルトのデータベースに対してSQL文またはPL/SQLブロックを宣言します。

statement_name block_name

文に対して宣言する識別子。

E.10.5 DECLARE STATEMENTの使用上のノート

DECLARE STATEMENT文を使用してSQL文またはPL/SQLブロックの識別子を宣言する必要があるのは、その識別子を参照するDECLARE CURSOR文が、埋込みSQLプログラム内で、その文またはブロックを解析して識別子と関連付けるPREPARE文よりも物理的に(論理的にではなく)前にある場合のみです。

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・ユニット内全体に及びます。このコマンドの詳細は、プログラム要件への対応および動的SQLの使用方法を参照してください。

E.10.6 DECLARE STATEMENTの例I

この例では、DECLARE STATEMENT文の使用方法を示しています。

```
EXEC SQL AT remote_db
 DECLARE my_statement STATEMENT
EXEC SQL PREPARE my_statement FROM :my_string
EXEC SQL EXECUTE my_statement
```

E.10.7 DECLARE STATEMENTの例II

Pro*C/C++埋込みSQLプログラムからのこの例では、DECLARE CURSOR文がPREPARE文の前にあるため、DECLARE STATEMENT文が必要です。

```
EXEC SQL DECLARE my_statement STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;
EXEC SQL PREPARE my_statement FROM :my_string;
...
```

E.10.8 DECLARE STATEMENTの関連項目

CLOSE(実行可能埋込みSQL)、DECLARE DATABASE(Oracle埋込みSQLディレクティブ)、FETCH(実行可能埋込みSQL)、OPEN(実行可能埋込みSQL)および"PREPARE(実行可能埋込みSQL)

E.11 DECLARE TABLE (Oracle埋込みSQLディレクティブ)

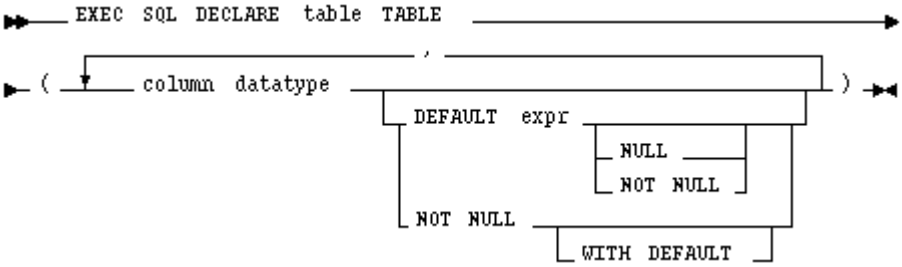
E.11.1 DECLARE TABLEの用途

各列のデータ型、デフォルト値、Oracleプリコンパイラによるセマンティック・チェックのためのNULLまたはNOT NULLの指定など、表またはビューの構造を定義します。

E.11.2 DECLARE TABLEの前提条件

なし。

E.11.3 DECLARE TABLEの構文



E.11.4 DECLARE TABLEのキーワードとパラメータ

table

宣言した表の名前。

column

*table*の列。

datatype

*column*のデータ型。

DEFAULT

*column*のデフォルト値を指定します。

NULL

*column*にNULLを含めてよいことを指定します。

NOT NULL

*column*にはNULLを含められないことを指定します。

WITH DEFAULT

IBM DB2データベースとの互換性のためにサポートされています。

E.11.5 DECLARE TABLEの使用上のノート

このコマンドの使用方法の詳細は、プログラム要件への対応を参照してください。

E.11.6 DECLARE TABLEの例

次の文により、PARTNO、BIN、QTY列のあるPARTS表を宣言します。

```
EXEC SQL DECLARE parts TABLE
 (partno NUMBER NOT NULL,
 bin NUMBER,
 qty NUMBER)
```

E.11.7 DECLARE TABLEの関連項目

なし。

E.12 DELETE(実行可能埋込みSQL)

E.12.1 DELETEの用途

表またはビューの実表から行を削除します。

E.12.2 DELETEの前提条件

表から行を削除するには、表が自分のスキーマにあるか、または表に対するDELETE権限が必要です。

ビューの実表から行を削除するには、ビューが属するスキーマの所有者に、実表に対するDELETE権限が必要です。また、ビューが自分以外のスキーマにある場合は、そのビューに対するDELETE権限が必要です。

DELETE ANY TABLEシステム権限があれば、どの表またはビューの実表からでも行を削除できます。

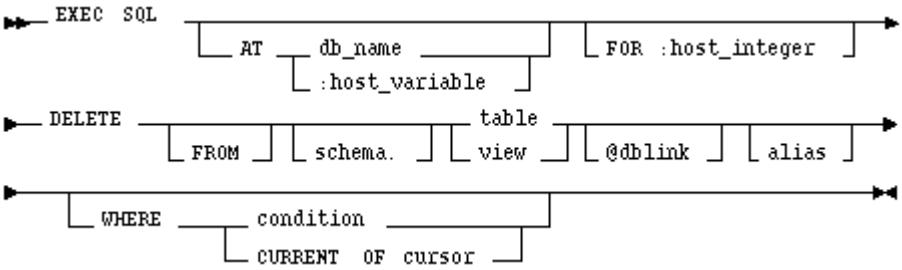
OracleをDBMS MACモードで使用している場合は、DBMSラベルが表またはビューの作成ラベルより上位にあるか、次の条件の1つを満たしている必要があります。

- 表またはビューの作成ラベルがDBMSラベルより上位にある場合は、READUPおよびWRITEUPシステム権限が必要です。
- 表またはビューの作成ラベルがDBMSラベルと同等でない場合は、READUP、WRITEUPおよびWRITEDOWNシステム権限が必要です。

さらに、各表を削除するには、DBMSラベルがその行のラベルと一致するか、次の条件の1つを満たす必要があります。

- 行のラベルがDBMSラベルより上位にある場合は、READUPおよびWRITEUPシステム権限が必要です。
- 行ラベルがDBMSラベルより下位にある場合は、WRITEDOWNシステム権限が必要です。
- 行ラベルがDBMSラベルと同等でない場合は、READUP、WRITEUPおよびWRITEDOWNシステム権限が必要です。

E.12.3 DELETEの構文



E.12.4 DELETEのキーワードとパラメータ

AT

DELETE文の発行先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、DELETE文はデフォルトのデータベースに対して発行されます。

FOR :*host_integer*

WHERE句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略した場合、Oracleは最小の配列の各コンポーネントにつき1回ずつ文が実行されます。

schema

表またはビューを含むスキーマ。*schema*を省略した場合、Oracleは表またはビューがユーザーの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。*view*を指定すると、Oracle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ベース・リンクの完全または部分的な名前。Oracleを分散オプションで使用している場合にのみ、リモートの表またはビューから行を削除できます。

*dblink*を省略した場合、Oracleは表またはビューがローカル・データベースにあるとみなします。

alias

表に割り当てられた別名。別名は一般に、相関問合せのあるDELETE文で使用されます。

WHERE

削除する行を指定します。

*condition*は、条件を満たす行のみを削除します。この条件には、ホスト変数およびオプションのインジケータ変数を含めることができます。

CURRENT OFは、*cursor*によって最後にフェッチされた行のみを削除します。*cursor*は、FOR UPDATE句により具体的に1つの表のみがロックされていないかぎり、結合を実行するSELECT文に関連付けることはできません。

この句を完全に省略した場合、Oracleは表またはビューからすべての行を削除します。

E.12.5 DELETEの使用上のノート

WHERE句のホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。変数がスカラーの場合、OracleではDELETE文が1回しか実行されません。変数が配列の場合、Oracleは配列のコンポーネント・セットごとに1回ずつこの文を実行します。1回の実行で0行、1行または複数行を削除できます。

WHERE句の配列ホスト変数は、サイズが異なっていてもかまいません。この場合、Oracleが文を実行する回数は、次のうちの小さい方の値によって決定します。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値

この条件を満たす行がない場合、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ついて削除された行の合計数がこの値に反映されます。

条件を満たす行がない場合、OracleからはSQLCAのSQLCODEを介してエラーが戻されます。WHERE句を省略した場合、OracleによりSQLCAのSQLWARNの第5コンポーネントに警告フラグが設定されます。このコマンドおよびSQLCAの詳細は、エラーの処理および診断を参照してください。

DELETE文ではコメントを使用して、指示、すなわちヒントをOracleオプティマイザに渡すことができます。オプティマイザでは、このヒントを使用して文の実行計画が選択されます。

E.12.6 DELETEの例

この例では、Pro*C/C++埋込みSQLプログラム内でのDELETE文の使用方法を示しています。

```
EXEC SQL DELETE FROM emp
 WHERE deptno = :deptno
 AND job = :job; ...
EXEC SQL DECLARE emp_cursor CURSOR
 FOR SELECT empno, comm
 FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH c1
 INTO :emp_number, :commission;
EXEC SQL DELETE FROM emp
 WHERE CURRENT OF emp_cursor;
```

E.12.7 DELETEの関連項目

DECLARE DATABASE(Oracle埋込みSQLディレクティブ)およびDECLARE STATEMENT(埋込みSQLディレクティブ)

E.13 DESCRIBE(実行可能埋込みSQL)

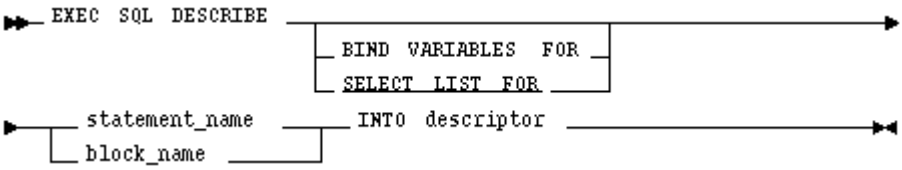
E.13.1 DESCRIBEの用途

動的SQL文またはPL/SQLブロックのホスト変数の説明を保持する記述子を初期化します。

E.13.2 DESCRIBEの前提条件

埋込みSQL PREPARE文を使用して、SQL文またはPL/SQLブロックを事前に準備しておく必要があります。

E.13.3 DESCRIBEの構文



E.13.4 DESCRIBEのキーワードとパラメータ

BIND VARIABLES

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

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 block_name

PREPARE文で事前に準備した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。

E.13.5 DESCRIBEの使用上のノート

埋込みSQLプログラム内のバインド記述子または選択記述子を操作するには、その前にDESCRIBE文を発行する必要があります。

入力変数と出力変数の両方を同じ記述子に記述することはできません。

DESCRIBE文で検出される変数の数は、一意に名前が指定されたプレースホルダの合計数ではなく、準備するSQL文またはPL/SQLブロックのプレースホルダの合計数です。このコマンドの詳細は、動的SQLの使用方法を参照してください。

E.13.6 DESCRIBEの例

この例では、Pro*C埋込みSQLプログラムでのDESCRIBE文の使用方法を示しています。

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL DECLARE emp_cursor
 FOR SELECT empno, ename, sal, comm
 FROM emp
 WHERE deptno = :dept_number
EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement
 INTO bind_descriptor;
```

```
EXEC SQL OPEN emp_cursor
 USING bind_descriptor;
EXEC SQL DESCRIBE SELECT LIST FOR my_statement
 INTO select_descriptor;
EXEC SQL FETCH emp_cursor
 INTO select_descriptor;
```

E.13.7 DESCRIBEの関連項目

PREPARE(実行可能埋込みSQL)

E.14 EXECUTE ...END-EXEC (実行可能埋込みSQL拡張機能)

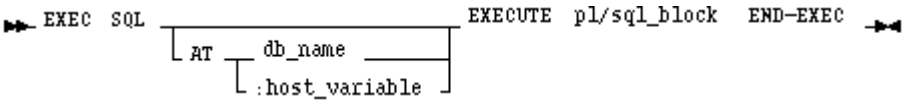
E.14.1 EXECUTE ... END-EXECの用途

Oracleプリコンパイラ・プログラムに無名PL/SQLブロックを埋め込みます。

E.14.2 EXECUTE ... END-EXECの前提条件

なし。

E.14.3 EXECUTE ... END-EXECの構文



E.14.4 EXECUTE ... END-EXECのキーワードおよびパラメータ

AT

PL/SQLブロックが実行されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、PL/SQLブロックはデフォルトのデータベースに対して実行されます。

pl/sql_block

END-EXEC

Oracleプリコンパイラ・プログラムで使用されるプログラミング言語に関係なく、埋込みPL/SQLブロックの後に指定する必要があります。もちろん、END-EXECキーワードの後にはその言語の埋込みSQL文終了記号を入れる必要があります。

E.14.5 EXECUTE ... END-EXECの使用上のノート

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ブロックの埋込みの詳細は、埋込みPL/SQLの使用方法を参照してください。

E.14.6 EXECUTE ... END-EXECの例

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
```

E.14.7 EXECUTE ... END-EXECの関連項目

EXECUTE IMMEDIATE(実行可能埋込みSQL)

E.15 EXECUTE (実行可能埋込みSQL)

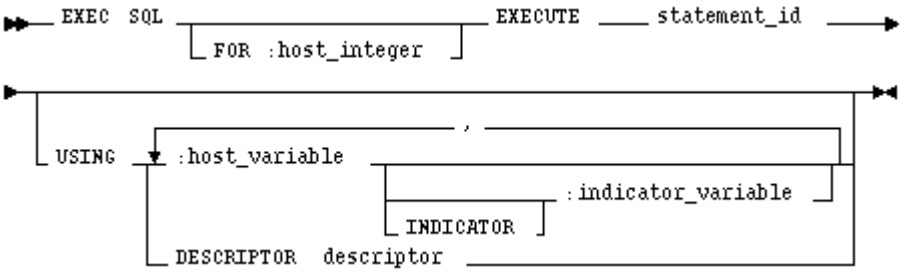
E.15.1 EXECUTEの用途

埋込みSQL PREPARE文を使用して事前に準備されたDELETE文、INSERT文またはUPDATE文、あるいはPL/SQLブロックを実行します。

E.15.2 EXECUTEの前提条件

SQL PREPARE文を使用して、まずSQL文またはPL/SQLブロックを準備する必要があります。

E.15.3 EXECUTEの構文



E.15.4 EXECUTEのキーワードとパラメータ

FOR :*host_integer*

USING句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略した場合、Oracleでは最小の配列のコンポーネントごとに文が1回ずつ実行されます。

statement_id

実行するSQL文またはPL/SQLブロックに関連付けられるプリコンパイラ識別子。プリコンパイラ識別子を文またはPL/SQLブロックに関連付けるには、埋込みSQL PREPAREコマンドを使用します。

USING

オプションのインジケータ変数を使用して、Oracleで実行する文に入力変数として代入するホスト変数のリストを指定します。ホスト変数およびインジケータ変数は、すべてがスカラーであるか、あるいはすべてが配列であることが必要です。

E.15.5 EXECUTEの使用上のノート

このコマンドの詳細は、動的SQLの使用方法を参照してください。

E.15.6 EXECUTEの例

この例では、Pro*C/C++埋込みSQLプログラムでのEXECUTE文の使用方法を示しています。

```
EXEC SQL PREPARE my_statement
 FROM :my_string;
EXEC SQL EXECUTE my_statement
 USING :my_var;
```

E.15.7 EXECUTEの関連項目

DECLARE DATABASE(Oracle埋込みSQLディレクティブ)およびPREPARE(実行可能埋込みSQL)

E.16 EXECUTE IMMEDIATE (実行可能埋込みSQL)

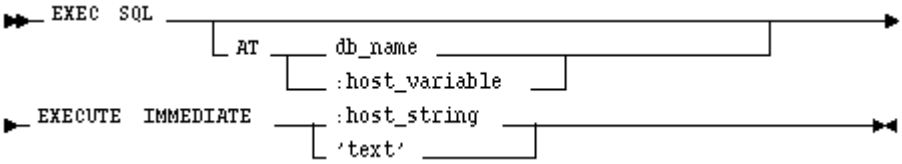
E.16.1 EXECUTE IMMEDIATEの用途

ホスト変数を含まないDELETE文、INSERT文またはUPDATE文またはPL/SQLブロックを準備し、実行します。

E.16.2 EXECUTE IMMEDIATEの前提条件

なし。

E.16.3 EXECUTE IMMEDIATEの構文



E.16.4 EXECUTE IMMEDIATEのキーワードとパラメータ

AT

SQL文またはPL/SQLブロックが実行されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、文またはブロックはデフォルトのデータベースに対して実行されます。

:*host_string*

実行するSQL文またはPL/SQLブロックが値であるホスト変数。

text

実行するSQL文またはPL/SQLブロックを含む引用符付きのテキスト・リテラル。

SQL文は、DELETE文、INSERT文またはUPDATE文のいずれかです。

E.16.5 EXECUTE IMMEDIATEの使用上のノート

EXECUTE IMMEDIATE文を発行すると、Oracleでは指定した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。

このコマンドの詳細は、動的SQLの使用方法を参照してください。

E.16.6 EXECUTE IMMEDIATEの例

この例では、EXECUTE IMMEDIATE文の使用方法を示しています。

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = 9460'
```

E.16.7 EXECUTE IMMEDIATEの関連項目

EXECUTE(実行可能埋込みSQL)およびPREPARE(実行可能埋込みSQL)

E.17 FETCH (実行可能埋込みSQL)

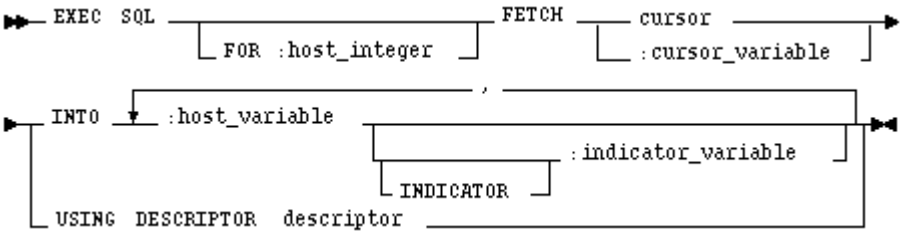
E.17.1 FETCHの用途

選択リストの値を変数に割り当てて、問合せによって戻された1行以上の行を取得します。

E.17.2 FETCHの前提条件

OPEN文を使用して、まずカーソルをオープンする必要があります。

E.17.3 FETCHの構文



E.17.4 FETCHのキーワードおよびパラメータ

FOR :*host_integer*

配列ホスト変数を使用している場合、フェッチする行数を制限します。この句を省略した場合、Oracleは最小の配列を満たすのに十分な数の行をフェッチします。

cursor

DECLARE CURSOR文を使用して宣言したカーソル。FETCH文は、カーソルに関連付けられた問合せにより選択された行のうち1行を戻します。

:*cursor_variable*

ALLOCATE文を使用して割り当てたカーソル変数。FETCH文は、カーソル変数に関連付けられた問合せにより選択された行のうちの1行を戻します。

INTO

フェッチしたデータを格納するホスト変数およびオプションのインジケータ変数のリストを指定します。これらのホスト変数および標

識変数は、プログラム内で宣言されている必要があります。

USING

DESCRIBE文で事前に参照している記述子を指定します。この句は、動的埋込みSQL方法4以外では使用しないでください。また、カーソル変数を使用しているときも、USING句は適用されません。

E.17.5 FETCHの使用上のノート

FETCH文は、アクティブ・セットの行を読み取り、結果が含まれる出力変数の名前を付けます。対応付けられたホスト変数がNULLの場合、インディケータ変数の値は-1に設定されます。最初のカーソルのFETCH文は、必要に応じて、アクティブ・セットの行のソートも行います。

フェッチする行数は、出力ホスト変数のサイズおよびFOR句で指定した値で指定します。データを取得するホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。スカラーの場合は、Oracleは1行のみフェッチします。配列の場合、Oracleは配列を満たすのに十分な数の行をフェッチします。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracleがフェッチする行数は、次の値のうち小さい方です。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値

フェッチする行数は、実際に問合せを満たす行数によってさらに限定できます。

FETCH文が、問合せで戻された行すべてを取得しなかった場合、カーソルは戻された次の行に配置されます。問合せで戻された最後の行を取得すると、次のFETCH文を実行すると、SQLCAのSQLCODE要素にエラー・コードが戻されることになります。

FETCHコマンドにはAT句がないので注意してください。カーソルでアクセスするデータベースは、DECLARE CURSOR文で指定する必要があります。

FETCH文では、アクティブ・セット内で進めるのは前方向のみです。すでにフェッチした行に戻る場合は、カーソルを再オープンして各行を順番に取り出す必要があります。アクティブ・セットを変更するには、新しい値をカーソルの問合せの入力ホスト変数に割り当て、カーソルを再オープンします。

E.17.6 FETCHの例

この例では、擬似コード埋込みSQLプログラムでのFETCHコマンドの使用方法を示しています。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
 SELECT job, sal FROM emp WHERE deptno = 30;
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
LOOP
 EXEC SQL FETCH emp_cursor INTO :job_title1, :salary1;
 EXEC SQL FETCH emp_cursor INTO :job_title2, :salary2;
...
END LOOP;
...
```

E.17.7 FETCHの関連項目

CLOSE(実行可能埋込みSQL)、DECLARE CURSOR(埋込みSQLディレクティブ)、OPEN(実行可能埋込みSQL)およびPREPARE(実行可能埋込みSQL)

E.18 INSERT(実行可能埋込みSQL)

E.18.1 INSERTの用途

表またはビューの実表に行を追加します。

E.18.2 INSERTの前提条件

表に行を挿入するには、表が自分のスキーマにあるか、または表に対するINSERT権限が必要です。

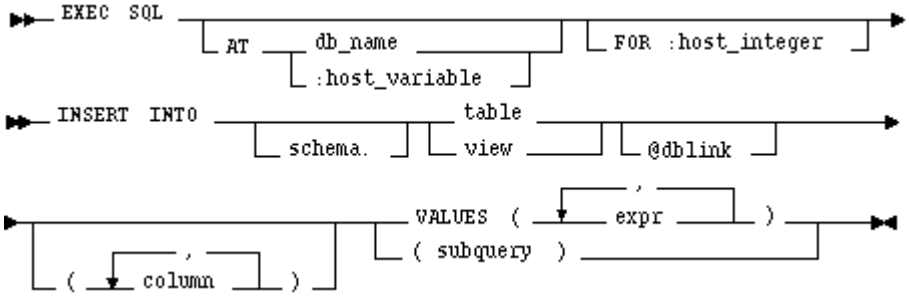
ビューの実表に行を挿入するには、ビューが属するスキーマの所有者に、実表に対するINSERT権限が必要です。また、ビューが自分以外のスキーマにある場合は、そのビューに対するINSERT権限が必要です。

INSERT ANY TABLEシステム権限があれば、どの表またはビューの実表にも行を挿入できます。

OracleをDBMS MACモードで使用している場合は、DBMSラベルが表またはビューの作成ラベルと一致する必要があります。

- 表またはビューの作成ラベルがDBMSラベルより上位にある場合は、WRITEUPシステム権限が必要です。
- 表またはビューの作成ラベルがDBMSラベルより下位にある場合は、WRITEDOWNシステム権限が必要です。
- 表またはビューの作成ラベルがDBMSラベルと同等でない場合は、WRITEUPおよびWRITEDOWNシステム権限が必要です。

E.18.3 INSERTの構文



E.18.4 INSERTのキーワードとパラメータ

AT

INSERT文を実行するデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、INSERT文はデフォルトのデータベースで実行されます。

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。この句を省略した場合、Oracleは最小の配列の各コンポーネントにつき1回ずつ文が実行されます。

schema

表またはビューを含むスキーマ。*schema*を省略した場合、Oracleは表またはビューがユーザーのスキーマ内にあるとみなします。

table view

行を挿入する表の名前。*view*を指定する場合、Oracleはビューの実表に行を挿入します。

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ベース・リンクの完全または部分的な名前。Oracleを分散オプションで使用している場合にのみ、リモートの表またはビューに行を挿入できます。

*dblink*を省略した場合、Oracleでは表またはビューがローカル・データベースにあるとみなされます。

column

表またはビューの列。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句または問合せで表のすべての列に値を指定する必要があります。

VALUES

表またはビューに挿入する値の表を指定します。ホスト変数には、オプションのインディケータ変数と合せて式も使用できます。VALUES句では、列リストの列ごとに式を指定する必要があります。

subquery

表に挿入される行を戻す副問合せ。この副問合せの選択リストの列数は、INSERT文の列リストの列数と同じであることが必要です。

E.18.5 INSERTの使用上のノート

WHERE句のホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。変数がスカラーの場合、OracleではINSERT文が1回しか実行されません。変数が配列の場合は、配列のコンポーネントのセットごとにINSERT文が1回実行され、1行ずつ挿入されます。

WHERE句の配列ホスト変数は、サイズが異なっていてもかまいません。この場合、Oracleが文を実行する回数は、次のうちの小さい方の値によって決定します。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値。

このコマンドの詳細は、埋込みSQLの使用方法を参照してください。

E.18.6 INSERTの例I

この例では、埋込みSQL INSERTコマンドの使用方法を示しています。

```
EXEC SQL
 INSERT INTO emp (ename, empno, sal)
 VALUES (:ename, :empno, :sal);
```

E.18.7 INSERTの例II

この例では、副問合せを使用した埋込みSQL INSERTコマンドを示しています。

```
EXEC SQL
 INSERT INTO new_emp (ename, empno, sal)
 SELECT ename, empno, sal FROM emp
 WHERE deptno = :deptno;
```

E.18.8 INSERTの関連項目

DECLARE DATABASE(Oracle埋込みSQLディレクティブ)

E.19 OPEN (実行可能埋込みSQL)

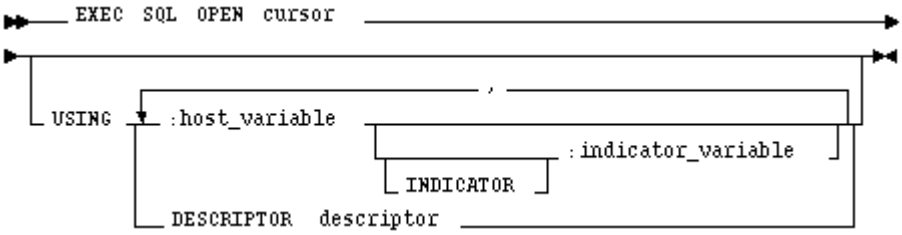
E.19.1 OPENの用途

カーソルをオープンし、関連付けられた問合せを評価して、USING句で指定したホスト変数名を問合せのWHERE句に代入します。

E.19.2 OPENの前提条件

カーソルは、オープンする前に埋込みSQLのDECLARE CURSOR文を使用して宣言する必要があります。

E.19.3 OPENの構文



E.19.4 OPENのキーワードとパラメータ

cursor

オープンするカーソル。

USING

関連付けられた問合せのWHERE句に代入するホスト変数を指定します。

:*host_variable*は、カーソルに関連付けられた文に代入する、オプションのインジケータ変数を持つホスト変数を指定します。

DESCRIPTOR

関連付けられた問合せのWHERE句に代入するホスト変数を説明する記述子を指定します。記述子は、DESCRIBE文で事前に初期化しておく必要があります。

代入は、位置に基づきます。この文で指定するホスト変数名は、対応付けられた問合せの変数名と異なってもかまいません。

E.19.5 OPENの使用上のノート

OPENコマンドは、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。このコマンドは、実際には行を取り出しません。行はFETCHコマンドを使用して取り出されます。

カーソルと一度オープンすると、その入力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。入力ホスト変数およびアクティブ・セットを変更するには、カーソルを再オープンする必要があります。

プログラム内のすべてのカーソルは、プログラムを開始するとき、またはCLOSEコマンドを使用して明示的にクローズした後はクローズ状態です。

カーソルは事前にクローズしなくても、再オープンできます。このコマンドの詳細は、埋込みSQLの使用方法を参照してください。

E.19.6 OPENの例

この例では、Pro*C/C++の埋込みSQLプログラムでのOPENコマンドの使用方法を示しています。

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

E.19.7 OPENの関連項目

CLOSE(実行可能埋込みSQL)。DECLARE CURSOR(埋込みSQLディレクティブ)、FETCH(実行可能埋込みSQL)およびPREPARE(実行可能埋込みSQL)

E.20 PREPARE (実行可能埋込みSQL)

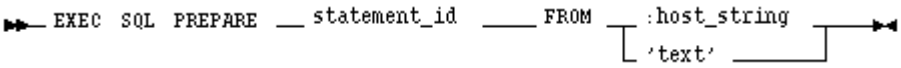
E.20.1 PREPAREの用途

ホスト変数で指定したSQL文またはPL/SQLブロックを解析し、識別子に関連付けます。

E.20.2 PREPAREの前提条件

なし。

E.20.3 PREPARE文の構文



E.20.4 PREPAREのキーワードおよびパラメータ

statement_id

準備済のSQL文またはPL/SQLブロックに関連付ける識別子。この識別子がすでに別の文またはブロックに割り当てられている場合は、以前の割当てが置き換えられます。

:*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。

text

準備するSQL文またはPL/SQLブロックを含む文字列リテラル。

E.20.5 PREPAREの使用上のノート

:*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コマンドのUSING句(入力ホスト変数)またはFETCHコマンドのINTO句(出力ホスト変数)に割り当てます。

SQL文は一度準備すると、何回でも実行できます。

E.20.6 PREPAREの例

この例では、Pro*C/C++埋込みSQLプログラムでのPREPARE文の使用方法を示しています。

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

E.20.7 PREPAREの関連項目

CLOSE(実行可能埋込みSQL)、DECLARE CURSOR(埋込みSQLディレクティブ)、FETCH(実行可能埋込みSQL)およびOPEN(実行可能埋込みSQL)

E.21 ROLLBACK (実行可能埋込みSQL)

E.21.1 ROLLBACKの用途

現行のトランザクションで実行した作業を取り消します。

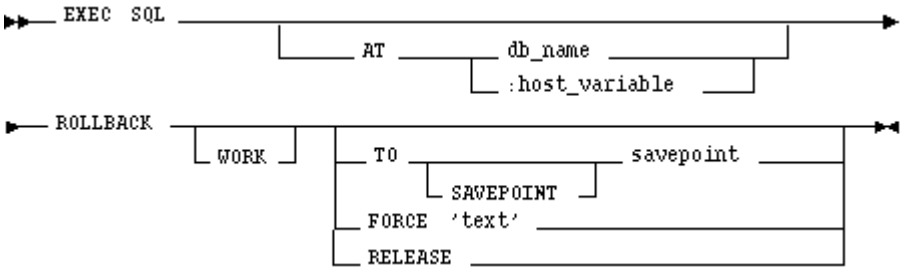
このコマンドは、インダウトの分散トランザクションで実行された作業を手動で取り消すときにも使用できます。

E.21.2 ROLLBACKの前提条件

現行のトランザクションをロールバックする場合、権限は不要です。

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システム権限が必要です。

E.21.3 ROLLBACKの構文



E.21.4 ROLLBACKのキーワードおよびパラメータ

WORK

オプションで、ANSIとの互換性のために用意されています。

TO

指定したセーブポイントまで現行の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。このトランザクションのIDを確認する場合は、データ・ディクショナリ・ビューDBA_2PC_PENDINGを問い合せます。

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

すべてのリソースを解放し、アプリケーションのOracleサーバーとの接続を切断します。RELEASE句は、SAVEPOINT句およびFORCE句とは併用できません。

E.21.5 ROLLBACKの使用上のノート

トランザクション(または論理作業単位)は、Oracleが1つの単位として扱う一連のSQL文です。トランザクションは、COMMIT、ROLLBACKまたはデータベースへの接続の後の最初の実行SQL文から始まります。トランザクションは、COMMIT文、ROLLBACK文またはデータベースとの接続を切断(意図的または不用意な切断)により終了します。Oracleでは、データ定義言語文の処理前および処理後に暗黒的COMMIT文が発行されます。

TO SAVEPOINT句を指定せずにROLLBACKコマンドを使用すると、次の処理が実行されます。

- トランザクションを終了します。
- カレント・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コマンドを使用すると、次の処理が実行されます。

- トランザクションのセーブポイント後の部分のみ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をすべて消去します。指定したセーブポイントは保持されるため、そのセーブポイントまで複数回ロールバックできます。指定したセーブポイントより前に作成されたセーブポイントも残ります。
- 指定したセーブポイント後に取得した表および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する必要があります。行を要求していない他のトランザクションは、すぐに行の要求およびアクセスができます。

アプリケーション・プログラムでは、COMMITまたはROLLBACK文のいずれかを使用してトランザクションを明示的に終了することをお薦めします。トランザクションを明示的にコミットしなかった場合にプログラムが異常終了すると、Oracleはコミットされていない最後のトランザクションをロールバックします。

E.21.6 ROLLBACKの例I

次の文は、現行のトランザクション全体をロールバックします。

```
EXEC SQL ROLLBACK;
```

E.21.7 ROLLBACKの例II

次の文はカレント・トランザクションをセーブポイントSP5までロールバックします。

```
EXEC SQL ROLLBACK TO SAVEPOINT sp5;
```

E.21.8 ROLLBACKの分散トランザクション

Oracleで分散オプションを使用すると、分散トランザクション、または複数のデータベースで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の場合と同様に、COMMIT文またはROLLBACK文を発行すれば済みます。

分散トランザクションのコミット・プロセス中にネットワーク障害が発生すると、トランザクションの状態が不明、つまりインダウトになる可能性があります。そのトランザクションに関連する他のデータベースの管理者に問い合せて、ローカル・データベースのトランザクションを手動でコミットするか、ロールバックするかを決定できます。ローカル・データベースのトランザクションは、FORCE句を指定したROLLBACK文を発行すれば、手動でロールバックできます。

インダウトのトランザクションを手動でセーブポイントまでロールバックすることはできません。

FORCE句を指定したROLLBACK文でロールバックできるのは、指定したトランザクションのみです。この文は、現行のトランザクションには影響しません。

E.21.9 ROLLBACKの例III

次の文は、インダウト分散トランザクションを手動でロールバックします。

```
EXEC SQL
 ROLLBACK WORK
 FORCE '25.32.87';
```

E.21.10 ROLLBACKの関連項目

COMMIT(実行可能埋込みSQL)およびSAVEPOINT(実行可能埋込みSQL)

E.22 SAVEPOINT (実行可能埋込みSQL)

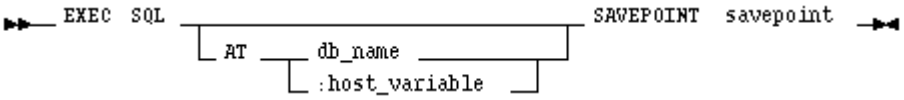
E.22.1 SAVEPOINTの用途

後でロールバックできる位置をトランザクション内に指定します。

E.22.2 SAVEPOINTの前提条件

なし。

E.22.3 SAVEPOINTの構文



E.22.4 SAVEPOINTのキーワードとパラメータ

AT

セーブポイントを宣言するデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、セーブポイントはデフォルトのデータベースに対して作成されます。

savepoint

作成するセーブポイントの名前。

E.22.5 SAVEPOINTの使用上のノート

このコマンドの詳細は、トランザクションの定義および制御を参照してください。

例

この例では、埋込みSQL SAVEPOINTコマンドの使用方法を示しています。

```
EXEC SQL SAVEPOINT save3;
```

E.22.6 SAVEPOINTの関連項目

COMMIT(実行可能埋込みSQL)およびROLLBACK(実行可能埋込みSQL)

E.23 SELECT (実行可能埋込みSQL)

E.23.1 SELECTの用途

ホスト変数に選択した値を割り当て、1つ以上の表、ビューまたはスナップショットからデータを取得します。

E.23.2 SELECTの前提条件

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にあるか、またはその表あるいはスナップショットに対するREADまたはSELECT権限が必要です。

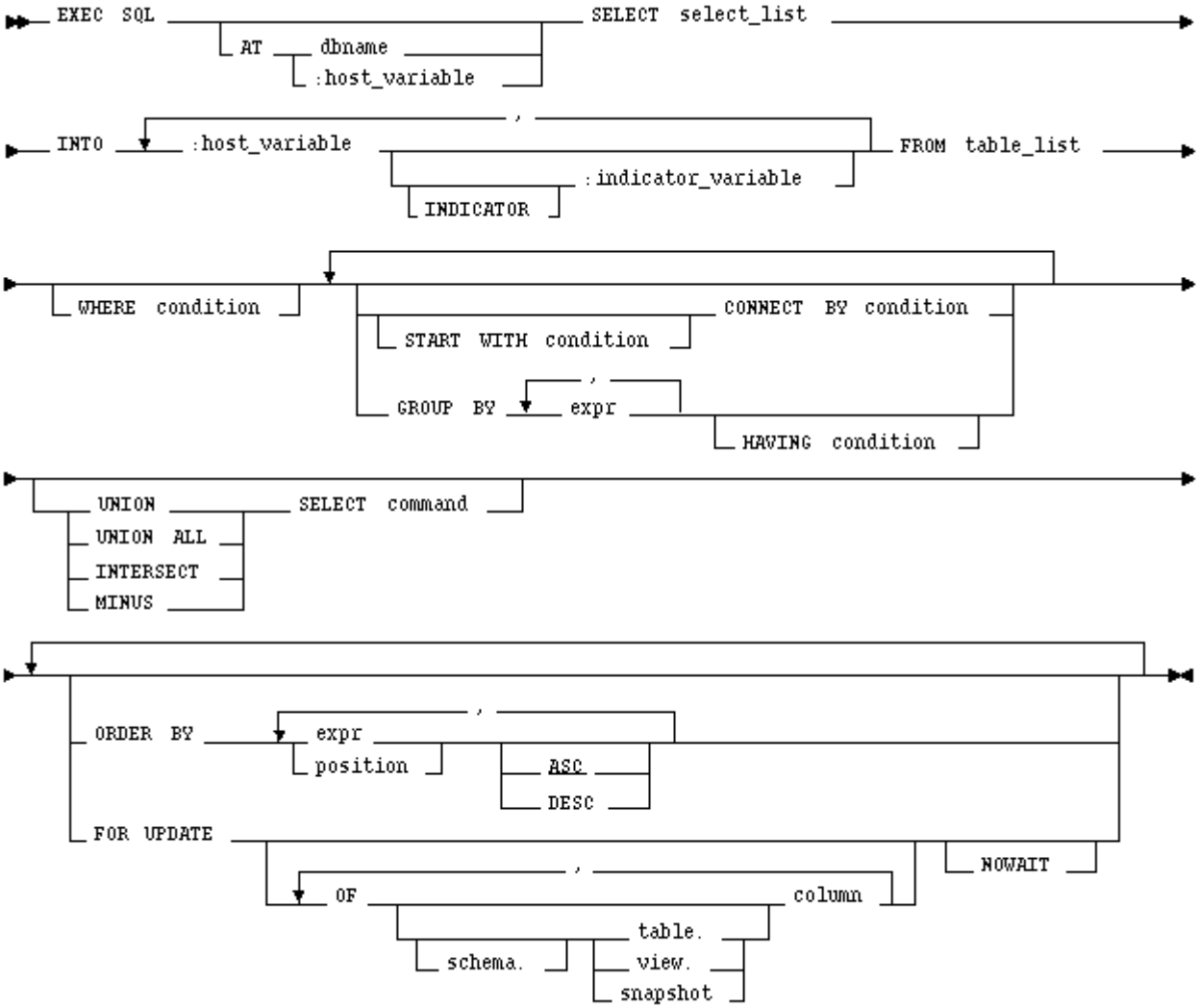
ビューの実表から行を選択するには、ビューが属するスキーマの所有者に、実表に対するREADまたはSELECT権限が必要です。また、ビューが自分以外のスキーマにある場合は、そのビューに対するREADまたはSELECT権限が必要です。

READ ANY TABLEまたはSELECT ANY TABLEシステム権限を使用すると、すべての表、スナップショットまたはビューの実表からデータを選択できます。

OracleをDBMS MACモードで使用している場合は、DBMSラベルが問合せ先の各表、ビューまたはスナップショットの作成ラベルより上位にあるか、READUPシステム権限が必要です。

READ権限は、SELECT ... FOR UPDATE操作には使用できません。

E.23.3 SELECTの構文



E.23.4 SELECTのキーワードとパラメータ

AT

SELECT文の発行先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、SELECT文はデフォルトのデータベースに対して発行されます。

select_list

非埋込みSELECTコマンドと同じですが、リテラルのかわりにホスト変数を使用できます。

INTO

SELECT文が戻すデータを受け取る出力ホスト変数とオプションのインディケータ変数を指定します。これらの変数は、すべてスカラーか、すべて配列である必要があります。ただし、配列は同じサイズでなくてもかまいません。

WHERE

戻される行を、条件がTRUEの行のみに制限します。*condition*には、ホスト変数を使用できますが、インジケータ変数は使用できません。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およびパラメータはすべて、非埋込みSQL SELECTコマンドと同じです。

E.23.5 SELECTの使用上のノート

WHERE句の条件を満たす行がない場合、行は取得されず、OracleからはSQLCAのSQLCODEコンポーネントを使用してエラー・コードが戻されます。

SELECT文ではコメントを使用して指示、すなわちヒントをOracleオプティマイザに渡すことができます。オプティマイザでは、このヒントを使用して文の実行計画が選択されます。

E.23.6 SELECTの例

この例では、埋込みSQL SELECTコマンドの使用方法を示しています。

```
EXEC SQL SELECT ename, sal + 100, job
 INTO :ename, :sal, :job
 FROM emp
 WHERE empno = :empno
```

E.23.7 SELECTの関連項目

DECLARE CURSOR(埋込みSQLディレクティブ)、DECLARE DATABASE(Oracle埋込みSQLディレクティブ)、EXECUTE(実行可能埋込みSQL)、FETCH(実行可能埋込みSQL)およびPREPARE(実行可能埋込みSQL)

E.24 UPDATE (実行可能埋込みSQL)

E.24.1 UPDATEの用途

表またはビューの実表で既存の値を変更します。

E.24.2 UPDATEの前提条件

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にあるか、または表に対するUPDATE権限が必要です。

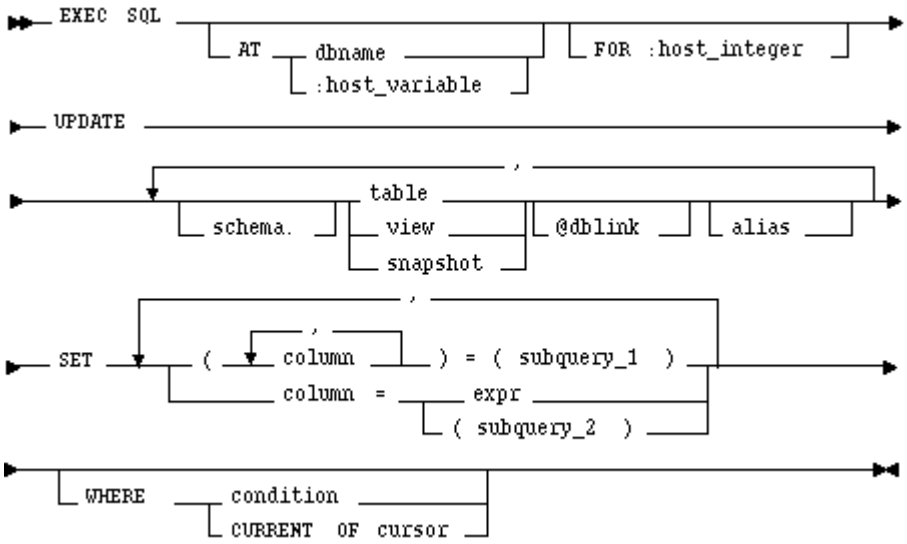
ビューの実表の値を更新するには、ビューが属するスキーマの所有者に、実表に対するUPDATE権限が必要です。また、ビューが自分以外のスキーマにある場合は、そのビューに対するUPDATE権限が必要です。

UPDATE ANY TABLEシステム権限があれば、どの表またはビューの実表でも値を更新できます。

OracleをDBMS MACモードで使用している場合は、DBMSラベルが表またはビューの作成ラベルと一致する必要があります。

- 表またはビューの作成ラベルがDBMSラベルより上位にある場合は、READUPおよびWRITEUPシステム権限が必要です
- 表またはビューの作成ラベルがDBMSラベルより下位にある場合は、WRITEDOWNシステム権限が必要です。
- 表またはビューの作成ラベルがDBMSラベルと同等でない場合は、READUP、WRITEUPおよびWRITEDOWNシステム権限が必要です。

E.24.3 UPDATEの構文



E.24.4 UPDATEのキーワードとパラメータ

AT

UPDATE文の発行先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。

*db_name*は、DECLARE DATABASE文で事前に宣言したデータベース識別子。

:*host_variable*は、値が事前に宣言した*db_name*であるホスト変数。

この句を省略した場合、UPDATE文はデフォルトのデータベースに対して発行されます。

FOR :*host_integer*

SET句およびWHERE句に配列ホスト変数が含まれる場合に、UPDATE文を実行する回数を制限します。この句を省略した場合、Oracleは最小の配列の各コンポーネントにつき1回ずつ文が実行されます。

schema

表またはビューを含むスキーマ。*schema*を省略した場合、Oracleは表またはビューがユーザーのスキーマ内にあるとみなします。

table view

更新する表の名前。*view*を指定する場合、Oracleではビューのベース表を更新します。

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ベース・リンクの完全または部分的な名前。Oracleを分散オプションで使用している場合にのみ、リモートの表またはビューを更新するためにデータベース・リンクを使用できます。

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ために使用する名前。

column

表またはビューで更新する列の名前。SET句から表の列を省略すると、その列値は変更されないままです。

expr

対応する列に割り当てられた新しい値。この式には、ホスト変数およびオプションの標識変数を含めることができます。

subquery_1

対応する列に割り当てられた新しい値を戻す副問合せ。

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。

WHERE

表またはビューで更新する行を指定します。

*condition*は、条件を満たす行のみを更新します。この条件には、ホスト変数およびオプションのインジケータ変数を含めることができます。

CURRENT OFは、*cursor*によって最後にフェッチされた行のみを更新します。*cursor*は、FOR UPDATE句により明示的に1つの表のみがロックされていないかぎり、結合を実行するSELECT文に関連付けることはできません。

この句を完全に省略した場合、Oracleは表またはビューのすべての行を更新します。

E.24.5 UPDATEの使用上のノート

SET句およびWHERE句のホスト変数は、すべてがスカラーか、あるいはすべてが配列であることが必要です。変数がスカラーの場合、OracleではUPDATE文が1回しか実行されません。変数が配列の場合、Oracleは配列のコンポーネント・セットごとに1回ずつこの文を実行します。1回の実行で、0行または1行、複数行を更新できます。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracleが文を実行する回数は、次のうちの小さい方の値によって決定します。

- 最小の配列のサイズ
- オプションのFOR句の:*host_integer*の値

更新された行の累計数は、SQLCAのSQLERRDコンポーネントの第3要素を介して戻されます。入力ホスト変数として配列を使用した場合、この数値はUPDATE文で処理された配列のすべてのコンポーネントにおよぶ更新数の合計を示します。条件を満たす行がない場合、行は更新されず、OracleからはSQLCAのSQLCODE要素を使用してエラー・メッセージが戻されます。WHERE句を省略した場合、すべての行が更新され、OracleではSQLCAのSQLWARN要素の第5コンポーネントに警告フラグが設定されます。

UPDATE文ではコメントを使用して、指示、すなわちヒントをOracleオプティマイザに渡すことができます。オプティマイザでは、このヒントを使用して文の実行計画が選択されます。

このコマンドの詳細は、埋込みSQLの使用方法およびトランザクションの定義および制御を参照してください。

E.24.6 UPDATEの例

次の例では、埋込みSQL UPDATEコマンドの使用方法を示しています。

```
EXEC SQL UPDATE emp
 SET sal = :sal, comm = :comm INDICATOR :comm_ind
 WHERE ename = :ename;

EXEC SQL UPDATE emp
 SET (sal, comm) =
 (SELECT AVG(sal)*1.1, AVG(comm)*1.1
 FROM emp)
 WHERE ename = 'JONES';
```

E.24.7 UPDATEの関連項目

DECLARE DATABASE(Oracle埋込みSQLディレクティブ)

E.25 VAR (Oracle埋込みSQLディレクティブ)

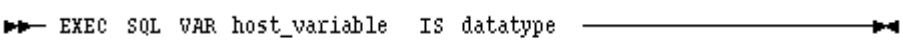
E.25.1 VARの用途

ホスト変数の同値化を実行するか、個々のホスト変数に特定のOracle外部データ型を割り当て、デフォルトのデータ型の割当てをオーバーライドします。

E.25.2 VARの前提条件

ホスト変数は、埋込みSQLプログラムの宣言部で事前に宣言しておく必要があります。

E.25.3 VARの構文



E.25.4 VARのキーワードおよびパラメータ

host_variable

Oracle外部データ型を割り当てるホスト変数。

datatype

Oracleプリコンパイラによって認識されるOracle外部データ型(Oracle内部データ型ではありません)。データ型には、長さ、精度または位取りを含めることができます。この外部データ型が*host_variable*に割り当てられます。外部データ型のリストは、プログラム要件への対応を参照してください。

E.25.5 VARの使用上のノート

ホスト変数の同値化は、一種のデータ型の同値化です。データ型の同値化は次の目的に有効です。

- 文字ホスト変数を自動的にNULLで終了させます。
- 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変換をオーバーライドします。

E.25.6 VARの例

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(2000)に同値化しています。

```
EXEC SQL BEGIN DECLARE SECTION;
 ...
 dept_name CHARACTER(15); -- default datatype is CHAR
 EXEC SQL VAR dept_name IS STRING; -- reset to STRING
 ...
 buffer CHARACTER(200); -- default datatype is CHAR
 EXEC SQL VAR buffer IS RAW(200); -- refer to RAW
 ...
EXEC SQL END DECLARE SECTION;
```

E.25.7 VARの関連項目

なし。

E.26 WHENEVER(埋込みSQLディレクティブ)

E.26.1 WHENEVERの用途

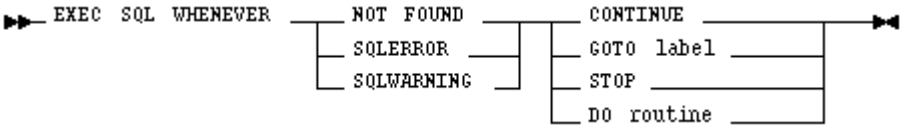
埋込みSQLプログラムの実行時に、エラーまたは警告が発生した場合の処置を指定します。

E.26.2 WHENEVERの前提条件

なし。

E.26.3 WHENEVERの構文

次の構文図は、WHENEVER文の作成方法を示しています。



E.26.4 WHENEVERのキーワードとパラメータ

NOT FOUND

エラー・コード+1403(または、MODE=ANSIのときは+100コード)をSQLCODEに戻す例外状態を示します。

SQLERROR

負のリターン・コードを戻す状態を示します。

SQLWARNING

致命的でない警告状態を示します。

CONTINUE

プログラムが次の文に進む必要があることを示します。

GOTO

プログラムが*label*で指定した名前の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

プログラムがホスト言語ルーチンをコールするように指示します。*routine*の構文はホスト言語によって異なります。使用する言語固有のOracleプリコンパイラ・ガイドの補足資料を参照してください。

E.26.5 WHENEVERの使用上のノート

WHENEVERコマンドを使用すると、埋込みSQL文でエラーまたは警告が発生したときに、プログラムからエラー処理のルーチンに制御を移すことができます。

WHENEVER文の有効範囲は、論理的なものではなく、位置的なものです。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に続くすべての埋込み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。

このコマンドの詳細は、トランザクションの定義および制御を参照してください。WHENEVER埋込みSQLコマンドとWHENEVER SQL*Plusコマンドを混同しないでください。

E.26.6 WHENEVERの例

次の例では、Pro*C/C++埋込みSQLプログラムでのWHENEVERコマンドの使用方法を示しています。

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...
EXEC SQL WHENEVER SQLERROR GOTO sql_error:
...
sql_error:
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK RELEASE;
```

E.26.7 WHENEVERの関連項目

なし。

索引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X