Oracle9*i*

アプリケーション開発者ガイド - 基礎編

リリース 2（9.2）

2002 年 7 月

部品番号 : J06267-01

ORACLE®

Oracle9*i* アプリケーション開発者ガイド - 基礎編 , リリース 2（9.2）

部品番号 : J06267-01

原本名：Oracle9*i* Application Developer's Guide - Fundamentals, Release 2 (9.2)

原本部品番号：A96590-01

原本著者：John Russell

原本協力者：T. Brooksfuller、T. Burroughs、M. Cowan、J. Levinger、R. Moran、R. Strohm、D. Alpern、G. Arora、C. Barclay、D. Bronnikov、T. Chang、M. Davidson、G. Doherty、D. Elson、A. Ganesh、M. Hartstein、J. Huang、N. Jain、R. Jenkins Jr.、S. Kotsovolos、S. Kumar、C. Lei、D. Lorentz、R. Murthy、R. Pang、B. Sinha、S. Vemuri、W. Wang、D. Wong、A. Yalamanchi、Q. Yu

グラフィック・デザイナ：V. Moore

目次

第 II 部　データベースの設計

2　スキーマ・オブジェクトの管理

3 データ型の選択

4 制約によるデータ整合性のメンテナンス

7 Oracle における SQL 文の処理方法

8 動的 SQL 文のコーディング

9 プロシージャおよびパッケージの使用

10 外部プロシージャのコール

第 III 部　アプリケーション・セキュリティ

11　アプリケーション開発者のためのデータベース・セキュリティの概要

12　アプリケーション・セキュリティ・ポリシーの実装

13 プロキシ認証

14 DBMS_OBFUSCATION_TOOLKIT を使用したデータの暗号化

第 IV 部　アクティブ・データベース

15　トリガーの使用

16　システム・イベントの処理

17　アプリケーションに対するパブリッシュ・サブスクライブ・モデルの使用

第 V 部　特殊アプリケーションの開発

18　PL/SQL を使用した Web アプリケーションの開発

19 Oracle 以外のアプリケーションから Oracle9*i* への移植

20 Oracle XA でのトランザクション・モニターの操作

索引

はじめに

このマニュアルでは、Oracle9*i* データベース用のアプリケーション開発に関する機能について説明します。このマニュアルの内容は、すべてのプラットフォーム上で同様に動作する機能に適用されるもので、システム固有の情報は含みません。

内容は次のとおりです。

対象読者

このマニュアルは、新しいアプリケーションを開発したり、既存のアプリケーションをOracle 環境で実行できるように変換するプログラマを対象にしています。このマニュアルは、システム・アナリスト、プロジェクト管理者およびデータベース・アプリケーション開発に関心がある方にも役立ちます。

このマニュアルは、アプリケーション・プログラミングの実践的な知識があり、Structured Query Language（SQL）を使用してリレーショナル・データベース・システムの情報にアクセスできることを前提としています。

また、このマニュアルの特定の項では、オブジェクト指向プログラミングの基本的な概念を理解していることも前提としています。

アプリケーション開発者の役割

通常、アプリケーション開発者に要求される作業は次のとおりです。

- SQL でのプログラミング。このマニュアルの主な情報源は、『Oracle9*i* SQL リファレンス』です。高度な問合せ機能、分析の実行および単一の問合せでのデータの取得の詳細は、『Oracle9*i* データ・ウェアハウス・ガイド』を参照してください。
- PL/SQL、Java、C/C++ などの他言語による SQL へのインタフェース作成。これらの他言語についての情報源を次に示します。
  - 『PL/SQL ユーザーズ・ガイドおよびリファレンス』
  - 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』
  - 『Oracle9*i* Java Developer's Guide』
  - 『Pro*C/C++ Precompiler プログラマーズ・ガイド』
  - 『Oracle Call Interface プログラマーズ・ガイド』および『Oracle C++ Call Interface プログラマーズ・ガイド』
  - Oracle Objects for OLE C++ Class Library のオンライン・ヘルプ
  - 『Oracle COM Automation 機能 開発者ガイド』
- 複数言語環境間の対話およびマッピングの設定。詳細は、10-1 ページの「外部プロシージャのコール」を参照してください。
- スキーマ・オブジェクトの使用。スキーマの一部またはすべてを設計する場合があります。また、既存のスキーマに合うコードを書き込む場合もあります。概要については2-1 ページの「スキーマ・オブジェクトの管理」を、詳細については『Oracle9*i* データベース管理者ガイド』を参照してください。
- データベース管理者との連携によるスキーマのバックアップおよびリストア。たとえば、システム障害の発生後、ステージング・マシンと本番マシン間での移行の実行時などに行います。

- ストアド・プロシージャ、制約およびトリガーの形式での、データベース自体へのアプリケーション・ロジックの構築。これによって、複数のアプリケーションで、エラーのチェックおよび処理に使用されるアプリケーション・ロジックおよびコードを再利用できます。データベースの機能の詳細は、9-1 ページの「プロシージャおよびパッケージの使用」、4-1 ページの「制約によるデータ整合性のメンテナンス」および 15-1 ページの「トリガーの使用」を参照してください。
- ある程度のパフォーマンス・チューニング。データベース管理者の支援が必要になる場合があります。詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』および『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。
- ある程度のデータベース管理（開発の維持またはシステムのテストが必要な場合）。管理の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。
- 『Oracle9*i* データベース・エラー・メッセージ』に示されているエラー・メッセージのデバッグおよび解析。
- ネットワーク上（特にインターネットまたは企業のイントラネットなど）でのアプリケーションの使用可能化。概要については 18-1 ページの「PL/SQL を使用した Web アプリケーションの開発」を、各種言語およびテクノロジの詳細については Oracle9*i*AS のマニュアルを参照してください。
- データの改ざんまたは不当アクセスを防ぐためのセキュリティ機能の構築。開発者は、第 III 部の「アプリケーション・セキュリティ」を参照してください。データベース・セキュリティに対する責任が広範囲に渡る場合は、『Oracle9*i* セキュリティ概要』を参照してください。
- アプリケーションがオブジェクト指向の場合のクラス構造の設計およびオブジェクト指向の方法の選択。詳細は、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』、『PL/SQL ユーザーズ・ガイドおよびリファレンス』および『Oracle9*i* Java Developer's Guide』を参照してください。

このマニュアルの構成

このマニュアルは、次の章で構成されています。

第 I 部 : 概要

ここでは、Oracle アプリケーションを開発するための様々な方法を紹介します。単一のアプリケーションに対し、2 つ以上の言語または開発環境を使用する必要がある場合があります。データベース機能によっては、特定の言語にのみサポートされるものや、特定の言語からのアクセスが簡単なものもあります。

第 1 章「Oracle プログラム環境の理解」では、それらの言語の強み、開発環境および Oracle が提供する API について説明します。

第 II 部 : データベースの設計

アプリケーションを開発する前に、関連するデータベースの特性について計画する必要があります。データベースに格納するすべてのもの、およびそれの構成方法を選択する必要があります。優れたデータベース設計によってパフォーマンスおよびスケーラビリティが保証され、エラー・チェック、高速データ・アクセスなどをデータベースで管理することによって、コーディングするアプリケーション・ロジックを削減できます。

第 2 章「スキーマ・オブジェクトの管理」では、表、ビュー、数値順序、シノニムなどのオブジェクトを管理する方法を説明します。索引およびクラスタを使用して、データ検索のパフォーマンスを向上させる方法についても説明します。

第 3 章「データ型の選択」では、ビジネス・データをデータベースでどのように表現するかについて説明します。データ型には、固定長文字列、可変長文字列、数値データ、日付、ロー・バイナリ・データ、行識別子（ROWID）などがあります。

第 4 章「制約によるデータ整合性のメンテナンス」では、エラー・チェック・ロジックをアプリケーションからデータベースへ移動するための制約の使用方法を説明します。

第 5 章「索引計画の選択」および第 6 章「索引構成表による索引アクセスの スピードアップ」では、問合せを高速化する方法を説明します。

第 7 章「Oracle における SQL 文の処理方法」では、アプリケーションで利用できるコミット、カーソル、ロックなどの SQL 項目について説明します。

第 8 章「動的 SQL 文のコーディング」では、動的 SQL について説明し、ネイティブ動的 SQL と DBMS_SQL パッケージを比較して、動的 SQL を使用するタイミングについて説明します。

第 9 章「プロシージャおよびパッケージの使用」では、再利用可能なプロシージャをデータベースに格納する方法について説明します。また、複数のプロシージャをパッケージにグループ化する方法についても説明します。

第 10 章「外部プロシージャのコール」では、計算集中型プロシージャの本体を、PL/SQL 以外の言語でコード化する方法について説明します。

第 III 部 : アプリケーション・セキュリティ

第 11 章「アプリケーション開発者のためのデータベース・セキュリティの概要」では、アプリケーションのセキュリティ問題を解決するために必要な予備知識を示します。

第 12 章「アプリケーション・セキュリティ・ポリシーの実装」では、アプリケーション・コンテキスト、ファイングレイン・アクセス・コントロール、仮想プライベート・データベースなど、アプリケーションで使用できる主なセキュリティ・メカニズムについて説明します。

第 13 章「プロキシ認証」では、Web サーバーまたはアプリケーション・サーバーによる認証と、データベース・サーバーのセキュリティ・メカニズムを統合する方法を説明します。

第 14 章「DBMS_OBFUSCATION_TOOLKIT を使用したデータの暗号化」では、許可されていないユーザーにデータが参照されても、デコードされないようにデータを保護する方法を説明します。

第 IV 部 : アクティブ・データベース

データベース自体に様々な種類のプログラミング論理を取り入れることができます。これによって、多くのアプリケーションでそのメリットを利用することができるので、コーディングの重複を削減できます。

第 15 章「トリガーの使用」では、SQL 文を実行する前後、または実行するかわりに、データベースで特別な処理を行う方法について説明します。データの検証や変換、データベース・アクセスのロギングなどにトリガーを使用できます。

第 16 章「システム・イベントの処理」では、ユーザー ID やデータベース名など、トリガーを起動するイベントに関する情報を取得する方法を説明します。

第 17 章「アプリケーションに対するパブリッシュ・サブスクライブ・モデルの使用」では、メッセージ機能またはキューイングとも呼ばれる非同期通信用の Oracle モデルを紹介します。

第 V 部 : 専用アプリケーションの開発

第 18 章「PL/SQL を使用した Web アプリケーションの開発」では、インターネット、電子メールなどで動作する動的 Web ページおよびアプリケーションを、PL/SQL 言語を使用して作成する方法を説明します。

第 19 章「Oracle 以外のアプリケーションから Oracle9i への移植」では、他のデータベース・システム用に作成されたアプリケーションを、Oracle9*i* 上で実行するために使用できる機能および方法を説明します。

第 20 章「Oracle XA でのトランザクション・モニターの操作」では、トランザクション・モニターで Oracle に接続する方法を説明します。

関連文書

詳細は、次の Oracle ドキュメントを参照してください。

PL/SQL について学習し、オラクル社が提供する SQL の手続き型拡張要素であるこの高水準プログラム言語の詳しい説明が必要な場合は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Oracle Call Interface（OCI）については、『Oracle Call Interface プログラマーズ・ガイド』および『Oracle C++ Call Interface プログラマーズ・ガイド』を参照してください。

OCI を使用して、Oracle サーバーにアクセスする 3GL アプリケーションを作成できます。

オラクル社は、プリコンパイラの Pro* シリーズも提供しています。これを使用することで、ご使用のアプリケーション・プログラムに SQL および PL/SQL を組み込むことができます。

埋込み SQL を取り込んだ 3GL アプリケーション・プログラムを、C、C++、COBOL または FORTRAN で作成する場合は、該当するプリコンパイラのマニュアルを参照してください。たとえば、C または C++ でプログラミングする場合は、『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

Oracle Developer/2000 は、フォーム作成プログラム、レポート作成ツール、PL/SQL 用のデバッグ環境などを含む数種類のツールを提供するコオペラティブ開発環境です。Developer/2000 を使用する場合は、該当する Oracle のツール製品のマニュアルを参照してください。

SQL の詳細は、『Oracle9*i* SQL リファレンス』および『Oracle9*i* データベース管理者ガイド』を参照してください。Oracle の基本概念については、『Oracle9*i* データベース概要』を参照してください。

XML データを処理するアプリケーションの開発については、『Oracle9*i* XML Developer's Kit ガイド -XDK』および『Oracle9*i* XML API リファレンス - XDK および Oracle XML DB』を参照してください。

このマニュアルの多くの例では、Oracle のインストール時にデフォルトでインストールされるシード・データベースのサンプル・スキーマが使用されます。これらのスキーマの作成方法および使用方法については、『Oracle9*i* サンプル・スキーマ』を参照してください。

リリース・ノート、インストール・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

`http://otn.oracle.co.jp/membership/`

すでに OTN-J のユーザー名およびパスワードを取得済であれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

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

表記規則

この項では、このマニュアルの本文およびコード例で使用される表記規則について説明します。この項の内容は次のとおりです。

本文中の表記規則

本文では、特別な用語をより迅速に識別できるように、様々な表記規則を使用します。次の表に、それらの表記規則を説明し、使用例を示します。

表記規則	意味	例
太字	太字は、本文中で定義されている用語または用語集に記載されている用語（あるいはその両方）を示します。	この句を指定すると、**索引構成表**が作成されます。
固定幅フォントの大文字	固定幅フォントの大文字は、システムが提供する要素を示します。このような要素には、パラメータ、権限、データ型、Recovery Manager キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージおよびメソッドが含まれます。また、システムが提供する列名、データベース・オブジェクト、データベース構造、ユーザー名およびロールも含まれます。	NUMBER 列に対してのみ、この句を指定できます。 BACKUP コマンドを使用して、データベースのバックアップを取ることができます。 USER_TABLES データ・ディクショナリ・ビュー内の TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ディレクトリ名およびユーザーが提供する要素のサンプルを示します。このような要素には、コンピュータ名、データベース名、ネット・サービス名および接続識別子が含まれます。また、ユーザーが提供するデータベース・オブジェクトとデータベース構造、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータ値も含まれます。 **注意：**大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は記載されているとおりに入力してください。	sqlplus と入力し、SQL*Plus を起動します。 パスワードは、orapwd ファイルに指定されています。 /disk1/oracle/dbs ディレクトリ内のデータ・ファイルおよび制御ファイルのバックアップを取ります。 department_id、department_name および location_id 列は、hr.departments 表にあります。 QUERY_REWRITE_ENABLED 初期化パラメータを true に設定します。 oe ユーザーとして接続します。 JRepUtil クラスはこれらのメソッドを実装します。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリック体は、プレースホルダまたは変数を示します。	*parallel_clause* を指定できます。 U*old_release*.SQL を実行します。ここで、*old_release* は、アップグレード前にインストールしたリリースです。

コード例中の表記規則

コード例は、SQL、PL/SQL、SQL*Plus またはその他のコマンドライン文を説明します。これらは、固定幅フォントで表示され、この例に示すとおり、通常のテキストと区別されます。

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

次の表に、コード例で使用される表記規則を説明し、その使用例を示します。

表記規則	意味	例
[ ]	大カッコは、任意に選択する 1 つ以上の項目を囲みます。大カッコは入力しないでください。	DECIMAL (*digits* [, *precision*])
{ }	中カッコは、2 つ以上の項目を囲み、そのうちの 1 つの項目は必須です。中カッコは入力しないでください。	{ENABLE | DISABLE}
|	縦線は、大カッコまたは中カッコ内の 2 つ以上の選択項目を表します。オプションのうちの 1 つを入力します。縦線は入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
. . .	水平の省略記号は、次のいずれかを示します。 ■ 例に直接関連しないコードの一部が省略されている。 ■ コードの一部を繰り返すことができる。	CREATE TABLE ...AS *subquery*; SELECT *col1*, *col2*, ..., *coln* FROM employees;
. . .	垂直の省略記号は、例に直接関連しない複数の行が省略されていることを示します。	
その他の句読点	大カッコ、中カッコ、縦棒線および省略記号以外の句読点は、表示されているとおり入力する必要があります。	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
イタリック体	イタリック体は、特定の値を指定する必要があるプレースホルダまたは変数を示します。	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
大文字	大文字は、システムが提供する要素を示します。これらの用語は、ユーザー定義の用語と区別するために大文字で示されます。用語が大カッコ内にないかぎり、表示されているとおりの順序および綴りで入力します。ただし、これらの用語は大 / 小文字が区別されないため、小文字でも入力できます。	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
小文字	小文字は、ユーザー定義のプログラム要素を示します。たとえば、表名、列名、ファイル名などです。 **注意 :** 大文字と小文字を組み合せて使用するプログラム要素もあります。これらの要素は記載されているとおりに入力してください。	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

アプリケーション開発のための新機能

ここでは、Oracle9*i* で提供される、アプリケーション開発のための新機能について説明します。

- アプリケーション開発のための Oracle9i の新機能

アプリケーション開発のための Oracle9*i* の新機能

リリース 2（9.2）

- **フラッシュバック問合せの拡張機能**

  フラッシュバック問合せは、DBMS_FLASHBACK パッケージではなく、SELECT 文の AS OF 句を使用して実行できます。この方法は融通性に富んでいるため、異なる日付 / 時刻または問合せ内の各表の SCN 設定を使用して、結合、集合演算、副問合せおよびビューを実行できます。INSERT 文または CREATE TABLE AS SELECT 文内のフラッシュバック問合せを使用して、リストアや過去のデータの取得もできます。

  参照： 7-40 ページの「特定時点のデータの問合せ（フラッシュバック問合せ）」を参照してください。

- **INSERT 文および UPDATE 文での PL/SQL レコードの使用**

  PL/SQL レコードを使用して関連データ項目を指定すると、別々に各レコード・フィールドを指定するかわりにレコード全体を使用して、挿入操作または更新操作を実行できます。

  参照： 8-20 ページの「SQL の INSERT 文および UPDATE 文での PL/SQL レコードの使用」を参照してください。

- **Java プログラミングの実行に対する変更**

  EJB または CORBA を使用して開発するには、Oracle9*i* Application Server の一部である J2EE コンポーネントを使用する必要があります。EJB および CORBA は、データベース内ではサポートされていません。ただし、中間層アプリケーション・サーバーからのアクセスと同様に、これらのコンポーネントからもデータベースへのアクセスは可能です。データベース内のオブジェクト型に対して、Java ストアド・プロシージャおよび Java メソッドの書込みもできます。

  参照： 1-8 ページの「Java ストアド・プロシージャ、JDBC および SQLJ の概要」を参照してください。

- **制約名の変更機能**

  新しい制約の作成時にすでに制約が存在していたため、データ・アプリケーション管理に問題が発生した場合は、既存の制約名を変更して競合を回避できます。暗号のようなシステム生成名の制約を調べてわかりやすい名前を付けると、後で簡単に使用可能または使用禁止にできます。

  参照： 4-22 ページの「整合性制約名の変更」を参照してください。

- **NCHAR 型、NVARCHAR2 型および NCLOB 型の拡張サポート**

  グローバリゼーション・サポート型は、SQL および PL/SQL オブジェクト型の属性として使用でき、また、PL/SQL コレクション型（VARRAY、ネストした表など）の中で使用できます。

- **XML プログラミングの新機能**

  新しい組込み型および拡張組込み型（XMLType、XDBUriType など）を使用して、XML の解析、格納および検索をデータベースで実行できます。詳細は、XML に関するドキュメントを参照してください。

- **UTL_FILE パッケージの拡張**

  `UTL_FILE` パッケージには、一般的なファイル操作の実行が可能な、多くの新しいファンクションが含まれています。たとえば、検索、自動フラッシュ、バイナリ・データの読込みおよび書込み、ファイルの削除、ファイル権限の変更などができます。`UTL_FILE_DIR` 初期化パラメータではなく、`CREATE DIRECTORY` 文（すべての小文字名を二重引用符で囲む）を使用して開始する必要があります。

  参照： これらの拡張機能の詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

- **ユーザー定義コンストラクタ**

  独自のファンクションを使用して、オブジェクト型のシステム・デフォルト・コンストラクタをオーバーライドできます。

  参照： 『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

- **トリガー内の LOB データへのアクセス**

  `:NEW` 変数を使用して、`BEFORE` および `INSTEAD OF` トリガー内の LOB データへのアクセスまたは変更ができます。

  参照： 15-15 ページの「例 : トリガーによる LOB 列の変更」を参照してください。

- **型のシノニム**

  様々な型に、シノニムを定義できます。

  参照： 2-25 ページの「シノニムの管理」を参照してください。

- **Pro*C/C++ アプリケーション内のスクロール可能なカーソル**

  スクロール可能カーソルを使用して、Pro*C/C++ アプリケーションの結果セット内を移動できます。

  参照： 1-21 ページの「Pro*C/C++ 機能の特長」を参照してください。

- **Pro*C/C++ 内の接続プーリングのサポート**

  Pro*C/C++ 内の接続プール機能で、Pro*C/C++ アプリケーションのパフォーマンスを最適化できます。

  参照： 1-21 ページの「Pro*C/C++ 機能の特長」を参照してください。

- **オンライン・マニュアルのリンクの向上**

  このマニュアルから他のマニュアルへの相互参照の多くは、より正確になったため、他のマニュアルの目次ではなく特定の位置にリンクします。ただし、オンライン・マニュアルの改善は現在進行中のため、このリリースのすべてのリンクが改善されたわけではありません。

リリース 1（9.0.1）

- **SQL パーサーと PL/SQL パーサーの統合**

  PL/SQL は、INSERT、UPDATE、DELETE など、SQL 文のすべての構文をサポートします。これまで PL/SQL プログラムではエラーが発生していた有効な SQL 文は、正常に動作します。

  参照： エラー・チェック機能の一貫性が向上したため、実行時にエラーが生成されるのではなく、コンパイル時に無効なコードが検出される場合があります。また、逆の場合もあります。移行手順の一部として、ソース・コードの変更が必要な場合もあります。移行手順の詳細は、『Oracle9*i* データベース移行ガイド』を参照してください。

- **再開可能な記憶域の割当て**

  アプリケーションで記憶域の割当てエラーが発生すると、動作が一時停止し、問題の解決、オペレータへの通知などの処置が行われます。この動作は、記憶域が追加または解放されると再開します。

  参照： 7-38 ページの「記憶域エラー状態後の実行の再開」を参照してください。

- **フラッシュバック問合せ**

  ある時点で存在していた表データを問い合せることができます。フラッシュバック問合せを使用すると、アプリケーションは、DBA を介さなくても、コストのかかるリカバリ操作も行わずに、過去のデータに対する問合せ、比較またはリカバリを実行できます。現行の表データは、他のアプリケーションからいつでも使用できます。

  参照： 7-40 ページの「特定時点のデータの問合せ（フラッシュバック問合せ）」を参照してください。

- **WITH 句を使用した複雑な副問合せの再利用**

  複雑な副問合せを繰り返すのではなく、副問合せに名前を付けて、同じ問合せ内でこの名前を複数回参照できます。これはコーディング時に便利な機能です。また、オプティマイザが、最適化できる共通コードを簡単に検索できるようになります。

  参照： 2-7 ページの「ヒント : 同一副問合せの複数回の参照」を参照してください。

- **新しい日時データ型**

  新しい TIMESTAMP データ型では、小数点以下の秒を含む時間値を記録できます。新しい TIMESTAMP WITH TIME ZONE および TIMESTAMP WITH LOCAL TIME ZONE データ型を使用すると、タイムゾーンに合わせて日時値を調整できます。タイムゾーンで夏時間が使用されているかどうかを指定して、時間の進みまたは遅れに対応できます。新しい INTERVAL DAY TO SECOND および INTERVAL YEAR TO MONTH データ型は、2 つの日時の差を表し、日付の算術を簡単にします。

  参照：

  - 3-3 ページの「Oracle 組込みデータ型の概要」を参照してください。
  - 3-10 ページの「日時データの表現」を参照してください。

- **LOB データ型の統合の向上**

  LOB 型を、他の同様の型と同じように操作できます。CLOB 型および NCLOB 型に文字関数を使用できます。また、BLOB 型を RAW として処理できます。LOB 型と他の型との変換もさらに簡単に行えます。特に、LONG 型から LOB 型への変換が簡単になりました。

  参照：

  - 3-21 ページの「ラージ・データ型の表現」を参照してください。
  - 3-31 ページの「Oracle によるデータ型の変換」を参照してください。
  - 3-6 ページの「文字データの表現」を参照してください。

- **グローバリゼーションおよびグローバリゼーション・サポートの改善**

  固定幅または可変幅キャラクタ・セットを使用して、データを Unicode 形式で格納できます。文字列処理および記憶域宣言は、バイト長、またはバイト数が自動的に計算される場合はキャラクタ長を使用して指定できます。データベース全体で文字列の長さに同じセマンティクスを使用するように設定するか、または個別のプロシージャに対して設定できます。この設定は、プロシージャが無効になっても保持されます。

  参照： 3-6 ページの「文字データの表現」を参照してください。

- **バルク操作の改善**

  バルク・フェッチなどのバルク SQL 操作を、ネイティブ動的 SQL（`EXECUTE IMMEDIATE` 文）を使用して実行できます。いくつかの行にエラーがあっても続行するバルク挿入操作またはバルク更新操作を実行し、操作が完了してから問題を調べることができます。

  参照： 9-17 ページの「バルク・バインドの概要」を参照してください。

- **PL/SQL Web アプリケーションのサポートの改善**

  `UTL_HTTP` パッケージおよび `UTL_SMTP` パッケージには、多くの拡張機能（パスワードで保護されている Web ページへのアクセス、添付ファイル付き電子メールの送信など）が含まれています。

  参照： 18-1 ページの第 18 章「PL/SQL を使用した Web アプリケーションの開発」を参照してください。

- **PL/SQL コードのネイティブ・コンパイル**

  一般的な C 言語の開発ツールを使用して、Oracle が提供する、またはユーザーが作成したストアド・プロシージャをネイティブ実行可能ファイルにコンパイルすることでパフォーマンスが向上します。この設定は保存されるため、プロシージャは、後で無効にされた場合でも同じ方法でコンパイルされます。

  参照： 9-20 ページの「ネイティブ実行のための PL/SQL プロシージャのコンパイル」を参照してください。

- **Oracle C++ Call Interface（OCCI）API**

  OCCI API を使用すると、C++ を使用して高速な低レベルのデータベース・アプリケーションを作成できます。これは、既存の OCI API と同様です。

  参照： 1-25 ページの「OCI および OCCI の概要」を参照してください。

- **アプリケーション・ロールの保護**

  Oracle9*i* では、アプリケーション開発者は、アプリケーションにパスワードを埋め込んでロールを保護する必要がありません。開発者は、アプリケーション・ロールを作成して、そのロールを使用可能にする権限がある PL/SQL パッケージを指定できます。PL/SQL パッケージによって使用可能になったアプリケーション・ロールを、保護アプリケーション・ロールといいます。

  参照： 11-12 ページの「保護アプリケーション・ロールの作成」を参照してください。

- **アプリケーション・コンテキストの作成**

  次のようなコマンドを入力して、アプリケーション・コンテキストを作成できます。

  ```
  CREATE CONTEXT Order_entry USING Apps.Oe_ctx;
  ```

  また、Oracle Policy Manager を使用してアプリケーション・コンテキストを作成することもできます。

  参照： 12-16 ページの「手順 2. アプリケーション・コンテキストの作成」を参照してください。

- **専用外部プロシージャ・エージェント**

  異なる Oracle インスタンスまたは別のマシン上で、外部プロシージャ・エージェント（`tnsnames.ora` 内の `EXTPROC` エントリ）を実行できます。これによって、外部プロシージャをより強力に構成し、1 つの外部プロシージャがクラッシュしても、異なるエージェント・プロセス内の他の外部プロシージャが実行を続けるようにできます。

  参照：

  - 10-6 ページの「外部 C プロシージャのロード」を参照してください。
  - 10-8 ページの「外部プロシージャの公開」を参照してください。

第I部

Oracle9*i* アプリケーション開発の概要

第 I 部に含まれる章は、次のとおりです。

1

Oracle プログラム環境の理解

この章では、次のアプリケーション開発システムを簡単に紹介します。

- PL/SQL の概要
- Java ストアド・プロシージャ、JDBC および SQLJ の概要
- Pro*C/C++ の概要
- Pro*COBOL の概要
- OCI および OCCI の概要
- Oracle Objects For OLE（OO4O）の概要
- プログラミング環境の選択

Oracle アプリケーション開発の概要

アプリケーション開発者がデータベースと対話するためのプログラムを作成する場合、次のような多くの選択肢があります。

クライアント / サーバー・モデル

従来のクライアント / サーバー・プログラムでは、アプリケーションのコードはデータベース・サーバー以外のマシン上で実行されます。データベース・コールは、このクライアント・マシンからデータベース・サーバーに送信されます。挿入および更新操作時には、データがクライアントからサーバーに送信され、問合せ操作時には、データがサーバーからクライアントに戻されます。データは、クライアント・マシン上で処理されます。クライアント / サーバー・プログラムは、SQL 文が C、C++、COBOL などの別の言語のコード内に埋め込まれており、通常、プリコンパイラを使用して作成されます。

サーバー側のコーディング

データベースで変更が発生した場合に自動的に実行されるトリガー、または明示的にコールされるストアド・プロシージャを使用して、完全にデータベース内に常駐するアプリケーション・ロジックを開発できます。アプリケーションから作業をオフロードすると、検証およびクリーンアップの実行に使用されるコードの再利用、および様々なクライアントからのデータベース処理の制御ができます。たとえば、Web サーバーを介してストアド・プロシージャをコールできるようにすることによって、クライアント / サーバー・アプリケーションと同じ機能を実行する Web ベースのユーザー・インタフェースを構成できます。

2 層モデルと 3 層モデル

クライアント / サーバー計算は、**2 層モデル**と呼ばれることがあります。このモデルでは、アプリケーションはデータベース・サーバーと直接通信します。**3 層モデル**では、別のサーバー（**アプリケーション・サーバー**）が要求を処理します。アプリケーション・サーバーは、基本的な Web サーバーである場合も、キャッシュやロード・バランスなどの拡張機能を実行する場合もあります。この中間層の処理能力を向上させると、クライアント・システムに必要なリソースを減らすことができるため、**Thin クライアント**構成になります。この構成では、Web ブラウザ、または TCP/IP や HTTP プロトコルを介して要求を送信する他の手段のみがクライアント・マシンに必要です。

ユーザー・インタフェース

アプリケーションがエンド・ユーザーに対して表示するインタフェースは、そのアプリケーションの背後にあるテクノロジおよびユーザー自身のニーズによって異なります。経験豊富なユーザーの場合は、データベースに渡される SQL コマンドを入力します。初心者ユーザーの場合は、クライアント・システム（Windows、X-Window など）のグラフィックス・ライブラリを使用可能な Graphical User Interface（GUI）が表示されます。また、これらの従来のユーザー・インタフェースは、HTML および Java を使用して Web で提供することもできます。

ステートフル・ユーザー・インタフェースとステートレス・ユーザー・インタフェース

従来のクライアント / サーバー・アプリケーションでは、アプリケーションはユーザーのアクションを記録し、この情報を 1 つまたは複数のセッションを通じて使用します。たとえば、前に選択した項目を再入力する必要がないように、その項目をメニュー内に表示できます。アプリケーションがこのような情報を保存できる場合、そのアプリケーションは**ステートフル**であるといいます。

最も簡単に開発できる種類の Web アプリケーションまたは Thin クライアント・アプリケーションは、**ステートレス**のアプリケーションです。これらのアプリケーションは、必要なすべての情報を収集し、データベースを使用してその情報を処理し、ユーザーが変わるとそれらの操作を始めから再実行します。これは、顧客登録などの単一画面要求を処理する一般的な方法です。

デフォルトでステートレスの Web アプリケーションにステートフルの動作を追加するには、多くの方法があります。たとえば、Web ページ上のエントリ・フォームから後続の Web ページに情報を渡して、ウィザードのように、異なる手順を通して項目を記憶可能なインタフェースを構成できます。Cookie を使用して、少ない情報項目をクライアント・マシン上に格納し、ユーザーが Web サイトに再度アクセスしたときに、それらの項目を取得できます。また、サーブレットを使用して、データベース・セッションを開いたままにし、同じクライアントからの要求間で、変数の格納もできます。

PL/SQL の概要

PL/SQL は、標準データベース・アクセス言語である SQL に対して Oracle が提供するプロシージャ拡張です。PL/SQL は、高機能 4GL プログラミング言語として、透過的な SQL アクセス、Oracle サーバーおよびツール製品との緊密な統合、移植性、セキュリティ、さらにデータのカプセル化、オーバーロード、例外処理、情報隠蔽などの最新のソフトウェア・エンジニアリング機能を提供します。

PL/SQL を使用すると、SQL 文でデータを操作したり、IF-THEN や LOOP などのプロシージャ構造でプログラム・フローを制御することができます。さらに、定数および変数の宣言、プロシージャおよびファンクションの定義、コレクションまたはオブジェクト型の使用、ランタイム・エラーのトラップも可能です。

Oracle プログラム・インタフェースのいずれかを使用して作成したアプリケーションであれば、PL/SQL ストアド・プロシージャをコールしたり、PL/SQL コードのブロックをサーバーに送信して実行できます。3GL アプリケーションの場合は、ホスト変数および暗黙的データ型変換を使用して、PL/SQL のスカラー・データ型およびコンポジット・データ型にアクセスできます。

PL/SQL コードはデータベース内で実行されるため、データ処理集中型の操作が非常に効率的に行われ、クライアント / サーバー・アプリケーションでのネットワークの通信量が最小化されます。

PL/SQL は Oracle Developer と緊密に統合されているため、アプリケーションのクライアント・コンポーネントおよびサーバー・コンポーネントを同一言語で開発し、それらのコンポーネントを分割してパフォーマンスおよびスケーラビリティを最適化することができます。さらに、Oracle の Web Forms を使用することで、アプリケーションを複数層のインターネットまたはイントラネット環境で実行できるため、アプリケーションのコードは 1 行も変更する必要はありません。

詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

単純な PL/SQL の例

debit_account プロシージャは、銀行口座からの出金処理を行います。このプロシージャは、口座番号および金額をパラメータとして受け入れます。口座番号を使用して、データベースから口座残高が取得されます。次に、新規残高が計算されます。新規残高が 0（ゼロ）より小さい場合はエラー・ルーチンが実行され、それ以外の場合は銀行口座が更新されます。

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
   old_balance REAL;
   new_balance REAL;
   overdrawn   EXCEPTION;
BEGIN
   SELECT bal INTO old_balance FROM accts
      WHERE acct_no = acct_id;
   new_balance := old_balance - amount;
   IF new_balance < 0 THEN
      RAISE overdrawn;
   ELSE
      UPDATE accts SET bal = new_balance
         WHERE acct_no = acct_id;
   END IF;
   COMMIT;
EXCEPTION
   WHEN overdrawn THEN
      -- handle the error
END debit_account;
```

PL/SQL のメリット

PL/SQL は、完全に移植可能な高性能トランザクション処理言語で、次のようなメリットがあります。

SQL の完全サポート

PL/SQL では、SQL のデータ操作、カーソル制御およびトランザクション制御のすべてのコマンドを使用できる他に、SQL 関数、演算子、疑似列もすべて使用できます。このため、Oracle のデータを柔軟かつ安全に操作できます。PL/SQL は SQL のすべてのデータ型をサ

ポートします。これによって、アプリケーションとデータベース間でやり取りされるデータを変換する必要性が少なくなります。

動的 SQL とは、実行時に SQL 文をその場で作成および処理できるプログラミング手法です。この手法によって、PL/SQL に Perl、Korn シェル、Tcl などのスクリプト言語と同等の柔軟性が得られます。

Oracle との緊密な統合

PL/SQL は SQL のすべてのデータ型をサポートします。SQL が提供する直接アクセスと組み合せることで、これらの共有データ型によって、PL/SQL と Oracle データ・ディクショナリが統合されます。

`%TYPE` および `%ROWTYPE` 属性を使用すると、コードを表定義の変更に対応させることができます。たとえば、`%TYPE` 属性を使用すると、データベースの列の型を基に変数を宣言できます。列の型が変更されると、変数は実行時に正しい型を使用します。これによってデータの独立性が提供され、メンテナンスのコストが減少します。

パフォーマンスの向上

データベース集中型のアプリケーションの場合は、SQL 文を Oracle に送信し実行する前に、PL/SQL ブロックを使用して SQL 文をグループ化できます。これによって、アプリケーションと Oracle 間の通信オーバーヘッドが大幅に軽減されます。

PL/SQL ストアド・プロシージャは、1 回のコンパイルで実行可能形式に格納されるため、プロシージャ・コールを迅速かつ効率的に行うことができます。1 回のコールで計算集中型のストアド・プロシージャを起動できるため、ネットワーク通信量が軽減され、往復の応答時間も短縮されます。実行可能コードは自動的にキャッシュされ、複数のユーザーが共有します。このため、必要なメモリー量および起動オーバーヘッドが減少します。

生産性の向上

PL/SQL によって、Oracle Forms や Oracle Reports などに対してプロシージャ機能を追加できます。たとえば、Oracle Forms のトリガーに 1 つの PL/SQL ブロック全体を使用することができます。複数のトリガー・ステップ、マクロまたはユーザー・イグジットを使用する必要はありません。

さらに、PL/SQL はどの環境でも同等に使用できます。Oracle のあるツール製品で PL/SQL を使用できるようになると、この知識を他のツール製品にも使用できるため、生産性が向上します。たとえば、あるツール製品で作成したスクリプトは、他のツール製品でも使用できます。

スケーラビリティ

PL/SQL ストアド・プロシージャは、アプリケーション処理がサーバー上で集中化されているため、スケーラビリティが増大します。また、ストアド・プロシージャの依存性を自動的にトラッキングできるため、スケーラブルなアプリケーション開発が容易になります。

共有サーバー（以前のマルチスレッド・サーバー（MTS））の共有メモリー機能によって、Oracle は、単一ノード上で何千もの同時ユーザーをサポートできます。さらにスケーラビリティを向上させる場合は、Oracle Net Connection Manager を使用してネットワーク接続を多重化できます。

メンテナンス性の向上

PL/SQL ストアド・プロシージャは、一度妥当性チェックを実施した後は、どのアプリケーションで使用しても問題はありません。PL/SQL ストアド・プロシージャの定義が変更された場合、そのプロシージャに影響するのみで、プロシージャをコールするアプリケーションには影響しません。このため、メンテナンスおよび拡張が簡単になります。また、様々なクライアント・マシン上の多数のコピーをメンテナンスするよりも、サーバー上の 1 つのプロシージャをメンテナンスする方が簡単です。

PL/SQL によるオブジェクト指向プログラミングのサポート

オブジェクト型 オブジェクト型とは、データ構造、およびデータの操作に必要なファンクションおよびプロシージャをカプセル化したユーザー定義のコンポジット・データ型です。データ構造の変数を、属性といいます。オブジェクト型の動作を特徴付けるファンクションおよびプロシージャをメソッドといい、PL/SQL で実装できます。

オブジェクト型は理想的なオブジェクト指向モデリング・ツールであり、これを使用して、複雑なアプリケーションの作成に必要なコストおよび時間を節約できます。オブジェクト型を使用すると、モジュール方式で、メンテナンス性が高く、再利用が可能なソフトウェア・コンポーネントを作成できるのみでなく、様々なプログラマ・チームが同時にソフトウェア・コンポーネントを開発できます。

コレクション コレクションとは、型が同じで順序付けられた要素で構成されるグループです（たとえば、1 クラス内の生徒の成績など）。各要素は、コレクション内でのその要素の位置を決定する一意のサブスクリプトを持っています。PL/SQL は、ネストした表および VARRAY（可変サイズ配列）の 2 種類のコレクションを提供します。

コレクションは、ほとんどの 3GL プログラミング言語のセット、キュー、スタックおよびハッシュ表データ構造と同様に機能し、オブジェクト型のインスタンスを格納でき、また、オブジェクト型の属性にもなります。コレクションをパラメータとして渡すこともできます。このため、コレクションを使用して、データベースの表に（または表から）データ列を移動したり、クライアント側アプリケーションとストアド・サブプログラム間でデータ列を移動することができます。また、PL/SQL パッケージ内にコレクション型を定義することで、多くのアプリケーション間で同じ型を使用できます。

完全な移植性

PL/SQL で作成したアプリケーションは、Oracle が実行されている場合は、どのオペレーティング・システムおよびハードウェア・プラットフォーム上でも実行できます。そのため、移植可能なプログラム・ライブラリを作成して、それを様々な環境で再利用できます。

セキュリティ

PL/SQL ストアド・プロシージャを使用すると、クライアント / サーバー間でアプリケーション・ロジックを分割し、機密性の高い Oracle データをクライアント・アプリケーションからは操作できないようにすることができます。PL/SQL で作成したデータベース・トリガーを使用すると、アプリケーションによる特定の更新を防ぎ、ユーザーからの問合せを監査できます。

また、制限付きの一連の権限が付与されたストアド・プロシージャを介してのみ、ユーザーによる Oracle データの操作を許可することによって、ユーザーの Oracle データへのアクセスを制限できます。たとえば、表を更新するプロシージャへのアクセスは許可し、表自体へのアクセスを禁止することができます。

アプリケーション開発用の組込みパッケージ

- DBMS_PIPE は、セッション間の通信に使用します。
- DBMS_ALERT は、ユーザーへの警告のブロードキャストに使用します。
- DBMS_LOCK および DBMS_TRANSACTION は、ロック・マネージメントおよびトランザクション管理に使用します。
- DBMS_AQ は、アドバンスト・キューイング（AQ）に使用します。
- DBMS_LOB は、ラージ・オブジェクトの操作に使用します。
- DBMS_ROWID は、ROWID の使用に使用します。
- UTL_RAW は、RAW 機能に使用します。
- UTL_REF は、REF での作業に使用します。

サーバー管理用の組込みパッケージ

- DBMS_SESSION は、DBA によるセッション管理に使用します。
- DBMS_SYSTEM は、イベントのデバッグに使用します。
- DBMS_SPACE および DBMS_SHARED_POOL は、メモリー情報を取得し、共有プール・リソースを予約します。
- DBMS_JOB は、サーバー内でのジョブのスケジューリングに使用します。

分散データベース・アクセス用の組込みパッケージ

スナップショット、Advanced Replication、競合解消、遅延トランザクションおよびリモート・プロシージャ・コールへのアクセスを提供します。

Java ストアド・プロシージャ、JDBC および SQLJ の概要

Oracle9*i* には、J2SE 1.3 準拠の OracleJVM が組み込まれています。Oracle では、Java クラスを格納し、それらをストアド・プロシージャおよびトリガーとしてデータベース内で実行できます。これらのクラスは、データは処理できますが、AWT コンポーネントや Swing コンポーネントなどの GUI 要素は表示できません。Java クラスをデータベース内で実行すると、クライアント・マシン上での実行に伴う処理オーバーヘッドおよびネットワーク・オーバーヘッドを発生させずに、これらの Java クラスを何度もコールし、大量のデータを処理できます。

Oracle9*i* には、`java.lang`、`java.io` などのコア JDK ライブラリが含まれています。Oracle9*i* では、JDBC、SQLJ などのクライアント側 Java 標準がサポートされています。また、データーベース内でデータ処理集中型の Java コードを実行可能な、サーバー側の JDBC および SQLJ ドライバが提供されています。

Java および Oracle による Java のサポート状況については、『Oracle9*i* データベース概要』を参照してください。

Java でのプロシージャおよびファンクションの作成の概要

名前付きブロックを作成し、次に、`loadjava` コマンド、SQL の `CREATE FUNCTION`、`CREATE PROCEDURE` または `CREATE PACKAGE` 文を使用してブロックを定義します。これらの Java メソッドは引数を受け入れ、次のものからコールできます。

- SQL の `CALL` 文
- 埋込み SQL の `CALL` 文
- PL/SQL のブロック、サブプログラムおよびパッケージ
- DML 文（`INSERT`、`UPDATE`、`DELETE` および `SELECT`）
- OCI、Pro*C/C++、Oracle Developer などの Oracle 開発ツール
- JDBC、SQLJ 文、CORBA、Enterprise JavaBeans などの Oracle Java インタフェース
- オブジェクト型からのメソッドのコール

Java でのデータベース・トリガーの作成の概要

データベース・トリガーとは、DML 操作で特定の表が変更された場合など、特定のイベントが発生した場合に、Oracle が自動的に開始（起動）するストアド・プロシージャです。トリガーを使用すると、ビジネス・ルールを徹底し、無効な値の格納を防ぐことができます。また、各アプリケーションでチェックおよびクリーンアップ操作を実行する必要性が少なくなります。

ストアド・プロシージャおよびトリガーに Java を使用する理由

- ストアド・プロシージャおよびトリガーは 1 回コンパイルするのみで、簡単に使用でき、メンテナンス性が高く、必要なメモリーおよび計算オーバーヘッドが少なくて済みます。
- ネットワークのボトルネックが回避され、応答時間が短縮されます。分散アプリケーションの作成および使用がより容易になります。
- 計算集中型のプロシージャは、サーバーで実行する方が高速に実行されます。
- 実行者権限ではなく定義者権限によって実行されるストアド・プロシージャおよびトリガーのみをユーザーに対して使用可能にすることで、データ・アクセスを制御できます。
- PL/SQL および Java のストアド・プロシージャが相互にコールできます。
- サーバー内の Java は Java 言語仕様（JLS）に従い、SQLJ 標準を使用できるため、Oracle 以外のデータベースもサポートされます。
- ストアド・プロシージャおよびトリガーは、場所が異なるサイトおよび様々なアプリケーション内で再利用できます。

Oracle JDBC の概要

Java Database Connectivity（JDBC）は、Oracle などのオブジェクト・リレーショナル・データベースに対して Java から SQL 文を送信するための Application Program Interface（API）です。

JDBC 標準では、次の 4 種類の JDBC ドライバが定義されています。

- タイプ 1: JDBC-ODBC ブリッジ。ソフトウェアはクライアント・システム上にインストールする必要があります。
- タイプ 2: ネイティブ・メソッド（C または C++ をコール）および Java メソッド。ソフトウェアはクライアント上にインストールする必要があります。
- タイプ 3: Pure Java。クライアントはソケットを使用してサーバー上のミドルウェアをコールします。
- タイプ 4: 最も Pure Java 度の高いソリューション。Java ソケットを使用してデータベースと直接対話します。

JDBC は X/Open の SQL Call Level Interface に基づき、SQL92 エントリ・レベル標準に準拠しています。

動的 SQL を実行するには JDBC を使用します。動的 SQL とは、実行される埋込み SQL 文が何であるかはアプリケーションが実行されるまでわからないということを意味し、SQL 文の作成に入力を必要とするものです。

Oracle によって実装されたドライバには、Sun 社が定義した JDBC 標準機能に対する拡張が含まれています。Oracle による JDBC ドライバの実装を次に説明します。JDBC 標準の

Oracle データベース・サーバーでのサポートおよび様々なレベルでの拡張機能については、1-11 ページの「JDBC 標準のサポートおよび拡張機能」を参照してください。

JDBC Thin ドライバ

JDBC Thin ドライバはタイプ 4（100% Pure Java）のドライバで、Java ソケットを使用してデータベース・サーバーに直接接続します。このドライバには、独自の Two-Task Common（TTC）実装（Oracle Net の TCP/IP を軽量小型用に実装したもの）が含まれています。このドライバは、完全に Java で作成されているため、プラットフォームには依存しません。

Thin ドライバでは、クライアント側に Oracle ソフトウェアは必要ありません。サーバー側には TCP/IP リスナーが必要です。このドライバは、Web ブラウザをダウンロードする Java アプレット内、または Oracle クライアント・ソフトウェアをインストールしないアプリケーション内で使用します。Thin ドライバは自己完結型ですが、Java ソケットをオープンするため、ソケットをサポートするブラウザでのみ実行できます。

JDBC OCI ドライバ

OCI ドライバはタイプ 2 の JDBC ドライバです。C で作成されている OCI をコールして、Oracle データベース・サーバーと対話します。したがって、ネイティブ・メソッドおよび Java メソッドの両方を使用します。

OCI ドライバは、Thin ドライバより多くの機能（透過的アプリケーション・フェイルオーバー、高度なセキュリティ、高度な LOB 操作など）にアクセスできます。

OCI ドライバは、Oracle7 から Oracle9*i* までの異なるバージョン間の互換性を提供します。また、OCI ドライバは、IPC、Named Pipes および TCP/IP を含むすべてのインストール済 Oracle Net アダプタもサポートします。

OCI ドライバは、ネイティブ・メソッド（Java と C の組合せ）を使用するため、プラットフォーム固有のドライバです。Oracle Net（以前の Net8）、OCI ライブラリ、CORE ライブラリなどのすべての依存ファイルとともに、Oracle8*i* 以上をクライアントにインストールする必要があります。OCI ドライバは、通常、Thin ドライバより実行速度が速くなります。

OCI ドライバは、プラットフォーム固有の C ライブラリを使用しており、Web ブラウザにはダウンロードできないため、Java アプレット用には適していません。Oracle9*i* Application Server などの中間層アプリケーション・サーバー内で実行中の J2EE コンポーネントでは、OCI ドライバを使用できます。Oracle9*i* AS は、アプリケーションとブラウザ間のアクセスをサポートするミドルウェア・サービスおよびツール製品を提供します。

JDBC サーバー側の内部ドライバ

JDBC サーバー側の内部ドライバは、タイプ 2 のドライバで、データベース・サーバー内で実行されます。このため、大量データへのアクセスに必要なラウンドトリップの数が削減されます。このドライバ、Java サーバー VM、データベース、実行速度を最大で 10 倍も上げる Java ネイティブ・コンパイラ、および SQL エンジンは、すべて同じアドレス空間内で実行されます。

このドライバは、データベースで使用されるすべての Java プログラムに対してサーバー側サポート（SQLJ ストアド・プロシージャ、SQLJ ファンクション、SQLJ トリガーおよび Java ストアド・プロシージャ）を提供します。PL/SQL ストアド・プロシージャ、ファンクションおよびトリガーをコールすることもできます。

JDBC サーバー・ドライバは、クライアント側ドライバと同じ機能および拡張機能を完全にサポートします。

JDBC 標準のサポートおよび拡張機能

次に、JDBC 1.22 標準に対する Oracle 拡張機能のいくつかを示します。

- Oracle データ型のサポート
- 行のプリフェッチによるパフォーマンス強化
- バッチ処理の実行によるパフォーマンス強化
- ラウンドトリップを削減する問合せ列型の指定
- DatabaseMetaData コールの制御

Oracle は、JDBC 2.0 標準のすべての API（コア API、オプションのパッケージおよび多数の拡張機能を含む）をサポートしています。その拡張機能のいくつかには、データ ソース、JTA および分散トランザクションが含まれます。

Oracle は、JDBC 3.0 標準の次の機能をサポートしています。

- JDK 1.4 のサポート
- ローカル・トランザクションとグローバル・トランザクション間の切替え
- トランザクションのセーブポイント
- 接続プールによってプリコンパイルされた SQL 文の再利用

JDBC 2.0 のサンプル・プログラム

次の例では、JDBC 2.0 の JNDI を使用してデータ・ソースを検索する場合の典型的な例を示します。

```
// import the JDBC packages
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.pool.*;
...
   InitialContext ictx = new InitialContext();
   DataSource ds = (DataSource)ictx.lookup("jdbc/OracleDS");
   Connection conn = ds.getConnection();
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT ename FROM emp");
   while ( rs.next() ) {
```

```
   out.println( rs.getString("ename") + "<br>");
   }
conn.close();
```

JDBC 2.0 より前のサンプル・プログラム

次のソース・コードは、Oracle JDBC Thin ドライバの登録、データベースへの接続、Statement オブジェクトの作成、問合せの実行および結果セットの処理を行います。

SELECT 文は、EMP 表の ENAME 列を取得してその内容をリストします。

```
import java.sql.*
import java.math.*
import java.io.*
import java.awt.*

class JdbcTest {
  public static void main (String args []) throws SQLException {
    // Load Oracle driver
    DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:thin:@myhost:1521:orcl",
                                   "scott", "tiger");

    // Query the employee names
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("SELECT ENAME FROM EMP");

    // Print the name out
    while (rset.next ())
      System.out.println (rset.getString (1));
    // Close the result set, statement, and the connection
    rset.close();
    stmt.close();
    conn.close();
  }
}
```

Properties オブジェクトを使用する getConnection() メソッドによって、JDBC ドライバに対する Oracle の拡張を実行します。Properties オブジェクトを使用すると、ユーザー、パスワードおよびデータベース情報の他に、行プリフェッチおよびバッチ実行化を指定できます。

このコード内で OCI ドライバを使用する場合は、Connection 文を次のコードで置換します。

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:oci8:@MyHostString",
    "scott", "tiger");
```

ここで MyHostString は、TNSNAMES.ORA ファイル内のエントリです。

アプレットを作成する場合の getConnection() および registerDriver() の文字列の指定は異なります。

SQLJ アプリケーションでの JDBC

JDBC コードおよび SQLJ コード（1-13 ページの「Oracle SQLJ の概要」を参照）は相互運用が可能なため、JDBC の動的 SQL 文を SQLJ の静的および動的 SQL 文とともに使用できます。SQLJ のイテレータ・クラスが、JDBC の結果セットに対応します。JDBC の詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

Oracle SQLJ の概要

SQLJ の特長は次のとおりです。

- Java ソース・コード内に静的 SQL 文を埋め込むための言語仕様で、オラクル社および Java の作成者である Sun 社をはじめとするデータベース企業コンソーシアムによって合意されたものです。この仕様は、ANSI によってソフトウェア標準として受諾されています。
- 標準を基に Oracle によって開発されたソフトウェア・ツールで、Oracle 機能をサポートするための拡張が追加されています。このツールの概要を次に説明します。

SQLJ

Oracle ソフトウェア・ツール SQLJ は、トランスレータおよびランタイムという 2 つの部分で構成されています。どの JVM 上でも JDBC ドライバおよび SQLJ ランタイム・ライブラリを使用して実行します。

SQLJ のソース・ファイルには、静的 SQL 文が埋め込まれた Java ソースが含まれています。SQLJ トランスレータは 100% Pure Java で、標準の JDK 1.1 以上であれば、どの VM にでも移植できます。

通常、Oracle SQLJ の実装は次の 3 つの手順で実行されます。

- SQLJ ソースを、SQLJ ランタイムへのコールを持った Java コードに変換します。SQLJ トランスレータは、ソース・コードを Pure Java のソース・コードに変換し、データベース・スキーマに対して静的 SQL 文の構文およびセマンティクスを確認し、ホスト変数と SQL とで型に互換性があるかどうかを検証します。
- Java コンパイラを使用してコンパイルします。
- ターゲット・データベース用にカスタマイズします。SQLJ で、Oracle 固有にカスタマイズされたプロファイル・ファイルが生成されます（オプション）。

Oracle9*i* は、データ・サーバーと統合された JVM 内で実行される SQLJ のストアド・プロシージャ、ファンクションおよびトリガーをサポートします。SQLJ は、Oracle の

JDeveloper と統合されています。ソース・レベルのデバッグ・サポートは JDeveloper で使用できます。

次に、最も単純で実行可能な SQLJ 文の例を示します。emp 表では empno が一意であるため、この文は値を 1 つ戻します。

```
String name;
#sql  { SELECT ename INTO :name FROM emp WHERE empno=67890 };
System.out.println("Name is " + name + ", employee number = " + empno);
```

各ホスト変数（または、修飾名か複雑な Java ホスト式）の前にはコロン（:）が付きます。その他、宣言 SQLJ 文は（Java の型を宣言し）、これを使用して多数の値を取得する問合せのイテレータ（データベース・カーソルに関連する構造体）を宣言できます。

```
#sql iterator EmpIter (String EmpNam, int EmpNumb);
```

SQLJ のメリット

SQLJ は、Java に対して単純な拡張機能を提供することによって、埋込み SQL を介してデータベース操作を実行するアプリケーションを迅速に開発し、簡単にメンテナンスできるようにします。

Oracle の SQLJ 実装には、特に、次の特長があります。

- 静的 SQL からデータベースにアクセスする簡潔でわかりやすいメカニズムを提供します。アプリケーション内の SQL は、ほとんどの場合静的です。SQLJ は、JDBC よりさらに簡潔で、エラーが発生しにくい静的 SQL 構文を提供します。
- 変換時に静的 SQL を確認します。
- 柔軟な配置構成を提供します。これによって、クライアント側やデータ側または中間層で SQL を実装できるようになります。
- ソフトウェア標準をサポートします。SQLJ はベンダー・グループの努力の集積であり、携わったすべてのベンダーは、今後、SQLJ をサポートしていきます。アプリケーションで複数ベンダーのデータベースにアクセスできます。
- ソース・コード・レベルの移植性を提供します。各ベンダーの DBMS コードがベンダー固有の機能に依存していない場合は、実行可能ファイルをすべてのベンダーの DBMS で使用できます。
- クライアントおよびサーバーで統一されたプログラミング・スタイルを施行します。
- SQLJ トランスレータとグラフィカルな統合開発環境である JDeveloper を統合します。この開発環境では、SQLJ 変換、Java コンパイル、プロファイルのカスタマイズ、ソース・コード・レベルのデバッグがすべて 1 つの手順で提供されます。
- 変換時に構文およびセマンティクスの検証を行う SQL チェッカー・モジュールを提供します。

- Oracle データ型の拡張機能が含まれています。サポートされているデータ型には、LOB、ROWID、REF CURSOR、VARRAY、ネストした表、ユーザー定義のオブジェクト型の他に、RAW や NUMBER などもあります。

SQLJ と JDBC の比較

JDBC は、Java からデータベースへの完全な動的 SQL インタフェースを提供します。経験豊富なプログラマは、JDBC を使用して、データベース処理を完全に制御できます。SQLJ を使用すると、Java データベース・プログラミングを簡略化して、プログラマの生産性を向上できます。

JDBC は、Java からの動的 SQL 実行を詳細に制御します。SQLJ は、特定のデータベース・スキーマ内での SQL 操作に対して、高レベルの静的バインディングを提供します。次に、SQLJ と JDBC の相違点を示します。

- SQLJ のソース・コードは同等の JDBC ソース・コードより簡潔です。
- SQLJ では、データベース接続を使用して静的 SQL コードの型チェックを行います。JDBC は完全に動的な API であるため、型チェックを行いません。
- SQLJ プログラムでは、SQL 文の中に Java バインド式を直接埋め込むことができます。JDBC では、各バインド変数ごとにコールの取得または設定用の文が個別に必要になり、バインドは位置番号によって指定されます。
- SQLJ では、問合せ出力および戻りパラメータの厳密な型指定が提供され、コールに対する型チェックを実行できます。JDBC は、SQL との間でやり取りされる値についてコンパイル時の型チェックを行いません。
- SQLJ は、SQL ストアド・プロシージャおよびファンクションのコールに関して簡潔な規則を提供します。JDBC では、ストアド・プロシージャ（またはファンクション）（*fun*）に対する汎用コールに次の構文を使用する必要があります（SQL92 構文および Oracle エスケープ構文を示します。両方とも使用可能です）。

```
prepStmt.prepareCall("{call fun(?,?)}");       //stored procedure SQL92
prepStmt.prepareCall("{? = call fun(?,?)}");   //stored function SQL92
prepStmt.prepareCall("begin fun(:1,:2);end;"); //stored procedure Oracle
prepStmt.prepareCall("begin :1 := fun(:2,:3);end;");//stored func Oracle
```

SQLJ では、次のように簡潔に表記できます。

```
#sql {call fun(param_list) };  //Stored procedure
// Declare x
...
#sql x = {VALUES(fun(param_list)) };  // Stored function
// where VALUES is the SQL construct
```

SQLJ と JDBC の類似点は次のとおりです。

- SQLJ ソース・ファイルには JDBC コールを含めることができます。SQLJ および JDBC は相互運用が可能です。
- Oracle の JPublisher Tool はカスタム Java クラスを生成します。このクラスは、SQLJ または JDBC のアプリケーションで Oracle オブジェクト型およびコレクション型にマップできます。
- Java と PL/SQL のストアド・プロシージャは完全に互換性があります。

オブジェクト型の SQLJ の例

ユーザー定義オブジェクトおよびオブジェクト参照の単純な例を次に示します。SQLJ の詳細は、『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

次の項目は、次の SQL スクリプトを使用して作成されます。

- PERSON および ADDRESS という 2 つのオブジェクト型
- PERSON オブジェクトを格納するオブジェクト表
- ADDRESS 列および PERSON オブジェクトを参照する 2 つの列を含む EMPLOYEE 表

```
SET ECHO ON;
/
/*** Clean up in preparation ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/
/*** Create address UDT ***/
CREATE TYPE address AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
/*** Create person UDT containing an embedded address UDT ***/
CREATE TYPE person AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    address
)
```

```
/
/*** Create a typed table for person objects ***/
CREATE TABLE persons OF person
/
/*** Create a relational table with two columns that are REFs
     to person objects, as well as a column which is an Address ADT. ***/
CREATE TABLE  employees
(
  empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER
)
/*** Insert some data--2 objects into the persons typed table ***/
INSERT INTO persons VALUES (
            person('Wolfgang Amadeus Mozart', 123456,
               address('Am Berg 100', 'Salzburg', 'AT','10424')))
/
INSERT INTO persons VALUES (
            person('Ludwig van Beethoven', 234567,
               address('Rheinallee', 'Bonn', 'DE', '69234')))
/
/** Put a row in the employees table **/
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
            1001,
            address('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
            50000)
/
/** Set the manager and person REFs for the employee **/
UPDATE employees
   SET manager =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
   SET person_data =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
/
COMMIT
/
QUIT
```

次に、JPublisher を使用して、Oracle の ADDRESS オブジェクトをマップする Address クラスを作成します。ここでは、詳細は省略します。

次の SQLJ のコード例では、Java の Address 型の入力ホスト変数を宣言および設定して、employees 表の列にある ADDRESS オブジェクトを更新します。更新前および更新後に、オフィスの住所が Address 型の出力ホスト変数に入れられ、確認のために出力されます。

```
...
// Updating an object

static void updateObject()
{

   Address addr;
   Address new_addr;
   int empno = 1001;

   try {
      #sql {
         SELECT office_addr
         INTO :addr
         FROM employees
         WHERE empnumber = :empno };
      System.out.println("Current office address of employee 1001:");

      printAddressDetails(addr);

      /* Now update the street of address */

      String street ="100 Oracle Parkway";
      addr.setStreet(street);

      /* Put updated object back into the database */

      try {
         #sql {
            UPDATE employees
            SET office_addr = :addr
            WHERE empnumber = :empno };
         System.out.println
            ("Updated employee 1001 to new address at Oracle Parkway.");

         /* Select new address to verify update */

         try {
            #sql {
               SELECT office_addr
               INTO :new_addr
               FROM employees
               WHERE empnumber = :empno };
```

```
            System.out.println("New office address of employee 1001:");
            printAddressDetails(new_addr);

          } catch (SQLException exn) {
          System.out.println("Verification SELECT failed with "+exn); }

      } catch (SQLException exn) {
      System.out.println("UPDATE failed with "+exn); }

   } catch (SQLException exn) {
   System.out.println("SELECT failed with "+exn); }
}
...
```

Address インスタンスのアクセッサ・メソッド setStreet() の使用方法に注意してください。JPublisher は、JPublisher で生成されるどのカスタム Java クラスについても、そのすべての属性に対してこのようなアクセッサ・メソッドを提供するということを認識しておいてください。

サーバー内の SQLJ ストアド・プロシージャ

SQLJ アプリケーションは、サーバー内に格納して実行することができます。SQL ソースをクライアント上で変換、コンパイルおよびカスタマイズした後、loadjava ユーティリティを使用して、生成済のクラスおよびリソースをサーバーにロードする方法もあります。通常、この場合は Java アーカイブ（.jar）ファイルを使用します。

また、loadjava を使用して SQLJ ソース・コードをサーバーにロードし、サーバーの埋込みトランスレータを使用してこのコードを変換およびコンパイルするという別の方法もあります。

J2EE、OC4J、SOAP、JAAS、サーブレット、JSP、EJB、CORBA および UDDI を使用したプログラミング

これらの業界標準コンポーネントをすべて使用してアプリケーションを開発するには、Oracle9*i* Application Server の Java サポートを使用します。

Oracle9*i* データベースリリース 2（9.2）以上では、Oracle9*i*AS Containers for J2EE（OC4J）（新しく軽量で、使用しやすく、高速な保証済 J2EE コンテナ）が導入されたため、Java 2 Platform, Enterprise Edition（J2EE）およびデータベースの CORBA スタックはサポートされません。ただし、データベース埋込み JVM（Oracle JVM）はそのまま存在し、継続して拡張され、データベースに Java 2 Platform, Standard Edition（J2SE）機能、Java ストアド・プロシージャ、JDBC および SQLJ を提供します。

Oracle9*i* データベース リリース 2（9.2）では、次のデータベースのテクノロジはサポートされていません。

- J2EE スタックの次の機能
  - Enterprise JavaBeans（EJB）コンテナ
  - Oracle JavaServer Pages（OJSP）エンジン
  - Oracle Servlet Engine（OSE）
- Java の Visibroker に基づいて埋め込まれた Common Object Request Broker Architecture（CORBA）のフレームワーク

サーブレット、JSP ページ、EJB および CORBA オブジェクトは、Oracle データベースに配置できません。J2EE スタックおよび CORBA スタックをサポートしていたのは、Oracle9*i* データベースリリース 1（9.0.1）までです。データベースで実行している既存の J2EE アプリケーションを OC4J へ移行することをお薦めします。

Pro*C/C++ の概要

Pro*C/C++ プリコンパイラは、プログラマが C または C++ のソース・ファイル内に SQL 文を埋め込むことができるようにするためのソフトウェア・ツールです。Pro*C/C++ はソース・ファイルを入力として読み込み、C または C++ のソース・ファイルを出力します。この出力ソース・ファイルで埋込み SQL 文が Oracle ランタイム・ライブラリ・コールによって置き換えられ、C または C++ コンパイラによってコンパイルされます。

プリコンパイル中またはその後のコンパイル中にエラーが検出された場合は、プリコンパイラの入力ファイルを変更して、プリコンパイルおよびコンパイルの 2 つの手順を再実行します。

Pro*C/C++ アプリケーションの実装方法

次に、スキーマ SCOTT 内の EMP 表を問い合せる C ソース・ファイル内の単純なコード・フラグメントを示します。

```
...
#define  UNAME_LEN   10
...
int   emp_number;
/* Define a host structure for the output values of a SELECT statement. */
/* No declare section needed if precompiler option MODE=ORACLE          */
struct {
    VARCHAR  emp_name[UNAME_LEN];
    float    salary;
    float    commission;
} emprec;
/* Define an indicator structure to correspond to the host output structure. */
```

```
struct {
    short emp_name_ind;
    short sal_ind;
    short comm_ind;
} emprec_ind;
...
/* Select columns ename, sal, and comm given the user's input for empno. */
    EXEC SQL SELECT ename, sal, comm
        INTO :emprec INDICATOR :emprec_ind
        FROM emp
        WHERE empno = :emp_number;
...
```

埋込み SELECT 文と対話型（SQL*Plus）のバージョンとの違いはごくわずかです。すべての埋込み SQL 文は EXEC SQL で始まります。コロン（:）がすべてのホスト（C）変数の前に付きます。データおよび（データ値が NULL の場合または文字列が切り捨てられたときに設定される）インジケータの戻り値は、（前述のコード・フラグメントにあるような）構造体、配列または構造体配列の中に格納できます。結果セットの値が複数の場合でも、前述の例（社員番号が一意のため結果が 1 つのみ）と同様の方法で簡単に処理されます。埋込み SQL では、列および表の実際の名前を使用します。

プリコンパイラ・オプションのデフォルト値を使用することも、リソースの使用方法、エラーの通知方法、出力形式および（特定の接続または SQL 文に対応する）カーソルの管理方法を制御するための値を入力することもできます。カーソルは、結果セットの値が複数の場合に使用します。

オプションは、構成ファイル内に入力するか、コマンドラインに入力するか、または EXEC ORACLE で始まる特殊な文でソース・コード内に直接入力します。エラーが見つからなかった場合は、次に、C プログラムの場合と同じように出力ソース・ファイルをコンパイルおよびリンクして実行します。

クライアントからのサーバー・データベース・アクセスを様々なプラットフォーム上に配置できるように作成するには、プリコンパイラを使用します。Pro*C/C++ を使用すると、独自のユーザー・インタフェースを自由に設計し、既存のアプリケーションに自由にデータベース・アクセスを追加できます。

埋込み SQL 文を作成する前に、SQL*Plus で対話型 SQL をテストしておきます。それによって、埋込み SQL アプリケーションのテストは少しの変更のみで開始できます。

Pro*C/C++ 機能の特長

Pro*C/C++ の機能のいくつかを次に簡単に説明します。機能全体の詳細は、『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

- アプリケーションは、C または C++ のいずれでも作成できます。

- スレッド・パッケージをサポートしているプラットフォームの場合は、マルチスレッド・プログラムを作成できます。同時接続は、シングル・スレッド・アプリケーションでもマルチスレッド・アプリケーションでもサポートされます。
- PL/SQL ブロックを埋め込むことによって、パフォーマンスを向上できます。PL/SQL ブロックは、ユーザーが独自に作成したか、または Oracle パッケージで提供されているファンクションまたはプロシージャ（Java または PL/SQL で作成されたもの）をコールできます。
- プリコンパイラ・オプションを使用すると、プリコンパイル時のみでなく、実行時にも SQL 文または PL/SQL 文の構文およびセマンティクスを確認できます。
- PL/SQL および Java のストアド・サブプログラムをコールできます。COBOL または C で作成したモジュールは、Pro*C/C++ からコールできます。共有ライブラリ内の外部 C プロシージャは、プログラムからコールできます。
- 様々な環境で実行できるように、条件付きでコード・セクションをプリコンパイルできます。
- パフォーマンスを向上させるには、コード内で配列、構造体または構造体配列をホスト変数およびインジケータ変数として使用できます。
- エラーおよび警告を処理してデータの整合性を保証することができます。エラーをどのように処理するかは、プログラマが制御します。
- プログラムで内部データ型から C 言語データ型（またはその逆）に変換できます。
- 低レベルの C インタフェースおよび C++ インタフェースである OCI および OCCI は、プリコンパイラ・ソース内で使用できます。
- Pro*C/C++ は、動的 SQL（変数の値および文の構文をユーザーが入力できる手法）をサポートします。
- Pro*C/C++ で、特殊な SQL 文を使用してユーザー定義のオブジェクト型を含む表を操作できます。Object Type Translator（OTT）で、データベース内のオブジェクト型および名前付きコレクション型を、ソースに含める構造体およびヘッダーにマップします。
- ネストした表および VARRAY という 2 種類のコレクション型が一連の SQL 文でサポートされています。これによって、データを高度に制御することができます。
- ラージ・オブジェクト（LOB、CLOB、NCLOB および BFILE と呼ばれる外部ファイル）は、別の一連の SQL 文によってアクセスされます。
- 動的 SQL のための新しい ANSI SQL 標準が新しいアプリケーション用にサポートされ、様々なホスト変数を使用して SQL 文を実行できます。動的 SQL のための従来の手法も、既存のアプリケーションで使用できます。
- グローバリゼーション・サポートによって、マルチバイト・キャラクタおよび UCS2 Unicode データを使用できます。

- スクロール可能カーソルを使用して、結果セット内を移動できます。たとえば、結果セットの最終行のフェッチ、または結果セット内の絶対位置または相対位置へのジャンプが可能です。
- 接続プールは、いくつかの名前付き接続で共有されているデータベースへの物理接続のグループです。接続プールのオプションを使用可能にすると、Pro*C/C++ アプリケーションのパフォーマンスを最適化できます。接続プールのオプションは、デフォルトでは使用できません。

Pro*COBOL の概要

Pro*COBOL プリコンパイラは、プログラマが COBOL のソース・コード・ファイル内に SQL 文を埋め込めるようにするためのソフトウェア・ツールです。Pro*COBOL はソース・ファイルを入力として読み込み、COBOL ソース・ファイルを出力します。この出力ソース・ファイルでは埋込み SQL 文が Oracle ランタイム・ライブラリ・コールによって置き換えられ、COBOL コンパイラによってコンパイルされます。

プリコンパイル中またはその後のコンパイル中にエラーが検出された場合は、プリコンパイラの入力ファイルを変更して、プリコンパイルおよびコンパイルの 2 つの手順を再実行します。

Pro*COBOL アプリケーションの実装方法

次に、スキーマ SCOTT 内の EMP 表を問い合せるソース・ファイル内の単純なコード・フラグメントを示します。

```
...
 WORKING-STORAGE SECTION.
*
* DEFINE HOST INPUT AND OUTPUT HOST AND INDICATOR VARIABLES.
* NO DECLARE SECTION NEEDED IF MODE=ORACLE.
*
 01  EMP-REC-VARS.
     05  EMP-NAME    PIC X(10) VARYING.
     05  EMP-NUMBER  PIC S9(4) COMP VALUE ZERO.
     05  SALARY      PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMMISSION  PIC S9(5)V99 COMP-3 VALUE ZERO.
     05  COMM-IND    PIC S9(4) COMP VALUE ZERO.
...
 PROCEDURE DIVISION.
...
     EXEC SQL
         SELECT ENAME, SAL, COMM
         INTO :EMP-NAME, :SALARY, :COMMISSION:COMM-IND
         FROM EMP
         WHERE EMPNO = :EMP_NUMBE
```

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

埋込み SELECT 文と対話型（SQL*Plus）のバージョンとの違いはごくわずかです。すべての埋込み SQL 文は EXEC SQL で始まります。コロン（:）がすべてのホスト（COBOL）変数の前に付きます。SQL 文は END-EXEC によって終了します。データおよび（データ値が NULL の場合または文字列が切り捨てられたときに設定される）インジケータの戻り値は、（前述のコード・フラグメントにあるような）グループ項目、表またはグループ項目の表に格納できます。結果セットの値が複数の場合でも、前述の例（社員番号が一意のため結果が 1 つのみ）と同様の方法で簡単に処理されます。埋込み SQL では、列および表の実際の名前を使用します。

プリコンパイラ・オプションのデフォルト値を使用することも、リソースの使用方法、エラーの通知方法、出力形式および（特定の接続または SQL 文に対応する）カーソルの管理方法を制御するための値を入力することもできます。

オプションは、構成ファイル内に入力するか、コマンドラインに入力するか、または EXEC ORACLE で始まる特殊な文でソース・コード内に直接入力します。エラーが見つからなかった場合は、次に、COBOL プログラムの場合と同じように出力ソース・ファイルをコンパイルおよびリンクして実行します。

クライアントからのサーバー・データベース・アクセスを様々なプラットフォーム上に配置できるように作成するには、プリコンパイラを使用します。Pro*COBOL を使用すると、独自のユーザー・インタフェースを自由に設計し、既存の COBOL アプリケーションに自由にデータベース・アクセスを追加できます。

使用できる埋込み SQL 文は ANSI 規格に準拠しているため、Oracle Net 経由で多数のデータベースのデータにアクセスできます。

埋込み SQL 文を作成する前に、SQL*Plus で対話型 SQL をテストしておきます。それによって、埋込み SQL アプリケーションのテストは少しの変更のみで開始できます。

Pro*COBOL 機能の特長

次に、Pro*COBOL の機能のいくつかを簡単に説明します。機能全体の詳細は、『Pro*COBOL Precompiler プログラマーズ・ガイド』を参照してください。

PL/SQL または Java のストアド・サブプログラムをコールできます。PL/SQL ブロックを埋め込むことによって、パフォーマンスを向上できます。PL/SQL ブロックでは、独自に作成したか、または Oracle パッケージで提供されているファンクションまたはプロシージャをコールできます。

プリコンパイラ・オプションを使用すると、カーソル、エラー、構文検査、ファイル形式などの処理方法を定義できます。

プリコンパイラ・オプションを使用すると、プリコンパイル時のみでなく、実行時にも SQL 文または PL/SQL 文の構文およびセマンティクスを確認できます。

様々な環境で実行できるように、条件付きでコード・セクションをプリコンパイルできます。

パフォーマンスを向上させるには、コード内で表、グループ項目またはグループ項目の表をホスト変数またはインジケータ変数として使用します。

エラーおよび警告の処理方法をプログラミングし、データの整合性を保証できます。

Pro*COBOL は、動的 SQL（変数の値および文の構文をユーザーが入力できる手法）をサポートします。

OCI および OCCI の概要

OCI および OCCI は、API です。これらによって、3GL 固有のプロシージャまたはファンクション・コールを使用して、Oracle データベース・サーバーにアクセスしたり、SQL 文実行のすべての過程を制御するアプリケーションを作成できます。これらの API は次のものを提供します。

- システム・メモリーおよびネットワーク接続の効率的な使用による、パフォーマンスおよびスケーラビリティの向上
- 2 層クライアント / サーバー環境または複数層環境における、動的なセッションおよびトランザクション管理のための一貫したインタフェース
- N 層認証
- Oracle オブジェクトを使用したアプリケーション開発の包括的サポート
- 外部データベースへのアクセス
- 増加するユーザー数および要求数に対応したサービスを、ハードウェアを追加しなくても提供できるアプリケーション開発

OCI を使用すると、Oracle データベース内のデータおよびスキーマを C 言語のようなホスト・プログラミング言語を使用して操作できます。OCCI は、C++ 言語との使用に適したオブジェクト指向インタフェースです。これらの API は、標準的なデータベース・アクセス関数および検索関数のライブラリを、実行時にアプリケーション内にリンクされる動的ランタイム・ライブラリ（OCILIB）の形式で提供します。これによって、SQL または PL/SQL を 3GL プログラムに埋め込む必要がなくなります。

OCI および OCCI コールの詳細は、『Oracle Call Interface プログラマーズ・ガイド』、『Oracle C++ Call Interface プログラマーズ・ガイド』、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』、『Oracle9*i* Database グローバリゼーション・サポート・ガイド』および『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

OCI のメリット

OCI は、Oracle データベースに対する他のアクセス方法に比べて、次のような優れたメリットがあります。

- アプリケーション設計のすべての面に対する細かな制御
- プログラムに対する高度な実行制御
- 使い慣れた 3GL プログラム手法およびアプリケーション開発ツール（ブラウザやデバッガなど）の使用
- 動的 SQL（方法 4）のサポート
- 様々なプラットフォームにおいて、すべての Oracle プログラム・インタフェースが使用可能
- コールバックを使用した動的バインドおよび定義
- サーバー・メタデータ層を表す記述機能
- 登録されたクライアント・アプリケーションに対する非同期のイベント通知
- INSERT、UPDATE および DELETE における配列のデータ操作言語（DML）の機能拡張
- コミット要求を実行に対応付けてラウンドトリップを減らす機能
- 透過的プリフェッチ・バッファを使用して問合せを最適化しラウンドトリップを減らす機能
- OCI ハンドルに対する相互排他ロック（mutex）が不要になるスレッド・セーフティ
- 非ブロック化モードでのサーバー接続コールが実行中か、または完了できない場合、制御が OCI コードに戻ります。

OCI の構成要素

OCI には、次の 4 つの基本機能があります。

- OCI リレーショナル機能データベース・アクセスを管理し SQL 文を処理します。
- OCI ナビゲーション機能 Oracle データベース・サーバーから取得したオブジェクトを操作します。
- OCI データ型マッピングおよび操作機能 Oracle データ型のデータ属性を操作します。
- OCI 外部プロシージャ機能 PL/SQL からの C コールバックを作成します。

手続き型要素および非手続き型要素

OCI を使用すると、SQL が提供する強力な非手続き型データ・アクセス機能と、C や C++ など数多くのプログラミング言語で提供される手続き型機能を組み合せたアプリケーションを開発できます。

- 非手続き型言語プログラムでは、操作対象のデータ群は指定されますが、実行される操作およびその操作の実行方法は指定されません。SQL は非手続き型であるため、データベース・トランザクションの実行用として簡単に覚えやすく使用しやすい言語です。また、SQL は、最新のリレーショナル・データベース・システムおよびオブジェクト・リレーショナル・データベース・システムにおけるデータのアクセスおよび操作に使用される標準言語でもあります。
- 手続き型言語プログラムでは、文の実行の多くが、その前後の文や、ループまたは条件ブランチなどの制御構造に依存します。これらは SQL では使用できません。このような言語は、手続き型であるために SQL より複雑になりますが、非常に柔軟で強力でもあります。

OCI プログラムで非手続き型言語の要素と手続き型言語の要素を組み合せることができるため、構造化プログラミング環境の中で Oracle データベースに簡単にアクセスすることができます。

OCI では、Oracle データベースで使用可能な SQL のデータ定義、データ操作、問合せおよびトランザクション制御のすべての機能をサポートします。たとえば、OCI プログラムは Oracle データベースに対する問合せを実行できます。この問合せで、次のように、入力（バインド）変数を使用してデータベースにデータを提供するようにプログラムに要求できます。

```
SELECT name FROM employees WHERE empno = :empnumber
```

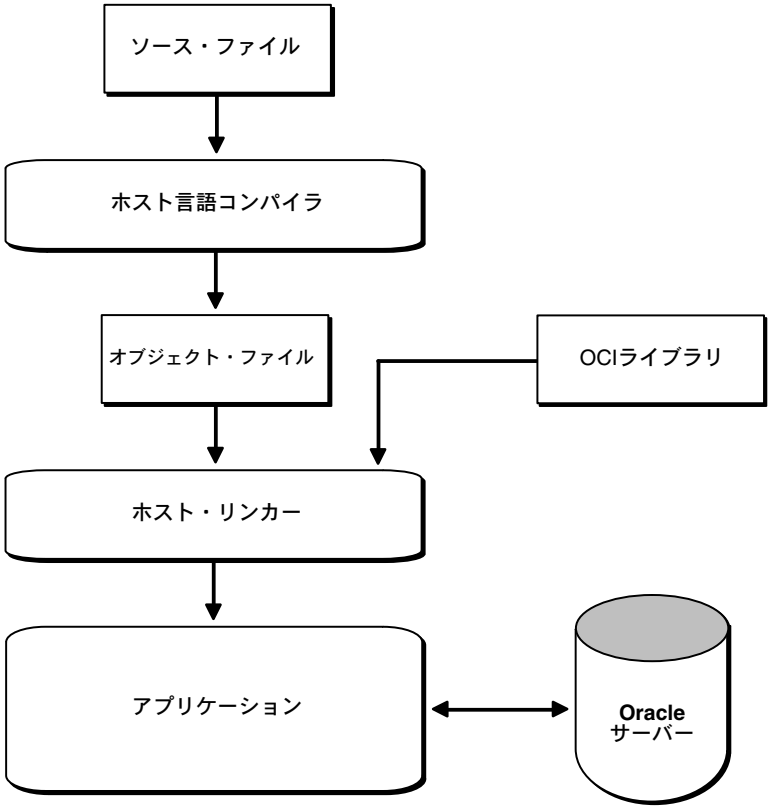
この SQL 文では、`:empnumber` がアプリケーションによって提供される値のプレースホルダです。

また、SQL に対する Oracle の手続き型拡張言語である PL/SQL を使用することもできます。PL/SQL で開発するアプリケーションは、SQL のみで作成したアプリケーションより強力で柔軟です。OCI は、Oracle データベース・サーバーにあるオブジェクトにアクセスし操作する機能も提供しています。

OCI アプリケーションの作成

図 1-1 に示すとおり、OCI プログラムは非データベース・アプリケーションと同じ方法でコンパイルおよびリンクします。特別な事前処理またはプリコンパイルは必要ありません。

図 1-1 OCI の開発過程



注意： プラットフォームによっては、OCI プログラムを正しくリンクするために、OCI ライブラリ以外のライブラリを含める必要があります。必要な追加ライブラリの詳細は、システム固有の Oracle マニュアルを参照してください。

Oracle Objects For OLE（OO4O）の概要

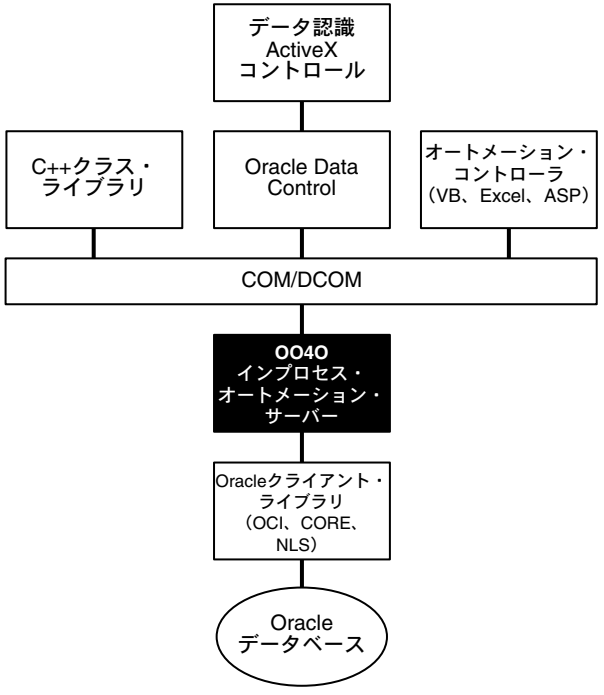
OO4O は、Microsoft COM Automation および ActiveX テクノロジをサポートするプログラミング言語またはスクリプト言語であれば、どの言語からでも Oracle データベース内に格納されているデータに簡単にアクセスできるように設計された製品です。このような言語には、Visual Basic、Visual C++、Visual Basic For Applications（VBA）、IIS Active Server Pages（VBScript および JavaScript）などがあります。

OO4O は、次のソフトウェア・レイヤーで構成されています。

- OO4O インプロセス・オートメーション・サーバー
- Oracle Data Control
- OO4O C++ クラス・ライブラリ

図 1-2「ソフトウェア・レイヤー」に、OO4O のソフトウェア・コンポーネントを示します。

図 1-2 ソフトウェア・レイヤー



注意： OO4O の使用方法の詳細は、OO4O のオンライン・ヘルプを参照してください。

OO4O オートメーション・サーバー

OO4O オートメーション・サーバーは、Oracle データベース・サーバーに接続し、SQL 文および PL/SQL ブロックを実行してその結果にアクセスするための一連の COM オートメーション・オブジェクトです。

Microsoft ADO などの COM ベースのデータベース接続 API とは異なり、OO4O オートメーション・サーバーは、特に Oracle データベース・サーバーを対象に開発されたものです。

これは、Oracle 固有の機能にアクセスするために最適化された API を提供します。この API を使用せずに ODBC または OLE データベース固有のコンポーネントからアクセスすると、煩雑で非効率です。

OO4O は、Visual Basic または Excel で開発される典型的な 2 層クライアント / サーバー・アプリケーションから、Microsoft Internet Information Server（IIS）の Web サーバー・アプリケーションや Microsoft Transaction Server（MTS）などの複数層アプリケーション・サーバー環境で実行されるアプリケーション・サーバーまで、広範囲な環境にわたり、Oracle データベースに効率的かつ簡単にアクセスするための重要な機能を提供します。

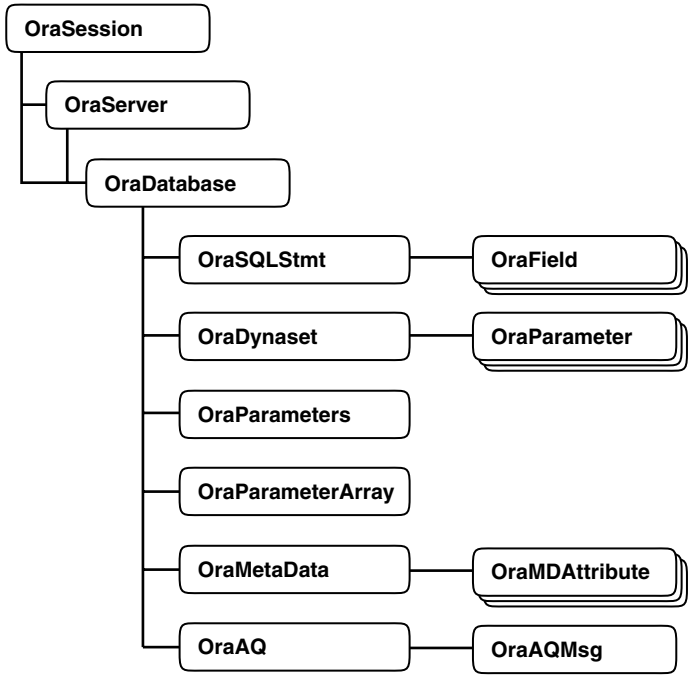
次のような機能があります。

- PL/SQL および Java のストアド・プロシージャ、および無名 PL/SQL ブロックの実行サポートこれには、PL/SQL カーソルなど、ストアド・プロシージャのパラメータとして使用される Oracle データ型のサポートが含まれます。1-35 ページの「Oracle LOB およびオブジェクト・データ型のサポート」を参照してください。
- 問合せの結果セットに簡単かつ効率的にアクセスし、スクロール可能で更新可能なカーソルのサポート
- 効率的な Web サーバー・アプリケーション開発のための、スレッド・セーフなオブジェクトおよび接続プール管理機能
- Oracle オブジェクト・リレーショナル・データ型および LOB データ型の完全サポート
- AQ の完全サポート
- 配列の挿入および更新のサポート
- Microsoft Transaction Server（MTS）のサポート

OO4O オブジェクト・モデル

図 1-3「オブジェクトおよびそのリレーション」に、OO4O オブジェクト・モデルを示します。

図 1-3 オブジェクトおよびそのリレーション



OraSession

OraSession オブジェクトは、アプリケーション内で使用される OraDatabase、OraConnection および OraDynaset オブジェクトのコレクションを管理します。

通常は、1 つのアプリケーションに対して OraSession オブジェクトが 1 つ作成されますが、アプリケーション内とアプリケーション間で共有する名前付き OraSession オブジェクトを作成できます。

OraSession オブジェクトは、アプリケーションの最上位レベル・オブジェクトです。OO4O のメソッドではなく、CreateObject VB/VBA API によって作成される唯一のオブジェクトです。次に、OraSession オブジェクトの作成方法を示すコード・フラグメントを示します。

```
Dim OraSession as Object
Set OraSession = CreateObject("OracleInProcServer.XOraSession")
```

OraServer

OraServer は、Oracle データベース・サーバーへの物理的なネットワーク接続を表します。

OraServer インタフェースは、OCI で提供される接続多重化機能を公開して使用可能にするために導入されています。OraServer オブジェクトの作成後、OpenDatabase メソッドを起動することでこのオブジェクトに複数のユーザー・セッション（OraDatabase）を連結できます。この機能は、IIS などのように、N 層分散環境で OO4O を使用するアプリケーション・コンポーネント用として特に便利です。

多数のアクティブ・ユーザー・セッションを持った Oracle サーバーにアクセスする際に接続多重化機能を使用することによって、サーバーの処理要件およびリソース要件が削減され、サーバーのスケーラビリティも向上します。

OraDatabase

OraDatabase インタフェースは、トランザクション制御用メソッド、および Oracle オブジェクト型を表すインタフェース作成用のメソッドを追加します。スキーマ・オブジェクトの属性は、OraDatabase インタフェースの Describe メソッドを使用して取得できます。

Oracle8 以下では、OraSession インタフェースの OpenDatabase メソッドを起動して OraDatabase オブジェクトを作成しました。Oracle Net の別名、ユーザー名およびパスワードが、引数としてこのメソッドに渡されます。Oracle8*i* 以上では、このメソッドを起動すると OraServer オブジェクトが暗黙的に作成されます。

OraServer インタフェースの説明にあるとおり、OraDatabase オブジェクトは OraServer インタフェースの OpenDatabase メソッドを使用して作成できます。

トランザクション制御メソッドは、OraDatabase（ユーザー・セッション）レベルで使用できます。トランザクションは、読込み / 書込み（デフォルト）、シリアル化可能、または読込み専用として開始できます。このメソッドには次のものが含まれます。

- BeginTrans
- CommitTrans
- RollbackTrans

次に例を示します。

```
UserSession.BeginTrans(OO4O_TXN_READ_WRITE)
UserSession.ExecuteSQL("delete emp where empno = 1234")
UserSession.CommitTrans
```

OraDynaset

OraDynaset オブジェクトによって、SQL の SELECT 文で作成されたデータを参照および更新できます。

OraDynaset オブジェクトは 1 つのカーソルと考えることができますが、実際に OraDynaset のセマンティクスを実装するには実カーソルをいくつか使用する場合があります。

OraDynaset は、サーバーからフェッチされたデータのローカル・キャッシュを自動的に保持し、ブラウズ・データ内のスクロール可能カーソルを透過的に実装します。大規模な問合せの場合は、かなりのローカル・ディスク領域が必要になる場合があります。アプリケーションを実装する場合は、ディスクの使用が少なくなるように、問合せをさらに微調整することをお薦めします。

OraField

OraField オブジェクトは、ダイナセットの行内の単一の列またはデータ項目を表します。

現在行が更新中の場合は、OraField オブジェクトは現在更新中の値を表します。ただし、この値はまだデータベースにコミットされていない可能性があります。

フィールドの値プロパティへの割当ては、（Edit を使用して）レコードを編集中の場合、または（AddNew を使用して）新しいレコードを追加中の場合のみ許可されます。他の方法でフィールドの値プロパティにデータを割り当てると、エラーが発生します。

OraMetaData

OraMetaData オブジェクトは、データベース内の特定のスキーマ・オブジェクトに関する記述情報を表す OraMDAttribute オブジェクトのコレクションです。

OraMetaData オブジェクトは、次の 3 つの列を持つ表として表すことができます。

- メタデータの属性名
- メタデータの属性値
- 値が別の OraMetaData オブジェクトかどうかを示すフラグ

OraMetaData オブジェクトに含まれている OraMDAttribute オブジェクトは、序数を使用した添字またはプロパティの名前を使用してアクセスできます。コレクションにない添字（0 からカウント -1 まで）を参照すると、NULL の OraMDAttribute オブジェクトが戻されます。

OraParameter

OraParameter オブジェクトは、SQL 文または PL/SQL ブロック内のバインド変数を表します。

OraParameter オブジェクトは、OraDatabase オブジェクトの OraParameters コレクションを介して間接的に作成、アクセスおよび削除されます。各パラメータには、それぞれ識別名および関連値があります。SQL 文または PL/SQL 文の中でパラメータ名をプレースホルダとして使用すると、（オブジェクトの記述に示されているとおり）他のオブジェクトの SQL 文および PL/SQL 文にパラメータを自動的にバインドできます。このようにパラメータを使用することで、動的な問合せを容易にし、プログラムのパフォーマンスを向上させます。

OraParamArray

OraParamArray オブジェクトは、OraParameter オブジェクトによって表されるスカラー型バインド変数に対して、SQL 文または PL/SQL ブロック内の配列型バインド変数を表します。

OraParamArray オブジェクトは、OraDatabase オブジェクトの OraParameters コレクションを介して間接的に作成、アクセスおよび削除されます。各パラメータには、それぞれ識別名および関連値があります。

OraSQLStmt

OraSQLStmt オブジェクトは、単一の SQL 文を表します。OraDatabase から OraSQLStmt オブジェクトを作成するには、CreateSQL メソッドを使用します。

OraSQLStmt オブジェクトは、作成中およびリフレッシュ中に、パラメータ名を SQL 文でプレースホルダとして使用して、使用可能なすべての関連入力パラメータを指定された SQL 文に自動的にバインドします。これによって、SQL 文の再解析をしなくても SQL 文の実行パフォーマンスが向上します。

SQLStmt

SALARY プレースホルダに別の値を使用して、後で同一の問合せを実行するために、SQLStmt オブジェクト（updateStmt）を使用できます。次に例を示します。

```
OraDatabase.Parameters("SALARY").value = 200000
updateStmt.Parameters("ENAME").value = "KING"
updateStmt.Refresh
```

OraAQ

OraAQ オブジェクトは、OraDatabase インタフェースの CreateAQ メソッドを起動してインスタンス化します。このオブジェクトは、データベース内にあるキューを表します。

OO4O は、Oracle の AQ 機能にアクセスするためのインタフェースを提供します。これによって、Visual Basic などの一般的な COM ベースの開発環境から AQ 機能にアクセスできます。Oracle AQ の詳細は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

OraAQMsg

OraAQMsg オブジェクトは、キューまたはデキューされるメッセージをカプセル化します。メッセージは、ユーザー定義型または RAW 型です。

Oracle AQ の詳細は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

OraAQAgent

OraAQAgent オブジェクトはメッセージの受信者を表し、複数コンシューマに対応したキューに対してのみ有効です。

OraAQAgent オブジェクトは、AQAgent メソッドを起動してインスタンス化できます。次に例を示します。

```
Set agent = qMsg.AQAgent(name)
```

OraAQAgent オブジェクトは、AddRecipient メソッドを起動してインスタンス化することもできます。次に例を示します。

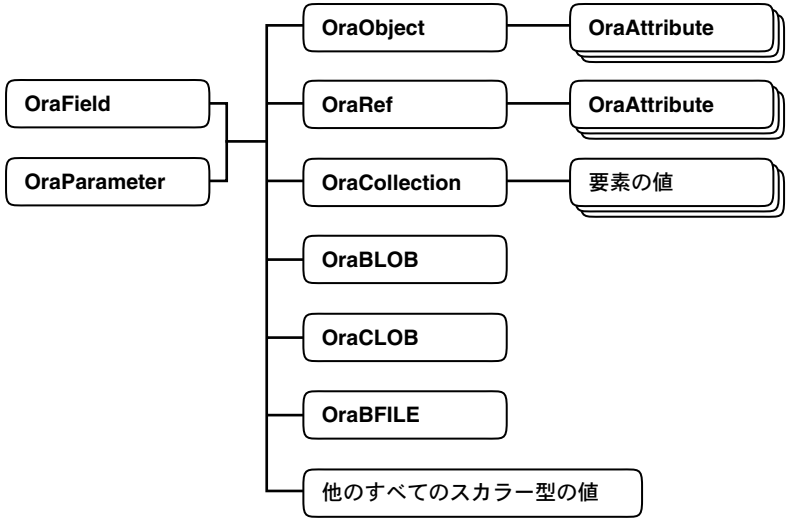
```
Set agent = qMsg.AddRecipient(name, address, protocol).
```

Oracle LOB およびオブジェクト・データ型のサポート

OO4O は、Oracle データベース・サーバーでのオブジェクト・データ型および LOB のインスタンスの、アクセスおよび操作を完全にサポートします。図 1-4「サポートされている Oracle データ型」に、OO4O がサポートするデータ型を示します。

このようなデータ型のインスタンスは、データベースからフェッチするか、あるいは入力変数または出力変数として、ストアド・プロシージャおよびファンクションを含む SQL 文および PL/SQL ブロックに渡すことができます。すべてのインスタンスは、属性の動的アクセスおよび操作用のメソッドを提供する COM オートメーション・インタフェースにマップされます。このインタフェースは、次のものから取得できます。

図 1-4 サポートされている Oracle データ型



OraBLOB および OraCLOB

OO4O の OraBlob および OraClob インタフェースは、データベース内で BLOB、CLOB および NCLOB データ型を持つラージ・オブジェクトを操作するメソッドを提供します。ここでは、BLOB、CLOB および NCLOB データ型を LOB データ型といいます。

LOB データは、Read メソッドおよび CopyToFile メソッドを使用してアクセスします。

LOB データは、Write、Append、Erase、Trim、Copy、CopyFromFile および CopyFromBFile メソッドを使用して変更します。行の中の LOB 列の内容を変更する前に、行のロックを取得する必要があります。LOB 列が OraDynaset のフィールドの場合は、ロックは Edit メソッドを起動して取得します。

OraBFILE

OO4O の OraBFile インタフェースは、データベース内の BFILE データ型のラージ・オブジェクトに対して操作を実行するメソッドを提供します。

BFILES は、データベース表領域外のオペレーティング・システム・ファイル（外部ファイル）内に格納される大規模なバイナリ・データ・オブジェクトです。

Oracle Data Control

Oracle Data Control（ODC）は、Visual Basic およびカスタム・コントロールをサポートするその他の開発ツールにおける、ビジュアル・コントロール（編集、テキスト、リストおよびグリッド）と Oracle データベースとの間のデータ交換を簡素化するために設計された ActiveX コントロールです。

ODC は、Oracle データベースと、それにバインドされたグリッド・コントロールなどのビジュアルなデータ認識コントロールからの情報の流れを処理するエージェントとして機能します。ODC は、データの表示および編集などの様々なユーザー・インタフェース（UI）・タスクを管理します。また、データベースに対する問合せの実行および結果の管理も行います。

ODC は、Visual Basic に含まれている Microsoft データ・コントロールに相当します。isual Basic のデータ・コントロールを使い慣れている場合は、ODC の使用方法をすぐに理解できます。データ認識コントロールと ODC との間の通信は、Microsoft の指定したプロトコルによって制御されます。

OO4O C++ クラス・ライブラリ

OO4O C++ クラス・ライブラリは、Oracle Object Server に対するプログラム・アクセスを提供する C++ クラスのコレクションです。このクラス・ライブラリは OLE オートメーションを使用して実装されていますが、クラス・ライブラリの使用には OLE 開発キットまたは OLE 開発知識は必要ありません。このライブラリを使用すると、C++ 開発者は、OO4O インタフェースにアクセスする COM クライアント・コードを作成する必要がありません。

その他の情報源

OO4O の詳細は、OO4O 製品とともに提供されている次のオンライン・ヘルプを参照してください。

- OO4O のヘルプ
- OO4O C++ クラス・ライブラリのヘルプ

OO4O の使用例は、Oracle インストールの ORACLE_HOME/OO4O ディレクトリにあるサンプルを参照してください。OO4O の例は、次の Oracle マニュアルに記載されています。

- 『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』
- 『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』
- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』

プログラミング環境の選択

新しい開発プロジェクト用のプログラミング環境を選択します。

- 各環境については、前述の概要およびマニュアルを参照してください。
- プラットフォーム固有のマニュアルを参照してください。各プラットフォームでどのコンパイラの使用が承認されているかが説明されています。
- 特定の言語で必要な機能が提供されない場合、この章で記述したどの言語で作成されたコードからも PL/SQL および Java のストアド・プロシージャをコールできることを認識しておいてください。ストアド・プロシージャには、トリガーおよびオブジェクト型メソッドが含まれます。
- C 言語で作成された外部プロシージャは、OCI、Java、PL/SQL または SQL からコールできます。外部プロシージャ自体は、SQL、OCI または Pro*C（C++ ではなく）のいずれかを使用してデータベースにコールバックできます。

簡単な選択基準の例を次に示します。

- Pro*COBOL は、オブジェクト型またはコレクション型をサポートしませんが、Pro*C/C++ ではサポートされています。
- SQLJ が動的 SQL をサポートする方法は、JDBC とは異なります。

OCI またはプリコンパイラの使用の選択

通常、プリコンパイラ・アプリケーションには同等の OCI アプリケーションより少ないコードが含まれているため、生産性が向上します。

データベースを詳細に制御する必要がある場合は、OCI アプリケーション（純 OCI アプリケーションまたは OCI コールが埋め込まれたプリコンパイラ・アプリケーション）の方が適しています。

- OCI では、セッションの多重化および移行をより詳細に制御できます。
- OCI では、リストを含む任意の構造体に使用できるコールバックを使用して、動的バインドおよび定義が提供されます。
- OCI にはメタデータを扱う多くのコールがあります。
- OCI では、クライアント・アプリケーションに対して非同期のイベントを通知できます。クライアントに対し、他のクライアントに伝播するための通知を生成する手段が提供されます。
- OCI では、DML 文は配列を使用してできるだけ多くの反復を完了し、その後、一連のエラーを戻します。
- 特殊な目的のための OCI コールには、アドバンスト・キューイング、グローバリゼーション・サポート、データ・カートリッジおよび日時データ型のサポートが含まれます。
- OCI コールは、Pro*C/C++ アプリケーションに埋め込むことができます。

組込みパッケージおよびライブラリの使用

Java および PL/SQL には、組込みパッケージおよびライブラリがあります。

PL/SQL および Java は、サーバー内で相互運用します。Java から PL/SQL パッケージを実行するか、または Java ラッパーで PL/SQL クラスをラップして、分散 CORBA および EJB クライアントからコールできるようにできます。次の表に、PL/SQL パッケージ、およびそれぞれの相当する Java での操作を示します。

表 1-1 PL/SQL および Java の等価性

PL/SQL パッケージ	相当する Java での操作
DBMS_ALERT	SQLJ または JDBC でパッケージをコールします。
DBMS_DDL	JDBC を使用します。
DBMS_JOB	Java ストアド・プロシージャを持つジョブをスケジューリングします。
DBMS_LOCK	SQLJ または JDBC でコールします。
DBMS_MAIL	JavaMail を使用します。
DBMS_OUTPUT	サブクラス `oracle.aurora.rdbms.OracleDBMSOutputStream` または Java ストアド・プロシージャ `DBMS_JAVA.SET_STREAMS` を使用します。
DBMS_PIPE	SQLJ または JDBC でコールします。
DBMS_SESSION	JDBC を使用して `ALTER SESSION` 文を実行します。

表 1-1 PL/SQL および Java の等価性

PL/SQL パッケージ	相当する Java での操作
DBMS_SNAPSHOT	SQLJ または JDBC でコールします。
DBMS_SQL	JDBC を使用します。
DBMS_TRANSACTION	JDBC を使用して ALTER SESSION 文を実行します。
DBMS_UTILITY	SQLJ または JDBC でコールします。
UTL_FILE	JAVAUSERPRIV 権限を付与して Java I/O エントリ・ポイントを使用します。

Java および PL/SQL

Java および PL/SQL は、ともにデータベース内でアプリケーションを作成するために使用でき、将来的なパフォーマンスの向上が見込まれます。使用に対するガイドラインを次に示します。

データベース・アクセス用に最適化された PL/SQL

PL/SQL は、SQL と同じデータ型を使用します。したがって、特に大量のデータが扱われる場合、データベース・アクセスに処理が集中する場合、またはバルク操作が使用された場合に、より簡単に SQL のデータ型を使用できます。また、SQL を使用すると、Java より速く操作できます。

データベースと統合された PL/SQL

PL/SQL は SQL の拡張であり、同じデータ型を使用します。PL/SQL にはデータ・カプセル化、情報隠蔽、オーバーロードおよび例外処理の機能があります。

Oracle9*i* では、一部の高度な PL/SQL 機能が Java で使用できない場合があります。たとえば、自律型トランザクションやリモート・データベース用の dblink 機能などです。PL/SQL でのコード開発は、通常、Java を使用した場合より速く行うことができます。

Java および PL/SQL が持つオブジェクト指向機能

Java には、分散システムを開発するための継承、ポリモフィズムおよびコンポーネント・モデルがあります。PL/SQL には、継承および**型進化**（型のメソッドおよび属性を、その型を使用するサブタイプおよび表データを保持しながら変更する機能）があります。

オープン分散アプリケーションに使用される Java

Java は、PL/SQL より豊富な型体系を持つ、オブジェクト指向言語です。Java は CORBA（クライアントに様々なコンピュータ言語を持つことができます）および EJB を使用できます。ただし、PL/SQL パッケージも CORBA または EJB クライアントからコールできます。

Java によって、XML ツール、Internet File System または JavaMail を実行できます。また、多くの Java ベース開発ツールが存在します。

第 II 部

データベースの設計

第 II 部に含まれる章は、次のとおりです。

2

スキーマ・オブジェクトの管理

この章では、ユーザーのスキーマ内に異なるタイプのオブジェクトを作成し、それらを管理するために必要な手順について説明します。この章の内容は次のとおりです。

参照：

- 索引およびクラスタについては、第 5 章「索引計画の選択」を参照してください。
- プロシージャ、ファンクションまたはパッケージについては、第 9 章「プロシージャおよびパッケージの使用」を参照してください。
- オブジェクト型については、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。
- 依存性情報については、第 9 章「プロシージャおよびパッケージの使用」を参照してください。
- 対称型レプリケーションを使用する場合、スナップショットなどのスキーマ・オブジェクトの管理の詳細は、『Oracle9*i* アドバンスト・レプリケーション』を参照してください。

表の管理

表は、リレーショナル・データベースにおいてデータを保持するデータ構造です。表は行および列で構成されます。

1 つの表で、システム内で追跡されるエンティティを 1 つ表します。このような表の例として、従業員リストまたは製品の受注リストがあります。

表によって 2 つのエンティティ間の関連を表現することもできます。従業員とその職種の関連、および注文と製品の関連がその例です。このような表では、外部キーを使用して関連を表現します。

適切に設計すると、表はエンティティを表現するとともに、あるエンティティと他のエンティティ間の関連を記述することもできます。ただし、多くの表はエンティティまたは関連の一方のみを表現します。たとえば、EMP_TAB 表は、ある会社の従業員について記述します。この表には DEPTNO という外部キー列も含まれていて、これによって従業員と部門との関連が表現されます。

次の項では、表を作成、変更および削除する方法について説明します。データベース内の表を管理するときの参考になる簡単なガイドラインも示します。

参照： 詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。また、リレーショナル・データベースまたは表の設計に関するマニュアルも参照してください。

表の設計

表を設計するときには、次のガイドラインを参考にしてください。

- 表、列、索引およびクラスタに対して、内容の連想が簡単な名前を使用します。
- 表名および列名に対して省略形、および単数形や複数形を使用する場合には一貫性のあるものにします。

- COMMENT コマンドを使用して、各表およびその列の意味を文書化します。
- 各表を正規化します。
- 各列に適したデータ型を選択します。
- 最後に NULL を許可する列を定義して記憶領域を節約します。
- 必要に応じて表をクラスタ化して記憶領域を節約し、SQL 文のパフォーマンスを最適化します。

また、表を作成する前に、整合性制約を使用するかどうかを判断してください。表の列に整合性制約を定義して、データベースのビジネス・ルールを自動的に適用できます。

参照：

- 表のガイドライン、およびスキーマ・オブジェクトに使用される領域の管理の一般的なガイドラインについては、『Oracle9*i* データベース管理者ガイド』を参照してください。
- データ型については、第 3 章「データ型の選択」を参照してください。
- 整合性制約のガイドラインについては、第 4 章「制約によるデータ整合性のメンテナンス」を参照してください。

表の作成

表を作成するには、SQL コマンド CREATE TABLE を使用します。たとえば、次の文を発行すると、クラスタ化されていない Emp_tab 表が作成されます。Emp_tab 表は、物理的には USERS 表領域内に格納されます。表のいくつかの列に整合性制約が定義されていることに注意してください。

```
CREATE TABLE Emp_tab (
   Empno      NUMBER(5) PRIMARY KEY,
   Ename      VARCHAR2(15) NOT NULL,
   Job        VARCHAR2(10),
   Mgr        NUMBER(5),
   Hiredate   DATE DEFAULT (sysdate),
   Sal        NUMBER(7,2),
   Comm       NUMBER(7,2),
   Deptno     NUMBER(3) NOT NULL,
              CONSTRAINT dept_afkey REFERENCES Dept_tab(Deptno))
   PCTFREE 10
   PCTUSED 40
   TABLESPACE users
   STORAGE (  INITIAL 50K
              NEXT 50K
              MAXEXTENTS 10
              PCTINCREASE 25 );
```

一時表の管理

Oracle8*i* から、一時データを保持するための特別な表が提供されています。データがセッション固有かトランザクション固有かを指定します。セッションまたはトランザクションが終了すると、挿入された行は削除されます。複数のセッションまたはトランザクションは同じ一時表を使用でき、各セッションまたはトランザクションは、それぞれが作成した行のみを参照します。

一時表は、結果セットをバッファリングしたり、複数の DML 操作を実行することによって結果セットを構成する場合に有効です。一時表が有効な具体例を次に示します。

- Web ベースの航空券予約アプリケーションでは、オプションの旅行日程をいくつか作成できます。それぞれの旅行日程は一時表内の行で表されます。アプリケーションによって行が更新されると、変更は旅行日程にも反映されます。どの旅行日程を使用するかを決めると、アプリケーションによってその日程の行が表に移されます。

  セッション中、旅行日程のデータはそのセッション専用です。セッションを終了すると、作成したオプションの旅行日程は削除されます。

- 大規模書店の複数の営業担当者は、電話で顧客から注文を受けながら、1 つの一時表を同時に使用します。顧客注文を入力、変更するために、各担当者は 1 つのセッション内でこの一時表にアクセスしますが、これを他の営業担当者が使用することはできません。担当者がセッションを閉じると、そのセッションのデータは自動的に削除されますが、表の構造は存続し、他の担当者が使用できます。

- 管理者は、他の方法では複雑でコストのかかる問合せの実行パフォーマンスを向上させるために一時表を使用します。このために、より複雑な問合せからの値を一時表にキャッシュし、結合などの SQL 文をこの一時表に対して実行します。この実行方法の詳細は、2-6 ページの「例 : 一時表を使用したパフォーマンスの改善」を参照してください。

一時表の作成

一時表は特殊な ANSI キーワードを使用して作成します。データを**セッション固有**として指定するには、ON COMMIT PRESERVE ROWS キーワードを使用します。データを**トランザクション固有**として指定するには、ON COMMIT DELETE ROWS キーワードを使用します。

例 2-1 セッション固有一時表の作成

```
CREATE GLOBAL TEMPORARY TABLE ...
     [ON COMMIT PRESERVE ROWS ]
```

例 2-2 トランザクション固有一時表の作成

```
CREATE GLOBAL TEMPORARY TABLE ...
     [ON COMMIT DELETE ROWS ]
```

一時表の使用

一時表には、表の場合と同じように索引を作成できます。

セッション固有の一時表の場合、セッション内で最初に一時表に挿入が実行されたときに、そのセッションが一時表にバインドされます。このバインドは、セッションの終了時またはセッション内で表の TRUNCATE が発行されたときに消滅します。

トランザクション固有の一時表の場合は、トランザクション内で最初に一時表に挿入が実行されたときに、そのトランザクションが一時表にバインドされます。このバインドはトランザクションの終了時に消滅します。

既存の一時表に対して DDL 操作（TRUNCATE 以外）を実行できるのは、その一時表にセッションがバインドされていないときのみです。

表とは異なり、一時表およびその索引が作成されるときにセグメントが自動的に割り当てられることはありません。セグメントは、最初の INSERT（または CREATE TABLE AS SELECT）が実行されるときに割り当てられます。これは、最初の INSERT の前に SELECT、UPDATE または DELETE が実行されると、表は空のように見えるということを意味します。

一時セグメントの割当ては、トランザクション固有の一時表の場合はトランザクションの終了時、セッション固有の一時表の場合はセッション終了時に解除されます。

トランザクションをロールバックした場合、表定義は残りますが、入力したデータは失われます。

トランザクション固有であり、セッション固有でもあるという表は作成できません。

トランザクション固有の一時表に許可されるトランザクションは 1 回に 1 つのみです。1 つのトランザクション・スコープ内に自律型トランザクションがいくつかある場合、自律型トランザクションは、その前のトランザクションがコミットした直後でないと表を使用できません。

一時表の中のデータは一時的であるため、システム障害時にはバックアップされずリカバリもされません。このような障害に備えて、一時表のデータを保持しておく方法を作成する必要があります。

例 : 一時表の使用

例 : セッション固有の一時表

次の文は、航空券自動予約システムに使用するセッション固有の一時表 FLIGHT_SCHEDULE を作成します。顧客はそれぞれ独自のセッションを占有し、仮の予約を格納できます。仮の予約は、セッション終了時に削除されます。

```
CREATE GLOBAL TEMPORARY TABLE flight_schedule (
   startdate DATE,
   enddate DATE,
   cost NUMBER)
ON COMMIT PRESERVE ROWS;
```

例 : 一時表を使用したパフォーマンスの改善

一時表を使用すると、複合問合せを実行した際のパフォーマンスを改善できます。複数の複合問合せの実行は、戻される行ごとに表が複数回アクセスされるため、比較的時間がかかります。一時表内の複合問合せから値をキャッシュし、一時表に対して問合せを実行すると処理が速くなります。

たとえば、その後の問合せを簡単にするために定義された次のようなビューであっても、ビューに対する問合せは、ビューの内容が毎回再計算されるため、時間がかかることがあります。

```
CREATE OR REPLACE VIEW Profile_values_view AS
SELECT d.Profile_option_name, d.Profile_option_id, Profile_option_value,
       u.User_name, Level_id, Level_code
  FROM Profile_definitions d, Profile_values v, Profile_users u
 WHERE d.Profile_option_id = v.Profile_option_id
   AND ((Level_code = 'USER' AND Level_id = U.User_id) OR
        (Level_code = 'DEPARTMENT' AND Level_id = U.Department_id) OR
        (Level_code = 'SITE'))
   AND NOT EXISTS (SELECT 1 FROM PROFILE_VALUES P
                   WHERE P.PROFILE_OPTION_ID = V.PROFILE_OPTION_ID
                     AND ((Level_code = 'USER' AND
                           level_id = u.User_id) OR
                          (Level_code = 'DEPARTMENT' AND
                           level_id = u.Department_id) OR
                          (Level_code = 'SITE'))
                     AND INSTR('USERDEPARTMENTSITE', v.Level_code) >
                         INSTR('USERDEPARTMENTSITE', p.Level_code));
```

一時表を使用すると計算は 1 回のみ行われ、その結果は後の SQL 問合せおよび結合用にキャッシュされます。

```
CREATE GLOBAL TEMPORARY TABLE Profile_values_temp
        (
            Profile_option_name   VARCHAR(60)   NOT NULL,
            Profile_option_id     NUMBER(4)     NOT NULL,
            Profile_option_value  VARCHAR2(20)  NOT NULL,
            Level_code            VARCHAR2(10)          ,
            Level_id              NUMBER(4)             ,
            CONSTRAINT Profile_values_temp_pk
               PRIMARY KEY (Profile_option_id)
        ) ON COMMIT PRESERVE ROWS ORGANIZATION INDEX;

INSERT INTO Profile_values_temp
       (Profile_option_name, Profile_option_id, Profile_option_value,
        Level_code, Level_id)
SELECT Profile_option_name, Profile_option_id, Profile_option_value,
        Level_code, Level_id
  FROM Profile_values_view;
COMMIT;
```

このように、一時表は問合せの高速化に使用できます。また、一時表にキャッシュされた結果は、セッション終了時にデータベースによって自動的に解放されます。

ヒント : 同一副問合せの複数回の参照

同じ副問合せを複数回処理する複合問合せでは、一時表に副問合せの結果を格納し、その一時表に対して問合せを行うことが有効な場合があります。WITH 句を使用すると、副問合せを因数化し、名前を付けて、元の複合問合せ内で複数回参照させることができます。

これによって、オプティマイザは、一時表を作成して副問合せの結果を処理するか、またはビューとしてインライン処理するかを選択できます。

たとえば、次の問合せでは、2 つの表を結合し、SUM(SAL) 集計を 2 回以上計算します。太字は、問合せが繰り返される部分を表します。

```
SELECT dname, SUM(sal) AS dept_total
  FROM emp, dept
  WHERE emp.deptno = dept.deptno
  GROUP BY dname HAVING
    SUM(sal) >
   (
      SELECT SUM(sal) * 1/3
       FROM emp, dept
        WHERE emp.deptno = dept.deptno
   )
   ORDER BY SUM(sal) DESC;
```

副問合せを 1 回実行し、メインの問合せ内の適切な位置で参照させることによって、問合せを改善できます。太字は、副問合せの共通部分および副問合せが参照されている箇所を表します。

```
WITH
summary AS
(
  SELECT dname, SUM(sal) AS dept_total
    FROM emp, dept
    WHERE emp.deptno = dept.deptno
    GROUP BY dname
)
SELECT dname, dept_total
   FROM summary
  WHERE dept_total >
(
  SELECT SUM(dept_total) * 1/3
    FROM summary
)
ORDER BY dept_total DESC;
```

参照：

- WITH 句の完全な構文については、『Oracle9*i* SQL リファレンス』を参照してください。
- この機能を使用して、パフォーマンスを向上させる方法については、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

ビューの管理

ビューは、別の表または複数の表の組合せを論理的に表すものです。ビューは、そのデータを基礎となる表から導出します。これらの表は、**実表**と呼ばれます。実表は、表でもビューでもかまいません。

ビューに対して行われる操作は、すべてビューの実表に影響します。ビューは、表とほとんど同じ方法で使用できます。表と同じように、ビューに対しても問合せ、更新、挿入および削除を行うことができます。

ビューによって、他の表およびビューに存在するデータを別の形（サブセットやスーパーセットなど）で表現できます。ユーザーのタイプに応じてデータの外観を変更できるため、ビューは非常に強力です。

次の項では、SQL コマンドを使用して、ビューを作成、置換および削除する方法について説明します。

ビューの作成

ビューを作成するには、SQL コマンド CREATE VIEW を使用します。たとえば、次の文は、EMP_TAB 表のデータのサブセットにビューを作成します。

```
CREATE VIEW Sales_staff AS
    SELECT Empno, Ename, Deptno
    FROM Emp_tab
    WHERE Deptno = 10
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

オブジェクト名は、ビューが作成されたとき、または SQL を含むプログラムがコンパイルされたときに、ビューの所有者のスキーマで解決されます。

ビューは、表、スナップショット、または他のビューを参照する問合せによって定義できます。

SALES_STAFF ビューを定義する問合せは、部門 10 の行のみを参照します。また、WITH CHECK OPTION は、そのビューを制約付き（ビューに対して発行された INSERT 文および UPDATE 文を使用すると、ビューで選択できない行は、作成できないし、結果としても戻せない）で作成します。

前述の例の場合、次の INSERT 文は、SALES_STAFF ビューを介して EMP_TAB 表に行を挿入します。

```
INSERT INTO Sales_staff VALUES (7584, 'OSTER', 10);
```

ただし、次の INSERT 文は、SALES_STAFF ビューでは選択できない部門番号 30 の行を挿入するため、ロールバックされ、エラーを戻します。

```
INSERT INTO Sales_staff VALUES (7591, 'WILLIAMS', 30);
```

次の文は、Emp_tab 表と Dept_tab 表のデータを結合するビューを作成します。

```
CREATE VIEW Division1_staff AS
    SELECT Ename, Empno, Job, Dname
    FROM Emp_tab, Dept_tab
    WHERE Emp_tab.Deptno IN (10, 30)
    AND Emp_tab.Deptno = Dept_tab.Deptno;
```

Division1_staff ビューは、Emp_tab 表と Dept_tab 表の情報を結合する問合せによって定義されます。WITH CHECK OPTION を使用した結合を含む問合せで定義されたビューへの行の挿入または更新はできないため、この CREATE VIEW 文に WITH CHECK OPTION は指定されていません。

ビュー作成時の問合せ定義の展開

ビューが作成され、その結果としてできる問合せをデータ・ディクショナリに格納するときに、Oracle は ANSI/ISO 規格に従って、トップレベルのビュー問合せにあるワイルド・カードを列リストに展開します（副問合せは元のままの状態です）。展開される列リストの列名は引用符で囲まれます。ベース・オブジェクトの列が引用符付きで登録されており、正しい構文の問合せにするには引用符が必要となる可能性があることを考慮して、このような処理を行います。

Dept_view ビューが次の文で作成される例を考えます。

```
CREATE VIEW Dept_view AS SELECT * FROM scott.Dept_tab;
```

Oracle は、Dept_view ビューの定義の問合せを次のように格納します。

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.Dept_tab;
```

ビューの作成でエラーが発生すると、ワイルド・カードは展開されません。ただし、ビューがエラーなしでコンパイルされると、ビューを定義している問合せのワイルド・カードが展開されます。

エラーを伴うビューの作成

ビューを定義している問合せが実行できなくても、CREATE VIEW コマンドに構文エラーがないかぎりビューを作成できます。このようなビューは、**エラーを伴うビュー**と呼ばれます。たとえば、ビューが存在しない表または既存の表の無効な列を参照している場合、あるいはビューの所有者が必要な権限を持っていない場合にも、ビューは作成され、データ・ディクショナリに登録されます。

CREATE VIEW コマンドに FORCE オプションを使用した場合にかぎり、エラーを伴うビューを作成できます。

```
CREATE FORCE VIEW AS ...;
```

エラーを伴うビューが作成された場合、Oracle はメッセージを戻し、ビューを INVALID のまま残します。その後、無効なビュー問合せが実行できるようになると、そのビューは再コンパイルされ、有効（使用可能）になります。無効なビューを使用しようとすると、Oracle はそのビューを動的にコンパイルします。

ビューの作成に必要な権限

ビューを作成するには、次の権限が必要です。

- 自スキーマにビューを作成するには、CREATE VIEW システム権限が必要です。また、別のユーザーのスキーマにビューを作成するには、CREATE ANY VIEW システム権限が必要です。これらの権限は、明示的に取得されるか、またはロールを介して取得されます。
- ビューの**所有者**は、ビューの定義で参照するすべてのオブジェクトにアクセスするために必要な権限を明示的に付与されている必要があります。ただし、ロールを介して必要な権限を取得することはできません。また、ビューの機能は、ビューの所有者が持っている権限に依存します。たとえば、Scott の EMP_TAB 表に対して INSERT 権限のみが付与されている場合は、Scott の EMP_TAB 表に対してビューを作成できます。ただし、このビューは、EMP_TAB 表に新しい行を挿入するためにのみ使用できます。
- ビューの所有者が、ビューへのアクセス権限を他のユーザーに付与する場合、ベース・オブジェクトの Grant Option 付きのオブジェクト権限または Admin Option 付きのシステム権限が必要です。これらの権限がないと、ビューへのアクセス権限を他のユーザーに付与する権限としては不十分です。

ビューの置換

ビューの定義を変更するには、次のいずれかの方法を使用してビューを置換する必要があります。

- ビューは、削除してから再作成できます。ビューが削除されると、対応するすべてのビュー権限の付与がロールおよびユーザーから取り消されます。ビューの再作成後、必要な権限を再度付与する必要があります。

- OR REPLACE オプションを含む CREATE VIEW 文で再定義することによって、ビューを置換できます。このオプションは、ビューの現行の定義は置換しますが、現在のセキュリティ認可はそのまま残します。

  たとえば、前述の例のように、SALES_STAFF ビューを作成すると想定します。また、ロールおよび他のユーザーにいくつかのオブジェクト権限も付与します。ただし、部門番号は 30 にする必要があるため、定義している問合せの WHERE 句で指定されている部門番号を 30 に修正するために、SALES_STAFF ビューを再定義する必要があります。オブジェクト権限の付与を保存するために、次の文を使用して、SALES_STAFF を現行のバージョンに置換できます。

```
CREATE OR REPLACE VIEW Sales_staff AS
    SELECT Empno, Ename, Deptno
    FROM Emp_tab
    WHERE Deptno = 30
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

ビューの置換による影響は、次のとおりです。

- ビューの置換は、データ・ディクショナリ内のビューの定義を置換します。ビューによって参照される基本オブジェクトは影響を受けません。
- 前に定義されたが、新しいビュー定義には含められなかった場合、ビュー定義の WITH CHECK OPTION に対応付けられている制約は削除されます。
- 置換されたビューに依存するすべてのビューおよび PL/SQL プログラム・ユニットは無効になります。

ビューの置換に必要な権限

ビューを置換するには、ビューの作成に必要な権限およびビューの削除に必要なすべての権限が必要です。

問合せのビューの使用

表と同じ方法でビューを問い合せることができます。たとえば、Division1_staff ビューを問い合せるには、そのビューを参照する有効な SELECT 文を入力します。

```
SELECT * FROM Division1_staff;

ENAME        EMPNO       JOB             DNAME
-----------------------------------------------------
CLARK          7782      MANAGER         ACCOUNTING
KING           7839      PRESIDENT       ACCOUNTING
MILLER         7934      CLERK           ACCOUNTING
ALLEN          7499      SALESMAN        SALES
WARD           7521      SALESMAN        SALES
JAMES          7900      CLERK           SALES
TURNER         7844      SALESMAN        SALES
```

```
MARTIN          7654        SALESMAN        SALES
BLAKE           7698        MANAGER         SALES
```

いくつかの制限はありますが、ビューを使用して、実表に行を挿入、更新または削除できます。次の文は、SALES_STAFF ビューを使用して、EMP_TAB 表に新しい行を挿入します。

```
INSERT INTO Sales_staff
    VALUES (7954, 'OSTER', 30);
```

ビューに対する DML 操作制限では、次の基準がリストされた順に使用されます。

1. ビューが、SET または DISTINCT 演算子、GROUP BY 句、あるいはグループ関数を含む問合せによって定義される場合、ビューを使用して実表に行を挿入、更新または削除することはできません。
2. ビューが WITH CHECK OPTION 付きで定義されている場合、そのビューが実表から行を選択できない場合は、実表に対して（ビューを使用して）行を挿入または更新することはできません。
3. ビューから DEFAULT 句を持たない NOT NULL 列が省略されている場合、そのビューを使用して実表に行を挿入することはできません。
4. ビューが DECODE（deptno, 10, "SALES", ...）などの式を使用して作成された場合、そのビューを使用して実表に行を挿入または更新することはできません。

SALES_STAFF ビューの WITH CHECK OPTION で作成した制約によって、EMP_TAB 表で部門番号が 10 の行のみを挿入したり、更新することができます。また、SALES_STAFF ビューが次の文によって定義されるものとします（DEPTNO 列を除外しています）。

```
CREATE VIEW Sales_staff AS
    SELECT Empno, Ename
    FROM Emp_tab
    WHERE Deptno = 10
    WITH CHECK OPTION CONSTRAINT Sales_staff_cnst;
```

このビュー定義では、既存のレコードについて EMPNO フィールドおよび ENAME フィールドを更新できます。ただし、ユーザーは DEPTNO フィールドを変更できないため、SALES_STAFF ビューを介して EMP_TAB 表に行を挿入できません。DEPTNO フィールドの DEFAULT 値として 10 が設定されている場合、挿入を実行できます。

無効なビューの参照 ユーザーが無効なビューを参照しようとすると、Oracle は次のようなエラー・メッセージを戻します。

```
ORA-04063: 'view_name' にエラーがあります。
```

ビューが存在しても、問合せのエラーが原因でビューを使用できない（作成されたときにビューのエラーがあったか、またはビューの作成には成功したが、基礎となるオブジェクトが変更または削除されたために使用できなくなった）場合に、このエラー・メッセージが戻されます。

ビューの使用に必要な権限

ビューに対して問合せ、あるいは INSERT 文、UPDATE 文または DELETE 文を発行するには、そのビューに対する SELECT、INSERT、UPDATE または DELETE の各オブジェクト権限がそれぞれ明示的に、またはロールを介して付与されている必要があります。

ビューの削除

ビューを削除するには、SQL コマンド DROP VIEW を使用します。次に例を示します。

```
DROP VIEW Sales_staff;
```

ビューの削除に必要な権限

自スキーマに含まれるビューはどれでも削除できます。別のユーザーのスキーマのビューを削除するには、DROP ANY VIEW システム権限が必要です。

結合ビューの変更

Oracle では、制限はいくつかありますが、結合に関連するビューを変更できます。次の単純なビューについて検討してみます。

```
CREATE VIEW Emp_view AS
    SELECT Ename, Empno, deptno FROM Emp_tab;
```

このビューには結合操作は含まれていません。次のような SQL 文を発行した場合、

```
UPDATE Emp_view SET Ename = 'CAESAR' WHERE Empno = 7839;
```

そのビューの基礎となる EMP_TAB 実表が変更され、EMP_TAB 表内の従業員 7839 の名前が KING から CAESAR に変更されます。

ただし、結合操作を含む次のようなビューを作成した場合、

```
CREATE VIEW Emp_dept_view AS
  SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
    FROM Emp_tab e, Dept_tab d    /* JOIN operation */
     WHERE e.Deptno = d.Deptno
       AND d.Loc IN ('DALLAS', 'NEW YORK', 'BOSTON');
```

このビューを介して EMP_TAB 実表または DEPT_TAB 実表を変更するには制限があります。たとえば、次のような文を使用する場合です。

```
UPDATE Emp_dept_view SET Ename = 'JOHNSON'
    WHERE Ename = 'SMITH';
```

変更可能な結合ビューとは、SELECT 文のトップレベルの FROM 句で複数の表が定義され、次のいずれも含まないビューのことです。

- DISTINCT 演算子
- 集計関数 : AVG、COUNT、GLB、MAX、MIN、STDDEV、SUM、VARIANCE
- 集合演算 : UNION、UNION ALL、INTERSECT、MINUS
- GROUP BY 句または HAVING 句
- START WITH 句または CONNECT BY 句
- ROWNUM 疑似列

結合ビューが変更可能であるためのもう 1 つの条件として、ビューが他のネストされたビューでの結合である場合、それらのネストされたビューは、トップレベル・ビューにマージ可能である必要があります。

> **参照 :**
>
> マージ可能なビューの詳細は、『Oracle9*i* データベース概要』を参照してください。
>
> カスタマイズされたトリガーを記述して、結合ビューの更新シミュレーションを行う方法については、15-6 ページの「複合ビューの変更 (INSTEAD OF トリガー)」を参照してください。

結合ビューの変更例

この項の例では、EMP_TAB 表および DEPT_TAB 表を使用しています。ただし、これらの例は、表で主キーおよび外部キーを明示的に定義するか、または一意索引を定義しないと役に立ちません。EMP_TAB および DEPT_TAB のための適切な制約付き表定義を次に示します。

```
CREATE TABLE Dept_tab (
        Deptno    NUMBER(4) PRIMARY KEY,
        Dname     VARCHAR2(14),
        Loc       VARCHAR2(13));

CREATE TABLE Emp_tab (
        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),
FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno));
```

前述の主キーおよび外部キー制約を省略し、DEPT_TAB (DEPTNO) に UNIQUE INDEX を作成すると、後述の例を使用できます。

キー保存表

キー保存表という概念は、結合ビューの変更についての制限を理解するうえでの基本です。表は、その表の各キーが結合の結果、キーになる可能性がある場合に「キー保存」といいます。キー保存表では、結合によってそのキーが保存されます。

注意：

- ある表がキー保存表であるには、その表のキーすべてが選択される必要はありません。その表のキー（複数の場合もある）が選択されたときに、そのキーが結合の結果のキーであれば十分です。
- 表のキー保存特性は、その表に入っている実際のデータに依存しません。この特性は、スキーマの特性であり、表内のデータの特性ではありません。たとえば、EMP_TAB 表の各部門に多くても 1 人の従業員以外存在しない場合、EMP_TAB と DEPT_TAB を結合すると DEPT_TAB.DEPTNO は一意になりますが、DEPT_TAB はキー保存表にはなりません。

「結合ビューの変更」の項で定義されている EMP_DEPT_VIEW からすべての行を SELECT した場合、結果は次のようになります。

```
EMPNO      ENAME      DEPTNO     DNAME          LOC
-----------------------------------------------------------
7782       CLARK      10         ACCOUNTING     NEW YORK
7839       KING       10         ACCOUNTING     NEW YORK
7934       MILLER     10         ACCOUNTING     NEW YORK
7369       SMITH      20         RESEARCH       DALLAS
7876       ADAMS      20         RESEARCH       DALLAS
7902       FORD       20         RESEARCH       DALLAS
7788       SCOTT      20         RESEARCH       DALLAS
7566       JONES      20         RESEARCH       DALLAS
8 rows selected.
```

このビューで、EMPNO は EMP_TAB 表のキーであり、この結合の結果のキーでもあるため、EMP_TAB はキー保存表です。DEPTNO は DEPT_TAB 表のキーですが、結合のキーではないため、DEPT_TAB はキー保存表ではありません。

結合ビューの DML 文の規則

結合ビューの UPDATE 文、INSERT 文または DELETE 文はいずれも、基礎となる実表を 1 つのみ変更できます。

結合ビューの更新

次の例は、EMP_DEPT_VIEW ビューを正常に変更する UPDATE 文です。

```
UPDATE Emp_dept_view
  SET Sal = Sal * 1.10
    WHERE Deptno = 10;
```

次の UPDATE 文は、EMP_DEPT_VIEW ビューでは使用できません。

```
UPDATE Emp_dept_view
  SET Loc = 'BOSTON'
    WHERE Ename = 'SMITH';
```

この文は、基礎となる DEPT_TAB 表（DEPT_TAB は EMP_DEPT ビューではキー保存されていません）を変更しようとしたため、ORA-01779 エラー（キー保存されていない表にマップする列は変更できません）が発生します。

一般的に、結合ビューの変更可能なすべての列は、キー保存表の列にマップする必要があります。WITH CHECK OPTION 句を指定してビューを定義した場合、すべての結合列および反復する表のすべての列が変更可能になるわけではありません。

したがって、たとえば、EMP_DEPT ビューが WITH CHECK OPTION を使用して定義されている場合、次の UPDATE 文は失敗します。

```
UPDATE Emp_dept_view
    SET Deptno = 10
        WHERE Ename = 'SMITH';
```

この文は、結合列を更新しようとしているため、正常に実行されません。

結合ビューからの削除

結合内でキー保存表が 1 つのみの場合は、結合ビューから削除できます。

次の DELETE 文は、EMP_DEPT ビューで使用できます。

```
DELETE FROM Emp_dept_view
    WHERE Ename = 'SMITH';
```

EMP_DEPT ビューに対するこの DELETE 文は有効です。これは、基礎となる EMP_TAB 表で DELETE 操作に変換することができ、この EMP_TAB 表は結合内の唯一のキー保存表であるためです。

次のビューでは、E1 および E2 は両方ともキー保存表であるため、このビューに対して DELETE 操作を実行できません。

```
CREATE VIEW emp_emp AS
    SELECT e1.Ename, e2.Empno, e1.Deptno
        FROM Emp_tab e1, Emp_tab e2
        WHERE e1.Empno = e2.Empno;
```

WITH CHECK OPTION 句を指定してビューを定義し、キー保存表が繰り返される場合、このビューから行を削除できません。次に例を示します。

```
CREATE VIEW Emp_mgr AS
    SELECT e1.Ename, e2.Ename Mname
       FROM Emp_tab e1, Emp_tab e2
            WHERE e1.mgr = e2.Empno
            WITH CHECK OPTION;
```

このビューは、キー保存表の自己結合に関連しているため、このビューでの削除は実行できません。

結合ビューへの挿入

EMP_DEPT ビューに対する次の INSERT 文は正常に実行されます。これは、変更されるキー保存実表が 1 つのみ（EMP_TAB）であり、40 は DEPT_TAB 表で有効な DEPTNO である（EMP_TAB 表の外部キー整合性制約に一致する）ためです。

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES ('KURODA', 9010, 40);
```

次の INSERT 文は、正常に実行されません。実表 EMP_TAB の UPDATE も正常に実行されません。これは、EMP_TAB 表の外部キー整合性制約に違反するためです。

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES ('KURODA', 9010, 77);
```

次の INSERT 文は、ORA-01776 エラー（結合ビューを介して複数の実表を変更できません。）で正常に実行されません。

```
INSERT INTO Emp_dept (Ename, Empno, Deptno)
    VALUES (9010, 'KURODA', 'BOSTON');
```

INSERT では、明示的にも暗黙的にも、キー保存されていない表の列を参照できません。WITH CHECK OPTION 句を指定して定義された結合ビューに対して、INSERT を実行できません。

UPDATABLE_COLUMNS ビューの使用

表 2-1 に、結合ビューの変更に使用できる 3 つのビューを示します。

表 2-1 UPDATABLE_COLUMNS ビュー

ビュー名	説明
USER_UPDATABLE_COLUMNS	ユーザー・スキーマのすべての表およびビュー内のすべての変更可能な列を示します。
DBA_UPDATABLE_COLUMNS	DBA スキーマに入っているすべての表およびビュー内のすべての変更可能な列を示します。
ALL_UPDATABLE_COLUMNS	すべての表およびビュー内のすべての変更可能な列を示します。

外部結合

場合によっては、外部結合を含むビューが変更可能である場合があります。次に例を示します。

```
CREATE VIEW Emp_dept_oj1 AS
    SELECT Empno, Ename, e.Deptno, Dname, Loc
    FROM Emp_tab e, Dept_tab d
    WHERE e.Deptno = d.Deptno (+);
```

次の文を指定したとします。

```
SELECT * FROM Emp_dept_oj1;
```

結果は次のようになります。

```
EMPNO   ENAME      DEPTNO  DNAME          LOC
------- ---------- ------- -------------- -------------
7369    SMITH      40      OPERATIONS     BOSTON
7499    ALLEN      30      SALES          CHICAGO
7566    JONES      20      RESEARCH       DALLAS
7654    MARTIN     30      SALES          CHICAGO
7698    BLAKE      30      SALES          CHICAGO
7782    CLARK      10      ACCOUNTING     NEW YORK
7788    SCOTT      20      RESEARCH       DALLAS
7839    KING       10      ACCOUNTING     NEW YORK
7844    TURNER     30      SALES          CHICAGO
7876    ADAMS      20      RESEARCH       DALLAS
7900    JAMES      30      SALES          CHICAGO
7902    FORD       20      RESEARCH       DALLAS
7934    MILLER     10      ACCOUNTING     NEW YORK
7521    WARD       30      SALES          CHICAGO
14 rows selected.
```

EMP_TAB は結合内のキー保存表であるため、EMP_DEPT_OJ1 の基礎となる EMP_TAB 表内の列は変更可能です。

次のビューにも外部結合が含まれます。

```
CREATE VIEW Emp_dept_oj2 AS
SELECT e.Empno, e.Ename, e.Deptno, d.Dname, d.Loc
FROM Emp_tab e, Dept_tab d
WHERE e.Deptno (+) = d.Deptno;
```

次の文を指定したとします。

```
SELECT * FROM Emp_dept_oj2;
```

結果は次のようになります。

```
EMPNO      ENAME      DEPTNO    DNAME          LOC
---------- ---------- --------- -------------- ----
7782       CLARK      10        ACCOUNTING     NEW YORK
7839       KING       10        ACCOUNTING     NEW YORK
7934       MILLER     10        ACCOUNTING     NEW YORK
7369       SMITH      20        RESEARCH       DALLAS
7876       ADAMS      20        RESEARCH       DALLAS
7902       FORD       20        RESEARCH       DALLAS
7788       SCOTT      20        RESEARCH       DALLAS
7566       JONES      20        RESEARCH       DALLAS
7499       ALLEN      30        SALES          CHICAGO
7698       BLAKE      30        SALES          CHICAGO
7654       MARTIN     30        SALES          CHICAGO
7900       JAMES      30        SALES          CHICAGO
7844       TURNER     30        SALES          CHICAGO
7521       WARD       30        SALES          CHICAGO
                                 OPERATIONS     BOSTON
15 rows selected.
```

このビューでは、結合の結果の EMPNO 列には NULL も有効なため（前述の SELECT 内の最後の行）、EMP_TAB はすでにキー保存表ではありません。そのため、UPDATE、DELETE および INSERT の各操作をこのビューで実行できません。

他のネストされたビューとの外部結合を含むビューの場合、表がキー保存となるためには、表を含むビュー（複数の場合もある）が、その外部のビューに順にマージされ、そのマージが一番上まで達している必要があります。外部結合されるビューは、単純な場合にのみ、この時点でマージされます。次に例を示します。

```
SELECT Col1, Col2, ... FROM T;
```

このビューの SELECT 構文のリストに式はなく、WHERE 句はありません。

次の一連のビューを検討してください。

```
CREATE VIEW Emp_v AS
    SELECT Empno, Ename, Deptno
        FROM Emp_tab;
CREATE VIEW Emp_dept_oj1 AS
    SELECT e.*, Loc, d.Dname
        FROM Emp_v e, Dept_tab d
            WHERE e.Deptno = d.Deptno (+);
```

前述の例では、EMP_V は単純なビューであるため EMP_DEPT_OJ1 にマージされます。したがって、EMP_TAB はキー保存表です。ただし、EMP_V が次のように変更された場合、

```
CREATE VIEW Emp_v_2 AS
    SELECT Empno, Ename, Deptno
        FROM Emp_tab
            WHERE Sal > 1000;
```

WHERE 句が存在するため、EMP_V_2 を EMP_DEPT_OJ1 にマージできません。そのため、EMP_TAB は、すでにキー保存表ではありません。

ビューが変更可能であるかどうかが不明な場合、USER_UPDATABLE_COLUMNS ビューから選択（SELECT）して確認できます。次に例を示します。

```
SELECT * FROM USER_UPDATABLE_COLUMNS WHERE TABLE_NAME = 'EMP_DEPT_VIEW';
```

次のような結果が戻されます。

```
OWNER       TABLE_NAME      COLUMN_NAM      UPD
----------  ----------      ----------      ---
SCOTT       EMP_DEPT_V      EMPNO           NO
SCOTT       EMP_DEPT_V      ENAME           NO
SCOTT       EMP_DEPT_V      DEPTNO          NO
SCOTT       EMP_DEPT_V      DNAME           NO
SCOTT       EMP_DEPT_V      LOC             NO
5 rows selected.
```

順序の管理

シーケンス・ジェネレータは順序番号を生成します。順序番号の生成は、データに対して一意の主キーを自動的に生成する場合、あるいは複数の行または表にまたがるキーを調整する場合に有効です。

順序が存在しない場合、連続的な値は、プログラムによってのみ生成できます。新しい主キーの値は、最も新しく生成された値を選択し、その値を増分することによって取得できます。この方法では、トランザクション時にロックが必要なため、複数のユーザーが次の主キーの値を待機する必要があります。この待機を**シリアル化**と呼びます。アプリケーションにこのような構成がある場合、順序にアクセスするように置換する必要があります。順序によってシリアル化がなくなり、アプリケーションの同時実行性が改善されます。

後述の項では、SQL コマンドを使用して、順序を作成、使用、変更および削除する方法を説明します。

順序の作成

順序を作成するには、SQL コマンド CREATE SEQUENCE を使用します。次の文は、EMP_TAB 表の EMPNO 列に対して従業員番号を生成するために使用する順序を作成します。

```
CREATE SEQUENCE Emp_sequence
    INCREMENT BY 1
    START WITH 1
    NOMAXVALUE
    NOCYCLE
    CACHE 10;
```

パラメータの中には、順序の機能を制御するために指定できるものがあることに注意してください。これらのパラメータを使用して、順序の昇順または降順、順序の開始点、最大値および最小値、順序値の間隔を指定できます。NOCYCLE オプションは、最大値または最小値に到達した後は、順序が値を生成できないことを示します。

CREATE SEQUENCE コマンドの CACHE オプションは、順序番号をより速くアクセスできるように、事前に順序番号の集合をメモリー内に割り当て、それらを保持します。キャッシュ内の最後の順序番号が使用されると、別の順序の集合がキャッシュ内に読み込まれます。

> **参照：** Oracle Real Application Clusters を使用するときの順序番号のキャッシュの詳細は、『Oracle9*i* Real Application Clusters 概要』、『Oracle9*i* Real Application Clusters セットアップおよび構成』、『Oracle9*i* Real Application Clusters 管理』、『Oracle9*i* Real Application Clusters 配置およびパフォーマンス』を参照してください。
>
> 順序番号のキャッシュに関する一般的な情報は、2-24 ページの「順序番号のキャッシュ」を参照してください。

順序の作成に必要な権限

自スキーマに順序を作成するには、CREATE SEQUENCE システム権限が必要です。別のユーザーのスキーマに順序を作成するには、CREATE ANY SEQUENCE システム権限が必要です。

順序の変更

対応する順序番号の生成方法を定義するパラメータを変更できます。ただし、順序の開始番号を変更することはできません。これを行うには、順序を削除してから、再作成する必要があります。

順序を変更するには、SQL コマンド ALTER SEQUENCE を使用します。次に例を示します。

```
ALTER SEQUENCE Emp_sequence
INCREMENT BY 10
```

```
    MAXVALUE 10000
    CYCLE
    CACHE 20;
```

順序の変更に必要な権限

順序を変更するには、自スキーマにその順序が含まれているか、または ALTER ANY SEQUENCE システム権限が必要です。

順序の使用

一度定義された順序は、複数のユーザーが待機することなく、アクセスおよび増分できます。Oracle は、順序を増分したトランザクションの完了を待たずに、その順序を増分します。

次の項では、マスター / ディテール表関連で順序を使用する方法について説明します。顧客の注文に関する情報を保持する受注システムが、ORDERS_TAB（マスター表）および LINE_ITEMS_TAB（ディテール表）の 2 つの表によって構成されていると想定します。順序 ORDER_SEQ は、次の文によって定義されます。

```
CREATE SEQUENCE Order_seq
    START WITH 1
    INCREMENT BY 1
    NOMAXVALUE
    NOCYCLE
    CACHE 20;
```

順序の参照

順序は、NEXTVAL 疑似列および CURRVAL 疑似列を使用した SQL 文で参照されます。現行の順序番号は、順序の疑似列 CURRVAL を使用して繰り返し参照できますが、新しい順序番号はそれぞれ、疑似列 NEXTVAL を参照することによって生成されます。

NEXTVAL および CURRVAL は、予約語またはキーワードではなく、SELECT、INSERT、UPDATE などの SQL 文において疑似列名として使用できます。

NEXTVAL を使用した順序番号の生成 順序番号を生成および使用するには、*seq_name*.NEXTVAL を参照します。たとえば、顧客が発注する場合を想定します。順序番号は値リストで参照できます。次に例を示します。

```
INSERT INTO Orders_tab (Orderno, Custno)
    VALUES (Order_seq.NEXTVAL, 1032);
```

また、順序番号は UPDATE 文の SET 句でも参照できます。次に例を示します。

```
UPDATE Orders_tab
```

```
    SET Orderno = Order_seq.NEXTVAL
    WHERE Orderno = 10112;
```

順序番号は、問合せまたは副問合せの一番外側の SELECT でも参照できます。次に例を示します。

```
SELECT Order_seq.NEXTVAL FROM dual;
```

定義どおり、ORDER_SEQ.NEXTVAL への最初の参照は値 1 を戻します。ORDER_SEQ.NEXTVAL を参照するそれぞれの後続する文は、次の順序番号（2、3、4、...）を生成します。疑似列 NEXTVAL を使用して、新しい順序番号を必要なだけ生成できます。ただし、各行に生成できる順序番号は 1 つのみです。つまり、1 つの文で NEXTVAL が繰り返し参照されると、最初の参照によって次の番号が生成され、その文のそれ以降のすべての参照は同じ番号を戻します。

一度生成された順序番号は、その番号を生成したセッションに対してのみ使用できます。トランザクションのコミットまたはロールバックとは関係なく、ORDER_SEQ.NEXTVAL を参照している他のユーザーは、一意の値を取得します。2 人のユーザーが同時に同じ順序にアクセスしている場合、順序番号は 2 人以外のユーザーによっても生成される可能性があるため、各ユーザーが受け取る順序番号は INCREMENT で指定された値以上に離れている可能性があります。

CURRVAL を使用した順序番号の使用 セッションの現行の順序値を使用または参照するには、*seq_name*.CURRVAL を参照します。*seq_name*.NEXTVAL が（現行のトランザクションまたは前トランザクション内の）現行のユーザー・セッションで参照されている場合にのみ CURRVAL を使用できます。同じ文で何度でも、必要なだけ CURRVAL を参照できます。NEXTVAL が参照されるまで、次の順序番号は生成されません。前述の例を続けて、注文に対して明細項目を挿入することによって、顧客の注文の処理を完了します。

```
INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 20321, 3);

INSERT INTO Line_items_tab (Orderno, Partno, Quantity)
    VALUES (Order_seq.CURRVAL, 29374, 1);
```

前の項で指定された INSERT 文が新しい順序番号 347 を生成したと想定すると、この項の文によって挿入された行は、ともに順序番号 347 を持つ行を挿入します。

NEXTVAL および CURRVAL の使用および制限 CURRVAL および NEXTVAL は、次の場所で使用できます。

- INSERT 文の VALUES 句
- SELECT 構文のリスト
- UPDATE 文の SET 句

CURRVAL および NEXTVAL は、次の位置で使用できません。

- 副問合せ
- ビューの問合せまたはスナップショットの問合せ
- DISTINCT 演算子を使用した SELECT 文
- GROUP BY 句または ORDER BY 句を使用した SELECT 文
- 集合演算子 UNION、INTERSECT または MINUS によって別の SELECT 文と結合されている SELECT 文
- SELECT 文の WHERE 句
- CREATE TABLE 文または ALTER TABLE 文における列の DEFAULT 値
- CHECK 制約の条件

順序番号のキャッシュ

順序番号は、システム・グローバル領域（SGA）内の順序キャッシュに保持できます。順序番号は、ディスクから読み込むより、順序キャッシュからの方が高速にアクセスできます。

順序キャッシュはいくつかのエントリで構成されます。各エントリは、単一の順序に対して数多くの順序番号を保持できます。

すべての順序番号に高速アクセスするには、次のガイドラインに従ってください。

- 順序キャッシュが、アプリケーションで同時に使用されるすべての順序を保持できることを確認してください。
- 順序キャッシュに保持する各順序値の数を増やしてください。

順序キャッシュ内のエントリ数 アプリケーションが順序キャッシュ内の順序にアクセスすると、順序番号は高速に読み込まれます。ただし、アプリケーションが順序キャッシュ内にない順序にアクセスすると、順序をディスクからキャッシュに読み込んでから順序番号を使用する必要があります。

アプリケーションが同時に数多くの順序を使用する場合、順序キャッシュがすべての順序を保持できるほど十分に大きくない場合があります。この場合、順序番号にアクセスすると、ディスクの読込みが頻繁に必要となる可能性があります。すべての順序に高速にアクセスするには、キャッシュに十分なエントリを用意して、アプリケーションが同時に使用するすべての順序を保持できるようにしてください。

各順序キャッシュ・エントリ内の値の数 順序が順序キャッシュに読み込まれるときに、順序値が作成され、キャッシュ・エントリに格納されます。これらの値には高速にアクセスできます。キャッシュに格納される順序値の数は、CREATE SEQUENCE 文の CACHE パラメータによって決まります。このパラメータのデフォルト値は 20 です。

次の CREATE SEQUENCE 文は、SEQUENCE キャッシュに 50 個の順序値が格納されるように、SEQ2 順序を作成します。

```
CREATE SEQUENCE Seq2
    CACHE 50;
```

SEQ2 の最初の 50 個の値をキャッシュから読み込むことができます。51 番目の値がアクセスされると、次の 50 個の値がディスクから読み込まれます。

CACHE に選択する値を大きくすることによって、ディスクから順序キャッシュに読み込まずにアクセスできる順序番号の数が増加します。ただし、インスタンス障害が発生すると、キャッシュ内のすべての順序値は失われます。また、エクスポートの実行中にトランザクションが順序番号にアクセスし続けた場合、エクスポートおよびインポート後に、キャッシュされていた順序番号のスキップが発生することがあります。

CREATE SEQUENCE 文で NOCACHE オプションを使用すると、順序値は順序キャッシュに格納されません。この場合、順序にアクセスするたびに、ディスクの読込みが必要になります。このようなディスクの読込みは、順序へのアクセスを遅くします。次の CREATE SEQUENCE 文は、順序値がキャッシュに格納されないように、SEQ3 順序を作成します。

```
CREATE SEQUENCE Seq3
    NOCACHE;
```

順序の使用に必要な権限

順序を使用するには、自スキーマにその順序が含まれているか、または別のユーザーの順序に対する SELECT オブジェクト権限が付与されている必要があります。

順序の削除

順序を削除するには、SQL コマンド DROP SEQUENCE を使用します。たとえば、次の文は、ORDER_SEQ 順序を削除します。

```
DROP SEQUENCE Order_seq;
```

順序を削除すると、その定義がデータ・ディクショナリから削除されます。シノニムはそのまま残りますが、参照されるとエラーを戻します。

順序の削除に必要な権限

自スキーマにある順序はどれでも削除できます。別のスキーマの順序を削除するには、DROP ANY SEQUENCE システム権限が必要です。

シノニムの管理

シノニムは、表、ビュー、スナップショット、順序、プロシージャ、ファンクション、パッケージまたはオブジェクト型の別名です。シノニムを使用すると、スキーマ修飾子を使用せずに他のスキーマからオブジェクトを参照できます。後述の項では、SQL コマンドを使用して、シノニムを作成、使用および削除する方法について説明します。

シノニムの作成

シノニムを作成するには、SQL コマンド CREATE SYNONYM を使用します。次の文は、JWARD のスキーマに含まれる EMP_TAB 表のパブリック・シノニム PUBLIC_EMP を作成します。

```
CREATE PUBLIC SYNONYM Public_emp FOR jward.Emp_tab;
```

シノニムの作成に必要な権限

自スキーマにビューを作成するには、CREATE SYNONYM システム権限が必要です。また、別のユーザーのスキーマにビューを作成するには、CREATE ANY SYNONYM システム権限が必要です。パブリック・シノニムを作成するには、CREATE PUBLIC SYNONYM システム権限が必要です。

DML 文でのシノニムの使用

シノニムの元となるオブジェクトを参照する場合と同じ方法で、シノニムを DML 文で参照できます。たとえば、シノニム EMP_TAB が、表またはビューを参照する場合、次の文は有効です。

```
INSERT INTO Emp_tab (Empno, Ename, Job)
    VALUES (Emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

シノニム FIRE_EMP がスタンドアロン・プロシージャまたはパッケージ・プロシージャを参照する場合、このシノニムは、次のコマンドを使用して SQL*Plus または Enterprise Manager で実行できます。

```
EXECUTE Fire_emp(7344);
```

GRANT 文および REVOKE 文でシノニムを使用することもできますが、別の DML 文では使用できません。

シノニムの使用に必要な権限

基礎となるオブジェクトにアクセスするために必要な権限を持っていると想定すると、その権限が、使用可能なロールから明示的に付与されたか、PUBLIC から付与されたかに関係なく、自スキーマに含まれる任意のプライベート・シノニムまたは任意のパブリック・シノニムを使用できます。また、プライベート・シノニムに対して必要なオブジェクト権限が付与されている場合、別のスキーマに含まれるそのプライベート・シノニムを参照することもできます。別ユーザーのシノニムは、付与されたオブジェクト権限を使用してのみ参照できます。たとえば、JWARD.EMP_TAB シノニムの SELECT 権限を持っている場合、JWARD.EMP_TAB シノニムを問い合せることはできますが、JWARD.EMP_TAB のシノニムを使用して行を挿入することはできません。

シノニムの削除

シノニムを削除するには、SQL コマンド DROP SYNONYM を使用します。プライベート・シノニムを削除する場合、PUBLIC キーワードは指定しないでください。ただし、パブリック・シノニムを削除する場合は、PUBLIC キーワードを指定します。次の文は、EMP_TAB という名前のプライベート・シノニムを削除します。

```
DROP SYNONYM Emp_tab;
```

次の文は、パブリック・シノニム PUBLIC_EMP を削除します。

```
DROP PUBLIC SYNONYM Public_emp;
```

シノニムが削除されると、その定義がデータ・ディクショナリから削除されます。削除されたシノニムを参照するすべてのオブジェクト（ビュー、プロシージャなど）はそのまま残りますが、無効になります。

シノニムの削除に必要な権限

自スキーマにあるプライベート・シノニムは自由に削除できます。別のユーザーのスキーマのプライベート・シノニムを削除するには、DROP ANY SYNONYM システム権限が必要です。パブリック・シノニムを削除するには、DROP PUBLIC SYNONYM システム権限が必要です。

1 回の操作による複数の表およびビューの作成

SQL コマンド CREATE SCHEMA を使用すると、1 回の操作で、複数の表およびビューの作成および複数の権限の付与が可能です。1 回の操作で複数の表およびビューの作成および権限付与を確実に行うには、CREATE SCHEMA コマンドが有効です。個々の表またはビューの作成に失敗した場合、あるいは権限付与に失敗した場合は、文全体がロールバックされるため、オブジェクトの作成または権限の付与は行われません。

たとえば、次の文は、2 つの表とその 2 つの表のデータを結合するビューを作成します。

```
CREATE SCHEMA AUTHORIZATION scott
    CREATE VIEW Sales_staff AS
        SELECT Empno, Ename, Sal, Comm
        FROM Emp_tab
        WHERE Deptno = 30  WITH CHECK OPTION CONSTRAINT
               Sales_staff_cnst

CREATE TABLE Dept_tab (
    Deptno      NUMBER(3) PRIMARY KEY,
    Dname       VARCHAR2(15),
    Loc         VARCHAR2(25))
CREATE TABLE Emp_tab (
    Empno       NUMBER(5) PRIMARY KEY,
```

```
    Ename       VARCHAR2(15) NOT NULL,
    Job         VARCHAR2(10),
    Mgr         NUMBER(5),
    Hiredate    DATE DEFAULT (sysdate),
    Sal         NUMBER(7,2),
    Comm        NUMBER(7,2),
    Deptno      NUMBER(3) NOT NULL
        CONSTRAINT Dept_fkey REFERENCES Dept_tab(Deptno))

GRANT SELECT ON Sales_staff TO human_resources;
```

CREATE SCHEMA コマンドは、ANSI の CREATE TABLE コマンドおよび CREATE VIEW コマンドに対する Oracle の拡張機能（たとえば、STORAGE 句）をサポートしません。

複数のスキーマ・オブジェクトの作成に必要な権限

CREATE SCHEMA コマンドを使用して、複数の表のようなスキーマ・オブジェクトを作成するには、コマンド文内のそれぞれの操作に対する権限が必要です。

スキーマ・オブジェクトのネーミング

ビュー、シノニムおよびプロシージャを定義する際に、部分グローバル・オブジェクト名および完全グローバル・オブジェクト名をいつ使用するか判断する必要があります。データベース名は固定する必要があり、ネットワーク内で不必要にデータベースを移動しないようにしてください。

分散データベース・システムにおいて、各データベースは一意のグローバル名を持つ必要があります。グローバル名は、データベース名およびそのデータベースを含むネットワーク・ドメインで構成されます。データベース内の各スキーマ・オブジェクトは、スキーマ・オブジェクト名およびグローバル・データベース名で構成されるグローバル・オブジェクト名を持ちます。

Oracle では、データベース内でスキーマ・オブジェクト名の一意性が確保されるため、一意のグローバル・データベース名を割り当てることによって、すべてのデータベース間にわたってそのスキーマ・オブジェクト名の一意性を確保できます。データベース名を割り当てる責任はデータベース管理者にあるため、このような仕事についてはデータベース管理者と相談して調整する必要があります。

SQL 文における名前解決の規則

オブジェクト名の形式は次のとおりです。

[*schema*.]*name*[@*database*]

次に例を示します。

Emp_tab

```
Scott.Emp_tab
Scott.Emp_tab@Personnel
```

セッションは、ユーザーがデータベースにログインした時点で確立されます。オブジェクト名は、現行のユーザー・セッションを元に変換されます。現行のユーザーのユーザー名はデフォルトのスキーマであり、ユーザーが直接ログインしたデータベースが、デフォルトのデータベースです。

Oracle は、異なるクラスのオブジェクトに対して別々の名前空間を持っています。同じ名前空間のすべてのオブジェクトは異なる名前を持っている必要がありますが、異なる名前空間にある 2 つのオブジェクトは、同じ名前を持つことができます。表、ビュー、スナップショット、順序、シノニム、プロシージャ、ファンクションおよびパッケージは、同じ名前空間に存在します。トリガー、索引およびクラスタは、それぞれ別々の名前空間を持っています。たとえば、表、トリガーおよび索引のすべてに SCOTT.EMP_TAB という同じ名前を付けることができます。

オブジェクト名のコンテキストに基づいて、Oracle は名前をオブジェクトに変換するときに、適切な名前空間を検索します。次に例を示します。

```
DROP CLUSTER Test
```

Oracle はクラスタの名前空間内の TEST を調べます。

オブジェクト名を直接与えるのではなく、シノニムを使用してオブジェクトを参照することもできます。プライベート・シノニムの名前には、普通のオブジェクト名と同じ構文を使用します。パブリック・シノニムはスキーマ PUBLIC 内に暗黙的に存在しますが、ユーザーはスキーマ PUBLIC によって明示的にシノニムを修飾できません。

シノニムを使用して参照できるのは、表と同じ名前空間内のオブジェクトのみです。名前がシノニムである可能性があるため、表の名前空間内のオブジェクトを必要とするコンテキスト内の名前を解決する場合は、次の規則を使用します。

1. 表の名前空間の名前を調べます。
2. 名前がシノニムではないオブジェクトに変換される場合、それ以上の作業は必要ありません。
3. 名前がプライベート・シノニムに変換された場合、その名前をシノニムの定義で置き換えて、手順 1 に戻ります。
4. 名前がスキーマで修飾されていた場合、エラーが戻されます。そうでない場合、名前がパブリック・シノニムであるかどうかを確認します。
5. 名前がパブリック・シノニムでない場合、エラーが戻されます。そうでない場合、名前をパブリック・シノニムの定義で置き換えて、手順 1 に戻ります。

分散データベースにおいてグローバル・オブジェクト名が使用されると（明示的に、またはシノニム内で間接的に）、ローカルの Oracle セッションはローカルで必要な分だけその参照を変換します（たとえば、シノニムをリモート表のグローバル・オブジェクト名に変換しま

す)。部分的に変換された文がリモート・データベースに転送された後で、リモート Oracle セッションは、前述のとおり、オブジェクトの変換を完了します。

> **参照：** 分散データベースにおける名前解決の詳細は、『Oracle9*i* データベース概要』を参照してください。

スキーマ・オブジェクトの名前の変更

必要に応じて、2 つの違った方法を使用してスキーマ・オブジェクトの名前を変更できます。オブジェクトを削除して再作成するか、または SQL コマンド RENAME を使用してオブジェクトの名前を変更します。

> **注意：** オブジェクトを削除して再作成する場合、オブジェクトが削除されると、そのオブジェクトに対するすべての権限付与が失われます。オブジェクトを再作成するときに、再度権限を付与してください。

表、ビュー、順序または表のプライベート・シノニムの名前を変更するために RENAME コマンドを使用する場合、オブジェクトに対して実施された権限付与は、新しい名前に引き継がれます。たとえば、次の文は、SALES_STAFF ビューの名前を変更します。

```
RENAME Sales_staff TO Dept_30;
```

ストアド PL/SQL プログラム・ユニット、パブリック・シノニム、索引またはクラスタは名前を変更できません。そのようなオブジェクトの名前を変更するには、削除してから再作成します。

スキーマ・オブジェクトの名前を変更すると、次のような影響があります。

- 名前が変更されたオブジェクトに依存しているすべてのビューおよび PL/SQL プログラム・ユニットは無効になります（次に使用する前に再コンパイルが必要です)。
- 名前が変更されたオブジェクトに対するすべてのシノニムは、使用したときにエラーを戻します。

オブジェクトの名前の変更に必要な権限

オブジェクトの名前を変更するには、そのオブジェクトの所有者である必要があります。

異なるスキーマへの切替え

次の文は、セッションの現在のスキーマを、文に指定されているスキーマ名に設定します。

```
ALTER SESSION SET CURRENT_SCHEMA = <schema name>
```

この後に続く SQL 文では、スキーマ修飾子が指定されていない場合、このスキーマ名をスキーマ修飾子として使用します。このセッションには、現行のユーザーの権限のみ付与されており、前述の ALTER SESSION 文による追加権限はまだ付与されていないことに注意してください。

次に例を示します。

```
CONNECT scott/tiger
ALTER SESSION SET CURRENT_SCHEMA = joe;
SELECT * FROM emp_tab;
```

emp_tab にはスキーマ修飾子が指定されていないため、この表の名前はスキーマ joe によって変換されます。ただし、表 joe.emp_tab に対する選択権限が scott にない場合、scott は SELECT 文を実行できません。

スキーマ・オブジェクトに関する情報のリスト

データ・ディクショナリは、スキーマ・オブジェクトに関する情報を提供する多くのビューを提供します。次に、スキーマ・オブジェクトに関係するビューを示します。

- ALL_OBJECTS、USER_OBJECTS
- ALL_CATALOG、USER_CATALOG
- ALL_TABLES、USER_TABLES
- ALL_TAB_COLUMNS、USER_TAB_COLUMNS
- ALL_TAB_COMMENTS、USER_TAB_COMMENTS
- ALL_COL_COMMENTS、USER_COL_COMMENTS
- ALL_VIEWS、USER_VIEWS
- ALL_MVIEWS、USER_MVIEWS
- ALL_INDEXES、USER_INDEXES
- ALL_IND_COLUMNS、USER_IND_COLUMNS
- USER_CLUSTERS
- USER_CLU_COLUMNS
- ALL_SEQUENCES、USER_SEQUENCES
- ALL_SYNONYMS、USER_SYNONYMS
- ALL_DEPENDENCIES、USER_DEPENDENCIES

例 1: タイプ別スキーマ・オブジェクトのリスト 次の問合せは、問合せを発行しているユーザーによって所有されるすべてのオブジェクトをリストします。

```
SELECT Object_name, Object_type FROM User_objects;
```

前述の問合せが戻す結果は、次のようになります。

```
OBJECT_NAME                    OBJECT_TYPE
------------------------------ -------------------
EMP_DEPT                       CLUSTER
EMP_TAB                        TABLE
DEPT_TAB                       TABLE
EMP_DEPT_INDEX                 INDEX
PUBLIC_EMP                     SYNONYM
EMP_MGR                        VIEW
```

例 2: 列情報のリスト _COLUMNS 接尾辞で終わるビューの 1 つを使用して、名前、データ型、長さ、精度、スケール、デフォルト・データ値などの列情報をリストできます。たとえば、次の問合せは、EMP_TAB 表および DEPT_TAB 表のすべてのデフォルト列値をリストします。

```
SELECT Table_name, Column_name, Data_default
    FROM User_tab_columns
    WHERE Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB';
```

この項のはじめに示した例文を仮定すると、次のようなリストが表示されます。

```
TABLE_NAME  COLUMN_NAME     DATA_DEFAULT
----------  --------------- --------------------
DEPT_TAB    DEPTNO
DEPT_TAB    DNAME
DEPT_TAB    LOC             ('NEW YORK')
EMP_TAB     EMPNO
EMP_TAB     ENAME
EMP_TAB     JOB
EMP_TAB     MGR
EMP_TAB     HIREDATE        (sysdate)
EMP_TAB     SAL
EMP_TAB     COMM
EMP_TAB     DEPTNO
```

> **注意：** すべての列がユーザー指定デフォルトを持っているわけではありません。これらの列に対して値が指定されていない行が挿入された場合、列の値として NULL が想定されます。

例 3: ビューおよびシノニムの依存性のリスト ビューまたはシノニムを作成する場合、ビューまたはシノニムはその基礎になるベース・オブジェクトを基にします。_DEPENDENCIES データ・ディクショナリ・ビューは、ビューに対する依存性を表示するために使用できます。また、_SYNONYMS データ・ディクショナリ・ビューは、シノニムの

ベース・オブジェクトをリストするために使用できます。たとえば、次の問合せは、ユーザー JWARD によって作成されたシノニムに対するベース・オブジェクトをリストします。

```
SELECT Table_owner, Table_name
    FROM All_synonyms
    WHERE Owner = 'JWARD';
```

この問合せが戻す情報は、次のようになります。

```
TABLE_OWNER                    TABLE_NAME
------------------------------ ------------
SCOTT                          DEPT_TAB
SCOTT                          EMP_TAB
```

3

データ型の選択

この章では、アプリケーションにおける Oracle 組込みデータ型の使用方法について説明します。内容は次のとおりです。

参照 :

- オブジェクト型、VARRAY、ネストした表などの複合型の詳細は、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。
- LOB データ型については、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。
- PL/SQL データ型については、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。多くの SQL データ型は、PL/SQL の場合と同じか、または類似しています。

Oracle 組込みデータ型の概要

データ型は、ある限定されたプロパティの集合を、表の列、あるいはプロシージャまたはファンクションの引数に使用できる値と対応付けます。このプロパティの集合によって、Oracle はあるデータ型の値を別のデータ型の値とは異なるものとして扱うようになります。たとえば、Oracle は、NUMBER データ型の値は加算できますが、RAW データ型の値は加算できません。

Oracle は、次の組込みデータ型を提供します。

- 文字データ型
  - CHAR
  - NCHAR
  - VARCHAR2 および VARCHAR
  - NVARCHAR2
  - CLOB
  - NCLOB
  - LONG
- NUMBER データ型
- 日時データ型
  - DATE
  - INTERVAL DAY TO SECOND
  - INTERVAL YEAR TO MONTH
  - TIMESTAMP
  - TIMESTAMP WITH TIME ZONE
  - TIMESTAMP WITH LOCAL TIME ZONE
- バイナリ・データ型
  - BLOB
  - BFILE
  - RAW
  - LONG RAW

これらの他に ROWID データ型があり、表の各行の単一のアドレスを表す ROWID 疑似列内の値に使用されます。

参照： データ型の一般的な説明は、『Oracle9*i* SQL リファレンス』を参照してください。LOB データ型の詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

表 3-1 に、各 Oracle 組込みデータ型の概要を示します。

表 3-1 Oracle 組込みデータ型の概要

データ型	説明	列の長さおよびデフォルト
CHAR (*size* [BYTE | CHAR])	長さが *size* バイトまたは *size* 文字の固定長文字データ。	表の各行で固定されます（後続空白が付きます）。最大サイズは 1 行当たり 2000 バイトで、デフォルト・サイズは 1 行当たり 1 バイトです。*size* を設定する前に、キャラクタ・セット（シングルバイトまたはマルチバイト）を考慮してください。
VARCHAR2 (*size* [BYTE | CHAR])	最大長が *size* バイトまたは *size* 文字の可変長文字データ。	各行で可変で、1 行当たり最大 4000 バイトです。*size* を設定する前に、キャラクタ・セット（シングルバイトまたはマルチバイト）を考慮してください。最大の *size* を指定する必要があります。
NCHAR (*size*)	長さが *size* 文字の固定長 Unicode 文字データ。	表の各行で固定されます（後続空白が付きます）。列の *size* 列は文字数です（バイト数は、AL16UTF16 エンコーディングの場合はこの数値の 2 倍、UTF8 エンコーディングの場合はこの数値の 3 倍です）。最大サイズは 1 行当たり 2000 バイトです。デフォルトは 1 文字です。
NVARCHAR2 (*size*)	長さが *size* 文字の可変長 Unicode 文字データ。最大の *size* を指定する必要があります。	各行で可変です。*size* 列は文字数です（最大バイト数は、AL16UTF16 エンコーディングの場合はこの数値の 2 倍、UTF8 エンコーディングの場合はこの数値の 3 倍です）。最大サイズは 1 行当たり 4000 バイトです。デフォルトは 1 文字です。
CLOB	シングルバイト文字データ。	最大 2^{32} - 1 バイト（4GB）です。
NCLOB	Unicode 各国語キャラクタ・セット（NCHAR）・データ。	最大 2^{32} - 1 バイト（4GB）です。

表 3-1 Oracle 組込みデータ型の概要（続き）

データ型	説明	列の長さおよびデフォルト
LONG	可変長文字データ。	表の各行で可変で、1 行当たり最大 2^{31} - 1 バイト（2GB）です。下位互換性のために提供されています。
NUMBER (*p*, *s*)	可変長数値データ。最大精度 *p* またはスケール *s*（あるいはその両方）は 38。	各行で可変です。指定された列に必要な最大値は、1 行当たり 21 バイトです。
DATE	固定長の日付（時間）データ。紀元前 4712 年 1 月 1 日～西暦 4712 年 12 月 31 日の範囲です。	表の各列で 7 バイトに固定されます。デフォルト形式は、NLS_DATE_FORMAT パラメータで指定された文字列（たとえば DD-MON-RR）です。
INTERVAL YEAR (*precision*) TO MONTH	年および月で表される期間。*precision* の値は、日付の YEAR フィールドの桁数を指定します。*precision* は 0 ～ 9 に指定でき、年のデフォルト値は 2 です。	5 バイトに固定されます。
INTERVAL DAY (*precision*) TO SECOND (*precision*)	日、時間、分および秒で表される期間。*precision* の値は、日付の DAY フィールドの桁数および SECOND フィールドの小数点以下の桁数を指定します。*precision* は 0 ～ 9 に指定でき、日のデフォルト値は 2、秒のデフォルト値は 6 です。	11 バイトに固定されます。
TIMESTAMP (*precision*)	小数点以下の秒数を含む、日時を表す値（正確な値は、オペレーティング・システム時計によって異なります）。 *precision* の値は、日付の SECOND フィールドの小数点以下の桁数を指定します。*precision* は 0 ～ 9 に指定でき、デフォルト値は 6 です。	*precision* に応じて、7 ～ 11 バイトの範囲で可変です。デフォルト値は、NLS_TIMESTAMP_FORMAT 初期化パラメータによって決まります。
TIMESTAMP (*precision*) WITH TIME ZONE	日時を表す値、および対応するタイムゾーン。タイムゾーンは、「-5:0」のように UTC からのオフセット、または「US/Pacific」のように地域名で設定できます。	13 バイトに固定されます。デフォルト値は、NLS_TIMESTAMP_TZ_FORMAT 初期化パラメータによって決まります。

表 3-1 Oracle 組込みデータ型の概要（続き）

データ型	説明	列の長さおよびデフォルト
TIMESTAMP (*precision*) WITH LOCAL TIME ZONE	TIMESTAMP WITH TIME ZONE と同じ。ただし、データは、格納時にはデータベースのタイムゾーンに正規化され、取得時にはクライアントのタイムゾーンに合わせて調整されます。	*precision* に応じて、7 ～ 11 バイトの範囲で可変です。デフォルト値は、NLS_TIMESTAMP_FORMAT 初期化パラメータによって決まります。
BLOB	非構造化バイナリ・データ。	最大 2^{32} - 1 バイト（4GB）です。
BFILE	外部ファイルに格納されるバイナリ・データ。	最大 2^{32} - 1 バイト（4GB）です。
RAW (*size*)	可変長のロー・バイナリ・データ。	各行で可変で、1 行当たり最大 2000 バイトです。最大の *size* を指定する必要があります。下位互換性のために提供されています。
LONG RAW	可変長のロー・バイナリ・データ。	表の各行で可変で、1 行当たり最大 2^{31} - 1 バイト（2GB）です。下位互換性のために提供されています。
ROWID	行のアドレスを表すバイナリ・データ。	表の各行で、10 バイト（拡張 ROWID の場合）または 6 バイト（制限 ROWID の場合）に固定されます。

文字データの表現

文字データ型は、文字数字データを格納するために使用します。

- CHAR データ型および NCHAR データ型は、固定長文字列を格納します。
- VARCHAR2 データ型および NVARCHAR2 データ型は、可変長文字列を格納します（VARCHAR データ型は、VARCHAR2 データ型と同義です）。
- NCHAR データ型および NVARCHAR2 データ型は、Unicode 文字データのみを格納します。
- CLOB データ型および NCLOB データ型は、最大 4GB のシングルバイト文字列およびマルチバイト文字列を格納します。

  参照：『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

- LONG データ型は、最大 2GB を含む可変長文字列を格納します。ただし、多くの制限があります。

参照：「LONG データ型および LONG RAW データ型に関する制限」を参照してください。

このデータ型は、既存のアプリケーションとの下位互換性を保つために提供されています。通常、新しいアプリケーションで大量の文字データを格納するには、CLOB データ型および NCLOB データ型を使用し、大量のバイナリ・データを格納するには、BLOB データ型および BFILE データ型を使用します。

表の文字数字データを格納する列にどのデータ型を使用するかを決めるには、次の相違点を考慮してください。

使用領域

- データをより効率的に格納するには、VARCHAR2 データ型を使用します。CHAR データ型ではすべての列値に空白を埋め込み、固定列長の最後まで後続空白を格納しますが、VARCHAR2 データ型では空白を追加しません。

比較セマンティクス

- 比較セマンティクスで ANSI 互換性が必要な場合（文字列の比較において後続空白が重要でない（比較対象としない）場合）は、CHAR データ型を使用します。文字列の比較において後続空白が重要である（比較対象とする）場合は、VARCHAR2 データ型を使用します。

将来の互換性

- CHAR データ型および VARCHAR2 データ型は、将来の完全なサポートが保証されています。現在、VARCHAR データ型は VARCHAR2 データ型と同義であるとみなされるため、将来も使用できる予定です。

CHAR データ、VARCHAR2 データおよび LONG データは、データベース・キャラクタ・セットから、NLS_LANGUAGE パラメータでユーザー・セッションに対して定義しているキャラクタ・セットに（これらのキャラクタ・セットが異なる場合）自動的に変換されます。

シングルバイト・キャラクタ・セットおよびマルチバイト・キャラクタ・セットの列の長さ

CHAR 列および VARCHAR2 列の長さは、バイト単位または文字単位で指定できます。

NCHAR 列および NVARCHAR2 列の長さは、必ず文字単位で指定し、1 つの文字が複数バイトで構成される場合に Unicode データを格納するために適した長さにします。

```
-- ID contains only single-byte data, up to 32 bytes.
ID VARCHAR2(32 BYTE);
-- NAME contains data in the database character set. The 32 characters might
-- be physically stored as more than 32 bytes, if the database character set allows
-- multibyte characters.
```

```
NAME VARCHAR2(32 CHAR);
-- BIOGRAPHY can represent 2000 characters in any Unicode-representable language.
-- The exact encoding depends on the national character set, but the column
-- can contain multibyte values even if the database character set is single-byte.
BIOGRAPHY NVARCHAR2(2000);
-- The representation of COMMENTS, as 2000 bytes or 2000 characters, depends
-- on the initialization parameter NLS_LENGTH_SEMANTICS.
COMMENTS VARCHAR2(2000);
```

マルチバイトのデータベース文字コード体系を使用する場合は、文字データ型の列を持つ表のために必要となる領域を慎重に検討してください。データベースの文字コード体系がシングルバイトである場合、1 つの列内のバイト数と文字数は同じです。マルチバイトの場合は、通常、このような対応はありません。1 つの文字は、個々のマルチバイトのコード体系と、シフトイン / シフトアウト制御コードの有無に応じて、1 バイトで構成される場合と、複数バイトで構成される場合があります。バッファのオーバーフローを回避するため、データベース・キャラクタ・セットと異なる Unicode エンコーディングを使用する可能性がある場合、データを NCHAR または NVARCHAR2 として指定します。

> **参照：** Oracle でのグローバリゼーション・サポートおよび各種文字コード体系の詳細は、次のマニュアルを参照してください。
>
> - 『Oracle9*i* Database グローバリゼーション・サポート・ガイド』
> - 『Oracle9*i* SQL リファレンス』

CHAR/VARCHAR2 と NCHAR/NVARCHAR2 間での暗黙的変換

以前のリリース（Oracle8*i* 以下）では、NCHAR および NVARCHAR2 型は、CHAR および VARCHAR2 型に交換できなかったため、使用が困難でした。たとえば、NVARCHAR2 リテラルには、N'*string_value*' のような特別な表記法が必要でした。今回のリリースでは、NCHAR および NVARCHAR2 を N 修飾子なしで指定したり、SQL 文および SQL 関数で CHAR 値および VARCHAR2 値と混在させることができます。

比較セマンティクス

Oracle は、空白埋め比較セマンティクスによって CHAR 値と NCHAR 値を比較します。2 つの値の長さが異なる場合、Oracle は、両方の長さが等しくなるまで、短い方の値の後ろに空白を追加します。続いて、Oracle は、異なる文字が見つかるところまで、2 つの値を 1 文字ごとに比較します。異なる文字がある場合、最初の異なる文字の大きい方の値が、より大きい値とみなされます。後続空白の文字数のみが異なる場合、2 つの値は等しいとみなされます。

Oracle は、非空白埋め比較セマンティクスによって VARCHAR2 値と NVARCHAR2 値を比較します。2 つの値は、同じ文字を持ち、長さが等しい場合にのみ、等しいとみなされます。Oracle は、異なる文字が見つかるところまで、2 つの値を 1 文字ずつ比較していきます。異なる文字がある場合、その位置の文字が大きい方が、より大きい値とみなされます。

Oracle では、CHAR 列に格納されている値には空白を埋めますが、VARCHAR2 列に格納された値にはこれを行わないため、VARCHAR2 列に格納されている値が占有する領域は、CHAR

列に格納された場合より少なくなる可能性があります。このため、VARCHAR2 列のある大規模な表の全表スキャンでは、CHAR 列に同じデータが格納されている表の全表スキャンより、読み込むデータ・ブロックが少なくなる可能性があります。アプリケーションが文字データを含む大規模な表に対して全表スキャンを行う場合、CHAR 列より VARCHAR2 列にデータを格納することによってパフォーマンスを改善できる可能性が高くなります。

ただし、使用するデータ型を決定するときに考慮する必要がある要因は、パフォーマンスのみではありません。Oracle は、各データ型の値を比較するために異なる方法を使用します。アプリケーションがこれらの比較セマンティクスの相違に敏感な場合、特定のデータ型を選択した方がよい場合があります。たとえば、文字値の比較で、Oracle に後続空白を無視させる場合は、これらの値は CHAR 列に格納する必要があります。

> **参照：** これらのデータ型の比較セマンティクスの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

数値データの表現

NUMBER データ型は、実数を固定小数点形式または浮動小数点形式で格納するために使用します。このデータ型を使用している数値は、様々な Oracle プラットフォーム間で移植性があることが保証され、最大桁数 38 桁の精度を提供します。$1 \times 10^{-130} \sim 9.99 \times 10^{125}$ の正と負の数値および 0（ゼロ）を、NUMBER 列に格納できます。

浮動小数点数を含む列を指定できます。次に例を示します。

```
distance NUMBER
```

または、次のように、精度（全部の桁数）およびスケール（小数点以下の桁数）を指定することもできます。

```
price NUMBER (8, 2)
```

必須ではありませんが、精度およびスケールを指定すると、不正な入力値を簡単に識別できます。精度が指定されないと、列は指定されたとおりに値を格納します。次の表に、異なるスケール位置を使用したデータの格納例を示します。

表 3-2 スケール位置の数値データ記憶域への影響

入力データ	指定	格納
7,456,123.89	NUMBER	7456123.89
7,456,123.89	NUMBER(9)	7456124
7,456,123.89	NUMBER(9,2)	7456123.89
7,456,123.89	NUMBER(9,1)	7456123.9
7,456,123.89	NUMBER(6)	（精度を超え、受け入れられません）
7,456,123.89	NUMBER(7,-2)	7456100

参照： NUMBER データ型の内部形式については、『Oracle9*i* データベース概要』を参照してください。

日時データの表現

DATE データ型は、時間の値（日時）を表に格納するために使用します。DATE データ型は、世紀、年、月、日、時間、分および秒を格納します。

TIMESTAMP データ型は、小数点以下を含む正確な値を格納するために使用します。たとえば、2 つのイベントのどちらが最初に発生するかを決定する必要があるアプリケーションでは、TIMESTAMP を使用します。ジョブの実行時間を指定する必要があるアプリケーションでは、DATE を使用します。

TIMESTAMP WITH TIME ZONE もタイムゾーン情報を格納できるため、特に、複数の地域にまたがって収集または調整する必要がある日付情報の記録に適しています。

TIMESTAMP WITH LOCAL TIME ZONE 値は、タイムゾーンが重要でない場合に使用します。たとえば、テレビ会議のスケジュールを立てるアプリケーションで使用できます。これらのテレビ会議では、各参加者がローカルのタイムゾーンで開始時間および終了時間を確認します。

TIMESTAMP WITH LOCAL TIME ZONE 型は、クライアント・システムのタイムゾーンを使用して日時を表示する必要がある 2 層アプリケーションに適しています。このデータ型は、Web サーバーを起動するアプリケーションなどの 3 層アプリケーションでは使用できません。これは、クライアントが Web サーバーである場合に、Web ブラウザに表示されるデータがブラウザのタイムゾーンではなく Web サーバーのタイムゾーンによってフォーマットされるためです。

INTERVAL DAY TO SECOND は、2 つの日時の正確な違いを表すために使用します。たとえば、この値を使用して、アラームを 36 時間後に設定したり、レースの開始から終了までの時間を記録することができます。2 年以上の長い時間を高精度で表すには、日の部分に大きな値を使用します。

INTERVAL YEAR TO MONTH は、年および月のみが重要な場合に、2 つの日時の違いを表すために使用します。たとえば、この値を使用して、アラームを 18 か月後の日付に設定したり、特定の日付から 6 か月経過したことを確認することができます。

Oracle は独自の内部形式を使用して日付を格納します。日付データは、それぞれ 7 バイトの固定長フィールドに格納され、それぞれのバイトは、世紀、年、月、日、時間、分および秒に対応します。

参照： Oracle の内部日付書式の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

日付書式

日付の入出力に対する Oracle の標準日付書式は、DD-MON-RR です。次に例を示します。

```
'13-NOV-19'
```

インスタンス全体を単位としたデフォルトの日付書式を変更するには、NLS_DATE_FORMAT パラメータを使用します。セッションでの書式を変更するには、ALTER SESSION 文を使用します。現行のデフォルト日付書式以外の日付を入力するには、書式マスク付き TO_DATE 関数を使用します。次に例を示します。

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

> **参照：** Oracle のユリウス暦は、他の日付アルゴリズムで生成されたユリウス暦と互換性がない場合があります。ユリウス暦日付については、『Oracle9*i* データベース概要』を参照してください。

DD-MON-YY のような日付書式を使用する場合は注意が必要です。YY は、現世紀の年を示します。たとえば、**31-DEC-92** は、1992 年ではなく **2092 年 12 月 31 日**を示します。現世紀以外の年を示す場合は、デフォルトで RR など、異なる書式マスクを使用してください。

例 : BC/AD 表記の日付の印刷

```
SQL> -- By default, the date is printed without any BC or AD qualifier.
SQL> select sysdate from dual;

SYSDATE
---------
24-JAN-02
SQL> -- Adding BC to the format string prints the date with BC or AD
SQL> -- as appropriate.
SQL> select to_char(sysdate, 'DD-MON-YYYY BC') from dual;

TO_CHAR(SYSDAT
--------------
24-JAN-2002 AD
```

2 つの DATA 値が同じ日を参照するかどうかのチェック

時刻データ付きの日付を比較する場合、時刻構成要素を無視するには、SQL 関数 TRUNC を使用します。

現在の日付および時刻の表示

システム日付および時刻に戻すには、SQL 関数 SYSDATE を使用します。

ヒント : SYSDATE への定数値の設定

FIXED_DATE 初期化パラメータを使用すると、SYSDATE を定数に設定できるため、テスト時に便利です。

時刻書式

時刻は 24 時間形式 HH24:MI:SS で格納されます。デフォルトでは、時刻部分に何も入力しない場合、または DATE 列が切り捨てられた場合、DATE 列内の時刻は 12:00:00 A.M.（真夜中）となります。時刻のみの入力では、日付部分は現在の月の最初の日が想定されます。日付の時刻部分を入力するには、次のように、時刻部分を示す書式マスク付きの TO_DATE 関数を使用します。

> **注意：** 次のようなデータ構造を設定しないと機能しない例もあります。
>
> ```
> CREATE TABLE Birthdays_tab (Bname VARCHAR2(20),Bday DATE)
> ```

```
INSERT INTO Birthdays_tab (bname, bday) VALUES
    ('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

日付算術の実行

Oracle では、日付算術に有効な多数の機能が提供されるため、1 日の秒数、1 か月の日数などを独自に計算する必要はありません。

有効な関数の例を次に示します。

- ADD_MONTHS
- SYSDATE
- SYSTIMESTAMP
- TRUNC

  DATE 値に適用した場合は、時刻部分を切り捨てるため、その日の一番始め（真夜中）を表します。2 つの DATE 値を切り捨て比較することによって、それらが同じ日を参照しているかどうかをチェックできます。また、GROUP BY 句とともに使用して、日計の作成もできます。
- 算術演算子（+ や - など）
- INTERVAL データ型

  データ型日付算術を実行する場合の定数を表すには、独自の計算をするのではなく、INTERVAL データ型を使用できます。たとえば、DATE 値に INTERVAL 定数を加算または減算したり、2 つの DATE 値を減算して結果を INTERVAL と比較することができます。
- 比較演算子（>、<、= 、BETWEEN など）

日時データ型間の比較

有効な関数の例を次に示します。

- EXTRACT
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- TO_DATE（およびその逆の TO_CHAR）
- TO_DSINTERVAL
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL

> **参照：** 各関数の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

タイムゾーンの操作

Oracle では、タイムゾーンなどの計算に有効な多数の関数が提供されます。たとえば、TO_DATE は、TIMESTAMP WITH TIME ZONE 型の値で動作しません。かわりに、TO_TIMESTAMP_TZ を使用する必要があります。

有効な関数の例を次に示します。

- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE
- EXTRACT
- FROM_TZ
- LOCALTIMESTAMP
- SESSIONTIMEZONE
- SYS_EXTRACT_UTC
- SYSTIMESTAMP
- TO_TIMESTAMP_TZ

> **参照：** 各関数の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

日時データ型のインポートおよびエクスポート

TIMESTAMP WITH TIME ZONE 値および TIMESTAMP WITH LOCAL TIME ZONE 値は常に正規化された書式で格納されるため、タイムゾーンのオフセットを考慮することなくエクスポート、インポートおよび比較ができます。DATE 値および TIMESTAMP 値は対応付けられたタイムゾーンを格納しないため、ソース・データベースとターゲット・データベース間のタイムゾーンの違いを考慮して調整する必要があります。

西暦 2000 年準拠の設定

西暦 2000 年（Y2K）準拠の要件を満たすには、アプリケーションが次の条件を満たす必要があります。

- 西暦 2000 年 1 月 1 日より前、その当日、およびその後も、日付情報をエラーなしで処理できること。これには、日付入力の受入れ、日付出力の生成、日付情報の格納、さらに日付または日付の一部に対する計算の実行が含まれます。
- 西暦 2000 年 1 月 1 日より前、その当日、およびその後も、21 世紀の到来による操作の変更がなく、マニュアルに記述されているとおりのサービスを提供できること。
- 2 桁日付の入力に対して、明確に定義された方法で世紀の不明確さを解決できること。
- 西暦 2000 年に発生するうるう年を、400 年規則に従って管理できること。

これらの条件は、英国規格協会（BSI）が DISC PD-2000-1「A Definition of Year 2000 Conformity Requirements（西暦 2000 年準拠要件の定義）」として設定した西暦 2000 年準拠要件のスーパーセットです。

次のすべてのシステム・レベルにおいて西暦 2000 年に準拠していることを検証できた場合にのみ、アプリケーションを 2000 年準拠として保証できます。

- ハードウェア
- データベース、トランザクション・プロセッサおよびオペレーティング・システムを含むシステム・ソフトウェア
- アプリケーション・ソフトウェア（サード・パーティから入手したものまたは自社開発したもの）

世紀および西暦 2000 年

Oracle は年データを世紀情報とともに格納します。たとえば、Oracle データベースでは、単に 96 または 01 などの値ではなく、1996 または 2001 という値が格納されます。DATE データ型では、内部的に年を常に 4 桁で格納しているため、データベースに内部的に格納される他のすべての日付も 4 桁の年を使用します。インポート、エクスポート、リカバリなどの Oracle ユーティリティでも、4 桁の年を正常に処理します。

Oracle RDBMS（Oracle7 以上）を使用し、日付値または日付時刻値に DATE データ型を使用しているアプリケーションでは、格納データおよび西暦 2000 年に関する問題はありません。

Oracle7 の DATE データ型では、日付および時刻データを年については 4 桁年、時刻については秒単位の精度で格納します（通常は 'YYYY:MM:DD:HH24:MI:SS'）。

ただし、アプリケーションの中には、年に関する前提事項（すべての年が 19*xx* の形を取るなど）を基に作成されているものもあります。アプリケーションからデータベースに 2 桁の年が渡された場合、世紀の決定のために Oracle で使用されるプロシージャが、プログラマの予期したものとは異なる可能性があります（3-18 ページの「アプリケーションにおける西暦 2000 年問題のトラブルシューティング」を参照）。したがって、西暦 2000 年に関して、使用するコードを検査およびテストする必要があります。

RR 日付書式の例

TO_DATE 関数および TO_CHAR 関数の RR 日付書式要素によって、データベース・サイトでは、2 桁の年に従ってデフォルトの世紀を複数の異なる値に設定できます。これによって、50 ～ 99 の年はデフォルトの 19*xx* 形式に、00 ～ 49 の年はデフォルトの 20*xx* の形式に設定されます。したがって、データが入力される時点の世紀に関係なく、RR 書式では、データベース内に格納される年は必ず次のようになります。

- 現在の年が世紀の後半（50 ～ 99）で、00 ～ 49 の 2 桁年が入力された場合、このデータは次世紀年として格納されます。たとえば、1996 年に 02 が入力された場合は、2002 が格納されます。
- 現在の年が世紀の後半（50 ～ 99）で、50 ～ 99 の 2 桁年が入力された場合、このデータは現世紀年として格納されます。たとえば、1996 年に 97 が入力された場合は、1997 が格納されます。
- 現在の年が世紀の前半（00 ～ 49）で、00 ～ 49 の 2 桁年が入力された場合、このデータは現世紀年として格納されます。たとえば、2001 年に 02 が入力された場合は、2002 が格納されます。
- 現在の年が世紀の前半（00 ～ 49）で、50 ～ 99 までの 2 桁年が入力された場合、このデータは前世紀年として格納されます。たとえば、2001 年に 97 が入力された場合は、1997 が格納されます。

RR 日付書式は、データベースの DATE データの挿入および更新に使用できます。データベースにすでに格納されているデータを検索するか、または問い合せる場合には、この書式は不要です。これは、Oracle では日付の YEAR コンポーネントは今までも常に 4 桁で格納されているためです。

RR 日付書式の例を次に示します。

```
INSERT INTO emp (empno, deptno,hiredate) VALUES
   (9999, 20, TO_DATE('01-jan-03', 'DD-MON-RR'));

 INSERT INTO emp (empno, deptno,hiredate) VALUES
    (8888, 20, TO_DATE('01-jan-67',  'DD-MON-RR'));

SELECT empno, deptno,
   TO_CHAR(hiredate, 'DD-MON-YYYY') hiredate
```

```
FROM emp;
```

これによって、次のデータが生成されます。

```
EMPNO          DEPTNO         HIREDATE
----------     ----------     -----------------
8888           20             01-JAN-1967
9999           20             01-JAN-2003
```

CC 日付書式の例

TO_CHAR 関数の CC 日付書式要素では、指定された日付の世紀を戻します。次に例を示します。

```
SELECT TO_CHAR(TO_DATE('01-JAN-2000','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
20

SELECT TO_CHAR(TO_DATE('01-JAN-2001','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
21
```

TO_CHAR 関数の CC 日付書式要素では、世紀の値を 4 桁で表される年の最初の 2 桁より 1 大きい値に設定します（たとえば、「1900」の場合は「20」と設定されます）。100 の倍数となる年については、これは真の世紀とはなりません。厳密には、「1900」の世紀は 20 世紀ではなく、19 世紀です。20 世紀は 1901 年から始まります。

次の操作によって、Common Era（CE、以前の AD）の日付について正しい世紀が算出されます。*userdate* が CE の日付で、そこから真の世紀を算出する場合は、次の式を使用します。

```
SELECT DECODE (TO_CHAR (Hiredate, 'YY'),
    '00', TO_CHAR (Hiredate - 366, 'CC'),
    TO_CHAR (Hiredate, 'CC'))  FROM Emp_tab;
```

この式は、次のような処理を行います。まず、年の下 2 桁を取得します。「00」の場合、Oracle 世紀でいうと 1 年多い年となるため、その 1 年前の年で日付を計算します（1 年前の年の Oracle 世紀が真の世紀となります）。「00」以外は、Oracle 世紀を使用します。

> **参照：** 日付書式コードの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

文字データ型への日付の格納

アプリケーションで日付値を CHAR または VARCHAR2 データ型に格納し、世紀情報が保持されない場合は、アプリケーションを変更して、このような日付が世紀変更の影響を受ける場合に正しく処理されるルーチンを含める必要があります。このためには、世紀情報を保持する文字列を変更するか、または多少の制約がありますが、文字列を日付として解析するときに RR 日付書式を使用できます。

新規アプリケーションを作成する場合、または文字列として格納される日付が西暦 2000 年対応になるようにアプリケーションを変更する場合は、Oracle DATE データ型を使用して日付を変換することをお薦めします。これができない場合は、言語および書式から独立し、かつ 4 桁年を処理できる形式で日付を格納します。たとえば、SYYYY/MM/DD を使用し、必要であれば時刻要素を HH24:MI:SS として使用します。この形式で格納される日付を表示するとき、または受け取るときは、正しい外部形式に変換する必要があるので注意してください。

書式「SYYYY/MM/DD HH24:MI:SS」には、次のようなメリットがあります。

- 月表記が数値であり言語から独立しています。
- 完全な 4 桁年が含まれているため、世紀が明確です。
- 時刻が完全に表されます。最も重要な要素が最初にあるため、キャラクタ・ベースのソート操作で日付が正しく処理されます。

S 書式要素によって、BC 日付に接頭辞「-」が付けられます。

日付設定の表示

次のビューを表示して設定を検証できます。

- V$NLS_DATABASE_PARAMETERS は、インスタンス全体のグローバリゼーション・サポート・パラメータがデフォルトか初期化パラメータ・ファイルに明示的に宣言されている値かを示します。
- NLS_SESSION_PARAMETERS は、ALTER SESSION によって変更された可能性のある現在のセッション値を示します。

書式モデルは、文字列で格納されている DATE データまたは NUMBER データの書式を記述する文字です。書式モデルを TO_CHAR 関数または TO_DATE 関数の引数として使用すると、次のいずれかを行うことができます。

- データベースから値が戻されるときに使用される書式の指定
- Oracle に対してデータベース格納を指定した値の書式の指定

書式によって、データベース内の値の内部表現が変更されることはありません。

タイムゾーンの地域およびタイムゾーンの略称を参照するには、V$TIMEZONE_NAMES ビューを問い合せます。

日付設定の変更

日付書式は、環境内で設定するか、またはデータベース全体のデフォルトとして設定できます。日付書式を環境内で設定する場合は、初期化パラメータの設定がオーバーライドされます。

NLS_DATE_FORMAT パラメータ設定を変更するには、次の手順に従います。

1. クライアント側を設定します（Windows NT レジストリや UNIX 環境変数など）。
2. ALTER SESSION SET NLS_DATE_FORMAT を使用してセッションを設定します。セッションの日付書式を変更するには、次の SQL コマンドを発行します。

   ```
   ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-RR'
   ```

3. init.ora ファイルの NLS_DATE_FORMAT パラメータを使用してサーバーを設定します。データベース全体のデフォルトの日付書式を変更するには、init.ora に次の設定を追加します。

   ```
   NLS_DATE_FORMAT = DD-MON-RR
   ```

NLS_DATE_FORMAT 設定は、前述の順序に依存します。そのため、クライアント / サーバー・アプリケーションの場合、NLS_DATE_FORMAT はサーバーとクライアントの両方で設定する必要があります。

注意： このパラメータをデータベース・レベルで変更すると、前述のように既存のすべての日付フィールドが変更されます。すべてのユーザーおよび現在実行中のすべてのアプリケーションが 1950 ～ 2049 年の日付を処理する場合を除いて、変更はセッション・レベルで行うことをお薦めします。

アプリケーションにおける西暦 2000 年問題のトラブルシューティング

この項では、西暦 2000 年対応に関するプログラミング上の問題のうち、よくあるものを説明します。このような問題は、データベース・エンジンによって西暦 2000 年が正しく処理されないことが原因であると思われる場合がありますが、次の例では、Oracle テクノロジを正しく利用していないことが原因です。

西暦 2000 年問題の例 : 短すぎる日付列

アプリケーションでは、ディスク領域の節約のために、日付の年を CHAR(2) または NUMBER(2) の列を使用して定義していることがあります。これは、20xx 日付と 19xx 日付が混在したときに予期しない結果を戻す可能性があります。この問題を解決するには、完全な 4 桁年を使用するようにアプリケーションを変更します。

西暦 2000 年問題の例 : 4 桁年と 2 桁年の混在

アプリケーションが 4 桁年を格納するように設計されていても、コードでは 4 桁年の行とともに 2 桁年の行も間違って格納できるようになっていることがあります。これによって、日付列に 1900 年より前の日付が含まれている場合、日付による問合せで予期しない結果が戻されることになります。この問題を解決するには、アプリケーションで 1900 年より前の日付を含む行を調べ、修正します。

西暦 2000 年問題の例 : 2 桁で格納された広範囲の年

アプリケーションで 1950 年より前または 2049 年より後の日付を処理するかどうか、および年を 2 桁で格納しているかどうかを調べます。両方の条件に一致する場合は、RR 書式は使用せず、2 桁年「YY」を 4 桁の「YYYY」に拡張し、データベースに 4 桁の数値を格納します。

西暦 2000 年問題の例 : 2000 年 2 月 29 日の処理

通常、次のエラーは発生しませんが、このエラーによって、NLS_DATE_FORMAT と Oracle の RR 書式マスクとの相互作用がよくわかります。次の文は構文としては正しいですが、論理エラーがあります。

```
SELECT TO_CHAR(TO_DATE(LAST_DAY('01-FEB-00'),'DD-MON-RR'),'MM/DD/RRRR')
FROM DUAL;
```

前述の問合せは 02/28/2000 を戻します。これは RR 書式要素の定義済動作とは矛盾していませんが、西暦 2000 年はうるう年であるため、正しくないことになります。

問題は、この操作ではデフォルトの NLS_DATE_FORMAT つまり DD-MON-YY を使用しているということです。NLS_DATE_FORMAT を DD-MON-RR に変更すると、同じ SELECT 文で 02/29/2000（正しい値）が戻されます。

Oracle サーバー・エンジンと同じように問合せを評価してみます。処理される最初の関数は最も内側の関数 LAST_DAY です。NLS_DATE_FORMAT が YY のため、1900 年を使用して式が評価され、2/28 を正しく戻します。次に、値 2/28 がもう 1 つ外側の関数に戻されます。これで、TO_DATE 関数および TO_CHAR 関数が RR 書式マスクを使用して値 02/28/00 を書式化し、結果を 02/28/2000 として表示します。

SELECT LAST_DAY('01-FEB-00') FROM DUAL が発行された場合、結果は NLS_DATE_FORMAT により異なります。YY の場合は、年が 1900 年として解析されるため、戻される LAST_DAY は 28-Feb-00 になります。RR の場合は、年が 2000 年として解析されるため、戻される LAST_DAY は 29-Feb-00 になります。西暦 1900 年はうるう年ではありませんが、西暦 2000 年はうるう年です。

西暦 2000 年問題の例 : DECODE 内での暗黙的な日付変換

DECODE 関数が使用されたときに、3 番目の引数が CHAR データ型か VARCHAR2 データ型、または NULL の場合、戻り値は VARCHAR2 データ型に変換されます。したがって、次の文は日付 31.12.1900 を挿入します。

```
INSERT INTO destination_table (date_column)
```

```
    SELECT DECODE('31.12.2000', '00000000', NULL,
        TO_DATE('31.12.2000','DD.MM.YYYY'))
    FROM DUAL;
```

次にもう 1 つの例を示します。

```
INSERT INTO destination_table (date_column)
    SELECT DECODE('01.11.1999', '00000000', NULL, sysdate+1000)
FROM DUAL;
```

これは、日付 04.10.1901 を挿入します。

この例では、DECODE 引数リストの 3 番目の引数が NULL 値であるため、Oracle はデフォルトの書式マスクを使用して、暗黙的に DATE 値を VARCHAR2 文字列に変換します。デフォルトは DD-MON-YY であるため、年の上 2 桁が失われます。

注意 : レコードを表に挿入すると、Oracle は暗黙的に現在の年の上 2 桁を使用して文字列を日付に変換します。年が適切に解析されるようにするには、「RR」または「YYYY」を使用して *NLS_DATE_FORMAT* を設定してください。

西暦 2000 年問題の例 : DATE 列に基づく表のパーティション化

パーティション・キー内の DATE データ型列を使用してパーティション表を作成する場合、日付範囲を指定するときに 4 桁の年を使用します。次に例を示します。

```
CREATE TABLE stock_xactions (stock_symbol CHAR(5),
    stock_series CHAR(1),
    num_shares NUMBER(10),
    price NUMBER(5,2),
    trade_date DATE)
    STORAGE (INITIAL 100K NEXT 50K) LOGGING
    PARTITION BY RANGE (trade_date)
      (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
    TABLESPACE ts0
        NOLOGGING,
        PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
    TABLESPACE ts1,
      PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
    TABLESPACE ts2);
```

西暦 2000 年問題の例 : 2 桁年を使用して定義されたビュー

Oracle のビューはセッションの状態によって異なります。特に、次のような 2 桁の年の述語は、ビュー内で使用できます。

```
WHERE col > '12-MAY-99'
```

4 桁の年の解析は、NLS_DATE_FORMAT の設定によって異なります。

地理座標データの表現

MDSYS.SDO_GEOMETRY データ型などの Oracle Spatial の機能を使用して、地理情報システム（GIS）または空間データを表すことができます。オブジェクト・リレーショナル・モデルまたはリレーショナル・モデルのいずれかを使用してデータベースにデータを格納し、PL/SQL パッケージのセットを使用してデータを操作および問い合せることができます。

詳細は、『Oracle Spatial ユーザーズ・ガイドおよびリファレンス』を参照してください。

イメージ、オーディオおよびビデオ・データの表現

イメージ、オーディオ、ビデオなどのマルチメディア・データを BLOB または BFILE としてデータベースに格納したり、Web またはその他のサーバーに外部的に格納する場合、*inter*Media を使用して、オブジェクト・リレーショナル・モデルまたはリレーショナル・モデルのいずれかでデータにアクセスし、一連のオブジェクト型を使用してデータを操作および問い合せることができます。

詳細は、『Oracle9*i* *inter*Media User's Guide and Reference』を参照してください。

検索可能なテキスト・データの表現

全文検索を行うには、低レベルのコードを作成するのではなく、Oracle Text（以前の ConText および *inter*Media Text）を使用します。Oracle Text を使用すると、検索データが特別な索引に格納され、演算子および PL/SQL パッケージを使用して検索データを問い合せることができます。これによって、表、ファイルまたは URL のデータを使用する独自の検索エンジンの作成、および検索ロジックとリレーショナル問合せの結合が簡単になります。この方法では、XPath 表記法を使用して XML データを検索することもできます。

詳細は、『Oracle Text アプリケーション開発者ガイド』を参照してください。

ラージ・データ型の表現

以前は、LONG データ型、RAW データ型および LONG RAW データ型を使用して、データベースでラージ・データ・オブジェクトを表しました。現行のアプリケーションでは、これらのデータに CLOB、BLOB、BFILE などの様々な LOB 型を使用することをお薦めします。

LOB データ型の詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

次の項では、古いデータ型から LOB 型にデータを移行する方法について説明します。LOB 型は、以前は LONG や VARCHAR2 などの他のデータ型が必要であった多くの場合に使用できます。

LONG データ型から LOB データ型への移行

LONG データ型は、使用可能メモリーに応じて、最大 2GB の情報を含む可変長文字データを格納できます。LONG 列には、VARCHAR2 列と同じ特性が多数あります。SELECT 構文のリスト、UPDATE 文の SET 句、INSERT 文の VALUES 句で使用できます。

LONG データ型は、既存のアプリケーションとの下位互換性を保つためにのみ使用することをお薦めします。新しいアプリケーションで大量の文字データを格納するには、CLOB データ型および NCLOB データ型を使用してください。通常、既存のアプリケーションを変更しなくても、表の LONG データを LOB 型に変更できます。LONG データ用の SQL、PL/SQL および OCI インタフェースは、LOB データにも同様に機能します。

参照 :

- CLOB データ型および NCLOB データ型の詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。
- OCI の各機能の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。
- ALTER TABLE コマンドの構文は、『Oracle9*i* SQL リファレンス』を参照してください。

LONG 列または LONG RAW 列から LOB データ型への変更

ALTER TABLE コマンドを使用して、列の基礎となるデータ型を LONG から CLOB に、または LONG RAW から BLOB に変更できます。次に例を示します。

```
ALTER TABLE employees MODIFY (resume BLOB) DISABLE STORAGE IN ROW;
ALTER TABLE newspaper MODIFY (article CLOB DEFAULT 'Has not been written yet');
```

この方法では、表に対するすべての制約およびトリガーが保たれます。すべての索引は再作成する必要があります。データ・カートリッジや *inter*Media アプリケーション用の索引のような LONG 列のドメイン索引は、列の型を変更する前にすべて削除する必要があります。

LONG 列または LONG RAW 列から LOB データ型への変更に関する制限

1. LOB は、クラスタ化表には使用できません。表がクラスタの一部である場合、その表の LONG 列または LONG RAW 列は LOB に変更できません。
2. レプリケートされた表またはマテリアライズド・ビューを作成された表で、LONG 列を LOB に変更する場合、レプリカを手動で修正する必要があります。
3. 列を LOB データ型に変更したとき、すべてのトリガーが保たれるわけではありません。LOB 列は UPDATE トリガーの列リストでは使用できません。たとえば、列の型を LOB に変更すると、次のトリガーは無効になり、再コンパイルできません。

```
CREATE TABLE t(changed_col LONG);
CREATE TRIGGER trig BEFORE UPDATE OF lobcol ON t ...;
```

LONG データ型および LONG RAW データ型を使用するアプリケーションから LOB への透過的アクセス

LONG データまたは LONG RAW データに SQL または PL/SQL の DML（INSERT、UPDATE、DELETE）文を使用するアプリケーションの場合、列を LOB データ型に変換した後も、これらの文は同様に動作します。様々なキャラクタ・タイプの入力パラメータおよび出力バッファを使用できます。これらの入力パラメータおよび出力バッファは対応する LOB データ型に変換され、出力型がすべての結果を十分保持できる大きさでない場合は切り捨てられます。たとえば、文字変数に CLOB、または RAW 変数に BLOB を選択できます。

キャラクタ・タイプを受け入れる次の SQL 関数は、CLOB データを受け入れたり、出力することができます。

||, CONCAT, INSTR, INSTRB, LENGTH, LENGTHB, LIKE, LOWER, LPAD, LTRIM, NLS_LOWER, NLS_UPPER, NVL, REPLACE, RPAD, RTRIM, SUBSTR, SUBSTRB, TRIM, UPPER

PL/SQL では、前述したすべての SQL 関数、比較演算子（>、=、< および !=）およびすべてのユーザー定義のプロシージャおよびファンクションが、CLOB データ型をパラメータまたは出力型として受け入れます。PL/SQL では、CLOB を文字変数に割り当てることが（その逆も）できます。

OCI コールを使用して LONG データをピース単位で挿入、更新またはフェッチするアプリケーションの場合、列を LOB データ型に変換した後も、これらのコールは同様に動作します。CLOB 列を SQLT_CHR、または BLOB 列を SQLT_BIN と定義すると、最初にロケータを選択しなくても、LOB データを直接 CHARACTER バッファまたは RAW バッファに選択できます。SQLT_LNG、SQLT_CHR、SQLT_BIN、SQLT_LBI などのデータ型を受け入れることによって、このような透過的アクセスを提供する OCI 関数を次に示します。

- OCIBindByName()
- OCIBindByPos()
- OCIDefineByPos()

LONG データ型および LONG RAW データ型に関する制限

LONG 列は、SQL 文の次の場所で参照できます。

- SELECT 構文のリスト
- UPDATE 文の SET 句
- INSERT 文の VALUES 句

LONG 値の使用には、次の制限があります。

- 表に複数の LONG 列を含めることはできません。
- LONG 属性を持つオブジェクト型は作成できません。

- LONG 列は WHERE 句または整合性制約に使用できません（NULL 制約および NOT NULL 制約に使用する場合は除きます）。
- LONG 列には索引を付けることができません。
- ストアド・ファンクションは LONG 値を戻せません。
- LONG データ型を使用して、PL/SQL プログラム・ユニットの変数または引数を宣言できます。ただし、SQL からプログラム・ユニットをコールすることはできません。
- 単一の SQL 文では、すべての LONG 列、更新された表およびロックされた表が、同一データベース上にある必要があります。
- LONG 列および LONG RAW 列は、分散 SQL 文で使用したり、レプリケートすることができません。
- 表に LONG 列と LOB 列の両方がある場合、同一 SQL 文の LONG 列および LOB 列の両方に 4000 バイト以上のデータをバインドできません。ただし、LONG 列または LOB 列のどちらかには 4000 バイト以上のデータをバインドできます。
- LONG 列を含む表は、自動セグメント領域管理の表領域に格納できません。

LONG 列は、SQL 文の次の特定の場所で使用できません。

- SELECT 文の GROUP BY 句、ORDER BY 句または CONNECT BY 句、または DISTINCT 演算子を含む SELECT 文内
- SELECT 文の UNIQUE 演算子
- CREATE CLUSTER 文の列リスト
- CREATE MATERIALIZED VIEW 文の CLUSTER 句
- SQL の組込み関数、式または条件
- GROUP BY 句を含む問合せの SELECT 構文のリスト
- 集合演算子（UNION、INTERSECT または MINUS）による複合型の副問合せまたは問合せの SELECT 構文のリスト
- CREATE TABLE ... AS SELECT 文の SELECT 構文のリスト
- ALTER TABLE ... MOVE 文
- INSERT 文にある副問合せの SELECT 構文のリスト

トリガーで LONG データ型を使用する方法を次に示します。

- トリガー内の SQL 文では LONG 列にデータを挿入できます。
- LONG 列のデータが制約データ型（CHAR や VARCHAR2 など）に変換できる場合、トリガー内の SQL 文は LONG 列を参照できます。
- トリガーの変数は、LONG データ型を使用して宣言できません。

- LONG 列では、:NEW および :OLD は使用できません。

OCI 関数を使用して、データベースから LONG 値の部分を取得できます。

> **参照：** 『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

> **注意：** LONG データまたは LONG RAW データを含む表を設計する場合は、LONG 列または LONG RAW 列と、その列に関連するデータを同じ表に格納するのではなく、LONG 列または LONG RAW 列をそれぞれ別の表に格納してください。このような設計によって、他の列のみにアクセスする SQL 文は、LONG または LONG RAW データを読み込む必要がなくなります。

LONG データ型の例

各記事のテキストも含めて、雑誌記事に関する情報を格納するために、2 つの表を作成します。次に例を示します。

```
CREATE TABLE Article_header
    (Id               NUMBER  PRIMARY KEY,
    Title             VARCHAR2(200),
    First_author      VARCHAR2(30),
    Journal           VARCHAR2(50),
    Pub_date          DATE);

CREATE TABLE article_text
    (Id               NUMBER
                      REFERENCES
                      Article_header,
    Text              LONG);
```

ARTICLE_TEXT 表は、各記事のテキストのみを格納します。ARTICLE_HEADER 表は、タイトル、主著者、雑誌、発行日をはじめ、記事に関するその他の情報を格納します。2 つの表は、各表の ID 列に対する参照整合性制約によって対応付けられます。

このように設計することによって、SQL 文は、記事のテキストを読み込まずにテキスト以外のデータを問い合せることができます。1991 年の 7 月に発行された雑誌『NATURE』のすべての主著者を選択する場合、ARTICLE_HEADER 表を問い合せる次の文を発行します。

```
SELECT First_author
    FROM Article_header
    WHERE Journal = 'NATURE'
        AND TO_CHAR(Pub_date, 'MM YYYY') = '07 1991';
```

各記事のテキストが主著者、雑誌、発行日と同じ表に格納されていた場合、Oracle はこの問合せを実行するためにテキストを読み込む必要があります。

RAW データ型および LONG RAW データ型の使用

注意： RAW データ型および LONG RAW データ型は、既存のアプリケーションとの下位互換性のために提供されています。新しいアプリケーションで大量のバイナリ・データを格納するには、BLOB データ型および BFILE データ型を使用してください。

参照： BLOB データ型および BFILE データ型の詳細は、『Oracle9*i* アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

RAW データ型および LONG RAW データ型は、Oracle によって解析されない（異なるシステム間でデータを移動するときに変換されない）データを格納します。これらのデータ型は、2 進データおよびバイト列のために用意されています。たとえば、LONG RAW は、図形データ、音声データ、文書データおよび 2 進データの配列に格納できます。解析方法は使用方法によって異なります。

Oracle Net、エクスポート・ユーティリティおよびインポート・ユーティリティは、RAW データまたは LONG RAW データの送信中は文字変換を行いません。Oracle が RAW データまたは LONG RAW データと CHAR データ間で自動的に変換を行う場合（INSERT 文で文字として RAW データを入力する場合など）、データは 1 つの 16 進文字として表現され、RAW データの 4 ビットごとのビット・パターンを表します。たとえば、ビット 11001011 の 1 バイトの RAW データは、「CB」として表示され、入力されます。

LONG RAW データには索引を付けることができませんが、RAW データには索引を付けることができます。

参照： LONG RAW データの制限の詳細は、3-23 ページの「LONG データ型および LONG RAW データ型に関する制限」を参照してください。

ROWID データ型による行の直接アドレッシング

Oracle 表のすべての行には、行の物理アドレスに対応する ROWID が割り当てられます。行が長すぎて単一のデータ・ブロックに格納できない場合、ROWID によって最初の行断片が識別されます。通常、ROWID は一意ですが、同じデータ・ブロック内に存在し、異なるクラスタ化表に存在する行の場合、異なる行が同じ ROWID を持つ可能性があります。

Oracle データベース内の各表は、ROWID という名前の疑似列を持っています。

参照： ROWID 疑似列および ROWID データ型については、『Oracle9*i* データベース概要』を参照してください。

拡張 ROWID 形式

Oracle サーバーでは、表パーティション、索引パーティション、クラスタなどの機能をサポートする拡張 *ROWID* 形式を使用しています。

拡張 ROWID は、次の情報を含みます。

- データ・オブジェクト（セグメント）識別子
- データ・ファイル識別子
- ブロック識別子
- 行識別子

データ・オブジェクト識別子は、Oracle がデータベース内のスキーマ・オブジェクト（非パーティション表やパーティションなど）に対して割り当てる識別番号です。次に例を示します。

```
SELECT DATA_OBJECT_ID FROM ALL_OBJECTS
       WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP_TAB';
```

この問合せによって、SCOTT スキーマ内の EMP_TAB 表のデータ・オブジェクト識別子が戻されます。

> **参照：** DBMS_ROWID パッケージのファンクションを使用してデータ・オブジェクト識別子を取得するその他の方法については、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

ROWID の異なるフォーム

Oracle マニュアルでは、ROWID という用語をコンテキストによって使い分けています。

ROWID 疑似列 表および非結合ビューはそれぞれ、ROWID と呼ばれる疑似列を持っています。次に例を示します。

```
CREATE TABLE T_tab (col1 Rowid);
INSERT INTO T_tab SELECT Rowid FROM Emp_tab WHERE Empno = 7499;
```

このコマンドは、問合せを満たす EMP_TAB 表の行の ROWID 疑似列を戻し、これを T1 表に挿入します。

内部 ROWID 内部 ROWID 形式は、サーバー・コードが行にアクセスするために必要な情報を保持する内部構造体です。制限付き内部 ROWID は、ほとんどのプラットフォームにおいて 6 バイトです。拡張 ROWID は、ほとんどのプラットフォームで 10 バイトです。

外部文字 ROWID 拡張 ROWID 疑似列は、18 文字の文字列形式（たとえば、AAAA8mAALAAAAQkAAA）でクライアントに戻されます。この文字列は、4 つの部分からなる形式、OOOOOOFFFBBBBBBRRR の拡張 ROWID のコンポーネントを base 64 にエンコードしたものを表します。

- OOOOOO: **データ・オブジェクト番号**は、データベース・セグメントを識別します（前述の例では、AAAA8m）。同じセグメントのスキーマ・オブジェクト（表のクラスタなど）は、同じデータ・オブジェクト番号を持ちます。
- FFF: 行を含む**データ・ファイル**（前述の例では、AAL ファイル）です。ファイル番号は 1 つのデータベース内で一意です。
- BBBBBB: 行を含む**データ・ブロック**（前述の例では、AAAAQk ブロック）です。ブロック番号は、表領域に対してではなく、それぞれのデータ・ファイルに関連します。したがって、同一のブロック番号を持つ 2 つの行が、同じ表領域の異なる 2 つのデータ・ファイルに存在する可能性があります。
- RRR: ブロックの**行**（前述の例では、AAA 行）です。

外部 ROWID はデコードする必要はありません。DBMS_ROWID パッケージのファンクションを使用して、拡張 ROWID の個々のコンポーネントを取得できます。

> **参照：** DBMS_ROWID パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

制限付き ROWID 疑似列は 18 文字の文字列形式で、16 進数でコード化された ROWID のデータ・ブロック、行およびデータ・ファイル・コンポーネントとともにクライアントに戻されます。

外部バイナリ ROWID 一部のクライアント・アプリケーションでは、バイナリ形式の ROWID を使用します。たとえば、OCI および一部のプリコンパイラ・アプリケーションでは、バインドまたは定義のコールで 3GL 構造体に対して ROWID をマップできます。バイナリ ROWID のサイズは、拡張および制限付き ROWID のサイズと同じです。拡張 ROWID に関する情報は、制限付き ROWID 構造体の未使用フィールドに入っています。

拡張バイナリ ROWID の形式は、C の構造体で表すと、次のようになります。

```
struct riddef {
    ub4    ridobjnum; /* data obj#--this field is
                         unused in restricted ROWIDs */
    ub2    ridfilenum;
    ub1    filler;
    ub4    ridblocknum;
    ub2    ridslotnum;
}
```

ROWID の移行および互換性の問題

下位互換性のために、ROWID の制限付き形式がサポートされています。これらの ROWID は Oracle7 データ内に存在しており、ROWID の拡張形式は、パーティション表のグローバル索引のみで必要となります。新しい表では、常に拡張 ROWID が得られます。

参照：『Oracle9*i* データベース管理者ガイド』を参照してください。

Oracle7 クライアントは、Oracle8 以上のデータベースにアクセスできます。同様に、Oracle8 以上のクライアントは、Oracle7 サーバーにアクセスできます。ここで、クライアントには、データベース・リンクを使用してサーバーにアクセスするリモート・データベースおよびサーバーにアクセスするクライアント 3GL または 4GL アプリケーションも含まれます。

参照： ROWID_TO_EXTENDED ファンクションの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』および『Oracle9*i* データベース移行ガイド』を参照してください。

Oracle9*i* クライアントから Oracle7 データベースへのアクセス 戻される ROWID 値は、常に、制限付き ROWID です。また、Oracle9*i* では、Oracle7 以下のサーバーに ROWID 値を戻す場合に、制限付き ROWID を使用します。

Oracle7 サーバーにアクセスする場合、次に示す ROWID の機能が働きます。

- ROWID を選択し、取得した値を WHERE 句で使用します。
- WHERE CURRENT OF カーソル操作を使用します。
- ROWID データ型または CHAR データ型のユーザー列に ROWID を格納します。
- 16 進エンコードを使用して、ROWID を解析します（推奨できない方法であるため、DBMS_ROWID 関数を使用してください）。

Oracle7 クライアントから Oracle9*i* データベースへのアクセス Oracle9*i* では、ROWID が拡張形式で戻されます。つまり、次の操作のみ行えます。

- ROWID を選択し、それを WHERE 句で使用します。
- WHERE CURRENT OF カーソル操作を使用します。
- ROWID を CHAR(18) データ型のユーザー列に格納します。

インポートおよびエクスポート 表の行に拡張 ROWID 値が含まれる場合、Oracle7 クライアントは、ROWID 列（ROWID 疑似列ではない）を持つ Oracle8 以上の表をインポートできません。

ANSI/ISO データ型、DB2 データ型および SQL/DS データ型

Oracle データベース内の表の列は、ANSI/ISO データ型、DB2 データ型および SQL/DS データ型を使用して定義できます。ただし、Oracle はそのようなデータ型を内部的に変換して、Oracle データ型にします。

表 3-3 に、ANSI データ型から Oracle データ型への変換を示します。ANSI/ISO データ型の NUMERIC、DECIMAL および DEC は、固定小数点数のみを指定できます。これらのデータ型に対して、s のデフォルトは 0 です。

表 3-3 ANSI データ型から Oracle データ型への変換

ANSI SQL データ型	Oracle データ型
CHARACTER (n)、CHAR (n)	CHAR (n)
NUMERIC (p,s)、DECIMAL (p,s)、DEC (p,s)	NUMBER (p,s)
INTEGER、INT、SMALLINT	NUMBER(38)
FLOAT (p)	FLOAT (p)
REAL	FLOAT (63)
DOUBLE PRECISION	FLOAT (126)
CHARACTER VARYING(n)、CHAR VARYING(n)	VARCHAR2 (n)
TIMESTAMP	TIMESTAMP
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE

IBM 社の製品 SQL/DS と DB2 の TIME、GRAPHIC、VARGRAPHIC および LONG VARGRAPHIC データ型には、対応する Oracle データ型がないため使用できません。

表 3-4 に、DB2 データ型と SQL/DS データ型の変換を示します。

表 3-4 SQL/DS および DB2 データ型の Oracle データ型への変換

DB2 データ型または SQL/DS データ型	Oracle データ型
CHARACTER (n)	CHAR (n)
VARCHAR (n)	VARCHAR2 (n)
LONG VARCHAR	LONG
DECIMAL (p,s)	NUMBER (p,s)
INTEGER、SMALLINT	NUMBER(38)
FLOAT (p)	FLOAT (p)

表 3-4 SQL/DS および DB2 データ型の Oracle データ型への変換

DB2 データ型または SQL/DS データ型	Oracle データ型
DATE	DATE
TIMESTAMP	TIMESTAMP

Oracle によるデータ型の変換

Oracle は、あるデータ型のデータを、別のデータ型のデータとして処理できる場合があります。通常、1 つの式に異なるデータ型の値を含めることはできません。ただし、次の関数を使用して、データを必要なデータ型に自動的に変換できます。

- TO_NUMBER()
- TO_CHAR()
- TO_NCHAR()
- TO_DATE()
- HEXTORAW()
- RAWTOHEX()
- RAWTONHEX()
- ROWIDTOCHAR()
- ROWIDTONCHAR()
- CHARTOROWID()
- TO_CLOB()
- TO_NCLOB()
- TO_BLOB()
- TO_RAW()

暗黙的なデータ型変換が、この後に説明する規則に従って行われます。

代入中のデータ変換

代入では、Oracle は次のような変換を自動的に行うことができます。

- VARCHAR2、NVARCHAR2、CHAR または NCHAR から NUMBER へ
- NUMBER から VARCHAR2 または NVARCHAR2 へ
- VARCHAR2、NVARCHAR2、CHAR または NCHAR から DATE へ

- DATE から VARCHAR2 または NVARCHAR2 へ
- VARCHAR2、NVARCHAR2、CHAR または NCHAR から ROWID へ
- ROWID から VARCHAR2 または NVARCHAR2 へ
- VARCHAR2、NVARCHAR2、CHAR、NCHAR または LONG から CLOB へ
- VARCHAR2、NVARCHAR2、CHAR、NCHAR または LONG から NCLOB へ
- CLOB から CHAR、NCHAR、VARCHAR2、NVARCHAR2 および LONG へ
- NCLOB から CHAR、NCHAR、VARCHAR2、NVARCHAR2 および LONG へ
- NVARCHAR2、NCHAR または BLOB から RAW へ
- RAW から BLOB へ
- VARCHAR2 または CHAR から HEX へ
- HEX から VARCHAR2 へ

Oracle が、代入する値のデータ型を代入先のデータ型に変換できる場合、代入は正常に行われます。

次の例では、次のように宣言されたパブリック変数および表を持つパッケージを想定しています。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE PACKAGE Test_Pack AS var1 CHAR(5); END;
CREATE TABLE Table1_tab (col1 NUMBER);
```

- `variable := expression`

  expression のデータ型は、*variable* のデータ型と同じか、そのデータ型に変換可能である必要があります。たとえば、次に示す代入で指定されるデータをストアド・プロシージャ本体内で自動的に変換します。

  ```
  VAR1 := 0;
  ```

- `INSERT INTO table VALUES (expression1, expression2, ...)`

  expression1 と *expression2*、および後に続くデータ型は、*table* 中の対応する列のデータ型と同じか、またはそのデータ型に変換可能である必要があります。たとえば、Oracle は、Table1_tab に対して次の INSERT 文で指定されるデータを自動的に変換します（前述の表定義を参照）。

  ```
  INSERT INTO Table1_tab VALUES ('19');
  ```

- `UPDATE` *table* `SET` *column* = *expression*

expression のデータ型は、*column* のデータ型と同じか、またはそのデータ型に変換可能である必要があります。たとえば、Table1_tab に対して発行された次の UPDATE 文で指定されるデータを自動的に変換します。

```
UPDATE Table1_tab SET col1 = '30';
```

- SELECT *column* INTO *variable* FROM *table*

  column のデータ型は、*variable* のデータ型と同じか、またはそのデータ型に変換可能である必要があります。たとえば、Oracle は表から選択されたデータを次の文の変数に代入する前に、自動的に変換します。

```
SELECT Col1 INTO Var1 FROM Table1_tab WHERE Col1 = 30;
```

式の評価中のデータ変換

式の評価では、Oracle は割当ての場合と同じ変換を自動的に行うことができます。式は、その内容に基づいて特定の型に変換されます。たとえば、算術演算子へのオペランドは NUMBER に変換され、文字列関数へのオペランドは VARCHAR2 に変換されます。

Oracle は、次のような変換を自動的に行うことができます。

- VARCHAR2 または CHAR から NUMBER へ
- VARCHAR2 または CHAR から DATE へ

文字列が有効な数値を表している場合にのみ、CHAR から NUMBER への変換が正常に行われます。文字列が NLS_DATE_FORMAT 初期化パラメータで指定されたセッションのデフォルト形式を満たす場合にのみ、CHAR から DATE への変換が正常に行われます。

一般的な式は次のとおりです。

- 次のような単純式

```
commission + '500'
```

- 次のようなブール式

```
bonus > salary / '10'
```

- 次のようなファンクション・コールおよびプロシージャ・コール

```
MOD (counter, '2')
```

- 次のような WHERE 句の条件

```
WHERE hiredate = TO_DATE('1997-01-01','yyyy-mm-dd')
```

- 次のような WHERE 句の条件

```
WHERE rowid = 'AAAAaoAATAAAADAAA'
```

通常、割当て変換の規則でカバーされないところでデータ型変換が必要になると、Oracle は、次の式評価の規則を使用します。

次のような割当ての場合、

```
variable := expression
```

Oracle は、式変換の規則を最初に使用して、*expression* を評価します。*expression* は単純な式でも複雑な式でもかまいません。評価が正常に行われると、結果として単一の値およびデータ型が戻されます。その後、Oracle は、割当て変換の規則を使用して、割当て先の変数にこの値を割り当てようとします。

動的に型指定されたデータの表現

いくつかの言語では、実行時にデータ型の変更を可能にしたり、プログラムに変数の型を確認させる機能があります。たとえば、C 言語には union キーワードおよび void * ポインタがあり、Java には typeof 演算子および Number などのラッパー型があります。Oracle9*i* には、すべての型のデータを保持できる変数および列を作成し、データ値をテストして基礎となる表現を参照できる機能が含まれています。これらの機能を使用すると、表の単一の列で、数値、文字列およびオブジェクトをそれぞれ別の行に表現できます。

SYS.ANYDATA 組込みデータ型を使用すると、すべてのスカラー型またはオブジェクト型の値を表現できます。この型はオブジェクト型で、すべての型のスカラー値を取り込み、値をスカラーまたはオブジェクトに戻すメソッドが含まれています。

同様に、SYS.ANYDATASET 組込みデータ型を使用すると、すべてのコレクション型の値を表現できます。

型に関する情報を処理および確認するには、SYS.ANYTYPE 組込みデータ型を DBMS_TYPES パッケージと組み合せて使用します。たとえば、次のプログラムは、表内の基となる型が異なるデータを表し、各列の基となる型を解析して、それぞれの値を適切に処理します。

```
-- The example below defines and executes a PL/SQL procedure that
-- uses methods built into SYS.ANYDATA to access information about
-- data stored in a SYS.ANYDATA table column.

DROP TYPE Employee FORCE;
DROP TABLE mytab;
CREATE OR REPLACE TYPE Employee AS OBJECT ( empno NUMBER,
  ename VARCHAR2(10) );
/
CREATE TABLE mytab ( id NUMBER, data SYS.ANYDATA );

INSERT INTO mytab VALUES (1, SYS.ANYDATA.ConvertNumber(5));
INSERT INTO mytab VALUES (2, SYS.ANYDATA.ConvertObject(Employee(5555, 'john')));
commit;
```

```
CREATE OR REPLACE procedure P IS
  CURSOR cur IS SELECT id, data FROM mytab;
  v_id           mytab.id%TYPE;
  v_data         mytab.data%TYPE;
  v_type         SYS.ANYTYPE;
  v_typecode     PLS_INTEGER;
  v_typename     VARCHAR2(60);
  v_dummy        PLS_INTEGER;
  v_n            NUMBER;
  v_employee     Employee;
  non_null_anytype_for_NUMBER exception;
  unknown_typename            exception;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

/* The typecode is a number that signifies what type is represented by v_data.
   GetType also produces a value of type SYS.AnyType with methods you can call
   to find precision and scale of a number, length of a string, and so on. */
    v_typecode := v_data.GetType ( v_type /* OUT */ );

/* Now we compare the typecode against constants from DBMS_TYPES to see what
   kind of data we have, and decide how to display it. */
    CASE v_typecode

      WHEN Dbms_Types.Typecode_NUMBER THEN
        IF v_type IS NOT NULL
-- This condition should never happen, but we check just in case.
          THEN RAISE non_null_anytype_for_NUMBER; END IF;
-- For each type, there is a Get method.
        v_dummy := v_data.GetNUMBER ( v_n /* OUT */ );
        Dbms_Output.Put_Line (
          To_Char(v_id) || ': NUMBER = ' || To_Char(v_n) );

      WHEN Dbms_Types.Typecode_Object THEN
        v_typename := v_data.GetTypeName();
-- An object type's name is qualified with the schema name.
        IF v_typename NOT IN ( 'SCOTT.EMPLOYEE' )
-- If we encounter any object type besides EMPLOYEE, raise an exception.
          THEN RAISE unknown_typename; END IF;
        v_dummy := v_data.GetObject ( v_employee /* OUT */  );
        Dbms_Output.Put_Line (
          To_Char(v_id) || ': user-defined type = ' || v_typename ||
            ' ( ' || v_employee.empno || ', ' || v_employee.ename || ' )' );
    END CASE;
```

```
  END LOOP;
  CLOSE cur;
EXCEPTION
  WHEN non_null_anytype_for_NUMBER THEN
    RAISE_Application_Error ( -20000,
      'Paradox: the return AnyType instance FROM GetType ' ||
      'should be NULL for all but user-defined types' );
  WHEN unknown_typename THEN
    RAISE_Application_Error ( -20000, 'Unknown user-defined type ' ||
      v_typename || ' - program written to handle only SCOTT.EMPLOYEE' );
END;
/
-- The query and the procedure P in the preceding code sample
-- produce output like the following:

SQL> SELECT t.data.gettypename() FROM mytab t;

T.DATA.GETTYPENAME()
------------------------------------------------------------------------------
SYS.NUMBER
SCOTT.EMPLOYEE

SQL> EXEC P;
1: NUMBER = 5
2: user-defined type = SCOTT.EMPLOYEE ( 5555, john )
```

OCIType、OCIAnyData および OCIAnyDataSet インタフェースを使用すると、OCI インタフェースを介して同じ機能にアクセスできます。

参照：

- DBMS_TYPES パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- ANYDATA データ型、ANYDATASET データ型および ANYTYPE データ型については、『Oracle9*i* アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。
- OCI インタフェースの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

XML データの表現

XML 形式のファイルとして格納されている情報がある場合、またはオブジェクト型を取り、XML として格納する場合、XMLType 組込み型を使用します。

XMLType 列にはデータが CLOB として格納されます。既存の CLOB、VARCHAR2 または任意のオブジェクト型を取り、XMLType コンストラクタをコールして XML オブジェクトに変換します。

XML オブジェクトがデータベース内に入ると、問合せを使用してオブジェクトを検索し（XML XPath 表記法を使用する）、データの全体または一部を抽出できます。

既存のリレーショナル・データから XML 出力を生成し、リレーショナル表および列にまたがって XML 文書を分割することもできます。DBMS_XMLQUERY パッケージ、DBMS_XMLGEN パッケージおよび DBMS_XMLSAVE パッケージ、および SYS_XMLGEN 関数および SYS_XMLAGG 関数を使用して、XML データをリレーショナル表に、またはリレーショナル表から転送できます。

参照：

- XML を使用した処理については、『Oracle9*i* XML Developer's Kit ガイド -XDK』を参照してください。
- XMLType 型、および DBMS_XMLQuery パッケージ、DBMS_XMLGEN パッケージおよび DBMS_XMLSave パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- SYS_XMLGEN 関数および SYS_XMLAGG 関数については、『Oracle9*i* SQL リファレンス』を参照してください。

4

制約によるデータ整合性のメンテナンス

この章では、データベースに対応付けたビジネス・ルール（業務規則）を施行する方法、および整合性制約を使用して表に無効な情報が入力されないようにする方法について説明します。内容は次のとおりです。

整合性制約の概要

整合性制約を使用して表のデータにビジネス・ルールを施行できます。ビジネス・ルールは、常に真または偽になる必要がある条件および関連を指定します。給与、従業員数、在庫調査などに関する方針は会社ごとに異なるため、それぞれのデータベース表に対して異なる規則を指定できます。

整合性制約が表に適用された場合、表のすべてのデータは対応する規則に準拠する必要があります。表のデータを変更する SQL 文を発行すると、Oracle は新しいデータが整合性制約を満たしていることを保証します。プログラムで確認する必要はありません。

整合性制約でビジネス・ルールを施行する場合

整合性制約を定義して規則を施行すると、アプリケーションにロジックを追加して同じ規則を施行するより信頼性が高くなります。Oracle は、アプリケーションよりも高速に表のデータが整合性制約に従っているかどうかを確認できます。

ビジネス・ルールに対する整合性制約の例

各従業員が有効な部門に所属しているかどうかを確認するには、まず部門表のすべての値が一意であるという規則を作成します。

```
ALTER TABLE Dept_tab
    ADD PRIMARY KEY (Deptno);
```

次に、従業員表に示されるすべての部門が、部門表内のいずれかの値と一致するという規則を作成します。

```
ALTER TABLE Emp_tab
    ADD FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno);
```

これ以降、表に新しい従業員レコードを追加すると、その部門番号が部門表に存在するかどうかが Oracle によって確認されます。

整合性制約を使用しないでこの規則を施行するには、部門表を問い合せるトリガーを使用して、新しい従業員の部門が有効かどうかをテストします。ただし、Oracle の SELECT では読込み一貫性が使用されるため、この問合せは、他のトランザクションからのコミットされていない変更を見落とす場合があります。このため、この方法は整合性制約より信頼性が低くなります。

アプリケーションでビジネス・ルールを施行する場合

挿入または更新を行う前に不正なデータをフィルタ除去できる場合は、アプリケーション・ロジックおよび整合性制約を介してビジネス・ルールを施行できます。これによって、ユーザーにすぐにフィードバックでき、データベースの負荷を削減できます。この方法は、表内のデータを確認しなくても、データの値が不正または範囲外であると判断できる場合に適しています。

制約で使用する索引の作成

使用可能なすべての一意キーおよび主キーには、対応する索引が必要です。これらの索引は、データベースに自動的に作成させるのではなく、手動で作成する必要があります。次のことに注意してください。

- 新しい索引を作成するのではなく、できるだけ既存の索引を使用するようにします。
- 一意キーおよび主キーは、一意索引と同様に非一意索引を使用できます。非一意索引の最初の数列のみでも使用できます。
- 多くても 1 つの一意キーまたは主キーが、それぞれの非一意索引を使用できます。
- 索引の列順序と制約は一致する必要がありません。
- 索引を削除する場合など、制約が索引を使用しているかどうかを確認する必要がある場合、一意キー制約または主キー制約が使用している索引のオブジェクト番号は、その制約の CDEF$.ENABLED に格納されます。カタログ・ビューには表示されません。

ほとんどの場合、外部キーには索引を付ける必要があります。データベース側ではこの作業が行われません。

NOT NULL 整合性制約を使用する場合

デフォルトでは、すべての列が NULL を含むことができます。NOT NULL 制約は、常に値が必要とされる表の列のみに定義します。

たとえば、一時的に新しい従業員のマネージャまたは入社日に値が入っていなくても、特に問題はありません。また、従業員の中には、コミッション（歩合）を受けていない人もいます。これらの列は、NOT NULL 整合性制約の対象としては適切ではありません。ただし、従業員の名前は最初から必要であるため、NOT NULL 整合性制約を使用してこの規則を施行できます。

NOT NULL 制約を他の整合性制約と組み合せて、表の特定の列に存在できる値をさらに制限することがよくあります。一意キーに必ず値を入力するには、NOT NULL と一意キー整合性制約を組み合せます。このようにデータ整合性規則を組み合せることで、新しい行のデータと既存の行のデータが競合する可能性はなくなります。

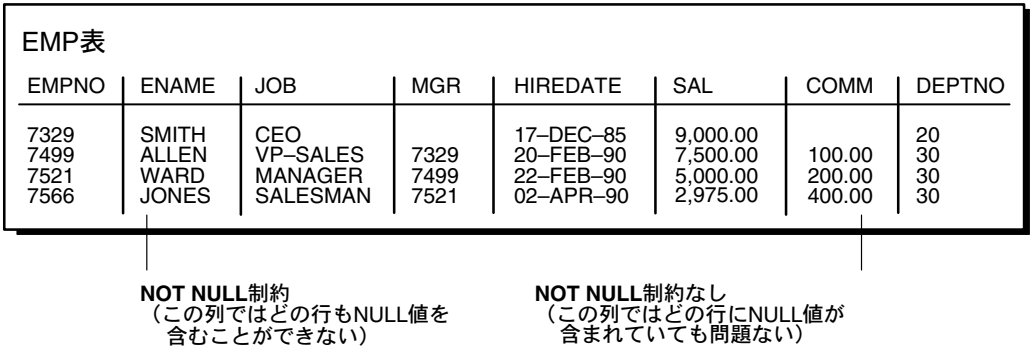
Oracle 索引は、すべてが NULL のキーは格納しません。したがって、表の索引のみをスキャンする場合、またはすべての行に索引付けが必要な操作を実行する場合は、1 つ以上の索引列に NOT NULL 制約を付けます。

参照： 4-10 ページの「親表と子表との関連の定義」を参照してください。

NOT NULL 制約は、次のとおり指定します。

```
ALTER TABLE emp MODIFY ename NOT NULL;
```

図 4-1 NOT NULL 整合性制約を使用した表



デフォルトの列値を使用する場合

デフォルト値は、代表的な値を含む列に割り当てます。たとえば、DEPT_TAB 表において、ほとんどの部門がある 1 つの所在地に位置する場合、この値（NEW YORK など）を LOC 列のデフォルト値に設定できます。

エントリを持たない列に 0（ゼロ）などのデフォルト値が適用される場合、エラーの回避に有効です。たとえば、デフォルト値を 0（ゼロ）に設定すると、次のテストは、

```
IF sal IS NOT NULL AND sal < 50000
```

次のように、より簡単な書式に変更できます。

```
IF sal < 50000
```

ビジネス・ルールに応じて、デフォルト値を 0（ゼロ）または偽に設定できます。また、デフォルト値を NULL のままにして、不明な値を示すこともできます。

また、表の一部の列を参照可能にするためのビューを作成する場合も、デフォルトは有効です。たとえば、ビューを介してユーザーが行を挿入できるようにしていることがあります。実表には、ビューの定義には含まれていない列で、各行を挿入するユーザーを記録する INSERTER 列がある場合もあります。ユーザー名を自動的に記録するには、USER 関数をコールするデフォルト値を定義します。

```
CREATE TABLE audit_trail
(
    value1   NUMBER,
    value2   VARCHAR2(32),
    inserter VARCHAR2(30) DEFAULT USER
);
```

参照： デフォルトの列値に関するもう 1 つの例は、2-4 ページの「表の作成」を参照してください。

デフォルトの列値の設定

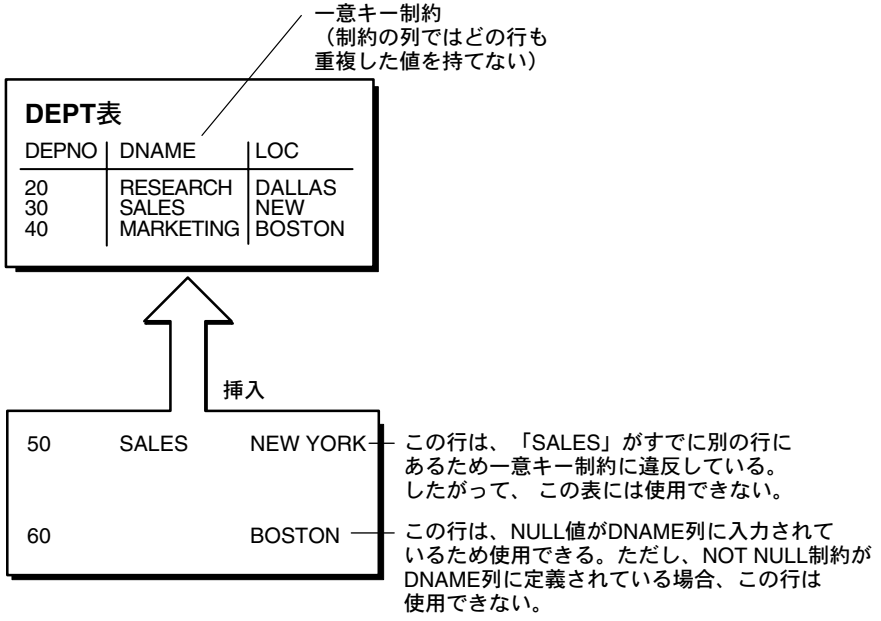
デフォルト値には、任意のリテラルまたはほとんどの式を含めることができます。これには、SYSDATE、SYS_CONTEXT、USER、USERENV および UID のコールも含まれます。デフォルト値には、順序、PL/SQL ファンクション、列、LEVEL、ROWNUM または PRIOR を参照する式を含めることはできません。デフォルトのリテラルまたは式のデータ型は、その列のデータ型と一致しているか、または変換できる必要があります。

デフォルト値が、SQL 関数の結果の場合もあります。たとえば、SYS_CONTEXT をコールすると、ユーザー名などの条件に応じて異なるデフォルト値が設定されます。デフォルト値として使用するには、SQL 関数のパラメータがすべてリテラルで、他の列の参照および他の関数のコールができない必要があります。

デフォルト値が列に対して明示的に定義されない場合、その列のデフォルトは暗黙的に NULL に設定されます。

INSERT 文内で、リテラル値のかわりに DEFAULT キーワードを使用できます。これによって、対応するデフォルト値が挿入されます。

図 4-2　一意キー制約を使用した表



表の主キーの選択

表ごとに、1 つの主キーを含めることができます。主キーを使用すると、表の各行を一意に識別でき、また、行の重複も回避できます。主キーを選択するときには、次のガイドラインに従ってください。

- 順序番号を含む列を使用します。これは、他のすべてのガイドラインを満たす簡単な方法です。
- 複合主キーの使用は最小限に抑えます。複合主キーは、使用できますが、他のすべての推奨項目を満たすわけではありません。たとえば、複合主キー値は、順序番号によって割り当てることができません。
- データ値が一意である列を選択します。これは、主キーの目的が表の各行を一意に識別することであるためです。
- データ値が変更されない列を選択します。主キー値は、表の行を識別するためにのみ使用されます。主キーには、他の目的のために使用されるデータを含めないでください。このため、主キー値は、ほとんど変更されないデータにする必要があります。

- NULL を含まない列を選択します。定義によって、主キー制約では、主キーを構成するいずれかの列に NULL が設定されている行を入力できません。
- 短い数値型の列を選択します。短い主キーは入力が簡単です。数値の主キーは、順序番号を使用して簡単に生成できます。

一意キー整合性制約を使用する場合

一意キーに指定する列は、慎重に選択してください。この制約の目的は、主キーの目的とは異なります。一意キー制約は、値の重複が許可されない列に適用できます。主キーは表の各行を一意に識別し、通常、一意であること以外は重要でない値を含みます。

注意： 一意キー制約では NULL 値を入力できますが、複合一意キー制約のある非 NULL 列内には同一の値を入力できません。

適切な一意キーの使用例は次のとおりです。

- 従業員の社会保障番号（主キーは従業員番号）
- トラックのナンバー・プレートの番号（主キーはトラック番号）
- 市外局番と電話番号の 2 列からなる顧客電話番号（主キーは顧客番号）
- 部門名と所在地（主キーは部門名）

パフォーマンスのため（データの整合性ではなく）のビューに対する制約

この章で説明する制約は、ビューではなく表に適用されます。

ビューにも制約を宣言できますが、その制約はデータの整合性を保持するには有効ではありません。これらの制約は、ビューを伴う問合せをリライトするために使用されます。これによって、マテリアライズド・ビューおよびその他のデータ・ウェアハウス機能を使用した場合のパフォーマンスが向上します。これらの制約は、常に DISABLE キーワードを使用して宣言されます。VALIDATE キーワードは使用できません。この場合、制約は施行されず、対応付けられた索引はありません。

参照： クエリー・リライト、マテリアライズド・ビューおよびビューに制約を宣言するパフォーマンス上の理由については、『Oracle9*i* データ・ウェアハウス・ガイド』を参照してください。

制約を使用した参照整合性の施行

2 つの表に 1 つ以上の共通の列が含まれる場合、Oracle は参照整合性制約を使用して 2 つの表の関連を規定します。親表（完全な列値の集合を持つ表）の列に、主キー制約または一意キー制約を定義します。子表の列（他の表の値を参照する値を含む表）には、外部 キー制約を定義します。

> **参照：** 関連によっては、4-10 ページの「親表と子表との関連の定義」に示すように、外部キーを含む整合性制約を追加するのが望ましい場合があります。

図 4-3 に、部門番号に定義された外部キーを示します。この外部キーは、この列の値がそれぞれ部門表の主キーの値と一致することを保証します。制約によって、間違った部門番号が従業員表に入力される可能性を回避できます。

外部キーは、複数の列で構成することもできます。このような**複合外部キー**は、正確に同じ構造（列の数とデータ型が同一）を持つ複合主キーまたは複合一意キーを参照する必要があります。複合主キーまたは複合一意キーには 32 列までという制限があるため、複合外部キーも最大 32 列に制限されます。

NULL および外部キー

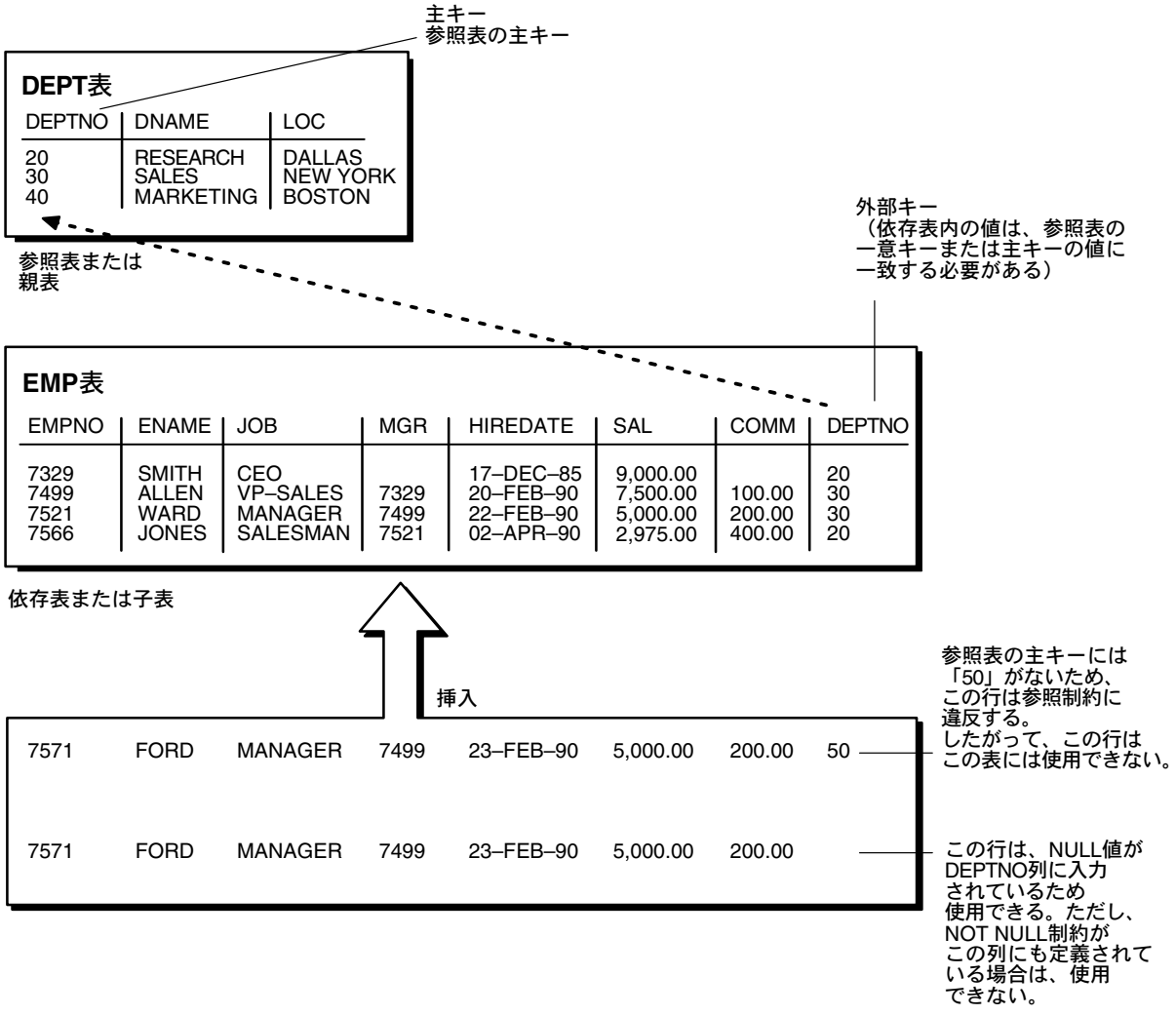
外部キーには、一致する主キーまたは一意キーがない場合でも、すべて NULL のキー値を使用できます。

- デフォルトでは（NOT NULL 句または CHECK 句を指定しない場合）、外部 キー制約は ANSI/ISO 規格の複合外部キーに対して「不一致」規則を施行します。
- 複合外部キーの NULL に対する「完全一致」規則（キーのすべての構成要素が NULL であるか、NULL 以外のものであることを要求する）を施行するには、すべての複合外部キーが NULL または非 NULL であることのみを許可する CHECK 制約を定義します。たとえば、列 A、B、C で構成される複合キーを次のとおり指定できます。

```
CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR
       (A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))
```

- 一般に、宣言型参照整合性を基にして、複合外部キーの NULL に対する「部分一致」規則（NULL 以外の列が、参照先の主キーまたは一意キー列の同等の位置にあること）を施行することはできません。この場合、第 15 章「トリガーの使用」で説明するように、トリガーを使用して処理できることがよくあります。

図 4-3 参照整合性制約を使用した表



親表と子表との関連の定義

親表と子表との関連のいくつかは、子表の外部キーで定義されている他のタイプの整合性制約によって判断できます。

外部キーに他の制約がない場合 外部キーについて他の制約が定義されていない場合は、子表の行は何行でも同一の親キー値を参照できます。このモデルでは、外部キーに NULL が許可されます。

このモデルは、外部キーに未定の値（NULL）を許可する親キーと外部キーとの間に「1 対多」関連を確立します。従業員表と部門表の間のこのような関連の例を、4-9 ページの図 4-3に示します。各部門（親キー）には多数の従業員（外部キー）が所属しますが、一部の従業員は、部門に所属していない場合があります（外部キーで NULL）。

外部キーに NOT NULL 制約がある場合 外部キーで NULL が許可されていない場合は、子表の各行は親キーの値を明示的に参照する必要があります。ただし、子表の行は何行でも同一の親キー値を参照できます。

このモデルは、親キーと外部キーとの間に「1 対多」関連を確立します。ただし、子表の各行は、必ず親キー値に対する参照を持っている必要があり、外部キーに値の欠如（NULL）があってはいけません。前述の項の例を使用して、この関連を説明できます。ただし、このモデルでは、従業員は必ず特定の部門への参照を持つ必要があります。

外部キーに一意キー制約がある場合 外部キーに一意キー制約が定義されている場合は、子表の中の 1 行のみが、親キー値を参照できます。このモデルでは、外部キーに NULL が許可されます。

このモデルは、外部キーに未定義の値（NULL）が許可される、親キーと外部キーとの「1 対 1」関連を確立します。たとえば、従業員表に、企業の保険計画の従業員の会員番号を参照する MEMBERNO という名前の列があると想定します。また、INSURANCE という表には、MEMBERNO という主キーがあり、その他の列は保険証書に関連した各従業員の情報を保持しているとします。次の理由によって、従業員表の MEMBERNO は、外部キーかつ一意キーである必要があります。

- EMP_TAB 表および INSURANCE 表の参照整合性規則を施行するため（外部キー制約）
- 各従業員の会員番号を一意にするため（一意キー制約）

外部キーに一意キー制約および NOT NULL 制約がある場合 一意キー制約および NOT NULL 制約の両方が外部キーに定義されている場合は、子表の 1 行のみが親キー値を参照できます。外部キーには NULL が許可されていないため、子表の各行は、明示的に親キーの値を参照する必要があります。

このモデルは、外部キーに未定義の値（NULL）が許可されない親キーと外部キーとの「1 対 1」関連を確立します。前述の例を拡張して、各従業員が一意の会員番号を持つように保証するとともに、従業員表の MEMBERNO 列に NOT NULL 制約を追加することで、従業員表の MEMBERNO 列に未定義の値（NULL）が許可されないようにできます。

複数の外部キー制約に関する規則

Oracle では、1 つの列を複数の外部キー制約で参照できます。依存キーの数に制限はありません。ある列が 2 つの異なる複合外部キーの一部になっている場合に、この状況が発生する可能性があります。

制約チェックの遅延

Oracle が制約を確認する場合に、制約が満たされないときは、エラーが表示されます。SET CONSTRAINTS 文を使用して、トランザクションが終了するまで制約の妥当性チェックを遅延できます。

> **注意：** SET CONSTRAINT 文は、トリガー内では発行できません。

SET CONSTRAINTS の設定は、トランザクションが終了するか、または別の SET CONSTRAINTS 文でモードがリセットされるまで継続されます。

> **参照：** SET CONSTRAINTS 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

制約チェックの遅延に関するガイドライン

データを適切に選択する データに次のいずれかの特性がある場合、一意キーおよび外部キーの制約チェックを遅延する必要があります。

- 表がスナップショットの場合
- 別のアプリケーションで処理された大量のデータを含む表であり、同じ順序でデータを戻すかどうかわからない場合
- 外部キーに対する更新カスケード操作の場合

別のアプリケーションで処理されている大量のデータを扱う場合は、制約チェックをトランザクションの終了まで遅延できます。

制約が遅延可能かどうかを確認する 適切な表を識別し、選択した後、表の外部キー、一意キーおよび主キーが遅延可能に作成されているかどうかを確認します。確認するには、次のような文を発行します。

```
CREATE TABLE dept (
     deptno NUMBER PRIMARY KEY,
     dname VARCHAR2 (30)
     );
CREATE TABLE emp (
     empno NUMBER,
```

```
     ename VARCHAR2 (30),
     deptno NUMBER REFERENCES (dept),
     CONSTRAINT epk PRIMARY KEY (empno) DEFERRABLE,
     CONSTRAINT efk FOREIGN KEY (deptno)
     REFERENCES (dept.deptno) DEFERRABLE);
INSERT INTO dept VALUES (10, 'Accounting');
INSERT INTO dept VALUES (20, 'SALES');
INSERT INTO emp VALUES (1, 'Corleone', 10);
INSERT INTO emp VALUES (2, 'Costanza', 20);
COMMIT;

SET CONSTRAINT efk DEFERRED;
UPDATE dept SET deptno = deptno + 10
     WHERE deptno = 20;

SELECT * from emp ORDER BY deptno;
EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       20
UPDATE emp SET deptno = deptno + 10
     WHERE deptno = 20;
SELECT * FROM emp ORDER BY deptno;

EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       30
COMMIT;
```

すべての制約に遅延を設定する データを処理するアプリケーション内で、任意のデータを処理する前に、すべての制約に遅延を設定する必要があります。遅延可能なすべての制約に遅延を設定するには、次の DML 文を使用します。

```
SET CONSTRAINTS ALL DEFERRED;
```

> **注意：** SET CONSTRAINTS 文は、カレント・トランザクションにのみ適用されます。制約の作成時に指定したデフォルト値は、制約が存在するかぎり維持されます。ALTER SESSION SET CONSTRAINTS 文は、カレント・セッションにのみ適用されます。

コミットを確認する（オプション） COMMIT を発行する直前に SET CONSTRAINTS ALL IMMEDIATE 文を発行すると、コミットする前に制約違反を確認できます。制約に問題があった場合、この文は失敗し、エラーの原因になった制約が識別されます。制約が違反しているときにコミットした場合、トランザクションはロールバックされ、エラー・メッセージが表示されます。

対応付けられた索引がある制約の管理

一意キーまたは主キーを作成すると、Oracle は、制約の一意性を強制するために既存の索引が使用できるかどうかを確認します。既存の索引がない場合、Oracle は索引を作成します。

制約に対応付けられた索引の領域と時間のオーバーヘッドの最小化

Oracle が一意索引を使用して制約を施行し、それに対応付けられた制約が削除または使用禁止にされる場合、索引は削除されます。索引に関連付けられた統計を保持する場合、または索引の再作成に時間がかかる場合は、制約の DROP コマンドに KEEP INDEX 句を指定できます。

使用可能な外部キーが主キーまたは一意キーを参照している間は、主キー制約または一意キー制約、または索引を使用禁止にしたり削除することはできません。

注意： 遅延可能なすべての一意キーおよび主キーには、非一意索引を使用してください。

一意キー制約および主キー制約の作成時に既存の索引を再利用するには、制約句に USING INDEX を含めます。次に例を示します。

```
CREATE TABLE b
(
    b1 INTEGER,
    b2 INTEGER,
    CONSTRAINT unique1 (b1, b2) USING INDEX (CREATE UNIQUE INDEX b_index on b(b1,
b2),
    CONSTRAINT unique2 (b1, b2) USING INDEX b_index
);
```

外部キーを索引付けするためのガイドライン

ほとんどの場合、外部キーには索引を付ける必要があります。一致する一意キーまたは主キーが決して更新または削除されない場合にのみ、索引を付ける必要はありません。

参照： 索引およびキーを含むロック・メカニズムの詳細は、『Oracle9*i* データベース概要』を参照してください。

分散データベース内の参照整合性

参照整合性制約の宣言では、リモート表の主キーまたは一意キーを参照する外部キーを指定できません。

ただし、トリガーを使用すると、複数のノードにまたがる親子の表の関連をメンテナンスできます。

> **参照：** 参照整合性を施行するトリガーの詳細は、第 15 章「トリガーの使用」を参照してください。

注意： トリガーを使用して分散データベースの複数のノードにまたがる参照整合性を定義する場合、ネットワーク障害が親表および子表へのアクセスを制限する可能性があることに注意してください。たとえば、子表が SALES データベースに存在し、親表が HQ データベースに存在すると想定します。

2 つのデータベース間のネットワーク接続に障害が発生すると、参照整合性トリガーが HQ データベース内の親表へアクセスする必要があるため、子表に対する DML 文が処理（行を挿入したり、外部キーの値を更新するような処理）を進めることができない場合があります。

CHECK 整合性制約を使用する場合

比較などの論理式をベースとした整合性規則を施行する必要がある場合、CHECK 制約を使用します。その他のタイプの整合性制約で必要なチェックができる場合には、CHECK 制約は使用しないでください。

> **参照：** 4-16 ページの「CHECK および NOT NULL 整合性制約の選択」を参照してください。

CHECK 制約の例を次に示します。

- 給与の値が 10000 を超えないように、従業員の給与に CHECK 制約を定義します。
- 「BOSTON」、「NEW YORK」および「DALLAS」のみが許可されるように、部門の所在地に CHECK 制約を定義します。
- コミッションの額が給与より多くならないように、給与およびコミッションの列に CHECK 制約を定義します。

CHECK 制約の制限

CHECK 整合性制約では、条件は表のすべての行に対して真または不明である必要があります。条件が偽であると評価された場合、その文はロールバックされます。CHECK 制約の条件には、次のような制限があります。

- 条件は、挿入または更新が行われている行の値を使用して評価できるブール式である必要があります。
- 条件に副問合せまたは順序を含めることはできません。
- 条件に SQL 関数 SYSDATE、UID、USER または USERENV を含めることはできません。
- 条件に疑似列 LEVEL、PRIOR または ROWNUM を含めることはできません。

  参照： これらの疑似列については、『Oracle9*i* SQL リファレンス』を参照してください。

- 条件にユーザー定義の SQL 関数を含めることはできません。

CHECK 制約の設計

CHECK 制約は、条件が偽であると評価される場合にのみ CHECK 制約に違反します。真および不明（NULL と比較して）はチェック条件には違反しません。したがって、定義する CHECK 制約が、規則を施行するために十分明確であることを確認してください。

たとえば、次の CHECK 制約について考えます。

```
CHECK (Sal > 0 OR Comm >= 0)
```

この規則は、「従業員の給与が 0（ゼロ）以下の場合、または従業員のコミッションが 0（ゼロ）未満の場合は、従業員表の行を許可しない」と解釈されます。ただし、給与に NULL の値を持つ行は、チェック条件全体が不明であると評価されるため、そのコミッションの値の評価にかかわらず、CHECK 制約に違反しません。このような場合には、SAL 列と COMM 列の両方に NOT NULL 整合性制約を設定することによって、このような違反を回避できます。

注意： どのような場合に不明な値が NULL 条件になるかについては、『Oracle9*i* SQL リファレンス』の論理演算子 AND および OR の真理値表を参照してください。

複数の CHECK 制約に関する規則

1 つの列に、その定義で列を参照する複数の CHECK 制約を指定できます。定義できる CHECK 制約の数に制限はありません。

制約が評価される順序は定義されません。そのため、順序に依存したり、互いに競合するような複数の制約を定義しないでください。

CHECK および NOT NULL 整合性制約の選択

ANSI/ISO 規格によると、NOT NULL 整合性制約は CHECK 整合性制約の 1 つであり、その条件は次のとおりです。

```
CHECK (Column_name IS NOT NULL)
```

このため、単一列に対する NOT NULL 整合性制約は、実際には、NOT NULL 制約または CHECK 制約を使用して 2 種類の形式で記述できます。使用しやすさという点で、IS NOT NULL 条件を指定した CHECK 制約でなく、NOT NULL 整合性制約を定義することをお薦めします。

複合キーがすべて NULL またはすべて値を持つ場合は、CHECK 整合性制約を使用する必要があります。たとえば、次の CHECK 整合性制約の式を指定すると、列 C1 および C2 を構成する複合キーのキー値が、すべて NULL またはすべて値を持つことができます。

```
CHECK ((C1 IS NULL AND C2 IS NULL) OR
    (C1 IS NOT NULL AND C2 IS NOT NULL))
```

整合性制約の定義の例

次に、データベース設計のプロトタイプ・フェーズでの簡単な制約の作成方法を示します。

すべての制約に対する名前の付け方に注意してください。制約に名前を付けると、DDL が複数回実行された場合に、システムが生成した異なる名前で、データベースが同じ制約の複数のコピーを作成することを回避できます。

> **参照：** 大規模な本番データベースに対する制約の作成方法およびメンテナンス方法の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。

CREATE TABLE コマンドを使用した整合性制約の定義 : 例

次の CREATE TABLE 文で、いくつかの整合性制約の定義の具体例を示します。

```
CREATE TABLE Dept_tab (
    Deptno  NUMBER(3) CONSTRAINT Dept_pkey PRIMARY KEY,
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15),
                CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
                CONSTRAINT Loc_check1
                    CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));

CREATE TABLE Emp_tab (
    Empno    NUMBER(5) CONSTRAINT Emp_pkey PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
```

```
Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
                   REFERENCES Emp_tab,
Hiredate DATE,
Sal      NUMBER(7,2),
Comm     NUMBER(5,2),
Deptno   NUMBER(3) NOT NULL
         CONSTRAINT dept_fkey REFERENCES Dept_tab ON DELETE CASCADE);
```

ALTER TABLE コマンドを使用した制約の定義 : 例

整合性制約は、ALTER TABLE コマンドの制約句を使用しても定義できます。たとえば、次の ALTER TABLE 文で、いくつかの整合性制約の定義の具体例を示します。

```
CREATE UNIQUE INDEX I_dept ON Dept_tab(deptno);
ALTER TABLE Dept_tab
    ADD CONSTRAINT Dept_pkey PRIMARY KEY (deptno);

ALTER TABLE Emp_tab
    ADD CONSTRAINT Dept_fkey FOREIGN KEY (Deptno) REFERENCES Dept_tab;
ALTER TABLE Emp_tab MODIFY (Ename VARCHAR2(15) NOT NULL);
```

制約に違反する行が表にすでに存在している場合、VALIDATED 状態の制約は作成できません。

制約の作成に必要な権限

制約を作成するには、制約のある表を作成する権限（CREATE TABLE または CREATE ANY TABLE システム権限）または変更する権限（表に対する ALTER オブジェクト権限または ALTER ANY TABLE システム権限）が必要です。さらに、一意キーおよび主キー整合性制約では、表の所有者に、対応する索引を含む表領域の割当て制限または UNLIMITED TABLESPACE システム権限のいずれかが必要です。外部キー整合性制約では、その他にもいくつかの権限が必要です。

> **参照：** 4-24 ページの「外部キー整合性制約の作成に必要な権限」を参照してください。

整合性制約のネーミング

NOT NULL、一意キー、主キー、外部キーおよび CHECK の各制約に対して、制約句の CONSTRAINT オプションを使用して名前を割り当ててください。この名前は、そのユーザーが所有している他の制約名に対して一意である必要があります。制約名を指定しない場合、Oracle が名前を生成して割り当てます。

独自の名前を指定すると、制約違反のエラー・メッセージがよりわかりやすくなります。また、SQL 文が複数回実行された場合に複数の制約が作成されることを回避できます。

制約句の CONSTRAINT オプションの例として、前述の CREATE TABLE 文および ALTER TABLE 文の例を参照してください。なお、データ・ディクショナリでは、制約名の他に、その制約に関する他の情報も参照できます。

> **参照：** データ・ディクショナリ・ビューの例は、4-26 ページの「整合性制約の定義のビュー」を参照してください。

整合性制約の使用可能および使用禁止

この項では、整合性制約をユーザー自身で使用可能および使用禁止にするしくみ、および手順について説明します。

使用可能にされた制約 制約が使用可能な場合、対応する規則が対応付けられた列のデータ値に施行されます。制約の定義は、データ・ディクショナリ内に格納されます。

使用禁止にされた制約 制約が使用禁止の場合、それに対応する規則は施行されません。制約の定義は、データ・ディクショナリ内に格納されたままです。

整合性制約は、データベース内のデータに関するアサーションを表します。このアサーションは、制約を使用可能にすると必ず真になります。制約を使用禁止にすると、整合性制約に違反するデータがデータベース内に存在する可能性があるため、アサーションは真である場合も真でない場合もあります。

制約を使用禁止にする理由

日常の操作では、制約は常に使用可能にしておく必要があります。特定の状況においては、パフォーマンス上の理由から、表の整合性制約を一時的に使用禁止にする必要がある場合があります。次に例を示します。

- SQL*Loader を使用して、表に大量のデータをロードする場合
- 表に対して大規模な変更を行うバッチ作業を実施する場合（たとえば、既存の番号に 1000 を加えてすべての従業員番号を変更する場合）
- 表を 1 つずつインポートまたはエクスポートする場合

整合性制約を一時的に使用禁止にすると、これらの操作が高速になります。

整合性制約の例外

表の行が整合性制約に違反する場合、この行は制約違反になり、制約に対する**例外**とされます。例外が存在する場合、制約を使用可能にはできません。制約に違反する行は、制約を使用可能にする前に更新または削除する必要があります。

制約を使用可能にするときに、特定の整合性制約に対する例外を指定できます。

> **参照：** この手順については、4-21 ページの「制約の例外の修正」を参照してください。

制約を使用可能にする

CREATE TABLE 文または ALTER TABLE 文で整合性制約を定義するときには、Oracle はその制約を自動的に使用可能にします。コードをわかりやすくするために、その定義に ENABLE 句を含めることによって、制約を明示的に使用可能にできます。

初期状態が空で、個々のトランザクションによって 1 つずつ行が移入される表を作成する場合は、この方法を使用します。この場合は、データが常に一貫しており、各 DML 操作におけるパフォーマンス上のオーバーヘッドが小さいことを確認する必要があります。

次の CREATE TABLE 文および ALTER TABLE 文は、どちらも整合性制約を定義し、使用可能にします。

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY);
 ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno);
```

整合性制約を使用可能にする ALTER TABLE 文は、表の行がその整合性制約に違反するとエラーになります。文はロールバックされ、制約定義は格納されず使用可能にもなりません。

> **参照：** 整合性制約に違反する行の詳細は、4-21 ページの「制約の例外の修正」を参照してください。

制約を使用禁止にする

次の CREATE TABLE 文および ALTER TABLE 文は、どちらも整合性制約を定義し、使用禁止にします。

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY DISABLE);

ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno) DISABLE;
```

大量のデータがロードされる表を作成する場合は、他のユーザーが表にアクセスする前にこの方法を使用します。特に、データのロード後にデータをクリーンアップしたり、空の列に順序番号または親子関係を埋める必要がある場合です。

整合性制約を定義して使用禁止にする ALTER TABLE 文は、その規則が施行されていないため、エラーとなることはありません。

既存の整合性制約の使用可能および使用禁止

次の目的で ALTER TABLE コマンドを使用します。

- ENABLE 句を使用して、使用禁止になっている制約を使用可能にする場合
- DISABLE 句を使用して、使用可能になっている制約を使用禁止にする場合

既存の制約を使用可能にする

データをクリーンアップしたり、空の列を埋めた後で、データのロード中に使用禁止にされていた制約を使用可能にできます。

次の 2 つの文は、使用禁止にされた整合性制約を使用可能にする例です。

```
ALTER TABLE Dept_tab
    ENABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    ENABLE PRIMARY KEY
    ENABLE UNIQUE (Dname)
    ENABLE UNIQUE (Loc);
```

整合性制約を使用可能にする ALTER TABLE 文は、表の行がその整合性制約に違反するとエラーとなります。この場合、文はロールバックされ、制約は使用可能になりません。

> **参照：** 整合性制約に違反する行の詳細は、4-21 ページの「制約の例外の修正」を参照してください。

既存の制約を使用禁止にする

すでにデータが含まれている表に対して、大規模なロードまたは更新を実行する必要がある場合は、整合性制約を一時的に使用禁止にして、バルク操作のパフォーマンスを向上させることができます。

次の文は、使用可能にされた整合性制約を使用禁止にする例です。

```
ALTER TABLE Dept_tab
    DISABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    DISABLE PRIMARY KEY
    DISABLE UNIQUE (Dname)
    DISABLE UNIQUE (Loc);
```

ヒント : データ・ディクショナリを使用した制約の検索

前述の例では、制約の名前および影響される列を把握している必要があります。この情報を検索するには、制約に対して定義されているデータ・ディクショナリ・ビュー USER_CONSTRAINTS または USER_CONS_COLUMNS の 1 つを問い合せます。これらのビューの詳細は、4-26 ページの「整合性制約の定義のビュー」および『Oracle9*i* データベース・リファレンス』を参照してください。

キー整合性制約の使用可能および使用禁止に関するガイドライン

一意キー、主キーおよび外部キーの各整合性制約を使用可能または使用禁止にする場合は、いくつかの重要な問題および前提条件を認識しておく必要があります。一意キー制約および主キー制約は、通常、データベース管理者が管理します。

> **参照：** 4-24 ページの「外部キー整合性制約の管理」および『Oracle9*i* データベース管理者ガイド』を参照してください。

制約の例外の修正

制約を作成または使用可能にするときに、整合性制約の例外があるために文が正常に実行されなかった場合、文はロールバックされます。この場合、すべての例外が更新または削除されるまで制約を使用可能にできません。整合性制約に違反している行を判断するには、CREATE TABLE 文または ALTER TABLE 文の ENABLE 句に EXCEPTIONS オプションを指定します。

> **参照：** 制約の例外を修正する場合は、『Oracle9*i* データベース管理者ガイド』を参照してください。

整合性制約の変更

Oracle8*i* 以上では、MODIFY CONSTRAINT 句を使用して、既存の制約状態を変更できます。

> **参照：** 変更できるパラメータに関する詳細は、『Oracle9*i* SQL リファレンス』の「ALTER TABLE」の項を参照してください。

MODIFY CONSTRAINT の例 1

次のコマンドは、CHECK 制約が施行されるかどうか、およびその制約チェックが実行されたときの選択肢を示します。

```
CREATE TABLE X1_tab (a1 NUMBER CONSTRAINT y CHECK (a1>3) DEFERRABLE DISABLE);

ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt RELY;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt INITIALLY DEFERRED;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE NOVALIDATE;
```

MODIFY CONSTRAINT の例 2

次のコマンドは、NOT NULL 制約が施行されるかどうか、およびその制約チェックが実行されたときの選択肢を示します。

```
CREATE TABLE X1_tab (A1 NUMBER CONSTRAINT Y_cnstrt
NOT NULL DEFERRABLE INITIALLY DEFERRED NORELY DISABLE);
```

```
ALTER TABLE X1_tab ADD CONSTRAINT One_cnstrt UNIQUE(A1)
DEFERRABLE INITIALLY IMMEDIATE RELY USING INDEX PCTFREE = 30
ENABLE VALIDATE;

ALTER TABLE X1_tab MODIFY UNIQUE(A1)
INITIALLY DEFERRED NORELY USING INDEX PCTFREE = 40
ENABLE NOVALIDATE;

ALTER TABLE X1_tab MODIFY CONSTRAINT One_cnstrt
INITIALLY IMMEDIATE RELY;
```

MODIFY CONSTRAINT の例 3

次のコマンドは、主キー制約が施行されるかどうか、およびその制約チェックが実行されたときの選択肢を示します。

```
CREATE TABLE T1_tab (A1 INT, B1 INT);
ALTER TABLE T1_tab add CONSTRAINT P1_cnstrt PRIMARY KEY(a1) DISABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY INITIALLY IMMEDIATE
USING INDEX PCTFREE = 30 ENABLE NOVALIDATE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY
USING INDEX PCTFREE = 35 ENABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

整合性制約名の変更

制約名は、複数のスキーマにわたって一意である必要があるため、表およびそのすべての制約の複製時に新しい表の制約名と元の表の制約名が競合すると、問題が発生します。また、制約は、デフォルトのシステム生成名で作成すると、簡単に使用可能または使用禁止にするために、後で覚えやすい制約名を付ける必要がある場合があります。

制約の変更可能なプロパティの 1 つは、制約名です。次に、制約のシステム生成名を検索し、任意の名前へ変更する場合に使用する SQL*Plus スクリプトを示します。

```
prompt Enter table name to find its primary key:
accept table_name
select constraint_name from user_constraints
  where table_name = upper('&table_name.')
  and constraint_type = 'P';

prompt Enter new name for its primary key:
accept new_constraint

set serveroutput on
```

```
declare
-- USER_CONSTRAINTS.CONSTRAINT_NAME is declared as VARCHAR2(30).
-- Using %TYPE here protects us if the length changes in a future release.
  constraint_name user_constraints.constraint_name%type;
begin
  select constraint_name into constraint_name from user_constraints
    where table_name = upper('&table_name.')
    and constraint_type = 'P';

  dbms_output.put_line('The primary key for ' || upper('&table_name.') || ' is: ' ||
constraint_name);

  execute immediate
    'alter table &table_name. rename constraint ' || constraint_name ||
    ' to &new_constraint.';
end;
/
```

整合性制約の削除

整合性制約は、施行する規則が業務に適応しなくなった場合、またはその制約が不要になった場合に削除します。整合性制約は、ALTER TABLE コマンドの DROP 句を使用して削除します。たとえば、次の文は整合性制約を削除します。

```
ALTER TABLE Dept_tab
    DROP UNIQUE (Dname);
ALTER TABLE Dept_tab
    DROP UNIQUE (Loc);

ALTER TABLE Emp_tab
    DROP PRIMARY KEY,
    DROP CONSTRAINT Dept_fkey;

DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

一意キー、主キーおよび外部キーの各整合性制約を削除する場合は、いくつかの重要な問題および前提条件を把握しておく必要があります。一意キー制約および主キー制約は、通常、データベース管理者が管理します。

参照： 4-24 ページの「外部キー整合性制約の管理」および『Oracle9*i* データベース管理者ガイド』を参照してください。

外部キー整合性制約の管理

すべてのタイプの整合性制約の定義、使用可能、使用禁止および削除に関する一般的な情報については、前の項で説明しました。次の項では、特に外部キー整合性制約の問題に重点を置き、それらの情報を補足します。この整合性制約は、異なる表の列同士の関連を規定します。

外部キー整合性制約の規則

次に示す情報は、外部キー整合性制約を定義するときに重要です。

外部キー列のデータ型および名前

依存する側の表と参照される側の表の対応する列は、同じデータ型である必要があります。列の名前は一致する必要はありません。

複合外部キーにおける列の制限

外部キーは、親表の主キーまたは一意キーを参照し、主キー制約および一意キー制約は索引を使用して施行されるため、複合外部キーは 32 列以内に制限されています。

デフォルトでの外部キーの主キー参照

外部キー制約（単一列または複合列）を定義するときに REFERENCES オプションに列リストが指定されていないと、Oracle は、指定した表の主キーが参照されるものとみなします。または、カッコの中に親表で参照する列を明示的に指定できます。この列リストが親表の主キーまたは一意キーを参照するかどうかは、Oracle によって自動的に確認されます。参照していない場合は、エラー情報が戻されます。

外部キー整合性制約の作成に必要な権限

外部キー制約を作成するには、制約の作成者に親表および子表に対するアクセス権限が必要です。

- **親表** 参照整合性制約の作成者は、親表を所有するか、または親表の親キーを構成する列に対する REFERENCES オブジェクト権限が必要です。
- **子表** 参照整合性制約の作成者は、表を作成する権限（CREATE TABLE または CREATE ANY TABLE のシステム権限）、または子表を変更する権限（子表の ALTER オブジェクト権限または ALTER ANY TABLE システム権限）が必要です。

どちらの場合も、必要な権限をロールを介して取得することはできません。権限は、明示的に制約の作成者に付与する必要があります。

これらの制限によって、次のことが可能となります。

- 子表の所有者は、自分自身の表に施行される制約の種類、または自分自身の表に対して制約を作成できる他のユーザーを、明示的に決定できます。
- 親表の所有者は、外部キーが所有者自身の表の主キーおよび一意キーに依存可能であるかどうかを明示的に決定できます。

外部キーによる参照整合性の施行方法の選択

Oracle では、外部キー制約の定義の指定どおりに、異なるタイプの参照整合性アクションを施行できます。

- **親キーの更新または削除の回避** このデフォルト設定を指定すると、キーを参照する行が子表内にある場合、親キーが更新または削除されることはありません。次に例を示します。

```
CREATE TABLE Emp_tab (
FOREIGN KEY (Deptno) REFERENCES Dept_tab);
```

- **親キー削除時の子である行の削除** ON DELETE CASCADE アクションを指定すると、子表が参照する親キーのデータを削除できます。ただし、更新はできません。親キー内のデータが削除されると、削除された親キー値に依存する子表のすべての行も削除されます。この参照アクションを指定するには、外部キー制約の定義に ON DELETE CASCADE オプションを指定します。次に例を示します。

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE CASCADE);
```

- **親キー削除時の外部キーへの NULL 設定** ON DELETE SET NULL アクションを指定すると、親キーを参照するデータを削除できます。ただし、更新はできません。親キー内の参照データが削除されると、削除された親キー値に依存する子表内のすべての行の外部キーが NULL に設定されます。この参照アクションを指定するには、外部キー制約の定義に ON DELETE SET NULL オプションを指定します。次に例を示します。

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE SET NULL);
```

外部キー整合性制約の使用可能に関する制限

参照中の主キーまたは一意キーの制約が存在していないか、または使用可能になっていない場合、外部キー整合性制約を使用可能にできません。

整合性制約の定義のビュー

データ・ディクショナリには、整合性制約について次のビューがあります。

- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS

これらのビューを問い合せて、制約の名前、影響がある列および制約の管理に有効な他の情報を検索できます。

> **参照：** 各ビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

整合性制約の定義の例

多くの整合性制約を定義する次の CREATE TABLE 文について考えてみます。

```
CREATE TABLE Dept_tab (
    Deptno   NUMBER(3) PRIMARY KEY,
    Dname    VARCHAR2(15),
    Loc      VARCHAR2(15),
    CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
    CONSTRAINT LOC_CHECK1
        CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));

CREATE TABLE Emp_tab (
    Empno    NUMBER(5) PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
         REFERENCES Emp_tab ON DELETE CASCADE,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
    CONSTRAINT Dept_fkey REFERENCES Dept_tab);
```

例 1: アクセス可能なすべての制約のリスト 次の問合せで、ユーザーがアクセス可能なすべての表に定義されているすべての制約がリストされます。

```
SELECT Constraint_name, Constraint_type, Table_name,
    R_constraint_name
    FROM User_constraints;
```

この項の最初の文の場合、次のリストが戻されます。

```
CONSTRAINT_NAME  C TABLE_NAME  R_CONSTRAINT_NAME
---------------  - ----------- ------------------
SYS_C00275       P DEPT_TAB
DNAME_UKEY       U DEPT_TAB
LOC_CHECK1       C DEPT_TAB
SYS_C00278       C EMP_TAB
SYS_C00279       C EMP_TAB
SYS_C00280       P EMP_TAB
MGR_FKEY         R EMP_TAB     SYS_C00280
DEPT_FKEY        R EMP_TAB     SYS_C00275
```

次のことに注意してください。

- 制約名には、ユーザー指定（たとえば、DNAME_UKEY）のものと、システム指定（たとえば、SYS_C00275）のものがあります。
- 各制約タイプは、CONSTRAINT_TYPE 列に別々の文字で表示されます。次の表に、各制約タイプに対応する文字を示します。

制約タイプ	文字
主キー	P
一意キー	U
外部キー	R
CHECK、NOT NULL	C

注意： その他の制約タイプで、CONSTRAINT_TYPE 列に「V」の文字で表示されるものがあります。この制約タイプは、ビューに対する WITH CHECK OPTION によって作成された制約に対応しています。各ビューおよび WITH CHECK OPTION の詳細は、第 2 章「スキーマ・オブジェクトの管理」を参照してください。

例 2: NOT NULL 制約と CHECK 制約の区別 例 1 では、「C」というタイプの制約がいくつか表示されています。どの制約が EMP_TAB 表および DEPT_TAB 表の NOT NULL 制約であり、CHECK 制約であるかを区別するには、次の問合せを発行します。

```
SELECT Constraint_name, Search_condition
    FROM User_constraints
    WHERE (Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB') AND
        Constraint_type = 'C';
```

この項の最初の CREATE TABLE 文の場合、次のリストが戻されます。

```
CONSTRAINT_NAME  SEARCH_CONDITION
---------------  ---------------------------------------
LOC_CHECK1       loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C00278       ENAME IS NOT NULL
SYS_C00279       DEPTNO IS NOT NULL
```

次のことに注意してください。

- NOT NULL 制約は SEARCH_CONDITION 列で明確に識別されています。
- ユーザー定義による CHECK 制約の条件は、明示的に SEARCH_CONDITION 列にリストされます。

例 3: 整合性制約を構成する列名のリスト 次の問合せで、ユーザーがアクセス可能なすべての表に定義されている制約を構成するすべての列がリストされます。

```
SELECT Constraint_name, Table_name, Column_name
    FROM User_cons_columns;
```

この項の最初の文の場合、次のリストが戻されます。

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  ----------- ---------------
DEPT_FKEY        EMP_TAB     DEPTNO
DNAME_UKEY       DEPT_TAB    DNAME
DNAME_UKEY       DEPT_TAB    LOC
LOC_CHECK1       DEPT_TAB    LOC
MGR_FKEY         EMP_TAB     MGR
SYS_C00275       DEPT_TAB    DEPTNO
SYS_C00278       EMP_TAB     ENAME
SYS_C00279       EMP_TAB     DEPTNO
SYS_C00280       EMP_TAB     EMPNO
```

5

索引計画の選択

この章では、アプリケーションで異なるタイプの索引を使用する際の考慮点について説明します。この章の内容は次のとおりです。

- アプリケーション固有の索引のガイドライン
- 索引の作成の基本例
- ファンクション索引を使用する場合

参照：

- 索引の使用方法の詳細は、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。
- 索引の作成および管理については、『Oracle9*i* データベース管理者ガイド』を参照してください。
- 索引を処理するコマンドの構文については、『Oracle9*i* SQL リファレンス』を参照してください。

アプリケーション固有の索引のガイドライン

Oracle では、表の行に高速にアクセスするために索引が使用されています。索引を使用すると、表の行全体のわずかな部分を戻す操作でのデータ・アクセスが、より高速になります。

Oracle では、表に対して作成できる索引の数の制限はありません。ただし、索引は、問合せの高速化を目的として使用される場合にのみ有効です。そうでない場合は、領域を占有し、索引列が更新されるときにオーバーヘッドが増加するのみです。実行計画機能を使用して、問合せ内での索引の使用方法を、判断する必要があります。索引がデフォルトで使用されていない場合は、問合せヒントを使用して、索引が使用されるようにできます。

次の項では、SQL コマンドを使用して索引を作成、変更および削除する方法を説明します。索引を管理するときの簡単なガイドラインも示します。

> **参照：** 問合せヒント、および索引によるパフォーマンス向上率の測定方法については、『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

表データ挿入後の索引の作成

索引は、通常、データが表に挿入された後、または（SQL*Loader またはインポートによって）ロードされた後で作成します。そうしないと、索引更新のオーバーヘッドによって、挿入操作またはロード操作に時間がかかります。この規則の例外として、クラスタにデータを挿入するときは、事前にそのクラスタに索引を作成しておく必要があります。

索引作成時の問題を回避するための一時表領域の切替え

すでにデータを持っている表に対して索引を作成する場合、Oracle はソート領域を使用します。Oracle は、索引の作成者に対して割り当てられたメモリー内のソート領域（ユーザー当たりの容量は `SORT_AREA_SIZE` 初期化パラメータによって決まります）を使用しますが、さらに索引作成のために割り当てられた一時セグメントとの間で、ソート情報をスワップする必要があります。索引が非常に大きい場合、次の手順を完了すると効果があることがあります。

1. `CREATE TABLESPACE` コマンドを使用して新しい一時表領域を作成します。
2. `ALTER USER` コマンドの `TEMPORARY TABLESPACE` オプションを使用して、この表領域を自分の新しい一時表領域にします。
3. `CREATE INDEX` コマンドを使用して索引を作成します。
4. `DROP TABLESPACE` コマンドを使用してこの表領域を削除します。`ALTER USER` コマンドを使用して自分の一時表領域を元の一時表領域にリセットします。

条件によっては、SQL*Loader のダイレクト・パス・ロードを使用してデータを表にロードし、データをロードしながら索引を作成できます。

参照： ダイレクト・パス・ロードについては、『Oracle9*i* データベース・ユーティリティ』を参照してください。

正しい表および列の索引付け

どのような場合に索引を作成するかは、次のガイドラインを使用して判断してください。

- 大きな表で頻繁に検索される行の割合が 15% 未満の場合は索引を作成してください。この行の割合は、表スキャンの相対速度、および索引キーに対してクラスタ化された行データの量によって異なります。表スキャンが高速であるほど割合は低くなり、クラスタ化されている行データが多いほど割合は高くなります。
- 複数の表の結合におけるパフォーマンスを改善するために、結合に使用される列に索引を付けてください。
- 主キーおよび一意キーは自動的に索引を持ちますが、外部キーにも索引を作成する必要がある場合があります。詳細は、第 4 章「制約によるデータ整合性のメンテナンス」を参照してください。
- 小さな表には索引は必要ありません。問合せにかなり時間がかかるときには、表が大きくなっていることがあります。

列の中には、索引付けの候補があります。次の特長を 1 つでも持つ列は、索引付けの候補列となります。

- 列の値が比較的一意である。
- 値の範囲が広い（通常の索引に適している）。
- 値の範囲が狭い（ビットマップ索引に適している）。
- 列には多くの NULL が含まれるが、問合せで、値を持つすべての行が選択されることがよくある。この場合、すべての非 NULL 値が一致する比較は、

  ```
  WHERE COL_X >= -9.99 *power(10,125)
  ```

  の方が、次の句より適しています。

  ```
  WHERE COL_X IS NOT NULL
  ```

  これは、最初の句が COL_X に対する索引を使用するためです（COL_X は数値列であると想定）。

次の特長を持つ列は、索引を付ける対象として適していません。

- 列の中に多くの NULL があり、非 NULL 値は検索されない。

LONG 列および LONG RAW 列は、索引付けできません。

単一の索引エントリのサイズは、データ・ブロック内の使用可能領域の約 2 分の 1（さらにオーバーヘッドを差し引いたもの）を超えることはできません。データベース管理者に相談して、索引に必要な領域を判断してください。

表当たりの索引数の制限

索引が多いほど、表を変更するときに多くのオーバーヘッドが発生します。行が挿入または削除されるとき、その表のすべての索引も更新される必要があります。列が更新されるときには、その列に対するすべての索引も更新される必要があります。

索引による問合せパフォーマンスの向上と、更新によるパフォーマンス・オーバーヘッドを比較する必要があります。たとえば、表が主に読込み専用の場合は索引数を増やすと有効ですが、表が頻繁に更新される場合は索引数を減らす方が有効です。

複合索引の列の順序付け

CREATE INDEX コマンドには、任意の順序で列を指定できますが、CREATE INDEX 文における列の順序は、問合せのパフォーマンスに影響する可能性があります。一般に、最も使用頻度が高いと予想される列を索引の先頭に置いてください。複合索引は、複数の列を使用して作成できます。これらすべての列またはその一部を参照する問合せには、同じ複合索引を使用できます。

たとえば、図 5-1 に示すような VENDOR_PARTS 表の列を想定します。

図 5-1 VENDOR_PARTS 表

VENDOR_PARTS表

VEND ID	PART NO	UNIT COST
1012	10–440	.25
1012	10–441	.39
1012	457	4.95
1010	10–440	.27
1010	457	5.10
1220	08–300	1.33
1012	08–300	1.19
1292	457	5.28

ベンダーは 5 社、各ベンダーはおよそ 1000 個の部品を持っていると想定しています。

VENDOR_PARTS 表は、次の SQL 文によって問い合せるとします。

```
SELECT * FROM vendor_parts
    WHERE part_no = 457 AND vendor_id = 1012;
```

このような問合せのパフォーマンスを向上させるには、次のように最も選択頻度の高い列（最も多くの値を持つ列）を先頭にした複合索引を作成します。

```
CREATE INDEX ind_vendor_id
    ON vendor_parts (part_no, vendor_id);
```

複合索引は、索引の先頭部分を使用する問合せの検索速度を向上させます。したがって、前述の例では、PART_NO 列のみを使用する WHERE 句が指定された問合せでもパフォーマンスは向上します。個別値が 5 つのみのため、VENDOR_ID に単独で索引を指定しても有効ではありません。

索引の使用率をさらに正確にするための統計の収集

データベースは、問合せに関係する表についての統計情報があると、索引をより有効に使用できます。統計は、CREATE INDEX 文にキーワード COMPUTE STATISTICS を指定して索引を作成した場合に収集できます。データが更新され、値の分散が変更されたとき、ユーザーまたは DBA は DBMS_STATS.GATHER_TABLE_STATISTICS や DBMS_STATS.GATHER_SCHEMA_STATISTICS などのプロシージャをコールすることで定期的に統計をリフレッシュできます。

不要な索引の削除

索引は、次の場合には削除できます。

- 問合せを高速化しない。表が非常に小さいか、または表に数多くの行があっても索引エントリが非常に少ない。
- アプリケーションに、索引を使用する問合せが含まれていない。
- 索引を再作成する前に削除する必要がある。

索引が削除されると、その索引のセグメントのすべてのエクステントは、索引を含む表領域に戻され、表領域内の他のオブジェクトに対して使用可能になります。

索引を削除するには、SQL コマンド DROP INDEX を使用します。たとえば、固有の名前が付いた索引を削除するには、次の文を入力します。

```
DROP INDEX Emp_ename;
```

表が削除されると、対応付けられていたすべての索引は削除されます。

索引を削除するには、その索引が自スキーマに含まれているか、または DROP ANY INDEX システム権限が必要です。

索引の作成に必要な権限

アプリケーションで索引を使用する場合、DBA に、権限の付与または初期化パラメータの変更を要求する必要がある場合があります。

新しい索引を作成するには、対応する表を所有するか、またはその表に対する INDEX オブジェクト権限が必要です。また、索引を含むスキーマは、索引を含む予定の表領域に対する割当て制限、または UNLIMITED TABLESPACE システム権限が必要です。別のユーザーのスキーマに索引を作成するには、CREATE ANY INDEX システム権限が必要です。

また、ファンクション索引を作成するには QUERY_REWRITE 権限が必要であり、QUERY_REWRITE_ENABLED 初期化パラメータを TRUE に設定する必要もあります。

索引の作成の基本例

表に対する索引を作成して、対応する表に対して発行される問合せのパフォーマンスを改善できます。また、クラスタに対して索引を作成することもできます。最大 32 列までの複数列に対して複合索引を作成できます。複合索引キーは、データ・ブロック内の使用可能領域の約 2 分の 1（さらにオーバーヘッドを差し引いたもの）を超えることはできません。

Oracle は、一意キー整合性制約または主キー整合性制約を施行するために、索引を自動的に作成します。一般に、一意性を施行するには、このような制約を作成する方が、廃止された CREATE UNIQUE INDEX 構文を使用するより有効です。

索引を作成するには、SQL コマンド CREATE INDEX を使用します。

次の例では、単一の列をテストする問合せを高速化するために、その列に対して索引を作成します。

```
CREATE INDEX emp_ename ON emp_tab(ename);
```

次の例では、索引に対して複数の記憶域設定を明示的に指定します。

```
 CREATE INDEX emp_ename ON emp_tab(ename)
    TABLESPACE users
    STORAGE (INITIAL     20K
             NEXT        20k
             PCTINCREASE 75)
             PCTFREE      0
             COMPUTE STATISTICS;
```

次の例では、2 つの列に索引を適用して、最初の列または両方の列をテストする問合せを高速化します。

```
CREATE INDEX emp_ename ON emp_tab(ename, empno) COMPUTE STATISTICS;
```

次の例では、問合せは UPPER(ENAME) 関数の結果をソートします。ENAME 列の索引自体はこの操作を高速化せず、結果の行ごとに関数をコールすると時間がかかる場合があります。ファンクション索引は、列の値ごとに関数の結果を事前に計算するため、検索またはソートに関数を使用する問合せが高速になります。

```
CREATE INDEX emp_upper_ename ON emp_tab(UPPER(ename)) COMPUTE STATISTICS;
```

ドメイン索引を使用する場合

ドメイン索引は、データ・カートリッジを使用して実装された、特殊な目的を持つアプリケーションに適しています。ドメイン索引は、空間データ、時系列データ、オーディオ・データ、ビデオ・データなどの複雑なデータの操作に有効です。このようなアプリケーションを開発する必要がある場合は、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

Oracle は、このような複雑なデータの管理に有効な多くの特殊なデータ・カートリッジを提供しています。検索エンジンまたは地理情報システムを作成する必要がある場合に、適切な種類の索引を作成するのみで、ほとんどの作業を行えます。

ファンクション索引を使用する場合

ファンクション索引とは、式に対して作成される索引です。これによって、列のみの場合より索引機能が拡張されます。ファンクション索引により、データ・アクセスの方法が多様化します。

注意：

- QUERY_REWRITE_ENABLED 初期化パラメータを TRUE に設定する必要があります。
- 索引は、DBMS_STATS パッケージのプロシージャを使用して表またはスキーマに対する統計を収集すると、より有効になります。
- 索引には NULL 値を指定できません。列に NULL 値が含まれていないことを確認するか、または索引式に NVL 関数を使用して NULL を他の値に置き換えてください。

ファンクション索引で索引付けされた式は、算術式、あるいは PL/SQL ファンクション、パッケージ・ファンクション、C コールアウトまたは SQL 関数を含む式のいずれかです。ファンクション索引は、照合キーに基づく言語ソート、SQL 文の効率的な言語依存照合、および大 / 小文字を区別しないソートもサポートします。

他の索引と同様に、ファンクション索引も問合せのパフォーマンスを改善します。たとえば、複雑な計算式に頻繁にアクセスする必要がある場合は、式を索引内に格納しておくことができます。こうしておくと、その式にアクセスする必要があるときには、式はすでに計算済です。ファンクション索引のメリットについては、5-8 ページの「ファンクション索引のメリット」を参照してください。

ファンクション索引には、列に対する索引と同じすべてのプロパティがあります。ただし、列に対する索引はコストベース最適化とルールベース最適化の両方で使用できますが、ファンクション索引を使用できるのはコストベース最適化のみです。ファンクション索引に関するその他の制限については、5-10 ページの「ファンクション索引の制限」を参照してください。

> **参照：** ファンクション索引の詳細は、『Oracle9*i* データベース概要』を参照してください。ファンクション索引の作成の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。

ファンクション索引のメリット

ファンクション索引のメリットを次に説明します。

- **オプティマイザが全表スキャンのかわりにレンジ・スキャンを実行できる機会が増加します。**たとえば、WHERE 句に次のような式があるとします。

  ```
  CREATE INDEX Idx ON Example_tab(Column_a + Column_b);
  SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
  ```

  索引が（column_a + column_b）に対して作成されているため、オプティマイザはこの問合せに対してレンジ・スキャンを使用できます。述語によって大きな表の 15% 未満の行が選択された場合、レンジ・スキャンでは、通常、応答時間が短くなります。式がファンクション索引に実体化されていると、オプティマイザは式で選択される行数をより正確に見積もることができます（ファンクション索引の式が仮想列として表されるので、ANALYZE でこのような列のヒストグラムを作成できます）。

- **計算集中型の関数の値を事前に計算し、この値を索引内に格納します。**索引には、頻繁にアクセスする計算集中型の式を格納できます。その式にアクセスする必要があるときには、値はすでに計算済です。これによって、問合せ実行パフォーマンスが大幅に改善されます。

- **オブジェクト列および REF 列に対して索引を作成します。**オブジェクトを記述するメソッドは、索引作成対象の関数として使用できます。たとえば、MAP メソッドを使用してオブジェクト型列の索引を作成できます。

- **より強力なソートを作成します。**UPPER 関数および LOWER 関数を使用した大 / 小文字を区別しないソート、DESC キーワードを使用した降順ソート、および NLSSORT 関数を使用した言語ベースのソートを実行できます。

> **注意：** DESC キーワードを使用すると、列は降順にソートされます。このような索引は、ファンクション索引として扱われます。降順索引は、ビットマップ化または逆キー索引化できません。ビットマップ最適化に使用することもできません。Oracle8*i* より前のリリースの DESC の機能を使用する場合は、CREATE INDEX 文から DESC キーワードを削除してください。

表内の各都市に対してオブジェクト・メソッド distance_from_equator をコールするファンクション索引を作成します。このメソッドは、オブジェクト列 Reg_Obj に対して適用されます。問合せはこの索引を使用して、赤道からの距離が 1000 マイルを超える都市を高速に検索できます。

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));

SELECT * FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

最低気温と最高気温の差、および最高気温を格納する索引を作成します。気温差の結果は降順でソートされます。問合せはこの索引を使用して、表の行のうち、気温差が 20 未満で、最高気温が 75 より高いものを高速に検索できます。

```
CREATE INDEX compare_index
ON Weatherdata_tab ((Maxtemp - Mintemp) DESC, Maxtemp);

SELECT * FROM Weatherdata_tab
WHERE ((Maxtemp - Mintemp) < '20' AND Maxtemp > '75');
```

ファンクション索引の例

例 : 大 / 小文字を区別しない検索用のファンクション索引

次のコマンドは、表 EMP_TAB での大 / 小文字を区別しない検索をより高速にします。

```
CREATE INDEX Idx ON Emp_tab (UPPER(Ename));
```

SELECT コマンドでは、UPPER(e_name) に対するファンクション索引を使用して、:KEYCOL のような名前を持つすべての従業員を戻します。

```
SELECT * FROM Emp_tab WHERE UPPER(Ename) like :KEYCOL;
```

例 : ファンクション索引による算術式の事前計算

次のコマンドは、列 A、B および C を使用して各行の値を計算し、その結果を索引に格納します。

```
CREATE INDEX Idx ON Fbi_tab (A + B * (C - 1), A, B);
```

SELECT 文では、索引のレンジ・スキャン（式が索引 IDX の接頭辞であるため）または高速全索引スキャン（索引で高い並列度を指定してある場合はこの方が適切な場合があります）のいずれかを使用できます。

```
SELECT a FROM Fbi_tab WHERE A + B * (C - 1) < 100;
```

例 : 言語依存ソート用のファンクション索引

次の例では、各国語の照合順序に基づいたソートでのファンクション索引の使用方法を示します。NLSSORT 関数は、照合順序 GERMAN を使用して名前ごとにソート・キーを戻します。

```
CREATE INDEX Nls_index
    ON Nls_tab (NLSSORT(Name, 'NLS_SORT = German'));
```

次の SELECT 文は、表のすべての内容を選択し、NAME によって順序付けます。行はドイツ語照合順序を使用して順序付けられます。ドイツ語のセッションでは、NLS_SORT は

German、NLS_COMP は ANSI に設定されるため、グローバリゼーション・サポート・パラメータを SELECT 文に指定する必要はありません。

```
SELECT * FROM Nls_tab WHERE Name IS NOT NULL
    ORDER BY Name;
```

ファンクション索引の制限

ファンクション索引には、次の制限があることに注意してください。

- ファンクション索引を使用できるのは、コストベース最適化のみです。ファンクション索引を有効にするには、QUERY_REWRITE_ENABLED 初期化パラメータを TRUE に設定し、DBMS_STATS.GATHER_TABLE_STATISTICS または DBMS_STATS.GATHER_SCHEMA_STATISTICS をコールしてください。
- 添え字式で使用されるすべてのトップレベルまたはパッケージ・レベルの PL/SQL ファンクションは、DETERMINISTIC として宣言する必要があります。これらのファンクションは、UPPER 関数のように、同じ入力に対して常に同じ結果を戻します。Oracle はアサーションが真であることを確認しないので、サブプログラムが実際に DETERMINISTIC であることを確認する必要があります。

  キーワード DETERMINISTIC の使用方法のセマンティクスの規則を次に示します。

  - トップレベルのサブプログラムは DETERMINISTIC として宣言できます。
  - パッケージ・レベルのサブプログラムは、パッケージ仕様部では DETERMINISTIC として宣言できますが、パッケージ本体では宣言できません。パッケージ本体内で DETERMINISTIC を使用すると、エラーが発生します。
  - プライベート・サブプログラム（別のサブプログラム内またはパッケージ本体内で宣言されるサブプログラム）は、DETERMINISTIC として宣言できません。
  - DETERMINISTIC サブプログラムは、コールされるプログラムが DETERMINISTIC として宣言されているかどうかにかかわらず、別のサブプログラムをコールできます。
- ファンクション索引で使用される式には集計関数を含めることはできません。式で参照できるのは、表の同一行の列のみです。
- COMPATIBLE 初期化パラメータを 8.1.0.0.0 以上に設定し、QUERY_REWRITE_ENABLED=TRUE および QUERY_REWRITE_INTEGRITY=TRUSTED を設定する必要があります。
- 索引を使用する前に、表または索引を分析する必要があります。
- ビットマップ最適化では、降順索引を使用できません。
- ファンクション索引は、OR 拡張が実行された場合、使用されません。
- 索引ファンクションは、NOT NULL とマーク付けできません。全表スキャンを回避するには、問合せが NULL 値をフェッチできないことを保証する必要があります。

- ファンクション索引は、PL/SQL ファンクションから長さが不明な VARCHAR2 データ型または RAW データ型を戻す式は使用できません。回避策として、既知の長さのサブストリング関数を索引付けすることで、関数出力のサイズを制限します。

```
-- The INITIALS() function might return 1 letter, 2 letters, 3 letters, etc.
-- We limit the return value to 10 characters for purposes of the index.
CREATE INDEX func_substr_index ON
  emp_tab(substr(initials(ename),1,10);

-- Call SUBSTR both when creating the index and when referencing
-- the function in queries.
SELECT SUBSTR(initials(ename),1,10) FROM emp_tab;
```

6

索引構成表による索引アクセスのスピードアップ

この章の内容は次のとおりです。

> **参照：** CREATE TABLE 文の ORGANIZATION INDEX 句の構文については、『Oracle9*i* SQL リファレンス』を参照してください。

索引構成表の概要

通常の表とは対照的に、索引構成表では独自の方法でデータを構造化して格納し、索引を付けます。索引構成表の特殊性は、通常の表と比較するとわかりやすくなります。

索引構成表および通常の表

通常の表では、行の物理的な位置が決まっています。一度行に物理位置が設定されると、その行が完全に移動してしまうことはありません。新規データが追加されて、行が部分的に移動したとしても、（元の物理 ROWID によって識別される）元の物理アドレスにはその行の一部が常に残ります。システムは、元の物理アドレスから行の残りを見つけることができます。行が存在するかぎり、その物理 ROWID は変更されません。通常の表の索引は、列データと ROWID の両方を格納します。

索引構成表の行は、物理的な位置が決まっていません。索引構成表は、表の主キーに対して作成されている B* ツリー索引のリーフ内に、ソートされた順序でデータを保持します。このような行は、ソート順序を保持するために移動することがあります。たとえば、データの挿入によって既存の行が別のスロットまたは別のブロックに移動されることがあります。

B* ツリー索引のリーフには、主キーおよび実際の行データが保持されます。表データを変更すると（たとえば、新規行の追加や既存行の更新または削除など）、索引のみが更新されます。

> **参照：** B* ツリー索引の詳細は、『Oracle9*i* データベース概要』を参照してください。

索引構成表のメリット

索引構成表では、主キーを基にしたアクセスのために最適化された形式で行を格納するため、通常の表と比べて次のようなメリットがあります。

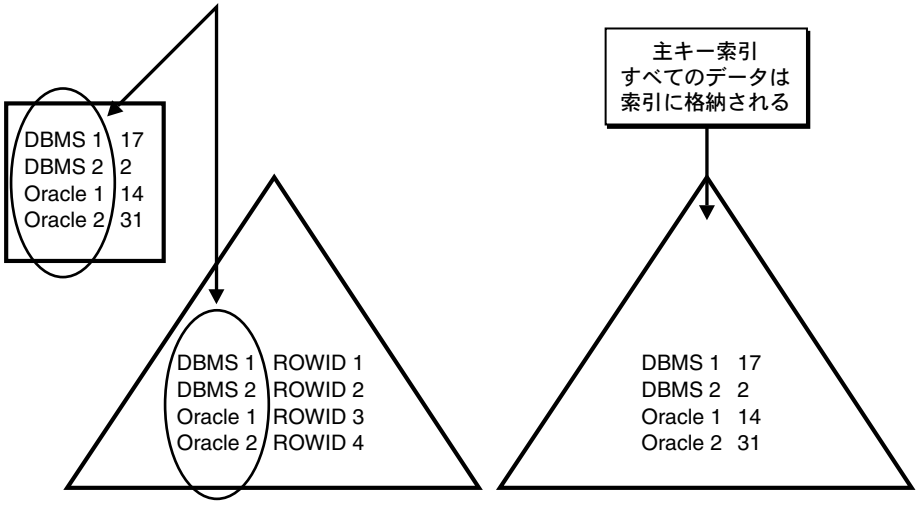
主キーの完全一致または範囲検索（あるいはその両方）を伴う問合せでの表データの高速アクセス　検索でキー値が見つかると、残りのデータはその位置に存在します。索引構成表は、ROWID をたどって表データに戻る I/O 操作が必要ありません。

24 × 7 体制（常時稼働）の操作での最善の表構成　データベースを常時使用可能にする必要がある場合、索引構成表には次のようなメリットがあります。

- 索引構成表または索引構成表のパーティションは、2 次索引を作成しなおさなくても、（領域のリカバリまたはパフォーマンスの改善の目的で）再編成できます。これによって、再編成のためのメンテナンス期間が短くなります。
- 索引構成表は、オンラインで再編成できます。2 次索引もオンラインで再編成できるため、再編成のためのメンテナンス期間がさらに短くなります。

記憶域必要量の削減　キー列は表と索引とで重複しないので、ROWID には追加記憶域が不要です。キー列が行の大部分を占有している場合、記憶域を最大で半分も節約できます。

図 6-1 索引付きの通常の表と索引構成表



索引構成表の機能

既存のデータを索引構成表に移動し、通常の表に対する操作と同じすべての操作を実行できます。次に、索引構成表の機能のいくつかを示します。

ALTER TABLE オプションの完全サポート　通常の表で使用できるすべての表変更オプションは、索引構成表に対しても使用できます。これには、ADD、MODIFY、DROP COLUMNS および DROP CONSTRAINTS が含まれます。ただし、索引構成表の主キー制約は、削除、遅延または使用禁止にできません。

論理 ROWID のサポート　B* ツリー索引内では行が移動するため、索引構成表に対する 2 次索引は、位置が固定されている物理 ROWID を基には作成できません。かわりに、索引構成表の 2 次索引は、論理 ROWID を基にしています。索引構成表の行には永続的な物理アドレスがなく、新しい行が挿入されたときはデータ・ブロックにまたがって移動できます。ただし、行の物理位置を変更しても、論理 ROWID は有効なまま残ります。

論理 ROWID には、表の主キー、および行が検出される可能性があるデータベース・ブロックのアドレスを示す物理推測が含まれます。この物理推測によって、不揮発性の索引構成表に対する ROWID アクセスが、通常の表に対する同様のアクセスと同等になります。

論理 ROWID は、次のような点で物理 ROWID と類似しています。

- 索引構成表から ROWID を選択し、WHERE 句に列名として ROWID を指定して行にアクセスできます。
- 論理 ROWID を介したアクセスは、複数のブロックにアクセスしたとしても、特定の行に最も速くたどりつく方法です。
- 行へのアクセスに使用される論理 ROWID は、行に対する主キー値が変更されないかぎり変更されません。

データベース・サーバーは、ユニバーサル ROWID という単一のデータ型を使用して、論理 ROWID と物理 ROWID の両方をサポートします。

索引構成表に切り替えるには、ROWID を使用しているアプリケーションをユニバーサル ROWID を使用するように変更する必要がありますが、UROWID データ型を使用できるため変更は簡単です。これによって、アプリケーションが統一された方法で論理 ROWID または物理 ROWID にアクセスできます。

> **参照：** 6-9 ページの「ユニバーサル ROWID データ型の宣言例」を参照してください。

2 次索引のサポート 索引構成表の 2 次索引は、通常の表の索引とは次の 2 つの点で異なります。

- 物理 ROWID ではなく論理 ROWID を格納します。このため、ALTER TABLE MOVE などの表のメンテナンス操作によって 2 次索引が使用できなくなることはありません。
- 論理 ROWID には、索引構成表の行を格納する索引リーフ・ブロックに直接アクセスするための物理推測も含まれます。物理推測が正しい場合、セカンダリ・キーが見つかった後は、2 次索引スキャンによって発生する追加 I/O は 1 回のみです。このときのパフォーマンスは、通常の表に対する 2 次索引スキャンと同じ程度になります。

ファンクション 2 次索引の他に、一意および非一意 2 次索引もサポートされています。索引構成表がマッピング表で作成されている場合は、パーティション化されていない索引構成表のビットマップ索引がサポートされます。マッピング表の詳細は、『Oracle9*i* データベース概要』の索引構成表に関する項を参照してください。

LOB 列 索引構成表には、オーディオ、ビデオ、イメージなど、構造化されていない大規模データを格納する内部 LOB 列および外部 LOB 列を作成できます。索引構成表内の LOB 列に対する SQL、DDL、DML およびピース単位の操作は、通常の表と同様の動作をします。主な違いは次のとおりです。

- 表領域マッピング : デフォルトでは（または、特にそれ以外の指定をしないかぎり）、LOB のデータおよび索引セグメントは、索引構成表の主キー索引セグメントが作成された表領域に作成されます。
- インライン記憶域および行外記憶域 : オーバーフロー・セグメント付きの索引構成表の LOB は、通常の表の LOB と同じものです。オーバー・フロー・セグメントなしで作成された索引構成表の LOB は、すべて行外に格納されます（デフォルトの記憶域属性は

DISABLE STORAGE IN ROW です）。このような LOB に ENABLE STORAGE IN ROW を指定すると、エラーになります。

LOB 列は、レンジ・パーティション化された索引構成表でサポートされます。

その他の LOB 機能（BFILE、一時 LOB、可変文字幅 LOB など）も、索引構成表でサポートされます。これらの機能は、通常の表の場合と同じように使用します。

パラレル問合せ　索引構成表に対する主キーの索引スキャンを伴う問合せは、パラレルに実行できます。

オブジェクトのサポート　索引構成表では、ほとんどのオブジェクト機能（オブジェクト型、VARRAY、ネストした表、REF 列など）がサポートされます。

SQL*Loader　このユーティリティは、索引構成表のロード（通常パスとダイレクト・パスの両方）およびそれに関連する索引（パーティション化サポートを含む）をサポートします。ただし、索引構成表に対するダイレクト・パスのパラレル・ロードはサポートされません。これと同じ結果を得るための他の方法として、SQL*Loader を使用して通常の表にパラレル・ロードを実行してから、パラレルの CREATE TABLE AS SELECT オプションを使用して索引構成表を作成します。

エクスポート / インポート　このユーティリティは、索引構成表がパーティション化されているかどうかにかかわらず、索引構成表のエクスポート（通常パスとダイレクト・パスの両方）およびインポートをサポートします。

分散データベースおよびレプリケーション・サポート　索引構成表は、パーティション化されているかどうかにかかわらずレプリケートできます。

他のツール　Oracle Enterprise Manager は、索引構成表に対する CREATE 操作および ALTER 操作に使用する SQL 文の生成をサポートします。

キーの圧縮　キーの圧縮によって、索引構成表および索引内で繰り返されるキー列接頭辞を排除できます。このスキームの主な特長は次のとおりです。

- キーの圧縮によって、索引キーが接頭辞エントリと接尾辞エントリに分割されます。圧縮は、索引ブロック内のすべての接尾辞エントリで、接頭辞エントリを共有することで行われます。
- B* ツリーのリーフ・ブロック内のキーのみが圧縮されます。B* ツリーのブランチ・ブロック内のキーの接尾辞は、切り捨てられますが、キー圧縮の対象にはなりません。

索引構成表の使用時期

通常の表ではなく、索引構成表を使用する方が適している場合がいくつかあります。

Oracle AQ の一部である場合　Oracle AQ は、データベース・サーバーの統合機能としてメッセージ・キューイングを提供し、索引構成表を使用して複数のコンシューマ・キューのメタデータ情報を保持します。

データの重複格納を回避する場合　ほとんどの列が主キーの表の場合、かなりの冗長なデータが格納されます。この重複格納は、索引構成表を使用して回避できます。また、索引構成表を使用することによって、キー列以外の列に対する主キー・ベースのアクセス効率も向上します。

VLDB アプリケーションおよび OLTP アプリケーションを開発する場合　索引構成表は列値の範囲でパーティション化できるため、VLDB アプリケーションに適しています。

索引構成表の主なメリットの 1 つは、2 次索引の論理性にあります。`ALTER TABLE MOVE` および `SPLIT` 操作の後でも、索引構成表の*グローバル*索引は、索引行に論理 ROWID が含まれているため、そのまま使用できます。このため、非常にコストのかかる索引の完全な再構築を回避できます。また、`ALTER TABLE MOVE` 操作はオンラインで実行できるため、索引構成表は 24 × 7 の可用性を必要とするアプリケーションにとって理想的な表になります。

同様に、`ALTER TABLE MOVE` 操作の後でも、索引構成表のローカル索引はそのまま使用できます。

パーティション・メンテナンス操作を実行すると、論理 ROWID の推測コンポーネントが無効になるため、索引構成表のローカル索引およびグローバル索引のパフォーマンスは低下します。ただし、論理 ROWID の主キー・コンポーネントを使用すると索引を使用できます。これら索引の無効な物理推測は、`ALTER INDEX ... UPDATE BLOCK REFERENCES` 操作によってオンラインで修正可能です。

時系列アプリケーションを開発する場合　時系列アプリケーションは、株価など単一の項目に属するタイムスタンプ付きの一連の行を使用します。索引構成表には主キーを基に行をクラスタ化できる機能があるため、時系列アプリケーションにとっては魅力的です。Oracle8 Time Series オプションでは、主キー（株記号、タイムスタンプ）で索引構成表を定義することによって、時系列データを効率的に格納し操作できます。キー圧縮付きの索引構成表を使用し、時系列内で繰り返し発生する項目識別子（たとえば、株記号）を圧縮すると、記憶域をさらに節約できます。

ネストした表を使用する場合　ネストした表の列の場合、ネストした表のすべての行を保持する記憶表が内部的に作成されます。

次に示すように、ネストした表を索引構成表として格納できます。

```
CREATE TYPE Project_t AS OBJECT(Pno NUMBER, Pname VARCHAR2(80));
CREATE TYPE Project_set AS TABLE OF Project_t;
```

```
CREATE TABLE Employees (Eno NUMBER, Projects PROJECT_SET)
   NESTED TABLE Projects_ntab STORE AS Emp_project_tab
               ((PRIMARY KEY(Nested_table_id, Pno)) ORGANIZATION INDEX)
               RETURN AS LOCATOR;
```

ネストした表の 1 つのインスタンスに属する行は、NESTED_TABLE_ID 列によって識別されます。ネストした表の列を格納するために通常の表を使用した場合は、通常、ネストした表の行のクラスタ化が解除されます。これに対して、索引構成表を使用した場合は、ネストした表の行を NESTED_TABLE_ID 列を基にクラスタ化できます。

拡張可能な索引データを使用する場合　拡張索引作成機能フレームワークでは、データベースに対して新しいアクセス方法を追加できます。通常、ドメイン固有の索引スキームには、索引データを保持するためのなんらかの格納方法が必要です。このようなドメインでの索引の格納には、索引構成表が理想的です。Oracle Spatial および Oracle Text では、索引データの格納に索引構成表を使用します。

索引構成表の例

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
GRANT CREATE TABLESPACE TO scott;
CONNECT scott/tiger
CREATE TABLESPACE Ind_tbs DATAFILE 'disk1:moredata2' SIZE 100K;
CREATE TABLESPACE Doc_tab DATAFILE 'disk1:moredata2' SIZE 100K;
CREATE TABLESPACE Ovf_tbs DATAFILE 'disk1:moredata3' SIZE 100K;
CREATE TABLESPACE Ind_ts0 DATAFILE 'disk1:moredata5' SIZE 100K
REUSE;
CREATE TABLESPACE Ov_ts0 DATAFILE 'disk1:moredata6' SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts1 DATAFILE 'disk1:moredata7' SIZE 100K
REUSE;
CREATE TABLESPACE Ov_ts1 DATAFILE 'disk1:moredata8' SIZE 100K REUSE;
CREATE TABLESPACE Ind_ts2 DATAFILE 'disk1:moredata9' SIZE 100K
REUSE;
CREATE TABLESPACE Ov_ts2 DATAFILE 'disk1:moredata10' SIZE 100K
REUSE;
CREATE TABLE Doc_tab (tok VARCHAR2(4),id  VARCHAR2(14),freq NUMBER);
```

この例には、索引構成表を作成し使用するための基本的な作業がいくつか示されています。この例では、テキスト検索エンジンは、特定の単語または語句を使用するすべての Web ページの記録を保持しており、検索問合せへの応答としてハイパーテキスト・リンクのリストを戻します。

この例には、次の作業が示されています。

通常の表から索引構成表への既存データの移動例

CREATE TABLE AS SELECT コマンドを使用すると、通常の表から既存データを索引構成表に移動できます。次の例では、docindex という索引構成表が doctable という通常の表から作成されます。

```
CREATE TABLE Docindex
   (   Token,
       Doc_id,
       Token_frequency,
       CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs
PARALLEL (DEGREE 2)
AS SELECT * from Doc_tab;
```

PARALLEL 句を使用すると、表の作成をパラレルに実行できることに注意してください。

索引構成表の作成例

索引構成表を作成するには、ORGANIZATION INDEX 句を使用します。次の例では、Web のテキスト検索エンジンとして通常使用される逆索引で、索引構成表が使用されています。

```
CREATE TABLE Docindex
   (  Token           CHAR(20),
      Doc_id          NUMBER,
      Token_frequency NUMBER,
      CONSTRAINT Pk_docindex PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs;
```

ユニバーサル ROWID データ型の宣言例

次の例は、UROWID データ型の宣言方法です。

```
DECLARE
     Rid UROWID;
BEGIN
     INSERT INTO Docindex VALUES ('Or80', 2, 30)
                 RETURNING Rowid INTO RID;
     UPDATE Docindex SET Token='Or81' WHERE ROWID = Rid;
END;
```

索引構成表に対する 2 次索引の作成例

索引構成表に 2 次索引を作成して、複数のアクセス・パスを提供できます。次の例は、(doc_id,token) に対する索引の作成方法です。

```
CREATE INDEX Doc_id_index on Docindex(Doc_id, Token);
```

次の例に示すとおり、この 2 次索引の使用によって doc_id の述語を含む問合せが効率的に処理されます。

```
SELECT Token FROM Docindex WHERE Doc_id = 1;
```

索引構成表の操作例

アプリケーションでは、索引構成表を通常の表の場合と同じように SELECT、INSERT、UPDATE または DELETE 操作用の標準 SQL 文を使用して操作します。たとえば、docindex 表は次のように操作できます。

```
INSERT INTO Docindex VALUES ('Oracle8.1', 3, 17);
SELECT * FROM Docindex;
UPDATE Docindex SET Token = 'Oracle8' WHERE Token = 'Oracle8.1';
DELETE FROM Docindex WHERE Doc_id = 1;
```

また、SELECT FOR UPDATE 文を使用して、索引構成表の行をロックすることもできます。これらのすべての操作は、結果的に主キーの B* ツリー索引を操作します。索引構成表に関連する問合せ操作および DML 操作は、このコストベースの方法で最適化されます。

オーバーフロー・データ・セグメントの指定例

すべての非キー列を、主キーの B* ツリー索引構造に格納することが常に適切であるとはかぎりません。これは次のような理由によります。

- 主キー索引に非キー列を 1 つ追加格納するたびに、B* ツリー索引のリーフ・ブロック内にある索引行のクラスタ密度が下がります。

  また次のような理由もあります。

- B* ツリーのリーフ・ブロックでは索引行を 2 行以上保持する必要があり、すべての非キー列を索引行の一部に入れることができない場合があります。

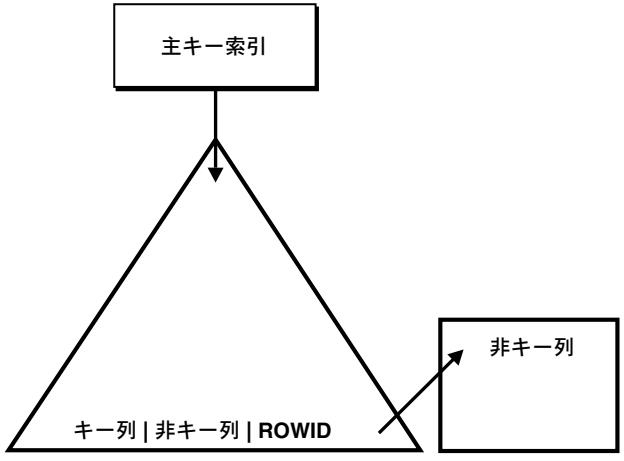
このような問題を解決するために、オーバーフロー・データ・セグメントを索引構成表に対応付けることができます。次の例では、docindex 表に token_offsets という追加列が必要です。この例は、索引構成表を作成する方法、および OVERFLOW オプションを使用してオーバーフロー・データ・セグメントを作成する方法です。

```
CREATE TABLE Docindex2
   (   Token           CHAR(20),
       Doc_id          NUMBER,
       Token_frequency NUMBER,
       Token_offsets   VARCHAR(512),
       CONSTRAINT Pk_docindex2 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs PCTTHRESHOLD 20
OVERFLOW TABLESPACE Ovf_tbs INITRANS 4;
```

オーバーフロー・データ・セグメントには、TABLESPACE、INITRANS などの物理記憶域属性を指定できます。

オーバーフロー・セグメントを持つ索引構成表の場合、索引行に 1 対の <key, row head> が含まれます。row head には、最初のいくつかの主キー列、および残りの列値を含むオーバーフロー行部分を指す ROWID が含まれます。この方法では、1 行に 1 つずつ ROWID を格納する必要がありますが、主キーの重複は回避できます。

図 6-2 オーバーフロー・セグメント



索引の行ヘッドに含める最後の非キー列の決定例

索引の行ヘッドに含める最後の非キー列を決定するには、リーフ・ブロック・サイズの割合として指定される PCTTHRESHOLD オプションを使用します。残りの非キー列は、オーバーフロー・データ・セグメントに 1 つ以上の行ピースとして格納されます。具体的には、行ヘッドに含める最後の非キー列は、索引行サイズ（キー + 行ヘッド）が指定されたしきい値（次の例では索引リーフ・ブロックの 20%）を超えないように選択されます。デフォルトでは、PCTTHRESHOLD は 50 に設定されます。

PCTTHRESHOLD オプションは、索引に含まれる最後の非キー列を行単位で決定します。ただし、このオプションでは、表の中のすべての行に対する索引に、この一連の同じ列を含めるように指定できません。指定するためには、INCLUDING オプションが提供されています。

次の例の CREATE TABLE 文は、token_frequency 列までのすべての列を索引リーフ・ブロックに入れ、token_offsets 列をオーバーフロー・セグメントに強制移動します。

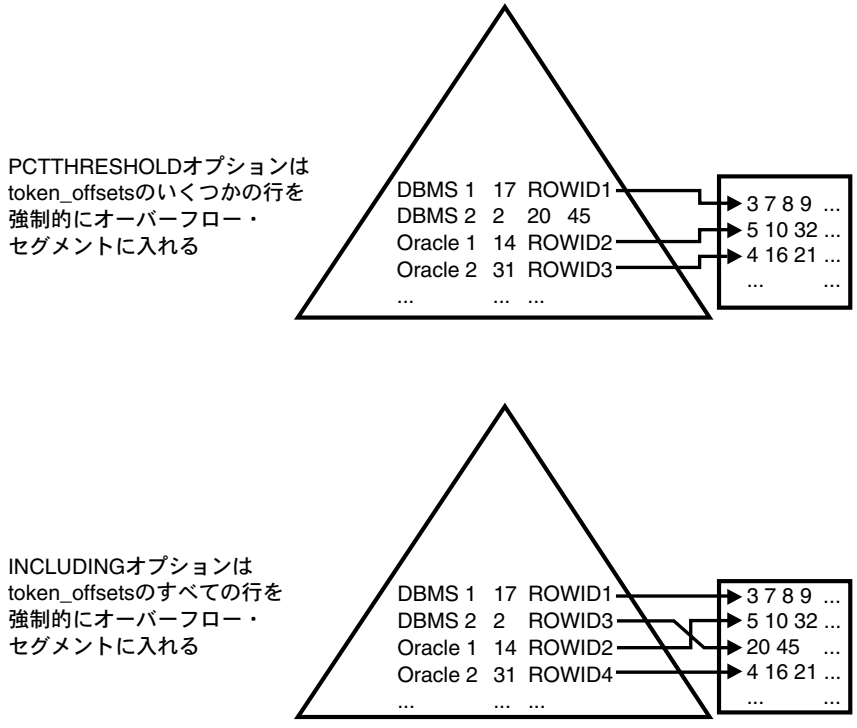
```
CREATE TABLE Docindex3
   (   Token           CHAR(20),
       Doc_id          NUMBER,
       Token_frequency NUMBER,
       Token_offsets   VARCHAR(512),
       CONSTRAINT Pk_docindex3 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs INCLUDING Token_frequency
OVERFLOW TABLESPACE Ovf_tbs;
```

このように、索引とデータ・セグメントとの間で行を垂直にパーティション化することによって、索引内での行のクラスタ密度が高くなります。これによって、索引内に格納されている列に対する問合せのパフォーマンスが向上します。たとえば、token_offsets 列があまり頻繁にアクセスされない場合は、この列を索引から外すことで、主キーの B* ツリー構造内で索引行のクラスタ化が進みます（図 6-3 を参照）。これによって、問合せの全体的なパフォーマンスも向上します。ただし、オーバーフロー・データ・セグメントに格納されている列には追加のブロック・アクセスが発生し、これでパフォーマンスが低下することもあります。

オーバーフロー・セグメントへの列の格納例

INCLUDING オプションによって、指定された列より後のすべての列がオーバーフロー・セグメントに格納されます。指定された INCLUDING 列に対応する索引行サイズが、指定されたしきい値を超える場合、含まれる最後の非キー列は PCTTHRESHOLD オプションに従って決定されます。

図 6-3 PCTTHRESHOLD 対 INCLUDING 列の使用



物理属性および格納属性の変更例

ALTER TABLE コマンドは、索引セグメントおよびオーバーフロー・データ・セグメントの物理属性および格納属性を変更する場合の他に、PCTTHRESHOLD 列値および INCLUDING 列値を変更する場合にも使用できます。次の例では、索引セグメントの INITRANS を 4 に、PCTTHRESHOLD を 20 に設定し、オーバーフロー・データ・セグメントの INITRANS を 6 に設定します。変更された値は、表に対するその後の操作で使用されます。

```
ALTER TABLE Docindex INITRANS 4 PCTTHRESHOLD 20 OVERFLOW INITRANS 6;
```

オーバーフロー・データ・セグメントなしで作成された索引構成表の場合、ALTER TABLE ADD OVERFLOW オプションを使用してオーバーフロー・データ・セグメントを追加できます。次の例は、docindex 表にオーバーフロー・セグメントを追加する方法です。

```
ALTER TABLE Docindex ADD OVERFLOW;
```

索引構成表の分析例

索引構成表は、通常の表の場合と同じように ANALYZE コマンドを使用して分析します。次の例は、docindex 表を分析する方法です。

```
ANALYZE TABLE Docindex COMPUTE STATISTICS;
```

ANALYZE コマンドを使用すると、主キー索引セグメントとオーバーフロー・データ・セグメントの両方が分析され、表の物理統計のみでなく論理統計も計算されます。また、ANALYZE LIST CHAINED ROWS オプションを使用すると、1 つ以上の連鎖オーバーフロー行ピースを持つ行の数を判断できます。論理 ROWID 機能を使用すると、別の CHAINED_ROWS 表は不要です。

索引構成表のロード、エクスポート、インポートまたはレプリケート

SQL*Loader で通常のパスまたはダイレクト・パスを使用すると、データはパーティション化されていない索引構成表にもパーティション化された索引構成表にもロードできます。また、エクスポート / インポート・ユーティリティを使用すると、データをエクスポートまたはインポートできます。さらに、索引構成表は、通常の表と同じように、分散データベースにレプリケートすることもできます。

索引構成表のパーティション化例

索引構成表は、列値の範囲、または列の集合から導出されるハッシュ値ごとにパーティション化できます。パーティション化列の集合は、主キー列のサブセットである必要があります。DML 操作中に主キーの一意性を判断するために検索するパーティションは 1 つのみになります。これによって、パーティションの独立性が保たれます。

パーティション化された索引構成表の主な特長は次のとおりです。

- 表レベル属性の一部として索引構成表を作成するには、ORGANIZATION INDEX 句を指定する必要があります。このプロパティは、すべてのパーティションに暗黙的に継承されます。
- オーバーフロー・データ・セグメント付きの索引構成表を作成するには、表レベル属性の一部として OVERFLOW オプションを指定する必要があります。
- OVERFLOW オプションの指定によって、オーバーフロー・データ・セグメントが作成されます。このセグメントは、それ自体が主キー索引セグメントで同一レベル・パーティション化されています。つまり、各パーティションには、索引セグメントおよびオーバーフロー・データ・セグメントが含まれます。
- ハッシュ・パーティション化された表には、CREATE TABLE 文の ROW MOVEMENT ENABLE 句を指定します。ハッシュ値の導出に使用される列が変更されると、行がパーティション間で移動される場合があります。
- 通常のパーティション表の場合と同じように、表レベルの物理属性にデフォルト値を指定できます。これらの値は、パーティションごとに（索引セグメントおよびオーバーフロー・データ・セグメントの両方で）オーバーライドできます。

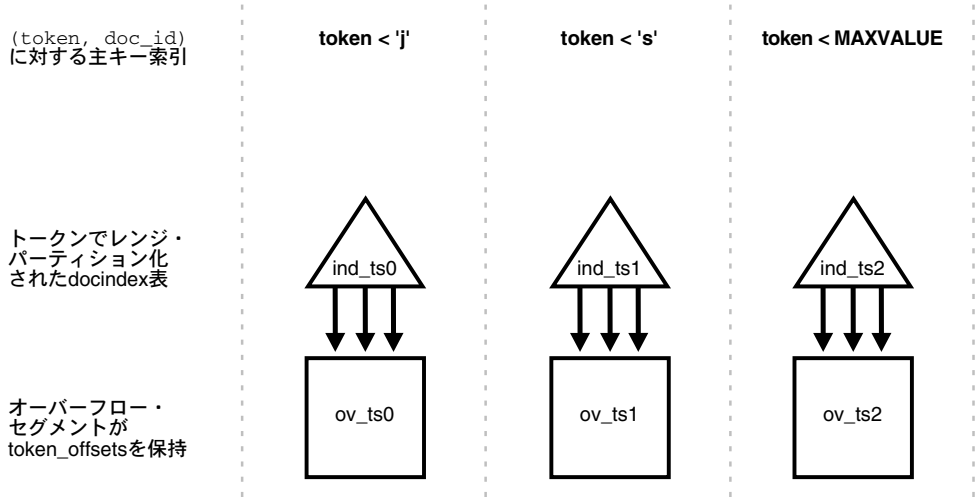
- 索引セグメントの表領域は、パーティション用に指定されていない場合は、表レベルのデフォルトに設定されます。表レベルのデフォルトが指定されていない場合は、そのユーザーのデフォルト表領域が使用されます。
- PCTTHRESHOLD 列および INCLUDING 列のデフォルト値は、表レベルでのみ指定できます。
- OVERFLOW キーワードの前に指定されるすべての属性は、主キー索引セグメントに適用されます。OVERFLOW キーワードの後に指定されるすべての属性は、オーバーフロー・データ・セグメントに適用されます。
- オーバーフロー・データ・セグメントの表領域は、パーティション用に指定されていない場合は、表レベルのデフォルトに設定されます。表レベルのデフォルトが指定されていない場合は、対応するパーティションの索引セグメントの表領域が使用されます。

次の例は、docindex 表の例の続きです。トークン値のレンジ・パーティション化を示します。

```
CREATE TABLE Docindex4
   (Token           CHAR(20),
    Doc_id          NUMBER,
    Token_frequency NUMBER,
    Token_offsets   VARCHAR(512),
    CONSTRAINT Pk_docindex4 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX INITRANS 4  INCLUDING Token_frequency
OVERFLOW INITRANS 6
      PARTITION BY RANGE(token)
          ( PARTITION P1 VALUES LESS THAN ('j')
TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
   PARTITION P2 VALUES LESS THAN ('s')
TABLESPACE Ind_ts1 OVERFLOW TABLESPACE Ov_ts1,
  PARTITION P3 VALUES LESS THAN (MAXVALUE)
TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2);
```

この例から、図 6-4 に示す表が作成されます。INCLUDING 句によって、token_offsets 列が各パーティションのオーバーフロー・データ・セグメントに格納されます。

図 6-4 レンジ・パーティション化されたオーバーフロー・セグメント付きの索引構成表



索引構成表に対するパーティション索引がサポートされます。索引構成表では、ローカル接頭辞、ローカル非接頭辞およびグローバル接頭辞によるパーティション索引がサポートされます。唯一の違いは、このような索引には物理 ROWID ではなく、論理 ROWID が格納されることです。

パーティション化された索引構成表では、すべての ALTER TABLE 操作を使用できます。ただし、これらの操作は、通常の表で実行される場合と比べてその動作に少し違いがあります。

- ALTER TABLE MOVE パーティション操作では、索引に論理 ROWID が含まれているため、すべての（ローカル、グローバルおよび非パーティション化）索引は USABLE のまま残ります。ただし、論理 ROWID に格納されている推測は無効になります。
- SPLIT パーティション操作では、すべての非パーティション化索引またはグローバル索引パーティションは使用可能のまま残ります。
- ALTER TABLE EXCHANGE パーティション操作の場合、ターゲット表が索引構成表と同等である必要があります。
- ユーザーは、ALTER TABLE ADD OVERFLOW コマンドを使用してオーバーフロー・セグメントを追加し、表レベルのデフォルトとパーティション・レベルの物理属性および格納属性を指定できます。この操作の結果、オーバーフロー・データ・セグメントが各パーティションに追加されます。

ALTER INDEX 操作は、通常の表の場合と非常に似ています。唯一の違いは、索引全体を再作成する操作（ALTER INDEX REBUILD および SPLIT_PARTITION）を実行すると、論理

ROWID の一部として格納されている推測が再作成されることです。新しい ALTER INDEX UPDATE BLOCK REFERENCES 構文は、索引を再作成せずに無効な物理推測を修正します。

パーティション化された索引構成表に対する問合せおよび DML 操作は、通常のパーティション表と同様に機能します。

索引構成表のキーの圧縮例

キーの圧縮を使用可能にするには、索引セグメントに物理属性を指定するときに COMPRESS 句を使用します。接頭辞の長さ（列数）を指定して、キーを接頭辞および接尾辞にどのように分割するかを指定できます。接頭辞の長さは、1 ～（主キー列の数 -1）に指定できます。

```
CREATE TABLE Docindex5
   ( Token           CHAR(20),
     Doc_id          NUMBER,
     Token_frequency NUMBER,
     Token_offsets   VARCHAR(512),
     CONSTRAINT pk_docindex5 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX TABLESPACE Ind_tbs COMPRESS 1 INCLUDING Token_frequency
OVERFLOW TABLESPACE Ovf_tbs;
```

長さが 1 の共通接頭辞（token 列）は、主キー（token, doc_id）項目に圧縮されます。主キー値のリストが（'DBMS', 1）、（'DBMS', 2）、（'Oracle', 1）、（'Oracle', 2）の場合は、DBMS および Oracle という繰返し項目が圧縮されます。

接頭辞の長さを指定しない場合は、デフォルトで、（主キー列の数 -1）が設定されます。索引構成表を作成する場合、または ALTER TABLE MOVE を使用して索引構成表を移動する場合は、圧縮オプションを指定できます。たとえば、次のように圧縮を使用禁止にできます。

```
ALTER TABLE Docindex5 MOVE NOCOMPRESS;
```

同様に、通常の表の索引および索引構成表の索引を、COMPRESS オプションを使用して圧縮できます。

パーティション化された索引構成表のキーの圧縮例 パーティション化された索引構成表のキーは、COMPRESS 句を表レベルのデフォルトの一部として指定しても圧縮できます。圧縮は、パーティションごとに使用可能または使用禁止にできます。接頭辞の長さはパーティション・レベルでは変更できません。

```
CREATE TABLE Docindex6
   ( Token           CHAR(20),
     Doc_id          NUMBER,
     Token_frequency NUMBER,
     Token_offsets   VARCHAR(512),
     CONSTRAINT Pk_docindex6 PRIMARY KEY (Token, Doc_id)
   )
ORGANIZATION INDEX INITRANS 4 COMPRESS 1 INCLUDING Token_frequency
```

```
OVERFLOW INITRANS 6
            PARTITION BY RANGE(Token)
           ( PARTITION P1 VALUES LESS THAN ('j')
TABLESPACE Ind_ts0 OVERFLOW TABLESPACE Ov_ts0,
   PARTITION P2 VALUES LESS THAN ('s')
TABLESPACE Ind_ts1 NOCOMPRESS OVERFLOW TABLESPACE Ov_ts1,
  PARTITION P3 VALUES LESS THAN (MAXVALUE)
TABLESPACE Ind_ts2 OVERFLOW TABLESPACE Ov_ts2
);
```

接頭辞の長さについては、すべてのパーティションが表レベルのデフォルトを継承します。パーティション P1 および P3 は、キーの圧縮が使用可能な状態で作成されています。パーティション P2 の場合は、キーの圧縮はパーティション・レベルの NOCOMPRESS オプションによって使用禁止にされています。

ALTER TABLE MOVE 操作および SPLIT 操作では、COMPRESS オプションを変更できます。次の例では、キーの圧縮を使用可能にしてパーティションを再作成します。

```
ALTER TABLE Docindex6 MOVE PARTITION P2 COMPRESS;
```

索引構成表の再作成例

SQL コマンド ALTER TABLE MOVE を使用すると、表を再作成できます。このコマンドは、大量の挿入、更新または削除によって、索引構成表を含む B* ツリー構造が断片化されたときに使用します。MOVE オプションは、主キーの B* ツリー索引を再作成します。

デフォルトでは、次の場合以外はオーバーフロー・データ・セグメントは再作成されません。

- OVERFLOW 句が明示的に指定されている場合
- PCTHRESHOLD 列または INCLUDING 列（あるいはその両方）の値が、MOVE 文の一部として変更された場合
- すべての LOB が明示的に移動される場合

デフォルトでは、索引セグメントおよびデータ・セグメントに関連する LOB 列は再作成されません。ただし、LOB 列が MOVE 文の一部として明示的に指定されている場合は再作成されます。次の例では、索引ブロックの INITRANS を 6 に設定した後、表データを含む B* ツリー索引を再作成します。

```
ALTER TABLE docindex MOVE INITRANS 6;
```

次の例では、主キー索引とオーバーフロー・データ・セグメントの両方を再作成します。

```
ALTER TABLE docindex MOVE TABLESPACE Ovf_tbs OVERFLOW TABLESPACE ov_ts0;
```

デフォルトでは、移動中の表を他の操作で使用できません。ただし、ONLINE オプションを使用すると索引構成表を移動できます。次の例では、実際の移動操作中に、表に対する

DML 操作および問合せ操作が可能になります。この機能によって、索引構成表は 24 × 7 の可用性が必要なアプリケーションに適しています。

注意： 次の文を実行するには、COMPATIBLE 初期化パラメータを 8.1.3.0 以上に設定する必要がある場合があります。

```
ALTER TABLE Docindex MOVE ONLINE;
```

ONLINE 移動がサポートされているのは、オーバーフロー・セグメントを持たない索引構成表のみです。

7

Oracle における SQL 文の処理方法

この章では、Oracle における SQL 文の処理方法について説明します。内容は次のとおりです。

Oracle のツール製品および Applications の中には、SQL の使用を簡素化または隠蔽しているものがありますが、Oracle に組み込まれているセキュリティおよびデータ整合性の機能を利用するために、すべてのデータベース操作は SQL を使用して実行されます。

SQL 文実行の概要

図 7-1 に、SQL 文の処理および実行に通常使用される手順の概要を示します。これらの手順が少し異なる順序で実行される場合もあります。たとえば、コードの作成方法によっては、DEFINE 手順が FETCH 手順の直前に実行される場合もあります。

Oracle のツール製品の多くでは、いくつかの手順は自動的に実行されるため、ユーザーがこのような詳細事項を考慮または認識する必要はほとんどありません。ただし、アプリケーションを作成するときに、この情報が有効な場合もあります。

> **参照：** 各種 SQL 文に対する SQL 処理手順については、『Oracle9*i* データベース概要』を参照してください。

SQL92 に対する拡張の識別（FIPS フラグ付け）

SQL に関する連邦情報処理標準（FIPS 127-2）では、ベンダー提供の拡張機能を使用する SQL 文を識別する方法が必要です。Oracle では、移植性のあるアプリケーションの作成に有効な FIPS フラガーを提供しています。

FIPS フラグ付けがアクティブな場合は、SQL 文を確認して、ANSI/ISO SQL92 規格外の拡張が含まれていないかどうかを確認します。規格外の構造体が見つかると、Oracle サーバーはそれにエラーのフラグを付け、違反している構文を表示します。

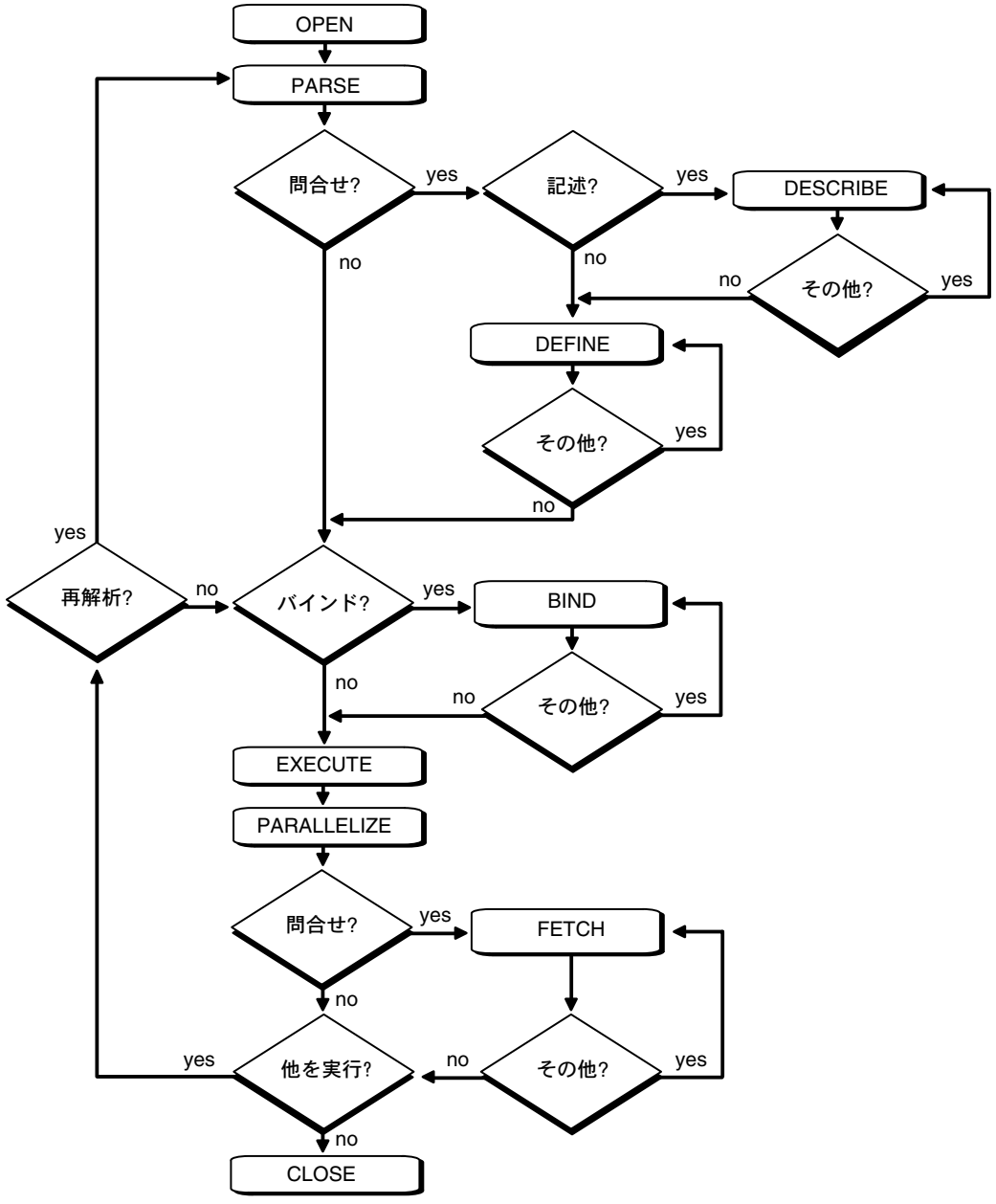
FIPS フラグ付け機能では、Enterprise Manager または SQL*Plus を使用して送信される対話型 SQL 文を介して、フラグ付けがサポートされています。また、Oracle プリコンパイラおよび SQL*Module では、埋込み SQL およびモジュール言語 SQL の FIPS フラグ付けもサポートされています。

フラグ付けがオンの場合に規格外の SQL が検出されると、次のメッセージが戻されます。

```
ORA-00097: Oracle SQL 機能は、SQL92 level レベルでは使用できません。
```

level は、ENTRY、INTERMEDIATE、FULL のいずれかです。

図 7-1 SQL 文の処理手順



操作のトランザクションへのグループ化

一般に、Oracle のプログラム・インタフェースを使用するアプリケーション設計者のみが、どのタイプのアクションを 1 つのトランザクションとしてグループ化する必要があるかということに関心を持っています。トランザクションは、論理単位ごとに作業が完成し、データの一貫性が保たれるように、正しく定義する必要があります。トランザクションは、1 つの論理作業単位に必要なすべての部分で構成される必要があります。これより多くても少なくてもいけません。すべての参照表の中のデータは、トランザクションが開始する前および終了した後で、一貫した状態である必要があります。トランザクションは、データに対する一貫した変更を 1 つ含む SQL 文または PL/SQL ブロックのみで構成する必要があります。

たとえば、2 つの口座間の預金の移動（トランザクションまたは論理作業単位）には、一方の口座の借方への記帳（1 つの SQL 文）ともう一方の口座の貸方への記帳（1 つの SQL 文）が含まれています。2 つの記帳で 1 つの作業単位となり、両方が成立するか両方が不成立かのいずれかです。つまり、借方を伴わない貸方はコミットしてはいけません。1 つの口座への新規預金など、関連のないアクション処理は、預金移動トランザクションに入れてはいけません。

トランザクションのパフォーマンスの改善

アプリケーションを設計するときには、トランザクションを形成するアクションのタイプを決定する他に、パフォーマンスの改善のためになんらかの処置を講じられるかどうかも判断する必要があります。アプリケーションの設計および作成では、次のパフォーマンス要件を考慮する必要があります。特に指定がない場合、詳細は、『Oracle9*i* データベース概要』を参照してください。

- `BEGIN_DISCRETE_TRANSACTION` プロシージャを使用して、短い非分散型トランザクションのパフォーマンスを改善します。
- `USE ROLLBACK SEGMENT` パラメータを指定した `SET TRANSACTION` コマンドを使用して、トランザクションを適切なロールバック・セグメントに明示的に割り当てます。これによって、システム全体のパフォーマンスを低下させる可能性のある追加エクステントの動的割当ての必要がなくなります。
- `ISOLATION LEVEL` を `SERIALIZABLE` に指定した `SET TRANSACTION` コマンドを使用して、ANSI/ISO シリアル化可能トランザクションを生成します。

  参照：

  - 7-22 ページの「シリアル化可能トランザクションの相互作用」を参照してください。
  - 『Oracle9*i* データベース概要』を参照してください。

- 共有 SQL 領域を利用できるように、SQL 文の発行基準を確立します。Oracle が同一 SQL 文を認識し、SQL 文がメモリー領域を共有できるようにします。これによって、データベース・サーバー上のメモリー使用量が減少し、システム・スループットが向上します。

- ANALYZE コマンドを使用して統計を収集し、SQL 文を最適化するコストベースの方法を可能にします。オプティマイザには、必要に応じてヒントを追加できます。
- トランザクションを開始する前に DBMS_APPLICATION_INFO.SET_ACTION プロシージャをコールし、トランザクションを登録して名前を付け、アプリケーション全体にわたるパフォーマンス測定で使用できるようにします。後でシステムをチューニングする際に、どのトランザクションが 1 番多くシステム・リソースを必要とするかがわかるように、トランザクションで実行するアクションのタイプを指定する必要があります。
- 9-46 ページの「SQL 式からのストアド・ファンクションのコール」に説明するとおり、ユーザーが作成した PL/SQL ファンクションを SQL 式に組み込んで、ユーザーの生産性および問合せ効率を向上させます。
- PL/SQL アプリケーションを作成するときに、明示的カーソルを作成します。
- プリコンパイラ・プログラムを作成するときに、MAX_OPEN_CURSORS を使用してカーソルの数を増加させると、解析頻度が削減され、パフォーマンスが改善されることがよくあります。

  参照： 7-8 ページの「アプリケーションでのカーソルの使用」を参照してください。

トランザクションのコミット

トランザクションをコミットするには、COMMIT コマンドを使用します。次の 2 つの文は同等で、現行のトランザクションをコミットします。

```
COMMIT WORK;
COMMIT;
```

COMMIT コマンドには、コミットされるトランザクションに関する情報を示すコメント（49 文字以下）を指定した COMMENT パラメータを含めることができます。このオプションは、分散トランザクションをコミットするときに、トランザクションの起点に関する情報を含める場合に有効です。

```
COMMIT COMMENT 'Dallas/Accts_pay/Trans_type 10B';
```

参照： インダウト分散トランザクションのコミットの詳細は、『Oracle8 分散システム』を参照してください。

トランザクションのロールバック

トランザクションの全体または一部を（セーブポイントまで）ロールバックするには、ROLLBACK コマンドを使用します。たとえば、次の文はどちらも、現行のトランザクション全体をロールバックします。

```
ROLLBACK WORK;
ROLLBACK;
```

ROLLBACK コマンドの WORK オプションには、何も機能はありません。

現行のトランザクション内に定義されたセーブポイントまでロールバックするには、ROLLBACK コマンドの TO オプションを使用する必要があります。たとえば、次の文はいずれも POINT1 という名前の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。

```
SAVEPOINT Point1;
...
ROLLBACK TO SAVEPOINT Point1;
ROLLBACK TO Point1;
```

> **参照：** インダウト分散トランザクションのロールバックの詳細は、『Oracle8 分散システム』を参照してください。

トランザクションのセーブポイントの定義

トランザクション内に**セーブポイント**を定義するには、SAVEPOINT コマンドを使用します。次の文は、現行のトランザクション内に ADD_EMP1 という名前のセーブポイントを作成します。

```
SAVEPOINT Add_emp1;
```

前のセーブポイントと同じ識別子で 2 番目のセーブポイントを作成すると、前のセーブポイントが消去されます。セーブポイントを作成した後は、そのセーブポイントまでロールバックできます。

セッション当たりのアクティブ・セーブポイントの数に制限はありません。アクティブ・セーブポイントとは、最後のコミットまたは最後のロールバック以降に指定されたセーブポイントのことです。

COMMIT、SAVEPOINT および ROLLBACK の例

次の一連の SQL 文で、トランザクション内での COMMIT 文、SAVEPOINT 文および ROLLBACK 文の使用方法を具体的に説明します。

SQL 文	結果
SAVEPOINT a;	このトランザクションの最初のセーブポイント
DELETE...;	このトランザクションの最初の DML 文
SAVEPOINT b;	このトランザクションの 2 番目のセーブポイント
INSERT INTO...;	このトランザクションの 2 番目の DML 文
SAVEPOINT c;	このトランザクションの 3 番目のセーブポイント
UPDATE...;	このトランザクションの 3 番目の DML 文

SQL 文	結果
ROLLBACK TO c;	UPDATE 文がロールバックされ、セーブポイント C は定義されたままになります。
ROLLBACK TO b;	INSERT 文がロールバックされ、セーブポイント C は失われます。セーブポイント B は定義されたままです。
ROLLBACK TO c;	ORA-01086 エラー。セーブポイント C は設定されていません。
INSERT INTO...;	このトランザクションの新しい DML 文
COMMIT;	最初の DML 文（DELETE 文）および最後の DML 文（2 番目の INSERT 文）によって行われたすべてのアクションがコミットされます。 トランザクションのその他すべての文（2 番目および 3 番目の文）は COMMIT の前にロールバックされています。セーブポイント A は、すでにアクティブではありません。

トランザクションの管理に必要な権限

自身のトランザクションを制御するときに権限は不要です。どのユーザーでもトランザクション内で COMMIT 文、ROLLBACK 文および SAVEPOINT 文を発行できます。

読込み専用トランザクションでのリピータブル・リードの保証

特に何も指定しない場合、Oracle の一貫性モデルは、文レベルの読込み一貫性は保証しますが、トランザクション・レベルの読込み一貫性（リピータブル・リード）は保証しません。トランザクション・レベルの読込み一貫性が必要で、トランザクションが更新を必要としない場合は、**読込み専用トランザクション**を指定できます。トランザクションを読込み専用に指定した後、読込み専用トランザクションの問合せ結果が、特定の時点での一貫性を保持していることを確認すると、問合せを任意のデータベース表に対して必要な回数だけ実行できます。

読込み専用トランザクションでは、トランザクション・レベルの読込み一貫性を提供するための追加データ・ロックは取得されません。文レベルの読込み一貫性のために使用される複数バージョンの一貫性モデルが、トランザクション・レベルの読込み一貫性を提供するために使用されます。すべての問合せは、読込み専用トランザクションが開始したときに決定されたシステム制御番号（SCN）に関連する情報を戻します。データ・ロックが取得されていないため、読込み専用トランザクションが問い合せているデータを他のトランザクションが同時に問い合せたり更新することができます。

読込み専用トランザクションによって問合せを受けた変更済データ・ブロックは、ロールバック・セグメントのデータを使用して再構成されます。そのため、長時間実行される読込み専用トランザクションでは、「ORA-01555: スナップショットが古すぎます : ロールバック・セグメント番号 *string*、名前 *string* が小さすぎます」というエラーが戻されることがあ

ります（*string* には文字列が入ります）。この問題を回避するには、さらに多くのロールバック・セグメントを作成するか、さらに大きなロールバック・セグメントを作成します。また、オンライン・トランザクション処理が最小であるときに実行時間の長い問合せを発行するか、または問合せ前に表に対して共有ロックを取得して、トランザクション中のその他すべての更新を禁止することもできます。

読込み専用トランザクションは、READ ONLY オプションを含む SET TRANSACTION 文で始まります。次に例を示します。

```
SET TRANSACTION READ ONLY;
```

SET TRANSACTION 文は、新しいトランザクションの最初の文である必要があります。任意の DML 文（問合せを含む）または他の DDL 以外の文（たとえば、SET ROLE）が SET TRANSACTION READ ONLY 文より前に指定されていると、エラーが戻されます。SET TRANSACTION READ ONLY 文が正常に実行されると、そのトランザクションでは、SELECT 文（FOR UPDATE 句なし）、COMMIT 文、ROLLBACK 文または DML 以外の文（たとえば、SET ROLE、ALTER SYSTEM、LOCK TABLE）のみが使用できます。これら以外の文では、エラーが戻されます。COMMIT 文、ROLLBACK 文または DDL 文によって、読込み専用トランザクションは終了します（DDL 文によって、読込み専用トランザクションは暗黙的にコミットされ、独自のトランザクション内でコミットされます）。

アプリケーションでのカーソルの使用

PL/SQL では、1 行のみ戻す問合せも含めて、すべての SQL データ操作文に対してカーソルを暗黙的に宣言します。複数行を戻す問合せの場合、カーソルを明示的に宣言して行を別々に処理できます。

カーソルは、特定のプライベート SQL 領域へのハンドルです。カーソルは、特定のプライベート SQL 領域の名前であると考えられます。PL/SQL **カーソル変数**を使用すると、ストアド・プロシージャから複数の行を取得できます。カーソル変数を使用すると、3GL アプリケーション内のパラメータとしてカーソルを渡すことができます。カーソル変数の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

ほとんどの Oracle ユーザーは、Oracle ユーティリティの自動カーソル処理を使用しますが、アプリケーション設計者は、プログラム・インタフェースを使用した方がカーソルを制御しやすくなります。アプリケーション開発では、カーソルはプログラムで使用できる名前付きのリソースであり、アプリケーションに埋め込まれた SQL 文の解析に特に有効です。

カーソルの宣言およびオープン

1 つのセッションで同時にオープンできるカーソルの総数に絶対的な制限はありませんが、次の 2 つの制約があります。

- 各カーソルは仮想メモリーを必要とするため、セッションの総カーソル数は、プロセスで使用できるメモリーによって制限されます。

- セッションごとのカーソル数に関するシステム全体の上限は、パラメータ・ファイル（INIT.ORA など）内の OPEN_CURSORS 初期化パラメータによって設定されます。

  参照： パラメータの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

プリコンパイラ・プログラムに対して明示的にカーソルを作成すると、アプリケーションのチューニング時に有効です。たとえば、カーソル数を増加させると、解析頻度が削減されパフォーマンスが改善されることがよくあります。ある時点で必要となるカーソル数がわかっている場合、その数のカーソルを必ず同時にオープンできるように宣言できます。

カーソルを使用した文の再実行

各ステージの実行後、カーソルはその SQL 文に関する十分な情報を保持しているため、同じカーソルに他の SQL 文が対応付けられていないかぎり、最初から実行しなおさなくてもその文を再実行できます。これについては、図 7-1 に示してあります。SQL 文は、解析手順を含めなくても再実行できます。

カーソルをいくつかオープンすることによって、いくつかの SQL 文の解析済表現を保存できます。同じ SQL 文を繰り返し実行する場合、記述手順、定義手順、バインド手順または実行手順から開始することができ、カーソルのオープンおよび解析を繰り返す必要がなくなります。

あるカーソルのパフォーマンス特性を理解するために、DBA は、V$SQL カタログ・ビューを使用して、そのカーソルが表現する問合せのテキストを取得できます。元の問合せに対する実行計画の結果は、問合せの実際の処理方法とは異なる場合があります。そのため DBA は、V$SQL_PLAN および V$SQL_PLAN_STATS カタログ・ビューを調べることによってより正確な情報を取得できます。V$SQL_PLAN_ENV カタログ・ビューは、実行計画の出力結果およびカーソルの実際の実行計画との違いの原因になった可能性のある、デフォルト値から変更されたパラメータを示します。

参照： これらの各カタログ・ビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

カーソルのクローズ

カーソルのクローズとは、関連付けられているプライベート領域に現在ある情報が失われ、そのメモリーの割当てが解除されることです。カーソルは、一度オープンされると次のいずれかが発生するまでクローズされません。

- ユーザー・プログラムがサーバーとの接続を終了した場合。
- ユーザー・プログラムが OCI プログラムまたはプリコンパイラ・アプリケーションのとき、そのプログラムの実行中に宣言済のカーソルを明示的にクローズした場合（ただし、これらのプログラムの終了時にオープンしたままのカーソルがあると、カーソルは暗黙的にクローズされます）。

カーソルの取消し

カーソルを取り消すと、現在のフェッチからリソースが解放されます。対応付けられたプライベート領域に現在ある情報は失われますが、カーソルはオープンしたままになり、解析され、バインド変数に対応付けられます。

注意： Pro*C または PL/SQL を使用してカーソルを取り消すことはできません。

参照： カーソルの取消しの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

明示的なデータのロック

Oracle は、データの並行性と整合性、および文レベル読込み一貫性を保証するために、常に必要なロックを実行します。これらのデフォルト・ロック・メカニズムはオーバーライドできます。たとえば、次のような場合は、Oracle のデフォルト・ロックをオーバーライドした方が効果的です。

- トランザクション・レベルの読込み一貫性（リピータブル・リード）が必要な場合。この場合、トランザクションは、そのトランザクションの存続中は一貫した一連のデータを問い合せます。読み込まれるデータが他のトランザクションによって変更されていないことを確認する必要があります。トランザクション・レベルの読込み一貫性を実現するには、明示的なロック、読込み専用トランザクション、シリアル化可能トランザクションまたはシステムのデフォルト・ロックのオーバーライドを使用します。
- リソースに対して排他的にアクセスするトランザクションが必要な場合。リソースに排他的にアクセスが可能なトランザクションは、文の処理時に他のトランザクションの完了まで待機しません。

自動ロック・メカニズムは、次の 2 つの異なるレベルでオーバーライドできます。

明示的ロックの型	使用方法
トランザクション・レベル	LOCK TABLE コマンド、FOR UPDATE 句を含む SELECT コマンド、および READ ONLY オプションまたは ISOLATION LEVEL SERIALIZABLE オプション付きの SET TRANSACTION コマンドを含むトランザクションは、Oracle のデフォルト・ロックをオーバーライドします。これらの文によって取得されるロックは、トランザクションのコミット後またはロールバック後に解除されます。

明示的ロックの型	使用方法
システム・レベル	初期化パラメータ SERIALIZABLE および ROW_LOCKING を調整することによって、デフォルト以外のロックでインスタンスを開始できます。

次の項では、Oracle のデフォルト・ロックをオーバーライドするために使用できる各オプションを説明します。DML_LOCKS 初期化パラメータによって、使用可能な DML ロックの最大数が決定されます。

> **参照：** パラメータの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

通常、デフォルト値には十分な値が設定されていますが、手動ロックを追加して使用する際に、この値を大きくする必要がある場合があります。

> **注意： いずれかのレベルで Oracle のデフォルト・ロックをオーバーライドする場合は、新しいロック手順が正しく動作することを確認する必要があります。データ整合性が保証されていること、データ並行性が許容されていること、およびデッドロックの可能性がないことまたはデッドロックが適切に処理されていることを確認してください。**

ロック方法の選択

LOCK TABLE 文が実行されると、トランザクションは指定された表ロックを明示的に取得します。LOCK TABLE 文は、デフォルト・ロックを手動でオーバーライドします。ビューに対して LOCK TABLE 文が発行されると、基礎となる実表がロックされます。次の文は、EMP_TAB 表および DEPT_TAB 表を含むトランザクションにかわって、この 2 つの表に対する排他表ロックを取得します。

```
LOCK TABLE Emp_tab, Dept_tab
    IN EXCLUSIVE MODE NOWAIT;
```

ロック・モードが同じ場合は、ロックする表またはビューを複数指定できます。ただし、1 つの LOCK TABLE 文に指定できるロック・モードは 1 つのみです。

> **注意：** 表がロックされると、その表のすべての行がロックされます。他のユーザーは、その表を変更できません。

また、ロックの取得を待つか待たないかも指示できます。NOWAIT オプションを指定すると、表ロックがすぐに使用可能である場合にのみ表ロックを取得します。すぐに使用可能でない場合は、その時点ではロックが使用できないことを示すエラーが戻されます。その場合

は、後でリソースに対するロックを再試行できます。NOWAIT を指定しないと、要求した表ロックが取得されるまで、トランザクションは処理を続行しません。表ロックに対する待ち時間が長すぎる場合は、そのロック操作を取り消して、後で再試行できます。このロジックは、アプリケーション内に作成できます。

ROW SHARE および ROW EXCLUSIVE モードでロックする場合

```
LOCK TABLE Emp_tab IN ROW SHARE MODE;
LOCK TABLE Emp_tab IN ROW EXCLUSIVE MODE;
```

行共有（ROW SHARE）表ロックおよび行排他（ROW EXCLUSIVE）表ロックは、最も高い並行性を提供します。次のような場合に使用します。

- トランザクション内で表を更新する前に、別のトランザクションが共有表ロック、行共有表ロックまたは排他表ロックを割り込んで取得しないようにする必要がある場合。別のトランザクションが共有表ロック、行共有表ロックまたは排他表ロックを割り込んで取得した場合、他のどのトランザクションも、そのロックしているトランザクションがコミットまたはロールバックされるまで、表を更新できません。
- 後で自トランザクションで表を変更できるようになるまで、表の変更または削除を防止する必要がある場合。

SHARE モードでロックする場合

```
LOCK TABLE Emp_tab IN SHARE MODE;
```

共有表（SHARE）ロックは非常に制限の多いデータ・ロックです。次のような場合に使用します。

- トランザクションが表を問い合せるのみで、そのトランザクションの存続中は一貫した一連の表データを必要とする場合。
- 表に対して SHARE ロックを保持するすべてのトランザクションがコミットまたはロールバックするまで、ロックされている表を更新しようとする他のトランザクションを阻止できる場合。
- 他のトランザクションが、ロックされている表に対して同時に SHARE 表ロックを取得でき、またトランザクション・レベルの読込み一貫性のオプションを使用できる場合。

注意： **同じトランザクション内で、後で表を更新することもしないこともあります。ただし、複数のトランザクションが同じ表に対して共有表ロックを同時に保持している場合は（SELECT...FOR UPDATE 文の結果によって行ロックが保持されている場合でも）、どのトランザクションも表を更新できません。したがって、同じ表に対する同時共有表ロックがよく発生する場合は、更新処理を継続できず、デッドロックがよく発生することになります。このような場合には、かわりに共有行排他ロックまたは排他表ロックを使用してください。**

たとえば、2 つの表 EMP_TAB および BUDGET_TAB には、第 3 の表 DEPT_TAB の一貫した一連のデータが必要であると仮定します。特定の部門番号に関して、2 つの表の情報を更新し、この 2 つのトランザクションの間に新しいメンバーが部門に追加されないように保証するものとします。

この使用例はきわめてまれな場合ですが、次の例で示すように、SHARE MODE で DEPT_TAB 表をロックすることによって対処できます。DEPT_TAB 表の更新はまれなため、ロックしても他の多くのトランザクションの待ち時間が長くなることはありません。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE dept_tab(
   deptno NUMBER(2) NOT NULL,
   dname VARCHAR2(14),
   loc VARCHAR2(13));

CREATE TABLE emp_tab (
   empno NUMBER(4) NOT NULL,
   ename VARCHAR2(10),
   job VARCHAR2(9),
   mgr NUMBER(4),
   hiredate DATE,
   sal NUMBER(7,2),
   comm NUMBER(7,2),
   deptno NUMBER(2));

CREATE TABLE Budget_tab (
   totsal NUMBER(7,2),
   deptno NUMBER(2) NOT NULL);
```

```
LOCK TABLE Dept_tab IN SHARE MODE;
UPDATE Emp_tab
    SET sal = sal * 1.1
    WHERE deptno IN
      (SELECT deptno FROM Dept_tab WHERE loc = 'DALLAS');
```

```
UPDATE Budget_tab
    SET Totsal = Totsal * 1.1
    WHERE Deptno IN
      (SELECT Deptno FROM Dept_tab WHERE Loc = 'DALLAS');

COMMIT; /* This releases the lock */
```

SHARE ROW EXCLUSIVE モードでロックする場合

```
LOCK TABLE Emp_tab IN SHARE ROW EXCLUSIVE MODE;
```

共有行排他（SHARE ROW EXCLUSIVE）表ロックは、次のような場合に使用します。

- トランザクションで、指定された表に対するトランザクション・レベルの読込み一貫性、およびロックされている表の更新が必要な場合。
- 他のトランザクションによる明示的行ロックの取得（SELECT... FOR UPDATE による）を意識する必要がない場合。ロック中のトランザクション内の UPDATE および INSERT 文が待機させられ、デッドロックが発生する可能性があります。
- 前述のように動作するトランザクションが 1 つのみ必要な場合。

EXCLUSIVE モードでロックする場合

```
LOCK TABLE Emp_tab IN EXCLUSIVE MODE;
```

排他（EXCLUSIVE）表ロックは、次のような場合に使用します。

- トランザクションが、ロックされている表にすぐに更新アクセスをする必要がある場合。トランザクションが排他表ロックを保持していると、他のトランザクションはロックされた表の中の特定の行をロックできません。
- トランザクションがコミットまたはロールバックされるまで、ロックされた表に対してトランザクション・レベルの読込み一貫性が保持される場合。
- 低レベルのデータ並行性を意識する必要がなく、排他表ロックを要求するトランザクションを順次待機させて表を順番に更新させる場合。

必要な権限

自スキーマ内の表に対しては、どの種類の表ロックでも自動的に取得できます。他スキーマ内の表に対して表ロックを取得するには、LOCK ANY TABLE システム権限またはその表に対する（SELECT や UPDATE などの）オブジェクト権限が必要です。

Oracle による表ロック制御

Oracle に表ロック制御を任せると、アプリケーションに必要なプログラム・ロジックが少なくて済みます。ただし、表ロックを自分で管理する場合より制御範囲が小さくなります。

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE コマンドまたはALTER SESSION ISOLATION LEVEL SERIALIZABLE コマンドを発行すると、基礎となるロック・プロトコルを変更しなくても、ANSI のシリアル化可能性を維持できます。この手法によって、ANSI のシリアル化可能性を維持しながら表へ同時アクセスできます。表ロックの取得によって、並行性が大幅に減少します。

また、表ロックは ROW_LOCKING 初期化パラメータおよび SERIALIZABLE 初期化パラメータによっても制御されます。デフォルトでは、SERIALIZABLE は FALSE に設定され、ROW_LOCKING は ALWAYS に設定されます。ほとんどの場合、これらのパラメータは変更しないでください。このパラメータは、ANSI/ISO 互換モードでの実行が必要なサイト、または以前のバージョンの Oracle で実行するように作成されたアプリケーションを使用するサイトのために用意されたものです。このようなサイトでのみパラメータの変更を検討してください。デフォルト以外の設定を使用するとパフォーマンスが大幅に低下するためです。

> **参照：** SET TRANSACTION 文および ALTER SESSION 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

インスタンスが停止されているときにのみ、これらのパラメータの設定を変更可能です。複数インスタンスが単一データベースにアクセスする場合は、すべてのインスタンスでこれらのパラメータの設定を同じにする必要があります。

デフォルト以外のロック・オプションの概要

トランザクションのロック方法を変更するには、デフォルト設定以外に、SERIALIZABLE と ROW_LOCKING の 3 通りの組合せを使用できます。表 7-1 に、デフォルト以外の設定、およびトランザクションを特定の設定で実行する理由を示します。

表 7-1 デフォルト以外のロック・オプションの概要

状況	説明	SERIALIZABLE	ROW_LOCKING
1	Oracle バージョン 5 以下と等価（表に対して同時に挿入、更新または削除できません）	使用禁止（デフォルト）	INTENT
2	ANSI 互換	使用可能	ALWAYS
3	表レベル・ロックの ANSI 互換（表に対して同時に挿入、更新または削除できません）	使用可能	INTENT

表 7-2 に、表 7-1 で示した 3 通りの SERIALIZABLE オプションおよび ROW_LOCKING 初期化パラメータの設定について、それぞれロック動作の結果の違いを示します。

表 7-2 デフォルト以外のロックの動作

文	ケース 1		ケース 2		ケース 3	
	行	表	行	表	行	表
SELECT	-	-	-	S	-	S
INSERT	X	SRX	X	RX	X	SRX
UPDATE	X	SRX	X	SRX	X	SRX
DELETE	X	SRX	X	SRX	X	SRX
SELECT...FOR UPDATE	X	RS	X	S	X	S
LOCK TABLE... IN..	-	-	-	-	-	-
ROW SHARE MODE	-	RS	-	RS	-	RS
ROW EXCLUSIVE MODE	-	RX	-	RX	-	RX
SHARE MODE	-	S	-	S	-	S
SHARE ROW EXCLUSIVE MODE	-	SRX	-	SRX	-	SRX
EXCLUSIVE MODE	-	X	-	X	-	X
DDL 文	-	X	-	X	-	X

行ロックの明示的な取得

FOR UPDATE 句を含む SELECT 文を使用すると、デフォルト・ロックをオーバーライドできます。この文は、選択されている行がその後の文で更新されることを想定し、（UPDATE 文のように）選択された行に対する明示的な行ロックを取得します。

SELECT... FOR UPDATE 文を使用すると、実際にその行を変更しないで行をロックできます。たとえば、第 15 章「トリガーの使用」では、いくつかのトリガーで参照整合性を実装する方法を示しています。EMP_DEPT_CHECK トリガー（12-41 ページの「子表に対する外部キー・トリガー」を参照）では、参照される親キー値を含む行が、トランザクションの存続中は同じ値のままであることを保証するためにロックされます。親キーが更新または削除された場合、参照整合性違反になります。

SELECT... FOR UPDATE 文は、ユーザーが 1 行以上の特定行のフィールドを変更できる（時間がかかることがあります）ような対話型のプログラムでよく使用されます。どの時点でも、行を更新しているのは対話型プログラムの 1 ユーザーのみになるように、行ロックが取得されます。

カーソル定義に SELECT... FOR UPDATE 文が使用される場合は、カーソルがオープンされるとき（最初のフェッチの前）に結果セット内の行がロックされます。行は、カーソルから

フェッチされるときに、個別にロックされるわけではありません。カーソルをオープンしたトランザクションがコミットまたはロールバックされたときにのみ、ロックが解除されます。カーソルがクローズされるときには、ロックは解除されません。

SELECT... FOR UPDATE 文の結果セット内の各行は、個別にロックされます。SELECT... FOR UPDATE 文は、競合する行ロックを他のトランザクションが解除するまで待機します。したがって、SELECT...FOR UPDATE 文が表の行を多数ロックし、表に対して非常に多くの更新アクティビティが発生する場合は、EXCLUSIVE 表ロックを取得する方がパフォーマンスが改善する場合があります。

SELECT... FOR UPDATE を使用して行ロックを取得する場合は、NOWAIT オプションを指定してロック取得時の待機を避けることができます。すぐにロックが取得できない場合は、この時点ではロックできないことを示すエラーが戻されます。その行のロックは、後で再試行できます。

デフォルトでは、要求された行ロックが取得されるまでトランザクションは待機します。行ロックに対する待ち時間が長すぎる場合は、そのロック操作を取り消して後で再試行するロジックを、アプリケーションに作成できます。

ユーザー・ロック

DBMS_LOCK パッケージをコールすることで、アプリケーションで Oracle ロック・マネージメント・サービスを使用できます。特定のモードのロックを要求し、同一のインスタンスまたは別のインスタンスの別のプロシージャで認識できる一意の名前を付け、ロック・モードを変更し、解除することができます。確保されるユーザー・ロックは Oracle ロックと同一であるため、デッドロックの検出などの Oracle ロックのすべての機能を持っています。分散トランザクションで使用されるユーザー・ロックは、COMMIT と同時に解除されるようになっていることを確認してください。解除されないと、検出されないデッドロックが発生する可能性があります。

> **参照：** DBMS_LOCK パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

ユーザー・ロックを使用する場合

次のような場合、ユーザー・ロックを使用します。

- 端末などの装置に対して排他的アクセスを可能にする場合
- アプリケーション・レベルの読込みロックを適用する場合
- ロックの解除を検出し、アプリケーションの終了後にクリーンアップする場合
- アプリケーションを同期化して、順次処理を実行する場合

ユーザー・ロックの例

次の Pro*COBOL プリコンパイラの例は、複数のユーザーが 1 つの装置にアクセスする必要がある場合に、競合が発生しないように保証するためのロックの使用方法を示しています。

```
*****************************************************************
* Print Check                                                   *
* Any cashier may issue a refund to a customer returning goods. *
* Refunds under $50 are given in cash, above that by check.     *
* This code prints the check. The one printer is opened by all  *
* the cashiers to avoid the overhead of opening and closing it  *
* for every check. This means that lines of output from multiple*
* cashiers could become interleaved if we don't ensure exclusive*
* access to the printer. The DBMS_LOCK package is used to       *
* ensure exclusive access.                                      *
*****************************************************************
CHECK-PRINT
*
*    Get the lock "handle" for the printer lock.
   MOVE "CHECKPRINT" TO LOCKNAME-ARR.
   MOVE 10 TO LOCKNAME-LEN.
   EXEC SQL EXECUTE
      BEGIN DBMS_LOCK.ALLOCATE_UNIQUE ( :LOCKNAME, :LOCKHANDLE );
      END; END-EXEC.
*
*   Lock the printer in exclusive mode (default mode).
   EXEC SQL EXECUTE
      BEGIN DBMS_LOCK.REQUEST ( :LOCKHANDLE );
      END; END-EXEC.
*   We now have exclusive use of the printer, print the check.

  ...

*
*   Unlock the printer so other people can use it
*
   EXEC SQL EXECUTE
      BEGIN DBMS_LOCK.RELEASE ( :LOCKHANDLE );

      END; END-EXEC.
```

ロックの表示および監視

Oracle では、インスタンスで処理中のトランザクションに対するロック情報を表示するために、次の 2 つの機能が用意されています。

ロック情報の表示方法	説明
Enterprise Manager モニター (ロック・モニターおよびラッチ・モニター)	Enterprise Manager のモニター機能は、インスタンスのロック情報を表示するために 2 つのモニターを用意しています。Enterprise Manager モニターの詳細は、『Oracle Enterprise Manager 管理者ガイド』を参照してください。
UTLLOCKT.SQL	UTLLOCKT.SQL スクリプトは、ツリー構造の簡単なロック待機グラフを表示します。スクリプトの実行に非定型 SQL ツール（SQL*Plus など）を使用し、システム内のロック待機中のセッション、およびそれに対応するブロッキング・ロックを出力します。このスクリプト・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（UTLLOCKT.SQL を使用する前に、CATBLOCK.SQL スクリプトを実行する必要があります）。

シリアル化可能トランザクションを使用した並行性の制御

Oracle サーバーは、デフォルトでは、同時に実行されるトランザクションの同じ表および同じデータ・ブロック内で行の修正、追加または削除を許可しています。あるトランザクションによって行われた変更は、その変更を行ったトランザクションがコミットされるまで、別の同時トランザクションでは参照できません。

トランザクション A が、（DML 文または SELECT... FOR UPDATE 文を使用して）別のトランザクション B によってロックされている行を更新または削除しようとすると、トランザクション A の DML コマンドは、トランザクション B がコミットまたはロールバックされるまでブロックされます。トランザクション B がコミットされると、トランザクション A は、トランザクション B がそのデータベースに対して行った変更を参照できます。

この並行性モデルはほとんどのアプリケーションに適しています。これは、より高度な並行性が提供されて、パフォーマンスが改善されるためです。ただし、まれにシリアル化可能なトランザクションが必要な場合もあります。シリアル化可能トランザクションは、同時にではなく、一度に 1 トランザクションずつ（シリアルで）実行しているように見える方法で実行する必要があります。シリアル・モードで実行中の同時トランザクションでは、それらのトランザクションが 1 つずつ順次実行された場合に可能となるデータベースの変更のみが可能です。

ANSI/ISO SQL 規格 SQL92 は、考えられる 3 種類のトランザクションの相互作用、およびそれらの相互作用に対する保護を強化する 4 レベルの分離を定義しています。表 7-3 に、これらの相互作用および分離レベルの概要を示します。

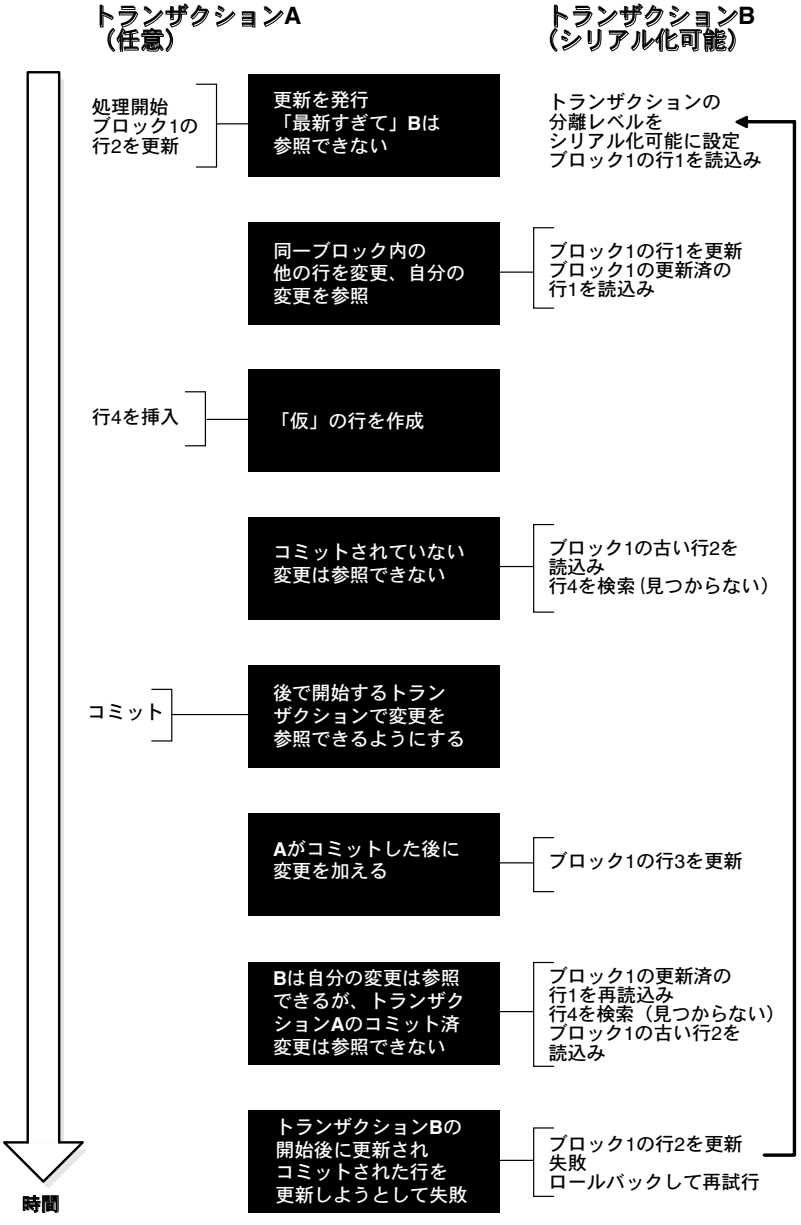
表 7-3 ANSI 分離レベルの概要

分離レベル	内容を保証しない読込み（1）	反復不能な読込み（2）	仮読込み（3）
非コミット読込み	可能	可能	可能
コミット読込み	不可能	可能	可能
リピータブル・リード	不可能	不可能	可能
SERIALIZABLE	不可能	不可能	不可能
注意：	（1）トランザクションは、別のトランザクションで変更されたコミットされていないデータを読み込むことができます。 （2）トランザクションは、別のトランザクションでコミットされたデータを再度読み込み、その新しいデータを参照します。 （3）トランザクションは問合せを再実行し、コミットされた別のトランザクションによって挿入された新しい行を検出できます。		

これらの分離レベルに関する Oracle の動作について、次に概要を示します。

分離レベル	説明
非コミット読込み	Oracle では「内容を保証しない読込み」は許可されません。他のデータベース製品の中には、スループットの改善のために、この方法を使用する場合もありますが、スループットの高い Oracle では不要です。
コミット読込み	Oracle は、コミット読込み分離標準に準拠しています。これは、すべての Oracle アプリケーションのデフォルト・モードです。Oracle の問合せは、問合せの始め（スナップショット時）にコミット済のデータのみを参照するため、Oracle では、コミット読込み分離について実際に ANSI/ISO SQL92 規格で要求される以上の整合性を提供します。
リピータブル・リード	通常、Oracle は、シリアラル化可能トランザクションによって提供されるもの以外は、この分離レベルをサポートしていません。
SERIALIZABLE	この分離レベルは、SET TRANSACTION コマンドまたは ALTER SESSION コマンドを使用して設定できます。

図 7-2 2 つのトランザクションの時系列的働き



シリアル化可能トランザクションの相互作用

7-24 ページの図 7-3 は、シリアル化可能トランザクション（トランザクション B）と、別のトランザクション（トランザクション A、シリアル化可能かコミット読込みのいずれか）との相互作用を示しています。

シリアル化可能トランザクションが「ORA-08177: このトランザクションのアクセスをシリアル化できません」のエラーで失敗したとき、アプリケーションは次のいずれかで対処できます。

- そのポイントまで実行された作業をコミットします。
- その他の様々な文を、トランザクション内の直前のセーブポイントまでロールバックした後に実行します。
- トランザクション全体をロールバックし再試行します。

Oracle では、同時トランザクションによるアクセスを管理するために、各データ・ブロックに制御情報を格納します。シリアル化可能分離レベルを使用するには、CREATE TABLE コマンドまたは ALTER TABLE コマンドの INITRANS 句を使用して、この制御情報の格納を取り消す必要があります。シリアル化可能モードを使用するには、INITRANS を 3 以上に設定する必要があります。

トランザクションの分離レベルの設定

トランザクションの分離レベルは、SET TRANSACTION コマンドの ISOLATION LEVEL 句を使用して変更できます。SET TRANSACTION コマンドは、トランザクションで最初に発行されるコマンドである必要があります。

トランザクション分離レベルをセッション全体に設定するには、ALTER SESSION コマンドを使用します。

> **参照：** SET TRANSACTION コマンドおよび ALTER SESSION コマンドの完全な構文については、『Oracle9*i* データベース・リファレンス』を参照してください。

INITRANS パラメータ

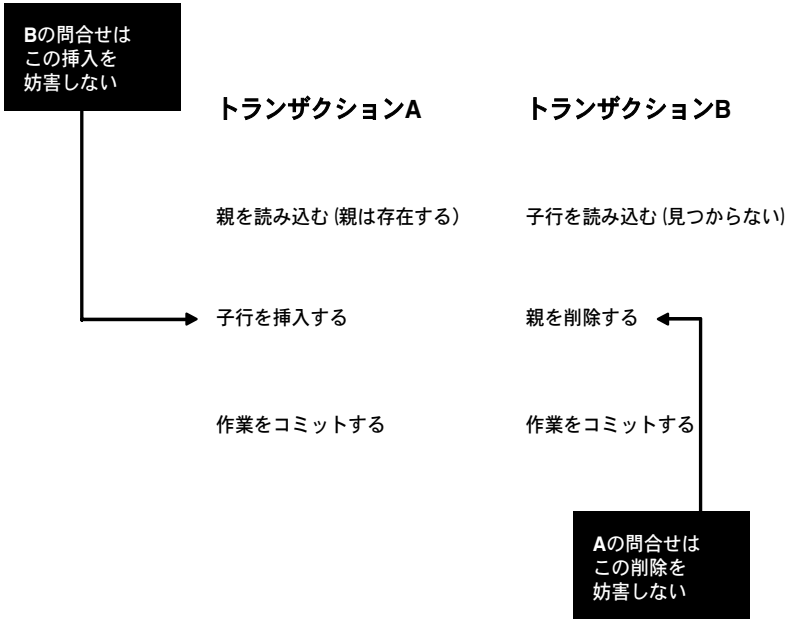
Oracle では、同時トランザクションによるアクセスを管理するために、各データ・ブロックに制御情報を格納します。したがって、トランザクション分離レベルをシリアル化可能に設定する場合は、ALTER TABLE コマンドを使用して、INITRANS を 3 以上に設定する必要があります。このパラメータを使用すると、Oracle は、ブロックにアクセスした最新トランザクションの履歴を記録するために十分な記憶域を、それぞれのブロック内に割り当てます。同じブロックを更新するトランザクションの数が多い表には、さらに大きい値を使用する必要があります。

参照整合性およびシリアル化可能トランザクション

Oracle は、シリアル化可能トランザクション内であっても読込みロックを使用しないため、あるトランザクションによって読み込まれたデータは、別のトランザクションでオーバーライドできます。アプリケーション・レベルでデータベースの整合性チェックを実行するトランザクションでは、読み込んだデータはトランザクション実行中には変更されないと考えないでください（そのような変更がトランザクションからはわからない場合も）。シリアル化可能トランザクションを使用した場合でも、アプリケーション・レベルの整合性チェックのコードを十分注意して作成しないと、データベースの不整合が発生する可能性があります。ただし、この項に示されている例は、コミット読込みトランザクションおよびシリアル化可能トランザクションの両方に該当するため注意してください。

7-24 ページの図 7-3 に、2 つの表の間の参照整合性の親子関係を保持するために、アプリケーション・レベルでチェックを実行する異なる 2 つのトランザクションを示します。一方のトランザクションは、親表に特定の主キー値を持つ行が存在するかどうかをチェックした後、対応する子である行を挿入します。もう一方のトランザクションは、対応するディテール行が存在しないことをチェックした後、親である行を削除します。この場合、両方のトランザクションが読み込んだデータは、そのトランザクションが完了する前には変更されないものと想定しています（確認はしません）。

図 7-3 参照整合性チェック



トランザクション A が実行した読込みのために、トランザクション B が親である行を削除できなくなることはありません。同様に、トランザクション B が子である行の問合せを行っても、トランザクション A が子である行を挿入できなくなることもありません。したがって、この使用例では、対応する親である行を持たない子である行がデータベースに残ります。どちらのトランザクションも、整合性チェックのため読み込んだデータに対する変更を他方が防げないため、両方のトランザクションがシリアル化可能トランザクションであっても、前述の不整合が発生する可能性があります。

この例に示されているとおり、一方のトランザクションで読み込まれたデータがもう一方のトランザクションで同時に書き込まれないように、処置を行う必要がある場合があります。これには、SQL92 シリアル化可能モードで定義されているトランザクション分離レベルよりもかなり高いレベルが必要です。

SELECT FOR UPDATE の使用

Oracle では、前述の矛盾を簡単に防ぐことができます。

- トランザクション A は、SELECT FOR UPDATE を使用して、親である行を問い合せてロックし、トランザクション B がその行を削除しないように設定できます。

- 逆に、トランザクション B は、トランザクション A が親である行へのアクセスを取得しないように設定できます。トランザクション B は、まず親である行を削除し、その後の問合せで、子表の中に対応する行が存在することが判明した場合、ロールバックします。

Oracle では、前述のトランザクション A の場合のような独立した問合せのかわりに、データベース・トリガーを使用して参照整合性を施行することもできます。たとえば、子表への INSERT によって、BEFORE INSERT 行レベル・トリガーを起動して、対応する親である行の有無をチェックできます。このトリガーは、SELECT FOR UPDATE を使用して親表を問い合せ、子である行を挿入するトランザクションの処理中に、親である行が（存在する場合）データベース内に残るようにします。対応する親である行が存在しない場合、トリガーは子である行の挿入を拒否します。

データベース・トリガーによって発行された SQL 文は、そのトリガーを起動した SQL 文のコンテキスト内で実行されます。1 つのトリガー内で実行されるすべての SQL 文は、トリガー起動文から参照する状態と同じデータベース状態を参照します。そのため、コミット読込みトランザクションでは、トリガー内の SQL 文は、トリガー文の実行開始時点のデータベースを参照し、シリアル化可能モードで実行するトランザクションでは、SQL 文は、そのトランザクションの開始時点のデータベースを参照します。いずれの場合も、トリガーで SELECT FOR UPDATE を使用すると、参照整合性が正しく施行されます。

コミット読込み分離およびシリアル化可能分離

Oracle の場合、アプリケーション開発者は異なる特性を持つ 2 つのトランザクション分離レベルのうちの 1 つを選択できます。コミット読込み分離レベルおよびシリアル化可能分離レベルは、どちらも高度な一貫性および並行性を提供します。これら 2 つの分離レベルは競合を軽減し、実社会でのアプリケーション配置用に設計されています。この項の後半では、2 つの分離レベルを比較し、その選択の際に有効な情報を示します。

トランザクション集合の整合性

Oracle のコミット読込み分離レベルおよびシリアル化可能分離レベルについて説明するには、次のことを考慮すると効果的です。

- データベース表（または任意の一連のデータ）の集合
- それらの表内の行の特定の読込み順序
- 特定の時刻（任意）にコミットされるトランザクションの集合

どの読込みにおいても、同じコミット済トランザクション集合によって書き込まれたデータが戻される操作（問合せまたはトランザクション）を、**「トランザクション集合整合である」**といいます。トランザクション集合整合でない操作では、ある集合のトランザクションの変更が反映される読込みと、他のトランザクションによって行われた変更が反映される読込みが存在します。そのような操作では、そのデータベースは、コミットされたトランザクション集合が反映されていない状態のデータベースのように見えます。

Oracle のコミット読込みモードでは、問合せによって読み込まれたすべての行が、その問合せが始まる前にコミットされているため、文単位でのトランザクション集合整合です。

Oracle のシリアル化可能モードでは、シリアル化可能トランザクション内のすべての文が、トランザクション開始時点のデータベースのイメージに対して実行されるため、トランザクション単位でのトランザクション集合整合です。

他のデータベース・システムでは、コミット読込みモードで問合せを 1 回実行すると、トランザクション集合の整合性は失われます。この問合せでは、別のトランザクションによって行われた変更のサブセット以外見えないため、トランザクション集合は整合していません。たとえば、ディテール表とマスター表を結合すると、別のトランザクションによって挿入されたマスター・レコードを見ることはできますが、そのトランザクションによって挿入された対応するディテールは見えません（その逆も同じです）。Oracle のコミット読込みモードではこのような問題は回避されるため、読込みロック・システムより高い整合性が得られます。

読込みロック・システムでは、同時更新ができないようにするかわりに、SQL92 REPEATABLE READ 分離によって、トランザクション・レベルではなく、文レベルでトランザクション集合の整合性が提供されます。仮（phantom）の保護がないということは、同一のトランザクションによって発行された 2 つの問合せが、他のトランザクションの別の集合によってコミットされたデータを参照できることを意味します。これらのシステムでは、スループットに制限がありデッドロックされやすいシリアル化可能モードの場合のみ、トランザクション・レベルでのトランザクション集合の整合性が提供されます。

コミット読込みトランザクションとシリアル化可能トランザクションの比較

表 7-4 に、コミット読込みトランザクションとシリアル化可能トランザクションとの間の主な類似点および相違点を説明します。

表 7-4 コミット読込みトランザクションおよびシリアル化可能トランザクション

操作	コミット読込み	シリアル化可能
内容を保証しない書込み	不可能	不可能
内容を保証しない読込み	不可能	不可能
反復不能な読込み	可能	不可能
仮読込み	可能	不可能
ANSI/ISO SQL 92 への準拠	あり	あり
スナップショット読込み時間	文	トランザクション
トランザクション集合の整合性	文レベル	トランザクション・レベル
行レベル・ロック	あり	あり

表 7-4 コミット読込みトランザクションおよびシリアル化可能トランザクション（続き）

操作	コミット読込み	シリアル化可能
読込みが書込みをブロック	なし	なし
書込みが読込みをブロック	なし	なし
異なる行の書込みが書込みをブロック	なし	なし
同じ行の書込みが書込みをブロック	あり	あり
阻止しているトランザクションの待機	あり	あり
「このトランザクションのアクセスをシリアル化できません」というエラーが発生する可能性	なし	あり
阻止しているトランザクションが異常終了後のエラー	なし	なし
阻止しているトランザクションがコミット後のエラー	なし	あり

トランザクションの分離レベルの選択

それぞれのアプリケーションおよび作業負荷に適した分離レベルを選択する必要があります。また、異なるトランザクションにはそれぞれ個別の分離レベルを選択できます。分離レベルは、パフォーマンスと整合性のニーズ、およびアプリケーション・コーディング要件を考慮して選択します。

多数のユーザーが、トランザクションを同時に次々に送る環境の場合、予期されるトランザクション到着頻度および応答時間要件に対するトランザクション・パフォーマンスを評価して、十分なパフォーマンスを確保しながら必要な整合性を提供する分離レベルを選択する必要があります。多くの場合、高パフォーマンス環境では、整合性と並行性（トランザクションのスループット）を考慮して妥協点を見つける必要があります。

どちらの Oracle 分離モードも、行レベル・ロックと Oracle の複数バージョン並行処理制御システムとを組み合せることによって、高レベルの整合性および並行性（およびパフォーマンス）を提供します。Oracle では読込みと書込みの相互干渉がないため、問合せで整合性のあるデータが参照できる一方、コミット読込み分離およびシリアル化可能分離により、コミットされていない（内容が保証されない）データの読込みを防止し高レベルの並行性を提供することで、高いパフォーマンスを実現しています。

コミット読込み分離レベルでは、一部のトランザクションについては（仮読込みおよび反復不能な読込みのため）一貫性のない結果が生成される可能性は多少高くなりますが、かなり高い並行性を提供できます。シリアル化可能分離レベルの場合は、仮読込みおよび反復不能な読込みから保護されているため、より高い整合性が提供され、読込み / 書込みトランザクションが問合せを 2 回以上実行する場合にはこの分離レベルは重要です。ただし、シリアル化可能モードでは、アプリケーションが「このトランザクションのアクセスをシリアル化できません」というエラーの有無を確認する必要があり、多数の同時トランザクションが更新

のために同じデータにアクセスする環境では、スループットはかなり低下する可能性があります。データベースの整合性を確認するアプリケーション・ロジックでは、いずれのモードでも読込みが書込みをブロックしないように設定する必要があります。

トランザクションのためのアプリケーションのヒント

トランザクションがシリアル化可能モードで実行する場合、シリアル化可能トランザクションの開始以降に、別のトランザクションにより変更されたデータを変更しようとすると、次のようなエラーが発生します。

```
ORA-08177: このトランザクションのアクセスをシリアル化できません。
```

このエラーが発生した場合は、現行のトランザクションをロールバックし、操作を再実行します。トランザクションが新しいトランザクション・スナップショットを取得するため、操作が成功する可能性が高くなります。

トランザクションのロールバックおよび再実行によるパフォーマンスのオーバーヘッドを最小化するには、他の同時トランザクションと競合する可能性のある DML 文は、できるだけトランザクションの始めの方に置くようにしてください。

自律型トランザクション

この項では、自律型トランザクション（AT）の概要およびこのトランザクションの機能を簡単に説明します。

参照： 自律型トランザクションの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』および第 15 章「トリガーの使用」を参照してください。

場合によっては、プライマリ・トランザクションの最終結果とは関係なく、表に対してある種の変更をコミットまたはロールバックする必要がある場合があります。たとえば、株式売買トランザクションでは、全体的な株式売買行為が実際に遂行されるかどうかに関係なく、顧客情報のコミットが必要な場合があります。またはそのトランザクションを実行中にトランザクション全体がロールバックされた場合でも、エラー・メッセージをデバッグ表にログする必要がある場合もあります。自律型トランザクションを使用すると、これらのタスクを実行できます。

自律型トランザクションは、別のトランザクション（**メイン・トランザクション**（MT））によって開始される独立したトランザクションです。自律型トランザクションを使用すると、メイン・トランザクションを停止して、SQL 操作を実行し、その SQL 操作をコミットまたはロールバックした後でメイン・トランザクションを再開できます。

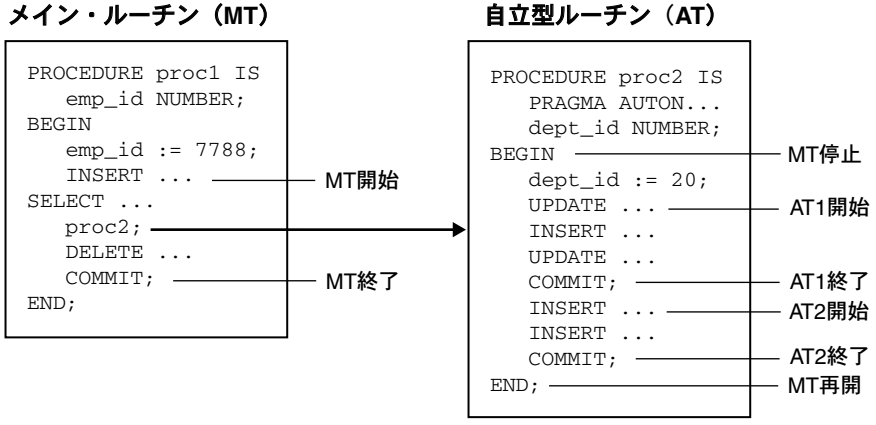
自律型トランザクションは、**自律型スコープ**内で実行されます。自律型スコープとは、プラグマ（コンパイラ・ディレクティブ）AUTONOMOUS_TRANSACTION でマークされたルーチ

ンです。このプラグマは、PL/SQL コンパイラに対して、ルーチンを自律型（非依存）としてマークするように指示します。ここで意味するルーチンには、次のものが含まれます。

- 最上位（ネストされていない）の無名 PL/SQL ブロック
- ローカルなスタンドアロンのパッケージ・ファンクションおよびパッケージ・プロシージャ
- SQL オブジェクト型のメソッド
- PL/SQL トリガー

図 7-4 に、メイン・ルーチン（MT）と自律型ルーチン（AT）との間の制御フローを示します。図からわかるように、自律型ルーチンでは、制御がメイン・ルーチンに戻る前に複数のトランザクション（AT1 および AT2）をコミットできます。

図 7-4 トランザクション制御フロー



自律型ルーチンの実行可能セクションに入ると、メイン・トランザクションが停止します。自律型ルーチンを終了すると、メイン・トランザクションが再開します。COMMIT および ROLLBACK によって、アクティブな自律型トランザクションは終了しますが、自律型ルーチンは終了しません。図 7-4 に示すとおり、1 つのトランザクションが終了すると、次の SQL 文が別のトランザクションを開始します。

自律型トランザクションの特長をさらにいくつか示します。

- 自律型トランザクションが加える変更は、メイン・トランザクションの状態または最終的な処理には依存しません。次に例を示します。

  - 自律型トランザクションは、メイン・トランザクションによって加えられた変更を認識しません。
  - 自律型トランザクションがコミットまたはロールバックしても、メイン・トランザクションの結果には影響しません。
- 自律型トランザクションが加える変更は、そのトランザクションがコミットした直後に他のトランザクションで参照できます。これは、メイン・トランザクションがコミットするまで待機しなくても、ユーザーは更新された情報にアクセスできるということを意味します。
- 自律型トランザクションは他の自律型トランザクションを開始できます。

図 7-5 に、自律型トランザクションが従う実行順序の例を示します。

図 7-5 自律型トランザクションの実行順序の例

メイン・トランザクション・スコープ（MTスコープ）がメイン・トランザクションMTxを開始する。MTxは最初の自立型トランザクション・スコープ（ATスコープ1）を起動する。MTxが停止する。ATスコープ1はトランザクションTx1.1を開始する。

ATス コープ1はTx1.1をコミットまたはロールバックして、終了する。MTxが再開する。

MTxがATスコープ2を起動する。MTが停止し、制御をATスコープ2に渡す。ATスコープ2は最初に問合せを実行する。

ATスコープ2が、更新などを実行するTx2.1を開始する。ATスコープ2がTx2.1をコミットまたはロールバックする。

その後で、ATスコープ2が2番目のトランザクション, Tx2.2を開始し、これをコミットまたはロールバックする。

ATスコープ2は問合せをいくつか実行して、終了し、MTxに制御を戻す。

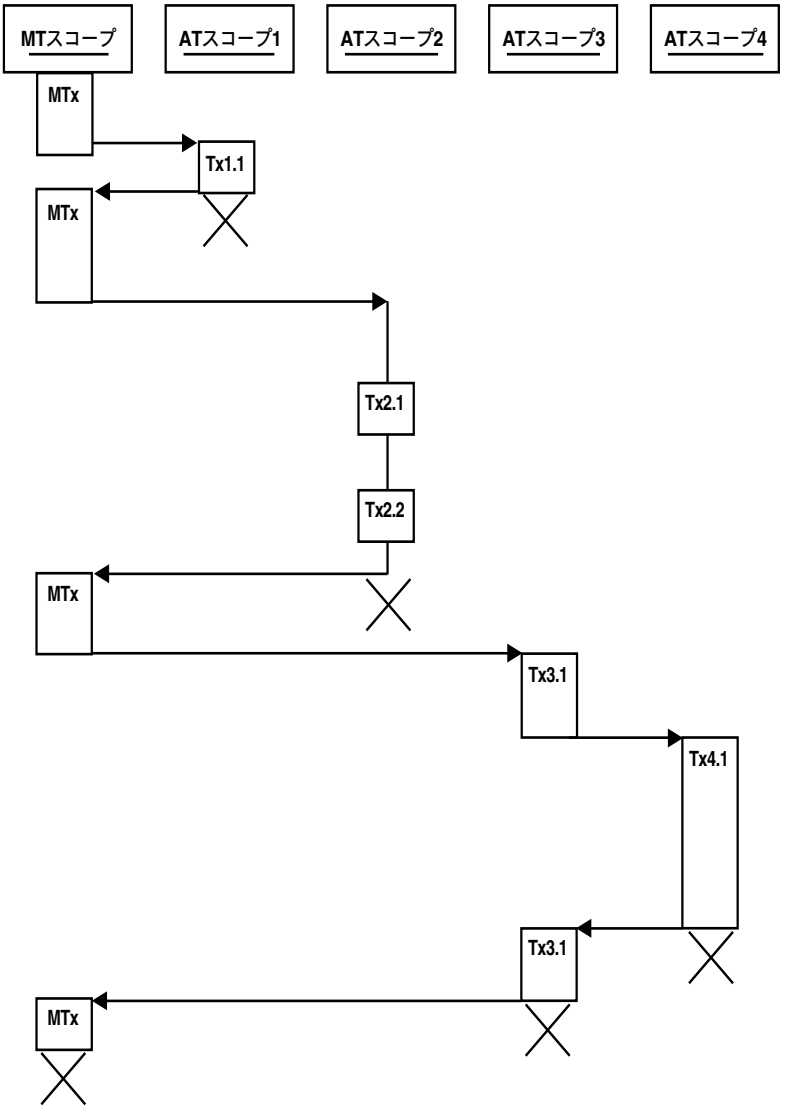
MTxはATスコープ3を起動する。MTx は停止し、ATスコープ3が開始する。

ATスコープ3はTx3.1を開始する。Tx3.1はATスコープ4を起動する。Tx3.1が停止し、ATスコープが開始する。

ATスコープ4はTx4.1を開始し、これをコミットまたはロールバックして、終了する。ATスコープ3が再開する。

ATスコープ3はTx3.1をコミットまたはロールバックしてから終了する。MTxが再開する。

最後にMTスコープがMTxをコミットまたはロールバックして、終了する。



自律型トランザクションの例

この項の 2 つの例では、自律型トランザクションの使用方法をいくつか説明します。

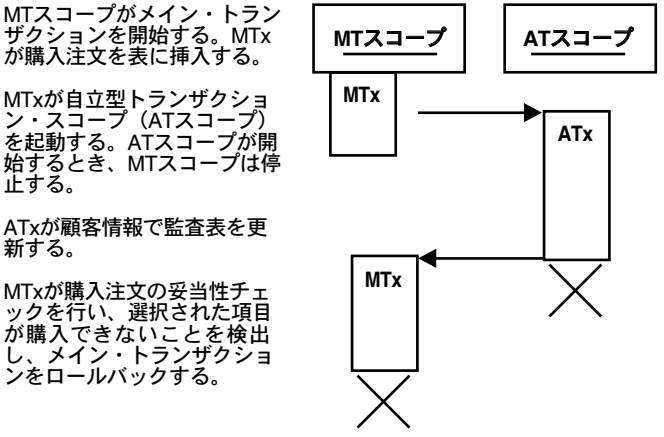
例に示すとおり、自律型トランザクションおよびメイン・トランザクションを使用する場合は、4 種類の結果が考えられます。この結果を次の表に示します。表からわかるように、自律型トランザクションの結果とメイン・トランザクションの結果との間に依存性はありません。

自律型トランザクション	メイン・トランザクション
コミット	コミット
コミット	ロールバック
ロールバック	コミット
ロールバック	ロールバック

購入注文の入力

この例では、顧客が購入注文を入力します。購買契約が成立しなくても、その顧客の情報（名前、住所、電話番号など）は顧客情報表にコミットされます。

図 7-6 例 : 購買指示



例 : 預金払戻しの実行

次の銀行アプリケーションでは、顧客は自分の口座から払戻しを実行しようとします。この処理で、メイン・トランザクションは 2 つの自律型トランザクション・スコープ（AT スコープ 1 および AT スコープ 2）のいずれかをコールします。

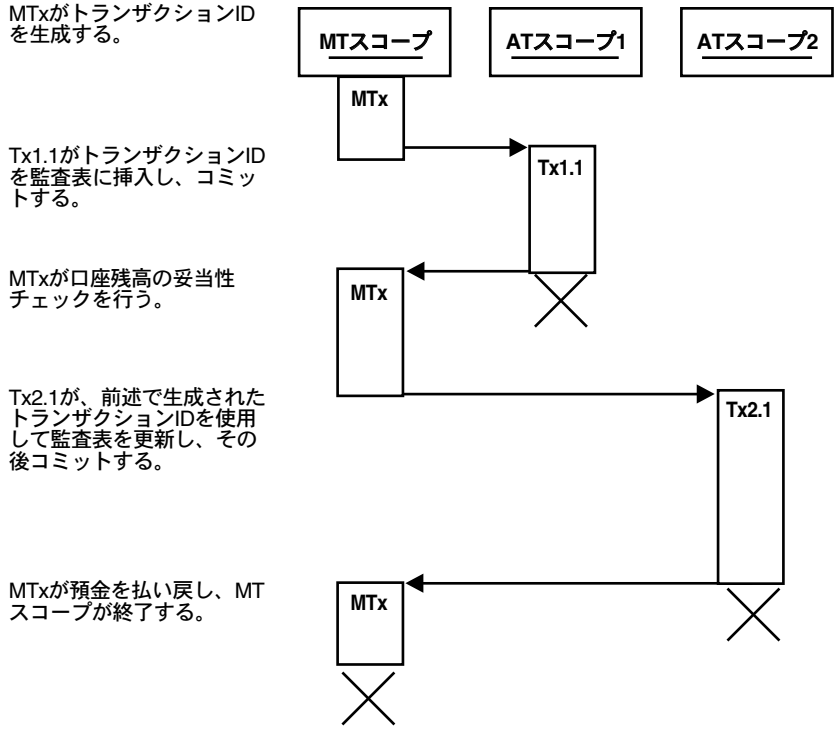
次の図には、このトランザクションで考えられる使用例を 3 つ示します。

- 使用例 1: 払戻しに十分な預金残高があり、銀行が払戻しに応じます。
- 使用例 2: 払戻しに十分な預金残高はありませんが、この顧客には貸越し保護があります。したがって、銀行は払戻しに応じます。
- 使用例 3: 払戻しに十分な預金残高はなく、この顧客には貸越し保護もありません。したがって、銀行は払戻しを差し止めます。

使用例 1:

払戻しに十分な預金残高があり、銀行が払戻しに応じます。

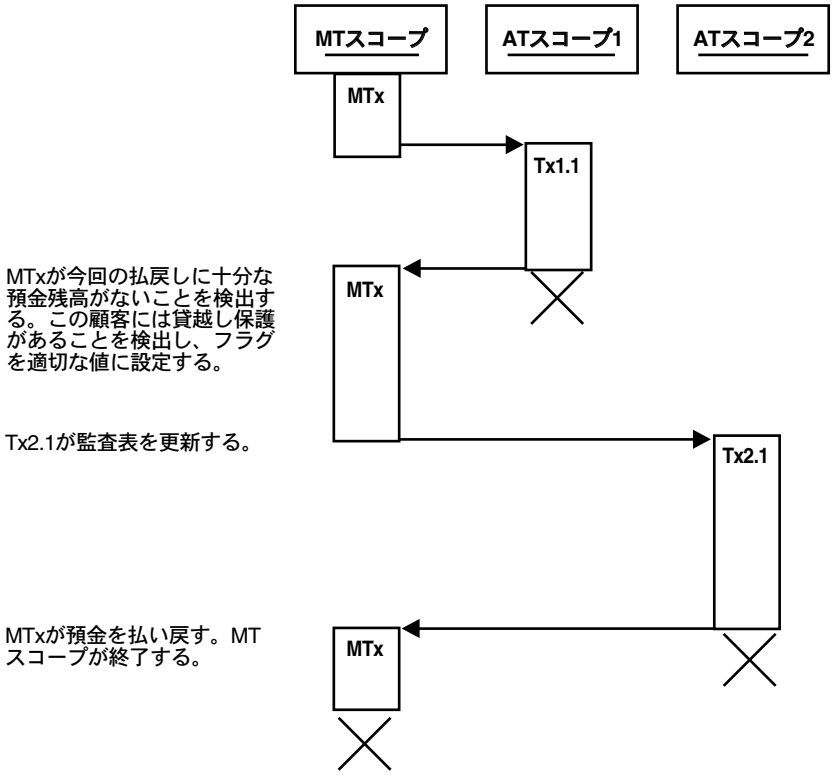
図 7-7 例 : 預金払戻し－十分な預金残高



使用例 2:

払戻しに十分な預金残高はありませんが、この顧客には貸越し保護があります。したがって、銀行は払戻しに応じます。

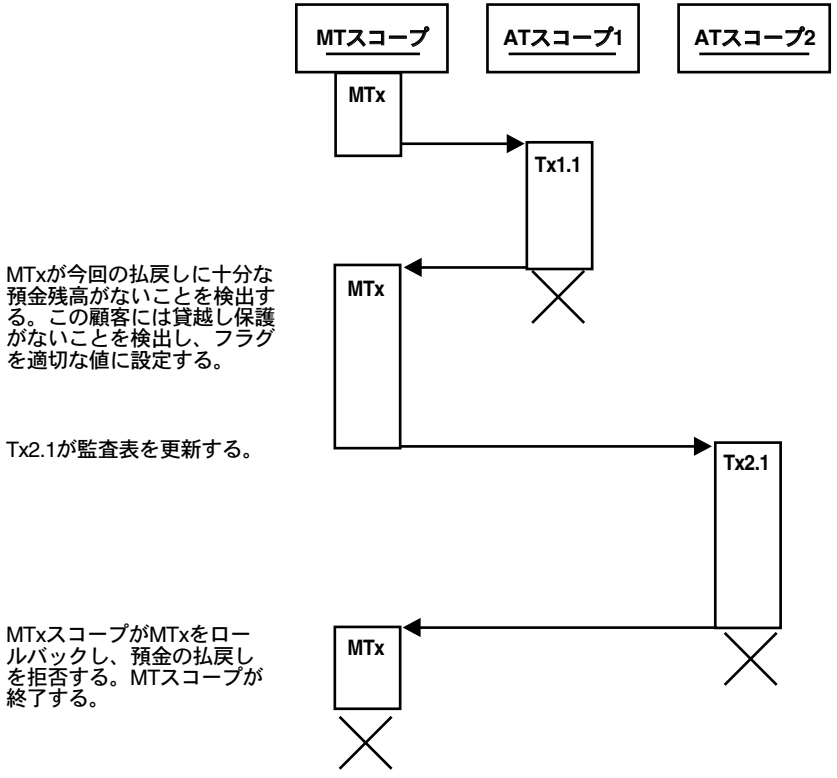
図 7-8 例 : 預金払戻し－不十分な預金残高で貸越し保護あり



使用例 3:

払戻しに十分な預金残高はなく、この顧客には貸越し保護もありません。したがって、銀行は払戻しを差し止めます。

図 7-9 例 : 預金払戻し－不十分な預金残高で貸越し保護なし



自律型トランザクションの定義

> **注意：** この項は、自律型トランザクションに対する一般的な理解を深める目的で提供されています。自律型トランザクションの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

自律型トランザクションを定義するには、プラグマ（コンパイラ・ディレクティブ）AUTONOMOUS_TRANSACTION を使用します。このプラグマは、PL/SQL コンパイラに対して、プロシージャ、ファンクションまたは PL/SQL ブロックを自律型（非依存）としてマークするように指示します。

このプラグマは、プロシージャ、ファンクションまたは PL/SQL ブロックの宣言セクション内のどこにでも作成できます。ただし、コードを読みやすくするために、プラグマはセクションの一番上に作成するようにします。構文は次のとおりです。

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

次の例では、パッケージ・ファンクションを自律型としてマークします。

```
CREATE OR REPLACE PACKAGE Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    -- add additional functions and packages
END Banking;

CREATE OR REPLACE PACKAGE BODY Banking AS
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        My_bal REAL;
    BEGIN
       --add appropriate code
    END;
    -- add additional functions and packages...
END Banking;
```

パッケージ内のすべてのサブプログラム（またはオブジェクト型のすべてのメソッド）を自律型としてマークする目的では、プラグマを使用できません。自律型としてマークできるのは、個々のルーチンのみです。たとえば、次のプラグマは正しくありません。

```
CREATE OR REPLACE PACKAGE Banking AS
    PRAGMA AUTONOMOUS_TRANSACTION; -- illegal
    FUNCTION Balance (Acct_id INTEGER) RETURN REAL;
    END Banking;
```

記憶域エラー状態後の実行の再開

長時間にわたって実行されるトランザクションが領域不足エラー状態によって中断されたときに、アプリケーションによって問題が発生した文を一時停止し、領域問題が修正された後でその文を再開することが可能です。この機能を、**再開可能記憶域割当て**といいます。この機能によって、時間がかかるロールバックを回避できます。また、操作を小さく分割したり、処理過程を追跡するコードを作成する必要がなくなります。

参照：

- 『Oracle9*i* データベース概要』を参照してください。
- 『Oracle9*i* データベース管理者ガイド』を参照してください。

エラー状態後に再開可能な操作

問合せ、DML 操作および特定の DDL 操作は、領域不足エラーが発生した場合、すべて再開可能です。この機能は、操作が SQL 文によって直接実行されているか、または、ストアド・プロシージャ、無名 PL/SQL ブロック、SQL*Loader、`OCIStmtExecute()` などの OCI コール内で実行されている場合に適用されます。

操作は、次のようなエラーの後で再開可能です。

- ORA-01653 などの領域不足エラー
- ORA-01628 などの領域制限エラー
- ORA-01536 などの領域割当てエラー

エラー状態後の操作再開における制限

前述の方法では処理できない記憶域エラーもあります。ディクショナリ管理された表領域では、ロールバック・セグメントの上限に達するか、または索引または表の作成中にエクステントの最大数に達した場合、操作を再開できません。このような場合、ローカル管理表領域および自動 UNDO 管理を使用することをお薦めします。

一時停止された記憶域割当てを処理するアプリケーションの作成

操作が一時停止された場合、アプリケーションは通常のエラー・コードを受信しません。かわりに、`AFTER SUSPEND` イベントを検出し `DBMS_RESUMABLE` パッケージのファンクションをコールして問題についての情報を取得するトリガーをコーディングして、ロギングまたは通知を実行します。このパッケージを使用すると、次のことが可能です。

- `DBMS_RESUMABLE.SPACE_ERROR_INFO` ファンクションを使用して、エラー・メッセージを解析できます。このファンクションの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。
- `SET_TIMEOUT` プロシージャを使用して新しいタイムアウト値を設定できます。

トリガーの本体内では、通知を実行できます。たとえば、オペレータにメール・メッセージを送り、領域問題について警告できます。

また、DBA が、DBA_RESUMABLE、USER_RESUMABLE および V$_SESSION_WAIT データ・ディクショナリ・ビューを使用して、一時停止された文を定期的に確認することもできます。

領域状態が（通常 DBA によって）修正されると、一時停止された文は自動的に実行を再開します。タイムアウト周期が終了するまでに領域状態が修正されなかった場合は、その操作によって SERVERERROR 例外が発生します。

トリガー自体の中で領域不足エラーが発生する可能性を減らすには、トリガーを自律型トランザクションとして宣言し、トリガーが SYSTEM 表領域内のロールバック・セグメントを使用するようにする必要があります。一時停止された文が保持するロックによってトリガーにデッドロック状態が発生した場合、そのトリガーは強制終了され、アプリケーションは一時停止が発生しなかった場合の本来のエラー状態を受信します。トリガーによって領域不足状態が発生した場合、そのトリガーおよび一時停止された文はロールバックされます。トリガー内の例外ハンドラを介してロールバックを回避し、文が再開されるまで待つこともできます。

> **参照：** DBA_RESUMABLE、USER_RESUMABLE および V$_SESSION_WAIT データ・ディクショナリ・ビューに関する詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

再開可能記憶域割当ての例

次に示すトリガーは、データベース内の適用可能な記憶域エラーを処理します。いくつかのエラーでは、このトリガーは文を強制終了し、エラーに関する警告をメール・メッセージによって DBA に通知します。他の一時的なエラーでは、8 時間以内に記憶域の問題が解決していることを想定して、8 時間後に文を再開するように指定しています。

```
CREATE OR REPLACE TRIGGER suspend_example
  AFTER SUSPEND
  ON DATABASE
  DECLARE
  cur_sid NUMBER;
  cur_inst NUMBER;
  err_type VARCHAR2(64);
  object_owner VARCHAR2(64);
  object_type VARCHAR2(64);
  table_space_name VARCHAR2(64);
  object_name VARCHAR2(64);
  sub_object_name VARCHAR2(64);
  msg_body VARCHAR2(64);
  ret_value boolean;
  error_txt varchar2(64);
  mail_conn utl_smtp.connection;
```

```
  BEGIN
  SELECT DISTINCT(sid) INTO cur_sid FROM v$mystat;
  cur_inst := userenv('instance');
  ret_value := dbms_resumable.space_error_info(err_type, object_owner, object_type,
table_space_name, object_name, sub_object_name);
  IF object_type = 'ROLLBACK SEGMENT' THEN
  INSERT INTO sys.rbs_error ( SELECT sql_text, error_msg, suspend_time FROM
dba_resumable WHERE session_id = cur_sid AND instance_id = cur_inst);
  SELECT error_msg into error_txt FROM dba_resumable WHERE session_id = cur_sid AND
instance_id = cur_inst;
  msg_body := 'Subject: Space error occurred: Space limit reached for rollback
segment  '|| object_name || ' on ' || to_char(SYSDATE, 'Month dd, YYYY, HH:MIam') ||
'. Error message was: ' || error_txt;
  mail_conn := utl_smtp.open_connection('localhost', 25);
  utl_smtp.helo(mail_conn, 'localhost');
  utl_smtp.mail(mail_conn, 'sender@localhost');
  utl_smtp.rcpt(mail_conn, 'recipient@localhost');
  utl_smtp.data(mail_conn, msg_body);
  utl_smtp.quit(mail_conn);
  dbms_resumable.abort(cur_sid);
  ELSE
  dbms_resumable.set_timeout(3600*8);
  END IF;
  COMMIT;
  END;
```

特定時点のデータの問合せ（フラッシュバック問合せ）

デフォルトでは、データベースでの操作には、使用可能な最新のコミット済データが使用されます。過去の特定の時点におけるデータベースに対して問合せを実行する必要がある場合、フラッシュバック問合せ機能を使用します。この機能によって、時間またはシステム変更番号（SCN）を指定して、対応する時間のコミット済データを使用する問合せを実行できます。

次にフラッシュバック問合せの適用例を示します。

- 変更をコミットした後で、失われたデータをリカバリしたり不適切な変更を取り消す場合。たとえば、ユーザーが行を削除または更新し、その後コミットしていても、すぐに間違いを修復できます。
- 現在のデータを過去の特定の時点のデータと比較する場合。たとえば、現行のデータのみでなく、前日からの変更を示す日報を作成できます。
- 特定の時点でのトランザクション・データの状態を確認する場合。たとえば、特定の日の預金残高を確認できます。

- 特定の種類の一時データを格納する必要をなくすことによって、アプリケーションの設計を簡略化する場合。
- レポート作成ツールなどのパッケージ・アプリケーションを使用して、過去のデータを処理する場合。

フラッシュバック問合せメカニズムは、自動 UNDO 管理を使用すると最も効果的です。DBA は UNDO データが特定の期間保存されるように要求します。使用可能な記憶域の容量によっては、データベースは要求されたすべての UNDO データを保存できない場合があります。フラッシュバック問合せを使用する場合、自動 UNDO 管理の機能および制限事項を理解しておく必要がある場合があります。

失われたデータのリカバリは他の機能によっても可能ですが、フラッシュバック問合せでは、過去のデータを参照した後、その情報の処理方法を具体的に選択できます。たとえば、追加処理のためのデータの分析、変更の取消しまたは変更したデータの取得などが可能です。

参照：

- フラッシュバック問合せおよび自動 UNDO 管理のバックグラウンド情報の詳細は、『Oracle9*i* データベース概要』を参照してください。
- 自動 UNDO 管理の設定や権限付与などの関連 DBA の責任の詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。
- フラッシュバック問合せとの組合せで使用可能な DBMS_FLASHBACK パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- バックアップとリカバリに使用され、いくつかの V$ データ・ディクショナリ・ビューにあるシステム変更番号の詳細は、『Oracle9*i* データベース・リファレンス』、『Oracle9*i* Recovery Manager ユーザーズ・ガイド』および『Oracle9*i* Recovery Manager リファレンス』を参照してください。

フラッシュバック問合せのためのデータベースの設定

フラッシュバック問合せの前に、DBA に次のことを依頼してください。

- 読込み一貫性を保持するために、ロールバック・セグメントを使用する既存の方法のかわりに、自動 UNDO 管理を使用します。特に、DBA は次のことを実行してください。
  - UNDO_RETENTION 初期化パラメータに、問合せの対象とする過去の期間を示す値を設定します。値は必要に応じて設定します。誤った変更がコミットされた直後にデータをリカバリするのみである場合は、パラメータに小さい値を設定します。数日前に削除されたデータをリカバリする場合は、ある程度大きい値を設定する必要があります。
  - UNDO_MANAGEMENT 初期化パラメータに AUTO を設定します。

    - 要求されたデータを保持するために十分な領域を持つ UNDO 表領域を作成します。データの更新回数が増加するほど、より多くの領域が必要になります。DBA のジョブである領域要件の計算式については、『Oracle9*i* データベース管理者ガイド』を参照してください。
- SQL の AS OF 句を使用してフラッシュバック問合せを実行する必要があるユーザー、ロールまたはアプリケーションに対して、適切な表または FLASHBACK ANY TABLE の FLASHBACK 権限を付与します。
- DBNS_FLASHBACK パッケージを使用してフラッシュバック問合せを実行する必要があるユーザー、ロールまたはアプリケーションに対して、DBMS_FLASHBACK パッケージの EXECUTE 権限を付与します。
- 必要に応じて、RETENTION オプション付きの ALTER TABLE コマンドを使用して、特定の LOB 列のフラッシュバック問合せを可能にします。LOB 列では、UNDO データの保存に記憶域が大量に使用される場合があるため、フラッシュバック問合せとして使用可能な LOB 列を定義する必要があります。

フラッシュバック問合せを使用するアプリケーションの作成

アプリケーションでフラッシュバック問合せ機能を使用するには、次のコーディング方法を使用します。

- SQL 問合せの AS OF 句を使用して、過去の特定の時間を指定します。各表に対してこの句を指定または省略できます。また、異なる表に異なる時間を指定できます。AS OF 句を使用して、問合せと同じセッションで、表の作成、切捨てなどの DDL 操作または挿入、削除などの DML 操作を実行できます。
- DBMS_FLASHBACK パッケージに対するコールを、過去のデータに対する問合せの集合または SQL を変更できない問合せの周辺に置きます。
    - 最初の問合せを実行する前に、DBMS_FLASHBACK.ENABLE_AT_TIME または DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER をコールします。
    - 問合せを実行した後に、DBMS_FLASHBACK.DISABLE をコールします。
    - これらのコールの間では、問合せのみ実行できます。DDL 文または DML 文は実行できません。
- フラッシュバック問合せの結果を、データベースの現在の状態に対して DDL 文または DML 文内で使用する場合は、INSERT 文または CREATE TABLE AS SELECT 文の AS OF 句を使用すると簡単です。

  DBMS_FLASHBACK パッケージを使用する場合は、DBMS_FLASHBACK.DISABLE をコールする前にカーソルをオープンする必要があります。過去のデータの結果をカーソルからフェッチし、その後現行状態のデータベースに対して INSERT 文または UPDATE 文を発行できます。

  DBMS_FLASHBACK パッケージを使用して、現在のデータを過去のデータと比較するには、フラッシュバック機能を使用可能にしたカーソルをオープンし、そのカーソルを使

用禁止にした後で別のカーソルをオープンします。最初のカーソルからはフラッシュバック時点に基づいたデータがフェッチされ、2 番目のカーソルからは現在のデータがフェッチされます。古いデータを一時表に格納し、その後 MINUS や UNION などの集合演算子を使用して、データの違いを示したり、過去のデータと現在のデータを結合することができます。

- アプリケーション使用中の任意の時点で、DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER をコールできます。戻された番号を格納し、後でその値を使用してその時点におけるデータに対してフラッシュバック問合せを実行できます。このプロシージャをコールする前に COMMIT を実行し、データベースに整合性がある状態にして、後で復帰できるようにしてください。

フラッシュバック問合せの制限事項

- 表の構造を変更するいくつかの DDL（列の削除 / 変更、表の移動、パーティションの削除、表 / パーティションの切捨てなど）では、表に対する古い UNDO データは無効になります。このような DDL 文が実行された時点より古いデータのスナップショットを取得することはできません。このような問合せ実行すると、ORA-01466 エラーが発生します。この制限事項は、表の記憶域属性（PCTFREE、INITTRANS、MAXTRANS など）を変更する DDL 操作には適用されません。

- DBMS_FLASHBACK.ENABLE_AT_TIME または AS OF TIMESTAMP 句で指定された時間には SCN 値がマップされます。現在は、SCN 値と時間のマッピングは、データベース起動後 5 分ごとに記録されます。このため、指定された時間が最大 5 分間切り捨てられたように表示される場合があります。

  たとえば、SCN 値 1000 および 1005 が、午前 8 時 41 分および 8 時 46 分にそれぞれマップされていると仮定します。午前 8 時 41 分 00 秒と 8 時 45 分 59 秒の間のいずれかの時間に対するフラッシュバック問合せは、SCN 1000 にマップされ、午前 8 時 46 分に対するフラッシュバック問合せは、SCN 1005 にマップされます。

  このような時間と SCN 値のマッピングによって、表の作成直後の時点でのフラッシュバック問合せでは ORA-01466 エラーが発生する可能性があります。この場合、SCN ベースのフラッシュバック問合せを使用すると、より確実にデータの過去のスナップショットを取得できます。

- フラッシュバック問合せで使用される SCN は 5 分ごとにのみ記録されるため、DDL 操作の少し後の時間または SCN を指定しても、データベースは DDL 操作より少し前の SCN を使用する場合があります。そのため、DDL 操作の直後の時点に対するフラッシュバック問合せを実行する場合に、前述の制限事項と同じエラーを取得することがあります。

- 現在、フラッシュバック問合せ機能では、最大 5 日間まで時間の追跡を保持します。この期間は、実時間ではなく、サーバーのアップタイムを反映します。たとえば、この期間中にサーバーが 1 日ダウンした場合は、最高 6 日前を指定できます。これより前のデータを問い合せるには、日付と時間ではなく SCN を指定する必要があります。

DELETE の実行前など、必要があると判断した場合に SCN を記録しておく必要があります。

- DBMS_FLASHBACK パッケージを使用する場合、別の時間に対してフラッシュバックを使用可能にするには、フラッシュバックを一度使用禁止にしてください。ENABLE / DISABLE のペアはネストできません。DISABLE を何度も連続してコールすることはできますが、次の例に示すように、これによって多少余分なオーバーヘッドが発生します。
- フラッシュバック問合せによって影響を受けるのは表データの状態のみです。問合せ中は、現行状態のデータ・ディクショナリが使用されます。たとえば、データを格納した後にキャラクタ・セットを変更した場合は、問合せには現在のキャラクタ・セットを使用します。
- リモート表に対してデータベース・リンクを通じてフラッシュバック問合せを実行することはできません。
- データ・ディクショナリの V$ ビューから過去のデータは取得できません。このようなビューにフラッシュバック問合せを実行すると、現在のデータを戻します。データ・ディクショナリの他のビュー（USER_TABLES など）にもフラッシュバック問合せを実行できます。
- SYS ユーザーは、DBMS_FLASHBACK パッケージをコールできません。このユーザーは、AS OF 句を使用して、フラッシュバック問合せを実行できます。
- マテリアライズド・ビューに対するフラッシュバック問合せには、クエリー・リライトの効果はありません。

フラッシュバック問合せの使用のヒント

- 優れたパフォーマンスを得るには、DBMS_STATS パッケージを使用して、フラッシュバック問合せに含まれるすべての表に統計情報を生成し、統計情報を現在の状態で保持します。フラッシュバック問合せは、常に、これらの統計情報に依存するコストベース・オプティマイザを使用します。
- 5 分ごとに丸め誤差が発生する可能性がないため、データが重要なすべての操作で AS OF SCN 句を使用します。時間指定の利便性がより重要な非定型問合せおよびレポートにも AS OF TIMESTAMP 句を使用します。
- 表のフラッシュバック問合せは、表が作成された後数分間は動作しない場合があるため、すべての表（元のデータが含まれた表および一時記憶域に使用された表）が作成されたことを事前に確認してください。このヒントは、表がすでに設定されている本番環境よりも、フラッシュバック問合せを表示またはテストする場合に適用されます。
- フラッシュバック問合せを試すと、誤ってコミットされていないデータやフラッシュバック・モードにセッションを残すこともあります。COMMIT 文または ROLLBACK 文（必要に応じて）、およびフラッシュバック問合せを実行する SQL スクリプトの先頭にある DBMS_FLASHBACK.DISABLE へのコールを常に含むようにしてください。

- フラッシュバック問合せのパフォーマンスは、再作成の必要があるデータ量によって異なります。フラッシュバック問合せは、全表スキャンが必要な問合せではなく、主に索引に使用する少量のデータの集合の選択に使用します。全表スキャンをする必要がある場合は、問合せへのパラレル・ヒントの追加を検討します。
- ビューを定義する SELECT 文の AS OF 句を使用して、過去のデータを参照するビューを作成できます。SYSDATE から引いて相対時間を指定すると、過去の時間が各問合せで再計算されます。次に例を示します。

```
CREATE VIEW hour_ago AS
  SELECT * FROM EMPLOYEES AS OF
    TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

  この方法を使用する場合、夏時間およびうるう年では、結果に異常が発生することがあります。たとえば、夏時間に変更した直後に SYSDATE - 1 は、23 または 25 時間前とみなされる場合があります。
- タイムスタンプまたは SCN 引数としてファンクション戻り値を使用して、AS OF 句で使用されている過去の時間の計算または取得ができます。たとえば、INTERVAL 値を SYSTIMESTAMP 関数に加算または減算すると、日時の計算ができます。
- 自己結合または集合演算（INTERSECT、MINUS など）の AS OF 句は、2 つの異なる時間のデータを抽出または比較するために使用できます。CREATE TABLE AS SELECT 文または INSERT INTO TABLE SELECT 文の問合せに含めて、結果を格納できます。
- 記述する SQL には、比較および単一の問合せへの結果の格納が可能な AS OF 句を使用します。DBMS_FLASHBACK パッケージに対するコールを、制御しない SQL または過去の同じ時刻を使用する複数の連続した問合せの SQL の周辺に置きます。
- AS OF SCN 構成の句を使用する場合、戻される SCN を検索するには、まず DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER を使用します。
- 過去に有効だったものではなく、現在の各国語およびその他の設定で、すべての処理が実行されます。

フラッシュバック問合せの例

フラッシュバック問合せメカニズムは柔軟なため、多くの場合に使用できます。次に例を示します。

- 過去に存在していたデータの問合せ。
- 現在のデータと過去のデータの比較。個々の行の比較、または共通部分や結合の検索などのより複雑な比較が可能です。
- 削除または変更されたデータのリカバリ。
- DATE 値または TIMESTAMP 値を使用した特定の時間の参照、または SCN の記録。

過去のデータの取得例

```
-- Find out someone's salary at the beginning of a specified day.
-- Because of the limit on undo information, the date must be recent.

set serveroutput on;
set echo off;

-- Use a single SQL query.
SELECT salary FROM employees AS OF TIMESTAMP to_timestamp('20-FEB-2002','DD-MON-YY')
WHERE employee_id = 205;

-- Use the DBMS_FLASHBACK package. It involves more code, but several
-- operations can be performed all referring to the same past time.
DECLARE
  the_salary NUMBER;
  num_employees NUMBER;
BEGIN
  dbms_flashback.enable_at_time(to_timestamp('20-FEB-2002','DD-MON-YY'));
  SELECT salary INTO the_salary FROM employees WHERE employee_id = 205;
  dbms_output.put_line('Salary = ' || the_salary);
  SELECT COUNT(*) INTO num_employees FROM employees;
  dbms_output.put_line('Number of employees = ' || num_employees);
  dbms_flashback.disable;
END;
/
```

時間のない日付は、その日の一番始めを表すことに注意してください。時間に格納されるマッピング・データ割当て量には制限があるため、このような問合せは過去数日間以外検索できない可能性があります。さらに前の問合せを行うには、必要に応じてSCNを格納しておく必要があります。SCN値を使用しても、必要な時点まで保存されているUNDOデータがない場合は、データは使用できません。

不適切に更新または削除されたデータのリカバリ例

次の例では、不適切な更新を行って変更をコミットし、その直後にフラッシュバック問合せを使用して古い情報をリカバリします。

```
-- In this example, the deletion of a manager triggers the accidental
-- deletion of everyone in the reporting chain.
-- Using flashback query, we can recover and re-insert the missing employees.

------------------------ Setup -------------------------------
set echo off;
set serveroutput on size 50000;

-- KEEP_SCN is a temporary table to store SCNs that we are interested in.
DROP TABLE keep_scn;
```

```
CREATE TABLE keep_scn (scn NUMBER);
DELETE FROM keep_scn;
COMMIT;
EXECUTE dbms_flashback.disable;

DROP TABLE my_employees;
CREATE TABLE my_employees (
  employee_no   NUMBER(5) PRIMARY KEY,
  employee_name VARCHAR2(20),
  employee_mgr  NUMBER(5)
     CONSTRAINT mgr_fkey REFERENCES my_employees ON DELETE CASCADE,
  salary        NUMBER,
  hiredate      DATE );

DELETE FROM my_employees;

-- Now populate the company with employees.
INSERT INTO my_employees values (1, 'Dennis Potter', null, 1000000, '5-jul-91');
INSERT INTO my_employees values (10, 'Margaret O''Neil', 1, 500000, '12-aug-94');
INSERT INTO my_employees values (20, 'Charles Evans', 10, 250000, '13-dec-97');
INSERT INTO my_employees values (100, 'Roger Smith', 20, 200000, '3-feb-96');
INSERT INTO my_employees values (200, 'Juan Hernandez', 100, 150000, '22-mar-98');
INSERT INTO my_employees values (210, 'Jonathan Takeda', 100, 100000, '11-apr-97');
INSERT INTO my_employees values (220, 'Nancy Schoenfeld', 100, 100000, '18-sep-95');
INSERT INTO my_employees values (300, 'Edward Ngai', 210, 75000, '4-nov-96');
INSERT INTO my_employees values (310, 'Amit Sharma', 210, 65000, '3-may-95');
COMMIT;

-- Wait a little more than 5 minutes to avoid ORA-01466 error, because the
-- table is so new.  Only needed in demo situations like this.
EXECUTE dbms_output.put_line('Pausing for 5 minutes...');
EXECUTE dbms_lock.sleep(320);

------------------------- Real work -------------------------------
-- Show the entire org
SELECT lpad(' ', 2*(level-1)) || employee_name "Original Org Chart"
  FROM my_employees CONNECT BY PRIOR employee_no = employee_mgr
  START WITH employee_no = 1 ORDER BY LEVEL;

-- Store this snapshot for later access through FlashBack.
DECLARE
  I number;
BEGIN
  I := dbms_flashback.get_system_change_number;
  INSERT INTO keep_scn VALUES (I);
  COMMIT;
END;
```

```
/

-- Now Roger decides it's time to retire, but the HR department does
-- the transaction incorrectly. The DELETE CASCADE deletes Roger's organization.
DELETE FROM my_employees WHERE employee_name = 'Roger Smith';
COMMIT;

-- Notice that all of Roger's employees are now gone.
SELECT lpad(' ', 2*(level-1)) || employee_name "Post-Roger Org Chart"
  FROM my_employees CONNECT BY PRIOR employee_no = employee_mgr
  START WITH employee_no = 1 ORDER BY LEVEL;

-- Let's put back Roger's organization.
DECLARE
  restore_scn NUMBER;
  rogers_emp NUMBER;
  rogers_mgr NUMBER;
-- We use AS OF clauses on both the main query and the subquery.
  CURSOR c1(the_scn NUMBER) IS
   SELECT employee_no, employee_name, employee_mgr, salary, hiredate
     FROM my_employees AS OF SCN the_scn
     CONNECT BY PRIOR employee_no = employee_mgr
     START WITH employee_no =
       (SELECT employee_no FROM my_employees AS OF SCN the_scn
         WHERE employee_name = 'Roger Smith');
   c1_rec c1%ROWTYPE;

BEGIN

  SELECT scn INTO restore_scn FROM keep_scn;
--  dbms_flashback.enable_at_system_change_number(restore_scn);
  dbms_output.put_line('Using system change number: ' || restore_scn);

  SELECT employee_no, employee_mgr INTO rogers_emp, rogers_mgr
    FROM my_employees AS OF SCN restore_scn
    WHERE employee_name = 'Roger Smith';

  dbms_output.put_line('Roger was employee #' || rogers_emp ||
    ' and his manager was employee #' || rogers_mgr);

-- Open c1 at the point specified by restore_scn.
  OPEN c1(restore_scn);

-- Only the cursor uses past information, so we can insert into the original
-- table within the loop.
    LOOP
      FETCH c1 INTO c1_rec;
```

```
      EXIT WHEN c1%NOTFOUND;
      IF c1_rec.employee_mgr = rogers_emp THEN
-- If someone reported directly to Roger, we restore them but make
-- them report to Roger's manager.
        dbms_output.put_line('Inserting employee who used to report to Roger: ' ||
c1_rec.employee_name);
        INSERT INTO my_employees VALUES (c1_rec.employee_no,
          c1_rec.employee_name, rogers_mgr, c1_rec.salary, c1_rec.hiredate);
      ELSIF c1_rec.employee_no != rogers_emp THEN
-- If someone didn't report directly to Roger, we restore them with the same
-- manager as before.
        dbms_output.put_line('Inserting employee who didn''t report directly to
Roger: ' || c1_rec.employee_name);
        INSERT INTO my_employees VALUES (c1_rec.employee_no,
          c1_rec.employee_name, c1_rec.employee_mgr, c1_rec.salary,
          c1_rec.hiredate);
      END IF;
    END LOOP;
END;
/

-- Now show that Roger's org is back.
SELECT lpad(' ', 2*(level-1)) || employee_name "Restored Org Chart"
  FROM my_employees CONNECT BY PRIOR employee_no = employee_mgr
  START WITH employee_no = 1 ORDER BY LEVEL;
```

システム変更番号（SCN）での AS OF 句の使用例

```
-- A precise method of specifying the flashback point uses the SCN directly,
-- instead of specifying a timestamp, but also requires that we store the SCN in
-- advance. Go to this extra trouble whenever it is crucial to retrieve data
-- from a precise point.

set serveroutput on;
-- Make a copy of the EMPLOYEES table without the constraints, so we
-- can make arbitrary changes.
DROP TABLE employees2;
CREATE TABLE employees2 AS SELECT * FROM employees;
COMMIT;
-- Wait for slightly more than 5 minutes, because the table is so new.
-- Only needed in demo situations like this.
EXECUTE dbms_lock.sleep(320);

SELECT count(*) FROM employees2 WHERE salary = 9000;

DECLARE
```

```
  TYPE employee_cursor IS REF CURSOR;
  c employee_cursor;
  cvar employees%ROWTYPE;
  old_scn NUMBER;
BEGIN
  COMMIT;
  dbms_flashback.disable;
  old_scn := dbms_flashback.get_system_change_number;
  DELETE FROM employees2 WHERE salary = 9000;
  COMMIT;

-- Those updates and deletes were in error. We need to recover the data.
-- Use the data as it existed immediately before the update and delete.
  OPEN c FOR 'SELECT * FROM employees2 AS OF SCN :1 WHERE salary = 9000'
    USING old_scn;
  LOOP
    FETCH c INTO cvar;
    EXIT WHEN c%NOTFOUND;
    dbms_output.put_line('Recovering employee: ' || cvar.last_name);
    INSERT INTO employees2 VALUES cvar;
  END LOOP;
  COMMIT;
END;
/

select count(*) from employees2 where salary = 9000;
```

今日追加したすべての行の検索例

```
-- TRUNC(SYSDATE) gives us midnight this morning, the start of the day.
-- By using the MINUS operator, we can see which rows are present now but
-- were not present at the start of the day. They might be entirely new
-- employees, or employees whose data was changed today so their row doesn't
-- match the row from yesterday.

DROP TABLE employees_changed_today;
CREATE TABLE employees_changed_today AS
  SELECT * FROM employees
  MINUS
  SELECT * FROM employees AS OF TIMESTAMP TRUNC(SYSDATE);
```

副問合せを使用したフラッシュバック問合せの実行例

```
-- A query similar to the previous one shows the flashback query
-- can be inside a subquery.
```

```
SELECT MAX(salary) FROM
(
  SELECT * FROM employees
  MINUS
  SELECT * FROM employees AS OF TIMESTAMP TRUNC(SYSDATE)
);
```

DBMS_FLASHBACK を使用した明示的カーソルおよび暗黙的カーソルの使用例

```
-- It is more efficient to use an explicit cursor than a FOR loop with an
-- implicit cursor. To use flashback query effectively, it helps to understand
-- different techniques for using cursors.

-- Before: many employees have null values for this column.
select count(*) from employees where commission_pct is null;

-- Most efficient technique. Open the cursor, disable flashback, then use
-- the older data in DML statements on the current table.
DECLARE
  TYPE employee_cursor IS REF CURSOR;
  c employee_cursor;
  cvar employees%ROWTYPE;
BEGIN
-- Make sure at the start that we aren't in the middle of a transaction.
  COMMIT;
-- Make sure we are not already in flashback mode.
  dbms_flashback.disable;

-- Flashback to a known time.
  dbms_flashback.enable_at_time(SYSDATE - 1);
-- Open a cursor to retrieve data from the past.
  OPEN c FOR 'SELECT * FROM employees WHERE employee_id < 200';
-- When flashback is disabled, the cursor continues to refer to past data,
-- and we can perform DML again.
  dbms_flashback.disable;

  LOOP
    FETCH c INTO cvar;
    EXIT WHEN c%NOTFOUND;
-- If the employee had a null value for commission_pct yesterday,
-- then overwrite the current row, setting the value to zero.
    IF cvar.commission_pct IS NULL THEN
       cvar.commission_pct := 0;
       UPDATE employees SET ROW = cvar WHERE employee_id = cvar.employee_id;
    END IF;
```

```
    COMMIT;
  END LOOP;
END;
/

-- After: no employee should have a null value for this column.
select count(*) from employees where commission_pct is null;
```

また、ループの反復ごとに DBMS_FLASHBACK.DISABLE をコールするため多少効率は落ちますが、次のように暗黙的カーソルを使用することもできます。

```
-- Setup: create an empty table with the same definition as EMPLOYEES.
drop table yesterdays_employees;
create table yesterdays_employees as select * from employees where 1 = 0;
set serveroutput on size 500000;

-- Less efficient technique. To allow DML statements against the current
-- table within the loop body, DISABLE must be called for each loop iteration.

-- Make sure database is in a consistent state.
COMMIT;
-- First make sure flashback is turned off initially.
EXECUTE DBMS_FLASHBACK.DISABLE;
-- Then set it to a known value.
EXECUTE DBMS_FLASHBACK.ENABLE_AT_TIME(SYSDATE - 1);

BEGIN
-- The FOR loop is examining data values from the past.
  FOR c in (SELECT * FROM employees WHERE employee_id < 200)
  LOOP
-- DISABLE gets called multiple times, but that's OK.
    DBMS_FLASHBACK.DISABLE;

   dbms_output.put_line('Inserting employee #' || c.employee_id);

-- Because flashback is disabled within the loop body, we can access the
-- present state of the data, and issue DML statements to undo the changes
-- or store the old data somewhere else.
    INSERT INTO yesterdays_employees VALUES c;
  END LOOP;
END;
/

-- After the loop, once more make sure flashback is turned off.
EXECUTE DBMS_FLASHBACK.DISABLE;

select count(*) from yesterdays_employees;
```

8

動的 SQL 文のコーディング

動的 SQL とは、SQL 文を実行時に動的に作成できるプログラミング手法のことです。SQL 文のすべてのテキストはコンパイル時にわかっているわけではないため、動的 SQL を使用すると、より汎用的で柔軟なアプリケーションを作成できます。たとえば、動的 SQL を使用すると、実行時まで名前がわからない表に対して操作するプロシージャも作成できます。

Oracle では、次の 2 つの方法によって、PL/SQL アプリケーションで動的 SQL を実装できます。

- 動的 SQL 文を PL/SQL ブロック内に直接入れることができるネイティブ動的 SQL
- DBMS_SQL パッケージ内のプロシージャのコール

この章の内容は次のとおりです。

DBMS_SQL パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

動的 SQL の概要

動的 SQL を使用すると、実行時まで完全なテキストがわからない SQL 文を参照するプログラムを作成できます。動的 SQL の詳細を説明する前に、静的 SQL を明確に定義しておくと、動的 SQL をよく理解できます。静的 SQL 文は、何度実行しても文の意味は変わりません。静的 SQL 文はコンパイル時に完全なテキストがわかっているもので、そのメリットは次のとおりです。

- コンパイルが正常終了すると、SQL 文が有効なデータベース・オブジェクトを参照していることが検証されます。
- コンパイルが正常終了すると、データベース・オブジェクトのアクセスに必要な権限があることが検証されます。
- 静的 SQL のパフォーマンスは、通常、動的 SQL より良好です。

静的 SQL には前述のメリットがあるため、動的 SQL の使用は、静的 SQL では目的が達成できない場合、または静的 SQL の方が動的 SQL より使用が煩雑になる場合のみにする必要があります。ただし、静的 SQL には、動的 SQL の使用で解決できる制約もあります。PL/SQL プロシージャで実行する必要がある SQL 文のテキストは、常に完全にわかっているわけではありません。実行する SQL 文を定義するユーザー入力をプログラムで受け入れることもあれば、正しいアクションを決定するために、プログラムでなんらかの処理作業をする必要があることもあります。このような場合は動的 SQL を使用します。

たとえば、データ・ウェアハウス環境のレポート・アプリケーションでは、表の正確な名前が実行時までわからない場合があります。表の名前は、その四半期の西暦年と四半期の開始月に従って付けられる場合があります。たとえば、`INV_01_1997`、`INV_04_1997`、`INV_07_1997`、`INV_10_1997`、`INV_01_1998` などの名前です。この場合、レポート・アプリケーションで動的 SQL を使用し、表の名前を実行時に指定するようにできます。

また、ユーザーが選択するソート順で複合問合せを明示的に実行する場合もあります。問合せのコーディングを 2 回行うかわりに、`ORDER BY` 句を使用して、指定された `ORDER BY` 句を含む問合せを動的に作成できます。

動的 SQL プログラムは、再コンパイルを行わずにデータ定義の変更に対応できます。この点で、動的 SQL は静的 SQL よりはるかに柔軟です。動的 SQL は、様々な環境に簡単に適用できるため、再利用可能なコードを記述できます。

また、動的 SQL では、DDL 文および静的 SQL のみで作成されたプログラムではサポートされていないその他の SQL 文を実行できます。

動的 SQL を使用する理由

動的 SQL は、実行する操作が静的 SQL ではサポートされていない場合、または PL/SQL プロシージャによって実行される正確な SQL 文がわからない場合に使用します。このような SQL 文は、ユーザー入力に依存することがあり、プログラムが実行する処理に依存することもあります。次の項では、動的 SQL を使用する典型的な場面と、動的 SQL を使用して解決できる典型的な問題を説明します。

PL/SQL での DDL 文および SCL 文の実行

PL/SQL では、次の種類の文は静的 SQL では実行できないため、動的 SQL で実行する必要があります。

- データ定義言語（DDL）文（CREATE、DROP、GRANT、REVOKE など）
- セッション制御言語（SCL）文（ALTER SESSION、SET ROLE など）

> **参照：** DDL 文および SCL 文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

また、SELECT 文で TABLE 句を使用するには、動的 SQL を使用する必要があります。たとえば、次の PL/SQL ブロックには、TABLE 句およびネイティブ動的 SQL を使用する SELECT 文が含まれています。

```
CREATE TYPE t_emp AS OBJECT (id NUMBER, name VARCHAR2(20))
/
CREATE TYPE t_emplist AS TABLE OF t_emp
/

CREATE TABLE dept_new (id NUMBER, emps t_emplist)
    NESTED TABLE emps STORE AS emp_table;

INSERT INTO dept_new VALUES (
    10,
    t_emplist(
        t_emp(1, 'SCOTT'),
        t_emp(2, 'BRUCE')));

DECLARE
    deptid NUMBER;
    ename  VARCHAR2(20);
BEGIN
    EXECUTE IMMEDIATE 'SELECT d.id, e.name
        FROM dept_new d, TABLE(d.emps) e  -- not allowed in static SQL
                                          -- in PL/SQL
        WHERE e.id = 1'
        INTO deptid, ename;
END;
/
```

動的問合せの実行

動的 SQL を使用すると、動的問合せ（実行時まで完全なテキストがわからない問合せ）を実行するアプリケーションを作成できます。動的問合せは、様々なアプリケーションで使用する必要があります。次に例を示します。

- ユーザーが実行時に問合せ検索またはソート基準を入力または選択できるアプリケーション
- ユーザーが実行時にオプティマイザ・ヒントを入力または選択できるアプリケーション
- 表のデータ定義が常に変更されるデータベースを問い合せるアプリケーション
- 新しい表が頻繁に作成されるデータベースを問い合せるアプリケーション

前述の例については、8-16 ページの「動的 SQL を使用した問合せの例」を参照してください。問合せの例については、8-7 ページの「ネイティブ動的 SQL の使用例」を参照してください。

コンパイル時には存在しないデータベース・オブジェクトの参照

多くのアプリケーションでは、定期的に生成されるデータと対話する必要があります。たとえば、表定義はコンパイル時にわかっても、表の名前はわからない場合があります。

動的 SQL を使用すると、表の名前を実行時まで待って指定できるため、この問題を解決できます。たとえば、8-2 ページの「動的 SQL の概要」で説明されているデータ・ウェアハウス・アプリケーションの例では、新しい表が四半期ごとに生成され、この表の定義は常に同じです。この場合、次のような動的 SQL 問合せを使用すると、実行時にユーザーが表の名前を指定できます。

```
CREATE OR REPLACE PROCEDURE query_invoice(
       month VARCHAR2,
       year VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
    c cur_typ;
    query_str VARCHAR2(200);
    inv_num NUMBER;
    inv_cust VARCHAR2(20);
    inv_amt NUMBER;
BEGIN
    query_str := 'SELECT num, cust, amt FROM inv_' || month ||'_'|| year
      || ' WHERE invnum = :id';
    OPEN c FOR query_str USING inv_num;
    LOOP
        FETCH c INTO inv_num, inv_cust, inv_amt;
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
```

実行の動的最適化

動的 SQL を使用すると、ヒントを動的に SQL 文に連結して実行を最適化する方法で SQL 文を作成できます。これによって、再コンパイルしなくても、現在のデータベースの詳細情報に基づいてヒントを変更できます。

たとえば、次のプロシージャでは、a_hint という変数を使用して、ユーザーが SELECT 文にヒント・オプションを渡せるようになっています。

```
CREATE OR REPLACE PROCEDURE query_emp
      (a_hint VARCHAR2) AS
   TYPE cur_typ IS REF CURSOR;
   c cur_typ;
BEGIN
   OPEN c FOR 'SELECT ' || a_hint ||
      ' empno, ename, sal, job FROM emp WHERE empno = 7566';
      -- process
END;
/
```

この例では、ユーザーは、a_hint として次の値のいずれか、またはその他の有効なヒント・オプションを渡すことができます。

```
a_hint = '/*+ ALL_ROWS */'
a_hint = '/*+ FIRST_ROWS */'
a_hint = '/*+ CHOOSE */'
```

> **参照：** ヒントの使用の詳細は『Oracle9*i* データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』を参照してください。

動的 PL/SQL ブロックの実行

無名 PL/SQL ブロックを実行するには、EXECUTE IMMEDIATE 文を使用できます。実行時にブロックの内容を構成することによって、柔軟性を向上できます。

たとえば、イベント番号を受け入れ、それをそのイベントのハンドラにディスパッチするアプリケーションを作成します。ハンドラの名前は EVENT_HANDLER_*event_num* という形式で、*event_num* がイベントの番号です。1 つの方法として、ディスパッチャをスイッチ文として実装する方法があります。この場合、該当するハンドラにコードが静的コールを実行して、それぞれのイベントを処理します。このコードは、新しいイベント用のハンドラが追加されるたびにディスパッチャ・コードを更新する必要があるため、あまり拡張可能ではありません。

```
CREATE OR REPLACE PROCEDURE event_handler_1(param number) AS BEGIN
   -- process event
   RETURN;
END;
/
```

```
CREATE OR REPLACE PROCEDURE event_handler_2(param number) AS BEGIN
   -- process event
   RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_handler_3(param number) AS BEGIN
   -- process event
   RETURN;
END;
/

CREATE OR REPLACE PROCEDURE event_dispatcher
   (event number, param number) IS
BEGIN
  IF (event = 1) THEN
    EVENT_HANDLER_1(param);
  ELSIF (event = 2) THEN
    EVENT_HANDLER_2(param);
  ELSIF (event = 3) THEN
    EVENT_HANDLER_3(param);
  END IF;
END;
/
```

ネイティブ動的 SQL を使用すると、次のように、より小さく、柔軟なイベント・ディスパッチャを作成できます。

```
CREATE OR REPLACE PROCEDURE event_dispatcher
   (event NUMBER, param NUMBER) IS
BEGIN
  EXECUTE IMMEDIATE
     'BEGIN
        EVENT_HANDLER_' || to_char(event) || '(:1);
     END;'
  USING param;
END;
/
```

実行者権限を使用した動的操作の実行

動的 SQL とともに実行者権限の機能を使用すると、実行者権限およびスキーマに基づいて動的 SQL 文を発行するアプリケーションを作成できます。実行者権限および動的 SQL という 2 つの機能によって、実行者のデータおよびモジュールに対して操作およびアクセスできる再利用可能なアプリケーション・サブコンポーネントを作成できます。

参照： 実行者権限およびネイティブ動的 SQL の使用の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

ネイティブ動的 SQL の使用例

この使用例には、ネイティブ動的 SQL を使用して次の操作を実行する方法が説明されています。

- DDL および DML 操作の実行
- 単一行問合せおよび複数行問合せの実行

この使用例のデータベースは、ある企業の人材データベース hr で、次のデータ・モデルが含まれています。

offices というマスター表には、この企業のすべての事務所のリストが含まれています。offices 表は、次のように定義されています。

Column Name	Null?	Type
LOCATION	NOT_NULL	VARCHAR2(200)

emp_*location* 表は複数あり、これには社員情報が含まれています。*location* は事務所のある都市の名前です。たとえば、emp_houston という表には、ヒューストン事務所の社員の情報が含まれ、emp_boston という表には、ボストン事務所の社員の情報が含まれています。

各 emp_location 表は、次のように定義されています。

Column Name	Null?	Type
EMPNO	NOT_NULL	NUMBER(4)
ENAME	NOT_NULL	VARCHAR2(10)
JOB	NOT_NULL	VARCHAR2(9)
SAL	NOT_NULL	NUMBER(7,2)
DEPTNO	NOT_NULL	NUMBER(2)

次の項では、hr データベース内のデータに対して実行できる様々なネイティブ動的 SQL 操作を説明します。

ネイティブ動的 SQL による DML 操作例

次のネイティブ動的 SQL プロシージャによって、特定の役職を持つ社員全員の給料が増額されます。

```
CREATE OR REPLACE PROCEDURE salary_raise (raise_percent NUMBER, job VARCHAR2) IS
    TYPE loc_array_type IS TABLE OF VARCHAR2(40)
        INDEX BY binary_integer;
    dml_str VARCHAR2(200);
    loc_array loc_array_type;
```

```
BEGIN
    -- bulk fetch the list of office locations
    SELECT location BULK COLLECT INTO loc_array
        FROM offices;
    -- for each location, give a raise to employees with the given 'job'
    FOR i IN loc_array.first..loc_array.last LOOP
        dml_str := 'UPDATE emp_' || loc_array(i)
        || ' SET sal = sal * (1+(:raise_percent/100))'
        || ' WHERE job = :job_title';
    EXECUTE IMMEDIATE dml_str USING raise_percent, job;
    END LOOP;
END;
/
SHOW ERRORS;
```

ネイティブ動的 SQL による DDL 操作例

EXECUTE IMMEDIATE 文は、DDL 操作を実行できます。たとえば、次のプロシージャは、事務所の所在地を追加します。

```
CREATE OR REPLACE PROCEDURE add_location (loc VARCHAR2) IS
BEGIN
    -- insert new location in master table
    INSERT INTO offices VALUES (loc);
    -- create an employee information table
    EXECUTE IMMEDIATE
    'CREATE TABLE ' || 'emp_' || loc ||
    '(
        empno   NUMBER(4) NOT NULL,
        ename   VARCHAR2(10),
        job     VARCHAR2(9),
        sal     NUMBER(7,2),
        deptno  NUMBER(2)
    )';
END;
/
SHOW ERRORS;
```

次のプロシージャは、事務所の所在地を削除します。

```
CREATE OR REPLACE PROCEDURE drop_location (loc VARCHAR2) IS
BEGIN
    -- delete the employee table for location 'loc'
    EXECUTE IMMEDIATE 'DROP TABLE ' || 'emp_' || loc;
    -- remove location from master table
    DELETE FROM offices WHERE location = loc;
END;
```

```
/
SHOW ERRORS;
```

ネイティブ動的 SQL による単一行問合せ例

EXECUTE IMMEDIATE 文は、動的な単一行問合せを実行できます。USING 句にバインド変数を指定し、この文の INTO 句に指定されているターゲットに結果の行をフェッチできます。

次の関数は、特定の所在地で特定の職務を遂行している社員数を取得します。

```
CREATE OR REPLACE FUNCTION get_num_of_employees (loc VARCHAR2, job VARCHAR2)
    RETURN NUMBER IS
    query_str VARCHAR2(1000);
    num_of_employees NUMBER;
BEGIN
    query_str := 'SELECT COUNT(*) FROM '
        || ' emp_' || loc
        || ' WHERE job = :job_title';
    EXECUTE IMMEDIATE query_str
        INTO num_of_employees
        USING job;
    RETURN num_of_employees;
END;
/
SHOW ERRORS;
```

ネイティブ動的 SQL による複数行問合せ例

動的な複数行問合せは、OPEN-FOR 文、FETCH 文および CLOSE 文を使用して実行できます。たとえば、次のプロシージャは、特定の所在地で特定の職種についているすべての社員をリストします。

```
CREATE OR REPLACE PROCEDURE list_employees(loc VARCHAR2, job VARCHAR2) IS
    TYPE cur_typ IS REF CURSOR;
    c           cur_typ;
    query_str   VARCHAR2(1000);
    emp_name    VARCHAR2(20);
    emp_num     NUMBER;
BEGIN
    query_str := 'SELECT ename, empno FROM emp_' || loc
        || ' WHERE job = :job_title';
    -- find employees who perform the specified job
    OPEN c FOR query_str USING job;
    LOOP
        FETCH c INTO emp_name, emp_num;
```

```
        EXIT WHEN c%NOTFOUND;
        -- process row here
    END LOOP;
    CLOSE c;
END;
/
SHOW ERRORS;
```

ネイティブ動的 SQL または DBMS_SQL パッケージの選択

Oracle では、PL/SQL 内で動的 SQL を使用するために、ネイティブ動的 SQL および DBMS_SQL パッケージという 2 つの方法を提供しています。ネイティブ動的 SQL を使用すると、動的 SQL 文を PL/SQL コード内に直接入れることができます。このような動的な文には、DML 文（問合せを含む）、無名 PL/SQL ブロック、DDL 文、トランザクション制御文、セッション制御文があります。

ほとんどのネイティブ動的 SQL 文を処理するには、EXECUTE IMMEDIATE 文を使用します。複数行問合せ（SELECT 文）を処理するには、OPEN-FOR 文、FETCH 文および CLOSE 文を使用します。

注意： ネイティブ動的 SQL を使用するには、COMPATIBLE 初期化パラメータを 8.1.0 以上に設定してください。COMPATIBLE パラメータの詳細は、『Oracle9*i* データベース移行ガイド』を参照してください。

DBMS_SQL パッケージは、SQL 文を動的に処理する API を提供する PL/SQL ライブラリです。DBMS_SQL パッケージには、カーソルのオープン、カーソルの解析、バインドの提供などを行うプロシージャがあります。DBMS_SQL パッケージを使用するプログラムは、このパッケージをコールして動的 SQL 操作を実行します。

次の項では、この 2 つの方法のメリットを詳しく説明します。

> **参照：** ネイティブ動的 SQL の使用方法の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。DBMS_SQL パッケージの使用方法の詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。『PL/SQL ユーザーズ・ガイドおよびリファレンス』では、ネイティブ動的 SQL は単に動的 SQL と呼ばれています。

ネイティブ動的 SQL のメリット

ネイティブ動的 SQL には、DBMS_SQL パッケージと比べて次のようなメリットがあります。

使用しやすさ

ネイティブ動的 SQL は SQL と統合されているため、現在 PL/SQL コードで静的 SQL を使用している方法と同じ方法で使用できます。ネイティブ動的 SQL コードは、通常、DBMS_SQL パッケージを使用した同等のコードより簡潔で読みやすくなります。

DBMS_SQL パッケージを使用すると、数多くのプロシージャおよびファンクションを厳密な順序に従ってコールする必要があり、単純な操作を実行するのみでも多量のコードが必要になります。ネイティブ動的 SQL を使用すると、それほど複雑になることはありません。

表 8-1 に、同じ操作を DBMS_SQL パッケージを使用して実行したときとネイティブ動的 SQL を使用して行ったときに必要なコード量の違いを示します。

表 8-1 DBMS_SQL パッケージおよびネイティブ動的 SQL でのコード量の比較

DBMS_SQL パッケージ

```
CREATE PROCEDURE insert_into_table (
      table_name  VARCHAR2,
      deptnumber  NUMBER,
      deptname    VARCHAR2,
      location    VARCHAR2) IS
   cur_hdl         INTEGER;
   stmt_str        VARCHAR2(200);
   rows_processed  BINARY_INTEGER;

BEGIN
   stmt_str := 'INSERT INTO ' ||
      table_name || ' VALUES
      (:deptno, :dname, :loc)';

   -- open cursor
   cur_hdl := dbms_sql.open_cursor;

   -- parse cursor
   dbms_sql.parse(cur_hdl, stmt_str,
      dbms_sql.native);

   -- supply binds
   dbms_sql.bind_variable
      (cur_hdl, ':deptno', deptnumber);
   dbms_sql.bind_variable
      (cur_hdl, ':dname', deptname);
   dbms_sql.bind_variable
      (cur_hdl, ':loc', location);

    -- execute cursor
    rows_processed :=
    dbms_sql.execute(cur_hdl);

    -- close cursor
    dbms_sql.close_cursor(cur_hdl);

END;
/
SHOW ERRORS;
```

ネイティブ動的 SQL

```
CREATE PROCEDURE insert_into_table (
      table_name  VARCHAR2,
      deptnumber  NUMBER,
      deptname    VARCHAR2,
      location    VARCHAR2) IS
   stmt_str    VARCHAR2(200);

BEGIN
   stmt_str := 'INSERT INTO ' ||
      table_name || ' values
      (:deptno, :dname, :loc)';

   EXECUTE IMMEDIATE stmt_str
      USING
      deptnumber, deptname, location;

END;
/
SHOW ERRORS;
```

DBMS_SQL パッケージを超える速度

PL/SQL インタプリタにはネイティブ動的 SQL のサポートが組み込まれているため、PL/SQL でのネイティブ動的 SQL のパフォーマンスは、静的 SQL のパフォーマンスと同等になります。ネイティブ動的 SQL を使用するプログラムは、DBMS_SQL パッケージを使用するプログラムよりはるかに高速です。通常、ネイティブ動的 SQL 文は、DBMS_SQL コールを使用する同等の文より 1.5 ～ 3 倍パフォーマンスが向上します（パフォーマンスの向上は、アプリケーションによっても異なります）。

ネイティブ動的 SQL では、文の準備手順、バインド手順および実行手順を 1 つの操作の中にまとめます。これによって、データ・コピーおよびプロシージャ・コールによるオーバーヘッドが最小化され、パフォーマンスが向上します。

DBMS_SQL パッケージはプロシージャ API に基づいているため、プロシージャ・コールが多くなり、データをコピーするオーバーヘッドが発生します。DBMS_SQL パッケージは、変数がバインドされるたびに PL/SQL バインド変数を自身の領域にコピーし、実行時に使用できるようにします。同様に、フェッチを実行するたびに、まず DBMS_SQL パッケージが管理する領域にデータがコピーされ、次に、フェッチされたデータが 1 列ずつ適切な PL/SQL 変数にコピーされます。この結果、データのコピーによるオーバーヘッドがかなり多くなります。

パフォーマンス上のヒント : バインド変数の使用 ネイティブ動的 SQL および DBMS_SQL パッケージのどちらの場合も、バインド変数を使用することでパフォーマンスを向上できます。これは、バインド変数の使用によって、Oracle が複数の SQL 文に対して 1 つのカーソルを共有できるためです。

たとえば、次のネイティブ動的 SQL コードではバインド変数は使用されていません。

```
CREATE OR REPLACE PROCEDURE del_dept (
   my_deptno  dept.deptno%TYPE) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = ' || to_char (my_deptno);
END;
/
SHOW ERRORS;
```

各 my_deptno 変数に対して新しいカーソルが作成されます。これによってリソース競合が発生し、パフォーマンスが低下する可能性があります。かわりに、次の例のように、my_deptno をバインド変数として使用します。

```
CREATE OR REPLACE PROCEDURE del_dept (
   my_deptno  dept.deptno%TYPE) IS
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :1' USING my_deptno;
END;
/
SHOW ERRORS;
```

ここでは、1 つのカーソルがバインド変数 my_deptno の異なる複数の値に再利用されているため、パフォーマンスおよびスケーラビリティが向上します。

ユーザー定義型のサポート

ネイティブ動的 SQL は、PL/SQL で静的 SQL がサポートしているすべての型（ユーザー定義のオブジェクト、コレクション、REF などのユーザー定義型を含む）をサポートしています。DBMS_SQL パッケージでは、ユーザー定義型はサポートされません。

注意： DBMS_SQL パッケージは、配列を限定的にサポートします。詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

レコードへのフェッチのサポート

ネイティブ動的 SQL および静的 SQL は、どちらもレコードへのフェッチをサポートしますが、DBMS_SQL パッケージはこれをサポートしません。ネイティブ動的 SQL の場合は、問合せの結果の行を PL/SQL レコードに直接フェッチできます。

次の例では、問合せ結果の行が emp_rec レコードにフェッチされます。

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    c EmpCurTyp;
    emp_rec emp%ROWTYPE;
    stmt_str VARCHAR2(200);
    e_job emp.job%TYPE;

BEGIN
   stmt_str := 'SELECT * FROM emp WHERE job = :1';
    -- in a multi-row query
    OPEN c FOR stmt_str USING 'MANAGER';
    LOOP
        FETCH c INTO emp_rec;
        EXIT WHEN c%NOTFOUND;
    END LOOP;
    CLOSE c;
    -- in a single-row query
    EXECUTE IMMEDIATE stmt_str INTO emp_rec USING 'PRESIDENT';

END;
/
```

DBMS_SQL パッケージのメリット

DBMS_SQL パッケージには、ネイティブ動的 SQL と比べて次のようなメリットがあります。

クライアント側プログラムでのサポート

DBMS_SQL パッケージはクライアント側プログラム内でサポートされていますが、ネイティブ動的 SQL はサポートされていません。クライアント側プログラムからの DBMS_SQL パッケージへのすべてのコールは、PL/SQL リモート・プロシージャ・コール（RPC）に変換されます。変数のバインド、変数の定義または文の実行が必要なときに、これらのコールが発生します。

DESCRIBE のサポート

DBMS_SQL パッケージの DESCRIBE_COLUMNS プロシージャを使用すると、DBMS_SQL によってオープンおよび解析されるカーソルの列を記述できます。このプロシージャの機能は、SQL*Plus の DESCRIBE コマンドと同様です。ネイティブ動的 SQL には DESCRIBE 機能はありません。

RETURNING 句を使用した複数行の更新および削除

DBMS_SQL パッケージは、複数行を更新または削除する RETURNING 句を指定した文をサポートします。ネイティブ動的 SQL では、行が 1 つ戻される場合にのみ RETURNING 句をサポートします。

> **参照：** RETURNING 句を使用する DBMS_SQL パッケージ・コードおよびネイティブ動的 SQL コードの例は、8-18 ページの「動的 SQL を使用した RETURNING 句を持つ DML の実行例」を参照してください。

32KB を超える大規模 SQL 文のサポート

DBMS_SQL パッケージは、32KB を超える大規模な SQL 文をサポートします。ネイティブ動的 SQL はサポートしません。

SQL 文の再利用

DBMS_SQL パッケージの PARSE プロシージャは、SQL 文を 1 回解析します。最初に解析した後、この文は異なるバインド引数を指定して複数回使用できます。

ネイティブ動的 SQL では、SQL 文を使用するたびにその文を準備します。文の準備では、通常、解析、最適化および計画の生成が行われます。この準備によって、パフォーマンスはわずかに低下しますが、ネイティブ動的 SQL によって、全体ではパフォーマンスは向上します。

DBMS_SQL パッケージ・コードおよびネイティブ動的 SQL コードの例

次の例に、DBMS_SQL パッケージを使用した場合、およびネイティブ動的 SQL を使用した場合の操作に必要なコードの違いを示します。次の操作について例を示します。

- 問合せ
- DML 操作
- DML リターニング操作

一般に、ネイティブ動的 SQL コードの方が読みやすく簡潔なため、開発生産性が向上します。

動的 SQL を使用した問合せの例

次に、バインド変数 1 つ（:jobname）および選択列 2 つ（ename および sal）で構成される動的問合せ文を示します。

```
stmt_str := 'SELECT ename, sal FROM emp WHERE job = :jobname';
```

この例では、emp 表の job 列の職種が SALESMAN の社員を問い合せます。表 8-2 に、DBMS_SQL パッケージおよびネイティブ動的 SQL を使用してこの問合せを行うコード例を示します。

表 8-2 DBMS_SQL パッケージおよびネイティブ動的 SQL を使用した問合せ

DBMS_SQL の問合せ操作	ネイティブ動的 SQL の問合せ操作

DECLARE
  stmt_str varchar2(200);
  cur_hdl int;
  rows_processed int;
  name varchar2(10);
  salary int;
BEGIN
cur_hdl := dbms_sql.open_cursor; -- open cursor
stmt_str := 'SELECT ename, sal FROM emp WHERE
job = :jobname';
dbms_sql.parse(cur_hdl, stmt_str,
dbms_sql.native);
 	
-- supply binds (bind by name)
dbms_sql.bind_variable(
  cur_hdl, 'jobname', 'SALESMAN');
 	
-- describe defines
dbms_sql.define_column(cur_hdl, 1, name, 200);
dbms_sql.define_column(cur_hdl, 2, salary);
 	
rows_processed := dbms_sql.execute(cur_hdl); --
execute
 	
LOOP
  -- fetch a row
  IF dbms_sql.fetch_rows(cur_hdl) > 0 then
 	
  -- fetch columns from the row
  dbms_sql.column_value(cur_hdl, 1, name);
  dbms_sql.column_value(cur_hdl, 2, salary);
 	
  -- <process data>
 	
  ELSE
  EXIT;
  END IF;
END LOOP;
dbms_sql.close_cursor(cur_hdl); -- close cursor
END;
/
	DECLARE
	  TYPE EmpCurTyp IS REF CURSOR;
	  cur EmpCurTyp;
	  stmt_str VARCHAR2(200);
	  name VARCHAR2(20);
	  salary NUMBER;
	BEGIN
	  stmt_str := 'SELECT ename, sal FROM emp
	  WHERE job = :1';
	  OPEN cur FOR stmt_str USING 'SALESMAN';
	
	LOOP
	  FETCH cur INTO name, salary;
	  EXIT WHEN cur%NOTFOUND;
	  -- <process data>
	END LOOP;
	CLOSE cur;
	END;
	/

動的 SQL を使用した DML の実行例

次に、列が 3 つある表に対する動的 INSERT 文を示します。

```
stmt_str := 'INSERT INTO dept_new VALUES (:deptno, :dname, :loc)';
```

この例は、PL/SQL 変数 deptnumber、deptname および location を列値とする新しい行を挿入します。表 8-3 に、DBMS_SQL パッケージおよびネイティブ動的 SQL を使用してこの DML 操作を行うコード例を示します。

表 8-3 DBMS_SQL パッケージおよびネイティブ動的 SQL を使用した DML 操作

DBMS_SQL の DML 操作	ネイティブ動的 SQL の DML 操作

DECLARE
  stmt_str VARCHAR2(200);
  cur_hdl NUMBER;
  deptnumber NUMBER := 99;
  deptname VARCHAR2(20);
  location VARCHAR2(10);
  rows_processed NUMBER;
BEGIN
  stmt_str := 'INSERT INTO dept_new VALUES
  (:deptno, :dname, :loc)';
  cur_hdl := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(
  cur_hdl, stmt_str, DBMS_SQL.NATIVE);
  -- supply binds
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':deptno', deptnumber);
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':dname', deptname);
  DBMS_SQL.BIND_VARIABLE
  (cur_hdl, ':loc', location);
  rows_processed := dbms_sql.execute(cur_hdl);
  -- execute
  DBMS_SQL.CLOSE_CURSOR(cur_hdl); -- close
END;
/
	DECLARE
	  stmt_str VARCHAR2(200);
	  deptnumber NUMBER := 99;
	  deptname VARCHAR2(20);
	  location VARCHAR2(10);
	BEGIN
	  stmt_str := 'INSERT INTO dept_new VALUES
	  (:deptno, :dname, :loc)';
	  EXECUTE IMMEDIATE stmt_str
	  USING deptnumber, deptname, location;
	END;
	/

動的 SQL を使用した RETURNING 句を持つ DML の実行例

次に、部門の所在地を更新し、部門の名前を戻す動的 UPDATE 文を示します。

```
stmt_str := 'UPDATE dept_new
            SET loc = :newloc
            WHERE deptno = :deptno
            RETURNING dname INTO :dname';
```

表 8-4 に、DBMS_SQL パッケージおよびネイティブ動的 SQL を使用してこの操作を行うコード例を示します。

表 8-4 DBMS_SQL パッケージおよびネイティブ動的 SQL を使用した DML RETURNING 操作

DBMS_SQL の DML RETURNING 操作	ネイティブ動的 SQL の DML RETURNING 操作

DECLARE
  deptname_array dbms_sql.Varchar2_Table;
  cur_hdl INT;
  stmt_str VARCHAR2(200);
  location VARCHAR2(20);
  deptnumber NUMBER := 10;
  rows_processed NUMBER;
BEGIN
 stmt_str := 'UPDATE dept_new
  SET loc = :newloc
  WHERE deptno = :deptno
  RETURNING dname INTO :dname';
 	
  cur_hdl := dbms_sql.open_cursor;
  dbms_sql.parse
  (cur_hdl, stmt_str, dbms_sql.native);
  -- supply binds
  dbms_sql.bind_variable
  (cur_hdl, ':newloc', location);
  dbms_sql.bind_variable
  (cur_hdl, ':deptno', deptnumber);
  dbms_sql.bind_array
  (cur_hdl, ':dname', deptname_array);
  -- execute cursor
  rows_processed := dbms_sql.execute(cur_hdl);
  -- get RETURNING column into OUT bind array
  dbms_sql.variable_value
  (cur_hdl, ':dname', deptname_array);
  dbms_sql.close_cursor(cur_hdl);
END;
/
	DECLARE
	  deptname_array dbms_sql.Varchar2_Table;
	  stmt_str VARCHAR2(200);
	  location VARCHAR2(20);
	  deptnumber NUMBER := 10;
	  deptname VARCHAR2(20);
	BEGIN
	  stmt_str := 'UPDATE dept_new
	  SET loc = :newloc
	  WHERE deptno = :deptno
	  RETURNING dname INTO :dname';
	  EXECUTE IMMEDIATE stmt_str
	  USING location, deptnumber, OUT deptname;
	END;
	/

PL/SQL 以外の言語における動的 SQL の使用

この章では、動的 SQL に対する PL/SQL サポートについて説明しましたが、その他の言語でも動的 SQL をコールできます。

- C/C++ を使用する場合は、OCI を使用して動的 SQL をコールするか、Pro*C/C++ プリコンパイラで動的 SQL 拡張を C コードに追加できます。
- COBOL を使用する場合は、Pro*COBOL プリコンパイラで動的 SQL 拡張を COBOL コードに追加できます。
- Java を使用する場合は、JDBC で動的 SQL を使用するアプリケーションを開発できます。

動的 SQL の実行に OCI、Pro*C/C++ または Pro*COBOL を使用するアプリケーションの場合は、PL/SQL ストアド・プロシージャおよびストアド・ファンクション内のネイティブ動的 SQL に切り替えることを検討する必要があります。クライアント側アプリケーションからの動的 SQL 操作に必要なネットワーク・ラウンドトリップは、パフォーマンスを低下させる場合があります。ストアド・プロシージャはサーバーに常駐できるため、ネットワークのオーバヘッドをなくすことができます。PL/SQL ストアド・プロシージャおよびストアド・ファンクションは、OCI、Pro*C/C++ または Pro*COBOL のアプリケーションからコールできます。

> **参照：** Oracle ストアド・プロシージャおよびストアド・ファンクションをその他の言語からコールする方法の詳細は、次のマニュアルを参照してください。
>
> - 『Oracle Call Interface プログラマーズ・ガイド』
> - 『Pro*C/C++ Precompiler プログラマーズ・ガイド』
> - 『Pro*COBOL Precompiler プログラマーズ・ガイド』
> - 『Oracle9*i* Java Stored Procedures Developer's Guide』

SQL の INSERT 文および UPDATE 文での PL/SQL レコードの使用

表の行を更新または挿入する場合、PL/SQL レコードの各フィールドを列挙できますが、結果コードは特に読込みやメンテナンスはできません。かわりに、これらの文では直接 PL/SQL レコードを使用できます。最も有効な方法は、%ROWTYPE 属性を使用してレコードを宣言することで、SQL 表と同じフィールドになります。

```
DECLARE
    emp_rec emp%ROWTYPE;
BEGIN
   emp_rec.eno := 1500;
   emp_rec.ename := 'Steven Hill';
   emp_rec.sal := '40000';
-- A %ROWTYPE value can fill in all the row fields.
    INSERT INTO emp VALUES emp_rec;

-- The fields of a %ROWTYPE can completely replace the table columns.
    UPDATE emp SET ROW = emp_rec WHERE eno = 100;
END;
/
```

この方法を使用すると PL/SQL 変数および PL/SQL 型を SQL DML 文と統合できますが、PL/SQL レコードを動的 SQL 文のバインド変数としては使用できません。

> **参照：** PL/SQL レコードの詳細は『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

9

プロシージャおよびパッケージの使用

この章では、Oracle がアプリケーション開発用に提供するプロシージャ機能のいくつかを説明します。この章の内容は次のとおりです。

- PL/SQL プログラム・ユニットの概要
- PL/SQL ラッパーによる PL/SQL コードの隠蔽
- リモート依存性
- カーソル変数
- PL/SQL コンパイル時のエラー処理
- PL/SQL のランタイム・エラー処理
- ストアド・プロシージャのデバッグ
- ストアド・プロシージャのコール
- リモート・プロシージャのコール
- SQL 式からのストアド・ファンクションのコール
- ファンクションからの大量のデータの戻し
- 独自の集計関数のコード化

PL/SQL プログラム・ユニットの概要

PL/SQL は、ブロック構造化プログラミング言語です。この言語が持ついくつかの機能を使用すると、高性能のデータベース・アプリケーションを容易に作成できます。たとえば、PL/SQL は、ループ文や条件文など標準 SQL にはないプロシージャ構造を提供します。

PL/SQL ブロック内部で SQL の DML 文を直接入力できます。また、Oracle が提供するプロシージャを使用して、DDL 文を実行できます。

PL/SQL コードはサーバー上で実行されるため、PL/SQL を使用するとデータベース・アプリケーションのかなりの部分を集中化でき、メンテナンス性およびセキュリティが強化されます。また、クライアント / サーバー・アプリケーションでは、ネットワークのオーバーヘッドも大幅に削減できます。

注意： Oracle Forms など、一部の Oracle のツール製品には PL/SQL エンジンが組み込まれ、ローカルで PL/SQL を実行できます。

また、一部のデータベース・アプリケーションでは、埋込み SQL または OCI を使用する 3GL プログラムのかわりに PL/SQL を使用できます。

PL/SQL プログラム・ユニットには、次のものが含まれます。

- 無名ブロック
- ストアド・プログラム・ユニット（プロシージャ、ファンクションおよびパッケージ）
- トリガー

参照： PL/SQL パッケージの構文および操作例は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Oracle データベース・サーバーの PL/SQL パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。

無名ブロック

無名ブロックとは、名前のない PL/SQL プログラム・ユニットのことで、実行文を囲む BEGIN および END キーワードを明示的に指定する必要のないブロックです。無名ブロックは、オプションの宣言部分、実行可能部分および 1 つまたは複数のオプションの例外ハンドラで構成されます。

宣言部には PL/SQL の変数、例外およびカーソルを宣言します。実行可能部分には PL/SQL コードおよび SQL 文を含むネストされたブロックを含めることができます。例外ハンドラには、例外状況が発生したときに、事前定義の PL/SQL 例外（NO_DATA_FOUND または ZERO_DIVIDE）として、またはユーザー定義の例外としてコールされるコードが含まれています。

次の無名 PL/SQL ブロックの例では、DBMS_OUTPUT パッケージを使用して、Emp_tab 表の部門 20 に所属するすべての従業員の名前を表示します。

```
DECLARE
   Emp_name    VARCHAR2(10);
   Cursor      c1 IS SELECT Ename FROM Emp_tab
                  WHERE Deptno = 20;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO Emp_name;
      EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Emp_name);
   END LOOP;
END;
```

注意： SQL*Plus を使用してこのブロックをテストする場合は、DBMS_OUTPUT プロシージャを使用する出力（たとえば、PUT_LINE）がアクティブになるように、SET SERVEROUTPUT ON を入力します。また、出力をアクティブにするには、スラッシュ（/）を付けて例を終了します。

参照： DBMS_OUTPUT パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

例外を使用すると、PL/SQL プログラム・ロジック内の Oracle エラー条件を処理できます。これによって、使用中のアプリケーションで、クライアント・アプリケーションを異常終了させるようなエラーをサーバーが発行しないようにできます。次の無名ブロックは、事前定義された Oracle 例外 NO_DATA_FOUND を処理します（この例外が処理されない場合は、ORA-01403 エラーが発生します）。

```
DECLARE
   Emp_number   INTEGER := 9999;
   Emp_name     VARCHAR2(10);
BEGIN
   SELECT Ename INTO Emp_name FROM Emp_tab
      WHERE Empno = Emp_number;   -- no such number
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No such employee: ' || Emp_number);
END;
```

また、独自の例外を定義してブロックの宣言部に宣言し、それをブロックの例外部分に指定できます。次に例を示します。

```
DECLARE
   Emp_name           VARCHAR2(10);
   Emp_number         INTEGER;
   Empno_out_of_range EXCEPTION;
BEGIN
   Emp_number := 10001;
   IF Emp_number > 9999 OR Emp_number < 1000 THEN
      RAISE Empno_out_of_range;
   ELSE
      SELECT Ename INTO Emp_name FROM Emp_tab
         WHERE Empno = Emp_number;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
END IF;
EXCEPTION
   WHEN Empno_out_of_range THEN
      DBMS_OUTPUT.PUT_LINE('Employee number ' || Emp_number ||
       ' is out of range.');
END;
```

参照： 9-34 ページの「PL/SQL のランタイム・エラー処理」および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

無名ブロックは、通常、SQL*Plus などのツール製品から対話形式で使用するか、プリコンパイラ、OCI または SQL*Module アプリケーションで使用します。通常、ストアド・プロシージャをコールするか、カーソル変数をオープンするために使用します。

参照： 9-29 ページの「カーソル変数」を参照してください。

ストアド・プログラム・ユニット（プロシージャ、ファンクションおよびパッケージ）

ストアド・プロシージャ、ファンクションおよびパッケージは、次の特長を持つ PL/SQL プログラム・ユニットです。

- 固有の名前を持っています。
- パラメータをとり、値を戻すことができます。
- データ・ディクショナリに格納されます。
- 多数のユーザーがコールできます。

注意： ストアド・プロシージャという用語は、包括的な意味で使用している場合があり、その場合にはストアド・プロシージャおよびストアド・ファンクションの両方を表しています。プロシージャとファンクションの違いは、ファンクションはコール側に対して常に値を 1 つ戻し、プロシージャはコール側に値を戻さないということのみです。

プロシージャおよびファンクションのネーミング

プロシージャまたはファンクションはデータベース内に格納されるため、名前を付ける必要があります。名前を付けることによって、他のストアド・プロシージャと区別され、アプリケーションでコールすることができます。パブリックで参照できるスキーマ内の個々のプロシージャまたはファンクションは、一意の名前を持つ必要があります。その名前は、有効な PL/SQL 識別子である必要があります。

注意： SQL*Module によって生成されたスタブを使用してストアド・プロシージャをコールする場合、ストアド・プロシージャ名は、コール側ホストの 3GL 言語（Ada や C など）の有効な識別子である必要もあります。

プロシージャおよびファンクションのパラメータ

ストアド・プロシージャおよびファンクションには、パラメータを指定できます。次に、9-2 ページの「無名ブロック」で説明されている無名ブロックに類似したストアド・プロシージャの例を示します。

注意： 次の文を実行するには、CREATE OR REPLACE PROCEDURE... を使用してください。

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
   Emp_name       VARCHAR2(10);
   CURSOR         c1 (Depno NUMBER) IS
                     SELECT Ename FROM Emp_tab
                        WHERE deptno = Depno;
BEGIN
   OPEN c1(Dept_num);
   LOOP
      FETCH c1 INTO Emp_name;
      EXIT WHEN C1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Emp_name);
   END LOOP;
   CLOSE c1;
END;
```

このストアド・プロシージャの例では、部門番号が入力パラメータになっています。入力パラメータは、パラメータ化されたカーソル c1 のオープン時に使用されます。

プロシージャの仮パラメータには、次の 3 つの主要な属性があります。

パラメータ属性	説明
名前	名前は、有効な PL/SQL 識別子である必要があります。
モード	入力のみのパラメータ（IN）、出力のみのパラメータ（OUT）、入力と出力の両方のパラメータ（IN OUT）のどれであるかを示します。モードを指定しないと、IN が想定されます。
データ型	パラメータのデータ型は、標準 PL/SQL データ型です。

パラメータ・モード パラメータ・モードは、仮パラメータの動作を定義します。3 つのパラメータ・モード、IN（デフォルト）、OUT および IN OUT は、どのようなサブプログラムを使用する場合にも使用できます。ただし、OUT モードおよび IN OUT モードはファンクションには使用しないでください。ファンクションの目的は、引数をとらず、1 つの値を戻すことです。ファンクションが複数の値を戻すようなプログラミングは、効率的ではありません。また、サブプログラムでローカルではない変数の値を変更するような副作用をファンクションが与えないようにしてください。

表 9-1 に、パラメータ・モードの概要を示します。

> **参照：** パラメータ・モードの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

表 9-1 パラメータ・モード

IN	OUT	IN OUT
デフォルト	指定する必要があります。	指定する必要があります。
値をサブプログラムに渡します。	値をコール側に戻します。	初期値をサブプログラムに渡し、更新された値をコール側に戻します。
仮パラメータが定数として動作します。	仮パラメータが未初期化変数として動作します。	仮パラメータが初期化変数として動作します。
仮パラメータに値を割り当てることはできません。	仮パラメータを式の中で使用できません。値を割り当てる必要があります。	仮パラメータに値を割り当てる必要があります。
実パラメータを、定数、初期化変数、リテラルまたは式にできます。	実パラメータは変数である必要があります。	実パラメータは変数である必要があります。

パラメータのデータ型 仮パラメータのデータ型は、次のいずれかで構成されています。

- NUMBER や VARCHAR2 などの無制約の型名
- %TYPE 属性や %ROWTYPE 属性を使用して制約される型

> **注意：** NUMBER(2) または VARCHAR2(20) などの数値が制約される型は、パラメータ・リストでは使用できません。

%TYPE 属性および %ROWTYPE 属性 型属性 %TYPE および %ROWTYPE は、パラメータを制約するために使用します。たとえば、9-5 ページの「プロシージャおよびファンクションのパラメータ」にある Get_emp_names プロシージャの仕様部は、次のように作成できます。

```
PROCEDURE Get_emp_names(Dept_num IN Emp_tab.Deptno%TYPE)
```

これによって、Dept_num パラメータが Emp_tab 表の Deptno 列と同じデータ型を取ります。%TYPE（または %ROWTYPE）を使用した宣言を作成する場合は、列および表が使用可能である必要があります。

表の列の型が変更されてもアプリケーション・コードを変更する必要がないため、%TYPE の使用をお薦めします。

Get_emp_names プロシージャがパッケージの一部である場合は、前に宣言したパブリック（パッケージ）変数を使用して、パラメータのデータ型を制約できます。次に例を示します。

```
Dept_number     number(2);
...
PROCEDURE Get_emp_names(Dept_num IN Dept_number%TYPE);
```

%ROWTYPE 属性は、指定された表のすべての列を含むレコードを作成するために使用します。次の例では、Get_emp_rec プロシージャを定義して、指定された empno に関する PL/SQL レコード内の Emp_tab 表のすべての列を戻します。

> **注意：** 次の文を実行するには、CREATE OR REPLACE PROCEDURE... を使用してください。

```
PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                       Emp_ret     OUT Emp_tab%ROWTYPE) IS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

次のようにして、PL/SQL ブロックからこのプロシージャをコールできます。

```
DECLARE
   Emp_row      Emp_tab%ROWTYPE;     -- declare a record matching a
                                     -- row in the Emp_tab table
BEGIN
   Get_emp_rec(7499, Emp_row);   -- call for Emp_tab# 7499
   DBMS_OUTPUT.PUT(Emp_row.Ename || ' '                || Emp_row.Empno);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Job || ' ' || Emp_row.Mgr);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Hiredate   || ' ' || Emp_row.Sal);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Comm || ' '|| Emp_row.Deptno);
   DBMS_OUTPUT.NEW_LINE;
END;
```

ストアド・ファンクションは、%ROWTYPE を使用して宣言される値を戻すこともできます。次に例を示します。

```
FUNCTION Get_emp_rec (Dept_num IN Emp_tab.Deptno%TYPE)
   RETURN Emp_tab%ROWTYPE IS ...
```

表およびレコード PL/SQL 表を、パラメータとしてストアド・プロシージャおよびファンクションに渡せます。レコードの表も、パラメータとして渡せます。

注意： リモート・プロシージャに PL/SQL 表やレコードなどのユーザー定義型を渡す場合、タイプ・チェック者がソースを検証できるように PL/SQL で同じ定義を使用するには、余分なループバック DBLINK を作成してください。PL/SQL のコンパイル時に、両方のソースが同じ位置から引き出されます。

デフォルトのパラメータ値 パラメータには、デフォルト値を設定できます。パラメータにデフォルト値を設定するには、DEFAULT キーワードまたは代入演算子を使用します。たとえば、Get_emp_names プロシージャの仕様部は次のように作成できます。

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER DEFAULT 20) IS ...
```

または

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER := 20) IS ...
```

パラメータにデフォルト値を使用する場合は、プロシージャのコール時に実パラメータ・リストからそのパラメータを省略できます。コール時にパラメータ値を指定すると、デフォルト値がオーバーライドされます。

注意： 無名 PL/SQL ブロック内とは異なり、ストアド・プロシージャ内では、変数、カーソルおよび例外の宣言の前にキーワード DECLARE を使用しないでください。使用するとエラーが発生します。

ストアド・プロシージャおよびファンクションの作成

プロシージャまたはファンクションを作成するには、テキスト・エディタを使用します。プロシージャの先頭に、次の文を記述します。

```
CREATE PROCEDURE Procedure_name AS   ...
```

たとえば、9-7 ページの「%TYPE 属性および %ROWTYPE 属性」にある例を使用する場合は、次のコードを含む get_emp.sql というテキスト（ソース）・ファイルを作成します。

```
CREATE PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                              Emp_ret     OUT Emp_tab%ROWTYPE) AS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
/
```

その後、SQL*Plus などの対話形式のツール製品を使用して次の文を入力し、プロシージャを含むテキスト・ファイルをロードします。

```
SQL> @get_emp
```

プロシージャが、get_emp.sql ファイル（sql は、デフォルトのファイル拡張子）から現行のスキーマにロードされます。コードの終わりにはスラッシュ（/）を付けてください。これはコードの一部ではありません。プロシージャのロードをアクティブにするためのものです。

ファンクションを格納するには、CREATE [OR REPLACE] FUNCTION... 文を使用します。

注意： 新しいプロシージャを作成する場合、通常は CREATE OR REPLACE... PROCEDURE 文を使用する方が便利です。このコマンドは、同一スキーマ内の前のバージョンのプロシージャを新しいバージョンに置き換えます。ただし、これは警告なしで実行されます。

プロシージャ・パラメータ・リストの後にキーワード IS または AS を使用できます。

参照： CREATE PROCEDURE 文および CREATE FUNCTION 文の構文の詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

プロシージャおよびファンクションの作成に必要な権限 スタンドアロン・プロシージャまたはファンクション、あるいはパッケージ仕様部または本体を作成するには、次の権限が必要です。

- 自スキーマにプロシージャまたはパッケージを作成するには、CREATE PROCEDURE システム権限が必要です。他のユーザーのスキーマにプロシージャまたはパッケージを作成するには、CREATE ANY PROCEDURE システム権限が必要です。

注意： エラーなしで作成する（プロシージャまたはパッケージを正常にコンパイルする）には、さらに次の権限が必要です。

- プロシージャまたはパッケージの所有者は、コード本体内で参照されるすべてのオブジェクトに必要なオブジェクト権限を明示的に付与されている必要があります。
- 所有者は、ロールを使用して必要な権限を取得することはできません。

プロシージャまたはパッケージの所有者の権限が変更された場合、実行前にそのプロシージャを再認証する必要があります。参照オブジェクトに必要な権限が、そのプロシージャまたはパッケージの所有者から取り消されている場合、そのプロシージャは実行できません。

プロシージャの EXECUTE 権限があれば、他のユーザーが所有するプロシージャを実行できます。権限が付与されたユーザーは、そのプロシージャの所有者のセキュリティ・ドメインでプロシージャを実行します。このため、ユーザーは、プロシージャが参照するオブジェクトの権限を得る必要はありません。これによって、データベース・アプリケーションおよびそのユーザーによるさらに統制のとれた効率的なセキュリティ計画が可能になります。また、すべてのプロシージャおよびパッケージが（SYSTEM 表領域内の）データ・ディクショナリに格納されます。プロシージャおよびパッケージを作成するユーザーが使用できる領域の容量は、割当て制限によっては制御されません。

注意： パッケージの作成にはソートが必要です。このため、パッケージを作成するユーザーは、対応付けられている一時表領域にソート・セグメントを作成できる必要があります。

参照： 9-42 ページの「プロシージャの実行に必要な権限」を参照してください。

ストアド・プロシージャおよびファンクションの変更

ストアド・プロシージャまたはファンクションを変更するには、DROP PROCEDURE 文または DROP FUNCTION 文を使用して削除（DROP）した後、CREATE PROCEDURE 文または CREATE FUNCTION 文を使用して再作成する必要があります。または、CREATE OR REPLACE PROCEDURE 文または CREATE OR REPLACE FUNCTION 文を使用します。この文は、プロシージャまたはファンクションが存在する場合、まずそれを削除してから、指定どおりに再作成します。

注意： プロシージャまたはファンクションは警告なく削除されます。

PL/SQL プロシージャおよびファンクションの動作の制御

データベースにパラメータを設定することによって、PL/SQL プロシージャおよびファンクションの実行時の動作を制御できます。これらのパラメータは、すべてのまたは特定の PL/SQL プロシージャおよびファンクションに適用できます。次に例を示します。

```
-- Set default behavior for PL/SQL procedures and functions
ALTER SESSION SET PLSQL_V2_COMPATIBILITY = TRUE;
-- Use a different setting for this one procedure
ALTER PROCEDURE myproc SET PLSQL_V2_COMPATIBILITY = FALSE;
```

パラメータは、プロシージャ、ファンクション、パッケージ、型およびトリガーに適用できます。適用後は、オブジェクトが無効とされ、CREATE OR REPLACE または自動再コンパイルによってこれらのスキーマ・オブジェクトが更新されると、その設定が適用されます。スキーマ・オブジェクトが削除された場合、そのオブジェクトに適用された設定のみが失われます。

ALL_PLSQL_SWITCH_SETTINGS カタログ・ビューおよび USER_PLSQL_SWITCH_SETTINGS カタログ・ビューを問い合せると、有効な設定を検索できます。ALL_PLSQL_SWITCHES ビューを問い合せると、アクセス可能な名前およびパラメータをすべて検索できます。また、アプリケーション内で、DBMS_DESCRIBE パッケージのファンクションをコールして、同様の情報を検索できます。

プロシージャおよびファンクションの削除

SQL 文の DROP PROCEDURE、DROP FUNCTION、DROP PACKAGE BODY および DROP PACKAGE を使用して、スタンドアロン・プロシージャ、スタンドアロン・ファンクション、パッケージ本体またはパッケージ全体を、それぞれ削除できます。DROP PACKAGE 文は、パッケージの仕様部と本体の両方を削除します。

次の文は、スキーマ内にある Old_sal_raise プロシージャを削除します。

```
DROP PROCEDURE Old_sal_raise;
```

プロシージャおよびファンクションの削除に必要な権限 プロシージャ、ファンクションまたはパッケージを削除するには、それらが自スキーマ内にあるか、または DROP ANY PROCEDURE 権限が必要です。パッケージ内の個々のプロシージャは削除できません。これらを削除せずに、パッケージ仕様部および本体を再作成する必要があります。

外部プロシージャ

Oracle サーバー上で実行する PL/SQL プロシージャは、3GL で作成された外部プロシージャをコールできます。3GL プロシージャは、Oracle サーバーのアドレス空間とは別のアドレス空間で実行されます。

> **参照：** 外部プロシージャの詳細は、第 10 章「外部プロシージャのコール」を参照してください。

PL/SQL パッケージ

パッケージとは、データベース内に格納されている関連プログラム・オブジェクト（プロシージャ、ファンクション、変数、定数、カーソル、例外など）がカプセル化されたコレクションです。

パッケージは、プロシージャおよびファンクションをスタンドアロンのスキーマ・オブジェクトとして作成するかわりに使用します。パッケージは、スタンドアロンのプロシージャおよびファンクションに比べて、多数のメリットがあります。たとえば、次のことができます。

- アプリケーション開発をより効率的に行えます。
- 権限をより効率的に付与できます。
- 依存スキーマ・オブジェクトを再コンパイルせずにパッケージ・オブジェクトを変更できます。
- Oracle で複数のパッケージ・オブジェクトを一度にメモリー内に読み込めます。
- パッケージ内のすべてのプロシージャおよびファンクションが使用できるグローバル変数およびグローバル・カーソルを、そのパッケージ内に含めることができます。
- プロシージャまたはファンクションをオーバーロードします。プロシージャをオーバーロードするということは、同一パッケージ内に同じ名前のプロシージャを複数作成するということです。それぞれのプロシージャが異なる数またはデータ型の引数をとることができます。

  > **参照：** サブプログラム名のオーバーロードの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

パッケージ仕様部は、パッケージの有効範囲外で参照できるパブリック型、変数、定数およびサブプログラムを宣言します。パッケージ本体は、パッケージ外のアプリケーションが参

照できないプライベート・オブジェクトのみでなく、仕様部で宣言されているオブジェクトも定義します。

PL/SQL パッケージ仕様部と本体の例 次に、Employee_management というパッケージのパッケージ本体を示します。パッケージには、1 つのストアド・ファンクションおよび 2 つのストアド・プロシージャが含まれています。このパッケージ本体は、ファンクションおよびプロシージャを定義します。

```
CREATE PACKAGE BODY Employee_management AS
   FUNCTION Hire_emp (Name VARCHAR2, Job VARCHAR2,
      Mgr NUMBER, Hiredate DATE, Sal NUMBER, Comm NUMBER,
      Deptno NUMBER) RETURN NUMBER IS
       New_empno    NUMBER(10);

-- This function accepts all arguments for the fields in
-- the employee table except for the employee number.
-- A value for this field is supplied by a sequence.
-- The function returns the sequence number generated
-- by the call to this function.

   BEGIN
      SELECT Emp_sequence.NEXTVAL INTO New_empno FROM dual;
      INSERT INTO Emp_tab VALUES (New_empno, Name, Job, Mgr,
         Hiredate, Sal, Comm, Deptno);
      RETURN (New_empno);
   END Hire_emp;

   PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- This procedure deletes the employee with an employee
-- number that corresponds to the argument Emp_id. If
-- no employee is found, then an exception is raised.

   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
      Raise_application_error(-20011, 'Invalid Employee
         Number: ' || TO_CHAR(Emp_id));
   END IF;
END fire_emp;

PROCEDURE Sal_raise (Emp_id IN NUMBER, Sal_incr IN NUMBER) AS

-- This procedure accepts two arguments. Emp_id is a
-- number that corresponds to an employee number.
-- SAL_INCR is the amount by which to increase the
-- employee's salary. If employee exists, then update
```

```
-- salary with increase.

   BEGIN
      UPDATE Emp_tab
         SET Sal = Sal + Sal_incr
         WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
         Raise_application_error(-20011, 'Invalid Employee
            Number: ' || TO_CHAR(Emp_id));
      END IF;
   END Sal_raise;
END Employee_management;
```

注意： この例を実行する場合、まず順序番号 Emp_sequence を作成します。次の SQL 文を使用して作成します。

```
SQL> CREATE SEQUENCE Emp_sequence
   > START WITH 8000 INCREMENT BY 10;
```

PL/SQL オブジェクト・サイズの制限

プロシージャ、ファンクション、トリガー、パッケージなどの PL/SQL ストアド・データベース・オブジェクトのサイズは、共有プール内の DIANA のサイズ（バイト単位）に制限されています。フラット化された DIANA/pcode のサイズは、UNIX では 64KB に制限されていますが、Windows などのデスクトップ・プラットフォームでは 32KB に制限されている場合があります。

ユーザーがアクセスできるもので最も密接に関連する数値は、データ・ディクショナリ・ビュー USER_OBJECT_SIZE の PARSED_SIZE です。これには、SYS.IDL_xxx$ 表に格納された DIANA のサイズがバイト単位で示されています。これは共有プールでのサイズではありません。（コンパイル中に使用される）PL/SQL コードの DIANA 部分のサイズは、システム表内より共有プール内で非常に大きくなります。

バージョンごとのサイズ制限 PL/SQL パッケージのサイズは、リリース 7.3 では約 128KB（解析サイズ）に制限されています。リリース 7.2 以下では、64KB に制限されています。

パッケージの作成

パッケージの各部は、異なる文を使用して作成します。パッケージ仕様部は、CREATE PACKAGE 文を使用して作成します。CREATE PACKAGE 文でパブリック・パッケージ・オブジェクトを宣言します。

パッケージ本体を作成するには、CREATE PACKAGE BODY 文を使用します。CREATE PACKAGE BODY 文は、パッケージ仕様部で宣言されているパブリック・プロシージャおよびファンクションの手続き型コードを定義します。

パッケージ本体にはプライベート（またはローカル）・パッケージ・プロシージャ、ファンクションおよび変数を定義することもできます。これらのオブジェクトは、同一パッケージの本体内の他のプロシージャおよびファンクションでのみアクセスできます。外部ユーザーはどの権限を持っていても参照できません。

初めてアプリケーションを開発する場合、CREATE PACKAGE 文または CREATE PACKAGE BODY 文に OR REPLACE 句を追加すると便利な場合があります。このオプションの効果は、警告なしでパッケージまたはパッケージ本体が削除されることです。CREATE 文は、次のようになります。

```
CREATE OR REPLACE PACKAGE Package_name AS ...
```

および

```
CREATE OR REPLACE PACKAGE BODY Package_name AS ...
```

パッケージ・オブジェクトの作成 パッケージ本体には、次のものを含めることができます。

- パッケージ仕様部に宣言されているプロシージャおよびファンクション
- パッケージ仕様部に宣言されているカーソルの定義
- パッケージ仕様部に宣言されていないローカル・プロシージャおよびファンクション
- ローカル変数

パッケージ仕様部に宣言されているプロシージャ、ファンクション、カーソルおよび変数はグローバルです。これらをコールまたは使用できるのは、パッケージに対する EXECUTE 権限を持つ外部ユーザーまたは EXECUTE ANY PROCEDURE 権限を持つ外部ユーザーです。

パッケージ本体を作成する場合は、本体に定義する個々のプロシージャが、パッケージ仕様部と同じパラメータ（名前、データ型およびモード）を持っていることを確認してください。パッケージ本体内のファンクションの場合は、パラメータと戻り型の名前およびデータ型が一致する必要があります。

パッケージの作成または削除に必要な権限 パッケージ仕様部またはパッケージ本体を作成または削除するために必要な権限は、スタンドアロン・プロシージャまたはファンクションの作成または削除に必要な権限と同じです。

> **参照：** 9-10 ページの「プロシージャおよびファンクションの作成に必要な権限」および 9-12 ページの「プロシージャおよびファンクションの削除に必要な権限」を参照してください。

パッケージおよびパッケージ・オブジェクトのネーミング

パッケージおよびパッケージ内のすべてのパブリック・オブジェクトの名前は、所定のスキーマ内で一意である必要があります。パッケージ仕様部およびその本体は、同じ名前であ

る必要があります。また、パッケージのメンバーの名前は、プロシージャ名の重複が必要な場合を除き、そのパッケージの有効範囲内で一意である必要があります。

パッケージの無効性およびセッションの状態

パッケージ・オブジェクトを参照する各セッションは、対応するパッケージの独自のインスタンスを持っています。この中には、パブリック変数、プライベート変数、カーソルおよび定数に対する持続状態が含まれます。セッションのパッケージ・インスタンス（仕様部または本体）のいずれかが、その後無効になり再コンパイルされると、そのセッションに対する他のすべての依存パッケージのインスタンス（状態を含む）が失われます。

たとえば、セッション S がパッケージ P1 および P2 をインスタンス化し、パッケージ P1 のプロシージャがパッケージ P2 のプロシージャをコールするとします。P1 が無効になり再コンパイルされる（たとえば、DDL 操作の結果として）と、P1 と P2 の両方のセッション S のインスタンスが失われます。このような状況で、セッションが無効なパッケージ依存のオブジェクトを使用しようとすると、1 回目は次のエラーが戻されます（string には文字列が入ります）。

```
ORA-04068: パッケージ string string string の既存状態は廃棄されました。
```

2 度目にセッションがこのようなパッケージ・コールを行うと、エラーは発生せずに、パッケージはセッションに対して再インスタンス化されます。

> **注意：** Oracle は、無効にしたパッケージをコールするセッションにこのメッセージを戻さないように最適化されています。したがって、前述の例では、セッション S が初めてパッケージ P2 をコールするときにこのメッセージが戻されますが、P1 をコールするときはこのメッセージは戻されません。

本番環境の多くでは、パッケージが無効になる DDL 操作は、通常、業務時間外に行われます。したがって、エンド・ユーザー・アプリケーションでは、このような状況は問題にならない可能性もあります。ただし、パッケージ仕様部または本体が業務時間中に無効になることがよくある場合は、パッケージ・コールが行われたときにそのエラーを検出するアプリケーションを作成することもできます。

Oracle 提供のパッケージ

データベースの機能性を拡張できるように、または PL/SQL で SQL 機能を使用できるように、Oracle データベースには多数のパッケージが組み込まれています。これらのパッケージはアプリケーションからコールできます。

これらのパッケージは、パッケージ所有者ではなく、コール側ユーザーとして実行されます。特に指定がないかぎり、パッケージは、同じ名前のパブリック・シノニムを使用してコールできます。

これらの Oracle 提供パッケージの詳細は、次のマニュアルを参照してください。

- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』
  これらのパッケージの多くは、特定の状況でのみ使用されます。汎用なパッケージは、DBMS_JOB、DBMS_LOB、DBMS_LOCK、DBMS_OUTPUT、DBMS_RANDOM、DBMS_SQL、DBMS_UTILITY、DBMS_XMLGEN、UTL_FILE、UTL_HTTP および UTL_SMTP などです。
- 『Oracle Spatial ユーザーズ・ガイドおよびリファレンス』

バルク・バインドの概要

Oracle は 2 つのエンジンを使用して、PL/SQL ブロックおよびサブプログラムを実行します。PL/SQL エンジンは手続き型の文を実行し、SQL エンジンは SQL 文を実行します。実行中は、すべての SQL 文がこの 2 つのエンジン間でコンテキストをスイッチングするため、パフォーマンスが低下します。

特定のブロックまたはサブプログラムの実行に必要なコンテキストのスイッチング回数を最小化すると、パフォーマンスを大幅に改善できます。バインド変数としてコレクション要素を使用するループ内で SQL 文が実行される場合、ブロックが必要とする多数のコンテキストのスイッチングによってパフォーマンスが低下することがあります。コレクションには次のものが含まれます。

- VARRAY
- ネストした表
- Index-by table
- ホスト配列

バインドとは、SQL 文内の PL/SQL 変数に対して値を代入することです。バルク・バインドとは、コレクション全体を一度にバインドすることです。バルク・バインドは 1 つの操作でコレクション全体を 2 つのエンジン間で受け渡すことができます。

通常、バルク・バインドの使用によって、4 つ以上のデータベース行に影響する SQL 文のパフォーマンスが改善されます。SQL 文によって影響される行数が多いほど、バルク・バインドによるパフォーマンスの向上率は高くなります。

注意： この項では、PL/SQL アプリケーション内でバルク・バインドを使用するかどうかを判断するために役立つ、バルク・バインドの概要を説明します。バルク・バインドを操作する場合の例外の処理方法を含むバルク・バインドの使用方法については、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

バルク・バインドを使用する場合

アプリケーションで次のような使用例がある場合、パフォーマンスを向上させるために、バルク・バインドの使用を検討します。

コレクションを参照する DML 文 FORALL キーワードによって、コレクション文を参照する INSERT 文、UPDATE 文または DELETE 文のパフォーマンスを改善できます。

たとえば、次の PL/SQL ブロックは、バルク・バインドを使用して管理者の ID 番号が 7902、7698 または 7839 の従業員の給料を増額します。

```
DECLARE
   TYPE Numlist IS VARRAY (100) OF NUMBER;
   Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN

-- Efficient method, using a bulk bind
   FORALL i IN Id.FIRST..Id.LAST   -- bulk-bind the VARRAY
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);

-- Slower method, running the UPDATE statements within a regular loop
   FOR i IN Id.FIRST..Id.LAST LOOP
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);
   END LOOP;
END;
```

PL/SQL がバルク・バインドを使用しないで、各従業員を更新するために SQL エンジンに SQL 文を送信すると、コンテキストのスイッチングによってパフォーマンスが低下します。

PL/SQL 表に用意された一連の行がある場合、次のようなループを使用して、データをバルク挿入またはバルク更新できます。

```
FORALL i in Emp_Data.FIRST..Emp_Data.LAST
    INSERT INTO Emp_tab VALUES(Emp_Data(i));
```

コレクションを参照する SELECT 文 BULK COLLECT INTO 句によって、コレクションを参照する問合せのパフォーマンスを改善できます。

たとえば、次の PL/SQL ブロックは、バルク・バインドを使用して複数の値を PL/SQL 表に問い合せます。

```
-- Find all employees whose manager's ID number is 7698.
DECLARE
   TYPE Var_tab IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;
   Empno VAR_TAB;
   Ename VAR_TAB;
```

```
   Counter NUMBER;
   CURSOR C IS
      SELECT Empno, Ename FROM Emp_tab WHERE Mgr = 7698;
BEGIN

-- Efficient method, using a bulk bind
    SELECT Empno, Ename BULK COLLECT INTO Empno, Ename
        FROM Emp_Tab WHERE Mgr = 7698;

-- Slower method, assigning each collection element within a loop.

   counter := 1;
   FOR rec IN C LOOP
      Empno(Counter) := rec.Empno;
      Ename(Counter) := rec.Ename;
      Counter := Counter + 1;
   END LOOP;
END;
```

スカラー値表または %TYPE 値表に BULK COLLECT INTO を使用できます。

PL/SQL がバルク・バインドを使用しないで、選択された各従業員に対して SQL エンジンに SQL 文を送信すると、コンテキストのスイッチングによってパフォーマンスが低下します。

コレクションおよび RETURNING INTO 句を参照する FOR ループ BULK COLLECT INTO キーワードとともに FORALL キーワードを使用すると、コレクションを参照し DML を戻す FOR ループのパフォーマンスを改善できます。

たとえば、次の PL/SQL ブロックは、従業員コレクションの賞与を計算して EMP_TAB 表を更新し、次に Bonlist という列に賞与を戻します。この操作をバルク・バインドを使用して行います。

```
DECLARE
   TYPE Emplist IS VARRAY(100) OF NUMBER;
   Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
   TYPE Bonlist IS TABLE OF Emp_tab.sal%TYPE;
   Bonlist_inst BONLIST;
BEGIN
   Bonlist_inst := BONLIST(1,2,3,4,5);

   FORALL i IN Empids.FIRST..empIDs.LAST
      UPDATE Emp_tab SET Bonus = 0.1 * Sal
      WHERE Empno = Empids(i)
      RETURNING Sal BULK COLLECT INTO Bonlist;

   FOR i IN Empids.FIRST..Empids.LAST LOOP
```

```
      UPDATE Emp_tab Set Bonus = 0.1 * sal
         WHERE Empno = Empids(i)
       RETURNING Sal INTO BONLIST(i);
   END LOOP;
END;
```

PL/SQL がバルク・バインドを使用しないで、各従業員を更新するために SQL エンジンに SQL 文を送信すると、コンテキストのスイッチングによってパフォーマンスが低下します。

トリガー

トリガーは、特殊な種類の無名 PL/SQL ブロックです。文レベルで、または影響を受ける各行に対して、SQL 文の前後で起動するようにトリガーを定義できます。INSTEAD OF トリガーまたはシステム・トリガー（DATABASE または SCHEMA に対するトリガー）も定義できます。

参照： 第 15 章「トリガーの使用」を参照してください。

PL/SQL ラッパーによる PL/SQL コードの隠蔽

PL/SQL ラッパーを使用して、ストアド・プロシージャをオブジェクト・コード形式で引き渡すことができます。PL/SQL コードをラップすると、アプリケーションの内部は隠されます。PL/SQL ラッパーを実行するには、次の構文を使用して、システム・プロンプトから WRAP 文を入力します。

```
wrap INAME=input_file [ONAME=output_file]
```

参照： PL/SQL ラッパーの使用方法の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

ネイティブ実行のための PL/SQL プロシージャのコンパイル

共有ライブラリにあるネイティブ・コードに PL/SQL プロシージャをコンパイルすると、PL/SQL プロシージャの処理を高速化できます。プロシージャは C コードに変換され、通常の C コンパイラでコンパイルされた後、Oracle プロセスにリンクされます。

Oracle の PL/SQL パッケージおよびユーザーが作成したプロシージャの両方に、この方法を使用できます。ALTER SYSTEM または ALTER SESSION コマンドを使用するか、あるいは初期化ファイルを更新して、PLSQL_COMPILER_FLAGS パラメータを NATIVE に設定できます。デフォルトの設定は INTERPRETED です。

この方法では、これらのプロシージャからコールされた SQL 文の処理は高速化されないため、SQL 実行にあまり時間を必要としない計算集中型プロシージャに最も有効です。

Java の場合は、ncomp ツールを使用して独自のパッケージおよびクラスをコンパイルできます。

参照： PL/SQL のネイティブ・コンパイルの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』のチューニングに関する章を参照してください。

JAVA ネイティブ・コンパイルの詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

リモート依存性

PL/SQL プログラム・ユニット間の依存性は、次の 2 つの方法で処理できます。

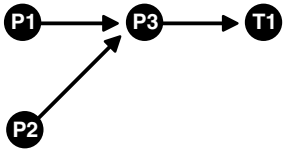
- タイムスタンプ
- シグネチャ

タイムスタンプ

PL/SQL プログラム・ユニット間の依存性を処理するためにタイムスタンプを使用する場合は、プログラム・ユニットまたは関連するスキーマ・オブジェクトを変更するたびに、すべての依存ユニットに無効のマークが付けられるため、再コンパイルしないかぎり実行できません。

各プログラム・ユニットは、そのユニットが作成または再コンパイルされるときに、サーバーによってタイムスタンプが設定されます。図 9-1 に、この依存性について示します。プロシージャ P1 および P2 は、ストアド・プロシージャ P3 をコールします。ストアド・プロシージャ P3 は T1 表を参照します。この例では、各プロシージャは、いずれも T1 表に依存しています。P3 は T1 に直接依存しますが、P1 と P2 は間接的に依存します。

図 9-1 依存性の関係



P1 および P2 が P3 と同じサーバー上にある場合に、P3 が変更されると、P1 および P2 に無効のマークがすぐに付けられます。P1 および P2 がコンパイルされた状態では、P3 のタイムスタンプのレコードが含まれています。そのため、プロシージャ P3 が変更され再コンパイルされた場合、P3 のタイムスタンプは、P1 および P2 のコンパイル中に P3 に対して記録された値と一致しなくなります。

P1 および P2 がクライアント・システム上にある場合、または分散環境内の別の Oracle サーバー上にある場合、実行時に、タイムスタンプ情報を使用して、この 2 つのプロシージャに無効のマークが付けられます。

タイムスタンプ・モデルのデメリット

この依存性モデルのデメリットは、必要以上に制限が多いということです。ネットワークを介した依存オブジェクトは、必ずしも必要でないときにも再コンパイルされることが多いため、パフォーマンスが低下します。

さらに、クライアント側のアプリケーションが PL/SQL バージョン 2 を使用して作成されている場合には、クライアント側では、タイムスタンプ・モデルによってアプリケーションが実行しない状態になる可能性があります。クライアント側で PL/SQL バージョン 1 を使用していた Oracle Forms などの初期のリリースのツールでは、この依存性モデルを使用していませんでした。PL/SQL バージョン 1 がストアド・プロシージャをサポートしていなかったためです。

クライアント側の PL/SQL バージョン 2 と統合された Oracle Forms のリリースの場合、タイムスタンプ・モデルで問題が発生する可能性があります。たとえば、そのアプリケーションが使用するクライアント側の PL/SQL プロシージャがクライアント側で再コンパイルされないかぎり、アプリケーションはインストール中に無効にされます。また、クライアント側のプロシージャがサーバー側のプロシージャに依存しており、そのサーバー側のプロシージャが変更または自動的に再コンパイルされた場合には、クライアント側の PL/SQL プロシージャも再コンパイルする必要があります。ただし、多くのアプリケーション環境（たとえば、Forms ランタイム・アプリケーション）では、クライアントで使用できる PL/SQL コンパイラはありません。このため、アプリケーションは実行できません。このような場合、クライアント・アプリケーションの開発者は、すべての顧客に対して、そのアプリケーションの新しいバージョンを再度配布する必要があります。

シグネチャ

タイムスタンプのみの依存性モデルに関する問題のいくつかを軽減するために、Oracle ではシグネチャを使用したリモート依存性という追加機能を提供します。シグネチャ機能は、リモート依存性のみに影響します。ローカル（同一サーバー）依存性には影響しません。この環境では、再コンパイルが常に可能なためです。

シグネチャは、コンパイル済の各ストアド・プログラム・ユニットと対応付けられます。シグネチャは、次の基準でユニットを識別します。

- ユニットの名前（パッケージ、プロシージャまたはファンクションの名前）
- サブプログラムの各パラメータの型
- パラメータのモード（IN、OUT、IN OUT）
- パラメータの数
- ファンクションの戻り値の型

ユーザーは、シグネチャまたはタイムスタンプがリモート依存性を管理するかどうかを制御できます。

参照： 9-27 ページの「リモート依存性の制御」を参照してください。

シグネチャ依存性モデルが使用されるとき、その依存ユニットに親ユニット内のサブプログラムへのコールが含まれており、このサブプログラムのシグネチャの変更により非互換が発生した場合は、リモート・プログラム・ユニットへの依存性によって、その依存ユニットは無効になります。

たとえば、ボストンにあるサーバー（BOSTON_SERVER）に格納されているプロシージャ Get_emp_name について検討してみます。このプロシージャは、次のように定義されています。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
CREATE PUBLIC DATABASE LINK boston_server USING 'inst1_alias';
CONNECT scott/tiger
```

```
CREATE OR REPLACE PROCEDURE get_emp_name (
   emp_number   IN  NUMBER,
   hire_date    OUT VARCHAR2,
   emp_name     OUT VARCHAR2) AS
BEGIN
   SELECT ename, to_char(hiredate, 'DD-MON-YY')
      INTO emp_name, hire_date
      FROM emp
      WHERE empno = emp_number;
END;
```

Get_emp_name が BOSTON_SERVER でコンパイルされると、そのシグネチャは、そのタイムスタンプとともに記録されます。

ここで、カリフォルニアにある別のサーバーで、PL/SQL コードが BOSTON_SERVER という DB リンクを使用して get_emp_name を識別し、次のように get_emp_name をコールするとします。

```
CREATE OR REPLACE PROCEDURE print_ename (emp_number IN NUMBER) AS
   hire_date    VARCHAR2(12);
   ename        VARCHAR2(10);
BEGIN
   get_emp_name@BOSTON_SERVER(emp_number, hire_date, ename);
   dbms_output.put_line(ename);
   dbms_output.put_line(hire_date);
END;
```

このカリフォルニアのサーバー・コードがコンパイルされるときに、次の処理が行われます。

- ボストンにあるサーバーに接続されます。
- get_emp_name のシグネチャがカリフォルニアのサーバーに転送されます。
- シグネチャは、print_ename のコンパイルされた状態で記録されます。

実行時には、変更の有無にかかわらず、カリフォルニアのサーバーからボストンのサーバーへのリモート・プロシージャのコール中に、print_ename のコンパイルされた状態で保存されていた get_emp_name の記録済のシグネチャが、ボストンのサーバーまで送信されます。

タイムスタンプ依存性モードが有効である場合、タイムスタンプの不一致によって、コール側プロシージャにエラー状況が戻されます。

ただし、シグネチャ・モードが有効である場合は、タイムスタンプに不一致があっても無視され、カリフォルニアのサーバーの Print_ename のコンパイル済状態にある get_emp_name の記録済のシグネチャが、ボストンのサーバーにある get_emp_name の現行のシグネチャと比較されます。2 つのシグネチャが一致した場合は、コールは正常に進行します。2 つのシグネチャが一致しない場合は、エラー・ステータスが print_name プロシージャに戻されます。

ボストンのサーバーの get_emp_name プロシージャは、変更されている可能性もあることに注意してください。または、そのタイムスタンプは、サーバーが新しいリリースでインストールされたなどの理由で、カリフォルニアのサーバーの print_name プロシージャに記録されたタイムスタンプと異なっている場合があります。シグネチャ・リモート依存性モードがカリフォルニアのサーバーで有効であるかぎり、get_emp_name がコールされたときにタイムスタンプの不一致が原因でエラーが発生することはありません。

注意： DETERMINISTIC、PARALLEL_ENABLE および純粋度情報は、シグネチャ・モードでは使用されません。リモート・システム上のファンクションが別の設定で再定義された場合、これらの設定に基づく最適化は、自動的には再考慮されません。したがって、SQL 文でこのリモート・ファンクションへのコールが（間接的にでも）発生した場合、またはファンクション索引でリモート・ファンクションが（間接的にでも）使用された場合に、問合せ結果は正しくないことがあります。

シグネチャが変更される時点

データ型クラスの切替え

シグネチャは、あるクラスのデータ型を別のクラスに切り替えた場合に変更されます。各データ型クラスには、複数の型が存在している可能性があります。1 つのクラス内でパラメータのデータ型を別の型に変更しても、シグネチャが変更されることはありません。次の

表に示された NCHAR や TIMESTAMP などのデータ型は、どのクラスの一部でもありません。その型を変更すると、必ずシグネチャの不一致が発生します。

VARCHAR 型 VARCHAR2、VARCHAR、STRING、LONG、ROWID

文字型 CHARACTER、CHAR

RAW 型 RAW、LONG RAW

整数型 BINARY_INTEGER、PLS_INTEGER、BOOLEAN、NATURAL、POSITIVE、POSITIVEN、NATURALN

数値型 NUMBER、INTEGER、INT、SMALLINT、DECIMAL、DEC、REAL、FLOAT、NUMERIC、DOUBLE PRECISION、DOUBLE PRECISION、NUMERIC

日付型 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL YEAR TO MONTH、INTERVAL DAY TO SECOND

モード デフォルトのパラメータ・モード IN の明示的指定に変更があっても、サブプログラムのシグネチャは変更されません。たとえば、次の 2 つの間で変更すると、シグネチャは変更されません。

```
PROCEDURE P1 (Param1 NUMBER);
PROCEDURE P1 (Param1 IN NUMBER);
```

これ以外のパラメータ・モードの変更が行われた場合、シグネチャは変更されます。

デフォルトのパラメータ値 デフォルトのパラメータ値の指定を変更しても、シグネチャは変更されません。たとえば、プロシージャ P1 は、次の 2 つの例では同じシグネチャを持っています。

```
PROCEDURE P1 (Param1 IN NUMBER := 100);
PROCEDURE P1 (Param1 IN NUMBER := 200);
```

コール側で新しいデフォルト値を取得できるようにする必要がある場合、アプリケーション開発者は、コールされたプロシージャを再コンパイルする必要がありますが、デフォルトのパラメータ値の割当てが変更されても、シグネチャに基づいて無効にされることはありません。

プロシージャ・シグネチャの変更例

9-5 ページの「プロシージャおよびファンクションのパラメータ」に定義されている Get_emp_names プロシージャを使用します。このプロシージャ本体が次のように変更されるとします。

```
DECLARE
   Emp_number  NUMBER;
   Hire_date   DATE;
BEGIN
```

```
-- date format model changes

   SELECT Ename, To_char(Hiredate, 'DD/MON/YYYY')
      INTO Emp_name, Hire_date
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

プロシージャ仕様部は変更されていないため、そのシグネチャも変更されません。

ただし、プロシージャ仕様部が次のように変更されたとします。

```
CREATE OR REPLACE PROCEDURE Get_emp_name (
   Emp_number  IN  NUMBER,
   Hire_date   OUT DATE,
   Emp_name    OUT VARCHAR2) AS
```

それに応じて本体が変更された場合、シグネチャは変更されます。これは、パラメータ Hire_date が別のデータ型になっているためです。

ただし、そのパラメータが When_hired に変更され、データ型は VARCHAR2、モードは OUT のままの場合、シグネチャは変更されません。仮パラメータの名前が変更されても、そのユニットのシグネチャは変更されません。

次の例について検討してみます。

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
END;

CREATE OR REPLACE PACKAGE BODY Emp_package AS
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type) IS
   BEGIN
       SELECT Empno, Ename, TO_CHAR(Hiredate, 'DD/MON/YY')
           INTO Emp_data
           FROM Emp_tab
           WHERE Empno = Emp_data.Emp_number;
   END;
END;
```

レコードのフィールド名が変更されるようにパッケージ仕様部が変更された場合、型がそのままである場合はシグネチャには影響しません。たとえば、次のパッケージ仕様部は、前のパッケージ仕様部の例と同じシグネチャを持っています。

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_num    NUMBER,         -- was Emp_number
        Hire_dat   VARCHAR2(12),   -- was Hire_date
        Empname    VARCHAR2(10));  -- was Emp_name
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
END;
```

型が以前のままである場合には、パラメータの型の名前を変更してもシグネチャは変更されません。たとえば、次のような Emp_package のパッケージ仕様部は、最初のものと同じです。

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_record_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_record_type);
END;
```

リモート依存性の制御

タイムスタンプまたはシグネチャの依存性モデルが有効であるかどうかは、REMOTE_DEPENDENCIES_MODE 動的初期化パラメータによって制御されます。

- 初期化パラメータ・ファイルに、次の指定が含まれているとします。

  ```
  REMOTE_DEPENDENCIES_MODE = TIMESTAMP
  ```

  この場合は、タイムスタンプのみを使用して依存性が解決されます（動的に明示的にオーバーライドされない場合）。

- 初期化パラメータ・ファイルに、次のパラメータ指定が含まれているとします。

  ```
  REMOTE_DEPENDENCIES_MODE = SIGNATURE
  ```

  この場合は、シグネチャを使用して依存性が解決されます（動的に明示的にオーバーライドされない場合）。

- モードは、DDL 文を使用することによって動的に変更できます。たとえば、この例では、カレント・セッションの依存性モデルが変更されます。

  ```
  ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}


  Thise example alters the dependency model systemwide after startup:
  ```

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =
    {SIGNATURE | TIMESTAMP}
```

REMOTE_DEPENDENCIES_MODE パラメータが、init.ora パラメータ・ファイル内で指定されていないか、DDL 文 ALTER SESSION または ALTER SYSTEM を使用して指定されていない場合には、タイムスタンプがデフォルト値です。したがって、明示的に REMOTE_DEPENDENCIES_MODE パラメータまたは適切な DDL 文を使用しないかぎり、使用中のサーバーは、タイムスタンプ依存性モデルを使用して動作します。

REMOTE_DEPENDENCIES_MODE=SIGNATURE を使用する場合は、次の点に注意してください。

- リモート・プロシージャのパラメータのデフォルト値を変更しても、リモート・プロシージャをコールするローカル・プロシージャは無効にされません。リモート・プロシージャへのコールでパラメータが指定されない場合、デフォルト値が使用されます。この場合、無効化 / 再コンパイルは自動的に実行されないため、古いデフォルト値が使用されます。新しいデフォルト値に置き換える場合は、コール側プロシージャを手動で再コンパイルする必要があります。
- オーバーロードされた新しいプロシージャ（既存のものと同じ名前を持つ新しいプロシージャ）をパッケージに追加しても、リモート・プロシージャをコールするローカル・プロシージャは無効にされません。このオーバーロードの結果として、ローカル・プロシージャからの既存のコールがタイムスタンプ・モードで再バインドされた場合、ローカル・プロシージャは無効にされないため、シグネチャ・モードではこの再バインドは行われません。新しい再バインドを成功させるには、ローカル・プロシージャを手動で再コンパイルする必要があります。
- 新しい型が古い型と同じになるように既存のパッケージ・プロシージャのパラメータの型が変更された場合は、ローカルのコール側プロシージャは自動的には無効化または再コンパイルされません。新しい型のセマンティクスに置き換えるためには、コール側プロシージャを手動で再コンパイルする必要があります。

依存性の解決

REMOTE_DEPENDENCIES_MODE = TIMESTAMP（デフォルト値）の場合、プログラム・ユニット間の依存性は、実行時にタイムスタンプを比較して処理されます。コールされたリモート・プロシージャのタイムスタンプが、コールされたプロシージャのタイムスタンプと一致しない場合、コール側の（依存）ユニットは無効になり、再コンパイルする必要があります。この場合、ローカル PL/SQL コンパイラがない場合は、コール側のアプリケーションを処理できません。

タイムスタンプ依存性モードでは、シグネチャは比較されません。ローカル PL/SQL コンパイラがある場合には、コール側プロシージャが実行されると自動的に再コンパイルされます。

REMOTE_DEPENDENCIES_MODE = SIGNATURE の場合、コール側のユニット内に記録されたタイムスタンプは、まず最初に、コールされたリモート・ユニット内の現在のタイムスタンプと比較されます。2 つのタイムスタンプが一致した場合は、コールが進行します。タイム

スタンプが一致しなかった場合、コールされたリモート・サブプログラムのシグネチャは、コール側サブプログラムに記録されたままの状態で、コールされたサブプログラムの現行のシグネチャと比較されます。この 2 つのシグネチャが一致しない場合は、9-24 ページの「シグネチャが変更される時点」の項で説明されている基準を使用した結果、コール側セッションにエラーが戻されます。

依存性を管理するための提案

Oracle では、REMOTE_DEPENDENCIES_MODE パラメータを設定するために次のガイドラインに従うことをお薦めします。

- サーバー側の PL/SQL ユーザーは、このパラメータを TIMESTAMP に設定して（または TIMESTAMP をデフォルト値にして）、タイムスタンプ依存性モードにできます。
- サーバー側の PL/SQL ユーザーは、分散システムを使用している場合には不要な再コンパイルを回避するために、シグネチャ依存性モードの使用を選択できます。
- クライアント側の PL/SQL ユーザーは、このパラメータを SIGNATURE に設定する必要があります。このように設定すると、次のことができます。
  - プロシージャを再コンパイルしなくても、クライアント側のサイトで新しいアプリケーションをインストールできます。
  - タイムスタンプの不一致を発生させずに、サーバーをアップグレードできます。
- サーバー側でシグネチャ・モードを使用するときには、パッケージ仕様部内のプロシージャ（またはファンクション）宣言の終わりに新しいプロシージャを追加します。新しいプロシージャを宣言リストの中間に追加すると、依存プロシージャが不必要に無効にされ、再コンパイルされる可能性があります。

カーソル変数

カーソルは静的オブジェクトであり、カーソル変数はカーソルへのポインタです。そのため、プロシージャおよびファンクションとの間でパラメータとして受け渡すことができます。カーソル変数は、その存続期間内に別のカーソルを参照することもできます。

カーソル変数には、前述以外に次のような利点もあります。

- カプセル化 カーソル変数をオープンするストアド・プロシージャに問合せを集中化できます。
- メンテナンスの容易性 カーソルの変更が必要な場合は、1 つの場所、つまりストアド・プロシージャの変更のみで済みます。個々のアプリケーションを変更する必要はありません。
- セキュリティの利便性 アプリケーションのユーザーは、アプリケーションがサーバーに接続したときに使用したユーザー名です。ユーザーには、カーソルをオープンするストアド・プロシージャに対する EXECUTE 権限が必要です。ただし、ユーザーには、問合

せで使用される表に対する READ 権限は必要ありません。この機能は、表の列へのアクセスおよび他のストアド・プロシージャへのアクセスを制限するために使用できます。

> **参照：** カーソル変数の詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

カーソル変数の宣言およびオープン

メモリーは、通常、適切な ALLOCATE 文を使用してクライアント・アプリケーションのカーソル変数に割り当てられます。Pro*C では、EXEC SQL ALLOCATE <cursor_name> 文を使用します。OCI では、カーソル・データ域を使用します。

また、1 つのサーバー・セッションのみで実行するアプリケーションでも、カーソル変数を使用できます。PL/SQL サブプログラムでカーソル変数を宣言してオープンし、他の PL/SQL サブプログラムのパラメータとして使用できます。

カーソル変数の例

この項には、PL/SQL でのカーソル変数の使用例がいくつか示されています。プログラム・インタフェースを使用するカーソル変数の例がさらに必要な場合は、次のマニュアルを参照してください。

- 『Pro*C/C++ Precompiler プログラマーズ・ガイド』
- 『Pro*COBOL Precompiler プログラマーズ・ガイド』
- 『Oracle Call Interface プログラマーズ・ガイド』
- 『SQL*Module for Ada Programmer's Guide』

データのフェッチ

次のパッケージは、PL/SQL カーソル変数型 Emp_val_cv_type および 2 つのプロシージャを定義しています。最初のプロシージャ（Open_emp_cv）は、WHERE 句にバインド変数を使用してカーソル変数をオープンします。2 番目のプロシージャ（Fetch_emp_data）は、カーソル変数を使用して Emp_tab 表から行をフェッチします。

```
CREATE OR REPLACE PACKAGE Emp_data AS
  TYPE Emp_val_cv_type IS REF CURSOR RETURN Emp_tab%ROWTYPE;
  PROCEDURE Open_emp_cv (Emp_cv          IN OUT Emp_val_cv_type,
                         Dept_number     IN     INTEGER);
  PROCEDURE Fetch_emp_data (emp_cv       IN     Emp_val_cv_type,
                            emp_row      OUT    Emp_tab%ROWTYPE);
END Emp_data;

CREATE OR REPLACE PACKAGE BODY Emp_data AS
  PROCEDURE Open_emp_cv (Emp_cv      IN OUT Emp_val_cv_type,
                         Dept_number IN     INTEGER) IS
  BEGIN
```

```
    OPEN emp_cv FOR SELECT * FROM Emp_tab WHERE deptno = dept_number;
  END open_emp_cv;
  PROCEDURE Fetch_emp_data (Emp_cv      IN  Emp_val_cv_type,
                            Emp_row     OUT Emp_tab%ROWTYPE) IS
  BEGIN
    FETCH Emp_cv INTO Emp_row;
  END Fetch_emp_data;
END Emp_data;
```

次に、PL/SQL ブロックから Emp_data パッケージ・プロシージャをコールする方法を示します。

```
DECLARE
-- declare a cursor variable
   Emp_curs Emp_data.Emp_val_cv_type;
   Dept_number Dept_tab.Deptno%TYPE;
   Emp_row Emp_tab%ROWTYPE;

BEGIN
   Dept_number := 20;
-- open the cursor using a variable
   Emp_data.Open_emp_cv(Emp_curs, Dept_number);
-- fetch the data and display it
   LOOP
     Emp_data.Fetch_emp_data(Emp_curs, Emp_row);
     EXIT WHEN Emp_curs%NOTFOUND;
     DBMS_OUTPUT.PUT(Emp_row.Ename || '  ');
     DBMS_OUTPUT.PUT_LINE(Emp_row.Sal);
   END LOOP;
END;
```

可変レコードの実装

カーソル変数は、異なるカーソルを指す機能にその本質があります。次のパッケージ例では、判別子を使用して、2 つの異なるカーソルのうちの 1 つを指すようにカーソル変数をオープンします。

```
CREATE OR REPLACE PACKAGE Emp_dept_data AS
  TYPE Cv_type IS REF CURSOR;
  PROCEDURE Open_cv (Cv          IN OUT cv_type,
                     Discrim     IN     POSITIVE);
END Emp_dept_data;

CREATE OR REPLACE PACKAGE BODY Emp_dept_data AS
  PROCEDURE Open_cv (Cv      IN OUT cv_type,
                     Discrim IN     POSITIVE) IS
  BEGIN
    IF Discrim = 1 THEN
```

```
      OPEN Cv FOR SELECT * FROM Emp_tab WHERE Sal > 2000;
    ELSIF Discrim = 2 THEN
      OPEN Cv FOR SELECT * FROM Dept_tab;
    END IF;
  END Open_cv;
END Emp_dept_data;
```

Open_cv プロシージャをコールしてカーソル変数をオープンし、Emp_tab 表または Dept_tab 表に関する問合せを指すことができます。次の PL/SQL ブロックは、カーソル変数を使用してフェッチする方法、および ROWTYPE_MISMATCH 事前定義例外を使用して各フェッチ・レコードを処理する方法を示しています。

```
DECLARE
  Emp_rec  Emp_tab%ROWTYPE;
  Dept_rec Dept_tab%ROWTYPE;
  Cv       Emp_dept_data.CV_TYPE;

BEGIN
  Emp_dept_data.open_cv(Cv, 1); -- Open Cv For Emp_tab Fetch
  Fetch cv INTO Dept_rec;       -- but fetch into Dept_tab record
                                -- which raises ROWTYPE_MISMATCH
  DBMS_OUTPUT.PUT(Dept_rec.Deptno);
  DBMS_OUTPUT.PUT_LINE('  ' || Dept_rec.Loc);

EXCEPTION
  WHEN ROWTYPE_MISMATCH THEN
    BEGIN
      DBMS_OUTPUT.PUT_LINE
           ('Row type mismatch, fetching Emp_tab data...');
      FETCH Cv INTO Emp_rec;
      DBMS_OUTPUT.PUT(Emp_rec.Deptno);
      DBMS_OUTPUT.PUT_LINE('  ' || Emp_rec.Ename);
    END;
```

PL/SQL コンパイル時のエラー処理

SQL*Plus を使用して PL/SQL コードを送り、そのコードにエラーがあると、コンパイル・エラーが発生したことが通知されますが、エラーの種類はすぐには識別されません。たとえば、ファイル proc1.sql のスタンドアロン（またはストアド）・プロシージャ PROC1 を次のように送るとします。

```
SQL> @proc1
```

コードに 1 つ以上のエラーが存在すると、次のようなエラー・メッセージが戻されます。

```
MGR-00072: Warning: Procedure proc1 created with compilation errors
```

この場合、SQL*Plus で SHOW ERRORS 文を使用し、検出されたエラーのリストを取得します。引数を持たない SHOW ERRORS は、最後のコンパイルで発生したエラーをリストします。SHOW ERRORS は、プロシージャ、ファンクション、パッケージまたはパッケージ本体の名前を使用して修飾できます。

```
SQL> SHOW ERRORS PROC1
SQL> SHOW ERRORS PROCEDURE PROC1
```

> **参照：** SHOW ERRORS 文の詳細は、『SQL*Plus ユーザーズ・ガイドおよびリファレンス』を参照してください。

> **注意：** 長い行を出力するには、SHOW ERRORS 文を発行する前に SET LINESIZE 文を使用してください。通常、次のように値を 132 に指定することをお薦めします。次に例を示します。
>
> ```
> SET LINESIZE 132
> ```

SQL*Plus を使用して従業員表のレコードを削除する簡単なプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE Fire_emp(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHER Empno = Emp_id;
   END
/
```

CREATE PROCEDURE 文にエラーが 2 つあることに注意してください。まず、DELETE 文にエラーがあります（WHERE に「E」がありません）。また、END の後にセミコロン（;）がありません。

CREATE PROCEDURE 文が入力されエラーが戻されると、SHOW ERRORS 文は次の行を戻します。

```
SHOW ERRORS;

ERRORS FOR PROCEDURE Fire_emp:
LINE/COL       ERROR
-------------- -----------------------------------------
3/27           PL/SQL-00103: Encountered the symbol "EMPNO" wh. . .
5/0            PL/SQL-00103: Encountered the symbol "END" when . . .
2 rows selected.
```

SHOW ERRORS 文によって、エラーが発生した行および列の番号がそれぞれ表示されます。

他のツールまたはアプリケーションを使用している場合は、次のデータ・ディクショナリ・ビューを使用してエラーを表示できます。

- USER_ERRORS
- ALL_ERRORS
- DBA_ERRORS

プロシージャのコンパイルに関するエラー・メッセージは、プロシージャを置き換えると更新され、プロシージャを削除すると削除されます。

ALL_SOURCE、USER_SOURCE、DBA_SOURCE の各ビューを使用すると、データ・ディクショナリから元のソース・コードを取得できます。

> **参照：** これらのデータ・ディクショナリ・ビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

PL/SQL のランタイム・エラー処理

Oracle では、ユーザー定義エラーの番号およびメッセージがクライアント・アプリケーションに戻されるように PL/SQL コード内のユーザー定義エラーを処理できます。クライアント・アプリケーションでは、Oracle が戻したユーザー定義エラーの番号およびメッセージに基づいて、エラーを処理します。

ユーザー定義エラーのメッセージは、RAISE_APPLICATION_ERROR プロシージャを使用して戻されます。次に例を示します。

```
RAISE_APPLICATION_ERROR(Error_number, 'text', Keep_error_stack)
```

このプロシージャはプロシージャの実行を停止し、プロシージャによるすべての影響をロールバックして、ユーザー定義エラー番号およびメッセージを戻します（例外ハンドラによってエラーが検出されない場合）。ERROR_NUMBER は、-20000 ～ -20999 の範囲内にある必要があります。

エラー番号 -20000 は、ユーザーに情報を伝えることが重要で、一意のエラー番号は必要とされないメッセージの一般的な番号として使用します。テキストは、2KB 以下の文字式である必要があります（それより長いメッセージは無視されます）。スタック上の既存のエラーにエラーを追加する場合は Keep_error_stack を TRUE に、既存のエラーと置き換える場合は FALSE にします。デフォルトでは、このオプションは FALSE です。

> **注意：** DBMS_OUTPUT、DBMS_DESCRIBE、DBMS_ALERT など、Oracle 提供のパッケージの中には、-20000 ～ -20005 の範囲のアプリケーション・エラー番号を使用するものがあります。

RAISE_APPLICATION_ERROR プロシージャは、例外ハンドラまたは論理 PL/SQL コードによく使用されます。たとえば、次の例外ハンドラは、ユーザー定義エラー・メッセージに関係する文字列を選択した後、RAISE_APPLICATION_ERROR プロシージャをコールします。

```
...
```

```
WHEN NO_DATA_FOUND THEN
   SELECT Error_string INTO Message
   FROM Error_table,
   V$NLS_PARAMETERS V
   WHERE Error_number = -20101 AND Lang = v.value AND
      v.parameter = "NLS_LANGUAGE";
   Raise_application_error(-20101, Message);
...
```

> **参照：** リモート・プロシージャをコールする場合の例外処理の詳細は、9-37 ページの「リモート・プロシージャでのエラー処理」を参照してください。

次の項では、ユーザー定義エラーの番号をトリガーからプロシージャに渡す例を示します。

例外および例外処理ルーチンの宣言

ユーザー定義例外は、そのアプリケーションに固有のエラーの処理を制御するために PL/SQL ブロック内で明示的に定義され、通知されます。例外が発生する（通知される）と、通常の PL/SQL ブロックの実行は停止し、例外ハンドラと呼ばれるルーチンがコールされます。この例外ハンドラによって内部例外またはユーザー定義例外が処理されます。

アプリケーション・コードを使用すると、IF 文により特に注意が必要な条件をチェックできます。エラー条件がある場合、次の 2 つのオプションのどちらかを選択できます。

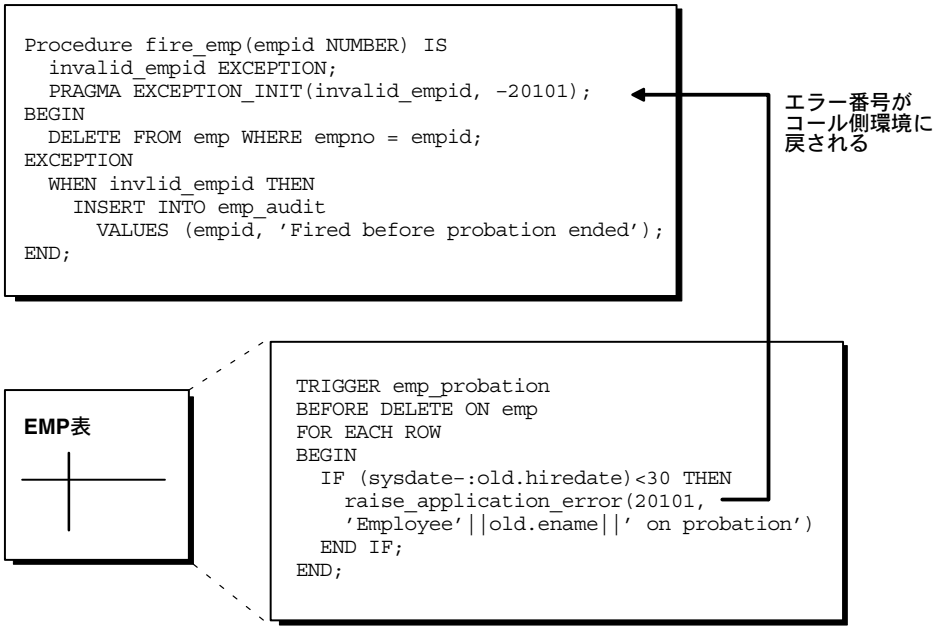
- 適切な例外を指示する RAISE 文を入力します。RAISE 文によってプロシージャの実行は中断され、例外ハンドラがあれば制御が渡されます。
- RAISE_APPLICATION_ERROR プロシージャをコールして、ユーザー定義エラーの番号およびメッセージを戻します。

例外ハンドラは、ユーザー定義エラー・メッセージを処理するために定義することもできます。たとえば、9-36 ページの図 9-2 では、次のことが示されています。

- プロシージャ内の例外およびその例外に対応する例外ハンドラ
- エラー（預金がないのに振込みを行うなど）をチェックし、ユーザー定義エラーの番号およびメッセージをトリガーに入力する条件文
- ユーザー定義エラー番号をコールした環境（この場合はプロシージャ）に戻す方法、およびアプリケーションでユーザー定義エラーの番号に対応する例外を定義する方法

ユーザー定義例外は、プロシージャ本体またはパッケージ本体で宣言するか（プライベート例外）、パッケージ仕様部で宣言します（パブリック例外）。例外ハンドラは、プロシージャ本体（スタンドアロンまたはパッケージ）に定義します。

図 9-2 例外およびユーザー定義エラー



未処理例外

データベースの PL/SQL プログラム・ユニットでは、適切な例外ハンドラによって検出されない未処理のユーザー・エラー条件または内部エラー条件が原因で、プログラム・ユニットの暗黙的なロールバックが発生します。プログラム・ユニットで未処理例外がある場所の前に COMMIT 文が含まれている場合、そのプログラム・ユニットの暗黙的なロールバックは直前の COMMIT まで実行されます。

さらに、データベースに格納された PL/SQL のプログラム・ユニットの未処理例外は、プログラム・ユニットをコールするクライアント側のアプリケーションに渡されます。このアプリケーションでは、その例外はデータベースに SQL 文として送られるため、アプリケーション・プログラム・ユニット・コールのみがロールバックされます（アプリケーション・プログラム・ユニット全体ではありません）。

データベースの PL/SQL プログラム・ユニット内の未処理例外がデータベース・アプリケーションに戻される場合は、例外を処理するためにデータベースの PL/SQL コードを変更する必要があります。アプリケーションで、データベース・プログラム・ユニットをコールしたときに未処理例外を検出し、それらのエラーを処理できます。

分散問合せでのエラー処理

分散問合せは、トリガーまたはストアド・プロシージャを使用して作成できます。この分散問合せは、ローカルの Oracle によって、対応する数のリモート問合せに分解されてリモート・ノードに送られます。リモート・ノードはその問合せを実行し、ローカル・ノードにその結果を送ります。その後、ローカル・ノードは必要な後処理を行い、ユーザーまたはアプリケーションに結果を戻します。

たとえば、整合性制約違反のために分散問合せ文の一部でエラーが発生すると、Oracle はエラー番号 ORA-02055 を戻します。後続する文またはプロシージャ・コールは、ロールバック、またはセーブポイントまでのロールバックが入力されるまで、エラー番号 ORA-02067 を戻します。

分散更新の一部でエラーが発生したことを示すエラー・メッセージがチェックされるように、アプリケーションを設計してください。エラーを検出した場合、アプリケーションが処理を継続する前に、トランザクション全体をロールバック（またはセーブポイントまでロールバック）してください。

リモート・プロシージャでのエラー処理

プロシージャがローカルまたはリモートで実行される場合、次の 4 種類の例外が発生する可能性があります。

- キーワード EXCEPTION を使用した宣言が必要な PL/SQL のユーザー定義例外
- NO_DATA_FOUND などの PL/SQL 事前定義例外
- ORA-00900 や ORA-02015 などの SQL エラー
- RAISE_APPLICATION_ERROR() プロシージャを使用して生成されるアプリケーション例外

ローカル・プロシージャを使用する場合、これらのすべてのメッセージは例外ハンドラを作成することによって検出できます。次に例外ハンドラの例を示します。

```
EXCEPTION
    WHEN ZERO_DIVIDE THEN
    /* ...handle the exception */
```

なお、WHEN 句の例外名は必須です。RAISE_APPLICATION_ERROR で生成される 例外のように、発生した例外に名前がない場合は、プラグマ PRAGMA_EXCEPTION_INIT を使用して名前を割り当てることができます。次に例を示します。

```
DECLARE
    ...
    Null_salary EXCEPTION;
    PRAGMA EXCEPTION_INIT(Null_salary, -20101);
BEGIN
    ...
```

```
    RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
    ...
EXCEPTION
    WHEN Null_salary THEN
        ...
```

また、リモート・プロシージャをコールするときには、ローカル例外ハンドラを作成することによって例外を処理します。リモート・プロシージャは、ローカルのコール側プロシージャにエラー番号を戻す必要があります。その後、ローカル・プロシージャは、先の例に示したように、例外を処理します。PL/SQL のユーザー定義例外は、常にローカル・プロシージャに ORA-06510 を戻すため、これらの例外は処理できません。その他すべてのリモート例外は、ローカル例外と同じ方法で処理できます。

ストアド・プロシージャのデバッグ

ストアド・プロシージャのコンパイルにはコードの構文エラーの修正が含まれます。プロシージャが正常に実行され、エラーが修正されていることを確認するには、追加のデバックを行う必要があります。次のようなデバッグが考えられます。

- 余分な出力文を追加して、実行処理の検証およびプロシージャ内の任意の点のデータ値のチェックをする。
- 別のデバッガを実行して、実行を詳細に分析する。

Oracle Procedure Builder および TEXT_IO パッケージ

Oracle Procedure Builder は、データベース・アプリケーションを透過的にデバッグする高度なクライアント / サーバー・デバッガです。Oracle Procedure Builder を使用すると、制御されたデバッグ環境で PL/SQL プロシージャおよびトリガーを実行し、ブレークポイントの設定、変数の値のリスト、その他のデバッグ作業を実行できます。Oracle Procedure Builder は、Oracle Developer Tool セットの一部です。デバッグ情報を印刷する場合に有効な TEXT_IO パッケージも提供されています。

DBMS_OUTPUT パッケージ

DBMS_OUTPUT パッケージを使用すると、ストアド・プロシージャおよびトリガーもデバッグできます。コードには PUT 文および PUT_LINE 文を入れて、変数および式の値を端末に出力します。

Oracle JDeveloper

JDeveloper の最新のリリースには、PL/SQL、Java およびマルチ言語プログラムをデバッグする新しい機能が含まれています。新しいデバッグ機能は、Java Debug Wire Protocol(JDWP) を利用できます。各種 Oracle 製品の一部として JDeveloper を取得できます。

ストアド・プロシージャをデバッグするために Oracel JDeveloper や他の JDWP ベースのデバッガを使用する前に、データベースとデバッガ間の接続を確立するユーザーに DEBUG ANY PROCEDURE や DEBUG CONNECT SESSION 権限を付与する必要があります。

低レベルのデバッグ・コードの書込み

実際にデバッガの一部のコードを書き込む場合は、DBMS_DEBUG_JDWP や DBMS_DEBUG などのパッケージを使用する必要があります。

DBMS_DEBUG_JDWP パッケージ

Oracle9*i* リリース 2 から提供されている DBMS_DEBUG_JDWP パッケージは、将来 DBMS_DEBUG パッケージと置き換わるマルチ言語デバッグ用のフレームワークを提供します。PL/SQL と Java の組合せのプログラムには特に有効です。

DBMS_DEBUG パッケージ

Oracle8*i* から提供されている DBMS_DEBUG パッケージでは、サーバー側のデバッガが実装されていて、サーバー側の PL/SQL プログラム・ユニットをデバッグする方法を提供します。Oracle Procedure Builder やその他の様々なサード・パーティ・ベンダーが提供するソリューションなど、現在使用可能なデバッガのいくつかでは、この API が使用されています。

参照：

- 『Oracle Procedure Builder Developer's Guide』を参照してください。
- DBMS_DEBUG、DBMS_DEBUG_JDWP および DBMS_OUTPUT パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

ストアド・プロシージャのコール

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Emp_tab (
   Empno     NUMBER(4) NOT NULL,
   Ename     VARCHAR2(10),
   Job       VARCHAR2(9),
   Mgr       NUMBER(4),
   Hiredate  DATE,
   Sal       NUMBER(7,2),
   Comm      NUMBER(7,2),
   Deptno    NUMBER(2));

CREATE OR REPLACE PROCEDURE fire_emp1(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
   END;
VARIABLE Empnum NUMBER;
```

プロシージャは、次のように様々な環境から起動できます。次に例を示します。

- プロシージャは、別のプロシージャまたはトリガーの本体内でコールできます。
- プロシージャは、Oracle のツール製品を使用して、ユーザーが対話形式でコールできます。
- プロシージャは、SQL*Forms やプリコンパイラ・アプリケーションなどのアプリケーション内で明示的にコールできます。
- ストアド・ファンクションは、LENGTH または ROUND などの組込み SQL 関数のコールと同様の方法で SQL 文からコールできます。

この項では、これらの環境からのプロシージャの起動に関して一般的な例をいくつか紹介します。

参照： 9-46 ページの「SQL 式からのストアド・ファンクションのコール」を参照してください。

別のプロシージャをコールするプロシージャまたはトリガー

プロシージャまたはトリガーによって、別のストアド・プロシージャをコールできます。たとえば、プロシージャ本体に次の行を組み込むことができます。

```
. . .
Sal_raise(Emp_id, 200);
. . .
```

この行が Sal_raise プロシージャをコールします。Emp_id は、このプロシージャのコンテキスト内の変数です。PL/SQL 内では再帰プロシージャ・コールが可能なため、プロシージャがプロシージャ自身をコールできます。

Oracle のツール製品からのプロシージャの対話形式コール

プロシージャは、SQL*Plus など、Oracle のツール製品から対話形式でコールできます。たとえば、自分が所有している SAL_RAISE というプロシージャをコールするには、次のように無名 PL/SQL ブロックを使用できます。

```
BEGIN
    Sal_raise(7369, 200);
END;
```

> **注意：** SQL*Plus などの対話形式のツール製品では、PL/SQL ブロックを実行するには、これらの行の終わりにスラッシュ（/）を付けてください。

SQL*Plus の EXECUTE 文を使用すると、ブロックをより簡単に実行できます。これによって、コードに入力する BEGIN 文および END 文がラップされます。次に例を示します。

```
EXECUTE Sal_raise(7369, 200);
```

対話形式のツール製品を使用すると、セッション変数を作成できます。SQL*Plus を使用している場合、次の文によってセッション変数が作成されます。

```
VARIABLE Assigned_empno NUMBER
```

一度定義したセッション変数は、そのセッション中にのみ有効です。たとえば、あるファンクションを実行して、その戻り値をセッション変数に格納できます。

```
EXECUTE :Assigned_empno := Hire_emp('JSMITH', 'President',
   1032, SYSDATE, 5000, NULL, 10);
PRINT Assigned_empno;
ASSIGNED_EMPNO
--------------
          2893
```

> **参照：** SQL*Plus の詳細は、『SQL*Plus ユーザーズ・ガイドおよびリファレンス』を参照してください。開発ツールを使用して同様の操作を実行する詳細は、ご使用のツール製品のドキュメントを参照してください。

3GL アプリケーション内でのプロシージャのコール

プリコンパイラや OCI アプリケーションなどの 3GL データベース・アプリケーションでは、プロシージャへのコールをそのアプリケーションのコード内に記述できます。

アプリケーションの PL/SQL ブロック内のプロシージャを実行するには、単にそのプロシージャをコールします。次の PL/SQL ブロックでは、Fire_emp プロシージャをコールします。

```
Fire_emp1(:Empnun);
```

この場合、:Empno は、アプリケーションのコンテキスト内のホスト（バインド）変数です。

プリコンパイラ・アプリケーションからプロシージャを実行するには、EXEC コール・インタフェースを使用する必要があります。たとえば、次の文は、プリコンパイラ・アプリケーションのコード内で Fire_emp プロシージャをコールします。

```
EXEC SQL EXECUTE
   BEGIN
      Fire_emp1(:Empnum);
   END;
END-EXEC;
```

参照： 3GL アプリケーション内部からの PL/SQL プロシージャのコールの詳細は、次のマニュアルを参照してください。

- 『Oracle Call Interface プログラマーズ・ガイド』
- 『Pro*C/C++ Precompiler プログラマーズ・ガイド』
- 『SQL*Module for Ada Programmer's Guide』

プロシージャ・コール時の名前の変換

プロシージャおよびパッケージに対する参照は、第 2 章「スキーマ・オブジェクトの管理」の「SQL 文における名前解決の規則」で説明されているアルゴリズムに従って変換されます。

プロシージャの実行に必要な権限

スタンドアロン・プロシージャまたはパッケージを所有している場合は、前の項で説明したように、スタンドアロン・プロシージャまたはパッケージ・プロシージャ、あるいはパブリック・プロシージャまたはパッケージ・プロシージャをいつでも実行できます。他のユーザーが所有するスタンドアロン・プロシージャまたはパッケージ・プロシージャを実行するには、次の条件を満たす必要があります。

- プロシージャを含むスタンドアロン・プロシージャまたはパッケージの EXECUTE 権限、または EXECUTE ANY PROCEDURE システム権限が必要です。リモート・プロシージャを実行する場合は、EXECUTE 権限または EXECUTE ANY PROCEDURE システム権限が、ロールを介してではなく直接付与されている必要があります。
- 所有者の名前をコールに指定する必要があります。次に例を示します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT sys/change_on_install AS Sysdba;
CREATE USER Jward IDENTIFIED BY Jward;
GRANT CREATE ANY PACKAGE TO Jward;
GRANT CREATE SESSION TO Jward;
GRANT EXECUTE ANY PROCEDURE TO Jward;
CONNECT Scott/Tiger
```

```
EXECUTE Jward.Fire_emp (1043);

EXECUTE Jward.Hire_fire.Fire_emp (1043);
```

注意： ストアド・サブプログラムまたはパッケージは、そのプロシージャの所有者の権限ドメイン内で実行されます。所有者は、コード本体内で参照されるすべてのオブジェクトに必要なオブジェクト権限を明示的に付与されている必要があります。

プロシージャ引数の値の指定

プロシージャをコールするときには、そのプロシージャの引数に対してそれぞれ値またはパラメータを指定します。引数の値は次のどちらか、または両方を組み合せて指定します。

- プロシージャに宣言されている順序で引数の値をリストします。
- 引数の名前およびその値を、任意の順序で指定します。

たとえば、次の文はどちらも Sal_raise プロシージャをコールして、従業員番号が 7369 の社員の給与を 500 増額します。

```
Sal_raise(7369, 500);

Sal_raise(Sal_incr=>500, Emp_id=>7369);

Sal_raise(7369, Sal_incr=>500);
```

最初の文ではプロシージャ仕様部で宣言されている順序で、引数の値を指定します。

2 番目の文では引数の値を名前で指定し、プロシージャ仕様部で宣言されている順序とは異なる順序で指定します。引数の名前を指定する場合、引数は任意の順序で指定できます。

3 番目の文では、前述の 2 つの方法を組み合せて引数の値を指定します。引数の名前と順序を組み合せて指定する場合は、順序で指定する値が名前で指定する値よりも前にある必要があります。

DEFAULT オプションを使用して、サブプログラムに対する IN パラメータのデフォルト値を定義した場合（『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照）は、異なる数

のパラメータをサブプログラムに渡し、デフォルト値をそのまま使用するか、またはオーバーライドします。実際の値が渡されない場合は、対応するデフォルト値が使用されます。指定を省略する引数（対応するデフォルト値を使用する引数）の後の引数に値を割り当てる場合は、その値のみでなく引数の名前も明示的に指定する必要があります。

リモート・プロシージャのコール

リモート・プロシージャは、適切なデータベース・リンクおよびプロシージャの名前を使用してコールします。次の SQL*Plus 文は、BOSTON_SERVER というローカル・データベース・リンクが示すデータベース内にあるプロシージャ Fire_emp を実行します。

```
EXECUTE fire_emp1@boston_server(1043);
```

> **参照：** リモート・プロシージャをコールする場合の例外処理の詳細は、9-37 ページの「リモート・プロシージャでのエラー処理」を参照してください。

リモート・プロシージャのコールおよびパラメータ値

デフォルト値がある場合でも、すべてのリモート・プロシージャのパラメータに対して値を明示的に渡す必要があります。リモート・パッケージ変数および定数にはアクセスできません。

リモート・オブジェクトの参照

リモート・オブジェクトは、ローカルで定義されているプロシージャ本体で参照できます。次のプロシージャでは、リモートの従業員表の行を削除します。

```
CREATE OR REPLACE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@boston_server WHERE empno = emp_id;
END;
```

リモート・プロシージャをコールする方法を、コール側の環境別に示します。

- リモート・プロシージャ（スタンドアロン・プロシージャおよびパッケージ・プロシージャ）は、そのリモート・プロシージャ名、データベース・リンクおよびリモート・プロシージャの引数を指定することによって、プロシージャ、OCI アプリケーションまたはプリコンパイラ・アプリケーションの中からコールできます。

  ```
  CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
  BEGIN
    fire_emp1@boston_server(arg);
  END;
  ```

- 前述の例では、FIRE_EMP1@BOSTON_SERVER にシノニムを作成できます。これによって、プロシージャ、OCI アプリケーションまたはプリコンパイラ・アプリケーションの

みでなく、SQL*Forms アプリケーションなど、Oracle のツール製品のアプリケーションからリモート・プロシージャをコールできるようになります。

```
CREATE SYNONYM synonym1 for  fire_emp1@boston_server;
CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
BEGIN
  synonym1(arg);
END;
```

- シノニムを使用しない場合は、リモート・プロシージャをコールするローカルのカバー・プロシージャを作成する方法もあります。

```
DECLARE
   arg NUMBER;
BEGIN
   local_procedure(arg);
END;
```

ここで、local_procedure はこのリストの最初の項目と同じ定義です。

参照： 9-46 ページの「プロシージャおよびパッケージのシノニム」を参照してください。

注意： コンパイル時バインディングを使用するストアド・プロシージャとは異なり、リモート・プロシージャを参照する場合は、実行時バインディングが使用されます。接続するユーザー・アカウントは、データベース・リンクに依存します。

リモート・プロシージャのコールは、更新処理を前提とします。このためこのような参照でも、常に 2 フェーズ・コミットのトランザクションが必要です（リモート・プロシージャが読込み専用の場合を含む）。また、リモート・プロシージャを含むトランザクションをロールバックする場合、リモート・プロシージャのコールによって実行された処理も同時にロールバックされます。

リモート・プロシージャでは、ローカル・プロシーシャと同じ COMMIT、ROLLBACK、SAVEPOINT 文を実行できます。ただし、次のように、動作に少し違いがあります。

- トランザクションが Oracle 以外のデータベースによって開始された場合（XA アプリケーションなどの場合）、リモート・プロシージャでこれらの操作はできません。
- これらの操作の 1 つを実行すると、リモート・プロシージャは独自の分散トランザクションを開始できません。
- リモート・プロシージャがその作業をコミットまたはロールバックしない場合、データベース・リンクがクローズすると、コミットが暗黙的に実行されます。このとき、トラ

ンザクションが実行中とみなされるため、リモート・プロシージャをコールできません。

分散更新ではデータは複数のノードで更新されます。異なるノードのデータにアクセスする複数のリモート更新を含むプロシージャを使用できます。構文内の文はリモート・ノードに送信され、構文の実行はユニット単位で正常終了または異常終了します。分散更新の一部がエラーとなり、一部が正常終了した場合、処理を続けるには（トランザクション全体またはセーブポイントまでの）ロールバックが必要です。分散更新を実行するプロシージャを作成する場合は、この点を考慮する必要があります。

リモート・プロシージャのコールにローカル・プロシージャを使用する場合は、特に注意が必要です。ローカル・プロシージャの実行中にタイムスタンプの不一致が見つかると、リモート・プロシージャは実行されず、そのローカル・プロシージャは無効となります。

プロシージャおよびパッケージのシノニム

スタンドアロン・プロシージャおよびパッケージのシノニムは、次の目的で作成します。

- プロシージャまたはパッケージの名前および所有者の名前を非表示にします。
- （スタンドアロンまたはパッケージ内の）リモート・ストアド・プロシージャに対して位置の透過性を提供します。

権限が付与されているユーザーがプロシージャをコールする場合、対応するシノニムを使用できます。パッケージ内に定義されているプロシージャは個々のオブジェクトではないため（パッケージはオブジェクトです）、パッケージ内の個々のプロシージャのシノニムは作成できません。

SQL 式からのストアド・ファンクションのコール

ユーザー作成の PL/SQL ファンクションを SQL 式に組み込むことができます（ただし、PL/SQL リリース 2.1 以上が必要です）。SQL 文で PL/SQL ファンクションを使用すると、次のことができます。

- SQL の拡張によって、ユーザーの生産性が向上します。実行する内容が SQL 文のみで表現するには複雑すぎたり、非常に扱いにくかったり、不可能な場合に、SQL 文の表現機能が強化されます。
- 問合せの効率を向上します。問合せの `WHERE` 句にファンクションを指定すると、条件を使用してデータをフィルタできます。ファンクションを使用できない場合は、アプリケーションで評価する必要があります。
- 特殊なデータ型（緯度、経度、温度など）を表すための文字列を操作できます。
- パラレル問合せの実行を提供できます。問合せがパラレル化されると、PL/SQL ファンクション内の SQL 文も（パラレル問合せオプションを使用して）パラレルに実行できます。

PL/SQL ファンクションの使用

PL/SQL ファンクションは、SQL 文内で使用される前に、トップレベルのファンクションとして作成するか、またはパッケージ仕様部内部で宣言する必要があります。ストアド PL/SQL ファンクションは、組込み Oracle ファンクション（SUBSTR、ABS など）と同じ方法で使用できます。

PL/SQL ファンクションは、SQL 文内で Oracle ファンクションを入れることができる場所、または SQL 内で式を入れることができる場所であればどこにでも入れることができます。たとえば、PL/SQL ファンクションは、次の場所からコールできます。

- SELECT 構文のリスト
- WHERE 句および HAVING 句の条件
- CONNECT BY 句、START WITH 句、ORDER BY 句および GROUP BY 句
- INSERT 文の VALUES 句
- UPDATE 文の SET 句

CREATE 文または ALTER TABLE 文の CHECK 制約句から PL/SQL ストアド・ファンクションをコールしたり、PL/SQL ストアド・ファンクションを使用して列のデフォルト値を指定することはできません。このような状況では、不変の定義が必要になるためです。

注意： 式の一部としてコールされるファンクションとは異なり、プロシージャは文としてコールされます。したがって、PL/SQL プロシージャは、SQL 文からは直接コールできません。ただし、PL/SQL 文からコールされるファンクションまたは SQL 式で参照されるファンクションは、PL/SQL プロシージャをコールできます。

PL/SQL ファンクションをコールする SQL 構文

SQL から PL/SQL ファンクションを参照するには、次の構文を使用します。

```
[[schema.]package.]function_name[@dblink][(param_1...param_n)]
```

たとえば、Scott スキーマ内の My_funcs_pkg パッケージに作成した、2 つの数値パラメータをとる My_func という名前のファンクションを参照するには、次のようにコールします。

```
SELECT Scott.My_funcs_pkg.My_func(10,20) FROM dual;
```

ネーミング規則

オプションのスキーマ名またはパッケージ名のどちらか 1 つが指定されている場合、最初の識別子は、スキーマ名またはパッケージ名のいずれかです。たとえば、リファレンス Payroll.Tax_rate の Payroll がスキーマ名かパッケージ名かを判断するために、Oracle は次の処理を行います。

- Oracle は、まず現行スキーマ内の Payroll パッケージを確認します。
- 現行スキーマ内に PAYROLL パッケージが見つかった場合、Oracle は Payroll パッケージ内で Tax_rate ファンクションを探します。Payroll パッケージ内に Tax_rate ファンクションが見つからない場合は、エラー・メッセージが戻されます。
- Payroll パッケージが見つからない場合、Oracle はトップレベルの Tax_rate ファンクションを含む Payroll というスキーマを探します。Payroll スキーマ内に Tax_rate ファンクションが見つからない場合は、エラー・メッセージが戻されます。

トップレベルのストアド・ファンクションに対して定義したシノニムを使用して、そのファンクションを参照することもできます。

名前の優先順位

SQL 文では、データベースの列の名前は、パラメータなしのファンクションの名前より優先されます。たとえば、スキーマ Scott は、次の 2 つのオブジェクトを作成します。

```
CREATE TABLE Emp_tab(New_sal NUMBER ...);
CREATE FUNCTION New_sal RETURN NUMBER IS ...;
```

その後、次の 2 つの文では、New_sal への参照は、列 Emp_tab.New_sal を参照します。

```
SELECT New_sal FROM Emp_tab;
SELECT Emp_tab.New_sal FROM Emp_tab;
```

new_sal ファンクションにアクセスするには、次のように入力します。

```
SELECT Scott.New_sal FROM Emp_tab;
```

SQL からの PL/SQL ファンクションのコール例 スキーマ Scott から PL/SQL ファンクション Tax_rate をコールし、このファンクションを Tax_table 内の Ss_no 列および sal 列に対して実行し、その結果を変数 Income_tax に入れるには、次のように指定します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Tax_table (
   Ss_no  NUMBER,
   Sal    NUMBER);

CREATE OR REPLACE FUNCTION tax_rate (ssn IN NUMBER, salary IN
NUMBER) RETURN NUMBER IS
   sal_out NUMBER;
   BEGIN
      sal_out := salary * 1.1;
   END;
```

```
DECLARE
   Tax_id     NUMBER;
   Income_tax NUMBER;
BEGIN
   SELECT scott.tax_rate (Ss_no, Sal)
      INTO Income_tax
      FROM Tax_table
      WHERE Ss_no = Tax_id;
END;
```

これらの PL/SQL ファンクションのコール例は、SQL 式で使用できます。

```
Circle_area(Radius)
Payroll.Tax_rate(Empno)
scott.Payroll.Tax_rate@boston_server(Dependents, Empno)
```

引数

任意の数の引数を 1 つのファンクションに渡すには、引数をカッコに入れます。この場合、位置表記法を使用する必要があります。名前表記法は、現在ではサポートされていません。引数の不要なファンクションの場合は、カッコを使用します。

デフォルト値の使用

ストアド・ファンクション Gross_pay では、次のように DEFAULT 句を使用してその仮パラメータの 2 つをデフォルト値に初期化します。次に例を示します。

```
CREATE OR REPLACE FUNCTION Gross_pay
    (Emp_id  IN NUMBER,
```

```
    St_hrs  IN NUMBER DEFAULT 40,
    Ot_hrs  IN NUMBER DEFAULT 0) RETURN NUMBER AS
  ...
```

手続き型の文から Gross_pay をコールする場合は、常に St_hrs のデフォルト値を受け入れることができます。これは、パラメータをスキップする名前表記法を使用できるためです。次に例を示します。

```
IF Gross_pay(Eenum, Ot_hrs => Otime) > Pay_limit
THEN ...
```

ただし、SQL 式から Gross_pay をコールする場合、Ot_hrs のデフォルト値を受け入れないかぎり、St_hrs のデフォルト値を受け入れることはできません。

権限

SQL から PL/SQL ファンクションをコールするには、その所有者であるか、またはそのファンクションに対する EXECUTE 権限が必要です。PL/SQL ファンクションを使用して定義されているビューから選択するには、そのビューに対する SELECT 権限が必要です。そのビューからの選択には、別の EXECUTE 権限は必要ありません。

SQL 式からの PL/SQL ファンクションのコール要件

SQL 式からコールできるようにするには、ユーザー定義の PL/SQL ファンクションが次の基本的な要件を満たす必要があります。

- PL/SQL ブロックまたはサブプログラム内に定義されたファンクションではなく、ストアド・ファンクションである必要があります。
- 列（グループ）ファンクションではなく、行ファンクションである必要があります。つまり、データの列全体をその引数としてとることはできません。
- すべての仮パラメータが IN パラメータである必要があります。どの仮パラメータも、OUT または IN OUT パラメータにすることはできません。
- 仮パラメータのデータ型は、CHAR、DATE、NUMBER などの Oracle サーバーの内部型である必要があります。BOOLEAN、RECORD、TABLE などの PL/SQL 型にはできません。
- 戻り型（結果値のデータ型）が Oracle サーバーの内部型である必要があります。

たとえば、次のストアド・ファンクションは、これらの基本的な要件を満たしています。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Payroll(
                  Srate             NUMBER
                  Orate             NUMBER
                  Acctno            NUMBER);
```

```
CREATE FUNCTION Gross_pay
      (Emp_id IN NUMBER,
       St_hrs IN NUMBER DEFAULT 40,
       Ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
   St_rate  NUMBER;
   Ot_rate  NUMBER;

BEGIN
   SELECT Srate, Orate INTO St_rate, Ot_rate FROM Payroll
      WHERE Acctno = Emp_id;
   RETURN St_hrs * St_rate + Ot_hrs * Ot_rate;
END Gross_pay;
```

副作用の制御

ストアド・ファンクションの純粋度とは、データベース表またはパッケージ変数に対してそのファンクションが及ぼす副作用のことをいいます。副作用によって、問合せのパラレル処理が妨害されたり、処理順序に依存する（したがって、不確定な）結果が発生したり、ユーザー・セッションにまたがったパッケージ状態のメンテナンスが必要になります。ファンクションが SQL 問合せまたは DML 文からコールされる場合は、様々な副作用は受け入れられません。

以前のリリースでは、PL/SQL コンパイラを使用して、ストアド・ファンクションまたは SQL 文のコンパイル中に制限を施行していました。Oracle8*i* からは、コンパイル時の制限は緩和され、実行中の制限も少なくなっています。

参照： 9-52 ページの「制限事項」を参照してください。

この変更によって、PL/SQL、Java および C で作成されたストアド・ファンクションが統一してサポートされ、プログラマには最大限の柔軟性が提供されています。

PL/SQL コンパイル・チェック

コンパイル時に純粋度を確認しなくても、SQL 文からユーザー定義ファンクションをコールできるようになりました。SQL 文からのファンクション・コールには、PRAGMA RESTRICT_REFERENCES が必要がありません。

PRAGMA RESTRICT_REFERENCES は、予期される副作用のみがファンクションに含まれているかどうかを PL/SQL コンパイラに対して尋ねる手段として残されています。宣言済の制限に違反するファンクションに対する SQL 文、パッケージ変数アクセスまたはコールは、引き続き PL/SQL コンパイル・エラーを表示し、予期しない副作用のあるコードの分離に役立ちます。

Oracle では、SQL 文からコールされるファンクションにはプラグマが不要になったため、PRAGMA RESTRICT REFERENCES を使用するかどうか、またどこに使用するかは、アプリケーションごとに任意に選択できます。既存の PL/SQL アプリケーションでは、既存コードとの統合を容易にするために、新しいファンクションに対してもプラグマを引き続き使用することが考えられます。新しく作成される Java アプリケーションでは、プラグマは使用しないものと思われます。これは、Java コンパイラには、予期しない副作用の分離を支援する機能がないためです。

> **参照：** 9-56 ページの「PRAGMA RESTRICT_REFERENCES の使用」を参照してください。

制限事項

SQL 文が実行されるとき、すでに実行中の SQL 文の中にこの SQL 文が論理的に埋め込まれているかどうかが確認されます。文がトリガーまたはすでに実行中の SQL 文からコールされたファンクションから実行されると、この確認が行われます。このような場合は、新しい SQL 文が特定のコンテキスト内で安全かどうかを判断するために、さらに確認が行われます。

次の制限が適用されます。

- 問合せまたは DML 文からコールされるファンクションは、現行のトランザクションの終了、セーブポイントの作成またはセーブポイントまでのロールバック、あるいはシステムまたはセッションの変更（ALTER）を実行できません。
- 問合せ（SELECT）文またはパラレル化された DML 文からコールされるファンクションは、DML 文を実行できません。またはデータベースを変更できません。
- DML 文からコールされるファンクションは、その DML 文が変更中の表の読込みおよび変更はできません。

前述のすべての制限は、ファンクションまたはトリガー内で SQL 文が実行される方法にかかわらず適用されます。次に例を示します。

- 前述の制限は、PL/SQL からコールされる SQL 文（ファンクションまたはトリガーの本体に直接埋め込まれている文かどうかにかかわらず）が、新機能のネイティブ動的メカニズム（EXECUTE IMMEDIATE）を使用して実行されるか、または DBMS_SQL パッケージを使用して実行される場合、その SQL 文に適用されます。
- SQLJ 構文を使用して Java に埋め込まれている文、または JDBC を使用して実行される文に適用されます。
- 外部 C 関数内からコールバック・コンテキストを使用して OCI で実行される文に適用されます。

新しい SQL 文の実行が、すでに実行中の文のコンテキストに論理的に埋め込まれていない場合は、前述の制約を回避できます。PL/SQL の新機能の自律型トランザクションが 1 つの回避方法を提供します。外部 C 関数から OCI を使用するという別の回避方法もあります。この場合は、OCIExtProcContext 引数から使用できるハンドルを使用せず、新しい接続を作成します。

ファンクションの宣言

キーワード DETERMINISTIC および PARALLEL_ENABLE は、ファンクションを宣言する構文内で使用できます。この 2 つのキーワードは最適化ヒントで、問合せオプティマイザおよび他のソフトウェア・コンポーネントのその他の機能に対して、重複してコールする必要のないファンクションまたはパラレル問合せ、あるいは DML 文の中で使用できるファンクションについて情報を提供します。ファンクション索引および特定のスナップショットやマテリアライズド・ビューで使用できるのは、DETERMINISTIC を指定したファンクションのみです。

引数として渡される値のみに依存し、パッケージ変数またはデータベースの内容に意味のある参照または変更を実行しないファンクション、あるいは他の副作用を持たないファンクションを、確定的なファンクションといいます。このようなファンクションは、渡される引数値の組合せが同じであるかぎり、必ず同じ戻り値を生成します。

DETERMINISTIC キーワードは、ファンクションの宣言の中で戻り値の型の後に入れます。次に例を示します。

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN P1 * 2;
END;
```

このキーワードは、CREATE FUNCTION 文で定義されるファンクション内、CREATE PACKAGE 文のファンクションの宣言内、または CREATE TYPE 文のメソッドの宣言内に入れられます。CREATE PACKAGE BODY 文または CREATE TYPE BODY 文のファンクションまたはメソッドの本体では指定しないでください。

DETERMINISTIC とマークされているファンクションに対するコールでは、他にアクションを実行しなくても、パフォーマンスがある程度最適化されます。ただし、ファンクションが本当に確定的かどうかを認識する方法が、データベースにはありません。その動作が本当の意味で確定的ではないファンクションに対して DETERMINISTIC キーワードが適用されると、そのファンクションが関係する問合せの結果は予測できません。

次の 2 つの新機能では、その機能で使用されるすべてのファンクションを DETERMINISTIC として宣言する必要があります。

- ファンクション・ベースの索引で使用されるすべてのファンクションは、DETERMINISTIC である必要があります。
- マテリアライズド・ビューで使用されるファンクションは、そのビューが ENABLE QUERY REWRITE としてマークされる場合は、DETERMINISTIC である必要があります。

この 2 つの機能では、ファンクションをコールするのではなく、事前に計算されている結果をできるだけ使用しようとします。

マテリアライズド・ビューまたは REFRESH FAST として宣言されているスナップショットでは、DETERMINISTIC として宣言されているファンクションのみを使用することをお薦めします。Oracle では、REFRESH FAST スナップショットに、RNDS であることを示す PRAGMA RESTRICT_REFERENCES を持つファンクション、および CREATE FUNCTION 文を使用して定義された PL/SQL ファンクション（コードから、データベースの読込みも、データベースを読み込む可能性のある他のルーチンのコールもしないことを判断できるファンクション）を従来どおり使用できます。

WHERE 句、ORDER BY 句または GROUP BY 句の中で使用されるファンクション、SQL 型の MAP メソッドまたは ORDER メソッドであるファンクション、それ以外では、結果セットに行を入れるか、またはどこに入れるかを決定するファンクションの一部であるファンクションも、前述のように DETERMINISTIC である必要があります。オラクル社では、既存のアプリケーションを壊すことなく、このようなファンクションを明示的に DETERMINISTIC と宣言するように要求することはできませんが、このキーワードを使用することはアプリケーションのスタイルとして賢明な選択といえます。

パラレル問合せおよびパラレル DML

Oracle のパラレル実行機能により、SQL 文の実行作業が複数のプロセスにわたって分割されます。パラレルで実行される SQL 文からコールされるファンクションは、各プロセス内で実行される個別のコピーを持つことができ、それぞれのコピーは、そのプロセスによって処理される行のサブセットのためにのみコールされます。

各プロセスには、そのプロセス専用のパッケージ変数のコピーがあります。パラレル実行が開始されると、コピーされたパッケージ変数は、新しいユーザーがシステムにログインするときのように、パッケージ仕様部および本体の情報に基づいて初期化されます。パッケージ変数内の値は、元のログイン・セッションからはコピーされません。パッケージ変数に対する変更は、多数のセッション間で伝播したり、元のセッションに伝播することはありません。Java の STATIC クラス属性は、同様に、各プロセス内で独立して初期化され変更されます。ファンクションでは、検出される様々な行の値をパッケージ（または Java の STATIC）変数を使用して蓄積できるため、Oracle では、すべてのユーザー定義ファンクションの実行をパラレル化しても安全であるとはいえません。

検索（SELECT）文の場合、以前のリリースでは、ファンクションに対して PRAGMA RESTRICT_REFERENCES 宣言内で RNPS および WNPS として記述されているかどうかをパラレル問合せ最適化機能が調べていました。RNPS および WNPS としてマークされているファンクションは、パラレルで実行できました。CREATE FUNCTION 文を使用して定義されているファンクションでは、ファンクションが実際に十分に純粋であるかどうかを判断するために、明示的にコードが調べられていました。パラレル実行は、これらのファンクションに対してプラグマを指定できない場合でも発生する可能性があります。

> **参照：** 9-56 ページの「PRAGMA RESTRICT_REFERENCES の使用」を参照してください。

DML 文の場合、以前のリリースでは、ファンクションに対して PRAGMA RESTRICT_REFERENCES 宣言内で RNDS、WNDS、RNPS、WNPS の 4 つがすべて記述されているかどうかをパラレル化最適化機能が調べていました。データベースまたはパッケージ変数のいずれかに対して読込みでもなく書込みでもないとマークされたファンクションは、パラレルで実行できました。ここでも、CREATE FUNCTION 文を使用して定義されているファンクションでは、ファンクションが実際に十分に純粋であるかどうかを判断するために、明示的にコードが調べられていました。パラレル実行は、これらのファンクションに対してプラグマを指定できない場合でも発生する可能性があります。

Oracle9*i* でも、以前のリリースでパラレル化可能として認識されていたファンクションは、引き続きパラレル化されます。パラレル実行用として安全であるとマークする方法として、PARALLEL_ENABLE キーワードをお薦めします。このキーワードは、前述の DETERMINISTIC と構文的に類似しています。このキーワードは、次に示すように、ファンクション宣言の戻り値型の後に入れます。

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN P1 * 2;
END;
```

このキーワードは、CREATE FUNCTION 文で定義されるファンクション内、CREATE PACKAGE 文のファンクションの宣言内、または CREATE TYPE 文のメソッドの宣言内に入れられます。CREATE PACKAGE BODY 文または CREATE TYPE BODY 文のファンクションまたはメソッドの本体では指定しないでください。

CREATE FUNCTION を使用して定義される PL/SQL ファンクションは、そのファンクションがパッケージ変数の読込みも書込みも行わず、パッケージ変数の読込みまたは書込みを行う可能性のあるファンクションもコールしないことをシステムで判断できる場合は、パラレルで実行しても安全であると明示的に宣言しなくても、パラレルで実行できることに注意してください。Java メソッドまたは C 関数は、プログラマがコール仕様に PARALLEL_ENABLE と明示的に指定するか、または PRAGMA RESTRICT_REFERENCES を指定してファンクションが十分に純粋であることを示さないかぎり、システムはパラレルでの実行が安全であるとはみなしません。

パラレル DML 文の一部としてパラレル実行されるファンクションに対しては、追加のランタイム制約が設けられています。このようなファンクションは、DML 文の実行を許可されません。検索（SELECT）文の中で実行されるファンクションに対して適用される制約と同じ制約を受けます。

参照： 9-52 ページの「制限事項」を参照してください。

PRAGMA RESTRICT_REFERENCES の使用

純粋度レベルを宣言するには、PRAGMA RESTRICT_REFERENCES を（パッケージ本体ではなく）パッケージ仕様部にコーディングします。このプラグマは、ファンクション宣言の後に置く必要がありますが、直後に置く必要はありません。所定のファンクション宣言を参照できるプラグマは、1 つのみです。

PRAGMA RESTRICT_REFERENCES をコーディングするには、次の構文を使用します。

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

パラメータは次のとおりです。

キーワード	説明
WNDS	データベース状態を書き込みません（Writes no database state）。データベース表を変更しないということです。
RNDS	データベース状態を読み込みません（Reads no database state）。データベース表を問い合せないということです。
WNPS	パッケージ状態を書き込みません（Writes no package state）。パッケージ変数の値を変更しないということです。
RNPS	パッケージ状態を読み込みません（Reads no package state）。パッケージ変数の値を参照しないということです。
TRUST	RESTRICT_REFERENCES 宣言を持つファンクションからこの宣言を持たないファンクションへのコールが簡単になります。

引数はどんな順序でも渡せます。ファンクション本体にある SQL 文のいずれかが規則に違反する場合は、その文が解析されるときにエラーが発生します。

次の例では、ファンクション compound により、データベースまたはパッケージ状態の読込みおよび書込みは行われないため、最大の純粋度レベルを宣言できます。ファンクションで可能な最高の純粋度レベルを常に保つようにしてください。これによって、PL/SQL コンパイラがファンクションを必要以上に拒否することはなくなります。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Accts (
 Yrs      NUMBER
 Amt      NUMBER
 Acctno   NUMBER
 Rte      NUMBER);
```

```
CREATE PACKAGE Finance AS  -- package specification
   FUNCTION Compound
         (Years  IN NUMBER,
          Amount IN NUMBER,
          Rate   IN NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (Compound, WNDS, WNPS, RNDS, RNPS);
END Finance;

CREATE PACKAGE BODY Finance AS  --package body
   FUNCTION Compound
         (Years  IN NUMBER,
          Amount IN NUMBER,
          Rate   IN NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN Amount * POWER((Rate / 100) + 1, Years);
   END Compound;
                  -- no pragma in package body
END Finance;
```

後で、次のように PL/SQL ブロックから compound をコールできます。

```
DECLARE
   Interest NUMBER;
   Acct_id NUMBER;
BEGIN
   SELECT Finance.Compound(Yrs, Amt, Rte)  -- function call
   INTO   Interest
   FROM   Accounts
   WHERE  Acctno = Acct_id;
```

キーワード TRUST の使用 キーワード TRUST を RESTRICT_REFERENCES 構文で使用すると、RESTRICT_REFERENCES 宣言を持つファンクションからこの宣言を持たないファンクションへのコールが容易になります。TRUST が指定されていると、プラグマにリストされている制限は実際には適用されませんが、真であると判断できます。

プラグマを使用するコード・セクションからプラグマを使用しないコード・セクションをコールするときは、2 種類の使用スタイルがあります。1 つは、コールされるルーチン上にプラグマを入れるスタイルです。たとえば、Java メソッド用のコール仕様上に入れます。こ

れで、PL/SQL からこのメソッドをコールした場合、メソッドの制約がコール側のファンクションの制約より少ない場合、コールでエラーが発生します。次に例を示します。

```
CREATE OR REPLACE PACKAGE P1 IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
      PRAGMA RESTRICT_REFERENCES(F1,WNDS,TRUST);
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;

   PRAGMA RESTRICT_REFERENCES(F2,WNDS);
END;

CREATE OR REPLACE PACKAGE BODY P1 IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

ここで、F1 は WNDS として宣言されているため、F2 が F1 をコールできます。

もう 1 つのスタイルは、コール側のみをマークする方法です。マークされたコール側は、エラーなしで任意のファンクションをコールできます。次に例を示します。

```
CREATE OR REPLACE PACKAGE P1a IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(F2,WNDS,TRUST);
END;

CREATE OR REPLACE PACKAGE BODY P1a IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

ここでは、F2 が F1 をコールできます。これは、F2 は WNDS と指定されています（TRUST が指定されているため）が、F2 の本体が WNDS 制約を本当に満たすかどうかが実際に調べられていないためです。F2 が調べられないため、F1 には PRAGMA RESTRICT_REFERENCES が指定されていないにもかかわらず、F2 から F1 にコールできます。

静的 SQL 文と動的 SQL 文の違い 静的な INSERT、UPDATE および DELETE 文は、表の列などのデータベース状態を明示的に読み込まない場合は、RNDS には違反しません。ただし、動的な INSERT、UPDATE および DELETE 文の場合は、データベース状態を明示的に読み込むかどうかにかかわらず、常に RNDS に違反します。

次の INSERT は、動的に実行される場合は RNDS に違反しますが、静的に実行される場合は RNDS に違反しません。

```
INSERT INTO my_table values(3, 'SCOTT');
```

次の UPDATE は、my_table の列名を明示的に読み込むため、静的に実行された場合も動的に実行された場合も RNDS に違反します。

```
UPDATE my_table SET id=777 WHERE name='SCOTT';
```

パッケージ PL/SQL ファンクションのオーバーロード

PL/SQL では、パッケージ・ファンクション（スタンドアロン以外）のオーバーロードが可能です。仮パラメータの数、順序、データ型ファミリなどが異なっていれば、別のファンクションに対して同じ名前を使用できます。

ただし、RESTRICT_REFERENCES プラグマは、1 つのファンクション宣言にのみ適用できます。したがって、オーバーロードされたファンクションの名前を参照するプラグマは、常に前にある、最も近いファンクション宣言に適用されます。

次の例では、プラグマは、valid の 2 番目の宣言に適用されます。

```
CREATE PACKAGE Tests AS
    FUNCTION Valid (x NUMBER) RETURN CHAR;
    FUNCTION Valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
 END;
```

逐次再利用可能 PL/SQL パッケージ

通常、PL/SQL パッケージでは、パッケージ内のパッケージ変数およびカーソルの数に応じて、ユーザー・グローバル領域（UGA）のメモリーが消費されます。このメモリーはユーザー数に比例して増加するため、スケーラビリティが制限されます。これを解決するには、プラグマ構文を使用して一部のパッケージに SERIALLY_REUSABLE のマークを付けます。

逐次再利用可能パッケージの場合、パッケージのグローバル・メモリーは各ユーザーの UGA ではなく、小さなプールに保持され、複数の異なるユーザーのために再利用されます。これは、このようなパッケージのグローバル・メモリーは、作業単位内でのみ使用されることを意味します。そのため、作業単位の終了時にメモリーはそのプールに解放され、（すべてのグローバル変数の初期化コードの実行後）別のユーザーによって再利用されます。

逐次再利用可能パッケージの作業単位とは、サーバーへの OCI コール、PL/SQL のクライアント・サーバー間 RPC コール、PL/SQL のサーバー間 RPC コールなどのサーバーへのコールです。

パッケージ状態

再利用不可能パッケージ（SERIALLY_REUSABLE のマークが付いていない）の状態は、セッションの存続期間を通じて持続します。パッケージの状態には、グローバル変数やカーソルなどがあります。

逐次再利用可能パッケージの状態は、サーバーへのコールの存続期間中のみ持続します。サーバーへの後続のコールでは、逐次再利用パッケージが参照される場合、逐次再利用可能パッケージの新しいインスタンシエーションが作成され、すべてのグローバル変数を NULL に、または指定したデフォルト値に初期化します。サーバーへの以前のコール内の逐次再利用可能パッケージ状態に対して行われた変更は参照できません。

注意： サーバーへのコール時に逐次再利用可能パッケージの新しいインスタンシエーションが作成されても、Oracle によってメモリーが割り当てられたり、インスタンス化オブジェクトが構成されるとはかぎりません。SGA の前回使用されてから最も時間の経過している（LRU）プールにある、このパッケージで使用可能な（割当ておよび構成済の）インスタンス化作業領域を探します。

サーバーへのコールの終了時に、この作業域は LRU プールに戻されます。SGA 内にプールがあるのは、同じパッケージに対する要求を持つユーザー間で、作業域を再利用できるようにするためです。

逐次再利用可能パッケージを使用する理由

再利用不可能パッケージの状態はセッションの存続期間を通じて持続するので、セッション全体の UGA メモリーがロックされます。Oracle Office などのアプリケーションでは、ログイン・セッションは一般に何日も持続します。アプリケーションでは、特定のパッケージをセッション中の特定のローカル期間のみで使用することが必要な場合がよくあります。また、パッケージの使用後は、セッションの途中でパッケージ状態を非インスタンス化するのが理想的です。

逐次再利用可能パッケージ（SERIALLY_REUSABLE）を使用すると、アプリケーション開発者は、メモリーをより適切に管理してスケーラビリティを向上させるアプリケーションをモデル化できます。サーバーへのコール中にのみ管理されるパッケージ状態は、SERIALLY_REUSABLE パッケージ内に獲得する必要があります。

逐次再利用可能パッケージの構文

パッケージは、PRAGMA SERIALLY_REUSABLE により逐次再利用可能のマークが付けられます。プラグマの構文は、次のとおりです。

```
PRAGMA SERIALLY_REUSABLE;
```

パッケージ仕様部は、対応するパッケージ本体の有無にかかわらず、逐次再利用可能のマークを付けられます。パッケージに本体がある場合、対応する仕様部に逐次再利用可能プラグ

マがあると、本体にもそのプラグマが必要です。逐次再利用可能プラグマは、仕様部にそのプラグマがないかぎり、本体に含めることはできません。

逐次再利用可能パッケージのセマンティック

SERIALLY_REUSABLE のマークが付いたパッケージには、次のプロパティがあります。

- パッケージ変数は、サーバーへのコールに対応した作業境界（OCI コール境界またはサーバーへの PL/SQL RPC コールのいずれか）の中でのみ使用できます。

  注意： アプリケーション・プログラマが、誤って前の作業単位で設定されたパッケージ変数を使用した場合、このアプリケーション・プログラムは、正常に実行されない可能性があります。PL/SQL では、このようなケースはチェックされません。

- パッケージ・インスタンシエーションのプールが保持され、作業単位にこのパッケージが必要な場合は、必ずこのインスタンシエーションの 1 つが次のように再利用されます。
  - パッケージ変数が再度初期化されます（たとえば、パッケージ変数がデフォルト値の場合、これらの変数は再度初期化されます）。
  - このパッケージ本体の初期化コードが、再実行されます。
- 作業終了境界で、クリーンアップが行われます。
  - カーソルがオープンされたままの場合、暗黙的にクローズされます。
  - 再利用不可能なセカンダリ・メモリーの一部が解放されます（コレクション変数や長い VARCHAR2 のメモリーなど）。
  - このパッケージ・インスタンシエーションは、このパッケージ用に保持された再利用可能インスタンシエーションのプールに戻されます。
- トリガー内から逐次再利用可能パッケージへのアクセスはできません。トリガーから逐次再利用可能パッケージにアクセスすると、「トリガーのコンテキストでは逐次再利用可能パッケージ < パッケージ名 > にアクセスできません。」というエラー・メッセージが表示されます。

逐次再利用可能パッケージの例

例 1: コール境界を超えるパッケージ変数の機能　次に、逐次再利用可能パッケージ仕様部の例を示します（本体はありません）。

```
CONNECT Scott/Tiger

CREATE OR REPLACE PACKAGE Sr_pkg IS
```

```
  PRAGMA SERIALLY_REUSABLE;
  N NUMBER := 5;                -- default initialization
END Sr_pkg;
```

Enterprise Manager（または SQL*Plus）アプリケーションから次の文が発行されるとします。

```
CONNECT Scott/Tiger

# first CALL to server
BEGIN
   Sr_pkg.N := 10;
END;

# second CALL to server
BEGIN
   DBMS_OUTPUT.PUT_LINE(Sr_pkg.N);
END;
```

結果が出力されます。

```
5
```

> **注意：** パッケージに PRAGMA SERIALLY_REUSABLE がない場合は、10 が出力されます。

例 2: コール境界を超えるパッケージ変数の機能 次に、逐次再利用可能なパッケージ仕様部およびパッケージ本体の例を示します。

```
CONNECT Scott/Tiger

DROP PACKAGE Sr_pkg;
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   TYPE Str_table_type IS TABLE OF VARCHAR2(200) INDEX BY BINARY_INTEGER;
   Num     NUMBER        := 10;
   Str     VARCHAR2(200) := 'default-init-str';
   Str_tab STR_TABLE_TYPE;

    PROCEDURE Print_pkg;
    PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2);
END Sr_pkg;
CREATE OR REPLACE PACKAGE BODY Sr_pkg IS
   -- the body is required to have the pragma because the
  -- specification of this package has the pragma
  PRAGMA SERIALLY_REUSABLE;
```

```
   PROCEDURE Print_pkg IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('num: ' || Sr_pkg.Num);
      DBMS_OUTPUT.PUT_LINE('str: ' || Sr_pkg.Str);
      DBMS_OUTPUT.PUT_LINE('number of table elems: ' || Sr_pkg.Str_tab.Count);
      FOR i IN 1..Sr_pkg.Str_tab.Count LOOP
         DBMS_OUTPUT.PUT_LINE(Sr_pkg.Str_tab(i));
      END LOOP;
   END;
   PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2) IS
   BEGIN
   -- init the package globals
      Sr_pkg.Num := N;
      Sr_pkg.Str := V;
      FOR i IN 1..n LOOP
         Sr_pkg.Str_tab(i) := V || ' ' || i;
   END LOOP;
   -- now print the package
   Print_pkg;
   END;
 END Sr_pkg;

SET SERVEROUTPUT ON;

Rem SR package access in a CALL:

BEGIN
   -- initialize and print the package
   DBMS_OUTPUT.PUT_LINE('Initing and printing pkg state..');
   Sr_pkg.Init_and_print_pkg(4, 'abracadabra');
   -- print it in the same call to the server.
   -- we should see the initialized values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL...');
   Sr_pkg.Print_pkg;
END;

Initing and printing pkg state..
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL...
num: 4
str: abracadabra
```

```
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4

REM SR package access in subsequent CALL:
BEGIN
   -- print the package in the next call to the server.
   -- We should that the package state is reset to the initial (default) values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the next CALL...');
   Sr_pkg.Print_pkg;
END;
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
```

例 3: コール境界を超える逐次再利用可能パッケージのオープン・カーソル この例は、逐次再利用可能パッケージのすべてのオープン・カーソルが作業境界（コール）の終了時に自動的にクローズされることを示します。また、新しいコールでは、これらのカーソルを再度オープンする必要があります。

```
REM  For serially reusable pkg: At the end work boundaries
REM  (which is currently the OCI call boundary) all open
REM  cursors will be closed.
REM
REM  Because the cursor is closed - every time we fetch we
REM  will start at the first row again.

CONNECT Scott/Tiger
DROP PACKAGE  Sr_pkg;
DROP TABLE People;
CREATE TABLE People (Name VARCHAR2(20));
INSERT INTO  People  VALUES ('ET');
INSERT INTO  People  VALUES ('RAMBO');
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   CURSOR C IS SELECT Name FROM People;
END Sr_pkg;
SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE Fetch_from_cursor IS
Name VARCHAR2(200);
BEGIN
   IF (Sr_pkg.C%ISOPEN) THEN
```

```
      DBMS_OUTPUT.PUT_LINE('cursor is already open.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('cursor is closed; opening now.');
      OPEN Sr_pkg.C;
   END IF;
   -- fetching from cursor.
   FETCH sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   FETCH Sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   -- Oops forgot to close the cursor (Sr_pkg.C).
   -- But, because it is a Serially Reusable pkg's cursor,
   -- it will be closed at the end of this CALL to the server.
END;
EXECUTE fetch_from_cursor;
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
```

ファンクションからの大量のデータの戻し

データ・ウェアハウス環境では、大量のデータを変換するために PL/SQL ファンクションを使用します。データは、異なるファンクションによる一連の変換を経由して渡されます。以前、このような変換は、かなりのメモリー・オーバーヘッドを必要としたり、各変換の間で、データを表に格納する必要がありました。

このような変換を低オーバーヘッドで行うには、PL/SQL 表ファンクションを使用します。これらのファンクションは、複数の行を受け入れて戻すことが可能で、行は、一度ではなく準備できた順に戻すことができます。さらにパラレル化も可能です。

この方法は、次のように行います。

- 生成側の問合せサーバーのファンクションの宣言に、PIPELINED キーワードを使用します。
- 生成側の問合せサーバーのファンクションに、OUT パラメータ（レコード）を使用します。このパラメータは結果セットの行に対応しています。
- 各出力レコードが終了すると、そのレコードは PIPE ROW キーワードを使用するコンシューマ・ファンクションに送信されます。
- 生成側の問合せサーバーのファンクションを、戻り値を指定しない RETURN 文で終了します。
- コンシューマ・ファンクションまたは SQL 文に TABLE キーワードを使用して、通常の表のように結果としての行を処理します。

次に例を示します。

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
-- Function accepts multiple rows through a REF CURSOR argument.
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
-- Return value is a record type that matches the table definition.
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
-- Once a result row is ready, we send it back to the calling program,
-- and continue processing.
    PIPE ROW(out_rec);
-- This function outputs twice as many rows as it receives as input.
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
-- The function ends with a RETURN statement that does not specify any value.
  RETURN;
END;
/

-- Here we use the result of this function in a SQL query.
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));

-- Here we use the result of this function in a PL/SQL block.
DECLARE
  total NUMBER := 0;
  price_type VARCHAR2(1);
BEGIN
  FOR item IN (SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable))))
  LOOP
-- Access the values of each output row.
-- We know the column names based on the declaration of the output type.
-- This computation is just for illustration.
    total := total + item.price;
    price_type := item.price_type;
  END LOOP;
END;
/
```

独自の集計関数のコード化

一連の行を分析して結果値を計算するために、次のように SUM などの組込み集計と同様に機能する集計関数をコード化することができます。

- 次のようなメンバー関数を定義する SQL オブジェクト・タイプを定義します。
  - ODCIAggregateInitialize
  - ODCIAggregateIterate
  - ODCIAggregateMerge
  - ODCIAggregateTerminate
- メンバー関数をコーディングします。特に、ODCIAggregateIterate は、処理された各行に対して一度コールされたように結果を蓄積します。オブジェクト・タイプの属性を使用して中間結果を格納します。
- 集計関数を作成して、新しいオブジェクト・タイプと対応付けます。
- SQL 問合せ、DML 文または組込み集計に使用する他の場所から集計関数をコールします。集計関数に対するコールに DISTINCT や ALL などの一般的なオプションを含めることができます。

参照： このプロセスおよびメンバー関数の要件の詳細は、『Oracle9*i* Data Cartridge Developer's Guide』を参照してください。

10

外部プロシージャのコール

必要とする機能が提供されない言語を使用する場合、または別の言語で作成された既存コードを再利用する場合は、外部プロシージャをコールして、その他の言語で作成されたコードを使用できます。

この章の内容は次のとおりです。

複数言語プログラムの概要

Oracle では、次に示す複数の異なる言語での操作が可能です。

- **PL/SQL:** 詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。
- **C:** OCI を使用します。詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。
- **C または C++:** Pro*C/C++ プリコンパイラを使用します。詳細は、『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。
- **COBOL:** Pro*COBOL プリコンパイラを使用します。詳細は、『Pro*COBOL Precompiler プログラマーズ・ガイド』を参照してください。
- **Visual Basic:** OO4O を使用します。
- **Java:** JDBC アプリケーション・プログラム・インタフェース（API）を使用します。詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

以上のような実装言語の中から何をどのように選択すればよいのでしょうか？これらの言語には、それぞれ異なるメリットがあります。使用しやすさ、特定の専門知識を持つプログラマがいるかどうか、移植が必要かどうか、さらに既存コードの有無などが重要な決定要因になります。

ただし、アプリケーションで Oracle をどのように使用するかによって、次のように選択範囲が狭くなる可能性があります。

- PL/SQL は、SQL トランザクション処理に特化した強力な開発ツールです。
- 演算集中型のタスクは、C などの下位レベル言語で最も効率的に実行されます。
- セキュリティの必要性とともに移植の必要がある場合は、Java を選択する場合があります。

最も重要なことは、パフォーマンスの観点では、サーバーのアドレス空間内で実行されるのは PL/SQL および Java メソッドのみであるという認識が必要であるということです。C/C++ メソッドは外部プロシージャとしてディスパッチされ、サーバー・マシン上で実行されますが、データベース・サーバーのアドレス空間外で実行されます。Pro*COBOL および Pro*C はプリコンパイラで、Visual Basic は C で実装されている OCI を介して Oracle にアクセスします。

以上のすべての要因を考慮すると、アプリケーションを複数の言語で実装する必要がある場合が多数発生する可能性があることがわかります。たとえば、サーバーのアドレス空間内で実行される Java の登場によって、既存の Java アプリケーションをデータベース内にインポートし、PL/SQL および SQL から Java 関数をコールしてこのテクノロジを利用するという方法も可能になります。

Oracle8 が登場するまでは、Oracle RDBMS は SQL およびストアド・プロシージャ言語の PL/SQL をサポートしていました。Oracle8 で、PL/SQL に**外部プロシージャ**が導入され、C

関数を PL/SQL の本体として作成できるようになりました。このような C 関数は、PL/SQL からは直接、また SQL からは PL/SQL プロシージャ・コールを介してコールできます。Oracle8*i* では、**コール仕様**というインタフェースが提供されています。コール仕様を使用すると、他の言語から**外部プロシージャ**をコールできます。このサービスは、SQL、PL/SQL、C および Java 間の相互通信用に設計されていますが、このような言語をコールできる基本言語ならどの言語からでもアクセスできます。たとえば、Java または C 以外の言語で作成したプロシージャでも、C からコール可能なプロシージャであれば、SQL または PL/SQL からでも使用できます。使用したい C++ プロシージャがある場合は、そのプロシージャで C++ の `extern "C"` 文を使用してそのプロシージャを C からコールできるようにします。

これは、複数の異なる言語の強みおよび機能をプログラム環境に関係なく利用できるということを表します。固有の制約を持つ 1 つの言語に制限する必要はありません。外部プロシージャの使用によって、特定の目的に特定の言語を使用できるため、再利用性およびモジュール性が向上します。

外部プロシージャの概要

外部プロシージャ（外部ルーチン）とは、動的リンク・ライブラリ（DLL）に格納されているプロシージャ、または Java クラス・メソッドの場合はライブラリ・ユニットのことです。このプロシージャを基本言語に登録し、これをコールして特定の処理を実行できます。

たとえば、PL/SQL を使用する場合、PL/SQL が実行時にライブラリを動的にロードし、次に、プロシージャを PL/SQL サブプログラムであるかのようにコールします。このようなプロシージャは、現行のトランザクションに完全に組み込まれ、データベースをコールバックして SQL 操作を実行できます。

このようなプロシージャは必要なときにのみロードされるため、メモリーが節約されます。コール仕様が実装本体から切り離されているため、コール側プログラムに影響することなく、プロシージャを拡張できます。

外部プロシージャを使用すると、次のようなことができます。

- 演算集中型プログラムをクライアントから実行速度の速いサーバーへ移動できます（ネットワーク通信でのラウンドトリップを回避するため）。
- データベース・サーバーと外部システムおよびデータ・ソースとのインタフェースが実現します。
- データベース・サーバー自体の機能を拡張できます。

外部プロシージャ用のコール仕様の概要

以前のリリースでは、外部プロシージャは PL/SQL ラッパー内の `AS EXTERNAL` 句を介して Oracle に発行しました。このラッパーが、外部 C プロシージャへのマッピングを定義し、このプロシージャをコールできるようにしていました。現在では、外部プロシージャの発行は**コール仕様**を介して行いますが、このコール仕様には、`AS LANGUAGE` 句を介した `AS`

EXTERNAL 機能のサブセットが含まれます。AS LANGUAGE コール仕様を使用すると、今までのように外部 C プロシージャを発行できる他に、Java クラス・メソッドも発行できます。

> **注意：** コール仕様によって、Oracle8 で導入された AS EXTERNAL 句を使用して発行できます。ただし、新しいアプリケーションの場合は、AS LANGUAGE 句を使用するようにしてください。

一般に、コール仕様を使用することによって、次のようなことができます。

- 適切な C または Java のターゲット・プロシージャのディスパッチ
- データ型の変換
- パラメータ・モードのマッピング
- 自動的なメモリー割当ておよびクリーンアップ
- SQL からコールされるパッケージ・ファンクションに必要に応じて指定される純粋度制約
- データベース・トリガーからの Java メソッドまたは C プロシージャのコール
- 位置の柔軟性パフォーマンスを最適化し実装の詳細を隠すために、AS LANGUAGE コール仕様を、パッケージ仕様または型仕様、あるいはパッケージ（または型）本体に入れることができます。

既存プログラムを外部プロシージャとして使用するには、そのプログラムをロードして発行した後、コールします。

外部プロシージャのロード

外部 C プロシージャまたは Java メソッドを PL/SQL で使用できるようにするには、プロシージャまたはメソッドをロードする必要があります。ロード方法は、プロシージャが C で作成されているか Java で作成されているかによって異なります。

> **参照：** 『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。
>
> DLL 作成の詳細は、RDBMS サブディレクトリ public を参照してください。この中にテンプレート makefile があります。

Java クラス・メソッドのロード

Java プログラムのロード方法の 1 つとして、SQL*Plus から対話形式で実行できる CREATE JAVA 文を使用する方法があります。CREATE JAVA 文から Java Virtual Machine（JVM）ライブラリ・マネージャが暗黙的に起動され、そのときにこのライブラリ・マネージャが Java バイナリ（.class ファイル）およびリソースをローカルな BFILE または LOB 列から RDBMS ライブラリ・ユニットにロードします。

コンパイル済の Java クラスが次のオペレーティング・システム・ファイル内に格納されているとします。

```
/home/java/bin/Agent.class
```

ファイル Agent.class から、スキーマ scott 内にクラス・ライブラリ・ユニットを作成するには、2 つの手順を行う必要があります。まず、サーバーのファイル・システム上にディレクトリ・オブジェクトを作成します。ディレクトリ・オブジェクトの名前は、Agent.class のディレクトリ・パスの別名です。

ディレクトリ・オブジェクトを作成するには、次のように、ユーザー scott に CREATE ANY DIRECTORY 権限を付与してから、CREATE DIRECTORY 文を実行する必要があります。

```
CONNECT System/Manager
GRANT CREATE ANY DIRECTORY TO Scott IDENTIFIED BY Tiger;
CONNECT Scott/Tiger
CREATE DIRECTORY Bfile_dir AS '/home/java/bin';
```

これで、次のようにクラス・ライブラリ・ユニットを作成する準備ができました。

```
CREATE JAVA CLASS USING BFILE (Bfile_dir, 'Agent.class');
```

ライブラリ・ユニットの名前は、クラスの名前から導出されます。

別の方法として、コマンドライン・ユーティリティ LoadJava を使用できます。このユーティリティは、Java バイナリおよびリソースを、システムによって生成されたデータベース表にアップロードしてから、CREATE JAVA 文を使用して Java ファイルを RDBMS ライブラリ・ユニットにロードします。Java ファイルは、ファイル・システム、Java IDE、イントラネットまたはインターネットからアップロードできます。

> **参照：**『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

外部 C プロシージャのロード

C で作成された外部プロシージャまたは C からコール可能な外部プロシージャを使用できるように設定するには、開発者および DBA が次の手順を実行します。

注意： この機能は、DLL をサポートするプラットフォームまたは Solaris の .so ライブラリなどの動的にロード可能な共有ライブラリをサポートするプラットフォームのみで使用できます。

---

1. 環境を設定する

DBA は、ファイル tnsname.ora および listener.ora にエントリを追加し、外部プロシージャ専用のリスナー・プロセスを開始することによって、外部プロシージャをコールする環境を設定します。

デフォルトでは、外部プロシージャを処理するエージェントを extproc といい、メイン・アプリケーションと同じデータベース・インスタンス上で実行します。

エージェント・プロセスを起動する、データベース・サーバー、エージェント・プロセスおよびリスナー・プロセスは、すべて同じホスト上に存在する必要があります。

信頼性が非常に重要な場合は、外部プロシージャを別のデータベース・インスタンス（同じホスト上）で実行すると、プロシージャがクラッシュしても何も停止しません。エージェント・プロセスおよび外部プロシージャのコードが個別のインスタンスにある場合、データベース・リンクの名前を使用してサーバーを指定できます。

参照： 『Oracle9*i* データベース管理者ガイド』を参照してください。

リスナーは、外部プロシージャ・エージェントに必要な環境変数（ORACLE_HOME、ORACLE_SID、LD_LIBRARY_PATH など）をいくつか設定します。また、リスナーが listener.ora エントリの ENVS セクションで固有の環境変数を定義すると、その環境変数はエージェント・プロセスに渡されます。それ以外は、エージェントに対してクリーンな環境を提供します。エージェント用に設定される環境変数は、クライアントおよびサーバー用に設定される環境変数から独立しています。したがって、エージェント・プロセス内で実行される外部プロシージャは、クライアントまたはサーバーのプロセス用に設定される環境変数を読み込むことはできません。

注意： 環境変数自体は標準 C プロシージャの setenv() および getenv() を使用してそれぞれ設定し読み込むことができます。この方法で設定された環境変数は、エージェント・プロセス専用です。そのプロセス内で実行されるすべての関数で読み込めますが、同一ホスト上で実行されている他のプロセスで読み込むことはできません。

---

2. DLL を識別する

ここでの DLL とは、外部プロシージャを格納する動的ロードが可能な任意のオペレーティング・システム・ファイルです。

安全上の理由から、DLL へのアクセスは DBA が制御します。DBA は、CREATE LIBRARY 文を使用して、DLL を表す**別名ライブラリ**というスキーマ・オブジェクトを作成します。次に、許可されているユーザーの場合は、DBA が別名ライブラリに対する EXECUTE 権限を付与します。このかわりに、DBA は、ユーザーに CREATE ANY LIBRARY 権限を付与することもあり、この場合、ユーザーは、次の構文を使用して自分の別名ライブラリを作成できます。

```
CREATE LIBRARY [schema_name.]library_name
  {IS | AS} 'file_path'
  [AGENT 'agent_link'];
```

リンカーは DLL 名のみへの参照を解決することができないため、DLL にはフル・パスを指定する必要があります。次の例では、DLL である utils.so を表す別名ライブラリ c_utils を作成します。

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

DLL を柔軟に指定するには、表記規則 ${*VAR_NAME*} を使用してパスのルート部分を環境変数として指定し、その変数を listener.ora エントリの ENVS セクションに設定できます。

次の例では、agent_link という名前で指定されるエージェントを使用して、ライブラリ C_Utils で外部プロシージャを実行します。環境変数 EP_LIB_HOME は、エージェントによって /usr/bin/dll などのそのインスタンス用の適切なパスに展開されます。

```
create or replace database link agent_link using 'agent_tns_alias';
create or replace library C_utils is
  '${EP_LIB_HOME}/utils.so' agent 'agent_link';
```

注意：

a. Windows システムの場合、パスのドライブ文字およびバックスラッシュ記号（\）を使用して、パスを指定します。

b. この方法は、VMS システムには適用できません。VMS システムは、listener.ora の ENVS セクションをサポートしていません。

---

3. 外部プロシージャを発行する

新しい外部 C プロシージャを作成してから、これを DLL に追加します。プロシージャが DLL 内にある場合、次の項で説明されるコール仕様のメカニズムを使用して、プロシージャを公開します。

外部プロシージャの公開

Oracle で使用できる外部プロシージャは、コール仕様で公開される外部プロシージャのみです。Java クラス・メソッドまたは C 外部プロシージャの名前、パラメータ型および戻り型が、対応する SQL の要素にマップされます。これは PL/SQL ストアド・サブプログラムと同様に作成しますが、AS LANGUAGE 句を宣言や BEGIN..END ブロックではなく、本体内に作成します。

AS LANGUAGE 句は、次のことを指定します。

- プロシージャの作成言語
- Java メソッド
  - Java メソッドのシグネチャ
- C プロシージャ
  - DLL に対応する C プロシージャ用の別名ライブラリ
  - DLL 内の C プロシージャの名前
  - パラメータの受渡し方法を指定するための様々なオプション
  - 異なるマシン上でプロシージャを実行するために、外部プロシージャ・エージェントの名前を保持するパラメータ（ある場合）

プロシージャ、ファンクション、パッケージ仕様部、パッケージ本体、型仕様または型本体用の通常の CREATE OR REPLACE 構文を使用して、宣言を開始します。

コール仕様は、名前宣言およびパラメータ宣言の後に指定します。構文は次のとおりです。

```
{IS | AS} LANGUAGE {C | JAVA}
```

> **注意：** Oracle では、ANSI SQL92 外部プロシージャを PL/SQL 用に変更して使用しますが、ANSI キーワードの AS EXTERNAL がコール仕様構文に置き換えられています。Java クラス・メソッド用として導入されたこの新しい構文が、C プロシージャにまで拡張されています。

これに続いて、次のいずれかを指定します。

```
NAME  java_string_literal_name
```

java_string_literal_name は、Java メソッドのシグネチャです。

```
LIBRARY library_name
[NAME c_string_literal_name]
[WITH CONTEXT]
[PARAMETERS (external_parameter[, external_parameter]...)];
```

library_name は別名ライブラリの名前で、c_string_literal_name は外部 C プロシージャの名前です。external_parameter は次を意味します。

```
{  CONTEXT
 | SELF [{TDO | property}]
 | {parameter_name | RETURN} [property] [BY REFERENCE] [external_datatype]}
```

property は次を意味します。

```
{INDICATOR [{STRUCT | TDO}] | LENGTH | MAXLEN | CHARSETID | CHARSETFORM}
```

注意： Java と異なり、C では SQL 型は認識されません。したがって、構文がより複雑になります。

Java クラス・メソッド用の AS LANGUAGE 句

AS LANGUAGE 句は、PL/SQL と Java クラス・メソッドの間のインタフェースです。

参照：『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

外部 C プロシージャ用の AS LANGUAGE 句

次の副次句は、PL/SQL に対して、外部 C プロシージャを検索する場所、プロシージャのコール方法、プロシージャに何を渡すかを指示します。必要な副次句は、LIBRARY 副次句のみです。

LIBRARY

ローカルな別名ライブラリを指定します（リモート・ライブラリの指定にデータベース・リンクは指定できません）。ライブラリ名は PL/SQL 識別子です。したがって、名前を二重引用符で囲むと、名前の大 / 小文字が区別されます（デフォルトでは、名前は大文字で格納されます）。別名ライブラリには EXECUTE 権限が必要です。

NAME

コール対象の外部 C プロシージャを指定します。プロシージャ名を二重引用符で囲むと、名前の大 / 小文字が区別されます（デフォルトでは、名前は大文字で格納されます）。この副次句を省略すると、プロシージャ名は PL/SQL サブプログラム名を大文字で表したものがデフォルト設定されます。

注意： LANGUAGE および CALLING STANDARD は、旧版の AS EXTERNAL 句のみに適用されます。

LANGUAGE

外部プロシージャが作成されている 3GL を指定します。この副次句を省略すると、言語名は C にデフォルト設定されます。

CALLING STANDARD

外部プロシージャがコンパイルされた Windows NT のコール規格（C または Pascal）を指定します（Pascal コール規格では、スタック上の引数の順序が逆になり、コールされる関数がスタックからデータを取得する必要があります）。この副次句を省略すると、コール規格は C にデフォルト設定されます。

WITH CONTEXT

コンテキスト・ポインタが外部プロシージャに渡されることを指定します。コンテキスト・データ構造体は外部プロシージャに対して不透明ですが、外部プロシージャによってコールされるサービス・プロシージャでは使用できます。

PARAMETERS

外部プロシージャに渡されるパラメータの位置およびデータ型を指定します。現在の長さや最大長などのパラメータ・プロパティや、パラメータの受渡し方法（値によるか参照によるか）も指定できます。

AGENT IN

このプロシージャを実行するエージェント・プロセスの名前を保持するパラメータを指定します。これは、外部プロシージャ・エージェントが、複数のエージェント・プロセスを使用して実行される場合に指定します。これによって、1 つの外部プロシージャのエージェント・プロセスがクラッシュした場合の信頼性が保証されます。エージェント・プロセスの名前（データベース・リンクの名前に対応）を渡すことができ、tnsnames.ora および listener.ora が両方のインスタンス間で正しく設定されている場合、外部プロシージャは、もう一方のインスタンスでも起動します。両方のインスタンスは、同じホスト上に存在する必要があります。

この操作は、CREATE LIBRARY 文の AGENT 句の場合と同様です。AGENT IN を使用して実行時の値を指定すると、柔軟性が向上します。

このようにエージェント名を指定すると、別名ライブラリで宣言されるエージェント名はいずれもオーバーライドされます。エージェント名が指定されていない場合は、extproc エージェントがコール側プログラムと同じインスタンス上でデフォルト設定されます。

Java クラス・メソッドの発行

Java クラスおよびそのメソッドは、RDBMS ライブラリ・ユニットに格納されます。この中には、LOADJAVA ユーティリティまたは SQL 文 CREATE JAVA を使用して、Java ソース、バイナリおよびリソースをロードできます。ライブラリ・ユニットは、たとえば C で作成された DLL に類似していますが、DLL には複数のプロシージャを入れられるのに対して、ライブラリ・ユニットは Java クラスと 1 対 1 で対応します。

> **参照：**『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

NAME 句文字列は、Java メソッドを一意に識別します。PL/SQL のファンクションまたはプロシージャおよび Java は、パラメータが対応している必要があります。Java メソッドがパラメータをとらない場合は、空のパラメータ・リストを作成する必要があります。

Java クラスを RDBMS にロードするとき、クラスは SQL に対して自動的には発行されません。これは、ほとんどの Java クラスのメソッドは他の Java クラスからのみコールされるか、該当する SQL 型が存在しないパラメータをとるためです。

次に示す、引数の階乗を戻す Java メソッド（J_calcFactorial）を発行するものとします。

```
package myRoutines.math;
public class Factorial {
   public static int J_calcFactorial (int n) {
      if (n == 1) return 1;
      else return n * J_calcFactorial(n - 1);
   }
}
```

次のコール仕様では、SQL*Plus を使用して、Java メソッド J_calcFactorial を PL/SQL ストアド・ファンクション plsToJavaFac_func として発行します。

```
CREATE OR REPLACE FUNCTION Plstojavafac_func (N NUMBER) RETURN NUMBER AS
   LANGUAGE JAVA
   NAME 'myRoutines.math.Factorial.J_calcFactorial(int) return int';
```

外部 C プロシージャの発行

次の例では、C 関数 Cdivisor_func を外部関数として発行する plsCallsCdivisor_func という PL/SQL スタンドアロン・ファンクションを作成します。

```
CREATE OR REPLACE FUNCTION Plscallscdivisor_func (
/* Find greatest common divisor of x and y: */
   x     BINARY_INTEGER,
   y     BINARY_INTEGER)
```

```
RETURN BINARY_INTEGER
AS LANGUAGE C
   LIBRARY C_utils
   NAME "Cdivisor_func"; /* Quotation marks preserve case. */
```

コール仕様の位置

Java クラス・メソッドおよび外部 C プロシージャはともに、次のいずれかの位置にコール仕様を指定できます。

- スタンドアロンの PL/SQL プロシージャおよびファンクション
- PL/SQL パッケージ仕様部
- PL/SQL パッケージ本体
- オブジェクト型仕様
- オブジェクト型本体

> **注意：** Oracle8 では、AS EXTERNAL コール仕様をパッケージ本体または型本体に入れることはできませんでした。

スタンドアロン PL/SQL ファンクションでのコール仕様の例はすでに説明しました。ここでは、その他の位置について例をいくつか示します。

> **注意：** 次のいくつかの例では、コール仕様の完全指定に AUTHID 句および SQL_NAME_RESOLVE 句が必要な場合と必要でない場合があります。この 2 つの句の配置およびデフォルトに関する規則については、8-7 ページの「実行者権限を使用した動的操作の実行」を参照してください。

例 : PL/SQL パッケージ内へのコール仕様の配置

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;
```

例 : PL/SQL パッケージ本体内へのコール仕様の配置

```
CREATE OR REPLACE PACKAGE Demo_pack
   AUTHID CURRENT_USER
AS
   PROCEDURE plsToC_demoExternal_proc(x BINARY_INTEGER, y VARCHAR2, z DATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
   SQL_NAME_RESOLVE CURRENT_USER
AS
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE JAVA
      NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
END;
```

例 : オブジェクト型仕様内へのコール仕様の配置

> **注意 :** 次のようなデータ構造を設定しないと機能しない例もあります。
>
> ```
> CONN SYS/CHANGE_ON_INSTALL AS SYSDBA;
> GRANT CREATE ANY LIBRARY TO scott;
> CONNECT scott/tiger
> CREATE OR REPLACE LIBRARY SOMELIB AS '/tmp/lib.so';
> ```

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID DEFINER
AS OBJECT
   (Attribute1   VARCHAR2(2000), SomeLib varchar2(20),
   MEMBER PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
    --  PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE)
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE, SELF)
);
```

例 : オブジェクト型本体内へのコール仕様の配置

```
CREATE OR REPLACE TYPE Demo_typ
AUTHID CURRENT_USER
AS OBJECT
   (attribute1 NUMBER,
   MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
);
```

```
CREATE OR REPLACE TYPE BODY Demo_typ
AS
   MEMBER PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE JAVA
      NAME 'pkg1.class4.J_demoExternal(int,java.lang.String,java.sql.Date)';
END;
```

例 : AUTHID を指定した Java

スタンドアロン PL/SQL サブプログラム内に Java クラス・メソッドを発行する例を次に示します。

```
CREATE OR REPLACE PROCEDURE plsToJ_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2,
z DATE)
   AUTHID CURRENT_USER
AS LANGUAGE JAVA
   NAME 'pkg1.class4.methodProc1(int,java.lang.String,java.sql.Date)';
```

例 : オプションの AUTHID を指定した C

スタンドアロン PL/SQL プログラム内に C プロシージャを AS EXTERNAL として発行する例を次に示します。AUTHID 句はオプションです。これによって、Oracle8 の外部プロシージャとの互換性が保たれます。

```
CREATE OR REPLACE PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2,
z DATE)
AS
   EXTERNAL
   LANGUAGE C
   NAME "C_demoExternal"
   LIBRARY SomeLib
   WITH CONTEXT
   PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
```

例 : パッケージ内でのコール仕様の混合使用

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
   PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);
   PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE);

   PROCEDURE plsToJ_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   IS LANGUAGE JAVA
      NAME 'pkg1.class4.J_InSpec_meth(int,java.lang.String,java.sql.Date)';
```

```
PROCEDURE C_InSpec_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;

CREATE OR REPLACE PACKAGE BODY Demo_pack
AS
PROCEDURE plsToC_InBodyOld_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS EXTERNAL
      LANGUAGE C
      NAME "C_InBodyOld"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToC_demoExternal_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);

PROCEDURE plsToC_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_InBody"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
PROCEDURE plsToJ_InBody_proc (x BINARY_INTEGER, y VARCHAR2, z DATE)
   IS LANGUAGE JAVA
      NAME 'pkg1.class4.J_InBody_meth(int,java.lang.String,java.sql.Date)';
END;
```

コール仕様による Java クラス・メソッドへのパラメータ受渡し

参照： 『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

コール仕様による外部 C プロシージャへのパラメータの受渡し

コール仕様によって、PL/SQL データ型と C データ型とのマッピングが可能になります。データ型のマッピングを次に示します。

外部 C プロシージャへのパラメータの受渡しは、次のような理由で複雑になります。

- 使用可能な PL/SQL 集合が、C データ型の集合と 1 対 1 で対応していない。
- C とは異なり、PL/SQL には NULL かどうかという RDBMS 概念が含まれる。したがって、PL/SQL パラメータは NULL にできますが、C パラメータは `NULL` にできません。
- 外部プロシージャで、`CHAR`、`LONG RAW`、`RAW` および `VARCHAR2` パラメータの現在の長さまたは最大長が必要。
- 外部プロシージャで、`CHAR`、`VARCHAR2` および `CLOB` パラメータに関するキャラクタ・セット情報が必要。
- PL/SQL で、外部プロシージャによって戻された値の現在の長さ、最大長または NULL 状態が必要。

次の項では、前述の状況に対処するパラメータ・リストを指定する方法を説明します。

注意： C 外部プロシージャに渡すことができるパラメータの最大数は 128 です。ただし、FLOAT 型または DOUBLE 型のパラメータを値渡しする場合は、最大数は 127 以下になります。どのくらい減るかは、パラメータの個数およびオペレーティング・システムによって異なります。目安としては、値渡しされる FLOAT 型または DOUBLE 型のパラメータ 1 つを 2 つ分として数えます。

データ型の指定

パラメータは外部プロシージャに直接渡さないでください。かわりに、外部プロシージャを発行した PL/SQL サブプログラムに渡します。したがって、パラメータには PL/SQL データ型を指定する必要があります。PL/SQL データ型は、それぞれデフォルトの外部データ型にマップされます（表 10-1 を参照）。

表 10-1 パラメータのデータ型のマッピング

PL/SQL データ型	サポートされている外部データ型	デフォルトの外部データ型
BINARY_INTEGER BOOLEAN PLS_INTEGER	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	INT
NATURAL NATURALN POSITIVE POSITIVEN SIGNTYPE	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	UNSIGNED INT
FLOAT REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR CHARACTER LONG NCHAR NVARCHAR2 ROWID VARCHAR VARCHAR2	STRING OCISTRING	STRING
LONG RAW RAW	RAW OCIRAW	RAW
BFILE BLOB CLOB NCLOB	OCILOBLOCATOR	OCILOBLOCATOR
NUMBER	OCINUMBER	OCINUMBER

表 10-1 パラメータのデータ型のマッピング（続き）

PL/SQL データ型	サポートされている外部データ型	デフォルトの外部データ型
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
コンポジット・オブジェクト型 : ADT	dvoid	dvoid
コンポジット・オブジェクト型 : コレクション（VARRAYS、ネストした表、index-by tables	OCICOLL	OCICOLL

外部データ型のマッピング

外部データ型はそれぞれ C データ型にマップされ、データ型変換が暗黙的に実行されます。C プロトタイプ・パラメータの宣言時のエラーを回避するには、表 10-2 を参照してください。この表には、特定の外部データ型および PL/SQL パラメータ・モードに指定する C データ型が示されています。たとえば、OUT パラメータの外部データ型が STRING 型の場合は、C プロトタイプにはデータ型 char * を指定します。

表 10-2 外部データ型のマッピング

PL/SL 型に対応する外部データ型	IN モードまたは RETURN モードの場合、C プロトタイプで指定 ...	参照による IN モードまたは参照による RETURN モードの場合、C プロトタイプで指定 ...	IN OUT モードまたは OUT モードの場合、C プロトタイプで指定 ...
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *

表 10-2 外部データ型のマッピング（続き）

PL/SL 型に対応する外部データ型	IN モードまたは RETURN モードの場合、C プロトタイプで指定 ...	参照による IN モードまたは参照による RETURN モードの場合、C プロトタイプで指定 ...	IN OUT モードまたは OUT モードの場合、C プロトタイプで指定 ...
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
CHAR	char	char *	char *
UNSIGNED CHAR	unsigned char	unsigned char *	unsigned char *
SHORT	short	short *	short *
UNSIGNED SHORT	unsigned short	unsigned short *	unsigned short *
INT	int	int *	int *
UNSIGNED INT	unsigned int	unsigned int *	unsigned int *
LONG	long	long *	long *
UNSIGNED LONG	unsigned long	unsigned long *	unsigned long *
SIZE_T	size_t	size_t *	size_t *
SB1	sb1	sb1 *	sb1 *
UB1	ub1	ub1 *	ub1 *
SB2	sb2	sb2 *	sb2 *
UB2	ub2	ub2 *	ub2 *
SB4	sb4	sb4 *	sb4 *
UB4	ub4	ub4 *	ub4 *
FLOAT	float	float *	float *
DOUBLE	double	double *	double *
STRING	char *	char *	char *
RAW	unsigned char *	unsigned char *	unsigned char *
OCILOBLOCATOR	OCILobLocator *	OCILobLocator **	OCILobLocator **
OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
OCIDATE	OCIDate *	OCIDate *	OCIDate *

表 10-2 外部データ型のマッピング（続き）

PL/SL 型に対応する外部データ型	IN モードまたは RETURN モードの場合、C プロトタイプで指定 ...	参照による IN モードまたは参照による RETURN モードの場合、C プロトタイプで指定 ...	IN OUT モードまたは OUT モードの場合、C プロトタイプで指定 ...
OCICOLL	OCIColl *、OCIArray * または OCITable *	OCIColl **、OCIArray ** または OCITable **	OCIColl **、OCIArray ** または OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT（FINAL 型）	dvoid*	dvoid*	dvoid*
ADT（NOT FINAL 型）	dvoid*	dvoid*	dvoid**

コンポジット・オブジェクト型は、自己記述型でありません。記述は、**型記述子オブジェクト**（TDO）に格納されています。オブジェクトおよびオブジェクトのインジケータ構造体に事前定義の OCI データ型はありませんが、Oracle の**オブジェクト型トランスレータ**（OTT）によって生成されるデータ型を使用する必要があります。INDICATOR およびコンポジット・オブジェクトのオプションの TDO 引数は、通常、C データ型の OCIType* を持ちます。

REF およびコレクション引数の OCICOLL はオプションで、完全性を保つためにのみ存在しています。REF またはコレクションを別のデータ型にマップすることは（その逆も）できません。

IN および IN OUT パラメータ・モードの BY VALUE/BY REFERENCE

BY VALUE を指定すると、スカラー IN および RETURN 引数が値渡しされます（デフォルト）。BY REFERENCE を指定すると、参照によって渡すことができます。

デフォルトまたは BY REFERENCE を指定した場合は、スカラー IN OUT および OUT 引数が参照によって渡されます。IN OUT および OUT 引数での BY VALUE の指定は C 用にはサポートされていません。BY REFERENCE/VALUE 句の使用は、デフォルトで値渡しされる外部データ型に制限されます。これは、次の外部データ型の IN および RETURN 引数に適用されます。

```
[UNSIGNED] CHAR
[UNSIGNED] SHORT
[UNSIGNED] INT
[UNSIGNED] LONG
SIZE_T
SB1
SB2
SB4
```

```
UB1
UB2
UB4
FLOAT
DOUBLE
```

このリストに記載されていない外部データ型の IN および RETURN 引数、IN OUT および OUT 引数は、すべて参照によって渡されます。

PARAMETERS 句

一般に、外部プロシージャを発行する PL/SQL サブプログラムは、次の例に示されているように、仮パラメータのリストを宣言します。

> **注意：** 次のようなデータ構造を設定しないと機能しない例もあります。
>
> ```
> CREATE LIBRARY MathLib AS '/tmp/math.so';
> ```

```
CREATE OR REPLACE FUNCTION Interp_func (
/* Find the value of y at x degrees using Lagrange interpolation: */
   x    IN FLOAT,
   y    IN FLOAT)
RETURN FLOAT AS
   LANGUAGE C
   NAME "Interp_func"
   LIBRARY MathLib;
```

それぞれの仮パラメータ宣言では、名前、パラメータ・モードおよび（デフォルトの外部データ型にマップされる）PL/SQL データ型が指定されます。これが、外部プロシージャが必要とするすべての情報である可能性があります。これがすべてではない場合は、PARAMETERS 句を使用して追加情報を指定できます。指定できる追加情報は次のとおりです。

- デフォルト以外の外部データ型
- パラメータの現在の長さまたは最大長（あるいはその両方）
- パラメータの NULL/NOT NULL インジケータ
- キャラクタ・セットの ID および形式
- リスト内のパラメータの位置
- IN パラメータを渡す方法（値による方法か参照による方法）

PARAMETERS 句を使用する場合、次のことを認識しておく必要があります。

- 各仮パラメータには、PARAMETERS 句に対応するパラメータが必要です。

- WITH CONTEXT 句を含める場合は、パラメータ・リスト内でのコンテキスト・ポインタの位置を示すパラメータ CONTEXT を指定する必要があります。
- 外部プロシージャがファンクションの場合は、パラメータ RETURN を最後の位置に指定する必要があります。RETURN を指定しないときは、デフォルトの外部型が使用されます。

デフォルトのデータ型マッピングのオーバーライド

場合によっては、PARAMETERS 句を使用して、デフォルトのデータ型マッピングをオーバーライドすることができます。たとえば、外部データ型 INT から外部データ型 CHAR に PL/SQL データ型の BOOLEAN をマッピングしなおすことができます。

プロパティの指定

PARAMETERS 句は、PL/SQL の仮パラメータおよびファンクション結果に関する追加情報を外部プロシージャに渡すために使用することもできます。これは、次のプロパティを 1 つ以上指定して行います。

```
INDICATOR [{STRUCT | TDO}]
LENGTH
MAXLEN
CHARSETID
CHARSETFORM
SELF
```

次の表は、使用可能外部データ型とデフォルト外部データ型、PL/SQL データ型および特定のプロパティに使用できる PL/SQL パラメータを示します。(C から PL/SQL にデータを戻す場合に指定する) MAXLEN は、IN パラメータには適用できません。

表 10-3 プロパティのデータ型のマッピング

プロパティ	使用可能外部データ型 (C)	デフォルトの外部データ型 (C)	使用可能 PL/SQL データ型	使用可能 PL/SQL モード	デフォルトの PL/SQL 受渡し方法
INDICATOR	SHORT	SHORT	all scalars	IN	BY VALUE
				IN OUT	BY REFERENCE
				OUT	BY REFERENCE
				RETURN	BY REFERENCE
LENGTH	[UNSIGNED] SHORT	INT	CHAR	IN	BY VALUE
	[UNSIGNED] INT		LONG RAW	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	OUT	BY REFERENCE
			VARCHAR2	RETURN	BY REFERENCE
MAXLEN	[UNSIGNED] SHORT	INT	CHAR	IN OUT	BY REFERENCE
	[UNSIGNED] INT		LONG RAW	OUT	BY REFERENCE
	[UNSIGNED] LONG		RAW	RETURN	BY REFERENCE
			VARCHAR2		
CHARSETID	[UNSIGNED] SHORT	[UNSIGNED] INT	CHAR	IN	BY VALUE
CHARSETFORM	[UNSIGNED] INT		CLOB	IN OUT	BY REFERENCE
	[UNSIGNED] LONG		VARCHAR2	OUT	BY REFERENCE
				RETURN	BY REFERENCE

次の例では、PARAMETERS 句は PL/SQL の仮パラメータおよびファンクション結果のプロパティを指定します。

```
CREATE OR REPLACE FUNCTION plsToCparse_func  (
   x   IN BINARY_INTEGER,
   Y   IN OUT CHAR)
RETURN CHAR AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_parse"
   PARAMETERS (
      x,            -- stores value of x
      x INDICATOR,  -- stores null status of x
      y,            -- stores value of y
      y LENGTH,     -- stores current length of y
      y MAXLEN,     -- stores maximum length of y
      RETURN INDICATOR,
      RETURN);
```

この PARAMETERS 句を使用すると、C プロトタイプは次のようになります。

```
char * C_parse(int x, short x_ind, char *y, int *y_len,
               int *y_maxlen, short *retind);
```

C プロトタイプ内の追加パラメータは、INDICATOR（x 用）、LENGTH（y 用）および MAXLEN（y 用）の他に、PARAMETERS 句のファンクション結果の INDICATOR にも対応します。パラメータ RETURN は結果値を格納し、C 関数識別子に対応します。

INDICATOR

INDICATOR は、別のパラメータが NULL かどうかを示す値を持つパラメータです。PL/SQL では、RDBMS における NULL かどうかという概念が言語に組み込まれているため、インジケータは必要ありません。ただし、外部プロシージャでは、パラメータまたはファンクション結果が NULL かどうかを認識することが必要な場合があります。また、外部プロシージャでは、戻り値が実際に NULL でありそれに応じた処理が必要であることをサーバーに指示する必要がある場合もあります。

このような場合は、プロパティ INDICATOR を使用して、インジケータを仮パラメータに対応付けることができます。PL/SQL サブプログラムがファンクションの場合は、前述のようにインジケータをファンクション結果に対応付けることもできます。

インジケータの値を調べるには、定数 OCI_IND_NULL および OCI_IND_NOTNULL を使用できます。インジケータが OCI_IND_NULL の場合は、対応付けられているパラメータまたはファンクション結果は NULL です。インジケータが OCI_IND_NOTNULL の場合は、対応付けられているパラメータまたはファンクション結果は NULL ではありません。

IN パラメータは読込み専用ですが、この場合、INDICATOR は（BY REFERENCE を指定しないかぎり）値渡しされ、（BY REFERENCE を指定しても）読込み専用です。OUT、IN OUT および RETURN パラメータの場合、INDICATOR はデフォルトで参照によって渡されます。

INDICATOR には STRUCT オプションまたは TDO オプションを指定することもできます。INDICATOR をオブジェクトのプロパティとして指定することはサポートされていないため、また、オブジェクトの引数には INDICATOR スカラーのかわりに完全なインジケータ構造体があるため、STRUCT オプションを使用して指定する必要があります。コンポジット・オブジェクトおよびコレクションには、型記述子オブジェクト（TDO）を使用する必要があります。

LENGTH および MAXLEN

PL/SQL では、RAW パラメータまたは文字列パラメータの長さを示す標準的な方法はありません。ただし、このようなパラメータの長さの受渡しを外部プロシージャとの間で行う必要がある場合が数多くあります。プロパティ LENGTH または MAXLEN を使用すると、仮パラメータの現在の長さおよび最大長を格納するパラメータを指定できます。

注意： 型が RAW または LONG RAW のパラメータの場合は、プロパティ LENGTH を使用してください。また、そのパラメータが IN OUT および NULL か、または OUT および NULL の場合は、対応する C パラメータを長さ 0（ゼロ）に設定してください。

IN パラメータの場合は、LENGTH は（BY REFERENCE を指定しないかぎり）値渡しされ、読込み専用です。OUT、IN OUT および RETURN パラメータの場合は、LENGTH はデフォルトで参照によって渡されます。

前述のように、MAXLEN は IN パラメータには適用されません。OUT、IN OUT および RETURN パラメータの場合、MAXLEN は参照によって渡され、読込み専用です。

CHARSETID および CHARSETFORM

Oracle ではグローバリゼーション・サポートを提供していますが、これを使用すると、シングルバイトおよびマルチバイトの文字データを処理し、キャラクタ・セット間で変換を行うことができます。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。

デフォルトでは、サーバーおよびエージェントが同じ $ORACLE_HOME 値を使用する場合、エージェントはサーバー（ALTER SESSION コマンドで指定した設定を含む）と同一のグローバリゼーション・サポート設定を使用します。

エージェントが別の $ORACLE_HOME（2 つの異なる別名およびシンボリック・リンクによって同じ場所が指定されている場合でも）で実行されている場合、キャラクタ・セット以外はサーバーと同じグローバリゼーション・サポート設定を使用します。エージェント用のデフォルト・キャラクタ・セットはエージェントの環境設定 NLS_LANG および NLS_NCHAR によって定義します。

プロパティ CHARSETID および CHARSETFORM は、文字データを渡す際にデフォルト以外のキャラクタ・セットにする場合に指定します。CHAR、CLOB および VARCHAR2 型パラメータの場合は、CHARSETID および CHARSETFORM を使用してキャラクタ・セットの ID および形式を外部プロシージャに渡すことができます。

IN パラメータの場合、CHARSETID および CHARSETFORM は（BY REFERENCE を指定しないかぎり）値渡しされ、（BY REFERENCE を指定しても）読込み専用です。OUT、IN OUT および RETURN パラメータの場合、CHARSETID および CHARSETFORM は参照によって渡され、読込み専用です。

これらのプロパティの OCI 属性名は、OCI_ATTR_CHARSET_ID および OCI_ATTR_CHARSET_FORM です。

参照： OCI で各国語データを使用する場合の詳細は、『Oracle Call Interface プログラマーズ・ガイド』および『Oracle9*i* Database グローバリゼーション・サポート・ガイド』を参照してください。

パラメータの再配置

外部プロシージャの仮パラメータは、それぞれ PARAMETERS 句の中に対応するパラメータが必要です。PL/SQL ではパラメータを位置ではなく名前によって対応付けるため、対応するパラメータの位置は異なる可能性があります。ただし、PARAMETERS 句および外部プロシージャのための C プロトタイプでは、同数のパラメータが同じ順序で必要です。

SELF の使用

SELF は、オブジェクト型のメンバーであるファンクションまたはプロシージャに常に存在する引数で、オブジェクトのインスタンスそのものです。ほとんどの場合、この引数は暗黙的で、PL/SQL プロシージャの引数リストには含まれていません。ただし、SELF は、PARAMETERS 句の引数として明示的に指定する必要があります。

たとえば、ユーザーが、個人の名前および誕生日で構成される Person オブジェクトを作成し、さらに、このオブジェクト型の表を作成するとします。ユーザーは、後でこの表の各 Person の年令を判断する必要があるとします。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
GRANT CONNECT,RESOURCE,CREATE LIBRARY TO scott IDENTIFIED BY tiger;
CONNECT scott/tiger
CREATE OR REPLACE LIBRARY agelib UNTRUSTED IS
  '/tmp/scott1.so';.
```

この例は、Solaris 専用です。他のプラットフォームでは、他のライブラリまたはパスが必要になる場合があります。

SQL*Plus では、Person オブジェクト型は次のように作成できます。

```
CREATE OR REPLACE TYPE Person1_typ AS OBJECT
( Name      VARCHAR2(30),
  B_date    DATE,
  MEMBER FUNCTION calcAge_func RETURN NUMBER,
  PRAGMA RESTRICT_REFERENCES(calcAge_func, WNDS)
);
```

通常、メンバー関数は PL/SQL で実装しますが、この例では、これを外部プロシージャとして作成します。これを行うには、このメンバー関数の本体を次のように宣言します。

```
CREATE OR REPLACE TYPE BODY Person1_typ AS
  MEMBER FUNCTION calcAge_func RETURN NUMBER
```

```
  AS LANGUAGE C
  NAME "age"
  LIBRARY agelib
  WITH CONTEXT
  PARAMETERS
  ( CONTEXT,
    SELF,
    SELF INDICATOR STRUCT,
    SELF TDO,
    RETURN INDICATOR
  );
END;
```

calcAge_func メンバー関数は引数をとらず、数値を戻すのみです。メンバー関数は、常に、対応付けられているオブジェクト型のインスタンスに対して起動されます。オブジェクト・インスタンス自体は、常にメンバー関数の暗黙的な引数になります。暗黙的な引数を参照するには、SELF キーワードを使用します。これは、PARAMETERS 句の中で SELF への参照をサポートすることによって、外部プロシージャ構文内に組み込まれています。

次に、対応表が作成され移入されます。

```
CREATE TABLE Person_tab OF Person1_typ;

INSERT INTO Person_tab VALUES
   ('SCOTT', TO_DATE('14-MAY-85'));

INSERT INTO Person_tab VALUES
   ('TIGER', TO_DATE('22-DEC-71'));
```

最後に、この表から対象の情報を取得します。

```
SELECT p.name, p.b_date, p.calcAge_func() FROM Person_tab p;

NAME                           B_DATE    P.CALCAGE_
------------------------------ --------- ----------
SCOTT                          14-MAY-85          0
TIGER                          22-DEC-71          0
```

外部メンバー関数と、OTT によって生成される構造体の定義を実装する C コードを次に示します。

```
#include <oci.h>

struct PERSON
{
    OCIString   *NAME;
    OCIDate      B_DATE;
};
```

```
typedef struct PERSON PERSON;

struct PERSON_ind
{
    OCIInd    _atomic;
    OCIInd    NAME;
    OCIInd    B_DATE;
};
typedef struct PERSON_ind PERSON_ind;

OCINumber *age (ctx, person_obj, person_obj_ind, tdo, ret_ind)
OCIExtProcContext *ctx;
PERSON         *person_obj;
PERSON_ind     *person_obj_ind;
OCIType        *tdo;
OCIInd         *ret_ind;
{
    sword      err;
    text       errbuf[512];
    OCIEnv    *envh;
    OCISvcCtx *svch;
    OCIError  *errh;
    OCINumber *age;
    int        inum = 0;
    sword      status;

    /* get OCI Environment */
    err = OCIExtProcGetEnv( ctx, &envh, &svch, &errh );

    /* initialize return age to 0 */
    age = (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINumber));
    status = OCINumberFromInt(errh, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
                              age);
    if (status != OCI_SUCCESS)
    {
      OCIExtProcRaiseExcp(ctx, (int)1476);
      return (age);
    }

    /* return NULL if the person object is null or the birthdate is null */
    if ( person_obj_ind->_atomic == OCI_IND_NULL ||
         person_obj_ind->B_DATE  == OCI_IND_NULL )
    {
        *ret_ind = OCI_IND_NULL;
        return (age);
    }
```

```
    /* The actual implementation to calculate the age is left to the reader,
       but an easy way of doing this is a callback of the form:
            select trunc(months_between(sysdate, person_obj->b_date) / 12)
            from dual;
    */
    *ret_ind = OCI_IND_NOTNULL;
    return (age);
}
```

参照によるパラメータ渡し

C では、IN スカラー・パラメータは値渡しされる（パラメータの値が渡される）か、参照によって渡す（値へのポインタが渡される）ことができます。スカラーを指すポインタが外部プロシージャで必要な場合は、BY REFERENCE 句を指定し、参照によってパラメータを渡します。

```
CREATE OR REPLACE PROCEDURE findRoot_proc (
   x IN REAL)
AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_findRoot"
   PARAMETERS (
      x BY REFERENCE);
```

この場合、C プロトタイプは次のように指定します。

```
void C_findRoot(float *x);
```

PARAMETERS 句がない場合は次のように指定します。

```
void C_findRoot(float x);
```

WITH CONTEXT

WITH CONTEXT 句を含めることによって、外部プロシージャで、パラメータ、例外、メモリー割当ておよびユーザー環境に関する情報にアクセスできるようになります。WITH CONTEXT 句は、コンテキスト・ポインタが外部プロシージャに渡されることを指定します。たとえば、次の PL/SQL ファンクションを作成するとします。

```
CREATE OR REPLACE FUNCTION getNum_func (
   x IN REAL)
RETURN BINARY_INTEGER AS LANGUAGE C
   LIBRARY c_utils
   NAME "C_getNum"
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      x BY REFERENCE,
```

```
      RETURN INDICATOR);
```

この場合、C プロトタイプは次のように指定します。

```
int C_getNum(
   OCIExtProcContext *with_context,
   float *x,
   short *retind);
```

コンテキスト・データ構造体は外部プロシージャに対して不透明ですが、外部プロシージャによってコールされるサービス・プロシージャでは使用できます。

PARAMETERS 句も含める場合は、パラメータ・リスト内でのコンテキスト・ポインタの位置を示すパラメータ CONTEXT を指定する必要があります。PARAMETERS 句を省略すると、コンテキスト・ポインタは外部プロシージャに渡される最初のパラメータです。

言語間のパラメータ・モードのマッピング

PL/SQL は、IN、IN OUT および OUT パラメータ・モードのみでなく、値を戻すプロシージャの RETURN 句もサポートします。

> **参照：**『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

PL/SQL および C パラメータ・モードの規則が示されています。

CALL 文による外部プロシージャの実行

これで、Java クラス・メソッドまたは外部 C プロシージャが発行されました。次は、これを起動します。

外部プロシージャは直接コールしないでください。かわりに、外部プロシージャを発行した PL/SQL サブプログラムをコールします。このようなコールは、通常の PL/SQL プロシージャまたはファンクションに対するコールと同じようにコーディングして、次のような場所に入れることができます。

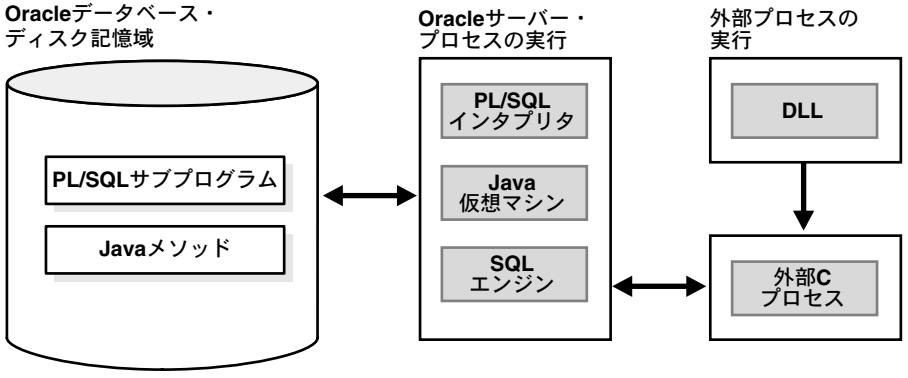
- 無名ブロック
- スタンドアロンのパッケージ・サブプログラム
- オブジェクト型のメソッド
- データベース・トリガー
- SQL 文（パッケージ関数のみへのコール）

次に説明する CALL 文は SELECT に限定されますが、WHERE 句または SELECT 構文のリストのいずれかに入れることができます。

注意： SQL 文からパッケージ・ファンクションをコールするには、PRAGMA RESTRICT_REFERENCES を使用してください。このプラグマはそのファンクションの純粋度レベル（ファンクションが副作用を与えない度合い）を確認します。PL/SQL は、対応する外部プロシージャの純粋度レベルはチェックできません。したがって、外部プロシージャがプラグマに違反していないことを確認してください。そうしないと、予期しない結果になることがあります。

サーバー側またはクライアント側（たとえば、Oracle Forms のようなツール内）で実行される PL/SQL ブロックまたはサブプログラムは、外部プロシージャをコールできます。サーバー側では、外部プロシージャは別のプロセスのアドレス空間内で実行されるため、データベースが保護されます。図 10-1 に、Oracle と外部プロシージャ間の処理を示します。

図 10-1 Oracle と外部プロシージャ



準備

外部プロシージャをコールする前に、実行環境を完全に理解しておく必要があります。具体的には、権限、許可およびシノニムを理解しておいてください。

権限

外部プロシージャがコール仕様経由でコールされるとき、プロシージャは実行者権限ではなく定義者権限によって実行されます。

実行者権限プログラムは、特定のスキーマには限定されません。コール側のサイトで実行され、コールした側の可視性と許可でデータベース項目（表やビューなど）にアクセスします。一方、定義者権限プログラムでは、プログラムが定義されているスキーマに限定されま

す。このプログラムは定義を行う側で、サイトの定義者のスキーマ内で実行され、定義者の可視性および許可でデータベース項目にアクセスします。

許可の管理

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
GRANT CREATE ANY DIRECTORY to scott;
CONNECT scott/tiger
CREATE OR REPLACE DIRECTORY bfile_dir AS '/tmp';
CREATE OR REPLACE JAVA RESOURCE NAMED "appImages" USING BFILE
(bfile_dir,'bfile_audio');
```

外部プロシージャをコールするには、ユーザーはコール仕様およびプロシージャによって使用されるリソースに対する EXECUTE 権限が必要です。

SQL*Plus では、GRANT および REVOKE データ制御文を使用して許可を管理できます。次に例を示します。

```
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Public;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Public;
GRANT EXECUTE ON JAVA RESOURCE "appImages" TO Public;
GRANT EXECUTE ON plsToJ_demoExternal_proc TO Scott;
REVOKE EXECUTE ON plsToJ_demoExternal_proc FROM Scott;
```

参照：

- 『Oracle9*i* SQL リファレンス』を参照してください。
- 『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

外部プロシージャのシノニムの作成

開発者または DBA は、便宜上、CREATE [PUBLIC] SYNONYM 文を使用して外部プロシージャのシノニムを作成できます。次の例では、すべてのユーザーがアクセス可能なパブリック・シノニムを DBA が作成します。PUBLIC を指定しない場合、シノニムはプライベートになり、そのスキーマ内以外ではアクセスできません。

```
CREATE PUBLIC SYNONYM Rfac FOR Scott.RecursiveFactorial;
```

CALL 文の構文

外部プロシージャは、SQL の CALL 文を使用して起動します。CALL 文は SQL*Plus から対話形式で実行できます。構文は次のとおりです。

```
CALL [schema.][{object_type_name | package_name}]procedure_name[@dblink_name]
```

```
   [(parameter_list)] [INTO :host_variable][INDICATOR][:indicator_variable];
```

これは、SELECT foo(...) FROM dual という形式のプロシージャ foo() の実行と基本的には同じです。ただし、この場合は、SELECT の実行に関連するオーバーヘッドが発生しません。

たとえば、この章で発行した plsToC_demoExternal_proc を動的 SQL を使用してコールする無名 PL/SQL ブロックを次に示します。PL/SQL は、外部 C プロシージャ C_demoExternal_proc に 3 つのパラメータを渡します。

```
DECLARE
   xx NUMBER(4);
   yy VARCHAR2(10);
   zz DATE;
 BEGIN
    EXECUTE IMMEDIATE 'CALL plsToC_demoExternal_proc(:xxx, :yyy, :zzz)' USING
xx,yy,zz;
 END;
```

CALL 文のセマンティクスは、同等の BEGIN..END ブロックと同じです。

> **注意：** CALL は、それ自体では PL/SQL の BEGIN...END ブロック内に入れることができない唯一の SQL 文です。BEGIN...END ブロック内の EXECUTE IMMEDIATE 文の一部にすることはできます。

Java クラス・メソッドのコール

以前に発行された J_calcFactorial クラス・メソッドのコール方法を次に示します。まず、次のように、SQL*Plus のホスト変数を 2 つ宣言して初期化します。

```
VARIABLE x NUMBER
VARIABLE y NUMBER
EXECUTE :x := 5;
```

ここで、J_calcFactorial をコールします。

```
CALL J_calcFactorial(:x) INTO :y;
PRINT y
```

結果は次のようになります。

```
Y
------
   120
```

参照： 『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

データベース・サーバーによる外部 C プロシージャのコール方法

外部 C プロシージャをコールするには、PL/SQL エンジンが適切な DLL パスを見つける必要があります。PL/SQL エンジンは、プロシージャ宣言の AS LANGUAGE 句からのライブラリ別名に基づいて、パスをデータ・ディクショナリから取得します。

次に、PL/SQL はリスナー・プロセスにシグナルを発行し、リスナー・プロセスがセッション固有のエージェントを起動します。デフォルトでは、このエージェントを extproc といいますが、listener.ora ファイルで別の名前で指定できます。リスナーは接続をエージェントに渡し、PL/SQL は DLL の名前、外部プロシージャの名前およびパラメータをエージェントに渡します。

次に、エージェントは DLL をロードし、外部プロシージャを実行します。また、エージェントは（例外を呼び出すなどの）サービス・コールおよび Oracle サーバーへのコールバックも処理します。最後に、エージェントは、外部プロシージャによって戻された値を PL/SQL に渡します。

注意： DLL のキャッシュはいくらか発生しますが、DLL がキャッシュ内に残るという保証はありません。したがって、グローバル変数は DLL には格納しないでください。

外部プロシージャが完了した後、エージェントは Oracle セッションの終わりまでアクティブなまま残ります。ログオフするとエージェントが消去されます。この結果、何度コールしても、エージェントを起動するのは 1 回で済みますが、演算によるメリットがコールのコストを上回るときにのみ外部プロシージャをコールするようにします。

ご使用のデータベース・サーバーとは異なるマシン上で、エージェントを実行できます。詳細は、10-6 ページの「外部 C プロシージャのロード」を参照してください。

ここでは、この章で発行した PL/SQL ファンクション plsCallsCdivisor_func を無名ブロックからコールします。PL/SQL は整数パラメータを 2 つ外部関数 Cdivisor_func に渡し、この関数が最大公約数を戻します。

```
DECLARE
   g    BINARY_INTEGER;
   a    BINARY_INTEGER;
   b    BINARY_INTEGER;
CALL plsCallsCdivisor_func(a, b);
IF g IN (2,4,8) THEN ...
```

複数言語のプログラム・エラーおよび例外処理

一般的なコンパイル時のコール仕様エラー

PL/SQL コンパイラは、次の条件が構文内で検出された場合に、コンパイル時エラーを呼び出します。

- AS EXTERNAL コール仕様が TYPE 仕様部または PACKAGE 仕様部で見つかった場合

Java の例外処理

参照：『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

C の例外処理

C プログラムは、OCIExtproc... 関数を介して例外を呼び出すことができます。

外部 C プロシージャでのサービス・プロシージャの使用

サービス・ルーチンが外部プロシージャからコールされると、サービス・プロシージャは例外を呼び出し、メモリーを割り当て、サーバーへのコールバック用の OCI ハンドルを起動します。サービス・ルーチンを使用するには、WITH CONTEXT 句を指定する必要があります。WITH CONTEXT 句を使用すると、コンテキスト構造体を外部プロシージャに渡すことができます。コンテキスト構造体は、ヘッダー・ファイル ociextp.h の中に次のように宣言されています。

```
typedef struct OCIExtProcContext OCIExtProcContext;
```

注意： ociextp.h は、UNIX では $ORACLE_HOME/plsql/public にあります。

OCIExtProcAllocCallMemory

このサービス・ルーチンは、外部プロシージャ・コールの間、*n* バイトのメモリーを割り当てます。この関数によって割り当てられるメモリーは、PL/SQL に制御が戻った直後に自動的に解放されます。

> **注意：** 外部プロシージャでは、このサービス・ルーチンによって割り当てられたメモリーが自動的に処理されるため、C 関数の free () をコールする必要はありません（コールしないでください）。

この関数の C プロトタイプは、次のとおりです。

```
dvoid *OCIExtProcAllocCallMemory(
   OCIExtProcContext *with_context,
   size_t amount);
```

パラメータ with_context および amount は、それぞれ、コンテキスト・ポインタと割り当てられるバイト数です。この関数は、割り当てられたメモリーを指す型が未定のポインタを戻します。戻り値が 0（ゼロ）の場合は、失敗を示します。

SQL*Plus で、外部関数 plsToC_concat_func を次のように発行するとします。

> **注意：** 次のようなデータ構造を設定しないと機能しない例もあります。
>
> ```
> CONNECT system/manager
> DROP USER y CASCADE;
> GRANT CONNECT,RESOURCE,CREATE LIBRARY TO y IDENTIFIED BY y;
> CONNECT y/y
> CREATE LIBRARY stringlib AS
> '/private/varora/ilmswork/Cexamples/john2.so';
> ```

```
CREATE OR REPLACE FUNCTION plsToC_concat_func (
   str1 IN VARCHAR2,
   str2 IN VARCHAR2)
RETURN VARCHAR2 AS LANGUAGE C
NAME "concat"
LIBRARY stringlib
WITH CONTEXT
PARAMETERS (
CONTEXT,
str1   STRING,
str1   INDICATOR short,
str2   STRING,
str2   INDICATOR short,
RETURN INDICATOR short,
RETURN LENGTH short,
RETURN STRING);
```

C_concat は、コールされたときに 2 つの文字列を連結し、結果を戻します。

```
select plsToC_concat_func('hello ', 'world') from dual;
```

```
PLSTOC_CONCAT_FUNC('HELLO','WORLD')
--------------------------------------------------------------------------------
hello world
```

いずれの文字列も NULL の場合は、結果も NULL になります。次の例で示されるように、C_concat は OCIExtProcAllocCallMemory を使用して結果文字列のメモリーを割り当てます。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#include <ociextp.h>

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}
```

```
#ifdef LATER
static void checkerr (/*_ OCIError *errhp, sword status _*/);

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

char *concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l)
OCIExtProcContext *ctx;
char   *str1;
short  str1_i;
```

```
char   *str2;
short  str2_i;
short  *ret_i;
short  *ret_l;
{
  char *tmp;
  short len;
  /* Check for null inputs. */
  if ((str1_i == OCI_IND_NULL) || (str2_i == OCI_IND_NULL))
  {
      *ret_i = (short)OCI_IND_NULL;
      /* PL/SQL has no notion of a NULL ptr, so return a zero-byte string. */
      tmp = OCIExtProcAllocCallMemory(ctx, 1);
      tmp[0] = '\0';
      return(tmp);
  }
  /* Allocate memory for result string, including NULL terminator. */
  len = strlen(str1) + strlen(str2);
  tmp = OCIExtProcAllocCallMemory(ctx, len + 1);

  strcpy(tmp, str1);
  strcat(tmp, str2);

  /* Set NULL indicator and length. */
  *ret_i = (short)OCI_IND_NOTNULL;
  *ret_l = len;
  /* Return pointer, which PL/SQL frees later. */
  return(tmp);
}

/*============================================================================*/
int main(char *argv, int argc)
{
  OCIExtProcContext *ctx;
  char           *str1;
  short          str1_i;
  char           *str2;
  short          str2_i;
  short          *ret_i;
  short          *ret_l;
  /* OCI Handles */
  OCIEnv        *envhp;
  OCIServer     *srvhp;
  OCISvcCtx     *svchp;
  OCIError      *errhp;
  OCISession    *authp;
  OCIStmt       *stmthp;
```

```
OCILobLocator *clob, *blob;
OCILobLocator *Lob_loc;

/* Initialize and Logon */
(void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 );

(void) OCIEnvInit( (OCIEnv **) &envhp,
                   OCI_DEFAULT, (size_t) 0,
                    (dvoid **) 0 );

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                  (size_t) 0, (dvoid **) 0);

/* Server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                  (size_t) 0, (dvoid **) 0);

/* Service context */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                  (size_t) 0, (dvoid **) 0);

/* Attach to Oracle */
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);

/* Set attribute server context in the service context */
(void) OCIAttrSet ((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                    (dvoid *)srvhp, (ub4) 0,
                   OCI_ATTR_SERVER, (OCIError *) errhp);

(void) OCIHandleAlloc((dvoid *) envhp,
                      (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                      (size_t) 0, (dvoid **) 0);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4)4,
                (ub4) OCI_ATTR_USERNAME, errhp);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) "samp", (ub4) 4,
                (ub4) OCI_ATTR_PASSWORD, errhp);

/* Begin a User Session */
checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                        (ub4) OCI_DEFAULT));
```

```
  (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                    (dvoid *) authp, (ub4) 0,
                    (ub4) OCI_ATTR_SESSION, errhp);

  /* ----------------------User Logged In-----------------------------*/
  printf ("user logged in \n");

  /* allocate a statement handle */
  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
           OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &Lob_loc,
                                     (ub4) OCI_DTYPE_LOB,
                                     (size_t) 0, (dvoid **) 0));

  /* ------- subroutine called  here----------------------*/
  printf ("calling concat...\n");
  concat(ctx, str1, str1_i, str2, str2_i, ret_i, ret_l);

  return 0;
}

#endif
```

OCIExtProcRaiseExcp

このサービス・ルーチンは、事前定義の例外を呼び出します。この例外には、1 ～ 32767 の有効な Oracle エラー番号が含まれている必要があります。クリーンアップ処理後、外部プロシージャはすぐにリターンする必要があります（OUT または IN OUT パラメータには値は何も割り当てられません）。この関数の C プロトタイプは、次のとおりです。

```
int OCIExtProcRaiseExcp(
   OCIExtProcContext *with_context,
   size_t errnum);
```

パラメータ with_context および error_number は、それぞれ、コンテキスト・ポインタおよび Oracle エラー番号です。戻り値の OCIEXTPROC_SUCCESS および OCIEXTPROC_ERROR は、正常終了および失敗を示します。

SQL*Plus で、外部プロシージャ plsTo_divide_proc を次のように発行するとします。

```
CREATE OR REPLACE PROCEDURE plsTo_divide_proc (
   dividend IN BINARY_INTEGER,
   divisor  IN BINARY_INTEGER,
   result   OUT FLOAT)
AS LANGUAGE C
   NAME "C_divide"
```

```
   LIBRARY MathLib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      dividend INT,
      divisor  INT,
      result   FLOAT);
```

C_divide は、コールされたときに 2 つの数値の比率を計算します。次の例で示されるように、除数が 0（ゼロ）の場合、C_divide は OCIExtProcRaiseExcp を使用して事前定義の例外 ZERO_DIVIDE を次のように呼び出します。

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
{
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise exception ZERO_DIVIDE, which is Oracle error 1476. */
    if (OCIExtProcRaiseExcp(ctx, (int)1476) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /* Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = (float)dividend / (float)divisor;
}
```

OCIExtProcRaiseExcpWithMsg

このサービス・ルーチンはユーザー定義例外を呼び出し、ユーザー定義エラー・メッセージを戻します。この関数の C プロトタイプは、次のとおりです。

```
int OCIExtProcRaiseExcpWithMsg(
   OCIExtProcContext *with_context,
   size_t  error_number,
   text   *error_message,
   size_t  len);
```

パラメータ with_context、error_number、error_message は、それぞれ、コンテキスト・ポインタ、Oracle エラー番号、エラー・メッセージ・テキストです。パラメータ len

は、エラー・メッセージの長さを格納します。メッセージが NULL で終了する文字列の場合、len は 0（ゼロ）です。戻り値の OCIEXTPROC_SUCCESS および OCIEXTPROC_ERROR は、正常終了および失敗を示します。

この前の例では、外部プロシージャ plsTo_divide_proc を発行しました。次の例では、別の実装方法を使用します。ここでは、除数が 0（ゼロ）の場合、C_divide は OCIExtProcRaiseExcpWithMsg を使用してユーザー定義例外を呼び出します。

```
void C_divide (ctx, dividend, divisor, result)
OCIExtProcContext *ctx;
int    dividend;
int    divisor;
float  *result;
  /* Check for zero divisor. */
  if (divisor == (int)0)
  {
    /* Raise a user-defined exception, which is Oracle error 20100,
       and return a null-terminated error message. */
    if (OCIExtProcRaiseExcpWithMsg(ctx, (int)20100,
          "divisor is zero", 0) == OCIEXTPROC_SUCCESS)
    {
      return;
    }
    else
    {
      /*  Incorrect parameters were passed. */
      assert(0);
    }
  }
  *result = dividend / divisor;

}
```

外部 C プロシージャを使用したコールバックの実行

OCIExtProcGetEnv

このサービス・ルーチンを使用すると、外部プロシージャ・コール中にデータベースへの OCI コールバックが可能になります。このルーチンはコールバックのみに使用されます。さらに、これは、使用される唯一のコールバック・ルーチンです。この関数によって取得された OCI ハンドルを使用する場合は、ハンドルがデータベースへの新しい接続を確立するため、同一トランザクション内でのコールバックには使用できません。つまり、外部プロシージャ・コール中は、OCI ハンドルはコールバック用か新しい接続用のどちらかとして使用できますが、両方の目的では使用できません。

この関数の C プロトタイプは、次のとおりです。

```
sword OCIExtProcGetEnv ( OCIExtProcContext *with_context,
   OCIEnv envh,
   OCISvcCtx svch,
   OCIError errh )
```

パラメータ with_context はコンテキスト・ポインタで、パラメータ envh、svch、errh は、それぞれ、OCI 環境、サービス、エラー・ハンドルです。戻り値の OCIEXTPROC_SUCCESS および OCIEXTPROC_ERROR は、正常終了および失敗を示します。

外部 C プロシージャおよび Java クラス・メソッドでは、どちらもデータベースをコールバックして SQL 操作を実行できます。実際の例は、10-48 ページの「デモ・プログラム」を参照してください。

> **参照：** Java の例外については、『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

注意： コールバックは、必ずしも同一セッションで発生するとはかぎりません。OCIlogon を使用して、別セッション内で SQL 文を実行することができます。

Oracle サーバー上で実行される外部 C プロシージャは、サービス・ルーチンをコールして OCI 環境およびサービス・ハンドルを取得できます。OCI を使用すると、SQL 文および PL/SQL サブプログラムの実行、データのフェッチ、および LOB の操作のためにコールバックを使用することができます。

SQL*Plus で、次のスクリプトを実行するとします。

```
CREATE TABLE Emp_tab (empno NUMBER(10))

CREATE PROCEDURE plsToC_insertIntoEmpTab_proc (
   empno BINARY_INTEGER)
AS LANGUAGE C
   NAME "C_insertEmpTab"
   LIBRARY insert_lib
   WITH CONTEXT
   PARAMETERS (
      CONTEXT,
      empno LONG);
```

後で、次のように外部プロシージャ plsToC_insertIntoEmpTab_proc からサービス・ルーチン OCIExtProcGetEnv をコールできます。

```
#include <stdio.h>
#include <stdlib.h>
#include <oratypes.h>
#include <oci.h>   /* includes ociextp.h */
```

```
...
void C_insertIntoEmpTab (ctx, empno)
OCIExtProcContext *ctx;
long empno;
{
  OCIEnv    *envhp;
  OCISvcCtx *svchp;
  OCIError  *errhp;
  int        err;
  ...
  err = OCIExtProcGetEnv(ctx, &envhp, &svchp, &errhp);
  ...
}
```

コールバックを使用しない場合は、oci.h を含める必要はなく、ociextp.h のみを含めます。

OCI コールバック用のオブジェクト・サポート

外部プロシージャからオブジェクト関連コールバックを実行するために、extproc エージェント内の OCI 環境がオブジェクト・モードで完全に初期化されています。この環境へのハンドルは、OCIExtProcGetEnv() プロシージャを使用して取得します。

オブジェクト・ランタイム環境を使用すると、静的サポートのみでなく、OCI によって提供されている動的なオブジェクト・サポートも使用できます。静的サポートを利用するには、OTT を使用して該当するオブジェクト型用の C 構造体を生成し、次に、従来の C コードを使用してオブジェクトの属性にアクセスします。

外部プロシージャの作成時に型がわからないオブジェクトについては、かわりの動的なオブジェクト・アクセス方法によって、OCIDescribeAny() が起動されてその型の属性およびメソッド情報が取得されます。その後、OCIObjectGetAttr() および OCIObjectSetAttr() をコールして、属性値を取得して設定します。

現在の外部プロシージャ・モデルはステートレスのため、コールバックを実行するかまたは OCIExtProc...() サービス・ルーチンを起動する外部プロシージャ内では、OCIExtProcGetEnv() をコールする必要があります。外部プロシージャが起動されるたびに、起動後にコールバック機構がクリーンアップされ、OCI ハンドルが解放されます。

コールバックに関する制限事項

コールバックでは、次の SQL コマンドおよび OCI プロシージャはサポートされていません。

- COMMIT などのトランザクション制御コマンド
- CREATE などのデータ定義コマンド

- 次のオブジェクト指向 OCI プロシージャ

```
OCIObjectNew
OCIObjectPin
OCIObjectUnpin
OCIObjectPinCountReset
OCIObjectLock
OCIObjectMarkUpdate
OCIObjectUnmark
OCIObjectUnmarkByRef
OCIObjectAlwaysLatest
OCIObjectNotAlwaysLatest
OCIObjectMarkDeleteByRef
OCIObjectMarkDelete
OCIObjectFlush
OCIObjectFlushRefresh
OCIObjectGetTypeRef
OCIObjectGetObjectRef
OCIObjectExists
OCIObjectIsLocked
OCIObjectIsDirtied
OCIObjectIsLoaded
OCIObjectRefresh
OCIObjectPinTable
OCIObjectArrayPin
OCICacheFlush,
OCICacheFlushRefresh,
OCICacheRefresh
OCICacheUnpin
OCICacheFree
OCICacheUnmark
OCICacheGetObjects
OCICacheRegister
```

- `OCIGetPieceInfo` などのポーリング・モード OCI プロシージャ
- 次の OCI プロシージャ

```
OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIServerAttach
OCIServerDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
```

```
OCITransRollback
OCITransStart
```

また、OCI プロシージャ OCIHandleAlloc では、次のハンドル・タイプはサポートされていません。

```
OCI_HTYPE_SERVER
OCI_HTYPE_SESSION
OCI_HTYPE_SVCCTX
OCI_HTYPE_TRANS
```

外部プロシージャのデバッグ

参照：『Oracle9*i* Java Stored Procedures Developer's Guide』を参照してください。

外部プロシージャが失敗する場合は、通常、そのプロトタイプに不具合があります。プロトタイプが、PL/SQL によって内部的に生成されたプロトタイプと一致しないということです。これは、互換性のない C データ型を指定した場合に発生する可能性があります。たとえば、型が REAL の OUT パラメータを渡すために float* を指定したとします。float、double * または他の C データ型を指定すると、結果が不一致になります。

このような場合は、次のエラーを受け取ることがあります。

```
lost RPC connection to external routine agent
```

このエラーは、外部プロシージャによってコア・ダンプが発生したため、エージェント extproc が異常終了したことを表します。C プロトタイプ・パラメータの宣言時にエラーを回避するには、前述の表を参照してください。

パッケージ DEBUG_EXTPROC の使用

PL/SQL では、外部プロシージャのデバッグを支援する目的でユーティリティ・パッケージ DEBUG_EXTPROC が提供されています。このパッケージをインストールするには、PL/SQL デモ・ディレクトリにあるスクリプト dbgextp.sql を実行します（ディレクトリの場所については、ご使用の Oracle のインストール・ガイドまたはユーザーズ・ガイドを参照してください）。

このパッケージを使用するには、dbgextp.sql の指示に従ってください。Oracle アカウントに、パッケージに対する EXECUTE 権限および CREATE LIBRARY 権限が必要です。

注意： DEBUG_EXTPROC は、実行中のプロセスにアタッチできるデバッガ付きのプラットフォームでのみ動作します。

デモ・プログラム

PL/SQL デモ・ディレクトリには、外部プロシージャのコール方法を示すスクリプト extproc.sql もあります。付属ファイル extproc.c には、外部プロシージャ用の C ソース・コードが含まれています。

デモを実行するには、extproc.sql にある指示に従ってください。SCOTT/TIGER アカウントを使用する必要があり、このアカウントには CREATE LIBRARY 権限が必要です。

外部 C プロシージャのためのガイドライン

グローバル変数の操作

グローバル変数は関数の外で宣言され、その値はプログラムのすべての関数によって共有されます。外部プロシージャの場合、これは、DLL 内のすべての関数がグローバル変数の値を共有するということです。グローバル変数の使用は、次の 2 つの理由のため、お薦めしません。

- **スレッド化**: 現在、エージェント・プロセスは非スレッド構成になっているため、一度にアクティブになる関数は 1 つのみです。ただし、将来は、エージェント・プロセスがスレッド化される可能性があります。これは、複数の関数を同時にアクティブにできるということです。その場合は、2 つ以上の関数が同時にグローバル変数をアクセスしようとして、正常な結果にはならない可能性があります。
- **DLL のキャッシング**: 関数の存続期間を超えて存続するデータの格納にもグローバル変数が使用されます。たとえば、2 つの関数 *func1*() および *func2*() が相互にデータを受け渡すとします。DLL キャッシュ機能によって、*func1*() の完了後 DLL はアンロードされ、この結果、グローバル変数のすべての値が失われる可能性があります。*func2*() が実行されるときに、DLL は再ロードされ、グローバル変数はすべて 0（ゼロ）に初期化されます。これは *func1*() の完了時の値と整合しません。

静的変数の操作

静的変数には外部変数および内部変数という 2 種類があります。外部静的変数は、グローバル変数の特殊な場合で、その使用は前述の 2 つの理由によってお薦めしません。内部静的変数は、特定の関数に対してローカルな変数ですが、関数がアクティブになるたびに生成、消滅するのではなく、存在したまま残ります。したがって、この変数は 1 つの関数内にプライベートで永続的な記憶域を提供します。このような変数は、同一関数を後で起動するときにデータを渡すために使用します。ただし、DLL には前述のキャッシュ機能があるため、DLL が起動ごとにロードおよび再ロードされる可能性があります。これは、内部静的変数がその値を失う可能性があることを表します。

> **参照：** 動的リンク・ライブラリの作成の詳細は、RDBMS サブディレクトリ *public* を参照してください。この中に、テンプレート *makefile* があります。

コール仕様および CALL 文のガイドライン

外部プロシージャをコールするときは、次のことに注意してください。

- IN パラメータには書込みを行わないでください。OUT パラメータの容量をオーバーフローさせないようにしてください（PL/SQL では、実行時にこの 2 つのエラー状態はチェックしません）。
- OUT パラメータまたは関数結果を読み込まないでください。
- IN OUT パラメータと OUT パラメータおよび関数結果には、必ず値を代入してください。そうしないと、外部プロシージャが正常に戻りません。
- WITH CONTEXT および PARAMETERS 句を含める場合は、パラメータ・リスト内でのコンテキスト・ポインタの位置を示すパラメータ CONTEXT を指定する必要があります。
- PARAMETERS 句を含めたときに、外部プロシージャが関数の場合は、パラメータ RETURN を最後の位置に指定する必要があります。
- 各仮パラメータには、PARAMETERS 句に対応するパラメータが必要です。また、PARAMETERS 句のパラメータのデータ型が、C プロトタイプのデータ型と互換性があることも確認します。暗黙的変換が実行されないためです。
- 型が RAW または LONG RAW のパラメータの場合は、プロパティ LENGTH を使用する必要があります。また、そのパラメータが IN OUT または OUT および NULL の場合は、対応する C パラメータの長さを 0（ゼロ）に設定する必要があります。

外部 C プロシージャに関する制限事項

現在、外部プロシージャには次の制限が適用されます。

- この機能は、DLL をサポートするプラットフォームのみで使用可能です。
- C プロシージャまたは C からコール可能なプロシージャのみがサポートされます。
- PL/SQL カーソル変数またはレコードを外部プロシージャに渡すことはできません。レコードについては、かわりにオブジェクト型のインスタンスを使用できます。
- LIBRARY 副次句には、リモート・ライブラリを指定するデータベース・リンクは使用できません。
- 外部プロシージャに渡すことができるパラメータの最大数は 128 です。ただし、FLOAT 型または DOUBLE 型のパラメータを値渡しする場合は、最大数は 127 以下になります。どのくらい減るかは、パラメータの個数およびオペレーティング・システムによって異なります。目安としては、値渡しされる FLOAT 型または DOUBLE 型のパラメータ 1 つを 2 つ分として数えます。

第 III 部

アプリケーション・セキュリティ

第 III 部に含まれる章は、次のとおりです。

11

アプリケーション開発者のためのデータベース・セキュリティの概要

この章では、アプリケーションおよびデータベースのセキュリティ・ポリシーの基本的な概念について説明します。内容は次のとおりです。

- データベース・セキュリティ・ポリシーの概要
- アプリケーション・セキュリティ・ポリシーの概要

参照： 『Oracle9*i* セキュリティ概要』を参照してください。

データベース・セキュリティ・ポリシーの概要

この項では、セキュリティ・ポリシーについて簡単に説明します。内容は次のとおりです。

セキュリティの侵害および対策

組織は、セキュリティ・ポリシーを文書で作成し、防ぐべきセキュリティへの侵害、およびこの侵害に対して組織が取るべき対策を列挙する必要があります。セキュリティの侵害に対して、次のような対策を取ることができます。

- 手続き上の対策（データ・センターの社員にセキュリティ・バッジを提示させるなど）
- 人事的対策（経歴の確認や主要な社員の調査など）
- 物理的対策（アクセス制限された施設でのコンピュータの保護など）
- 技術的対策（重要な業務システムに対する厳密認証要件の実装など）

セキュリティの侵害に対して適切な対策が、手続き上の対策、物理的対策、技術的対策または人事的対策か、あるいはこれらの対策を組み合せたものかどうかを判断してください。

たとえば、可能性のあるセキュリティの侵害として、不当な人間がコンピュータを故意に破損させ、重要な業務システムが破壊されることなどが考えられます。この侵害に対する物理的対策は、施錠できる施設内に重要なビジネス・コンピュータを設置することです。手続き上の対策は、定期的にシステムのバックアップを取ることです。人事的対策は、重要な業務システムにアクセスまたは管理する従業員の経歴を確認することです。

Oracle9*i* では、高度なセキュリティ・ポリシーの技術的対策を実装する多くのメカニズムを提供しています。

情報セキュリティ・ポリシーで対処できる事項

ご使用の環境に固有の要件以外に、次の重要な問題を解決するための情報セキュリティ・ポリシーを設計および実装してください。

- アプリケーション・レベルでのセキュリティのレベル
- システムおよびオブジェクトの権限
- データベース・ロール
- エンタープライズ・ロール
- 権限とロールの付与方法および取消し方法
- ロールの作成、変更および削除方法

- ロール使用の制御方法
- アクセス制御の細分性のレベル
- データベースへのアクセスを決定するユーザー属性
- 暗号化の使用または不使用
- 3 層アプリケーションにおけるセキュリティの実装方法

セキュリティ・ポリシーの設定に使用する機能

次に示す Oracle9*i* の要素を使用すると、セキュリティの技術的問題を解決できます。

Oracle 機能	推奨使用
アプリケーション・セキュリティ	各アプリケーションに権限およびロールを連結して、ユーザーがアプリケーションを使用していないときに、そのアプリケーションに連結された権限およびロールが間違って使用されないようにします。
ファイングレイン・アクセス・コントロール	セキュリティ・ポリシーを高水準の細分性で実装します。たとえば、行レベルのセキュリティを施行できます。これは、アプリケーションで使用される表、ビューまたはシノニムに連結するセキュリティ・ポリシー関数を作成することによって行います。こうすることによって、ユーザーがそのオブジェクトに対して DML 文を入力したときに、Oracle はその文を動的かつユーザーに透過的に変更します。
アプリケーション・コンテキスト	セッション・ベースの属性を確実に設定します。たとえば、ユーザー名、従業員番号、職制上の位置付け（職位）などのユーザー属性を確実に格納できます。その情報を後でセッション内で取得し、ファイングレイン・アクセス・コントロールに使用できます。
保護アプリケーション・ロール	ユーザーが定義した基準に基づいてロールを使用します。たとえば、特定の IP アドレスから接続するユーザー、または特定の中間層を介してデータベースにアクセスするユーザーのみにロールの使用を許可することができます。
ファイングレイン監査	内容に基づいて問合せアクセスを監視します。たとえば、表内の特定の行にアクセスするユーザーを監視できます。また、データの間違った使用を検出したり、侵入検出システムとしても使用できます。

Oracle 機能	推奨使用
Oracle Label Security	この Oracle9*i* データ・サーバー・オプションは、ファイングレイン・アクセス・コントロールおよびラベル・ベースのアクセス・コントロールを自動的に施行します。たとえば、データを「社外秘」または「パートナへ公開可能」とラベル付けして、そのデータのラベル、およびユーザーがアクセスを許可されているデータのラベルに基づいて、データへのアクセスを自動的に制限できます。Oracle Label Security を使用すると、多くの場合は追加のプログラミングを行わずに、組織ごとにファイングレイン・アクセス・コントロールおよびラベル・ベースのアクセス・コントロールを素早く実装できます。
プロキシ認証	個別のデータベース接続のオーバーヘッドなしで、中間層からデータベースの間でユーザー ID を保持します。パスワードや X.509 証明書などのユーザー ID および資格証明を、中間層からデータベースの間にプロキシ化できます。また、ユーザーのかわりに接続の監査をサポートします。
データの暗号化	セキュリティの特別な対策として情報を暗号化します。

参照： 『Oracle9*i* セキュリティ概要』を参照してください。

リスク削減のための推奨アプリケーション設計の実行

潜在的な問題を回避するために、データベース・ロールの実装時には、次の処理を行うことをお薦めします。詳細は、各項を参照してください。

- ヒント 1: ロールの即時使用可能および使用禁止
- ヒント 2: ストアド・プロシージャでの権限のカプセル化
- ヒント 3: ユーザーが知らないロール・パスワードの使用
- ヒント 4: プロキシ認証および保護アプリケーション・ロールの使用
- ヒント 5: 保護アプリケーション・ロールを使用した IP アドレスの検証
- ヒント 6: アプリケーション・コンテキストおよびファイングレイン・アクセス・コントロールの使用

ヒント 1: ロールの即時使用可能および使用禁止

アプリケーションが起動したときに適切なロールを使用可能にし、アプリケーションが終了したときにそのロールを使用禁止にします。これを行うには、次のように設定する必要があります。

- 各アプリケーションには明確に区別したロールを付与し、1 つのロールにアプリケーションの使用に必要なすべての権限を含めます。状況に応じて、アプリケーションの実行中により厳密なまたはより緩やかなセキュリティを提供するために、様々な権限を含むロールをいくつか設定する場合があります。各データベース・ロールは、不当な使用を防ぐために、パスワード（またはオペレーティング・システム認証）によって保護する必要があります。

  別のロールには、そのアプリケーションに対応付けられている非破損権限（アプリケーションに対応付けられている特定の表またはビューに対する SELECT 権限）のみを含めます。読込み専用ロールによって、アプリケーション・ユーザーは、SQL*Plus などの非定型ツールを使用してカスタム・レポートを生成できます。ただし、このロールを使用して、アプリケーション・ユーザーがアプリケーション外にある表データを更新することはできません。非定型の問合せツール用に設計されたロールは、パスワード（またはオペレーティング・システム認証）によって保護される場合と保護されない場合があります。

- 起動時に、各アプリケーションは SET ROLE 文を使用して、そのアプリケーションに対応付けられているデータベース・ロールの 1 つを使用可能にする必要があります。パスワードがロールを認証するために使用されている場合、そのパスワードをアプリケーション内の SET ROLE 文に含める必要があります（可能な場合は、アプリケーションによって暗号化されます）。ロールがオペレーティング・システムによって認証されている場合、アプリケーションを使用するときにアプリケーション・ユーザーが適切なオペレーティング・システム権限を取得できるように、システム管理者がユーザー・アカウントおよびアプリケーションを設定しておく必要があります。
- 終了時に、各アプリケーションは、使用可能にしていたデータベース・ロールを使用禁止にする必要があります。
- アプリケーション・ユーザーには、必要に応じてデータベース・ロールを付与する必要があります。

注意： ユーザーに付与されているデータベース・ロールは、アプリケーション外では、ユーザーによって使用可能にできます。このような使用は、アプリケーション・ベースのセキュリティによって制御されません。この問題を解決するには、仮想プライベート・データベース（VPD）を使用します。また、3 層システムでは、保護アプリケーション・ロールを使用することによって、ユーザーがアプリケーション外でロールを使用することを制限できます。

さらに次の処理が可能です。

- PRODUCT_USER_PROFILE 表を使用して、ユーザーが SQL*Plus を起動するときに使用可能にするロールを指定します。これは、SET ROLE 文を発行して、アプリケーションの起動時に特定のロールを使用可能にするプリコンパイラまたは Oracle Call Interface（OCI）アプリケーションと同様の機能です。
- PRODUCT_USER_PROFILE 表を使用して、SQL*Plus ユーザーに対して SET ROLE 文を使用禁止にします。これによって、SQL*Plus ユーザーには、SQL*Plus を起動するときに使用可能にされるロールに対応付けられている権限のみが許可されます。

  他の非定型の問合せツールおよびレポート作成ツールでも、PRODUCT_USER_PROFILE 表を使用して、そのツールを実行中に各ユーザーが使用できるロールおよびコマンドを制限できます。

  参照：

  - 12-7 ページの「ファイングレイン・アクセス・コントロールでのアプリケーション・コンテキストの使用方法」を参照してください。
  - 11-14 ページの「保護アプリケーション・ロールを使用したデータベース接続の保証」を参照してください。
  - 該当するツールのマニュアル（『SQL*Plus ユーザーズ・ガイドおよびリファレンス』など）を参照してください。

ヒント 2: ストアド・プロシージャでの権限のカプセル化

ユーザーが、非定型の問合せツールによってアプリケーション権限の使用を制限する別の方法として、権限をストアド・プロシージャにカプセル化する方法があります。ユーザーに権限を直接付与するのではなく、プロシージャに対する実行権限をユーザーに付与します。このようにして、その権限が論理を持つことになります。

これによって、ユーザーが権限を使用できるのは、適切に作成されたビジネス・アプリケーションのコンテキスト内のみになります。たとえば、ユーザーに対して表の更新を許可する場合、表を直接更新するのではなく、ストアド・プロシージャの実行によって更新するようにします。こうすることにより、SELECT 権限を持つユーザーがアプリケーション外で権限を使用するという問題がなくなります。

参照： 12-20 ページの「例 3: イベント・トリガー、アプリケーション・コンテキスト、ファイングレイン・アクセス・コントロールおよび権限のカプセル化」を参照してください。

ヒント 3: ユーザーが知らないロール・パスワードの使用

ユーザーが知らないパスワードを必要とするロールを介して権限を付与します。

ユーザーがアプリケーション内のみで使用する権限がある場合、ロールの作成者のみが知っているパスワードによって、ロールを使用可能にできます。アプリケーションを使用して

SET ROLE 文を発行します。ユーザーはパスワードを知らないため、パスワードをアプリケーション内に埋め込むか、またはストアド・プロシージャを使用してロール・パスワードをデータベース表から取得する必要があります。この方法によって、ユーザーがアプリケーションの使用を避けることはなくなります。ただし、この方法はアプリケーション・セキュリティを向上させますが、確実ではありません。

アプリケーション・コードにアクセスするユーザーが、アプリケーションに埋め込まれたパスワードを検出する可能性があります。この不明瞭なセキュリティでは、セキュリティが適切に施行されません。パスワードをアプリケーションに埋め込むと、アプリケーションを回避するユーザー（一般的なユーザー）からは保護されます。データにアクセスし、アプリケーションを回避して、権限を故意に間違えて使用するユーザー（不正なユーザー）からは保護されません。クライアント・コードを逆コンパイルすると、埋込みパスワードを元の状態に戻せるため、埋込みパスワードは一般的なユーザーから保護する場合にのみ使用します。

ストアド・プロシージャを使用してデータベース表からロール・パスワードを取得するには、EXECUTE 権限が必要です。ユーザーは、権限を取得してから、プロシージャを実行してパスワードを取得し、アプリケーション外でロールを使用します。

ヒント 4: プロキシ認証および保護アプリケーション・ロールの使用

3 層システムでは、ユーザーが中間層アプリケーションを介してデータベースにアクセスする場合にのみロールを使用可能にできます。これには、プロキシ認証および保護アプリケーション・ロールを使用する必要があります。プロキシ認証では、ユーザーにかわってセッションを作成する中間層と、直接接続するユーザーが識別されます。プロキシ・ユーザー（中間層）および実際のユーザーの情報は、ユーザー・セッション内に獲得されます。保護アプリケーション・ロールは、パッケージによって実装され、ユーザーへのそのロールでの権限の付与を許可する前に、必要な妥当性チェックを行うことができます。アプリケーションがプロキシ認証を使用する場合、保護アプリケーション・ロールは、ユーザー・セッションがプロキシによって作成されているかどうか、およびユーザーが直接ではなくアプリケーションを介してデータベースに接続しているかどうかを検証できます。

HR 管理ロールの使用を、中間層 HRSERVER を介して（プロキシによって）データベースにアクセスするユーザーに制限する場合を考えます。次の保護アクセス・ロールを作成できます。

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

ここで、hr.admin は必要な妥当性チェックを実行するパッケージです。このパッケージは、ユーザーが SYS_CONTEXT ('userenv', 'proxy_userid') または SYS_CONTEXT ('userenv', 'proxy_user') を使用してプロキシによって接続されているかどうか、または両方がプロキシ・ユーザー（この場合は HRSERVER）の ID およびユーザー名を戻すかどうかを判断できます。ユーザーがデータベースに直接接続しようとすると、hr.admin パッケージはロールの設定を許可しません。

ヒント 5: 保護アプリケーション・ロールを使用した IP アドレスの検証

保護アプリケーション・ロールは、アクセスを制限するために、ユーザー・セッション内の追加情報を使用できます。IP アドレス・ベースのセキュリティは、偽 IP アドレスによって不当に侵入される可能性があるため、確実ではありません。このため、主アクセス制御の決定には IP アドレスを使用しないでください。ただし、その他の制御に加え、さらにアクセスを制限するために、IP アドレスを使用できます。たとえば、ユーザー・セッションがプロキシによって作成されたこと、および特定の IP アドレスから接続する中間層ユーザーがユーザー・セッションを作成したことを確認する場合などです。中間層は、軽量セッションを作成する前に、データベースに対して自身を認証する必要があります。また、データベースは、ユーザーにかわってセッションを作成する権限を中間層が持っていることを確認します。保護アプリケーション・ロールは、`SET ROLE` の実行を許可する前に、`SYS_CONTEXT ('userenv', 'ip_address')` を使用して、接続元の IP アドレスを検証し、`HRSERVER` 接続（または軽量ユーザー・セッション）が適切な IP アドレスからのものであることを保証できます。これによって、セキュリティ・レイヤーが追加されます。

ヒント 6: アプリケーション・コンテキストおよびファイングレイン・アクセス・コントロールの使用

この使用例では、アプリケーション・コンテキストを介して、サーバー施行のファイングレイン・アクセス・コントロールとセッション・ベースの属性を組み合せます。

参照： 12-7 ページの「ファイングレイン・アクセス・コントロールでのアプリケーション・コンテキストの使用方法」を参照してください。

アプリケーション・セキュリティ・ポリシーの概要

セキュリティ・ポリシーは、データベース・アプリケーションごとに作成します。たとえば、各データベース・アプリケーションには、アプリケーションの実行時に異なるセキュリティ・レベルを提供する、1 つ以上のデータベース・ロールが必要です。データベース・ロールは、ユーザー・ロールに付与するか、または特定のユーザー名に直接付与できます。

（SQL*Plus などのツールを使用して）SQL 文を制限なしで実行できるアプリケーションについても、機密扱いのスキーマ・オブジェクトまたは重要なスキーマ・オブジェクトに対する不法なアクセスを防ぐために、セキュリティ・ポリシーが必要です。

この項では、アプリケーション・セキュリティ・ポリシーに関する次の項目について説明します。

- アプリケーション・ベースのセキュリティの使用に関する考慮点
- アプリケーション管理者のセキュリティ関連作業
- アプリケーション権限の管理
- 保護アプリケーション・ロールの作成

- ユーザーのデータベース・ロールに対する権限の対応付け
- スキーマの使用によるデータベース・オブジェクトの保護
- オブジェクト権限の管理
- ロールの使用可能および使用禁止
- システム権限およびロールの付与と取消し
- スキーマ・オブジェクト権限およびロールの付与と取消し
- ユーザー・グループ PUBLIC に対する権限付与および取消し

アプリケーション・ベースのセキュリティの使用に関する考慮点

アプリケーション・セキュリティを作成および実装する場合、多くの考慮点があります。主な考慮点は、次の 2 つです。

- アプリケーション・ユーザーがデータベース・ユーザーであるかどうか
- セキュリティがアプリケーションまたはデータベースのどちらで施行されるか

アプリケーション・ユーザーがデータベース・ユーザーであるかどうか

オラクル社では、アプリケーション・ユーザーがデータベース・ユーザーであるアプリケーションを作成することをお薦めします（可能な場合）。こうすると、データベースのセキュリティ・メカニズムを使用できます。

多くの商用パッケージ・アプリケーションでは、アプリケーション・ユーザーはデータベース・ユーザーではありません。これらのアプリケーションでは、複数のユーザーがアプリケーションに対して認証され、アプリケーションは高い権限を持つ単一のユーザーとしてデータベースに接続します。これを、「One Big Application User」モデルといいます。

この方法で作成されたアプリケーションは、ユーザーの認証がデータベースでは認識されないため、データベースの基本的なセキュリティ機能の多くを使用できません。

One Big Application User モデルによって使用できる機能の例は、次のとおりです。

Oracle 機能	One Big Application User モデルの制限
監査	セキュリティの基本原理は、監査による信頼性です。ただし、データベースにおけるすべてのアクションが One Big Application User によって行われる場合、データベース監査は、個々のユーザーのアクションに対する信頼性を保持できません。アプリケーションはその監査メカニズムを実装して、個々のユーザーのアクションを受け入れる必要があります。

Oracle 機能	One Big Application User モデルの制限
Oracle Advanced Security の拡張認証	データベースに対して認証するクライアントが個々のユーザーではなくアプリケーションである場合、Oracle Advanced Security がサポートする強力な認証形式（SSL を介したクライアント認証、トークンなど）は使用できません。
ロール	ロールはデータベース・ユーザーに割り当てられます。エンタープライズ・ロールを割り当てられるエンター・プライズ・ユーザーは、データベース内で作成されていない場合でも、データベースに認識されている必要があります。アプリケーション・ユーザーがデータベース・ユーザーでない場合、ロールの有用性は低くなります。そのため、アプリケーションで独自のメカニズムを構成し、様々なアプリケーション・ユーザーがそのアプリケーション内でデータにアクセスするために必要な権限を識別する必要があります。
Oracle Advanced Security のエンタープライズ・ユーザー管理機能	この機能を使用すると、ユーザーおよびユーザー認証を Oracle Internet Directory などの Lightweight Directory Accsss Protocol（LDAP）ベースのディレクトリ内で集中管理できます。エンタープライズ・ユーザーは、データベース内で作成する必要はありませんが、データベースによって認識される必要はあります。One Big Application User モデルでは、LDAP でのユーザーおよび認証の管理を使用できません。

セキュリティがアプリケーションまたはデータベースのどちらで施行されるか

データベース・ユーザーでもあるユーザーが使用するアプリケーションは、アプリケーションに対してセキュリティを構築するか、または詳細な権限、VPD（アプリケーション・コンテキストを持つファイングレイン・アクセス・コントロール）、ロール、ストアド・プロシージャ、監査（ファイングレイン監査を含む）などの、基本的なデータベース・セキュリティ・メカニズムに依存することができます。オラクル社では、できるだけ、アプリケーションでデータベースのセキュリティ施行メカニズムを使用することをお薦めします。

セキュリティがアプリケーションではなくデータベースで施行される場合、セキュリティを回避することができません。アプリケーション・ベースのセキュリティの大きなデメリットは、ユーザーがアプリケーションを介さずにデータにアクセスすると、セキュリティが回避されることです。たとえば、データベースへの SQL*Plus アクセスを持つユーザーは、Human Resource アプリケーションを使用しなくても問合せを実行できます。そのため、ユーザーは、アプリケーションでのすべてのセキュリティ保護規準を回避します。

One Big Application User モデルを使用するアプリケーションは、データベース・セキュリティ・メカニズムを使用するのではなく、アプリケーションに対してセキュリティを施行する必要があります。この場合、ユーザーを認識するのはデータベースではなくアプリケー

ションであるため、アプリケーションは、ユーザーごとのセキュリティ保護規準をそのアプリケーション自体に施行する必要があります。

この場合、セキュリティは、データにアクセスするすべてのアプリケーションごとに再実装される必要があります。たとえば、組織が新しいレポート作成ツールを実装する場合、ユーザーが、アプリケーションよりレポート作成ツールでより多くのデータ・アクセスを行わないようにするために、その組織は、セキュリティも実装する必要があります。組織は、同じセキュリティ・ポリシーを複数のアプリケーションに実装する必要があるため、セキュリティにはコストがかかります。新しい各アプリケーションには、コストのかかる再実装が必要です。

参照： 12-51 ページの「セキュリティに関する潜在的な問題となる非定型ツールの使用」を参照してください。

アプリケーション管理者のセキュリティ関連作業

アプリケーションを多数使用する大規模データベース・システムでは、アプリケーション管理者を設定する方がよい場合があります。アプリケーション管理者には、次の作業に対する責任があります。

- データベース・アプリケーションに対するロールの作成および各データベース・ロールの権限の管理
- アプリケーションによって使用されるオブジェクトの作成および管理
- アプリケーション・コードや Oracle プロシージャおよび Oracle パッケージの必要に応じたメンテナンスおよび更新

アプリケーション権限の管理

ほとんどのデータベース・アプリケーションでは、様々なスキーマ・オブジェクトに様々な権限が関連するため、各アプリケーションにどの権限が必要かを追跡することは非常に複雑になることがあります。さらに、アプリケーションを実行するユーザーの認可には、多数の GRANT 操作が必要になります。この項では、アプリケーション権限を管理するためのいくつかの機能について説明します。内容は次のとおりです。

- アプリケーション権限の管理を簡単にするためのロールの作成
- アプリケーション権限のロールへのグループ化のメリット

アプリケーション権限の管理を簡単にするためのロールの作成

アプリケーション権限の管理を簡単にするために、各アプリケーションに対してロールを作成し、そのロールに対して、ユーザーがアプリケーションの実行に必要とするすべての権限を付与することができます。実際には、アプリケーションには多数のロールが割り当てら

れ、各ロールに対して、アプリケーション実行中の利用機能の多少を決める異なる権限が付与されます。

たとえば、部門メンバーの取得休暇を記録する Vacation アプリケーションをすべての重役補佐が使用する場合を想定します。このアプリケーションの最適な管理を行うには、次のことを実行してください。

1. VACATION ロールを作成します。
2. VACATION ロールに対して、Vacation アプリケーションで必要なすべての権限を付与します。
3. 重役補佐全員または ADMIN_ASSISTS という名前の付いたロール（事前に定義してある場合）に VACATION ロールを付与します。

アプリケーション権限のロールへのグループ化のメリット

1 つのロールにアプリケーション権限をグループ化しておくと、権限を管理する場合に有効です。次の管理オプションを考慮してください。

- アプリケーションを実行するユーザーに対して、個々に多数の権限を付与するのではなく、ロールを付与することができます。そのため、社員の職種が変わったときは、多数の権限ではなく、1 つのロールを付与または取り消すのみで済みます。
- アプリケーションのすべてのユーザーが保持している権限ではなく、ロールに対して付与されている権限のみを修正することによって、アプリケーションに対応付けられている権限を変更できます。
- ROLE_TAB_PRIVS および ROLE_SYS_PRIVS の各データ・ディクショナリ・ビューを問い合せることによって、特定のアプリケーションの実行に必要な権限を判断できます。
- DBA_ROLE_PRIVS データ・ディクショナリ・ビューを問い合せることによって、どのユーザーがどのアプリケーションに対して権限を持っているかを判断できます。

保護アプリケーション・ロールの作成

データベース・アクセスは権限に基づきます。多くの場合、権限はロールにグループ化されます。一度グループ化されると、そのロールはアプリケーション・ユーザーに付与されます。以前のリリースでは、ユーザーが付与されたロールをアプリケーション内でのみ使用可能にすることを保証するために、アプリケーション内にパスワードが埋め込まれました。アプリケーション内にパスワードを埋め込むことによって保護されるロールを、アプリケーション・ロールといいます。

Oracle9*i* では、アプリケーション開発者は、アプリケーションにパスワードを埋め込んでロールを保護する必要がありません。開発者は、アプリケーション・ロールを作成して、そのロールを使用可能にする権限がある PL/SQL パッケージを指定できます。PL/SQL パッケージによって使用可能になったアプリケーション・ロールを、保護アプリケーション・ロールといいます。

保護アプリケーション・ロールを実装するパッケージ内では、次の処理を行う必要があります。

- アプリケーションは、必要な妥当性チェックを行う必要があります。たとえば、アプリケーションは、ユーザーが特定の部門に属しているかどうかを検証し、ユーザー・セッションが特定の IP アドレスからプロキシによって作成されているか、またはユーザーが X.509 証明書を使用して認証されていることを確認する必要があります。アプリケーションは、妥当性チェックを実行するために、SYS_CONTEXT ('userenv', <session_attribute>) を介してアクセス可能なユーザー・セッション情報を使用できます。このアクセス可能な情報は、ユーザーが認証された方法、クライアントの IP アドレス、およびユーザーがプロキシ化されているかどうかを示します。
- アプリケーションは、動的 SQL（DBMS_SESSION.SET ROLE）を使用して、SET_ROLE コマンドを発行する必要があります。

この項の内容は次のとおりです。

注意： ユーザーは定義者権限プロシージャ内のセキュリティ・ドメインを変更できないため、保護アプリケーション・ロールは実行者権限プロシージャ内でのみ使用可能にできます。

保護アプリケーション・ロールの作成の例

保護アプリケーション・ロールを作成するには、次の手順に従います。

1. ロールをアプリケーション・ロールとして作成し、ロールを使用可能にすることを許可されたパッケージを指定します。この例では、hr.hr_admin が、許可されたパッケージです。

```
CREATE ROLE admin_role IDENTIFIED USING hr.hr_admin;
CREATE ROLE staff_role IDENTIFIED USING hr.hr_admin;
```

2. 実行者権限プロシージャを作成します。

```
CREATE OR REPLACE PACKAGE hr_admin
AUTHID CURRENT_USER
IS
PROCEDURE hr_app_report;
END;
/
CREATE OR REPLACE PACKAGE BODY hr_admin IS
PROCEDURE hr_app_report IS
BEGIN
```

```
    /* set application context in 'responsibility' namespace */
    hr_logon.hr_set_responsibility;
    /* authentication check here */
    if (Hr.MySecurityCheck = TRUE)
    then
        /* check 'responsibility' being set, then enable the roles without
        supplying the password */
        if (sys_context('hr','role') = 'admin' )
    then
        dbms_session.set_role('admin_role');
    else
        dbms_session.set_role('staff_role');
        end if;
    end if;
END;
END;
/* Create a dedicated authentication function for manageability so that changes
in authentication policies would not affect the source code of the application -
this design is up the application developers */
/* the only policy in this function is that current user must have been
authenticated using the proxy user 'SCOTT' */
CREATE OR REPLACE FUNCTION hr.MySecurityCheck RETURN BOOLEAN
AS
BEGIN
    /* a simple check to see if current session is authenticated
    by the proxy user 'SCOTT' */
    if (sys_context('userenv','proxy_user') = 'SCOTT')
    then
        return TRUE;
    else
        return FALSE;
    end IF;
END;
```

保護アプリケーション・ロールを使用可能にする場合、Oracle は、許可された PL/SQL パッケージがコール・スタック上にあるかどうかを検証します。この手順では、許可された PL/SQL パッケージがロールを使用可能にするコマンドを発行しているかどうかが検証されます。また、ユーザーのデフォルト・ロールを使用可能にする場合、アプリケーション・ロールに対するチェックは行われません。

保護アプリケーション・ロールを使用したデータベース接続の保証

保護アプリケーション・ロールはパッケージによって実装されるロールであるため、パッケージは、ユーザーが中間層または特定の IP アドレスによってデータベースに接続できることの確認など、必要な妥当性チェックを実行できます。このようにして、ユーザーがアプ

リケーション外のデータにアクセスすることを防ぎます。ユーザーは、付与されているアプリケーション権限のフレームワーク内で作業することになります。

ユーザーのデータベース・ロールに対する権限の対応付け

1 人のユーザーが、多数のアプリケーションおよびその関連ロールを使用できます。ただし、ユーザーが実行中のデータベース・ロールに対応付けられた権限のみを持っていることを確認する必要があります。次の使用例を考えます。

- ORDER ロール（ORDER アプリケーション用）には、INVENTORY 表に対する UPDATE 権限が含まれています。
- INVENTORY ロール（INVENTORY アプリケーション用）には、INVENTORY 表に対する SELECT 権限が含まれています。
- 注文入力オペレータの何人かは、ORDER ロールおよび INVENTORY ロールの両方が付与されています。

この使用例では、両方のロールを付与されている注文入力オペレータは、INVENTORY 表を更新する INVENTORY アプリケーションを実行するときに、ORDER ロールの権限を使用できます。問題は、INVENTORY 表の更新は、INVENTORY アプリケーションの使用中は許可されず、ORDER アプリケーションの使用中にのみ許可されているということです。

このような問題を回避するには、次に説明する SET ROLE 文または SET_ROLE プロシージャのどちらかの使用を検討します。また、保護アプリケーション・ロール機能を使用して、ロールをユーザー定義の基準に基づいて設定することもできます。

この項の内容は次のとおりです。

SET ROLE 文の使用

各アプリケーションの最初に SET ROLE 文を使用して、関連ロールを自動的に使用可能にします。その結果として、それ以外のロールは自動的に使用禁止になります。このようにして、各アプリケーションは、必要な場合にのみ、ユーザーが特定の権限を動的に使用できるようにします。

SET ROLE 文を使用すると、ユーザーがどの情報にアクセスできるかの制御以外にも、ユーザーが情報にいつアクセスできるかを制御できるため、権限の管理が容易になります。また、SET ROLE 文によって、ユーザーは正しく定義された権限ドメイン内で操作できます。ユーザーがロールのみから権限を取得する場合は、それらを組み合せて不当な操作を実行することはできません。

参照： 11-23 ページの「ロールの使用可能および使用禁止」を参照してください。

SET_ROLE プロシージャの使用

PL/SQL パッケージ DBMS_SESSION.SET_ROLE は、機能的に SQL の SET ROLE 文と同等です。

ロールの制限事項は、定義者権限プロシージャ内の SET ROLE を実行できないことです。これは、定義者権限プロシージャに対して、データベースが実行時ではなくコンパイル時に権限を確認するためです。つまり、データベースは、プロシージャのコンパイル時に、プロシージャの所有者が必要な権限（ロールを使用してではなく、直接付与された）を持っているかどうかを検証します。データベースが権限を確認するとき、ロールはコンパイル時には使用禁止であるため、SET ROLE 文は動作しません。実行時にロールは使用可能ですが、データベースは所有者の権限を確認しません。データベースはプロシージャのユーザーがそのプロシージャに対する EXECUTE 権限を持っているかどうかのみを確認します。

データベースが、コンパイル時ではなく実行時に権限を確認する場合は、SET ROLE を実行できます。そのため、DBMS_SESSION.SET_ROLE コマンドは次のものからコールできます。

- 無名 PL/SQL ブロック
- 実行者権限ストアド・プロシージャ（定義者権限プロシージャ内から起動されたものを除く）

前述のどちらの場合も、データベースは、コンパイル時ではなく実行時に権限を確認します。そのため、データベースは、ユーザーが適切な権限を持っているかどうか（設定されたロールがユーザーに付与されているかどうか）を妥当性チェックできます。

注意： DBMS_SESSION.SET_ROLE を実行者権限プロシージャ内で使用する場合、ロールを使用禁止にするまでは、そのロールは有効のままです。最小権限の原理（ユーザーは、ジョブを行うための必要最小限の権限を持つ）に従って、実行者権限プロシージャ内のロールの設定を、プロシージャの終わりで明示的に使用禁止にする必要があります。

PL/SQL では、無名ブロックのコンパイル時に SQL に対してセキュリティ・チェックを行うため、SET_ROLE は埋込み SQL 文またはプロシージャ・コールのセキュリティ・ロールには影響しません（使用可能になっているロールには影響しません）。

静的および動的 SQL を使用したロールの割当ての例

この項では、静的および動的 SQL がどのようにロールの割当てに影響するかについて説明します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。次を設定します。

```
CONNECT system/manager
DROP USER joe CASCADE;
CREATE USER joe IDENTIFIED BY joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO scott;
DROP ROLE acct;
CREATE ROLE acct;
GRANT acct TO scott;

CONNECT joe/joe;
CREATE TABLE finance (empno NUMBER);
GRANT SELECT ON finance TO acct;
CONNECT scott/tiger
```

たとえば、JOE スキーマ内の表 FINANCE を選択する権限を付与されている ACCT という名前のロールを持っているとします。この場合、次のブロックは正常に実行されません。

```
DECLARE
    n NUMBER;
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SELECT empno INTO n FROM JOE.FINANCE;
END;
```

このブロックは正常に実行されません。これは、表 JOE.FINANCE に対する SELECT 権限を持っているかどうかを検証するセキュリティ・チェックがコンパイル時に発生するためです。ただし、コンパイル時には、ACCT ロールは使用可能ではありません。このロールは、ブロックが実行されるまで使用可能にはなりません。

ただし、DBMS_SQL パッケージは、この制限を受けません。このパッケージを使用する場合、セキュリティ・チェックは実行時に行われます。したがって、SET_ROLE をコールすると、DBMS_SQL パッケージへのコールを使用して実行される SQL が影響を受けます。そのため、次のブロックは正常に実行されます。

```
CREATE OR REPLACE PROCEDURE dynSQL_proc
AUTHID CURRENT_USER AS
   n NUMBER;
BEGIN
   SYS.DBMS_SESSION.SET_ROLE('acct');
   EXECUTE IMMEDIATE 'select empno from joe.finance' INTO n;
    --other calls to SYS.DBMS_SQL
END;
```

参照： 8-10 ページの「ネイティブ動的 SQL または DBMS_SQL パッケージの選択」を参照してください。

スキーマの使用によるデータベース・オブジェクトの保護

スキーマとは、データベース・オブジェクトを含むことができるセキュリティ・ドメインです。各ユーザーまたはロールに付与された権限が、これらのデータベース・オブジェクトへのアクセスを制御します。この項の内容は次のとおりです。

- 一意スキーマ
- 共有スキーマ

一意スキーマ

ほとんどのスキーマは、ユーザー名であると考えてかまいません。ユーザー名は、ユーザーがデータベースに接続して、そのデータベースのオブジェクトにアクセスすることを許可するために設定されたアカウントです。ただし、一意スキーマは、データベースへの接続は許可せず、関連した一連のオブジェクトを含むために使用されます。このようなスキーマは通常のユーザーとして作成されますが、（明示的にもロールを介しても）CREATE SESSION システム権限は付与されません。ただし、単一トランザクションで複数の表およびビューを作成するために CREATE SCHEMA 文を使用する場合は、このようなスキーマに対して CREATE SESSION および RESOURCE 権限を一時的に付与する必要があります。

たとえば、特定のアプリケーションのスキーマ・オブジェクトは、所定のスキーマによって所有されます。アプリケーション・ユーザーは、権限がある場合は、通常のデータベース・ユーザー名を使用してデータベースに接続し、そのアプリケーションおよび対応オブジェクトを使用できます。ただし、ユーザーは、アプリケーションに設定されたスキーマを使用して、データベースに接続することはできません。この構成によって、スキーマを介して対応付けられたオブジェクトへのアクセスが防止され、スキーマ・オブジェクトを保護する別のレイヤーが提供されます。この場合、アプリケーションは ALTER SESSION SET CURRENT_SCHEMA 文を発行して、ユーザーを正しいアプリケーション・スキーマに接続させることができます。

共有スキーマ

多くのアプリケーションでは、ユーザーは、データベースにユーザー自身のアカウント（またはスキーマ）を必要としません。これらのユーザーは、アプリケーション・スキーマにアクセスする必要があるのみです。たとえば、ユーザー John、Firuzeh および Jane の 3 人が、Payroll アプリケーションのすべてのユーザーであり、Finance データベースの Payroll スキーマにアクセスする必要があるとします。これらのユーザーは、それ自身のオブジェクトをデータベースに作成する必要はなく、Payroll オブジェクトにアクセスする必要があるのみです。この問題を解決するために、Oracle Advanced Security では、エンタープライズ・ユーザー（スキーマ非依存ユーザー）を提供しています。

エンタープライズ・ユーザー（ディレクトリ・サービスで管理されるユーザー）は、共有スキーマにアクセスできます。これらのユーザーは、データベース・ユーザーとして作成される必要はなく、データベースの共有スキーマ・ユーザーになります。ユーザー・アカウント（ユーザー・スキーマ）を、エンタープライズ・ユーザーがアクセスする必要のある各データベースおよびディレクトリに作成するかわりに、管理者は、ディレクトリ内にエンタープライズ・ユーザーを 1 回のみ作成し、そのユーザーが、多くの他のエンタープライズ・ユーザーもアクセスできる共有スキーマを指定するように設定できます。

前述の例では、John、Firuzeh および Jane の 3 人が、Finance データベースに加えて Sales データベースにもアクセスする場合、管理者は、Sales データベースに、これら 3 人のユーザーがアクセスできる単一のスキーマを作成する必要があるのみです。Sales データベースに各ユーザーのアカウントを作成する必要はありません。この場合、Sales データベースの DBA は、sales_application という共有スキーマを次のようにして作成します。

```
CREATE USER sales_application IDENTIFIED GLOBALLY AS ' ';
```

エンタープライズ・ユーザーとスキーマとの間のマッピングは、1 つ以上のマッピング・オブジェクトを使用して、ディレクトリで行われます。マッピング・オブジェクトは、ユーザーの識別名（DN）を、ユーザーがアクセスするデータベース・スキーマにマップします。これは、次のいずれかの方法で行うことができます。

- 完全 DN のマッピングは、単一のディレクトリ・ユーザーの DN をデータベース・スキーマにマップします。そのため、このユーザーを、データベースのある特定のスキーマに対応付けます。
- 部分 DN のマッピングは、DN の一部を共有するすべてのユーザーをデータベース・スキーマにマップします。部分 DN のマッピングは、共通点のある複数のエンタープライズ・ユーザーが、ディレクトリ・ツリーの共通のルートですでにグループ化されている場合に有効です。たとえば、技術部門に対応するディレクトリのサブツリーにあるすべてのエンタープライズ・ユーザーは、エラー・データベース上の 1 つの共有スキーマにマップできます。そのため、DN の一部を共有する複数のエンタープライズ・ユーザーは、同じ共有スキーマにアクセスできます。

データベースが、ディレクトリ内のエンタープライズ・ユーザーのスキーマ（データベースがユーザーを接続するスキーマ）を判断する場合、データベースは完全 DN マッピングを検索します。完全 DN マッピングが検出されなかった場合は、部分 DN マッピングを検索します。そのため、完全 DN マッピングは、部分 DN マッピングより優先順位が高くなります。

SSL によってデータベースに対して認証されているユーザー、または X.509 証明書や証明書に含まれる DN がデータベースに対してプロキシ化されているユーザーの場合、データベースは DN を使用してそのユーザーをディレクトリ内で検索します。パスワードによって認証されたエンタープライズ・ユーザーの場合、DN はディレクトリから取得されます。つまり、認証のためにデータベースに対してユーザー名（JANE など）が提示された場合、データベースはローカル・ユーザー Jane が存在するかどうかを内部で検索します。存在しない場合、データベースはディレクトリで Jane を検索し、Jane に対応付けられた DN を取得します。その後、データベースは、前述のとおりマッピング・オブジェクトを参照して、Jane が接続する正しい共有スキーマを判断します。

一連の権限をユーザーのグループに付与する必要がある場合、共有スキーマにロールおよび権限を付与します。スキーマを共有しているすべてのユーザーは、エンタープライズ・ロールに加えて、これらのローカル・ロールおよびローカル権限を取得します。

各エンタープライズ・ユーザーは、ユーザーがアクセスする必要のある各データベース上の共有スキーマにマップできます。そのため、これらのスキーマ非依存ユーザーには、各データベース上の専用データベース・スキーマは必要ありません。こうして、共有スキーマによって、企業内でのユーザー管理のコストが低くなります。

参照：

- 2-30 ページの「異なるスキーマへの切替え」を参照してください。
- 『Oracle Advanced Security 管理者ガイド』を参照してください。

オブジェクト権限の管理

アプリケーション設計の一部として、アプリケーションを使用するユーザーの種類、およびユーザーが業務を遂行するために必要なアクセス・レベルを決定する必要があります。これらのユーザーをロール・グループに分類したうえで、各ロールに対して付与する権限を決定します。この項の内容は次のとおりです。

オブジェクト権限

通常、エンド・ユーザーにはオブジェクト権限が付与されます。オブジェクト権限によって、ユーザーは、特定の表、ビュー、順序、プロシージャ、ファンクションまたはパッケージに対して特定のアクションを実行できます。表 11-1 に、各オブジェクト型で使用可能なオブジェクト権限の概要を示します。

表 11-1 スキーマ・オブジェクトで使用可能な権限

オブジェクト権限	表への適用	ビューへの適用	順序への適用	プロシージャへの適用 (1)
ALTER	あり	なし	あり	なし
DELETE	あり	あり	なし	なし
EXECUTE	なし	なし	なし	あり
INDEX	あり (2)	なし	なし	なし
INSERT	あり	あり	なし	なし
REFERENCES	あり (2)	なし	なし	なし
SELECT	あり	あり (3)	あり	なし
UPDATE	あり	あり	なし	なし

注意：

1. スタンドアロンのストアド・プロシージャ、ファンクションおよびパブリック・パッケージ構造体
2. ロールに付与できない権限
3. スナップショットにも付与できる権限

オブジェクト権限によって許可される SQL 文

アプリケーションを実装およびテストするときには、必要な各ロールを作成する必要があります。アプリケーションのユーザーがデータベースに適切にアクセスすることを確認するために、各ロールの使用例をテストします。テストの完了後、各ユーザーに適切なロールが割り当てられていることをアプリケーション管理者と調整し確認する必要があります。

表 11-2 に、表 11-1 に記載されているオブジェクト権限によって許可される SQL 文を示します。

表 11-2 データベース・オブジェクト権限によって許可される SQL 文

オブジェクト権限	許可される SQL 文
ALTER	ALTER オブジェクト（表または順序）
	CREATE TRIGGER ON オブジェクト（表のみ）
DELETE	DELETE FROM オブジェクト（表、ビューまたはシノニム）
EXECUTE	EXECUTE オブジェクト（プロシージャまたはファンクション）
	パブリック・パッケージ変数の参照
INDEX	CREATE INDEX ON オブジェクト（表、ビューまたはシノニム）
INSERT	INSERT INTO オブジェクト（表、ビューまたはシノニム）
REFERENCES	オブジェクト（表のみ）に対する外部キー整合性制約を定義する CREATE 文または ALTER TABLE 文
SELECT	SELECT...FROM オブジェクト（表、ビュー、シノニムまたはスナップショット）
	順序を使用する SQL 文

ロールの作成およびロール使用の保護

この項では、新しいロールの作成方法およびそのロールの使用の保護方法について説明します。この項の内容は次のとおりです。

- 新しいロールの作成および実装
- ロールの管理

新しいロールの作成および実装

ロールを作成するには、CREATE ROLE システム権限が必要です。

新しいロールの名前は、データベースの既存のユーザー名およびロール名の中で一意にする必要があります。ロールはユーザーのスキーマ内には含まれません。

作成直後のロールには、対応付けられた権限はありません。権限を新しいロールに対応付けるには、そのロールに、権限またはその他のロールを付与する必要があります。

ロールの管理

ロールの使用が、オペレーティング・システム、ネットワーク認証サービスまたは LDAP ベースのディレクトリからの情報を使用して許可されるようにロールを作成することもできます。これによって、ロールの管理を集中化できます。

ロールの集中管理には、多くのメリットがあります。たとえば、従業員が退職する場合、その従業員のすべてのロールおよび許可は、1 箇所で変更できます。

ロール使用の保護

ロールの使用は、対応するパスワードで保護できます。次に例を示します。

```
CREATE ROLE Clerk IDENTIFIED BY Bicentennial;
```

パスワードによって保護されたロールがユーザーに付与されている場合、このユーザーがロールを使用可能または使用禁止にするには、SET ROLE 文を使用してそのロールの正しいパスワードを入力する以外に方法はありません。保護なしでロールが作成された場合、権限受領者であれば、だれでもそのロールを使用可能または使用禁止にできます。

個別の SET ROLE 文を使用して、ユーザーの 1 つのデータベース・ロールを使用可能にし、その他すべてのロールを使用禁止にできます。これによって、ユーザーは他のアプリケーション用の（ロールからの）権限を使用できなくなります。SQL*Plus、Enterprise Manager などの非定型の問合せツールを使用すると、ユーザーは、許可されているロールのみを明示的に使用可能にできます。

保護アプリケーション・ロールは、追加のロジックを取り込み、どのような条件下でロールが使用可能になるかを判断できます。条件では、ユーザー・セッション内の使用可能なすべての情報を参照できます。これは、セッションの接続元の IP アドレス、認証方法、ユーザーがプロキシ化（中間層を介して接続）されているかどうかなどの情報に、USERENV アプリケーション・コンテキスト・ネームスペースを介してアクセスできることを意味します。

参照：

- 11-24 ページの「ロールの明示的な使用可能」を参照してください。
- 『Oracle9*i* データベース管理者ガイド』を参照してください。
- ネットワーク認証サービスの詳細は、『Oracle Advanced Security 管理者ガイド』を参照してください。

ロールの使用可能および使用禁止

ユーザーにロールが付与されている場合、そのロールがあらかじめ使用可能になっていないと、ロールに対応付けられた権限をユーザーのカレント・セッションで使用できません。ユーザー・ロールは、いくつでも任意に使用可能または使用禁止にできます。次の項では、ロールを使用可能または使用禁止にする場合、およびユーザーがロールを使用可能または使用禁止にする様々な方法について説明します。この項の内容は次のとおりです。

- ロールを使用可能にする場合
- デフォルト・ロール
- ロールの明示的な使用可能
- OS_ROLES=TRUE の場合のロールの使用可能および使用禁止
- ロールの削除

ロールを使用可能にする場合

通常、ユーザーのセキュリティ・ドメインでは、ユーザーが現在のタスクを実行できるように許可しますが、現行のジョブに不要な権限は持たないように制限します。たとえば、ユーザーは、現在使用中のデータベース・アプリケーションの操作に必要なすべての権限を持つ必要がありますが、他のデータベース・アプリケーションに必要な権限は必要ありません。権限が多すぎると、ユーザーは予想外の方法で情報にアクセスしてしまう可能性があります。

ユーザーに直接付与された権限は、そのユーザーが常に使用できます。このため、直接付与された権限を、ユーザーの現在のタスクに応じて選択的に使用可能および使用禁止にすることはできません。一方、ロールに付与された権限は、そのロールが付与されたユーザーに対して選択的に使用可能にできます。ロールを使用可能にしても、ユーザーに対して明示的に付与された権限には影響しません。次の項では、ユーザーのロールを選択的に使用可能にする（および使用禁止にする）方法について説明します。

デフォルト・ロール

デフォルト・ロールは、ユーザーがセッションを作成したときに自動的に使用可能になります。ユーザーのデフォルト・ロールのリストには、そのユーザーの主な職種機能に対応するロールを含める必要があります。

各ユーザーは、0（ゼロ）または1つ以上のデフォルト・ロールのリストを持っています。ユーザーに直接付与されたすべてのロールは、デフォルト・ロールにすることができます。間接的に付与されているロール（ロールに付与されているロール）は、デフォルト・ロールにすることはできません。

ユーザーのデフォルト・ロールの数は、（MAX_ENABLED_ROLES 初期化パラメータによって指定された）ユーザーごとに許可されている使用可能ロールの最大数を超えることはできません。ユーザーのデフォルト・ロールがこの最大数を超えると、ユーザーが接続しようとしてもエラーが戻され、ユーザーの接続は許可されません。

> **注意：** デフォルト・ロールは、ユーザーがセッションを作成したときに自動的に使用可能になります。ユーザーのデフォルト・ロール・リストにロールを追加すると、そのロールがパスワードまたはオペレーティング・システムを使用して許可されているかどうかにかかわらず、そのロールの認証が回避されます。

ユーザーのデフォルト・ロール・リストは、SQL 文の ALTER USER を使用して設定および変更できます。ユーザーのデフォルト・ロール・リストを ALL に指定すると、ユーザーに付与されているすべてのロールが、そのユーザーのデフォルト・ロール・リストに自動的に追加されます。新しく付与されたロールをユーザーのデフォルト・ロール・リストから削除できるのは、デフォルト・ロール・リストを後で変更するときのみです。

ユーザーのデフォルト・ロール・リストに対する変更は、その変更の後またはロールが付与された後に作成されたセッションにのみ適用されます。いずれの場合も、ユーザー変更またはロール付与が実行されるときに処理中のセッションには適用されません。

ロールの明示的な使用可能

権限受領者がロールのパスワードを設定する場合、ユーザー（またはアプリケーション）は、SET ROLE 文を使用して、付与されたロールを必要に応じて使用可能にすることができます。

ロールがあらかじめユーザーに付与されている場合、SET ROLE 文は指定されたすべてのロールを使用可能にします。ユーザーに付与されている、SET ROLE 文で明示的に指定されていないロールは、以前に使用可能にされたロールを含み、すべて使用禁止になります。

他のロールを含むロールを使用可能にすると、間接的に付与されたすべてのロールが明示的に使用可能になります。間接的に付与された個々のロールは、ユーザーに対して明示的に使用可能または使用禁止にすることができます。

ロールがパスワードによって保護されている場合、ロールを使用可能にできるのは、SET ROLE 文でロールのパスワードを指定する場合のみです。ロールがパスワードによって保護されていない場合は、単純に SET ROLE 文を使用して、そのロールを使用可能にできます。

次の例に、ロールを使用可能および使用禁止にする方法を示します。

ユーザー Morris のセキュリティ・ドメインが次のとおりであるとします。

- Morris には次の 3 つのロールが付与されています。

  PAYROLL_CLERK（パスワード BICENTENNIAL）

  ACCTS_PAY（パスワード GARFIELD）

  ACCTS_REC（外部的に識別されたロール）

- PAYROLL_CLERK ロールには、間接的に付与されたロール PAYROLL_REPORT（外部的に識別されたロール）が含まれています。
- Morris のデフォルト・ロールは PAYROLL_CLERK のみです。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE ROLE Payroll_clerk;
CREATE ROLE Payroll_report;

GRANT PAYROLL_CLERK TO hr;
GRANT ACCTS_PAY TO hr;
GRANT ACCTS_REC TO hr;
```

次の文によって、Morris の現在使用可能なロールを、デフォルト・ロール PAYROLL_CLERK から ACCTS_PAY および ACCTS_REC に変更できます。

```
SET ROLE accts_pay IDENTIFIED BY garfield;
SET ROLE accts_pay IDENTIFIED BY accts_rec;
```

最初の文では、単一の SET ROLE 文で複数のロールを使用可能にすることができます。また、SET ROLE 文の ALL オプションおよび ALL EXCEPT オプションを使用すると、ユーザーに直接付与されている複数のロールを、次の 1 つの文で使用可能にすることもできます。

```
SET ROLE ALL EXCEPT Payroll_clerk;
```

この文は、SET ROLE 文の ALL EXCEPT オプションの使用方法を示しています。このオプションは、ユーザーのほとんどのロールを使用可能にし、1 つまたは 2 つのロールのみを使用禁止にする場合に使用します。同様に、Morris のすべてのロールは、次の文によって使用可能にできます。

```
SET ROLE ALL;
```

SET ROLE 文の ALL オプションまたは ALL EXCEPT オプションを使用する場合、使用可能にするすべてのロールは、パスワードが不要かまたはオペレーティング・システムを使用して認証するかのどちらかである必要があります。ロールにパスワードが必要な場合は、SET ROLE ALL 文または SET ROLE ALL EXCEPT 文はロールバックされ、エラーが戻されます。また、ユーザーは、そのユーザーに間接的に付与された任意のロールを、別のロールを明示的に付与することによって明示的に使用可能にできます。このため、Morris は次の文を発行できます。

```
SET ROLE Payroll_report;
```

OS_ROLES=TRUE の場合のロールの使用可能および使用禁止

OS_ROLES が TRUE に設定されている場合は、オペレーティング・システムによって付与されている任意のロールを、SET ROLE 文を使用して動的に使用可能にできます。ただし、ユーザーのオペレーティング・システム・アカウントで識別されないロールは、GRANT 文を使用してロールが付与されていても、SET ROLE 文には指定できません（無視されます）。

OS_ROLES が TRUE に設定されている場合、ユーザーは初期化パラメータ MAX_ENABLED_ROLES で指定されている数までのロールを使用可能にできます。

> **参照：** ロール認証に対するオペレーティング・システムの使用については、『Oracle9*i* データベース管理者ガイド』を参照してください。

ロールの削除

ロールを削除すると、そのロールが付与されているすべてのユーザーおよびロールのセキュリティ・ドメインがすぐに変更され、削除されたロールの権限がなくなったことが反映されます。削除されたロールの中で間接的に付与されているすべてのロールも、影響を受けたセキュリティ・ドメインから削除されます。ロールを削除すると、すべてのユーザーのデフォルト・ロール・リストからそのロールが自動的に削除されます。

オブジェクトの作成はロールを介して付与される権限に依存しないため、ロールを削除する場合は、オブジェクトに関する波及効果を考慮する必要はありません（たとえば、ロールを削除するときに、表または他のオブジェクトは削除されません）。

ロールは、SQL 文 DROP ROLE を使用して削除できます。次に例を示します。

```
DROP ROLE clerk;
```

ロールを削除するには、DROP ANY ROLE システム権限を持っているか、または Admin Option 付きのロールが付与されている必要があります。

システム権限およびロールの付与と取消し

次の項では、システム権限およびロールの付与方法および取消し方法を説明します。

- システム権限およびロールの付与
- Admin Option 付きのシステム権限およびロールの付与
- システム権限およびロールの取消し

システム権限およびロールの付与

システム権限およびロールは、次の例に示すとおり、SQL コマンド GRANT を使用して他のロールまたはユーザーに付与できます。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT sys/change_on_install AS sysdba;
CREATE USER jward IDENTIFIED BY jward;
CREATE USER tsmith IDENTIFIED BY tsmith;
CREATE USER finance IDENTIFIED BY finance;
CREATE USER michael IDENTIFIED BY michael;
CREATE ROLE Payroll_report;
GRANT CREATE TABLE, Accts_rec TO finance IDENTIFIED BY finance;
GRANT CREATE TABLE, Accts_rec TO tsmith IDENTIFIED BY tsmith;
GRANT REFERENCES ON Dept_tab TO jward;
CONNECT scott/tiger
CREATE VIEW Salary AS SELECT Empno,Sal from Emp_tab;
```

```
GRANT CREATE SESSION, Accts_pay TO jward, finance;
```

同一の GRANT 文中で、システム権限およびロールとともにスキーマ・オブジェクト権限を付与することはできません。

Admin Option 付きのシステム権限およびロールの付与

システム権限またはロールは、Admin Option 付きで付与できます。このオプションを付与された場合、権限受領者はさらに次の機能を使用できます。

- 権限受領者は、データベースの任意のユーザーまたは他のロールに対して、システム権限またはロールを付与または取り消すことができます。ただし、ユーザーは自分のロールを取り消すことはできません。
- 権限受領者は、Admin Option 付きのシステム権限またはロールを付与できます。
- ロールの権限受領者は、ロールを変更または削除することができます。

Admin Option を付与されていない場合、権限受領者は前述の操作を実行できません。また、このオプションは、ロールを別のロールに対して付与するときには無効です。

ロールを作成すると、作成者には、Admin Option 付きでそのロールが自動的に付与されます。

次の文を使用して、MICHAEL に NEW_DBA ロールを付与する場合を想定します。

```
GRANT new_dba TO michael WITH ADMIN OPTION;
```

ユーザー MICHAEL は、NEW_DBA ロールに暗黙的に含まれているすべての権限を使用できるのみでなく、必要に応じて NEW_DBA ロールを付与、取消しまたは削除することができます。

システム権限またはロールの付与に必要な権限 システム権限またはロールを付与するには、付与するすべてのシステム権限およびロールに対する Admin Option が、権限付与者に必要です。さらに、GRANT ANY ROLE システム権限を持つすべてのユーザーは、データベース内の任意のロールを付与できます。

システム権限およびロールの取消し

システム権限およびロールは、SQL コマンド REVOKE を使用して取り消すことができます。次に例を示します。

```
REVOKE CREATE TABLE, Accts_rec FROM tsmith, finance;
```

システム権限またはロールに対する Admin Option を選択して取り消すことはできません。これを行うには、まず権限またはロールを取り消してから、Admin Option なしでその権限またはロールを付与する必要があります。

システム権限およびロールを取り消すために必要な権限 システム権限またはロールに対する Admin Option を持つすべてのユーザーは、他のすべてのデータベース・ユーザーまたはロールの権限またはロールを取り消すことができます。権限またはロールを取り消すユーザーは、その権限またはロールを付与したユーザーである必要はありません。さらに、GRANT ANY ROLE 権限を持つすべてのユーザーは、すべてのロールを取り消すことができます。

システム権限の取消しの波及効果 DDL 操作に関連するシステム権限を取り消す場合、権限が Admin Option 付きで付与されているかどうかにかかわらず、波及効果はありません。たとえば、次の条件を想定します。

1. JWARD に対して、CREATE TABLE システム権限を With Admin Option 付きで付与します。
2. JWARD は、表を作成します。
3. JWARD は、TSMITH に対して CREATE TABLE システム権限を付与します。
4. TSMITH は、表を作成します。
5. JWARD から CREATE TABLE 権限を取り消します。
6. JWARD の表は引き続き存在します。TSMITH は引き続き CREATE TABLE システム権限を持ち、その表も存在します。

波及効果は、DML 操作に関連するシステム権限を取り消すときに発生します。たとえば、SELECT ANY TABLE を付与されたユーザーがプロシージャを作成した場合、そのユーザーのスキーマ内に含まれているすべてのプロシージャが再認可されないかぎり、それらのプロシージャは（取消し後）再利用できません。

スキーマ・オブジェクト権限およびロールの付与と取消し

スキーマ・オブジェクト権限は、SQL コマンド GRANT を使用して、ロールまたはユーザーに付与できます。次の文は、ユーザー JWARD および TSMITH に対して、EMP_TAB 表のすべての列に関する SELECT、INSERT および DELETE の各オブジェクト権限を付与します。

```
GRANT SELECT, INSERT, DELETE ON Emp_tab TO jward, tsmith;
```

ユーザー JWARD および TSMITH に対して、EMP_TAB 表の ENAME 列および JOB 列のみに関する INSERT オブジェクト権限を付与するには、次の文を入力します。

```
GRANT INSERT(Ename, Job) ON Emp_tab TO jward, tsmith;
```

SALARY ビューのすべてのスキーマ・オブジェクト権限をユーザー WALLEN に付与するには、ALL ショート・カットを使用します。次に例を示します。

```
GRANT ALL ON Salary TO wallen;
```

同一の GRANT 文中で、スキーマ・オブジェクト権限とともにシステム権限およびロールを付与することはできません。

次の項では、スキーマ・オブジェクト権限の付与および取消しについて説明します。内容は次のとおりです。

Grant Option 付きのスキーマ・オブジェクト権限の付与および取消し

スキーマ・オブジェクト権限は、Grant Option 付きでユーザーに付与できます。この特別な権限によって、権限受領者には次のような権限が許可されます。

- 権限受領者は、データベース内の任意のユーザーまたは任意のロールにスキーマ・オブジェクト権限を付与できます。
- 権限受領者は、他のユーザーに対してスキーマ・オブジェクト権限を Grant Option 付きまたは Grant Option なしで付与できます。
- 権限受領者が表に対するスキーマ・オブジェクト権限を Grant Option 付きで受領し、その権限受領者に CREATE VIEW または CREATE ANY VIEW システム権限がある場合、この権限受領者は、その表のビューを作成し、データベース上の任意のユーザーまたはロールにそのビューに対応する権限を付与できます。

スキーマにオブジェクトが含まれているユーザーには、対応するすべてのスキーマ・オブジェクト権限が自動的に Grant Option 付きで付与されます。

注意： Grant Option は、スキーマ・オブジェクト権限をロールに付与するときは無効です。ロールの権限受領者がロールを介して受領したオブジェクト権限を伝播することができないように、Oracle では、ロールを介したスキーマ・オブジェクト権限の伝播を防止しています。

スキーマ・オブジェクト権限の付与に必要な権限 スキーマ・オブジェクト権限を付与するには、権限付与者が次のいずれかの条件に該当している必要があります。

- 指定されたスキーマ・オブジェクトの所有者である。
- 該当するスキーマ・オブジェクト権限を Grant Option 付きで付与されている。
- オブジェクト権限を付与および取り消す所有者権限の委任が可能な GRANT ANY OBJECT PRIVILEGE システム権限を所有している。

GRANT ANY OBJECT PRIVILEGE システム権限の使用

GRANT ANY OBJECT PRIVILEGE システム権限を使用すると、複数のスキーマにわたる特定の準備タスクおよび構成タスクの処理が簡単になります。この権限を使用して処理を行うと、オブジェクトの所有者がした処理のように行われますが、監査レコードは実際のユーザーを示します。

GRANT ANY OBJECT PRIVILEGE システム権限を所有するユーザーは、すべてのオブジェクト権限を他のユーザーに付与できます。権限付与を行うユーザーが、Grant Option で指定されたオブジェクト権限をすでに所有している場合、権限付与は通常の方法で行われます。権限付与を行うユーザーが、付与する特定のオブジェクト権限を所有していない場合、権限付与はオブジェクトの所有者がした処理のように行われます。

他のシステム権限と同様に、GRANT ANY OBJECT PRIVILEGE は、With Admin Option 付きでこの権限を所有するユーザーのみが付与できます。

スキーマ・オブジェクト権限の取消し

スキーマ・オブジェクト権限は、SQL コマンド REVOKE を使用して取り消すことができます。たとえば、権限付与者は、次の文を入力して、ユーザー JWARD および TSMITH から EMP_TAB 表の SELECT 権限および INSERT 権限を取り消すことができます。

```
REVOKE SELECT, INSERT ON Emp_tab FROM jward, tsmith;
```

権限付与者は、次の文を入力して、ロール HUMAN_RESOURCES に対して付与した DEPT_TAB 表のすべての権限を取り消すこともできます。

```
REVOKE ALL ON Dept_tab FROM human_resources;
```

この文は、付与されている権限が 1 つのみの場合でも有効です。前述の文は、権限付与者が許可した権限のみを取り消し、他のユーザーが付与した権限は取り消しません。スキーマ・オブジェクト権限の Grant Option は選択的に取り消すことはできません。まず、スキー

マ・オブジェクト権限を取り消して、次に、Grant Option なしで再付与する必要があります。ユーザーは、自スキーマ・オブジェクト権限を取り消すことはできません。

列選択のスキーマ・オブジェクト権限の取消し 前述のとおり、列固有の INSERT、UPDATE、REFERENCES 権限を表またはビューに付与できます。ただし、類似する REVOKE 文を使用して、列固有の権限を選択的に取り消すことはできません。そのかわり、権限付与者は、最初に表、ビューまたはシノニムのすべての列についてスキーマ・オブジェクト権限を取り消し、新しい列固有の権限を選択的に再付与する必要があります。

たとえば、ロール HUMAN_RESOURCES に、表 DEPT_TAB の DEPTNO 列および DNAME 列に対する UPDATE 権限が付与されているとします。この UPDATE 権限を DEPTNO 列からのみ取り消すには、次の 2 つの文を入力します。

```
REVOKE UPDATE ON Dept_tab FROM human_resources;
GRANT UPDATE (Dname) ON Dept_tab TO human_resources;
```

REVOKE 文によって、ロール HUMAN_RESOURCES から DEPT_TAB 表のすべての列に対する UPDATE 権限が取り消されます。GRANT 文が、ロール HUMAN_RESOURCES に対して、DNAME 列に関する UPDATE 権限を再付与します。

REFERENCES スキーマ・オブジェクト権限の取消し REFERENCES オブジェクト権限の受領者が、この権限を使用して外部キー制約（現行の制約）を作成した場合、権限付与者は、REVOKE 文に CASCADE CONSTRAINTS オプションを指定することによってのみ、その権限を取り消すことができます。

```
REVOKE REFERENCES ON Dept_tab FROM jward CASCADE CONSTRAINTS;
```

CASCADE CONSTRAINTS オプションを指定すると、取り消された REFERENCES 権限を使用するように現在定義されている外部キー制約が削除されます。

スキーマ・オブジェクト権限を取り消すために必要な権限 スキーマ・オブジェクト権限を取り消すユーザーは、通常、取消し対象となる権限の付与者である必要があります。ただし、システム権限 GRANT ANY OBJECT PRIVILEGE を持つ別のユーザーも取消しを正常に実行できます。これは、取り消されるオブジェクト権限が、オブジェクトの元の所有者によって権限を付与されたとみなされるためです。取り消されるオブジェクト権限が他のユーザーによって付与された場合、システム権限 GRANT ANY OBJECT PRIVILEGE では元の所有者としての権限付与（または取消し）以外は実行できないため、そのエンティティによってのみ取消しが可能になります。

GRANT ANY OBJECT PRIVILEGE システム権限の取消し GRANT ANY OBJECT PRIVILEGE システム権限を所有するユーザーは、権限所有者が付与したすべての指定されたオブジェクト権限を取り消すか、GRANT ANY OBJECT PRIVILEGE を所有する一部のユーザーによって行われた権限付与を、権限所有者にかわって取り消すことができます。ただし、権限を付与されたユーザーは、他の任意のユーザーによって行われた権限付与を取り消すことはできません。

GRANT ANY OBJECT PRIVILEGE システム権限を使用した権限付与は、オブジェクト所有者によって実行されているように見えます。権限付与者が GRANT ANY OBJECT PRIVILEGE を使用してオブジェクト権限を付与し、このシステム権限が後で取り消された場合、権限付与者は、付与されたオブジェクト権限を取り消すことはできません。ただし、そのオブジェクト権限は、権限所有者または GRANT ANY OBJECT PRIVILEGE を持つ他のユーザーによって取り消すことができます。

通常の SQL REVOKE 構文は、取り消す権限の権限受領者を指定し、権限付与者は指定しません。これは、REVOKE を実行するユーザーは権限付与者であると想定されるためです。GRANT ANY OBJECT PRIVILEGE システム権限の導入によって、暗黙の権限付与者は、REVOKE を実行するユーザーまたはオブジェクトの所有者のいずれかになります。オブジェクト権限が、オブジェクトの所有者および REVOKE を実行するユーザーの両方によって付与された場合は、REVOKE を実行するユーザーによって付与されたオブジェクト権限のみを取り消します。たとえば、GRANT ANY OBJECT PRIVILEGE システム権限を所有するユーザー SCOTT が他のユーザーからオブジェクト権限を削除するとします。SCOTT が以前にこれと同じオブジェクト権限を付与していた場合、SCOTT による権限付与は取り消されます。SCOTT ではなく、権限所有者がそのオブジェクト権限を付与した場合、その権限所有者からのオブジェクト権限は取り消されます。それ以外の場合、REVOKE は実行できません。

スキーマ・オブジェクト権限の取消しの波及効果 スキーマ・オブジェクト権限を取り消すと、次のような様々な波及効果が発生する可能性があるため、REVOKE 文を発行する前に、必ず十分に検討してください。

- DML オブジェクト権限を取り消すと、その DML オブジェクト権限に依存するスキーマ・オブジェクト定義が影響を受ける可能性があります。たとえば、TEST プロシージャのプロシージャ本体に、EMP_TAB 表のデータを問い合せる SQL 文が含まれているとします。EMP_TAB 表の SELECT 権限が TEST プロシージャの所有者から取り消されると、そのプロシージャを正常に実行できなくなります。
- ALTER または INDEX オブジェクト権限を取り消しても、ALTER および INDEX の DDL オブジェクト権限を必要とするスキーマ・オブジェクト定義には影響しません。たとえば、別のユーザーの表に索引を作成したユーザーから INDEX 権限を取り消しても、その索引は存在します。
- 表に対する REFERENCES 権限をユーザーから取り消すと、そのユーザーによって定義され、削除された REFERENCES 権限を必要とするすべての外部キー整合性制約が、自動的に削除されます。たとえば、ユーザー JWARD に DEPT_TAB 表の DEPTNO 列に対する REFERENCES 権限が付与されていて、DEPTNO 列を参照する外部キーを EMP_TAB 表の DEPTNO 列に対して作成するとします。DEPT_TAB 表の DEPTNO 列に対する REFERENCES 権限を取り消すと、EMP_TAB 表の DEPTNO 列に対する外部キー制約が権限の取消し操作中に削除されます。
- 権限付与者のオブジェクト権限が取り消されると、Grant Option を使用して伝播したスキーマ・オブジェクト権限付与が取り消されます。たとえば、USER1 には SELECT オブジェクト権限が Grant Option を使用して付与され、USER1 は USER2 に対して EMP_TAB 表の SELECT 権限を付与するとします。その後、SELECT 権限が USER1 から取り消されます。この取消しは USER2 にも波及します。USER1 および USER2 の取り消

された SELECT 権限に依存するすべてのスキーマ・オブジェクトにも影響する可能性があります。

権限付与が依存オブジェクトに与える影響 スキーマ・オブジェクトに対して GRANT 文を発行すると、そのオブジェクトの「last DDL time」属性が変更されます。これによって、依存スキーマ・オブジェクト、特にこのスキーマ・オブジェクトを参照する PL/SQL パッケージ本体が無効になる可能性があります。その場合には、再コンパイルが必要です。

ユーザー・グループ PUBLIC に対する権限付与および取消し

権限およびロールを、ユーザー・グループ PUBLIC に対して付与および取り消すことができます。PUBLIC は、すべてのデータベース・ユーザーがアクセス可能であるため、PUBLIC に付与された権限およびロールには、すべてのデータベース・ユーザーがアクセスできます。

ある権限またはロールをすべてのデータベース・ユーザーが必要とする場合のみ、その権限またはロールを PUBLIC に付与してください。ここでも、一般的な規則として、各データベース・ユーザーには現在のタスクの正常な実行に必要な権限のみが付与されていることをお薦めします。

> **注意：** デフォルトで PUBLIC に付与されている特定の権限およびロールが不要な場合もあります。システム管理者が PUBLIC への権限付与を再確認し、不要な権限を取り消すことをお薦めします。

この項では、ユーザー・グループ PUBLIC に対する付与および取消しについて説明します。内容は次のとおりです。

- PUBLIC からのセキュリティの低いパッケージの取消し
- PUBLIC からの取消しの波及効果
- 権限付与および取消しが有効になる場合

PUBLIC からのセキュリティの低いパッケージの取消し

すべての不要な権限、権限付与およびロールは、PUBLIC から取り消す必要があります。すべてのデータベース・ユーザーは、PUBLIC に付与された権限を実行できます。このような権限には、様々な PL/SQL パッケージの EXECUTE が含まれます。これらのパッケージでは、最小限の権限のみを付与されているユーザーが、直接アクセスを許可されていないパッケージにアクセスしたり、そのパッケージを実行することを許可される場合があります。セキュリティが低い可能性があるパッケージには、次のものが含まれます。

パッケージ	EXECUTE を PUBLIC に付与しない理由
UTL_SMTP	任意のユーザーによる電子メール・メッセージの送受信を許可します。このパッケージを PUBLIC に付与すると、電子メール・メッセージの不当な送受信が許可される場合があります。
UTL_TCP	データベース・サーバーから着信ネットワーク・サービスへの発信ネットワーク接続の確立を許可します。このため、データベース・サーバーと待機中ネットワーク・サービスがデータの送受信を行います。
UTL_HTTP	データベースによる HTTP を介したデータの要求および取得を許可します。このパッケージを PUBLIC に付与すると、不当な Web サイトへの HTML フォームによるデータの送信が許可される場合があります。
UTL_FILE	正しく構成されていない場合、ホスト・オペレーティング・システム上のすべてのファイルへのテキスト・レベルのアクセスを許可します。正しく構成されている場合でも、このパッケージは、コール側アプリケーションを識別せず、他のアプリケーションが書き込んだ場所と同じ場所に任意のデータを書き込む場合があります。
DBMS_RANDOM	ユーザーによる格納データの暗号化を許可します。ほとんどのユーザーがデータを暗号化する権限を持たないようにする必要があります。キーが確実に保証、格納および管理されていないと、暗号化データはリカバリできない場合があります。このパッケージの EXECUTE をロールに付与し、データを暗号化する必要があるユーザーにそのロールを付与することをお薦めします。

これらのパッケージは、それらを必要とし、正しい構成と使用を保証するアプリケーションに対しては有効ですが、その他のアプリケーションには不適切または不要な場合があります。必要に応じて、PUBLIC およびデータベース・ユーザーからこれらのパッケージを取り消してください。

PUBLIC からの取消しの波及効果

取り消される権限によっては、PUBLIC からの取消しによって深刻な影響を受ける可能性があります。DML 操作に関連する権限を PUBLIC から取り消した場合（たとえば、SELECT ANY TABLE、UPDATE ON EMP_TAB など）、データベース内のすべてのプロシージャ、ファンクションおよびパッケージを使用するには、それらを再認可する必要があります。したがって、DML 関連の権限を PUBLIC に付与する場合は、十分な注意が必要です。

権限付与および取消しが有効になる場合

付与される権限または取り消される権限によって、権限付与または取消しが有効になる時点は異なります。

- ユーザー、ロールまたは PUBLIC に対する権限（システムおよびスキーマ・オブジェクト）付与 / 取消しは、すべてすぐに有効になります。

- ユーザー、他のロールまたは PUBLIC に対するロールの付与 / 取消しは、現在のユーザー・セッションが付与 / 取消し後に SET ROLE 文を発行してそのロールを再び使用可能にしたとき、または付与 / 取消し後に新しいユーザー・セッションが作成されたときにのみ、すべて有効になります。

  参照：『Oracle9*i* データベース管理者ガイド』の「権限とロールに関する情報の表示」を参照してください。

12

アプリケーション・セキュリティ・ポリシーの実装

この章では、アプリケーション・セキュリティ・ポリシーの実装方法について説明します。内容は次のとおりです。

アプリケーション・コンテキストの概要

アプリケーション・コンテキストを使用すると、ユーザーのセッション情報の特定の局面でアプリケーションを作成できます。また、アプリケーションがアクセス制御（特にファイングレイン・アクセス・コントロール）を施行するために使用する属性の定義方法、設定方法およびアクセス方法を提供します。

ほとんどのアプリケーションには、アクセスが制限される基になる情報が含まれます。たとえば、受注アプリケーションでは、顧客のアクセスは自分の注文（ORDER_NUMBER）および顧客番号（CUSTOMER_NUMBER）に制限されます。これらは、セキュリティ属性として使用できます。

Human Resources（人材管理）アプリケーションを実行しているユーザーを考えてみます。アプリケーションの初期化プロセスの中には、ユーザー ID に基づいてユーザーの職種を判断する部分が含まれています。この職種 ID が、Human Resources アプリケーションのコンテキストの一部になります。これは、セッションを通してユーザーがどのデータにアクセスできるかに影響します。

この項では、アプリケーション・コンテキストの使用について説明します。内容は次のとおりです。

- アプリケーション・コンテキストの機能
- ファイングレイン・アクセス・コントロールでのアプリケーション・コンテキストの使用方法
- ユーザー・モデルおよび仮想プライベート・データベース
- Oracle Policy Manager による仮想プライベート・データベース・ポリシーの作成
- アプリケーション・コンテキストの使用方法
- 例 : ファイングレイン・アクセス・コントロール関数内のアプリケーション・コンテキスト
- 自動再解析
- グローバルにアクセスされるアプリケーション・コンテキストの概要
- アプリケーション・コンテキストの外部での初期化
- アプリケーション・コンテキストのグローバルな初期化

アプリケーション・コンテキストの機能

アプリケーション・コンテキストは、次の重要なセキュリティ機能を提供します。

- 各アプリケーションへの属性の指定
- セキュリティ妥当性チェックの提供
- USERENV ネームスペースを介した事前定義属性へのアクセスの提供

- 外部化されたアプリケーション・コンテキスト

各アプリケーションへの属性の指定

各アプリケーションには、独自の属性を持つ独自のコンテキストを指定できます。たとえば、General Ledger（相勘定元帳）、Order Entry（受注）および Human Resources（人材管理）という 3 つのアプリケーションがあるとします。各アプリケーションには、異なる属性を指定することができます。そのため、次のことが可能になります。

- General Ledger アプリケーション・コンテキストには、属性 SET_OF_BOOKS および TITLE を指定できます。
- Order Entry アプリケーション・コンテキストには、属性 CUSTOMER_NUMBER を指定できます。
- Human Resources アプリケーション・コンテキストには、属性 ORGANIZATION_ID、POSITION および COUNTRY を指定できます。

いずれの場合も、アプリケーション・コンテキストを厳密なセキュリティ・ニーズに適応させることができます。

セキュリティ妥当性チェックの提供

使用されている一連の帳簿に基づいてアクセスを制御する General Ledger アプリケーションがあるとします。このアプリケーションにアクセスするユーザーが、作業対象の帳簿を 01 から 02 に変更した場合、アプリケーション・コンテキストによって、次のことが保証されます。

- 02 が有効な帳簿であること。
- このユーザーには帳簿 02 をアクセスする権限があること。

妥当性チェック機能は、アプリケーションのメタデータ表をチェックして前述の判断を行い、組み合された属性がセキュリティ・ポリシー全体に沿っていることを確認します。このセキュリティ妥当性チェックなしでユーザーがコンテキストの属性を変更しないように、Oracle では、コンテキストを実装する特定のパッケージのみが属性を変更することが保証されています。

USERENV ネームスペースを介した事前定義属性へのアクセスの提供

Oracle9*i* では、組込みアプリケーション・コンテキスト・ネームスペース USERENV が提供されています。これによって、事前定義属性にアクセスできます。これらの属性は、セッション・プリミティブで、データベースはユーザー・セッションに関する情報を獲得します。たとえば、ユーザーが接続元である IP アドレス、ユーザー名およびプロキシ・ユーザー名（ユーザー接続が中間層でプロキシ化されている場合）は、USERENV アプリケーション・コンテキストを介して、すべて事前定義属性として使用可能です。

事前定義属性は、アクセス制御に非常に役に立ちます。たとえば、OCI または Thick JDBC を介して軽量ユーザー・セッションを作成する 3 層アプリケーションを使用している場合、

USERENV アプリケーション・コンテキストの PROXY_USER 属性にアクセスして、ユーザー・セッションが中間層アプリケーションによって作成されたかどうかを判断できます。ポリシー関数を使用すると、ユーザーがプロキシ化されている接続に対してのみ、そのユーザーはデータにアクセスできます。ユーザーがプロキシ化されていない場合（そのユーザーがデータベースに直接接続している場合）、ユーザーはどのデータにもアクセスできません。

VPD 内で PROXY_USER 属性を使用してユーザーが特定の中間層アプリケーションからのみデータにアクセスすることを保証できますが、もう 1 つの方法として、安全なアプリケーション・ロールを作成することもできます。ユーザーがデータベースにアクセスする際に HRAPPSERVER を介してプロキシ化されることを、各ポリシーではなく、安全なアプリケーション・ロールによって保証できます。

事前定義属性は、USERENV アプリケーション・コンテキストを介してアクセスできますが、変更はできません。表 12-1 に、事前定義属性を示します。

次の構文を使用して、カレント・セッションに関する情報を戻します。

```
SYS_CONTEXT('userenv', 'attribute')
```

注意： USERENV アプリケーション・コンテキスト・ネームスペースは、データベースの以前のリリースで提供されている USERENV 関数にかわる機能です。

参照： USERENV ネームスペースおよびその事前定義属性の詳細は、『Oracle9*i* SQL リファレンス』の「SYS_CONTEXT」を参照してください。

表 12-1 USERENV ネームスペースの主な事前定義属性

事前定義属性	意味
TERMINAL	カレント・セッションのクライアントのオペレーティング・システム識別子を戻します。TCP/IP では「仮想」です。
LANGUAGE	セッションが現在使用している言語および地域を、データベース・キャラクタ・セットとともに、次の形式で戻します。*language_territory.characterset*
LANG	言語名の略称を戻します。
SESSIONID	監査セッション識別子を戻します。
INSTANCE	現行のインスタンスのインスタンス識別番号を戻します。
ENTRYID	使用可能な監査エントリ識別子を戻します。

表 12-1 USERENV ネームスペースの主な事前定義属性 （続き）

事前定義属性	意味
ISDBA	DBA ロールを使用可能にしている場合は TRUE を戻します。それ以外の場合は FALSE を戻します。
CLIENT_INFO	DBMS_APPLICATION_INFO パッケージを使用して、アプリケーションによって格納されるユーザー・セッション情報（最大 64 バイト）を戻します。
NLS_TERRITORY	カレント・セッションの地域を戻します。
NLS_CURRENCY	カレント・セッションの通貨記号を戻します。
NLS_CALENDAR	カレント・セッションの日付に使用する NLS カレンダを戻します。
NLS_DATE_FORMAT	カレント・セッションの現行の日付書式を戻します。
NLS_DATE_LANGUAGE	カレント・セッションの日付表現に使用する言語を戻します。
NLS_SORT	ソート・ベースがバイナリか言語かを示します。
CURRENT_USER	カレント・セッションがその権限下にあるユーザーのユーザー名を戻します。ストアド・プロシージャ（実行者権限プロシージャなど）内の SESSION_USER と異なる場合があります。
CURRENT_USERID	カレント・セッションがその権限下にあるユーザーのユーザー ID を戻します。ストアド・プロシージャ（実行者権限プロシージャなど）内の SESSION_USERID と異なる場合があります。
SESSION_USER	現行のユーザーが認証されるデータベース・ユーザー名を戻します。
SESSION_USERID	現行のユーザーが認証されるデータベース・ユーザー名の識別子を戻します。
CURRENT_SCHEMA	カレント・セッションで使用されているデフォルトのスキーマ名を戻します。これは、ALTER SESSION SET SCHEMA 文で変更できます。
CURRENT_SCHEMAID	カレント・セッションで使用されているデフォルトのスキーマの識別子を戻します。これは、ALTER SESSION SET SCHEMAID 文で変更できます。
PROXY_USER	SESSION_USER の代理としてカレント・セッションを開いたデータベース・ユーザー（通常は中間層）の名前を戻します。
PROXY_USERID	SESSION_USER の代理としてカレント・セッションを開いたデータベース・ユーザー（通常は中間層）の識別子を戻します。
DB_DOMAIN	DB_DOMAIN 初期化パラメータで指定されているデータベースのドメインを戻します。

表 12-1 USERENV ネームスペースの主な事前定義属性 （続き）

事前定義属性	意味
DB_NAME	DB_NAME 初期化パラメータで指定されているデータベースの名前を戻します。
HOST	データベースが起動しているホスト・マシン名を戻します。
OS_USER	データベース・セッションを開始するクライアント・プロセスのオペレーティング・システム・ユーザー名を戻します。
EXTERNAL_NAME	データベース・ユーザーの外部名を戻します。
IP_ADDRESS	接続元のクライアント・マシンの IP アドレスを戻します（入手可能な場合）。
NETWORK_PROTOCOL	接続文字列（PROTOCOL=*protocol*）で指定されているプロトコルを戻します。
BG_JOB_ID	バックグラウンド・ジョブ ID を戻します。
FG_JOB_ID	フォアグラウンド・ジョブ ID を戻します。
AUTHENTICATION_TYPE	ユーザーがどのように認証されているか（DATABASE、OS、NETWORK、PROXY）を表示します。
AUTHENTICATION_DATA	ログイン・ユーザーを認証するために使用するデータを戻します。ユーザーが SSL を介して認証されている場合、またはユーザーの証明書がデータベースにプロキシ化されている場合は、ユーザーの X.509 証明書が含まれます。
CURRENT_SQL	ファイングレイン監査のイベント・ハンドラを呼び出す問合せの SQL 文です。イベント・ハンドラ内のみで有効です。
CLIENT_IDENTIFIER	セッションに対するユーザー定義のクライアント識別子です。
GLOBAL_CONTEXT_MEMORY	グローバル・アプリケーション・コンテキストによって使用される共有メモリーの容量（バイト単位）です。

外部化されたアプリケーション・コンテキスト

多くのアプリケーションは、ファイングレイン・アクセス・コントロールに使用される属性を、アクセス・コントロールに使用するデータベース・メタデータ表に格納します。たとえば、EMPLOYEES 表に、コスト・センター、職種、署名認証などのファイングレイン・アクセス・コントロールに有効な情報を含めることができます。ただし、多くの組織ではユーザー情報およびユーザー管理を Oracle Internet Directory などの LDAP ベースのディレクトリに集中化しています。これらの組織では、アクセス・コントロールに使用されるユーザー情報も集中化する必要があります。アプリケーション・コンテキスト属性を Oracle Internet Directory に格納し、1 人以上のエンタープライズ・ユーザーに割り当てることができます。これらの属性は、エンタープライズ・ユーザーがログインすると自動的に取得され、アプリケーション・コンテキストの初期化に使用されます。

注意： エンタープライズ・ユーザー・マネジメントは Oracle Advanced Security の機能です。

参照：

- 12-32 ページの「アプリケーション・コンテキストのグローバルな初期化」を参照してください。
- 『Oracle Advanced Security 管理者ガイド』を参照してください。

ファイングレイン・アクセス・コントロールでのアプリケーション・コンテキストの使用方法

セキュリティ・ポリシーをより簡単に実装するために、ファイングレイン・アクセス・コントロール関数内でアプリケーション・コンテキストを使用することができます。

注意： ファイングレイン・アクセス・コントロールとアプリケーション・コンテキストを組み合せて使用することを VPD といいます。

アプリケーション・コンテキストは、ファイングレイン・アクセス・コントロールの次の用途に使用できます。

- 保護データ・キャッシュとしてのアプリケーション・コンテキストの使用
- 特定の述語（セキュリティ・ポリシー）を戻すためのアプリケーション・コンテキストの使用
- 述語のバインド変数のような属性を指定するためのアプリケーション・コンテキストの使用

保護データ・キャッシュとしてのアプリケーション・コンテキストの使用

ファイングレイン・アクセス・コントロールのポリシー関数の中でアプリケーション・コンテキストをアクセスするということは、頻繁に使用する電話番号を書き留めて、電話の隣に貼っておくようなものです。こうしておくと、番号が必要なときは、探さなくても簡単に見つかります。

たとえば、ORDERS_TAB 表へのアクセスを顧客番号を基にして行うとします。顧客番号が必要になるたびにログインしたユーザーに問い合せるのではなく、アプリケーション・コンテキスト内に顧客番号を格納しておくことができます。このようにすると、顧客番号は必要なときに使用できます。

アプリケーション・コンテキストは、セキュリティ・ポリシーが複数のセキュリティ属性に基づく場合に特に効果的です。たとえば、述語が 4 つの属性（従業員番号、コスト・センター、職種、支出制限など）に基づくポリシー関数は、この情報を取得するために複数の副問合せを実行する必要があります。このデータがすべてアプリケーション・コンテキストを介して使用可能な場合、パフォーマンスは大きく向上します。

特定の述語（セキュリティ・ポリシー）を戻すためのアプリケーション・コンテキストの使用

アプリケーション・コンテキストを使用して、正しい述語（セキュリティ・ポリシー）を戻すことができます。

「顧客は自分の注文のみを参照でき、店員はすべての顧客のすべての注文を参照できる」というルールを規定している受注アプリケーションを考えます。この場合、2 つの異なるポリシーがあります。position 属性でアプリケーション・コンテキストを定義できます。この属性はポリシー関数内でアクセスでき、その属性の値に基づいて正しい述語を戻します。そのため、position が Clerk（店員）であるユーザーはすべての注文を取得できますが、Customer であるユーザーは自分のレコードのみ参照できます。

属性に対して特定の述語を戻すファイングレイン・アクセス・コントロール・ポリシーを設計するには、ポリシーを実装するファンクション内でアプリケーション・コンテキストにアクセスします。たとえば、顧客が自分のレコードのみを参照するように制限するには、ファイングレイン・アクセス・コントロールを使用して、ユーザーの問合せを動的に変更します。たとえば、次の文、

```
SELECT * FROM Orders_tab
```

が、次のように変更されます。

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

述語のバインド変数のような属性を指定するためのアプリケーション・コンテキストの使用

前述の例で、50,000 人の顧客を持ち、各顧客に対して戻される述語を同じにするとします。すべての顧客は同一のポリシーを共有し、自分の注文のみを参照できます。顧客間で異なるのは、顧客番号のみです。

アプリケーション・コンテキストを使用すると、50,000 人の顧客に適用するポリシー関数内で、1 つの述語を戻せます。その結果、1 つの共有カーソルが、顧客ごとに異なる形態で実行されます。これは、実行時に顧客番号が評価されるためです。この値は、顧客ごとに異なります。このようなアプリケーション・コンテキストの使用によって、ファイングレイン・セキュリティおよび最適なパフォーマンスが得られます。

SYS_CONTEXT 関数がバインド変数のように動作するのは、SYS_CONTEXT 引数が定数の場合のみであることに注意してください。

ユーザー・モデルおよび仮想プライベート・データベース

アプリケーションには様々なユーザー・モデルを含めることができますが、ユーザーごとにアクセスを制限するために、仮想プライベート・データベース（VPD）を使用する必要がある場合があります。Oracle は、ユーザーがデータベース・ユーザーか、データベースが認識できないアプリケーション・ユーザーかによって、異なる方法を提供します。アプリケーションはその方法を使用して、ユーザーごとにファイングレイン・アクセス・コントロールを施行できます。

アプリケーション・ユーザーがデータベース・ユーザーであるアプリケーションの場合、VPD の施行は比較的簡単です。ユーザーがデータベースに接続すると、アプリケーションはセッションごとのアプリケーション・コンテキストを設定できます。各セッションはユーザー名ごとに開始されるので、ユーザー Jane とユーザー John に対して異なるファイングレイン・アクセス・コントロール条件を簡単に施行できます。これはプロキシ認証を使用しても行えます。OCI または Thick JDBC の軽量セッションは個別のデータベース・セッションであり、独自のアプリケーション・コンテキストを持つことができるためです。

プロキシ認証は Enterprise User Security と統合できるため、VPD の施行に使用できる他の属性と同様、Oracle Internet Directory からユーザー・ロールを取得できます。

シングル・ユーザー（たとえば、One Big Application User）がすべてのユーザーのかわりにデータベースに接続するアプリケーションでも、ユーザーごとのファイングレイン・アクセス・コントロールを施行できます。アプリケーション開発者は、アプリケーション・ユーザー（たとえば、realuser）を表すコンテキスト属性を作成できます。すべてのデータベース・セッション（およびすべての監査レコード）は One Big Application User として開始されますが、各セッションは実際のユーザーに応じて異なる属性を持つことができます。このモデルは、ユーザーの数が限定されておりセッションを再利用する必要がないアプリケーションに最適です。データベースの観点からすると、各セッションは同一のデータベース・ユーザーとして作成されるので、ロールやデータベース監査などの有用性は、前述の理由によって大幅に低下します。

Web ベースのアプリケーションには、通常、何百人ものユーザーがおり、Web はステートレスです。（多数のユーザー要求に対するデータの取得をサポートするため）データベースへの接続が持続的に行われる場合がありますが、このような接続は Web ベースのユーザーごとに持つわけではありません。通常、Web ベースのアプリケーションはスケーラビリティを提供するため、ユーザーごとに異なるセッションを持つのではなく、接続を設定して再利用します。たとえば、Web ユーザー Jane および Ajit が中間層アプリケーションに接続すると、2 人のユーザーのかわりにこのアプリケーションが、使用するデータベースにセッションを確立します。通常、Jane も Ajit もデータベースには認識されません。アプリケーションが接続するユーザー名を切り替えるので、どの時点でも、セッションを使用しているのは Jane または Ajit のいずれかです。

Oracle9*i* VPD の機能を使用すると、個別のユーザー・セッションに対してアプリケーション・コンテキストを設定するのではなく、複数の接続が 1 つ以上のグローバル・アプリケーション・コンテキストにアクセスできるようになるため、接続プーリングが簡単になります。

アプリケーションは、グローバル・アプリケーション・コンテキストを参照するために、CLIENT_IDENTIFIER（個々のアプリケーション・ユーザー名またはグループ）を使用します。グローバル・アプリケーション・コンテキストは、セッションごとのアプリケーション・コンテキストを設定するのではなく、複数のセッションに共通のアプリケーション・コンテキストを再利用することによって、パフォーマンスを向上させるのみでなく、Web ベースのアプリケーションで VPD を使用するための高い柔軟性を提供します。CLIENT_IDENTIFIER はユーザー・セッションでも参照でき、USERENV ネーミング・コンテキストでアクセスできます。

CLIENT_IDENTIFIER は「アプリケーション・ユーザー名」の獲得に使用できるため、CLIENT_IDENTIFIER の使用はアプリケーション・ユーザー・プロキシとして機能します。グローバル・アプリケーション・コンテキストと組み合せて使用するために CLIENT_IDENTIFIER をデータベースに渡す機能は、OCI、Thick JDBC および Thin JDBC でサポートされています。OCI ベースの接続では、パフォーマンス向上のため、CLIENT_IDENTIFIER の変更は次の OCI コールに自動的に伝達されます。

アプリケーション・ユーザー・プロキシ認証は、グローバル・アプリケーション・コンテキストと組み合せて使用すると、アプリケーション作成時の柔軟性およびパフォーマンスを向上できます。たとえば、情報をビジネス・パートナに提供する Web ベースのアプリケーションに、ゴールド・パートナ、シルバー・パートナおよびブロンズ・パートナの 3 タイプのユーザーがあり、それぞれ使用可能な情報レベルが異なる場合を考えます。各ユーザーが個別のアプリケーション・コンテキストを持つ独自のセッションを設定するのではなく、アプリケーションが、ゴールド・パートナ、シルバー・パートナまたはブロンズ・パートナ用のグローバル・アプリケーション・コンテキストを設定し、正しいコンテキストのセッションを示すクライアント識別子を使用して、適切な型のデータを取得します。アプリケーションは、3 つのグローバル・コンテキストを 1 回のみ初期化し、クライアント識別子を使用して正しいアプリケーション・コンテキストにアクセスすることで、データ・アクセスを制限します。このように、各セッションのアプリケーション・コンテキストを個別に初期化するかわりに、セッションを再利用し、設定済のグローバル・アプリケーション・コンテキストに 1 回のみアクセスすることによって、パフォーマンスが向上します。

Oracle Policy Manager による仮想プライベート・データベース・ポリシーの作成

仮想プライベート・データベース（VPD）を実装する開発者は、DBMS_RLS パッケージを使用して、表およびビューにセキュリティ・ポリシーを適用できます。また、開発者は、CREATE CONTEXT コマンドを使用してアプリケーション・コンテキストを作成できます。

または、Oracle Enterprise Manager から Oracle Policy Manager の GUI にアクセスして、表やビューなどのスキーマ・オブジェクトにセキュリティ・ポリシーを適用したり、アプリケーション・コンテキストを作成することができます。Oracle Policy Manager は、セキュリティ・ポリシーおよびアプリケーション・コンテキストの管理に使いやすいインタフェースを提供するので、VPD の開発が簡単になります。

Oracle Policy Manager は Oracle Label Security 用の管理ツールです。Oracle Label Security は、機能的で独創的な VPD ポリシーを提供し、行レベルのセキュリティを実装する機能を

向上させます。Oracle Label Security では、ラベルベースのアクセス制御フレームワークというインフラストラクチャを提供します。これによって、ユーザーおよびデータにラベルを指定できます。また、ラベル・アクセスの判断に使用される 1 つ以上のカスタム・セキュリティ・ポリシーを作成できます。これらのポリシーは、プログラミング言語に関する知識がなくても実装できます。追加のコードを作成する必要はありません。1 つの手順で、特定の表にセキュリティ・ポリシーを適用できます。このように、Oracle Label Security を使用すると、データ・ラベリング・テクノロジを使用してファイングレイン・セキュリティ・ポリシーを簡単かつ効果的に実装できます。Oracle Label Security のラベル構造では、アプリケーション・データのみからは簡単に導出できない細分性および柔軟性が得られます。Oracle Label Security は、様々な状況で使用できる一般的なソリューションです。

VPD ポリシーを作成するには、ユーザーが、スキーマ名、表名（またはビュー名）、ポリシー名、述語を生成する関数名およびポリシーを適用する文の型（SELECT、INSERT、UPDATE、DELETE）を提供する必要があります。次に、Oracle Policy Manager が DBMS_RLS.ADD_POLICY 関数を実行します。コンテキストの名前およびコンテキストを実装するパッケージを提供して、アプリケーション・コンテキストを作成します。

参照： 『Oracle Label Security 管理者ガイド』を参照してください。

アプリケーション・コンテキストの使用方法

アプリケーション・コンテキストを使用するには、次の作業を行います。

- 作業 1: アプリケーションにコンテキストを設定する PL/SQL パッケージの作成
- 作業 2: 一意のコンテキストの作成および PL/SQL パッケージへの対応付け
- 作業 3: ユーザーがデータを取得する前のコンテキストの設定
- 作業 4: ポリシー関数でのコンテキストの使用

作業 1: アプリケーションにコンテキストを設定する PL/SQL パッケージの作成

ご使用のアプリケーションに、コンテキストを設定するファンクションを持つ PL/SQL パッケージを作成します。この項では例を示し、その後で SYS_CONTEXT 構文および動作について説明します。

注意： ユーザーのコンテキスト（EMPNO、GROUP、MANAGER などの情報）は、ユーザーがデータにアクセスする前に設定されるため、ログイン・トリガーを使用できます。

SYS_CONTEXT の例 次の例は、パッケージ app_security_context を作成します。

```
CREATE OR REPLACE PACKAGE App_security_context IS
   PROCEDURE Set_empno;
END;

CREATE OR REPLACE PACKAGE BODY App_security_context IS
   PROCEDURE Set_empno
   IS
   Emp_id NUMBER;
   BEGIN
    SELECT Empno INTO Emp_id FROM Emp_tab
       WHERE Ename = SYS_CONTEXT('USERENV',
                                 'SESSION_USER');
    DBMS_SESSION.SET_CONTEXT('app_context', 'empno', Emp_id);
   END;
END;
```

参照： 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

SYS_CONTEXT 構文 このファンクションの構文は、次のとおりです。

```
SYS_CONTEXT ('namespace', 'attribute', [length])
```

このファンクションは、コンテキスト・ネームスペースに現在対応付けられているパッケージに定義されている attribute の値を戻します。それぞれの文を実行するたびに 1 回評価され、最適化のためのタイプ・チェック中は定数のように扱われます。事前定義の USERENV ネームスペースを使用して、ユーザー ID、NLS パラメータなどの基本形コンテキストにアクセスできます。

注意： パラレル問合せ環境で SYS_CONTEXT を実行しようとすると、問合せエラーが戻されます。12-13 ページのパラレル問合せの SYS_CONTEXT の使用を参照してください。

参照： 12-3 ページの「USERENV ネームスペースを介した事前定義属性へのアクセスの提供」を参照してください。

属性の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

SYS_CONTEXT での動的 SQL の使用

> **注意：** この機能は、COMPATIBLE が 8.0 または 8.1 に設定されている場合に適用できます。

指定した問合せを実行する間にポリシーを変更するセッション中は、その問合せでは動的 SQL を使用する必要があります。これは、静的 SQL と動的 SQL が行う文の解析が異なるためです。

- 静的 SQL 文はコンパイル時に解析され、パフォーマンス上の理由から実行時には再解析されません。
- 動的 SQL 文は、実行されるたびに解析されます。

SQL 文のコンパイル時にはポリシー A を施行し、その後、ポリシー B に変更して文を実行する場合を考えます。静的 SQL では、ポリシー A が施行されたままです。文はコンパイル時に解析され、実行時には再解析されません。ただし、動的 SQL では、文は実行時に解析されるため、ポリシー B への変更が実行されます。

たとえば、次のポリシーを考えます。

```
EMPLOYEE_NAME = SYS_CONTEXT ('userenv', 'session_user')
```

「従業員名をデータベース・ユーザー名に一致させる」というポリシーが、SQL 述語の形式で表現されます。この述語は、基本的にはポリシーです。述語が変更された場合、正しい結果を生成するために、文は再解析される必要があります。

参照： 12-47 ページの「自動再解析」を参照してください。

パラレル問合せの SYS_CONTEXT の使用 SYS_CONTEXT が、パラレル問合せに埋め込まれている SQL 関数内部で使用されている場合、この関数はアプリケーション・コンテキストを選択できません。アプリケーション・コンテキストは、ユーザー・セッション内にのみ存在するためです。これらの機能を組み合せて使用するには、SYS_CONTEXT を問合せから直接コールする必要があります。

ユーザー ID を 5 に設定する、SQL 文内のユーザー定義ファンクションを考えます。

```
CREATE FUNC proc1 AS RETURN NUMBER;
BEGIN
     IF SYS_CONTEXT ('hr', 'id') = 5
     THEN RETURN 1; ELSE RETURN 2;
     END
END;
```

次の文を考えます。

```
SELECT * FROM EMP WHERE proc1( ) = 1;
```

この文を単一の問合せとして実行する場合（1 つのプロセスを使用して問合せ全体を実行する場合）、問題はありません。

ただし、この文をパラレル問合せとして実行する場合、パラレル実行サーバー（問合せスレーブ・プロセス）は、アプリケーション・コンテキスト情報を含むユーザー・セッションへはアクセスしません。この問合せでは、目的とする結果は生成されません。

SYS_CONTEXT 関数を問合せ内で使用する場合、問題はありません。たとえば、次のような場合です。

```
SELECT * FROM EMP WHERE SYS_CONTEXT ('hr', 'id') = 5;
```

この場合、この関数はバインド変数のように動作します。問合せコーディネータはアプリケーション・コンテキスト情報にアクセスでき、その情報をパラレル実行サーバーに渡します。

アプリケーション・コンテキストのバージョン化 文を実行する場合、Oracle9*i* では、SYS_CONTEXT によって設定されたアプリケーション・コンテキスト全体のスナップショットが取られます。問合せの存続期間中、コンテキストはその問合せのすべてのフェッチに対して同じままです。

ユーザー（または関数）がコンテキストを問合せ内で変更しようとすると、変更は現行の問合せでは有効になりません。この場合、SYS_CONTEXT を使用すると、セッションに変数を格納できます。

作業 2: 一意のコンテキストの作成および PL/SQL パッケージへの対応付け

この作業を実行するには、CREATE CONTEXT 文を使用します。コンテキストはそれぞれ一意の属性を持つ必要があり、ネームスペースに属する必要があります。つまり、コンテキスト名は、スキーマ内のみでなくデータベース内でも一意である必要があります。コンテキストは、常にスキーマ SYS によって所有されます。

次に例を示します。

```
CREATE CONTEXT order_entry USING oe_context;
```

order_entry はコンテキスト・ネームスペース、oe_context はコンテキスト・ネームスペースに属性を設定できるトラステッド・パッケージです。

コンテキストを作成した後、DBMS_SESSION.SET_CONTEXT パッケージを使用してコンテキスト属性を設定または再設定できます。設定した属性の値は、再設定するまでまたはユーザーがセッションを終了するまでそのまま残ります。

コンテキスト属性を設定できるのは、CREATE CONTEXT 文に名前を指定したトラステッド・プロシージャ内のみです。このため、ユーザーは、適切な属性妥当性チェックなしで、故意にコンテキスト属性を変更することはできません。

または、Oracle Policy Manager の GUI を使用してコンテキストを作成し、PL/SQL パッケージに対応付けることができます。Oracle Enterprise Manager から Oracle Policy Manager にアクセスすると、データベース・オブジェクトにポリシーを適用したり、アプリケーション・コンテキストを作成することができます。また、Oracle Label Security のポリシーの作成および管理に使用できます。

作業 3: ユーザーがデータを取得する前のコンテキストの設定

ログイン時に必ずイベント・トリガーを使用して、セッション情報をコンテキストに挿入します。これによって、ユーザーのセキュリティ制限属性をデータベースに設定して評価し、データベースによる適切なセキュリティの決定が可能になります。

その他の考慮点については、一連の帳簿が変更された場合または職位が定期的に変更される場合に考慮します。これらの場合には、新しい属性値はすぐに有効にならない場合があり、その属性を有効にするためには、強制的にカーソルの再解析を行う必要があります。

参照 :

- 12-2 ページの「アプリケーション・コンテキストの機能」を参照してください。
- 12-25 ページの「グローバルにアクセスされるアプリケーション・コンテキストの概要」を参照してください。

作業 4: ポリシー関数でのコンテキストの使用

これまでの手順で、コンテキストおよび PL/SQL パッケージを設定しました。次に、ポリシー関数でアプリケーション・コンテキストを使用し、異なるコンテキスト値に基づくポリシーを決定します。

例 : ファイングレイン・アクセス・コントロール関数内のアプリケーション・コンテキスト

この項では、ファイングレイン・アクセス・コントロール関数内でアプリケーション・コンテキストを使用する例を示します。

- 例 1: ポリシーの実装
- 例 2: アプリケーションによるユーザー・アクセスのコントロール
- 例 3: イベント・トリガー、アプリケーション・コンテキスト、ファイングレイン・アクセス・コントロールおよび権限のカプセル化

例 1: ポリシーの実装

この例では、アプリケーション・コンテキストを使用して、「顧客は自分の注文のみを参照できる」というポリシーを実装します。

この例では、アプリケーション作成について、次の手順で説明します。

- 手順 1. アプリケーションにコンテキストを設定する PL/SQL パッケージの作成
- 手順 2. アプリケーション・コンテキストの作成
- 手順 3. パッケージ内のアプリケーション・コンテキストへのアクセス
- 手順 4. 新しいセキュリティ・ポリシーの作成

この例の手順は、ユーザーと顧客の 1 対 1 の関係を想定しています。ユーザーの顧客番号（Cust_num）を検索し、アプリケーション・コンテキスト内に顧客番号をキャッシュします。受注コンテキスト（order_entry_ctx）の cust_num 属性は、後でセキュリティ・ポリシーの関数内で参照できます。

初期コンテキストの設定にはログイン・トリガーを使用できることに注意してください。

手順 1. アプリケーションにコンテキストを設定する PL/SQL パッケージの作成

次のようにパッケージを作成します。

```
CREATE OR REPLACE PACKAGE apps.oe_ctx AS
   PROCEDURE set_cust_num ;
END;

CREATE OR REPLACE PACKAGE BODY apps.oe_ctx AS
   PROCEDURE set_cust_num IS
     custnum NUMBER;
   BEGIN
      SELECT cust_no INTO custnum FROM customers WHERE username =
         SYS_CONTEXT('USERENV', 'session_user');
     /* SET cust_num attribute in 'order_entry' context */
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
   END set_cust_num;
 END;
```

注意： この例では、エラー処理を行っていません。

SYS_CONTEXT('userenv', *session_primitive*) を使用して、事前定義属性（セッション・ユーザーなど）にアクセスできます。

詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

手順 2. アプリケーション・コンテキストの作成

次の文を入力して、アプリケーション・コンテキストを作成します。

```
CREATE CONTEXT Order_entry USING Apps.Oe_ctx;
```

また、Oracle Policy Manager を使用してアプリケーション・コンテキストを作成することもできます。

手順 3. パッケージ内のアプリケーション・コンテキストへのアクセス

データベース・オブジェクトにセキュリティ・ポリシーを実装するパッケージ内のアプリケーション・コンテキストにアクセスします。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE PACKAGE Oe_security AS
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2)
RETURN VARCHAR2;
END;
```

パッケージ本体が、ORDERS_TAB 表に対する SELECT 文に動的な述語を追加します。この述語は、顧客表に対する副問合せのかわりに cust_num コンテキスト属性をアクセスすることによって、戻された注文をユーザーの顧客番号の注文のみに制限します。

```
CREATE OR REPLACE PACKAGE BODY Oe_security AS

/* limits select statements based on customer number: */
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2) RETURN VARCHAR2
IS
    D_predicate VARCHAR2 (2000)
    BEGIN
     D_predicate = 'cust_no = SYS_CONTEXT("order_entry", "cust_num")';
     RETURN D_predicate;
    END Custnum_sec;
END Oe_security;
```

手順 4. 新しいセキュリティ・ポリシーの作成

次のようにポリシーを作成します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT sys/change_on_install AS sysdba;
CREATE USER secusr IDENTIFIED BY secusr;
```

```
DBMS_RLS.ADD_POLICY ('scott', 'orders_tab', 'oe_policy', 'secusr',
                     'oe_security.custnum_sec', 'select')
```

この文は、スキーマ SCOTT 内に表示する OE_POLICY というポリシーを ORDERS_TAB 表に追加します。SECUSR.OE_SECURITY.CUSTNUM_SEC 関数がこのポリシーを実装します。この関数は SECUSR スキーマに格納され、SELECT 文のみに適用されます。

これで、ORDERS_TAB 表に対する顧客別の SELECT 文では、自動的にその顧客の注文のみが戻されます。動的な述語によって、次のユーザーの文、

```
SELECT * FROM Orders_tab;
```

が、次のように変更されます。

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT('order_entry','cust_num');
```

この例では、次のことに注意してください。

- 実際には、ユーザーの職位に基づく述語がいくつかある場合があります。たとえば、営業担当者は自分の担当顧客のレコードのみを参照でき、注文入力オペレータはすべての顧客注文を参照できます。custnum_sec 関数は、ユーザーの職位のコンテキスト値に基づいて異なる述語を戻すように拡張できます。
- ファイングレイン・アクセス・コントロール・パッケージ内でアプリケーション・コンテキストを使用することによって、実際には解析済の文の中にバインド変数が指定されます。次に例を示します。

  ```
  SELECT * FROM Orders_tab
     WHERE Custno = SYS_CONTEXT('order_entry', 'cust_num')
  ```

  この文は完全に解析および最適化されますが、ORDER_ENTRY コンテキストに対するユーザーの CUST_NUM 属性の評価は、実行時に行われます。これは、最適化された文は、その文を実行するユーザーごとに異なる形態で実行されるという利点があることを表します。

  注意： この例の関数のパフォーマンスは、CUST_NO に索引を作成することで、さらに向上させることができます。

- コンテキスト属性は、データベース表または表のデータ、または LDAP を使用するディレクトリ・サーバーのデータに基づいて設定できます。

参照： 動的に生成された述語の中でアプリケーション・コンテキストを使用するこの例と、述語の中に副問合せを使用する 12-39 ページの「ファイングレイン・アクセス・コントロールの動作」を比較してください。

第 15 章「トリガーの使用」を参照してください。

例 2: アプリケーションによるユーザー・アクセスのコントロール

この例では、アプリケーション・コンテキストを使用して Human Resources アプリケーションでのユーザー・アクセスを制御します。次の 3 つの作業を順に説明します。それぞれの作業の詳細は、その後で説明します。

- 手順 1. コンテキストを設定する PL/SQL パッケージの作成
- 手順 2. コンテキストの作成およびパッケージへの対応付け
- 手順 3. アプリケーションに対する初期化スクリプトの作成

この例では、Human Resources アプリケーションのアプリケーション・コンテキストが HR_CTX ネームスペースに割り当てられていると想定しています。

手順 1. コンテキストを設定する PL/SQL パッケージの作成

アプリケーションにコンテキストを設定する多数のファンクションを持つ PL/SQL パッケージを作成します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE OR REPLACE PACKAGE apps.hr_sec_ctx IS
   PROCEDURE set_resp_id (respid NUMBER);
   PROCEDURE set_org_id (orgid NUMBER);
  /* PROCEDURE validate_respid (respid NUMBER); */
  /* PROCEDURE validate_org_id (orgid NUMBER); */
END hr_sec_ctx;
```

APPS は、このパッケージを所有するスキーマです。

```
CREATE OR REPLACE PACKAGE BODY apps.hr_sec_ctx IS
/* function to set responsibility id */
PROCEDURE set_resp_id (respid NUMBER) IS
BEGIN

/* validate respid based on primitive and other context */
/*    validate_respid (respid); */

/* set resp_id attribute under namespace 'hr_ctx'*/
    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'resp_id', respid);
END set_resp_id;

/* function to set organization id */
PROCEDURE set_org_id (orgid NUMBER) IS

BEGIN
/* validate organization ID */
```

```
/*    validate_org_id(orgid); /*
/* set org_id attribute under namespace 'hr_ctx' */
    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'org_id', orgid);
END set_org_id;

/* more functions to set other attributes for the HR application */
END hr_sec_ctx;
```

手順 2. コンテキストの作成およびパッケージへの対応付け

次に例を示します。

```
CREATE CONTEXT Hr_ctx USING Apps.Hr_sec_ctx;
```

手順 3. アプリケーションに対する初期化スクリプトの作成

パッケージ HR_SEC_CTX の実行権限が、アプリケーションを実行するスキーマに付与されているとします。スクリプトの一部でコールが実行され、HR_CTX コンテキストの様々な属性が設定されます。ここでは、コンテキストがどのように決定されるかは示しません。通常は、基本形コンテキストまたは他の導出コンテキストに基づいて行われます。

```
APPS.HR_SEC_CTX.SET_RESP_ID(1);
APPS.HR_SEC_CTX.SET_ORG_ID(101);
```

このアプリケーション・コンテキストに基づくデータ・アクセス制御には、SYS_CONTEXT 関数を使用できます。たとえば、属性 ORG_ID に基づいて行へのアクセスを制限するビューによって、ベース表 HR_ORGANIZATION_UNIT を保護できます。

> **注意：** 次のデータ構造を設定しないと機能しない例もあります。
>
> ```
> CREATE TABLE hr_organization_unit (organization_id NUMBER);
> ```

```
CREATE VIEW Hr_organization_secv AS
   SELECT * FROM hr_organization_unit
      WHERE Organization_id = SYS_CONTEXT('hr_ctx','org_id');
```

例 3: イベント・トリガー、アプリケーション・コンテキスト、ファイングレイン・アクセス・コントロールおよび権限のカプセル化

この例では、Oracle9*i* で提供されている次のセキュリティ機能の使用方法を示します。

- イベント・トリガー
- アプリケーション・コンテキスト
- ファイングレイン・アクセス・コントロール
- ストアド・プロシージャへの権限のカプセル化

この例では、セキュリティ・ポリシーを DIRECTORY という表に対応付けます。この表には、次の列が含まれています。

列	説明
EMPNO	各社員の識別番号
MGRID	各社員の管理者の社員識別番号
RANK	社内階層における社員の職位

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Payroll(
   Srate  NUMBER,
   Orate  NUMBER,
   Acctno NUMBER,
   Empno  NUMBER,
   Name   VARCHAR2(20));
CREATE TABLE Directory_u(
   Empno NUMBER,
   Mgrno NUMBER,
   Rank  NUMBER);
CREATE SEQUENCE Empno_seq
CREATE SEQUENCE Rank_seq
```

この表に対応付けられるセキュリティ・ポリシーには、次の 2 つの要素があります。

- 特定の EMPNO に対する MGRID の検索はすべてのユーザーが行えます。これを実装するには、表に対する SELECT を実行する定義者権限パッケージを Human Resources スキーマ（HR）に作成します。
- 管理者が社内階層における職位を変更できる対象は、直属の部下に制限されます。この実行には、管理者は指定されたアプリケーションのみを使用する必要があります。これを実装するには、次の処理を行います。
  - EMPNO およびアプリケーション・コンテキストに基づいて、表に対するファイングレイン・アクセス・ポリシーを定義します。
  - ログイン・トリガーを使用して EMPNO を設定します。
  - 更新処理用に指定されたパッケージを使用して、アプリケーション・コンテキストを設定します（イベント・トリガーおよびアプリケーション・コンテキスト）。

注意： ファイングレイン・アクセス・コントロールでは許可されていないユーザーが誤って行を変更することを防止できるので、この例では、表に対する UPDATE 権限を PUBLIC に付与します。

```
CONNECT system/manager AS sysdba
GRANT CONNECT,RESOURCE,UNLIMITED TABLESPACE,CREATE ANY CONTEXT, CREATE PROCEDURE,
CREATE ANY TRIGGER TO HR IDENTIFIED BY HR;
CONNECT hr/hr;
CREATE TABLE Directory (Empno   NUMBER(4) NOT NULL,
                        Mgrno   NUMBER(4) NOT NULL,
                        Rank    NUMBER(7,2) NOT NULL);

CREATE TABLE Payroll (Empno  NUMBER(4) NOT NULL,
                      Name   VARCHAR(30) NOT NULL );

/* seed the tables with a couple of managers: */
INSERT INTO Directory VALUES (1, 1, 1.0);
INSERT INTO Payroll VALUES (1, 'KING');
INSERT INTO Directory VALUES (2, 1, 5);
INSERT INTO Payroll VALUES (2, 'CLARK');

/* Create the sequence number for EMPNO: */
CREATE SEQUENCE Empno_seq START WITH 5;

/* Create the sequence number for RANK:  */
CREATE SEQUENCE Rank_seq START WITH 100;

CREATE OR REPLACE CONTEXT Hr_app USING Hr.Hr0_pck;
CREATE OR REPLACE CONTEXT Hr_sec USING Hr.Hr1_pck;

CREATE or REPLACE PACKAGE Hr0_pck IS
PROCEDURE adjustrankby1(Empno NUMBER);
END;

CREATE or REPLACE PACKAGE BODY Hr0_pck IS
/* raise the rank of the empno by 1:  */
PROCEDURE Adjustrankby1(Empno NUMBER)
IS
   Stmt   VARCHAR2(100);
   BEGIN

   /*Set context to indicate application state */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',1);
   /* Now we can issue DML statement:  */
   Stmt := 'UPDATE SET Rank := Rank +1 FROM Directory d WHERE d.Empno = '
```

```
   || Empno;
   EXECUTE IMMEDIATE STMT;

/* Re-set application state: */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',0);
   END;
END;

CREATE or REPLACE PACKAGE hr1_pck IS PROCEDURE setid;
END;
/
/* Based on userid, find EMPNO, and set it in application context */

CREATE or REPLACE PACKAGE BODY Hr1_pck IS
PROCEDURE setid
  IS
  id NUMBER;
  BEGIN
    SELECT Empno INTO id FROM Payroll WHERE Name =
      SYS_CONTEXT('userenv','session_user') ;
    DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
    DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
  EXCEPTION
    /* For purposes of demonstration insert into payroll table
    /  so that user can continue on and run example. */
    WHEN NO_DATA_FOUND THEN
      INSERT INTO Payroll (Empno, Name)
         VALUES (Empno_seq.NEXTVAL, SYS_CONTEXT('userenv','session_user'));
      INSERT INTO Directory (Empno, Mgrno, Rank)
         VALUES (Empno_seq.CURRVAL, 2, Rank_seq.NEXTVAL);
      SELECT Empno INTO id FROM Payroll WHERE Name =
        sys_context('userenv','session_user') ;
      DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
      DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
    WHEN OTHERS THEN
      NULL;
    /* If this is to be fired via a "logon" trigger,
    /  you need to handle exceptions if you want the user to continue
    /  logging into the database. */
  END;
END;

GRANT EXECUTE ON Hr1_pck TO public;

CONNECT system/manager AS sysdba

CREATE OR REPLACE TRIGGER Databasetrigger
```

```
AFTER LOGON
ON DATABASE
BEGIN
   hr.Hr1_pck.Setid;
END;

/* Creates the package for finding the MGRID for a particular EMPNO
using definer's right (encapsulated privileges). Note that users are
granted EXECUTE privileges only on this package, and not on the table
(DIRECTORY) it is querying. */

CREATE or REPLACE PACKAGE hr2_pck IS
   FUNCTION Findmgr(Empno NUMBER) RETURN NUMBER;
END;

CREATE or REPLACE PACKAGE BODY hr2_pck IS
   /* insert a new employee record: */
   FUNCTION findmgr(empno number) RETURN NUMBER IS
   Mgrid NUMBER;
   BEGIN
      SELECT mgrno INTO mgrid FROM directory WHERE mgrid = empno;
   RETURN mgrid;
   END;
END;

CREATE or REPLACE FUNCTION secure_updates(ns varchar2,na varchar2)
  RETURN VARCHAR2 IS
     Results VARCHAR2(100);
    BEGIN
       /* Only allow updates when designated application has set the session
       state to indicate we are inside it. */
       IF (sys_context('hr_sec','adjstate') = 1)
          THEN results := 'mgr = SYS_CONTEXT("hr_sec","empno")';
       ELSE results := '1=2';
      END IF;
      RETURN Results;
   END;

/* Attaches fine-grained access policy to all update operations on
hr.directory */

CONNECT system/manager AS sysdba;
BEGIN
   DBMS_RLS.ADD_POLICY('hr','directory_u','secure_update','hr',
                       'secure_updates','update',TRUE,TRUE);
END;
```

グローバルにアクセスされるアプリケーション・コンテキストの概要

多くのアプリケーション・アーキテクチャでは、中間層アプリケーションがアプリケーション・ユーザーのセッション・プーリングを管理します。この場合、複数のユーザーがアプリケーションで認証され、アプリケーションは単一の識別情報を使用してデータベースにログインし、すべての接続をメンテナンスします。このような環境のアプリケーションは、セッションを使用しないため、セッションに依存する保護アプリケーション・コンテキストを使用してアプリケーション属性のメンテナンスをすることができません。

また、ユーザーがアプリケーション（Oracle Forms など）を介してデータベースに接続する場合、このアプリケーションがデータベースに接続するために、他のアプリケーション（Oracle Reports など）を起動する場合もあります。これらのアプリケーションは、同一のデータベース・セッションを共有しているように見えるように、セッション属性を共有する必要がある場合があります。

グローバル・アプリケーション・コンテキストは、トラステッド・セッション間で共有できる保護アプリケーション・コンテキストの 1 つです。アプリケーション（特に中間層アプリケーション）でこのサポートを使用すると、ファイングレイン・アクセス・コントロール・ポリシーの施行のみでなく、アプリケーション属性を安全かつグローバルに管理できます。

注意： グローバル・アプリケーション・コンテキストは Real Application Clusters では使用できません。

注意： パブリック Java クラスの AppCtxManager および AppCtxMessages は、パッケージ oracle.security.rdbms.appctx で公開されています。この API は、ユーザーのアプリケーション・コンテキストを集中的に格納する場所を提供します。これらのクラスの詳細は、『Oracle9*i* Java パッケージ・プロシージャ・リファレンス』を参照してください。関連情報は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

クライアント・セッションでアプリケーション・コンテキストを管理するための DBMS_SESSION インタフェースの使用

アプリケーション・コンテキストを管理するための DBMS_SESSION インタフェースには、各アプリケーション・コンテキストに対するクライアント識別子があります。このため、アプリケーション・コンテキストをグローバルに管理しながら、各クライアントは自分に割り当てられたアプリケーション・コンテキストのみを参照できます。次のような DBMS_SESSION のインタフェースを使用すると、管理者はクライアント・セッションでアプリケーション・コンテキストを管理できます。

- SET_CONTEXT
- CLEAR_CONTEXT
- SET_IDENTIFIER
- CLEAR_IDENTIFIER

中間層アプリケーション・サーバーは、SET_CONTEXT を使用して特定のクライアント ID にアプリケーション・コンテキストを設定できます。クライアントの要求を処理するためにデータベース接続を割り当てるときは、アプリケーション・サーバーが、アプリケーション・セッションの ID を表す SET_IDENTIFIER を発行する必要があります。すると、クライアントが SYS_CONTEXT を起動するたびに、設定されている識別子に対応付けられたコンテキストのみが戻されます。つまり、アプリケーション・サーバーは SET_IDENTIFIER を使用して、データベース・セッションを特定のユーザーまたはグループに対応付けます。CLIENT_IDENTIFIER はセッションの属性であり、セッション情報で参照できます。CLIENT_IDENTIFIER は、グローバル・アプリケーション・コンテキストにアクセスする場合にも重要です。たとえば、ビジネス・パートナに情報を提供する Web ベースのアプリケーションには、ゴールド・パートナ、シルバー・パートナおよびブロンズ・パートナの 3 つのタイプのユーザーがあると想定します。これらのユーザーは、それぞれに使用可能な情報レベルが異なります。各ユーザーがアプリケーション・コンテキストに独自のセッションを設定するのではなく、アプリケーションで、ゴールド・パートナ、シルバー・パートナおよびブロンズ・パートナ用のグローバル・アプリケーション・コンテキストを設定できます。こうすると、次のことができます。

- SET_IDENTIFIER を使用して、ゴールド・パートナ、シルバー・パートナまたはブロンズ・パートナに特定のセッションを設定します。
- ゴールド・パートナ、シルバー・パートナおよびブロンズ・パートナがそれぞれ適切なデータを取得できるように、セッションを正しいグローバル・アプリケーション・コンテキストに対応付けます。

アプリケーションは、この 3 つのグローバル・コンテキストを 1 回のみ初期化し、CLIENT_IDENTIFIER を使用して、正しいアプリケーション・コンテキストにアクセスすることによってデータ・アクセスを制限できます。このように、各セッションのアプリケーション・コンテキストを個別に初期化するかわりに、セッションを再利用し、設定済のグローバル・アプリケーション・コンテキストにアクセスすることによって、パフォーマンスが向上します。

例 1: アプリケーション・コンテキストのグローバル・アクセス

次に、グローバルにアクセスされるアプリケーション・コンテキストの例を示します。

1. クライアント SCOTT にクライアント識別子 12345 を割り当てているアプリケーション・サーバーを考えます。このアプリケーション・サーバーは、クライアント識別子 12345 に対して、13 という値を持つアプリケーション・コンテキスト RESPONSIBILITY が HR ネームスペースにあることを示す、次のような文を発行します。

```
DBMS_SESSION.SET_CONTEXT( 'HR', 'RESPONSIBILITY' , '13', 'SCOTT', '12345' );
```

HR は、次のように作成されたグローバル・コンテキスト・ネームスペースである必要があることに注意してください。

```
CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
```

2. クライアント SCOTT が新規のデータベース・セッションに割り当てられると、識別情報を示すために、APPSMGR を使用してデータベースへの接続を確立する各クライアント・セッションに対して、次のようなコマンドが発行されます。

```
DBMS_SESSION.SET_IDENTIFIER('12345');
```

3. データベース・セッション内では、SYS_CONTEXT('HR','RESPONSIBILITY') コールがあると、データベース・エンジンがクライアント識別子 12345 をグローバル・コンテキストに一致させ、13 という値を戻します。
4. このデータベース・セッションの終了時に、中間層が次のようなコマンドを発行してクライアント識別子を消去できます。

```
DBMS_SESSION.CLEAR_IDENTIFIER( );
```

セッションのクライアント識別子は、消去されると NULL 値になります。それ以降の SYS_CONTEXT コールでは、SET_IDENTIFIER インタフェースを使用してクライアント識別子を再設定しないかぎり、クライアント識別子が NULL のアプリケーション・コンテキストのみが取得されます。

> **注意：** グローバルにアクセスされるアプリケーション・コンテキストでは、バージョン化は使用できません。バージョン化を使用すると、特定の時刻の SYS_CONTEXT 値が戻されます。複数のクライアント・セッションが、常に同一のグローバル・アプリケーション・コンテキストの値にアクセスしている可能性があるため、バージョン化は使用できません。単一のセッションに基づく単純なアプリケーション・コンテキストは、バージョン化できます。

例 2: プロキシ認証アプリケーションに対するアプリケーション・コンテキストのグローバル・アクセス

次に、プロキシ認証アプリケーションの例を示します。

1. 管理者が次の文を発行して、グローバル・コンテキスト・ネームスペースを作成します。

```
CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
```

2. HR アプリケーション・サーバー（AS）が起動し、ユーザー APPSMGR として HR データベースへの複数の接続を確立します。

3. ユーザー SCOTT が HR AS にログインします。
4. AS がアプリケーションに対して SCOTT を認証します。
5. AS がこの接続に対して一時的なセッション ID 12345 を割り当てます（または単にアプリケーション・ユーザー ID を使用します）。
6. セッション ID が、Cookie の一部として SCOTT のブラウザに戻されるか、または AS によって保持されます。

> **注意：** アプリケーションが CLIENT_IDENTIFIER として使用するセッション ID を生成する場合、そのセッション ID は適度にランダムで、暗号化によってネットワーク上で保護されている必要があります。セッション ID がランダムでない場合、不正なユーザーがセッション ID を推測して、他のユーザーのデータにアクセスする可能性があります。セッション ID がネットワーク上で暗号化されていない場合、不正なユーザーがセッション ID を取得し、接続にアクセスする可能性があります。

7. AS が HR.INIT パッケージをコールして、このクライアントのアプリケーション・コンテキストを初期化すると、次の文が発行されます。

```
DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', 'APPSMGR', 12345 );
DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', 'APPSMGR', 12345 );
```

8. AS がこのセッションにデータベース接続を割り当て、次の文を発行してセッションを初期化します。

```
DBMS_SESSION.SET_IDENTIFIER( 12345 );
```

9. このデータベース・セッション内のすべての SYS_CONTEXT コールが、そのクライアントのセッションのみに属するアプリケーション・コンテキストの値を戻します。たとえば、SYS_CONTEXT('hr','id') は、SCOTT という値を戻します。
10. セッションが終了すると、AS は次の文を発行してクライアントの識別情報を消去できます。

```
DBMS_SESSION.CLEAR_IDENTIFIER ( );
```

他のデータベース・ユーザー（ADAMS）がデータベースにログインしても、そのユーザーは AS によって設定されているグローバル・コンテキストにアクセスできません。AS によって、APPSMGR としてログインしたユーザーのみがこのグローバル・コンテキストを参照できるように指定されているためです。AS が次の文を使用した場合は、12345 に設定されたクライアント ID を持つすべてのユーザー・セッションが、グローバル・コンテキストを参照できます。

```
DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', NULL , 12345 );
DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', NULL , 12345 );
```

この方法を使用すると、異なるユーザーが同一のコンテキストを共有できます。

ただし、ユーザーは、グローバル・コンテキストの設定が異なると、セキュリティ上の意味が変わることに注意する必要があります。基本的に、ユーザー名に NULL が指定されている場合は、すべてのユーザーがそのグローバル・コンテキストにアクセスできます。グローバル・コンテキストのクライアント ID に NULL が指定されている場合は、初期化されていないクライアント ID を持つセッションのみがそのグローバル・コンテキストにアクセスできます。

ユーザーは、セッションに設定されたクライアント識別子を次のように問合せできます。

```
SYS_CONTEXT('USERENV','CLIENT_IDENTIFIER')
```

DBA は、V$SESSION ビューの CLIENT_IDENTIFIER および USERNAME を問い合せることによって、どのセッションにクライアント識別子が設定されているかを参照できます。

ユーザーは、SYS_CONTEXT('USERENV', 'GLOBAL_CONTEXT_MEMORY') を使用して、グローバル・コンテキストの使用領域（バイト単位）を参照できます。

参照：

- 『Oracle9*i* SQL リファレンス』を参照してください。
- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。
- クライアント識別子の詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』および『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

アプリケーション・コンテキストの外部での初期化

この機能を使用すると、外部リソースからの属性値の初期化を許可する、特別なタイプのネームスペースを指定できます。この機能によって、パフォーマンスが向上し、セッションからセッションへ属性を自動伝播できるようになります。たとえば、多くの組織では、LDAP ベースのディレクトリでユーザー情報を集中管理する必要があります。Oracle9*i* Enterprise User Security 機能は、Oracle Internet Directory でのユーザーおよび認可の集中管理をサポートします。ただし、VPD の施行に使用するために、アプリケーションが LDAP から次のような属性を取得する必要がある場合があります。

- ユーザーの職位
- ユーザーが所属する組織
- ユーザーの物理的な位置

LDAP などの外部ソースからアプリケーション・コンテキストを初期化する機能を使用すると、組織が VPD の施行のために所有し、集中管理している既存の情報を、データベース表にレプリケートまたはコピーすることなく利用できます。

この項の内容は次のとおりです。

ユーザーからのデフォルト値の取得

ユーザーからデフォルト値を取得することが望ましい場合があります。これらのデフォルト値は、まずヒントまたはプリファレンスとして機能し、値の妥当性チェック後にトラステッド・コンテキストになります。同様に、クライアントが、一部のデフォルト値を初期化し、次にログイン・イベント・トリガーまたはアプリケーションを使用して値の妥当性チェックができる有効な方法を必要とする場合があります。

ジョブ・キューに関しては、管理者が、ジョブ発行ルーチンで、ジョブの発行時に設定されたすべてのコンテキストを記録し、バッチ・ジョブの実行時にそのコンテキストを格納する必要がある場合があります。コンテキストの整合性を保持するため、ジョブ・キューは、コンテキストを設定する際に特定の PL/SQL パッケージを回避できません。一方、外部で初期化されたアプリケーション・コンテキストは、ジョブ・キュー・プロセスからのコンテキスト値の初期化を許可します。

リモート・セッションへのコンテキストの自動伝播がセキュリティ上の問題を発生させる可能性があるのに対して、開発者または管理者は、特定の PL/SQL プロシージャ以外のリソースからデフォルト値を取るこの新しいタイプのコンテキストを効果的に使用できます。さらに、この機能は、1 回の `OCISessionBegin()` コールでより多くの情報をサーバーにバンドルできる拡張可能なインタフェースを OCI クライアントに提供するため、パフォーマンスが向上します。

他の外部リソースからの値の取得

外部で初期化されたアプリケーション・コンテキストは、特定のトラステッド・パッケージを使用する他に、OCI インタフェース、ジョブ・キュー・プロセスまたはデータベース・リンクなどの外部リソースによる属性および値の初期化も許可します。特長を次に示します。

- リモート・セッションでは、外部で初期化されたコンテキスト・ネームスペースにあるコンテキスト値を自動伝播できます。
- ジョブ・キューでは、外部で初期化されたコンテキスト・ネームスペースにあるコンテキスト値をリストアできます。
- OCI インタフェースには、外部で初期化されたコンテキスト・ネームスペースにあるコンテキストを初期化する方法を提供します。

この新しいタイプのネームスペースは、OCI を使用するどのクライアント・プログラムによっても初期化できますが、ログイン・イベント・トリガーによって値の妥当性チェックが行われます。属性の値の解析および信頼性の判断は、アプリケーションが行います。

この機能を使用する場合、中間層サーバーが、データベース・ユーザーのかわりに実際にコンテキスト値を初期化できることに注意してください。コンテキスト属性は、リモート・セッションの起動時にそのセッションに対して伝播されます。リモート・データベースは、ネームスペースが外部で初期化されている場合はこれらの値を受け入れます。

データベースに認識されないユーザーに関する値の取得

外部で初期化されたアプリケーション・コンテキストは、特に、ユーザーがデータベースに認識されない場合に有効です。このような場合、アプリケーションは、通常、単一のデータベース・ユーザーとして接続し、すべてのアクションがそのユーザーで行われます。すべてのユーザー・セッションが同一のユーザーとして作成されるため、このセキュリティ・モデルでは、通常、VPD 機能を使用してユーザーごとまたは顧客データごとにデータを分離することが、不可能または非常に困難です。ただし、これらのアプリケーションでは、クライアント識別子をアプリケーション・ユーザー・プロキシとして使用できます。この場合、アプリケーションはクライアント識別子を使用して、実際のアプリケーション・ユーザー名をデータベースにプロキシ化します。

この方法には次のようなメリットがあります。アプリケーション・ユーザー・プロキシを使用すると、クライアント識別子（実際のアプリケーション・ユーザーの名前を獲得するために使用する）を変更するのみで、複数のユーザーがセッションを再利用できます。これによって、ユーザーに対して個別のセッションおよび個別の属性を設定することで発生するオーバーヘッドを回避し、（新規のアプリケーション・ユーザー名を表す）クライアント識別子を変更するのみで、アプリケーションによるセッションの再利用を可能にします。クライアントがクライアント識別子を変更した場合、パフォーマンス向上のため、その変更が次の OCI（または Thick JDBC）コールに伝達されます。（クライアント識別子を介する）アプリケーション・ユーザー・プロキシは、OCI、Thick JDBC および Thin JDBC で使用できます。

たとえば、ユーザー Daniel が Web Expense アプリケーションに接続するとします。Daniel はデータベース・ユーザーではなく、一般的な Web Expense アプリケーション・ユーザーです。アプリケーションは、一般的な Web ユーザーに対してグローバル・アプリケーション・コンテキストを設定し、クライアント識別子として `DANIEL` を設定します。Daniel が Web Expense のフォームを完了し、アプリケーションを終了します。次に Ajit が Web Expense アプリケーションに接続します。アプリケーションは、Ajit に対して新規のセッションを設定するのではなく、Daniel に関する既存のセッションを、クライアント識別子を `AJIT` に変更するのみで再利用します。これによって、データベースへ新規の接続を設定するためのオーバーヘッド、および新規のアプリケーション・コンテキストを初期化するためのオーバーヘッドの両方を回避します。

クライアント識別子には、アプリケーションがアクセス制御の基盤とするすべての要素を指定できます。必ずしもアプリケーション・ユーザー名である必要はありません。

アプリケーション・ユーザーがデータベース・ユーザーでない場合にクライアント識別子を使用するもう 1 つの方法は、グループのタイプまたはロールのメカニズムとしてクライアント識別子を使用する方法です。たとえば、Marketing アプリケーションに標準パートナ、シルバー・パートナおよびゴールド・パートナの 3 タイプのユーザーがある場合を考えます。アプリケーションは、グローバル・アプリケーション・コンテキスト機能を使用して、3 タ

イプのコンテキスト（標準、シルバーおよびゴールド）を設定できます。アプリケーションは、次に、ユーザーのパートナ・タイプを判断し、セッションに対してデータベースにクライアント識別子を渡します。クライアント識別子（標準、シルバーおよびゴールド）は、ここでは正しいアプリケーション・コンテキストを示すポインタとして動作します。たとえば、クライアント識別子が *silver* である複数のセッションは、同一のアプリケーション・コンテキストを共有します。

参照：

- 『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。
- 『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

アプリケーション・コンテキストのグローバルな初期化

この機能は、ユーザーのアプリケーション・コンテキストを集中的に格納する場所を提供します。この機能によって、アプリケーションは、初期化中にユーザーの識別情報に基づいてユーザーのコンテキストを設定できます。特にこの機能では、Oracle Label Security のラベルおよび権限がサポートされます。この機能を使用すると、管理者は、多数のユーザーおよびデータベースのコンテキストをより簡単に管理できます。

この項の内容は次のとおりです。

LDAP を使用したアプリケーション・コンテキスト

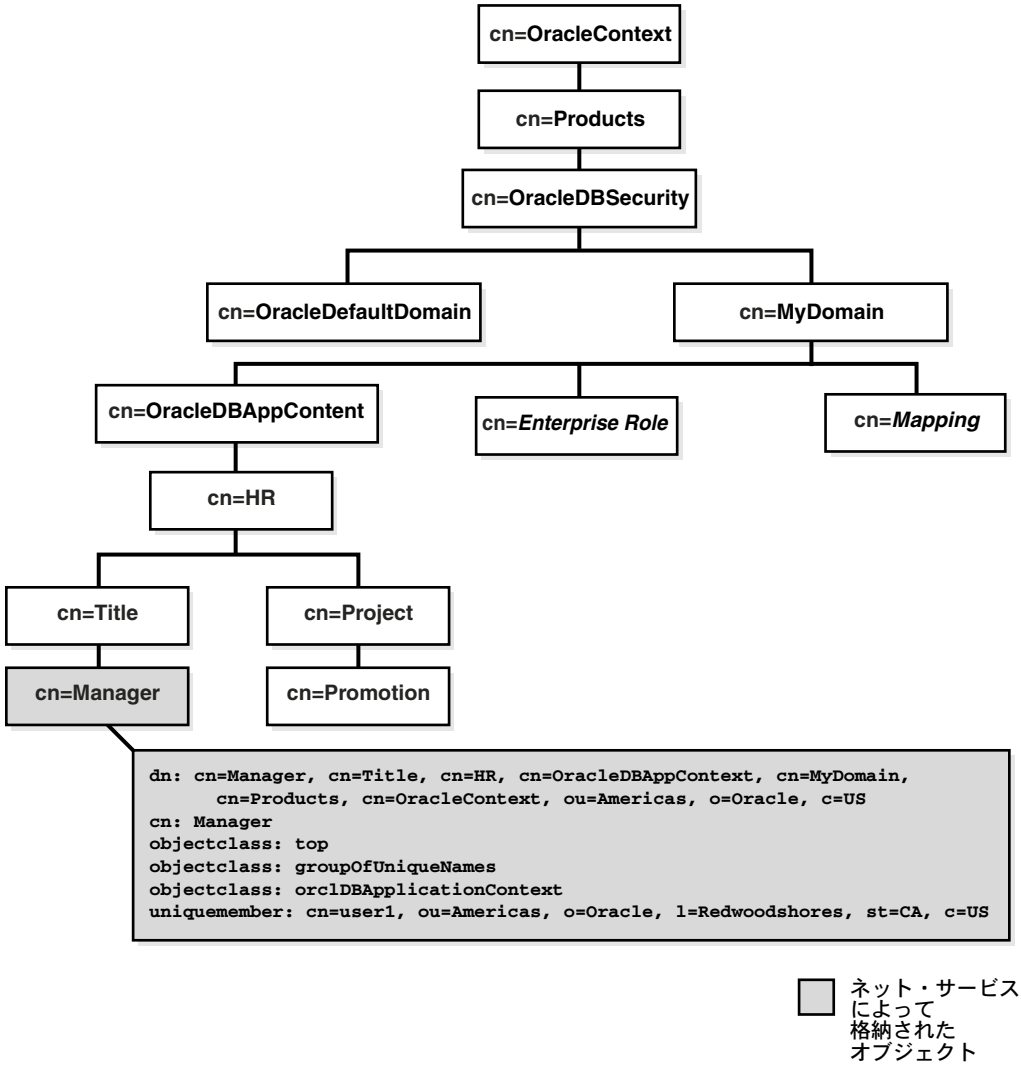
グローバルに初期化されたアプリケーション・コンテキストは、LDAP を使用します。LDAP は拡張可能で効率的な標準ディレクトリ・アクセス・プロトコルです。LDAP ディレクトリには、このアプリケーションが割り当てられているユーザーのリストが格納されます。Oracle9*i* では、エンタープライズ・ユーザーの認証および認可用のディレクトリ・サービスとして、Oracle Internet Directory を使用できます（エンタープライズ・ユーザーのセキュリティには Oracle Advanced Security が必要であることに注意してください）。

LDAP オブジェクト `orclDBApplicationContext`（`groupOfUniqueNames` のサブクラス）は、アプリケーション・コンテキスト値をディレクトリに格納するように定義されています。図 12-1 に、Human Resources の例に基づいて、アプリケーション・コンテキスト・オブジェクトの位置を示します。

`orclDBApplicationContext` の値を取得するには、内部 C 関数が必要であることに注意してください。アプリケーション・コンテキスト値のリストは、RDBMS に戻されます。

注意： この例では、HR はネームスペース、Title および Project は属性、Manager および Promotion は値です。

図 12-1 LDAP のディレクトリ情報ツリー（DIT）におけるアプリケーション・コンテキストの位置



グローバルに初期化されたアプリケーション・コンテキストの動作

管理者は、ユーザーのグローバル・アプリケーション・コンテキスト値をデータベースおよびディレクトリに設定します。

グローバル・ユーザーがデータベースに接続すると、Oracle Advanced Security オプションが認証を行い、データベースに接続するユーザーの識別情報を確認します。識別が完了すると、ユーザーのグローバル・ロールが LDAP から取得されます。次に、ユーザーのグローバル・アプリケーション・コンテキストが LDAP から取得されます。ユーザーがデータベースにログインすると、グローバル・ロールおよび初期アプリケーション・コンテキストがすでに設定されています。

例 : アプリケーション・コンテキストのグローバルな初期化

部門名や職位などのユーザーの初期アプリケーション・コンテキストは、LDAP ディレクトリで設定および格納できます。これらの値はユーザーのログイン中に取得されるので、コンテキストが適切に設定されます。さらに、ユーザーに関するすべての情報が取得され、アプリケーション・コンテキスト・ネームスペース SYS_USER_DEFAULTS に格納されます。次に例を示します。

1. データベースにアプリケーション・コンテキストを作成します。

```
CREATE CONTEXT HR USING hrapps.hr_manage_pkg INITIALIZED GLOBALLY;
```

2. 新規のエントリを作成し、LDAP ディレクトリに追加します。

   LDAP ディレクトリに追加されたエントリの例を次に示します。これらのエントリは、アプリケーション（ネームスペース）HR に属性値 Manager を持つ属性名 Title を作成し、ユーザー名 user1 および user2 を割り当てます。

```
dn: cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=OracleC
ontext,ou=Americas,o=oracle,c=US
changetype: add
cn: OracleDBAppContext
objectclass: top
objectclass: orclContainer

dn: cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=O
racleContext,ou=Americas,o=oracle,c=US
changetype: add
cn: HR
objectclass: top
objectclass: orclContainer

dn: cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Prod
ucts,cn=OracleContext,ou=Americas,o=oracle,c=US
changetype: add
cn: Title
objectclass: top
```

```
objectclass: orclContainer

dn:
cn=Manager,cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,
cn=Products,cn=OracleContext,ou=Americas,o=oracle,c=US
cn: Manager
objectclass: top
objectclass: groupofuniquenames
objectclass: orclDBApplicationContext
uniquemember: CN=user1,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
uniquemember: CN=user2,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
```

3. そのユーザーに関する LDAP inetOrgPerson オブジェクト・エントリが存在する場合、接続時に inetOrgPerson からのすべての属性も取得され、これらの属性がネームスペース SYS_LDAP_USER_DEFAULT に割り当てられます。inetOrgPerson エントリの例を次に示します。

```
dn: cn=user1,ou=Americas,O=oracle,L=redwoodshores,ST=CA,C=US
changetype: add
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: user1
sn: One
givenName: User
initials: UO
title: manager, product development
uid: uone
mail: uone@us.oracle.com
telephoneNumber: +1 650 123 4567
employeeNumber: 00001
employeeType: full time
```

4. データベースに接続します。

user1 がドメイン myDomain に属するデータベースに接続する場合、user1 は自分の Title を Manager に設定します。user1 に関するすべての情報が、LDAP ディレクトリから取得されます。次の構文を使用して値を取得できます。

```
SYS_CONTEXT('namespace','attribute name')
```

次に例を示します。

```
DECLARE
tmpstr1 VARCHAR2(30);
tmpstr2 VARCHAR2(30);
BEGIN
```

```
tmpstr1 = SYS_CONTEXT('HR','TITLE');
tmpstr2 = SYS_CONTEXT('SYS_LDAP_USER_DEFAULT','telephoneNumber');
DBMS_OUTPUT.PUT_LINE('Title is ' || tmpstr1);
DBMS_OUTPUT.PUT_LINE('Telephone Number is ' || tmpstr2);
END;
```

前述の例の出力を次に示します。

```
Title is Manager
Telephone Number is +1 650 123 4567
```

ファイングレイン・アクセス・コントロールの概要

ファイングレイン・アクセス・コントロールを使用すると、詳細レベルでセキュリティ・ポリシーを施行するアプリケーションを作成できます。これを使用することによって、たとえば Oracle サーバーにアクセスするユーザーが自分のアカウントのみを表示したり、外科医が自分の患者の記録のみを表示したり、管理者が自分の部下の記録のみを表示できるように制限することができます。

ファイングレイン・アクセス・コントロールを使用する場合は、アプリケーションの基になった表、ビューまたはシノニムに付加されるセキュリティ・ポリシー関数を作成します。その後、ユーザーがそのオブジェクトに対して DML 文（SELECT、INSERT、UPDATE または DELETE）を入力すると、その文で正しいアクセス制御が実装されるように、Oracle は、ユーザーの文を（ユーザーに対して透過的に）動的に変更します。

この項の内容は次のとおりです。

- ファイングレイン・アクセス・コントロールの機能
- ファイングレイン・アクセス・コントロールの動作
- ポリシー・グループの設定方法
- 表、ビューまたはシノニムへのポリシーの追加方法
- 文に適用されるポリシーの確認方法
- EXEMPT ACCESS POLICY システム権限
- セキュリティに関する潜在的な問題となる非定型ツールの使用

ファイングレイン・アクセス・コントロールの機能

ファイングレイン・アクセス・コントロールは、次のような機能を提供します。

- 表ベース、ビュー・ベースまたはシノニム・ベースのセキュリティ・ポリシー
- 各表、ビューまたはシノニムに対する複数のポリシー

- セキュリティ・ポリシーのグループ化
- 高いパフォーマンス
- デフォルト・セキュリティ・ポリシー

表ベース、ビュー・ベースまたはシノニム・ベースのセキュリティ・ポリシー

アプリケーションではなく、表、ビューまたはシノニムにセキュリティ・ポリシーを付加すると、セキュリティ、簡潔性、柔軟性のすべてが向上します。

セキュリティ

表、ビューまたはシノニムにセキュリティ・ポリシーを付加することによって、アプリケーション・セキュリティの重大な問題が解決されます。たとえば、アプリケーションの使用を許可されているユーザーが、そのアプリケーションに対応付けられている権限を利用し、SQL*Plus などの非定型の問合せツールを使用してデータベースを誤って変更してしまう可能性があります。ファイングレイン・アクセス・コントロールでは、セキュリティ・ポリシーを表、ビューまたはシノニムに付加することによって、ユーザーがどのような方法でデータにアクセスしても、同じセキュリティが施行されます。

簡潔性

セキュリティ・ポリシーを表、ビューまたはシノニムに追加するということは、表ベース、ビュー・ベースまたはシノニム・ベースのアプリケーションごとにポリシーを繰り返し追加するのではなく、1 回のみ追加することを意味します。

柔軟性

SELECT 文には 1 つのセキュリティ・ポリシーを、INSERT 文には別のポリシーを、さらに UPDATE 文および DELETE 文にはまた別のポリシーを指定できます。たとえば、人事部の担当者には、その部門内のすべての社員のレコードを SELECT できるようにし、名字が A ～ F で始まるその部門内の社員の給与のみを UPDATE できるようにすることができます。

注意： 表にポリシーを定義することはできますが、表に定義されたポリシーから表を選択することはできません。

各表、ビューまたはシノニムに対する複数のポリシー

同一の表、ビューまたはシノニムに対して複数のポリシーを設定できます。たとえば、受注用の基本アプリケーションがあり、社内の各部門にはそれぞれ独自の特殊なデータ・アクセス規則があるとします。基本アプリケーションのポリシー関数を作成しなおさなくても、部門固有のポリシー関数を表に追加できます。

表に適用されるすべてのポリシーは、AND 構文で施行されることに注意してください。そのため、CUSTOMERS 表に 3 つのポリシーを適用している場合、各ポリシーが表のすべてのアクセスに適用されます。データにアクセスするアプリケーションに応じて異なるポリシーが適用されるように、ポリシー・グループおよび駆動アプリケーション・コンテキストを使用して、ファイングレイン・アクセス・コントロールの施行を分割できます。これによって、開発グループ間でポリシーを調整する必要がなくなり、アプリケーションの開発が容易になります。また、常に適用する（たとえば、ホスト環境でサブスクライバによってデータ分割を施行する）デフォルト・ポリシー・グループを持つこともできます。

セキュリティ・ポリシーのグループ化

複数のセキュリティ・ポリシーを持つ複数のアプリケーションが同一の表、ビューまたはシノニムを共有できるため、表、ビューまたはシノニムがアクセスされたときに有効になるポリシーを識別することが重要です。

たとえば、ホスト環境で、A 社が、BENEFIT 表を B 社および C 社のホストとします。この表は、2 つの異なるセキュリティ・ポリシーを持つ、HUMAN RESOURCES および FINANCE という 2 つの異なるアプリケーションによってアクセスされます。HUMAN RESOURCES アプリケーションは社内の序列に基づいてユーザーを認可し、FINANCE アプリケーションは部門に基づいてユーザーを認可します。これらの 2 つのポリシーを BENEFIT 表に統合するには、2 つの企業が共同でポリシーを開発する必要がありますが、それは不可能です。ベース・オブジェクトに一連の特定のポリシーを施行するアプリケーション・コンテキストを定義することにより、各アプリケーションがセキュリティ・ポリシーの集合を固別に実装できます。

これを行うには、セキュリティ・ポリシーをグループ化します。アプリケーション・コンテキストを参照することにより、Oracle サーバーが、実行時に有効になるポリシーのグループを判断します。Oracle サーバーは、そのポリシー・グループに属するすべてのポリシーを施行します。

高いパフォーマンス

ファイングレイン・アクセス・コントロールを使用すると、指定された問合せに対するポリシー関数は、それぞれ 1 回のみ、文の解析時に評価されます。さらに、動的に変更される問合せ全体が最適化され、解析済の文を共有および再利用することができます。これは、再作成された問合せで、ディクショナリ・キャッシュ、共有カーソルなどの Oracle の高パフォーマンス機能を利用できるということを意味します。

デフォルト・セキュリティ・ポリシー

アプリケーションによってセキュリティ・ポリシーを分割することが理想的ですが、常に有効であるセキュリティ・ポリシーを設定しておく方法も有効です。前述の例では、Human Resources アプリケーションまたは Finance アプリケーションのいずれを使用している場合も、ホストされるアプリケーションはいつでも subscriber_ID によってデータ分割を施行できます。デフォルト・セキュリティ・ポリシーを使用すると、開発者はすべての条件下で基礎となるセキュリティを施行できます。一方、アプリケーションごとにセキュリティ・ポリシーを分割（セキュリティ・グループを使用）すると、デフォルト・セキュリティ・ポリ

シーにアプリケーション固有のセキュリティ・レイヤーを追加できます。デフォルト・セキュリティ・ポリシーを実装するには、SYS_DEFAULT ポリシー・グループにポリシーを追加してください。

ファイングレイン・アクセス・コントロールの動作

ファイングレイン・アクセス・コントロールは、この項に示す例のような、動的に変更された文に基づいて行われます。ORDERS_TAB 表に、「顧客は自分の注文のみ参照できる」というセキュリティ・ポリシーを追加する場合を考えます。この項では、その手順について説明します。

1. ユーザーの DML 文に述語を追加するファンクションを作成します。

   注意： 述語とは WHERE 句のことです。つまり、演算子（=、!=、IS、IS NOT、>、>=）の 1 つに基づく選択基準句のことです。

   ここでは、次の述語を追加するファンクションを作成します。

   ```
   Cust_no = (SELECT Custno FROM Customers WHERE Custname =
              SYS_CONTEXT ('userenv','session_user'))
   ```

2. ユーザーが次の文を入力します。

   ```
   SELECT * FROM Orders_tab;
   ```

3. Oracle サーバーは、作成済のファンクションをコールし、セキュリティ・ポリシーを実装します。

4. ファンクションが、ユーザーの文を次のように動的に変更します。

   ```
   SELECT * FROM Orders_tab WHERE Custno = (
      SELECT Custno FROM Customers
          WHERE Custname = SYS_CONTEXT('userenv', 'session_user'))
   ```

5. Oracle サーバーが、動的に変更された文を実行します。

実行時に、ファンクションは SYS_CONTEXT ('userenv', 'session_user') が戻すユーザー名を使用して、対応する顧客を検索し、ORDERS_TAB 表から戻されるデータをその顧客のデータのみに制限します。

参照： ファイングレイン・アクセス・コントロールの使用の詳細は、12-25 ページの「グローバルにアクセスされるアプリケーション・コンテキストの概要」および『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

ポリシー・グループの設定方法

ポリシー・グループは、アプリケーションに属するセキュリティ・ポリシーの集合です。アプリケーション・コンテキスト（駆動コンテキスト）を指定して、有効なポリシー・グループを指定できます。次に、表、ビューまたはシノニムがアクセスされると、サーバーが駆動コンテキスト（ポリシー・コンテキスト）を調べ、有効なポリシー・グループを判断します。サーバーは、そのポリシー・グループに属するすべての関連ポリシーを施行します。

この項の内容は次のとおりです。

デフォルト・ポリシー・グループ : SYS_DEFAULT

Oracle Policy Manager のツリー構造では、Fine-Grained Access Control Policies フォルダの下に Policy Groups フォルダがあります。Policy Groups フォルダには、SYS_DEFAULT ポリシー・グループのアイコンの他に、各ポリシー・グループのアイコンがあります。

デフォルトでは、すべてのポリシーが SYS_DEFAULT ポリシー・グループに属しています。このグループ内に定義されている、特定の表、ビューまたはシノニムに対するポリシーは、必ず、駆動コンテキストによって特定されているポリシー・グループと一緒に実行されます。SYS_DEFAULT ポリシー・グループには、ポリシーが含まれる場合と含まれない場合があります。SYS_DEFAULT ポリシー・グループを削除しようとすると、エラーが発生します。

SYS_DEFAULT ポリシー・グループに、複数のオブジェクトに対応付けられているポリシーを追加する場合、これらの各オブジェクトには個別に SYS_DEFAULT ポリシー・グループが対応付けられます。たとえば、SCOTT スキーマの EMP 表に 1 つの SYS_DEFAULT ポリシー・グループがある場合、SCOTT スキーマの DEPT 表には別の SYS_DEFAULT ポリシー・グループが対応付けられます。これは、ツリー構造では次のように表示されます。

```
SYS_DEFAULT
  - policy1 (SCOTT/EMP)
  - policy3 (SCOTT/EMP)
SYS_DEFAULT
  - policy2 (SCOTT/DEPT)
```

注意： 同一の名前を持つポリシー・グループがサポートされています。特定のポリシー・グループを選択すると、対応付けられているスキーマおよびオブジェクト名が、画面の右側のプロパティ・シートに表示されます。

新しいポリシー・グループ

表、ビューまたはシノニムにポリシーを追加する場合、DBMS_RLS.ADD_GROUPED_POLICY インタフェースを使用して、ポリシーが属するグループを指定できます。有効になるポリシーを指定するには、DBMS_RLS.ADD_POLICY_CONTEXT インタフェースを使用して駆動コンテキストを追加します。駆動コンテキストが不明なポリシー・グループを戻す場合は、エラーが戻されます。

駆動コンテキストが定義されていない場合は、すべてのポリシーが実行されます。同様に、駆動コンテキストが NULL である場合は、すべてのポリシー・グループのポリシーが施行されます。このようにすると、データにアクセスするアプリケーションはセキュリティ設定モジュール（アプリケーション・コンテキストを設定するモジュール）を回避できないため、該当するすべてのポリシーが適用されます。

同一の表、ビューまたはシノニムに複数の駆動コンテキストを適用して、それぞれを個別に処理できます。これにより、複数のアクティブなポリシーを設定して、施行できます。

たとえば、Benefits アプリケーションおよび Financial アプリケーションをホストする企業があり、アプリケーション間でいくつかのデータベース・オブジェクトが共有されている場合を考えます。2 つのアプリケーションは、SYS_DEFAULT ポリシー・グループの SUBSCRIBER ポリシーを使用して、ホスティング用にストライプ化されます。データ・アクセスは、まずサブスクライバ ID ごとに分割され、次に、ユーザーが Benefits アプリケーションと Financial アプリケーションのどちらにアクセスしているか（駆動コンテキストによって決定される）によって分割されます。ホスティング・サービスを使用する A 社が、A 社のデータ・アクセスにのみ関連付けられるカスタム・ポリシーを適用するとします。駆動コンテキスト（COMPANY A SPECIAL など）を追加して、特別な追加のポリシー・グループが A 社のデータ・アクセスのみに適用されるようにします。このポリシーは A 社のみに関連付けられているため、SUBSCRIBER ポリシーの下では適用できません。基本的なホスティング・ポリシーは、他のポリシーから分離した方がより明確になります。

Oracle Policy Manager を使用したポリシー・グループの設定

Oracle Enterprise Manager から Oracle Policy Manager の GUI にアクセスして、DBMS_RLS.CREATE_POLICY_GROUP コマンドライン・プロシージャを使用してポリシー・グループを作成することもできます。

注意： Oracle Policy Manager を使用してポリシー・コンテキストを作成することもできます。

ポリシー・グループの実装方法

ポリシー・グループを作成するには、管理者は次の 2 つの作業を行う必要があります。

- 有効なポリシー・グループを識別する駆動コンテキストを設定します。
- 必要に応じて、ポリシー・グループにポリシーを追加します。

次に、これらの作業の実行例を示します。

手順 1: 駆動コンテキストの設定

次のコマンドを使用して、駆動コンテキストのネームスペースを作成します。

```
CREATE CONTEXT appsctx USING apps.apps_security_init;
```

次のコマンドを使用して、駆動コンテキストを管理するパッケージを作成します。

```
CREATE OR REPLACE PACKAGE BODY apps.apps_security_init
PROCEDURE setctx ( policy_group varchar2 )
BEGIN

REM  Do some checking to determine the current application.
REM  You can check the proxy if using the proxy authentication feature.
REM  Then set the context to indicate the current application.
.
.
.
DBMS_SESSION.SET_CONTEXT('APPSCTX','ACTIVE_APPS', policy_group);
END;
END;
```

表 APPS.BENEFIT に対する駆動コンテキストを定義します。

```
DBMS_RLS.ADD_POLICY_CONTEXT('apps','benefit','APPSCTX','ACTIVE_APPS')
```

手順 2: デフォルト・ポリシー・グループへのポリシーの追加

データを企業ごとに分割する述語を戻すセキュリティ関数を作成します。

```
CREATE OR REPLACE FUNCTION by_company (schema varchar2, table varchar2)
RETURN VARCHAR2;
BEGIN
  RETURN 'COMPANY = SYS_CONTEXT(''ID'',''MY_COMPANY'')';
END;
```

SYS_DEFAULT 内のポリシーは（SYS、または EXEMPT ACCESS POLICY システム権限を持つユーザー以外の場合）常に実行されるため、このセキュリティ・ポリシー（SECURITY_BY_COMPANY）は、実行中のアプリケーションに関係なく必ず施行されます。これによって、表に対する汎用セキュリティ要件、つまり、実行中のアプリケーションに関係なく各企業が自社のデータを参照できるという要件が満たされます。

APPS.APPS_SECURITY_INIT.BY_COMPANY 関数が、自社のデータのみを確実に参照できるようにする述語を戻します。

```
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','SYS_DEFAULT',
'security_by_company',
'apps','by_company');
```

手順 3: HR ポリシー・グループへのポリシーの追加

HR グループを作成します。

```
CREATE OR REPLACE FUNCTION hr.security_policy
RETURN VARCHAR2;
AS
BEGIN
 RETURN 'SYS_CONTEXT(''ID'',''TITLE'') = ''MANAGER'' ';
END;
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','HR');
```

次に、HR ポリシー・グループに HR_SECURITY というポリシーを追加します。HR.SECURITY_POLICY 関数が、APPS.BENEFIT 表に対して HR のセキュリティを施行する述語を戻します。

```
DBMS_RLS.ADD_GROUPED_POLICYS('apps','benefit','HR',
'hr_security','hr','security_policy');
```

手順 4: FINANCE ポリシー・グループへのポリシーの追加

FINANCE ポリシーを作成します。

```
CREATE OR REPLACE FUNCTION finance.security_policy
RETURN VARCHAR2;
AS
BEGIN
 RETURN 'SYS_CONTEXT(''ID'',''DEPT'') = ''FINANCE'' ';
END;
```

FINANCE というポリシー・グループを作成します。

```
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','FINANCE');
```

FINANCE ポリシー・グループに FINANCE ポリシーを追加します。

```
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','FINANCE',
'finance_security','finance', 'security_policy');
```

これにより、データベースにアクセスすると、認証後にアプリケーションが駆動コンテキストを初期化するようになります。HR アプリケーションでの例を次に示します。

```
execute apps.security_init.setctx('HR');
```

接続に使用されるアプリケーションの妥当性チェック

駆動コンテキストを実装するパッケージは、使用されるアプリケーションの妥当性チェックを正しく行う必要があります。データベースは、駆動コンテキストを実装するパッケージがコンテキスト属性を設定することを（コール・スタックの確認によって）保証しますが、これだけではパッケージ内の不適切または不十分な妥当性チェックを防ぐことはできません。

たとえば、データベース・ユーザーまたはエンタープライズ・ユーザーがデータベースに認識されるアプリケーションでは、駆動コンテキストを設定するパッケージに対して、ユーザーが EXECUTE 権限を持っている必要があります。次のことを認識しているユーザーを考慮します。

- HR アプリケーションより BENEFITS アプリケーションの方が自由なアクセスが可能であること。
- （正しいポリシー・グループを駆動コンテキスト内に設定する）setctx プロシージャでは、実際に接続しているアプリケーションを判断する妥当性チェックを実施しないこと。つまり、このプロシージャでは、（3 層システムに対する）接続元の IP アドレス、またはユーザー・セッションの proxy_user 属性を確認できないこと。

このような状況では、ユーザーが、実際には HR アプリケーションにアクセスしながら、より自由な BENEFITS ポリシー・グループにコンテキストを設定する引数（BENEFITS）を駆動コンテキスト・パッケージに渡す可能性があります。このように、パッケージが行う妥当性チェックが不十分であるために、ユーザーが、用意された厳密なセキュリティ・ポリシーを簡単に回避できる場合があります。

一方、VPD によるプロキシ認証を実装すると、ユーザーのかわりにデータベースに実際に接続する中間層（アプリケーション）の識別情報を確認できます。こうすると、データ・アクセスの仲介に、アプリケーションごとの正しいポリシーが適用されます。たとえば、開発者は、プロキシ認証機能を使用して、データベースに接続するアプリケーション（中間層）が HRAPPSERVER であることを確認できます。駆動コンテキストを実装するパッケージは、HR ポリシー・グループを使用するために駆動コンテキストを設定する前に、ユーザー・セッションの proxy_user が HRAPPSERVER であるかどうかの確認ができます。また、proxy_user が HRAPPSERVER でない場合は、アクセスを禁止できます。

このような場合に、次の問合せが実行されるとします。

```
SELECT * FROM APPS.BENEFIT;
```

Oracle は、デフォルト・ポリシー・グループ（SYS_DEFAULT）およびアクティブなネームスペース HR のポリシーを選択します。問合せは、次のように内部的に再作成されます。

```
SELECT * FROM APPS.BENEFIT WHERE COMPANY = SYS_CONTEXT('ID','MY_COMPANY') and
SYS_CONTEXT('ID','TITLE') = 'MANAGER';
```

表、ビューまたはシノニムへのポリシーの追加方法

DBMS_RLS パッケージを使用すると、セキュリティ・ポリシーを管理できます。このパッケージ内のプロシージャを使用すると、ポリシーおよびポリシーに関連する様々なデータを追加する表、ビューまたはシノニムを指定できます。ポリシーに関連するデータには、ポリシー名、ポリシー・グループ名、ポリシーを実装するファンクション、ポリシーが適用される文の種類（SELECT、INSERT、UPDATE または DELETE）、および追加情報が含まれます。パッケージには次のプロシージャが含まれます。

表 12-2 DBMS_RLS プロシージャ

プロシージャ	用途
DBMS_RLS.ADD_POLICY	表、ビューまたはシノニムにポリシーを追加できます。
DBMS_RLS.DROP_POLICY	表、ビューまたはシノニムからポリシーを削除できます。
DBMS_RLS.REFRESH_POLICY	ポリシーに対応付けられたオープン・カーソルを強制的に再解析できるため、新しいセキュリティ・ポリシーまたはセキュリティ・ポリシーへの変更がすぐに有効になります。
DBMS_RLS.ENABLE_POLICY	表、ビューまたはシノニムに事前に追加したポリシーを使用可能または使用禁止にできます。
DBMS_RLS.CREATE_POLICY_GROUP	ポリシー・グループを作成できます。
DBMS_RLS.ADD_GROUPED_POLICY	特定のポリシー・グループにポリシーを追加できます。
DBMS_RLS.ADD_POLICY_CONTEXT	アクティブなアプリケーションにコンテキストを追加できます。
DBMS_RLS.DELETE_POLICY_GROUP	ポリシー・グループを削除できます。
DBMS_RLS.DROP_GROUPED_POLICY	特定のグループに属するポリシーを削除できます。
DBMS_RLS.DROP_POLICY_CONTEXT	アプリケーションのコンテキストを削除できます。
DBMS_RLS.ENABLE_GROUPED_POLICY	ポリシー・グループ内のポリシーを使用禁止にできます。
DBMS_RLS.REFRESH_GROUPED_POLICY	リフレッシュされたポリシーに対応付けられた SQL 文を再解析できます。

参照：『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

Oracle Policy Manager を使用してセキュリティ・ポリシーを管理することもできます。

文に適用されるポリシーの確認方法

V$VPD_POLICY を使用すると、SQL 文に適用されているポリシーを確認するために、動的ビューを実行できます。デバッグ時に特定の SQL 文に対応するポリシーを検索する場合は、次の表を使用します。

表 12-3 V$VPD_POLICY

列名	型
ADDRESS	RAW(4)
PARADDR	RAW(4)
SQL_HASH	NUMBER
CHILD_NUMBER	NUMBER
OBJECT_OWNER	VARCHAR2(30)
OBJECT_NAME	VARCHAR2(30)
POLICY_GROUP	VARCHAR2(30)
POLICY	VARCHAR2(30)
POLICY_FUNCTION_OWNER	VARCHAR2(30)
PREDICATE	VARCHAR2(30)
DBMS_RLS.REFRESH_GROUPED_POLICY	VARCHAR2(4096)

EXEMPT ACCESS POLICY システム権限

システム権限 EXEMPT ACCESS POLICY を持つユーザーは、すべての DML 操作（SELECT、INSERT、UPDATE および DELETE）において、ファイングレイン・アクセス・コントロールの対象から除外されます。これによって、インストールや、SYS 以外のスキーマでのデータベースのインポートおよびエクスポートなどの管理アクティビティが使用しやすくなります。

また、使用しているユーティリティまたはアプリケーションに関係なく、ユーザーに EXEMPT ACCESS POLICY 権限が付与されている場合、そのユーザーは VPD および Oracle Label Security ポリシーの施行の対象から除外されます。ユーザーは、データ・アクセス時に、VPD または Oracle Label Security のポリシーを適用されません。

EXEMPT ACCESS POLICY は、ファイングレイン・アクセス・コントロールを無効にするため、この権限は、ファイングレイン・アクセス・コントロールの施行を回避する正当な理由を持つユーザーに対してのみ付与してください。ユーザーが他のユーザーに EXEMPT ACCESS POLICY 権限を譲渡して、ファイングレイン・アクセス・コントロールを回避する権限を伝播することがないように、この権限には With Admin Option を付加しないでください。

自動再解析

注意： この機能は、COMPATIBLE が 9.0.1 に設定されている場合に適用できます。

Oracle9*i* から、ファイングレイン・アクセスに対する問合せによって、オブジェクトでポリシー関数を常に実行して、最新の述語が各ポリシーに使用されているかどうかを確認できるようになりました。たとえば、時間ベースのポリシー関数の場合、問合せは午前 8 時から午後 5 時の間のみできます。正午に解析されるカーソルを実行すると、その実行時間にポリシー関数が実行され、ポリシー関数で再度問合せを参照するかどうかが確認されます。

この規則には 2 つの例外があります。1 つは、ポリシー関数が常に同じ述語を戻すことを示すポリシーを追加をするときに STATIC_POLICY=TRUE を指定することです。もう 1 つは、セキュリティ・ポリシーがデータベース・セッション内の異なる述語を戻さないユーザーの場合で、実行のオーバヘッド削減のために init.ora parameter _dynamic_rls_policies を FALSE に設定することです。

最新のアプリケーション・コンテキスト値が常に適切な値の配置環境の場合は、init.ora parameter _app_ctx_vers を FALSE に設定すると、古いアプリケーション・コンテキスト値を保持する必要がないため、オーバーヘッドを削減できます。デフォルトでは TRUE のため、SQL 内の値の変更は参照できません。将来、このデフォルトは変更される可能性があります。したがって、開発者は、SQL 内で実行されるユーザー定義関数のデータベースへの書込み禁止状態に関する他の要件と同様に、SQL 文内のアプリケーション・コンテキスト値を、ユーザー定義関数を使用して変更できないように注意する必要があります。一般的に、SQL 文内の実行順序に依存すると、問合せ計画によっては一貫性のない結果を得る可能性があるため、SQL 文内の実行順序には依存しないでください。

次回のリリースでは、これらの 2 つのパラメータは新しい機能に置き換えられ、使用できない場合があるため注意してください。

参照： 12-13 ページの「SYS_CONTEXT での動的 SQL の使用」を参照してください。

ファイングレイン監査

この項では、Oracle9*i* の監査機能のファイングレイン監査について説明します。内容は次のとおりです。

標準監査およびファイングレイン監査の概要

Oracle9*i* の標準監査では、権限およびオブジェクトを監視し、INSERT、UPDATE、DELETE などの DML 操作を監視するトリガーを提供します。一方、SELECT 文の監視には、内容に基づいてデータ・アクセスを監視できるファイングレイン監査を使用すると効果的です。この場合、監査条件を指定して、環境および問合せ結果に関するより正確な情報を得ることができます。この追加情報は、監査イベントの再作成、およびアクセス権に違反していないかどうかの判断に役立ちます。

たとえば、麻薬取締局では、局の情報データベースへのアクセスを詳細に追跡する必要があります。同様に、連邦税務局では、雇用者によるデータの傍受を防止するために、納税申告に関する情報へのアクセスを追跡する必要があります。このような局では、SCOTT が INFORMANTS 表で SELECT 権限を使用したというような単純な情報ではなく、アクセスされたデータを判断するために十分な詳細情報が必要です。ファイングレイン監査は、詳細情報のレベルを提供することができます。

Oracle9*i* の標準監査方法

Oracle では、ユーザーおよびサーバーの信頼性を監査する、170 以上の構成可能な監査オプションを提供しています。Oracle9*i* の監査機能を使用すると、文ごと、システム権限の使用ごと、オブジェクトごと、またはシステム管理者を含むユーザーごとにデータベース・アクティビティを監査できます。たとえば、すべてのユーザーによるデータベースへの接続時にアクティビティ全般を監査することも、特定のユーザーによる表の作成時に特定のアクティビティを監査することもできます。正常に実行された操作のみを監査することも、正常に実行されなかった操作のみを監査することもできます。正常に実行されなかった SELECT 文を監査すると、参照する権限を持たないデータにアクセスしようとしているユーザーを発見できる場合があります。

監査は高度に構成可能ですが、標準監査オプションに含まれる監査イベントに関する詳細はそれほど多くありません。通常、1 つの監査レコードでは、ユーザー、アクセスされたオブジェクト、使用された権限、アクセスが正常に行われたかどうか、およびタイムスタンプが識別されます。

監査レコードに自動的に含まれない情報は、トリガーを使用して記録するようにカスタマイズできます。この方法により、独自の監査条件および監査レコードの内容を設計することもできます。たとえば、EMP 表に、従業員の給料が 10%を超えて増加するたびに監査記録を生成するトリガーを定義するとします。これには、SALARY の昇給前および昇給後の値などの情報を選択して含めることができます。

```
CREATE TRIGGER audit_emp_salaries
AFTER INSERT OR DELETE OR UPDATE ON employee_salaries
for each row
begin
if (:new.salary> :old.salary * 1.10)
      then
      insert into emp_salary_audit values (
      :employee_no,
```

```
      :old.salary,
      :new.salary,
      user,
      sysdate);
      endif;
end;
```

さらに、イベント・トリガーを使用して、特定のユーザーのログイン時に監査オプションを使用可能にし、そのユーザーのログオフ時に使用禁止にすることもできます。

業務において、問合せからの結果セットの他に、実行された文を獲得する必要がある場合もあります。ファイングレイン監査では、データ・アクセス権の不正な使用を管理者にアクティブに警告するイベント・ハンドラの他に、詳細な監査を行うための主要な条件の定義をサポートする、拡張可能な監査メカニズムを提供しています。

Oracle9*i* では、データベースの監査証跡またはオペレーティング・システムの監査証跡（オペレーティング・システムが監査証跡を受信できる場合）に監査レコードを送信するオプションも提供しています。たとえば、データベース管理者の監査証跡は、通常、オペレーティング・システムの保護位置に書き込まれます。監査証跡をオペレーティング・システムに書き込むと、オペレーティング・システムのルート・ユーザーである監査人は、（ルート・アクセス権を所有していない）すべての DBA が実行したアクションに対する責任を DBA 自身に持たせることができます。このオプションを様々な監査オプション、カスタマイズ可能なトリガーまたはストアド・プロシージャに追加して使用すると、特定のビジネス・ニーズに適した監査体系を実装できます。

ファイングレイン監査

ファイングレイン監査メカニズムを使用すると、より詳細な監査を実現できます。監査の条件の選択には、表オブジェクトに対する単純なユーザー定義の SQL 述語が使用されます。フェッチ中、問合せブロックからポリシー条件を満たす行が戻ると、問合せが監査されます。

ファイングレイン監査を使用すると、監査イベントをトリガーするデータ・アクセスの条件を指定する監査ポリシーを定義したり、イベントのトリガーが発生したことを管理者に通知する柔軟なイベント・ハンドラを使用できます。たとえば、人事部社員に従業員の給料に関する情報へのアクセスを許可しながら、500,000 ドルを超える給料にアクセスした場合に監査イベントをトリガーできます。監査ポリシー（SALARY>500,000）は、監査ポリシー・インタフェース（PL/SQL パッケージ）を介して EMPLOYEES 表に適用されます。

実装時の柔軟性を向上させるため、ユーザー定義の関数を使用してポリシー条件を決定し、監査列を識別して監査ポリシーをさらに詳細にできます。たとえば、関数によって、ユーザーがイントラネット内でデータにアクセスしているかぎり、すべての給料情報へのアクセスを監査なしで許可し、役員クラスの給料にインターネットからアクセスした場合は、アクセスを監査することができます。関連列を使用すると、問合せで特定の列が参照されたときにのみ監査がトリガーされるため、不適切または不要な監査レコードの発生を削減できます。たとえば、従業員名がアクセスされたときのみ、役員の給料へのアクセスを監査できま

す。これは、人事部社員が対応する従業員名を選択していなければ、給料情報へのアクセスのみでは意味がないためです。

PL/SQL パッケージ DBMS_FGA を使用して、これらのファイングレイン監査ポリシーを管理できます。問合せブロックから監査条件に一致する行が戻された場合、これらの行は疑わしい行として識別されます。ユーザー名、SQL 文、ポリシー名、セッション ID、タイムスタンプおよびその他の属性の監査イベント・エントリは、監査証跡に挿入されます。オプションで、イベントを処理する監査イベント・ハンドラを定義することもできます。たとえば、イベント・ハンドラによって、管理者に警告ページを送信できます。

次の例は、*hr.emp* 表の SELECT 文を監査して、*sales* 部門の従業員記録の *salary* 列にアクセスするすべての問合せを監視する方法を示します。

```
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary');
```

次のいずれかの SQL 文によって、データベースが監査イベント・レコードを記録します。

```
SELECT count(*) FROM hr.emp WHERE dept = 'SALES' and salary > 10000000;
```

または

```
SELECT salary FROM hr.emp WHERE dept = 'SALES';
```

すべての使用可能な関連情報、およびトリガーのように動作するメカニズムを使用して、管理者は、記録する項目および監査イベントの処理方法を定義できます。

次のコマンドが発行された場合の動作を考えます。最初の疑わしい行のフェッチ後、イベントが記録され、監査関数 SEC.LOG_ID が実行されます。生成された監査イベント・レコードは、SQL 文、ポリシー名および他の情報を記録するための列が予約された DBA_FGA_AUDIT_TRAIL に格納されます。

```
/* create audit event handler */
CREATE PROCEDURE sec.log_id (schema varchar2, table varchar2, policy varchar2) AS
BEGIN
UTIL_ALERT_PAGER(schema, table, policy);      -- send an alert note to my pager
END;

/* add the policy */
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary',
```

```
handler_schema => 'sec',
handler_module => 'log_id',
enable               =>  TRUE);
```

注意： ファイングレイン監査は、コストベース最適化のみでサポートされています。ルールベース最適化を使用する問合せでは、行フィルタを適用する前に監査が行われるため、不要な監査イベント・トリガーが発生します。

参照：『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。

アプリケーション・セキュリティの施行

この項では、アプリケーション・セキュリティの施行について説明します。内容は次のとおりです。

- セキュリティに関する潜在的な問題となる非定型ツールの使用
- SQL*Plus ユーザーからのデータベースのロールの制限

セキュリティに関する潜在的な問題となる非定型ツールの使用

事前作成データベース・アプリケーションは、アプリケーション使用中にユーザーのロールを使用可能および使用禁止にすることも含めて、ユーザーのアクションを明示的に制御します。一方、SQL*Plus などの非定型の問合せツールを使用すると、ユーザーは付与されたロールを使用可能および使用禁止にする文も含めて、どのような SQL 文でも送信できます(正常終了する場合としない場合があります)。

アプリケーションのユーザーは、そのアプリケーションに連結された権限を使用して、非定型ツールによってデータベース表に破壊的な SQL 文を発行できます。

たとえば、次の使用例を考えます。

- Vacation アプリケーションはそれに対応する VACATION ロールを持っています。
- VACATION ロールには、EMP_TAB 表に対して SELECT、INSERT、UPDATE および DELETE 文を発行する権限が含まれています。
- Vacation アプリケーションは、VACATION ロールを介して取得した権限の使用を制御します。

ここで、VACATION ロールを付与されたユーザーを考えてみます。このユーザーが、Vacation アプリケーションを使用するかわりに、SQL*Plus を実行するとします。この時点でユーザーが制約を受けるのは、明示的に付与された権限またはロール（VACATION ロールを含む）を介して付与されている権限からのみです。SQL*Plus は非定型の問合せツールであるため、設計されたデータベース・アプリケーションを使用する場合のように、ユーザーは一連の事前定義アクションに制限されることはありません。ユーザーは、EMP_TAB 表のデータを自由に問合せまたは変更できます。

SQL*Plus ユーザーからのデータベースのロールの制限

この項では、SQL*Plus ユーザーのデータベースのロールを制限するために使用できる機能について説明します。次のような機能があります。

- PRODUCT_USER_PROFILE を介するロールの制限
- ビジネス・ロジックをカプセル化するストアド・プロシージャの使用
- 最高のセキュリティのための仮想プライベート・データベース
- 仮想プライベート・データベースおよび Oracle Label Security

PRODUCT_USER_PROFILE を介するロールの制限

Oracle9*i* では、PRODUCT_USER_PROFILE 表を介して、ユーザーがアプリケーションからアクセスするロールを制限できます。

DBA は、PRODUCT_USER_PROFILE を使用して、SQL*Plus 環境において、ある特定の SQL および SQL*Plus コマンドをユーザーごとに使用禁止にできます。Oracle ではなく SQL*Plus が、このセキュリティを施行します。DBA は、ユーザーによるデータベース権限の変更を制御するために、GRANT、REVOKE および SET ROLE コマンドへのアクセスを制限することもできます。

PRODUCT_USER_PROFILE 表を使用すると、ユーザーがアプリケーションでアクティブにできないロールをリストできます。また、様々なコマンド（SET ROLE など）の使用を明示的に禁止できます。たとえば、PRODUCT_USER_PROFILE 表にエントリを作成して、次の処理を行うことができます。

- SQL*Plus で、CLERK および MANAGER ロールの使用を禁止できます。
- SQL*Plus で、SET ROLE の使用を禁止できます。

ユーザー Jane が、SQL*Plus を使用してデータベースに接続するとします。Jane には、CLERK、MANAGER および ANALYST ロールがあります。前述の PRODUCT_USER_PROFILE のエントリによって、Jane は、SQL*Plus で ANALYST ロールのみを使用できます。また、Jane が SET ROLE 文を発行する場合、PRODUCT_USER_PROFILE 表にあるエントリが SET ROLE の使用を禁止するため、Jane によるこの文の発行は明示的に禁止されます。

PRODUCT_USER_PROFILE 表の使用は、様々な理由から、セキュリティを完全には保証しません。前述の例では、SET ROLE が SQL*Plus で禁止されていますが、Jane に直接付与され

ているその他の権限がある場合、Jane は SQL*Plus を使用してそれらの権限を使用できます。

> **参照：** PRODUCT_USER_PROFILE 表の詳細は、『SQL*Plus ユーザーズ・ガイドおよびリファレンス』を参照してください。

ビジネス・ロジックをカプセル化するストアド・プロシージャの使用

ストアド・プロシージャは、ビジネス・ロジックに権限の使用をカプセル化し、適切に作成されたビジネス・トランザクションのコンテキストでのみ権限が実行されるようにします。たとえば、アプリケーション開発者は、EMPLOYEES 表にある従業員の名前および住所を、通常の勤務時間内にのみ更新するプロシージャを作成できます。また、開発者（またはアプリケーション管理者）は、人事部社員に EMPLOYEES 表の UPDATE 権限を付与するのではなく、プロシージャのみに権限を付与できます。これによって、人事部社員が権限を使用できるのはプロシージャのコンテキスト内のみになり、EMPLOYEES 表を直接更新することはできなくなります。

最高のセキュリティのための仮想プライベート・データベース

Oracle9*i* では、仮想プライベート・データベース（VPD）を実装することにより、表、ビューまたはシノニムに対して、直接詳細レベルでセキュリティを施行できます。セキュリティ・ポリシーは、表、ビューまたはシノニムに直接連結されており、ユーザーがデータにアクセスすると自動的に適用されるため、セキュリティを回避できません。

集中管理され、データに直接適用される強力なセキュリティ・ポリシーによって、ユーザーのデータへのアクセス方法（アプリケーション、問合せ、レポート作成ツールなど）にかかわらず、セキュリティを施行できます。

ユーザーが VPD セキュリティ・ポリシーに対応付けられている表、ビューまたはシノニムに直接的または間接的にアクセスすると、サーバーはユーザーの SQL 文を動的に変更します。変更は、セキュリティ・ポリシーを実装する関数によって戻された WHERE 条件（述語）に基づいて行われます。SQL 文は、関数内に記述された条件、または関数が戻す条件を使用して、動的かつユーザーに対して透過的に変更されます。

また、述語を戻す関数は、その他の関数へのコールアウトを含むこともできます。PL/SQL パッケージ内に、オペレーティング・システム情報にアクセスしたり、または WHERE 句をオペレーティング・システム・ファイルやセントラル・ポリシー・ストアから戻す、C または Java コールアウトを埋め込むことができます。ポリシー関数は、各ユーザー、ユーザーの各グループまたは各アプリケーションに、異なる述語を戻すことができます。シノニムに対するポリシー関数の使用は、ユーザーまたはユーザーのクラスごとの個別ビューのメンテナンス、メモリー内への大量オーバーヘッドおよびリソースの処理の節減にかわる手段として使用可能です。

アプリケーション・コンテキストを使用すると、独自のセキュリティ・ポリシーが基づく属性に安全にアクセスできます。たとえば、manager の属性を持つユーザーには、employee の属性を持つユーザーとは異なるセキュリティ・ポリシーがあります。

航空部門の従業員記録の閲覧のみを許可された人事部社員がいるとします。そのユーザーは、次のように問合せを開始します。

```
SELECT * FROM emp;
```

セキュリティ・ポリシーを実装する関数は述語 `division = 'AIRCRAFT'` を戻し、データベースは問合せを透過的に再度書き込みます。実際に実行される問合せは、次のようになります。

```
SELECT * FROM emp WHERE division = 'AIRCRAFT';
```

セキュリティ・ポリシーは、アプリケーション内ではなく、データベース内で適用されます。これは、異なるアプリケーションを使用しても、セキュリティ・ポリシーが回避されないことを意味します。セキュリティは、複数のアプリケーションに再実装するのではなく、データベースに一度作成するのみです。このため、VPD は、アプリケーション・ベースのセキュリティよりも強力なセキュリティを提供し、所有権のコストもより低くなります。

データにアクセスしているアプリケーションに応じて異なるセキュリティ・ポリシーを施行することが望ましい場合があります。Order Entry および Inventory の 2 つのアプリケーションが、両方とも ORDERS 表にアクセスする場合を考えます。Inventory アプリケーションで、製品の種類に基づいてアクセスを制限するポリシーを表に適用するとします。同時に、Order Entry アプリケーションでも、顧客番号に基づいてアクセスを制限するポリシーを同一の表に適用するとします。

この場合、アプリケーションによるファイングレイン・アクセスの使用を分割する必要があります。そうしないと、2 つのポリシーが自動的に AND 処理され、目的とする結果が得られません。1 つ以上のポリシー・グループ、および特定のトランザクションに対して有効なポリシー・グループを判断する駆動アプリケーションのコンテキストを指定できます。また、データベースに必ず適用するデフォルト・ポリシーも指定できます。たとえば、ホストされたアプリケーションでは、データ・アクセスは必ずサブスクライバ ID によって制限されます。

参照： 12-7 ページの「ファイングレイン・アクセス・コントロールでのアプリケーション・コンテキストの使用方法」を参照してください。

仮想プライベート・データベースおよび Oracle Label Security

仮想プライベート・データベース（VPD）および Oracle Label Security は、ダイレクト・パスのエクスポート中には実行されません。また、VPD および Oracle Label Security のポリシーは、SYS スキーマのオブジェクトには適用できません。そのため、SYS ユーザーおよびデータベースに DBA 権限でアクセスするユーザー（CONNECT/AS SYSDBA など）の場合、操作に VPD または Oracle Label Security のポリシーが適用されません。データベース管理者は、データベースを管理できる必要があります。VPD ポリシーが適用されたため、表の一部のみのエクスポートでは不十分です。ただし、SYSDBA のアクションは、インストール時に監査を有効にし、監査証跡をオペレーティング・システムの保護位置に格納するように指定することによって監査可能です。

直接的に、またはデータベース・ロールによって、Oracle9*i* の EXEMPT ACCESS POLICY 権限を付与されているデータベース・ユーザーは、VPD および Oracle Label Security の施行の対象から除外されます。ユーザーは、データベースからデータを抽出するために使用するエクスポート・モード、アプリケーションまたはユーティリティに関係なく、VPD および Oracle Label Security の施行の対象から除外されます。EXEMPT ACCESS POLICY 権限は強力な権限であるため、慎重に管理する必要があります。通常は、除外の対象ユーザーはほとんどいないため、With Admin Option 付きのこの権限を付与することはお薦めしません。

注意： EXEMPT ACCESS POLICY 権限は、SELECT、INSERT、UPDATE および DELETE などのオブジェクト権限の施行には影響しません。これらのオブジェクト権限は、ユーザーに EXEMPT ACCESS POLICY 権限が付与されている場合も施行されます。

13

プロキシ認証

この章では、複数層システムのプロキシ認証について説明します。内容は次のとおりです。

- プロキシ認証のメリット
- 3 層コンピューティングのセキュリティ問題
- Oracle9i のプロキシ認証のソリューション

プロキシ認証のメリット

複数層環境では、プロキシ認証によって、すべての層でクライアントの認証および権限が保持され、クライアントのかわりに実行される操作が監査されるため、中間層アプリケーションのセキュリティを制御できます。たとえば、この機能を使用すると、Web アプリケーション（プロキシ）を使用してユーザーの認証を行い、アプリケーションを介してデータベース・サーバーに渡すことができます。

3 層システムを使用すると、組織には次のような多くのメリットがあります。

- アプリケーション・サーバーおよび Web サーバーによって、ユーザーは、レガシー・アプリケーションに格納されているデータにアクセスできます。
- ユーザーは、使い慣れたブラウザ・インタフェースを使用できます。
- 組織は、アプリケーション・ロジックをアプリケーション・サーバーに、データ記憶域をデータベースにパーティション化することによって、アプリケーション・ロジックとデータ記憶域を分離できます。
- 組織は、多くの Fat クライアントを Thin クライアントおよびアプリケーション・サーバーに置き換えることによって、コンピューティング・コストを低く抑さえることができます。

さらに、Oracle のプロキシ認証には、次のようなセキュリティ上のメリットがあります。

- 中間層がかわりに接続できるユーザーおよび中間層が、そのユーザーに対して想定できるロールを制御することによって制限付きトラスト・モデルが実現します。
- OCI および Thick JDBC で軽量ユーザー・セッションをサポートし、クライアント再認証のオーバーヘッドを排除することによってスケーラビリティが得られます。
- 実際のユーザーの認証をデータベースで保護し、実際のユーザーのかわりに行われる操作の監査を可能にする信頼性が得られます。
- ユーザーがデータベースに認識された環境と、ユーザーが単なる「アプリケーション・ユーザー」でデータベースには認識されていない環境の両方をサポートすることによって柔軟性が得られます。

注意： Oracle9*i* は、前述の機能を 3 層でのみサポートし、複数の中間層ではサポートしません。

---

3 層コンピューティングのセキュリティ問題

3 層コンピューティングには多くのメリットがありますが、多くの新しいセキュリティの問題も発生します。次の項では、これらの問題について説明します。

- 実際のユーザーとは
- 中間層の権限が適切であるか

実際のユーザーとは

多くの組織では、アクセス制御または監査の理由から、データベースに実際にアクセスしているユーザーの認証を把握する必要があります。ユーザーの認証がアプリケーションのすべての層でトレースできない場合、そのユーザーの信頼性は低下します。

さらに、アプリケーション・サーバーのみがそのユーザーを認識する場合、ユーザーごとのすべてのセキュリティは、アプリケーション自身によって施行される必要があります。アプリケーション・ベースのセキュリティは非常にコストがかかります。データにアクセスする各アプリケーションがセキュリティを施行すると、すべてのアプリケーションでセキュリティを再実装する必要が生じます。データベース内で施行されるユーザーごとの信頼性のためには、データ自身にセキュリティを作成する方が適切な場合もあります。

中間層の権限が適切であるか

企業内に 3 層システムを受け入れることを望む組織もあります。この場合、トランザクション処理（TP）モニターなどのすべての権限を持つ中間層は、すべてのユーザーに対してすべてのアクションを実行できます。このアーキテクチャでは、中間層は、すべてのアプリケーション・ユーザーに対する同じユーザーとして、データベースに接続します。そのため、このアーキテクチャには、アプリケーション・ユーザーがそのジョブを実行するために必要なすべての権限が必要です。

このコンピューティング・モデルは、ファイアウォール外、ファイアウォール上またはファイアウォール内に中間層が常駐するインターネットでは適切でない場合があります。このコンテキストの場合、制限付きトラスト・モデルがより適切です。制限付きトラスト・モデルでは、実際のクライアントの認証がデータベース・サーバーに認識され、アプリケーション・サーバー（または他の中間層）には制限付きの権限セットがあります。

また、中間層がかわりに接続できるユーザーおよび中間層が、そのユーザーに対して想定できるロールを制限する機能も有効です。たとえば、多くの組織では、ユーザーの接続元に基づいて、そのユーザーが異なる権限を持つようにする場合があります。ファイアウォール上の Web サーバーまたはアプリケーション・サーバーに接続するユーザーは、データにアクセスする最小の権限以外使用できませんが、企業内の Web サーバーまたはアプリケーション・サーバーに接続するユーザーは、許可されているすべての権限を実行できます。

監査方法および監査対象のユーザー

監査による信頼性は、情報セキュリティの基本原理です。ほとんどの組織では、トランザクションを実行する特定のアプリケーション・サーバーのみでなく、どのユーザーによってトランザクションが行われたかを認識する必要があります。そのため、システムは、トランザクションを実行しているユーザーと、ユーザーのかわりにトランザクションを実行しているアプリケーション・サーバーを区別する必要があります。

3 層システムでの監査は、実際のユーザーを認識する問題と対応付ける必要があります。3 層アプリケーションの中間層でユーザー ID を保持できない場合、そのユーザーのかわりのアクションは監査できません。

ユーザーがデータベースに対して再認証されるかどうか

クライアント / サーバー・システムでは、認証は単純です。クライアントはサーバーに対して認証されます。3 層システムでは、いくつかの潜在的な認証タイプがあるため、認証はより複雑になります。

- 中間層に対するクライアントの認証
- データベースに対する中間層の認証
- 中間層からデータベースへのクライアントの再認証

中間層に対するクライアントの認証

システムが基本セキュリティ原理に従う場合、中間層に対するクライアント認証が必要です。中間層は、通常、ユーザーがアクセスできる有効な情報への出発点になります。そのため、ユーザーを中間層に対して認証する必要があります。このような認証には相互関係があることに注意してください。つまり、クライアントが中間層に対して認証されるのと同じように、中間層はクライアントに対して認証されます。

データベースに対する中間層の認証

中間層は、通常、（データベース自身としてまたはユーザーのかわりに）データを取得するためにデータベースへの接続を開始する必要があるため、この接続が認証される必要があります。実際に、Oracle9*i* データベースは、認証されていない接続は許可しません。SSL などのデータベース認証をサポートするプロトコルを使用する場合は、データベースに対する中間層の認証についても共通にする必要があります。

中間層からデータベースへのクライアントの再認証

3 層システムでの中間層からデータベースへのクライアント再認証は、問題が発生します。ユーザー名が、中間層とデータベース上とでは異なる場合があるためです。この場合、ユーザーは、中間層がユーザーのかわりに接続するためのユーザー名およびパスワードを再入力する必要がある場合もあります。または、中間層は、指定されたユーザー名をデータベース・ユーザー名にマップする必要がある場合もあります。このマッピングは、Oracle Internet Directory などの LDAP 準拠のディレクトリ・サービスで行われます。

データベースに対して再認証するクライアントの場合、中間層がユーザーにパスワード（データベースに確実に渡される）を要求するか、または中間層がそのユーザーに対するパスワードを取得してユーザーを認証する必要があります。中間層は、ユーザーのパスワードを正常に処理し、そのパスワードを不正に使用することはないと信頼されているため、どちらの方法にもセキュリティ・リスクが伴います。

中間層への信頼を伴わない再認証のケースの 1 つとして、中間層がアプレットをクライアントにダウンロードし、クライアントがそのアプレットを介してデータベースに直接接続する場合があります。この場合、アプリケーション・サーバーは、アプリケーション（アプレット）をユーザーに提供するのみで、ユーザーにそれ以上の認証を求めることはありません。

バックエンド・データベースに対してクライアントを再認証することが常に有効とはかぎりません。各ユーザーに 2 組の認証を組み合せることによって、かなりのネットワーク・オーバーヘッドが発生します。また、ユーザーを認証させるために、中間層を信頼する必要があります（中間層がユーザーのパスワードを取得する場合、または中間層がユーザーのパスワードに関係している場合には、中間層を確実に信頼する必要があります）。そのため、中間層が適切な認証を行ったと判断することは、データベースにとって適切ではありません。つまり、データベースは、実際のクライアントにそのクライアント自身の認証を要求せずに、実際のクライアントの認証を受け入れます。

いくつかの認証プロトコルでは、クライアントの再認証ができない場合もあります。たとえば、多くのブラウザおよびアプリケーション・サーバーは、Secure Sockets Layer（SSL）プロトコルをサポートしています。（Oracle Advanced Security を介する）Oracle9*i* データベースおよび Oracle9*i* Internet Application Server は、クライアント認証に SSL の使用をサポートしています。ただし、SSL は Point-to-Point プロトコルであり、End-to-End プロトコルではありません。SSL は、データベースに対するブラウザ・クライアントの（中間層を介する）再認証には使用できません。

その理由は、ユーザーが、クライアントの再認証を行うために、中間層に対するそのユーザーの秘密鍵を安全に引き渡すことができないためです。ユーザーの秘密鍵が解決されると、ユーザーの認証も解決されます。また、ブラウザ・クライアントがデータベースに対して直接認証されるように、中間層を通過する方法はありません。

つまり、3 層システムを配置する組織には、クライアントの再認証に関する柔軟性が必要です。クライアントを再認証できない場合もあります。また、クライアントを再認証するかしないかを選択する場合もあります。

Oracle9*i* のプロキシ認証のソリューション

次の項では、Oracle9*i* が提供する特定の認証問題のソリューションについて説明します。

- 実際のユーザーの認証を介する引渡し
- 中間層の権限の制限
- 実際のユーザーの再認証
- 実際のユーザーのかわりに行われる操作の監査
- アプリケーション・ユーザー・モデルのサポート

実際のユーザーの認証を介する引渡し

多くの組織は、中間層のメリットを失わずに、アプリケーションのすべての層を介して実際のユーザーを認識する必要があります。Oracle9*i* は、アプリケーションの中間層を介してユーザーの認証を保持する多数の方法をサポートします。Oracle9*i* では、エンタープライズ・ユーザー（Oracle Internet Directory で管理される、データベースの共有スキーマにアクセスするユーザー）またはデータベース・ユーザーに、OCI または Thick JDBC によってプロキシ認証が提供されます。アプリケーション・ユーザー（アプリケーションには認識されるがデータベースには認識されないユーザー）には、OCI、Thick JDBC および Thin JDBC によってアプリケーション・ユーザーのプロキシ認証がサポートされます。

エンタープライズ・ユーザーまたはデータベース・ユーザーは、OCI または Thick JDBC を使用して、単一のデータベース接続内で、中間層に接続したユーザーを一意に識別する多数の軽量ユーザー・セッションを設定できます。これらの軽量セッションによって、中間層からデータベースへ個別にネットワーク接続を作成するために発生するネットワーク・オーバーヘッドが削減されます。アプリケーションは、これらのセッションを必要に応じて切り替え、ユーザーのかわりにトランザクションを処理できます。

データベースに対するクライアントから中間層への完全認証の順序を次に示します。

1. クライアントは、中間層が許容するすべての認証形式を使用して、中間層に対して認証します。たとえば、クライアントは、ユーザー名 / パスワード、または SSL による X.509 証明書を使用して、中間層に対して認証できます。
2. 軽量セッションを作成する中間層は、エンタープライズ・ユーザーではなく、データベース・ユーザーである必要があります。中間層は、Oracle9*i* が許容するすべての認証形式を使用して、中間層自体を Oracle9*i* に対して認証します。これには、パスワード、または Kerberos チケットや X.509 証明書（SSL）などの Oracle Advanced Security がサポートする認証メカニズムなどがあります。
3. 認証後、中間層は OCI または Thick JDBC を使用して、ユーザーに対して 1 つ以上のセッションを作成します。
   - ユーザーがデータベース・ユーザーの場合、軽量セッションには少なくともデータベース・ユーザー名が含まれている必要があります。データベースが要求する場合は、セッションに（データベースがデータベース内のパスワード・ストアに対して検証する）パスワードが含まれている必要があります。また、セッションに、ユーザーに対するデータベース・ロールのリストが含まれる場合もあります。
   - ユーザーがエンタープライズ・ユーザーの場合、軽量セッションが、ユーザーの認証方法によって、異なる情報を提供する場合があります。ユーザーが SSL を介して中間層に認証された場合、中間層は、ユーザーの X.509 証明書またはセッション内の証明書自体から DN を提供できます。データベースは、DN を使用して、Oracle Internet Directory でユーザーを検索します。ユーザーがパスワードで認証されるエンタープライズ・ユーザーの場合、中間層は、少なくともグローバルに一意のユーザー名を提供する必要があります。データベースは、この名前を使用して、Oracle Internet Directory でユーザーを検索します。セッションもユーザーに対するパスワードを提供する場合、データベースは、Oracle Internet Directory に対してそのパ

スワードを検証します。ユーザーのロールは、セッションが確立した後、自動的に Oracle Internet Directory から取得されます。

- ユーザーがエンタープライズ・ユーザーではなく、データベース・ユーザーの場合、中間層は、オプションでクライアントに対するデータベース・ロールのリストを提供します。ユーザーがエンタープライズ・ユーザーの場合、ユーザーのロールは、セッションが確立した後、自動的に Oracle Internet Directory から取得されます。

4. ユーザーがデータベース・ユーザーの場合、データベースは提供されたロールを使用して、ユーザーのかわりにセッションを作成する権限が中間層にあるかどうかを検証します。

   アプリケーション・サーバーがクライアントのプロキシとなることが管理者によって許可されていない場合、またはアプリケーション・サーバーが特定のロールをアクティブにすることを許可されていない場合、`OCISessionBegin` コールは正常に実行されません。

中間層の権限の制限

最小の権限とは、ユーザーが、その目的を実行するために必要最小限の権限のみを持ち、それ以上の権限は持たないという原理です。中間層アプリケーションに適用されるように、これは、中間層に必要以上の権限を設定するべきではないことを表します。Oracle9*i* では、特定のデータベース・ロールのみを使用して、中間層が特定のデータベース・ユーザーのかわりとしてのみ接続できるように制限できます。中間層がエンタープライズ・ユーザーのかわりに接続するための権限、または OCI または Thick JDBC 軽量接続でのエンタープライズ・ロールが付与されたユーザーは制限できません。

たとえば、ユーザー Sarah が中間層 `appsrv`（データベース・ユーザーでもある）を介してデータベースに接続するとします。Sarah には複数のロールがありますが、Sarah のかわりに `clerk` ロールのみを使用するように中間層を制限するとします。

DBA は、次の構文を使用して、Sarah のかわりに `clerk` ロールのみを使用して接続を開始する許可を `appsrv` に付与できます。

```
ALTER USER Sarah GRANT CONNECT THROUGH appsrv WITH ROLE clerk;
```

デフォルトでは、中間層は、すべてのクライアントに対して接続できません。許可はユーザーごとに付与される必要があります。

クライアント Sarah に付与されているすべてのロールの使用を `appsrv` に許可するには、次の文を使用します。

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv;
```

中間層が別のデータベース・ユーザーの軽量（OCI）セッションまたは Thick JDBC セッションを開始するたびに、データベースは指定されたロールを使用して、そのユーザーに対して接続する権限が中間層にあるかどうかを検証します。

中間層に下位層のデータベース・ユーザーとして接続できる権限を付与することによって、中間層がエンタープライズ・ユーザーのかわりに接続する権限を制限できます。たとえば、エンタープライズ・ユーザーが APPUSER スキーマにマップされた場合、中間層には、少なくとも APPUSER のかわりに接続できる権限を付与する必要があります。付与しない場合、エンタープライズ・ユーザーに対するセッションの作成が正常に実行されません。

実際のユーザーの再認証

データベース・パスワードによって認証する場合、クライアントのパスワードは中間層サーバーに渡されます。次に、中間層サーバーは、そのパスワードを属性として、検証のためにデータ・サーバーに渡します。これによる主なメリットは、クライアント・マシンに、Oracle ソフトウェアを実際にインストールする必要がないということです。

前述のとおり、中間層がユーザーを認証した後で、それらのユーザーをデータベースに対して再認証することが必ずしも有効とはかぎりません。ただし、追加のセキュリティ対策としてこのような再認証を行う場合は、OCIAttrSet コールの OCI_ATTR_PASSWORD 属性を使用して、データベースにユーザーのパスワードを渡すことができます。

> **参照：** 3 層アーキテクチャのセキュリティの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

エンタープライズ・ユーザーでのプロキシ認証の使用

中間層がエンタープライズ・ユーザーであるクライアントとしてデータベースに接続している場合、識別名または識別名を含む X.509 証明書のいずれかが、データベース・ユーザー名のかわりに渡されます。ユーザーがパスワードで認証されるエンタープライズ・ユーザーの場合、中間層は、少なくともグローバルに一意のユーザー名を提供する必要があります。データベースは、この名前を使用して、Oracle Internet Directory でユーザーを検索します。

クライアントの識別名を渡すために、アプリケーション・サーバーは、次の疑似インターフェースを使用して OCIAttrSet() をコールします。

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
lxstp *distinguished_name,
(ub4) 0,
OCI_ATTR_DISTINGUISHED_NAME,
OCIError *error_handle);
```

証明書全体を渡すには、中間層は次の疑似インタフェースを使用します。

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
ub1 *certificate,
ub4 certificate_length,
OCI_ATTR_CERTIFICATE,
OCIError *error_handle);
```

注意： OCI_ATTR_CERTIFICATE は、DER でコード化されます。

証明書のタイプが指定されない場合、サーバーはデフォルト証明書タイプの X.509 を使用します。

パスワードで認証されるエンタープライズ・ユーザーに対してプロキシ認証を使用する場合は、パスワード（OCI_ATTR_USERNAME など）によって認証されるデータベース・ユーザーと同じ OCI 属性を使用します。データベースは、まずデータベースに対してユーザー名を確認します。ユーザーが見つからない場合は、ディレクトリでユーザー名を確認します。このユーザー名は、グローバルに一意である必要があります。

実際のユーザーのかわりに行われる操作の監査

Oracle9*i* のプロキシ認証機能を使用すると、ユーザーのかわりに中間層が実行する操作を監査できます。たとえば、アプリケーション・サーバー hrappserver が、ユーザー Ajit および Jane に対して複数の軽量セッションを作成するとします。DBA は、hrappserver が Jane のかわりに次のように開始する bonus 表に対する SELECT を監査できるようにします。

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF Jane;
```

また、中間層を介して接続している複数ユーザー（この場合は Jane および Ajit）のかわりに、次のようにして監査を使用可能にすることもできます。

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF ANY;
```

この監査オプションは、他のユーザーのかわりに hrappserver によって開始された SELECT 文の監査のみを行います。DBA は、個別の監査オプションを使用可能にして、データベースに直接接続しているクライアントから、bonus 表に対する SELECT を獲得できます。

```
AUDIT SELECT TABLE.
```

実際のユーザーのかわりに行われる操作の監査については、識別名がデータベースでは認識されないため、CONNECT ON BEHALF OF *DN* を監査できません。ただし、ユーザーが共有スキーマ（APPUSER など）にアクセスする場合は、CONNECT ON BEHALF OF APPUSER を監査できます。

アプリケーション・ユーザー・モデルのサポート

多くのアプリケーションでは、セッション・プーリングを使用して、マルチユーザーによって再利用される多くのセッションが設定されます。ここでいう、「アプリケーション・ユーザー」とは、アプリケーションの中間層に対しては認証されますが、データベースでは認識されないユーザーを示します。Oracle9*i* は、これらのタイプのアプリケーションに対するアプリケーション・ユーザー・プロキシをサポートします。

このアプリケーション・ユーザー・モデルでは、中間層が、セッション確立時にクライアント識別子をデータベースに渡します（クライアント識別子は、実際には Cookie や IP アドレスなど、中間層に接続しているクライアントを表すものです）。アプリケーション・ユーザーを表すクライアント識別子は、ユーザー・セッション情報の中で使用可能です。また、アプリケーション・コンテキスト（USERENV ネーミング・コンテキスト）を介してアクセスすることもできます。これによって、アプリケーションは、セッション内のアプリケーション・ユーザーを追跡しながら、セッションを設定および再利用できます。アプリケーションはクライアント識別子をリセットできるため、異なるユーザーにセッションを再利用して、高パフォーマンスを実現します。OCI ベースの接続では、CLIENT_IDENTIFIER の変更が他の OCI コールによって伝達されるため、パフォーマンスはさらに向上します。アプリケーション・ユーザー・プロキシは、Thin JDBC、Thick JDBC および OCI で使用でき、異なるユーザー・セッション（軽量セッションも含む）のオーバーヘッドなしで、接続プーリングのメリットを提供します。

アプリケーション・ユーザー・プロキシは、グローバル・アプリケーション・コンテキストで使用できるため、アプリケーションの作成がさらに柔軟になり、パフォーマンスが向上します。たとえば、ビジネス・パートナに情報を提供する Web ベースのアプリケーションには、ゴールド・パートナ、シルバー・パートナまたはブロンズ・パートナの 3 つのタイプのユーザーがあると想定します。これらのパートナは、それぞれに使用可能な情報レベルが異なります。アプリケーションには、ユーザーがデータベースで認識されるかどうかは重要ではありません。ただし、アプリケーションは、それぞれゴールド・パートナ、シルバー・パートナまたはブロンズ・パートナを表すコンテキストに基づいて、データ・アクセスを制限する必要があります。各ユーザーが、個別のアプリケーション・コンテキストが設定された独自のセッションを持つかわりに、アプリケーションは、ゴールド・パートナ、シルバー・パートナまたはブロンズ・パートナ用のグローバル・アプリケーション・コンテキストを設定し、クライアント識別子を使用して正しいコンテキストのセッションを示すことによって、適切なタイプのデータを取得します。アプリケーションは、この 3 つのグローバル・コンテキストを初期化し、クライアント識別子を使用して正しいアプリケーション・コンテキストにアクセスするのみで、データ・アクセスを制限できます。このように、各セッションのアプリケーション・コンテキストを個別に初期化するかわりに、セッションを再利用し、設定済のグローバル・アプリケーション・コンテキストに 1 回のみアクセスすることによって、パフォーマンスが向上します。

参照：

- 12-32 ページの「アプリケーション・コンテキストのグローバルな初期化」を参照してください。
- 『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

14

DBMS_OBFUSCATION_TOOLKIT を使用したデータの暗号化

この章では、データベース内のデータを暗号化できる、データ暗号化のパッケージ（DBMS_OBFUSCATION_TOOLKIT）について説明します。内容は次のとおりです。

機密情報の保護

インターネットで、特に E-Business の展開を目指す組織にとっては、情報セキュリティに関する新しい問題が発生しています。これらの多くのセキュリティ問題は、次に示す従来のセキュリティ・メカニズム方法によって解決できます。

- 厳正なユーザー認証によるユーザーの識別
- より詳細なアクセス制御によるユーザーのアクセス権限の制限
- 信頼性の監査
- ネットワーク暗号化による、送信時における機密性の高いデータの機密保護

暗号化は、前述のいくつかのソリューションにおける重要な構成要素です。たとえば、インターネットの標準ネットワーク暗号化および認証プロトコルである SSL では、暗号化を使用して、X.509 デジタル証明書によってユーザーが厳正に認証されます。また、SSL では、暗号化を使用してデータの機密性が保証され、暗号化チェックサムを使用してデータの整合性が保証されます。これらの多くの暗号化は、ユーザーまたはアプリケーションに対して比較的透過的に使用されます。たとえば、SSL は多くのブラウザでサポートされており、通常、ユーザーが特別な作業を行わなくても、SSL 暗号化は使用可能になります。

Oracle7 以上では、データベース・クライアントと Oracle データベース間のネットワーク暗号化が提供されています。Oracle9*i* のオプションである Oracle Advanced Security では、Oracle Net、JDBC（Thick JDBC と Thin JDBC の両方）、Internet Intra-Orb Protocol（IIOP）を含めて、Oracle9*i* がサポートするすべてのプロトコルの暗号化および暗号化の整合性チェックが提供されています。また、Oracle Advanced Security では、Oracle Net、Thick JDBC および IIOP 接続の SSL もサポートされています。

暗号化によってすべてのセキュリティ問題が解決されるわけではありませんが、暗号化は、特定のセキュリティ侵害の対策に使用される重要なツールです。特に、格納データの暗号化の分野は、E-Business の発展に伴い、ますます重要になります。たとえば、通常、クレジット・カード番号は Web サイトへの転送中は SSL で保護されますが、ファイル・システム上またはデータベース内では、クリアテキストで（暗号化されずに）格納されています。ファイル・システムでは、ホストに侵入したり、ルート・アクセスすることができるユーザーによって、データが不正アクセスされる可能性があります。データベースは、適切に構成すると完全に保護できますが、ホストを正しく構成していない場合は、ホストに侵入される可能性があります。いくつかの有名な侵入例では、ハッカーがデータベースに侵入し、大量のクレジット・カード番号のリストを入手しています。

このように、格納データの暗号化は E-Business の新しい問題点であり、特定のセキュリティ侵害に対する対策では重要なツールとなります。

データ暗号化の原則

データの暗号化には、多くのメリットがありますが、多くのデメリットもあります。暗号化によってすべてのセキュリティ問題が解決されるわけではなく、問題によっては事態が悪化する場合もあります。次の項では、格納データの暗号化に関するいくつかの誤解について説明します。内容は次のとおりです。

- 原則 1: 暗号化ではアクセス制御問題を解決できない
- 原則 2: 暗号化では不当な DBA から保護できない
- 原則 3: すべてのデータを暗号化してもデータを保護できない

原則 1: 暗号化ではアクセス制御問題を解決できない

ほとんどの組織では、データを参照する必要があるユーザーのみに、データのアクセスを制限する必要があります。たとえば、人事システムでは、従業員には、各自の雇用レコードのみを参照できるように制限し、マネージャには、部下の雇用レコードを参照できるようにする場合があります。また、人事部門の担当者が、多数の従業員の雇用レコードを参照する必要がある場合もあります。

このタイプのセキュリティ・ポリシー（データを参照する必要があるユーザーのみに、データのアクセスを制限する）は、通常、アクセス制御メカニズムによって解決されます。Oracle データベースには、長年にわたって、独自に評価された強力なアクセス制御メカニズムが提供されています。現在、Oracle9*i* には、仮想プライベート・データベース機能を介して、きわめて詳細なレベルでアクセス制御を施行できる機能が追加されています。

人事部門のレコードは機密性の高い情報であると考えられているため、すべての情報は、より強力なセキュリティのために暗号化する必要があると考えられる傾向があります。ただし、暗号化では前述の詳細なアクセス制御を施行することはできないため、実際にはデータ・アクセスの妨げとなる場合があります。人事部門の例では、従業員、マネージャおよび人事部門の担当者が、従業員レコードにアクセスする必要があります。従業員データ が暗号化される場合、それぞれが暗号化されていない形式のデータにもアクセスできる必要があります。したがって、従業員、マネージャおよび人事部門の担当者は、同じ暗号化キーを共有してデータを復号化する必要があります。そのため、暗号化では、アクセス制御の強化の観点ではセキュリティが強化されず、実際には適切なアプリケーションの機能が妨げられる場合があります。その他の問題としては、複数のシステム・ユーザー間で、暗号化キーを安全に転送および共有することが非常に困難なことがあげられます。

格納データを暗号化する場合の基本原則は、暗号化によってアクセス制御が妨げられないようにすることです。たとえば、EMP 表の SELECT 権限を持つユーザーが、暗号化メカニズムによって、基本的には参照できるすべてのデータへの参照を制限されないようにする必要があります。また、ユーザーが表のすべての暗号化データを参照する必要がある場合は、あるキーを使用して表の一部を暗号化し、別のキーを使用して表の他の部分を暗号化するメリットはほとんどありません。ユーザーがデータを参照する前に、データを復号化するオーバーヘッドが増加するのみです。適切なアクセス制御が実装されている場合、暗号化によってデータベース自体が得られるセキュリティ強化はほとんどありません。データベース内の

データにアクセス権を持つユーザーに対しては、暗号化しても権限は変わりません。したがって、暗号化を使用して、アクセス制御問題を解決しないでください。

原則 2: 暗号化では不当な DBA から保護できない

いくつかの組織では、不正なユーザーが、パスワードを推測することによって、上位権限（DBA）を取得できることを懸念しています。これらの組織では、格納データを暗号化して、この侵害から保護しようと考えます。ただし、この問題の適切な解決策は、DBA アカウントを保護し、他の権限が付与されたアカウントのデフォルトのパスワードを変更することです。データベースに侵入する最も簡単な方法は、権限が付与されたアカウントの変更されていないパスワード（SYS/CHANGE_ON_INSTALL など）を使用することです。

不正なユーザーが DBA 権限を取得すると、この他にもデータベースに対して多くの破壊行為を行うことができます。たとえば、データの破壊または削除、ファイル・システムへのユーザー・データのエクスポートによる電子メールでのデータ返送を介したパスワード・クラッカーの実行などを含みます。暗号化では、このような不正行為からデータを保護できません。

いくつかの組織では、通常、すべての権限が付与された DBA が、データベース内のすべてのデータを参照できることを懸念しています。これらの組織では、DBA はデータベースを管理するのみで、データベースに含まれるデータを参照できないようにする必要があると考えます。また、いくつかの組織では、1 人のユーザーに権限が集中することを懸念し、DBA の機能を分割するか、または 2 人の DBA をおくルールを施行しています。

すべてのデータ（または大量のデータ）を暗号化すると、前述の問題が解決できると考える傾向がありますが、このような不正行為からデータを保護するよりよい方法が他にあります。まず、Oracle では、DBA 権限の制限付きの分割がサポートされていません。Oracle9*i* では、SYSDBA および SYSOPER ユーザーに対する固有のサポートが提供されています。SYSDBA はすべての権限を持っていますが、SYSOPER は制限付きの権限セット（データベースの起動および停止など）を持っています。さらに、組織は、多数のシステム権限を含む下位ロールを作成できます。たとえば、JR_DBA ロールには、すべてのシステム権限は含まれませんが、データベースの副管理者に相当する権限（CREATE TABLE や CREATE USER など）のみが含まれる場合があります。Oracle は、SYS（または SYS の権限を持つユーザー）が実行する操作の監査、および安全性の高いオペレーティング・システム位置への監査証跡の格納ができます。この方法を使用すると、オペレーティング・システムにルートを持つ個別の監査人が SYS によるすべてのアクションを監査し、監査人が、すべての DBA を、実行したアクションに対して報告可能な状態にできます。

> **参照：**『Oracle9*i* データベース管理者ガイド』の audit_sys_operations パラメータを参照してください。

さらに、DBA は、基本的に、信頼される立場にあります。最も機密性の高いデータを持つ組織（情報機関など）でも、通常、DBA 機能は分割されません。そのかわりに、DBA は信頼される立場にあるため、十分に調査されます。定期的な監査は、不適当なアクティビティの発見に有効です。

格納データの暗号化によって、データベース管理を妨げることはできません。そうしないと、より重大なセキュリティ問題を招く結果となります。たとえば、データの暗号化によってデータが破損した場合、データが無効になり、リカバリ不可能になるという別のセキュリティ問題が発生します。

暗号化を使用して、DBA（または他の権限を持っているユーザー）がデータベース内のデータを参照可能な権限を削減することができます。ただし、これによって、DBA を適切に調査したり、強力なシステム権限の使用を制限することができるわけではありません。信頼できないユーザーが重要な権限を持っている場合、そのユーザーが組織に与える影響は大きく、状況は暗号化されていないクレジット・カード番号を参照されるより、はるかに深刻な場合があります。

原則 3: すべてのデータを暗号化してもデータを保護できない

データを暗号化して格納するとセキュリティが強化され、すべてのデータを暗号化すると、すべてのデータを保護できる、という考え方が浸透しています。

前述のとおり、暗号化ではアクセス制御問題が適切に解決されません。本番データベース全体を暗号化した結果について考えてみます。すべてのデータを読み込み、更新または削除するために、データを復号化する必要があります。ただし、暗号化によって通常のアクセス制御を妨げることはできません。暗号化は、パフォーマンスが集中する操作であるため、すべてのデータを暗号化すると、パフォーマンスが大きく影響されます。可用性は、セキュリティの重要な側面です。データを暗号化すると、データが使用不可能になったり、パフォーマンスが可用性に悪影響を与えるという別のセキュリティ問題が発生する場合があります。

セキュリティ強化の一環として、暗号化キーを定期的に変更する必要があります。この場合、単一または複数の暗号化キーを使用してデータを復号化および再暗号化している間は、データベースをアクセス不可能にする必要があります。これも、可用性に悪影響を与えます。

本番データベースのすべてのデータまたはほとんどのデータを暗号化することは明らかに問題ですが、オフラインで格納されるデータの暗号化にはメリットがある場合もあります。たとえば、組織が 6 か月～ 1 年間、別の場所にオフラインでバックアップを格納する場合があります。まず、物理的に、アクセスが制御される設備内にデータを格納する必要があります。ただし、このデータを格納する前に、データの暗号化を行うメリットがある場合もあります。データはオンラインでアクセスされないため、パフォーマンスを考慮する必要はありません。Oracle9*i* ではこのような設備が提供されていませんが、いくつかのベンダーはこのような暗号化サービスを提供しています。この方法を検討している組織は、大規模なバックアップ・データの暗号化を実行する前に、プロセスを徹底的にテストする必要があります。オフラインで格納する前に暗号化したすべてのデータが、正常に復号化および再インポートできる必要があります。

Oracle9*i* における格納データの暗号化のソリューション

DBMS_OBFUSCATION_TOOLKIT には、前述のセキュリティ問題を解決するためのいくつかの方法があります。この項の内容は次のとおりです。

Oracle9*i* のデータ暗号化機能

暗号化では適切に解決できないセキュリティ侵害は多くありますが、データベースに格納する前に、機密性の高いデータを選択的に暗号化することによって、追加のセキュリティ対策を得ることができます。このようなデータの例は次のとおりです。

- クレジット・カード番号
- 国民識別番号
- ユーザーがデータベース・ユーザーではないアプリケーションに対するパスワード

Oracle9*i* では、これらのニーズに応えるため、格納データを暗号化および復号化するための PL/SQL パッケージが提供されています。DBMS_OBFUSCATION_TOOLKIT パッケージは、Oracle9*i* Standard Edition と Oracle9*i* Enterprise Edition の両方で提供されています。現在、このパッケージでは、データ暗号化規格（DES）アルゴリズムを使用したバルク・データの暗号化がサポートされ、DES を使用した暗号化（DESEncrypt）および復号化（DESDecrypt）のプロシージャが含まれます。DBMS_OBFUSCATION_TOOLKIT には、外部暗号ブロック連鎖（CBC）モードで、2 つおよび 3 つのキーの DES を使用して暗号化および復号化するファンクションも含まれます。キーの長さは、それぞれ 128 および 192 ビット必要です。

DBMS_OBFUSCATION_TOOLKIT には、暗号化チェックサム機能（MD5）および安全な乱数を生成する機能（GetKey）が含まれます。安全な乱数の生成は、暗号化の重要な部分です。予測可能なキーは簡単に推測されるため、簡単にデータを復号化される可能性があります。ほとんどの暗号は、総当たり分析（可能性があるすべてのキーを循環させる）ではなく、簡単に解読可能なキーまたは格納方法が単純なキーを発見することによって、解読されます。

注意： DBMS_RANDOM は、暗号化キーの生成には適切でないため使用しないでください。

キー管理は、プログラムで行います。アプリケーション（またはファンクションのコール側）は、暗号化キーを提供する必要があります。そのため、アプリケーション開発者は、キーを安全に格納し、取得する方法を検討する必要があります。様々なキー管理の方法に関連するメリットおよびデメリットは、次の項を参照してください。文字列と RAW データの

両方を処理できる DBMS_OBFUSCATION_TOOLKIT パッケージには、64 ビットのキーを送る必要があります。DES アルゴリズム自体の有効なキーの長さは、56 ビットです。

DBMS_OBFUSCATION_TOOLKIT には、デフォルトで PUBLIC 権限が付与されています。この権限を取り消すことをお薦めします。通常、ユーザーは、アプリケーションのコンテキスト外で、格納データを暗号化する必要がないためです。

参照： DBMS_OBFUSCATION_TOOLKIT パッケージについては、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

データ暗号化の問題

暗号化によって追加セキュリティが提供される場合でも、技術的な問題はあります。次の項では、このような問題について説明します。内容は次のとおりです。

- 索引データの暗号化
- キーの管理
- キーの転送
- キーの格納
- 暗号化キーの変更
- バイナリ・ラージ・オブジェクト（BLOB）

索引データの暗号化

索引付けされた暗号化データの処理には、特殊な問題が発生します。たとえば、企業が国民識別番号（アメリカの社会保障番号（SSN）など）を従業員番号として使用するとします。企業は、従業員番号を非常に機密性の高いデータであると考えているため、EMPLOYEES 表の EMPLOYEE_NUMBER 列のデータを暗号化します。EMPLOYEE_NUMBER には一意の値が含まれているため、データベース設計者は、パフォーマンスを向上するためにデータを索引付けします。

ただし、DBMS_OBFUSCATION_TOOLKIT（または別のメカニズム）を使用して列内のデータが暗号化される場合は、列の索引にも暗号化された値が含まれます。列の索引に暗号化された値が含まれる場合、等価性の検査（'SELECT * FROM emp WHERE employee_number = '123245'）に索引を使用できる場合でも、その索引を他の目的に使用することはできません。このため、開発者が索引データを暗号化することはお薦めしません。

この問題を解決する方法の 1 つは、国民識別番号の暗号化を検討している企業の場合、各従業員の番号を一意に識別する他の番号を作成することです。これによって、これらの代替従

業員番号に索引を作成し、クリアテキストで保持することができます。対応する国民識別番号は、索引付けしないで別の列に格納し、復号化も適切に処理できるアプリケーションによって列の値を暗号化させることができます。この方法では、必要に応じて国民識別番号が取得されますが、従業員を識別する一意の番号としては使用されません。

国民識別番号の乱用に関連するプライバシー問題（個人情報の盗難）、一意であるはずの国識別番号に（アメリカの社会保障番号と）重複している番号があるという事実、および順序を使用して一意の番号を簡単に生成できることを考慮すると、国民識別番号を一意の ID として使用することが適切でないことがわかります。

キーの管理

安全な暗号化キーの生成問題を解決するため、Oracle9*i* では、安全な乱数を生成する DBMS_OBFUSCATION_TOOLKIT の GetKey プロシージャのサポートが追加されています。GetKey プロシージャは、以前は Oracle Advanced Security FIPS-140 の評価の一部として米国連邦情報処理標準（FIPS）の FIPS-140 に対して保証されていた、安全な乱数ジェネレータ（RNG）をコールします。開発者は、どのような状況でも、DBMS_RANDOM パッケージを使用しないでください。DBMS_RANDOM パッケージは、疑似乱数を生成します。RFC-1750 には、「疑似乱数プロセスを使用して機密の数を生成すると、疑似セキュリティを招く」と示されています。

キーの転送

キーがアプリケーションによってデータベースに渡される場合、キーを暗号化する必要があります。キーを暗号化しない場合、キーの転送時に、傍受者にキーを盗まれることがあります。Oracle Advanced Security で提供されているようなネットワーク暗号化を使用すると、暗号化キーを含む、転送中のすべてのデータを修正または傍受から保護できます。

キーの格納

キーの格納は、暗号化の最も重要で複雑な側面の 1 つです。対称鍵で暗号化されたデータを元の状態に戻すには、データを復号化するアプリケーションまたはユーザーが、キーにアクセスできる必要があります。キーは、パフォーマンスを大幅に低下させることなく、ユーザーが暗号化データにアクセスできるように、簡単に取得できる必要があります。キーを十分に保護して、アクセス権のない暗号化データに不正アクセスしようとするユーザーが、簡単にキーを元の状態に戻せないようにする必要があります。

開発者が使用できる 3 つの基本的なオプションは、次のとおりです。

- データベースへのキーの格納
- オペレーティング・システムへのキーの格納
- ユーザーによるキーの管理

データベースへのキーの格納

DBA の暗号化データへのアクセスを防ぐ場合、データベースにキーを格納すると、常に完全なセキュリティを得られない可能性があります。これは、すべての権限を持つ DBA は、暗号化キーを含む表にアクセスできるためです。ただし、傍受者またはオペレーティング・システム上のデータベース・ファイルを侵害するユーザーに対しては、非常に適切なセキュリティ対策となる可能性が高い方法です。

たとえば、従業員データを含む表（EMP）の作成について考えてみます。各従業員の社会保障番号（1 つの列）を暗号化します。別の列に格納されているキーを使用して、各従業員の SSN を暗号化できます。ただし、表全体に対する SELECT 権限を持っているユーザーは、暗号化キーを取得し、対応する SSN を復号化することができます。

この暗号化スキームは簡単に侵害できるように見えますが、わずかな工数で、非常に強力な不正侵入対策ソリューションを作成できます。たとえば、この方法で SSN を暗号化する前に、employee_number 上でデータをさらに変換する方法で、SSN を暗号化できます。この方法は、employee_number と従業員の誕生日の XOR（排他的論理和）演算を行うことと 同様に簡単です。

さらなる保護として、暗号化を実行する PL/SQL パッケージ本体を（WRAP ユーティリティを使用して）ラップできます。これによって、コードがわかりにくくなります。開発者は、次のとおり、KEYMANAGE というパッケージ本体をラップできます。

```
wrap iname=/mydir/keymanage.sql
```

開発者は、次に、ラップされたパッケージに含まれるキーを使用して、パッケージ内のファンクションに DBMS_OBFUSCATION_TOOLKIT をコールさせることができます。

ラップは解読不可能ではありませんが、傍受者がキーを取得することはより困難になります。リテラルは、パッケージ・ファイル内では読込み可能なため、キーをパッケージ内で分割し、使用前にプロシージャで再作成できます。各暗号化データの値に異なるキーが提供され、キーの値がパッケージ内に埋め込まれない場合でも、キーの管理を実行するパッケージをラップすること（データの変換またはデータの埋込み）をお薦めします。

参照： WRAP ユーティリティの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス 』を参照してください。

また、暗号化キーを格納する表を個別に作成し、プロシージャでキー表に対するコールを包むという方法もあります。主キー・外部キー関係を使用して、キー表をデータ表に結合できます。たとえば、EMPLOYEE_NUMBER は、従業員の情報および暗号化された SSN を格納する EMPLOYEES 表の主キーです。EMPLOYEE_NUMBER は、各従業員の SSN の暗号化キーを格納する SSN_KEYS 表に対する外部キーです。SSN_KEYS 表に格納されるキーも、使用する前に（XOR 演算により）変換できるため、キー自体は暗号化されずに格納されることがありません。プロシージャ自体は、ラップして、キーを使用する前に変換する方法を隠す必要があります。

この方法のメリットは次のとおりです。

- 表への直接アクセス権を持つユーザーが、暗号化されていない機密性の高いデータを参照したり、キーを取得してデータを復号化することができません。
- 復号化されたデータへのアクセスは、（暗号化されている）データを選択し、キー表から復号化キーを取得し、データの復号化に使用される前に復号化キーを変換するプロシージャを介して制御されます。
- プロシージャをラップすることによって、傍受者からデータ変換アルゴリズムを隠し、プロシージャ・コードをわかりにくくします。
- データ表とキー表の両方への SELECT 権限を取得していても、キーが使用前に変換されるため、この権限を持つユーザーがデータを復号化できる保証はありません。

この方法のデメリットは、キー表とデータ表の両方への SELECT 権限を取得しており、キー変換アルゴリズムを導出できるユーザーが、暗号化スキームを解読できるということです。

前述の方法は完全ではありませんが、クリアテキストで格納されている機密性の高い情報（クレジット・カード番号など）を簡単に取得されないように保護するには十分です。

オペレーティング・システムへのキーの格納

別のオプションは、オペレーティング・システム（フラット・ファイル）にキーを格納することです。Oracle9*i* では、暗号化キーを取得するために使用できる PL/SQL からコールアウトできます。ただし、オペレーティング・システムにキーを格納し、コールアウトする場合、データの安全性はオペレーティング・システム上で保護する場合と同じです。データベースに格納されたデータを暗号化するまでのセキュリティ上の主な理由が、オペレーティング・システムからデータベースに侵入される可能性があるということの場合は、オペレーティング・システムにキーを格納すると、データベース自体にキーを格納するより、ハッカーは簡単に暗号化データを取得することができます。

ユーザーによるキーの管理

ユーザーにキーを提供させると、ユーザーはキーに対して責任を負います。ヘルプ・デスクへの問合せの 40% が、パスワードを忘れたユーザーからのものであるということを考慮すると、ユーザーが暗号化キーを管理するのは危険であることがわかります。ほとんどの場合、ユーザーが暗号化キーを忘れるか、キーを書き留めておくため、セキュリティ上のデメリットになります。ユーザーが暗号化キーを忘れたり、退職した場合、データはリカバリ不可能になります。

ユーザーにキーを提供または管理させる場合は、ネットワーク暗号化を使用して、クライアントからサーバーにクリアテキストでキーが渡されないようにする必要があります。また、困難なセキュリティ問題であるキー・アーカイブ・メカニズムを開発する必要があります。キー・アーカイブまたはバックドアではセキュリティ上のデメリットとなるため、暗号化が問題解決策の第 1 候補になります。

暗号化キーの変更

慎重にセキュリティを実行するには、定期的に暗号化キーを変更します。格納データについては、別の適切なキーで、定期的にデータを復号化し、再暗号化する必要があります。これは、データがアクセスされていない間に行う必要があるため、別の問題が発生します。特に、クレジット・カード番号を暗号化する Web ベースのアプリケーションに該当します。暗号化キーを切り替える間、アプリケーション全体を停止できないためです。

バイナリ・ラージ・オブジェクト（BLOB）

特定のデータ型では、暗号化により多くの作業が必要です。たとえば、Oracle では、BLOB の格納がサポートされています。これによって、非常に大きなオブジェクト（GB など）をデータベースに格納できます。BLOB は、列として内部的に格納するか、外部ファイルに格納することができます。DBMS_OBFUSCATION_TOOLKIT を使用するには、データを 32,767 文字列チャンク（PL/SQL での最大サイズ）に分割し、チャンクを暗号化して、BLOB に追加する必要があります。復号化するには、同じ手順を逆の順序で行う必要があります。

データ暗号化 PL/SQL プログラムの例

次に、PL/SQL プログラムを使用したデータの暗号化の例を示します。これは、テスト文字列データの暗号化および復号化を示します。RAW データを暗号化するインタフェースと類似しています。

```
DECLARE
 input_string        VARCHAR2(16) := 'tigertigertigert';
 key_string          VARCHAR2(8)  := 'scottsco';

 encrypted_string            VARCHAR2(2048);
 decrypted_string            VARCHAR2(2048);
   error_in_input_buffer_length EXCEPTION;
   PRAGMA EXCEPTION_INIT(error_in_input_buffer_length, -28232);
   INPUT_BUFFER_LENGTH_ERR_MSG VARCHAR2(100) :=
    '*** DES INPUT BUFFER NOT A MULTIPLE OF 8 BYTES ***';

BEGIN
   dbms_output.put_line('> ========= BEGIN TEST =========');
   dbms_output.put_line('> Input String                          : ' ||
 input_string);
   BEGIN
      dbms_obfuscation_toolkit. input_string => input_string,
               key_string => key_string, encrypted_string => encrypted_string );
      dbms_output.put_line('> encrypted string              : ' ||
encrypted_string);
      dbms_obfuscation_toolkit.DESDecrypt(input_string => encrypted_string,
               key => raw_key, decrypted_string => decrypted_string);
```

```
   dbms_output.put_line('> Decrypted output             : ' ||
                decrypted_string);
   dbms_output.put_line('>  ');
   if input_string =
                decrypted_string THEN
      dbms_output.put_line('> DES Encryption and Decryption successful');
   END if;
EXCEPTION
   WHEN error_in_input_buffer_length THEN
         dbms_output.put_line('> ' || INPUT_BUFFER_LENGTH_ERR_MSG);
END;
```

参照：『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

第 IV 部

アクティブ・データベース

データベース・サーバーの信頼性およびパフォーマンスを活用し、データベース内のすべてのアプリケーションでプログラム・ロジックを再利用するには、いくつかのプログラム・ロジックをデータベース自体に移動します。これによって、別のアプリケーションがなくても、データベースは操作をクリーンアップし、イベントに応答できます。

第 IV 部に含まれる章は、次のとおりです。

15

トリガーの使用

トリガーとは、データベース内に格納され、何かが発生したときに暗黙的に実行つまり**起動**されるプロシージャです。

これまでトリガーは、表またはビューに対して INSERT、UPDATE または DELETE が発生したときの PL/SQL ブロックの実行をサポートしてきました。Oracle8*i* からは、トリガーは、DATABASE および SCHEMA に対するシステム・イベントおよびその他のデータ・イベントもサポートしています。Oracle では、PL/SQL プロシージャまたは Java プロシージャの実行もサポートします。

この章では、DML トリガー、INSTEAD OF トリガーおよびシステム・トリガー（DATABASE および SCHEMA に対するトリガー）について説明します。内容は次のとおりです。

トリガーの設計

トリガーを設計するときは、次のガイドラインを使用してください。

- トリガーは、ある操作が実行されたときに、関係するアクションが確実に実行されるようにする場合に使用してください。
- Oracle にすでに組み込まれている機能は、トリガーに重複定義しないでください。たとえば、宣言整合性制約で同じチェックができる場合、不良データを拒否するためにトリガーを定義しないでください。
- トリガーのサイズを制限してください。トリガーのロジックが 60 行をはるかに超える PL/SQL コードを必要とする場合は、コードの大部分をストアド・プロシージャに組み込んで、トリガーからそのプロシージャをコールすることをお薦めします。
- ユーザーまたはデータベース・アプリケーションのどちらがトリガーを実行する文を発行するかに関係なく、トリガーを実行する文に対して起動される集中的グローバル操作にのみトリガーを使用してください。
- **再帰トリガーは作成しないでください。**たとえば、Emp_tab 表に対して UPDATE 文を発行する AFTER UPDATE 文トリガーを Emp_tab 表に作成すると、このトリガーはメモリー不足になるまで再帰的に起動し続けます。
- DATABASE に対するトリガーは、慎重に使用してください。このようなトリガーは、トリガーの対象イベントが発生するたびに、すべてのユーザーに対して実行されます。

トリガーの作成

トリガーは、CREATE TRIGGER 文を使用して作成します。この文は、SQL*Plus や Enterprise Manager などの対話型ツールで使用できます。対話型ツールを使用する場合、CREATE TRIGGER 文をアクティブにするには、最終行にスラッシュ（/）を 1 つ付けます。

次の文は、Emp_tab 表に対するトリガーを作成します。

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
WHEN (new.Empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :new.sal  - :old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put('  New salary: ' || :new.sal);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
/
```

DML 操作（INSERT 文、UPDATE 文および DELETE 文）が表に実行されると、トリガーが起動されます。トリガーを起動する操作の組合せは選択できます。

トリガーは、BEFORE キーワードを使用するため、表に入る前に新しい値にアクセスできます。また、:NEW.*column_name* に割り当てることによって、簡単に修正されたエラーがある場合は、その値を変更できます。トリガーは、初期変更が適用され、表が一貫性のある状態に戻った後にのみ同じ表の問合せまたは変更を実行できるため、トリガーでこれらの操作をする場合は、AFTER キーワードを使用します。

トリガーは、FOR EACH ROW 句を使用するため、複数行の更新または削除時などに複数回実行される場合があります。操作が発生したという事実のみを記録し、各行のデータを調べない場合は、この句を省略します。

トリガーの作成後に、次の SQL 文を入力します。

```
UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;
```

これによって、トリガーは更新される行ごとに 1 回起動され、いずれの場合も、新旧の給与およびその差額を出力します。

PL/SQL ブロックにエラーがあると、CREATE 文（または CREATE OR REPLACE 文）は正常に実行されません。

注意： トリガーのサイズは、32KB 未満で指定してください。

次の項では、トリガーの各要素の指定方法を説明します。

> **参照：** CREATE TRIGGER 文の例の詳細は、15-29 ページの「トリガー・アプリケーションの例」を参照してください。

トリガーの種類

トリガーは、ストアド PL/SQL ブロックか、あるいは表、ビュー、スキーマまたはデータベース自体に対応付けられる PL/SQL、C または Java のプロシージャです。Oracle では、指定されたイベントが発生したときに、トリガーを自動的に実行します。イベントはシステム・イベントか、または表に対して発行される DML 文の形態をとります。

トリガーは次のいずれかです。

- 表に対する DML トリガー
- ビューに対する INSTEAD OF トリガー
- DATABASE または SCHEMA に対するシステム・トリガー：DATABASE の場合は、トリガーはイベントごとにすべてのユーザーに対して起動されます。SCHEMA の場合は、トリガーはイベントごとに特定ユーザーに対して起動されます。

参照： トリガーを作成する構文の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

システム・イベントの概要

次のいずれかに対して起動されるトリガーを作成できます。

- DML 文（DELETE、INSERT、UPDATE）
- DDL 文（CREATE、ALTER、DROP）
- データベース処理（SERVERERROR、LOGON、LOGOFF、STARTUP、SHUTDOWN）

システム・イベント属性の取得

トリガーが起動されるときに、イベント固有の特定の属性を取得できます。

参照： イベント属性の取得のためにコールする関数の詳細なリストは、第 16 章「システム・イベントの処理」を参照してください。

DATABASE に対してトリガーを作成するということは、トリガーになるイベントがユーザーの有効範囲の外にある（たとえば、データベースの STARTUP および SHUTDOWN）ことを意味し、そのトリガーはすべてのユーザーに適用されます（たとえば、LOGON イベントに対して DBA により作成されるトリガー）。

SCHEMA に対してトリガーを作成するということは、トリガーが現行のユーザーのスキーマ内に作成され、そのユーザーに対してのみ起動されることを意味します。

それぞれのトリガーに関して、DML およびシステム・イベントに対して発行を指定できます。

参照： 15-46 ページの「トリガーを介したシステム・イベントに対する応答」を参照してください。

トリガーのネーミング

トリガーの名前は、同一スキーマ内の他のトリガーに対して一意である必要があります。他のスキーマ・オブジェクト（表、ビュー、プロシージャなど）名とは重複してもかまいません。たとえば、表とトリガーに同じ名前を付けることもできます（ただし、間違えやすいため、違う名前を付けることをお薦めします）。

トリガーが起動するタイミング

トリガーは、トリガーを実行する文に基づいて起動されます。トリガーを実行する文では次のものを指定します。

- SQL 文あるいはトリガー本体を起動するシステム・イベント、データベース・イベントまたは DDL イベント。オプションとして、DELETE、INSERT および UPDATE がありま

す。これらのオプションのうちのいずれか、またはすべてをトリガーを実行する文の仕様部に組み込むことができます。

- トリガーに対応付けられる表、ビュー、DATABASE または SCHEMA。

> **注意：** トリガーを実行する文には、表またはビューを 1 つのみ指定できます。INSTEAD OF オプションを使用する場合、トリガーを実行する文にはビューのみを指定します。逆に、ビューがトリガーを実行する文に指定されている場合は、INSTEAD OF オプションのみを使用できます。

たとえば、PRINT_SALARY_CHANGES トリガーは、Emp_tab 表に対して DELETE、INSERT または UPDATE のいずれかが実行されたときに起動されます。次のいずれかの文によって、前述の例で使用されている PRINT_SALARY_CHANGES トリガーが実行されます。

```
DELETE FROM Emp_tab;
INSERT INTO Emp_tab VALUES ( ... );
INSERT INTO Emp_tab SELECT ... FROM ... ;
UPDATE Emp_tab SET ... ;
```

インポートおよび SQL*Loader によるトリガー起動

INSERT トリガーは、インポート中および SQL*Loader による通常のロード中に起動されます（ダイレクト・ロードの場合、トリガーはロードの前に使用禁止になります）。

IMP コマンドの IGNORE パラメータは、インポート中にトリガーを起動するかどうかを決定します。

- IGNORE=N（デフォルト）および表がすでに存在する場合は、インポートでは表の変更はされず、既存のトリガーは起動されません。
- 表が存在しない場合は、トリガーが定義される前にインポートによって表が作成されてロードされるため、この場合もトリガーは起動されません。
- IGNORE=Y の場合は、インポートによって行が既存の表にロードされます。既存のトリガーがすべて起動され、インポートされたデータを含めるために索引が更新されます。

列リストの UPDATE トリガーに対する影響

UPDATE 文が列のリストを含む場合があります。トリガーを実行する文に列リストが含まれる場合、トリガーは指定された列の 1 つが更新される場合にのみ起動されます。トリガーを実行する文で列リストが省略された場合、トリガーは対応付けられた表のいずれかの列が更新された場合に起動されます。INSERT または DELETE トリガーを実行する文に列リストを指定することはできません。

前述の PRINT_SALARY_CHANGES トリガーの例では、トリガーを実行する文に列リストを指定することができます。次に例を示します。

```
... BEFORE DELETE OR INSERT OR UPDATE OF ename ON Emp_tab ...
```

注意：

- INSTEAD OF トリガーを指定して UPDATE に列リストを指定できません。
- UPDATE OF 句に指定された列がオブジェクト列の場合は、オブジェクトの属性のどれかが変更された場合にもトリガーが起動されます。
- UPDATE OF 句はコレクション列には指定できません。

トリガー起動の制御（BEFORE オプションおよび AFTER オプション）

CREATE TRIGGER 文に BEFORE または AFTER オプションを指定して、実行中のトリガー文によってトリガー本体が起動されるタイミングを指定できます。CREATE TRIGGER 文では、トリガーを実行する文の直前に BEFORE または AFTER オプションを指定します。たとえば、前述の例では、PRINT_SALARY_CHANGES トリガーが BEFORE トリガーです。

注意： AFTER 行トリガーを使用すると、BEFORE 行トリガーより多少効率が上がります。BEFORE 行トリガーでは、影響を受けるデータ・ブロックをトリガーのために 1 回読み込み（論理的な読込みであり、物理的な読込みではない）、トリガーを実行する文のために再び読み込む必要があります。

AFTER 行トリガーでは、データ・ブロックを、トリガーを実行する文およびトリガーの両方に対して 1 回読み込むのみで済みます。

複合ビューの変更（INSTEAD OF トリガー）

トリガー内では INSTEAD OF オプションも使用できます。INSTEAD OF トリガーを使用すると、UPDATE 文、INSERT 文および DELETE 文では直接変更できないビューを透過的に変更できます。このようなトリガーが INSTEAD OF トリガーと呼ばれる理由は、他の種類のトリガーと異なり、トリガーを実行する文を実行するかわりにトリガーが起動されるためです。トリガーは、対象となる操作を判断し、UPDATE、INSERT または DELETE 操作を基礎となる表に対して直接実行する必要があります。

INSTEAD OF トリガーを使用して、標準的な UPDATE 文、INSERT 文および DELETE 文をビューに対して書き込むと、正しいアクションが実行されるように INSTEAD OF トリガーがバックグラウンドで動作します。

INSTEAD OF トリガーをアクティブにできるのは、それぞれの行に対してのみです。

参照： 15-11 ページの「1 回または複数回のトリガーの起動（FOR EACH ROW オプション）」を参照してください。

注意：

- INSTEAD OF オプションが使用できるのは、ビューに対して作成されるトリガーのみです。
- BEFORE および AFTER オプションは、ビューに対して作成されるトリガーでは使用できません。
- ビューの CHECK オプションは、ビューに対する挿入または更新が INSTEAD OF トリガーを使用して行われる場合は施行されません。INSTEAD OF トリガー本体でチェックを施行する必要があります。

INSTEAD OF トリガーが必要なビュー

ビューの問合せに次のいずれかの構造体が含まれている場合、UPDATE 文、INSERT 文または DELETE 文を使用してビューを変更できません。

- 集合演算子
- グループ・ファンクション
- GROUP BY 句、CONNECT BY 句または START WITH 句
- DISTINCT 演算子
- 結合（結合ビューのサブセットは更新可能）

疑似列または式が含まれたビューを更新するには、疑似列または式のどちらも参照しない UPDATE 文のみを使用します。

INSTEAD OF トリガーの例

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Project_tab (
   Prj_level NUMBER,
   Projno    NUMBER,
   Resp_dept NUMBER);
CREATE TABLE Emp_tab (
   Empno     NUMBER NOT NULL,
   Ename     VARCHAR2(10),
   Job       VARCHAR2(9),
   Mgr       NUMBER(4),
   Hiredate  DATE,
   Sal       NUMBER(7,2),
   Comm      NUMBER(7,2),
   Deptno    NUMBER(2) NOT NULL);

CREATE TABLE Dept_tab (
   Deptno    NUMBER(2) NOT NULL,
   Dname     VARCHAR2(14),
   Loc       VARCHAR2(13),
   Mgr_no    NUMBER,
   Dept_type NUMBER);
```

次の例では、MANAGER_INFO ビューに行を挿入する INSTEAD OF トリガーを示します。

```
CREATE OR REPLACE VIEW manager_info AS
    SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level,
           p.projno
        FROM   Emp_tab e, Dept_tab d, Project_tab p
        WHERE  e.empno =  d.mgr_no
        AND    d.deptno = p.resp_dept;

CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n                -- new manager information

FOR EACH ROW
DECLARE
   rowcnt number;
BEGIN
   SELECT COUNT(*) INTO rowcnt FROM Emp_tab WHERE empno = :n.empno;
   IF rowcnt = 0  THEN
       INSERT INTO Emp_tab (empno,ename) VALUES (:n.empno, :n.ename);
   ELSE
```

```
      UPDATE Emp_tab SET Emp_tab.ename = :n.ename
         WHERE Emp_tab.empno = :n.empno;
   END IF;
   SELECT COUNT(*) INTO rowcnt FROM Dept_tab WHERE deptno = :n.deptno;
   IF rowcnt = 0 THEN
      INSERT INTO Dept_tab (deptno, dept_type)
         VALUES(:n.deptno, :n.dept_type);
   ELSE
      UPDATE Dept_tab SET Dept_tab.dept_type = :n.dept_type
         WHERE Dept_tab.deptno = :n.deptno;
   END IF;
   SELECT COUNT(*) INTO rowcnt FROM Project_tab
      WHERE Project_tab.projno = :n.projno;
   IF rowcnt = 0 THEN
      INSERT INTO Project_tab (projno, prj_level)
         VALUES(:n.projno, :n.prj_level);
   ELSE
      UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
         WHERE Project_tab.projno = :n.projno;
   END IF;
END;
```

MANAGER_INFO ビューに行を挿入するというアクションでは、まず、MANAGER_INFO の導出元の実表に該当する行があるかどうかを調べます。その後、必要に応じて、新しい行を挿入するか、または既存の行を更新します。同じようなトリガーを使用して、UPDATE および DELETE 用のアクションを指定できます。

オブジェクト・ビューおよび INSTEAD OF トリガー

INSTEAD OF トリガーによって、クライアント側で OCI コールを介してオブジェクト・ビューのインスタンスを変更できます。

> **参照：**『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

クライアント側のオブジェクト・キャッシュ内でオブジェクト・ビューによって具体化されたオブジェクトを変更し、永続記憶域にそれをフラッシュするには、オブジェクト・ビューが変更可能なものでないかぎり、INSTEAD OF トリガーを指定する必要があります。ただし、オブジェクトが読込み専用の場合は、トリガーを定義して確保する必要はありません。

ネストした表のビューの列に対するトリガー

INSTEAD OF トリガーは、ネストした表のビューの列に対しても作成できます。このトリガーは、ネストした表の要素を更新する手段を提供します。このトリガーは、ネストした表の各更新対象要素に対して起動されます。トリガー内の行相関変数が、ネストした表の要素

に対応します。この種のトリガーは、変更対象のネストした表を含む親である行にアクセスするための追加相関名も提供します。

注意： このトリガーの特長は次のとおりです。

- ビューの中のネストした表の列に対してのみ定義できます。
- ネストした表の要素が、THE() 句または TABLE() 句を使用して変更されるときにのみ起動されます。ビューに対して DML 文が実行されるときには起動されません。

たとえば、社員用のネストした表を含む部門ビューについて考えてみます。

```
CREATE OR REPLACE VIEW Dept_view AS
SELECT d.Deptno, d.Dept_type, d.Dept_name,
   CAST (MULTISET ( SELECT e.Empno, e.Empname, e.Salary
      FROM Emp_tab e
      WHERE e.Deptno = d.Deptno) AS Amp_list_ Emplist
FROM Dept_tab d;
```

CAST(MULTISET..) 演算子によって、部門ごとに社員の多重集合が作成されます。ここで、社員のネストした表である emplist 列を変更する場合は、この列に対して INSTEAD OF トリガーを定義して処理できます。

次の例は、挿入トリガーの作成方法を示します。

```
CREATE OR REPLACE TRIGGER Dept_emplist_tr
   INSTEAD OF INSERT ON NESTED TABLE Emplist OF Dept_view
   REFERENCING NEW AS Employee
      PARENT AS Department
   FOR EACH ROW
BEGIN
-- The insert on the nested table is translated to an insert on the base table:
   INSERT INTO Emp_tab VALUES (
      :Employee.Empno, :Employee.Empname,:Employee.Salary, :Department.Deptno);
END;
```

ネストした表に INSERT が実行されるとトリガーが起動され、Emp_tab 表が正しい値で埋められます。次に例を示します。

```
INSERT INTO TABLE (SELECT d.Emplist FROM Dept_view d WHERE Deptno = 10)
   VALUES (1001, 'John Glenn', 10000)
```

この例の :department.deptno 相関変数には、値 10 が入ります。

1 回または複数回のトリガーの起動（FOR EACH ROW オプション）

FOR EACH ROW オプションによって、トリガーが行トリガーになるか文トリガーになるかが決定されます。FOR EACH ROW を指定すると、トリガーを実行する文によって影響を受ける表の各行に対してトリガーが 1 回起動されます。FOR EACH ROW オプションを指定しないと、トリガーは該当する個々の文に対して 1 回のみ起動され、その文によって影響される各行に対して別々に起動されることはありません。

たとえば、次のようなトリガーを定義します。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Emp_log (
   Emp_id     NUMBER,
   Log_date   DATE,
   New_salary NUMBER,
   Action     VARCHAR2(20));
```

```
CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON Emp_tab
FOR EACH ROW
WHEN (new.Sal > 1000)
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
       VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;
```

次に、次の SQL 文を入力します。

```
UPDATE Emp_tab SET Sal = Sal + 1000.0
    WHERE Deptno = 20;
```

部門 20 に 5 人の社員がいる場合、この文が入力されるとトリガーが 5 回起動されます。これは、5 つの行が影響を受けるためです。

次のトリガーは、Emp_tab 表の各 UPDATE に対して 1 回のみ起動します。

```
CREATE OR REPLACE TRIGGER Log_emp_update
AFTER UPDATE ON Emp_tab
BEGIN
    INSERT INTO Emp_log (Log_date, Action)
       VALUES (SYSDATE, 'Emp_tab COMMISSIONS CHANGED');
END;
```

参照： トリガーの起動順序の詳細は、『Oracle9*i* データベース概要』を参照してください。

文レベル・トリガーは、文全体の妥当性チェックを実行するときに有効です。

条件に基づいたトリガーの起動（WHEN 句）

行トリガー定義にトリガー制約をオプションで指定できます。これには、WHEN 句に SQL のブール式を指定します。

注意： WHEN 句は、文トリガーの定義に含めることはできません。

このオプションを指定すると、WHEN 句の式がトリガーの処理対象となる行ごとに評価されます。

行に対して式が TRUE と評価されると、その行のかわりにトリガー本体が起動されます。ただし、行に対して式が FALSE または NOT TRUE と評価された場合（NULL の場合のように不明な場合）、その行に対してトリガー本体は起動されません。WHEN 句の評価は、トリガーを実行する SQL 文の実行には影響しません（WHEN 句の式が FALSE と評価されても、トリガーを実行する文はロールバックされません）。

たとえば、PRINT_SALARY_CHANGES トリガーでは、Empno の新しい値が 0（ゼロ）、NULL または負の場合、トリガー本体は実行されません。より具体的な例としては、ある列の値が他の列の値より小さいかどうかテストする場合があります。

行トリガーの WHEN 句の式に相関名を指定できます。相関名については次に説明します。WHEN 句の式は SQL 式にする必要があり、副問合せを含むことはできません。WHEN 句では、PL/SQL 式（ユーザー定義ファンクションを含む）は使用できません。

注意： WHEN 句は INSTEAD OF トリガーには指定できません。

トリガー本体のコーディング

トリガー本体は、SQL 文または PL/SQL 文を含めることができる CALL プロシージャまたは PL/SQL ブロックです。CALL プロシージャは、PL/SQL または PL/SQL ラッパーにカプセル化された Java プロシージャのどちらかです。これらの文は、トリガーを実行する文が入力され、トリガー制約（含まれている場合）が TRUE と評価された場合に実行されます。

行トリガーのトリガー本体には、相関名、REFERENCEING オプション、条件述語の INSERTING、DELETING、UPDATING などの特殊な要素を含められます。これらの要素は、PL/SQL ブロックのコードにも含めることができます。

注意： INSERTING、DELETING、UPDATING 条件述語は、CALL プロシージャには使用できません。使用できるのは PL/SQL ブロック内のみです。

例 : トリガーを使用したログインの監視

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
GRANT ADMINISTER DATABASE TRIGGER TO scott;
CONNECT scott/tiger
CREATE TABLE audit_table (
   seq number,
   user_at  VARCHAR2(10),
   time_now DATE,
   term     VARCHAR2(10),
   job      VARCHAR2(10),
   proc     VARCHAR2(10),
   enum     NUMBER);
```

```
CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
   BEGIN
      INSERT INTO Audit_table (user_at) VALUES(c);
   END;

CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
-- Just call an existing procedure. The ORA_LOGIN_USER is a function
-- that returns information about the event that fired the trigger.
CALL foo (ora_login_user)
/
```

例 : トリガーからの Java プロシージャのコール

トリガーは、PL/SQL で宣言されますが、Java などの他の言語でプロシージャをコールすることができます。

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:old.Id, :old.Ename)
```

対応する Java ファイルは thjvTriggers.java です。

```
import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers
{
public state void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
   {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stmt = conn.CreateStatement();
   String sql = "insert into logtab values
   ("+ old_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
   stmt.executeUpdate (sql);
   stmt.close();
   return;
   }
}
```

行トリガーでの列値のアクセス

行トリガーのトリガー本体では、PL/SQL コードおよび SQL 文は、トリガーを実行する文によって影響を受ける現在の行に含まれる old 列値および new 列値にアクセスできます。変更される表の各列に 2 つの相関名（old 列値用および new 列値用に 1 つずつ）があります。トリガーを実行する文の種類によっては、相関名が意味を持たない場合もあります。

- INSERT 文によって起動されるトリガーは、new 列値に対してのみ意味のあるアクセスを行います。行は INSERT によって作成されるため、old 値は NULL です。
- UPDATE 文によって起動されるトリガーは、BEFORE および AFTER の両方の行トリガーで、old 列値および new 列値の両方にアクセスします。
- DELETE 文によって起動されるトリガーは、:old 列値に対してのみ意味のあるアクセスを行います。行を削除すると行はなくなるため、:new 値は NULL です。ただし、:new 値は変更できません。:new 値を変更しようとすると、ORA-04084 が発生します。

元の列値は、列名の前に old 修飾子を指定して参照し、新しい列値は列名の前に new 修飾子を指定して参照します。たとえば、トリガーを実行する文が Emp_tab 表（列 SAL、COMM などを持つ）に対応付けられている場合、トリガー本体に文を含めることができます。次に例を示します。

```
IF :new.Sal > 10000 ...
IF :new.Sal < :old.Sal ...
```

old 値および new 値は、BEFORE および AFTER 行トリガー内で使用できます。new 列値は BEFORE 行トリガー内に割り当てることができますが、（AFTER 行トリガーが起動される前にトリガーを実行する文が有効となるため）AFTER 行トリガーに new 列値を割り当てることはできません。BEFORE 行トリガーによって new.column の値が変更されると、同じ文によって起動される AFTER 行トリガーは、BEFORE 行トリガーによって割り当てられた変更を参照します。

WHEN 句のブール式には相関名を使用することもできます。old および new 修飾子をトリガー本体で使用する場合は、修飾子の前にコロン（:）を付ける必要があります。ただし、修飾子を WHEN 句または REFERENCING オプションで使用する場合、コロンは使用できません。

例：トリガーによる LOB 列の変更

以前は、トリガー本体内の LOB 列の検査はできましたが、変更はできませんでした。現在は、通常の SQL および CLOB 列を持つ PL/SQL ファンクションを使用して、他の列と同様に処理できます。また、BLOB 列を持つ DBMS_LOB パッケージをコールできます。

```
drop table tab1;

create table tab1 (c1 clob);
insert into tab1 values ('<h1>HTML Document Fragment</h1><p>Some text.');

create or replace trigger trg1
  before update on tab1
  for each row
begin
  dbms_output.put_line('Old value of CLOB column: '||:OLD.c1);
  dbms_output.put_line('Proposed new value of CLOB column: '||:NEW.c1);

-- Previously, we couldn't change the new value for a LOB.
-- Now, we can replace it, or construct a new value using SUBSTR, INSTR, etc.
-- operations for a CLOB, or DBMS_LOB calls for a BLOB.
  :NEW.c1 := :NEW.c1 || to_clob('<hr><p>Standard footer paragraph.');

  dbms_output.put_line('Final value of CLOB column: '||:NEW.c1);
end;
/

set serveroutput on;
update tab1 set c1 = '<h1>Different Document Fragment</h1><p>Different text.';

select * from tab1;
```

ネストした表のビューの列に対する INSTEAD OF トリガー

ネストした表のビューの列に対する INSTEAD OF トリガーの場合は、new および old 修飾子が、ネストした表の新しい要素および古い要素に対応します。このネストした表の要素に対応する親である行は、parent 修飾子を使用してアクセスできます。親相関名は、ネストした表のトリガー内でのみ意味があり有効です。

トリガーとの名前の競合の回避（REFERENCING オプション）

行トリガーのトリガー本体に REFERENCING オプションを指定して、old または new とネーミングされる相関名または表の重複を避けることができます。ただし、このようなことはほとんど発生しないため、このオプションはほとんど使用されません。

たとえば、field1（数値）および field2（文字）の列を含む new 表があるとします。次の CREATE TRIGGER の例は、相関名を指定できる表 new に対応付けられるトリガーの例です。相関名と表名の重複が避けられています。

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE new (
   field1     NUMBER,
   field2     VARCHAR2(20));
```

```
CREATE OR REPLACE TRIGGER Print_salary_changes
BEFORE UPDATE ON new
REFERENCING new AS Newest
FOR EACH ROW
BEGIN
   :Newest.Field2 := TO_CHAR (:newest.field1);
END;
```

REFERENCING オプションを使用して new 修飾子を newest に名前を変更し、その後でトリガー本体に使用していることに注意してください。

トリガー（INSERTING、UPDATING および DELETING 述語）を起動する DML 操作の検出

種類が異なる複数の DML 操作でトリガーを起動する場合（たとえば、ON INSERT OR DELETE OR UPDATE OF Emp_tab）は、トリガー本体で条件述語の INSERTING、DELETING および UPDATING を使用して、トリガーを起動する文の種類の確認ができます。

トリガー本体のコード内で、トリガーを起動する DML 操作の種類に応じて、次のコード・ブロックを実行できます。

```
IF INSERTING THEN ... END IF;
IF UPDATING THEN ... END IF;
```

最初の条件は、トリガーを起動した文が INSERT 文の場合にのみ TRUE と評価されます。2 番目の条件は、トリガーを起動した文が UPDATE 文の場合にのみ TRUE と評価されます。

UPDATE トリガーでは、UPDATING 条件述語に列名を指定して、指定した列が更新されているかどうかを判断できます。たとえば、トリガーが次のように定義されているとします。

```
CREATE OR REPLACE TRIGGER ...
... UPDATE OF Sal, Comm ON Emp_tab ...
BEGIN

... IF UPDATING ('SAL') THEN ... END IF;

END;
```

THEN 句のコードは、トリガー UPDATE 文が SAL 列を更新する場合にのみ実行されます。このように、対象の列が変更されていない場合は、トリガーはオーバーヘッドを最小化できます。

トリガー本体内のエラー条件および例外

トリガー本体の実行中に、事前定義またはユーザー定義のエラー条件または例外が発生すると、トリガーを実行する文のみでなくトリガー本体のすべての影響が（エラーが例外ハンドラによって検出された場合を除き）ロールバックされます。したがって、トリガー本体は例外を発生させることによって、トリガーを実行する文を実行しないで済みます。ユーザー定義例外は、複雑なセキュリティ認可または整合性制約を施行するトリガーによく使用されます。

これに対する唯一の例外は、対象イベントがデータベースの STARTUP、SHUTDOWN、またはログインしているユーザーが SYSTEM のときの LOGIN の場合です。このような場合は、トリガー・アクションのみがロールバックされます。

トリガーおよびリモート例外処理

リモート・サイトにアクセスするトリガーは、ネットワーク・リンクが使用できない場合はリモート例外処理を実行できません。次に例を示します。

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
  INSERT INTO Emp_tab@Remote     -- <- compilation fails here
  VALUES ('x');                  --    when dblink is inaccessible
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO Emp_log
    VALUES ('x');
END;
```

トリガーは作成されたときにコンパイルされます。したがって、トリガーをコンパイルする必要があるときにリモート・サイトを使用できないと、Oracle はリモート・データベースにアクセスする文の妥当性チェックができず、コンパイルは正常に実行されません。前述の例外文の例は、トリガーがコンパイルを完了しないため実行できません。

ストアド・プロシージャはコンパイル済の形式で格納されるので、前述の例の解決策は次のとおりです。

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
   Insert_row_proc;
END;

CREATE OR REPLACE PROCEDURE Insert_row_proc AS
BEGIN
    INSERT INTO Emp_tab@Remote
    VALUES ('x');
EXCEPTION
   WHEN OTHERS THEN
       INSERT INTO Emp_log
       VALUES ('x');
END;
```

この例のトリガーは正常にコンパイルし、ストアド・プロシージャをコールします。このストアド・プロシージャは、リモート・データベースにアクセスするための妥当性チェック済の文をすでに持っています。したがって、リンクが使用できないためにリモート INSERT 文が失敗すると、例外が捕捉されます。

トリガー作成の制限

トリガーのコーディングには、標準 PL/SQL ブロックにはない、いくつかの制約があります。次の項では、トリガーのこのような制約を説明します。

トリガーの最大サイズ

トリガーのサイズは、32KB 以下に指定する必要があります。

トリガー本体で使用可能な SQL 文

トリガー本体には、DML SQL 文を含めることができます。また、SELECT 文を含めることはできますが、SELECT... INTO... 文またはカーソル定義中の SELECT 文を指定する必要があります。

DDL 文はトリガー本体には含めることはできません。また、トランザクション制御文もトリガーには含めることはできません。ROLLBACK、COMMIT および SAVEPOINT は使用でき

ません。システム・トリガーの場合は、{CREATE/ALTER/DROP} TABLE 文および ALTER...COMPILE を使用できます。

> **注意：** トリガーによってコールされるプロシージャは、トリガー本体のコンテキスト内で実行されるため、このようなプロシージャが前述のトランザクション制御文を実行することはできません。

トリガー内の文では、リモート・スキーマ・オブジェクトを参照できます。ただし、ローカル・トリガー内からリモート・プロシージャをコールするときは、特に注意が必要です。トリガーの実行中にタイムスタンプまたはシグネチャの不一致が見つかると、リモート・プロシージャは実行されず、トリガーが無効になります。

LONG および LONG RAW データ型のトリガーの制限

トリガー内の LONG および LONG RAW データ型には、次の制限があります。

- トリガー内の SQL 文で、LONG または LONG RAW データ型の列にデータを挿入できます。
- LONG または LONG RAW 列のデータが制約データ型（CHAR や VARCHAR2 など）に変換できる場合、トリガー内の SQL 文が LONG または LONG RAW 列を参照できます。これらのデータ型の最大長は 32000 バイトです。
- LONG または LONG RAW データ型を指定して変数を宣言することはできません。
- LONG または LONG RAW 列では、:NEW および :PARENT は使用できません。

複数回起動する BEFORE トリガー

UPDATE 文または DELETE 文が同時実行中の UPDATE との競合を検出すると、Oracle は SAVEPOINT までの透過的 ROLLBACK を実行して、更新を再起動します。文が正常に完了するまで、これは何度も行われる可能性があります。文が再起動されるたびに、BEFORE 文トリガーが再起動されます。セーブポイントまでのロールバックでは、トリガー内で参照されるパッケージ変数への変更は取り消されません。パッケージには、このような状況を検出するためのカウンタ変数を含める必要があります。

トリガーに対する行の評価順序

リレーショナル・データベースは、SQL 文による行の処理順序を保証しません。したがって、行の処理順序に基づくトリガーは作成しないでください。たとえば、グローバル変数の現在の値が、行トリガーによって処理される行に依存する場合は、行トリガー内のグローバル・パッケージ変数に値を割り当てないでください。また、グローバル・パッケージ変数の値がトリガー内で更新される場合は、これらの変数を BEFORE 文トリガー内で初期化するようにしてください。

トリガー本体内の文によって他のトリガーが起動される場合、それらのトリガーはカスケードしているといいます。Oracle では、一時点に最大 32 個のトリガーをカスケードできます。なお、OPEN_CURSORS 初期化パラメータを使用して、カスケード可能なトリガーの数を制

限することもできます。これは、トリガーを実行するたびにカーソルがオープンされるためです。

トリガーの評価順序

トリガーは、インラインで、またはプロシージャをコールすることによって一連の操作を実行できますが、同じ型の複数のトリガーを使用すると、同じ表に対するトリガーを持つアプリケーションのインストールをモジュール化できるため、データベース管理が強化されます。

Oracle は、別の型のトリガーを実行する前に、同じ型のすべてのトリガーを実行します。1 つの表に対して同じ型のトリガーが複数ある場合、Oracle では任意の順序を選択してこれらのトリガーを実行します。

> **参照：** トリガーの起動順序の詳細は、『Oracle9*i* データベース概要』を参照してください。

後続の各トリガーは、前に起動されたトリガーが変更した内容を参照します。個々のトリガーは、old 値および new 値を参照できます。old 値は元の値で、new 値は一番最後に起動された UPDATE トリガーまたは INSERT トリガーが設定した現在の値です。

トリガーされた複数のアクションが特定の順序で確実に実行されるようにするには、これらのアクションをまとめて 1 つのトリガーに統合する必要があります（たとえば、トリガーが一連のプロシージャをコールする方法を使用します）。

Oracle7 リリース 7.1 以下を使用している場合は、同じ種類の複数のトリガーを含むデータベースはオープンできません。また、COMPATIBLE 初期化パラメータがリリース 7.1.0 以下に設定されている場合も、データベースはオープンできません。システム・トリガーの場合は、互換性は 8.1.0 である必要があります。

変更表のトリガー制限

変更表とは、UPDATE 文、DELETE 文、INSERT 文で現在修正されている表、または DELETE CASCADE 制約の影響によって更新する必要がある表のことです。

トリガーを実行する文を発行したセッションは、変更表の問合せや変更をできません。この制限によって、トリガーは一貫性のないデータは参照しません。

この制限は、FOR EACH ROW 句を使用するすべてのトリガー、および DELETE CASCADE の結果として起動される文トリガーに適用されます。INSTEAD OF トリガー内で変更されたビューは、変更ビューとみなされません。

変更表でトリガーが起動されると、ランタイム・エラーが発生し、トリガー本体の処理結果およびトリガーを実行する文がロールバックされ、ユーザーまたはアプリケーションに制御が戻ります。

次のトリガーについて検討してみます。

```
CREATE OR REPLACE TRIGGER Emp_count
```

```
AFTER DELETE ON Emp_tab
FOR EACH ROW
DECLARE
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n FROM Emp_tab;
    DBMS_OUTPUT.PUT_LINE(' There are now ' || n ||
        ' employees.');
END;
```

次の SQL 文が入力されるとします。

```
DELETE FROM Emp_tab WHERE Empno = 7499;
```

行が削除されるときに表が変更されるため、次のエラーを戻します。

```
ORA-04091: 表 SCOTT.Emp_tab は変更しています。トリガー / 関数は見ることができません
```

トリガーから FOR EACH ROW 行を削除すると、このトリガーは文トリガーになり、制限やトリガーの対象にはなりません。

変更表を更新する必要がある場合、一時表、PL/SQL 表またはパッケージ変数を使用してこれらの制限を回避することもできます。たとえば、元の表を更新する 1 つの AFTER 行トリガーが変更表エラーとなった場合、かわりに、一時表を更新する AFTER 行トリガーおよび一時表からの値を使用して元の表を更新する AFTER 文トリガーの 2 つのトリガーを使用できる場合があります。

宣言整合性制約は、行トリガーに関して随時テストされます。

> **参照：** トリガー間の相互作用と整合性制約の詳細は、『Oracle9*i* データベース概要』を参照してください。

分散データベースの異なるノードの表の間では、現在、宣言参照整合性制約はサポートされていないため、変更表の制限は、リモート・ノードにアクセスするトリガーには適用されません。これらの制限は、ループバック・データベース・リンクで接続されている同一データベース内の表の間でも施行されません。ループバック・データベース・リンクでは、リンクを含むデータベースに戻る Oracle Net パスを定義して、ローカル表がリモートで表示されます。

トリガー制限を迂回するために、ループバック・データベース・リンクは使用できません。このようなアプリケーションは、予測不可能な動作をする場合があります。

変更表の制約の緩和

Oracle8*i* より前では、親文が暗黙的に表を読み込んで外部キー制約を施行する場合、行トリガーが表を変更できないようにする制約エラーが存在しました。Oracle8*i* 以上では、制約エラーは存在しません。さらに、外部キーのチェックは、少なくとも親文の終わりまで遅延されます。

ただし、変更エラーは存在するため、親文が変更する表をトリガーが読込みまたは変更を行うことはできません。ただし、Oracle8*i* 以上では、親表に対して削除を行うと、BEFORE/AFTER 文トリガーが 1 回起動されます。これによって、(行トリガー以外の) トリガーを作成して親表および子表の読込みおよび変更を行うことができます。

これによって、ほとんどの外部キー制約アクションはそれらの明白な AFTER 行トリガーを経由して実装されるため、制約は自己参照的ではなくなります。更新カスケード、更新セット NULL、更新セット・デフォルト、削除セット・デフォルト、欠落した親の挿入および子件数メンテナンスは、すべて簡単に実装できます。次に、更新カスケードの実装の例を示します。

```
create table p (p1 number constraint ppk primary key);
create table f (f1 number constraint ffk references p);
create trigger pt after update on p for each row begin
  update f set f1 = :new.p1 where f1 = :old.p1;
end;
/
```

この実装の場合、複数行を更新するときに注意が必要です。たとえば、表 p が値 (1)、(2)、(3) を持つ 3 つの行を持ち、表 f も値 (1)、(2)、(3) を持つ 3 つの行を持つとすると、次の文は p を正常に更新しますが、トリガーが f を更新するときに問題が発生します。

```
update p set p1 = p1+1;
```

まず、この文は p の値 (1) から (2) への更新を行い、トリガーは f の (1) から (2) への更新を行い、f に値 (2) の 2 つの行を残します。次に、文は p の値 (2) から (3) への更新を行い、トリガーは f の値 (2) から (3) への更新を行います。最後に、文は p の値 (3) から (4) への更新を行い、トリガーは f の 3 つの行すべてを (3) から (4) へ更新します。p と f のデータの関連は失われます。

この問題を回避するため、主キーを変更する p の複数行更新を禁止し、既存の主キー値を再利用する必要があります。また、どの外部キーがすでに更新されたかを追跡し、どの行も 2 回更新されないようにトリガーを変更することによっても解決できます。

これが、外部キーの更新に関するこの方法の唯一の問題です。トリガーは、別のトランザクションによってコミットされていない変更済の行を見逃すことはありません。これは、AFTER 行トリガーがコールされるまでは、いずれの一致する外部キー行もロックされないことを外部キー制約が保証するためです。

システム・トリガーの制限

イベントの違いによって、様々なイベント属性関数が使用できます。たとえば、ある種の DDL 操作を DDL イベントに対して使用できない場合があります。イベント属性関数は、エラー条件を作成するのではなく、定義しない場合があるため、イベント属性関数を使用する前に、16-2 ページの「イベント属性関数」を確認してください。

コミットされたトリガーのみが起動されます。たとえば、すべての CREATE イベントの後に起動されるトリガーを作成した場合、このトリガーはそのトリガー自身の作成後には起動さ

れません。これは、CREATE イベントが起動された時点では、このトリガーに関する正しい情報はまだコミットされていないためです。一方、すべての DROP イベントの前に起動されるトリガーを DROP した場合は、トリガーがこの DROP の前に起動されます。

たとえば、次の SQL 文を実行するとします。

```
CREATE OR REPLACE TRIGGER Foo AFTER CREATE ON DATABASE
BEGIN null;
END;
```

トリガー foo は、foo の作成後には起動されません。Oracle では、コミットされていないトリガーは起動されません。

外部関数のコールアウト

外部関数のコールアウトに関するすべての制限も適用されます。

トリガー・ユーザーとは

次の文では、トリガーには、トリガーの所有者の名前は戻されますが、表を更新しているユーザーの名前は戻されません。

```
SELECT Username FROM USER_USERS;
```

トリガーの使用に必要な権限

ご使用のスキーマに対してトリガーを作成するには、CREATE TRIGGER システム権限および次のいずれかが必要です。

- トリガーを起動する文で指定した表を所有していること
- トリガーを起動する文中の表に対する ALTER 権限を持っていること
- ALTER ANY TABLE システム権限を持っていること

他のユーザーのスキーマ内にトリガーを作成、または自スキーマ内のトリガーから他のスキーマ内の表を参照するには、CREATE ANY TRIGGER システム権限が必要です。この権限があると、任意のスキーマ内にトリガーを作成し、任意のユーザーの表と対応付けることができます。さらに、トリガーを作成するユーザーには、参照するプロシージャ、ファンクションまたはパッケージに対する EXECUTE 権限も必要です。

DATABASE に対してトリガーを作成するには、ADMINISTER DATABASE TRIGGER 権限が必要です。この権限が後になって取り消された場合、トリガーを削除することはできますが、変更することはできません。

トリガー本体で参照されるスキーマ・オブジェクトへのオブジェクト権限は、トリガーの所有者に（ロールを介さずに）明示的に付与する必要があります。トリガー本体の文は、トリガーを実行する文を発行するユーザーの権限ドメインではなく、そのトリガーの所有者の権

限ドメインから操作します。これは、ストアド・プロシージャの権限モデルと類似しています。

トリガーのコンパイル

トリガーは、無名 PL/SQL ブロックに :new および :old 機能を追加したものと類似していますが、コンパイル方法が異なります。無名 PL/SQL ブロックは、メモリーにロードされると、常に、コンパイルされます。コンパイルには、次の 3 段階が必要です。

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

これに対して、トリガーは、CREATE TRIGGER 文が入力されたときに完全にコンパイルされ、pcode はデータ・ディクショナリに格納されます。そのため、トリガーを起動するときに、共有カーソルをオープンしてトリガー・アクションを実行する必要はなくなります。そのかわり、トリガーは直接実行されます。

トリガーのコンパイル中にエラーが発生しても、トリガーは作成されます。ただし、DML 文がこのトリガーを起動すると、その文は失敗します（ランタイム・トリガー・エラーが発生すると、DML 文は必ず失敗します）。トリガーの作成時にすべてのコンパイル・エラーが表示されるように、SQL*Plus または Enterprise Manager 内で SHOW ERRORS コマンドを使用するか、または USER_ERRORS ビューからエラーを SELECT することができます。

トリガーの依存関係

コンパイル済のトリガーには依存関係があります。このようなトリガーは、トリガー本体からコールされるファンクションまたはストアド・プロシージャのような依存対象となるオブジェクトが修正されると無効になります。依存性の理由で無効になったトリガーは、次に起動された時点で再コンパイルされます。

ALL_DEPENDENCIES ビューを調べると、トリガーの依存関係がわかります。たとえば、次の文は、SCOTT スキーマ内のトリガーの依存関係を示します。

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
    FROM ALL_DEPENDENCIES
    WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

トリガーは他のファンクションまたはパッケージに依存することがあります。トリガー内に指定されているファンクションまたはパッケージが削除されると、トリガーは無効とマークされます。イベントの発生時点で、トリガーの有効性が妥当性チェックされます。トリガーが有効でない場合は、VALID WITH ERRORS とマークされ、イベントが失敗します。

注意：

- STARTUP イベントに関しては例外が 1 つあります。STARTUP イベントは、トリガーが失敗しても正常に実行されます。SYSTEM としてログインした場合は、SHUTDOWN イベントおよび LOGON イベントに関しても例外があります。
- メッセージのエンキューには DBMS_AQ パッケージが使用されるため、トリガーとキューの間の依存性は維持されません。

トリガーの再コンパイル

トリガーを手動で再コンパイルするには、ALTER TRIGGER コマンドを使用します。たとえば、次の文は PRINT_SALARY_CHANGES トリガーを再コンパイルします。

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

トリガーを再コンパイルするには、トリガーを所有しているか、または ALTER ANY TRIGGER システム権限が必要です。

トリガーの移行の問題

コンパイルされていないトリガーは、コンパイル済のトリガーのリリース（Oracle 7 リリース 7.3 および Oracle8）では起動できません。コンパイルされていないトリガーのリリースをコンパイル済のトリガーのリリースにアップグレードする場合、既存のすべてのトリガーをコンパイルする必要があります。アップグレード・スクリプト cat73xx.sql を実行すると、トリガーが最初の実行時に自動的に再コンパイルされるように、すべてのトリガーが無効にされます（*xx* は、可変のマイナー・リリース番号を表します）。

Oracle 7 リリース 7.3 以上をリリース 7.3 より前にダウングレードするには、ダウングレード・スクリプト cat73xx.sql を実行する必要があります。このスクリプトによって、ストアド・トリガーのリリースとそれ以外のトリガーのリリースとの間の移植性に関する問題が処理されます。

トリガーの変更

ストアド・プロシージャと同様に、トリガーは明示的に変更することはできません。つまり、新しい定義と置き換える必要があります（ALTER TRIGGER 文は、トリガーを再コンパイルするか、使用可能にするかまたは使用禁止にするためにのみ使用します）。

トリガーを置き換えるときは、CREATE TRIGGER 文に OR REPLACE オプションを指定する必要があります。OR REPLACE オプションを使用することによって、元のトリガーの権限付与に影響せずに、古いトリガーを新しいトリガーに置き換えることができます。

また、トリガーは DROP TRIGGER 文を使用して削除でき、削除してから CREATE TRIGGER 文を再実行できます。

トリガーを削除するには、トリガーが自スキーマ内にあるか、または DROP ANY TRIGGER システム権限が必要です。

トリガーのデバッグ

ストアド・プロシージャで使用できる機能と同じ機能を使用して、トリガーをデバッグできます。

> **参照：** 9-38 ページの「ストアド・プロシージャのデバッグ」を参照してください。

トリガーの使用可能および使用禁止

トリガーは、次の 2 つのモードのどちらかです。

使用可能 : トリガーを実行する文が入力され、トリガー制限が（存在する場合に）TRUE と評価された場合、使用可能トリガーによってトリガー本体が実行されます。

使用禁止 : トリガーを実行する文が入力され、トリガー制限が（存在する場合に）TRUE と評価された場合でも、使用禁止トリガーはトリガー本体を実行しません。

トリガーの使用可能

デフォルトでは、トリガーは、作成時に自動的に使用可能に設定されます。ただし、トリガーは後で使用禁止にできます。トリガーを使用禁止にする必要がある作業を終了した後は、トリガーが適切なときに起動されるように、再び使用可能にしておきます。

使用禁止にしたトリガーを使用可能にするには、ALTER TRIGGER 文の ENABLE オプションを使用します。INVENTORY 表の使用禁止にされた REORDER トリガーを使用可能にするには、次のように入力します。

```
ALTER TRIGGER Reorder ENABLE;
```

ALTER TABLE 文の ENABLE 句に ALL TRIGGERS オプションを使用すると、1 つの文で、ある表に定義されているすべてのトリガーを使用可能にできます。たとえば、INVENTORY 表に定義されているすべてのトリガーを使用可能にするには、次のように指定します。

```
ALTER TABLE Inventory
    ENABLE ALL TRIGGERS;
```

トリガーの使用禁止

次のような場合、一時的にトリガーを使用禁止にできます。

- トリガーが参照するオブジェクトが使用できない場合
- 大規模なデータ・ロードを実行する必要があり、トリガーを起動せずに迅速に処理する場合
- データを再ロードする場合

デフォルトでは、トリガーは作成時に使用可能に設定されます。トリガーを使用禁止にするには、ALTER TRIGGER 文の DISABLE オプションを使用します。

たとえば、INVENTORY 表の REORDER トリガーを使用禁止にするには、次のように指定します。

```
ALTER TRIGGER Reorder DISABLE;
```

ALTER TABLE 文の DISABLE 句に ALL TRIGGERS オプションを使用すると、1 つの文で、ある表に対応づけられたすべてのトリガーを使用禁止にできます。たとえば、INVENTORY 表に定義されているすべてのトリガーを使用禁止にするには、次のように指定します。

```
ALTER TABLE Inventory
    DISABLE ALL TRIGGERS;
```

トリガーに関する情報のリスト

次のデータ・ディクショナリ・ビューには、トリガーに関する情報が表示されます。

- USER_TRIGGERS
- ALL_TRIGGERS
- DBA_TRIGGERS

新しい列 BASE_OBJECT_TYPE は、トリガーが DATABASE、SCHEMA、表またはビューのどれに基づくかを示します。基になるオブジェクトが表またはビューでない場合は、古い列 TABLE_NAME が NULL です。

ACTION_TYPE 列は、トリガーがコール型のトリガーか PL/SQL トリガーかを示します。

TRIGGER_TYPE 列には、システム・イベントにのみ適用される他の 2 つの値、BEFORE EVENT および AFTER EVENT が含まれます。

TRIGGERING_EVENT 列には、システム・イベントおよび DML イベントがすべて含まれます。

参照： これらのデータ・ディクショナリ・ビューの詳細は、『Oracle9*i* データベース・リファレンス』を参照してください。

たとえば、REORDER トリガーを作成する次の文を考えます。

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE OR REPLACE TRIGGER Reorder
AFTER UPDATE OF Parts_on_hand ON Inventory
FOR EACH ROW
WHEN(new.Parts_on_hand < new.Reorder_point)
DECLARE
   x NUMBER;
BEGIN
   SELECT COUNT(*) INTO x
      FROM Pending_orders
      WHERE Part_no = :new.Part_no;
   IF x = 0  THEN
      INSERT INTO Pending_orders
         VALUES (:new.Part_no, :new.Reorder_quantity,
                 sysdate);
   END IF;
END;
```

次の 2 つの問合せは、REORDER トリガーに関する情報を戻します。

```
SELECT Trigger_type, Triggering_event, Table_name
   FROM USER_TRIGGERS
   WHERE Trigger_name = 'REORDER';

TYPE             TRIGGERING_STATEMENT       TABLE_NAME
---------------- -------------------------- ------------
AFTER EACH ROW   UPDATE                     INVENTORY

SELECT Trigger_body
   FROM USER_TRIGGERS
   WHERE Trigger_name = 'REORDER';

TRIGGER_BODY
-----------------------------------------
DECLARE
   x NUMBER;
BEGIN
   SELECT COUNT(*) INTO x
      FROM Pending_orders
```

```
      WHERE Part_no = :new.Part_no;
   IF x = 0
      THEN INSERT INTO Pending_orders
         VALUES (:new.Part_no, :new.Reorder_quantity,
            sysdate);
   END IF;
END;
```

トリガー・アプリケーションの例

トリガーを使用して、様々な方法で Oracle データベースの情報管理をカスタマイズできます。一般に、トリガーは次の用途に使用します。

- 高度な監査
- 無効なトランザクションの排除
- 参照整合性の施行（宣言整合性制約がサポートしないアクション、または分散データベース中の複数ノードにまたがるアクションの両方に対して）
- 複雑なビジネス・ルールの施行
- 複雑なセキュリティ認可の施行
- 透過的なイベント・ロギング
- 導出列値の自動生成
- 更新可能な複合ビューの作成
- システム・イベントの追跡

この項では、これらのトリガー・アプリケーションの例を紹介します。これらの例をそのまま使用することはできませんが、トリガーを設計するときの参考にしてください。

トリガーを使用した監査 : 例

トリガーは、Oracle の組込み監査機能を補うためによく使用されます。トリガーを作成して、AUDIT コマンドによって記録される情報と同様の情報を記録することはできますが、トリガーは、より詳細な監査情報が必要な場合に使用します。たとえば、トリガーを使用すると、行単位の値に基づく監査が可能です。

Oracle の AUDIT 文が、機密保護監査機能と考えられていることに対して、トリガーは、ファイナンシャル監査機能を提供します。

データベース・アクティビティを監査するトリガーを作成するかどうかを判断するときは、トリガーで定義される監査に比べて Oracle の監査機能で何が提供されるかを検討します。

監査機能	組込み監査とトリガー・ベースの監査の比較
DML および DDL の監査	標準監査オプションによって、すべてのタイプのスキーマ・オブジェクトと構造体に関する DML 文と DDL 文の監査が可能です。これに比べて、トリガーでは、表に対して入力された DML 文の監査と、SCHEMA または DATABASE レベルでの DDL 監査ができます。
集中監査証跡	すべてのデータベース監査情報は、Oracle の監査機能によって自動的、集中的に記録されます。
宣言方式	トリガーで定義された監査機能と比べ、Oracle の標準機能で使用可能になる監査機能は宣言およびメンテナンスが簡単で、エラーが発生しにくくなります。
監査オプションの監査	既存の監査オプションの変更を監査して、不当なデータベース・アクティビティを防止できます。
セッションおよび実行時の監査	データベース監査機能を使用して、監査文が入力されるたびに（BY ACCESS）、または監査文を入力するセッションごとに（BY SESSION）、レコードを生成できます。トリガーではセッション単位の監査はできません。監査レコードは、トリガーで監査される表が参照されるたびに生成されます。
失敗したデータ・アクセスの監査	データ・アクセスがエラーとなった場合、データベース監査を実施するように設定できます。ただし、自律型トランザクションが使用されないかぎり、トリガーを実行する文がロールバックされると、トリガーによって生成された監査情報もロールバックされます。自律型トランザクションの詳細は、『Oracle9*i* データベース概要』を参照してください。
セッションの監査	標準データベース監査機能を使用して、接続および切断のみでなく、セッション・アクティビティ（物理 I/O、論理 I/O、デッドロックなど）も記録できます。

トリガーを使用して高度な監査を行うには、通常、AFTER トリガーを使用します。AFTER トリガーを使用すると、トリガーを実行する文が適切な整合性制約に従った後で、監査情報が記録されます。これによって、整合性制約の例外を生成する文に対する無効な監査処理の実行を防止します。

AFTER 行トリガーと AFTER 文トリガーの使い分けは、監査情報に応じて異なります。たとえば、行トリガーを使用すると、表の行単位の値に基づく監査が可能です。トリガーでは、監査済 SQL 文を発行するための理由コードの入力をユーザーに要求することもできます。これは、行レベルおよび文レベルの両方の監査状況に有効です。

次の例では、Emp_tab 表に対する変更を行ベースで監査するトリガーを示します。この例では、更新前に理由コードをグローバル・パッケージ変数に格納する必要があります。トリ

ガーを使用して値ベースの監査を実行する方法、およびパブリック・パッケージ変数を使用する方法を示します。

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE OR REPLACE PACKAGE Auditpackage AS
   Reason VARCHAR2(10);
PROCEDURE Set_reason(Reason VARCHAR2);
END;
CREATE TABLE Emp99 (
   Empno               NOT NULL   NUMBER(4)
   Ename               VARCHAR2(10)
   Job                 VARCHAR2(9)
   Mgr                 NUMBER(4)
   Hiredate            DATE
   Sal                 NUMBER(7,2)
   Comm                NUMBER(7,2)
   Deptno              NUMBER(2)
   Bonus               NUMBER
   Ssn                 NUMBER
   Job_classification  NUMBER);

CREATE TABLE Audit_employee (
   Oldssn              NUMBER
   Oldname             VARCHAR2(10)
   Oldjob              VARCHAR2(2)
   Oldsal              NUMBER
   Newssn              NUMBER
   Newname             VARCHAR2(10)
   Newjob              VARCHAR2(2)
   Newsal              NUMBER
   Reason              VARCHAR2(10)
   User1               VARCHAR2(10)
   Systemdate          DATE);
```

```
CREATE OR REPLACE TRIGGER Audit_employee
AFTER INSERT OR DELETE OR UPDATE ON Emp99
FOR EACH ROW
BEGIN
/* AUDITPACKAGE is a package with a public package
   variable REASON.  REASON could be set by the
   application by a command such as EXECUTE
   AUDITPACKAGE.SET_REASON(reason_string). Note that a
   package variable has state for the duration of a
   session and that each session has a separate copy of
   all package variables. */
```

```
IF Auditpackage.Reason IS NULL THEN
   Raise_application_error(-20201, 'Must specify reason'
      || ' with AUDITPACKAGE.SET_REASON(Reason_string)');
END IF;

/* If the above conditional evaluates to TRUE, the
   user-specified error number and message is raised,
   the trigger stops execution, and the effects of the
   triggering statement are rolled back.  Otherwise, a
   new row is inserted into the predefined auditing
   table named AUDIT_EMPLOYEE containing the existing
   and new values of the Emp_tab table and the reason code
   defined by the REASON variable of AUDITPACKAGE.  Note
   that the "old" values are NULL if triggering
   statement is an INSERT and the "new" values are NULL
   if the triggering statement is a DELETE. */

INSERT INTO Audit_employee VALUES
   (:old.Ssn, :old.Ename, :old.Job_classification, :old.Sal,
   :new.Ssn, :new.Ename, :new.Job_classification, :new.Sal,
   auditpackage.Reason, User, Sysdate );
END;
```

更新のたびに強制的に理由コードを設定する場合は、理由コードを NULL に設定しなおすこともできます。次の簡単な AFTER 文トリガーは、トリガーを実行する文が実行された後で理由コードを NULL に設定します。

```
CREATE OR REPLACE TRIGGER Audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON Emp_tab
BEGIN
   auditpackage.set_reason(NULL);
END;
```

前述のトリガーは、2 つとも同じ種類の SQL 文によって起動されます。ただし、AFTER 行トリガーが、トリガーを実行する文によって影響を受ける表の行ごとに 1 回起動されるのに対して、AFTER 文トリガーは、トリガーを実行する文の実行が終了したときに 1 回のみ起動されます。

次に示すトリガーもトリガーを使用して監査を行います。このトリガーは、Emp_tab 表に加えられる変更を追跡し、この情報を AUDIT_TABLE と AUDIT_TABLE_VALUES に格納します。

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Audit_table (
   Seq      NUMBER,
   User_at  VARCHAR2(10),
   Time_now DATE,
   Term     VARCHAR2(10),
   Job      VARCHAR2(10),
   Proc     VARCHAR2(10),
   enum     NUMBER);
CREATE SEQUENCE Audit_seq;
CREATE TABLE Audit_table_values (
   Seq      NUMBER,
   Dept     NUMBER,
   Dept1    NUMBER,
   Dept2    NUMBER);
```

```
CREATE OR REPLACE TRIGGER Audit_emp
   AFTER INSERT OR UPDATE OR DELETE ON Emp_tab
   FOR EACH ROW
   DECLARE
      Time_now DATE;
      Terminal CHAR(10);
   BEGIN
      -- get current time, and the terminal of the user:
      Time_now := SYSDATE;
      Terminal := USERENV('TERMINAL');
      -- record new employee primary key
      IF INSERTING THEN
         INSERT INTO Audit_table
            VALUES (Audit_seq.NEXTVAL, User, Time_now,
               Terminal, 'Emp_tab', 'INSERT', :new.Empno);
      -- record primary key of the deleted row:
      ELSIF DELETING THEN
         INSERT INTO Audit_table
            VALUES (Audit_seq.NEXTVAL, User, Time_now,
               Terminal, 'Emp_tab', 'DELETE', :old.Empno);
      -- for updates, record the primary key
      -- of the row being updated:
      ELSE
         INSERT INTO Audit_table
            VALUES (audit_seq.NEXTVAL, User, Time_now,
               Terminal, 'Emp_tab', 'UPDATE', :old.Empno);
         -- and for SAL and DEPTNO, record old and new values:
         IF UPDATING ('SAL') THEN
```

```
            INSERT INTO Audit_table_values
               VALUES (Audit_seq.CURRVAL, 'SAL',
                  :old.Sal, :new.Sal);

         ELSIF UPDATING ('DEPTNO') THEN
            INSERT INTO Audit_table_values
               VALUES (Audit_seq.CURRVAL, 'DEPTNO',
                  :old.Deptno, :new.DEPTNO);
         END IF;
      END IF;
END;
```

整合性制約およびトリガー : 例

トリガーおよび宣言整合性制約は、両方ともデータ入力の制限に使用できます。ただし、トリガーと整合性制約には大きな違いがあります。

宣言整合性制約はデータベースに関する文で、これは常に TRUE です。表内の既存のデータおよび表を操作するすべての文に対して制約が適用されます。

> **参照 :** 第 4 章「制約によるデータ整合性のメンテナンス」を参照してください。

トリガーは、トランザクションで可能な処理を制約します。トリガーは、トリガーが定義される前にロードされたデータには適用されません。このため、表内のすべてのデータが、対応付けられたトリガーによって確立されたルールに適合するかどうかは確認できません。

トリガーを使用して Oracle の宣言整合性制約機能がサポートするものと同様のルールの多くを施行することもできますが、トリガーは、標準の整合性制約では定義できない複雑なビジネス・ルールを規定するためにのみ使用するようにしてください。Oracle の宣言整合性制約機能には、トリガーで定義する制約に比べて、次のメリットがあります。

一元化された整合性チェック : すべてのデータ・アクセス・ポイントは、各スキーマ・オブジェクトに対応する整合性制約によって定義されたグローバルな一連のルールに準拠する必要があります。

宣言方式 : 標準の整合性制約機能を使用して定義された制約は、トリガーで定義された同等の制約と比較して、より作成しやすくエラーが発生しにくいというメリットがあります。

データ整合性のほとんどは、宣言整合性制約によって定義して施行できますが、トリガーは宣言整合性制約では定義できない複雑なビジネス制約の施行に使用できます。たとえば、トリガーを使用して次を施行できます。

- UPDATE と DELETE SET NULL、および UPDATE と DELETE SET DEFAULT の参照アクション
- 親表と子表が分散データベースの異なるノード上にある場合の参照整合性
- CHECK 制約で指定できる式では定義できない複雑な CHECK 制約

トリガーを使用した参照整合性

参照整合性のほとんどは、トリガーを使用して施行できます。ただし、トリガーを使用するのは、UPDATE および DELETE SET NULL 参照アクション（参照データが更新または削除されると、関連するすべての従属データは NULL に置き換えられます）または UPDATE および DELETE SET DEFAULT 参照アクション（参照データが更新または削除されると、関連するすべての従属データはデフォルト値に置き換えられます）を施行する場合、あるいは、分散データベースの異なるノードにある親表と子表の間で参照整合性を施行する場合のみにします。

トリガーを使用して参照整合性をメンテナンスするときは、親表に主キー（または一意キー）制約を宣言します。同じデータベース内の親表と子表間の参照整合性をトリガーでメンテナンスしている場合は、子表にも外部キーを宣言できますが、外部キーは使用禁止に設定してください。使用禁止にすることによって、対応する主キー制約が（CASCADE オプションを使用して、主キー制約を明示的に削除しないかぎり）削除されなくなります。

トリガーを使用して参照整合性をメンテナンスするには、次のようにします。

- 子表にトリガーを 1 つ定義する必要があります。このトリガーは、外部キーに挿入または更新される値が親キーの値と対応することを保証します。
- 親表には、1 つ以上のトリガーを定義する必要があります。このトリガーによって、親キーで値が更新または削除されたときに、外部キーの値に対して適切な参照アクション（RESTRICT、CASCADE または SET NULL）が実行されることを保証します。親表への挿入にアクションは不要です（依存する外部キーはありません）。

次の項では、参照整合性の規定に必要なトリガーの例を紹介します。これらの例では Emp_tab 表および Dept_tab 表を使用します。

トリガーのいくつかには、行をロックする文（SELECT... FOR UPDATE）が含まれています。この操作は、行を処理するときの並行性のメンテナンスに必要です。

子表に対する外部キー・トリガー 次のトリガーでは、INSERT 文または UPDATE 文が外部キーに影響する前に、対応する値が親キー内に確実に存在するようにします。次の例に含まれる変更表例外によって、このトリガーを UPDATE_SET_DEFAULT トリガーおよび UPDATE_CASCADE トリガーとともに使用できるようになります。このトリガーを単独で使用する場合は、この例外を削除できます。

```
CREATE OR REPLACE TRIGGER Emp_dept_check
BEFORE INSERT OR UPDATE OF Deptno ON Emp_tab
FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the Emp_tab
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the Dept_tab table.
DECLARE
   Dummy              INTEGER;  -- used for cursor fetch below
   Invalid_department EXCEPTION;
   Valid_department   EXCEPTION;
```

```
   Mutating_table     EXCEPTION;
   PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

-- Cursor used to verify parent key value exists.  If
-- present, lock parent key's row so it can't be
-- deleted by another transaction until this
-- transaction is committed or rolled back.
  CURSOR Dummy_cursor (Dn NUMBER) IS
   SELECT Deptno FROM Dept_tab
      WHERE Deptno = Dn
         FOR UPDATE OF Deptno;
BEGIN
   OPEN Dummy_cursor (:new.Deptno);
   FETCH Dummy_cursor INTO Dummy;

   -- Verify parent key.  If not found, raise user-specified
   -- error number and message.  If found, close cursor
   -- before allowing triggering statement to complete:
   IF Dummy_cursor%NOTFOUND THEN
      RAISE Invalid_department;
   ELSE
      RAISE valid_department;
   END IF;
   CLOSE Dummy_cursor;
EXCEPTION
   WHEN Invalid_department THEN
      CLOSE Dummy_cursor;
      Raise_application_error(-20000, 'Invalid Department'
         || ' Number' || TO_CHAR(:new.deptno));
   WHEN Valid_department THEN
      CLOSE Dummy_cursor;
   WHEN Mutating_table THEN
      NULL;
END;
```

親表に対する UPDATE トリガーおよび DELETE RESTRICT トリガー 次のトリガーを DEPT_TAB 表に定義し、DEPT_TAB 表の主キーに対して UPDATE および DELETE RESTRICT 参照アクションを施行します。

```
CREATE OR REPLACE TRIGGER Dept_restrict
BEFORE DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, check for dependent
-- foreign key values in Emp_tab; rollback if any are found.
DECLARE
```

```
   Dummy                 INTEGER;      -- used for cursor fetch below
   Employees_present     EXCEPTION;
   employees_not_present EXCEPTION;

   -- Cursor used to check for dependent foreign key values.
   CURSOR Dummy_cursor (Dn NUMBER) IS
      SELECT Deptno FROM Emp_tab WHERE Deptno = Dn;

BEGIN
   OPEN Dummy_cursor (:old.Deptno);
   FETCH Dummy_cursor INTO Dummy;
   -- If dependent foreign key is found, raise user-specified
   -- error number and message.  If not found, close cursor
   -- before allowing triggering statement to complete.
   IF Dummy_cursor%FOUND THEN
      RAISE Employees_present;     -- dependent rows exist
   ELSE
      RAISE Employees_not_present; -- no dependent rows
   END IF;
   CLOSE Dummy_cursor;

EXCEPTION
   WHEN Employees_present THEN
      CLOSE Dummy_cursor;
      Raise_application_error(-20001, 'Employees Present in'
         || ' Department ' || TO_CHAR(:old.DEPTNO));
   WHEN Employees_not_present THEN
      CLOSE Dummy_cursor;
END;
```

注意： このトリガーは、自己参照型の表（主キーまたは一意キーが存在し、さらに外部キーが存在する表）では機能しません。また、このトリガーでは、トリガーの循環（A が B を起動し、B が A を起動する）は使用できません。

親表に対する UPDATE および DELETE SET NULL トリガー：例 次のトリガーを DEPT_TAB 表に定義し、DEPT_TAB 表の主キーに対して UPDATE および DELETE SET NULL 参照アクションを施行します。

```
CREATE OR REPLACE TRIGGER Dept_set_null
AFTER DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, set all corresponding
-- dependent foreign key values in Emp_tab to NULL:
```

```
BEGIN
   IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
      UPDATE Emp_tab SET Emp_tab.Deptno = NULL
         WHERE Emp_tab.Deptno = :old.Deptno;
   END IF;
END;
```

親表に対する DELETE CASCADE トリガー : 例 DEPT_TAB 表に対する次のトリガーは、DEPT_TAB 表の主キーに対して DELETE CASCADE 参照アクションを施行します。

```
CREATE OR REPLACE TRIGGER Dept_del_cascade
AFTER DELETE ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab, delete all
-- rows from the Emp_tab table whose DEPTNO is the same as
-- the DEPTNO being deleted from the Dept_tab table:
BEGIN
   DELETE FROM Emp_tab
      WHERE Emp_tab.Deptno = :old.Deptno;
END;
```

> **注意 :** 通常、DELETE CASCADE のコードは、更新および削除の両方の可能性を考慮して、UPDATE SET NULL または UPDATE SET DEFAULT のコードと組み合されます。

親表に対する UPDATE CASCADE トリガー : 例 次のトリガーは、Dept_tab 表の部門番号が更新されたときに、その変更が Emp_tab 表の依存外部キーに確実に伝播されるようにします。

```
-- Generate a sequence number to be used as a flag for
-- determining if an update has occurred on a column:
CREATE SEQUENCE Update_sequence
    INCREMENT BY 1 MAXVALUE 5000
    CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AS
   Updateseq NUMBER;
END Integritypackage;

CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
-- create flag col:
ALTER TABLE Emp_tab ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER Dept_cascade1 BEFORE UPDATE OF Deptno ON Dept_tab
```

```
DECLARE
   Dummy NUMBER;

-- Before updating the Dept_tab table (this is a statement
-- trigger), generate a new sequence number and assign
-- it to the public variable UPDATESEQ of a user-defined
-- package named INTEGRITYPACKAGE:
BEGIN
   SELECT Update_sequence.NEXTVAL
      INTO Dummy
      FROM dual;
   Integritypackage.Updateseq := Dummy;
END;

CREATE OR REPLACE TRIGGER Dept_cascade2 AFTER DELETE OR UPDATE
   OF Deptno ON Dept_tab FOR EACH ROW

-- For each department number in Dept_tab that is updated,
-- cascade the update to dependent foreign keys in the
-- Emp_tab table.  Only cascade the update if the child row
-- has not already been updated by this trigger:
BEGIN
   IF UPDATING THEN
      UPDATE Emp_tab
         SET Deptno = :new.Deptno,
         Update_id = Integritypackage.Updateseq   --from 1st
         WHERE Emp_tab.Deptno = :old.Deptno
         AND Update_id IS NULL;
         /* only NULL if not updated by the 3rd trigger
            fired by this same triggering statement */
   END IF;
   IF DELETING THEN

   -- Before a row is deleted from Dept_tab, delete all
   -- rows from the Emp_tab table whose DEPTNO is the same as
   -- the DEPTNO being deleted from the Dept_tab table:
      DELETE FROM Emp_tab
      WHERE Emp_tab.Deptno = :old.Deptno;
   END IF;
END;
CREATE OR REPLACE TRIGGER Dept_cascade3 AFTER UPDATE OF Deptno ON Dept_tab
BEGIN  UPDATE Emp_tab
   SET Update_id = NULL
   WHERE Update_id = Integritypackage.Updateseq;
END;
```

注意： このトリガーによって Emp_tab 表が更新されるため、Emp_dept_check トリガーも起動されます（トリガーが使用可能になっている場合）。結果の変更表エラーは、Emp_dept_check トリガーによって検出されます。エラーの検出を必要とするトリガーは、本番環境で常に正常に動作することを保証するために、慎重にテストする必要があります。

複雑な CHECK 制約に対するトリガー : 例

トリガーは、参照整合性以外の整合性ルールも施行できます。たとえば、次のトリガーは、トリガーを実行する文の実行を許可する前に、複雑なチェックを実行します。

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Salgrade (
   Grade                NUMBER,
   Losal                NUMBER,
   Hisal                NUMBER,
   Job_classification   NUMBER)
```

```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
FOR EACH ROW
DECLARE
   Minsal               NUMBER;
   Maxsal               NUMBER;
   Salary_out_of_range  EXCEPTION;
BEGIN

/* Retrieve the minimum and maximum salary for the
   employee's new job classification from the SALGRADE
   table into MINSAL and MAXSAL: */

SELECT Minsal, Maxsal INTO Minsal, Maxsal FROM Salgrade
   WHERE Job_classification = :new.Job;


/* If the employee's new salary is less than or greater
   than the job classification's limits, the exception is
   raised.  The exception message is returned and the
   pending INSERT or UPDATE statement that fired the
   trigger is rolled back:*/

   IF (:new.Sal < Minsal OR :new.Sal > Maxsal) THEN
```

```
      RAISE Salary_out_of_range;
   END IF;
EXCEPTION
   WHEN Salary_out_of_range THEN
      Raise_application_error (-20300,
         'Salary '||TO_CHAR(:new.Sal)||' out of range for '
         ||'job classification '||:new.Job
         ||' for employee '||:new.Ename);
   WHEN NO_DATA_FOUND THEN
      Raise_application_error(-20322,
         'Invalid Job Classification '
         ||:new.Job_classification);
END;
```

複雑なセキュリティ認可およびトリガー : 例

トリガーは、一般に、表データに対する複雑なセキュリティ認可の施行によく使用されます。トリガーは、Oracle が提供するデータベース・セキュリティ機能では定義できない複雑なセキュリティ認可の施行にのみ使用します。たとえば、トリガーを使用して、週末、休日および休業日には、Emp_tab 表の給与データを更新できないようにすることができます。

複雑なセキュリティ認可の施行にトリガーを使用する場合は、BEFORE 文トリガーを使用すると最も効果的です。BEFORE 文トリガーを使用すると、次のようなメリットがあります。

- トリガーを実行する文の実行が許可される前に、セキュリティ・チェックが実行されるため、不当な文による無駄な作業をせずに済みます。
- セキュリティ・チェックは、トリガーを実行する文に影響される行ごとではなく、トリガーを実行する文に対して 1 回のみ実施されます。

次の例は、セキュリティを施行するために使用するトリガーを示します。

注意 : 次のデータ構造を設定しないと機能しない例もあります。

```
CREATE TABLE Company_holidays (Day DATE);
```

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON Emp99
DECLARE
   Dummy             INTEGER;
   Not_on_weekends   EXCEPTION;
   Not_on_holidays   EXCEPTION;
   Non_working_hours EXCEPTION;
BEGIN
   /* check for weekends: */
   IF (TO_CHAR(Sysdate, 'DY') = 'SAT' OR
      TO_CHAR(Sysdate, 'DY') = 'SUN') THEN
```

```
      RAISE Not_on_weekends;
   END IF;
   /* check for company holidays:*/
   SELECT COUNT(*) INTO Dummy FROM Company_holidays
      WHERE TRUNC(Day) = TRUNC(Sysdate);
      /* TRUNC gets rid of time parts of dates: */
   IF dummy > 0 THEN
      RAISE Not_on_holidays;
   END IF;
   /* Check for work hours (8am to 6pm): */
   IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
       TO_CHAR(Sysdate, 'HH24') > 18) THEN
       RAISE Non_working_hours;
   END IF;
EXCEPTION
   WHEN Not_on_weekends THEN
      Raise_application_error(-20324,'May not change '
         ||'employee table during the weekend');
   WHEN Not_on_holidays THEN
      Raise_application_error(-20325,'May not change '
         ||'employee table during a holiday');
   WHEN Non_working_hours THEN
      Raise_application_error(-20326,'May not change '
      ||'Emp_tab table during non-working hours');
END;
```

透過的なイベント・ロギングおよびトリガー

特定のイベントに続いてデータベースに透過的な変更を実行する場合、トリガーは非常に効果的です。

REORDER トリガーの例では、一定の条件が満たされる（トリガーを実行する文が入力され、PARTS_ON_HAND 値が REORDER_POINT 値より小さい場合）と、必要に応じて部品を再注文するトリガーを示します。

導出列値およびトリガー : 例

トリガーは、INSERT 文または UPDATE 文に指定される値に基づいて、列の値を自動的に取得できます。この種のトリガーは、同一行上の別の列の値に依存する特定の列に値を埋め込む場合に便利です。この種の操作を実行するには、BEFORE 行トリガーが必要ですが、これは次の理由によります。

- トリガーを実行する文で導出値を使用できるようにするには、INSERT または UPDATE が発生する前に依存値を取得する必要があります。
- トリガーを実行する INSERT 文または UPDATE 文によって影響される行ごとに、トリガーを起動する必要があります。

次の例では、行が挿入または更新されるたびに、表の新しい列値を導出するトリガーの使用方法を示します。

注意： 次のデータ構造を設定しないと機能しない例もあります。

```
ALTER TABLE Emp99 ADD(
   Uppername   VARCHAR2(20),
   Soundexname VARCHAR2(20));
```

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp99

/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Users should be
   restricted from updating these fields directly: */
FOR EACH ROW
BEGIN
   :new.Uppername := UPPER(:new.Ename);
   :new.Soundexname := SOUNDEX(:new.Ename);
END;
```

トリガーを使用した複合更新可能ビューの作成 : 例

ビューは、表データに対して論理ウィンドウを提供する優れた手段です。ただし、ビュー問合せが複雑になると、ビューに対する DML から基礎となる表に対する DML への変換を、システムが暗黙的には実行できなくなります。この問題の解決には、INSTEAD OF トリガーが有効です。このトリガーは、ビューに対して定義することができ、実際の DML のかわりに起動されます。

書籍が書名の順に配置されているライブラリ・システムを考えてみます。このライブラリは、書籍型オブジェクトのコレクションで構成されています。次の例はこのスキーマについて説明したものです。

```
CREATE OR REPLACE TYPE Book_t AS OBJECT
(
   Booknum   NUMBER,
   Title     VARCHAR2(20),
   Author    VARCHAR2(20),
   Available CHAR(1)
);
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
```

関係スキーマに次の表があるとします。

```
Table Book_table (Booknum, Section, Title, Author, Available)
```

Booknum	Section	Title	Author	Available
121001	Classic	Iliad	Homer	Y
121002	Novel	Gone With the Wind	Mitchell M	N

このライブラリは、library_table(section) で構成されています。

Section
Geography
Classic

これらの表に対して複合ビューを定義し、セクションおよび各セクション内の書籍集合を示すライブラリの論理ビューを作成することができます。

```
CREATE OR REPLACE VIEW Library_view AS
SELECT i.Section, CAST (MULTISET (
   SELECT b.Booknum, b.Title, b.Author, b.Available
   FROM Book_table b
   WHERE b.Section = i.Section) AS Book_list_t) BOOKLIST
FROM Library_table i;
```

このビューに対して INSTEAD OF トリガーを定義し、このビューを更新可能にします。

```
CREATE OR REPLACE TRIGGER Library_trigger INSTEAD OF INSERT ON Library_view FOR EACH
ROW
   Bookvar BOOK_T;
   i       INTEGER;
BEGIN
   INSERT INTO Library_table VALUES (:NEW.Section);
   FOR i IN 1..:NEW.Booklist.COUNT LOOP
      Bookvar := Booklist(i);
      INSERT INTO book_table
         VALUES ( Bookvar.booknum, :NEW.Section, Bookvar.Title, Bookvar.Author,
bookvar.Available);
   END LOOP;
END;
/
```

これで library_view は更新可能ビューになり、このビューに対するすべての INSERT は、自動的に起動されるトリガーによって処理されます。次に例を示します。

```
INSERT INTO Library_view VALUES ('History', book_list_t(book_t(121330, 'Alexander',
'Mirth', 'Y');
```

同様に、ネストした表である booklist に対してトリガーを定義して、ネストした表の要素の変更を処理することもできます。

トリガーを使用したシステム・イベントの追跡

トリガーを使用したファイングレイン・アクセス・コントロール : 例 システム・トリガーは、アプリケーション・コンテキストの設定に使用できます。アプリケーション・コンテキストは比較的新しい機能であり、ファイングレイン・アクセス・コントロールを実装する機能が向上します。アプリケーション・コンテキストは保護されたセッション・キャッシュで、セッション固有の属性の格納に使用できます。

次に示す例では、set_ctx プロシージャがユーザー・プロファイルに基づいてアプリケーション・コンテキストを設定します。setexpensectx トリガーは、コンテキストがユーザーごとに確実に設定されるようにします。

```
CONNECT secdemo/secdemo

CREATE OR REPLACE CONTEXT Expenses_reporting USING Secdemo.Exprep_ctx;

REM ================================================================
REM Creation of the package which implements the context:
REM ================================================================

CREATE OR REPLACE PACKAGE Exprep_ctx AS
 PROCEDURE Set_ctx;
END;

SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY Exprep_ctx IS
   PROCEDURE Set_ctx IS
      Empnum   NUMBER;
      Countrec NUMBER;
      Cc       NUMBER;
      Role     VARCHAR2(20);
   BEGIN

      -- SET emp_number:
      SELECT Employee_id INTO Empnum FROM Employee
         WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');

      DBMS_SESSION.SET_CONTEXT('expenses_reporting','emp_number', Empnum);

      -- SET ROLE:
      SELECT COUNT (*) INTO Countrec FROM Cost_center WHERE Manager_id=Empnum;
      IF (countrec > 0) THEN
```

```
         DBMS_SESSION.SET_CONTEXT('expenses_reporting','exp_role','MANAGER');
      ELSE
         DBMS_SESSION.SET_CONTEXT('expenses_reporting','exp_role','EMPLOYEE');
      END IF;

      -- SET cc_number:
      SELECT Cost_center_id INTO Cc FROM Employee
         WHERE Last_name = SYS_CONTEXT('userenv','session_user');
      DBMS_SESSION.SET_CONTEXT(expenses_reporting','cc_number',Cc);
   END;
END;
```

CALL 文の構文

```
CREATE OR REPLACE TRIGGER Secdemo.Setexpseetx
AFTER LOGON ON DATABASE
CALL Secdemo.Exprep_etx.Set_otx
```

トリガーを介したシステム・イベントに対する応答

Oracle がシステム・イベントを発行することによって、アプリケーションは、他のアプリケーションからのメッセージにサブスクライブするのと同じ方法で、データベース・イベントにサブスクライブできます。

参照： 第 16 章「システム・イベントの処理」を参照してください。

Oracle のシステム・イベント発行フレームワークには、次の機能が含まれています。

- パブリッシュ / サブスクライブのためのインフラストラクチャ。データベースがイベントのアクティブな発行者になります。
- サーバーへのデータ・カートリッジの統合。システム・イベントを発行して、サーバー内の状態の変更をカートリッジに通知することができます。
- サーバーへのファイングレイン・アクセス・コントロールの統合。

トリガーを作成すると、イベント発生時に実行するプロシージャを指定できます。DML イベントは表でサポートされ、システム・イベントは、DATABASE およびスキーマでサポートされます。ALTER TRIGGER 文を使用してトリガーを使用可能および使用禁止にすると、通知をオンおよびオフにすることができます。

この機能は、アドバンスト・キューイング・エンジンに統合されています。パブリッシュ / サブスクライブ・アプリケーションは、DBMS_AQ.ENQUEUE() プロシージャを使用し、他のアプリケーション（カートリッジなど）は、コールアウトを使用します。

参照：

- 『Oracle9*i* SQL リファレンス』を参照してください。
- 発行済のイベントへのサブスクライブ方法の詳細は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

トリガーを介したイベントの発行方法

サーバーによってイベントが検出されると、トリガー・メカニズムがトリガー内に指定されているアクションを実行します。このアクションの一部として、DBMS_AQ パッケージを使用してイベントをキューに発行でき、キューによって、サブスクライバが通知を取得します。

注意： この方法で検出できるのは、イベントの組込みセットのみです。独自のイベント条件は定義できません。

イベントが発生すると、そのイベントに対して使用可能なすべてのトリガーが起動されます。これには次の例外があります。

- トリガーがトリガー・イベントのターゲットである場合、このトリガーは起動されません。たとえば、すべての DROP イベントのトリガーは、削除される場合は起動されません。
- トリガーが変更されているが、イベントの起動と同じトランザクション内でコミットされていない場合、このトリガーは起動されません。たとえば、システム・トリガー内の再帰 DDL がトリガーを変更する場合がありますが、これによって変更されたトリガーが同じトランザクション内で起動されなくなります。

1 個のオブジェクトに複数のトリガーを作成できます。イベントが複数のトリガーを起動する場合、順序は定義されないため、特定の順序内で起動されるトリガーに依存しないでください。

発行コンテキスト

イベントが発行されるときは、パラメータ・リストに指定されている特定の実行時コンテキストおよび属性がコールアウト・プロシージャに渡されます。イベント属性関数と呼ばれる一連の関数が提供されています。

参照： イベント固有の属性の詳細は、16-2 ページの「イベント属性関数」を参照してください。

サポートされているシステム・イベントごとに、イベント固有の属性が指定され、事前定義されています。パラメータ・リストには、他の単純な式とともにこの属性のいずれかを選択できます。コールアウトの場合、これらは IN 引数として渡されます。

エラー処理

発行コールアウト関数からの戻り状態は、すべてのイベントに関して無視されます。たとえば、SHUTDOWN イベントの場合、サーバーは戻り状態に関しては何も実行できません。

> **参照：** 戻り状態の詳細は、16-6 ページの「データベース・イベントのリスト」を参照してください。

実行モデル

トリガーは、従来からトリガーの定義者として実行されてきました。イベントのトリガー・アクションは、アクションの定義者として（コールアウト内のパッケージまたはファンクションの定義者、またはキュー内のトリガーの所有者として）実行されます。トリガーの所有者には、基になるキュー、パッケージまたはプロシージャに対する EXECUTE 権限が必要なため、この動作には一貫性があります。

16

システム・イベントの処理

LOGON や SHUTDOWN などのシステム・イベントでは、システムの変更を追跡する機能が提供されます。Oracle では、この追跡をデータベース・イベント通知と結びつけることができます。データベース・イベント通知によって、簡単かつ優れた方法でアプリケーションに非同期メッセージを配信できます。

この章には、トリガーを作成できる様々なイベントの説明が含まれています。イベント属性関数のリストも提供されています。内容は次のとおりです。

- イベント属性関数
- データベース・イベントのリスト

参照：

この章を読む前に、第 15 章「トリガーの使用」の内容を理解しておいてください。トリガー本体内にあるイベント情報にアクセスする必要があります。

イベント処理のための柔軟なシステムを設定するには、Oracle Enterprise Manager の「Event」パネルを使用します。この方法を使用すると、この章で説明するシステム・イベントより多くの条件を検出できます。また、シェル・スクリプトの起動などのアクションも設定できます。

イベント属性関数

トリガーが起動されたら、トリガーを起動したイベントの属性を検索できます。各属性は、ファンクション・コールによって検索されます。

注意 :

- これらの属性を使用可能にするには、まず、CATPROC.SQL スクリプトを実行する必要があります。
- トリガー・ディクショナリ・オブジェクトは、発行されるイベントに関するメタデータおよびそれに対応する属性をメンテナンスします。
- 以前のリリースでは、これらのファンクションは SYS パッケージを介してアクセスされていました。名前が ora_ で始まるこれらのパブリック・シノニムを使用することをお薦めします。

表 16-1 システムで定義されたイベント属性

属性	型	説明	例

ora_client_ip_address	VARCHAR2	基礎となるプロトコルが TCP/IP のとき、LOGON イベントでクライアントの IP アドレスを戻します。	if (ora_sysevent = 'LOGON')
			  then addr :=
			ora_client_ip_address;
			end if;

ora_database_name	VARCHAR2(50)	データベース名を戻します。	DECLARE
			 db_name VARCHAR2(50);
			BEGIN
			  db_name := ora_database_name;
			END;

ora_des_encrypted_password	VARCHAR2	作成または変更されるユーザーの DES 暗号化パスワードを戻します。	IF (ora_dict_obj_type = 'USER')
			 THEN INSERT INTO event_table
			(ora_des_encrypted_password);
			END IF;

ora_dict_obj_name	VARCHAR(30)	DDL 操作が発生したディクショナリ・オブジェクトの名前を戻します。	INSERT INTO event_table ('Changed object is ' || ora_dict_obj_name');

ora_dict_obj_name_list
(name_list OUT
ora_name_list_t)
	BINARY_INTEGER	このイベントで変更されるオブジェクトのオブジェクト名のリストを戻します。	if (ora_sysevent = 'ASSOCIATE
			STATISTICS')
			  then number_modified :=
			ora_dict_obj_name_list
			(name_list);
			end if;

表 16-1 システムで定義されたイベント属性 （続き）

属性	型	説明	例
ora_dict_obj_owner	VARCHAR(30)	DDL 操作が発生したディクショナリ・オブジェクトの所有者を戻します。	INSERT INTO event_table ('object owner is' || ora_dict_obj_owner');
ora_dict_obj_owner_list(owner_list OUT ora_name_list_t)	BINARY_INTEGER	このイベントで変更されるオブジェクトのオブジェクト所有者のリストを戻します。	if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_of_modified_objects := ora_dict_obj_owner_list(owner_list); end if;
ora_dict_obj_type	VARCHAR(20)	DDL 操作が発生したディクショナリ・オブジェクトのタイプを戻します。	INSERT INTO event_table ('This object is a ' || ora_dict_obj_type);
ora_grantee(user_list OUT ora_name_list_t)	BINARY_INTEGER	OUT パラメータで権限付与イベントの権限受領者を戻し、戻り値で権限受領者の数を戻します。	if (ora_sysevent = 'GRANT') then number_of_users := ora_grantee(user_list); end if;
ora_instance_num	NUMBER	インスタンス番号を戻します。	IF (ora_instance_num = 1) THEN INSERT INTO event_table ('1'); END IF;
ora_is_alter_column(column_name IN VARCHAR2)	BOOLEAN	指定された列が変更された場合、true を戻します。	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then alter_column := ora_is_alter_column('FOO'); end if;
ora_is_creating_nested_table	BOOLEAN	現行のイベントがネストした表を作成しているときに、true を戻します。	if (ora_sysevent = 'CREATE' and ora_dict_obj_type = 'TABLE' and ora_is_creating_nested_table) then insert into event_tab values ('A nested table is created'); end if;

表 16-1 システムで定義されたイベント属性 （続き）

属性	型	説明	例
ora_is_drop_column(column_name IN VARCHAR2)	BOOLEAN	指定された列が削除された場合、true を戻します。	if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_drop_column('FOO'); end if;
ora_is_servererror	BOOLEAN	指定されたエラーがエラー・スタック上にある場合は true、ない場合は false を戻します。	IF (ora_is_servererror(*error_number*)) THEN INSERT INTO event_table ('Server error!!'); END IF;
ora_login_user	VARCHAR2(30)	ログイン・ユーザー名を戻します。	SELECT ora_login_user FROM dual;
ora_partition_pos	BINARY_INTEGER	CREATE TABLE の INSTEAD OF トリガーでは、PARTITION 句を挿入できる SQL テキスト内の位置を戻します。	-- Retrieve ora_sql_txt into -- sql_text variable first. n := ora_partition_pos; new_stmt := substr(sql_text, 1, n-1) || ' ' || my_partition_clause || ' ' || substr(sql_text, n));
ora_privilege_list(privilege_list OUT ora_name_list_t)	BINARY_INTEGER	権限受領者によって付与された権限のリストまたは取消し側から取り消された権限のリストを、OUT パラメータで戻します。戻り値として権限の数を戻します。	if (ora_sysevent = 'GRANT' or ora_sysevent = 'REVOKE') then number_of_privileges := ora_privilege_list(priv_list); end if;
ora_revokee (user_list OUT ora_name_list_t)	BINARY_INTEGER	OUT パラメータで取消しイベントの取消し側を戻します。戻り値として取消し側の数を戻します。	if (ora_sysevent = 'REVOKE') then number_of_users := ora_revokee(user_list);

表 16-1 システムで定義されたイベント属性 （続き）

属性	型	説明	例
ora_server_error	NUMBER	（スタックの一番上を 1 として）位置を指定すると、エラー・スタック上のその位置のエラー番号を戻します。	INSERT INTO event_table ('top stack error ' || ora_server_error(1));
ora_server_error_depth	BINARY_INTEGER	エラー・スタック上のエラー・メッセージの合計数を戻します。	n := ora_server_error_depth; -- This value is used with -- other functions such as -- ora_server_error
ora_server_error_msg (position in binary_integer)	VARCHAR2	（スタックの一番上を 1 として）位置を指定すると、エラー・スタック上のその位置のエラー・メッセージを戻します。	INSERT INTO event_table ('top stack error message' || ora_server_error_msg(1));
ora_server_error_num_params (position in binary_integer)	BINARY_INTEGER	（スタックの一番上を 1 として）位置を指定すると、「%s」などの書式を使用してエラー・メッセージ内で置き換えられた文字列数を戻します。	n := ora_server_error_num_params(1);
ora_server_error_param (position in binary_integer, param in binary_integer)	VARCHAR2	（スタックの一番上を 1 として）位置およびパラメータ番号を指定すると、「%s」、「%d」などに一致するエラー・メッセージ内の代入値を戻します。	-- E.g. the 2rd %s in a message -- like "Expected %s, found %s" param := ora_server_error_param(1,2);

表 16-1 システムで定義されたイベント属性 （続き）

属性	型	説明	例

ora_sql_txt (sql_text out ora_name_list_t)	BINARY_INTEGER	OUT パラメータ内のトリガーを実行する文の SQL テキストを戻します。この文が長い場合、複数の PL/SQL 表要素に分割されます。ファンクションの戻り値には、PL/SQL 表内の要素数が指定されます。	sql_text ora_name_list_t;
			stmt VARCHAR2(2000);
			...
			n := ora_sql_txt(sql_text);
			FOR i IN 1..n LOOP
			stmt := stmt || sql_text(i);
			END LOOP;
			INSERT INTO event_table ('text of triggering statement: ' || stmt);

ora_sysevent	VARCHAR2(20)	トリガーを起動するシステム・イベントを戻します。イベント名は、構文にある名前と同じです。	INSERT INTO event_table (ora_sysevent);

ora_with_grant_option	BOOLEAN	権限オプションとともに権限が付与された場合、true を戻します。	if (ora_sysevent = 'GRANT' and ora_with_grant_option = TRUE)
			then insert into event_table ('with grant option');
			end if;

space_error_info(
error_number OUT NUMBER,
error_type OUT VARCHAR2,
object_owner OUT VARCHAR2,
table_space_name OUT VARCHAR2,
object_name OUT VARCHAR2,
sub_object_name OUT VARCHAR2)
	BOOLEAN	エラーが領域不足状態に関連する場合 true を戻し、エラーの原因となったオブジェクトに関する情報を OUT パラメータで戻します。	if (space_error_info(eno, typ, owner, ts, obj, subobj) = TRUE)
			then
			dbms_output.put_line('The object ' || obj || ' owned by ' || owner || ' has run out of space.');
			end if;

データベース・イベントのリスト

システム・イベント

システム・イベントは、個々の表または行ではなく、インスタンスまたはスキーマ全体に関係します。起動イベントおよび停止イベントに対して作成されたトリガーは、データベース・インスタンスに対応付けられる必要があります。エラー・イベントおよび一時停止イベントに対して作成されたトリガーは、データベース・インスタンスか特定のスキーマのいずれかに対応付けられる必要があります。

表 16-2 に、システム・マネージャ・イベントのリストを示します。

表 16-2 システム・マネージャ・イベント

イベント	起動するタイミング	条件	制限事項	トランザクション	属性関数
STARTUP	データベースのオープン時	なし	トリガーではデータベース処理はできません。 戻り状態は無視されます。	トリガーの起動後、別のトランザクションを開始してこれをコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name
SHUTDOWN	サーバーがインスタンス停止を開始する直前 これによって、カートリッジを完全に停止できます。インスタンスの異常停止の場合は、このイベントは起動されない場合があります。	なし	トリガーではデータベース処理はできません。 戻り状態は無視されます。	トリガーの起動後、別のトランザクションを開始してこれをコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name
SERVERERROR	eno エラーの発生時条件が指定されない場合、このイベントは任意のエラーが発生したときに起動します。 ORA-01034、ORA-01403、ORA-01422、ORA-01423 および ORA-04030 の条件には適用されません。本当のエラーでないか、問題が深刻で処理を続行できないためです。	ERRNO = eno	エラーに依存します。 戻り状態は無視されます。	トリガーの起動後、別のトランザクションを開始してこれをコミットします。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info

クライアント・イベント

クライアント・イベントは、ユーザーのログイン / ログオフ、DML および DDL の操作に関係するイベントです。次に例を示します。

```
CREATE OR REPLACE TRIGGER On_Logon
  AFTER LOGON
  ON The_user.Schema
BEGIN
  Do_Something;
END;
```

LOGON イベントおよび LOGOFF イベントによって、UID( ) および USER( ) に単純な条件を使用できます。他のすべてのイベントによって、UID( ) や USER( ) のような関数のみでなく、オブジェクトの型および名前に単純な条件を使用できます。

LOGON イベントは、トリガーの起動後、別のトランザクションを開始してこれをコミットします。他のすべてのイベントは、既存のユーザー・トランザクションでトリガーを起動します。

LOGON イベントおよび LOGOFF イベントは、すべてのオブジェクトを操作できます。他のすべてのイベントでは、対応するトリガーは、そのイベントを生成させるオブジェクト上で DROP や ALTER などの DDL 操作を実行できません。

これらのトリガーで使用できる DDL は、表の変更、作成または削除、トリガーの作成および処理のコンパイルです。

イベント・トリガーが DDL 操作（CREATE TRIGGER など）の対象になる場合、このトリガーを、後で同じトランザクション中に起動することはできません。

表 16-3 に、クライアント・イベントのリストを示します。

表 16-3 クライアント・イベント

イベント	起動するタイミング	属性関数
BEFORE ALTER AFTER ALTER	カタログ・オブジェクトの変更時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column, ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	カタログ・オブジェクトの削除時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
BEFORE ANALYZE AFTER ANALYZE	分析文の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	関連情報の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	監査または非監査文の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name

表 16-3 クライアント・イベント （続き）

イベント	起動するタイミング	属性関数
BEFORE COMMENT AFTER COMMENT	オブジェクトへのコメントの追加時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	カタログ・オブジェクトの作成時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)
BEFORE DDL AFTER DDL	ほとんどの SQL DDL 文の発行時。アドバンスト・キューの作成などの PL/SQL プロシージャ・インタフェースを介して発行されたALTER DATABASE、CREATE CONTROLFILE、CREATE DATABASE および DDL に対しては起動されません。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	非関連情報の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list

表 16-3 クライアント・イベント （続き）

イベント	起動するタイミング	属性関数
BEFORE GRANT AFTER GRANT	権限付与文の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privileges
BEFORE LOGOFF	ユーザー・ログオフの開始時	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER LOGON	ユーザーが正常にログインした後	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	名前の変更文の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type

表 16-3 クライアント・イベント （続き）

イベント	起動するタイミング	属性関数
BEFORE REVOKE AFTER REVOKE	取消し文の発行時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privileges
AFTER SUSPEND	領域不足状態のため SQL 文が一時停止された後。トリガーは、文が再開されるようにこの状態を修正する必要があります。	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info
BEFORE TRUNCATE AFTER TRUNCATE	オブジェクトの切捨て時	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

17

アプリケーションに対するパブリッシュ・サブスクライブ・モデルの使用

データベースは企業内の最も重要な情報資源であるため、この役割を補完するために、オラクル社では、企業の情報配信およびメッセージ交換用にパブリッシュ・サブスクライブ・ソリューションを作成しました。内容は次のとおりです。

- パブリッシュ・サブスクライブの概要
- パブリッシュ・サブスクライブのアーキテクチャ
- パブリッシュ・サブスクライブの概念
- パブリッシュ・サブスクライブ・メカニズムの例

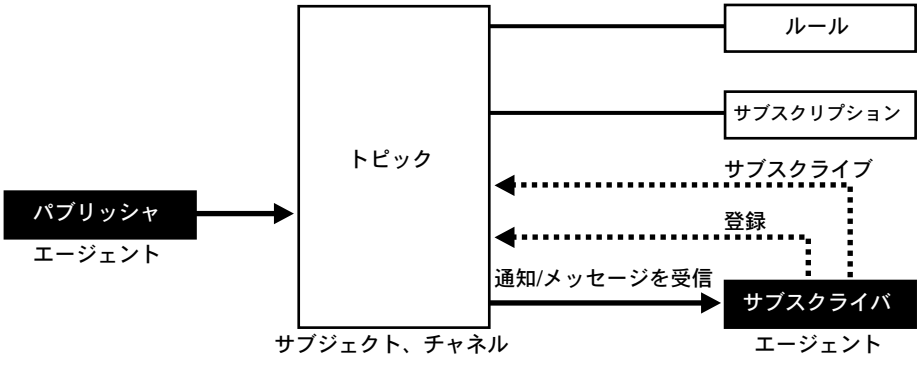
パブリッシュ・サブスクライブの概要

ネットワーキング・テクノロジおよびネットワーキング製品によって、現在では多数のコンピュータ、アプリケーションおよびユーザーにまたがる高度な接続が可能です。これらの環境では、疎結合で自律的に動作する分散システムに非同期通信を提供することが重要であり、ネットワーク障害に強い操作性が要求されます。この要件は、メッセージ機能、メッセージ指向ミドルウェア（MOM）、メッセージ・キューイング、パブリッシュ・サブスクライブなどの特長を持つ様々なミドルウェア製品によって満たされています。

パブリッシュ・サブスクライブ・パラダイムを介して通信するアプリケーションには、受信者を明示的に指定したり意図された受信者を知らせなくても、メッセージを発行できる送信アプリケーション（パブリッシャ）が必要です。同様に、受信アプリケーション（サブスクライバ）は、サブスクライバが登録しているメッセージのみを受信する必要があります。

この送信者と受信者の切離しは、通常、パブリッシャとサブスクライバの間に存在して間接的なレベルとして機能するエンティティによって実現されます。このエンティティが、サブジェクトまたはチャネルを表すキューです。図 17-1 に、パブリッシュおよびサブスクライブの機能を示します。

図 17-1 Oracle のパブリッシュ・サブスクライブ機能



サブスクライバは、あるキューにエンキューされたメッセージに関心を示すことによって、また、サブジェクト・ベースまたはコンテンツ・ベースのルールをフィルタとして使用することによって、キューをサブスクライブします。この結果、指定されたキューに一連のルールベースのサブスクリプションが対応付けられます。

実行時、パブリッシャは様々なキューにメッセージを転送します。次に、キュー（基礎となるインフラストラクチャの配信メカニズム）は、様々なサブスクリプションに一致したメッセージを該当するサブスクライバに配信します。

パブリッシュ・サブスクライブのアーキテクチャ

Oracle には、データベースに対応したパブリッシュ・サブスクライブのメッセージ機能をサポートする機能が含まれています。

- データベース・イベント
- アドバンスト・キューイング
- クライアント通知

データベース・イベント

データベース・イベントは、データベース・イベントを公開するための宣言定義、検出およびこれらのイベントの実行時の発行をサポートします。この機能によって、イベント駆動方式でエンドユーザーに情報を能動的に発行し、従来のプル型の情報アクセスの方法を補足することができます。

参照： 第 16 章「システム・イベントの処理」を参照してください。

アドバンスト・キューイング

Oracle アドバンスト・キューイング（AQ）では、キュー・ベースのパブリッシュ・サブスクライブ・パラダイムがサポートされます。データベース・キューは、メッセージの永続的な格納場所として機能し、キューを基にしたパブリッシュおよびサブスクライブが可能になります。ルール・エンジンおよびサブスクリプション・サービスが、設定された関心事項に基づいて、受信者にメッセージを動的に送付します。これによって、送信者と受信者の間のアドレス関係が切り離され、送信者と受信者間での明示的なメッセージ・アドレッシングを補足することができます。

参照：『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

クライアント通知

クライアント通知は、メッセージに関心を持つサブスクライバに対するメッセージの非同期配信をサポートします。これによって、データベース・クライアントは特定のキューに対する関心を登録し、そのようなキューで発行が発生した場合に通知を受け取ることができます。データベース・クライアントへのメッセージの非同期配信は、情報を取得するために使用される従来のポーリングとは対照的な技法です。

参照：『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

パブリッシュ・サブスクライブの概念

この項では、パブリッシュ・サブスクライブに関連する様々な概念を説明します。

キュー

キューは、名前付きの関心対象サブジェクトをサポートするエンティティです。キューには、次のような特長があります。

非永続キュー（軽量キュー）

基礎となるキュー・インフラストラクチャでは、公開されたメッセージを接続クライアントに対して最高で 1 回送信します。

永続キュー

キューは、メッセージの永続的なコンテナとして機能します。メッセージは、遅延付き高信頼モードで配信されます。

エージェント

パブリッシャおよびサブスクライバは、内部的にはエージェントとして表されます。エージェントとクライアントは区別されます。

エージェントは、サブスクリプションを介してキューに関心を示す持続的で論理的なサブスクライブ・エンティティです。エージェントには、関連するサブスクリプション、アドレス、メッセージの配信モードなどのプロパティがあります。この意味では、エージェントはパブリッシャまたはサブスクライバの電子的なプロキシです。

クライアントは、一時的な物理エンティティです。クライアントの属性には、クライアント・プログラムが実行される物理プロセス、ノード名、クライアント・アプリケーション・ロジックが含まれます。単一のエージェントにかわって複数のクライアントが動作する場合もあります。認証されている場合には、同一のクライアントが複数のエージェントにかわって動作することもできます。

キューでのルール

キューでのルールは、メッセージ形式属性またはメッセージ・ヘッダー属性に関する一連の事前定義済演算子を使用する条件式として指定されます。各キューには、そのキューが示すメッセージの構造を記述したメッセージ内容形式が対応付けられています。メッセージ形式は、構造化されていなくても（RAW）、正しく定義された構造体（ユーザー定義型）であってもかまいません。これによって、サブジェクト・ベースおよびコンテンツ・ベースの両方のサブスクリプションが可能です。

サブスクライバ

サブスクライバ（エージェント）は、ルールを使用してキューにサブスクリプションを指定できます。サブスクライバは永続的であり、カタログに格納されます。

データベース・イベント発行フレームワーク

データベースは、重要な情報公開元を表します。イベント・フレームワークは、データベース・イベント発行の宣言定義ができるように提案されたものです。これらの事前定義済イベントが発生すると、このフレームワークがイベントを検出および公開します。これによって、パブリッシュ・サブスクライブ機能の一部として、エンドユーザーに対してイベント駆動方式によるアクティブな情報配信が可能になります。

登録

登録は、エージェントにかわって動作する所定のクライアントによって対応付けられた配信情報のプロセスです。エージェントとクライアントの区別に関連して、サブスクリプションと登録の間には重要な区別があります。

サブスクリプションは、エージェントによる特定のキューへの関心を示します。配信の場所および方法は指定しません。配信情報は、クライアントに対応付けられた物理的なプロパティで、論理エージェント（サブスクライバ）の一時的な発現です。エージェントにかわって動作する特定のクライアント・プロセスは、配信を行う場所を示すホストとポート、および配信の方法を示すコールバックを対応付けることによって、配信情報を登録します。

メッセージの公開

パブリッシャは、適切なキューイング・インタフェースを使用して、キューにメッセージを公開します。インタフェースは、キューが実装されているモデルに依存することがあります。たとえば、エンキュー・コールは、メッセージの公開を表します。

ルール・エンジン

指定されたキューにメッセージが転送または公開されると、ルール・エンジンはそのキューに関して定義されたすべてのルールから、公開されたメッセージと一致する一連の候補となるルールを取得します。

サブスクリプション・サービス

指定されたキュー上の候補ルールのリストに応じて、候補ルールに一致する一連のサブスクライバを評価できます。次に、このサブスクリプション・リストに対応する一連のエージェントが決定され、通知されます。

ポスト

キューは、登録されたすべてのクライアントに対して、公開された該当メッセージを通知します。この概念は**ポスト**と呼ばれます。関心があるすべてのクライアントに通知が必要な場合、キューは、登録されたすべてのクライアントにメッセージをポストします。

メッセージの受信

サブスクライバは、次のメカニズムのいずれかを介して、メッセージを受信できます。

- サブスクライバにかわって動作するクライアント・プロセスが、登録メカニズムを使用してコールバックを指定します。その後、メッセージがサブスクライバのサブスクリプションと一致する場合、ポスト・メカニズムによってコールバックが非同期に起動され

ます。メッセージ・コンテンツは、コールバック関数に渡される場合があります（非永続キューの場合のみ）。

- サブスクライバにかわって動作するクライアント・プロセスが、登録メカニズムを使用してコールバックを指定します。その後、ポスト・メカニズムによってコールバック関数が非同期に起動されますが、完全なメッセージ・コンテンツは渡されません。コールバック関数はクライアントへの通知として機能し、これに続いてプル型の方法でメッセージ・コンテンツを取得します（永続的キューの場合のみ）。
- サブスクライバにかわって動作するクライアント・プロセスが、周期的またはその他の方法で適宜キューから単純にメッセージを取得します。メッセージの遅延がある場合、エンド・クライアントへの非同期配信は行われません。

パブリッシュ・サブスクライブ・メカニズムの例

注意： 次のようなデータ構造を設定しないと機能しない例もあります。

```
CONNECT system/manager
DROP USER pubsub CASCADE;
CREATE USER pubsub IDENTIFIED BY pubsub;
GRANT CONNECT, RESOURCE TO pubsub;
GRANT EXECUTE ON DBMS_AQ to pubsub;
GRANT EXECUTE ON DBMS_AQADM to pubsub;
GRANT AQ_ADMINISTRATOR_ROLE TO pubsub;
CONNECT pubsub/pubsub
```

使用例 : この例では、システム・イベント、クライアント通知、AQ がどのように連携してパブリッシュ・サブスクライブを実装するかを示します。

- ユーザー・スキーマの下で、パブリッシュ・サブスクライブ・メカニズムをサポートするために必要なすべてのオブジェクトを持つ pubsub を作成します。このコードでは、エージェント snoop は、ログイン・イベントで公開されるメッセージをサブスクライブします。ユーザー pubsub が AQ 機能を使用するには、AQ_ADMINISTRATOR_ROLE 権限が必要であることに注意してください。
-

```
Rem ------------------------------------------------------
REM create queue table for persistent multiple consumers:
Rem ------------------------------------------------------

CONNECT pubsub/pubsub;

Rem  Create or replace a queue table
BEGIN
DBMS_AQADM.CREATE_QUEUE_TABLE(
```

```
   Queue_table        =>  'Pubsub.Raw_msg_table',
   Multiple_consumers =>   TRUE,
   Queue_payload_type =>  'RAW',
   Compatible         =>  '8.1');
END;
/
Rem ------------------------------------------------------
Rem  Create a persistent queue for publishing messages:
Rem ------------------------------------------------------

Rem  Create a queue for logon events
begin
BEGIN
   DBMS_AQADM.CREATE_QUEUE(
         Queue_name     =>   'Pubsub.Logon',
         Queue_table    =>   'Pubsub.Raw_msg_table',
         Comment        =>   'Q for error triggers');
END;
/

Rem ------------------------------------------------------
Rem  Start the queue:
Rem ------------------------------------------------------

BEGIN
   DBMS_AQADM.START_QUEUE('pubsub.logon');
END;
/

Rem ------------------------------------------------------
Rem  define new_enqueue for convenience:
Rem ------------------------------------------------------

CREATE OR REPLACE PROCEDURE New_enqueue(
               Queue_name      IN VARCHAR2,
               Payload         IN RAW ,
               Correlation     IN VARCHAR2 := NULL,
               Exception_queue IN VARCHAR2 := NULL)
AS

Enq_ct     DBMS_AQ.Enqueue_options_t;
Msg_prop   DBMS_AQ.Message_properties_t;
Enq_msgid  RAW(16);
Userdata   RAW(1000);

BEGIN
   Msg_prop.Exception_queue := Exception_queue;
```

```
   Msg_prop.Correlation := Correlation;
   Userdata := Payload;

DBMS_AQ.ENQUEUE(Queue_name, Enq_ct, Msg_prop, Userdata, Enq_msgid);
END;
/

Rem ----------------------------------------------------
Rem  add subscriber with rule based on current user name,
Rem  using correlation_id
Rem ----------------------------------------------------


DECLARE
Subscriber Sys.Aq$_agent;
BEGIN
   Subscriber := sys.aq$_agent('SNOOP', NULL, NULL);
DBMS_AQADM.ADD_SUBSCRIBER(
    Queue_name         => 'Pubsub.logon',
    Subscriber         => subscriber,
    Rule               => 'CORRID = ''SCOTT'' ');
END;
/

Rem ----------------------------------------------------
Rem  create a trigger on logon on database:
Rem ----------------------------------------------------


Rem  create trigger on after logon:
CREATE OR REPLACE TRIGGER pubsub.Systrig2
   AFTER LOGON
   ON DATABASE
   BEGIN
      New_enqueue('Pubsub.Logon', HEXTORAW('9999'), Dbms_standard.login_user);
   END;
/
```

- サブスクリプションが作成された後、次の手順として、クライアントがコールバック関数を使用した通知を登録します。これには、OCI を使用します。次のコードは、登録に必要な手順を実行します。セッション・ハンドルの割当ておよび初期化を行う最初の手順は、例をわかりやすくするために、ここでは省略します。

```
ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

/* callback function for notification of logon of user 'scott' on database: */
```

```
ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4 payl;
dvoid *desc;
ub4 mode;
{
    printf("Notification : User Scott Logged on\n");
}

int main()
{
    OCISession *authp = (OCISession *) 0;
    OCISubscription *subscrhpSnoop = (OCISubscription *)0;

    /*****************************************************
       Initialize OCI Process/Environment
       Initialize Server Contexts
       Connect to Server
       Set Service Context
    *****************************************************/

    /* Registration Code Begins */

    /* Each call to initSubscriptionHn allocates
           and Initialises a Registration Handle */


    initSubscriptionHn(    &subscrhpSnoop,    /* subscription handle */
        "ADMIN:PUBSUB.SNOOP", /* subscription name */
                  /* <agent_name>:<queue_name> */
        (dvoid*)notifySnoop); /* callback function */

     /*****************************************************
       The Client Process does not need a live Session for Callbacks
       End Session and Detach from Server
     *****************************************************/

    OCISessionEnd ( svchp,  errhp, authp, (ub4) OCI_DEFAULT);

    /* detach from server */
    OCIServerDetach( srvhp, errhp, OCI_DEFAULT);

    while (1)     /* wait for callback */
        sleep(1);
```

```
}

void initSubscriptionHn (subscrhp,
subscriptionName,
func)

OCISubscription **subscrhp;
char* subscriptionName;
dvoid * func;
{

    /* allocate subscription handle: */

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

    /* set subscription name in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) subscriptionName,
        (ub4) strlen((char *)subscriptionName),
        (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

    /* set callback function in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) func, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) 0, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

    /* set namespace in handle: */

    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) &namespace, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

    checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,

        OCI_DEFAULT));
}
```

これで、ユーザー SCOTT がデータベースにログインすると、クライアントに通知され、コールバック関数 notifySnoop がコールされます。

第 V 部

特殊アプリケーションの開発

第 V 部では、一部の開発者が直面するアプリケーション開発の使用例について説明します。Oracle では、あらゆる種類のアプリケーションに有効な機能が、幅広く提供されます。

第 V 部に含まれる章は、次のとおりです。

- 第 18 章「PL/SQL を使用した Web アプリケーションの開発」
- 第 19 章「Oracle 以外のアプリケーションから Oracle9i への移植」
- 第 20 章「Oracle XA でのトランザクション・モニターの操作」

18

PL/SQL を使用した Web アプリケーションの開発

Java や Java スクリプトのような新しい言語のみが、ネットワーク操作を実現し、動的な Web コンテンツを生成できるわけではありません。PL/SQL には、データベースを Web 上で使用可能にし、業務上のデータをイントラネット・ユーザーまたは顧客に対して、対話的にアクセス可能なものにする多くの機能があります。

この章の内容は次のとおりです。

PL/SQL Web アプリケーションの概要

一般的に、PL/SQL で作成される Web アプリケーションは、HTTP プロトコルを介して Web ブラウザと対話するストアド・プロシージャの集合です。

- ハイパーテキスト・リンクをたどって Web ページにアクセスするか、HTML フォーム上の「Submit」ボタンを押すと、データベース・サーバーでストアド・プロシージャが実行されます。
- HTML フォームでユーザーが選択した項目は、パラメータとしてストアド・プロシージャに渡されます。パラメータは、ストアド・プロシージャの起動に使用する URL にハードコード化することもできます。
- ストアド・プロシージャの結果はタグ付き HTML テキストとして出力され、Web ページとしてブラウザに表示されます。この方法で生成された Web ページは、**動的**になります。動的な Web ページでは、コードがデータベース・サーバー内部で実行され、データベースの内容および入力パラメータによって変化する HTML が作成されます。

  このような種類の動的コンテンツは**動的 HTML**（DHTML）とは異なります。DHTML の場合、コードは Java スクリプトなどのスクリプト言語としてダウンロードされ、HTML とともにブラウザで処理されます。PL/SQL Web アプリケーションは、出力内に Java スクリプトなどのスクリプト・コードを出力して、手動で生成するには長すぎる複雑な DHTML を生成できます。
- 動的ページに、より多くのストアド・プロシージャをコールするリンクおよび HTML フォームを含めて、表示データをドリルダウンしたり、他の操作を実行することができます。相互リンクされた一連の HTML ページが、Web アプリケーションのユーザー・インタフェースになります。
- 動的ページのコーディングには多くの方法がありますが、データベース処理に基づいて動的ページを生成する場合は、PL/SQL が特に適しています。DML 文、動的 SQL、カーソルおよびサーバーの緊密な統合のサポートによって、強力で柔軟な Web アプリケーションが作成できます。
- PL/SQL ストアド・プロシージャを使用して動的コンテンツを生成すると、新しい CGI プロセスを指定するたびにメモリーのオーバーヘッドを発生させることなく、CGI プログラムの柔軟かつインタラクティブな動作を使用できます。典型的なストアド・プロシージャは、一部のヘッダー情報の出力、問合せの発行および結果セットのループによる HTML リストまたは表内のデータのフォーマットを実行します。このプログラムの流れは、テキスト・ファイルで処理される Perl スクリプトに類似しています。

PL/SQL からの HTML 出力の生成方法

従来、PL/SQL Web アプリケーションでは、PL/SQL Web Toolkit パッケージを使用して、ファンクション・コールによって出力用の各 HTML タグを生成していました。これらのパッケージは、Oracle9*i* Application Server（*i*AS）、Oracle Application Server（OAS）および WebDB に付属しています。

```
owa_util.mime_header('text/html');

htp.htmlOpen;
htp.headOpen;
htp.title('Title of the HTML File');
htp.headClose;

htp.bodyOpen( cattributes => 'TEXT="#000000" BGCOLOR="#FFFFFF"');
htp.header(1, 'Heading in the HTML File');
htp.para;
htp.print('Some text in the HTML file.');
htp.bodyClose;

htp.htmlClose;
```

各タグに対応する Application Program Interface（API）コールを覚えるか、または HTP.PRINT などの基本的なファンクション・コールを使用して、テキストとタグをまとめて出力できます。

```
htp.print('<html>');
htp.print('<head>');
htp.print('<meta http-equiv="Content-Type" content="text/html">');
htp.print('<title>Title of the HTML File</title>');
htp.print('</head>');

htp.print('<body TEXT="#000000" BGCOLOR="#FFFFFF">');
htp.print('<h1>Heading in the HTML File</h1>');
htp.print('<p>Some text in the HTML file.');
htp.print('</body>');

htp.print('</html>');
```

この章では、新しい方法の PL/SQL Server Pages（PSP）について説明します。この方法では、一連のファンクション・コールを新しく覚えるのではなく、既知の HTML タグに基づいて出力を生成できます。

PL/SQL Server Pages の集合として作成されたアプリケーションでも、PL/SQL Web Toolkit のファンクションを使用して、表の表示、永続データ（Cookie）の格納および CGI プロトコル内部の作業に関連する処理を簡単にできます。

PL/SQL Web アプリケーションへのパラメータ渡し

Web アプリケーションを様々な状況で有効に活用するには、対話性を十分に高め、ユーザーによる選択を可能にする必要があります。簡単な対話で、決定事項またはデータ入力を少なくし、ユーザーが選択項目を簡単に指定できるようにする必要があります。

PL/SQL Web アプリケーションにパラメータを渡す主な方法は、次のとおりです。

- HTML フォーム・タグを使用します。ユーザーが 1 つの Web ページ上のフォームに記入して、「Submit」ボタンをクリックすると、すべてのデータおよび選択項目がストアド・プロシージャに転送されます。
- URL にハードコード化します。ユーザーがリンクをクリックすると、一連の事前定義パラメータがストアド・プロシージャに転送されます。通常、ユーザーが選択する可能性があるすべての選択項目には、Web ページ上に個別のリンクを挿入します。

HTML フォームからのリスト・パラメータおよびドロップダウン・リスト・パラメータ渡し

リスト・ボックスおよびドロップダウン・リストは、同じ HTML タグ（<SELECT>）を使用して実装されます。

選択項目の数が多く、ユーザーがすべての項目を表示するためにスクロールする必要がある場合、または複数の選択を可能にする必要がある場合は、リスト・ボックスを使用します。リスト・ボックスは、項目をアルファベット順に表示する場合に適しています。これによって、ユーザーがすべての項目を読まなくても目的の項目を短時間で検索できます。

選択項目の数が少ない場合、画面のスペースがかぎられている場合、または選択項目の順序が不規則な場合は、ドロップダウン・リストを使用します。ドロップダウン・リストは、初めてアクセスするユーザーの注意を引き、すべての選択項目を参照させることができます。選択項目および順序を固定することで、ユーザーはドロップダウン・リストから項目を選択するときの動作に慣れ、素早く選択できるようになります。

HTML フォームからのラジオ・ボタン・パラメータおよびチェックボックス・パラメータ渡し

ラジオ・ボタンによって、NULL 値（グループ内のいずれのラジオ・ボタンもチェックされていない場合）、またはチェックされたラジオ・ボタンに指定されている値のいずれかが渡されます。

一連のラジオ・ボタンのデフォルト値を指定するには、INPUT タグの 1 つに CHECKED 属性を含めるか、ストアド・プロシージャ内のパラメータに DEFAULT 句を含めます。ラジオ・ボタンのグループを設定するときは、「どちらでもない」ことを示す選択項目を含める必要があります。これは、ラジオ・ボタンを一度選択すると、別のボタンを選択することはできますが、選択を完全に取り消すことができないためです。たとえば、あるユーザーが選択を行った後に、それが間違っていたことに気付くという状況を考慮して、「はい」および「いいえ」とともに「興味なし」または「わからない」という選択項目を含めます。

チェックボックスでは、ストアド・プロシージャが NULL 値、単一の値、または複数の値を受け取る場合があるため、特別な処理が必要です。

同じ NAME 属性を持つすべてのチェックボックスによって、1 つのチェックボックス・グループが形成されます。グループ内のいずれのチェックボックスもチェックされていない場合、ストアド・プロシージャは、対応するパラメータに対して NULL 値を受け取ります。

グループ内のチェックボックスの 1 つがチェックされている場合、ストアド・プロシージャは単一の VARCHAR2 パラメータを受け取ります。

グループ内の複数のチェックボックスがチェックされている場合、ストアド・プロシージャは PL/SQL の TABLE OF VARCHAR2 データ型のパラメータを受け取ります。類似のデータ型を宣言するか、OWA_UTIL.IDENT_ARR などの事前定義のデータ型を使用する必要があります。値を検索するには、次のようにループを使用します。

```
CREATE OR REPLACE PROCEDURE handle_checkboxes ( checkboxes owa_util.ident_arr )
AS
BEGIN
  ...
  FOR i IN 1..checkboxes.count
  LOOP
    htp.print('<p>Checkbox value: ' || checkboxes(i));
  END LOOP;
  ...
END;
/
show errors;
```

HTML フォームからの入力フィールド・パラメータ渡し

入力フィールドでは、ユーザーが間違った形式のデータや範囲外のデータなどを入力する可能性があるため、最も厳密な妥当性チェックが必要です。可能な場合、動的 HTML または Java を使用してクライアント側でデータの妥当性チェックを行い、ユーザーにかわって正しいフォーマットにするか、ユーザーに再入力を促します。

次に例を示します。

- ユーザーが数値入力フィールドにアルファベット文字を入力したり、文字数制限を超えた後に文字を入力することができないようにします。
- クレジット・カード番号から空白およびハイフンを暗黙的に削除します（ストアド・プロシージャでこの形式の値が要求されている場合）。
- ユーザーが最大値を超える数値を入力した場合、すぐに通知して、ユーザーに再入力を促します。

これらの妥当性チェックが常に成功するとはかぎらないため、このような状況に対応できるようにストアド・プロシージャをコーディングします。ユーザーが間違ったデータを入力したときに「Back」ボタンを押さなくて済むように、エラー・メッセージと他のすべての値が記入された元のフォームを、1 つのページに表示します。

パスワードなどの機密性の高い情報の場合、入力フィールドの <INPUT TYPE=PASSWORD> という特殊なフォームによって、入力されたテキストが非表示になります。

たとえば、次のプロシージャは、入力として 2 つの文字列を受け入れます。このプロシージャが初めてコールされた場合、値を入力するための単純なフォームのプロンプトが表示されます。ユーザーが情報を送ると、同じプロシージャが再コールされ、入力が正しいかどう

かが確認されます。入力が正しい場合、プロシージャは入力を処理します。正しくない場合、プロシージャは、再入力するためのプロンプトを、最初の値が入力された状態で表示します。

```
-- Store a name and associated zip code in the database.
CREATE OR REPLACE PROCEDURE associate_name_with_zipcode
(
  name VARCHAR2 DEFAULT NULL,
  zip VARCHAR2 DEFAULT NULL
)
AS
  booktitle VARCHAR2(256);
BEGIN
-- Both entry fields must contain a value. The zip code must be 6 characters.
-- (In a real program you would perform more extensive checking.)
  IF name IS NOT NULL AND zip IS NOT NULL AND length(zip) = 6 THEN
    store_name_and_zipcode(name, zip);
    htp.print('<p>The person ' || name || ' has the zip code ' || zip || '.');
-- If the input was OK, we stop here and the user does not see the form again.
    RETURN;
  END IF;

-- If some data was entered, but it is not correct, show the error message.
  IF (name IS NULL AND zip IS NOT NULL)
    OR (name IS NOT NULL AND zip IS NULL)
    OR (zip IS NOT NULL AND length(zip) != 6)
  THEN
    htp.print('<p><b>Please re-enter the data. Fill in all fields, and use a 6-digit
zip code.</b>');
  END IF;

-- If the user has not entered any data, or entered bad data, prompt for
-- input values.

-- Make the form call the same procedure to check the input values.
  htp.formOpen( 'scott.associate_name_with_zipcode', 'GET');
  htp.print('<p>Enter your name:</td>');
  htp.print('<td valign=center><input type=text name=name value="' || name || '">');
  htp.print('<p>Enter your zip code:</td>');
  htp.print('<td valign=center><input type=text name=zip value="' || zip || '">');
  htp.formSubmit(NULL, 'Submit');
  htp.formClose;
END;
/
show errors;
```

HTML フォームからの非表示パラメータ渡し

ユーザーが同じ選択項目を毎回指定しなくても、一連のストアド・プロシージャを介して情報を渡すには、ストアド・プロシージャをコールするフォームに非表示パラメータを含める方法があります。最初のストアド・プロシージャによって、このストアド・プロシージャで生成された HTML フォームにユーザー名などの情報が入力されます。非表示パラメータの値は、ユーザーがその値をラジオ・ボタンまたは入力フィールドを介して入力した場合と同様に、次のストアド・プロシージャに渡されます。

1 つのストアド・プロシージャから別のプロシージャに情報を渡すには、次の方法もあります。

- 永続情報を含む「Cookie」をブラウザに送信します。ブラウザは、同じサイトから他の Web ページにアクセスするときに、同じ情報をサーバーに戻します。Cookie は、各 Web ページの HTML テキストの前に、ブラウザと Web サーバーの間で転送される HTTP ヘッダーを介して設定され、検索されます。
- データベース自体に情報を格納し、後でストアド・プロシージャが検索できるようにします。この方法ではデータベース・サーバーに余分なオーバーヘッドが発生し、複数のユーザーがサーバーに同時アクセスしたときに、各ユーザーを追跡する方法を見つける必要があります。

記入済 HTML フォームの送信

デフォルトでは、HTML フォームに「Submit」ボタンが必要です。このボタンを押すと、データが HTML フォームからストアド・プロシージャまたは CGI プログラムに転送されます。このボタンには「Search」、「Register」など、任意の名前を指定できます。

1 つのページに複数のフォームを挿入し、各フォームに独自のフォーム要素および「Submit」ボタンを設定できます。非表示パラメータのみで構成されるフォームも作成できます。このフォームでは、ユーザーはボタンをクリックする以外の選択は行いません。

Java スクリプトなどのスクリプト言語を使用すると、「Submit」ボタンを排除して、ドロップダウン・リストからの選択など、別のアクションに応答してフォームが送られるようにすることができます。この方法は、ユーザーが選択する項目が 1 つのみで、「Submit」ボタンで確認する手順が必要でない場合に最適です。

HTML フォームの欠落した入力の処理

HTML フォームが送られると、ストアド・プロシージャは、記入されていないフォーム要素に対して NULL パラメータを受け取ります。NULL パラメータの原因には、入力フィールドが空である、一連のチェックボックス、ラジオ・ボタンまたはリスト項目がチェックされていない、または VALUE パラメータの値が ""（空の引用符）である、などがあります。

クライアント側で実行する妥当性チェックの内容にかかわらず、常にストアド・プロシージャをコーディングして、いくつかのパラメータが NULL になる状況に対処します。

- すべてのパラメータ宣言に DEFAULT 句を使用して、フォーム・パラメータが欠落した状態でストアド・プロシージャがコールされたときに発生する例外を回避します。数値

に対するデフォルトは 0（ゼロ）に設定できます（適切な場合）。ユーザーが実際に値を指定したかどうか確認するには、DEFAULT NULL を使用します。

- DEFAULT NULL が宣言された入力パラメータ値を使用する前に、その値が NULL かどうかを確認します。
- 一部の入力パラメータが指定されていないときでも、プロシージャが有効な結果を生成するようにします。レポートから一部のセクションを除外するか、パラメータが指定されていない場所を示すテキスト文字列またはイメージをレポート内に表示します。
- 欠落している値を結果ページから直接記入して、ストアド・プロシージャを再実行する方法を提供します。たとえば、追加パラメータを指定して同じストアド・プロシージャをコールするリンクを含めたり、元のフォームに出力の一部として値を挿入して表示させることができます。

Web ページ間での状態情報の保持

Web アプリケーションでは、**状態**（特定の時点における現行のデータ・セット）の概念が特に重要です。Web ページを切り替えると状態情報が簡単に失われ、ユーザーに何度も同じ選択をさせることになる場合があります。

状態情報は、HTML フォームを使用して動的 Web ページ間で渡されます。情報は一連の名前値の組として渡され、ストアド・プロシージャのパラメータになります。

ユーザーが複数の選択をする必要がある場合、多くの選択項目から 1 つを選択する必要がある場合、または偶然に誤って選択することを回避することが重要な場合は、HTML フォームを使用します。ユーザーは、すべての選択を行って、選択した項目を再確認すると、「Submit」ボタンを押して選択を決定します。それ以降のページでは、非表示パラメータ（<INPUT TYPE=HIDDEN> タグ）を含むフォームを使用して、これらの選択項目をページからページへと渡すことができます。

ユーザーが 1 つまたは 2 つの選択項目のみに関心がある場合、または選択事項が Web ページ内に散在している場合、アクションをハイパーリンクとして表し、必要な名前値の組を問合せ文字列（URL 内の ? の後に続く部分）に含めることによって、ユーザーが「Submit」ボタンを簡単に探せるようになります。

状態情報は、『Oracle Servlet Engine User's Guide』に記載されている Oracle9*i* Application Server およびその mod_ose モジュールを使用して保持することもできます。この方法では、状態情報をパッケージ変数に格納して、ユーザーが Web サイトを移動しても、情報を使用可能のままにできます。

PL/SQL ストアド・プロシージャでのネットワーク操作の実行

PL/SQL の組込み機能が、従来のデータベース処理およびプログラミング論理に重点を置く一方、Oracle は、PL/SQL プログラマに対してインターネット・コンピューティングの道を開くパッケージを提供しています。

PL/SQL からの電子メールの送信

PL/SQL プログラムまたはストアド・プロシージャから電子メールを送信するには、UTL_SMTP パッケージを使用します。このパッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

次のコード例は、アプリケーションが SMTP パッケージを使用して電子メールを送信する方法を示します。このアプリケーションは、ポート 25 で SMTP サーバーに接続し、単純なテキスト・メッセージを送信します。

```
PROCEDURE send_test_message
IS
    mailhost    VARCHAR2(64) := 'mailhost.fictional-domain.com';
    sender      VARCHAR2(64) := 'me@fictional-domain.com';
    recipient   VARCHAR2(64) := 'you@fictional-domain.com';
    mail_conn  utl_smtp.connection;
BEGIN
    mail_conn := utl_smtp.open_connection(mailhost, 25);
    utl_smtp.helo(mail_conn, mailhost);
    utl_smtp.mail(mail_conn, sender);
    utl_smtp.rcpt(mail_conn, recipient);
-- If we had the message in a single string, we could collapse
-- open_data(), write_data(), and close_data() into a single call to data().
    utl_smtp.open_data(mail_conn);
    utl_smtp.write_data(mail_conn, 'This is a test message.' || chr(13));
    utl_smtp.write_data(mail_conn, 'This is line 2.' || chr(13));
    utl_smtp.close_data(mail_conn);
    utl_smtp.quit(mail_conn);
    EXCEPTION
        WHEN OTHERS THEN
           -- Insert error-handling code here
           NULL;
END;
```

PL/SQL からのホスト名またはアドレスの取得

PL/SQL プログラムまたはストアド・プロシージャから、ローカル・マシンのホスト名または特定のホスト名の IP アドレスを判断するには、UTL_INADDR パッケージを使用します。このパッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。UTL_TCP パッケージへのコール結果を使用します。

PL/SQL からの TCP/IP 接続を使用した処理

ネットワーク上のマシンに TCP/IP 接続をオープンして、対応するソケットへの読込みまたは書込みを行うには、UTL_TCP パッケージを使用します。このパッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

PL/SQL からの HTTP URL のコンテンツの取得

HTTP URL のコンテンツを取得するには、UTL_HTTP パッケージを使用します。通常、コンテンツは HTML のタグ付きテキスト形式ですが、プレーン・テキスト、JPEG イメージなど、Web サーバーからダウンロードできるファイルであればどのようなものでもかまいません。このパッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

UTL_HTTP パッケージを使用して、次の操作を実行できます。

- HTTP セッションの詳細（ヘッダー行、Cookie、リダイレクト、プロキシ・サーバー、保護付きサイト用の ID およびパスワード、GET メソッドまたは POST メソッドを介した CGI パラメータなど）の制御
- HTTP 1.1 永続接続を使用した同じ Web サイトへの同時アクセスの高速化
- UTL_URL パッケージの ESCAPE ファンクションおよび UNESCAPE ファンクションを介した、UTL_HTTP に使用する URL の作成および解析

通常、開発者は Java または Perl を使用してこれらの操作を行いますが、このパッケージでは PL/SQL で同じ操作が実行できます。

```
CREATE OR REPLACE PROCEDURE show_url
(
    url      IN VARCHAR2,
    username IN VARCHAR2 DEFAULT NULL,
    password IN VARCHAR2 DEFAULT NULL
) AS
    req       utl_http.req;
    resp      utl_http.resp;
    name      VARCHAR2(256);
    value     VARCHAR2(1024);
    data      VARCHAR2(255);
    my_scheme VARCHAR2(256);
```

```
    my_realm  VARCHAR2(256);
    my_proxy  BOOLEAN;
BEGIN
-- When going through a firewall, pass requests through this host.
-- Specify sites inside the firewall that don't need the proxy host.
  utl_http.set_proxy('proxy.my-company.com', 'corp.my-company.com');

-- Ask UTL_HTTP not to raise an exception for 4xx and 5xx status codes,
-- rather than just returning the text of the error page.
  utl_http.set_response_error_check(FALSE);

-- Begin retrieving this web page.
  req := utl_http.begin_request(url);

-- Identify ourselves. Some sites serve special pages for particular browsers.
  utl_http.set_header(req, 'User-Agent', 'Mozilla/4.0');

-- Specify a user ID and password for pages that require them.
  IF (username IS NOT NULL) THEN
    utl_http.set_authentication(req, username, password);
  END IF;

  BEGIN
-- Now start receiving the HTML text.
    resp := utl_http.get_response(req);

-- Show the status codes and reason phrase of the response.
    dbms_output.put_line('HTTP response status code: ' || resp.status_code);
    dbms_output.put_line('HTTP response reason phrase: ' || resp.reason_phrase);

-- Look for client-side error and report it.
    IF (resp.status_code >= 400) AND (resp.status_code <= 499) THEN

-- Detect whether the page is password protected, and we didn't supply
-- the right authorization.
      IF (resp.status_code = utl_http.HTTP_UNAUTHORIZED) THEN
        utl_http.get_authentication(resp, my_scheme, my_realm, my_proxy);
        IF (my_proxy) THEN
          dbms_output.put_line('Web proxy server is protected.');
          dbms_output.put('Please supply the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the proxy server.');
        ELSE
          dbms_output.put_line('Web page ' || url || ' is protected.');
          dbms_output.put('Please supplied the required ' || my_scheme ||
            ' authentication username/password for realm ' || my_realm ||
            ' for the Web page.');
```

```
        END IF;
      ELSE
        dbms_output.put_line('Check the URL.');
      END IF;

      utl_http.end_response(resp);
      RETURN;

-- Look for server-side error and report it.
    ELSIF (resp.status_code >= 500) AND (resp.status_code <= 599) THEN

      dbms_output.put_line('Check if the Web site is up.');
      utl_http.end_response(resp);
      RETURN;

    END IF;

-- The HTTP header lines contain information about cookies, character sets,
-- and other data that client and server can use to customize each session.
    FOR i IN 1..utl_http.get_header_count(resp) LOOP
      utl_http.get_header(resp, i, name, value);
      dbms_output.put_line(name || ': ' || value);
    END LOOP;

-- Keep reading lines until no more are left and an exception is raised.
    LOOP
      utl_http.read_line(resp, value);
      dbms_output.put_line(value);
    END LOOP;
  EXCEPTION
    WHEN utl_http.end_of_body THEN
    utl_http.end_response(resp);
  END;

END;
/
SET serveroutput ON
-- The following URLs illustrate the use of this procedure,
-- but these pages do not actually exist. To test, substitute
-- URLs from your own web server.
exec show_url('http://www.oracle.com/no-such-page.html')
exec show_url('http://www.oracle.com/protected-page.html')
exec show_url('http://www.oracle.com/protected-page.html', 'scott', 'tiger')
```

PL/SQL からの表、イメージ・マップ、Cookie、CGI などを使用した処理

これらのすべての機能のパッケージは、Oracle8*i* 以上で提供されています。これらのパッケージは、Oracle Internet Application Server（iAS）や WebDB などの PL/SQL Gateway と組み合せて使用できます。問合せの結果を HTML 表にフォーマットしたり、イメージ・マップを作成することができます。また、HTTP Cookie の設定および取得を行ったり、CGI 変数の値を確認することもできます。また、PL/SQL プログラムを使用して、その他の一般的な Web 操作を組み合せることもできます。

これらのパッケージのドキュメントは、データベース・ドキュメント・ライブラリの一部ではありません。ドキュメントの位置は、実行中の特定のアプリケーションによって異なります。これらのパッケージを使用するには、次に示す SQL*Plus の `DESCRIBE` コマンドを使用して、プロシージャ名およびパラメータを確認します。

```
DESCRIBE HTP;
DESCRIBE HTF;
DESCRIBE OWA_UTIL;
```

Web ページへの PL/SQL コードの埋込み（PL/SQL Server Pages）

Web ページの内容に、SQL 問合せの結果を含めて動的な内容を含めるには、PL/SQL Server Pages（PSP）を介してサーバー側のスクリプトを使用します。スクリプトを利用する HTML オーサリング・ツールを使用して Web ページを作成し、個々の PL/SQL コードを適切な場所に配置できます。最先端の Web ページを作成するには、この方法の方が、HTP パッケージおよび HTF パッケージを使用して HTML コンテンツを 1 行ずつ書いていくよりはるかに有効です。

処理はサーバー上（この場合は、Web サーバーというよりデータベース・サーバー）で行われるため、ブラウザは特別なスクリプト・タグを使用しないプレーンな HTML ページを受け取り、すべてのブラウザおよびブラウザのレベルを同等にサポートできます。また、サーバーのラウンドトリップ数を最小限に抑えられるため、ネットワーク通信を効率化します。

作成した HTML ページに PL/SQL コードを埋め込むと、内容を素早く書き込むことができ、開発プロセスが迅速で相互的なものになります。クライアント・マシン上に Web ブラウザがあれば、ソフトウェアを集中制御できます。

PL/SQL Server Pages を使用した Web ベースのソリューションを実装するには、次の手順に従います。

- ソフトウェア構成の選択
- PL/SQL Server Pages へのコードおよびコンテンツの書込み
- ストアド・プロシージャとしての PL/SQL Server Pages のデータベースへのロード

ソフトウェア構成の選択

PL/SQL Server Pages（PSP）を開発および配置するには、PL/SQL Web ゲートウェイの他に、Oracle8*i* リリース 8.1.6 以上のサーバーが必要です。現在、Web ゲートウェイは、Oracle Internet Application Server（*i*AS）、WebDB PL/SQL Gateway および OAS PL/SQL Cartridge です。PSP を実行する前に、これらのゲートウェイのいずれかのデータベース・サーバーおよび Web サーバーにアクセスする必要があります。

PSP または PL/SQL Web Toolkit の選択

次のどちらの方法を使用しても、結果は同じです。

- 埋込み PL/SQL コードを使用した HTML ページを作成し、それを PL/SQL Server Pages としてコンパイルします。PL/SQL Web Toolkit からプロシージャをコールすることもできますが、HTML 出力の各行を生成することはできません。
- PL/SQL Web Toolkit にある `HTP` および `OWA_*` パッケージをコールすることによって、HTML を生成する完全なストアド・プロシージャを作成します。

これらの方法を選択する際のキーは、次のとおりです。

- 開始点として使用するソースの内容
  - HTML の本体が大きく、かつ動的な内容を含める場合、またはデータベース・アプリケーションのフロントエンドとする場合には、PSP を使用します。
  - フォーマットされた出力を生成する PL/SQL コードの本体が大きい場合は、出力文を変更して PL/SQL Web Toolkit の HTP パッケージをコールし、HTML タグを作成する方が有効です。
- 自分のグループにとって最も迅速で便利なオーサリング環境
  - ほとんどの作業が HTML オーサリング・ツールを使用する場合は、PSP を使用します。
  - WebDB のページ作成ウィザードのような、PL/SQL コードを作成するオーサリング・ツールを使用する場合は、PSP を使用すると不便な場合があります。

PSP とその他のスクリプティング・ソリューションとの関連

PL/SQL Server Pages（PSP）を使用すると、すべてのタグが変更されずにブラウザに渡されるため、PL/SQL Server Pages（PSP）には、Java スクリプトまたはその他のクライアント側のスクリプト・コードを含めることができます。

PSP に、サーバー側にあるその他のサーバー側スクリプト機能を混在させることはできません。多くの場合、対応する PSP 機能を使用することによって、同じ結果を得ることができます。

PSP では、JavaServer Pages（JSP）と同じスクリプト・タグ構文を使用しているため、双方の切替えは簡単です。

PSP では、Active Server Pages（ASP）と類似した構文を使用していますが、同一ではないため、通常、VBScript または Jscript から PL/SQL に変換する必要があります。移行に最適なのは、Active Data Object（ADO）を使用してデータベース処理を行うページです。

PL/SQL Server Pages へのコードおよびコンテンツの書込み

既存の Web ページまたは既存のストアド・プロシージャを使用できます。どちらの方法でも、わずかな追加と変更のみで、データベース処理を実行し、結果を表示する動的な Web ページを作成できます。

PSP ファイルの書式

PL/SQL Server Pages のファイルの拡張子は、`.psp` である必要があります。

PL/SQL Server Pages には、PSP ディレクティブ、宣言およびスクリプトレットとともにテキストまたはタグを配置することによって、どのような内容でも含めることができます。

- 最も簡単な例では、HTML ファイルのみを使用します。HTML ファイルを PL/SQL Server Pages としてコンパイルすると、同じ HTML ファイルを出力するストアド・プロシージャが作成されます。
- 最も複雑な例では、PL/SQL プロシージャを使用します。PL/SQL プロシージャを使用して、タイトル、本体およびヘッダーのタグを含む Web ページのコンテンツ全体を生成します。
- 一般的な例では、HTML（ページの静的な部分）と PL/SQL（動的コンテンツを追加）の両方を使用します。

別のファイルが挿入される場合以外は、PL/SQL Server Pages ディレクティブおよび宣言の順序や位置は、重要ではありません。メンテナンスを簡単にするために、ディレクティブおよび宣言は、ともにファイルの最初の方に配置することをお薦めします。

次の項では、PSP スクリプティング要素を使用して様々な結果を生成する方法について説明します。すでに動的 HTML について理解し、コードの作成をすぐに開始できる場合は、18-20 ページの「PL/SQL Server Pages 要素の構文」および 18-24 ページの「PL/SQL Server Pages の例」に進んでください。

スクリプト言語の指定

ファイルを PL/SQL Server Pages として識別するには、ファイルの任意の場所に `<%@ page language= "PL/SQL" %>` ディレクティブを含めます。このディレクティブは、他のスクリプト環境との互換性のためのものです。

ユーザー入力のアクセプト

ユーザー入力は、HTML ページを取得する URL にエンコード化されています。URL を生成するには、ユーザー入力を HTML リンクでハードコード化するか、または HTML 形式のア

クションとしてページをコールします。これによって、ページは、PL/SQL ストアド・プロシージャへのパラメータとして入力を受け取ります。

PL/SQL Server Pages へのパラメータ渡しを設定するには、<%@ plsql parameter="*varname*" %> ディレクティブを含めます。デフォルトでは、パラメータは VARCHAR2 型です。異なる型を使用するには、type="*typename*" 属性をディレクティブに含めます。パラメータがオプションになるようにデフォルト値を設定するには、default="*expression*" 属性をディレクティブに含めます。この属性の値は、直接 PL/SQL 文に置き換えられます。そのため、すべての文字列を一重引用符で囲む必要があります。また、NULL などの特別な値を使用することもできます。

HTML の表示

ページの PL/SQL 部分は、特別なデリミタで囲まれています。他のすべての内容は（空白を含めて）、ブラウザに逐語的に渡されます。テキストまたは HTML タグを表示するには、通常の Web ページと同じように書き込みます。出力ファンクションをコールする必要はありません。

条件によって、出力を 1 行ずつ表示したり属性の値を変更する必要がある場合があります。後の項で説明するとおり、PSP デリミタに IF/THEN ロジックおよび置換変数を含めます。

XML、テキストまたは他のドキュメント・タイプの戻し方

デフォルトでは、PL/SQL Gateway はファイルを HTML ドキュメントとして送信します。そのため、ブラウザはそれらを HTML タグに基づいてフォーマットします。ブラウザに、ドキュメントを XML、プレーン・テキスト（書式なし）またはその他のドキュメント・タイプとして解析させるには、<%@ page contentType="*MIMEtype*" %> ディレクティブを含めます（属性名は大 / 小文字が区別されます。contentType のように、正確に大文字を使用してください）。text/html、text/xml、text/plain、image/jpeg、またはブラウザやその他のクライアント・プログラムが認識できるその他の MIME タイプを指定します。MIME タイプの中には、ユーザーがブラウザの設定を変更しないと認識されないものもあります。

通常、PL/SQL Server Pages は、Web ブラウザで表示されるように意図されています。また、Java や Perl アプリケーションなど、HTTP 要求を作成可能なプログラムによって取得または解析されることもあります。

別のキャラクタ・セットを含むページの戻し方

デフォルトでは、PL/SQL Gateway は、Web ゲートウェイによって定義されたキャラクタ・セットを使用してファイルを送信します。ブラウザに表示するためにデータを別のキャラクタ・セットに変換するには、<%@ page charset="*encoding*" %> ディレクティブを含めます。Shift_JIS、Big5、UTF-8、またはブラウザやその他のクライアント・プログラムが認識できるその他のエンコーディングを指定します。

Web ゲートウェイのデータベース・アクセサ記述子（DAD）に設定するキャラクタ・セットも構成する必要があります。データが適切に表示されるようにするには、ユーザーが使用するブラウザで同じエンコーディングを選択する必要がある場合があります。

たとえば、EUC エンコーディングを使用するデータベース・キャラクタ・セットが日本のデータベースに含まれていることがありますが、Web ブラウザは、Shift_JIS エンコーディングで表示するように設定されています。

スクリプト・エラーの操作

HTML タグのすべてのエラーは、ブラウザによって処理されます。PL/SQL Server Pages のロード・プロセスは、それらをチェックしません。

PL/SQL コード内に構文エラーがあった場合、ローダーは停止します。継続するには、エラーを修正する必要があります。構文エラーを含むストアド・プロシージャの以前のバージョンおよびスクリプトを置換しようとすると、そのストアド・プロシージャは消去されるので注意してください。1 つのデータベースをプロトタイプおよびデバック用に使用し、その後、最終的なストアド・プロシージャを別の本番データベースにロードすることもできます。コマンドライン・フラグを使用すると、どのソース・コードを変更しなくてもデータベースを切り替えることができます。

スクリプトの実行時に発生するデータベース・エラーを処理するには、PSP ファイルに PL/SQL 例外処理コードを含め、未処理の例外があれば、特別なページに提示させます。未処理の例外用のページは、拡張子 .psp を持つ、別の PL/SQL Server Pages です。エラー・プロシージャは、どのパラメータも受け取りません。そのため、エラーの原因を判断するには、SQLCODE ファンクションおよび SQLERRM ファンクションをコールします。

また、エラーが発生した場合は、スクリプトを使用しない標準 HTML ページを表示することもできます。ただし、拡張子を .psp とし、ストアド・プロシージャとしてデータベースにロードする必要があります。

PSP スクリプトでの PL/SQL ストアド・プロシージャのネーミング

トップレベルの各 PL/SQL Server Pages は、サーバー内の 1 つのストアド・プロシージャと対応しています。デフォルトでは、プロシージャには元のファイルと同じ名前が、拡張子 .psp なしで付いています。プロシージャに別の名前を付けるには、
<%@ page procedure="*procname*" %> ディレクティブを含めます。

PSP スクリプトへの別のファイルのコンテンツの挿入

挿入メカニズムを設定することによって、通常、静的 HTML コンテンツまたはより多くの PL/SQL スクリプト・コードのいずれかを含む別のファイルのコンテンツを含めることができます。別のファイルのコンテンツを表示させる部分で、
<%@ include file="*filename*" %> ディレクティブを含めます。ファイルは、ストアド・プロシージャをデータベースにロードするときに処理されるため、置換は、ページの提示ごとではなく、1 回のみ行われます。

インクルード・ファイルには、どのような名前および拡張子でも使用できます。インクルード・ファイルに PL/SQL スクリプト・コードが含まれている場合、プロシージャ名やキャラクタ・セットなどを識別するための独自のディレクティブは必要ありません。

PSP ローダーにファイル名を指定するときは、すべてのインクルード・ファイルの名前も含める必要があります。インクルード・ファイルの名前は、すべての .psp ファイル名より前に指定してください。

ナビゲーション・バナーなどの同じ内容を多数のファイルに含める場合、この機能を使用できます。また、これをマクロ機能として使用し、スクリプト・コードの同じセクションを 1 つのページの複数の位置に含めることもできます。

PSP スクリプトでの変数の宣言

スクリプト内でグローバル変数を使用する必要がある場合、デリミタ <%! %> 内に宣言ブロックを含めることができます。標準的なすべての PL/SQL 構文は、ブロック内で使用できます。デリミタは略記として機能するため、DECLARE キーワードを省略できます。すべての宣言は、ファイル内のその後のコードに使用できます。

複数の宣言ブロックを内部的に指定できます。それらはすべて、PSP ファイルがストアド・プロシージャとして作成されたときに単一ブロックにマージされます。

後で説明する <% %> デリミタ内に明示的な DECLARE ブロックを使用することもできます。これらの宣言は、後続の BEGIN/END ブロックのみが参照できます。

PSP スクリプトでの実行可能文の指定

デリミタ <% %> には、どのような PL/SQL 文でも含めることができます。文は、完全なものでも、IF-THEN-ELSE の IF 部分のような複合文の句でもかまいません。DECLARE ブロック内に宣言されたすべての変数は、後続の BEGIN/END ブロックのみが参照できます。

PSP スクリプトでの式結果の置換え

PL/SQL 式の結果に依存する値を含めるには、デリミタ <%= %> 内にその式を含めます。結果は常にテキストまたはタグの中で置き換えられるため、値は文字列または文字列にキャストできるものである必要があります。DATE など、暗黙的にキャストできないタイプの場合、値を PL/SQL の TO_CHAR ファンクションに渡します。

<%= %> デリミタ間の内容は、HTP.PRN ファンクションによって処理されます。これは、先行または後続空白をすべて切り捨て、リテラル文字列を引用符で囲むように要求します。

PSP スクリプトでの文字列の引用符およびエスケープの規則

PSP 属性に指定された値が PL/SQL 処理に使用された場合、それらは PSP ファイルに指定されたとおりに渡されます。PL/SQL が一重引用符付き文字列を要求する場合、その文字列を一重引用符で囲み、すべてを二重引用符で囲んで指定する必要があります。

一重引用符の中に一重引用符付きの文字列をネストすることもできます。この場合、シーケンス \' を指定してネストされた一重引用符をエスケープする必要があります。

ほとんどの文字および文字列は、PSP ローダーで変更しなくても PSP ファイルに含めることができます。シーケンス %> を含めるには、エスケープ・シーケンス %\> を指定します。シーケンス <% を含めるには、エスケープ・シーケンス <\% を指定します。

PSP スクリプトへのコメントの挿入

PL/SQL Server Pages の HTML 部分にコメントを挿入する場合、PSP ソース・コードが簡単に読めるように次の構文を使用します。

```
<%-- Comment text --%>
```

これらのコメントは PL/SQL Server Pages からの HTML 出力には表示されません。

HTML 出力に表示されるコメントを作成するには、HTML 部分にコメントを挿入して、次のような通常の HTML コメント構文を使用します。

```
<!-- Comment text -->
```

PSP 内で PL/SQL ブロックの中にコメントを挿入するには、通常の PL/SQL コメント構文を使用します。

次に、数種類のコメントを表示した PSP ファイルの一部を示します。

```
<p>Today we introduce our new model XP-10.
<%--
This is the project with code name "Secret Project".
People viewing the HTML page will not see this comment.
--%>
<!--
Some pictures of the XP-10.
People viewing the HTML page will see this comment.
-->
<%
for image_file in (select pathname, width, height, description
  from image_library where model_num = 'XP-10')
-- Comments interspersed with PL/SQL statements.
-- People viewing the HTML page will not see this comment.
loop
%>
<img src="<%= image_file.pathname %>" width=<% image_file.width %>
height=<% image_file.height %> alt="<% image_file.description %>">
<br>
<%
end loop;
%>
```

PSP スクリプトでの問合せの結果セットの取得

ここでは、HTML を設計する場合に、データベースからデータを取得して表示する例を示します。

複数行を戻す問合せ結果を表示するには、結果セットの各行を反復し、HTML リストまたは表タグを使用して適切な列を出力します。

```
<% FOR item IN (SELECT * FROM some_table) LOOP %>
  <TR>
  <TD><%= item.col1 %></TD>
  <TD><%= item.col2 %></TD>
  </TR>
<% END LOOP; %>
```

1 回の操作でデータベース表全体を出力するには、PL/SQL Web Toolkit から OWA_UTIL.TABLEPRINT プロシージャまたは OWA_UTIL.CELLSPRINT プロシージャをコールします。

```
<% OWA_UTIL.TABLEPRINT(CTABLE => 'some_table', CATTRIBUTES => 'border=2', CCOLUMNS
=> 'col1, col2', CCLAUSES => 'WHERE col1 > 5'); %>

htp.tableOpen('border=2');
owa_util.cellsprint( 'select col1, col2 from some_table where col1 > 5');
htp.tableClose;
```

PSP スクリプトのコーディングに関するヒント

異なる PL/SQL Server Pages 間でプロシージャ、定数および型を共有するには、プレーンな PL/SQL ソース・ファイルを使用して、それらをデータベース内の個別のパッケージにコンパイルします。PSP スクリプトからパッケージ・プロシージャ、定数および型を参照できますが、PSP スクリプトで生成できるのは、パッケージではなくスタンドアロン・プロシージャのみです。

すべてのディレクティブおよび宣言を PL/SQL Server Pages の最初の方にまとめて配置すると、メンテナンスがより簡単になります。

PL/SQL Server Pages 要素の構文

これらの要素の例については、18-24 ページの「PL/SQL Server Pages の例」を参照してください。

ページ・ディレクティブ

次の PL/SQL Server Pages の特性を指定します。

- 使用するスクリプト言語
- 生成する情報のタイプ（MIME タイプ）
- 補足されなかったすべての例外を処理するために実行するコード。このコードは、意味がわかるメッセージを含む HTML ファイルの場合があり、.psp ファイルに名前が変更されます。主な PSP ファイルをコンパイルする loadpsp コマンドにも同じファイル名を指定する必要があります。errorPage ディレクティブおよび loadpsp コマンドの

両方に、../include/ などのすべての関連パス名を含む同じ名前を正確に指定する必要があります。

属性名 contentType および errorPage は、大 / 小文字を区別することに注意してください。

構文

```
<%@ page [language="PL/SQL"] [contentType="content type string"] [errorPage="file.psp"] %>
```

プロシージャ・ディレクティブ

PSP ファイルによって生成されたストアド・プロシージャの名前を指定します。デフォルトでは、この名前は拡張子 .psp のないファイル名です。

構文

```
<%@ plsql procedure="procedure name" %>
```

パラメータ・ディレクティブ

PSP ストアド・プロシージャによって予測される各パラメータの名前、および（オプションで）型とデフォルト値を指定します。通常、パラメータは、HTML フォームから名前値の組を使用して渡されます。キャラクタ・タイプのデフォルト値を指定するには、値を一重引用符で囲み、さらにディレクティブで必要な二重引用符で囲みます。次に例を示します。

```
<%@ parameter="username" type="varchar2" default="'nobody'" %>
```

構文

```
<%@ plsql parameter="parameter name" [type="PL/SQL type"] [default="value"] %>
```

インクルード・ディレクティブ

PSP ファイル内の特定のポイントに含めるファイルの名前を指定します。ファイルには、.psp 以外の拡張子を付ける必要があります。ファイルには、HTML、PSP スクリプト要素またはこれらの組合せを含めることができます。名前の解決およびファイルの挿入は、PSP ファイルがストアド・プロシージャとしてデータベースにロードされたときに行われます。そのため、それ以降のファイルへの変更は、ストアド・プロシージャの実行時には反映されません。

インクルード・ディレクティブと loadpsp コマンドの両方に、../include/ などのすべての関連パス名を含む同じ名前を正確に指定する必要があります。

構文

```
<%@ include file="path name" %>
```

宣言ブロック

次の BEGIN/END ブロック内のみでなく、ページ全体を通して参照できる、一連の PL/SQL 変数を宣言します。通常、この要素は複数行にまたがっており、最後にセミコロンが付いた個々の PL/SQL 変数宣言を持ちます。

構文

```
<%! PL/SQL declaration;
    [ PL/SQL declaration; ] ... %>
```

コード・ブロック（スクリプトレット）

ストアド・プロシージャの実行時に、一連の PL/SQL 文を実行します。通常、この要素は複数行にまたがっており、最後にセミコロンが付いた個々の PL/SQL 変数宣言を持ちます。この文は、完全なブロックを含むことも、IF/THEN/ELSE または BEGIN/END ブロックの大カッコで囲まれた部分になることもできます。コード・ブロックが複数のスクリプトレットに分割されると、HTML またはその他のディレクティブをそれらの間に置くことができます。これによって、これら複数のスクリプトレットは、ストアド・プロシージャが実行されたときに、条件付きで実行されます。

構文

```
<% PL/SQL statement;
   [ PL/SQL statement; ] ... %>
```

式ブロック

文字列、算術式、ファンクション・コールまたはこれらを組み合せて、単一の PL/SQL 式を指定します。結果は、ストアド・プロシージャによって生成された HTML ページ内の該当部分で文字列に置き換えられます。PL/SQL 式の最後にセミコロンを付ける必要はありません。

構文

```
<%= PL/SQL expression %>
```

ストアド・プロシージャとしての PL/SQL Server Pages のデータベースへのロード

1 つ以上の PSP ファイルをストアド・プロシージャとしてデータベースにロードします。各 .psp ファイルは 1 つのストアド・プロシージャと対応します。開発サイクルを短くするために、ページのコンパイルおよびロードは 1 手順で行われます。

```
loadpsp [ -replace ] -user username/password[@connect_string]
    [ include_file_name ... ] [ error_file_name ] psp_file_name ...
```

ストアド・プロシージャに対して作成および置換を行うには、-replace フラグを含めます。

ローダーは、指定されたユーザー名、パスワードおよび接続文字列を使用して、データベースにログインします。ストアド・プロシージャは、対応するスキーマ内で作成されます。

PL/SQL Server Pages の名前（拡張子が .psp）の前に、すべてのインクルード・ファイルの名前（拡張子が .psp ではない）を含めます。さらに、page ディレクティブの errorPage 属性に指定されたファイルの名前も含めます。loadpsp コマンドライン上のこれらのファイル名は、../include/ などの関連パス名も含めて、PL/SQL Server Pages の include ディレクティブおよび page ディレクティブに指定された名前と完全に一致する必要があります。

次に例を示します。

```
loadpsp -replace -user scott/tiger@WEBDB banner.inc error.psp display_order.psp
```

この例は、次のことを示しています。

- ストアド・プロシージャは、データベース WEBDB で作成されます。ストアド・プロシージャを作成および実行するときは、データベースはユーザー scott、パスワード tiger でアクセスされます。
- banner.inc は、ボイラープレート・テキストとスクリプト・コードを含むファイルで、.psp ファイルに含まれています。banner.inc は、ストアド・プロシージャが実行されるときではなく、PL/SQL Server Pages がデータベースにロードされるときに含まれます。
- error.psp は、未処理例外が発生したときに処理されるコードまたはテキスト（あるいはその両方）を含むファイルで、内部エラー・メッセージではなく意味がわかる内容のページを表示します。
- display_order.psp には、Web ページの主要コードおよびテキストが含まれています。デフォルトでは、対応するストアド・プロシージャは DISPLAY_ORDER という名前です。

URL を介した PL/SQL Server Pages の実行

PL/SQL Server Pages が一度ストアド・プロシージャに変換されると、Web ブラウザまたは他のインターネット関連クライアント・プログラムを使用して HTTP URL を取得することによって、そのプロシージャを実行できます。URL の仮想パスは、Web ゲートウェイが構成された方法によって異なります。

ストアド・プロシージャへのパラメータは、HTTP プロトコルの POST メソッドまたは GET メソッドのいずれかを使用して渡されます。POST メソッドの場合、パラメータは HTML フォームから直接渡され、URL には表示されません。GET メソッドの場合、パラメータは URL の問合せ文字列で名前値の組として渡されます。URL の問合せ文字列は、エンコード形式のほとんどの英数字以外の文字（たとえば、空白は `%20`）で、`&` 文字で区切られています。PSP ページを HTML フォームからコールするには、GET メソッドを使用します。また、特定のパラメータ・セットを持つストアド・プロシージャをコールするには、ハードコード化された HTML リンクを使用します。

サンプル PSP URL

`METHOD=GET` を使用した場合、URL は次の例のようになります。

`http://`*`sitename`*`/`*`schemaname`*`/`*`pspname`*`?`*`parmname1`*`=`*`value1`*`&`*`parmname2`*`=`*`value2`*

`METHOD=POST` を使用した場合、パラメータは URL に表示されません。

`http://`*`sitename`*`/`*`schemaname`*`/`*`pspname`*

`METHOD=GET` 形式は、デバッグに有効です。また、ページの閲覧者がブックマークを使用して再度ページを開く際、同じパラメータを渡すことができます。

`METHOD=POST` 形式は、さらに大きいパラメータ・データを扱えます。また、URL に表示してはならない機密情報を渡す場合に適しています（URL は、ブラウザの履歴リスト、および次のアクセス・ページに渡される HTTP ヘッダーの中に残ります）。この方法でコールされたページにブックマークを付けるのは、効果的ではありません。

PL/SQL Server Pages の例

この項では、非常に単純な PL/SQL Server Pages から始め、徐々に複雑なバージョンを作成していく方法を説明します。

各手順を行いながら、18-23 ページの「ストアド・プロシージャとしての PL/SQL Server Pages のデータベースへのロード」および 18-24 ページの「URL を介した PL/SQL Server Pages の実行」に記載されているプロシージャを使用して、PSP ファイルをコンパイルし、それらをブラウザで試してください。

サンプル表

この例では、製品カタログを示す非常に小さい表を使用します。この表には、品目名、価格、および製品の写真と説明を参照できる URL が表示されています。

```
Name       Type
---------- -------------
PRODUCT    VARCHAR2(100)
PRICE      NUMBER(7,2)
URL        VARCHAR2(200)
PICTURE    VARCHAR2(200)

Guitar
455.5
http://auction.fictional_site.com/guitar.htm
http://auction.fictional_site.com/guitar.jpg


Brown shoe
79.95
http://retail.fictional_site.com/loafers.htm
http://retail.fictional_site.com/shoe.gif

Radio
9.95
http://promo.fictional_site.com/freegift.htm
http://promo.fictional_site.com/alarmclock.jpg
```

サンプル表のダンプ

独自にデバッグを行うには、SQL 表の完全な内容を表示する必要がある場合があります。このためには、OWA_UTIL.TABLEPRINT を 1 回コールします。後続の反復では、他の方法でより詳細に表示を制御します。

```
<%@ plsql procedure="show_catalog_simple" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Complete Dump)</TITLE></HEAD>
<BODY>
<%
declare
dummy boolean;
begin
dummy := owa_util.tableprint('catalog','border');
end;
%>
</BODY>
</HTML>
```

ループを使用したサンプル表の出力

次に、表の中の項目をループして、必要なもののみを明示的に出力します。

- SELECT 文を調整して、行または列のサブセットのみを取得できます。
- HTML または式の位置を変更して、各品目の外観または列の表示順序を変更できます。
- この段階では、一致していない表タグまたはクローズしていない表タグによる問題を回避するために、非常に単純な一連のリスト品目を例として示します。

```
<%@ plsql procedure="show_catalog_raw" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Raw Form)</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select * from catalog order by price desc) loop %>
<LI>
Item = <%= item.product %><BR>
price = <%= item.price %><BR>
URL = <I><%= item.url %></I><BR>
picture = <I><%= item.picture %></I>
<% end loop; %>
</UL>
</BODY>
</HTML>
```

前述の単純な例が正常に表示されたら、次に、より使用価値の高い形式で内容を表示できます。

- 特定の値を強調するために、それらの値を HTML タグで囲みます。
- 説明および写真のある URL を出力するかわりに、読み手が写真を見たりリンクをたどったりできるように、リンク・タグおよびイメージ・タグを使用します。

```
<%@ plsql procedure="show_catalog_pretty" %>
<HTML>
<HEAD><TITLE>Show Contents of Catalog (Better Form)</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select * from catalog order by price desc) loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>
</BODY>
</HTML>
```

ユーザー選択の許可

動的なページができあがりましたが、ユーザーの視点からはまだ少しものたりないかもしれません。カタログ表を更新しないかぎり、結果はいつも同じになります。

- 必要に応じて、最低価格を表示することも、より高価な品目のみを表示することもできます（顧客の購買基準は様々です）。
- ページがブラウザに表示されたとき、デフォルトの最低価格は 100（単位は該当通貨）です。その後、ユーザーが最低価格を選択できるように設定されています。

```
<%@ plsql procedure="show_catalog_partial" %>
<%@ plsql parameter="minprice" default="100" %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
<P>This report shows the items whose price is greater than <%= minprice %>.
<UL>
<% for item in (select * from catalog where price > minprice order by price desc)
loop %>
<LI>
Item = <A HREF="<%= item.url %>"><%= item.product %></A><BR>
price = <BIG><%= item.price %></BIG><BR>
<IMG SRC="<%= item.picture %>">
<% end loop; %>
</UL>
</BODY>
</HTML>
```

前述のとおり、結果をフィルタする方法は、ユーザーに対する選択肢が多すぎる心配のある検索結果など、一部のアプリケーションには適用できます。ただし、小売りの場合は、顧客が他の品目も選択できるように、別の方法を使用する必要があります。

- WHERE 句を使用して結果をフィルタするかわりに、すべての結果セットを取得し、戻された各行に対して別々のアクションを実行します。
- HTML を変更して、基準に合う出力のみをハイライト表示させます。ここでは、HTML 表の行にバックグラウンド・カラーを使用します。最も重要な行が目立つようにするために、特別なアイコンを挿入したり、フォント・サイズを大きくすることもできます。
- ここで、あるユーザーの経験を示す結果を HTML 表に表示してみます。

```
<%@ plsql procedure="show_catalog_highlighted" %>
<%@ plsql parameter="minprice" default="100" %>
<%! color varchar2(7); %>

<HTML>
<HEAD><TITLE>Show Items Greater Than Specified Price</TITLE></HEAD>
<BODY>
```

```
<P>This report shows all items, highlighting those whose price is
 greater than <%= minprice %>.
<TABLE BORDER>
<TR>
<TH>Product</TH>
<TH>Price</TH>
<TH>Picture</TH>
</TR>
<%
for item in (select * from catalog order by price desc) loop
  if item.price > minprice then
    color := '#CCCCFF';
  else
    color := '#CCCCCC';
  end if;
%>
<TR BGCOLOR="<%= color %>">
<TD><A HREF="<%= item.url %>"><%= item.product %></A></TD>
<TD><BIG><%= item.price %></BIG></TD>
<TD><IMG SRC="<%= item.picture %>"></TD>
</TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

PL/SQL Server Pages をコールするためのサンプル HTML フォーム

次に、HTML フォームの要点を示します。ここに価格を入力し、入力した値を MINPRICE パラメータとして渡す SHOW_CATALOG_PARTIAL ストアド・プロシージャをコールします。

このフォームの ACTION= 属性で、ストアド・プロシージャの URL 全体をコード化することを回避するには、そのストアド・プロシージャがコールする PSP ファイルと同じディレクトリに入るように、このフォームを PSP ファイルとして作成します。この HTML ファイルには PL/SQL コードがありませんが、これに .psp 拡張子を付け、ストアド・プロシージャとしてデータベースにロードできます。ストアド・プロシージャが実行されたとき、HTML はファイルに表示されるとおりに表示されます。

```
<html>
<body>
<form method="POST" action="show_catalog_partial">
<p>Enter the minimum price you want to pay:
<input type="text" name="minprice">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

注意： HTML フォームは、ツールおよびプログラム言語を使用して生成する他のフォームとは異なります。HTML フォームは、HTML ファイルの一部であり、<FORM> タグおよび </FORM> タグで囲まれています。ここで、項目を選択したりデータを入力することができます。それらの選択は、CGI プロトコルを使用してサーバー側のプログラムに転送されます。

PSP を使用して完全なアプリケーションを生成するには、構文 <INPUT>、<SELECT> およびフォームに関連する他の HTML タグを習得してください。

PSP ファイルへの Java スクリプトの挿入

Java スクリプトなどの動的コンテンツを含むような、複雑な HTML ファイルを生成するには、ソース・コードを PL/SQL Server Pages として実装することによって単純化します。この方法では、ネストされた引用符、エスケープ文字、連結リテラルや変数および埋込みコンテンツのインデントを考慮する必要がなくなります。

次に、PL/SQL Server Pages を使用して Java スクリプトを含む HTML ファイルを生成する方法を示します。

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="graph" %>
<%!
-- Begin with a date that does not exist in the audit table
last_timestamp date := sysdate + 1;
%>
<html>
<head>
<title>Usage Statistics</title>
<script language="JavaScript">
<!--
d=document

// Draw a horizontal graph line using a graphic that is stretched
// by a scaling factor.
function graph(howmuch)
{
  preamble = "<img src='/images/graph_line.gif' height='8' width='"
  climax = howmuch * 4;
  denouement = "'> (" + howmuch + ")\n"
  d.write( preamble + climax + denouement )
}
// -->
</script>
</head>
<body text="#000000" bgcolor="#FFFFFF">
```

```
<h1>Usage Statistics</h1>

<table border=1>

<%
-- For each day, count how many times each procedure was called.
for item in (select trunc(time_stamp) t, count(*) n, procname p
  from audit_table group by trunc(time_stamp), procname
  order by trunc(time_stamp) desc, procname)
loop
-- At the start of each day's data, print the date.
  if item.t != last_timestamp then
    htp.print('<tr><td colspan=2><font size="+2">');
    htp.print(htf.bold(item.t));
    htp.print('</font></td></tr>');
    last_timestamp := item.t;
end if;
%>

<tr><td><%= item.p %></a>:
<td>
<!-- Render an image of variable width to represent the data value. -->
<script language="JavaScript">
<!--
graph(<%= item.n %>)
// -->
</script>
</td>
</tr>

<% end loop; %>

</table>

</body>
</html>
```

このプロシージャを通常の PL/SQL ストアド・プロシージャとしてコーディングすると、次のとおり、二重アポストロフィを含む非常に複雑な行が生成されます。

```
htp.print('preamble = "<img src=''/images/graph_line.gif'' height=''8'' width=''"');
```

PL/SQL Server Pages のデバックに関する問題

PL/SQL Server Pages に取り組み始めた際、および最初の単純なページからより複雑なページに進む過程で、問題が発生した場合は、次のガイドラインに従ってください。

- まず、すべての PL/SQL 構文および PSP ディレクティブ構文を正しく作成します。ここで間違っていれば、ファイルはコンパイルできません。
  - セミコロンで終了する必要のある行に、セミコロンが付いているかどうかを確認します。
  - 引用符で囲む必要のある値に、引用符が付いているかどうかを確認します。一重引用符で囲まれた値（PL/SQL で必要）を二重引用符（PL/SQL Server Pages で必要）で囲む必要がある場合もあります。
  - PSP ディレクティブ内の誤りは、通常、PL/SQL 構文メッセージによってレポートされます。適切な構文が使用され、適切にクローズされ、内容に応じた適切な要素（宣言、式またはコード・ブロック）が使用されているかどうかについて、ディレクティブをチェックします。
  - PSP 属性名は、大 / 小文字を区別します。ほとんどの場合、すべて小文字で指定します。`contentType` および `errorPage` は、大文字と小文字を組み合せて指定する必要があります。
- 次に、Web ブラウザで URL を要求することによって PSP ファイルを実行します。このときに、ファイルが見つからないというエラーが表示される場合があります。
  - Web ゲートウェイの構成方法に合った、正確な仮想パスを要求しているかどうかを確認します。通常、パスにはホスト名が含まれます。また、オプションで、ポート番号、スキーマ名およびストアド・プロシージャの名前（`.psp` 拡張子なし）が含まれます。
  - ファイルをコンパイルするときに `-replace` オプションを使用した場合、ストアド・プロシージャの古いバージョンは消去されます。そのため、コンパイルに失敗した場合は、エラーを修正しなければページは使用できません。新しいスクリプトは、準備ができるまで別のスキーマでテストし、その後、本番スキーマにロードするという方法をとることもできます。
  - ファイルを別のファイルからコピーした場合、ソース内のプロシージャ名ディレクティブを新しいファイル名と一致させるように変更してください。
  - ファイルが見つからないというエラーが 1 つでも戻された場合、次回は必ず最新バージョンのページを要求してください。ブラウザがエラー・ページをキャッシュする場合があります。キャッシュを回避するには、ブラウザで [Shift] を押しながら「Reload」を押す必要がある場合があります。
- PSP スクリプトが実行されると、結果はブラウザに戻されます。標準のデバッグ方法を使用して、出力のチェックおよび修正を行ってください。ここでは、すべての適切な値がページに渡されるように、異なる HTML フォーム、スクリプト、CGI プログラムの

間でインタフェースを設定することが重要です。パラメータが一致しなければ、ページがエラーを戻す場合があります。

- ページに渡される内容を正確に把握するには、URL でパラメータを参照できるように FORM タグの中で METHOD=GET を使用します。
- ページをコールするフォームまたは CGI プログラムが適切なパラメータ数を渡すかどうかを確認します。また、フォームの NAME= 属性で指定した名前が、PSP ファイルのパラメータ名と一致していることも確認します。フォームに非表示の入力フィールドがある場合、あるいは「Submit」ボタンまたは「Reset」ボタンに NAME= 属性が使用されている場合、PSP ファイルは同等のパラメータを宣言する必要があります。
- パラメータが文字列から適切な PL/SQL 型にキャストされることを確認します。たとえば、PSP ファイルのパラメータが NUMBER で宣言されている場合、アルファベット文字を含めることはできません。
- ページへのハードコード・リンクを構成してパラメータを渡す場合は、特に、URL の問合せ文字列が、等号で区切られた名前値の組で構成されていることを確認します。
- 大きい文字列など、多数のパラメータ・データを渡す場合、METHOD=GET で渡すことができるボリュームを超えてしまう場合があります。その場合、PSP ファイルを変更せずに、FORM タグの中で METHOD=POST に切り替えることができます。
- ソース・ファイルに構文エラーがある場合、loadpsp コマンドは、行番号を正しくレポートしますが、実行中のエラーに対してレポートされた行番号は、ソースの変換バージョンを参照し、元のソースの行番号とは一致しません。予想した Web ページではなくエラー・トレースが表示されるようなエラーが発生した場合は、例外ハンドラおよびデバッグ出力の表示によって、エラーの位置を特定する必要があります。

PL/SQL Server Pages を使用したアプリケーションの商品化

PL/SQL Server Pages を使用したアプリケーションの開発では、ほとんどの時間を論理的に正しいスクリプトを作成することに費やす場合があります。アプリケーションを商品化する前に、有用性やダウンロード速度など、他の問題も考慮する必要があります。

- すべてのイメージに HEIGHT= 属性および WIDTH= 属性が指定されると、ブラウザがページを生成する速度が速くなります。また、写真のサイズを標準化したり、イメージの高さおよび幅をデータまたは URL とともにデータベースに格納しておくこともできます。
- 図形を表示しないビューア、またはテキストを音声で読み上げるブラウザを使用しているビューア用には、ALT= 属性を使用して重要な図形の説明を含めます。これも、イメージとともにデータベースに格納しておきます。
- HTML の表はデータ表示の方法として有効ですが、大きい表が原因でアプリケーションの速度が遅く感じることがあります。表全体がダウンロードされるまで、ビューアには

空白のページが表示されます。HTML 表のデータ量が大きければ、出力を複数の表に分割することを考慮してください。

- テキスト、フォントまたはバックグランドにそれぞれカラーを設定している場合、アプリケーションをブラウザのカラー設定の様々な組合せでテストします。
  - ブラウザでフォアグラウンド・カラーのみをオーバーライドした場合、バックグラウンド・カラーのみをオーバーライドした場合、または両方をオーバーライドした場合に、それぞれどのような結果になるかをテストします。
  - 一般的に、1 つのカラー（たとえば、フォアグラウンド・テキスト・カラー）を設定した場合に、白のバックグランドに白のテキストというように読みにくい組合せを回避するために、すべてのカラーを <BODY> タグで設定する必要があります。
  - バックグランド・イメージを使用する場合、図形をロードしないビューアに対して適切なコントラストを設定できるように、同系色のバックグランド・カラーを指定します。
  - 異なるカラーで表示する情報が重要な意味を持つ場合、カラーのかわりに別の方法を使用するか、またはカラーに別の方法を加えることも検討します。たとえば、表内の特別な項目の横に図形アイコンを置いたりします。ビューアによっては、ページをモノクロ画面で表示するものや、様々なカラーを表示できないブラウザを使用しているビューアもあります（洋服のポケットに入るような、ペン状の器具を使用して入力するタイプのブラウザなどです）。
- ユーザーに内容の情報を提供すると、ユーザーが迷わなくなります。ページに、わかりやすい <TITLE> タグを含めます。ユーザーが手順の途中にいる場合は、表示されているページがどの手順なのかを示します。手順を継続するか、前の手順に戻るか、または手順を完全に中止するかを示す論理点へのリンクを設定します。多くのページには、インクルード・ディレクティブを使用して埋め込んだ、標準のリンクが使用されています。
- ユーザーは、入力フィールドに不適切な値を入力してしまう場合があります。できるだけ、選択肢を示した SELECT 構文のリストを使用します。フィールドに入力されたテキストは、SQL に渡す前に、すべて妥当性チェックを行う必要があります。妥当性は早い段階でチェックする方が効果的です。Java スクリプト・ルーチンは、不適切なデータを検出し、ユーザーが「Submit」ボタンを押してデータベースにコールする前に、ユーザーに対して修正を促します。
- ブラウザが、間違った HTML を表示する場合もあります。ただし、1 つのブラウザでは正常に表示されても、別のブラウザでは正常に表示されなかったり、表示されない場合もあります。
  - 引用符、タグのクローズおよび表に関する HTML ルールには注意してください。
  - 単一のブラウザのみでサポートされているタグに依存することは、最小限にしてください。そのようなタグを使用することによって便利な場合もありますが、アプリケーションは他のブラウザでも使用可能である必要があります。

- 多くの Web サイトで、HTML の妥当性および（場合によっては）有用性が無償でチェックできます。

XML に対する PL/SQL Web アプリケーションの使用可能化

PL/SQL Web アプリケーションで XML 形式のデータを受け入れたり、HTML ではなく XML のタグ付き出力を生成する必要がある場合があります。

出力を表示するときは、Web ページの MIME タイプを `text/xml` に設定して、XML 対応ブラウザまたは他の Web クライアント・ソフトウェアが、出力を XML として認識できるようにします。

また、アプリケーション内で、`XMLTYPE` 型、`DBMS_XMLQUERY` パッケージや `DBMS_XMLSAVE` パッケージ、`SYS_XMLGEN` ファンクションや `SYS_XMLAGG` ファンクションなど、多くの組込み機能も使用できます。これらの機能の詳細は、『Oracle9*i* XML Developer's Kit ガイド -XDK』を参照してください。

19

Oracle 以外のアプリケーションから Oracle9*i* への移植

多くの場合、プログラミング・プロジェクトでは、新しくコードを作成するより既存のコードを改良する必要があります。他のデータベース・プラットフォームからコードを移植する場合は、移植を簡略化するために設計された Oracle の機能を理解することが重要です。

内容は次のとおりです。

移植に関する FAQ

自然結合および内部結合を実行するにはどうすればよいですか？

他のデータベース・システムから Oracle に問合せを移植する場合、以前は、ANSI の結合表記法を Oracle のカンマ表記法に変換する必要がありました。

現在は、ANSI 準拠の結合表記法を使用して、Oracle の問合せをコード化できます。次に例を示します。

```
SELECT * FROM a NATURAL JOIN b;
SELECT * FROM a JOIN b USING (c1);
SELECT * FROM a JOIN b USING (c1) WHERE c2 > 100;
SELECT * FROM a NATURAL JOIN b INNER JOIN c;
```

標準の表記法では、表間の関連が明示的になるため、WHERE 句内で結合条件に対する等価テストをコード化する必要がありません。また、完全外部結合のサポートによって、これらの問合せを実行するために、複雑な操作を行う必要もありません。

ベンダーによってサポートする標準の結合構文の数が異なったり、一部のベンダーが独自の拡張構文を導入している場合もあるため、結合クエリー・リライトが必要な場合もあります。

SELECT 文の完全な構文および結合表記法の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

他のデータベース・システムからスキーマおよび関連するデータを自動的に移行する方法はありますか？

あります。他のデータベース製品から Oracle8 および Oracle8*i* にスキーマ（データ、トリガーおよびストアド・プロシージャを含む）を変換する Oracle Migration Workbench という製品が無償で提供されます。製品自体は Windows 上で実行されますが、他のオペレーティング・システム上のデータベースから、任意のオペレーティング・システム上で実行している Oracle データベースに、データを転送できます。

この製品を使用すると、Oracle8*i* に切り替えるときに、アプリケーションを作成してレガシー・データを変換する必要がありません。関連テクノロジによって、特定の種類のソース・コードを変換できます。たとえば、Visual Basic のコードを Java に移植できます。

このマニュアルの発行時点では、移植は次のデータベースでサポートされます。

- MS SQL Server 6.5
- MS SQL Server 7.0
- MS Access 2.0
- MS Access 95

- MS Access 97
- Sybase Adaptive Server 11
- MySQL 3.22

問合せ内で多数の比較を実行するにはどうすればよいですか？

問合せ内で様々な条件から選択を行う場合は、次の構文を使用します。

- SQL 文の CASE

```
SELECT CASE
  WHEN day IN
    ('Monday','Tuesday','Wednesday','Thursday','Friday')
    THEN 'weekday'
  WHEN day in
    ('Saturday','Sunday')
    THEN 'weekend'
  ELSE 'unknown day' END
FROM DUAL;
```

この方法を使用すると、PL/SQL ファンクションへのコールを SQL で直接行うテストに置換できる場合に、パフォーマンスが向上します。

Oracle8*i* では、検索 CASE、単純 CASE、NULLIF および COALESCE に対する SQL-92 の表記法がサポートされます。

- SQL 関数 DECODE

```
SELECT DECODE (day,
  'Monday', 'weekday',
  'Tuesday', 'weekday',
  'Wednesday', 'weekday',
  'Thursday', 'weekday',
  'Friday', 'weekday',
  'Saturday', 'weekend',
  'Sunday', 'weekend,
  'unknown day')
INTO day_category FROM DUAL;
```

この構造では、様々な選択肢に違反する変数をテストし、状況に応じて異なる値を戻すことができます。一致する選択肢がない場合は最後の値が使用されます。

CASE を使用する方法は、移植性が高く、新規のコードにより適しています。

Oracle ではスカラー副問合せがサポートされますか？

Oracle9*i* では、スカラー副問合せはサポートされません。

20

Oracle XA でのトランザクション・モニターの操作

この章では、Oracle XA ライブラリの使用方法を説明します。Oracle XA ライブラリは、通常、トランザクション・モニターとともに機能するアプリケーションで使用します。XA の機能は、トランザクションが複数のデータベースと相互処理するアプリケーションで最も効果的です。

Oracle XA ライブラリは、Oracle サーバー以外のトランザクション・マネージャ（TM）でグローバル・トランザクションを調整できるようにする外部インタフェースです。これによって、リソース・マネージャ（RM）と呼ばれる Oracle 以外のエンティティを分散トランザクションに組み込むことができます。

Oracle XA ライブラリは、X/Open Distributed Transaction Processing（DTP）ソフトウェア・アーキテクチャの XA インタフェース仕様に準拠しています。

この章の内容は次のとおりです。

参照：

- 基本アーキテクチャを含む XA の概要は、『X/Open CAE Specification - Distributed Transaction Processing: The XA Specification』を参照してください。
- Oracle XA ライブラリのバックグラウンドおよび参照情報は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。
- ライブラリ・リンク・ファイル名の詳細は、オペレーティング・システム固有の Oracle マニュアルを参照してください。
- `README.doc` ファイルは、オペレーティング・システム固有の Oracle マニュアルで指定されているディレクトリにあり、ご使用のプラットフォームに対応する Oracle XA ライブラリの直前のバージョンからの変更点、エラーまたは制限事項について説明しています。

X/Open DTP

X/Open DTP アーキテクチャは、複数のアプリケーション・プログラム（AP）が複数の異なる RM から提供されるリソースを共有できるようにするための、標準のアーキテクチャまたはインタフェースを定義しています。AP と RM 間の作業を調整し、グローバル・トランザクションを実現します。

図 20-1 に、X/Open DTP モデルの例を示します。

RM は、障害発生後に通常の状態に戻すことができる共有かつリカバリ可能なリソースを制御します。たとえば、Oracle データベース・サーバーは RM であり、障害発生後に REDO ログおよびロールバック・セグメントを使用して通常の状態に復帰します。RM は、データベース、ファイル・システム、プリンタ・サーバーなどの共有リソースにアクセスする方法を提供します。

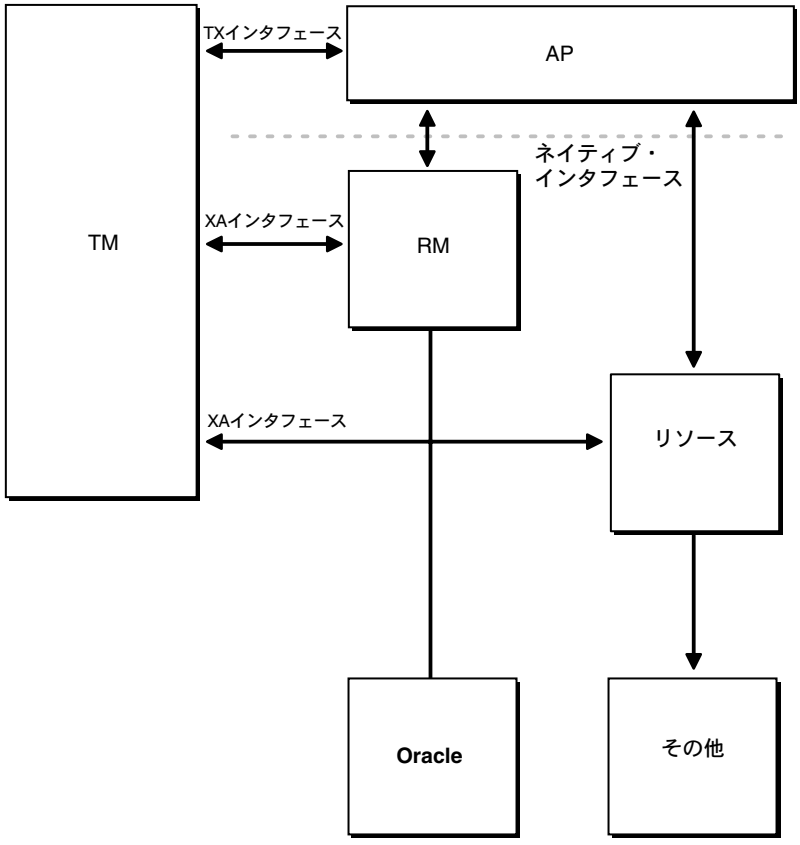
TM は、トランザクションの境界を指定するための Application Program Interface（API）を提供し、コミットおよびリカバリ手順を管理します。

通常、Oracle は Oracle 自体の TM として機能し、コミットおよびリカバリを管理します。ただし、業界標準に準拠した TM を使用することで、Oracle は単一のトランザクション内で他の異機種間 RM と協調できます。

TM は、通常、TPM ベンダーが提供するコンポーネントです。TM はトランザクションに識別子を割り当て、その進行状況を監視し、調整します。TM は、トランザクション内のすべての RM に関する情報をもとに、Oracle XA ライブラリ・サブルーチンを使用して Oracle にトランザクションの処理方法を指示します。この項の後半で、XA サブルーチンとその説明を示します。

AP は、トランザクションの境界を定義し、トランザクションを構成するアクションを指定します。たとえば、AP はプリコンパイラでも OCI プログラムでもかまいません。AP は、RM のネイティブ・インタフェース（たとえば SQL）を使用して、RM のリソースを操作します。ただし、AP は、TM を使用して、TX と呼ばれるインタフェースを介してすべてのトランザクション操作を開始および完了します。AP 自体が、XA インタフェースを直接使用することはありません。

図 20-1　DTP モデルの例



注意： TX インタフェースおよびその関連のサブルーチンのネーミング規則はベンダー固有で、ここで使用している名前とは異なる場合があります。たとえば、tx_open コールは、ご使用のシステムで tp_open と呼ばれている場合があります。用語については、TPM に付属のマニュアルを参照してください。

必須のパブリック情報

Oracle は、RM として、次の情報を発行する必要があります。

XA 機能	Oracle の詳細
xa_switch_t 構造体	Oracle サーバーでは、静的登録のための xa_switch_t 構造体名は xaosw です。動的登録の xa_switch_t 構造体名は xaoswd です。これらの構造体には、RM のエントリ・ポイントおよびその他の情報が含まれています。
xa_switch_t リソース・マネージャ	xa_switch_t 構造体内の Oracle サーバーの RM 名は Oracle_XA です。
クローズ文字列	xa_close() で使用される文字列は無視され、NULL として扱われます。
オープン文字列	xa_open() で使用されるオープン文字列の書式の詳細は、20-10 ページの「xa_open 文字列の定義」を参照してください。
ライブラリ	Oracle XA を使用するアプリケーションをリンクするために必要なライブラリには、オペレーティング・システム固有の名前が付けられています。TPM 固有のライブラリをリンクする必要があることを除けば、通常のプリコンパイラまたは OCI プログラムをリンクすることと同じです。sqllib を使用していない場合は、必ず $ORACLE_HOME/lib/xaonsl.o にリンクしてください。
要件	ありません。XA をサポートする機能は Standard Edition および Enterprise Edition に含まれています。

XA および 2 フェーズ・コミット・プロトコル

Oracle XA ライブラリ・インタフェースは、準備フェーズおよびコミット・フェーズで構成される 2 フェーズ・コミット・プロトコルに従ってトランザクションをコミットします。

第 1 のフェーズである準備フェーズでは、TM は各 RM に対して、トランザクションの任意の部分をコミットできるように要求します。これが可能な場合は、RM は準備ができた状態を記録し、TM に肯定的に応答します。可能でない場合は、RM はすべての作業をロールバックし、TM に否定的に応答し、そのトランザクションに関する情報を消去します。プロトコルによって、アプリケーションまたは RM は、準備フェーズが完了する前にトランザクションを一方的にロールバックできます。

第 2 のフェーズであるコミット・フェーズでは、TM はコミット決定を記録します。その後で、TM はトランザクションに参加しているすべての RM に対してコミットまたはロールバックを発行します。

注意： TM は、すべての RM がフェーズ 1 に対して肯定的に応答した場合にのみ、RM のコミットを発行できます。

トランザクション処理モニター（TPM）

TPM は、トランザクション要求を発行するクライアント・プロセスとその要求を処理するバックエンド・サーバー間の要求フローを調整します。TPM は、基本的にネットワーク上に分散されているアプリケーション・サーバーや RM などのように、様々な種類のバックエンド・プロセスに対してサービスを要求するトランザクションを調整します。

TPM は、分散トランザクションを完了するために必要なコミットおよびロールバックを同期化します。TPM の TM 部分が、分散コミットおよび分散ロールバックの発生するタイミングを制御します。このため、分散 AP で TPM を利用するように作成されている場合、TPM の TM 部分が 2 フェーズ・コミット・プロトコルを制御します。RM を使用して、TM はこの制御を実行します。

TM は分散コミットまたはロールバックを制御するため、Oracle XA ライブラリ・インタフェースを介して Oracle（または他の RM）と直接通信する必要があります。

動的登録および静的登録のサポート

Oracle は、動的登録と静的登録の両方をサポートします。動的登録の場合、RM は最初にアプリケーション・コールバックを実行します。静的登録の場合、関係のない RM があっても、最初に、各 RM に対して `xa_start` を呼び出す必要があります。

動的登録を使用するには、クライアントおよびサーバーの両方が Oracle8 以上である必要があります。Oracle7 以下の場合、使用できるのは静的登録のみです。

Oracle XA ライブラリ・インタフェース・サブルーチン

Oracle XA ライブラリ・サブルーチンを使用することで、TM はトランザクションに関する処理を Oracle に指示できます。一般に、TM は（xa_open を使用して）リソースをオープンする必要があります。通常、これは、AP で tx_open をコールした結果発生します。一部の TM は、アプリケーションが開始したときに、暗黙的に xa_open をコールします。同様に、アプリケーションがリソースの使用を完了したときに発生するクローズがあります（xa_close を使用）。これは、AP が tx_close をコールしたとき、またはアプリケーションが終了したときです。

その他にも、TM が RM に実行するように指示する作業があります。その作業のうち、いくつかを次に示します。

- 新しいトランザクションの開始、およびトランザクションに対する ID の対応付け
- トランザクションのロールバック
- トランザクションの準備およびコミット

XA ライブラリ・サブルーチン

次に示す XA ライブラリ・サブルーチンを使用できます。

XA サブルーチン	説明
xa_open	RM へ接続します。
xa_close	RM からの接続を切断します。
xa_start	新規トランザクションを開始し、指定のトランザクション ID（XID）に対応付けるか、またはプロセスと既存トランザクションを対応付けます。
xa_end	指定された XID からプロセスを切断します。
xa_rollback	指定された XID に対応付けられているトランザクションをロールバックします。
xa_prepare	指定された XID に対応付けられているトランザクションを準備します。これは、2 フェーズ・コミット・プロトコルの第 1 のフェーズです。
xa_commit	指定された XID に対応付けられているトランザクションをコミットします。これは、2 フェーズ・コミット・プロトコルの第 2 のフェーズです。
xa_recover	準備されヒューリスティックにコミットまたはロールバックされたトランザクションのリストを取得します。

XA サブルーチン	説明
xa_forget	指定された XID に対応付けられているヒューリスティック・トランザクションの情報を消去します。

一般に、AP では、xa_open 文字列で実行されるロールを理解する以外に、これらのサブルーチンについて考慮する必要はありません。

XA インタフェースの拡張

Oracle の XA インタフェースには、いくつかの関数が追加されています。

1. `OCISvcCtx *xaoSvcCtx(text *dbname):`

   この関数は、指定された XA 接続の OCI サービス・ハンドルを戻します。dbname パラメータは、xa_open 文字列で渡された dbname パラメータと同じである必要があります。OCI アプリケーションでは、接続ハンドルを取得するために、sqlld2 コールのかわりにこのルーティングを使用できます。このため、OCI アプリケーションは SQLLIB ライブラリにリンクする必要はありません。サービス・ハンドルは、OCISvcCtxToLda()（OCI バージョン 8）を使用して OCI バージョン 7 のログイン・データ領域（LDA）に変換できます。クライアント・アプリケーションは、OCI コールを完了した後で、OCILdaToSvcCtx() を使用して Oracle7 の LDA をサービス・ハンドルに変換する必要があります。

2. `OCIEnv *xaoEnv(text *dbname):`

   この関数は、指定された XA 接続の OCI 環境ハンドルを戻します。dbname パラメータは、xa_open 文字列で渡された dbname パラメータと同じである必要があります。

3. `int xaosterr(OCISvcCtx *SvcCtx, sb4 error):`

   動的登録にのみ適用できるこの関数は、Oracle エラー・コードを XA エラー・コードに変換します。最初のパラメータは、データベースで作業を実行するために使用するサービス・ハンドルです。2 番目のパラメータは、Oracle から戻されるエラー・コードです。この関数は、OCI コマンドから戻されたエラーが xa_start の違反によって発生したものかどうかを判断するために使用します。この関数は、エラーが XA モジュールによって生成されたものでない場合は XA_OK を戻し、エラーが XA モジュールによって生成されたものである場合は有効な XA エラーを戻します。

XA ライブラリを使用するアプリケーションの開発およびインストール

この項では、Oracle XA アプリケーションの開発およびインストールについて説明します。

DBA またはシステム管理者の責任

DBA またはシステム管理者の責任は次のとおりです。

1. アプリケーション開発者の支援によってオープン文字列を定義します。

   20-10 ページの「xa_open 文字列の定義」を参照してください。

2. DBA_PENDING_TRANSACTIONS ビューがデータベースに存在していることを確認します。

   Oracle8 の場合 :

   xa_open 文字列で指定したすべての Oracle サーバー・ユーザーに対して、DBA_PENDING_TRANSACTIONS ビューの SELECT 権限を付与します。

   Oracle7 リリース 7.3 の場合 :

   V$XATRANS$ が存在することを確認します。

   このビューは、XA ライブラリのインストール時に作成されています。必要な場合は、SQL スクリプト XAVIEW.SQL を実行することによって、ビューを手動で作成できます。この SQL スクリプトは、Oracle ユーザー SYS として実行する必要があります。Oracle XA ライブラリ・アプリケーションが使用するすべての Oracle サーバー・アカウントに対して、V$XATRANS$ ビューの SELECT 権限を付与します。

   参照 : XAVIEW.SQL スクリプトの位置については、ご使用のオペレーティング・システム固有の Oracle マニュアルを参照してください。

3. TPM ベンダーの指示に従い、オープン文字列情報を使用して RM を TPM 構成にインストールします。

DBA またはシステム管理者は、Oracle データベース・サーバーに接続するプロセスを TPM システムが開始することを認識する必要があります。TPM のマニュアルを参照して、このプロセスのためにどのような環境が存在するのか、ユーザー ID は何であるかを判断してください。

ORACLE_HOME および ORACLE_SID に正しい値が設定されていることを確認してください。

> **参照：** デフォルトとは異なる sid またはトレース・ディレクトリを指定する方法の詳細は、20-10 ページの「xa_open 文字列の定義」を参照してください。

また、このユーザーには、必ず DBA_PENDING_TRANSACTIONS に対する SELECT 権限を付与してください。

4. Oracle XA アプリケーションをオンラインにするために、適切なデータベースを起動します。

   この処理は、TPM サーバーを開始する前に行ってください。

アプリケーション開発者の責任

アプリケーション開発者の責任は次のとおりです。

1. DBA またはシステム管理者の支援によってオープン文字列を定義します。

   オープン文字列の定義方法については、この項で後述します。

2. アプリケーションを開発します。

   プリコンパイラ用のトランザクション指向の SQL 文に関する特別な制限事項を守ってください。

   > **参照：** 20-17 ページの「プリコンパイラと OCI の XA インタフェース」を参照してください。

3. TPM ベンダーの指示に従ってアプリケーションをリンクします。

xa_open 文字列の定義

オープン文字列は、トランザクション・モニターがデータベースをオープンするために使用します。オープン文字列内の最大文字数は 256 文字です。

この項の内容は次のとおりです。

xa_open 文字列の構文

`Oracle_XA{+required_fields...} [+optional_fields...]`

required_fields は次のいずれかです。

`Acc=P//`

または

`Acc=P/user/password`

`SesTm=session_time_limit`

optional_fields は次のとおりです。

`DB=db_name`

`LogDir=log_dir`

`MaxCur=maximum_#_of_open_cursors`

`Objects=true/false`

`SqlNet=connect_string`

`Loose_Coupling=true/false`

`SesWt=session_wait_limit`

`Threads=true/false`

注意：

- オープン文字列の作成時には、必須フィールドおよびオプションのフィールドをどのような順序でも入力できます。
- すべてのフィールド名は大 / 小文字が区別されません。その値の大 / 小文字が区別されるかどうかは、プラットフォームによって異なります。
- 実際の情報文字列の一部に「+」文字は使用できません。

必須のフィールド

この項では、オープン文字列の必須フィールドについて説明します。

Acc=P//

または

Acc=P/*user/password*

構文要素	説明
Acc	ユーザー・アクセス情報を指定します。
P	明示的なユーザーおよびパスワード情報が提供されることを示します。
P//	ユーザーおよびパスワード情報が明示的には提供されないこと、およびオペレーティング・システム認証フォームが使用されることを示します。 詳細は、『Oracle9*i* データベース管理者ガイド』を参照してください。
user	有効な Oracle サーバー・アカウントです。
password	対応する現行パスワードです。

たとえば、Acc=P/scott/tiger は、ユーザーおよびパスワード情報が提供されることを示します。この場合、ユーザーは scott で、パスワードは tiger です。

前述のとおり、scott に DBA_PENDING_TRANSACTIONS 表の SELECT 権限があることを確認してください。

Acc=P// は、ユーザーおよびパスワード情報が提供されないことと、そのためにデフォルトとしてオペレーティング・システムの認証が使用されることを示します。

SesTm=*session_time_limit*

構文要素	説明
SesTm	システムによって自動的に異常終了されるまで、トランザクションがアクティブでない状態でいられる最大時間を指定します。
session_time_limit	この値は、トランザクション内で、あるサービスとその次のサービスとの間、またはあるサービスとトランザクションのコミットまたはロールバックとの間で許可される最大時間にする必要があります。 たとえば、TPM でクライアントとサーバー間にリモート・プロシージャ・コールが使用されている場合は、SesTM は、ある RPC の完了から次の RPC の初期化の間、tx_commit または tx_rollback との間に適用されます。 この時間の単位は秒です。値が 0（ゼロ）の場合は、制限がないことを示します。たとえば、SesTM=15 は、セッション・アイドル時間の上限が 15 秒であることを示します。 値 0（ゼロ）はできるだけ指定しないでください。問題があった場合に、リソースを長時間拘束することがあります。また、子プロセスに SesTM=0 がある場合、親プロセスが終了した後は、この設定の効果がなくなります。

オプションのフィールド

次にオプションのフィールドについて説明します。

DB=*db_name*

構文要素	説明
DB	データベース名を指定します。
db_name	Oracle プリコンパイラがデータベースの識別に使用する名前を示します。 Oracle プリコンパイラのデフォルトのデータベースのみを使用する（SQL 文で AT 句を使用しない）AP では、オープン文字列の DB=*db_name* 句を省略する必要があります。 明示的に指定したデータベースを使用するアプリケーションは、DB=*db_name* フィールドにそのデータベース名を指定する必要があります。 Oracle7 の OCI プログラムは、正しい lda_def（サービス・コンテキストと等価）を取得するために、sqlld2() 関数をコールする必要があります。Oracle8 の OCI プログラムでは、xaoSvcCtx 関数をコールして、OCISvcCtx サービス・コンテキストを取得する必要があります。 *db_name* は *sid* ではありません。オープンするデータベースを検索するためには使用されません。このオープン文字列でオープンされたデータベースと、SQL 文を実行するために AP で使用される名前とを対応付けます。*sid* は、TPM アプリケーション・サーバーの環境変数 ORACLE_SID か、オープン文字列の Oracle Net（以前の SQL*Net および Net8）句で指定した *sid* から設定されます。Oracle Net 句については、この項で後述します。 一部の TPM ベンダーは、同じオープン文字列を使用するサーバー・グループの名前を指定する方法を提供しています。DBA にとっては、グループ名と *db_name* の両方に対して同じ名前を選択する方が便利な場合があります。

たとえば、DB=payroll は、データベース名が payroll であり、アプリケーション・サーバー・プログラムがこの名前を AT 句で使用することを示しています。

LogDir=*log_dir*

構文要素	説明
LogDir	Oracle XA ライブラリのエラー情報およびトレース情報が記録されるローカル・マシン上のディレクトリを指定します。
log_dir	トレース情報が格納されるディレクトリのパス名を示します。デフォルトは、ORACLE_HOME が設定されている場合 $ORACLE_HOME/rdbms/log で、設定されていない場合は現在のディレクトリです。

たとえば、LogDir=/xa_trace は、/xa_trace ディレクトリにエラー情報およびトレース情報があることを示しています。

> **注意：** ログ用に指定したディレクトリが存在し、アプリケーション・サーバーでそのディレクトリに確実に書き込めるようにしてください。

Loose_Coupling=*true/false*

詳細は、20-30 ページの「トランザクション・ブランチ」を参照してください。

Objects=*true/false*

構文要素	説明
Objects	アプリケーション・プロセスをオブジェクト・モードで初期化するかどうかを指定します。デフォルト値は false です。
true/false	アプリケーションが、OCIAssignRawbytes() などのオブジェクト・モードを必要とする特定の API コールを使用する必要がある場合、true を指定します。

MaxCur=*maximum_#_of_open_cursors*

構文要素	説明
MaxCur	データベースのオープン時に割り当てられるカーソルの数を指定します。これは、プリコンパイラ・オプション maxopencursors と同じ用途で機能します。
maximum_#_of_open_cursors	キャッシュするオープン・カーソルの数を示します。

たとえば、MaxCur=5 は、プリコンパイラで 5 つのオープン・カーソルがキャッシュされることを示します。

> **注意：** このパラメータは、ソース・コード内またはコンパイル時に指定したプリコンパイラ・オプション maxopencursors をオーバーライドします。

> **参照：** 『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

SqlNet=*db_link*

構文要素	説明
SqlNet	Oracle Net（以前の SQL*Net および Net8）データベース・リンクを指定します。
db_link	システムにログインするために使用する文字列を示します。この文字列の構文は、環境変数 TWO_TASK を設定するために使用する構文と同じです。

たとえば、SqlNet=hqfin@NEWDB は、ホスト hqfin で TCP/IP によってアクセスされる *sid*=NEWDB のデータベースを示します。

サーバー環境変数を制御できない場合は、SqlNet パラメータを使用して ORACLE_SID を指定できます。また、サーバーが複数の Oracle サーバー・データベースにアクセスする必要があるときにも、このパラメータを使用する必要があります。リモート・データベースに実際にはアクセスしないで Oracle Net 文字列を使用するには、パイプ・ドライバを使用します。

次に例を示します。

```
SqlNet=localsid1
```

localsid1 は、tnsnames.ora ファイルで定義されている別名です。

Oracle Net データベース・リンクでアクセスされるすべてのデータベースが、/etc/oratab 内にエントリを持つようにしてください。

SesWt=*session_wait_limit*

構文要素	説明
SesWt	別のセッションに使用されているトランザクション・ブランチを待機するときのタイムアウト限度を指定します。デフォルト値は 60 秒です。
session_wait_limit	XA_RETRY が戻されるまで Oracle が待機する秒数を指定します。

Threads=*true/false*

構文要素	説明
Threads	アプリケーションがマルチスレッドかどうかを指定します。デフォルト値は false です。
true/false	アプリケーションがマルチスレッドである場合、設定は *true* です。

プリコンパイラと OCI の XA インタフェース

この項では、プリコンパイラおよび OCI とともに Oracle XA ライブラリを使用する方法について説明します。

Oracle XA ライブラリとプリコンパイラの使用

カーソルは、Oracle XA アプリケーションで使用する場合はトランザクションの存続期間のみ有効です。明示的カーソルは、トランザクションが開始した後にオープンし、コミットまたはロールバックの前にクローズする必要があります。

プリコンパイラとのインタフェースでは、次の 2 つのオプションのどちらかを選択します。

- デフォルトのデータベースでのプリコンパイラの使用
- 指定されたデータベースでのプリコンパイラの使用

次の例では、プリコンパイラ Pro*C/C++ を使用しています。

デフォルトのデータベースでのプリコンパイラの使用

デフォルトのデータベースでプリコンパイラとインタフェースするには、オープン文字列で使用される DB=*db_name* フィールドが存在しないことを確認してください。このフィールドが存在しない場合は、デフォルトの接続が指定され、プロセスごとに 1 つのデフォルト接続のみが使用できます。

次に、デフォルトの Pro*C/C++ 接続を識別するオープン文字列の例を示します。

```
ORACLE_XA+SqlNet=host@MAIL+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/logs
```

DB=d*b_name* が存在せず、空のデータベース ID 文字列を示していることに注意してください。

SQL 文の構文は次のようになります。

```
EXEC SQL UPDATE Emp_tab SET Sal = Sal*1.5;
```

指定されたデータベースでのプリコンパイラの使用

指定されたデータベースでプリコンパイラにインタフェースするには、オープン文字列に DB=*db_name field* を含めます。参照するすべてのデータベースは、対応するオープン文字列で指定した同一の *db_name* を参照する必要があります。

アプリケーションには、次の例に示すように、デフォルト・データベースの他に、名前を指定されたデータベースが 1 つ以上含まれることがあります。

たとえば、あるデータベースで従業員の給与を更新し、別のデータベースでその従業員の部門番号（DEPTNO）を、第 3 のデータベースでその従業員の管理者を更新するとします。このような場合は、TM に次のようなオープン文字列を構成します。

```
ORACLE_XA+DB=MANAGERS+SqlNet=hqfin@SID1+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+DB=PAYROLL+SqlNet=SID2+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
ORACLE_XA+SqlNet=hqemp@SID3+ACC=P/scott/tiger
  +SesTM=10+LogDir=/usr/local/xalog
```

最後のオープン文字列に DB=*db_name* フィールドがないことに注意してください。

アプリケーション・サーバー・プログラムでは、次のような宣言を入力します。

```
EXEC SQL DECLARE PAYROLL DATABASE;
EXEC SQL DECLARE MANAGERS DATABASE;
```

ここでも、デフォルトの接続（db_name フィールドを含まない 3 番目のオープン文字列に対応）には宣言は必要ありません。

更新を実行するときは、次に示すような文を入力します。

```
EXEC SQL AT PAYROLL UPDATE Emp_Tab SET Sal=4500 WHERE Empno=7788;
EXEC SQL AT MANAGERS UPDATE Emp_Tab SET Mgr=7566 WHERE Empno=7788;
EXEC SQL UPDATE Emp_Tab SET Deptno=30 WHERE Empno=7788;
```

最後の文は、デフォルトのデータベースを参照しているため AT 句はありません。

Oracle プリコンパイラのリリース 1.5.3 以上では、次に示す例のように、AT 句に文字ホスト変数を使用できます。

```
EXEC SQL BEGIN DECLARE SECTION;
  DB_NAME1 CHARACTER(10);
  DB_NAME2 CHARACTER(10);
EXEC SQL END DECLARE SECTION;
    ...
SET DB_NAME1 = 'PAYROLL'
SET DB_NAME2 = 'MANAGERS'
    ...
EXEC SQL AT :DB_NAME1 UPDATE...
EXEC SQL AT :DB_NAME2 UPDATE...
```

注意： Oracle では、接続の作成に XA アプリケーションを使用することはお薦めしません。実行されるすべての作業は、グローバル・トランザクションの範囲から外れることになり、個別にコミットする必要があります。

Oracle XA ライブラリと OCI の使用

Oracle XA ライブラリを使用する OCI アプリケーションでは、RM にログインするために OCISessionBegin()（バージョン 7 の olon() または orlon()）をコールしないでください。ログインは、TPM を介して行うようにしてください。そのようなアプリケーションでは、関数 xaoSvcCtx()（バージョン 7 の sqlld2()）を実行して、RM のアクセスに必要なサービス・コンテキスト（バージョン 7 の lda）構造体を取得できます。

環境ハンドルを OCI 関数に渡す必要があるアプリケーションでは、そのハンドルを検索するために xaoEnv() もコールする必要があります。

アプリケーション・サーバーは同時に複数の Oracle サーバーの RM をオープンできるため、適切なサービス・コンテキストを取得するために適切な引数を使用して関数 xaoSvcCtx() をコールする必要があります。

リリース 7.3 の場合

DB=*db_name* がオープン文字列にない場合は、次のように実行します。

```
sqlld2(lda, NULL, 0);
```

これで、この RM の lda が取得されます。

DB=*db_name* がオープン文字列に存在する場合は、次のように実行します。

```
sqlld2(lda, db_name, strlen(db_name));
```

これで、この RM の lda が取得されます。

リリース 8.0 の場合

DB=*db_name* がオープン文字列にない場合は、次のように実行します。

```
xaoSvcCtx(NULL);
```

DB=*db_name* がオープン文字列に存在する場合は、次のように実行します。

```
xaoSvcCtx(db_name);
```

これで、この RM のサーバー・コンテキストが取得されます。

同様に、次のように実行します。

```
xaoEnv(NULL);
```

または

```
xaoEnv(db_name);
```

オープン文字列によっては、このようにして環境ハンドルを取得します。

> **参照：** OCISvcCtx の使用の詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

XA を使用したトランザクション制御

この項では、Oracle XA ライブラリ環境内でトランザクション制御を使用する方法について説明します。

XA ライブラリを使用する場合、トランザクションは、トランザクションをコミットまたはロールバックする SQL 文によっては制御されません。制御は、トランザクションを開始および終了する TM に受け入れられる API によって行われます。次に示す XA 関数ではなく、TM によって定義された API をコールします。

TM は、通常、TX インタフェースを介してトランザクションを制御します。TX インタフェースには次の関数が含まれています。

TX 関数	説明
tx_open	RM にログインします。
tx_close	RM からログアウトします。
tx_begin	新規トランザクションを開始します。
tx_commit	トランザクションをコミットします。
tx_rollback	トランザクションをロールバックします。

ほとんどの TPM アプリケーションは、アプリケーション・クライアントがサービスを要求し、アプリケーション・サーバーがサービスを提供するというクライアント / サーバー・アーキテクチャを使用して作成されています。次に示す例では、そのようなクライアント / サーバー・モデルを使用しています。サービスとは論理作業単位であり、Oracle サーバーが RM である場合は、関連する作業単位を実行する一連の SQL 文で構成されます。

たとえば、「貸方記入」というサービスが口座番号および貸方記入額を受け取ると、このサービスは、データベース内の特定の表にある情報を更新する SQL 文を実行します。さらに、サービスはその他のサービスを要求することもあります。たとえば、「振替」サービスの場合は、「貸方記入」および「借方記入」にサービスを要求します。

通常、アプリケーション・クライアントは、トランザクション内で作業を実行するためにアプリケーション・サーバーにサービスを要求します。ただし、一部の TPM システムでは、アプリケーション・クライアント自身がローカルにサービスを提供できます。

例に示すとおり、トランザクション制御文は、クライアントまたはサーバーのどちらの側でも記述できます。

複数のプロセスを同一のトランザクションに参加させるために、TPM は参加プロセス間でトランザクション情報が伝達できる通信 API を提供しています。通信 API の例として、RPC、疑似 RPC 関数、送信 / 受信関数があります。

主要ベンダーが異なる通信関数をサポートしているため、次の例では、通信 API を汎用化するために通信疑似関数 tpm_service を使用しています。

X/Open の準備段階の仕様には、通信関数を提供する代替方法がいくつか取り入れられています。主要 TPM ベンダーでは、これらの代替方法を 1 つ以上はサポートしています。

プリコンパイラ・アプリケーションの例

次の例は、プリコンパイラ・アプリケーションを示しています。アプリケーション・サーバーは、TPM 固有の方法で TPM システムにすでにログインしているとします。

最初の例は、アプリケーション・サーバーによって開始されるトランザクションを示し、2 番目の例は、アプリケーション・クライアントによって開始されるトランザクションを示します。

例 1: アプリケーション・サーバーによって開始されるトランザクション

クライアント :

```
tpm_service("ServiceName");              /*Request Service*/
```

サーバー :

```
ServiceName()
{
<get service specific data>
tx_begin();                              /* Begin transaction boundary*/
EXEC SQL UPDATE ....;

/*This application server temporarily becomes*/
/*a client and requests another service.*/

tpm_service("AnotherService");
tx_commit();                              /*Commit the transaction*/
<return service status back to the client>
}
```

例 2: アプリケーション・クライアントによって開始されるトランザクション

クライアント :

```
tx_begin();                             /* Begin transaction boundary */
tpm_service("Service1");
tpm_service("Service2");
tx_commit();                            /* Commit the transaction */
```

サーバー :

```
Service1()
{
<get service specific data>
EXEC SQL UPDATE ....;
<return service status back to the client>
}
Service2()
{
```

```
<get service specific data>
EXEC SQL UPDATE ....;
...
<return service status back to client>
}
```

プリコンパイラまたは OCI アプリケーションの TPM アプリケーションへの移行

既存のプリコンパイラまたは OCI アプリケーションを、Oracle XA ライブラリを使用して TPM アプリケーションに移行するには、次の手順に従います。

1. アプリケーションを「サービス」のフレームワークに再編成します。

   これは、アプリケーション・クライアントがアプリケーション・サーバーにサービスを要求することを意味します。

   TPM には、アプリケーションで tx_open 関数および tx_close 関数を使用するように要求する TPM と、ログインおよびログオフを暗黙的に実行する TPM があります。

   オープン文字列に sqlnet パラメータを指定しないと、アプリケーションはデフォルトの Oracle Net ドライバを使用します。このため、環境変数 ORACLE_HOME および ORACLE_SID を正しく定義して、アプリケーション・サーバーを立ち上げる必要があります。これは、TPM に固有の方法で行われます。これを行う方法の詳細は、TPM ベンダーのマニュアルを参照してください。

2. アプリケーションで正規の接続文および切断文を置き換えます。

   たとえば、接続文 EXEC SQL CONNECT（プリコンパイラの場合）または OCISessionBegin()（OCI の場合）を tx_open() で置き換えます。切断文 EXEC SQL COMMIT/ROLLBACK WORK RELEASE（プリコンパイラの場合）または OCISessionEnd()（OCI の場合）を tx_close() で置き換えます。バージョン 7 では、OCISessionBegin() は olon() で、OCISessionEnd() は ologof() でした。

3. アプリケーションで正規のコミット / ロールバック文を置き換え、トランザクションを明示的に開始します。

   たとえば、tx_commit()/tx_rollback() によってコミット / ロールバック文 EXEC SQL COMMIT/ROLLBACK WORK（プリコンパイラの場合）または ocom()/orol()（OCI の場合）を置き換え、tx_begin() をコールすることによってトランザクションを開始します。

4. アプリケーションで、トランザクションを終了する前にフェッチ状態をリセットします。一般的には、release_cursor=no を使用します。release_cursor=yes は、文が 1 回しか実行されないことが確実なときにのみ使用します。

表 20-1 に、プリコンパイラまたは OCI アプリケーションを TPM アプリケーションに移行するときに、正規の Oracle コマンドを置き換える TPM 関数を示します。

表 20-1 TPM 用に置き換えるコマンド

正規の Oracle コマンド	TPM 関数
CONNECT *user/password*	tx_open（暗黙の可能性あり）
トランザクションの暗黙的な開始	tx_begin
SQL	SQL を実行するサービス
COMMIT	tx_commit
ROLLBACK	tx_rollback
切断	tx_close（暗黙の可能性あり）
SET TRANSACTION READ ONLY	無効

XA ライブラリ・スレッド・セーフティ

スレッドをサポートするトランザクション・モニターを使用する場合は、Oracle XA ライブラリを使用してスレッド・セーフなアプリケーションを作成できます。ただし、注意が必要な問題がいくつかあります。

制御のスレッド（またはスレッド）とは、RM への一連の接続のことです。スレッド化されていないシステムでは、各プロセスを制御のスレッドとみなすことができます。これは、各プロセスには RM への接続がそれぞれ独自にあり、それぞれが独立した RM 表をメンテナンスしているためです。

スレッド化されているシステムでは、各スレッドには RM との自律型接続があり、プライベートな RM 表をメンテナンスします。このプライベートな RM 表は新しいスレッドのそれぞれに対して割り当てる必要があり、またスレッドが終了したときは（異常終了であっても）割当てを解除する必要があります。

注意： Oracle システムでは、あるスレッドが開始して接続を確立した後、その接続を使用できるのはそのスレッドのみです。他のスレッドは、その接続上でコールを行うことはできません。

オープン文字列でのスレッドの指定

xa_open 文字列パラメータの xa_info では、Threads= 句が指定されます。トランザクション・モニターでスレッドを使用できるようにするには、この句を true に指定する必要があります。デフォルトは false です。ほとんどの場合、スレッドはトランザクション・モニターによって作成され、アプリケーションは新しいスレッドがいつ作成されたのか認識しないことに注意してください。このため、サービス・コンテキスト（バージョン 7 では lda）は、トランザクション・モニター・アプリケーション用に作成される各サービス内のスタック上に割り当てることをお薦めします。そのサービスで Oracle 関連のコールを行う場合は、その前に xaoSvcCtx（バージョン 7 の OCI では sqlld2）関数をコールして、サービス・コンテキストを初期化する必要があります。以降は、この LDA をそのサービス内のすべての OCI コールに使用できます。

XA のスレッドにおける制限事項

スレッドを使用するときは、次の制限事項が適用されます。

- 新規アプリケーション・スレッドがそれぞれいつ開始されるかが明示的にトランザクション・モニターに通知されないかぎり、トランザクション・モニター上でアプリケーション・サーバー・プロセスの一部として実行される Pro* または OCI コードをスレッド化することはできません。通常、これは、トランザクション・モニター・ベンダーが提供する特別な C コンパイラを使用して実現されます。
- Pro* 文 EXEC SQL ALLOCATE および EXEC SQL USE はサポートされません。このため、スレッド化が使用可能であるときは、埋込み SQL 文を非 XA 接続にまたがって使用することはできません。
- プロセスの 1 つのスレッドが XA を介して Oracle に接続した場合、そのプロセスの他のすべてのスレッドも XA を介して Oracle に接続する必要があります。1 つのスレッドで EXEC SQL を介して接続し、他のスレッドで XA を介して接続することはできません。

XA アプリケーションのトラブルシューティング

この項では、問題またはシステム障害の発生時に情報を検索する方法を説明します。また、トレース・ファイルおよびペンディング・トランザクションのリカバリについても説明します。

XA トレース・ファイル

Oracle XA ライブラリでは、すべてのエラーおよびトレース情報がトレース・ファイルに記録されます。この情報は、XA エラー・コードを補足するときに役立ちます。たとえば xa_open 障害が発生したときに、その原因としてオープン文字列が正しくなかったか、Oracle サーバー・インスタンスの検索が正常に実行されなかったか、またはログイン認可が正常に実行されなかったかを示します。

トレース・ファイルの名前は次のとおりです。

xa_*db_name**date*.trc

ここで、*db_name* はオープン文字列のフィールド DB=*db_name* で指定したデータベース名であり、*date* は情報がトレース・ファイルに記録された日付です。

オープン文字列で DB=*db_name* を指定しない場合、自動的にデフォルトで NULL という名前になります。

xa_open 文字列 DbgFl

通常、XA トレース・ファイルはエラーが検出された場合にのみオープンされます。xa_open 文字列 DbgFl は、XA ライブラリに関する詳細を記録するトレース機能を提供します。デフォルト値は 0（ゼロ）です。次の組合せのいずれかに設定できます。これらの組合せは独立しているため、複数のフラグを出力するには、それぞれを設定する必要があります。

- 0x1　XA インタフェース内の各プロシージャの入口および出口をトレースします。これは、TP モニターがどの XA コールを作成し、どのトランザクション識別子を生成しているかを正確に調べるときに役立ちます。
- 0x2　他の非公開 XA ライブラリ・ルーチンへの入口および出口をトレースします。通常、これは Oracle 開発者用の機能です。
- 0x4　OCI に対する専用コールのように、XA ライブラリが作成する他の様々な「関心のある」コールをトレースします。通常、これは Oracle 開発者用の機能です。

トレース・ファイルの位置

トレース・ファイルは次のいずれかの位置に作成することができます。

- トレース・ファイルは、オープン文字列で指定したとおりに、LogDir ディレクトリに作成できます。

- オープン文字列に LogDir を指定しないと、Oracle XA アプリケーションは、$ORACLE_HOME の位置を判断できる場合は $ORACLE_HOME/rdbms/log ディレクトリにトレース・ファイルを作成しようとします。
- Oracle XA アプリケーションで $ORACLE_HOME の位置が判断できない場合は、トレース・ファイルは現在の作業ディレクトリに作成されます。

トレース・ファイルの例

2 種類のトレース・ファイルの例を次に説明します。

例 xa_NULL04021992.trc は、1992 年 4 月 2 日に作成されたトレース・ファイルを示しています。RM がオープンされたときに、DB フィールドがオープン文字列に指定されていませんでした。

例 xa_Finance12151991.trc は、1991 年 12 月 15 日に作成されたトレース・ファイルを示しています。RM がオープンされたときに、DB フィールドはオープン文字列で「Finance」と指定されていました。

注意： オープン文字列に同一の DB フィールドおよび LogDir フィールドを持つ複数の Oracle XA ライブラリ RM は、同じ日に発生したすべてのトレース情報を同じトレース・ファイルに記録します。

トレース・ファイル内の各入口には、次のような情報が含まれています。

```
1032.12345.2:  ORA-01017:  invalid username/password;  logon denied
1032.12345.2:  xaolgn:  XAER_INVAL;  logon denied
```

この場合、「1032」は情報がログされたときの時刻、「12345」はプロセス ID（PID）、「2」は RM の ID、xaolgn はモジュール名、XAER_INVAL は XA 標準仕様どおりに戻されたエラー、ORA-01017 は戻された Oracle サーバー情報です。

インダウト・トランザクションまたはペンディング・トランザクション

インダウト・トランザクションまたはペンディング・トランザクションとは、準備はできているがデータベースに対してコミットされていないトランザクションです。

一般的に、TPM システムが提供する TM は、インダウト・トランザクションまたはペンディング・トランザクションの障害を解決し、リカバリします。ただし、インダウト・トランザクションが次のような状態であるときは、DBA がインダウト・トランザクションをオーバーライドする必要がある場合があります。

- 他のトランザクションが必要とするデータをロックしている場合
- 適切な時間の経過後も解決されない場合

前述のような状況でインダウト・トランザクションをオーバーライドする方法の詳細、またはインダウト・トランザクションをコミットまたはロールバックするかどうかを決定する方法の詳細は、TPM のマニュアルを参照してください。

Oracle サーバーの SYS アカウント表

Oracle サーバーの SYS アカウントには 4 つの表があり、正規の Oracle サーバー・アプリケーションおよび Oracle XA アプリケーションによって生成されたトランザクションが含まれています。4 つの表とは、DBA_PENDING_TRANSACTIONS、V$GLOBAL_TRANSACTIONS、DBA_2PC_PENDING および DBA_2PC_NEIGHBORS です。

Oracle XA アプリケーションによって生成されたトランザクションの場合は、次に示す列情報が DBA_2PC_NEIGHBORS 表に適用されます。

- DBID 列は常に xa_orcl
- DBUSER_OWNER 列は常に *db_name*xa.oracle.com

db_name は、オープン文字列で常に DB=*db_name* と指定されることに注意してください。オープン文字列にこのフィールドを指定しないと、この列の値は Oracle XA アプリケーションが生成するトランザクションについては NULLxa.oracle.com になります。

たとえば、次の SQL 文を使用すると、Oracle XA アプリケーションによって生成されたインダウト・トランザクションの詳細情報を取得できます。

```
SELECT * FROM Dba_2pc_pending p, Dba_2pc_neighbors n
   WHERE p.Local_tran_id = n.Local_tran_id
      AND
      n.Dbid = 'xa_orcl';
```

別の方法として、TPM が使用するフォーマット ID がわかっている場合は、DBA_PENDING_TRANSACTIONS または V$GLOBAL_TRANSACTIONS を使用できます。DBA_PENDING_TRANSACTIONS ではアクティブな準備済トランザクションおよび正常に実行されなかった準備済トランザクションの両方のリストが提供されますが、V$GLOBAL_TRANSACTIONS では、すべてのアクティブなグローバル・トランザクションのリストが提供されます。

XA の問題および制限事項

データベース・リンク

Oracle XA アプリケーションは、次の制限事項に違反しないかぎり、データベース・リンク経由で他の Oracle サーバー・データベースにアクセスできます。

- 共有サーバー（マルチスレッド・サーバーともいう）構成を使用する必要があります。

  これは、TPM が共有サーバーを使用して Oracle への接続をオープンすることを意味します。データベース・リンクに必要なオペレーティング・システム・ネットワーク接続

は、Oracle サーバー・プロセスではなくディスパッチャによってオープンされます。このため、特定のサービスまたは RPC が完了すると、他のサービスまたは RPC で使用できるようにトランザクションをサーバーから連結解除できます。

- 他のデータベースへのアクセスには、SQL*Net バージョン 2、Net8 または Oracle Net を使用する必要があります。
- アクセスする他のデータベースは、別の Oracle サーバー・データベースである必要があります。

これらの制限事項が満たされている場合、Oracle サーバーはデータベース・リンクを許可し、トランザクション・プロトコルを他の Oracle サーバー・データベースに伝播（準備、ロールバックおよびコミット）します。

注意： これらの制限事項が満たされていない場合、XA トランザクション内でデータベース・リンクを使用すると、TPM サーバー・プロセスに接続されている Oracle Server 内に O/S ネットワーク接続が作成されます。この O/S ネットワーク接続はプロセス間で移動できないため、このサーバーから連結解除することはできません。データベース・リンクを介してデータベースにアクセスしたときに、ORA-24777 エラーが戻されます。

共有サーバー構成を使用できない場合は、リモート・データベースに EXEC SQL AT 構文を使用して Pro*C/C++ アプリケーション経由でアクセスします。

パラメータ open_links_per_instance は、移行可能なオープン・データベース・リンク接続の数を指定します。これらの dblink 接続は、トランザクションのコミット後に接続をキャッシュできるように、XA トランザクションによって使用されます。接続を作成したユーザーがトランザクションを作成したユーザーと同じである場合は、別のトランザクションが自由に dblink 接続を使用できます。このパラメータは、セッションからの dblink 接続数である open_links パラメータとは異なります。open_links パラメータは、XA アプリケーションには適用できません。

Oracle Real Application Clusters

障害トランザクションは、Oracle Real Application Clusters のどのインスタンスからでもリカバリできます。インダウト・トランザクションのヒューリスティックな（試行錯誤的な）コミットも、どのインスタンスからでもできます。XA リカバリ・コールによって、すべてのインスタンスについて準備されたすべてのトランザクションのリストが提供されます。

SQL に基づく制限事項

ロールバックおよびコミット

グローバル・トランザクションの進行状況の調整および監視は TM に責任があるため、グローバル・トランザクションのロールバックまたはコミットを独立して行う Oracle サー

バー固有の文をアプリケーションに入れないでください。ただし、ローカル・トランザクションではロールバックおよびコミットを使用できます。

グローバル・トランザクションの途中で、プリコンパイラ・アプリケーションに EXEC SQL ROLLBACK WORK を使用しないでください。同様に、OCI アプリケーションでは OCITransRollback() またはバージョン 7 で同等の orol() を実行しないでください。グローバル・トランザクションをロールバックするには、tx_rollback() をコールします。

同様に、グローバル・トランザクションの途中で、プリコンパイラ・アプリケーションに EXEC SQL COMMIT WORK 文を使用しないでください。OCI アプリケーションでは、OCITransCommit() またはバージョン 7 で同等の ocom() を実行しないでください。かわりに、tx_commit() または tx_rollback() を使用して、グローバル・トランザクションを終了してください。

DDL 文

CREATE TABLE などの SQL DDL 文は暗黙的なコミットを意味するため、Oracle XA アプリケーションで SQL DDL 文を実行できません。

セッション状態

Oracle では、セッション状態が複数のサービス間で有効であることを保証していません。たとえば、あるサービスがセッション変数（グローバル・パッケージ変数など）を更新した場合、同じグローバル・トランザクションの一部として実行している別のサービスにはこの変更が認識されないことがあります。セーブポイントは 1 つのサービス内で使用してください。アプリケーションが、別のサービスで作成されたセーブポイントを参照しないようにしてください。同様に、アプリケーションが、別のサービスで実行されたカーソルからフェッチしないようにしてください。

SET TRANSACTION

SET TRANSACTION READ ONLY | READ WRITE | USE ROLLBACK SEGMENT SQL 文を使用しないでください。

EXEC SQL を使用した接続または切断

接続および切断に EXEC SQL コマンド（EXEC SQL COMMIT WORK RELEASE または EXEC SQL ROLLBACK WORK RELEASE）を使用しないでください。

トランザクション・ブランチ

同じグローバル・トランザクション内の複数の Oracle サーバー・トランザクション・ブランチは、密結合または疎結合のどちらの方法でもロックを共有できます。ただし、Oracle Real Application Clusters の実行時に、各ブランチが異なるインスタンス上に存在している場合は、ブランチは疎結合になります。

密結合トランザクション・ブランチでは、ロックはトランザクション・ブランチ間で共有されます。あるトランザクション・ブランチで実行された更新は、更新がコミットされる前に同じグローバル・トランザクションに属している他のブランチでも認識できます。Oracle

サーバーは、密結合ブランチで文を実行する前に DX ロックを取得します。疎結合トランザクション・ブランチを使用するメリットは、並行性が増すということです（文が実行される前にロックが取得されないため）。デメリットは、すべてのトランザクション・ブランチが 2 フェーズ・コミットを実行する必要があることです。XA の 1 フェーズ最適化は使用できません。表 20-2 に、これらの密結合ブランチと疎結合ブランチ間のトレードオフを示します。

表 20-2 密結合および疎結合トランザクション・ブランチ

属性	密結合ブランチ	疎結合ブランチ
2 フェーズ・コミット	読込み専用最適化 [全ブランチについて準備、最後のブランチについてコミット]	2 フェーズ [全ブランチについて準備およびコミット]
シリアル化	データベース・コール	なし

対応付けの移行

Oracle サーバーでは、対応付けの移行はサポートされません（対応付けの移行とは、TM が、中断中のブランチ対応付けを別のブランチで再開する手段です）。

非同期コール

XA のオプションの機能である非同期 XA コールはサポートされません。

初期化パラメータ

`transactions` `init.ora` パラメータを、グローバル・トランザクションの同時実行予測数に設定します。

パラメータ `open_links_per_instance` は、移行可能なオープン・データベース・リンク接続の数を指定します。これらの dblink 接続は、トランザクションのコミット後に接続をキャッシュできるように、XA トランザクションによって使用されます。

参照： 20-28 ページの「データベース・リンク」を参照してください。

スレッドごとの最大接続数

スレッドごとの `xa_opens` の最大数は、現在は 32 です。以前は 8 でした。

インストール

XA を使用するためにスクリプトを実行する必要はありません。ただし、Oracle8 以上のサーバーでリリース 7.3 アプリケーションを実行するには、`xaview.sql` スクリプトを実行する必要があります。XA インタフェースを介して Oracle に接続するすべてのユーザーに、`SYS.DBA_PENDING_TRANSACTIONS` に対する `SELECT` 権限を付与してください。

ライブラリの互換性

リリース 7.3 で提供されている XA ライブラリは、リリース 8.0 以上の Oracle サーバーでも使用できます。リリース 7.2 の XA ライブラリは、リリース 7.2 の Oracle サーバーで使用する必要があります。リリース 8.0 のライブラリはリリース 7.3 の Oracle サーバーでも使用できます。下位互換性が保たれるのは 1 つのケースのみです。リリース 8.0 の OCI を使用する XA アプリケーションをリリース 7.3 の Oracle サーバーで動作させることができますが、これは、SQL 文を実行する前に `sqlld2` を使用して `lda_def` を取得する場合のみです。クライアント・アプリケーションは、OCI コールを完了した後で、`OCILdaToSvcCtx()` を使用して Oracle7 の LDA をサービス・ハンドルに変換する必要があります。

Oracle8 は、7.1.6 XA のコールをサポートしません（7.3 XA のコールはサポートします）。7.1.6 XA のコールは、Tuxedo アプリケーションを Oracle8 XA ライブラリに再リンクする必要があります。

ライブラリのリンク順序

TP モニター XA アプリケーションを構築する場合、リンク・ラインで、TP モニター・ライブラリ（`ax_reg` および `ax_unreg` 記号を定義する）を Oracle クライアント共有ライブラリの前に指定する必要があります。プラットフォームで共有ライブラリがサポートされていない場合、またはリンカーがリンク・ライン内のライブラリの順序を区別しない場合は、Oracle の非共有ライブラリを使用します。これらのリンク制限は、XA の動的登録（Oracle XA スイッチ `xaoswd`）を使用する場合のみ適用できます。

Oracle XA サポートの変更

リリース 8.0 からリリース 8.1 への XA の変更点

リリース 8.1 での変更点はありません。

リリース 7.3 からリリース 8.0 への XA の変更点

次に示す変更が追加されています。

- セッションの変更は不要
- 動的な登録のサポート
- 疎結合トランザクション・ブランチのサポート
- OCI アプリケーションに SQLLIB は不要
- XA を実行するためのインストール・スクリプトが不要
- すべてのプラットフォームで XA ライブラリを Oracle Real Application Clusters オプションとともに使用可能

- Oracle Real Application Clusters のトランザクション・リカバリの改善
- グローバル・トランザクションおよびローカル・トランザクションの両方が使用可能
- xa_open 文字列の変更

セッションの変更は不要

セッション・キャッシュは、新しい OCI では不要です。このため、古い xa_open 文字列パラメータ、SesCacheSz がなくなりました。その結果、init.ora の *sessions* パラメータを削減できます。かわりに、init.ora の *transactions* パラメータを同時グローバル・トランザクションの同時実行予測数に設定します。グローバル・トランザクションの再開時にセッションは移行されないため、アプリケーションではサービスの範囲を超えたセッション状態を参照しないようにする必要があります。

アプリケーションをいくつかのサービスに編成する方法の詳細は、TPM に付属のマニュアルを参照してください。特に、トランザクションが一時停止されると、セーブポイントとカーソル・フェッチ状態が取り消されます。これは、2 つのサービスが同じグローバル・トランザクションに属していたとしても、片方のサービス内のアプリケーションによって使用されるセーブポイントが、もう一方のサービスでは無効になるということです。

動的な登録のサポート

XA アプリケーションおよび Oracle サーバーがともにバージョン 8 である場合に、動的な登録を使用できます。

参照： 20-8 ページの「XA インタフェースの拡張」を参照してください。

疎結合トランザクション・ブランチのサポート

Oracle8 サーバー以上では、単一の Oracle インスタンスにおいて疎結合と密結合の両方のトランザクション・ブランチを使用できます。Oracle7 サーバーの場合、シングル・インスタンスでは密結合トランザクション・ブランチのみがサポートされ、疎結合トランザクション・ブランチは別のインスタンスでサポートされていました。

OCI アプリケーションに SQLLIB は不要

これまでは、OCI アプリケーションで SQLLIB を使用する必要がありました。つまり、Pro* アプリケーションを開発する必要がない場合でも、OCI プログラマは、SQLLIB を購入する必要がありました。今後は、その必要がなくなります。

XA を実行するためのインストール・スクリプトが不要

Oracle8 での XA アプリケーションの実行には SQL スクリプト XAVIEW.SQL は必要ありません。ただし、XAVIEW.SQL はリリース 7.3 アプリケーションには必要です。

参照： 20-9 ページの「DBA またはシステム管理者の責任」を参照してください。

すべてのプラットフォームで XA ライブラリを Oracle Real Application Clusters オプションとともに使用可能

Oracle7 では、特定のプラットフォームにおいて Oracle Real Application Clusters オプションとともに Oracle XA ライブラリを使用することができませんでした（プラットフォームでの分散ロック・マネージャの実装が、プロセス・ベースではなくトランザクション・ベースのロックをサポートしていないと、Oracle Real Application Clusters オプションと Oracle XA ライブラリを一緒に使用できませんでした）が、この制限はなくなりました。Oracle Real Application Clusters オプションを実行できる場合は、Oracle XA ライブラリも実行できます。

Oracle Real Application Clusters のトランザクション・リカバリの改善

すべてのトランザクションは、Oracle Real Application Clusters のどのインスタンスからでもリカバリできます。ペンディング・トランザクションのスナップショットの指定には、`xa_recover` コールを使用します。

グローバル・トランザクションおよびローカル・トランザクションの両方が使用可能

同一 XA 接続内で、グローバル・トランザクションおよびローカル・トランザクションの両方を使用できるようになりました。ローカル・トランザクションとは、Oracle サーバーによって完全に調整されるトランザクションです。たとえば、次に示す更新処理はローカル・トランザクションに属します。

```
CONNECT scott/tiger;
UPDATE Emp_tab SET Sal = Sal + 1; /* begin local transaction*/
COMMIT;                           /* commit local transaction*/
```

これに対してグローバル・トランザクションは、TPM などの外部の TM によって調整されます。グローバル・トランザクションでは、Oracle は従属的に動作し、TM が発行する XA コマンドを処理します。次に示す更新処理はグローバル・トランザクションに属します。

```
xa_open(oracle_xa+acc=p/SCOTT/TIGER+sestm=10", 1, TMNOFLAGS);
                            /* Transaction manager opens */
                            /* connection to the Oracle server*/
tpbegin();                  /* begin global transaction, the transaction*/
                            /* manager issues XA commands to the oracle*/
                            /* server to start a global transaction */
UPDATE Emp_tab SET Sal = Sal + 1;
                            /* Update is performed in the */
                            /* global transaction*/
tpcommit();                 /* commit global transaction, */
                            /* the transaction manager issues XA commands*/
```

```
                              /* to the Oracle server to commit */
                              /* the global transaction */
```

Oracle7 サーバーでは、ローカル・トランザクションが XA 接続内で開始されるのを禁止しています。次に示す更新処理は ORA-02041 エラー・コードを戻します。

```
xa_open("oracle_xa+acc=p/SCOTT/TIGER+sestm=10" , 1, TMNOFLAGS);
                              /* Transaction manager opens */
                              /*connection to the Oracle server */
UPDATE Emp_tab SET Sal = Sal + 1; /* Oracle 7 returns an error */
```

Oracle8 サーバー以上では、XA 接続内でローカル・トランザクションを開始できます。唯一の制限事項は、接続内でグローバル・トランザクションを開始する前に、ローカル・トランザクションをコミットまたはロールバックする必要があるということです。

xa_open 文字列の変更

次の 2 つの新しいパラメータが追加されました。

- Loose_Coupling

  このパラメータはブール値を取ります。Oracle7 サーバーへ接続しているときは、このパラメータを false に設定します。true に設定すると、グローバル・トランザクション・ブランチが疎結合になります。つまり、ロックはブランチ間で共有されません。

- SesWt

  このパラメータの値は、別セッションによって使用中のトランザクション・ブランチを待機するタイムアウト限度を示します。SesWt 秒内に Oracle がそのトランザクション・ブランチに切り替えることができない場合は、XA_RETRY が戻されます。

次の 2 つのパラメータが廃止されました。この 2 つは、Oracle7 リリース 7.3 への接続時にのみ使用します。

- GPWD

  Oracle8 以上では、グループ・パスワードは使用されません。トランザクション・ブランチを作成したセッションと同じユーザー名でログインしているセッションは、そのトランザクション・ブランチに切り替えられます。

- SesCacheSz

  Oracle8 以上ではセッション・キャッシュがなくなったため、このパラメータは使用されません。

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

ふ

へ

ほ

み

ろ