

Oracle8*i*

アプリケーション開発者ガイド 基礎編

リリース 8.1

2000 年 2 月

部品番号 : J00941-01

ORACLE®

Oracle8i アプリケーション開発者ガイド 基礎編, リリース 8.1

部品番号 : J00941-01

原本名 : Application Developer's Guide - Fundamentals, Release 2 (8.1.6)

原本部品番号 : A76939-01

原本著者 : John Russell

原本協力者 : M. Bauer, M. Cyran, J. Gibb, G. Gonzalez, V. Krishnamurthy, M. Krishnaprasad, J. Melnick, R. Moran, D. Raphaely, R. Smith, R. UrbanoContributors: D. Alpern, A. Amor, G. Arora, V. Arora, J. Basu, R. Baylis, E. Beldin, S. Chandrasekar, T. Chang, A. Chaudry, W. Creekbaum, D. Das, M. Davidson, G. Doherty, J. Draaijer, B. Goyal, M. Hartstein, J. Haydu, K. Jacobs, M. Jaganath, N. Jain, H. Jakobsson, A. Jasuja, R. Jenkins Jr., R. Kasamsetty, J. Klein, R. Kooi, S. Krishnamurthy, R. Krishnan, S. Krishnaswamy, P. Lane, N. Le, C. Lei, L. Leverenz, J. Loaiza, D. Lorentz, W. Maimone, D. McMahon, A. Mendelsohn, M. Moore, R. Murthy, K. Muthiah, K. Muthukkaruppan, R. Narayanan, T. Nhu Bui, V. Nimani, T. Portfolio, M. Pratt, S. Puranik, T. Pystynen, M. Ramacher, S. Samu, U. Sangam, A. Sethi, P. Shah, N. Shariatpanahy, T. Smith, J. Srinivasan, S. Subramanian, U. Sundaram, D. Surber, S. Suri, N. Tang, J. Tsai, A. Tsukerman, S. Urman, P. Vasterd, G. Viswana, W. Wang, D. Wong, B. Wright, R. Yaseen

グラフィック・デザイナー : V. Moore

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されています。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation（米国オラクル）または日本オラクル株式会社（日本オラクル）を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation（米国オラクル）およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的のみ使用されており、それぞれの所有者の商標または登録商標です。

目次

第 I 部 サーバー操作の概要

1 Oracle プログラム環境の理解

PL/SQL の概要	1-2
PL/SQL の動作	1-2
PL/SQL の利点	1-3
OCI の概要	1-7
OCI の利点	1-7
OCI の構成要素	1-8
プロシージャ型要素および非プロシージャ型要素	1-8
OCI アプリケーションの作成	1-9
Oracle Objects for OLE の概要	1-11
OO4O オートメーション・サーバー	1-12
OO4O オブジェクト・モデル	1-12
Oracle LOB およびオブジェクト・データ型のサポート	1-17
Oracle Data Control	1-19
Oracle Objects for OLE C++ クラス・ライブラリ	1-19
その他の情報源	1-19
Pro*C/C++ の概要	1-20
Pro*C/C++ アプリケーションの実装方法	1-20
Pro*C/C++ 機能の特徴	1-21
Oracle8i の新機能	1-22
Pro*COBOL の概要	1-23
Pro*COBOL アプリケーションの実装方法	1-23
Pro*COBOL 機能の特徴	1-24

Oracle8i の新機能	1-25
Oracle JDBC の概要	1-26
JDBC Thin Driver	1-26
JDBC OCI Driver	1-26
JDBC サーバー・ドライバ	1-27
JDBC の拡張機能	1-27
JDBC Thin Driver のサンプル・プログラム	1-27
RDBMS における Java	1-29
ストアド・プロシージャを使用する理由	1-29
SQLJ アプリケーションにおける JDBC	1-30
Oracle SQLJ の概要	1-30
SQLJ Tool	1-30
SQLJ の設計目標	1-31
Oracle の SQLJ 実装の強み	1-32
SQLJ と JDBC の比較	1-32
オブジェクト型の SQLJ の例	1-33
サーバー内の SQLJ ストアド・プロシージャ	1-36
プログラム環境の選択	1-37
OCI またはプリコンパイラの使用の選択	1-37
組込みパッケージおよびライブラリ	1-38
Java および PL/SQL	1-39

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

表の管理	2-2
表の設計	2-3
表の作成	2-4
表の変更	2-9
表の削除	2-10
一時表の管理	2-11
一時表の作成	2-12
一時表の使用	2-12
例：一時表の使用	2-13
ビューの管理	2-15

ビューの作成	2-15
ビューの置換	2-17
ビューの使用	2-18
ビューの削除	2-20
結合ビューの変更	2-21
キー保存表	2-22
結合ビューの DML 文の規則	2-23
UPDATABLE_COLUMNS ビューの使用	2-26
外部結合	2-26
順序の管理	2-29
順序の作成	2-29
順序の変更	2-30
順序の使用	2-30
順序の削除	2-34
シノニムの管理	2-35
シノニムの作成	2-35
シノニムの使用	2-35
シノニムの削除	2-36
1 回の操作による複数の表およびビューの作成	2-37
スキーマ・オブジェクトのネーミング	2-38
SQL 文における参照オブジェクトの名前変換	2-38
スキーマ・オブジェクトの改名	2-39
スキーマの改名	2-40
スキーマ・オブジェクトに関する情報のリスト	2-41

3 データ型の選択

Oracle 組込みデータ型	3-2
文字データ型の使用	3-5
NUMBER データ型の使用	3-7
DATE データ型の使用	3-8
西暦 2000 年準拠の設定	3-9
LONG データ型の使用	3-18
RAW データ型および LONG RAW データ型の使用	3-20
ROWID および ROWID データ型	3-21
ANSI/ISO データ型、DB2 データ型および SQL/DS データ型	3-24

データ変換	3-25
ルール 1: 割当て	3-25
ルール 2: 式の評価	3-27

4 データ整合性のメンテナンス

整合性制約の使用	4-2
整合性制約でビジネス・ルールを施行する場合	4-2
アプリケーションでビジネス・ルールを施行する場合	4-3
制約で使用する索引の作成	4-3
NOT NULL 整合性制約の使用	4-3
デフォルトの列値の設定	4-4
表の主キーの選択	4-6
一意キー整合性制約の使用	4-7
参照整合性制約の使用	4-8
NULL および外部キー	4-8
親表と子表の関連	4-10
複数の外部キー制約	4-11
制約チェックの遅延	4-11
対応付けられた索引がある制約の管理	4-13
同時実行性制御、索引および外部キー	4-13
分散データベース内の参照整合性	4-14
CHECK 整合性制約の使用	4-15
CHECK 制約の制限	4-15
CHECK 制約の設計	4-16
複数の CHECK 制約	4-16
CHECK および NOT NULL 整合性制約	4-16
整合性制約の定義	4-17
CREATE TABLE コマンド	4-17
ALTER TABLE コマンド	4-18
必要な権限	4-18
整合性制約のネーミング	4-18
整合性制約の使用可能および使用禁止	4-19
制約を使用禁止にする理由	4-19
整合性制約違反	4-20
定義時	4-20

既存の整合性制約の使用可能および使用禁止	4-21
キー整合性制約の使用可能および使用禁止	4-22
例外のレポート	4-22
整合性制約の変更	4-23
MODIFY CONSTRAINT の例	4-23
整合性制約の削除	4-24
外部キー整合性制約の管理	4-25
外部キー整合性制約の定義	4-25
外部キー整合性制約の使用可能	4-27
整合性制約定義のリスト	4-27
例	4-27

5 索引計画の選択

索引の管理	5-1
索引の作成	5-5
索引の削除	5-5
ファンクション索引	5-6
ファンクション索引の使用	5-7
ファンクション索引の例	5-11
ファンクション索引の要件および制限	5-12
クラスタ、クラスタ化表およびクラスタ索引の管理	5-14
クラスタを作成するためのガイドライン	5-14
パフォーマンスに関する考慮点	5-15
クラスタ、クラスタ化表およびクラスタ索引の作成	5-15
クラスタに対する記憶域の手動割当て	5-17
クラスタ、クラスタ化表およびクラスタ索引の削除	5-17
ハッシュ・クラスタおよびクラスタ化表の管理	5-19
ハッシュ・クラスタおよびクラスタ化表の作成	5-19
ハッシュ・クラスタ内のスペース使用の制御	5-20
ハッシュ・クラスタの削除	5-20
ハッシュの使用時期	5-20

6 索引構成表による索引アクセスのスピードアップ

索引構成表の概要	6-2
索引構成表および通常の表	6-2

索引構成表の利点	6-2
索引構成表の機能	6-4
索引構成表の使用時期	6-7
例	6-9

7 SQL 文の処理

SQL 文の実行	7-2
SQL92 に対する拡張の識別 (FIPS フラグ付け)	7-2
トランザクションの制御	7-4
パフォーマンスの改善	7-4
トランザクションのコミット	7-5
トランザクションのロールバック	7-6
トランザクションのセーブポイントの定義	7-6
トランザクションの管理に必要な権限	7-7
読取り専用トランザクションでのリピータブル・リードの保証	7-8
カーソルの使用	7-9
カーソルの宣言およびオープン	7-9
カーソルを使用した文の再実行	7-9
カーソルのクローズ	7-10
カーソルの取消し	7-10
明示的なデータのロック	7-11
ロック方法の選択	7-12
Oracle に対する表ロック制御の許可	7-16
デフォルト以外のロック・オプションの概要	7-17
行ロックの明示的な取得	7-18
ユーザー・ロック	7-20
ユーザー・ロックの作成	7-20
ユーザー・ロックの例	7-20
ロックの表示および監視	7-21
シリアル化可能トランザクションを使用した同時実行性の 制御	7-22
シリアル化可能トランザクションの相互作用	7-25
分離レベルの設定	7-25
参照整合性およびシリアル化可能トランザクション	7-26
コミット読込み分離およびシリアル化可能分離	7-28
アプリケーションのヒント	7-31

自律型トランザクション	7-32
例	7-35
自律型トランザクションの定義	7-40

8 動的 SQL

動的 SQL とは	8-2
動的 SQL の使用時期	8-3
動的 DML 文の実行	8-3
静的 SQL ではサポートされていない文の PL/SQL での実行	8-3
動的問合せの実行	8-4
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	8-5
実行の動的最適化	8-6
動的 PL/SQL ブロックの起動	8-7
実行者権限を使用した動的操作の実行	8-8
システム固有の動的 SQL を使用した使用例	8-9
データ・モデル	8-9
DML 操作の例	8-10
DDL 操作の例	8-10
動的な単一行問合せの例	8-11
動的な複数行問合せの例	8-12
システム固有の動的 SQL と DBMS_SQL パッケージの対比	8-12
システム固有の動的 SQL の利点	8-13
DBMS_SQL パッケージの利点	8-17
DBMS_SQL パッケージ・コードおよびシステム固有の動的 SQL コードの例	8-19
PL/SQL 以外のアプリケーション開発言語	8-24

9 プロシージャおよびパッケージの使用

PL/SQL プログラム・ユニット	9-2
無名ブロック	9-2
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） 9-5	
PL/SQL コードのラッピング	9-28
リモート依存性	9-28
タイムスタンプ	9-28
シグネチャ	9-30

リモート依存性の制御	9-35
カーソル変数	9-38
カーソル変数の宣言およびオープン	9-38
カーソル変数の例	9-38
コンパイル時エラー	9-41
ランタイム・エラー処理	9-43
例外および例外処理ルーチンの宣言	9-44
未処理例外	9-45
分散問合せでのエラー処理	9-46
リモート・プロシージャでのエラー処理	9-46
ストアド・プロシージャのデバッグ	9-47
ストアド・プロシージャのコール	9-49
リモート・プロシージャのコール	9-53
プロシージャおよびパッケージのシノニム	9-56
SQL 式からのストアド・ファンクションのコール	9-56
PL/SQL ファンクションの使用	9-56
構文	9-57
命名規則	9-57
基本的な要件の充足	9-60
副作用の制御	9-61
オーバーロード	9-69
逐次再使用可能 PL/SQL パッケージ	9-70

10 外部ルーチン

複数の言語を扱う必要性	10-1
外部ルーチンとは	10-2
コール仕様	10-3
外部ルーチンのロード	10-4
Java クラス・メソッドのロード	10-4
外部 C ルーチンのロード	10-5
外部ルーチンの発行	10-6
Java クラス・メソッド用の AS LANGUAGE 句	10-8
外部 C ルーチン用の AS LANGUAGE 句	10-8
Java クラス・メソッドの発行	10-9
外部 C ルーチンの発行	10-10
コール仕様の位置	10-10

コール仕様を使用した Java クラス・メソッドへのパラメータ渡し	10-14
コール仕様を使用した外部 C ルーチンへのパラメータ渡し	10-14
データ型の指定	10-16
外部データ型のマッピング	10-18
IN および IN OUT パラメータ・モードの BY VALUE/BY REFERENCE	10-19
PARAMETERS 句	10-20
デフォルトのデータ型マッピングのオーバーライド	10-21
プロパティの指定	10-21
外部ルーチンの実行: CALL 文	10-29
準備	10-30
CALL 文の構文	10-32
Java クラス・メソッドのコール	10-33
外部 C ルーチンのコール	10-33
エラーおよび例外	10-34
一般的なコンパイル時のコール仕様エラー	10-34
Java の例外処理	10-35
C の例外処理	10-35
外部 C ルーチンでのサービス・ルーチンの使用	10-35
外部 C ルーチンを使用したコールバックの実行	10-43
OCI コールバック用のオブジェクト・サポート	10-45
コールバックに関する制限事項	10-46
外部ルーチンのデバッグ	10-47
デモ・プログラム	10-48
外部 C ルーチンのためのガイドライン	10-48
外部 C ルーチンに関する制限事項	10-50

11 セキュリティ・ポリシーの設定

セキュリティ・ポリシーの概要	11-2
セキュリティの侵害および対策	11-2
任意のセキュリティ・ポリシーで扱える項目	11-3
セキュリティ・ポリシーの設定に使用する機能	11-4
アプリケーション・セキュリティ	11-5
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	11-6
アプリケーション管理者の作業	11-8
ロールおよびアプリケーション権限の管理の概要	11-8

ユーザーの現在のアプリケーション・ロールへの権限の対応付け	11-9
ツール・ユーザーからのアプリケーション・ロールの制限	11-12
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	11-19
オブジェクト権限の管理	11-21
ロールの作成およびロール使用の保護	11-23
ロールの使用可能および使用禁止	11-24
システム権限およびロールの付与と取消し	11-28
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	11-30
ユーザー・グループ PUBLIC に対する権限付与および取消し	11-34
ファイングレイイン・アクセス・コントロール	11-35
ファイングレイイン・アクセス・コントロールの機能	11-36
表またはビューへのポリシーの追加方法	11-38
動的に変更される文の例	11-39
アプリケーション・コンテキスト	11-40
アプリケーション・コンテキストの機能	11-41
アプリケーション・コンテキストの機能設計原理	11-45
ファイングレイイン・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法	11-47
アプリケーション・コンテキストの使用法	11-49
例	11-53
中間層を介する認証	11-64
n 層認証の利点	11-64
3 層コンピューティングのセキュリティ問題	11-65
Oracle8i の n 層認証のソリューション	11-69
データの暗号化	11-72
DBMS_OBFUSCATION_TOOLKIT パッケージ	11-72
開発時の考慮点	11-73

第 III 部 アクティブ・データベース

12 トリガーの使用

トリガーの設計	12-2
トリガーの作成	12-3
トリガー作成の前提条件	12-4
トリガーの種類	12-4
トリガーのネーミング	12-5

トリガー文	12-5
BEFORE オプションおよび AFTER オプション	12-7
INSTEAD OF トリガー	12-7
FOR EACH ROW オプション	12-12
WHEN 句	12-13
トリガー本体	12-14
トリガーおよびリモート例外処理	12-18
トリガー作成の制限	12-20
トリガー・ユーザーとは	12-26
権限	12-26
トリガーのコンパイル	12-28
依存性	12-28
トリガーの再コンパイル	12-29
移行の問題	12-29
トリガーの変更	12-30
トリガーのデバッグ	12-30
トリガーの使用可能および使用禁止	12-31
トリガーの使用可能	12-31
トリガーの使用禁止	12-31
トリガーに関する情報のリスト	12-32
トリガー・アプリケーションの例	12-34
イベント発行のトリガー	12-54
発行フレームワーク	12-54

13 システム・イベントの処理

イベント属性関数	13-2
イベントのリスト	13-6
リソース・マネージャ・イベント	13-6
クライアント・イベント	13-7

14 パブリッシュ/サブスクライブの使用

パブリッシュ/サブスクライブの概要	14-2
パブリッシュ/サブスクライブのインフラストラクチャ	14-3
パブリッシュ/サブスクライブの概念	14-4
例	14-6

第 IV 部 特殊アプリケーションの開発

15 PL/SQL を使用した Web アプリケーションの開発

PL/SQL を使用したネットワーク操作の実行	15-1
メールの送信	15-1
ホスト名またはアドレスの取得	15-1
TCP/IP 接続を使用した作業	15-1
HTTP URL の内容の取出し	15-2
表、イメージ・マップ、Cookie、CGI などを使用した処理	15-2
Web ページ (PL/SQL サーバー・ページ) への PL/SQL コードの埋込み	15-2
ソフトウェア構成を選択する	15-3
PL/SQL サーバー・ページにコードおよびコンテンツを書き込む	15-4
PL/SQL サーバー・ページ要素の構文	15-9
PL/SQL サーバー・ページをストアド・プロシージャとしてデータベースにロードする	15-11
URL を介した PL/SQL サーバー・ページの実行	15-12
PL/SQL サーバー・ページの例	15-12
PL/SQL サーバー・ページのデバックに関する問題	15-18
PL/SQL サーバー・ページを使用したアプリケーションの商品化	15-19

16 Oracle XA でのトランザクション・モニターの操作

X/Open Distributed Transaction Processing (DTP)	16-2
必須のパブリック情報	16-5
XA および 2 フェーズ・コミット・プロトコル	16-5
トランザクション処理モニター (TPM)	16-6
動的登録および静的登録のサポート	16-6
Oracle XA ライブラリ・インタフェース・サブルーチン	16-7
XA ライブラリ・サブルーチン	16-7
XA インタフェースの拡張	16-8
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	16-9
DBA またはシステム管理者の責任	16-9
アプリケーション開発者の責任	16-10
xa_open 文字列の定義	16-10
プリコンパイラと OCI のインタフェース	16-17
XA を使用したトランザクション制御	16-20
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	16-23

XA ライブラリ・スレッド・セーフティ	16-24
XA アプリケーションのトラブルシューティング	16-26
XA トレース・ファイル	16-26
トレース・ファイルの例	16-27
インダウト・トランザクションまたは保留中のトランザクション	16-27
Oracle Server の SYS アカウント表	16-28
XA の一般的な問題および制限事項	16-29
データベース・リンク	16-29
Oracle Parallel Server オプション	16-30
SQL に基づく制限事項	16-30
XA に関するその他の問題	16-31
Oracle XA サポートの変更	16-32
リリース 8.0 からリリース 8.1 への XA の変更点	16-32
リリース 7.3 からリリース 8.0 への XA の変更点	16-33

索引

はじめに

『Oracle8i アプリケーション開発者ガイド 基礎編』では、Oracle Server リリース 8.1 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で動作する Oracle Server のバージョンに適用されますが、システム固有の情報は含みません。

この章の内容は次のとおりです。

- [このマニュアルについて](#)
- [対象読者](#)
- [機能範囲および可用性](#)
- [関連マニュアル](#)
- [このマニュアルの構成](#)
- [このマニュアルの表記規則](#)

このマニュアルについて

アプリケーション開発者は、アプリケーション開発を容易にし、パフォーマンスを改善できる Oracle Server の数多くの機能について理解する必要があります。このマニュアルは、アプリケーション開発に関する Oracle Server の機能について説明しています。このマニュアルでは、PL/SQL 言語については説明していません。また、クライアント側でのアプリケーション開発についても説明していません。記載内容の詳細は、目次および「[このマニュアルの構成](#)」を参照してください。「[関連マニュアル](#)」では、関連情報を含むその他の Oracle マニュアルを示しています。

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、構造化問合せ言語 (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。

機能範囲および可用性

このマニュアルには、Oracle8i および Oracle8i Enterprise Edition 製品の特徴および機能を説明した情報が含まれています。Oracle8i と Oracle8i Enterprise Edition の基本機能は同じです。ただし、最新機能の中には、Enterprise Edition のみで使用可能であり、オプションのものもあります。たとえば、オブジェクト機能を使用するには、Enterprise Edition およびオブジェクト・オプションが必要です。

関連マニュアル

PL/SQL について学習し、オラクル社が提供する SQL（構造化問合せ言語）のプロシージャ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、『Oracle8i 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 コール・インタフェース（OCI）については、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

OCI を使用して、Oracle Server にアクセスする第 3 世代言語（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、Ada、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・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、『Oracle8i Pro*C/C++ プリコンパイラ・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 Tools のドキュメントを参照してください。

SQL の詳細は、『Oracle8i SQL リファレンス』および『Oracle8i 管理者ガイド』を参照してください。Oracle の基本概念は、『Oracle8i 概要』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、次のように構成されています。

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。

「[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論理を削減できます。

「[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。

「[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。

「[データ整合性のメンテナンス](#)」では、エラー・チェック論理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。

「[索引計画の選択](#)」および「[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。

「[SQL 文の処理](#)」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 トピックについて説明します。

「[動的 SQL](#)」では、動的 SQL、システム固有の動的 SQL と DBMS_SQL パッケージ、および動的 SQL を使用するタイミングについて説明します。

「[プロシージャおよびパッケージの使用](#)」では、再使用可能なプロシージャをデータベースに格納する方法を説明します。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。

「[外部ルーチン](#)」では、計算集中型プロシージャの本体を、PL/SQL 以外の言語でコーディングする方法について説明します。

「[セキュリティ・ポリシーの設定](#)」では、認証論理をアプリケーションからデータベースへ移動する方法を説明します。

第Ⅲ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利点を利用することができ、コーディング動作を繰り返す必要が少なくなります。

「[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・アクセスのログなどにトリガーを使用できます。

「[システム・イベントの処理](#)」では、ユーザー ID および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り出す方法を説明します。

「[パブリッシュ / サブスクライブの使用](#)」では、メッセージ機能またはキュー機能とも呼ばれる非同期通信用の Oracle モデルを紹介します。

第Ⅳ部：専用アプリケーションの開発

「[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。

「[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。

このマニュアルの表記規則

このマニュアルで使用する表記規則および本文の形式は、次のとおりです。

[]

大カッコは、任意に選択する項目を示します。カッコは入力しないでください。

{ }

中カッコは、複数の項目を囲み、その中の 1 つのみが必須であることを示します。

|

縦棒は、中カッコ内の項目を分割します。また、複数の値を関数のパラメータに渡すことを示す場合にも使用します。

...

コード例の省略記号は、その説明内容に関係のないコードを省略していることを示します。

固定幅フォント

固定幅フォントは、SQL または C のコード例を示します。

イタリック

イタリックは、OCI パラメータ、OCI ルーチン名、ファイル名およびデータ・フィールドを示します。

大文字

大文字は、SELECT または UPDATE などの SQL キーワードを示します。

このマニュアルでは、ある情報に対して読者の注意を促すために、特別な形式を使用しています。太字テキストの見出しで始まる段落は、特別な意味を持つ場合があります。次の見出しで始まる各段落では、それぞれ特別な情報を示します。

注意：共通の問題を避けたり、概念の理解を深めるために、情報に特別な注意を払う必要があることを示します。

警告：OCI アプリケーションが正しく動作するために注意して行う必要があること、および行ってはいけないことを OCI プログラマに示します。

参照：説明する項目についての追加情報が記載されているこのマニュアルの他の項、または他のマニュアルを示します。

第I部

サーバー操作の概要

第I部に含まれる章は、次のとおりです。

- 第1章「[Oracle プログラム環境の理解](#)」

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](#)
- [OCI の概要](#)
- [Oracle Objects for OLE の概要](#)
- [Pro*C/C++ の概要](#)
- [Pro*COBOL の概要](#)
- [Oracle JDBC の概要](#)
- [Oracle SQLJ の概要](#)
- [プログラム環境の選択](#)

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（第 4 世代プログラミング言語）として、シームレスな SQL アクセス、Oracle Server および Tools 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、情報の非表示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 では、SQL 文を使用した Oracle データの操作や、制御フロー文を使用したデータ処理を行うことができます。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。このように、PL/SQL は SQL の強力なデータ操作機能およびプロシージャ型言語の強力なデータ処理機能を結合したものです。

Oracle プログラム・インタフェース（Oracle コール・インタフェース、Java、Pro*C/C++ または Pro*COBOL）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 スタッド・プロシージャをコールし、PL/SQL コードの無名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・データ型に完全にアクセスできます。

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 1 つの言語で開発し、アプリケーション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、『Oracle8i 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。

PL/SQL の動作

PL/SQL を理解するには、次に示すプロシージャのサンプル・プログラムを参照してください。このプロシージャは、銀行勘定の借方記入を行うものです。このプロシージャ `debit_account` は、コール時に口座番号および借方金額を受け入れます。口座番号を使用して、データベース表から口座残高が選択されます。次に、借方金額を使用して新規残高が計算されます。新規残高が 0（ゼロ）より小さい場合は例外処理が発生し、それ以外の場合は勘定が更新されます。

```
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 の利点

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、次のような利点があります。

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。さらに、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。

PL/SQL は動的 SQL もサポートします。動的 SQL とは、アプリケーションの柔軟性および適応性を高めるための高度なプログラミング手法です。データ定義、データ制御およびセッション制御用の SQL 文を、プログラムの実行時に、その場で作成して処理することができます。

Oracle との緊密な統合

PL/SQL および Oracle は SQL ベースです。また、PL/SQL はすべての SQL 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わせることで、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 属性および %ROWTYPE 属性によって、PL/SQL とデータ・ディクショナリの統合がさらに緊密になります。たとえば、%TYPE 属性を使用すると、データベースの列定義を基に変数を宣言できます。列定義が変更された場合、変数宣言も実行時に変更されます。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、新しいビジネス・ニーズに伴うデータベースの変更に、プログラムが対応できるようになります。

より高いパフォーマンス

PL/SQL を使用しない場合、Oracle では、SQL 文を 1 つずつ処理する必要があります。各 SQL 文がそれぞれ Oracle をコールすることになり、パフォーマンス上のオーバーヘッドが増加します。ネットワーク環境では、このオーバーヘッドが重要な意味を持ちます。SQL 文は、発行ごとにネットワークを介して送信されるため、通信量が増大します。

しかし、PL/SQL を使用することによって、文のブロック全体を一度に Oracle に送信することができます。これによって、アプリケーションと Oracle 間の通信量が大幅に軽減されます。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。

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。また、サーバーで実行されるストアド・プロシージャは、低速ネットワーク接続の場合でも 1 回のコールで起動できます。これによって、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、必要なメモリ量および起動オーバーヘッドが減少します。

より高い生産性

PL/SQL によって、Oracle Forms および Oracle Reports などの非プロシージャ型 Tools に対して機能を追加できます。このような Tools で PL/SQL を使用すると、使い慣れたプロシージャ型の構造体を使用してアプリケーションを作成できるようになります。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、マクロおよびユーザー・イグジットを使用する必要はありません。このように、PL/SQL によって使用しやすく効率のよい手段が提供されるため、生産性が向上します。

さらに、PL/SQL はどの環境でもまったく同等に使用できます。Oracle Tool のいずれかで PL/SQL が使用できるようになれば、この知識を他の Tools にも利用できるため、生産性が向上します。たとえば、ある Tool で作成したスクリプトは、他の Tools でも使用できます。

スケーラビリティ

PL/SQL のストアド・プロシージャは、アプリケーション処理がサーバー上で独立しているため、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。

マルチスレッド・サーバー (MTS) の共有メモリ機能によって、Oracle は、単一ノード上で 10,000 以上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、Net8 Connection Manager を使用して Net8 接続を多重化できます。

より高いメンテナンス性

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まったく影響しません。このため、メンテナンスおよび拡張が簡素化されます。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の方が簡単です。

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した場合、モジュール方式でメンテナンス性が高く、再利用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・チームがソフトウェア・コンポーネントを同時に開発できます。

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、NESTED TABLE 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、ほとんどの第 3 世代プログラミング言語の配列と同じように機能し、オブジェクト型のインスタンスを格納でき、また、逆にオブジェクト型の属性にもなります。さらに、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。さらに、PL/SQL パッケージ内にコレクション型を定義することで、アプリケーション・プログラムで使用できます。

完全な移植性

PL/SQL で作成したアプリケーションは、Oracle が実行されていれば、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使用できます。

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論理をパーティション化できます。このようにすることで、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更新を選択的に禁止することができ、ユーザーからの問合せをその内容に基づいて監査することができます。

さらに、ユーザーによる Oracle データの操作を、定義者権限で実行されるストアド・プロシージャを介してのみで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。

OCI の概要

Oracle コール・インタフェース（OCI）は、アプリケーション・プログラム・インタフェース（API）です。これによって、第3世代言語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したりするアプリケーションを作成できます。OCI 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。OCI 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できる動的ランタイム・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 コールの詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』、『Oracle8i アプリケーション開発者ガイド アドバンスド・キューイング』、『Oracle8i NLS ガイド』および『Oracle8i データ・カートリッジ開発者ガイド』を参照してください。

OCI の利点

OCI は、Oracle データベースに対する他のアクセス方法と比較して、次のような優れた利点があります。

- 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・サーバーから取り出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 C コールバックを作成します。

プロシージャ型要素および非プロシージャ型要素

Oracle コール・インタフェース（OCI）を使用すると、SQL が提供する強力な非プロシージャ型データ・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プロシージャ型機能を組み合わせたアプリケーションを開発できます。

- 非プロシージャ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 は非プロシージャ型であるため、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。

- プロシージャ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、プロシージャ型であるために SQL よりも複雑になりますが、非常に柔軟で強力でもあります。

OCI プログラムで非プロシージャ型言語の要素とプロシージャ型言語の要素を組み合わせることができるため、構造化プログラム環境の中で 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・サーバー経由で使用可能な 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
```

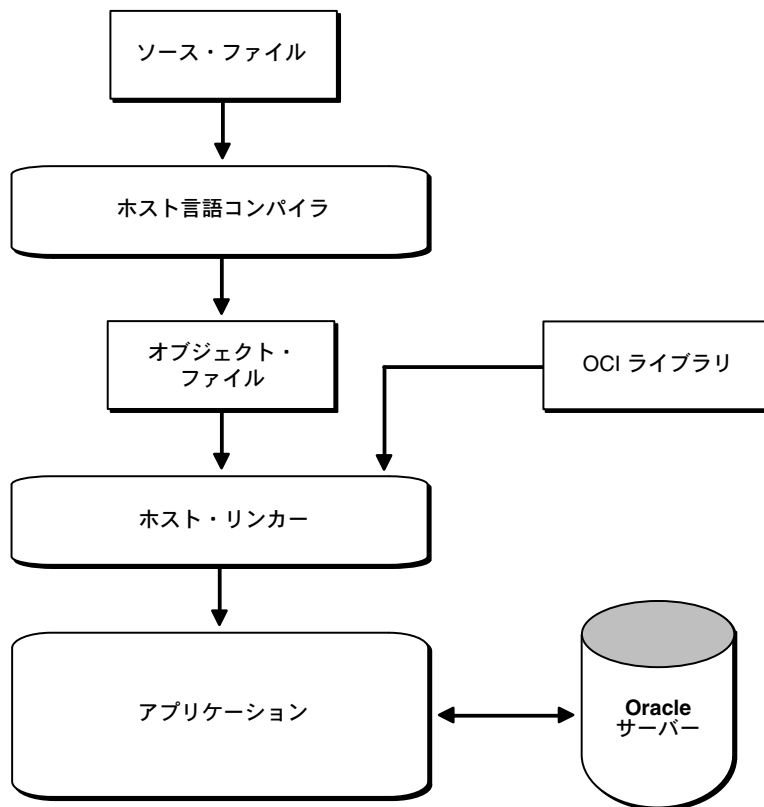
この SQL 文では、:empnumber がアプリケーションによって提供される値のプレース・ホルダです。

また、SQL に対する Oracle のプロシージャ型拡張言語である 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 に示すように、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 の開発プロセス



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 の概要

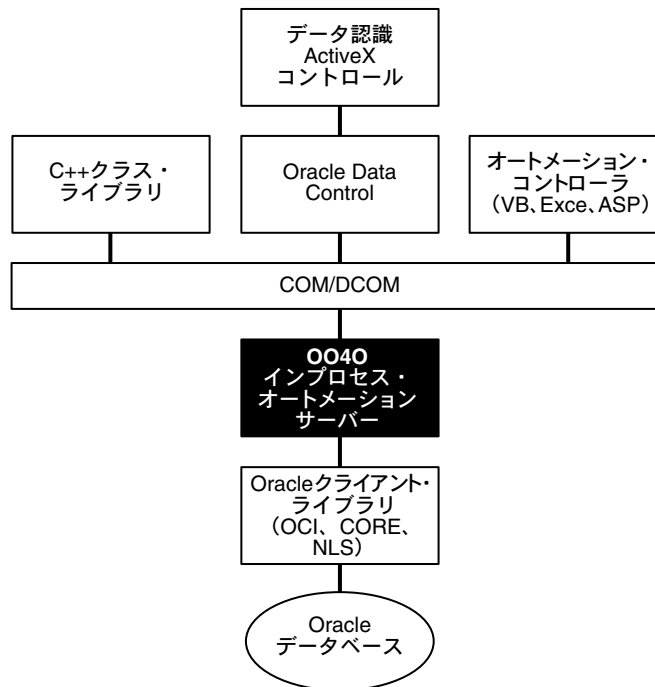
Oracle Objects for OLE (OO4O) は、Microsoft COM Automation および ActiveX テクノロジをサポートするプログラミング言語またはスクリプト言語であれば、どの言語からでも Oracle データベース内に格納されているデータに簡単にアクセスできるように設計された製品です。このような言語には、Visual Basic、Visual C++、Visual Basic For Applications (VBA)、IIS Active Server Pages (VBScript および JavaScript) などがあります。

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
- Oracle Objects for OLE 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。

0040 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、広範囲な環境に渡り、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。

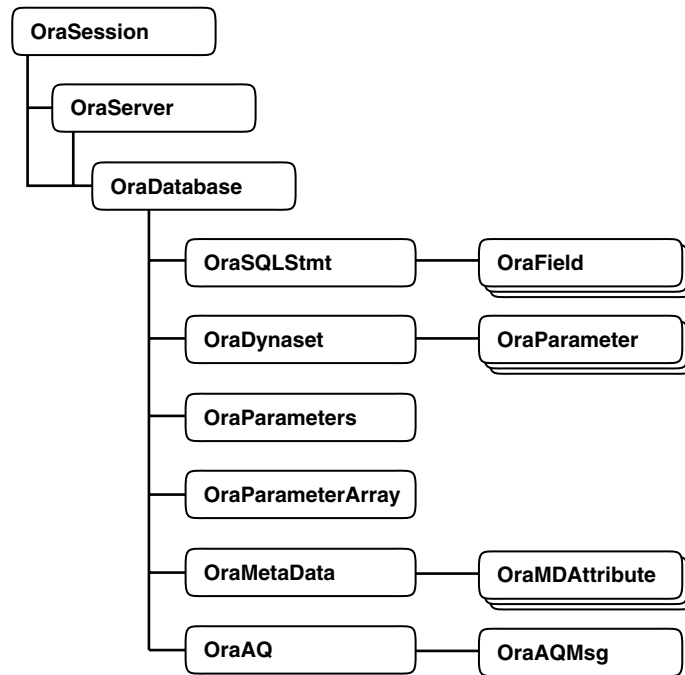
次のような機能があります。

- PL/SQL の無名ブロックおよびストアド・プロシージャの実行サポート
これには、PL/SQL カーソルなど、PL/SQL ストアド・プロシージャの入出力パラメータとして使用可能な Oracle データ型のサポートが含まれます。詳細は、1-17 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
- Oracle8i 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
- Oracle8i でのアドバンスド・キューイングの完全サポート
- 配列の挿入および更新のサポート
- Microsoft Transaction Server (MTS) のサポート

0040 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、Oracle Objects for OLE 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。Oracle Objects for OLE 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。次のコードは、OraSession オブジェクトの作成方法の一部を示したものです。

```
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、Oracle コール・インタフェース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、Internet Information Server (IIS) などのように、N 層分散環境で Oracle Objects for OLE 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 Server 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、サーバーのスケラビリティも向上します。

OraDatabase

Oracle8i リリースの OraDatabase インタフェースには、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が新しく追加されています。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り出すことができます。

以前のリリースでは、OraDatabase オブジェクトは OraSession インタフェースの OpenDatabase メソッドを起動して作成されました。Net8 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。Oracle8i リリースでは、このメソッドの起動で 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ように、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することができます。

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、読み込み / 書き込み (デフォルト)、シリアル化可能、または読み取り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

使用例を次に示します。

```
UserSession.BeginTrans (OO4O_TXN_READ_WRITE)
UserSession.ExecutesSQL("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勧めします。

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、序数を使用したサブスクリプト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 (0 からカウント -1 まで) にないサブスクリプトを参照すると、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ように) 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、動的な問合せを簡素化し、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。

Oracle Objects for OLE は、Oracle のアドバンスト・キューイング (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、『Oracle8i 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、『Oracle8i 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、複数利用者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。使用例を次に示します。

```
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。使用例を次に示します。

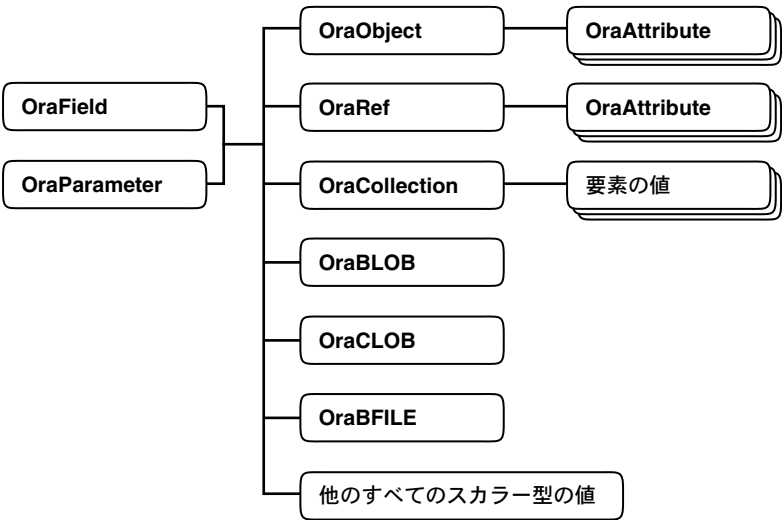
```
Set agent = qMsg.AddRecipient (name, address, protocol).
```

Oracle LOB およびオブジェクト・データ型のサポート

Oracle Objects for OLE 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したり、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 ブロックへの入力変数または出力変数として渡したりできます。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 の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。

Oracle Data Control 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。Visual Basic のデータ・コントロールを使い慣れている場合は、Oracle Data Control 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 Oracle Data Control 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。

Oracle Objects for OLE C++ クラス・ライブラリ

Oracle Objects for OLE 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

Oracle Objects for OLE 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。

- Oracle Objects for OLE のヘルプ
- Oracle Objects for OLE C++ クラス・ライブラリのヘルプ

Oracle Object for OLE 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。

- 『Oracle8i アプリケーション開発者ガイド ラージ・オブジェクト』
- 『Oracle8i 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』
- 『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

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 文を埋め込め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 つのステップを再実行します。

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 表を問い合わせる C ソース・ファイル内の簡単なコードの一部を示します。

```
...
#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 文と対話形式 (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。コロンの (:) がすべてのホスト (C) 変数の前に付きます。データまたは (データ値が NULL の場合またはキャラクタ列が切り捨てられたときに設定される) 標識の戻り値は、(前述のコード例にあるような) 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、社員番号が一意であるため、結果が1つのみのこの例とよく似た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、エラーのレポート方法、出力形式、(特定の接続や SQL 文などに対応する) カーソルの管理方法など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 で対話形式の 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++ 機能の特徴

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、『Oracle8i Pro*C/C++ プリコンパイラ・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、単一スレッド・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で、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チェックできます。

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標識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 インタフェースである Oracle コール・インタフェース (OCI) 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。

Oracle8i の新機能

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) が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構造体およびヘッダーにマップし、この構造体およびヘッダーをソースに含めます。

NESTED TABLE および VARRAY という 2 種類のコレクション型が、データを高度に制御するための一連の SQL 文とともにサポートされています。

ラージ・オブジェクト (LOB、CLOB、NCLOB および BFILE といわれる外部ファイル) 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。

マルチバイト・キャラクタ用の各国語サポートおよび UCS2 Unicode サポートが提供されています。

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 文を埋め込むようにするためのソフトウェア Tool 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 つのステップを再実行します。

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 表を問い合わせるソース・ファイル内の簡単なコードの一部を示します。

```
...
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 文と対話形式 (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。コロンの (:) 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 の場合または文字の列が切り捨てられたときに設定される) 標識の戻り値は、(前述のコード例にあるような) 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、社員番号が一意であるため、結果が1つのみのこの例とよく似た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、エラーのレポート方法、出力形式、(特定の接続や SQL 文などに対応する) カーソルの管理方法など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、Net8 経由でネットワーク化されているリモート・サーバーをはじめとして、多数のデータベースのデータにプログラムでアクセスできます。

埋込み SQL 文を作成する前に、SQL*Plus で対話形式の 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 機能の特徴

Pro*COBOL 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、『Oracle8i Pro*COBOL プリコンパイラ・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、構文チェック、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チェックできます。

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標識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。

Oracle8i の新機能

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。

Pro*COBOL には、移行しやすいように DB2 互換機能が多数含まれています。

Oracle JDBC の概要

JDBC (Java Database Connectivity) は、Oracle8i 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 API (アプリケーション・プログラミング・インタフェース) 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 – 固有のメソッド (C または C++ をコール) および Java のメソッドがあります。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 文の作成には入力が必要です。

Oracle によって実装されたドライバには、サン・マイクロシステムズ社が定義した 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 Thin Driver

JDBC Thin Driver 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 TTC 実装 (Oracle の Net8 の TCP/IP バージョン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、プラットフォームから独立しています。

Thin Driver 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 ブラウザにダウンロードされる Java アプレット内で使用します。Thin Driver 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しか実行できません。

JDBC OCI Driver

OCI Driver はタイプ 2 の JDBC ドライバです。C で作成されている OCI (Oracle コール・インタフェース) をコールして、Oracle データベース・サーバーと対話します。つまり、固有のメソッドおよび Java メソッドの両方を使用します。

OCI Driver は固有のメソッド（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。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、Oracle8i をクライアントにインストールする必要があります。OCI Driver は、通常、Thin Driver より実行速度が速くなります。

OCI Driver 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。このドライバは、ブラウザおよび CORBA（共通オブジェクト・リクエスト・ブローカ・アーキテクチャ）クライアントからアプリケーションへ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 Tools のコレクションである Oracle Web Application Server では使用できます。

JDBC サーバー・ドライバ

JDBC サーバー・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、大量データのアクセスに必要なラウンドトリップが削減されます。このドライバ、Java サーバー VM、データベース、実行速度を 10 倍も上げる NCOMP ネイティブ・コンパイラ、および SQL エンジン、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、関数、トリガー、Java ストアド・プロシージャ、CORBA オブジェクトおよび EJB（Enterprise Java Beans））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まったく同じ機能および拡張機能を完全にサポートします。

JDBC の拡張機能

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の向上
- バッチ処理の実行によるパフォーマンスの向上
- ラウンドトリップを削減する問合せ列タイプの指定
- DatabaseMetaData コールの制御

JDBC Thin Driver のサンプル・プログラム

次のソース・コードは、Oracle JDBC Thin Driver の登録、データベースの接続、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 列を取り出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.driver.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()` メソッドを実行することによって、JDBC ドライバに対して Oracle を拡張できます。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、行フェッチおよび実行のバッチ処理を指定できます。

このコード内で OCI Driver を使用する場合は、Connection 文を次のコードで置き換えます。

```
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。

RDBMS における Java

Oracle データベース・サーバーは、PL/SQL サブプログラムの他に Java クラスも格納します。GUI 関連のメソッドを除き、Java メソッドは RDBMS ではストアド・プロシージャとして実行できます。次に示すデータベース構造体がサポートされています。

ファンクションおよびプロシージャ

これらの名前付きブロックを作成し、次に、loadjava、SQL CREATE FUNCTION、CREATE PROCEDURE または CREATE PACKAGE 文（あるいはそのすべて）を使用してこれらの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 CALL 文
- 埋込み SQL の CALL 文
- PL/SQL のブロック、サブプログラム、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 Tools
- JDBC、SQLJ 文、CORBA、Enterprise Java Beans などの Oracle Java インタフェース

データベース・トリガー

データベース・トリガーとは、指定された DML 操作でトリガーの表またはスキーマが変更されるたびに 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ぎ、また、明示的にコールしなくてもその他多数のアクションを実行できます。

ストアド・プロシージャを使用する理由

- ストアド・プロシージャは 1 回コンパイルするのみで、使用しやすく、メンテナンス性が高く、必要なメモリーおよび演算オーバーヘッドが少なく済みます。
- ネットワークのボトルネックが回避され、応答時間が改善されます。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。
- 演算集中型のプロシージャは、サーバーで実行する方がより速く実行されます。
- 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・プロシージャのみをユーザーに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・プロシージャは、様々なアプリケーション内で再使用できる他に、地理的に異なるサイトでも再使用できます。

SQLJ アプリケーションにおける JDBC

JDBC コードおよび SQLJ コード（1-30 ページの「[Oracle SQLJ の概要](#)」を参照）は相互運用できるため、JDBC の動的 SQL 文と SQLJ の静的 SQL 文と一緒に使用することができます。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、『Oracle8i 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 の特徴：

- 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 の作成者であるサン・マイクロシステムズ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 Tool で、Oracle8i 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 Tool の概要を次に説明します。

SQLJ Tool

Oracle ソフトウェア Tool 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 Java VM 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・ファイルは、静的 SQL 文が埋め込まれた Java ソース・ファイルです。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。

Oracle8i の SQLJ は、次の 3 つのステップで実行されます。

- SQLJ ソースを、SQL 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チェックし、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 は、Oracle 固有にカスタマイズされたプロファイル・ファイルを生成します。

Oracle8i は、データ・サーバーと統合された Java VM 内で実行される SQLJ のストアード・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、最も簡単で実行可能な 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、修飾名が複雑な Java ホスト式）の前にはコロン (:) が付きます。その他の SQLJ 文は宣言であり（Java の型を宣言し）、これを使用して多数の値を取り出す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 の設計目標

最大の目標は、Java に対して単純な拡張を提供することによって、データベースとの通信に埋込み SQL を使用する Java 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することです。

この目標を達成するための具体的な目標は次のとおりです。

- 静的 SQL 経由で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 構造体を提供します。
- 変換時に静的 SQL をチェックします。
- 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。

Oracle の SQLJ 実装の強み

- 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
- 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 との統合
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 ステップで提供されます。
- 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
- Oracle データ型の拡張
サポートされているデータ型は、LOB、ROWID、REF CURSOR、VARRAY、NESTED TABLE、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 からデータベースに対する完全な動的 SQL インタフェースを提供します。SQLJ は、補足的な役割を果たします。

JDBC は、Java からの動的 SQL 実行を細かく制御します。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ドを提供します。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（あるいはその両方）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型を設定できる強力な機能が提供され、コールに対する型チェックが実行できます。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）に対する汎用コール（fun）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簡単な例を次に示します。詳細は、『Oracle8i 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。

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 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・サンプルでは、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
            };
        } catch (SQLException e) {
            // Handle exception
        }
    } catch (SQLException e) {
        // Handle exception
    }
}

```

```
        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ス・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ス・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。

プログラム環境の選択

新しい開発プロジェクト用のプログラム環境を選択します。

- 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。
- PL/SQL ストアド・プロシージャは、この章で記述したどの言語で作成されたコードでもコールできるということを認識しておいてください。また、Java ストアド・プロシージャは、Pro*C/C++、OCI および Pro*COBOL プログラムからも使用できます。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も、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

プリコンパイラ文のみでは実行できないタスクがあるため、技術的な要件から、プリコンパイラではなく OCI の使用が指示される場合があります。

- 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を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。
- 特殊な目的に対する OCI コールには、アドバンスド・キューイング、各国語サポート、データ・カートリッジ、DATETIME 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

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 には次のライブラリがあります。

アプリケーション開発パッケージ

- DBMS_PIPE は、セッション間の通信に使用します。
- DBMS_PIPE は、ユーザーへの警告の伝達に使用します。
- 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、空白情報を取得し、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

スナップショット、アドバンスト・レプリケーション、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。

表 9-2「[Oracle パッケージ・プロシージャ一覧](#)」を参照してください。

Java ライブラリには次のものがあります。

コア JDK ライブラリ

java.lang、java.io などのライブラリです。

その他のライブラリ

Java ベースの CORBA ORB、EJB（Enterprise Java Beans）、Java ベースの XML Parser、Java Web Server などのライブラリです。

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。
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 IO 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、ほとんどのデータベース・アクセスが実行された場合、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、情報の非表示、オーバーロードおよび例外処理の機能があります。

PL/SQL がこれまでに備えてきた最適化機能の多くは、まだ Oracle8i の Java に実装されていません。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。コード開発は、通常、Java を使用した場合より速く行えます。

計算用に最適化された Java

Java は、データベース・アクセスを実行していない場合は PL/SQL よりかなり高速に実行します。また、複雑なオブジェクト指向アプリケーションおよび CPU 集中アプリケーションでは、PL/SQL より適しています。

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 にはありません。

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。

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。

- 第2章「スキーマ・オブジェクトの管理」
- 第3章「データ型の選択」
- 第4章「データ整合性のメンテナンス」
- 第5章「索引計画の選択」
- 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」
- 第7章「SQL文の処理」
- 第8章「動的SQL」
- 第9章「プロシージャおよびパッケージの使用」
- 第10章「外部ルーチン」
- 第11章「セキュリティ・ポリシーの設定」

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。内容は次のとおりです。

- 表の管理
- 一時表の管理
- ビューの管理
- 結合ビューの変更
- 順序の管理
- シノニムの管理
- 1回の操作による複数の表およびビューの作成
- スキーマ・オブジェクトのネーミング
- スキーマ・オブジェクトの改名
- スキーマ・オブジェクトに関する情報のリスト

参照：

- 索引およびクラスター – 第5章
- プロシージャ、ファンクション、パッケージ – 第9章
- オブジェクト型 – 『Oracle8i 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』
- 依存性情報 – 第9章
- 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、『Oracle8i レプリケーション・ガイド』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： 詳細な提案については、『Oracle8i 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ドキュメント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、クラスタに対して内容を連想しやすい名前を使用します。
- 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。
- NULL を許可する列を最後に定義して、記憶領域を節約します。
- 適切な時にはいつでも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： ガイドラインについては、[第4章「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、ユーザー SCOTT が次の文を発行すると、クラスタ化されていない Emp_tab 表が SCOTT のスキーマに作成されます。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は整合性制約が定義されていることに注目してください。

```
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 );
```

データ・ブロックの領域使用管理

次の項では、PCTFREE パラメータおよび PCTUSED パラメータを次のような目的で使用方法について説明します。

- データ・セグメントまたは索引セグメントの書込みおよび検索のパフォーマンスの向上
- データ・ブロック内の未使用領域の削減
- データ・ブロック間の行連鎖の削減

PCTFREE の指定

PCTFREE のデフォルト値は 10 パーセントです。PCTFREE と PCTUSED の合計が 100 を超えない限り、0 ～ 99 の任意の整数を指定できます。(PCTFREE が 99 に設定されると、行のサイズにかかわらず、各ブロック内に少なくとも 1 行が格納されます。行が非常に小さく、ブロックが非常に大きい場合、1 行よりも多くの行が格納されることもあります。)

PCTFREE を小さくすると、次のようになります。

- 既存の表の行を更新するために確保される領域が小さくなります。
- 挿入によるブロック充てん密度が高くなります。

- 表または索引のデータ全体を格納するブロック数が少なくなる（ブロックあたりの行またはエントリは多くなる）ため、領域が節約される場合があります。
- 空き領域が新しいデータまたは更新データで一杯になるので、頻繁にブロックを再編成する必要があるため、処理コストが増加します。
- 行または索引エントリの更新で行が大きくなり複数のブロックにまたがる場合、処理コストおよび必要領域が増加する可能性があります（与えられた行に対して UPDATE 文、DELETE 文および SELECT 文がより多くのブロックを読み込む必要があり、連鎖行の断片は他の断片への参照を含むため）。

PCTFREE を大きくすると、次のようになります。

- 既存の表の行を更新するために確保される領域が大きくなります。
- 同じ量の挿入データであっても、必要なブロックが増加する可能性があります（ブロックあたりの挿入行が少なくなります）。
- ブロックにおける空き領域の再編成がほとんど必要ないため、処理コストは小さくなります。
- Oracle が行の断片を連鎖する頻度が減るため、更新パフォーマンスが改善される場合があります。

PCTFREE の設定では、表データまたは索引データの特性を理解しておく必要があります。更新によって行が長くなることがあります。NUMBER、VARCHAR2、LONG または LONG RAW を使用している場合、更新後の値は、元の値と同じサイズにならない場合があります。データ値が長くなるような更新が多い場合、PCTFREE を大きくしてください。行を更新しても全体の行の幅に影響しない場合には、PCTFREE を小さくできます。

目標は、高密度で詰め込まれたデータ（低い PCTFREE、一杯のブロック）と高い更新パフォーマンス（高い PCTFREE、余裕のあるブロック）の間で許容できる妥協点を見つけることです。

PCTFREE は、他のユーザーに属していてコミットされていないトランザクションを持つ表に対する、特定のユーザーの問合せのパフォーマンスにも影響します。読取り一貫性の保証によって、空き領域がほとんどないブロックでデータの再編成が頻繁に発生する場合もあります。

クラスタ化されていない表の PCTFREE クラスタ化されていない表の行のデータ・サイズが増加する可能性がある場合、これらの更新に備えてある程度領域を確保します。更新に備えて領域を確保しておかないと、更新行はブロック間で連鎖され、これらの行にかかわる I/O パフォーマンスが低下することになります。

クラスタ化された表の PCTFREE クラスタ化されていない表の説明は、クラスタ化された表にも適用されます。ただし、PCTFREE に達すると、同じクラスタ・キーに含まれる任意の表の新しい行が、既存のクラスタ・キーに連鎖された新しいデータ・ブロックに入ります。

索引の PCTFREE 索引データの更新に、索引が空き領域を使用する可能性はあまりありません。したがって、索引セグメント・データ・ブロックに対する PCTFREE 値は、通常、非常に小さい値（5 以下など）になります。

PCTUSED の指定

データ・ブロック内の空き領域の割合が PCTFREE に達すると、使用されている領域の割合が PCTUSED より低くなるまで、新しい行はそのブロックには挿入されません。Oracle は、少なくとも PCTUSED の使用率までデータ・ブロックを維持しようとします。これは、全体の領域からオーバーヘッドを差し引いた後、データに対して使用可能なブロック領域に対する割合（%）です。

PCTUSED のデフォルト値は 40% です。PCTUSED と PCTFREE の合計が 100 を超えない限り、0 ～ 99 の任意の整数を指定できます。

PCTUSED を小さくすると、次のようになります。

- 通常、ブロックは、PCTUSED の値が大きいときより、余裕がある状態に維持されます。
- PCTUSED より低くなったブロックを空きリストへ移動するため、UPDATE 文および DELETE 文の実行時に必要な処理コストが削減されます。
- データベース内の未使用領域が増加します。

PCTUSED を大きくすると、次のようになります。

- 通常、ブロックは、PCTUSED の値が小さいときより一杯の状態に維持されます。
- 領域効率が改善されます。
- INSERT 文および UPDATE 文の実行時の処理コストが増加します。

関連した PCTUSED および PCTFREE の値の選択

PCTFREE および PCTUSED にデフォルト値を使用しない場合は、次のガイドラインに従ってください。

- PCTFREE と PCTUSED の合計は 100 以下である必要があります。
- 合計が 100 より少ない場合、領域使用率および I/O パフォーマンスの理想的な妥協点は、PCTFREE と PCTUSED の合計値が、使用可能ブロック内で平均的な行が占有する割合を 100 から引いた値です。たとえば、データ・ブロック・サイズは 2048 バイトで、オーバーヘッドの 100 バイトを引いた 1948 バイトがデータに対して使用できるとします。平均的な行が 195 バイト、つまり 1948 の 10% を必要とする場合、PCTUSED と PCTFREE を適切に組み合わせて合計が 90% になる点で、データベース領域が最適な状態で使用できます。

- 合計が 100 の場合、Oracle は空き領域が PCTFREE を超えないようにするため、処理コストが最も高くなります。
- 固定値であるブロック・オーバーヘッドは、PCTUSED または PCTFREE の計算に含めません。
- PCTFREE と PCTUSED の合計と 100 との差が小さいほど（PCTUSED が 75、PCTFREE が 20 の場合など）、あるパフォーマンス・コストで領域使用の効率がよくなります。

PCTFREE および PCTUSED の値の選択例

次に、与えられたいくつかのシナリオで、PCTFREE および PCTUSED の値を正しく指定する方法を示します。

例 1

シナリオ：	一般的なアクティビティに、行サイズを増加する UPDATE 文が含まれています。パフォーマンスは重要です。
設定：	PCTFREE = 20 PCTUSED = 40
説明：	PCTFREE を 20 に設定して、更新の結果によってサイズが大きくなる行に対して十分な余裕をとります。PCTUSED を 40 に設定して、実行頻度の高い更新アクティビティ時に行われる処理を削減し、パフォーマンスを改善します。

例 2

シナリオ：	ほとんどのアクティビティに INSERT 文、DELETE 文および UPDATE 文が含まれていますが、影響を受ける行のサイズは増加しません。パフォーマンスは重要です。
設定：	PCTFREE = 5 PCTUSED = 60
説明：	ほとんどの UPDATE 文は行サイズを増加させないため、PCTFREE を 5 に設定します。PCTUSED を 60 に設定することで、DELETE 文によって解放される領域が比較的すぐに使用され、処理量も最小限に抑えられます。

例 3

シナリオ：	表が非常に大きいため、まず、記憶域を考慮する必要があります。ほとんどのアクティビティには、読取り専用トランザクションが含まれません。したがって、問合せのパフォーマンスが重要です。
設定：	PCTFREE = 5 PCTUSED = 90
説明：	UPDATE 文はほとんど発行されないため、PCTFREE を 5 に設定します。各ブロックでより多くの領域が表データの格納に使用されるように、PCTUSED を 90 に設定します。PCTUSED をこのように設定することによって、表データの格納に必要なデータ・ブロックの数が減少し、問合せのためにスキャンするデータ・ブロックの平均数が少なくなるため、問合せのパフォーマンスが向上します。

表の作成に必要な権限

使用するスキーマに新しい表を作成するには、CREATE TABLE システム権限が必要です。別のユーザーのスキーマに表を作成するには、CREATE ANY TABLE システム権限が必要です。さらに、表の所有者には、その表を含む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。

表の変更

Oracle データベース内の表は、次のような場合で変更できます。

- 表に 1 つ以上の新しい列を追加する場合
- 表に 1 つ以上の整合性制約を追加する場合
- 既存の列定義（データ型、長さ、デフォルト値、NOT NULL 整合性制約）を変更する場合
- データ・ブロック領域使用パラメータ（PCTFREE、PCTUSED）を変更する場合
- トランザクション・エントリ設定（INITRANS、MAXTRANS）を変更する場合
- 記憶域パラメータ（NEXT、PCTINCREASE など）を変更する場合
- 表に対応付けられた整合性制約を使用可能または使用禁止にする場合
- 表に対応付けられた整合性制約を削除する場合

表の列定義を変更する場合、既存の列の長さを大きくできるのは、表にレコードが存在していないときに限ります。また、空の表の列の長さを小さくすることもできます。データ型が CHAR の列の長さを拡大すると、特に表に数多くの行が含まれている場合には相当な追加記憶域を必要とし、時間を浪費する操作になる場合があります。これは、新しい長さになるまで、各行の CHAR 値に空白が埋め込まれるためです。

データ型を（たとえば、VARCHAR2 から CHAR へ）変更しても、その列内のデータは変更されません。ただし、空白埋込みが必要であるため、新しい CHAR 列の長さは変わる場合があります。

表を変更するには、SQL コマンド ALTER TABLE を使用します。たとえば、次のとおりです。

```
ALTER TABLE Emp_tab
  PCTFREE 30
  PCTUSED 60;
```

表の変更は、次のようなことを意味しています。

- 新しい列が表に追加される場合、列の初期値は NULL となります。表が行を含まない場合のみ、NOT NULL 制約付きの列をその表に追加できます。
- ビューまたは PL/SQL プログラム・ユニットがベース表に依存している場合、ベース表を変更すると依存オブジェクトに影響する場合があります、依存オブジェクトは必ず無効になります。

表の変更に必要な権限

表を変更するには、その表が自スキーマに含まれているか、またはその表に対する ALTER オブジェクト権限または ALTER ANY TABLE システム権限のいずれかを持っている必要があります。

表の削除

表を削除するには、SQL コマンド DROP TABLE を使用します。たとえば、次の文は、EMP_TAB 表を削除します。

```
DROP TABLE Emp_tab;
```

削除する表の中に、他の表の外部キーによって参照される主キーまたは一意キーが含まれている場合、その子表の外部キー制約を削除するには、次のように DROP TABLE コマンドに CASCADE オプションを指定します。たとえば、次のとおりです。

```
DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

表の削除による影響は、次のとおりです。

- 表定義がデータ・ディクショナリから削除され、その表のすべての行にアクセスできなくなります。
- 表に対応付けられたすべての索引およびトリガーは削除されます。
- 削除された表に依存していたすべてのビューおよび PL/SQL プログラム・ユニットはそのまま残りますが、無効になります（使用できません）。
- 削除された表に対するすべてのシノニムはそのまま残りますが、使用するとエラーが戻ります。
- 削除された表がクラスタ化されていない表の場合、割り当てられていたすべてのエクステンツは表領域の空き領域に戻され、新しいエクステンツを必要とする他のオブジェクトによってその領域が使用されます。
- クラスタ化された表に対応するすべての行は、そのクラスタのブロックから削除されます。
- その表がスナップショットのマスター表である場合、そのスナップショットは削除されませんが、スナップショットのログは削除されます。スナップショットは使用できますが、表を再作成しない限りリフレッシュできません。

表の定義を残してすべての行を削除する場合は、TRUNCATE TABLE コマンドを使用します。

参照：『Oracle8i 管理者ガイド』を参照してください。

表の削除に必要な権限

表を削除するには、その表が自スキーマに含まれているか、または DROP ANY TABLE システム権限を持っている必要があります。

一時表の管理

一時表の定義または構造は通常の表と同じように存続しますが、表の中に含まれているデータは、1つのトランザクションまたはセッション内でのみ存在します。Oracle8i では、そのセッション専用のデータを保持する一時表を作成できます。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。

一時表が役に立つ例をいくつか示します。

- 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。旅行日程を1つ作成するたびに、アプリケーションはそのデータを1つの一時表の1つの行に入れます。旅行日程を変更すると、アプリケーションによって該当する行が更新されます。最後に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。

セッション中に入力する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クローズすると、そのセッションのデータは自動的に削除されますが、表の構造は存続して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、2-13 ページの「[例 2: 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、現在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一度に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 1: 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 2: 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度のみ行われ、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。

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、ベース表と呼ばれます。ベース表は、表でもビュー自身でもかまいません。

ビューに対して行われる操作は、すべてビューのベース表に影響します。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 を含んだプログラムがコンパイルされたとき、ビューの所有者のスキーマに関連して変換されます。

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。ただし、ビューを定義する問合せには、`ORDER BY` 句または `FOR UPDATE` 句を含むことはできません。

`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。さらに、`WITH CHECK OPTION` は、そのビューに対して発行される `INSERT` 文および `UPDATE` 文は、ビューで選択できない行を作成したり、結果として戻したりできないという制約付きのビューを作成します。

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、その結果として生じる問合せをデータ・ディクショナリに格納するとき、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限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 オプションを指定した場合に限り、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しか使用できません。
- 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まったく影響を受けません。
- 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。

ビューの使用

表と同じ方法でビューを問い合わせることができます。たとえば、Division1_staff ビューを問い合わせるには、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、ベース表に行を挿入、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 句またはグループ関数を含む問合せによって定義される場合、ビューを使用しているベース表に行を挿入、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、そのビューがベース表から行を選択できない場合は、ベース表に対して（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ベース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ベース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、更新したりできます。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、その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にエラーがあったか、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をそれぞれ明示的に、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。たとえば、次のとおりです。

```
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 Server 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必要ありません。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 ベース表が変更され、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 ベース表または DEPT_TAB ベース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が、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、『Oracle8i 概要』を参照してください。

表の例

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、または固有の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上での基本です。表は、その表の各キーが結合の結果のキーになる可能性がある場合にキー保存されます。したがって、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：

- ある表をキー保存表にするために、その表のキー（複数の場合もある）を選択する必要はありません。その表のキー（複数の場合もある）を選択したときに、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 人の従業員しか存在しない場合、EMP_TAB と DEPT_TAB とを結合すると 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、基礎となるベース表を 1 つしか変更できません。

UPDATE 文

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（複数表にマップする列を変更できません。）で失敗します。

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。

DELETE 文

結合内でキー保存表が 1 つしかない場合は、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;
  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、このようなビューからは行を削除できません。たとえば、次のとおりです。

```
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、キー保存表の内部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。

INSERT 文

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ベース表が 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、同じ理由で正常に実行されません。ベース表 EMP_TAB のこの UPDATE も正常に実行されません。理由は、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ベース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 句を使用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	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_VIEWS	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んでいるビューの場合、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。たとえば、次の文を指定したとします。

```
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、ビュー USER_UPDATABLE_COLUMNS 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_NAME	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・ジェネレーターは連続的な数値を生成します。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し、複数の列または行にまたがるキーを調整する上で有効です。

順序がないと、連続的な値はプログラムによってしか生成できません。新しい主キーの値は、最も最近に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。このような待機をシリアライズ化と呼びます。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ライズ化がなくなり、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、順序がそれ以上、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 Parallel Server を使用するときの順序番号のキャッシュの詳細は、Oracle8i Parallel Server マニュアル・セット (『Oracle8i Parallel Server 概要』、『Oracle8i Parallel Server セットアップおよび構成ガイド』、『Oracle8i Parallel Server 管理、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、2-33 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が完了するのを待たずに、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);
```

または、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 が複数回参照されると、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 演算子を指定した SELECT 文
- GROUP BY 句または ORDER BY 句を指定した 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から構成されます。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、順序番号は高速にアクセスされます。ただし、アプリケーションが順序キャッシュ内には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、ディスクの読み込みが頻繁に必要な可能性があります。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。

順序キャッシュ内のエントリ数は、初期化パラメータ `SEQUENCE_CACHE_ENTRIES` によって決まります。このパラメータのデフォルト値は 10 エントリです。Oracle は、内部的に、監査、システム権限の付与、オブジェクト権限の付与、プロファイル、ストアド・プロシージャのデバッグ、ラベルなどのために順序を作成し、使用します。アプリケーションが使用する順序のみでなく、これらの順序も保持できるように、順序キャッシュに十分なエントリ数を確保していることを確認してください。

`SEQUENCE_CACHE_ENTRIES` パラメータの値が小さすぎると、順序の値がスキップされる可能性があります。たとえば、このパラメータが 4 に設定され、同時に使用される順序が 4 つキャッシュされているとします。

5 つ目の順序を作成すると、その順序がキャッシュ内で最も長い間使用されていない順序に置換されます。この置換された順序の残りのすべての値は失われます。つまり、置換された順序が最初に 10 個の順序値をキャッシュし、その中の 1 つの順序値のみが使用されていた場合、順序が置換された時点で 9 個の順序値が失われることになります。

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、エクスポートとインポートの後で、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、ディスクの読み込みが必要になります。このようなディスクの読み込みは、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。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、ファンクションまたはパッケージの別名です。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。

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 文のシノニムを使用することもできますが、別の DML 文とともに使用できません。

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を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。付与されたオブジェクト権限を使用して参照できるのは、別のユーザーのシノニムのみです。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 シノニムを問い合わ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 を使用して、1 回の操作で複数の表およびビューを作成したり、複数の権限を付与したりできます。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 拡張（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定義域から構成されます。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・データベース名から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そのオブジェクト名が一意であることが保証されるため、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渡ってそのオブジェクト名の一意性を保証できます。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 文における参照オブジェクトの名前変換

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。カレント・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 つのオブジェクトが、同じ名前を持つことはできます。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変換する場合は、次のルールを使用します。

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られて、ステップ 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チェックします。
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、ステップ 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に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ると、リモート Oracle セッションは、前述したように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変換の詳細は、『Oracle8i 概要』を参照してください。

スキーマ・オブジェクトの改名

必要に応じて、2 つの違った方法を使用してスキーマ・オブジェクトを改名できます。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を改名します。

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・シノニムを改名するために 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 ビューを改名します。

```
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改名できません。そのようなオブジェクトを改名するには、削除してから再作成します。

スキーマ・オブジェクトを改名すると、次のような影響があります。

- 改名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）。
- 改名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。

オブジェクトの改名に必要な権限

オブジェクトを改名するには、そのオブジェクトの所有者である必要があります。

スキーマの改名

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、まだカレント・ユーザーの権限しか付与されておらず、前述の ALTER SESSION 文による追加権限はまだ取得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

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。

- Oracle 組込みデータ型
- ANSI/ISO データ型、DB2 データ型および SQL/DS データ型
- データ変換

参照： オブジェクト型、VARRAY、NESTED TABLE などの複合型の詳細は、『Oracle8i 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。

Oracle 組込みデータ型

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
- DATE データ型
- バイナリ・データ型
 - BLOB
 - BFILE
 - RAW
 - LONG RAW

これらの他にデータ型 ROWID 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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。LOB データ型の詳細は、『Oracle8i 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 組込みデータ型に関する概要を示します。

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 (<i>size</i>)	長さが <i>size</i> バイトの固定長文字データ	表の各行で固定されます（後続ブランクが付きます）。最大サイズは 1 行あたり 2000 バイトで、デフォルト・サイズは 1 行あたり 1 バイトです。 <i>size</i> を設定する前に、キャラクタ・セット（シングルバイトかマルチバイトか）を考慮してください。
VARCHAR2 (<i>size</i>)	可変長文字データ	各行で可変で、1 行あたり最大 4000 バイトです。 <i>size</i> を設定する前に、キャラクタ・セット（シングルバイトかマルチバイトか）を考慮してください。最大 <i>size</i> を指定する必要があります。
NCHAR (<i>size</i>)	長さが <i>size</i> 文字または <i>size</i> バイト（各国語キャラクタ・セットによって異なる）の固定長文字データ	表の各行で固定されます（後続ブランクが付きます）。列サイズは固定幅の各国語キャラクタ・セットの文字数、または可変幅の各国語キャラクタ・セットの文字数です。最大サイズは 1 文字を格納するために必要なバイト数に応じて決まり、1 行あたり最大 2000 バイトです。デフォルトは 1 文字または 1 バイトです（キャラクタ・セットによって異なります）。
NVARCHAR2 (<i>size</i>)	長さが <i>size</i> 文字または <i>size</i> バイト（各国語キャラクタ・セットによって異なる）の可変長文字データ 最大 <i>size</i> を指定する必要があります。	各行で可変です。列サイズは固定幅の各国語キャラクタ・セットの文字数、または可変幅の各国語キャラクタ・セットの文字数です。最大サイズは 1 文字を格納するために必要なバイト数に応じて決まり、1 行あたり最大 4000 バイトです。デフォルトは 1 文字または 1 バイトです（キャラクタ・セットによって異なります）。
CLOB	シングルバイト文字データ	最大 2 の 32 乗 - 1 バイト（4GB）です。

表 3-1 Oracle 組み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
NCLOB	シングルのバイトまたは固定長マルチバイトの各国語キリカタ・セツ（NCHAR）データ	最大 2 の 32 乗 - 1 バイト（4GB）です。
LONG	可変長文字データ	表の各行で可変で、1 行あたり最大 2 の 31 乗 - 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 は 38（あるいはその両方）	各行で可変です。指定された列に必要な空白の最大値は、1 行あたり 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 日から西暦 4712 年 12 月 31 日までの範囲です。	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。
BLOB	非構造化バイナリ・データ	最大 2 の 32 乗 - 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 の 32 乗 - 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 行あたり最大 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 行あたり最大 2 の 31 乗 - 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。

文字データ型の使用

文字データ型は、英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)。
- 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：『Oracle8i 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： 3-18 ページの「[LONG データおよび LONG RAW データに関する制限](#)」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 データ型を使用してください。

表の英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。

空白の使用方法

- データをより効率的に格納するために、VARCHAR2 データ型を使用してください。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ブランクを格納しますが、VARCHAR2 データ型では列値に空白を埋め込まず、後続ブランクも格納しません。

比較方法

- 比較方法で ANSI 互換性が必要な場合、つまり文字列の比較において後続ブランクが重要でない（比較対象としない）場合には、CHAR データ型を使用してください。文字列の比較において後続ブランクが重要である（比較対象とする）場合には、VARCHAR2 データ型を使用してください。

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。

シングルスバイト・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、文字単位ではなく、バイト単位で指定され、指定したとおりに制約されます。NCHAR 列および NVARCHAR2 列の長さは、使用する各国語キャラクタ・セットに応じて、バイト単位または文字単位で指定します。

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ときには、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スバイトである場合、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。

参照：

- 『Oracle8i NLS ガイド』
- 『Oracle8i SQL リファレンス』
- 『Oracle8i Time Series ユーザーズ・ガイド』

Oracle の各国語サポート機能および各種文字コード体系のサポートの詳細は、これらのドキュメントを参照してください。

比較方法

Oracle は、空白埋め比較方法によって 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文字ごとに比較していきます。異なる文字がある場合、最初に異なっていた文字が大きい方の値が、より大きい値とみなされます。後続ブランクの文字数のみが異なる場合、2つの値は等しいものとみなされます。

Oracle は、非空白埋め比較方法により 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、等しいものとみなされます。Oracle は、異なる文字が見つかるところまで、2つの値を1文字ごとに比較していきます。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も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フル・テーブル・スキャンでは、CHAR 列に同じデータが格納されている表のフル・テーブル・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フル・テーブル・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方法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 に後続ブランク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方法の詳細は、『Oracle8i リファレンス・マニュアル』を参照してください。

NUMBER データ型の使用

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、最大小数桁数 38 桁の精度を提供します。1 × (10 の -130 乗) ~ 9.99... × (10 の 125 乗) の正と負の数値および 0 (ゼロ) を、NUMBER 列に格納できます。

数値列の場合、その列を浮動小数点数として指定できます。

Column_name NUMBER

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

Column_name NUMBER (<precision>, <scale>)

数字フィールドに対して精度およびスケールを指定することによって、入力時に整合性チェックを追加できます。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。[表 3-2](#) 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	格納	指定
7,456,123.89	NUMBER	7456123.89

表 3-2 スケール位置の数値データ記憶域への影響（続き）

入力データ	格納	指定
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、『Oracle8i 概要』を参照してください。

DATE データ型の使用

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。

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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

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-1992'
```

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。ユリウス暦日付に関する情報は、『Oracle8i 概要』を参照してください。

日付書式 DD-MON-YY が使用される場合、YY は 20 世紀の年を示します（たとえば、31-DEC-92 は 1992 年の 12 月 31 日です）。20 世紀以外の世紀の年を示す場合は、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。

時刻書式

時刻は 24 時間形式 HH:MM: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と、日付フィールドの中の時刻は 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.'));
```

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。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。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。

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、日付入力の実入、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。

以上の条件は、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）

Oracle Server の西暦 2000 年対応

Oracle Server は西暦 2000 年に対応しています。Oracle Server、ネットワーク製品およびシステム管理製品で操作上の問題は想定されていません。オラクル社の開発部門では、新世紀に移るときにユーザーへの影響が何もないことを検証するために、様々な西暦 2000 年操作シナリオに基づいてテストを実施しています。これらのシナリオには、レプリケーション、Point-in-Time リカバリ、分散トランザクションなどのテストが含まれています。システム管理機能およびネットワーク機能は、タイムゾーン / 日付変更線 / 世紀にまたがったテストも実施済です。

Oracle 製品は西暦 2000 年対応済ですが、アプリケーションのテストの必要がないということではないので注意してください。最も重要なことは、アプリケーション・ソフトウェアを Oracle Server 上でテストして、西暦 2000 年関連の操作が期待どおり実行されることを確認する必要があるということです。このようなテストをせずに 2000 年対応を保証する汎用プロトコル定義は存在しないため、アプリケーション・ソフトウェアが西暦 2000 年対応として認証されている場合でも、このテストは非常に重要です。

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 といった値ではなく、1996 または 2001 といった値が格納されます。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 および Oracle8 Server) を使用し、日付値または日付時刻値 (あるいはその両方) に DATE データ型を使用しているアプリケーションでは、西暦 2000 年が近付いても、格納データについて心配する必要は何もありません。Oracle7 および Oracle8 Server 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 19xx の形を取るなど) を基に作成されているものもあります。アプリケーションではデータベースに 2 桁の年が渡されたり、世紀を決定するために Oracle が使用するプロシージャが、プログラマの予想するものとは異なったりする可能性があります (3-15 ページの「[プログラミング上のヒント](#)」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

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 19xx 形式に、00 ~ 49 の年はデフォルトの 20xx 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、現在 1996 年で 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、現在 1996 年で 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、現在 2001 年で 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、現在 2001 年で 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、または問い合わせる場合には、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

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。CC 日付書式の例を次に示します。

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

これによって、次の結果が生成されます。

```
CC
----
20
```

CC 日付書式の 2 つ目の例を次に示します。

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

これによって、次の結果が生成されます。

```
CC
----
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。厳密に言えば、「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 世紀で言うと 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、『Oracle8i 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勧めします。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、「YYYY/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、あるいはユーザーまたは他のプログラムから受け取るときは、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「YYYY/MM/DD HH24:MI:SS」には、次のような利点があります。

- 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ので、キャラクタ・ベースのソート・オペレーション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` – インスタンス全体の NLS パラメータが `INIT.ORA` に明示的に宣言されているかデフォルト設定かを示します。
- `NLS_SESSION_PARAMETERS` – `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。

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 レジストリおよび 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勧めします。

プログラミング上のヒント

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、よくあるものをいくつか説明します。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、詳細な調査によると、Oracle テクノロジを正しく利用していないことが原因であるとわかりました。

例 1

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。

例 2

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。

例 3

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。

例 4

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。ただし、西暦 2000 年はうるう年なので、これは正しくありません。

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 Server 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。

例 5

DECODE 関数が使用されたときに、3 番目の引数がデータ型 CHAR か VARCHAR2、または NULL の場合、戻り値はデータ型 VARCHAR2 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。

例 6

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);
```

例 7

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、「where col '12-MAY-99」などの 2 桁の年の述語は、ビュー内で使用できます。4 桁の年の解析は、NLS_DATE_FORMAT の設定によって異なります。

LONG データ型の使用

注意： LONG 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。CLOB データ型および NCLOB データ型の詳細は、『Oracle8i 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 の情報を含む可変長文字データを格納できます。LONG 列の長さは、使用しているコンピュータで利用できるメモリによって制限される場合があります。

LONG として定義された列は、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 列には、VARCHAR2 列と同じ特性が多数あります。

LONG データおよび LONG RAW データに関する制限

LONG（および LONG RAW、次を参照）列には多くの用途がありますが、使用方法には次のようにいくつかの制約があります。

- 表あたり 1 列の LONG 列しか使用できません。
- LONG 列には索引を付けることができません。
- LONG 列は整合性制約に使用できません。
- LONG 列は、WHERE 句、GROUP BY 句、ORDER BY 句または CONNECT BY 句の中、SELECT 文の DISTINCT 演算子と一緒に使用できません。
- LONG 列は、SUBSTR や INSTR などの SQL 関数によって参照できません。
- LONG 列は、副問合せの SELECT 構文のリスト、または集合演算子（UNION、INTERSECT または MINUS）による複合型の問合せでは使用できません。
- LONG 列は SQL 式では使用できません。
- 問合せ（CREATE TABLE... AS SELECT...）を使用して表を作成する場合、または問合せ（INSERT INTO... SELECT...）を使用して表またはビューに挿入する場合は、LONG 列を参照できません。
- 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せん。

- LONG データ型または LONG RAW データ型を使用して、データベース・トリガーの変数を宣言できません。
- :NEW データベース・トリガーおよび :OLD データベース・トリガーへの参照は、LONG 列または LONG RAW 列と一緒に使用できません。
- LONG 列および LONG RAW 列は、分散 SQL 文では使用できません。
- LONG 列および LONG RAW 列は、レプリケートできません。

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、それぞれの 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、別の表に格納してください。この 2 つの表は、参照整合性制約によって対応付けることができます。このように設計することによって、関連するデータのみアクセスする SQL 文は、LONG データまたは 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わ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 表を問い合わ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、『Oracle8i 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。

Net8、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、3-13 ページの「[LONG データおよび LONG RAW データに関する制限](#)」を参照してください。

ROWID および ROWID データ型

Oracle データベースの非クラスタ化表の各行に、行の行断片（複数の行断片の間で連鎖される行の場合には最初の行断片）の物理アドレスに対応する一意の ROWID が割り当てられます。クラスタ化表の場合、同じデータ・ブロックに存在する異なる表の中の行は、同じ ROWID を持つことができます。

Oracle データベース内の各表は、ROWID という名前の疑似列を内部的に持っています。

参照： ROWID 疑似列および ROWID データ型についての一般的な情報は、『Oracle8i 概要』を参照してください。

拡張 ROWID 形式

Oracle Server 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、『Oracle8i PL/SQL 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。

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。この項では、そのような様々な使用方法を説明します。

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。

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。

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、OOOOOFFFFBBBBBRRR の拡張 ROWID のコンポーネントを base64 にコード化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、ファイル AAL）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、ブロック AAAAQk）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有する 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、行 AAA）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、『Oracle8i PL/SQL 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。

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 ROWID および制限付き ROWID のサイズと同じです。拡張 ROWID についての情報は、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 は大量の 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：『Oracle8i 管理者ガイド』を参照してください。

Oracle7 クライアントは、Oracle8 データベースにアクセスできます。同様に、Oracle8 クライアントは、Oracle7 サーバーにアクセスできます。つまり、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めることができます。

参照： ROWID_TO_EXTENDED 関数の詳細は、『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』および『Oracle8i 移行ガイド』を参照してください。

Oracle8 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、Oracle8 では、Oracle7 またはそれ以前のサーバーに ROWID 値を戻す場合に、制限付き ROWID を使用します。

Oracle7 Server 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 Oracle8 データベースへのアクセス** Oracle8 では、ROWID が拡張形式で戻されます。つまり、次の操作しか行えません。
- 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 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)

IBM 社の製品 SQL/DS と DB2 のデータ型 TIME、TIMESTAMP、GRAPHIC、VARGRAPHIC および LONG VARGRAPHIC には、対応する Oracle データ型がないため使用できません。

TIME データ型および TIMESTAMP データ型は、Oracle データ型 DATE の副構成要素です。
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)
DATE	DATE

データ変換

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_DATE()
- HEXTORAW()
- RAWTOHEX()
- ROWIDTOCHAR()
- CHARTOROWID()

暗黙的なデータ型変換が、この後に説明するルールに従って行われます。

ルール 1: 割当て

割当てでは、Oracle は次のような変換を自動的に行うことができます。

- VARCHAR2 または CHAR から NUMBER へ
- NUMBER から VARCHAR2 へ
- VARCHAR2 または CHAR から DATE へ
- DATE から VARCHAR2 へ
- VARCHAR2 または CHAR から ROWID へ
- ROWID から VARCHAR2 へ
- VARCHAR2 または CHAR から HEX へ
- HEX から VARCHAR2 へ

Oracle が、割り当てる値のデータ型を割当て先のデータ型に変換できる場合、割当て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： 次のように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、次に示す割当て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 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 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割り当てる前に、自動的に変換します。

```
SELECT Col1 INTO Var1 FROM Table1_tab WHERE Col1 = 30;
```

ルール 2: 式の評価

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。

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。文字列が初期化パラメータ NLS_DATE_FORMAT 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

```
Comm + '500'
```

- 次のようなブール式

```
Bonus > Sal / '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 = 'AAAAaoAATAAADAAA'
```

通常、割当て変換のルール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、次の式評価のルールを使用します。

次のような割当ての場合、

```
variable := expression
```

Oracle は、ルール 2 が行う変換を最初に使用して、*expression* を評価します。*expression* は単純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、ルール 1 を使用して、この値を割り当てようとします。

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。

- [整合性制約の使用](#)
- [参照整合性制約の使用](#)
- [分散データベース内の参照整合性](#)
- [CHECK 整合性制約の使用](#)
- [整合性制約の定義](#)
- [整合性制約の使用可能および使用禁止](#)
- [整合性制約の変更](#)
- [整合性制約の削除](#)
- [外部キー整合性制約の管理](#)
- [整合性制約定義のリスト](#)

整合性制約の使用

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。いったん整合性制約が使用可能になると、表のデータはすべて指定したルールに従う必要があります。その後、表のデータを変更する SQL 文を発行すると、Oracle は結果のデータが整合性制約を満たしていることを保証します。整合性制約を使用しない場合、このようなビジネス・ルールは、アプリケーションのプログラム中で施行する必要があります。

整合性制約でビジネス・ルールを施行する場合

整合性制約によってルールを施行すると、アプリケーションで SQL 文を発行して同じルールを施行するより信頼性が高くなります。整合性制約のセマンティクスは明確に定義され、Oracle がそれらを施行するための内部処理は、SQL より下位レベルで最適化されます。アプリケーションは SQL を使用するため、このレベルでの最適化を行うことはできません。

例 EMP_TAB 表の各従業員が、DEPT_TAB 表にリストされている部門に確実に所属しているようにするには、最初に、DEPT_TAB 表の DEPTNO 列に主キー制約を次のように作成します。

```
ALTER TABLE Dept_tab
  ADD PRIMARY KEY (Deptno);
```

次に、DEPT_TAB 表の主キーを参照する EMP_TAB 表の DEPTNO 列に参照整合性制約を作成します。次に例を示します。

```
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、その部門番号は Oracle によって確実に部門表の中に置かれます。

整合性制約を使用しないでこのルールを施行するには、トリガーを使用して、各新規従業員レコードの部門番号が、既存の部門に属することを確認するテストを行います。そのためには、SELECT 文を発行して DEPT_TAB 表を問い合わせる必要があります。ただし、Oracle の SELECT は、読取り一貫性を使用するため、この問合せは、別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。整合性制約によって、この問題は回避されます。

アプリケーションでビジネス・ルールを施行する場合

場合によっては、整合性制約のみでなく、アプリケーションでビジネス・ルールを施行する場合があります。アプリケーションでビジネス・ルールを施行することによって、整合性制約より早くユーザーに結果をフィードバックできる場合があります。たとえば、ユーザーがアプリケーションに 20 件の値を入力するために、これらの値を含む INSERT 文を発行する場合、ビジネス・ルールに違反する値が入力された時点で、すぐにユーザーに知らせる必要があります。

整合性制約は SQL 文の発行時にのみ施行されるため、ユーザーが 20 件のすべての値を入力し、アプリケーションが INSERT 文を発行した後でなければ、ユーザーに不正な値があったことを知らせることができません。ただし、値が入力されたときにその整合性を検証し、不正な値を見つけた場合、すぐにユーザーに知らせるようにアプリケーションを設計できます。

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索引が必要です。また、外部キーには常に索引を作成する必要があります。一意キーおよび主キーは一意索引を作成できますが、パフォーマンス上の理由で索引が必要な場合は、キー列に自動的に作成される索引は使用しないでください。かわりに、手動で索引を作成してください。

次のことに注意してください。

- 制約は既存の索引を使用します。必要がない限り、索引は作成されません。
- 一意キーおよび主キーは、一意索引と同様に非一意索引も使用できます。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チェック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。

NOT NULL 整合性制約の使用

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、EMP_TAB では、一時的に従業員のマネージャまたは入社日に値が入ってなくても、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。

このため、これら 3 つの列は、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、従業員の名前は各行に必要です。したがって、この列は NOT NULL 整合性制約に適しています。

NOT NULL 制約を他の整合性制約と組み合わせて、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わせます。このようにデータ整合性規則を組み合わせることで、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 NULL のキーを格納しません。したがって、表の索引のみをスキャンするか、またはすべての行に索引を付ける必要がある操作を実行するには、少なくとも 1 つの索引列に NOT NULL 制約を付けます。

参照： 4-10 ページの「親表と子表の関連」を参照してください。

図 4-1 NOT NULL 整合性制約

ENAME の制約は、ALTER TABLE "EMP" MODIFY "ENAME" NOT NULL で指定されています。

EMP表							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7329	SMITH	CEO		17-DEC-85	9,000.00		20
7499	ALLEN	VP-SALES	7329	20-FEB-90	7,500.00	100.00	30
7521	WARD	MANAGER	7499	22-FEB-90	5,000.00	200.00	30
7566	JONES	SALESMAN	7521	02-APR-90	2,975.00	400.00	30

NOT NULL制約
(この列ではどの行もNULL値を
含むことができません。)

NOT NULL制約なし
(この列ではどの行にNULL値が
含まれていてもかまいません。)

デフォルトの列値の設定

有効なデフォルト値は、任意のリテラル、あるいは順序、PL/SQL ファンクション、列、LEVEL、ROWNUM または PRIOR を参照しない任意の式を含みます。デフォルト値は、式 SYSDATE、USER、USERENV および UID を含むことができます。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。

デフォルト値を使用する場合

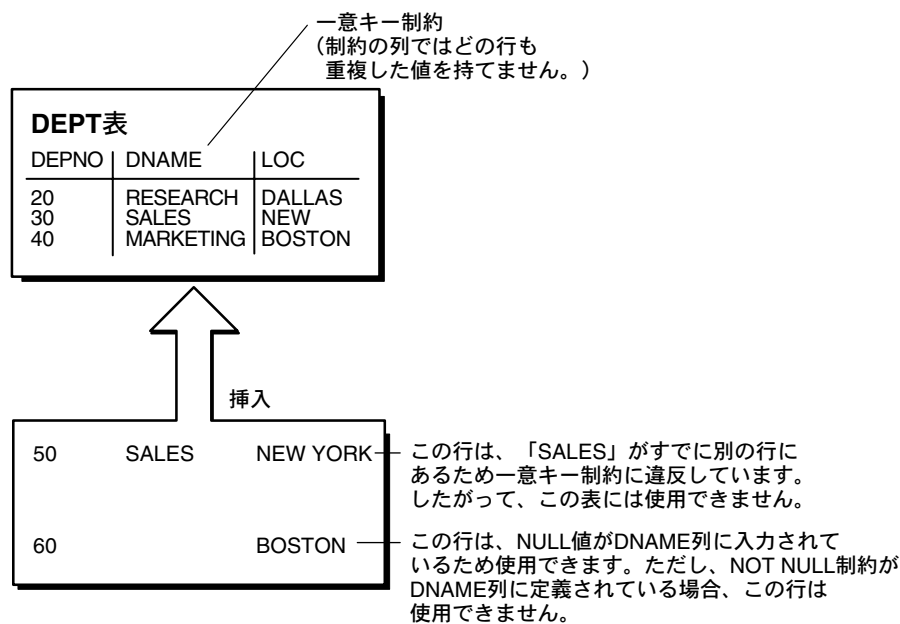
デフォルト値は、一般的な値を含む列にのみ割り当てます。たとえば、DEPT_TAB 表では、1 つの事業所にほとんどの部門がある場合、この値（たとえば、NEW YORK）を LOC 列のデフォルト値に設定できます。

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ときにも、デフォルトは便利です。たとえば、ビューを介してユーザーが表に行を挿入できるようにしていることがあります。ビューは、エンド・ユーザーの操作に関係するすべての列を表示するように定義されます。ただし、ベース表には、ビューの定義に含まれていない列で、表の各行を挿入するユーザーを記録する列 INSERTER がある場合もあります。列 INSERTER は、USER 関数を使用して定義することによって、行を挿入するユーザーの名前を記録できます。次に例を示します。

```
. . ., inserter VARCHAR2(30) DEFAULT USER, . . .
```

参照： デフォルト列値に関するもう 1 つの例は、2-4 ページの「表の作成」を参照してください。

図 4-2 一意キー制約



表の主キーの選択

1つの表につき、1つの主キーを持つことができます。主キーによって、表の各行を一意に識別することが可能になり、行が重複しないようにできます。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。

- 通常は、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に従うことができる簡単な方法です。
- データ値が一意となる列を選択します。

表の主キーの目的は、表の各行を一意に識別することにあります。したがって、主キーの列または列の集合は、各行に対して一意の値を含んでいる必要があります。

- 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。主キー値には、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、変更する必要性がほとんどないデータにする必要があります。

- NULL をまったく含まない列を選択します。
定義によって、主キー制約は、主キーを構成する任意の列に 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。
- コンポジット主キーの選択を避けます。
コンポジット主キーは使用できますが、前述の条件を満たしません。たとえば、コンポジット主キー値は長いため、順序番号を割り当てることができません。

一意キー整合性制約の使用

一意キーは、慎重に選択してください。多くの場合、本来は表の主キーの一部にする必要がある列に、間違っ一意キーが構成されていることがあります（主キーについての詳細な説明は、前述の項を参照してください）。一意キー制約は、表の行の間でキー値の重複を避ける場合にのみ使用します。一意キーのデータが表上で重複することはできません。

注意： 一意キー制約では NULL を入力できますが、一意キー制約の検索は複数列にまたがって行われるため、一部が NULL のコンポジット一意キー制約である非 NULL 列内に同一の値を入力できません。

主キーと一意キーの概念を混同しないようにしてください。主キーは、表の各行を一意に識別するために使用します。したがって、一意キーには、表における行を識別する用途はありません。

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）

参照整合性制約の使用

2つの表が共通の列（または列の集合）によって対応付けられた場合は、2つの表の間の関連を維持するために、必ず主キーまたは一意キー制約をその親表の列で定義し、外部キー制約を子表の列で定義します。

参照： この関連によっては、4-10 ページの「[親表と子表の関連](#)」に示すように、外部キーを含むその他の整合性制約を定義する必要があります。

図 4-3 は、EMP_TAB 表の DEPTNO 列に定義された外部キーを示しています。この外部キーは、この列に含まれるそれぞれの値が DEPT_TAB 表の主キー（DEPTNO 列）の値と一致することを保証します。このため、EMP_TAB 表の DEPTNO 列に間違った部門番号が存在することはありません。

外部キーは、複数の列で構成することもできます。ただし、このようなコンポジット外部キーは、正確に同じ構造（列の数とデータ型が同一）を持つコンポジット主キーまたはコンポジット一意キーを参照する必要があります。コンポジット主キーまたはコンポジット一意キーには 32 列までという制限があるため、コンポジット外部キーも最大 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）、ANSI/ISO 規格にもあるとおり、外部キー制約はコンポジット外部キーに対して「不一致」規則を施行します。CHECK および NOT NULL 制約を使用して、次のように、「完全一致」および「部分一致」の規則を施行することもできます。

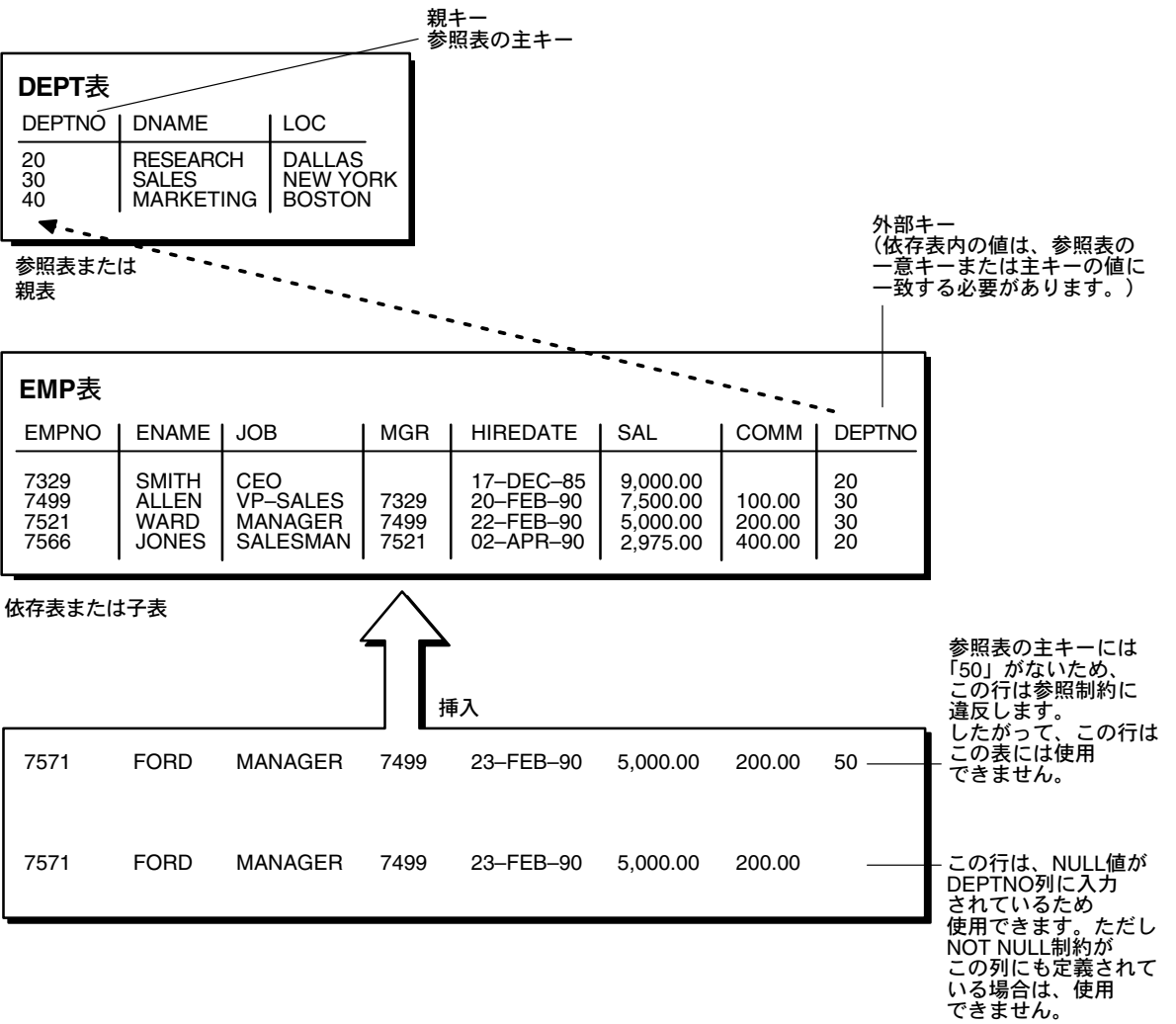
- コンポジット外部キーの NULL に対する「完全一致」規則（キーのすべての構成要素が NULL であるか、NULL 以外のものであることを要求する）を施行するには、すべてのコンポジット外部キーが NULL または非 NULL であることのみを許可する CHECK 制約を定義します。たとえば、列 A、B、C で構成されるコンポジット・キーを次のとおり指定できます。

```
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、宣言参照整合性を基にして、コンポジット外部キーの 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 12 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



親表と子表の関連

親表と子表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まったく定義されていないときは、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。EMP_TAB と DEPT_TAB の間のこのような関連の例は、[図 4-3](#) に示されています。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ときは、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っている必要があります。

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ときは、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、EMP_TAB 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。次の理由により、EMP_TAB 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ときは、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、EMP_TAB 表の MEMBERNO 列に NOT NULL 制約を追加することで、EMP_TAB 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

Oracle では、1 つの列を複数の外部キー制約で参照できます。事実上、依存キーの数に制限はありません。ある列が 2 つの異なるコンポジット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 が制約をチェックするときに、制約が満たされない場合は、エラーが表示されます。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。

SET CONSTRAINTS 文を発行すると、SET CONSTRAINTS モードは、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。

注意： SET CONSTRAINT 文は、トリガー内では発行できません。

参照：SET CONSTRAINTS 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

制約の一般的な情報は、『Oracle8i 概要』を参照してください。

制約チェックの遅延方法

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分からない場合
- 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 データの処理に使用するアプリケーション内で、実際に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、現行の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、制約が存在する限り維持されます。ALTER SESSION SET CONSTRAINTS 文は、現行のセッションにのみ適用されます。

コミットをチェックする（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チェックできます。制約に何か問題があった場合、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、制約の一意性を施行するために既存の索引が使用できるかどうかをチェックします。既存の索引がない場合、Oracle は索引を作成します。

Oracle が一意索引を使用して制約を施行しており、かつ、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。

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したりできません。

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。

同時実行性制御、索引および外部キー

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、『Oracle8i 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

Oracle では、宣言参照整合性制約を、分散データベースの複数のノードにまたがって定義できません（ある表に対する宣言参照整合性制約で、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 12 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には、ネットワーク障害が親表のみでなく子表へのアクセスも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（子表に行を挿入したり、子表の中の外部キーの値を更新したりするような処理）を進めることができない場合があります。

CHECK 整合性制約の使用

比較などの論理式をベースとした整合性ルール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](#)」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、EMP_TAB 表の SAL 列に 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、DEPT_TAB 表の LOC 列に CHECK 制約を定義します。
- 行の SAL 値と COMM 値を比較して、COMM 値が SAL 値よりも大きくならないように、SAL 列および COMM 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。条件が偽に評価される場合、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 SYSDATE、UID、USER または USERENV SQL 関数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、『Oracle8i 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 制約を使用するときには、条件が偽に評価される場合にのみ CHECK 制約に違反する、という ANSI/ISO 規格を考慮してください。つまり、真と不明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 制約が実際に必要な規則を施行す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、「従業員の給料が 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）より小さい場合は、EMP_TAB 表の行を許可しない」と解釈されます。ただし、給与に NULL、コミッションに負の値を持つ行は、チェック条件全体が不明と評価されるため、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、このような違反を処理できます。

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、『Oracle8i 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

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たりしないように注意してください。

CHECK および NOT NULL 整合性制約

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は、IS NOT NULL 条件を指定した CHECK 制約のかわりに、NOT NULL 整合性制約を定義する必要があります。

コンポジット・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 を構成するコンポジット・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

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に対して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、『Oracle8i 管理者ガイド』を参照してください。

CREATE TABLE コマンド

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

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。

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： 4-25 ページの「[外部キー整合性制約に必要な権限](#)」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ます。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のを回避できます。

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、4-27 ページの「[整合性制約定義のリスト](#)」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にしたり、使用禁止にしたりするしくみおよび手順について説明します。

使用可能にした制約 制約が使用可能な場合、制約によって定義した規則は、制約が定義されている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。

使用禁止にした制約 制約が使用禁止の場合、制約によって定義した規則は、制約に含まれる列のデータ値に施行されません。ただし、制約の定義は、データ・ディクショナリ内に格納されます。

つまり、整合性制約は、データベース内のデータに関する文と考えられます。この文は、制約を使用可能にすると必ず真になります。ただし、制約を使用禁止にすると、整合性制約に違反したデータがデータベース内に存在する可能性があるため、この文は真であることも真でないこともあります。

制約を使用禁止にする理由

整合性制約によって定義した規則を施行するには、必ずその制約を使用可能にする必要があります。ただし、状況によっては、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ことが望ましい場合があります。たとえば、次のような場合です。

- 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

このような場合、作業パフォーマンスの改善のために整合性制約を一時的に使用禁止にできます。

整合性制約違反

表の行が整合性制約を守らない場合、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ため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識別できます。

参照： この手順については、「[例外のレポート](#)」を参照してください。

定義時

CREATE TABLE 文または ALTER TABLE 文で整合性制約を定義するときには、その定義に ENABLE 句を指定して制約を使用可能にしたり、DISABLE 句を指定して制約を使用禁止にしたりできます。制約定義に ENABLE 句または DISABLE 句を指定しない場合、Oracle はその制約を自動的に使用可能にします。

制約を使用可能にする

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);
```

整合性制約を定義して使用可能にする ALTER TABLE 文を発行すると、表の行がその整合性制約に違反している場合、エラーとなる可能性があります。この場合、文はロールバックされ、制約定義は格納されず、また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、4-22 ページの「[例外のレポート](#)」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;
```

整合性制約を定義して使用禁止にする ALTER TABLE 文は、エラーとなることはありません。制約の定義は、その規則が施行されていないため許可されます。

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

使用禁止になっている制約を使用可能にする

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、4-22 ページの「[例外のレポート](#)」を参照してください。

使用可能になっている制約を使用禁止にする

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 – 参照のためのデータ・ディクショナリ使用方法： 前述の項の例文において、制約を使用可能または使用禁止にするために、制約についての情報を把握しておく必要があります。

たとえば、各項の最初の文では制約の名前を把握しており、2 番目の文では一意キーの列のリストを把握していることが前提条件になっています。このような情報を把握していない場合、制約について定義されているいくつかのデータ・ディクショナリ・ビューからこれらの情報を得ることができます。これらのビューの詳細は、4-27 ページの「[整合性制約定義のリスト](#)」および『Oracle8i リファレンス・マニュアル』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

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ときには、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： 4-25 ページの「[外部キー整合性制約の管理](#)」および『Oracle8i 管理者ガイド』を参照してください。

例外のレポート

CREATE TABLE... ENABLE... 文または ALTER TABLE... ENABLE... 文を発行するときに、整合性制約の例外があるために、これらの文が正常に実行されない場合、文はロールバックされ、制約に対するすべての例外を更新または削除しない限り、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、『Oracle8i 管理者ガイド』を参照してください。

整合性制約の変更

Oracle8 では、ENABLE 句または DISABLE 句を使用して変更できる制約状態は特定のものに限定されていました。Oracle8i では、MODIFY CONSTRAINT 句を使用して既存の制約状態を変更できるように機能拡張されています。

参照： 変更できるパラメータに関する詳細は、『Oracle8i 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

MODIFY CONSTRAINT の例 1

```
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

```
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

```
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;
```

注意： RELY および NORELY は、制約の作成時または変更時に設定または再設定できる新しい状態です。

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ときには、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： 4-25 ページの「[外部キー整合性制約の管理](#)」および『Oracle8i 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いて、それらの情報を補足します。

外部キー整合性制約の定義

次の内容は、外部キー整合性制約を定義するときに重要な情報です。

データ型の一致

参照整合性制約を定義するときに、依存する側の表と参照される側の表の対応する列名が一致している必要はありません。ただし、データ型は同じである必要があります。

コンポジット外部キー

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基に施行されるため、コンポジット外部キーは 32 列以内に制限されています。

主キーの暗黙的な参照

外部キー制約（1 列またはコンポジット列）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 によって自動的にチェックされません。参照していない場合には、エラー情報が戻されます。

外部キー整合性制約に必要な権限

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には、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できません。つまり、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。

外部キーに対する参照アクションの指定

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。

- **UPDATE/DELETE アクションなし制限** このアクションを指定すると、キーを参照する行が子表内にある場合、親キーは更新または削除されません。特に何も指定しなければ、すべての外部キー制約はアクションなし制限を施行します。アクションなし制限を施行するために、制約の定義時にオプションを指定する必要はありません。次に例を示します。

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

- **ON DELETE CASCADE アクション** この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（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。次に例を示します次に例を示します。

```
CREATE TABLE Emp_tab (  
  FOREIGN KEY (Deptno) REFERENCES Dept_tab  
    ON DELETE CASCADE);
```

- **ON DELETE SET NULL アクション** この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（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

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。

整合性制約定義のリスト

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
- CONSTRAINT_COLUMNS
- CONSTRAINT_DEFS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_CROSS_REFS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS
- DBA_CROSS_REFS

参照： 各ビューの詳細は、『Oracle8i リファレンス・マニュアル』を参照してください。

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。

- 制約名は、ユーザー指定（たとえば、DNAME_UKEY）のものも、システム指定（たとえば、SYS_C00275）のものもあります。
- 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 NOT NULL 制約であり、どれが EMP_TAB 表および DEPT_TAB 表の 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

索引計画の選択

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。内容は次のとおりです。

- [索引の管理](#)
- [ファンクション索引](#)
- [クラスタ、クラスタ化表およびクラスタ索引の管理](#)
- [ハッシュ・クラスタおよびクラスタ化表の管理](#)

参照： 次の情報については、次の各章またはマニュアル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](#) – [第9章](#)
- [オブジェクト型](#) – 『Oracle8i 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』
- [依存性情報](#) – [第9章](#)
- [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方法の詳細は、『Oracle8i レプリケーション・ガイド』を参照してください。](#)

索引の管理

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に制限はありません。ただし、索引によるパフォーマンス上の利点およびデータベース・アプリケーションのニーズを考慮して、索引を付ける列を判断してください。

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： 索引作成のパフォーマンスについては、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

表データ挿入後の索引の作成

重要な例外は 1 つありますが、索引は、通常、データが挿入された後、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てください。索引のない表に行データを挿入して、後から索引を作成した方が、より効率的です。表データがロードされる前に索引を作成すると、表に行を挿入するたびにすべての索引を更新する必要があります。このルールの例外は、クラスタにデータを挿入する前にクラスタに索引を作成しておく必要がある点です。

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（ユーザーあたりの容量は初期化パラメータ SORT_AREA_SIZE 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、次のステップ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： 『Oracle8i ユーティリティ・ガイド』を参照してください。

正しい表および列に索引を付ける 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、テーブル・スキャンの相対速度、および行データが索引キーについてどのようにクラスタ化されているかによって大きく変動します。テーブル・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。

参照： 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「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、索引付けの有力候補があります。次の特徴が 1 つでもある列は、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。
- 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、次の句

```
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)。

次の特徴を持つ列は、索引を付ける対象としてあまり適していません。

- 異なる値をほとんど持たない (たとえば、従業員の性別の列)。
- 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およそ 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。

表あたりの索引数を制限する 表は、索引をいくつでも持つことができます。ただし、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。

したがって、表に対する問合せの検索速度とその表に対する更新速度の間に相対的な関係が存在します。たとえば、表が主として読取り専用の場合には索引数を増やすと有効ですが、表がかなり頻繁に更新される場合は索引数を減らす方が有効です。

パフォーマンスのために索引列を順序付ける CREATE INDEX コマンドの中での列の指定順序は、表の中の順序に対応している必要はありません。ただし、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るため、重要です。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。

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 文によって問い合わせるとします。

```
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度の高い列（最も多くの値を持つ列）を先頭にしたコンポジット索引を作成します。

```
CREATE INDEX ind_vendor_id
  ON vendor_parts (part_no, vendor_id);
```

索引は、索引の先頭部分を使用して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 つしかないため、VENDOR_ID に単独で索引を指定しても有効ではありません。

索引の作成

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コンポジット索引を作成できます。コンポジット索引キーは、データ・ブロック内の使用可能領域のおよそ 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有効であり、廃止された CREATE UNIQUE INDEX 構文は明示的に使用しないようにします。

索引を作成するには、SQL コマンド CREATE INDEX を使用します。次の文のように指定してください。

```
CREATE INDEX emp_ename ON Emp_tab(ename)
  TABLESPACE users
  STORAGE (INITIAL      20K
           NEXT         20k
           PCTINCREASE 75)
  PCTFREE      0;
```

いくつかの記憶域設定が、索引に対して明示的に指定されていることに注目してください。

索引の作成に必要な権限

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。

索引の削除

索引は、次のような理由で削除することもあります。

- 関連する表に対して発行した問合せで、索引によって予想されたほどパフォーマンスが改善されていないため（表が非常に小さい、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、など）。
- アプリケーションに、索引を使用する問合せが含まれないため。
- 索引が不要になり、再作成の前に削除する必要があるため。

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、`EMP_ENAME` 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。

索引の削除に必要な権限 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。

ファンクション索引

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、列のみの場合よりも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： ファンクション索引を作成できるのは、Oracle8i 以降のリリースを使用する場合のみです。

ファンクション索引に使用される式は、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、言語ソート・キー（照合）に基づく言語ソート、SQL 文の効率的な言語照合、および大文字 / 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利点の詳細は、5-7 ページの「[ファンクション索引の使用](#)」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の詳細は、5-12 ページの「[ファンクション索引の要件および制限](#)」を参照してください。

参照： ファンクション索引の詳細は、『Oracle8i 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、『Oracle8i 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使用

ファンクション索引の利点を次に詳しく説明します。

- **オプティマイザがフル・テーブル・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て。

```
CREATE INDEX Idx ON Example_tab(Column_a + Column_b);  
SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
```

CREATE INDEX 文で、idx は索引の名前、Example_tab は表の名前、column_a および column_b は列を表します。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。レンジ・スキャンでは、通常、述語の選択性が低い場合（大きな表の 15% 未満の行が述語で選択された場合）は応答時間が短くなります。さらに、式がファンクション索引に含まれていると、オプティマイザは式に関係する述語の選択性を推定できます（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 コマンド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。**たとえば、計算集中型の式を頻繁にアクセスする場合は、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、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 関数を使用した大文字 / 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、さらに、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： CREATE INDEX 文の DESC キーワードが無視されなくなりました。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。ビットマップの最適化に使用することもできません。Oracle 8.1 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。

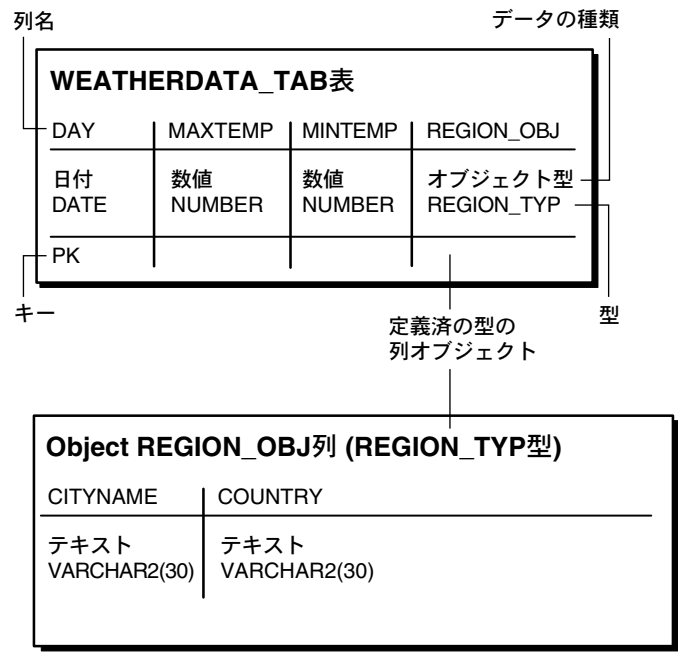
参照： ファンクション索引の使用例は、『Oracle8i 管理者ガイド』を参照してください。

例

各都市の気象データの表をメンテナンスしている気象研究所を例として考えてみます。この研究所のプロジェクトの中には、1 年を通して毎日の気温の変動を追跡するものがあります。別のプロジェクトでは、各都市の赤道からの距離の関数として気温変動を追跡します。この研究所では、計算の必要な複雑な関数に対して索引を作成することにより、問合せの実行を最適化できます。作成可能な索引およびその索引を使用できる問合せの例を、次の項で説明します。

Weatherdata_tab 表には、毎日の最低温度 (Mintemp)、毎日の最高温度 (Maxtemp)、温度記録作成日 (Day) および地域 (Region_Obj) を示す列が含まれています。Region_Obj は、国 (Country) および都市 (Cityname) の列を含むオブジェクト列です。[図 5-2](#) に、Weatherdata_tab スキーマを示します。

図 5-2 WEATHERDATA_TAB のスキーマの設計



表に含まれている都市間の気温の差を計算する索引を作成します。delta_index 索引を使用できる問合せによって、気温差が 20 度未満の表の内容が戻されます。

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 FUNCTION distance_from_equator(input
NUMBER) RETURN NUMBER DETERMINISTIC IS
    distance NUMBER;
BEGIN
    distance := 100000;
    RETURN (distance);
END;
```

```
CREATE INDEX Delta_index
ON Weatherdata_tab (Maxtemp - Mintemp);
```

```
SELECT *
FROM Weatherdata_tab
WHERE (Maxtemp - Mintemp) < '20';
```

オブジェクト・メソッド `distance_from_equator` をコールする索引を作成し、表の各都市について赤道からの距離を計算します。このメソッドは、オブジェクト列 `Region_Obj` に対して適用されます。`distance_index` 索引を使用できる問合せであれば、赤道からの距離が 1000 マイルより大きい都市の名前を戻します。

```
CREATE INDEX Distance_index
ON Weatherdata_tab (Distance_from_equator (Reg_obj));
```

```
SELECT *
FROM Weatherdata_tab
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

都市名別に気温をソートするドイツ語ユーザー用の問合せを満たす索引を作成します。`City_index` 索引を使用できる問合せであれば、表の内容を都市名別に並べて戻します。都市名に対するドイツ語のソート順序が使用されます。`SELECT` 文には、`WHERE` 句は必要ありません。それは、ドイツ語のセッションでは、`NLS_SORT` が `German` に設定され、`NLS_COMP` が `ANSI` に設定されているためです。

```
CREATE INDEX City_index
ON Weatherdata_tab (NLSSORT(Cityname, 'NLS_SORT=German'));
```

```
SELECT *
FROM Weatherdata_tab WHERE Cityname IS NOT NULL
ORDER BY Cityname;
```

最低気温と最高気温の差、および最高気温に対して索引を作成します。気温差の結果が降順でソートされます。`compare_index` 索引を使用できる問合せであれば、表の内容のうち、気温差が 20 未満で、最高温度が 75 より大きい条件を満たすものを戻します。

```
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 1:

次のコマンドは、大文字 / 小文字を区別しない検索が効果的に行えるように、EMP_TAB 表にファンクション索引 IDX を作成します。

```
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 2:

次のコマンドは、Fbi_tab 表に対してファンクション索引 IDX を作成します。A、B および C は列を表します。

```
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 の接頭辞であることに注意）または索引の高速フル・テーブル・スキャン（索引で高い並列度を指定してある場合はこのほう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 3:

次の例は、ファンクション索引を使用して NLS ソート索引をサポートする方法です。文字列が指定されている場合、NLSSORT 関数はソート・キーを戻します。次の CREATE INDEX 文は、NLS_TAB 表に対する NLS_SORT ソートを GERMAN の照合順序で作成します。

```
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。

```
SELECT *
```

```
FROM Nls_tab WHERE Name IS NOT NULL  
ORDER BY Name;
```

ファンクション索引の要件および制限

ファンクション索引には、次の要件および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。
- 式で使用される PL/SQL ファンクション（トップレベルのファンクションおよびパッケージレベルのファンクション）は、DETERMINISTIC として宣言する必要があります。サブプログラムが DETERMINISTIC として修飾されているかどうかを調べるエラー・チェックはありません。サブプログラムが 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 PACKAGE BODY 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は、LOB 列、NESTED TABLE または VARRAY には作成できません。
- ファンクション索引に使用される式で参照できるのは、表の中の連続した列のみです。このため、このような式には集計関数を含めることはできません。
- 初期化パラメータ COMPATIBLE を 8.1.0.0.0 以上に設定し、QUERY_REWRITE_ENABLED=TRUE および QUERY_REWRITE_INTEGRITY=TRUSTED に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。フル・テーブル・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。

- PL/SQL ファンクションから VARCHAR2 データ型または RAW データ型を戻すファンクション索引は、長さに制限があるため使用できません。可能な回避策として、関数の出力の長さを制限するサブストリング関数を使用します。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。

```
CREATE OR REPLACE FUNCTION x(input IN VARCHAR2)
RETURN VARCHAR2 AS
output VARCHAR2(12);
BEGIN
    output :=input;
    RETURN (output);
END;

SELECT SUBSTR(x('hello'),1,100) FROM DUAL;
```

SUBSTR (F(X), 1, 100)

F(X) は PL/SQL ファンクションを表します。索引を作成するとき、および問合せでファンクションを参照するときは、ファンクションに SUBSTR コマンドを使用する必要があります。

クラスタ、クラスタ化表およびクラスタ索引の管理

クラスタは、異なる表の関連した行を、同じデータ・ブロックに格納するため、クラスタを適切に使用すると、次の2つの点で有利になります。

- ディスク I/O は少なくなり、クラスタ化表の結合でのアクセス時間は短くなります。
- クラスタでは、クラスタ・キー値（関連した値）は、値を含む異なる表の列の行にかかわらず、1 回格納されるのみです。そのため、クラスタに関連した表データを格納するために必要となる記憶域は、非クラスタ化表の形式で必要となる記憶域より少なくなる可能性があります。

クラスタを作成するためのガイドライン

クラスタを作成するためのガイドラインは、次のとおりです。

参照： パフォーマンスの特性については、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

クラスタ化に適した表を選択する 問合せが主体で（挿入または更新が頻繁に行われず）、その問合せが頻繁にクラスタ内の複数の表のデータを結合したり、単一の表から関連したデータを取り出したりする 1 つ以上の表を格納するために、クラスタを使用します。

クラスタ・キーに適切な列を選択する クラスタ・キー列は慎重に選択してください。表を結合する問合せで複数の列が使用される場合、クラスタ・キーをコンポジット・キーにしてください。一般に、適切な索引を作成する列特性が、そのままクラスタ索引にもあてはまります。

参照： これらのガイドラインの詳細は、5-2 ページの「[正しい表および列に索引を付ける](#)」を参照してください。

適切なクラスタ・キーは、各キー値に対応している行のグループのみでほぼ 1 つのデータ・ブロックがいっぱいになるような、十分な一意の値を持っています。クラスタ・キーの値あたりの行が少なすぎる場合、領域を浪費し、パフォーマンスの向上はわずかになります。キーの値が特殊で、わずかな行のみが共通の値を共有するようなクラスタ・キーは、クラスタ作成時に小さい SIZE を指定しない限り、ブロック内の領域を浪費する可能性があります。

クラスタ・キーの値あたりの行が多すぎると、そのキーに対応する行を見つけるために余分な検索が発生する可能性があります。種類が少ない値（たとえば、MALE および FEMALE）に対するクラスタ・キーは、余分な検索が発生し、クラスタ化しない場合よりもパフォーマンスが低下する可能性があります。

クラスタ索引は、一意にできません。また、LONG として定義した列を含むこともできません。

パフォーマンスに関する考慮点

個々に索引を付けた表を別々に格納することに比べて、クラスタが DML 文（INSERT、UPDATE および DELETE）のパフォーマンスを低下させる可能性があることにも注意してください。このような短所は、表のスキャンに使用する領域の使用量およびブロック数に関係があります。複数の表が個々のブロックを共有するため、クラスタ化表の格納に使用するブロックの数は、その同じ表をクラスタ化しないで格納した場合よりも多くなります。このようなトレードオフを念頭に置いて、クラスタを使用するかどうか判断する必要があります。

非クラスタ化形式よりもクラスタ化した形式で格納する方が適しているデータを見極めるには、参照整合性制約を介して対応付けられている表、および 2 つ以上の表のデータを結合する SELECT 文を使用して頻繁にアクセスされる表を探してください。表データの結合に使用される列で表をクラスタ化すると、問合せの処理のためにアクセスする必要があるデータ・ブロックの数が少なくなります。クラスタ・キーの結合に必要なすべての行は同じブロックに入っています。そのため、結合のための問合せパフォーマンスが向上します。

同様に、個々の表をクラスタ化すると有効な場合もあります。たとえば、EMP_TAB 表は、同一部門の従業員の行をクラスタ化するために、DEPTNO 列でクラスタ化することもできます。これは、アプリケーションが一般に部門ごとに行を処理する場合に有効です。

索引と同様、クラスタはアプリケーション設計に影響しません。クラスタの存在は、ユーザーおよびアプリケーションに対して透過的です。クラスタ化表に格納されたデータは、非クラスタ化表に格納されたデータと同じように、SQL を介してアクセスされます。

クラスタ、クラスタ化表およびクラスタ索引の作成

問合せで頻繁に結合される 1 つまたは複数の表を格納する場合には、クラスタを使用します。一方、頻繁に個別アクセスを行う表は、クラスタ化しないでください。

いったんクラスタを作成すると、そのクラスタ内に表を作成できます。ただし、クラスタ化表に行を挿入する前に、クラスタ索引を作成する必要があります。クラスタを使用しても、クラスタ化表に対する追加の索引作成には影響しません。通常どおり作成または削除を行うことができます。

クラスタを作成するには、SQL コマンド CREATE CLUSTER を使用します。次の文は、EMP_TAB 表および DEPT_TAB 表を格納するクラスタ EMP_DEPT を作成し、その際、DEPTNO 列によってクラスタ化します。

```
CREATE CLUSTER Emp_dept (Deptno NUMBER(3))  
PCTUSED 80
```

```
PCTFREE 5;
```

CLUSTER オプション付きの SQL コマンド CREATE TABLE を使用して、クラスタ内に表を作成します。たとえば、次の文を使用して、EMP_DEPT クラスタ内に EMP_TAB 表および DEPT_TAB 表を作成できます。

```
CREATE TABLE Dept_tab (  
    Deptno NUMBER(3) PRIMARY KEY,  
    . . . )  
    CLUSTER Emp_dept (Deptno);  
  
CREATE TABLE Emp_tab (  
    Empno NUMBER(5) PRIMARY KEY,  
    Ename VARCHAR2(15) NOT NULL,  
    . . .  
    Deptno NUMBER(3) REFERENCES Dept_tab)  
    CLUSTER Emp_dept (Deptno);
```

クラスタ内に作成される表は、CREATE TABLE 文で指定したスキーマに含まれます。クラスタ化表は、そのクラスタを含む同じスキーマに存在しない場合もあります。

クラスタ化表に行を挿入する前に、クラスタ索引を作成する必要があります。たとえば、次の文は、EMP_DEPT クラスタに対するクラスタ索引を作成します。

```
CREATE INDEX Emp_dept_index  
    ON CLUSTER Emp_dept  
    INITRANS 2  
    MAXTRANS 5  
    PCTFREE 5;
```

注意： クラスタ索引は、一意にできません。さらに、Oracle では、一意キー制約または主キー制約があってもクラスタ・キーの列について一意性の施行は保証されません。

クラスタ・キーは、クラスタ内の複数の表の関連を確立します。

クラスタ、クラスタ化表およびクラスタ索引の作成に必要な権限

自スキーマにクラスタを作成するには、CREATE CLUSTER システム権限およびそのクラスタ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クラスタを作成するには、CREATE ANY CLUSTER システム権限が必要で、しかも所有者にそのクラスタ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。

クラスタ内に表を作成するには、CREATE TABLE システム権限または CREATE ANY TABLE システム権限が必要です。クラスタ内に表を作成するには、表領域に対する割当て制限または UNLIMITED TABLESPACE システム権限は必要ありません。

クラスタ索引を作成するには、自スキーマにクラスタが含まれるか、または次の権限が必要です。

- CREATE ANY INDEX システム権限、またはクラスタを所有している場合は CREATE INDEX システム権限
- クラスタ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

クラスタに対する記憶域の手動割当て

Oracle は、クラスタのデータ・セグメントに対して、追加のエクステントを必要に応じて動的に割り当てます。状況によっては、クラスタに対する追加のエクステントを明示的に割り当てるほうがよい場合もあります。たとえば Oracle Parallel Server を使用しているときに、クラスタのエクステントを特定のインスタンスに対して明示的に割り当てることができません。

ALLOCATE EXTENT オプションを指定した SQL コマンド ALTER CLUSTER を使用して、クラスタに新しいエクステントを割り当てることができます。

参照： Oracle8i Parallel Server のマニュアル・セット (『Oracle8i Parallel Server 概要』、『Oracle8i Parallel Server セットアップおよび構成ガイド』、『Oracle8i Parallel Server 管理、配置およびパフォーマンス』) を参照してください。

クラスタ、クラスタ化表およびクラスタ索引の削除

クラスタ内に現在ある表が不要になった場合は、クラスタを削除します。クラスタを削除すると、そのクラスタ内にある表および対応するクラスタ索引も削除されます。クラスタ・データ・セグメントとクラスタ索引の索引セグメントの両方に属するすべてのエクステントは、それらを含む表領域に戻され、表領域内の他のセグメントに対して使用可能になります。

クラスタ化表は、その表のクラスタ、他のクラスタ化表またはクラスタ索引に影響することなく、個別に削除できます。非クラスタ化表を削除する場合と同様に、SQL コマンド DROP TABLE を使用して、クラスタ化表を削除します。

表を個別に削除する際の詳細は、2-10 ページの「表の削除」を参照してください。

注意： 単一のクラスタ化表をクラスタから削除するときには、その表の各行をクラスタから削除する必要があります。効率を最大にするために、すべての表を含めてクラスタ全体を削除する場合は、INCLUDING TABLES オプション付きの DROP CLUSTER コマンドを使用してください。

DROP TABLE コマンドは、クラスタの残りの部分はそのまま残る場合にのみ、個別の表をクラスタから削除するために使用してください。

クラスタ索引は、クラスタまたはそのクラスタ化表に影響することなく、削除できます。ただし、クラスタ索引がないと、クラスタ化表を使用できません。断片化したクラスタ索引を再構築するための手順の一部として、クラスタ索引を削除することもあります。

参照： 5-5 ページの「[索引の削除](#)」を参照してください。

表を含まないクラスタおよびそのクラスタ索引（ある場合）を削除するには、SQL コマンド DROP CLUSTER を使用します。たとえば、次の文は空のクラスタ EMP_DEPT を削除します。

```
DROP CLUSTER Emp_dept;
```

クラスタが 1 つ以上のクラスタ化表を含んでいて、その表も削除する場合は、DROP CLUSTER コマンドに INCLUDING TABLES オプションを追加します。次に例を示します。

```
DROP CLUSTER Emp_dept INCLUDING TABLES;
```

INCLUDING TABLES オプションを指定しない場合、クラスタに表が含まれていると、エラーが戻されます。

クラスタ内の 1 つ以上の表が、そのクラスタ外に存在する表の外部キー制約によって参照される主キーまたは一意キーを含む場合、その依存関係を持つ外部キー制約も削除しない限り、そのクラスタを削除できません。次のように、DROP CLUSTER コマンドに CASCADE CONSTRAINTS オプションを使用してください。

```
DROP CLUSTER Emp_dept INCLUDING TABLES CASCADE CONSTRAINTS;
```

このオプションを使用しない場合、エラーが戻されることがあります。

クラスタの削除に必要な権限

クラスタを削除するには、自スキーマにそのクラスタが含まれているか、または DROP ANY CLUSTER システム権限が必要です。クラスタ化表がそのクラスタの所有者によって所有されていなくても、その表を含むクラスタを削除するための特別が権限は必要ありません。

ハッシュ・クラスタおよびクラスタ化表の管理

次の項では、SQL コマンドを使用して、ハッシュ・クラスタおよびクラスタ化表を作成、変更および削除する方法を説明します。

ハッシュ・クラスタおよびクラスタ化表の作成

ハッシュ・クラスタは、等式による問合せの対象となることが多い静的なクラスタ化表のグループまたは個別の表を格納するために使用します。ハッシュ・クラスタを作成してから、表を作成します。ハッシュ・クラスタを作成するには、SQL コマンド CREATE CLUSTER を使用します。次の文は、TRIAL_TAB 表を格納するために使用するクラスタ TRIAL_CLUSTER を作成し、TRIALNO 列でクラスタ化します。

注意： 次のような設定を使用（システム表領域ヘデータ・ファイルを追加）しないと機能しない例もあります。

```
ALTER TABLESPACE SYSTEM ADD DATAFILE 'disk1:moredata1' SIZE 50K
AUTOEXTEND ON;
```

```
CREATE CLUSTER Trial_cluster (
    Trialno NUMBER(5,0)
    PCTUSED 80
    PCTFREE 5
    SIZE    2K
    HASH IS Trialno HASHKEYS 100000;
```

```
CREATE TABLE Trial_tab (
    Trialno NUMBER(5) PRIMARY KEY,
    ...)
CLUSTER Trial_cluster (Trialno);
```

ハッシュ・クラスタ内のスペース使用の制御

ハッシュ・クラスタを作成するときには、クラスタに対してパフォーマンスおよび領域使用が適切になるように、クラスタ・キーを正しく選択し、HASH IS、SIZE および HASHKEYS の各パラメータを設定することが重要です。次の項では、これらのパラメータを設定する例およびガイドラインを示します。

キーの選択

正しいクラスタ・キーの選択は、クラスタ化表に対して最もよく発行される問合せのタイプに依存しています。たとえば、ハッシュ・クラスタ内の EMP_TAB 表について検討します。問合せが従業員番号で行を選択することが多い場合、EMPNO 列をクラスタ・キーにしてください。問合せが部門番号で行を選択することが多い場合には、DEPTNO 列をクラスタ・キーにしてください。単一の表を含むハッシュ・クラスタの場合、通常、その表の主キー全体をクラスタ・キーにします。コンポジット・キーを持つハッシュ・クラスタは、Oracle の内部ハッシュ関数を使用する必要があります。

HASH IS の設定

クラスタ・キーが NUMBER データ型の単一の列であり、しかも均等分散した整数を含んでいる場合にのみ、HASH IS パラメータを指定します。この条件が適用される場合、それぞれ一意のクラスタ・キー値が、(競合なしで) 一意のハッシュ値にハッシュされるように、クラスタ内に行を分散させることができます。条件が適用されない場合、内部ハッシュ関数を使用する必要があります。

ハッシュ・クラスタの削除

次のように SQL コマンド DROP CLUSTER を使用して、ハッシュ・クラスタを削除します。

```
DROP CLUSTER Emp_dept;
```

SQL コマンド DROP TABLE を使用して、ハッシュ・クラスタ内の表を削除します。ハッシュ・クラスタおよびハッシュ・クラスタ内の表の削除に関する問題は、索引クラスタの場合と同じです。

参照： 5-17 ページの「[クラスタ、クラスタ化表およびクラスタ索引の削除](#)」を参照してください。

ハッシュの使用時期

ハッシュ・クラスタに表を格納することは、索引を持つ同じ表を格納することにかわるものです。ハッシュは、次のような状況で有効です。

- ほとんどの問合せが、クラスタ・キーに対する等式による問合せである場合。次に例を示します。

```
SELECT . . . WHERE Cluster_key = . . . ;
```

このような場合、等号条件の成り立つクラスタ・キーがハッシュされ、対応するハッシュ・キーは、通常単一の読み込みによって検索されます。索引付きの表では、キー値を最初に索引で検索する必要があり（通常複数の読み込み）、次に行が表から読み込まれます（別の読み込み）。

- ハッシュ・クラスタ内の表のサイズがほぼ不変であり、行の数およびそのクラスタ内の表に対して必要となる領域の容量を決定できる場合。ハッシュ・クラスタ内の表が、そのクラスタに対する初期割当てよりも多くの領域を必要とする場合は、オーバーフロー・ブロックが必要となるために、パフォーマンスが大幅に低下する可能性があります。
- `HASH IS col`、`HASHKEYS n` および `SIZE m` 句を指定したハッシュ・クラスタは、各項目が m バイトのデータから構成される n 項目（行）の配列（表）の理想的な表現である場合。次に例を示します。

```
ARRAY X[100] OF NUMBER(8)
```

これは次のように表すこともできます。

```
CREATE CLUSTER C(Subscript INTEGER)
  HASH IS Subscript HASHKEYS 100 SIZE 100;
```

```
CREATE TABLE X(Subscript NUMBER(2), Value NUMBER(8))
  CLUSTER C(Subscript);
```

一方、ハッシュは次のような状況では有効ではありません。

- 表に対する問合せのほとんどが、クラスタ・キー値の範囲外の行を検索する場合。たとえば、フル・テーブル・スキャン、または次のような問合せです。

```
SELECT . . . WHERE Cluster_key < . . . ;
```

ハッシュ関数は、特定のハッシュ・キーの位置を決めるために使用することはできません。そのかわりに、問合せの行を取り出すためにフル・テーブル・スキャンに相当する処理が行われます。索引では、キー値はその索引内で順序付けられており、問合せの `WHERE` 句を満足するクラスタ・キー値を、比較的少ない I/O で見つけることができます。

- 表が固定的でなく継続して成長する場合。表が無制限に成長する場合、表（そのクラスタ）の存続期間にわたって必要となる領域を事前に定義できません。

- アプリケーションが表に対して頻繁にフル・テーブル・スキャンを行い、その表内のデータが散在している場合。この条件でフル・テーブル・スキャンを行うとハッシュによって検索時間が長くなります。
- ハッシュ・クラスタによって最終的に必要となる領域を事前に割り当てる余裕がない場合。

多くの場合、ハッシュを使用するか索引を使用するかを（前述の情報に基づいて）決める必要があります。索引を使用する場合、表を個別に格納することが最適か、クラスタの一部として格納することが最適かを検討してください。

参照： 5-14 ページの「[クラスタを作成するためのガイドライン](#)」を参照してください。

ハッシュの使用を決めた場合でも、（クラスタ・キーも含めて）表はどの列に対しても個別に索引を持つことができます。

参照： ハッシュ・クラスタのパフォーマンス特性のガイドラインの詳細は、『*Oracle8i パフォーマンスのための設計およびチューニング*』を参照してください。

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。

- [索引構成表の概要](#)
- [索引構成表の機能](#)
- [索引構成表の使用時期](#)
- [例](#)

参照： CREATE TABLE 文の ORGANIZATION INDEX 句の構文については、『Oracle8i 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。いったん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。行が存在する限り、その物理 ROWID は変更されません。

通常の表の列に索引を付けるとき、新しく作成された索引は、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、一方では、1 つ以上の列に索引が付いた通常の表のようなものです。ただし、索引構成表は、データを決まった行に保持するのではなく、表の主キーに対して作成されている B*-tree 索引のリーフ内にソートされた順序でデータを保持するという点が特殊です。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索引リーフが分割され、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*-tree 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*-tree 索引の詳細は、『Oracle8i 概要』を参照してください。

索引構成表の利点

索引構成表では、主キーを基にした B*-tree 索引に行が格納されるため、通常の表と比べて次のような利点があります。

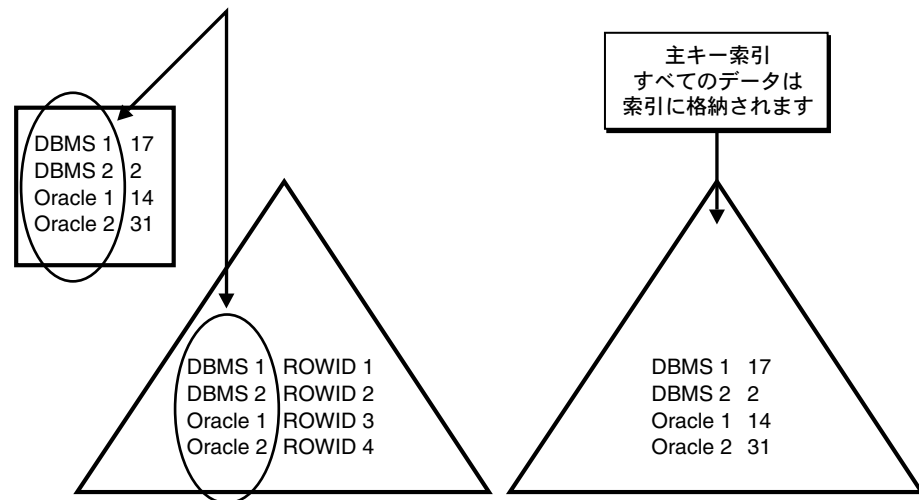
主キーの完全一致または範囲検索（あるいはその両方）を伴う問合せでの表データの高速アクセス 検索でキー値が見つかり、残りのデータはその位置に存在します。索引構造になった通常の表の場合は、ROWID をたどって表データに戻る必要がありますが、索引構成表の場合は、この必要がありません。このように、索引構成表では I/O（表の読み込み）が 1 回少なくなるため効率が向上します。

1 日 24 時間 1 週 7 日体制（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利点があります。

- 索引構成表または索引構成表のパーティションは、索引を作成し直さなくても、（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。さらに、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なくなります。

記憶域必要量の削減 これは、キー列が表と索引とで重複しないこと、および ROWID に必要な追加記憶域が不要であることによります。

図 6-1 索引付きの通常の表と索引構成表の比較



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。索引構成表に使用できる機能のいくつかを次に示します。

通常の表と同じ表変更オプションのサポート 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*-tree 索引内では、その性質上、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 を基に作成することはできません。かわりに、索引構成表の 2 次索引は、論理 ROWID と呼ばれるものを基にしています。論理 ROWID 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、および推測が行われた時点での行のブロック位置を示す推測値が含まれます。推測値によって、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。

- ユーザーは、索引構成表から ROWID を選択し、
WHERE ROWID = <value predicate> を使用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。
- 行の論理 ROWID は、主キー値が変更されない限り変更されません。ただし、どのような更新があっても不変の物理 ROWID とは違い、論理 ROWID は新しい行が挿入されたときに移動する可能性があります。

Oracle8i のリリース 8.1 では、論理 ROWID と物理 ROWID の両方をサポートする汎用 ROWID という単一のデータ型が追加されています。

現在、ROWID を使用中のアプリケーションで索引構成表を使用する場合、汎用 ROWID を使用するように変更することが必要な場合がありますが、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 および物理 ROWID にアクセスできます。

参照： 6-11 ページの「汎用 ROWID データ型の宣言」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り 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、主キー索引のリーフ・ブロックに直接アクセスできる推測値も含まれます。推測値が正しい場合、2 次キーが見つかった後は、2 次索引スキャンにより発生する追加 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。ただし、索引構成表に対するビットマップ 2 次索引は現在はサポートされていません。

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限り）、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る表領域に作成されます。
- インライン格納および行外格納 — デフォルトでは、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（つまり、デフォルトの格納属性は 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。ただし、オーバーフロー・セグメント付きの索引構成表に作成された LOB は、通常の表の場合と同じ特性になります。

その他の LOB 機能（BFILE、テンポラリ 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。パーティション化された索引構成表では LOB は現在サポートされていません。

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。ただし、2 次索引のみの索引スキャンを伴う問合せは、まだパラレルには実行できません。

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、NESTED TABLE、REF 列など）がサポートされます。ただし、オブジェクト表（TABLE OF <object type>）を索引構成表として作成できません。

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・パスの平行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平行ロードを実行してから、平行の 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いないに関わらず、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いないに関わらず、どちらもレプリケートできます。

Tools 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*-tree のリーフ・ブロック内のキーのみが圧縮されます。B*-tree 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。¹

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 アプリケーションでの使用に適しています。

通常の表と比べた索引構成表の大きな利点は、索引構成表の 2 次索引が備えた論理性にあります。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。これとは対照的に通常の表の場合は、このような操作を実行するとグローバル索引が使用できなくなり、索引全体を作成し直すことが必要になって非常にコストが上がります。

同様に、ALTER TABLE MOVE 操作の後、索引構成表のローカル索引も使用できます。一方、通常の表の場合は MOVE 操作を実行すると 2 次ローカル索引は使用できなくなります。

前述のパーティション・メンテナンス操作を実行すると、論理 ROWID の推定値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。

さらに、ALTER TABLE MOVE 操作はオンラインで実行できます。この機能のため、索引構成表は 1 日 24 時間 1 週 7 日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。

時系列アプリケーションを開発する場合 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。時系列とは、通常、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いいます。データは通常、株記号やタイム・スタンプなどの項目識別子を介してアクセスされます。Oracle8 Time Series Data Cartridge 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。

¹ Oracle アドバンスド・キューイングを使用している場合は、すでに索引構成表に精通している可能性があります。Oracle アドバンスド・キューイングは、Oracle8i Server の統合された部分の 1 つとしてメッセージ・キューイングを提供し、索引構成表を使用して複数コンシューマ・キューのメタデータ情報を保存します。この場合、索引構成表は索引として機能し、キュー識別子に対する主キーの B*-tree 索引の一部としてキューのメタデータを格納します。DML 操作でこの索引を更新する必要がありますが、これは基礎となる索引構成表を更新することによって効率的に行われます。

NESTED TABLE を使用する場合 NESTED TABLE の列の場合、NESTED TABLE 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。NESTED TABLE 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。NESTED TABLE の列の記憶域として通常の表を使用した場合は、通常、NESTED TABLE のクラスタ化が解除されます。これと比べて、索引構成表を使用した場合は、NESTED TABLE の行を NESTED_TABLE_ID 列を基にクラスタ化できます。Oracle8i リリース 8.1 では、次に示すように、NESTED TABLE の格納先を索引構成表に指定できます。

```
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;
```

拡張可能な索引を使用する場合 Oracle8i リリース 8.1 では、データベースに対して新しいアクセス方法を追加する拡張可能な索引フレームワークが新しく導入されています。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何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。Oracle8 の Spatial Cartridge および Text Database Cartridge では、索引データの格納に索引構成表を使用するドメイン固有の索引スキームが実装されています。

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 TABLE 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);
```

この例は、索引構成表を作成し使用するための基本的なタスク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逆索引¹を使用しています。これによって、ユーザーは Web 上で特定の単語または語句を問い合わせることができます。次に、問い合わせた単語および語句を含むドキュメントへのハイパーテキスト・リンクのリストがユーザーに戻されます。リンクには関連度の高い順に順位が付けられます。

この例には、次のタスクが示されています。

¹ 逆索引は、各ドキュメントを個別のワードまたはトークンに分割します。逆索引では、各ワードに対して、そのワードが発生するドキュメントのリストを作成し、そのリストをデータベースに格納します。アプリケーションでは、逆索引をスキャンして対象のトークンを探すことによって、内容ベースの検索を実行します。開発の観点からすると、通常、逆索引にはドキュメント内の異なった各ワードに対して、書式 <token, document_id, occurrence_data> のエントリが含まれます。

- 通常の表から索引構成表への既存データの移動
- 索引構成表の作成
- 汎用 ROWID データ型の宣言
- 索引構成表に対する 2 次索引の作成
- 索引構成表の操作
- オーバーフロー・データ・セグメントの指定
- 索引の行ヘッドに含める最後の非キー列の決定
- オーバーフロー・セグメントへの列の格納
- 物理属性および格納属性の変更
- 索引構成表のパーティション化
- 索引構成表の再作成

通常の表から索引構成表への既存データの移動

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

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;

```

汎用 ROWID データ型の宣言

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

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);
```

次の例に示されているとおり、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

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。これらすべての操作は、結果的に主キーの B*-tree 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

すべての非キー列を主キーの B*-tree 索引構造に格納することが常に適切であるとは限りません。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*-tree 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*-tree のリーフ・ブロックでは索引行を最低 2 行保持する必要があるため、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。

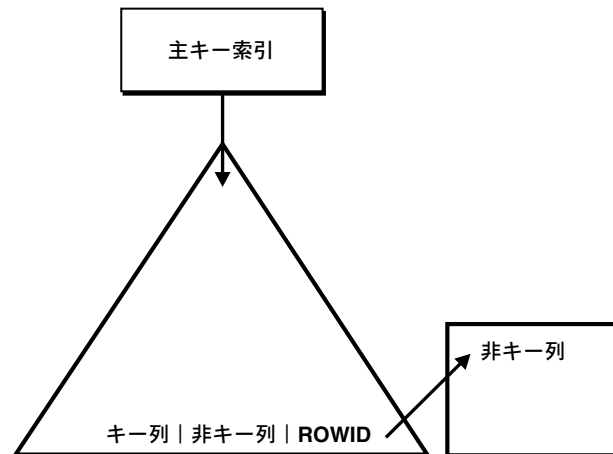
このような問題を解決するために、オーバーフロー・データ・セグメントを索引構成表に対応付けることができます。次の例では、docindex 表に token_offsets という追加列が必要です。この例は、索引構成表を作成する方法、および OVERFLOW オプションを使用してオーバーフロー・データ・セグメントを作成する方法です。

```
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

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;
```

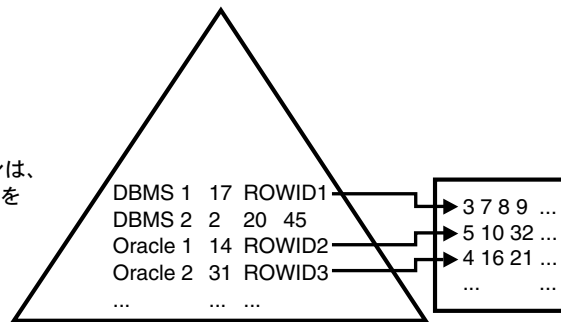
このように、索引とデータ・セグメントとの間で行を垂直にパーティション化することによって、索引内での行のクラスタ密度が高くなります。これによって、索引内に格納されている列に対する問合せのパフォーマンスが向上します。たとえば、`token_offsets` 列があまり頻繁にアクセスされない場合は、この列を索引から外すことで、主キーの B*-tree 構造内で索引行のクラスタ化が進みます（図 6-3 を参照）。これによって、問合せの全体的なパフォーマンスも向上します。ただし、オーバーフロー・データ・セグメントに格納されている列には追加のブロック・アクセスが発生し、これでパフォーマンスが低下することもあります。

オーバーフロー・セグメントへの列の格納

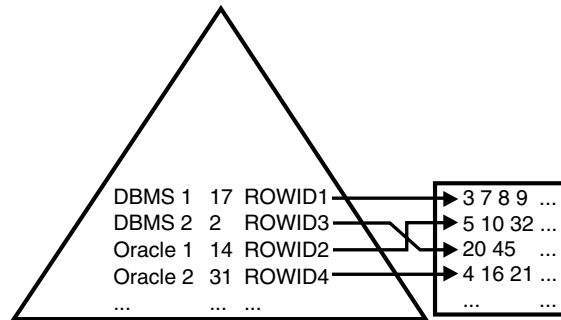
`INCLUDING` オプションによって、指定された列より後のすべての列がオーバーフロー・セグメントに格納されます。指定された `INCLUDING` 列に対応する索引行サイズが、指定されたしきい値を超える場合、含まれる最後の非キー列は `PCTTHRESHOLD` オプションに従って決定されます。

図 6-3 PCTTHRESHOLD 対 INCLUDING 列の使用

PCTTHRESHOLD オプションは、
token_offsets のいくつかの行を
強制的にオーバーフロー・
セグメントに入れます。



INCLUDING オプションは
token_offsets のすべての行を
強制的にオーバーフロー・
セグメントに入れます。



物理属性および格納属性の変更

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

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、ANALYZE コマンドを使用して 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およびオーバーフロー・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。ただし、連鎖行を識別するには、主キー列を含む少し異なる CHAINED_ROWS 表を作成する必要があります。Oracle8i リリース 8.1 では論理 ROWID サポートが追加されているため、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、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

索引構成表は、列値の範囲ごとにパーティション化できます。ただし、このようにパーティション化された索引構成表を作成するには、一連のパーティション化列が主キー列のサブセットになっている必要があります。この制約を加えることによって、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パーティション化されているのと同じです。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。

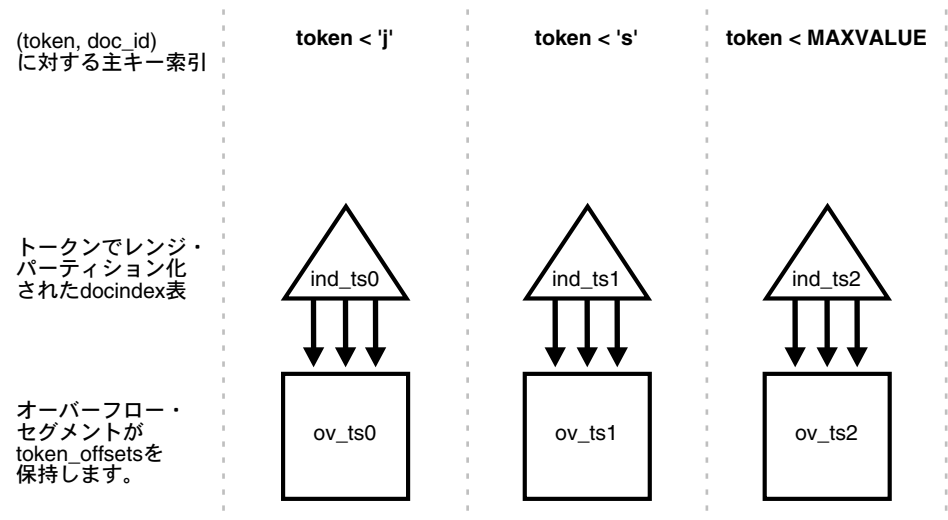
- 通常のパーティション表の場合と同じように、表レベルの物理属性にデフォルト値を指定できます。これらの値は、パーティションごとに（索引セグメントおよびオーバーフロー・データ・セグメントの両方で）オーバーライドできます。
- 索引セグメントの表領域がパーティション用に指定されていない場合は、表レベルのデフォルトに設定されます。表レベルのデフォルトが指定されていない場合は、そのユーザーのデフォルト表領域が使用されます。
- PCTTHRESHOLD 列および INCLUDING 列のデフォルト値は、表レベルでのみ指定できます。
- OVERFLOW キーワードの前に指定されるすべての属性は、主キー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が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 列によって、token_offsets 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のサポートは、通常の表と非常に似ています。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、MERGE を除くすべての ALTER TABLE 操作を使用できます。ただし、次のように、通常の表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、すべての索引（ローカル、グローバル、非パーティション化）は USABLE のまま残ります。ただし、論理 ROWID に格納されている推定値は無効になります。
- SPLIT パーティション操作では、すべての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値が再作成されることです。

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じように機能します。

キーの圧縮

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。さらに、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。接頭辞の長さの値の有効範囲は、[1, 主キー列の数 - 1] です。

```
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 (トークン列) は、主キー (token, doc_id) のオカレンスに圧縮されます。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 という繰返しオカレンスが圧縮されます。

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。圧縮オプションは、索引構成表の作成時、または ALTER TABLE MOVE オプションを使用した索引構成表の移動の一部として指定できます。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 キーの圧縮は、パーティション化された索引構成表でもサポートされます。COMPRESS 句は、表レベルのデフォルトの一部として指定する必要があります。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。

ただし、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

新しい SQL コマンド ALTER TABLE MOVE を使用すると、索引構成表を移動つまり再作成できます。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*-tree 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*-tree 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
- PCTTHRESHOLD 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*-tree 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 1 日 24 時間 1 週 7 日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;
```

MOVE オプションは、通常の表でも使用できます。ただし、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。

SQL 文の処理

この章では、Oracle における構造化問合せ言語（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。

- SQL 文の実行
- トランザクションの制御
- 読取り専用トランザクションでのリPEATブル・リードの保証
- カーソルの使用
- 明示的なデータのロック
- 行ロックの明示的な取得
- Oracle に対する表ロック制御の許可
- ユーザー・ロック
- シリアライズ可能トランザクションを使用した同時実行性の制御
- 自律型トランザクション

Oracle Tools および Applications の中には、SQL の使用を簡素化またはマスクしているものがありますが、すべてのデータベース操作は SQL を使用して行われます。他のデータ・アクセス方法の場合、どれを使用しても、Oracle に組み込まれているセキュリティをバイパスすることがあり、データのセキュリティおよび整合性が損なわれる可能性があります。

SQL 文の実行

表 7-1 に、SQL 文の処理および実行に通常使用されるステップの概要を示します。これらのステップ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 ステップが FETCH ステップの直前に実行される場合もあります。

Oracle Tools の多くでは、いくつかのステップ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、Oracle Applications を作成するときに、この情報が役立つこともあります。

参照： 各種 SQL 文に対する SQL 処理ステップについては、『Oracle8i 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役立つ 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 文をチェックして、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ったら、Oracle Server 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。

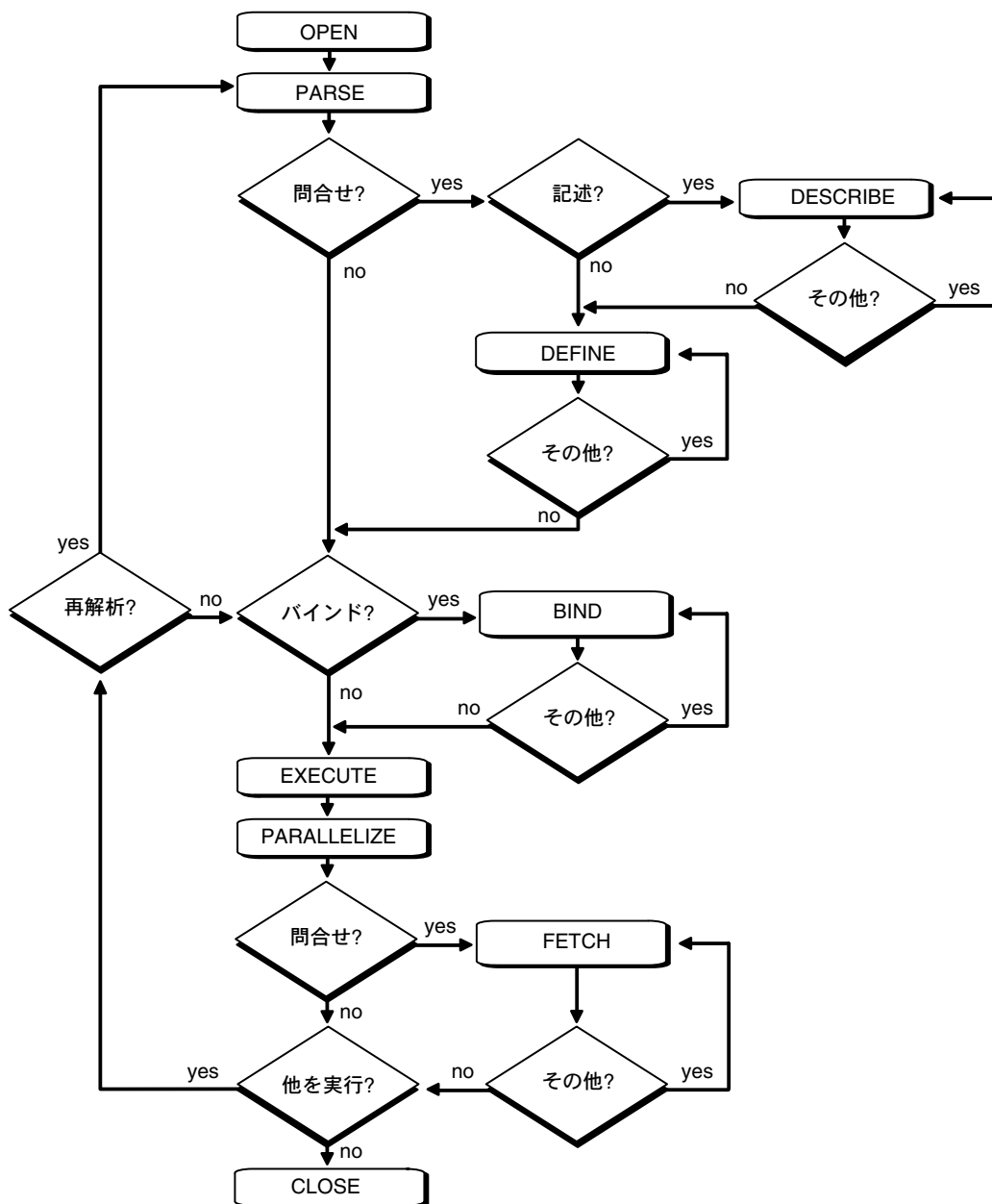
FIPS フラグ付け機能では、Enterprise Manager または SQL*Plus を使用して送信される対話型 SQL 文を介して、フラグ付けをサポートされています。また、Oracle プリコンパイラおよび SQL*Module では、埋込み SQL およびモジュール言語 SQL の FIPS フラグ付けもサポートされています。

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 文の処理ステップ



トランザクションの制御

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。

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には、次に挙げるパフォーマンス要件を考慮する必要があります。特に指定しない限り、詳細は、『Oracle8i 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ツ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 シリアライズ可能トランザクションを生成します。

参照：

- 7-25 ページの「シリアライズ可能トランザクションの相互作用」を参照してください。
- 『Oracle8i 概要』も参照してください。
- 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 は同一 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、SQL 文を最適化するコストベースのアプローチを実装するために Oracle が使用する統計を収集します。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えるようにします。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。
- 第9章にある「SQL 式からのストアド・ファンクションのコール」に説明されているように、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： 7-9 ページの「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（50文字未満）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のに役立ちます。

```
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、『Oracle8i 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、『Oracle8i 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。

セッションあたりのアクティブ・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ている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
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 は定義されたままになります。
ROLLBACK TO c;	ORA-01086 エラー。セーブポイント C は定義されません。
INSERT INTO...;	このトランザクション内の新しい 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

独自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。

読取り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取り一貫性を保証しますが、トランザクション・レベルの読取り一貫性（リピータブル・リード）は保証しません。トランザクション・レベルの読取り一貫性が必要で、トランザクションが更新を必要としない場合には、読取り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取り専用指定した後は、読取り専用トランザクションの問合せ結果に時刻に対する一貫性があることがわかっているため、どのデータベース表に対しても必要な回数だけ問合せを実行できます。

読取り専用トランザクションでは、トランザクション・レベルの読取り一貫性を提供するために、追加データ・ロックは何も取得しません。文レベルの読取り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・レベルの読取り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、読取り専用トランザクションが開始したときに判断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、読取り専用トランザクションが問い合わ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わせたり更新したりできます。

読取り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取り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。または、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、または問合せ中の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。

読取り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 文よりも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 文により、読取り専用トランザクションは終了します（DDL 文によって、読取り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）。

カーソルの使用

PL/SQL では、1 行しか戻さない問合せも含めて、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、行を別々に処理するカーソルを明示的に宣言できます。

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。つまり、カーソルは、特定のプライベート SQL 領域の名前と考えられます。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り出せます。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。カーソル変数については、『Oracle8i 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 文の解析に特に便利です。

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 つの制約を受けます。

- 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、プロセスで利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）内の初期化パラメータ OPEN_CURSORS によって設定されます。

参照： パラメータの詳細は、『Oracle8i リファレンス・マニュアル』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役に立ちます。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と、その数のカーソルを必ず同時にオープンできます。

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限り、最初から実行し直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、解析ステップ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と、記述ステップ、定義ステップ、バインド・ステップまたは実行ステップ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。

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、いったん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する。
- 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場合、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する（ただし、こ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、それら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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

明示的なデータのロック

Oracle は、データの同時実行性と整合性、および文レベル読取り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。ただし、デフォルト・ロック・メカニズムを無効にするオプションも使用できます。次のような場合は、Oracle のデフォルト・ロックを無効にした方が有利です。

- トランザクション・レベルの読取り一貫性またはリピータブル・リードをアプリケーションが要求する場合。トランザクションは、その存続時間に一貫した一連のデータを問い合わせ、データがシステムの他のどのトランザクションによっても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・レベルの読取り一貫性は、明示的なロック、読取り専用トランザクション、シリアライズ可能トランザクションまたはシステムのためのデフォルト・ロックの無効化を使用することによって実現されます。
- トランザクションがリソースに排他的にアクセスすることをアプリケーションが必要とする場合。リソースに排他的にアクセスできるトランザクションは、その文を続行するために、他のトランザクションが完了するのを待つ必要はありません。

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。

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、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・ロックを無効にします。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。
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・ロックを無効にするために使用できる各オプションを説明します。初期化パラメータ DML_LOCKS 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、『Oracle8i リファレンス・マニュアル』を参照してください。

デフォルト値として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場合には、この値を大きくすることが必要になる場合があります。

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。つまり、データ整合性が保証され、データ同時実行性が許容され、デッドロックの可能性がないか、またはデッドロックが適切に処理されるか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、基礎となるベース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。この論理は、アプリケーション内に作成できます。

注意： 表ロックを待機中の分散トランザクションは、経過時間が初期化パラメータ `DISTRIBUTED_LOCK_TIMEOUT` で設定された時間間隔に達した場合、要求したロックに対する待機をタイムアウトできます。データは変更されていないため、タイムアウトの後に処置は必要ありません。アプリケーションでは、デッドロックが検出されたときと同様の処理を続行します。分散トランザクションの詳細は、『Oracle8i 分散システム』を参照してください。

次に、`LOCK TABLE` コマンドを使用した各種の表ロックの取得時期について説明します。

ROW SHARE および ROW EXCLUSIVE

```
LOCK TABLE Emp_tab IN ROW SHARE MODE;  
LOCK TABLE Emp_tab IN ROW EXCLUSIVE MODE;
```

行共有表ロックおよび行排他表ロックは、最も高い同時実行性を提供します。行共有表ロックまたは行排他表ロックが明示的に取得されるのは、次のような場合です。

- 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。
- 後にトランザクション内で表が変更される前に、表の変更または削除を防止する必要がある場合。

SHARE

```
LOCK TABLE Emp_tab IN SHARE MODE;
```

共有表ロックはかなり制限の多いデータ・ロックです。共有表ロックは、次のような条件の場合に明示的に取得されます。

- トランザクションが表を問い合わせるだけで、トランザクションの存続期間に一貫した表データの集合を必要とする場合（ロックされている表に対して、トランザクション・レベルの読取り一貫性が必要な場合）。

- 共有表ロックを持つ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されるまで、ロックされている表を同時に更新しようとしている他のトランザクションが待機する必要がある場合。
- 他のトランザクションに対して、同じ表に対する同時共有表ロックの取得を許可し、さらに、トランザクション・レベルの読取り一貫性というオプションも許可する場合。

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には、(SELECT...FOR UPDATE 文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 の一貫したデータ・セット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り対処できます。DEPT_TAB 表が更新されることはほとんどないため、EMP_TAB 表および BUDGET_TAB 表の更新のために DEPT_TAB 表がロックされている間に DEPT_TAB 表を更新する必要があるユーザーはほとんどいません。

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 (  
    totalsal 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 Totalsal = Totalsal * 1.1  
    WHERE Deptno IN  
        (SELECT Deptno FROM Dept_tab WHERE Loc = 'DALLAS');  
  
COMMIT; /* This releases the lock */
```

SHARE ROW EXCLUSIVE

```
LOCK TABLE Emp_tab IN SHARE ROW EXCLUSIVE MODE;
```

共有行排他表ロックは、次のような場合に明示的に取得されます。

- 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・レベルの読取り一貫性、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）を意識しなくてよい場合。これによって、ロック中のトランザクション内の UPDATE 文および INSERT 文が表の更新を待機する場合もあり待機しない場合もあります（デッドロックが発生することがあります）。
- 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

```
LOCK TABLE Emp_tab IN EXCLUSIVE MODE;
```

排他表ロックは、次のような場合に明示的に取得されます。

- 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。したがって、トランザクションが排他表を保持する場合、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・レベルの読取り一貫性が保持される場合。
- 低レベルのデータ同時実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。ただし、別の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任意のオブジェクト権限（たとえば、SELECT または UPDATE）が必要です。

Oracle に対する表ロック制御の許可

Oracle に対して表ロック制御を許可すると、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も制御範囲が小さくなります。

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE または ALTER SESSION ISOLATION LEVEL SERIALIZABLE 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 のシリアライズ可能性を維持できます。この手法によって、ANSI のシリアライズ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、同時実行性が大幅に減少します。

また、表ロックは ROW_LOCKING および SERIALIZABLE でも制御されます。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 Oracle の以前のバージョン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 コマンドおよび ALTER SESSION コマンドの詳細は、『Oracle8i 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します。複数インスタンスが単一データベースをアクセスする場合、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 を 3 通りの組合せで使用できます。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デフォルト以外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	バージョン 5 以前の Oracle リリースと等価（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。SELECT... FOR UPDATE は、選択されている行を実際に更新する予定で、(UPDATE 文のように) 選択されている行に対して排他行ロックを取得するために使用します。

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 12 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、参照された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、行に対する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に、最初のフェッチの前に結果セット内の行すべてがロックされます。

つまり、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、かなりの更新アクティビティが表に対して発生する場合、かわりに排他表ロックを取得すると、多くの場合パフォーマンスが改善されます。

SELECT... FOR UPDATE によって行ロックを取得する場合、ロックの取得を待つかどうかを指定できます。NOWAIT オプションを指定すると、そのロックがすぐに使用できる場合にのみ行ロックが取得されます。すぐに使用できない場合は、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。この場合は、後でその行のロックを再試行できます。

NOWAIT を指定しないと、要求した行ロックが取得されるまで、トランザクションは処理を継続しません。行ロックに対する待ち時間が長すぎる場合には、そのロック操作を取り消して、後で再試行できます。この論理はアプリケーション内に作成できます。

7-12 ページの「[ロック方法の選択](#)」に説明されているように、行ロックを待機中の分散トランザクションは、経過時間が初期化パラメータ DISTRIBUTED_LOCK_TIMEOUT で設定された時間間隔に達した場合、要求したロックに対する待機をタイムアウトできます。

ユーザー・ロック

ユーザー・ロックの作成

アプリケーションで、Oracle ロック管理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。確保されているユーザー・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、『Oracle8i PL/SQL 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。

ユーザー・ロックの例

ユーザー・ロックは、次のような場合に使用します。

- 端末などの装置に対して排他アクセスを可能にする場合
- アプリケーション・レベルの読み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させて、逐次処理を実行する場合

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。

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 Tool (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)。

シリアルライズ可能トランザクションを使用した同時実行性の制御

Oracle Server 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、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。

ほとんどのアプリケーションの場合、この同時実行性モデルが適切なモデルです。ただし、トランザクションがシリアルライズ可能になるようにした方がよい場合もあります。シリアルライズ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。つまり、シリアル・モードで実行される同時トランザクションは、1 つずつ実行するようにスケジュールされた場合に可能なデータベースの変更しか行えません。

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。

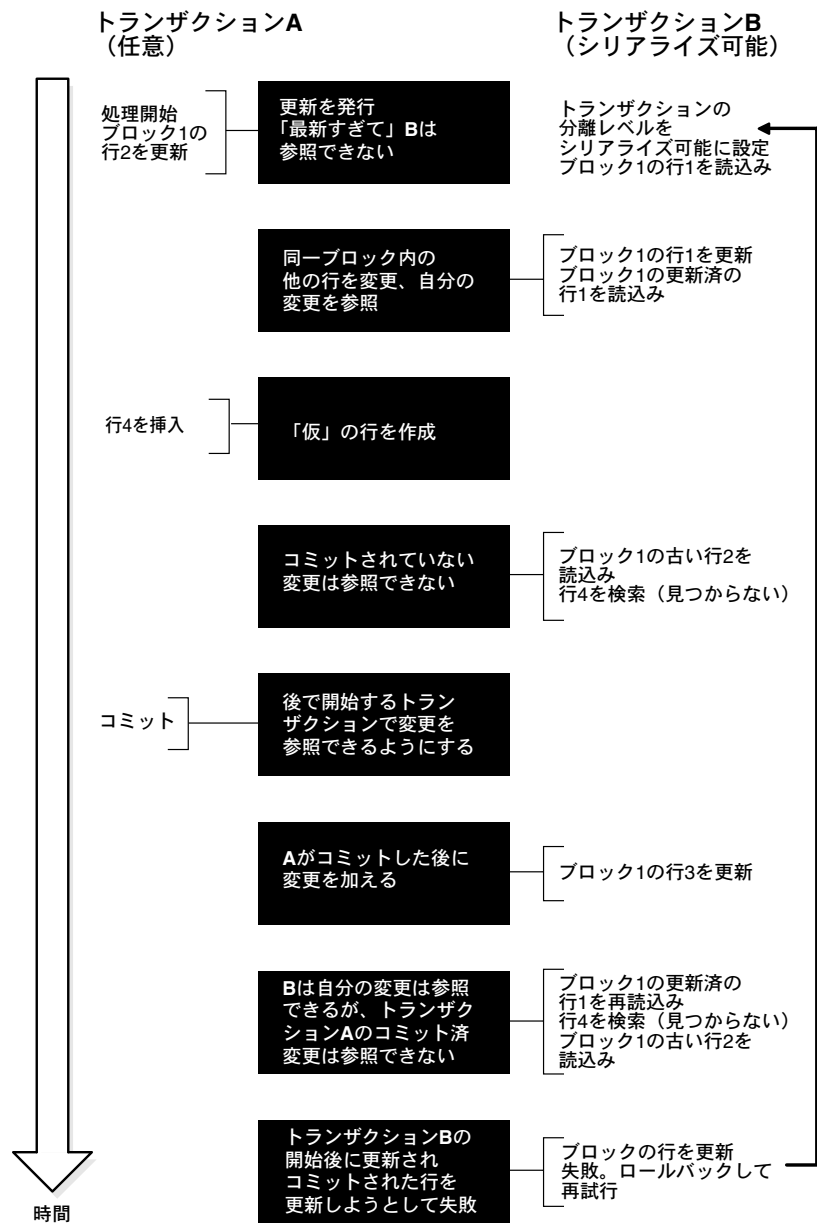
表 7-3 ANSI 分離レベル

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
シリアルライズ可能	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。

非コミット読み	Oracle では「内容を保証しない読み」は許可されません。これはスループットの高い Oracle では不要です。
コミット読み	Oracle は、コミット読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	Oracle は、シリアライズ可能により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。
シリアライズ可能	この分離レベルは、後述するように、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



シリアライズ可能トランザクションの相互作用

7-27 ページの図 7-3 は、シリアライズ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、シリアライズ可能またはコミット読込みのいずれか）との間の相互作用を示しています。

シリアライズ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逐次化できません。」で失敗したとき、アプリケーションは次の処置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。シリアライズ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。シリアライズ可能モードを使用するには、INITRANS を少なくとも 3 に設定する必要があります。

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。これ以外の場合、次のエラーが発生します。

ORA-01453: SET TRANSACTION はトランザクションの最初の文でなければなりません。

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、『Oracle8i リファレンス・マニュアル』を参照してください。

INITRANS パラメータ

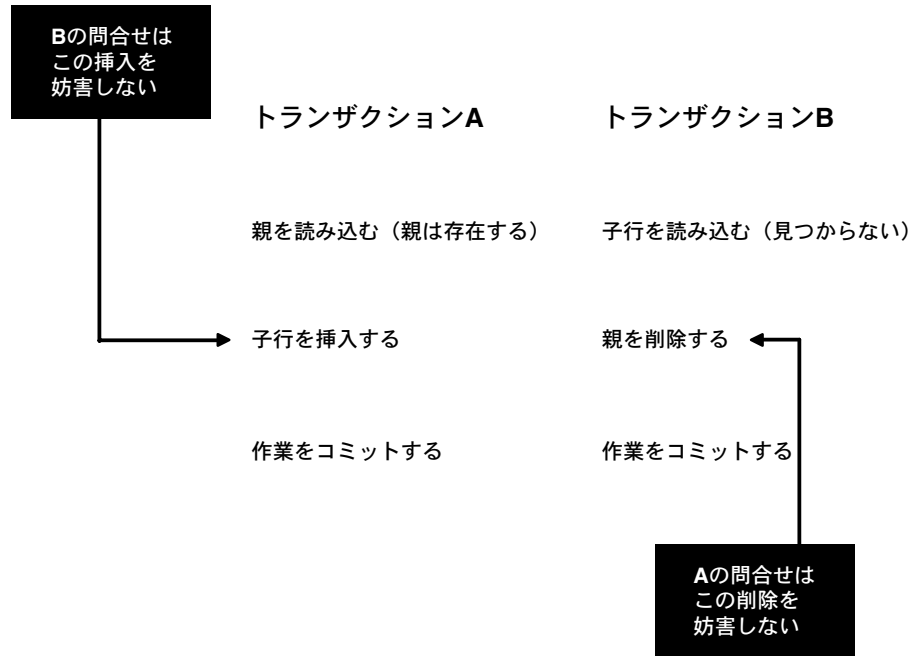
Oracle では、同時トランザクションによるアクセスを管理するために、各データ・ブロックに制御情報を格納します。したがって、トランザクション分離レベルをシリアライズ可能に設定する場合は、ALTER TABLE コマンドを使用して、INITRANS を少なくとも 3 に設定する必要があります。このパラメータを使用すると、Oracle は、ブロックにアクセスした最新トランザクションの履歴を記録するのに十分な記憶域を、それぞれのブロック内に割り当てます。同じブロックを更新するトランザクションの数が多い表には、さらに大きい値を使用する必要があります。

参照整合性およびシリアルライズ可能トランザクション

Oracle は、シリアルライズ可能トランザクション内であっても読み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上書きできます。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）。シリアルライズ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、コミット読み込みトランザクションおよびシリアルライズ可能トランザクションの両方に当てはまるため注意してください。

7-27 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。対応する子行を挿入する前に、一方のトランザクションが親表を読み込み、特定の主キー値を持つ行が存在するかどうかを判断します。もう一方のトランザクションは、親行を削除する前に、対応するディテール行が存在しないことをチェックします。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 が実行した読み込みのために、トランザクション B が親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 が子行の問合せを行っても、トランザクション A が子行を挿入できなくなることはありません。したがって、前述の使用例の場合、対応する親行を持たない子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ライズ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、トランザクションによっては、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、アプリケーション開発者が特に保証する必要があります。これには、SQL92 シリアライズ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。トランザクション B は、処理ステップの順序を逆にして、トランザクション A が親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、まず親行を削除し、その後の問合せで子表内に対応する行が存在することが判明したら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も参照整合性を施行できます。たとえば、子表への INSERT によって、PRE-INSERT 行レベル・トリガーを起動して、対応する親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、子行を挿入するトランザクションの処理中に、親行が（存在する場合）データベース内に残るようにします。対応する親行が存在しない場合、トリガーは子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、コミット読み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、シリアルライズ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、前述のとおりに参照整合性が正しく適用されます。

コミット読み込み分離およびシリアルライズ可能分離

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 1 つを選択できます。コミット読み込み分離レベルおよびシリアルライズ可能分離レベルは、どちらも高度な一貫性および同時実行性を提供します。これら 2 つのレベルは、Oracle の読取り一貫性の複数バージョン同時実行性の制御モデルおよび排他行レベル・ロックの実装による競合状態の軽減という利点を提供するもので、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。この項の残りの部分では、前述の 2 つの分離レベルを比較し、その選択の際に役立つ情報を示します。

トランザクション集合の整合性

Oracle のコミット読み込み分離レベルおよびシリアルライズ可能分離レベルについて説明するには、次の項目を考慮すると役立ちます。

- 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み込み順序
- 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

どの読み込みにおいても、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）は、「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み込みもあれば、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み込みもあります。トランザクション集合整合でない操作には、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まったく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。

コミット読み込みモードで実行するトランザクションでは、文単位のトランザクション集合の整合性が提供されます（問合せによって読み込まれたすべての行は、その問合せが始まる前にコミットされているはずであるため）。同様に、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の整合性が提供されます。

（Oracle8 の場合と異なり）他のデータベース・システムでは、コミット読み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しか見えないため、トランザクション集合は整合していません。このことは、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ない（その逆もまた同じである）ことを意味します。Oracle のコミット読み込みモードではこのような影響を受けないため、読み込みロック・システムよりも高い整合性が得られます。

読み込みロック・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。

機能比較の概要

表 7-4 に、コミット読み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概説します。

表 7-4 コミット読み込みトランザクションおよびシリアルライズ可能トランザクション

	コミット読み込み	シリアルライズ可能
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み込み	不可能	不可能
反復不能読み込み	可能	不可能
仮機能	可能	不可能
ANSI/ISO SQL 92 への準拠	はい	はい
スナップショット時間の読み込み	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	はい	はい
読み込みブロック書き込み機能	いいえ	いいえ
書き込みブロック読み込み機能	いいえ	いいえ
異なる行の書き込みブロック書き込み機能	いいえ	いいえ
同じ行の書き込みブロック書き込み機能	はい	はい
阻止しているトランザクションの待機	はい	はい
「このトランザクションのアクセスを逐次化できません。」というエラーが発生する可能性	いいえ	はい
トランザクション異常終了のブロック化後のエラー	いいえ	いいえ
トランザクション・コミットのブロック化後のエラー	いいえ	はい

分離レベルの選択

アプリケーションの設計者および開発者は、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。この選択は、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行う必要があります。

多数のユーザーが、同時にトランザクションを次々に送る環境の場合、設計者は、予期されるトランザクション到着頻度と応答時間要件という観点からトランザクション・パフォーマンス要件を評価して、パフォーマンス期待値を満たしつつ必要十分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。ほとんどの場合、高パフォーマンス環境では、分離レベルを選択するときに、整合性と同時実行性（トランザクションのスループット）との間でのトレードオフが必要です。

どちらの Oracle 分離モードも、行レベル・ロックと Oracle のマルチバージョン同時実行性制御システムとを組み合わせることによって、高レベルの整合性および同時実行性（およびパフォーマンス）を提供します。Oracle では読み込みと書き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、コミット読み込み分離およびシリアライズ可能分離により、コミットされていない（内容が保証されない）データの読み込みを防止し、高レベルの同時実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。

コミット読み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（仮読み込みおよび反復不能読み込みのため）矛盾した結果が生成される可能性は多少高くなりますが、かなり高い同時実行性を提供できます。シリアライズ可能分離レベルの場合は、仮読み込みおよび反復不能読み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、読み込み / 書き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、シリアライズ可能モードでは、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逐次化できません。」というエラーの有無をチェックする必要があり、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チェック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み込みが書き込みをブロックしないという事実を考慮する必要があります。

アプリケーションのヒント

トランザクションがシリアライズ可能モードで実行する場合、シリアライズ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、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逐次化できません。

このエラーが発生した場合の適切な処置は、現行のトランザクションをロールバックし、それを再実行することです。ロールバック後、このトランザクションは新しいトランザクション・スナップショットを取得し、DML 操作は成功する可能性があります。

トランザクションのロールバックおよび繰返しが必要なため、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、『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』および第 12 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ことが必要な場合があります。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ことが必要な場合もあります。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。

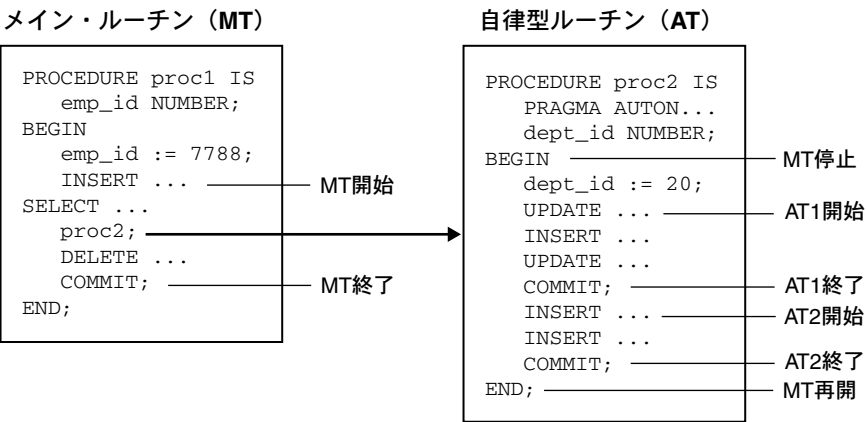
自律型トランザクション (AT) は、別のトランザクション (メイン・トランザクション (MT)) によって開始される独立したトランザクションです。自律型トランザクションを使用すると、メイン・トランザクションを停止し、SQL 操作を実行し、その SQL 操作をコミットまたは実行し、その後でメイン・トランザクションを再開できます。

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 (コンパイラ指示句) AUTONOMOUS_TRANSACTION でマークされたルーチンです。プラグマは、PL/SQL コンパイラに対して、ルーチンを自律型 (独立) 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) 自律型 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



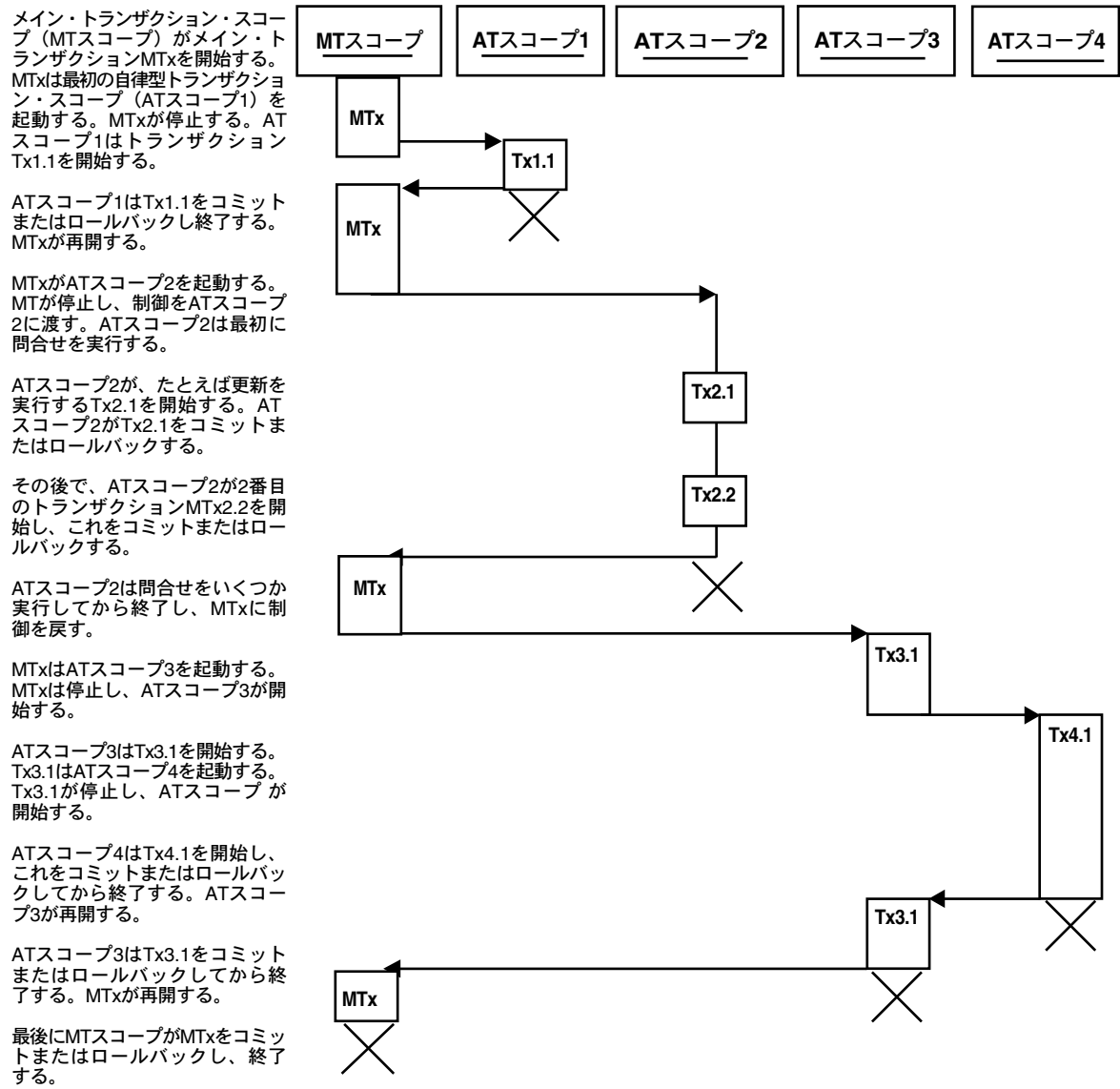
自律型ルーチンの実行可能セクションに入ると、メイン・トランザクションが停止します。自律型ルーチンを終了すると、メイン・トランザクションが再開します。COMMITおよびROLLBACKによって、アクティブな自律型トランザクションは終了しますが、自律型ルーチンは終了しません。図 7-4 に示されているように、1つのトランザクションが終了すると、次の SQL 文が別のトランザクションを開始します。

自律型トランザクションの特性をさらにいくつか挙げます。

- 自律型トランザクションが影響する変更は、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のを待たなくても、ユーザーは更新された情報にアクセスできることを意味します。
- 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



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。

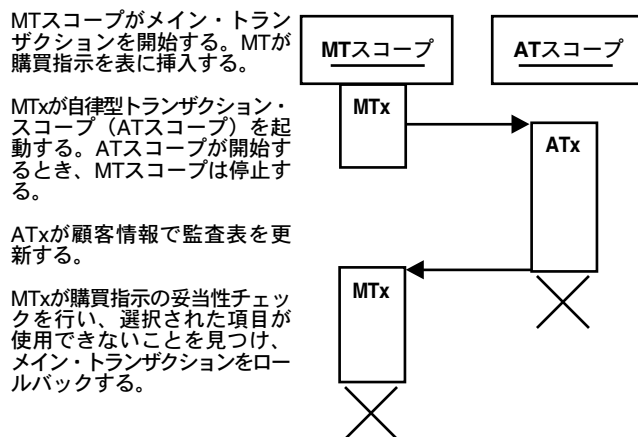
例に示されているように、自律型トランザクションおよびメイン・トランザクションを使用する場合は、4 種類の結果があり得ます。この結果を次の表に示します。表からわかるように、自律型トランザクションの結果とメイン・トランザクションの結果との間に依存性はありません。

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

購買指示の入力

この例では、顧客が購買指示を入力します。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）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ています。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。

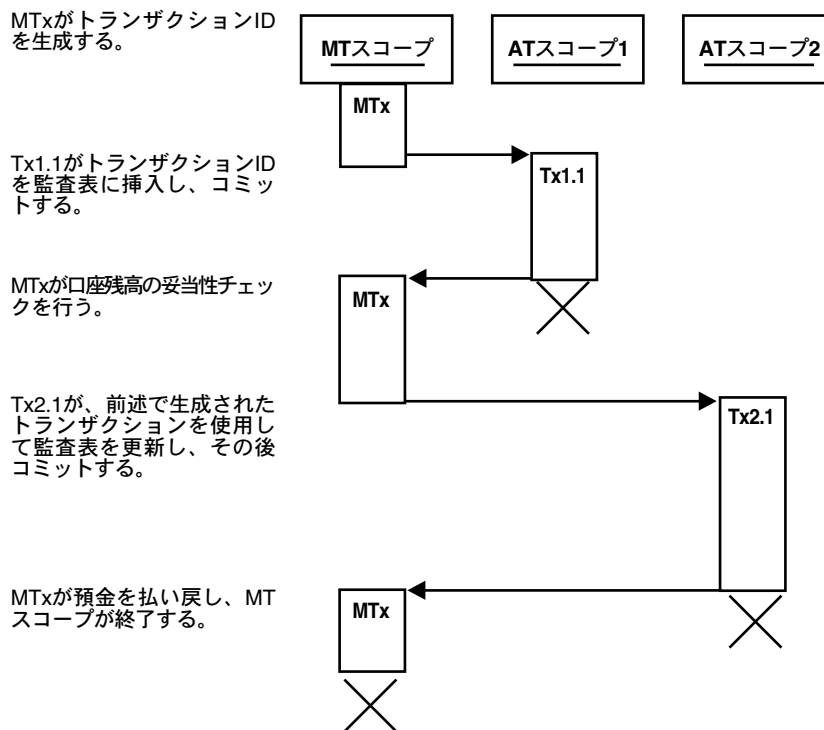
次の図には、このトランザクションで考えられる使用例を3つ示します。

- 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。

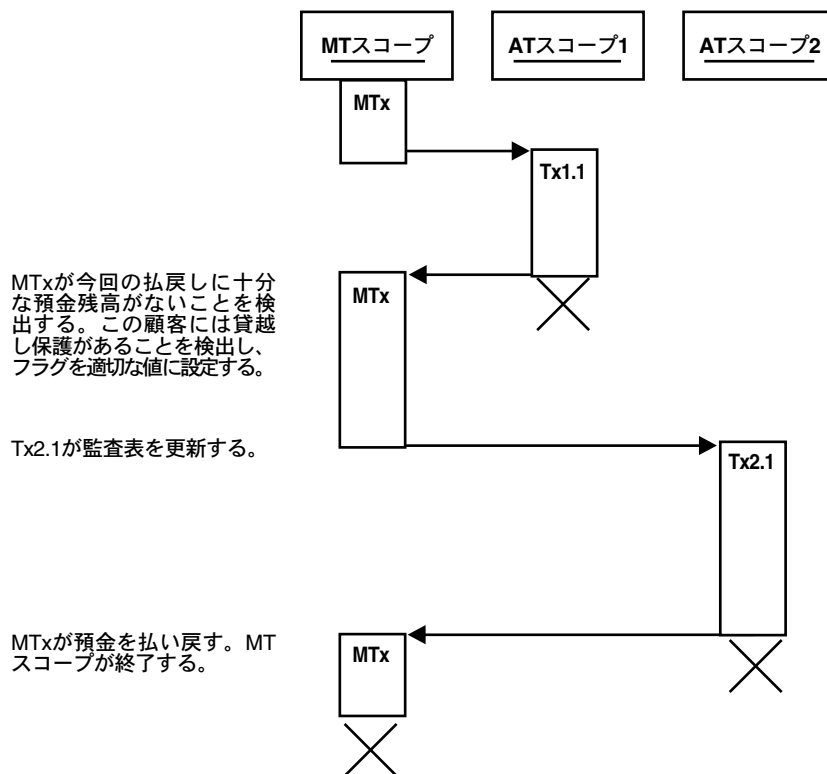
図 7-7 例：預金払戻し—十分な預金残高



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。

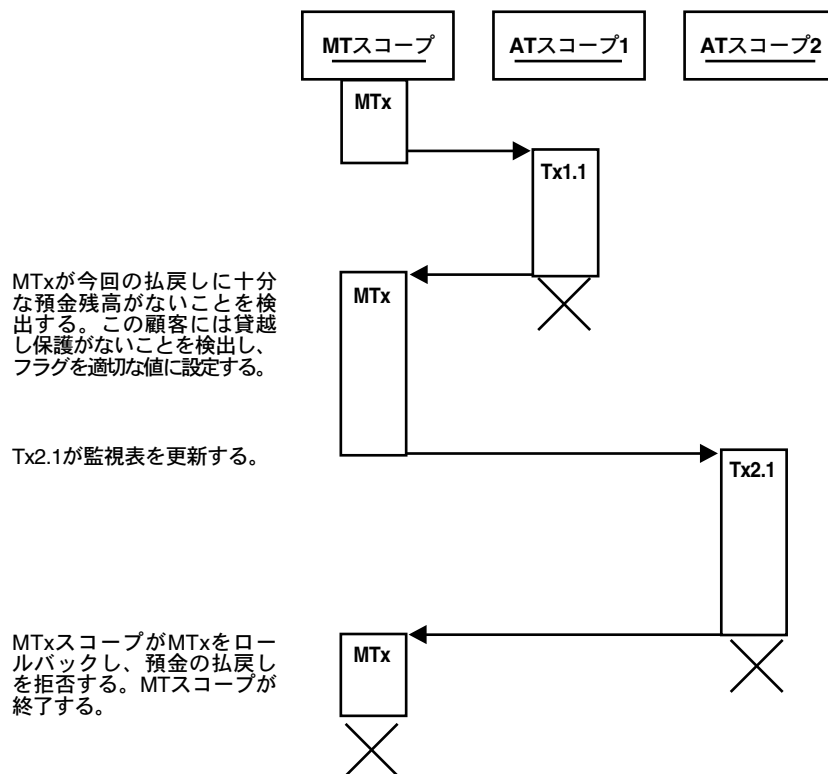
図 7-8 例：預金払戻し—不十分な預金残高で貸越し保護あり



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也没有せん。したがって、銀行は払戻しを差し止めます。

図 7-9 例：預金払戻しー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、『Oracle8i 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、PRAGMA（コンパイラ指示句）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（独立）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/or 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/or 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;
```

動的 SQL

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 の以前のリリースでは、DBMS_SQL パッケージを使用しないと PL/SQL アプリケーションで動的 SQL を実装できませんでした。Oracle8i では、DBMS_SQL パッケージのかわりにシステム固有の動的 SQL が導入されています。システム固有の動的 SQL を使用すると、動的 SQL 文を PL/SQL ブロック内に直接入れることができます。

この章の内容は次のとおりです。

- 動的 SQL とは
- 動的 SQL の使用時期
- システム固有の動的 SQL を使用した使用例
- システム固有の動的 SQL と DBMS_SQL パッケージの対比
- PL/SQL 以外のアプリケーション開発言語

動的 SQL とは

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、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、この利点は次のとおりです。

- 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 には前述のような利点があるため、動的 SQL の使用は、静的 SQL では目標が達成できない場合、または静的 SQL の方が動的 SQL よりも使用が煩雑になる場合に限る必要があります。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、プログラムで何らかの処理作業を終了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表に対して標準の問合せを実行するレポート・アプリケーションがあり、表の正確な名前が実行時までわからないとします。データ・ウェアハウス内に大量のデータを効率的に入れるために、四半期ごとに新しい表を作成して、該当四半期の請求情報を格納するものとします。表の定義はすべて同じで、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ます。たとえば、INV_01_1997、INV_04_1997、INV_07_1997、INV_10_1997 または INV_01_1998 のような名前です。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。

静的 SQL の場合、プログラムの SQL 文によって参照されるデータ定義情報（表定義など）がコンパイル時にわかっている必要があります。データ定義を変更した場合、プログラムも変更して再コンパイルする必要があります。動的 SQL プログラムの場合は、データ定義情報の変更に対応できます。これは、実行時に SQL 文をその場で変更できるためです。このため、動的 SQL は静的 SQL よりはるかに柔軟です。動的 SQL を使用すると、再使用可能なアプリケーション・コードを作成できます。これは、使用される特定の SQL 文から独立したプロセスをコードで定義するためです。

さらに、動的 SQL では静的 SQL プログラムではサポートされていない SQL 文（データ定義言語（DDL）文など）を実行できます。このような SQL 文がサポートされるため、PL/SQL プログラムで多くのことが実現できるようになります。

注意： 動的 SQL プログラムという用語は、動的 SQL を含むプログラムという意味です。静的 SQL プログラムは静的 SQL のみを含み、動的 SQL は含まないプログラムです。

動的 SQL の使用時期

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。

動的 DML 文の実行

動的 SQL は DML 文の実行に使用します。DML 文では正確な SQL 文は実行時までわかりません。この例については、8-19 ページの「[DBMS_SQL パッケージ・コードおよびシステム固有の動的 SQL コードの例](#)」および 8-10 ページの「[DML 操作の例](#)」にある DML の例を参照してください。

静的 SQL ではサポートされていない文の PL/SQL での実行

PL/SQL の静的 SQL では、次の種類の文は実行できません。

- 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、『Oracle8i SQL リファレンス』を参照してください。

PL/SQL ブロック内で以上のような文を実行する必要がある場合は、動的 SQL を使用します。

また、PL/SQL では静的 SQL で SELECT 文に TABLE 句を使用できません。動的 SQL にはこの制約はありません。たとえば、次の PL/SQL ブロックには、TABLE 句およびシステム固有の動的 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が、その中のいくつかを次に挙げます。

- 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わ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わせるアプリケーション

前述の例については、8-19 ページの「[問合せの例](#)」を参照してください。問合せの例については、8-9 ページの「[システム固有の動的 SQL を使用した使用例](#)」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、データベースの定義はコンパイル時に決定できますが、表の名前は決定できないことがあります。これは、新しい表が定期的に生成されるためです。アプリケーションではこのようなデータにアクセスする必要がありますが、実行時まで表の正確な名前を知る方法はありません。

動的 SQL を使用すると、アクセスの必要な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 とは](#)」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 を使用して、実行時にユーザーが表の名前を指定できるようにできます。

```
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

静的 SQL を使用する場合は、SQL 文の作成方法（文の中にヒントを入れるか、ヒントを含めるかどうか、どのヒントを入れるかなど）をコンパイル時に決定する必要があります。ただし、動的 SQL を使用すると、実行の最適化または SQL 文へのヒントの動的な連結（あるいはその両方）が可能な方法で 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、ユーザーが `a_hint` として次の値のいずれかを渡すことができます。

- `a_hint = '/**+ ALL_ROWS */'`
- `a_hint = '/**+ FIRST_ROWS */'`
- `a_hint = '/**+ CHOOSE */'`
- その他の有効なヒント・オプション

参照： ヒントの使用の詳細は、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

動的 PL/SQL ブロックの起動

無名 PL/SQL ブロックを起動するには、EXECUTE IMMEDIATE 文を使用できます。動的 PL/SQL ブロックを起動できると、実行されるモジュールが実行時に動的に決定されるようなアプリケーションの拡張およびカスタマイズに役立ちます。

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。

```
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;
/
```

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。ただし、システム固有の動的 SQL を使用すると、次のような拡張可能な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使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： 実行者権限およびシステム固有の動的 SQL の使用の詳細は、『Oracle8i 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。

システム固有の動的 SQL を使用した使用例

この項で説明されている使用例には、システム固有の動的 SQL が提供する能力および柔軟性が示されています。この使用例には、システム固有の動的 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

データ・モデル

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システム固有の動的 SQL 操作を説明します。

DML 操作の例

次のシステム固有の動的 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;
/
```

DDL 操作の例

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;
/
```

次のプロシージャは、事務所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;
/
```

動的な単一行問合せの例

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り出します。

```
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;
/
```

動的な複数行問合せの例

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;
/
```

システム固有の動的 SQL と DBMS_SQL パッケージの対比

Oracle では、PL/SQL 内で動的 SQL を使用するために、システム固有の動的 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。システム固有の動的 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。

最もシステム固有度の高い動的 SQL 文を処理するには、EXECUTE IMMEDIATE 文を使用します。ただし、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： システム固有の動的 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、『Oracle8i 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プログラム 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ためのプログラム・インタフェースが含まれています。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 つの方法の利点を詳しく説明します。

参照： システム固有の動的 SQL の使用方法の詳細は、『Oracle8i 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、『Oracle8i PL/SQL 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。『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』では、システム固有の動的 SQL は単に動的 SQL と呼ばれています。

システム固有の動的 SQL の利点

システム固有の動的 SQL には、DBMS_SQL と比べて次のような利点があります。

使用しやすさ

システム固有の動的 SQL は、DBMS_SQL パッケージと比べて簡単に使用できます。システム固有の動的 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。さらに、システム固有の動的 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もコンパクトで読みやすくなります。

DBMS_SQL パッケージは、システム固有の動的 SQL ほど使用しやすくはありません。多数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使用する必要があります。DBMS_SQL パッケージを使用する場合は、通常、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。システム固有の動的 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 は、同じ操作を DBMS_SQL パッケージを使用して実行したときとシステム固有の動的 SQL を使用して行ったときの必要コード量の違いを示します。

表 8-1 DBMS_SQL パッケージおよびシステム固有の動的 SQL でのコード量の比較

DBMS_SQL パッケージ	システム固有の動的 SQL
<pre>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; /</pre>	<pre>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; /</pre>

パフォーマンスの向上

PL/SQL インタプリタにはシステム固有の動的 SQL のサポートが組み込まれているため、PL/SQL でのシステム固有の動的 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。このため、システム固有の動的 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もパフォーマンスが向上します。通常、システム固有の動的 SQL 文では、DBMS_SQL パッケージを使用する同等の文よりも 1.5 ～ 3 倍パフォーマンスが向上します。もちろん、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る可能性があります。

DBMS_SQL パッケージはプロシージャ型 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。たとえば、DBMS_SQL パッケージでは、変数をバインドするたびに PL/SQL バインド変数を自分の領域にコピーし、後で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。これに対して、システム固有の動的 SQL では、文の準備ステップ、バインド・ステップおよび実行ステップを 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てパフォーマンスが向上します。

パフォーマンス上のヒント システム固有の動的 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、次のシステム固有の動的 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;
/
```

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;
/
```

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使用されているため、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

システム固有の動的 SQL は、PL/SQL で静的 SQL がサポートしているすべての型をサポートしているため、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サポートします。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、『Oracle8i PL/SQL 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。

レコードへのフェッチのサポート

システム固有の動的 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 パッケージはサポートしません。システム固有の動的 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 パッケージの利点

DBMS_SQL パッケージは、システム固有の動的 SQL と比べて次のような利点があります。

クライアント側プログラムのサポート

現在、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、システム固有の動的 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 コマンドと非常に似ています。システム固有の動的 SQL には DESCRIBE 機能はありません。

バルク動的 SQL のサポート

バルク SQL とは、1 つの DML 文内で複数のデータ行を処理する機能です。バルク SQL では、SQL とホスト言語間でのコンテキストの切替え数を少なくして、パフォーマンスを改善します。DBMS_SQL パッケージでは、現在、バルク動的 SQL をサポートしています。

システム固有の動的 SQL にはバルク操作の直接サポートはありませんが、「BEGIN ... END」ブロックにバルク SQL 文を入れて、ブロックを動的に実行することによって、システム固有のバルク動的 SQL 文をシミュレーションできます。この方法によって、システム固有の動的 SQL 内でバルク SQL の利点を実現できます。たとえば、次のシステム固有の動的 SQL コードでは、1 つの表の ename 列を別の表にコピーします。

```
CREATE TYPE name_array_type IS
    VARRAY(100) of VARCHAR2(50)
/

CREATE OR REPLACE PROCEDURE copy_ename_column
    (table1 VARCHAR2, table2 VARCHAR2) IS
    ename_col NAME_ARRAY_TYPE;
```

```
BEGIN
  -- bulk fetch the 'ename' column into a VARRAY of VARCHAR2s.
  EXECUTE IMMEDIATE
    'BEGIN
      SELECT ename BULK COLLECT INTO :tab
      FROM ' || table1 || ';
    END; '
  USING OUT ename_col;

  -- bulk insert the 'ename' column into another table.
  EXECUTE IMMEDIATE
    'BEGIN
      FORALL i IN :first .. :last
        INSERT INTO ' || table2 || ' VALUES (:tab(i));
    END; '
  USING ename_col.first, ename_col.last, ename_col;
END;
/
```

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。システム固有の動的 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・コードおよびシステム固有の動的 SQL コードの例は、8-22 ページの「[DML RETURNING 操作の例](#)」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。システム固有の動的 SQL はサポートしません。

SQL 文の再使用

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。

これに対して、システム固有の動的 SQL では、SQL 文を使用するたびに実行用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。使用のたびに文を準備すると、パフォーマンスが少し低下します。

ただし、Oracle の共有カーソル・メカニズムによってこの低下は最小限に抑えられ、システム固有の動的 SQL によるパフォーマンスの向上に比べれば、通常、ごくわずかです。

DBMS_SQL パッケージ・コードおよびシステム固有の動的 SQL コードの例

次の例に、DBMS_SQL パッケージを使用した場合、およびシステム固有の動的 SQL を使用した場合の、操作の完了に必要なコードの違いを示します。具体的には、次の操作の例を示します。

- 問合せ
- DML 操作
- DML リターニング操作

一般に、システム固有の動的 SQL コードの方が読みやすくコンパクトなため、開発生産性が向上します。

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）からな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 の社員を問い合わせます。[表 8-2](#)に、DBMS_SQL パッケージおよびシステム固有の動的 SQL を使用してこの問合せを行うサンプル・コードを示します。

表 8-2 DBMS_SQL パッケージおよびシステム固有の動的 SQL を使用した問合せ

DBMS_SQL の問合せ操作	システム固有の動的 SQL の問合せ操作
<pre>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; /</pre>	<pre>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; /</pre>

DML の例

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 変数の deptnumber、deptname および location に列値が含まれて
いる新しい行を挿入します。表 8-3 に、DBMS_SQL パッケージおよびシステム固有の動的
SQL を使用してこの DML 操作を行うサンプル・コードを示します。

表 8-3 DBMS_SQL パッケージおよびシステム固有の動的 SQL を使用した DML 操作

DBMS_SQL の DML 操作	システム固有の動的 SQL の DML 操作
<pre>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; /</pre>	<pre>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; /</pre>

DML RETURNING 操作の例

次に、部門番号 (deptnumber) および新しい所在地 (location) がわかっているときに、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';
```

この例は、PL/SQL 変数の deptnumber、deptname および location に列値が含まれている新しい行を挿入します。表 8-4 に、DBMS_SQL パッケージおよびシステム固有の動的 SQL を使用してこの DML RETURNING 操作を行うサンプル・コードを示します。

表 8-4 DBMS_SQL パッケージおよびシステム固有の動的 SQL を使用した DML RETURNING 操作

DBMS_SQL の DML RETURNING 操作	システム固有の動的 SQL の DML RETURNING 操作
<pre>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; /</pre>	<pre>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; /</pre>

PL/SQL 以外のアプリケーション開発言語

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。動的 SQL を使用するプログラムの実装には、他のアプリケーション開発言語も使用できます。このようなアプリケーション開発言語には、C/C++、COBOL、Java などがあります。

C/C++ を使用する場合は、Oracle コール・インタフェース (OCI) で動的 SQL を使用するアプリケーションを開発し、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。同様に、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。

これまで、DBMS_SQL パッケージを使用しないと、PL/SQL アプリケーションで動的 SQL を実装できませんでした。このパッケージを使用するときは、パフォーマンス上の考慮点をはじめとして多数の制約があります。このため、アプリケーション開発者は、PL/SQL のかわりに前述の言語のいずれかを使用して動的 SQL を実装していることがあります。ただし、PL/SQL に新しくシステム固有の動的 SQL が導入されたことで、動的 SQL に PL/SQL を使用する際の欠点の多くがなくなっています。

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、動的 SQL 操作の実行に必要なネットワークのラウンドトリップによって、パフォーマンスが低下する可能性があります。このようなアプリケーションは、通常、クライアント上に常駐するため、動的 SQL 操作の完了にはより多くのネットワーク・コールを必要とします。このようなアプリケーションの場合は、システム固有の動的 SQL を使用する PL/SQL のストアド・プロシージャおよびストアド・ファンクションに、動的 SQL の機能を移行することを検討してください。ストアド・プロシージャはサーバーに置くことができるため、ネットワーク・オーバーヘッドをなくすることができます。この方法でアプリケーションのパフォーマンスが向上する可能性があります。これで、アプリケーションからストアド・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 PL/SQL 以外のアプリケーション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。

- 『Oracle8i コール・インタフェース・プログラマーズ・ガイド』
- 『Oracle8i Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』
- 『Oracle8i Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』
- 『Oracle8i Java ストアド・プロシージャ開発者ガイド』

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。内容は次のとおりです。

- [PL/SQL プログラム・ユニット](#)
- [PL/SQL コードのラッピング](#)
- [リモート依存性](#)
- [カーソル変数](#)
- [コンパイル時エラー](#)
- [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](#)

PL/SQL プログラム・ユニット

PL/SQL は、最新の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体を提供します。

PL/SQL ブロック内部で SQL データ操作言語（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、データ定義言語（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 Tools 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 Oracle コール・インタフェース（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 言語の詳細は、『Oracle8i 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

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、『Oracle8i PL/SQL 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。

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と、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： 詳細は、9-43 ページの「[ランタイム・エラー処理](#)」および『Oracle8i 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 Tool から対話形式で使用するか、プリコンパイラ、OCI、SQL*Module アプリケーションで使います。通常は、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います。

参照： 9-38 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、次の特徴を持つ 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ます、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 つ返し、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・プロシージャから区別され、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つの主要な部分があります。

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	入力のみパラメータ (IN)、出力のみパラメータ (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、サブプログラムに対してローカルではない変数の値を変更する副作用の影響をファンクションが受け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、『Oracle8i 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 または VARCHAR2 などの無制約型名
- %TYPE 属性または %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関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 の使用をお勧めします。

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、重複ループバック 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 Tool 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。

```
SQLPLUS> @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、『Oracle8i リファレンス・マニュアル』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・セグメントを作成する必要があります。

参照： 9-51 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、まず DROP PROCEDURE 文または DROP FUNCTION 文を使用して削除（DROP）してから、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。

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、パッケージ本体およびパッケージ全体を、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 Server 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 Server 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「外部ルーチン」](#)を参照してください。

PL/SQL パッケージ

パッケージとは、データベース内に一緒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、利点が多くあります。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みます。
- 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、『Oracle8i 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、パッケージの直接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。

例 次に、Employee_management というパッケージのパッケージ仕様部を示します。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*Plus 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、DOS や 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。7.3 より前のリリースでは、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プロシージャ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しかアクセスできません。外部ユーザーはどの権限を持っ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よくあります。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、パッケージの構造体の名前は、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。セッションのインスタンス化されたパッケージ（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化（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* のインスタンス化が失われます。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、1 回目は次のエラーが戻されます（*string* に文字列が入ります）。

ORA-04068: パッケージ *stringstringstring* 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* をコールした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。

実働環境の多くでは、パッケージが無効となるような 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には、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 パッケージ・プロシージャ

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 Server には多数のパッケージが組み込まれています。アプリケーションの作成時にこれらのパッケージで提供される機能を使用するか、または、単に独自のストアード・プロシージャの作成時にこれらのパッケージを参考にすることができます。

この項では、提供される個々のパッケージをリストし、詳しい説明が記載されているマニュアルを示します。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。

特に指定がない限り、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。

表 9-2 Oracle パッケージ・プロシージャ一覧

パッケージ名	説明	ドキュメント
Calendar (注意 #2 を参照)	カレンダー・メンテナンス機能を提供します。	『Oracle8i Time Series ユーザーズ・ガイド』
DBMS_ALERT	データベース・イベントの非同期通知をサポートします。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_APPLICATION_INFO	監査またはパフォーマンスの追跡のために、アプリケーションの名前をデータベースに登録します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_AQ	メッセージ（事前定義済のオブジェクト型）をキューに追加するか、またはメッセージをキューから削除します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_AQADM	事前定義済オブジェクト型のメッセージ用のキューまたはキュー・テーブルに関する管理機能を実行します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_BACKUP_RESTORE (Windows NT のみ)	Windows NT 環境でファイル名を正規化します。	『Oracle8i 移行ガイド』
DBMS_DDL	ストアド・プロシージャからの SQL DDL 文に対するアクセスを提供し、DDL としては使用できない特殊な管理操作を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_DEBUG	Oracle Server 内の PL/SQL デバッガ・レイヤー（プローブ）に対する PL/SQL API です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_DEFER	レプリケートされたトランザクション型の遅延リモート・プロシージャ・コール機能に対するユーザー・インタフェースを提供します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_DEFER_QUERY	ビュー経由では表示されない遅延リモート・プロシージャ・コールのキュー・データに対する問合せを許可します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_DEFER_SYS	レプリケートされたトランザクション型の遅延リモート・プロシージャ・コール機能に対するシステム管理者用インタフェースを提供します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_DESCRIBE	名前の完全な変換およびセキュリティ・チェックを使用したストアド・プロシージャの引数を記述します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

表 9-2 Oracle パッケージ・プロシージャ一覧 (続き)

パッケージ名	説明	ドキュメント
DBMS_DISTRIBUTED_TRUST_ADMIN	Trusted Database リストをメンテナンスします。Trusted Database リストは、特定のサーバーからの特権データベース・リンクが受け入れられるかどうかの判断に使用されます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_HS	異機種間サービス・ディクショナリのオブジェクトを作成および変更できます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_HS_PASSTHROUGH	異機種間サービスを使用して非 Oracle システムに通過的 SQL 文を送信できます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_IOT	索引構成表の連鎖行に対する参照を入れることができる表を、ANALYZE コマンドを使用して作成します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_JOB	定期的に行う管理プロシージャをスケジューリングします。ジョブ・キューのインタフェースとしても機能します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_LOB	Oracle ラージ・オブジェクト (LOB)・データ型 (BLOB、CLOB (読み込み / 書き込み)、BFILE (読み取り専用)) に対する操作のための汎用ルーチンを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_LOCK	Oracle ロック・マネージメント・サービスを使用してロックの要求、交換および解放ができます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_LOGMNR	ログ・リーダーを初期化および実行する機能を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_MVIEW	DBMS_SNAPSHOT のシノニムです。同じリフレッシュ・グループに属していないスナップショットをリフレッシュし、ログを削除します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_OBFUSCATION_TOOLKIT	データ暗号化規格を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_OFFLINE_OG	マスター・グループのオフライン・インスタンスエーションのためのパブリック API を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_OFFLINE_SNAPSHOT	スナップショットのオフライン・インスタンスエーションのためのパブリック API を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_OLAP	サマリー、ディメンションおよびクエリー・リライ用のプロシージャを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_ORACLE_TRACE_AGENT	Oracle7 Server 内の Oracle Trace 機能に対して、クライアントがコール可能なインタフェースを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

表 9-2 Oracle パッケージ・プロシージャ一覧 (続き)

パッケージ名	説明	ドキュメント
DBMS_ORACLE_TRACE_USER	Oracle7 Server の Oracle Trace 機能に対して、コール側ユーザー用のパブリック・アクセスを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_OUTPUT	後で取り出せるように、情報をバッファ内に蓄積します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_PCLXUTIL	パーティション・ワイズ・ローカル索引を作成するパーティション内パラレル化を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_PIPE	セッション間でのメッセージ送信を可能にするデータベース管理システム・パイプ・サービスを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_PROFILER	既存の PL/SQL アプリケーションのプロファイルを作成し、パフォーマンスのボトルネックを識別する Probe Profiler API を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_RANDOM	組込みの乱数ジェネレータを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_RECTIFIER_DIFF	レプリケートされた 2 つのサイト間でのデータの不整合を検出および解決するための API を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REFRESH	トランザクション処理として一貫性のある時点で、同時にリフレッシュできるスナップショットのグループを作成します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPAIR	データ破損時の修復プロシージャを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPCAT	レプリケーションのカatalogおよび環境を管理および更新するルーチンを提供します。レプリケーション・オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPCAT_ADMIN	対称レプリケーション機能に必要な権限を持つユーザーを作成します。レプリケーション・オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPCAT_INSTANTIATE	実行テンプレートをインスタンス化します。レプリケーション・オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPCAT_RGT	リフレッシュ・グループ・テンプレートのメンテナンスおよび定義を制御します。レプリケーション・オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_REPUTIL	表複製のためのシャドウ表、トリガーおよびパッケージを生成するルーチンを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

表 9-2 Oracle パッケージ・プロシージャ一覧 (続き)

パッケージ名	説明	ドキュメント
DBMS_RESOURCE_MANAGER	プラン、コンシューマ・グループおよびプラン指示句をメンテナンスします。プラン・スキーマに対する変更をまとめてグループ化できるような方法も提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_RESOURCE_MANAGER_PRIVS	リソース・コンシューマ・グループに関連する権限をメンテナンスします。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_RLS	行レベルのセキュリティ管理インタフェースを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_ROWID	ROWID を作成しその内容を解析するプロシージャを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SESSION	ストアド・プロシージャから ALTER SESSION 文および他のセッション情報へのアクセスを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SHARED_POOL	オブジェクトは共有メモリーに保管されます。通常の LRU メカニズムと異なり、メモリーからエージ・アウトしません。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SNAPSHOT (シノニム DBMS_MVIEW)	同じリフレッシュ・グループに属していないスナップショットをリフレッシュし、ログを削除します。分散オプションが必要です。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SPACE	標準 SQL では使用できないセグメント領域情報を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SPACE_ADMIN	標準 SQL では使用できない表領域およびセグメント領域の管理を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_SQL	動的 SQL を使用してデータベースにアクセスできます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_STANDARD (注意 #1 を参照)	アプリケーションと Oracle との対話を支援する言語機能を提供します。	文書化されていません。
DBMS_STATS	データベース・オブジェクト用に収集されたオプティマイザの詳細情報をユーザーが表示および変更する方法を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_TRACE	PL/SQL トレースを開始および停止するルーチンを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_TRANSACTION	ストアド・プロシージャから SQL トランザクション文へのアクセスを提供し、トランザクション・アクティビティを監視します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

表 9-2 Oracle パッケージ・プロシージャ一覧（続き）

パッケージ名	説明	ドキュメント
DBMS_TTS	トランSPORTABLE・セットが自己完結型かどうかをチェックします。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DBMS_UTILITY	各種ユーティリティ・ルーチンを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
DEBUG_EXTPROC	実行中のプロセスに接続可能なデバッグのあるプラットフォームで、外部プロシージャをデバッグできます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
OUTLN_PKG	格納されているアウトラインの管理に関連するプロシージャおよびファンクションへのインタフェースを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
PLITBLM (注意 #1 を参照)	索引表操作を実行します。	文書化されていません。
SDO_ADMIN (注意 #3 を参照)	空間オブジェクト用に空間索引の作成およびメンテナンスを実装するファンクションを提供します。	『Oracle8i Spatial ユーザーズ・ガイドおよびリファレンス』
SDO_GEOM (注意 #3 を参照)	空間オブジェクトに対する形状操作を実装するファンクションを提供します。	『Oracle8i Spatial ユーザーズ・ガイドおよびリファレンス』
SDO_MIGRATE (注意 #3 を参照)	空間データをリリース 7.3.3 および 7.3.4 から 8.1.x に移行するためのファンクションを提供します。	『Oracle8i Spatial ユーザーズ・ガイドおよびリファレンス』
SDO_TUNE (注意 #3 を参照)	Spatial Cartridge 内で使用される空間索引スキームの動作を決定するパラメータを選択するためのファンクションを提供します。	『Oracle8i Spatial ユーザーズ・ガイドおよびリファレンス』
STANDARD (注意 #1 を参照)	すべての PL/SQL プログラムで自動的に使用できるデータ型、例外およびサブプログラムを宣言します。	ドキュメント化されていません。
TimeSeries (注意 #2 を参照)	時系列データに対して抽出、取出し、算術、集合などの操作を実行するファンクションを提供します。	『Oracle8i Time Series ユーザーズ・ガイド』
TimeScale (注意 #2 を参照)	拡大および縮小ファンクションを提供します。	『Oracle8i Time Series ユーザーズ・ガイド』
TSTools (注意 #2 を参照)	管理 Tools プロシージャを提供します。	『Oracle8i Time Series ユーザーズ・ガイド』
UTL_COLL	問合せおよび更新の際に PL/SQL プログラムでコレクション・ロケータを使用できます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』

表 9-2 Oracle パッケージ・プロシージャ一覧 (続き)

パッケージ名	説明	ドキュメント
UTL_FILE	PL/SQL プログラムでオペレーティング・システム (OS) のテキスト・ファイルの読み込みおよび書き込みができる機能を提供し、制限付きの標準 OS ストリーム・ファイル I/O を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_HTTP	PL/SQL および SQL から HTTP コールアウトして、インターネット上のデータへのアクセスまたは Oracle Web Server カートリッジのコールを行うことができます。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_INADDR	インターネット・アドレッシングを提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_PG	COBOL の数値データを Oracle の数値に、Oracle の数値を COBOL 数値データに変換する関数を提供します。	『Oracle Procedural Gateway for APPC User's Guide』
UTL_RAW	RAW データ型の連結や減算などを実行する SQL 関数を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_REF	オブジェクトへの参照を提供し、PL/SQL プログラムでオブジェクトにアクセスできるようにします。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_SMTP	電子メールを送る PL/SQL 機能を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
UTL_TCP	サーバーと外部間の単純な TCP/IP ベースの通信をサポートする PL/SQL 機能を提供します。	『Oracle8i PL/SQL パッケージ・プロシージャ リファレンス』
Vir_Pkg (注意 #2 を参照)	Visual Information Retrieval 用の分析および変換ファンクションを提供します。	『Oracle8i Visual Information Retrieval User's Guide and Reference』

注意 #1:

DBMS_STANDARD、STANDARD および PLITBLM の各パッケージには、基本的な言語機能の実装を支援するサブプログラムが含まれています。このようなサブプログラムを直接コールすることはお薦めしません。したがって、このようなパッケージについてはマニュアルに記載していません。

注意 #2:

Time-Series、Image、Visual Information Retrieval、Audio および Server-Managed Video Cartridge の各パッケージは、パブリック・シノニムなしでユーザーの ORDSYS にインストールされます。

注意 #3:

Spatial Cartridge パッケージは、パブリック・シノニムなしでユーザーの MDSYS にインストールされます。

バルク・バインド

Oracle では、PL/SQL エンジンおよび SQL エンジンの 2 つのエンジンを使用して PL/SQL のブロックおよびサブプログラムを実行します。PL/SQL エンジンはプロシージャ型の文を実行し、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切り替えるため、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切替えの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切替えにより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が含まれます。

- VARRAY
- NESTED TABLE
- 索引付き表
- 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・バインドがない場合、コレクション内の要素は個々に SQL エンジンに送信されます。バルク・バインドでは、2 つのエンジン間でコレクション全体が渡されます。

バルク・バインドを使用すると、コレクション要素を使用する SQL 文の実行に必要なコンテキスト切替え数が抑えられ、パフォーマンスを改善できます。通常、バルク・バインドの使用により、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 文により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・バインドの使用の詳細は、『Oracle8i 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。

注意： 次の例を実行するには、データ構造を設定または削除しないと機能しない例もあります。

バルク・バインドを使用する場合 次の項では、バルク・バインドによってパフォーマンスが改善される使用例を説明します。アプリケーションでこのような使用がある場合、またはこのような使用を計画している場合は、バルク・バインドの使用を検討してください。

コレクションを参照する DML 文 バルク・バインドは、コレクションを参照する DML 文のパフォーマンスを改善するために使用できます。入力収集を SQL エンジンに送信する前にバルク・バインドするには、FORALL キーワードを使用します。SQL 文は、コレクション要素を参照する INSERT 文、UPDATE 文または DELETE 文である必要があります。

たとえば、次の PL/SQL ブロックは、バルク・バインドを使用しないで、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
  FOR i IN Id.FIRST..Id.LAST LOOP
    UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);
  END LOOP;
END;
```

このブロックを実行するには、更新される各従業員の SQL エンジンに対して、PL/SQL が SQL 文を送信します。更新対象の従業員数が多い場合は、PL/SQL エンジンと SQL エンジンとの間でコンテキストの切替えが多数発生し、パフォーマンスが低下する可能性があります。

次のように、FORALL キーワードを使用してコレクションをバルク・バインドし、パフォーマンスを改善します。

```
DECLARE
  TYPE Numlist IS VARRAY (100) OF NUMBER;
  Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN
  FORALL i IN Id.FIRST..Id.LAST -- bulk-bind the VARRAY
    UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);
END;
```

コレクションを参照する SELECT 文 バルク・バインドは、コレクションを参照する SELECT 文のパフォーマンスを改善するために使用できます。出力収集を PL/SQL エンジンに戻す前にバルク・バインドするには、キーワード BULK COLLECT INTO を使用します。

たとえば、次の PL/SQL ブロックでは、バルク・バインドを使用しないで、管理者の ID 番号が 7698 の従業員の名前および職種を戻します。

```
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

    -- Initialize variable tracing number of employees returned.

    counter := 1;

    -- Find all employees whose manager's ID number is 7698.

    FOR rec IN C LOOP
        Empno(counter) := rec.Empno;
        Ename(counter) := rec.Ename;
        Counter := Counter + 1;
    END LOOP;
END;
```

選択された各従業員の SQL エンジンに対して、PL/SQL が SQL 文を送信します。選択される従業員数が多い場合は、PL/SQL エンジンと SQL エンジンとの間でコンテキストの切替えが多数発生し、パフォーマンスが低下する可能性があります。

次のように、BULK COLLECT INTO キーワードを使用してコレクションをバルク・バインドし、パフォーマンスを改善します。

```
DECLARE
    TYPE Emplist IS VARRAY(100) OF NUMBER;
    Empids EMLIST := EMLIST(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_inst;
END;
```

コレクションおよび RETURNING INTO 句を参照する FOR ループ バルク・バインドは、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するために使用できます。これを行う PL/SQL コードがある場合、またはこのようなコードを計画している場合は、FORALL キーワードおよび BULK COLLECT INTO キーワードを一緒に使用して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 2 つのアクションがバルク・バインドを使用しないで実行されています。

```

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);
    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;
```

更新される各従業員の SQL エンジンに対して、PL/SQL が SQL 文を送信します。更新される従業員数が多い場合は、PL/SQL エンジンと SQL エンジンとの間でコンテキストの切替えが多数発生し、パフォーマンスが低下する可能性があります。

次のように、FORALL キーワードおよび BULK COLLECT INTO キーワードを一緒に使用してコレクションをバルク・バインドし、パフォーマンスを改善します。

```

DECLARE
    TYPE Emplist IS VARRAY(100) OF NUMBER;
    TYPE Numlist IS TABLE OF Emp_tab.Sal%TYPE;
    Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
    Bonlist NUMLIST;
BEGIN
    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;
```

```
END;
```

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（データベースまたはスキーマに対するトリガー）も定義できます。

参照： [第 12 章「トリガーの使用」](#) を参照してください。

PL/SQL コードのラッピング

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、『Oracle8i 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。

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。

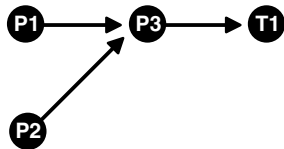
- [タイムスタンプ](#)
- [シグネチャ](#)

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限り実行できません。

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くなくなります。

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 Server 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・モデルの不利な点

この依存性モデルの不利な点は、必要以上に制限的であるということです。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く、この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まったく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 Tools 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・モデルは問題を起こす可能性があります。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限り、アプリケーションはインストール中に無効であると解釈されます。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、しかも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まったく実行できません。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： 9-35 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方法に矛盾があった場合は、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 と呼ばれる DB リンクを使用して Get_emp_name を識別して、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、次のアクションが行われます。

- 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ために、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限り、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・モードでは表示されません。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 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。各データ型クラス内には、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。

表 9-3 に、データ型のクラスを示します。

表 9-3 データ型

VARCHAR 型	数値型
VARCHAR2	NUMBER
VARCHAR	INTEGER
STRING	INT
LONG	SMALLINT
ROWID	DECIMAL
文字型	DEC
CHARACTER	REAL
CHAR	FLOAT
RAW 型	NUMERIC
RAW	DOUBLE PRECISION
LONG RAW	NUMERIC

表 9-3 データ型（続き）

整数型	日付型
BINARY_INTEGER	DATE
PLS_INTEGER	
BOOLEAN	
NATURAL	
POSITIVE	
POSITIVEN	
NATURALN	

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、次の指定があるとします。

```
PROCEDURE P1 (Param1 NUMBER);
```

これを、次のように変更します。

```
PROCEDURE P1 (Param1 IN NUMBER);
```

ただし、これによってシグネチャは変更されません。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。

シグネチャの例

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 のままの場合は、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、次のパッケージ指定は、前のパッケージ指定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、動的初期化パラメータ REMOTE_DEPENDENCIES_MODE 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。次に例を示します。

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =  
    {SIGNATURE | TIMESTAMP}
```

この例では、現行セッションの依存性モデルが変更されます。

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =  
    {SIGNATURE | TIMESTAMP}
```

この例では、起動後にシステム全体で依存性モデルが変更されます。

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、ALTER SESSION 文または ALTER SYSTEM DDL 文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 文を使用しない限り、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 を使用するときには、次の点に注意する必要があります。

- 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すると、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すると、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。

コールされたりリモート・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、コールされたりリモート・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、コールされたりリモート・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、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（9-32 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勧めします。

- 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（あるいは `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起こさずに、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必要以上に無効にされ、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にパラメータとして渡したり、戻したりできます。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、『Oracle8i 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。

- 『Oracle8i Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』
- 『Oracle8i Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』
- 『Oracle8i コール・インタフェース・プログラマーズ・ガイド』
- 『Programmer's Guide to SQL*Module for Ada』

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;
```

次の例は、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ています。

```
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;
```

コンパイル時エラー

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: 警告 : プロシージャ proc1 は作成されました。コンパイル・エラーです。
```

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、『Oracle8i 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 CHARWIDTH 文を使用してください。通常は、次のように値を 132 に指定することをお勧めします。次に例を示します。

```
SET CHARWIDTH 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 WHERE Empno = Emp_id;
END
/
```

前述の CREATE PROCEDURE 文にはエラーが 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 の「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 Tool 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・コードを取り出せます。

参照： これらのデータ・ディクショナリ・ビューの詳細は、『Oracle8i リファレンス・マニュアル』を参照してください。

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限り）。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（それよりも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。詳細は、これらのパッケージの説明を参照してください。

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、9-46 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 文の使用時に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。

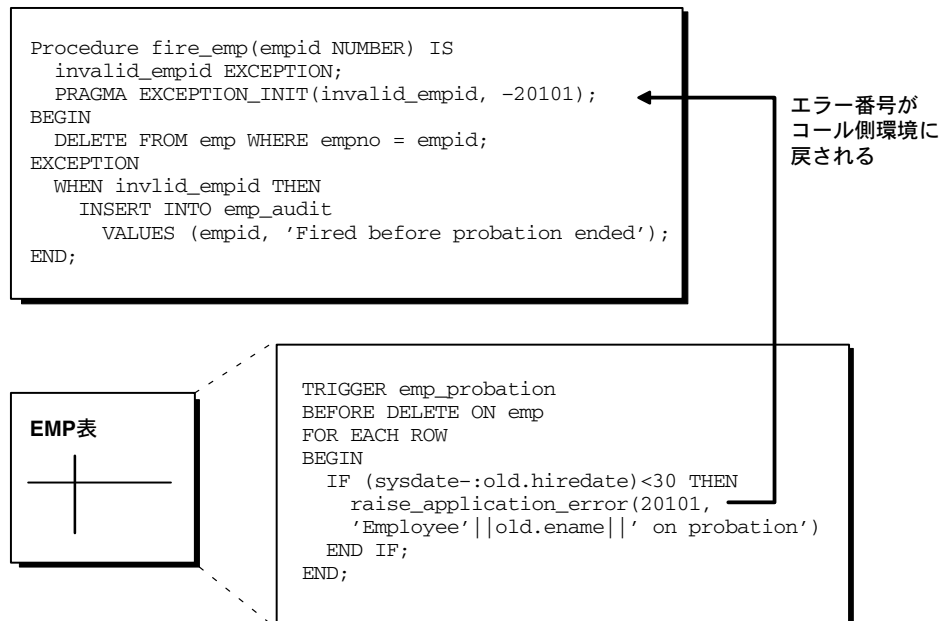
例外ハンドラは、ユーザー定義エラー・メッセージを処理するために定義することもできます。たとえば、9-45 ページの [図 9-2](#) では、次のことが示されています。

- 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 までに限り実行されます。

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、プリAGMA 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

PL/SQL アプリケーションのデバッグ用に、オラクル社では Java ベースのデバッガを無料で提供しています。これはコンパクトで使用しやすいデバッガで、Microsoft または Netscape のブラウザからアプレットとして実行するか、またはスタンドアロン・アプリケーションとして PL/SQL ストアド・プロシージャのデバッグ用に実行できます。このデバッガは、Oracle 7.3.4 以降で動作します。

Application Server を使用する Web ベースのアプリケーションの作成に PL/SQL を使用し、デバッグの必要があって生成された HTML を表示する場合も、このデバッガを使用することができます (README ファイルにある制約を参照)。

Oracle8i で提供されている DBMS_DEBUG API 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。

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。

参照：『Oracle Procedure Builder Developer's Guide』を参照してください。

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。

参照： DBMS_DEBUG および DBMS_OUTPUT パッケージの詳細は、『Oracle8i PL/SQL 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。

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 Tool を使用して、ユーザーが対話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 関数のコールに類似した方法で 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： 9-56 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、プロシージャの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 Tools 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 などの Oracle Tool 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 Tools 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 Tools 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
```

いったん定義したセッション変数は、そのセッション中に限り有効です。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、『Oracle8i 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 Tool 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(:Empnum);
```

この場合、:Empnum 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。

『Oracle8i コール・インタフェース・プログラマーズ・ガイド』

『Oracle8i Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』

『Programmer's Guide to SQL*Module for Ada』

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 文における参照オブジェクトの名前変換](#)」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 システム権限を、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わせて指定します。

- 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 つの方法を組み合わせる引数の値を指定します。引数の名前と順序を組み合わせる指定する場合は、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（『Oracle8i 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、データベース内にあり、BOSTON_SERVER というローカル・データベース・リンクが示すプロシージャ 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、9-46 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・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の Oracle Tool 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パー・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 がこのリストの最初の項目として定義されます。

参照： 9-56 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： コンパイル時に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。このためこの種の参照では、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・プロシージャが読取り専用の場合でも）。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・プロシージャと同じ 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非 Oracle 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ったと、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 以降が必要です）。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、ユーザーの生産性を向上します。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（たとえば、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 文内でネーミングする前に、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（たとえば、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。

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;
```

命名規則

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 パッケージをチェックします。

- 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;
```

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、カッコを省略します。

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  
...
```

プロシージャ型の文から 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(Enum, 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限り、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。

基本的な要件の充足

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 Server 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 Server 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、問合せの平行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てパッケージ状態をメンテナンスする必要が発生したりします。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。Oracle8i では、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： 9-62 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チェックしなくても、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・エラーを提示し、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、プラグマはまったく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： 9-66 ページの「PRAGMA RESTRICT_REFERENCES の使用」を参照してください。

制限事項

SQL 文が実行される時、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チェックされます。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、このチェックが行われます。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、さらにチェックが行われます。

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み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、新機能のシステム固有の動的メカニズム（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 Oracle8i 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まったく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何であっても、必ずまったく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、他にアクションを何も実行しなくても、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。

Oracle8i で提供されている次の 2 つの新機能では、機能とともに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ています。

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勧めします。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、データベースの読み込みも、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）を今まで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。オラクル社としては、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 / 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 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： 9-66 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み込みでもなく書き込みでもない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。

Oracle8i でも、Oracle7 および Oracle8 でパラレル化可能として認識されるファンクションは、引き続きパラレル化されます。さらに、新しいキーワード PARALLEL_ENABLE が追加されています。ユーザーがコードを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は、こちらをお薦めします。このキーワードは、前述の DETERMINISTIC と構文的に非常に似ています。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みも書きも行わず、パッケージ変数の読みまたは書き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限り、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： 9-62 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。

プラグマ 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。

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わ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みおよび書きは行われないため、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 Pl1 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 Pl1 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';
```

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しか適用できません。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;
```

逐次再使用可能 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。

逐次再使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使用されます。これは、このようなパッケージのグローバル・メモリーは、作業単位内ではしか使用されないことを意味します。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使用されます。

逐次再使用可能パッケージの作業単位とは、たとえば、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

再使用不能パッケージ（`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。

逐次再使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しか持続しません。サーバーへの後続のコールでは、逐次再使用パッケージが参照される場合、逐次再使用パッケージの新しいインスタンス（後述）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使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使用可能パッケージの新しいインスタンス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限りません。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すのみです。

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、作業域を再使用できるようにするためです。

逐次再使用可能パッケージを使用する理由

再使用不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。

逐次再使用可能パッケージ（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。

構文

パッケージは、プラグマ SERIALLY_REUSABLE により逐次再使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、逐次再使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使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。逐次再使用可能プラグマは、仕様部にそのプラグマがない限り、本体に含めることはできません。

セマンティクス

SERIALLY_REUSEABLE 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）の中でしか使用できません。

注意： アプリケーション・プログラマが、間違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・インスタンス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スエーションの 1 つが次のように再使用されます。
 - 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。
 - 再使用不能な 2 次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・インスタンスエーションは、このパッケージ用に保持された再使用可能インスタンス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使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使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内にある逐次再使用可能パッケージパッケージ <パッケージ名> には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。

例

例 1 次に、逐次再使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）。これは、パッケージ変数がコール境界を超えてどのように機能するかを示しています。

```
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;
```

このプログラムは、次のような結果を戻します。

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 次に、逐次再使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。[例 1](#) と同様に、この例ではパッケージ変数が、全境界にまたがりどのように機能するかを示しています。

```
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 この例は、逐次再使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
```


複数の言語を扱う必要性

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、『Oracle8i 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**
Oracle コール・インタフェース（OCI）を使用します。詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。
- **C++**
Pro*C/C++ プリコンパイラを使用します。詳細は、『Oracle8i Pro*C/C++ プリコンパイラ・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、『Oracle8i Pro*COBOL プリコンパイラ・プログラマーズ・ガイド』を参照してください。
- **Visual Basic**
Oracle Objects For OLE（OO4O）を使用します。詳細は、『Oracle Objects for OLE/ActiveX Programmer's Guide』を参照してください。
- **Java**
JDBC アプリケーション・プログラム・インタフェースを使用します。詳細は、『Oracle8 Database Programming with Java』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、それぞれ異なる利点があります。使用しやすさ、特定の専門知識を持つプログラマーがいるかどうか、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 ORDBMS 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、サーバー上で実行されますが、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ことが必要にな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ーを利用するという方法も可能になります。

Oracle 8.0 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。Oracle 8.0 になって、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 および (PL/SQL 経由で) SQL からコールできます。Oracle 8.1 では、コール仕様という特殊な目的の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、他の言語から外部ルーチンをコールできます。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ルーチンでも、C からコール可能なルーチンであれば、SQL または PL/SQL からでも使用できます。したがって、使用する C++ ルーチンがある場合は、そのルーチンで C++ の `extern` 文を使用してそのルーチンを 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。外部ルーチンの使用によって、特定の目的に特定の言語を実行できるため、再使用性およびモジュール性が向上します。

外部ルーチンとは

外部ルーチンは、以前は外部プロシージャと呼ばれていたもので、動的リンク・ライブラリ (DLL) に格納されているルーチンです。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。このルーチンを基本言語に登録し、これをコールして特定目的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 がライブラリを実行時に動的にロードし、次に、ルーチンを PL/SQL サブプログラムであるかのようにコールします。このようなルーチンは、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。

このようなルーチン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ということは、コール側プログラムに影響することなく、ルーチンを拡張できるということです。

外部ルーチンを使用すると、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（ネットワークにまたがる通信に伴う往復通信を回避するため）。
- 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をとれます。
- データベース・サーバー自体の機能を拡張できます。

コール仕様

今まで、外部ルーチンは PL/SQL ラッパー内の AS EXTERNAL 句を介して Oracle に発行してきました。このラッパーが、外部 C ルーチンへのマッピングを定義し、このルーチンをコールできるようにしていました。Oracle 8.1 ではコール仕様が導入されていますが、このコール仕様には、AS EXTERNAL ラッパーが新しい AS LANGUAGE 句のサブセットとして含まれています。AS LANGUAGE コール仕様を使用すると、今までのように外部 C ルーチンを発行できる他に、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、Oracle 8.0 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・ルーチンのディスパッチ
- 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な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 ルーチンのコール

- 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ルーチン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、次にコールします。

外部ルーチンのロード

外部 C ルーチンまたは Java メソッドを PL/SQL で使用できるようにするには、ルーチンまたはメソッドをまずロードする必要があります。ロード方法は、ルーチンが 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：『Oracle8i Java ストアド・プロシージャ開発者ガイド』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 仮想マシン (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 (クラス・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 OS 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、ステップが 2 つ必要になります。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、コマンド行・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、OS 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：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

外部 C ルーチンのロード

C で作成された外部ルーチンまたは C からコール可能な外部ルーチンを使用できるようにセットアップするには、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. 環境をセットアップする

DBA は、ファイル tnsname.ora および listener.ora にエントリを追加し、外部ルーチン専用のリスナー・プロセスを開始することによって、外部ルーチンをコールする環境をセットアップします。

参照：『Oracle8i 管理者ガイド』を参照してください。

リスナーは、extproc 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。それ以外の場合は、extproc に対してクリーンな環境を提供します。extproc 用に設定される環境変数は、クライアント、サーバーおよびリスナ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、extproc プロセス内で実行される外部ルーチンは、クライアント、サーバーまたはリスナ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 ルーチンの `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、`extproc` プロセス専用です。つまり、そのプロセス内で実行されるすべての関数で読み込みますが、同一マシン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、外部ルーチン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。DBA は、このかわりに、ユーザーに `CREATE ANY LIBRARY` 権限を付与することもあり、この場合、ユーザーが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 library_name {IS | AS} 'file_path';
```

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';
```

3. 外部ルーチンを指定する

新しい外部 C ルーチンを検索または作成してからこれを DLL に追加するか、またはすでに DLL 内に存在するルーチンを指定します。

外部 C ルーチンが DLL の中にロードされます。外部ルーチンを作成して DLL 内に組み込んだ後、次のように、DLL を表す別名ライブラリを作成します。

```
CREATE LIBRARY C_utils AS '/DLLs/utils.so';
```

外部ルーチンの発行

Oracle で使用できる外部ルーチンは、発行済外部ルーチンのみです。発行するにはコール仕様を使用します。Java クラス・メソッドまたは C 外部ルーチンの名前、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じように作成しますが、`AS LANGUAGE` 句は宣言および `BEGIN..END` ブロックではなく、本体内に作成します。

コール仕様は、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 構文に続いて指定します。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 がこの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 ルーチンにまで拡張されています。

『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

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 ルーチンの名前です。*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・メソッドとの間のインタフェースです。

参照：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

外部 C ルーチン用の AS LANGUAGE 句

次の副次句は、PL/SQL に対して、外部 C ルーチンを検索する場所、ルーチンのコール方法、ルーチン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・リンクは使用できません）。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、名前の大文字 / 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 ルーチンを指定します。ルーチン名を二重引用符で囲むと、名前の大文字 / 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、ルーチン名は 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

外部ルーチンが作成されている第 3 世代言語を指定します。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

外部ルーチン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り出す必要があります）。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・ポインタが外部ルーチン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ルーチンに対して不透明ですが、外部ルーチンによってコールされるサービス・ルーチンでは使用できます。

PARAMETERS

外部ルーチン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。

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 CREATEJAVA SQL 文を使用して、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 に似ていますが、DLL には複数のルーチン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：『Oracle8i Java ストアド・プロシージャ開発者ガイド』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 内にロードするとき、クラスは SQL に対して自動的に発行されません。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、引数の階乗を返す 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 スタンド・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 ルーチンの発行

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 ルーチンはともに、次の位置のいずれか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 ファンクション内でのコール仕様の例はすでに説明済です。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の詳細は、このマニュアルの「実行者権限」の項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 SOME LIB 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容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 ルーチンを 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;
```

コール仕様を使用した Java クラス・メソッドへのパラメータ渡し

参照：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

コール仕様を使用した外部 C ルーチンへのパラメータ渡し

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 ルーチンへのパラメータ渡しは、次のような状況のときには複雑になります。

- 使用可能な PL/SQL 集合が、C データ型の集合と 1 対 1 で対応していない場合。

- C とは異なり、PL/SQL には NULL かどうかという RDBMS 概念が含まれています。したがって、PL/SQL 文は NULL にできますが、C パラメータは NULL にできません。
- 外部ルーチンで、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になる場合。
- 外部ルーチンで、CHAR、VARCHAR2 および CLOB パラメータに関するキャラクタ・セット情報が必要になる場合。
- PL/SQL で、外部ルーチンによって戻された値の現在の長さ、最大長または NULL 状態が必要になる場合。

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 外部ルーチンに渡せ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によって渡す場合は、最大数は 128 より少なくなります。どのくらい少なくなるかは、そのような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、値によって渡される 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

パラメータは外部ルーチンに直接渡さないでください。かわりに、外部ルーチンを発行した 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	[UNSIGNED] CHAR	INT
BOOLEAN	[UNSIGNED] SHORT	
PLS_INTEGER	[UNSIGNED] INT	
	[UNSIGNED] LONG	
	SB1, SB2, SB4	
	UB1, UB2, UB4	UNSIGNED INT
	SIZE_T	
NATURAL	[UNSIGNED] CHAR	
NATURALN	[UNSIGNED] SHORT	
POSITIVE	[UNSIGNED] INT	
POSITIVEN	[UNSIGNED] LONG	
SIGNTYPE	SB1, SB2, SB4	
	UB1, UB2, UB4	
	SIZE_T	
FLOAT	FLOAT	FLOAT
REAL		
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR	STRING	STRING
CHARACTER	OCISTRING	
LONG		
NCHAR		
NVARCHAR2		
ROWID		
VARCHAR		
VARCHAR2		
LONG RAW	RAW	RAW
RAW	OCIRAW	
BFILE	OCILOBLOCATOR	OCILOBLOCATOR
BLOB		
CLOB		
NCLOB		

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
NUMBER	[UNSIGNED] CHAR	OCINUMBER
DEC	[UNSIGNED] SHORT	
DECIMAL	[UNSIGNED] INT	
INT	[UNSIGNED] LONG	
INTEGER	SB1, SB2, SB4	
NUMERIC	UB1, UB2, UB4	
SMALLINT	SIZE_T OCINUMBER	
DATE	OCIDATE	OCIDATE
コンポジット・ オブジェクト型 : ADTs	dvoid	dvoid
コンポジット・ オブジェクト型 : コレクション (VARARRAYS、 NESTED TABLE、索引付き表)	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 PLSQL 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

外部データ型	C プロトタイプで使用するデータ型		
	IN、RETURN	参照による IN、参照による RETURN	IN OUT、OUT
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 **

表 10-2 外部データ型のマッピング（続き）

外部データ型	C プロトタイプで使用されるデータ型		
	IN、RETURN	参照による IN、参照による RETURN	IN OUT、OUT
OCINUMBER	OCINumber *	OCINumber *	OCINumber *
OCISTRING	OCIString *	OCIString *	OCIString *
OCIRAW	OCIRaw *	OCIRaw *	OCIRaw *
OCIDATE	OCIDate *	OCIDate *	OCIDate *
OCICOLL	OCIColl * or OCIArray *, or OCITable *	OCIColl ** or OCIArray **, or OCITable **	OCIColl ** or OCIArray **, or OCITable **
OCITYPE	OCIType *	OCIType *	OCIType *
TDO	OCIType *	OCIType *	OCIType *
ADT	dvoid*	dvoid*	dvoid*

コンポジット・オブジェクト型は、自己記述型ではありません。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標識構造体には事前定義の 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 の OCICOLL およびコレクション引数はオプションで、完全を期す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 引数が値によって渡されます（デフォルト）。このかわりに BY REFERENCE を指定すると、参照によって渡すことができます。

デフォルトでは、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 での BY VALUE の指定および OUT 引数は C 用にはサポートされていません。BY REFERENCE/BY VALUE 句の使用は、デフォルトで値によって渡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 引数および OUT 引数はすべて参照によって渡されます。

PARAMETERS 句

一般に、外部ルーチンを発行する 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、外部ルーチン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 標識
- 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。
- 外部ルーチン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。

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 をマッピングし直すことができます。

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ルーチン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	C パラメータ		PL/SQL パラメータ		
	使用可能外部 データ型	デフォルトの外部 データ型	使用可能データ型	使用可能モード	デフォルトの 受渡し方法
INDICATOR	SHORT	SHORT	すべてのスカラー	IN	BY VALUE
	INT			IN OUT	BY REFERENCE
	LONG			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 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,  
        x INDICATOR, -- stores null status of x  
        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、標識は必要ありません。ただし、外部ルーチンでは、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、外部ルーチンでは、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、標識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、前述のように標識をファンクション結果に対応付けることもできます。

標識の値を調べるには、定数 OCI_IND_NULL および OCI_IND_NOTNULL を使用できます。標識が OCI_IND_NULL と同等の場合は、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。標識が OCI_IND_NOTNULL と同等の場合は、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。

IN パラメータは読取り専用ですが、この場合は INDICATOR は (BY REFERENCE を指定しない限り) 値によって渡され、(BY REFERENCE を指定しても) 読取り専用です。OUT、IN OUT および RETURN パラメータの場合は、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標識構造体があるため、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ルーチンとの間で行うことが必要な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 を指定しない限り）値によって渡され、読取り専用です。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り渡されます。

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、読取り専用です。

CHARSETID および CHARSETFORM

Oracle では各国語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。

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 および形式を外部ルーチンに渡すことができます。

IN パラメータの場合は、CHARSETID および CHARSETFORM は（BY REFERENCE を指定しない限り）値によって渡され、（BY REFERENCE を指定しても）読取り専用です。OUT、IN OUT および RETURN パラメータの場合は、CHARSETID および CHARSETFORM は参照によって渡され、読取り専用です。

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 NLS データを使用する場合の詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』および『Oracle8i NLS ガイド』を参照してください。

パラメータの再配置

外部ルーチンの仮パラメータは、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 句および外部ルーチンのための 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す。つまり、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から構成される 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 の年齢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組込み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。

ここで、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り出します。

```
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、オブジェクト型トランスレータ（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
{
    OCIStrng    *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・パラメータは値によって渡す（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ルーチンで必要な場合は、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、パラメータ、例外、メモリ割当ておよびユーザー環境に関する情報が外部ルーチンでアクセスできるようになります。WITH CONTEXT 句は、コンテキスト・ポインタが外部ルーチン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ルーチンに対して不透明ですが、外部ルーチンによってコールされるサービス・ルーチンでは使用できます。

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・ポインタは外部ルーチン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、値を戻すルーチンの RETURN 句もサポートします。

参照：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

PL/SQL および C パラメータ・モードの規則が列記されています。

外部ルーチンの実行 : CALL 文

これで Java クラス・メソッドまたは外部 C ルーチンが発行されたので、次はこれを起動します。

外部ルーチンは直接コールしてはいけません。かわりに、外部ルーチンを発行した 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普通どおり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

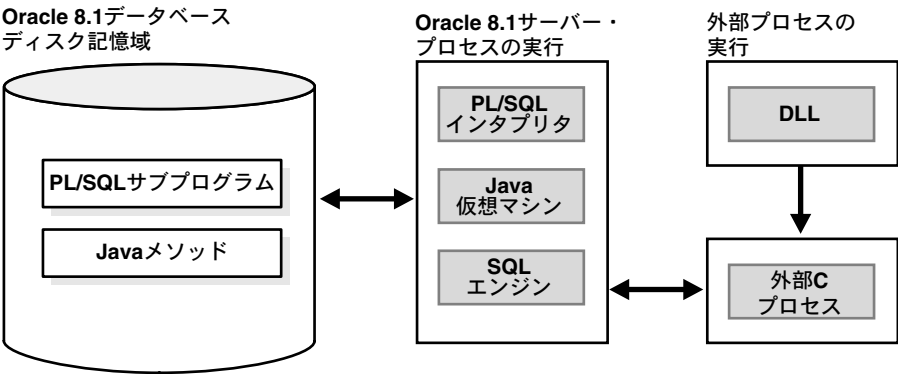
■ SQL 文（パッケージ関数のみへのコール）

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、プラグマ 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の制限を受けない度合い）を確認します。PL/SQL は、対応する外部ルーチンの純粋度レベルはチェックできません。したがって、外部ルーチン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、Oracle8 と外部ルーチンのやりとりを示します。

図 10-1 Oracle8 と外部ルーチン



準備

外部ルーチンをコールする前に、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

外部ルーチンがコール仕様経由でコールされるとき、ルーチン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束縛されません。このプログラムは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。しかし、定義者権限プログラムは、プログラムが定義されているスキーマに束縛されます。このプログラムは定義を行う側の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');
```

外部ルーチンをコールするには、ユーザーはコール仕様およびルーチン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 :

- 『Oracle8i SQL リファレンス』を参照してください。
 - 『Oracle8i Java ストアド・プロシージャ開発者ガイド』も参照してください。
-
-

シノニムの作成

開発者または DBA は、便宜上、CREATE [PUBLIC] SYNONYM 文を使用して外部ルーチン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 を指定しないと、シノニムはプライベートになり、そのスキーマ内でのみアクセスできません。

```
CREATE PUBLIC SYNONYM Rfac FOR Scott.RecursiveFactorial;
```

CALL 文の構文

外部ルーチンは、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}]routine_name[@dblink_name]
    [(parameter_list)] [INTO :host_variable] [INDICATOR] [:indicator_variable];
```

これは、SELECT foo(...) FROM dual という形式のルーチン foo() の実行と本質的には同じです。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 ルーチン C_demoExternal_proc にパラメータを 3 つ渡します。

```
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 ブロックとまったく同じです。

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 class 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：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

外部 C ルーチンのコール

外部 C ルーチンをコールするには、どの DLL にこのルーチンが含まれているかを PL/SQL で認識する必要があります。PL/SQL エンジンには、これを行うために、AS LANGUAGE 句内に記述されているライブラリ別名のデータ・ディクショナリ内を検索します。このライブラリ内に含まれている DLL に対応付けられているファイル名が検索されます。

次に、PL/SQL はリスナー・プロセスに警告を発行し、リスナー・プロセスが、extproc というセッション固有のエージェントを起動します。リスナーは接続を extproc に渡し、PL/SQL は DLL の名前、外部ルーチンの名前およびパラメータを extproc に渡します。

次に、extproc は DLL をロードし、外部ルーチンを実行します。また、extproc は（例外を呼び出すなどの）サービス・コールおよび Oracle Server へのコールバックも処理します。最後に、extproc は、外部ルーチン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。

外部ルーチンが完了した後、extproc 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 extproc が消去されます。この結果、何度コールしても、extproc を起動するのは 1 回で済みます。それでも、演算による利点がコールのコストを上回るときにのみ外部ルーチンをコールするようにします。

注意： リスナーは、tnsnames.ora および listener.ora ファイル内の情報を使用して、Oracle Server が動作するマシン上で extproc を起動する必要があります。別のマシン上での extproc の起動は、サポートされていません。

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 ...
```

エラーおよび例外

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：『Oracle8i Java ストアド・プロシージャ開発者ガイド』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せます。

外部 C ルーチンでのサービス・ルーチンの使用

サービス・ルーチンが外部ルーチンからコールされると、サービス・ルーチンは例外を呼び出し、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ルーチン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します。

```
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、外部ルーチン・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： 外部ルーチンでは、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（コールしてはいけません）。

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返します。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、結果を返します。

```
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 = OCIEExtProcAllocCallMemory(ctx, 1);
    tmp[0] = '\0';
    return(tmp);
}
/* Allocate memory for result string, including NULL terminator. */
len = strlen(str1) + strlen(str2);
tmp = OCIEExtProcAllocCallMemory(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)
{
    OCIEExtProcContext *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;
    OCISstmt      *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_RDEMS,
(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。必要なクリーン・アップを実行後、外部ルーチンは即座に戻る必要があります（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、外部ルーチン 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` と `OCIEXTPROC_ERROR` は、正常終了および失敗を示します。

この前の例では、外部ルーチン `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 ルーチン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、外部ルーチン・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、外部ルーチン・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 ルーチンおよび 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。

Java の例外：

参照： 詳細は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』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限りません。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 Server 上で実行される外部 C ルーチンは、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 文と 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。さらに、コールバックおよび外部ルーチンは、同一ユーザー・セッション内の同一トランザクション・コンテキスト内で動作するため、同一のユーザー権限を持ちます。

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、次のように外部ルーチン `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

これで、外部ルーチン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()` ルーチンを使用して取り出します。

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ってい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、まず、`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。次に、`OCIObjectGetAttr()` および `OCIObjectSetAttr()` をコールして、属性値を取り出して設定します。

現在の外部ルーチン・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ルーチン内では、`OCIExtProcGetEnv()` をコールする必要があります。サービス・ルーチン外部ルーチン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 ルーチン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 ルーチン

```
OCIObjectNew  
OCIObjectPin  
OCIObjectUnpin  
OCIObjectPinCountReset  
OCIObjectLock  
OCIObjectMarkUpdate  
OCIObjectUnmark  
OCIObjectUnmarkByRef  
OCIObjectAlwaysLatest  
OCIObjectNotAlwaysLatest  
OCIObjectMarkDeleteByRef  
OCIObjectMarkDelete  
OCIObjectFlush  
OCIObjectFlushRefresh  
OCIObjectGetTypeRef  
OCIObjectGetObjectRef  
OCIObjectExists  
OCIObjectIsLocked  
OCIObjectIsDirty  
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 ルーチン
- 次の OCI ルーチン

```

OCIEnvInit
OCIInitialize
OCIPasswordChange
OCIServerAttach
OCIServerDetach
OCISessionBegin
OCISessionEnd
OCISvcCtxToLda
OCITransCommit
OCITransDetach
OCITransRollback
OCITransStart

```

また、OCI ルーチン 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

```

外部ルーチンのデバッグ

参照：『Oracle8i Java ストアド・プロシージャ開発者ガイド』を参照してください。

外部ルーチンが失敗する場合は、通常、そのプロトタイプに欠陥があります。つまり、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、外部ルーチンによってコア・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、外部ルーチン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` にある指示に従ってください。Oracle アカウトに、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接続できるデバッグ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、外部ルーチン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、外部ルーチン用の 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 アカウトを使用する必要があり、このアカウトには `CREATE LIBRARY` 権限が必要です。

外部 C ルーチンのためのガイドライン

グローバル変数および静的変数の操作

グローバル変数 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。外部ルーチンの場合、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、`extproc` 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、`extproc` 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**: 関数の存続期間を越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 はアンロードされますが、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 静的変数には外部変数および内部変数という2つの種類があります。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出入りするのではなく、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。

コール仕様およびコールのガイドライン

外部ルーチン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、外部ルーチン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、外部ルーチンが関数の場合は、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。暗黙的変換は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 ルーチンに関する制限事項

現在、外部ルーチン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 ルーチンまたは C からコール可能なルーチンのみがサポートされます。
- PL/SQL カーソル変数、レコード、コレクションまたはオブジェクト型のインスタンスを外部ルーチンに渡すことはできません。
- 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。
- リスナーは、Oracle Server が動作するマシン上でエージェント extproc を起動する必要があります。別のマシン上での extproc の起動は、サポートされていません。
- 外部ルーチンに渡せ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によって渡す場合は、最大数は 128 より少なくなります。どのくらい少なくなるかは、そのようなパラメータの個数とオペレーティング・システムによって異なります。目安としては、値によって渡される 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。

セキュリティ・ポリシーの設定

この章では、セキュリティ・ポリシーを設定する手順を説明します。内容は次のとおりです。

- 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
- ファイングレイン・アクセス・コントロール
- アプリケーション・コンテキスト
- 中間層を介する認証
- データの暗号化

セキュリティ・ポリシーの概要

この項では、任意のセキュリティ・ポリシーの概要について説明します。内容は次のとおりです。

- [セキュリティの侵害および対策](#)
- [任意のセキュリティ・ポリシーで扱える項目](#)
- [セキュリティ・ポリシーの設定に使用する機能](#)

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（経歴のチェック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しい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わ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、悪意を持った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、錠を掛け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チェックすることです。

Oracle8i 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。

任意のセキュリティ・ポリシーで扱える項目

ご使用の環境に固有の要件以外に、次のような技術的問題を判断するための任意のセキュリティ・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
- 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
- アクセス制御の細かさのレベル
- 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

この章では、セキュリティ・ポリシーの設定に使用できるいくつかの Oracle8i の要素について説明します。

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追加し、ユーザーが実際にアプリケーションを使用していないときにこれらの権限およびロールが誤って使用されないようにします。
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、そのユーザーがアクセスを許可されている帳簿類、管理職階層におけるそのユーザーの職位などのユーザー属性を確実に格納できます。こうすることによって、その情報を後でセッション内で取り出すことができます。
ファイングレイン・アクセス・コントロール	セキュリティ・ポリシーを詳細レベルで実装します。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、アプリケーションの基になる表またはビューに連結されるセキュリティ・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 はその文を動的およびユーザーに透過的に変更します。
中間層での認証および監査	個別のデータベース接続のオーバーヘッドなしで、各ユーザーがデータベース・パスワードによって認証されるようにします。また、この機能を使用して、中間層を介してデータベースに実際にアクセスしているユーザーを識別できます。
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。

アプリケーション・セキュリティ

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、複数のアプリケーション・ロールが必要です。アプリケーション・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することができます。

(SQL*Plus などツールを使用して) 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作業](#)
- [ロールおよびアプリケーション権限の管理の概要](#)
- [ユーザーの現在のアプリケーション・ロールへの権限の対応付け](#)
- [ツール・ユーザーからのアプリケーション・ロールの制限](#)
- [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](#)
- [ロールの作成およびロール使用の保護](#)
- [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で施行されるか、データベースで施行されるか

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勧めします（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、データベースの本来の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。

- 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 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（Kerberos、DCE、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。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。

セキュリティがアプリケーションで施行されるか、データベースで施行されるか

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、別のアプリケーションを使用してデータにアクセスしても、セキュリティを回避することができません。

データベース・ユーザーであるユーザーを持つ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作成することも、本来のデータベース・セキュリティ・メカニズム（詳細な権限、仮想プライベート・データベース（アプリケーション・コンテキストを持つファイイングレイン・アクセス・コントロール）、ロール、ストアド・プロシージャ、監査など）に依存することもできます。

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。

アプリケーションがそのセキュリティ・メカニズムを施行する場合、データにアクセスする各アプリケーションは、セキュリティを再実装する必要があります。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、セキュリティを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。

アプリケーション・ベースのセキュリティの大きな欠点は、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。

アプリケーション管理者の作業

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の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アプリケーション・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

ロールおよびアプリケーション権限の管理の概要

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。アプリケーション権限の管理を簡素化するには、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なすべての権限を付与します。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。その場合、次の手順に従います。

1. VACATION ロールを作成します。
2. VACATION ロールに対して、Vacation アプリケーションに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。

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、権限を管理する上で便利です。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、社員の職務が変わったときは、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・ビューを問い合わせることによって、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・ビューを問い合わせることによって、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。

ユーザーの現在のアプリケーション・ロールへの権限の対応付け

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、ユーザーには、現在実行中のアプリケーション・ロールに関連する権限のみを許可する必要があります。たとえば、次の使用例を考えてみます。

- `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。

SET ROLE 文

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。ユーザーがロールからすべての権限を取得しても、それらを組み合わせて無許可の操作を実行することはできません。

参照： 11-24 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

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。その理由は、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チェックするためです。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、所有者に直接付与された）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チェックするとき、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チェックしません。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チェックする場合は、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）

前述の両方の場合で、データベースは、コンパイル時ではなく実行時に権限をチェックします。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）。

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。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 から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 IS
    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： 8-12 ページの「システム固有の動的 SQL と DBMS_SQL パッケージの対比」を参照してください。

ツール・ユーザーからのアプリケーション・ロールの制限

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）。これには、セキュリティ上重大な問題があります。

この項の内容は次のとおりです。

- [潜在的な問題](#)
- [PRODUCT_USER_PROFILE を介するロールの制限](#)
- [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](#)
- [リスク削減のための推奨アプリケーション設計の実行](#)
- [ロールの即時使用可能および使用禁止](#)
- [ストアド・プロシージャでの権限のカプセル化](#)
- [ユーザーが知らないロール・パスワードの使用](#)
- [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](#)

潜在的な問題

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付与されている権限を使用し、非定型ツールを使用して、データベース表に対して破壊的な 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（文をいつ発行するかはアプリケーションが制御します）。

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。

PRODUCT_USER_PROFILE を介するロールの制限

Oracle8i では、PRODUCT_USER_PROFILE 表を介して、ユーザーがアプリケーションで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。SQL*Plus（Oracle ではなく）は、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、『Oracle8i 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。

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

Oracle8i では、サーバー施行のファイグレイン・アクセス・コントロールとアプリケーション・コンテキストの組合せである仮想プライベート・データベースを使用して、ユーザーは、セキュリティ（詳細レベルまで）を表またはビューに直接施行できます。セキュリティ・ポリシーは、表またはビュー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。

集中的に管理され、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、ユーザーがデータにどのようにアクセスするか（アプリケーション、問合せ、レポート作成ツールなどによって）に関係なく、セキュリティが施行されます。

仮想プライベート・データベースを使用すると、セキュリティ・ポリシーを対応付けられた表またはビューに直接的または間接的にユーザーがアクセスすることによって、セキュリティ・ポリシーを実装する関数が戻す WHERE 条件（述語）に基づいた文を、サーバーが動的に変更します。ユーザーの SQL 文は、関数で表現される、または関数が戻す条件を使用して、動的に（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。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 PL/SQL パッケージ内に、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ごとに、異なる述語を戻すことができます。

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・ポリシーに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書き込みします。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、アプリケーションではなく、データベース内で適用されます。つまり、異なるアプリケーションを使用しても、セキュリティ・ポリシーは回避されません。また、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、仮想プライベート・データベースは、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。

参照： 11-47 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。

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、アプリケーション・ロールの実装時には、次の処理を行うことをお勧めします。詳細は、各項を参照してください。

- [ロールの即時使用可能および使用禁止](#)
- [ストアド・プロシージャでの権限のカプセル化](#)
- [ユーザーが知らないロール・パスワードの使用](#)
- [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](#)

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。

1. 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ます。

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。アプリケーションの実行中に、状況に応じて、より厳密またはより緩やかなセキュリティを提供するために、より多いまたはより少ない権限を含むロールをいくつか設定する場合があります。各アプリケーション・ロールは、許可されていない使用を未然に防ぐために、パスワード（またはオペレーティング・システム認証）によって保護されている必要があります。

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。読取り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。

2. 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アプリケーション・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。
3. 終了時に、各アプリケーションは、使用可能にしていたアプリケーション・ロールを使用禁止にする必要があります。
4. アプリケーション・ユーザーには、必要に応じてアプリケーション・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・データベースを使用します。11-47 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。

さらに次の処理が可能です。

5. 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。この機能は、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 コール・インタフェース (OCI)・アプリケーションの機能に似ています。
6. PRODUCT_USER_PROFILE 表を使用して、SQL*Plus ユーザーに対して SET ROLE 文の使用を禁止します。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・ツールでも、PRODUCT_USER_PROFILE 表を利用して、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： 該当するツールのマニュアル (『Oracle8i 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。

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 Tool 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： 11-59 ページの「[例 3: Human Resources アプリケーション #2](#)」を参照してください。

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持っていないため、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り出します。

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。この方法には、次の欠点があります。

- 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。
- ユーザーには、パスワードを取り出すために使用するストアド・プロシージャの EXECUTE 許可が必要です。ユーザーは、プロシージャを実行してパスワードを取り出し、アプリケーション外でロールを使用できます。

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、サーバー実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わせます。

参照： 11-47 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。

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
- スキーマに依存しないユーザー

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、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 SCHEMA` 文を発行して、ユーザーを正しいアプリケーション・スキーマへ接続させることができます。

スキーマに依存しないユーザー

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・ユーザーおよびスキーマに依存しない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する必要はなく、データベースのスキーマに依存しないユーザーです。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が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 X.509 証明書に含まれるユーザーの識別名 (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便利です。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、バグ・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、データベースは最初に完全 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。

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。このため、これらのスキーマに依存しないユーザーには、各データベース上に専用データベース・スキーマが必要ありません。そのため、ユーザーとスキーマを分離することによって、企業内でのユーザー管理のコストが低くなります。

参照： 2-40 ページの「[スキーマの改名](#)」を参照してください。

『Oracle8i Advanced Security 管理者ガイド』も参照してください。

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上で、各ロールに対して付与する権限を決定します。

オブジェクト権限

通常、エンドユーザ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 オブジェクト権限

オブジェクト権限	表	ビュー	順序	プロシージャ (1)
ALTER	3		3	
DELETE	3	3		
EXECUTE				3
INDEX	3 (2)			
INSERT	3	3		
REFERENCES	3 (2)			
SELECT	3	3 (3)	3	
UPDATE	3	3		

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

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。

アプリケーションを実装およびテストするときに、これらの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 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（表またはビュー）
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（表またはビュー）
INSERT	INSERT INTO オブジェクト（表またはビュー）
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	順序を使用する SELECT...FROM オブジェクト（表、ビューまたはスナップショット）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。

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・システムまたはネットワーク認証サービス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。こうすることによって、ロールの管理を集中化できます。

ロールの集中管理には、多くの利点があります。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箇所を変更できます。

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つのアプリケーション・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。こうすることによって、ユーザーは別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。

参照： 11-25 ページの「[ロールの明示的な使用可能](#)」および 11-12 ページの「[ツール・ユーザーからのアプリケーション・ロールの制限](#)」を参照してください。

『Oracle8i 管理者ガイド』も参照してください。

ネットワーク認証サービスの詳細は、『Oracle8i 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

ユーザーにはロールを付与できますが、そのロールがあらかじめ使用可能になっていない場合、ロールに対応付けられている権限をユーザーの現行のセッションで使用することができません。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。

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。つまり、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。

デフォルトのロール

デフォルトのロールは、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務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ているどのロールも、ユーザーのデフォルト・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、デフォルトの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、（初期化パラメータ `MAX_ENABLED_ROLES` 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： デフォルトのロールは、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。たとえば、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。

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。

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。

```
GRANT PAYROLL_CLERK TO hr;  
GRANT ACCTS_PAY TO hr;  
GRANT ACCTS_REC TO hr;
```

```
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。

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;
```

```
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。たとえば、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、『Oracle8i 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 を使用して削除します。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 を使用して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

システム権限およびロールは、次の例に示すように、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 の使用による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 を使用して付与できます（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）。このオプションを使用して付与された場合、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 を使用して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 で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。

ユーザーが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、権限付与者に、付与されるすべてのシステム権限およびロールに対する ADMIN OPTION が必要です。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 を選択的に取り消すことはできません。必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再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から、権限またはロールを取り消すことができます（このユーザーは、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 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 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 がユーザーに付与されており、そのユーザーがプロシージャを作成している場合、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を許可し直さなければ、それらのプロシージャは（取消し後）再使用できません。

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します。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。

GRANT OPTION の使用によるスキーマ・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 を使用して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 を付けてまたは付けないで付与できます。
- 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 GRANT OPTION 付きで自動的に付与されます。

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。
- GRANT OPTION 付きで付与されているスキーマ・オブジェクト権限が、権限付与者に付与されている。

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、ユーザーが権限付与者である場合に、ユーザー JWARD および TSMITH から、EMP_TAB 表の SELECT 権限および INSERT 権限を取り消すには、次の文を入力します。

```
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 に対して付与した表 DEPT_TAB のすべての権限を取り消すこともできます（付与されている権限が1つのみの場合も可）。

```
REVOKE ALL ON Dept_tab FROM human_resources;
```

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 を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から、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 前述したように、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、最初に表またはビュー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、取消し対象となる権限の付与者である必要があります。

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わせる 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、その索引は権限が取り消された後も存在します。
- 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最後の DDL 時刻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 に付与する必要があります。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勧めします。

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 から取り消す場合（たとえば、SELECT ANY TABLE、UPDATE ON EMP_TAB など）、データベース内の（ファンクションとパッケージを含む）すべてのプロシージャを使用するには、それらを再許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。

ファイングレイイン・アクセス・コントロール

ファイングレイイン・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 Server 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またはビュー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。

- [ファイングレイイン・アクセス・コントロールの機能](#)
- [表またはビューへのポリシーの追加方法](#)
- [動的に変更される文の例](#)

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。

- 表ベースまたはビュー・ベースのセキュリティ・ポリシー
- 各表またはビューに対する複数のポリシー
- 高いパフォーマンス

表ベースまたはビュー・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、表またはビューにセキュリティ・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	表またはビューにセキュリティ・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・ポリシーを表またはビュー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・ポリシーを表またはビューに追加することは、表ベースまたはビュー・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、ポリシーは 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。

各表またはビューに対する複数のポリシー

同一の表またはビュー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します。基本アプリケーションのポリシー関数を作成し直さなくても、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、これらのポリシーはそれぞれ、表のすべてのアクセスに適用されます。アプリケーションによってこのポリシーを分割することはできません。

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使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。

注意： パフォーマンス上の理由から、インスタンスiertされた PL/SQL パッケージ内の解析済静的 SQL 文は、再解析されないことがあります。ポリシー関数を強制的に再評価する場合は、動的 SQL の使用をお勧めします。詳細は、11-50 ページの「[SYS_CONTEXT](#) での使用に推奨される動的 SQL」を参照してください。

表またはビュー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。このパッケージには、次の 4 つのプロシージャが含まれます。

表 11-3 DBMS_RLS プロシージャ

プロシージャ	用途
DBMS_RLS.ADD_POLICY	表またはビューにポリシーを追加できます。
DBMS_RLS.DROP_POLICY	表またはビュー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	表またはビュー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。

これらのプロシージャを使用すると、ポリシーを追加する表またはビュー、ポリシー名、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）、および追加情報を指定できます。

たとえば、ADD_POLICY プロシージャには、挿入（または更新）の前後にポリシーをチェックする UPDATE_CHECK パラメータが含まれています。更新または挿入後にポリシーがチェックされない場合は、更新または挿入は許可されていません。オラクル社は、UPDATE_CHECK パラメータを使用することをお勧めします。UPDATE_CHECK パラメータを使用すると、レコードの挿入は可能であったのに、そのレコードを後で選択できないというユーザーの不満を解消できます。

参照：『Oracle8i PL/SQL 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。

動的に変更される文の例

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。この場合のプロセスは、次のようになります。

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 Server 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 Server が、動的に変更された文を実行します。
6. 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だけに制限します。

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、11-53 ページの「例」および『Oracle8i PL/SQL 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。

アプリケーション・コンテキスト

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。これは、ユーザーのアクセス権限に基づいた安全なアプリケーションを開発する際に、特に役立ちます。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。

Oracle Human Resource 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務を判断する部分が含まれています。この職務 ID が、Oracle Human Resource 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](#)
- [アプリケーション・コンテキストの機能設計原理](#)
- [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](#)
- [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に指定された属性に合うように調整されたセキュリティ
- 妥当性チェックを介するセキュリティ
- 事前定義属性にアクセスするための USERENV アプリケーション・コンテキスト名前空間

各アプリケーションに指定された属性に合うように調整されたセキュリティ

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。

どの場合も、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。

妥当性チェックを介するセキュリティ

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、組み合わ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属性を変更しないように、Oracle 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を検証します。

事前定義属性にアクセスするための USERENV アプリケーション・コンテキスト名前空間

Oracle8i では、組込みアプリケーション・コンテキスト名前空間 USERENV が提供されています。これによって、事前定義属性（ユーザー・セッションに関してデータベースが獲得する情報であるセッションの基本形）へのアクセスができます。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 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。それ以外の場合（そのユーザーが直接データベースに接続している場合）は、ユーザーはどのデータにもアクセスできません。

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。これらの属性を、[表 11-4](#) に示します。USERENV 名前空間およびその事前定義属性の詳細は、『Oracle8i SQL リファレンス』の SYS_CONTEXT を参照してください。

次の構文を使用して、現行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・コンテキスト名前空間は、データベースの以前のリリースで提供されている USERENV 関数と置き換えることができます。

表 11-4 USERENV 名前空間の主な事前定義属性

事前定義属性	意味
TERMINAL	現行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。 <i>language_territory.characterset</i>
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。
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	現行セッションの地域を戻します。
NLS_CURRENCY	現行セッションの通貨記号を戻します。
NLS_CALENDAR	現行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	現行セッションの現行の日付書式を戻します。
NLS_DATE_LANGUAGE	現行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	現行セッションがその権限下にある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	現行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	カレント・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。

表 11-4 USERENV 名前空間の主な事前定義属性 (続き)

事前定義属性	意味
SESSION_USERID	カレント・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返します。
CURRENT_SCHEMA	現行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返します。これは、ALTER SESSION SET SCHEMA 文で変更できます。
CURRENT_SCHEMAID	現行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返します。これは、ALTER SESSION SET SCHEMAID 文で変更できます。
PROXY_USER	SESSION_USER の代理として現行セッションをオープンしたデータベース・ユーザー (通常は中間層) の名前を返します。
PROXY_USERID	SESSION_USER の代理として現行セッションをオープンしたデータベース・ユーザー (通常は中間層) の識別子を返します。
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返します。
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返します。
HOST	データベースが実行しているホスト・マシン名を返します。
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・ユーザー名を返します。
EXTERNAL_NAME	データベース・ユーザーの外部名を返します。
IP_ADDRESS	クライアントの接続元のマシンの IP アドレスを返します。
NETWORK_PROTOCOL	接続文字列 (PROTOCOL= <i>protocol</i>) で指定されているプロトコルを返します。
BG_JOB_ID	バックグラウンド・ジョブ ID を返します。
FG_JOB_ID	フォアグラウンド・ジョブ ID を返します。
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返します (認証コンテンツがある場合は、それを返します)。

アプリケーション・コンテキストの機能設計原理

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・コントロールは静的アプリケーションを処理するように設計されています。つまり、アプリケーション・コンテキスト内にセキュリティ属性のあるアプリケーションは、ユーザー・セッション内で静的です。つまり、この機能は、ユーザーがログインし、そのユーザーのアプリケーション・コンテキストがセッションに設定され、ユーザーがログオフするまでそのコンテキストが変更されないようなアプリケーションに使用する場合に最適です。この設計原理を使用すると、多くのユーザーが、完全に解析済の最適化された文を共有できるため、アプリケーション・コンテキストのスケーラビリティが向上します。

アプリケーションがコンテキスト属性をセッション内で変更する必要がある場合は、ユーザーが実行するカーソルに、新しいアプリケーション・コンテキスト属性が反映されていることをプログラムで確認する必要があります。たとえば、アプリケーションが、ユーザーの `position` 属性を `Human Resources` コンテキスト内で変更して、ユーザーがそのセッション内でより多くのデータを参照するとします。このアプリケーションはカーソル再解析を強制的に行い、新しい `position` 属性をデータへのアクセス制限に使用する必要があります。

カーソル再解析を強制的に行うには、少なくとも次の3つの方法があります。

- ポリシーのリフレッシュ
- 明示的な再解析の強制実行
- 動的 SQL の使用

ポリシーのリフレッシュ

ファイングレイン・アクセス・コントロールへの管理インタフェースは、DBMS_RLS パッケージです。DBMS_RLS.REFRESH_POLICY プロシージャを使用して、セキュリティ・ポリシーをリフレッシュできます。これによって、現行トランザクションを強制的にコミットすることができます。この方法の欠点は、指定された表またはビューに対応するすべてのカーソル（アプリケーション・コンテキストが変更されていないユーザーのカーソルを含む）をフラッシュすることです。この方法によって、目的の効果は得られますが、パフォーマンスが低下することがあります。

参照：『Oracle8i PL/SQL 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。

明示的な再解析の強制実行

カーソルの明示的な再解析を強制的に実行します。Oracle コール・インタフェースを使用しているアプリケーションの場合、REPARSE 文を発行することによって、オープン・カーソルを明示的に再解析できます。これによって SQL 文を強制的に再解析し、ユーザーのアプリケーション・コンテキストの変更を反映し、その変更を（新しく解析された）文に適用できます。ただし、PL/SQL では、文の明示的な再解析はできません。そのため、この方法は PL/SQL 文では有効ではありません。

動的 SQL の使用

PL/SQL 文では、動的 SQL を使用して、アプリケーションに一意のコンテキストに対応付けられたパッケージを作成できます。この方法については、次の項で説明します。

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。仮想プライベート・データベース（VPD）は、アプリケーション・コンテキストとファイングレイン・アクセス・コントロールの組合せを表す用語です。

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・コントロール固有の次の用途に使用できます。

- 安全なデータ・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として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

安全なデータ・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、探さなくても番号が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わせるのではなく、アプリケーション・コンテキスト内に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便利です。たとえば、述語が4つの属性（従業員番号、コスト・センター、職務、支出制限など）に基づくポリシー関数は、この情報を取り出す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、つまり正しいセキュリティ・ポリシーを戻すことができます。

受注アプリケーションでは、「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」というポリシーを施行します。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り出せますが、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、次の文があるとしたします。

```
SELECT * FROM Orders_tab
```

これを、次のように動的に変更します。

```
SELECT * FROM Orders_tab
WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

述語のバインド変数として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。すべての顧客は同じポリシー（顧客は自分の注文のみ参照できる）を共有します。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。その結果として、1つの共有カーソルが各ユーザーに別々に実行されます。これは、バインド変数（顧客番号）が実行時に評価されるためです。この変数は、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。

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り出す前のコンテキストの設定
- 手順 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

まず、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 SQL_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ている必要があるため、ログイン・トリガーの使用をお勧めします。

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：『Oracle8i PL/SQL 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。

SYS_CONTEXT 構文

この関クションの構文は、次のとおりです。

```
SYS_CONTEXT ('namespace', 'attribute', [length])
```

この関クションは、コンテキスト名前空間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。事前定義の名前空間 USERENV 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。

参照： 11-42 ページの「事前定義属性にアクセスするための USERENV アプリケーション・コンテキスト名前空間」を参照してください。属性の詳細は、『Oracle8i SQL リファレンス』を参照してください。

SYS_CONTEXT での使用に推奨される動的 SQL

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 では、パフォーマンス上の理由から、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。動的 SQL では、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。

SYS_CONTEXT での使用に推奨しないパラレル問合せ

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します。

たとえば、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、Oracle8i 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とられます。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、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、名前空間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 はコンテキスト名前空間、oe_context はコンテキスト名前空間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。

作業 3: ユーザーがデータを取り出す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： 11-45 ページの「[アプリケーション・コンテキストの機能設計原理](#)」を参照してください。

11-53 ページの「[例](#)」も参照してください。

手順 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

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3つ説明します。

- [例 1: Order Entry アプリケーション](#)
- [例 2: Human Resources アプリケーション #1](#)
- [例 3: Human Resources アプリケーション #2](#)

例 1: Order Entry アプリケーション

この比較的単純な例では、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、次のステップで説明します。

- [ステップ 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](#)
- [ステップ 2. アプリケーション・コンテキストの作成](#)
- [ステップ 3. 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でのアプリケーション・コンテキストへのアクセス](#)
- [ステップ 4. 新しいセキュリティ・ポリシーの作成](#)

この例での手順は次のとおりです。

- ユーザーと顧客は1対1の関係を想定します。
- ユーザーの顧客番号 (Cust_num) を検索します。
- 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。

注意： 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 11-39 ページの「[動的に変更される文の例](#)」を比較してください。

[第 12 章「トリガーの使用」](#) も参照してください。

ステップ 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

```
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、『Oracle8i SQL リファレンス』を参照してください。

ステップ 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;
```

ステップ 3. 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でのアプリケーション・コンテキストへのアクセス

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 コンテキスト属性に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;

```

ステップ 4. 新しいセキュリティ・ポリシーの作成

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。つまり、動的な述語によって、次のユーザーの文が、

```
SELECT * FROM Orders_tab;
```

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、すべての担当顧客のレコードを参照でき、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 (Lightweight Directory Access Protocol) 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。

例 2: Human Resources アプリケーション #1

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。

- ステップ 1. 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 パッケージの作成
- ステップ 2. コンテキストの作成およびパッケージへの対応付け
- ステップ 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 名前空間に割り当てられていると想定しています。

ステップ 1. 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 パッケージの作成

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 (resp_id NUMBER);
  PROCEDURE set_org_id (org_id NUMBER);
  /* PROCEDURE validate_resp_id (resp_id NUMBER); */
  /* PROCEDURE validate_org_id (org_id 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 (resp_id NUMBER) IS
  BEGIN

    /* validate resp_id based on primitive and other context */
    /* validate_resp_id (resp_id); */

    /* set resp_id attribute under namespace 'hr_ctx' */
    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'resp_id', resp_id);
  END set_resp_id;

  /* function to set organization id */
  PROCEDURE set_org_id (org_id 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;
```

ステップ 2. コンテキストの作成およびパッケージへの対応付け

```
CREATE CONTEXT Hr_ctx USING Apps.Hr_sec_ctx;
```

ステップ 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: Human Resources アプリケーション #2

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
```

この例では、Oracle8i 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。

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

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、直属の部下に限定されます。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、表に対するファイニングレイン・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・コントロールによって、許可されていないユーザーが誤って行を変更してしまうことを防ぐため、この例では、表に対する UPDATE 権限をパブリックに付与します。

```
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;
```

中間層を介する認証

3 層システム（たとえば、データベース、アプリケーション・サーバー、ブラウザの 3 層システム）は、インターネットの発展に伴って飛躍的に発展しました。このため、3 層アプリケーションは、「インターネット・コンピューティング・モデル」と呼ばれることがあります。

Oracle8i の n 層認証は、3 層アプリケーションで発生する多くのセキュリティ上の問題を解決します。これによって、組織は 3 層システムにおけるセキュリティのリスクを最小限に抑さえながら、インターネット・コンピューティングの利点を得ることができます。この項の内容は次のとおりです。

- [n 層認証の利点](#)
- [3 層コンピューティングのセキュリティ問題](#)
- [Oracle8i の n 層認証のソリューション](#)

n 層認証の利点

3 層システムによって、組織には次のような多くの利点があります。

- 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、アプリケーション論理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・コストを低く抑えることができます。

さらに、Oracle の n 層認証には、次のようなセキュリティの利点があります。

- 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る制限付きトラスト・モデル。
- 軽量ユーザー・セッションを OCI でサポートし、クライアントの再認証のオーバーヘッドを排除することによって得られるスケーラビリティ。
- 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アクションの監査を可能にする信頼性。

注意： Oracle8i 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利点がありますが、多くの新しいセキュリティの問題も発生します。

- 実際のユーザーとは
- 中間層の権限が適切であるか
- 監査方法および監査対象のユーザー
- 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

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、セキュリティは、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。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の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便利です。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しか使用できませんが、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則です。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) データを取り出す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、Oracle8i データベースは、認証されていない接続は許可しません。中間層に対するデータベース認証についても相互関係があります。

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り出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 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限りません。各ユーザーに2組の認証を組み合わせることによって、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り出す場合、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、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) Oracle8i データベースおよび Oracle 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。

Oracle8i の n 層認証のソリューション

次の項では、前述した各問題に対する Oracle8i での対処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アクションの監査

実際のユーザーの認証を介する引渡し

多くの組織は、中間層の利点を失わずに、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。Oracle8i では、Oracle コール・インタフェース (OCI) を介してクライアント認証を保持する機能を提供します。

OCI を使用すると、中間層は、単一のデータベース接続内で、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. 中間層は、Oracle8i が許容するすべての認証形式を使用して、中間層自体を Oracle8i 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 Oracle コール・インタフェースを使用して、ユーザーに対して 1 つ以上のセッションを作成します。軽量セッション情報には、少なくともユーザー名が含まれている必要があります。中間層はオプションで、クライアントに対するパスワードおよびロールを指定します。
4. データベースは、中間層にクライアントのパスワードを指定するように要求できないため、認証は OCI によって実行されます。クライアントのセッションを作成するために、中間層サーバーは OCISessionBegin 関数をコールします。OCISessionBegin をコールする前に、クライアントに関する必要な情報を中間層サーバーに提供するために、OCIAttrSet 関数がコールされます。この関数は、次の属性でコールされます。

OCI_ATTR_USERNAME	クライアントのデータベース・ユーザー名を設定します。 この属性は必須です。
OCI_ATTR_PASSWORD	クライアントがデータベース・パスワードを指定して、データベースによって妥当性チェックが行われると、中間層サーバーは、そのパスワードをそのユーザー名とともに渡します。この属性を指定しないと、中間層サーバーがクライアントを認証したとみなされます。
OCI_ATTR_PROXY_CREDENTIALS	この属性は、クライアントが中間層サーバーを介して接続していることをサーバーに知らせます。
OCI_ATTR_INITIAL_CLIENT_ROLES	中間層サーバーがクライアントとして接続するときに、一連のロールをアクティブにする場合、ロールのリストがこの属性とともに渡されます。

5. データベースは、指定されているロールを使用して、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（後述の「[中間層の権限の制限](#)」を参照）。

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

最少の権限とは、ユーザーが、その目的を実行するために必要最少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。Oracle8i では、特定のロールのみを使用して、中間層が特定の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。

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 WITH ROLE ALL;
```

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。

実際のユーザーの再認証

前述したように、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限りません。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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

実際のユーザーのかわりに行われるアクションの監査

Oracle8i の *n* 層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アクションを監査できます。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 ON bonuses 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 ON bonuses 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 ON bonuses;
```

データの暗号化

特定のアプリケーションでは、追加のセキュリティ対策として、データを暗号化できる場合があります。

適切な認証およびアクセス制御によって、適切に識別および認証されたユーザーのみがデータにアクセスできるようにすることによって、データ・セキュリティのほとんどの問題は処理できます。ただし、DBA はすべての権限を持っているため、データベースのデータは、通常、データベース管理者のアクセスからは保護できません。さらに、組織には、オフラインで格納されている機密性の高いデータ（サード・パーティーで格納されているバックアップ・ファイルなど）の保護に関する問題がある場合があります。

機密性が特に高く、暗号化を保証する必要がある情報には、クレジット・カード番号、プライバシーに関する厳密な法律を持つ国の国識別番号、または産業原則などの取引機密事項などがあります。データベースではなく、アプリケーションに対してユーザーが認証されるアプリケーションの場合も、暗号化を使用して、アプリケーション・ユーザーのパスワードまたは Cookie を保護する場合があります。

DBMS_OBFUSCATION_TOOLKIT パッケージ

このように機密性の高いデータを扱うアプリケーションでは、Oracle は DBMS_OBFUSCATION_TOOLKIT PL/SQL パッケージを提供して、（文字列入力およびロー入力を含む）データの暗号化および復号化を行います。この機能は、Data Encryption Standard (DES) などの選ばれたアルゴリズムに制限されています。開発者が開発者自身の暗号化アルゴリズムをプラグインすることはできず、キーの長さも固定されています。この機能は、暗号化を複数渡すことを禁止しています。つまり、暗号化コールをネストできないため、暗号化された値を暗号化します。これらの制限は、暗号化製品の輸出を管理する米国の法律で決められています。

暗号化機能は暗号化および復号化サービスのみを提供し、組込みキー管理サービス（自動キー・リカバリなど）は提供しません。このパッケージを使用している開発者は、プログラムでキー記憶域を処理する必要があります。たとえば、DBA が暗号化データを参照できないようにするには、暗号化キーをデータベースとは別に格納します。

開発時の考慮点

暗号化を使用しているアプリケーションでは、暗号化がその他のアクセス制御を妨げないようにする必要があります。つまり、アクセス権限が与えられているオブジェクトにユーザーがアクセスするのを、暗号化によって妨げないようにする必要があります。暗号化がアクセスを妨げると、ユーザーがジョブを実行できなくなります。たとえば、EMP に SELECT 権限を持つユーザーが、暗号化メカニズムによって、参照権限がある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。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利点はほとんどありません。ユーザーがデータを読み込む前に、データを復号化するオーバーヘッドが増えるだけです。

索引付けされたデータの暗号化は、サポートされていません。

慎重にセキュリティを実行することによって、定期的に暗号化キーを変更し、解決キーの侵害の可能性を軽減します。キーを変更するには、新しいキーを使用して、暗号化されたオブジェクトを復号化および再暗号化する必要があります。この処理には時間がかかる場合があります。また、この処理は、データがアクセスされないときに行う必要があります。

参照：『Oracle8i 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 III 部

アクティブ・データベース

第III部に含まれる章は、次のとおりです。

- 第 12 章「トリガーの使用」
- 第 13 章「システム・イベントの処理」
- 第 14 章「パブリッシュ / サブスクライブの使用」

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。Oracle8i になり、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
- トリガーの作成
- トリガーのコンパイル
- トリガーの変更
- トリガーの使用可能および使用禁止
- トリガーに関する情報のリスト
- トリガー・アプリケーションの例
- イベント発行のトリガー

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、データ整合性規則の適用のためにトリガーを定義しないでください。宣言整合性制約を使用して簡単に実現できます。
- 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勧めします。
- 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になる文を発行するかに関係なく、トリガーになる文に対して起動される集中的グローバル操作にのみ使用してください。
- **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 または Enterprise Manager などの対話型 Tool で使用できます。対話型 Tool 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;
/
```

次の SQL 文を入力するとします。

```
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、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、12-34 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。

トリガー作成の前提条件

トリガーを作成する前に、SYS として接続して CATPROC.SQL スクリプトを実行します。このスクリプトは、Oracle Server へのプロシージャ拡張に必要なスクリプト、またはプロシージャ拡張内で使用されるスクリプトをすべて自動的に実行します。

参照： 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。ご使用のプラットフォーム固有の Oracle マニュアル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

注意： Oracle8i リリース 8.1.5 で INSTEAD OF トリガーを使用できるのは Enterprise Edition のみです。これは将来のリリースでは変更される可能性があります。

- 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、『Oracle8i 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13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、トリガーがカレント・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： 12-54 ページの「[イベント発行のトリガー](#)」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、同スキーマ内の他のトリガーに関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、間違いやすいため、違う名前を付けることをお勧めします）。

トリガー文

トリガー文では、次のものを指定します。

- SQL 文の種類、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1つ、2つまたは3つすべてをトリガー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は、トリガー文にはビューのみを指定します。逆に、ビューがトリガー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 . . . ;
```

INSERT トリガーの動作

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）。

たとえば、A、B、C という 3 つの表があるとします。表 A には、表 B を検索して表 C に挿入する INSERT トリガーがあります。表 A をインポートすると、表 C も更新されます。

注意： 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）を指定すると、インポートでは既存の表はロードされません。したがって、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。トリガーが起動され、索引がメンテナンスされます。

UPDATE のための列リスト

トリガー文で UPDATE を指定すると、そのトリガー文にオプションで列リストを含めることができます。列リストを含めると、指定した列の 1 つが更新された場合にのみ、UPDATE 文のトリガーが起動されます。列リストを指定しないと、対応付けられた表のいずれかの列が更新されたときにトリガーが起動されます。INSERT または DELETE トリガー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 . . .
```

使用上の注意

- `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。

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` 行トリガーよりも、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一度読み込み（論理的な読み込みであり、物理的な読み込みではない）、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度読み込むだけで済みます。

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。トリガーによって、`UPDATE`、`INSERT` または `DELETE` 操作が基礎となる表に対して直接実行されます。

標準的な `UPDATE`、`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： 12-12 ページの「[FOR EACH ROW オプション](#)」を参照してください。

使用上の注意

- Oracle8i リリース 8.1.5 では、INSTEAD OF トリガーを使用できるのは Enterprise Edition のみです。将来のリリースでは、Standard Edition でも使用可能になる可能性があります。
- 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適用されません。INSTEAD OF トリガー本体でチェックを適用する必要があります。

変更不能のビュー

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 の導出元のベース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：『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

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、持続記憶域にそれをフラッシュ・バックするには、オブジェクト・ビューが変更可能なものでない限り、INSTEAD OF トリガーを指定する必要があります。ただし、オブジェクトが読取り専用の場合は、トリガーを定義して確保する必要はありません。

NESTED TABLE のビューの列に対するトリガー

INSTEAD OF トリガーは、NESTED TABLE のビューの列に対しても作成できます。このトリガーは、NESTED TABLE の要素を更新する手段を提供します。このトリガーは、NESTED TABLE の更新対象要素のそれぞれに対して起動されます。トリガー内の行相関変数が、NESTED TABLE の要素に対応します。この種のトリガーは、変更対象の NESTED TABLE を含む親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 NESTED TABLE の列に対してのみ定義できます。
 - NESTED TABLE 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 NESTED TABLE 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 NESTED TABLE 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;
```

NESTED TABLE 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。

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 回のみ起動されますが、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わけではありません。

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、『Oracle8i 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役立ちます。

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実行されます。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 の場合のように不定の場合）、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 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

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、関連名、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 1 次の例は、DBA がすべてのログイン・ユーザーをモニターする方法を示します。

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 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
CALL foo (ora_login_user)
/

```

例 2 この例は、Java プロシージャを起動するトリガーを示します。

```

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)

```

thjvTriggers.java

```

import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.oracore.*;
public class thjvTriggers
{
    public static void
    beforeDelete (NUMBER old_id, CHAR old_name)
    Throws SQLException, CoreException
    {
        Connection conn = JDBCConnection.defaultConnection();
        Statement stmt = conn.createStatement();
        String sql = "insert into logtab values
        (" + old_id.intValue() + ", '" + old_ename.toString() + "', BEFORE DELETE)";
        stmt.executeUpdate (sql);
        stmt.close();
        return;
    }
}

```

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、トリガー文により影響を受ける現在の行に含まれる new 列値および old 列値にアクセスできます。変更される表の各行に 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。トリガー文の種類によっては、相関名が意味を持たない可能性もあります。

- 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、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、トリガー文が Emp_tab 表（列 SAL、COMM など）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は、コロンは使用できません。

NESTED TABLE のビューの列に対する INSTEAD OF トリガー

NESTED TABLE 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、NESTED TABLE の新しい要素および古い要素に対応します。この NESTED TABLE 要素に対応する親行は、parent 修飾子を使用してアクセスできます。parent 相関名は、NESTED TABLE トリガー内でのみ意味があり有効です。

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起きないため、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）の列を含む表 new 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 に改名し、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。

条件述語

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、トリガー本体に条件述語の INSERTING、DELETING、UPDATING を使用し、トリガーを起動する文の種類に応じて特定のコード・ブロックを実行できます。次のトリガー文を考えてみます。

```
INSERT OR UPDATE ON Emp_tab
```

トリガー本体のコード内に、次の条件を指定できます。

```
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。次の文は前述のトリガーを実行し、UPDATING(sal) 条件述語は TRUE と評価されます。

```
UPDATE Emp_tab SET Sal = Sal + 100;
```

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引き起こすことによって、トリガー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適用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。

トリガーのサイズ トリガーのサイズは、32KB より小さく指定する必要があります。

トリガー本体で有効な 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ったら、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、LONG RAW および LOB データ型の使用 トリガー内の LONG、LONG RAW および LOB 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 および 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。
- :NEW 変数の LOB 値は、トリガー本体では変更できません。次に例を示します。

```
:NEW.Column := ...
```

これは、列のデータ型が LOB の場合は実行できません。

注意： 以前は、この例の column がオブジェクト、配列変数または NESTED TABLE の場合は、この文を指定できませんでした。この制約が、リリース 8.1.5 ではなくなっています。

パッケージ変数の参照 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。

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えます。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、初期化パラメータの OPEN_CURSORS 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、『Oracle8i 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）。

リリース 7.1 より前の Oracle を使用している場合は、同じ型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 7.1.0 よりも以前のバージョンに設定されている場合にも、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 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 宣言参照整合性制約の影響によって更新する必要がある表のことです。このような表の制約は、処理中の文を発行したセッションにのみ適用されます。

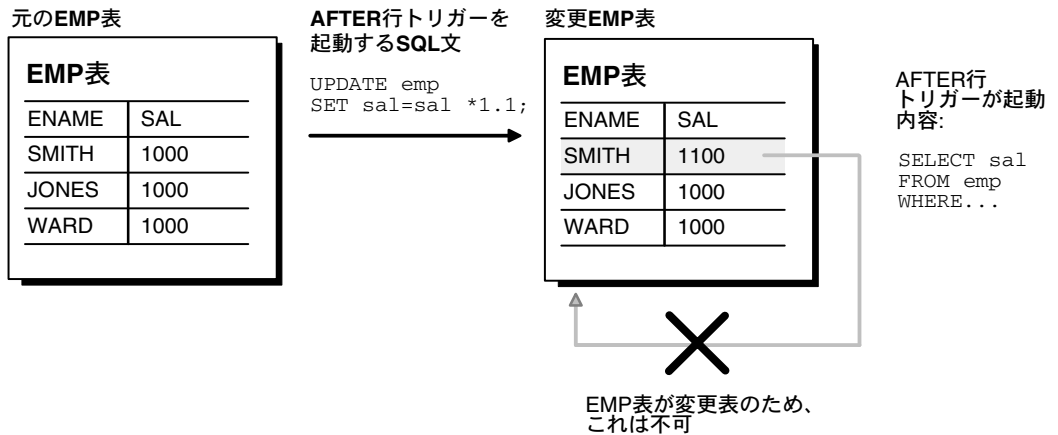
DELETE CASCADE の結果としてトリガーが起動されない場合、表は、文トリガーの変更表または制約表とみなされません。ビューは、INSTEAD OF トリガー内では変更ビューまたは制約ビューとみなされません。

すべての行トリガー、または DELETE CASCADE の結果起動された文トリガーには、変更表に関する 2 つの重要な制限があります。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。

- トリガー中の SQL 文は、トリガーを実行する文の変更表を変更したり、変更表から読み込む（問い合わせる）ことはできません。

図 12-1 に、変更表に対する制約を示します。

図 12-1 変更表



SQL 文が表の最初の行に対して実行され、次に AFTER 行トリガーが起動されることに注意してください。次に、AFTER 行トリガー本体内の文が、元の表を問い合わせようとします。ただし、EMP 表は変更表であるため、この問合せは Oracle では使用できません。無理に実行するとランタイム・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;
```

この結果、次のエラーが戻されます。

ORA-04091: 表 SCOTT.Emp_tab は変更中の為、トリガー機能はこの表を読み込み / 修正することができません。

トリガーが起動すると Oracle はこのエラーを戻します。これは、最初の行が削除されるときに、表が変更表のためです。(Empno が主キーのため、この文によっては 1 行しか削除されません。ただし、Oracle では、削除文が 1 行しか削除していないことを認識しません。)

前述のトリガーから 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、トリガーの起動時に表は変更表ではなくなり、そのトリガーは正しいデータを出力します。

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、『Oracle8i 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適用されません。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 Net8 パスを介して、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、予測不能な動作をする場合があります。

変更表の制約の緩和 Oracle8i 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。Oracle8i 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み込みまたは変更することはできません。

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使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、変更済の、別のトランザクションによってコミットされていない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た後は、ど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

イベントの特性 イベントの違いに応じて、発行機能による制限も様々になります。サーバーですべての制限を適用できない場合があります。完全には適用できない制限については、明確に文書化されています。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。

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。

参照： 他の制限の詳細は、13-6 ページの「[イベントのリスト](#)」を参照してください。

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

次の文を入力するとします。

```
SELECT Username FROM USER_USERS;
```

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。

権限

トリガーの作成権限

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。

- トリガー文に指定した表を所有していること
- トリガー文中の表に対する 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。

参照スキーマ・オブジェクトに対する権限

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・プロシージャの場合と似ています。

トリガーのコンパイル

トリガーは、PL/SQL 無名ブロックに `:new` および `:old` 機能を追加したものと似ていますが、コンパイル方法が異なります。PL/SQL 無名ブロックは、メモリーにロードされるたびにコンパイルされます。コンパイルには、次の 3 段階が必要です。

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

依存性

コンパイル済のトリガーには依存性があります。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、トリガーの依存性がわかります。たとえば、次の文は、SCOTT スキーマ内のトリガーの依存性を示します。

```
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。

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 7.3 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 7.3 以降のリリースを 7.3 より前のリリース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、ストアード・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 コマンドは、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： 9-47 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

トリガーは、作成時にデフォルトで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 コマンドの ENABLE オプションを使用します。表 INVENTORY の使用禁止にされたトリガー REORDER 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 コマンドの ENABLE 句に ALL TRIGGERS オプションを使用すると、1つの文で、ある表に定義してあるすべてのトリガーを使用可能にできます。たとえば、表 INVENTORY 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る、トリガーは実行せずにただちに処理する場合
- データを再ロードする場合

トリガーの作成時には、トリガーはデフォルトで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 コマンドの DISABLE オプションを使用します。

たとえば、表 INVENTORY のトリガー REORDER 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、ある表に関連するトリガーをすべて使用禁止にできます。たとえば、表 INVENTORY に定義されているトリガーをすべて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。

列 ACTION_TYPE 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。

列 TRIGGER_TYPE 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。

列 TRIGGERING_EVENT 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、『Oracle8i リファレンス・マニュアル』を参照してください。

トリガー REORDER 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、トリガー REORDER 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
- 参照整合性の実施（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・ルールの適用
- 複雑なセキュリティ認可の適用
- 透過的なイベント・ロギング
- 導出列値の自動生成
- 更新可能な複雑なビューの作成
- システム・イベントの追跡

この項では、上記のトリガー・アプリケーションの例をそれぞれ紹介します。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

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、行単位の表に対する値に基づく監査ができます。

ときには、Oracle の AUDIT 文が機密保護監査機能とみなされるのに対して、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。

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に、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 の標準機能で利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限り、トリガー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、『Oracle8i 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 トリガーによって、トリガー文が適切な整合性制約に従った後、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意味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、行トリガーは、表に対する行単位の値に基づく監査を提供します。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便利です。

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 に設定し直すこともできます。次の簡単な AFTER 文トリガーは、トリガー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 行トリガーの場合がトリガー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、トリガー文の実行が終了したときにのみ 1 回起動されます。

トリガーを使用して監査を行う別の例を次に示します。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

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あらゆる文に対して制約が適用されます。

参照： [第4章「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、表中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。

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適用するトリガーを記述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・ルールを適用するためにのみ使用するようになっています。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、次の利点があります。

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利点があります。
------	---

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適用できますが、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適用に使用できます。たとえば、トリガーを使用して次を適用できます。

- 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適用できます。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、関連する従属データはすべてデフォルト値に置き換えられます）を適用する場合か、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適用する場合に限定するようにします。

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限り）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つまたは複数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、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、参照整合性の適用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同時実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 参照アクションを適用します。

```
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と外部キーの両方がある表）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 トリガーおよび DELETE SET NULL 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 SET NULL 参照アクションを適用します。

```
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 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 参照アクションを適用します。

```
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 のコードと組み合わせられます。

親表に対する UPDATE CASCADE トリガー 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反映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、本番環境で常に正しく作動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

トリガーは、参照整合性以外の整合性ルールも適用できます。たとえば、次のトリガーは、トリガー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証およびトリガー

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証の適用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セキュリティ認証の適用にのみ使用します。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証の適用にトリガーを使用する場合は、BEFORE 文トリガーを使用するのが最もよい方法です。BEFORE 文トリガーを使用すると、次のような利点があります。

- トリガー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、未許可の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、トリガー文に影響される各行ごとではなく、トリガー文に対して 1 回のみ実施されます。

次の例は、セキュリティを適用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、トリガーは非常に便利です。

REORDER トリガーの例は、一定の条件が満たされる（トリガー文が入力されること、PARTS_ON_HAND 値が REORDER_POINT 値より小さくなること）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

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り出すことができます。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。

- 導出値をトリガー文で使えるようにするには、INSERT または UPDATE が発生する前に依存値を取り出す必要があります。
- トリガーとなる 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;
```

更新可能な複合ビューの作成

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 からベース表に対する 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 トリガーが役立ちます。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を説明したものです。


```

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、セクションと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、NESTED TABLE である booklist に対してトリガーを定義して、NESTED TABLE の要素の変更を処理することもできます。

システム・イベントの追跡

ファイングレイイン・アクセス・コントロール 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 Oracle8i の機能で、ファイングレイイン・アクセス・コントロールを実装する機能です。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、プロシージャ set_ctx 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。トリガー setexpensectx 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_ctx.Set_ctx

```

イベント発行のトリガー

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 13 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イン・アクセス・コントロールの統合。

発行フレームワーク

Oracle のフレームワークによって、システム・イベントの発行を宣言的に定義できます。これによって、トリガーでデータベース・イベントをサポートすることができるため、ユーザーは、イベントが発生したときに実行され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。

システム・イベント発行サブシステムは、アドバンスト・キューイングのパブリッシュ / サブスクライブ・エンジンと緊密に統合されています。パブリッシュ / サブスクライブ・アプリケーションでは DBMS_AQ.ENQUEUE() プロシージャが使用され、カートリッジなどのパブリッシュ / サブスクライブ以外のアプリケーションではコールアウトが使用されます。

ユーザーまたはシステム管理者は、発行属性を指定するトリガーを作成することによって、システム・イベントの発行を使用可能にできます。デフォルトでは、トリガー（したがって、トリガー内に指定されているイベントの発行）が使用可能になります。ユーザーは、ALTER TRIGGER 文を使用してトリガーを使用禁止にし、システム・イベントの発行を使用禁止にすることもできます。

参照： 『Oracle8i 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および発行済イベントの配布方法の詳細は、『Oracle8i 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』および『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

イベントの発行

サーバーによってイベントが検出されると、トリガー機構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ます。次に、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： イベントの検出は、サーバーの特定リリースごとに事前に定義されています。ユーザー指定イベントを検出する機能は装備されていません。

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。

オブジェクトに対しては複数のトリガーを作成できるため、同一イベントに対する応答として複数の発行が行われる可能性があります。この場合、発行順序を想定してはいけません。発行はシステム・イベントが発生した順に行われます。

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、13-2 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、13-6 ページの「[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。

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。

参照： この章の情報を使用する前に、[第 12 章「トリガーの使用」](#)の内容を理解しておいてください。

イベント属性関数

トリガーが起動されたときに、イベント固有の属性を取得できます。これらの属性は、スタンドアロン関数として使用できます。

使用上の注意

- 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、これらの関数は 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勧めします。

表 13-1 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。	<pre>if (ora_sysevent = 'LOGON') then addr := ora_client_ip_ address; end if;</pre>
ora_database_name	VARCHAR2 (50)	データベース名。	<pre>DECLARE db_name VARCHAR2 (50); BEGIN db_name := ora_database_name; END;</pre>
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。	<pre>IF (ora_dict_obj_type = 'USER') THEN INSERT INTO event_table (ora_des_encrypted_password); END IF;</pre>
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。	<pre>INSERT INTO event_table ('Changed object is ' ora_dict_obj_ name');</pre>

表 13-1 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_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;
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。	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。	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。	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;

表 13-1 システムで定義されたイベント属性（続き）

属性	型	説明	例
<code>ora_is_creating_nested_table</code>	BOOLEAN	現在のイベントが NESTED TABLE を作成しているときに、TRUE を返します。	<pre>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;</pre>
<code>ora_is_drop_column(column_name IN VARCHAR2)</code>	BOOLEAN	指定された列が削除された場合、TRUE を返します。	<pre>if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_drop_ column('FOO'); end if;</pre>
<code>ora_is_servererror</code>	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 を返します。	<pre>IF (ora_is_servererror(error_ number)) THEN INSERT INTO event_table ('Server error!!'); END IF;</pre>
<code>ora_login_user</code>	VARCHAR2(30)	ログイン・ユーザー名。	<pre>SELECT ora_login_user FROM dual;</pre>
<code>ora_privileges(privilege_list OUT ora_name_list_t)</code>	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 パラメータで返します。つまり、戻り値として権限の数を返します。	<pre>if (ora_sysevent = 'GRANT' or ora_ sysevent = 'REVOKE') then number_of_privileges := ora_privileges(priv_list); end if;</pre>

表 13-1 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_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。つまり、 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);
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_sysevent	VARCHAR2 (20)	トリガーを起動するシ ステム・イベントです。 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;

イベントのリスト

リソース・マネージャ・イベント

リソース・マネージャ・イベントは、インスタンスの起動および停止に関係します。リソース・マネージャ・イベントに関して作成されるトリガーは、データベース・オブジェクトと対応付ける必要があります。

表 13-2 に、リソース・マネージャ・イベントのリストを示します。

表 13-2 リソース・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	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	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-4030 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から 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

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() および 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 および ALTER などの DDL 操作を実行できません。

これらのトリガーで利用できる 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（たとえば CREATE TRIGGER）の対象になる場合、このトリガーが、後で、同じトランザクション中に起動されるようにすることはできません。

表 13-3 に、クライアント・イベントのリストを示します。

表 13-3 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
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 LOGOFF	ユーザー・ログオフの開始時	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE CREATE	カタログ・オブジェクトの作成時	ora_sysevent
AFTER CREATE		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 ALTER	カタログ・オブジェクトの変更時	ora_sysevent
AFTER ALTER		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	カタログ・オブジェクトの削除時	ora_sysevent
AFTER DROP		ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner

表 13-3 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 ANALYZE	分析文の発行時	ora_sysevent
AFTER ANALYZE		ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS	関連情報の発行時	ora_sysevent
AFTER ASSOCIATE STATISTICS		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	監査または非監査文の発行時	ora_sysevent
AFTER AUDIT		ora_login_user ora_instance_num ora_database_name
BEFORE NOAUDIT		
AFTER NOAUDIT		
BEFORE COMMENT	オブジェクトへのコメントの追加時	ora_sysevent
AFTER COMMENT		ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE	オブジェクトの作成時	ora_sysevent
AFTER CREATE		ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)

表 13-3 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 DDL	ほとんどの SQL DDL 文の発行時。 アドバンスド・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
AFTER DDL		ora_login_user
		ora_instance_num
		ora_database_name
		ora_dict_obj_name
		ora_dict_obj_type
		ora_dict_obj_owner
BEFORE DISASSOCIATE STATISTICS	非関連情報の発行時	ora_sysevent
AFTER DISASSOCIATE STATISTICS		ora_login_user
		ora_instance_num
		ora_database_name
		ora_dict_obj_name
		ora_dict_obj_type
		ora_dict_obj_owner
		ora_dict_obj_name_list
		ora_dict_obj_owner_list
BEFORE GRANT	権限付与文の発行時	ora_sysevent
AFTER GRANT		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

表 13-3 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 RENAME	改名文の発行時	ora_sysevent
		ora_login_user
AFTER RENAME		ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
BEFORE REVOKE	取消し文の発行時	ora_sysevent
		ora_login_user
AFTER REVOKE		ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privileges
BEFORE TRUNCATE	オブジェクトの切捨て時	ora_sysevent
		ora_login_user
AFTER TRUNCATE		ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

パブリッシュ / サブスクライブの使用

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。この章の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 / サブスクライブのインフラストラクチャ](#)
- [パブリッシュ / サブスクライブの概念](#)
- [例](#)

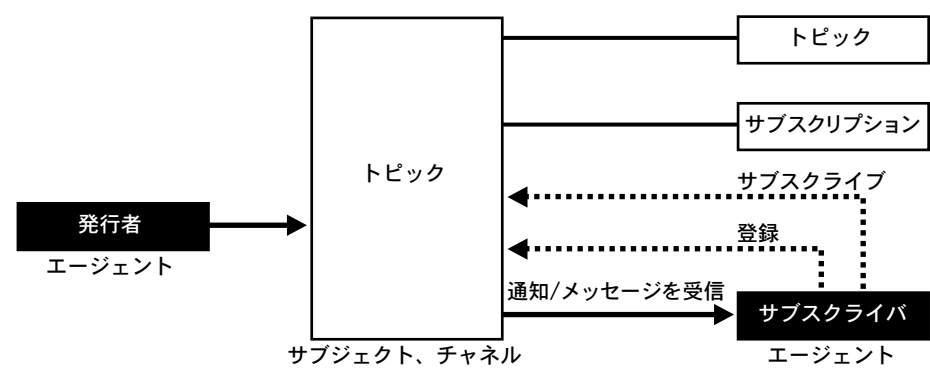
パブリッシュ / サブスクライブの概要

ネットワーキング・テクノロジー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・キューイングまたはパブリッシュ / サブスクライブといった特徴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たりしなくても、メッセージを発行できる送信アプリケーション（発行者）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、発行者とサブスクライバの間に介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。介在するこのエンティティが、サブジェクトまたはチャンネルを表すキューです。図 14-1 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 14-1 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、キューに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、発行者は様々なキューにメッセージを転記します。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、次に、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 / サブスクライブのインフラストラクチャ

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インフラストラクチャおよび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・イベントを発行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 13 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： 『Oracle8i 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、情報を取り出す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：『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

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、次のような特徴があります。

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、発行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

発行者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があります。

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バ・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、エージェントは発行者またはサブスクライバの電子的な代理人です。

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・アプリケーション論理があります。単一のエージェントにか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かわって動作することもできます。

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（ADT）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、重要な情報発行元を表します。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発行します。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能動的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、つまり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。

メッセージの発行

発行者は、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発行します。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、メッセージの発行を表します。

ルール・エンジン

指定されたキューにメッセージが転記または発行されると、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、発行されたメッセージに合う一連の候補となるルールを取り出します。

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。

転記

キューは、登録されたすべてのクライアントに対して、発行された該当メッセージを通知します。この概念は転記と呼ばれます。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、登録されたクライアントすべてにメッセージを転記します。

メッセージの受信

サブスクライバは、次のメカニズムのどれかを介して、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に一致する場合、転記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、転記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り出します。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。

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 が連携してどのようにパブリッシュ / サブスクライブが実装されるかを示します。

- 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・イベントで発行されるメッセージを待機します。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、次のステップとして、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、Oracle コール・インタフェース (OCI) を使用します。次のコードは、登録に必要なステップを実行します。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ステップは、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 IV 部

特殊アプリケーションの開発

第 IV 部に含まれる章は、次のとおりです。

- 第 15 章「PL/SQL を使用した Web アプリケーションの開発」
- 第 16 章「Oracle XA でのトランザクション・モニターの操作」

PL/SQL を使用した Web アプリケーションの開発

Java および 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数多くの機能があります。

PL/SQL を使用した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。パッケージの詳細および使用例は、『Oracle8i PL/SQL 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。

メールの送信

PL/SQL プログラムまたはストアド・プロシージャからメールを送信するには、UTL_SMTP パッケージを使用します。

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。UTL_TCP パッケージへのコール結果を使用します。

TCP/IP 接続を使用した作業

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み込みまたは書き込みを行うには、UTL_TCP パッケージを使用します。

HTTP URL の内容の取出し

HTTP URL のコンテンツを取り出すには、UTL_HTTP パッケージを使用します。コンテンツは、通常、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。

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、Oracle Applilcation Server (OAS) や WebDB などの Oracle Web ゲートウェイで提供されています。問合せの結果を HTML 表にフォーマットしたり、イメージ・マップを作成したり、HTTP Cookie の設定および取得を行ったり、CGI 変数の値をチェックしたりできます。また、PL/SQL プログラムを使用して、その他の一般的な Web 操作を組み合わせることもできます。

詳細は、Web ゲートウェイに付属の PL/SQL Web Toolkit ドキュメントを参照してください。

Web ページ (PL/SQL サーバー・ページ) への PL/SQL コードの埋込み

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 サーバー・ページ (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 ページを作成でき、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 の内容を 1 行ずつ書いていくよりはるかに便利です。

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 サーバー・ページを使用した Web ベースのソリューションを実装するには、次の手順で行います。

- ソフトウェア構成を選択する
- PL/SQL サーバー・ページにコードおよびコンテンツを書き込む
- PL/SQL サーバー・ページをストアド・プロシージャとしてデータベースにロードする

ソフトウェア構成を選択する

PL/SQL サーバー・ページを開発および配置するには、PL/SQL Web ゲートウェイの他に、Oracle Server のリリース 8.1.6 以降が必要です。現在、Web ゲートウェイは、Web DB 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 サーバー・ページ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 を作成する完全なストアド・プロシージャを書き込みます。

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作成する 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便利です。
- 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 サーバー・ページを使用すると、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 サーバー・ページには、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。

PL/SQL サーバー・ページに、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、Java Server Pages (JSP) と同ジスクリプト・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) と似た構文を使用していますが、同一ではないので、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 サーバー・ページにコードおよびコンテンツを書き込む

既存の Web ページまたは既存のストアード・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 サーバー・ページのファイルの拡張子は、.psp である必要があります。

PL/SQL サーバー・ページには、PSP 指示句、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 サーバー・ページとしてコンパイルすると、まったく同じ HTML ファイルを出力するストアード・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、PSP 指示句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、指示句および宣言は、ともにファイルの最初の方に配置することをお勧めします。

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 について理解していて、コードの作成をすぐに開始できる場合は、これらの項を省略してもかまいません。

スクリプト言語の指定

ファイルを PL/SQL サーバー・ページとして識別するには、ファイルの任意の場所に `<%@ page language="PL/SQL" %>` 指示句を含めます。この指示句は、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 ページを取り出す 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 サーバー・ページへのパラメータ渡しを設定するには、

`<%@ plsql parameter="..." %>` 指示句を含めます。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="..."` 属性を指示句に含めます。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="..."` 属性を指示句に含めます。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したりする必要がある場合があります。後の項で説明するように、PSP デリミタに IF/THEN 論理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 ゲートウェイはファイルを 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="..." %>` 指示句を含めます (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 サーバー・ページは、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 要求を作成できるプログラムによって取り出されたり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 ゲートウェイは、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="..." %>` 指示句を含めます。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コード化を指定します。

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。ユーザーがブラウザでデータを適切に表示するには、使用するブラウザで同じコード化を選択する必要がある場合があります。

たとえば、EUC コード化されたデータ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 コード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。PSP 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、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 サーバー・ページです。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。

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 サーバー・ページは、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。

別の名前を付けるには、`<%@ page procedure="..." %>` 指示句を含めます。

別のファイルの内容の挿入

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 スクリプティング・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="..." %>` 指示句を含めます。ファイルは、ストアード・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 スクリプティング・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指示句は必要ありません。

PSP ロード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・パ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。

スクリプトで使用するための変数宣言

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。すべての通常の 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 ファイルがストアード・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。

スクリプトで実行される文の指定

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、完全なもののでも、IF-THEN-ELSE 文の IF 部分のようなコンパウンド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。

式の結果のスクリプトでの置換え

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 TO_CHAR ファンクションに渡します。

`<%= %>` デリミタ間の内容は、HTML.PRIN ファンクションによって処理されます。これは、先行または後続ブランクをすべて切り捨て、リテラル文字列を引用符で囲むように要求します。

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、順序 `\'` 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 ロードで変更しなくても PSP ファイルに含めることができます。順序 `%>` を含めるには、エスケープ順序 `%\>` を指定します。順序 `<%` を含めるには、エスケープ順序 `<%\` を指定します。

問合せからの結果セットの取出し

データベースから 1 つの行を取り出すには、カーソルをオープンし、そのカーソルにデータをフェッチし、カーソル変数から列を取り出します。

複数行を戻す問合せ結果を表示するために、結果セットの各行を反復できます。

表全体を印刷するには、PL/SQL Web Toolkit から OWA_UTIL.TABLEPRINT プロシージャをコールします。

ユーザー入力に基づくデータベースの更新

挿入、更新および削除の各操作を、PL/SQL サーバー・ページ内で実行できます。結果が HTML で戻されることが予想されるプログラムでは、そのようなページには、操作が正常に実行されたこと、または更新された状態を示す出力が含まれています。

PL/SQL サーバー・ページのコード化のヒント

異なる PL/SQL サーバー・ページ間でプロシージャ、定数および型を共有するには、(PSP 方法を使用するのではなく) それらをデータベース内の個別のパッケージの中にコンパイルします。

すべての指示句および宣言を PL/SQL サーバー・ページ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 サーバー・ページ要素の構文

これらの要素の例については、15-12 ページの「[PL/SQL サーバー・ページの例](#)」を参照してください。

ページ指示句

次の PL/SQL サーバー・ページの特徴を指定します。

- 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別の PSP ファイル

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"] %>
```

プロシージャ指示句

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" %>
```

パラメータ指示句

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。パラメータは、通常、HTML フォームから CGI プロトコルを使用して渡されます。

構文

```
<@ plsql parameter="parameter name" [type="PL/SQL type"] [default="value"] %>
```

インクルード指示句

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。ネーム・リゾルバ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・プロシージャが実行されたときに反映されません。

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。この要素は、通常、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。この要素は、通常、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 ブロックの大力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 またはその他の指示句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わせて、単一の PL/SQL 式を指定します。結果は、ストアード・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として置き換えられます。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 %>
```

PL/SQL サーバー・ページをストアード・プロシージャとしてデータベースにロードする

1 つ以上の PSP ファイルをストアード・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 つのストアード・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 ステップで行われます。

```
loadpsp [ -replace ] -user username/password[@connect_string]  
[ include_file_name ... ] psp_file_name ...
```

ストアード・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。ストアード・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 サーバー・ページの名前（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。

使用例を次に示します。

```
loadpsp -replace -user scott/tiger@WEBDB timestamp.inc banner.inc display_order.psp
```

この例は、次のことを示しています。

- ストアード・プロシージャは、データベース WEBDB で作成されます。ストアード・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。
- timestamp.inc および banner.inc は、ボイラープレート・テキストおよびスクリプト・コードを含むファイルで、.psp ファイルに含まれています。
- 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、対応するストアード・プロシージャは DISPLAY_ORDER という名前です。

URL を介した PL/SQL サーバー・ページの実行

PL/SQL サーバー・ページが一度ストアド・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 を取り出すことによって、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、POST メソッドまたは GET メソッドのいずれかの CGI メカニズム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、スペースは %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 には表示されません。

```
http://sitename/schemaname/pspname
```

METHOD=GET 形式は、デバッグに便利です。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、まったく同じパラメータを渡せます。

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 に表示するべきで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、および次の訪問ページに渡される CGI 変数の中に残ります）。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 サーバー・ページの例

この項では、非常に単純な PL/SQL サーバー・ページから始め、慣れるに従って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。

各ステップを行いながら、15-11 ページの「[PL/SQL サーバー・ページをストアド・プロシージャとしてデータベースにロードする](#)」および 15-12 ページの「[URL を介した PL/SQL サーバー・ページの実行](#)」に記載し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。そのためには、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。

```
<@% page language="PL/SQL" %>
<%@ 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り出せます。
- 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。

```
<@% page language="PL/SQL" %>  
<@% 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。

```
<@% page language="PL/SQL" %>
```

```

<@ 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限り、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。

```

<@% page language="PL/SQL" %>
<@ plsql procedure="show_catalog_partial" %>
<@ plsql parameter="minprice" type="NUMBER" 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>
```

前述のように、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り出し、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・サイズを大きくしたり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。

```
<@% page language="PL/SQL" %>
<@% plsql procedure="show_catalog_highlighted" %>
<@% plsql parameter="minprice" type="NUMBER" 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 := '#CCCCC';
  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 サーバー・ページ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したりできます。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。

PL/SQL サーバー・ページのデバックに関する問題

PSP 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 指示句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 (PSP で必要) で囲む必要がある場合もあります。
 - PSP 指示句内の誤りは、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、指示句をチェックします。
 - PSP 属性名は、大 / 小文字を区別します。ほとんどは、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わせで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指示句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、スクリプトおよび 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。
- ページに渡されるものを正確に確認するには、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、または 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 ファイルは等価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 に切り換えることができます。

PL/SQL サーバー・ページを使用したアプリケーションの商品化

PSP 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 表はデータを表示する場合に便利ですが、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、ビューアは空白のページを表示している必要があります。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ウ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、白のバックグラウ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。
 - バックグラウンド・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ウンド・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たのであれば、表示されているページがどのステップなのかを示します。手順を継続するか、前のステップに戻るか、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、インクルード指示句を使用して埋め込んだ、標準のリンクが使用されている場合があります。
- 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の方が効果的です。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、まったく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こともありますが、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）有用性が無料でチェックできます。

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、Oracle8 Server 以外のトランザクション・マネージャでグローバル・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 Oracle8 Server 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。

- [X/Open Distributed Transaction Processing（DTP）](#)
- [XA および 2 フェーズ・コミット・プロトコル](#)
- [トランザクション処理モニター（TPM）](#)
- [動的登録および静的登録のサポート](#)
- [Oracle XA ライブラリ・インタフェース・サブルーチン](#)
- [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](#)
- [XA アプリケーションのトラブルシューティング](#)
- [XA の一般的な問題および制限事項](#)
- [Oracle XA サポートの変更](#)

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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。
- 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 Distributed Transaction Processing (DTP)

X/Open DTP アーキテクチャは、複数のアプリケーション・プログラムが複数の（異なる）リソース・マネージャ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。アプリケーション・プログラムとリソース・マネージャ間の作業を調整し、グローバル・トランザクションを実現します。

図 16-1 に、X/Open DTP モデルの例を示します。

リソース・マネージャ（RM）は、障害発生後に通常の状態に戻ることができる共有およびリカバリ可能なリソースを制御します。たとえば、Oracle8 Server は RM であり、障害発生後に REDO ログおよび UNDO 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。

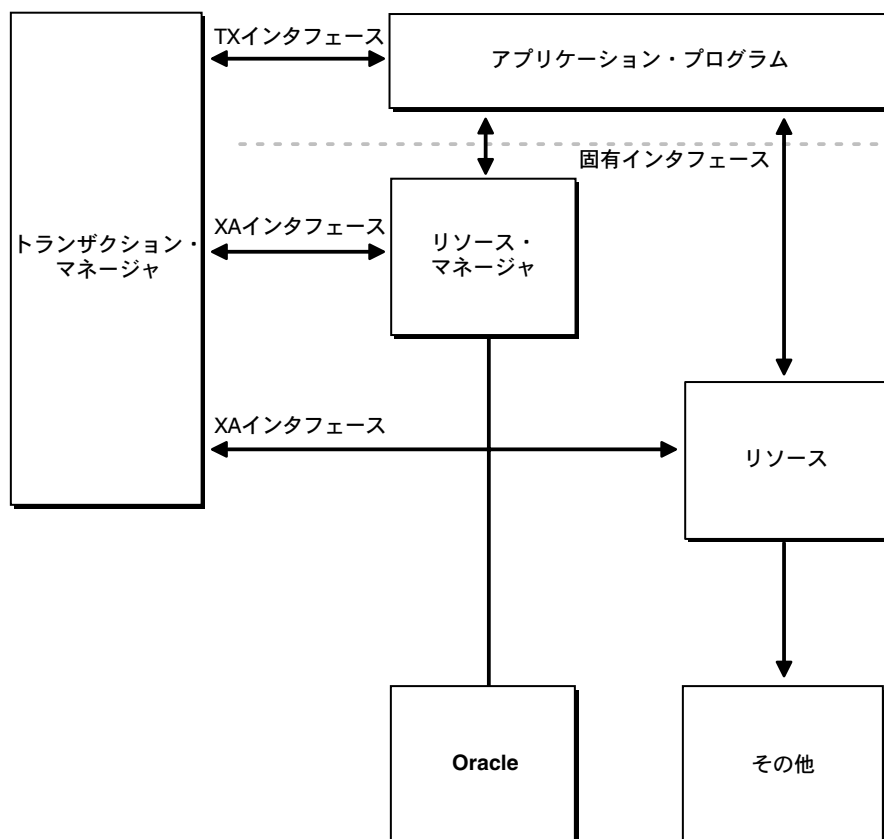
トランザクション・マネージャ（TM）は、トランザクションの境界を指定するためのアプリケーション・プログラム・インタフェース（API）を提供し、コミットおよびリカバリ手順を管理します。

通常、Oracle8 Server は Oracle8 Server 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、Oracle8 Server 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、トランザクション処理モニター (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 に関する情報を基に、Oracle XA ライブラリ・サブルーチンを使用して Oracle8 Server にトランザクションの処理方法を指示します。この項の後の方で、XA サブルーチンとその説明をリストしています。

アプリケーション・プログラム (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 の固有インタフェース (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、トランザクション・マネージャを使用して、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 16-1 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、トランザクション処理モニターに付属のドキュメントを参照してください。

必須のパブリック情報

Oracle は、リソース・マネージャとして、次の情報を発行する必要があります。

xa_switch_t 構造体	Oracle Server では、静的登録のための xa_switch_t 構造体名は <code>xaosw</code> です。動的登録の xa_switch_t 構造体名は <code>xaoswd</code> です。これらの構造体には、リソース・マネージャのエントリ・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 Server リソース・マネージャ名は <code>Oracle_XA</code> です。
クローズ文字列	<code>xa_close()</code> 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	<code>xa_open()</code> で使用されるオープン文字列の書式の詳細は、16-10 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のと同じです。sql1lib を使用していない場合は、必ず <code>\$ORACLE_HOME/lib/xaons1.o</code> にリンクしてください。
要件	購入し、インストールした分散データベース・オプションです。

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時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に対して肯定的な返答をした場合にのみ、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)

トランザクション処理モニター (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・サーバーやリソース・マネージャなどのように、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 のトランザクション・マネージャ (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、分散アプリケーション・プログラムで 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 (または他のリソース・マネージャ) 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

Oracle8 Server 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 Oracle 8.0 以降である必要があります。そうでなければ、静的登録しか使用できません。

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 Oracle8 Server 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_open	リソース・マネージャへ接続します。
xa_close	リソース・マネージャから切断します。
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・プロトコルの最初のフェーズです。
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 2 番目のフェーズです。
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り出します。
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ル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() (バージョン 8 OCI) を使用してバージョン 7 OCI 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() を使用してバージョン 7 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 の失敗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 を返し、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、Oracle8 Server 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](#) (16-9 ページ)
- [アプリケーション開発者の責任](#) (16-10 ページ)
- [xa_open 文字列の定義](#) (16-10 ページ)
- [プリコンパイラと OCI のインタフェース](#) (16-17 ページ)
- [XA を使用したトランザクション制御](#) (16-20 ページ)
- [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](#) (16-23 ページ)
- [XA ライブラリ・スレッド・セーフティ](#) (16-24 ページ)

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。
16-10 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。

Oracle Server リリース 8.0 の場合：

xa_open 文字列で指定したすべての Oracle Server ユーザーに対して、DBA_PENDING_TRANSACTIONS ビューの SELECT 権限を付与します。

Oracle Server 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 Server アカントに対して、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リソース・マネージャを 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、Oracle8 Server 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 のドキュメントを参照して、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、16-10 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： 16-17 ページの「[プリコンパイラと OCI のインタフェース](#)」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](#)

- 必須のフィールド
- オプションのフィールド

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

<i>Acc</i>	ユーザー・アクセス情報を指定します。
<i>P</i>	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。
<i>P//</i>	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、『Oracle8i 管理者ガイド』を参照してください。
<i>user</i>	有効な Oracle Server アカウントです。
<i>password</i>	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。

前述したように、*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

<i>SesTm</i>	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非アクティブ状態でいられる最長時間を指定します。
--------------	---

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長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*

DB

データベース名を指定します。

<i>db_name</i>	<p>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。</p> <p>Oracle プリコンパイラのデフォルト・データベースのみを使用する (SQL 文で AT 句を使用しない) アプリケーション・プログラムでは、オープン文字列の <code>DB=db_name</code> 句を省略する必要があります。</p> <p>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、<code>DB=db_name</code> 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。</p> <p>Oracle7 の OCI プログラムは、正しい <code>lda_def</code> (サービス・コンテキストと等価) を取得するために、<code>sqlld2()</code> 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、<code>xaoSvcCtx</code> 関数をコールして、<code>OCISvcCtx</code> サービス・コンテキストを取得する必要があります。</p> <p><i>db_name</i> は <i>sid</i> 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。むしろ、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アプリケーション・プログラムで使用される名前とを対応付けます。<i>sid</i> は、TPM アプリケーション・サーバーの環境変数 <code>ORACLE_SID</code> か、オープン文字列の <code>Net8</code> (以前の <code>SQL*Net</code>) 句で指定した <i>sid</i> から設定されます。<code>Net8</code> 句については、この項で後述します。</p> <p>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。<code>DBA</code> にとっては、グループ名と <i>db_name</i> 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。</p>
----------------	---

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`

<code>LogDir</code>	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。
<i>log_dir</i>	<p>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、<code>ORACLE_HOME</code> が設定されている場合 <code>\$ORACLE_HOME/rdbms/log</code> で、設定されていない場合は現行のディレクトリです。</p>

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、16-31 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`

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`

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_` キャッシュするオープン・カーソルの数を示します。
`open_cursors`

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： 『Oracle8i Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド』を参照してください。

SqlNet=db_link

SqlNet	Net8（以前の SQL*Net）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、TWO-TASK 環境変数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 Server 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 Net8 文字列を使用するには、パイプ・ドライバを使用します。

使用例を次に示します。

```
SqlNet=localsid1
```

パラメータは次のとおりです。

localsid1 Net8 tnsnames.ora ファイルで定義されている別名です。

Net8 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
```

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
```

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 のインタフェース

この項では、プリコンパイラおよび Oracle コール・インタフェース (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。明示的なカーソルは、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・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われる DB=*db_name* フィールドが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と、デフォルトの接続が指示されます。1つのプロセスでは1つのデフォルト接続しか使用できません。

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=*db_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、トランザクション・マネージャ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・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 以降では、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勧めしません。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、リソース・マネージャ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、リソース・マネージャ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 Server リソース・マネージャ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、このリソース・マネージャの *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、このリソース・マネージャの *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、このリソース・マネージャのサーバー・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、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

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、トランザクション・マネージャによって定義された API をコールします。

トランザクション・マネージャは、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。

<code>tx_open</code>	リソース・マネージャにログインします。
<code>tx_close</code>	リソース・マネージャからログアウトします。
<code>tx_begin</code>	新規トランザクションを開始します。
<code>tx_commit</code>	トランザクションをコミットします。
<code>tx_rollback</code>	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 Server がリソース・マネージャである場合は、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。

例に示されているように、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少なくとも 1 つ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、次を行う必要があります。

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 Net8 ドライバを使用します。このため、`ORACLE_HOME` および `ORACLE_SID` 環境変数を正しく定義して、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 ベンダーのドキュメント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 RELEASE WORK`（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()` はでした。

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 16-1 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 16-1 TPM 用に置き換えるコマンド

正規の Oracle コマンド	TPM 関数
CONNECT <i>user/password</i>	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、リソース・マネージャ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、各プロセスにはリソース・マネージャへの接続がそれぞれ独自にあり、それぞれが独立したリソース・マネージャ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、各スレッドにはリソース・マネージャとの自律型接続があり、プライベートなリソース・マネージャ表をメンテナンスします。このプライベートなリソース・マネージャ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す、またスレッドが終了したときは（異常終了であっても）割り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ら、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ておくことをお勧めします。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限り、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。これは、通常は、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、および保留中の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、その原因はオープン文字列が正しくなかったことか、Oracle Server インスタンスの検索に失敗したことか、またはログイン認可に失敗したことかを示します。

トレース・ファイルの名前は次のとおりです。

`xa_db_namdate.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。これは、通常、Oracle 開発者用の機能です。
- 0x4 Oracle コール・インタフェース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。これは、通常、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入れることができます。

- 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・ファイルは現行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。リソース・マネージャ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。リソース・マネージャ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・リソース・マネージャは、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: ユーザー名 / パスワードが無効です。ログオンは拒否されました。
1032.12345.2:  xaolgn:  XAER_INVALID; 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」はリソース・マネージャ ID、xaolgn はモジュール名、XAER_INVALID 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 Server 情報です。

インダウト・トランザクションまたは保留中のトランザクション

インダウト・トランザクションまたは保留中の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トランザクション・マネージャは、インダウト・トランザクションまたは保留中の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 のドキュメントを参照してください。

Oracle Server の SYS アカウント表

Oracle Server の SYS アカウントには 4 つの表があり、正規の Oracle Server 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_namexa.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、トランザクション処理モニター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失敗し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、アクティブなすべてのグローバル・トランザクションが提供されます。

XA の一般的な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限り、データベース・リンク経由で他の Oracle Server データベースにアクセスできます。

- マルチスレッド・サーバー構成を使用する必要があります。
これは、トランザクション処理モニター (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 O/S ネットワーク接続は、Oracle Server 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 または Net8 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 Server 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 Server 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 Server 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。

マルチスレッド・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 Parallel Server オプション

失敗したトランザクションは、Oracle Parallel Server 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トランザクション・マネージャに責任があるため、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 Server 固有の文をアプリケーションに入れしないでください。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 で等価の 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 で等価の 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 コマンドを使用しないでください。つまり、EXEC SQL COMMIT WORK RELEASE または EXEC SQL ROLLBACK WORK RELEASE を使用しないでください。

XA に関するその他の問題

Oracle XA に関しては、次の情報についても注意してください。

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 Server 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 Parallel Server 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。つまり、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 Server 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。つまり、疎結合トランザクション・ブランチを使用する利点は、同時実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）。欠点は、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。つまり、XA の 1 フェーズ最適化を使用できません。表 16-2 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 16-2 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	読取り専用最適化 [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]
シリアライズ化	データベース・コール	なし

対応付けの移行

Oracle Server 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、トランザクション・マネージャが、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： 16-29 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 Server 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。

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 の Oracle Server 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 Server 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 Server 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 Oracle Server リリース 7.3 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。

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 Parallel Server オプションとともに使用可能
- Oracle Parallel Server 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 パラメータも削減できます。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詳細は、トランザクション処理モニターに付属のドキュメントを参照してください。特に、トランザクションが停止されると、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 Server 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： 16-8 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 Server 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 Server の場合、単一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ある場合でも、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： 16-9 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 Parallel Server 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 Parallel Server 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 Parallel Server オプションと Oracle XA ライブラリをいっしょに使用できませんでした）。この制限がなくなりました。Oracle Parallel Server 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 Parallel Server 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を Oracle Parallel Server の任意のインスタンスからリカバリできます。保留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 Server 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、トランザクション処理モニターなどの外部のトランザクション・マネージャ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 Server は従属的に動作し、トランザクション・マネージャが発行する 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 Server 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 */
```

これに対して、Oracle8 Server 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終了（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 Server 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、Oracle Server リリース 7.3 への接続時にのみ使用します。

- GPWD

Oracle8 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 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。

記号

.psp ファイル, 15-4
%ROWTYPE 属性, 9-7
 ストアド・ファンクションで使用, 9-8
%TYPE 属性, 9-7

数字

1 対 1 関連
 外部キーを使用, 4-10
1 対多関連
 外部キーを使用, 4-10
3 層システム, 11-64, 11-65, 11-66

A

ADD_POLICY プロシージャ, 11-38
AFTER トリガー
 監査, 12-35, 12-38
 指定, 12-7
 関連名, 12-16
ALL_ERRORS ビュー
 ストアド・プロシージャのデバッグ, 9-43
ALL_SOURCE ビュー, 9-43
ALTER CLUSTER コマンド, 2-6
 ALLOCATE EXTENT オプション, 5-17
ALTER INDEX コマンド, 2-6
ALTER SEQUENCE コマンド, 2-30
ALTER SESSION SET SCHEMA 文, 11-44
ALTER SESSION コマンド
 SERIALIZABLE, 7-25
 SET SCHEMA, 11-19
ALTER TABLE コマンド, 2-6, 2-9
 DISABLE ALL TRIGGERS オプション, 12-32

 DISABLE 整合性制約オプション, 4-21
 DROP 整合性制約オプション, 4-24
 ENABLE ALL TRIGGERS オプション, 12-31
 ENABLE 整合性制約オプション, 4-21
 INITTRANS パラメータ, 7-25
 整合性制約の定義, 4-18
ALTER TRIGGER コマンド
 DISABLE オプション, 12-31
 ENABLE オプション, 12-31
ALTER 権限, 11-22
ANSI SQL92
 FIPS フラグ付け, 7-2
ANSI (米国規格協会)
 ANSI 互換のロック, 7-17
AUTHENTICATION_DATA 属性, 11-44
AUTHENTICATION_TYPE 属性, 11-44
AUTONOMOUS_TRANSACTION, 7-32

B

BEFORE トリガー
 指定, 12-7
 関連名, 12-16
 導出列値, 12-49
 複雑なセキュリティ認可, 12-48
BY REF 句, 10-28

C

CACHE オプション
 CREATE SEQUENCE コマンド, 2-34
CASCADE CONSTRAINTS オプション, 11-32
CASCADE オプション
 整合性制約, 5-18
CATPROC.SQL スクリプト, 12-4, 13-2

- CC 日付書式, 3-12
- CGI 変数, 15-2
- CHARSETFORM プロパティ, 10-24
- CHARSETID プロパティ, 10-24
- CHARTOROWID 関数, 3-25
- CHAR データ型, 3-5
 - 列の長さ, 3-6
 - 列を長くする, 2-9
- CHECK 制約
 - 使用方法, 4-15 ~ 4-20
 - トリガー, 12-40, 12-47
- CLIENT_INFO 属性、USERENV, 11-43
- COMMIT コマンド, 7-5
- cookie, 15-2
- CREATE CLUSTER コマンド, 2-6, 5-15
 - HASH IS オプション, 5-20
 - HASHKEYS オプション, 5-20
 - ハッシュ・クラスタ, 5-19
- CREATE CONTEXT 文, 11-52
- CREATE INDEX コマンド, 2-6, 5-5
 - ON CLUSTER オプション, 5-16
- CREATE PACKAGE BODY コマンド, 9-15
- CREATE PACKAGE コマンド, 9-15
- CREATE ROLE 文, 11-23
- CREATE SCHEMA コマンド, 2-37
 - 必要な権限, 2-37
- CREATE SCHEMA 文, 11-19
- CREATE SEQUENCE コマンド
 - CACHE オプション, 2-29, 2-34
 - NOCACHE オプション, 2-34
 - 例, 2-34
- CREATE SESSION 文, 11-19
- CREATE TABLE コマンド, 2-3, 2-4, 2-6
 - CLUSTER オプション, 5-16
 - INITTRANS パラメータ, 7-25
 - 整合性制約の定義, 4-17
- CREATE TRIGGER コマンド, 12-3
 - REFERENCING オプション, 12-17
- CREATE VIEW コマンド, 2-15
 - OR REPLACE オプション, 2-18
 - WITH CHECK OPTION, 2-15, 2-19
- CURRENT_SCHEMA 属性、USERENV, 11-44
- CURRENT_USER 属性、USERENV, 11-43
- CURRVAL 疑似列, 2-31
 - 制限, 2-32

D

- DATE データ型, 3-8
 - 世紀, 3-10
 - データ変換, 3-25
- DB_DOMAIN 属性、USERENV, 11-44
- DBA_ERRORS ビュー
 - ストアド・プロシージャのデバッグ, 9-43
- DBA_ROLE_PRIVS ビュー, 11-9
- DBA_SOURCE ビュー, 9-43
- DBMS_LOCK パッケージ, 7-20
- DBMS_OBFUSCATION_TOOLKIT パッケージ, 11-72
- DBMS_RLS パッケージ, 11-38, 11-46
- DBMS_SESSION パッケージ
 - SET_CONTEXT プロシージャ, 11-52
 - SET_ROLE プロシージャ, 11-10
- DBMS_SQL パッケージ
 - DESCRIBE, 8-17
 - RETURNING 句, 8-18
 - SET_ROLE プロシージャ, 11-11
 - クライアント側プログラム, 8-17
 - システム固有の動的 SQL との違い, 8-12
 - 「動的 SQL」を参照
 - バルク SQL, 8-17
 - 複数行の更新および削除, 8-18
 - 利点, 8-17
- DDL 文
 - パッケージ状態, 9-17
- DEBUG_EXTPROC パッケージ, 10-48
- DELETE 権限, 11-22
- DELETE コマンド
 - 参照整合性に関するトリガー, 12-43, 12-44
 - データ整合性, 7-11
 - 列値およびトリガー, 12-16
- DESC 関数, 5-8
- DETERMINISTIC キーワード, 9-63
- dictionary_obj_owner_list イベント属性, 13-3
- dictionary_obj_owner イベント属性, 13-3
- dictionary_obj_type イベント属性, 13-3
- DML_LOCKS パラメータ, 7-11
- DROP CLUSTER コマンド, 5-18, 5-20
- DROP INDEX コマンド, 5-6
 - 必要な権限, 5-6
- DROP ROLE 文, 11-27
- DROP TABLE コマンド, 2-10
- DROP TRIGGER コマンド, 12-30
- DROP_POLICY プロシージャ, 11-38

DTP アーキテクチャ, 16-2

E

ENABLE_POLICY プロシージャ, 11-38

EXECUTE 権限, 11-22

EXTERNAL_NAME 属性、USERENV, 11-44

extproc プロセス, 10-33

F

FIPS フラグ付け

対話型 SQL 文, 7-2

FIXED_DATE 初期化パラメータ, 3-9

FOR EACH ROW 句, 12-12

G

GRANT OPTION, 11-30

GRANT コマンド

ADMIN OPTION, 11-28

オブジェクト権限, 11-21, 11-30

最後の DDL 時刻, 11-33

システム権限, 11-28

有効時, 11-34

H

HEXTORAW 関数, 3-25

HTML

PL/SQL からの取出し, 15-2

PSP ファイル内での表示, 15-5

HTP パッケージおよび HTF パッケージ, 15-2

HTTP URL, 15-2

I

IN OUT パラメータ・モード, 9-6

INDEX 権限, 11-22

INDICATOR プロパティ, 10-23

INITTRANS パラメータ, 7-25

INSERT 権限, 11-22

INSERT コマンド

読取り一貫性, 7-11

列値およびトリガー, 12-16

instance_num イベント属性, 13-3

INSTEAD OF トリガー, 12-7

NESTED TABLE のビューの列に対する, 12-16

IN パラメータ・モード, 9-6

is_alter_column イベント属性, 13-3

is_creating_nested_table イベント属性, 13-4

is_drop_column イベント属性, 13-4

is_servererror イベント属性, 13-4

ISDBA 属性、USERENV, 11-43

ISOLATION LEVEL

SERIALIZABLE, 7-25

変更, 7-25

J

Java

JDBC の概要, 1-26

JPublisher を使用したラッパー・クラスの生成,
1-33

PL/SQL, 1-39

RDBMS における, 1-29

SQLJ の概要, 1-30

コール仕様を介したコール・メソッド, 10-3

データベースへのロード, 10-4

Java サーバー・ページ

PSP への変換, 15-3

Java スクリプト

PSP への変換, 15-3

JDBC

「Oracle JDBC」を参照

JPublisher, 1-35

JScript

PSP への変換, 15-3

L

Lightweight Directory Access Protocol (LDAP), 11-56

loadjava ユーティリティ, 1-36

loadpsp コマンド, 15-11

LOB データ型

OO4O でのサポート, 1-17

トリガー内での使用, 12-20

LOCK TABLE コマンド, 7-12

login_user イベント属性, 13-4

LONG RAW データ型, 3-18, 3-20

制限, 3-18

トリガー内での使用, 12-20

LONG データ型, 3-18

制限, 3-18

トリガー内での使用, 12-20
LOWER 関数, 5-8

M

MAX_ENABLED_ROLES パラメータ, 11-24, 11-27
MAXTRANS オプション, 2-6

N

NCHAR データ型, 3-2, 3-5
new 相関名, 12-16
NEXTVAL 疑似列, 2-31
制限, 2-32
NLS_DATE_FORMAT パラメータ, 3-8
NLSSORT 順序および索引, 5-8
NOCACHE オプション
CREATE SEQUENCE 文, 2-34
NOT NULL 制約
CHECK 制約, 4-16
使用する場合, 4-3
データの整合性, 4-20
NOWAIT オプション, 7-12
NUMBER データ型, 3-7
NVARCHAR2 データ型, 3-2, 3-5
n 層認証, 11-64, 11-65, 11-66

O

OAS, 15-2
OCI, 9-2
アプリケーション, 9-4
カーソルのクローズ, 7-10
カーソルの取消し, 7-10
概要, 1-7
構成要素, 1-8
利点, 1-7
ロールを使用可能にする, 11-17
OCI およびプリコンパイラ, 1-37
old 相関名, 12-16
OO4O
「Oracle Objects for OLE」を参照
OO4O での LOB のサポート, 1-17
OO4O でのオブジェクトのサポート, 1-17
OO4O におけるデータ・コントロール, 1-19
OPEN_CURSORS パラメータ, 7-9

OR REPLACE 句

パッケージ作成用, 9-16
ora_dictionary_obj_owner_list イベント属性, 13-3
ora_dictionary_obj_owner イベント属性, 13-3
ora_dictionary_obj_type イベント属性, 13-3
ora_grantee イベント属性, 13-3
ora_instance_num イベント属性, 13-3
ora_is_alter_column イベント属性, 13-3
ora_is_creating_nested_table イベント属性, 13-4
ora_is_drop_column イベント属性, 13-4
ora_is_servererror イベント属性, 13-4
ora_login_user イベント属性, 13-4
ora_privileges イベント属性, 13-4
ora_revokee イベント属性, 13-4
ora_server_error イベント属性, 13-4
ora_sysevent イベント属性, 13-4
ora_with_grant_option イベント属性, 13-5
ORA-21301 エラーの修正, 16-15
OraAQAgent オブジェクト, 1-17
OraAQMsg オブジェクト, 1-17
OraAQ オブジェクト, 1-16
OraBFILE オブジェクト, 1-18
OraBLOB オブジェクト, 1-18
Oracle Advanced Security, 11-19
Oracle Applilcation Server (OAS), 15-2
Oracle Data Control (ODC), 1-19
Oracle Internet Directory, 11-67
Oracle JDBC
OCI Driver, 1-26
Oracle の拡張機能, 1-27
Thin Driver, 1-26
サーバー・ドライバ, 1-27
ストアド・プロシージャ, 1-29
定義, 1-26
例, 1-27
Oracle Objects for OLE
C++ クラス・ライブラリ, 1-19
LOB およびオブジェクトのサポート, 1-17
オートメーション・サーバー, 1-12
オブジェクト・モデル, 1-12
概要, 1-11
データ・コントロール, 1-19
Oracle SQLJ
JDBC と比較した優位性, 1-32
サーバー内, 1-36
ストアド・プログラム, 1-36
設計, 1-31

- 定義, 1-30
- 例, 1-33
- Oracle エラー, 9-3
- Oracle コール・インタフェース
 - 「OCI」を参照
- Oracle 提供パッケージ, 9-17
- OraCLOB オブジェクト, 1-18
- OraDatabase オブジェクト, 1-14
- OraDynaset オブジェクト, 1-14
- OraField オブジェクト, 1-15
- OraMetaData オブジェクト, 1-15
- OraParamArray オブジェクト, 1-16
- OraParameter オブジェクト, 1-15
- OraServer オブジェクト, 1-13
- OraSession オブジェクト, 1-13
- OraSQLStmt オブジェクト, 1-16
- OS_ROLES パラメータ, 11-27
- OS_USER 属性, USERENV, 11-44
- OUT パラメータ・モード, 9-6
- OWA* パッケージ, 15-2

P

- Parallel Server
 - 順序番号, 2-30
 - 分散ロック, 7-11
- PARALLEL_ENABLE キーワード, 9-63
- Pascal コール規格, 10-8
- pcode
 - トリガー生成時, 12-28
- PCTFREE 記憶域パラメータ, 2-4 ~ 2-9
- PCTUSED 記憶域パラメータ, 2-6 ~ 2-9
 - 設定のためのガイドライン, 2-6
- PL/SQL, 9-2
 - Java, 1-39
 - RAISE 文, 9-44
 - Web Toolkit, 15-2
 - 依存性、ライブラリ・ユニット間, 9-28
 - オブジェクト, 1-5
 - カーソル変数, 9-38
 - 機能, 1-3
 - コードを隠すためのラッパー, 9-28
 - コンテキストの設定, 11-49
 - サーバー・ページ, 15-2 ~ 15-12
 - 再解析, 11-45, 11-46
 - サンプル・コード, 1-2
 - ソース・コードの隠蔽, 9-28

- 逐次再使用可能パッケージ, 9-70
- 動的 SQL を使用した起動, 8-7
- 動的に SQL 文を変更, 11-14
- トリガー本体, 12-14, 12-15
- パッケージ, 9-12
 - 表, 9-9
 - レコード, 9-9
 - ファンクション
 - RESTRICT_REFERENCES プラグマ, 9-66
 - オーバーロード, 9-70
 - 純粋度レベル, 9-69
 - 使用, 9-56
 - パラメータのデフォルト値, 9-59
 - 引数, 9-59
 - プログラム・ユニット, 9-2
 - 置き換えられたビュー, 2-18
 - 削除された表との関係, 2-10
 - 無名ブロック, 9-2, 11-10
 - ユーザー定義エラー, 9-44
 - 利点, 1-3
 - リモート・ストアド・プロシージャのコール, 9-54
 - 例外ハンドラ, 9-2
- PL/SQL コードの隠蔽, 9-28
- PL/SQL コードを隠すためのラッパー, 9-28
- PL/SQL でのプログラム・ユニット, 9-2
- privileges イベント属性, 13-4
- Pro*C/C++
 - アプリケーション開発の概要, 1-20 ~ 1-22
- Pro*COBOL
 - アプリケーション開発の概要, 1-23 ~ 1-25
- PRODUCT_USER_PROFILE 表, 11-13, 11-14, 11-17
- PROXY_USER 属性, 11-42, 11-44
- PSP
 - 「PL/SQL サーバー・ページ」を参照, 15-2
- PUBLIC ユーザー・グループ, 11-34

R

- RAISE_APPLICATION_ERROR プロシージャ, 9-43
 - リモート・プロシージャ, 9-46
- RAISE 文, 9-44
- RAWTOHEX 関数, 3-25
- RAW データ型, 3-20
- REFERENCES 権限, 11-22, 11-32
- REFERENCING オプション, 12-17
- REFRESH_POLICY プロシージャ, 11-38, 11-46

REF 列
 索引, 5-7

REMOTE_DEPENDENCIES_MODE パラメータ, 9-35

RENAME コマンド, 2-39, 2-40

REPARSE 文, 11-46

RESOURCE 権限, 11-19

RESTRICT_REFERENCES プラグマ
 構文, 9-66
 副作用の制御としての使用, 9-66

REVOKE コマンド, 11-29, 11-34

RM (リソース・マネージャ), 16-2

RNDS 引数, 9-66

RNPS 引数, 9-66

ROLE_SYS_PRIVS ビュー, 11-9

ROLE_TAB_PRIVS ビュー, 11-9

ROLLBACK コマンド, 7-6

ROW_LOCKING パラメータ, 7-11

ROWIDTOCHAR 関数, 3-25

ROWID データ型, 3-21
 移行, 3-23
 拡張 ROWID 形式, 3-21

ROWTYPE_MISMATCH 例外, 9-41

RR 日付書式, 3-11

RS ロック
 LOCK TABLE コマンド, 7-13

RX ロック
 LOCK TABLE コマンド, 7-13

S

SAVEPOINT コマンド, 7-6

Secure Sockets Layer (SSL) プロトコル, 11-68

SELECT 権限, 11-22

SELECT コマンド
 SELECT ... FOR UPDATE, 7-18
 読取り一貫性, 7-11

SEQUENCE_CACHE_ENTRIES パラメータ, 2-33

SERIALIZABLE オプション
 ISOLATION LEVEL, 7-25

SERIALIZABLE パラメータ, 7-11

SERIALLY_REUSABLE プラグマ, 9-71

server_error イベント属性, 13-4

SESSION_USER 属性、USERENV, 11-43

SET ROLE 文
 ALL EXCEPT オプション, 11-26
 ALL オプション, 11-26
 SET_ROLE と等価, 11-10

オペレーティング・システムのロール, 11-27

起動時, 11-16

使用禁止, 11-17

ロール使用の保護, 11-23

ロールに対応付けられた権限, 11-9

ロール・パスワード, 11-18

ロールを使用可能にする, 11-25

SET TRANSACTION コマンド, 7-8
 ISOLATION LEVEL 句, 7-25
 SERIALIZABLE, 7-25

SET_CONTEXT プロシージャ, 11-52

SET_ROLE プロシージャ, 11-10

SGA
 「システム・グローバル領域」を参照

SORT_AREA_SIZE パラメータ
 索引作成, 5-2

SQL*Loader
 索引, 5-2

SQL*Module
 アプリケーション, 9-4

SQL*Plus
 SET SERVEROUTPUT ON コマンド, 9-3

SHOW ERRORS コマンド, 9-41

コンパイル時エラー, 9-41

ストアド・プロシージャの起動, 9-49

非定型使用の制限, 11-12, 11-13

プロシージャのロード, 9-10

無名ブロック, 9-4

SQLStmt オブジェクト, 1-16

SQL 文
 実行, 7-2
 動的, 11-50
 トリガーでは使用できない文, 12-20
 トリガー本体内部, 12-15, 12-20
 非定型使用の制限, 11-12, 11-13

SRX ロック
 LOCK TABLE コマンド, 7-15

SYS_CONTEXT 関数
 USERENV 名前空間, 11-42
 アクセス制御, 11-58
 構文, 11-50
 セッション変数を格納, 11-51
 動的 SQL 文, 11-50
 パラレル問合せ, 11-51

SYSDATE 関数, 3-9

sysevent イベント属性, 13-4

SYS スキーマ, 11-52

S ロック

LOCK TABLE コマンド, 7-13

T

TCP/IP, 15-1
TERMINAL 属性、USERENV, 11-43
TM トランザクションマネージャ, 16-2
TO_CHAR 関数, 3-25
 CC 日付書式, 3-12
 RR 日付書式, 3-6
TO_DATE 関数, 3-8, 3-25
 RR 日付書式, 3-11
TO_NUMBER 関数, 3-25
TRUNCATE TABLE コマンド, 2-10
TRUNC 関数, 3-9
TRUST キーワード, 9-68

U

UPDATE_CHECK パラメータ, 11-38
UPDATE コマンド
 参照整合性に関するトリガー, 12-43, 12-44
 データ整合性, 7-11
 トリガー, 12-6, 12-18
 列値およびトリガー, 12-16
UPPER 関数, 5-8
URL, 15-2
USER_ERRORS ビュー
 ストアド・プロシージャのデバッグ, 9-42
USER_SOURCE ビュー, 9-43
USERENV 名前空間, 11-42, 11-43
USERENV 関数, 11-42
USER 関数, 4-5
UTL_HTTP パッケージ, 15-2
UTL_INADDR パッケージ, 15-1
UTL_SMTP パッケージ, 15-1
UTL_TCP パッケージ, 15-1
UTLLOCKT.SQL スクリプト, 7-21

V

VARCHAR2 データ型, 3-2, 3-5
 使用する場合, 3-5
 列の長さ, 3-6
VARCHAR データ型, 3-5

VBScript

PSP への変換, 15-3

W

WebDB, 15-2
web ページ
 動的, 15-2
WHEN 句, 12-13
 EXCEPTION の例, 12-18, 12-41, 12-47, 12-48
 PL/SQL 式の使用不可, 12-13
 相関名, 12-16
 例, 12-3, 12-12, 12-32, 12-41
WHERE 句、動的 SQL, 11-14
WITH CONTEXT 句, 10-28
with_grant_option イベント属性, 13-5
WNDS 引数, 9-66
WNPS 引数, 9-66

X

xa_open 文字列, 16-10
XA のオープン文字列, 16-10
XA ライブラリ, 16-1 ~ 16-36
XML
 PSP ファイルのドキュメント・タイプとして, 15-5
X/Open DTP アーキテクチャ, 16-2
X ロック
 LOCK TABLE コマンド, 7-16

あ

アクセス
 オブジェクト
 順序, 2-30
 スキーマ・オブジェクト, 11-30, 11-31
 トリガー, 12-3, 12-48
 リモート整合性制約, 4-14
 データベース, 11-28, 11-29
アクティブ・サーバー・ページ
 PSP への変換, 15-3
アクティブ・データ・オブジェクト
 PSP への変換, 15-3
アプリケーション
 One Big Application User モデル, 11-6, 11-7
 管理者, 11-8

- ストアド・プロシージャおよびパッケージのコード, 9-51
- セキュリティ, 11-7, 11-12
- データベース・ユーザー, 11-6
- 未処理例外, 9-46
- ロール, 11-5, 11-9
- アプリケーション・コンテキスト
 - USERENV 名前空間, 11-42
 - 安全なデータ・キャッシュとして, 11-47
 - 概要, 11-4, 11-40
 - 作成, 11-52
 - 述語を戻す, 11-48
 - 使用方法, 11-49
 - セキュリティ機能, 11-41
 - 設計原理, 11-45
 - 設定, 11-52
 - バージョン化, 11-51
 - バインド変数, 11-48
 - パフォーマンス, 11-56
 - パラレル問合せ, 11-51
 - ファイングレイン・アクセス・コントロール, 11-14, 11-18, 11-47
 - ポリシーでの使用, 11-52
 - 例, 11-53
- アプリケーション・セキュリティ
 - 概要, 11-4, 11-5
 - 使用に関する考慮点, 11-6
 - 制限事項, 11-15
 - 属性の指定, 11-41
 - 妥当性チェックを介する, 11-41
- 暗号化, 11-4, 11-72, 11-73
- 暗号化機能, 11-72

い

- 移行
 - ROWID 形式, 3-23
- 依存性
 - PL/SQL ライブラリ・オブジェクト間, 9-28
 - スキーマ・オブジェクト
 - トリガー管理, 12-20
 - ストアド・トリガーでの, 12-28
 - タイムスタンプ・モデル, 9-29
- 一意キー制約
 - NOT NULL 制約との組合せ, 4-4
 - コンポジット・キーおよび NULL, 4-7
 - 主キー制約との比較, 4-7

- 使用可能にする, 4-20
- 使用禁止にする, 4-20
- 使用する場合, 4-7
- 一時セグメント
 - 索引作成, 5-2
- イベント属性関数, 13-2
- イベント・トリガー, 11-59
- イベントの発行, 12-54 ~ 12-56, 13-1
 - トリガー, 12-54
- イメージ・マップ, 15-2

う

- 埋込み SQL, 9-2

え

- エクステンツ
 - 削除された表との関係, 2-10
 - 割当て, 5-17
- エラー
 - Oracle パッケージによって発生するアプリケーション・エラー, 9-43
 - エラーを伴うビューの作成, 2-16
 - ユーザー定義, 9-43, 9-44
 - リモート・プロシージャ, 9-46
- エンタープライズ・ユーザー, 11-19, 11-20

お

- オーバーロード
 - RESTRICT_REFERENCES を使用, 9-70
 - パッケージ・ファンクション, 9-70
- オブジェクト
 - GRANT OPTION, 11-30
 - 権限, 11-21
 - 権限の取消し, 11-31
 - 権限付与, 11-22, 11-30
- オブジェクト、スキーマ
 - 改名, 2-40
 - 情報のリスト, 2-41
- オブジェクト列、索引, 5-7
- オブジェクト、スキーマ
 - ネーム・リゾルバ, 2-38
- オペレーティング・システム
 - ロール, 11-27

か

- カーソル, 7-9
 - クローズ, 7-10
 - 最大数, 7-9
 - 取消し, 7-10
 - プライベート SQL 領域, 7-9
 - ポインタ, 9-38
- カーソル, 共有, 11-48
- カーソル変数, 9-38
 - 宣言およびオープン, 9-38
- 解析ツリー, 12-28
- 外部キー制約
 - 1 対 n 関連, 4-10
 - 1 対多関連, 4-10
 - NOT NULL 制約, 4-10
 - 一意キー制約, 4-10
 - 使用可能にする, 4-20, 4-27
 - 定義, 4-25, 4-26
- 外部結合, 2-26
 - キー保存表, 2-27
- 外部プロシージャ, 10-2
 - DEBUG_EXTPROC パッケージ, 10-48
 - データ型の指定, 10-16
 - パラメータの最大数, 10-50
 - 制限, 10-50
 - デバッグ, 10-47
- 拡張 ROWID 形式, 3-21
- 拡張性
 - 逐次再使用可能パッケージ, 9-70
- 仮想プライベート・データベース (VPD), 11-7,
11-14, 11-15, 11-17, 11-47
- 監査
 - n 層システム, 11-66, 11-71
 - One Big Application User によって解決される,
11-6
 - トリガー, 12-34

き

- キー
 - 一意
 - コンポジット, 4-7
 - 外部キー, 4-25
- キー保存表
 - 外部結合における, 2-27
 - 結合ビュー内, 2-22

- 記憶域パラメータ
 - PCTFREE, 2-9
 - PCTUSED, 2-9
- 疑似列
 - ビューの変更, 12-8
- キャッシュ
 - 順序キャッシュ, 2-33
 - 順序番号, 2-29
- 行
 - ROWID 内に示される, 3-22
 - サイズ, 2-2
 - 書式, 2-2
 - 整合性制約の違反, 4-20
 - ブロックにまたがる連鎖, 2-5
 - ヘッダー, 2-2
- 行トリガー
 - REFERENCING オプション, 12-17
 - UPDATE 文, 12-6, 12-18
 - タイミング, 12-7
 - 定義, 12-12
- 共有行排他ロック (SRX)
 - LOCK TABLE コマンド, 7-15
- 共有ロック (S)
 - LOCK TABLE コマンド, 7-13
- 行ロック
 - 手動ロック, 7-18

く

- 空白埋めデータ
 - パフォーマンス上の考慮点, 3-7
- クライアント・イベント, 13-7
- クライアントの再認証, 11-67, 11-68, 11-71
- クラスタ, 5-14 ~ 5-19
 - エクステンツの割当て, 5-17
 - 削除された表との関係, 2-10
 - パフォーマンス上の考慮点, 5-15

け

- 軽量セッション, 11-69
- 結合ビュー, 2-21
 - DELETE 文, 2-24
 - UPDATE 文, 2-24
 - キー保存表, 2-22
 - 変更, 2-23
 - 変更可能な場合, 2-21

マージ可能, 2-22
権限
 n 層システム, 11-66
 PUBLIC に付与された, 11-34
 オブジェクト, 11-22
 オブジェクトの改名, 2-40
 管理, 11-8, 11-21
 クラスタ作成, 5-16
 索引作成, 5-5
 シノニムの作成, 2-35
 手動によるロックの取得, 7-16
 順序の削除, 2-35
 順序の作成, 2-30
 順序の使用, 2-34
 順序の変更, 2-30
 ストアド・プロシージャでのカプセル化, 11-17
 ストアド・プロシージャの実行, 9-51
 整合性制約の作成, 4-18
 選択された列に対する, 11-31
 中間層, 11-70
 トリガー, 12-26
 トリガーの再コンパイル, 12-29
 トリガーの削除, 12-30
 トリガーの作成, 12-26
 取消し, 11-29, 11-31
 ビューの置換え, 2-18
 ビューの削除, 2-20
 ビューの作成, 2-17
 ビューの使用, 2-20
 表の削除, 2-11
 表の作成, 2-8
 表の変更, 2-10
 付与, 11-28, 11-30
権限受領者イベント属性, 13-3

リ

コール仕様, 10-3 ~ 10-50
コールバック, 10-44 ~ 10-46
コマンド、SQL
 制約チェックが発生した場合, 4-17
コンパイル時エラー, 9-41
コンボジット・キー
 NULL の制限, 4-16

さ

サービス・ルーチン, 10-35
 例, 10-35
再解析, 11-37, 11-45, 11-46, 11-52
再使用可能パッケージ, 9-70
索引
 PCTFREE の指定, 2-6
 SQL*Loader, 5-2
 一時セグメント, 5-2
 ガイドライン, 5-2
 権限, 5-5
 削除, 5-5
 削除された表との関係, 2-10
 作成, 5-5
 作成する場合, 5-2
 ファンクション・ベース, 5-6
 列の順序, 5-4
索引構成表, 6-1 ~ 6-20
削除
 クラスタ, 5-17
 索引, 5-5
 シノニム, 2-36
 順序, 2-34
 整合性制約, 4-24
 トリガー, 12-30
 パッケージ, 9-12
 ハッシュ・クラスタ, 5-20
 ビュー, 2-20
 表, 2-10
 プロシージャ, 9-11
 ロール, 11-27
作成
 クラスタ, 5-15
 索引, 5-5
 シノニム, 2-35
 順序, 2-34
 整合性制約, 4-2
 トリガー, 12-3, 12-20
 パッケージ, 9-15
 ハッシュ・クラスタ, 5-19
 ビュー, 2-15
 表, 2-3, 2-4
 複数オブジェクト, 2-37
参照整合性
 1 対 1 関連, 4-10
 1 対多関連, 4-10

- 外部キーの作成に必要な権限, 4-25
- 自己参照型制約, 12-44
- トリガー, 12-41, 12-45
- 分散データベース, 4-14

し

- 識別名, 11-20
- シグネチャ
 - PL/SQL ライブラリ・ユニットの依存性, 9-28
 - リモート依存性を管理するため, 9-30
- システム固有の動的 SQL
 - レコードへのフェッチ, 8-16
- システム・イベント, 13-1
 - クライアント・イベント, 13-7
 - 属性, 13-2
 - 追跡, 12-52, 13-1
 - リソース・マネージャ・イベント, 13-6
- システム・グローバル領域
 - 順序番号のキャッシュの保持, 2-33
- システム権限, 11-28, 11-29
- システム固有の動的 SQL
 - DBMS_SQL パッケージとの違い, 8-12
 - 「動的 SQL」を参照
 - パフォーマンス, 8-15
 - ユーザー定義型, 8-16
 - 利点, 8-13
- システム別 Oracle マニュアル, 12-4
 - PL/SQL ラッパー, 9-28
- 実行者権限
 - 動的 SQL, 8-8
- 実行者権限ストアド・プロシージャ, 11-10
- シノニム
 - 削除された表との関係, 2-10
 - 使用, 2-35 ~ 2-36
 - ストアド・プロシージャおよびパッケージ, 9-56
- 主キー制約
 - 一意キー制約との比較, 4-7
 - 主キーの選択, 4-6
 - 使用可能にする, 4-20
 - 使用禁止にする, 4-20
 - 複数列, 4-7
- 手動ロック, 7-11
 - LOCK TABLE コマンド, 7-12
- 順序
 - CURRVAL, 2-30, 2-32
 - NEXTVAL, 2-31

- Parallel Server, 2-30
- アクセス, 2-30
- 削除, 2-34
- 作成, 2-29, 2-34
- 番号のキャッシュ, 2-29
- 順序番号のキャッシュ, 2-33
- 初期化パラメータ, 2-29
- シリアライズ化の低減, 2-31
- 必要な権限, 2-30 ~ 2-35
- 変更, 2-30
- 純粋度レベル, 9-61
- 使用可能
 - トリガー, 12-31
 - ロール, 11-16
- 使用可能にする
 - 整合性制約, 4-20
- 使用禁止
 - トリガー, 12-31
 - ロール, 11-16
- 使用禁止にする
 - 整合性制約, 4-20
- 条件述語
 - トリガー本体, 12-14, 12-17
- 状態
 - パッケージ・オブジェクトのセッション, 9-17
- 初期化パラメータ
 - DML_LOCKS, 7-11
 - OPEN_CURSORS, 7-9
 - REMOTE_DEPENDENCIES_MODE, 9-35
 - ROW_LOCKING, 7-11
 - SERIALIZABLE, 7-11
- 書式マスク
 - TO_DATE 関数, 3-8
- シリアライズ可能トランザクション, 7-22
- 自律型トランザクション, 7-32 ~ 7-40
- 自律型ルーチン, 7-32
- 自律スコープ
 - 自律型トランザクションにおける, 7-32

す

- スキーマ
 - 一意, 11-19
 - デフォルト, 11-44
- スキーマに依存しないユーザー, 11-19, 11-20
- スクリプティング, 15-2
- スコープ、自律, 7-32

- ストアド・ファンクション, 9-5
 - 作成, 9-9
- ストアド・プロシージャ, 9-5
 - Web ページへの変換, 15-2
 - 格納, 9-9
 - 起動, 9-49
 - 権限, 9-51
 - 権限のカプセル化, 11-17
 - 作成, 9-9
 - 実行者権限, 11-10
 - シノニム, 9-56
 - 名前, 9-5
 - 名前のオーバーロード, 9-13
 - パラメータ
 - デフォルト値, 9-9
 - 引数の値, 9-52
 - 分散問合せの作成, 9-46
 - リモート, 9-53
 - リモート・オブジェクト, 9-54
 - 例外, 9-43, 9-44

せ

- 世紀, 3-10
 - 日付書式マスク, 3-9
- 制限
 - システム・トリガー, 12-25
- 整合性
 - 読取り専用トランザクション, 7-8
- 整合性制約
 - アプリケーションが使用, 4-2
 - 違反, 4-20
 - 違反があれば使用可能にする, 4-20
 - クラスタ, 5-16
 - 削除, 4-24
 - 作成に必要な権限, 4-18
 - 使用可能にする, 4-19
 - 使用禁止にする, 4-19, 4-20, 4-21
 - 使用禁止にする場合, 4-19
 - 使用する場合, 4-2
 - 定義, 4-17
 - 定義のリスト, 4-27
 - トリガーとの対比, 12-2, 12-40
 - ネーミング, 4-18
 - パフォーマンス上の考慮点, 4-3
 - 例, 4-2
 - 例外, 4-22

- 制約
 - コンポジット一意キー, 4-7
 - ストアド・ファンクションの制限事項, 9-57
- 制約表, 12-22
- 西暦 2000 年, 3-10
- セーブポイント
 - 最大数, 7-6
 - ロールバック, 7-6
- セキュリティ
 - Oracle8i の機能, 11-4
 - アプリケーション・コンテキスト, 11-40
 - アプリケーションでの施行, 11-7
 - アプリケーションのポリシー, 11-5, 11-12
 - 脅威および対策, 11-2
 - データベースでの施行, 11-7
 - 表ベースまたはビュー・ベース, 11-35
 - ファイングレイン・アクセス・コントロール, 11-35
 - ロール、利点, 11-8
- セキュリティ・ポリシー
 - 管理, 11-38
 - 技術的問題, 11-3
 - 脅威および対策, 11-2
 - 実装, 11-47
 - 集中的に管理される, 11-14
 - 設定, 11-1, 11-5
 - データベース内で適用, 11-15
 - 表ごとの複数のポリシー, 11-37
 - 表またはビュー, 11-36
 - 例, 11-39
- セッション
 - パッケージ状態, 9-17
- セッション基本形, 11-42

そ

- 相関名, 12-13 ~ 12-17
 - new, 12-16
 - old, 12-16
 - REFERENCING オプション, 12-17
 - 列にコロンが付く場合, 12-16
- ソート
 - ファンクション索引, 5-6
- 属性、USERENV, 11-43

た

第3世代言語, 9-2

タイムスタンプ

PL/SQL ライブラリ・ユニットの依存性, 9-28

対話形式のブロック実行, 9-50

妥当性チェック機能, 11-41

ち

逐次再使用可能 PL/SQL パッケージ, 9-70

中間層システム, 11-42, 11-64, 11-65, 11-66

チューニング

LONG の使用, 3-19

て

データ暗号化規格 (DES), 11-72

データ・オブジェクト番号

拡張 ROWID, 3-21, 3-22

データ型, 3-2

ANSI/ISO, 3-24

CHAR, 3-2, 3-5

DATE, 3-8, 3-10

DB2, 3-24

LONG, 3-18

LONG RAW, 3-18, 3-20

NCHAR, 3-2, 3-5

NUMBER, 3-7

NVARCHAR2, 3-2, 3-5

RAW, 3-20

ROWID, 3-21

SQL/DS, 3-24

VARCHAR, 3-5

VARCHAR2, 3-2, 3-5

データ型の概要, 3-2

データ変換, 3-25

文字型の列の長さ, 3-6

文字データ型の選択, 3-5

データ・ディクショナリ

コンパイル時エラー, 9-42

削除された表との関係, 2-10

スキーマ・オブジェクト・ビュー, 2-41

整合性制約, 4-27

プロシージャ・ソース・コード, 9-43

データの暗号化, 11-4, 11-72

データ・ファイル

ROWID 内に示される, 3-22

データ・ブロック

ROWID 内に示される, 3-22

サイズに影響する要因, 2-6

データベース

アプリケーションおよびセキュリティ, 11-5

アプリケーション管理者, 11-8

セキュリティおよびスキーマ, 11-19

分散システムにおけるグローバル名, 2-38

ユーザーおよびアプリケーション・ユーザー, 11-6

データベース・イベント通知, 13-1, 14-5

データ変換, 3-25

ANSI データ型, 3-24

SQL/DS データ型と DB2 データ型, 3-25

式の評価, 3-27

割当て, 3-25

デバッグ

ストアド・プロシージャ, 9-47

トリガー, 12-30

デフォルト

PCTFREE オプション, 2-4

PCTUSED オプション, 2-6

ストアド・ファンクションのパラメータ, 9-59

セーブポイントの最大数, 7-6

列値, 4-4, 9-57

ロール, 11-24

電子メール

PL/SQL から送信する, 15-1

と

問合せ

一時表でのスピードアップ, 2-11

動的, 8-4

ビューとして獲得, 2-15

分散問合せでのエラー, 9-46

同時実行性, 7-22

動的 SQL, 11-14, 11-37, 11-39

「DBMS_SQL パッケージ」を参照

DML 文, 8-3

PL/SQL ブロックの起動, 8-7

アプリケーション開発言語, 8-24

最適化, 8-6

「システム固有の動的 SQL」を参照

実行者権限, 8-8

使用方法, 8-3

- 使用例, 8-9
- 問合せ, 8-4
- 動的 Web ページ, 15-2
- トランザクション
 - SET TRANSACTION コマンド, 7-8
 - 手動ロック, 7-11
 - シリアル化可能, 7-22
 - 自律型, 7-32 ~ 7-40
 - 読取り専用, 7-8
- トランザクションのロールバック
 - セーブポイントまで, 7-6
- トランザクション・マネージャ, 16-2
- トリガー
 - AFTER, 12-7, 12-16, 12-35, 12-38
 - BEFORE, 12-7, 12-16, 12-48, 12-49
 - CHECK 制約, 12-47, 12-48
 - CREATE TRIGGER ON, 11-22
 - FOR EACH ROW 句, 12-12
 - INSTEAD OF トリガー, 12-7
 - LONG、LONG RAW および LOB データ型の使用, 12-20
 - REFERENCING オプション, 12-17
 - UPDATE の列リスト, 12-6, 12-18
 - WHEN 句, 12-13
 - 移行の問題, 12-29
 - イベント, 11-59, 12-5
 - エラー条件および例外, 12-18
 - 監査, 12-34, 12-36
 - 行, 12-12
 - 行の評価順序, 12-21
 - クライアント・イベント, 13-7
 - 権限, 12-26
 - 削除, 12-30
 - コンパイル済み, 12-28
 - 再コンパイル, 12-29
 - 削除された表との関係, 2-10
 - 作成, 12-3, 12-20, 12-26
 - 作成の前提条件, 12-4
 - 参照整合性, 12-41 ~ 12-45
 - システム・トリガー, 12-4
 - DATABASE に対する, 12-4
 - SCHEMA に対する, 12-4
 - 使用可能, 12-31
 - 使用禁止, 12-31
 - 条件述語, 12-14, 12-17
 - 情報のリスト, 12-32
 - スキャンする順序, 12-21

- ストアド, 12-28
- 制限, 12-13, 12-20
- 整合性制約との対比, 12-2, 12-40
- 設計, 12-2
- 説明, 9-28
- データ・アクセスの制限, 12-48
- デバッグ, 12-30
- 導出列値の生成, 12-49
- トリガーの評価順序, 12-21
- ネーミング, 12-5
- パッケージ変数, 12-21
- 複数の同じ型, 12-21
- プロシージャ, 12-20
- 分散問合せの作成, 9-46
- 変更, 12-30
- 変更表, 12-22
- 本体, 12-14, 12-17, 12-18, 12-20
- 無効な SQL 文, 12-20
- リソース・マネージャ・イベント, 13-6
- リモート依存性, 12-20
- リモート例外, 12-18
- 例, 12-34 ~ 12-50
- 列値のアクセス, 12-15
- レポートされるユーザー名, 12-26
- ログイン, 11-49, 11-52, 11-53

取消し

- 選択された列に対する権限, 11-31
- ロールおよび権限, 11-29

取消し、カーソル, 7-10

取消し側イベント属性, 13-4

に

認証

- n 層システム, 11-69
- One Big Application User によって解決される, 11-6
- 中間層, 11-4, 11-64, 11-65, 11-66, 11-67

ね

ネーム・リゾルバ, 2-38

は

バージョン化、アプリケーション・コンテキスト, 11-51

排他ロック

LOCK TABLE コマンド, 7-16

バイナリ・データ

RAW および LONG RAW, 3-20

バインド変数, 11-48

パスワード

ロール, 11-18, 11-25

パッケージ, 1-38

DBMS_OUTPUT

使用例, 9-3

DEBUG_EXTPROC, 10-48

PL/SQL での, 9-12

削除, 9-12

作成, 9-15

作成に必要な権限, 9-16

実行権限, 9-51

シノニム, 9-56

セッション状態, 9-17

説明箇所, 9-17

逐次再使用可能パッケージ, 9-70

ネーミング, 9-16

プロシージャの作成に必要な権限, 9-10

パッケージ仕様部, 9-12

パッケージ本体, 9-12

ハッシュ・クラスタ

使用方法, 5-19 ~ 5-20

パフォーマンス

クラスタ, 5-15

索引列の順序, 5-4

システム固有の動的 SQL, 8-15

パブリッシュ / サブスクライブ, 14-2 ~ 14-6

パラメータ

デフォルト値, 9-9

ストアド・ファンクション, 9-59

モード, 9-6

パラレル実行サーバー, 11-51

パラレル問合せ、および SYS_CONTEXT, 11-51

バルク・バインド, 9-24

DML 文, 9-25

FOR ループ, 9-27

SELECT 文, 9-26

使用方法, 9-25

反復読み, 7-8, 7-11

ひ

比較演算子

空白埋めデータ, 3-6

日付の比較, 3-9

日付算術, 3-27

ビュー

FOR UPDATE 句, 2-15

ORDER BY 句, 2-15

WITH CHECK OPTION, 2-15

エラーを伴う作成, 2-16

置換え, 2-17

疑似列, 12-8

結合ビュー, 2-21

権限, 2-17

削除, 2-20

削除された表との関係, 2-10

作成, 2-15

式を含む, 12-8

使用, 2-18

使用する場合, 2-15

制限, 2-19

変更可能, 12-8

無効, 2-20

元々変更可能, 12-8

表

PCTFREE の指定, 2-5

PCTUSED の指定, 2-6

PL/SQL での, 9-9

位置, 2-4

ガイドライン, 2-2, 2-4

キー保存, 2-22

切捨て, 2-10

クラスタ化した表のスキーマ, 5-16

索引構成, 6-1 ~ 6-20

削除, 2-10

削除権限, 2-11

作成, 2-3, 2-4

作成権限, 2-8

制約表, 12-22

設計, 2-3

表領域の指定, 2-4

変更, 2-9

変更権限, 2-10

変更表, 12-22

列を長くする, 2-9

標準
ANSI, 7-17
表ベースまたはビュー・ベースのセキュリティ, 11-35

ふ

ファイングレイン・アクセス・コントロール
アプリケーション・コンテキスト, 11-14, 11-18,
11-45, 11-47
概要, 11-4, 11-35
機能, 11-36
パフォーマンス, 11-37
ファンクション
「PL/SQL」を参照
副作用, 9-6, 9-61
付与
システム権限, 11-28, 11-29
ロール, 11-28
プライベート SQL 領域
カーソル, 7-9
プラグマ, 7-32, 7-40
RESTRICT_REFERENCES プラグマ, 9-66
SERIALLY_REUSABLE プラグマ, 9-70, 9-71
プリコンパイラ, 9-51
アプリケーション, 9-4
ストアド・プロシージャおよびパッケージのコー
ル, 9-51
プリコンパイラおよび OCI, 1-37
プロシージャ
外部, 10-2
トリガーによりコール, 12-20
プロパティ
CHARSETFORM, 10-24
CHARSETID, 10-24
INDICATOR, 10-23
分散データベース
参照整合性, 4-14
トリガー, 12-20
リモート・ストアド・プロシージャ, 9-53, 9-54
分散問合せ
エラー処理, 9-46
文トリガー
SQL 文の指定, 12-5
UPDATE 文, 12-6, 12-18
行の評価順序, 12-21
タイミング, 12-7
トリガーの評価順序, 12-21

文の条件コード, 12-17
有効な SQL 文, 12-20

へ

変換ファンクション, 3-25
TO_CHAR 関数, 3-12
TO_CHAR 関数、年および世紀の考慮点, 3-11
TO_DATE 関数, 3-11
変更
記憶域パラメータ, 2-9
表, 2-9
変更可能な結合ビュー
定義, 2-21
変更表, 12-22

ほ

ホスト名, 15-1
本体
トリガー, 12-14 ~ 12-20

み

未処理例外, 9-45

む

無効なビュー, 2-20
無名 PL/SQL ブロック, 11-10
説明, 9-2
トリガーとの比較, 9-28

め

明示的ロック
手動ロック, 7-11
メール
PL/SQL から送信する, 15-1
メモリー
拡張性, 9-70

も

モード
パラメータ, 9-6

ゆ

ユーザー

- PUBLIC グループ, 11-34
- アプリケーション・ロールの制限, 11-12
- エンタープライズ, 11-19
- 削除されたロール, 11-27
- スキーマに依存しない, 11-19
- ロールの使用可能, 11-9

ユーザー定義エラー, 9-43, 9-44

ユーザー名

- スキーマ, 11-19
- トリガー内でレポートされている, 12-26

ユーザー・ロック

- 要求, 7-20

よ

読取り専用トランザクション, 7-8

ら

ライブラリ, 1-38

ライブラリ・ユニット

- リモート依存性, 9-28

ランタイム・エラー処理, 9-43

り

リソース・マネージャ, 16-2

リソース・マネージャ・イベント, 13-6

利点

- OCI, 1-7

リモート依存性, 9-28

- シグネチャ, 9-30

- タイムスタンプまたはシグネチャの指定, 9-35

リモート例外処理, 9-46, 12-18

る

ルーチン

- 外部, 10-2
- サービス, 10-35
- 自律型, 7-32

れ

例外

- アプリケーションへの影響, 9-46
- トリガー実行中, 12-18
- 未処理, 9-45
- 無名ブロック, 9-3
- リモート・プロシージャ, 9-46

例外の発生

- トリガー, 12-18

例外ハンドラ

- PL/SQL での, 9-2

列

- CHECK 制約の制限数, 4-16
- UPDATE トリガーでのリスト, 12-6, 12-18
- デフォルト値, 4-4
- トリガーでの導出列値の生成, 12-49
- トリガー内でのアクセス, 12-15
- 長くする, 2-9

- 複数の外部キー制約, 4-11

列、権限, 11-30, 11-31

ろ

ロール

- ADMIN OPTION, 11-29
- GRANT および REVOKE コマンド, 11-27
- PUBLIC に付与された, 11-34
- SET ROLE 文, 11-27
- WITH GRANT OPTION, 11-31
- アプリケーション, 11-9, 11-12, 11-21
- オペレーティング・システム, 11-27
- 解決される有用性, 11-6
- 管理, 11-8, 11-21
- 削除, 11-27
- 作成, 11-23
- システム権限, 11-28
- 集中管理, 11-23
- 使用可能, 11-9
- 使用可能および使用禁止, 11-16
- 推奨される実行, 11-16
- ツール・ユーザーからの制限, 11-12
- デフォルト, 11-24
- 取消し, 11-29
- パスワード, 11-18, 11-25
- 付与, 11-28
- 保護, 11-23

- ユーザー, 11-9, 11-21
- 利点, 11-8
- ログイン・トリガー, 11-49, 11-52, 11-53
- ロック
 - LOCK TABLE コマンド, 7-12, 7-13
 - UTLLOCKT.SQL スクリプト, 7-21
 - 手動での取得に必要な権限, 7-16
 - 手動 (明示的), 7-11
 - 分散, 7-11
 - ユーザー・ロック, 7-20

わ

- 割当て
 - エクステント, 5-17